CHAPTER 8

# Customizing the PowerShell Environment

PowerShell provides an amazing level of customization and transparency. In this chapter, we will discuss how PowerShell supports configuration using three types of variables: automatic, preference, and environment. Automatic variables are created and used by PowerShell to support its operations and can be read by scripts to get context-specific information. Preference variables control PowerShell's behavior and can be modified to suit our needs. Environment variables are not specific to PowerShell. They are variables used by Windows applications including PowerShell and can be created, read, and set by PowerShell scripts. We'll review how to use aliases to create our own names for PowerShell cmdlets. We will use PowerShell drives to dynamically create short names for provider locations. Finally, we'll discuss how to use all these features to customize PowerShell sessions on startup using a special script called a profile.

## PowerShell Variables

Thus far, we've done a lot with user-defined variables. Three other types of variables PowerShell supports are automatic variables, preference variables, and environment variables. Automatic variables are created by PowerShell and store the state of the PowerShell session. Preference variables are created and assigned default values when a PowerShell session starts. The values of these variables can be changed to suit the user's needs. Environment variables are not PowerShell specific. They are variables used to configure the Windows environment and applications, which include the command shell (CMD.EXE) and PowerShell. Although PowerShell can create and maintain environment variables, they can also be created and changed outside of PowerShell. An interesting feature is that PowerShell has a provider for variables, which means we can navigate variables like we do a system folder. For example, enter the following command at the console prompt:

```
Get-ChildItem Variable:
```

A partial listing of the output follows:

```
Name                           Value
----                           -----
$                              cls
?                              True
^                              cls
args                           {}
ConfirmPreference              High
```

```
ConsoleFileName
DebugPreference                 SilentlyContinue
Error                           {The term 'now' is not recognized as the name of a cmdlet...
ErrorActionPreference           Continue
ErrorView                       NormalView
ExecutionContext                System.Management.Automation.EngineIntrinsics
false                           False
FormatEnumerationLimit          4
HOME                            C:\Users\BryanCafferky
Host                            System.Management.Automation.Internal.Host.InternalHost
input                           System.Collections.ArrayList+ArrayListEnumeratorSimple
MaximumAliasCount               4096
MaximumDriveCount               4096
MaximumErrorCount               256
MaximumFunctionCount            4096
MaximumHistoryCount             4096
MaximumVariableCount            4096
```

As we can see from this list, there are a lot of PowerShell variables, and the `Variable` provider gives us an easy way to access them.

## Automatic Variables

Automatic variables are created, assigned values, and used by PowerShell. We've used some of them already. For example, the `$_` variable references the current object in the pipeline. `$Args` contains an array of undeclared parameters for a script or function. When we discussed error handling, we used `$Error`, which is an automatic variable that contains information about errors that occurred. Let's review some of the other interesting automatic variables.

### $?

`$?` contains `true` if the last statement executed was successful and `false` if it failed. It can be used as a way to check for a failed statement. Consider the code here:

```
Get-Date

If ($?) { 'Succeeded' } Else { 'Failed' }

Gibberish   #  Gibberish to force an error.

If ($?) { 'Succeeded' } Else { 'Failed' }
```

The first line executes the `Get-Date` cmdlet. The next line evaluates `$?`, which returns `true` if the last statement executed successfully. Since it did, 'Succeeded' displays on the console. The third line, `Gibberish`, is not a valid command, so it fails. The line after that will write '`Failed`' to the console after a series of error messages. The output should look similar to the output here:

```
Friday, August 14, 2015 3:07:04 PM
Succeeded
Gibberish: The term ' Gibberish' is not recognized as the name of a cmdlet, function, script
file, or operable program.
```

```
Check the spelling of the name, or if a path was included, verify that the path is correct
and try again.
At line:5 char:1
+ Gibberish   #  Gibberish to force an error.
+ ~~~~
    + CategoryInfo          : ObjectNotFound: (Gibberish:String) [], CommandNotFoundException
    + FullyQualifiedErrorId: CommandNotFoundException
Failed
```

## $Host

The variable `$Host` points to the host application for PowerShell and has a number of methods and properties. Some of the properties can be used to change the behavior of PowerShell or to find out information about the PowerShell environment. Some examples of how we can use this variable follow:

```
$Host.Name          # ISE displays 'Windows PowerShell ISE Host', CLI displays 'ConsoleHost'
$Host.Version       # The version of PowerShell host
$Host.CurrentUICulture # The localization, i.e. language setting.
```

To make accessing `$Host` easier, we can use the cmdlet `Get-Host` as shown here:

```
Get-Host
```

And you should see output similar to the display here:

```
Name             : Windows PowerShell ISE Host
Version          : 4.0
InstanceId       : 626a51e9-e433-4282-ac58-bdb2dc38b820
UI               : System.Management.Automation.Internal.Host.InternalHostUserInterface
CurrentCulture   : en-US
CurrentUICulture : en-US
PrivateData      : Microsoft.PowerShell.Host.ISE.ISEOptions
IsRunspacePushed : False
Runspace         : System.Management.Automation.Runspaces.LocalRunspace
```

`Get-Host` will return a reference to `$Host`, which can be stored in a variable. The variable can then be used to set the properties of `$Host` as shown here:

```
$myhost = Get-Host
$myhost.currentculture.DateTimeFormat # List the localized date and time formats
$myhost.UI.RawUI.WindowTitle = "Bryan's Window"   # Sets the window title of the host
```

In the first line in this code, we set `$myhost` to point to `$host` by using the `Get-Host` cmdlet. The next line displays a list of date and time formats for the machine's localization. Finally, we set the host window's title to "`Bryan's Window`". In the CLI, that will show up as in Figure 8-1, but this works in the ISE as well.

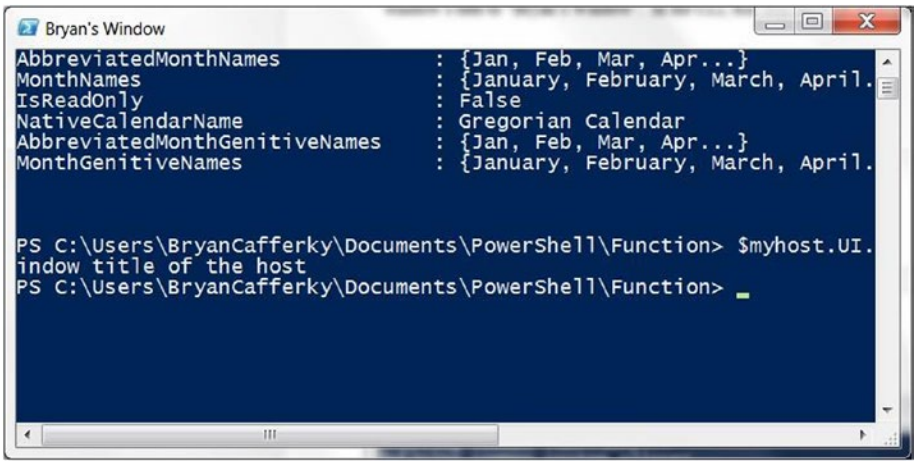


***Figure 8-1.*** *The PowerShell CLI with custom window title*

As we can see in Figure 8-1, we changed the title of the CLI window by changing a property of `$host`.

## $Input

`$Input` is available only within a function or script block. It enumerates the collection in the pipeline. If there is a `Process` block, `$Input` is enumerated through that block, and the `End` block receives an empty collection. If there is no process block, `$Input` is passed to the `End` block once, i.e., as the pipeline collection. `$Input` gives us greater control processing the pipeline than using `$_` does. Let's look at some examples in Listing 8-1 to illustrate.

***Listing 8-1.*** Using $Input

```
function Invoke-UdfProcessPipeline() {
  Begin {
    # Executes once, before the pipeline is loaded, i.e., $input is empty.
    "Begin block pipeline..."
    $Input
  }

  Process {
   # Executes after pipeline is loaded, i.e., enumerates through each item in the collection.
   "Process block pipeline..."
   $Input
  }

  End {
    # Executes once after the Process block has finished. Since there is a Process block,
      $input is emptied.
    "End block pipeline..."
    $Input
  }
}

#  Pipe data into the function Invoke-UdfProcessPipeline
("A", "B", "C") | Invoke-UdfProcessPipeline
```

When we pipe data into this function, the following messages display on the console:

```
Begin block pipeline...
Process block pipeline...
A
Process block pipeline...
B
Process block pipeline...
C
End block pipeline...
```

We see in this example that the `Begin` block has no data in `$Input`. The `Begin` block is used to do work before the pipeline is passed into the function, and it only executes once. The `Process` block is called once for each item in the pipeline, so `$Input` contains one item each time the `Process` block is called. Therefore, we see the "`Process block pipeline`" message displayed multiple times. The `End` block is executed once after the `Process` block has executed. Because there is a `Process` block, `$Input` is exhausted, i.e., has no data, which is why we do not see the `End` process message. What if we don't include a process block? Let's look at that scenario in Listing 8-2.

***Listing 8-2.*** Processing pipeline end block

```
function Invoke-UdfProcessPipelineEndBlock() {
  Begin {
    # Executes before the pipeline is loaded, i.e., $input is empty.
    "Begin block pipeline..."
    $Input
  }

  #  No process block.

  End {
    # Since the Process block
    "End block pipeline..."
    $Input
  }
}

#  Pipe data into the function Invoke-UdfProcessPipeline
("A", "B", "C") | Invoke-UdfProcessPipelineEndBlock
```

When we pipe data into this function, the messages that follow display on the console:

```
Begin block pipeline...
End block pipeline...
A
B
C
```

In Listing 8-2, there is no `Process` block. Therefore, the pipeline is still full when the `End` block is executed, and `$Input` contains the entire pipeline. However, the `End` block is only executed once, so the entire collection in `$Input` sent to the `End` block—i.e., it is not enumerated. This is why we only see "`End block pipeline`" once in the output. What if we want to process the pipeline both in the `Process` block and the `End` block? We can do this by resetting the `$Input` enumerators, as shown in Listing 8-3.

***Listing 8-3.*** Resetting $Input

```
function Invoke-UdfProcessPipelineEndBlockReset() {
  Begin {
    # Executes before the pipeline is loaded, i.e., $input is empty.
    "Begin block pipeline..."
    $Input
  }

  End {
    # Since the Process block
    "End block pipeline..."
    "Iterate through first time to process..."
    $Input | ForEach-Object {$_}
    $Input.Reset()  # Reset to top of the pipeline
    "Show the pipeline has been reset by listing it again..."
    $Input
  }
}

#  Pipe data into the function Invoke-UdfProcessPipeline
"A", "B", "C") | Invoke-UdfProcessPipelineEndBlockReset
```

When we pipe data into this function, the messages that follow display on the console:

```
Begin block pipeline...
End block pipeline...
Iterate through first time to process...
A
B
C
Show the pipeline has been reset by listing it again...
A
B
C
```

Here, we see there is no `Process` block. Instead, we need to completely control the pipeline through the `End` block. Recall that the `End` block is only called once. Therefore, to process each item in the pipeline we need to iterate through `$input`, which is what the statement `$input | Foreach-Object {$_}` does. However, that means the pipeline will be exhausted, so in the next line we reset it with the `Reset` method of `$Input`. To prove we really did reset the pipeline, we display it again. The point of this discussion is to be aware that you can take direct control of the pipeline when needed. For more information on pipeline processing, see the following link:

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

## $MyInvocation

$MyInvocation provides a wealth of information about the content of the currently executing code. To get an idea of the information available, let's try the code in Listing 8-4.

***Listing 8-4.*** Showing $MyInvocation

```
function Invoke-UdfMyFunction ([string]$parm1, [int]$parm2, $parm3 )
{

   $MyInvocation

#  Do some work...
}
```

We call this function with the statement here:

```
Invoke-UdfMyFunction "test" 1 "something"
```

A version of the function Invoke-UdfMyFunction is defined that simply displays the contents of $MyInvocation. We should get output similar to what is shown here:

```
MyCommand             : Invoke-UdfMyFunction
BoundParameters       : {[parm1, test], [parm2, 1], [parm3, something]}
UnboundArguments      : {}
ScriptLineNumber      : 10
OffsetInLine          : 1
HistoryId             : 100
ScriptName            : C:\Users\BryanCafferky\Documents\Book\Code\Chapter08\scr_
                        myinvokation.ps1
Line                  : Invoke-UdfMyFunction "test" 1 "something"
PositionMessage       : At C:\Users\BryanCafferky\Documents\Book\Code\Chapter08\scr_
                        myinvokation.ps1:10 char:1
                        + Invoke-UdfMyFunction "test" 1 "something"
                        + ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
PSScriptRoot          : C:\Users\BryanCafferky\Documents\Book\Code\Chapter08
PSCommandPath         : C:\Users\BryanCafferky\Documents\Book\Code\Chapter08\scr_
                        myinvokation.ps1
InvocationName        : Invoke-UdfMyFunction
PipelineLength        : 1
PipelinePosition      : 1
ExpectingInput        : False
CommandOrigin         : Internal
DisplayScriptPosition :
```

Here, we can see that $MyInvocation has a lot of information. We can see the function name is in the MyCommand property, the parameters are in BoundParameters, the path to the script is in ScriptName, and the full path to the script is in PSCommandPath, and we can see a number of other properties containing other attributes. This information can be used generally when you need to locate files related to the executing script. For example, it could be used to execute or load other scripts in the same folder as the called function. Suppose we wanted to log details about each call to a function and we don't want to hard code the path to the log file. Instead, regardless of where the script with the called function is stored, we will write to a log in the same folder as the called function. Let's look at Listing 8-5 to see this in action.

***Listing 8-5.*** Logging using $MyInvocation.PSScriptRoot

```
function Invoke-UdfMyFunctionWithlogging ([string]$parm1, [int]$parm2, $parm3 )
{

  $logfilepath = $MyInvocation.PSScriptRoot + '\logfile.txt'

  $logline = "function called: " + $MyInvocation.InvocationName + " at " `
             + (Get-Date -Format g) + " with parms: " + $MyInvocation.BoundParameters.Keys

  $logline  >> $logfilepath
  $logline

}

Invoke-UdfMyFunctionWithlogging "test" 1 "something"
```

In Listing 8-5, three parameters are accepted: a string named `$parm1`, an integer named `$parm2`, and an object named `$parm3`. The first executable line stores the full path to the log file as the current script's folder, i.e., `$MyInvocation.PSScriptRoot` concatenated with a backslash and the file name `'logfile.txt'`. Then, we format a string to be written to the log file, which uses `$MyInvocation.InvocationName` to get the function called, `Get-Date`, to get the current date and time, and `$MyInvocation.BoundParameters.Keys` to get the names of the function parameters. Notice that the `Get-Date Format` parameter is used. This is required because without it a carriage return and line feed precede the value—the log row is broken into two rows. Then the log string is appended to the log file pointed to by `$logfilepath`. The last line just displays the variable `$logline`.

The content of the log file should look like the contents here:

```
function called: Invoke-UdfMyFunction_withlogging at 2/8/2015 4:13 PM with parms:
parm1 parm2 parm3
function called: Invoke-UdfMyFunction_withlogging at 2/8/2015 4:13 PM with parms:
parm1 parm2 parm3
```

Note: By default, `Get-Date` prepends the value with a carriage return and line feed. Use the `Format` parameter to eliminate this.

## $null

Database developers in particular should beware the `$null` automatic variable. In T-SQL, a null is an unknown value and will never equal another value, even another null—`null <> null`. In PowerShell, `null` is not like the SQL Server `null`. Instead it represents a default minimum value for a given data type. Let's look at Listing 8-6 to see how `$null` works.

***Listing 8-6.*** Experimenting with $null

```
function Invoke-UdfMyFunctionWithlogging ([string]$parm1, [int]$parm2 = 3, $parm3 )
{

  "The value of `$parm2 is $parm2"

  if ($parm1) {"Has a value"} else {"No value"}
```

```
  if ($parm1 -eq $null) {"No parm passed equals `$null"} else {"No value does not equal
`$null"}

  "Let's get the string version of null and try again."
  if ($parm1 -eq [string]$null) {"`$null test worked"} else {"A real value was passed"}

  "Default for `$parm1 is '$parm1'"
  "Default for `$parm2 is $parm2"
  "Default for `$parm3 is $parm3"

  "Notice that `$null equals `$null"
  ($null -eq $null)

  "If we concatenate a $null with other strings, we do not get `$null back"
  $myvar = "Bryan " + $null + "Cafferky"
  $myvar

  [string]$mystr = $null
  "`$null assigned to a string = '$mystr'."
  [int]$myint = $null
  "`$null assigned to a number = $myint."

}

#  Call the function...
Invoke-UdfMyFunctionWithlogging
```

When we run this code, we should get the following output:

```
The value of $parm2 is 3
No value
No value does not equal $null
Let's get the string version of null and try again.
$null test worked
Default for $parm1 is ''
Default for $parm2 is 3
Default for $parm3 is
Notice that $null equals $null
True
If we concatenate a  with other strings, we do not get $null back
Bryan Cafferky
$null assigned to a string = ''.
$null assigned to a number = 0.
```

The purpose of the function in Listing 8-6 is to show how `$null` and unassigned parameters get values. The function call omits all parameters. The first line of the function displays the value of `$parm2`. Since `$parm2` has a default value assigned, it displays that value, which is 3. The next line just tests for the `$parm1` parameter, and since none was passed, displays the message "`No value`". The next line is interesting, so let's look at it closely:

```
if ($parm1 -eq $null) {"No parm passed equals `$null"} else {"No value does not equal `$null"}
```

We might think that no parameter value is the same as $null, but, as the output confirms, it is not. $null is a type-specific default value. For a string it is a zero-length string, and for a number it is zero. So, if we want to test for a $null, we need to convert it to the data type first by putting the data type just before the variable, as shown in the line here:

```
if ($parm1 -eq [string]$null) {"`$null test worked"} else {"A real value was passed"}
```

We are comparing $parm's default value to $null, cast to a string data type. The default for a string parameter is a zero-length string, and a $null type cast to a string is also a zero-length string. They are equal, so we get the message "$null test worked".

Then we have three lines just meant to show us the parameter default values.

Just to show that $null is not like the database null, we see the lines here:

```
"Notice that `$null equals `$null"
($null -eq $null)
```

The test "$null –eq $null" returns True.

In T-SQL, if you concatenate a null value with other strings, a null is returned. The lines that follow test this in PowerShell:

```
"If we concatenate a $null with other strings, we do not get `$null back"
$myvar = "Bryan " + $null + "Cafferky"
$myvar
```

The $null is just a zero-length string and has no effect. We still get the concatenated value of the other strings.

Finally, just to show the $null has different values depending on the data type, we see the lines shown here:

```
[string]$mystr = $null
"`$null assigned to a string = '$mystr'."
[int]$myint = $null
"`$null assigned to a number = $myint."
```

$mystr is assigned a zero-length string, and $myint is assigned a value of zero.

Let's call the function again, passing $null as the value for all three parameters:

```
Invoke-UdfMyFunctionWithlogging $null $null $null
```

We should see the output shown here:

```
The value of $parm2 is 0
No value
No value does not equal $null
Let's get the string version of null and try again.
$null test worked
Default for $parm1 is ''
Default for $parm2 is 0
Default for $parm3 is
Notice that $null equals $null
True
```

```
If we concatenate a with other strings, we do not get $null back
Bryan Cafferky
$null assigned to a string = ''.
$null assigned to a number = 0.
```

---

Notice that the first line of output shows `$parm2`, which is assigned a default value of 3 in the function, has a value of zero. This is because the `$null` overrode the parameter's default value. The point is that `$null` is a value and is not the same as not passing a value.

---

I hope the point here is clear—`$null` can be useful, but its behavior may not be what you expect.

Caution: `$null` can be misleading as it represents a value, not an unknown as in SQL Server. Be very careful when using `$null` as it may not behave as you would expect.

## $OFS

`$OFS` is the string used by PowerShell to separate values when an array in converted to a string. Depending on your requirements, this could come in handy. For example, if you need to convert arrays to strings to be consumed by an application, you could use `$OFS` to help. Let's review in the example that follows.

First, let's set `$OFS` to its default value of a space, load an array, convert the array to a string, and write the string to the console:

```
$OFS = ' '  # Setting to the original default value.
$myarray ="Bob","Mary","Harry","Sue"
[string] $mystring1 = $myarray
$mystring1
```

The output for this code is shown next. Each element is separated by a space:

```
Bob Mary Harry Sue
```

Now let's set `$OFS` to a comma, load an array, convert the array to a string, and write the string to the console. This makes outputting a CSV file easy:

```
$OFS = ','
$myarray ="Bob","Mary","Harry","Sue"
[string] $mystring2 = $myarray
$mystring2
```

The output for this code is shown next. Each element is separated by a comma:

```
Bob,Mary,Harry,Sue
```

Next, we set `$OFS` to a tab control character, load an array, convert the array to a string, and write the string to the console:

```
$OFS = "`t"
$myarray ="Bob","Mary","Harry","Sue"
[string] $mystring3 = $myarray
$mystring3
```

The output for this code is shown next. Each element is separated by a tab:

```
Bob     Mary     Harry     Sue
```

Next, we set $OFS to a string 'XXXX', load an array, convert the array to a string, and write the string to the console. This example shows how you can use $OFS to do some very custom output:

```
$OFS =  'XXXX'
$myarray ="Bob","Mary","Harry","Sue"
[string] $mystring4 = $myarray
$mystring4
```

The output for this code is shown next. Each element is separated by the string 'XXXX':

```
BobXXXXMaryXXXXHarryXXXXSue
```

What's significant about the $OFS variable beyond how handy it is to be able to define the value separator is that we are not limited to one character. We can use escape characters, strings, and even embed variables and functions into the value. Consider the odd example here:

```
$OFS =  "- field separators like " + (get-date) + " are odd - "
$myarray ="Bob","Mary","Harry","Sue"
[string] $mystring5 = $myarray
$mystring5
```

Which produces the following output:

```
Bob- field separators like 02/07/2015 17:45:49 are odd - Mary- field separators like
02/07/2015 17:45:49 are odd - Harry- field separators like 02/07/2015 17:45:49 are odd – Sue
```

In this example, we are embedding the current date and time by concatenating the Get-Date cmdlet into the $OFS string we assign.

## $PSBoundParameters

The $PSBoundParameters variable is a handy one for testing and debugging. $PSBoundParameters gets loaded with a dictionary object of each parameter name and value when a function is called. From within the function, we can use it to see what was passed in. Consider the code in Listing 8-7.

***Listing 8-7.*** Inspecting $PSBoundParameters

```
function Invoke-UdfMyFunction ([string]$parm1, [int]$parm2, $parm3 )
{

   $PSBoundParameters

#  Do some work...
}

Invoke-UdfMyFunction "test" 1 "something"
```

In Listing 8-7 we can see the definition for the function `Invoke-UdfMyFunction`. The only executable statement displays the value of `$PSBoundParameters`. Then, we call the function seen in the last line; we should see the following output:

```
Key                                                                  Value
---                                                                  -----
parm1                                                                test
parm2                                                                1
parm3                                                                something
```

We can see each parameter name in the `Key` column and its value in the `Value` column. This can be very useful when debugging code.

## $PSCmdlet

The `$PSCmdlet` object is rich with function execution-context information. The function in Listing 8-8 will help us explore this variable.

***Listing 8-8.*** Exploring $PSCmdlet

```
function Invoke-UdfPSCmdlet {
 [CmdletBinding()]
        param (
          )

    Write-Host "Here is a dump of `$PSCmdLet..."
    $PSCmdlet

    Write-Host "Here is a list of the MyInvocaton attributes..."
    $PSCmdlet.MyInvocation

    if($PSCmdlet.ShouldContinue('Click Yes to see GridView of methods and properties...',
    'Show GridView'))
    {
       $PSCmdlet | Get-Member | Out-GridView
    }

}

Invoke-UdfPSCmdlet
```

`$PSCmdlet` is initialized with data when code that includes the `CmdletBinding` attribute is executed, and it is only available within the called code. The function in Listing 8-8 will do three things to help us see what is available in `$PSCmdlet`. First, it writes the object to the console. Then, it writes just the `MyInvocation` property, which is a set of very useful information about the calling context. Finally, the `ShouldContinue` method of `$PSCmdlet` is called to prompt us as to whether we want to continue, end, or suspend the script. This is the same behavior as when we use the `Confirm` common parameter. The output is shown here:

```
Here is a dump of $PSCmdLet...
ParameterSetName     : __AllParameterSets
MyInvocation         : System.Management.Automation.InvocationInfo
PagingParameters     :
```

```
InvokeCommand        : System.Management.Automation.CommandInvocationIntrinsics
Host                 : System.Management.Automation.Internal.Host.InternalHost
SessionState         : System.Management.Automation.SessionState
Events               : System.Management.Automation.PSLocalEventManager
JobRepository        : System.Management.Automation.JobRepository
JobManager           : System.Management.Automation.JobManager
InvokeProvider       : System.Management.Automation.ProviderIntrinsics
Stopping             : False
CommandRuntime       : Invoke-UdfPSCmdlet
CurrentPSTransaction :
CommandOrigin        : Internal

Here is a list of the MyInvocaton attributes...
MyCommand            : Invoke-UdfPSCmdlet
BoundParameters      : {}
UnboundArguments     : {}
ScriptLineNumber     : 20
OffsetInLine         : 1
HistoryId            : 210
ScriptName           : C:\Users\BryanCafferky\Documents\Book\Code\Invoke-UdfPSCmdlet.ps1
Line                 : Invoke-UdfPSCmdlet
PositionMessage      : At C:\Users\BryanCafferky\Documents\Book\Code\Chapter08\Invoke-
                       UdfPSCmdlet.ps1:20 char:1
                       + Invoke-UdfPSCmdlet
                       + ~~~~~~~~~~~~~~~~~~
PSScriptRoot         : C:\Users\BryanCafferky\Documents\Code\Chapter08
PSCommandPath        : C:\Users\BryanCafferky\Documents\Code\Chapter08\Invoke-UdfPSCmdlet.ps1
InvocationName       : Invoke-UdfPSCmdlet
PipelineLength       : 1
PipelinePosition     : 1
ExpectingInput       : False
CommandOrigin        : Internal
DisplayScriptPosition :

True
```

Then you are prompted as shown in Figure 8-2.

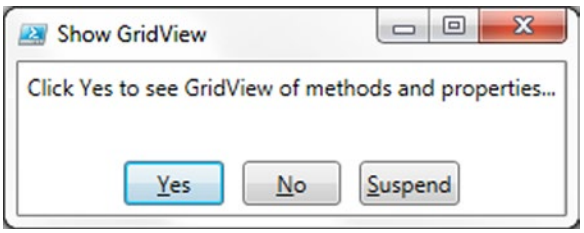


***Figure 8-2.*** *The Confirmation method causes the execution to stop so the user can decide on an action*

If you click Yes, True is returned to the `if` expression and the code in the braces executes—i.e., `$PSCmdlet` is piped into `Get-Member`, which is piped into `Out-Gridview`. If you click No, the program stops. If you click Suspend, the program suspends running and you are able to enter commands into the console until you type `'exit'` to resume program execution.

## $PsCulture

`$PsCulture` contains the culture name, language code, and country code currently in use, such as `'en-US'` for United States English.

```
$PsCulture
```

The value of `$PsCulture` will display as shown here:

```
En-US
```

The value is obtained from `System.Globalization.CultureInfo.CurrentCulture.Name`. We can use the cmdlet `Get-Culture` to get a reference to this object, as shown:

```
$myculture = Get-Culture
```

Try the statements that follow to explore the information returned by `Get-Culture`:

```
$myculture | Get-Member
```

The output of this command shows that there is a lot of information in this object:

```
Name                          MemberType Definition
----                          ---------- ----------
ClearCachedData               Method     void ClearCachedData()
Clone                         Method     System.Object Clone(), System.Object ICloneable...
Equals                        Method     bool Equals(System.Object value)
GetConsoleFallbackUICulture   Method     cultureinfo GetConsoleFallbackUICulture()
GetFormat                     Method     System.Object GetFormat(type formatType),
GetHashCode                   Method     int GetHashCode()
GetType                       Method     type GetType()
ToString                      Method     string ToString()
Calendar                      Property   System.Globalization.Calendar Calendar {get;}
CompareInfo                   Property   System.Globalization.CompareInfo CompareInfo
CultureTypes                  Property   System.Globalization.CultureTypes CultureTypes
DateTimeFormat                Property   System.Globalization.DateTimeFormatInfo
DisplayName                   Property   string DisplayName {get;}
EnglishName                   Property   string EnglishName {get;}
IetfLanguageTag               Property   string IetfLanguageTag {get;}
IsNeutralCulture              Property   bool IsNeutralCulture {get;}
IsReadOnly                    Property   bool IsReadOnly {get;}
KeyboardLayoutId              Property   int KeyboardLayoutId {get;}
LCID                          Property   int LCID {get;}
Name                          Property   string Name {get;}
NativeName                    Property   string NativeName {get;}
NumberFormat                  Property   System.Globalization.NumberFormatInfo NumberFormat
```

```
OptionalCalendars              Property   System.Globalization.Calendar[] OptionalCalendars
Parent                         Property   cultureinfo Parent {get;}
TextInfo                       Property   System.Globalization.TextInfo TextInfo {get;}
ThreeLetterISOLanguageName     Property   string ThreeLetterISOLanguageName {get;}
ThreeLetterWindowsLanguageName Property   string ThreeLetterWindowsLanguageName {get;}
TwoLetterISOLanguageName       Property   string TwoLetterISOLanguageName {get;}
UseUserOverride                Property   bool UseUserOverride {get;}
```

We can get information about the calendar by displaying the `Calendar` property:

```
$myculture.Calendar
```

The contents are written to the console as shown here:

```
MinSupportedDateTime : 1/1/0001 12:00:00 AM
MaxSupportedDateTime : 12/31/9999 11:59:59 PM
AlgorithmType        : SolarCalendar
CalendarType         : Localized
Eras                 : {1}
TwoDigitYearMax      : 2029
IsReadOnly           : False
```

To display information about the culture-specific date and time, use the `DateTimeFormat` property, as shown here:

```
$myculture.DateTimeFormat
```

The following detailed date and time format information is then displayed:

```
AMDesignator                   : AM
Calendar                       : System.Globalization.GregorianCalendar
DateSeparator                  : /
FirstDayOfWeek                 : Sunday
CalendarWeekRule               : FirstDay
FullDateTimePattern            : dddd, MMMM dd, yyyy h:mm:ss tt
LongDatePattern                : dddd, MMMM dd, yyyy
LongTimePattern                : h:mm:ss tt
MonthDayPattern                : MMMM dd
PMDesignator                   : PM
RFC1123Pattern                 : ddd, dd MMM yyyy HH':'mm':'ss 'GMT'
ShortDatePattern               : M/d/yyyy
ShortTimePattern               : h:mm tt
SortableDateTimePattern        : yyyy'-'MM'-'dd'T'HH':'mm':'ss
TimeSeparator                  : :
UniversalSortableDateTimePattern : yyyy'-'MM'-'dd HH':'mm':'ss'Z'
YearMonthPattern               : MMMM, yyyy
AbbreviatedDayNames            : {Sun, Mon, Tue, Wed...}
ShortestDayNames               : {Su, Mo, Tu, We...}
DayNames                       : {Sunday, Monday, Tuesday, Wednesday...}
AbbreviatedMonthNames          : {Jan, Feb, Mar, Apr...}
MonthNames                     : {January, February, March, April...}
```

```
IsReadOnly                      : False
NativeCalendarName              : Gregorian Calendar
AbbreviatedMonthGenitiveNames   : {Jan, Feb, Mar, Apr...}
MonthGenitiveNames              : {January, February, March, April...}
```

We can use the `ThreeLetterISOLanguageName` property to get the ISO standard three-character code for the language:

```
$myculture.ThreeLetterISOLanguageName
```

For my machine, I see the value here:

```
eng
```

This code stands for English.
To get a longer, more descriptive name, we can use the `DisplayName` property as shown here:

```
$myculture.DisplayName
```

This will show the following value on my machine:

```
English (United States)
```

Piping the `$myculture` object to `Get-Member` shows us that it contains a lot of information. The statements after that show information about the calendar, data/time formats, the ISO three-character code, and the descriptive display name.

## Other Automatic Variables

Table 8-1 describes some other potentially useful automatic variables.

***Table 8-1.*** *Useful Automatic Variables*

| Variable | Description |
|---|---|
| `$HOME` | Path to user's home directory. |
| `$LastExitCode` | Contains the Exit code of the last Windows program that ran. |
| `$PSCommandPath` | The path of where the currently executing script is located. |
| `$PSDebugContext` | When debugging, this object provides information about the debugging session. |
| `$PsHome` | The full path to the PowerShell installation directory. |

For more information on automatic variables, see the following link:

https://technet.microsoft.com/en-us/library/hh847768.aspx

# Preference Variables

Some of PowerShell's behavior that can be changed is controlled by preference variables, which are created and assigned values when a PowerShell session starts. There are many preference variables, but let's focus on those that have significant value for development. For a complete list of all the preference variables, see the following link: https://technet.microsoft.com/en-us/library/hh847796.aspx. Let's take a look at `$ConfirmPreference`.

## $ConfirmPreference

The `$ConfirmPreference` variable has a default value of `High`. This means that when a cmdlet is used that PowerShell considers to pose a high risk if performed incorrectly, the user will be asked to confirm they want to continue with the operation before it is actually performed. Consider the code here:

```
"junk" > myjunk.txt

Remove-Item myjunk.txt  # The file is deleted with no prompt.
```

If you run these two lines, a file is created in the first line and deleted in the second line. Let's try this again, first changing the `$ConfirmPreference` variable as shown here:

```
$ConfirmPreference = "Low"

"junk" > myjunk.txt

Remove-Item myjunk.txt  # you are asked to confirm this action before it is executed.
```

In the first line, we set the `$ConfirmPreference` variable to `"Low"`, thus telling PowerShell to prompt us to confirm the execution of any cmdlet with a `"Low"` or higher risk. When the lines that try to either write to the file or delete it execute, a popup dialog appears asking us to confirm the operation.

What if we wanted to override the `$ConfirmPreference` value? We can do that by passing the `Confirm` parameter on the cmdlet call, as shown here:

```
Remove-Item myjunk.txt -Confirm:$false # Confirm default is overridden.
```

The notation `Confirm:$false` passed a false value to the `Confirm` parameter. Normally a switch parameter like `Confirm` is not passed when a false value is desired, but when we need to override a default value for it, we can pass `false` using this notation. For example, in a custom function, we could pass a false value to a switch parameter as shown here:

```
function Invoke-MyTest ([switch]$test)
{
  if ($test) { "Swell" } else { "Doh!" }
}

Invoke-MyTest -test:$false
```

We should see the following output:

```
Doh!
```

The output shows that we successfully passed a value of `false` to the `$test` parameter.

## $DebugPreference

By default, if you want `Write-Debug` output statements to display on the console, you need to pass the `Debug` common cmdlet parameter. The `$DebugPreference` variable affects not only the display of messages but also whether processing should continue. The default value is `SilentlyContinue`, which suppresses messages and continues processing as if they were not there.

---

■ **Note** You must include the `CmdletBinding` attribute in code for it to support the `Write-Debug` and `Write-Verbose` statements.

---

Setting the `$DebugPreference` variable is a convenient way to configure an environment to display or suppress debug messages without changing any code. For example, in the development environment, it might be useful to set this variable to `Continue` so we can see the messages. In production, we could let this variable default to `SilentlyContinue`, which ignores the `Write-Debug` statements. The `$DebugPreference` valid values and related effects are listed in Table 8-2.

***Table 8-2.*** *DebugPreference Valid Values*

| Value | Effect |
|---|---|
| `Stop` | Displays the message, stops execution, and outputs an error message to the console |
| `Inquire` | Displays the message and asks the user if they want to continue. If the cmdlet call overrides the preference variable with the `Debug` common parameter, PowerShell changes the `$DebugPreference` variable to `Inquire` and will ask the user if they want to continue processing. |
| `Continue` | Displays the message and continues execution |
| `SilentlyContinue` | This is the default value. Suppresses messages and continues processing, i.e., ignores the `Write-Debug` or `Write-Verbose` statements, respectively. |

Be aware that if the `$DebugPreference` setting is overridden by a cmdlet call that uses the `Debug` common parameter, the `$DebugPreference` variable is changed to `Inquire` for the duration of the call. This means that the current and all subsequent `Write-Debug` statements will cause processing to halt while the user is prompted to continue, suspend, or stop processing. Let's look at Listing 8-9 to see how this works.

***Listing 8-9.*** Experimenting with $DebugPreference

```
$DebugPreference = "SilentlyContinue"

function Show-UdfMyDebugMessage {
[CmdletBinding()]
        param ()

  Write-Debug "Debug message"

  $DebugPreference

  Write-Debug "2nd Debug message"

}
```

Let's call the function in Listing 8-9 with no parameters:

```
Show-UdfMyDebugMessage
```

The `Write-Debug` statements in the function were ignored and all we get is the value of `$DebugPreference`, which is `SilentlyContinue`.

Now, let's make the call passing the `Debug` common parameter:

```
Show-UdfMyDebugMessage -Debug
```

We should see the following lines written to the console. Not only are the `Write-Debug` messages being written to the console, but we also get prompted on each `Write-Debug` message about whether we want execution to continue, halt, or be suspended. Notice that the value of `$DebugPreference` is `Inquire`. It was set to this because we passed the `Debug` common parameter. See the following:

```
DEBUG: Debug message
Inquire
DEBUG: 2nd Debug message
```

So, did the value of `$DebugPreference` stay `Inquire`? Let's test that by calling the function again without the `Debug` parameter:

```
Show-UdfMyDebugMessage
```

We see the following output on the console, which proves the value has returned to its original value:

```
SilentlyContinue
```

Finally, just to prove `$DebugPreference` is really back to its original value, let's display it as shown here:

```
$DebugPreference
```

The behavior of PowerShell temporarily changing the value of `$DebugPreference` is called out, as it may cause unanticipated behavior.

## $VerbosePreference

By default, if you want `Write-Verbose` output statements to write to the console, you need to pass the `Verbose` common cmdlet parameter. This preference variable affects not only the display of messages, but also whether processing should continue. The default value is `SilentlyContinue`, which suppresses the messages and continues processing as if they were not there. Remember, you must include the `CmdletBinding` attribute in any code that you want to support the `Write-Debug` and `Write-Verbose` statements. Table 8-3 lists the possible `$VerbosePreference` values and the effect of each value.

***Table 8-3.*** *$VerbosePreference Valid Values*

| Value | Effect |
|---|---|
| `Stop` | Displays the message, stops execution, and outputs an error message to the console |
| `Inquire` | Displays the message and asks the user if they want to continue |
| `Continue` | Displays the verbose message and continues processing |
| `SilentlyContinue` | This is the default value. Suppresses messages and continues processing, i.e., ignores the `Write-Verbose` statement |

## Log Event Variables

Some PowerShell events are written to the Windows event log. To see what is logged, enter the command seen here:

```
Get-EventLog "Windows PowerShell"
```

You should see a list of PowerShell events that were logged. There are two Windows event logs, which are the classic and the new event logs. The log PowerShell writes to is the classic event log. Starting with Windows Vista, Windows added a new event log that was redesigned to use XML, and a new viewer was added to support viewing the new format.

The PowerShell events that are logged by default are listed here:

| | |
|---|---|
| `$LogEngineHealthEvent` | Logs PowerShell session errors and failures |
| `$LogEngineLifecycleEvent` | Logs when PowerShell is started and stopped |
| `$LogProviderLifecycleEvent` | Logs the starting and stopping of Providers |
| `$LogProviderHealthEvent` | Logs any Provider errors |

To stop logging any of these events, just set the associated variable to `$false`. The events that follow are not logged by default:

| | |
|---|---|
| `$LogCommandHealthEvent` | Logs any command errors |
| `$LogCommandLifecycleEvent` | Logs when commands start and end |

The log event preference variables have either a true or false value. To turn on logging for an event, set its corresponding preference variable to true, i.e., `$LogCommandHealthEvent = $true` will start the logging of command errors. To turn off logging for an event, set its corresponding preference variable to true, i.e., `$LogEngineHealthEvent = $false` will stop the logging of session errors and failures

The following cmdlets can be used to manage the PowerShell event log. You must be running PowerShell with administrator rights to use these cmdlets.

| | |
|---|---|
| `Clear-EventLog` | Deletes all the entries in the log(s) specified on the machine specified. |
| `Limit-EventLog` | Sets limits on how big the log can grow and how long to retain entries. |
| `New-EventLog` | Creates a custom event log. |
| `Remove-EventLog` | Removes an event log. This does not just clear entries, it actually deletes the log. |
| `Show-EventLog` | Opens up the Classic Event Viewer. |
| `Write-EventLog` | Writes an entry to the specified event log. |

Using these cmdlets, we can create custom logs and write to them in our applications. Let's look at the example in Listing 8-10. Note: To run this script you need to start PowerShell as administrator.

***Listing 8-10.*** Creating a custom log

```
New-EventLog -LogName DataWarehouse -Source ETL

Limit-EventLog -LogName DataWarehouse -MaximumSize 10MB -RetentionDays 10

For ($i=1; $i -le 5; $i++)
{
   Write-EventLog -LogName DataWarehouse -Source ETL -EventId 9999 -Message ("DW Event
   written " + (Get-Date))
}

Get-EventLog -LogName DataWarehouse
```

The first line in Listing 8-10 creates a new event log named `DataWarehouse` with a source of ETL. The next line limits the log to 10 megabytes and a 10-day retention of entries. Then the `For` loop writes some entries to the log. Finally, we view the log entries with the `Get-EventLog` cmdlet. The display should look similar to the output here:

```
Index   Time          EntryType   Source   InstanceID   Message
-----   ----          ---------   ------   ----------   -------
13      Feb 23 17:03  Information ETL      9999         DW Event written 02/23/2015 17:03:27
12      Feb 23 17:03  Information ETL      9999         DW Event written 02/23/2015 17:03:27
11      Feb 23 17:03  Information ETL      9999         DW Event written 02/23/2015 17:03:27
10      Feb 23 17:03  Information ETL      9999         DW Event written 02/23/2015 17:03:27
9       Feb 23 17:03  Information ETL      9999         DW Event written 02/23/2015 17:03:27
```

Let's clear and remove the log with the statements here:

```
Clear-EventLog -LogName DataWarehouse

Remove-EventLog -LogName DataWarehouse
```

The first line deletes all entries from the log, but the log is still available to be written to. The `Remove-EventLog` cmdlet deletes the log itself so it can no longer be written to.

The `EventLog` cmdlets make creating and using custom event logs easy. In production environments this can be a good way to audit applications and log information to help debug problems.

## $PSDefaultParameterValues

`$PSDefaultParameterValues` provides a way to define values to be used for cmdlet parameters when they are omitted from the call. To get an idea of how this works, let's look at the following example:

```
$PSDefaultParameterValues=@{
"Get-EventLog:logname"="Windows PowerShell";
"Get-EventLog:EntryType"="Warning";
}

Get-EventLog
```

`$PSDefaultParameterValues` is a hash table, so it is loaded with the standard hash table assignment we've seen before. It is loaded using a series of name/value pairs. The name is the cmdlet name you want to assign the default value to, followed by a colon and the parameter name, i.e., `Get-ChildItem:Path`. This is assigned the value on the opposite side of the equal sign. In our example, we are assigning a default value for the `LogName` parameter and the `EntryType` parameter of `Get-EventLog`. When we run the statement `Get-EventLog`, we will only see the PowerShell log's Warning messages.

We can use wildcards on the cmdlet name, and we are not limited to built-in cmdlets. We can create default parameter values for our own functions, as shown in Listing 8-11.

***Listing 8-11.*** Using $PSDefaultParameterValues

```
function Invoke-UdfSomething
{
[CmdletBinding()]
        param (
              [string] $someparm1,
              [string] $someparm2
               )

   Write-Host "`$someparm1 is $someparm1"
   Write-Host "`$someparm2 is $someparm2"
}

$PSDefaultParameterValues=@{
"Invoke-Udf*:someparm1"="my paramater value";
}

Invoke-UdfSomething
```

`Invoke-UdfSomething` is defined with the parameters `$someparm1` and `$someparm2`. The function just displays the parameter values. Then we assign a default value for `$someparm1` for any cmdlets that begin with `"Invoke-Udf"`, which limits this to our custom code. Finally, we call `Invoke-UdfSomething`, and we should see the output that follows:

```
$someparm1 is my parmater value
$someparm2 is
```

We can see `$someparm1` has the default value, but `$someparm2` does not. Bear in mind, any other functions with a name that starts with `"Invoke-Udf"` and has a parameter named `$someparm1` will also get the default value when no value is passed. However, we cannot use wildcards for parameter names. PowerShell will let you load the default, but it will generate an error when it sees more than one parameter matching the name. Consider the code here:

```
$PSDefaultParameterValues=@{
"Invoke-Udf*:someparm*"="my parmater value";
}

Invoke-UdfSomething
```

When we run this code, we see the output that follows. PowerShell saw the conflict and did not supply the default to either parameter:

```
WARNING: The following name or alias defined in $PSDefaultParameterValues for this cmdlet
resolves to multiple parameters: somepar
m*. The default has been ignored.
$someparm1 is
$someparm2 is
```

`$PSDefaultParameterValues` offers some interesting possibilities. However, it a good idea to restrict the parameters being defaulted as narrowly as possible to avoid unintended side effects.

## Environment Variables

Windows environment variables are exposed in PowerShell as a file-system drive through the environment provider. We can access them using the `Env:` drive, as shown here:

```
Get-ChildItem Env:
```

This command will list all the environment variables. On my machine, the list of environment variables is as follows:

```
Name                           Value
----                           -----
ALLUSERSPROFILE                C:\ProgramData
AMDAPPSDKROOT                  c:\Program Files (x86)\AMD APP\
APPDATA                        C:\Users\BryanCafferky\AppData\Roaming
COMPUTERNAME                   BRYANCAFFERKYPC
HOMEDRIVE                      C:
HOMEPATH                       \Users\BryanCafferky
LOGONSERVER                    \\BRYANCAFFERKYPC
Path                           C:\Program Files\Common Files\Microsoft Shared\Windows Live;
PATHEXT                        .COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC;.CPL
PSModulePath                   C:\Users\BryanCafferky\Documents\WindowsPowerShell\Modules;
USERDOMAIN                     BRYANCAFFERKYPC
USERNAME                       BryanCafferky
windir                         C:\Windows
```

We can use the environment variables in our scripts to make them more robust and flexible. The code that follows is an example of this:

```
$filepath = "$Env:HOMEDRIVE$env:HOMEPATH\Documents\mylogfile.txt"

"Writing row for $Env:UserName." |  Out-File -FilePath $filepath -Append
```

The first line assigns `$filepath` as the current user's documents folder and sets the file name as `mylogfile.txt`. The next line writes a line to the file that includes the current user's name, which is in `$Env:UserName`. Using this technique, the code can run on any machine and store the file in the appropriate folder for that user.

## Using PS Drives

PowerShell drives are similar to network drive mappings except they are specific to PowerShell in creation and usage. They are handy for creating easy-to-remember short names for locations we frequently refer to, like FTP folders, network drives, and UNC paths. By default, PowerShell drives automatically go away when the PowerShell session ends. What is interesting about PowerShell drives is that unlike Windows drive mappings, PowerShell drives can be used for any provider, which includes things like the Windows registry and the certificates store.

Here is an example of creating and using a PS drive:

```
New-PSDrive -name myps -psprovider FileSystem -root C:\Users\

Get-ChildItem myps:   # Get a list of files where the psdrive points.
```

The first line creates a new PS drive called `myps`. The `psprovider` parameter tells PowerShell what provider is being used, which in this case is `FileSystem`. The `Root` parameter is the path to be used as the root of the drive. The second line lists all the files in the folder the PS drive `myps` points to. Notice the colon suffix to the drive name. This is required, because a provider simulates a file system, which requires the drive notation.

Since the `SQLPS` modules create a provider named `SqlServer`, once we have loaded the module, we can create a drive that maps to a level in the SQL Server hierarchy. Let's look at an example of this:

```
Import-Module SQLPS -DisableNameChecking

New-PsDrive -name at -psprovider SqlServer -root ` SQLSERVER:\SQL\Machine\DEFAULT\Databases\
Adventureworks\Tables

Get-ChildItem aworks:
```

After the SQLPS module has been imported, a PS drive named `aworks` is created that points to the tables in the `AdventureWorks` database on the default instance of SQL Server. Note: You will need to change the path to the instance of where your copy of `AdventureWorks` is located. The second line will list all the `AdventureWorks` tables. PowerShell provides a number of built-in PS drives to make accessing resources easier. Let's enter the command you see here:

```
Get-PSDrive
```

When we enter this command, we should see something like the output here:

```
Name      Used (GB)  Free (GB)  Provider    Root
----      ---------  ---------  --------    ----
Alias                           Alias
at                              SqlServer   SQLSERVER:\SQL\User\D...
C         267.97      648.33    FileSystem  C:\
Cert                            Certificate \
D                               FileSystem   D:\
Env                             Environment
F                               FileSystem   F:\
Function                        Function
HKCU                            Registry     HKEY_CURRENT_USER
HKLM                            Registry     HKEY_LOCAL_MACHINE
```

```
myps                  648.33    FileSystem   C:\Users\BryanCafferky\Documents...
SQLSERVER                       SqlServer    SQLSERVER:\
Variable                        Variable
WSMan                           WSMan
```

These predefined drives provide an easy way to query about the environment. Notice there is a PS drive automatically created for the physical drives on your machine. To get a list of all defined aliases, we can enter the command `Get-ChildItem Alias:`, or to see all the environment variables, we just enter `Get-ChildItem Env:`. We can also see the provider names for the predefined drives, which we can use to create more drives of the same type. Another way to get the provider names is with the `Get-PSProvider` cmdlet, as shown here:

```
Get-PSProvider
```

You should see the provider list as follows. Notice that because we loaded the `SQLPS` module, `SqlServer` shows as a provider:

```
Name                 Capabilities                                    Drives
----                 ------------                                    ------
Alias                ShouldProcess                                   {Alias}
Environment          ShouldProcess                                   {Env}
FileSystem           Filter, ShouldProcess, Credentials              {C, myps, D, F}
Function             ShouldProcess                                   {Function}
Registry             ShouldProcess, Transactions                     {HKLM, HKCU}
Variable             ShouldProcess                                   {Variable}
SqlServer            Credentials                                     {SQLSERVER, at}
Certificate          ShouldProcess                                   {Cert}
WSMan                Credentials                                     {WSMan}
```

If we want to delete a PS drive, we can use the `Remove-PSDrive` cmdlet. Let's assume we want to remove the `myps` drive we created earlier and add it back, but this time mapped to a network drive and persisted like a standard drive mapping, and we also want to make it global, which means all our PowerShell sessions can access it. We can do that with the statements here:

```
Remove-PSDrive myps
New-PSDrive -name myps -psprovider FileSystem -root \\Server\Share -Persist -Scope global
```

The first line removes the PS drive `myps`. The line after that creates it again, but this time pointing to a network share named `\\Server\Share`. The `Persist` parameter tells PowerShell not to remove the drive when we exit PowerShell, and the `global` scope makes the drive available to all our PowerShell sessions. PowerShell will not let you persist a local drive mapping—it must be a remote connection. Be sure to replace `\\Server\Share` with a shared location in your environment.

PS drives are a powerful feature not only for being more productive in PowerShell but also as a means to configure the environment for applications. For example, if a number of scripts need to use an FTP folder, a PS drive could be created that all the scripts use. In the development environment, perhaps it points to a test folder, but in production it would point to the production folder. If an FTP server needed to be renamed, only the path pointed to by the PS drive would need to be changed. This is an important consideration for application deployments.

# Aliases

PowerShell allows us to map commands to other names. For example, we could map the word `showfiles` to the cmdlet `Get-ChildItem`. Then, whenever we want a directory listing, we just enter `showfiles`. This alternate name for a cmdlet is called an *alias*. This is a powerful feature that can greatly help productivity when using PowerShell interactively. In fact, PowerShell has many built-in aliases that map cmdlets to Linux and DOS aliases to make the transition easier. To see a list of all the aliases that are defined in your PowerShell environment, enter the following command:

```
Get-Alias
```

A list of all the defined aliases is displayed. Let's take a look at the alias `dir` by entering the command here:

```
Get-Alias dir
```

We should see the following output:

```
CommandType     Name                                               ModuleName
-----------     ----                                               ----------
Alias           dir -> Get-ChildItem
```

The `Name` column shows that `dir` is mapped to `Get-ChildItem`. This means that `dir` and `Get-ChildItem` will both do the same thing. In addition to allowing us to map PowerShell commands to commands more familiar to us, aliases are used to create abbreviations for commands. For example, `gci` is a built-in alias for `Get-ChildItem`. Table 8-4 shows some of the more commonly used aliases that you should be aware of. Notice that many DOS and Linux commands have aliases provided. Other aliases are abbreviations, and you need to know them as you may run across them in other people's code.

***Table 8-4.*** *Built-in aliases and the related cmdlets*

| Alias | PowerShell Cmdlet |
|---|---|
| % | ForEach-Object |
| ? | Where-Object |
| cat | Get-Content |
| cd | Set-Location |
| cls | Clear-Host |
| copy | Copy-Item |
| cp | Copy-Item |
| del | Remove-Item |
| dir | Get-ChildItem |
| echo | Write-Output |
| erase | Remove-Item |
| foreach | ForEach-Object |
| gc | Get-Content |

(*continued*)

***Table 8-4.*** *(continued)*

| Alias | PowerShell Cmdlet |
|---|---|
| gci | Get-ChildItem |
| ls | Get-ChildItem |
| pwd | Get-Location |
| rd | Remove-Item |
| ren | Rename-Item |
| rm | Remove-Item |
| rmdir | Remove-Item |
| select | Select-Object |
| where | Where-Object |

See the following link by Microsoft for guidelines on creating aliases:

https://msdn.microsoft.com/en-us/library/dd878329%28v=vs.85%29.aspx

While aliases are great for increasing productivity when working interactively in PowerShell, I would not recommend using them in your PowerShell programs. A key objective of good programming is to make your code as easy to understand as possible. Aliases that are abbreviations make code cryptic, while aliases that remap cmdlets to different words make it hard for someone reading your code to know which cmdlets are being called.

## Customizing the Console

You've seen how to change the settings in the ISE. However, we can change settings in the console just as easily. Start the PowerShell CLI, click on the PowerShell icon in the upper left corner of the window, and select Properties, as shown in Figure 8-3.

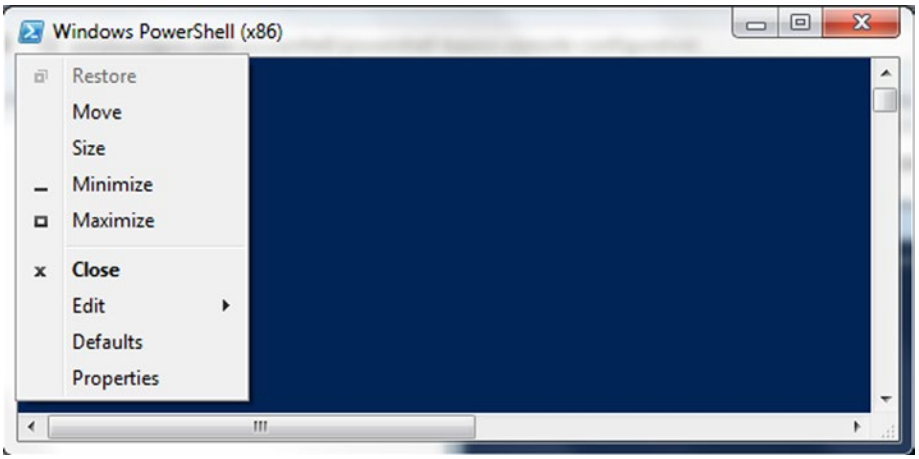


***Figure 8-3.*** *Click on the Properties item in the PowerShell Console drop-down menu to change settings*

The Properties window is displayed; this is where we can customize much of the behavior of the console to fit our needs. On the Options tab, shown in Figure 8-4, we can set the cursor size to small, medium, or large. We can also set the amount of command history to be maintained and whether we want duplicate commands in the history deleted. QuickEdit mode lets you use the mouse to select and copy text in the console. Insert mode adds new text into a line rather than replacing the text.

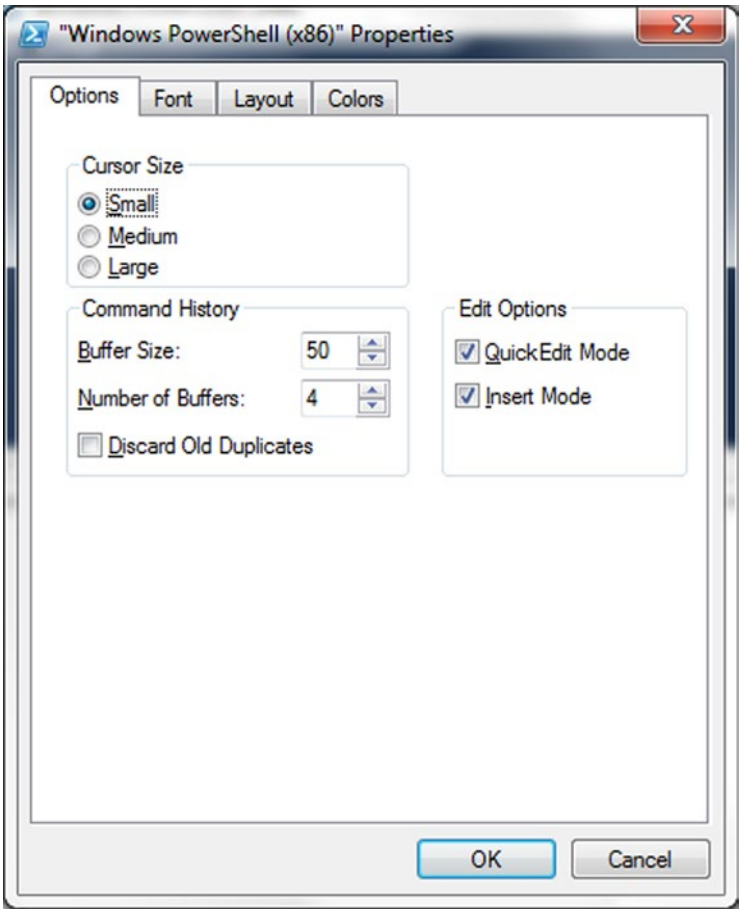


***Figure 8-4.*** *The PowerShell Console Properties window Options tab*

If we click on the Font tab, shown in Figure 8-5, we can select the font and point size of text displayed in the console.

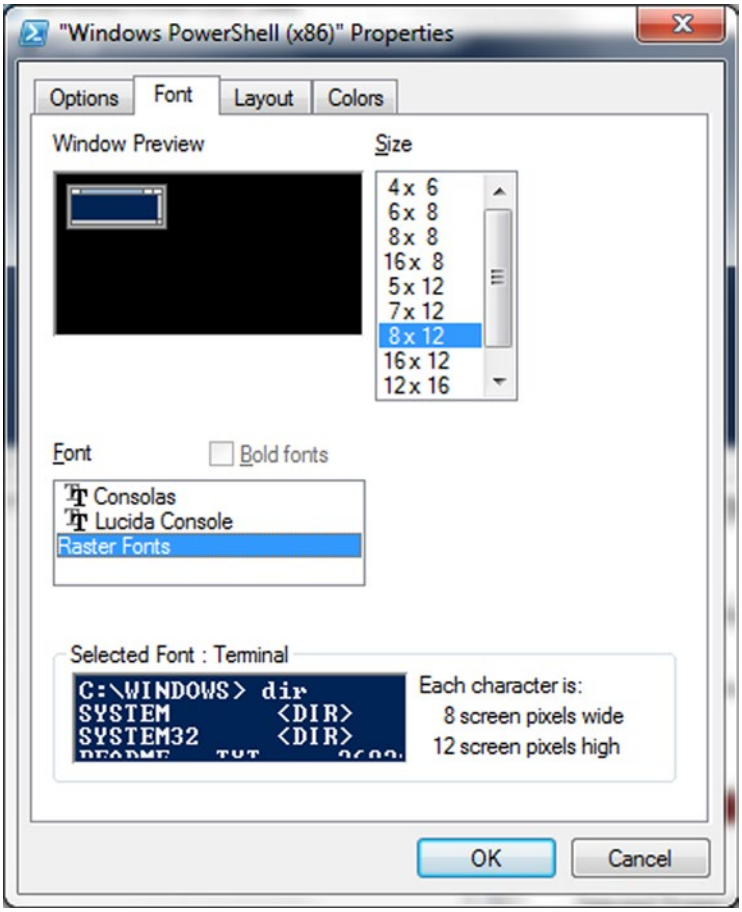


***Figure 8-5.*** *The PowerShell Console Properties window Font tab*

The Layout tab, shown in Figure 8-6, allows us to control the size and placement of the console window. Screen Buffer Size, for example, controls how much text the virtual window can hold, which must always be at least one physical Window Size's worth.

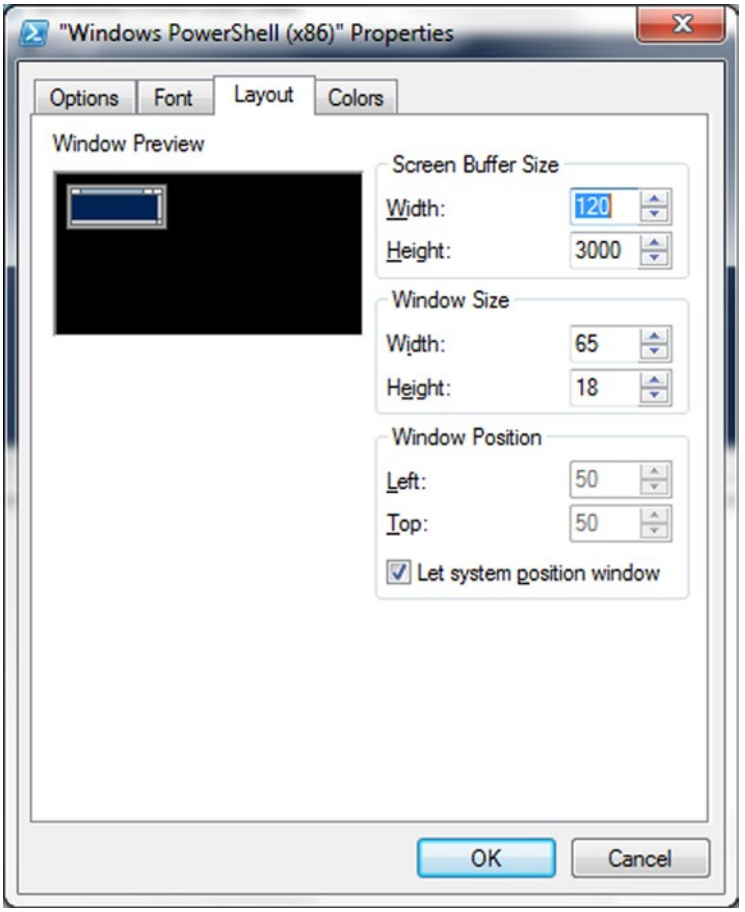


***Figure 8-6.*** *The PowerShell Console Properties window Layout tab*

The Colors tab, shown in Figure 8-7, allows us to set the text and background colors used by the ISE.

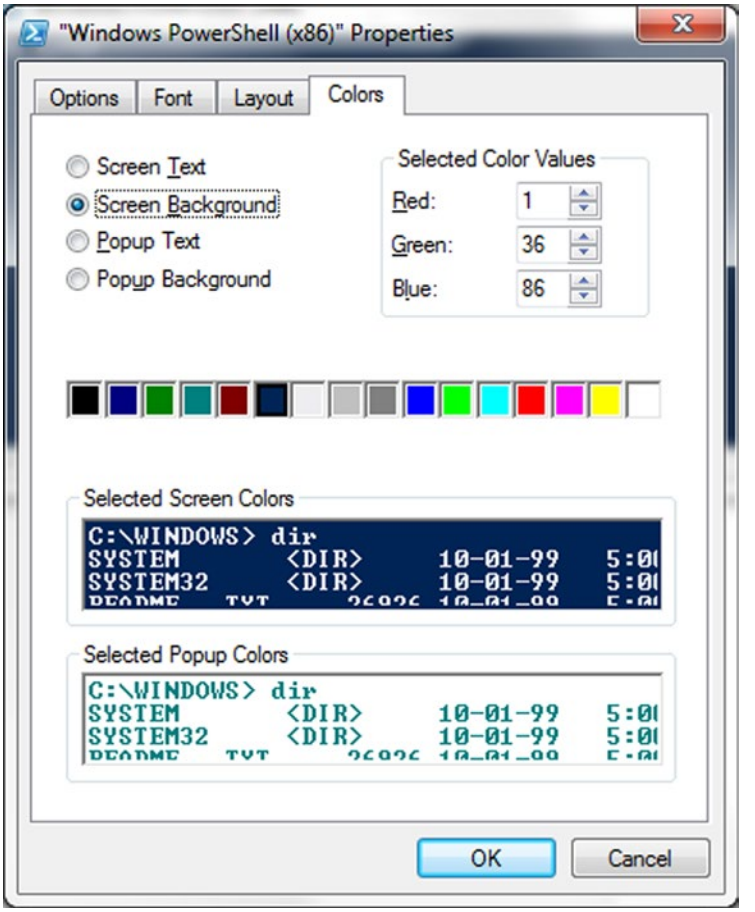


***Figure 8-7.*** *The PowerShell Console Properties window Color tab*

## Profiles

The PowerShell environments support the inclusion of a special startup script called a *profile*. The purpose of this script is to customize the environment for your needs. For example, a database developer may want to load the `SQLPS` module automatically when PowerShell starts. A developer may want to set some default environment values, such as the name of the development database server, so they can be used in scripts and on the command line. In the old DOS days, the equivalent script file was named `Autoexec.bat`. A profile can be defined for the current user or for all users of the computer. A profile can also be created to apply to a specific host like the ISE or the CLI or to be used for all hosts. The variable `$Profile` is an object with a property that stores the path to each type of possible profile script, whether or not the script actually exists. To see the values of these properties enter the following command:

```
$profile | Get-Member -MemberType NoteProperty
```

In the ISE, I get the output here:

```
Name                  MemberType   Definition
----                  ----------   ----------
AllUsersAllHosts      NoteProperty System.String
                                   AllUsersAllHosts=C:\Windows\System32\WindowsPowerShell\
                                   v1.0\profile.ps1
AllUsersCurrentHost   NoteProperty System.String
                                   AllUsersCurrentHost=C:\Windows\System32\
                                   WindowsPowerShell\v1.0\Microsoft.Pow...
CurrentUserAllHosts   NoteProperty System.String
                                   CurrentUserAllHosts=C:\Users\BryanCafferky\Documents\
                                   WindowsPowerShell\profi...
CurrentUserCurrentHost NoteProperty System.String
                                   CurrentUserCurrentHost=C:\Users\BryanCafferky\Documents\
                                   WindowsPowerShell\Mi...
```

If you reference just `$profile`, it references the `CurrentUserCurrentHost` value. Let's try a quick demonstration to see how this works. If the ISE is not already started, start it now. In the ISE console, enter the command seen here:

```
Notepad $profile
```

Assuming you have not created a profile previously, you will get prompted to create one with a dialog box similar to the one in Figure 8-8.

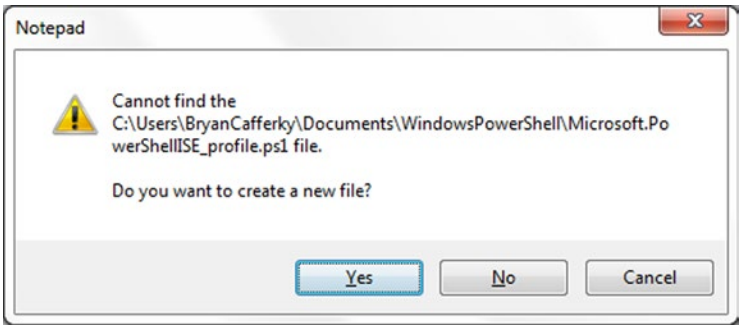


***Figure 8-8.*** *Dialog to create the profile script*

Click `Yes` and you will be presented with an empty document in Notepad. Enter the commands seen here:

```
write-host 'The CurrentUserCurrentHost profile has started...'
write-host "Welcome $env:username." -Foreground Yellow
```

Save the file and exit Notepad. Then exit the ISE and restart it. After the ISE starts, you should see messages displayed to the console similar to the ones here:

```
The CurrentUserCurrentHost profile has started...
Welcome BryanCafferky.
```

The profile script we created will run every time we start the ISE but it will not execute when we enter the PowerShell CLI. If we had created the profile while we were in the CLI, the profile would execute when the CLI was started. This profile is specific to the user who created it. To create a profile that would be executed for all users, we need to start the ISE as an Administrator. Otherwise, Windows will not let us create the file. After starting the ISE as an administrator, we can create a profile that applies to all ISE users by entering:

```
Notepad $profile.AllUsersCurrentHost
```

Then we could enter the script and save it.

An important concept here is that there can be a layering of profiles. If there is a profile of each type, we could have up to four profile scripts execute when we start a host. To demonstrate, I created a short script of each type that just wrote to the console, showing the profile type it was running from. The output is shown here:

```
The AllUsersAllHosts profile has started...
The AllUsersCurrentHost profile has started...
The CurrentUserAllHosts profile has started...
The CurrentUserCurrentHost profile has started...
Welcome BryanCafferky.
```

The point of this is to show that multiple profiles can execute and that they do so in order of broadest scope to most specific scope. If we were to run the CLI, we should see the first three lines display but not the last two, because those are specific to the ISE.

Profiles are generally used to make working interactively with PowerShell easier. For example, we can have modules we use often load automatically or we can customize the console prompt. However, we can also use them to configure the PowerShell environment to execute scripts in a consistent manner. If we have a dedicated Windows account to run the PowerShell scripts, we can customize the current user current host profile to optimize the way scripts are deployed and executed.

## Current User Current Host

As we saw earlier, the narrowest scoped profile is for the current user current host. When we reference `$profile`, the location of this profile is the value returned. Start the PowerShell ISE, and in the console enter the command here:

```
Notepad.exe $profile
```

This command opens the profile for the current user's ISE in Notepad. Note: The 32-bit and 64-bit ISE share the same profile. And the 32-bit and 64-bit CLI share the same profile.

Now, let's enter the following lines into the profile and save it:

```
write-host 'The CurrentUserCurrentHost profile has started...'
Write-host "Welcome $env:username." -Foreground Yellow

$Host.UI.RawUI.WindowTitle = "$env:UserName"
```

---

■ **Note** Remember both the CLI and the ISE provide windows to configure the fonts, window sizes and placement, and other environment features, so a profile may not be needed in some cases.

---

## Current User All Hosts

When we are running PowerShell from our own user account, the current user current host profile, `$Profile.CurrentUserAllHosts`, is our own personal profile, and we are free to do whatever we want in it without affecting others. It is the place to customize PowerShell to suit our tastes and make ourselves more productive. Let's look at the simple profile that follows to customize PowerShell to our needs:

```
write-host 'The CurrentUserAllHosts profile has started...'
# Simple script to establish commonly used PowerShell custom functions...
Import-Module umd_database

New-Alias GetFileName Invoke-UdfCommonDialogSaveFile

$PSDefaultParameterValues=@{
"*:dbservername"="(local)";
}

$global:pscodepath = $env:HomeDrive + $env:HomePath + "\Documents\PowerShell\Scripts"

Set-Location $global:pscodepath
```

In this code, the first line writes a message to the console stating that the `CurrentUserAllHosts` profile has been executed. The module `umd_database` is imported. A new alias `GetFileName` is created for the function `Invoke-UdfCommonDialogSaveFile`. A default parameter value for `dbservername` for all cmdlets is added to `$PSDefaultParameterValues`. Then, a globally scoped variable, `$global:pscodepath`, is created and assigned the concatenation of `$env:HomeDrive, $env:HomePath`, and the string `"\Documents\PowerShell\Scripts"` . The current location is then set to `$global:pscodepath`. Note: To use this alias, the module containing the related function, `umd_database`, must be imported.

## All Users Current Host

To create or change the `$Profile.AllUsersCurrentHost`, we need to start PowerShell as the Administrator. Then execute the following command:

```
Notepad.exe $Profile.AllUsersCurentHost
```

This profile will execute for all users, but only for the host from which we invoked the editor. Following is an example of a customization we might want to make in this profile:

```
Write-Host "The AllUsersCurrentHost profile has started..."
$Host.UI.RawUI.BufferSize.Width = 100
$Host.UI.RawUI.BufferSize.Height = 1000
```

After we display the message that the profile has started, we set the buffer width and size properties to give all users a little extra space. `BufferHeight` refers to how many lines of screen output is retained for scrolling backward.

## All Users All Hosts

There may be some configuration settings that are common to all users and not specific to the host—things such as the mail server and the company name. It may be a security requirement to log the start of all PowerShell sessions with information such as the user ID, date, and time. These are things we can handle in the all users all hosts profile. See here:

```
# Import Modules...
Import-Module umd_database

$global:dbserver = '(local)'
$global:dbname  = 'Adventureworks'

# Some useful aliases
Set-Alias edit   notepad.exe
Set-Alias open   Invoke-Item
Set-Alias now    Get-Date
Set-Alias rsql   Invoke-UdfSQLQuery

$PSDefaultParameterValues = @{"*:Verbose"=$true;
                              "Invoke-UdfSQLQuery:sqlserver"=$global:dbserver;
                              "Invoke-UdfSQLQuery:sqldatabase"=$global:dbname}

"Current User Current Machine Default Parameter Values set are..."
$PSDefaultParameterValues
```

The first non-comment line loads the `umd_database` module so its functions are available to us when we start PowerShell. Then we see two statements assigning values to global variables `dbserver` and `dbname`. Having these available to us means we can pass them into any functions we call. On each machine, such as development, QA, and production, we would set the values as required. Then we see some handy aliases being declared. For example, rather than type "`notepad.exe`", we can just type `"edit"`. The `Invoke-Item` cmdlet is handy to open files of different types because it launches the program registered for the file type. However, it is not intuitively named, so the alias `"open"` is created for it. We often need the current date, so we have a convenient alias of "`now`" for `Get-Date`. The function `Invoke-UdfSQLQuery` will submit a query to a SQL Server database and return the results. For convenience in the CLI, we create the "`rsql`" alias. It would be nice if we could include parameter values for an alias, but that is not supported. Aliases are just a different name for a cmdlet. The use of `$PSDefaultParameterValues` demonstrates how we can set default values based on variables in the profile. We could copy this profile to another machine and change the variables appropriately. Note: The modules you want to load depend on what you use most often, but I have found some modules such as `SQLPS` take a long time to load, and so I do not load it in my profile.

# Summary

In this chapter we saw the many ways PowerShell provides customization and transparency. We discussed how PowerShell supports configuration through automatic, preference, and environment variables. We learned how to use automatic variables to get session and execution context–specific information. Preference variables control PowerShell's behavior and can be modified to suit our needs. Environment variables are used by various applications including PowerShell and can be created, read, and set by PowerShell scripts. We learned how to use aliases to create our own names for PowerShell cmdlets. We saw how PowerShell drives allow us to dynamically create short names for any provider location. Finally, we discussed how to use all these features in a special script called a profile that executes when a session starts.