CHAPTER 5

Writing Reusable Code

This chapter will cover how to use PowerShell to write the most robust reusable code possible. To do this, we will discuss how to use `CmdletBinding` in our functions. There are two types of `CmdletBinding` attributes: general attributes and parameter-specific attributes. The general attributes control how the overall function behaves. The parameter-specific attributes give us control over each parameter, including value validation. When a parameter is not provided, a default value is given. PowerShell defaults may not be what you expect, so we'll cover what they are and how to handle them. We'll also review how to define default values for parameters. Object-orientated languages support function overloading, which allows the same function name to be used with different types of parameters. We'll discuss how to simulate this using parameter sets, and we'll cover how to combine the pipeline with passed parameters to maximize the power and flexibility of your functions. Finally, in this age of globalization, we often need to support multiple languages, so we'll discuss how multiple language support can be implemented easily with special PowerShell features.

CmdletBinding Arguments

The use of the `CmdletBinding` attribute is so powerful that Microsoft documentation calls functions that employ it *advanced functions*. But don't let that scare you. Using the `CmdletBinding` is simple and makes your code much easier to use, test, and maintain. Although there are many attributes available with `CmdletBinding`, we don't have to include any of them to get the benefits of advanced functions. Let's look at a simple example below that uses `CmdletBinding` with no attributes.

```
function Write-UdfMessages
{
  [CmdletBinding()]
  param ()

  Write-Host    "This is a regular message."
  Write-Verbose "This is a verbose message."
  Write-Debug   "This is a debug message."

}

Write-UdfMessages -Verbose
Write-UdfMessages -Debug
```

The function uses minimal code to apply `CmdletBinding`, i.e. the `CmdletBinding` attribute followed by the `param` keyword. Just by adding those two lines, our function will support all the PowerShell common parameters including `Verbose` and `Debug`. When the `Verbose` parameter is passed, our function will display message output by the `Write-Verbose` statement. When the `Debug` parameter is passed, our function will display message output by `Write-Debug`, and then prompt us for the action we want to take. Without the `CmdletBinding` attribute, the `Verbose` and `Debug` parameters will be ignored along with the `Write-Verbose` and `Write-Debug` statements. Run the code. You should see the following output:

```
This is a regular message.
This is a regular message.
VERBOSE: This is a verbose message.
This is a regular message.
DEBUG: This is a debug message.
```

The real power of `CmdletBinding` comes with the numerous attributes it supports that allow us to customize how our functions should behave. Much of the additional functionality is automatically provided by PowerShell. Let's review the `CmdletBinding` attributes and how to use them starting with `SupportsShouldProcess`.

SupportsShouldProcess

The `SupportsShouldProcess` attribute adds support for the `Confirm` parameter which prompts the user for confirmation before performing a potentially risky action like deleting a file. It also adds support for the `WhatIf` parameter that lets you run a command and have it tell you what it will do without actually doing it. Many built-in cmdlets use this attribute, so let's take a look at one to see how this works:

```
"test" > myfile.txt   # Creates a text file.

Remove-Item myfile.txt -Confirm
```

Here we create a file with the line `"test" > myfile.txt`. Then, when we execute the `Remove-Item` cmdlet with the `Confirm` parameter, we are prompted on whether we want to perform the operation or not. Running the command from the ISE, we get the dialog box shown in Figure 5-1.

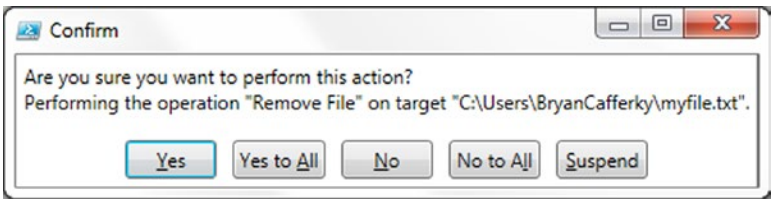

Figure 5-1. *The cmdlet confirmation dialog box*

We can also run the cmdlet, passing the `WhatIf` parameter to see what would happen if we ran the command. For example, the line below will perform the `Remove-Item` cmdlet with the `WhatIf` parameter:

```
Remove-Item myfile.txt -Whatif
```

When we run this command, we get a message like the one here, but the file is not deleted:

```
What if: Performing the operation "Remove File" on target "C:\Users\BryanCafferky\myfile.txt".
```

We can add support for the `Whatif` and `Confirm` parameters to our functions by adding the `SupportsShouldProcess` attribute. Let's look at the function in Listing 5-1 to see how the `SupportsShouldProcess` attribute works.

Listing 5-1. Function using the SupportShouldProcess attribute

```
function Invoke-UdfFileAction {
[CmdletBinding(SupportsShouldProcess=$true)]
        param (
            [Parameter(Mandatory = $true, Position = 0, ParameterSetName = "delete")]
            [switch]$dodelete,
            [Parameter(Mandatory = $true, Position = 1, ParameterSetName = "delete")]
            [string]$filetodelete,
            [Parameter(Mandatory = $true, Position = 0, ParameterSetName = "copy")]
            [switch]$docopy,
            [Parameter(Mandatory = $true, Position = 1, ParameterSetName = "copy")]
            [string]$filefrom,
            [Parameter(Mandatory = $true, Position = 2, ParameterSetName = "copy")]
            [string]$fileto
            )

if ($dodelete)
   {
     try
     {
         Remove-Item $filetodelete -ErrorAction Stop
         write-host "Deleted $filetodelete ."
     }
     catch
     {
         "Error:  Problem deleting old file."
     }
   }

if ($docopy)
   {
     try
     {
         Copy-Item $filefrom $fileto -ErrorAction Stop
         write-host "File $filefrom copied to $fileto."
     }
     catch
     {
         "Error:  Problem copying file."
     }
   }

}
```

Let's create a file to test the function using the following statement:

```
"somestuff" > somedata.txt   # Creates a file.
```

Now let's call the function using the `Whatif` parameter.

```
Invoke-UdfFileAction -dodelete 'somedata.txt' -Whatif
```

This function call would normally delete the text file because we passed the `dodelete` switch. However, because we included the `Whatif` parameter, the file is not deleted. Instead we see these messages:

```
What if: Performing the operation "Remove File" on target "C:\Users\BryanCafferky\somedata.txt".
Deleted somedata.txt .
```

Let's call the function again to copy a file:

```
"somestuff" > copydata.txt
Invoke-UdfFileAction -docopy 'copydata.txt' 'newcopy.txt' -Whatif
```

This code calls the function passing the `-docopy` switch, but since we passed the `-whatif` parameter, we see messages telling us what would be copied, with no copy operation actually being performed.

We will get the output seen here:

```
What if: Performing the operation "Copy File" on target "Item: C:\Users\BryanCafferky\
copydata.txt Destination: C:\Users\BryanCafferky\newcopy.txt".
File copydata.txt copied to newcopy.txt.
```

We can also use the `Confirm` parameter to be prompted before the action is taken. Since ETL processes are usually performed in batch, be careful to remove these parameters after testing.

HelpURI

The `HelpURI` attribute lets you specify a web link where help information can be found. The function in Listing 5-2 shows how it is used.

Listing 5-2. Usingthe HelpURI attribute

```
function Invoke-UdfHelp
{
[CmdletBinding(HelpURI="http://www.microsoft.com/en-us/default.aspx")]
        param (
            )

  Write-Host "Just to show the HelpURI attribute."
}
```

If we request help with the online switch parameter as shown below, the page specified in the `HelpURI` attribute is displayed in the default browser:

```
Get-Help -Online Invoke-UdfHelp
```

This help link is just to demonstrate how to use the `HelpURI` attribute. It is Microsoft's website.

SupportsPaging

The `SupportsPaging` attribute is used by built-in cmdlets to add some paging support. For example, the `Get-Content` command leverages this attribute to allow us to get portions of a file. Let's look at some examples.

This command will display the last ten lines of a file:

```
Get-Content somedata.txt  -Tail 10
```

This command displays the first ten lines of a file:

```
Get-Content somedata.txt  -Head 10
```

For a large file, we can speed up the display of data by using `ReadCount` to tell PowerShell to break up the pipeline into blocks of rows indicated by the `ReadCount` parameter. Let's look at an example:

```
Get-Content somedata.txt -ReadCount 1000
```

This statement will stream the file through the pipeline into blocks of 1,000 rows.

As an ETL developer, the obvious use in custom functions for the `SupportsPaging` attribute would be to control the reading of files. However, as just seen, the `Get-Content` cmdlet already provides a great deal of flexibility.

Validating Parameters

There are a number of parameter attributes that support validating that the parameter values meet criteria you specify. To understand what they are and how to use them, we'll discuss validation by data type.

Validating a String

There are a number of validation attributes to ensure that string parameters comply with our requirements. It is common in flat files to have strings storing dollar amounts that must be loaded. The function in Listing 5-3 requires a value be passed with a minimum length of 5 and a maximum length of 15 and that it consists of numbers followed by a decimal point and two digits, i.e., is a monetary amount.

Listing 5-3. Validating monetary amounts

```
function Invoke-UdfTestValidationMoney
{
 [CmdletBinding()]
        param (
        [parameter(mandatory=$true,
                   HelpMessage="Enter a number with two decimal places.")]
                   [ValidateLength(5,15)]
                   [ValidatePattern("^-?\d*\.\d{2}$")]
              [string] $inparm1
          )

   Write-Verbose $inparm1
}
```

In this code, the Mandatory attribute will force the caller to pass a value. ValidateLength requires a minimum length of 5 and a maximum of 15. Note: This includes the decimal point. The ValidatePattern uses a regular expression to make sure the value is a number with two decimal places.

Let's look at some calls to this function:

```
Invoke-UdfTestValidationMoney "123.22" -Verbose  # Good value.

Invoke-UdfTestValidationMoney "1.1" –Verbose  # Not enough digits after the decimal.

Invoke-UdfTestValidationMoney "12345678912345.25" -Verbose # String is too long.
```

The first call is good and the number passes validation. The second fails the format validation because it has only one digit after the decimal. The third call fails because it is passing a number that is greater than 15 characters in length.

Another option for validating strings is to restrict the accepted values to a list using the ValidationSet attribute, as shown in Listing 5-4.

Listing 5-4. Validating against a set

```
function Invoke-UdfTestValidationNEState
{
 [CmdletBinding()]
        param (
        [parameter(mandatory=$true,
                   HelpMessage="Enter a New England state code .")]
                   [ValidateSet("ME","NH","VT", "MA", "RI", "CT")]
              [string] $newenglandstates
          )

   write-verbose $newenglandstates
}
```

The ValidateSet attribute allows us to restrict the allowed values to the list we specify. In this case, we only want New England state codes.

Here are some examples of calling this function:

```
Invoke-UdfTestValidationNEState "RI" -Verbose   # Passes - RI is in the validation list.

Invoke-UdfTestValidationNEState "NY" -Verbose   # Fails - NY is not in the validation list
```

The first call succeeds because "RI" is in the validation set but the second call fails because "NY" is not in the validation set.

Validating a Number

To validate an integer, we can use the ValidateRange and ValidateSet. Let's look at the code in Listing 5-5.

Listing 5-5. Validating an integer

```
function Invoke-UdfTestValidationInteger
{
 [CmdletBinding()]
        param (
        [parameter(mandatory=$true,
                   HelpMessage="Enter a number between 10 and 999.")]
                   [ValidateRange(10,999)]
                   [ValidateSet(11,12,13, 14, 15)]
              [int] $inparm1
          )

   write-verbose $inparm1
}
```

The `ValidateRange` restricts the value to be between 10 and 999 inclusive. The `ValidateSet` attribute limits the values to those specified in the parentheses, i.e., 11, 12, 13, 14, or 15. Note that even though a value may pass the `ValidateRange` attribute, it may fail the `ValidateSet` attribute. Validating decimal numbers is similar to validating integers, except you can include decimal numbers in the `ValidateRange` and `ValidationSet` attributes.

Here are some examples of calling this function with various values:

```
Invoke-UdfTestValidationInteger 12 -Verbose     # Good value.

Invoke-UdfTestValidationInteger 17 -Verbose     # Rejected - Within the range but not in the set.

Invoke-UdfTestValidationInteger 9999 -Verbose   # Value too large.

Invoke-UdfTestValidationInteger 11.25 -Verbose  # Accepts the value but truncates it.
```

The first call is successful. The second is rejected because 17 is not in the validation set. The third call passes a value that exceeds the range. The fourth call shows that you need to be careful. You might think that a decimal value would be rejected since it is not an integer, but instead the decimal is just truncated without warning.

Validating an Array

You can still use all the validation attributes we've seen so far to validate individual elements of the array, but we have another useful attribute, `ValidateCount`, that sets the minimum and maximum number of array elements the array may have. In the example in Listing 5-6, the function has a parameter that is an array of Social Security numbers.

Listing 5-6. Validating an array

```
function Invoke-UdfTestValidationArray
{
 [CmdletBinding()]
        param (
        [parameter(mandatory=$true,
                   HelpMessage="Enter a string value.")]
                   [ValidateLength(9,11)]
```

```
                [ValidatePattern("^\d{3}-?\d{2}-?\d{4}$")]  # Pattern for SSN
                [ValidateCount(1,3)]
            [string[]] $inparm1
        )

   foreach ($item in $inparm1) {
   write-verbose $item
   }
}
```

Notice that we use the attribute `ValidateCount`, which takes the minimum number of elements as the first parameter and the maximum number of elements as the second parameter. If zero elements are passed or more than three elements are passed, the value will fail validation. Notice also that we still have `ValidateLength` and `ValidatePattern` attributes. These are applied to individual array elements. The following code will call this function with various values:

```
Write-Host "Good parameters..."
Invoke-UdfTestValidationArray @("123445678","999-12-1234") -Verbose  # Passes validation
write-host ""

Write-host "Bad parameters, Invalid format..."
Invoke-UdfTestValidationArray @("1234456789","999-12-123488") -Verbose # Fails validation -
first element does not match pattern.
write-host ""

Write-host "Bad parameters, Too many elements in the array..."
Invoke-UdfTestValidationArray @("123445678","999-12-1234","123-22-8888","999121234")
-Verbose
```

The first call above passes validation. It has the correct number of elements, each element is an acceptable length, and each element matches the validation pattern for a Social Security number. The second and third calls fail.

Using the ValidateScript Attribute

One of the most powerful validation attributes is `ValidateScript` as it allows us to call custom code to perform the validation. The final result returned must be a Boolean value, i.e., `$true` or `$false`. To give you an idea of just how useful this can be, let's consider an ETL-orientated application. We need to load a file, and as part of the process we want to validate the function parameters being passed to our load function. The two function parameters are a folder path and a state code. We want to confirm the path is valid—i.e., actually exists—and we want to make sure the state code is validated against a SQL Server table of state codes. To present the ideas clearly, the code that follows is focused on the validation without actually loading anything.

Before we validate the state code, we'll need to load them into an array. Technically, we could just validate each code by running a `select` query against SQL Server, but that would not be scalable. The goal here is to show you a practical example. Let's look at the code in Listing 5-7, which loads the state codes into an array.

Listing 5-7. Getting values from a table

```
#  Load the state code array
function Get-udfStatesFromDatabase
{
 [CmdletBinding()]
        param (
          )

$conn = new-object System.Data.SqlClient.SqlConnection("Data Source=(local);Integrated
Security=SSPI;Initial Catalog=AdventureWorks");
$conn.Open()

$command = $conn.CreateCommand()

$command.CommandText = "SELECT cast(StateProvinceCode as varchar(2)) as StateProvinceCode
FROM [AdventureWorks].[Person].[StateProvince] where CountryRegionCode = 'US';"

$SqlAdapter = New-Object System.Data.SqlClient.SqlDataAdapter
$SqlAdapter.SelectCommand = $command
$DataSet = New-Object System.Data.DataSet
$SqlAdapter.Fill($DataSet)

Return $DataSet.Tables[0]

$conn.Close()
}
```

Notice that the function in Listing 5-7 does not take any parameters. Rather, we want to load an array of state codes and pass it back to the calling code. The first thing we'll need to do is open a SQL Server connection, which the code that follows does:

```
$conn = new-object System.Data.SqlClient.SqlConnection("Data Source=(local);Integrated
Security=SSPI;Initial Catalog=Development");
$conn.Open()
```

Notice: We are not using the SQLPS module. We're going directly to the SQL Client API. This saves us the overhead of loading the SQLPS module. The first line creates a connection object pointing to a database named `AdventureWorks` on the local server instance. Then we just call the `Open` method of the connection object.

Next, we need to define the SQL command to be executed, which this code does:

```
$command = $conn.CreateCommand()

$command.CommandText = "SELECT cast(StateProvinceCode as varchar(2)) as StateProvinceCode
FROM [AdventureWorks].[Person].[StateProvince] where CountryRegionCode = 'US';"
```

The first line creates a SQL command object. The second line assigns the SQL statement to be executed but does not run it.

Commands that do not return data are simpler to code for, but as we *do* need to return data, we will use the following code:

```
$SqlAdapter = New-Object System.Data.SqlClient.SqlDataAdapter
$SqlAdapter.SelectCommand = $command
$DataSet = New-Object System.Data.DataSet
$SqlAdapter.Fill($DataSet)
```

The first line creates a `SQLDataAdapter` object. Then we assign the command object to the SQL Data Adapter's `SelectCommand` property. To hold the retrieved data, we create a data-set object with the statement `$DataSet = New-Object System.Data.DataSet`. Finally, we load the data set, `$DataSet`, with the query results, i.e., `$SqlAdapter.Fill($DataSet)`.

Now that the results are in `$DataSet`, we can pass it back to the calling code with this statement:

```
Return $DataSet.Tables[0]
```

`DataSet` objects can hold multiple result sets, so the results are stored in a collection called Tables. Since we only ran a single `select` statement, we just need the first item in that collection, i.e., subscript 0. Finally, we close the connection object with the statement `$conn.Close()`.

The code to call the above function is shown below.

```
$global:statelistglobal = Get-udfStatesFromDatabase #  Loads the state codes into variable
$global:statelistglobal
```

We need to have the state code array available to the validation script in our function shown in Listing 5-8. We could create a new state table array each time we call the function with the validate script, but that is very inefficient. Instead, we create a globally accessible array variable before we call the function with the validation script. Prefixing the variable name with `$global:` gives it a global scope, which means it is accessible to the entire PowerShell session. Note: It is not available to other PowerShell sessions.

The function that will validate the state code is shown in Listing 5-8.

Listing 5-8. Validating the state code

```
function Invoke-UdfValidateStateCode ([string] $statecode)
{
   $result = $false

   foreach ($states in $global:statelistglobal)
   {
       if ($states.StateProvinceCode -eq  $statecode)
       {
            $result = $true  }
       }

   Return $result
 }
```

The code in Listing 5-7 must be executed prior to running the code in Listing 5-8. This is so the variable `$global:statelistglobal` is available. The function `Invoke-UdfValidateStateCode` takes only the state code as a parameter. We'll use a Boolean variable to hold the result, i.e. `$true` if the state code is in the list and `$false` if it is not. We initialize the `$result` to `$false` and then use `foreach` to loop through the `StateProvinceCode` values to see if any match the `$statecode` value passed into the function. If a match is found, `$result` is set to `$true`. If all items in the global array `$global:statelistglobal` are compared and no match is found, `$result` remains `$false`. The last line just returns `$result`.

The previous functions in Listings 5-7 and 5-8 laid the groundwork for our validation function, which is presented in Listing 5-9. You must run the other listings before running Listing 5-9.

Listing 5-9. Testing the validations

```
function Invoke-UdfTestValidation
{
 [CmdletBinding()]
        param (
        [         parameter(mandatory=$true, HelpMessage="Enter a folder path .")]
                  [ValidateScript({ Test-Path $_ -PathType Container })]
                  [string] $folder,
                  [parameter(mandatory=$true, HelpMessage="Enter a New England state code.")]
                  [ValidateScript({ Invoke-UdfValidateStateCode $_ })]
                  [string] $statecode

          )
   write-verbose $statecode
}
```

This function accepts two parameters: `$folder`, which is the path to a folder where a file resides, and `$statecode`, which is a two-character state code. Let's look at the `ValidateScript` attribute for the `$folder` parameter:

```
[ValidateScript({ Test-Path $_ -PathType Container })]
```

The script enclosed in the braces within the parentheses will execute with every function call. In this case, we have an in-stream code block that executes. `$_` references the value being passed to the `$folder` parameter. We can use this to test its value. The `Test-Path` statement is verifying that the path actually exists.

As powerful as an in-stream script is, the real power comes when we call an external function, which is what we do to validate the state code. Let's look at the `ValidateScript` attribute for `$statecode`:

```
[ValidateScript({ Invoke-UdfValidateStateCode $_ })]
```

The function `Invoke-UdfValidateStateCode` is the function defined previously in the script. The `ValidateScript` is calling that function and passing the parameter `$statecode` to it via the `$_` variable. If the state code is not in the state code array, the parameter will be rejected.

Listing 5-10 provides some examples of how to call our function. The comment on each line explains the expected output.

Listing 5-10. Running the validations. Note: You must run Listings 5-7, 5-8, and 5-9 prior to this to load the functions

```
$global:statelistglobal = Get-udfStatesFromDatabase # To load the global statelist

$rootfolder = $env:HOMEDRIVE + $env:HOMEPATH + "\Documents\"

Invoke-UdfTestValidation $rootfolder 'MA'  # Passes - good path and state code

Invoke-UdfTestValidation $rootfolder 'XX'  # Fails - Invalid state code

Invoke-UdfTestValidation "c:\xyzdir" 'RI'  # Fails - Invalid path
```

Default Parameter Values

What happens if An optional parameter is not passed to a function? What value will the parameter get assigned? The answer is that it depends. The function can specify a default value for the parameter. Default parameter values can also be set in the `$PSDefaultParameterValues` preference variable. If no default value is defined, a default value indicating no value is assigned.

Specifying a Default Parameter Value

As a best practice, it is good to provide default parameter values when possible. Fortunately, setting a default value for a parameter is easy. Just add the equal sign and the value after the parameter name, i.e., `$myparm = 'somevalue'`. The function in Listing 5-11 has defaults set for the most common parameter data types.

Listing 5-11. Setting default parameter values

```
function Invoke-UdfTestDefaultParameters
{
 [CmdletBinding()]
        param (
          [string]    $p_string = 'DefaultString',
          [int]       $p_int    = 99,
          [decimal]   $p_dec    = 999.99,
          [datetime]  $p_date   = (Get-Date),
          [bool]      $p_bool   = $True,
          [array]     $p_array  = @(1,2,3),
          [hashtable] $p_hash   = @{'US' = 'United States'; 'UK' = 'United Kingdom'}
          )
  "String is : $p_string"
  "Int is: $p_int"
  "Decimal is: $p_dec"
  "DateTime is  $p_date"
  "Boolean is: $p_bool"
  "Array is: $p_array"
  "Hashtable is: "
  $p_hash
}
```

Let's call the function with the following statement:

```
Invoke-UdfTestDefaultParameters
```

We should see output similar to these lines:

```
String is : DefaultString
Int is: 99
Decimal is: 999.99
DateTime is  12/04/2014 13:40:55
Boolean is: True
Array is: 1 2 3
Hashtable is:

Name                           Value
----                           -----
US                             United States
UK                             United Kingdom
```

Using $PSDefaultParameterValues

Suppose we have a parameter such as a server name that will always be the same value. We want a way to set this parameter with a default value so it does not need to be specified on every function call. This is where `$PSDefaultParameterValues` comes in. The PowerShell built-in variable `$PSDefaultParameterValues` is a hash table of key/value pairs that define default values to be used when a parameter that matches the key value is missing from a function call. This is useful as it can provide a good way to support multiple configurations. For example, we can specify one set of values for the Development environment, another for QA, and another for Production. Also, by setting the value as a default it only has to be maintained in one place. For example, if the development SQL Server name changes, we just have to change the name in the default setting.

Since `$PSDefaultParameterValues` is a hash table, we can make changes to it using standard hash table commands. The format of adding a new default to this variable is shown here:

```
$PSDefaultParameterValues.Add("cmdlet:Parameter Name","Parameter Value")
```

The `Add` method will insert an entry into `$PSDefaultParameterValues`. The first parameter is the key, and in the examples we just looked at, we want all cmdlets and functions to use these defaults so the first part is a wildcard, followed by a colon and the parameter name. The second parameter is the default value.

If we only want the default to apply to a specific cmdlet, we can specify the cmdlet as the first part of the key parameter as shown here:

```
$PSDefaultParameterValues.Add("Send-MailMessage:SmtpServer","smtp.live.com")
```

We could use a partial wildcard as show here:

```
$PSDefaultParameterValues.Add("Send*:SmtpServer","smtp.live.com")
```

This statement sets the parameter default only for cmdlets that start with "`Send`".

Code Using $PSDefaultParameterValues

In the previous examples, we wanted the parameter specified to use the default value for all cmdlets and functions. Each environment could have a different set of values. To see how this might be used in practice, let's look at the function in Listing 5-12, which queries a SQL Server database.

Listing 5-12. Running a SQL Query

```
$PSDefaultParameterValues.Add("*:sqlserver","(local)")
$PSDefaultParameterValues.Add("*:sqldatabase","AdventureWorks")

function Invoke-UdfSQLQuery
{
 [CmdletBinding()]
        param (
              [string] $sqlserver       ,   # SQL Server
              [string] $sqldatabase     ,   # SQL Server Database.
              [string] $sqlquery            # SQL Query
          )

  $conn = new-object System.Data.SqlClient.SqlConnection("Data Source=$sqlserver;Integrated
  Security=SSPI;Initial Catalog=$sqldatabase");

  $command = new-object system.data.sqlclient.sqlcommand($sqlquery,$conn)

  $conn.Open()

  $adapter = New-Object System.Data.sqlclient.sqlDataAdapter $command
  $dataset = New-Object System.Data.DataSet
  $adapter.Fill($dataset) | Out-Null

  RETURN $dataset.tables[0]

  $conn.Close()
}
```

Notice that before we call the function, we add default values for the parameters `sqlserver` and `sqldatabase`. Then to run a query, we just need to call the following function:

```
Invoke-UdfSQLQuery   -sqlquery 'select top 100 * from person.address'
```

Assuming `$PSDefaultParameterValues` has been loaded with the default parameter values, this call will work in any environment. Ideally we would want the default parameter values to load automatically so that there is no risk that they will not be set. This can be accomplished by adding the code that loads `$PSDefaultParameterValues` into the user's profile script. We'll cover how to do that in Chapter 8, "Customizing the PowerShell Environment."

To remove items from `$PSDefaultParameterValues`, just use the `Remove` method, as shown here:

```
$PSDefaultParameterValues.Remove("*:sqlserver")
```

A nice feature of $PSDefaultParameterValues is that we can disable the entire set of defaults without removing them. We can do this by adding a special key of Disabled as shown here:

```
$PSDefaultParameterValues.Add("Disabled", $true)
```

To reactivate parameter defaults, we can either remove it as follows:

```
$PSDefaultParameterValues.Remove("Disabled")
```

Or we can just set the value of Disabled to $false as shown here:

```
$PSDefaultParameterValues[Disabled]=$false
```

For more information on using $PSDefaultParameterValues, see this link:

http://technet.microsoft.com/en-us/library/hh847819.aspx

Default Parameter Values When No Default Is Specified

It is tempting to assume that if a parameter is passed to a function with no default value specified, the default value assigned will be NULL. However, this is not the case with PowerShell. The value assigned depends on the data type of the parameter. The easiest way to understand this is to see it. The function in Listing 5-13 evaluates each parameter passed to determine the default value assigned.

Listing 5-13. Testing for unassigned parameters

```
function Invoke-UdfTestParameter
{
 [CmdletBinding()]
        param (
          [string]    $p_string,
          [int]       $p_int,
          [decimal]   $p_dec,
          [datetime]  $p_date,
          [bool]      $p_bool,
          [array]     $p_array,
          [hashtable] $p_hash
          )

#  Test the string...
   Switch ($p_string)
      {
       $null      { "String is null"  }
       ""         { "String is empty" }
       default    { "String has a value: $p_string" }
      }

    Switch ($p_int)
      {
       $null      { "Int is null"  }
       0          { "Int is zero" }
       default    { "Int has a value: $p_int" }
      }
```

```
    Switch ($p_dec)
     {
      $null      { "Decimal is null"  }
      0          { "Decimal is zero" }
      default    { "Decimal has a value: $p_dec" }
     }

    Switch ($p_date)
     {
      $null      { "Date is null"  }
      0          { "Date is zero" }
      default    { "Date has a value: $p_date" }
     }

    Switch ($p_bool)
     {
      $null      { "Boolean is null"  }
      $true      { "Boolean is True" }
      $false     { "Boolean is False" }
      default    { "Boolean has a value: $p_bool" }
     }

    Switch ($p_array)
     {
      $null      { "Array is null"  }
      default    { "Array has a value: $p_array" }
     }

    Switch ($p_hash)
     {
      $null      { "Hashtable is null"  }
      default    { "Hashtable has a value: $p_hash" }
     }

}
```

In this simple function we use the `Switch` statement to test the value of each parameter. On the left side is the test value that, if matched, will cause the code in the braces to the right to be executed. Let's call the function with no parameters with the command that follows:

```
Invoke-UdfTestParameter
```

The output to the console is shown here:

```
String is empty
Int is zero
Decimal is zero
Date is null
Boolean is False
Array is null
Hashtable is null
```

We can see from the output that only the `datetime`, `array`, and `hashtable` data types are assigned `null`. Strings default to an empty string. Numeric parameters are set to zero. Boolean parameters default to `False`. It is important to be aware of this behavior and write code accordingly. For example, if a function were writing sales values to a table and the `amount` parameter were omitted, `zero` might be written, erroneously indicating there were zero sales. When testing for a Boolean value, the function's user may not be aware that by omitting the parameter they are actually selecting the `False` value.

Distinguishing Parameter Default Values from Intended Values

The default values PowerShell assigns when no parameter value is passed can cause problems. For example, a function may need to know you intended to pass a zero to an `amount` parameter versus omitting the parameter or passing a null. The `CmdletBinding` supports a `Mandatory` attribute that forces the calling code to provide a value. However, that alone will not solve the issue. Let's look at the function in Listing 5-14, which uses the `Mandatory` attribute.

Listing 5-14. Using the mandatory attribute

```
function Invoke-UdfTestMandatoryParameter
{
 [CmdletBinding()]
        param (
          [Parameter(Mandatory = $true)]
          [string]    $p_string,
          [Parameter(Mandatory = $true)]
          [int]       $p_int,
          [Parameter(Mandatory = $true)]
          [bool]      $p_bool,
          [Parameter(Mandatory = $true)]
          [array]     $p_array,
          [Parameter(Mandatory = $true)]
          [hashtable] $p_hash
          )
  "String is : $p_string"
  "Int is: $p_int"
  "Boolean is: $p_bool"
  "Array is: $p_array"
  Write-Host "Hashtable is: "
  $p_hash
}
```

String Parameters

If we call the function in Listing 5-14 without any parameters, we will be prompted for each parameter value. For the `string` parameter, suppose we want to force the user to pass a value, but also want to allow the value to be a zero-length string. See the following:

```
Invoke-UdfTestMandatoryParameter -p_string ''
```

Since the string is the first parameter, we can omit the others for now. When we run the function call, the value is rejected because a zero-length string is not considered a value. Suppose we try to pass a `null` as the string parameter as shown here:

```
Invoke-UdfTestMandatoryParameter -p_string $null
```

We still get an error. What we want is to accept an empty string but reject `null` or having the parameter omitted from the call. We can do this by using the `AllowEmptyString` attribute as shown in **bold** in Listing 5-15.

Listing 5-15. Using AllowEmptyString

```
function Invoke-UdfTestMandatoryParameter2
{
 [CmdletBinding()]
        param (
          [Parameter(Mandatory = $true)]
          [AllowEmptyString()]
          [string]    $p_string,
          [Parameter(Mandatory = $true)]
          [int]       $p_int,
          [Parameter(Mandatory = $true)]
          [bool]      $p_bool,
          [Parameter(Mandatory = $true)]
          [array]     $p_array,
          [Parameter(Mandatory = $true)]
          [hashtable] $p_hash
          )
  "String is : $p_string"
  "Int is: $p_int"
  "Boolean is: $p_bool"
  "Array is: $p_array"
  Write-Host "Hashtable is: "
  $p_hash
}
```

By adding the `AllowEmptyString` attribute, we get the desired behavior. Let's try a few calls to test this:

```
Invoke-UdfTestMandatoryParameter2                 # Will be prompted for parameter values.

Invoke-UdfTestMandatoryParameter2 -p_string ''    # Parameter passes validation.

Invoke-UdfTestMandatoryParameter -p_string $null  # Parameter fails validation.
```

The first call causes PowerShell to prompt you for a value. The second call accepts the string value. The third call rejects the string value.

Numeric Parameters

What about numeric parameters? Ideally, we want to accept zero only if it is passed to the function. Omitting the parameter or passing a `null` should be rejected. Let's test calling our function while omitting the integer parameter as shown here:

```
Invoke-UdfTestMandatoryParameter2 -p_string ''
```

When we run this statement, we are prompted for a value for `$p_int`. Let's try again but pass a `null` to `$p_int`:

```
Invoke-UdfTestMandatoryParameter2 -p_string '' -p_int $null
```

PowerShell accepted the `null` value, so it prompts us for the next parameter, `$p_bool`. What we want is to require that a value for `$p_int` be passed and to not allow it to be `null`. However, a zero should be accepted. Unfortunately, if we pass `$null` to the `$p_int` parameter, it is translated into a zero before it gets validated. How can we distinguish an intentional zero versus a null? I found the answer to this question at the link here:

http://stackoverflow.com/questions/25459799/passing-null-to-a-mandatory-parameter-to-a-function

The solution is ingenious in that by replacing the parameter data type `int` with the .NET equivalent, `[System.Nullable[int]]`, that allows nulls, we can now accept zero and reject null values. Let's look at the revised function in Listing 5-16.

Listing 5-16. Testing for null integers

```
function Invoke-UdfTestMandatoryParameter2
{
 [CmdletBinding()]
        param (
          [Parameter(Mandatory = $true)]
          [AllowEmptyString()]
          [string]    $p_string,
          [Parameter(Mandatory = $true)]
          [ValidateNotNull()]
          [System.Nullable[int]]  $p_int,
          [Parameter(Mandatory = $true)]
          [bool]      $p_bool,
          [Parameter(Mandatory = $true)]
          [array]     $p_array,
          [Parameter(Mandatory = $true)]
          [hashtable] $p_hash
          )
  "String is : $p_string"
  "Int is: $p_int"
  "Boolean is: $p_bool"
  "Array is: $p_array"
  Write-Host "Hashtable is: "
  $p_hash
}
```

Now let's test the function by passing null for $p_int:

```
Invoke-UdfTestMandatoryParameter2 -p_string '' -p_int $null
```

As expected, the null value is rejected. Now let's try passing a zero into $p_int:

```
Invoke-UdfTestMandatoryParameter2 -p_string '' -p_int 0
```

The zero value is accepted, so we are prompted for $p_bool. Just press Control+Break to exit the prompt.

Boolean Parameters

Remember, PowerShell supports the [switch] type, which in many cases is better suited to a task than a Boolean parameter. Since the boolean type parameter defaults to false, how do we require the user to provide a value and still allow both true and false? In this case, PowerShell does not translate the default before doing the validation, so we can address this by simply adding the ValidateNotNull attribute, as shown in Listing 5-17.

Listing 5-17. Testing for null Boolean parameters

```
function Invoke-UdfTestMandatoryParameter3
{
 [CmdletBinding()]
        param (
          [Parameter(Mandatory = $true)]
          [AllowEmptyString()]
          [string]    $p_string,
          [Parameter(Mandatory = $true)]
          [ValidateNotNull()]
          [System.Nullable[int]]  $p_int,
          [Parameter(Mandatory = $true)]
          [ValidateNotNull()]
          [bool]      $p_bool,
          [Parameter(Mandatory = $true)]
          [array]     $p_array,
          [Parameter(Mandatory = $true)]
          [hashtable] $p_hash
          )
  "String is : $p_string"
  "Int is: $p_int"
  "Boolean is: $p_bool"
  "Array is: $p_array"
  Write-Host "Hashtable is: "
  $p_hash
}
```

Let's test the change with the two statements that follow:

```
Invoke-UdfTestMandatoryParameter3 -p_string '' -p_int 0 -p_bool $null     # Rejected

Invoke-UdfTestMandatoryParameter3 -p_string '' -p_int 0 -p_bool $false    # Accepted
```

The first statement is rejected due to the null $p_bool parameter. The second statement has the $p_bool parameter accepted, so we are prompted for the $p_array parameter.

Array Parameters

A common requirement for an array parameter would be to allow an empty array but require the parameter, and to not allow a `null` value. We can get this behavior by adding the `AllowEmptyCollection` attribute to the parameter. The code in Listing 5-18 has this modification.

Listing 5-18. Using AllowEmptyCollection

```
function Invoke-UdfTestMandatoryParameter3
{
 [CmdletBinding()]
        param (
          [Parameter(Mandatory = $true)]
          [AllowEmptyString()]
          [string]    $p_string,
          [Parameter(Mandatory = $true)]
          [ValidateNotNull()]
          [System.Nullable[int]]  $p_int,
          [Parameter(Mandatory = $true)]
          [ValidateNotNull()]
          [bool]      $p_bool,
          [Parameter(Mandatory = $true)]
          [AllowEmptyCollection()]
          [array]     $p_array,
          [Parameter(Mandatory = $true)]
          [hashtable] $p_hash
          )
  "String is : $p_string"
  "Int is: $p_int"
  "Boolean is: $p_bool"
  "Array is: $p_array"
  Write-Host "Hashtable is: "
  $p_hash
}
```

Let's test the function with the following statement:

```
Invoke-UdfTestMandatoryParameter3 -p_string '' -p_int 0 -p_bool $false -p_array $null
```

This statement will cause PowerShell to reject the `$p_array` parameter. Let's try the call with an empty array as shown here:

```
Invoke-UdfTestMandatoryParameter3 -p_string '' -p_int 0 -p_bool $false -p_array @()
```

PowerShell accepts the empty array from the statement.

Interestingly, hashtable parameters behave differently than arrays do. For the hashtable parameter in the previous function, a `null` value will be rejected without adding the `[AllowEmptyCollection()]` attribute but an empty hashtable is accepted. Let's test this with the statements that follow:

```
Invoke-UdfTestMandatoryParameter3 -p_string '' -p_int 0 -p_bool $false -p_array @()
-p_hash $null  # Rejected
Invoke-UdfTestMandatoryParameter3 -p_string '' -p_int 0 -p_bool $false -p_array @()
-p_hash @{}   # Accepted
```

Validating Parameters the Old-Fashioned Way

Sometimes, with so many advanced features, we forget that we can do things ourselves using the traditional approach. If the built-in `CmdletBinding` validation attributes do not fit your needs, you can write the validation code into the function itself, as shown in Listing 5-19.

Listing 5-19. Validating parameters in the function

```
function Invoke-UdfTestManualValidation
{
 [CmdletBinding()]
        param (
          [string]    $p_string,
          [System.Nullable[int]]  $p_int,
          [bool]      $p_bool,
          [array]     $p_array,
          [hashtable] $p_hash
          )

  if ($p_string.Length -eq 0) {  Write-Host -foregroundcolor Red 'Specify a string value
  please.';       return    }

  if ($p_int -le 0) {  Write-Host -foregroundcolor Red 'Specify an integer greater than zero
  please.';        return  }

  if ($p_array.Count -eq 0 ) {  Write-Host -foregroundcolor Red 'Specify an array with at
  least one element please.';    return  }

  if ($p_hash.Count -eq 0) {  Write-Host -foregroundcolor Red 'Specify a hashtable with at
  least one element please.';    return  }

  "String is : $p_string"
  "Int is: $p_int"
  "Boolean is: $p_bool"
  "Array is: $p_array"
  Write-Host "Hashtable is: "
  $p_hash
}
```

This function is simple but shows that you can write custom code to do any validations you need. It just tests the values of the parameters passed into the function. Note: The `Return` statement at the end of each code block exits the function. Play with calling the function with various values. The statement that follows passes the validations:

```
Invoke-UdfTestManualValidation -p_string 'x' -p_int 1 -p_array @(1) -p_hash @{"a"="Test"}
```

Using Parameter Sets

Object-orientated languages usually support a feature called *function overloading*. This feature allows multiple functions with the same name to be created with different sets of parameters. When the function is called, the correct function is determined based on the parameter data types passed. In this way, the function will behave differently depending on the parameters passed. For example, we might have a function called `get_size` that returns the size of the variable passed to it. If the parameter is a string, we get back the length in bytes. If the parameter passed is a number, we get the value of the number. If the object passed is an array, we get the number of elements in the array. The nice thing about this feature is that it simplifies using a function library. For example, regardless of what the data is, you always call the `print-output` function. If we had a separate `print` function for every type of object we needed to print, the library would become cumbersome to use. Technically, PowerShell does not support function overloading. However, it has a feature called Parameter Sets that can be used to accomplish the same thing.

When we declare the parameters to a function, we can group them with a name, i.e., `ParameterSetName`. When the function is called, PowerShell determines which parameter set to use based on the parameter data types passed to the function. Let's consider an example. Suppose we want to have a function that will add type parameters. If two integers are passed to it, they should be added together. If two strings are passed to it, they should be concatenated. If a datetime and integer are passed to it, the integer value should be added as days to the datetime. Let's look at the function in Listing 5-20.

Listing 5-20. Using parameter sets to similate function overloading

```
function Invoke-UdfAddValue
{
[CmdletBinding(SupportsShouldProcess=$false)]
        param (
            [Parameter(Mandatory = $true, Position = 0, ParameterSetName = "number")]
            [int]$number1,
            [Parameter(Mandatory = $true, Position = 1, ParameterSetName = "number")]
            [int]$number2,
            [Parameter(Mandatory = $true, Position = 0, ParameterSetName = "string")]
            [string]$string1,
            [Parameter(Mandatory = $true, Position = 1, ParameterSetName = "string")]
            [string]$string2,
            [Parameter(Mandatory = $true, Position = 0, ParameterSetName = "datetime")]
            [datetime]$datetime1,
            [Parameter(Mandatory = $true, Position = 1, ParameterSetName = "datetime")]
            [int]$days
            )

   Switch ($PScmdlet.ParameterSetName)
   {
   "number"    { "Added Value = " + ($number1 + $number2)   }
   "string"    { "Added Value = " + $string1 + $string2   }
   "datetime"  { "Added Value = " + $datetime1.AddDays($days)  }
   }
}
```

In Listing 5-20, we can see there are three parameter sets, i.e., with the `ParameterSetNames` of `number`, `string`, and `datetime`. Depending on the parameter set used, a different action will be performed by the function. Providing a single method that can work with different types of objects is known as *polymorphism*, a key feature of an object-orientated language. Notice that the `Position` attribute uses positions 0 and 1 for all the parameter sets. This allows us to call the function and pass the parameters by position. Depending on the data types we pass, PowerShell will select the corresponding parameter set. Let's run the statements that follow to test our function:

```
Invoke-UdfAddValue 1 2
Invoke-UdfAddValue "one" "two"
Invoke-UdfAddValue (Get-Date)  5
```

We get output like the following lines:

```
Added Value = 3
Added Value = onetwo
Added Value = 12/11/2014 19:02:39
```

The one requirement for this to work is that for each parameter set, there must be at least one parameter that is not in any other sets.

We can have a parameter that participates in more than one parameter set. We do this by having each parameter set's name specified. The function in Listing 5-21 has a third parameter that is in both parameter sets. Note: If no parameter set is specified, the parameter is in all parameter sets.

Listing 5-21. Example of a parameter that is in multiple parameter sets

```
function Invoke-UdfMuliParameterSet
{
[CmdletBinding()]
        param (
            [Parameter(Mandatory = $true, Position = 0, ParameterSetName = "set1")]
            [int]$int1,
            [Parameter(Mandatory = $true, Position = 0, ParameterSetName = "set2")]
            [string]$string2,
            [Parameter(Mandatory = $true, Position = 1, ParameterSetName = "set2")]
            [Parameter(ParameterSetName = "set1")]
            [string]$string3
            )

   Switch ($PScmdlet.ParameterSetName)
   {
   "set1"    { "You called with set1."   }
   "set2"    { "You called with set2."   }
   }
   "You Always pass '$string3' which has a value of $string3."
}
```

We call the function as follows:

```
Invoke-UdfMuliParameterSet "one" "two"
```

Since only parameter set `set2` has two string parameters, PowerShell selects that parameter set. `$string3` is in both parameter sets.

Sometimes it is not clear to PowerShell which parameter set to use based on the data types passed. To avoid having the call fail, we can define a default parameter set to be used. Consider this function:

```
function Invoke-UdfDefaultParameterSet
{
[CmdletBinding()]
        param (
            [Parameter(Mandatory = $true, Position = 0, ParameterSetName = "set1")]
            [string]$string1,
            [Parameter(Mandatory = $true, Position = 0, ParameterSetName = "set2")]
            [string]$string2
            )

   Switch ($PScmdlet.ParameterSetName)
   {
   "set1"    { "You called with set1."   }
   "set2"    { "You called with set2."   }
   }

}
```

In this function, we have two parameter sets with the same data type. Try running it as shown here:

```
Invoke-UdfDefaultParameterSet "test"
```

The call fails because PowerShell cannot determine which parameter set to use. Let's try revising the function to add a `DefaultParameterseName` attribute, as shown in Listing 5-22.

Listing 5-22. Assigning a default parameter set

```
function Invoke-UdfDefaultParameterSet
{
[CmdletBinding(DefaultParametersetName="set1")]
        param (
            [Parameter(Mandatory = $true, Position = 0, ParameterSetName = "set1")]
            [string]$string1,
            [Parameter(Mandatory = $true, Position = 0, ParameterSetName = "set2")]
            [string]$string2
            )

   Switch ($PScmdlet.ParameterSetName)
   {
   "set1"    { "You called with set1."   }
   "set2"    { "You called with set2."   }
   }

}

Invoke-UdfDefaultParameterSet "test"
```

When we call the function as shown above, we get the message "`You called with set1`."

Using Switch Parameters with Parameter Sets

A handy technique for leveraging multiple parameter sets is to combine them with `switch` parameters. By including a `switch`—which groups the parameters—the function is easy to read. The switch serves as a parameter PowerShell can use to identify the correct parameter set. As an example, the function that follows will perform one of two actions. If called with the `-dodelete` parameter, it will delete the file identified in the `$filetodelete` parameter. If called with the `-docopy` parameter, the function will copy the file name in `$filefrom` to the file name in `$fileto`. Since the `switch` parameters are in different parameter sets, PowerShell will not let you pass both of them, and if you pass one of them, the related parameters will be required. Let's look at Listing 5-23.

Listing 5-23. Using a switch parameter with parameter sets

```
function Invoke-UdfFileAction {
[CmdletBinding()]
        param (
            [Parameter(Mandatory = $true, Position = 0, ParameterSetName = "delete")]
            [switch]$dodelete,
            [Parameter(Mandatory = $true, Position = 1, ParameterSetName = "delete")]
            [string]$filetodelete,
            [Parameter(Mandatory = $true, Position = 0, ParameterSetName = "copy")]
            [switch]$docopy,
            [Parameter(Mandatory = $true, Position = 1, ParameterSetName = "copy")]
            [string]$filefrom,
            [Parameter(Mandatory = $true, Position = 2, ParameterSetName = "copy")]
            [string]$fileto
            )

if ($dodelete)
   {
     try
     {
         Remove-Item $filetodelete -ErrorAction Stop
         write-host "Deleted $filetodelete ."
     }
     catch
     {
         "Error:  Problem deleting old file."
     }
   }

if ($docopy)
   {
     try
     {
         Copy-Item $filefrom $fileto -ErrorAction Stop
         write-host "File $filefrom copied to $fileto."
     }
     catch
     {
         "Error:  Problem copying file."
     }
   }
}
```

The following code shows some examples of calling the function `Invoke-UdfFileAction` using the different parameter sets:

```
"somestuff" > somedata.txt

Invoke-UdfFileAction -dodelete 'somedata.txt'

"somestuff" > copydata.txt
Invoke-UdfFileAction -docopy 'copydata.txt' 'newcopy.txt'
```

In the function in Listing 5-23, we tested which switch was passed using the `if` statement, and since a switch is a Boolean, we just need to say `if ($myswitch)` to test the value. If the switch is not passed, its value is `false`. The `Try` statement catches any errors. Notice that we use the `ErrorAction` parameter on the `Remove-Item` and `Copy-Item` cmdlets to force an error if there is a problem—i.e., the `Catch` block will handle it. The technique of combining `switch` parameters with parameter sets provides a great way to build multi-purpose functions that are easy to use and maintain. Admittedly, this example puts two very different behaviors together into one function. This is just to make the point that parameter sets can make functions quite dynamic.

ValueFromPipeline and ValueFromPipelineByPropertyName Attributes

We saw earlier how to take data in from the pipeline. We can combine the pipeline with passed parameters to get even greater flexibility. By setting the parameter attribute `ValueFromPipeline=$true`, we flag a parameter as coming from the pipeline. The pipeline collection can contain any number of properties. PowerShell lets us map specific pipeline properties to a parameter with the `ValueFromPipelineByPropertyName` attribute. When we use this attribute, we need to make the parameter name match the property from the pipeline we want. To see how this works, we'll revise the `Invoke-UdfFileAction` function we saw earlier by having the `copy` action take file names from the pipeline. Let's look at the code in Listing 5-24.

Listing 5-24. Using ValueFromPipelineByPropertyName

```
function Invoke-UdfFileAction {
[CmdletBinding(SupportsShouldProcess=$true)]
        param (
            [Parameter(Mandatory = $true, Position = 0, ParameterSetName = "delete")]
            [switch]$dodelete,
            [Parameter(Mandatory = $true, Position = 1, ParameterSetName = "delete")]
            [string]$filetodelete,
            [Parameter(Mandatory = $true, Position = 0, ParameterSetName = "copy")]
            [switch]$docopy,
            [Parameter(Mandatory = $true,
                       ValueFromPipelineByPropertyName=$True, ParameterSetName = "copy")]
            [string[]]$fullname,
            [Parameter(Mandatory = $true,
                       ValueFromPipelineByPropertyName=$True, ParameterSetName = "copy")]
            [long[]]$length,
            [Parameter(Mandatory = $true, Position = 1, ParameterSetName = "copy")]
            [string]$destpath
            )
```

```
Begin
{
if ($dodelete)
   {
     try
     {
         Remove-Item $filetodelete -ErrorAction Stop
         write-host "Deleted $filetodelete ."
     }
     catch
     {
         "Error:  Problem deleting old file."
     }
   }
}

Process
{
if ($docopy)
   {
     try
     {
         foreach ($filename in $fullname)
         {
         Copy-Item $filename $destpath -ErrorAction Stop
         write-host "File $filename with length of $length copied to $destpath."
         }
     }
     catch
     {
         "Error:  Problem copying file."
     }
   }
}
}
```

The first thing to notice in the function is that the copy parameter set's file name property is called `$fullname` and now has the `$ValueFromPipelineByName=$true`. This tells PowerShell that 1) the value will be provided by the pipeline and 2) the property we want passed into this parameter is the `FullName` property. We also get the `$length` property from the pipeline, so we can display the length of each file as it is copied. This is included to show that you can pull multiple properties by name from the pipeline. Notice also that we now need to have separate code blocks in the function. By default, a function's code runs in the `Begin` block, so it does not need to be explicitly coded. However, now we need the `Begin` and the `Process` blocks. The `Begin` block will handle the `Delete` action, which does not use the pipeline. The `Process` block is used to copy each item in the pipeline.

The statements that follow will call our new function. Note: We're using the `–Whatif` parameter so that we can see what would happen without actually doing anything.

```
"somestuff" > somedata.txt

Invoke-UdfFileAction -dodelete 'somedata.txt' -whatif

Get-ChildItem *.txt | Invoke-UdfFileAction -docopy 'c:\users\BryanCafferky\hold\' -WhatIf
```

Don't forget that we can still call the function without using the pipeline. The statement here calls the function to copy a single file, passing the `$fullname` directly:

```
Invoke-UdfFileAction -docopy –length 0  -fullname 'c:\users\BryanCafferky\test.txt' `
'c:\users\BryanCafferky\hold\' –WhatIf
```

The `length` parameter does not make sense, since the user would have to look it up. In a case like this, it is easier just to pass a zero in. If you just want to take the whole pipe in, use the parameter attribute `$ValueFromPipeline=$true`. However, you will need to write code to pull out the properties you need, and the user could pass the function a pipe without the required property.

Internationalization with the Data Section

Nowadays, businesses are global and need to support multiple cultures and languages. PowerShell has a nice feature call the data section to support an easy implementation of multiple languages. The idea is to put all the strings that will be displayed into a separate file from the code. A separate file will store the string values in a specific language. Remember, PowerShell is used by Windows, so you can see this implemented on your computer. If you go to the folder `C:\Windows\diagnostics\system\Power`, you will see a folder with the default language for your machine. On my machine I see a folder named `en-US`. In this folder are a number of files, all with the file extension "`.psd1`". These files are PowerShell data section scripts. Let's take a look at one named `RS_Balanced.psd1`:

```
# Localized    11/20/2010 12:55 PM (GMT)    303:4.80.0411     RS_Balanced.psd1
ConvertFrom-StringData @'
###PSLOC
set_Balanced=Set active power plan to Balanced
Report_name_SetActiveSchemeGuid_result=The result of setting active scheme GUID
Report_name_ActiveSchemeGuid=Power Plan Adjustment
activeschemeguid_original=Original Plan
activeschemeguid_reset=New Plan
###PSLOC
'@
```

If this code looks familiar, that's because it is mostly a `Here` string. The `ConvertFrom-StringData` is a cmdlet that loads the key/value pairs defined in the string into a hash table. The file is loaded by the PowerShell script `C:\Windows\diagnostics\system\Power\RS_Balanced.ps1`. Let's look at the first few lines of that script to see how it works:

```
# Copyright © 2008, Microsoft Corporation. All rights reserved.

# Break on uncaught exceptions
trap {break}

. .\Powerconfig.ps1

#Localization Data
Import-LocalizedData localizationString

Write-DiagProgress -activity $localizationString.set_Balanced
```

The line that runs the `Import-LocalizedData` cmdlet is loading the data section from `RS_Balances.psd1` into the hashtable variable `$localizationString`. When the statement `'Write-DiagProgress -activity $localizationString.set_Balanced'` is executed, it will display the text associated with the key 'set_Balanced', which we can see in the data section is `"Set active power plan to Balanced"`.

So, how did Windows know where to find the localization data section script? To determine the folder, Windows checked the default culture for the machine. Enter the following line at the PowerShell prompt:

```
$PSUICulture
```

On my machine this displays `en-US`. The first two characters identify the language as English. The two characters after the dash identify the country, i.e., dialect, as United States. The data section script is stored in a folder named `en-US` under the script that uses it. That explains how PowerShell knew where to find the file, but how did it know the name? Simple, the data section script has the same file name as the script that loads it, but with a `psd1` extension instead of the script extension of `ps1`. If I were a user in Germany, my culture setting would probably be `de-DE`. The Windows installation would create a subdirectory with that name and store the German-language versions of the strings there. This allows the same PowerShell script to display the language appropriate to the machine.

One of the challenges to supporting a language is knowing the correct localization code to use for the folder name. The link that follows can help you with this:

http://msdn.microsoft.com/en-us/library/windows/desktop/dd318693(v=vs.85).aspx

To get a better idea of how we can use data sections in our own code, let's consider a simple demonstration. We want to display the day of the week in the language of the user. We'll assume we only need to support US English (`en_US`), Germany's German (`de-DE`) and Spain's Spanish (`es-ES`). Under the folder where our PowerShell script will run, we'll create the following folders: `en-US`, `de-DE`, and `es-ES`. Our script will be called `translate.ps1`, so in each of these folders we will create a data section called `translate.psd1`. Let's review the content of these files next. Listing 5-25 contains the US English version. Listing 5-26 has the Germany German translation. Listing 5-27 has the Spain Spanish translation.

Listing 5-25. En-US translation to be saved as translate.psd1 in the subfolder en-US

```
#  Folder: en-US
ConvertFrom-StringData @"
    Sunday=Sunday
    Monday=Monday
    Tuesday=Tuesday
    Wednesday=Wednesday
    Thursday=Thursday
    Friday=Friday
    Saturday=Saturday
"@
```

Listing 5-26. de-DE translation to be saved as translate.psd1 in the subfolder de-DE

```
# Folder:  de-DE
ConvertFrom-StringData @"
    Sunday=Sonntag;
    Monday=Montag;
    Tuesday=Deinstag;
    Wednesday=Mittwoch;
```

```
    Thursday=Donnerstag;
    Friday=Freitag;
    Saturday=Samstag
"@
```

Listing 5-27. es-ES translation to be saved as translate.psd1 in the subfolder es-ES

```
# Folder: es-ES
ConvertFrom-StringData @"
    Sunday=domingo
    Monday=lunes
    Tuesday=martes
    Wednesday=miércoles
    Thursday=jueves
    Friday=viernes
    Saturday=sábado
"@
```

We have three files—one to translate each language. We're using English as an anchor language to make this easier to follow. The key is always in English, but the translation will be whatever language is appropriate for the user. Note: If we need to support another language in the future, we would just need to add another subfolder with the appropriate localization name and then store the file `translate.psd1` in it along with the required translation. Let's try the translations out with the script in Listing 5-28.

Listing 5-28. Translating to the default language

```
Import-LocalizedData -BindingVariable ds                    # Machine Default - English

$ds.Sunday
$ds.Monday
$ds.Tuesday
$ds.Wednesday
$ds.Thursday
$ds.Friday
$ds.Saturday
```

The output is shown below.

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```

The first line loads the data section using the `Import-LocalizedData` cmdlet. Again, since the data section script name is the same as the calling script, we do not need to supply the file name. The script will use the translation appropriate for your machine. On my machine, it will display English. Rather than change the default language on our machine to test the other translations, we supply the `UICulture` parameter to the `Import-LocalizedData` cmdlet, specifying the language we want. Let's try the German translation by passing the `UICulture` parameter `de-DE`, as shown in Listing 5-29.

Listing 5-29. Translating to the German language

```
Import-LocalizedData -BindingVariable ds -UICulture de-DE  # German

$ds.Sunday
$ds.Monday
$ds.Tuesday
$ds.Wednesday
$ds.Thursday
$ds.Friday
$ds.Saturday
```

Which produces the following output:

```
Sonntag
Montag
Deinstag
Mittwoch
Donnerstag
Freitag
Samstag
```

Finally, let's try the Spanish translation by passing the `UICulture` parameter `es-ES`, as shown in Listing 5-30.

Listing 5-30. Translating to the Spanish language

```
Import-LocalizedData -BindingVariable ds -UICulture es-ES  # Spanish

$ds.Sunday
$ds.Monday
$ds.Tuesday
$ds.Wednesday
$ds.Thursday
$ds.Friday
$ds.Saturday
```

The output for this is as follows:

```
domingo
lunes
martes
miércoles
jueves
viernes
sábado
```

This simple example shows how easy it is to add support for multiple languages. PowerShell lets you transparently leverage the localization configuration setting in Windows. Normally, we would not need to call the `Import-LocalizedData` with the language specified, because PowerShell would select the appropriate language for the machine on which the script is running. If there is any chance you will need to support multiple languages, it may be a good idea to separate your display strings into separate data section files so you can easily add more languages.

Summary

This chapter explained how to use PowerShell to write robust, reusable code. We discussed how to use `CmdletBinding` in our functions, with detailed discussions of using `CmdletBinding` general attributes and parameter-specific attributes. The general attributes control how the overall function behaves. The parameter-specific attributes give us control over each parameter, including value validation. Then we discussed what default values are given when a parameter is not passed to a function and how to handle them. We reviewed how to define default values for parameters. We discussed function overloading and how to simulate it using parameter sets. Then we discussed how to combine the pipeline with passed parameters to maximize the power and flexibility of our functions. Finally, we covered how to use PowerShell's unique features to implement multiple-language support in your code.