CHAPTER 4

Writing Scripts

This chapter focuses on PowerShell features critical to developing professional code. We will start by discussing the `Set-StrictMode` cmdlet that protects us from uninitialized variables, incorrect method calls, and invalid property references. Then we'll discuss how to gracefully handle errors raised when your scripts execute. Error handling leads us into a discussion of common cmdlet parameters, so called, because all the built-in cmdlets support them. Then we introduce the PowerShell Integrated Script Environment's debugging features. Since PowerShell has excellent support for event-driven programming, we proceed with a discussion of implementing Windows forms and events to provide a GUI for our scripts. Building on this, we show how events can be used with other .NET objects. Then we discuss PowerShell's implementation of transaction support via a set of special cmdlets. This is an important section, as PowerShell transactions are not what we might expect. Finally, we will briefly discuss the PowerShell ISE options; what they do, and how to use them. It is easy to overlook these features, but they can make development easier.

Strict Mode

PowerShell is very loose about enforcing variable initialization, property name references, and call formats. This can cause bugs that are difficult to track down. The `Set-StrictMode` cmdlet was added in version 2.0 to help with this. By default, strict mode is off, but you can turn it on with the `Set-StrictMode` cmdlet. You need to pass the `Version` parameter to tell PowerShell the level of enforcement you want.

To get the benefits of strict mode with the fewest rule enforcements, `-Version 1` is passed, i.e., `Set-StrictMode -Version 1`.

To get stronger enforcement, you want to pass `–Version 2`, i.e., `Set-StrictMode -Version 2`.

To get the strictest enforcement available, you would enter `Set-StrictMode -Version Latest`. The PowerShell documentation states that using `–Version Latest` will automatically give you the strictest rules available, even in future PowerShell releases. In other words, if Microsoft adds a new Version parameter value of 3, using `Latest` would automatically upgrade the enforcement to that level. Be careful if you use this, because it may mean that scripts that had been working suddenly break when you upgrade to a new release of PowerShell.

So, what do these `Version` values actually mean? The table that follows documents the specific rules enforced by each version.

Table 4-1. *Strict Mode Version Valid Values and Their Effect*

1.0	Variables referenced must be initialized except for references within strings.
2.0	All variables must be initialized.
	Will not allow references to non-existent properties of an object.
	Will not allow function calls that use the calling syntax of a method.
	Will not allow a variable without a name such as (`${}`).
Latest	Uses the strictest rules available with the idea that stronger versions will be added.

It would be nice if one of the versions forced variable types to be declared, as undeclared variables can also cause bugs. As a best practice, it is a good idea to set strict mode on in new scripts you write and try to declare variable types as much as possible. I hesitate to say *always*, because sometimes it can take a bit of trial and error to determine what object type is returned from a cmdlet.

■ **Note** An alternative to using `Set-StrictMode` is to use the `Set-PSDebug` cmdlet, but that approach affects all PowerShell scopes, whereas the `Set-StrictMode` just affects the current execution scope.

Error Handling

PowerShell supports error handling with the `Try/Catch` block for specific statements and the `Trap` statement as a general error handler. Error information is stored in the PowerShell `$Error` object. To control how we want PowerShell's built-in cmdlets to handle errors, we use the `ErrorAction` parameter and the `$ErrorActionPreference` variable.

Using the Try/Catch Block

Errors can be trapped in scripts by using the `Try/Catch` block. This can be combined with the cmdlet `ErrorAction` parameter to control how errors are processed. PowerShell provides information about the error to the `Catch` block via the current object variable `$_` and the `$Error` automatic variable. There are two types of errors; terminating and non-terminating. As the name implies, terminating errors will cause the program to stop. The good news is that terminating errors are easily trapped and handled. Non-terminating errors cannot be trapped by default and are usually generated by cmdlets. However, you can use the `ErrorAction` cmdlet parameter to tell PowerShell to treat non-terminating errors as if they were terminating errors, i.e., by passing `ErrorAction Stop`. Then the `Catch` block can handle the error. What follows is a script that will search a given folder for files that match a filename pattern, merge the file contents, and write out the merged file. It will also add the original filename to each line. This is a script I developed because the ETL tool I was using had to be mapped to a single input file. So why add the original filename to each line? Since the plan is to load the merged file into a SQL Server table, having the originating filename will help us track down the source of bad data. Let's take a look at Listing 4-1.

Listing 4-1. Merging files and using the Try/Catch block

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

# Let's use Here strings to create our input files...

@"
Joe Jones|22 Main Street|Boston|MA|12345
Mary Smith|33 Mockingbird Lane|Providence|RI|02886
"@ > outfileex1.txt

@"
Tom Jones|11 Ellison Stree|Newton|MA|12345
Ellen Harris|12 Warick Aveneu|Warwick|RI|02885
"@ > outfileex2.txt

#  We could make these parameters, but we'll just use variables so the code stands alone...
$sourcepath =  "c:" + $env:HOMEPATH + "\Documents"
$filter = "outfileex*.txt"

$filelist = Get-ChildItem -Path $sourcepath  -filter $filter

$targetpathfile =  "c:" + $env:HOMEPATH + "\Documents\outmergedex.txt"

try
{
 Remove-Item $targetpathfile -ErrorAction Stop
 }
catch
{
   "Error removing prior files."
    Write-Host $_.Exception.Message
}
finally
{
   Write-Host "Done removing file if it existed."
   "out files merged at " + (Get-Date) | out-file outmerge.log -append
}

foreach ($file in $filelist)
 {
     "Merging file $file..."
     $fc = Get-Content $file
     foreach ($line in $fc)  {$line + "|" + $file.name >> $targetpathfile  }
 }
```

Look at the Try/Catch block in listing 4-1. Remove-Item will try to delete the specified file, but since the file is not there, an error is generated. Our ErrorAction parameter is Stop, so we are forcing a terminating error, which means the code in the Catch block will execute and display the error message. The Finally block is always executed, so it is a good place to put clean-up code or things you always want done, such as logging. The foreach block is looping through each file in the list and loading the file contents into the variable $fc using the Get-Content cmdlet. Then the script loops through each line in the file, appends the file name, and writes the line out to the path specified in $targetpathfile. Remember: >> means to append to the filename that follows.

Sometimes it is useful to handle some types of errors differently than other errors. Some errors may be anticipated while others require intervention. The script in Listing 4-2 uses the Get-Process cmdlet to demonstrate how to handle one error type differently than others.

Listing 4-2. Using Catch to trap different types of errors

```
Try
{
    Get-Process -Id 255 -ErrorAction Stop
}
Catch [Microsoft.PowerShell.Commands.ProcessCommandException]
{
   write-host "The process id was not found." -ForegroundColor Red
}
Catch
{
  "Error: $Error" >> errorlog.txt
}
Finally
{
  "Process complete."
}
```

In this example, Get-Process tries to access process ID 255, which does not exist, so an error is thrown. Since the ErrorAction is Stop, the Catch blocks will be executed. The first Catch block tests for whether the error name is [Microsoft.PowerShell.Commands.ProcessCommandException] and will only execute if the name matches. The second Catch block processes any error. However, if the specific Catch block executes, the second general handler will not. So, how do you know what the error name is for a given error? If you can trigger the error, you can get the name with the following statement:

```
$error[0].Exception.GetType().FullName
```

Note: You can list multiple error names on the catch line to handle a list of error types the same way.

The Trap Statement

The Trap statement lets us define a default error handler, i.e., one that catches any errors not otherwise caught. While the Try/Catch block traps errors on specific statements, the Trap statement will catch any unhandled errors for the entire script. Consider the following code:

```
$Error.Clear()

trap { "Error Occurred: $Error." }
Get-Content "nosuchfile" -ErrorAction Stop

"The script continues..."
```

In the first line in the script, `$Error.Clear()` clears out any prior errors so the script is re-runnable. Having clean-up code at the top of scripts like this can really help when testing. The `trap` line defines a block of code between the braces to be executed on any terminating error. In this case, the file "`nosuchfile`" does not exist so error messages will be displayed. Since by default, after the `trap` block has executed, the script continues to run so the message "`The script continues`" will be written to the console. You can override this behavior and have the script stop by adding the `break` keyword. The code that follows shows the previous script but with the `break` statement in the `trap` block. Notice that the line "`The script does not continue`" is not written to the console:

```
trap { "Error Occurred: $Error."; break }
Get-Content "nosuchfile" -ErrorAction Stop

"The script does not continue..."
```

The `trap` statement has a bug that can cause some havoc if you are not aware of it. You cannot replace a `trap` statement block that was previously defined. Once set, any more `trap` statements are ignored. Listing 4-3 provides an example to illustrate this.

Listing 4-3. Demonstrates a trap cannot be replaced

```
trap { "Error Occurred: $Error." }
Get-Content "nosuchfile" -ErrorAction Stop

"The script continues..."

trap { "Another Error Occurred: $Error."; Break }

Get-Content "nosuchfile" -ErrorAction Stop

"This line will still execute"
```

The console output is shown here:

```
Error Occurred: Cannot find a process with the name "255". Verify the process name and call
the cmdlet again..
Get-Process : Cannot find a process with the name "255". Verify the process name and call
the cmdlet again.At
C:\Users\BryanCafferky\Documents\GitHub\Source\PowerShell\Book\Chapter04\ch04_script8_error_
trap_statement.ps1:5 char:1
+ Get-Process 255 -ErrorAction Stop
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (255:String) [Get-Process],
                              ProcessCommandException
    + FullyQualifiedErrorId : NoProcessFoundForGivenName,Microsoft.PowerShell.Commands.
                              GetProcessCommand

The script continues...
Error Occurred: Cannot find a process with the name "255". Verify the process name and call
the cmdlet again. Cannot find a process with the name "255
". Verify the process name and call the cmdlet again..
Get-Process : Cannot find a process with the name "255". Verify the process name and call
the cmdlet again.At
```

```
C:\Users\BryanCafferky\Documents\GitHub\Source\PowerShell\Book\Chapter04\ch04_script8_error_
trap_statement.ps1:11 char:1
+ Get-Process 255 -ErrorAction Stop
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (255:String) [Get-Process],
                              ProcessCommandException
    + FullyQualifiedErrorId : NoProcessFoundForGivenName,Microsoft.PowerShell.Commands.
                              GetProcessCommand

This line will still execute
```

In listing 4-3, the first `trap` statement displays the error, and since it does not have a `break` statement, the script will continue, which is proven because the line "`The script continues...`" is displayed. Then a new trap is defined, and this one has the `break` statement. In spite of this, we can see that the old trap code was executed, as the message "`Another Error Occurred`" is not displayed and "`This line will still execute`" is displayed. There may be times when you want to modify the behavior of a `trap` block, but you can't do it directly. Instead, you can set variables that the `trap` block uses to modify its behavior. The code in Listing 4-4 uses this approach.

Listing 4-4. Workaround to replace a trap

```
[boolean] $stopscript = $false

trap { "Error Occurred: $Error."; If ($stopscript -eq $true) {break}  }
Get-Process 255 -ErrorAction Stop

"The script continues..."
$stopscript = $true

Get-Process 255 -ErrorAction Stop

"This line will NOT execute"
```

In this code, the `trap` block conditionally executes a `break` statement depending on the value of the variable `$stopscript`.

Using Try/Catch and the Trap Statement Together

Let's look at how to use the `Try/Catch` block together with a `Trap` statement to complement each other. Let's do this while solving a real-world problem. Sometimes database object names like table names and column names are changed. Subsequently, all the code that uses them needs to be updated with the new names.

For example, I once had a client decide to rename all the primary key columns in a source database. I needed a way to update the stored procedures that referenced these columns with the new names. I solved the problem by creating an old-to-new name mapping file in csv format, which was used to automate the updates.

To demonstrate, let's simplify the solution to just update one script. So you can set this up and test it, the code uses Adventure Works. First, let's create a copy of the table HumanResources.Department called HumanResources.CompanyUnit and then insert all the rows from HumanResources.Department to the new table. Use listing 4-5 to create the table. Note: You will need a copy of the AdventureWorks database to run the `Insert` statement. I used the SQL Server 2008 R2 version of AdventureWorks but the `Insert` should be compatible with later versions.

Listing 4-5. Creating the CompanyUnit table

```
CREATE TABLE [HumanResources].[CompanyUnit](
        [UnitID] [smallint]  NOT NULL,
        [UnitName] [dbo].[Name] NOT NULL,
        [UnitGroupName] [dbo].[Name] NOT NULL,
        [UpdateDate] [datetime] NOT NULL,
 CONSTRAINT [PK_DepartmentNew_UnitID] PRIMARY KEY CLUSTERED
(
        [UnitID] ASC
) )
Go

insert into  [HumanResources].[CompanyUnit]
select * from [HumanResources].[Department]
go
```

The original table will serve as the table we are migrating from and the new table will serve as the table we are migrating to. The SQL script that follows will serve as a legacy script that we will rewrite using our PowerShell script. Note: This script is a data file in the files accompanying the book and should be copied to your Documents folder.

```
USE AdventureWorks
GO

SELECT [DepartmentID]
      ,[Name]
      ,[GroupName]
      ,[ModifiedDate]
  FROM [HumanResources].[Department];

SELECT distinct DepartmentID
      ,[Name]
  FROM [HumanResources].[Department];

SELECT DepartmentID, [Name], GroupName, ModifiedDate
  FROM [HumanResources].[Department];
```

The text file named `columnmapping.csv` will serve as the driver to the remapping process. This file accompanies the book and should be copied to your Documents folder.

```
SourceColumn,TargetColumn
DepartmentID,UnitID
[Name],[UnitName]
GroupName,UnitGroupName
ModifiedDate,UpdateDate
```

The file has just two columns, `SourceColumn` and `TargetColumn`, which are the source column name and new column name, respectively. Having column names as the first row will make it easier to use this file with the `Import-CSV` cmdlet.

Let's look at Listing 4-6, which shows the script that rewrites the SQL code.

Listing 4-6. Script to modify SQL scripts

```
$sourcepath =  $env:HOMEDRIVE + $env:HOMEPATH + "\Documents\"

#  Let's trap any errors...
trap { "Error Occurred: $Error." >> ($sourcepath + "errorlog.txt") }

$file = Get-Content ($sourcepath + "script.sql") -ErrorAction Stop

$file = $file.replace('[HumanResources].[Department]', '[HumanResources].[CompanyUnit]');

#  Load the column Mappings...
$incolumnemapping = Import-CSV ($sourcepath + "columnmapping.csv")

foreach ($item in $incolumnemapping) { $file = $file.replace(($item.SourceColumn), ($item.
TargetColumn) ) }

Try
{
   $file >  ($sourcepath + "script_revised.sql")
   "File script.sql has been processed."

}
Catch
{
   "Error Writing file Occurred: $Error." >> ($sourcepath + "errorlog.txt")
}
Finally
{
   "Mapping process executed on " + (Get-Date) + "." >> ($sourcepath + "executionlog.txt")
}
```

Let's step through this code to see how it works.

```
$sourcepath =  $env:HOMEDRIVE + $env:HOMEPATH + "\Documents\"
```

To make the script runnable on any machine, the last line in the script sets the folder path to the logged-on user's Documents folder. The script will look for all files there.

We want a general error handler in case something fails, which the following line will take care of:

```
trap { "Error Occurred: $Error." >> ($sourcepath + "errorlog.txt") }
```

If an error occurs, the previous line will log the error message to `errorlog.txt`.

Now that we have a general error handler, we can start the real work. We will start by loading the script file into variable `$file` using the `Get-Content` cmdlet as shown:

```
$file = Get-Content ($sourcepath + "script.sql")
```

Since `$sourcepath` and `script.sql` need to be concatenated before the `Get-Content` uses it as a file path, the concatenation is enclosed in parentheses, which means it will be done prior to the `Get-Content` operation.

Let's look at the first remapping line of code:

```
$file = $file.replace('[HumanResources].[Department]', '[HumanResources].[CompanyUnit]');
```

This line just replaces all occurrences of the old table name with the new table name. I hard coded this to keep things simple. We need to assign the result to `$file` or the replace result will not be retained. We are using the `replace` method of the string object to do the replacement, but we could have used the `replace` operator instead. We are using the `replace` method because it does not support regular expressions whereas the `replace` operator does. In some cases we will want to use regular expressions, but since the input file has the special regular expression characters of periods and braces we do not want them used here. In fact, if you use the `replace` operator, you will get some odd results.

To load the column mappings, we can use the `Import-CSV` cmdlet as shown here:

```
$incolumnmapping = Import-CSV ($sourcepath + "columnmapping.csv")
```

We can now loop through each row of `$incolumnmapping` and use the `SourceColumn` and `TargetColumn` to replace each `SourceColumn` with the `TargetColumn` value. The code that follows does this:

```
foreach ($item in $incolumnmapping) { $file = $file.replace(($item.SourceColumn),
($item.TargetColumn) ) }
```

Finally, we want to write the new version of the script out to a file while handling any errors raised rather than use the default error handler we created with the `Trap` statement. Here we use the `Try/Catch/Finally` block as shown:

```
Try
{
   $file >  ($sourcepath + "script_revised.sql")
}
Catch
{
   "Error Writing file Occurred: $Error." >> ($sourcepath + "errorlog.txt")
}
Finally
{
   "Mapping process executed on " + (Get-Date) + "." >> ($sourcepath + "executionlog.txt")
}
```

Let's review this code. The code in the braces after the `Try` statement will execute. If an error is raised, the code between the `Catch` block braces will execute, which logs a message along with the error message provided by the `$Error` automatic variable. We are piping the `$file` variable into a file using the `>` operator to tell PowerShell to overwrite the file. We use `>>` when we want to append the data to the file, as in the case of logging errors.

In practice, you will probably need to update many stored procedures. You can use the SQLPS module to automatically write out all the stored procedures to text files, as the code in Listing 4-7 does.

Listing 4-7. Scripting out stored procedures

```
Import-Module "sqlps" -DisableNameChecking

$outpath =  $env:HOMEDRIVE + $env:HOMEPATH + "\Documents\storedprocedures.sql"

# Let's clear out the output file first.
"" > $outpath

# Set where you are in SQL Server...
set-location SQLSERVER:\SQL\BryanCafferkyPC\DEFAULT\Databases\Adventureworks\
StoredProcedures

foreach ($Item in Get-ChildItem)
{
    $Item.Schema + "_" + $Item.Name
    $Item.Script() | Out-File -Filepath $outpath -append
}
```

Here, the script starts by importing the SQLPS module so we can use its cmdlets. Then we create the variable `$outpath` to store the output file path. Note: This should work on any machine.

It's a good idea to clear out any contents that might be in the output file from a prior run, which we can do with this statement:

```
"" > $outpath
```

Here we are just writing an empty string to `$outpath` using the > operator, which overwrites the file contents with the empty string.

The SQLPS module lets us navigate SQL Server as if it were a file system. We can use `Set-Location` to place us in the virtual folder where the AdventureWorks stored procedures are located by using the statement that follows:

```
set-location SQLSERVER:\SQL\BryanCafferkyPC\DEFAULT\Databases\Adventureworks\
StoredProcedures
```

Since we can treat SQL Server like a file system, we can loop through the stored procedures as it they were files with this code:

```
foreach ($Item in Get-ChildItem)
{
    $Item.Schema + "_" + $Item.Name
    $Item.Script() | Out-File -Filepath $outpath -append
}
```

The `foreach` statement will loop through each stored procedure and save it in the `$Item` variable. We use `$Item` to write to the console the procedures schema and procedure name, separated by an underscore. The `Script` method will extract the code of the SQL object, i.e., the stored procedure code in this case. By piping this into the `Out-File` cmdlet with a `FilePath` of `$outfile` and using the `append` parameter, the stored procedure code is written to our output file. Note: In this example, we are writing all the code to one output file, but you can easily modify the script to write out separate files for each procedure by using the `$Item.Schema` and `$Item.Name` to build a unique output filename for each object.

As useful as this is, it may take some playing with to get reliable results. For example, if the word `Department` were to be replaced with a new value, it would change `Department`, `DepartmentID`, and `DepartmentName`. When the SQL code is auto-generated from the database, as we did earlier, it places braces around columns and table names, which can help, as we could replace `[Department]` and it would then update only the values required. It is well worth the time to automate these kinds of tasks. Once we have, we can modify the script to fit similar situations. As Mr. Scott on the old *Star Trek* series would say, "How do you think I got the reputation as a miracle worker?"

Setting the $ErrorActionPreference Preference Variable

The PowerShell variable `$ErrorActionPreference` controls how PowerShell will handle non-terminating errors for all cmdlet calls where the `ErrorActionPreference` parameter is not passed. The table that follows shows the possible values for `$ErrorActionPreference` and its effect.

Table 4-2. *ErrorActionPreference Values*

`Continue`	Shows the error message but continues running the script. It will not be trapped by an error handler as it does not raise an error. The error message is added to the $Error automatic variable.
`Ignore`	Suppresses error messages and continues running the script. It does not add the error to the $Error automatic variable.
`Inquire`	Shows the error message and prompts the user for the action to take. Choice may affect how the $Error variable is set.
`SilentlyContinue`	Suppresses error messages and continues running the script.
`Stop`	Displays error message and stops running the script.
`Suspend`	Only used to suspend Windows PowerShell workflows.

Using the $Error Variable and the ErrorAction Parameter

The `$Error` variable is an object and, except for when `ErrorActionPreference` is set to `Ignore`, will get populated with the error message. To prove this, let's try the script in listing 4-8. Play with it in the ISE to see how `$Error` is handled. Notice that the line that passes `ErrorAction Ignore` does not display an error message. The first two lines in listing 4-8 just clean things up. `Clear-Host` erases the console and the `Set-Location` is just used to clear out the SQL Server provider context which may be set from prior code.

Listing 4-8. Exploring ErrorAction

```
Clear-Host

Set-Location ($env:HOMEDRIVE + $env:HOMEPATH + "\Documents")
$Error.Clear()
Get-Content "nonfile" -ErrorAction Continue
"Continue: $Error "

$Error.Clear()
Get-Content "nonfile" -ErrorAction Ignore    # Error message is not set.
"Ignore: $Error "
```

```
$Error.Clear()
Get-Content "nonfile" -ErrorAction Inquire
"Inquire: $Error "

$Error.Clear()
Get-Content "nonfile" -ErrorAction SilentlyContinue
"SilentlyContinue: $Error "

$Error.Clear()
Get-Content "nonfile" -ErrorAction Stop
"Stop: $Error "

#  Suspend is only available for workflows.
$Error.Clear()
Get-Content "nonfile" -ErrorAction Suspend
"Suspend: $Error "
```

A nice feature of cmdlets is that they let us store error messages in a variable we specify via the `ErrorVariable` parameter. We can have the message overwrite previous values in the variable or have it append the messages. Let's look at the same code in Listing 4-9 to see how this works.

Listing 4-9. Storing the ErrorVariable

```
for($i=1; $i -le 2; $i++) {
Get-Process -Id 255 -ErrorAction SilentlyContinue -ErrorVariable hold
"Loop Iteration: $i"
}
write-host "`r`nError Variable contains..."
$hold
```

The script output is as follows:

```
Loop Iteration: 1
Loop Iteration: 2

Error Variable contains...
Get-Process : Cannot find a process with the process identifier 255.At
C:\Users\BryanCafferky\Documents\GitHub\Source\PowerShell\Book\Chapter04\ch04_script2_error_
variable.ps1:2 char:1
+ Get-Process -Id 255 -ErrorAction SilentlyContinue -ErrorVariable hold
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (255:Int32) [Get-Process],
                              ProcessCommandException
    + FullyQualifiedErrorId : NoProcessFoundForGivenId,Microsoft.PowerShell.Commands.
                              GetProcessCommand
```

The script in Listing 4-9 loops two times trying to execute `Get-Process`, passing an invalid process ID and thereby generating an error. The `ErrorAction` of `SilentlyContinue` tells PowerShell to store the message but not to display it. The `ErrorVariable` parameter is followed by the name of the variable where the message is to be stored. Notice that the $ is not included in the variable name. Each iteration replaces the variable with the error message, so at the end there is only the last message in the variable. If you want to append the error message to the variable, prefix the variable name with +. Listing 4-10 is the same code but with that small change.

Listing 4-10. Appending to the variable that stores ErrorVariable

```
for($i=1; $i -le 2; $i++) {
Get-Process -Id 255 -ErrorAction SilentlyContinue -ErrorVariable +hold
"Loop Iteration: $i"
}
write-host "`r`nError Variable contains..."
$hold
```

The script output is as follows:

```
Loop Iteration: 1
Loop Iteration: 2

Error Variable contains...
Get-Process : Cannot find a process with the process identifier 255.At
C:\Users\BryanCafferky\Documents\GitHub\Source\PowerShell\Book\Chapter04\ch04_script3_error_
variable2.ps1:4 char:1
+ Get-Process -Id 255 -ErrorAction SilentlyContinue -ErrorVariable +hold
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (255:Int32) [Get-Process],
                              ProcessCommandException
    + FullyQualifiedErrorId : NoProcessFoundForGivenId,Microsoft.PowerShell.Commands.
                              GetProcessCommand

Get-Process : Cannot find a process with the process identifier 255.At
C:\Users\BryanCafferky\Documents\GitHub\Source\PowerShell\Book\Chapter04\ch04_script3_error_
variable2.ps1:4 char:1
+ Get-Process -Id 255 -ErrorAction SilentlyContinue -ErrorVariable +hold
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (255:Int32) [Get-Process],
                              ProcessCommandException
    + FullyQualifiedErrorId : NoProcessFoundForGivenId,Microsoft.PowerShell.Commands.
                              GetProcessCommand
```

Being able to store error messages in our variables is useful if we want to process different types of error messages from different parts of a script.

The variable that collects multiple error messages is an array, and we can access an individual item using array notation, i.e., use `$hold[0]` to get the first item.

More on the $Error Variable

In addition to holding the error message, `$Error` is an object with a number of methods and properties. To get an idea of what these are, enter the following statement:

```
$error | Get-Member
```

And you should see output in the console similar to what follows:

```
Name                    MemberType        Definition
----                     ----------       ----------
Equals                  Method            bool Equals(System.Object obj)
GetHashCode             Method            int GetHashCode()
GetObjectData           Method            void GetObjectData(System....
GetType                 Method            type GetType()
ToString                Method            string ToString()
CategoryInfo            Property          System.Management.Automation.ErrorCategory...
ErrorDetails            Property          System.Management.Automation.ErrorDetails
                                          ErrorDetails...
Exception               Property          System.Exception Exception {get;}
FullyQualifiedErrorId   Property          string FullyQualifiedErrorId {get;}
InvocationInfo          Property          System.Management.Automation.InvocationInfo
                                          Invocat...
PipelineIterationInfo   Property          System.Collections.ObjectModel.
                                          ReadOnlyCollection[int]...
ScriptStackTrace        Property          string ScriptStackTrace {get;}
TargetObject            Property          System.Object TargetObject {get;}
PSMessageDetails        ScriptProperty    System.Object PSMessageDetails
                                          {get=& { Set-StrictMod..
```

■ **Note** `$Error.Count` will return the number of errors stored in the `$Error` object. `$Error.InvocationInfo` will return details about what caused the error.

Common cmdlet Parameters

`ErrorAction` is called a common cmdlet parameter because it is implemented in all cmdlets. Table 4-3 documents the other common parameters and how to use them.

Table 4-3. *Common cmdlet Paramters*

Parameter	Description	Example
`Debug`	Causes `Write-Debug` messages to print in custom code. You must have a cmdlet parameter binding in your code for this to work.	`Invoke-Something –Debug`
`ErrorAction`	Overrides the default `$ErrorActionPreference` variable to tell PowerShell how to handle non-terminating errors.	`Get-Content "stuff" –ErrorAction Stop`
`ErrorVariable`	If an error occurs, causes the error message to be stored in the specified variable in addition to `$Error`. Prefix with a + to have the output appended to variable contents.	`Get-Content "stuff" –ErrorVariable +v`
`OutVariable`	Stores the output of the command to the specified variable and displays it. Prefix with a + to have the output appended to variable contents.	`Get-Content "stuff" –OutVariable v`
`OutBuffer`	Specifies the number of items to accumulate in the pipeline before they are sent through the pipeline. If not specified, objects are sent as they are generated.	`Get-ChildItem -OutBuffer 15 | Where-Object -Property Name -like "t*"`
`PipelineVariable`	Stores the value of the current pipeline item to a variable.	`Get-ChildItem -OutVariable v`
`Verbose`	Causes extra informational messages to display for built-in PowerShell cmdlets and `Write-Verbose` messages to print in custom code. You must have a cmdlet parameter binding in your code for this to work.	`Invoke-Something -Verbose`
`WarningAction`	Controls how PowerShell should respond to warnings generated from the command. Valid values are `Continue`, `Inquire`, `SilentlyContinue`, and `Stop`.	`Get-Content –WarningAction Stop`
`WarningVariable`	Stores any warning messages generated by the command in the specified variable. Prefix with a + to have the output appended to variable contents.	`Get-Content –WarningVariable +v`
`WhatIf`	Will display a message about the effect of the command instead of executing the command. This parameter is only supported by cmdlets that can potentially do damage.	`Stop-Process 340 -WhatIf`
`Confirm`	Prompts for a confirmation before executing the command. This parameter is only supported by cmdlets that can potentially do damage.	`Stop-Process 340 -Confirm`

Debugging

PowerShell has some great debugging features, but the editor itself provides such nice visibility into the script variables that I find I rarely need the full debugging tools. Consider the following simple script:

```
$sourcepath =  $env:HOMEDRIVE + $env:HOMEPATH + "\Documents\state_table.csv"

$CheckFile = Import-CSV $sourcepath
```

After running the script in the ISE, the variables are loaded with values. Simply highlight the variable and click on the Execute Selected toolbar button as shown in Figure 4-1.

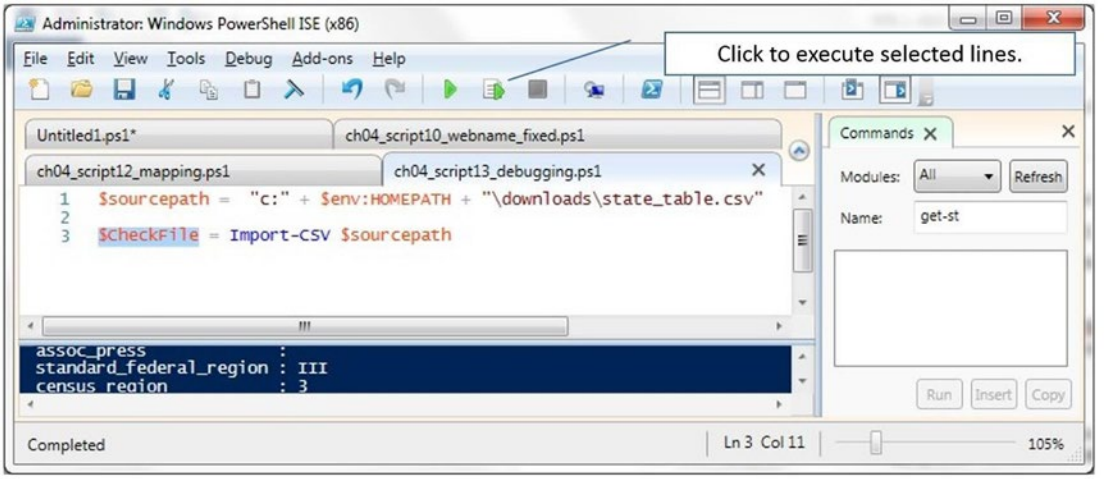

Figure 4-1. *Showing how to execute only selected lines in the PowerShell ISE*

Intellisense helps you find properties and methods on objects. By entering a period after an object name, the intellisense menu pops up, as shown in Figure 4-2. The `Count` property contains the number of lines in the variable.

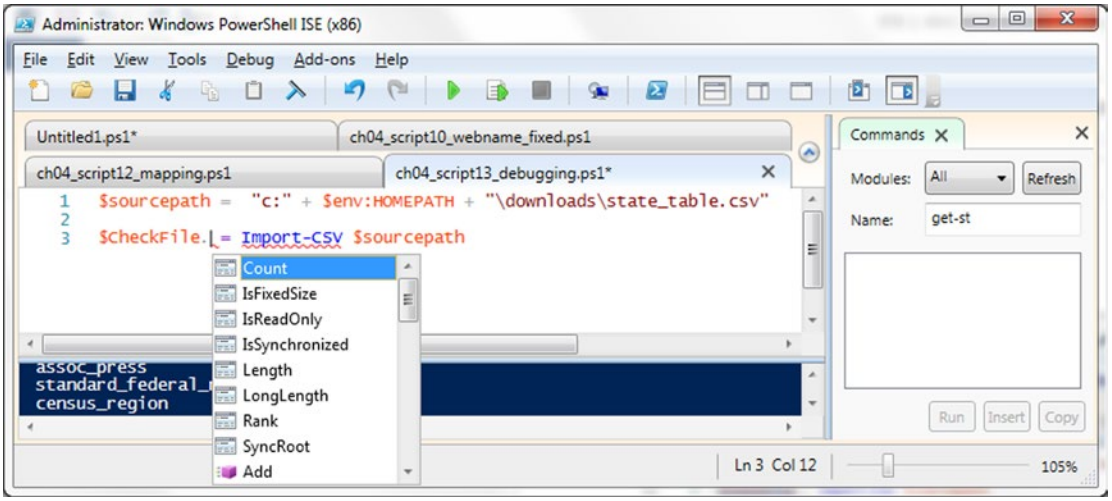

Figure 4-2. *Using intellisense in the PowerShell ISE*

I do most of my debugging by playing with selected lines of code and viewing the value of variables. Because of this, and because there are a lot of great references available on using the advanced debugging facilities of PowerShell, we will not go into it in detail here. A good reference on debugging and other subjects is available at this link:

http://gegeek.com/documents/eBooks/Windows%20Powershell%203.0%20First%20Steps.pdf

Events and Script Blocks: The Perfect Marriage

In this section, we will discuss how PowerShell supports events and how you can leverage this in your code. The best place to start this discussion is with Windows' form objects and events. PowerShell scripts can create Windows forms and then connect code to when an event—such as a button click—occurs. By first explaining how that works, we will leverage that knowledge to explain how you can use the same concepts on other .NET objects.

Using Windows Events

We've seen script blocks, and we know that Windows traps events, but PowerShell puts these together in way that greatly extends the functionality of event trapping. We can register for events we want trapped and assign code to be executed when the event occurs. It's surprisingly simple, yet powerful. A good way to demonstrate this is to walk through a simple script that displays a Windows form. Review Listing 4-11. Note: The window may appear in the background. If so, hovering the mouse over the PowerShell icon in the taskbar should bring it up.

Listing 4-11. Displaying a Windows form

```
#  Note:  This form code was originally generated by AdminScriptEditor 4.0
#         available for free at http://www.itninja.com/community/admin-script-editor

#  First, let's define our Windows form objects...

[void][System.Reflection.Assembly]::LoadWithPartialName("System.Windows.Forms")
[void][System.Reflection.Assembly]::LoadWithPartialName("System.Drawing")

#~~< Form1 >~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
$Form1 = New-Object System.Windows.Forms.Form
$Form1.ClientSize = New-Object System.Drawing.Size(167, 148)
$Form1.Text = "Form1"
#~~< Label1 >~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
$Label1 = New-Object System.Windows.Forms.Label
$Label1.Location = New-Object System.Drawing.Point(41, 50)
$Label1.Size = New-Object System.Drawing.Size(100, 23)
$Label1.TabIndex = 1
$Label1.Text = "Start"
#~~< Button1 >~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
$Button1 = New-Object System.Windows.Forms.Button
$Button1.Location = New-Object System.Drawing.Point(41, 87)
$Button1.Size = New-Object System.Drawing.Size(75, 23)
$Button1.TabIndex = 0
$Button1.Text = "Button1"
$Button1.UseVisualStyleBackColor = $true

# Second, add the controls to the form...
$Form1.Controls.Add($Label1)     #  Add the label to the form.
$Form1.Controls.Add($Button1)    #  Add the button to the form.

# Third, the function to handle the button click...
function DaButtonWasClicked( $object ){
  $Label1.Text = "Button Clicked";
}

# Fourth, hook up the button-click event to the function that will execute...
$Button1.add_Click({DaButtonWasClicked($Button1)})

# Fifth, declare the function to open the form and start the event trapping...
function Main{
        [System.Windows.Forms.Application]::EnableVisualStyles()
        [System.Windows.Forms.Application]::Run($Form1)
}

#  Finally, call the function Main to open the Windows form...
Main # This call must remain below all other event functions
```

A nice thing about PowerShell is that we can create a Windows form application completely within the language—i.e., no extra add-on to deploy. However, as we can see from the script, it takes a lot of lines of code to create the form objects. Thankfully there are tools out there to help. I found a free comprehensive PowerShell development tool called Admin Script Editor 4.0, available at the link: http://www.itninja.com/community/admin-script-editor, which supports designing Windows forms. While I was happy to get such a useful tool for free, it is unfortunate that the company that developed it has gone out of business. Therefore, there will not be any updates to it. Nonetheless, it is full featured and includes excellent support for developing Windows form–based applications in PowerShell. Eventually, I found I needed a full powered development environment that supported Windows forms and other advanced features. So I purchased Sapien's PowerShell Studio, which I have been very happy with. It's like Visual Studio for PowerShell. You can learn more about this product at the link: https://www.sapien.com/software/powershell_studio.

Let's review the code. Note: I commented the code sections so it is easier to follow what is happening.

```
#  First, let's define our Windows form objects...
[void][System.Reflection.Assembly]::LoadWithPartialName("System.Windows.Forms")
[void][System.Reflection.Assembly]::LoadWithPartialName("System.Drawing")

#~~< Form1 >~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
$Form1 = New-Object System.Windows.Forms.Form
$Form1.ClientSize = New-Object System.Drawing.Size(167, 148)
$Form1.Text = "Form1"
#~~< Label1 >~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
$Label1 = New-Object System.Windows.Forms.Label
$Label1.Location = New-Object System.Drawing.Point(41, 50)
$Label1.Size = New-Object System.Drawing.Size(100, 23)
$Label1.TabIndex = 1
$Label1.Text = "Start"
#~~< Button1 >~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
$Button1 = New-Object System.Windows.Forms.Button
$Button1.Location = New-Object System.Drawing.Point(41, 87)
$Button1.Size = New-Object System.Drawing.Size(75, 23)
$Button1.TabIndex = 0
$Button1.Text = "Button1"
$Button1.UseVisualStyleBackColor = $true
```

Notice how PowerShell easily creates the required .NET objects and calls their methods. The first thing the script does is create references to the Windows form object libraries via the lines here:

```
#  First, let's define our Windows form objects...
[void][System.Reflection.Assembly]::LoadWithPartialName("System.Windows.Forms")
[void][System.Reflection.Assembly]::LoadWithPartialName("System.Drawing")
```

Most of the time you don't need to explicitly define namespaces in PowerShell, but in the case of the Windows form libraries, you do. Once we have the namespaces defined we can create the form objects as shown here:

```
#~~< Form1 >~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
$Form1 = New-Object System.Windows.Forms.Form
$Form1.ClientSize = New-Object System.Drawing.Size(167, 148)
$Form1.Text = "Form1"
#~~< Label1 >~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
$Label1 = New-Object System.Windows.Forms.Label
$Label1.Location = New-Object System.Drawing.Point(41, 50)
$Label1.Size = New-Object System.Drawing.Size(100, 23)
$Label1.TabIndex = 1
$Label1.Text = "Start"
#~~< Button1 >~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
$Button1 = New-Object System.Windows.Forms.Button
$Button1.Location = New-Object System.Drawing.Point(41, 87)
$Button1.Size = New-Object System.Drawing.Size(75, 23)
$Button1.TabIndex = 0
$Button1.Text = "Button1"
$Button1.UseVisualStyleBackColor = $true
```

In the code above, we use `New-Object` to create the required form objects, i.e., the form, the label, and the button. Notice that after the object is defined, we can set the properties that control how it will be displayed. The `text` property is the caption that will be displayed for the object.

Now let's look at how the form objects are added to the form with the code here:

```
# Second, add the controls to the form...
$Form1.Controls.Add($Label1)     #  Add the label to the form.
$Form1.Controls.Add($Button1)    #  Add the button to the form.
```

The question comes to mind, where did these object names come from? The answer is that they came from the form editor when we created the form. The editor creates default names, but we can change them if we want to. I kept the default names to keep this simple. Figure 4-3 shows how the form looks in the editor. Since the button is selected, you can see its properties, including the `text` property, which is the display value, and the `name` property, which is the name of the object. Since the form editor will generate PowerShell code, it turns these names into variables by prefixing them with a `$`. If you review the code, you will see how each object was created as a variable in the script.

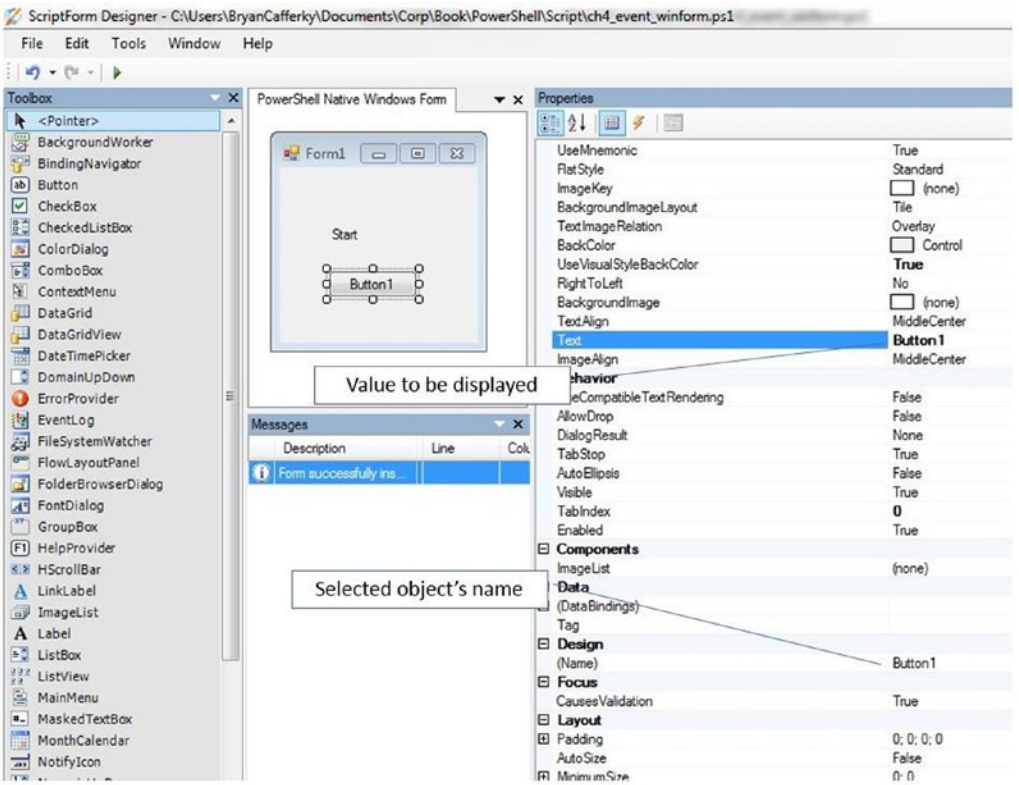

Figure 4-3. *The Windows Form Designer in AdminScriptEditor 4.0*

We can see that the form editor generated the related PowerShell code as standard .NET Windows calls storing the objects in PowerShell variables named using the object name we assigned in the editor.

The form editor lets us define event handlers when we click on the lightning bolt icon of the properties window. Figure 4-4 shows an event handler being defined for the button-click event.

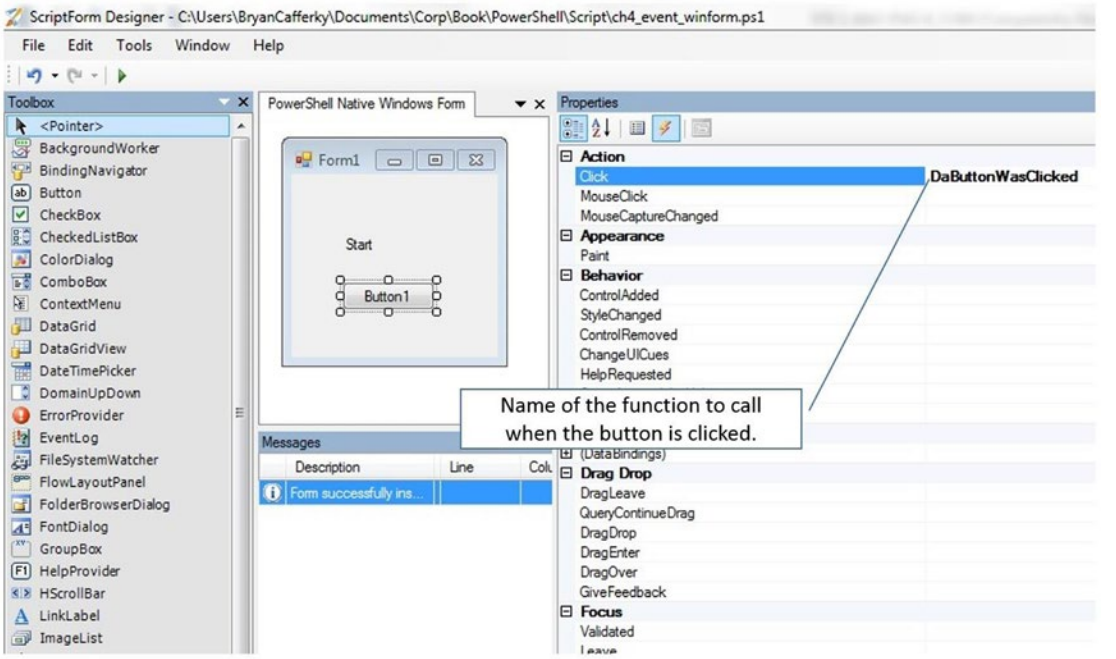

Figure 4-4. *Showing the Click event setting in the AdminScriptEditor 4.0 Windows Form Desginer*

Notice that the `Click` event property is `DaButtonWasClicked`. What does that mean? We are saying that when the user clicks on the button, a script block executes that calls a function named `DaButtonWasClicked`. A script block is just an inline script. Script blocks can be anonymous, i.e., unnamed, or can be assigned to a variable. The form editor does not let us actually enter the code to execute, but it will create the inline script that calls the function we specify. Then, we need to go into the script and add the function code as shown here:

```
# Third, the function to handle the button click...
function DaButtonWasClicked( $object ){
  $Label1.Text = "Button Clicked";
}
```

But how does the script block get connected to the event? PowerShell lets us connect code to an event with the object's `add` event method; i.e., `add_someevent` where `someevent` is the event name we want to hook the code to. `Click` is an event defined to Windows, so PowerShell understands we are linking the code to the `Click` event. Since we execute the `add` method on the `$Button1` object variable, it connects the event code to the button's click event. The line that follows connects the function `$DaButtonWasClicked` to `$Button`'s click event:

```
# Fourth, hook up the button-click event to the function that will execute...
$Button1.add_Click({DaButtonWasClicked($Button1)})
```

The code between the braces is the anonymous script block, i.e., `{DaButtonWasClicked($Button1)}`. We can see that our function will be called, passing the button object variable to it. Notice, the function definition accepts an object parameter. This allows the function access to the button's methods and properties.

We have the objects defined and code linked to the button's clicked event. Now we need the statements to open the form and get things going. The form editor wrote the code to do that for us. First, it created a function to create the objects and show the form. Then, it added a statement at the end of the overall script to call that function. The code is shown here:

```
# Fifth, declare the function to open the form and start the event trapping...
function Main{
        [System.Windows.Forms.Application]::EnableVisualStyles()
        [System.Windows.Forms.Application]::Run($Form1)
}

#  Finally, call the function Main to open the Windows form...
Main # This call must remain below all other event functions
```

The `Main` function first enables the visual elements of the application with the `EnableVisualStyles` method. The line following calls the `Run` method, passing the form object variable we created, `$Form1`, thereby opening the form. Since these two lines are defined in the function named `Main`, the last line of the script calls that function.

Using .NET Object Events

Similar to the way Windows events can call PowerShell code, we can subscribe to .NET object events and supply code that we want executed when the event fires. A common ETL requirement is to load external flat files into SQL Server tables when the files arrive. The problem is, how do we know when files arrive? One way to run code when a new file is created in a specific folder is to use the built-in .NET object `FileSystemWatcher`. Three events are supported by `FileSystemWatcher`: file created, file deleted, and file changed. The coding to register for an event is similar to what we have already seen, except for the event name. Let's look at Listing 4-12, which shows an example of calling a function that registers for a file-create event.

Listing 4-12. Trapping the file-create event

```
function Register-UdfFileCreateEvent([string] $source, [string] $filter)
{
   try
   {
     Write-Host "Watching $source for new files..."
     $filesystemwatcher = New-Object IO.FileSystemWatcher $source,
     $filter -Property @{IncludeSubdirectories = $false; NotifyFilter = [IO.
     NotifyFilters]'FileName, LastWrite'}
     Register-ObjectEvent $filesystemwatcher Created -SourceIdentifier FileCreated -Action {
     write-host "Your file has arrived. The filename is " $Event.SourceEventArgs.Name "."
}
    }
   catch
    {
      "Error registering file create event."
    }
}

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

In the code in listing 4-12, we start by defining a function named `Register-UdfFileCreateEvent` that takes a folder path as the first parameter and a file pattern filter as the second parameter. After displaying an informational message, we create the variable `$filesystemwatcher` using the `New-Object` cmdlet as an instance of the `IO.FileSystemWatcher` .NET object. The `IO.FileSystemWatcher` object requires that we pass the path to the folder we want monitored, which is why we pass `$source`. We pass `$filter` to limit the files being watched to just the files that match the filename pattern. We use the `Property` parameter to set some preferences, such as not including subdirectories, i.e., `@{IncludeSubdirectories = $false`. The statement '`NotifyFilter = [IO.NotifyFilters]'FileName, LastWrite`' is asking to watch for changes to the `FileName` and `LastWrite` properties.

Let's look at the line that tells Windows to monitor the event:

```
Register-ObjectEvent $filesystemwatcher Created -SourceIdentifier FileCreated -Action {
     write-host "Your file has arrived. The filename is " $Event.SourceEventArgs.Name "."
}
```

The `RegisterObjectEvent` cmdlet is used to assign the code we want executed when the `FileCreated` event is fired. The `Action` parameter defines the code block that will be executed—i.e., the code that is contained between the braces, {}. In the example, we just display a message that the file has arrived with the name of the file.

The line that runs all this is the one here:

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

This line calls the function `Register-UdfFileCreateEvent`, passing the folder to be monitored as the first parameter and the filename pattern filter as the second; i.e., only filenames that meet the filter pattern will cause our code to execute. When we run this, we get the following output:

```
Watching C:\Users\BryanCafferky\Documents\ for new files...

Id  Name            PSJobTypeName       State       HasMoreData     Location     Command
--  ----            -------------       -----       -----------     --------     -------
9   FileCreated                         NotStarted  False                        ...
```

To test this script, we use Notepad to create and save a new text file that meets the filter criteria. The console output should confirm the file-creation event code is firing, as shown here:

```
Your file has arrived. The file name is  testit.txt .
Your file has arrived. The file name is  testit.txt .
```

We are not seeing double. There is a bug in the `FileSystemWatcher` object that causes it to fire multiple times if the program that created the file performs multiple file system actions. For more details on this, see this link:

http://stackoverflow.com/questions/1764809/filesystemwatcher-changed-event-is-raised-twice

Nonetheless, this is still a useful function, and if you are monitoring an FTP folder or something similar, we may not see this happen. To trap file deletion or file changed, use event names `FileDeleted` and `FileChanged` respectively. One drawback of using PowerShell to trap events is that if the script stops running, the event will no longer fire your code. You can get around this by making the script a service and installing it on the server. Although the PowerShell ISE does not support compiling scripts, which is a required step in registering a program as a service. Both the Admin Script Editor and PowerShell Studio do support code compilation. Another free and popular PowerShell tool is PowerGUI, available at: `http://en.community.dell.com/techcenter/powergui`. This product has a *Compile Script into Service* option. See the link: `https://technet.microsoft.com/en-us/library/Hh849830%28v=wps.630%29.aspx` for information on registering a service using the `New-Service` cmdlet.

Using PowerShell Transactions

The PowerShell documentation covers a feature called `Transactions` that on the surface sounds great. The idea is that we can make all kinds of changes to the environment and then roll them back as if they never happened. First, we'll briefly review the overall concept. Then, we will discuss a real-world script that automates a Windows application and includes the use of transactions. Finally, we will discuss why the transaction support in the example does not work and the limitations of PowerShell transactions.

Conceptual Overview

As a database developer you can understand and appreciate SQL Server's support for transactions. The ability to define a set of updates that must all be successfully committed or rolled back is critical to complex database updates. So when we see that PowerShell supports transactions, we are intrigued. Imagine—we could delete files, move folders, update data, encrypt and unzip and perform any number of other actions, and if something went wrong we could just roll it back. The documentation seems to support that conclusion, but I could not find any examples of using transactions other than code that updated the Window's registry. Let's cover how transactions are supposed to work. First, transactions are supported by the cmdlets `Start-Transaction`, `Complete-Transaction`, and `Undo-Transaction`. Similar to database transactions, the overall flow would be something like in the following pseudo-code:

```
Start-Transaction
Some PowerShell statements that change things.
IF all updates were successful
        Complete-Transaction
Else
        Undo-Transaction
```

The program starts by creating a transaction context using the `Start-Transaction` cmdlet. A transaction context simply means a transaction has begun and subsequent statements may enroll themselves to be included in the transaction. By enrollment, I mean that all the updates in the transaction are either committed together or rolled back. The script makes changes to the environment under the transaction context and at the end, if everything was successful, the work is committed, meaning it is made permanent. The `Complete-Transaction` cmdlet commits the changes. If there was a problem, the script has the option of cancelling the changes, i.e., rolling them back, with the `Undo-Transaction` cmdlet. The idea is that some types of changes need to all happen together or the data is left in a corrupt state. For example, if a customer order header were committed to the database but the order detail were lost, the data for that order would not be in a valid state.

An Example of How Transactions Should Work

Now that we have the basics, let's discuss a scenario in which you might use this functionality. The state table object we saw earlier uses a file downloaded from a website, so it is reasonable to assume the data may change and need to be updated. We have a script that will download a new file and replace the old file with it. However, it is possible something could go wrong and the new file would not be complete. To handle this the script wraps the `file copy` operation in a transaction and checks the new file to see if there are at least 50 rows in it, and if not, rolls back the update using `Undo-Transaction`. Since we are covering transactions, we'll include some other useful ideas in the example too. The website does not just let us simply connect and download the file. Instead, there are a series of menus to navigate that determine the specifics of the data we want. So the example uses application automation, in which PowerShell clicks on links and enters data into the Internet Explorer session automatically, as shown in Listing 4-13.

■ **Note** Please be patient when you run the script as it may take a moment to load Internet Explorer.

Listing 4-13. A script using transactions

```
Import-Module WASP

$url = "http://statetable.com/"

get-process -name iexplore -ErrorAction SilentlyContinue | Stop-Process -Force -ErrorAction
SilentlyContinue
$ie = New-Object -comobject InternetExplorer.Application
$ie.visible = $true
$ie.silent = $true
$ie.Navigate( $url )

while( $ie.busy){Start-Sleep 1}
Select-Window "iexplore" | Set-WindowActive

$btn=$ie.Document.getElementById("USA")
$btn.Click()

while( $ie.busy){Start-Sleep 1}
$btn=$ie.Document.getElementById("major")
$btn.Click()

while( $ie.busy){Start-Sleep 1}
$btn=$ie.Document.getElementById("true")
$btn.Click()

while( $ie.busy){Start-Sleep 1}
$btn=$ie.Document.getElementById("current")
$btn.Click()

while( $ie.busy){Start-Sleep 1}
$btn = $ie.Document.getElementsByTagName('A') | Where-Object {$_.innerText -eq 'Do not
include the US Minor Outlying Islands '}
$btn.Click()
```

```
while( $ie.busy){Start-Sleep 1}
$btn=$ie.Document.getElementById("csv")
$btn.Click()

start-sleep 8
while( $ie.busy){Start-Sleep 1}
Select-Window iexplore | Set-WindowActive | Send-Keys "%S"

Start-Transaction -RollbackPreference TerminatingError

$sourcepath =  $env:HOMEDRIVE + $env:HOMEPATH + "\downloads\state_table.csv"
$targetpath =  $env:HOMEDRIVE + $env:HOMEPATH + "\Documents\state_table.csv"

Copy-Item $sourcepath $targetpath  -UseTransaction

$CheckFile = Import-CSV $targetpath

If ($CheckFile.Count -lt 50) { Write-host "Error"; Undo-Transaction } else { "Transaction
Committed" ; Complete-Transaction }
```

Let's review the code above step by step. The first statement, copied here, imports a module that greatly aids us in our task:

```
Import-Module WASP
```

We will cover modules in detail later. Modules extend PowerShell's functionality by adding cmdlets. There are many free modules available for download, and typically a module adds extra support for a specific type of functionality. For example, `SQLPS` adds SQL Server connectivity. The module being loaded above, `WASP`, adds some powerful Windows application interaction functionality. Unless you have already downloaded `WASP`, you will need to do so. You can download it from `http://wasp.codeplex.com/`. After you download it, extract the file `WASP.dll` and copy it to a folder where PowerShell looks for modules. To determine where that is, enter the following into PowerShell:

```
$env:PSModulePath
```

We should see a list of folder paths separated by semicolons. This environment variable holds the places PowerShell will look for modules. Pick one of the folders, ideally one under your documents folder, and create a new folder named `WASP` in it. Then paste the unzipped `WASP.dll` in that folder. A module must have its own folder. We should be able to import the module now.

■ **Note** A shortcut to the precedincg process is to create a folder named `WindowsPowerShell` in your Documents folder, create a folder named `Modules` under it, and add the `WASP` module folder and DLL there. That `WASP` module folder is the PowerShell default location for user modules.

Now we can look at the application automation code. If you have trouble running the code, it may be that IE is not giving your script the HTML source for the web page. If so, you should be able to resolve this by running PowerShell as administrator:

```
Get-Process -name iexplore -ErrorAction SilentlyContinue | Stop-Process -Force -ErrorAction
SilentlyContinue
$ie = New-Object -comobject InternetExplorer.Application
$ie.visible = $true
$ie.silent = $true
$ie.Navigate( $url )
```

The `Get-Process` is piping any running instances of Internet Explorer that are running into the `Stop-Process` cmdlet so that when we create a new instance it will be the only one. Then, we create a new instance of Internet Explorer and load the reference to it into the variable `$ie`. Using the `$ie` variable, we set the `visible` property to `$true` so the window will show. Setting it to `$false` will make the window invisible. The silent property is to reduce messages. Then we just use the `Navigate` method to go to the URL stored in the variable `$url`. The code below activates IE.

```
while( $ie.busy){Start-Sleep 1}
Select-Window "iexplore" | Set-WindowActive
```

These statements above wait for IE to finish bringing up the page and then set the focus to that screen.

Now we need to navigate through the web page to get the data, which we do with the following code:

```
$btn=$ie.Document.getElementById("USA")
$btn.Click()

while( $ie.busy){Start-Sleep 1}
$btn=$ie.Document.getElementById("major")
$btn.Click()

while( $ie.busy){Start-Sleep 1}
$btn=$ie.Document.getElementById("current")
$btn.Click()
```

The web page source code is available to the `$ie` object, and we can navigate around the web page by using the `document` object model. The first line searches the document for an element with the `ID = "USA"` and stores the reference in `$btn`. Then we can use the `$btn` click method to click the link. The `while` statement is just giving IE time to complete the action before continuing. The statement following the wait locates the link with the `ID = "major"` followed by a statement that clicks that. The same thing is done for the "`current`" link. What happens if the element you need does not have a unique ID? The code that follows shows a way to handle that:

```
$btn = $ie.Document.getElementsByTagName('A') | Where-Object {$_.innerText -eq 'Do not
include the US Minor Outlying Islands '}
$btn.Click()
```

Due to different browser settings, the behavior when a file is downloaded may be slightly different. On my browser, a pop-up dialog appears at the bottom of the screen with various options. If your browser behaves differently, you may need to play with the line of code here to get the script to work for you:

```
Select-Window iexplore | Set-WindowActive | Send-Keys "%S"
```

`Send-Keys` triggers key strokes to the window. The '%' is used to represent the `Alt` key, i.e., `Alt + S` is being pressed. You can play with these if the script does not work right at this point.

Here we find the elements with a `TagName = "A"` and then pipe them into a `Where-Object` cmdlet, which filters on the element with the `innerText = 'Do not include the US Minor Outlying Islands'`. An important point here is that sometimes more than one element meets the initial criteria, so remember that you can pipe the results into a `Where-Object` cmdlet and filter on specific attributes of the collection returned. The challenge in IE automation is getting to the element you need.

Finally, we get to the transaction, as shown here:

```
Start-Transaction -RollbackPreference TerminatingError

$sourcepath =  "c:" + $env:HOMEPATH + "\downloads\state_table.csv"
$targetpath =  "c:" + $env:HOMEPATH + "\Documents\state_table.csv"

Copy-Item $sourcepath $targetpath  -UseTransaction

$CheckFile = Import-CSV $targetpath

If ($CheckFile.Count -lt 50) { Write-host "Error"; Undo-Transaction } else { "Transaction
Committed" ; Complete-Transaction }
```

`Start-Transaction` kicks off the transaction; it has an interesting parameter `RollbackPreference`, which tells PowerShell under what error conditions, if any, the transaction should automatically be rolled back. Then we just set the source and destination path variables. The `Copy-Item` cmdlet call includes the `UseTransaction` parameter, which we use to tell PowerShell we want this action included in the transaction. Then we load the newly copied file into `$CheckFile`. Finally, we test that the new file has at least 50 rows, and if it does not we roll back the transaction using the `Undo-Transaction` cmdlet. Otherwise, we commit the transaction using the `Complete-Transaction` cmdlet.

Why Doesn't It Work?

PowerShell transactions sound good, but there is one problem: They don't work! Instead, PowerShell complains '`The provider does not support transactions. Perform the operation again without the -UseTransaction parameter`'. You might think you are doing something wrong. After all, the documentation clearly shows `UseTransaction` as a valid parameter not only for `Copy-Item` but for many cmdlets. It turns out that the only context in which transactions are supported is in making changes to the Windows registry. This is not mentioned anywhere in the standard Microsoft documentation, but I confirmed it on some blogs and in conferring with others. Although Microsoft may decide to expand on transaction functionality, it seems unlikely since it was added in version 2.0 with no subsequent enhancements. The bottom line is that if you need to edit registry settings, you may want to use transactions. Otherwise, there's not much here of value to a developer. Try running the statement that follows as another way to verify this:

```
Get-Psprovider | where {$_.Capabilities -like "*transactions*"}
```

Sure enough, only the registry will come back as supported.

I focused a lot on this to save you wasted time trying to use PowerShell transactions. Database developers might be attracted to them because they are used to SQL Server transactions and the ability to commit and rollback changes. The problem is not so much the limited implementation as the lack of documentation on these limitations.

However, we can code in a way that precludes the need for transactions. For example, the previous transaction related code can be rewritten as shown below which would provide the same benefit without the need for transactions. Even on SQL Server, you need to consider the overhead and potential locking issues explicit transactions can generate.

```
$CheckFile = Import-CSV $sourcepath

If ($CheckFile.Count -lt 50) { Write-host "Error"; Undo-Transaction } else
{ "Transaction Committed" ; Copy-Item $sourcepath $targetpath  }
```

We just check the downloaded file before we copy over the file we use for our application.

■ **Warning!** The only context in which transactions are supported is in making changes to the Windows registry.

Setting the Integrated Script Editor (ISE) Options

The ISE provides a number of options that control the behavior of the editor, and we can access these settings by selecting Tools from the drop-down menu and then selecting Options, as shown in Figure 4-5. It is worth taking the time to become familiar with these options, as we can access a number of time-saving features by selecting the right settings.

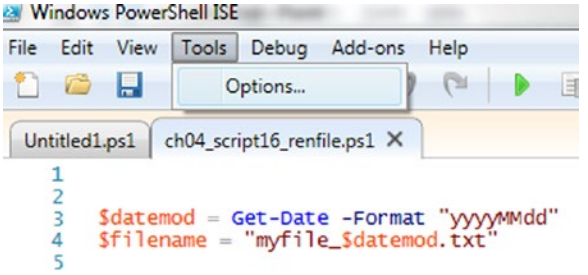

Figure 4-5. *Accessing the PowerShell ISE Tools options menu*

The first tab, shown in Figure 4-6, Colors and Fonts, lets us control the ISE color. Select the General tab to view the scripting settings as shown in Figure 4-6.

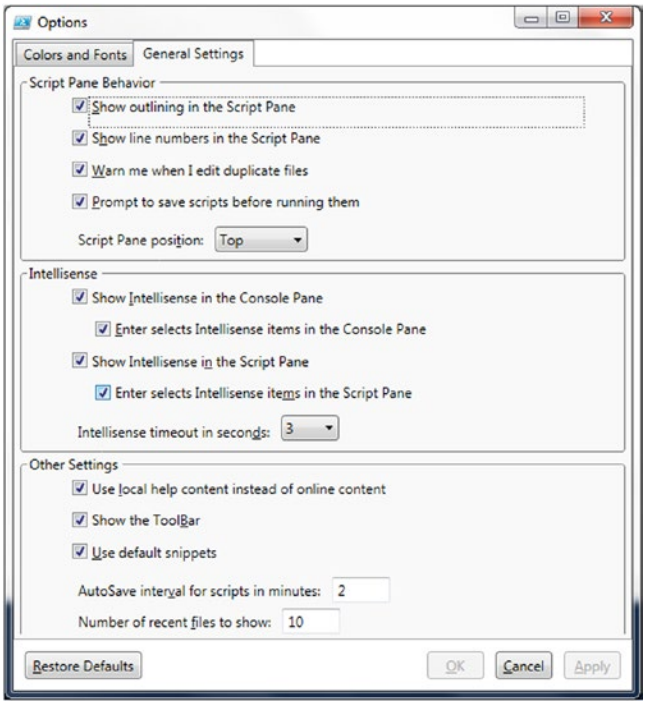

Figure 4-6. *The PowerShell ISE tool options General Settings panel*

Let's review what the options in Figure 4-6 do.

Show Outlining in the Script Pane enables the ability to hide blocks of code to make it easier to view large scripts. Figure 4-7 shows these feature enabled and the function's code hidden.

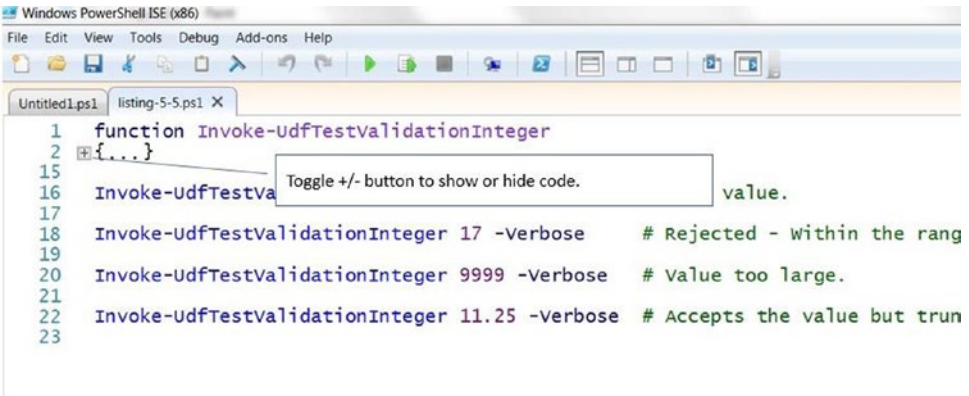

Figure 4-7. *The Script Editor in outline mode with function code hidden*

Clicking on the + will expand the code as shown in Figure 4-8.

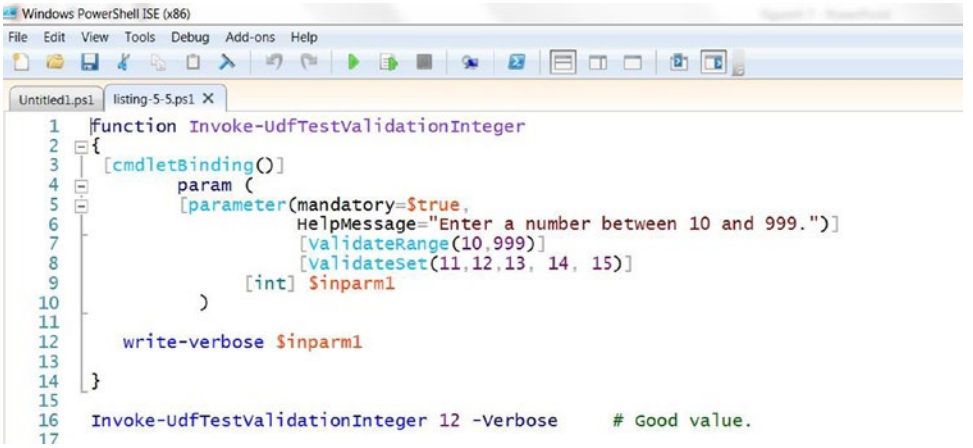

Figure 4-8. *The Script Edttor in outline mode with function code expande*

Show line numbers in the script pane will display line numbers in the left margin of the script when checked. When unchecked, line numbers are suppressed, as shown in Figure 4-9.

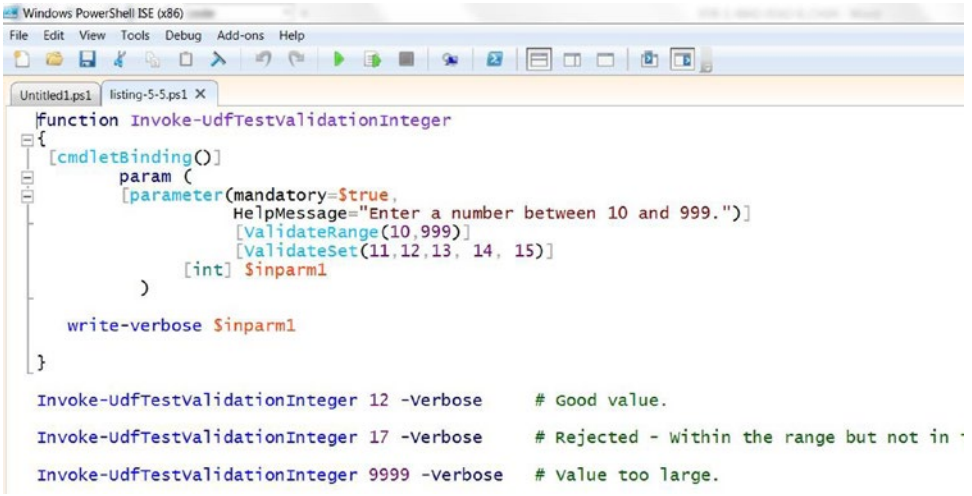

Figure 4-9. *The Script Editor with the line numbers option unchecked*

Warn me when I edit duplicate files is a feature to protect you from yourself. When enabled, if you open the same script in two different ISE tabs, you will be warned.

Prompt to save scripts before running them is another safety feature; after you make changes to a script that you have not saved, you will be prompted to do so when you try to run the script.

The **Script Pane position** drop-down combo box lets you control the placement of the script pane.

Show Intellisense in the Console Pane enables intellisense in the console window, which is very helpful in testing and development as you can try out cmdlets interactively. When enabled, you can get intellisense in the console, as shown in Figure 4-10.

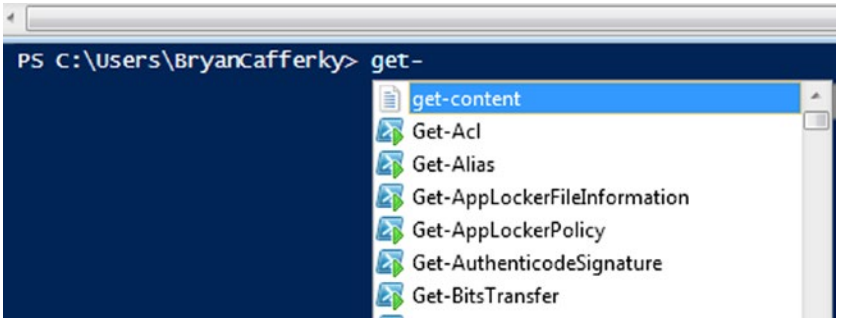

Figure 4-10. *The ISE's Console Pane with intellisense enabled for the console pane*

Enter selects Intellisense items in the Console Pane relates to intellisense being enabled in the Console Pane.

Show Intellisense in the Script Pane enables intellisense in the script window, which is very helpful in testing and development.

Enter selects Intellisense items in the Script Pane relates to intellisense being enabled in the script pane. If enabled, when intellisense pops up with a help list, pressing Enter on a selected item will copy it to the script at the current line.

Intellisense timeout in seconds sets the amount of time PowerShell will allow the ISE to load the required intellisense help. If it takes longer, no intellisense is provided, which can happen as the ISE has to look up the required information. If you have this problem, just increase the time allowed.

Use local help content instead of online content will get help information stored on your machine rather than getting it over the Internet.

Show the toolbar will display the ISE toolbar when checked.

Use default snippets enables the ability to bring up and insert a default set of snippets. Snippets are code fragments that you can insert into your scripts and that provide a starting point for some coding. The delivered snippets provide a skeleton set of code for many statements and cmdlets. If this feature is checked, you can bring up available snippets by right mouse clicking in the script pane and selecting snippets, as shown in Figure 4-11. You can also access snippets from the File drop-down menu.

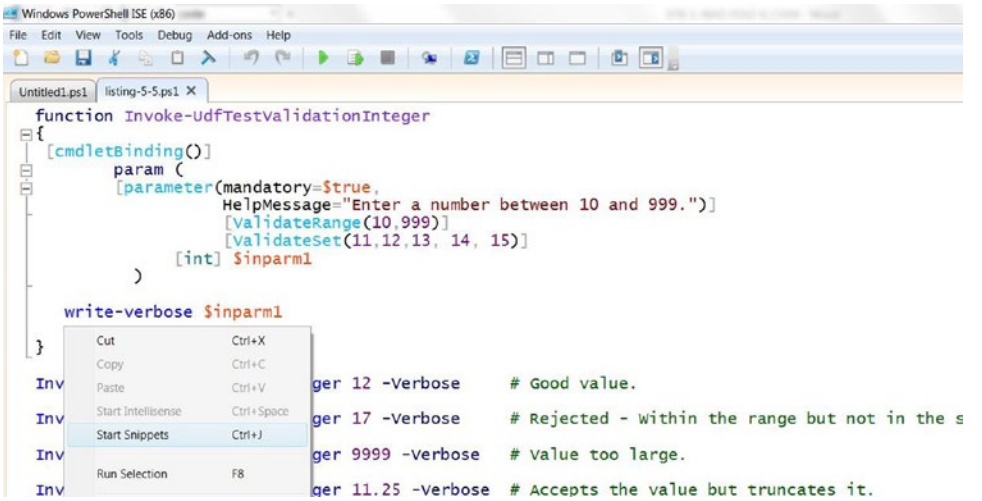

Figure 4-11. *Right mouse click in the script pane and select Start Snippets to bring up a pop up list of default snippets*

To select the desired snippet, select it from the scrolling snippet menu and press Enter, as shown in Figure 4-12.

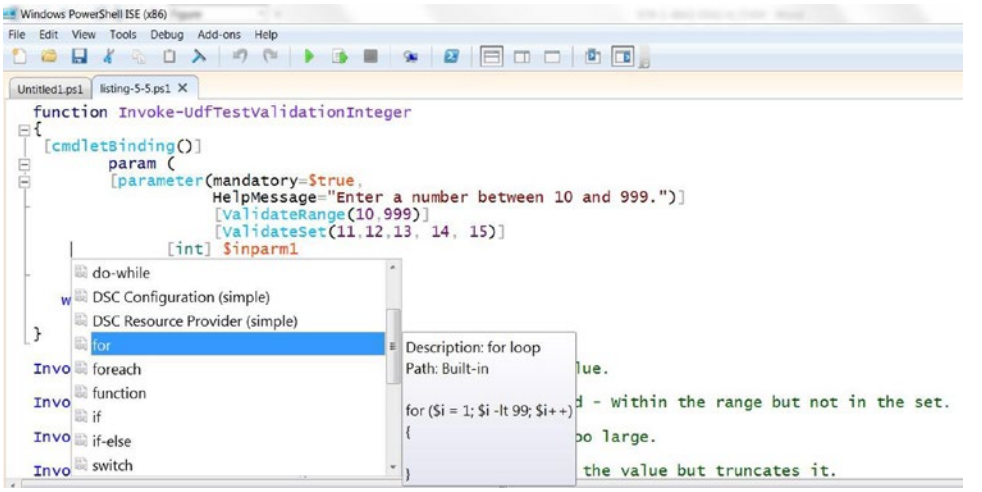

Figure 4-12. *The snippet selection menu*

AutoSave interval for scripts in minutes sets the amount of time the ISE waits after a change to save the code.

Number of recent files to show controls how many files are listed is the File drop-down menu.

Summary

We started the chapter by learning how to use `Set-StrictMode` to enforce coding rules that help avoid uninitialized variables, incorrect method calls, and invalid property references. Using `Set-StrictMode` during development can help avoid difficult-to-debug errors. Then, we had a detailed discussion on how to handle errors that get raised in scripts. This was followed by a review of the common cmdlet parameters, so called because all built-in cmdlets support them. From there, we touched briefly on PowerShell's debugging features. Then, we delved into how PowerShell can implement Windows forms to provide a GUI for our applications. This led into a discussion on how you can leverage .NET object events. We discussed PowerShell transactions and how they may not be as useful as we would have hoped. Finally, we discussed the PowerShell ISE options and how to use them.