CHAPTER 1

PowerShell Basics

In this chapter we will discuss what PowerShell is and do a quick review of its history. We'll describe the two PowerShell environments, which are the Command Line Interface (CLI) and the Integrated Script Environment (ISE). The PowerShell CLI is used for the immediate execution of commands entered at a prompt. The ISE is the PowerShell development environment and is intended for the creation and debugging of scripts. By default, PowerShell will not allow scripts to be executed. Therefore, we will discuss how to enable scripting. Then we will discuss using the ISE to write scripts. We'll review how we can customize the ISE settings to fit our needs. Then we'll run a script with a bug in it to see how PowerShell displays error messages.

What Is PowerShell?

If you have ever used Linux or any command-line-based operating system, then you have interacted with the system by entering commands at a prompt something like this one:

```
C:\
```

Likely, you sometimes stacked a bunch of commands together and stored them in a file where they could be executed as a script. In the same way, PowerShell is a command interface to Windows, and commands can be stored in a file and then executed as script.

For those who have only used Graphical User Interface (GUI) based operating systems, a little history is in order. Before Windows, the desktop operating system was called DOS (Disk Operating System), and you had to interact with the computer via a command prompt. For example, entering `dir` would cause the system to display the list of files in the current folder. Needless to say, when Microsoft released Windows people were happy to leave the command line behind and click their way to happiness. In those days, the environment was simple, so doing everything, even systems administration, via a GUI was fine. But the world has changed, and now there are many resources to manage: Active Directory, Internet Information Services, SharePoint, SQL Server, Outlook, and the list goes on. As a result, doing administration by clicking and dragging is not very effective. Imagine that you need to reassign permissions on a thousand users by clicking on each one and navigating a series of GUI screens. Suddenly, the ability to create and execute command scripts is appealing. Technically, Windows always had the command line (CMD.EXE), but this feature was based on the ancient DOS technology. The commands are cryptic, and it is difficult to write and maintain complex scripts.

Microsoft saw the need for an effective scripting tool a long time ago, and their first product in this area was called Windows Script Host in 1998. However, this tool fell short of the job, and in 2006 Microsoft came out with PowerShell 1.0. It has gone through many enhancements, and currently the latest production version is 4.0 with 5.0 in prerelease.

Why Is PowerShell Important?

At first glance, we might think, so what? What's the big deal about another scripting language? The answer is that PowerShell is a lot more than just a Windows version of Linux scripting. First, it is very extensible, thus allowing users and Microsoft to add functionality very quickly. Second, assuming we install the associated modules, PowerShell can easily connect to and manage virtually any resource available to Windows, which includes SQL Server, Active Directory, IIS, SharePoint, the file system, and so forth, as shown in Figure 1-1. Third, PowerShell is completely object based, meaning all variables are objects—as is anything returned by a PowerShell command. Not only that, but PowerShell seamlessly enables us to use the latest .NET framework and COM object libraries. Technically, PowerShell is not object oriented, because it does not support creating classes or inheritance, nor does it support polymorphism. However, I will cover these limitations and discuss some effective workarounds. Note: PowerShell 5.0 is adding these features.

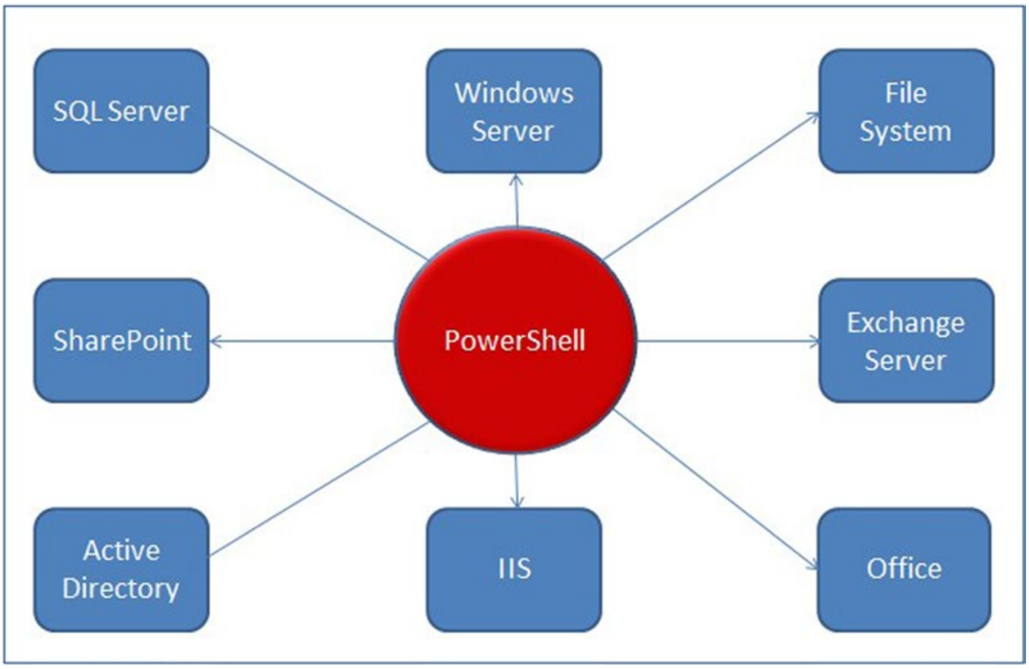


Figure 1-1. *Diagram showing how PowerShell fits into the Windows Environment*

PowerShell is architected to execute scripts either locally or remotely, provided the remote machines have been configured for remote execution. This means that we can run a script on one machine that executes scripts on any number of remote machines. Imagine deploying an upgrade to a hundred SQL Servers simultaneously or building a number of virtual machines. If you ever wondered how a large and complex environment like Azure is administered, PowerShell is the answer. PowerShell provides a high degree of scalability. Figure 1-2 shows that PowerShell can query remote machines, restart computers, and invoke commands on remote machines—all from a single PowerShell session. No wonder it is the main tool for system administrators.

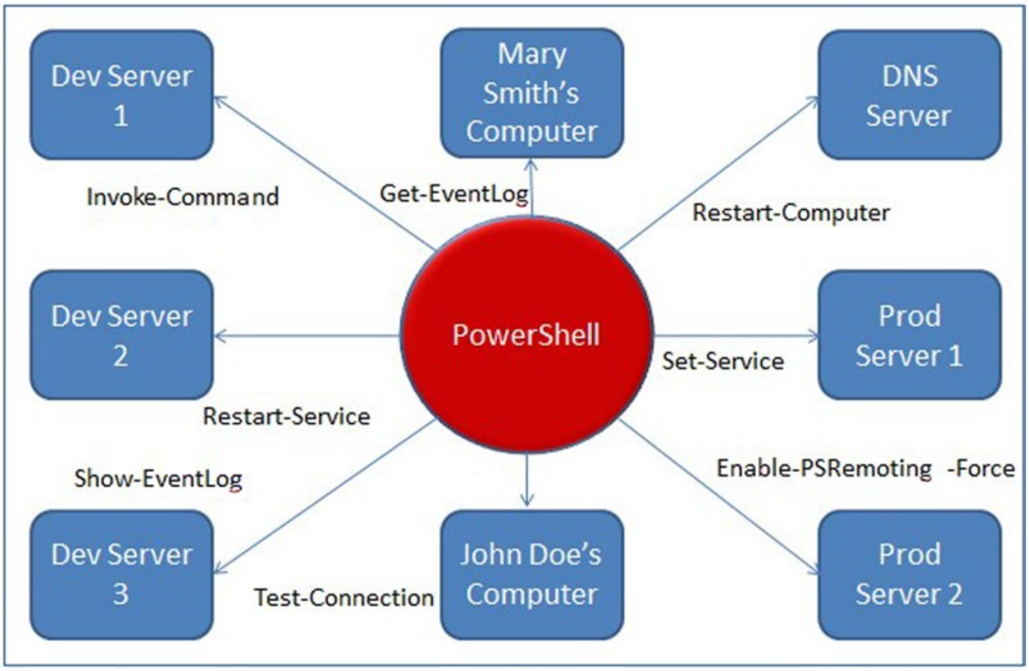


Figure 1-2. *Diagram showing that PowerShell can execute commands both locally and remotely*

History of PowerShell Versions

PowerShell was first released in 2006 and is now on version 4.0 with 5.0 in prerelease. Each release of PowerShell typically corresponds to a .NET framework version, as PowerShell is integrated with the .NET framework. Table 1-1 provides a review of each PowerShell version.

Table 1-1. *PowerShell Version History*

Version	Release Year	Description
1.0	2006	Initial release
2.0	2008	Many revisions and extensions, including support for remote execution, background execution, modules, ability to write cmdlets, and the addition of the Integrated Scripting Environment (ISE)
3.0	2011	Added new cmdlets and support for scheduled jobs, intellisense in the ISE, and delegation of tasks
4.0	2014	Addition of desired state configuration to support deployment and management of configuration data for systems, save help, and enhanced debugging; default execution policy changed to RemoteSigned
5.0	2015	Added ability to create classes with support for inheritance, Windows Management Framework 5.0 (WMF5) and extension of desired state configuration, an advanced set of features to support the automated installation of software packages.

Starting PowerShell

We can run PowerShell in two modes: the Command Line Interface (CLI) or the Integrated Scripting Environment (ISE). The CLI is a command mode in which you enter one command line at a time to be processed. It is interactive and meant as an alternative to the Windows Graphical User Interface (GUI) to get work done. If you have used the Windows command prompt, know that the PowerShell CLI is a much more powerful alternative. The PowerShell ISE, however, is a development studio for writing PowerShell code. It is provided so you can store a series of commands—called a *script*—to be executed as a program. While the CLI is focused on one-time tasks, i.e., get in, copy a file, and get out, the ISE is focused on writing a series of statements that will need to be executed repeatedly.

To start in either mode, click on the Windows Start menu, then select All Programs ➤ Accessories ➤ Windows PowerShell. Depending on your computer, you may see several versions of PowerShell, as shown in Figure 1-3.

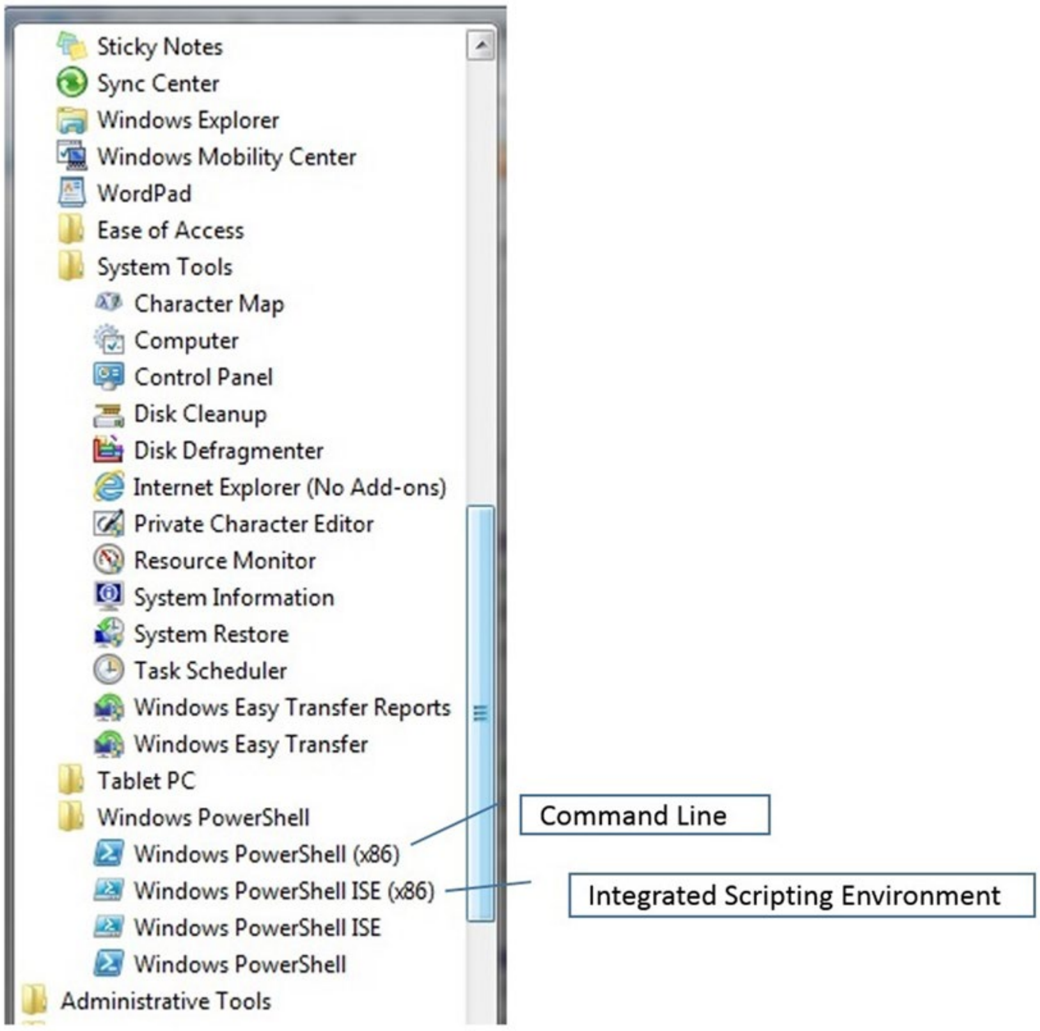
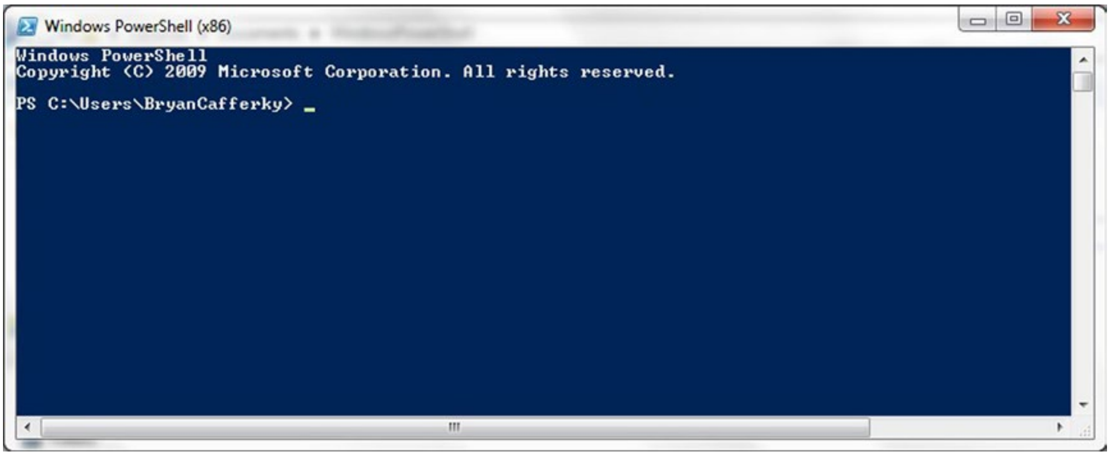


Figure 1-3. *Starting PowerShell*

The programs with (x86) next to them are 32-bit versions, while the programs without the (x86) are 64-bit versions. If your computer is 64 bit, you probably want to use the 64-bit versions, which, in theory, should run faster. However, if the PowerShell code requires it, you may need to use the 32-bit version. A good example of this is when you need to use 32-bit drivers. If you only see the (x86) versions, then you have a 32-bit machine.

The Command Line Interface (CLI)

To start the CLI, click on the program Windows PowerShell, or if your machine is 32 bit, click Windows PowerShell (x86). You will see a screen similar to that shown in Figure 1-4. This is the command line interface, because it only accepts one command line at a time. Note: It is possible to stack statements on a line by separating each with a semicolon which is the statement terminator.

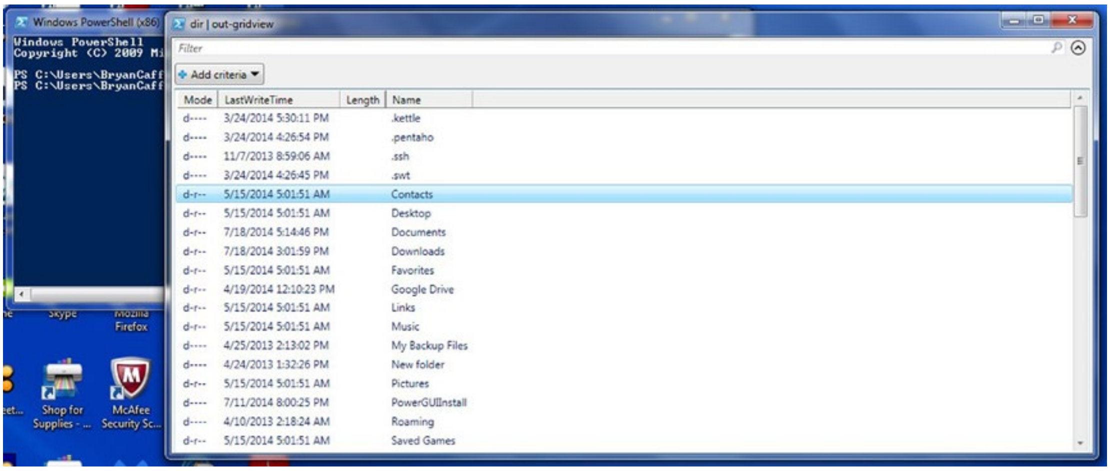


```
Windows PowerShell
Copyright (C) 2009 Microsoft Corporation. All rights reserved.

PS C:\Users\BryanCafferky> _
```

Figure 1-4. The 32-bit PowerShell Command Line Interface (CLI)

Using the CLI, enter your first PowerShell command. Enter the line below and you should see the result similar to that shown in Figure 1-5.

```
DIR | Out-Gridview
```



```
Windows PowerShell
Copyright (C) 2009 Mi

PS C:\Users\BryanCaff
PS C:\Users\BryanCaff
```

Mode	LastWriteTime	Length	Name
d----	3/24/2014 5:30:11 PM		.kettle
d----	3/24/2014 4:26:54 PM		.pentaho
d----	11/7/2013 8:59:06 AM		.ssh
d----	3/24/2014 4:26:45 PM		.swt
d-r--	5/15/2014 5:01:51 AM		Contacts
d-r--	5/15/2014 5:01:51 AM		Desktop
d-r--	7/18/2014 5:14:46 PM		Documents
d-r--	7/18/2014 3:01:59 PM		Downloads
d-r--	5/15/2014 5:01:51 AM		Favorites
d-r--	4/19/2014 12:10:23 PM		Google Drive
d-r--	5/15/2014 5:01:51 AM		Links
d-r--	5/15/2014 5:01:51 AM		Music
d----	4/25/2013 2:13:02 PM		My Backup Files
d----	4/24/2013 1:32:26 PM		New folder
d-r--	5/15/2014 5:01:51 AM		Pictures
d----	7/11/2014 8:00:25 PM		PowerGUIInstall
d----	4/10/2013 2:18:24 AM		Roaming
d-r--	5/15/2014 5:01:51 AM		Saved Games

Figure 1-5. *File list displayed in PowerShell's Gridview*

Congratulations! You've entered your first PowerShell command. We added the `| Out-Gridview` to the command because that causes the output of the `dir` command to be displayed in the grid. You can click on different column headings to sort by that column, and you can even enter filter criteria. A nice feature of PowerShell is that you can pipe the output of one command into the next command, which is what was done here. Stacking commands like this, called *piping*, is a key feature that we will use often. You can even write PowerShell scripts that process the data piped into them. Another thing to note here is that `DIR` is known as an alias. An *alias* is a command-name mapping. In this case `dir` is an alias for `Get-ChildItem`. In the old DOS world, `DIR` was the command to list the files in a folder. PowerShell has a lot of aliased commands to make it easier to learn. Many old DOS and Linux commands are aliased to equivalent PowerShell commands. You can create your own aliases to help you in your work. However, while aliases are great for use in the CLI, I don't recommend their use in scripts that are to be used by others because it is not obvious what they mean.

There are two commands that are particularly useful to PowerShell newcomers, which are `Get-Command` and `Get-Help`. Let's see how they work.

Enter the following:

```
Get-Command
```

You will get a list of all the PowerShell commands. You can filter on the command name using wild cards, as shown below:

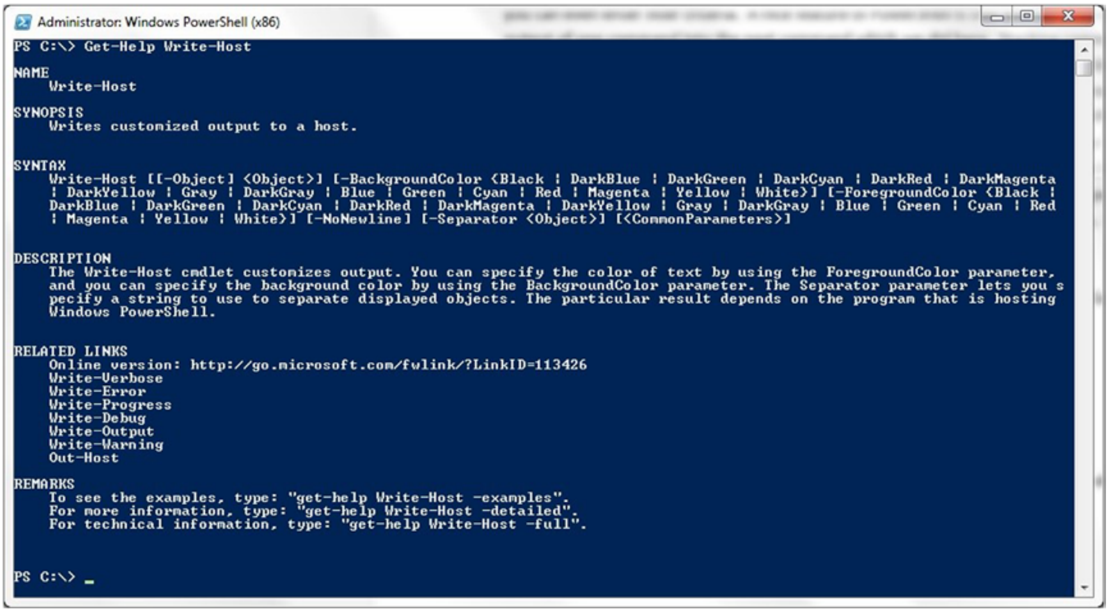
```
Get-Command *win*
```

This will display a list of all commands with the string "`win`" in the name.

If you want to know more about a specific command, such as `Write-Host`, you can enter:

```
Get-Help Write-Host
```

And you will get a screen full of information about the command, similar to Figure 1-6.



```
PS C:\> Get-Help Write-Host

NAME
    Write-Host

SYNOPSIS
    Writes customized output to a host.


SYNTAX
    Write-Host [[-Object] <Object>] [-BackgroundColor <Black | DarkBlue | DarkGreen | DarkCyan | DarkRed | DarkMagenta
    | DarkYellow | Gray | DarkGray | Blue | Green | Cyan | Red | Magenta | Yellow | White>] [-ForegroundColor <Black |
    DarkBlue | DarkGreen | DarkCyan | DarkRed | DarkMagenta | DarkYellow | Gray | DarkGray | Blue | Green | Cyan | Red
    | Magenta | Yellow | White>] [-NoNewline] [-Separator <Object>] [<CommonParameters>]


DESCRIPTION
    The Write-Host cmdlet customizes output. You can specify the color of text by using the ForegroundColor parameter,
    and you can specify the background color by using the BackgroundColor parameter. The Separator parameter lets you s
    pecify a string to use to separate displayed objects. The particular result depends on the program that is hosting
    Windows PowerShell.


RELATED LINKS
    Online version: http://go.microsoft.com/fwlink/?LinkID=113426
    Write-Verbose
    Write-Error
    Write-Progress
    Write-Debug
    Write-Output
    Write-Warning
    Out-Host

REMARKS
    To see the examples, type: "get-help Write-Host -examples".
    For more information, type: "get-help Write-Host -detailed".
    For technical information, type: "get-help Write-Host -full".


PS C:\> _
```

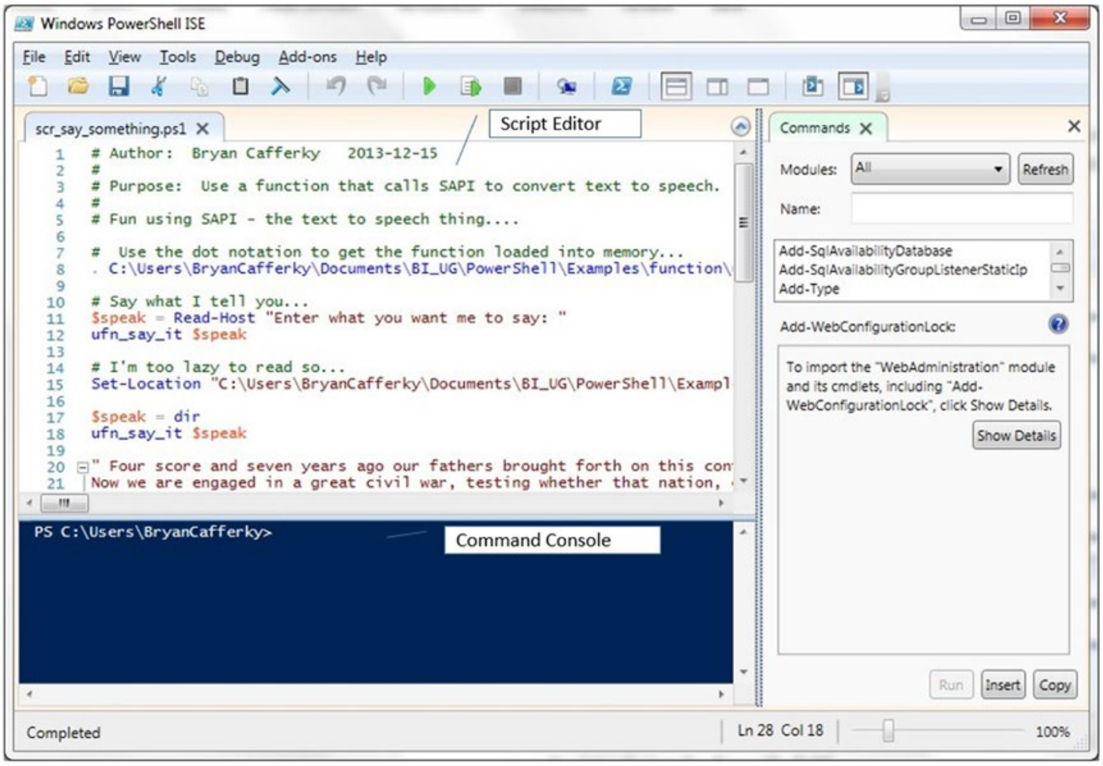
Figure 1-6. *The PowerShell console showing output of Get-Help*

Later, we will discuss how we can get PowerShell to display similar information about our scripts via the `Get-Help` command.

The CLI is very useful in working in the Windows environment, but let's shift gears and focus on developing scripts using the Integrated Scripting Environment (ISE). To exit the CLI, just close the CLI window.

The Integrated Scripting Environment (ISE)

Start the PowerShell ISE by clicking on the Windows Start menu ➤ All Programs ➤ Accessories ➤ Windows PowerShell ISE (or Windows PowerShell ISE (x86) if you are running a 32-bit machine). You should see a screen similar to that shown in Figure 1-7 less the code in the script editor window.



```
# Author:  Bryan Cafferky   2013-12-15
#
# Purpose:  Use a function that calls SAPI to convert text to speech.
#
# Fun using SAPI - the text to speech thing....

#  Use the dot notation to get the function loaded into memory...
. C:\Users\BryanCafferky\Documents\BI_UG\PowerShell\Examples\function\

# Say what I tell you...
$speak = Read-Host "Enter what you want me to say: "
ufn_say_it $speak

# I'm too lazy to read so...
Set-Location "C:\Users\BryanCafferky\Documents\BI_UG\PowerShell\Exampl

$speak = dir
ufn_say_it $speak

" Four score and seven years ago our fathers brought forth on this con
Now we are engaged in a great civil war, testing whether that nation,
```

```
PS C:\Users\BryanCafferky>
```

To import the "WebAdministration" module and its cmdlets, including "Add-WebConfigurationLock", click Show Details.

Figure 1-7. *The PowerShell Integrated Scripting Environment (ISE)*

If you have a 64-bit machine, you may think you should always just run the 64-bit version of PowerShell, i.e., the one not suffixed with x86. However, there are times when you MUST use the 32-bit version, such as when you need to use 32-bit drivers as required by Microsoft Office products such as Access. If you have trouble getting a script to run under one version, try running it under the other version.

Enabling Scripting

By default, PowerShell will not let you run scripts at all. Note: The Microsoft documentation says the exception to this is Windows Server 2012 R2 which defaults to allow local scripting. You can enable scripting by using the `Set-Execution-Policy` command. However, to do this you need to start PowerShell as an administrator. You can start the PowerShell CLI as administrator by clicking the Start Menu ➤ All Programs ➤ Accessories ➤ Windows PowerShell and right mouse-clicking on either Windows PowerShell or Windows PowerShell (x86), whichever applies to your machine, and selecting *Run as administrator* as shown in Figure 1-8.

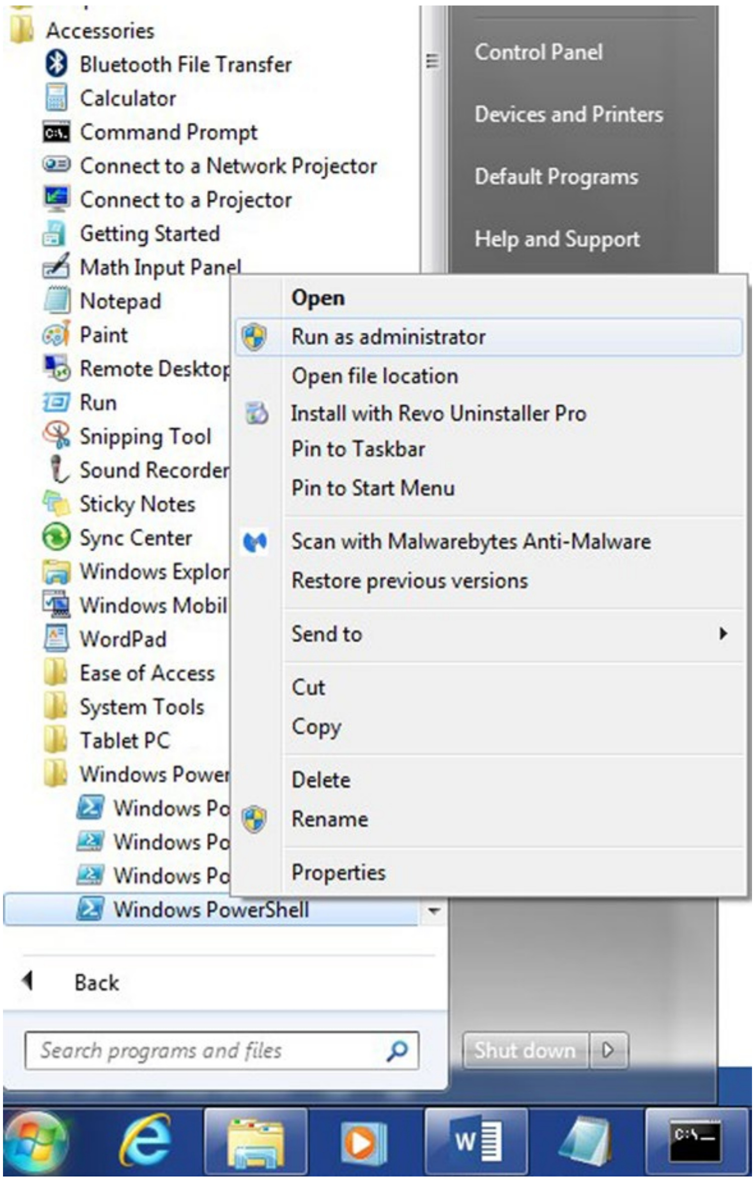
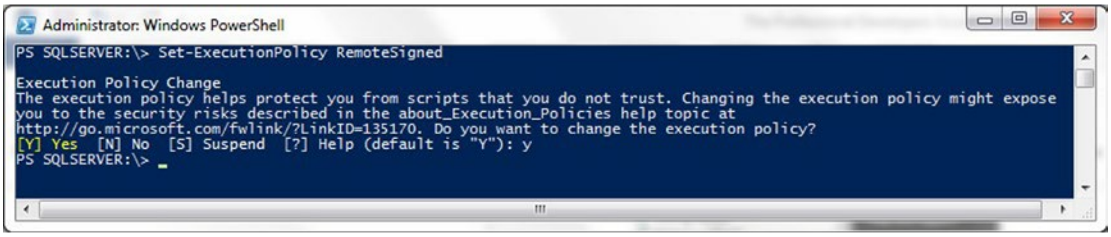


Figure 1-8. *Starting PowerShell as an administrator*

The PowerShell CLI takes one command line at a time. You want to enable scripting so you can execute lists of commands that are stored in files. To enable scripting, you need to use the command `Set-ExecutionPolicy`, as shown here:

```
Set-ExecutionPolicy RemoteSigned
```

When you enter this statement, you should be prompted, as shown in Figure 1-9:



```
PS SQLSERVER:\> Set-ExecutionPolicy RemoteSigned

Execution Policy Change
The execution policy helps protect you from scripts that you do not trust. Changing the execution policy might expose
you to the security risks described in the about_Execution_Policies help topic at
http://go.microsoft.com/fwlink/?LinkID=135170. Do you want to change the execution policy?
[Y] Yes  [N] No  [S] Suspend  [?] Help (default is "Y"): y
PS SQLSERVER:\> _
```

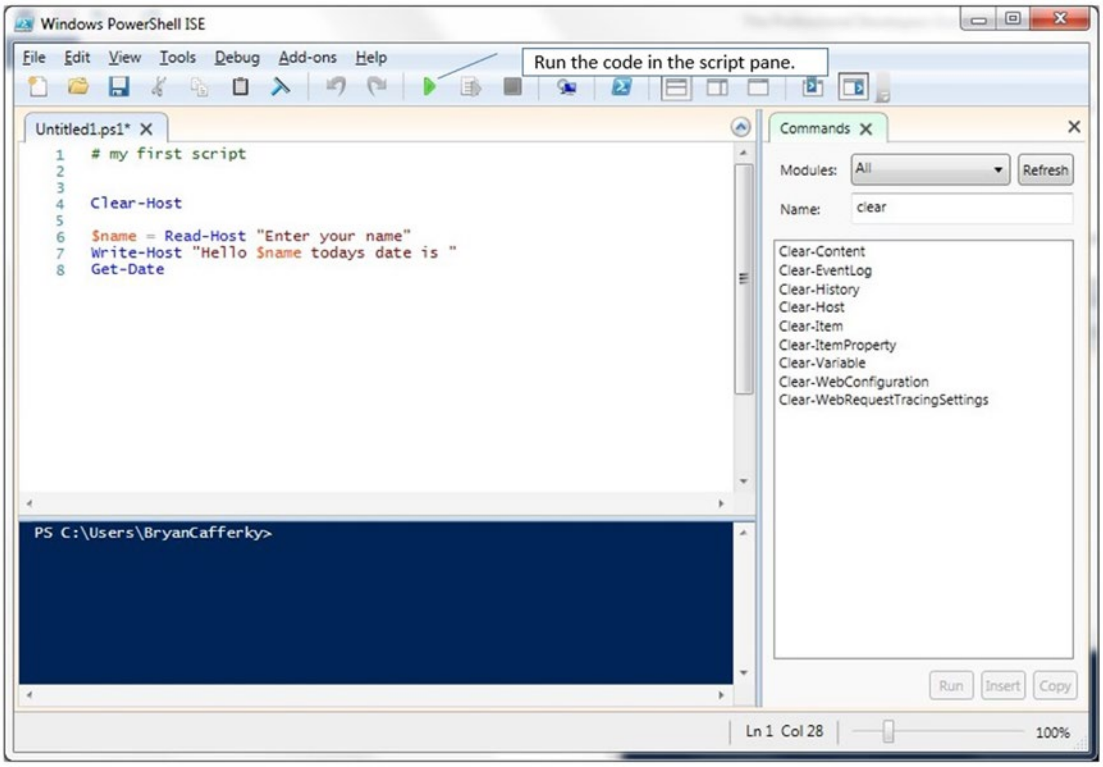
Figure 1-9. *Setting the execution policy to RemoteSigned*

Enter `'Y'` at the prompt to enable scripting. The execution policy controls the modes in which PowerShell scripting is allowed. By setting it to `RemoteSigned` you are allowing unsigned scripts to be executed locally only, i.e., not by a remote machine. Getting scripts signed is a way of guaranteeing the script was written by who it is claimed to have been written by. This is particularly an issue if you bring in scripts from the outside, such as third-party modules. Signed scripts have a related certificate that guarantees the code has not been tampered with.

Exit the PowerShell CLI by closing the window and go back to the PowerShell ISE. Now that scripting is enabled you don't need to run the PowerShell ISE as an administrator.

Using the Integrated Scripting Environment

If it is not running, start the ISE. See Figure 1-3 for help. Enter your PowerShell script into the script pane as shown in Figure 1-10. The toolbar provides easy access to some common functions, such as the green arrow that will execute the script when clicked.

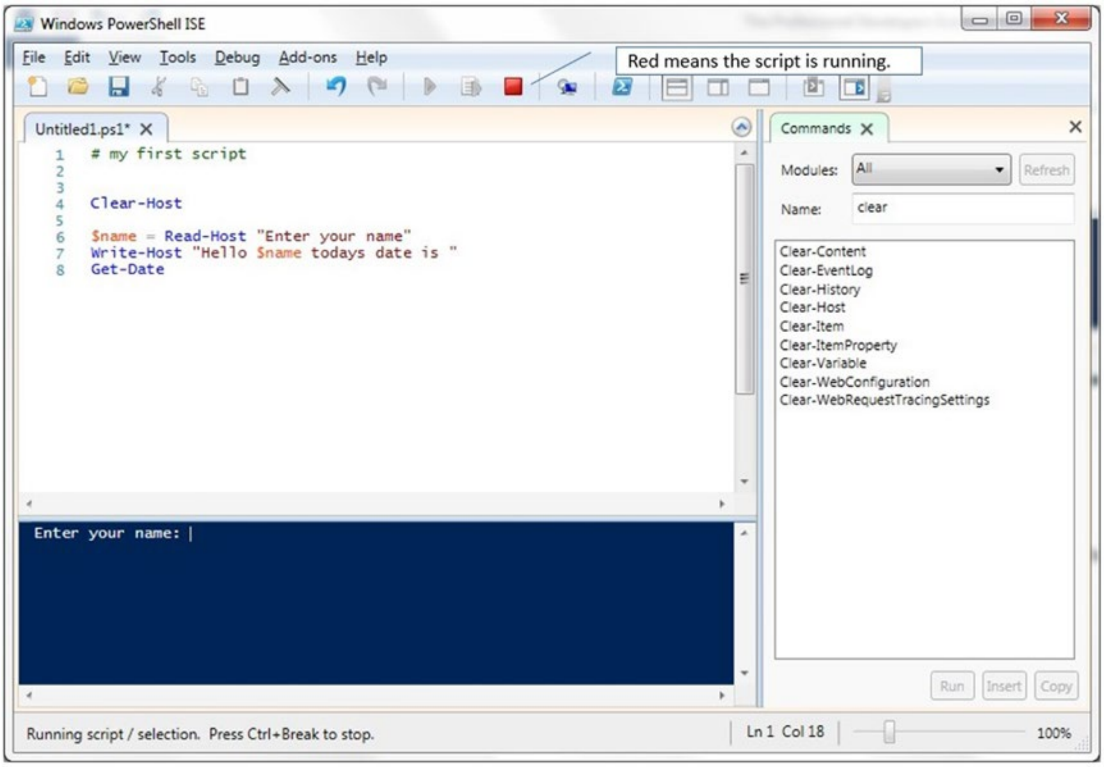


```
# my first script


Clear-Host

$name = Read-Host "Enter your name"
Write-Host "Hello $name todays date is "
Get-Date
```

```
PS C:\Users\BryanCafferky>
```

Figure 1-10. *The PowerShell ISE — Running the script*

Now let’s write your first script. In the script pane, enter the script shown in Figure 1-10 and click on the green arrow which will run the script. Notice that the gray box in the toolbar turned red, as shown in Figure 1-11.



```
# my first script


Clear-Host

$name = Read-Host "Enter your name"
Write-Host "Hello $name todays date is "
Get-Date
```

```
Enter your name: |
```

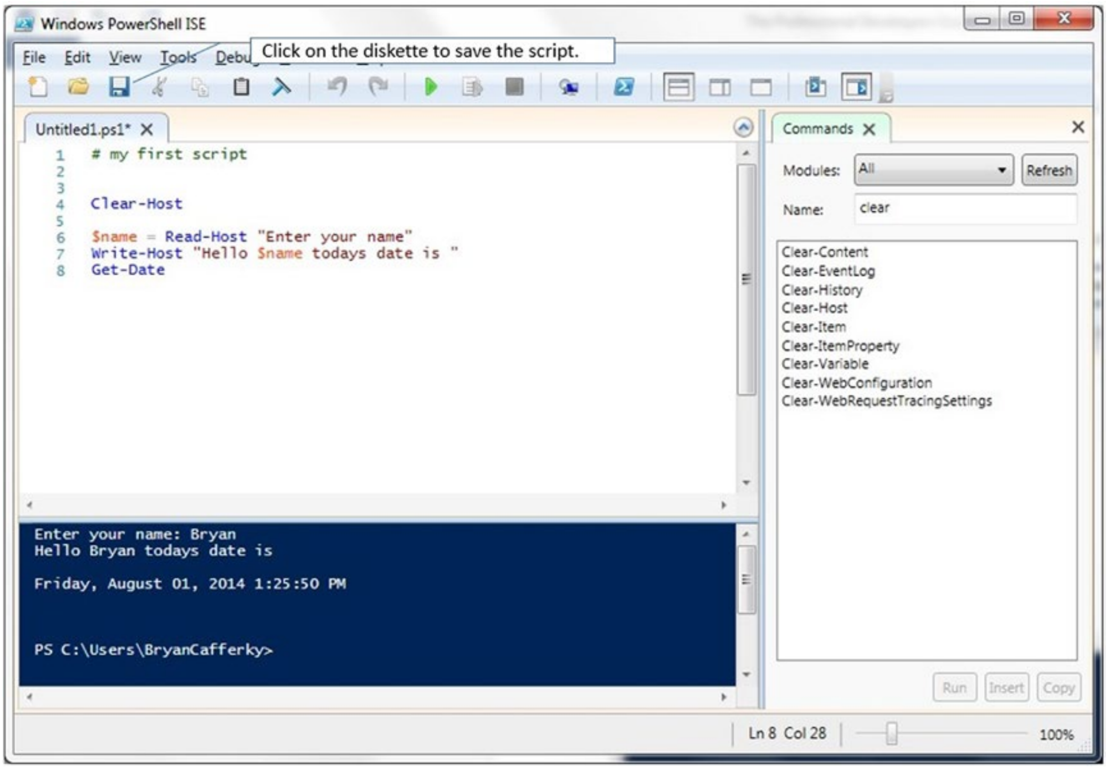
```
Clear-Content
Clear-EventLog
Clear-History
Clear-Host
Clear-Item
Clear-ItemProperty
Clear-Variable
Clear-WebConfiguration
Clear-WebRequestTracingSettings
```

Running script / selection. Press Ctrl+Break to stop. Ln 1 Col 18 100%

Figure 1-11. *The PowerShell ISE —A running script*

The red box indicates that the script is running. Remember this, because sometimes you may not realize your script is still executing when you are testing it.

Enter your name when prompted in the command pane. You should see a message come back greeting you and telling you the current date and time, as shown in Figure 1-12.



```
# my first script


Clear-Host

$name = Read-Host "Enter your name"
Write-Host "Hello $name todays date is "
Get-Date
```

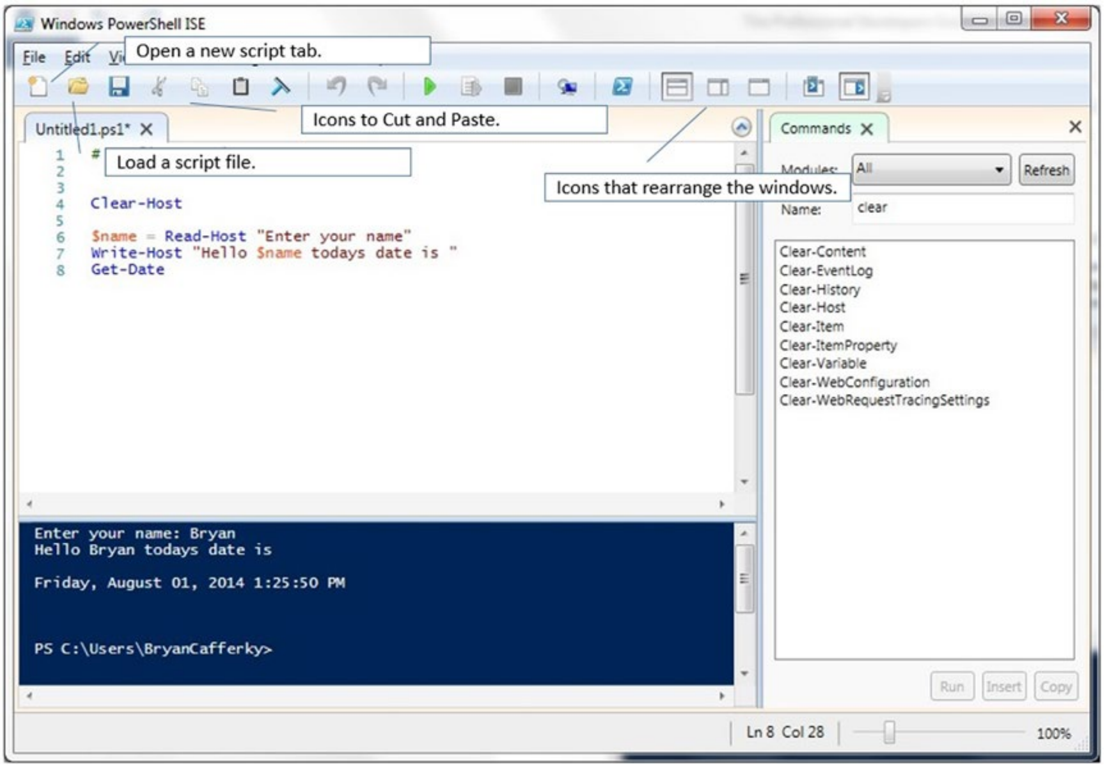
```
Enter your name: Bryan
Hello Bryan todays date is

Friday, August 01, 2014 1:25:50 PM



PS C:\Users\BryanCafferky>
```

```
Clear-Content
Clear-EventLog
Clear-History
Clear-Host
Clear-Item
Clear-ItemProperty
Clear-Variable
Clear-WebConfiguration
Clear-WebRequestTracingSettings
```

Figure 1-12. *The PowerShell ISE —Saving the script*

Congratulations! You’ve written your first PowerShell script! Now to save it, click on the diskette icon in the toolbar. This might be a good time to create a subfolder named WindowsPowerShell in your Documents folder if you don’t already have one. WindowsPowerShell is the default place PowerShell looks for a profile script, which is a special script we will discuss later.

Let’s look at the ISE screen more closely. The toolbar provides easy access to the most commonly used ISE functions. Note: Hovering the mouse over an icon will cause a short description to appear. Figure 1-13 shows what the tollbar buttons do. The toolbar is handy, but most toolbar buttons have equivalent menu bar item options as well. Some actions can also be performed using function keys.



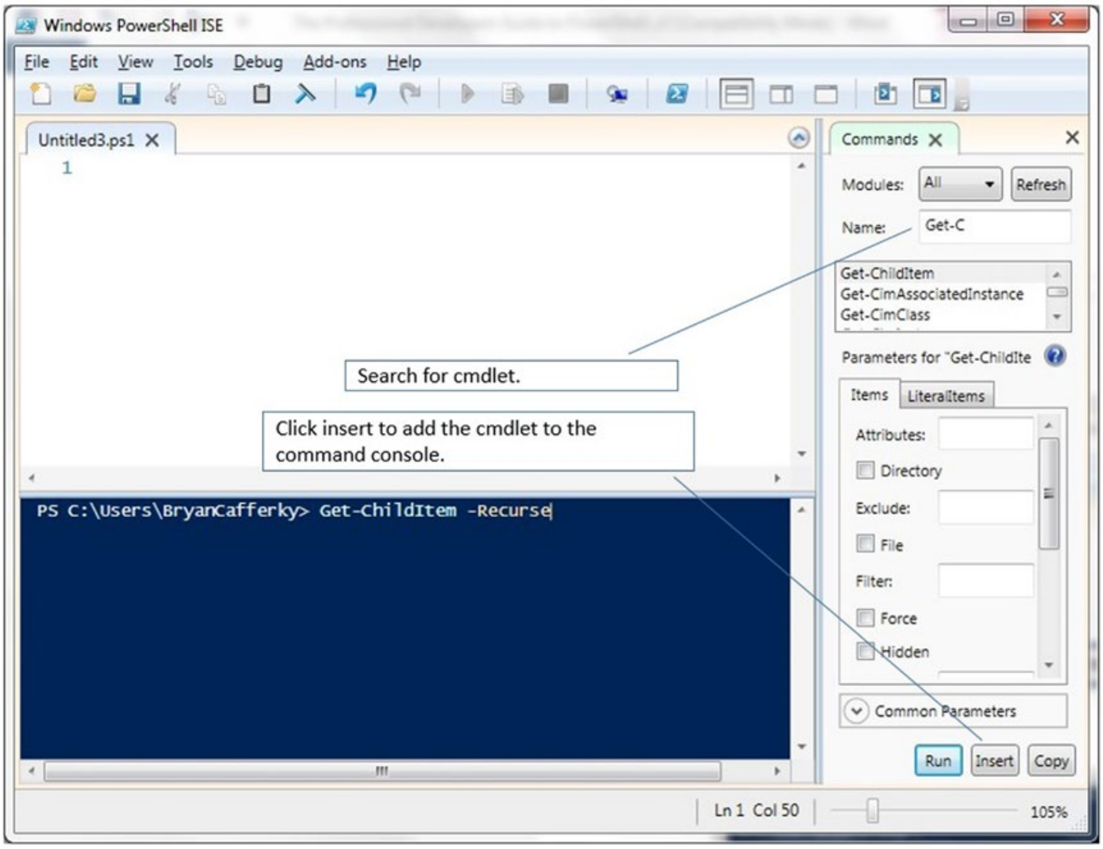
```
#

Clear-Host

$name = Read-Host "Enter your name"
Write-Host "Hello $name todays date is "
Get-Date
```

```
Enter your name: Bryan
Hello Bryan todays date is

Friday, August 01, 2014 1:25:50 PM


PS C:\Users\BryanCafferky>
```

```
Clear-Content
Clear-EventLog
Clear-History
Clear-Host
Clear-Item
Clear-ItemProperty
Clear-Variable
Clear-WebConfiguration
Clear-WebRequestTracingSettings
```

Figure 1-13. *The PowerShell ISE — Running the script*

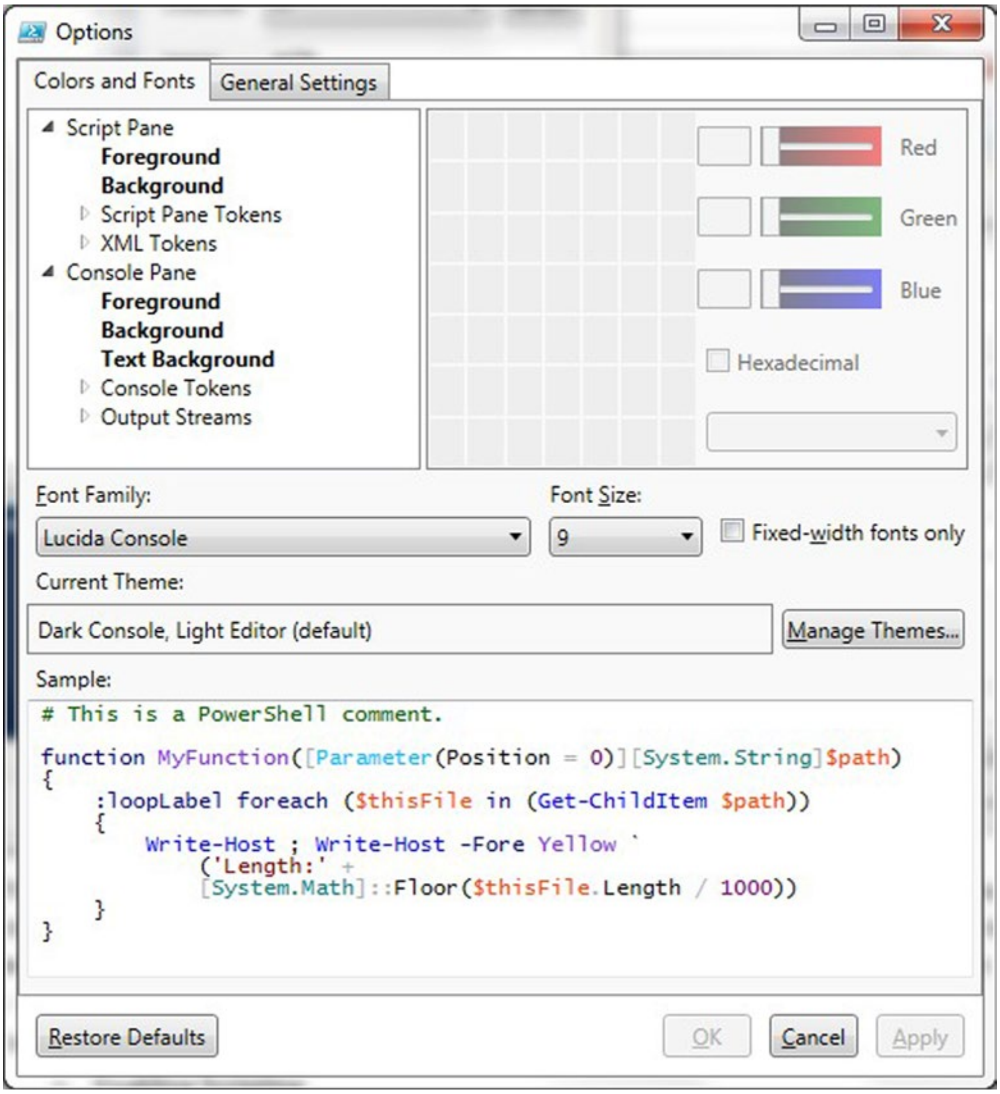
Notice the tab on the right in Figure 1-14 labeled *Commands*. If you don't see it, select the View menu and select "Show Command Add-on." This feature is especially useful to newcomers because it provides a simple wizard-like tool to write PowerShell statements. For example, click in the Name text box and type `Get-C` as shown in Figure 1-14. Notice that the list filters down to commands that start with those letters. Now double click on `Get-ChildItem`, and the tab displays the possible parameters. Fill in some values and click the Insert button. The command is written to the Command pane, where you can execute it or copy and paste it into the Script Pane. This is a fun way to explore PowerShell commands.



```
PS C:\Users\BryanCafferky> Get-ChildItem -Recurse
```

Figure 1-14. *The PowerShell ISE — The Commands add-on*

Most of the menus and options follow standard Windows conventions and are easy to learn. The File menu has options to load and save files. The Edit menu has typical editing options like Find and Replace. The View menu has some features you may not be used to including ones that will move the ISE panes around. I like the configuration above, but there are a number of possibilities. Clicking on the Tools menu bar and selecting *Options*. . . will display the Options screen as shown in Figure 1-15. Here you can set the ISE options to fit your preferences.



Font Family: Lucida Console

Font Size: 9

Fixed-width fonts only

Current Theme: Dark Console, Light Editor (default)

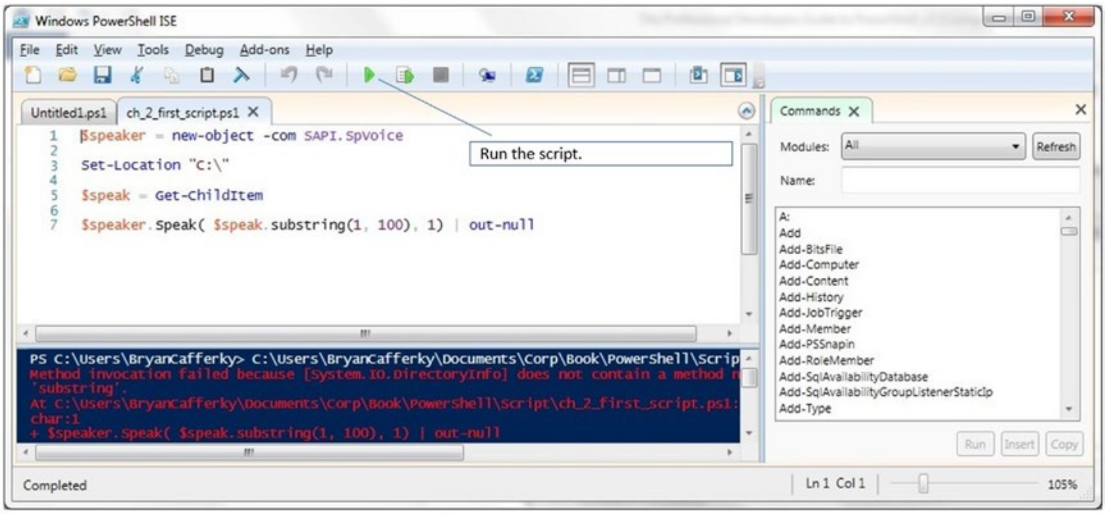
Sample:

```
# This is a PowerShell comment.

function MyFunction([Parameter(Position = 0)][System.String]$path)
{
    :loopLabel foreach ($thisFile in (Get-ChildItem $path))
    {
        Write-Host ; Write-Host -Fore Yellow `
            ('Length:' +
            [System.Math]::Floor($thisFile.Length / 1000))
    }
}
```

Figure 1-15. *The PowerShell ISE — Options*

Encountering Script Errors

Let's enter another script, but one with a bug. First, click the leftmost icon in the toolbar that looks like a page. Then enter the script shown in Figure 1-16.



```
$speaker = new-object -com SAPI.SpVoice

Set-Location "C:\"

$speak = Get-ChildItem

$speaker.Speak( $speak.substring(1, 100), 1) | out-null
```

```
PS C:\Users\BryanCafferky> C:\Users\BryanCafferky\Documents\Corp\Book\PowerShell\Scrip
Method invocation failed because [System.IO.DirectoryInfo] does not contain a method n
'substring'.
At C:\Users\BryanCafferky\Documents\Corp\Book\PowerShell\Script\ch_2_first_script.ps1:
char:1
+ $speaker.Speak( $speak.substring(1, 100), 1) | out-null
```

Figure 1-16. *The PowerShell ISE—Writing a script*

After entering the code in the script window, click on the green arrow to run it. Notice, you get red error messages in the command window. What went wrong? When the variable `$speak` was assigned a value, the variable type was not defined. You can declare the variable type or let it default to object. However, if you take the default type, you don't get the handy string methods like `substring`. The line causing the error is copied below.

```
$speaker.Speak( $speak.substring(1, 100), 1) | out-null
```

This line fails because `$speak` is not a string. Let's change the `$speak` variable to be a string. To do this we need to modify the line that assigns a value to `$speak` as shown here:

```
[string] $speak  = Get-ChildItem
```

Now click on the green arrow (or press F5), and this time you should hear your computer read the contents of the root directory of your C: drive. Note: To run just highlighted code, you would press F8.

Now save the script by clicking on the diskette icon in the toolbar. I like to save scripts like this in a subfolder named `Scripts`. This makes it easy to remember where I put it. As with other types of code, it is a good practice to put PowerShell scripts under source-code control and back up the files frequently.

Summary

In this chapter we discussed what PowerShell is and reviewed its history. We described the two modes of using PowerShell—i.e., the Command Line Interface (CLI) and the Integrated Script Environment (ISE). The PowerShell CLI is used for the immediate execution of commands entered at a prompt. The ISE is the PowerShell script development environment for PowerShell. By default, PowerShell will not allow scripts to be executed. We discussed how to enable scripting. Then we used the ISE to write our first script. We discussed how we can customize the ISE settings to fit our needs. Finally, we reviewed a script with an error and showed how to correct the error.