PowerShell for SQL Server Essentials helps us to manage and monitor server administration and application deployment. Use PowerShell along with SQL Server to perform common DBA tasks.

Starting with a basic introduction to PowerShell, the initial chapters will provide the SQL Server professional PowerShell fundamentals, covering topics such as PowerShell notations and syntax, cmdlets, pipeline, and getting help. Succeeding chapters build upon these fundamentals, and illustrate how to administer and automate SQL Server. Tasks covered throughout include profiling the SQL Server instance, performing backup and restores, invoking T-SQL scripts using PowerShell, and monitoring jobs, security, and permissions.

Packed with practical examples and numerous ready-to-use snippets, this book gets you to an intermediate level in using PowerShell for SQL Server.

Who this book is written for
This book is written for SQL Server administrators and developers who want to leverage PowerShell to work with SQL Server. Some background with scripting will be helpful but not necessary.
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Getting Started with PowerShell'
- A synopsis of the book’s content
- More information on **PowerShell for SQL Server Essentials**

### About the Author

**Donabel Santos** (SQL Server MVP) is a business intelligence architect, trainer/instructor, consultant, author, and principal at QueryWorks Solutions, based in Vancouver, Canada. She works primarily with SQL Server for database/ data warehouse, reporting, and ETL solutions. She scripts and automates tasks with PowerShell and creates dashboards and visualizations with Tableau and Power BI.

She is a Microsoft Certified Trainer (MCT). She provides consulting and corporate training to clients. She is also the lead instructor for SQL Server and Tableau (Visual Analytics) courses at British Columbia Institute of Technology (BCIT).

Donabel is an MCITP DBA and a developer for SQL Server and MCTS for SharePoint. She is also a Tableau Desktop 7 Core Certified and a Tableau Desktop 8 Certified Professional. She is currently working on her SQL Server 2012 (and upcoming 2014) certifications.

She is a self-confessed data geek. She loves working with data and thinks SQL Server is a lot of fun and Tableau is just amazing at delivering insights. She authored *SQL Server 2012 with PowerShell V3 Cookbook*, Packt Publishing, and contributed to *PowerShell Deep Dives*, Manning Publications. She blogs at www.sqlbelle.com and tweets at @sqlbelle.
Acknowledgments

I didn't think I had it in me to write another book. However, my niece came along after the first book was published and she wasn't in my acknowledgements. So, I wanted to have an opportunity to mention her in another book.

To my dearest Chiyo: I hope you always remember that Tita loves you very much. Tita will always be there for you whenever you need her.

In my first book, I apologized for the lengthy acknowledgements. In this second book, I will do the same.

To Eric: thank you for still being here with me through the ups and downs, the happy times, and the crazy times. I am looking forward to many more adventures, side by side, hand in hand. I love you.

To Papa and Mama: you always give me strength and inspiration. I keep on going because of you. Thank you for everything that you've done for us, and I am so happy that your granddaughter gives you a lot of joy. I love you both very much.

To JR and RR: you will always be my baby brothers, and I am so proud to be your elder sister.

To Lisa: you're my sister, and I wouldn't have it any other way. I'm there for you and will be there to support you as best as I can.

To my in laws: Mom Lisa, Dad Richard, Ama, Aunt Rose, Catherine, David, Jayden, and Kristina; thank you for being my family. Thank you for all the fun times and all the support all these years. Thank you for being there whenever I needed you; words cannot express my gratitude. Jayden and Kristina, Agi and Agu love you two very much, and we'll be there for you to play with you, teach you, and support you. We just want hugs and kisses in return.

To my BCIT family: Kevin Cudihee, Joanne Atha, Elsie Au, Cynthia van Ginkel, Steve Eccles, Dean Hildebrand, and to all my students, past and present; thank you. BCIT is my second home. It has paved the way for many good things in my life and I will always be grateful.

To my UBC family: my super wonderful boss extraordinaire, Pradeep Nair, and my superb teammates Joe Xing, Min Zhu, George Firican, Mary Mootatamby, Jason Metcalfe, Tom Yerex, and Suzanne Landry. I love going to work everyday. You are all awesome; we have a great team and it is a privilege to work with all of you.

To the Packt team: Meeta Rajani for contacting me to author this book and Akshay Nair, who has helped me throughout the process; thank you.
I didn't do this alone. I have learned so much from so many other people, all the SQL Server and PowerShell MVPs, and each technology's communities and bloggers. The Tableau community is also quite inspiring, from Zen masters (Joe Mako, Jonathan Drummey, Kelly Martin, and Dan Murray) to all the bloggers and vizzers. Special thank you to Dan Murray, Tim Costello, Jason Schumacher, John Pain, and Liz Feller. Thank you all for making learning fun again.

There are so many other people who inspired and helped me along the way, including friends, students, and acquaintances. Thank you.

Most importantly, thank you Lord for all the miracles and blessings in my life.
PowerShell for SQL Server Essentials

PowerShell is Microsoft's platform for task automation. It comes with both a shell and scripting language, and is now more deeply integrated with Microsoft's suite of products. Microsoft applications such as Windows, Exchange, and SharePoint have increased their PowerShell support, and many tasks can now be done without having to go through the user interface. These automated and streamlined tasks equate to time savings and increased productivity for developers, administrators, and IT professionals.

As a database professional, you can also leverage PowerShell in your work. This book introduces you to PowerShell and taps into how you can use PowerShell in the context of SQL Server.

What This Book Covers

Chapter 1, Getting Started with PowerShell, introduces you to PowerShell and its importance in server management and automation. This chapter is a good starting point for readers who are new to PowerShell and want to get started with its environment and other components.

Chapter 2, Using PowerShell with SQL Server, dives into using SQL Server-specific PowerShell support in different operating systems and SQL Server versions. You will learn about SQL Server-specific modules, cmdlets, and SQL Management Objects (SMO).

Chapter 3, Profiling and Configuring SQL Server, covers how to quickly profile SQL Server and change SQL Server configurations using PowerShell. You will learn more about Get-WmiObject and the SMO Server object.

Chapter 4, Basic SQL Server Administration, covers the tasks in a DBA's checklist. These tasks include getting space/memory usage, backup/restore, enabling features, jobs, alerts, and so on.

Chapter 5, Querying SQL Server with PowerShell, shows the methods to query SQL Server from within PowerShell, its pros and cons, and how to export results.

Chapter 6, Monitoring and Automating SQL Server, teaches you how to perform SQL Server usage and performance monitoring, logging, alerting, and error checking using PowerShell.

Appendix, Implementing Reusability with Functions and Modules, shows some snippets required to accomplish the task at hand. It covers the basics of creating and deploying functions and modules.
PowerShell is an object-based Microsoft scripting language that comes with its own console and GUI-based environments. PowerShell provides building blocks for automation and system integration. You can think of PowerShell as glue that can keep different Microsoft components and applications together (and make them play nicely with each other).

Knowing PowerShell can lead to power (pun intended). Treat PowerShell like a new member of your high performance team. To achieve high performance, you need to get to know PowerShell and learn its strengths before you can expect to maximize your results.

The list of topics that you will come across in this chapter is as follows:

- A brief history of PowerShell
- The PowerShell environment
- Cmdlets
- PowerShell providers
- Snap-ins and modules
- PowerShell Pipeline
- Scripting basics
- Running PowerShell scripts
- Getting help
A brief history of PowerShell

Before PowerShell, systems and network administrators managing Microsoft software stacks had to resort to using different tools, languages, and technologies to enable automation and integration tasks. For some tasks, administrators used batch files that could be run using Command Prompt (or DOS Shell, for those of you who still remember this term). For other tasks, maybe Visual Basic Scripting Edition (VBScript) was used. Yet, for additional tasks, maybe Windows Scripting Host (WSH) was used. The list goes on.

In a lot of ways, administrators had to be creative because there was not one single language and tool they could use to bridge different Microsoft (and non-Microsoft) tasks together. Unix and Linux administrators, on the other hand, always had C-shell and trusty bash to rely on. At that time, Microsoft just did not have that powerful a command-line tool.

Enter PowerShell. PowerShell was born out of this need for integration and automation. Jeffrey Snover, the inventor of PowerShell, initially incubated PowerShell under the project named Monad. He originally described Monad as the next generation platform for automation.


More than 10 years after this manifesto was written, PowerShell has already improved and matured in leaps and bounds and has indeed become the platform for automation and integration for Microsoft products (and even non-Microsoft packages).

As of today, many Microsoft products have adopted PowerShell and delivered numerous cmdlets (we will talk about them later) with their respective product installations. Windows Server, Active Directory, Exchange, SharePoint, SQL Server are products that support PowerShell (to different extents), but the support has widened through the years.
The PowerShell environment

There are two environments that come with PowerShell when you install it: the PowerShell console and the **PowerShell Integrated Scripting Environment (ISE)**. These environments have improved a lot since the first version and should be more than sufficient for you to start working with PowerShell. If you prefer a different environment, there are other PowerShell editors out there. Some editors are free and some commercial. Some vendors that provide PowerShell editors are Idera (PowerShell Plus), Dell (PowerGUI), and SAPIEN Technologies (PowerShell Studio 2014).

This book uses the current released version at the time of writing, which is PowerShell v4. The screenshots you will see in this book reflect the screens in PowerShell v4.

In a 64-bit system, PowerShell will come in two flavors: 32 bit and 64 bit. The 32-bit version has the label suffix (**x86**). Note that 64-bit add-ons and snap-ins for PowerShell will only load in the 64-bit console or ISE. The following screenshot shows the result of searching PowerShell in Windows:
The PowerShell console
The PowerShell console is very similar to the Command Prompt. By default, the interface is blue, compared to the usual black of the Command Prompt:

![PowerShell Console](image)

The PowerShell console is great for administrators and IT professionals who prefer to work on a purely command-line environment. The console is also great for running predefined scripts either manually or through a job via the Windows task scheduler or SQL Server Agent.

The PowerShell ISE
A standard installation of PowerShell also comes with an Integrated Scripting Environment (ISE). The PowerShell ISE is a more Graphical User Interface (GUI) way of working with PowerShell and comes with a few handy features, including IntelliSense and syntax help, as shown in the following screenshot:
Some of the compelling features that the ISE has are listed as follows:

- The script editor and PowerShell console in a single environment
- The autocomplete and on-hover usage/syntax guide
- A command pane that allows you to visually fill in parameters and transfer the syntax over to your editor
- Multiple tabs that allows the opening of multiple scripts at the same time
- A zoom slider, which is great for presentations or just basic readability

We will use the PowerShell ISE for most examples in this book.

**Running PowerShell as an administrator**

Most of the time, you will use PowerShell to perform administrative tasks, so you will need to run it as an administrator. You can do this by right-clicking on the application (console or ISE) and clicking on **Run as administrator**.
Getting Started with PowerShell

You will know you've successfully run the application as the administrator by looking at the title bar. It should show **Administrator: Windows PowerShell**:

![PowerShell Administrator Window](image)

If you do not run your PowerShell environment as the administrator, you might not have sufficient permission to run some of your commands or scripts. You will most likely get **Access Denied** errors.

A useful trick to identify whether you are running the shell as the administrator is to change the appearance of the shell based on the elevation status of the session. This can be accomplished by adding a snippet of code to your profile that checks whether the session is run by an administrator and then changing some properties accordingly.

First you need to check whether your profile exists. You can check the path to your profile by typing the following command:

```powershell
$profile
```

If this file doesn't exist, you can simply create it by typing the following:

```powershell
New-Item -ItemType File $profile -Force
```

The `$profile` command is equivalent to `$profile.CurrentUserCurrentHost`, which means the settings you provided will only work on the current host. Note that your console and ISE will each have its own profile, so you may need to create one for each. The values you can specify with the profile are `AllUUsersAllHosts`, `AllUsersCurrentHost`, `CurrentUserAllHosts`, and `CurrentUserCurrentHost`.

Here is a simple snippet you can add to your profile that changes the background and foreground color of your shell if you running the shell as an administrator:

```powershell
if ($host.UI.RawUI.WindowTitle -match "Administrator")
{
    $host.UI.RawUI.BackgroundColor = "DarkRed"
    $host.UI.RawUI.ForegroundColor = "White"
}
```

![Snippet Code Example](image)
The execution policy

At the risk of sounding like a dictionary, I will define execution policy as the policy applied to determine whether a script can be executed by PowerShell. Execution policies do not make the scripts more secure. They simply lay the ground rules before a script is executed.

The available execution policies are provided in the following table:

<table>
<thead>
<tr>
<th>Policy</th>
<th>Runs a command?</th>
<th>Runs a local script?</th>
<th>Runs a remote script?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Restricted</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>AllSigned</td>
<td>Yes</td>
<td>Must be signed</td>
<td>Must be signed</td>
</tr>
<tr>
<td>RemoteSigned</td>
<td>Yes</td>
<td>Yes</td>
<td>Must be signed</td>
</tr>
<tr>
<td>Unrestricted</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes — prompts before running downloaded scripts</td>
</tr>
<tr>
<td>Bypass</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes — no warnings or prompts</td>
</tr>
</tbody>
</table>

The default execution policy depends on the operating system you are using. For Windows 8, Windows Server 2012, and Windows 8.1, the default policy is Restricted. For Windows Server 2012 R2, it is RemoteSigned.

Should you need to sign your scripts, you can refer to Scott Hanselman's blog post available at http://www.hanselman.com/blog/SigningPowerShellScripts.aspx. Although this was written a few years ago, the content is still relevant. Patrick Fegan from Risual also has a good, more recent tutorial on self-signing PowerShell scripts at http://consulting.risualblogs.com/blog/2013/09/20/signing-powershell-scripts/.

To get more information about execution policies, including risks and suggestions on how to manage them, you can type Get-Help about_Execution_Policies in the command-line window, or you can visit the TechNet page at http://technet.microsoft.com/en-us/library/hh847748.aspx for more detailed descriptions.

If you want to check which execution policy you are running on, you can use the following command:

Get-ExecutionPolicy
If you want to change it, use the following command:

`Set-ExecutionPolicy`

The following is a screenshot of what you can expect when you run these two cmdlets:

```
PS C:\> Get-ExecutionPolicy
RemoteSigned
PS C:\> Set-ExecutionPolicy Unrestricted

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=135178. Do you want to change the
execution policy?
[Y] Yes  [N] No  [S] Suspend  [?] Help (default is "Y"): Y
PS C:\>  
```

It would be good to read more on execution policies, evaluate the risks that come with the different settings, and evaluate your needs before deciding which setting you should use.

**PowerShell versions**

PowerShell has matured since its inception and has undergone several version upgrades. At the time of writing of this book, the most recent version is PowerShell V4.

The following table shows the different PowerShell versions that Microsoft released, operating systems that support them, required .NET Framework version, and some of the notable features:

<table>
<thead>
<tr>
<th>PowerShell version</th>
<th>OS support</th>
<th>.NET version</th>
<th>Notable features/ additions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version 1, which is a separate download</td>
<td>Windows XP, Windows Server 2003, and Windows Vista</td>
<td>.NET Framework 2.0</td>
<td>Over 130 cmdlets</td>
</tr>
<tr>
<td>PowerShell version</td>
<td>OS support</td>
<td>.NET version</td>
<td>Notable features/additions</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>---------------------------------------</td>
<td>-----------------------</td>
<td>-------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| Version 2, which is part of WMF 2.0 | • Integrated with Windows 7 and Windows Server 2008 R2  
• Available for XP and Windows Server 2003  
• Can be downloaded separately as part of WMF 2.0 | .NET Framework 2.0 or .NET Framework 3.5 SP1 | Over 240 cmdlets, which includes PowerShell ISE, remoting, eventing, background jobs, script debugging, and modules |
| Version 3, which is part of WMF 3.0 | • Integrated with Windows 8 and Windows Server 2012  
• Available for 7 and Windows Server 2008 and later | .NET Framework 4.0 full | • Over 400 cmdlets  
• Workflows, improved sessions, scheduled jobs, and the Update-Help cmdlet  
• PowerShell ISE improvements, which include IntelliSense, command pane, and collapsible regions |
| Version 4, which is part of WMF 4.0 | • Integrated with Windows 8.1 and Windows Server 2012 R2  
• Available for Windows 7 and Windows Server 2008 and later | .NET Framework 4.5 full | • Over 520 cmdlets  
• Desired state configuration  
• Shell and scripting improvements |
| Version 5, which is part of WMF 5.0 | At the time of writing this, a CTP version is available with Windows Management Framework 5.0 | NA | NA |
PowerShell matures with every release and the requirements and features will change with different operating systems.


To determine which PowerShell version you are using, you can type in $PSVersionTable in your console or ISE:

If you have PowerShell v3 or v4, you can also downgrade your PowerShell session. You can do this by supplying the -Version parameter when you start your session:

Powershell.exe -Version 2

PowerShell cmdlets

At the heart of PowerShell is a cmdlet (pronounced as commandlet). A cmdlet is described in MSDN (available at http://msdn.microsoft.com/en-us/library/ms714395(v=vs.85).aspx) as:

"... a lightweight command that is used in the Windows PowerShell environment.

... cmdlets perform an action and typically return a Microsoft .NET Framework object to the next command in the pipeline."
In other words, cmdlets get the job done in PowerShell. You can think of cmdlets as *small commands* — very specific commands — which you can use to accomplish your task.

To explore the cmdlets available in your PowerShell version, you can use the Get-Command cmdlet. You can filter the results as well. For example, if you want to look for log-related cmdlets, you can use the following command:

```
Get-Command -Name "*Log*"
```

**Cmdlet naming convention**

Cmdlets have a very specific naming convention. They follow the *Verb-Noun* format and they are typically self-explanatory. More specifically, it is *Verb-SingularNoun*.

The following are some example cmdlets available in PowerShell:

- `Get-Service`
- `Test-Path`
- `Set-Content`
- `ConvertTo-Csv`

Note that cmdlet names are self-documenting. You don't really have to guess what the `Get-Service` cmdlet does; it gets the corresponding services in your system.

You can get a list of legal, endorsed verbs by Microsoft using the `Get-Verb` cmdlet. Granted, not all the terms you see are really verbs, but for our purposes, we will treat them as such. For example, Microsoft uses the `New` verb to create new items:

- `New-Service`
- `New-Event`
- `New-Object`

Another verb that Microsoft considers is `Out`, mostly used for output. Take a look at the following examples:

- `Out-File`
- `Out-GridView`
- `Out-Null`
Cmdlet parameters

Note that cmdlets can accept parameters or switches. This makes cmdlets quite flexible. You can supply parameters to cmdlets by specifying a dash followed by a parameter name, space, and the parameter value:

```
Cmdlet -ParameterName ParameterValue -ParameterName ParameterValue
```

It will be easier to understand how parameters work if we go through an example. Let's take a look at the usage syntax for `Get-Service`:

```
PS C:\Users\Administrator.QUERYWORKS> Get-Help Get-Service
NAME
SYNOPSIS
   Gets the services on a local or remote computer.
SYNTAX
   Get-Service [[-Name <String[]>] [-ComputerName <String[]>]
      [-DependentServices] [-Exclude <String[]>] [-Include <String[]>]
      [-RequiredServices] [<CommonParameters>]
   Get-Service [-ComputerName <String[]>] [-DependentServices] [-Exclude
      <String[]>] [-Include <String[]>] [-RequiredServices] -DisplayName
      <String[]> [<CommonParameters>]
   Get-Service [-ComputerName <String[]>] [-DependentServices] [-Exclude
      <String[]>] [-Include <String[]>] [-InputObject <ServiceController[]>]
      [-RequiredServices] [<CommonParameters>]
```

Each block in the help section, shown in the preceding screenshot, represents a parameter set. Each parameter set specifies different combinations of parameters and switches that are all valid when you use `Get-Service`.

Anything in square brackets is optional; anything between angle brackets is mandatory.

Let's consider the following first line of command:

```
Get-Service [[-Name <String[]>] [-ComputerName <String[]>]]
```
The [[-Name] <String[]>] part means that you can specify -Name, which should be your parameter name:

Get-Service -Name *SQL*

Since [[-Name] <String[]>] is surrounded by square brackets, it means it's optional. This parameter name can be left out and you can provide just the value. This makes it positional, meaning the value you provide will map to the parameter defined for that cmdlet at that position. In the following example, the first value will be mapped to the first parameter for Get-Service:

Get-Service *SQL*

The next part [-ComputerName <String[]>] is still overall an optional parameter. However, if you decide to supply the value, you have to specify the parameter name, which is ComputerName. Note that there is no square bracket around ComputerName.

When you specify parameter names, you can also take shortcuts. You can specify just the first few characters of the parameter name, and as long as it's unique, PowerShell will figure out which parameter you are referring to:

Get-Service -Na *SQL*

Although it's quite tempting to use shortcuts, when you are first learning how to use PowerShell, try to always completely spell out the parameter names. This will make your code more readable and easier for the rest of your team to work with your code.

If you have a cmdlet that requires input and you don't provide it, you will be prompted for the values interactively:

```powershell
PS C:\Users\Administrator.QUERYWORKS> Restart-Service
cmdlet Restart-Service at command pipeline position 1
Supply values for the following parameters:
InputObject[0]:
```
Cmdlet aliases

Some of the cmdlets also have aliases by default. This means these cmdlets can be invoked by using a different name than their formal cmdlet name. For example, the following screenshot shows the aliases for `Get-ChildItem`:

![Screenshot showing aliases for Get-ChildItem cmdlet]

You can also create your own aliases using `New-Alias`. Aliases can be useful because in some ways, they allow you to use some of the terms you may already be familiar with and leverage them in PowerShell. Aliases also let you personalize PowerShell to your liking. Be careful not to create too many of these though; it may make your PowerShell scripts confusing and even unreadable to others.

PowerShell providers

Simply put, a PowerShell provider provides a way for PowerShell to access a data store. To get a visual of a provider, think of the file system. The file system is a data store that contains information about files and folders and their properties. We can access the file system via the Command Prompt, PowerShell console, or Windows Explorer. Now try to apply this concept to another data store, for example, SQL Server. Imagine that you can navigate through the objects of SQL Server just the way you navigate your file system.

Learning about providers is important because this allows you to extend what you can do with PowerShell. To list the current providers in your system, use the `Get-PSProvider` cmdlet:
What you see in the preceding screenshot are the default available providers that come with PowerShell v4 on a Windows Server 2012 R2 Standard server. A lot of the providers are accessed using what is called drives. To list the current drives, you can use `Get-PSDrive`:

```
PS C:\Users\Administrator.QUERYWORKS> Get-PSDrive

Name       Capabilities
----------- ----------------
Alias      ShouldProcess
Environment ShouldProcess
FileSystem  Filter, ShouldProcess, Credentials
Function   ShouldProcess
Registry    ShouldProcess, Transactions
Variable   ShouldProcess
```

In a file system, if you wanted to change drives, you can use the `cd` command, which is an alias for `Set-Location`:

```
C:\> cd J: 
```

To navigate to a different provider, you can use the same concept. For example, if you want to navigate the HKLM registry hive (which stands for `HKEY_LOCAL_MACHINE`), you can use the following command lines:

```
C:\> cd HKLM:
HKLM: 
```
To work with items in PSDrive, Microsoft has provided a number of Item cmdlets that are generic enough to perform the task regardless of which drive you are in. To get a list of these cmdlets, you can type Get-Command *Item*. For example, if you are using a file system, you can use the New-Item cmdlet to create a new folder or file. If you are in the registry, it will create a new registry entry.

The recent releases of Microsoft products come with their own PowerShell providers, which you can readily use. You can also create your own providers if you prefer.

MSDN has some documentation on how you can create your own provider available at http://msdn.microsoft.com/en-us/library/ee126192(v=vs.85).aspx. There are even tutorials on how to create providers for non-Microsoft data stores. For example, the version control system Git by @manojlds is available at http://stacktoheap.com/blog/2012/12/01/writing-a-git-provider-for-windows-powershell-part-1/.

Snap-ins and modules

You can extend PowerShell by loading snap-ins and modules. Snap-ins or PSSnapins are dynamic linked library (DLLs) compiled from .NET code, which may contain additional cmdlets and PSProvider. The PSSnapins are old school—they are primarily how you extend version 1, but still supported in version 2, version 3, and version 4. Although considered old school, you can still create snap-ins. Refer to http://msdn.microsoft.com/en-us/library/ms714450(v=vs.85).aspx on how to do this.

The related snap-in cmdlets are as follows:

- Add-PSSnapin
- Get-PSSnapin
- Remove-PSSnapin

Instead of snap-ins, the recommended way of extending the PowerShell functionality from version 2 onwards is using modules. Modules are similar to snap-ins when it comes to extending functionality, but unlike snap-ins, modules can also add functions. Modules also support autoloading, which means the module can be loaded as soon as one of its cmdlets/functions/PSDrive are used.

Modules can be script-based or binary-based. A script module uses PowerShell code saved in a .psm1 file. A binary module is more similar to PSSnapin, where it references a .NET DLL.
Modules are the *new school* way of extending PowerShell, from version 2 onwards. Related cmdlets are listed as follows:

- Import-Module
- Get-Module
- Remove-Module

If you want to write PowerShell extensions, Microsoft recommends that you create modules instead of snap-ins.

**PowerShell Pipeline**

A pipeline is defined in [www.TheFreeDictionary.com](http://www.TheFreeDictionary.com) as follows:

"*a linked series of pipes with pumps and valves for flow control, used to transport crude oil, water, etc., esp. over great distances.*"

I think this definition is very fitting to a pipeline in PowerShell. Instead of crude oil or water, what PowerShell transports is pieces of information. PowerShell also has these *pumps and valves for flow control*—we will see more of these in the later chapters.

The pipe symbol in PowerShell is `|`, also called a bar. You can pipe multiple cmdlets together. When you pipe these cmdlets, the output of one cmdlet becomes the input of the next cmdlet:

```
Get-Service | Select-Object ServiceName, Status | Sort-Object Status -Descending
```

When you are writing your scripts, you may want to add a new line after the pipe and continue typing the next cmdlet on the new line:

```
Get-Service | Select-Object ServiceName, Status | Sort-Object Status -Descending
```

Many script authors also prefer to indent the succeeding lines a little bit to emphasize that these are all part of the same block.
Scripting basics
Let's get a few syntax basics down. This section is not meant to be an exhaustive tutorial on PowerShell's syntax but should serve as a good, brief introduction.

Let's walk through the following script:

```powershell
$currdate = (Get-Date -Format "yyyyMMdd hhmmtt")
$servers = @("ROGUE", "CEREBRO")

#save each server's running services into a file
$servers | 
ForEach-Object {

    $computername = $_
    Write-Host `
`n`nProcessing $computername

    $filename = "C:\Temp\$(computername) - $(currdate).csv"
    Get-Service -ComputerName $computername |
    Where-Object -Property Status -EQ "Running" |
    Select-Object Name, DisplayName |
    Export-Csv -Path $filename -NoTypeInformation
}
```

Even if you are not very familiar with PowerShell yet, you may already be able to tell what the preceding script is trying to accomplish. Simply put, the script iterates over the listed servers and saves the list of running services into a file that acts as a timestamp.

This line creates a variable called $currdate that gets the current system date in the "yyyyMMdd hhmmtt" format:

```
$currdate = (Get-Date -Format "yyyyMMdd hhmmtt")
```

The snippet with an at (@) sign, @("ROGUE", "CEREBRO"), creates an array, which is then stored in another variable called $servers:

```
$servers = @("ROGUE", "CEREBRO")
```
Since $servers contains multiple values, when you pipe it to the Foreach-Object cmdlet, each value is fed into the script block inside Foreach-Object:

```powershell
#save each server's running services into a file
$servers  |
Foreach-Object {

}
```

You are also introduced to a few concepts inside the Foreach-Object block.

To get the current pipeline object, you can use $_. The $_, also referred to as $PSItem, is a special variable. It is part of what PowerShell calls automatic variables. This variable only exists and can only be used in the content of a pipeline. The $ variable contains the current object in the pipeline, allowing you to perform specific actions on it during the iteration:

```powershell
$computername = $_
```

A backtick is an escape character, for example, to add a newline. It is also a line continuation character:

```powershell
Write-Host "\n\nProcessing $computername"
```

Note that the strings are enclosed in double quotes:

```powershell
Write-Host "\n\nProcessing $computername"
```

Strings in PowerShell can also be enclosed in single quotes. However, if you have variables you want to be evaluated within the string, as in the preceding example, you will have to use double quotes. Single quotes will simply output the variable name verbatim.

PowerShell has a subexpression operator, $(). This allows you to embed another variable or expression inside a string in double quotes, and PowerShell will still extract the variable value or evaluate the expression:

```powershell
$filename = "C:\Temp\$(computername) - $(currdate).csv"
```

Here is another example that demonstrates when subexpressions will be useful. The expression to get the date that is 10 days from today is as follows:

```powershell
(Get-Date).AddDays(10)
```

If we want to display the value this expression returns, you may be tempted to use:

```powershell
Write-Host "10 days from now is (Get-Date).AddDays(10)"
```
However, this simply redisplays the expression; it doesn't evaluate it. One way to get around this without using a subexpression would be to create a new variable and then use it in the double-quoted string:

```powershell
$currdate = (Get-Date).AddDays(10)
Write-Host "10 days from now is $currdate"
```

With the subexpression, you don't need to create the new variable:

```powershell
Write-Host "10 days from now is $((Get-Date).AddDays(10))"
```

The example we walked through should give you a taste of simple scripting in PowerShell.

The following is a table that outlines some of these common scripting components and operators:

<table>
<thead>
<tr>
<th>Component</th>
<th>Symbol</th>
<th>Description/examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single line comment</td>
<td>#</td>
<td>This component allows you to include any comments or documentation about your code; text after # in a line is not executed, for example, #get the current date.</td>
</tr>
<tr>
<td>Multiline comment</td>
<td>&lt;#</td>
<td>This allows you to create comments that span multiple lines, as shown in the following example:</td>
</tr>
<tr>
<td></td>
<td>#&gt;</td>
<td>get the current date #&gt;</td>
</tr>
<tr>
<td>Backtick</td>
<td>`</td>
<td>Backtick can be used as an escape character: $name = &quot;Hello `n world!&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This is also a line continuation character; it allows you to break a command into multiple lines—some find it more readable, but beware that some will find it less readable because the backtick character can be conspicuous:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Get-Service <code>-Name *SQL*</code> -ComputerName ROGUE</td>
</tr>
<tr>
<td>Dollar sign</td>
<td>$</td>
<td>By default, variables in PowerShell are loosely typed (that is, the data type changes based on the value stored by the variable):</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$dt = Get-Date</td>
</tr>
<tr>
<td>Single quotes</td>
<td>'</td>
<td>This component allows you to enclose string literals: $name = 'sqlbelle'</td>
</tr>
<tr>
<td>Component</td>
<td>Symbol</td>
<td>Description/examples</td>
</tr>
<tr>
<td>---------------</td>
<td>--------</td>
<td>----------------------</td>
</tr>
</tbody>
</table>
| Double quotes | "      | This component allows you to enclose string literals:  
$\text{name} = "sqlbelle"$
This component also allows you to expand variables (that is, replace variable names within the string to their values) or interpret escape characters:  
$\text{name} = "sqlbelle"$
$\text{message} = "Hello \n \text{name}"$
| Plus          | +      | This component is a string concatenation operator:  
$\text{name} = "sqlbelle"$
$\text{message} = "Hello " + \text{name}$
| Dot           | .      | This component allows you to access properties or methods with the corresponding object:  
$\text{dt}.\text{AddDays}(10)$
| Subexpression | $()$  | This component allows you to embed a variable or expression in a double-quoted string; PowerShell evaluates the expression inside this operator:  
$\text{Write-Host "Date: \$dt.\text{AddDays}(10)\"}$
| At sign       | @()    | This component is an array subexpression operator:  
@("ROGUE", "CEREBRO")
| Square brackets | []    | This component is an index operator. It allows you to access indexed collections (arrays and hash tables):  
$\text{servers} = @\("ROGUE", "CEREBRO"\)$
$\text{servers}[0]$
It also acts as a casting operator:  
$\text{[datetime]}\text{dt}$
| Here-String   | @"    | This component allows you to create a multiline string to assign to a variable without having to break the string into multiple string expressions concatenated by a plus (+) sign. It starts with @" and must end with "@ in a line by itself (no characters or spaces before ending ";@):  
$\text{x} = '@$
Hello $\text{name}$.
This is a multiline string
"@
The table is not a comprehensive list of operators or syntax about PowerShell. As you learn more about PowerShell, you will find a lot of additional components and different variations from what has been presented here.

To learn more about operators, use `Get-Help *Operator*` and go through all the available topics. You can also go to the TechNet page specifically for operators, which is available at http://technet.microsoft.com/en-us/library/hh847732.aspx.

### Running PowerShell scripts

Once you’ve written your script, save your script in a file with a `.ps1` extension. From the PowerShell console, you can run the script by specifying the full path to the script:

```
PS C:\> C:\Scripts\Get-RunningServices.ps1
```

Note that your scripts can also be parameterized so that it can take an incoming value when invoked. If this is the case, you can specify the parameter the same way you specify it in a regular cmdlet:

```
PS C:\> C:\Scripts\Get-RunningServices.ps1 -ComputerName ROGUE
```

If you are at the script directory, you don't have to specify the path. You can also use a dot-sourcing operator to run the script. Dot sourcing a script means that any of the variables and functions in the script are loaded into the current scope and available for use in the same console session:

```
PS C:\Scripts> .\Get-RunningServices.ps1
PS C:\Scripts> .\Get-RunningServices.ps1 -ComputerName ROGUE
```

Note that depending on your execution policy settings, the script may run or get access denied errors. If this is the case, you may either need to adjust your execution policy or sign your script.

### Getting help

PowerShell used to come bundled with help documentation. If you’ve worked with *nix systems, it’s similar to the man page.
Starting with PowerShell v3, however, the help files/system were not installed with PowerShell. One of the chronic problems with a help system that comes bundled with an application is that the contents get outdated right away. Applications are continuously being patched, improved, and changed, and thus the documentation needs to be updated. You will need to consciously download and install the help files when you are ready.

Once ready, run PowerShell as an administrator and just type in the following command:

```
Update-Help
```

This will connect you to a Microsoft server to download the most recent version of PowerShell help:

When you need to look for syntaxes or examples from the help system, you can use `Get-Help` and then the cmdlet name. For example, if you want to get `Get-ChildItem`, you can use the following command:

```
Get-Help Get-ChildItem
```
Getting Started with PowerShell

Other switches available for Get-Help that you might find useful are as follows:

- Get-Help Get-ChildItem -Detailed
- Get-Help Get-ChildItem -Examples
- Get-Help Get-ChildItem -Full

Get-Help can also be simply referred to as help.

Sometimes you may prefer to open the local help system in a different window, in which case you can use the following command:

Get-Help Get-ChildItem -ShowWindow

The result is shown in the following screenshot:

![Help Screen Screenshot]

Having the help document in a different window allows you to do simultaneous tasks, that is, write your script and refer to the syntaxes and examples. The help window also allows for searching and highlighting keywords.
If what you prefer is to view the help online and get the most recent version to date, you can use the following command instead:

`Get-Help Get-ChildItem -Online`

This will open the corresponding Microsoft TechNet entry in your default browser:

---

**Getting help from other cmdlets**

In addition to `Get-Help`, there are two other trusty cmdlets you should know if you want to know PowerShell a lot better. If you need to use a command but only remember the name or part of the name or if you want to get a list of commands based on parameters, you can use `Get-Command`. For example, as introduced earlier in the chapter, you can get log-related cmdlets using the following command:

`Get-Command -Name "**Log**"`
If you need to know what properties and methods are available for an object—for example, a variable or the result returned by a cmdlet—you can use Get-Member, as shown in the following example:

```
$message = "Hello World!"
$message | Get-Member
```

Since a message is a string, the preceding snippet returns all the properties and methods supported for a string data type.

Two risk-mitigation parameters that you should also get acquainted with are -WhatIf and -Confirm. You can add these two parameters to most cmdlets, and they can help you avoid really stressful "oops" situations.

The -WhatIf parameter describes the effect of a command instead of executing it. The -Confirm parameter forces a prompt before executing the command. It pays to be careful before you run scripts in your environment. It pays to be extra careful; as much as possible, test your scripts in a test environment first.

---

**Summary**

This chapter has provided a very basic introduction to PowerShell, from a brief history to environments, cmdlets, and pipelines. This should be enough to get you familiarized with PowerShell fundamentals, a skill you will need to work with the next chapters. It is also important to remember how you can learn more about PowerShell using cmdlets such as Get-Help, Get-Command, and Get-Member. The more comfortable you are looking for resources on your own, the faster and better it will be for you when it comes to learning PowerShell.

This chapter is not meant to be an exhaustive, one-stop shop for PowerShell. There are a number of excellent PowerShell books out there that dig deeper into PowerShell's technicalities, syntaxes, and advanced features.

In the next chapter, we will look at how PowerShell can be integrated with SQL Server.
Where to buy this book

You can buy PowerShell for SQL Server Essentials from the Packt Publishing website. Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.