Introduction

IN THIS PART

Chapter 1
Introduction to Windows PowerShell

Chapter 2
What's New in Windows PowerShell V2
Windows PowerShell is Microsoft’s strategic administrative task automation platform. It began life over 10 years ago and has now become mainstream. Before looking at all of the wonderful things that Windows PowerShell can do, this chapter starts by looking at how we got here, and then examining what Windows PowerShell is. This includes a brief overview of the language and syntax of Windows PowerShell.

Cross-Reference
The contents of this chapter mainly refer to Windows PowerShell Version 1. Version 2 added some great new features, and those are described more in Chapter 2, “What’s New in Windows PowerShell V2.” The features described in this chapter are all contained within Version 2, so everything you learn in this chapter is fully usable in Version 2.

Managing Windows — The Challenges of the Past
The path to Windows PowerShell has been a long but steady one that really started with the launch of the IBM PC in 1981. Since then management of systems has grown from something of a rarity to where we are today. This book starts by looking at where we have come from and the challenges that have arisen.

Management in the Early Days
Microsoft entered the PC operating system (OS) field in 1981, with the launch of the IBM PC. The original PC was a non-networked floppy
disk-based machine. Those who had more than one machine managed by carrying around floppy disks, copying them as needed. There was no hard disk to hold either programs or data. Subsequent versions of the DOS operating system added hard disk support, and eventually, there was local area networking capability.

The growth in corporate networks was greatly enhanced by the introduction of Windows. But management was more an afterthought than designed as a feature. This, of course, led to tools like Symantec’s Ghost to help to manage DOS and Windows systems. While the need to manage the systems was increasing, a number of architectural constraints of the older 16-bit architecture made this more difficult. And of course, at that time, Microsoft was not quite as focused on management as is the case today.

Management with Windows NT
The release of Windows NT 3.1 in the summer of 1993 marked a huge advance both in terms of the product and also the start of focusing on enterprise management. Not only was there a networking stack built in, but there was also a server version that enabled domains. In those days, most management tasks were conducted with GUI applications (for example, File Manager or User Manager). There was a rudimentary shell with a few commands, but coverage was far from complete.

Subsequent releases of NT (Windows NT 3.5 and 3.51) added features, but there was no real change in the overall management approaches within Windows NT itself. Microsoft was embarking on the creation of the Systems Management Server, but the creation of what we now know as System Center Configuration Manager took a number of years.

By the release of Windows 2000, some things had begun to change. Microsoft was pushing hard into the enterprise market where manageability was a prerequisite. As any Unix administrator would tell you, to manage large numbers of systems, you need automated scripting. To some degree, it felt like the mandate changed from “You manage from the GUI” to “You manage from the GUI and the command line.” There was finally some acceptance that all those Unix guys had been right all along. But management was still very much piecemeal, with no overarching strategy or consistent toolset.

For Windows 2000, and more so for Windows Server 2003 and Windows XP, there was a push for command-line parity. If you can do something from the GUI, you should be able to do it from the command line. This led to a plethora of command-line tools from each different product group and subgroup. This change was highly welcome, of course, but not without challenges. None of the tools resembled any of the other tools, so what you learned about one was definitely not transferrable.

Management with Windows Server 2003
During the Windows 2003 days, things continued on — much as with Windows 2000 — but with improved feature parity between the command line and GUI. There were really no fundamental changes in the approach to managing Windows desktop and server systems, at least for public consumption.
Chapter 1: Introduction to Windows PowerShell

By the time Microsoft released XP and Windows Server 2003, the very earliest version of Windows PowerShell, or Monad as it was then called, had begun to surface. But Monad wasn’t really enterprise-ready. Some groups within Microsoft began talking up this new approach, but the mainstream audiences were not taking much heed at that point.

Another key aspect of managing this generation of systems was the huge number of Group Policies added into the client OS (XP). Microsoft also beefed up the Windows Management Instrumentation (WMI) components, although to some degree, this was probably more useful to folks writing management tools than to IT professionals.

During this time period, Microsoft was pushing Systems Management Server (SMS, later to be renamed System Center Configuration Manager), which was homegrown, as well as Microsoft Operations Manager (renamed later to System Center Operations Manager), which Microsoft acquired from a purchase. However, in those days, the individual products (that is, Operations Manager and SMS) were very distinct and separate products. The package we now recognize as Systems Center, and the other members of the family, were still some years off.

Introducing Windows PowerShell

To some degree, the death knell of the Management By GUI age was the publication of the Monad Manifesto in August 2002. You can download this document from http://blogs.msdn.com/b/powershell/archive/2007/03/19/monad-manifesto-the-origin-of-windows-powershell.aspx.

The manifesto suggested that the key issue was the lack of “administrator-oriented composable tools to type commands and automate management,” which were the domain of scripting languages. The main scripting tools of the day, however, worked by using “very low level abstractions such as complex object models, schemas and APIs.”

The paper goes on to suggest a broad architecture of components. Though a lot of details have changed since that document was written, Windows PowerShell today delivers on the promise.

A year later, in September 2003, Microsoft demonstrated Monad in public for the first time at the Professional Developers Conference. Though it took a number of years to get from Monad to where Windows PowerShell is today, the result has made the journey worthwhile.

What Is Windows PowerShell?

Before you begin to use Windows PowerShell, you must understand a bit about it. This section takes a look at what Windows PowerShell is and what it contains.

Windows PowerShell as a Task Automation Platform

Windows PowerShell is, first and foremost, Microsoft’s strategic administrative task automation platform. It aims to help the administrator, the IT professional, to manage all
Part I: Introduction

aspects of a Windows system and the applications that run on them as efficiently as possible, both locally and remotely. Such a tool needs to be focused on the administrator and work with high-level task-oriented abstractions. For example, rather than worrying about bits inside a file, the tool should work at the level of a user, process, service, and so on.

Since 2009, Windows PowerShell has been a part of Microsoft’s Common Engineering Criteria (CEC) for Windows and Windows applications. The CEC mandates that all new applications and all parts of Windows must have at least adequate Windows PowerShell support. If a product or component does not meet those criteria, it does not ship. At least that’s the theory.

Note

Windows PowerShell has several components:

- **Rich administrative shell**: On a par with the best of Unix shells in terms of both ease of use and power
- **Powerful scripting language**: As rich and powerful as Perl, Ruby, and VBScript
- **Production orientation**: Aimed at IT professionals running large enterprise environments where there is a strong need for secure, robust, and scalable scripting
- **Focus on Windows and Windows applications**: Works across all supported versions of Windows and has to support all the applications

Although not stated in the Monad Manifesto, but noted at the first public outing of Monad a year later, there was also a need for a rich, vibrant community. The community needed to, and indeed has, focused Microsoft on doing the right things with Windows PowerShell and has filled the gaps in terms of additional features you can just plug into Windows PowerShell. The staggering support provided by the community is nothing short of amazing.

This book examines every aspect of Windows PowerShell and shows you the product, warts and all. But before diving deep, it’s necessary to review some of the key concepts behind Windows PowerShell. If you are new to Windows PowerShell, you should take the time to read this, but if you have a good basic understanding of Windows PowerShell, feel free to skip over this next section.

**Windows PowerShell’s Scripting Language**

Windows PowerShell provides both a shell and a shell scripting language. In the Windows PowerShell console, you can enter individual lines of Windows PowerShell’s language constructs (for example, `Create-AdUser`, to create a new Active Directory account). But you can also add a number of Windows PowerShell statements together into a script file to automate more complex administrative tasks such as provisioning a user into your environment (creating the Active Directory account, adding a SharePoint Site, adding the users to groups, and so on).
Chapter 1: Introduction to Windows PowerShell

Windows PowerShell’s language is broadly based on C#, with concepts (for example, the pipeline) taken from other great scripting languages. Windows PowerShell is, as Microsoft points out, “on the glide scope” to C#. If you know Windows PowerShell, then reading C# should be relatively straightforward and vice versa. Having said that, a number of constructs in C# have not been added to Windows PowerShell because the focus of the two languages is quite different: C# is aimed at professional programmers building applications, whereas Windows PowerShell is aimed at IT professionals who manage those applications.

Later, this chapter presents the basics of this language. The description is brief and provides only the basics. To really understand and use Windows PowerShell, you need practice. Later chapters expand on the introduction you get in this chapter.

In writing this book, the authors wish to concentrate on using and leveraging Windows PowerShell in Windows, and all the key applications you’re likely to run into. To avoid hundreds of pages describing the details of the syntax and language in minute detail, we prefer to let you refine that on the job. What follows here are the basics of the Windows PowerShell language.

Note
Microsoft has done a fantastic job in adding great documentation on Windows PowerShell’s fundamentals into the product. You can find these topics by typing `Get-Help about_*` at the Windows PowerShell prompt. There are more than 90 help files that contain great details of each of the specific language features, including examples.

Windows PowerShell in Production Scripts and Admin GUls

Windows PowerShell was designed for use both at the command line and in production-oriented scripts. This requirement gives rise to the need to be very pithy at the command-line console while verbose and rich in a production script. At the command line, you can issue terse commands, making use of Windows PowerShell’s alias and parameter naming conventions, which enable you to specify only the minimum. In production-oriented scripts, spelling things out in full, along with providing rich validation and error-handling features, becomes much more important.

Another aspect of Windows PowerShell is the ability to use it in building GUI administration tools. In this approach, the key administrative functions are actually built as cmdlets. The GUI just gathers enough data to call these cmdlets and then renders the output. This enables you to create a simple GUI for the most common administrative tasks, which are often performed by less skilled individuals. The less common administrative tasks, which are usually performed by more skilled administrators, are carried out solely using cmdlets.

A great example of this is Microsoft Exchange. With Exchange 2007 and Exchange 2010, the GUI (the Exchange Management Console) is relatively simple (certainly when compared with the Microsoft Management Console snap-in that was included in earlier versions of
Part I: Introduction

Microsoft Exchange!). Adding a mailbox, for example, is done by the GUI gathering the information (mailbox name and so on) and constructing a call to the `New-Mailbox` cmdlet. The output from this cmdlet is then returned to Exchange. Exchange can then show the results (i.e., an updated list of mailboxes).

With Exchange, at any rate, the command issued to create a new mailbox is shown once the administrative action is complete. This allows you to copy it and then use it as the basis for writing scripts to add more users. Other products, notably Microsoft Lync Server 2010, do not provide such a feature. But in both cases, everything you can do at the GUI can be done from a Windows PowerShell console. And from the Windows PowerShell console, you can do more than you can in the GUI.

Next, you take a look at the concepts of Windows PowerShell and how you can take advantage of them.

Key Windows PowerShell Concepts

Within Windows PowerShell are three core conceptual pillars: cmdlets, objects, and the pipeline. It’s hard to talk about one without talking about the other two, so the definitions of these pillars, these key concepts, intertwine to some degree.

Cmdlets

A cmdlet is a small bit of executable code that performs some administrative task such as deleting a file, adding a user, or changing the registry. Cmdlets are named with a verb-noun syntax with strict guidelines for verb naming. An example cmdlet is `Get-Process`, which returns information about processes running on a machine.

To ensure consistency, the set of verbs that developers can use is restricted through the use of formal guidance (and runtime checking that emits an error if unapproved verbs are used in a cmdlet). That helps to ensure that the “get” verb has the same semantics in Active Directory as in Exchange — and that’s the same semantics for `Get-Process`.

Cmdlet nouns can vary more because they are task-specific. A cmdlet’s noun, however, should always be singular, possibly with a prefix to avoid collision (where two product groups produce similarly named cmdlets that do potentially different things). Quest’s Active Directory tools use the noun prefix `QAD`, whereas Microsoft’s Active Directory cmdlets use the prefix `AD`. So, although both cmdlet sets provide a way to get a user in the AD, Quest’s tool uses `Get-QADuser`, whereas Microsoft’s cmdlet is `Get-AdUser`.

To some degree, learning the verbs Windows PowerShell uses for any given task domain is easy — these are standard (`Get`, `New`, `Remove`, and so on). What differs are the nouns, which are in effect the task domain objects. Thus, in Active Directory (AD), you work with users (`Get-AdUser`), groups (`Get-AdGroup`), and domains (`Get-AdDomain`), whereas in Lync
Chapter 1: Introduction to Windows PowerShell

Server you work with topology (Enable-CSTopology), analog device (Get-CSAnalogDevice), location policy (Get-CSLocationPolicy), and so on.

Cmdlets can have aliases — shortcut names to simplify typing, particularly at the command prompt. Thus, GPS is an alias for Get-Process. Windows PowerShell comes with some built-in aliases, but you can easily add your own aliases in profile files that run each time you run Windows PowerShell.

Cmdlets can take parameters that tell the cmdlet how to work. The Get-Process cmdlet has a property, -Name, which is used to tell Windows PowerShell the name of the processes you want information about. Cmdlet property names always begin with a hyphen (-) and are separated from the parameter value and other parameters by a space.

Windows PowerShell provides you with parameter value globbing; that is, specifying a parameter value with wildcards to match potentially more than one object. Thus, you could issue the cmdlet Get-Process -Name P*W to get all the processes that begin with a “p” and have a “w” somewhere later in the process name.

Parameter full names, which can get long in some cases, can also be abbreviated. Windows PowerShell lets you use the fewest number of characters necessary to distinguish one parameter name from all the others.

Objects

Cmdlets consume and produce objects — we say Windows PowerShell is object-oriented. An object is a computer representation of some tangible thing, such as a process running on a computer, or a user in the Active Directory. The Get-Process cmdlet produces a set of zero, one, or more process objects. In the absence of any direction from you, Windows PowerShell renders the objects produced onto the screen in a format defined by Microsoft.

An object has some definition, or class, that defines what each object occurrence contains. Get-Process produces objects belonging to the .NET class System.Diagnostics.Process. A cmdlet can produce zero, one, or more occurrences of the class — Get-Process can return any number of process instances, each representing a single process.

Note

Windows Powershell is built on top of .NET, but you don’t need to be a .NET expert to use Windows PowerShell. As you learn more about Windows PowerShell, you will naturally learn more about .NET, including the details of .NET objects.

Class instances have members that include properties, methods, and events. A property is some attribute of the instance, for example, the CPU time used by a particular process. A method is some function that the class knows how to do on an instance; for example, to kill a specific process, you could call that instance’s Kill() method. Events are specific things that an object can trigger and that you detect using Register-ObjectEvent.
Classes can also have both static methods and static properties. These are properties and methods of the class in general as opposed to a particular instance. For example, the [System.Int32] class has a static property called MaxValue, which is the largest value of a 32-bit integer. This class also contains a static method called TryParse, which attempts to parse a string into a 32-bit value (and returns a value to indicate if the parsing was successful).

**Note**
For some help on objects, type **Get-Help About_Objects** in Windows PowerShell.

### The Pipeline
The pipeline is a device in Windows PowerShell that takes the output objects produced by one cmdlet and uses them as input to another cmdlet. For example, taking the output of **Get-Process** and sending it to **Sort-Object** to change the order of the process objects would look like this in Windows PowerShell:

```
Get-Process -Name * | Sort-Object -Property Handles
```

The pipeline is not really a new concept. The Unix and Linux operating systems have had this feature for decades. However, with Unix/Linux, the pipeline is most often used to pass just text — with Windows PowerShell, the pipeline uses objects. That means when the **Sort-Object** cmdlet in this pipeline gets a set of process objects to sort, it can tell exactly what kind of object is being passed and precisely where to find the field(s) to sort on (that is, it knows what the Handles property is and how to sort it).

By comparison, with Unix, you'd need to take the text output produced by one command and do some text parsing, often called prayer-based parsing, and hopefully get the right answer. Thanks to a cool feature in .NET called Reflection, a cmdlet can look at actual objects passed and not have to rely on pure text parsing.

**Note**

The pipeline is an amazingly powerful construct, although it does take a bit of time for many administrators to understand the concept and to start to use it efficiently.

**Note**
For more information on the pipeline in Windows PowerShell, type **Get-Help About_Pipeline** in Windows PowerShell.

### Discovery and the Community
Discovery is a central component of Windows PowerShell, because it enables you to find out more about Windows PowerShell by using it. Windows PowerShell is in many ways self-documenting, which is of huge benefit to new and seasoned users alike.
Chapter 1: Introduction to Windows PowerShell


Get-Help

The Get-Help cmdlet provides a good introduction to individual Windows PowerShell cmdlets. Get-Help provides details on each cmdlet, including how it works, its syntax, parameter information, and examples of the cmdlet in use.

Get-Help can also provide information about Windows PowerShell concepts. More than 90 built-in “About_” files describe Windows PowerShell language constructs and concepts. The conceptual help built into Windows PowerShell is an important part of discovery — Get-Help really is your friend!

Every cmdlet in Windows PowerShell supports the -? switch, which gives basic help information about that cmdlet. This enables you to type the following to get basic help information about the Get-Process cmdlet:

```
Get-Process -?
```

Get-Command

The Get-Command cmdlet returns related, but different, discovery information. With Get-Command, you can find out the names of the command that meet a certain criteria, such as having a particular verb or noun, or coming from a particular add-in module.

For example, to find the name of the cmdlets that have a “Get” verb, you could type:

```
Get-Command -Verb Get
```

To find all the cmdlets that were added when you imported the Bitstransfer module (a set of cmdlets shipped with Windows 7 and Windows Server 2008 R2), you could type:

```
Import-Module BitsTransfer
Get-Command -Module BitsTransfer
```

Note

Modules and the Import-Module cmdlet are features that are added with Version 2. Modules provide a simple way of adding new sets of cmdlets into Windows PowerShell. Get-Command provides a great way to discover the cmdlets added by a particular module.
Part I: Introduction

If you are about to start using some new module, one key way to discover the nouns that belong to the module, such as `BitsTransfer`, is to type:

```
Get-Command -Module BitsTransfer | Group-Object -Property Noun | Sort-Object Count -Descending
Count Name            Group
----- ----            -----            
 7 BitsTransfer    {Complete-BitsTransfer, Get-BitsTransfer...}
 1 BitsFile        {Add-BitsFile}
```

**Get-Member**

The `Get-Member` is another key discovery-based cmdlet. `Get-Member` takes any object and tells you what’s inside. Thus, if you pipe the output of `Get-Process` to `Get-Member`, Windows PowerShell returns details about the members of the `System.Diagnostics.Process` objects that are produced by `Get-Member`. This description includes the methods and properties supported by that object. By piping an unfamiliar object to `Get-Member`, you can discover what it contains and how to interact with it.

**The Windows PowerShell Community**

Windows PowerShell was designed from the outset to be extensible. The Windows PowerShell team alone could not produce all the cmdlets needed to manage Windows and all the Windows applications. From the very beginning, Windows PowerShell had an add-in model, the `PsSnapin`, that enabled developers to create new cmdlets and other extensions. A developer could write a Windows PowerShell snap-in, known as a `PsSnapin`, in a .NET language, typically C#. This could then be loaded and used on any system that has Windows PowerShell loaded. Writing cmdlets was relatively easy and developers both inside and outside Microsoft jumped at the challenge.

With Version 2 of Windows PowerShell, Microsoft added a new model for adding functionality into Windows PowerShell: the module. A module enables you to do nearly everything a snap-in could, but also enables you to write what are in effect script cmdlets — functions that act like fully featured cmdlets. These functions could be used standalone as well as in a pipeline, and could support the `Get-Help` facilities noted earlier.

The community has produced a number of outstanding additions to Windows PowerShell — a full description of all the various add-ons would require a small book! Two noteworthy examples are the PowerShell Community Extension (PSCX) and the Quest AD tools. PSCX adds a number of highly useful cmdlets, for example, a set that works with Microsoft’s message queuing feature. An even larger add-in was the Windows PowerShell Pack, a mega-module that shipped as part of the Windows 7 resource kit (and is available for free for download). This add-in provides hundreds of additional functions for use in a variety of situations.
Chapter 1: Introduction to Windows PowerShell

Note


In addition, countless blogs and other areas provide great community support. Pretty much anywhere someone can ask a question, or provide an answer to a question, you’ll find passionate Windows PowerShell advocates. This includes Twitter, the microblogging site, where you can ask simple questions and get answers in near–real time.

As with other Microsoft technologies, Microsoft has rewarded a number of Windows PowerShell community members with the coveted Microsoft Most Valuable Professional (MVP) award. If there’s somewhere someone can add to the Windows PowerShell evangelism, you’ll probably find MVPs!

The community has played, and continues to play, a vital role in both guiding the future of Windows PowerShell and in providing great resources to anyone who wants, or needs, to find out more.

Windows PowerShell Language Constructs

As with any scripting or programming language, there is an underlying set of language constructs you need to learn in order to use Windows PowerShell. You can divide these into two broad camps: the basics of Windows PowerShell when operating from the keyboard, and the extra features you use when writing production-oriented scripts. This section introduces the key concepts.

Variables

Like most languages, Windows PowerShell supports the concept of a variable, a named object you assign a value to and then use in other aspects of Windows PowerShell. Variables are indicated in a script or from the command line by a $ and a variable name. Thus, $A and $ThisIsALongVariable are both variables.

To assign a value to a variable, you use the assignment operator =. The following are examples of creating variables:

```powershell
$MagicNumber = 42
$MyName = "Rebecca Marie"
$Files = Get-ChildItem C:\PowerShellScripts
```
Part I: Introduction

The first example sets a variable to the value of 42. Windows PowerShell sets the value of $MyName to the string "Rebecca Marie" in the second example, and in the third example, the $Files variable (which most Windows PowerShell users just call $files) gets the output of Get-ChildItem cmdlet on a particular folder.

In Windows PowerShell, you can use a variable to hold any sort of object, from simple objects like numbers or strings to more complex objects like a Windows service or process. In fact, because the data types come from .NET, a variable can hold any .NET data type you assign to the variable. In .NET, each object you can create is known as a class. Classes are at the core of .NET, and you use them all the time with PowerShell to do all the detailed work.

In the first part of the preceding example, Windows PowerShell sets the type of $MagicNumber to be a 32-bit integer, System.Int32, and in the second example, Windows PowerShell sets the type to string, or more formally, System.String. The third example is a little harder because a folder can hold two different types of .NET objects: folders (System.IO.DirectoryInfo) and files (System.IO.FileInfo). In these three cases, Windows PowerShell works out what is the most appropriate type for a given assignment.

If you want to override the type, you can specify the type name explicitly. To assign a value of 42 to $MagicNumber, but have that number be a 64-bit integer (to enable the use of much larger numbers), you would use:

```powershell
[System.Int64] $BigMagicNumber = 424242424242424242
```

If you create a variable in this way, you cannot assign another type (for example, System.Int32) to the variable because the type is set for the duration of the Windows PowerShell session.

Note

For more help on variables, type Get-Help about_Variables in Windows PowerShell. ■

Operators

Operators act on variables and constants to produce new values that you can use in Windows PowerShell scripts either to control the flow of execution or to assign to a variable. Like most programming and scripting languages, Windows PowerShell supports a rich set of operators, which include:

- **Arithmetic operators:** These operators perform basic arithmetic on numeric types and include + (addition), − (subtraction), * (multiplication), \ (division), and % (modulo). Note that you can add two strings and you can multiply a string by a number. See the about_Arithmetic_Operators help file for more information on these operators.

- **Assignment operators:** These operators assign the value of an expression to a variable. Assignment operators include = (simple assignment) and +=, −=, *=, /=,
The latter operators assign a variable the value of that variable plus the expression to the right of the assignment operator. `$s += 10`, for example, adds 10 to the value of `$s` and assigns the results back to `$s`. You can use the same approach to multiply (`$a *= 3`), subtract (`$a -= 32`) or divide (`$a /= 10`). See the about_Assignment_Operators help file for more information on these operators.

- **Comparison operators:** These operators compare two expressions and return true (if the two expressions compare appropriately) or false. The comparison operators include `-eq (equal), -ne (not equal), -lt (less than), -ge (greater than or equal), -like (wildcard match), -notlike (wildcard nonmatch), -match (regular expression match), -notmatch (regular expression nonmatch), -band (Boolean and), -bor (Boolean or), -bxor (Boolean exclusive or) and -bnot (Boolean not). See the about_Comparison_Operators help file for more information on these operators.

- **Logical operators:** These enable you to build more complex expressions and include `-and, -or, -xor (exclusive or), and -not (the alias for -not is !). See the about_Logical_Operators help file for more information on these operators.

Windows PowerShell also has a number of more specialized operators, as follows:

- **Redirection operators:** These operators enable you to redirect output to a file and include `>` (send output to a file), `>>` (append output to a file), `2>` (send error stream to a file), `2>>` (append error stream to a file), and `2>&1` (send error and regular output to the same file). See the about_Redirection help file for more information on the redirection operators.

- **Split operator:** This operator splits one or more strings into substrings. See the about_Split help file for more detail on the Split operator.

- **Join operator:** This operator joins one or more strings. See the about_Join help file for more information on this operator.

- **Type operators:** These operators enable you to check if a variable or expression is (or is not) of a particular type, and to convert an expression to another type. See the about_Type_Operators help file for more details on the type operators.

- **Contains operator:** This operator returns true if an element is contained within an array, or false otherwise. For more information on arrays and the contains operator, see the about_Arrays help file.

- **Unary operators:** These two operators (`++` and `--`) add and subtract one from a variable and store the result back into the variable. `$a++` is the same as `$a=$a+1` (and `$a+=1`), and `$a --` is the same as `$a=$a-1` (or `$a-=1`).

- **Format operator:** The `-f` operator is used to format a composite format string, which precedes the `-f` operator using values from the array following the operator.
Expressions

An expression is a set of operators and operands that result in a value. An operand is some value that an operator can act on. Adding two numbers involves two operands (the numbers) and an operand (that tells Windows PowerShell to add the two numbers).

In some cases, the resultant value can be a simple Boolean (that is, either true or false), and in other cases it may be a numeric or some other value. Like most modern programming languages, you can affect the order of calculation by enclosing sub-expressions in parentheses. For example, here are some simple expressions:

```powershell
$a = 1; $b = $a * 10       # $b is assigned an expression based on the
                    # value of $a
$a -gt 100 -or $b -le 21   # expression is true if a is more than
                    # or $b is less than 22.
-not (1,2,3) -contains 3   # returns false
$area = $pi * ($radius *$radius) # area of a circle with a radius of $radius
```

Wildcards (–like) and Regular Expressions (–match)

As noted earlier, Windows PowerShell provides two types of special string comparison operators, -like and -match (plus their alter egos of -notlike and -notmatch). The -like and -notlike operators compare a string with a wildcard string returning true if there is a match. The -match and -notmatch operators do much the same thing, but match against a .NET regular expression. If you are not familiar with regular expressions, they are explained later in this chapter.

You can specify wildcards to match on both one or multiple characters and also range. In addition to “*” to match zero or more characters, and “?” to match either zero or one character, Windows PowerShell wildcards also enable you to specify a range of characters [a-b] or a set of characters [asfl] to compare. Here are some examples:

```powershell
'Cookham' -like 'C*'        # true
'Cookham' -like 'Cook*'     # true
'Cookham' -like 'C?kh?m'    # true
'Cookham' -like 'C[aeiou][a-o]?ham'  # true
```

Windows PowerShell also supports the -match and -notmatch operators, which perform regular expression matching. Regular expressions are a way of specifying rich pattern-matching criteria that Windows PowerShell can use to match (or not) against another string. People are easily able to differentiate strings like doctordns@gmail.com, 131.107.2.200, and \lon-dc1\documents\letter.docx. Simple wildcards are not adequate to do this sort of rich pattern matching. Instead, Windows PowerShell uses .NET regular expressions.

For example:

```powershell
'rmlt@psp.co.uk' -match '[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}' # true
'131.107.2.200' -match '\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}' # true
```
Chapter 1: Introduction to Windows PowerShell

**Note**
Regular expressions are a valuable skill and are complex in their own right. To learn more about regular expressions, see the `About_RegularExpressions` help file. Also take a look at [www.regular-expressions.info](http://www.regular-expressions.info) for a tutorial on regular expressions, as well as a wealth of examples.

**Case Sensitivity — or Not**

For the most part, Windows is a case-insensitive operating system, in regard to the various names and naming conventions used (for example, DNS names, NetBios names, filenames, registry key names, and UPN names, to name a few). With very few exceptions, names are case-insensitive. Windows does remember the case used and tries to preserve it for display purposes, but in operation, Windows does not differentiate on the basis of case. That means that a filename `C:\FOO\FooBarXxyyXX.txt` is the "same" as `c:\foo\foobarxyyxx.TXT`. The exceptions to case-insensitivity are small (you run across one case when accessing Windows Active Directory using the ADSI interface).

Because Windows is, in effect, case-insensitive, it makes sense that, by default, Windows PowerShell should be case-insensitive. And it is. The various comparison operators noted earlier are case-insensitive. And in most cases, that makes sense. Most scripters use the default comparison operators, which are case-insensitive. This can confuse users who have more experience with Unix and Linux, where case sensitivity does matter.

For most administrative tasks in Windows and Microsoft applications, case sensitivity is rarely important, although there may be cases where it does matter. Windows PowerShell caters to those instances by providing case-sensitive versions of all the comparison operators. This is done by adding a "c" to the start of the operator, to give us `-ceq (case-sensitive equal), -cne (not equal), -clt (less than), -cge (greater than or equal), -clike (wildcard match), -cnotlike (wildcard nonmatch), -cmatch (regular expression match), and -cnotmatch (regular expression nonmatch).

But case sensitivity does not end there. Because you have the ability to explicitly state case sensitivity in a comparison operation, there’s an argument that says you should have the ability to explicitly perform operations in a case-insensitive way. There is some symmetry (being able to explicitly compare with case-insensitivity and case-sensitivity). To support that, Windows PowerShell uses an “i” instead of a “c” at the start of each operator; thus, you have `-ieq (case-sensitive equal), -ine (not equal), -ilt (less than), -ige (greater than or equal), -ilike (wildcard match), -inotlike (wildcard nonmatch), -imatch (regular expression match), and -inotmatch (regular expression nonmatch).

This is demonstrated in the example here:

```
'a' -eq 'A'                        # True
'a' -ceq 'A'                       # False
'a' -ieq 'A'                       # True
'COOKHAM' -eq 'cookham'            # True
'COOKHAM' -ceq 'cookham'           # False
'COOKHAM' -ieq 'cookham'           # True
```
Part I: Introduction

'COOKHAM' -like 'c*' # True
'COOKHAM' -clike 'c*' # False
'Cookham' -ilike 'c*' # True

Providers

Providers are Windows PowerShell data access components that provide a consistent interface to different data stores. This enables you to use a consistent set of cmdlets to access any data store for which a provider exists. Windows PowerShell comes with a set of Providers, including:

- **Alias**: Provides access to the set of cmdlet aliases you have defined (using `New-Alias` or `Set-Alias`)
- **Environment**: Provides access to the Windows environment variables set on your computer
- **FileSystem**: Provides access to the file store in a way similar to how both Unix shells and the Windows `cmd.exe` program display the file store
- **Function**: Provides access to the set of functions defined on your computer
- **Registry**: Provides access to the Windows registry
- **Variable**: Provides access to the set of variables in use
- **Certificate**: Provides access to the certificate store

Each provider enables you to create provider-specific drives. When you use them, Windows PowerShell accesses the different underlying data stores. To see Windows PowerShell’s Provider coverage, try running the following on your computer:

```powershell
Cd c:\
Dir
Ls
Get-ChildItem
Cd hkey:
Ls
Cd cert:
Ls
cert:
Ls alias:dir
```

**Note**

For more information on Providers, see the *about_Providers* built-in help file.

Formatting Output

Unlike other scripting or programming languages, such as VBScript, Windows PowerShell was designed from the outset to create output by default, thus keeping the user from having to do a lot of work to get sensible output. This can dramatically simplify both command-line...
ad hoc usage as well as production scripts. You can also override Windows PowerShell’s
default formatting to create as complex an output as you might wish to.

**Default Formatting**

Whenever you run a cmdlet/pipeline/script, that action can leave objects in the pipeline. For example, when you call `Get-Process` on its own, you leave a set of process objects in the pipeline. Even just typing the name of a Windows PowerShell variable leaves object(s) in the pipeline (that is, the object contained in the pipeline). In such cases, Windows PowerShell attempts to format the objects using a set of simple rules that are supported by customizable XML.

Windows PowerShell supports formatting XML, which describes how a particular object class should be output, by default. Additionally, Windows PowerShell supports type XML, which can state the properties that are to be output when a given object is displayed (the type XML includes the properties to be output and not the specific format to be used). Microsoft’s default formatting and type XML are loaded each time you run Windows PowerShell and provide a good default starting set. You can, of course, write your own to either add to or improve what Windows PowerShell does by default.

When Windows PowerShell finishes a pipeline (which can be one or more commands), it looks to see if any objects are left over. If so, Windows PowerShell first looks at the loaded format XML to see if there is a view of the objects (in the pipeline). For example, if you run `Get-Process`, Windows PowerShell produces a set of `System.Diagnostics.Process` objects. Windows PowerShell would then look to see if there is a view that’s been defined of these objects in any of the loaded format XML files. If so, that view is chosen and defines how Windows PowerShell formats the remaining objects.

If there are no view declarations, Windows PowerShell has to work out how to format the properties. Via the .NET reflection capability, Windows PowerShell can “see” what objects are in the pipeline and what properties they have, so this is relatively straightforward.

If there is a `PropertySet` declaration in any of the registered type XML files, this defines the specific properties to be displayed. If there is no `PropertySet` declaration, Windows PowerShell uses all the properties in the objects.

Finally, Windows PowerShell has to work out whether to format the objects in a table or a list. If the number of properties to be displayed is four or less, Windows PowerShell formats them as a table; with five or more, Windows PowerShell formats the objects as a list. When formatting a list, Windows PowerShell, by default, determines the width to be used for each column (unless there is display XML that specifies a specific column width). Windows PowerShell also uses the property name as the column header.

When formatting the `System.Diagnostics.Process` objects, Windows PowerShell discovers a view for that object class in one of the predefined format XML files that directs Windows PowerShell to generate a table with a set of predefined properties. This format
XML also gets Windows PowerShell to perform some calculations on the underlying property, for example, displaying the virtual memory used by a process in megabytes (versus bytes) to improve readability.

### Formatting Using Format-Table and Format-List

When composing a pipeline, rather than leaving objects in the pipeline for Windows PowerShell to format by default, you can pipe them to either `Format-Table` or `Format-List`. This enables you to override the properties displayed, their order, and whether to display the objects as a table or list.

With both `Format-Table` and `Format-List`, you specify the specific properties to be displayed. Thus, you could do the following:

```powershell
Get-Process -Name * | Format-Table -Property ProcessName, StartTime, WorkingSet64, CPU
```

This would produce the output you see in Figure 1-1.

![Figure 1-1](image-url)

Formatting a table with Format-Table

As you can see, this simple pipeline produces a nice output, although Windows PowerShell is quite generous with the amount of space between each column. To avoid using so much space, you can specify the `AutoSize` parameter. When you specify this parameter, `Format-Table` first works out the largest width for a column (based on the actual data being displayed) and then uses the minimum number of characters to ensure only the minimum of space is left between each column in the table, as you can see in Figure 1-2.
By using `Format-Table` or `Format-List`, you can display any property of any object in either a table or list format. If you don’t know an object’s property names (that is, the names you specify to `Format-Table` or `Format-List`), then pipe the object to `Get-Member`. This outputs a list of all the properties, their types, and whether you can get (only) or both get and set that property on an instance of the object’s class. Piping `Get-Process` to `Get-Member` shows you the properties of the `System.Diagnostics.Process` class, such as `Priorityclass` and `Starttime`, but also that `Priorityclass` can be set and read, but `Starttime` is read-only.

Windows PowerShell offers a third useful format cmdlet, `Format-Wide`. This cmdlet displays the values of just a single property of the object being displayed, for example, the process name for each process that is returned by `Get-Process`. Figure 1-3 shows the use of `Format-Wide` to format the process name of the processes.

Something you notice when using the format cmdlets is that the default format used to display each property and the column/row labels are fixed. Windows PowerShell, again by default, chooses the best display format based on the data type being output and uses the property name for the column/row header.
Part I: Introduction

Formatting with Windows PowerShell Hashtables

Windows PowerShell supports an object called a hashtable, a special sort of array that contains entries with just a key and a value. Hashtables are discussed in more detail later in this chapter. But for now, the hashtable(s) you use has a predefined set of keys, making setting up a hashtable simple (although the syntax is a bit on the ugly side for most new to Windows PowerShell, and is probably ugly for the rest of us, too).

You use a hashtable to tell Format-Table or Format-List how to format a particular column or row. You can use what are known as calculated properties to include a row or column title, an expression defining the actual value to display (for example, VM as megabytes), and detailed format instructions on how to format numbers/dates. For use with Format-Table, the hashtable can also contain a column width and a justification (right/left).

To format a table of processes that contains process name, CPU time, and virtual memory used when using a hashtable to alter the column headers and to specify how each property is calculated and used, you could use the following script — with the results as shown in Figure 1-4:

```powershell
$ProcessHT = @{Label="Process Name"; Expression={$_.Name}; Alignment="Right" Width=25}
$CpuHT     = @{Label="CPU Used"; Expression={$_.Cpu}; FormatString="N2"; Width=10}
$VmmHT     = @{Label="Virtual Memory (MB)"; Expression={$_.VirtualMemorySize64/$(1mb)} FormatString="N1"; Alignment="Center"; Width=15
Get-Process notepad| Format-Table $ProcessHT,$CpuHT,$VmmHT
```

![Figure 1-4](image-url)

Formatting a table with hashtables
In this example, you create three hashtables, each describing a column you want `Format-Table` to display. The first column is 25 characters wide and uses the process's name property as the data, which is right-aligned in the column. The second hashtable, `$CpuHT`, uses "Virtual Memory" as the header of a 10 character–wide column that displays the object's CPU time. When this value is converted to a display string, the .NET format string `N2` (numeric with two digits of precision) is used to neatly format the result. The last hashtable displays a final column entitled "Virtual Memory (MB)." This column contains the virtual memory size, divided by 1 MB, that is left-centered in a 15-character column.

This example, which is a bit advanced, shows you how you can take advantage of the .NET formatting and `Format-Table` or `Format-List` to format nearly any table or list just the way you like it. There are other ways to create complex output, but I leave those as an exercise you can complete once you have more experience with Windows PowerShell.

The way that you tell Windows PowerShell how to convert numbers and dates into text is via the `FormatString` hashtable key. The format of what goes into the key is based on .NET. You can get the full details of .NET’s numeric format strings at [http://msdn.microsoft.com/en-us/library/427bttx3(VS.71).aspx](http://msdn.microsoft.com/en-us/library/427bttx3(VS.71).aspx) and .NET’s date and time format strings at [http://msdn.microsoft.com/en-us/library/97x6twsz(VS.71).aspx](http://msdn.microsoft.com/en-us/library/97x6twsz(VS.71).aspx). Of course, as an administrator, you might not want to take the time to customize the output because the default output may be good enough.

### Scripting

In this section, you look at the concept of Windows PowerShell scripts, what they contain, and how you use them.

### What Is a Script?

A script is nothing more than a text file of Windows PowerShell commands. *Scripting* is the art and science of creating these files of commands and then executing them as a single entity. You could create a script to provision a new user into your organization. This script might take data from an Excel spreadsheet about the users, and might include creating a new AD user account, adding that account to some security groups, adding a mailbox, a Unified Messaging mailbox, a Lync account, or a SharePoint site, plus setting all the necessary ACLs.

This complex script, the details of which are a matter of company business policy, is just a set of calls to cmdlets (for example, `New-AdUser`, `New-Mailbox`, and so on) or calls to functions you develop locally. They are all things you could do, a step at a time, from the console. The only problem with that is it could take a long time, even assuming you typed every statement perfectly each time. If your boss walks in with a spreadsheet containing 1000 new users he needs to get created as soon as possible, the thought of all that typing would drive most folks over the edge!
Part I: Introduction

The beauty of a script is, once it is created, you can just run it, sit back, and watch it do all the work. The script completes the same actions you might have performed at the console far faster and more reliably. Scripting is the key to repeated and reliable automation, which is, after all, the primary focus of Windows PowerShell.

Scripts can be of virtually any length, and generally consist of some or all of the following components:

- **Business logic:** What the script is meant to do through the use of cmdlets and associated processing pipelines. In the case of the provisioning script noted earlier, it might add a user to the AD using `New-ADUser` (an AD cmdlet), then create a mailbox for that account using `New-Mailbox` (an Exchange cmdlet), and so on.

- **Error handling:** Every cmdlet can fail based on a large variety of factors. Trying to add a user to AD might fail if you already have an account with the same name as you are trying to add, or if AD is for some reason down.

- **User input validation:** Any time you get input from any user (even you!), treat it with suspicion until you validate it thoroughly.

- **Logging:** Creating a detailed log that can be audited at some later date. If nothing else, the logging can show your boss that you just added the 1,000 new users he asked you to add 10 minutes ago.

- **Windows PowerShell language constructs:** You use these to orchestrate the individual actions the script performs. These provide the rich glue that binds a script together.

Two important programming constructs that you use in most scripts are known as *alternation* and *iteration*. A script can do different things, that is, alternate, based on some condition (create a special set of log entries if the creation of the user was not successful). Also, scripts often process groups of objects (those 1,000 users you just added), iterating through one or more individual objects one at a time, for example, creating each user for each line on the Excel spreadsheet using the values on that spreadsheet line. Windows PowerShell has rich syntax to enable you to do both of these.

**Alternation or Conditional Execution**

As noted, alternation happens when a script takes a different action depending on a condition. Windows PowerShell provides several language control features for managing alternation, namely the *if* statement and the *switch* statement. The *if* statement, which has several variations, involves evaluation and expression, and depending on the value, it performs different actions.

The basic form of an *if* statement is *if* ( <condition> ) { <action> ). For example:

```powershell
if ($a -gt 100) {Write-Host '$a is big'}
```
The second form uses an else clause, taken if the condition is not true. For example:

```powershell
if ($a -gt 100) {Write-Host '$a is big'} else {Write-Host '$a is small'}
```

A third form enables you to have multiple mutually exclusive if clauses:

```powershell
if   ($A -gt 100){Write-Host '$a is big'}` elseif ($A -gt 50) {Write-Host '$a is fairly big'}` elseif ($A -gt 18) {Write-host '$a is fairly small'}` else {'$A is small or tiny'}
```

When writing more complex if statements, you may need to use the line continuation character (`) at the end of the line, as in the preceding example. This stops Windows PowerShell from just executing the first line of the if statement and enables the elseif and final else statement in this example.

The second alternation construct supported by Windows PowerShell is the switch statement (also known as the case statement in VB and other languages). This statement takes a value and repeatedly compares it with a set of values — and takes the indication action when these values are the same. Several variations on the switch statement make it really preferable to the if statement for handling complex types of alternation.

The switch statement has the basic syntax:

```powershell
switch (<expression}) {
<value 1> {<action for expression = value 1}
<value 2> {<action for expression = value 2}
<etc.>
default {<action take for expression not equal to any value}
}
```

To see this in action, here's a more real-life example that makes use of the PowerShell Community Extensions:

```powershell
$a = 1..4 | Get-Random -count 1
Switch ($a) {
  1 {Write-host 'number chosen is 1'}
  2 {Write-host 'number chosen is 2'}
  3 {Write-host 'number chosen is 3'}
  4 {Write-host 'number chosen is 4'}
  Default {Write-Host 'some other number chosen'}
}
```

This example assigns $a to a random number between 1 and 4 and then tests its value using the switch statement. Of course, as long as the random number generator in Get-Random is working, this snippet can only generate a random number greater than or equal to 1 and less than or equal to 4, thus the default action can never be taken, so you could probably omit that last line in the switch statement.
With the `switch` statement, each potential value is checked. Thus, after checking if $a$ is 1 (and taking the action in the script block if so), by default, the `switch` statement then checks if $a$ is 2, and so on. In some cases, $a$ may end up matching multiple values, for example:

```powershell
$a = Read-Host 'enter Y/yes or N/No'
switch ($a.toupper()) {
    'Y'    {$response = 'YES'}
    'YES'  {$response='YES'}
    'N'    {$Response='NO'}
    'NO'   {$Response='NO'}
    default {$Response='Unknown'}}
```

In most situations, however, you want to avoid multiple comparisons because the values you are checking against are mutually exclusive. In those cases, you can end the script block with a `break` statement, which tells Windows PowerShell to jump to the end of the `switch` statement. For example, the earlier `switch` statement might be more efficiently coded as:

```powershell
$a = 1..4 | get-random –count 1
switch ($a) {
1 {Write-Host 'number chosen is 1'; break}
2 {Write-Host 'number chosen is 2'; break}
3 {Write-Host 'number chosen is 3'; break}
4 {Write-Host 'number chosen is 4'; break}
Default {Write-Host 'some other number chosen'}
}
```

In this case, if $a$ is 1, Windows PowerShell performs the `Write-Host`, then jumps to the line after the end of the `switch` statement. If $a$ is 1, then it can't be equal to 2, 3, or 4, so the additional checking is redundant. The `break` statement gives you flexibility to handle mutually exclusive values sensibly.

Two alternatives to the `switch` statement make use of wildcards and regular expressions. In these variations, Windows PowerShell uses either a wildcard or regular expression comparison: Does the value match the wildcard expression or the regular expression? Examples of this are:

```powershell
$a = read-host 'enter Yes or No'
switch -wildcard ($a.ToLower())
{
    'y*' { $response = 'You entered Yes'}
    'n*' { $response = 'You entered No' }
    default { "You entered something else" } }
```

and

```powershell
$a = read-host 'Enter Yes or No'; $reponse = ""
switch -regularexpression ($a.ToLower())
{                      
```

26
"^y" { $response = 'You entered Yes'; break}
"^n" { $response = 'You entered No'; break}
default { "You entered something else" } }

In the first of these two examples, Windows PowerShell did a wildcard comparison between the expression $a.tolower() and the wildcard string 'y*'. In the second example, Windows PowerShell did a regular expression match between the expression $a.tolower() and the regular expression '^y'. As you can see, you are free to use the break statement as and when appropriate.

**Iteration — Operating on a Collection or Array**

As noted earlier, iteration involves looking at a number of objects one at a time. Windows PowerShell has rich iteration support with a variety of syntax to carry out iteration.

Iteration is a programming construct present in every scripting or programming language worth discussing. The idea is quite simple: you create some collection or array (all the files in the folder M:\GratefulDeadShows, or all the processes that are consuming either over 1000 handles or 500 MB of virtual memory), then do some action or set of actions for the members of that collection.

There are four basic iteration operators in Windows PowerShell (more than adequate for all situations). Two of these have two alternative methods of use:

- **for loop**
- **do until / do while loop**
- **while loop**
- **ForEach-Object and foreach statement**

The for loop is one a number of programming languages have, and has the basic syntax:

```
for (<expression 1>; <expression 2>; <expression 3>) {<statements>}
```

The for loop starts by evaluating the expression <expression 1>. Typically, this initializes a loop counter. Then, <expression 2> is evaluated and, if true, the statement block is executed. Finally, <expression 3> is evaluated (typically, this just advances the loop counter that was set in <expression 1>). The loop continues by reevaluating <expression 2>, running the script block if it’s still true, and so on. Here’s an example:

```
For ($i=0; $i -lt 100; $i++){
    $i
}
```

In this example, $i is initialized to zero. Windows PowerShell then evaluates the expression and, because $i is less than 100, the loop body is executed (which just prints out the current value of $i, which the first time is zero). After the loop body is executed,
$i$ is incremented by 1, then tested again to see if it’s still less than 100. In summary, this loop prints out the numbers from 0 to 99. Many old-school programmers find this loop contrast similar to what much earlier programming languages had.

The next three iteration constructs are really just variations on the theme of running a script block until or while some condition is true.

The first, the do...until loop, runs a script block until some condition is true. For example:

```
$i$=0
Do {
   $i
   $i++
} until ($i –ge 100)
```

This example does the same thing as the for loop earlier. A simple variant on this is the do...while loop, which outputs the numbers 1 through 99:

```
$i$=0
Do {
   $i
   $i++
} while ($i –lt 100)
```

A third variation is the while loop, which looks like this:

```
$i$=0
While ($i –lt 100)
{
   $i
   $i++
}
```

All of these looping constructs do broadly the same thing: run some script block multiple times, ending when some condition is true. In the case of the for loop and the while loop, depending on how you construct it, the script block may not run, whereas for the do...while or do...until case, the script block is always run at least once. These iteration constructs work just fine and may ease you into Windows PowerShell. But none of them makes use of the pipeline, which is Windows PowerShell’s secret weapon against complex scripting!

With the foreach constructs, Windows PowerShell runs a script block inside a pipeline — once for each member of the pipeline. Rather than having to construct some means to determine when the loop should terminate, Windows PowerShell can simply run a script block for each member.

The first foreach construct is the ForEach-Object cmdlet, which has a syntax like this:

```
ForEach-Object {<script block>
```

```
$collection = @('apple', 'banana', 'cherry', 'date', 'elderberry')
ForEach-Object {$_.Upper()}  # 'APPLE', 'BANANA', 'CHERRY', 'DATE', 'ELDERBERRY'
```

Chapter 1: Introduction to Windows PowerShell

In this construct, Windows PowerShell runs the script block for every object in the collection. Suppose you had a set of music files in a single folder — some were .mp3, some text (.txt), plus other files. Using the `ForEach-Object` statement, you could categorize these like so:

```powershell
#Initialize variables
$txtfile = $mp3file = $m4a = $other = 0
#Look at all files in c:\music
ForEach-Object ($file in (Get-Childitem c:\music) { switch ($file.extension) {
  *.txt" { $Txtfile++
  *.mp3" { $Mp3file++
  default { $other++ } }
}
#Display results
"$txtfile text files"
"$mp3file MP3 files"
"$other other files"
```

For a well-populated MP3 collection, the output might look like:

<table>
<thead>
<tr>
<th>Type</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>text files</td>
<td>232</td>
</tr>
<tr>
<td>MP3 files</td>
<td>1232</td>
</tr>
<tr>
<td>other files</td>
<td>280</td>
</tr>
</tbody>
</table>

In this `foreach` variant, you state the name you are going to use for the current object being evaluated. Each time the preceding `ForEach-Object` loop is executed, the `$file` variable is set to be the current object (that is, the current file in the C:\Music folder).

A simplified version of `foreach`, the `foreach` statement, is used within a pipeline only. In this variant, there is no “name in collection-name” clause. Instead, each time the loop runs, the current object is represented by the variable `$_. To recast the preceding example, you might have this:

```powershell
#Initialize variables
$txtfile = $mp3file = $m4a = $other = 0
#Look at all files in c:\music
Get-Childitem c:\music | foreach {
  switch ($_.Extension) {
    "txt" { $Txtfile++
    "mp3" { $Mp3file++
      default { $other++ }
    }
  }
}
#Now Display results
"$txtfile text files"
"$mp3file MP3 files"
"$other other files"
```
In this example, each time through the loop, the current file object is represented by \$_ and therefore has a file extension property of \$_.Extension.

It is these last two constructs that are most commonly used with Windows PowerShell. The ForEach-Object cmdlet is, at least for some, preferable for more complex script blocks, whereas the second is more appropriate in short one-liner type pipelines. But both can be used interchangeably.

Error and Exception Handling

In the world of administrative scripting, errors occur. Some are minor and can be fixed easily as you develop a script. Others can be anticipated, trapped, and possibly worked around. Although writing business logic is going to be your focus, you need to anticipate and manage the rich possibility today's computing environment provides as a source of error.

You can divide these errors into three broad classes. First are the syntax errors, where you just typed the wrong syntax or perhaps misspelled a variable or cmdlet. For the most part, these syntax errors are corrected pretty easily because Windows PowerShell won't run the script until the basic syntax is right and terminates if you try to access nonexistent cmdlets, providers, and so on. Reasonable testing of your script exposes these issues for you to correct.

The second type of error is a logic error — your script runs fine, but it produces the wrong results. Logic errors can be hard to find, especially as the script grows in size and complexity, though sometimes, you can look at a script and just see the error and quickly fix it. Other cases may be much harder to work out and discover the underlying issue.

The final type of error is the runtime error — something that should work, but doesn't. For example, if you use the Get-AdUser cmdlet from the Active Directory module, you should get the relevant user(s) returned. But what if the domain controller is down, or the network between you and the domain controller is down?

For pretty much any cmdlet that does something outside the local box, there is the potential for a runtime error. The same applies for operations on your own computer — for example, the comma-separated value file containing users you want to add to the Active Directory does not exist, and so on.

Using an Advanced IDE

One thing that really helps you to detect and correct syntax errors is an advanced Interactive Development Environment (IDE) in which you develop your code. Two specific features that really help you to eliminate syntax and possibly some logic errors are syntax color coding and IntelliSense.

Color coding occurs when the code editor you are using displays different syntax tokens using different colors. For example, if your strings are all color-coded dark red, and suddenly you see a huge block of dark red characters, chances are you have missed either a closing or opening string delimiter.
IntelliSense is where the editor “knows” Windows PowerShell’s syntax and helps you type it. For example, if you start to type the cmdlet Get-WmiObject, a suitably smart IDE would recognize you’ve typed Get-, and pop up all the Get- cmdlets available. In effect, this is tab completion on steroids and can save you a lot of time and effort — not only should it be a bit quicker to type your scripts because the tool does the typing, but you also ensure the tool types the syntax correctly and in full.

In terms of tools you can use, you can start using the Windows PowerShell’s Interactive Script Environment (ISE). This comes as part of the installation of Windows PowerShell on most systems. Windows PowerShell ISE is loaded by default on Windows 7 and when you install Windows PowerShell on earlier client operating systems. For server systems, particularly Server 2008 R2, you need to add this component separately. Sadly, Windows PowerShell ISE is not supported on Server Core. Other tools you can use include Idera’s Windows PowerShell Plus Professional, Quest’s PowerGui (free), and Sapien’s Primal Script.

Cross-Reference
See Chapter 25, “Using the Windows PowerShell ISE,” for more information on both the ISE and alternative products.

Set-StrictMode Cmdlet
The Set-StrictMode cmdlet finds a number of instances of incorrect syntax that might otherwise work (albeit incorrectly) and reports at runtime on the error. For example, you can call a cmdlet using .NET method invocation syntax and, though Windows PowerShell may not complain when you enter such a statement, it almost certainly will not call the cmdlet in the way you intended. Also, you might type a variable name incorrectly and refer to a nonexistent and noninitialized variable, or perhaps a nonexistent property of an object. These are easy mistakes to make, and can be hard to see in a large script.

Using Set-StrictMode causes Windows PowerShell’s parser to be extra strict and report on issues like these (and others). When StrictMode is turned on, Windows PowerShell generates a terminating error if best-practice coding rules are violated (that is, your script stops when such things happen).

Using Set-StrictMode is a great idea while you are developing your script. You might consider setting it in your profile. See the “Customizing Windows PowerShell with Profiles” section later in this chapter for more information about using profiles.

Debugging
Debugging is the process of removing logic and other errors from your script. Windows PowerShell (both from the console and using the ISE) and other third-party tools provide you a wealth of runtime debugging tools.

Although Windows PowerShell is “new,” the concept and practice of debugging has been a part of the computing environment ever since Grace Hopper removed a moth from a valve-based computer in the mid-1950s.
You can take two broad approaches to runtime debugging. First, you can add diagnostic statements to your script that display key information as your script runs. For example, if you issue a `Get-AdUser` cmdlet, the diagnostic information output might include the number of users returned and the names of the users. This might help you fix a problem in the filter (filtering which users you want to get out of AD).

The second approach is to use debugger to step through your program line by line — stopping now and again to look at the values of certain variables (perhaps even setting some values temporarily). Windows PowerShell, both Windows PowerShell console and Windows PowerShell ISE, have a debugging platform you can use.

To produce debug output, Windows PowerShell provides the `Write-Debug` cmdlet. This cmdlet writes debug information to the console when directed. The neat thing about `Write-Debug` is that it prints information only when you set the variable `$PSDebugPreference`.

Windows PowerShell’s core debugging features are provided via seven core cmdlets:

- **Set-PsDebug**: Turns script debugging on and off, sets trace level, and can set a strict level
- **Set-PsBreakpoint**: Enables you to set a breakpoint. You can break at a line/column in a script, whenever a variable is used/set, or whenever a function/script is called
- **Get-PsBreakpoint**: Gets a list of breakpoints currently set
- **Disable-PsBreakpoint**: Disables a particular breakpoint, but does not remove it
- **Enable-PsBreakpoint**: Enables a previously disabled breakpoint
- **Remove-PsBreakpoint**: Removes a previously set breakpoint
- **Get-PsCallStack**: Gets details on how a particular function or script was called (that is, who called who to call who, and so on)

**Note**

For more information on the debugging features inside Windows PowerShell, see the `about_Debuggers` help file. Also, run `Get-Help` on each of the preceding cmdlets for more information on how to use them.

**Trapping Runtime Errors**

As mentioned earlier, runtime errors can affect almost any script/function/cmdlet, even if that bit of code ran thousands of times previously without issue. As Murphy’s Law posits: anything that can go wrong, does so at the most inopportune time; Mrs. Murphy’s corollary was that Murphy was an optimist.

Two Windows PowerShell syntax components enable you to catch and handle errors that would otherwise be fatal. One is the `trap` statement and the other `try/catch/finally` construct.
Chapter 1: Introduction to Windows PowerShell

The *trap* statement enables you to specify a set of commands, a script block to run when an otherwise fatal error has occurred. For example, if you have a script that iterates through a list of, say, 500 systems and does something with those systems — if one system is down, the script would fail. In your script, you can trap such errors, write the information away to a log file, or perhaps send mail or a page alert to an administrator, then continue. This turns fatal errors into recoverable errors.

The *trap* statement on its own traps all errors in any code that follows, such as:

```powershell
Trap {
    "Error Encountered in script - continuing"
    $Error[0]            | out-file c:\scriptlog.txt -append
    "In {0}"         -f | out-file c:\scriptlog.txt -append
    "On {0}, at {1}" –f $(hostname, $(get-date) |           
                        out-file c:\scriptlog.txt -append
    Continue
}
...
```

In this example, the script can go along executing and, if any error occurs, the *trap* statement catches that error, prints out some information to a log file, and then continues.

**Note**
For more information on errors in general, see the *about_Errors* help file. For more information on the *trap* statement, see the *about_Trap* help file. And see the *about_Try_Catch_Finally* help file for how to use the *try/catch/finally* blocks to trap and handle runtime terminating errors.

**Nonterminating Errors**
In the preceding text, the errors discussed were terminating errors — that is, when the script encountered an error, Windows PowerShell terminated the execution of that script. However, a lot of errors that occur can be nonterminating. That means that Windows PowerShell displays error information at the console and then continues to execute your script.

Suppose you had a simple script that takes a file of file names and deletes them. This might look something like this:

```powershell
$Files = Get-Contents C:\Del.txt
Foreach ($File in $Files) {
    Remove-Item $file
}
```

Next, let’s suppose that one of the files (`c:\foo\Foobar.txt`) did not actually exist. In that case, Windows PowerShell would produce an error like this:

```
Remove-Item : Cannot find path 'C:\foo\foobar.txt' because it does not exist.
At line:1 char:12
```
You have two options as to how to handle these nonterminating errors. First, you could use the `-ErrorAction` parameter. Or you could use the `-ErrorVariable` parameter.

The `-ErrorAction` parameter is a common parameter (available on all cmdlets) that tells Windows PowerShell what to do with nonterminating errors. When you call a cmdlet, in this case `Remove-Item`, you can specify four different values of `-ErrorAction`:

- **SilentlyContinue**: Windows PowerShell ignores the error, displays no error text, and continues.
- **Stop**: Windows PowerShell stops, in effect turning a nonterminating error into a terminating error.
- **Continue**: Windows PowerShell displays the error message and then continues, which is the default action.
- **Inquire**: Windows PowerShell asks you want to do next.

You can also use the `-ErrorVariable` common parameter, typically in conjunction with `-ErrorAction`. If you specify the `-ErrorVariable` parameter and provide a variable name, Windows PowerShell stores any nonterminating errors in the variable name. If you precede the variable name with a plus sign (“+”), Windows PowerShell appends any errors to the variable, thus creating an array of errors found. Note that you must specify the variable name without using a “$,” as follows:

```powershell
Remove-Variable x
$Files = Get-Contents C:\Del.txt
Foreach ($File in $Files) {
  Remove-Item $file -ErrorAction SilentlyContinue -ErrorVariable +x
} If ($x.count) {Write-Host "(0) errors deleting files" -f $x.count)}
```

**Extending Windows PowerShell with Snap-ins and Modules**

Windows PowerShell was designed from the start to be extensible, which allows product teams, third parties, and the community to create extensions. This section introduces the snap-in, which came with V1, and shows a sample snap-in.
Cross-Reference

The Module construct, added with Windows PowerShell V2, is another way to add functionality into Windows PowerShell. Modules are explained in more detail in Chapter 2, “What’s New in Windows PowerShell V2.”

Windows PowerShell Snap-ins

When Windows PowerShell shipped, as Version 1, there was a single method of adding functionality — the snap-in, or PsSnapIn. The PSSnapin enabled developers to create installable packages of cmdlets and providers, which could be used on other machines and within other organizations. These extra cmdlets could be free (for example, the Active Directory toolset from Quest) or commercial (for example, /n Software’s networking cmdlets). Or they could be provided by some product or operating system component (for example, the System Migration cmdlets included with Windows Server 2008 R2).

Each snap-in has a full name (for example, Quest’s add-in tools for Active Directory: Quest.ActiveRoles.ADMigration). You can use the Add-PsSnapIn cmdlet to add the snap-in and the Remove-PsSnapIn cmdlet to remove the snap-in. Adding the snap-in makes the cmdlets etc. available for use at the console or within a script.

From your Windows PowerShell console, you can find out what snap-ins have already been added by using Get-PsSnapIn. Unless you have customized your environment by using profile files, if you run Get-PsSnapIn, you can see that Windows PowerShell has loaded a core set of seven Windows PowerShell snap-ins:

- Microsoft.PowerShell.Diagnostics
- Microsoft.WSMan.Management
- Microsoft.PowerShell.Core
- Microsoft.PowerShell.Utility
- Microsoft.PowerShell.Host
- Microsoft.PowerShell.Management
- Microsoft.PowerShell.Security

To find out what cmdlets are inside each of the snap-ins, you can use the Get-Command cmdlet, and specify a module name (that is, Microsoft.PowerShell.Core). The results are shown in Figure 1-5.
Windows PowerShell Modules

The snap-in was a good way of adding functionality, but it had several weaknesses:

- It was a compiled add-on, requiring developers to use a .NET language such as C#. This made it difficult for nondevelopers to construct.
- It had to be installed, so developers needed to create an installer (which fortunately was pretty easy to do!) and the user had to run the installation process.
- It had to be registered in the system registry — for some locked-down workstations, this meant the installation process failed.

For these key reasons, Microsoft created the module, which is part of Version 2. Windows PowerShell modules are discussed in more detail in Chapter 2.

Installing Windows PowerShell

Before you can use Windows PowerShell, you need to install it. This ranges from the trivial (it’s already there!) to the impossible (it’s not supported on Windows 2000 or earlier).
Chapter 1: Introduction to Windows PowerShell

Windows PowerShell Version Support

Basically, you need to do two things to get Windows PowerShell loaded on your system. These two things vary a bit depending on what OS you have. The first thing is getting the binary bits, and the second is installing them. Depending on your operating system, here’s how to proceed:

- **Windows 2000 (workstation or server) and earlier**: Windows PowerShell is not available or supported for these versions of Windows. There is some anecdotal evidence that you can hack Windows PowerShell into Windows 2000, but it’s only going to be a hack — there are missing components that make your experience with Windows PowerShell on this OS extremely suboptimal.

- **Windows XP, Windows Embedded, Windows Server 2003/Windows Server 2003 R2, and Windows Vista**: For these operating systems, you can download an OS Patch and install it.

- **Windows Server 2008**: Server 2008 includes Windows PowerShell V1 as a feature you can install (but not in Server Core installations). You should just download V2 and use that, unless there is some business reason why you need V1.

- **Windows 7 and Server 2008 R2 (Full install)**: Windows PowerShell is included and is installed on these OSs. For Server 2008 R2, there’s even a shortcut icon to Windows PowerShell prepopulated on the Start bar. For Server 2008 R2, the ISE is included as a separate feature, which you can add. The ISE requires the .NET Framework 3.5 SP1, which is also included if you choose to install the ISE.

- **Server 2008 R2 (Server Core)**: For this version of Server Core, Windows PowerShell V2 is included in the binaries, but you need to add Windows PowerShell (and the .NET Framework to support it) before you can use it. Use the Sconfig.exe program to add these two components. After starting Sconfig, just enter 42 to add Windows PowerShell (some may find that amusing — after all, isn’t Windows PowerShell the answer to everything?).

Getting Windows PowerShell for Downlevel OSs

Windows PowerShell is now part of the Windows Management Framework Core (WMFC) component as described in KB article 968930 (see [http://support.microsoft.com/kb/968930](http://support.microsoft.com/kb/968930)). To add Windows PowerShell to your downlevel system, you need to add the latest version of this component, which you can obtain from the KB page on Microsoft’s website.

When finding the version for your system, be careful, because there are seven separate versions of the WMFC component for different versions of the OS and for different hardware platforms. Sadly, there is no support for Itanium.

Script Security and Execution Policy

After installing either the latest OS or installing the WMFC component onto an older OS, you are ready to start running and using Windows PowerShell. You can use either the Windows
Part I: Introduction

PowerShell console (`PowerShell.exe`), Windows PowerShell ISE (`PowerShellISE.exe`) on supported platforms, or any of the Windows PowerShell third-party applications such as Windows PowerShell Plus, PowerGUI, and so on.

However, the first time you try to run a script within Windows PowerShell, you see the ugly error message shown in Figure 1-6.

**FIGURE 1-6**
Scripts being blocked by Windows PowerShell’s default execution policy

Windows PowerShell has an execution policy that applies to each system on which you install Windows PowerShell. This policy tells what scripts can run (all, signed, or none). It is set up to be restrictive by default, but it is very easy for you to change.

The idea behind this is that it might be easy for a malware site to drop a malicious script on your system that you could then be persuaded to execute. So far, there has been no reported case of this vulnerability, but for naïve users, it may be safer to not have the ability to run scripts until they know enough to not be too dangerous. In some higher-security environments, you might want to prevent any but signed (and therefore well-scrutinized) scripts.

However, turning on a restrictive execution policy does not stop determined administrators — they can easily just cut the script from the file in Notepad, and paste it into a Windows PowerShell console. Even if execution policy is restricted, there are plenty of ways a rogue user with administrative privileges can damage a system. Don’t forget, cmdlets are only dangerous if you have the necessary permissions — Windows PowerShell just makes it more efficient to do damaging things for those who already can!

The execution policy can take one of the following values:

- **Unrestricted**: You can run any script.
- **RemoteSigned**: You can run any local script, but scripts from a remote source must be digitally signed (and that signature must be valid).
Chapter 1: Introduction to Windows PowerShell

- **AllSigned**: You can only run scripts that were digitally signed.
- **Restricted**: You can run NO scripts.

You can set the execution policy in three ways:

- **Specify an ExecutionPolicy parameter when starting Windows PowerShell**: This allows you to set the policy for this invocation of Windows PowerShell.
- **Enter Windows PowerShell and run Set-ExecutionPolicy and select a less restricted policy (for example, RemoteSigned or Unrestricted)**: From then on, all Windows PowerShell consoles obey this setting.
- **Use Group Policy**: By setting a group policy object (gpo), you can be granular in which execution policy applies to which systems. Note that group policy overrides any manual setting.


Once you download and install this template, you can use it to set Windows PowerShell’s default execution policy. Once set for a given machine, users on that machine can run any scripts allowed by the execution policy you have set (and they can’t change it without changing or removing the GPO).

**Customizing Windows PowerShell with Profiles**

Windows PowerShell has four scripts it can run at startup. Known as profiles, these scripts are run in “dot source” mode — thus, variables, functions, and so on that you create in the profile are persisted in the Windows PowerShell console. Most users create profile files to customize their Windows PowerShell console or ISE usage.

**What Is a Profile?**

A *profile* is a file that Windows PowerShell runs as part of starting up a Windows PowerShell session. You can take advantage of four profiles, each of which runs before you see the prompt in your Windows PowerShell window. Any variable, alias, or function you define or any provider you load is available for your use in the Windows PowerShell session. For more information on scope, see the *about_Scopes* help file.

A profile file is where you can put all the variables you want to persist in a session, define small functions or aliases, and where useful, create new provider drives. If you are going to be developing large functions that you want to be made available within your PowerShell
console (or for a script), it might be preferable to bundle them up into a module and then just load that module either in your profile or script, or only when you actually need it. This might be a way to speed up your startup times!

Windows PowerShell enables you to use four separate profiles. These enable you to best manage the Windows PowerShell environment. The four are known as:

- **AllUsersAllHosts**: This profile runs for every user and for every Windows PowerShell host.
- **AllUsersCurrentHost**: This profile runs for every user running this specific Windows PowerShell host (for example, `PowerShell.exe`, `PowerShell_ISE.exe`, and so on).
- **CurrentUserAllHosts**: This profile runs for the current user only but for all hosts.
- **CurrentUserCurrentHost**: This profile runs for the current user within only the current host.

This flexibility enables you to have, for example, different profiles when you run `PowerShell.exe` versus the ISE and to have different profiles for different users. It also enables system-wide profiles (that is, for any user using this system) and individual user profiles.

When Windows PowerShell starts, it creates a variable for you, `$profile`, which points to the `CurrentUserCurrentHost` profile, which means you could have two — one for `PowerShell.exe` and the other for `PowerShell_ISE.exe`. For most users, this is sufficient. For multiuser systems, the `AllUsers` profiles can be useful over and above the per-user profile.

**Where Are Your Profiles?**

The four profile files for use with `PowerShell.exe` are listed in Table 1-1. If you installed Windows to a different drive, the location would change.

<table>
<thead>
<tr>
<th>Table 1-1: Default Windows PowerShell Profiles for PowerShell.exe</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Profile Name</strong></td>
</tr>
<tr>
<td>AllUsersAllHosts</td>
</tr>
<tr>
<td>AllUsersCurrentHost</td>
</tr>
<tr>
<td>CurrentUserAllHosts</td>
</tr>
<tr>
<td>CurrentUserCurrentHost</td>
</tr>
</tbody>
</table>
Chapter 1: Introduction to Windows PowerShell

The profiles available for use with Windows PowerShell ISE are listed in Table 1-2.

### TABLE 1-2

**Default Windows PowerShell Profiles for PowerShell_ISE.exe**

<table>
<thead>
<tr>
<th>Profile Name</th>
<th>Profile Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllUsersAllHosts</td>
<td>C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1</td>
</tr>
<tr>
<td>AllUsersCurrentHost</td>
<td>C:\Windows\System32\WindowsPowerShell\v1.0\Microsoft.PowerShellISE_profile.ps1</td>
</tr>
<tr>
<td>CurrentUserAllHosts</td>
<td>C:\Users&lt;username&gt;\Documents\WindowsPowerShell\profile.ps1</td>
</tr>
<tr>
<td>CurrentUserCurrentHost</td>
<td>C:\Users\tfl\Documents\WindowsPowerShell\Microsoft.PowerShellISE_profile.ps1</td>
</tr>
</tbody>
</table>

If you have other Windows PowerShell hosts, they may or may not implement additional CurrentHost profiles. Once you’ve started up your Windows PowerShell host, you can find out the profile files for that host easily. Just run the following:

```
$profile | format-list *host* -force
```

### Managing Profiles in the Enterprise

You have options as to how you use profiles, how you coordinate them, and how you keep them up to date. This can provide flexibility for large and small organizations alike. Some things you could do include:

- Letting your Windows PowerShell users do their own thing and not control profiles centrally.
- Putting all key corporate functions, aliases, and any locally developed scripts and providers into a module.
- Using the AllUsersAllHosts profile for common corporate standards, letting users further customize their per-user profiles.
- Using group policy to deploy a startup script or a logon script that copies your corporate AllUsersAllHosts profile to the appropriate folder on administrative workstations.
- Creating multiple profile files for the different hosts used in your organization and deploying these with group policy (startup or login script), depending on the level of control you want to maintain.
- Providing a sample set of profile contents and letting users download and use them as appropriate.
Summary

In this chapter, you looked at the basics of Windows PowerShell. You reviewed the path leading to the release of Windows PowerShell, and you learned about the core components of Windows PowerShell’s language. These components — cmdlets, objects, and the pipeline — are the fundamentals on which the rest of this book rests.

You learned that you can use these concepts to create rich production-oriented scripts to run your enterprise. Scripts can be a mixture of cmdlets, and existing console applications that are combined with iteration and alternation processes. Add in a mixture of user input validation and error handling to resolve the unpredictability inevitable in automation of your computer environment, and you have rich tools to perform all manner of task automation.

This chapter finished with a look at both installing Windows PowerShell and how you might extend your Windows PowerShell environment with snap-ins and modules. These additions come from a variety of places: some are built in, whereas others are either commercially provided or have been created by Windows PowerShell’s vibrant community. You also examined how you install Windows PowerShell (for those operating systems where Windows PowerShell is not automatically loaded) as well as how to customize Windows PowerShell using profile files.

In the next chapter, you learn about the features added to Windows PowerShell’s Version 2. One could devote an entire book to just what’s new in V2, but Chapter 2 avoids this by providing a concise look at key features.