Part I
PowerShell for Exchange Fundamentals

1

Getting Started with Windows PowerShell

Windows PowerShell is the next-generation command-line shell and scripting language for Windows. Exchange Server 2007 is the first Microsoft application to utilize Windows PowerShell for deployment and administration. This chapter introduces Windows PowerShell and explains the basic concepts you'll need to know to use Windows PowerShell effectively.

The first section, "What Is Windows PowerShell?" includes command shell history and describes the features that make Windows PowerShell the ideal management platform for Exchange Server 2007.

The section that follows, "Windows PowerShell Basics," covers the fundamentals of Windows PowerShell. This section describes the components of Windows PowerShell and how to find commands and then learn how to use them.

To understand Windows PowerShell and its benefit to Exchange administrators, this chapter covers the following key areas:

- ❑ Command shells vs. Graphical User Interfaces
- ❑ Windows PowerShell components
- ❑ Windows PowerShell built-in help
- ❑ Composing commands using pipelines

What Is Windows PowerShell?

Windows PowerShell is a new command-line shell and scripting language for Windows. It was designed by Microsoft specifically to give administrators an extensible command shell for managing Windows environments with greater control and flexibility. This section includes a brief discussion of some traditional administrative interfaces to help you understand why there is a need for an advanced interface like Windows PowerShell. What follows is a discussion of the main features that set Windows PowerShell apart from other management interfaces and make it the most powerful administrative interface that Microsoft has ever produced.

Shell History

Before there was the Graphical User Interface (GUI), there was the Command Line Interface (CLI). The CLI was born of a need to quickly interact with the operating system at a time when computers were mostly controlled using punch card or paper tape input. The first CLIs used teletype machines to enter commands directly into the computer for execution, with the results returned to the operator as printed output. Teletypes were later replaced with dedicated text-based CRT terminals that offered an even greater advantage in speed and the amount of information available to the operator.

All CLIs rely on a program that interprets textual commands entered on the command line and turns them into machine instructions. This program is known as a command-line interpreter or shell. Every major operating system includes some sort of shell interface. UNIX administrators may be familiar with several shells (SH, KSH, CSH, and BASH) as well as the text processing languages AWK and PERL. Windows users may also be familiar with `cmd.exe`, the Windows command-line interpreter and the Windows Script Host for running scripts.

All these shells make possible direct communication between the operating system and the user. They include built-in commands and provide an environment for running text-based applications and utilities.

When Shells Are Better than GUI Interfaces

GUI interfaces came later in computer development and opened the door to less-technically-advanced users looking for a more "comfortable" way to interact with the operating system. Although they provide a simple-to-use interface, GUI applications are prone to user error because their use requires direct interaction between the user and the interface through menus, controls, and fields. For each administrator in the organization to complete the same tasks as all other administrators, they must learn and then use the correct menu choices and controls in order to get consistent results.

GUI-based management programs also constrain administrators to predetermined properties and controls. They lack provisions for special or one-off tasks because they are designed and written with specific functionality that is appropriate for general purposes.

Shells offer a powerful solution for overcoming these GUI shortcomings by providing a method to gather commands into a batch file, also known as a script, and then run them as if they had been entered one at a time at the command line. Administrators create and run scripts to automate everyday tasks and resolve difficult issues GUI interfaces are not designed to handle. Scripts allow a reliable, sustainable method for administering an environment.

Once a script has been written and proven to work for its intended purpose, it can be distributed throughout an organization and used as needed by any administrator, with expected and consistent results. Examples of some common script solutions you might find in most organizations are used for unattended machine deployments, user account provisioning, and nightly database backups.

Common Shell Limitations

The traditional shells mentioned earlier offer an administrator greater control and flexibility for tackling everyday or even unusual management tasks, but they all suffer from significant drawbacks.

Command shells operate by executing built-in commands that run within the process of the shell, or by executing a command or application in a new process outside of the shell. Many applications lack command-line equivalents for controls found in their GUI management programs. And the number of built-in commands offered by most shells is usually small, requiring more applications and utilities to run outside the shell to accomplish critical tasks. Most organizations lack the resources to develop special applications and utilities on their own and may struggle to accomplish more complex tasks using available commands alone.

Another drawback shared by most shells is the way in which they handle information. The results of running a command or utility is returned as text to the command line. If you need to use this text as input for another command, which is common in scripting, it has to be parsed. Parsing is the process of evaluating text and extracting the meaningful values in a form that can then be properly interpreted by another command. Parsing is prone to error and can be time consuming because the format required for preparing the textual input can vary greatly between different commands, applications, and utilities.

One final limitation to consider is the lack of integration between a shell and the scripting languages you would use in that shell. For example, Windows Script Host provides a method for implementing a variety of scripting languages from the command line (via `cmd.exe`), but it is not integrated with `cmd.exe` and is thus not interactive. It also lacks readily accessible documentation from the command line as you would find in many other shells and scripting environments.

The Power Behind PowerShell

What sets Windows PowerShell apart from all other command shells is that it is built on top of .NET Framework version 2.0. Windows PowerShell exposes .NET classes as built-in commands. When these commands are executed they create a collection of one or more structured objects as output. Instead of text, all actions in Windows PowerShell are based on .NET objects.

Windows PowerShell objects have a specific type based on the class used to create them. They have properties (which are characteristics) and methods (which are actions you can take). Because objects have a defined structure, a collection of objects created by one command can be passed to another command as input without the need for parsing the data in-between.

Windows PowerShell includes a fully integrated and intuitive scripting language for managing .NET objects. The language is consistent with higher-level languages used in programming .NET. Those administrators familiar with the C# programming language will find many similarities in the grammar, syntax, and keywords used by the Windows PowerShell scripting language.

Windows PowerShell includes more than 130 built-in commands for performing the most common system administrative tasks. The commands are designed to be easy to understand and use because they share common naming and parameter conventions. Learning how to use one command makes it easy to understand how similar commands are also used.

Because Windows PowerShell is fully extensible, software developers can create their own custom built-in commands to handle those administrative tasks not already addressed in the default built-in command set. Exchange Management Shell is an example of Windows PowerShell extended to include more than 500 built-in commands.

Windows PowerShell not only allows access to the local disk drives as a file system, but it also exposes the local Registry, certificate store, and system environment variables and allows you to navigate them using the same familiar methods you would use for navigating a file system. Windows PowerShell also provides additional data stores for variables, functions, and alias definitions used inside the shell.

GUI management applications can be built on top of Windows PowerShell. Software developers can ensure that all administrative functions found in a GUI management application built on Windows PowerShell have a corresponding scriptable equivalent in the Windows PowerShell CLI. Exchange Management Console is an example of a GUI management application built on top of Windows PowerShell.

PowerShell Basics

You may be asking yourself why a book about Exchange Management Shell is spending so much time in the beginning talking about Windows PowerShell. Because Exchange Management Shell is built on top of Windows PowerShell, you need to understand the basic concepts and components of Windows PowerShell first.

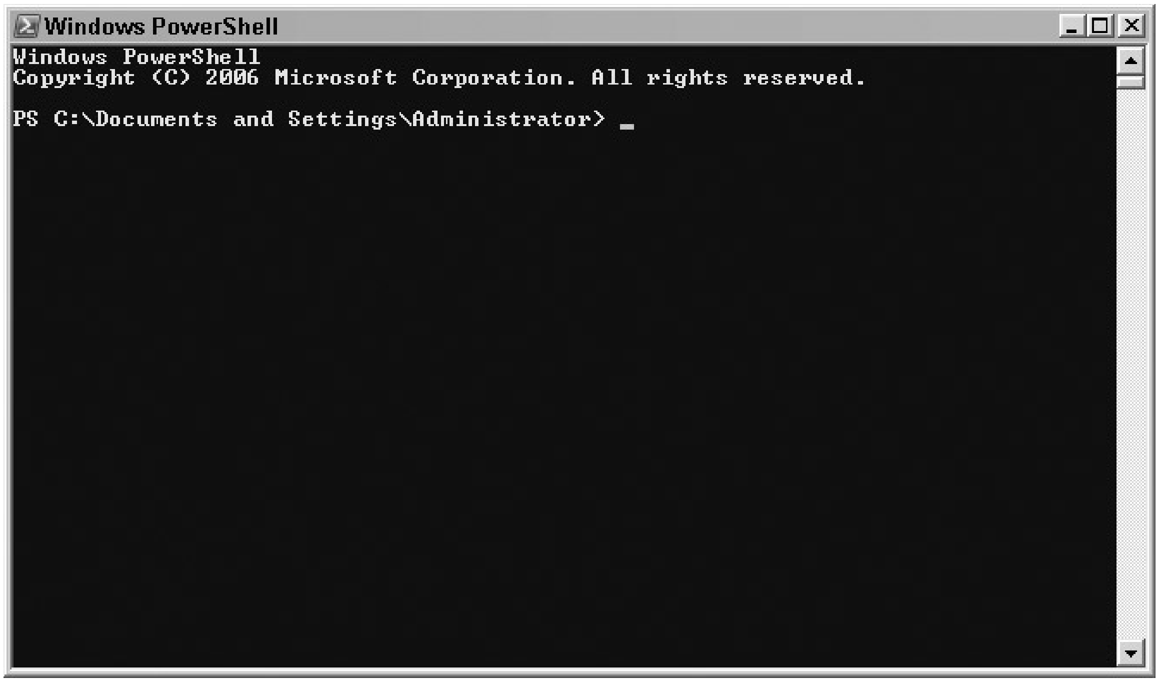
The Command-Line Interface

Windows PowerShell operates within a hosting application. The default application is `powershell.exe`, a console application that presents a command line to the user. To start PowerShell from the Start menu select All Programs ⇨ Windows PowerShell 1.0 ⇨ Windows PowerShell. This opens Windows PowerShell with the default console application as shown in Figure 1-1.

Many first-time users of Exchange Management Shell may be confused when after opening the default Windows PowerShell console application that they are unable to run any Exchange-specific commands. This is because Exchange Management Shell is an extension of Windows PowerShell. The default Windows PowerShell hosting application does not include any Exchange-specific commands. Windows PowerShell is extended by the use of a component called a snap-in. A snap-in provides a method for loading custom PowerShell commands and functionality contained in an application extension file.

To start Exchange Management Shell from the Start menu, select All Programs ⇨ Exchange Server 2007 ⇨ Exchange Management Shell. The target definition for this program shortcut contains the following underlying command line:

```
C:\WINDOWS\system32\WindowsPowerShell\v1.0\PowerShell.exe -PSConsoleFile
"C:\Program Files\Microsoft\Exchange Server\bin\exshell.psc1" -noexit -command ".
'C:\Program Files\Microsoft\Exchange Server\bin\Exchange.ps1'"
```



```
Windows PowerShell
Copyright (C) 2006 Microsoft Corporation. All rights reserved.

PS C:\Documents and Settings\Administrator> _
```

Figure 1-1

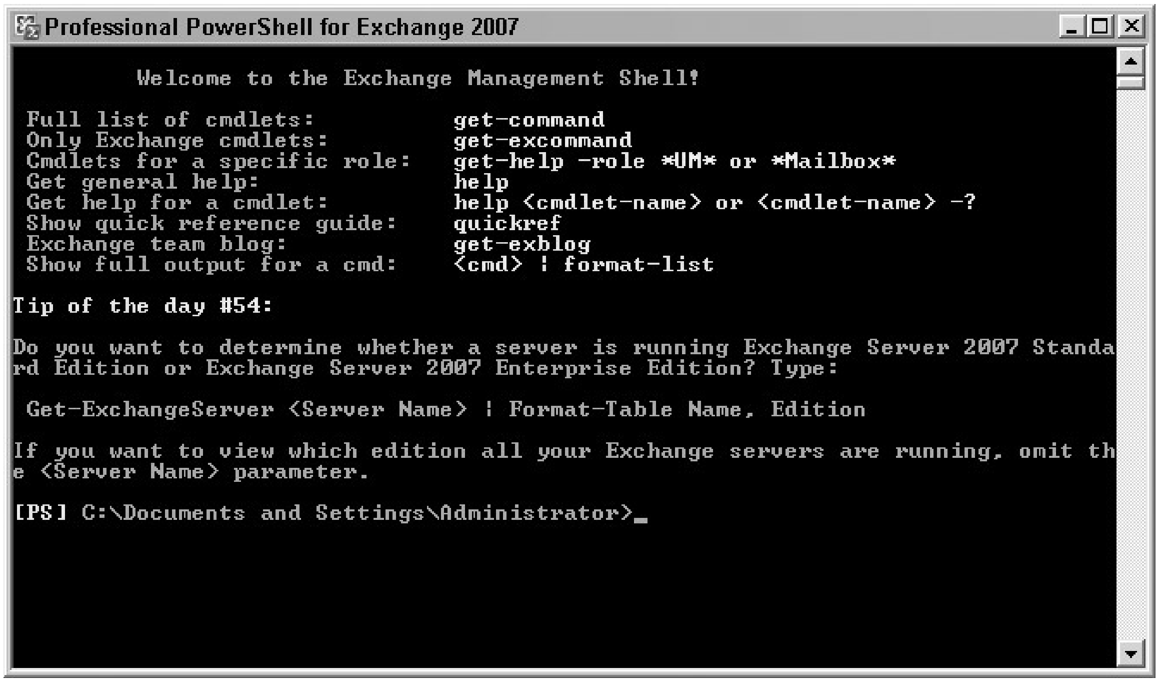
While invoking Windows PowerShell, this command specifies a console definition file identified by the `PSConsoleFile` parameter. The `exshell.psc1` file contains a pointer to the Exchange Management Shell snap-in definition stored in the Registry at `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\PowerShell\1\PowerShellSnapIns\Microsoft.Exchange.Management.PowerShell.Admin`.

The `ModuleName` value stored in this location contains the path to the application extension file `Microsoft.Exchange.PowerShell.Configuration.dll`, located in the `%ProgramFiles%\Microsoft\Exchange Server\Bin` directory. Windows PowerShell loads this `.dll` file to make the Exchange commands available.

In addition to loading the snap-in for Exchange Management Shell, the underlying command also uses the `command` parameter to specify additional commands to run at startup, in this case the script file `Exchange.ps1`. This script file contains definitions for aliases, functions, and variables specific to

Exchange management. It also defines the appearance of the command-line prompt and the initial welcome banner shown in Figure 1-2.

As you can see, the appearance of the Exchange Management Shell is a bit different from the default Windows PowerShell console application, yet all the functionality of the core shell remains intact. Because Windows PowerShell is hosted in a console application, all the familiar properties and controls for a console application are available. Later in this section you learn how to set up your Exchange Management Shell for the best user experience when following the examples in this book.



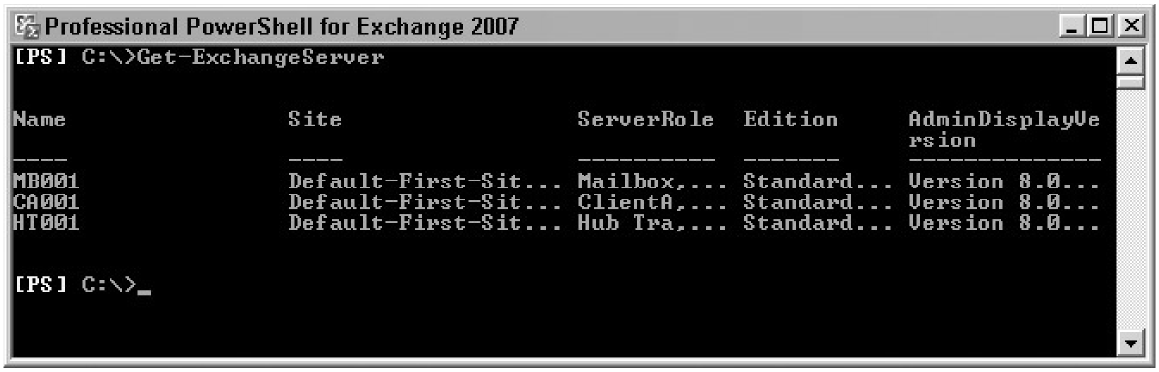
```
         Welcome to the Exchange Management Shell!

 Full list of cmdlets:           get-command
 Only Exchange cmdlets:          get-excommand
 Cmdlets for a specific role:    get-help -role *UM* or *Mailbox*
 Get general help:               help
 Get help for a cmdlet:          help <cmdlet-name> or <cmdlet-name> -?
 Show quick reference guide:     quickref
 Exchange team blog:             get-exblog
 Show full output for a cmd:     <cmd> | format-list

Tip of the day #54:

Do you want to determine whether a server is running Exchange Server 2007 Standa
rd Edition or Exchange Server 2007 Enterprise Edition? Type:

 Get-ExchangeServer <Server Name> | Format-Table Name, Edition

If you want to view which edition all your Exchange servers are running, omit th
e <Server Name> parameter.

[PS] C:\Documents and Settings\Administrator>_
```

Figure 1-2

Cmdlets

The most basic component of Windows PowerShell is the built-in commands, called cmdlets (pronounced command-lets). Almost all the work done through Windows PowerShell is done through the use of cmdlets. Cmdlets are similar to built-in commands found in other shells; for example, the built-in command `DIR` found in `cmd.exe`. In Exchange Management Shell, cmdlets that perform a specific administrative function are often referred to as tasks.

All cmdlets share the same basic structure. They have a name and take one or more parameters as input. Entering the name of a cmdlet, followed by any necessary parameter names and values, will result in the execution of the cmdlet. For example, the cmdlet `Get-ExchangeServer` returns a list of all Exchange servers in the organization in a formatted list as shown in Figure 1-3.



```
[PS] C:\>Get-ExchangeServer


Name                Site                 ServerRole  Edition     AdminDisplayVe
                                                                 rsion
----                ----                 ----------  -------     --------------
MB001               Default-First-Sit... Mailbox,... Standard... Version 8.0...
CA001               Default-First-Sit... ClientA,... Standard... Version 8.0...
HT001               Default-First-Sit... Hub Tra,... Standard... Version 8.0...


[PS] C:\>_
```

Figure 1-3

Windows PowerShell commands are case-insensitive. The examples given in this section use the default form of capitalizing the first letter of each distinct word in the command elements. Only spelling and syntax count when entering Windows PowerShell commands.

Cmdlet Names: The Verb-Noun Pair

Cmdlet names always take the form of two or more words, separated by a dash or hyphen (-). The first word is known as the verb and refers to an action the cmdlet will take. The second word or group of words is known as the noun, and refers to the target of the verb. The verb and noun describe the action and the target of the action. Using this convention for naming cmdlets makes discovering and learning cmdlets more intuitive.

Cmdlet nouns may contain multiple words but have no spaces between them.

Common Verb Names

Microsoft has produced a list of common verb names recommended for use by software programmers developing Windows PowerShell cmdlets. This helps maintain a well-known list of verb names an administrator needs to know when learning about cmdlets. Here are some common verb names used in Exchange Management Shell cmdlets and what they do:

- `Get`: The `Get` verb retrieves information about the target of the cmdlet. In the previous example, `Get-ExchangeServer`, the cmdlet retrieved information about Exchange servers.
- `Set`: The `Set` verb sets a condition or makes a configuration change to the cmdlet target.
- `New`: The `New` verb creates a new instance of the cmdlet target.
- `Remove`: The `Remove` verb deletes the cmdlet target.

The `Get` *verb is the most common verb used in Exchange Management Shell cmdlets. It is also known as the default verb. When a cmdlet noun name is entered without a verb, Windows PowerShell assumes that the* `Get` *verb was implied and runs that cmdlet. In the preceding example, entering* `ExchangeServer` *instead of* `Get-ExchangeServer` *would yield the same results.*

Noun Names

Nouns always represent the target of the cmdlet, in other words the thing on which the cmdlet will act. Noun names are usually straightforward and simply describe the target item. For example, consider the cmdlet `Get-ClusteredMailboxServerStatus`. From looking at this cmdlet's name you should be able to figure out that its purpose is to retrieve the status of Clustered Mailbox Servers. When you apply this logic to other cmdlet names you quickly begin to understand how easy it can be to discover and learn cmdlets.

Another concept of noun names you should understand is that many cmdlet names share the same noun. For example, there are 10 different cmdlets that all affect mailbox items. Here are examples of just a few of these cmdlets:

- `Get-Mailbox` is used to retrieve information about one or more mailbox-enabled users.
- `Set-Mailbox` is used to change configuration settings for one or more mailbox-enabled users.
- `New-Mailbox` is used to create a new mailbox-enabled user.
- `Move-Mailbox` is used to move one or more mailboxes from one mailbox database to another.

As you can see, these examples all use a common noun name, yet each cmdlet yields very different results when it is coupled with a different verb name.

Parameters

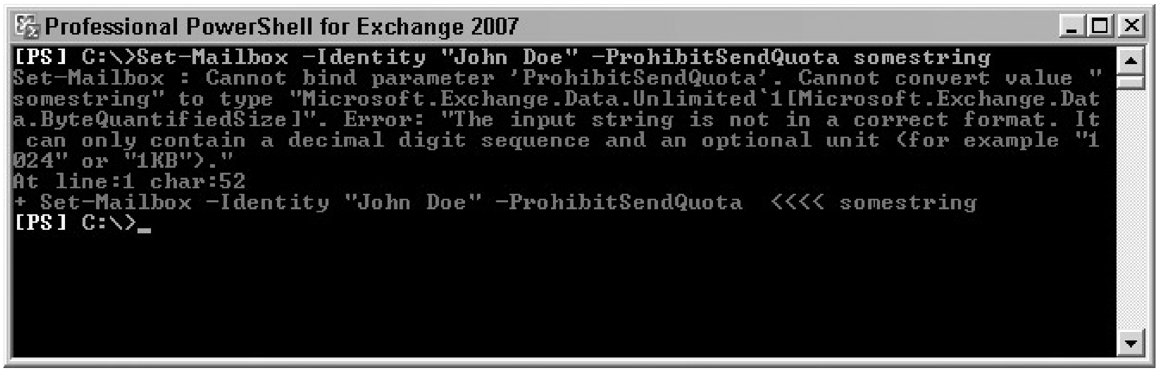
Parameter names are preceded by a dash or hyphen (-) and can be made up of a single word or multiple words with no spaces between them. Parameter names are typically followed by one or more values that are used either to provide input data for setting property values or to dictate the behavior of the cmdlet. Parameters that dictate behavior act as switches and typically do not require an input value.

Parameters have certain characteristics that determine how they are used. You can find out these characteristics via the built-in help information for each cmdlet that is readily available from the command line. Later this section covers how to get help and how to interpret that information to know how to use parameters effectively.

Parameter Input Values

Parameter input values are typically integer (numbers), string (words), or Boolean (true or false) data types. Other more specialized data types are also possible as defined by the class the cmdlet represents. For example, many cmdlets specific to Exchange Management Shell have data type input values specific to Exchange configuration components. The parameter data type is set when the cmdlet is defined. Windows PowerShell validates parameter input values as the cmdlet executes. If an invalid value is used or the format of the input data does not meet the cmdlet's specification, the cmdlet fails to execute. For example, if a parameter takes as input an integer value, but a string value is entered instead, the cmdlet fails with an error that states the wrong data type was used.

In Figure 1-4, the `Set-Mailbox` cmdlet is being used to set the `ProhibitSendQuota` attribute on mailbox-enabled user John Doe. The expected data input type for parameter `ProhibitSendQuota` is an integer value or integer value with a standard byte size abbreviation as a suffix. Because an alphanumeric string value (`somestring`) was entered instead, the command fails to execute and the error message shown describes the exact cause for the error. The solution is to provide the input value in the correct format, in this case 2GB to specify a `ProhibitSendQuota` value of 2,147,483,648 bytes.

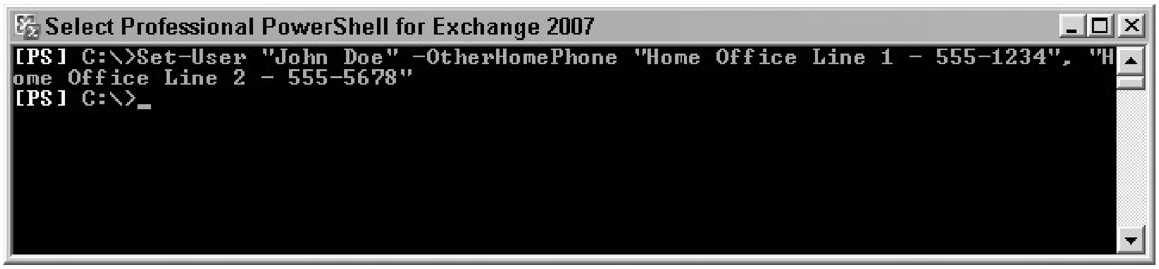


```
[PS] C:\>Set-Mailbox -Identity "John Doe" -ProhibitSendQuota somestring
Set-Mailbox : Cannot bind parameter 'ProhibitSendQuota'. Cannot convert value "
somestring" to type "Microsoft.Exchange.Data.Unlimited`1[Microsoft.Exchange.Dat
a.ByteQuantifiedSize]". Error: "The input string is not in a correct format. It
 can only contain a decimal digit sequence and an optional unit (for example "1
024" or "1KB")."
At line:1 char:52
+ Set-Mailbox -Identity "John Doe" -ProhibitSendQuota  <<<< somestring
[PS] C:\>_
```

Figure 1-4

Single-word string values can be entered as is, but string values that contain multiple words with spaces must be encapsulated in single or double quotes. Some parameters take as input multiple values. Each value must be separated by commas. When entering multiple string values with spaces, encapsulate each value in quotes, and separate each value with commas.

In Figure 1-5, the `Set-User` cmdlet is being used to set the multi-valued attribute `OtherHomePhone` with two separate string values that both contain spaces.

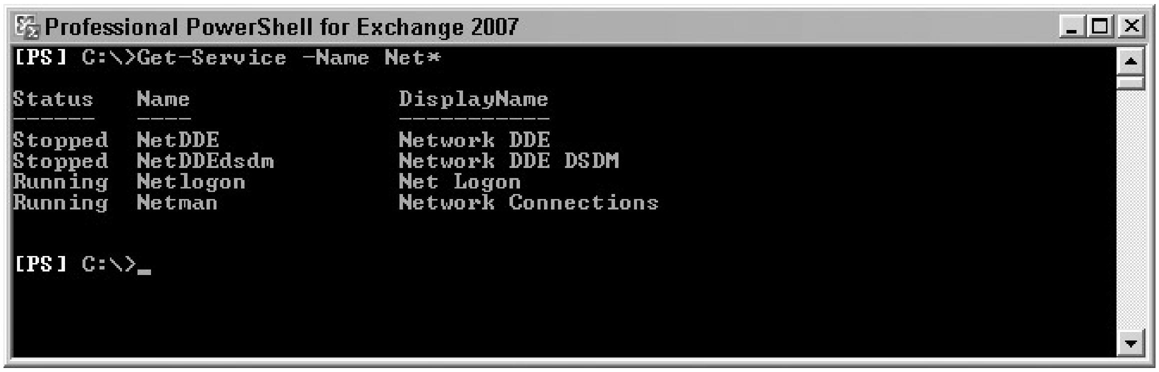


```
[PS] C:\>Set-User "John Doe" -OtherHomePhone "Home Office Line 1 - 555-1234", "H
ome Office Line 2 - 555-5678"
[PS] C:\>_
```

Figure 1-5

Some parameters support wildcards as input. Windows PowerShell handles wildcard matching so all cmdlets that accept wildcard input behave the same way. The most commonly known wildcard you will find useful is the asterisk or star (*). The asterisk wildcard can be used to stand for zero or more characters in a string.

For example, the `Get-Service` cmdlet is used to gather information about services and supports wildcards for the `Name` parameter used to identify those services. Using the asterisk wildcard you can generate a list of all services with names that match the given pattern, as shown in Figure 1-6 for services that begin with `Net`.



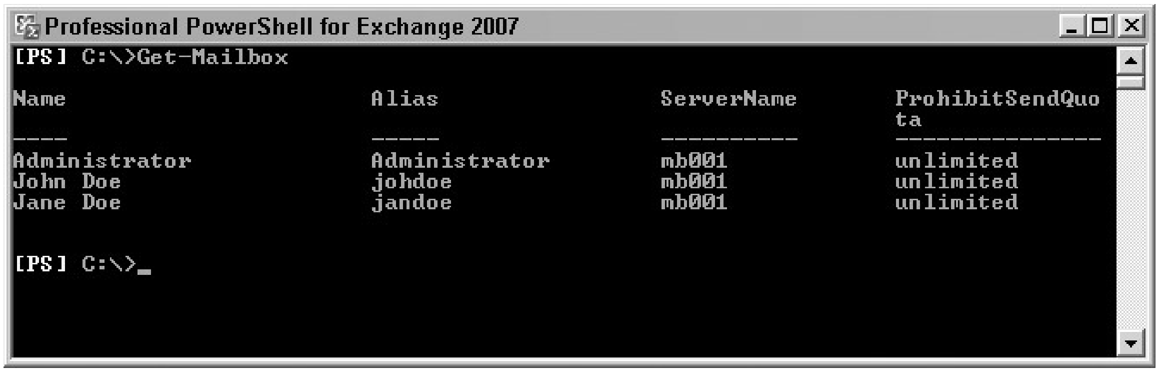
```
[PS] C:\>Get-Service -Name Net*

Status   Name               DisplayName
------   ----               -----------
Stopped  NetDDE             Network DDE
Stopped  NetDDEdsdm         Network DDE DSDM
Running  Netlogon           Net Logon
Running  Netman             Network Connections


[PS] C:\>_
```

Figure 1-6

Most cmdlets that use the `Identity` parameter support wildcards as input. Also most cmdlets that use the `Get` verb and the `Identity` parameter support a default value of `*` for the `Identity` parameter. This means that when you enter the cmdlet name without any parameters or values, it is implied you want to gather information about all the possible matches.

For example, typing and entering `Get-Mailbox` returns information about all mailbox-enabled accounts. In large organizations this could result in thousands of matches so cmdlets like `Get-Mailbox` limit the results to 1,000 matches. This can be increased by including the `ResultSize` parameter with an appropriate higher value.

In Figure 1-7, `Get-Mailbox` is used to retrieve all mailboxes in the organization.



```
[PS] C:\>Get-Mailbox

Name                      Alias                ServerName       ProhibitSendQuo
                                                                ta
----                      -----                ----------       ---------------
Administrator             Administrator        mb001            unlimited
John Doe                  johdoe               mb001            unlimited
Jane Doe                  jandoe               mb001            unlimited


[PS] C:\>_
```

Figure 1-7

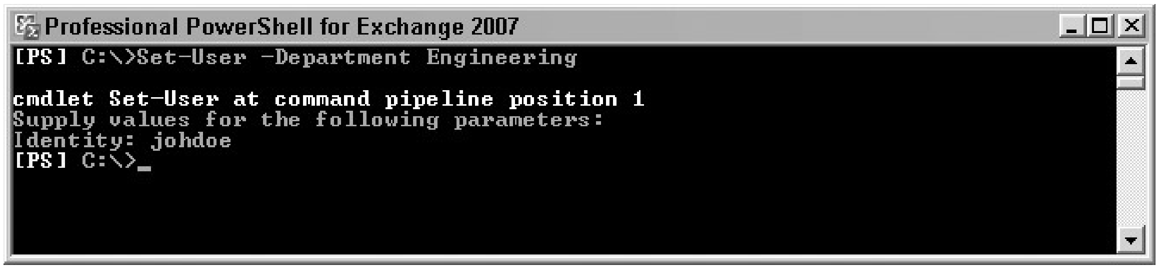
Optional and Required Parameters

Cmdlets may have some parameters that are not required to be used each time the cmdlet is run and are considered optional. You will find that most cmdlets have at least some optional parameters, especially cmdlets that modify items, because not all properties of an item require changing at the same time. This allows you to use only the optional parameters necessary to make the desired changes while leaving out all other optional parameters.

Then there are other parameters that must always be used when the cmdlet is run. You will find that most cmdlets that take an action such as creating, modifying, or removing items have at minimum one required parameter to identify the items on which to take action. If any required parameters are left out when running a cmdlet, Windows PowerShell prompts the user to enter an input value for each of the missing required parameters.

The `Identity` parameter is one of the most common required parameters, typically used by cmdlets that need as input the name of the object on which to take some action. For example, the `Set-User` cmdlet modifies attributes on an existing user account in the Active Directory directory service. The `Identity` parameter is required when using `Set-User` and is used to identify the user account on which the changes are to be made.

In Figure 1-8, the `Set-User` cmdlet is being used to set the `Department` attribute, but the `Identity` parameter was not used to name the target user so the shell prompts the operator for the missing value.



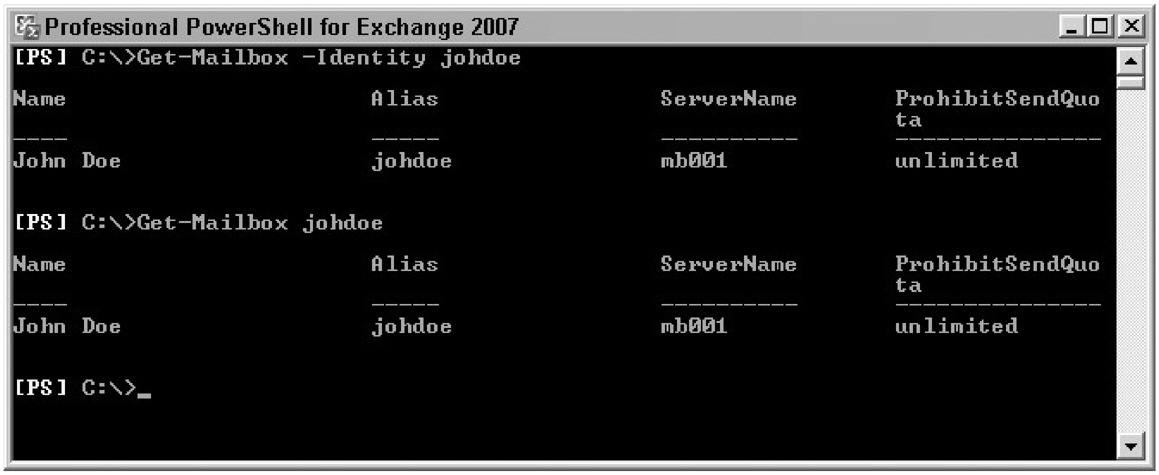
```
[PS] C:\>Set-User -Department Engineering

cmdlet Set-User at command pipeline position 1
Supply values for the following parameters:
Identity: johdoe
[PS] C:\>_
```

Figure 1-8

Positional and Named Parameters

Another parameter characteristic to consider is whether a parameter is positional or named. A positional parameter can be used without actually entering the parameter name, as long as the input value is in the position where the parameter name would normally have been used. Positional parameters are designated with a number, starting with position 1, then position 2, and so on. Using positional parameters effectively can be a real time-saving practice.

For example, the `Identity` parameter is typically a positional parameter used in position 1 after the cmdlet name. The `Get-Mailbox` cmdlet uses the `Identity` parameter in position 1 to identify the mailbox-enabled user for which to retrieve information. In Figure 1-9, you can see that the results of running the `Get-Mailbox` cmdlet with and without the `Identity` parameter name are identical as long as the input value is supplied in the first position after the cmdlet name.



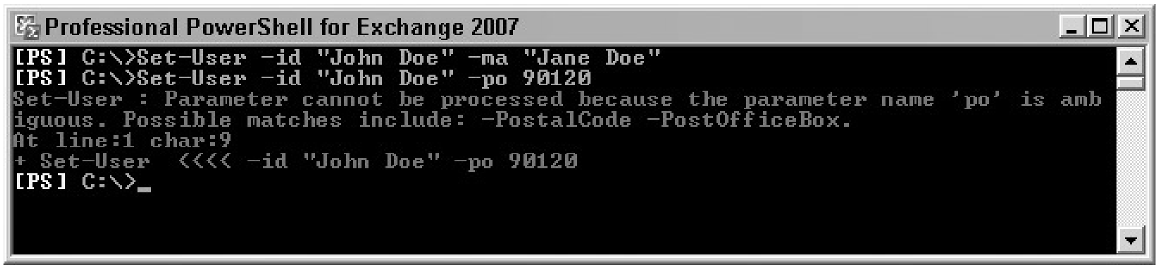
```
[PS] C:\>Get-Mailbox -Identity johdoe

Name                      Alias                ServerName       ProhibitSendQuo
                                                                ta
----                      -----                ----------       ---------------
John Doe                  johdoe               mb001            unlimited

[PS] C:\>Get-Mailbox johdoe

Name                      Alias                ServerName       ProhibitSendQuo
                                                                ta
----                      -----                ----------       ---------------
John Doe                  johdoe               mb001            unlimited

[PS] C:\>_
```

Figure 1-9

If a parameter is not positional, then it is named. To use a named parameter you must always enter the parameter name followed by the input value. The order in which you enter named parameters and their input value on the command line does not matter because the shell's command parser interprets the command in total before execution.

Parameter Shortcuts

Another time-saving feature you may find useful is parameter name shortcuts. When entering the name of a parameter, you need to supply only enough of a parameter's name to disambiguate it from any other parameter name. In the following example the first command uses the `Set-User` cmdlet to set the `Manager` attribute on user account `John Doe` to his manager `Jane Doe`. `-ma` is enough information for the shell to interpret the parameter name `Manager` so the command succeeds. In the second command, `-po` is being used to refer to the `PostalCode` parameter. However, `-po` is ambiguous and also matches parameter `PostOfficeBox`. In this case the command fails with the error shown Figure 1-10.



```
[PS] C:\>Set-User -id "John Doe" -ma "Jane Doe"
[PS] C:\>Set-User -id "John Doe" -po 90120
Set-User : Parameter cannot be processed because the parameter name 'po' is amb
iguous. Possible matches include: -PostalCode -PostOfficeBox.
At line:1 char:9
+ Set-User  <<<< -id "John Doe" -po 90120
[PS] C:\>_
```

Figure 1-10

The solution is to provide enough of the parameter name to make it unique, in this case `posta` would be enough to disambiguate `PostalCode` from `PostOfficeBox`.

Discovering Commands and Getting Help

Now that you have learned the basic concept of using cmdlets, we'll discuss how to go about discovering cmdlets and learning how to use them.

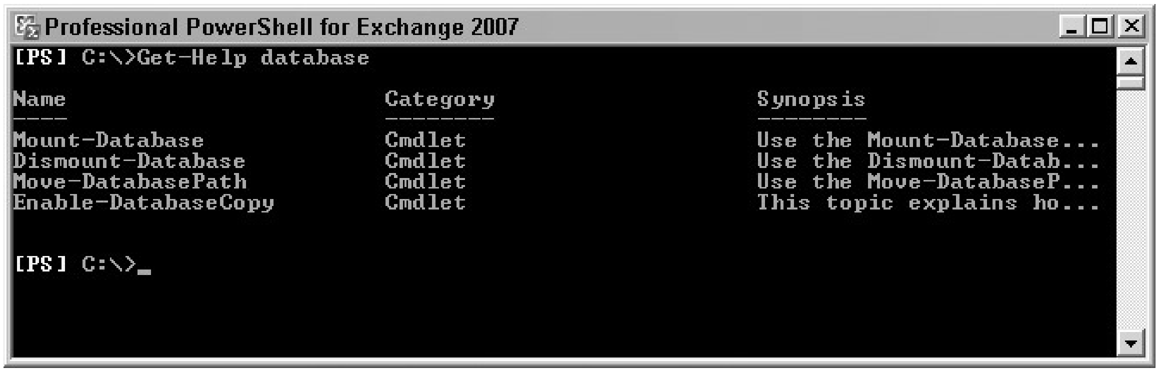
Even with more than 500 cmdlets in Exchange Management Shell, finding the right cmdlet to accomplish a task is easier than you might think. Earlier in this section you learned that a cmdlet's name is typically descriptive of the cmdlet's purpose. Using this knowledge along with some simple commands, you can quickly and easily find any cmdlet.

Using Get-Help to Find Cmdlets

Windows PowerShell provides powerful built-in help information available directly from the command line. Most cmdlets have some level of help content stored in a cmdlet help file that can be accessed from the command line using the `Get-Help` cmdlet. You don't need to know where the help file is or how to get to help information for a specific cmdlet; PowerShell works out these details as part of built-in help.

Besides displaying cmdlet help information, `Get-Help` is a powerful tool for finding cmdlets based on ambiguous name matching. When supplied with a specific and unique cmdlet name as input to the `Name` parameter, `Get-Help` displays the help information for that cmdlet. But if the input is ambiguous, `Get-Help` displays a list of all cmdlets that are a close match.

Using this approach you simply need to supply enough of the possible cmdlet name to generate a list of cmdlets from which to choose. For example, say you would like to learn about cmdlets that are used for managing Exchange databases but you don't know the exact names, or even which cmdlets might be available. The command shown in Figure 1-11 generates a list of cmdlets that contain the word `database`.

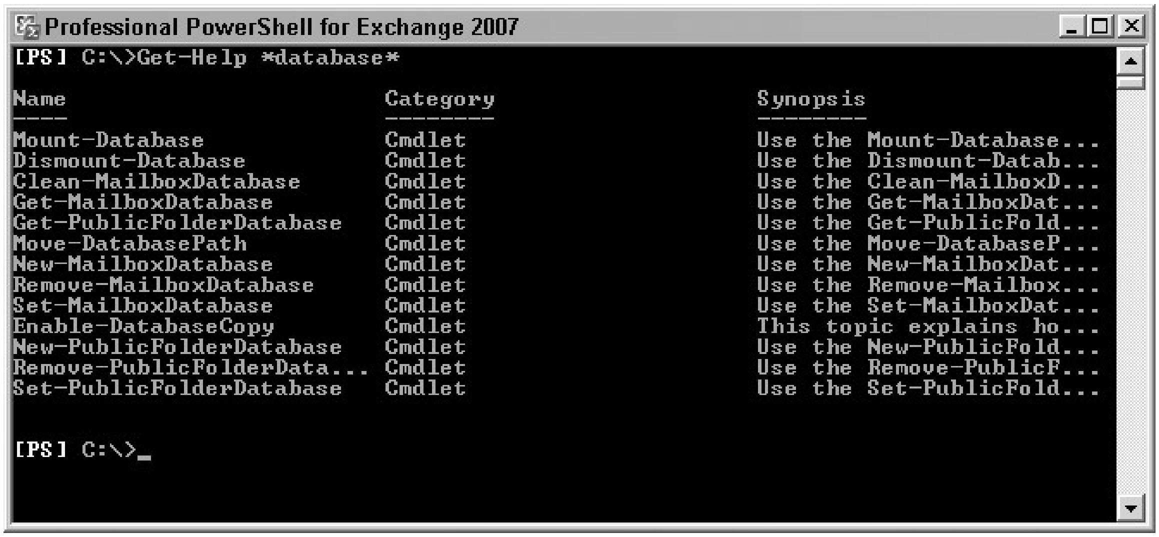


```
[PS] C:\>Get-Help database

Name                       Category                   Synopsis
----                       --------                   --------
Mount-Database             Cmdlet                     Use the Mount-Database...
Dismount-Database          Cmdlet                     Use the Dismount-Datab...
Move-DatabasePath          Cmdlet                     Use the Move-DatabaseP...
Enable-DatabaseCopy        Cmdlet                     This topic explains ho...


[PS] C:\>_
```

Figure 1-11

In this example for `Get-Help` *and for those that follow later in this section, input values are used without specifying the parameter* `Name`*. Because* `Name` *is a positional parameter (for position 1), it is not required to be named as long as the input value appears on the command line in the first position after* `Get-Help`*.*

Using the whole word `database` produces a list of cmdlets that have at least the word at the beginning of the noun name. But can you be sure that this is a list of every cmdlet possible that can be used to manage databases? Luckily, `Get-Help` supports the use of wildcards to search for matching cmdlet names. To display a list of cmdlets that have the word `database` anywhere in the cmdlet name, add the `*` wildcard to the beginning and end of the name. This causes Windows PowerShell to return a list of all possible matches as shown in Figure 1-12.

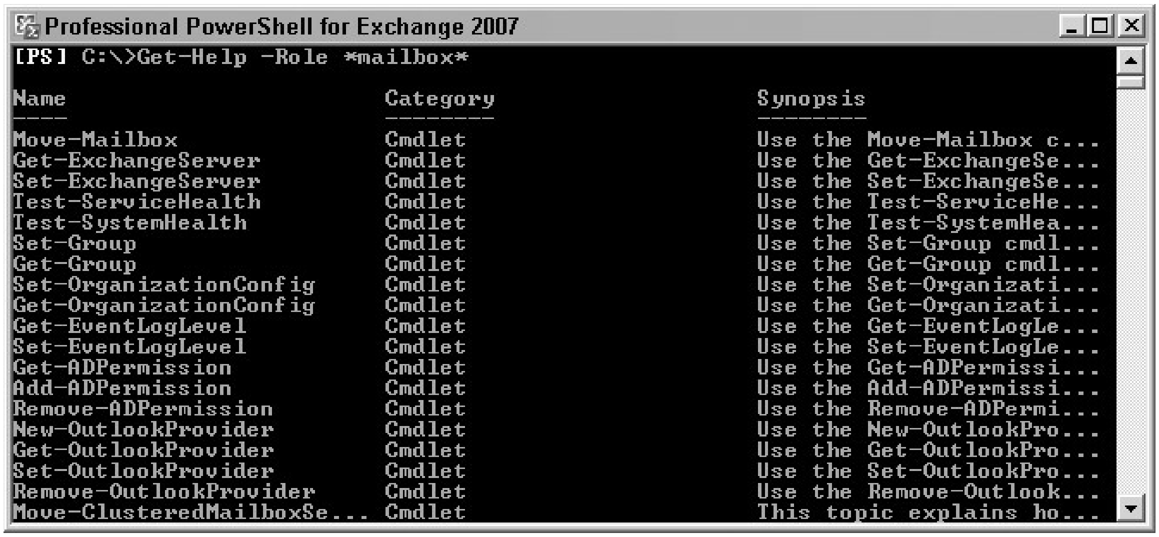


```
[PS] C:\>Get-Help *database*

Name                          Category              Synopsis
----                          --------              --------
Mount-Database                Cmdlet                Use the Mount-Database...
Dismount-Database             Cmdlet                Use the Dismount-Datab...
Clean-MailboxDatabase         Cmdlet                Use the Clean-MailboxD...
Get-MailboxDatabase           Cmdlet                Use the Get-MailboxDat...
Get-PublicFolderDatabase      Cmdlet                Use the Get-PublicFold...
Move-DatabasePath             Cmdlet                Use the Move-DatabaseP...
New-MailboxDatabase           Cmdlet                Use the New-MailboxDat...
Remove-MailboxDatabase        Cmdlet                Use the Remove-Mailbox...
Set-MailboxDatabase           Cmdlet                Use the Set-MailboxDat...
Enable-DatabaseCopy           Cmdlet                This topic explains ho...
New-PublicFolderDatabase      Cmdlet                Use the New-PublicFold...
Remove-PublicFolderData...    Cmdlet                Use the Remove-PublicF...
Set-PublicFolderDatabase      Cmdlet                Use the Set-PublicFold...


[PS] C:\>_
```

Figure 1-12

This produces a comprehensive list of all available cmdlets that deal with the management of Exchange databases. Now you would simply need to select the most likely cmdlet for accomplishing a given task based on how closely the cmdlet name describes what the cmdlet does, then access the help information for that cmdlet to learn how it is used. For example, if you want to learn how to create a mailbox database, the cmdlet `New-MailboxDatabase` is the most likely choice.

Another simple way to use `Get-Help` is with the `Role` parameter. Exchange Server 2007 architecture allows for the installation of different server roles on a given server to match the needs of an organization's messaging system. There are five server roles, and by specifying a wildcard role value with the `Role` parameter, `Get-Help` displays a list of all cmdlets used to manage that role. For example, to display all cmdlets used to manage the Mailbox server role, the command shown in Figure 1-13 would be used.



```
Professional PowerShell for Exchange 2007
[PS] C:\>Get-Help -Role *mailbox*

Name                          Category                      Synopsis
----                          --------                      --------
Move-Mailbox                  Cmdlet                        Use the Move-Mailbox c...
Get-ExchangeServer            Cmdlet                        Use the Get-ExchangeSe...
Set-ExchangeServer            Cmdlet                        Use the Set-ExchangeSe...
Test-ServiceHealth            Cmdlet                        Use the Test-ServiceHe...
Test-SystemHealth             Cmdlet                        Use the Test-SystemHea...
Set-Group                     Cmdlet                        Use the Set-Group cmdl...
Get-Group                     Cmdlet                        Use the Get-Group cmdl...
Set-OrganizationConfig        Cmdlet                        Use the Set-Organizati...
Get-OrganizationConfig        Cmdlet                        Use the Get-Organizati...
Get-EventLogLevel             Cmdlet                        Use the Get-EventLogLe...
Set-EventLogLevel             Cmdlet                        Use the Set-EventLogLe...
Get-ADPermission              Cmdlet                        Use the Get-ADPermissi...
Add-ADPermission              Cmdlet                        Use the Add-ADPermissi...
Remove-ADPermission           Cmdlet                        Use the Remove-ADPermi...
New-OutlookProvider           Cmdlet                        Use the New-OutlookPro...
Get-OutlookProvider           Cmdlet                        Use the Get-OutlookPro...
Set-OutlookProvider           Cmdlet                        Use the Set-OutlookPro...
Remove-OutlookProvider        Cmdlet                        Use the Remove-Outlook...
Move-ClusteredMailboxSe...    Cmdlet                        This topic explains ho...
```

Figure 1-13

The other possible role values you can use with the `Role` parameter are:

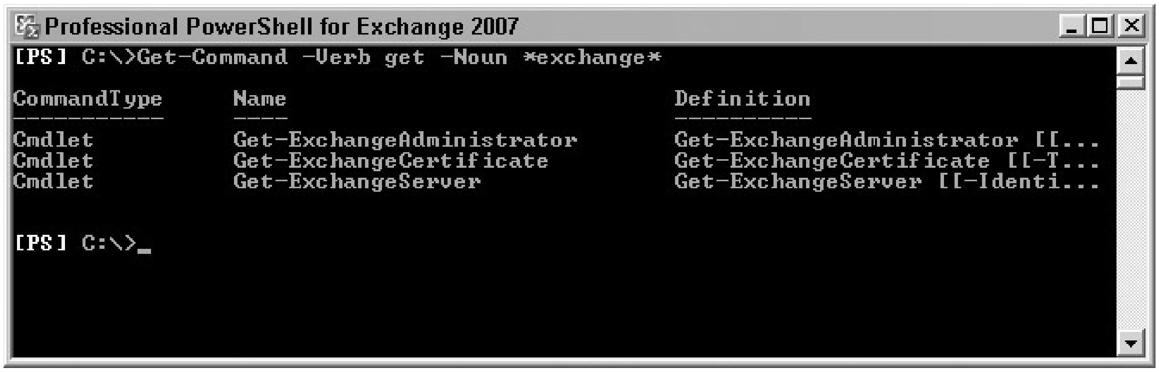
- ❑ `*client*` for Client Access Server
- ❑ `*hub*` for Hub Transport server
- ❑ `*um*` for Unified Messaging server
- ❑ `*edge*` for Edge Transport server

Using Get-Command to Find Cmdlets

In addition to the `Get-Help` cmdlet, the `Get-Command` cmdlet is very useful for discovering cmdlets and other Windows PowerShell command elements such as functions, aliases, applications, and external scripts.

Running `Get-Command` without any parameters produces a list of every available cmdlet. With more than 500 available cmdlets in Exchange Management Shell, this extensive list is not very efficient for discovering individual cmdlets. The parameters for `Get-Command` allow you to refine the list into something comprehensive. Using the `Name` parameter you can supply enough of the cmdlet name with wildcards to create a list of ambiguous matches similar to the previous example using `Get-Help`.

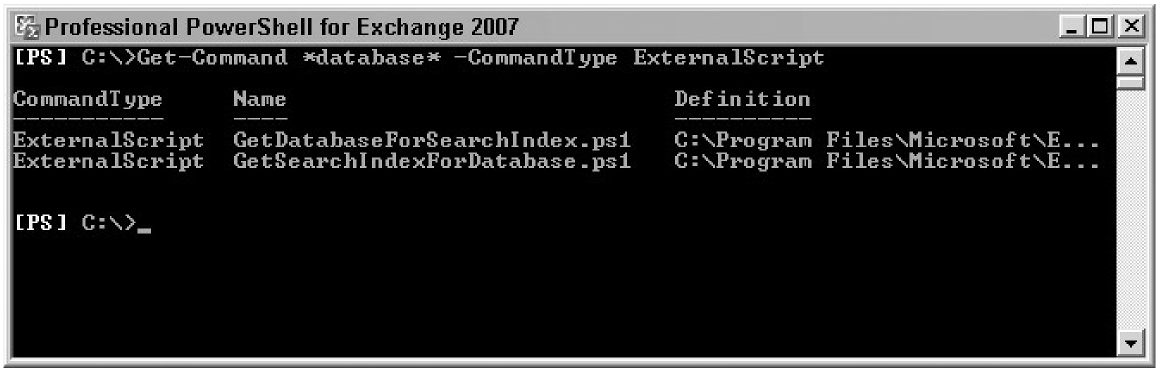
The parameters `Verb` and `Noun` are used either alone or together to search for cmdlets with matching verb and noun names. Wildcards are permitted for both of these parameters. The `Name` parameter cannot be used in conjunction with either the Verb or Noun parameters. In Figure 1-14, Get-Command is used with the Verb and Noun parameters to return a list of matching cmdlets.



```
[PS] C:\>Get-Command -Verb get -Noun *exchange*

CommandType     Name                            Definition
-----------     ----                            ----------
Cmdlet          Get-ExchangeAdministrator       Get-ExchangeAdministrator [[...
Cmdlet          Get-ExchangeCertificate         Get-ExchangeCertificate [[-T...
Cmdlet          Get-ExchangeServer              Get-ExchangeServer [[-Identi...


[PS] C:\>_
```

Figure 1-14

The CommandType parameter allows you to specify the type of command for which to return matches. Possible values are Alias, Function, Cmdlet, ExternalScript, Application, and All. Using Get-Command in this way allows you to find these additional command elements that are not exposed when searching for cmdlets using Get-Help. For example, the command in Figure 1-15 uses the CommandType parameter to find external scripts that contain the word database somewhere in their name.



```
[PS] C:\>Get-Command *database* -CommandType ExternalScript

CommandType     Name                            Definition
-----------     ----                            ----------
ExternalScript  GetDatabaseForSearchIndex.ps1   C:\Program Files\Microsoft\E...
ExternalScript  GetSearchIndexForDatabase.ps1   C:\Program Files\Microsoft\E...


[PS] C:\>_
```

Figure 1-15

ExternalScript command elements are Windows PowerShell scripts located in the %ProgramFiles%\Microsoft\Exchange Server\Scripts directory. In this example two scripts included with Exchange Server 2007 match the search criteria for names that include database.

`Get-Command` can also be used to return detailed information about the syntax of a given cmdlet using the `Syntax` parameter. However, you may find the syntax information exposed in a cmdlet's help information to be more useful in the long run because it is accompanied by other help details.

Using Help Information Effectively

Cmdlet help information is very detailed and you may find it difficult to follow when you first start learning about a given cmdlet. Luckily `Get-Help` makes it possible to access specific areas of help information in varying degrees of detail. Using `Get-Help` effectively allows you to access the information you are interested in without displaying the entire help information available for a cmdlet.

There are three versions of `Get-Help` that display help information differently depending on how they are used:

- `Get-Help` displays help information without pausing when the console display is full. Parameters are used with `Get-Help` to determine the type of information and detail level displayed. The basic syntax is `Get-Help <cmdlet name><parameters>`.
- `Help` is a function based on `Get-Help` that displays help information one screenful at a time, pausing when the console screen is full to allow the operator to advance the display either one full page using the space bar, or one line using the Enter key. The parameters available for `Get-Help` also work with `Help`. The basic syntax is `Help <cmdlet name><parameters>`.
- `-?` is a pseudo-parameter that displays basic help information without pausing when the console display is full. `-?` takes no parameters as input like the other versions of `Get-Help`. The basic syntax is `<cmdlet name> -?`.

The information contained in cmdlet help files you will find most interesting is divided into six major topics. By using certain parameters with `Get-Help`, you can display each of these topics in varying degrees of detail:

- **Synopsis:** A brief description of the cmdlet and what it does.
- **Syntax:** One or more syntax diagrams that detail the use of the cmdlet and its input parameters.
- **Detailed Description:** A more detailed description than the synopsis.
- **Parameters:** A detailed description of each parameter and how they are used.
- **Examples:** One or more examples of how the cmdlet is executed.
- **Related Links:** The names of other cmdlets that may be related in some way to this cmdlet.

The command `Get-Help <cmdlet name>` without any parameters displays the Synopsis, Syntax, Detailed Description, and Related Links topics. This is the same information displayed when using the command `<cmdlet name> -?`.

The command `Get-Help <cmdlet name> -Detailed` displays additional information about the cmdlet including descriptions of each parameter (but not details) along with the Examples topic.

The command `Get-Help <cmdlet name> -Full` displays the entire contents of the help file for the cmdlet including detailed information about each parameter.

The command `Get-Help <cmdlet name> -Examples` displays the Examples topic along with the Synopsis topic.

The command `Get-Help <cmdlet name> -Parameter <parameter name>` displays the detailed information about the specified parameter. Wildcards are permitted.

Although the descriptions and examples included in the help files are useful, you may find that the most beneficial information for learning how to use a cmdlet are the details contained in the Syntax and Parameter topics.

Syntax Details

The information included in the Syntax topic contains one or more syntax diagrams showing how the cmdlet and its parameters are used. Some cmdlets can have more than one syntax diagram depending on how the parameters work in combination with each other.

For example, the `Get-PublicFolderDatabase` cmdlet has three distinct syntax diagrams in its help file. Each diagram shows a different way to run the cmdlet depending on the parameters being used:

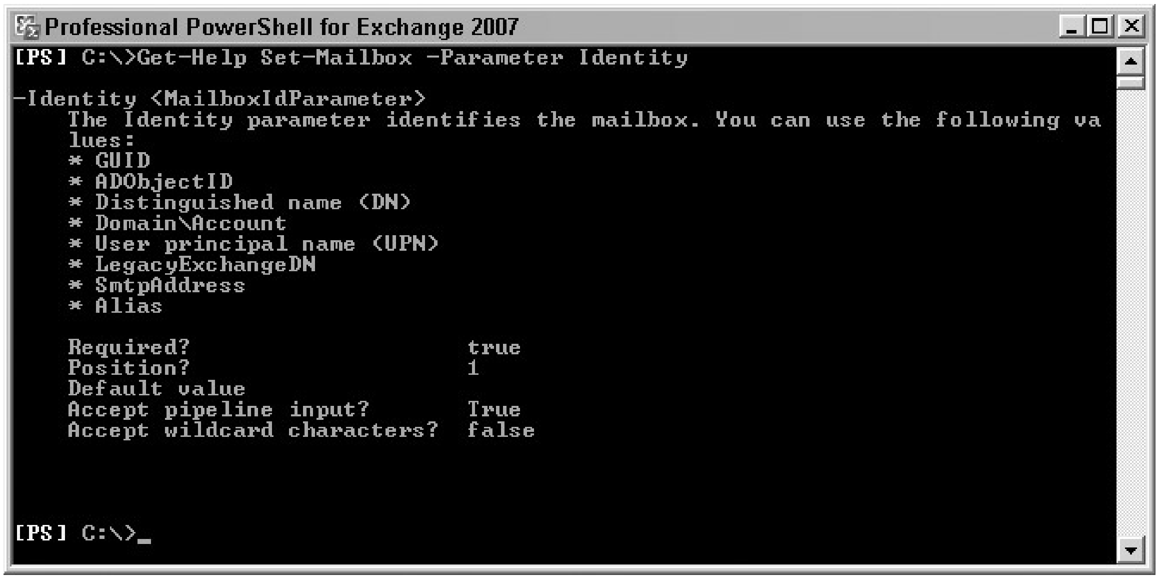
```
Get-PublicFolderDatabase [-Identity <DatabaseIdParameter>] [-DomainControll
er <Fqdn>] [-IncludePreExchange2007 <SwitchParameter>] [-Status <SwitchPara
meter>] [<CommonParameters>]

Get-PublicFolderDatabase -Server <ServerIdParameter> [-DomainController <Fq
dn>] [-IncludePreExchange2007 <SwitchParameter>] [-Status <SwitchParameter>
] [<CommonParameters>]

Get-PublicFolderDatabase -StorageGroup <StorageGroupIdParameter> [-DomainCo
ntroller <Fqdn>] [-IncludePreExchange2007 <SwitchParameter>] [-Status <Swit
chParameter>] [<CommonParameters>]
```

The `Get-PublicFolderDatabase` can be used with the `Identity` parameter to identify a specific database, the `Server` parameter to specify the server where the database is located, and the `StorageGroup` parameter to specify the storage group that holds the database. Each of these parameters is exclusive and cannot be used in combination with one another, therefore the separate syntax diagrams are necessary to show how each is used.

Parameter Details

Two levels of parameter details can be displayed using `Get-Help`. The `Detailed` parameter causes the output to include the name and description of each parameter, but omits technical details. The `Full` parameter results in the display of all parameter details. To display the full details of a single given parameter, the `Parameter` parameter is used followed by the name of the parameter. The `Detail`, `Full`, and `Parameter` parameters cannot be used in conjunction with one another.

Parameter details describe whether the parameter is required or optional and if it is positional or named. They also describe whether the parameter has a default value and if it accepts pipeline input and wildcard characters. For example, the parameter details for the `Identity` parameter as used with the `SetMailbox` cmdlet contain the information displayed in Figure 1-16 using `Get-Help` and the `Parameter` parameter.



```
[PS] C:\>Get-Help Set-Mailbox -Parameter Identity

-Identity <MailboxIdParameter>
    The Identity parameter identifies the mailbox. You can use the following va
    lues:
    * GUID
    * ADObjectID
    * Distinguished name (DN)
    * Domain\Account
    * User principal name (UPN)
    * LegacyExchangeDN
    * SmtpAddress
    * Alias

    Required?                    true
    Position?                    1
    Default value
    Accept pipeline input?       True
    Accept wildcard characters?  false



[PS] C:\>_
```

Figure 1-16

As you can see in these details, the `Identity` parameter is required (`true`) and positional (for position 1), has no default value, and accepts pipeline input (`true`) but not wildcard characters (`false`).

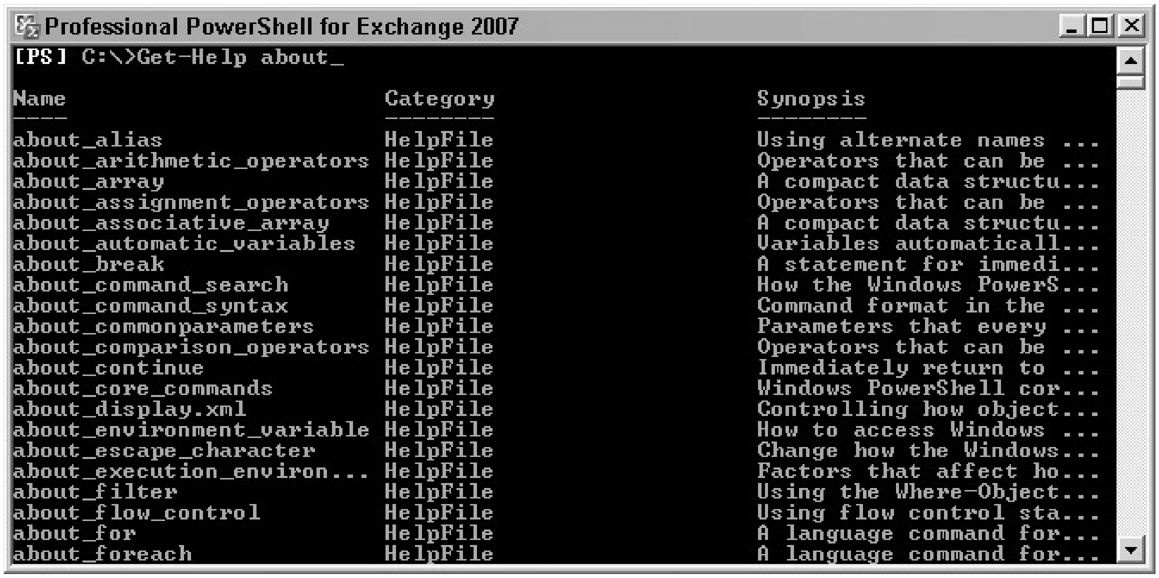
Learning More

In addition to help information for individual cmdlets, there are several supplementary help files that cover conceptual topics related to using Windows PowerShell. The names of the individual help files by and large describe the topic they cover and are prefixed with the string `about_`. To see a complete list of available topics simply type the command shown in Figure 1-17.

To access the contents of one of these help files simply enter `Get-Help about_<topic name>`. For example, to read the help file that covers the usage of wildcards in Windows PowerShell, type `Get-Help about_wildcard`.

Using Tab Expansion to Enter Cmdlets and Parameters

At this point you may be asking yourself how you will ever be able to remember exact cmdlet names and type them in without making spelling mistakes. Fortunately that is not a problem once you understand how to use the tab expansion feature of Windows PowerShell.



```
Professional PowerShell for Exchange 2007
[PS] C:\>Get-Help about_

Name                         Category                     Synopsis
----                         --------                     --------
about_alias                  HelpFile                     Using alternate names ...
about_arithmetic_operators   HelpFile                     Operators that can be ...
about_array                  HelpFile                     A compact data structu...
about_assignment_operators   HelpFile                     Operators that can be ...
about_associative_array      HelpFile                     A compact data structu...
about_automatic_variables    HelpFile                     Variables automaticall...
about_break                  HelpFile                     A statement for immedi...
about_command_search         HelpFile                     How the Windows PowerS...
about_command_syntax         HelpFile                     Command format in the ...
about_commonparameters       HelpFile                     Parameters that every ...
about_comparison_operators   HelpFile                     Operators that can be ...
about_continue               HelpFile                     Immediately return to ...
about_core_commands          HelpFile                     Windows PowerShell cor...
about_display.xml            HelpFile                     Controlling how object...
about_environment_variable   HelpFile                     How to access Windows ...
about_escape_character       HelpFile                     Change how the Windows...
about_execution_environ...   HelpFile                     Factors that affect ho...
about_filter                 HelpFile                     Using the Where-Object...
about_flow_control           HelpFile                     Using flow control sta...
about_for                    HelpFile                     A language command for...
about_foreach                HelpFile                     A language command for...
```

Figure 1-17

Most shells offer some form of automatic completion to take some of the drudgery and guesswork out of entering certain command elements. Even `cmd.exe` offers automatic completion using the tab key when typing directory paths and filenames. Windows PowerShell takes this feature to whole new levels of functionality by providing tab expansion of cmdlet and parameter names as well.

To use tab expansion when typing a cmdlet name, simply type the verb name followed by the hyphen, then the first few letters of the noun name. When you press the Tab key, Windows PowerShell automatically expands what you entered to the first matching cmdlet name. If there are other possible matches, pressing the Tab key repeatedly cycles through the available choices. Pressing the Tab key while holding down the Shift key causes Windows PowerShell to cycle backwards through the available choices. The more characters you enter before pressing the Tab key make the search more specific and narrows the number of possible matches.

For example, say you need to run the cmdlet `Get-MailboxFolderStatistics`. This cmdlet is useful for determining the size and number of items in given mailbox folders. Using tab expansion you can enter this long cmdlet name with no mistakes and a minimal number of keystrokes using the following procedure:

1. Type `get-ma` and press the Tab key. This expands to `Get-Mailbox`. Notice the name automatically changes to the standard form of uppercase first letters.
2. Now press the Tab key a second time. This time the cmdlet name expands to `Get-MailboxCalendarSettings`.
3. Press the Tab key again and the name expands to `Get-MailboxDatabase`.
4. Press the Tab key one last time to expand the name to `Get-MailboxFolderStatistics`. To continue at this point simply hit the space bar and continue typing the rest of the command.

Using this procedure you can enter complex, mistake-free cmdlet names using a minimal number of characters and Tab keystrokes. One major benefit of tab expansion is you don't have to remember exact cmdlet names as long as you can enter at least the verb name followed by a few letters of the noun name.

Tab expansion works for parameter names in the same manner. This works especially well when you don't know all the possible parameter names a cmdlet is using. To cycle through all the parameter names type a hyphen and press the Tab key repeatedly. When the correct parameter name appears, continue typing to enter the parameter value as applicable.

Be careful when using this procedure because Windows PowerShell does not validate the parameter names entered on the command line until the command is parsed at run time. Using tab expansion it is possible to inadvertently enter the same parameter name twice, causing the command to fail.

Using Cmdlet Aliases

Windows PowerShell allows you to refer to cmdlets using a shorter, simpler name called an alias. The default installation of Windows PowerShell comes complete with several predefined alias names that approximate a similar function in other command shells.

You may have already noticed that Windows PowerShell accepts `dir` as a command to display items in the current location. There is no real cmdlet called `dir`, instead it is an alias for the underlying Windows PowerShell cmdlet `Get-ChildItem`. Several other familiar command names have been defined as alias names for the matching Windows PowerShell command. To see a list of all alias definitions, run `Get-Alias`.

Windows PowerShell also allows you to define your own alias definitions using the `New-Alias` cmdlet. The lifetime of alias definitions is linked to the lifetime of the current shell session. When the shell closes the definition is lost. To learn more about aliases, type `Get-Help about_Alias`.

Using Pipelines

As mentioned earlier in this chapter, the results of running a Windows PowerShell cmdlet is a collection of one or more .NET objects. These objects have a structure that describes the properties (attributes) of the objects and the states (current value) of these properties. This feature of Windows PowerShell makes it possible to take the results of one cmdlet and pass it via pipeline as input to another cmdlet for further processing. Using a pipeline to pass data from one cmdlet to another is known as *composition*.

The vertical pipeline operator (`|`) is used to instruct Windows PowerShell to pass the collected objects from the command just prior to the pipeline to the next command. Commands can be constructed using multiple pipelines to accomplish tasks too complex for a single cmdlet to accomplish alone.

Some cmdlets that use the `Get` verb provide a way to limit the collection of objects based on a parameter value that acts as a filter. The resulting collection can then be passed by pipeline to a cmdlet that uses the `Set` verb to modify one or more properties on each object. For example, the `Get-User` cmdlet includes the `OrganizationalUnit` parameter.

Say your organization has implemented Organizational Units as a way to contain all users located in the same geographical office. A need arises to change the fax number attribute for all user accounts in the same office. Using the `Get-User` cmdlet with the appropriate value for the `OrganizationalUnit` parameter you can create a collection of user objects limited to the users in the office. By passing this collection to the `Set-User` cmdlet along with the appropriate value for the `Fax` parameter you can change the fax number quickly and easily on every user account using a single command line.

Commands that use multiple cmdlets and pipelines on a single line are often referred to as "one-liners." The following one-line command demonstrates the previous example for changing the fax number for all users contained in the "`Denver`" organizational unit:

```
[PS] C:\>Get-User -OrganizationalUnit "CN=Denver,DC=exchangeexchange,DC=local" |
Set-User -Fax 555-1234
```

Whether there are 10 or 10,000 users in the organizational unit really does not matter in this example. By collecting the user objects based on their organizational unit container with the first cmdlet, we are able to modify the fax number on all users in bulk with the second cmdlet without additional complex programming.

Filtering Objects

Not all cmdlets may provide parameters for filtering objects like the one shown in the previous example. And though some cmdlets may provide a few filtering parameters, they may not provide a parameter for the specific property you may need to use as a filter condition. That's when you need to become familiar with the `Where-Object` filter cmdlet.

`Where-Object` allows you to filter objects out of the command stream based on one or more test conditions you specify in a script block. The test conditions are based on one or more of the objects' properties. Only the objects that meet the test conditions are passed on to the next command, while all others are discarded. The most basic syntax of `Where-Object` is easy to learn:

```
<command> | Where-Object { <test condition> } | <command>
```

`Where` *and* `"?"` *are both shorthand alias names for the* `Where-Object` *cmdlet.*

The test condition is an expression that resolves to either Boolean true or false. Only the objects that resolve true when tested are passed down the pipeline to the next command. The syntax of the test condition is made up of the following elements:

```
{ $_.<property name><comparison operator><value to test><conjunction> $_.<property
name><comparison operator><value to test> ...}
```

The first element `$_` is a special variable (called an automatic variable) and is used to refer to objects in the pipeline stream. Using a technique called dot notation a property name is appended to the `$_` variable to refer to the specific object property to which the test applies. For example, `$_.Identity` refers to an object's `Identity` property.

A comparison operator is used to set the condition of the test. Windows PowerShell supports a number of named comparison operators. The most frequently used operators for comparing whole property values are `eq` (equals), `ne` (not equals), `lt` (less than), and `gt` (greater than). The `like` and `notlike` operators are used to compare string values using wildcard rules. To see a complete list of all comparison operators type `Get-Help about_Comparison_Operators` at the command line.

The value to test must be of the same data type of the property being tested. For example, if the property data type is string, a string value enclosed in quotes must be used for the test. If the property data type is integer, a numeric value must be used for the test and so on.

While Windows PowerShell validates the syntax used inside the script block, the result of entering an unknown property name or an invalid data type is a failure to pass any objects down the pipeline without reporting any error to the operator.

Multiple test conditions can be used in the same script block as long as they are separated by one or more conjunctive or disjunctive operators:

- The `and` conjunction operator is used to compare the Boolean results of two or more test conditions to render a concluding Boolean value. If any of the test conditions are true, a `true` is returned.
- The `or` disjunctive operator is used to compare the Boolean results of only two test conditions. If either one or both test conditions are false, a `false` is returned.
- Conjunctive and disjunctive groups of test conditions can be used in the same script block as long as they are enclosed in parentheses.

Now let's look at a practical example of using `Where-Object` as a filter in the pipeline stream. Say that you need to change the `Manager` property for several users based on the department to which they belong (Engineering), and the office from which they work (the Dallas office) to show they report to manager John Doe. The command would look like this:

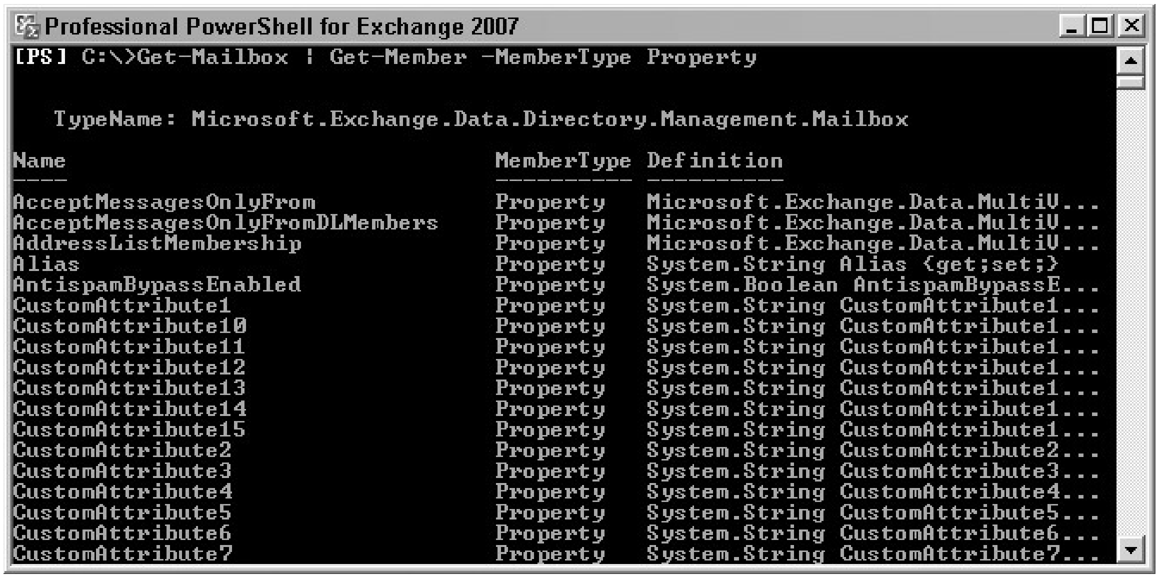
```
[PS] C:\>Get-User | Where-Object { $_.Office -eq "Dallas" -and $_.Department -eq
 "Engineering" } | Set-User -Manager "John Doe"
```

After collecting all users with `Get-User`, the collection of objects is passed to `Where-Object` to apply the test conditions on each object one at a time. The first test checks for the Office property equal to `"Dallas"`. The second condition checks the Department property equal to `"Engineering"`. If both test conditions result in true, the object is passed to the next command. If one of the test conditions results in false, the object is disposed. After processing all objects in the stream, they are passed to `Set-User` for applying the modification.

Finding Property Names and Data Types

To use `Where-Object` effectively you need to know the available property names and data types for the objects being passed by a given cmdlet. By passing the results of any cmdlet that uses the `Get` verb to the `Get-Member` cmdlet you can generate a list of properties and their data type. For example, the command in Figure 1-18 sends the objects collected by the `Get-Mailbox` cmdlet to `Get-Member` and displays the objects' properties and their definition.

It is important to know the exact name of each property used in the `Where-Object` script block. Mistyping a property name results in a failure to pass any objects down the pipeline stream without reporting any error to the operator.

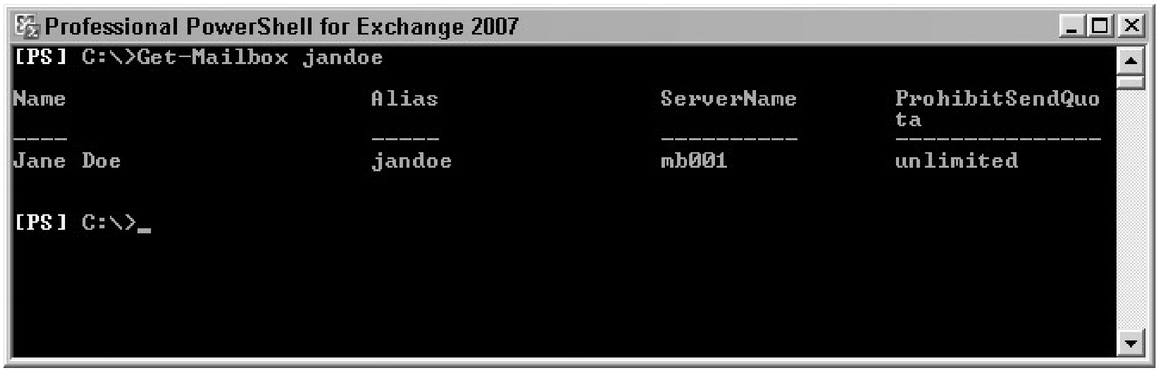


```
[PS] C:\>Get-Mailbox | Get-Member -MemberType Property


   TypeName: Microsoft.Exchange.Data.Directory.Management.Mailbox

Name                               MemberType Definition
----                               ---------- ----------
AcceptMessagesOnlyFrom             Property   Microsoft.Exchange.Data.MultiV...
AcceptMessagesOnlyFromDLMembers    Property   Microsoft.Exchange.Data.MultiV...
AddressListMembership              Property   Microsoft.Exchange.Data.MultiV...
Alias                              Property   System.String Alias {get;set;}
AntispamBypassEnabled              Property   System.Boolean AntispamBypassE...
CustomAttribute1                   Property   System.String CustomAttribute1...
CustomAttribute10                  Property   System.String CustomAttribute1...
CustomAttribute11                  Property   System.String CustomAttribute1...
CustomAttribute12                  Property   System.String CustomAttribute1...
CustomAttribute13                  Property   System.String CustomAttribute1...
CustomAttribute14                  Property   System.String CustomAttribute1...
CustomAttribute15                  Property   System.String CustomAttribute1...
CustomAttribute2                   Property   System.String CustomAttribute2...
CustomAttribute3                   Property   System.String CustomAttribute3...
CustomAttribute4                   Property   System.String CustomAttribute4...
CustomAttribute5                   Property   System.String CustomAttribute5...
CustomAttribute6                   Property   System.String CustomAttribute6...
CustomAttribute7                   Property   System.String CustomAttribute7...
```

Figure 1-18

Controlling Output

When you run a cmdlet in Windows PowerShell, the type of data displayed as output, if any, is determined by the default format of the cmdlet. The output format is determined at the time the cmdlet is created. Most cmdlets that use the `Get` verb have some form of default display formatting that results in what the programmer determined the most useful information. For example, running the `Get-Mailbox` cmdlet for a specific user results in the output shown in Figure 1-19.



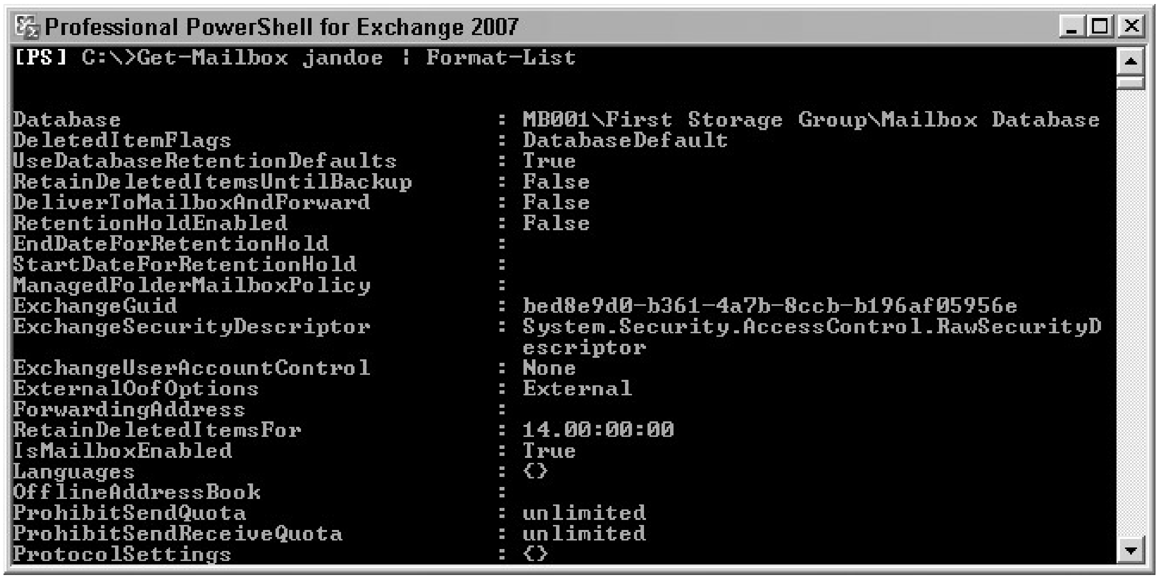
```
[PS] C:\>Get-Mailbox jandoe

Name                      Alias                    ServerName        ProhibitSendQuo
                                                                      ta
----                      -----                    ----------        ---------------
Jane Doe                  jandoe                   mb001             unlimited


[PS] C:\>_
```

Figure 1-19

At times you need to see more specific information not included in the default format. By using the pipeline operator to pass objects to the `Format-List` and `Format-Table` cmdlets you can control a cmdlet's output to see either all properties or only those properties you specify.

Format-List

In the previous example `Get-Mailbox` displayed only those properties specified as default output for the cmdlet. By passing the object to `Format-List` (or its alias name `fl`) without any additional parameters, all properties are displayed in a list format as shown in Figure 1-20.

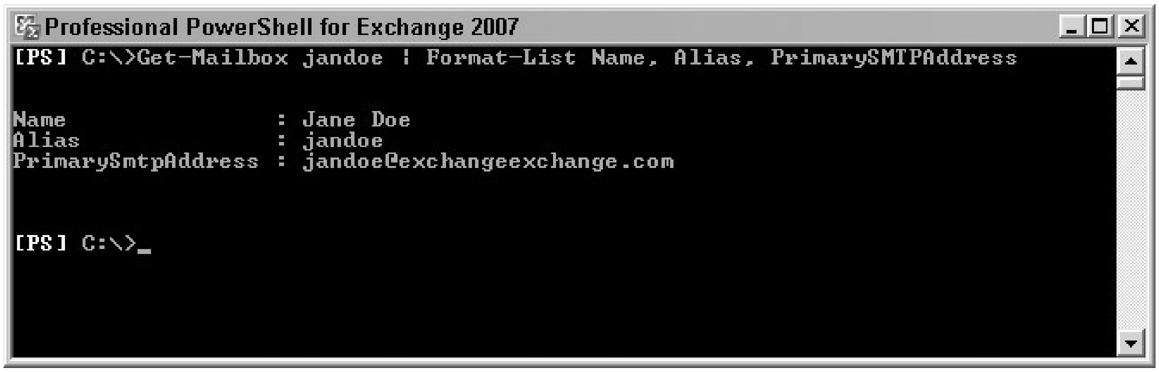


Professional PowerShell for Exchange 2007

```
[PS] C:\>Get-Mailbox jandoe | Format-List


Database                                  : MB001\First Storage Group\Mailbox Database
DeletedItemFlags                          : DatabaseDefault
UseDatabaseRetentionDefaults              : True
RetainDeletedItemsUntilBackup             : False
DeliverToMailboxAndForward                : False
RetentionHoldEnabled                      : False
EndDateForRetentionHold                   :
StartDateForRetentionHold                 :
ManagedFolderMailboxPolicy                :
ExchangeGuid                              : bed8e9d0-b361-4a7b-8ccb-b196af05956e
ExchangeSecurityDescriptor                : System.Security.AccessControl.RawSecurityD
                                            escriptor
ExchangeUserAccountControl                : None
ExternalOofOptions                        : External
ForwardingAddress                         :
RetainDeletedItemsFor                     : 14.00:00:00
IsMailboxEnabled                          : True
Languages                                 : {}
OfflineAddressBook                        :
ProhibitSendQuota                         : unlimited
ProhibitSendReceiveQuota                  : unlimited
ProtocolSettings                          : {}
```

Figure 1-20

To display only specific properties, add the property names, separated by commas, after `Format-List`. For example, if you only want to see the `Name`, `Alias`, and `PrimarySMTPAddress` properties for user `jandoe`, you would then run the command shown in Figure 1-21.



Professional PowerShell for Exchange 2007

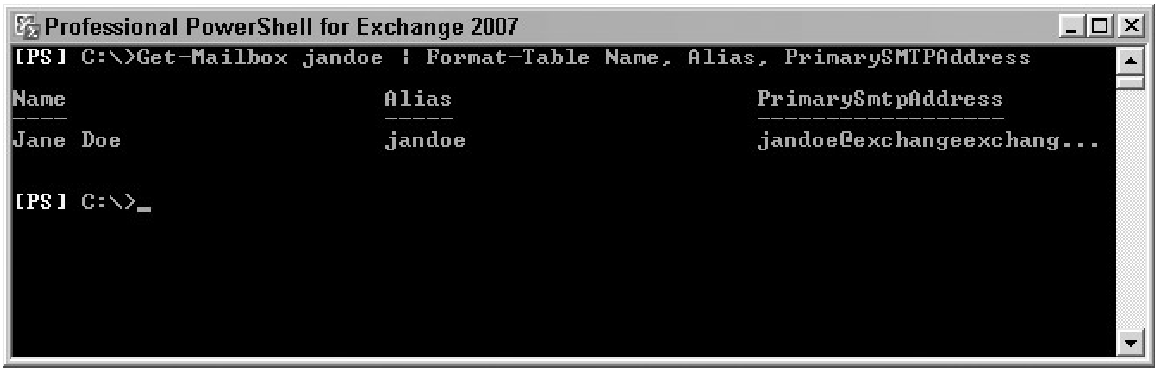
```
[PS] C:\>Get-Mailbox jandoe | Format-List Name, Alias, PrimarySMTPAddress


Name               : Jane Doe
Alias              : jandoe
PrimarySmtpAddress : jandoe@exchangeexchange.com



[PS] C:\>_
```

Figure 1-21

Format-Table

Format-Table (and its alias, ft) is similar to Format-List except it allows you to format output in table format. Unlike Format-List, passing objects to Format-Table without any additional parameters results in one of two displays. If the default format for displaying output is table format of select properties, the default format is used for display. If the default format is a list of all properties, Windows PowerShell attempts to display as many of the properties as possible in table format. This is usually very impractical so Format-Table is typically used with a list of properties to display.

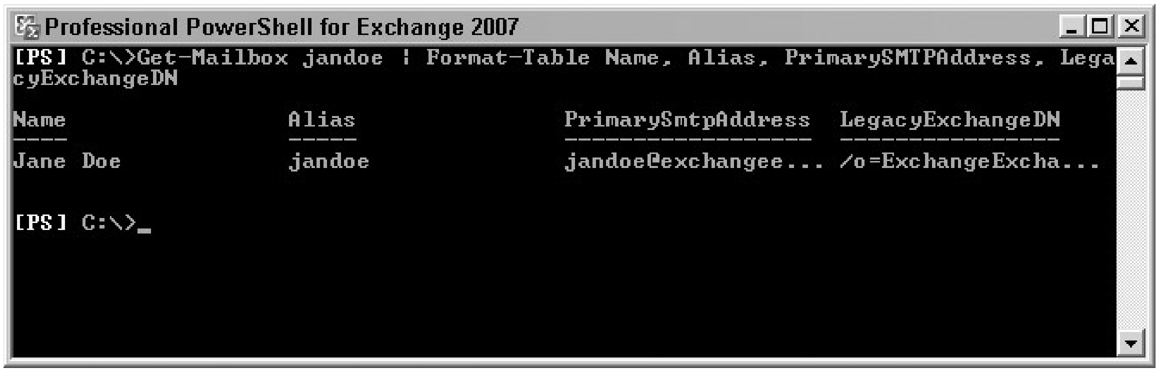
If Format-Table is used in place of Format-List in the previous example, the resulting display would look like Figure 1-22.



```
[PS] C:\>Get-Mailbox jandoe | Format-Table Name, Alias, PrimarySMTPAddress

Name                          Alias                         PrimarySmtpAddress
----                          -----                         ------------------
Jane Doe                      jandoe                        jandoe@exchangeexchang...


[PS] C:\>_
```

Figure 1-22

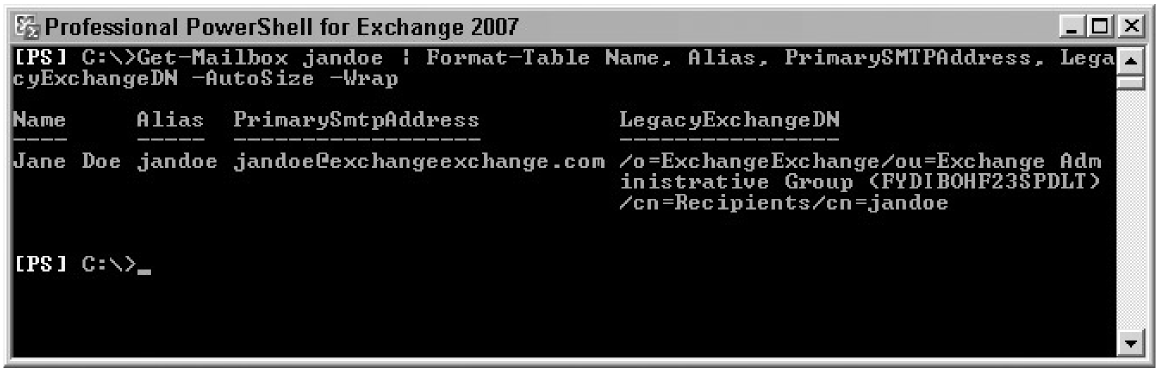
As more properties are specified, or if property values are longer than can be displayed on a single line, Windows PowerShell truncates the output with ellipses to indicate more information is available as shown in Figure 1-23 where the LegacyExchangeDN property has been added to the previous command.



```
[PS] C:\>Get-Mailbox jandoe | Format-Table Name, Alias, PrimarySMTPAddress, Lega
cyExchangeDN

Name                    Alias                   PrimarySmtpAddress  LegacyExchangeDN
----                    -----                   ------------------  ----------------
Jane Doe                jandoe                  jandoe@exchangee... /o=ExchangeExcha...


[PS] C:\>_
```

Figure 1-23

To change this behavior, add the `AutoSize` parameter to force `Format-Table` to change column widths to make the most of the console screen width, and the `Wrap` parameter to wrap long values that won't fit on a single line to the next line. The addition of these two parameters to the previous example yields the results shown in Figure 1-24.



```
[PS] C:\>Get-Mailbox jandoe | Format-Table Name, Alias, PrimarySMTPAddress, Lega
cyExchangeDN -AutoSize -Wrap

Name     Alias  PrimarySmtpAddress          LegacyExchangeDN
----     -----  ------------------          ----------------
Jane Doe jandoe jandoe@exchangeexchange.com /o=ExchangeExchange/ou=Exchange Adm
                                            inistrative Group (FYDIBOHF23SPDLT)
                                            /cn=Recipients/cn=jandoe


[PS] C:\>_
```

Figure 1-24

Running Scripts

As you learn to use Exchange Management Shell to manage your Exchange organization you will most likely identify several command sequences that you run on a regular basis to accomplish some task. Store these commands in a Windows PowerShell script file so you can run them all by simply executing the script file. Use your favorite text editing software to create and edit script files. Windows PowerShell script files use `.ps1` as the file extension name.

To run a script, type its name at the command line. You do not have to include the `.ps1` file extension. However, you do have to pay attention to the drive location where the script is stored and the current location from which the script is being run. You must supply the full path to the script file even if the script is stored in the current location. To tell Windows PowerShell the script is in the current directory, either type the full path or use a dot and backslash (.\) to indicate the current directory as shown in this example:

```
[PS] C:\scripts>.\myscript
```

Exchange Management Shell provides a default directory for storing several script files provided with Exchange Server 2007. You do not have to provide the full path name when running any script located in the `%ProgramFiles%\Microsoft\Exchange Server\scripts` directory because this path is stored in the Windows system variable path statement as part of Exchange Server installation. By placing your script files in this directory you can keep them in a known directory and run them from any drive location without providing the full path.

The chance that a script may include destructive code may raise security concerns among administrators. Windows PowerShell provides a method for applying a security policy for controlling which scripts are allowed to run on a machine. The execution policy determines whether or not scripts are allowed to run, and whether they must include a digital signature that verifies the origin of the script and if it has been tampered with in any way since it was digitally signed by its creator.

All scripts included with Exchange Server 2007 have been code signed by Microsoft to ensure the scripts comply with the execution policy model for ensuring scripts can be accounted for before execution. The default execution policy setting for Exchange Management Shell is `RemoteSigned`. This level allows you to run scripts you create locally and warns you when scripts provided by Microsoft have been altered.

To learn more about Windows PowerShell execution policies, type `Get-Help about_signing`.

Preparing Exchange Management Shell

Before continuing on to the rest of the book, take a moment to review the following procedure for customizing your Exchange Management Shell console application. You'll find these options convenient when trying the examples shown in the following chapters.

1. Navigate to the Exchange Management Shell shortcut: from the Start menu, select All Programs ➪ Exchange Server 2007 and then right-click Exchange Management Shell and select Properties.
2. Select the Options tab and make the following modifications:
   a. To make it possible to select, copy, and paste text in the console screen, under Edit Options click to select the QuickEdit Mode checkbox. With this option selected, you can select text in the console window by dragging the left mouse button. Copy the selected text to the clipboard using the right mouse button or by pressing Enter.
   b. The Insert Mode checkbox is typically already selected, but make sure it is checked as well. This option allows you to paste text into the command line by positioning the cursor at the desired position, then using the right mouse button to paste the contents of the clipboard.
   c. Under Command History set the Buffer Size to at least 100. This number determines the number of commands stored in the console buffer. Previously entered commands can be recalled by using the up and down cursor keys. Click to select the Discard Old Duplicate checkbox to automatically discard any duplicate commands from the console buffer.
3. Select the Layout tab and make the following modifications:
   a. Under Screen Buffer Size change the Height setting to 9,999. This setting determines the number of lines of output held in the console buffer you can view using the console window scroll control.
   b. Under Window Size, set the Width setting to a number between 80 and 120. This setting determines the number of characters displayed across the console window. Though a higher number setting allows you to type more characters before wrapping to the next line, the default output format of most Exchange Management Shell cmdlets is based on an 80-character display. If you change this number, make sure the value for Width under Screen Buffer matches.
4. Click OK to commit these changes and close Properties. The changes take effect the next time Exchange Management Shell is started.

Summary

- Windows PowerShell is the next-generation command-line shell and scripting language for Windows. Exchange Server 2007 is the first Microsoft application to utilize Windows PowerShell for deployment and administration.
- Command shells provide a more flexible administrative interface compared to Graphical User Interfaces (GUIs). Administrators use scripts to automate everyday tasks and resolve issues GUI interfaces are not able to handle.
- Windows PowerShell is built on top of .NET Framework version 2.0 and exposes .NET classes as built-in commands. Actions in Windows PowerShell are based on .NET objects that carry their structure definition as well as the current state of their attributes. Windows PowerShell objects have properties (which are characteristics) and methods (which are actions that you can take) and can be passed from one command to another without the need for parsing.
- Exchange Management Shell extends Windows PowerShell to include more than 500 built-in commands. Exchange Management Console is a GUI management application built on top of Windows PowerShell.
- The most basic component of Windows PowerShell is the built-in commands called cmdlets. Cmdlet names are made up of a verb name that identifies the action to take and the noun name that identifies the object on which to take action. Cmdlets use named parameters to identify individual properties or control how the cmdlet executes.
- Windows PowerShell includes a powerful help system available directly from the command line that makes it easy to first discover and then learn how to use cmdlets.
- The Windows PowerShell tab expansion feature takes the drudgery and guesswork out of typing commands by allowing you to automatically complete partially entered cmdlet and parameter names using the Tab key.
- Windows PowerShell makes it possible to take the results of one cmdlet and pass it via pipeline as input to another cmdlet for further processing. Using a pipeline to pass data from one cmdlet to another is known as composition.
- Command sequences that are run on a regular basis can be stored in a Windows PowerShell script file for execution. Sharing these scripts between all administrators in an organization ensures consistent results.

Further Reading

If you want a more basic understanding of general Windows PowerShell usage outside of Exchange, explore another fine Wrox publication:

Professional Windows PowerShell; Andrew Watt; ISBN: 978-0-471-94693-9; Wrox, 2004.