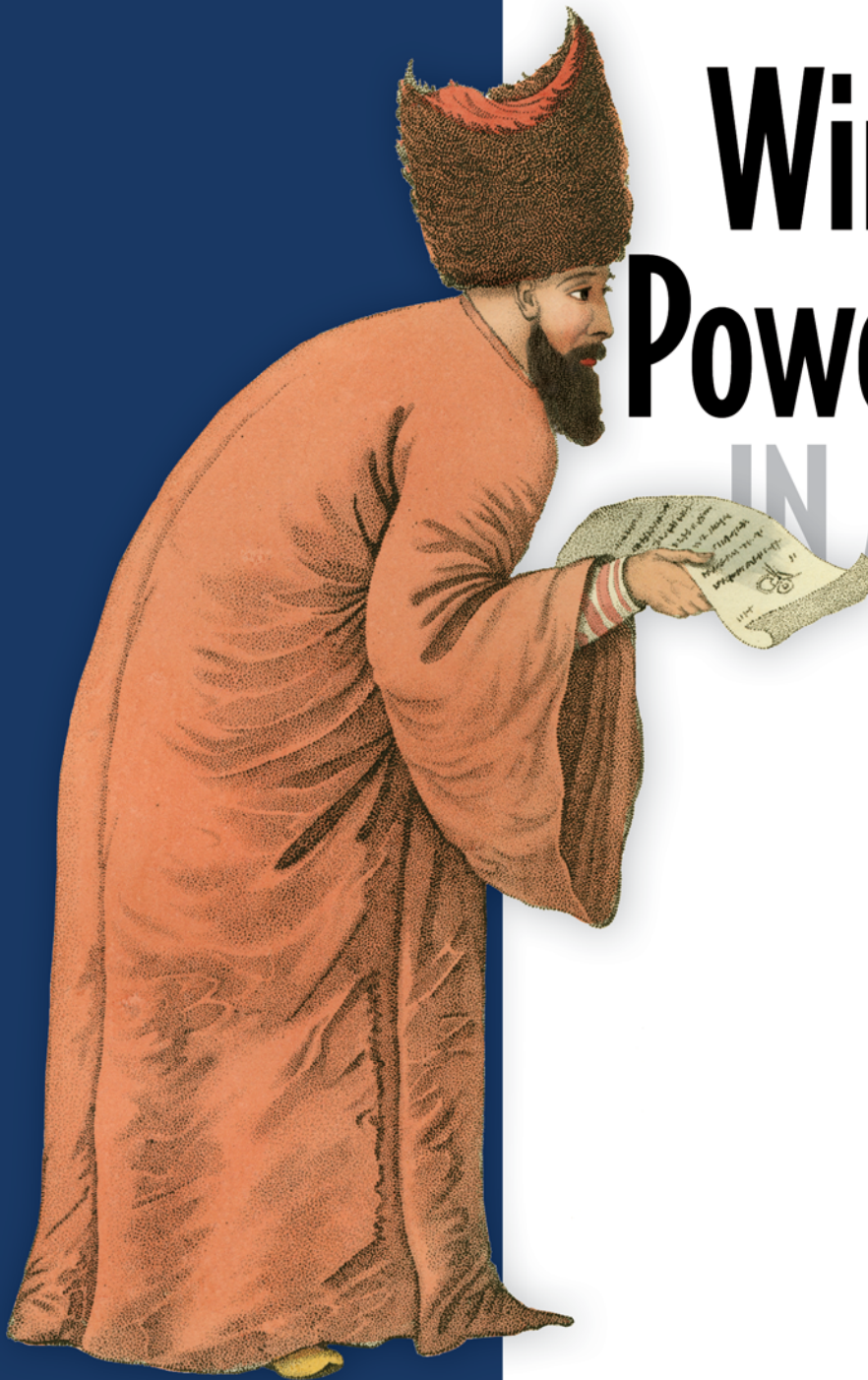


Covers PowerShell v2

APPENDIXES



Windows PowerShell IN ACTION

SECOND EDITION

Bruce Payette



Windows PowerShell in Action
Second Edition

by Bruce Payette
APPENDIXES

Copyright 2011 Manning Publications

brief contents

Part 1 Learning PowerShell 1

- 1 Welcome to PowerShell 3*
- 2 Foundations of PowerShell 36*
- 3 Working with types 72*
- 4 Operators and expressions 110*
- 5 Advanced operators and variables 151*
- 6 Flow control in scripts 198*
- 7 PowerShell functions 236*
- 8 Advanced functions and scripts 275*
- 9 Using and authoring modules 322*
- 10 Module manifests and metadata 361*
- 11 Metaprogramming with scriptblocks and dynamic code 392*
- 12 Remoting and background jobs 447*
- 13 Remoting: configuring applications and services 502*
- 14 Errors and exceptions 553*
- 15 The PowerShell ISE and debugger 606*

Part 2 Using PowerShell 661

- 16 Working with files, text, and XML 663*
- 17 Extending your reach with .NET 719*
- 18 Working with COM 760*
- 19 Management objects: WMI and WS-MAN 797*
- 20 Responding in real time with eventing 847*
- 21 Security, security, security 888*

- appendix A Comparing PowerShell to other languages 1*
- appendix B Examples 35*
- appendix C PowerShell Quick Reference 73*
- appendix D Additional PowerShell topics 123*



A P P E N D I X A

Comparing PowerShell to other languages

A.1 PowerShell and cmd.exe	2	A.5 PowerShell and Perl	26
A.2 PowerShell and UNIX shells	14	A.6 PowerShell and C#	27
A.3 PowerShell and VBScript	19	A.7 Summary	34
A.4 WMI shootout—VBScript versus PowerShell	22		

Many people will come to PowerShell with experience using other languages or shells, so in this appendix we'll compare PowerShell to a number of common shells and languages people may know. We'll spend most of our time on `cmd.exe` (the traditional Windows command-line shell) and the UNIX shells. We'll also look at a variety of issues that Perl, VBScript, and C# programmers may encounter. Along the way, we'll introduce a number of handy techniques that will be of interest to the general PowerShell user.

NOTE These sections aren't strictly feature-by-feature comparisons. Rather, they're sets of hints and tips that I have gathered over the years based on questions people have asked. They represent the most common stumbling blocks and questions that new users seem to have. It's impossible to capture every problem in an appendix. The community, through blogs and forums, is a tremendous resource for assisting new users in becoming successful with PowerShell.

A.1 POWERSHELL AND CMD.EXE

The most commonly used shell on Windows today is `cmd.exe`. Let's look at some of the things a `cmd.exe` user might need to know in order to use PowerShell successfully.

A.1.1 Basic navigation and file operations

PowerShell provides a set of default aliases so the basic command names that a `cmd.exe` user knows are also available in PowerShell. You can do basic operations such as `dir`, `copy`, and `sort`, and they will do more or less what you expect. It becomes more complex when you start to specify options to these commands, because PowerShell uses different option syntax.

Commands are also factored differently in PowerShell. By *factored*, we mean how the functionality is distributed across the various commands. `Cmd.exe` has a small number of big commands with a lot of functionality in each command. The problem is that when you need to do something that isn't built-in, these commands are hard to compose together in a script. PowerShell has a much larger set of commands with fewer options that are designed to be composed. For example, the PowerShell equivalent of `dir` doesn't have a sort option; you use the `sort` command instead.

In the following tables, we'll present some of the most common command patterns that `cmd.exe` users encounter. Table A.1 shows the basic navigation commands in `cmd.exe` and their equivalent in PowerShell. We mentioned earlier that the commands in PowerShell are aliases. In the table, there are sometimes second examples. These second examples are the *unaliased* versions of the commands. For example, `dir` is an alias for `Get-ChildItem`.

Table A.1 Basic navigation operations in `cmd.exe` and PowerShell

Operation description	cmd.exe syntax	PowerShell syntax
Get a listing of the current directory.	<code>dir</code>	<code>dir</code> <code>Get-ChildItem</code>
Get a listing of all the files matching a particular pattern.	<code>dir *.txt</code>	<code>dir *.txt</code> <code>Get-ChildItem *.txt</code>
Get a listing of all the files in all the sub-directories of the current directory.	<code>dir /s</code>	<code>dir -rec</code> <code>Get-ChildItem -rec</code>
List all text files in all subdirectories.	<code>dir /s *.txt</code>	<code>dir -Recurse -Filter</code> <code>Get-ChildItem -Recurse</code> <code>-Filter *.txt</code>
Sort files in order by last write time.	<code>dir /o:-d</code>	<code>dir sort -Descending</code> <code>LastWriteTime</code>
Set the current working directory to a particular location.	<code>cd c:\windows</code>	<code>cd c:\windows</code> <code>Set-Location c:\windows</code>

Copying, moving, and deleting files are also common operations. Table A.2 covers a set of common scenarios comparing the `cmd.exe` commands against their PowerShell equivalents.

Table A.2 Basic file operations in `cmd.exe` and PowerShell

Operation	cmd.exe syntax	PowerShell syntax
Copy a file to the screen.	<code>type file.txt</code>	<code>type file.txt</code> <code>Get-Content file.txt</code>
Copy a file.	<code>copy f1.txt f2.txt</code>	<code>copy f1.txt f2.txt</code> <code>Copy-Item f1.txt f2.txt</code>
Copy several files.	<code>copy f1.txt, f2.txt, f3.txt c:\</code>	<code>copy f1.txt, f2.txt, f3.txt c:\</code> <code>Copy-Item f1.txt, f2.txt, f3.txt c:\</code>
Concatenate several files.	<code>copy f1.txt+f2.txt+f3.txt f4.txt</code>	<code>type f1.txt, f2.txt, f3.txt > f4.txt</code> <code>Get-Content f1.txt, f2.txt > f4.txt</code>
Delete a file.	<code>del file.txt</code>	<code>del file.txt</code> <code>Remove-Item file.txt</code>
Delete all text files in the current directory.	<code>del *.txt</code>	<code>del *.txt</code> <code>Remove-Item *.txt</code>
Delete all text files in all sub-directories of the current directory.	<code>del /s *.txt</code>	<code>del -rec *.txt</code> <code>Remove-Item -rec *.txt</code>

Another common way to do file operations is using the redirection operators. PowerShell supports the pipe operator (`|`). It also supports the same set of redirection operators (`>`, `>>`, `2>`, `2>&1`) that are in `cmd.exe`, but it doesn't support input redirection. Instead, you have to use the `Get-Content` command (or its alias `type`).

In the next section, we'll look at some of the syntactic features of each environment that are used for scripting.

A.1.2 Variables and substitution

In `cmd.exe`, environment variables are enclosed in percent (%) signs and are set using the `set` command, as shown in the following example:

```
C:\>set a=3

C:\>echo a is %a%
a is 3
```


In PowerShell, variables are indicated with a dollar sign (\$) in front of the variable. No special command is required to set a variable's value—simple assignment is enough. Here's the previous example using PowerShell syntax:

```
PS (1) > $a = 3
PS (2) > echo a is $a
a is 3
```

There is another thing to note about variables. PowerShell supports different kinds of variables; for the most part, `cmd.exe` variables are environment variables. This means these variables are automatically exported to child processes when `cmd.exe` creates a process. On the other hand, in PowerShell, environment variables are stored in a separate namespace `ENV:.` To set an environment variable in PowerShell, you do this:

```
$env:envVariable = "Hello"
```

The next thing to discuss is how to perform calculations. The `set` command in `cmd.exe` is used to do arithmetic calculations. Here's what a calculation looks like in `cmd.exe`:

```
C:\>set /a sum=33/9
3
C:\>echo sum is %sum%
sum is 3
```

Again, PowerShell requires no special syntax. To do a calculation, you write the expressions as shown in the next few examples:

```
PS (1) > $a = 2 + 4
PS (2) > $a
6
PS (3) > $b = $a/3 -[math]::sqrt(9)
PS (4) > $b
-1
PS (5) > [math]::sin( [math]::pi * 33 )
4.88487288813344E-16
PS (6) >
```

Because PowerShell is built on top of .NET, it has the full mathematical capabilities that languages such as C# and VB have, including floating-point and access to advanced math functions.

Finally, in `cmd.exe`, you can do a variety of string operations as a side effect of expanding a variable. In PowerShell, these types of operations are done with expressions and operators. For example, if you have a variable containing a file named `myscript.txt` and you want to change it to be `myscript.ps1`, you would do it with the `-replace` operator:

```
PS (1) > $file = "myscript.txt"
PS (2) > $file -replace '.txt$', '.ps1'
myscript.ps1
```

This displays the changed string. Let's update the variable itself:

```
PS (3) > $file = $file -replace '.txt$', '.ps1'
```

And now verify that it has been changed:

```
PS (4) > $file  
myscript.ps1
```

Using operators to update variable values isn't as concise as the variable expansion notation that `cmd.exe` uses, but it's consistent with the rest of PowerShell instead of being a special-case feature that only applies to variable expansion.

A.1.3 Running commands

Next, let's look at differences in how commands are run in the two environments. In PowerShell, you don't have to use a command to echo something to the screen. A string on the command line is directly output:

```
PS (3) > "a is $a"  
a is 3
```

On the other hand, you also need to be able to run commands whose names contain spaces. You do so with the PowerShell call operator `&`. To run a command with a space in the name, you do this:

```
& "command with space in name.exe"
```

Another difference between the environments is how scripts are run. Normally, with `cmd.exe`, when you run a script, any changes that the script makes affect your current shell session. This has led to a common practice where BAT files are used to set up the environment. The `vcvarsall.bat` file that's part of Microsoft Visual Studio is a good example of this. When you run the file, it updates the path and sets all the variables that are necessary to use the Visual Studio command-line tools. (In section A.1.7, we'll talk about how to use this type of batch file.)

This isn't the default behavior in PowerShell. By default, when a script is run, it runs in its own scope (see section 7.6) so that any nonglobal variables that are created are cleaned up when the script exits. To use a PowerShell script to modify the environment, you need to dot-source it. You put a dot and a space in front of the script to run. This is described in detail in section 11.1.1.

In `cmd.exe`, when you want to create a local scope for variables, you use the `setlocal/endlocal` keywords. PowerShell has the equivalent ability, again using the ampersand notation. Here's what it looks like:

```
PS (1) > $a = 3  
PS (2) > $a  
3  
PS (3) > & { $a = 22; $a }  
22  
PS (4) > $a  
3
```

In this example, in the outer scope, you set the value of `$a` to 3 and then display it. Then you use the `&` operator and braces to create a nested scope. In this nested scope, you set `$a` to 22 and then display the variable. Once you exit the nested scope, you again display the value of `$a`, which is the original value of 3.

A.1.4 Differences in syntax

The PowerShell syntax is obviously different from `cmd.exe`, but beyond basic syntax, there are some significant differences in the way commands are processed. One thing that a lot of people coming from `cmd.exe` find annoying is that in `cmd.exe`, you don't have to put spaces between built-in commands and their arguments. This means you can issue commands like the following:

```
C:\>cd\windows

C:\WINDOWS>cd..

C:\>dir\files\a.txt
Volume in drive C is C_Drive
Volume Serial Number is F070-3264

Directory of C:\files

04/25/2006  10:55 PM                98 a.txt
               1 File(s)                98 bytes
               0 Dir(s)  94,158,913,536 bytes free
```

With `cmd.exe`, entering commands like this works fine. But in PowerShell V1, they would both result in errors (in PowerShell V2 we added a built-in function for "cd.." so the second one won't generate an error in V2 systems.)

```
PS (1) > cd\windows
The term 'cd\windows' is not recognized as a cmdlet, function,
operable program, or script file. Verify the term and try again.
At line:1 char:10
+ cd\windows <<<<
PS (2) > cd..
The term 'cd..' is not recognized as a cmdlet, function, operable
program, or script file. Verify the term and try again.
At line:1 char:4
+ cd.. <<<<
```

Commands can be used this way in `cmd.exe` because `cmd.exe` treats its built-in commands as special cases and doesn't require spaces to separate the commands from the arguments. PowerShell doesn't have any special built-in commands—all commands are treated the same. This allows for greater consistency in PowerShell and, down the road, greater extensibility. But that doesn't help all the people who have `cd..` or `cd\` burned into their finger-memory. For people who find it to be a real problem, it's possible to define functions to work around the difficulty. Let's look at an example of this type of function:

```
PS (1) > function cd.. { cd .. }  
PS (2) > function cd\ { cd \ }
```

Now, try it out. First `cd` into the root of the file system, then into the windows directory, and finally back to the root:

```
PS (3) > cd\  
PS (4) > cd windows  
PS (5) > cd..
```

Defining functions like this works around specific instances of the problem, but it doesn't fix everything. As we saw, you still have to put a space between `cd` and `windows`. Even so, many people find this approach useful when they are starting out with PowerShell. If you want to make functions available every time you start PowerShell, you can put them in the personal profile, which is named by the variable `$profile`. Run this:

```
notepad $profile
```

In Notepad, add any new definitions you want to have available, and then save the file.

NOTE You don't need to do this in PowerShell version 2 because these functions are now predefined for you as part of the installation.

The next time you start PowerShell, the functions you've defined in the profile will be available.

A.1.5 Searching text: `findstr` and `Select-String`

A common command for searching through files from `cmd.exe` is `findstr.exe`. (Note that because it's an external command, it will also work fine from PowerShell.) PowerShell has a similar command, `Select-String`. Why have a new cmdlet when the old executable already works? There are a couple of reasons. First, the `Select-String` cmdlet returns objects that include the matching text, the number of the line that matched, and the name of the file as separate fields, making it easier to process the output. Second, it uses the .NET regular expression library, which is much more powerful than the patterns `findstr` can handle.

If you look at the help for `findstr`, you see that it has a lot of operations that aren't built into `Select-String`. This is because PowerShell uses a composition model. Instead of building a large but fixed set of operations into one command, there are more small composable commands. For example, to search all the C# files in all the subdirectories with `findstr`, the command is

```
findstr /s Main *.cs
```

With `Select-String`, you pipe the output of `dir` into the command:

```
dir -rec -filter *.cs | Select-String main
```

A.1.6 For loop equivalents

Iteration (operating over collections of things) is done in `cmd.exe` with the `for` statement. This is a powerful flow-control statement, but it's also complex. Again, PowerShell has several simpler mechanisms for doing the same thing using pipelines. Table

A.3 shows a number of simple examples comparing a `cmd.exe` `for` statement with the equivalent PowerShell construct.

Table A.3 Examples of iteration in `cmd.exe` and PowerShell

Operation	Cmd.exe syntax	Powershell syntax
Iterate over files.	<code>for %f in (*) do echo %f</code>	<code>dir where { -not \$_.PSIsContainer} foreach {\$_ }</code>
Iterate over directories.	<code>for /d %f in (*) do echo %f</code>	<code>dir foreach { \$_ .PSIsContainer} foreach {\$_ }</code>
Iterate over the numbers from 1 to 9 by twos.	<code>for /l %i in (1,2,10) do (@echo %i)</code>	<code>for (\$i=1; \$i -lt 10; \$i+=2) { \$i }</code>

Now let's look at a somewhat more complex example. As well as iterating over files, the `cmd.exe` `for` statement can be used to parse files. The following shows a `for` command that will extract and print the first three tokens from a data file:

```
for /f "tokens=1-3" %a in (c:\temp\data.txt) do (
@echo a is %a b is %b c is %c)
```

The corresponding command in PowerShell is

```
Get-Content c:\temp\data.txt | foreach {
    $a,$b,$c,$null = -split $_; "a is $a b is $b c is $c" }
```

The `for` statement is monolithic—there's no way to use the tokenizing capability of the `for` statement other than in the `for` statement. In PowerShell, all the operations (reading the file, tokenizing, and so on) are done with separate components. For example, you can use the `-split` operator anywhere because it's not part of the `foreach` command.

A.1.7 Batch files and subroutines

In `cmd.exe`, subroutines are invoked with the `goto` statement and also use a `goto` to return to the calling location. A `cmd.exe` procedure is invoked using the `call` statement. PowerShell, on the other hand, has first-class functions including named parameters, optionally typed parameters, and recursion. PowerShell scripts are also callable as commands, and again recursion and named parameters are permitted. PowerShell doesn't have a `goto` statement, but you can use labels with the PowerShell loop statements.

Also note that there are no differences in behavior between code typed on the command line and code executed out of a function or script in PowerShell. The syntax and semantics are the same everywhere.

Running the `vcvarsall.bat` batch file

One of the most common uses for `cmd.exe` batch files is to set up environment variables. As mentioned previously, if Visual Studio is installed, a batch file called `vcvarsall.bat` is installed along with the product; it's used to set up the environment variables in `cmd.exe` to do development work. It's also possible to use these batch files from PowerShell by executing them, dumping the changes that have been made to the environment, and then importing those changes back into the PowerShell environment. This sounds complicated but turns out to be simple. First you define the batch command you want to run in a variable called `$cmd`:

```
PS (1) > $cmd =  
>> '"C:\Program Files\Microsoft Visual Studio 8\VC\vcvarsall.bat"' +  
>> ' & set'  
>>
```

Next, you invoke the command, piping the output into the `foreach` command:

```
PS (2) > cmd /c $cmd | foreach {  
>> $n,$v = $_ -split '='; set-item -path ENV:$n -value $v }  
>>
```

In the body of the scriptblock, the incoming command is split into name (`$n`) and value (`$v`) pieces. These pieces are then passed to `Set-Item` to set the values of corresponding environment variables. Let's check the result of what you've done:

```
PS (3) > dir ENV:v*
```

Name	Value
----	-----
VS80COMNTOOLS	C:\Program Files\Microsoft Visual...
VSINSTALLDIR	C:\Program Files\Microsoft Visual...
VCINSTALLDIR	C:\Program Files\Microsoft Visual...

You can see that the variables have been set properly. Let's generalize this into a function that can work with any batch file. This `Get-BatchFile` function looks like this:

```
function Get-BatchFile ($file)  
{  
    $cmd = "`"$file`" & set"  
    cmd /c $cmd | Foreach-Object {  
        $n,$v = $_.split('=')  
        Set-Item -Path ENV:$n -Value $v  
    }  
}
```

This function does the same thing as the commands you typed in. The only difference is that the batch file to run is passed in as an argument.

A.1.8 Setting the prompt

One of the most common questions people moving to PowerShell ask is, “How can I customize my prompt?” In `cmd.exe`, this is done by setting the `%PROMPT%` variable. The typical setting for `%PROMPT%` is

```
C:\files>set prompt
PROMPT=$P$G
```

In PowerShell, the prompt is controlled by the `prompt` function. This is a function that should return a single string. The equivalent of `PG` is

```
PS (31) > function prompt {"$PWD> "}
C:\files>
```

The nice thing about `prompt` being a function in PowerShell is that it can do anything. For example, if you want to display the day of the week as your prompt, you can do this:

```
C:\files> function prompt { "$(Get-Date).DayOfWeek)> " }
Monday>
```

You redefine the function, and now you see what day it is. Here’s something else you can do: the problem with displaying the path in the prompt is that it can become long. As a consequence, many people prefer to show it in the window title. You can do this using a function like the following:

```
function prompt {
    $host.ui.rawui.WindowTitle = "PS $pwd"
    "PS > "
}
```

The result of this prompt definition is shown in figure A.1. The string `"PS > "` is still displayed as the prompt, but the function also sets the window title.

Because the prompt is a function, it can do pretty much anything—log commands, play sounds, print quotes, and so on.

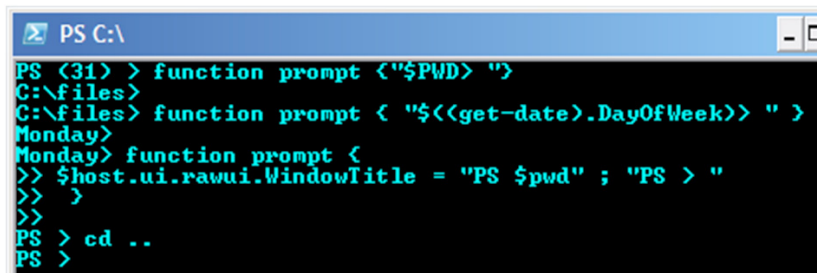


Figure A.1 Two examples of how to customize the prompt in PowerShell by redefining the `prompt` function

A.1.9 Using doskey in PowerShell

The `doskey` tool lets you define keyboard macros in a console window. What do we mean by this? `Doskey` macros are processed by the console subsystem—the part of the Windows operating system that handles rendering the console window and reading from the keyboard. When a console program does a `Readline()` call, the console subsystem checks to see whether any macros are available for that program. If there are, it does the macro substitution on the string before they're returned to the user.

NOTE Because `doskey` relies on the console infrastructure, it only applies to console programs. `powershell_ise.exe` isn't a console program, so the things we're discussing here don't apply to it.

Why do we care? Because it means we can also use `doskey` macros in PowerShell. Here's an example that shows how to use the `doskey` utility from PowerShell. First look to see whether any macros are defined for PowerShell initially:

```
PS (2) > doskey /macros:powershell.exe
```

Nothing is returned, so, obviously, there are currently no `doskey` macros for PowerShell. Notice that you have to specify the full name of the executable file. The default is `cmd.exe`, so to make the `doskey` commands apply to PowerShell, you always have to specify the name `powershell.exe`. Let's define a macro:

```
PS (3) > doskey /exename=powershell.exe `
>> ddir = dir `*$* `| ? `{ '$_.PSIsContainer' `}
>>
```

This requires a fair bit of quoting to make sure the arguments get passed through to `doskey` properly. If you want to define a number of macros, it's probably easiest to define them using the `doskey /file` option.

Now let's make sure the macro was defined properly. Remember, the text will be substituted on the command line, so the resulting command line has to be syntactically correct:

```
PS (4) > doskey /macros:powershell.exe
ddir=dir $* | ? { $_.PSIsContainer }
```

It looks fine. Notice the use of `$*` in the macros. When `doskey` macro substitution is done, `$*` will be replaced by any arguments to the macro. Let's try it:

```
PS (5) > ddir
```

Directory: Microsoft.PowerShell.Core\FileSystem::C:\files

Mode		LastWriteTime	Length	Name
----		-----	-----	----
d----	8/19/2006	2:35 PM		d1
d----	8/19/2006	2:36 PM		d2
d----	8/19/2006	2:35 PM		d3

It displays only the directories in the current directory. Let's give it the option `-rec` and see what happens:

```
PS (6) > ddir -rec

Directory: Microsoft.PowerShell.Core\FileSystem::C:\files

Mode                LastWriteTime         Length Name
----                -
d-----            8/19/2006   2:35 PM             d1
d-----            8/19/2006   2:36 PM             d2
d-----            8/19/2006   2:35 PM             d3

Directory: Microsoft.PowerShell.Core\FileSystem::C:\files\d2

Mode                LastWriteTime         Length Name
----                -
d-----            8/19/2006   2:36 PM             dd1
d-----            8/19/2006   2:36 PM             dd2
```

This time, you get all the directories including subdirectories. `doskey` also lets you look at the console command history. Let's try it; again, you have to specify the full executable name:

```
PS (7) > doskey /exename=powershell.exe /h
cd c:\files
doskey /macros:powershell.exe
doskey /exename=powershell.exe `
  ddir = dir `*$ `| ? `{ '$_PSIsContainer' `}
doskey /macros:powershell.exe
ddir
ddir -rec
doskey /exename=powershell.exe /h
```

This shows you all the commands you've typed. But PowerShell also maintains a history of all the commands it's executed. Because these commands are recorded after the `doskey` substitutions, the PowerShell history should have the expanded commands instead of what you typed:

```
PS (8) > Get-History

Id CommandLine
--
1 cd c:\files
2 doskey /macros:powershell.exe
3 doskey /exename=powershell.exe `...
4 doskey /macros:powershell.exe
5 dir | ? { $_.PSIsContainer }
6 dir -rec | ? { $_.PSIsContainer }
7 doskey /exename=powershell.exe /h
```

Notice the commands with IDs 5 and 6. These are the expanded commands that correspond to the typed commands `ddir` and `ddir -rec`. This is a way you can see what the macro expansion did.

The `doskey` utility is another tool you can use to help ease your transition from `cmd.exe` to PowerShell. It lets you define parameterized macros that can expand simple strings into more complex PowerShell expressions.

A.1.10 Using `cmd.exe` from PowerShell.

The final topic is how you can use `cmd.exe` from PowerShell. In particular, how can you use the existing scripts? The answer is that, for the most part, you can use them. If PowerShell sees a file with a `.cmd` file extension, it will run it. The part that doesn't work comes in with all the configuration scripts that people use. These are scripts that set a number of variables and then exit. They won't work when run from PowerShell because the `cmd.exe` process that's created to run them will exit when the batch file has completed, discarding any changes.

You can also run any of the `cmd` built-ins from PowerShell using `cmd /c`. Here's an example of using `cmd.exe` for `command` from PowerShell:

```
PS (1) > cmd /c 'for %f in (*) do @echo %f'
a.txt
b.txt
c.txt
d.txt
```

Now let's use the `cmd.exe` `for` command to generate a set of files that you then process using the PowerShell `foreach` statement:

```
PS (2) > foreach ($f in cmd /c 'for %f in (*) do @echo %f')
>> { $f.ToUpper() }
>>
A.TXT
B.TXT
C.TXT
D.TXT
```

From this, you can see that as you're learning to use PowerShell, you don't have to abandon all the hard-won knowledge you've accumulated with `cmd.exe` scripting over the years. You can mix and match as you see fit.

A.1.11 Calling PowerShell from `cmd.exe`

You need to know one last important thing when you're coming to PowerShell from a `cmd.exe` background: how to use PowerShell from `cmd.exe`. This allows you to start using PowerShell features incrementally in your batch files. The PowerShell executable makes this easy because any command-line arguments it doesn't understand are passed through to the interpreter. For example, you can use `powershell.exe` to perform mathematical operations

```
c:\>powershell 2+2
4
```

or access WMI:

```
c:\>powershell Get-WmiObject Win32_Bios
SMBIOSBIOSVersion : 7SET33WW (1.19 )
Manufacturer      : LENOVO
Name              : Ver 1.00PARTTBL
SerialNumber      : LVB7KY3
Version           : LENOVO - 1190
```

You can even mix and match commands on the same line. In the following, you use the [find](#) utility to extract the lines containing “Ver” from the output of [Get-WmiObject](#):

```
c:\>powershell Get-WmiObject Win32_Bios | find "Ver"
SMBIOSBIOSVersion : 7SET33WW (1.19 )
Name              : Ver 1.00PARTTBL
Version           : LENOVO - 1190
```

It’s also possible to do PowerShell piping in the same command that uses [cmd.exe](#) piping. You have to put the pipe symbol in double quotes:

```
c:\>powershell Get-Process "|" foreach {$_ .Name} | find "cs"
csrss
csrss
```

In this example, you call [Get-Process](#) to get a list of running processes and then pipe this output into the [foreach](#) command, which extracts name from the objects. The overall output from [powershell.exe](#) is then passed into [find](#) through [cmd.exe](#)’s implementation of pipes.

Passing unknown arguments into the interpreter usually works fine, but there is a case where this causes problems: when you mistype a switch to [powershell.exe](#). This gets passed to the interpreter, which then emits the following rather incomprehensible error:

```
c:\>powershell -badoption
Missing expression after unary operator '-'.
At line:1 char:2
+ ~~~~~ badoption
+ ~~~~~ CategoryInfo          : ParserError: (String) [], ParentCont
+ ~~~~~ ainsErrorRecordException
+ ~~~~~ FullyQualifiedErrorId : MissingExpressionAfterOperator
```

What’s happening here is that the interpreter is trying to parse the string it received. It understands that `-` is a valid operator, but what follows that operator isn’t a valid PowerShell expression—and so you get a syntax error.

A.2 **POWERSHELL AND UNIX SHELLS**

In this section, we’ll look at examples that compare PowerShell to the UNIX shells—in particular the Bourne shell family ([sh](#), [ksh](#), [bash](#), and so on).

NOTE This section in the first edition of the book caused a lot of irate UNIX shell users to warm up their flamethrowers. (Given that my background prior to Microsoft was mostly in UNIX and UNIX-like systems,

I probably would have been one of them.) The point here isn't to say that PowerShell is better than the UNIX shells. Rather, it's that they take different approaches to solving similar problems on different systems. As a result, this becomes something of an apples and oranges discussion.

Although inspired by the UNIX shells, PowerShell is different from them. The most obvious difference is that PowerShell uses objects as the basic model of interaction instead of strings. Second, the list of built-in commands is both larger and end-user extensible, by which I mean that users can create extensions that are loaded into the PowerShell process.

Because the same extension mechanism is used by all commands, there is no difference between the built-in commands and user-created extension cmdlets. This model is necessitated by and a consequence of the decision to use objects. The out-of-process extension model used by traditional shells is impractical for an object-based shell. Even using XML as an intermediate representation is impractical due to the cost of serializing and deserializing each object.

NOTE There are UNIX shells that allow end-user extensions. For example, the Desktop Korn Shell (dtksh), part of the Common Desktop Environment (CDE), allows for in-process end-user extensions. This lets it call into various GUI libraries including the Tk part of Tcl/Tk.

Instead of doing a feature-by-feature comparison of PowerShell and the UNIX shells, the approach in this section is to work through a set of illustrative examples showing how a problem can be solved in each environment. Because the examples are designed to illustrate features in PowerShell, the corresponding UNIX examples are non-optimal.

A.2.1 Stopping all processes

To stop all processes that begin with the letter *p* on a UNIX system, you type the following shell command line:

```
$ ps -e | grep " p" | awk '{ print $1 }' | xargs kill
```

The `ps` command retrieves a list of processes and sends the output text to `grep`. The `grep` command searches the string for processes whose names begin with *p*. The output of `grep` is, in turn, sent to the `awk` command, which selects the first column in the input text (which is the process ID) and then passes those to the `xargs` command. The `xargs` command executes the `kill` command for each process it receives as input. Beyond the complexity of the number of stages that need to be executed, this command is also fragile. The problem is that the `ps` command behaves differently on different systems (and sometimes on different versions of the same system). For example, the `-e` flag on `ps` may not be present; or if the processed command isn't in column 1 of the output, this command-line procedure will fail.

Now let's look at the equivalent command in PowerShell. It's both simpler and more understandable:

```
PS (1) > Get-Process p* | Stop-Process
```

This command line says, “Get the processes whose names start with *p* and stop them.” The `Get-Process` cmdlet takes an argument that matches the process name; the objects returned by `Get-Process` are passed directly to the `Stop-Process` cmdlet, which acts on those objects by stopping them. Next, let's look at a more sophisticated example.

A.2.2 Stopping a filtered list of processes

This section tackles a more complex task: “Find processes that use more than 10 MB of memory, and kill them.” The UNIX commands to do this are as follows:

```
$ ps -el | awk '{ if ( $6 > (1024*10)) { print $3 } }' |  
grep -v PID | xargs kill
```

The success of this command line relies on the user knowing that the `ps -el` command will return the size of the process in kilobytes (KB) in column 6 and that the PID of the process is in column 3. It also requires that the first row in the output of `ps` be removed.

Now let's look at the corresponding PowerShell commands. Again, the command is shorter and simpler:

```
PS (2) > Get-Process | where { $_.WS -gt 10MB } | Stop-Process
```

Here you can see that the commands act against objects rather than against text. There is no issue with determining the column that contains the size of the process, or which column contains the `ProcessID`. The memory size may be referred to logically, by its name. The `where` command can inspect the incoming object directly and refer to its properties. The comparison of the value for that property is direct and understandable.

A.2.3 Calculating the size of a directory

In this example, you want to calculate the number of bytes in the files in a directory. You iterate over the files, getting the length and adding it to a variable, and then print the variable. First, here's the UNIX shell code:

```
$ tot=0; for file in $( ls )  
> do  
>     set -- $( ls -log $file )  
>     echo $3  
>     (( tot = $tot + $3 ))  
> done; echo $tot
```

This example uses the `set` shell command that creates numbered variables for each whitespace-separated element in the line rather than the `awk` command as in earlier

examples. If the `awk` command were used, it would be possible to reduce the steps to the following:

```
$ ls -l | awk '{ tot += $5; print tot; }' | tail -1
```

This reduces the complexity of what you type but requires that you know both the shell language and also how to script in `awk`, which has its own complete language.

The PowerShell loop is similar; each file in the directory is needed, but it's far simpler, because the information about the file is already retrieved as part of the file information object:

```
PS (3) > Get-ChildItem | Measure-Object -Property length
```

The `Measure-Object` cmdlet interacts with objects, and if it's provided with a property from the object, it will sum the values of that property. Because the property `length` represents the length of the file, the `Measure-Object` cmdlet is able to act directly on the object by referring to the property name rather than knowing that the length of the file is in column 3 or column 5.

A.2.4 Working with dynamic values

Many objects provided by the system aren't static but dynamic. This means that after an object is acquired, it's not necessary to reacquire the object at a later time because the data in the object is continually updated as the conditions of the system change. Conversely, any changes you make to these objects are reflected immediately in the system. We call these *live objects*.

As an example, suppose you wanted to collect the amount of processor time that a process used over time. In the traditional UNIX model, the `ps` command would need to be run repeatedly, the appropriate column in the output would need to be found, and then the subtraction would need to be done. With a shell that's able to access live process objects, you only have to get the process object once and, because this object is continually updated by the system, you can keep rereading the same property. The following examples illustrate the differences, where the memory size of an application is checked in 10-second intervals and the differences are output. First here's the UNIX shell script to do this:

```
$ while [ true ]
do
    msize1=$(ps -el|grep application|grep -v grep|awk '{ print $6}')
    sleep 10
    msize2=$(ps -el|grep application|grep -v grep|awk '{print $6}')
    expr $msize2 - $msize1
    msize1=$msize2
done
```

Now, here's the same example in PowerShell:

```
PS> $app = Get-Process application
PS> while ( $true ) {
>> $msize1 = $app.VS
```

```
>> start-sleep 10
>> $app.VS - $msize1
>> }
```

Again, the PowerShell script is quite a bit simpler and more easily understood.

A.2.5 Monitoring the life of a process

It's even more difficult to determine whether a specific process is no longer running. In this case, the UNIX user must collect the list of processes and compare them to another list:

```
$ processToWatch=$( ps -e | grep application | awk '{ print $1 }'
$ while [ true ]
> do
>     sleep 10
>     processToCheck=$(ps -e |grep application |awk '{print $1}' )
>     if [ -z "$processToCheck" -or \
>         "$processToWatch" != "$processToCheck" ]
>     then
>         echo "Process application is not running"
>         return
>     fi
> done
```

In PowerShell it looks like this:

```
PS (1) > $processToWatch = Get-Process application
PS (2) > $processToWatch.WaitForExit()
```

As you can see in this example, the PowerShell user need only collect the object and then wait to be notified that the object has exited.

A.2.6 Checking for prerelease binaries

Suppose you want to determine which processes were compiled as prerelease code. This information isn't kept in the standard UNIX executable, so you would need a set of specialized utilities to add this information to the binary and then another set of utilities to collect this information. These utilities don't exist; it isn't possible to accomplish this task. But this information is part of the standard Windows executable file format. Here's how you can use PowerShell to find out which of the running processes on the system are marked as prerelease binaries:

```
PS (1) > Get-Process | where {
>> $_.mainmodule.FileVersionInfo.IsPreRelease}
>>
```

Handles	NPM(K)	PM(K)	WS(K)	VS(M)	CPU(s)	Id	ProcessName
643	88	1024	1544	15	14.06	1700	AdtAgent
453	15	25280	7268	199	91.70	3952	devenv

In this example, you're using a cascade of properties. The appropriate property from the process object (`MainModule`) is inspected, the property `FileVersionInfo` is referenced (a property of `MainModule`), and the value of the property `IsPreRelease`

is used to filter the results. If `IsPreRelease` is true, the objects that are output by the `Get-Process` cmdlet are output.

A.2.7 Upper casing a string

The availability of methods on objects creates an explosion of possibilities. For example, if you want to change the case of a string from lowercase to uppercase, in a UNIX shell you do either this

```
$ echo "this is a string" | tr [:lower:] [:upper:]
```

or this:

```
$ echo "this is a string" | tr 'a-z' 'A-Z'
```

Now let's see what this looks like in PowerShell:

```
PS (1) > "this is a string".ToUpper()
```

You can use the `ToUpper()` method on the string object instead of having to use external commands such as `tr` to do the mapping.

A.2.8 Inserting text into a string

Let's look at another example using methods. Suppose you want the string "ABC" to be inserted after the first character in the word *string*, so you have the result *sABCtring*. Here's how to do it with the UNIX shell, which requires using the `sed` command:

```
$ echo "string" | sed "s|\\(\\.\\)\\(\\.\\*)|\\1ABC\\2|"
```

You can use the same approach—regular expressions—in PowerShell, which looks like this:

```
PS (1) > "string" -replace '(\\.)(\\.\\*)','$1ABC$2'
sABCtring
```

Or you can use the `insert` method on the string object to accomplish the same thing, but much more directly:

```
PS (2) > "string".Insert(1,"ABC")
sABCtring
```

Although both examples require specific knowledge, using the `Insert()` method is more intuitive than using regular expressions.

A.3 POWERSHELL AND VBSCRIPT

If `cmd.exe` was the traditional shell on Windows, VBScript was the standard scripting tool on Windows. Let's look at some things a VBScript user should know when working with PowerShell.

PowerShell shares little syntax with VBScript, which is mostly due to the verbosity of that syntax. Because the primary mode of use of PowerShell is as an interactive shell, the PowerShell development team chose the more concise C-style syntax. (The

fact that the two languages are so different may help the VBScript user, because they're less likely to get mixed up between PowerShell and VBScript.)

Because management scripting in VBScript is mostly about working with COM and WMI objects, the most important thing for a VBScript user to know about are the equivalents to `CreateObject()` for creating COM objects and `GetObject()` for getting instances of WMI objects. You create a COM object in PowerShell with the `New-Object` cmdlet:

```
$ie = New-Object -com InternetExplorer.Application
```

And you get a WMI object with the `Get-WmiObject` cmdlet:

```
$tz = Get-WmiObject Win32_timezone
```

Chapters 18 (COM) and 19 (WMI) cover these subjects in detail, so let's turn our attention to a more complex problem: the syntactic differences between the two languages.

A.3.1 Syntactic differences

In this section, we'll list the syntactic differences that are most likely to cause problems for VBScript:

- Variables always begin with `$`, as in `$a`.
- Method invocations must always include the parentheses in the method name, because it's possible for an object to have a property named `SomeName` and a method `SomeName()`.
- Attempting to read nonexistent object properties doesn't cause an error.
- There is no `Set` keyword for setting object properties.
- PowerShell strings can be delimited with either single or double quotes. Inside double quotes, escape sequences and variable references are expanded. See chapter 3 for details on how this works and how to use it.
- PowerShell uses different comparison operators: `-lt` instead of `<` for less-than, `-gt` instead of `>` for greater-than, and so on. PowerShell comparisons are also case-insensitive by default.
- Arrays are indexed using square brackets instead of parentheses. Assigning to an element in an array looks like this: `$a[2] = "Hello"`.
- The plus (+) operator is used for concatenating strings and arrays. The type of the left-hand argument controls the type of the conversion.
- The PowerShell syntax is C-like in that statement blocks are delimited with braces { and } instead of keywords. For example, in PowerShell you write

```
if ($false -neq $true) { "false is not true " }
```

instead of

```
If (False <> True) Then
    MsgBox "false is not true"
End If
```

- Multiple statements on one line are separated with a semicolon (;) instead of a colon as in VBScript.

As in VBScript, if a statement is syntactically complete at the end of the line, no line termination is needed. But if a PowerShell statement isn't complete, it may be spread across several lines without explicit continuation. If a *continuation* character is needed, continuation is specified by a single backtick (`) at the end of the line. Note that a backtick isn't the same character as single quote (').

A.3.2 Strict mode and option explicit

PowerShell doesn't have the exact equivalent of `option explicit`, to turn on extra checks, but it does have a feature that requires that variables be initialized before they're used. You can turn this on either by using the `Set-PSDebug` (v1 and v2) command

```
Set-PSDebug -Strict
```

or by using the `Set-StrictMode` (v2 only) command:

```
Set-StrictMode -Version 2
```

The full set of checks that the strict modes provide are covered in sections 14.3.1 and 14.3.2.

Any expression that returns a value in a function will become part of the return value of the function. There is no need to assign to the function name. PowerShell also has a `return` statement that's only needed if you want to change the flow of control in the function and return early. For example, in VBScript, you might write this:

```
Function GetHello
    GetHello = "Hello"
End Function
```

The PowerShell equivalent is

```
function Get-Hello { "Hello" }
```

The closest equivalent to the VB `on error` construct is the PowerShell `trap` statement, which is covered in section 14.2.1.

Even with all these differences, sometimes it's surprisingly easy to translate a VBScript into a PowerShell script. This is because, in many cases, you're working with the same set of WMI or COM objects. Some other things are done differently—string manipulation being a prime example. This is where additional work is useful because naive translations, although simple, rarely take advantage of the features that PowerShell has for creating more concise scripts. In the next section, we'll look at how this works.

A.4 WMI SHOOTOUT—VBSCRIPT VERSUS POWERSHELL

We said earlier that the traditional scripting tool for WMI is VBScript. PowerShell is the new kid on the block. Let's examine why PowerShell is better than VBScript.

DISCLAIMER The example we'll look at is a bit of a straw man. The deficiencies we'll address don't have much to do with VBScript. The point is to highlight a key difference between a programming language and a shell environment. Shell environments provide automatic facilities for things such as default presentations of data so you don't have to write the same tedious formatting code over and over. (In fact, this kind of thing is so tedious that the Scriptomatic tool was created to automatically generate formatting code for ActiveScript languages such as VBScript and JScript.)

Wait a minute. Didn't section 18.5 cover hosting VBScript in PowerShell because PowerShell can't do everything? Correct. When you're working with COM, there are some things that VBScript can do that PowerShell can't (yet). PowerShell has an edge in that it has simpler access to system resources than VBScript, but where it wins is in presenting the output of an object. Remember, separating presentation from logic was one of the driving forces that led to PowerShell's creation. A significant amount of code in many VBScripts exists to format output. In PowerShell, most of the time this is free—the default output rendering mechanism works.

A.4.1 A VBScript example

Let's start with a simple VBScript that uses WMI—the kind of thing that the Scriptomatic tool generates. This comes from Microsoft's ScriptCenter, a repository for all things scripting. ScriptCenter is available at <http://mng.bz/SbTK>, and the repository of scripts is available at <http://mng.bz/r58O>.

The script we're going to look at uses WMI to get a list of the codecs installed on your system.

NOTE The term *codec* stands for, variously, coder-decoder, compressor/decompressor, or compression/decompression algorithm. A codec is a piece of software that allows you to encode or decode a data stream. The most common use these days is for media formats such as WMA, MP3, and so on. By checking the list of codecs, you can tell whether the system will be able to decode and play a particular file.

The VBScript code to do this is shown in the following listing. (This has been simplified somewhat from the original example in the TechNet script repository.)

Listing A.1 VBScript to list codecs

```
strComputer = "."
Set objWMIService = GetObject("winmgmts:" _
    & "{impersonationLevel=impersonate}!\\" & strComputer _
    & "\root\cimv2")
Set colItems = objWMIService.ExecQuery(
    "Select * from Win32_CodecFile") #1

For Each objItem in colItems
    Wscript.Echo "Manufacturer: " & objItem.Manufacturer
    Wscript.Echo "Name: " & objItem.Name
    Wscript.Echo "Path: " & objItem.Path
    Wscript.Echo "Version: " & objItem.Version
    Wscript.Echo "Caption: " & objItem.Caption
    Wscript.Echo "Drive: " & objItem.Drive
    Wscript.Echo "Extension: " & objItem.Extension
    Wscript.Echo "File Type: " & objItem.FileType
    Wscript.Echo "Group: " & objItem.Group
    strCreationDate = WMIDateStringToDate(objItem.CreationDate)
    Wscript.Echo "Creation Date: " & strCreationDate
    strInstallDate = WMIDateStringToDate(objItem.InstallDate)
    Wscript.Echo "Install Accessed: " & strInstallDate
    strLastModified = WMIDateStringToDate(objItem.LastModified)
    Wscript.Echo "Last Modified: " & strLastModified
    Wscript.Echo ""
Next

Function WMIDateStringToDate(dtmDate)
    WMIDateStringToDate = CDate(Mid(dtmDate, 5, 2) & "/" & _
        Mid(dtmDate, 7, 2) & "/" & Left(dtmDate, 4) _
        & " " & Mid(dtmDate, 9, 2) & ":" & _
        Mid(dtmDate, 11, 2) & ":" & Mid(dtmDate, _
            13, 2))
End Function
```

1 WMI preamble

2 Format data

3 Date helper function

This script begins with the standard preamble ❶ that you see in most VBScripts that use WMI. It sets up a query against the local WMI provider for this machine.

Next you display the set of fields ❷ you're interested in. This is straightforward but tedious. The code formats and prints each field. One thing to note is how the date fields are handled. WMI uses a string encoding of a date object. To convert this into a date object, you need to use a function. This function ❸ takes the string apart and puts it into a format that the system can convert into a date object. Now let's look at the PowerShell version.

A.4.2 The PowerShell version

You do this in two steps. You may have noticed that the VBScript function to parse the date is a bit complex. Rather than converting it into PowerShell, you'll reuse it for now through the `ScriptControl` object you saw earlier.

NOTE This COM object isn't available on a default Win7/Windows Server 2008R2 installation. You'll need to add it before trying these examples.

The first version, which is still using the VBScript date-converter function, is shown in the next listing (there are much easier ways of doing this, as you'll see later).

Listing A.2 PowerShell script to list codecs

```
$code = @'
Function WMIDateStringToDate(dtmDate)
    WMIDateStringToDate = CDate(Mid(dtmDate, 5, 2) & "/" & _
        Mid(dtmDate, 7, 2) & "/" & Left(dtmDate, 4) _
        & " " & Mid (dtmDate, 9, 2) & ":" & _
        Mid(dtmDate, 11, 2) & ":" & Mid(dtmDate, _
            13, 2))
End Function
'@

$vbss = New-Object -COM ScriptControl
$vbss.language = 'vbscript'
$vbss.AllowUI = $false
$vbss.addcode($code)
$vc = $vbss.CodeObject

Get-WmiObject Win32_CodecFile |
    %{ $_ | format-list Manufacturer, Name, Path, Version,
        Caption, Drive, Extension, FileType, Group,
        @{l="Creation Date"
            e={$vc.WMIDateStringToDate($_.CreationDate)}} ,
        @{l="Install Date"
            e={$vc.WMIDateStringToDate($_.InstallDate)}} ,
        @{l="Last Modified Date"
            e={$vc.WMIDateStringToDate($_.LastModified)}} }
```

1 VBScript code

2 Compile VBScript code

3 Process data

You use a here-string to hold the VBScript code **1** for the date converter function. Then you use the `ScriptControl` object to compile it into a `CodeObject` **2**. You use `$vc` to hold the `CodeObject` to make things a bit more convenient. This lets you invoke the method using `$vc.WMIDateStringToDate()`.

Next is the PowerShell code to retrieve and print out the data **3**. As you might expect, it's shorter than the VBScript code. You use `Get-WmiObject` to directly get the data and `Format-List` to format the output. You have to specify the set of fields to display; otherwise you get everything (this shows PowerShell not at its best, because you have to work harder to do less). Also of note is how the date fields are specified. In section 11.3, we show an example of using this construction with `Select-Object`. You can use the same pattern with the formatting commands. In the example, you're using label(l) and expression (e) to control what gets displayed. The label specifies the caption to use for the field, and the expression is a scriptblock used to calculate the value to display.

If you're going to work with WMI objects a lot and expect to run into dates on a regular basis, it behooves you to add a PowerShell native date converter to your toolbox. The second version of the script does this.

Listing A.3 The `WMIDateStringToDate` function

```
function WMIDateStringToDate($dtmDate)
{
    [datetime] ($dtmDate -replace
        '^(....)(..)(..)(..)(..)(..)(.*)$', '$1-$2-$3 $4:$5:$6')
}
Get-WmiObject Win32_CodecFile |
    foreach { $_ | Format-List Manufacturer, Name, Path, Version, Caption,
        Drive, Extension, FileType, Group,
        @{n="Creation Date"
            e={WMIDateStringToDate $_.CreationDate}},
        @{n="Install Date"
            e={WMIDateStringToDate $_.InstallDate}},
        @{n="Last Modified Date"
            e={WMIDateStringToDate $_.LastModified}} }
```

`WMIDateStringToDate` is the function that converts the WMI dates. You use regular expressions to do it, because they're so convenient in PowerShell. This date format looks like

```
20051207090550.505000-480
```

The first four digits are the year; the next two are the month, followed by the day, hours, minutes, and seconds. You use the `sub-match` feature with `-replace` to rearrange the date into something that .NET can convert into a `[DateTime]` object. The rest of the script is unchanged.

NOTE The `WMIDateStringToDate` function is only needed in PowerShell V1. In PowerShell V2, two script methods are available to take care of this issue: `ConvertFromDateTime` and `ConvertToDateTime`.

Let's look at the progress you've made. The VBScript version of the function is 29 lines long. The first PowerShell version that used the VBScript date function is 25 lines—not much better, because of the overhead of using the script control. The final version is only 14 lines—half the size.

When exploring WMI, a baseline VBScript is roughly six lines plus one line per property exposed by the object, merely to show values of all instances. And it won't work for individual values that are arrays (for example, a network adapter has an array `IPAddress` property, even if there is only one address in the array). For PowerShell, getting a complete, viewable result from a WMI class collection is always one line of code.

An interesting observation to take away from this exercise is that script is as long as it is because you don't want to show everything about the object. In VBScript (and most other non-shell languages), the more you want to show, the more work you

Date help

If you consult the documentation, you'll see that there is a COM class that deals with these dates, saving you a bunch of entertaining but unnecessary hacking about:

```
PS (1) > $d = New-Object -com WbemScripting.SWbemDateTime
PS (2) > $d.value = "20051207090550.505000-480"
PS (3) > $d.GetVarDate()
```

Wednesday, December 07, 2005 9:05:50 AM

And there is also a fully functional .NET class that lets you do things like this:

```
[Management.ManagementDateTimeConverter]::ToDateTime(
    "20051207090550.505000-480")
```

need to do. In PowerShell, the output and formatting subsystem takes care of this for you. When you want to dump all the fields, the script becomes as simple as this:

```
Get-WmiObject Win32_CodecFile
```

This is where PowerShell has a significant advantage in terms of “whipupitude” over a more programmer-oriented language such as VBScript. It also means you can access much of WMI with simple interactive commands, making it an everyday tool instead of a programmer-only thing.

A.5 POWERSHELL AND PERL

If you can figure out Perl, PowerShell should be a breeze. But Perl programmers need to be aware of a few things. The first two also apply to most other programming languages:

- Functions in PowerShell are invoked like commands.
- The result of a statement isn't voided by default.

These two items are discussed more in the section on C# (A.6.8) and in chapters 7 and 8, which cover functions and scripts.

A couple of things are Perl-specific. Where Perl uses different sigils for different types of variables (\$ for scalar, @ for array, and % for hashtables), PowerShell uses only the dollar sign for all types of variables. Because it's based on .NET, PowerShell has to deal with many more data types than Perl does, so it's not possible to use sigils for each type.

Another significant difference for Perl users is that arrays are passed to functions by reference automatically. If you have a variable containing three objects and you pass it to a function, it will be passed as a single argument containing a reference to the array. Let's look at an example to illustrate this. First you define a function that takes three arguments:

```
PS (3) > function foo ($a,$b,$c) { "a=$a`nb=$b`nc=$c" }
```

Next you invoke it with the arguments 1, 2, and 3:

```
PS (4) > foo 1 2 3
a=1
b=2
c=3
```

Each argument is printed as expected. Now let's define an array containing 1, 2, 3 and pass that array to the function:

```
PS (5) > $a = 1,2,3
PS (6) > foo $a
a=1 2 3
b=
c=
```

This time, the three values all end up in `$a` because the array is passed by reference as a single argument instead of the elements being distributed across the arguments.

Finally, a common question that Perl users ask is whether PowerShell has the equivalent of the Perl `map` operation. The answer is yes—approximately. The `ForEach-Object` cmdlet (or its alias, `%`) is the equivalent of Perl's `map`. The Perl `map` operation looks like this:

```
map <BLOCK> <LIST>
```

The PowerShell equivalent is

```
<LIST> | foreach <BLOCK>
```

In practice, the `ForEach-Object` cmdlet is more powerful than `map` because it also allows initialization and completion blocks:

```
$list | foreach {begin code...} {process code...} {end code...}
```

And because it's a pipelined operation, it's more easily composable than the `map` operator. For example, here's a way to find out what cmdlets have no alias that takes advantage of nested pipelines with `begin` and `end` blocks:

```
gal | %{ $ac = @{} } { $ac[$_.definition] = $true } {
    gcm | ?{ ! $ac[$_.name] }
```

This example initializes a hashtable in `$ac` in the `begin` clause; then, for each alias returned by `gal`, it adds the definition as the hashtable key and sets its value to `true`. Finally, in the `end` clause, it uses the `Where-Object` cmdlet (whose alias is `?`) to filter the output of `gcm` so only commands that don't have entries in the hashtable are emitted.

A.6 POWERSHELL AND C#

PowerShell is syntactically similar to C#. For example, the flow-control statements are mostly the same in PowerShell as they are in C# (except that PowerShell isn't case-sensitive). But C# users encounter a number of common problems when they start

using PowerShell. These problems stem from the fact that PowerShell has shell-like parsing and semantics.

A.6.1 Calling functions and commands

PowerShell functions are commands and are invoked like commands, not like methods. This means that if you have a function called `my-function` that takes three arguments, it will be invoked like this

```
my-function 1 2 3
```

rather than this:

```
my-function(1,2,3)
```

The latter example invokes the command with a single argument that's an array of three values, not three separate arguments.

A.6.2 Calling methods

Methods are invoked in PowerShell as they are in C#, except that spaces aren't permitted between the name of a method call and the opening parenthesis of the arguments. Therefore the expression

```
$data.method($a1, $a2)
```

is valid but

```
$data.method ($a1, $a2)
```

will result in a syntax error. Similarly, spaces aren't permitted around the period (.) between the expression and the method name. These restrictions are needed because of the way PowerShell parses expressions and how it parses command parameters. Because command parameters are separated by spaces, allowing spaces in method calls can lead to confusion. Chapter 2 discusses this topic in much greater detail.

A.6.3 Returning values

PowerShell supports multiple implicit returns from a function. By *implicit*, we mean that values are emitted from a function without using the `return` statement. The following function

```
function foo { 13 }
```

returns the number 13, and the function

```
function bar ( 10; 11; 12 )
```

returns three values: 10, 11, and 12. Although this seems odd in a programming language, it makes perfect sense in a shell (remember, it's named PowerShell for a reason). This characteristic can greatly simplify the code because you don't need to explicitly accumulate the data when you want to return a collection from a function. The system takes care of that for you.

A corollary is that, by default, the return value of a statement isn't voided. This means that if you call a method that returns a value you aren't going to use, you have to explicitly discard it, either by casting it to `[void]` or by redirecting output to `$null`. For example, adding a value to an `ArrayList` returns a number indicating the number of elements in the collection.

A.6.4 Variables and scoping

Unlike most programming languages (but like most shells), PowerShell is dynamically scoped. This means that the variables in the calling function are visible in the called function. Variables come into existence on first assignment and vanish when they go out of scope. You can use scope modifiers to explicitly change variables in other scopes if necessary.

PowerShell doesn't require variables to be typed, but it's possible to add type constraints to them. The semantics aren't quite the same as in C#. In PowerShell, a type-constrained variable will accept any value that can be converted to the constraining type rather than strictly requiring that the value be of the same type or a subtype.

A.6.5 Automatic unraveling of enumerators

Another problem that people run into with .NET methods that return enumerators is that PowerShell will unravel the enumerator. This behavior is correct and by design for PowerShell, but it can be confusing for .NET programmers. Common practice in C# is to use the `Open()` method to get an enumerator, process that enumerator, and then call `Close()`. PowerShell sees the enumerator returned from the `Open()` call and processes it immediately. This is confusing when people try to use the `return` keyword. They expect `return` to return a single value, but

```
return 1,2,3,4
```

is equivalent to

```
1,2,3,4  
return
```

To return an enumerable object, you have to wrap it in another array using the unary comma operator, as in

```
return ,(1,2,3,4)
```

or

```
,(1,2,3,4)  
return
```

NOTE You might think that using the array subexpression operator `@(...)` would work here; but as described in section 5.3.2, all this operator does is guarantee that the result is an array. You need to construct a new one-element array containing the array you want to return.

This new array is discarded in the unraveling process, but its presence ensures that the contained array is returned as a single element.

As an example, say you're writing a function that executes a query against a database. It calls `Open()` to return a database reader object. But this `$reader` object is an enumerator, so instead of being returned as a single element, it's streamed out of the function. For the function to return it atomically, it should look like the following listing.

Listing A.4 `Get-DatabaseReader` function

```
function Get-DatabaseReader ($query , $connection)
{
    $SqlCmd = New-Object System.Data.SqlClient.SqlCommand `
        $query,$connection
    if ( "Open" -ne $connection.state ) { $connection.Open() }
    $reader = $SqlCmd.ExecuteReader()

    , $reader
}
```

You execute the query, and `$reader` is an enumerator for the results of that query. To return the enumerator instead of the results, you use the unary comma.

By doing this, you make the example work like the C# equivalent. But you're not writing C#. To make this more PowerShell-like, consider following the model that commands such as `Get-Content` use. These commands hide the details of opening and closing the stream so the user never has to worry about forgetting to close a handle. The command pushes objects into the pipeline instead of requiring the user to pull them out with a read call. The next listing shows the revised, more PowerShell-like function.

Listing A.5 `Get-FromDatabase` function

```
function Get-FromDatabase ($cmd, $connection)
{
    If ($connection -is [string])
    {
        $conn = New-Object -TypeName System.Data.SqlClient.SqlConnection
        $conn.ConnectionString = $string
        $conn.Open()
    }
    elseif ($connection -is [System.Data.SqlClient.SqlConnection])
    {
        $conn = $connection
        if ( "Open" -ne $conn.s\State ) { $conn.Open() }
    }
    else {
        throw `
            '$connection must be either a database connection or a string'
    }
    $SqlCmd = New-Object System.Data.SqlClient.SqlCommand $cmd,$conn
```

```

        $SqlCmd.ExecuteReader()
    }
    $connection.Close()
}

```

In the revised function, all the open/read/close details are hidden, and you stream the results into the `foreach` cmdlet. Using this revised function to process a query looks like this:

```
Get-FromDatabase $query, $connection | foreach { process-data... }
```

The other advantage this approach provides, besides usability, is that when you write PowerShell functions and scripts, you avoid any problems with the enumerators. The code becomes simpler overall because you don't have to write an explicit loop. PowerShell takes care of all the details.

In summary, if you write PowerShell like PowerShell, it works. If you write PowerShell like C#, you run into problems because PowerShell isn't C#.

A.6.6 Using methods that take path names

You should also be aware that when you're using any .NET method that takes path names, you must always use full path names. This requirement stems from the fact that PowerShell maintains its own idea of what the current working directory is, and this may not be the same as the process current working directory. .NET methods that take paths, on the other hand, always use the process current directory when resolving non-absolute paths.

Let's clarify the current directory question by looking at an example. Start PowerShell, and then use the command `pwd` (which is an alias for `Get-Location`) to see where you are:

```

PS (1) > pwd

Path
----
C:\Documents and Settings\brucepay

```

Use the `CurrentDirectory` static method on the .NET class `System.Environment` to check the process working directory:

```

PS (2) > [System.Environment]::CurrentDirectory
C:\Documents and Settings\brucepay

```

So far they match. Next, use the PowerShell `cd` command to set the PowerShell current working directory to the root of the C: drive, and then verify the path with `pwd`:

```

PS (3) > cd c:\
PS (4) > pwd

Path
----
C:\

```

Fine—everything is as you would expect. But now check the process current working directory:

```
PS (5) > [Environment]::CurrentDirectory
C:\Documents and Settings\brucepay
```

It still points to the original location. Clearly, using `cd` in PowerShell doesn't affect the process current working directory.

Let's look at another reason for always using full path names. `cd` into the root of the Registry:

```
PS (6) > cd hklm:\
PS (7) > pwd
```

```
Path
----
HKLM:\
```

The PowerShell current directory is now in the Registry. This is something the process current directory can't handle; it can only point to some place in the file system. Clearly, the PowerShell and process notions of current directory have to be different.

Let's reiterate why this behavior is a problem when using .NET methods: any .NET method that's passed a relative pathname uses the process current working directory to resolve the path instead of the PowerShell current working directory. To check this out, `cd` back into the root of the C: drive, and create a text file called `hello.txt`:

```
PS (8) > cd c:\
PS (9) > "Hello there" > hello.txt
```

You can get this file from PowerShell using `Get-Content` and specifying a relative path:

```
PS (10) > Get-Content hello.txt
Hello there
```

It works. But when you try using a .NET method and specify a relative path, it fails:

```
PS (11) > [io.file]::ReadAllText("hello.txt")
Exception calling "ReadAllText" with "1" argument(s): "Could not
find file 'C:\Documents and Settings\brucepay\hello.txt'."
At line:1 char:23
+ [io.file]::ReadAllText( <<<< "hello.txt")
```

This is because it's using the process current directory to resolve the relative path, and that's still pointing to the directory where PowerShell was started:

```
PS (12) > [environment]::currentdirectory
C:\Documents and Settings\brucepay
```

The PowerShell environment includes a cmdlet `Resolve-Path`, which is intended to make this scenario easy to work around. When the output of this command is

converted into a string, it's the full provider path to the target object—in this case the file. Let's try this:

```
PS (13) > [io.file]::ReadAllText((Resolve-Path "hello.txt"))
Hello there
```

There is another, even easier way to do this, although it isn't strictly speaking per the guidelines. Instead of `Resolve-Path`, you can use the `$PWD` shell variable along with string expansion to prefix the path:

```
PS (13) > [io.file]::ReadAllText("$pwd\hello.txt")
Hello there
```

Not only is this easier, but it also has the advantage that it will work to produce paths that don't exist yet. If you're creating a file, `Resolve-Path` will fail because it can only resolve existing paths. With the string-expansion approach, this problem doesn't exist. String expansion doesn't know anything about paths—it's giving you a new string.

This is an important rule to keep in mind. If you look at the examples in chapter 17 that use the .NET XML APIs to process files, you always make sure to pass in absolute paths. If you're consistent and always use absolute file paths with methods that take paths, there won't be any problems. (Note that it's usually easiest to use the `Get-Content` cmdlet instead. If you do, everything will work, and you won't have to remember this extra step or close the handle when you're done with it.)

A.6.7 Calling PowerShell from C#

Throughout this book, you've been working with PowerShell as an application, but the core engine is a .NET library that can be used from languages like C#. As a library, PowerShell runs in-process with the rest of the application, allowing live objects to be passed between PowerShell and the host application. The PowerShell API is designed to make it easy to use PowerShell in C# in a natural way. For a given PowerShell pipeline, the equivalent C# code follows the pipeline pattern. Given a PowerShell pipeline like the following

```
Get-Process | select -First 5
```

you can translate it into a series of method calls mimicking the pipeline, as shown in the following fragment of a C# program:

```
foreach (var p in PowerShell.Create().AddCommand("Get-Process")
    .AddCommand("select").AddParameter("First", 5).Invoke<Process>())
{
    Console.WriteLine("The process name is " + p.ProcessName);
}
```

In this example, you use the `PowerShell` class to create a pipeline, adding the `Get-Process` command to it followed by the `select` command, which takes a parameter `First` with value 5. To invoke this pipeline, you use the `Invoke()` method. But because you know that the objects coming back are `Process` objects, you can use the

generic version of `Invoke-Process()` to get back strongly typed objects. This is a simple example with a lot of details omitted; for more complete examples, look at the PowerShell SDK.

A.7 SUMMARY

The goal of this appendix was to provide some context for people coming to PowerShell from other languages. We began by looking at other shells, specifically `cmd.exe` and the UNIX shell. Then we looked at two scripting languages: VBScript and Perl. Finally, we looked at the C# programming language. Notice the broad range of languages we've examined, from simple shells to full programming languages. This is a consequence of the nature of PowerShell—it's an interactive shell, but at the same time it has many of the characteristics of a full-fledged programming language. This breadth of capability makes PowerShell the best option for managing Windows.



A P P E N D I X B

Examples

- | | |
|---|---|
| B.1 Working with files and executables 35 | B.4 .NET examples 52 |
| B.2 WMI examples 39 | B.5 Working with Active Directory and ADSI 66 |
| B.3 Additional COM examples 47 | B.6 Summary 72 |

Although this book isn't intended to be a solutions cookbook, it's good to present as many examples as possible. This appendix contains additional examples in various areas that build on the material presented in the book.

B.1 WORKING WITH FILES AND EXECUTABLES

In this section, we'll look at some more shell-like examples where you're working with files and executables.

B.1.1 Managing schedules with `schtasks.exe`

In section 18.6, you saw how to use the task scheduler COM API to manage scheduled tasks. An alternative approach is to use the existing `schtasks.exe` utility. Because PowerShell is a *shell*, calling external utilities is a perfectly acceptable solution in many cases. The downside to external utilities is that they typically only return text. This means you can't do all the object-based manipulations you're used to with cmdlets. Let's look at how you can fix this.

In this example, you're going to take the text output you get from `schtasks` and convert it into a form that's more usable in PowerShell. Here are the first five lines of text output from this command:

```
PS (1) > schtasks | select -First 5
```

```
Folder: \
TaskName                                     Next Run Time             Status
=====
Microsoft IT DirectAccess - Setup - CORP 5/19/2011 12:00:00 PM Ready
```

Take a look at how this output is structured. With the post-Windows Vista task scheduler, tasks are grouped by folders; and for each folder there are a number of records, one per scheduled task, each of which has three fields. Mixed in with this data are headers and such that need to be discarded. The goal is to convert this stream of formatted text into objects so you can use the PowerShell object commands on them. The converted output of the command should look like this:

```
PS (2) > Get-ScheduledTask | select -First 1 | Format-List
```

```
NextRunTime : 5/19/2011 12:00:00 PM
Status       : Ready
Taskname     : Microsoft IT DirectAccess - Setup - CORP
Folder       : \
```

After the text is converted to objects, it becomes easy to process. You can do things like find the next scheduled task

```
PS (3) > Get-ScheduledTask | sort NextRunTime -Descending |
>> select -First 1 | Format-List
NextRunTime : 6/9/2011 3:00:00 AM
Status       : Ready
Taskname     : PCDoctorBackgroundMonitorTask
Folder       : \
```

or find out how many registered tasks there are

```
PS (4) > (Get-ScheduledTask).Count
81
```

and how many of them are scheduled to run:

```
PS (5) > ( Get-ScheduledTask | where { $_.NextRunTime } ).Count
21
```

The code for the `Get-ScheduledTask` script is shown in the following listing.

Listing B.1 `Get-ScheduledTask` script

```
$props = @{}
$field = 0,0,0
switch -regex (schtasks /query)
{
```

```

'^Folder: *(\\.*)$' {
    $props.Folder = $matches[1]
    continue
}
'^TaskName *Next' { continue}
'^===== ' {
    $field = -split $_ |
        foreach { $_.Length }
    continue
}
'^INFO:' {
    Write-Warning ("In folder $($props.Folder) $_")
    $props = @{}
    continue
}
'^ *$' { $props = @{}; continue }
default {
    $props.Taskname = $_.SubString(0, $field[0]).Trim()
    $props.NextRunTime = try {
        [DateTime] $_.SubString($field[0]+1, $field[1])
    }
    catch
    {
        $null
    }
    $props.Status = $_.Substring(
        $field[0]+$field[1]+1, $field[2]).Trim()
    New-Object PSObject -Property $props;
    continue
}
}

```

1 Get folder name

2 Calculate field lengths

3 Handle warnings

4 Reset on empty line

5 Process field data

The structure of this code is simple: use a `switch` statement to loop over each line of output, extracting useful data piece by piece until you have a complete set of fields, and then emit the new object. Because tasks are organized by folder, the first thing to do is get the folder name ❶. The next line containing the field names is ignored. This is followed by a line of separator characters (=) with spaces between each field. This line is used to calculate the field lengths ❷. In some cases, you won't have access to the tasks in a folder, so turn these error lines, which start with `INFO`, into warning messages ❸. Empty lines are used to separate each task collection; so if you see an empty line, reset the property collection to an empty table ❹. Anything else is treated as a data record. The field information gathered earlier is used to split these records into pieces and then add them to the property collection ❺. There is one issue to address when doing this. Processing the `NextRunTime` field has a problem: you want it to be a `DateTime` object, but if the task is not scheduled to run, this field will contain "N/A" instead of a valid `DateTime` string. You handle this by wrapping the cast in a `try/catch` statement and returning `$null` instead of a `DateTime` object for those fields. When the field collection is complete, you turn it into an object using `New-Object` and emit it to the output.

NOTE A couple of things are missing from this script: proper error handling for badly formatted data and a way to deal with localized messages like the “INFO” message. Addressing these issues is left as exercises for you.

This script shows how simple it is to bring the old world of string-based utilities into the new PowerShell world of objects.

B.1.2 Joining two sets of data

PowerShell cmdlets return collections of data. In many ways, these collections are like data tables. Sometimes you need to combine fields from two collections to produce a new object that includes properties from objects from each of the collections. In effect, what you need to do is execute a join across the two datasets.

A real-world scenario where this occurred was a customer who needed to export a list of mailbox users from an Exchange server to a CSV file, but also needed to merge in some additional data about each user that was stored in a separate CSV file.

Although PowerShell doesn’t have built-in tools to do this, it’s easy to do using hashtables. Here’s the basic solution. Get the first set of data into a hashtable indexed by the primary key property. Then traverse the second set, adding the additional properties extracted from the hashtable. (Or create new objects and add properties from both sets.)

Here’s an example showing how to do this. It merges properties from collections of `Process` and `ServiceController` objects into a single object and then exports the joined result as a CSV file.

Listing B.2 Get-ProcessServiceData.ps1 script

```
Get-Process | foreach {$processes = @{}} {  
    $processes[$_.processname] = $_  
Get-Service |  
    where {$_.Status -match "running" -and  
        $_.ServiceType -eq "Win32OwnProcess" } |  
    foreach {  
        New-Object PSObject -Property @{  
            Name      = $_.Name  
            PID       = $processes[$_.Name].Id  
            WS        = $processes[$_.Name].WS  
            Description = $_.DisplayName  
            FileName  = $processes[$_.Name].MainModule.FileName  
        }  
    } |  
    Export-Csv -NoTypeInformation ./service_data.csv
```

1 Create custom objects

2 Export as CSV file

First you get all the process data into a hashtable indexed by process name. Then, you get the `ServiceController` objects for all the services that are running in their own processes. You build up a new object **1**, extracting fields from service objects and, using the service name to index into the process data hashtable, add the additional

information from the process objects; then you export this information to a CSV file ❷. Note that the `-NoTypeInfo` parameter is used with the `Export-Csv` command—the synthetic object doesn't have a type, so there's no point in including that information.

You can see that this is a simple example, and by replacing the data sources (the cmdlets) and the keys (the names of the properties), you can use this technique to do an arbitrary join between two collections of data.

B.2 WMI EXAMPLES

WMI was introduced in chapter 19. In this section, we'll look at more examples of using WMI classes to get information from our computers.

B.2.1 Getting Active Directory domain information

The script in listing B.3 lists the Active Directory information for a list of computers. If no computer names are provided, it shows the domain information for this computer. You can optionally specify a set of properties to return.

To display the domain information for the current host, use

```
Get-DomainInfo
```

or

```
Get-DomainInfo .
```

To display the domain information for a set of machines, use

```
Get-DomainInfo machine1, machine2, machine3
```

To get the domain information for a list of machines stored in a text file, use

```
Get-Content machines.txt | Get-DomainInfo
```

And to list only the domain name and domain controller name for the current machine, use

```
Get-DomainInfo -Property DomainName, DomainControllerName
```

The code for the script is shown next.

Listing B.3 Get-DomainInfo script

```
param(
    [string[]] $ComputerName = @(),
    [string[]] $Property = @()
)

$ComputerName += @($input)

if (! $ComputerName)
{
    $ComputerName = "."
}
```

```

}

if ($Property.Length -eq 0)
{
    Get-WmiObject -Class Win32_NTDomain `
        -ComputerName $ComputerName
}
else
{
    Get-WmiObject -Class Win32_NTDomain `
        -ComputerName $ComputerName |
        Select-Object $Property
}

```

1 Retrieve information

2 Extract properties

The script uses the `Get-WmiObject` cmdlet ❶ to retrieve the information from a set of machines and return it. If the option list of properties is specified, the `Select-Object` cmdlet ❷ is used to extract those properties from the result set.

B.2.2 Listing installed software features

The script in listing B.4 displays a list of the software features installed on a set of computers. You can optionally specify a list of properties to return (by default, all properties are returned). To show all the properties for the current computer, run

```
Get-SoftwareFeature
```

To get the software features from a list of computers, you can either pass them on the command line

```
Get-SoftwareFeature machine1, machine2, machine2
```

or input them from the pipeline:

```
Get-Content machines.txt | Get-SoftwareFeature
```

You can also specify a subset of the properties to display. For example, to display only the vendor and caption fields, you would do

```
Get-SoftwareFeature -Property Vendor, Caption
```

The listing for this script is shown next.

Listing B.4 `Get-SoftwareFeature.ps1` script

```

param(
    [string[]] $ComputerName = @(),
    [string[]] $Property = @()
)

$ComputerName += @($input)

if (! $ComputerName)
{
    $ComputerName = "."
}

```

```

if ($Property.Length -eq 0)
{
    Get-WmiObject -Class Win32_SoftwareFeature `
        -ComputerName $ComputerName
}
else
{
    Get-WmiObject -Class Win32_SoftwareFeature `
        -ComputerName $ComputerName |
        select $Property
}

```

1 Retrieve information

2 Extract properties

As in the previous example, `Get-WmiObject` is used to retrieve the data ❶ and optionally filter it ❷.

B.2.3 Retrieving terminal server properties

Terminal server properties can also be retrieved using simple WMI queries. For example, to list the terminal server service properties on the current machine, use the following command:

```
Get-WmiObject -Class Win32_TerminalService -ComputerName .
```

To list the terminal services accounts, use the `Win32_TSAccount` object as follows:

```
Get-WmiObject -Class Win32_TSAccount -ComputerName . |
    select AccountName, PermissionsAllowed
```

To get the terminal services remote control setting from a computer, you can do this:

```
Get-WmiObject Win32_TSRemoteControlSetting |
    Select-Object TerminalName, LevelOfControl
```

Note that this example uses the fact that the `-Class` parameter is positional so you don't have to specify `-Class`. You also use the default value for `-ComputerName` with a dot (`.`)—the current computer.

To see a list of all the WMI classes that can be used for managing terminal services, run the following command:

```

PS (1) > Get-WmiObject -list |
>> where {$_.name -like "Win32_TS*"} | select name
>>

Name
----
Win32_TSNetworkAdapterSettingError
Win32_TSRemoteControlSettingError
Win32_TSEnvironmentSettingError
Win32_TSSessionDirectoryError
Win32_TSLogonSettingError
Win32_TSPermissionsSettingError
Win32_TSClientSettingError
Win32_TSGeneralSettingError
Win32_TSSessionSettingError

```

```

Win32_TSSessionDirectory
Win32_TSRemoteControlSetting
Win32_TSNetworkAdapterSetting
Win32_TSAccount
Win32_TSGeneralSetting
Win32_TSPermissionsSetting
Win32_TSClientSetting
Win32_TSEnvironmentSetting
Win32_TSNetworkAdapterListSetting
Win32_TSLogonSetting
Win32_TSSessionSetting
Win32_TSSessionDirectorySetting

```

This command searches all the WMI classes looking for those that have names starting with the sequence Win32_TS.

B.2.4 Listing hot fixes installed on a machine

The script in listing B.5 lists the hot fixes installed on a list of computers. This example is similar to the [Get-HotFix](#) cmdlet included with PowerShell V2. Here you'll see how you might implement such a cmdlet.

In the script, if no computer names are provided, it shows the hot fix information for this computer. You can optionally specify a set of properties to return. To get a list of all hot fixes installed on the current computer displaying all properties, do this:

```
Get-HotFix
```

If you only want to see certain properties, use this:

```
Get-HotFix -Prop ServicePackInEffect,Description
```

The listing for this script is shown next.

Listing B.5 Get-HotFix.ps1 script

```

param(
    [string[]] $ComputerName = @("."),
    [string[]] $Properties = @()
)

if ($Properties.Length -eq 0)
{
    Get-WmiObject -Class Win32_QuickFixEngineering `
        -ComputerName $ComputerName
}
else
{
    Get-WmiObject -Class Win32_QuickFixEngineering `
        -ComputerName $ComputerName |
        select-object $properties
}

```

At this point, you can see that there is a consistent pattern for all these examples. Once you know the WMI class for a particular feature, the pattern for getting information about that feature is the same. PowerShell makes it easy to use WMI on the command line to retrieve information about the system when you know the class name.

B.2.5 Finding machines missing a hot fix

Let's build on the script from the previous example to accomplish a more specific task. You'll write a new script that will search computers for missing hot fixes. Here's what you want the output to look like

```
PS (1) > ./Get-MachinesMissingHotfix.ps1 -computer . `
>> -hotfix KB902841,KB902842,KB902843,KB902844
>>

Name                                Value
----                                -
name                                .
missing                            {KB902842, KB902843, KB902844}
```

This result of the command shows that three of the four hot fixes aren't installed on the current machine.

NOTE Some of these hot fix identifiers are fictitious so you can see some failures. So don't be worried if you can't find them in the knowledge base.

Notice that the output retains structure. Instead of emitting strings, the script will emit hashtables so they can more easily be used in further processing such as building update packages for distribution. And because you want to be able to check a list of machines, the script can either take the list on the command line or read it from input stream as shown in the next example:

```
PS (2) > Get-Content machines.txt | ./Get-MachinesMissingHotfix.ps1 `
>> -hotfix KB902841,KB902842,KB902843,KB902844
>>

Name                                Value
----                                -
name                                machine1
missing                            {KB902842, KB902843, KB902844}
name                                machine4
missing                            {KB902842, KB902843, KB902844}
name                                machine5
missing                            {KB902841,KB902842, KB902843, KB902844}
```

The file `machines.txt` contains a list of machine names `machine1` through `machine5` to check. The output indicates that machines 2 and 3 are up to date—they don't appear in the output. Machines 1 and 4 are missing three hot fixes, and machine 5 is missing all four.

This script is shown in the following listing.

Listing B.6 Get-MachinesMissingHotfix.ps1 script

```
param(
    [string[]]
        $ComputerName = @("."),
    [Parameter(Mandatory=$true)]
        [string[]] $HotFix
)

$myDir = Split-Path $MyInvocation.MyCommand.Definition
$gh = Join-Path $myDir Get-HotFix.ps1

foreach ($name in $ComputerName)
{
    $sps = & $gh $name | foreach { $_.ServicePackInEffect}
    $result = @{name = $name; missing = @() }

    foreach ($hf in $HotFix)
    {
        if ($sps -notcontains $hf)
        {
            $result.missing += $hf
        }
    }
    if ($result.missing.length -gt 0)
    {
        $result
    }
}
```

1 Find path to Get-HotFix

2 Initialize result table

3 Add missing hot fixes

4 Emit result

This script takes two parameters: the list of computer names to check and the list of hot fixes to check for. The `$HotFix` parameter is required, but the list of computers is optional and defaults to `.` (the current computer).

You're requiring that the `Get-HotFix` script be in the same directory as this script. Given that, you can figure out the path to the `Get-HotFix` script by getting the path ❶ to the current script, which is available in `$MyInvocation`, and then use this to build the path to the `Get-Hotfix` script.

NOTE This is a useful technique to keep in mind when you're writing other scripts. For modules, you can use the `$PSScriptRoot` variable instead.

Once you have the path to the `Get-HotFix` command, you use it to get the list of hot fixes; but you want only the `ServicePackInEffect` field, so you'll use the `foreach` command to extract only this property.

You initialize the variable `$result` to be a hashtable object with the current machine name and set the list of missing hot fixes to be an empty array ❷. Note that

you may not return this object if there are no missing hot fixes. You check that by seeing whether the length of the missing member in that hashtable is 0.

Now you loop over the list of hot fixes, checking each hot fix to see whether it's in the list installed on the target machine. If the list of installed hot fixes doesn't contain the current hot fix identifier, append that identifier to the missing array ❸ in the result hashtable.

Finally, if after checking all the hot fixes, the missing array in the hashtable is still of length zero, this machine has all the hot fixes installed. If the array is non-zero, then you emit the `$result` object ❹.

B.2.6 Associations and related classes in WMI

All the object types that PowerShell works with have the basic member types—properties and methods. WMI is a little different in that it also has *associations*. For example, there is an association between a disk and the partitions on that disk that allows you to get partitions associated with a disk and also the disk on which a partition lives.

How associations work

Associations aren't properties of any particular class. Instead, they're represented as classes in their own right. Each association class has two cells that are used to link the two classes for which there is an association. This looks something like what is shown in figure B.1.

In this figure, `Class 1` has three related classes: `Related Class 1`, `Related Class 2`, and `Related Class 3`. The relations are defined by three instances of `Association Class A`. Similarly, `Class 2` is related to `Related Class 2` and `Related Class 3` by instances of `Association Class B`. Let's see how this works using the `Win32_DiskDrive` class. First you need to get an instance of this class, which you can do through the `Get-WmiObject` cmdlet:

```
PS (1) > $disk = Get-WmiObject Win32_DiskDrive | select -First 1
```

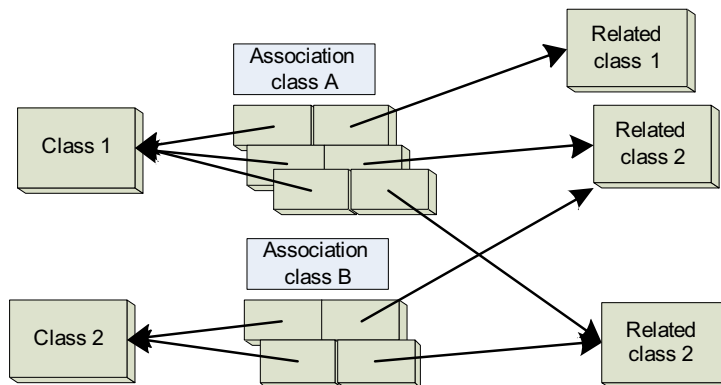


Figure B.1 This diagram shows how association classes are used to create relationships between objects.

The output of this command is saved in the variable `$disk`. Let's see what this object looks like:

```
PS (2) > $disk

Partitions : 3
DeviceID   : \\.\PHYSICALDRIVE0
Model      : HITACHI HTS722020K9SA00   FDE ATA Device
Size       : 200046551040
Caption    : HITACHI HTS722020K9SA00   FDE ATA Device
```

This tells you a lot about the disk, including the number of partitions:

```
PS (3) > $disk.Partitions
3
```

But it doesn't tell you anything about those partitions. To get that information, you need to use the association classes. First get the path that uniquely identifies the disk you're looking at:

```
PS (4) > $path = $disk.__PATH
PS (5) > $path
\\BRUCEPAYX61\root\cimv2:Win32_DiskDrive.DeviceID= "\\.\PHYSICALDRIVE0"
```

Now you can use this path in an association query. You need to ask for all the associators of this class, which you do with the following command:

```
PS (6) > Get-WmiObject -Query "ASSOCIATORS OF {$( $path )} WHERE
>> AssocClass = Win32_DiskDriveToDiskPartition"
>>
```

```
NumberOfBlocks : 12058624
BootPartition   : False
Name            : Disk #0, Partition #0
PrimaryPartition : True
Size            : 6174015488
Index           : 0
```

```
NumberOfBlocks : 204800
BootPartition   : True
Name            : Disk #0, Partition #1
PrimaryPartition : True
Size            : 104857600
Index           : 1
```

```
NumberOfBlocks : 378454016
BootPartition   : False
Name            : Disk #0, Partition #2
PrimaryPartition : True
Size            : 193768456192
Index           : 2
```

In the output from this query, you can see information about each of these disks. This works, but writing the association query can be tedious. The WMI objects

themselves make this type of operation easier through the `GetRelated()` method. First you can call the `GetRelated()` method with no arguments to see what kind of associations there exist for the disk class:

```
PS (7) > $disk.GetRelated() | foreach { $_.CreationClassName } |  
>> sort -Unique  
Win32_ComputerSystem  
Win32_DiskPartition  
Win32_PnPEntity
```

This gives you the name of the classes for which there is an association. Now you can pass the type name of the associated. You have to pass the name of the class to the `GetRelated()` method, and you can see all the partition information:

```
PS (8) > $disk.GetRelated("Win32_DiskPartition") |  
>> select Name, PrimaryPartition,@{n="Size"; e={$_.Size / 1gb }}  
>>  
Name                                PrimaryPartition                Size  
----                                -  
Disk #0, Partition #0                True                             5.75  
Disk #0, Partition #1                True                             0.09765625  
Disk #0, Partition #2                True                             180.4609375
```

You can also use this method to get other associated classes. For example, you can get information about the computer on which this disk is installed by retrieving the associated `Win32_ComputerSystem` class:

```
PS (9) > $disk.GetRelated("Win32_ComputerSystem")  
  
Domain                : redmond.corp.microsoft.com  
Manufacturer          : LENOVO  
Model                 : 7764CTO  
Name                  : BRUCEPAYX61  
PrimaryOwnerName      : brucepay  
TotalPhysicalMemory   : 4217683968
```

This shows the computer system class instance associated with this computer.

B.3 *ADDITIONAL COM EXAMPLES*

In this section, we'll look at a few more examples showing how to use COM from PowerShell. We'll also cover how to work with COM objects that don't support `IDispatch`.

B.3.1 *Using the WinHTTP class to retrieve an RSS feed*

Let's look at using the `WinHTTP` COM object to write a script that accesses an RSS feed. This is similar to what you did with .NET but illustrates how to use COM to do the same thing. The script will grab the most recent headlines from the popular digg.com RSS feed, format them as a page of links in HTML, and then display this page using the default browser.

First you define a function `Get-ComRSS` that will do the network access. This is shown in the following listing.

Listing B.7 `Get-ComRSS` function

```
function Get-ComRSS
{
    param($url = $(throw "You must specify a feed URL to read"))

    $objHTTP = New-Object -ComObject WinHTTP.WinHttpRequest.5.1
    $objHTTP.Open("GET", $url, $false)
    $objHTTP.SetRequestHeader("Cache-Control",
        "no-store, no-cache, must-revalidate")
    $objHTTP.SetRequestHeader("Expires",
        "Mon, 26 Jul 1997 05:00:00 GMT")

    $objHTTP.Send()
    $xmlResult = [xml]$objHTTP.ResponseText
    $xmlResult.rss.channel.item | Select-Object title, link
}
```

This function creates the `WinHTTP` request object, specify that you're doing a page `GET`, and then set some headers. These headers tell the channel not to do any caching. Because you want to get the latest and greatest headlines, getting stale cached data would be bad.

You send the request and then get the response text (note that you're not checking the result code from the request, which you probably should do). You take the response text, convert it into XML, and extract and return the title and link fields.

Now let's use this function. You'll write a script called `Get-Digg.ps1` that will download the RSS feed from the popular news-aggregation site digg.com, format it as a web page with links to the articles, and then display this page in the browser. You can run this script by typing

```
PS (1) > ./Get-Digg
```

After the script runs, the web browser should open, displaying a page like that shown in figure B.2.

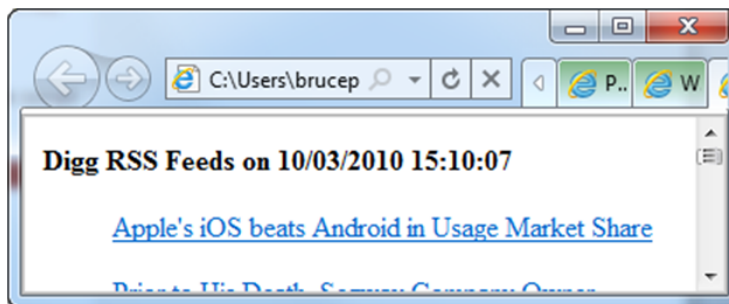


Figure B.2 Web browser showing the result of running the `Get-Digg` script. This script creates an HTML page with links to the current top stories on Digg.com and then displays this page in the browser.

Not the most exciting document in terms of appearance, but it gets the job done. The script to do this is shown in the next listing.

Listing B.8 Get-Digg script

```
$url = "http://digg.com/rss/indexnews.xml"
filter fmtData {
    "<p><a href='{0}'>{1}</a></p>" -f $_.link, $_.title
}

@"
    <html>
    <head>
        <title>Digg RSS Feed</title>
    </head>
    <body>
        <p><b>Digg RSS Feeds on $(get-date)</b></p>
        <ul>
            $(Get-ComRSS $url | fmtData)
        </ul>
    </body>
</html>
"@ > $env:temp\digg_rss.htm

& $env:temp\digg_rss.htm
```

First you put the URL you're going to fetch into a variable to use later. You also create a function to format your data with appropriate HTML tags. Each data row will be formatted as an anchor element with the body text as the element title and the HREF as the link. Next you build the document. You use a single here-string directed into a temporary file. In the here-string, you use string expansion to insert the headline data using the `fmtData` filter. The final step is to invoke this file using the default browser.

Obviously, a little work with table tags could make the result much more attractive. Also, because the main article content was also downloaded in the HTTP request, it should be possible to embed the content (or at least a synopsis) of the article in the page. This is left as an exercise for you.

B.3.2 Dealing with COM objects that don't support IDispatch

All the COM objects you've seen so far have been *self-describing*. This allows you to discover and use all the members on the object like any other PowerShell object. Now let's look at the situation when there is no *type library* for an object. This makes the object much more difficult to deal with because you don't have any type information for it. But it's still possible to do quite a bit with these objects. Let's work through an example using the Windows installer class `WindowsInstaller.Installer` to see how much you can accomplish.

First, you create an instance of the installer object:

```
PS (45) > $in = New-Object -Com WindowsInstaller.Installer
PS (46) > $in | Get-Member
```

TypeName: System.__ComObject

Name	MemberType	Definition
-----	-----	-----
CreateObjRef	Method	System.Runtime.Remo...
Equals	Method	System.Boolean Equa...
GetHashCode	Method	System.Int32 GetHas...
GetLifetimeService	Method	System.Object GetLi...
GetType	Method	System.Type GetType()
InitializeLifetimeService	Method	System.Object Initi...
ToString	Method	System.String ToStr...

When you run `Get-Member` on the result object, the output is underwhelming. What you see is an object whose type is simple `System.__ComObject` with next to nothing in the way of methods or properties on it. Is there more to this object? Can you do anything with it? The answer to these questions is yes, but it's not easy. You can use a couple of approaches.

The first thing you can do is use a tool such as `tlbimp.exe` to generate a *runtime-callable wrapper* (RCW) for the COM class. You use a tool to build the type information you didn't get by default. With this RCW wrapper, you can use the wrapped class the way you normally do. This works well, but it means you have to run the tool and then load the generated assembly before you can use these objects. This makes it significantly more complex to deploy or share the script. Let's look at a more technically complex but also more portable mechanism.

Instead of generating a wrapper with `tlbimp.exe`, you can use .NET reflection to build your own PowerShell-based wrapper library. Let's see how to do this.

NOTE This is an advanced topic and requires a pretty good understanding of `System.Reflection` to accomplish. If you run across this kind of problem and don't feel comfortable working with reflection, chances are good that someone in the PowerShell community has already solved the problem, so consulting the community resources may get you the answer you need.

First, you create a types extension file called `ComWrappers.ps1xml`. The following is a fragment from that file showing how the `InvokeMethod` extension method is defined:

```
<ScriptMethod>
<Name>InvokeMethod</Name>
<Script>
    $name, $methodargs=$args
    [System.__ComObject].invokeMember($name,
        [System.Reflection.BindingFlags]::InvokeMethod,
        $null, $this, @($methodargs))
</Script>
</ScriptMethod>
```

NOTE The complete types file is included with the source code for the book, which you can download from <http://manning.com/payette2/WPSiA2SourceCode.zip>

This script method uses the `InvokeMember` method on the type object to invoke a dynamically discovered method. In the type extension file, there are similar implementations for getting and setting properties. Load `ComWrappers.ps1xml`, and then examine the `WindowsInstaller` object again:

```
PS (1) > Update-TypeData ./ComWrappers.ps1xml
PS (2) > $in = New-Object -Com WindowsInstaller.Installer
PS (3) > $in | gm

        TypeName: System.__ComObject

Name                                     MemberType Definition
-----
CreateObjRef                             Method      System.Runtime.Re...
Equals                                    Method      System.Boolean Eq...
GetHashCode                               Method      System.Int32 GetH...
GetLifetimeService                       Method      System.Object Get...
GetType                                   Method      System.Type GetTy...
InitializeLifetimeService                Method      System.Object Ini...
ToString                                  Method      System.String ToS...
GetProperty                               ScriptMethod System.Object Get...
InvokeMethod                             ScriptMethod System.Object Inv...
InvokeParamProperty                     ScriptMethod System.Object Inv...
SetProperty                               ScriptMethod System.Object Set...
```

You can see the methods you added at the end of the list. Now let's look at how you can use these methods. You'll use the `WindowsIntaller` class to look at an MSI file called `myapplication.msi`. The code to do this is shown here.

Listing B.9 Get-FilesInMsiPackage script

```
$installer = New-Object -Com WindowsInstaller.Installer
$database = $installer.InvokeMethod("OpenDatabase", $msifile, 0) 1 Open database

$view = $database.InvokeMethod("OpenView",
    "Select FileName FROM File")
$view.InvokeMethod("Execute");

$r = $view.InvokeMethod("Fetch");
$r.InvokeParamProperty("StringData", 1);
while($r -ne $null)
{
    $r = $view.InvokeMethod("Fetch");
    if( $r -ne $null)
    {
        $r.InvokeParamProperty("StringData", 1);
    }
} 2 Iterate through package
```

You create the object and use the `InvokeMethod()` call to invoke the `OpenDatabase()` installer method ①, passing it the full path to the MSI file. Next you open a view on the installer database object to get a list of the files. And finally, you iterate ② through the contents of the installer package.

B.4 .NET EXAMPLES

We covered a number of examples where you built GUIs in chapter 17. Creating a GUI front-end for scripts makes them more usable by non-command-line users. We'll look at a few more examples of how to do this in this section. But first, we'll discuss how to do pure graphics programming from PowerShell, which can be useful for creating graphs and charts.

B.4.1 Using GDI+ to do graphics

In this example, you're going to do some graphics programming. All of the previous GUI examples have depended on the controls to do the drawing. Now you'll draw directly with PowerShell. In the process, you'll see how to use the `Paint` and `Timer` events on a `Form` object.

Graphics programming in Windows (at least, in the post-XP/Server 2003 world) is done using a set of APIs called GDI+. GDI stands for Graphics Device Interface. It's the abstraction that Windows uses to hide the details of working with specific graphics hardware. In .NET, this API surfaces through the `System.Drawing` collection of namespaces. The particular example is a script that draws a spiral on a form. This form is shown in figure B.3.

The script draws a spiral out from the center of the form. It periodically redraws the form, changing the foreground and background colors on each iteration. This redraw is handled by the `Timer` event on the form. Resizing the form also triggers the `Paint` event to cause the spiral to be redrawn.

The script takes parameters that allow you to specify the opacity (or translucency) of the form, as well as its initial size and the amount of detail used in drawing the form, as shown in the following listing.

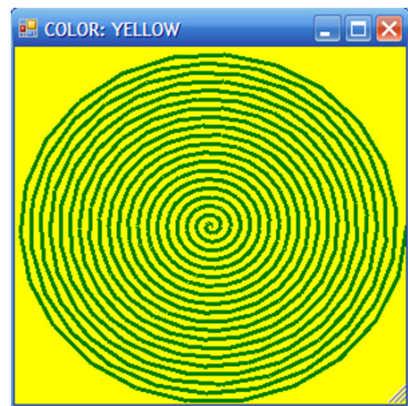


Figure B.3 Screen capture of the PwrSpiral GDI+ example form. This form is drawn by a PowerShell script. It redraws itself in a different color every 5 seconds. It can also be started such that it's displayed as a transparent window.

Listing B.10 The annotated PwrSpiral.ps1 script.

```
[CmdletBinding()]  
param(  
    $opacity=1.0,  
    $increment=50,  
    $numRevs=20,  
    $size=(500,500)  
)
```

← **1** Script parameters

```

Import-Module wpiaforms

$colors = . {$args} red blue yellow green orange `
    black cyan teal white purple gray
$index=0
$color = $colors[$index++]

$form = New-Control Form @(
    TopMost=$true
    Opacity=$opacity
    Size=size $size[0] $size[1]
)
$myBrush = Drawing SolidBrush $color
$pen = Drawing pen black @(Width=3)
$rec = Drawing Rectangle 0,0,200,200

function Spiral($grfx)
{
    $cx, $cy =$Form.ClientRectangle.Width,
        $Form.ClientRectangle.Height
    $iNumPoints = $numRevs * 2 * ($cx+$cy)
    $cx = $cx/2
    $cy = $cy/2
    $np = $iNumPoints/$numRevs
    $fAngle = $i*2.0*3.14159265 / $np
    $fScale = 1.0 - $i / $iNumPoints
    $x,$y = ($cx * (1.0 + $fScale * [math]::cos($fAngle))),
        ($cy * (1.0 + $fScale * [math]::sin($fAngle)))

    for ($i=0; $i -lt $iNumPoints; $i += 50)
    {
        $fAngle = $i*2.0*[math]::pi / $np
        $fScale = 1.0 - $i / $iNumPoints
        $ox,$oy,$x,$y = $x,$y,
            ($cx * (1.0 + $fScale * [math]::cos($fAngle))),
            ($cy * (1.0 + $fScale * [math]::sin($fAngle)))
        $grfx.DrawLine($pen, $ox, $oy, $x, $y)
    }
}

$handler = {
    $rec.width = $form.size.width
    $rec.height = $form.size.height
    $myBrush.Color = $color
    $formGraphics = $form.CreateGraphics()
    $formGraphics.FillRectangle($myBrush, $rec)
    $form.Text = "Color: $color".ToUpper()
    $color = $colors[$index++]
    $index %= $colors.Length
    $pen.Color = $color
    Spiral $formGraphics
    $formGraphics.Dispose()
}

$timer = New-Object system.windows.forms.timer

```

2 Load wpiaforms module

3 Set up colors

4 Build top-level form

5 Spiral function

6 Initialize values

7 Draw spiral

8 Timer event handler

9 Set up timer event

```
$timer.interval = 5000
$timer.add_Tick($handler)
$timer.Start()
```

```
$Form.add_paint($handler)
```

10 Add paint handler

```
$form.Add_Shown({$form.Activate()})
[void] $form.ShowDialog()
```

11 Show form

First you define the parameters ❶ for the script (remember, the `param` statement always has to be the first executable statement in the script). Specifying the `[Cmdlet-Binding()]` attribute causes the runtime to generate an error if too many parameters are specified. (See section 8.2.2.) Opacity of a form is a built-in capability in the GDI+, allowing for some cool visual effects.

An opacity of 1 is a solid form. The spiral is drawn using a series of line segments. The more segments there are, the smoother the curve, but the longer it takes to draw. This is controlled by the `$increment` parameter. The `$numRevs` parameter controls the number of revolutions used in drawing the spiral. The spiral will always fill the form, so the greater the number of revolutions, the closer together the curves will be.

You still need to load the basic assemblies and create a form to draw on, so you load the usual `wpiaforms` utility module (see chapter 17) ❷.

You want the spiral to be drawn in a different color on each iteration, so you set up a list ❸ of the colors to loop through. The `$index` variable is used to keep track of the last color used.

Next you create the objects ❹ you need: a top-level form, passing in the size and opacity arguments to the script; and a couple of drawing objects—a brush to do the drawing and a rectangle to use for drawing the form background.

The `Spiral` function ❺ is the routine that does all the drawing. It takes a graphics context to draw on and then uses the information about the number of revolutions and the increment to calculate the number of segments to draw.

Once you have all of the basic information—the number of points and the angle of rotation—calculated ❻, you loop ❼, drawing each segment until the spiral is complete. You use multivariable assignment in the loop to simplify the code and speed things up.

Next you create a scriptblock ❽ that you can use as the event handler for triggering drawing the spiral. This handler creates the graphics object for drawing on the form, fills it with the background color, and then calls the `Spiral` routine to draw on the graphics object.

With the basic pieces in place, you can create the timer control ❾ and add it to the form to trigger the redraws. The script sets up the timer control's interval to redraw the form every 5 seconds.

Any activity that triggers the paint 10 event will cause the spiral to be redrawn. For example, resizing the form will cause a new paint cycle. Finally, you show the form 11, blocking it from view until it's closed.

This example shows additional uses of the scriptblock as a timer event handler as well as using the `[math]` capabilities to do some fairly complex calculations. It's not a particularly practical application, but it gives you the basics of how to write an application that graphs a sequence of values.

B.4.2 Implementing a simple calculator

In this section, you'll build your own replacement for the Windows calculator applet. This exercise shows how the `wpi-aforms` module can help construct an application. We'll also use this example to introduce the table layout manager. In the process, you'll see how some of PowerShell's features can be used to create a flexible application architecture. At the end of the exercise, the result will be an application that looks like what's shown in figure B.4.

It's not as fancy as the calculator that ships with Windows, but it has more functions and, because it's a script, you can add custom features when you want to. The basic structure includes a simple File menu with two actions—Clear and Exit. Next is a text box that will hold the value of the most recent calculation. Finally, there are all the buttons in the calculator. This is where the table layout manager is important; you don't want to have to lay out each of these buttons by hand. (Even in an interface builder such as Visual Studio, this would be tedious.) The `TableLayoutPanel` allows you to lay out a grid of controls. It has a `ColumnCount` property that lets you control the number of columns that are used in laying out the buttons. You'll design the application so that by changing this value, you can lay out the buttons in two columns, producing a tall, skinny calculator; or set it to 10 columns, producing a shorter, wider layout.

Start the calculator script by loading the forms library:

```
Import-Module wpi-aforms
```

Next you'll set up some variables and functions that you'll use later in the script. The `clr` function will set the calculator back to zero. The variable `$op` holds the pending operation. This is the operation the user has clicked, but it won't be executed until `=` or another operation is selected:

```
$script:op = ''
$script:doClear = $false
```

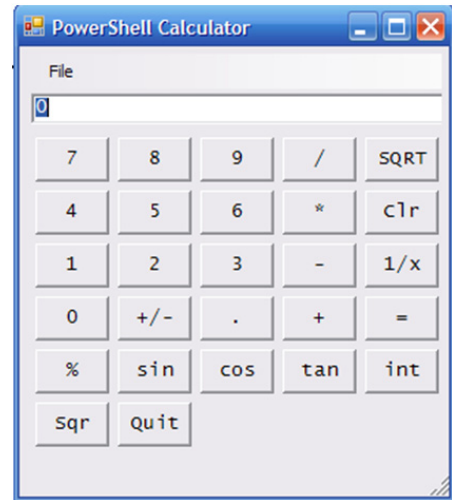


Figure B.4 The PowerShell calculator example form. This example uses the `WinForm` library to construct an extensible graphical calculator application.

```
function clr { $result.text = 0 }
[decimal] $script:value = 0
```

The basic logic for handling a number-key click is the same, so you'll build a common scriptblock for it. This scriptblock will incrementally append numbers to the display, resetting it when appropriate:

```
$handleDigit = {
    if ($doClear)
    {
        $result.text = 0
        $script:doClear = $false
    }

    $key = $this.text
    $current = $result.Text
    if ($current -match '^0$|NaN|Infinity')
    {
        $result.text = $key
    } else {
        if ($key -ne '.' -or $current -notmatch '\.')
        {
            $result.Text += $key
        }
    }
}
```

Clicking one of the operation keys such as + or - is handled in much the same way for all the keys, so again you'll define a common scriptblock to handle it. This scriptblock saves the current calculated value as well as the operation to perform, and then allows the user to enter the next value in the calculation:

```
$handleOp = {
    $script:value = $result.text
    $script:op = $this.text
    $script:doClear = $true
}
```

The next step is to build a table that holds all the actions that will be associated with each key. This table will bind these actions to all the button controls. It's defined as an array of hashtables. Each hashtable has a member name that specifies the name to put on the key and an action to perform when the key is clicked. This action is implemented by a scriptblock. Most of the keys call either the `handleDigit` scriptblock or the `handleOp` scriptblock, but some have a custom scriptblock:

```
$keys = {
    @{name='7'; action=$handleDigit},
    @{name='8'; action=$handleDigit},
    @{name='9'; action=$handleDigit},
    @{name='/'; action = $handleOp},
```

For example, the `SQRT` key implements the square root function. It uses the static method `[math]::sqrt()` to do the calculation. Because having your calculator

throw exceptions isn't friendly, you also add a trap handler to catch and discard any exceptions that may occur when calculating the square root. You'll see this pattern again in some of the other trap handlers:

```
@{name='SQRT'; action = {
    try
    {
        $result.Text = [math]::sqrt([decimal] $result.Text)
    }
    Catch
    {
        $result.Text = 0; continue }
    }
},
```

The rest of the key-binding definitions are similar, so we'll skip over them and move on to the code that defines the form for the calculator. You use the [New-Control](#) function to define the top-level form and each of the controls it contains:

```
$form = New-Control Form @{
    Text = "PowerShell Calculator"
    TopLevel = $true
    Padding=5
}
$table = New-Control TableLayoutPanel @{
    ColumnCount = 1
    Dock="fill"
}
$form.controls.add($table)
```

You use another function from the module to add a menu to the form:

```
$menu = New-MenuStrip $form {
    New-Menu File {
        New-MenuItem "Clear" { clr }
        New-Separator
        New-MenuItem "Quit" { $form.Close() }
    }
}
$table.Controls.Add($menu)
$cfont = New-Object Drawing.Font
    'Lucida Console',10.0,Regular,Point,0
$script:result = New-Control TextBox @{
    Dock="fill"
    Font = $cfont
    Text = 0
}
$table.Controls.Add($result)
```

Lay out the form as a table with five columns:

```
$columns = 5

$buttons = New-Control TableLayoutPanel @{
    ColumnCount = $columns
```

```

        Dock = "fill"
    }

```

When the table has been constructed, you need to add the buttons to it. Use the hashtable of button definitions you created earlier to do this. Loop through the [Keys](#) property to get the names of the buttons and add each control to the table as follows:

```

foreach ($key in $keys) {
    $b = New-Object Button @{
        text=$key.name
        font = $cfont;
        size = size 50 30
    }
    $b.add_Click($key.action)
    $buttons.controls.Add($b)
}
$table.Controls.Add($buttons)

```

The last thing you have to do is calculate the final size for the form and then call [ShowDialog\(\)](#) to display it. Here is the code to do that:

```

$height = ([math]::ceiling($keys.count / $columns)) *
    40 + 100
$width = $columns * 58 + 10
$result.size = size ($width - 10) $result.size.height
$form.size = size $width $height
$form.Add_Shown({$form.Activate()})
[void] $form.ShowDialog()

```

This completes the calculator example. By using a hashtable to define your button definitions, you can eliminate a lot of redundant code when constructing a GUI that includes a lot of similar controls.

B.4.3 A graphical process viewer

A common activity in user interface programming is displaying collections of data, usually in a table or grid.

NOTE This activity is so common that we added the [Out-GridView](#) cmdlet in PowerShell V2. What you're building here is somewhat similar to that cmdlet, but with substantially fewer features. But unlike [Out-GridView](#) in its current form, this example can be customized for specific applications.

The [Windows Forms](#) framework makes this easy through a feature called *data binding*. Data binding is the ability to tell a control such as a grid to use a collection of objects as the data it should display. You don't have to write any code; the control figures everything out by examining the data. PowerShell objects ([PSObjects](#)) also support data binding, so you can take the output from a pipeline and use that as the data source for a control. In this section, you'll work through a short script that does exactly this. You'll take the output of the [Get-Process](#) cmdlet and display it in a grid

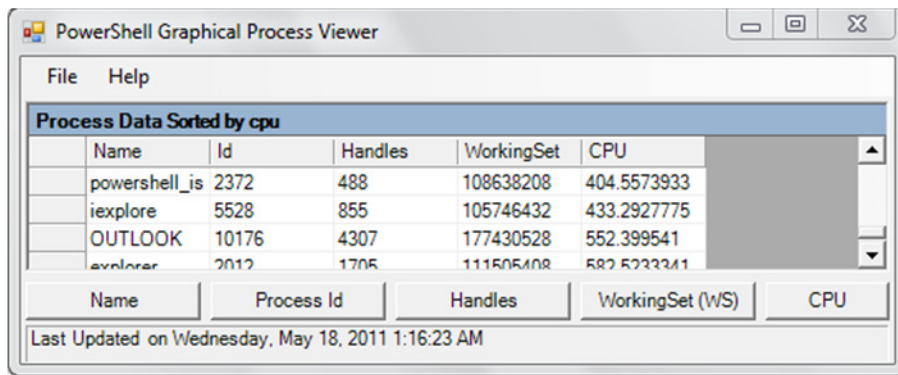


Figure B.5 The PowerShell graphics process viewer form. Clicking any of the buttons causes the data to be refreshed and sorted based on the property named by the button.

on a form. In the process, we'll look at additional features of the [TableLayoutPanel control](#). The resulting form is shown in figure B.5.

This GUI shows you information about all the running processes and provides buttons that enable the user to sort the information by name, id, and other criteria. The code that implements this form is shown in the following listing.

Listing B.11 PowerShell graphical process viewer

```
Import-Module wpiaforms

$sortCriteria="ProcessName"
function Update-GridData ($sortCriteria="ProcessName") {
    $grid.DataSource = New-Object System.Collections.ArrayList `
        (, (Get-Process |
            sort $sortCriteria |
            select name,id,handles,workingset,cpu))
    $grid.CaptionText = "Process Data Sorted by $sortCriteria"
    $status.Text =
        "Last Updated on $(Get-Date | Out-String)" -replace "`n"
}

$form = New-Object Form @{
    AutoSize=$true
    Text = "PowerShell Graphical Process Viewer"
    Events = @{
        Shown = {Update-GridData}
    }
}

$menu = New-Object MenuStrip $form {
    New-Object File {
        New-MenuItem "Update" { Update-GridData }
        New-Separator
        New-MenuItem "Quit" { $form.Close() }
    }
}
```

1 Define update function

2 Create main form

3 Create menus


```

New-Menu Help {
    New-MenuItem "About" {
        Show-Message (
            "PowerShell Process Viewer`n`n" +
            "Windows Forms Demo Applet`n" +
            "From Windows PowerShell in Action`n"
        )
    }
}
}

$grid = New-Control DataGridView @{
    Dock="fill"
    CaptionText = "PowerShell Graphical Process Viewer"
}

$table = New-Control TableLayoutPanel @{
    ColumnCount=6
    Dock="Fill"
    AutoSizeMode = "GrowOnly"
    AutoSize = $true
    Controls = { $menu, $grid }
}

[void] $table.RowStyles.Add((New-Style))
[void] $table.RowStyles.Add((New-Style -percent 50))
1..3 | foreach { [void] $table.RowStyles.Add((New-Style))}
1..4 | foreach { [void] $table.ColumnStyles.Add((New-Style column 17))}

$table.Controls.Add($menu)
$table.SetColumnSpan($menu, 6)

$table.Controls.Add($grid);
$table.SetColumnSpan($grid, 6)

function Add-Button($label,$action)
{
    $b = New-Control button @{text=$label; anchor = "left,right" }
    $b.add_Click($action);
    $table.Controls.Add($b);
}

Add-Button "Name"          {Update-GridData ProcessName}
Add-Button "Process Id"    {Update-GridData Id}
Add-Button "Handles"       {Update-GridData Handles}
Add-Button "WorkingSet (WS)" {Update-GridData WS}
Add-Button "CPU"           {Update-GridData cpu}

$status = New-Control label @{
    Dock="fill"
    flatstyle="popup"
    BorderStyle="fixed3d"
}
$table.Controls.Add($status);
$table.SetColumnSpan($status, 6)

```

4 Create grid control

5 Create TableLayout panel

6 Define styles

7 Add-Button helper function

```
$form.Controls.Add($table)
[void] $form.ShowDialog();
```

8 Show completed form

The process viewer example starts with the standard preamble where you load the `wpiaforms` modules. Then you define a function ❶ that will be used to update the form when a button is clicked. This function is also used to update the form when it is first displayed. You also need to define the main form ❷ and set up the menus ❸.

You create a `DataGrid` ❹ control to display the process data and a `TableLayoutPanel` ❺ to lay out all of the controls. The `New-Style` helper function ❻ in the `wpi-aforms` library sets up how the form will be resized. You want the grid to occupy most of the form space, with the buttons and menus remaining at the top and bottom.

To simplify creating the buttons at the bottom of the form, you define a helper function `Add-Button` ❼ so you don't have to repeat the code. The last thing to do is run the `Update-GridData` function to fill the grid with an initial collection of data values and then display the form ❽. And that's the end of this example.

B.4.4 Building a GUI history browser with WPF

PowerShell provides a history mechanism that lets you see what commands have been run. This is done by using the `Get-History` cmdlet or its alias, `h`. When the history gets long, having a GUI history browser can come in handy. Let's see how you can build one using WPF.

NOTE For space reasons, we'll only cover sections of the code here. You can download the complete example from the Manning website at <http://manning.com/payette2/WPSiA2SourceCode.zip>

You want the history browser tool to look like figure B.6.

In this figure, there are three tabs: one for history, one for a list of PowerShell script snippets, and one showing all the directories visited in this session. Because the

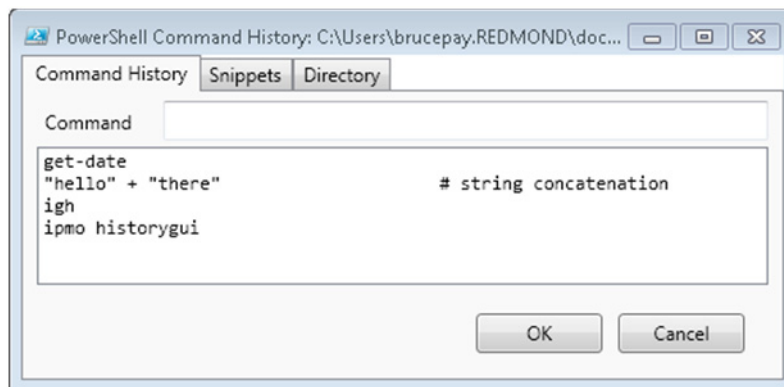


Figure B.6 The history GUI when complete. It has individual tabs for three different types of information.

basic implementation is the same for anything you want to pick from a list, you'll include more than history. Each tab will contain a list of items to choose from, a text box where you can type a pattern to filter the list, and some action buttons at the bottom.

The GUI will be implemented as a module that contains four files. These files are described in table B.1.

Table B.1 The files that make up the `historygui` module

File	Purpose
<code>historygui.psm1</code>	Contains code to set up the GUI and implement the logic for each tab
<code>Historygui.xaml</code>	Defines how the GUI should look
<code>Xamltools.psm1</code>	Contains utilities for working with XAML
<code>Snippets.ps1</code>	Contains a list of PowerShell script snippets showing various examples of how things are done

Let's start with the `xamltools` module. This code is shown in the following listing.

Listing B.12 `XamlTools` module

```
$mode = [System.Threading.Thread]::CurrentThread.ApartmentState
if ($mode -ne "STA")
{
    throw "This script requires PowerShell to be run with -sta"
}
Add-Type -AssemblyName PresentationCore, PresentationFramework
function Invoke-Xaml
{
    param([Parameter(Mandatory=$true)] $Path, [switch] $Show)
    $s = [System.IO.StreamReader] (Resolve-Path $Path).ProviderPath
    $form = [System.Windows.Markup.XamlReader]::Load($s.BaseStream)
    $s.Close()

    if ($Show)
    {
        $form.ShowDialog()
    }
    else
    {
        $form
    }
}
```

Check thread apartment state 1

Load XAMLGUI description 2

3 If -Show specified, show gui

The first thing you do in the module is check to see if you're running in STA mode **1**, which is required to use WPF. If you aren't running STA, then you throw an exception because you can't go any further.

When you know you can proceed, you load the WPF assemblies. This module defines only one function, `Invoke-Xaml`, which is used to load and display the GUI.

You use the `StreamReader` class to read the file because this is required by the `XamlReader` class. The `XamlReader` class converts the XAML text into a form object ❷. Finally, if the `-Show` option was specified ❸, you show the form; otherwise you return the form object.

Now let's walk through some portions of the XAML used to define how the history GUI looks. At the top level of the form, you use a `TabControl` to hold all the lists. For each list, you define a `TabItem` and set the header for that item appropriately:

```
<TabControl>
  <TabItem>
    <TabItem.Header>
      <TextBlock>Command History</TextBlock>
    </TabItem.Header>
```

For overall layout of the list portion of the GUI, you use a `Grid` with two columns:

```
<Grid>
<Grid.RowDefinitions>
  <RowDefinition /> <RowDefinition Height="50"/>
</Grid.RowDefinitions>
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"/> <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto"/> <RowDefinition />
  </Grid.RowDefinitions>
```

The search box is defined in the first row of the grid, with a `Label` in column 1 and a `TextBox` in column 2 for the search pattern:

```
<Label Content="Command" Margin="5, 2, 10, 0"
  FontSize="12" />
<TextBox Grid.Column="1" Height="25" Margin="5, 2, 5, 2"
  Name="Pattern" />
```

The `ListBox` control is placed in the second row of the grid and spans across both columns. Set the alignment properties so it will resize with the form, and set the `FontFamily` property to a monospaced font so the text lines up properly in the list:

```
<ListBox Grid.ColumnSpan="2" Grid.Row="1"
  Name="HistoryListBox"
  HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
  FontFamily="Consolas"
  Margin="5, 2, 5, 5" >
```

Next, add a context menu to the `ListBox` with the `ContextMenu` control. This menu will appear when the `ListBox` is right-clicked:

```
<ListBox.ContextMenu>
<ContextMenu>
  <MenuItem Header="_Bold" IsCheckable="True"/>
  <MenuItem Header="_Italic" IsCheckable="True"/>
```

```

        <Separator />
        <MenuItem Header="I_ncrease Font Size" />
        <MenuItem Header="_Decrease Font Size" />
    </ContextMenu>
</ListBox.ContextMenu>
</ListBox>

```

Use a nested `Grid` to hold the form buttons, which are aligned to the right side of the form:

```

<Grid Grid.Row="1" HorizontalAlignment="Right"
Margin="0, 10, 10, 10">
    <Grid.ColumnDefinitions>
        <ColumnDefinition /> <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Button x:Name="okButton" IsDefault="True" Grid.Column="0"
        Content="OK" Height="25" Width="80" Margin="0, 2, 10, 2" />
    <Button IsCancel="True" Grid.Column="1"
        Content="Cancel" Height="25" Width="80" Margin="0,2,10,2" />
</Grid>
</Grid>

```

And finally, close the tab item definition:

```
</TabItem>
```

Because the basic code for the other two tabs is similar to this, we won't go through it here.

The logic for the GUI is contained in the main `historygui` module. This module also sets up the GUI. The first thing you do in this module is to import the `xaml-tools` module:

```
Import-Module $PSScriptRoot/xamltools.psm1
```

Next you have to load the GUI, set up all the event actions, and define a way to launch your GUI. This is done with a function called `Invoke-GuiHistory`, which is shown in the following listing.

Listing B.13 Invoke-GuiHistory

```
function Invoke-GuiHistory
{
```

```
    param ($pattern)
```

```

    $xamlPath = Join-Path $PSScriptRoot historygui.xaml
    $script:f = Invoke-Xaml $xamlpath
    $f.Title = "PowerShell Command History: $pwd"

```

```

    setupHistory
    setupDirList
    setupSnippets

```

```
[void] $f.ShowDialog()
```

1 Load XAML definition file

2 Call tab-setup functions

```

        if ($host.Name -match "ise")
        {
            $psise.CurrentPowerShellTab.CommandPane.Text = $script:result
        }
        else
        {
            if ($script:result)
            {
                [System.Windows.Clipboard]::SetText($script:result)
            }
        }
    }
}

if ($host.Name -match "powershell ise" -and
    -not ($psise.CurrentPowerShellTab.AddOnsMenu.Submenus |
        where {$_.DisplayName -match 'History browser'}))
{
    $psISE.CurrentPowerShellTab.AddOnsMenu.Submenus.Add(
        "History Browser!", {Invoke-GuiHistory}, "Alt+h")
}

Set-Alias igh Invoke-GuiHistory
Export-ModuleMember -Function Invoke-GuiHistory -Alias igh

```

3 Define ISE menu item

4 Export aliases and functions

The first thing you do is define the `Invoke-GuiHistory` function. Next you load the XAML file and use `Invoke-Xaml` **1** to create your form object. Then, you call each of the tab-setup functions to bind the event actions for that tab **2**. With everything set up, you show the GUI. If you're running from the ISE, you insert the returned text into the ISE command pane; otherwise you save it to the clipboard. That completes the main function definition.

There are a few more things to do as part of the module setup. First, if you're running in the ISE, you define an add-on menu item and shortcut key binding **3** to make this easier to use from the ISE. To browse through history, you press Alt-H. You also define a convenience alias to make it easy to invoke from the command line. Finally, you export the function and alias from the module **4**.

As was the case with the XAML, the script logic for each tab is similar, so we'll only cover the History tab implementation. The logic for this tab is implemented as three functions: `setupHistory`, `updateHistory`, and `returnHistory`. The code for these functions is shown in the following listing.

Listing B.14 Code to set up the History tab in the GUI

```

function setupHistory
{
    $script:hlist = $f.FindName("HistoryListBox")
    $script:patternControl = $f.FindName("Pattern")
    $okButton = $f.FindName("okButton")
    $patternControl.Text = $pattern
    $patternControl.add_TextChanged({ updateHistory })
    $hlist.add_MouseDoubleClick({ returnHistory })
}

```

```

        $okButton.add_Click({ returnHistory })
        $script:result = $null
        updateHistory
    }
    function returnHistory
    {
        $script:result = ( $hlist.SelectedItem |
            foreach { $_.CommandLine } ) -join ';'
        $f.Close()
    }
    function updateHistory
    {
        $hlist.Items.Clear()
        $commands = @(Get-History -Count ([int16]::MaxValue) |
            where {$_ -match $script:patternControl.Text})
        if ($commands)
        {
            [Array]::Reverse($commands)
            $commands | % { [void] $hlist.Items.Add($_) }
        }
    }
}

```

The first function is `setupHistory`. This function locates each of the controls and places it into a variable. These variables must be defined in the module scope if the event handlers need to access them. Next you set up all the control actions. If the text in the search box is changed, you update the list box contents filtered by the updated search box text. The click handler for the button is set to return the results. Then you call `updateHistory` to load the initial contents of the list box. The `returnHistory` function returns all the selected lines in the list box, joined together with semicolons. This turns multiple lines in history into a single big command. The `updateHistory` function starts by clearing the `ListBox` and then uses `Get-History` to get all previously entered commands, filters the result using the contents of the search `TextBox`, and adds the filtered list to the `ListBox` items.

The final section in this appendix introduces some techniques for working with Active Directory.

B.5 WORKING WITH ACTIVE DIRECTORY AND ADSI

Active Directory (AD), which was introduced with Windows 2000, is the cornerstone of Windows enterprise management. It's a hierarchical database that's used to manage all kinds of enterprise data. In this section, we'll look at how you can use PowerShell to script AD.

NOTE All the examples shown in this section use ADAM—Active Directory Application Mode—which is a free download from Microsoft.com. ADAM is a standalone AD implementation that doesn't require Windows Server to run; it can be installed on a computer running Windows XP. It's a great tool for learning about AD. On

Windows 7, ADAM has been replaced with Active Directory Lightweight Directory Services (AD LDS).

As with WMI, the keys to PowerShell's AD support are the Active Directory Service Interface (ADSI) object adapter and the `[ADSI]` type shortcut. For the purpose of these examples, you'll use an AD installation for a fictitious company called PoshCorp.com.

B.5.1 Accessing the AD service

Here's how you can access the PoshCorp AD service. You take the Lightweight Directory Access Protocol (LDAP) URL for the service and cast it into an ADSI object:

```
PS (1) > $domain = [ADSI] `
>> "LDAP://localhost:389/dc=NA,dc=poshcorp,dc=com"
>>
```

Now that you've connected to the AD service, you want to create a new organizational unit (OU) for the human resources (HR) department. You can use the `Create()` method on the object in `$domain` to do this:

```
PS (2) > $newOU = $domain.Create("OrganizationalUnit", "ou=HR")
PS (3) > $newOU.SetInfo()
```

When you've created the object, you need to call `SetInfo()` to cause the server to be updated.

B.5.2 Adding a user

To retrieve the object that represents the OU you created, again you use an `[ADSI]` cast, but this time you include the element `ou=HR` in the URL:

```
PS (5) > $ou = [ADSI] `
>> "LDAP://localhost:389/ou=HR,dc=NA,dc=poshcorp,dc=com"
>>
```

Now you need to create a user object in this department. Use the `Create()` method on the object in `$ou` to create a new user object that has the common name (CN) Dogbert:

```
PS (6) > $newUser = $ou.Create("user", "cn=Dogbert")
```

You also want to put some properties on this user; use the `Put()` method on the user object to do this. (The set of properties you can set is defined by the AD schema for the user object.)

```
PS (7) > $newUser.Put("title", "HR Consultant")
PS (8) > $newUser.Put("employeeID", 1)
PS (9) > $newUser.Put("description", "Dog")
PS (10) > $newUser.SetInfo()
```

You set the `title`, `employeeID`, and `description` properties for this user and then call `SetInfo()` to update the server when you're done.

As you might expect, to retrieve this user object, you use a URL with the path element `cn=Dogbert` added to it:

```
PS (12) > $user = [ADSI] ("LDAP://localhost:389/" +  
>> "cn=Dogbert,ou=HR,dc=NA,dc=poshcorp,dc=com")  
>>
```

You should verify that the properties have been set, so let's display them:

```
PS (13) > $user.title  
HR Consultant  
PS (14) > $user.Description  
Dog
```

B.5.3 Adding a group of users

Next, let's see how to create a bunch of users all at once. You'll define a set of data objects where the object contains a `name` property that will be used to name the employee and additional properties to set for the user. In this example, you define this data as an array of hashtables:

```
PS (15) > $data =  
>> @{  
>> Name="Catbert"  
>> Title="HR Boss"  
>> EmployeeID=2  
>> Description = "Cat"  
>> },  
>> @{  
>> Name="Birdbert"  
>> Title="HR Flunky 1"  
>> EmployeeID=3  
>> Description = "Bird"  
>> },  
>> @{  
>> Name="Mousebert"  
>> Title="HR Flunky 2"  
>> EmployeeID=4  
>> Description = "Mouse"  
>> },  
>> @{  
>> Name="Fishbert"  
>> Title="HR Flunky 3"  
>> EmployeeID=5  
>> Description = "Fish"  
>> }  
>>
```

Now let's write a function to process this data and add these users to AD. Call this function `New-Employee`. It takes two arguments: the list of employee objects to create and, optionally, the OU to create them in. This defaults to the OU you created:

```
PS (16) > function New-Employee  
>> {
```

```

>> param (
>>     [Parameter(Mandatory=$true)]
>>     $employees,
>>     [ADSI] $ou =
>>         'LDAP://localhost:389/OU=HR, dc=NA, dc=poshcorp, dc=com '
>> )
>>
>> foreach ($record in $employees)
>> {
>>     $newUser = $ou.Create("user", "cn=$( $record.Name) ")
>>     $newUser.Put("title", $record.Title)
>>     $newUser.Put("employeeID", $record.employeeID)
>>     $newUser.Put("description", $record.Description)
>>     $newUser.SetInfo()
>> }
>> }
>>

```

This function iterates over the list of employees, creates each one, sets the properties, and writes the object back to the server.

This function doesn't care what type of objects are in `$employees` (or even if it's a collection). The only thing that matters is that the objects have the correct set of properties. This means that instead of using a hashtable, you could use an XML object or the result of using the `Import-Csv` cmdlet.

NOTE Using `Import-Csv` is particularly interesting because it means you can use a spreadsheet application to enter the data for your users, export the spreadsheet to a CSV file, and run a simple command like `New-Employee (Import-Csv usersToCreate.csv)` to import all the users from that spreadsheet into AD.

You also need to write another function `Get-Employee` that retrieves employees from an `OU`. This function allows wildcards to be used when matching the employee name. It's also optional, and all employees are returned by default. Again, default the `OU` to be `ou=HR`:

```

PS (17) > function Get-Employee (
>>     [string] $name='*',
>>     [adsisearchresult] $ou =
>>         "LDAP://localhost:389/ou=HR, dc=NA, dc=poshcorp, dc=com"
>> )
>> {
>>     [void] $ou.PSBase
>>     $ou.PSBase.Children | where { $_.name -like $name }
>> }
>>

```

Let's try these functions. First use `New-Employee` to populate the `OU` with user objects:

```

PS (18) > New-Employee $data

```

Next, use `Get-Employee` to retrieve the users. Display the `name`, `title`, and `homePhone` properties for each user:

```
PS (19) > Get-Employee | Format-Table name,title,homePhone
```

name	title	homePhone
{Birdbert}	{HR Flunky 1}	{}
{Catbert}	{HR Boss}	{}
{Dogbert}	{HR Consultant}	{}
{Fishbert}	{HR Flunky 3}	{}
{Mousebert}	{HR Flunky 2}	{}

This shows all the users and their titles. Because you didn't set the home phone number property when you created the user entries, that field appears as empty.

This raises a question—how can you update the user properties after you've created the users?

B.5.4 Updating user properties

You'll create another function to do this called `Set-EmployeeProperty`. This function will take a list of employees and a hashtable containing a set of properties to apply to each employee. As always, you'll default the OU to be HR:

```
PS (20) > function Set-EmployeeProperty (  
>>     $employees =  
>>     $(throw "You must specify at least one employee"),  
>>     [hashtable] $properties =  
>>     $(throw "You must specify some properties"),  
>>     [ADSI] $ou =  
>>     "LDAP://localhost:389/ou=HR, dc=NA,dc=poshcorp,dc=com "  
>> )  
>> {  
>>     foreach ($employee in $employees)  
>>     {  
>>         if ($employee -isnot [ADSI])  
>>         {  
>>             $employee = Get-Employee $employee $ou  
>>         }  
>>         foreach ($property in $properties.Keys)  
>>         {  
>>             $employee.Put($property, $properties[$property])  
>>         }  
>>         $employee.SetInfo()  
>>     }  
>> }  
>>
```

Unlike the `New-Employee` function, this time you're requiring the properties object to be a hashtable because you're going to use the `Keys` property to get the list of properties to set the user object. (This is similar to the `Form` function you saw back in chapter 11.) You're also using the `Get-Employee` function to retrieve the user objects to set.

Now let's use this function to set the `title` and `homePhone` properties on two of the users in this OU:

```
PS (21) > Set-EmployeeProperty dogbert,fishbert @{
>>     title="Supreme Commander"
>>     homePhone = "5551212"
>> }
>>
```

And verify the changes using the `Get-Employee` function:

```
PS (22) > Get-Employee | ft name,title,homePhone
```

name	title	homePhone
----	-----	-----
{Birdbert}	{HR Flunky 1}	{}
{Catbert}	{HR Boss}	{}
{Dogbert}	{Supreme Commander}	{5551212}
{Fishbert}	{Supreme Commander}	{5551212}
{Mousebert}	{HR Flunky 2}	{}

You can see that the titles for the specified objects have been updated and the phone numbers for those users are now set.

B.5.5 Removing users

The last thing to do is figure out how to remove a user. Again, you'll write a function to do this called `Remove-Employee`:

```
PS (23) > function Remove-Employee (
>>     $employees =
>>     $(throw "You must specify at least one employee"),
>>     [ADSI] $ou =
>>         "LDAP://localhost:389/ou=HR, dc=NA,dc=poshcorp,dc=com"
>> )
>> {
>>     foreach ($employee in $employees)
>>     {
>>         if ($employee -isnot [ADSI])
>>         {
>>             $employee = Get-Employee $employee $ou
>>         }
>>     }
>>     [void] $employee.psbasedelete()
>>     $employee.psbasedelete()
>> }
>> }
```

Use it to remove a couple of users:

```
PS (24) > Remove-Employee fishbert,mousebert
```

And verify that they have been removed:

```
PS (25) > Get-Employee

distinguishedName
-----
{CN=Birdbert,OU=HR,DC=NA,DC=poshcorp,DC=com}
{CN=Catbert,OU=HR,DC=NA,DC=poshcorp,DC=com}
{CN=Dogbert,OU=HR,DC=NA,DC=poshcorp,DC=com}
```

As you can see, with little effort, it's possible to significantly automate tasks involving AD by using PowerShell.

RSAT and the AD module

In this section, we looked at how to work with AD using the [ADSI] type. This was the only option in PowerShell V1. In PowerShell V2, this is still the default mechanism used on a client machine. But on Server 2008 R2, if the AD role is installed, an AD PowerShell module can also be installed as an optional feature. To add the same module to a Windows 7 client, download and install the Remote Server Administration Tools (RSAT) from Microsoft. (Go to <http://microsoft.com/downloads> and search for *RSAT*.)

B.6 SUMMARY

This appendix presented examples showing how to use PowerShell to build solutions in a variety of problem domains. We started with basic shell/scripting solutions and then looked at WMI (chapter 19) and COM (chapter 18) applications. You built some more GUIs using both Windows Forms and WPF. Finally, we looked at how to access AD through the [ADSI] type accelerator. Many more examples are available through Microsoft sites like <http://technet.microsoft.com> as well as third-party script repositories like <http://powershell.com> and <http://poshcode.org>.



A P P E N D I X C

PowerShell quick reference

C.1 Getting started 73	C.7 Error handling 108
C.2 PowerShell basics 76	C.8 PowerShell debugger 110
C.3 The PowerShell language 77	C.9 Performing tasks with PowerShell 111
C.4 Defining functions and scripts 98	C.10 Windows Management Instrumentation (WMI) 117
C.5 Modules 105	C.11 PowerShell eventing 119
C.6 Remoting 106	

C.1 GETTING STARTED

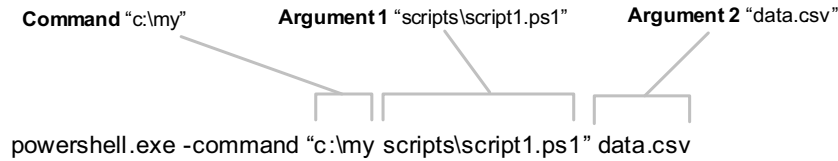
Windows PowerShell is a modern object-based command shell and scripting language designed for Microsoft Windows. Along with the usual shell features for working with files and programs, it also provides direct access to Windows through the various object models such as .NET, COM, and WMI.

This appendix gathers many useful tables and diagrams from *Windows PowerShell in Action 2nd Edition* to provide a reference guide to the contents of the book and PowerShell at large. It summarizes all language elements and operations, includes many handy tables, and provides concise descriptions of how things work.

C.1.1 Running powershell.exe

PowerShell is included on all Windows systems from Windows 7/Server 2008 R2 and later. For earlier systems, it's freely available through the Microsoft Windows Update service, packaged as an optional update. Once installed, use the Start menu to start the shell or run `powershell.exe`. The following figures show this command and its options.

There are two major parameter sets: the `-command` set and the `-file` set. This figure shows how the command line is processed when `-command` is specified.

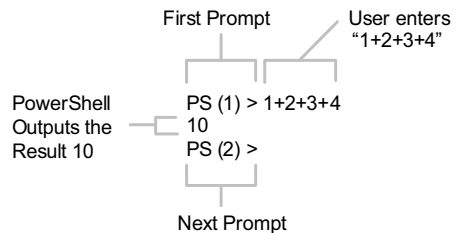


The `-command` parameter's first argument is parsed into two tokens. With `-file`, the entire first argument is treated as the name of a script to run.

A second way to use PowerShell v2 or later is through the PowerShell Integrated Scripting Environment (ISE). You start the ISE by running `powershell_ise.exe`.

C.1.2 Basic interactive use in a console window

This diagram shows the basic interaction pattern in a PowerShell session. The interpreter issues a prompt; the user enters some text to be evaluated then presses Enter. PowerShell interprets the text and displays the result followed by a new prompt.



The following list covers the basics you need to know for interactive use of the environment:

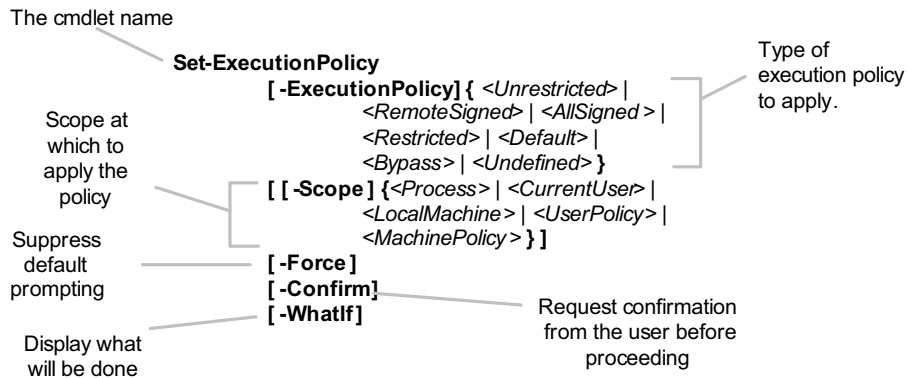
- Use the `exit` keyword to exit the shell.
- `Ctrl-C` will interrupt the current command and return you to the prompt.
- PowerShell is case-insensitive by default, so commands can be typed in whatever case you desire.
- A command can be spread over multiple lines. The interpreter will continue to prompt for additional input until a complete command is received or the user presses `Ctrl-C`.
- The line-continuation character is the backquote ``` (also called the backtick).
- To get help about a command, execute `Get-Help CommandName`. Running the `Get-Help` command by itself gives you a list of topics.
- The online documentation is usually much more up-to-date than the installed help files. Running `Get-Help -OnLine commandName` will cause the on-line help content to be displayed using the default web browser.
- The `Help` command supports wildcards, so `Get-Help Get-*` returns all the commands that start with `Get-`.
- You can also get basic help on a command by typing `commandName -?`. For example, `Get-Date -?` shows help for the `Get-Date` command.

- As well as help on commands, there's a collection of general help topics prefixed with `about_`. You can get a list of these topics by running `Get-Help about_*`.
- Command-line editing in the PowerShell console works just as it does in `cmd.exe`: use the arrow keys to go up and down, the Insert and Delete keys to insert and delete characters, and so on.

C.1.3 Execution policy

Before you can execute scripts, you need to make sure your *execution policy* allows this. The current policy in effect can be retrieved using the `Get-ExecutionPolicy` cmdlet and set with `Set-ExecutionPolicy`. (Execution policy doesn't affect interactive use.)

The following figure shows the cmdlet used to set the PowerShell execution policy.



Possible execution policy settings

Here are the available policy settings.

Policy	Description
Restricted	Script execution is disabled.
AllSigned	Scripts can be executed, but they must be Authenticode-signed before they will run.
RemoteSigned	All scripts that are downloaded from a remote location must be Authenticode-signed before they can be executed. This is the minimum recommended execution policy setting.
Unrestricted	PowerShell will run any script. However, it will still prompt the user when it encounters a script that has been downloaded.
Bypass (v2 only)	Nothing is blocked, and there are no warnings or prompts.
Undefined (v2 only)	The currently assigned execution policy is removed from the current scope. This parameter won't remove an execution policy that's set in a Group Policy scope (User scope or Machine scope).

The following table lists the execution policy scopes and their meanings. Policies that are only available in PowerShell v2 are indicated in the Version column.

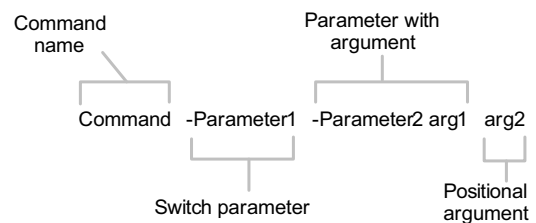
Setting	Description	Version
MachinePolicy	Group Policy setting for the machine.	v1 and v2
UserPolicy	Group Policy setting for the user.	v1 and v2
LocalMachine	Applies to all users on the system.	v1 and v2
CurrentUser	Applies to the current user. Other users on this machine aren't affected.	v1 and v2
Process	Applies to the current process and is discarded when the process exits. In this case, the execution policy is saved in the <code>PSExecutionPolicyPreference</code> environment variable (<code>\$env:PSExecutionPolicyPreference</code>), instead of in the Registry.	v2 only

C.2 POWERSHELL BASICS

This section presents a summary of the PowerShell language elements, including operators and statements.

C.2.1 Command syntax

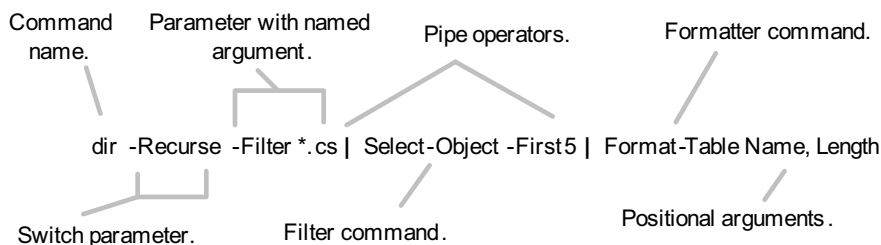
This figure shows the anatomy of a basic command. It begins with the name of the command, followed by parameters. These may be switch parameters that take no arguments, regular parameters that do take arguments, or positional parameters where the matching parameter is inferred by the argument's position on the command line.



Command names follow a *Verb-Noun* naming pattern, such as `Copy-Item` or `Stop-Process`. This pattern is enforced for compiled cmdlets and for commands exported from PowerShell modules. The words in the command name are *Pascal-cased* with leading capitals on all words: for example, `Set-ExecutionPolicy`.

C.2.2 Pipeline operation

Pipeline execution is a central concept in PowerShell. PowerShell pipe objects live between each stage rather than text or even XML. The objects are passed one at a time through all stages of the pipeline. This allows for streaming behavior where objects are returned to the user as soon as possible. The basic elements of a pipeline are shown in the next figure.



In this pipeline, the `dir` command emits objects that are passed to the `Select-Object` command. `Select-Object` passes the first five objects it receives on to the next stage of the pipeline where they're formatted as a table with two columns and displayed to the user.

C.3 THE POWERSHELL LANGUAGE

This section provides a concise but fairly complete summary of the PowerShell language.

NOTE A complete formal specification for the PowerShell v2 language is available through the Microsoft Open Specifications program under the Community Promise license.

C.3.1 PowerShell literals

Every language must have a way to create inline data through *literals*. PowerShell supports the following literal data types.

Numbers

All the .NET numeric types are supported: integers, longs, floating-point, and decimals. Hexadecimal notation can be used by prefixing the number with `0x`. Suffixes are used to specify a particular numeric type: `l` for long integers and `d` for decimal values. Quantifier suffixes can also be used: `kb` (kilobyte), `mb` (megabyte), `gb` (gigabyte), `tb` (terabyte), and `pb` (petabyte). Quantifier suffixes are applied after the type suffix; for example, `1.2dgb` produces a decimal number representing 1.2 gigabyte. Letters in a numeric value aren't case-sensitive.

Type	Examples
Integer	1, 2, 0xF3, 1.5mb
Long Integer	33l, 0xf3beL, 0x1elgb
Floating point	1.2, 1.3e5
Decimal	1.34d, 6d, 6dkb

String literals

PowerShell uses .NET strings. Single- and double-quoted strings are supported where variable substitution and escape sequence processing is done in double-quoted strings but not in single-quoted ones, as shown here:

```
PS (1) > $x="Hi"
PS (2) > "$x bob`nHow are you?"
Hi bob
How are you?
PS (3) > '$x bob`nHow are you?'
$x bob`nHow are you?
```

The escape character is a backtick (`) instead of a backslash so file paths can be written with either a forward slash or a backslash. PowerShell strings can span multiple lines. There is a second type of string literal called a *here-string* that's typically used to embed large blocks of text into a script. As with regular strings, both single and double quotes are supported. For example:

```
PS (4) > $s = @"
    In a here string you can have embedded quotes
    Like "the value of x is $x"
"@
```

The closing "@" or '@' sequence must start at the beginning of the line. The following table presents a number of examples showing string literals.

Example	Description
<code>\$a = 2; "a is \$a"</code>	Variable expansion in double-quoted strings.
<code>"The date is \$(Get-Date)"</code>	Subexpression evaluation in double-quoted strings.
<code>"a`t tab and a `n"</code>	Escape sequence for tab and newline. The escape character is a backtick (`).
<code>'a is \$a'</code>	No expansion in single-quoted strings.
<code>"hello" + "there"</code>	String concatenation.
<code>"hello" -like "he*"</code>	String match with a wildcard pattern.
<code>"hello" -match "el"</code>	String match with regular expressions.
<code>"hello" -replace 'l.*\$', "lp"</code>	String replacement using regular expressions.
<code>"abc" * 2 -eq "abcabc"</code>	String multiplication, which repeats the string.
<code>"a, b, c" -split ', *'</code>	Splitting a string using regular expressions.
<code>-split "a b c"</code>	Unary split on whitespace.
<code>"abcdefg"[0]</code>	String indexing, with origin 0.
<code>"abcdefg".Substring(3)</code>	Substring from the given position to the end of the string.
<code>"abcdefg".Substring(2,3)</code>	Substring from the given position, consisting of a given number of characters.

Arrays

Arrays are constructed using the comma (,) operator. Unless otherwise specified, arrays are of type `System.Object[]`. Indexing is done with square brackets starting at 0. Negative indexing is also allowed with -1 being the last element in the array. The `+` operator concatenates two arrays to produce a new array, copying the elements from the original arrays:

```
PS (1) > $a = 1, 2, 3
PS (2) > $a[1]
2
PS (3) > $a.length
3
PS (4) > [string] ($a + 4, 5)
1 2 3 4 5
```

Because PowerShell pipelines are streams, sometimes you don't know if a command returns an array or a scalar. This problem is addressed using a special notation:

```
@( <expr> )
```

The result of this expression will always be an array, regardless of what *expr* returns. If the result is already an array, it's returned as is. If it isn't an array, a new single-element array is constructed to hold this value.

Arrays of sequential numbers can be created using the range operator (`..`). For example

```
1..10
```

returns an array containing the numbers from 1 to 10. Arrays can also be indexed using a range of numbers to select a subsequence from the array.

Hash tables

The PowerShell `hashtable` literal produces an instance of the .NET type `System.Collections.Hashtable`. Hashtable keys may be written as unquoted strings or full expressions; individual key/value pairs are separated either by newlines or by semicolons:

```
PS (1) > $h = @{a=1; b=2+2
>> ("the" + "date") = Get-Date}
>>
PS (2) > $h
```

Name	Value
----	-----
thedata	5/24/2011 10:29:26 PM
a	1
b	4

```
PS (3) > $h["thedata"]
Tuesday, May 24, 2011 10:29:26 PM
```

```
PS (4) > $h.thedate
Tuesday, May 24, 2011 10:29:26 PM
```

You can use the `+` operator to merge the key/value pairs from two hash tables to produce a new hash table. If there are any duplicate keys in the two source objects, an error will be returned.

Type literals

Type literals are written using the notation `[typename]`:

```
[int]
[string]
[System.Collections.Generic.List[string]]
```

Type literals are case-insensitive so `[INT]`, `[int]` and `[Int]` are equivalent. The full name of the type must be specified unless the type has an accelerator (shortcut) for the name, such as `[xml]` for `[System.Xml.XmlDocument]`. If the type is in the `System` namespace, the prefix `System.` may be omitted, as in `[Diagnostics.Process]` instead of `[System.Diagnostics.Process]`. The type parameters for generic types are enclosed in nested square brackets: `[System.Collections.Generic.Dictionary[string,int]]`.

C.3.2 Variables

In PowerShell, variables are organized into namespaces. All variable references are identified by prefixing their names with `$` as in `$x = 3`. Variable names can be unqualified like `$a` or namespace qualified like `$variable:a` or `$env:path`. In the latter case, `$env:path` is the environment variable `Path` allowing access to the process environment strings. PowerShell namespaces are also used for accessing functions through the `function` namespace (for example, `$function:prompt`) and command aliases through the `alias` namespace (`$alias:dir`). Variables typically use only letters and numbers, but any character can be used if the reference is enclosed in braces:

```
PS (1) > ${2 + 2} = 4
PS (2) > ${2 + 2}
4
PS (3) >
```

Variables are dynamically scoped, but assignments always happen in the current scope unless a scope qualifier is used. The scope qualifiers are `global`, `script`, `local`, and `private`:

```
$global:aGlobalVariable = 123
$script:foo = "Hi there"
```

By default, variables can be assigned any type of value. But they can be type constrained by prefixing an assignment state with a type literal:

```
PS (3) > [int] $x = 1
PS (4) > $x = "abc"
Cannot convert value "abc" to type "System.Int32".
```

The variable namespaces are also manifested as drives, allowing you to manage things like functions and variables as you would a set of files. For example, variables can be removed from the environment by using the `Remove-Item` cmdlet (or its alias `del`) in the `variable:` drive:

```
Remove-Item variable:x
```

This removes the variable `x`.

Automatic variables

PowerShell defines a number of automatic variables. These variables are described in the following table.

Variable	Description
<code>\$^</code> , <code>\$\$</code>	First and last tokens (words) in the last interactive command string.
<code>\$?</code>	False if an error was generated by the last command.
<code>\$_</code>	In pipelines, the current pipeline object. In switch statement clauses, the matched object.
<code>\$args</code>	Arguments to a function or script for which there is no corresponding parameter.
<code>\$ConsoleFileName</code>	Name of the console file used to start the session. Rarely used.
<code>\$Error</code>	Circular buffer of all errors generated by the PowerShell engine. The most recent error is available in <code>\$error[0]</code> .
<code>\$Event</code>	In the action block of an event subscription, the event being processed. (See chapter 20.)
<code>\$EventSubscriber</code>	In the action block of an event subscription, information about the event subscription.
<code>\$ExecutionContext</code>	Provides access to PowerShell engine services outside of an advanced function.
<code>\$False</code> , <code>\$True</code>	Constants for the Boolean true and false values.
<code>\$foreach</code>	Enumerator in a <code>foreach</code> loop.
<code>\$Home</code>	User's home directory. Equivalent to <code>~</code> in a path.
<code>\$Host</code>	Provides access to host services such as prompting, writing to the console, and so on.
<code>\$input</code>	Enumerator containing the pipeline input.
<code>\$LastExitCode</code>	Numeric value specified when the most recently executed script or application exited.
<code>\$Matches</code>	Matches from the most recent use of the <code>-match</code> operator.
<code>\$MyInvocation</code>	Information about where a script or function was defined and where it was called from.
<code>\$NestedPromptLevel</code>	Number of levels of nested prompts that currently exist.
<code>\$Null</code>	Constant holding the null value.

(continued)

Variable	Description
\$PID	Current process id.
\$PROFILE	Path to the user's startup profile file.
\$PSBoundParameters	Hash table containing the parameters that were bound when a command is called.
\$PSCmdlet	In advanced functions, provides access to engine features and other information about the current execution.
\$PSCulture, \$PSUICulture	Variables that contain the current culture information.
\$PSDebugContext	When debugging, contains information about the debugging environment such as breakpoints.
\$PSHOME	Location of the PowerShell installation directory.
\$PSScriptRoot	In a script module, the full path to the module directory for the currently executing module.
\$PSVersionTable	Table of information about the current PowerShell process.
\$PWD	Current working directory.
\$Sender	In an event subscription action block, the object that generated the event.
\$ShellId	"Shell id" string for the current shell instance. Rarely used.
\$SourceArgs, \$SourceEventArgs	In an event subscription action block, the event arguments.
\$this	In a script property or method, the object to which the member is bound.

C.3.3 Operators

PowerShell has a large set of operators for numeric operations, array operations, and string- and pattern-matching operations.

Arithmetic operators

Arithmetic operators are polymorphic and work with a variety of data types. For example, the addition operator can add numbers, concatenate strings, and append arrays. The following table includes examples of these operations.

Operator	Description	Example	Result
+	Adds two values together	2+4	6
		"Hi " + "there"	Hi there
		1, 2, 3 + 4, 5, 6	1,2,3,4,5,6
*	Multiplies 2 values	2 * 4	8
		"a" * 3	aaa
		1, 2 * 2	1,2,1,2

(continued)

Operator	Description	Example	Result
-	Subtracts one value from another	6-2	4
/	Divides two values	6 / 2 7 / 4	3 1.75
%	Returns the remainder from a division operation	7%4	3

Assignment operators

PowerShell supports all the assignment operators typically found in C-like languages. Multiple assignment is supported. For example, if you execute the following expression

```
$a,$b,$c = 1,2,3,4,5,6
```

`$a` has the value 1, `$b` has the value 2, and `$c` has the remaining values. The assignment operators are listed in the following table.

Operator	Example	Equivalent	Description
=	<code>\$a = 3</code>		Sets the variable to the specified value
+=	<code>\$a += 2</code>	<code>\$a = \$a + 2</code>	Performs the addition operation in the existing value and then assigns the result back to the variable
-=	<code>\$a -= 13</code>	<code>\$a = \$a - 13</code>	Performs the subtraction operation in the existing value and then assigns the result back to the variable
*=	<code>\$a *= 3</code>	<code>\$a = \$a * 3</code>	Multiplies the value of a variable by the specified value or appends to the existing value
/=	<code>\$a /= 3</code>	<code>\$a = \$a / 3</code>	Divides the value of a variable by the specified value
%=	<code>\$a %= 3</code>	<code>\$a = \$a % 3</code>	Divides the value of a variable by the specified value and assigns the remainder (modulus) to the variable

Comparison Operators

Due to the need to support redirection operations, the PowerShell comparison operators are patterned after the comparison operators in the UNIX Korn shell: a dash followed by a two- or three-character sequence (for example, `-eq` for equals). The operators can be applied to any type of object. When an array is used on the left side of a comparison operator, the matching elements in the array are returned. For string comparisons, operators are case-insensitive by default; but each operator has a second form where the operator is either explicitly case-insensitive, indicated by an `i` before the operation, or explicitly case-sensitive, indicated by a `c` before the operation.

Operator	Description	Example	Result
-eq -ceq -ieq	Equals	5 -eq 5	\$true
-ne -cne -ine	Not equals	5 -ne 5	\$false
-gt -cgt -igt	Greater than	5 -gt 3	\$true
-ge -cge -ige	Greater than or equal	5 -ge 3	\$true
-lt -clt -ilt	Less than	5 -lt 3	\$false
-le -cle -ile	Less than or equal	5 -le 3	\$false

Collection containment operators

The collection containment operators return true if an element is contained in a collection.

Operator	Description	Example	Result
-contains -ccontains -icontains	The collection on the left side contains the value specified on the right side.	1,2,3 -contains 2	\$true
-notcontains -cnotcontains -inotcontains	The collection on the left side doesn't contain the value on the right side.	1,2,3 -notcontains 2	\$false

Wildcard operators

PowerShell wildcard pattern-matching operators perform a simple form of matching similar to command-line wildcards (or *globbing*) in other shells. These operators are as follows:

Operator	Description	Example	Result
-like -clike -ilike	Does a wildcard pattern match	"one" -like "o*"	\$true
-notlike -cnotlike -inotlike	Does a wildcard pattern match; true if the pattern doesn't match	"one" -notlike "o*"	\$false

Wildcard pattern metacharacters

The following special characters can be used in wildcard patterns.

Wildcard	Description	Example	Matches	Doesn't match
*	Matches zero or more characters anywhere in the string	a*	a aa abc ab	bc babc

(continued)

Wildcard	Description	Example	Matches	Doesn't match
?	Matches any single character	a?c	abc aXc	a ab
[<char>--<char>]	Matches a sequential range of characters	a[b-d]c	abc acc adc	aac aec afc abbc
[<char><char>...]	Matches any one character from a set of characters	a[bc]c	abc acc	a ab Ac adc

Regular expression operators

Pattern-matching operations and string substitutions using regular expressions are performed with the `-match` and `-replace` operators. PowerShell uses the .NET regular expression classes to implement these operators.

Operator	Description	Example	Result
<code>-match</code> <code>-cmatch</code> <code>-imatch</code>	Does a pattern match using regular expressions	"Hello" <code>-match</code> "[jkl]"	<code>\$true</code>
<code>-notmatch</code> <code>-cnotmatch</code> <code>-inotmatch</code>	Does a regex pattern match; returns true if the pattern doesn't match	"Hello" <code>-notmatch</code> "[jkl]"	<code>\$false</code>
<code>-replace</code> <code>-creplace</code> <code>-ireplace</code>	Does a regular expression substitution on the string on the left side and returns the modified string	"Hello" <code>-replace</code> "ello","i" "Hi"	
	Deletes the portion of the string matching the regular expression	"abcde" <code>-replace</code> "bcd"	"ae"

Regular expression metacharacters

Patterns in regular expressions are built using *metacharacters* that have special meaning to the pattern matcher. A subset of the metacharacters supported by .NET regular expressions is listed in the following table.

Pattern	Description
.	Matches any character except <code>`n</code> (newline).
[aeiou]	Matches any single character included in the set between the square brackets.
[^aeiou]	Matches any single character not in the specified set of characters.
[0-9a-zA-F]	Hyphen specifies contiguous character ranges.

(continued)

Pattern	Description
\w	Matches any word character where a word is equivalent to the Unicode character categories <code>[\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nd}\p{Pc}]</code> . (Refer to the regular expression documentation on MSDN for more information about character categories.)
\W	Matches any non-word character.
\s	Matches any whitespace character.
\S	Matches any non-whitespace character.
\d	Matches any decimal digit.
\D	Matches any non-digit.
*	Specifies zero or more matches; for example, <code>\w*</code> or <code>(abc)*</code> . Equivalent to <code>{0,}</code> .
+	Specifies one or more matches; for example, <code>\w+</code> or <code>(abc)+</code> . Equivalent to <code>{1,}</code> .
?	Specifies zero or one matches; for example, <code>\w?</code> or <code>(abc)?</code> . Equivalent to <code>{0,1}</code> .
{n}	Specifies exactly <i>n</i> matches; for example, <code>(pizza){2}</code> .
{n,}	Specifies at least <i>n</i> matches; for example, <code>(abc){2,}</code> .
{n,m}	Specifies at least <i>n</i> , but no more than <i>m</i> , matches.
()	Captures the substring matched by the pattern enclosed in the parentheses. Captures using <code>()</code> are numbered automatically based on the order of the opening parenthesis, starting at one. The first capture, capture element number zero, is the text matched by the whole regular expression pattern.
(?<name>)	Captures the matched substring into a group name or number name. The string used for name must not contain any punctuation, and it can't begin with a number. You can use single quotes instead of angle brackets; for example, <code>(?'name')</code> .
^	Specifies that the match must occur at the beginning of the string or the beginning of the line.
\$	Specifies that the match must occur at the end of the string, before <code>\n</code> at the end of the string, or at the end of the line.
\A	Specifies that the match must occur at the beginning of the string (ignores the Multiline option).
\Z	Specifies that the match must occur at the end of the string or before <code>\n</code> at the end of the string (ignores the Multiline option).
\z	Specifies that the match must occur at the end of the string (ignores the Multiline option).
\G	Specifies that the match must occur at the point where the previous match ended. When used with <code>Match.NextMatch()</code> , this ensures that matches are all contiguous. (Refer to the regular expression documentation on MSDN for more information.)

(continued)

Pattern	Description
<code>\b</code>	Specifies that the match must occur on a boundary between <code>\w</code> (alphanumeric) and <code>\W</code> (non-alphanumeric) characters. The match must occur on word boundaries—that is, at the first or last character in words separated by any non-alphanumeric characters.
<code>\number</code>	Backreference. For example, <code>(\w)\1</code> finds doubled word characters.
<code>\k<name></code>	Named backreference. For example, <code>(?<char>\w)\k<char></code> finds doubled word characters. The expression <code>(?<43>\w)\43</code> does the same. You can use single quotes instead of angle brackets; for example, <code>\k'char'</code> .

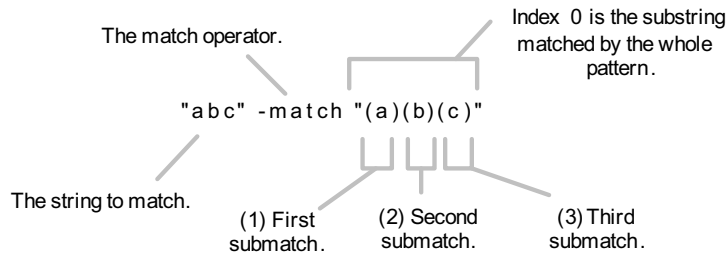
Escape sequences in regular expressions

PowerShell uses the backtick (```) as its quote character. Regular expressions use a separate quoting mechanism where the quote character is the backslash (`\`). The supported escape sequences are listed in the following table.

Escaped character	Description	Matched code
<code>\a</code>	Matches a bell (alarm) character.	<code>([char] 0x7.)</code>
<code>\b</code>	Matches a backspace.	<code>([char] 0x08)</code>
<code>\t</code>	Matches the tab character.	<code>([char] 0x09)</code>
<code>\r</code>	Matches the carriage return.	<code>([char] 0x0d)</code>
<code>\v</code>	Matches the vertical tab.	<code>([char] 0x0b)</code>
<code>\f</code>	Matches a form feed.	<code>([char] 0x0c)</code>
<code>\n</code>	Matches a new line.	<code>([char] 0x0a)</code>
<code>\e</code>	Matches an escape.	<code>([char] 0x1b)</code>
<code>\040</code>	Matches an ASCII character as octal (up to three digits). Numbers with no leading zero are backreferences if they have only one digit or if they correspond to a capturing group number.	
<code>\x20</code>	Matches an ASCII character using hexadecimal representation (exactly two digits).	
<code>\cC</code>	Matches an ASCII control character. For example, <code>\cC</code> is Ctrl-C.	
<code>\u0020</code>	Matches a Unicode character using hexadecimal representation (exactly four digits).	
<code>\</code>	When followed by a character that isn't recognized as an escaped character, matches that character.	

The -match operator

This diagram shows the syntax of a regular expression `-match` operator where the pattern contains submatches. Each of the bracketed elements of the pattern corresponds to a submatch pattern.



The -replace operator

This operator is used for updating or replacing the contents of a string or strings. For non-enumerable types, the `-replace` operator converts the left operand to a string before doing the replace. If the left operand is a collection, it will iterate over the collection, converting each element to a string then performing the replace operation.



Special characters in the replacement string

This table lists the special character sequences that can be used in the replacement string and describes what they do.

Character sequence	Description
<code>\$number</code>	Substitutes the last submatch matched by group number
<code>\${name}</code>	Substitutes the last submatch matched by a named capture of the form <code>(?<name>)</code>
<code>\$\$</code>	Substitutes a single <code>\$</code> literal
<code>\$\$</code>	Substitutes a copy of the entire match
<code>\$`</code>	Substitutes all the text from the argument string before the matching portion
<code>\$'</code>	Substitutes all the text of the argument string after the matching portion
<code>\$+</code>	Substitutes the last submatch captured
<code>\$_</code>	Substitutes the entire argument string

The -join operator

The unary `-join` operator allows you to join a collection of objects into a single string with nothing between the elements.

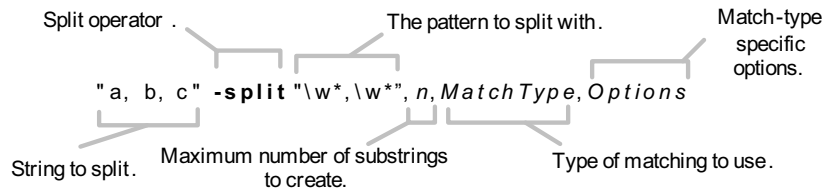
The join operator. 

The binary form of the join operator lets you join a collection of objects into a single string using the specified join string.

Join operator 

The -split operator

The `-split` operator allows you to split a string into a collection of smaller strings. It supports a variety of arguments and options that control how the target string is split.

Split operator `.` 

The following table lists the options for the binary form of the `-split` operator.

Option	Description	Applies to
IgnoreCase	Allows you to override default case-sensitive behavior when using the <code>-csplit</code> variant of the operator.	RegexMatch, SimpleMatch
CultureInvariant	Disables any culture-specific matching behavior (for example, what constitutes uppercase) when matching the separator strings.	RegexMatch
IgnorePatternWhitespace	Ignores unescaped whitespace and comments embedded in the pattern. This allows for commenting complex patterns.	RegexMatch
MultiLine	Treats a string as though it's composed of multiple lines. A line begins at a newline character and will be matched by the <code>^</code> pattern.	RegexMatch
Singleline	The default. Tells the pattern matcher to treat the entire string as a single line. Newlines in the string aren't considered the beginning of a line.	RegexMatch

(continued)

Option	Description	Applies to
ExplicitCapture	Specifies that the only valid captures are explicitly named or numbered ones of the form (?<name>...). This allows unnamed parentheses to act as non-capturing groups without the syntactic clumsiness of the expression (?:...).	RegexMatch

The `-split` operator also has a unary form that splits a string into a collection of smaller strings based on whitespace. The expression

```
-split "a b c"
```

produces an array with three elements in it: a, b, c.

Logical and bitwise operators

The logical operators are used to combine expressions to produce true or false answers. In PowerShell, 0, `$null`, `" "`, an empty array, and a one-element array containing a false value are treated as false. Everything else is treated as true. This table lists all the operators and provides examples of their use.

Operator	Description	Example	Result
-and	Does a logical and of the left and right values	0xff -and \$false	\$false
-or	Does a logical or of the left and right values	\$false -or 0x55	\$true
-xor	Does a logical exclusive-or of the left and right values	\$false -xor \$true \$true -xor \$true	\$true \$false
-not	Does the logical complement of the argument value	-not \$true	\$false
-band	Does a binary and of the bits in the values on the left and right side	0xff -band 0x55	85 (0x55)
-bor	Does a binary or of the bits in the values on the left and right side	0x55 -bor 0xaa	255 (0xff)
-bxor	Does a binary exclusive-or of the left and right values	0x55 -bxor 0xaa 0x55 -bxor 0xa5	255 (0xff) 240 (0xf0)
-bnot	Does the bitwise complement of the argument value	-bnot 0xff	-256 (0xffffffff00)

Operators for working with types

Here are the operators for testing object types and performing object type conversions.

<value> -is <type> <expr> -isnot <type> <expr> -as <type> [<type>] <expr>

Operator	Example	Result	Description
-is	\$true -is [bool]	\$true	This is true if the type of the left side matches the type of the object on the right side.
	\$true -is [object]	\$true	This is always true—everything is an object except \$null.
	\$true -is [ValueType]	\$true	The left side is an instance of a .NET value type such as an integer or floating-point number.
	"hi" -is [ValueType]	\$false	A string isn't a value type; it's a reference type. This expression returns false.
	"hi" -is [object]	\$true	A string is still an object.
	12 -is [int]	\$true	12 is an integer.
	12 -is "int"	\$true	The right side of the operator can be either a type literal or a string naming a type.
-isnot	\$true -isnot [string]	\$true	The object on the left side isn't of the type specified on the right side.
	\$null -isnot [object]	\$true	The null value is the only thing that isn't an object.
-as	"123" -as [int]	123	This takes the left side and converts it to the type specified on the right side.
	123 -as "string"	"123"	This turns the left side into an instance of the type named by the string on the right.

Additional unary operators

PowerShell includes unary operators usually found in C-like languages with the addition of the unary comma operator (creates a one-dimensional array) and casts using type literals (a type name enclosed in square brackets).

-not <value> +<value> -<value> [cast] <value> ,<value>
-- <assignableExpr> <assignableExpr> --
++ <assignableExpr> <assignableExpr> ++

Operator	Example	Result	Description
-	- (2+2)	-4	Negation. Tries to convert its argument to a number and then negates the result.

(continued)

Operator	Example	Result	Description
+	+ "123 "	123	Unary plus. Tries to convert its argument to a number and returns the result. This is effectively a cast to a number.
--	--\$a ; \$a--	Depends on the current value of the variable	Pre- and post-decrement operator. Converts the content of the variable to a number and then tries to subtract one from that value. The prefix version returns the new value; the postfix version returns the original value.
++	++\$a; \$a++	Depends on the current value of the variable	Pre- and post-increment. Converts the variable to a number and then adds 1 to the result. The prefix version returns the new value; the postfix version returns the original value.
[<type>]	[int] "0x123"	291	Typecast. Converts the argument into an instance of the type specified by the cast.
,	, (1+2)	One-element array containing the value of the expression	Unary comma operator. Creates a new one-element array of type [object[]] and stores the operand in it.

Expression and statement grouping operators

There are three grouping operators, as shown here.

(<pipeline>)	\$(<statementList>)	@(<statementList>)
----------------	-----------------------	----------------------

The semantics of these operators are described in the following table.

Operator	Example	Result	Description
(...)	(2 + 2) * 3 (Get-Date). dayofweek	12 The current week-day	Parentheses group expression operations and may contain either a simple expression or a simple pipeline. They may not contain more than one statement or thing, such as <code>while</code> loops.
\$(...)	\$(\$p = "a*"; Get-Process \$p)	Returns the process objects for all processes starting with the letter <i>a</i>	Subexpressions group collections of statements as opposed to being limited to a single expression. If the contained statements return a single value, it will be returned as a scalar. If the statements return more than one value, they will be accumulated in an array.

(continued)

Operator	Example	Result	Description
@(...)	@(dir c:\; dir d:\)	Returns an array containing the <code>FileInfo</code> objects in the root of the C:\ and D:\ drives	The array subexpression operator groups collections of statements in the same manner as the regular subexpression operator, but with the additional behavior that the result will always be returned as an array.

Array operators

PowerShell supports N-dimensional arrays as well as jagged arrays. Square brackets are used to reference elements in a type array including hash tables. If the index expression results in a collection of indexes, multiple array elements can be returned. Negative indexing is also supported. For example, -1 is the last element in the array.

```
<indexableValue> [ <indexExpression> ]  
  
<value1> , <value2> , <value3>  
  
<lowerBound> .. <upperBound>
```

An array is constructed using the comma (,) operator. This operator creates a new array using its left and right arguments. In a sequence of comma-separated values, all the values will be placed in the same array. The .. operator can be used to generate ranges of numbers.

Property and method operators

PowerShell uses the dot (.) operator to access an instance member on an object and :: to access static methods on a .NET class. No space is permitted before or after the dot. A member name can also be an expression (for example, a variable) as well as a constant string. This allows for indirection when referencing a member. The dot operator can also be used with hash tables. If a hash table key is used on the right side of the dot, the corresponding value will be returned.

```
<typeValue>::<memberNameExpr>    <typeValue>::<memberNameExpr>(<arguments>)  
  
<value>.<memberNameExpr>            <value>.<memberNameExpr>(<arguments>)
```

When calling a method, there can be no space between the opening parenthesis and the method name. If the parentheses are omitted on a method reference, the `PSMethod` object for that method will be returned.

The Get-Member cmdlet

In order to use a member on an object, you have to know that the member exists. This is what `Get-Member` does. If you pass an object to `Get-Member`, it will show you all of the object's members. Wildcards and member types can be used to limit the data returned:

```
PS (1) > Get-Date
Thursday, April 07, 2011 1:23:44 AM

PS (2) > Get-Date | Get-Member add* -MemberType method

    TypeName: System.DateTime
Name      MemberType Definition
-----
AddDays Method      System.DateTime AddDays(double value)

PS (3) > (Get-Date).AddDays(3)
Sunday, April 10, 2011 1:23:52 AM
```

The format operator (-f)

The format operator uses the .NET `System.String.Format` method to format text strings with a wide range of formatting options. The operator syntax is as follows:

`<formatSpecifiedString> -f <argumentList>`

For example, the following expression

```
"Decimal: {0} Hex: {0:x} 0 padded {0:d5}" -f 5,10,15
```

returns

```
Decimal: 5 Hex: 5 0 padded 00005
```

Format specifiers

The format string can contain any of the formatting specifiers listed in the following table.

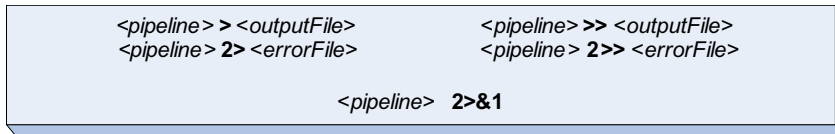
Format specifier	Description	Example	Output
{n}	Displays the <i>n</i> th argument to the operator	"{0} {1}" -f "a", "b"	a b
{0:x}	Displays a number in hexadecimal	"0x{0:x}" -f 181342	0x2c45e
{0:X}	Displays a number in hexadecimal with the letters in uppercase	"0x{0:X}" -f 181342	0x2C45E
{0:dn}	Displays a decimal number left-justified, padded with zeros	"{0:d8}" -f 3	00000003
{0:p}	Displays a number as a percentage	"{0:p}" -f .123	12.30 %

(continued)

Format specifier	Description	Example	Output
{0:C}	Displays a number using the currency symbol for the current culture	"{0:c}" -f 12.34	\$12.34
{0,n}	Displays with field width <i>n</i> , right aligned	" {0,5} " -f "hi"	hi
{0,-n}	Display with field width <i>n</i> , left aligned	" {0,-5} " -f "hi"	hi
{0:hh} {0:mm}	Displays the hours and minutes from a DateTime value	"{0:hh}:{0:mm}" -f (Get-Date)	01:34

PowerShell redirection operators

Like any good shell, PowerShell supports I/O redirection. The PowerShell redirection operators are shown in the following figure.



A description of each operator with an example is given in the next table.

Operator	Example	Result	Description
>	dir > out.txt	Contents of out.txt are replaced.	Redirects pipeline output to a file, overwriting the current contents.
>>	dir >> out.txt	Contents of out.txt are appended to.	Redirects pipeline output to a file, appending to the existing content.
2>	dir nosuchfile.txt 2> err.txt	Contents of err.txt are replaced by the error messages.	Redirects error output to a file, overwriting the current contents.
2>>	dir nosuchfile.txt 2>> err.txt	Contents of err.txt are appended with the error messages.	Redirects error output to a file, appending to the current contents.
2>&1	dir nosuchfile.txt 2>&1	The error message is written to the output.	The error messages are written to the output pipe instead of the error pipe.
<	Not implemented in PowerShell		This operator is reserved for input redirection, which isn't implemented in any version of PowerShell. Using this operator in an expression will result in a syntax error.

C.3.4 Flow-control statements

The following figure gives a summary of all the flow-control statements in PowerShell. More detailed coverage of some of the statements follows.

Conditional statements.

```
if ( <expr> ) { <statements> }  
if ( <expr> ) { <statements> } else { <statements> }  
if ( <expr> ) { <statements> } elseif ( <expr> ) { <statements> } else { <statements> }
```

Loop statements.

```
while ( <expr> ) { <statements> }  
do { <statements> } while ( <expr> )  
for ( <expr>; <expr>; <expr> ) { <statements> }  
foreach ( $var in <pipeline> ) { <statements> }
```

The break and Continue statements.

```
break  
continue  
break <label>  
continue <label>
```

The switch statement.

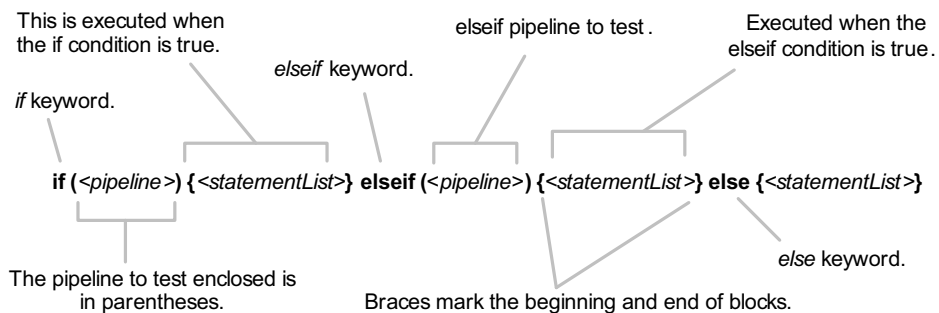
```
switch ( <expr> ) { <pattern1> { <statements> } <pattern2> { <statements> } }  
switch ( <expr> ) { <pattern1> { <statements> } default { <statements> } }
```

Flow-control cmdlets.

```
... | ForEach-Object <scriptBlock>  
... | ForEach-Object -Begin <scriptBlock> -Process <scriptBlock> -End <scriptBlock>  
... | Where-Object <scriptBlock>
```

The if statement

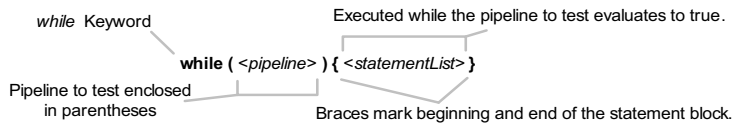
The basic conditional statement in PowerShell is the **if** statement, shown here.



In the figure, you can see that the basic **if** statement consists of an **if** clause followed by optional **elseif** and **else** clauses. In the condition part of the statement, a pipeline may be used as well as a simple expression.

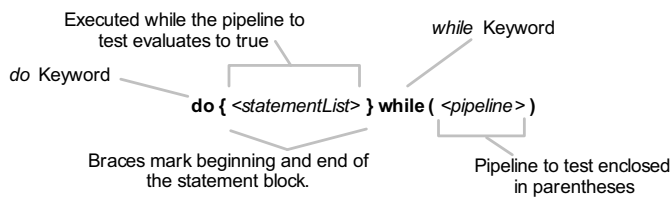
The while loop

The basic looping statement in PowerShell is the `while` statement shown in the following. As with the `if` statement, the condition part can be a pipeline as well as simple expressions.



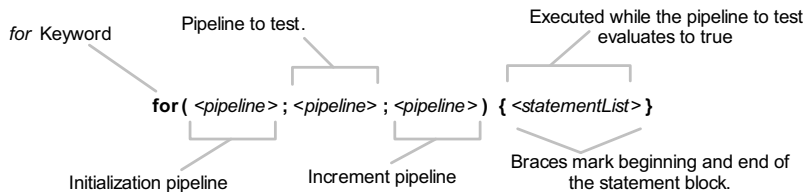
The do-while loop

This `do-while` loop is a variation of the `while` loop that always executes once with the conditional test at the bottom of the loop instead of at the top.



The for loop statement

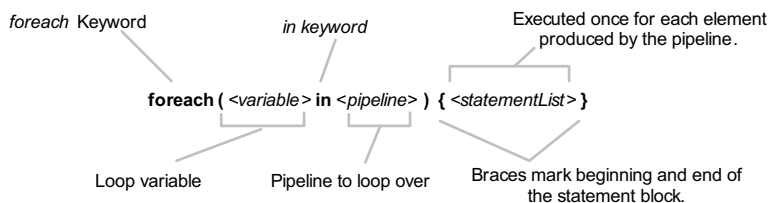
The `for` loop includes initialization and finalization clauses in the loop syntax.



One difference between the `for` loop in PowerShell and the `for` loop in C# is that the initialization part doesn't require a new variable to be defined.

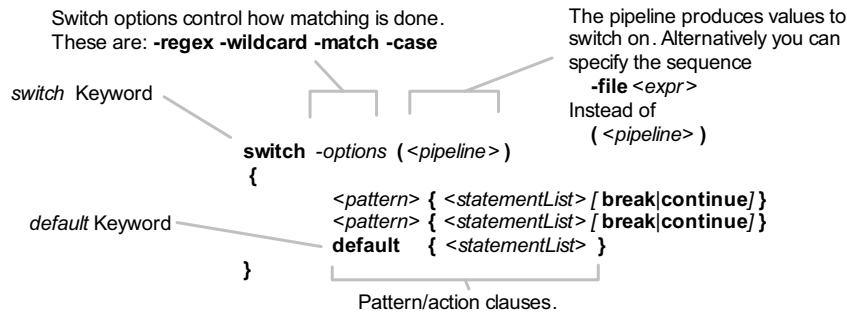
The foreach loop

The `foreach` loop cycles through each element in a collection. If a scalar value is specified instead of a collection, the statement will loop once for that object. It will also loop once if that value is `$null` because the empty list `$()` isn't the same as `$null`.



The switch statement

The PowerShell `switch` statement combines looping with pattern matching. The pipeline to test emits a collection; the `switch` statement will loop once for each item. The `-file` switch and a file path may be specified instead of a pipeline for text file processing.



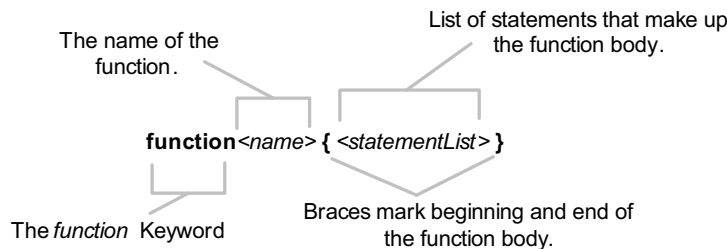
All matching clauses are executed unless a `break` or `continue` statement is executed. The `default` clause is executed only if there are no other matches.

C.4 DEFINING FUNCTIONS AND SCRIPTS

PowerShell supports a scalable syntax for defining functions, starting from very simple through the specification of complex constraints. Function definitions are statements executed at runtime. This means you can't call a function in a script before it has been defined. Functions can also be dynamically redefined at runtime using the `function:` drive or by following a different path through the code.

C.4.1 Simple function definitions

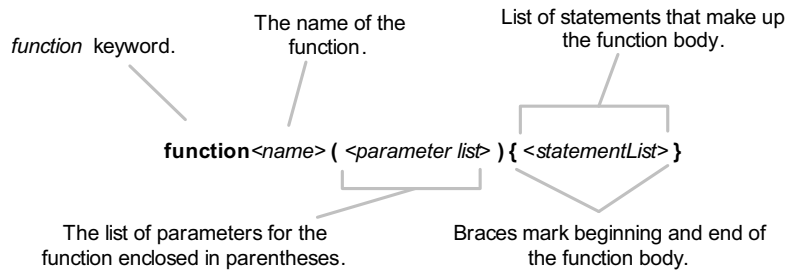
The following figure shows the simplest form of a function definition in PowerShell. It has no formal parameter declarations. Arguments are available through the special variable `$args`.



An example of this type of function specification is

```
function noParams { "I have " $args.Length + " arguments" }
```

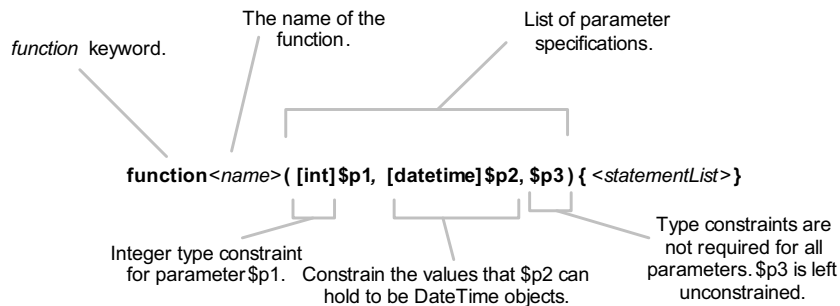
The next step up is to add format parameter definitions, as shown here.



An example of this syntax is

```
function sum ($x, $y) { $x + $y }
```

Parameters may include optional type constraints. These constraints don't require that the argument be of that type as long as it can be automatically converted to the target type.



Here is an example of this form of function:

```
function sum ([int] $x, [int] $y) { $x + $y }
```

Applying the function like this

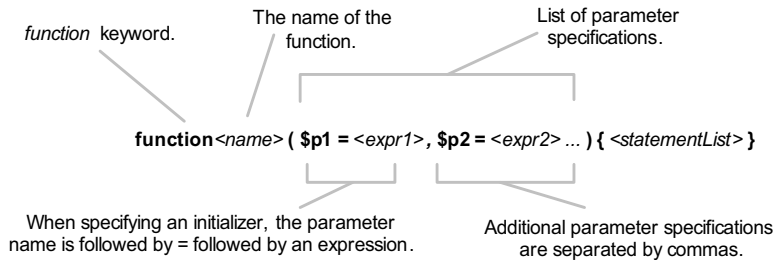
```
sum "12" 4
```

succeeds because the string "12" can be automatically converted to the number 12. But this

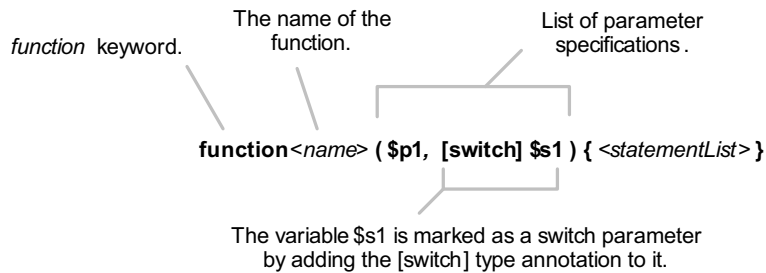
```
sum "abc" 3
```

fails because "abc" can't be converted to a number.

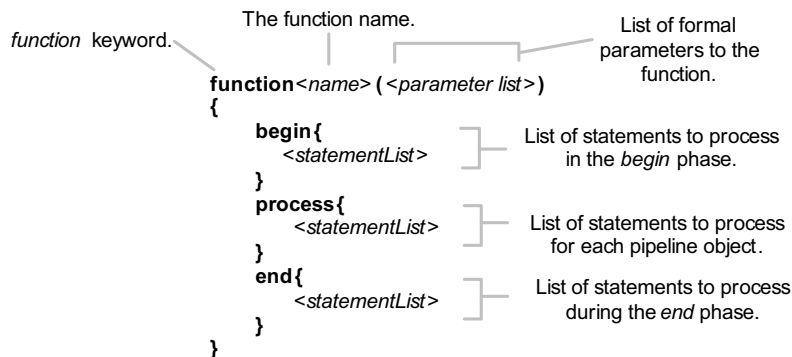
Function parameters can also have initializer expressions. This figure shows this more complex function definition syntax where initializer expressions are provided for each variable. Note that the initializers are constrained to be expressions, but using the subexpression notation, you can put anything here.



Parameters that don't take arguments are called *switches* and are defined by applying the `[switch]` type constraint to the parameter, as shown here.



Functions may also contain all the execution-phase clauses needed for pipeline processing. The `begin` block is executed at the beginning of statement execution. The `process` block is executed once per input object, which is available through the special variable `$_`. The `end` block is executed when the pipeline operation is complete.



The following function uses these phases to sum all the pipeline input objects:

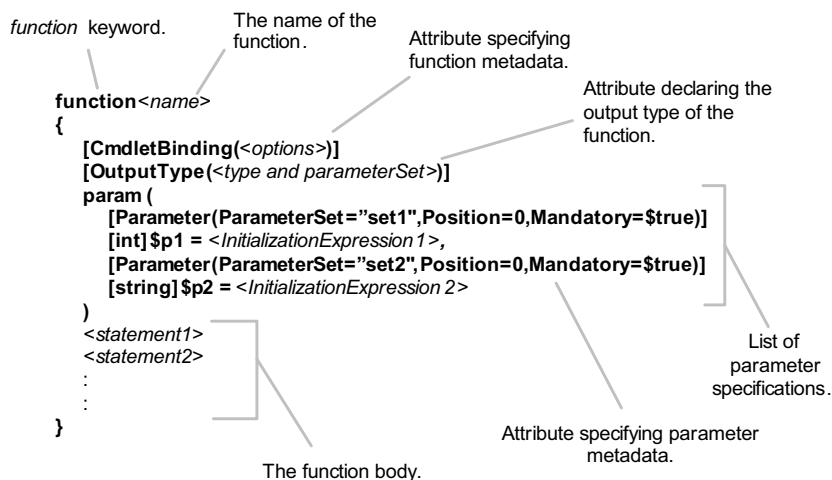
```
function sumPipe { begin {$total=0} process {$total += $_} end {$total}}
```

Calling this function in the following pipeline adds the objects from 1 to 5:

```
1,2,3,4,5 | sumPipe
```

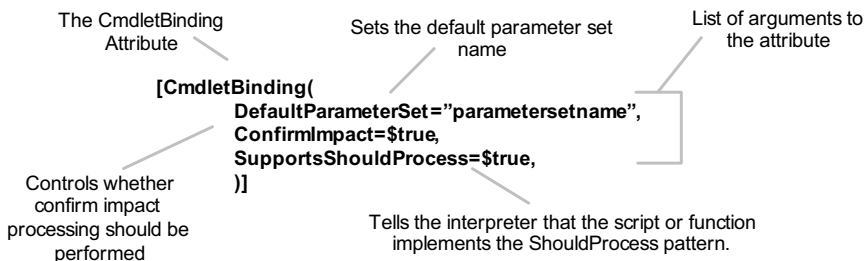
C.4.2 Advanced function declarations

You can enable additional parameter processing features by adding attributes to the function declaration. The following figure shows how these metadata attributes are added in function definitions. Attributes that apply to the entire function appear before the `param` statement, and attributes for an individual parameter appear before the parameter declaration.



The CmdletBinding Attribute

The following figure shows all of the properties that can be specified for the `[CmdletBinding()]` attribute. The PowerShell runtime uses these properties to control the execution of the associated function or script.



The following figure shows the function annotations needed to enable `ShouldProcess` support. The `SupportsShouldProcess` property of the `[CmdletBinding()]` attribute should be set to true, and there must be a call to the `ShouldProcess()` method in the body of the code (see section 8.2.2).

The *CmdletBinding* attribute must precede the *param* keyword.

```
function Stop-ProcessUsingWMI
{
    [CmdletBinding (SupportsShouldProcess = $True)] param (
        [regex] $pattern = "notepad "
    )
    foreach ($process in Get-WmiObject Win32_Process |
        where {$_.Name -match $pattern})
    {
        if ($PSCmdlet.ShouldProcess (
            "process $($process.Name) (id: $($process.ProcessId))",
            "Stop Process"))
        {
            $process.Terminate ()
        }
    }
}
```

The function must call the *ShouldProcess()* method.

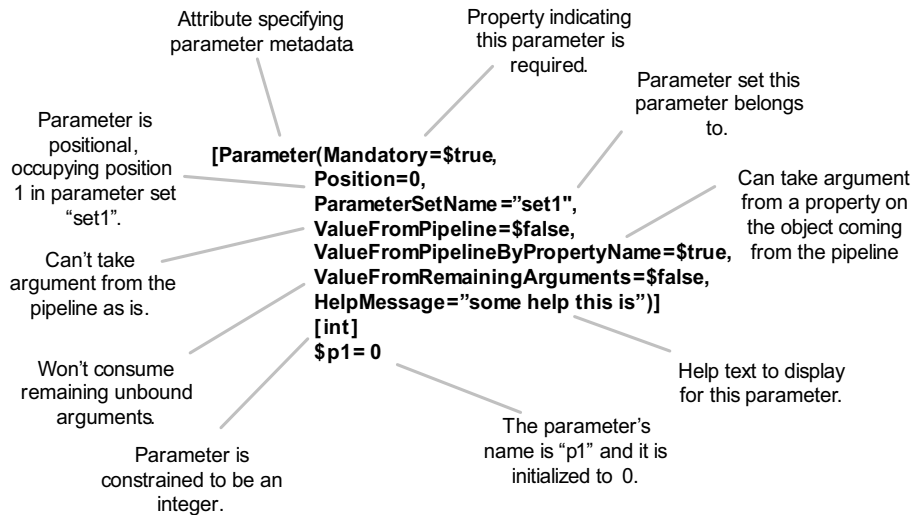
Action message.

If the call to *ShouldProcess()* returns true, execute the action.

Caption for prompting.

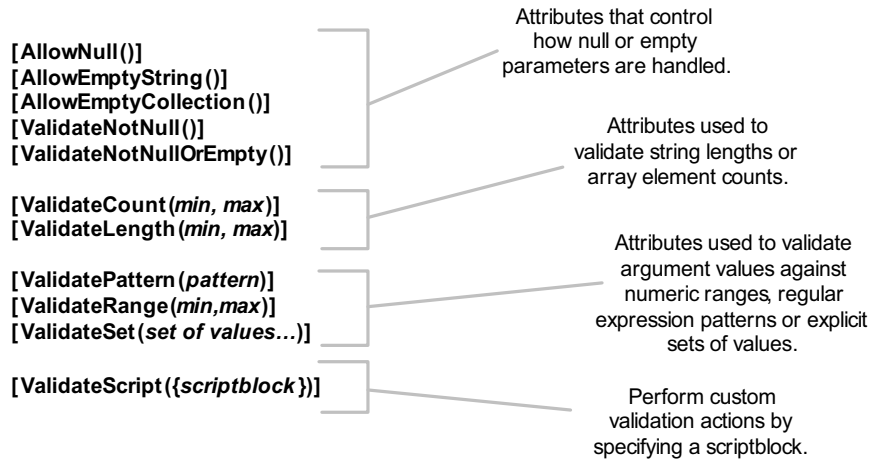
The *[Parameter()]* attribute

The following figure is a detailed view showing how the *[Parameter()]* attribute is applied as well as all the properties that can be specified for this attribute.



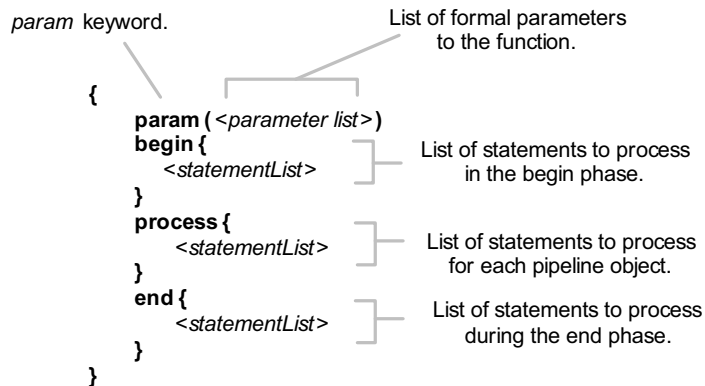
Parameter constraint attributes

The following figure shows the default set of constraint attributes that can be applied to parameters.



C.4.3 Scriptblocks

Scriptblocks are functions that don't have a name, also known as anonymous functions or *lambdas*. Scriptblocks can have all the syntactic elements used in named function definitions except for the `function` keyword and name.



A scriptblock is executed using the call operator (`&`):

```
$sb = { param ( [int] $x, [int] $y) $x + $y }
& $sb 3 6
```

You can turn a scriptblock into a named function by assigning it to a name using the `function:` drive:

```
$function:sum = { param( [int] $x, [int] $y) $x + $y }
```

C.4.4 Inline help for functions

Structured help information can be specified inline for functions and scripts. This information can then be retrieved using the [Get-Help](#) cmdlet. A certain amount of help information is generated automatically. This table describes these automatically generated fields.

Help element	Description
Name	Comes from the name of the function. For a script, it's taken from the name of the script file.
Syntax	Generated from the function or script syntax (in other words, the parameter definitions).
Parameter List	Generated from the function or script syntax and from the descriptions you add to the parameters.
Common Parameters	Added to the syntax and parameter list of the help topic, even if they have no effect.
Parameter Attribute Table	Appears when you use the <code>-Full</code> or <code>-Parameter</code> parameter of <code>Get-Help</code> . The values of the <code>Required</code> , <code>Position</code> , and <code>Default</code> properties are taken from the function or script definition.
Remarks	Generated from the function or script name.

You can define additional help topic elements by using comments containing special tags. The following tags are supported.

Tag name	Tag content
.SYNOPSIS	Brief description of the function or script. This tag can be used only once in each help topic.
.DESCRIPTION	Detailed description of the function or script.
.PARAMETER	Description of a parameter.
.EXAMPLE	Example showing how to use a command.
.INPUTS	Type of object that can be piped into a command.
.OUTPUTS	Types of objects that the command returns.
.NOTES	Additional information about the function or script.
.LINK	Name of a related topic.
.COMPONENT	Technology or feature that the command is associated with.
.ROLE	User role for this command.
.FUNCTIONALITY	Intended use of the function.
.FORWARDHELPTARGETNAME	Redirects to the help topic for the specified command.
.FORWARDHELPCATEGORY	Help category of the item in the <code>FORWARDHELPTARGETNAME</code> tag.
.EXTERNALHELP	Path to an external help file for the command.

Example function with help annotations

The following is an example function that specifies inline help and uses many of the parameter declaration attributes:

```
function sum2 {  
    <#  
    .SYNOPSIS  
    A function to sum two numbers  
    .DESCRIPTION  
    This function is used to add two numbers together  
    .EXAMPLE  
    sum2 2 3  
    #>  
    param (  
        # The first number  
        [Parameter(ParameterSetName="first",Mandatory=$true,Position=0)]  
        [ValidateRange(1,100)]  
        [int]  
        $x,  
        # the second number  
        [Parameter(ParameterSetName="first",Mandatory=$true,Position=1)]  
        [ValidateSet(2,4,6,8,10)]  
        $y)  
  
    $x + $y  
}
```

Comments specified before a parameter definition in the parameter declaration are used to generate help for that parameter.

C.5 MODULES

You can extend PowerShell by loading modules. There are two basic types of modules: binary modules implemented as a .NET assembly (.dll) and script modules implemented by a PowerShell script file with a .psm1 file extension. There are seven module cmdlets.

Command	Description
Get-Module	Lists loaded modules. If -ListAvailable is specified, then the modules available to be loaded are displayed instead. If -All is specified, all modules are listed, both root modules and nested modules.
Import-Module	Loads a module into memory. A module is only loaded into memory once, so to force a reload, use the -Force parameter.
Remove-Module	Removes a module from memory. Because a .NET assembly can't be removed from memory, it's unregistered instead.
New-ModuleManifest	Creates a module manifest for a module that contains metadata like module author, version number, and so on.
Test-ModuleManifest	Tests the validity of the manifest file. It doesn't verify that the information in the manifest is correct.

(continued)

Command	Description
New-Module	Creates a new in-memory-only module called a <i>dynamic</i> module.
Export-ModuleMember	Controls which function, cmdlet, aliases, or variables are exported from the script module.

Modules are isolated from the caller's environment. Only the global scope is shared. If a module contains no calls to `Export-ModuleMember`, then all commands defined in the module are exported by default. Modules are discovered using the `$ENV:PSModulePath` environment variable. Here are some examples:

```
PS (1) > Get-Module
PS (2) > Get-Module -ListAvailable bits*
```

ModuleType	Name	ExportedCommands
Manifest	BitsTransfer	{}

```
PS (3) > Import-Module BitsTransfer
PS (4) > Get-Module
```

ModuleType	Name	ExportedCommands
Manifest	BitsTransfer	{Start-BitsTransfer, Remove-B...

C.6 REMOTING

PowerShell has built-in support to provide remote access to other computers.

C.6.1 Enabling remoting

Remoting is enabled using the `Enable-PSRemoting` cmdlet:

```
Enable-PSRemoting
```

This command must be run elevated. Running it enables remote access to the default remoting configurations.

Default configuration names

PowerShell remoting connections are defined using two elements: the target computer and the target configuration. If no computer name is specified to a remoting command, it will default to `localhost`. If no configuration name is specified, the default configuration name on the target computer will be used.

There is one default configuration on 32-bit operating systems named `Microsoft.PowerShell`. On 64-bit operating systems, there are two: `Microsoft.PowerShell` (runs 64-bit) and `Microsoft.PowerShell132` (runs

32-bit.) If no configuration name is specified, the `Microsoft.PowerShell` configuration is used.

Creating custom configurations

You can create custom configurations with the `Register-PSSessionConfiguration` cmdlet:

```
Register-PSSessionConfiguration -Name configName `
    -StartupScript $pwd/configscript.ps1
```

Use `Unregister-PSSessionConfiguration` to remove configurations you have created.

Controlling access to a configuration

By default, only members of the local administrators group can access a computer through remoting. Use the `Set-PSSessionConfiguration` cmdlet to change this either by doing

```
Set-PSSessionConfiguration -ConfigurationName <name> `
    -SecurityDescriptorSDDL
```

or through the GUI:

```
Set-PSSessionConfiguration -ConfigurationName <name> `
    -ShowSecurityDescriptorUI
```

Setting the trusted hosts list

In a non-domain environment, machines that you want to contact through remoting need to be added to the trusted hosts list. This can be done as follows:

```
Set-Item wsman:\localhost\Client\TrustedHosts <hostname list>
```

To trust all hosts, set the list to `*`:

```
Set-Item wsman:\localhost\Client\TrustedHosts *
```

C.6.2 The Invoke-Command cmdlet

This figure shows the syntax for the `Invoke-Command` cmdlet. This cmdlet is used to execute commands and scripts on one or more computers. It can be used synchronously or asynchronously as a job.

```
Invoke-Command
[[[-ComputerName]<string[]>] [-Port<int>]
[-ApplicationName<string>]
[-ThrottleLimit<int>] [-UseSSL]
[-Authentication<authenticationMethod>]
[-ConfigurationName<string>]
[-Credential<PSCredential>]
[-HideComputerName]
[-JobName<string>]
[-AsJob]
[-SessionOption<PSSessionOption>]
[-ScriptBlock]<scriptblock>
[-ArgumentList<object[]>] [-InputObject<PSObject>]
```


.NET types serialized with fidelity by PowerShell remoting

The following table lists the types transmitted with fidelity using the PowerShell Remoting Protocol.

String	Character	Boolean	DateTime
Duration	Unsigned Byte	Signed Byte	Unsigned Short
Signed Short	Unsigned Int	Signed Int	Unsigned Long
Signed Long	Float	Double	Decimal
Array of Bytes	GUID	URI	Null Value
Version	XML Document	ScriptBlock	Secure String
Progress Record	Enums	Stack	Queue
List	Hash tables		

All other types are converted to property bags and rehydrated as instances of `PSCustomObject` with note properties containing the values of the original corresponding properties. The deserialized object also includes the type name of the original object prefixed with `Deserialized`.

C.7 ERROR HANDLING

Error handling in PowerShell blends elements from shells and programming languages using a mix of streams and exception handling.

C.7.1 Output streams

Nonterminating errors, warnings, debug messages, and verbose output are written to the corresponding streams. The behavior of the system when an object is written into the stream is controlled by the preference variables listed in the table.

Stream	Cmdlet	Preference variable	Default value
Error	Write-Error	\$ErrorActionPreference	Continue
Warning	Write-Warning	\$WarningPreference	Continue
Debug	Write-Debug	\$DebugPreference	SilentlyContinue
Verbose	Write-Verbose	\$VerbosePreference	SilentlyContinue
Progress	Write-Progress	\$ProgressPreference	Continue

Stream preference values

The following values determine the behavior of the object streams when assigned to the corresponding preference variable.

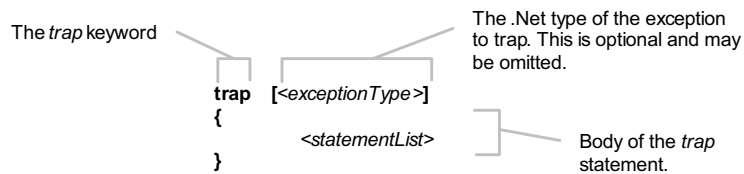
Preference value	Descriptions
Continue	The output object is written and execution continues. For errors, execution then continues at the next script line.
SilentlyContinue	The object isn't written to the output pipe before continuing execution.
Stop	The object is written, but execution doesn't continue.
Inquire	The object is written, and the user is promoted to Continue or Stop.

C.7.2 Error-handling statements

There are two error-handling statements in PowerShell: the `trap` statement, which is similar to the Basic `OnError` statement; and `try/catch/finally`, which matches the C# statement.

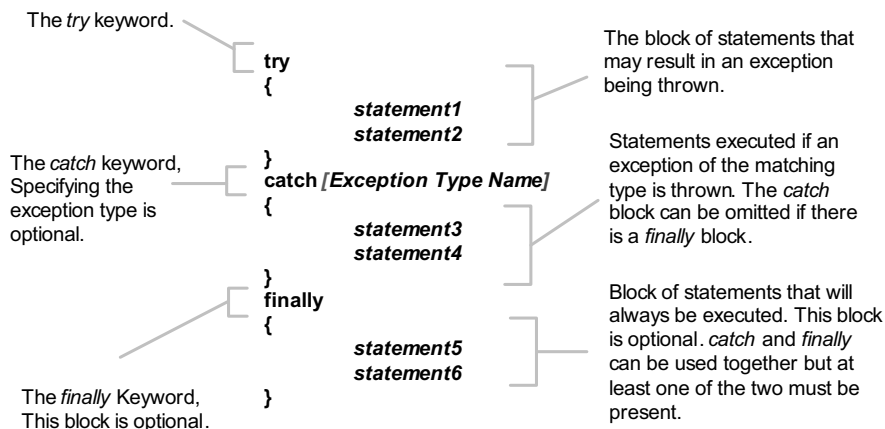
The trap statement

The following figure shows the syntax of the `trap` statement. The body of the statement is executed when an exception with matching type is caught. If no exception type is specified, all exceptions are caught.



The try/catch/finally statement

This next figure shows the syntax of the `try/catch/finally` statement. This statement allows exceptions to be processed in a more structured way than the `trap` statement.



The throw statement

To generate an exception, use the `throw` statement.



C.8 POWERSHELL DEBUGGER

The PowerShell debugger works both in the ISE and at the console. A script must be saved to a file before it can be debugged. Debugging of in-memory functions isn't supported. In the console window, cmdlets are the only interface to the debugger. In the ISE, the Debug menu provides access to debugger functionality.

C.8.1 PowerShell debugger cmdlets

Commands are used to set breakpoints and investigate the state of the interpreter.

Cmdlet	Description
Get-PSCallStack	Gets the current call stack
Enable-PSBreakPoint	Enables an existing breakpoint
Disable-PSBreakPoint	Disables a breakpoint without removing it
Set-PSBreakPoint	Sets a breakpoint
Get-PSBreakPoint	Gets the list of breakpoints
Remove-PSBreakPoint	Removes an existing breakpoint

C.8.2 Debug mode commands

This table lists the special commands that are available when you're stopped at a breakpoint.

Command	Full name	Description
S	Step-Into	Steps execution to the next statement. If the statement to execute is a function or script, the debugger will step into that command. If the script or function isn't currently displayed in an editor tab, a new editor tab will be opened.
V	Step-Over	The same as Step-Into except that function/script calls aren't followed.
O	Step-Out	Executes statement until the end of the current function or script is reached.
C	Continue	Resumes execution of the script. Execution continues until another breakpoint is reached or the script exits.

(continued)

Command	Full name	Description
L [<m> [<n>]]	List	Lists the portion of the script around the line where you're currently stopped. By default, the current line is displayed, preceded by the 5 previous lines and followed by the 10 subsequent lines. To continue listing the script, press the Enter key. The List command can be optionally followed by a number specifying the number of lines to display before and after the current line. If two numbers are specified, the first number is the number of preceding lines and the second is the number of following lines.
Q	Stop	Stops execution, which also stops the debugger.
K	GetPSCallStack	Displays the function call stack up from the current execution point. This command isn't specific to debugger mode and may be used anywhere, including non-interactively in a script.
<Enter>		Pressing the Enter key on an empty line repeats the last command entered. This makes it easy to continue stepping or list the script.
?, h		Displays all the special debugger commands in the output window.

C.9 PERFORMING TASKS WITH POWERSHELL

This section covers how to perform common tasks or operations using PowerShell.

C.9.1 File operations

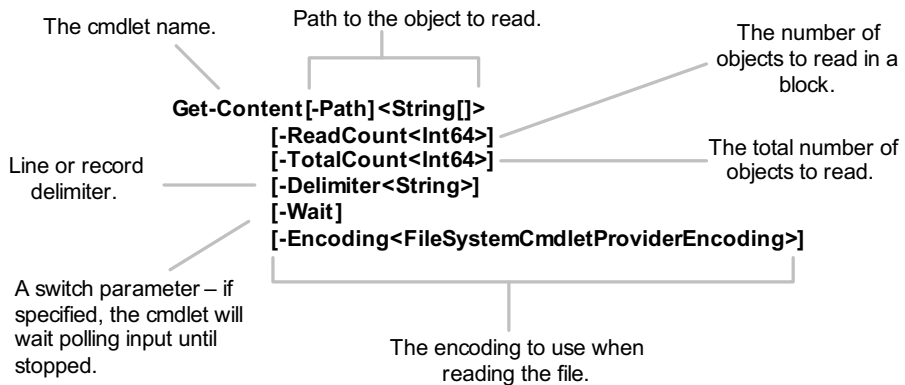
File operations in PowerShell are performed using what are called the *core cmdlets*. These cmdlets operate on top of the PowerShell provider layer, allowing the same set of commands to access the file system, Registry, environment, and so on. The following table lists the core cmdlets along with their aliases and the equivalent file commands in other shells.

Cmdlet	Canonical	Cmd.exe	UNIX	Description
Get-Location	gl	cd	pwd	Gets the current directory.
Set-Location	sl	cd, chdir	cd, chdir	Changes the current directory.
Copy-Item	cpi	copy	cp	Copies files.
Remove-Item	ri	del, rd	rm, rmdir	Removes a file or directory. PowerShell has no separate command for removing directories as opposed to files.
Move-Item	mi	move	mv	Moves a file.
Rename-Item	rni	ren	mv	Renames a file.

(continued)

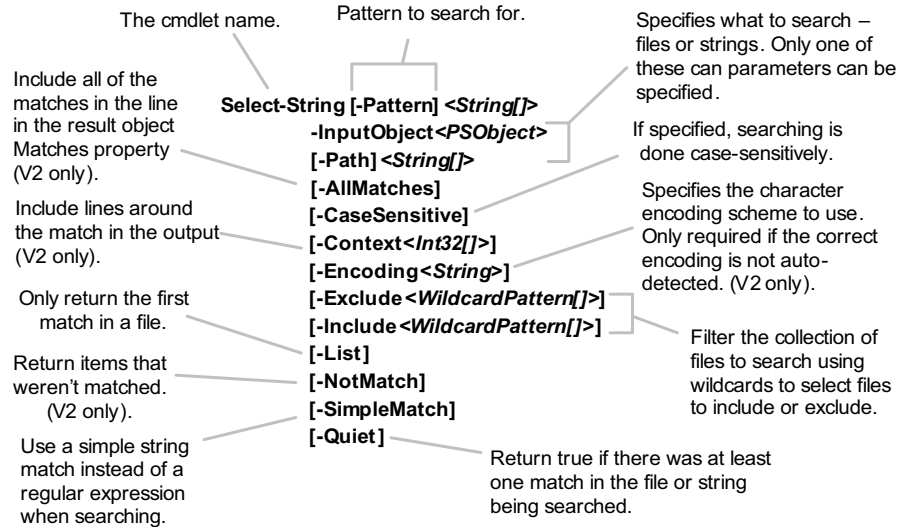
Cmdlet	Canonical	Cmd.exe	UNIX	Description
Set-Item	si			Sets the contents of a file. If the file doesn't exist, a new file is created.
Clear-Item	cli			Clears the contents of a file.
New-Item	ni			Creates a new empty file or directory. The type of object is controlled by the -Type parameter (file or folder).
MkDir		md	mkdir	Implemented as a function in PowerShell so you can create directories without having to specify the type of object to create.
Get-Content	gc	type	cat	Sends the contents of a file to the output stream.
Set-Content	sc			Sets the contents of a file. UNIX and cmd.exe have no equivalent; redirection is used instead.
Out-File				Write the contents of a file. This differs from Set-Content in that the objects are passed through the formatter before being written.

The `Get-Content` cmdlet is used to read content from a content provider's store. This includes the file system and Registry. In the file system, different encodings can be specified while reading a file. The supported encodings are `Unknown`, `String`, `Unicode`, `Byte`, `BigEndianUnicode`, `UTF8`, `UTF7`, and `ASCII`.



Searching through files

Files can be searched for text patterns using the [Select-String](#) cmdlet.



[Select-String](#) supports wildcard patterns for specifying sets of text to search but doesn't directly support recursively searching through a tree of files. To do a recursive search, pipe the output of [dir](#) ([Get-ChildItem](#)) into [Select-String](#) as follows:

```
dir $ENV:windir -Recurse -Filter *.ps1 | Select-String "function.*get"
```

To get a listing of just the files in a file tree that contain the pattern, use the [-List](#) parameter and look at the [Path](#) property on the output objects:

```
dir $ENV:windir -Recurse -Filter *.ps1 |  
  Select-String "function.*get" -List |  
  foreach { $_.Path }
```

C.9.2 Working with XML

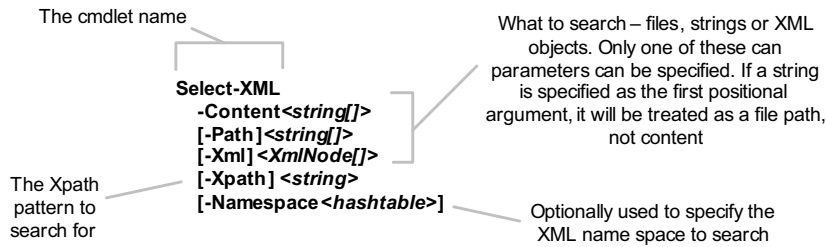
XML documents are constructed in PowerShell using the [\[xml\]](#) type accelerator and casting strings into XML documents as follows:

```
$doc = [xml] "<top><a>1</a><b>2</b></top>"
```

You can access elements in an XML document object using the dot operator as though they were object properties:

```
$doc.top.a
```

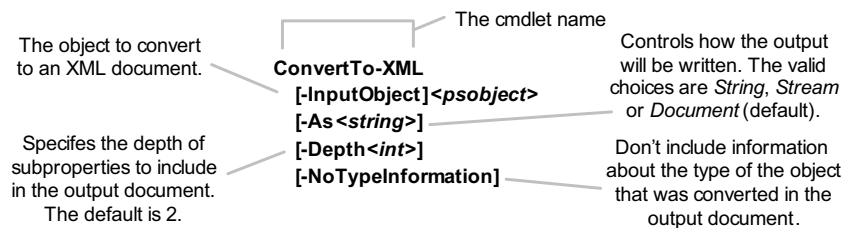
There are also cmdlets for working with XML. To query an XML document using [XPath](#), use the [Select-XML](#) cmdlet as shown here.



The following table shows examples of some common [XPath](#) patterns.

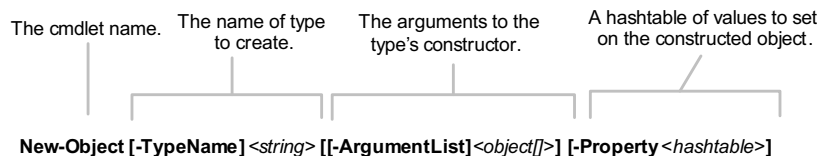
XPath expression	Description	Equivalent file operation
/	Gets all the nodes under the document root	dir /
.	Selects the current node	Get-Item .
..	Selects the parent node	Get-Item ..
a	Selects all the children under the node a	dir a
/a/b/c	Get all nodes under the path /a/b/c	dir /a/b/c
//b	Get all elements with the name b anywhere in the document	dir -rec -Filter b

The [ConvertTo-XML](#) cmdlet lets you convert objects into a simple XML encoding. The following shows the options that control how this encoding is done.



C.9.3 Creating .NET object instances

Instances of .NET classes are created using the [New-Object](#) cmdlet using this syntax.



If a hash table is passed to the `-Property` parameter, properties on the new object instance whose names correspond to hash table keys are initialized with the values from the hash table.

Specifying .NET type names

Unless there is a type accelerator or alias for a type name, the full name of the type must be specified when creating an instance: for example, `System.IO.File`. If the type is in the `System` namespace, `System` may be omitted: for example, `IO.File`. Type names aren't case sensitive.

A number of commonly used types have type accelerator aliases defined for them to make them easier to use. These are shown in the following table. Accelerators that aren't available in PowerShell v1 are indicated in the table.

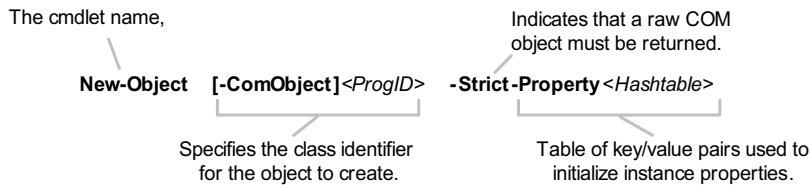
Type alias	Corresponding .NET type	Version
[int]	System.Int32	v1 and 2
[long]	System.Int64	v1 and 2
[string]	System.String	v1 and 2
[char]	System.Char	v1 and 2
[bool]	System.Boolean	v1 and 2
[byte]	System.Byte	v1 and 2
[double]	System.Double	v1 and 2
[decimal]	System.Decimal	v1 and 2
[float]	System.Single	v1 and 2
[single]	System.Single	v1 and 2
[regex]	System.Text.RegularExpressions.Regex	v1 and 2
[array]	System.Array	v1 and 2
[xml]	System.Xml.XmlDocument	v1 and 2
[scriptblock]	System.Management.Automation.ScriptBlock	v1 and 2
[switch]	System.Management.Automation.SwitchParameter	v1 and 2
[hashtable]	System.Collections.Hashtable	v1 and 2
[ref]	System.Management.Automation.PSReference	v1 and 2
[type]	System.Type	v1 and 2
[psobject]	System.Management.Automation.PSObject	v1 and 2
[pscustomobject]	System.Management.Automation.PSObject	v2
[psmoduleinfo]	System.Management.Automation.PSModuleInfo	v2
[powershell]	System.Management.Automation.PowerShell	v2
[runspacefactory]	System.Management.Runspaces.RunspaceFactory	v2

(continued)

Type alias	Corresponding .NET type	Version
[runspace]	System.Management.Automation.Runspaces.Runspace	v2
[ipaddress]	System.Net.IPAddress	v2
[wmi]	System.Management.ManagementObject	v1 and 2
[wmisearcher]	System.Management.ManagementClass	v1 and 2
[wmiiclass]	System.Management.ManagementClass	v1 and 2
[adsisearcher]	System.DirectoryServices.DirectoryEntry	v1 and 2
[adsisearcher]	System.DirectoryServices.DirectorySearcher	v2

C.9.4 Creating COM object instances

The `New-Object` cmdlet is also used for creating COM objects, using the following syntax.



If a hash table is passed to the `-Property` parameter, properties on the new object instance whose names correspond to hash table keys are initialized with the values from the hash table.

ProgIDs for common COM objects

To create a COM object, you need to know its `ProgID`. The following table lists some common `ProgIDs` useful for accessing Windows features.

Class ProgID	Description
Shell.Application	Provides access to Windows Explorer and its capabilities. Allows automation of many shell tasks such as opening file browser windows; launching documents or the help system; finding printers, computers, or files; and so on.
SAPI.SpVoice	Provides access to the Microsoft Speech API. Allows text-to-speech with the <code>Speak("string")</code> method.
WMPPlayer.OCX	Manipulates the Windows Media Player.
MMC20.Application	Manipulates the Microsoft Management Console application.
Microsoft.Update.Session	Provides access to the Windows Update Agent (WUA), allowing you to scan, download, and install updates.

(continued)

Class ProgID	Description
Schedule.Service	Lets you create and schedule tasks, get a list of running tasks, and delete task definitions (see section 18.6 for examples).
WScript.Shell	Provides access to system information and environment variables, and lets you create shortcuts and work with the Registry (see section 18.3 for examples).

C.10 WINDOWS MANAGEMENT INSTRUMENTATION (WMI)

WMI is the major source for systems management data on Windows. This information is organized as a set of types arranged in a hierarchy. A subset of this hierarchy is shown here.

Provider	Namespace	Description
Active Directory	root\directory\ldap	Provides access to Active Directory objects
Event log	root\cimv2	Provides classes to manage Windows event logs
Performance Counter	root\cimv2	Provides access to raw performance data counters
Registry	root\default	Provides access to the Registry, allowing you to read, write, enumerate, monitor, create, and delete Registry keys and values
SNMP	root\snmp	Provides access to SNMP MIB data and traps from SNMP-managed devices
WDM	root\wmi	Provides access to information about Windows device drivers
Win32	root\cimv2	Provides a broad array of classes for managing a computer including information about the computer, disks, peripheral devices, networking components, operating system, printers, processes, and so on
Windows Installer	root\cimv2	Provides access to information about software installed on this computer

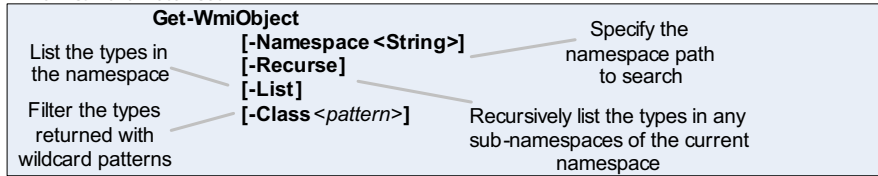
C.10.1 WMI cmdlets

The following cmdlets are used when working with WMI.

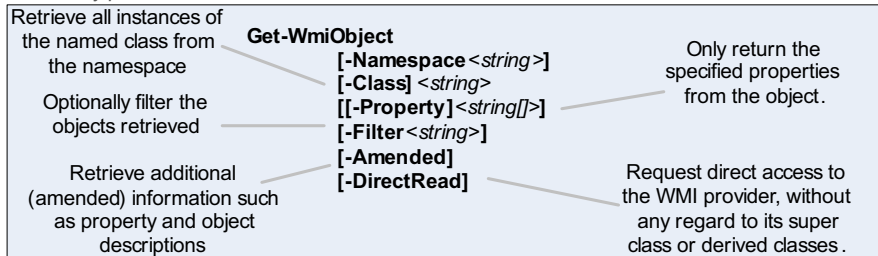
Cmdlet	Description	Availability
Get-WmiObject	Retrieves objects from WMI	v1 and v2
Set-WmiInstance	Sets the properties on a WMI class or object	v2
Invoke-WmiMethod	Invokes a method on a WMI class or instance	v2
Remove-WmiObject	Removes an object instance from the repository	v2

The signature for the `Get-WmiObject` cmdlet is shown in the following figure. This cmdlet has three parameter sets; you can explore WMI using the `-List` parameter and retrieve objects from the repository with the `-Class` and `-Query` parameters.

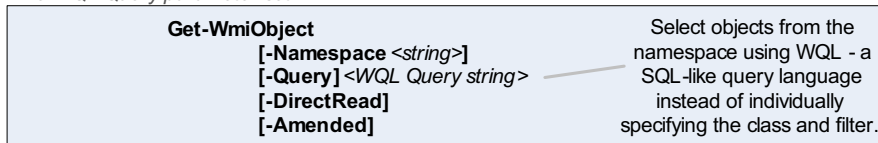
The List Parameter set



The Query parameter set



The WQL Query parameter set



WQL operators that can be used in the -Filter parameter

WMI has a native SQL-like query language for retrieving objects. The following table lists the operators that can be used in a WQL query, the nearest PowerShell language equivalents, and a description of what they do.

WQL operator	PowerShell operator	Operator description
=	-eq	Equal to.
<	-lt	Less than.
>	-gt	Greater than.
<=	-le	Less than or equal to.
>=	-ge	Greater than or equal to.
!= or <>	-ne	Not equal to.
AND	-and	ands two Boolean expressions, and returns TRUE when both expressions are TRUE; equivalent to -and in PowerShell.

(continued)

WQL operator	PowerShell operator	Operator description
OR	-or	Combines two conditions. When more than one logical operator is used in a statement, the OR operators are evaluated after the AND operators.
TRUE	\$true	Boolean operator that evaluates to -1 (minus one).
FALSE	\$false	Boolean operator that evaluates to 0 (zero).
NULL	\$null	Indicates that an object doesn't have an explicitly assigned value. NULL isn't equivalent to zero (0) or blank.
ISA	-as	Operator that applies a query to the subclasses of a specified class.
IS	n/a	Comparison operator used with NOT and NULL. The syntax for this statement is the following: IS [NOT] NULL (where NOT is optional).
LIKE	-like	Similar to the PowerShell -like operator that does wildcard matches, except the wildcard characters are different: % (any string), _ (any single character), [ab=z] (matches character range, ^ says don't match range).
NOT	not	Returns the logical complement of the argument value.
CLASS	n/a	References the class of the object in a query.

C.10.2 WMI type accelerators

Support of WMI is also included through three type aliases shown in the next table. All these types are in the [System.Management](#) namespace.

Alias	Type name	Description and example
[wmi]	ManagementObject	Represents a WMI object instance. Can be used to cast a WMI object path into an instance. <code>[wmi] '\\MyMachine\root\cimv2:Win32_Process.Handle="0"'</code>
[wmisearcher]	ManagementObject-Searcher	Represents a WMI query. Can be used to cast a string representing a query into a search object. <code>\$q = [wmisearcher]'select * from Win32_Process' ; \$q.Get()</code>
[wmiiclass]	ManagementClass	An instance of the metadata for the WMI class. <code>[wmiiclass]'Win32_BIOS'</code>

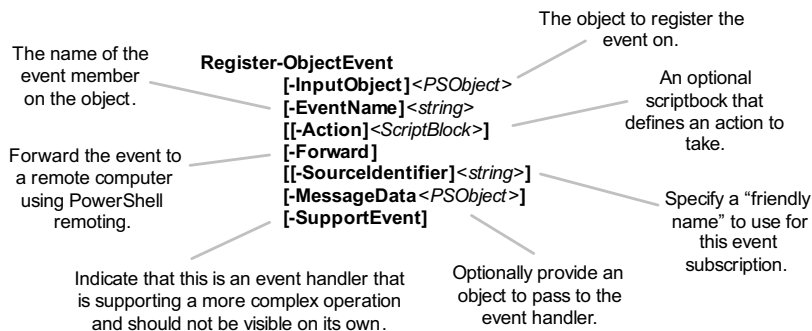
C.11 POWERSHELL EVENTING

PowerShell eventing is a feature introduced in PowerShell v2 that allows PowerShell scripts to deal with asynchronous events. This feature is accessed using the following cmdlets.

Cmdlet name	Description
Register-ObjectEvent	Registers an event subscription for events generated by .NET objects.
Register-WmiEvent	Registers an event subscription for events generated by WMI objects (see chapter 19).
Register-EngineEvent	Registers an event subscription for events generated by PowerShell.
Get-EventSubscriber	Gets a list of the registered event subscriptions in the session.
Unregister-Event	Removes one or more of the registered event subscriptions.
Wait-Event	Waits for an event to occur. This cmdlet can wait for a specific event or any event. You can also specify a timeout limiting how long it will wait for the event. The default is to wait forever.
Get-Event	Gets pending unhandled events from the event queue.
Remove-Event	Remove a pending event from the event queue.
New-Event	Called in a script to allow the script to add its own events to the event queue.

C.11.1 The Register-ObjectEvent cmdlet

To set up handlers for .NET events, use the `Register-ObjectEvent` cmdlet. The signature for this cmdlet is as follows.



Within an event handler, a number of predefined variables are available to the scriptblock. These variables make information about the event available to your code.

Example .NET object event subscription

This example shows how to use PowerShell eventing to set up a timer event handler:

- 1 Create the Timer object:

```
$timer = New-Object System.Timers.Timer -Property @{
    Interval = 500; AutoReset = $true
}
```

- 2 Register a scriptblock as the event handler using a scriptblock:

```
Register-ObjectEvent -InputObject $timer -EventName Elapsed `
    -Action {Write-Host "<Timer fired>"}
```

3 Start the timer firing:

```
$timer.Start()
```

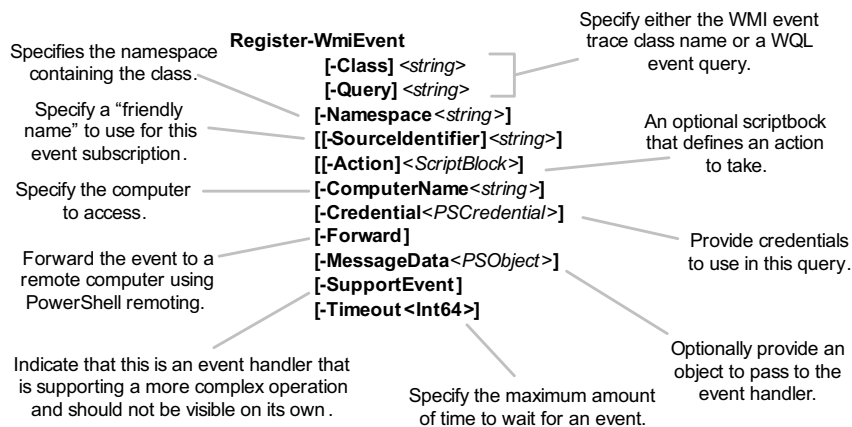
Automatic variables available in the event handler scriptblock

When the event handler scriptblock is executing, it has access to a number of variables containing information about the execution environment. These variables are described in the following table.

Variable	Description
<code>\$Event</code>	Contains an object of type <code>System.Management.Automation.PSEventArgs</code> that represents the event that's being handled. The value of this variable is the same object that the <code>Get-Event</code> cmdlet returns. Therefore, you can use the properties of the <code>\$Event</code> variable, such as <code>\$Event.TimeGenerated</code> in the handler script block.
<code>\$EventSubscriber</code>	Contains the <code>PSEventSubscriber</code> object that represents the event subscriber of the event that's being handled. The value of this variable is the same object that the <code>Get-EventSubscriber</code> cmdlet returns.
<code>\$Sender</code>	Contains the object that generated the event. This variable is a shortcut for <code>\$Event.Sender</code> .
<code>\$SourceEventArgs</code>	Contains an object that represents the first event argument that derives from <code>EventArgs</code> of the event that's being processed. This variable is a shortcut for <code>\$Event.SourceEventArgs</code> .
<code>\$SourceArgs</code>	Contains objects that represent the arguments of the event that's being processed. This variable is a shortcut for <code>\$Event.SourceEventArgs</code> .

C.11.2 The Register-WmiEvent cmdlet

This cmdlet is used to set up event handling for asynchronous WMI events.



This following is an example of how to register a WMI event:

```
Register-WmiEvent -Class Win32_ProcessStopTrace `
    -Action {
        "Process Stop: " +
            $event.SourceEventArgs.NewEvent.ProcessName |
            Out-Host
    }
```

This example registers an event handler that will trigger when a process stops.



A P P E N D I X D

Additional topics

- | | |
|--|---|
| D.1 The restricted language subset of PowerShell 123 | D.4 Alternate remoting mechanisms 142 |
| D.2 World-ready scripting 126 | D.5 Transacted operations in PowerShell 147 |
| D.3 Debugging scripts using low-level tracing 134 | D.6 Summary 161 |

This appendix covers a number of additional topics for PowerShell. These topics are less mainstream than the core contents of the book but are important in the scenarios where they apply, such as building world-ready scripts. The topics covered include a more detailed look at the restricted language subset of PowerShell (previously seen in chapter 10 with module manifests), handling localization tasks for scripts, low-level engine tracing, alternate remoting mechanisms, and the transactions feature introduced in PowerShell version 2.

D.1 THE RESTRICTED LANGUAGE SUBSET OF POWERSHELL

In section 10.2, we looked at how to use module manifests to add metadata to the modules you're creating. When writing a manifest, you use a subset of the PowerShell language called the *restricted language*. The restricted language is a strict subset of the core PowerShell language that's intended to safely allow you to use PowerShell to define data resources. It's used for things that you might conventionally use .INI files or XML for (which is why it's called the data language sometimes). The advantage of using PowerShell for this purpose compared to the alternatives is that you can do everything with one set of tools.

The following PowerShell operators either aren't permitted in a restricted language script or are severely limited:

- The `-f` format operator is disallowed due to vulnerabilities in format string processing in the underlying `System.String.Format` API.
- The range operator (for example, `1 .. 1000`) is disallowed because it can result in unbounded memory allocation.
- Array and string multiplication are limited to producing strings or arrays of length 1024 or less.
- `-match` and `-notmatch` are disallowed because they affect the `$matches` variable.
- `-split`, `-join`, and `-replace` are all disallowed because they can be used to create large strings.
- The `-as` operator is disallowed because casting a string to `[System.IO.File]` results in the creation of a new file, possibly replacing an existing one. (A limited set of casts is permitted, as discussed in the next section.)
- Looping statements are disallowed due to the possibility of unbounded CPU consumption.
- All mechanisms for creating or calling new code—such as defining functions or scriptblocks or using the call operators `.` and `&`—are disallowed.
- Assignment in any form is disallowed.
- All method and property access operations (`.` and `::`) are disallowed.

The following PowerShell operators *are* permitted in a restricted language script:

- The `if` statement.
- Both single line comments (lines starting with `#`) and block comments (enclosed in `<#` and `#>`).
- Pipelines of permitted commands and multiple statements.
- All simple or compound literals: all numeric literals, strings, and hashtables. The set of type literals (and therefore) casts is restricted to strings, any of the numeric types such as `[int]` or `[double]`, `[object]`, `[DBNull]`, `[bool]`, `[char]`, `[sbyte]`, `[byte]` and `[DateTime]`, and arrays of these types.
- Variables can be read. Only a small set of variables are available. The following automatic variables are available by default: `$PsCulture`, `$PsUICulture`, `$True`, `$False`, and `$Null`. In some circumstances, additional variables may be made available. For example, in module manifests, environment variables are permitted.
- Only certain cmdlets are permitted. This set is controlled by how the restricted language is used.

The goal of these restrictions is to allow the safe execution of an untrusted PowerShell script when confined to the restricted language. For example, you want to be able to read the contents of a module manifest without being concerned that your environment will be altered or compromised in the process. Another example is loading message catalogs (see section D.2.1). In the next section, we'll review some of the categories of attacks.

D.1.1 Types of attacks

The limitations in the restricted language are intended to produce a safe subset of the PowerShell language, with the intent of preventing three particular types of attacks. These attacks are briefly described in the following subsections along with how the restricted language mitigates them. (See chapter 21 for more details on PowerShell and security.)

Information disclosure

Information disclosure is a situation where an attacker can access data they aren't intended to have access to. This is why, for example, the environment isn't generally available in restricted language scripts. If the contents of the environment variables were available to external attackers, they might reveal information that would help with further attacks.

Denial of service

Denial of service involves the attacker causing the system to become unavailable to legitimate users and uses. To prevent this kind of attack, the restricted language removes all operations that could cause unbounded resource consumption like CPU use (for example, loops) or memory consumption (for example, the range operator or string multiplication).

Elevation of privilege

In an elevation of privilege attack, the attacker can perform operations and change the state of the system in ways that aren't intended to be permitted.

D.1.2 Limitations on the use of the restricted language

There is one rather glaring omission in our discussion of the restricted language so far: its application in remoting. Although the intent of the restricted language was to provide a safe subset of the language, that goal wasn't achieved as released with PowerShell version 2. In this release, the restricted language can't be considered a security boundary and shouldn't be used as such, especially in remoting scenarios. (See section 13.2.6 for a discussion of security and remoting.) But local application of the language as a data file format is still useful. In the next section, you'll see how these data files are applied in creating world-ready scripts.

D.2 WORLD-READY SCRIPTING

It's no longer enough to only think about local users when creating scripts. Geographic distance isn't that important any more. For example, there are PowerShell user groups all over the world, speaking many languages, all contributing scripts and other content to the community. Likewise, modern businesses are increasingly global both in operations and in their customer base. These organizations need to be able to provide content and services suitably localized for their target audience. For PowerShell to be an active participant in the global community, it has to support script and module localization.

Because it's based on .NET, the PowerShell runtime was Unicode-aware from day 1, so you can write scripts in (almost) any language (although the keywords never change). Likewise, all the .NET globalization features have been available by default. As a result, for compiled cmdlets, all the necessary infrastructure was there through the usual Windows localization tool chain.

NOTE Internationalization and localization techniques are deep topics. It's not practical to try to cover all the background in this book—doing so would require way too much space, and there are already a lot of books dedicated to this topic. All we're trying to do here is to create an awareness of how these techniques are implemented in PowerShell.

But what was missing in PowerShell version 1 were the features needed to write fully localizable scripts. This was remedied in version 2, and we'll go over the features that were added in the rest of this section.

D.2.1 Localized messages and message catalogs

The major feature that was needed was the ability to have localized messages. For compiled code, this is done through resource DLLs. Both the code and the messages used by the code are compiled into binary files for distribution. Scripts aren't compiled, so using resource DLLs to distribute messages wasn't practical. What was needed instead was a script-friendly way of creating message catalogs. To satisfy this need, a number of features were added to the PowerShell version 2 environment. These features are as follows:

- A `data` keyword for defining a section that separates and isolates data definition from executable code.
- Two new predefined variables, `$PSCulture` and `$PSUICulture`. The `$PSCulture` variable contains the name of the UI settings used for culture-specific elements such as date, time, and currency formats. The `$PSUICulture` variable, on the other hand, contains the name of the language used for messages, which includes things like menu items, prompts, and error messages.

- A new cmdlet, `ConvertFrom-StringData`, that converts text strings in a specific format into hashtables that are used as message catalogs. These message catalogs can either be part of the script or be stored in external data files with a `.psd1` extension.
- A second new cmdlet, `Import-LocalizedData`, that imports local-specific data such as message catalogs from the `.psd1` files as required by the `$PSUI-Culture` variable setting.

In the next few sections, we'll look at each of these features and see how they're used to build world-ready scripts. We'll start with the `data` statement.

The data statement

The `data` statement uses the restricted language to create regions containing only data declarations in a script. These data-only blocks isolate the data declaration section in a script and so reduce the amount of script text that needs careful scrutiny for bad behaviors. The syntax of the data statement is shown in figure D.1.



Figure D.1 The PowerShell `data` statement syntax. This statement is used to embed fixed data-definition blocks in a script or module.

This `data` statement can contain any of the permitted language elements described in the previous section. Any disallowed elements result in a terminating error being thrown. The value resulting from evaluating the data block is returned to the caller. By default, only the `ConvertFrom-StringData` cmdlet is permitted, but additional supported commands can be specified using the `-SupportedCommand` parameter with a list of command names. (Module manifest processing uses this mechanism internally to add the `Join-Path` cmdlet to the set of commands allowed in module manifests.)

Defining message catalogs

Error messages and user interaction messages need to be localized, which in turn means the scripts need to be *globalized* by separating the message text from the script and putting it into message catalogs or resource files. In a PowerShell script, these message catalogs are defined using the `data` statement. The following example shows how to use the `data` statement with a hashtable to define a simple message catalog containing two messages:

```

PS (1) > $msgTable = data {
>>     @{
>>         Hello = "Hello, World."
>>         Goodbye = "Goodbye."
>>     }
>> }
>>

```

When you look at the contents of the `$msgTable` variable, you see that it contains a hashtable mapping the message identifiers to the message text:

```

PS (2) > $msgTable

```

Name	Value
-----	-----
Goodbye	Goodbye.
Hello	Hello, World.

Hashtables provide a simple and familiar (to PowerShell users) way to create message catalogs. Unfortunately, this pattern isn't supported by the existing tooling used by localization teams in the Windows world. A more commonly used format is a text file that contains name value pairs, as follows:

```

Hello = Hello, World.
Goodbye = Goodbye.

```

This format is similar to a hashtable (but with less punctuation). To simplify processing this data representation, a cmdlet was added that can convert these string tables into a hashtable. This cmdlet is called `ConvertFrom-StringData`, and its signature is shown in figure D.2.

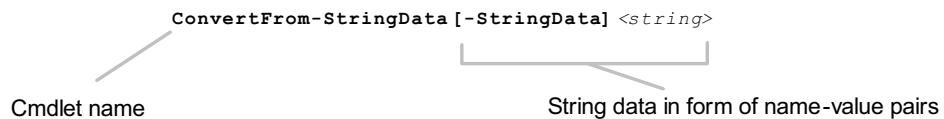


Figure D.2 This cmdlet converts a string in name/values pairs into a hashtable. It can be used in a data block or data file to define a message catalog.

This is a simple cmdlet. It takes a single string (usually a here-string) containing data in the required format, parses that data, and returns the corresponding hashtable. The following example shows how it's used in a `data` block to define a message catalog:

```

PS (1) > $msgTable = data {
>>     ConvertFrom-StringData @'
>>         Hello = Hello, World.
>>         Goodbye = Goodbye.
>>     '@
>> }
>>

```

```
PS (2) > $msgTable
```

Name	Value
-----	-----
Hello	Hello, World.
Goodbye	Goodbye.

As the output shows, the resulting table stored in `$msgTable` is identical to the previous example using a hashtable. For simple cases where you only need to support a couple of languages, it's reasonable to embed both languages in a script. The following function definition illustrates how this might work:

```
function Invoke-Hello
{
    $msgTable = Data {
        if ($PSUICulture -eq "fr-CA")
        {
            ConvertFrom-StringData @"
                HelloString = Salut tout le monde!
"@
        }
        else
        {
            ConvertFrom-StringData @"
                HelloString = Hello world
"@
        }
    }

    "$PSUICulture`: " + $msgTable.HelloString
}
```

To test the function, use the `CurrentCulture` API to temporarily change the culture of the current thread.

NOTE Because PowerShell uses a new thread for each execution, the example needs to be wrapped in a scriptblock to ensure that all statements are executed on the same thread.

Run the function, initially in the current local and then in the `fr-CA` local to see what it produces:

```
PS (1) > & {
>> Invoke-Hello
>> [System.Threading.Thread]::CurrentThread.CurrentUICulture =
>> [System.Globalization.CultureInfo]::CreateSpecificCulture("fr-CA")
>> Invoke-Hello
>> }
>>
en-US: Hello world
fr-CA: Salut tout le monde!
PS (2) >
```

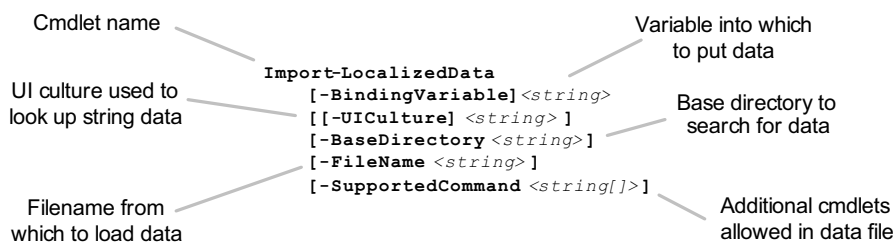


Figure D.3 This cmdlet is used to load a localized data file. The file to load is selected by using the base directory, the current `UICulture`, and the filename.

The first output is in the default local (`en-US`), and the second string is in the language expected in the `fr-CA` locale.

Message catalogs in modules

This in-script approach to localization works for a small number of languages but doesn't scale to larger sets of target languages. To address this need, the `Import-LocalizedData` cmdlet is provided, as shown in figure D.3

This cmdlet is used to load the message catalog from a data file on disk. The path of the file to load is generated by concatenating the base directory (usually the script or module directory), the name specified for the `-UICulture` parameter, and name of the message catalog file. If `-UICulture` isn't specified, then it defaults to the current `UICulture` setting for the process (`$PSUICulture`). The name of the message catalog file has no default so it must be specified, not including the `.psd1` extension.

NOTE The `Import-LocalizedData` cmdlet works with `.psd1` files. This is the same extension that is used for module manifests. In practice, all this extension indicates is that the file contains a PowerShell script that obeys the restricted language conventions. It has no special meaning for either modules or localization—it's a generic PowerShell data representation, similar in concept to the JavaScript Object Notation (JSON) used in many web-based applications.

In chapter 10 (section 10.1), we looked at how modules are organized out on disk. Let's review this and look at the structure in figure D.4. You'll implement this structure in the next example.

In the figure, the module manifest (`localized.psd1`) and script module (`localized.psm1`) are contained in a directory matching the module manifest's name: `localized`. The language-specific message catalogs are stored as subdirectories, one subdirectory for each supported language. The `messages.psd1` file in `en-US` contains

```
ConvertFrom-StringData @"
HelloString = Hello world|
"@
```

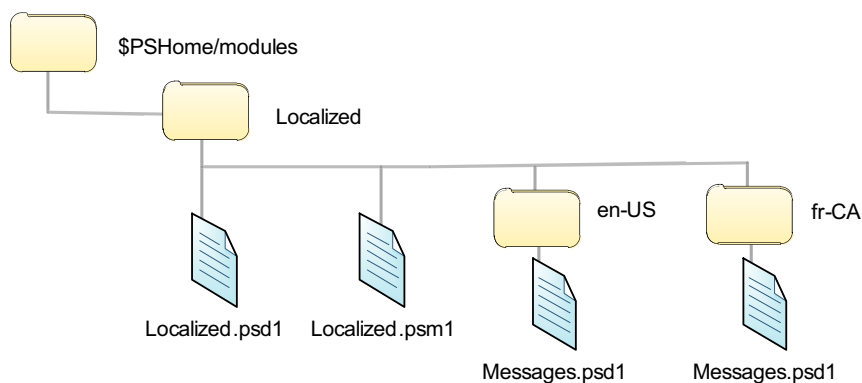


Figure D.4 The organization of the localized module, with the language-specific message catalogs stored in subdirectories of the module

and the `messages.psd1` file in `fr-CA` contains

```

ConvertFrom-StringData @"
HelloString = Salut tout le monde!
"@

```

The `Invoke-Hello` function from the earlier example is defined in the script module file (`localized.psm1`). The updated function looks like this:

```

function Invoke-Hello
{
    Import-LocalizedData -BindingVariable msgTable -File messages
    "$PSUICulture\": " + $msgTable.HelloString
}

```

In this function, the call to `Import-LocalizedData` only specifies the name of the variable to place the data in (`msgTable` with no leading `$`) and the name of the file to load (without a `.psd1` extension). The cmdlet will use the module directory and current UI culture as defaults when generating the name of the data file to load. Now import the module

```
PS (1) > Import-Module localized
```

and run the `Invoke-Hello` function twice, changing the `UICulture` between invocations. As in the earlier example, you're using a scriptblock to ensure that both commands are run on the same thread:

```

PS (2) > & {
>> Invoke-Hello
>> [System.Threading.Thread]::CurrentThread.CurrentUICulture =
>> [System.Globalization.CultureInfo]::CreateSpecificCulture("fr-CA")
>> Invoke-Hello
>> }
>>

```



```
en-US: Hello world
fr-CA: Salut tout le monde!
PS (3) >
```

As expected, the output shows the same properly localized messages as in the first example.

There is one final change that should be made for a production module: moving the call to `Import-LocalizedData` outside the function, saving the data in a module scoped variable:

```
Import-LocalizedData -BindingVariable msgTable -File messages
function Invoke-Hello
{
    "$PSUICulture`: " + $msgTable.HelloString
}
```

This is more efficient than reloading the message catalog each time the function is executed. The extra overhead of reloading the message catalog can become an issue if the catalog is large. The one drawback to only loading the catalog once is that the message language will no longer change when you dynamically change the UI culture. In practice, this is unlikely to be a problem in real applications, and you always have the option of reloading the module with the `-Force` parameter.

With message catalogs out of the way, we'll spend a little time looking at some of the other localization features in PowerShell.

D.2.2 Representation versus presentation

Early in the PowerShell project, a number of people asked, somewhat naively, if we were going to localize keywords and cmdlet names. The answer was a resounding “No!” because doing that would mean that script-sharing would almost be impossible between locales. All the keywords and so on are in the *invariant* locale, which means it doesn't change when you move around.

NOTE Suggesting that the keywords be localized isn't completely naive. As well as a scripting language, PowerShell is also a user interface of sorts. User interfaces are usually localized. Although nothing like this is included by default, using the aliasing mechanism it's possible to create language-specific sets of aliases that would map the command names into the local language.

Keyword invariance is all well and good, but you also have to think about the *data* in your scripts (other than messages)—constants like numbers, dates, and so on. Most of these are also treated as invariants by the language. For example, numbers always use a dot as the decimal indicator, never a comma, regardless of the locale settings. This is how numeric constants are *represented* in a script, which is (and must be) deterministic and fixed. But when you display the results of an expression, the number is displayed using the conventions of the current locale, which may use a comma as the decimal

separator. This is what we mean by *presentation*. If the text is meant to be processed by PowerShell, you should use an invariant representation. If the text is going to be presented on the computer screen, then you should use the presentation conventions.

Most of the time, you never have to worry about this—PowerShell takes care of the details. Sometimes you do have problems when trying to use the presentation form of a string as a data representation. Dates are the most common case where this comes up because PowerShell has no date literal. The way to include a literal date in a script is to write

```
[datetime] '12/25/2009'
```

which happens to be the way dates are represented in the United States. As was the case with numbers, this representation is fixed and will always work regardless of locale. When this value is displayed, it will use the date presentation for the current local. This leads to mistakes when script authors use the local date format in script text and are confused when it doesn't work as expected. Dates must always be written using the US format. You can also see this issue when casting strings into numbers, because casts are always done in the invariant locale. Regardless of the current culture-specific representation of numbers, this

```
[double] "3.14"
```

will always return the number 3.14, even if the local presentation is 3,14.

NOTE This isn't an example of US-centricity on the part of the PowerShell team. The team that developed PowerShell was made up of people from all over the world. Your humble author, as a Canadian, still gets tripped up by the US date format (month/day/year) instead of the (much more logical to me) day/month/year. In the end, we had to pick a common set of conventions, and the US conventions were the most convenient.

To process string data in the current culture, you should use the `Parse()` methods provided on the corresponding types. For example, to parse a floating-point number, you do this:

```
[double]::Parse( "1.23" )
```

Here's an example where the locale is changed during a set of operations. Again, a scriptblock is used to ensure that everything executes on the same thread. Start by parsing a number in the default `en-US` locale. Then set the current culture to `fr-FR`, where the decimal separator is a comma instead of a period, and display the parsed value. What started as 3.14 is now displayed as 3,14. When you call the `parse` method on 3,14 and compare the result to the original value, they are equal:

```
PS (1) > & {  
>> $n1 = [double]::Parse( "3.14")  
>> [System.Threading.Thread]::CurrentThread.CurrentCulture =
```

```
>> [System.Globalization.CultureInfo]::CreateSpecificCulture("fr-FR")
>> $n1
>> $n2 = [double]::Parse( "3,14")
>> $n1 -eq $n2
>> }
>>
3,14
True
```

In the output, you see the displayed value of 3,14. The number parsed as 3.14 in `en-US` is equal to the number parsed as 3,14 in the `fr-FR` locale.

CurrentCulture and CurrentUICulture

As mentioned earlier, Windows splits culture-specific data into two categories. The `CurrentUICulture` setting determines the language used for messages, and the `CurrentCulture` setting controls everything else (dates, numbers, and so on) for everything else. This can be confusing when you're trying to test locale-specific stuff. Make sure you use/check the correct setting to control what you're doing. For message catalogs, this maps into the `$PSUICulture` variable, and everything is determined by the setting in the `$PSCulture` variable.

Error records, CategoryInfo, and FullyQualifiedErrorID

Error messages should always be localized, but PowerShell requires that some portion of the error message be constant across all languages so you can write scripts that make decisions based on this information. In PowerShell error records, this is done through two fields: `CategoryInfo` and `FullyQualifiedErrorID`. These fields should never be localized, so they're safe to use in simple string comparisons, whereas testing the error string will likely fail in a locale where the messages are in a different language.

Although there is quite a bit more to cover about localization and writing world-ready scripts, we've covered the major points and features of PowerShell that exist to facilitate this kind of thing. Let's move on to a new topic: low-level tracing and debugging.

D.3 DEBUGGING SCRIPTS USING LOW-LEVEL TRACING

The next type of tracing we're going to cover is the internal expression tracing facility. This is a much lower-level tracing mechanism than script tracing. It's implemented using a tracing mechanism in the .NET Framework that is designed for use by application developers, not end users. It was originally intended to allow Microsoft to debug PowerShell applications deployed in the field, but it turns out to be useful for script developers as well. But it's not for the faint of heart. It traces the execution of the PowerShell engine at the level of object constructor and method calls. As such, it can be noisy, with a lot of detail that most users prefer to not see.

D.3.1 The Trace-Command cmdlet

Low-level expression tracing is controlled by the `Trace-Command` cmdlet. This cmdlet has a complex set of parameters. A subset of those parameters is shown in figure D.5.

As you can see from the names of the parameters, this is a somewhat developer-centric cmdlet. A listener is a mechanism for capturing the trace events and routing them to a particular location. There are three listeners that you can specify using this cmdlet, as described in table D.1.

Table D.1 The various trace listener options you can specify

Trace listener option	Description
PSHost	When this option is specified, the trace events are written to the console.
Debugger	If a debugger is attached to the PowerShell process, the debugger receives the trace events.
FilePath <string>	When this option is specified, trace records are written to a file for later examination.

You can specify any or all these listeners, and the trace records will be written to all that were specified.

The `-ListenerOption` parameter allows you to control the information that appears in each trace record. The type of information includes things such as the date and time, as well as more complex information such as the process and thread identifiers and the call stack.

The `-Option` parameter controls the type of operation to call. Types of operations include specific .NET activities such as object construction, method calls, and property references as well as more general categories such as `WriteLine`, `Verbose`, and so on. A list of these options can be specified, and all the categories mentioned will be

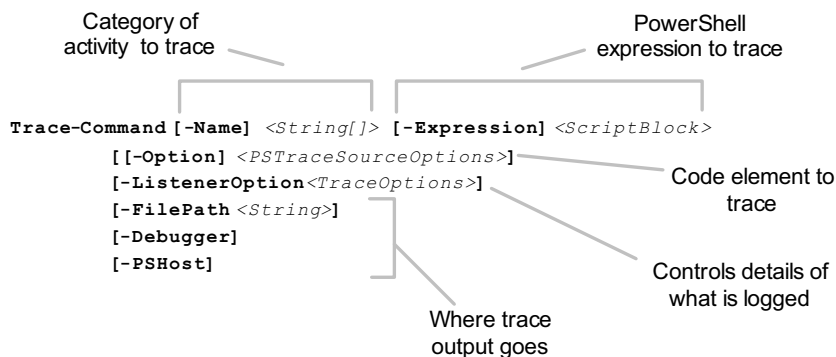


Figure D.5 The `Trace-Command` cmdlet is used to control the low-level tracing features in PowerShell. It can be used to turn on and off trace categories as well as control where the trace messages go.

displayed. Alternatively, if you specify the category [All](#), everything will be shown (this is what is usually used).

The last parameter to discuss is `-Name`. This parameter selects the category of the trace events to display. Although there are a large number of trace categories, only two are of interest to the script author: type conversions and parameter binding. We'll cover these in the next two sections.

D.3.2 Tracing type conversions

First, we'll talk about tracing type conversions. Because automatic type conversion is so important in PowerShell, having a way of seeing exactly what's going on is useful. Let's look at an example that traces a simple conversion, from a string to a number:

```
[int] "123"
```

You're going to trace all the activities, so you specify `-Option all`. And you want the output to go to the console, so you also specify `-PShost`. Here's what it looks like (this is all the output from one command, by the way!):

```
PS (1) > Trace-Command -Option all typeconversion -PShost `
>> {[int] "123"}
>>
DEBUG: TypeConversion Information: 0 : Converting "int" to
"System.Type".
```

This is the first conversion—taking the type literal `[int]` and resolving it to the instance of `System.Type` that represents an integer:

```
DEBUG: TypeConversion Information: 0 :      Original type before
getting BaseObject: "System.String".
DEBUG: TypeConversion Information: 0 :      Original type after
getting BaseObject: "System.String".
DEBUG: TypeConversion Information: 0 :      Standard type
conversion.
DEBUG: TypeConversion Information: 0 :      Converting
integer to System.Enum.
DEBUG: TypeConversion Information: 0 :      Type conversion
from string.
DEBUG: TypeConversion Information: 0 :      Conversion to
System.Type
```

You're done with step 1; you now have the type you need:

```
DEBUG: TypeConversion Information: 0 :      The conversion is a
standard conversion. No custom type conversion will be
attempted.
DEBUG: TypeConversion Information: 0 : Converting "123" to
"System.Int32".
```

The next step is to figure out how to convert the string “123” into an integer:

```
DEBUG: TypeConversion Information: 0 :      Original type before
getting BaseObject: "System.String".
DEBUG: TypeConversion Information: 0 :      Original type after
```

```

getting BaseObject: "System.String".
DEBUG: TypeConversion Information: 0 :      Standard type
conversion.
DEBUG: TypeConversion Information: 0 :      Converting
integer to System.Enum.
DEBUG: TypeConversion Information: 0 :      Type conversion
from string.
DEBUG: TypeConversion Information: 0 :      Converting to
integer.
DEBUG: TypeConversion Information: 0 :      The conversion is a
standard conversion. No custom type conversion will be
attempted.

```

This is a standard .NET type conversion, so no special steps are needed. The conversion is performed, and finally you get the result as a number:

```

123
PS (2) >

```

Did you follow all that? Remember what I said about this type of tracing being verbose? I wasn't kidding.

Let's look at a second example. Again, you'll trace everything and output to the console. This time, you'll trace casting a string into an `[xml]`. You'll also use the `-ListenerOption` parameter to say that you want to include the timestamp in the option. Here you go:

```

PS (6) > Trace-Command -Option all typeconversion -PSHost `
>> -listen timestamp `
>> { [xml] '<h>Hi</h>' }
>>
DEBUG: TypeConversion Information: 0 : Converting "xml" to
"System.Type".
DEBUG: Timestamp=5536598202692

```

Again, the first step is to resolve the type literal. Note that timestamp information is now being output as you requested:

```

DEBUG: TypeConversion Information: 0 :      Original type before
getting BaseObject: "System.String".
DEBUG: Timestamp=5536598216733
DEBUG: TypeConversion Information: 0 :      Original type after
getting BaseObject: "System.String".
DEBUG: Timestamp=5536598230212
DEBUG: TypeConversion Information: 0 :      Standard type
conversion.
DEBUG: Timestamp=5536598243271
DEBUG: TypeConversion Information: 0 :      Converting
integer to System.Enum.
DEBUG: Timestamp=5536598255383
DEBUG: TypeConversion Information: 0 :      Type conversion
from string.
DEBUG: Timestamp=5536598267714
DEBUG: TypeConversion Information: 0 :      Conversion to
System.Type

```

```

DEBUG: Timestamp=5536598279950
DEBUG: TypeConversion Information: 0 :      The conversion is a
standard conversion. No custom type conversion will be
attempted.
DEBUG: Timestamp=5536598292383
DEBUG: TypeConversion Information: 0 : Converting "<h>Hi</h>" to
"System.Xml.XmlDocument".

```

This tells you what the final type of the object will be:

```

DEBUG: Timestamp=5536598308660
DEBUG: TypeConversion Information: 0 :      Original type before
getting BaseObject: "System.String".
DEBUG: Timestamp=5536598321106
DEBUG: TypeConversion Information: 0 :      Original type after
getting BaseObject: "System.String".
DEBUG: Timestamp=5536598334410
DEBUG: TypeConversion Information: 0 :      Standard type
conversion.
DEBUG: Timestamp=5536598347058
DEBUG: TypeConversion Information: 0 :      Converting to
XmlDocument.
DEBUG: Timestamp=5536598382014
DEBUG: TypeConversion Information: 0 :      Standard type
conversion to XmlDocument.
DEBUG: Timestamp=5536598396299
DEBUG: TypeConversion Information: 0 :      The conversion is a
standard conversion. No custom type conversion will be
attempted.
DEBUG: Timestamp=5536598409092

```

```

h
-
Hi

```

Finally, the XML object is displayed. Now let's look at tracing parameter binding.

D.3.3 Tracing parameter binding

The other category of trace information that is interesting to the script user is parameter binding. This allows you to see how the parameters are being bound to a cmdlet. Let's look at an example that is the simple command

```
"c:\\" | Get-Item
```

In this example, `Get-Item` takes its mandatory parameter `-Path` from the pipeline. When you run the command, you see that the set of parameters is mostly the same as in the previous section, except that you're now using the `parameterbinding` trace category:

```

PS (7) > Trace-Command -Option all parameterbinding -PSHost `
>> { "c:\\" | get-item }
>>

```

The first step is to go through each of the command-line parameter binding steps—named parameters, positional parameters, and finally dynamic parameters (see chapter 2 for more information for each of these steps):

```
DEBUG: ParameterBinding Information: 0 : BIND NAMED cmd line
args [Get-Item]
DEBUG: ParameterBinding Information: 0 : BIND POSITIONAL cmd
line args [Get-Item]
DEBUG: ParameterBinding Information: 0 : BIND cmd line args to
DYNAMIC parameters.
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER
CHECK on cmdlet [Get-Item]
```

At this point, the parameter binder is checking to see if there are any unbound mandatory parameters that can't be bound to input from the pipeline. If this were the case, a terminating error would occur here. Because this isn't the case in this example, the binding process continues:

```
DEBUG: ParameterBinding Information: 0 : CALLING BeginProcessing
DEBUG: ParameterBinding Information: 0 : BIND PIPELINE object to
parameters: [Get-Item]
DEBUG: ParameterBinding Information: 0 : PIPELINE object
TYPE = [System.String]
```

You have an object from the pipeline to bind:

```
DEBUG: ParameterBinding Information: 0 : RESTORING pipeline
parameter's original values
```

First, restore all the parameters that can take pipeline input to their default state, because not all of them may be bound from this object:

```
DEBUG: ParameterBinding Information: 0 : Parameter [Path]
PIPELINE INPUT ValueFromPipeline NO COERCION
```

This is the first step in matching the parameter; if the parameter type exactly matches, then binding proceeds immediately:

```
DEBUG: ParameterBinding Information: 0 : BIND arg [c:\] to
parameter [Path]
DEBUG: ParameterBinding Information: 0 : Binding
collection parameter Path: argument type [String], parameter
type [System.String[]], collection type Array, element type
[System.String], no coerceElementType
```

In this case, the target type is a collection. But the element type of the collection matches the pipeline object, so the interpreter will wrap the pipeline object in an array so the binding can succeed:

```
DEBUG: ParameterBinding Information: 0 : Creating array
with element type [System.String] and 1 elements
DEBUG: ParameterBinding Information: 0 : Argument type
String is not IList, treating this as scalar
DEBUG: ParameterBinding Information: 0 : Adding scalar
```



```

element of type String to array position 0
DEBUG: ParameterBinding Information: 0 :      BIND arg
[System.String[]] to param [Path] SUCCESSFUL

```

At this point, you've bound the `-Path` parameter. Let's check the remaining parameters that can take their values from the pipeline:

```

DEBUG: ParameterBinding Information: 0 :      Parameter
[Credential] PIPELINE INPUT ValueFromPipelineByPropertyName NO
COERCION
DEBUG: ParameterBinding Information: 0 :      Parameter
[Credential] PIPELINE INPUT ValueFromPipelineByPropertyName WITH
COERCION

```

Nothing is bound at this point, so the last thing to do is check to make sure that all mandatory parameters for this cmdlet are now bound. If there were an unbound mandatory parameter, a nonfatal error would be generated and an error record would be written to the output pipe. Note how this is different from the command-line parameters. In that case, it's a fatal error; there's no way to continue. For the pipeline parameter, even if the current object doesn't result in all parameters being bound successfully, the next pipeline object may succeed. This is why pipeline binding failures are non-terminating and command-line binding failures are terminating:

```

DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER
CHECK on cmdlet [Get-Item]

```

Finally, now that all parameters are bound, the cmdlet's `ProcessRecord` and `EndProcessing` clauses are executed:

```

DEBUG: ParameterBinding Information: 0 : CALLING ProcessRecord
DEBUG: ParameterBinding Information: 0 : CALLING EndProcessing

```

Directory:

Mode	LastWriteTime	Length	Name
d--hs	6/9/2006 2:30 AM		C:\

Again, this is a verbose tracing mechanism, but it shows you the binding algorithm in great detail. If you're having a problem understanding why a pipeline is exhibiting some unexpected behavior, this is the way to see what's happening. You can also combine the two tracing mechanisms to see even more detail of what's going on.

You can try this with a user-defined function to see that this mechanism works with functions as well as cmdlets. First let's define a function:

```
PS (8) > function foo ([int] $x) {$x}
```

Now trace its execution:

```

PS (9) > Trace-Command -Option all parameterbinding,
>> typeconversion -PSHost {foo "123"}
>>
DEBUG: ParameterBinding Information: 0 : POSITIONAL parameter
[x] found for arg []

```

You see the parameter binding trace messages for positional parameter binding:

```
DEBUG: ParameterBinding Information: 0 : BIND arg [123] to
parameter [x]
DEBUG: ParameterBinding Information: 0 :      Executing DATA
GENERATION metadata:
[System.Management.Automation.ArgumentTypeConverterAttribute]
DEBUG: TypeConversion Information: 0 :      Converting "123"
to "System.Int32".
```

And now you see the type conversion messages:

```
DEBUG: TypeConversion Information: 0 :      Original type
before getting BaseObject: "System.String".
DEBUG: TypeConversion Information: 0 :      Original type
after getting BaseObject: "System.String".
DEBUG: TypeConversion Information: 0 :      Standard type
conversion.
DEBUG: TypeConversion Information: 0 :
Converting integer to System.Enum.
DEBUG: TypeConversion Information: 0 :      Type
conversion from string.
DEBUG: TypeConversion Information: 0 :
Converting to integer.
DEBUG: TypeConversion Information: 0 :      The
conversion is a standard conversion. No custom type conversion
will be attempted.
DEBUG: ParameterBinding Information: 0 :      result returned
from DATA GENERATION: 123
DEBUG: ParameterBinding Information: 0 :      COERCE arg type
[System.Int32] to [System.Int32]
DEBUG: ParameterBinding Information: 0 :      Parameter and
arg types the same.
DEBUG: TypeConversion Information: 0 :      Converting "123" to
"System.Int32".
DEBUG: TypeConversion Information: 0 :      Result type is
assignable from value to convert's type
```

Finally, the binding process is complete, and the script is executed:

```
DEBUG: ParameterBinding Information: 0 :      BIND arg [123] to
param [x] SUCCESSFUL
123
PS (10) >
```

In summary, the [Trace-Command](#) cmdlet provides a mechanism for tracing the execution of command in PowerShell in a low-level and detailed way. Sometimes this mechanism can be the only way to debug the behavior of a script. It isn't a tool intended for the casual user; it requires considerable sophistication to interpret the output of these trace logs. For the casual user, the most effective way to go about it is to use the [-FileLog](#) parameter to create a trace log that can then be analyzed by a developer.

D.4 ALTERNATE REMOTING MECHANISMS

In chapters 12 and 13, we explored PowerShell's built-in remoting capabilities in depth. It should be clear from those chapters that PowerShell remoting is both powerful and flexible. Even so, it does have one rather obvious drawback: it requires PowerShell version 2 or greater to be present on both ends of every connection.

NOTE The PowerShell Remoting Protocol (see section 13.1.1) contains version information so that future versions of PowerShell will be able to talk to earlier versions although perhaps with a reduced feature set. When PowerShell version 2 is available on all the machines you need to target, distributed management becomes relatively easy.

Because it isn't possible to wave a magic wand and put PowerShell remoting everywhere you need it to be, in this section we'll look at alternative technologies that can be used to perform remote operations in environments where PowerShell remoting isn't available yet.

D.4.1 Remoting with the PsExec utility

Assuming you're still working in a primarily Windows world, one of this simplest and easiest-to-use remoting tools is the **PsExec** program from [SysInternals](http://sysinternals.com) (now part of Microsoft; see <http://microsoft.com/sysinternals>). The **PsExec** program has the extremely nice characteristic that it doesn't require the user to install any client software on the remote machine. This utility can be used to run almost any program on another Windows machine. For example, running **hostname.exe** on another machine looks like this:

```
PS (1) > psexec \\brucepayx61 -u redmond\brucepay hostname
PsExec v1.96 - Execute processes remotely
Copyright (C) 2001-2009 Mark Russinovich
Sysinternals - www.sysinternals.com
Password:
brucepayx61
hostname exited on brucepayx61 with error code 0.
PS (2) >
```

This utility can also be used to run PowerShell. The next example invokes the **powershell.exe** program on the remote host, passing it a simple expression to evaluate:

```
PS (3) > psexec \\brucepayx61 -u redmond\brucepay powershell 2+2

PsExec v1.96 - Execute processes remotely
Copyright (C) 2001-2009 Mark Russinovich
Sysinternals - www.sysinternals.com

Password:

4
powershell exited on brucepayx61 with error code 0.
PS (4) >
```

The expression is passed to the remote end, where it's (eventually) evaluated. Unfortunately, using `psexec.exe` as a non-interactive command is fairly painful because it tends to take a long time to execute.

NOTE One ideal application of non-interactive `psexec.exe` would be to bootstrap more efficient remoting capabilities. For example, if PowerShell remoting isn't enabled on the target machine, you could use `psexec.exe` to enable it by running `Enable-PSRemoting -Force`.

A more effective approach is to use `psexec.exe` to set up an interactive session. To start an interactive session on the target computer, run the following command:

```
psexec \\brucepayx61 -u redmond\brucepay powershell -file -
```

Specifying `-` as the argument to `-File` tells `powershell.exe` to go into interactive mode, reading from standard input instead of from the console. This looks like

```
PS (4) > psexec \\brucepayx61 -u redmond\brucepay powershell -file -
```

```
PsExec v1.96 - Execute processes remotely
Copyright (C) 2001-2009 Mark Russinovich
Sysinternals - www.sysinternals.com
Password:
```

```
PS C:\Windows\system32> hostname
brucepayx61
PS C:\Windows\system32> 2+2
4
PS C:\Windows\system32> exit
powershell exited on brucepayx61 with error code 0.
```

Calling `exit` in the PowerShell also results in the termination of the `psexec.exe` session, returning you to your local session.

NOTE A problem with version 1 of PowerShell resulted in prompts not being issued when the executable was run with commands being read from standard input. PowerShell V2 added the `-File` parameter, which allows `powershell.exe` to directly invoke scripts. It also lets PowerShell be used interactively from tools like `psexec.exe` and `ssh`. If you specify `-` as the file name, PowerShell will read commands from standard input with prompting.

There are many more options to `psexec.exe`, but exploring them is left as an exercise for you. Instead, we'll move on the next tool on the list.

D.4.2 Remoting with `ssh`

The secure shell, commonly known as `ssh`, is a remote shell protocol based on Secure Sockets Layer (SSL) allowing for a secure remote shell connection to another computer.

It was developed as a replacement to the existing telnet facility, which didn't provide a secure channel. The protocol suite also includes secure file transfer capabilities.

This protocol is ubiquitously available on computers running UNIX and similar operating systems like Linux. Unfortunately, for a variety of reasons, no `ssh` facility, either client or server, is included as part of Windows. But there are a number of commercial and open-source implementations of `ssh` for Windows, Cygwin being the most common (<http://cygwin.org>). Cygwin is a port of the GNU UNIX utilities to Windows-based operating systems and includes an `ssh` server (also called an *ssh daemon* or `sshd`).

To use this protocol on a Windows server, you need to install one of the `ssh` services on the target computer. Once the `ssh` service is set up, you can use the same `-file` - convention used with `psexec` to start a PowerShell session from a Linux/UNIX machine. If you're working in a heterogeneous enterprise environment, depending on how things are set up, this can be useful.

The next two mechanisms we'll look at aren't specifically shell-oriented. We'll start with how you can use web services from PowerShell.

D.4.3 Using web services

In section 13.1.1, we discussed how PowerShell remoting is built on top of web service protocols. A web service, in turn, is an XML-based remote procedure call mechanism that exchanges data over a network, including over the internet. The web service provider is usually hosted by a web server, and the client uses `HTTP` to talk to it.

In version 1, there was no built-in support for web services in PowerShell. Version 2 added some client support for performing remote operations with the `New-WebServiceProxy` cmdlet. This cmdlet lets you connect to a web service, whereupon it creates a web service proxy object in PowerShell. This proxy object is a .NET object created in the PowerShell session that takes care of the details of remote execution.

NOTE This is similar to how implicit remoting works—local proxies are created to execute remote commands. The types of proxies are quite different. Implicit remoting proxies commands, and web services proxy objects and method calls.

Let's experiment with this cmdlet by using the `New-WebServiceProxy` cmdlet to access a web service that provides weather data. To access this service, first you need to create the proxy object using the URL of the service. You assign this URL to a variable:

```
PS (1) > $serviceUri = "http://www.webservices.net/uszip.aspx?WSDL"
```

Now, run the command to create the web service proxy:

```
PS (2) > $zipClass = New-WebServiceProxy -Uri $serviceUri `
>> -Namespace WebServiceProxy -Class ZipClass
>>
```

You put the instance of the proxy object into `$zipClass` so you can use it later.

Let's pause for a minute and discuss what exactly happened here. The [New-WebServiceProxy](#) cmdlet retrieves a description of the remote service interfaces expressed in Web Service Description Language (WSDL) and then uses this description to construct the proxy object. The end result of the operation is a .NET class that you can use to call the remote methods exposed by the web service.

Let's use [Get-Member](#) to look at what you got. To shorten the output, let's look at the methods that start with [GetInfo](#) and see the definitions of the methods:

```
PS (3) > $zipClass | Get-Member -Type method GetInfo* |
>> select definition
>>

Definition
-----
System.Xml.XmlNode GetInfoByAreaCode(string USAreaCode)
System.Void GetInfoByAreaCodeAsync(string USAreaCode), System...
System.Xml.XmlNode GetInfoByCity(string USCity)
System.Void GetInfoByCityAsync(string USCity), System.Void Ge...
System.Xml.XmlNode GetInfoByState(string USState)
System.Void GetInfoByStateAsync(string USState), System.Void ...
System.Xml.XmlNode GetInfoByZIP(string USZip)
System.Void GetInfoByZIPAsync(string USZip), System.Void GetI...
```

This output shows a number of ways you can use the generated proxy class to access information. You can use the [GetInfoByZip\(\)](#) method to get weather information for a particular zip code:

```
PS (4) > $info = $zipClass.GetInfoByZip(98052)
```

The method call completes successfully (assuming there are no network issues) so you can look at the type of the object returned:

```
PS (5) > $info | Get-Member -Type property

TypeName: System.Xml.XmlElement

Name MemberType Definition
----
Table Property System.Xml.XmlElement Table {get;}
xmlns Property System.String xmlns {get;set;}
```

The call returns an XML element that allows you to use the built-in XML support in PowerShell to extract the data.

As more things move to the cloud, a lot of information is being made available through web services, and the [New-WebServiceProxy](#) cmdlet allows you to access that data in a fairly painless way.

Web services provide a strongly typed structured layer on top of the basic web protocols and formats. Although web services are becoming more common, plain old HTTP and simple resource-based protocols are already everywhere. In the next section, we'll look at how you can use these protocols from PowerShell to access remote information.

D.4.4 The .NET web client interfaces

PowerShell doesn't have native support for HTTP, but because it's built on top of .NET, it can use the networking capabilities built into .NET. You saw this way back in chapter 1, when we looked at this example:

```
([xml](New-Object System.Net.WebClient).DownloadString(
"http://blogs.msdn.com/powershell/rss.aspx"
)).rss.channel.item | Format-Table title,link
```

This fragment of script uses the .NET `WebClient` APIs to download an XML document (a *resource*) from a remote host and render it as a table. This works because the target document is already encoded in a relatively simple XML format called Really Simple Syndication (RSS). Because it's XML, PowerShell can deal with it in a natural way using the XML type adapter (see section 16.4):

```
PS (1) > ([xml](New-Object System.Net.WebClient).DownloadString(
>> "http://blogs.msdn.com/powershell/rss.aspx"))
>>
```

```
xml                                xml-stylesheet            rss
---                                -
version="1.0" encodi... type="text/xsl" hre... rss
```

You can make the call to the `DownloadString` method reusable by turning it into a parameterized function similar to the `wget` function found in other environments:

```
function Get-WebPage
{
    [cmdletbinding()]
    param(
        [parameter(mandatory=$true)]
        [System.URI] $uri
    )
    $webClient = New-Object System.Net.WebClient
    $page = $webClient.DownloadString( $uri )
    $webClient.Dispose()
    $page
}
```

This simple function allows you to point to a web page and get the contents back as a string.

NOTE Adding all the missing features you might find on a `wget` command is left as an exercise for you. Alternatively, grab one of the many `wget` implementations that are floating around on the web. Remember—good programmers write good code, great programmers steal.

There is one more built-in remoting facility that you can use. Some .NET classes can do remote operations on their own without requiring PowerShell remoting. That's the last thing we'll look at in this section.

D.4.5 Remotable .NET classes

Some of the classes in .NET support their own remoting mechanisms. For example, the `RegistryKey` class allows you to access the Registry hive on a remote machine. Let's try this using the `OpenRemoteBaseKey()` static method. You access the `LocalMachine` key on a remote computer:

```
PS (1) > $reg = [Microsoft.Win32.RegistryKey]::OpenRemoteBaseKey(  
>> 'LocalMachine', "brucepayX61")  
>>
```

This returns (assuming there are no permission or networking issues) an instance of the `RegistryKey` class. You can use this instance to do things like get a list of subkey names:

```
PS (2) > $reg.GetSubKeyNames()  
BCD0000000  
HARDWARE  
SAM  
SECURITY  
SOFTWARE  
SYSTEM
```

You see in the output from the method call a list of the subkeys. Finally, you need to close the connection:

```
PS (3) > $reg.Close()
```

This approach can be useful when needed, but it's pretty much a programmer's experience when it gets to this level.

D.5 TRANSACTED OPERATIONS IN POWERSHELL

One more feature introduced in PowerShell V2 isn't cover in the main text of the book: transactions. In this section, we'll introduce the idea of transactions and what they're good for, talk about the PowerShell approach to using transactions, and discuss what you can do with transactions in the V2 release.

D.5.1 What is a transaction?

The idea behind a transaction is that you want to be able to treat a set of operations in such a way that either all the operations are performed successfully or none of them are performed at all. The word *transaction* comes from the business world, where financial transactions are performed all the time. Consider a simple financial transaction—moving money from your savings account to your checking account. This transaction consists of two operations: removing the money from the savings account (debiting) and adding this money to the checking account (crediting). For your overall balance to be correct, both of these operations must succeed. If the debit operation succeeds but the credit operation fails, then you'll appear to have lost the debited money. The way to deal with this in transactions is to mark the debit opera-

tion as incomplete. Then, if the credit operation fails, the incomplete debit operation can be rolled back; but if it succeeds, the debit operation is marked as complete or committed. This approach provides for *atomicity*—either everything succeeds or nothing does. In fact the characteristics of a transaction are described by the acronym ACID, which stands for atomicity, consistency, isolation, durability. We already covered atomicity (all or nothing).

Consistency is the requirement that whatever is being updated should always be in a consistent state. Atomicity is one of the characteristics that help with this.

Isolation implies that each transaction should neither affect nor be affected by any other transaction. While the operation in the earlier transaction is in progress, no other transaction can update those accounts until the current transaction is completed (all operations in the transaction are committed, or all operations are rolled back).

The final characteristic is *durability*. This means that once a transaction has been committed, that state won't be lost (for example, by a power outage or a disk failure). Obviously this is easier said than done, but techniques exist to guarantee durability to a sufficient degree.

Compensations versus rollback

Transactions may have come from the financial world, but the concepts have been applied in a lot of other fields. Unfortunately, whereas updating a database in a consistent way is relatively easy, in other fields, many operations can't be rolled back. For example, you can't unlaunch a missile. To deal with these situations, an alternate approach to provide a sufficient degree of consistency is to use *compensating actions*.

Instead of rolling back the original operation, you do something else to try to get the system back to an equivalent consistent state. In the missile example, if the missile is heading toward the wrong target, this might involve destroying that missile and then moving a new missile to the launching pad. The state of the system isn't identical to the original state, but it's still consistent. Now let's see how you can apply transactions in systems management.

Transactions in the management world

Transactions are obviously useful in system administrations: the ability to update a database in such a way that it's always consistent and that all changes are isolated and durable is the Holy Grail for many researchers working in the management space. Unfortunately, it's hard to achieve due to the heterogeneous nature of management operations. Many things can't be unlaunched—deleted files, reformatted disks, overwriting a file, and so on. Before you can perform transacted operations, you need to be able to roll back or compensate for each of these elements.

D.5.2 Transactions in PowerShell

In this section, we'll look at how transactions apply in PowerShell. As a caveat, unlike remoting or eventing, transactions aren't a fully elaborated feature in V2. This means

that although the infrastructure for transactions is in place, there isn't much you can do with them out of the box. That said, because the infrastructure pieces are in place, it's possible for third-party cmdlet authors to create transacted cmdlets and providers.

The transaction cmdlets

Transactions are handled in PowerShell using a set of cmdlets for managing the transaction and then incrementing the cmdlets that need to participate in a transaction. These cmdlets are shown in table D.2.

Table D.2 Cmdlets for working with transactions

Cmdlet	Description
Start-Transaction	Creates a new transaction context, represented by an instance of the PSTransaction object
Get-Transaction	Gets the active transaction context
Use-Transaction	Runs a scriptblock in the current transaction context
Complete-Transaction	Commits all the operations in the transaction context
Undo-Transaction	Rolls back all the operations in the transaction context

These cmdlets, along with any transacted operations, are called in a specific sequence, illustrated in figure D.6.

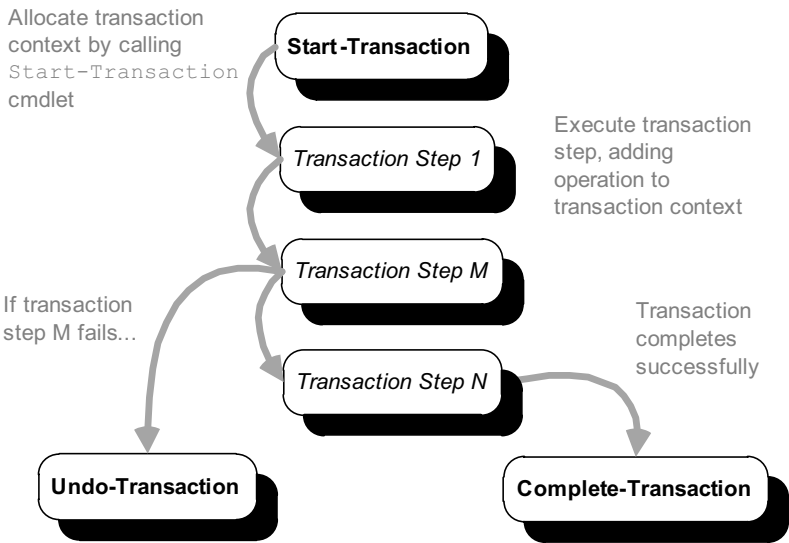


Figure D.6 This figure shows the steps in a transacted PowerShell script. The first step is to allocate a transaction context and then pass that to each operation in the sequence. At any point, if a step fails, the recorded operations are rolled back, restoring the system to the initial state. If all operations complete successfully, the operations are committed by calling Complete-Transaction.

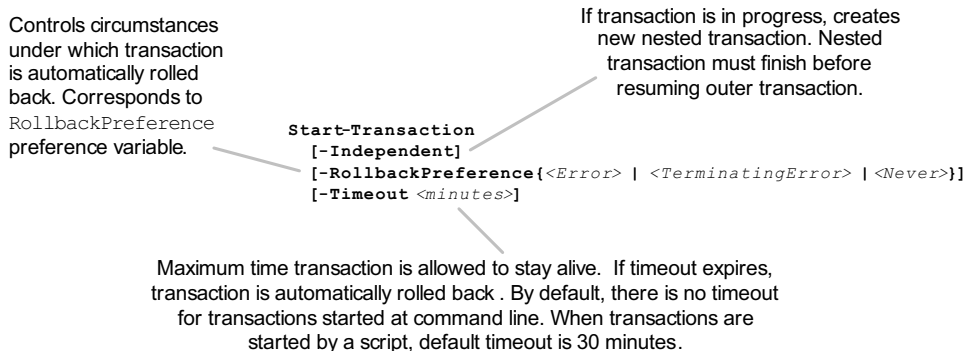


Figure D.7 The `Start-Transaction` cmdlet is used to start a transacted operation. You can specify a number of characteristics on this transaction—the default rollback behavior, the maximum time to live before the transaction is rolled back, and whether a nested transaction should be created if there is already a transaction in progress.

The transacted sequence begins with calling `Start-Transaction`, shown in figure D.7

The `Start-Transaction` cmdlet allocates the *transaction context* object. This object is used to keep track of all the operations in the transaction. This is needed so you know what has to be either committed or rolled back. Then the transaction ends.

When the transaction has started, operations can be added to the transaction context. For a command to be added to the transaction, it must be executed specifying the `-UseTransaction` parameter. If the parameter isn't specified, then the command won't be committed or rolled back in the transaction.

Using commands in a transaction

Obviously, only commands that have the `-UseTransaction` parameter can take part in a transaction. Commands that support transactions can be identified by the `SupportsTransactions` property on the `CommandMetadata` object for that command. The following command displays all the cmdlets that have this attribute:

```
PS (1) > $(
>>   foreach ($cmd in `
>>     Get-Command -CommandType Function,Cmdlet )
>>   {
>>     $commandMetadata =
>>       [System.Management.Automation.CommandMetadata] `
>>         $cmd
>>
>>     if ($commandMetadata.SupportsTransactions)
>>     {
>>       $cmd
>>     }
>>   }
>> ) | Format-Wide -AutoSize Name
>>
```

Add-Content	Clear-Content	Clear-Item
Clear-ItemProperty	Convert-Path	Copy-Item
Copy-ItemProperty	Get-Acl	Get-ChildItem
Get-Content	Get-Item	Get-ItemProperty
Get-Location	Get-PSDrive	Invoke-Item
Join-Path	mkdir	Move-Item
Move-ItemProperty	New-Item	New-ItemProperty
New-PSDrive	Pop-Location	Push-Location
Remove-Item	Remove-ItemProperty	Remove-PSDrive
Rename-Item	Rename-ItemProperty	Resolve-Path
Set-Acl	Set-Content	Set-Item
Set-ItemProperty	Set-Location	Split-Path
Test-Path	Use-Transaction	

Looking at this list, you see that not many commands support transactions. Most of the commands that do support them are provider-based commands that use PowerShell's provider model to access data stores. But this doesn't mean you can only perform transacted operations with providers. You can also execute transacted operations that involve transacted .NET objects through the use of the [Use-Transaction](#) cmdlet. PowerShell includes a demo transacted string class that was specifically to support experimenting with transactions. You'll try some experiments with this class in the next section.

A transacted script example

In this section, you'll work through an example that illustrates using transacted .NET objects in the context of PowerShell's transaction infrastructure. PowerShell includes a class designed to allow you to safely experiment with transactions. This class is named

```
Microsoft.PowerShell.Commands.Management.TransactedString
```

The [TransactedString](#) class wraps a string and supports a couple of methods for manipulating the contents of the string: [Append\(\)](#) and [Remove\(\)](#). These methods can take part in a transacted operation. The [TransactedString](#) class is an example of a transaction *resource manager* and derives from

```
System.Transactions.IEnlistmentNotification
```

A transaction resource manager is responsible for managing its internal state throughout the transaction. In the case of the [TransactedString](#) class, it holds two strings: the original and a copy that the methods update. This class, in turn, checks its environment to see if there is an active transaction it should enlist in. Let's see how this works in an example. As you saw in figure D.7, the first step in any transacted operation is to establish the transaction context by calling [Start-Transaction](#):

```
PS (1) > Start-Transaction
```

Suggestion [1,Transactions]: Once a transaction is started, only commands that get called with the `-UseTransaction` flag become part of that transaction.

NOTE Notice the suggestion/warning you get when you execute this command interactively. It's warning you that only commands executed with the `-UseTransaction` flag participate in the transaction. Don't call `Start-Transaction` followed by `Remove-Item *` and hope to get everything you deleted back by calling `Undo-Transaction`. Transactions aren't magic. They just look a little like magic.

Now that you've started your transaction, you create a couple of `TransactedString` objects to use in the context of this transaction:

```
PS (2) > $tsType =  
>> "Microsoft.PowerShell.Commands.Management.TransactedString"  
>>  
PS (3) > $ts1 = New-Object $tsType "First"  
PS (4) > $ts2 = New-Object $tsType "Second"
```

You use the `$tsType` variable to hold the type name to simplify things. Ok—you have your transacted objects. Next, modify them in the context of the transaction using the `Use-Transaction` cmdlet. This cmdlet sets the *ambient* (default) transaction for all operations executed in the scriptblock. This is how the transacted objects know what transaction to participate in:

```
PS (5) > Use-Transaction -UseTransaction -TransactedScript {  
>>     $ts1.Append(" string")  
>>     $ts2.Remove(0, 2)  
>> }  
>>
```

Suggestion [2,Transactions]: The Use-Transaction cmdlet is intended for scripting of transaction-enabled .NET objects. Its ScriptBlock should contain nothing else.

NOTE And again, because you're entering the command interactively, you get a suggestion on how to use the cmdlet effectively.

You've made some changes in the context of a transaction. Let's look at the strings outside the transaction:

```
PS (6) > $ts1.ToString()  
First  
PS (7) > $ts2.ToString()  
Second
```

The strings appear to be unchanged. Note that the transaction you started is still active; you haven't told the `TransactedString` objects to use it. Let's see what happens in the transaction context:

```
PS (8) > Use-Transaction -UseTransaction -TransactedScript {  
>>     $ts1.ToString()  
>>     $ts2.ToString()  
>> }
```

```
>>
First string
cond
Suggestion [2,Transactions]: The Use-Transaction cmdlet is intended for scripting of transaction-enabled .NET objects. Its ScriptBlock should contain nothing else.
```

This time you see the modified strings instead of the original. Now let's terminate the transaction and undo the modifications. You do this by calling [Undo-Transaction](#):

```
PS (9) > Undo-Transaction
PS (10) > $ts1.ToString()
First
PS (11) > $ts2.ToString()
Second
```

The output shows that the changes you've made have been discarded and the original strings preserved. Let's start a new sequence of transacted operations on the same objects:

```
PS (12) > Start-Transaction

Suggestion [1,Transactions]: Once a transaction is started, only commands that get called with the -UseTransaction flag become part of that transaction.
```

Make the same modifications you made in the previous example:

```
PS (13) > Use-Transaction -UseTransaction -TransactedScript {
>>     $ts1.Append(" string")
>>     $ts2.Remove(0, 2)
>> }
>>
```

```
Suggestion [2,Transactions]: The Use-Transaction cmdlet is intended for scripting of transaction-enabled .NET objects. Its ScriptBlock should contain nothing else.
```

As before, outside the context of the transaction, the strings are unchanged:

```
PS (14) > $ts1.ToString()
First
PS (15) > $ts2.ToString()
Second
```

This time, when you complete the transaction, you call [Complete-Transaction](#) to commit the changes instead of undoing them:

```
PS (16) > Complete-Transaction
Now look at the strings:
PS (17) > $ts1.ToString()
First string
PS (18) > $ts2.ToString()
cond
```

The output shows that the call to [Complete-Transaction](#) has committed the changes to both strings. The transaction context recorded both operations and performed them as a unit when you executed the command to commit the transaction.

You could continue to experiment with these simple string operations, but let's look at something practical. Earlier, when we displayed the commands that supported transactions, you saw that nearly all the in-box commands use the provider infrastructure. In the next section, you'll see how you can use these commands.

D.5.3 Provider support for transacted operations

In this section, we'll look at how transactions work with PowerShell providers and then work through an example using the Registry provider. Let's spend a minute and refresh what you know about the provider infrastructure. A PowerShell provider is a dynamically loadable plug-in that provides file-system-like access to hierarchical data stores. By default, PowerShell includes providers for the file system, the Registry, the certificate store, and so on. When a provider cmdlet is executed in the context of a transaction, the cmdlet has to pass the transaction context through to the provider. So for the cmdlet to be able to support transactions, the provider also has to support transactions. This means you need to be able to find out which providers support transactions and which ones don't. This turns out to be simple because the provider infrastructure lets you see what capabilities each provider has.

The following command lists all the loaded providers by name along with the drives bound to the provider and the provider's capabilities:

```
PS (1) > Get-PSProvider |  
>> Format-List Name, Drives, Capabilities  
>>  
  
Name           : WSMAN  
Drives         : {WSMAN}  
Capabilities    : Credentials  
  
Name           : Alias  
Drives         : {Alias}  
Capabilities    : ShouldProcess  
  
Name           : Environment  
Drives         : {Env}  
Capabilities    : ShouldProcess  
  
Name           : FileSystem  
Drives         : {C, G}  
Capabilities    : Filter, ShouldProcess  
  
Name           : Function  
Drives         : {Function}  
Capabilities    : ShouldProcess  
  
Name           : Registry  
Drives         : {HKLM, HKCU}  
Capabilities    : ShouldProcess, Transactions  
  
Name           : Variable  
Drives         : {Variable}  
Capabilities    : ShouldProcess
```

```
Name          : Certificate
Drives        : {cert}
Capabilities   : ShouldProcess
```

By inspecting the list, you can (eventually) see that only the Registry provider appears to support transactions. You can make this simpler and filter out invalid entries by checking for the `Transactions` provider capability. The capabilities are represented in the provider by a bitmap and defined by an `Enum` type. To do the filtering, first get the `Enum` object:

```
PS (2) > $pcType = `
>> [system.management.automation.provider.providercapabilities]
>>
```

Now use this type with the `-band` operator to select providers with the `Transactions` bit set:

```
PS (3) > Get-PSProvider |
>> where {$_.Capabilities -band $pcType::Transactions } |
>> Format-List Name, Drives, Capabilities
>>
```

```
Name          : Registry
Drives        : {HKLM, HKCU}
Capabilities   : ShouldProcess, Transactions
```

This output confirms that only the Registry provider supports transacted operations in PowerShell V2. This is unfortunate because although Registry settings are extremely important when managing and configuring systems, the file system and other stores are also extremely important. Without transaction support in all the stores involved in an operation, it isn't possible to perform transacted operations that span multiple stores.

Transaction support in the Registry is the first step toward broad transaction support in the PowerShell ecosystem. Even so, it's worth learning how to use transactions with the Registry provider, both for performing reliable operations in the Registry and getting an overall understanding of transactions in providers. Because the transaction infrastructure in PowerShell is complete and open, you aren't limited to what's in the box. Third parties can create their own transacted providers that can participate in the ecosystem. For now, let's see what you can do with the Registry provider.

Transacted Registry operations

In this section, you'll look at performing transacted operations in the Registry provider. You'll create some keys and values and then do transacted copies of the values to another key. You'll look at both the rollback and commit cases and see what happens when an error occurs during the transacted operation.

The first thing you do is create a safe place to experiment. Set the current directory to the current user Registry hive:

```
PS (1) > cd hkcu:\
```


Create a test subkey to contain your experiments:

```
PS (2) > mkdir TestContainer
```

```
Hive: HKEY_CURRENT_USER
```

SKC	VC	Name	Property
---	--	----	-----
0	0	TestContainer	{}

Next, `cd` into the test subkey

```
PS (3) > cd TestContainer
```

and create two more subkeys `Src` and `Dest`:

```
PS (4) > mkdir Src
```

```
Hive: HKEY_CURRENT_USER\TestContainer
```

SKC	VC	Name	Property
---	--	----	-----
0	0	Src	{}

```
PS (5) > mkdir Dest
```

```
Hive: HKEY_CURRENT_USER\TestContainer
```

SKC	VC	Name	Property
---	--	----	-----
0	0	Dest	{}

Now add three `DWord` values to the `Src` subkey: `one`, `two`, and `three`. These are the values you'll copy in your experiments:

```
PS (6) > New-Item -Path Src -Name one -Type Dword -Value 1
```

```
Hive: HKEY_CURRENT_USER\TestContainer\Src
```

SKC	VC	Name	Property
---	--	----	-----
0	1	one	{{default}}

```
PS (7) > New-Item -Path Src -Name two -Type Dword -Value 2
```

```
Hive: HKEY_CURRENT_USER\TestContainer\Src
```

SKC	VC	Name	Property
---	--	----	-----
0	1	two	{{default}}

```
PS (8) > New-Item -Path Src -Name three -Type Dword -Value 3
```

```
Hive: HKEY_CURRENT_USER\TestContainer\Src
```

SKC	VC	Name	Property
---	--	----	-----
0	1	three	{{default}}

This completes the setup for your experiments. Let's look at where you are:

```
PS (9) > dir Src
```

```
Hive: HKEY_CURRENT_USER\TestContainer\Src
```

SKC	VC	Name	Property
---	--	----	-----
0	1	one	{(default)}
0	1	three	{(default)}
0	1	two	{(default)}

The `Src` key has three values, and your task is to copy them to the `Dest` directory, transacted. You begin, as always, by calling `Start-Transaction` to establish the transaction context:

```
PS (10) > Start-Transaction
```

Suggestion [1,Transactions]: Once a transaction is started, only commands that get called with the `-UseTransaction` flag become part of that transaction.

As before, because you're doing this interactively, you get some helpful hints. Now let's start the `copy` operations. Instead of copying all three keys with a single command, you'll use individual `copy` commands to illustrate some of the features of transactions. Let's begin with the first copy:

```
PS (11) > copy -UseTransaction -Path Src\one -Destination Dest `
>> -Verbose
VERBOSE: Performing operation "Copy Key" on Target "Item:
HKEY_CURRENT_USER\TestContainer\Src\one Destination:
HKEY_CURRENT_USER\TestContainer\Dest".
```

You use `-UseTransaction` to have the `copy` commands enlist in the current transaction and `-Verbose` so you can see what's happening at each step. Proceed with the remaining two `copy` operations:

```
PS (12) > copy -UseTransaction -Path Src\two -Destination Dest -V
erbose
VERBOSE: Performing operation "Copy Key" on Target "Item:
HKEY_CURRENT_USER\TestContainer\Src\two Destination:
HKEY_CURRENT_USER\TestContainer\Dest".
PS (13) > copy -UseTransaction -Path Src\three -Destination Dest
-Verbose
VERBOSE: Performing operation "Copy Key" on Target "Item:
HKEY_CURRENT_USER\TestContainer\Src\three Destination:
HKEY_CURRENT_USER\TestContainer\Dest".
```

Let's examine the `Dest` key. In a non-transacted operation, the values copied should all be visible on the `Dest` key. Because you use `-Verbose` on your commands, you see that each of the copy operations is performed. Here's what happens when you call `dir` on the key, specifying the `-UseTransaction` to parameter to have the `dir` command execute in the transaction context:

```
PS (14) > dir -UseTransaction Dest

Hive: HKEY_CURRENT_USER\TestContainer\Dest
```

SKC	VC	Name	Property
---	--	----	-----
0	1	one	{(default)}
0	1	two	{(default)}
0	1	three	{(default)}

The output shows that the keys have all been copied as expected. But here's what happens when you call `dir` without `-UseTransaction`—that is, without using the transaction context:

```
PS (15) > dir Dest
PS (16) >
```

The key hasn't been updated. Roll back the transaction with `Undo-Transaction`

```
PS (16) > Undo-Transaction
```

and confirm that the key is unchanged:

```
PS (17) > dir Dest
PS (17) >
```

The output confirms that no changes were made to the key.

In the next experiment, first repeat the same basic steps to set up a new transaction context:

```
PS (18) > Start-Transaction
```

Suggestion [1,Transactions]: Once a transaction is started, only commands that get called with the `-UseTransaction` flag become part of that transaction.

Copy all the keys. The first two keys copy properly:

```
PS (19) > copy -UseTransaction -Path Src\one -Destination Dest -Verbose
VERBOSE: Performing operation "Copy Key" on Target "Item:
HKEY_CURRENT_USER\TestContainer\Src\one Destination:
HKEY_CURRENT_USER\TestContainer\Dest".
PS (20) > copy -UseTransaction -Path Src\two -Destination Dest -Verbose
VERBOSE: Performing operation "Copy Key" on Target "Item:
HKEY_CURRENT_USER\TestContainer\Src\two Destination:
HKEY_CURRENT_USER\TestContainer\Dest".
```

But wait—there's a typo in the last copy command. The source path name is wrong:

```
PS (21) > copy -UseTransaction -Path Src\threeee -Destination Dest -Verbose
Copy-Item : Cannot find path 'HKCU:\TestContainer\Src\threeee' because it does not exist.
At line:1 char:5
```

```
+ copy <<<< -UseTransaction -Path Src\threeee -Destination Dest
-Verbose
+ CategoryInfo          : ObjectNotFound: (HKCU:\TestContai
ner\Src\threeee:String) [Copy-Item], ItemNotFoundException
+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell
.Commands.CopyItemCommand
```

The error shows you the mistake. So where are you with the transaction? Let's try to commit what you've got so far. This would result in the `Dest` key having an incomplete set of keys, which isn't what you want to happen:

```
PS (22) > Complete-Transaction
Complete-Transaction : Cannot commit transaction. The transactio
n has been rolled back or has timed out.
At line:1 char:21
+ Complete-Transaction <<<<
+ CategoryInfo          : NotSpecified: (:) [Complete-Trans
action], TransactionAbortedException
+ FullyQualifiedErrorId : System.Transactions.TransactionAb
ortedException,Microsoft.PowerShell.Commands.CompleteTransa
ctionCommand
```

The error message tells you that the transaction has been rolled back and the pending changes discarded. By default, if any operation in a transaction has an error, the transaction is automatically rolled back. This behavior is controlled through the `-RollbackPreference` parameter, which can take one of the three values described in table D.3

Table D.3 The possible `-RollbackPreference` values and their descriptions

Preference value	Description
Error	The transaction failed, and pending operations are rolled back on any kind of error, terminating or non-terminating. (See section 14.1 for more on terminating versus non-terminating errors.) This is the default value.
TerminatingError	Only fail the transaction and roll back when a terminating error occurs. For example, calling the <code>throw</code> statement will fail a transaction when this setting is used.
Never	The transaction is never automatically failed. The transaction will only complete when either <code>Commit-Transaction</code> or <code>Undo-Transaction</code> is called or the transaction times out.

In the final experiment, you'll successfully commit the transaction. Start the transaction:

```
PS (23) > Start-Transaction

Suggestion [1,Transactions]: Once a transaction is started, only
commands that get called with the -UseTransaction flag become par
t of that transaction.
```

Perform the copies correctly this time:

```
PS (24) > copy -UseTransaction -Path Src\one -Destination Dest -V
erbose
```

```

VERBOSE: Performing operation "Copy Key" on Target "Item:
HKEY_CURRENT_USER\TestContainer\Src\one Destination:
HKEY_CURRENT_USER\TestContainer\Dest".
PS (25) > copy -UseTransaction -Path Src\two -Destination Dest -V
erbose
VERBOSE: Performing operation "Copy Key" on Target "Item:
HKEY_CURRENT_USER\TestContainer\Src\two Destination:
HKEY_CURRENT_USER\TestContainer\Dest".
PS (26) > copy -UseTransaction -Path Src\three -Destination Dest
-Verbose
VERBOSE: Performing operation "Copy Key" on Target "Item:
HKEY_CURRENT_USER\TestContainer\Src\three Destination:
HKEY_CURRENT_USER\TestContainer\Dest".

```

Call `dir` on the `Dest` key to verify that nothing has happened so far:

```

PS (27) > dir Dest
PS (28)

```

Everything is as expected. Now, call `Complete-Transaction` to commit the changes:

```

PS (28) > Complete-Transaction
PS (29)

```

And call `dir` on the `Dest` key one last time:

```

PS (29) > dir Dest

Hive: HKEY_CURRENT_USER\TestContainer\Dest

SKC  VC Name                                     Property
---  --  ---
0    1  one                                           {(default)}
0    1  three                                          {(default)}
0    1  two                                           {(default)}

```

The output confirms that the changes from all three `copy` commands have been committed and the `Dest` key has all three values.

A related but more realistic application of transactions is the case where the values are being copied to a backup key and then the current values are updated. If one of the values being copied is read-only, copying the keys will work the first time when the backup key has no properties. Subsequent copies will fail, though, because there is now a value on the backup key that needs to be overwritten—but you can't, because it's read-only. The net result is a mess—you don't have a complete copy of source values, and your backup key contains a mix of new and old values. If you haven't caught this error, then assigning new values to the keys that you think have been backed up will destroy the value you thought you'd backed up. If this is some unrecoverable value like an encryption key, it could be a bad situation.

To address this without transactions, you'd have to carefully check each operation to see if it was successful. But even if you caught 99 percent of the errors, that 1 percent you missed could still have catastrophic results.

NOTE This is why things like OS security are hard. If you miss an exploitable bug, then all the other work you've done becomes a waste of time. Close isn't enough. It has to be perfect. Close only counts in horseshoes and curling. Okay—and shuffleboard. And croquet. Hmm—I guess it counts in a lot of analog systems. But computers are binary, and the smallest error can potentially affect every part of the system. We talked about these topics at length when we explored security in chapter 21.

A more reliable approach would be to perform every step in the context of a transaction. If the backup fails, you roll back. If the backup succeeds but the update fails, you roll back the system. Both the original values and the backup copies are unchanged and will remain unchanged until the entire operation is completed successfully.

D.6 SUMMARY

This appendix introduced a number of additional topics that weren't covered in the core of the book. This is because these topics are less frequently encountered in day-to-day PowerShell use. We discussed the following:

- The restricted language (sometimes called the data language) subset of PowerShell. This feature is used in a number of places, most notably in the module manifests (chapter 10) and in the defining resource catalogs.
- The various features in PowerShell that support creating world-ready scripts and how to use these features to create a localizable script or module.
- Low-level tracing: a developer-level technique that allows you to see how PowerShell performs operations in a fine-grained way.
- Although PowerShell V2 introduced integrated remoting capabilities, there are still cases, such as performing remote management on a target machine where PowerShell isn't installed, that require the use of an alternate remote access mechanism. This appendix covered five additional mechanisms to consider.
- Transactions, some of the limitations in the current implementation, and how they can be applied using what is there today.

A detailed knowledge of all these subject areas isn't usually required for day-to-day material, but it's a good idea to at least be aware of the features that are available should they be needed. They're also interesting to look at as more examples of how things fit together in the Windows PowerShell world and how you might approach solving new problems using PowerShell.

Windows PowerShell IN ACTION

SECOND EDITION

Bruce Payette



Praise for the Second Edition

“First he wrote the language, then he wrote the book.”

—Jeffrey Snover, Microsoft

“Not just a reference. It’s worth reading cover to cover.”

—Jason Zions, Microsoft

“Unleashes the *power* in PowerShell.”

—Sam Abraham, SISCO

“Even better than the original.”

—Tomas Restrepo
winterdom.com

“Still the definitive reference.”

—Keith Hill
Agilent Technologies

This expanded, revised, and updated Second Edition preserves the original’s crystal-clear introduction to PowerShell and adds extensive coverage of v2 features such as advanced functions, modules, and remoting. It includes full chapters on these topics and also covers new language elements and operators, events, Web Services for Management, and the PowerShell Integrated Scripting Environment.

The First Edition’s coverage of batch scripting and string processing, COM, WMI, and .NET have all been significantly revised and expanded. The book includes many popular usage scenarios and is rich in interesting examples that will spark your imagination. This is the definitive book on PowerShell v2!

What’s Inside

- Batch scripting, string processing, files, and XML
- PowerShell remoting
- Application of COM and WMI
- Network and GUI programming
- Writing modules and scripts

Written for developers and administrators with intermediate level scripting knowledge. No prior experience with PowerShell is required.

Bruce Payette is a founding member of the PowerShell team at Microsoft. He is co-designer and principal author of the PowerShell language.

For access to the book’s forum and a free ebook for owners of this book, go to manning.com/WindowsPowerShellinActionSecondEdition



\$59.99 / Can \$68.99 [INCLUDING eBook]

ISBN 13: 978-1-935182-13-9
ISBN 10: 1-935182-13-7



9 781935 182139