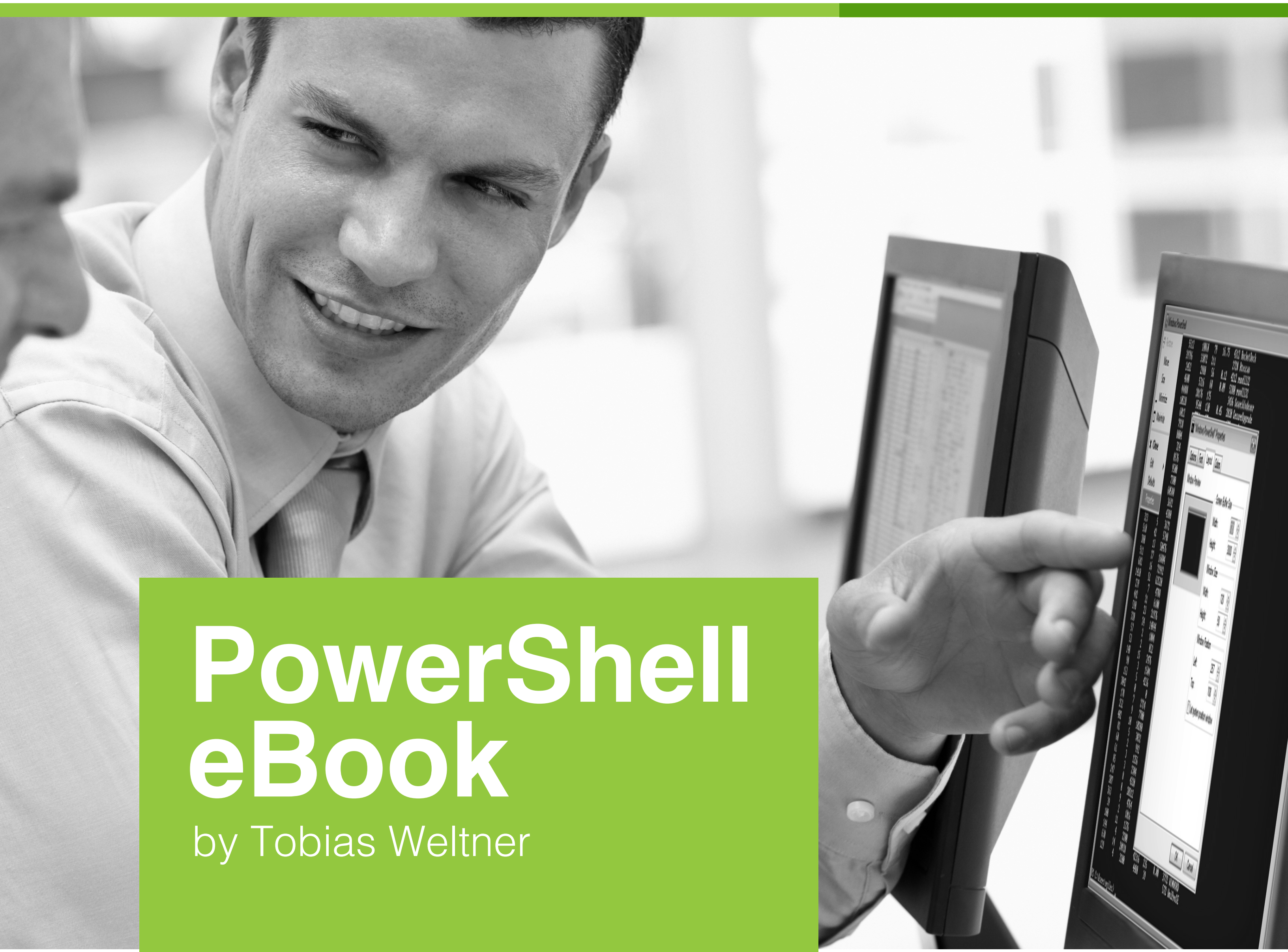


PowerShell eBook

by Tobias Weltner

Index

by Tobias Weltner

03 The Power Console

19 Interactive PowerShell

48 Variables

74 Arrays and Hashtables

94 The PowerShell Pipeline

112 Working with Objects

Chapter 1.
The PowerShell Console

Welcome to PowerShell! This chapter will introduce you to the PowerShell console and show you how to configure it, including font colors and sizes, editing and display options.

Topics Covered:

- **Starting PowerShell**
- **First Steps with the Console**
  - Incomplete and Multi-Line Entries
  - Important Keyboard Shortcuts
  - Deleting Incorrect Entries
  - Overtype Mode
  - Command History: Reusing Entered Commands
  - Automatically Completing Input
  - Scrolling Console Contents
  - Selecting and Inserting Text
  - QuickEdit Mode
  - Standard Mode
- **Customizing the Console**
  - Opening Console Properties
  - Defining Options
  - Specifying Fonts and Font Sizes
  - Setting Window and Buffer Size
  - Selecting Colors
  - Directly Assigning Modifications in PowerShell
  - Saving Changes
- **Piping and Routing**
  - Piping: Outputting Information Page by Page
  - Redirecting: Storing Information in Files
- **Summary**

Starting PowerShell

On Windows 7 and Server 2008 R2, Windows PowerShell is installed by default. To use PowerShell on older systems, you need to download and install it. The update is free. The simplest way to find the appropriate download is to visit an Internet search engine and search for "KB968930 Windows XP" (replace the operating system with the one you use). Make sure you pick the correct update. It needs to match your operating system language and architecture (32-bit vs. 64-bit).

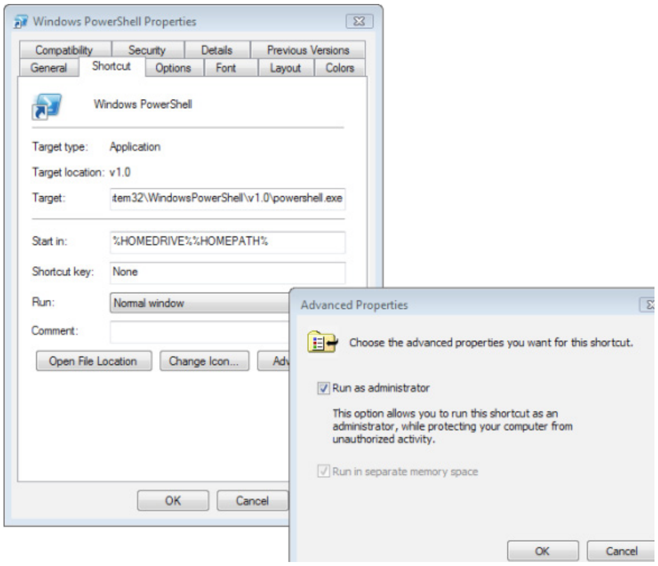
After you installed PowerShell, you'll find PowerShell in the Accessories program group. Open this program group, click on Windows PowerShell and then launch the PowerShell executable. On 64-bit systems, you will also find a version marked as (x86) so you can run PowerShell both in the default 64-bit environment and in an extra 32-bit environment for backwards compatibility.

You can also start PowerShell directly. Just press (Windows)+(R) to open the Run window and then enter powershell (Enter). If you use PowerShell often, you should open the program folder for Windows PowerShell and right-click on Windows PowerShell. That will give you several options:

· **Add to the start menu:** On the context menu, click on Pin to Start Menu so that PowerShell will be displayed directly on your start menu from now on and you won't need to open its program folder first.
· **Quick Launch toolbar:** Click Add to Quick Launch toolbar if you use Windows Vista and would like to see PowerShell right on the Quick Launch toolbar inside your taskbar. Windows XP lacks this command so XP users will have to add PowerShell to the Quick Launch toolbar manually.

· **Jump List:** On Windows 7, after launching PowerShell, you can right-click the PowerShell icon in your taskbar and choose Pin to Taskbar. This will not only keep the PowerShell icon in your taskbar so you can later easily launch PowerShell. It also gives access to its new "Jump List": right-click the icon (or pull it upwards with your mouse). The jump list contains a number of useful PowerShell functions: you can launch PowerShell with full administrator privileges, run the PowerShell ISE, or open the PowerShell help file. By the way: drag the pinned icon all to the left in your taskbar. Now, pressing WIN+1 will always launch PowerShell. And here are two more tips: hold SHIFT while clicking the PowerShell icon in your taskbar will open a new instance, so you can open more than one PowerShell console. Holding SHIFT+CTRL while clicking the PowerShell icon opens the PowerShell console with full Administrator privileges (provided User Account Control is enabled on your system).

· **Keyboard shortcuts:** Administrators particularly prefer using a keyboard instead of a mouse. If you select Properties on the context menu, you can specify a key combination in the hot-key field. Just click on this field and press the key combination intended to start PowerShell, such as (Alt)+(P). In the properties window, you also ha ve the option of setting the default window size to start PowerShell in a normal, minimized, or maximized window.



Windows PowerShell

Target type: Application

Target location: v1.0

Target: tem32\WindowsPowerShell\v1.0\powershell.exe

Start in: %HOMEDRIVE%%HOMEPATH%

Shortcut key: None

Run: Normal window

Comment:

Choose the advanced properties you want for this shortcut.

☑ Run as administrator

This option allows you to run this shortcut as an administrator, while protecting your computer from unauthorized activity.

☑ Run in separate memory space

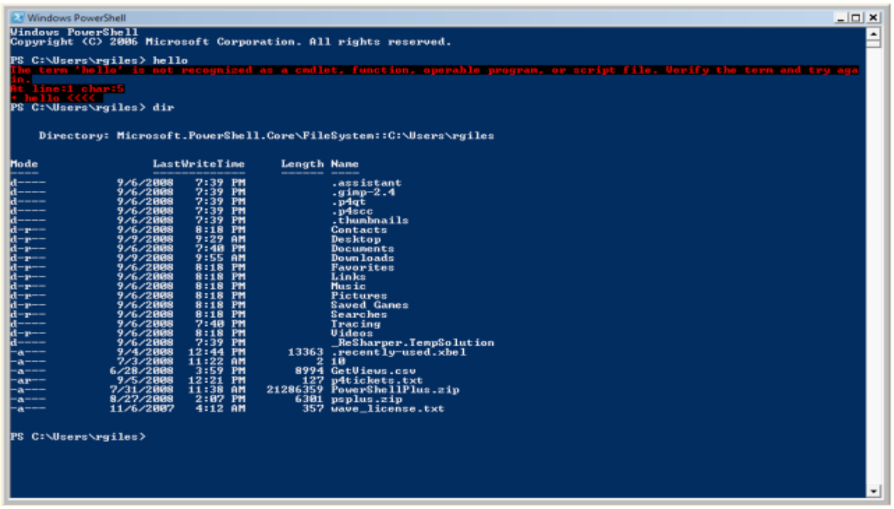
Figure 1.1: How to always open PowerShell with administrator rights
(Run without administrative privileges whenever possible)

First Steps with the Console

After PowerShell starts, its console window opens, and you see a blinking text prompt, asking for your input with no icons or menus. PowerShell is a command console and almost entirely operated via keyboard input. The prompt begins with “PS” and after it is the path name of the directory where you are located. Start by trying out a few commands. For example, type:

```
hello (Enter)
```

As soon as you press (Enter), your entry will be sent to PowerShell. Because PowerShell has never heard of the command “hello” you will be confronted with an error message highlighted in red.



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

PS C:\Users\rgiles> hello
The term 'hello' is not recognized as a cmdlet, function, operable program, or script file. Verify the term and try aga
in.
At line:1 char:5
+ hello <<<<
PS C:\Users\rgiles> dir


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\Users\rgiles


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----          9/6/2008   7:39 PM            .assistant
d----          9/6/2008   7:39 PM            .gimp-2.4
d----          9/6/2008   7:39 PM            .p4qt
d----          9/6/2008   7:39 PM            .p4scc
d----          9/6/2008   7:39 PM            .thumbnails
d-r--          9/6/2008   8:18 PM            Contacts
d-r--          9/9/2008   9:29 AM            Desktop
d-r--          9/6/2008   7:40 PM            Documents
d-r--          9/9/2008   9:55 AM            Downloads
d-r--          9/6/2008   8:18 PM            Favorites
d-r--          9/6/2008   8:18 PM            Links
d-r--          9/6/2008   8:18 PM            Music
d-r--          9/6/2008   8:18 PM            Pictures
d-r--          9/6/2008   8:18 PM            Saved Games
d-r--          9/6/2008   8:18 PM            Searches
d----          9/6/2008   7:40 PM            Tracing
d-r--          9/6/2008   8:18 PM            Videos
d----          9/6/2008   7:39 PM            _ReSharper.TempSolution
-a---          9/4/2008  12:44 PM      13363 .recently-used.xbel
-a---          7/3/2008  11:22 AM          2 10
-a---         6/28/2008   3:59 PM       8994 GetViews.csv
-ar--          9/5/2008  12:21 PM        127 p4tickets.txt
-a---         7/31/2008  11:38 AM   21286359 PowerShellPlus.zip
-a---         8/27/2008   2:07 PM       6301 psplus.zip
-a---         11/6/2007   4:12 AM        357 wave_license.txt


PS C:\Users\rgiles>
```

Figure 1.2: First commands in the PowerShell console

For example, if you’d like to see which files and folders are in your current directory, then type dir (Enter). You’ll get a text listing of all the files in the directory. PowerShell’s communication with you is always text-based. PowerShell can do much more than display simple directory lists. You can just as easily list all running processes or all installed hotfixes: Just pick a different command as the next one provides a list of all running processes:

```
Get-Process (Enter)
```

```
Get-Hotfix (Enter)
```

PowerShell’s advantage is its tremendous flexibility since it allows you to control and display nearly all the information and operations on your computer. The command cls deletes the contents of the console window and the exit command ends PowerShell.

Incomplete and Multi-line Entries

Whenever you enter something PowerShell cannot understand, you get a red error message, explaining what went wrong. However, if you enter something that isn't wrong but incomplete (like a string with one missing closing quote), PowerShell gives you a chance to complete your input. You then see a double-prompt (">>"), and once you completed the line and pressed ENTER twice, PowerShell executes the command. You can also bail out at any time and cancel the current command or input by pressing: (Ctrl)+(C).

The "incomplete input" prompt will also appear when you enter an incomplete arithmetic problem like this one:

```
 2 + (Enter)
 >> 6 (Enter)
 >> (Enter)
   8
```

This feature enables you to make multi-line PowerShell entries:

```
"This is my little multiline entry.(Enter)
>> I'm now writing a text of several lines. (Enter)
>> And I'll keep on writing until it's no longer fun."(Enter)
>>(Enter)

This is my little multiline entry.
I'm now writing a text of several lines.
And I'll keep on writing until it's no longer fun.
```

The continuation prompt generally takes its cue from initial and terminal characters like open and closed brackets or quotation marks at both ends of a string. As long as the symmetry of these characters is incorrect, you'll continue to see the prompt. However, you can activate it even in other cases:

```
dir `(Enter)
>> -recurse(Enter)
>>(Enter)
```

So, if the last character of a line is what is called a "back-tick" character, the line will be continued. You can retrieve that special character by pressing (`).

Important Keyboard Shortcuts

Shortcuts are important since almost everything in PowerShell is keyboard-based. For example, by pressing the keys (Arrow left) and (Arrow right), you can move the blinking cursor to the left or right. Use it to go back and correct a typo. If you want to move the cursor word by word, hold down (Ctrl) while pressing the arrow keys. To place the cursor at the beginning of a line, hit (Home). Pressing (End) will send the cursor to the end of a line.

Important

If you haven't entered anything, then the cursor won't move since it will only move within entered text. There's one exception: if you've already entered a line and pressed (Enter) to execute the line, you can make this line appear again character-by-character by pressing (Arrow right).

Deleting Incorrect Entries

If you've mistyped something, press (Backspace) to delete the character to the left of the blinking cursor. (Del) erases the character to the right of the cursor. And you can use (Esc) to delete your entire current line.

The hotkey (Ctrl)+(Home) works more selectively: it deletes all the characters at the current position up to the beginning of the line. Characters to the right of the current position (if there are any) remain intact. (Ctrl)+(End) does it the other way around and deletes everything from the current position up to the end of the line. Both combinations are useful only after you've pressed (Arrow left) to move the cursor to the middle of a line, specifically when text is both to the left and to the right of the cursor.

Overtype Mode

If you enter new characters and they overwrite existing characters, then you know you are in type-over mode. By pressing (Insert) you can switch between insert and type-over modes. The default input mode depends on the console settings you select. You'll learn more about console settings soon.The "incomplete input" prompt will also appear when you enter an incomplete arithmetic problem like this one:

Command History:
Reusing Entered Commands

The most awesome feature is a built-in search through all of the commands you used in your current session: simply type "#" and then some search text that you know exists in one or more of your previous commands. Next, type TAB one or more times to see all the commands that contained your keyword. Press ENTER to execute the command once you found it, or edit the command line to your liking.

If you just wanted to polish or correct one of your most recent commands, press (Arrow up) to re-display the command that you entered. Press (Arrow up) and (Arrow down) to scroll up and down your command history. Using (F5) and (F8) do the same as the up and down arrow keys.

This command history feature is extremely useful. Later, you'll learn how to configure the number of commands the console "remembers". The default setting is the last 50 commands. You can display all the commands in your history by pressing (F7) and then scrolling up and down the list to select commands using (Arrow up) and (Arrow down) and (Enter).

Important

The numbers before the commands in the Command History list only denote the sequence number. You cannot enter a number to select the associated command. What you can do is move up and down the list by hitting the arrow keys.

Simply press (F9) to ‘activate’ the numbers so that you can select a command by its number. This opens a menu that accepts the numbers and returns the desired command.

The keyboard sequence (Alt)+(F7) will clear the command history and start you off with a new list.

(F8) provides more functionality than (Arrow up) as it doesn’t just show the last command you entered, but keeps a record of the characters you’ve already typed in. If, for example, you’d like to see all the commands you’ve entered that begin with “d”, type:

The “incomplete input” prompt will also appear when you enter an incomplete arithmetic problem like this one:

```
d (F8)
```

Press (F8) several times. Every time you press a key another command will be displayed from the command history provided that you’ve already typed in commands with an initial “d.”

Automatically Completing Input

An especially important key is (Tab). It will save you a great deal of typing (and typing errors). When you press this key, PowerShell will attempt to complete your input automatically. For example, type:

```
cd (Tab)
```

The command cd changes the directory in which you are currently working. Put at least one space behind the command and then press (Tab). PowerShell suggests a sub-directory. Press (Tab) again to see other suggestions. If (Tab) doesn’t come up with any suggestions, then there probably aren’t any sub-directories available.

This feature is called Tab-completion, which works in many places. For example, you just learned how to use the command Get-Process, which lists all running processes. If you want to know what other commands there are that begin with “Get-”, then type:

```
Get-(Tab)
```

Just make sure that there’s no space before the cursor when you press (Tab). Keep hitting (Tab) to see all the commands that begin with “Get-”.

Important

A more complete review of the Tab-completion feature is available in **Chapter 9**.

Tab-completion works really well with long path names that require a lot of typing. For example:

```
c:\p(Tab)
```

Every time you press (Tab), PowerShell will prompt you with a new directory or a new file that begins with “c:\p.” So, the more characters you type, the fewer options there will be. In practice, you should type in at least four or five characters to reduce the number of suggestions.

When the list of suggestions is long, it can take a second or two until PowerShell has compiled all the possible suggestions and displays the first one.

Important

Wildcards are allowed in path names. For example, if you enter c:\pr*e (Tab) in a typical Windows system, PowerShell will respond with “c:\Program Files”.

PowerShell will automatically put the entire response inside double quotation marks if the response contains whitespace characters.

Scrolling Console Contents

The visible part of your console depends on the size of your console window, which you can change with your mouse. Drag the window border while holding down your left mouse button until the window is the size you want. Note that the actual contents of the console, the “screen buffer,” don’t change. So, if the window is too small to show everything, you should use the scroll bars.

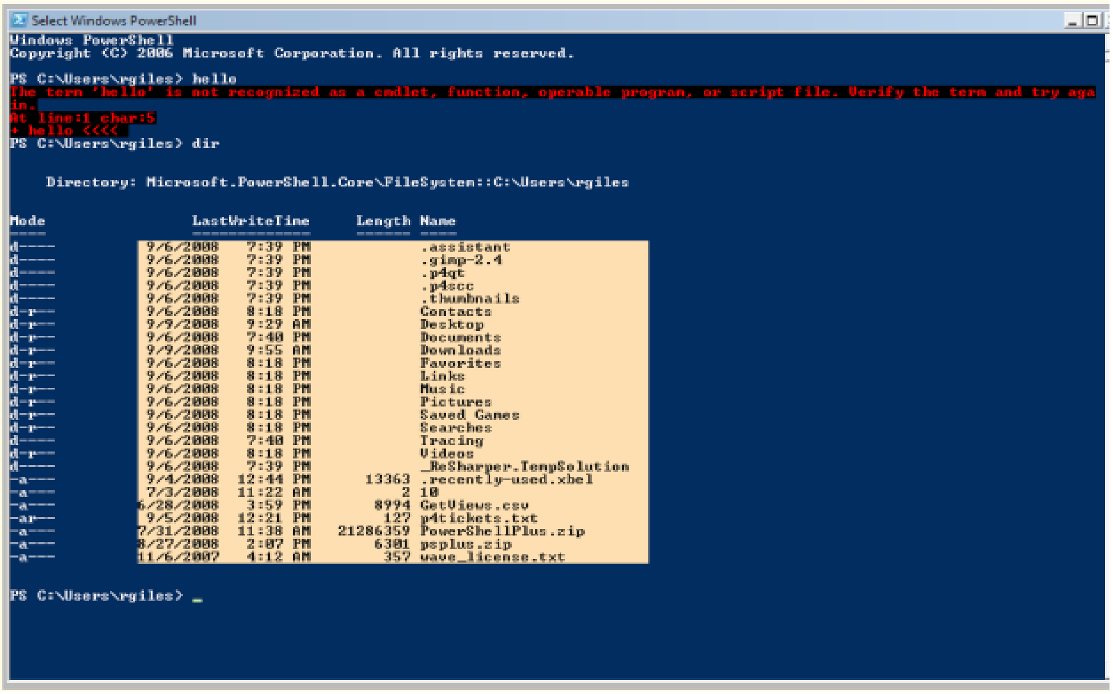
Selecting and Inserting Text

Use your mouse if you’d like to select text inside the PowerShell window and copy it onto the clipboard. Move the mouse pointer to the beginning of the selected text, hold down the left mouse button and drag it over the text area that you want to select.

Quick Edit Mode

QuickEdit is the default mode for selecting and copying text in PowerShell. Select the text using your mouse and PowerShell will highlight it. After you’ve selected the text, press (Enter) or right-click on the marked area. This will copy the selected text to the clipboard which you can now paste into other applications. To unselect press (Esc).

You can also insert the text in your console at the blinking command line by right-clicking your mouse.



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

PS C:\Users\rgiles> hello
The term 'hello' is not recognized as a cmdlet, function, operable program, or script file. Verify the term and try aga
in.
At line:1 char:5
+ hello <<<<
PS C:\Users\rgiles> dir


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\Users\rgiles


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----          9/6/2008   7:39 PM            .assistant
d----          9/6/2008   7:39 PM            .gimp-2.4
d----          9/6/2008   7:39 PM            .p4qt
d----          9/6/2008   7:39 PM            .p4scc
d----          9/6/2008   7:39 PM            .thumbnails
d-r--          9/6/2008   8:18 PM            Contacts
d-r--          9/9/2008   9:29 AM            Desktop
d-r--          9/6/2008   7:40 PM            Documents
d-r--          9/9/2008   9:55 AM            Downloads
d-r--          9/6/2008   8:18 PM            Favorites
d-r--          9/6/2008   8:18 PM            Links
d-r--          9/6/2008   8:18 PM            Music
d-r--          9/6/2008   8:18 PM            Pictures
d-r--          9/6/2008   8:18 PM            Saved Games
d-r--          9/6/2008   8:18 PM            Searches
d----          9/6/2008   7:40 PM            Tracing
d-r--          9/6/2008   8:18 PM            Videos
d----          9/6/2008   7:39 PM            _ReSharper.TempSolution
-a---          9/4/2008  12:44 PM      13363 .recently-used.xbel
-a---          7/3/2008  11:22 AM          2 10
-a---         6/28/2008   3:59 PM       8994 GetViews.csv
-ar--          9/5/2008  12:21 PM        127 p4tickets.txt
-a---         7/31/2008  11:38 AM   21286359 PowerShellPlus.zip
-a---         8/27/2008   2:07 PM       6301 psplus.zip
-a---         11/6/2007   4:12 AM        357 wave_license.txt


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

Figure 1.3: Marking and copying text areas in QuickEdit mode

Standard Mode

If QuickEdit is turned off and you are in Standard mode, the simplest way to mark and copy text is to right-click in the console window. If QuickEdit is turned off, a context menu will open.

Select Mark to mark text and Paste if you want to insert the marked text (or other text contents that you've copied to the clipboard) in the console.

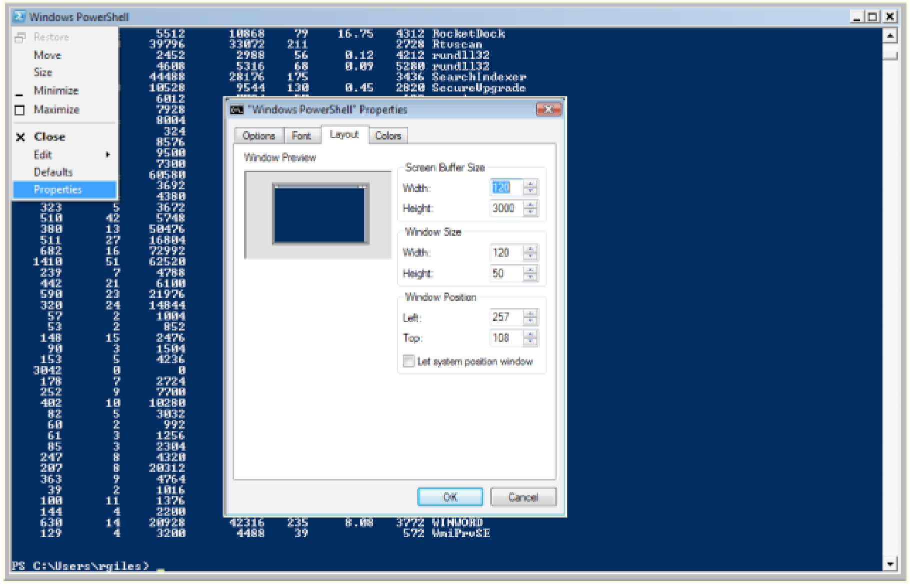
It's usually more practical to activate QuickEdit mode so that you won't have to use the context menu

Customizing the Console

You can customize a variety of settings in the console including edit mode, screen buffer size, font colors, font sizes etc.

Opening Console Properties

The basic settings of your PowerShell console are configured in a special Properties dialog box. Click on the PowerShell icon on the far left of the title bar of the console window to open it.



```
                5512     10868      79     16.75    4312 RocketDock
               39796     33072     211              2728 Rtvscan
                2452      2988      56      0.12    4212 rundll32
                4608      5316      68      0.09    5280 rundll32
               44488     28176     175              3436 SearchIndexer
               10528      9544     130      0.45    2820 SecureUpgrade
                6012
                7928
                8004
                 324
                8576
                9500
                7300
               60580
                3692
                4380
    323       5   3672
    510      42   5748
    380      13  50476
    511      27  16804
    682      16  72992
   1410      51  62520
    239       7   4788
    442      21   6100
    590      23  21976
    320      24  14844
     57       2   1004
     53       2    852
    148      15   2476
     90       3   1504
    153       5   4236
   3042       0      0
    178       7   2724
    252       9   7700
    402      10  10280
     82       5   3032
     60       2    992
     61       3   1256
     85       3   2304
    247       8   4320
    207       8  20312
    363       9   4764
     39       2   1016
    100      11   1376
    144       4   2200
    630      14  20928     42316     235      8.08    3772 WINWORD
    129       4   3200      4488      39               572 WmiPrvSE

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

Screen Buffer Size
Width: 120
Height: 3000

Window Size
Width: 120
Height: 50

Window Position
Left: 257
Top: 108

□ Let system position window

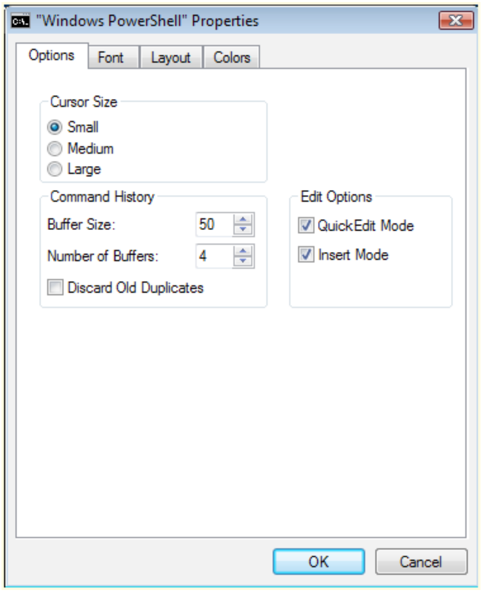
Figure 1.4: Opening console properties

That will open a context menu. You should select Properties and a dialog box will open.

To get help, click on the question mark button on the title bar of the window. A question mark is then pinned to your mouse pointer. Next, click on the option you need help for. The help appears as a ScreenTip window.

Opening Console Properties

Under the heading Options are four panels of options:



Cursor Size
● Small
○ Medium
○ Large

Command History
Buffer Size: 50
Number of Buffers: 4
□ Discard Old Duplicates

Edit Options
☑ QuickEdit Mode
☑ Insert Mode

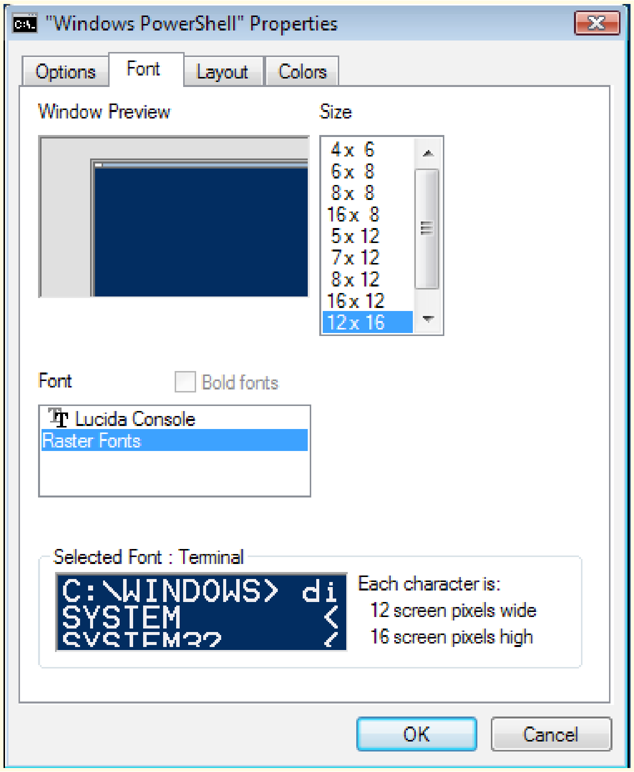
Figure 1.5: Defining the QuickEdit and Insert modes

- **Edit options:** You should select the QuickEdit mode as well as the Insert mode. We've already discussed the advantages of the
- **QuickEdit mode:** it makes it much easier to select, copy, and insert text. The Insert mode makes sure that new characters don't overwrite existing input so new characters will be added without erasing text you've already typed in when you're editing command lines.
- **Cursor size:** Here is where you specify the size of the blinking cursor.
- **Display options:** Determine whether the console should be displayed as a window or full screen. The "window" option is best so that you can switch to other windows when you're working. The full screen display option is not available on all operating systems.
- **Command history:** Here you can choose how many command inputs the console "remembers". This allows you to select a command from the list by pressing (Arrow up) or (F7). The option Discard Old Duplicates ensures that the list doesn't have any duplicate entries. So, if you enter one command twice, it will appear only once in the history list.

Specifying Fonts and Font Sizes

On the Font tab, you can choose both the font and the font size displayed in the console.

The console often uses the raster font as its default. This font is available in a specific range of sizes with available sizes shown in the "Size" list. Scalable TrueType fonts are much more flexible. They're marked in the list by a "TT" symbol. When you select a TrueType font, you can choose any size in the size list or enter them as text in the text box. TrueType fonts can be dynamically scaled.



"Windows PowerShell" Properties

Size: 12 x 16

Font: Raster Fonts

□ Bold fonts

Selected Font : Terminal

Each character is:
12 screen pixels wide
16 screen pixels high

Figure 1.6: Specifying new fonts and font sizes

You should also try experimenting with TrueType fonts by using the "bold fonts" option. TrueType fonts are often more readable if they're displayed in bold.

You should also try experimenting with TrueType fonts by using the “bold fonts” option. TrueType fonts are often more readable if they’re displayed in bold.

Pro Tip

Your choice of fonts may at first seem a bit limited. To get more font choices, you can add them to the console font list. The limited default font list is supposed to prevent you from choosing unsuitable fonts for your console.

One reason for this is that the console always uses the same width for each character (fixed width fonts). This restricts the use of most Windows fonts because they’re proportional typefaces: every character has its own width. For example, an “i” is narrower than an “m”. If you’re sure that a certain font will work in the console, then here’s how to add the font to the console font list.

Open your registry editor. In the key *HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\ CurrentVersion\Console\ TrueTypeFont* insert a new “string value” and give this entry the name “00” (numbers, not letters).

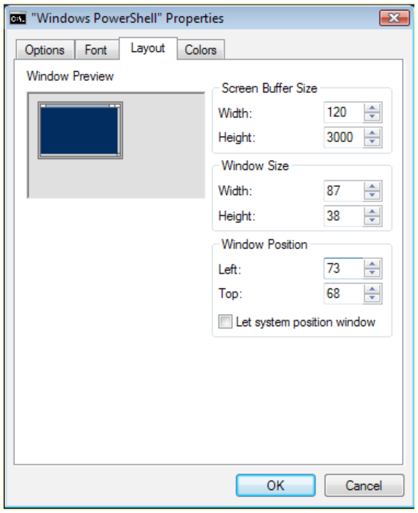
If there’s already an entry that has this name, then call the new entry “000” or add as many zeroes as required to avoid conflicts with existing entries. You should then double-click your new entry to open it and enter the name of the font. The name must be exactly the same as the official font name, just the way it’s stated under the key *HKEY_LOCAL_MACHINE\ SOFTWARE\Microsoft\Windows NT\CurrentVersion\Fonts.*

The newly added font will now turn up in the console’s option field. However, the new font will work only after you either log off at least once or restart your computer. If you fail to do so, the console will ignore your new font when you select it in the dialog box.

Setting Window and Buffer Size

On the Layout tab, you can specify how large the screen buffer should be, meaning how much information the console should “remember” and how far back you can scroll with the scroll bars.

You should select a width of at least 120 characters in the window buffer size area with the height should be at least 1,000 lines or larger. This gives you the opportunity to use the scroll bars to scroll the window contents back up so that you can look at all the results of your previous commands.



Screen Buffer Size
Width: 120
Height: 3000

Window Size
Width: 87
Height: 38

Window Position
Left: 73
Top: 68

☐ Let system position window

Figure 1.7: Specifying the size of the window buffer

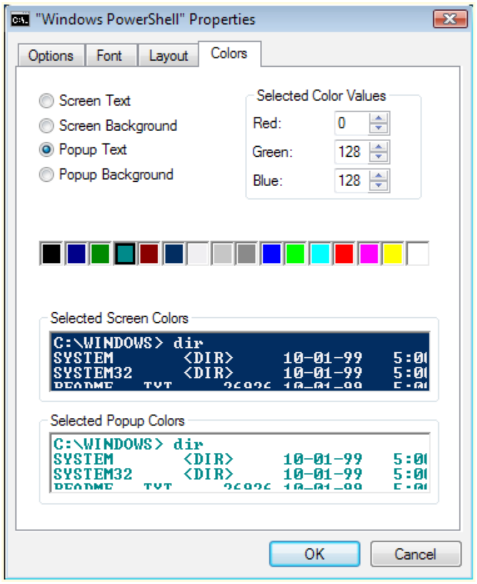
You can also set the window size and position on this tab if you'd like your console to open at a certain size and screen position on your display. Choose the option Let system position window and Windows will automatically determine at what location the console window will open.

Selecting Colors

On the Colors tab, you can select your own colors for four areas:

- **Screen text:** Console font
- **Screen background:** Console background color
- **Popup text:** Popup window font, such as command history's (F7)
- **Popup background:** Popup window background color

You have a palette of 16 colors for these four areas. So, if you want to specify a new font color, you should first select the option Screen Text and click on one of the 16 colors. If you don't like any of the 16 colors, then you can mix your own special shade of color. Just click on a palette color and choose your desired color value at the upper right from the primary colors red, green, and blue.



○ Screen Text
○ Screen Background
● Popup Text
○ Popup Background

Selected Color Values
Red: 0
Green: 128
Blue: 128

Selected Screen Colors

```
C:\WINDOWS> dir
SYSTEM          <DIR>       10-01-99    5:0
SYSTEM32        <DIR>       10-01-99    5:0
```

Selected Popup Colors

```
C:\WINDOWS> dir
SYSTEM          <DIR>       10-01-99    5:0
SYSTEM32        <DIR>       10-01-99    5:0
```

Figure 1.8: Select better colors for your console

Directly Assigning Modifications in PowerShell

Some of the console configuration can also be done from within PowerShell code. You'll hear more about this later. To give you a quick impression, take a look at this:

```
$host.ui.rawui (Enter)
$host.ui.rawui.ForegroundColor = “Yellow” (Enter)
$host.ui.rawui.WindowTitle = “My Console” (Enter)
```

These changes will only be temporary. Once you close and re-open PowerShell, the changes are gone. You would have to include these lines into one of your “profile scripts” which run every time you launch PowerShell to make them permanent. You can read more about this in **Chapter 10**.

Directly Assigning Modifications in PowerShell

Once you’ve successfully specified all your settings in the dialog box, you can close the dialog box. If you’re using Windows Vista or above, all changes will be saved immediately, and when you start PowerShell the next time, your new settings will already be in effect. You may need Admin rights to save settings if you launched PowerShell with a link in your start menu that applies for all users.

If you’re using Windows XP, you’ll see an additional window and a message asking you whether you want to save changes temporarily (Apply properties to current window only) or permanently (Modify shortcut that started this window).

Piping and Routing

You may want to view the information page by page or save it in a file since some commands output a lot of information.

Piping: Outputting Information Page by Page

The pipe command more outputs information screen page by screen page. You will need to press a button (like Space) to continue to the next page.

Piping uses the vertical bar (|). The results of the command to the left of the pipe symbol are then fed into the command on the right side of the pipe symbol. This kind of piping is also known in PowerShell as the “pipeline”:

```
Get-Process | more (Enter)
```

You can press (Ctrl)+(C) to stop output. Piping also works with other commands, not just more. For example, if you’d like to get a sorted directory listing, pipe the result to Sort-Object and specify the columns you would like to sort:

```
dir | Sort-Object -Property Length, Name (Enter)
```

You’ll find more background information on piping as well as many useful examples in **Chapter 5**.

Redirecting: Storing Information in Files

If you’d like to redirect the result of a command to a file, you can use the redirection symbol “>”:

```
Help > help.txt (Enter)
```

The information won’t appear in the console but will instead be redirected to the specified file. You can then open the file.

However, opening a file in PowerShell is different from opening a file in the classic console:

```
Help > help.txt (Enter)


      The term "help.txt" is not recognized as a cmdlet, function,
      operable program, or script file. Verify the term and try again.
      At line:1 character:8
      + help.txt <<<<
```

If you only specify the file name, PowerShell will look for it in all folders listed in the PATH environment variable. So to open a file, you will have to specify its absolute or relative path name. For example:

```
.\help.txt (Enter)
```

Or, to make it even simpler, you can use Tab-completion and hit (Tab) after the file name:

```
.\help.txt (Tab)
```

The file name will automatically be completed with the absolute path name, and then you can open it by pressing (Enter):

```
& “C:\Users\UserA\help.txt” (Enter)
```

You can also append data to an existing file. For example, if you’d like to supplement the help information in the file with help on native commands, you can attach this information to the existing file with the redirection symbol “>>”:

```
Cmd /c help >> help.txt (Enter)
```

If you’d like to directly process the result of a command, you won’t need traditional redirection at all because PowerShell can also store the result of any command to a variable:

```
$result = Ping 10.10.10.10
   $result

   Reply from 10.10.10.10: bytes=32 time<1ms TTL=128
   Reply from 10.10.10.10: bytes=32 time<1ms TTL=128
   Reply from 10.10.10.10: bytes=32 time<1ms TTL=128
   Reply from 10.10.10.10: bytes=32 time<1ms TTL=128
   Ping statistics for 10.10.10.10:
   Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
   Approximate round trip times in milli-seconds:
   Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

Variables are universal data storage and variable names always start with a “$”. You’ll find out more about variables in **Chapter 3**.

Summary

PowerShell is part of the operating system starting with Windows 7 and Server 2008 R2. On older operating systems such as Windows XP or Server 2003, it is an optional component. You will have to download and install PowerShell before using it.

The current version is 2.0, and the easiest way to find out whether you are using the most current PowerShell version is to launch the console and check the copyright statement. If it reads “2006”, then you are still using the old and outdated PowerShell 1.0. If it reads “2009”, you are using the correct version. There is no reason why you should continue to use PowerShell 1.0, so if you find it on your system, update to 2.0 as soon as possible. If you wanted to find out your current PowerShell version programmatically, output the automatic variable $psversiontable (simply by entering it). It not only tells you the current PowerShell version but also the versions of the core dependencies. This variable was introduced in PowerShell version 2.0, so on version 1.0 it does not exist.

The PowerShell console resembles the interactive part of PowerShell where you can enter commands and immediately get back results. The console relies heavily on text input. There are plenty of special keys listed in **Table 1.1**.

KEY	MEANING
`(Alt)+(F7)`	Deletes the current command history
`(PgUp), (PgDn)`	Display the first (PgUp) or last (PgDn) command you used in current session
`(Enter)`	Send the entered lines to PowerShell for execution
`(End)`	Moves the editing cursor to the end of the command line
`(Del)`	Deletes the character to the right of the insertion point
`(Esc)`	Deletes current command line
`(F2)`	Moves in current command line to the next character corresponding to specified character
`(F4)`	Deletes all characters to the right of the insertion point up to specified character
`(F7)`	Displays last entered commands in a dialog box
`(F8)`	Displays commands from command history beginning with the character that you already entered in the command line

KEY	MEANING
(F9)	Opens a dialog box in which you can enter the number of a command from your command history to return the command. (F7) displays numbers of commands in command history
(Left arrow), (Right arrow)	Move one character to the left or right respectively
(Arrow up), (Arrow down), (F5), (F8)	Repeat the last previously entered command
(Home)	Moves editing cursor to beginning of command line
(Backspace)	Deletes character to the left of the insertion point
(Ctrl)+(C)	Cancels command execution
(Ctrl)+(End)	Deletes all characters from current position to end of command line
(Ctrl)+(Arrow left), (Ctrl)+(Arrow right)	Move insertion point one word to the left or right respectively
(Ctrl)+(Home)	Deletes all characters of current position up to beginning of command line

Table 1.1: Important keys and their meaning in the PowerShell console

You will find that the keys (Arrow up), which repeats the last command, and (Tab), which completes the current entry, are particularly useful. By hitting (Enter), you complete an entry and send it to PowerShell. If PowerShell can't understand a command, an error message appears highlighted in red stating the possible reasons for the error. Two special commands are cls (deletes the contents of the console) and exit (ends PowerShell).

You can use your mouse to select information in the console and copy it to the Clipboard by pressing (Enter) or by right-clicking when you have the QuickEdit mode turned on. With QuickEdit mode turned off, you will have to right-click inside the console and then select Mark in a context menu.

The basic settings of the console—QuickEdit mode as well as colors, fonts, and font sizes—can be customized in the properties window of the console. This can be accessed by right-clicking the icon to the far left in the title bar of the console window. In the dialog box, select Properties.

Along with the commands, a number of characters in the console have special meanings and you have already become acquainted with three of them:

Piping: The vertical bar "|" symbol pipes the results of a command to the next. When you pipe the results to the command more, the screen output will be paused once the screen is full, and continued when you press a key.

Redirection: The symbol ">" redirects the results of a command to a file. You can then open and view the file contents. The symbol ">>" appends information to an existing file.

PowerShell 2.0 also comes with a simple script editing tool called "ISE" (Integrated Script Environment). You find it in PowerShell's jump list (if you are using Windows 7), and you can also launch it directly from PowerShell by entering ise ENTER. ISE requires .NET Framework 3.5.1. On Windows Server 2008 R2, it is an optional feature that needs to be enabled first in your system control panel. You can do that from PowerShell as well:

Import-Module ServerManager Add-WindowsFeature ISE -IncludeAll

Chapter 2. Interactive PowerShell

PowerShell has two faces: interactivity and script automation. In this chapter, you will first learn how to work with PowerShell interactively. Then, we will take a look at PowerShell scripts.

Topics Covered:

- **PowerShell as a Calculator**
  - Calculating with Number Systems and Units
- **Executing External Commands**
  - Starting the "Classic" Console
  - Discovering Useful Console Commands
  - Security Restrictions
  - Special Places
- **Cmdlets: PowerShell Commands**
  - Using Parameters
  - Using Named Parameters
  - Switch Parameter
  - Positional Parameters
  - Common Parameters
- **Aliases: Shortcuts for Commands**
  - Resolving Aliases
  - Creating Your Own Aliases
  - Removing or Permanently Keeping an Alias
  - Overwriting and Deleting Aliases
- **Functions: PowerShell-"Macros"**
  - Calling Commands with Arguments
- **Functions: PowerShell-"Macros"**
  - Starting Scripts
  - Overwriting and Deleting Aliases
- **Running Batch Files**
- **Running VBScript Files**
- **Running PowerShell Scripts**
- **Summary**

PowerShell as a Calculator

You can use the PowerShell console to execute arithmetic operations the same way you use a calculator. Just enter a math expression and PowerShell will give you the result:

```
2+4 (Enter)
  6
```

You can use all of the usual basic arithmetic operations. Even parentheses will work the same as when you use your pocket calculator:

```
(12+5) * 3  / 4.5 (Enter)
  11.3333333333333
```

Pro Tip

Parentheses play a special role in PowerShell. They always work from the inside out: the results inside the parentheses are produced before evaluating the expressions outside of the parentheses, i.e. (2*2)*2 = 4*2. For example, operations performed within parentheses have priority and ensure that multiplication operations do not take precedence over addition operations. As you'll discover in upcoming chapters, parentheses are also important when using PowerShell commands. For example, you can list the contents of sub-directories with the dir command and then determine the number of files in a folder by enclosing the dir command in parentheses.

```
(Dir $env:windir\*.exe).Count (Enter)

      12
```

In addition, there are two very similar constructions: @() and $().

@() will also execute the code inside the brackets but return the result always as an array. The previous line would have not returned the number of items if the folder contained only one or none file. This line will always count folder content reliably:

```
@(Dir $env:windir\*.exe -ErrorAction SilentlyContinue).Count (Enter)

      12
```

Note that PowerShell always uses the decimal point for numbers. Some cultures use other characters in numbers, such as a comma. PowerShell does not care. It always uses the decimal point. Using a comma instead of a decimal point will return something entirely different:

```
4,3 + 2 (Enter)

 4
 3
 2
```

The comma always creates an array. So in this example, PowerShell created an array with the elements 4 and 3. It then adds the number 2 to that array, resulting in an array of three numbers. The array content is then dumped by PowerShell into the console. So the important thing to take with you is that the decimal point is always a point and not a comma in PowerShell.

Calculating with Number Systems and Units

The next arithmetic problem is a little unusual.

```
4GB / 720MB (Enter)

 5.68888888888889
```

The example above calculates how many CD-ROMs can be stored on a DVD. PowerShell will support the common unit's kilobyte (KB), megabyte (MB), gigabyte (GB), terabyte (TB), and petabyte (PT). Just make sure you do not use a space between a number and a unit.

```
1 mb (Enter)

 1048576
```

Important

These units can be in upper or lower case – PowerShell does not care. However, whitespace characters do matter because they are always token delimiters. The units must directly follow the number and must not be separated from it by a space. Otherwise, PowerShell will interpret the unit as a new command and will get confused because there is no such command.

Take a look at the following command line:

```
12 + 0xAF (Enter)

  187
```

PowerShell can easily understand hexadecimal values: simply prefix the number with “0x”:

```
12 + 0xAF (Enter)
 45054
```

Here is a quick summary:

The example above calculates how many CD-ROMs can be stored on a DVD. PowerShell will support the common unit’s kilobyte (KB), megabyte (MB), gigabyte (GB), terabyte (TB), and petabyte (PT). Just make sure you do not use a space between a number and a unit.

Operators: Arithmetic problems can be solved with the help of operators. Operators evaluate the two values to the left and the right. For basic operations, a total of five operators are available, which are also called “arithmetic operators” **(Table2.1)**.

Brackets: Brackets group statements and ensure that expressions in parentheses are evaluated first.

Decimal point: Fractions use a point as a decimal separator (never a comma).

Comma: Commas create arrays and are irrelevant for normal arithmetic operations.

Special conversions: Hexadecimal numbers are designated by the prefix “0x”, which ensures that they are automatically converted into decimal values. If you add one of the KB, MB, GB, TB, or PB units to a number, the number will be multiplied by the unit. Whitespace characters aren’t allowed between numbers and values.

Results and formats: Numeric results are always returned as decimal values. You can use a format operator like -f if you’d like to see the results presented in a different way. This will be discussed in detail later in this book.

Operator	Description	Example	Result
+	Adds two values	5 + 4.5	9.5
		2gb + 120mb	2273312768
		0x100 + 5	261
		“Hello “ + “there” “	“Hello there”
-	Subtracts two values	5 - 4.5	0.5
		12gb - 4.5gb	8053063680
		200 - 0xAB	29
*	Multiplies two values	5 * 4.5	22.5
		4mb * 3	12582912
		12 * 0xC0	2304
		“x” * 5	“xxxxx”
/	Divides two values	5 / 4.5	1.1111111111
		1mb / 30kb	34.133333333
		0xFFAB / 0xC	5454,25
%	Supplies the rest of division	5%4.5	0.5

Table 2.1: Arithmetic operators

Executing External Commands

PowerShell can also launch external programs in very much the same way as the classic console. For example, if you want to examine the settings of your network card, you can enter the command ipconfig—it works in PowerShell the same way it does in the traditional console:

```
Ipconfig
Windows IP Configuration
Wireless LAN adapter Wireless Network Connection:

Connection-specific DNS Suffix:
Connection location IPv6 Address . : fe80::6093:8889:257e:8d1%8
IPv4 address . . . . . . . . . . . : 192.168.1.35
Subnet Mask . . . . . . . . . . . : 255.255.255.0
Standard Gateway . . . . . . . . . : 192.168.1.1

Connection-specific DNS Suffix:
Connection location IPv6 Address . : fe80::6093:8889:257e:8d1%8
```

This following command enables you to verify if a Web site is online and tells you the route the data packets are sent between a Web server and your computer:

```
Tracert powershell.com
Trace route to powershell.com [74.208.54.218] over a maximum of 30 hops:
1 12 ms 7 ms 11 ms TobiasWeltner-PC [192.168.1.1]
2 15 ms 16 ms 16 ms dslb-088-070-064-001.pools.arcor-ip.net [88.70.64.1]
3 15 ms 16 ms 16 ms han-145-254-11-105.arcor-ip.net [145.254.11.105]
(...)
17 150 ms 151 ms 152 ms vl-987.gw-ps2.slr.lxa.oneandone.net [74.208.1.134]
18 145 ms 145 ms 149 ms ratdog.info [74.208.54.218]
```

You can execute any Windows programs. Just type notepad (Enter) or explorer (Enter).

However, there’s a difference between text-based commands like ipconfig and Windows programs like Notepad. Text-based commands are executed synchronously, and the console waits for the commands to complete. Windows-based programs are executed asynchronously. Press (Ctrl)+(C) to cancel a text-based command.

Note that you can use the cmdlet Start-Process with all of its parameters when you want to launch an external program with special options. With Start-Process, you can launch external programs using different credentials; you can make PowerShell wait for Windows-based programs or control window size.

Type cls (Enter) to clear the console screen.

Starting the “Classic” Console

To temporarily switch back to the “classic” console, simply enter cmd (Enter).). Since the old console is just another text-based command, you can easily launch it from within PowerShell. To leave the old console, you can type exit (Enter). Even PowerShell can be closed by entering exit. Most text-based commands use exit to quit. Occasionally, the command quit is required in commands instead of exit.

Discovering Useful Console Commands

The cmd command can be used for just one command when you specify the parameter /c. This is useful for invoking an old console command like help. This command has no external program that you can access directly from PowerShell. It’s only available inside the classic console. Using this command will return a list of many other useful external console commands:

```
Cmd /c Help
For more information on a specific command, type HELP command-name

ASSOC Displays or modifies file extension associations.
AT Schedules commands and programs to run on a computer.
ATTRIB Displays or changes file attributes.
BREAK Sets or clears extended CTRL+C checking.
CACLS Displays or modifies access control lists (ACLs) of files.
CALL Calls one batch program from another.
CD Displays the name of or changes the current directory.
CHCP Displays or sets the active code page number.
CHDIR Displays the name of or changes the current directory.
CHKDSK Checks a disk and displays a status report.
CHKNTFS Displays or modifies the checking of disk at boot time.
CLS Clears the screen.
CMD Starts a new instance of the Windows command interpreter.
COLOR Sets the default console foreground and background colors.
COMP Compares the contents of two files or sets of files.
COMPACT Displays or alters the compression of files on NTFS partitions.
CONVERT Converts FAT volumes to NTFS. You cannot convert the current drive.
COPY Copies one or more files to another location.
DATE Displays or sets the date.
DEL Deletes one or more files.
DIR Displays a list of files and subdirectories in a directory.
DISKCOMP Compares the contents of two floppy disks.
DISKCOPY Copies the contents of one floppy disk to another.
DOSKEY Edits command lines, recalls Windows commands, and creates macros.
ECHO Displays messages, or turns command echoing on or off.
ENDLOCAL Ends localization of environment changes in a batch file.
ERASE Deletes one or more files.
EXIT Quits the CMD.EXE program (command interpreter).
FC Compares two files or sets of files, and displays the differences between them.
FIND Searches for a text string in a file or files.
FINDSTR Searches for strings in files.
FOR Runs a specified command for each file in a set of files.
```

```
FORMAT Formats a disk for use with Windows.
FTYPE Displays or modifies file types used in file extension associations.
GOTO Directs the Windows command interpreter to a labeled line in a batch program.
GRAFTABL Enables Windows to display an extended character set in graphics mode.
HELP Provides Help information for Windows commands.
IF Performs conditional processing in batch programs.
LABEL Creates, changes, or deletes the volume label of a disk.
MD Creates a directory.
MKDIR Creates a directory.
MODE Configures a system device.
MORE Displays output one screen at a time.
MOVE Moves one or more files from one directory to another directory.
PATH Displays or sets a search path for executable files.
PAUSE Suspends processing of a batch file and displays a message.
POPD Restores the previous value of the current directory saved by PUSHD.
PRINT Prints a text file.
PROMPT Changes the Windows command prompt.
PUSHD Saves the current directory then changes it.
RD Removes a directory.
RECOVER Recovers readable information from a bad or defective disk.
REM Records comments (remarks) in batch files or CONFIG.SYS.
REN Renames a file or files.
RENAME Renames a file or files.
REPLACE Replaces files.
RMDIR Removes a directory.
SET Displays, sets, or removes Windows environment variables.
SETLOCAL Begins localization of environment changes in a batch file.
SHIFT Shifts the position of replaceable parameters in batch files.
SORT Sorts input.
START Starts a separate window to run a specified program or command.
SUBST Associates a path with a drive letter.
TIME Displays or sets the system time.
TITLE Sets the window title for a CMD.EXE session.
TREE Graphically displays the directory structure of a drive or path.
TYPE Displays the contents of a text file.
VER Displays the Windows version.
VERIFY Tells Windows whether to verify that your files are written correctly to a disk.
VOL Displays a disk volume label and serial number.
XCOPY Copies files and directory trees.
```

You can use all of the above commands in your PowerShell console. To try this, pick some commands from the list. For example:

```
Cmd /c help vol
```

Important

As an added safety net, you can run PowerShell without administrator privileges when experimenting with new commands. That will protect you against mistakes as most dangerous commands can no longer be executed without administrator rights:

```
defrag c:

You must have Administrator privileges to defragment a volume.
Use an administrator command line and then run the program again.
```

Remember to start your PowerShell explicitly with administrator rights if you must use admin privileges and have enabled User Account Control.. To do this, right-click PowerShell.exe and in the context menu, select Run as Administrator.

*(Run without administrator privileges whenever possible)

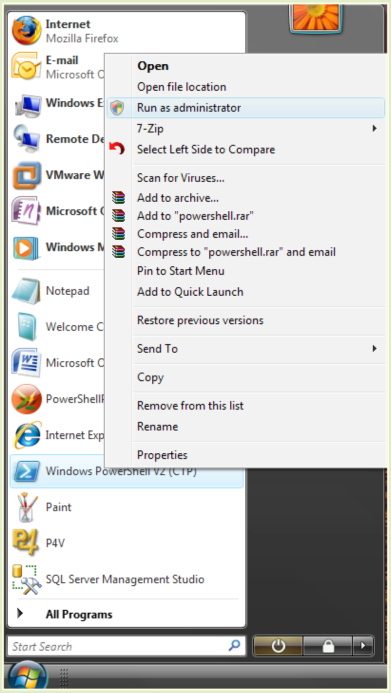


Figure 2.1: Run PowerShell as administrator

Discovering Useful Console Commands

While you can launch notepad, you cannot launch wordpad:

```
wordpad

The term "wordpad" is not recognized as a cmdlet, function,
operable program or script file. Verify the term and try again.
At line:1 char:7
+ wordpad <<<<
```

Here, PowerShell simply did not know where to find WordPad, so if the program is not located in one of the standard system folders, you can specify the complete path name like this:

```
C:\programs\Windows NT\accessories\wordpad.exe

The term " C:\program" is not recognized as a cmdlet,
function, operable program or script file. Verify the
term and try again.
At line:1 char:21
+ C:\programs\Windows <<<< NT\accessories\wordpad.exe
```

Since the path name includes whitespace characters and because PowerShell interprets them as separators, PowerShell is actually trying to start the program *C:\program*. So if path names include spaces, quote it. But that can cause another problem:

```
“C:\programs\Windows NT\accessories\wordpad.exe”

  C:\programs\Windows NT\accessories\wordpad.exe
```

PowerShell now treats quoted information as string and immediately outputs it back to you. You can prefix it with an ampersand to ensure that PowerShell executes the quoted text:

```
& “C:\programs\Windows NT\accessories\wordpad.exe”
```

Finally, WordPad starts.

Wouldn’t it be easier to switch from the current folder to the folder with the program we’re looking, and then launch the program right there?

```
Cd “C:\programs\Windows NT\accessories”
wordpad.exe

The term “wordpad” is not recognized as a cmdlet,
function, operable program or script file.
Verify the term and try again.
At line:1 char:11
+ wordpad.exe <<<< + wordpad <<<<
```

This results in another red exception because PowerShell wants a relative or absolute path. So, if you don’t want to use absolute paths like in the example above, you need to specify the relative path where “.” represents the current folder:

```
.\wordpad.exe
```

Special Places

You won’t need to provide the path name or append the file extension to the command name if the program is located in a folder that is listed in the PATH environment variable. That’s why common programs, such as regedit, notepad, powershell,or ipconfig work as-is and do not require you to type in the complete path name or a relative path.

You can put all your important programs in one of the folders listed in the environment variable Path. You can find this list by entering:

```
$env:Path

  C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\program
  Files\Softex\OmniPass;C:\Windows\System32\WindowsPowerShell\v1.0\;c
  :\program Files\Microsoft SQL Server\90\Tools\binn\;C:\program File
  s\ATI Technologies\ATI.ACE\Core-Static;C:\program Files\MakeMsi\;C:
  \program Files\QuickTime\QTSystem\
```

Important

You'll find more on variables, as well as special environment variables, in the **next chapter**.

As a clever alternative, you can add other folders containing important programs to your Path environment variables, such as:

```
$env:path += “;C:\programs\Windows NT\accessories”
wordpad.exe
```

After this change, you can launch WordPad just by entering its program name. Note that your change to the environment variable Path is valid only in the current PowerShell session. If you'd like to permanently extend Path, you will need to update the path environment variable in one of your profile scripts. Profile scripts start automatically when PowerShell starts and customize your PowerShell environment. Read more about profile scripts in Chapter 10.

Watch out for whitespace characters: If whitespace characters occur in path names, you can enclose the entire path in quotes so that PowerShell doesn't interpret whitespace characters as separators. Stick to single quotes because PowerShell “resolves” text in double quotation marks, replacing variables with their values, and unless that is what you want you can avoid it by using single quotes by default.
Specifying a path: You must tell the console where it is if the program is located somewhere else. To do so, specify the absolute or relative path name of the program.
The “&” changes string into commands: PowerShell doesn't treat text in quotes as a command. Prefix a string with “&” to actually execute it. The “&” symbol will allow you to execute any string just as if you had entered the text directly on the command line.

```
& (“note” + “pad”)
```

Tip

If you have to enter a very long path names, remember `(Tab)`, the key for automatic completion:

C:\`(Tab)`

Press `(Tab)` again and again until the suggested sub-directory is the one you are looking for. Add a “\” and press `(Tab)` once again to specify the next sub-directory.

The moment a whitespace character turns up in a path, the tab-completion quotes the path and inserts an “&” before it.

Cmdlets: PowerShell Commands

PowerShell's internal commands are called cmdlets. The mother of all cmdlets is called Get-Command:

```
Get-Command -commandType cmdlet
```

It retrieves a list of all available cmdlets, whose names always consist of an action (verb) and something that is acted on (noun). This naming convention will help you to find the right command. Let's take a look at how the system works.

```
Get-Command -verb get

CommandType Name Definition
----------- ---- ----------
Cmdlet Get-Acl Get-Acl [[-Path] <String[]>]...
Cmdlet Get-Alias Get-Alias [[-Name] <String[]...
Cmdlet Get-AuthenticodeSignature Get-AuthenticodeSignature [-...
Cmdlet Get-ChildItem Get-ChildItem [[-Path] <Stri...
Cmdlet Get-Command Get-Command [[-ArgumentList]...
Cmdlet Get-ComputerRestorePoint Get-ComputerRestorePoint [[-...
Cmdlet Get-Content Get-Content [-Path] <String[...
Cmdlet Get-Counter Get-Counter [[-Counter] <Str...
Cmdlet Get-Credential Get-Credential [-Credential]...
Cmdlet Get-Culture Get-Culture [-Verbose] [-Deb...
Cmdlet Get-Date Get-Date [[-Date] <DateTime>...
Cmdlet Get-Event Get-Event [[-SourceIdentifie...
Cmdlet Get-EventLog Get-EventLog [-LogName] <Str...
Cmdlet Get-EventSubscriber Get-EventSubscriber [[-Sourc...
Cmdlet Get-ExecutionPolicy Get-ExecutionPolicy [[-Scope...
Cmdlet Get-FormatData Get-FormatData [[-TypeName] ...
Cmdlet Get-Help Get-Help [[-Name] <String>] ...
Cmdlet Get-History Get-History [[-Id] <Int64[]>...
Cmdlet Get-Host Get-Host [-Verbose] [-Debug]...
Cmdlet Get-HotFix Get-HotFix [[-Id] <String[]>...
Cmdlet Get-Item Get-Item [-Path] <String[]> ...
Cmdlet Get-ItemProperty Get-ItemProperty [-Path] <St...
Cmdlet Get-Job Get-Job [[-Id] <Int32[]>] [-...
Cmdlet Get-Location Get-Location [-PSProvider <S...
Cmdlet Get-Member Get-Member [[-Name] <String[...
Cmdlet Get-Module Get-Module [[-Name] <String[...
Cmdlet Get-PfxCertificate Get-PfxCertificate [-FilePat...
Cmdlet Get-Process Get-Process [[-Name] <String...
Cmdlet Get-PSBreakpoint Get-PSBreakpoint [[-Script] ...
Cmdlet Get-PSCallStack Get-PSCallStack [-Verbose] [...
Cmdlet Get-PSDrive Get-PSDrive [[-Name] <String...
Cmdlet Get-PSProvider Get-PSProvider [[-PSProvider...
Cmdlet Get-PSSession Get-PSSession [[-ComputerNam...
Cmdlet Get-PSSessionConfiguration Get-PSSessionConfiguration [...
Cmdlet Get-PSSnapin Get-PSSnapin [[-Name] <Strin...
Cmdlet Get-Random Get-Random [[-Maximum] <Obje...
Cmdlet Get-Service Get-Service [[-Name] <String...
Cmdlet Get-TraceSource Get-TraceSource [[-Name] <St...
Cmdlet Get-Transaction Get-Transaction [-Verbose] [...
Cmdlet Get-UICulture Get-UICulture [-Verbose] [-D...
Cmdlet Get-Unique Get-Unique [-InputObject <PS...
Cmdlet Get-Variable Get-Variable [[-Name] <Strin...
Function Get-Verb ...
Cmdlet Get-WinEvent Get-WinEvent [[-LogName] <St...
Cmdlet Get-WmiObject Get-WmiObject [-Class] <Stri...
Cmdlet Get-WSManCredSSP Get-WSManCredSSP [-Verbose] ...
Cmdlet Get-WSManInstance Get-WSManInstance [-Resource...
```

There is an approved list of verbs that are used with cmdlet names. You can list it with Get-Verb.

You can also look up help for any cmdlet using *Get-Help*:

```
Get-Help Get-Command -detailed
```

You can easily discover cmdlets because Get-Command allows wildcards:

```
Get-Command *service* -CommandType cmdlet

CommandType Name Definition
----------- ---- ----------
Cmdlet Get-Service Get-Service [[-Name] <String...
Cmdlet New-Service New-Service [-Name] <String>...
Cmdlet New-WebServiceProxy New-WebServiceProxy [-Uri] <...
Cmdlet Restart-Service Restart-Service [-Name] <Str...
Cmdlet Resume-Service Resume-Service [-Name] <Stri...
Cmdlet Set-Service Set-Service [-Name] <String>...
Cmdlet Start-Service Start-Service [-Name] <Strin...
Cmdlet Stop-Service Stop-Service [-Name] <String...
Cmdlet Suspend-Service Suspend-Service [-Name] <Str...
```

Using Parameters

Parameters add information so a cmdlet knows what to do. Once again, Get-Help will show you which parameters are supported by any given cmdlet. For example, the cmdlet Get-ChildItem lists the contents of the current sub-directory. The contents of the current folder will be listed if you enter the cmdlet without additional parameters:

```
Get-ChildItem
```

For example, if you'd prefer to get a list of the contents of another sub-directory, you can enter the sub-directory name after the cmdlet:

```
Get-ChildItem c:\windows
```

You can use Get-Help to output full help on Get-ChildItem to find out which parameters are supported:

```
Get-Help Get-ChildItem -Full
```

This will give you comprehensive information as well as several examples. Of particular interest is the "Parameters" section that you can also retrieve specifically for one or all parameters:

```
Get-Help Get-ChildItem -Parameter *
```

-Exclude <string[]>

Omits the specified items. The value of this parameter qualifies the Path parameter. Enter a path element or pattern, such as "*.txt". Wildcards are permitted.

Required?	false
Position?	named
Default value	
Accept pipeline input	false
Accept wildcard characters?	false

-Filter <string[]>

Specifies a filter in the provider's format or language. The value of this parameter qualifies the Path parameter. The syntax of the filter, including the use of wildcards, depends on the provider. Filters are more efficient than other parameters, because the provider applies them when retrieving the objects, rather than having Windows PowerShell filter the objects after they are retrieved.

Required?	false
Position?	2
Default value	
Accept pipeline input	false
Accept wildcard characters?	false

-Force <string[]>

Allows the cmdlet to get items that cannot otherwise not be accessed by the user, such as hidden or system files. Implementation varies from provider to provider. For more information, see about_Providers. Even using the Force parameter, the cmdlet cannot override security restrictions.

Required?	false
Position?	named
Default value	
Accept pipeline input	false
Accept wildcard characters?	false

-Include <string[]>

Retrieves only the specified items. The value of this parameter qualifies the Path parameter. Enter a path element or pattern, such as “*.txt”. Wildcards are permitted.

The Include parameter is effective only when the command includes the Recurse parameter or the path leads to the contents of a directory, such as C:\Windows\*, where the wildcard character specifies the contents of the C:\Windows directory.

Required?	false
Position?	named
Default value	
Accept pipeline input	false
Accept wildcard characters?	false

-LiteralPath <string[]>

Specifies a path to one or more locations. Unlike Path, the value of LiteralPath is used exactly as it is typed. No characters are interpreted as wildcards. If the path includes escape characters, enclose it in single quotation marks. Single quotation marks tell Windows PowerShell not to interpret any characters as escape sequences.

Required?	true
Position?	1
Default value	
Accept pipeline input	true (ByPropertyName)
Accept wildcard characters?	false

-Name <string[]>

Retrieves only the names of the items in the locations. If you pipe the output of this command to another command, only the item names are sent.

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

Using Named Parameters

Named parameters really work like key-value pairs. You can specify the name of a parameter (which always starts with a hyphen), then a space, then the value you want to assign to the parameter. Let's say you'd like to list all files with the extension *.exe that are located somewhere in the folder c:\windows or in one of its sub-directories. You can use this command:

```
Get-ChildItem -path c:\windows -filter *.exe -recurse -name
```

There are clever tricks to make life easier. You don't have to specify the complete parameter name as long as you type as much of the parameter name to make it unambiguous:

```
Get-ChildItem -pa c:\windows -fi *.exe -r -n
```

Just play with it: If you shorten parameter names too much, PowerShell will report ambiguities and list the parameters that are conflicting:

```
Get-ChildItem -pa c:\windows -f *.exe -r -n

   Get-ChildItem : Parameter cannot be processed because
   the parameter name 'f' is ambiguous. Possible matches
   include: -Filter -Force.
   At line:1 char:14
```

Note

You can also turn off parameter recognition. This is necessary only rarely when the argument reads like a parameter name

```
Write-Host -BackgroundColor

Write-Host : Missing an argument for parameter
‘BackgroundColor’. Specify a parameter of type
“System.consoleColor” and try again.
At line:1 char:27
+ Write-Host -BackgroundColor <<<<
```

You can always quote the text. Or you can expressly turn off parameter recognition by typing “--”. Everything following these two symbols will no longer be recognized as a parameter:

```
Write-Host “-BackgroundColor”

-BackgroundColor

Write-Host -- -BackgroundColor

-BackgroundColor
```

Switch Parameters

Sometimes, parameters really are no key-value pairs but simple yes/no-switches. If they’re specified, they turn on a certain functionality. If they’re left out, they don’t turn on the function. For example, the parameter -recurse ensures that Get-ChildItem searches not only the -path specified sub-directories, but all sub-directories. And the switch parameter -name makes Get-ChildItem output only the names of files (as string rather than rich file and folder objects).

The help on Get-ChildItem will clearly identify switch parameters and place a “<SwitchParameter>” after the parameter name:

```
Get-Help Get-Childitem -parameter recurse

  -recurse <SwitchParameter>
  Gets the items in the specified locations and all child
  items of the locations.
  (...)
```

In all three cases, PowerShell will identify and eliminate the named arguments -recurse and -name first because they are clearly specified. The remaining arguments are “unnamed” and need to be assigned based on their position:

```
Get-ChildItem c:\windows *.exe
```

The parameter -path has the position 1, and no value has yet been assigned to it. So, PowerShell attaches the first remaining argument to this parameter.

```
-path <string[]>
Specifies a path to one or more locations. Wildcards are
permitted. The default location is the current directory (.).

Required? false
Position? 1
Standard value used <NOTE: if not specified uses
the Current location>
Accept pipeline input? true (ByValue, ByPropertyName)
Accept wildcard characters? true
```

The parameter -filter has the position 2. Consequently, it is assigned the second remaining argument. The position specification will make it easier to use a cmdlet because you don't have to specify any parameter names for the most frequently and commonly used parameters.

Here is a tip: In daily interactive PowerShell scripting, you will want short and fast commands so use aliases, positional parameters, and abbreviated parameter names. Once you write PowerShell scripts, you should not use these shortcuts. Instead, you can use the true cmdlet names and stick to fully named parameters. One reason is that scripts can be portable and not depend on specific aliases you may have defined. Second, scripts are more complex and need to be as readable and understandable as possible. Named parameters help other people better understand what you are doing.

Common Parameters

Cmdlets also support a set of generic "CommonParameters":

```
<CommonParameters>
This cmdlet supports the common parameters: -Verbose,
-Debug, -ErrorAction, -ErrorVariable, and -OutVariable.
For more information, type "get-help about_commonparameters".
```

These parameters are called "common" because they are permitted for (nearly) all cmdlets and behave the same way.

Common Parameter	Type	Description
-Verbose	Switch	Generates as much information as possible. Without this switch, the cmdlet restricts itself to displaying only essential information
-Debug	Switch	Outputs additional warnings and error messages that help programmers find the causes of errors. You can find more information in Chapter 11

Common Parameter	Type	Description
-ErrorAction	Value	Determines how the cmdlet responds when an error occurs. Permitted values: NotifyContinue: Reports error and continues (default) NotifyStop: Reports error and stops SilentContinue: Displays no error message, continues SilentStop: Displays no error message, stops Inquire: Asks how to proceed You can find more information in Chapter 11.
ErrorVariable	Value	Name of a variable in which in the event of an error information about the error is stored. You can find more information in Chapter 11.
-OutVariable	Value	Name of a variable in which the result of a cmdlet is to be stored. This parameter is usually superfluous because you can directly assign the value to a variable. The difference is that it will no longer be displayed in the console if you assign the result to a variable. $result = Get-ChildItem It will be output to the console and stored in a variable if you assign the result additionally to a variable: Get-ChildItem -OutVariable result

Table 2.3: Common parameters in effect for (nearly) all cmdlets

Aliases: Shortcuts for Commands

Cmdlet names with their verb-noun convention are very systematic, yet not always practical. In every day admin life, you will want short and familiar commands. This is why PowerShell has a built-in alias system as it comes with a lot of pre-defined aliases. This is why in PowerShell, both Windows admins and UNIX admins, can list folder contents with commands they are accustom to using. There are pre-defined “historic” aliases called “dir” and “ls” which both point to the PowerShell cmdlet Get-ChildItem.

```
Get-Command dir
CommandType Name Definition
----------- ---- ----------
Alias dir Get-ChildItem
Get-Alias -Definition Get-Childitem
CommandType Name Definition
----------- ---- ----------
Alias dir Get-ChildItem
Alias gci Get-ChildItem
Alias ls Get-ChildItem

Get-ChildItem c:\Dir c:\ls c:\
```

So, aliases have two important tasks in PowerShell:

- **Historical:** NFind and use important cmdlets by using familiar command names you know from older shells.
- **Speed:** Fast access to cmdlets using short alias names instead of longer formal cmdlet names.

Resolving Aliases

Use these lines if you'd like to know what "genuine" command is hidden in an alias:

```
$alias:Dir

  Get-ChildItem

$alias:ls

  Get-ChildItem

Get-Command Dir

Get-Command Dir
CommandType Name Definition
----------- ---- ----------
Alias dir Get-ChildItem
```

$alias:Dir lists the element Dir of the drive alias:. That may seem somewhat surprising because there is no drive called alias: in the classic console. PowerShell supports many additional virtual drives, and alias: is only one of them. If you want to know more, the cmdlet Get-PSDrive lists them all. You can also list alias: like any other drive with Dir. The result would be a list of aliases in their entirety:

```
Dir alias:
CommandType Name Definition
----------- ---- ----------
alias ac Add-Content
alias asnp Add-PSSnapin
alias clc Clear-Content
(...)
```

Get-Command can also resolve aliases. Whenever you want to know more about a particular command, you can submit it to Get-Command, and it will tell you the command type and where it is located.

You can also get the list of aliases using the cmdlet Get-Alias. You will receive a list of individual alias definitions:

```
Get-alias -name Dir
Get-ChildItem
```

This will get you all aliases pointing to the cmdlet or command you submitted to -Definition.

As it turns out, there's even a third alias for Get-ChildItem called "gci". There are more approaches to the same result. The next examples find alias definitions by doing a keyword search and by grouping:

```
Dir alias: | Out-String -Stream | Select-String "Get-ChildItem"
```

```
Count Name Group
----- ---- -----
1 Add-Content {ac}
1 Add-PSSnapin {asnp}
1 Clear-Content {clc}
1 Clear-Item {cli}
1 Clear-ItemProperty {clp}
1 Clear-Variable {clv}
3 Copy-Item {cpi, cp, copy}
1 Copy-ItemProperty {cpp}
1 Convert-Path {cvpa}
1 Compare-Object {diff}
1 Export-Alias {epal}
1 Export-Csv {epcsv}
1 Format-Custom {fc}
1 Format-List {fl}
2 ForEach-Object {foreach, %}
1 Format-Table {ft}
1 Format-Wide {fw}
1 Get-Alias {gal}
3 Get-Content {gc, cat, type}
3 Get-ChildItem {gci, ls, Dir}
1 Get-Command {gcm}
1 Get-PSDrive {gdr}
3 Get-History {ghy, h, history}
1 Get-Item {gi}
2 Get-Location {gl, pwd}
1 Get-Member {gm}
1 Get-ItemProperty {gp}
2 Get-Process {gps, ps}
1 Group-Object {group}
1 Get-Service {gsv}
1 Get-PSSnapin {gsnp}
1 Get-Unique {gu}
1 Get-Variable {gv}
1 Get-WmiObject {gwmi}
1 Invoke-Expression {iex}
2 Invoke-History {ihy, r}
1 Invoke-Item {ii}
1 Import-Alias {ipal}
1 Import-Csv {ipcsv}
3 Move-Item {mi, mv, move}
1 Move-ItemProperty {mp}
1 New-Alias {nal}
2 New-PSDrive {ndr, mount}
1 New-Item {ni}
1 New-Variable {nv}
1 Out-Host {oh}
1 Remove-PSDrive {rdr}
6 Remove-Item {ri, rm, rmdir, del...}
2 Rename-Item {rni, ren}
1 Rename-ItemProperty {rnp}
```

```
1 Remove-ItemProperty {rp}
1 Remove-PSSnapin {rsnp}
1 Remove-Variable {rv}
1 Resolve-Path {rvpa}
1 Set-Alias {sal}
1 Start-Service {sasv}
1 Set-Content {sc}
1 Select-Object {select}
1 Set-Item {si}
3 Set-Location {sl, cd, chdir}
1 Start-Sleep {sleep}
1 Sort-Object {sort}
1 Set-ItemProperty {sp}
2 Stop-Process {spps, kill}
1 Stop-Service {spsv}
2 Set-Variable {sv, set}
1 Tee-Object {tee}
2 Where-Object {where, ?}
2 Write-Output {write, echo}
2 Clear-Host {clear, cls}
1 Out-Printer {lp}
1 Pop-Location {popd}
1 Push-Location {pushd}
```

Creating Your Own Aliases

To temporarily switch back to the “classic” console, simply enter cmd (Enter).). Since the old console is just another text-based command, you can easily launch it from within PowerShell. To leave the old console, you can type exit (Enter). Even PowerShell can be closed by entering exit. Most text-based commands use exit to quit. Occasionally, the command quit is required in commands instead of exit.

```
Edit
Set-Alias edit notepad.exe
Edit
```

Edit typically launches the console-based Editor program. You can press `(Alt)+(F)` and then `(X)` to exit without completely closing the console window.

If you create a new alias called “Edit” and set it to “notepad.exe”, the command Edit will be re-programmed. The next time you enter it, PowerShell will no longer run the old Editor program, but the Notepad.

```
$alias:edit
```

Removing or Permanently Keeping an Alias

How do you remove aliases? You don't. All new aliases are discarded as soon as you exit PowerShell. All of your own aliases will be gone the next time you start PowerShell. "Built-in" aliases like "dir" and "cd" will still be there.

Try these options if you'd like to keep your own aliases permanently:

- **Manually each time:** Set your aliases after every start manually using Set-Alias. That is, of course, rather theoretical.
- **Automated in a profile:** Let your alias be set automatically when PowerShell starts: add your aliases to a start profile. You'll learn how to do this in **Chapter 10**.
- **Import and export:** You can use the built-in import and export function for aliases.

For example, if you'd like to export all currently defined aliases as a list to a file, enter:

```
Export-Alias
```

Because you haven't entered any file names after Export-Alias, the command will ask you what the name are under which you want to save the list. Type in:

```
alias1 (Enter)
```

The list will be saved. You can look at the list afterwards and manipulate it. For example, you might want the list to include a few of your own alias definitions:

```
Notepad alias1
```

You can import the list to activate the alias definitions:

```
Import-Alias alias1

    Import-Alias : Alias not allowed because an alias with the
    name "ac" already exists.
    At line:1 char:13
    + Import-Alias <<<< alias1 + Get-ChildItem <<<< -pa c:\windows
```

Import-Alias will notify you that it cannot create some aliases of the list because these aliases already exist. Specify additionally the option -Force to ensure that Import-Alias overwrites existing aliases:

```
Import-Alias alias1 -Force
```

Important

You can add the Import-Alias instruction to your start profile and specify a permanent path to the alias list. This will make PowerShell automatically read this alias list when it starts. Later, you can add new aliases. Then, it will suffice to update the alias list with Export-Alias and to write over the old file. This is one way for you to keep your aliases permanently.

Overwriting and Deleting Aliases

You can overwrite aliases with new definitions any time as long as an alias is not write-protected. Just redefine the alias with the cmdlet Set-Alias. Use this command if you'd like to remove an alias completely and don't want to wait until you end PowerShell:

```
Del alias: edit
```

This instruction deletes the "Edit" alias. Here, the uniform provider approach becomes evident. The very same "Del" command will allow you to delete files and sub-directories in the file system as well. Perhaps you're already familiar with the command from the classic console:

```
Del C:\garbage.txt
```

Pro Tip

Here is an example that finds all aliases that point to no valid target, which is a great way of finding outdated or damaged aliases:

```
Get-Alias | ForEach-Object {
if (!(Get-Command $_.Definition -ea SilentlyContinue)) {$_}}
```

Functions: PoweShell-"Macros"

Aliases are simple shortcuts to call commands with another name (shortcut names), or to make the transition to PowerShell easier (historic aliases). However, the arguments of a command can never be included in an alias. You will need to use functions if you want that.

Calling Commands with Arguments

If you find yourself using the command ping frequently to verify network addresses, you may want to make this easier by creating a shortcut that not only calls ping.exe, but also appends standard arguments to it. Let’s see how you can automate this call:

```
Ping -n 1 -w 100 10.10.10.10
```

Aliases won’t work here because they can’t specify command arguments. Functions can:

```
function quickping { ping -n 1 -w 100 $args }
quickping 10.10.10.10

Pinging 10.10.10.10 with 32 bytes of data:
Reply from 10.10.10.10: bytes=32 time<1ms TTL=128
Ping statistics for 10.10.10.10:
Packets: Sent = 1, Received = 1, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
Minimum = 0ms, Maximum = 0ms, Average = 0ms

Set-Alias qp quickping
qp 10.10.10.10
Pinging 10.10.10.10 with 32 bytes of data:
Reply from 10.10.10.10: bytes=32 time<1ms TTL=128
Ping statistics for 10.10.10.10:
Packets: Sent = 1, Received = 1, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

Unlike alias definitions, functions can run arbitrary code that is placed in brackets. Any additional information a user submitted to the function can be found in $args if you don’t specify explicit parameters. $args is an array and holds every piece of extra information submitted by the caller as separate array elements. You’ll read more about functions later.

Invoking Files and Scripts

To run files (like documents or scripts), PowerShell uses the same rules that apply to executables: either, you specify an absolute or relative path, or the file needs to be located in one of the special trustworthy folders defined in the Path environment variable.

```
# Save information on all running processes to HTML file
# (lasts several seconds):
Get-Process | ConvertTo-Html | Out-File test.htm

# File cannot be opened directly:
test.htm

The term “test.htm” is not recognized as a cmdlet, function,
operable program, or script file. Verify the term and try again.
At line:1 char:8
+ test.htm <<<<

# Specify a relative or absolute path name:
.\test.htm
```

Starting Scripts

Scripts and batch files are pseudo-executables. The script itself is just a plain text file, but it can be run by its associated script interpreter.

Running Batch Files

Batch files are text files with the extension “.bat”. They may include all the commands allowed in a classic cmd.exe console. When a batch file is opened, the classic console immediately starts to execute the commands it contains. Let’s check it out. First, create this test file:

```
Notepad ping.bat
```

Now enter this text:

```
@echo off
echo An attacker can do dangerous things here
pause
Dir %windir%
pause
Dir %windir%\system
```

Save the text and close Notepad. Your batch file is ready for action. Try to launch the batch file by entering its name:

```
Ping
```

The batch file won’t run. Because it has the same name and you didn’t specify any IP address or Web site address, the ping command spits out its internal help message. If you want to launch your batch file, you’re going to have to specify either the relative or absolute path name.

```
.\ping
```

Your batch file will open and then immediately runs the commands it contains.

PowerShell has just defended a common attack. If you were using the classic console, you would have been tricked by the attacker. Switch over to the classic console to see for yourself:

```
Cmd
Ping 10.10.10.10

An attacker can do dangerous things here
Press any key . . .
```

If an attacker had smuggled a batch file named “ping.bat” into your current folder, then the ping command, harmless though it might seem, could have had catastrophic consequences. A classic console doesn’t distinguish between files and commands. It will look first in the current folder, find the batch file, and execute it immediately. Such a mix-up will never happen in the PowerShell console. So, return to your much-safer PowerShell environment:

```
Exit
```

Running VBScript Files

VBScript is another popular automation language as its scripts are tagged with the file extension “.vbs”. What we have just discussed about batch files also applies to these scripts:

```
Notepad test.vbs
```

Enter this VBScript code in Notepad and save it as *test.vbs:*

```
result = InputBox(“Enter your name”)
WScript.Echo result
```

Next, run the script:

```
Cscript.exec:\samples\test.vbs (Enter)
```

Important

The script opens a small dialog window and asks for some information. The information entered into the dialog is then output to the console where PowerShell can receive it. This way, you can easily merge VBScript logic into your PowerShell solutions. You can even store the results into a variable and process it inside PowerShell:

```
$name = cscript.exe c:\samples\test.vbs

“Your name is $name”
```

Important

If you do not get back the name you entered into the dialog, but instead the VBScript copyright information, then the VBScript interpreter has output the copyright information first, which got in the way. The safest way is to turn off the copyright message explicitly:

```
$name = cscript.exe //NOLOGO c:\samples\test.vbs
```

You can also generally turn off VBScript logos. Try calling wscript.exe to open the settings dialog, and turn off the logo.

Running PowerShell Scripts

PowerShell has its own script files with the file extension “.ps1”. While you will learn much more about PowerShell scripts in **Chapter 10**, you already know enough to write your first script. Use the Windows editor to create and open your first script:

```
Notepad $env:temp\test.ps1
```

You can now enter any PowerShell code you want, and save the file. Once saved, you can also open your script with more sophisticated and specialized script editors. PowerShell comes with an editor called PowerShell ISE, and here is how you'd open the file you created with Notepad:

```
Ise $env:temp\test.ps1
```

Try to run your script after you've created it:

```
.\test.ps1

 File “C:\Users\UserA\test.ps1” cannot be loaded because the
 execution of scripts is disabled on this system. Please see
 “get-help about_signing” for more details.
 At line:1 char:10
 + .\test.ps1 <<<<
```

You'll probably receive an error message similar to the one in the above example. All PowerShell scripts are initially disabled. You need to allow PowerShell to execute scripts first. This only needs to be done once:

```
Set-ExecutionPolicy RemoteSigned -Scope CurrentUser
```

This grants permission to run locally stored PowerShell scripts. Scripts from untrusted sources, such as the Internet, will need to carry a valid digital signature or else they won't run. This is to protect you from malicious scripts, but if you want to, you can turn this security feature off. Replace RemoteSigned with Bypass. The implications of signatures and other security settings will be discussed in Chapter 10. For now, the line above is enough for you to experiment with your own PowerShell scripts. To restore the original setting, set the setting to Undefined:

```
Set-ExecutionPolicy Undefined -Scope CurrentUser
```

To get a complete picture, also try using the -List parameter with Get-ExecutionPolicy:

```
Get-ExecutionPolicy -List
Scope            ExecutionPolicy
-----            ---------------
MachinePolicy    Undefined
UserPolicy       Undefined
Process          Undefined
CurrentUser      RemoteSigned
LocalMachine     Restricted
```

You now see all execution policies. The first two are defined by Group Policy so a corporation can centrally control execution policy. The scope “Process” refers to your current session only. So, you can use this scope if you want to only temporarily change the execution policy. No other PowerShell session will be affected by your change. “CurrentUser” will affect only you, but no other users. That’s how you can change this scope without special privileges. “LocalMachine,” which is the only scope available in PowerShell v.1, will affect any user on your machine. This is the perfect place for companies to set initial defaults that can be overridden. The default setting for this scope is “Restricted.”

The effective execution policy is the first policy from top to bottom in this list that is not set to “Undefined.” If all policies are set to “Undefined,” then scripts are prohibited.

Note: To turn off signature checking altogether, you can set the execution policy to “Bypass.” This can be useful if you must run scripts regularly that are stored on file servers outside your domain. Otherwise, you may get security warnings and confirmation dialogs. Always remember: execution policy exists to help and protect you from potentially malicious scripts. If you are confident you can safely identify malicious scripts, then nothing is wrong by turning off signature checking. However, we recommend not using the “Bypass” setting if you are new to PowerShell.

Invoking Files and Scripts

The PowerShell console can run all kinds of commands interactively. You simply enter a command and the console will return the results.

Cmdlets are PowerShell’s own internal commands. A cmdlet name is always composed of a verb (what it does) and a noun (where it acts upon).

To find a particular command, you can either guess or use Get-Command. For example, this will get you a list if you wanted to find all cmdlets dealing with event logs:

```
Get-Command -Noun EventLog
```

Search for the verb “Stop” to find all cmdlets that stop something:

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

You can also use wildcards. This will list all cmdlets with the keyword “computer”:

```
Get-Command *computer* -commandType cmdlet
```

Once you know the name of a particular cmdlet, you can use Get-Help to get more information. This function will help you view help information page by page:

```
Get-Help Stop-Computer
Help Stop-Computer -examples
Help Stop-Computer -parameter *
```

Cmdlets are just one of six command types you can use to get work done:

- Alias: Shortcuts to other commands, such as dir or ls
- Function: “Macros” that run code and resemble “self-made” new commands

- Cmdlet: Built-in PowerShell commands
- Application: External executables, such as ipconfig, ping or notepad
- PowerShell scripts: Files with extension *.ps1 which can contain any valid PowerShell code
- Other files: Batch files, VBScript script files, or any other file associated with an executable

If commands are ambiguous, PowerShell will stick to the order of that list. So, since the command type “Alias” is at the top of that list, if you define an alias like “ping”, it will be used instead of ping.exe and thus can override any other command type.

Chapter 3. Variables

It is time to combine commands whenever a single PowerShell command can't solve your problem. One way of doing this is by using variables. PowerShell can store results of one command in a variable and then pass the variable to another command. In this chapter, we'll explain what variables are and how you can use them to solve more complex problems.

Topics Covered:

· **Personal Variables**

  · Selecting Variable Names

  · Assigning and Returning Values

  · Assigning Multiple Variable Values

  · Exchanging the Contents of Variables

  · Assigning Different Values to Several Variables ·

Listing Variables

  · Write-Protecting Variables: Creating Constants

  · Variables with Description

· **"Automatic" PowerShell Variables**

· **Environment Variables**

  · Reading Environment Variables

  · Searching for Environment Variables

  · Modifying Environment Variables

  · Permanent Modifica

tions of Environment Variables

· **Scope of Variables**

  · **Automatic Restriction**

  · **Changing Variable Visibility**

  · **Setting Scope**

· **Scope of Variables**

  · **Strongly Typing**

  · **The Advantages of Specialized Types**

· **Variable Management: Behind the Scenes**

  · Modification of Variable Options

  · Write Protecting Variables

  · Examining Strongly Typed Variables

  · Validating Variable Contents

· **Summary**

Personal Variables

Variables store pieces of information. This way, you can first gather all the information you may need and store them in variables. The following example stores two pieces of information in variables and then calculates a new result:

```
# Create variables and assign to values
$amount = 120
$amount = 0.19

# Calculate
$result = $amount * $VAT

# Output result
$result

  22.8

# Replace variables in text with values:
$text = “Net amount $amount matches gross amount $result”$amount
$text

  Net amount 120 matches gross amount 142.8
```

Of course, you can have hard-coded the numbers you multiplied. However, variables are the prerequisite for reusable code. By assigning your data to variables, you can easily change the information, either by manually assigning different values to your variables or by assigning user-defined values to your variables. By simply replacing the first two lines, your script can interactively ask for the variable content:

```
[Int]$amount = “Enter amount of money”
[Double]$VAT = “Enter VAT rate”
```

Note that I strongly-typed the variables in this example. You will hear more about variable typing later in that character , but whenever you use Read-Host or another method that accepts user input, you have to specify the variable data type or else PowerShell will treat your input as simple string. Simple text is something very different from numbers and you cannot calculate with pieces of text.

PowerShell creates new variables automatically so there is no need to specifically “declare” variables. Simply assign data to a variable. The only thing you do need to know is that variable names are always prefixed with a “$” to access the variable content.

You can then output the variable content by entering the variable name or you can merge the variable content into strings. Just make sure to use double-quotes to do that. Single-quoted text will not expand variable values.

Selecting Variable Names

You are free to call the variable anything you like – as long as the name is not causing misunderstandings. Variable names are always case-insensitive.

```
${#this is a strange variable name} = 12
${#this is a strange variable name}

  12
```

Assigning and Returning Values

The assignment operator "=" assigns a value to a variable. You can assign almost anything to a variable, even complete command results:

```
# Temporarily store results of a cmdlet:
$listing = Get-ChildItem c:\
$listing

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

Mode LastWriteTime Length Name
(...)

# Temporarily store the result of a legacy external command:
$result = ipconfig
$result

Windows IP Configuration

Ethernet adapter LAN Connection:

Media state

. . . . . . . . . . . . : Medium disconnected
Connection-specific DNS Suffix:
Ethernet adapter LAN Connection 2:

Media state

. . . . . . . . . . . . : Medium disconnected
Connection-specific DNS Suffix:
Wireless LAN adapter wireless network connection:

Media state

. . . . . . . . . . . . : Medium disconnected
```

Assigning Multiple Variable Values

If you'd like, you can use the assignment operator to assign values to multiple variables at the same time:

```
# Populate several variables with the same value in one step:
$a = $b = $c = 1 = Get-ChildItem c:\
$a

   1


$b

   1


$c

   1
```

Exchanging the Contents of Variables

Now and then you might want to exchange the contents of two variables. In traditional programming languages, that would require several steps:

```
$Value1 = 10
$Value2 = 20
$Temp = $Value1
$Value1 = $Value2
$Value2 = $Temp
```

With PowerShell, swapping variable content is much easier because you can assign multiple values to multiple variables. Have a look:

```
# Exchange variable values:
$Value1 = 10; $Value2 = 20
$Value1, $Value2 = 10,20
$Value1, $Value2 = $Value1, $Value2
```

Listing Variables

PowerShell keeps a record of all variables, which is accessible via a virtual drive called variable:. Here is how you see all currently defined variables:

```
Dir variable:
```

Aside from your own personal variables, you'll see many more. PowerShell also defines variables and calls them “automatic variables.” You'll learn more about this soon.

Finding Variables

Using the variable: virtual drive can help you find variables. If you'd like to see all the variables containing the word “Maximum,” try this:

```
Dir variable:*maximum*

Name                   Value
----                   -----
MaximumErrorCount      256
MaximumVariableCount   4096
MaximumFunctionCount   4096
MaximumAliasCount      4096
MaximumDriveCount      4096
MaximumHistoryCount    1000
```

The solution isn't quite so simple if you'd like to know which variables currently contain the value 20. It consists of several commands piped together.

```
dir variable: | Out-String -stream | Select-String “ 20 “

   value2 20
   $ 20
```

Here, the output from Dir is passed on to Out-String, which converts the results of Dir into string. The parameter -Stream ensures that every variable supplied by Dir is separately output as string. Select-String selects the lines that include the desired value, filtering out the rest. White space is added before and after the number 20 to ensure that only the desired value is found and not other values that contain the number 20 (like 200).

Verify Whether a Variable Exists

Using the cmdlet Test-Path, you can verify whether a certain file exists. Similar to files, variables are stored in their own “drive” called variable: and every variable has a path name that you can verify with Test-Path. You can use this technique to find out whether you are running PowerShell v1 or v2:

```
# Verify whether the variable $psversiontable exists which is present only in PS v2:
Test-Path variable:\psversiontable

   True

# Use this information to check for PS v2
If (Test-Path variable:psversiontable) {
‘You are running PowerShell v2’
} else {
‘You are running PowerShell v1 and should update to v2’
}

  False
```

Deleting Variables

PowerShell will keep track of variable use and remove variables that are no longer used so there is no need for you to remove variables manually. If you'd like to delete a variable immediately, again, do exactly as you would in the file system:

```
# create a test variable:
$test = 1
   True

# verify that the variable exists:
Dir variable:\te*

# delete variable:
del variable:\test

# variable is removed from the listing:
Dir variable:\te*
```

Using Special Variable Cmdlets

To manage your variables, PowerShell provides you with the five separate cmdlets listed in **Table 3.1**. Two of the five cmdlets offer substantially new options:

New-Variable enables you to specify options, such as a description or write protection. This makes a variable into a constant. Set-Variable does the same for existing variables.
Get-Variable enables you to retrieve the internal PowerShell variables store.

Cmdlet	Description	Example
Clear-Variable	Clears the contents of a variable, but not the variable itself. The subsequent value of the variable is NULL (empty). If a data or object type is specified for the variable, by using Clear-Variable the type of the objected stored in the variable will be preserved.	Clear-Variable a same as: *$a = $null*
Get-Variable	Gets the variable object, not the value in which the variable is stored.	Get-Variable a
New-Variable	Creates a new variable and can set special variable options.	New-Variable value 12
Remove-Variable	Deletes the variable, and its contents, as long as the variable is not a constant or is created by the system.	Remove-Variable a same as: *del variable:\a*
Set-Variable	Resets the value of variable or variable options, such as a description and creates a variable if it does not exist.	Set-Variable a 12 same as: *$a = 12*

Table 3.1: Cmdlets for managing variables

Write-Protecting Variables: Creating Constants

Constants store a constant value that cannot be modified. They work like variables with a write-protection.

PowerShell doesn’t distinguish between variables and constants. However, it does offer you the option of write-protecting a variable. In the following example, the write-protected variable $test is created with a fixed value of 100. In addition, a description is attached to the variable.

```
# Create new variable with description and write-protection:
New-Variable test -value 100 -description `
“test variable with write-protection” -option ReadOnly
$test

 100

# Create new variable with description and write-protection:
$test = 200

  The variable “test” cannot be overwritten since it is a
  constant or read-only.
  At line:1 char:6
  + $test <<<< = 200
```

The variable is now write-protected and its value may no longer be changed. You’ll receive an error message if you try it anyway. Because the variable is write-protected, it behaves like a read-only file. You’ll have to specify the parameter -Force to delete it:

```
del variable:\test -force
$test = 200
```

As you just saw, a write-protected variable can still be modified by deleting it and creating a new copy of it. If you need stronger protection, you can create a variable with the Constant option. Now, it can neither be modified nor deleted. Only when you quit PowerShell are constants removed. Variables with the Constant option may only be created with New-Variable. If a variable already exists, you cannot make it constant anymore because you’ll get an error message:

```
#New-Variable cannot write over existing variables:
New-Variable test -value 100 -description `
“test variable with copy protection” -option Constant

  New-Variable : A variable named “test” already exists.
  At line:1 Char:13
  + New-Variable <<<< test -value 100 -description
  “test variable with copy protection” -option Constant
```

```
# If existing variable is deleted, New-Variable can create
# a new one with the "Constant" option:
del variable:\test -force
New-Variable test -value 100 -description `
"test variable with copy protection" `
-option Constant

# variables with the "Constant" option may neither be
# modified nor deleted:
del variable:\test -force

  Remove-Item : variable "test" may not be removed since it is a
  constant or write-protected. If the variable is write-protected,
  carry out the process with the Force parameter.
  At line:1 Char:4
  + del <<<< variable:\test -force
```

You can overwrite an existing variable by using the -Force parameter of New-Variable if the existing variable wasn't created with the Constant option. Variables of the constant type are unchangeable once they have been created and -Force does not change this:

```
# Parameter -force overwrites existing variables if these do not
# use the "Constant" option:
New-Variable test -value 100 -description "test variable" -force

  New-Variable : variable "test" may not be removed since it is a
  constant or write-protected.
  At line:1 char:13
  + New-Variable <<<< test -value 100 -description "test variable"

# normal variables may be overwritten with -force without difficulty.
$available = 123
New-Variable available -value 100 -description "test variable" -force
```

Variables with Description

Variables can have an optional description to help you keep track of what the variable was intended for. However, this description appears to be invisible:

```
# Create variable with description:
New-Variable myvariable -value 100 -description "test variable" -force

# Variable returns only the value:
$myvariable

  100
```

```
# Dir and Get-Variable also do not deliver the description:
Dir variable:\myvariable

  Name        Value
  ----        -----
  myvariable 100
  Get-Variable myvariable
  Name        Value
  ----        -----
  myvariable 100
```

“Automatic” PowerShell Variables

PowerShell also uses variables for internal purposes and calls those “automatic variables.” These variables are available right after you start PowerShell since PowerShell has defined them during launch. The drive variable: provides you with an overview of all variables:

```
Get-Childitem variable:

  Name            Value
  ----            -----
  Error           {}
  DebugPreference SilentlyContinue
  PROFILE         C:\Users\Tobias Weltner\Documents\WindowsPowerShell\Micro...
  HOME            C:\Users\Tobias Weltner
  (...)
```

You can show their description to understand the purpose of automatic variables:

```
Get-Childitem variable: | Sort-Object Name |
Format-Table Name, Description -AutoSize -Wrap
```

Use Get-Help to find out more

Important

PowerShell write protects several of its automatic variables. While you can read them, you can't modify them. This makes sense because information, like the process-ID of the PowerShell console or the root directory, must not be modified.

```
$pid = 12

 Cannot overwrite variable "PID" because it is read-only or constant.
 At line:1 char:5
 + $pid <<<< = 12
```

A little later in this chapter, you'll find out more about how write-protection works. You'll then be able to turn write-protection on and off for variables that already exist. However, don't do this for automatic variables because PowerShell may crash. One reason is because PowerShell continually modifies some variables. If you set them to read-only, PowerShell may stop and not respond to any inputs.

Environment Variables

There is another set of variables maintained by the operating system: environment variables.

Working with environment variables in PowerShell is just as easy as working with internal PowerShell variables. All you need to do is add the prefix to the variable name: *env:*.

Variables with Description

You can read the location of the Windows folder of the current computer from a Windows environment variable:

```
$env:windir

  C:\Windows
```

By adding env:, you've told PowerShell not to look for the variable windir in the default PowerShell variable store, but in Windows environment variables. In other word, the variable behaves just like any other PowerShell variable. For example, you can embed it in some text:

```
"The Windows folder is here: $env:windir"
The Windows folder is here: C:\Windows
```

You can just as easily use the variable with commands and switch over temporarily to the Windows folder like this:

```
$env:windir

  C:\Windows
```

By adding env:, you've told PowerShell not to look for the variable windir in the default PowerShell variable store, but in Windows environment variables. In other word, the variable behaves just like any other PowerShell variable. For example, you can embed it in some text:

```
# save in current folder:
Push-Location

# change to Windows folder
cd $env:windir
Dir

# change back to initial location after executed task
Pop-Location
```

Searching for Environment Variables

PowerShell keeps track of Windows environment variables and lists them in the env: virtual drive. So, if you'd like an overview of all existing environment variables, you can list the contents of the *env:* drive:

```
Get-Childitem env:

  Name            Value
  ----            -----
  Path            C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\
  TEMP            C:\Users\TOBIAS~1\AppData\Local\Temp
  ProgramData     C:\ProgramData
  PATHEXT         .COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC;.4mm
  ALLUSERSPROFILE C:\ProgramData
  PUBLIC          C:\Users\Public
  OS              Windows_NT
  USERPROFILE     C:\Users\Tobias Weltner
  HOMEDRIVE       C:
  (...)
```

You'll be able to retrieve the information it contains when you've located the appropriate environment variable and you know its name:

```
$env:windir

  C:\Users\Tobias Weltner
```

Modifying Environment Variables

You can modify environment variables by simply assigning new variables to them. Modifying environment variables can be useful to change the way your machine acts. For example, all programs and scripts located in a folder that is listed in the “PATH” environment variable can be launched by simply submitting the file name. You no longer need to specify the complete path or a file extension.

The next example shows how you can create a new folder and add it to the PATH environment variable. Any script you place into that folder will then be accessible simply by entering its name:

```
# Create a special folder:
md c:\myTools

# Create and example script in this folder:
“ ‘Hello!’ “ > c:\myTools\sayHello.ps1

# Typically, you would have to specify a qualified path name:
C:\myTools\sayHello.ps1
Hello!

# The folder is now added to the path environment:
$env:path += “;C:\myTools”

# All scripts and commands in this folder can be launched by entering their name now:
sayHello
Hello!
```

Permanent Modifications of Environment

By default, PowerShell works with the so-called “process” set of environment variables. They are just a copy and only valid inside your current PowerShell session (and any programs you launch from it). Changes to these environment variables will not persist and are discarded once you close your PowerShell session.

You have two choices if you need to make permanent changes to your environment variables. You can either make the changes in one of your profile scripts, which get executed each time you launch PowerShell (then your changes are effective in any PowerShell session but not outside) or you can use sophisticated .NET methods directly to change the underlying original environment variables (in which case the environment variable change is visible to anyone, not just PowerShell sessions). This

```
$oldValue = [environment]::GetEnvironmentvariable(“Path”, “User”)
$newValue = “;c:\myTools”
[environment]::SetEnvironmentvariable(“Path”, $newValue, “User”)
```

Note

Access to commands of the .NET Framework as shown in this example will be described in depth in **Chapter 6.**

When you close and restart PowerShell, the Path environment variable will now retain the changed value. You can easily check this:

```
$env:Path
```

The permanent change you just made applies only to you, the logged-on user. If you'd like this change to be in effect for all computer users, you can replace the "User" argument by "Machine." You will need full administrator privileges to do that.

You should only change environment variables permanently when there is no other way. For most purposes, it is completely sufficient to change the temporary process set from within PowerShell. You can assign it the value of $null to remove a value.

Scope of Variables

PowerShell variables can have a "scope," which determines where a variable is available. PowerShell supports four special variable scopes: global, local, private, and script. These scopes allow you to restrict variable visibility in functions or scripts.

Automatic Restriction

Typically, a script will use its own variable scope and isolate all of its variables from the console. So when you run a script to do some task, it will not leave behind any variables or functions defined by that script once the script is done.

Changing Variable Visibility

You can change this default behavior in two different ways. One is to call the script "dot-sourced": type in a dot, then a space, and then the path to the script. Now, the script's own scope is merged into the console scope. Every top-level variables and functions defined in the script will behave as if they had been defined right in the console. So when the script is done, it will leave behind all such variables and functions.

Dot-sourcing is used when you want to (a) debug a script and examine its variables and functions after the script ran, and (b) for library scripts whose purpose is to define functions and variables for later use. The profile script, which launches automatically when PowerShell starts, is an example of a script that always runs dot-sourced. Any function you define in any of your profile scripts will be accessible in your entire PowerShell session – even though the profile script is no longer running.

Setting Scope

While the user of a script can somewhat control scope by using dot-sourcing, a script developer has even more control over scope by prefixing variable and function names. Let's use the scope modifiers private, local, script, and global.

Scope Allocation	Description
$private:test = 1	The variable exists only in the current scope. It cannot be accessed in any other scope.
$local:test = 1	Variables will be created only in the local scope. That’s the default for variables that are specified without a scope. Local variables can be read from scopes originating from the current scope, but they cannot be modified.
$script:test = 1	This scope represents the top-level scope in a script. All functions and parts of a script can share variables by addressing this scope.
$global:test = 1	

Table 3.3: Variable scopes and validity of variables

Script blocks represent scopes in which variables and functions can live. The PowerShell console is the basic scope (global scope). Each script launched from the console creates its own scope (script scope) unless the script is launched “dot-sourced.” In this case, the script scope will merge with the caller’s scope.

Functions again create their own scope and functions defined inside of other functions create additional sub-scopes.

```
$test = 1
$local:test

  1

$script:test = 12
$global:test

  12

$private:test

  12
```

Differences become evident only once you create additional scopes, such as by defining a function:

```
# Define test function:
Function test { “variable = $a”; $a = 1000 }

# Create variable in console scope and call test function:
$a = 12
Test

variable

# Check variable for modifications after calling test function in console scope:
$a

  12
```

Only when you create a completely new variable by using $private: is it in fact private. If the variable already existed, PowerShell will not reset the scope. To change scope of an existing variable, you will need to first remove it and then recreate it: *Remove-Variable* a would remove the variable $a. Or, you can manually change the variable options: (*Get-Variable a*). *Options* = *“Private.”* You can change a variable scope back to the initial default “local” by assigning (*Get-Variable a*)*.Options* = *“None.”*

Variable Types and “Strongly

Variables by default are not restricted to a specific data type. Instead, when you store data in a variable, PowerShell will automatically pick a suitable data type for you. To find out what data types really are, you can explore data types. Call the method *GetType()*. It will tell you the data type PowerShell has picked to represent the data:

```
(12).GetType().Name

Int32

(1000000000000).GetType().Name

Int64

(12.5).GetType().Name

Double

(12d).GetType().Name

Decimal

(“H”).GetType().Name

String

(Get-Date).GetType().Name

DateTime
```

PowerShell will by default use primitive data types to store information. If a number is too large for a 32-bit integer, it switches to 64-bit integer. If it’s a decimal number, then the Double data type best represents the data. For text information, PowerShell uses the String data type. Date and time values are stored in DateTime objects.

This process of automatic selection is called “weak typing,” and while easy, it’s also often restrictive or risky. Weakly typed variables will happily accept anything, even wrong pieces of information. You can guarantee that the variable gets the information you expected by strongly typing a variable — or else will throw an exception that can alarm you.

Also, PowerShell will not always pick the best data type. Whenever you specify text, PowerShell will stick to the generic string type. If the text you specified was really a date or an IP address, then there are better data types that will much better represent dates or IP addresses.

So, in practice, there are two important reasons for you to choose the data type yourself:

- **Type safety:** If you have assigned a type to a variable yourself, then the type will be preserved no matter what and will never be automatically changed to another data type. You can be absolutely sure that a value of the correct type is stored in the variable. If someone later on wants to mistakenly assign a value to the variable that doesn’t match the originally chosen type, this will cause an exception.
- **Special variable types:** When automatically assigning a variable type, PowerShell will choose from generic variable types like *Int32* or *String*. Often, it’s much better to store values in a specialized and more meaningful variable type like *DateTime*.

Strongly Typing

You can enclose the type name in square brackets before the variable name to assign a particular type to a variable. For example, if you know that a particular variable will hold only numbers in the range 0 to 255, you can use the *Byte* type:

```
[Byte]$flag = 12
$flag.GetType().Name

  Byte
```

The variable will now store your contents in a single byte, which is not only very memory-efficient, but it will also raise an error if a value outside the range is specified:

```
$flag = 300

 The value “300” cannot be converted to the type “System.Byte”.
 Error: “The value for an unsigned byte was too large or too small.”
 At line:1 char:6
 + $flag <<<< = 300
```

The Advantages of Specialized Types

If you store a date as String, you’ll have no access to special date functions. Only DateTime objects offer all kinds of methods to deal with date and time information. So, if you’re working with date and time information, it’s better to store it explicitly as DateTime:

```
$date = “November 12, 2004”
$date

   November 12, 2004
```

If you store a date as String, then you'll have no access to special date functions. Only DateTime objects make them available. So, if you're working with date and time indicators, it's better to store them explicitly as DateTime:

```
[datetime] = “November 12, 2004”
$date

  Friday, November 12, 2004 00:00:00
```

Now, since the variable converted the text information into a specific DateTime object, it tells you the day of the week and also enables specific date and time methods. For example, a DateTime object can easily add and subtract days from a given date. This will get you the date 60 days from the date you specified:

```
$date.AddDays(60)

 Tuesday, January 11, 2005 00:00:00
```

PowerShell supports all.NET data types. XML documents will be much better represented using the XML data type then the standard String data type:

```

# PowerShell stores a text in XML format as a string:
$t = “<servers><server name=’PC1’ ip=’10.10.10.10’/>” +
“<server name=’PC2’ ip=’10.10.10.12’/></servers>”
$t

<servers><server name=’PC1’ ip=’10.10.10.10’/>
<server name=’PC2’ ip=’10.10.10.12’/></servers>

# If you assign the text to a data type[xml], you’ll
# suddenly be able to access the XML structure:
[xml]$list = $t
$list.servers

server
------
{PC1, PC2}
$list.servers.server
name ip
---- --
PC1  10.10.10.10
PC2  10.10.10.12

# Even changes to the XML contents are possible:
$list.servers.server[0].ip = “10.10.10.11”
$list.servers

name ip
---- --
PC1  10.10.10.11
PC2  10.10.10.12
```

```
# The result could be output again as text, including the
# modification:
$list.get_InnerXML()

<servers><server name="PC1" ip="10.10.10.11" />
<server name="PC2" ip="10.10.10.12" /></servers>
```

Variable type	Description	Example
[array]	An array	
[bool]	Yes-no value	[boolean]$flag = $true
[byte]	Unsigned 8-bit integer, 0...255	[byte]$value = 12
[char]	Individual unicode character	[char]$a = “t”
[datetime]	Date and time indications	[datetime]$date = “12.Nov 2004 12:30”
[decimal]	Decimal number	[decimal]$a = 12 $a = 12d
[double]	Double-precision floating point decimal	$amount = 12.45
[guid]	Globally unambiguous 32-byte identification number	[guid]$id = [System.Guid]::NewGuid() $id.toString()
[hashtable]	Hash table	
[int16]	16-bit integer with characters	[int16]$value = 1000
[int32], [int]	32-bit integers with characters	[int32]$value = 5000
[int64], [long]	64-bit integers with characters	[int64]$value = 4GB
[nullable]	Widens another data type to include the ability to contain null values. It can be used, among others, to implement optional parameters	[Nullable``1[[System.DateTime]]]$test = Get-Date $test = $null
[psobject]	PowerShell object	
[regex]	Regular expression	$text = “Hello World” [regex]::split($text, “lo”)
[sbyte]	8-bit integers with characters	[sbyte]$value = -12
[scriptblock]	PowerShell scriptblock	
[single], [float]	Single-precision floating point number	[single]$amount = 44.67
[string]	String	[string]$text = “Hello”
[switch]	PowerShell switch parameter	
[timespan]	Time interval	[timespan]$t = New-TimeSpan $(Get-Date) “1.Sep 07”
[type]	Type	
[uint16]	Unsigned 16-bit integer	[uint16]$value = 1000
[uint32]	Unsigned 32-bit integer	[uint32]$value = 5000
[uint64]	Unsigned 64-bit integer	[uint64]$value = 4GB
[xml]	XML document	

Table 3.5: Commonly used .NET data types

Variable Management: Behind the Scenes

Whenever you create a new variable in PowerShell, it is stored in a PSVariable object. This object contains not just the value of the variable, but also other information, such as the description that you assigned to the variable or additional options like write-protection.

If you retrieve a variable in PowerShell, PowerShell will return only the variable value. If you'd like to see the remaining information that was assigned to the variable, you'll need the underlying PSVariable object. Get-Variable will get it for you:

```
$testvariable = "Hello"
$psvariable = Get-Variable testvariable
```

You can now display all the information about *$testvariable* by outputting *$psvariable*. Pipe the output to the cmdlet *Select-Object* to see all object properties and not just the default properties:

```
$psvariable | Select-Object

Name : testvariable
Description :
Value : Hello
Options : None
Attributes : {}
```

- **Description:** The description you specified for the variable.
- **Value:** The value assigned currently to the variable (i.e. its contents).
- **Options:** Options that have been set, such as write-protection or *AllScope.*
- **Attributes:** Additional features, such as permitted data type of a variable for strongly typed variables. The brackets behind Attributes indicate that this is an array, which can consist of several values that can be combined with each other.

Modification of Variables Options

One reason for dealing with the PSVariable object of a variable is to modify the variable's settings. Use either the cmdlet *Set-Variable* or directly modify the *PSVariable* object. For example, if you'd like to change the description of a variable, you can get the appropriate *PSVariable* object and modify its Description property:

```
# Create new variable:
$test = "New variable"
```

```
# Create PSVariable object:
$psvariable = Get-Variable test

# Modify description:
$psvariable.Description = “Subsequently added description”
Dir variable:\test | Format-Table name, description

Name Description
---- -----------
test Subsequently added description

# Get PSVariable object and directly modify the description:
(Get-Variable test).Description =
“An additional modification of the description.”
Dir variable:\test | Format-Table name, description

Name Description
---- -----------
test An additional modification of the description.

# Modify a description of an existing variable with Set-Variable:
Set-Variable test -description “Another modification”
Dir variable:\test | Format-Table name, description

Name Description
---- -----------
test Another modification
```

As you can see in the example above, you do not need to store the *PSVariable* object in its own variable to access its *Description* property. Instead, you can use a sub-expression, i.e. a statement in parentheses. PowerShell will then evaluate the contents of the sub-expression separately. The expression directly returns the required *PSVariable* object so you can then call the *Description* property directly from the result of the sub-expression. You could have done the same thing by using *Set-Variable*. Reading the settings works only with the *PSVariable* object:

```
(Get-Variable test).Description

An additional modification of the description.
```

Write-Protecting Variables

For example, you can add the ReadOnly option to a variable if you’d like to write-protect it:

```
$Example = 10

# Put option directly in PSVariable object:
(Get-Variable Example).Options = “ReadOnly”
```

```
$Example = 10

# Put option directly in PSVariable object:
(Get-Variable Example).Options = “ReadOnly”

# Modify option as wish with Set-Variable; because the variable
# is read-only, -force is required:
Set-Variable Example -option “None” -force

# Write-protection turned off again; variable contents may now
# be modified freely:
$Example = 20
```

The *Constant* option must be set when a variable is created because you may not convert an existing variable into a constant.

```
# A normal variable may not be converted into a constant:
$constant = 12345
(Get-Variable constant).Options = “Constant”


  Exception in setting “Options”: “The existing variable “constant”
  may not be set as a constant. Variables may only be set as
  constants when they are created.”
  At line:1 char:26
  + (Get-Variable constant).O <<<< options = “Constant”
```

Option	Description
“None”	NO option (default)
“ReadOnly”	Variable contents may only be modified by means of the -force parameter
“Constant”	Variable contents can’t be modified at all. This option must already be specified when the variable is created. Once specified this option cannot be changed.
“Private”	The variable is visible only in a particular context (local variable).
“AllScope”	The variable is automatically copied in a new variable scope.

Table 3.6: Options of a PowerShell variable

Examining Strongly Typed Variables

Once you assign a specific data type to a variable as shown above, PowerShell will add this information to the variable attributes. .

If you delete the Attributes property, the variable will be unspecific again so in essence you remove the strong type again:

```
# List attributes and delete:
$constant = 12345
(Get-Variable a).Attributes

 TypeId
 ------
 System.Management.Automation.ArgumentTypeConverterAttribute

 # Delete type specification:
 (Get-Variable a).Attributes.Clear()

# Strong type specification is removed; now the variable can
# store text again:
$a = "Test"
```

Validating Variable Contents

The *Attributes* property of a *PSVariable* object can include additional conditions, such as the maximum length of a variable. In the following example, a valid length from two to eight characters is assigned to a variable. An error will be generated if you try to store text that is shorter than two characters or longer than eight characters:

```
$a = "Hello"
$aa = Get-Variable a
$aa.Attributes.Add($(New-Object `
System.Management.Automation.ValidateLengthAttribute `
-argumentList 2,8))
$a = "Permitted"
$a = "This is prohibited because its length is not from 2 to 8 characters"

 Because of an invalid value verification (Prohibited because
 its length is not from 2 to 8 characters) may not be carried out for
 the variable "a".
 At line:1 char:3
 + $a <<<< = "Prohibited because its length is not from 2 to 8
```

In the above example *Add()* method added a new .NET object to the attributes with *New-Object*. You'll learn more about *New-Object* in **Chapter 6**. Along with ValidateLengthAttribute, there are additional restrictions that you can place on variables.

Restriction	Category
Variable may not be zero	ValidateNotNullAttribute
Variable may not be zero or empty	ValidateNotNullOrEmptyAttribute
Variable must match a Regular Expression	ValidatePatternAttribute
Variable must match a particular number range	ValidateRangeAttribute
Variable may have only a particular set value	ValidateSetAttribute

Table 3.6: Options of a PowerShell variable

In the following example, the variable must contain a valid e-mail address or all values not matching an e-mail address will generate an error. The e-mail address is defined by what is called a Regular Expression. You'll learn more about Regular Expressions in **Chapter 13**.

```
$email = “tobias.weltner@powershell.com”
$v = Get-Variable email
$pattern = “\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b”
$v.Attributes.Add($(New-Object `
System.Management.Automation.ValidatePatternAttribute `
-argumentList $pattern))
$email = “valid@email.de”
$email = “invalid@email”

  Because of an invalid value verification (invalid@email) may not
  be carried out for the variable “email”.
  At line:1 char:7
  + $email <<<< = “invalid@email”
```

If you want to assign a set number range to a variable, use *ValidateRangeAttribute*. The variable *$age* accepts only numbers from 5 to 100:

```
$age = 18
$v = Get-Variable age
$v.Attributes.Add($(New-Object `
System.Management.Automation.ValidateRangeAttribute `
-argumentList 5,100))
$age = 30
$age = 110

  Because of an invalid value verification (110) may not be
  carried out for the variable “age”.
  At line:1 char:7
  + $age <<<< = 110
```

Restriction	Category
Variable may not be zero	ValidateNotNullAttribute
Variable may not be zero or empty	ValidateNotNullOrEmptyAttribute
Variable must match a Regular Expression	ValidatePatternAttribute
Variable must match a particular number range	ValidateRangeAttribute
Variable may have only a particular set value	ValidateSetAttribute

Table 3.6: Options of a PowerShell variable

In the following example, the variable must contain a valid e-mail address or all values not matching an e-mail address will generate an error. The e-mail address is defined by what is called a Regular Expression. You'll learn more about Regular Expressions in **Chapter 13**.

```
$email = “tobias.weltner@powershell.com”
$v = Get-Variable email
$pattern = “\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b”
$v.Attributes.Add($(New-Object `
System.Management.Automation.ValidatePatternAttribute `
-argumentList $pattern))
$email = “valid@email.de”
$email = “invalid@email”

  Because of an invalid value verification (invalid@email) may not
  be carried out for the variable “email”.
  At line:1 char:7
  + $email <<<< = “invalid@email”
```

If you want to assign a set number range to a variable, use *ValidateRangeAttribute*. The variable *$age* accepts only numbers from 5 to 100:

```
$age = 18
$v = Get-Variable age
$v.Attributes.Add($(New-Object `
System.Management.Automation.ValidateRangeAttribute `
-argumentList 5,100))
$age = 30
$age = 110

  Because of an invalid value verification (110) may not be
  carried out for the variable “age”.
  At line:1 char:7
  + $age <<<< = 110
```

If you would like to limit a variable to special key values, ValidateSetAttribute is the right option. The variable $option accepts only the contents yes, no, or perhaps:

```
$age = “yes”
$v = Get-Variable option
$v.Attributes.Add($(New-Object `
System.Management.Automation.ValidateRangeAttribute `
-argumentList “yes”, “no”, “perhaps”))
$option = “no”
$option = “perhaps”
$option = “don’t know”


  Verification cannot be performed because of an invalid value
 (don’t know) for the variable “option”.
 At line:1 char:8
 + $option <<<< = “don’t know”
```

Summary

Variables store information. Variables are by default not bound to a specific data type, and once you assign a value to a variable, PowerShell will automatically pick a suitable data type. By strongly-typing variables, you can restrict a variable to a specific data type of your choice. You strongly-type a variable by specifying the data type before the variable name:

```
# Strongly type variable a:
[Int]$a = 1
```

You can prefix the variable name with “$” to access a variable. The variable name can consist of numbers, characters, and special characters, such as the underline character “_”. Variables are not case-sensitive. If you’d like to use characters in variable names with special meaning to PowerShell (like parenthesis), the variable name must be enclosed in brackets. PowerShell doesn’t require that variables be specifically created or declared before use.

There are pre-defined variables that PowerShell will create automatically. They are called “automatic variables.” These variables tell you information about the PowerShell configuration. For example, beginning with PowerShell 2.0, the variable $psversiontable will dump the current PowerShell version and versions of its dependencies:

```
PS > $PSVersionTable

Name                           Value
----                           -----
CLRVersion                     2.0.50727.4952
BuildVersion                   6.1.7600.16385
PSVersion                      2.0
WSManStackVersion              2.0
PSCompatibleVersions           {1.0, 2.0}
SerializationVersion           1.1.0.1
PSRemotingProtocolVersion      2.1
```

You can change the way PowerShell behaves by changing automatic variables. For example, by default PowerShell stores only the last 64 commands you ran (which you can list with *Get-History* or re-run with *Invoke-History*). To make PowerShell remember more, just adjust the variable $MaximumHistoryCount:

```
PS > $MaximumHistoryCount

  64

PS > $MaximumHistoryCount = 1000

PS > $MaximumHistoryCount

  1000
```

PowerShell will store variables internally in a PSVariable object. It contains settings that write-protect a variable or attach a description to it **(Table 3.6)**. It's easiest for you to set this special variable options by using the *New-Variable* or *Set-Variable* cmdlets **(Table 3.1)**.

Every variable is created in a scope. When PowerShell starts, an initial variable scope is created, and every script and every function will create their own scope. By default, PowerShell accesses the variable in the current scope, but you can specify other scopes by adding a prefix to the variable name\: *local:, private:, script:, and global:.*

Chapter 4. Arrays and Hashtables

Whenever a command returns more than one result, PowerShell will automatically wrap the results into an array. So dealing with arrays is important in PowerShell. In this chapter, you will learn how arrays work. We will cover simple arrays and also so-called "associative arrays," which are also called "hash tables."

Topics Covered:

- **PowerShell Commands Returns Arrays**
  - Discovering Arrays
  - Processing Array Elements in a Pipeline
  - Working with Real Objects
- **Creating New Arrays**
  - Polymorphic Arrays
  - Arrays With Only One (Or No) Element
- **Addressing Array Elements**
  - Choosing Several Elements from an Array
  - Adding Elements to an Array and Removing Them
- **Using Hash Tables**
  - Creating a New Hash Table
  - Creating Objects From Hash Tables
  - Using Hash tables To Calculate Properties
  - Storing Arrays in Hash Tables
  - Inserting New Keys in an Existing Hash Table
  - Modifying and Removing Values
- **Copying Arrays and Hash Tables**
- **Strongly Typed Arrays**
- **Summary**

PowerShell Commands Return Arrays

If you store the result of a command in a variable and then output it, you might at first think that the variable contains plain text:

```
$a = inconfig
$a

  Windows IP Configuration
  Ethernet adapter LAN Connection

  Media state

  . . . . . . . . . . . . : Medium disconnected

  Connection-specific DNS Suffix:
  Connection location IPv6 Address . : fe80::6093:8889:257e:8d1%8
  IPv4 address . . . . . . . . . . . : 192.168.1.35
  Subnet Mask . . . . . . . . . . . . : 255.255.255.0
```

In reality, the result consists of a number of pieces of data, and PowerShell returns them as an array. This occurs automatically whenever a command returns more than a single piece of data.

Discovering Arrays

You can check the data type to find out whether a command will return an array:

```
$a = “Hello”
$a -is [Array]

  False

$a = ipconfig
$a -is [Array]

  True
```

You can check the data type to find out whether a command will return an array:

```
$a .Count

  53
```

Here, the ipconfig command returned 53 single results that were all stored in $a. If you'd like to examine a single array element, you can specify its index number. If an array has 53 elements, then its valid index numbers are 0 to 52 (the index always starts at 0).

```
# Show the second element:
$a[1]

  Windows IP Configuration
```

It is important to understand just when PowerShell will use arrays. If a command returns just one result, it will happily return that exact result to you. Only when a command returns more than one result will it wrap them in an array.

```
$result = Dir
$result -is [array]

  True

$result = Dir C:\autoexec.bat
$result -is [array]

  False
```

Of course, this will make writing scripts difficult because sometimes you cannot predict whether a command will return one, none, or many results. That's why you can make PowerShell return any result as an array.

Use *@()* if you'd like to force a command to always return its result in an array. This way you find out the number of files in a folder:

```
$result = @(Dir $env:windir -ea 0)
$result .Count
```

Or in a line

```
$result = @(Dir $env:windir -ea 0).Count
```

Processing Array Elements in a Pipeline

Ipconfig will return each line of text as an array element. This is great since all the text lines are individual array elements, allowing you to process them individually in a pipeline. For example, you can filter out unwanted text lines:

```
# Store result of an array and then pass along a pipeline to Select-String:
$result = ipconfig
$result | Where-Object { $_ -like “*Address*”
```

```
     Connection location IPv6 Address . . . : fe80::6093:8889:257e:8d1%8
     IPv4 address . . . . . . . . . . . : 192.168.1.35
     Connection location IPv6 Address . : fe80::5efe:192.168.1.35%16

     Connection location IPv6 Address . . . : fe80::14ab:a532:a7b9:cd3a%11
```

As such, the result of ipconfig was passed to *Where-Object*, which filtered out all text lines that did not contain the keyword you were seeking. With minimal effort, you can now reduce the results of *ipconfig* to the information you deem relevant.

Working with Real Objects

Ipconfig is a legacy command, not a PowerShell cmdlet. While it is a command that will return individual information stored in arrays, this individual information will consist of plain text. Real PowerShell cmdlets will return rich objects, not text, even though the results will appear as plain text:

```
    Dir

      Directory: Microsoft.PowerShell.Core\FileSystem::C:\Users\
      Tobias Weltner
      Mode LastWriteTime Length Name
      ---- ------------- ------ ----
      d---- 10/01/2007 16:09 Application Data
      d---- 07/26/2007 11:03 Backup
      d-r-- 04/13/2007 15:05 Contacts
      d---- 06/28/2007 18:33 Debug
      d-r-- 10/04/2007 14:21 Desktop
      d-r-- 10/04/2007 21:23 Documents
      d-r-- 10/09/2007 12:21 Downloads
      (...)
```

Let’s check if the return value is an array:

```
    $result = Dir
    $result.Count

      82
```

Every element in an array will represent a file or a directory. So if you output an element from the array to the console, PowerShell will automatically convert the object into text:

```
     # Access the fifth element:
    $result[4]

    Directory: Microsoft.PowerShell.Core\FileSystem::C:\Users\
    Tobias Weltner
    Mode LastWriteTime Length Name
    ---- ------------- ------ ----
    d-r-- 04.10.2007 14:21 Desktop
```

In reality, each element returned by *Dir (Get-Childitem)* is really an object with a number of individual properties. Some of these properties surfaced in the previous example as column headers (like Mode, LastWriteTime, Length, and Name). The majority of properties did not show up, though. To see all object properties, you can pipe them on to *Select-Object* and specify an “*” to show all properties. PowerShell will now output them as list rather than table since the console is too narrow to show them all

```
# Display all properties of this element:
$result[4] | Format-List *

  PSPath : Microsoft.PowerShell.Core\FileSystem::
  C:\Users\Tobias Weltner\Desktop
  PSParentPath : Microsoft.PowerShell.Core\FileSystem::
  C:\Users\Tobias Weltner
  PSChildName : Desktop
  PSDrive : C
  PSProvider : Microsoft.PowerShell.Core\FileSystem
  PSIsContainer : True
  Mode : d-r--
  Name : Desktop
  Parent : Tobias Weltner
  Exists : True
  Root : C:\
  FullName : C:\Users\Tobias Weltner\Desktop
  Extension :
  CreationTime : 04/13/2007 01:54:53
  CreationTimeUtc : 04/12/2007 23:54:53
  LastAccessTime : 10/04/2007 14:21:20
  LastAccessTimeUtc : 10/04/2007 12:21:20
  LastWriteTime : 10/04/2007 14:21:20
  LastWriteTimeUtc : 10/04/2007 12:21:20
  Attributes : ReadOnly, Directory
```

You’ll learn more about these types of objects in **Chapter 5.**

Creating New Arrays

You can easily create your own arrays. Simply use a comma to place elements into an array:

```
$array = 1,2,3,4
$array

 1
 2
 3
 4
```

There’s even a shortcut for number ranges:

```
$array = 1..4
$array

 1
 2
 3
 4
```

Polymorphic Arrays

Just like variables, individual elements of an array can store any type of value you assign. This way, you can store whatever you want in an array, even a mixture of different data types. Again, you can separate the elements by using commas:

```
$array = “Hello”, “World”, 1, 2, (Get-Date)
$array

  Hello
  World
  1
  2
  Tuesday, August 21, 2007 12:12:28
```

Important

Why is the Get-Date cmdlet enclosed in parentheses? Just try it without parentheses. Arrays can only store data. Get-Date is a command and no data. Since you want PowerShell to evaluate the command first and then put its result into the array, you will need to use parentheses. Parentheses will identify a sub-expression and tell PowerShell to evaluate and process it first.

Arrays With Only One (Or No) Element

How do you create arrays with just one single element? Here’s how:

```
$array = ,1
$array .Length

  1
```

You'll need to use the construct @(...)to create an array without any elements at all:

```
$array = @()
$array.Length

  0

$array = @(12)
$array

  12

$array = @(1,2,3,"Hello")
$array

  1
  2
  3
  Hello
```

Why would you want to create an empty array in the first place? Because you can add elements to it like this when you start with an empty array:

```
$array = @()
$array += 1
$array += 3

  1
  3
```

Addresing Array Elements

Every element in an array is addressed using its index number. You will find that negative index numbers count from last to first. You can also use expressions that calculate the index value:

```
# Create your own new array:
$array = -5..12

# Access the first element:
$array [0]
```

```
# Access the last element (several methods):
$array = [-1]

  12

$array[$array.Count-1]

  12

$array[$array.lenght-1]

  12

# Access a dynamically generated array that is not stored in a variable:
(-5..12)[2]

  -3
```

Remember, the first element in your array will always have the index number 0. The index *-1* will always give you the last element in an array. The example demonstrates that the total number of all elements will be returned in two properties: *Count* and *Length.* Both of these properties will behave identically.

Here is a real-world example using arrays and accessing individual elements. First, assume you have a path and want to access only the file name. Every string object has a built-in method called *Split()* that can split the text into chunks. All you will need to do is submit the split character that is used to separate the chunks:

```
PS > $path = “c:\folder\subfolder\file.txt”

PS > $array = $path.Split(‘\’)

PS > $array

c:
folder
subfolder
file.txt
```

As you see, by splitting a path at the backslash, you will get its components. The file name is always the last element of that array. So to access the filename, you will access the last array element:

```
PS > $array[-1]

  file.txt
```

Likewise, if you are interested in the file name extension, you can change the split character and use “.” instead:

```
PS > $path.Split(‘.’)[-1]

 txt
```

Choosing Several Elements from an Array

You can also access more than one array element at once by specifying multiple index numbers. The result is a new array that contains the subset that you picked from the original array:

```
# Store directory listing in a variable:
$list = dir $home

# Output only the 2nd, 5th, 8th, and 13th entry:
$list[1,4,7,12]

Directory: Microsoft.PowerShell.Core\FileSystem::C:\Users\
Tobias Weltner
Mode LastWriteTime Length Name
---- ------------- ------ ----
d---- 07/26/2007 11:03 Backup
d-r-- 08/20/2007 07:52 Desktop
d-r-- 08/12/2007 10:21 Favorites
d-r-- 04/13/2007 01:55 Saved Games
```

The second line will select the second, fifth, eighth, and thirteenth elements (remember that the index begins at 0). You can use this approach to reverse the contents of an array:

```
# Create an array with values from 1 to 10
$array = 1..10

# Select the elements from 9 to 0 (output array contents
# in reverse order):
$array = $array[($array.length-1)..0]
$array

  10
  9
  ...
  1
```

Pro Tip

Reversing the contents of an array using the approach described above is not particularly efficient because PowerShell has to store the result in a new array. Instead, you can use the special array functions of the .NET Framework (see **Chapter 6**). This will enable you to reverse the contents of an array very efficiently:

```
# Create an array containing text and output contents:
$a = ipconfig
$a

# Reverse array contents and then output it again:
[array]::Reverse($a)
$a
```

Adding Elements to an Array and Removing Them

Arrays will always contain a fixed number of elements. You'll have to make a new copy of the array with a new size to add or remove elements later. You can simply use the "+=" operator to do that and then add new elements to an existing array:

```
# Store directory listing in a variable:
$array += "New Value"
$array

 1
 2
 3
 New Value
```

You will find that array sizes can't be modified so PowerShell will work behind the scenes to create a brand-new, larger array, copying the contents of the old array into it, and adding the new element. PowerShell will work exactly the same way when you want to delete elements from an array. Here, too, the original array is copied to a new, smaller array while disposing of the old array. For example, the next line will remove elements 4 and 5 using the indexes 3 and 4:

```
$array = $array [0..2] + $array[5..10]
$array.Count

 9
```

As you can imagine, creating new arrays to add or remove array elements is a slow and expensive approach and is only useful for occasional array manipulations. A much more efficient way is to convert an array to an ArrayList object, which is a specialized array. You can use it as a replacement for regular arrays and benefit from the added functionality, which makes it easy to add, remove, insert or even sort array contents:

```
PS > $array = 1..10
PS > $superarray = [System.Collections.ArrayList]$array
PS > $superarray.Add(11) | Out-Null
PS > $superarray.RemoveAt(3)
PS > $superarray.Insert(2,100)
PS > $superarray

  1
  2
  100
  3
  5
  6
  7
  8
  9
  10
  11

PS > $superarray.Sort()
PS > $superarray

  1
  2
  3
  5
  6
  7
  8
  9
  10
  11
  100

PS > $superarray.Reverse()
PS > $superarray

  100
  11
  10
  9
  8
  7
  6
  5
  3
  2
  1
```

Using Hash Tables

Hash tables store “key-value pairs.” So, in hash tables you do not use a numeric index to address individual elements, but rather the key you assigned to a value.

Creating a New Hash Table

When creating a new hash table, you can use @{} instead of @(), and specify the key-value pair that is to be stored in your new hash table. You can use semi-colons to separate key-value pairs:

```
# Create a new hash table with key-value pairs
$list = @{Name = “PC01”; IP=”10.10.10.10”; User=”Tobias Weltner”}

 Name Value
 ---- -----
 Name PC01
 IP 10.10.10.10
 User Tobias Weltner

# Access to the key “IP” returns the assigned value:
$list[“IP”]

 10.10.10.10

# As for arrays, several elements can be selected at the same time:
$list[“Name”, “IP”]

 PC01
 10.10.10.10

# A key can also be specified by dot notation:
$list.IP

 10.10.10.10

# A key can even be stored in a variable:
$key = “IP”
$list.$key

 10.10.10.10
```

```
# Keys returns all keys in the hash table:
$list.keys

 Name
 IP
 User

# If you combine this, you can output all values in the hash table
$list[$list.

 PC01
 10.10.10.10
 Tobias Weltner
```

The example shows that you how to retrieve the values in the hash table using the assigned key. There are two forms of notation you can use to do this:

- **Square brackets:** Either you use square brackets, like in arrays;
- **Dot notation:** *Or* you use dot notation, like with objects, and specify respectively the key name with the value you want to return. The key name can be specified as a variable.

The square brackets can return several values at the same time exactly like arrays if you specify several keys and separate them by a comma. Note that the key names in square brackets must be enclosed in quotation marks (you don't have to do this if you use dot notation).

Creating Objects From Hash Tables

One area where hash tables are used is when you want to return text results into real objects. First, create a hash table and then convert this into an object. Let's say you want to combine information you retrieved from different sources into one consolidated result object. Here is how you can do this:

```
PS> $info = @{}
PS> $info.BIOSVersion = Get-WmiObject Win32_BIOS | Select-Object -ExpandProperty Version
PS> $info.OperatingSystemVersion = Get-WmiObject win32_OperatingSystem | Select
-Object -ExpandProperty Version
PS> $info.PowerShellVersion = $PSVersionTable.psversion.ToString()
PS> New-Object PSObject -property $info

OperatingSystemVersion      BIOSVersion                 PowerShellVersion
----------------------      -----------                 -----------------
6.1.7600                    SECCSD - 6040000            2.0
```

Using Hash Tables to Calculate Properties

Another scenario where hash tables are used is to calculate properties that do not exist. For example, if you’d like to display file size in Megabytes instead of bytes, you can create a hash table with the keys “Name” and “Expression.” “Name” will hold the name of the calculated property, and “Expression” will define a script block used to calculate the property:

```
$MBSize = @{Name=’Size (MB)’; Expression={ if ($_.Length -ne $null) {‘{0:0.0} MB’
 -f ($_.Length / 1MB) } else { ‘n/a’} }}
```

You can now use your hash table to add the calculated property to objects:

```
Dir $env:windir | Select-Object Name, LastWriteTime, $MBSize
```

Note: Because of a PowerShell bug, this will only work when you create the hash table with initial values like in the example above. It will not work when you first create an empty hash table and then add the key-value pairs in a second step.

Hash tables can control even more aspects when using them in conjunction with the family of Format-* cmdlets. For example, if you use Format-Table, you can then pass it a hash table with formatting details:

- **Expression:** The name of object property to be displayed in this column
- **Width:** Character width of the column
- **Label:** Column heading
- **Alignment:** Right or left justification of the column

You can just define a hash table with the formatting information and pass it on to Format-Table:

```
# Setting formatting specifications for each column in a hash table:
$column1 = @{expression=”Name”; width=30; label=”filename”; alignment=”left”}
$column2 = @{expression=”LastWriteTime”; width=40; label=”last modification”;
alignment=”right”}

# Output contents of a hash table:
$column1
Name                           Value
----                           -----
alignment                      left
label                          File name
width                          30
expression                     Name
```

```
# Output Dir command result with format table and selected formatting:
Dir | Format-Table $column1, $column2
File Name                                           last modification
---------                                               ---------------
Application data                                    10/1/2007  16:09:57
Backup                                              07/26/2007 11:03:07
Contacts                                            04/13/2007 15:05:30
Debug                                               06/28/2007 18:33:29
Desktop                                             10/4/2007  14:21:20
Documents                                           10/4/2007  21:23:10
(...)
```

You’ll learn more about format cmdlets like Format-Table in the **Chapter 5.**

Storing Arrays in Hash Tables

You can store classic array inside of hash tables, too. This is possible because hash tables use the semi-colon as key-value pair separators, which leaves the comma available to create classic arrays:

```
# Create hash table with arrays as value:
$test = @{ value1 = 12; value2 = 1,2,3 }

# Return values (value 2 is an array with three elements):
$test.value1
12
$test.value2
1
2
3
```

Storing Arrays in Hash Tables

If you’d like to insert new key-value pairs in an existing hash table, you can just specify the new key and the value that is to be assigned to the new key. Again, you can choose between the square brackets and dot notations.

```
# Create hash table with arrays as value:
$list = @{Name = “PC01”; IP=”10.10.10.10”; User=”Tobias Weltner”}

# Insert two new key-value pairs in the list (two different notations are possible):
$list.Date = Get-Date
$list[“Location”] = “Hanover”
```

```
# Check result:
$list
Name                           Value
----                           -----
Name                           PC01
Location                       Hanover
Date                           08/21/2007 13:00:18
IP                             10.10.10.10
User                           Tobias Weltner
```

You can create empty hash tables and then insert keys as needed because it’s easy to insert new keys in an existing hash table:

```
# Create empty hash table
$list = @{}

# Subsequently insert key-value pairs when required
$list.Name = “PC01”
$list.Location = “Hanover”
(...)
```

Modifying and Removing Values

If all you want to do is to change the value of an existing key in your hash table, just overwrite the value:

```
# Overwrite the value of an existing key with a new value (two possible notations):
$list[“Date”] = (Get-Date).AddDays(-1)
$list.Location = “New York”

Name Value
---- -----
Name PC01
Location New York
Date 08/20/2007 13:10:12
IP 10.10.10.10
User Tobias Weltner
```

If you’d like to completely remove a key from the hash table, use Remove() and as an argument specify the key that you want to remove:

```
$list.remove(“Date”)
```

Using Hash Tables for Output Formatting

An interesting use for hash tables is to format text. Normally, PowerShell outputs the result of most commands as a table and internally uses the cmdlet *Format-Table:*

```
# Both lines return the same result:
Dir
Dir | Format-Table
```

If you use *Format-Table*, you can pass it a hash table with formatting specifications. This enables you to control how the result of the command is formatted.

Every column is defined with its own hash table. In the hash table, values are assigned to the following four keys:

- **Expression:** The name of object property to be displayed in this column
- **Width:** Character width of the column
- **Label:** Column heading
- **Alignment:** Right or left justification of the column

All you need to do is to pass your format definitions to Format-Table to ensure that your listing shows just the name and date of the last modification in two columns:

```
# Setting formatting specifications for each column in a hash table:
$list = @{expression=”Name”; width=30; `
label=”filename”; alignment=”left”}
$column2 = @{expression=”LastWriteTime”; width=40; `
label=”last modification”; alignment=”right”}

# Output contents of a hash table:
$column1

  Name Value
  ---- -----
  alignment left
  label File name
  width 30
  expression Name

# Output Dir command result with format table and
# selected formatting:
Dir | Format-Table $column1, $column2

File Name Last Modification
--------- ----------------
Application Data 10/1/2007 16:09:57
Backup 07/26/2007 11:03:07
```

```
File Name Last Modification
--------- ---------------
Application Data 10/1/2007 16:09:57
Backup 07/26/2007 11:03:07
Contacts 04/13/2007 15:05:30
Debug 06/28/2007 18:33:29
Desktop 10/4/2007 14:21:20
Documents 10/4/2007 21:23:10
(...)
```

You'll learn more about format cmdlets like *Format-Table* in the **Chapter 5**.

Copying Arrays and Hash Tables

Copying arrays or hash tables from one variable to another works, but may produce unexpected results. The reason is that arrays and hash tables are not stored directly in variables, which always store only a single value. When you work with arrays and hash tables, you are dealing with a reference to the array or hash table. So, if you copy the contents of a variable to another, only the reference will be copied, not the array or the hash table. That could result in the following unexpected behavior:

```
$array1 = 1,2,3
$array2 = $array1
$array2[0] = 99
$array1[0]

 99
```

Although the contents of *$array2* were changed in this example, this affects *$array1* as well, because they are both identical. The variables *$array1* and *$array2* internally reference the same storage area. Therefore, you have to create a copy if you want to copy arrays or hash tables,:

```
$array1 = 1,2,3
$array2 = $array1.Clone()
$array2[0] = 99
$array1[0]

 1
```

Whenever you add new elements to an array (or a hash table) or remove existing ones, a copy action takes place automatically in the background and its results are stored in a new array or hash table. The following example clearly shows the consequences:

```
# Create array and store pointer to array in $array2:
$array1 = 1,2,3
$array2 = $array1

# Assign a new element to $array2. A new array is created in the process and stored in
$array2 += 4
$array2[0]=99

# $array1 continues to point to the old array:
$array1[0]

  1
```

Strongly Typed Arrays

Arrays are typically polymorphic: you can store any type of value you want in any element. PowerShell automatically selects the appropriate type for each element. If you want to limit the type of data that can be stored in an array, use "strong typing" and predefine a particular type. You should specify the desired variable type in square brackets. You also specify an open and closed square bracket behind the variable type because this is an array and not a normal variable:

```
# Create a strongly typed array that can store whole numbers only:
[int[]]$array = 1,2,3

# Everything that can be converted into a number is allowed
# (including strings):
$array += 4
$array += 12.56
$array += "123"

# If a value cannot be converted into a whole number, an error
# will be reported:
$array += "Hello"

  The value "Hello" cannot be converted into the type "System.Int32".
  Error: "Input string was not in a correct format."
  At line:1 char:6
  + $array <<<< += "Hello"
```

In the example, *$array* was defined as an array of the *Integer type*. Now, the array is able to store only whole numbers. If you try to store values in it that cannot be turned into whole numbers, an error will be reported.

Summary

Arrays and hash tables can store as many separate elements as you like. Arrays assign a sequential index number to elements that always begin at 0. Hash tables in contrast use a key name. That's why every element in hash tables consists of a key-value pair.

You create new arrays with @(Element1, Element2, ...). You can also leave out @() for arrays and only use the comma operator. You create new hash tables with *@{key1=value1;key2=value2; ...)*. @{} must always be specified for hash tables. Semi-colons by themselves are not sufficient to create a new hash table.

You can address single elements of an array or hash able by using square brackets. Specify either the index number (for arrays) or the key (for hash tables) of the desired element in the square brackets. Using this approach you can select and retrieve several elements at the same time.

Chapter 5.
Arrays and Hashtables

The PowerShell pipeline chains together a number of commands similar to a production assembly. So, one command hands over its result to the next, and at the end, you receive the result.

Topics Covered:

- **Using the PowerShell Pipeline**
  - Object-oriented Pipeline
  - Text Not Converted Until the End
  - Table 5.1: Typical pipeline cmdlets and functions
  - Streaming: Real-time Processing or Blocking Mode?
  - "Blocking" Pipeline Commands
- **Converting Objects into Text**
  - Making Object Properties Visible
  - Formatting Pipeline Results
  - Displaying Particular Properties
  - Using Wildcard Characters
  - Scriptblocks and "Aggregate" Properties
  - Changing Column Headings
  - Optimizing Column Width
- **Sorting and Grouping Pipeline Results**
  - Sort Object and Hash Tables
  - Grouping Information
  - Using Grouping Expressions
- **Filtering Pipeline Results**
  - Limiting Number of Objects
- **Analyzing and Comparing Results**
  - Statistical Calculations
- **Exporting Pipeline Results**
  - Suppressing Results
  - HTML Outputs

Using the PowerShell Pipeline

Command chains are really nothing new. The old console was able to forward (or “pipe”) the results of a command to the next with the “pipe” operator “|”. One of the more known usages was to pipe data to the tool more, which would then present the data screen page by screen page:

```
Dir | more
```

In contrast to the traditional concept of text piping, the PowerShell pipeline will take an object-oriented approach and implement it in real time. Have a look:

```
Dir | Sort-Object Length | Select-Object Name, Length |
ConvertTo-Html | Out-File report.htm
.\report.htm
```

It returns an HTML report on the windows directory contents sorted by file size. All of this can start with a Dir command, which then passes its result to Sort-Object. The sorted result will then get limited to only the properties you want in the report. ConvertTo-Html will convert the objects to HTML, which is then written to a file.

Object-oriented Pipeline

What you see here is a true object-oriented pipeline so the results from a command remain rich objects. Only at the end of the pipeline will the results be reduced to text or HTML or whatever you choose for output.

Take a look at Sort-Object. It will sort the directory listing by file size. If the pipeline had simply fed plain text into Sort-Object, you would have had to tell Sort-Object just where the file size information was to be found in the raw text. You would also have had to tell Sort-Object to sort this information numerically and not alphabetically. Not so here. All you need to do is tell Sort-Object which object’s property you want to sort. The object nature tells Sort-Object all it needs to know: where the information you want to sort is found and whether it is numeric or letters.

You only have to tell Sort-Object which object’s property to use for sorting because PowerShell will send results as rich .NET objects through the pipeline. Sort-Object does the rest automatically. Simply replace Length with another object’s property, such as Name or LastWriteTime, to sort according to these criteria. Unlike text, information in an object is clearly structured: this is a crucial PowerShell pipeline advantage.

Text Not Converted Until the End

The PowerShell pipeline is always used, even when you provide only a single command. PowerShell will attach to your input the cmdlet Out-Default, which converts the resulting objects into text at the end of the pipeline.

Even a simple Dir command is appended internally and converted into a pipeline command:

```
Dir $env:windir | Out-Default
```

Of course, the real pipeline benefits show only when you start adding more commands. The chaining of several commands will allow you to use commands like Lego building blocks to assemble a complete solution from single commands. The following command will output only a directory's text files listing in alphabetical order:

```
Dir $env:windir *.txt | Sort-Object
```

The cmdlets in Table 5.1 have been specially developed for the pipeline and the tasks frequently performed in it. They will all be demonstrated in the following pages of this chapter.

Note

Just make sure that the commands you use in a pipeline actually do process information from the pipeline. While it is technically OK, the following line is really useless because notepad.exe cannot process pipeline results:

```
Dir $env:windir | Sort-Object | notepad
```

If you'd like to open pipeline results in an editor, you can put the results in a file first and then open the file with the editor

```
Dir $env:windir | Sort-Object | Out-File result.txt; notepad result.txt
```

Cmdlet/Function	**Description**
Compare-Object	Compares two objects or object collections and marks their differences
ConvertTo-Html	Converts objects into HTML code
Export-Clixml	Saves objects to a file (serialization)
Export-Csv	Saves objects in a comma-separated values file
ForEach-Object	Returns each pipeline object one after the other
Format-List	Outputs results as a list
Format-Table	Outputs results as a table
Format-Wide	Outputs results in several columns
Get-Unique	Removes duplicates from a list of values
Group-Object	Groups results according to a criterion
Import-Clixml	Imports objects from a file and creates objects out of them (deserialization)
Measure-Object	Calculates the statistical frequency distribution of object values or texts
more	Returns text one page at a time
Out-File	Writes results to a file
Out-Host	Outputs results in the console
Out-Host -paging	Returns text one page at a time
Out-Null	Deletes results
Out-Printer	Sends results to printer
Out-String	Converts results into plain text

Cmdlet/Function	Description
Select-Object	Filters properties of an object and limits number of results as requested
Sort-Object	Sorts results
Tee-Object	Copies the pipeline's contents and saves it to a file or a variable
Where-Object	Filters results according to a criterion

Table 5.1: Typical pipeline cmdlets and functions

Streaming: Real-time Processing or Blocking Mode?

When you combine several commands in a pipeline, you'll want to understand when each separate command will actually be processed: consecutively or at the same time? The pipeline will process the results in real time, at least when the commands chained together in the pipeline support real-time processing. That's why there are two pipeline modes:

· **Sequential (slow) mode:** In sequential mode, pipeline commands are executed one at a time. So the command's results are passed on to the next one only after the command has completely performed its task. This mode is slow and hogs memory because results are returned only after all commands finish their work and the pipeline has to store the entire results of each command. The sequential mode basically corresponds to the variable concept that first saves the result of a command to a variable before forwarding it to the next command.

· **Streaming Mode (quick):** The streaming mode immediately processes each command result. Every single result is passed directly onto the subsequent command. It will rush through the entire pipeline and is immediately output. This quick mode saves memory because results are output while the pipeline commands are still performing their tasks, and only one element is travelling the pipeline at a time. The pipeline doesn't have to store all of the command's results, but only one single result at a time.

"Blocking" Pipeline Commands

Sorting can only take place when all results are available. That is why Sort-Object is an example of a "blocking" pipeline command, which will first collect all data before it hands over the sorted result to the next command. This also means there can be long processing times and it can even cause instability if you don't pay attention to memory requirements:

```
# Attention: danger!
Dir C:\ -recurse | Sort-Object
```

Important

If you execute this example, you won't see any signs of life from PowerShell for a long time. If you let the command run too long, you may even run out of memory.

Here Dir returns all files and directors on your drive C:\. These results are passed by the pipeline to *Sort-Object*, and because *Sort-Object* can only sort the results after all of them are available, it will collect the results as they come in. Those results eventually block too much memory for your system to handle. The two problem areas in sequential mode are:

Important

First problem: You won't see any activity as long as data is being collected. The more data that has to be acquired, the longer the wait time will be for you. In the above example, it can take several minutes.

Second problem: Because enormous amounts of data have to be stored temporarily before *Sort-Object* can process them, the memory space requirement is very high. In this case, it's even higher so that the entire Windows system will respond more and more clumsily until finally you won't be able to control it any longer.

That's not all. In this specific case, confusing error messages may pile up. If you have Dir output a complete recursive folder listing, it may encounter sub-directories where you have no access rights. While *Sort-Object* continues to collect results (so no results appear), error messages are not collected by *Sort-Object* and appear immediately. Error messages and results get out of sync and may be misinterpreted.

Whether a command supports streaming is up to the programmer. For Sort-Object, there are technical reasons why this command must first wait for all results. Otherwise, it wouldn't be able to sort the results. If you use commands that are not designed for PowerShell then their authors had no way to implement the special demands of PowerShell. For example, it will work if you use the traditional command more.com to output information one page at a time, but more.com is also a blocking command that could interrupt pipeline streaming:

```
# If the preceding command can execute its task quickly,
# you may not notice that it can be a block:
Dir | more.com

# If the preceding command requires much time,
# its blocking action may cause issues:
Dir c:\ -recurse | more.com
```

But also genuine PowerShell cmdlets, functions, or scripts can block pipelines if the programmer doesn't use streaming. Surprisingly, PowerShell developers forgot to add streaming support to the integrated more function. This is why more essentially doesn't behave much differently than the ancient more.com command:

```
# If the preceding command can execute its task quickly, you may not notice that it can be
# a block:
Dir | more.com
# If the preceding command requires much time, its blocking action may cause issues:
Dir c:\ -recurse | more.com
```

Tip

Tip: Use Out-Host -Paging instead of more! Out-Host is a true PowerShell cmdlet and will support streaming:

```
Dir c:\ -recurse | Out-Host -paging
```

Converting Objects into Text

At the end of a day, you want commands to return visible results, not objects. So, while results stay rich data objects while traveling the pipeline, at the end of the pipeline, they must be converted into text. This is done by (internally) adding *Out-Default* to your input. The following commands are identical:

```
Dir
Dir | Out-Default
```

Out-Default will transform the pipeline result into visible text. To do so, it will first call *Format-Table* (or Format-List when there are more than five properties to output) internally, followed by *Out-Host*. *Out-Host* will output the text in the console. So, this is what happens internally:

```
Dir | Format-Table | Out-Host
```

Making Object Properties Visible

To really see all the object properties and not just the ones that PowerShell "thinks" are important, you can use Format-Table and add a "*" to select all object properties.

```
Dir | Format-Table *

PSPat PSPar PSChi PSDri PSPro PSIsC Mode Name Pare Exis Root Full Exte Crea Crea Last Last Last Last Attr
h     entPa ldNam ve    vider ontai           nt     ts      Name nsio tion tion Acce Acce Writ Writ ibut
      th    e                   ner                               n    Time Time ssTi ssTi eTim eTim   es
                                                                            Utc  me   meUt e    eUtc
                                                                                      c
----- ----- ----- ----- ----- ----- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
Mi... Mi... Ap... C     Mi...  True d... A... T... True C:\  C...      2... 2... 2... 2... 2... 2... ...y
Mi... Mi... Ba... C     Mi...  True d... B... T... True C:\  C...      2... 2... 2... 2... 2... 2... ...y
Mi... Mi... Co... C     Mi...  True d... C... T... True C:\  C...      1... 1... 1... 1... 1... 1... ...y
Mi... Mi... Debug C     Mi...  True d... D... T... True C:\  C...      2... 2... 2... 2... 2... 2... ...y
Mi... Mi... De... C     Mi...  True d... D... T... True C:\  C...      1... 1... 3... 3... 3... 3... ...y
```

You now get so much information that columns shrink to an unreadable format.

Tip

For example, if you'd prefer not to reduce visual display because of lack of space, you can use the -*Wrap* parameter, like this:

```
Dir | Format-Table * -wrap
```

Still, the horizontal table design is unsuitable for more than just a handful of properties. This is why PowerShell will use Format-List, instead of Format-Table, whenever there are more than five properties to display. You should do the same:

```
Dir | Format-List *
```

You will now see a list of several lines for each object's property. For a folder, it might look like this:

```
PSPath            : Microsoft.PowerShell.Core\FileSystem::C:\Users\Tobias Weltner\Music
PSParentPath      : Microsoft.PowerShell.Core\FileSystem::C:\Users\Tobias Weltner
PSChildName       : Music
PSDrive           : C
PSProvider        : Microsoft.PowerShell.Core\FileSystem
PSIsContainer     : True
Mode              : d-r--
Name              : Music
Parent            : Tobias Weltner
Exists            : True
Root              : C:\
FullName          : C:\Users\Tobias Weltner\Music
Extension         :
CreationTime      : 13.04.2007 01:54:53
CreationTimeUtc   : 12.04.2007 23:54:53
LastAccessTime    : 10.05.2007 21:37:26
LastAccessTimeUtc : 10.05.2007 19:37:26
LastWriteTime     : 10.05.2007 21:37:26
LastWriteTimeUtc  : 10.05.2007 19:37:26
Attributes        : ReadOnly, Directory
```

A file has slightly different properties:

```
PSPath            : Microsoft.PowerShell.Core\FileSystem::C:\Users\Tobias Weltner\views.PS1
PSParentPath      : Microsoft.PowerShell.Core\FileSystem::C:\Users\Tobias Weltner
PSChildName       : views.PS1
PSDrive           : C
PSProvider        : Microsoft.PowerShell.Core\FileSystem
PSIsContainer     : False
Mode              : -a---
Name              : views.PS1
Length            : 4045
DirectoryName     : C:\Users\Tobias Weltner
```

```
Directory         : C:\Users\Tobias Weltner
IsReadOnly        : False
Exists            : True
FullName          : C:\Users\Tobias Weltner\views.PS1
Extension         : .PS1
CreationTime      : 18.09.2007 16:30:13
CreationTimeUtc   : 18.09.2007 14:30:13
LastAccessTime    : 18.09.2007 16:30:13
LastAccessTimeUtc : 18.09.2007 14:30:13
LastWriteTime     : 18.09.2007 16:46:12
LastWriteTimeUtc  : 18.09.2007 14:46:12
Attributes        : Archive
```

You will now see a list of several lines for each object's property. For a folder, it might look like this:

Formatting Pipeline Results

Transforming objects produced by the pipeline is carried out by formatting cmdlets. There are four choices:

```
Get-Command -verb format
CommandType    Name                         Definition
-----------    ----                         ----------
Cmdlet         Format-Custom                Format-Custom [[-Property] <Object[]>] [-De...
Cmdlet         Format-List                  Format-List [[-Property] <Object[]>] [-Grou...
Cmdlet         Format-Table                 Format-Table [[-Property] <Object[]>] [-Aut...
Cmdlet         Format-Wide                  Format-Wide [[-Property] <Object>] [-AutoSi...
```

These formatting cmdlets are not just useful for converting all of an object's properties into text, but you can also select the properties you want to see.

Displaying Particular Properties

To accomplish this, you type the property that you want to see and not just an asterisk behind the cmdlet. If you do not want to explicitly use a table or list format, it is considered best practice to use *Select-Object* rather than *Format-** because *Select-Object* will automatically determine the best formatting and will also return objects that can be processed by subsequent cmdlets. When you use Format-* cmdlets, objects are converted into formatting information, which can only be interpreted by *Out-** cmdlets which is why *Format-** cmdlets must be used only at the end of your pipeline.

The next instruction will retrieve you a directory listing with only Name and Length. Because sub-directories don't have a property called Length, you will see that the Length column for the sub-directory is empty:

```
Dir | Select-Object Name, Length
Name                                                                  Length
----                                                                  ------
Sources
Test
172.16.50.16150.dat                                                   16
172.16.50.17100.dat                                                   16
output.htm                                                            10834
output.txt                                                            1338
```

Using Wildcard Characters

Wildcard characters are allowed. So, the next command will get you information about your video controller and output all properties that have a resolution keyword:

```
Get-WmiObject Win32_VideoController | Select-Object *resolution*
            CurrentHorizontalResolution               CurrentVerticalResolution
            ---------------------------               -------------------------
                                   1680                                    1050
```

Scriptblocks and "Aggregate" Properties

Script blocks can be used as columns as they basically act as PowerShell instructions included in brackets that work like synthetic properties to calculate their value. Within a script block, the variable $_ will contain the actual object. The script block could convert the Length property into kilobytes if you'd like to output file sizes in kilobytes rather than bytes:

```
Dir | Select-Object name, { [int]($_.Length/1KB) }
Name                                                                       [int]($_.Length/1KB)
----                                                                        ---------------------
output.htm                                                                                     11
output.txt                                                                                     13
backup.pfx                                                                                      2
cmdlet.txt                                                                                     23
```

Or maybe you'd like your directory listing to show how many days have passed since a file or a folder was last modified. By using the New-TimeSpan cmdlet, you can calculate how much time has elapsed up to the current date. To see how this works, you can look at the line below as an example that calculates the time difference between January 1, 2000, and the current date:/p>

```
Directory         : C:\Users\Tobias Weltner
IsReadOnly        : False
Exists            : True
FullName          : C:\Users\Tobias Weltner\views.PS1
Extension         : .PS1
CreationTime      : 18.09.2007 16:30:13
CreationTimeUtc   : 18.09.2007 14:30:13
LastAccessTime    : 18.09.2007 16:30:13
LastAccessTimeUtc : 18.09.2007 14:30:13
LastWriteTime     : 18.09.2007 16:46:12
LastWriteTimeUtc  : 18.09.2007 14:46:12
Attributes        : Archive
```

Or maybe you'd like your directory listing to show how many days have passed since a file or a folder was last modified. By using the New-TimeSpan cmdlet, you can calculate how much time has elapsed up to the current date. To see how this works, you can look at the line below as an example that calculates the time difference between January 1, 2000, and the current date:/p>

```
New-TimeSpan "01/01/2000"
Days              : 4100
Hours             : 21
Minutes           : 13
Seconds           : 15
Milliseconds      : 545
Ticks             : 3543163955453834
TotalDays         : 4100,8842077012
TotalHours        : 98421,2209848287
TotalMinutes      : 5905273,25908972
TotalSeconds      : 354316395,545383
TotalMilliseconds : 354316395545,383
```

Use this script block to output how many days have elapsed from the *LastWriteTime* property up to the current date and to read it out in its own column:

```
{(New-TimeSpan $_.LastWriteTime ).Days}
```

Dir would then return a sub-directory listing that shows how old the file is in days:

```
Dir | Select-Object Name, Length, {(New-TimeSpan $_.LastWriteTime ).Days}
Name                    Length (New-TimeSpan $_.LastWriteTime (Get-Date)).Days
----                    ------ -----------------------------------------------
Application data                                                            61
Backup                                                                      55
Contacts                                                                   158
Debug                                                                       82
Desktop                                                                     19
Documents                                                                    1
(...)
```

Changing Column Headings

When you use synthetic properties, you will notice that column headings look strange because PowerShell puts code in them that computes the column contents. However, after reading the last chapter, you now know that you can use a hash table to format columns more effectively and that you can also rename them:

```
$column = @{Expression={ [int]($_.Length/1KB) }; Name=”KB” }
Dir | Select-Object name, $column
Name                                                                    KB
----                                                                    --
output.htm                                                              11
output.txt                                                              13
backup.pfx                                                               2
cmdlet.txt                                                              23
```

Optimizing Column Width

Because the pipeline processes results in real time, PowerShell cannot know how wide of a space the column elements will occupy. As a result, it will tend to be generous in sizing columns. If you specify the *-AutoSize* parameter, *Format-Table* will collect all results first before setting the maximum width for all elements. You can optimize output, but the results will no longer be output in real time:

```
$column = @{Expression={ [int]($_.Length/1KB) }; Label=”KB” }
Dir | Format-Table name, $column -AutoSize
Name         KB
----         --
output.htm   11
output.txt   13
backup.pfx    2
cmdlet.txt   23
```

Sorting and Grouping Pipeline Results

Using the cmdlets *Sort-Object* und *Group-Object,* you can sort and group other command results. In the simplest scenario, you can just append *Sort-Object* to a pipeline command and your output will already be sorted. It’s really very simple:

```
Dir $env:windir | Sort-Object
```

When you do that, *Sort-Object* will select the property it uses for sorting. It’s better to choose the sorting criterion yourself as every object’s property may be used as a sorting criterion. For example, you could use one to create a descending list of a sub-directory’s largest files:

```
Dir $env:windir | Sort-Object -property Length -descending
```

Tip

You must know which properties are available to use Sort-Object and all the other following cmdlets. In the last section, you learned how to do that. Send the result of a cmdlet to Select-Object *, and you’ll get a list of all properties available that you can use for sorting:

```
Dir $env:windir | Select-Object *
```

Sort-Object can sort by more than one property at the same time. For example, if you’d like to sort all the files in a folder by type first *(Extension property)* and then by name *(Name property)*, you can specify both properties:

```
Dir | Sort-Object Extension, Name
```

Sort Object and Hash Tables

Sort-Object can use hash tables to better control the sorting. Let’s assume that you want to sort a folder by file size and name. While the file size should be sorted in descending order, file names should be sorted in ascending order. You can solve this problem by passing Sort-Object to a hash table (see **Chapter 4**).

```
Dir | Sort-Object @{expression=”Length”;Descending=$true},@{expression=”Name”;
Ascending=$true}
```

The hash table will allow you to append additional information to a property so you can separately specify for each property your preferred sorting sequence.

Grouping Information

Group-Object works by grouping objects based on one or more properties and then counting the groups. You will only need to specify the property you want to use as your grouping option. The next line will return a status overview of services:

```
Get-Service | Group-Object Status
Count Name       Group
----- ----       -----
   91 Running    {AeLookupSvc, AgereModemAudio, Appinfo, Ati External Event Utility...}
   67 Stopped    {ALG, AppMgmt, Automatic LiveUpdate - Scheduler, BthServ...}
```

The number of groups will depend only on how many different values are found in the property specified in the grouping operation. The results' object contains the properties *Count, Name,* and *Group*. Services are grouped according to the desired criteria in the *Group* property. The following will show you how to obtain a list of all currently running services:

```
$result = Get-Service | Group-Object Status
$result[0].Group
```

In a file system, Group-Object could group files based on type:

```
Dir $env:windir | Group-Object Extension
Dir $env:windir | Group-Object Extension | Sort-Object Count -descending
Count Name                      Group
----- ----                      -----
   22                           {Application data, Backup, Contacts, Debug...}
   16 .ps1                      {filter.ps1, findview.PS1, findview2.PS1, findview3.PS1...}
   12 .txt                      {output.txt, cmdlet.txt, ergebnis.txt, error.txt...}
    4 .csv                      {ergebnis.csv, history.csv, test.csv, test1.csv}
    3 .bat                      {ping.bat, safetycopy.bat, test.bat}
    2 .xml                      {export.xml, now.xml}
    2 .htm                      {output.htm, report.htm}
```

Using Grouping Expressions

The criteria used to group objects can also be calculated. The next example uses a script block which returns *True* if the file size exceeds 100 KB or False otherwise. All files larger than 100KB are in the True group:.

```
Dir | Group-Object {$_.Length -gt 100KB}

Count Name                      Group
----- ----                      -----
   67 False                     {Application data, Backup, Contacts, Debug...}
    2 True                      {export.xml, now.xml} in the column Count is reported which
```

The script block is not limited to returning *True* or *False*. The next example will use a script block that returns a file name's first letter. The result: *Group-Object* will group the sub-directory contents by first letters:

```
Dir | Group-Object {$_.name.SubString(0,1).toUpper()}
Count Name                      Group
----- ----                      -----
    4 A                         {Application data, alias1, output.htm, output.txt}
    2 B                         {Backup, backup.pfx}
    2 C                         {Contacts, cmdlet.txt}
    5 D                         {Debug, Desktop, Documents, Downloads...}
    5 F                         {Favorites, filter.ps1, findview.PS1, findview2.PS1...}
    3 L                         {Links, layout.lxy, liste.txt}
    3 M                         {MSI, Music, meinskript.ps1}
```

```
      3 P                         {Pictures, p1.nrproj, ping.bat}
      7 S                         {Saved Games, Searches, Sources, SyntaxEditor...}
     15 T                         {Test, test.bat, test.csv, test.ps1...}
      2 V                         {Videos, views.PS1}
      1 [                         {[test]}
      1 1                         {1}
      4 E                         {result.csv, result.txt, error.txt, export.xml}
      4 H                         {mainscript.ps1, help.txt, help2.txt, history.csv}
      1 I                         {info.txt}
      2 N                         {netto.ps1, now.xml}
      3 R                         {countfunctions.ps1, report.htm, root.cer}
      2 U                         {unsigned.ps1, .ps1}
```

This way, you can even create listings that are divided into sections:

```
   Dir | Group-Object {$_.name.SubString(0,1).toUpper()} | ForEach-Object { ($_.Name)*7;
 “=======”; $_.Group}
(...)
BBBBBBB
=======
d----        26.07.2007     11:03            Backup
-a---        17.09.2007     16:05       1732 backup.pfx
CCCCCCC
=======
d-r--        13.04.2007     15:05            Contacts
-a---        13.08.2007     13:41      23586 cmdlet.txt
DDDDDDD
=======
d----        28.06.2007     18:33            Debug
d-r--        30.08.2007     15:56            Desktop
d-r--        17.09.2007     13:29            Documents
d-r--        24.09.2007     11:22            Downloads
-a---        26.04.2007     11:43       1046 drive.vbs
(...)
```

You can use the parameter -NoElement if you don’t need the grouped objects and only want to know which groups exist. This will save a lot of memory:

```
Get-Process | Group-Object -property Company -noelement
Count Name
----- ----
   50
    1 AuthenTec, Inc.
    2 LG Electronics Inc.
    1 Symantec Corporation
    2 ATI Technologies Inc.
   30 Microsoft Corporation
    1 Adobe Systems, Inc.
    1 BIT LEADER
    1 LG Electronics
    1 Intel Corporation
```

```
      2 Apple Inc.
      1 BlazeVideo Company
      1 ShellTools LLC
      2 Infineon Technologies AG
      1 Just Great Software
      1 Realtek Semiconductor
      1 Synaptics, Inc.
```

Filtering Pipeline Results

If you’re only interested in certain objects, you can use *Where-Object* to filter results. For example, you can filter them based on their Status property if you want to list only running services.

```
Get-Service | Where-Object { $_.Status -eq “Running” }
Status   Name               DisplayName
------   ----               -----------
Running  AeLookupSvc        Applicationlookup
Running  AgereModemAudio    Agere Modem Call Progress Audio
Running  Appinfo            Applicationinformation
Running  AppMgmt            Applicationmanagement
Running  Ati External Ev... Ati External Event Utility
Running  AudioEndpointBu... Windows-Audio-Endpoint-building
Running  Audiosrv           Windows-Audio
Running  BFE                Basis filter Engine
Running  BITS               Intelligent Background Transmiss...
(...)
```

Where-Object takes a script block and evaluates it for every pipeline object. The current object that is travelling the pipeline is found in $_. So *Where-Object* really works like a condition (see **Chapter 7**): if the expression results in *$true*, the object will be let through.

```
Get-WmiObject Win32_Service | Where-Object {($_.Started -eq $false) -and ($_.StartMode -eq
   “Auto”)} | Format-Table
           ExitCode Name                     ProcessId StartMode        State          Status
          --------- ----                     --------- ---------        -----          ------
                  0 Automatic Li...                  0 Auto             Stopped        OK
                  0 ehstart                          0 Auto             Stopped        OK
                  0 LiveUpdate Notic...              0 Auto             Stopped        OK
                  0 WinDefend                        0 Auto             Stopped        OK
```

Limiting Number of Objects

Select-Object has a dual purpose. It can select the columns (properties) you want to see, and it can show only the first or the last results.

```
# List the five largest files in a directory:
Dir | Sort-Object Length -descending | Select-Object -first 5

# List the five longest-running processes:
Get-Process | Sort-Object StartTime | Select-Object -last 5 |
 Select-Object ProcessName, StartTime
```

If you aren't logged on with administrator privileges, you may not retrieve the information from some processes. However, you can avoid exceptions by adding *-ErrorAction SilentlyContinue* (shortcut: -ea 0):

```
Get-Process | Sort-Object StartTime -ea 0 | Select-Object -last 5 |
Select-Object ProcessName, StartTime
```

Analyzing and Comparing Results

Using the cmdlets *Measure-Object* and *Compare-Object,* you can measure and evaluate PowerShell command results. For example, *Measure-Object* will allow you to determine how often particular object properties are distributed. *Compare-Object* will enable you to compare before-and-after snapshots.

Statistical Calculations

Using the Measure-Object cmdlet, you can get statistic information. For example, if you want to check file sizes, let *Dir* give you a directory listing and then examine the *Length* property:

```
Dir $env:windir | Measure-Object Length -average -maximum -minimum -sum
Count    : 50
Average  : 36771,76
Sum      : 1838588
Maximum  : 794050
Minimum  : 0
Property : Length
```

Measure-Object also accepts text files and discovers the frequency of characters, words, and lines in them:

```
Get-Content $env:windir\windowsupdate.log | Measure-Object -character -line -word
```

Exporting Pipeline Results

If you'd like to save results into a text file or print it, rather than outputting it into the console, append Format-* | Out-* to any PowerShell command:

```
Get-Process | Format-Table -AutoSize | Out-File $env:temp\somefile.txt -Width 1000
```

Out-File will support the parameter -Encoding, which you can use to set the output format If you don't remember which encoding formats are allowed. Just specify an invalid value and then the error message will tell you which values are allowed:

```
Dir | Out-File -encoding Dunno
Out-File : Cannot validate argument "Dunno" because it does not belong to the set
   "unicode, utf7, utf8, utf32, ascii, bigendianunicode, default, oem".
```

*Out-** cmdlets turn results into plain text so you are reducing the richness of your results (*Out-GridView* is the only exception to the rule which displays the results in an extra window as a mini-spreadsheet).

Export it instead and use one of the xport-* cmdlets to preserve the richness of your results. For example, to open results in Microsoft Excel, do this:

```
Get-Process | Export-CSV -UseCulture -NoTypeInformation -Encoding UTF8
   $env:temp:\report.csv
Invoke-Item $env:temp\report.csv
```

Suppressing Results

You can send the output to *Out-Null* if you want to suppress command output:

```
# This command not only creates a new directory but also returns the new
directory:
md testdirectory
    Directory: Microsoft.PowerShell.Core\FileSystem::C:\Users\Tobias Weltner
Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        19.09.2007     14:31            testdirectory
rm testdirectory
```

```
# Here the command output is sent to “nothing”:
md testdirectory | Out-Null
rm testdirectory

# That matches the following redirection:
md testdirectory > $null
rm testdirectory
```

HTML Outputs

If you’d like, PowerShell can also pack its results into (rudimentary) HTML files. Converting objects into HTML formats is done by *ConvertTo-Html*:

```
Get-Process | ConvertTo-Html | Out-File output.hta
.\output.hta
Get-Process | Select-Object Name, Description | ConvertTo-Html -title  “Process Report” |
   Out-File output.hta
.\output.hta
```

Chapter 6.
Working with Objects

In this chapter, you will learn what objects are and how to get your hands on PowerShell objects before they get converted to simple text.

Topics Covered:

- **Objects = Properties + Methods**
  - Creating a New Object
  - Adding Properties
  - Adding Methods
- **Properties: What an Object "Is"**
  - Properties Containing Objects
  - Read-Only and Read-Write Properties
    - Table 6.1: Properties of the RawUI object
  - Property Types
  - Listing All Properties
- **Methods: What an Object "Can Do"**
  - Eliminating "Internal" Methods
    - Get_ and Set_ Methods
    - Standard Methods
      - Table 6.2: Standard methods of a .NET object
  - Calling a Method
  - Call Methods with Arguments
    - Which Arguments are Required?
  - Several Method "Signatures"
    - Playing with PromptForChoice
- **Working with Real-Life Objects**
  - Storing Results in Variables
    - Using Object Properties
    - PowerShell-Specific Properties
      - Table 6.3: Different property types
    - Using Object Methods
    - Different Method Types
      - Table 6.4: Different types of methods
- **Using Static Methods**
  - Table 6.5: Mathematical functions from the [Math] library
  - Finding Interesting .NET Types
    - Converting Object Types
    - Using Static Type Members
    - Using Dynamic Object Instance Members
- **Creating New Objects**
  - Creating New Objects with New-Object
    - Using Constructors
  - New Objects by Conversion
  - Loading Additional Assemblies: Improved Internet Download
  - Call Methods with Arguments
  - Using COM Objects
    - Which COM Objects Are Available?
    - How Do You Use COM Objects?
- **Summary**

Objects = Properties + Methods

In real life, you already know what an object is: everything you can touch. Objects in PowerShell are similar. Let's turn a typical real-world object, like a pocketknife, into a PowerShell object.

How would you describe this object to someone, over the telephone? You would probably carefully examine the object and then describe what it is and what it *can do*:

- **Properties:** A pocketknife has particular properties, such as its color, manufacturer, size, or number of blades. The object *is* red, weights 55 grams, has three blades, and is made by the firm Idera. So properties< describe what an object *is*.
- **Methods:** In addition, you can do things with this object, such as cut, turn screws, or pull corks out of wine bottles. The object *can* cut, screw, and remove corks. Everything that an object can is called its *methods*.

In the computing world, an object is very similar: its nature is described by properties, and the actions it can perform are called its methods. Properties and methods are called members.

Creating a Object

Let's turn our real-life pocketknife into a virtual pocketknife. Using *New-Object*, PowerShell can generate new objects, even a virtual pocketknife. First, you will need a new and empty object:

```
$pocketknife = New-Object Object
```

This new object is actually pretty useless. If you call for it, PowerShell will return "nothing":

```
$pocketknife
```

Adding Properties

Next, let's start describing what our object *is*. To do that, you can add properties to the object.

```
# Adding a new property:
Add-Member -MemberType NoteProperty -Name Color-Value Red -InputObject $pocketknife
```

You can use the *Add-Member* cmdlet to add properties. Here, you added the property *color* with the value *red* to the object $pocketknife. If you call for the object now, it suddenly has a property telling the world that its color is red:

```
$pocketknife
Color
-----
Red
```

You can then add more properties to describe the object even better. This time, we use positional parameters to shorten the code necessary to add members to the object:

```
$pocketknife | Add-Member NoteProperty Weight 55
$pocketknife | Add-Member NoteProperty Manufacturer Idera
$pocketknife | Add-Member NoteProperty Blades 3
```

By now, you’ve described the object in *$pocketknife* with a total of four properties. If you output the object in *$pocketknife* in the PowerShell console, PowerShell will automatically convert the object into readable text:

```
# Show all properties of the object all at once:
$pocketknife
Color                                    Weight   Manufacturer                      Blades
-----                                    -------  ----------                        -------
Red                                          55   Idera                                   3
```

You will now get a quick overview of its properties when you output the object to the console. You can access the value of a specific property by either using *Select-Object* with the parameter *-expandProperty*, or add a dot, and then the property name:

```
# Display a particular property:
$pocketknife | Select-Object -expandProperty Manufacturer
$pocketknife.manufacturer
```

Adding Methods

With every new property you added to your object, *$pocketknife* has been gradually taking shape, but it still really can’t do anything. Properties only describe what an object is, not what it can do.

The actions your object can do are called its methods. So let’s teach your object a few useful methods:

```
# Adding new methods:
$pocketknife | Add-Member ScriptMethod cut { “I’m whittling now” }
$pocketknife | Add-Member ScriptMethod screw { “Phew...it’s in!” }

$pocketknife | Add-Member ScriptMethod corkscrew { “Pop! Cheers!” }
```

Again, you used the *Add-Member cmdlet*, but this time you added a method instead of a property (in this case, a *ScriptMethod*). The value is a scriptblock marked by brackets, which contains the PowerShell instructions you want the method to perform. If you output your object, it will still look the same because PowerShell only visualizes object properties, not methods:

```
$pocketknife
Color                                    Weight   Manufacturer                      Blades
-----                                    -------  ----------                        -------
Red                                          55   Idera                                   3
```

You can add a dot and then the method name followed by two parentheses to use any of the three newly added methods. They are part of the method name, so be sure to not put a space between the method name and the opening parenthesis. Parentheses formally distinguishes properties from methods.

For example, if you’d like to remove a cork with your virtual pocketknife, you can use this code:

```
$pocketknife.corkscrew()
Pop! Cheers!
```

Your object really does carry out the exact script commands you assigned to the *corkscrew()* method. So, methods perform actions, while properties merely provide information. Always remember to add parentheses to method names. If you forget them, something interesting like this will happen:

```
# If you don’t use parentheses, you’ll retrieve information on a method:
$pocketknife.corkscrew
Script                  : “Pop! Cheers!”
OverloadDefinitions     : {System.Object corkscrew();}
MemberType              : ScriptMethod
TypeNameOfValue         : System.Object
Value                   : System.Object corkscrew();
Name                    : corkscrew
IsInstance              : True
```

You just received a method description. What’s interesting about this is mainly the *OverloadDefinitions* property. As you’ll see later, it reveals the exact way to use a command for any method. In fact, the *OverloadDefinitions* information is in an additional object. For PowerShell, absolutely everything is an object so you can store the object in a variable and then specifically ask the *OverloadDefinitions* property for information:

```
# Information about a method is returned in an object of its own:
$info = $pocketknife.corkscrew
$info.OverloadDefinitions
System.Object corkscrew();
```

The “virtual pocketknife” example reveals that objects are containers that contain data (properties) and actions (methods).

Our virtual pocketknife was a somewhat artificial object with no real use. Next, let’s take a look at a more interesting object: PowerShell! There is a variable called $host which represents your PowerShell *host*.

Properties: What an Object “Is”

There are just two important rules: Properties describe an object. And object properties are automatically turned into text when you output the object to the console. That’s enough to investigate any object. Check out the properties in $*host!*

```
$Host
Name             : ConsoleHost
Version          : 1.0.0.0
InstanceId       : e32debaf-3d10-4c4c-9bc6-ea58f8f17a8f
UI               : System.Management.Automation.Internal.Host.InternalHostUserInterface
CurrentCulture   : en-US
CurrentUICulture : en-US
PrivateData      : Microsoft.PowerShell.ConsoleHost+ConsoleColorProxy
```

The object stored in the variable $host apparently contains seven properties. The properties' names are listed in the first column. So, if you want to find out which PowerShell version you're using, you could access and return the *Version* property:

```
$Host.Version
Major  Minor  Build  Revision
-----  -----  -----  --------
1      0      0      0
```

It works—you get back the PowerShell host version. The version isn't displayed as a single number. Instead, PowerShell displays four columns: *Major, Minor, Build,* and *Revision*. Whenever you see columns, you know these are object properties that PowerShell has just converted into text. So, the version in itself is again a special object designed to store version numbers. Let's check out the data type that the *Version* property uses:

```
$version = $Host.Version
$version.GetType().FullName
System.Version
```

The version is not stored as a *String* object but as a *System*. *Version* object. This object type is perfect for storing versions, allowing you to easily read all details about any given version:

```
$Host.Version.Major
1
$Host.Version.Build
0
```

Knowing an object type is very useful because once you know there is a type called *System*. *Version*, you can use it for your own purposes as well. Try to convert a simple string of your choice into a rich version object! To do that, simply make sure the string consists of four numbers separated by dots (the typical format for versions), then make PowerShell convert the string into a System. Version type. You can convert things by adding the target type in square brackets in front of the string:

```
[System.Version]'12.55.3.28334'
Major  Minor  Build  Revision
-----  -----  -----  --------
12     55     3      28334
```

The *CurrentCulture* property is just another example of the same concept. Read this property to find out its type:

```
$Host.CurrentCulture
LCID             Name             DisplayName
----             ----             -----------
1033             en-US            English (United States)

$Host.CurrentCulture.GetType().FullName
System.Globalization.CultureInfo
```

Country properties are again stored in a highly specialized type that describes a culture with the properties *LCID*, *Name*, and *DisplayName*. If you want to know which international version of PowerShell you are using, you can read the *DisplayName* property:

```
$Host.CurrentCulture.DisplayName
English (United States)
$Host.CurrentCulture.DisplayName.GetType().FullName
System.String
```

Likewise, you can convert any suitable string into a *CultureInfo-object*. Try this if you wanted to find out details about the ‘de-DE’ locale:

```
[System.Globalization.CultureInfo]’de-DE’
LCID             Name             DisplayName
----             ----             -----------
1031             de-DE            German (Germany)
```

You can also convert the LCID into a CultureInfo object by converting a suitable number:

```
[System.Globalization.CultureInfo]1033
LCID             Name             DisplayName
----             ----             -----------
1033             en-US            English (United States)
```

Properties Containing Objects

The properties of an object store data. In turn, this data is stored in various other objects.Two properties in *$host* seem to be special: *UI* and *PrivateData*. When you output $host into the console, all other properties will be converted into readable text – except for the properties UI and PrivateData:

```
$Host
Name             : ConsoleHost
Version          : 1.0.0.0
InstanceId       : e32debaf-3d10-4c4c-9bc6-ea58f8f17a8f
UI               : System.Management.Automation.Internal.Host.InternalHostUserInterface
CurrentCulture   : en-US
CurrentUICulture : en-US
PrivateData      : Microsoft.PowerShell.ConsoleHost+ConsoleColorProxy
```

This is because both these properties again contain an object. If you'd like to find out what is actually stored in the *UI* property, you can read the property:

```
$Host.UI
RawUI
-----
System.Management.Automation.Internal.Host.InternalHostRawUserInterface
```

You see that the property *UI* contains only a single property called *RawUI*, in which yet another object is stored. Let's see what sort of object is stored in the *RawUI* property:

```
$Host.ui.rawui
ForegroundColor       : DarkYellow
BackgroundColor       : DarkMagenta
CursorPosition        : 0,136
WindowPosition        : 0,87
CursorSize            : 25
BufferSize            : 120,3000
WindowSize            : 120,50
MaxWindowSize         : 120,62
MaxPhysicalWindowSize : 140,62
KeyAvailable          : False
WindowTitle           : PowerShell
```

"RawUI" stands for "Raw User Interface" and exposes the raw user interface settings your PowerShell console uses. You can read all of these properties, but can you also change them?

Read-Only and Read-Write Properties

Can you actually change properties, too? And if you can, what happens next?

Properties need to accurately describe an object. So, if you modify a property, the underlying object has to also be modified to reflect that change. If this is not possible, the property cannot be changed and is called "read-only."

Console background and foreground colors are a great example of properties you can easily change. If you do, the console will change colors accordingly. Your property changes are reflected by the object, and the changed properties still accurately describe the object.

```
$Host.ui.rawui.BackgroundColor = “Green”
$Host.ui.rawui.ForegroundColor = “White”
```

Type *cls* so the entire console adopts this color scheme.

Other properties cannot be changed. If you try anyway, you'll get an error message:

```
$Host.ui.rawui.keyavailable = $true
"KeyAvailable" is a ReadOnly-property.
At line:1 char:16
+ $Host.ui.rawui.k <<<< eyavailable = $true
```

Whether the console receives key press input or not, depends on whether you pressed a key or not. You cannot control that by changing a property, so this property refuses to be changed. You can only read it.

Property	Description
ForegroundColor	*Text color. Optional values are Black, DarkBlue, DarkGreen, DarkCyan, DarkRed, DarkMagenta, DarkYellow, Gray, DarkGray, Blue, Green, Cyan, Red, Magenta, Yellow,* and *White.*
BackgroundColor	Background color. Optional values are *Black, DarkBlue, DarkGreen, DarkCyan, DarkRed, DarkMagenta, DarkYellow, Gray, DarkGray, Blue, Green, Cyan, Red, Magenta, Yellow,* and *White.*
CursorPosition	Current position of the cursor
WindowPosition	Current position of the window
CursorSize	Size of the cursor
BufferSize	Size of the screen buffer
WindowSize	Size of the visible window
MaxWindowSize	Maximally permissible window size
MaxPhysicalWindowSize	Maximum possible window size
KeyAvailable	Makes key press input available
WindowTitle	Text in the window title bar

Table 6.1: Properties of the RawUI object

Property Types

Some properties accept numeric values. For example, the size of a blinking cursor is specified as a number from *0* to and corresponds to the fill percentage. The next line sets a cursor size of 75%. Values outside the 0-100 numeric range will generate an error:

```
# A value from 0 to 100 is permitted:
$Host.ui.rawui.cursorsize = 75

# Values outside this range will generate an error:
$Host.ui.rawui.cursorsize = 1000
Exception setting "CursorSize": "Cannot process "CursorSize" because the cursor
    size specified is invalid.
Parameter name: value
Actual value was 1000."
At line:1 char:16
+ $Host.ui.rawui.c <<<< ursorsize = 1000
```

Other properties expect color settings. However, you cannot specify any color that comes to mind. Instead, PowerShell expects a "valid" color and if your color is unknown, you will receive an error message listing the colors you can use:

```
# Colors are specified as text (in quotation marks):
$Host.ui.rawui.ForegroundColor = "yellow"

# Not all colors are allowed:
$Host.ui.rawui.ForegroundColor = "pink"
Exception setting "ForegroundColor": "Cannot convert value "pink" to type
   "System.ConsoleColor" due to invalid enumeration values. Specify one of the
    following enumeration values and try again. The possible enumeration values are
    "Black, DarkBlue, DarkGreen, DarkCyan, DarkRed, DarkMagenta, DarkYellow, Gray,
    DarkGray, Blue, Green, Cyan, Red, Magenta, Yellow, White"."
At line:1 char:16
+ $Host.ui.rawui.F <<<< oregroundColor = "pink"
```

Tip

If you assign an invalid value to the property ForegroundColor, the error message will list the possible values. If you assign an invalid value to the property CursorSize, you get no hint. Why?

Every property expects a certain object type. Some object types are more specific than others. You can use Get-Member to find out which object types a given property will expect:

```
$Host.ui.RawUI | Get-Member -MemberType Property
   TypeName: System.Management.Automation.Internal.Host.InternalHostRawUserInterface

Name                  MemberType Definition
----                  ---------- ----------
BackgroundColor       Property   System.ConsoleColor BackgroundColor {get;set;}
BufferSize            Property   System.Management.Automation.Host.Size BufferSize {get;set;}
CursorPosition        Property   System.Management.Automation.Host.Coordinates CursorPosition {get;set;}
CursorSize            Property   System.Int32 CursorSize {get;set;}
ForegroundColor       Property   System.ConsoleColor ForegroundColor {get;set;}
KeyAvailable          Property   System.Boolean KeyAvailable {get;}
MaxPhysicalWindowSize Property   System.Management.Automation.Host.Size MaxPhysicalWindowSize {get;}
MaxWindowSize         Property   System.Management.Automation.Host.Size MaxWindowSize {get;}
WindowPosition        Property   System.Management.Automation.Host.Coordinates WindowPosition {get;set;}
WindowSize            Property   System.Management.Automation.Host.Size WindowSize {get;set;}
WindowTitle           Property   System.String WindowTitle {get;set;}
```

As you can see, ForegroundColor expects a System.ConsoleColor type. This type is a highly specialized type: a list of possible values, a so-called enumeration:

```
[system.ConsoleColor].IsEnum
True
```

Whenever a type is an enumeration, you can use a special .NET method called GetNames() to list the possible values defined in that enumeration:

```
[System.Enum]::GetNames([System.ConsoleColor])
Black
DarkBlue
DarkGreen
DarkCyan
DarkRed
DarkMagenta
DarkYellow
Gray
DarkGray
Blue
Green
Cyan
Red
Magenta
Yellow
White
```

If you do not specify anything contained in the enumeration, the error message will simply return the enumeration’s contents.

CursorSize stores its data in a System.Int32 object, which is simply a 32-bit number. So, if you try to set the cursor size to 1,000, you are actually not violating the object boundaries because the value of 1,000 can be stored in a System.Int32 object. You get an error message anyway because of the validation code that the CursorSize property executes internally. So, whether you get detailed error information will really depend on the property’s definition. In the case of CursorSize, you will receive only an indication that your value is invalid, but not why.

Sometimes, a property expects a value to be wrapped in a specific object. For example, if you’d like to change the PowerShell window size, you can use the *WindowSize* property. As it turns out, the property expects a new window size wrapped in an object of type *System.Management.Automation.Host.Size*. Where can you get an object like that?

```
$Host.ui.rawui.WindowSize = 100,100
Exception setting “WindowSize”: “Cannot convert “System.Object[]”
    to “System.Management.Automation.Host.Size”.”
At line:1 char:16
+ $Host.ui.rawui.W <<<< indowSize = 100,100
```

Tip

There are a number of ways to provide specialized objects for properties. The easiest approach: read the existing value of a property (which will get you the object type you need), change the result, and then write back the changes. For example, here’s how you would change the PowerShell window size to 80 x 30 characters:

```
$value = $Host.ui.rawui.WindowSize
$value
                                      Width                                       Height
                                      -----                                       ------
                                        110                                           64
$value.Width = 80
$value.Height = 30
$Host.ui.rawui.WindowSize = $value
```

Or, you can freshly create the object you need by using New-Object:

```
$value = New-Object System.Management.Automation.Host.Size(80,30)
$Host.ui.rawui.WindowSize = $value
```

Or in a line:

```
$host.ui.rawui.WindowSize = New-Object System.Management.Automation.Host.Size(80,30)
```

Listing All Properties

Get-Member will return detailed information about them because properties and methods are all members of an object. Let's use *Get-Member* to examine all properties defined in *$host*. To limit *Get-Member* to only properties, you can use the *memberType* parameter and specify "property":

```
$Host | Get-Member -memberType property
Name             MemberType Definition
----             ---------- ----------
CurrentCulture   Property   System.Globalization.CultureInfo CurrentCulture {get;}
CurrentUICulture Property   System.Globalization.CultureInfo CurrentUICulture {get;}
InstanceId       Property   System.Guid InstanceId {get;}
Name             Property   System.String Name {get;}
PrivateData      Property   System.Management.Automation.PSObject PrivateData {get;}
UI               Property   System.Management.Automation.Host.PSHostUserInterface UI {get;}
Version          Property   System.Version Version {get;}
```

In the column *Name*, you will now see all supported properties in *$host.* In the column *Definition*, the property object type is listed first. For example, you can see that the Name property stores a text as *System.String* type. The *Version* property uses the *System. Version* type.

At the end of each definition, curly brackets will report whether the property is read-only ({get;}) or can also be modified ({get;set;}). You can see at a glance that all properties of the *$host* object are only readable. Now, take a look at the *$host.ui.rawui* object:

```
$Host.ui.rawui | Get-Member -membertype property
BackgroundColor        Property   System.ConsoleColor BackgroundColor {get;set;}
BufferSize             Property   System.Management.Automation.Host.Size BufferSize {get;set;}
CursorPosition         Property   System.Management.Automation.Host.Coordinates CursorPosition {get;set;}
CursorSize             Property   System.Int32 CursorSize {get;set;}
ForegroundColor        Property   System.ConsoleColor ForegroundColor {get;set;}
KeyAvailable           Property   System.Boolean KeyAvailable {get;}
MaxPhysicalWindowSize Property   System.Management.Automation.Host.Size MaxPhysicalWindowSize {get;}
MaxWindowSize          Property   System.Management.Automation.Host.Size MaxWindowSize {get;}
WindowPosition         Property   System.Management.Automation.Host.Coordinates WindowPosition {get;set;}
WindowSize             Property   System.Management.Automation.Host.Size WindowSize {get;set;}
WindowTitle            Property   System.String WindowTitle {get;set;}
```

This result is more differentiated. It shows you that some properties could be changed, while others could not.

Pro Tip

There are different “sorts” of properties. Most properties are of the Property type, but PowerShell can add additional properties like *ScriptProperty*. So if you really want to list all properties, you can use the -MemberType parameter and assign it a value of **Property*. The wildcard in front of “property” will also select all specialized properties like “ScriptProperty.”

Methods: What an Object “Can Do”

Methods are things that an object *can do*.Only its properties are converted into readable text when you output an object to the console. Methods remain invisible. You can use *Get-Member* and the parameter “memberType” with the value “method” to list the methods of an object:

```
$Host | Get-Member -memberType Method

   TypeName: System.Management.Automation.Internal.Host.InternalHost

Name                     MemberType Definition
----                     ---------- ----------
EnterNestedPrompt        Method     System.Void EnterNestedPrompt()
Equals                   Method     bool Equals(System.Object obj)
ExitNestedPrompt         Method     System.Void ExitNestedPrompt()
GetHashCode              Method     int GetHashCode()
GetType                  Method     type GetType()
NotifyBeginApplication Method     System.Void NotifyBeginApplication()
NotifyEndApplication     Method     System.Void NotifyEndApplication()
PopRunspace              Method     System.Void PopRunspace()
PushRunspace             Method     System.Void PushRunspace(runspace runspace)
SetShouldExit            Method     System.Void SetShouldExit(int exitCode)
ToString                 Method     string ToString()
```

Eliminating “Internal” Methods

Get-Member does not list all methods defined by an object. It will skip methods that are used internally. You can force Get-Member to list all methods by adding the -Force parameter:

```
PS > $Host | Get-Member -memberType Method -Force


   TypeName: System.Management.Automation.Internal.Host.InternalHost

Name                   MemberType Definition
----                   ---------- ----------
(...)
get_CurrentCulture     Method     System.Globalization.CultureInfo get_Curre...
get_CurrentUICulture   Method     System.Globalization.CultureInfo get_Curre...
get_InstanceId         Method     System.Guid get_InstanceId()
get_IsRunspacePushed   Method     bool get_IsRunspacePushed()
get_Name               Method     string get_Name()
get_PrivateData        Method     psobject get_PrivateData()
get_Runspace           Method     runspace get_Runspace()
get_UI                 Method     System.Management.Automation.Host.PSHostUs...
get_Version            Method     System.Version get_Version()
(...)
```

Get_ and Set_ Methods

Any method that starts with “get_” is really designed to retrieve a property value. So the method “get_someInfo()” will retrieve the very same information you could also have gotten with the “someInfo” property.

```
# Query property:
$Host.version

Major  Minor  Build  Revision
-----  -----  -----  --------
2      0      -1     -1

# Query property value using getter method:
$Host.get_Version()
Major  Minor  Build  Revision
-----  -----  -----  --------
2      0      -1     -1
```

The same is true for Set_ methods: they change a property value and exist for properties that are read/writeable. Note in this example: all properties of the $host object can only be read so there are no Set_ methods. There can be more internal methods like this, such as Add_ and Remove_ methods. Generally speaking, when a method name contains an underscore, it is most likely an internal method.

Standard Methods

In addition, nearly every object contains a number of “inherited” methods that are also not specific to the object but perform general tasks for every object:

Method	Description
Equals	Verifies whether the object is identical to a comparison object
GetHashCode	Retrieves an object’s digital “fingerprint”
GetType	Retrieves the underlying object type
ToString	Converts the object into readable text

Table 6.2: Standard methods of a .NET object

Calling a Method

Before you invoke a method: make sure you know what the method will do. Methods are commands that do something, which could be dangerous. You can add a dot to the object and then the method name to call a method. Add an opened and closed parenthesis, like this:

```
$host.EnterNestedPrompt()
```

The PowerShell prompt changes to “>>” (unless you changed your default prompt function). You have used *EnterNestedPrompt()* to open a nested prompt. Nested prompts are not especially useful in a normal console, so be sure to exit it again using the exit command or call *$host.ExitNestedPrompt().*

Nested prompts can be useful in functions or scripts because they work like breakpoints. They can temporarily stop a function or script so you can verify variable contents or make code changes, after which you continue the code by entering exit. You’ll learn more about this in **Chapter 11**.

Call Methods with Arguments

There are many useful methods in the *UI* object. Here’s how you get a good overview:

```
$Host.ui | Get-Member -membertype Method
   TypeName: System.Management.Automation.Internal.Host.InternalHostUserInterface

Name                      MemberType Definition
----                      ---------- ----------
Equals                    Method     System.Boolean Equals(Object obj)
GetHashCode               Method     System.Int32 GetHashCode()
GetType                   Method     System.Type GetType()
get_RawUI                 Method     System.Management.Automation.Host.PSHostRawUserInterface get_RawUI()
Prompt                    Method     System.Collections.Generic.Dictionary`2[[System.String, mscorlib, Versio...
```

```
PromptForChoice        Method     System.Int32 PromptForChoice(String caption, String message, Collection`...
PromptForCredential    Method     System.Management.Automation.PSCredential PromptForCredential(String cap...
ReadLine               Method     System.String ReadLine()
ReadLineAsSecureString Method     System.Security.SecureString ReadLineAsSecureString()
ToString               Method     System.String ToString()
Write                  Method     System.Void Write(String value), System.Void Write(ConsoleColor foregrou...
WriteDebugLine         Method     System.Void WriteDebugLine(String message)
WriteErrorLine         Method     System.Void WriteErrorLine(String value)
WriteLine              Method     System.Void WriteLine(), System.Void WriteLine(String value), System.Voi...
WriteProgress          Method     System.Void WriteProgress(Int64 sourceId, ProgressRecord record)
WriteVerboseLine       Method     System.Void WriteVerboseLine(String message)
WriteWarningLine       Method     System.Void WriteWarningLine(String message)
```

Most methods require additional arguments from you, which are listed in the *Definition* column.

Which Arguments are Required?

Pick out a method from the list, and then ask *Get-Member* to get you more info. Let’s pick *WriteDebugLine()*:

```
# Ask for data on the WriteDebugLine method in $host.ui:
$info = $Host.UI | Get-Member WriteDebugLine

# $info contains all the data on this method:
$info
   TypeName: System.Management.Automation.Internal.Host.InternalHostUserInterface
Name           MemberType Definition
----           ---------- ----------
WriteDebugLine Method     System.Void WriteDebugLine(String message)

# Definition shows which arguments are required and which result will be returned:
$info.Definition
System.Void WriteDebugLine(String message)
```

The Definition property tells you how to call the method. Every definition will begin with the object type that a method returns. In this example, it is System.Void, a special object type because it represents “nothing”: the method doesn’t return anything at all. A method “returning” System.Void is really a procedure, not a function.

Next, a method’s name follows, which is then followed by required arguments. WriteDebugLine needs exactly one argument called message, which is of String type. Here is how you call WriteDebugLine():

```
$Host.ui.WriteDebugLine(“Hello!”)
Hello!
```

Several Method “Signatures”

Some methods accept different argument types, or even different numbers of arguments. To find out which “signatures” a method supports, you can use *Get-Member* again and look at the *Definition* property:

```
$info = $Host.UI | Get-Member WriteLine
$info.Definition
System.Void WriteLine(), System.Void WriteLine(String value), System.Void
    WriteLine(ConsoleColor foregroundColor, ConsoleColor backgroundColor, String value)
```

The definition is hard to read at first. You can make it more readable by using *Replace()* to add line breaks.

Tip

Remember the “backtick” character (“`”). It introduces special characters; “`n” stands for a line break.

```
$info.Definition.Replace("), ", ")`n")
System.Void WriteLine()
System.Void WriteLine(String value)
System.Void WriteLine(ConsoleColor foregroundColor, ConsoleColor backgroundColor, String value)
```

This definition tells you: You do not necessarily need to supply arguments:

```
$host.ui.WriteLine()
```

The result is an empty line.

To output text, you can specify one argument only, the text itself:

```
$Host.ui.WriteLine("Hello world!")
Hello world!
```

The third variant adds support for foreground and background colors:

```
$host.ui.WriteLine("Red", "White", "Alarm!")
```

WriteLine() actually is the low-level function of the *Write-Host cmdlet:*

```
Write-Host
Write-Host "Hello World!"
Write-Host -ForegroundColor Red -BackgroundColor White Alarm!
```

Playing with Prompt For Choice

So far, most methods you examined have turned out to be low-level commands for cmdlets. This is also true for the following methods: *Write()* (corresponds to *Write-Host -nonewline*) or *ReadLine()/ReadLineAsSecureString()* (read-host *-asSecureString*) or *PromptForCredential() (get-credential)*.

A new functionality is exposed by the method *PromptForChoice()*. Let’s first examine which arguments this method expects:

```
$info = $Host.UI | Get-Member PromptForChoice
$info.Definition
System.Int32 PromptForChoice(String caption, String message, Collection`1 choices,
    Int32 defaultChoice)
```

Tip

You can get the same information if you call the method without parentheses:

```
You can get the same information if you call the method without parentheses:
$Host.ui.PromptForChoice
MemberType           : Method
OverloadDefinitions  : {System.Int32 PromptForChoice(String caption, String message,
     Collection`1 choices, Int 32 defaultChoice)}
TypeNameOfValue      : System.Management.Automation.PSMethod
Value                : System.Int32 PromptForChoice(String caption, String message,
     Collection`1 choices, Int32 defaultChoice)
Name                 : PromptForChoice
IsInstance           : True
```

The definition reveals that this method returns a numeric value *(System.Int32)*. It requires a heading and a message respectively as text *(String)*. The third argument is a bit strange: *Collection`1 choices*. The fourth argument is a number *(Int32)*, the standard selection. You may have noticed by now the limitations of PowerShell’s built-in description.

This is how you can use *PromptForChoice()* to create a simple menu:

```
$yes = ([System.Management.Automation.Host.ChoiceDescription]”&yes”)
$no = ([System.Management.Automation.Host.ChoiceDescription]”&no”)
$selection = [System.Management.Automation.Host.ChoiceDescription[]]($yes,$no)
$answer = $Host.ui.PromptForChoice(‘Reboot’, ‘May the system now be rebooted?’,$selection,1)
$selection[$answer]
if ($selection -eq 0) {
  “Reboot”
} else {
  “OK, then not”
}
```

Working with Real-Life Objects

Every PowerShell command will return objects. However, it is not that easy to get your hands on objects because PowerShell converts them to text whenever you output them to the console.

Storing Results in Variables

Save the result to variable to examine the object nature of results you receive from cmdlets.

```
$listing = Dir $env:windir
```

When you dump the variable content to the console, the results stored inside of it will be converted to plain text, much like if you had output the information to the console in the first place:

```
$listing
    Directory: Microsoft.PowerShell.Core\FileSystem::C:\Users\Tobias Weltner
Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        20.07.2007     11:37            Application data
d----        26.07.2007     11:03            Backup
d-r--        13.04.2007     15:05            Contacts
d----        28.06.2007     18:33            Debug
(...)
```

To get to the real objects, you can directly access them inside of a variable. *Dir* has stored its result in $listing. It is wrapped in an array since the *listing* consists of more than one entry. Access an array element to get your hands on a real object:

```
# Access first element in listing
$object = $listing[0]

# Object is converted into text when you output it in the console
$object
    Directory: Microsoft.PowerShell.Core\FileSystem::C:\Users\Tobias Weltner
Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        20.07.2007     11:37            Application data
```

The object picked here happens to match the folder *Application Data*; so it represents a directory. You can do this if you prefer to directly pick a particular directory or file:

```
# Address a particular file:
$object = Get-Item $env:windir\explorer.exe

# Address a folder:
$object = Get-Item $env:windir
```

Using Results Properties

You can use *Get-Member* again to produce a list of all available properties:

```
# $object is a fully functional object that describes the “Application Data” directory
# First, list all object properties:
$object | Get-Member -membertype *property
Name              MemberType     Definition
----              ----------     ----------
Mode              CodeProperty   System.String Mode{get=Mode;}
PSChildName       NoteProperty   System.String PSChildName=Windows
PSDrive           NoteProperty   System.Management.Automation.PSDriveInfo PS...
PSIsContainer     NoteProperty   System.Boolean PSIsContainer=True
PSParentPath      NoteProperty   System.String PSParentPath=Microsoft.PowerS...
PSPath            NoteProperty   System.String PSPath=Microsoft.PowerShell.C...
PSProvider        NoteProperty   System.Management.Automation.ProviderInfo P...
Attributes        Property       System.IO.FileAttributes Attributes {get;set;}
CreationTime      Property       System.DateTime CreationTime {get;set;}
CreationTimeUtc   Property       System.DateTime CreationTimeUtc {get;set;}
Exists            Property       System.Boolean Exists {get;}
Extension         Property       System.String Extension {get;}
FullName          Property       System.String FullName {get;}
LastAccessTime    Property       System.DateTime LastAccessTime {get;set;}
LastAccessTimeUtc Property       System.DateTime LastAccessTimeUtc {get;set;}
LastWriteTime     Property       System.DateTime LastWriteTime {get;set;}
LastWriteTimeUtc  Property       System.DateTime LastWriteTimeUtc {get;set;}
Name              Property       System.String Name {get;}
Parent            Property       System.IO.DirectoryInfo Parent {get;}
Root              Property       System.IO.DirectoryInfo Root {get;}
BaseName          ScriptProperty System.Object BaseName {get=$this.Name;}
```

Properties marked with *{get;set;}* in the column *Definition* are readable and writeable. You can actually change their value, too, by simply assigning a new value (provided you have sufficient privileges):

```
# Determine last access date:
$object.LastAccessTime
Friday, July 20, 2007 11:37:39
```

```
# Change Date:
$object.LastAccessTime = Get-Date

# Change was accepted:
$object.LastAccessTime
Monday, October 1, 2007 15:31:41
```

PowerShell-Specific Properties

PowerShell can add additional properties to an object. Whenever that occurs, *Get-Member* will label the property accordingly in the *MemberType* column. Native properties are just called *"Property."* Properties that are added by PowerShell use a prefix, such as "ScriptProperty" or "NoteProperty."

A *NoteProperty* like *PSChildName* contains static data. PowerShell will add it to tag additional information to an object. A ScriptProperty like Mode executes PowerShell script code that calculates the property's value.

MemberType	**Description**
AliasProperty	Alternative name for a property that already exists
CodeProperty	Static .NET method returns property contents
Property	Genuine property
NoteProperty	Subsequently added property with set data value
ScriptProperty	Subsequently added property whose value is calculated by a script
ParameterizedProperty	Property requiring additional arguments

Table 6.3: Different property types

Using Objects Methods

Use *Get-Member* to find out the methods that an object supports:

```
# List all methods of the object:
$object | Get-Member -membertype *method
TypeName: System.IO.DirectoryInfo

Name                       MemberType Definition
----                       ---------- ----------
Create                     Method     System.Void Create(), System.Void Create(DirectorySecurity DirectoryS...
CreateObjRef               Method     System.Runtime.Remoting.ObjRef CreateObjRef(Type requestedType)
CreateSubDirectory         Method     System.IO.DirectoryInfo CreateSubDirectory(String path), System.IO.Di...
Delete                     Method     System.Void Delete(), System.Void Delete(Boolean recursive)
Equals                     Method     System.Boolean Equals(Object obj)
```

```
GetAccessControl          Method     System.Security.AccessControl.DirectorySecurity GetAccessControl(), S...
GetDirectories            Method     System.IO.DirectoryInfo[] GetDirectories(), System.IO.DirectoryInfo[]...
GetFiles                  Method     System.IO.FileInfo[] GetFiles(String searchPattern), System.IO.FileIn...
GetFileSystemInfos        Method     System.IO.FileSystemInfo[] GetFileSystemInfos(String searchPattern), ...
GetHashCode               Method     System.Int32 GetHashCode()
GetLifetimeService        Method     System.Object GetLifetimeService()
GetObjectData             Method     System.Void GetObjectData(SerializationInfo info, StreamingContext co...
GetType                   Method     System.Type GetType()
get_Attributes            Method     System.IO.FileAttributes get_Attributes()
get_CreationTime          Method     System.DateTime get_CreationTime()
get_CreationTimeUtc       Method     System.DateTime get_CreationTimeUtc()
get_Exists                Method     System.Boolean get_Exists()
get_Extension             Method     System.String get_Extension()
get_FullName              Method     System.String get_FullName()
get_LastAccessTime        Method     System.DateTime get_LastAccessTime()
get_LastAccessTimeUtc     Method     System.DateTime get_LastAccessTimeUtc()
get_LastWriteTime         Method     System.DateTime get_LastWriteTime()
get_LastWriteTimeUtc      Method     System.DateTime get_LastWriteTimeUtc()
get_Name                  Method     System.String get_Name()
get_Parent                Method     System.IO.DirectoryInfo get_Parent()
get_Root                  Method     System.IO.DirectoryInfo get_Root()
InitializeLifetimeService Method     System.Object InitializeLifetimeService()
MoveTo                    Method     System.Void MoveTo(String destDirName)
Refresh                   Method     System.Void Refresh()
SetAccessControl          Method     System.Void SetAccessControl(DirectorySecurity DirectorySecurity)
set_Attributes            Method     System.Void set_Attributes(FileAttributes value)
set_CreationTime          Method     System.Void set_CreationTime(DateTime value)
set_CreationTimeUtc       Method     System.Void set_CreationTimeUtc(DateTime value)
set_LastAccessTime        Method     System.Void set_LastAccessTime(DateTime value)
set_LastAccessTimeUtc     Method     System.Void set_LastAccessTimeUtc(DateTime value)
set_LastWriteTime         Method     System.Void set_LastWriteTime(DateTime value)
set_LastWriteTimeUtc      Method     System.Void set_LastWriteTimeUtc(DateTime value)
ToString                  Method     System.String ToString()
```

You can apply methods just like you did in the previous examples. For example, you can use the *CreateSubDirectory* method if you'd like to create a new sub-directory. First, you should find out which arguments this method requires and what it returns:

```
$info = $object | Get-Member CreateSubDirectory
$info.Definition.Replace("), ", ")`n")
System.IO.DirectoryInfo CreateSubDirectory(String path)
System.IO.DirectoryInfo CreateSubDirectory(String path, DirectorySecurity DirectorySecurity)
```

You can see that the method has two signatures. Try using the first to create a sub-directory and the second to add access permissions.

The next line creates a sub-directory called "My New Directory" without any special access privileges:

```
$object.CreateSubDirectory("My New Directory")
Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----         01.10.2007     15:49           My New Directory
```

Because the method returns a *DirectoryInfo* object as a result and you haven't caught and stored this object in a variable, the pipeline will convert it into text and output it. You could just as well have stored the result of the method in a variable:

```
$subdirectory = $object.CreateSubDirectory("Another subdirectory")
$subdirectory.CreationTime = "September 1, 1980"
$subdirectory.CreationTime
Monday, September 1, 1980 00:00:00
```

Using Results Properties

Similarly to properties, PowerShell can also add additional methods to an object.

MemberType	Description
CodeMethod	Method mapped to a static .NET method
Method	Genuine method
ScriptMethod	Method invokes PowerShell code

Table 6.4: Different types of methods

Using Static Methods

By now, you know that PowerShell stores information in objects, and objects always have a type. You know that simple text is stored in objects of type *System.String* and that a date, for example, is stored in an object of type *System.DateTime*. You also know by now that each .NET object has a *GetType()* method with a Fullname property, which tells you the name of the type this object was derived from:

```
$date = Get-Date
$date.GetType().FullName
System.DateTime
```

Every type can have its own set of private members called "static" members. You can simply specify a type in square brackets, pipe it to *Get-Member*, and then use the -static parameter to see the static members of a type.

```
[System.DateTime] | Get-Member -static -memberType *method
   TypeName: System.DateTime
Name                  MemberType Definition
----                  ---------- ----------
Compare               Method     static System.Int32 Compare(DateTime t1, DateTime t2)
DaysInMonth           Method     static System.Int32 DaysInMonth(Int32 year, Int32 month)
Equals                Method     static System.Boolean Equals(DateTime t1, DateTime t2), static Sys...
```

```
FromBinary            Method     static System.DateTime FromBinary(Int64 dateData)
FromFileTime          Method     static System.DateTime FromFileTime(Int64 fileTime)
FromFileTimeUtc       Method     static System.DateTime FromFileTimeUtc(Int64 fileTime)
FromOADate            Method     static System.DateTime FromOADate(Double d)
get_Now               Method     static System.DateTime get_Now()
get_Today             Method     static System.DateTime get_Today()
get_UtcNow            Method     static System.DateTime get_UtcNow()
IsLeapYear            Method     static System.Boolean IsLeapYear(Int32 year)
op_Addition           Method     static System.DateTime op_Addition(DateTime d, TimeSpan t)
op_Equality           Method     static System.Boolean op_Equality(DateTime d1, DateTime d2)
op_GreaterThan        Method     static System.Boolean op_GreaterThan(DateTime t1, DateTime t2)
op_GreaterThanOrEqual Method     static System.Boolean op_GreaterThanOrEqual(DateTime t1, DateTime t2)
op_Inequality         Method     static System.Boolean op_Inequality(DateTime d1, DateTime d2)
op_LessThan           Method     static System.Boolean op_LessThan(DateTime t1, DateTime t2)
op_LessThanOrEqual    Method     static System.Boolean op_LessThanOrEqual(DateTime t1, DateTime t2)
op_Subtraction        Method     static System.DateTime op_Subtraction(DateTime d, TimeSpan t), sta...
Parse                 Method     static System.DateTime Parse(String s), static System.DateTime Par...
ParseExact            Method     static System.DateTime ParseExact(String s, String format, IFormat...
ReferenceEquals       Method     static System.Boolean ReferenceEquals(Object objA, Object objB)
SpecifyKind           Method     static System.DateTime SpecifyKind(DateTime value, DateTimeKind kind)
TryParse              Method     static System.Boolean TryParse(String s, DateTime& result), static...
TryParseExact         Method     static System.Boolean TryParseExact(String s, String format, IForm...
```

Note

There are a lot of method names starting with “op_,” with “op” standing for “operator.” These are methods that are called internally whenever you use this data type with an operator. *op_GreaterThanOrEqual* is the method that does the internal work when you use the PowerShell comparison operator *“-ge”* with date values.

The *System.DateTime* class supplies you with a bunch of important date and time methods. For example, you should use Parse() to convert a date string into a real DateTime object and the current locale:

```
[System.DateTime]::Parse(“March 12, 1999”)
Friday, March 12, 1999 00:00:00
```

You could easily find out whether a certain year is a leap year:

```
[System.DateTime]::isLeapYear(2010)
False

for ($x=2000; $x -lt 2010; $x++) { if( [System.DateTime]::isLeapYear($x) ) { “$x is a leap year!” } }
2000 is a leap year!
2004 is a leap year!
2008 is a leap year!
```

Or you’d like to tell your children with absolute precision how much time will elapse before they get their Christmas gifts:

```
[DateTime]”12/24/2007 18:00” - [DateTime]::now
Days              : 74
Hours             : 6
Minutes           : 28
Seconds           : 49
Milliseconds      : 215
Ticks             : 64169292156000
TotalDays         : 74.2700140694444
TotalHours        : 1782,48033766667
TotalMinutes      : 106948,82026
TotalSeconds      : 6416929,2156
TotalMilliseconds : 6416929215,6
```

Two dates are being subtracted from each other here so you now know what happened during this operation:

The first time indication is actually text. For it to become a *DateTime* object, you must specify the desired object type in square brackets. Important: *Converting a String to a DateTime this way always uses the U.S. locale. To convert a String to a DateTime using your current locale, you can use the Parse() method as shown a couple of moments ago!*

- The second time comes from the *Now static* property, which returns the current time as *DateTime* object. This is the same as calling the *Get-Date* cmdlet (which you’d then need to put in parenthesis because you wouldn’t want to subtract the *Get-Date* cmdlet, but rather the result of the *Get-Date* cmdlet).
- The two timestamps are subtracted from each other using the subtraction operator (“-”). This was possible because the *DateTime* class defined the *op_Subtraction()* static method, which is needed for this operator.

Of course, you could have called the static method yourself and received the same result:

```
[DateTime]::op_Subtraction(“12/24/2007 18:00”, [DateTime]::Now)
```

Now it’s your turn. In the *System.Math* class, you’ll find a lot of useful mathematical methods. Try to put some of these methods to work.

Function	**Description**	**Example**
Abs	Returns the absolute value of a specified number (without signs).	[Math]::Abs(-5)
Acos	Returns the angle whose cosine is the specified number.	[Math]::Acos(0.6)
Asin	Returns the angle whose sine is the specified number.	[Math]::Asin(0.6)
Atan	Returns the angle whose tangent is the specified number.	[Math]::Atan(90)
Atan2	Returns the angle whose tangent is the quotient of two specified numbers.	[Math]::Atan2(90, 15)
BigMul	Calculates the complete product of two 32-bit numbers.	[Math]::BigMul(1gb, 6)
Ceiling	Returns the smallest integer greater than or equal to the specified number.	[Math]::Ceiling(5.7)
Cos	Returns the cosine of the specified angle.	[Math]::Cos(90)
Cosh	Returns the hyperbolic cosine of the specified angle.	[Math]::Cosh(90)

Function	Description	Example
DivRem	Calculates the quotient of two numbers and returns the remainder in an output parameter.	$a = 0 [Math]::DivRem(10,3,[ref]$a) $a
Exp	Returns the specified power of e (2.7182818).	[Math]::Exp(12)
Floor	Returns the largest integer less than or equal to the specified number.	[Math]::Floor(5.7)
IEEERemainder	Returns the remainder of division of two specified numbers.	[Math]::IEEERemainder(5,2)
Log	Returns the natural logarithm of the specified number.	[Math]::Log(1)
Log10	Returns the base 10 logarithm of the specified number.	[Math]::Log10(6)
Max	Returns the larger of two specified numbers.	[Math]::Max(-5, 12)
Min	Returns the smaller of two specified numbers.	[Math]::Min(-5, 12)
Pow	Returns a specified number raised to the specified power.	[Math]::Pow(6,2)
Round	Rounds a value to the nearest integer or to the specified number of decimal places.	[Math]::Round(5.51)
Sign	Returns a value indicating the sign of a number.	[Math]::Sign(-12)
Sin	Returns the sine of the specified angle.	[Math]::Sin(90)
Sinh	Returns the hyperbolic sine of the specified angle.	[Math]::Sinh(90)
Sqrt	Returns the square root of a specified number.	[Math]::Sqrt(64)
Tan	Returns the tangent of the specified angle.	[Math]::Tan(45)
Tanh	Returns the hyperbolic tangent of the specified angle.	[Math]::Tanh(45)
Truncate	Calculates the integral part of a number.	[Math]::Truncate(5.67)

Table 6.5: Mathematical functions from the [Math] library

Finding Interesting .NET Types

The .NET framework consists of thousands of types, and maybe you are getting hungry for more. Are there other interesting types? There are actually plenty! Here are the three things you can do with .NET types:

Converting Object Types

For example, you can use *System.Net.IPAddress* to work with IP addresses. This is an example of a .NET type conversion where a string is converted into a *System.Net.IPAddress type:*

```
[system.Net.IPAddress]’127.0.0.1’

IPAddressToString : 127.0.0.1
Address           : 16777343
AddressFamily     : InterNetwork
ScopeId           :
IsIPv6Multicast   : False
IsIPv6LinkLocal   : False
IsIPv6SiteLocal   : False
```

Using Static Type Members

Or you can use *System.Net.DNS* to resolve host names. This is an example of accessing a static type method, such as *GetHostByAddress()*:

```
[system.Net.Dns]::GetHostByAddress(“127.0.0.1”)

HostName          Aliases          AddressList
--------          -------          -----------
PCNEU01           {}               {127.0.0.1}
```

Using Dynamic Object Instance Members

Or you can derive an instance of a type and use its dynamic members. For example, to download a file from the Internet, try this:

```
# Download address of a file:
$address = “http://www.powershell.com/downloads/powershellplus.zip”

# Save the file to this location:
$target = “$home\psplus.zip”

# Carry out download:
$object = New-Object Net.WebClient
$object.DownloadFile($address, $target)
“File was downloaded!”
```

Creating New Objects

Most of the time, PowerShell cmdlets deliver objects. In addition, you can create new objects (instances) that are derived from a specific type. To get new instances, you can either convert an existing object to a new type or create a new instance using New-Object:

```
$datetime = [System.DateTime] ‘1.1.2000’
$datetime.GetType().Fullname
System.DateTime
$datetime = New-Object System.DateTime
$datetime.GetType().Fullname
System.DateTime
$datetime = Get-Date
$datetime.GetType().Fullname
System.DateTime
$datetime = [System.DateTime]::Parse(‘1.1.2000’)
$datetime.GetType().Fullname
System.DateTime
```

Creating New Objects with New-Object

You can create a .NET object with *New-Object,t* which gives you full access to all type "constructors." These are invisible methods that create the new object. the type needs to have at least one constructor to create a new instance of a type. If it has none, you cannot create instances of this type.

The DateTime type has one constructor that takes no argument. If you create a new instance of a DateTime object, you will get back a date set to the very first date a DateTime type can represent, which happens to be January 1, 0001:

```
New-Object System.DateTime
Monday, January 01, 0001 12:00:00 AM
```

You can use a different constructor to create a specific date. There is one that takes three numbers for year, month, and day:

```
New-Object System.DateTime
Monday, May 01, 0001 12:00:00 AM
```

If you simply add a number, yet another constructor is used which interprets the number as ticks, the smallest time unit a computer can process:

```
New-Object System.DateTime (568687676789080999)
Monday, February 07, 1803 7:54:38 AM
```

Using Static Type Members

When you create a new object using New-Object, you can submit additional arguments by adding argument values as a comma separated list enclosed in parentheses. New-Object is in fact calling a method called ctor, which is the type constructor. Like any other method, it can support different argument signatures.

Let's check out how you can discover the different constructors, which a type will support. The next line creates a new instance of a System.String and uses a constructor that accepts a character and a number:

```
New-Object System.String(".", 100)
....................................................................................................
```

To list the available constructors for a type, you can use the GetConstructors() method available in each type. For example, you can find out which constructors are offered by the *System.String* type to produce *System.String* objects:

```
[System.String].GetConstructors() | ForEach-Object { $_.toString() }
Void .ctor(Char*)
Void .ctor(Char*, Int32, Int32)
Void .ctor(SByte*)
Void .ctor(SByte*, Int32, Int32)
Void .ctor(SByte*, Int32, Int32, System.Text.Encoding)
Void .ctor(Char[], Int32, Int32)
Void .ctor(Char[])
Void .ctor(Char, Int32)
```

In fact, there are eight different signatures to create a new object of the *System.String type*. You just used the last variant: the first argument is the character, and the second a number that specifies how often the character will be repeated. PowerShell will use the next to last constructor so if you specify text in quotation marks, it will interpret text in quotation marks as a field with nothing but characters *(Char[]).*

New-Object by Conversion

Objects can often be created without New-Object by using type casting instead. You've already seen how it's done for variables in **Chapter 3**:

```
# PowerShell normally wraps text as a System.String:
$date = “November 1, 2007”
$date.GetType().FullName
System.String
$date
November 1, 2007

# Use strong typing to set the object type of $date:
[System.DateTime]$date = “November 1, 2007”
$date.GetType().FullName
System.DateTime
$date
Thursday, November 1, 2007 00:00:00
```

So, if you enclose the desired .NET type in square brackets and put it in front of a variable name, PowerShell will require you to use precisely the specified object type for this variable. If you assign a value to the variable, PowerShell will automatically convert it to that type. That process is sometimes called "implicit type conversion." Explicit type conversion works a little different. Here, the desired type is put in square brackets again, but placed on the right side of the assignment operator:

```
$value = [DateTime]”November 1, 2007”
$value
Thursday, November 1, 2007 00:00:00
```

PowerShell would first convert the text into a date because of the type specification and then assign it to the variable $value, which itself remains a regular variable without type specification. Because *$value* is not limited to DateTime types, you can assign other data types to the variable later on.

```
$value = “McGuffin”
```

Using the type casting, you can also create entirely new objects without *New-Object*. First, create an object using New-Object:

```
New-Object system.diagnostics.eventlog(“System”)
  Max(K) Retain OverflowAction       Entries Name
  ------ ------ --------------        -------- ----
  20,480      0 OverwriteAsNeeded      64,230 System
```

You could have accomplished the same thing without *New-Object*:

```
[System.Diagnostics.EventLog]”System”
  Max(K) Retain OverflowAction        Entries Name
  ------ ------ --------------        -------- ----
  20,480      0 OverwriteAsNeeded      64,230 System
```

In the second example, the string System is converted into the *System.Diagnostics.Eventlog* type: The result is an *EventLog* object representing the *System* event log.

So, when can you use New-Object and when type conversion? It is largely a matter of taste, but whenever a type has more than one constructor and you want to select the constructor, you should use New-Object and specify the arguments for the constructor of your choice. Type conversion will automatically choose one constructor, and you have no control over which constructor is picked.

```
# Using New-Object, you can select the constructor you wish of the type yourself:
New-Object System.String(“.”, 100)
..........................................................................................

# When casting types, PowerShell selects the constructor automatically
# For the System.String type, a constructor will be chosen that requires no arguments
# Your arguments will then be interpreted as a PowerShell subexpression in which
# a field will be created
# PowerShell will change this field into a System.String type
# PowerShell changes fields into text by separating elements from each other with whitespace:
[system.string](“.”,100)
. 100
# If your arguments are not in round brackets, they will be interpreted as a Field
# and the first field element # Cast in the System.String type:
[system.string]”.”, 100
.
100
```

Tip

Type conversion can also include type arrays (identified by “[]”) and can be a multi-step process where you convert from one type over another type to a final type. This is how you would convert string text into a character array:

```
[char[]]”Hello!”
H
e
l
l
o
!
```

Tip

You could then convert each character into integers to get the character codes:

```
[Int[]][Char[]]”Hello World!”
72
97
108
108
111
32
87
101
108
116
33
```

Conversely, you could make a numeric list out of a numeric array and turn that into a string:

```
[string][char[]](65..90)
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
$OFS = “,”
[string][char[]](65..90)
A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z
```

Just remember: if arrays are converted into a string, PowerShell uses the separator in the $ofs automatic variable as a separator between the array elements.

Loading Additional Assemblies: Improved Internet Download

To get access to even more functionality, you can load additional assemblies with more types and members. If you have ever written VBScript scripts, you may want to get back some of your beloved VisualBasic methods, such as MsgBox() or InputBox(). Simply load the Microsoft.VisualBasic assembly, which is located in the global assembly cache:

```
# Load required assembly:
[void][reflection.assembly]::LoadWithPartialName(“Microsoft.VisualBasic”)
```

Once you do that, you have access to a whole bunch of new types:

```
[Microsoft.VisualBasic.Interaction] | Get-Member -static

   TypeName: Microsoft.VisualBasic.Interaction

Name             MemberType Definition
----             ---------- ----------
AppActivate      Method     static System.Void AppActivate(Int32 Proces...
Beep            Method     static System.Void Beep()
CallByName      Method     static System.Object CallByName(Object Obje...
Choose          Method     static System.Object Choose(Double Index, P...
Command         Method     static System.String Command()
CreateObject    Method     static System.Object CreateObject(String Pr...
DeleteSetting   Method     static System.Void DeleteSetting(String App...
Environ         Method     static System.String Environ(Int32 Expressi...
Equals          Method     static System.Boolean Equals(Object objA, O...
GetAllSettings  Method     static System.String[,] GetAllSettings(Stri...
GetObject       Method     static System.Object GetObject(String PathN...
GetSetting      Method     static System.String GetSetting(String AppN...
IIf             Method     static System.Object IIf(Boolean Expression...
InputBox        Method     static System.String InputBox(String Prompt...
MsgBox          Method     static Microsoft.VisualBasic.MsgBoxResult M...
Partition       Method     static System.String Partition(Int64 Number...
ReferenceEquals Method     static System.Boolean ReferenceEquals(Objec...
SaveSetting     Method     static System.Void SaveSetting(String AppNa...
Shell           Method     static System.Int32 Shell(String PathName, ...
switch          Method     static System.Object switch(Params Object[]...
[microsoft.VisualBasic.Interaction]::InputBox(“Enter Name”, “Name”, “$env:username”)
Tobias
```

Or, you can use a much-improved download method, which shows a progress bar while downloading files from the Internet:

```
# Reload required assembly:
[void][reflection.assembly]::LoadWithPartialName(“Microsoft.VisualBasic”)

# Download address of a file:
$address = “http://www.idera.com/powershellplus”

# This is where the file should be saved:
$target = “$home\psplus.zip”

# Download will be carried out:
$object = New-Object Microsoft.VisualBasic.Devices.Network
$object.DownloadFile($address, $target, “”, “”, $true, 500, $true, “DoNothing”)
```

Using COM Objects

In addition to .NET, PowerShell can also load and access most COM objects, which work similar to .NET types and objects, but use an older technology.

Using Static Type Members

COM objects each have a unique name, known as *ProgID* or *Programmatic Identifier,* which is stored in the registry. So, if you want to look up COM objects available on your computer, you can visit the registry:

```
Dir REGISTRY::HKEY_CLASSES_ROOT\CLSID -include PROGID -recurse | foreach {$_.GetValue(“”)}
```

How Do You Use COM Objects?

Once you know the *ProgID* of a COM component, you can use *New-Object* to put it to work in PowerShell. Just specify the additional parameter *-COMObject*:

```
$object = New-Object -ComObject WScript.Shell
```

You’ll get an object which behaves very similar to .NET objects. It will contain properties with data and methods that you can execute. And, as always, *Get-Member* finds all object members for you. Let’s look at its methods:

```
# Make the methods of the COM objects visible:
$object | Get-Member -memberType *method
   TypeName: System.__ComObject#{41904400-be18-11d3-a28b-00104bd35090}

Name                     MemberType Definition
----                     ---------- ----------
AppActivate              Method     bool AppActivate (Variant, Variant)
CreateShortcut           Method     IDispatch CreateShortcut (string)
Exec                     Method     IWshExec Exec (string)
ExpandEnvironmentStrings Method     string ExpandEnvironmentStrings (string)
LogEvent                 Method     bool LogEvent (Variant, string, string)
Popup                    Method     int Popup (string, Variant, Variant, Variant)
RegDelete                Method     void RegDelete (string)
RegRead                  Method     Variant RegRead (string)
RegWrite                 Method     void RegWrite (string, Variant, Variant)
Run                      Method     int Run (string, Variant, Variant)
SendKeys                 Method     void SendKeys (string, Variant)
```

The information required to understand how to use a method may be inadequate. Only the expected object types are given, but not why the arguments exist. The Internet can help you if you want to know more about a COM command. Go to a search site of your choice and enter two keywords: the ProgID of the COM components (in this case, it will be WScript.Shell) and the name of the method that you want to use.

Some of the commonly used COM objects are WScript.Shell, WScript.Network, Scripting.FileSystemObject, InternetExplorer.Application, Word.Application, and Shell.Application. Let’s create a shortcut to powershell.exe using WScript.Shell Com object and its method CreateShorcut():

```
# Create an object:
$wshell = New-Object -comObject WScript.Shell
# Assign a path to Desktop to the variable $path
$path = [system.Environment]::GetFolderPath(‘Desktop’)

# Create a link object $link = $wshell.CreateShortcut(“$path\PowerShell.lnk”)
# $link is an object and has the properties and methods
$link | Get-Member
```

```
TypeName: System.__ComObject#{f935dc23-1cf0-11d0-adb9-00c04fd58a0b}

Name              MemberType   Definition
----              ----------   ----------
Load              Method       void Load (string)
Save              Method       void Save ()
Arguments         Property     string Arguments () {get} {set}
Description       Property     string Description () {get} {set}
FullName          Property     string FullName () {get}
Hotkey            Property     string Hotkey () {get} {set}
IconLocation      Property     string IconLocation () {get} {set}
RelativePath      Property      {get} {set}
TargetPath        Property     string TargetPath () {get} {set}
WindowStyle       Property     int WindowStyle () {get} {set}
WorkingDirectory Property     string WorkingDirectory () {get} {set}

# We can populate some of the properties
$link.TargetPath = 'powershell.exe'
$link.Description = 'Launch Windows PowerShell console'
$link.WorkingDirectory = $profile
$link.IconLocation = 'powershell.exe'
# And save the changes using Save() method
$link.Save()
```

Summary

Everything in PowerShell is represented by objects that have exactly two aspects: properties and methods, which both form the members of the object. While properties store data, methods are executable commands.

Objects are the result of all PowerShell commands and are not converted to readable text until you output the objects to the console. However, if you save a command's result in a variable, you will get a handle on the original objects and can evaluate their properties or call for their commands. If you would like to see all of an object's properties, then you can pass the object to Format-List and type an asterisk after it. This allows all—not only the most important—properties to be output as text.

The Get-Member cmdlet retrieves even more data, enabling you to output detailed information on the properties and methods of any object.

All the objects that you will work with in PowerShell originate from .NET framework, which PowerShell is layered. Aside from the objects that PowerShell commands provide to you as results, you can also invoke objects directly from the .NET framework and gain access to a powerful arsenal of new commands. Along with the dynamic methods furnished by objects, there are also static methods, which are provided directly by the class from which objects are also derived.

If you cannot perform a task with the cmdlets, regular console commands, or methods of the .NET framework, you can resort to the unmanaged world outside the .NET framework. You can directly access the low-level API functions, the foundation of the .NET framework, or use COM components.