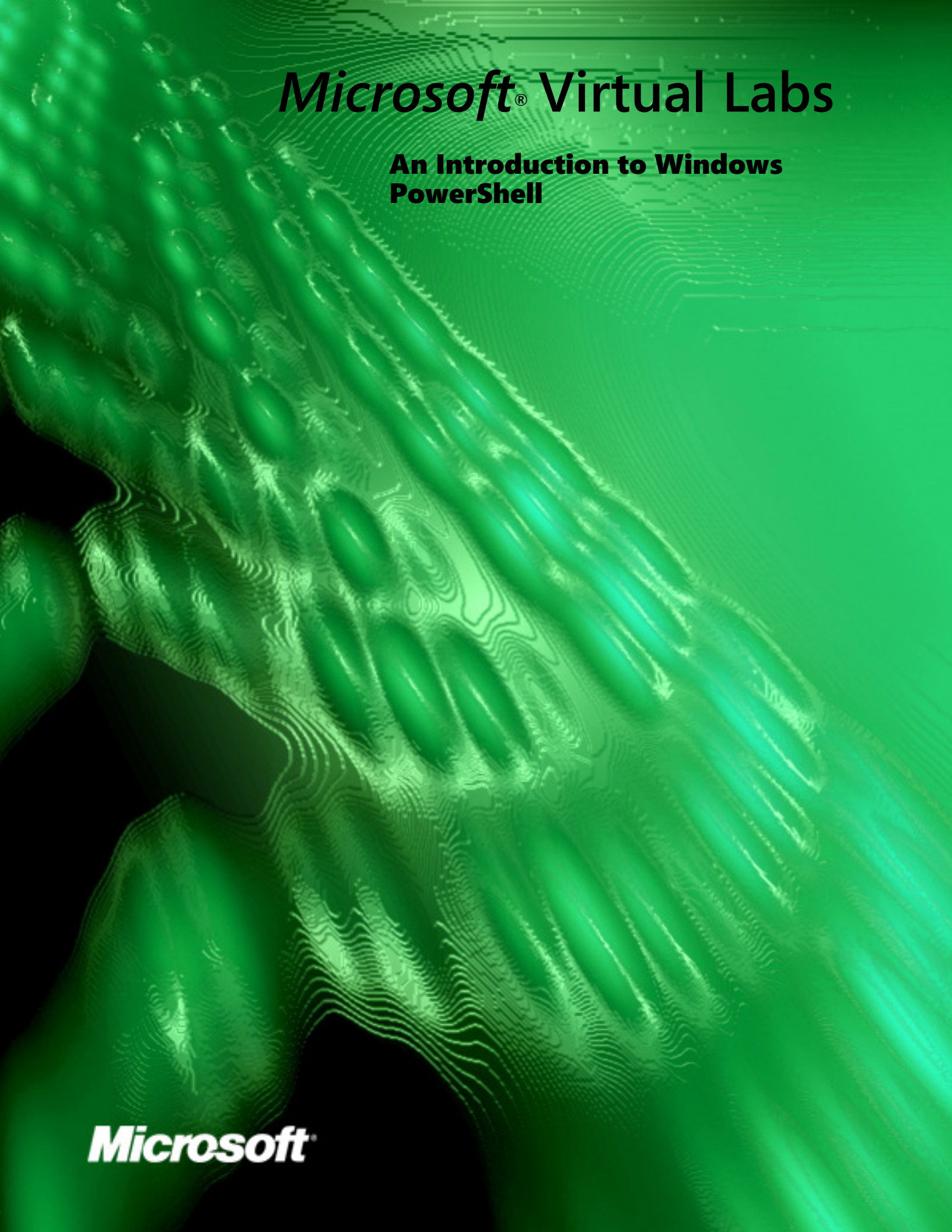
Microsoft® Virtual Labs
An Introduction to Windows PowerShell
Microsoft

Table of Contents

An Introduction to Windows PowerShell

Objectives

After completing this lab, you will be better able to:

- Use Windows PowerShell to carry out system administration tasks
- Take advantage of the capabilities built into the Windows PowerShell console

Scenario

It's every system administrator's worst nightmare: one of your file servers has crashed – hard – and no backup exists. A consulting firm was able to come in and retrieve files off one of the hard drives (fortunately, the drive containing user data). However, the best the consultants could do was to recover the files; the directory structure was essentially lost. Consequently, they simply grabbed all the files they could retrieve and saved them to a single folder: C:\Restored.

In this lab, your job is to analyze the collection of retrieved files, getting rid of files (such as temporary files) that are of no use, and arranging the files in some sort of temporary file structure. To perform this task you have chosen to use Windows PowerShell, and for reasons that should become apparent by the end of the lab.

Note: *Admittedly the steps outlined in this lab probably don't represent the preferred/recommended method for dealing with a situation such as the one described above. But that's OK: remember, the intent here is to introduce you to Windows PowerShell, not to tell you how to recover from a file server crash. To tell you the truth, the Scripting Guys are more likely to know how to cause a file sever crash than they are to know how to recover from such a crash.*

Tips for Completing the Lab

The lab requires you to do a considerable amount of typing. If you don't want to do a considerable amount of typing that's fine: all the commands used in the lab can be found in the file Virtual_Lab_Shortcuts.txt (this file is available from the desktop). If you prefer, you can simply copy commands from Virtual_Lab_Shortcuts.txt and paste them into Windows PowerShell.

Note, too that this lab contains a number of bonus exercises. If you are concerned about finishing on time we recommend that you skip the bonus exercises; after all, you can always go back and try them later, either as part of this lab or when running your own copy of Windows PowerShell.

Prerequisites

None. Basic knowledge of Windows system administration is useful, but not required.

Estimated Time to Complete This Lab

60 Minutes

Computer used in this Lab

 **Paris**

The password for the Administrator account on in this lab is: **pass@word1**.

Exercise 1
Listing Files

Scenario

Knowledge is power, which means the first thing we should do is take a look at the folder C:\Restored and get a better idea of the task ahead of us; we can't really decide what course of action to take until we know exactly what is – and isn't – in the folder C:\Restored. In the following series of exercises, we'll show you several different methods of exploring the file system using Windows PowerShell.

Tasks	Detailed Steps
1. Listing All the Files in a Folder	***Note:*** *As we noted, before we can do anything else we need to understand exactly what files we are working with, as well as how many files we are working with. With that in mind, our first step is to use the* ***Get-ChildItem*** *Cmdlet to list all the files in the folder C:\Restored. Because we're working with the file system, the Get-ChildItem Cmdlet will function very much like the* ***Dir*** *command. However, Get-ChildItem – which is designed to return information about the items found in a specified location – can also be used with other Windows PowerShell drives. For example, when working with the registry, Get-ChildItem can be used to return all the subkeys in a specified registry key.* **a.** If you have not done so, double-click the **desktop icon labeled Windows PowerShell**; this will star the application and open a Windows PowerShell command window. In the command window, type the following and then press **ENTER**: `get-childitem c:\restored` ***Note:*** *A list of all the files found in the folder C:\Restored should be displayed onscreen.*
2. Excluding Items from the Output	***Note:*** *If you browsed through the list of files found in C:\Restored you probably noticed that a good many of those files are temporary files, files with a .tmp file extension. For the purposes of this lab, we're going to say that all of these .tmp files are of no interest to us; all they do is take up space and get in the way of us being able to zero in on the files that are of interest. Wouldn't it be nice if we could tell the Get-ChildItem Cmdlet, "Listen, could you show us all the files except those with a .tmp file extension?"* ***Note:*** *Interestingly enough, we can do just that. Several Windows PowerShell Cmdlets – including Get-ChildItem – feature* ***–include*** *and* ***–exclude*** *parameters, which enable you to zero in on a more finely-targeted collection of items. The –include parameter, as the name implies, allows you to specify only those items to be included in a collection; by contrast, the aptly-named –exclude parameter enables you to specify items that should not be included in a collection. You say you'd prefer not to see the .tmp files in the Get-ChildItem output? No problem; in that case,* **a.** Just type the following command into the command windows and then press **ENTER**: `get-childitem c:\restored -exclude *.tmp` ***Note:*** *As you can see, all we did was tack on the –exclude parameter followed by the items we want to exclude; in this example, that's all the files that have a .tmp file extension. (Notice that we used the asterisk as a wildcard character.) Can we exclude*

Tasks	Detailed Steps
	more than one file type from the output? **b.** Of course we can; this command tells Get-ChildItem to ignore both .tmp and .temp files: `get-childitem c:\restored -exclude *.tmp,*.temp` ***Note:** The –include parameter works in a similar fashion, although it does have one minor little quirk you need to be aware of: when specifying the folder path your must do one of two things:* • *Include the wildcard characters *.* at the end of the path. For example, to work with the folder C:\Restored your file path would need to look like this:* *C:\Restored*.** • *Add the **–recurse** parameter to the command. The –recurse parameter causes Get-ChildItem to return not just the files in the target folder (e.g., C:\Restored), but also files found in any and all subfolders of C:\Restored. In this lab C:\Restored doesn't have any subfolders, so using –recurse is equivalent to using the wildcards *.*. If we had subfolders, however, these two approaches would return different collections: a command using –recurse would return files found in all the subfolders of C:\Restored, while *.* would not.* **c.** In other words, suppose you want to look at only the .doc and.xls files in the folder C:\Restored. Here's how you do that: `get-childitem c:\restored\*.* -include *.doc,*.xls`
3. Excluding Specific Properties from the Output	***Note:** You've probably noticed that, by default, running Get-ChildItem against a folder returns information similar to this:* `Mode LastWriteTime Length Name` `---- ------------- ------ ----` `-a--- 7/20/1999 3:27 PM 99005 api.chm` `-a--- 8/4/2004 5:00 AM 79996 apps.chm` `-a--- 8/4/2004 5:00 AM 299152 apps_sp.chm` `-a--- 8/4/2004 5:00 AM 2698341 article.bak` `-a--- 8/4/2004 5:00 AM 63244 Ascent.jpg` `-a--- 8/4/2004 5:00 AM 22219 atm.chm` *This is the default display mode for files and folders. However, this is not the only way to display file and folder information in Windows PowerShell. In some cases, you might want to show more property values than **Mode**, **LastWriteTime**, **Length**, and **Name**; in other cases, you might want to show fewer property values than the four shown by default. Either way, you can specify which property values you want to see (and which ones you don't want to see) by using the **Select-Object** Cmdlet,* *In the following command we use Get-ChildItem to return a collection of all the files (except the .tmp files) found in the folder C:\Restored. However, we don't immediately display those files. Instead, we "pipe" the collection to the Select-Object Cmdlet (the | character represents the pipeline). That simply means we hand the collection over to Select-Object and ask it to weed out all the properties except the ones we specifically ask for.* **a.** In this example, that's the **Name** and **Length** properties:

Tasks	Detailed Steps
	`get-childitem c:\restored -exclude *.tmp | select-object name,length` To see how this works, type the preceding command into the command window and then press **ENTER**. ***Note:*** *You should get back information similar to the following:* `Name Length` `---- ------` `api.chm 99005` `apps.chm 79996` `apps_sp.chm 299152` `article.bak 2698341` `Ascent.jpg 63244` `atm.chm 22219` `audit.chm 32750`
4. Changing Locations within Windows PowerShell	***Note:*** *To tell you the truth we hadn't really thought about it, but, yes, we can see where it could become a bit tedious to have to type the path C:\Restored each time you call the Get-ChildItem Cmdlet. (Especially when you consider the number of times we'll be calling Get-ChildItem in this lab.) Is there something we can do about it? Now that you mention it, there are at least two things we can do to help.* *For one, you can actually call Get-ChildItem without specifying a path; in that case Get-ChildItem simply returns items found in your current location within Windows PowerShell (a location you can identify, by default, simply by checking the Windows PowerShell prompt).* **a.** If we don't want to type the path **C:\Restored** then all we have to do is move to the **C:\Restored folder**, a move we can make using the **Set-Location** Cmdlet: `set-location c:\restored` **b.** Alternatively, you can use the alias (more on aliases in just a section) to switch to the **C:\Restored folder**: `cd c:\restored` **c.** To get back to your home directory type the following, with the tilde (~) being shorthand for your home folder: `cd ~` ***Note:*** *Another thing you can do is map a Windows PowerShell drive to the folder C:\Restored. (Note that this isn't a true mapped drive; the mapping applies only when you are working with Windows PowerShell.)* **d.** To map a Windows PowerShell drive to the folder **C:\Restored** use the **New-PSDrive** Cmdlet, like so: `new-psdrive x -psprovider filesystem -root c:\restored` ***Note:*** *As you can see, we simply call New-PSDrive and provide the Cmdlet with three parameters:* • ***x****, the drive letter to be assigned to the new Windows PowerShell drive.* • ***-psprovider filesystem****, which tells Windows PowerShell that the new drive maps to a location in the file system. (**Note**. Are there other locations you can map to? You bet; the command **New-PSDrive -name Y -psprovider Registry -***

Tasks	Detailed Steps
	root HKCU:\Software\Microsoft\Windows\CurrentVersion *maps drive Y to the registry key HKEY_CURRENT_USER\ Software\Microsoft\Windows\CurrentVersion.)* • ***-root c:\restored****, which simply assigns C:\Restored as the root folder for the new drive.* **e.** If you type in the preceding command and then press **ENTER** you'll no longer have to type the path **C:\Restored**; instead, you can access all the items in the folder by using this command: `get-childitem x:`
5. Aliases and Other Typing Tricks	***Note:*** *Already tired of typing a lengthy Cmdlet like Get-ChildItem over and over again? In that case you might check to see if an alias exists for Get-ChildItem. An alias is simply an alternate (and usually abbreviated) method for calling a Windows PowerShell Cmdlet. You can create your own aliases or you can use the aliases built into Windows PowerShell.* **a.** To view all the aliases that ship with Windows PowerShell type the following command into the command window and then press **ENTER**: `get-alias | sort-object definition` ***Note:*** *Included among the information displayed on the screen should be the following:* `CommandType Name Definition` `----------- ---- ----------` `Alias ac Add-Content` `Alias asnp Add-PSSnapin` `Alias clc Clear-Content` `Alias cls Clear-Host` `Alias clear Clear-Host` **b.** As you can see, there are actually three aliases predefined for Get-ChildItem. Tired of typing **get-childitem c:\restored**? Then type this instead: `ls c:\restored` ***Note:*** *Here's another little trick that can save wear-and-tear on your typing fingers. Windows PowerShell includes tab completion, a feature that allows you to type part of a command or path and then press the TAB key and have Windows PowerShell type the rest of the information for you.* **c.** What does *that* mean? Well, suppose the Restored folder is the only folder in C:\ whose name starts with the letter *r*. In that case, you could use Get-ChildItem to return the files and folders in C:\Restored by doing this: • Typing **ls c:\r** • Pressing the **TAB** key. • Pressing **ENTER**. ***Note:*** *Of course, sometimes you might not type enough information for Windows PowerShell to be able to distinguish between two items. For example, if you type* ***get-ch*** *and press TAB Windows PowerShell first suggests* ***.\Get-ChildItem*** *as the command it thinks you want. If that's not the command you want, press TAB a second*

Tasks	Detailed Steps
	*time and Windows PowerShell will then suggest **Get-ChildItem**. If there are even more commands that start with get-ch just keep pressing TAB and Windows PowerShell will continue offering up new alternatives.* *Note: **Quick tip**. Want to redo the command you just ran? Then press the UP arrow on the keyboard. And you can run any command in your session history by pressing F7, using the arrow keys to locate the command, and then pressing ENTER.* ***Note:** One last trick. When using parameters you need to type only as much of the parameter name as Windows PowerShell needs to uniquely identify the parameter. For example, Get-ChildItem includes three parameters whose name starts with the letter e:* • *-ErrorAction* • *-ErrorVariable* • *-Exclude* ***Note:** That means you can specify the –exclude parameter merely by typing **–ex**; that's enough to distinguish –exclude from –ErrorAction and –ErrorVariable. In addition, you can also use wildcards when specifying values for most Cmdlets and Cmdlet parameters. Remember this command?:* `get-childitem c:\restored -exclude *.tmp	select-object name,length` ***Note:** Using a combination of aliases (ls and select), "abbreviated" parameter names (-ex), and wildcard characters (n* and le*) we could type this command using no more characters than this:* `ls c:\restored -ex *.tmp	select n*,le*` *We won't use aliases or abbreviated parameters in the instructions provided for this lab. But you're welcome to use them on your own.*

Exercise 2
Sorting Data

Scenario

By default the Get-ChildItem Cmdlet sorts the returned collection of files by file name. What's wrong with that? Nothing; most of the time that's exactly how you want the collection sorted.

In this lab, however, one of our primary concerns is disk space: until the replacement file server arrives and is installed we anticipate being short on available space. In our situation it would be extremely useful if we could sort the returned collection by file size. That would do at least two things for us: make it easier for us to eyeball the amount of disk space currently in use, and help us identify files we might want to delete (or archive) should disk space start to run short. But how in the world can we sort our collection on file size?

Tasks	Detailed Steps
1. Sorting Data on a Different Property	***Note:*** *If you guessed that Windows PowerShell includes a Cmdlet designed to sort returned data, well, congratulations: you're absolutely right. (If you didn't guess that, that's OK; after all, who's going to know?) If we want to sort information in Windows PowerShell all we have to do is pipe that information to the* ***Sort-Object*** *Cmdlet and tell Sort-Object which property to sort by. For example, suppose we'd prefer to see our files sorted by file size (the* ***Length*** *property). Can we do that?* **a.** Try typing the following command and pressing **ENTER**, and let's find out: `get-childitem c:\restored | select-object name, length | sort-object length` ***Note:*** *With any luck at all you should get back the collection of files, only this time sorted by file size:* `Name Length` `---- ------` `Temp_File_1.tmp 26` `Temp_File_10.tmp 26` `Temp_File_11.tmp 26` `Temp_File_12.tmp 26` `Temp_File_13.tmp 26` ***Note:*** *Incidentally, you can sort by more than one property. If you look at the sample output above you'll see a number of files that have the same file size. In cases like that you'll often want to sort your data by file size and then by file name; in other words, files of the exact same size will, within their own little group, be sorted by name.* **b.** To get that output we simply need to specify the **Name** property as the second property to sort by. Type this command into the command window, press **ENTER**, and see what happens: `get-childitem c:\restored | select-object name, length | sort-object length,name`
2. Sorting Data in Descending Order	***Note:*** *As we noted, our primary concern is with excessively large files. We've managed to sort the files by file size, but – by default – the smallest files are shown first and the largest files are shown last. It would be far more convenient if we could flip that around and show the largest files first and the smallest files last.*

Tasks	Detailed Steps
	In other words, we'd like to sort in descending (Z to A, 9 to 0) order rather than ascending (A to Z, 0 to 9) order. Well, then why don't we just add the ***–descending*** *parameter to Sort-Object:* **a.** Type the following command into the command window and then press **ENTER**. `get-childitem c:\restored | select-object name, length | sort-object length -descending` ***Note:*** *You should get back something that looks like this:* `Name Length` `---- ------` `article.bak 2698341` `backup_c.bak 1440054` `backup_b.bak 1440054` `backup_a.bak 1440054` `backup_e.bak 1440054`
3. Returning *N* Number of Items	*We're definitely making progress now. On the other hand, the folder C:\Restored includes hundreds of files, only a handful of which are large enough to warrant our attention. Wouldn't it be nice if we could limit the returned data to, say, just the 10 largest files?* *Then why don't we do just that? Take a look at the following command, which – building off our previous exercise – returns the* ***Name*** *and* ***Length*** *of all the files in the folder C:\Restored, sorted, in descending order, by file size:* *get-childitem c:\restored | select-object name, length | sort-object length -descending | select-object -first 10* *You're right: we did tack on an additional pipeline, didn't we? After returning and sorting all the files, we then pipe the entire collection back to the* ***Select-Object*** *Cmdlet:* *select-object -first 10* *Why? Well, notice the parameter we're using with Select-Object:* ***-first 10****. This parameters instructs Select-Object to select (and then display) only the first 10 files in the sorted list; in this case, that equates to the 10 largest files in the folder.* **a.** Type the following command into the command window and then press **ENTER**; `get-childitem c:\restored | select-object name, length | sort-object length -descending | select-object -first 10` ***Note:*** *you should get back something similar to this* *Name Length* *---- ------* *article.bak 2698341* *backup_c.bak 1440054* *backup_b.bak 1440054* *backup_a.bak 1440054* *backup_e.bak 1440054* *backup_d.bak 1440054* *backup_f.bak 1440054*

Tasks	Detailed Steps
	rktools.bak 1237777 *ntart.bak 1227075* *SET4.tmp 1086058* ***Note:** Cool, huh? Of course, you aren't limited to returning the top 10 items: you can specify any number you wish (for example, **-first 37** returns the top 37 items). Alternatively, use the **–last** parameter to specify items at the bottom of the list (in this example, that would be files with the smallest file size).* **b.** Type this command into the command window, press **ENTER**, and see what you get back: `get-childitem c:\restored | select-object name, length | sort-object length -descending | select-object -last 25`
4. Viewing the Properties and Methods of an Object	***Note:** How did we know the name of the file size property in Windows PowerShell was Length? Well, in this case we cheated a bit; after all, the Length property appears in the default output when using Get-ChildItem. But what if Length didn't appear in the default output? (After all, back in Exercise 1C we hinted at the fact that the default output for Get-ChildItem does not include all the properties of a file.) What then?* *Relax; Windows PowerShell makes it very easy to retrieve information about all the properties and methods of an object. All you have to do is make a connection to the object and then pipe the object to the **Get-Member** Cmdlet. For example, this command returns a collection of items found in the folder C:\Restored, then pipes that collection object to Get-Member:* `get-childitem c:\restored | get-member` **a.** To see what properties and methods are available to you when working with files and folders other than the four we've already seen (Mode, LastWriteTime, Length, and Name) simply type the following preceding command into the command window and then press **ENTER**. `get-childitem c:\restored | get-member`

Exercise 3
Grouping Items and Calculating Statistics

Scenario

Having taken a preliminary look at the recovered files you've decided that there's no real rhyme or reason to the collection: these are simply a random series of files that the consultants were able to recover. Because of that, you're leaning towards simply grouping the files by file type, putting all the Excel spreadsheets in one folder, all the Word documents in another folder, etc. That's hardly the optimal way to sort and store files, but at least it provides a starting point for people who need to know, say, whether a particular PowerPoint presentation was recovered.

Of course, that only makes sense if there is a reasonable distribution of the file types; after all, if 99% of the files are Word documents and 1% are Excel spreadsheets then you haven't really gained much by grouping files by file type. With that in mind, you've decided to retrieve the collection and group it by file extension, the better to see how many files of each type can be found in the folder.

Tasks	Detailed Steps
1. Grouping Items by File Type	***Note:*** *Of course, it's one thing to say we'd like to group all the files by file extension; it's a whole 'nother thing to actually do this, however. Or is it?* *As it turns out, it's remarkably easy to group items based on a property value: that's what the* ***Group-Object*** *Cmdlet lives for.For example, consider the following command, which retrieves a collection of all the files found in the folder C:\Restored and then pipes that collection to Group-Object:* *get-childitem c:\restored | group-object extension* ***Note:*** *What will Group-Object give us back? This:* `Count Name                      Group` `----- ----                      -----` `   21 .chm                       {api.chm, apps.chm,...}` `    9 .bak                       {article.bak,...}` `    3 .jpg                       {Ascent.jpg, Power.jpg,...}` `    3 .gif                       {bullet.gif, exclam.gif,...}` `    9 .wav                       {chimes.wav, chord.wav,...}` `   63 .doc                       {Document_1.doc,...}` `   42 .xls                       {Example_Spreadsheet_1.xls,...` `    3 .mid                       {flourish.mid,...}` `   80 .log                       {KB835221.log,...}` `   53 .tmp                       {SET3.tmp, SET4.tmp,...}` `   13 .txt                       {Text_File_1.txt,...}` `    1 .hlp                       {wscript.hlp}` ***Note:*** *As you can see, there's nothing too terribly complicated about using Group-Object; in this example all we had to do was call the Cmdlet followed by the property we wanted to group on (****Extension****).* **a.** Try typing in the command and then pressing **ENTER**; you should get the same thing we did. `get-childitem c:\restored | group-object extension`

Tasks	Detailed Steps
	Note: *And after you do that, try this variation. In this command we group the items by file extension and then pass the grouped collection to the* ***Sort-Object*** *Cmdlet. In turn, Sort-Object sorts the collection on the* ***Count*** *property. (When you group objects, the grouped collection includes a property named Count which, as you can probably guess, tells you the number of items in each grouping.)* **b.** Type the following command into the command window, press **ENTER**, and see what happens: `get-childitem c:\restored | group-object extension | sort-object count` **c.** Here's an example of a fancier bit of grouping. We won't explain this command in any detail, but it groups all the files first by the year they were created and then by the month they were created. Type the following command into the command window, press **ENTER**, and you'll see what we're talking about: `get-childitem c:\restored | group-object {$_.CreationTime.Year},{$_.CreationTime.Month}`
2. Determining Overall File Size	***Note:*** *By now we should know that the folder C:\Restored contains a whole bunch of files; what we don't know is how many files, and – perhaps more important – how much disk space those files are taking up. If we were to sit down and count the number of files, then add the total amount of disk space being used, we'd come up with – well, to tell you the truth, we'd come up with nothing, because counting up all those files and adding up the file sizes would be too long and tedious a process.* *Of course, while we don't like long and tedious, Windows PowerShell doesn't seem to mind it. Therefore, why don't we ask Windows PowerShell to use its* ***Measure-Object*** *Cmdlet and calculate these statistics for us:* `get-childitem c:\restored | measure-object length -average -sum -maximum -minimum` *As you can see, we're – again – using Get-ChildItem to return a collection of all the files found in the folder C:\Restored. We then pipe that collection to the Measure-Object Cmdlet, which is requested to do two things:* • *Make its calculations based on the value of the* ***Length*** *property.* • *Return the average file size (****-average****); the total file sum (****-sum****); the size of the largest file (****-maximum****); and the size of the smallest file (****-minimum****).* **a.** If you type the following command into the command window and then press **ENTER** `get-childitem c:\restored | measure-object length -average -sum -maximum -minimum` ***Note:*** *you should get back information similar to the following:* *Count : 300* *Average : 100837.376666667* *Sum : 30251213* *Maximum : 2698341* *Minimum : 26* *Property : Length* *Of course, you can also use wildcards to focus in on a specific set of files. Remember*

Tasks	Detailed Steps
	those pesky .tmp files? This command returns the number and total size of all the .tmp files in C:\Restored: `get-childitem c:\restored*.tmp	measure-object length -sum` **b.** Type this command into the command window, press **ENTER**, and see what you get back.
3. Displaying a Single Property Value	***Note:*** *Any time you call the Measure-Object Cmdlet you get back all the statistical values, even those you didn't ask for and that Measure-Object didn't calculate for you.* **a.** For example, type the following command into the command window and then press **ENTER**: `get-childitem c:\restored	measure-object length -sum`  ***Note:*** *You should get back something similar to this, with empty placeholders for properties like Average, Minimum, and Maximum:*  `Count : 79` `Average :` `Sum : 3776752` `Maximum :` `Minimum :` `Property : length` ***Note:*** *That's OK, albeit a tad-bit confusing.* *But, hey, this is Windows PowerShell; you don't have to settle for "OK." If all you want to get back is a value representing the sum then that's all you should get back. Take a look at this command (and don't be put off by it; it's nowhere near as complicated as you might think):* `(get-childitem c:\restored | measure-object length -sum).sum` ***Note:*** *To explain how this command works it's important that you understand the purpose of parentheses in Windows PowerShell. As you doubtless recall from your junior high math classes, an arithmetical equation such as this is difficult to interpret:* `15 + 28 / 60` *After all, do you add 15 and 28 and then divide that answer by 60? Or, do you divide 28 by 60 and then add that answer to 15? Needless to say, depending on which method you choose you'll get very different answers.* *To help guard against misinterpretations, mathematicians use parentheses to indicate which operations should be performed first. In this revised equation, the parentheses tell us to first add 15 and 28, and then divide that answer by 60:* `(15 + 28) / 60` ***Note:*** *Parentheses serve a similar function in Windows PowerShell. Let's take another look at our command:* `(get-childitem c:\restored | measure-object length -sum).sum`

Tasks	Detailed Steps
	Notice the parentheses surrounding this command: `get-childitem c:\restored	measure-object length -sum` ***Note:*** *Because this command is surrounded by parentheses that means we want Windows PowerShell to execute this command before it does anything else. In other words, Windows PowerShell is going to retrieve a collection of all the files in the folder C:\Restored, then pipe that collection to the Measure-Object Cmdlet. In turn, Measure-Object is going to calculate the sum of the Length property.* *At this point in time we actually have a Windows PowerShell object (technically, a Microsoft.PowerShell.Commands.GenericMeasureInfo object). You can verify that for yourself by running the command and then piping the information to Get-Member:* *Typically any time you pipe something to Measure-Object the Cmdlet performs its statistical wizardry and then reports back the results. That's not what we want, however, and that's why we enclosed the command in parentheses. Because we're interested in only the* ***Sum*** *property we retrieve the object and then ask Windows PowerShell to echo back only the value of the Sum; that's why we tacked the* ***.sum*** *on the end.* *Make sense? Try typing the command into the command window and pressing ENTER; that should help. And then, for extra credit, use the preceding command as a template, and see if you can figure out how to return just the* ***Average*** *property.*

Exercise 4
Deleting Files

Scenario

Earlier in this lab we complained about the large number of temporary files – files of no interest to us – that "litter" the folder C:\Restored. Because of that, we showed you how to use wildcard filtering to hide these files from the information returned by the Get-ChildItem Cmdlet. That's fine, except for this: if these files truly aren't of any use to us (and they aren't) wouldn't it be better to just get rid of them altogether? Why keep them if we don't need or want them?

Hey, we wish *we* would have thought of that! With that in mind, in this exercise we'll delete all the .tmp files found in the folder C:\Restored. In addition, we've determined that all the files in the folder that are larger than 1 megabyte are backup files that can safely be deleted as well. Therefore, after we've deleted all the .tmp files we'll set our sights on deleting all the files bigger than 1 megabyte.

Tasks	Detailed Steps
1. Deleting All Files with a Specified File Extension	***Note:*** *Removing all the .tmp files in a folder is easy; all we have to do is call the* ***Remove-Item*** *Cmdlet followed by the list of files to be removed (in this case, we use wildcards to indicate that we want to remove all the files with a .tmp file extension):* **a.** Type the following command into the command window and then press **ENTER**. `remove-item c:\restored\*.tmp` **b.** To verify that all the .tmp files have been deleted from C:\Restored type *this* command into the command window and then press **ENTER**: `get-childitem c:\restored\*.tmp`
2. Deleting All Files Larger than a Specified Size	***Note:*** *In addition to deleting all the .tmp files, you've also determined that it's safe to delete all the files larger than 1 megabyte; as near as you can tell, those are primarily log files that, if needed, can be copied from another computer. So can we delete files greater than 1 megabyte using Windows PowerShell? Do you even need to ask?* ***Note:*** *Let's take a look at the command that deletes all the files larger than 1 megabyte from the folder C:\Restored; after giving you a chance to look at the command we'll explain how it works:* `get-childitem c:\restored | where-object {$_.length -gt 1048576 } | foreach-object {remove-item $_.fullname}` ***Note:*** *In this command we're introducing two new Cmdlets: Where-Object, which functions similar to a Where clause in a WMI query, and ForEach-Object, which enables us to create For Each loops. The way this works is that we use Get-ChildItem to return a collection of all the files in the folder C:\Restored. We then pipe that collection to Where-Object; the job of that Cmdlet is to weed out all files except those with a file size (Length) greater than 1 megabyte (1,048,576 bytes). Notice the syntax used with Where-Object: we call the Cmdlet followed by the filter criteria, which must be enclosed in curly braces.* *That's a good idea; let's spend a minute or two discussing the syntax of the filter itself. As you can see, we use the special variable $_ to represent the individual items in the collection. If you are familiar with VBScript then you probably are used to using code similar to this:*

Tasks	Detailed Steps

```
For Each objItem in colItems
    Wscript.Echo objItem.Name
Next
```

Note: In the preceding code snippet, objItem is simply a variable that – each time we run through the loop – takes on the characteristics of the current item in the loop. The $_ variable serves a similar role in Windows PowerShell. In our Where-Object filter we're simply asking Windows PowerShell to look at all the items in the collection and select only those individual items ($_) that have a length greater than 1,048,576 bytes. And yes, we also need to use –gt as the comparison operator. In Windows PowerShell you will typically use the following comparison operators:

Operator	***Definition***
-lt	*Less than*
-le	*Less than or equal to*
-gt	*Greater than*
-ge	*Greater than or equal to*
-eq	*Equal to*
-ne	*Not equal to*
-contains	*Determine elements in a group. This always returns Boolean $True or $False.*
-notcontains	*Determine excluded elements in a group. This always returns Boolean $True or $False.*
-like	*Like - uses wildcards for pattern matching*
-notlike	*Not Like - uses wildcards for pattern matching*
-match	*Match - uses regular expressions for pattern matching*
-notmatch	*Not Match - uses regular expressions for pattern matching*

***Note**. Interested in doing case-sensitive comparisons? Then simply insert the letter c after the hyphen in each operator. For example, the case-sensitive equals operator is this: -ceq.*

Our Where-Object filter is similar to a SQL/WQL query such as this:

```
Select * From colItems Where Length > 1048576
```

After grabbing all the files larger than 1 megabyte we pipe this trimmed-down collection to the ForEach-Object Cmdlet. Let's quickly review the syntax of this command:

```
foreach-object {remove-item $_.fullname}
```

***Note:** What we do here is call ForEach-Object, followed by the action we want to take on each item in the collection (this action must be enclosed in curly braces). We want to delete each item in the collection (that is, each file with a file size greater than 1 megabyte); therefore, we call the **Remove-Item** Cmdlet. In order to use Remove-Item to delete a file we need to pass the Cmdlet the complete path to that file; hence we use this syntax: **{Remove-Item $_.FullName}**, with $_ representing the individual items in*

Tasks	Detailed Steps
	*the collection, and **FullName** being the file property that contains the complete path.* **c.** And you're right: we *have* talked long enough about this one command, haven't we? With that in mind, why not just type in the following, press **ENTER**, and then see what happens?: `get-childitem c:\restored | where-object {$_.length -gt 1048576 } | foreach-object {remove-item $_.fullname}`
3. Using File Size Designators	***Note:** Specifying a condition like **$_.length –gt 1048576** is great, as long as you know that 1,048,576 bytes is equal to one megabyte. Of course, what if you were looking for files greater than 37 megabytes? In that case, you'd have to take 1,048,576 times 37, which would leave you with – look, wouldn't it just be easier if Windows PowerShell understood terms like kilobyte, megabyte, and gigabyte?* *Well, guess what: it does. If you want to work with kilobytes, megabytes, and gigabytes all you have to do is use the appropriate designator: KB, MB, or GB. For example, this command also deletes all the files with a file size greater than 1 megabyte; notice, though, that cryptic numbers like 1048576 don't appear anywhere in the command:* `get-childitem c:\restored \| where-object {$_.length -gt 1MB} | foreach-object {remove-item $_.fullname}` ***Note:** Here's another example, a command that lists all the files in C:\Restored that have a file size greater than 100 kilobyte (KB).* **a.** Type this command into the command window and then press **ENTER**: `get-childitem c:\restored \| where-object {$_.length -gt 100KB}` **b.** Or try this. Type *this* command into the command window and then press **ENTER**: `512KB + 512KB` ***Note:** And then take a careful look at the answer you get back. Cool, huh?*

Exercise 5
Creating Folders

Scenario

If everything went according to plan in Exercise 4, the only files left in the folder C:\Restored are files we not only want to keep, but files we want to reorganize as well. In our case, we've decided that "reorganizing files" simply means storing files of the same type (based on file extension) in the same folder. That means we're going to need to create a bunch of new folders, one for each file type (that is, one folder for .txt files, one folder for .doc files, etc.). In this exercise we'll show you a barebones command for creating a single folder, then demonstrate a fancier command that can automatically create one folder for each unique file extension found in C:\Restored.

Tasks	Detailed Steps
1. Create a New Folder	***Note:*** *If you need to create a new folder then you need the* ***New-Item*** *Cmdlet. To create a new folder all you have to do is call New-Item and specify two things:* *The path to the new folder.* *The* ***–type*** *parameter, with the type set to directory. Alternatively, you could create a new text file by setting the type to file.* *In other words, you need to use a command like this one, which creates the folder C:\Restored\Test:* `new-item c:\restored\test -type directory` **a.** To create the new folder, type the preceding command into the command window and then press **ENTER**.
2. Verifying the Existence of a Folder	***Note:*** *So did the new folder get created or not? One way to verify that is to use the* ***Test-Path*** *Cmdlet.* **a.** To determine whether or not the folder C:\Restored\Test exists type the following command into the command window and then press **ENTER**: `test-path c:\restored\test` ***Note:*** *Windows PowerShell returns True if the folder exists and False if the folder does not exist.* **b.** Here's a cool thing you can do with Test-Path. Want to know if there are any .xls files in the folder C:\Restored? Type this command into the command window, press **ENTER**, and see for yourself: `test-path c:\restored\*.xls`
3. Auto-Create New Folders Based on File Extension	***Note:*** *As you've seen, it's pretty easy to create a single file folder using Windows PowerShell. However, that's of only marginal use to us at the moment. After all, we need to create a whole bunch of file folders, one for each file type (based on file extension) found in C:\Restored. That leaves us with two problems: determining the unique file types in the C:\Restored, and creating a corresponding folder for each file type. That sounds like a lot of tedious and time-consuming work, doesn't it? It's probably too much to ask Windows PowerShell to take care of this for us, isn't it?* *Are you kidding; Windows PowerShell thrives on challenges such as this. To begin with, let's show you how easy it is to retrieve a collection of the unique file types found in C:\Restored.*

Tasks	Detailed Steps
	a. Type the following command into the command window and then press **ENTER**: `get-childitem c:\restored | select-object extension | sort-object extension -unique` ***Note:*** *You should get back something that looks like this:* *Extension* *---------* *.chm* *.doc* *.gif* *.hlp* *.jpg* *.log* *.mid* *.txt* *.wav* *.xls* ***Note:*** *Let's explain how this command works. As you can see, we've actually used the Windows PowerShell pipeline to string together three separate commands. In the first, we use Get-ChildItem to return a collection of all the files found in the folder C:\Restored; that should be old hat to you by now. We then pipe the collection to the Select-Object Cmdlet, which filters out all the properties (and property values) except for* ***Extension****. Again, nothing there that you haven't seen before.* *We then pipe the collection of file extensions to this command:* `sort-object extension -unique` *Here we're using Sort-Object to sort the collection by Extension (which happens to be the only property left in the collection). Notice, however, that we've tacked on the* ***–unique*** *parameter. The –unique parameter instructs Sort-Object to sort the collection and then extract only the unique items. What does that mean? Well, suppose our sorted collection looks like this, with a number of duplicate entries:* `.doc` `.doc` `.doc` `.txt` `.txt` `.xls` `.xls` `.xls` `.xls` *The –unique parameter weeds out the duplicate entries and leaves us with this:* `.doc` `.txt` `.xls`

Tasks	Detailed Steps
	OK; that gets us a list of unique file extensions. Now, how can we create a folder for each of these extensions? Here's one way: `get-childitem c:\restored	select-object extension	sort-object extension -unique` `| foreach-object {new-item ("c:\restored\restored_files" + $_.Extension) -type directory}` *Admittedly, this is a tiny bit more complicated than the commands we've used up to this point. But only a tiny bit more complicated. As you can see, we've combined four separate commands here; the first three return a collection of unique file extensions, using the same method we just discussed.* *That brings us to part 4:* `foreach-object {new-item ("c:\restored\restored_files" + $_.Extension) -type directory}` *In this command we're using the ForEach-Object Cmdlet to create a new file folder for each unique file extension. How do we do that? Well, for each object in the collection we call the New-Item Cmdlet, passing two parameters:* *The full path for the new item to be created.* *The **–type directory** parameter, which tells New-Item to create a new folder.* *The one tricky part occurs when we specify the path for each folder. To do that we use this command (enclosed in parentheses to ensure that Windows PowerShell completes this command before calling New-Item):* `("c:\restored\restored_files" + $_.Extension)` *In order to create the folder path we're simply concatenating two items: the string value **C:\Restored\Restored_Files** and the file extension of the current item in the collection (again, using the special variable $_). For example, suppose the first file extension in the collection is **.doc**. In that case the path name passed to New-Item will consist of **C:\Restored\Restored_Files** plus, **.doc**, yielding the following path:* `C:\Restored\Restored_Files.doc` ***Note:** We should probably point out that, in Windows PowerShell, the "dot" is part of the file extension: we have a **.doc** file rather than a **doc** file. This differs from WMI, where the dot is not part of the file extension.* **b.** To get a better idea how this works, type the following command in the command window and then press **ENTER**: `get-childitem c:\restored	select-object extension	sort-object extension -unique` `| foreach-object {new-item ("c:\restored\restored_files" + $_.Extension) -type directory}` ***Note:** Not only should the new folders be created, but you should see confirming information like this displayed in the command window:*  `Directory: Microsoft.PowerShell.Core\FileSystem::C:\restored`  `Mode LastWriteTime Length Name`

Tasks	Detailed Steps

```
----                 -------------         ------ ----
d----          9/28/2006   1:54 PM                restored_files
d----          9/28/2006   1:54 PM                restored_files.chm
d----          9/28/2006   1:54 PM                restored_files.doc
d----          9/28/2006   1:54 PM                restored_files.gif
d----          9/28/2006   1:54 PM                restored_files.hlp
d----          9/28/2006   1:54 PM                restored_files.jpg
d----          9/28/2006   1:54 PM                restored_files.log
d----          9/28/2006   1:54 PM                restored_files.mid
d----          9/28/2006   1:54 PM                restored_files.txt
d----          9/28/2006   1:54 PM                restored_files.wav
d----          9/28/2006   1:54 PM                restored_files.xls
```

Exercise 6
Moving Files to Different Folders

Scenario

With our folder structure in place that can mean only one thing: it's time to move each file to the appropriate folder. In this exercise, we'll use a single command to select all the files in the folder C:\Restored and move them to the appropriate folder, based on file extension. How hard is *that* going to be? Let's put it this way: nowhere near as hard as you might think.

Tasks	Detailed Steps
1. Moving Files Based on File Extension	***Note:*** *Believe it or not we weren't joking: we really can use a single command to move all the files in the folder C:\Restored to a new home. Let's take a look at the command, and then explain how it works:* `get-childitem c:\restored | where-object {$_.mode -notmatch "d"} \| foreach-object {$b = "c:\restored\restored_files" + $_.extension; move-item $_.fullname $b}` *As you can see, we do have a single command, albeit a single command with three different components. Part 1, which simply retrieves a collection of all the items found in the folder C:\Restored, is a command which, by now, is probably more familiar to you than your own children:* `get-childitem c:\restored` *This command works great, except for one thing. Previously any time we ran Get-ChildItem against C:\Restored we got back a collection of files; that's because that's all we had in the folder was a collection of files. Now, we're going to get back a collection of both files and folders. Is that a problem? In our case, it is. After all, when we start moving items around we want to move only the files; we want the folders to stay exactly where they are. That means we need to remove all the folders from the current collection.* *To do that, we pipe the collection to the Where-Object Cmdlet and run this command:* `where-object {$_.mode -notmatch "d"}` *What happens here? Well, Where-Object looks at each item in the collection (there's the $_ variable again) and checks the value of the* ***Mode*** *property. Among the attributes tracked by the Mode property is item type; that is, are we talking about a file or a folder? If you recall the output we got back when we created all our file folders in Exercise 5C, it looked something like this:* `Mode LastWriteTime Length Name` `---- ------------- ------ ----` `d---- 9/28/2006 1:54 PM restored_files` `d---- 9/28/2006 1:54 PM restored_files.chm` `d---- 9/28/2006 1:54 PM restored_files.doc` `d---- 9/28/2006 1:54 PM restored_files.gif`

Tasks	Detailed Steps
	d---- 9/28/2006 1:54 PM restored_files.hlp *See the lowercase d under Mode? That means we're dealing with a folder rather than a file. Files have no value set for this attribute:* *Mode LastWriteTime Length Name* *---- ------------- ------ ----* *-a--- 7/20/1999 3:27 PM 99005 api.chm* *-a--- 8/4/2004 5:00 AM 79996 apps.chm* *-a--- 8/4/2004 5:00 AM 299152 apps_sp.chm* *-a--- 8/4/2004 5:00 AM 2698341 article.bak* *To determine whether or not an object is a folder we use the **–notmatch** comparison operator, an operator that uses regular expressions to perform its comparisons. Take a look, again, at our Where-Object filter:* `{$_.mode -notmatch "d"}` *We're looking for items where the Mode property does not match the value d; in other words, we only want items that do not have the letter d anywhere within the Mode. That's how we eliminate folders from the collection; the Mode property for folders will include the letter d.* *After removing all the folders we then pipe the collection (which now consists entirely of files) to this command:* `foreach-object {$b = "c:\restored\restored_files" + $_.extension; move-item $_.fullname $b}` *Here we're using the ForEach-Object Cmdlet to loop through the collection of files. For each file in the collection we then do two things. First, we set a variable named $b to the value of the string **C:\Restored\Restored_Files** plus the file extension of the file (**$_.Extension**). If we're looking at a file named Test.doc, that means we're going to combine **C:\Restored\Restored_Files** plus **.doc**, making $b equal to this:* `C:\Restored\Restored_Files.doc` *And you're absolutely right: that is the name of the folder where we want to move all the .doc files.* *Which, as you might expect, is exactly what we do in the second half of this command. (Notice that these two "sub-commands" are separated using a semicolon.) In the second half of the command, we use the **Move-Item** Cmdlet to move the file to the appropriate folder:* `move-item $_.fullname $b` *Note that we need to pass Move-Item two parameters: the existing path to the file (**$_.FullName**) and the target folder for the file ($b). It's no more complicated than that: ForEach-Object will loop through the entire collection and move each file to the appropriate folder, based on the file extension.* **a.** But why take our word for it? Type the following command into the command window, press **ENTER**, and see for yourself: `get-childitem c:\restored \| where-object {$_.mode -notmatch "d"} | foreach-object {$b = "c:\restored\restored_files" + $_.extension; move-item`

Tasks	Detailed Steps
	`$_.fullname $b}`

Exercise 7
Viewing All the Files in a Folder and Its Subfolders

Scenario

So now, at long last, we've successfully moved all the files and reorganized the folder C:\Restored.

Or did we? Most likely Windows PowerShell took care of this without any problem. Still, it never hurts to make sure that everything truly *did* go off without a hitch. With that in mind, in this exercise we'll write a recursive command that will retrieve a list of *all* the files and folders found in C:\Restored, including all the subfolders and *their* files and subfolders.

And yes, we know: in VBScript writing a recursive command such as this is a daunting task, to say the least. Is this going to be equally difficult in Windows PowerShell? Let's find out.

Tasks	Detailed Steps
1. Listing All the Files in a Folder and Its Subfolders	***Note:*** *Earlier in the lab (Exercise 1B) we briefly mentioned Get-ChildItem's* ***–recurse*** *parameter, a parameter that enables you to retrieve – at least when dealing with the file system – all the files in a folder plus all the files in any subfolders of that folder. (Including, we might add, files found in sub-subfolders of those subfolders, and files found in any sub-sub-folders of – well, you get the idea.) We didn't use the –recurse parameter for one simple reason: our target folder (C:\Restored) didn't actually have any subfolders.* *Of course, that's no longer the case; now the folder C:\Restored contains all sorts of subfolders. To list all of these subfolders (and their contents) simply add the –recurse parameter to your Get-ChildItem call:* **a.** To view the results, type the following command into the command window and then press **ENTER**. `get-childitem c:\restored -recurse` ***Note:*** *Good question: what if you wanted to see only the folders and didn't care about the files? Although we won't discuss this in any detail, the following command should do the trick. With this command, you use the Get-ChildItem Cmdlet and the –recurse parameter to return a collection of all the items found in the folder C:\Restored. That information is then piped to the* ***Where-Object*** *Cmdlet, which picks out only those items where the* ***Mode*** *property includes the letter d (which means that the item is a directory).* **b.** To view the list of folders and subfolders only, type the following command into the command window and then press **ENTER**: `get-childitem c:\restored -recurse | where-object {$_.mode -match "d"}`

Exercise 8
Saving Data to a Text File

Scenario

Having successfully reorganized our file system it might not be a bad idea to save a copy of the new folder structure (and files) to a text file. In this exercise we'll retrieve file and folder information from the folder C:\Restored and then save the resulting data to a text file

Tasks	Detailed Steps
1. Outputting Data to a Text File	*Windows PowerShell provides a number of ways to write data to a text file, including the Cmdlets **Add-Content** and **Set-Content**. For today, however, we're going to take the easiest possible route and simply use the redirection character **>** to redirect our output to a text file. To retrieve a list of all the files and folders in the folder C:\Restored (and its subfolders),* **a.** Type the following command into the command window and then press **ENTER**: `get-childitem c:\restored -recurse > c:\restored\output.txt` ***Note:** Note that you won't see anything appear onscreen; that's because output is being redirected to the file C:\Restored\Output.txt.* **b.** However, you can verify that the output was saved by typing the following command into the command window and pressing **ENTER**: `get-content c:\restored\output.txt` ***Note:** As you can see, this command simply used the **Get-Content** Cmdlet to read the text file Output.txt and then display the contents onscreen.* ***Note**. Yes, you can append data to a file by using the **>>** redirection characters.* *Alternatively, we could have piped the results of Get-ChildItem to the **Out-File** Cmdlet and then used Out-File to save the data to C:\Restored\Test.txt:* `get-childitem c:\restored -recurse | out-file c:\restored\test.txt`
2. Outputting Data to a Text File and Displaying it On Screen	***Note:** If there's a problem with the preceding command it's this: the output is saved to the file Output.txt, but we don't ever see anything onscreen; in fact, there is no visual indication that the command is even doing anything. One simple way to solve that problem is to use the **Tee-Object** Cmdlet, which can simultaneously output data to two places, such as a text file and the screen.* **a.** To see what that means, type the following command into the command window and then press **ENTER**: `get-childitem c:\restored -recurse | tee-object c:\restored\output2.txt` ***Note:** You should see a list of all the folders (and the files contained within those folders) appear onscreen. And, if you look in the folder C:\Restored you should see that the exact same data was saved to the file Output2.txt.*

Exercise 9
Working With Properties and Methods

Scenario

We've saved what might be the best for last. Partly to show off the capabilities of Windows PowerShell and partly to help us identify the files that were restored from the crashed file server, we've decided to change the date modified property (**LastWriteTime**) of each file to the same date and time. No need to ask if it's possible to do that using Windows PowerShell; you already know the answer to that.

In this exercise, you'll use Windows PowerShell to modify the LastWriteTime property for each file in the folder C:\Restored.

Tasks	Detailed Steps
1. Modifying Property Values	**a.** Let's get right down to business. To modify the LastWriteTime property for all the files in the folder C:\Restored (including all the files in any subfolders of C:\Restored) type the following command into the command window and then press **ENTER**: `get-childitem c:\restored -recurse | foreach-object {$b=get-date; $_.lastwritetime = $b}` ***Note:*** *This is another multi-part command, with part 1 simply returning a collection of all the files and folders found in C:\Restored. Notice that, this time around, we don't bother using Where-Object to filter out folders. Why not? That's easy: we want to modify the LastWriteTime property of all the folders as well as all the files.* *After retrieving the collection we pipe the data to the ForEach-Object Cmdlet. Here's what ForEach-Object does to each object on the collection:* `{$b=get-date; $_.lastwritetime = $b}` *It's actually pretty simple. To begin with, we use the* ***Get-Date*** *Cmdlet to retrieve the current date and time and store it in the variable $b. And, yes, because we do this each time we could end up with slight differences in the date (or, at least, the exact time) modified property. However, we decided that was a small price to pay for keeping the command as simple as possible.* *After storing the current date and time in $b we then use this command to assign that date-time value to the LastWriteTime property for the current object in the pipeline:* `$_.lastwritetime = $b` *Believe it or not, that's all we have to do. In VBScript it's impossible to change the date modified property for a file or folder, let alone for all the files and folders in a target folder. And yet, with Windows PowerShell, all it takes is one reasonably-simple command.*