# Learn POWERSHELL TOOLMAKING IN A MONTH OF LUNCHES

MONDAY
TUESDAY
WEDNESDAY
THURSDAY
1 Powershell scripting overview ✓needed this
2 Simple scripts and language ✓interesting
3 Building an inventory tool ✓sounds good
7 Writing help and error handling
8 Debugging techniques ✓never guessed
9 Custom format views and script modules
10 Adding data base ✓easy peasy
11 Creating a new tool
14 Advanced toolmaking
15 Creating a custom type extension
16 Creating Powershell workflows
17 Troubleshooting pipeline input
18 Using object hierachies for complex output
22 Utilizing the .NET framework
23 Creating a GUI tool Part 1
24 Creating a GUI tool Part 2
25 Creating a GUI tool Part 3
29
30 Never the end
31 Final Exam

SAMPLE CHAPTER

DON JONES and JEFFERY D. HICKS

MANNING

# *Learn PowerShell Toolmaking in a Month of Lunches*

by Don Jones
and Jeffery Hicks

**Chapter 13**

Copyright 2013 Manning Publications

# *brief contents*

# 13 Script and manifest modules

We've been building `Get-SystemInfo` for several chapters now, and we've been testing it by inserting a line, at the end of our script, that runs the function. It's time to move away from that and into something that's a bit more formal, packaged distributable for our command. We also need to find a way to get our custom view XML file to load into memory automatically when someone wants to use our tool. In this chapter, we'll accomplish both.

## 13.1 Introducing modules

Introduced in PowerShell v2, modules are the shell's preferred means of extension (over the original PSSnapin extension technology). Modules can, in many cases, be file copied rather than requiring packagers or installers, which makes modules easy to distribute. Best of all—from our perspective—modules can be written in script, meaning you don't need to be a C# developer to create one.

When it comes to modules, much of PowerShell's capability relies on relatively low-tech techniques. Modules must follow a specific naming and location convention in order for PowerShell to "see" them. This can really throw people for a loop in the beginning—it's tough to comprehend that PowerShell can get sensitive over things like folder names and filenames. But that's how it is.

### 13.1.1 Module location

In order for PowerShell to fully utilize them, modules must live in a specific location. There can actually be more then one location; the `PSModulePath` environment variable defines those permitted locations. Here are the default contents of the variable:

```
PS C:\> get-content env:\psmodulepath
C:\Users\donjones\Documents\WindowsPowerShell\Modules;C:\Windows\system32\Win
     dowsPowerShell\v1.0\Modules\
```

You can modify this environment variable—using either Windows or a Group Policy object (GPO)—to contain additional paths. Some third-party PowerShell products might also modify this variable. The variable's contents must be a semicolon-separated list of paths where modules may be stored. For this chapter, we'll start with the first default path, which is in C:\Users\<username>\Documents\WindowsPowerShell\Modules. This path does not exist by default: You'll need to create it in order to begin using it.

> **CAUTION** In Windows Explorer, when you click the Documents library, you're actually accessing two folders: Public Documents and My Documents (or just Documents). The module path in `PSModulePath` refers only to the My Documents location. So if you're using Windows Explorer to create the folders in this path, be sure that you expand the Documents library and explicitly select My Documents or Documents.

We've created a handy command to create the necessary path:

```
PS C:\> New-Item -type directory -path (((get-content env:\psmodulepath)
  ➥ -split ';')[0])

    Directory: C:\Users\donjones\Documents\WindowsPowerShell

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----          5/6/2012   8:36 PM            Modules
```

Note that this path is user specific; if you want to put your modules into a shared location that's accessible by multiple users, then it's fine to add that path to `PSModulePath` for those users. Doing so with a GPO would be easiest, and it's fine to put UNC paths into `PSModulePath` rather than having to map a network drive.

### 13.1.2 Module name

Module names should consist of letters, numbers, and underscores, although Microsoft-provided modules tend to be named only with letters. Don't use module names that contain spaces (it isn't technically illegal, but it makes them a bit harder to work with).

Once you've come up with a good name for your module (we're going to use `MOLTools`), you need to create a folder for the module. In many ways, the folder you create is the module: If you distribute this to other users, for example, it's the entire folder that you will distribute. The folder must be created in one of the paths listed in `PSModulePath`; if you put the module folder elsewhere, then it won't participate in numerous PowerShell features (like module autodiscovery, autoloading, updatable help, and so on).

We'll change to the allowed module path and create a folder for `MOLTools`:

```
PS C:\> cd .\users\donjones\Documents\WindowsPowerShell\Modules

PS C:\users\donjones\Documents\WindowsPowerShell\Modules> mkdir
cmdlet mkdir at command pipeline position 1
Supply values for the following parameters:
Path[0]: MOLTools
Path[1]:

    Directory: C:\users\donjones\Documents\WindowsPowerShell\Modules

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----          5/6/2012   8:41 PM            MOLTools
```

We chose the name `MOLTools` after some serious thought. Keep in mind that PowerShell's command-naming convention allows for a prefix on the noun portion of command names. This prefix is designed to keep command names from overlapping. So, our `Get-SystemInfo` command should be named something like `Get-MOLSystemInfo` instead. The `MOL` stands for "Month of Lunches," and it's a noun prefix we feel is unlikely to be used by many others. That makes it private to us (although there's no way to enforce our ownership of it). Using `MOL` as our prefix will help ensure that our command can peacefully coexist with any `Get-SystemInfo` commands that someone else dreams up.

Having chosen `MOL` as our noun prefix, it makes sense to also include it in our module name. That way, the module name itself provides a clue as to the noun prefix used by the commands within the module.

**TRY IT NOW** Make sure you can create a MOLTools module folder as you follow along. Also, consider the prefix that you might use for your organization's commands and modules.

### 13.1.3 Module contents

With our module folder created, we can begin adding contents to it. We want to be able to load this module by running `Import-Module MOLTools` or by attempting to run one of the commands within the module (`Get-SystemInfo`, or `Get-MOLSystemInfo` if we rename it). In order for that to work, we need to understand a bit about how PowerShell loads modules.

First, if a module is located in a nonstandard path (that is, a path not listed in `PSModulePath`), we'll always have to manually load the module. Suppose we stored the module folder in C:\MyStuff. We'd need to run `Import-Module C:\MyStuff\MOLTools` in order to load the module, and PowerShell wouldn't be able to automatically load it for us.

That's why it's better to go with one of the supported module paths or to add a new supported path to the `PSModulePath` environment variable. That way, we can simply run `Import-Module MOLTools`, or just run one of the module's commands, to load the module.

When you run `Import-Module`, or when PowerShell attempts to automatically load a module for you, the shell looks in your module folder for one of these items, and it looks in this specific order:

1 A module manifest, which in our case would be MOLTools.psd1. Note that the filename must match the name of the module's folder, MOLTools.
2 A binary module, which in our example would be MOLTools.dll, if we were using a compiled binary, which we aren't. Again, the filename must be the complete module name plus the filename extension.
3 A script module, which for us would be MOLTools.psm1. Once again, you see that the filename must be the complete module name, exactly as the module's folder is named, plus the .psm1 filename extension.

This is the bit that really throws people. We see students put something like Test.psm1 into the \Modules\MOLTools folder, and that simply won't work. Most of PowerShell's magic is based upon the module folder being in one of the supported paths and on the module contents having the same name as that folder.

**CAUTION** Avoid putting modules into the other predefined path, which is under C:\Windows\System32—that location is reserved for Microsoft's use.

## 13.2 Creating a script module

Listing 13.1 shows our current script file, which we're still calling Test.ps1. Notice that we've renamed our command to `Get-MOLSystemInfo` (highlighted in boldface), and we've removed the final line of the script that was being used to run the function. We're saving this as C:\Users\donjones\WindowsPowerShell\Modules\MOLTools\MOLTools.psm1—in other words, making it into a script module.

**Listing 13.1 MOLTools.psm1**

```
function Get-MOLSystemInfo {
<#
.SYNOPSIS
Retrieves key system version and model information
from one to ten computers.
.DESCRIPTION
Get-SystemInfo uses Windows Management Instrumentation
(WMI) to retrieve information from one or more computers.
Specify computers by name or by IP address.
.PARAMETER ComputerName
One or more computer names or IP addresses, up to a maximum
of 10.
.PARAMETER LogErrors
Specify this switch to create a text log file of computers
that could not be queried.
.PARAMETER ErrorLog
When used with -LogErrors, specifies the file path and name
to which failed computer names will be written. Defaults to
C:\Retry.txt.
.EXAMPLE
 Get-Content names.txt | Get-MOLSystemInfo
.EXAMPLE
 Get-MOLSystemInfo -ComputerName SERVER1,SERVER2
#>
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True,
                   ValueFromPipeline=$True,
                   HelpMessage="Computer name or IP address")]
        [ValidateCount(1,10)]
        [Alias('hostname')]
        [string[]]$ComputerName,

        [string]$ErrorLog = 'c:\retry.txt',

        [switch]$LogErrors
    )
    BEGIN {
        Write-Verbose "Error log will be $ErrorLog"
    }
    PROCESS {
        Write-Verbose "Beginning PROCESS block"
        foreach ($computer in $computername) {
            Write-Verbose "Querying $computer"
            Try {
                $everything_ok = $true
                $os = Get-WmiObject -class Win32_OperatingSystem `
                                    -computerName $computer `
                                    -erroraction Stop
            } Catch {
                $everything_ok = $false
                Write-Warning "$computer failed"
                if ($LogErrors) {
                    $computer | Out-File $ErrorLog -Append
                    Write-Warning "Logged to $ErrorLog"
                }
            }

            if ($everything_ok) {
                $comp = Get-WmiObject -class Win32_ComputerSystem `
                                      -computerName $computer
                $bios = Get-WmiObject -class Win32_BIOS `
                                      -computerName $computer
                $props = @{'ComputerName'=$computer;
                           'OSVersion'=$os.version;
                           'SPVersion'=$os.servicepackmajorversion;
                           'BIOSSerial'=$bios.serialnumber;
                           'Manufacturer'=$comp.manufacturer;
                           'Model'=$comp.model}
                Write-Verbose "WMI queries complete"
                $obj = New-Object -TypeName PSObject -Property $props
                $obj.PSObject.TypeNames.Insert(0,'MOL.SystemInfo')
                Write-Output $obj
            }
        }
    }
    END {}
}
```

That's all we need to do, provided we only want the module to be visible to the currently logged-on user. Again, if we wanted the module to be shared among users, we'd have created a new path and added that to `PSModulePath`.

Running `Import-Module MOLTools` and then `Help Get-MOLSystemInfo` confirms that our module loads and works. We can then run `Get-MOLSystemInfo -computername localhost` to get the output of the command. But if you do that in a fresh shell window, you won't get the custom table view that we created in the previous chapter. Let's fix that next.

## 13.3 Creating a module manifest

A script module is intended to consist of a single .PSM1 file, and that's it. In our case, our module contents technically consist of MOLTools.psm1 and the XML view file we created in the previous chapter. A manifest would let us load both of those into memory at once, so let's create one. We'll start by copying the XML view file into our module folder:

```
PS C:\Users\donjones\Documents\WindowsPowerShell\Modules\MOLTools> copy C:\
test.format.ps1xml .\
PS C:\Users\donjones\Documents\WindowsPowerShell\Modules\MOLTools> ls

    Directory:
    C:\Users\donjones\Documents\WindowsPowerShell\Modules\MOLTools

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---          5/6/2012   5:52 PM       2833 MOLTools.psm1
-a---          5/6/2012   8:23 AM       2018 test.format.ps1xml
```

It seems silly to have that still named test.format.ps1xml, so let's rename it to MOLTools.format.ps1xml—that helps visually connect it to the script module file:

```
PS C:\Users\donjones\Documents\WindowsPowerShell\Modules\MOLTools> ren .\te
st.format.ps1xml MOLTools.format.ps1xml
PS C:\Users\donjones\Documents\WindowsPowerShell\Modules\MOLTools> ls

    Directory:
    C:\Users\donjones\Documents\WindowsPowerShell\Modules\MOLTools

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---          5/6/2012   8:23 AM       2018 MOLTools.format.ps1xml
-a---          5/6/2012   5:52 PM       2833 MOLTools.psm1
```

Now let's create a new module manifest. We're going to do so by running `NewModuleManifest` and providing the information needed using the command's parameters. Note that the module manifest filename must be MOLTools.psd1 in order for the shell to "see" the manifest.

```
PS C:\Users\donjones\Documents\WindowsPowerShell\Modules\MOLTools>
  New-ModuleManifest -Path MOLTools.psd1
                     -Author 'Don & Jeff'
        -CompanyName 'Month ofLunches'
```

```
-Copyright '(c)2012 Don Jones and Jeffery Hicks'
-Description 'Sample Module for Month of Lunches'
-FormatsToProcess .\MOLTools.format.ps1xml
-ModuleVersion 1.0
-PowerShellVersion 3.0
-RootModule .\MOLTools.psm1
```

**NOTE** We've formatted this nicely to fit in the book, but you'd type it all on one line.

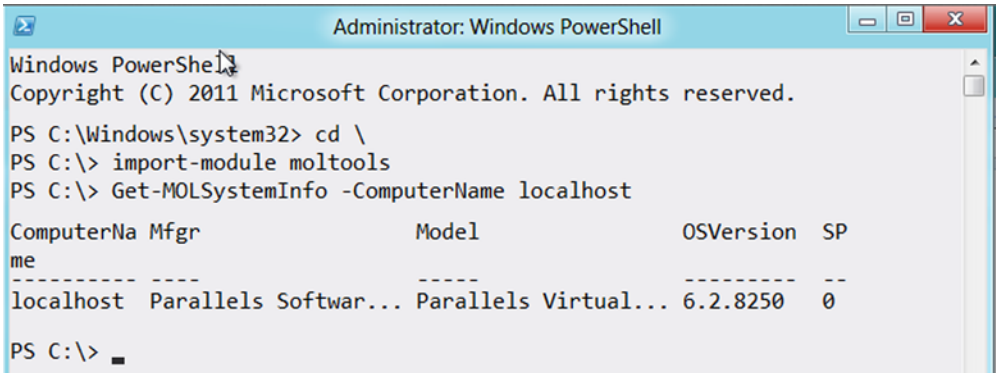
Aside from `-Path`, the `-FormatsToProcess` and `-RootModule` parameters are the really important ones. `-FormatsToProcess` is a comma-separated list of .format.ps1xml view files (or in our case, just the single file), and `-RootModule` is the "main" file in our module (in our case, our script module).

The root module is an important concept: Only the commands in the root module will be made visible to shell users. If our script module imported other modules, by including `Import-Module` commands within the script file or within one of its functions, those child modules wouldn't be visible to shell users (although someone could still manually import one of those modules, if they wanted to, to see their contents).

**TIP** Once you have a module manifest created, most likely a lot of it is boilerplate that you can reuse with other modules. There's nothing wrong with copying and pasting between .psd1 files and changing filenames as necessary. But you'll need to create a new GUID for each manifest, which is quite easy. Use this command in the shell to create one, `[guid]::NewGuid()`, and then copy and paste the result into your manifest. Any sections you don't need in the manifest you can comment out.

To test this, we're going to close the shell console and open a new one. Figure 13.1 shows that we can import the module, run the command, and get the formatted output defined in our XML view file. Success!

**TRY IT NOW** Make sure you can follow along to this point and get the same results that we do.



```
Windows PowerShell
Copyright (C) 2011 Microsoft Corporation. All rights reserved.

PS C:\Windows\system32> cd \
PS C:\> import-module moltools
PS C:\> Get-MOLSystemInfo -ComputerName localhost

ComputerNa Mfgr                 Model                OSVersion  SP
me
---------- ----                 -----                ---------  --
localhost  Parallels Softwar... Parallels Virtual... 6.2.8250   0

PS C:\> _
```

Figure 13.1 Testing the new module

## 13.4 Creating a module-level setting variable

Now that we've created a script module, we can take advantage of some other cool functionality provided by modules. For example, right now we're going to create a module variable. This will work a lot like the shell's built-in "preference" variables: The variable will be loaded into memory when the module is imported, and we'll use it to control an aspect of the module's behavior. The following listing shows the revised script file.

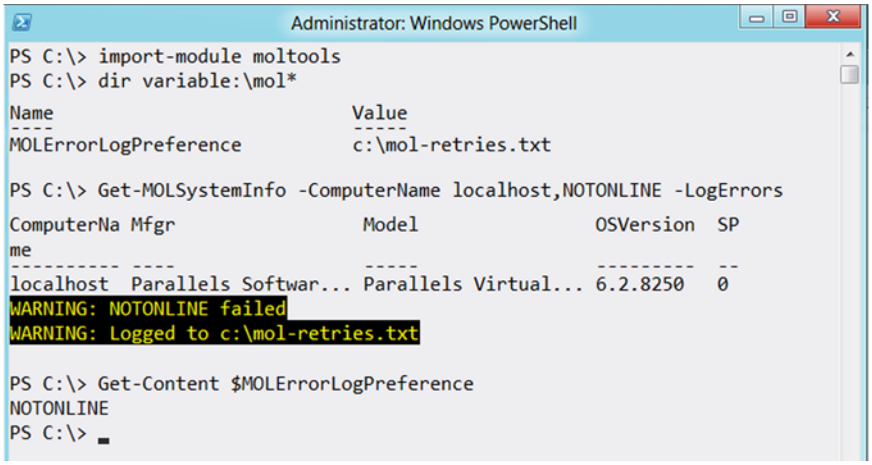
**Listing 13.2 Adding a module-level variable to MOLTools.psm1**

```
$MOLErrorLogPreference = 'c:\mol-retries.txt'          <-- Module-level variable

function Get-MOLSystemInfo {
<#
.SYNOPSIS
Retrieves key system version and model information
from one to ten computers.
.DESCRIPTION
Get-SystemInfo uses Windows Management Instrumentation
(WMI) to retrieve information from one or more computers.
Specify computers by name or by IP address.
.PARAMETER ComputerName
One or more computer names or IP addresses, up to a maximum
of 10.
.PARAMETER LogErrors
Specify this switch to create a text log file of computers
that could not be queried.
.PARAMETER ErrorLog
When used with -LogErrors, specifies the file path and name
to which failed computer names will be written. Defaults to
C:\Retry.txt.
.EXAMPLE
 Get-Content names.txt | Get-MOLSystemInfo
.EXAMPLE
 Get-MOLSystemInfo -ComputerName SERVER1,SERVER2
#>
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True,
                   ValueFromPipeline=$True,
                   HelpMessage="Computer name or IP address")]
        [ValidateCount(1,10)]
        [Alias('hostname')]
        [string[]]$ComputerName,

        [string]$ErrorLog = $MOLErrorLogPreference,      <-- Using the variable

        [switch]$LogErrors
    )
    BEGIN {
        Write-Verbose "Error log will be $ErrorLog"
    }
    PROCESS {
        Write-Verbose "Beginning PROCESS block"
        foreach ($computer in $computername) {
            Write-Verbose "Querying $computer"
            Try {
                $everything_ok = $true
                $os = Get-WmiObject -class Win32_OperatingSystem `
                                    -computerName $computer `
                                    -erroraction Stop
            } Catch {
                $everything_ok = $false
                Write-Warning "$computer failed"
                if ($LogErrors) {
                    $computer | Out-File $ErrorLog -Append
                    Write-Warning "Logged to $ErrorLog"
                }
            }

            if ($everything_ok) {
                $comp = Get-WmiObject -class Win32_ComputerSystem `
                                      -computerName $computer
                $bios = Get-WmiObject -class Win32_BIOS `
                                      -computerName $computer
                $props = @{'ComputerName'=$computer;
                           'OSVersion'=$os.version;
                           'SPVersion'=$os.servicepackmajorversion;
                           'BIOSSerial'=$bios.serialnumber;
                           'Manufacturer'=$comp.manufacturer;
                           'Model'=$comp.model}
                Write-Verbose "WMI queries complete"
                $obj = New-Object -TypeName PSObject -Property $props
                $obj.PSObject.TypeNames.Insert(0,'MOL.SystemInfo')
                Write-Output $obj
            }
        }
    }
    END {}
}
Export-ModuleMember -Variable MOLErrorLogPreference
Export-ModuleMember -Function Get-MOLSystemInfo
```

**Making the variable visible**

What we've done is add a `$MOLErrorLogPreference` variable to the module. It's not defined within one of the module's functions, so this becomes a module-level variable, meaning it will exist in the shell's memory as soon as the module is loaded. We've then utilized that to assign a default value to the `Get-MOLSystemInfo` command's `-ErrorLog` parameter. This now enables a user to set `$MOLErrorLogPreference` to a path and filename and have our command automatically use that as the default for the `-ErrorLog` parameter.

At the bottom of the revised script comes an important part. By default, module-level variables are private, meaning they can only be seen by other items within the module. Because our intent is to make the variable globally visible, we have to export it, using the `Export-ModuleMember` command. As soon as we use that command, everything in the module becomes private, meaning we also have to export our `Get-MOLSystemInfo` function in order for that to be globally visible as well.



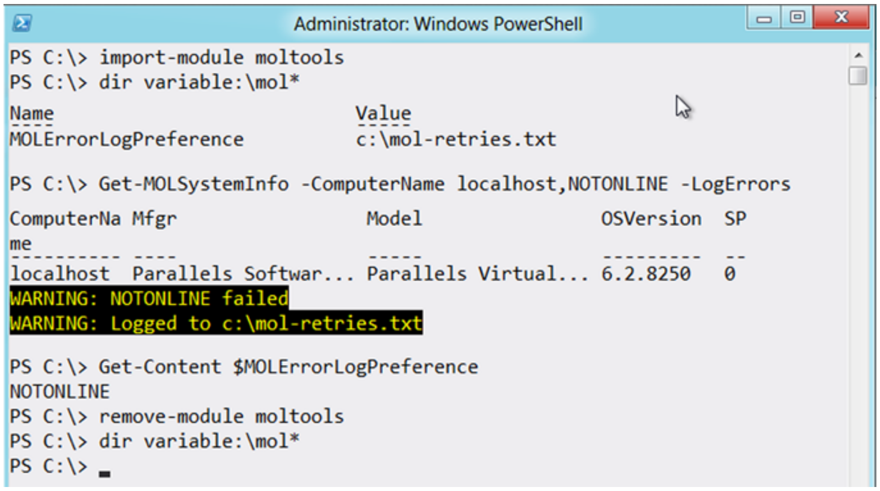
```
Administrator: Windows PowerShell
PS C:\> import-module moltools
PS C:\> dir variable:\mol*

Name                           Value
----                           -----
MOLErrorLogPreference          c:\mol-retries.txt

PS C:\> Get-MOLSystemInfo -ComputerName localhost,NOTONLINE -LogErrors

ComputerNa Mfgr                Model                OSVersion  SP
me
---------- ----                -----                ---------  --
localhost  Parallels Softwar... Parallels Virtual... 6.2.8250   0
WARNING: NOTONLINE failed
WARNING: Logged to c:\mol-retries.txt

PS C:\> Get-Content $MOLErrorLogPreference
NOTONLINE
PS C:\> _
```

**Figure 13.2 Testing the module-level variable**

Figure 13.2 shows that everything is working. We start by importing the module and then checking to see that `$MOLErrorLogPreference` has been added to the variable drive. We then run the command, adding the `-LogErrors` parameter. As you can see, the filename specified in `$MOLErrorLogPreference` has been created and filled with the name of the failed computer.

**NOTE** You can also create and export aliases in much the same way. Define the alias and then export it using `Export-ModuleMember`.

Figure 13.3 shows the real test: We removed the module and tested to make sure that `$MOLErrorLogPreference` was also removed from the shell. Our module is fully self-contained and can be completely loaded and unloaded on demand!



```
Administrator: Windows PowerShell
PS C:\> import-module moltools
PS C:\> dir variable:\mol*

Name                           Value
----                           -----
MOLErrorLogPreference          c:\mol-retries.txt

PS C:\> Get-MOLSystemInfo -ComputerName localhost,NOTONLINE -LogErrors

ComputerNa Mfgr                Model                OSVersion  SP
me
---------- ----                -----                ---------  --
localhost  Parallels Softwar... Parallels Virtual... 6.2.8250   0
WARNING: NOTONLINE failed
WARNING: Logged to c:\mol-retries.txt

PS C:\> Get-Content $MOLErrorLogPreference
NOTONLINE
PS C:\> remove-module moltools
PS C:\> dir variable:\mol*
PS C:\> _
```

**Figure 13.3 Removing the module from the shell's memory**

## 13.5 Coming up next

We're almost finished with `Get-SystemInfo`—but not quite. It's a "do something" function, and we'd like to show you some examples of "input" and "output" functions. We'd also like to show you how to access databases from within a PowerShell script, and we can probably take care of all of that in the next chapter.

## 13.6 Lab

In this chapter you're going to assemble a module called `PSHTools`, from the functions and custom views that you've been working on for the last several chapters. Create a folder in the user module directory, called PSHTools. Put all of the files you will be creating in the labs into this folder.

### 13.6.1 Lab A

Create a single ps1xml file that contains all of the view definitions from the three existing format files. Call the file PSHTools.format.ps1xml. You'll need to be careful. Each view is defined by the `<View></View>` tags. These tags and everything in between should go between the `<ViewDefinition></ViewDefinition>` tags.

### 13.6.2 Lab B

Create a single module file that contains the functions from the Labs A, B, and C in chapter 12, which should be the most current version. Export all functions in the module. Be careful to copy the function only. In your module file, also define aliases for your functions and export them as well.

### 13.6.3 Lab C

Create a module manifest for the `PSHTools` module that loads the module and custom format files. Test the module following these steps:

1. Import the module.
2. Use `Get-Command` to view the module commands.
3. Run help for each of your aliases.
4. Run each command alias using localhost as the computer name and verify formatting.
5. Remove the module.
6. Are the commands and variables gone?

**CAUTION** Once you finish these labs, please check the sample solutions at http://MoreLunches.com. Because you're going to continue building on these functions in some of the upcoming chapters, it's important that you have the correct solution (or close to it) before you continue.

POWERSHELL

# Learn POWERSHELL TOOLMAKING IN A MONTH OF LUNCHES

DON JONES and JEFFERY D. HICKS

**Free eBook**
see insert

You don't have to be a software developer to build PowerShell tools. With this book, a PowerShell user is a step away from becoming a proficient toolmaker.

*Learn PowerShell Toolmaking in a Month of Lunches* is the best way to learn PowerShell scripting and toolmaking in just one hour a day. It's packed with hands-on labs to reinforce what you're learning. It's an easy-to-follow guide that covers core scripting concepts using four practical examples. Each chapter builds on the previous one as you add custom formatting, error handling, input validation, help files, and more.

**What's Inside**

- Build your own administrative tools
- Learn by doing with hands-on labs
- Make scripts that feel like native PowerShell cmdlets

This book does not assume you are a programmer. Experience using PowerShell as a command-line interface is helpful but not required.

*Don Jones* is a PowerShell MVP, speaker, and trainer. *Jeffery Hicks* is a PowerShell MVP and an independent consultant, trainer, and author. Don and Jeff coauthored *Learn Windows PowerShell 3 in a Month of Lunches, Second Edition* (Manning 2012) and *PowerShell in Depth* (Manning 2012).

"Yet another great book from PowerShell legends, Don and Jeff."
—Thomas Lee
PowerShell MVP

"An excellent resource to begin learning a must-have skill set."
—Marc Johnson
Triangle Technology

"Take your PowerShell skills to the next level."
—Nathan Shelby
Snohomish Health District

"A perfect guide for creating PowerShell tools."
—Christoph Tohermes
netzkern AG

To download their free eBook in PDF, ePub, and Kindle formats,owners of this book should visit manning.com/LearnPowerShellToolmakinginaMonthofLunches

ISBN-13: 978-1617291166
ISBN-10: 1617291161
54499
9 781617 291166

MANNING US $44.99/CAN $47.99 [INCLUDING EBOOK]