CSC 368 – System Programming Languages
Spring 2017

Complete the following tutorial and answer the questions on pages 12.

Before we begin this homework, let's change the execution policy to allow scripts written on the local machine to be run. Open a PowerShell window as Administrator (the easiest way to do this and be in your preferred directory is to **open** File Explorer, browse to the directory of your choice, then **click on the File** tab, **selecting Open Windows PowerShell → Open Windows PowerShell as administrator**).
Once the prompt is available, type

```
Get-help  *execution*
```

to see what commands affect execution policy. For help on the command you find type get-help and the command name (e.g., get-help set-executionpolicy). As discussed in the slides and in many online sources, there are seven possible execution policies. You should choose RemoteSigned. Set your execution policy using the set-executionpolicy cmdlet.

You should recall that you always have to provide a path and filename in order to execute a script. You should also know the difference between running a script in the Integrated Scripting Environment (ISE) and the console. In the ISE, scripts run in the global scope. In the normal shell console, scripts get their own scope. For more information on the scope, click here.

Try the examples. If you type (or copy and paste) the script examples into the Windows PowerShell ISE or in PowerGUI (available on Blackboard).

Windows PowerShell Script Files

Just like Batch files, a Windows PowerShell script file is nothing more than a plain-text file. Rather than .BAT, it has a .PS1 filename extension. The "1" doesn't refer to the version of Windows PowerShell, but rather the version of the language engine. Windows PowerShell version 1 and 2 both use language engine version 1.

A Windows PowerShell script isn't exactly like a command-line batch file, and running a script isn't precisely the same as running the same commands yourself in the same sequence. For example, open a console window and run the following, pressing Enter after each line (remember not to type the line numbers):

```
Get-Service
Get-Process
```

Now **type those exact same lines into a script file**, or the ISE script editing pane, and run the script. You'll get different-looking results. Each time you hit Enter in Windows PowerShell, you start a new pipeline. Whatever commands you typed are run in that single pipeline. At the end of the pipeline, Windows PowerShell converts its contents into a text display. When you run the two commands in the normal console, you've done so in two distinct pipelines.

Windows PowerShell was able to construct a unique display for each set of output. When entered into a script, however, both commands ran in the same pipeline. The Windows PowerShell formatting system isn't sophisticated enough to construct the same unique output for two different sets of results. Try running this in the console:

```
Get-Service;Get-Process
```

Those results should look the same as they did when you ran the script containing those two commands. In this case, both commands ran in a single pipeline. That's what happened when you ran the script.

The practical upshot of all this is that a script should produce only one kind of output. It's a bad idea, due in large part to the limitations of the formatting system. There are other considerations, as well. You don't want a script dumping several different kinds of things into the pipeline at the same time.

Focus on that as a rule for everything we'll cover. A script should generate one, and only one, type of output. The only exception would be if it's a script being used as a repository for multiple functions. In that case, each function should generate one, and only one, type of output.

Variables

PowerShell is object-oriented, and all the data items used within it are objects. You can think of variables as boxes. You can put one or more things, even dissimilar things, into a box. Each box has a name, and in Windows PowerShell those names can include almost anything. "**`Var`**" can be a variable name, as can "**`{my variable}`**". In the second example, the curly braces enclose a variable name that contains spaces, which is pretty ugly. As a rule, try to stick with variable names that include letters, numbers and underscores.

Using a variable's name references the entire "**box**." However, because the box is an object, there is a lot of other information you may not want. If you want to reference the **contents of the box**, add a **dollar sign**: **`$var`**. You'll often see Windows PowerShell variables preceded with the dollar sign because the whole point of using one is to get at the contents. It's important to remember, however, that **the dollar sign isn't part of the variable name**. It's just a cue to tell Windows PowerShell that you want the contents, rather than the box itself. For example:

```
$var = 'hello'
$number = 1
$numbers = 1,2,3,4,5,6,7,8,9
```

Those examples show you how to place items into a variable using the assignment operator (**`=`**). That last example creates an array, because Windows PowerShell interprets all comma-separated lists as an array, or collection, of items. The first example assigns a string object, with the characters in the string contained within quotation marks.

There's one aspect of Windows PowerShell that can be confusing, Windows PowerShell doesn't "understand" any meaning you may associate with a variable name. A variable like **`$computername`** doesn't "tell" the shell that the variable will contain a computer name.

Similarly, **`$numbers`** doesn't "tell" the shell that a variable will contain more than one number. The shell doesn't care if you use a plural variable name. The statement

```
$numbers = 1
```

is **equally valid** to the shell, as is

```
$numbers = 'fred.'
```

When a variable does contain multiple values, however, you can use a special syntax to access just a single one of them. You would use **`$numbers[0]`** as the first item, **`$numbers[1]`** as the second, **`$numbers[-1]`** as the last, **`$numbers[-2]`** as the second-last, and so on.

Quotation Marks

As a best practice, **use single quotes to delimit a variable** unless you have a specific reason to do otherwise. There are three specific instances where you would want to use double quotes.

The first is **when you need to insert a variable's contents into a string**. Within double quotes only, Windows PowerShell will look for the $, and will assume that everything after the $, up to the first character that's illegal in a variable name, is a variable name. The contents of that variable will replace the variable name and the $:

```
$name = 'Don'
$prompt = "My name is $name"
```

The **`$prompt`** will now contain "My name is Don" because **`$name`** will be replaced with the variable contents. This is a great trick for joining strings together without having to concatenate them.

Use double quotes in Windows PowerShell when you need to look for PowerShell's **escape character, the backtick or grave accent**, and act accordingly. Here are a couple of examples:

```
$debug = "`$computer contains $computer"
$head = "Column`tColumn`tColumn"
```

In the first example, **the first $ is being "escaped."** That removes its special meaning as a variable accessor. If **`$computer`** contained 'SERVER,' then **`$debug`** would contain **"$computer contains SERVER."**

In the second example, **`` `t `` represents a horizontal tab character**, so Windows PowerShell will place a tab between each Column. You can read about other special escape characters here, or in the shell's **`about_escape_characters`** help topic.

Finally, use double quotes **when a string needs to contain single quotes**:

```
$filter1 = "name='BITS'"
$computer = 'BITS'
$filter2 = "name='$computer'"
```

In this example, the literal string is name=**`'BITS'`**. The double quotes contain the whole thing. Both **`$filter1`** and **`$filter2`** end up containing exactly the same thing, but **`$filter2`** gets there by using the variable-replacement trick of double quotes. Note that **only the outermost set of quotes actually matters**. The single quotes inside the outer double quotes are considered part of the string by Windows PowerShell.

Object Members and Variables

Everything in Windows PowerShell is an object. Even a simple string such as “name” is an object, of the type `System.String`. You can **pipe** (just as in Batch files and command line, the ‘|’ character is the pipe operator). any object to Get-Member to see its type name (that is, the kind of object it is) as well as its members, which includes its properties and methods:

```
$var = 'Hello'
$var | Get-Member
```

Use a **period after a variable name** to tell the shell, “I don’t want to access the entire object within this variable. I want to **access just one of its properties or methods**.” After the period, provide the property or method name.

Method names are always followed by a set of parentheses. Some methods accept **input arguments, and those go within the parentheses in a comma-separated list**. Other methods require no arguments, and so the parentheses are empty, but don't forget the parentheses:

```
$svc = Get-Service
$svc[0].name
$name = $svc[1].name
$name.length
$name.ToUpper()
```

Notice the second line starts by accessing the first item in the `$svc` variable. The period means, “I don’t want that entire object. I just want a property or method.” This **accesses just the name property**. **Line five illustrates how to access a method**, by providing its name after a period, followed by parentheses.

A period is normally an illegal character within a variable name, because the period means we want to access a property or method. That means line two in the following example won’t work the way you might expect:

```
$service = 'bits'
$name = "Service is $service.ToUpper()"
$upper = $name.ToUpper()
$name = "Service is $upper"
```

On line two, `$name` will contain "Service is BITS.ToUpper()" whereas on line four `$name` will contain “Service is BITS.”

Parentheses

Aside from their use with object methods, **parentheses also act as an order-of-execution marker** for Windows PowerShell, just like in algebra. In other words, parentheses tell the shell to “execute this first.” The entire parenthetical expression is replaced by whatever that expression produces. Here’s a mind-bending couple of examples:

```
$name = (Get-Service)[0].name
Get-Service -computerName (Get-Content names.txt)
```

In the first line, `$name` will contain the name of the first service on the system. Reading this takes a bit of effort. Start with the parenthetical expression. That's what Windows PowerShell will start with as well. The "`Get-Service`" resolves to a collection, or array, of services. The **[0] accesses the first item** in an array, so that will be the first service. **Because it's followed by a period, we know we're accessing a property or method of that service**, rather than the entire service object. Finally, we pull out just the name of the service.

On line two, the parenthetical expression is reading the contents of a text file. Assuming the file contains one computer name per line, "Get-Content"" will return an array of computer names. Those are fed to the "–computerName" parameter of "Get-Service." In this case, the shell can feed any parenthetical expression that returns an array of strings to the "–computerName" parameter, because the parameter is designed to accept arrays of strings.

Scope

Scope is a programming concept that acts as a container or compartment system. Things like variables, aliases, PSDrives and other Windows PowerShell elements are all stored in a scope. The shell maintains a **hierarchy of scopes**, and has a set of rules that determine how scopes can interact and share information with each other.

The **shell itself is a single scope, called the global scope**. **When you run a script, it constructs a new scope and the script runs within that**. Anything created by the script, such as a new variable, is stored within the script's scope. It isn't accessible by the top-level shell.

When the script finishes running, its scope is discarded, and anything created within that scope disappears. For example, create a script called `hw3drive.ps1` that contains the following and then run that script from the console window:

```
New-PSDrive -PSProvider FileSystem -Root C:\ -Name Sys
Dir SYS:
```

After running the script, manually run "Dir SYS:" and you should see an error. That's because the `SYS:` drive was created in the script. Once the script was done, everything it created was discarded. The `SYS:` drive no longer exists. Some things in a shell are not scoped. **Items such as modules are handled globally at all times**. A script can load a module and the module will remain loaded after the script is done.

If a scope tries to access something that wasn't created within that scope, then Windows PowerShell looks to the next-higher scope (the "parent" scope). That's why the Dir alias worked in that script you just entered. Although Dir didn't exist in the script's scope, it did exist in the next-higher scope: the global scope. A scope is free to create an item that has the same name as an item from a higher-level scope, though. Here's another script to try, save it as `hw3dir.ps1`:

```
New-Alias Fold Dir
Fold
Set-Alias Fold Get-Host
Fold
```

The first time Fold runs it is an alias for the globally defined Dir command. We can then redefine it to the Get-Host command. Now try setting Dir as an alias, enter the following in **`hw3dira.ps1`**.

```
Dir
Set-Alias Dir Get-Alias
Dir
```

Later versions of PowerShell prevent reserved words and some of the default aliases from accidently being erased or replaced.

Scope can be especially confusing when it comes to variables**. As a rule, a given scope should never access out-of-scope items**, especially variables. There's a syntax for doing so, such as using **`$global:var`** to forcibly access the global scope's **`$var`** variable, but that's a bad practice except under very specific circumstances.

Windows PowerShell Scripting Language

Windows PowerShell contains a very **simplified scripting language of less than two dozen keywords**. That's a stark contrast to a full programming language such as VBScript, which contains almost 300 keywords.

Simplified though it may be, the Windows PowerShell language is more than sufficient to do the job, especially since in integrates with .NET, WMI, and other Microsoft products. You can always get more help on the major constructs of the language by reading the appropriate "about" topic within the shell. For example, help **`about_switch`** contains information on the **`Switch`** construct, while help **`about_if`** contains information on the **`If`** construct. Run help about* for a list of all "about" topics.

The If Construct

This is the Windows PowerShell main **decision-making** construct. In its full form, it looks like this:

```
If ($this -eq $that) {
  # commands
} elseif ($those -ne $them) {
  # commands
} elseif ($we -gt $they) {
  # commands
} else {
  # commands
}
```

The "If" keyword is a mandatory part of this construct. A parenthetical expression follows that must evaluate to either True or False. Windows PowerShell will always interpret **zero as False**, and any **nonzero value as True**. If you've ever programmed in C this is very familiar.

Windows PowerShell also recognizes the built-in variables **`$True`** and **`$False`** as representing those Boolean values. If the expression in parentheses works out to True, then the commands in the following set of curly brackets will execute. If the expression is False, then the commands won't execute. That's really all you need for a valid If construct.

You can go a bit further by **providing one or more `ElseIf` sections**. These work the same way as the If construct. **They get their own parenthetical expression**. If it's True, the commands within the following curly brackets will execute. If not, they won't.

You can **wrap up with an `Else` block**, which will **execute if none of the preceding blocks execute**. Only **the block associated with the first True expression will execute**. For example, if **`$this`** did not equal **`$that`**, and **`$those`** did not equal **`$them`**, then the commands on line four would execute—and nothing else. Windows PowerShell **won't even evaluate the second `elseif` expression on line five**.

The **#** character is a comment character, making Windows PowerShell essentially ignore anything from there until the end of the line (a carriage return). Also, notice the care with which those constructs were formatted. You might also see formatting like this from some folks:

```
if ($those -eq $these)
{
  #commands
}
```

It **doesn't matter where you place the curly brackets**. However, what does matter is that you be consistent in your placement so your scripts are easier to read. It's also important to indent, to the exact same level, every line within the curly brackets.

The Windows PowerShell ISE lets you use the Tab key for that purpose, and it defaults to a four-character indent. **Indenting your code is a core best practice**. If you don't, you'll have a tough time properly matching opening and closing curly brackets in complex scripts. Also, all of the other Windows PowerShell kids will make fun of you. Consider this poorly formatted script:

```
function mine {
if ($this -eq $that){
get-service
}}
```

That's a lot harder to read, debug, troubleshoot and maintain. While the space after the closing parentheses isn't necessary, it does make your script easier to read. The indented code isn't necessary, but it makes your script easier to follow. Consider this instead:

```
function mine {
 if ($this -eq $that){
  get-service
 }
}
```

Placing a single closing curly bracket on a line by itself isn't required by the shell, but it's appreciated by human eyes. Be a neat formatter, and you'll have fewer problems in your scripts.

The Do While Construct

This is a looping construct in Windows PowerShell. It's designed to **repeat a block of commands as long as some condition is True**, or until a condition becomes True. Here's the basic usage:

```
Do {
  # commands
} While ($this -eq $that)
```

In this variation of the construct, **the commands within the curly brackets will always execute at least once**. The While condition isn't evaluated until after the first execution. You can **move the While, in which case the commands will only execute if the condition is True** in the first place:

```
While (Test-Path $path) {
  # commands
}
```

Notice **the second example doesn't use a comparison operator such as -eq**. That's **because the Test-Path cmdlet returns True or False** to begin with. There's no need to compare that to True or False in order for the expression to work.

The **parenthetical expression used with these scripting constructs merely needs to evaluate to True or False**. If you're using a command such as Test-Path, which always returns True or False, that's all you need. As always, there's an "about" topic in the shell that demonstrates other ways to use this construct.

The ForEach Construct

This construct is similar in operation to the **`ForEach-Object`** cmdlet. It differs only in its syntax. The purpose of **`ForEach`** is to take an array (or collection, which in Windows PowerShell is the same as an array) and enumerate the objects in the array so you can work with them one at a time:

```
$services = Get-Service
ForEach ($service in $services) {
    $service.Stop()
}
```

It's easy for newcomers to overthink this construct. Keep in mind that the plural English word "services" doesn't mean anything to Windows PowerShell. That variable name is used to remind us it contains one or more services. Just because it's plural doesn't make the shell behave in a special fashion.

The "**`in`**" keyword on line two is part of the **`ForEach`** syntax. The **`$service`** variable is made up. It could easily have been **`$fred`** or $**`coffee`** and it would have worked in just the same way.

Windows PowerShell will repeat the construct's commands—the ones contained within curly brackets—one time for each object in the second variable (**`$services`**). Each time, it will take a single object from the second variable (**`$services`**) and place it in the first variable (**`$service`**).

Within this construct, use the first variable (**`$service`**) to work with an individual object. On line three, the period indicates "I don't want to work with the entire object, just one of its members—the Stop method."

There are times when using **`ForEach`** is inevitable and desirable. However, if you have a bit of programming or scripting experience, you can sometimes leap to using **`ForEach`** when it isn't the best approach. The previous example isn't a good reason to use **`ForEach`**. Wouldn't the following code be easier:

```
Get-Service | Stop-Service
```

The point here is to evaluate your use of **`ForEach`**. Make sure it's the only way to accomplish the task at hand. Here are some instances where **`ForEach`** is probably the only way to go:

- When you need to execute a method against a bunch of objects and there's no cmdlet that performs the equivalent action.
- When you have a bunch of objects and need to perform several consecutive actions against each.
- When you have an action that can only be performed against one object at a time, but your script may be working with one or more objects, and you have no way of knowing in advance.

Other Constructs

Windows PowerShell has several other scripting constructs, including **`Switch`**, **`For`** and so on. These are all documented in "about" help topics within the shell and in the first PowerShell PowerPoint file on Blackboard. Sometimes, you can use the constructs covered here to replace those other constructs. For example, you can replace **`Switch`** with an **`If`** construct that uses multiple ElseIf sections. You can replace For with **`ForEach`**, or even with the **`ForEach-Object`** cmdlet. For example, having a loop that executes exactly 10 times:

```
1..10 | ForEach-Object -process {
  # code here will repeat 10 times
  # use $_ to access the current iteration
  # number
}
```

It's up to you to select the best construct to get the job done. If you're browsing the Internet for scripts, be prepared to run across any and all variations.

Functions

A **function** is a special kind of construct used to **contain a group of related commands that perform a single, specific task**. Generally speaking, you can take any Windows PowerShell script and "wrap" it within a function:

```
function Mine {
  Get-Service
  Get-Process
}
Mine
```

This **defines a new function called `Mine`.** That basically turns **`Mine`** into a command, meaning **you can run the function simply by entering its name**. That's what line five does. It runs the function.

Functions are typically contained within a script file. **A single script can contain multiple functions and functions can contain other functions**.

However, **functions are scoped items**. That means **you can only use a function within the same scope in which it was created**. If you put a function into a script, and then run that script, the function will only be available within the script and only for the duration of the script. When the script finishes running, the function—like everything else in the script's scope—goes away. Here's one example:

```
function One {
  function Two {
Dir
  }
  Two
}
One
Two
```

Suppose you enter this into a single script file and run that script. The seventh line executes the function `One`, which starts on line one. Line five executes a function named `Two`, which starts on line two. The result will be a directory listing, which is on line three inside function `Two`.

However, **the next line to execute will be line eight, and that will result in an error**. The script doesn't contain a function named `Two`. Function Two is buried within function One, and therefore exists only within the function One scope. **Only things within function `One` can see `Two`**. Attempting to call `Two` from anyplace else will result in an error.

Adding Parameters to a Script

It's rare to create a script that's intended to do exactly the same thing every time it runs. More frequently, you'll have scripts that contain some kind of variable data or variable behavior. You can accommodate these variations with parameters.

Parameters are defined in a special way at the top of the script. You can precede this definition with comments, but it must be the first executable lines of code within the script. Within the parameter definition area**, each parameter is separated from the next by a comma**. In keeping with the idea of neat formatting, it helps to place each parameter on a line of its own. Here is an example:

```
param (
  [string]$computername,
  [string]$logfile,
  [int]$attemptcount = 5
)
```

This **example defines three parameters**. **Within the script, these are simply used like any other variable**. You'll notice that on the fourth line, there is a default value assigned to the `$attemptcount` parameter. The default will be **overridden by any input parameter**, but will be used if the script is run without that parameter being specified.

Here are several ways in which the script might be run, assuming it was saved with the name **`Test.ps1`**:

```
.\test -computername SERVER
.\test -comp SERVER -log err.txt -attempt 2
.\test SERVER err.txt 2
.\test SERVER 2
.\test -log err.txt -attempt 2 -comp SERVER
```

The script accepts parameters pretty much like any cmdlet. Variable names are used as the parameter names, specified with the usual **dash** that precedes all parameter names in Windows PowerShell. Here's a breakdown of how it works:

- On line one there is only one parameter specified — **`$logfile`** will thus be empty, and **`$attemptcount`** will contain 5, its default.
- On line two, all three parameters, though it is done using the shortened parameter names. As with cmdlets, you only need **to type enough of the parameter name for Windows PowerShell to know which one you're talking about**.
- Line three also has all three parameters, this time **indicated by position**, without using parameter names. As long as **values are in the exact order in which the parameters are listed** in the script this will work fine.
- Line four shows what happens if you're not careful. Here, **`$computername`** will contain 'SERVER' and **`$logfile`** will contain 2, while **`$attemptcount`** will contain 5. That's probably not what was intended. When you don't use parameter names, it's harder to be accurate. It's also more difficult for someone else to decode what you meant, which makes it harder for them to troubleshoot any problems.
- Line five is a better example. Here, parameters are specified out of order, but that's fine because parameter names are used. As a general rule, I always use parameter names for the greatest degree of flexibility and accuracy.

Advanced Scripts

Windows PowerShell supports a technique for specifying additional information about parameters. This lets you declare a parameter as mandatory, accepting input from the pipeline and so forth. This technique is called **Cmdlet Binding**.

It doesn't change the way the script uses parameters. It simply gives the shell a bit more information about the parameters. You'll find **this technique more commonly used in a function, but the syntax is valid within a script** as well. Here's a simple example:

```
[CmdletBinding()]
param (
  [Parameter(Mandatory=$True)]
  [string]$computername,

  [Parameter(Mandatory=$True)]
  [string]$logfile,

  [int]$attemptcount = 5
)
```

The [CmdletBinding()] instruction must be the first executable line of code within the script. It's okay for comments to precede this, but nothing else. There are also [Parameter()] instructions added to two parameters. The [Paramater()] instructions indicate that these parameters are mandatory. If someone tries to run the script without specifying these parameters Windows PowerShell will prompt them for the information.

Notice that the last parameter doesn't have any special instructions, and all three parameters still appear in a comma-separated list (meaning the first two parameters are followed by commas). There are a ton of other instructions you can specify for a parameter, which you can read about here or use **`get-help`** on the **`about_functions_advanced_parameters`** topic at the PowerShell prompt.

Answer the following questions and post the answers to Blackboard:

1. Described what is returned by each of the following commands:
    - a. Get-Item env:\username
    - b. Get-Item env:\Computername
    - c. Get-ChildItem env:\userprofile

2. How would you change the commands in #1 to return only the value (look at the help information)?

3. How can you get the current date and time from the system within Powershell?

4. Describe the result of Get-WmiObject -Class for each of the following classes (see here or the help information for Get-WmiObject syntax).
    - a. Win32_ComputerSystem
    - b. Win32_BootConfiguration
    - c. Win32_BIOS
    - d. Win32_OperatingSystem
    - e. Win32_TimeZone
    - f. Win32_LogicalDisk
        - f.1. Add "-Filter DriveType=3"
    - g. Win32_Processor
    - h. Win32_PhysicalMemory
    - i. Win32_Product