16 Variables: A place to store your stuff

We've already mentioned that PowerShell contains a scripting language, and in a few more chapters, we'll start to play with it. But once you start scripting, you may want to store your objects as variables for later use, so we'll get those out of the way in this chapter. You can use variables in many places other than long, complex scripts, so we'll also use this chapter to show you some practical ways to use them.

16.1 Introduction to variables

A simple way to think of a *variable* is as a box in the computer's memory that has a name. You can put whatever you want into the box: a single computer name, a collection of services, an XML document, and so on. You access the box by using its name, and when accessing it, you can put things in it or retrieve things from it. Those things stay in the box, allowing you to retrieve them over and over.

PowerShell doesn't place a lot of formality around variables. For example, you don't have to explicitly announce or declare your intention to use a variable before you do so. You can also change the types or objects of the contents of a variable: one moment you might have a single process in it, and the next moment you can store an array of computer names in it. A variable can even contain multiple different things, such as a collection of services *and* a collection of processes (although we admit that, in those cases, using the variable's contents can be tricky).

16.2 Storing values in variables

Everything in PowerShell—and we do mean *everything*—is treated as an object. Even a simple string of characters, such as a computer name, is considered an object. For example, piping a string to `Get-Member` (or its alias, `gm`) reveals that the

object is of the type `System.String` and that it has a great many methods you can work with (we're truncating the following list to save space):

```
PS > "SRV-02" | Get-Member
```

This gives you:

```
   TypeName: System.String

Name                 MemberType           Definition
----                 ----------           ----------
Clone                Method               System.Object Clone(), System.O...
CompareTo            Method               int CompareTo(System.Object val...
Contains             Method               bool Contains(string value), bo...
CopyTo               Method               void CopyTo(int sourceIndex, ch...
EndsWith             Method               bool EndsWith(string value), bo...
EnumerateRunes       Method               System.Text.StringRuneEnumerato...
Equals               Method               bool Equals(System.Object obj),...
GetEnumerator        Method               System.CharEnumerator GetEnumer...
GetHashCode          Method               int GetHashCode(), int GetHashC...
GetPinnableReference Method               System.Char&, System.Private.Co...
GetType              Method               type GetType()
```

TRY IT NOW Run this same command in PowerShell to see if you get the complete list of methods—and even a property—that comes with a `System.String` object.

Although that string is technically an object, you'll find that folks tend to refer to it as a simple value like everything else in the shell. That's because, in most cases, what you're concerned about is the string itself—`"SRV-02"` in the previous example—and you're less concerned about retrieving information from properties. That's different from, say, a process where the entire process object is a big, abstract data construct, and you're usually dealing with individual properties such as `VM`, `PM`, `Name`, `CPU`, `ID`, and so forth. A `String` is an object, but it's a much less complicated object than something like a `Process`.

PowerShell allows you to store these simple values in a variable. To do this, specify the variable, and use the equals sign operator—the *assignment* operator—followed by whatever you want to put within the variable. Here's an example:

```
$var = "SRV-02"
```

TRY IT NOW Follow along with these examples, because then you'll be able to replicate the results we demonstrate. You should use your test server's name rather than `SRV-02`.

It's important to note that the dollar sign (`$`) isn't part of the variable's name. In our example, the variable name is `var`. The dollar sign is a cue to the shell that what follows

is going to be a variable name and that we want to access the contents of that variable. In this case, we're setting the contents of the variable.

Let's look at some key points to keep in mind about variables and their names:

- Variable names usually contain letters, numbers, and underscores, and it's most common for them to begin with a letter or an underscore.
- Variable names can contain spaces, but the name must be enclosed in curly braces. For example, `${My Variable}` represents a variable named `My Variable`. Personally, we dislike variable names that contain spaces because they require more typing, and they're harder to read.
- Variables don't persist between shell sessions. When you close the shell, any variables you created go away.
- Variable names can be quite long—long enough that you don't need to worry about how long. Try to make variable names sensible. For example, if you'll be putting a computer name into a variable, use `computername` as the variable name. If a variable will contain a bunch of processes, then `processes` is a good variable name.
- Some folks who have experience with other scripting languages may be used to using prefixes to indicate what is stored in the variable. For example, `strComputerName` is a common type of variable name, meaning that the variable holds a string (the `str` part). PowerShell doesn't care whether you do that, but it's no longer considered a desirable practice by the PowerShell community.

To retrieve the contents of a variable, use the dollar sign followed by the variable name, as shown in the following example. Again, the dollar sign tells the shell that you want to access the *contents* of a variable; following it with the variable name tells the shell which variable you're accessing:

```
$var
```

This outputs

```
SRV-02
```

You can use a variable in place of a value in almost any situation, for example, when using `Get-Process ID`. The command might typically look like this:

```
Get-Process -Id 13481
```

This outputs

```
 NPM(K)    PM(M)      WS(M)     CPU(s)      Id  SI ProcessName
 ------    -----      -----     ------      --  -- -----------
      0     0.00      86.21       4.12   13481 ...80 pwsh
```

You can substitute a variable for any of the values:

```
$var = "13481"

Get-Process -Id $var
```

Which gives you

```
NPM(K)    PM(M)      WS(M)     CPU(s)      Id  SI ProcessName
------    -----      -----     ------      --  -- -----------
     0     0.00      86.21       4.12   13481 ...80 pwsh
```

By the way, we realize that `var` is a pretty generic variable name. We'd normally use `processId`, but in this specific instance, we plan to reuse `$var` in several situations, so we decided to keep it generic. Don't let this example stop you from using more sensible variable names in real life. We may have put a string into `$var` to begin with, but we can change that anytime we want:

```
PS > $var = 5
PS > $var | get-member
   TypeName: System.Int32
Name        MemberType Definition
----        ---------- ----------
CompareTo   Method     int CompareTo(System.Object value), int CompareT...
Equals      Method     bool Equals(System.Object obj), bool Equals(int ...
GetHashCode Method     int GetHashCode()
GetType     Method     type GetType()
GetTypeCode Method     System.TypeCode GetTypeCode()
```

In the preceding example, we placed an integer into `$var`, and then we piped `$var` to `Get-Member`. You can see that the shell recognizes the contents of `$var` as a `System.Int32`, or a 32-bit integer.

16.3 Using variables: Fun tricks with quotes

Because we're talking about variables, this is an excellent time to cover a neat PowerShell feature. Up to this point in the book, we've advised you to generally enclose strings within single quotation marks. The reason for that is PowerShell treats everything enclosed in single quotation marks as a literal string.

Consider the following example:

```
PS > $var = 'What does $var contain?'
PS > $var
What does $var contain?
```

Here you can see that the `$var` within single quotes is treated as a literal. But in double quotation marks, that's not the case. Check out the following trick:

```
PS > $computername = 'SRV-02'
PS > $phrase = "The computer name is $computername"
PS > $phrase
The computer name is SRV-02
```

We start our example by storing `SRV-02` in the variable `$computername`. Next, we store `"The computer name is $computername"` in the variable `$phrase`. When we do this, we use double quotes. PowerShell automatically seeks out dollar signs within double quotes and replaces any variables it finds *with their contents*. Because we display the contents of `$phrase`, the `$computername` variable is replaced with `SRV-02`.

This replacement action happens only when the shell initially parses the string. At this point, `$phrase` contains `"The computer name is SRV-02"`—it doesn't contain the `"$computername"` string. We can test that by trying to change the contents of `$computername` to see whether `$phrase` updates itself:

```
PS > $computername = 'SERVER1'
PS > $phrase
The computer name is SRV-02
```

As you can see, the `$phrase` variable stays the same.

Another facet of this double-quotes trick is the PowerShell escape character. This character is the backtick (`), and on a US keyboard it's located on one of the upper-left keys, usually below the Esc key and often on the same key as the tilde (~) character. The problem is that, in some fonts, it's practically indistinguishable from a single quote. In fact, we usually configure our shell to use the Consolas font, because that makes distinguishing the backtick easier than when using the Lucida Console or Raster fonts.

Let's look at what this escape character does. It removes whatever special meaning might be associated with the character after it, or in some cases, it adds special meaning to the following character. We have an example of the first use:

```
PS > $computername = 'SRV-02'
PS > $phrase = "`$computername contains $computername"
PS > $phrase
$computername contains SRV-02
```

When we assign the string to `$phrase`, we use `$computername` twice. The first time, we precede the dollar sign with a backtick. Doing this takes away the dollar sign's special meaning as a variable indicator and makes it a literal dollar sign. You can see in the preceding output, on the last line, that `$computername` is stored in the variable. We don't use the backtick the second time, so `$computername` is replaced with the contents of that variable. Now let's look at an example of the second way a backtick can work:

```
PS > $phrase = "`$computername`ncontains`n$computername"
PS > $phrase
$computername
contains
SRV-02
```

Look carefully, and you'll notice we use `` `n `` twice in the phrase—once after the first `$computername` and once after `contains`. In the example, the backtick adds special meaning. Normally, *n* is a letter, but with the backtick in front of it, it becomes a carriage return and line feed (think *n* for *new line*).

Run `help about_escape` for more information, including a list of other special escape characters. You can, for example, use an escaped *t* to insert a tab, or an escaped *a* to make your computer beep (think *a* for *alert*).

16.4 Storing many objects in a variable

Up till now, we've been working with variables that contain a single object, and those objects have all been simple values. We've worked directly with the objects themselves, rather than with their properties or methods. Let's now try putting a bunch of objects into a single variable.

One way to do this is to use a comma-separated list, because PowerShell recognizes those lists as collections of objects:

```
PS > $computers = 'SRV-02','SERVER1','localhost'
PS > $computers
SRV-02
SERVER1
Localhost
```

Notice that we're careful in this example to put the commas outside the quotation marks. If we put them inside, we'd have a single object that includes commas and three computer names. With our method, we get three distinct objects, all of which are `String` types. As you can see, when we examine the contents of the variable, PowerShell displays each object on its own line.

16.4.1 Working with single objects in a variable

You can also access individual elements in the variable, one at a time. To do this, specify an index number for the object you want, in square brackets. The first object is always at index number `0`, and the second is at index number `1`, and so forth. You can also use an index of `-1` to access the last object, `-2` for the next-to-last object, and so on. Here's an example:

```
PS > $computers[0]
SRV-02
PS > $computers[1]
SERVER1
```

```
PS > $computers[-1]
localhost
PS > $computers[-2]
SERVER1
```

The variable itself has a property that lets you see how many objects are in it:

```
$computers.count
```

This results in

```
3
```

You can also access the properties and methods of the objects inside the variable as if they were properties and methods of the variable itself. This is easier to see, at first, with a variable that contains a single object:

```
PS > $computername.length
6
PS > $computername.toupper()
SRV-02
PS > $computername.tolower()
srv-02
PS > $computername.replace('02','2020')
SRV-2020
PS > $computername
SRV-02
```

In this example, we're using the `$computername` variable we created earlier in the chapter. As you may remember, that variable contains an object of the type `System.String`, and you should have seen the complete list of properties and methods of that type when you piped a string to `Get-Member` in section 16.2. We use the `Length` property, as well as the `ToUpper()`, `ToLower()`, and `Replace()` methods. In each case, we have to follow the method name with parentheses, even though neither `ToUpper()` nor `ToLower()` requires any parameters inside those parentheses. Also, none of these methods change what is in the variable—you can see that on the last line. Instead, each method creates a new `String` based on the original one, as modified by the method.

What if you want to change the contents of the variable? You can assign a new value to the variable pretty easily:

```
PS > $computers = "SRV-02"
PS > $computers
SRV-02

PS > $computers = "SRV-03"
PS > $computers
SRV-03
```

16.4.2 Working with multiple objects in a variable

When a variable contains multiple objects, the steps can get trickier. Even if every object inside the variable is of the same type, as is the case with our `$computers` variable, and you can call a method on every object, it might not be what you want to do. You probably want to specify which object within the variable you want and then access a property or execute a method on that specific object:

```
PS > $computers[0].tolower()
SRV-02
PS > $computers[1].replace('SERVER','CLIENT')
CLIENT1
```

Again, these methods are producing new strings, not changing the ones inside the variable. You can test that by examining the contents of the variable:

```
PS > $computers
SRV-02
SERVER1
Localhost
```

What if you want to change the contents of the variable? You assign a new value to one of the existing objects:

```
PS > $computers[1] = $computers[1].replace('SERVER','CLIENT')
PS > $computers
SRV-02
CLIENT1
Localhost
```

You can see in this example that we change the second object in the variable, rather than produce a new string.

16.4.3 Other ways to work with multiple objects

We want to show you two other options for working with the properties and methods of a bunch of objects contained in a variable. The previous examples executed methods on only a single object within the variable. If you want to run the `ToLower()` method on every object within the variable, and store the results back into the variable, you do something like this:

```
PS > $computers = $computers | ForEach-Object { $_.ToLower()}
PS > $computers
srv-02
client1
localhost
```

This example is a bit complicated, so let's break it down in figure 16.1. We start the pipeline with `$computers =`, which means the results of the pipeline will be stored in that variable. Those results overwrite whatever was in the variable previously.

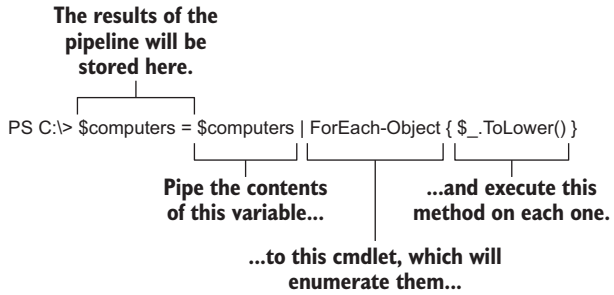

Figure 16.1 Using `ForEach-Object` to execute a method against each object contained within a variable

The pipeline begins with `$computers` being piped to `ForEach-Object`. The cmdlet enumerates each object in the pipeline (we have three computer names, which are `string` objects) and executes its script block for each. Within the script block, the `$_` placeholder contains one piped-in object at a time, and we're executing the `ToLower()` method on each object. The new `String` objects produced by `ToLower()` are placed into the pipeline—and into the `$computers` variable.

You can do something similar with properties by using `Select-Object`. This example selects the `Length` property of each object you pipe to the cmdlet:

```
$computers | select-object length
```

This gives you

```
Length
------
     6
     7
     9
```

Because the property is numeric, PowerShell right-aligns the output.

16.4.4 Unrolling properties and methods in PowerShell

You *can* access properties and methods by using a variable that contains multiple objects:

```
$processes = Get-Process
$processes.Name
```

Under the hood, PowerShell "sees" that you're trying to access a property in that example. It also sees that the collection in `$processes` doesn't have a `Name` property—but the individual objects within the collection do. So it implicitly enumerates, or unrolls, the objects and grabs the `Name` property of each. This is equivalent to the following:

```
Get-Process | ForEach-Object { $_.Name }
```

And it is also equivalent to this:

```
Get-Process | Select-Object -ExpandProperty Name
```

The same thing works for methods:

```
$objects = Get-ChildItem ./*.txt -File
$objects.Refresh()
```

16.5 More tricks with double quotes

We have another cool technique you can use with double quotes, which is a somewhat conceptual extension of the variable-replacement trick. Suppose, for example, that you've put a bunch of processes into the `$processes` variable. Now you want to put only the name of the first one into a string:

```
$processes = Get-Process
$firstname = "$processes[0].name"
$firstname
```

This results in:

```
System.Diagnostics.Process System.Diagnostics.Process
System.Diagnostics.Process System.Diagnostics.Process
System.Diagnostics.Process System.Diagnostics.Process
System.Diagnostics.Process System.Diagnostics.Process
System.Diagnostics.Process System.Diagnostics.Process
System.Diagnostics.Process System.Diagnostics.Process
System.Diagnostics.Process System.Diagnostics.Process
System.Diagnostics.Process System.Diagnostics.Process
System.Diagnostics.Process System.Diagnostics.Process
System.Diagnostics.Process System.Diagnostics.Process
System.Diagnostics.Process System.Diagnostics.Process
System.Diagnostics.Process System.Diagnostics.Process
System.Diagnostics.Process System.Diagnostics.Process
System.Diagnostics.Process System.Diagnostics.Process
System.Diagnostics.Process System.Diagnostics.Process
System.Diagnostics.Process System.Diagnostics.Process
System.Diagnostics.Process System.Diagnostics.Process
System.Diagnostics.Process System.Diagnostics.Process
System.Diagnostics.Process System.Diagnostics.Process[0].name
```

Err, oops. The `[` immediately after `$processes` in the example isn't normally a legal character in a variable name, which causes PowerShell to try to replace `$processes`. Doing this jams the name of every service into your string. The `[0].name` part isn't replaced at all. The solution is to put all of that into an expression:

```
$processes = Get-Process | where-object {$_.Name}
$firstname = "The first name is $($processes[0].name)"
$firstname
```

This results in

```
The first name is AccountProfileR
```

Everything within `$()` is evaluated as a normal PowerShell command, and the result is placed into the string, replacing anything that's already there. Again, this works only in double quotes. This `$()` construct is called a *subexpression*.

We have another cool trick you can do in PowerShell. Sometimes you'll want to put something more complicated into a variable and then display that variable's contents within quotation marks. In PowerShell, the shell is smart enough to enumerate all of the objects in a collection even when you refer to a single property or method, provided that all of the objects in the collection are of the same type. For example, we'll retrieve a list of processes and put them into the `$processes` variable, and then include only the process names in double quotes:

```
$processes = Get-Process | where-object {$_.Name}
$var = "Process names are $processes.name"
$var
```

This results in

```
Process names are System.Diagnostics.Process (AccountProfileR)
     System.Diagnostics.Process (accountsd) System.Diagnostics.Process
     (adprivacyd) System.Diagnostics.Process (AdvertisingExte)
     System.Diagnostics.Process (AirPlayUIAgent) System.Diagnostics.Process
     (akd) System.Diagnostics.Process (AMPArtworkAgent)
     System.Diagnostics.Process (AMPDeviceDiscov) System.Diagnostics.Process
     (AMPLibraryAgent) System.Diagnostics.Process (amsaccountsd)
     System.Diagnostics.Process (APFSUserAgent) System.Diagnostics.Process
     (AppleSpell) System.Diagnostics.Process (AppSSOAgent)
     System.Diagnostics.Process (appstoreagent) System.Diagnostics.Process
     (askpermissiond) System.Diagnostics.Process (AssetCacheLocat)
     System.Diagnostics.Process (assistantd) System.Diagnostics.Process
     (atsd) System.Diagnostics.Process (AudioComponentR)
     System.Diagnostics.Process (backgroundtaskm) System.Diagnostics.Process
     (bird)
```

We truncated the preceding output to save space, but we hope you get the idea. Obviously, this might not be the exact output you're looking for, but between this technique and the subexpressions technique we showed you earlier in this section, you should be able to get exactly what you want.

16.6 Declaring a variable's type

So far, we've put objects into variables and let PowerShell figure out what types of objects we were using. PowerShell doesn't care what kind of objects you put into the box. But you might care.

For example, suppose you have a variable that you expect to contain a number. You plan to do some arithmetic with that number, and you ask a user to input that number. Let's look at an example, which you can type directly into the command line:

```
PS > $number = Read-Host "Enter a number"
Enter a number: 100
PS > $number = $number * 10
PS > $number
100100100100100100100100100100
```

> **TRY IT NOW** We haven't shown you `Read-Host` yet—we're saving it for the next chapter—but its operation should be obvious if you follow along with this example.

What the heck? How can `100` multiplied by `10` be `100100100100100100100100100100`? What crazy new math is that?

If you're sharp-eyed, you may have spotted what's happening. PowerShell doesn't treat our input as a number; it treats it as a string. Instead of multiplying `100` by `10`, PowerShell *duplicated the string* `"100"` *10 times*. The result, then, is the string `100`, listed 10 times in a row. Oops.

We can verify that the shell is in fact treating the input as a string:

```
PS > $number = Read-Host "Enter a number"
Enter a number: 100
PS > $number | Get-Member
   TypeName: System.String
Name             MemberType            Definition
----             ----------            ----------
Clone            Method                System.Object Clone()
CompareTo        Method                int CompareTo(System.Object valu...
Contains         Method                bool Contains(string value)
```

Yep, piping `$number` to `Get-Member` confirms that the shell sees it as a `System.String`, not a `System.Int32`. There are a couple of ways to deal with this problem, and we'll show you the easiest one.

First, we tell the shell that the `$number` variable should contain an integer, which will force the shell to try to convert any input to a real number. We do that in the following example by specifying the desired data type, `int`, in square brackets immediately prior to the variable's first use:

```
PS > [int]$number = Read-Host "Enter a number" <---- Forces the variable to [int]
Enter a number: 100
PS > $number | Get-Member
   TypeName: System.Int32          <---- Confirms that the variable is Int32
Name        MemberType Definition
----        ---------- ----------
CompareTo   Method     int CompareTo(System.Object value), int CompareT...
Equals      Method     bool Equals(System.Object obj), bool Equals(int ...
GetHashCode Method     int GetHashCode()
GetType     Method     type GetType()
GetTypeCode Method     System.TypeCode GetTypeCode()
```

```
ToString    Method     string ToString(), string ToString(string format...
PS > $number = $number * 10
PS > $number
1000        <-- The variable was treated as a number.
```

In this example, we use `[int]` to force `$number` to contain only integers. After entering our input, we pipe `$number` to `Get-Member` to confirm that it is indeed an integer and not a string. At the end, you can see that the variable is treated as a number, and multiplication takes place.

Another benefit to using this technique is that the shell will throw an error if it can't convert the input into a number, because `$number` is capable of storing only integers:

```
PS > [int]$number = Read-Host "Enter a number"
Enter a number: Hello
MetadataError: Cannot convert value "Hello" to type "System.Int32". Error:
➥ "Input string was not in a correct format."
```

This is a great example of how to prevent problems down the line, because you're assured that `$number` will contain the exact type of data you expect it to.

You can use many object types in place of `[int]`, but the following list includes some of the ones you'll most commonly use:

- `[int]`—Integer numbers
- `[single]` *and* `[double]`—Single-precision and double-precision floating numbers (numbers with a decimal portion)
- `[string]`—A string of characters
- `[char]`—Exactly one character (e.g., `[char]$c = 'X'`)
- `[xml]`—An XML document; whatever string you assign to this will be parsed to make sure it contains valid XML markup (e.g., `[xml]$doc = Get-Content MyXML.xml`)

Specifying an object type for a variable is a great way to prevent certain tricky logic errors in more-complex scripts. As the following example shows, once you specify the object type, PowerShell enforces it until you explicitly retype the variable:

```
                          <-- Declares $x as an integer
PS > [int]$x = 5
PS > $x = 'Hello'         <-- Creates an error by putting a string into $x
MetadataError: Cannot convert value "Hello" to type "System.Int32".
➥ Error: "Input string was not in a correct format."
PS > [string]$x = 'Hello' <-- Retypes $x as a string
PS > $x | Get-Member
   TypeName: System.String                    <-- Confirms a new type of $x

Name             MemberType            Definition
----             ----------            ----------
Clone            Method                System.Object Clone()
CompareTo        Method                int CompareTo(System.Object valu...
```

You can see that we start by declaring `$x` as an integer and placing an integer into it. When we try to put a string into it, PowerShell throws an error because it can't convert that particular string into a number. Later we retype `$x` as a string, and we're able to put a string into it. We confirm that by piping the variable to `Get-Member` and checking its type name.

16.7 Commands for working with variables

We've started to use variables at this point, without formally declaring our intention to do so. PowerShell doesn't require advanced variable declaration, and you can't force it to make a declaration. (Some folks may be looking for something like `Option Explicit` and will be disappointed; PowerShell has something called `Set-StrictMode`, but it isn't exactly the same thing.) But the shell does include the following commands for working with variables:

- `New-Variable`
- `Set-Variable`
- `Remove-Variable`
- `Get-Variable`
- `Clear-Variable`

You don't need to use any of these except perhaps `Remove-Variable`, which is useful for permanently deleting a variable (you can also use the `Remove-Item` command within the `VARIABLE:` drive to delete a variable). You can perform every other function—creating new variables, reading variables, and setting variables—by using the ad hoc syntax we've used up to this point in the chapter. Using these cmdlets offers no specific advantages, in most cases, as you are forcing variable assignment until you run your script. This can be problematic for tools that provide completions as you type, such as Visual Studio Code. These complications will be more accurate if you use the normal assignment operators, because PowerShell can look at your script and predict the data style of the variable's value.

If you do decide to use these cmdlets, you give your variable name to the cmdlets' `-name` parameters. This is *only the variable name*—it doesn't include the dollar sign. The one time you might want to use one of these cmdlets is when working with something called an *out-of-scope* variable. Messing with out-of-scope variables is a poor practice, and we don't cover them (or much more on scope) in this book, but you can run `help about_scope` in the shell to learn more.

16.8 Variable best practices

We've mentioned most of these practices already, but this is a good time to review them quickly:

- Keep variable names meaningful but succinct. Whereas `$computername` is a great variable name because it's clear and concise, `$c` is a poor name because what it contains isn't clear. The variable name `$computer_to_query_for_data`

is a bit long for our taste. Sure, it's meaningful, but do you want to type that over and over?

- Don't use spaces in variable names. We know you can, but it's ugly syntax.
- If a variable contains only one kind of object, declare that when you first use the variable. This can help prevent confusing logic errors. Suppose you're working in a commercial script development environment (such as Visual Studio Code). In that case, the editor software can provide code-hinting features when you tell it the type of object that a variable will contain.

16.9 Common points of confusion

The biggest single point of confusion we see new students struggle with is the variable name. We hope we've done an excellent job explaining it in this chapter, but always remember that the dollar sign *isn't part of the variable's name.* It's a cue to the shell that you want to access the *contents* of a variable; what follows the dollar sign is taken as the variable's name. The shell has two parsing rules that let it capture the variable name:

- If the character immediately after the dollar sign is a letter, number, or underscore, the variable name consists of all the characters following the dollar sign, up to the next white space (which might be a space, tab, or carriage return).
- If the character immediately after the dollar sign is an opening curly brace `{`, the variable name consists of everything after that curly brace up to, but not including, the closing curly brace `}`.

16.10 Lab

1. Create a background job that gets all processes that start with `pwsh` from two computers (use `localhost` twice if you have only one computer to experiment with).
2. When the job finishes running, receive the results of the job into a variable.
3. Display the contents of that variable.
4. Export the variable's contents to a CLIXML file.
5. Get a list of all the services currently running on your local machine and save it in a variable `$processes`.
6. Replace `$processes` with just the bits and print spooler service.
7. Display the contents of `$processes`.
8. Export `$processes` to a CSV file.

16.11 Lab answers

```
1 Invoke-Command {Get-Process pwsh} -computername
  localhost,$env:computername -asjob
2 $results = Receive-Job 4 -Keep
3 $results
4 $results | Export-CliXml processes.xml
5 $processes = get-service
```

```
6 $processes = get-service -name bits,spooler
7 $processes
8 $processes | export-csv -path c:\services.csv
```

16.12 Further exploration

Take a few moments and skim through some of the previous chapters in this book. Given that variables are primarily designed to store something you might use more than once, can you find a use for variables in our topics in previous chapters?

For example, in chapter 13 you learned to create connections to remote computers. In that chapter, you created, used, and closed a connection more or less in one step. Wouldn't it be useful to create the connection, store it in a variable, and use it for several commands? That's only one instance of where variables can come in handy (and we'll show you how to do that in chapter 20). See if you can find any more examples.