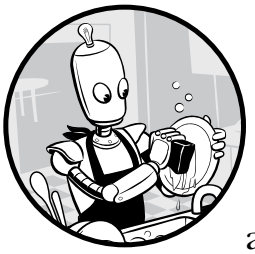
2

BASIC POWERSHELL CONCEPTS



This chapter covers four basic concepts in PowerShell: variables, data types, objects, and data structures. These concepts are fundamental to just about every common programming language, but there's something that makes PowerShell distinctive: everything in PowerShell is an object.

This may not mean much to you now, but keep it in mind as you move through the rest of this chapter. By the end of the chapter, you should have an idea of just how significant this is.

Variables

A *variable* is a place to store *values*. You can think of a variable as a digital box. When you want to use a value multiple times, for example, you can put it in a box. Then, instead of typing the same number over and over in your code, you can put it in a variable and call that variable whenever you need the value. But as you might have guessed from the name, the real power of

a variable is that it can change: you can add stuff to a box, swap what's in the box with something else, or take out whatever's in there and show it off for a bit before putting it back.

As you'll see later in the book, this variability lets you build code that can handle a general situation, as opposed to being tailored to one specific scenario. This section covers the basic ways to use a variable.

Displaying and Changing a Variable

All variables in PowerShell start with a dollar sign (`$`), which indicates to PowerShell that you are calling a variable and not a cmdlet, function, script file, or executable file. For example, if you want to display the value of the `MaximumHistoryCount` variable, you have to prepend it with a dollar sign and call it, as in Listing 2-1.

```
PS> $MaximumHistoryCount
4096
```

Listing 2-1: Calling the `$MaximumHistoryCount` variable

The `$MaximumHistoryCount` variable is a built-in variable that determines the maximum number of commands PowerShell saves in its command history; the default is 4096 commands.

You can change a variable's value by entering the variable name—starting with a dollar sign—and then using an equal sign (`=`) and the new value, as in Listing 2-2.

```
PS> $MaximumHistoryCount = 200
PS> $MaximumHistoryCount
200
```

Listing 2-2: Changing the `$MaximumHistoryCount` variable's value

Here you've changed the `$MaximumHistoryCount` variable's value to `200`, meaning PowerShell will save only the previous 200 commands in its command history.

Listings 2-1 and 2-2 use a variable that already exists. Variables in PowerShell come in two broad classes: *user-defined variables*, which are created by the user, and *automatic variables*, which already exist in PowerShell. Let's look at user-defined variables first.

User-Defined Variables

A variable needs to exist before you can use it. Try typing **`$color`** into your PowerShell console, as shown in Listing 2-3.

```
PS> $color
The variable '$color' cannot be retrieved because it has not been set.

At line:1 char:1
+ $color
+ ~~~~
```

```
    + CategoryInfo          : InvalidOperation: (color:String) [], RuntimeException
    + FullyQualifiedErrorId : VariableIsUndefined
```

Listing 2-3: Entering an undefined variable results in an error.

TURNING ON STRICT MODE

If you didn't get the error in Listing 2-3, and your console shows no output, try running the following command to turn on strict mode:

```
PS> Set-StrictMode -Version Latest
```

Turning on strict mode tells PowerShell to throw errors when you violate good coding practices. For example, strict mode forces PowerShell to return an error when you reference an object property that doesn't exist or an undefined variable. It's considered best practice to turn on this mode when writing scripts, as it forces you to write cleaner, more predictable code. When simply running interactive code from the PowerShell console, this setting is typically not used. For more information about strict mode, run `Get Help Set-StrictMode Examples`.

In Listing 2-3, you tried to refer to the `$color` variable before it even existed, which resulted in an error. To create a variable, you need to *declare* it—say that it exists—and then *assign* a value to it (or *initialize* it). You can do these at the same time, as in Listing 2-4, which creates a variable `$color` that contains the value `blue`. You can assign a value to a variable by using the same technique you used to change the value of `$MaximumHistoryCount`—by entering the variable name, followed by the equal sign, and then the value.

```
PS> $color = 'blue'
```

Listing 2-4: Creating a `color` variable with a value of `blue`

Once you've created the variable and assigned it a value, you can reference it by typing the variable name in the console (Listing 2-5).

```
PS> $color
blue
```

Listing 2-5: Checking the value of a variable

The value of a variable won't change unless something, or someone, explicitly changes it. You can call the `$color` variable any number of times, and it will return the value `blue` each time until the variable is redefined.

When you use the equal sign to define a variable (Listing 2-4), you're doing the same thing you'd do with the `Set-Variable` command. Likewise, when you type a variable into the console, and it prints out the value, as in

Listing 2-5, you're doing the same thing you'd do with the `Get-Variable` command. Listing 2-6 recreates Listings 2-4 and 2-5 by using these commands.

```
PS> Set-Variable -Name color -Value blue

PS> Get-Variable -Name color

Name                           Value
----                           -----
color                          blue
```

Listing 2-6: Creating a variable and displaying its value with the `Set-Variable` and `Get-Variable` commands

You can also use `Get-Variable` to return all available variables (as shown in Listing 2-7).

```
PS> Get-Variable

Name                           Value
----                           -----
$                              Get-PSDrive
?                              True
^                              Get-PSDrive
args                           {}
color                          blue
--snip--
```

Listing 2-7: Using `Get-Variable` to return all the variables.

This command will list all the variables currently in memory, but notice that there are some you haven't defined. You'll look at this type of variable in the next section.

Automatic Variables

Earlier I introduced automatic variables, the premade variables that PowerShell itself uses. Although PowerShell allows you to change some of these variables, as you did in Listing 2-2, I typically advise against it because unexpected consequences can arise. In general, you should treat automatic variables as *read-only*. (Now might be a good time to change `$MaximumHistoryCount` back to 4096!)

This section covers a few of the automatic variables that you're likely to use: the `$null` variable, `$LASTEXITCODE`, and the preference variables.

The $null Variable

The `$null` variable is a strange one: it represents nothing. Assigning `$null` to a variable allows you to create that variable but not assign a real value to it, as in Listing 2-8.

```
PS> $foo = $null
PS> $foo
PS> $bar
The variable '$bar' cannot be retrieved because it has not been set.
At line:1 char:1
+ $bar
+ ~~~~
    + CategoryInfo          : InvalidOperation: (bar:String) [], RuntimeException
    + FullyQualifiedErrorId : VariableIsUndefined
```

Listing 2-8: Assigning variables to `$null`

Here, you assign `$null` to the `$foo` variable. Then, when you call `$foo`, nothing is displayed, but no errors occur because PowerShell recognizes the variable.

You can see which variables PowerShell recognizes by passing parameters to the `Get-Variable` command. You can see in Listing 2-9 that PowerShell knows that the `$foo` variable exists but does not recognize the `$bar` variable.

```
PS> Get-Variable -Name foo

Name                           Value
----                           -----
foo


PS> Get-Variable -Name bar
Get-Variable : Cannot find a variable with the name 'bar'.
At line:1 char:1
+ Get-Variable -Name bar
+ ~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (bar:String) [Get-Variable], ItemNotFoundException
    + FullyQualifiedErrorId : VariableNotFound,Microsoft.PowerShell.Commands.GetVariableCommand
```

Listing 2-9: Using `Get-Variable` to find variables

You may be wondering why we bother defining anything as `$null`. But `$null` is surprisingly useful. For example, as you'll see later in this chapter, you often give a variable a value as a response to something else, like the output of a certain function. If you check that variable, and see that its value is still `$null`, you'll know that something went wrong in the function and can act accordingly.

The LASTEXITCODE Variable

Another commonly used automatic variable is `$LASTEXITCODE`. PowerShell allows you to invoke external executable applications like the old-school *ping.exe*, which pings a website to get a response. When external applications finish running, they finish with an *exit code*, or *return code*, that

indicates a message. Typically, a 0 indicates success, and anything else means either a failure or another anomaly. For *ping.exe*, a 0 indicates it was able to successfully ping a node, and a 1 indicates it could not.

When *ping.exe* runs, as in Listing 2-10, you'll see the expected output but not an exit code. That's because the exit code is hidden inside `$LASTEXITCODE`. The value of `$LASTEXITCODE` is always the exit code of the last application that was executed. Listing 2-10 pings *google.com*, returns its exit code, and then pings a nonexistent domain and returns its exit code.

```
PS> ping.exe -n 1 dfdfdfdfd.com

Pinging dfdfdfdfd.com [14.63.216.242] with 32 bytes of data:
Request timed out.

Ping statistics for 14.63.216.242:
    Packets: Sent = 1, Received = 0, Lost = 1 (100% loss),
PS> $LASTEXITCODE
1
PS> ping.exe -n 1 google.com

Pinging google.com [2607:f8b0:4004:80c::200e] with 32 bytes of data:
Reply from 2607:f8b0:4004:80c::200e: time=47ms

Ping statistics for 2607:f8b0:4004:80c::200e:
    Packets: Sent = 1, Received = 1, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 47ms, Maximum = 47ms, Average = 47ms
PS> $LASTEXITCODE
0
```

Listing 2-10: Using ping.exe *to demonstrate the `$LASTEXITCODE` variable*

The `$LASTEXITCODE` is 0 when you ping *google.com* but has a value of 1 when you ping the bogus domain name *dfdfdfdfd.com*.

The Preference Variables

PowerShell has a type of automatic variable referred to as *preference variables*. These variables control the default behavior of various output streams: `Error`, `Warning`, `Verbose`, `Debug`, and `Information`.

You can find a list of all of the preference variables by running `Get-Variable` and filtering for all variables ending in *Preference*, as shown here:

```
PS> Get-Variable -Name *Preference

Name                           Value
----                           -----
ConfirmPreference              High
DebugPreference                SilentlyContinue
ErrorActionPreference          Continue
InformationPreference          SilentlyContinue
ProgressPreference             Continue
```

```
VerbosePreference               SilentlyContinue
WarningPreference               Continue
WhatIfPreference                False
```

These variables can be used to configure the various types of output PowerShell can return. For example, if you've ever made a mistake and seen that ugly red text, you've seen the `Error` output stream. Run the following command to generate an error message:

```
PS> Get-Variable -Name 'doesnotexist'
Get-Variable : Cannot find a variable with the name 'doesnotexist'.
At line:1 char:1
+ Get-Variable -Name 'doesnotexist'
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (doesnotexist:String) [Get-Variable],
                              ItemNotFoundException
    + FullyQualifiedErrorId : VariableNotFound,Microsoft.PowerShell.Commands.GetVariableCommand
```

You should have gotten a similar error message, as this is the default behavior for the `Error` stream. If for whatever reason you didn't want to be bothered by this error text, and would rather nothing happen, you could redefine the `$ErrorActionPreference` variable to `SilentlyContinue` or `Ignore`, either of which will tell PowerShell not to output any error text:

```
PS> $ErrorActionPreference = 'SilentlyContinue'
PS> Get-Variable -Name 'doesnotexist'
PS>
```

As you can see, no error text is output. Ignoring error output is generally considered bad practice, so change the value of `$ErrorActionPreference` back to `Continue` before proceeding. For more information on preference variables, check out the `about_help` content by running **`Get-Help about_Preference_Variables`**.

Data Types

PowerShell variables come in a variety of forms, or *types*. All the details of PowerShell's data types are beyond the scope of this chapter. What you need to know is that PowerShell has several data types—including bools, strings, and integers—and you can change a variable's data type without errors. The following code should run with no errors:

```
PS> $foo = 1
PS> $foo = 'one'
PS> $foo = $true
```

This is because PowerShell can figure out data types based on the values you provide it. What's happening under the hood is a little too complicated for this book, but it's important you understand the basic types and how they interact.

Boolean Values

Just about every programming language uses *booleans*, which have a true or false value (1 or 0). Booleans are used to represent binary conditions, like a light switch being on or off. In PowerShell, booleans are called *bools*, and the two boolean values are represented by the automatic variables `$true` and `$false`. These automatic variables are hardcoded into PowerShell and can't be changed. Listing 2-11 shows how to set a variable to be `$true` or `$false`.

```
PS> $isOn = $true
PS> $isOn
True
```

Listing 2-11: Creating a bool variable

You'll see a lot more of bools in Chapter 4.

Integers and Floating Points

You can represent numbers in PowerShell in two main ways: via integer or floating-point data types.

Integer types

Integer data types hold only whole numbers and will round any decimal input to the nearest integer. Integer data types come in *signed* and *unsigned* types. Signed data types can store both positive and negative numbers; unsigned data types store values with no sign.

By default, PowerShell stores integers by using the 32-bit signed `Int32` type. The bit count determines how big (or small) a number the variable can hold; in this case, anything in the range –2,147,483,648 to 2,147,483,647. For numbers outside that range, you can use the 64-bit signed `Int64` type, which has a range of –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

Listing 2-12 shows an example of how PowerShell handles `Int32` types.

```
❶ PS> $num = 1
  PS> $num
  1
❷ PS> $num.GetType().name
  Int32
❸ PS> $num = 1.5
  PS> $num.GetType().name
  Double
❹ PS> [Int32]$num
  2
```

Listing 2-12: Using an `Int` type to store different values

Let's walk through each of these steps. Don't worry about all the syntax; for now, focus on the output. First, you create a variable `$num` and give it the value of 1 ❶. Next, you check the type of `$num` ❷ and see that PowerShell interprets 1 as an `Int32`. You then change `$num` to hold a decimal value ❸

and check the type again and see that PowerShell has changed the type to `Double`. This is because PowerShell will change a variable's type depending on its value. But you can force PowerShell to treat a variable as a certain type by *casting* that variable, as you do at the end by using the `[Int32]` syntax in front of `$num` ❹. As you can see, when forced to treat 1.5 as an integer, PowerShell rounds it up to 2.

Now let's look at the `Double` type.

Floating-Point Types

The `Double` type belongs to the broader class of variables known as *floating-point* variables. Although they can be used to represent whole numbers, floating-point variables are most often used to represent decimals. The other main type of floating-point variable is `Float`. I won't go into the internal representation of the `Float` and `Double` types. What you need to know is that although `Float` and `Double` are capable of representing decimal numbers, these types can be imprecise, as shown in Listing 2-13.

```
PS> $num = 0.1234567910
PS> $num.GetType().name
Double
PS> $num + $num
0.2469135782
PS> [Float]$num + [Float]$num
0.246913582086563
```

Listing 2-13: Precision errors with floating-point types

As you can see, PowerShell uses the `Double` type by default. But notice what happens when you add `$num` to itself but cast both as a `Float`—you get a strange answer. Again, the reasons are beyond the scope of this book, but be aware that errors like this can happen when using `Float` and `Double`.

Strings

You've already seen this type of variable. When you defined the `$color` variable in Listing 2-4, you didn't just type `$color = blue`. Instead, you enclosed the value in single quotes, which indicates to PowerShell that the value is a series of letters, or a *string*. If you try to assign the `blue` value to `$color` without the quotes, PowerShell will return an error:

```
PS> $color = blue
blue : The term 'blue' is not recognized as the name of a cmdlet, function, script file, or
operable program. Check the spelling of the name, or if a path was included, verify that the
path is correct and try again.
At line:1 char:10
+ $color = blue
+          ~~~~
    + CategoryInfo          : ObjectNotFound: (blue:String) [], CommandNotFoundException
    + FullyQualifiedErrorId : CommandNotFoundException
```

Without quotes, PowerShell interprets `blue` as a command and tries to execute it. Because the command `blue` doesn't exist, PowerShell returns an error message that says so. To correctly define a string, you need to use quotes around your value.

Combining Strings and Variables

Strings aren't restricted to words; they can be phrases and sentences as well. For instance, you can assign `$sentence` this string:

```
PS> $sentence = "Today, you learned that PowerShell loves the color blue"
PS> $sentence
Today, you learned that PowerShell loves the color blue
```

But maybe you want to use this same sentence, but with the words *PowerShell* and *blue* as the values of variables. For instance, what if you have a variable called `$name`, another called `$language`, and another called `$color`? Listing 2-14 defines these variables by using other variables.

```
PS> $language = 'PowerShell'
PS> $color = 'blue'

PS> $sentence = "Today, you learned that $language loves the color $color"
PS> $sentence
Today, you learned that PowerShell loves the color blue
```

Listing 2-14: Inserting variables in strings

Notice the use of double quotes. Enclosing your sentence in single quotes doesn't achieve the intended result:

```
PS> 'Today, $name learned that $language loves the color $color'
Today, $name learned that $language loves the color $color
```

This isn't just a weird bug. There's an important difference between single and double quotes in PowerShell.

Using Double vs. Single Quotes

When you're assigning a variable a simple string, you can use single or double quotes, as shown in Listing 2-15.

```
PS> $color = "yellow"
PS> $color
yellow
PS> $color = 'red'
PS> $color
red
PS> $color = ''
PS> $color
```

```
PS> $color = "blue"
PS> $color
blue
```

Listing 2-15: Changing variable values by using single and double quotes

As you can see, it doesn't matter which quotes you use to define a simple string. So why did it matter when you had variables in your string? The answer has to do with *variable interpolation*, or *variable expansion*. Normally, when you enter `$color` by itself into the console and hit ENTER, PowerShell *interpolates*, or *expands*, that variable. These are fancy terms that mean PowerShell is reading the value inside a variable, or opening the box so you can see inside. When you use double quotes to call a variable, the same thing happens: the variable is expanded, as you can see in Listing 2-16.

```
PS> "$color"
blue
PS> '$color'
$color
```

Listing 2-16: Variable behavior inside a string

But notice what happens when you use single quotes: the console outputs the variable itself, not its value. Single quotes tell PowerShell that you mean *exactly* what you're typing, whether that's a word like *blue* or what looks like a variable called `$color`. To PowerShell, it doesn't matter. It won't look past the value in single quotes. So when you use a variable inside single quotes, PowerShell doesn't know to expand that variable's value. This is why you need to use double quotes when inserting variables into your strings.

There's much more to say about bools, integers, and strings. But for now, let's take a step back and look at something more general: objects.

Objects

In PowerShell, *everything* is an object. In technical terms, an *object* is an individual instance of a specific template, called a class. A *class* specifies the kinds of things an object will contain. An object's class determines its *methods*, or actions that can be taken on that object. In other words, the methods are all the things an object can do. For example, a list object might have a `sort()` method that, when called, will sort the list. Likewise, an object's class determines its *properties*, the object's variables. You can think of the properties as all the data about the object. In the case of the list object, you might have a `length` property that stores the number of elements in the list. Sometimes, a class will provide default values for the object's properties, but more often than not, these are values you will provide to the objects you work with.

But that's all very abstract. Let's consider an example: a car. The car starts out as a plan in the design phase. This plan, or template, defines how the car should look, what kind of engine it should have, what kind of chassis

it should have, and so on. The plan also lays out what the car will be able to do once it's complete—move forward, move in reverse, and open and close the sunroof. You can think of this plan as the car's class.

Each car is built from this class, and all of that particular car's properties and methods are added to it. One car might be blue, while the same model car might be red, and another car may have a different transmission. These attributes are the properties of a specific car object. Likewise, each of the cars will drive forward, drive in reverse, and have the same method to open and close the sunroof. These actions are the car's methods.

Now with that general understanding of how objects work, let's get our hands dirty and work with PowerShell.

Inspecting Properties

First, let's make a simple object so you can dissect it and uncover the various facets of a PowerShell object. Listing 2-17 creates a simple string object called `$color`.

```
PS> $color = 'red'
PS> $color
red
```

Listing 2-17: Creating a string object

Notice that when you call `$color`, you get only the variable's value. But typically, because they're objects, variables have more information than just their value. They also have properties.

To look at an object's properties, you'll use the `Select-Object` command and the `Property` parameter. You'll pass the `Property` an asterisk argument, as in Listing 2-18, to tell PowerShell to return everything it finds.

```
PS>  Select-Object -InputObject $color -Property *

Length
------
     3
```

Listing 2-18: Investigating object properties

As you can see, the `$color` string has only a single property, called `Length`.

You can directly reference the `Length` property by using *dot notation*: you use the name of the object, followed by a dot and the name of the property you want to access (see Listing 2-19).

```
PS> $color.Length
3
```

Listing 2-19: Using dot notation to check an object's property

Referencing objects like this will become second nature over time.

Using the Get-Member cmdlet

Using `Select-Object`, you discovered that the `$color` string has only a single property. But recall that objects sometimes have methods as well. To take a look at all the methods *and* properties that exist on this string object, you can use the `Get-Member` cmdlet (Listing 2-20); this cmdlet will be your best friend for a long time. It's an easy way to quickly list all of a particular object's properties and methods, collectively referred to as an object's *members*.

```
PS> Get-Member -InputObject $color

   TypeName: System.String

Name             MemberType           Definition
----             ----------           ----------
Clone            Method               System.Object Clone(), System.Object ICloneable.Clone()
CompareTo        Method               int CompareTo(System.Object value),
                                      int CompareTo(string strB), int IComparab...
Contains         Method               bool Contains(string value)
CopyTo           Method               void CopyTo(int sourceIndex, char[] destination,
                                      int destinationIndex, int co...
EndsWith         Method               bool EndsWith(string value),
                                      bool EndsWith(string value, System.StringCompari...
Equals           Method               bool Equals(System.Object obj),
                                      bool Equals(string value), bool Equals(string...
--snip--
Length           Property             int Length {get;}
```

Listing 2-20: Using `Get-Member` to investigate object properties and methods

Now we're talking! It turns out that your simple string object has quite a few methods associated with it. There are lots more to explore, but not all are shown here. The number of methods and properties an object will have depends on its parent class.

Calling Methods

You can reference methods with dot notation. However, unlike a property, a method will always end in a set of opening and closing parentheses and can take one or more parameters.

For example, suppose you want to remove a character in your `$color` variable. You can remove characters from a string by using the `Remove()` method. Let's isolate `$color`'s `Remove()` method with the code in Listing 2-21.

```
PS> Get-Member -InputObject $color -Name Remove
Name   MemberType Definition
----   ---------- ----------
Remove Method     string Remove(int startIndex, int count), string Remove(int startIndex)
```

Listing 2-21: Looking at a string's `Remove()` method

As you can see, there are two definitions. This means you can use the method in two ways: either with `startIndex` and the `count` parameter, or with just `startIndex`.

So to remove the second character in `$color`, you specify the place of the character where you'd like to start removing, which we call the *index*. Indexes start from 0, so the first letter has a starting place of 0, the second an index of 1, and so on. Along with an index, you can provide the number of characters you'd like to remove by using a comma to separate the parameter arguments, as in Listing 2-22.

```
PS> $color.Remove(1,1)
Rd
PS> $color
red
```

Listing 2-22: Calling methods

Using an index of 1, you've told PowerShell that you want to remove characters starting with the string's second character; the second argument tells PowerShell to remove just one character. So you get `Rd`. But notice that the `Remove()` method doesn't permanently change the value of a string variable. If you'd like to keep this change, you'd need to assign the output of the `Remove()` method to a variable, as shown in Listing 2-23.

```
PS> $newColor = $color.Remove(1,1)
PS> $newColor
Rd
```

Listing 2-23: Capturing output of the `Remove()` method on a string

NOTE *If you need to know whether a method returns an object (as `Remove()` does) or modifies an existing object, you can check its description. As you can see in Listing 2-21, `Remove()`'s definition has the word `string` in front of it; this means that the function returns a new string. Functions with the word `void` in front typically modify existing objects. Chapter 6 covers this topic in more depth.*

In these examples, you've used one of the simplest types of object, the string. In the next section, you'll take a look at some more complex objects.

Data Structures

A *data structure* is a way to organize multiple pieces of data. Like the data they organize, data structures in PowerShell are represented by objects stored in variables. They come in three main types: arrays, ArrayLists, and hashtables.

Arrays

So far, I've described a variable as a box. But if a simple variable (such as a `Float` type) is a single box, then an *array* is whole bunch of boxes taped together—a list of items represented by a single variable.

Often you'll need several related variables—say, a standard set of colors. Rather than storing each color as a separate string, and then referencing each of those individual variables, it's much more efficient to store all of those colors in a single data structure. This section will show you how to create, access, modify, and add to an array.

Defining Arrays

First, let's define a variable called `$colorPicker` and assign it an array that holds four colors as strings. To do this, you use the at sign (`@`) followed by the four strings (separated by commas) within parentheses, as in Listing 2-24.

```
PS> $colorPicker = @('blue','white','yellow','black')
PS> $colorPicker
blue
white
yellow
black
```

Listing 2-24: Creating an array

The `@` sign followed by an opening parenthesis and zero or more elements separated by a comma signals to PowerShell that you'd like to create an array.

Notice that after calling `$colorPicker`, PowerShell displays each of the array's elements on a new line. In the next section, you'll learn how to access each element individually.

Reading Array Elements

To access an element in an array, you use the name of the array followed by a pair of square brackets (`[]`) that contain the index of the element you want to access. As with string characters, you start numbering arrays at 0, so the first element is at index 0, the second at index 1, and so on. In PowerShell, using –1 as the index will return the final element.

Listing 2-25 accesses several elements in our `$colorPicker` array.

```
PS> $colorPicker[0]
blue
PS> $colorPicker[2]
yellow
PS> $colorPicker[3]
black
PS> $colorPicker[4]
Index was outside the bounds of the array.
At line:1 char:1
+ $colorPicker[4]
+ ~~~~~~~~~~~~~~~
    + CategoryInfo          : OperationStopped: (:) [], IndexOutOfRangeException
    + FullyQualifiedErrorId : System.IndexOutOfRangeException
```

Listing 2-25: Reading array elements

As you can see, if you try to specify an index number that doesn't exist in the array, PowerShell will return an error message.

To access multiple elements in an array at the same time, you can use the *range operator* (`..`) between two numbers. The range operator will make PowerShell return those two numbers and every number between them, like so:

```
PS> 1..3
1
2
3
```

To use the range operator to access multiple items in an array, you use a range for an index, as shown here:

```
PS> $colorPicker[1..3]
white
yellow
black
```

Now that you've seen how to access elements in an array, let's look at how to change them.

Modifying Elements in an Array

If you want to change an element in an array, you don't have to redefine the entire array. Instead, you can reference an item with its index and use the equal sign to assign a new value, as in Listing 2-26.

```
PS> $colorPicker[3]
black
PS> $colorPicker[3] = 'white'
PS> $colorPicker[3]
white
```

Listing 2-26: Modifying elements in an array

Make sure you double-check that the index number is correct by displaying the element to your console before you modify an element.

Adding Elements to an Array

You can add items to an array with the addition operator (+), as in Listing 2-27.

```
PS> $colorPicker = $colorPicker + 'orange'
PS> $colorPicker
blue
white
yellow
white
orange
```

Listing 2-27: Adding a single item to an array

Notice that you enter `$colorPicker` on both sides of the equal sign. This is because you are asking PowerShell to interpolate the `$colorPicker` variable and then add a new element.

The + method works, but there's a quicker, more readable way. You can use the plus and equal signs together to form `+=` (see Listing 2-28).

```
PS> $colorPicker += 'brown'
PS> $colorPicker
blue
white
yellow
white
orange
brown
```

Listing 2-28: Using the += shortcut to add an item to an array

The `+=` operator tells PowerShell to *add this item to the existing array*. This shortcut prevents you from having to type out the array name twice and is much more common than using the full syntax.

You can also add arrays to other arrays. Say you'd like to add the colors pink and cyan to your `$colorPicker` example. Listing 2-29 defines another array with just those two colors and adds them just as you did in Listing 2-28.

```
PS> $colorPicker += @('pink','cyan')
PS> $colorPicker
blue
white
yellow
white
orange
brown
pink
cyan
```

Listing 2-29: Adding multiple elements to an array at once

Adding multiple items at once can save you a lot of time, especially if you're creating an array with a large number of items. Note that PowerShell treats any comma-separated set of values as an array, and you don't explicitly need the `@` or parentheses.

Unfortunately, there is no equivalent of `+=` to remove an element from an array. Removing elements from an array is more complicated than you might think, and we won't cover it here. To understand why, read on!

ArrayLists

Something strange happens when you add to an array. Every time you add an element to an array, you're actually creating a new array from your old (interpolated) array and the new element. The same thing happens when you remove an element from an array: PowerShell destroys your old array and makes a new one. This is because arrays in PowerShell have a fixed size.

When you change them, you can't modify the size, so you have to create a new array. For small arrays like the ones we've been working with, you won't notice this happening. But when you begin to work with *huge* arrays, with tens or hundreds of thousands of elements, you'll see a big performance hit.

If you know you'll have to remove or add many elements to an array, I suggest you use a different data structure called an *ArrayList*. ArrayLists behave nearly identically to the typical PowerShell array, but with one crucial difference: they don't have a fixed size. They can dynamically adjust to added or removed elements, giving a much higher performance when working with large amounts of data.

Defining an ArrayList is exactly like defining an array, except that you need to cast it as an ArrayList. Listing 2-30 re-creates the color picker array but casts it as a `System.Collections.ArrayList` type.

```
PS> $colorPicker = [System.Collections.ArrayList]@('blue','white','yellow','black')
PS> $colorPicker
blue
white
yellow
black
```

Listing 2-30: Creating an ArrayList

As with an array, when you call an ArrayList, each item is displayed on a separate line.

Adding Elements to an ArrayList

To add or remove an element from an ArrayList without destroying it, you can use its methods. You can use the `Add()` and `Remove()` methods to add or remove items from an ArrayList. Listing 2-31 uses the `Add()` method and enters the new element within the method's parentheses.

```
PS> $colorPicker.Add('gray')
4
```

Listing 2-31: Adding a single item to an ArrayList

Notice the output: the number 4, which is the index of the new element you added. Typically, you won't use this number, so you can send the `Add()` method output to the `$null` variable to prevent it from outputting anything, as shown in Listing 2-32.

```
PS> $null = $colorPicker.Add('gray')
```

Listing 2-32: Sending output to $null

There are a few ways to negate output from PowerShell commands, but assigning output to `$null` gives the best performance, as the `$null` variable cannot be reassigned.

Removing Elements from an ArrayList

You can remove elements in a similar way, using the `Remove()` method. For example, if you want to remove the value `gray` from the ArrayList, enter the value within the method's parentheses, as in Listing 2-33.

```
PS> $colorPicker.Remove('gray')
```

Listing 2-33: Removing an item from an ArrayList

Notice that to remove an item, you don't have to know the index number. You can reference the element by its actual value—in this case, `gray`. If the array has multiple elements with the same value, PowerShell will remove the element closest to the start of the ArrayList.

It's hard to see the performance difference with small examples like these. But ArrayLists perform much better on large datasets than arrays. As with most programming choices, you'll need to analyze your specific situation to determine whether it makes more sense to use an array or an ArrayList. The rule of thumb is the larger the collection of items you're working with, the better off you'll be using an ArrayList. If you're working with small arrays of fewer than 100 elements or so, you'll notice little difference between an array and an ArrayList.

Hashtables

Arrays and ArrayLists are great when you need your data associated with only a position in a list. But sometimes you'll want something more direct: a way to correlate two pieces of data. For example, you might have a list of usernames you want to match to real names. In that case, you could use a *hashtable* (or *dictionary*), a PowerShell data structure that contains a list of *key-value pairs*. Instead of using a numeric index, you give PowerShell an input, called a *key*, and it returns the *value* associated with that key. So, in our example, you would index into the hashtable by using the username, and it would return that user's real name.

Listing 2-34 defines a hashtable, called `$users`, that holds information about three users.

```
PS> $users = @{
    abertram = 'Adam Bertram'
    raquelcer = 'Raquel Cerillo'
    zheng21 = 'Justin Zheng'
}
PS> $users
Name                           Value
----                           -----
abertram                       Adam Bertram
raquelcer                      Raquel Cerillo
zheng21                        Justin Zheng
```

Listing 2-34: Creating a hashtable

PowerShell will not let you define a hashtable with duplicate keys. Each key has to uniquely point to a single value, which can be an array or even another hashtable!

Reading Elements from Hashtables

To access a specific value in a hashtable, you use its key. There are two ways to do this. Say you want to find out the real name of the user `abertram`. You could use either of the two approaches shown in Listing 2-35.

```
PS> $users['abertram']
Adam Bertram
PS> $users.abertram
Adam Bertram
```

Listing 2-35: Accessing a hashtable's value

The two options have subtle differences, but for now, you can choose whichever method you prefer.

The second command in Listing 2-35 uses a property: `$users.abertram`. PowerShell will add each key to the object's properties. If you want to see all the keys and values a hashtable has, you can access the `Keys` and `Values` properties, as in Listing 2-36.

```
PS> $users.Keys
abertram
raquelcer
zheng21
PS> $users.Values
Adam Bertram
Raquel Cerillo
Justin Zheng
```

Listing 2-36: Reading hashtable keys and values

If you want to see *all* the properties of a hashtable (or any object), you can run this command:

```
PS> Select-Object -InputObject $yourobject -Property *
```

Adding and Modifying Hashtable Items

To add an element to a hashtable, you can use the `Add()` method or create a new index by using square brackets and an equal sign. Both ways are shown in Listing 2-37.

```
PS> $users.Add('natice', 'Natalie Ice')
PS> $users['phrigo'] = 'Phil Rigo'
```

Listing 2-37: Adding an item to a hashtable

Now your hashtable stores five users. But what happens if you need to change one of the values in your hashtable?

When you're modifying a hashtable, it's always a good idea to check that the key-value pair you want exists. To check whether a key already exists in a hashtable, you can use the `ContainsKey()` method, part of every hashtable created in PowerShell. When the hashtable contains the key, it will return `True`; otherwise, it will return `False`, as shown in Listing 2-38.

```
PS> $users.ContainsKey('johnnyq')
False
```

Listing 2-38: Checking items in a hashtable

Once you've confirmed the key is in the hashtable, you can modify its value by using a simple equal sign, as shown in Listing 2-39.

```
PS> $users['phrigo'] = 'Phoebe Rigo'
PS> $users['phrigo']
Phoebe Rigo
```

Listing 2-39: Modifying a hashtable value

As you've seen, you can add items to a hashtable in a couple of ways. As you'll see in the next section, there's only one way to remove an item from a hashtable.

Removing Items from a Hashtable

Like ArrayLists, hashtables have a `Remove()` method. Simply call it and pass in the key value of the item you want to remove, as in Listing 2-40.

```
PS> $users.Remove('natice')
```

Listing 2-40: Removing an item from a hashtable

One of your users should be gone, but you can call the hashtable to double-check. Remember that you can use the `Keys` property to remind yourself of any key name.

Creating Custom Objects

So far in this chapter, you've been making and using types of objects built into PowerShell. Most of the time, you can stick with these types and save yourself the work of creating your own. But sometimes you'll need to create a custom object with properties and methods that you define.

Listing 2-41 uses the `New-Object` cmdlet to define a new object with a `PSCustomObject` type.

```
PS> $myFirstCustomObject = New-Object -TypeName PSCustomObject
```

Listing 2-41: Creating a custom object by using `New-Object`

This example uses the `New-Object` command, but you could do the same thing by using an equal sign and a cast, as in Listing 2-42. You define a hashtable in which the keys are property names, and the values are property values, and then cast it as `PSCustomObject`.

```
PS> $myFirstCustomObject = [PSCustomObject]@{OSBuild = 'x'; OSVersion = 'y'}
```

Listing 2-42: Creating a custom object by using the `PSCustomObject` type accelerator

Notice that Listing 2-42 uses a semicolon (`;`) to separate the key and value definitions.

Once you have a custom object, you use it as you would any other object. Listing 2-43 passes our custom object to the `Get_Member` cmdlet to check that it is a `PSCustomObject` type.

```
PS> Get-Member  -InputObject $myFirstCustomObject


   TypeName: System.Management.Automation.PSCustomObject

Name        MemberType   Definition
----        ----------   ----------
Equals      Method       bool Equals(System.Object obj)
GetHashCode Method       int GetHashCode()
GetType     Method       type GetType()
ToString    Method       string ToString()
OSBuild     NoteProperty string OSBuild=OSBuild
OSVersion   NoteProperty string OSVersion=Version
```

Listing 2-43: Investigating properties and methods of a custom object

As you can see, your object already has some preexisting methods (for example, one that returns the object's type!), along with the properties you defined when you created the object in Listing 2-42.

Let's access those properties by using dot notation:

```
PS> $myFirstCustomObject.OSBuild
x
PS> $myFirstCustomObject.OSVersion
y
```

Looks good! You'll use `PSCustomObject` objects a lot throughout the rest of the book. They're powerful tools that let you create much more flexible code.

Summary

By now, you should have a general understanding of objects, variables, and data types. If you still don't understand these concepts, please reread this chapter. This is some of the most foundational stuff we'll be covering. A high-level understanding of these concepts will make the rest of this book much easier to understand.

The next chapter covers two ways to combine commands in PowerShell: the pipeline and scripts.