8 Objects: Data by another name

We're going to do something a little different in this chapter. PowerShell's use of objects can be one of its most confusing elements, but at the same time it's also one of the shell's most critical concepts, affecting everything you do in the shell. We've tried various explanations over the years, and we've settled on a couple that each work well for distinctly different audiences. If you have programming experience and are comfortable with the concept of objects, we want you to skip to section 8.2. If you don't have a programming background and haven't programmed or scripted with objects before, start with section 8.1 and read straight through the chapter.

8.1 What are objects?

Take a second to run `Get-Process` in PowerShell. You should see a table with several columns, but those columns barely scratch the surface of the wealth of information available about processes. Each process object also has a machine name, a main window handle, a maximum working set size, an exit code and time, processor affinity information, and a great deal more. You'll find more than 60 pieces of information associated with a process. Why does PowerShell show so few of them?

The simple fact is that *most* of the things PowerShell can access offer more information than will comfortably fit on the screen. When you run any command, such as `Get-Process`, `Get-AzVm`, or `Get-AzStorageBlob`, PowerShell constructs—entirely in memory—a table that contains all of the information about those items. For `Get-Process`, that table consists of something like 67 columns, with one row for each process that's running on your computer. Each column contains a bit of information, such as virtual memory, CPU utilization, process name, process ID, and so on. Then PowerShell looks to see whether you've specified which of those columns

you want to view. If you haven't, the shell looks up a configuration file provided by Microsoft and displays only those table columns that Microsoft thinks you want to see.

One way to see all of the columns is to use `ConvertTo-Html`:

```
Get-Process | ConvertTo-Html | Out-File processes.html
```

That cmdlet doesn't bother filtering the columns. Instead, it produces an HTML file that contains all of them. That's one way to see the entire table.

In addition to all of those columns of information, each table row has actions associated with it. Those actions include what the operating system can do to, or with, the process listed in that table row. For example, the operating system can close a process, kill it, refresh its information, or wait for the process to exit, among other things.

Anytime you run a command that produces output, that output takes the form of a table in memory. When you pipe output from one command to another, like this

```
Get-Process | ConvertTo-Html
```

the entire table is passed through the pipeline. The table isn't filtered down to a smaller number of columns until every command has run.

Now for some terminology changes. PowerShell doesn't refer to this in-memory table as a *table*. Instead, it uses these terms:

- *Object*—This is what we've been calling a *table row*. It represents a single thing, such as a single process or a single storage account.
- *Property*—This is what we called a *table column*. It represents one piece of information about an object, such as a process name, a process ID, or a VM's running status.
- *Method*—This is what we called an *action*. A method is related to a single object and makes that object do something—for example, killing a process or starting a VM.
- *Collection*—This is the entire set of objects, or what we've been calling a *table*.

Figure 8.1 Showing that the object (file) has multiple properties such as `Author` and `FileType`

If you find the following discussion on objects to be confusing, refer to this four-point list. Always imagine a *collection* of objects as being a big, in-memory table of information, with *properties* as the columns and individual *objects* as the rows (figure 8.1).

8.2 Understanding why PowerShell uses objects

One of the reasons that PowerShell uses objects to represent data is that, well, you have to represent data *somehow*, right? PowerShell could have stored that data in a format such as XML, or perhaps its creators could have decided to use plain-text tables. But they had specific reasons for not taking those routes.

The first reason is due to PowerShell's history of previously being Windows-only. Windows itself is an object-oriented operating system—or at least, most of the software that runs on Windows is object-oriented. Choosing to structure data as a set of objects is easy, because most of the operating system lends itself to those structures. As it turns out, we can apply that object-oriented mindset to other operating systems, and even other paradigms like the cloud and DevOps.

Another reason to use objects is that they ultimately make things easier on you and give you more power and flexibility. For the moment, let's pretend that PowerShell doesn't produce objects as the output of its commands. Instead, it produces simple text tables, which is what you probably thought it was doing in the first place. When you run a command such as `Get-Process`, you're getting formatted text as the output:

```
PS /Users/travis> Get-Process
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     39       5     1876       4340    52    11.33   1920 Code
     31       4      792       2260    22     0.00   2460 Code
     29       4      828       2284    41     0.25   3192 Code
    574      12     1864       3896    43     1.30    316 pwsh
    181      13     5892       6348    59     9.14    356 ShipIt
    306      29    13936      18312   139     4.36   1300 storeaccountd
    125      15     2528       6048    37     0.17   1756 WifiAgent
   5159    7329    85052      86436   118     1.80   1356 WifiProxy
```

What if you want to do something else with this information? Perhaps you want to make a change to all of the processes running `Code`. To do this, you have to filter the list a bit. On UNIX or Linux, you might try to use a command such as `grep` (which you *could* run in PowerShell, by the way!), telling it, "Look at this text list for me. Keep only those rows where columns 58–64 contain the characters `Code`. Delete all of the other rows." The resulting list contains only those processes you specified:

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     39       5     1876       4340    52    11.33   1920 Code
     31       4      792       2260    22     0.00   2460 Code
     29       4      828       2284    41     0.25   3192 Code
```

You then pipe that text to another command, perhaps telling it to extract the process ID from the list. "Go through this and get the characters from columns 52–56, but drop the first two (header) rows." The result might be this:

```
1920
2460
3192
```

Finally, you pipe *that* text to yet *another* command, asking it to kill the processes (or whatever else you were trying to do) represented by those ID numbers.

This is exactly how IT professionals that use `bash` work. They spend a lot of time learning how to get better at parsing text; using tools such as `grep`, `awk`, and `sed`; and becoming proficient in the use of regular expressions. Going through this learning process makes it easier for them to define the text patterns they want their computer to look for. In the old days before PowerShell was cross-platform, UNIX and Linux IT professionals would rely on scripting languages like Perl and Python, which have more batteries included in terms of text parsing. But this text-based approach does present some problems:

- You can spend more time messing around with text than doing your real job.
- If the output of a command changes—say, moving the ProcessName column to the start of the table—then you have to rewrite all of your commands, because they're all dependent on things like column positions.
- You have to become proficient in languages and tools that parse text—not because your job involves parsing text, but because parsing text is a means to an end.
- Languages like Perl and Python are solid scripting languages . . . but are not shells as well.

PowerShell's use of objects helps to remove all of that text-manipulation overhead. Because objects work like tables in memory, you don't have to tell PowerShell which text column a piece of information is located at. Instead, you tell it the column name, and PowerShell knows exactly where to go to get that data. Regardless of how you arrange the final output on the screen or in a file, the in-memory table is always the same, so you never have to rewrite your commands because a column moved. You spend a lot less time on overhead tasks and more time focusing on what you want to accomplish.

True, you do have to learn a few syntax elements that let you properly instruct PowerShell, but you have to learn a *lot* less than if you were working in a purely text-based shell.

DON'T GET MAD None of the preceding is intended as a dig at Bash, Perl, or Python, by the way. Every tool has pros and cons. Python is a great general-purpose programming language that has even found its way into the machine learning and artificial intelligence space—but that's not why you're reading this book. You're looking for something to up your game as an IT professional, and PowerShell is the perfect tool for it.

8.3 Discovering objects: Get-Member

If objects are like a giant table in memory, and PowerShell shows you only a portion of that table on the screen, how can you see what else you have to work with? If you're thinking that you should use the `Get-Help` command, we're glad, because we've certainly been pushing that down your throat in the previous few chapters. But unfortunately, you'd be wrong.

The help system documents only background concepts (in the form of the about topics) and command syntax. To learn more about an object, you use a different

command: `Get-Member`. You should become comfortable using this command—so much so, that you start looking for a shorter way to type it. We'll give you that right now: the alias `gm`.

You can use `gm` after any cmdlet that normally produces output. For example, you already know that running `Get-Process` produces output on the screen. You can pipe it to `gm`:

```
Get-Process | gm
```

Whenever a cmdlet produces a collection of objects, as `Get-Process` does, the entire collection remains accessible until the end of the pipeline. It's not until every command has run that PowerShell filters the columns of information to be displayed and creates the final text output you see. Therefore, in the preceding example, `gm` has complete access to all of the process objects' properties and methods, because they haven't been filtered for display yet. `gm` looks at each object and constructs a list of the objects' properties and methods. It looks like this:

```
PS C:\> Get-Process | gm
   TypeName: System.Diagnostics.Process
Name                       MemberType     Definition
----                       ----------     ----------
Handles                    AliasProperty  Handles = Handlecount
Name                       AliasProperty  Name = ProcessName
NPM                        AliasProperty  NPM = NonpagedSystemMemo...
PM                         AliasProperty  PM = PagedMemorySize
VM                         AliasProperty  VM = VirtualMemorySize
WS                         AliasProperty  WS = WorkingSet
Disposed                   Event          System.EventHandler Disp...
ErrorDataReceived          Event          System.Diagnostics.DataR...
Exited                     Event          System.EventHandler Exit...
OutputDataReceived         Event          System.Diagnostics.DataR...
BeginErrorReadLine         Method         System.Void BeginErrorRe...
BeginOutputReadLine        Method         System.Void BeginOutputR...
CancelErrorRead            Method         System.Void CancelErrorR...
CancelOutputRead           Method         System.Void CancelOutput...
```

We've trimmed the preceding list because it's long, but hopefully you get the idea.

> **TRY IT NOW** Don't take our word for it. This is the perfect time to follow along and run the same commands we do, to see their complete output.

By the way, it may interest you to know that all of the properties, methods, and other things attached to an object are collectively called its *members*, as if the object itself were a country club and all of these properties and methods belonged to the club. That's where `Get-Member` takes its name from—it's getting a list of the objects' members. But remember, because the PowerShell convention is to use singular nouns, the cmdlet name is `Get-Member`, *not* `Get-Members`.

IMPORTANT It's easy to overlook, but pay attention to the first line of output from `Get-Member`. It's `TypeName`, which is the unique name assigned to that particular type of object. It may seem unimportant now—after all, who cares what it's named? But it's going to become crucial in the next chapter.

8.4 Using object attributes, or properties

When you examine the output of `gm`, you'll notice several kinds of properties:

- `ScriptProperty`
- `Property`
- `NoteProperty`
- `AliasProperty`

Above and beyond

Normally, objects in .NET—which is where all of PowerShell's objects come from—have only *properties*. PowerShell dynamically adds the other stuff: `ScriptProperty`, `NoteProperty`, `AliasProperty`, and so on. If you happen to look up an object type in Microsoft's documentation (you can plug the object's `TypeName` into your favorite search engine to find the docs.microsoft.com page), you won't see these extra properties.

PowerShell has an extensible type system (ETS) that's responsible for adding these last-minute properties. Why does it do this? In some cases, it's to make objects more consistent, such as adding a `Name` property to objects that natively have only something like `ProcessName` (that's what an `AliasProperty` is for). Sometimes it's to expose information that's deeply buried in the object (process objects have a few `ScriptProperties` that do this).

Once you're in PowerShell, these properties all behave the same way. But don't be surprised when they don't show up on the official documentation page: the shell adds these extras, often to make your life easier.

For your purposes, these properties are all the same. The only difference is in how the properties were originally created, but that's not something you need to worry about. To you, they're all properties, and you'll use them the same way.

A property always contains a value. For example, the value of a process object's `ID` property might be `1234`, and the `Name` property of that object might have a value of `Code`. Properties describe something about the object: its status, its ID, its name, and so on. In PowerShell, properties are often read-only, meaning you can't change the name of a service by assigning a new value to its `Name` property. But you can retrieve the name of a service by reading its `Name` property. We estimate that 90% of what you'll do in PowerShell will involve properties.

8.5 Using object actions, or methods

Many objects support one or more methods, which, as we mentioned earlier, are actions that you can direct the object to take. A process object has a `Kill` method, which terminates the process. Some methods require one or more input arguments that provide additional details for that particular action, but you won't be running into any of those this early in your PowerShell education. You may spend months or even years working with PowerShell and never need to execute a single object method. That's because many of those actions are also provided by cmdlets.

For example, if you need to terminate a process, you have three ways to do so. One way is to retrieve the object and then somehow execute its `Kill` method. Another way is to use a couple of cmdlets:

```
Get-Process -Name Code | Stop-Process
```

You can also accomplish that by using a single cmdlet:

```
Stop-Process -Name Code
```

Our focus in this book is entirely on using PowerShell cmdlets to accomplish tasks. They provide the easiest, most IT professional–centric, most task-focused way of accomplishing things. Using methods starts to edge into .NET programming, which can be more complicated and can require a lot more background information. For that reason, you'll rarely—if ever—see us execute an object method in this book. Our general philosophy at this point is, "If you can't do it with a cmdlet, go back and use the GUI." You won't feel that way for your entire career, we promise, but for now it's a good way to stay focused on the "PowerShell way" of doing things.

> **Above and beyond**
>
> You don't need to know about them at this stage in your PowerShell education, but in addition to properties and methods, objects can also have events. An *event* is an object's way of notifying you that something happened to it. A process object, for example, can trigger its `Exited` event when the process ends. You can attach your own commands to those events, so that, for example, an email is sent when a process exits. Working with events in this fashion is an advanced topic that's beyond the scope of this book.

8.6 Sorting objects

Most PowerShell cmdlets produce objects in a deterministic fashion, which means that they tend to produce objects in the same order every time you run the command. Both Azure VMs and processes, for example, are listed in alphabetical order by name. What if we want to change that?

Suppose we want to display a list of processes, with the biggest consumers of CPU at the top of the list and the smallest consumers at the bottom. We need to somehow

reorder that list of objects based on the CPU property. PowerShell provides a simple cmdlet, Sort-Object, that does exactly that:

```
Get-Process | Sort-Object -Property CPU
```

> **TRY IT NOW** We're hoping that you'll follow along and run the commands in this chapter. We aren't pasting the output into the book because these tables are long.

That command isn't exactly what we want. It does sort on CPU, but it does so in ascending order, with the largest values at the bottom of the list. Reading the help for Sort-Object, we see that it has a -Descending parameter that should reverse the sort order. We also notice that the -Property parameter is positional, so we don't need to type the parameter name.

We abbreviated -Descending to -desc, and we have the result we want. The -Property parameter accepts multiple values (which we're sure you saw in the help file, if you looked).

In the event that two processes are using the same amount of virtual memory, we want them sorted by process ID, and the following command accomplishes that:

```
Get-Process | Sort-Object CPU,ID -desc
```

As always, a comma-separated list is the way to pass multiple values to any parameter that supports them.

8.7 Selecting the properties you want

Another useful cmdlet is Select-Object. It accepts objects from the pipeline, and you can specify the properties that you want displayed. This enables you to access properties that are normally filtered out by PowerShell's configuration rules, or to trim down the list to a few properties that interest you. This can be useful when piping objects to ConvertTo-HTML, because that cmdlet usually builds a table containing every property. Compare the results of these two commands:

```
Get-Process | ConvertTo-HTML | Out-File test1.html
Get-Process | Select-Object -Property Name,ID,CPU,PM | ConvertTo-Html |
➥ Out-File test2.html
```

> **TRY IT NOW** Go ahead and run each of these commands separately, and then examine the resulting HTML files in a web browser to see the differences.

Look at the help for Select-Object (or you can use its alias, Select). The -Property parameter is positional, which means we could shorten that last command:

```
Get-Process | Select Name,ID,CPU,PM | ConvertTo-HTML | Out-File test3.html
```

Spend some time experimenting with `Select-Object`. Try variations of the following command, which allows the output to appear on the screen:

```
Get-Process | Select Name,ID,CPU,PM
```

Try adding and removing different process object properties from that list and reviewing the results. How many properties can you specify and still get a table as the output? How many properties force PowerShell to format the output as a list rather than as a table?

Above and beyond

`Select-Object` also has `-First` and `-Last` parameters, which let you keep a subset of the objects in the pipeline. For example, `Get-Process | Select -First 10` keeps the first 10 objects. There are no criteria involved, such as keeping certain processes; it's merely grabbing the first (or last) 10.

CAUTION People often get mixed up about two PowerShell commands: `Select-Object` and `Where-Object`, which you haven't seen yet. `Select-Object` is used to choose the properties (or columns) you want to see, and it can also select an arbitrary subset of output rows (using `-First` and `-Last`). `Where-Object` removes, or filters, objects out of the pipeline based on criteria you specify.

8.8 Objects until the end

The PowerShell pipeline always contains objects until the last command has been executed. At that time, PowerShell looks to see what objects are in the pipeline, and then looks at its various configuration files to see which properties to use to construct the onscreen display. It also decides whether that display will be a table or a list, based on internal rules and on its configuration files. (We'll explain more about those rules and configurations, and how you can modify them, in chapter 10.)

An important fact is that the pipeline can contain many kinds of objects over the course of a single command line. For the next few examples, we're going to take a single command line and physically type it so that only one command appears on a single line of text. That'll make it a bit easier to explain what we're talking about. Here's the first one:

```
Get-Process |
Sort-Object CPU -Descending |
Out-File c:\procs.txt
```

In this example, we start by running `Get-Process`, which puts process objects into the pipeline. The next command is `Sort-Object`. That doesn't change what's in the pipeline; it changes only the order of the objects, so at the end of `Sort-Object`, the pipeline

still contains processes. The last command is `Out-File`. Here, PowerShell has to produce output, so it takes whatever's in the pipeline—processes—and formats them according to its internal rule set. The results go into the specified file. Next up is a more complicated example:

```
Get-Process |
Sort-Object CPU -Descending |
Select-Object Name,ID,CPU
```

This starts off in the same way. `Get-Process` puts process objects into the pipeline. Those go to `Sort-Object`, which sorts them and puts the same process objects into the pipeline. But `Select-Object` works a bit differently. A process object always has the exact same members. In order to trim down the list of properties, `Select-Object` can't remove the properties you don't want, because the result wouldn't be a process object anymore. Instead, `Select-Object` creates a new kind of custom object called a PSObject. It copies over the properties you do want from the process, resulting in a custom object being placed into the pipeline.

> **TRY IT NOW** Try running this three-cmdlet command line, keeping in mind that you should type the whole thing on a single line. Notice how the output is different from the normal output of `Get-Process`?

When PowerShell sees that it's reached the end of the command line, it has to decide how to lay out the text output. Because there are no longer any process objects in the pipeline, PowerShell won't use the default rules and configurations that apply to process objects. Instead, it looks for rules and configurations for a PSObject, which is what the pipeline now contains. Microsoft doesn't provide any rules or configurations for PSObjects, because they're meant to be used for custom output. Instead, PowerShell takes its best guess and produces a table, on the theory that those three pieces of information probably will still fit in a table. The table isn't as nicely laid out as the normal output of `Get-Process`, though, because the shell lacks the additional configuration information needed to make a nicer-looking table.

You can use `gm` to see the objects that wind up in the pipeline. Remember, you can add `gm` after any cmdlet that produces output:

```
Get-Process | Sort-Object CPU -Descending | gm
Get-Process | Sort-Object CPU -Descending | Select Name,ID,CPU | gm
```

> **TRY IT NOW** Try running those two command lines separately, and notice the difference in the output.

Notice that, as part of the `gm` output, PowerShell shows you the type name for the object it sees in the pipeline. In the first case, that's a `System.Diagnostics.Process` object, but in the second case the pipeline contains a different kind of object. Those new *selected* objects contain only the three properties specified—`Name`, `ID`, and `CPU`—plus a couple of system-generated members.

Even `gm` produces objects and places them into the pipeline. After running `gm`, the pipeline no longer contains either process or the *selected* objects; it contains the type of object produced by `gm`: a `Microsoft.PowerShell.Commands.MemberDefinition`. You can prove that by piping the output of `gm` to `gm` itself:

```
Get-Process | gm | gm
```

> **TRY IT NOW** You'll definitely want to try this, and think hard about it to make sure it makes sense to you. You start with `Get-Process`, which puts process objects into the pipeline. Those go to `gm`, which analyzes them and produces its own `MemberDefinition` objects. Those are then piped to `gm`, which analyzes them and produces output that lists the members of each `MemberDefinition` object.

A key to mastering PowerShell is learning to keep track of the kind of object that's in the pipeline at any given point. While `gm` can help you do that, sitting back and verbally walking yourself through the command line is also a good exercise that can help clear up confusion.

8.9 Common points of confusion

Newcomers tend to make a few common mistakes as they get started with PowerShell. Most of these go away with a little experience, but we direct your attention to them with the following list, to give you a chance to catch yourself if you start heading down the wrong path.

- Remember that the PowerShell help files don't contain information on objects' properties. You'll need to pipe the objects to `gm` (`Get-Member`) to see a list of properties.
- Remember that you can add `gm` to the end of any pipeline that typically produces results. A command line such as `Get-Process -Name Code | Stop-Process` doesn't usually produce results, so tacking `| gm` onto the end won't produce anything either.
- Pay attention to neat typing. Put a space on either side of every pipeline character, because your command lines should read as `Get-Process | gm` and not `Get-Process|gm`. That spacebar key is extra large for a reason—use it.
- Remember that the pipeline can contain various types of objects at each step. Think about what type of object is in the pipeline, and focus on what the next command will do to that *type* of object.

8.10 Lab

> **NOTE** For this lab, you need any computer running PowerShell v7 or later.

This chapter has probably covered more, and more difficult, new concepts than any chapter to this point. We hope that you were able to make sense of it all and that these

exercises will help you cement what you've learned. The lab may be more challenging than previous labs, but we want you to start getting in the habit of figuring out which commands to use—and relying on `get-command` and help, rather than on us, to find the correct command. After all, that is what you'll be doing once you start working with PowerShell on the job and encountering all sorts of situations we don't address in the book. Some of these tasks draw on skills you've learned in previous chapters, to refresh your memory and keep you sharp:

1. Identify a cmdlet that produces a random number.
2. Identify a cmdlet that displays the current date and time.
3. What type of object does the cmdlet from task 2 produce? (What is the *TypeName* of the object produced by the cmdlet?)
4. Using the cmdlet from task 2 and `Select-Object`, display only the current day of the week in a table like the following (caution: the output will right-align, so make sure your PowerShell window doesn't have a horizontal scrollbar):

```
DayOfWeek
---------
   Monday
```

5. Identify a cmdlet that will show you all the times in a directory.
6. Using the cmdlet from task 5, display all the times in the directory of your choice. Then extend the expression to sort the list by the time the items were created and display only the filename(s) and the date created. Remember that the column headers shown in a command's default output aren't necessarily the real property names—you need to look up the real property names to be sure.
7. Repeat task 6, but this time sort the items by the last write time; then display the filename, creation time, and the last write time. Save this in a CSV file and an HTML file.

8.11 Lab answers

```
1 Get-Random
2 Get-Date
3 System.DateTime
4 Get-Date | select DayofWeek
5 Get-ChildItem
6 Get-ChildItem | Sort-Object CreationTime | Select-Object
  ➥ Name,CreationTime
7 Get-ChildItem | Sort-Object LastWritetime | Select-Object
  ➥ Name,LastWritetime,CreationTime | Export-CSV files.csv
  Get-ChildItem | Sort-Object LastWritetime | Select-Object
  ➥ Name,LastWritetime,CreationTime | Out-file files.html
```