17 Input and output

Up to this point in the book, we've primarily been relying on PowerShell's native ability to output tables and lists. As you start to combine commands into more complex scripts, you'll probably want to gain more precise control over what's displayed. You may also need to prompt a user for input. In this chapter, you'll learn how to collect that input and how to display whatever output you might desire.

We want to point out, however, that the contents of this chapter are useful only for scripts that interact with human eyeballs and fingertips. For scripts that run unattended, these aren't appropriate techniques, because there won't be a human being around to interact with.

17.1 Prompting for, and displaying, information

The way PowerShell displays and prompts for information depends on how it's being run. You see, PowerShell is built as a kind of under-the-hood engine.

What you interact with is called a *host application*. The command-line console you see when running the PowerShell executable in a terminal application is often called the *console host*. Another common host is called the *integrated host*, which is represented as the PowerShell Integrated Console supplied by the PowerShell extension for Visual Studio Code. Other non-Microsoft applications can host the shell's engine as well. In other words, you, as the user, interact with the hosting application, and it, in turn, passes your commands through to the engine. The hosting application displays the results that the engine produces.

NOTE Another well-known host is in the PowerShell worker for Azure Functions. Azure Functions is Microsoft Azure's serverless offering, which is fancy talk for a service that allows you to run an arbitrary PowerShell script in the cloud without managing the underlying environment that script is running in. This host is interesting—because it's run unattended, there's no interactive element of this host, unlike the console or integrated host.

Figure 17.1 illustrates the relationship between the engine and the various hosting applications. Each hosting application is responsible for physically displaying any output the engine produces and physically collecting any input the engine requests. That means PowerShell can display output and manage input in different ways.

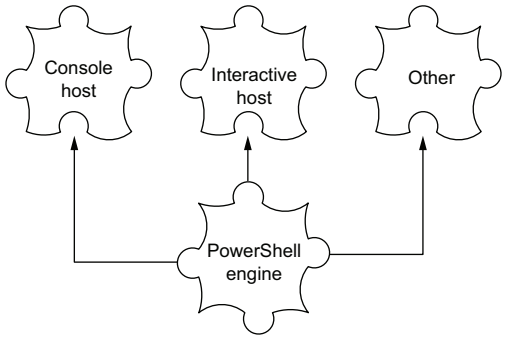

Figure 17.1 Various applications are capable of hosting the PowerShell engine.

We want to point out these differences because it can sometimes be confusing to newcomers. Why would one command behave one way in the command-line window but behave differently in, say, Azure Functions? It's because the hosting application determines the way in which you interact with the shell, not the PowerShell engine. The commands we're about to show you exhibit slightly different behavior depending on where you run them.

17.2 Read-Host

PowerShell's `Read-Host` cmdlet is designed to display a text prompt and then collect text input from the user. Because you saw us use this for the first time in the previous chapter, the syntax may seem familiar:

```
PS C:\> read-host "Enter a computer name"
Enter a computer name: SERVER-UBUNTU
SERVER-UBUNTU
```

This example highlights two important facts about the cmdlet:

- A colon is appended to the end of the line of text.
- Whatever the user types is returned as the result of the command (technically, it's placed into the pipeline, but more on that later).

You'll often capture the input into a variable, which looks like this:

```
PS C:\> $computername = read-host "Enter a computer name"
Enter a computer name: SERVER-UBUNTU
```

TRY IT NOW Time to start following along. At this point, you should have a valid computer name in the `$computername` variable. Don't use SERVER-UBUNTU unless that's the name of the computer you're working on.

17.3 Write-Host

Now that you can collect input, you'll want some way of displaying output. The `Write-Host` cmdlet is one way. It's not always the best way, but it's available to you, and it's important that you understand how it works.

As figure 17.2 illustrates, `Write-Host` runs in the pipeline like any other cmdlet, but it doesn't place anything into the pipeline. Instead, it does two things: writes a record into the "information stream" (don't worry, we'll cover this later!) and writes directly to the hosting application's screen.

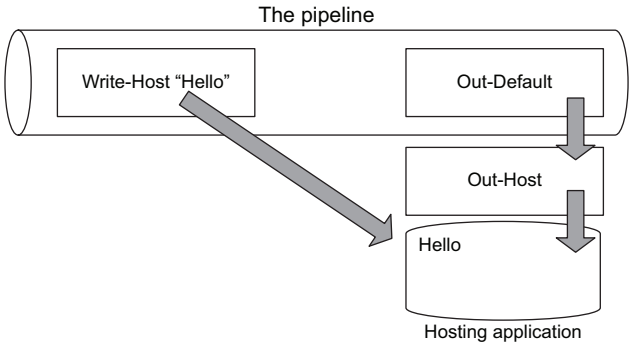

Figure 17.2 `Write-Host` bypasses the pipeline and writes directly to the hosting application's display.

Now, because `Write-Host` writes directly to the host app's screen, it's able to use alternate foreground and background colors through its `-ForegroundColor` and `-BackgroundColor` command-line parameters. You can see all the available colors by running `get-help -command write-host`.

TRY IT NOW Run `Get-Help Write-Host`. What colors are available for the `ForegroundColor` and `BackgroundColor` parameters? Now that we know what colors are available, let's have a little fun.

```
PS C:\> write-host "COLORFUL!" -Foreground yellow -BackgroundColor magenta
COLORFUL!
```

TRY IT NOW You'll want to run this command yourself to see the colorful results.

NOTE Not every application that hosts PowerShell supports alternate text colors, and not every application supports the full set of colors. When you attempt to set colors in such an application, it will usually ignore any colors it doesn't like or can't display. That's one reason we tend to avoid relying on special colors at all.

The `Write-Host` command has a bad reputation because in earlier version of PowerShell, it didn't do much. It acted as a mechanism to display information to the user via the console and didn't muddy any of the streams (yes, we know, we keep talking about these pesky things, and we will get to them, we promise). But starting in PowerShell 5, the `Write-Host` command was redesigned. It is now a wrapper for the `Write-Information` command, as it needed to be backward compatible. It still will output the text to your screen but will also put your text into the information stream so you can use it later. But `Write-Host` does have its limitations and may not always be the correct cmdlet for the job.

For example, you should never use `Write-Host` to manually format a table. You can find better ways to produce the output, using techniques that enable PowerShell itself to handle the formatting. We won't dive into those techniques in this book, because they belong more in the realm of heavy-duty scripting and tool making. However, you can check out *Learn PowerShell Scripting in a Month of Lunches* by Don Jones and Jeffery Hicks (Manning, 2017) for full coverage of those output techniques.

`Write-Host` is also not the best way to produce error messages, warnings, debugging messages, and so on—again, you can find more specific ways to do those things, and we'll cover those in this chapter. The only time you will really be using `Write-Host` is if you want to make a message on the screen with fancy colors in it.

NOTE We often see people using `Write-Host` to display what we call "warm and fuzzy" messages—things like "Now connecting to SERVER2," and "Testing for folder." We suggest you use the `Write-Verbose` messages instead. The reason we suggest this is because the output being sent the `Verbose` stream (which can be suppressed) as opposed to the `Information` stream.

Above and beyond

We'll dive into `Write-Verbose` and the other `Write` cmdlets a bit more in chapter 20. But if you try `Write-Verbose` now, you might be disappointed to discover that it doesn't produce any output. Well, not by default.

If you plan to use `Write` cmdlets, the trick is to turn them on first. For example, set `$VerbosePreference="Continue"` to enable `Write-Verbose`, and `$VerbosePreference="SilentlyContinue"` to suppress its output. You'll find similar "preference" variables for `Write-Debug` (`$DebugPreference`) and `Write-Warning` (`$WarningPreference`).

(continued)
Chapter 20 includes an even cooler way to use `Write-Verbose`.

It may seem *much* easier to use `Write-Host`, and if you want to, you can. But keep in mind that by using the other cmdlets, such as `Write-Verbose`, you're going to be following PowerShell's own patterns more closely, resulting in a more consistent experience.

17.4 Write-Output

Unlike `Write-Host`, `Write-Output` can send objects into the pipeline. Because it isn't writing directly to the display, it doesn't permit you to specify alternative colors or anything. In fact, `Write-Output` (or its alias, `Write`) isn't technically designed to display output at all. As we said, it sends objects into the pipeline—it's the pipeline itself that eventually displays those objects. Figure 17.3 illustrates how this works.

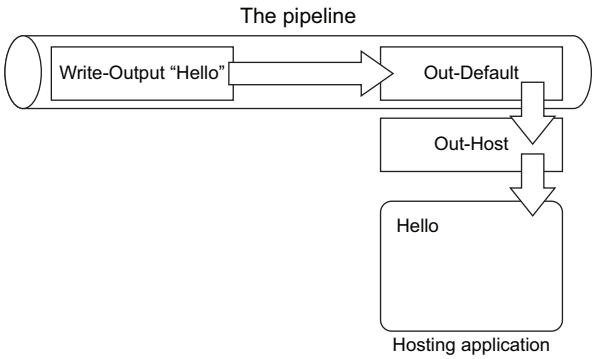

Figure 17.3 `Write-Output` puts objects into the pipeline, which in some cases eventually results in those objects being displayed.

Refer to chapter 11 for a quick review of how objects go from the pipeline to the screen. Let's look at the basic process:

1. `Write-Output` puts the `String` object `Hello` into the pipeline.
2. Because nothing else is in the pipeline, `Hello` travels to the end of the pipeline, where `Out-Default` always sits.
3. `Out-Default` passes the object to `Out-Host`.
4. `Out-Host` asks PowerShell's formatting system to format the object. Because in this example it's a simple `String`, the formatting system returns the text of the string.
5. `Out-Host` places the formatted result onto the screen.

The results are similar to what you'd get using `Write-Host`, but the object takes a different path to get there. That path is important, because the pipeline could contain other things. For example, consider the following command (which you're welcome to try):

```
PS C:\> write-output "Hello" | where-object { $_.length -gt 10 }
```

You don't see any output from this command, and figure 17.4 illustrates why. `Hello` is placed into the pipeline. But before it gets to `Out-Default`, it has to pass through `Where-Object`, which filters out anything having a `Length` property of less than or equal to `10`, which in this case includes our poor `Hello`. Our `Hello` gets dropped out of the pipeline, and because there's nothing left in the pipeline for `Out-Default`, there's nothing to pass to `Out-Host`, so nothing is displayed. Contrast that command with the following one:

```
PS C:\> write-host "Hello" | where-object { $_.length -gt 10 }
Hello
```

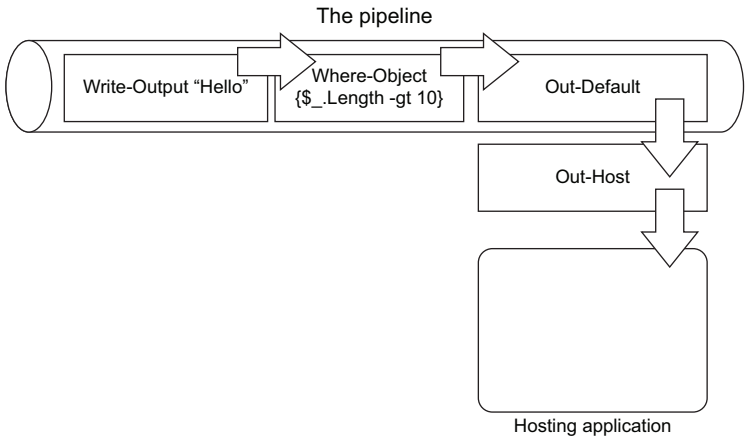

Figure 17.4 Placing objects into the pipeline means they can be filtered out before they're displayed.

All we've done is replace `Write-Output` with `Write-Host`. This time, `Hello` goes directly to the screen, not into the pipeline. `Where-Object` has no input and produces no output, so nothing is displayed by `Out-Default` and `Out-Host`. But because `Hello` has been written directly to the screen, we see it anyway.

`Write-Output` may seem new, but it turns out you've been using it all along. It's the shell's default cmdlet. When you tell the shell to do something that isn't a command, the shell passes whatever you typed to `Write-Output` behind the scenes.

17.5 Other ways to write

PowerShell has a few other ways to produce output. None of these write to the pipeline as `Write-Output` does; they work a bit more like `Write-Host`. But all of them produce output in a way that can be suppressed.

The shell comes with built-in configuration variables for each of these alternative output methods. When the configuration variable is set to `Continue`, the commands we're about to show you do indeed produce output. When the configuration variable is set to `SilentlyContinue`, the associated output command produces nothing. Table 17.1 contains the list of cmdlets.

Table 17.1 Alternative output cmdlets

Cmdlet	Purpose	Configuration variable
`Write-Warning`	Displays warning text, in yellow by default, and preceded by the label `WARNING:`	`$WarningPreference` (`Continue` by default)
`Write-Verbose`	Displays additional informative text, in yellow by default, and preceded by the label `VERBOSE:`	`$VerbosePreference` (`SilentlyContinue` by default)
`Write-Debug`	Displays debugging text, in yellow by default, and preceded by the label `DEBUG:`	`$DebugPreference` (`SilentlyContinue` by default)
`Write-Error`	Produces an error message	`$ErrorActionPreference` (`Continue` by default)
`Write-Information`	Displays informational messages and allows structured data to be written to an information stream	`$InformationPreference` (`SilentlyContinue` by default)

> **NOTE** `Write-Host` uses `Write-Information` under the hood, which means that `Write-Host` messages get sent to the information stream in addition to the host application. This gives us the ability to do more with `Write-Host` by controlling its behavior with `$InformationPreference`, among other things that we can do with PowerShell streams.

`Write-Error` works a bit differently because it writes an error to PowerShell's error stream. PowerShell also has a `Write-Progress` cmdlet that can display progress bars, but it works entirely differently. Feel free to read its help for more information and for examples; we don't cover it in this book.

To use any of these cmdlets, first make sure that its associated configuration variable is set to `Continue`. (If it's set to `SilentlyContinue`, which is the default for a couple of them, you won't see any output at all.) Then use the cmdlet to output a message.

> **NOTE** Some PowerShell hosting applications may display the output from these cmdlets in a different location. In Azure Functions, for example, debugging text is written to a log in Application Insights (an Azure log-reporting service) instead of a terminal window because in a serverless environment, you're not looking at a terminal; the PowerShell script is running somewhere up in the cloud. This is done for ease of debugging your scripts and so that you can see the output somewhere.

17.6 Lab

NOTE For this lab, you need a computer running the OS of your choice with PowerShell v7 or later.

`Write-Host` and `Write-Output` can be a bit tricky to work with. See how many of these tasks you can complete, and if you get stuck, it's okay to peek at the sample answers available at the end of this chapter.

1. Use `Write-Output` to display the result of 100 multiplied by 10.
2. Use `Write-Host` to display the result of 100 multiplied by 10.
3. Prompt the user to enter a name, and then display that name in yellow text.
4. Prompt the user to enter a name, and then display that name only if it's longer than five characters. Do this all with a single PowerShell expression—don't use a variable.

That's all for this lab. Because these cmdlets are all straightforward, we want you to spend more time experimenting with them on your own. Be sure to do that—we'll offer some ideas in section 17.8.

TRY IT NOW After you've completed this lab, try completing review lab 3, which you'll find in the appendix of this book.

17.7 Lab answers

1. `write-output (100*10)`
   or simply type the formula: `100*10`
2. Any of these approaches works:

```
$a= 100*10
Write-Host $a
Write-Host "The value of 100*10 is $a"
Write-Host (100*10)
```

3.
```
$name = Read-Host "Enter a name"
Write-host $name -ForegroundColor Yellow
```

4. `Read-Host "Enter a name" | where {$_.length -gt 5}`

17.8 Further exploration

Spend some time getting comfortable with all of the cmdlets in this chapter. Make sure you can display `Verbose` output, and accept input. You'll be using the commands from this chapter often from here on out, so you should read their help files and even jot down quick syntax reminders for future reference.