WINDOWS

PowerShell Fundamentals

The Object Pipeline, Not Text Streams

Providers, Cmdlets & Real Discovery

Variables, Control Flow & Functions

Error Handling, Execution Policy & Modules

Capstone: Automating a Real Admin Task

Philip Osztromok

Generated with Claude

Table of Contents

PowerShell Fundamentals — 12 Chapters

It's an easy first assumption: PowerShell opens in a black window, `dir` still lists a folder, `cd` still changes directory, so surely it's just the old MS-DOS/`cmd.exe` prompt with a longer command list bolted on. That assumption is exactly wrong, and it's worth correcting before anything else in this course, because the correction is the single idea everything else builds on. `dir` works in PowerShell — but so does `ls`, and so does `gci`, and none of that is a coincidence or a compatibility shim. It's a visible symptom of something genuinely different underneath: PowerShell doesn't pipe *text* between commands the way `cmd.exe` and Bash both do — it pipes real, structured **.NET objects**. That one design choice is what this whole chapter, and in a real sense this whole course, is built around.

Three Command-Line Lineages, Compared

	PIPELINE MODEL	WHERE IT'S COVERED ON THIS SITE
cmd.exe (DOS legacy)	Barely a pipeline at all — `|` exists, but every command's output is just a raw stream of characters, and batch scripting has no real data types or object model to speak of	Not separately covered — the legacy floor PowerShell was built to move past
Bash	A real pipeline, but a **text** one — every command's stdout is bytes, and tools like `grep` / `awk` / `sed` exist specifically to re-extract structure from that text	Bash Scripting Fundamentals, Intermediate & Advanced
PowerShell	An **object** pipeline — cmdlets pass real .NET objects to each other, with properties and methods still attached, no text parsing required	This course, and its own sequel

Seeing the Object Pipeline Directly

The clearest way to see the difference is to do the same everyday task both ways. Say you want the five files taking up the most space in a folder. In Bash, that means parsing `ls`'s own text output by column position:

```
# Bash — sort by size, but only because the size happens to sit in a known text column
ls -la | sort -k5 -rn | head -5
```

That works — until a locale change, a different `ls` implementation, or a filename with a space in it shifts the column layout, and the whole pipeline silently breaks. PowerShell's equivalent never touches text at all — it asks each file object directly for its `Length` property:

```
# PowerShell — sort by the real Length property on each file object, not a text column
Get-ChildItem | Sort-Object Length -Descending | Select-Object -First 5
```

`Sort-Object` isn't guessing which column holds the size — it's reading an actual `Length` property that was already sitting on every object `Get-ChildItem` produced. Nothing about the file's name, spacing, or the current locale can break that.

> **THE CENTRAL FACT THIS WHOLE COURSE IS BUILT ON**
>
> In Bash, "the pipeline" means "a stream of bytes that the next command has to re-parse." In PowerShell, "the pipeline" means "a stream of live objects that the next command can query directly." Every cmdlet name, every `Where-Object` / `Select-Object` filter, and every script you'll write in this course leans on that one distinction — it's the reason PowerShell reads as a genuinely different tool once you get past the surface-level `dir` / `cd` familiarity, not a reskinned `cmd.exe`.

Filtering by Property, Not by Pattern

Counting `.txt` files makes the same point a second way — Bash has to pattern-match text, PowerShell just asks for the property:

```
# Bash — a regex has to reconstruct "does this line end in .txt"
ls -la | grep '\.txt$' | wc -l

# PowerShell — Extension is already a real property on the object, no regex
needed
Get-ChildItem | Where-Object Extension -eq '.txt' | Measure-Object
```

`cmd.exe`: The Legacy Floor PowerShell Was Built to Replace

`cmd.exe` traces directly back to `COMMAND.COM` in MS-DOS — a command interpreter, not a scripting language in any real sense. Batch files (`.bat` / `.cmd`) can loop and branch, but every variable is a string, there's no object model, and error handling is famously primitive (`%ERRORLEVEL%` checked by convention, not enforced). PowerShell — started inside Microsoft in 2002 under the codename **Monad**, released in 2006 — was built specifically to replace that floor: a real, typed, .NET-backed scripting language with a genuine shell wrapped around it, not an incremental patch to `cmd.exe`.

To ease that transition, PowerShell ships with built-in **aliases** that map familiar `cmd.exe` and Unix command names onto its own cmdlets — which is exactly why your original instinct that `dir` would still work, and that `ls` might *also* work, was correct on both counts:

```
Get-Alias dir, ls, gci

# CommandType     Name                    Definition
# ———————————     ————                    ——————————
# Alias           dir → Get-ChildItem
# Alias           ls  → Get-ChildItem
# Alias           gci → Get-ChildItem
```

All three are just names for the same one cmdlet, `Get-ChildItem` — unlike `cmd.exe`'s own `dir`, which isn't an alias for anything; it's `cmd.exe`'s own single, built-in, non-substitutable implementation. And because PowerShell aliases are just names you can define yourself with `Set-Alias` (or a short function, for anything needing arguments), the same trick that gives you a Linux-style `ll` shortcut on a real terminal is available here too — Chapter 11 covers writing your own.

Windows PowerShell vs. PowerShell 7 — Which One Are You Actually Using?

One more thing worth sorting out before writing any real code: "PowerShell" today quietly refers to two different products.

- **Windows PowerShell 5.1** — built into every modern Windows install, running on the older .NET Framework, Windows-only, and now in maintenance mode: it still gets security fixes, but no new features.
- **PowerShell 7+** (sometimes called "PowerShell Core") — open-source, built on modern .NET, genuinely cross-platform (Windows, macOS, and Linux all run the identical `pwsh`), and where all active development happens.

```
$PSVersionTable.PSVersion

# Major  Minor  Patch
# ─────  ─────  ─────
# 7      4      2      ← PowerShell 7.4.x (pwsh.exe)
# 5      1      22621  ← Windows PowerShell 5.1 (powershell.exe)
```

A FIRST PRACTICAL HABIT

Run `$PSVersionTable.PSVersion` right now (or recall doing so) to see which one you're actually in. This course's examples target PowerShell 7+ (the `pwsh` executable) — everything shown works in Windows PowerShell 5.1 too unless a chapter says otherwise, but if you're setting up fresh, install PowerShell 7 rather than relying on whatever 5.1 build already shipped with Windows.

"IT'S JUST DOS WITH MORE COMMANDS" IS A COSTLY ASSUMPTION

Treat PowerShell's output as text you need to `grep` or regex apart, the way you might in Bash, and you'll write exactly the kind of brittle script this chapter's `Sort-Object` / `Where-Object` examples were built to avoid — one that quietly breaks the moment a filename, a locale, or a column width shifts. The object pipeline exists precisely so you never have to reconstruct structure that was never actually lost in the first place.

Where This Course Is Headed

Navigating the object filesystem through PowerShell's provider model, cmdlet discovery and getting real help (`Get-Help`, `Get-Command`, `Get-Member`), the pipeline in real depth, variables and PowerShell's own type system, control flow, functions and script files, working with files/text/formatted output, error handling and basic debugging, execution policy and script security, and modules/aliases/the PowerShell Gallery — closing with a capstone automating a real administrative task end to end. PowerShell Intermediate/Advanced then picks up from there: remoting, parallel execution, .NET/COM interop, calling REST APIs, publishing your own modules, and more.

Hands-On Exercises

EXERCISE 1

Using this chapter's own `.txt`-counting example, explain in your own words why PowerShell's object pipeline avoids the "same command name doesn't guarantee the same result" trap that a Bash pipeline built on `grep`/regex can fall into.

View solution

EXERCISE 2

Run `Get-Alias dir, ls, gci` (or recall this chapter's own output). What single cmdlet do all three point to? Then explain why `cmd.exe`'s own `dir` isn't an alias for anything the same way.

View solution

EXERCISE 3

Run `$PSVersionTable.PSVersion` (or recall this chapter's own explanation) and describe the practical difference between Windows PowerShell 5.1 and PowerShell 7+. Which one does this course target, and why?

View solution

CHAPTER 1 QUICK REFERENCE

- **Object pipeline** — PowerShell cmdlets pass real .NET objects to each other, properties and methods intact; Bash and `cmd.exe` both pass raw text
- `Get-ChildItem` — the real cmdlet behind the `dir`, `ls`, and `gci` aliases
- `Sort-Object` / `Where-Object` / `Select-Object` — filter and shape objects by real property, not by text pattern
- **Monad → PowerShell** — started inside Microsoft in 2002, released 2006, built specifically to replace `cmd.exe`'s text-only batch model
- **Windows PowerShell 5.1** — built in, .NET Framework, Windows-only, maintenance mode
- **PowerShell 7+** — cross-platform, modern .NET, actively developed; this course's own target
- `$PSVersionTable.PSVersion` — check which one you're actually running
- **Next chapter:** Getting Around: Providers, PSDrives & Navigating the Object Filesystem

Chapter 1 used `Get-ChildItem` (by way of its `dir` / `ls` / `gci` aliases) purely to list files, which makes it easy to read as "PowerShell's version of `dir`" and stop there. It's actually doing something bigger: `Get-ChildItem`, `Set-Location`, and the rest of PowerShell's navigation cmdlets don't know anything about files specifically at all. They know how to walk a generic, drive-and-item structure — and PowerShell ships several completely different things dressed up to look like that same structure. That's the **provider** model, and it's the reason a handful of cmdlets you already know can browse the registry, your environment variables, and your certificate store with zero new syntax to learn.

Providers: The Abstraction Underneath Every "Drive"

A **PSProvider** is a piece of PowerShell that takes some data store — a real filesystem, but also the Windows Registry, the current session's environment variables, even the certificate store — and exposes it as a hierarchy of drives, folders, and items. List every provider your session has loaded with `Get-PSProvider`:

```
Get-PSProvider
# Name                Capabilities                          Drives
# ----                ------------                          ------
# Registry            ShouldProcess, Transactions           {HKLM, HKCU}
# Alias               ShouldProcess                         {Alias}
# Environment          ShouldProcess                         {Env}
# FileSystem          Filter, ShouldProcess, Credentials     {C, D, Temp}
# Function             ShouldProcess                         {Function}
# Variable            ShouldProcess                         {Variable}
# Certificate          ShouldProcess                         {Cert}
```

`cmd.exe` has nothing resembling this — its `dir` only ever meant "a folder on a disk," full stop. Bash comes closer in spirit (Linux's own "everything is a file" philosophy exposes plenty of non-disk data as real files under `/proc` and `/sys`), but that's a kernel-level trick baked into the operating system itself. PowerShell's providers achieve a similar effect entirely at the shell layer, which is why the exact same list above includes things that have nothing to do with a hard drive at all.

PSDrives: Not Just C: and D:

Each provider exposes one or more **PSDrives** — and despite the colon-suffixed, drive-letter-looking name, most of them were never disks in the first place. `Get-PSDrive` lists every drive currently available to navigate:

```
Get-PSDrive
# Name           Used (GB)     Free (GB) Provider      Root
# ────           ─────────     ───────── ────────      ────
# C                    412           138 FileSystem    C:\
# Alias                                  Alias
# Cert                                   Certificate   \
# Env                                    Environment
# Function                               Function
# HKCU                                   Registry      HKEY_CURRENT_USER
# HKLM                                   Registry      HKEY_LOCAL_MACHINE
# Variable                               Variable
```

`C` is the only row here backed by an actual disk. `HKLM`, `Env`, `Cert`, and the rest have no size at all, because "used/free gigabytes" is a filesystem-specific idea that simply doesn't apply once the underlying data isn't a disk — the **Used**/**Free** columns are blank for exactly that reason.

Navigating with the Same Cmdlets, Everywhere

Here's the payoff: `Set-Location` (aliased `cd`), `Get-Location` (aliased `pwd`), `Push-Location`/`Pop-Location` (aliased `pushd`/`popd`), and Chapter 1's own `Get-ChildItem` all work against *any* PSDrive, not just `FileSystem` ones — because they're written against the provider abstraction, not against "files" specifically:

```
# Browsing the real filesystem — exactly what you'd expect
Set-Location C:\Windows
Get-ChildItem
# Browsing environment variables — the *same two cmdlets*, a completely
different data store
Set-Location Env:
Get-ChildItem
# Browsing the registry — still the same two cmdlets
Set-Location HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion
Get-ChildItem
```

> **THE CENTRAL FACT THIS CHAPTER IS BUILT ON**
>
> Learning to navigate *once* means you already know how to browse the registry, your environment variables, and your certificate store — no new tool, no new syntax, no separate GUI to learn. Compare that to the alternative: on Windows without PowerShell, reading the registry means opening `regedit.exe`, checking an environment variable means the System Properties dialog, and checking a certificate means the Certificate Manager snap-in — three completely different tools for three completely different jobs. On Linux, the registry's job doesn't really exist as a single store at all, so there's no direct equivalent to compare against. PowerShell's provider model collapses all of that into the same five or six cmdlets you already used in Chapter 1.

Reading a Value: Get-ChildItem vs. Get-ItemProperty

One real gotcha shows up the first time you try to read an actual registry *value* rather than just list subkeys. In the filesystem, `Get-ChildItem` lists a folder's contents (its children) — but a registry key's values aren't children the way subkeys are; they're **properties** attached to the key itself. Listing children won't show them:

```
# Lists subkeys — but never shows a key's own values, no matter how deep you go
Get-ChildItem HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion

# Get-ItemProperty reads the key's actual values — this is what you actually
want
Get-ItemProperty -Path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion -Name
ProgramFilesDir

# ProgramFilesDir
# ———————————
# C:\Program Files
```

This is a case where the provider abstraction is genuinely leaky, not seamless — the registry's own key-vs-value split doesn't map perfectly onto a filesystem's folder-vs-file split, and `Get-ItemProperty` exists specifically to bridge that gap.

Every Built-In Provider, at a Glance

PROVIDER	WHAT IT EXPOSES	EXAMPLE DRIVE
FileSystem	Files and folders — the one provider with a real disk underneath	`C:\`
Registry	The Windows Registry's two loaded hives	`HKLM:\`, `HKCU:\`
Environment	The current session's environment variables	`Env:\`
Certificate	Installed certificates and certificate stores	`Cert:\`
Alias	Every alias defined in the current session (Chapter 1's `dir` / `ls` / `gci` among them)	`Alias:\`
Variable	Every PowerShell variable currently in scope	`Variable:\`
Function	Every function currently defined in the session	`Function:\`

A FIRST PRACTICAL HABIT

Run `Get-PSDrive` right now (or recall this chapter's own output) and pick one non-filesystem drive you've never browsed — `Env:` is the easiest starting point. `Set-Location Env:` then `Get-ChildItem` and look at what's actually sitting in your own environment. It's the same two commands you already know; only the destination changed.

THE FILESYSTEM'S OWN HABITS DON'T TRANSFER PERFECTLY

Because the same cmdlets work everywhere, it's tempting to assume they work *identically* everywhere too — they don't. `Get-Content` reads a text file's lines just fine, but running it against a registry key doesn't give you that key's values (use `Get-ItemProperty`, shown above, instead). And `Remove-Item -Recurse` against a registry key deletes that key and every subkey underneath it permanently, with no Recycle Bin the way a deleted file might have one — treat any destructive command against `HKLM:` / `HKCU:` with real caution, not filesystem-level confidence.

Creating Your Own PSDrive

Beyond the built-in ones, you can map any FileSystem path to a short custom drive name with `New-PSDrive` — handy for a deep or frequently-used folder:

```
New-PSDrive -Name Proj -PSProvider FileSystem -Root
"C:\Users\me\Projects\website-content"
# From here on, for this session, this works:
Set-Location Proj:\
Get-ChildItem
```

`New-PSDrive` without `-Persist` only lasts for the current session — it disappears the moment you close the window, which is usually exactly what you want for a quick shortcut rather than a permanent change.

Where This Course Is Headed

Cmdlet syntax, discovery, and getting real help (`Get-Help`, `Get-Command`, `Get-Member`), the pipeline in real depth, variables and PowerShell's own type system, control flow, functions and script files, working with files/text/formatted output, error handling and basic debugging, execution policy and script security, and modules/aliases/the PowerShell Gallery — closing with a capstone automating a real administrative task end to end.

Hands-On Exercises

EXERCISE 1

Using this chapter's own finding-box, explain why `Set-Location` and `Get-ChildItem` work identically against `C:\Windows`, `Env:`, and `HKLM:\SOFTWARE` even though a disk folder, an environment variable, and a registry key have almost nothing in common as data.

View solution

EXERCISE 2

Run `Get-PSDrive` (or recall this chapter's own output) and name three drives with no **Used**/**Free** values shown. Explain why those columns are blank for those specific drives.

View solution

EXERCISE 3

Using the `ProgramFilesDir` example, explain why `Get-ChildItem` alone can't show you a registry key's actual value, and name the cmdlet that's needed instead. What's the underlying registry concept (keys vs. something else) that causes this gap?

View solution

CHAPTER 2 QUICK REFERENCE

- **Provider (PSProvider)** — exposes a data store as a drive/folder/item hierarchy; `Get-PSProvider` lists them
- **PSDrive** — a specific drive exposed by a provider; `Get-PSDrive` lists them — most have nothing to do with a disk
- **Built-in providers** — FileSystem, Registry, Environment, Certificate, Alias, Variable, Function
- `Set-Location` / `Get-Location` / `Push-Location` / `Pop-Location` — `cd` / `pwd` / `pushd` / `popd`; work identically against every provider
- `Get-ItemProperty` — reads a registry key's own values; `Get-ChildItem` only lists its subkeys
- `New-PSDrive` — map any FileSystem path to a short custom drive name for the current session
- **This chapter's own throughline:** one small set of navigation cmdlets, reused unmodified across every provider — learn once, browse everywhere
- **Next chapter:** Cmdlet Syntax, Discovery & Help

Every cmdlet used so far — `Get-ChildItem`, `Sort-Object`, `Where-Object`, `Get-PSDrive`, `Get-ItemProperty` — has followed the exact same two-word shape: a verb, a dash, a noun. That's not a coincidence, and it's not decoration. It's a deliberate, enforced naming convention, and once you know how it works you can often guess a cmdlet's name correctly before ever looking it up — and when a guess is wrong, three built-in tools (`Get-Command`, `Get-Help`, `Get-Member`) will get you to the right answer faster than a web search.

Verb-Noun: A Naming Convention You Can Actually Rely On

Every real PowerShell cmdlet is named `Verb-Noun` — `Get-Process`, `Stop-Service`, `New-Item`, `Remove-Item`. The noun is usually singular and specific to what it operates on; the verb comes from a fixed, curated list Microsoft calls the **approved verbs** — a deliberately short list, not an open vocabulary:

```
Get-Verb | Select-Object Verb, Group | Sort-Object Group

# Verb      Group
# ----      -----
# Get       Common
# Set       Common
# New       Common
# Remove    Common
# Start     Lifecycle
# Stop      Lifecycle
# Add       Common
# Clear     Common
# ...       (about 100 approved verbs total, grouped by intent)
```

There is no approved verb called "Delete," "Create," or "Fetch" — which is exactly why a plausible-sounding guess like `Delete-Item` or `Fetch-Process` fails outright. The real cmdlets are `Remove-Item` and `Get-Process`. This isn't PowerShell being needlessly picky; it's the entire point. **Every** module — built-in or from any third party — is expected to follow the same approved-verb list, which is what makes a skill like "I can usually guess a cmdlet name" transfer to modules you've never even installed yet.

	NAMING CONVENTION	CONSEQUENCE
cmd.exe	Short, often abbreviated, no shared pattern (`dir`, `copy`, `del`, `ren`)	Nothing to guess from — every command name has to be individually memorized
Bash	Independent standalone tools, each named by whoever wrote it (`ls`, `grep`, `awk`, `sed`, `curl`)	No shared convention at all — a tool's name tells you nothing about what family it belongs to
PowerShell	Verb-Noun, verb drawn from a fixed approved list, enforced across every module	A cmdlet's own name usually tells you what it does, and a wrong guess is often just one approved-verb swap away from correct

Get-Command: Discovering Cmdlets Without Memorizing Them

`Get-Command` searches every loaded cmdlet by verb, noun, or module — the tool for "I don't remember the exact name, but I know roughly what I want":

```
# Every cmdlet that operates on something process-related
Get-Command -Noun Process*

# Every "Get" cmdlet related to services - a search by both verb and noun
Get-Command -Verb Get -Noun *Service*

# CommandType     Name                    Version    Source
# ———————————     ————                    ———————    ——————
# Cmdlet          Get-Service             3.1.0.0
Microsoft.PowerShell.Management
```

Get-Help: Reading the Manual Without Leaving the Shell

Once you know a cmdlet's name, `Get-Help` is the built-in manual — full syntax, parameter descriptions, and (the genuinely useful part) worked examples:

```
# Just the worked examples — usually the fastest way to relearn a cmdlet you've used before
Get-Help Get-Service -Examples

# The complete reference: every parameter, every note, every example
Get-Help Get-Service -Full

# Opens the live, always-current Microsoft Docs page in your browser
Get-Help Get-Service -Online
```

The help text bundled with PowerShell isn't installed by default on a fresh system — run `Update-Help` once (as Administrator, since it downloads content) to populate it, or reach for `-Online` in the meantime.

Get-Member: Asking an Object What It Actually Has

This is the tool that closes the loop on Chapter 1's own object-pipeline claim. Back then, `Get-Process | Sort-Object CPU -Descending` just used `CPU` as if it were obviously a valid property to sort by. It wasn't a guess — it's discoverable, and `Get-Member` is how:

```
Get-Process | Get-Member -MemberType Property

# TypeName: System.Diagnostics.Process
#
# Name                Definition
# ----                ----------
# CPU                 double CPU {get;}
# Id                  int Id {get;}
# ProcessName         string ProcessName {get;}
# WorkingSet          long WorkingSet {get;}
# ...                 (dozens more)
```

Pipe *any* object into `Get-Member` and it lists every real property and method that object actually has — which is also exactly how you'd have discovered `Length` was a valid property on a file object back in Chapter 2's own `Sort-Object Length` example, without needing to already know it in advance.

> **THE CENTRAL FACT THIS CHAPTER IS BUILT ON**
>
> You don't need to memorize hundreds of cmdlets or thousands of object properties. You need to memorize **three commands** — `Get-Command` to find a cmdlet by what it does, `Get-Help` to learn exactly how to use one you've found, and `Get-Member` to see what any object flowing through the pipeline actually offers. That three-command discovery loop works identically against a module you installed five minutes ago as it does against anything built in — which is precisely why it's worth learning now, before Chapter 11 starts introducing modules you haven't seen yet.

A FIRST PRACTICAL HABIT

Whenever a chapter (in this course or anywhere else) shows a property you haven't seen before — like `CPU` or `Length` above — get in the habit of piping that same command into `Get-Member` instead of taking the property name on faith. It turns "the lesson used this property" into "I confirmed this property for myself," which is a much sturdier way to actually learn the object pipeline.

A PLAUSIBLE-SOUNDING GUESS STILL ISN'T A REAL CMDLET

`Delete-Item`, `Create-Item`, and `Fetch-Item` all read as reasonable English, and none of them exist — "Delete," "Create," and "Fetch" simply aren't approved verbs. Before assuming a cmdlet you've guessed at is genuinely missing (or worse, silently trying several near-guesses until one happens to run), reach for `Get-Command -Verb Get -Noun Item` or an equivalent search first — it'll get you to the real name (`Remove-Item`, `New-Item`, `Get-Item`) faster and more reliably than trial and error.

Hands-On Exercises

EXERCISE 1

Explain why `Delete-Item` produces an error in PowerShell rather than deleting something. Name the actual cmdlet, and name the concept (covered in this chapter) that explains why "Delete" specifically was never a valid choice.

View solution

EXERCISE 2

Run `Get-Process | Get-Member -MemberType Property` (or recall this chapter's own output). Explain how this connects back to Chapter 1's `Sort-Object CPU -Descending` example — specifically, how would you have confirmed `CPU` was a valid property to sort by without already knowing it in advance?

View solution

EXERCISE 3

You want to find a cmdlet for starting a Windows service, but you don't remember its exact name. Write the `Get-Command` call you'd use to find it, and explain your search strategy (which parts you searched by verb versus by noun, and why).

View solution

CHAPTER 3 QUICK REFERENCE

- **Verb-Noun** — every cmdlet's name; the verb is drawn from a fixed, enforced approved-verb list (`Get-Verb`)
- `Get-Command` — discover a cmdlet by `-Verb` / `-Noun` / `-Module` when you don't remember the exact name
- `Get-Help <cmdlet> -Examples / -Full / -Online` — the built-in manual, including worked examples
- `Update-Help` — populates local help content on a fresh install (run once, as Administrator)
- `Get-Member` — pipe any object into it to see every real property and method it actually has
- **This chapter's own throughline:** memorize the three-command discovery loop (`Get-Command` / `Get-Help` / `Get-Member`), not the cmdlets themselves
- **Next chapter:** The Pipeline: Passing Objects, Not Text

Chapters 1 through 3 have already used `Where-Object`, `Sort-Object`, and `Select-Object` in passing — enough to show that filtering happens by real property, not text pattern. This chapter goes underneath that and treats the pipeline as the subject in its own right: how it actually moves objects from one cmdlet to the next, the two different ways to write a `Where-Object` filter, the comparison operators available once you're inside one, and the real difference between `Select-Object`'s two shaping modes — a distinction that causes genuine confusion the first time it bites.

One Object at a Time, Not One Big Blob

A PowerShell pipeline doesn't run the first cmdlet to completion, hand off one giant collection, then run the second cmdlet on all of it. Each object produced by one stage is pushed into the next stage individually, as soon as it's available — `Get-Process` can start handing processes to `Where-Object` before it has even finished enumerating every running process on the system. This is part of why PowerShell pipelines stay responsive on large result sets: you often see output start scrolling before the upstream cmdlet has fully finished.

`$_` (a.k.a. `$PSItem`): "Whatever Object Is Passing Through Right Now"

Inside a script block running per pipeline object, `$_` refers to that one current object — the object being evaluated *this* time through, not the whole collection. `$PSItem` is an identical, more readable alias for the exact same thing; both are used interchangeably in real scripts.

```
Get-Process | Where-Object { $_.CPU -gt 100 }

# Identical — $PSItem is just a friendlier name for the same thing
Get-Process | Where-Object { $PSItem.CPU -gt 100 }
```

Two Ways to Write a Where-Object Filter

For a single, simple comparison, PowerShell offers a shorter **simplified syntax** that skips `$_` entirely — property name, operator, value:

```
# Simplified syntax — one property, one comparison, no $_ needed
Get-Process | Where-Object CPU -gt 100

# Script-block syntax — required once the condition is more than one simple
comparison
Get-Process | Where-Object { $_.CPU -gt 100 -and $_.ProcessName -like "chrome*"
}
```

The simplified form reads cleanly for the common case, but it only supports a single property comparison — the moment you need to combine two conditions with `-and`/`-or`, or reference a computed expression rather than a bare property, the script-block form with `$_` is what you actually need.

Comparison & Logical Operators

OPERATOR	MEANING	EXAMPLE
-eq / -ne	Equal / not equal	`$_.Status -eq 'Running'`
-gt / -ge / -lt / -le	Greater than / at least / less than / at most	`$_.CPU -gt 100`
-like	Wildcard text match (`*`, `?`) — not a regular expression	`$_.Name -like "*.txt"`
-match	Regular-expression match	`$_.Name -match '^\d{3}-'`
-contains / -in	Is a value present in a collection (reversed operand order between the two)	`$stopped -contains $_.Name`
-and / -or / -not	Combine multiple conditions — only valid inside the script-block form	`$_.CPU -gt 100 -and $_.Id -ne 0`

Every one of these is spelled with a leading dash, not a symbol — PowerShell doesn't use `==`, `>`, or `&&` for these comparisons the way Bash's `[[ ]]` or JavaScript would, precisely because `>` and `<` are already taken for file redirection, the same way they are in Bash itself.

Select-Object: Shaping What Comes Out

`Select-Object` does two genuinely different jobs depending on which switch you reach for, and mixing them up is a real, common gotcha:

```
# -Property keeps chosen properties, but each result is still a full object
Get-Process | Select-Object -Property ProcessName, CPU -First 3

# -ExpandProperty unwraps ONE property into a plain value — no longer a process object at all
Get-Process | Select-Object -ExpandProperty ProcessName -First 3

# chrome
# chrome
# explorer
```

`-Property` gives you back a trimmed-down object — still an object, with `ProcessName` and `CPU` attached, just fewer properties than before. `-ExpandProperty` gives you back the raw value itself, with the object wrapper stripped away entirely. That distinction matters the moment you pipe the result somewhere else, or compare it directly: a `-Property` result compared with `-eq "chrome"` will never match, because you're comparing a whole object against a string — you needed `-ExpandProperty` for that.

`Select-Object` also takes `-First`/`-Last` (already used above and in Chapter 1), `-Skip` (skip a number of leading results), and `-Unique` (drop duplicate rows, compared by whichever properties were selected).

ForEach-Object: Doing Something With Each Object

Where `Where-Object` decides whether an object continues down the pipeline, `ForEach-Object` runs an action against every object that reaches it — the pipeline's own equivalent of a loop body, using the same `$_`:

```
Get-Process chrome | ForEach-Object {
    "$($_.ProcessName) is using $($_.WorkingSet / 1MB) MB"
}
```

`foreach` as a standalone loop keyword (rather than the `ForEach-Object` cmdlet shown here) is genuinely different and gets its own proper treatment in Chapter 6 — for now, just note that this chapter's `ForEach-Object` is a pipeline stage, not the loop syntax you'll write inside a script body.

Putting It Together: A Real Multi-Stage Pipeline

```
Get-Process |
    Where-Object { $_.CPU -gt 10 } |
    Sort-Object CPU -Descending |
    Select-Object -First 5 -Property ProcessName, CPU, Id
```

THE CENTRAL FACT THIS CHAPTER IS BUILT ON

Every stage in that pipeline — filter, sort, trim to five, keep three properties — operated on the exact same live objects, in sequence, with zero re-parsing at any step. Nothing was ever converted to text and back. Compare that to the equivalent Bash pipeline, which would need `ps`'s text columns to survive unchanged through `awk`, `sort -k3 -nr`, and `head` all at once — three separate tools, each trusting the last one's exact text formatting. This is Chapter 1's own central claim, now shown as a full working pipeline rather than a single-stage example.

A FIRST PRACTICAL HABIT

When a pipeline isn't behaving the way you expect, build it up one stage at a time — run just `Get-Process`, check the output, add `| Where-Object { ... }` and check again, then add the next stage. Because each stage is a real, inspectable set of objects, you can always see exactly what the next cmdlet is actually receiving.

-PROPERTY AND -EXPANDPROPERTY ARE NOT INTERCHANGEABLE

Swapping one for the other doesn't just change formatting — it changes the actual type of what comes out. A script that expects `-ExpandProperty`'s plain string but receives a `-Property` object (or vice versa) will fail in ways that don't always throw an obvious error, especially when the result is immediately displayed rather than compared or piped onward. When something downstream unexpectedly stops matching, this is one of the first things worth checking.

Hands-On Exercises

EXERCISE 1

Explain the difference between `Get-Process | Where-Object CPU -gt 100` and `Get-Process | Where-Object { $_.CPU -gt 100 -and $_.Id -ne 0 }`. Why does the second condition require the script-block form?

View solution

EXERCISE 2

Explain the difference between `Select-Object -Property ProcessName` and `Select-Object -ExpandProperty ProcessName`, using a concrete situation (from this chapter or your own) where using the wrong one would actually cause a problem.

View solution

EXERCISE 3

Write a single pipeline that finds the 3 largest `.log` files in the current folder by size, and outputs only their names (not full file objects). Combine `Get-ChildItem`, `Where-Object`, `Sort-Object`, and `Select-Object`.

View solution

CHAPTER 4 QUICK REFERENCE

- **Streaming pipeline** — objects flow one at a time from stage to stage, not as one collected batch
- `$_` / `$PSItem` — the current pipeline object, inside a script block
- **Simplified vs. script-block** `Where-Object` — simplified handles one property comparison; script-block (with `$_`) is required for `-and` / `-or` or computed conditions
- `-eq -ne -gt -ge -lt -le -like -match -contains -in` — comparison operators, always dash-prefixed, never `==` / `>` / `&&`
- `Select-Object -Property` — trims an object's properties, still returns an object
- `Select-Object -ExpandProperty` — unwraps one property to its raw value, no longer an object
- `ForEach-Object` — runs an action per pipeline object; not the same as the `foreach` loop keyword (Chapter 6)
- **Next chapter:** Variables, Data Types & Operators

Every object pipelined through `Get-Member` in Chapter 3, and every property tested with `-eq` / `-gt` / `-like` in Chapter 4, was a real, typed .NET object the entire time — that's exactly what made those chapters work. Variables are no different, even though declaring one looks deceptively casual. This chapter is about that duality: PowerShell lets you create a variable with zero type declaration, the same relaxed feel as Bash or Python, while every value it ever holds stays a genuine, strongly-typed object underneath — not the loose, everything-is-basically-text world Bash actually lives in.

Declaring a Variable

Every variable name starts with `$` — on both assignment *and* read, which is one real, easy-to-trip-on difference from Bash, where `$` only appears when *reading* a variable (`name="Philip"` to assign, `$name` to read):

```
$name = "Philip"
$age = 42
Write-Output $name
```

No type was declared anywhere in that example — and yet neither variable is untyped. Ask any variable what it actually is with `.GetType()`:

```
$name.GetType().Name    # String
$age.GetType().Name     # Int32
```

> **THE CENTRAL FACT THIS CHAPTER IS BUILT ON**
>
> "Dynamically typed" here means the *variable* isn't locked to one type — `$age` could hold a string tomorrow if you reassign it. It does not mean the *value* is typeless the way Bash treats everything as fundamentally text until proven otherwise. Every value in PowerShell is a real .NET object with a real, discoverable type, the whole time — which is exactly why `Get-Member` (Chapter 3) always has something concrete to report, and why `-gt` / `-lt` (Chapter 4) can compare numbers as numbers instead of comparing digit characters as text.

Casting & Type Accelerators

A short bracketed name before a value — a **type accelerator** — converts it explicitly:

```
[int]"42" + 8      # 50 — real addition, because "42" was cast to a real number
first
"42" + 8            # "428" — no cast, so + falls back to string concatenation
# A type-constrained variable rejects a value that can't convert
[int]$count = 5
$count = "hello" # throws — "hello" can't become an Int32
```

ACCELERATOR	.NET TYPE	HOLDS
[string]	System.String	Text
[int]	System.Int32	Whole numbers
[double]	System.Double	Decimal numbers
[bool]	System.Boolean	`$true` / `$false`
[datetime]	System.DateTime	Dates and times
[array]	System.Array	An ordered collection
[hashtable]	System.Collections.Hashtable	Key-value pairs

Arithmetic — and a Genuinely Surprising Rounding Gotcha

```
10 + 3     # 13
10 % 3     # 1  - modulo, the remainder
7 / 2      # 3.5 - division auto-widens to a double, it does not truncate like
some languages
```

CASTING A .5 VALUE DOESN'T ALWAYS ROUND THE WAY YOU'D EXPECT

Casting a `double` back to `[int]` uses **banker's rounding** (round-half-to-even), not the "always round .5 up" rule most people assume by default: `[int]2.5` is `2`, but `[int]3.5` is `4` — both rounded to the nearest *even* number, not simply upward. If you specifically want traditional round-half-up behavior, reach for `[math]::Round($value, 0, [MidpointRounding]::AwayFromZero)` instead of a bare cast.

Strings: Quoting, Interpolation & Joining

```
# Single quotes - literal, no interpolation at all
'Hello, $name' # Hello, $name  (printed exactly as typed)
# Double quotes - variables and $( ... ) expressions are evaluated inline
"Hello, $name" # Hello, Philip
"Next year you'll be $($age + 1)" # Next year you'll be 43
# -join / -split move between an array and a single string
"a,b,c" -split "," # @('a', 'b', 'c')
@("a", "b", "c") -join "-" # "a-b-c"
```

That single-vs-double distinction is a real, common trip-up: reach for single quotes when you actually want `$` printed literally, and double quotes any time a variable or expression should be substituted in.

Arrays: `@()`

```
$fruits = @("apple", "banana", "cherry")
$fruits[0]        # apple  — zero-indexed
$fruits[-1]       # cherry — negative indexing counts from the end
$fruits[0..1]     # apple, banana — the .. range operator
$fruits.Count     # 3
# Arrays can freely mix types — nothing enforces a single element type
$mixed = @(1, "two", 3.0, $true)
```

Hashtables: `@{}`

```
$person = @{ Name = "Alice"; Age = 30 }
$person['Age']     # 30 — bracket access
$person.Age        # 30 — dot access, same result
# A plain @{} hashtable does NOT guarantee its keys stay in insertion order
# [ordered]@{} does — reach for it whenever display order actually matters
$ordered = [ordered]@{ First = 1; Second = 2; Third = 3 }
```

Looping over every key or value in a hashtable properly belongs to Chapter 6's own control-flow material — for now, know that `@{}` and `[ordered]@{}` are genuinely different types with a real behavioral difference, not just a stylistic choice.

Automatic Variables Worth Knowing Now

VARIABLE	HOLDS
$null	PowerShell's explicit "no value" — comparing something to `$null` uses `-eq` / `-ne` like any other value
$true / $false	The two `[bool]` literals
$_ / $PSItem	The current pipeline object inside a script block (Chapter 4)
$PSVersionTable	The running PowerShell version details (Chapter 1)
$Error	A list of recent errors — covered properly in Chapter 9
$args	Positional arguments passed to a script or function — covered properly in Chapter 7

A FIRST PRACTICAL HABIT

Whenever a value's behavior surprises you — a comparison that doesn't match, arithmetic that doesn't look right — reach for `.GetType()` before assuming PowerShell is wrong. Nine times out of ten, the type wasn't what you assumed it was, and `.GetType()` tells you immediately rather than leaving you to guess.

Hands-On Exercises

EXERCISE 1

Predict what `.GetType().Name` reports for `$x = "42"` versus `$x = 42`, and predict what `$x + 8` does in each case. Explain the reasoning behind both predictions using this chapter's own casting example.

View solution

EXERCISE 2

Explain why `[int]2.5` evaluates to `2` while `[int]3.5` evaluates to `4`. What should you use instead if you specifically want traditional round-half-up behavior?

View solution

EXERCISE 3

Given `$person = @{ Name = "Alice"; Age = 30 }`, show two different ways to read the `Age` value. Then explain why a plain `@{}` hashtable's key order isn't guaranteed the way an array's order is, and what fixes that.

View solution

CHAPTER 5 QUICK REFERENCE

- `$name = value` — `$` is used on both assignment and read, unlike Bash
- `.GetType()` — reveals a variable's real underlying .NET type
- **Type accelerators** — `[string]` `[int]` `[double]` `[bool]` `[datetime]` `[array]` `[hashtable]`, for explicit casting or constraining a variable
- **Division auto-widens** — `7 / 2` is `3.5`, a double, not a truncated integer
- **Casting to `[int]` rounds half-to-even** — `[int]2.5` is `2`; `[int]3.5` is `4`
- **Single vs. double quotes** — single is literal; double interpolates variables and `$( ... )` expressions
- `@()` **arrays** — zero-indexed, negative indexing, `..` range operator, freely mixed types
- `@{}` **vs.** `[ordered]@{}` — a plain hashtable doesn't guarantee key order; `[ordered]@{}` does
- **Next chapter:** Control Flow: Conditionals, Loops & Switch

Chapter 4 already introduced `ForEach-Object` — but strictly as one stage inside a pipeline, using `$_` to touch each object flowing through it. This chapter is about writing an actual script body: `if` / `elseif` / `else`, a genuinely more capable `switch` statement than most languages ship with, and a `foreach` loop keyword that looks almost identical to Chapter 4's cmdlet by name — while behaving differently enough underneath that mixing the two up is one of the most common early PowerShell mistakes.

if / elseif / else

Braces, not indentation (unlike Python) and not `then` / `fi` keywords (unlike Bash) — conditions go in parentheses, bodies go in braces:

```
if ($age -ge 65) {
    "Senior"
} elseif ($age -ge 18) {
    "Adult"
} else {
    "Minor"
}
```

switch: PowerShell's Own Surprisingly Powerful Version

A basic `switch` looks familiar from other languages, but it comes with real extras — wildcard matching, regex matching, and (unlike almost anything else called "switch") the ability to iterate an entire array directly:

```
# Basic value matching
switch ($status) {
    "Running" { "Service is up" }
    "Stopped" { "Service is down" }
    default   { "Unknown status" }
}

# -Wildcard — the same * / ? matching Chapter 4's -like used
switch -Wildcard ($filename) {
    "*.txt" { "Text file" }
    "*.log" { "Log file" }
    default { "Other" }
}

# Handed an array instead of a single value, switch iterates it automatically
switch (@(1, 2, 3, 4)) {
    { $_ % 2 -eq 0 } { "$_ is even" }
    default          { "$_ is odd" }
}
# 1 is odd / 2 is even / 3 is odd / 4 is even — all four run, one per array
element
```

> **SWITCH RUNS EVERY MATCHING CLAUSE, NOT JUST THE FIRST ONE**
>
> Most languages' `switch` stops at the first match unless you fall through on purpose. PowerShell's does the opposite by default: **every** clause whose condition matches actually runs, one after another, for the same input. `switch ($x) { {$_ -gt 0} {"positive"} {$_ -gt 10} {"big"} }` run with `$x = 15` prints *both* `"positive"` and `"big"` — both conditions are true for 15. If you want only the first match to run, add `break` at the end of each clause body.

foreach: The Loop Keyword (Not the Cmdlet)

`foreach` written as a bare keyword — no pipe, no `-Object` — is a completely different piece of syntax from Chapter 4's `ForEach-Object`, despite the near-identical name:

```
foreach ($p in Get-Process) {
    "$($p.ProcessName) is using $($p.CPU) seconds of CPU"
}
```

THE CENTRAL DISTINCTION THIS CHAPTER IS BUILT ON

`foreach` is a **language statement**: it fully evaluates `Get-Process` into a complete, in-memory collection first, then loops over that finished collection. `Get-Process | ForEach-Object { ... }` (Chapter 4) is a **pipeline stage**: it processes each process object as soon as it arrives, one at a time, without ever holding the full set in memory at once. On a small result set the difference is invisible. On something genuinely large — every row from a multi-million-row query, or every file under a massive directory tree — `ForEach-Object`'s streaming behavior can matter a great deal, while a bare `foreach` has to build the whole collection before the loop body runs even once.

while, do-while & do-until

```
# while — checks the condition BEFORE each pass; may run zero times
$i = 0
while ($i -lt 3) {
    "i is $i"
$i++
}

# do-while — checks AFTER each pass, so the body always runs at least once
do {
    "This runs even if the condition starts false"
} while ($false)

# do-until — same "runs at least once" shape, but inverted logic:
# do-while continues WHILE true; do-until continues UNTIL true (i.e. while
false)
do {
    "Also runs at least once"
} until ($true)
```

`do-while` and `do-until` are easy to swap by mistake specifically because their loop shape is identical — only the condition's meaning flips (continue *while true* vs. continue *until true*, i.e. while false). Reading the condition out loud in plain English before choosing which one to use avoids the mix-up.

for: The Classic Three-Part Loop

```
for ($i = 0; $i -lt 5; $i++) {
    "i is $i"
}
```

break & continue

Both work exactly as they do in most C-family languages — `break` exits the loop (or, per the warn-box above, a `switch` clause) entirely; `continue` skips straight to the next iteration:

```
foreach ($n in 1..10) {
    if ($n -eq 7) { break }       # stop the whole loop at 7
if ($n % 2 -eq 0) { continue } # skip even numbers
"$n"
}
# 1, 3, 5
```

Iterating a Hashtable, Properly

Chapter 5 introduced `@{}` / `[ordered]@{}` but deferred looping over one — here's the real pattern, using `.GetEnumerator()` rather than trying to `foreach` the hashtable directly:

```
$person = [ordered]@{ Name = "Alice"; Age = 30 }

foreach ($entry in $person.GetEnumerator()) {
    "$($entry.Key) = $($entry.Value)"
}
# Name = Alice
# Age = 30
```

A FIRST PRACTICAL HABIT

When choosing between `foreach` and `ForEach-Object`, ask one question: is this collection already sitting fully in memory (an array, a hashtable) or is it streaming out of a pipeline (`Get-ChildItem` on a huge tree, a database query)? Already-in-memory favors `foreach`; a live pipeline favors `ForEach-Object`.

Hands-On Exercises

EXERCISE 1

Using this chapter's own array-iteration example, explain why `switch (@(1, 2, 3, 4)) { {$_ % 2 -eq 0} {"$_ is even"} default {"$_ is odd"} }` produces four separate results rather than just one.

View solution

EXERCISE 2

Given `switch ($x) { {$_ -gt 0} {"positive"} {$_ -gt 10} {"big"} }` run with `$x = 15`, explain why **both** `"positive"` and `"big"` print. Then explain how you'd change the code so only `"big"` prints.

View solution

EXERCISE 3

Explain the practical difference between `foreach ($p in Get-Process) { ... }` and `Get-Process | ForEach-Object { ... }`. Describe a concrete situation where that difference would actually matter, not just a situation where it happens to be invisible.

View solution

CHAPTER 6 QUICK REFERENCE

- `if / elseif / else` — parentheses around the condition, braces around the body
- `switch` — supports `-Wildcard` / `-Regex`, auto-iterates an array; runs **every** matching clause unless you add `break`
- `foreach ($x in $collection)` — a loop keyword; evaluates the whole collection into memory first
- `ForEach-Object` **(Chapter 4)** — a pipeline cmdlet; streams one object at a time, never holding the whole set in memory
- `while` — checks before each pass, may run zero times
- `do-while` / `do-until` — always run at least once; inverted continue-condition logic (while true vs. until true)
- `for ($i=0; $i -lt N; $i++)` — the classic three-part loop
- `break` / `continue` — exit entirely / skip to next iteration; `break` also stops a `switch` clause from letting later clauses run
- `$hash.GetEnumerator()` — the correct way to `foreach` over a hashtable's key/value pairs
- **Next chapter:** Functions & Script Files (.ps1)

Everything so far has run one line, or one pipeline, at a time in an interactive session. This chapter is where PowerShell starts looking like a real programming language rather than a shell: packaging logic into reusable **functions**, giving them real typed parameters, and saving them into `.ps1` script files — plus one rule that catches almost everyone coming from a C-family language off guard, because it means `return` doesn't work quite the way you'd expect.

Basic Function Syntax

```
function Get-Greeting {
    param($Name)
    "Hello, $Name!"
}

Get-Greeting -Name "Philip" # named parameter
Get-Greeting "Philip" # positional — works too, in declaration order
```

Your own functions aren't required to follow Chapter 3's Verb-Noun convention, but it's worth adopting anyway — it's the difference between a function that reads clearly next to every built-in cmdlet, and one that stands out as obviously home-grown. Once functions start shipping inside a real module (Chapter 11), PowerShell will actually warn you if a function name uses an unapproved verb.

Parameters: Types, Defaults & Mandatory

```
function New-Greeting {
    param(
        [string]$Name = "World",

        [Parameter(Mandatory)]
        [int]$Age
    )
    "Hello, $Name - you are $Age years old."
}

New-Greeting -Age 42              # Hello, World - you are 42 years old.
New-Greeting -Name "Philip" -Age 42
New-Greeting "Philip" 42       # positional - same result, order matches param()
New-Greeting # PowerShell PROMPTS you for -Age - Mandatory guarantees it's never
missing
```

`[Parameter(Mandatory)]` doesn't throw immediately if you omit the value — it interactively prompts for it, which is a genuinely different failure mode from most languages simply erroring on a missing argument.

`$args`: Every Extra Positional Argument, With No `param()` at All

A function with no `param()` block isn't a function that takes no arguments — every positional argument you pass still lands somewhere, in the `$args` automatic variable flagged back in Chapter 5:

```
function Show-Args {
    $args
}

Show-Args 1 2 3
# 1
# 2
# 3
```

Return Values — The Single Biggest Habit to Unlearn

In most languages, only what follows an explicit `return` comes back from a function — everything else is just "stuff that ran." PowerShell doesn't work that way:

```
function Get-Doubled {
    param([int]$n)
    Write-Output "about to double $n" # this ALSO becomes part of the return
value
$n * 2
}

$result = Get-Doubled -n 5
$result
# about to double 5
# 10
$result.Count       # 2 — $result is an array holding BOTH values, not just 10
```

THE CENTRAL FACT THIS CHAPTER IS BUILT ON

Every un-suppressed line of output inside a function joins that function's return value — not just whatever follows `return`. `Write-Output "about to double $n"` was meant as a progress message, but because it was never captured, assigned, or redirected, it became a real, permanent part of what `Get-Doubled` actually returned. This is exactly the same object-pipeline behavior Chapter 1 built the whole course around, applied to a function body instead of a command line: anything that reaches the end of a statement uncaptured flows onward, whether that "onward" is the next pipeline stage or the function's own caller.

USE WRITE-HOST FOR MESSAGES THAT MUST NEVER POLLUTE THE RETURN VALUE

`Write-Host` writes straight to the console and never enters the output stream at all — unlike `Write-Output` (or a bare, uncaptured expression, which behaves identically to `Write-Output`), it can never accidentally become part of what a function returns. Reach for `Write-Host` specifically for progress/status messages a human should see but a caller should never receive as data. (`Write-Verbose`, suppressed by default unless `-Verbose` is passed, is another safe option — covered properly alongside error handling in Chapter 9.)

`return`: Exits Early, Optionally With One More Value

```
function Test-Positive {
    param([int]$n)
    if ($n -le 0) {
        return $false
    }
    return $true
}
```

`return`'s real job is stopping execution at that point — the value after it is just one more thing added to the output stream, exactly like any other uncaptured expression. It doesn't erase or override anything that was already output earlier in the function.

Script Files: Running vs. Dot-Sourcing

Save a function into a `.ps1` file, and how you invoke that file changes whether its contents are actually usable afterward:

```
# tools.ps1
function Get-Square { param([int]$n) $n * $n }

# Running it normally executes in a CHILD scope — Get-Square disappears once the
script ends
.\tools.ps1
Get-Square 4    # error — Get-Square isn't recognized out here
# Dot-sourcing (note the leading ". " with a space) runs it in the CURRENT scope
instead
. .\tools.ps1
Get-Square 4    # 16 — now it's genuinely available in this session
```

Dot-sourcing is how you'd load a personal library of reusable functions into your current session — Chapter 11's module system is the properly packaged, shareable version of the exact same idea.

> **A FIRST PRACTICAL HABIT**
>
> Before trusting any function's return value, assign it and check `.Count` the way this chapter's `$result.Count` example did. If it's more than 1 and you only expected one value back, an uncaptured `Write-Output` call (or a bare expression) is almost always the cause.

Hands-On Exercises

EXERCISE 1

A function does `Write-Output "Starting ... "` and then `return 42`. Explain exactly what `$x = MyFunc` assigns to `$x` — is it just `42`? Use this chapter's own `Get-Doubled` example to support your answer.

View solution

EXERCISE 2

Explain the difference between running `.\script.ps1` normally and dot-sourcing it with `. .\script.ps1`. Specifically, why is a function defined inside the script only callable afterward in the dot-sourced case?

View solution

EXERCISE 3

Write a function `Get-Square` that takes a mandatory `[int]` parameter and returns its square, with zero extra output polluting the return value. Explain how you'd actually verify — not just assume — that your function returns exactly one clean value.

View solution

CHAPTER 7 QUICK REFERENCE

- `function Verb-Noun { param( ... ) ... }` — basic function shape; Verb-Noun is a convention here, not enforced
- `[Parameter(Mandatory)]` — prompts interactively for a missing value, rather than erroring immediately
- `$args` — captures every positional argument when there's no `param()` block at all
- **Every uncaptured line joins the return value** — not just what follows `return`; this is the single biggest habit to unlearn
- `Write-Host` — console-only, never enters the output stream, safe for status messages
- `return` — exits early; the value after it is just one more thing added to the output, not a replacement for earlier output
- `.\script.ps1` — runs in a child scope; functions/variables don't persist afterward
- `. .\script.ps1` (dot-sourcing) — runs in the current scope; functions/variables persist in your session
- **Next chapter:** Working with Files, Text & Formatting Output

Reading and writing plain files is simple enough on its own — the real substance of this chapter is what happens at the two edges of the object pipeline. On one edge, `Format-Table` / `Format-List` exist purely to make objects readable on screen, and doing so deliberately breaks Chapter 1's own object pipeline on purpose. On the other edge, `ConvertTo-Json` / `ConvertTo-Csv` do the opposite job: turning real objects into genuine structured text that something else — a file, another program, a REST API — can consume and, mostly, reconstruct. Confusing the two is one of the most common mistakes in real-world PowerShell scripts.

Reading Files: Get-Content, and the -Raw Gotcha

```
# Default: an ARRAY of strings, one element per line
Get-Content .\log.txt
(Get-Content .\log.txt).Count      # the number of lines
# -Raw: a SINGLE string containing the whole file, newlines and all
Get-Content .\log.txt -Raw
(Get-Content .\log.txt -Raw).Count  # 1 — it's one string, not one-per-line
anymore
(Get-Content .\log.txt -Raw).Length # total character count instead
```

Without `-Raw`, `.Count` tells you how many lines the file has — exactly what you want for `foreach`-ing line by line. With `-Raw`, `.Count` is always `1`, because there's only one (much longer) string — what you want instead whenever a whole-file regex match (`-match`, Chapter 4) needs to see line breaks as part of the text it's searching, not as separators between array elements.

Writing Files: Set-Content, Add-Content & Out-File

```
"Line 1" | Set-Content .\out.txt     # overwrites the file
"Line 2" | Add-Content .\out.txt     # appends instead of overwriting
# Out-File captures the same FORMATTED text you'd see on screen - not the raw
objects
Get-Process | Out-File .\procs.txt   # a display-formatted table, subject to
console width
```

`Set-Content` and `Add-Content` write exactly the string content you give them. `Out-File` is different in a way that trips people up: piping objects straight into it captures whatever the console *would have displayed* — including the same automatic column formatting a wide table gets truncated to on screen — not a clean, structured dump of the underlying data.

Format-Table & Format-List: Display Only, Never Pipe Further

```
Get-Process | Format-Table Name, CPU -AutoSize
Get-Process | Format-List *
```

PIPING FORMAT-TABLE INTO ANYTHING ELSE SILENTLY BREAKS

`Get-Process | Format-Table Name, CPU | Where-Object { $_.CPU -gt 10 }` doesn't filter correctly — because by the time `Where-Object` receives anything, it's no longer receiving process objects at all. `Format-Table`/`Format-List` convert the pipeline into special internal *formatting instruction* objects meant only for a screen or a text file, and the real `CPU`/`Name` properties Chapter 3's `Get-Member` would have found on a process object simply aren't there anymore. The rule: a `Format-*` cmdlet should always be the **last** thing in a pipeline, followed only by `Out-Host`, `Out-File`, or nothing at all — filter and sort *before* formatting, never after.

THE CENTRAL FACT THIS CHAPTER IS BUILT ON

`Format-*` and `ConvertTo-*` look similar — both take real objects and turn them into text — but they exist for opposite purposes. `Format-Table`/`Format-List` are a deliberate, permanent exit from the object pipeline: display formatting for a human, with no path back. `ConvertTo-Json`/`ConvertTo-Csv` are a *bridge*: real, structured interchange formats designed specifically to be read back in — by `ConvertFrom-Json`, by `Import-Csv`, or by an entirely different program on the other end of an API call. Ask "is a human reading this on screen right now" versus "does this data need to leave PowerShell and come back later" before reaching for either family.

ConvertTo-Csv / Export-Csv: Real Structured Text

```
Get-Process | Select-Object Name, CPU -First 5 | Export-Csv .\procs.csv -
NoTypeInformation

$reloaded = Import-Csv .\procs.csv
$reloaded[0].CPU.GetType().Name    # String — NOT Double anymore!
```

CSV is plain text, and plain text has no concept of a number versus a string — every value that survives a round trip through `Export-Csv` / `Import-Csv` comes back as a plain `[string]`, even a column that started out as a real `[double]`. Comparing a re-imported value with `-gt 10` (Chapter 4) can behave unexpectedly for exactly this reason unless you explicitly cast it back — `[double]$reloaded[0].CPU -gt 10` — first.

ConvertTo-Json / ConvertFrom-Json: and the Depth Gotcha

```
$data = Get-Process | Select-Object Name, CPU -First 3
$json = $data | ConvertTo-Json
$json | ConvertFrom-Json # real objects again — PSCustomObject, close to the originals
# Default -Depth is only 2 — nested objects past that silently flatten to a plain string
$deeplyNested | ConvertTo-Json -Depth 10
```

`ConvertTo-Json`'s default depth of `2` is easy to hit without noticing — a nested object more than two levels deep gets silently collapsed into a string like `"System.Object[]"` instead of real, readable JSON. Any time nested data seems to have "gone missing" after converting to JSON, check `-Depth` before assuming the data was never there.

> **A FIRST PRACTICAL HABIT**
>
> Reach for `ConvertTo-Json` as a genuinely useful debugging tool even when you're not building any kind of API — `$someObject | ConvertTo-Json -Depth 5` shows you exactly what's really inside a complex object, in a readable structured form, faster than reading raw `Get-Member` output.

Hands-On Exercises

EXERCISE 1

Explain why `Get-Process | Format-Table Name, CPU | Where-Object { $_.CPU -gt 10 }` fails to filter correctly, using this chapter's own explanation of what `Format-Table` actually produces.

View solution

EXERCISE 2

Explain the difference between `Get-Content file.txt` and `Get-Content file.txt -Raw`. Describe a situation where using the wrong one would give you an incorrect line count or an incorrect character count.

View solution

EXERCISE 3

You export process data with `Export-Csv`, then read it back with `Import-Csv`. Explain why comparing a re-imported `CPU` value with `-gt 10` might not behave the way you expect, and how you'd fix it.

View solution

CHAPTER 8 QUICK REFERENCE

- `Get-Content` — array of lines by default; `-Raw` gives one whole-file string instead, changing what `.Count` means
- `Set-Content` / `Add-Content` — write exact string content; overwrite vs. append
- `Out-File` — captures the same display-formatted text the console would show, not a clean data dump
- `Format-Table` / `Format-List` — display only; converts objects into non-queryable formatting instructions — must be the last thing in a pipeline
- `Export-Csv` / `Import-Csv` — real structured text, but every value round-trips back as a plain `[string]`, even former numbers
- `ConvertTo-Json` / `ConvertFrom-Json` — real structured interchange; default `-Depth 2` silently flattens deeper nested data
- **This chapter's own throughline:** `Format-*` is a dead end for display; `ConvertTo-*` is a bridge back to real data
- **Next chapter:** Error Handling & Basic Debugging

`try` / `catch` looks exactly like it does in Java, JavaScript, or C# — which is precisely what makes PowerShell's own version so easy to get wrong the first time. Most built-in cmdlets don't actually stop and throw the way an exception would in those languages; they print a red error and simply move on to the next thing. A `catch` block wrapped around one of those cmdlets can sit there, correctly written, and never run at all. This chapter is about why, and about the one setting that fixes it.

Two Kinds of Errors: Terminating vs. Non-Terminating

PowerShell has two genuinely different error categories, and the distinction isn't cosmetic:

```
try {
    Get-Item C:\does-not-exist.txt
    "This line still runs!"
} catch {
    "This never prints — the catch block never fires"
}
# Get-Item's own error is NON-TERMINATING by default — PowerShell prints it in
red and moves on,
# which is why the very next line inside the try block still executed normally
```

Most cmdlet errors — a missing file, an unreachable path, a permission failure — are **non-terminating** by default: PowerShell displays the error and simply continues to the next statement, exactly as if nothing had interrupted it. `try` / `catch` only intercepts **terminating** errors — ones that actually stop execution at that point. A non-terminating error inside a `try` block just runs its course and lets the rest of the block keep going, right past the `catch` entirely.

-ErrorAction Stop: Making try/catch Actually Work

Nearly every cmdlet accepts an `-ErrorAction` parameter, and `Stop` is the one that converts a non-terminating error into a genuine terminating one `catch` can intercept:

```
try {
    Get-Item C:\does-not-exist.txt -ErrorAction Stop
    "This line never runs — the error above stopped execution"
} catch {
    "Caught it: $($_.Exception.Message)"
}
# Caught it: Cannot find path 'C:\does-not-exist.txt' because it does not exist.
```

-ERRORACTION VALUE	BEHAVIOR
Continue	Default — display the error, keep running (non-terminating; `catch` can't see it)
Stop	Turns the error terminating — the only value that reliably makes `catch` fire
SilentlyContinue	Suppress the display, keep running — the error still lands in `$Error` below
Ignore	Suppress the display and skip `$Error` too — the error leaves no trace at all
Inquire	Prompt interactively for what to do

> **THE CENTRAL FACT THIS CHAPTER IS BUILT ON**
>
> A `try`/`catch` block around a cmdlet call is not, by itself, real error handling — it's only real error handling once you've confirmed that cmdlet's failure is actually terminating, almost always by adding `-ErrorAction Stop` yourself. Skipping this is the single most common reason a PowerShell script's error handling silently does nothing: the code looks defensive, reads cleanly, and simply never activates.

`$_` Inside `catch`: A Completely Different Meaning

THE SAME SYMBOL, TWO UNRELATED MEANINGS

Chapter 4 established `$_` as "the current object flowing through the pipeline," inside `Where-Object` or `ForEach-Object`. Inside a `catch` block, `$_` means something entirely different: the **error record** that was just caught — not your data at all. `$_.Exception.Message` is how you pull the human-readable error text out of it. There's no relationship between these two uses beyond sharing the same variable name; which one applies is decided purely by which kind of block you're currently inside.

`$Error`: The Session's Own Error History

```
$Error[0]           # the most recent error, even ones you didn't try/catch
$Error.Count        # how many errors have accumulated this session
$Error.Clear()      # empty it out, e.g. before a section of a script you want to
check cleanly
```

`$Error` collects *every* error PowerShell has recorded this session — including `SilentlyContinue` ones from the table above — regardless of whether anything actually caught them. It's the automatic variable Chapter 5 flagged but deferred; this is its real, practical use.

`finally`: Always Runs

```
try {
    Get-Content .\data.txt -ErrorAction Stop
} catch {
    "Something went wrong: $($_.Exception.Message)"
} finally {
    "Done — this runs whether it succeeded or failed"
}
```

`finally` runs regardless of outcome — success, a caught error, or even an uncaught one — which makes it the right place for cleanup that must always happen (closing a connection, releasing a lock) rather than cleanup that only matters on failure.

Write-Verbose: The Safe Status Channel Promised in Chapter 7

Chapter 7 flagged `Write-Verbose` as a safer alternative to a bare, uncaptured status message — here's why: it's completely hidden by default, and only appears when the caller explicitly opts in with `-Verbose`:

```
function Get-Square {
    param([int]$n)
    Write-Verbose "Squaring $n"
$n * $n
}

Get-Square 4              # just 16 — the Write-Verbose line stays silent
Get-Square 4 -Verbose     # shows "VERBOSE: Squaring 4" AND still returns 16
```

Unlike `Write-Output` (Chapter 7), a `Write-Verbose` call never joins the function's return value under any circumstances — it writes to a genuinely separate stream that only `-Verbose` reveals.

> **A FIRST PRACTICAL HABIT**
>
> Whenever you write `try` / `catch` around a built-in cmdlet, add `-ErrorAction Stop` to that cmdlet call as a reflex — before assuming your error handling works, not after debugging why it silently didn't.

Hands-On Exercises

EXERCISE 1

Explain why wrapping `Get-Item C:\does-not-exist.txt` in `try`/`catch` (with no `-ErrorAction`) never triggers the `catch` block. What single change fixes it?

View solution

EXERCISE 2

Explain why `$_` means something completely different inside a `catch` block versus inside a `Where-Object` script block, referencing Chapter 4's own original meaning for `$_`.

View solution

EXERCISE 3

Write a `try`/`catch`/`finally` block that attempts `Get-Content -ErrorAction Stop` on a file, prints a friendly message using `$_.Exception.Message` in `catch`, and always prints `"Done"` in `finally` regardless of success or failure.

View solution

CHAPTER 9 QUICK REFERENCE

- **Non-terminating vs. terminating errors** — most cmdlet errors are non-terminating by default; only terminating errors reach `catch`
- `-ErrorAction Stop` — converts a non-terminating error into one `catch` can actually intercept
- `-ErrorAction SilentlyContinue / Ignore / Inquire` — suppress display (still logs to `$Error`) / suppress entirely / prompt
- `$_.Exception.Message` — the readable error text, inside a `catch` block only
- `$_` **inside** `catch` — the caught error record, a completely different meaning from `$_` inside `Where-Object` / `ForEach-Object` (Chapter 4)
- `$Error[0]` **/** `$Error.Count` **/** `$Error.Clear()` — the session's running error history
- `finally` — always runs, success or failure — the right place for guaranteed cleanup
- `Write-Verbose` — hidden by default, shown with `-Verbose`, never joins the return value
- **Next chapter:** Execution Policy & Script Security Basics

Chapter 7 already flagged this moment: the first time you try to run a local `.ps1` file, PowerShell may refuse outright with a message about scripts being disabled on the system. That's **execution policy** — and the single most important thing to understand about it, stated plainly by Microsoft's own documentation, is also the most counterintuitive: it is *not* a security boundary. It's a safety rail against running something by accident, not a lock that keeps a determined attacker out.

The Policy Levels

POLICY	WHAT IT ALLOWS
Restricted	No scripts at all — only interactive commands typed directly. The factory default on client Windows.
AllSigned	Every script must carry a valid digital signature — even ones you wrote yourself, on your own machine.
RemoteSigned	Locally created scripts run freely; scripts that came from the internet must be signed. The default on Windows Server.
Unrestricted	Everything runs, but a downloaded script shows a one-time confirmation prompt before executing.
Bypass	Nothing is blocked, no warnings shown at all — typically used deliberately for a single automated task, not as a standing setting.

Get-ExecutionPolicy / Set-ExecutionPolicy & Scope

```
Get-ExecutionPolicy -List

# Scope         ExecutionPolicy
# ─────         ───────────────
# MachinePolicy       Undefined
# UserPolicy          Undefined
# Process             Undefined
# CurrentUser         Undefined
# LocalMachine        Restricted     ← the actual effective policy here, since
nothing narrower is set
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
```

Multiple scopes can hold a policy at once — `MachinePolicy` and `UserPolicy` (set by Group Policy, taking precedence over everything else), then `Process`, `CurrentUser`, and `LocalMachine` in that order. The narrowest scope with a real (non-`Undefined`) value wins.

A Temporary Bypass, Scoped to One Session

```
Set-ExecutionPolicy -ExecutionPolicy Bypass -Scope Process
.\trusted-script.ps1
```

`-Scope Process` only affects the current PowerShell window — it evaporates the moment that window closes, leaving `LocalMachine`'s own policy completely untouched. This is the right tool for "let this one trusted script run right now" without permanently loosening anything.

Mark of the Web: What "Remote" in RemoteSigned Actually Means

`RemoteSigned`'s distinction between "local" and "downloaded" isn't a guess based on file location — Windows itself tags any file that arrived via a browser, email client, or similar with a hidden NTFS marker called the **Mark of the Web** (technically a `Zone.Identifier` alternate data stream). A script you typed and saved yourself never gets this tag; one you downloaded does, which is exactly what `RemoteSigned` checks for.

```
# Removes the Mark of the Web from a script you've reviewed and trust
Unblock-File -Path .\downloaded-script.ps1
```

EXECUTION POLICY IS NOT A SECURITY BOUNDARY

Microsoft's own documentation states this directly, in those words. It's trivially bypassed by anyone who actually wants to — pasting a script's contents straight into an interactive prompt, running `powershell.exe -EncodedCommand`, or piping text into `Invoke-Expression` all skip execution policy entirely, none of them counting as "running a script file." Execution policy exists to stop *accidental* execution — double-clicking an unfamiliar `.ps1`, or a script quietly shipped inside something else running without you noticing — not to stop someone determined to run code on a machine they already have access to.

THE CENTRAL FACT THIS CHAPTER IS BUILT ON

Execution policy answers "*did I mean to run this?*" — a speed bump requiring deliberate action before a script fires. It does not answer "*is this actually safe?*", and was never designed to. Real protection against malicious code comes from script signing with a genuinely trusted certificate, OS-level access controls, and — in a locked-down enterprise environment — tools like Constrained Language Mode and AppLocker, all beyond this course's own fundamentals scope. Treat execution policy as a helpful habit-forming guardrail, not a substitute for actually reading a script before running it.

A Practical Default

For a personal development machine, `RemoteSigned` is the sane, commonly recommended setting: your own scripts run without friction, while anything that arrived from the internet still needs either a real signature or a conscious `Unblock-File` first — a reasonable balance between convenience and the "did I mean to run this" habit this whole chapter is really about.

> **A FIRST PRACTICAL HABIT**
>
> When a script refuses to run, check `Get-ExecutionPolicy -List` before assuming why — it's easy to fix the wrong scope (loosening `LocalMachine` when a stricter `UserPolicy` set by Group Policy is the actual blocker) if you don't first see which scope is genuinely in control.

Hands-On Exercises

EXERCISE 1

Explain why `RemoteSigned` lets a script you wrote yourself run freely while blocking one you downloaded, even if both files sit in the same folder. What specific mechanism (named in this chapter) is actually responsible for a file counting as "remote"?

View solution

EXERCISE 2

Explain why Microsoft describes execution policy as "not a security boundary." Name one concrete way, from this chapter, that someone could run PowerShell code while completely bypassing it.

View solution

EXERCISE 3

You need to run a single trusted script once without permanently changing your machine's execution policy. Write the command(s) you'd use, and explain why `-Scope Process` is the right choice here rather than `-Scope LocalMachine`.

View solution

CHAPTER 10 QUICK REFERENCE

- **Restricted** — no scripts at all (client Windows default); **AllSigned** — every script needs a signature; **RemoteSigned** — local scripts free, remote ones need a signature (Windows Server default); **Unrestricted** — everything runs with a warning on remote scripts; **Bypass** — nothing blocked
- `Get-ExecutionPolicy -List` — shows every scope; the narrowest defined one wins
- `Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass` — a temporary, session-only override
- **Mark of the Web /** `Zone.Identifier` — the hidden NTFS tag that actually makes a file count as "remote" for `RemoteSigned`
- `Unblock-File` — removes the Mark of the Web from a script you've reviewed and trust
- **Execution policy is not a security boundary** — trivially bypassed via `-EncodedCommand`, `Invoke-Expression`, or pasting directly into the console; it prevents accidental execution only
- **Practical default:** `RemoteSigned` for a personal dev machine
- **Next chapter:** Modules, Aliases & the PowerShell Gallery

Chapter 1 opened this entire course on a promise: PowerShell aliases are just names you can define yourself, so the same trick that gives a real Linux terminal an `ll` shortcut is available here too. This chapter finally delivers on that — honestly, including the one place a plain alias genuinely can't do what a Bash alias can. Along the way, it covers the mechanism that lets any of this scale past what ships built-in: modules, and the public PowerShell Gallery they can be installed from.

What a Module Actually Is

A **module** is a packaged bundle of cmdlets, functions, and aliases — the same organizing unit a Python package or an npm package is for their own ecosystems, just for PowerShell commands specifically. Every built-in cmdlet used across this entire course already lives inside one:

```
Get-Module -ListAvailable | Select-Object Name, Version -First 5

# Chapter 3's own Get-Command search works against modules too
Get-Command -Module Microsoft.PowerShell.Management
```

Most built-in modules **auto-load** the first time you call one of their commands — which is exactly why nothing in this course has needed an explicit `Import-Module` so far. When a module isn't in a discoverable path (or you want to be explicit), `Import-Module <name>` loads it into the current session directly.

The PowerShell Gallery: Installing Modules Someone Else Wrote

The **PowerShell Gallery** (PSGallery) is the public, central repository for community and Microsoft-published modules — PowerShell's own equivalent of PyPI or npm:

```
Find-Module -Name Az -Repository PSGallery
Install-Module -Name Az -Scope CurrentUser
Update-Module -Name Az
```

`-Scope CurrentUser` installs just for your own account, no administrator rights required — the sane default unless a module genuinely needs to be available to every user on the machine.

> **THE CENTRAL FACT THIS CHAPTER IS BUILT ON**
>
> Chapter 3 trained you to discover anything already installed with `Get-Command` / `Get-Help` / `Get-Member` rather than memorizing it. The PowerShell Gallery is the exact same idea, one layer out — `Find-Module` discovers code you haven't installed yet, the same way `Get-Command` discovers cmdlets you haven't used yet. Neither one requires memorizing anything in advance; both are search tools built directly into the shell.

Set-Alias — And a Real Limit It Has

```
Set-Alias -Name ll -Value Get-ChildItem
ll # runs exactly Get-ChildItem — no extra flags baked in
```

A POWERSHELL ALIAS CAN'T CARRY DEFAULT ARGUMENTS THE WAY A BASH ALIAS CAN

In Bash, `alias ll='ls -la'` bundles both the command *and* its flags into one name. PowerShell's `Set-Alias` only ever maps a name to a single command — it has no way to attach `-Force` or any other default parameter to that name. `Set-Alias -Name ll -Value Get-ChildItem` makes `ll` behave *exactly* like bare `Get-ChildItem`, with none of the "show hidden items too" behavior a Bash user would reasonably expect from a real `ll`.

The `ll` Payoff: A Function, Not Just an Alias

Getting genuine Bash-`ll` behavior — a short name *with* default flags baked in — needs Chapter 7's own function syntax instead, using `@args` (Chapter 7) to still accept any extra arguments a caller adds on:

```
function ll {
    Get-ChildItem -Force @args
}

ll # Get-ChildItem -Force — shows hidden items too, like Bash's ls -la
ll -Filter "*.txt" # extra arguments still pass straight through via @args
```

This is the honest answer to Chapter 1's own question: PowerShell can absolutely give you an `ll` shortcut with real default behavior — it just isn't `Set-Alias` that does it. A short function is the actual tool.

Making It Permanent: `$PROFILE`

Both approaches above vanish the moment the current session closes (Chapter 7's own scoping rules apply here too) — unless they're saved into your **profile**, a `.ps1` script PowerShell runs automatically every time a new session starts:

```
$PROFILE # the path to your own profile script — may not exist yet
if (-not (Test-Path $PROFILE)) {
    New-Item -ItemType File -Path $PROFILE -Force
}

notepad $PROFILE # add the ll function here — now it's there every time you open PowerShell
```

> **A FIRST PRACTICAL HABIT**
>
> Anything you find yourself retyping at the start of every PowerShell session — a personal alias, a small function like `ll`, a default working directory — belongs in `$PROFILE`, not repeated by hand. It's the same "load a library of reusable functions" idea Chapter 7 introduced with dot-sourcing, just running automatically instead of by hand.

Hands-On Exercises

EXERCISE 1

Explain why `Set-Alias -Name ll -Value Get-ChildItem` does not reproduce Bash's `alias ll='ls -la'` behavior. What actual PowerShell mechanism is needed to get real default-flags behavior, and why does that mechanism work where a plain alias can't?

View solution

EXERCISE 2

Explain the difference between an `ll` function defined interactively in your current session and one added to `$PROFILE`. Specifically, why does one disappear when the window closes while the other doesn't?

View solution

EXERCISE 3

You want to install a module from the PowerShell Gallery, use one of its cmdlets, then confirm which module actually owns that cmdlet. Write the sequence of commands you'd use, referencing earlier chapters' own discovery tools where relevant.

View solution

CHAPTER 11 QUICK REFERENCE

- **Module** — a packaged bundle of cmdlets/functions/aliases; most built-in ones auto-load on first use
- `Get-Module -ListAvailable` **/** `Import-Module` — see what's installed / load one explicitly
- **PowerShell Gallery (PSGallery)** — the public module repository; `Find-Module` / `Install-Module -Scope CurrentUser` / `Update-Module`
- `Set-Alias` — maps a name to a command only; it *cannot* bake in default parameters, unlike a Bash `alias`
- **Real** `ll` **-with-defaults behavior** — needs a function (`function ll { Get-ChildItem -Force @args }`), not `Set-Alias`
- `$PROFILE` — your personal startup script; anything defined there persists across every future session
- **Next chapter:** Capstone — Automating a Real Admin Task

Dana just started as a junior sysadmin, and a shared logs folder is quietly filling up a server's disk. Before deleting anything, Dana wants a clean report of exactly what's old and large enough to matter — then, once it looks right, a safe way to actually clear it out, repeatable next time this happens without retyping everything by hand. Building that one real tool, `Get-OldFileReport`, touches every chapter in this course, in the order a genuine first attempt would actually hit them.

STEP 1 — CHECKING THE ENVIRONMENT FIRST

Before writing anything, Dana checks how much room is actually at stake, using Chapter 2's own provider knowledge — `Get-PSDrive` works here exactly the way it did against the registry back then, just against the `FileSystem` provider this time:

```
Get-PSDrive C | Select-Object Used, Free
```

Unsure of the exact cmdlet for safely deleting a file, Dana reaches for Chapter 3's own discovery habit rather than guessing:

```
Get-Command -Verb Remove -Noun Item
```

STEP 2 — FINDING THE OLD, LARGE FILES

The core of the whole tool is one object pipeline, straight from Chapters 1 and 4 — real properties (`LastWriteTime`, `Length`), no text parsing anywhere:

```
Get-ChildItem -Path $Path -Recurse -File |
    Where-Object { $_.LastWriteTime -lt $cutoff -and $_.Length -gt
$sizeBytes } |
    Sort-Object Length -Descending
```

STEP 3 — TYPED, HASHTABLE-BACKED CONFIGURATION

Rather than burying magic numbers in the pipeline itself, defaults live in an `[ordered]@{}` hashtable (Chapter 5), with the actual thresholds cast to real types before use — `$cutoff` a genuine `[datetime]`, `$sizeBytes` a genuine `[int64]` of bytes, not megabytes:

```
$DefaultConfig = [ordered]@{ DaysOld = 30; SizeThresholdMB = 10 }

[datetime]$cutoff    = (Get-Date).AddDays(-$DaysOld)
[int64]$sizeBytes = $SizeThresholdMB * 1MB
```

STEP 4 — CLASSIFYING WHAT WAS FOUND

Chapter 6's `switch -Wildcard` sorts each match into a category, with `break` unnecessary here since each file only has one extension to match:

```
$category = switch -Wildcard ($file.Extension) {
    ".log" { "Log" }
    ".tmp" { "Temp" }
    ".bak" { "Backup" }
    default   { "Other" }
}
```

STEP 5 — A WELL-BEHAVED FUNCTION

Everything gets wrapped in a real function with a proper `param()` block (Chapter 7) — and Chapter 7's own biggest lesson is respected deliberately: every progress message uses `Write-Verbose`, never a bare or `Write-Output` line, so nothing pollutes the one clean value the function actually returns.

STEP 6 — A CLEAN REPORT, NOT A SCREEN DUMP

Chapter 8's warning is respected directly: the report is built with `Export-Csv` and `ConvertTo-Json`, never `Format-Table`, since this data needs to survive being read back in later:

```
$results | Export-Csv -Path "$ReportPath.csv" -NoTypeInformation
$results | ConvertTo-Json -Depth 3 | Set-Content "$ReportPath.json"
```

STEP 7 — DELETING SAFELY

Only once the report looks right does Dana add real deletion — wrapped in Chapter 9's own `try` / `catch` / `finally`, with `-ErrorAction Stop` so a locked or permission-denied file is actually caught rather than silently skipped:

```
foreach ($file in $results) {
    try {
        Remove-Item -Path $file.FullPath -ErrorAction Stop
        Write-Verbose "Deleted $($file.FullPath)"
    } catch {
        Write-Warning "Couldn't delete $($file.FullPath):
$($_.Exception.Message)"
    } finally {
        Write-Verbose "Finished handling $($file.Name)"
    }
}
```

STEP 8 — RUNNING IT SAFELY, AND KEEPING IT AROUND

A colleague emails Dana the finished `.ps1` for a second opinion. Running it back fails outright — Chapter 10's own Mark of the Web, since it arrived as an email attachment. `Unblock-File .\Get-OldFileReport.ps1` clears it, confirmed safe by review first. Finally, Chapter 11's own lesson closes the loop: Dana dot-sources the script into `$PROFILE`, so `Get-OldFileReport` is simply available every time a new session opens, without retyping or re-navigating to the file — the exact same durability Chapter 11 gave the `ll` function now applied to a real production tool.

The Complete Script

```
function Get-OldFileReport {
    param(
        [Parameter(Mandatory)]
        [string]$Path,

        [int]$DaysOld = 30,
        [double]$SizeThresholdMB = 10,
        [string]$ReportPath = ".\old-file-report",
        [switch]$DeleteAfterReport
    )

    [datetime]$cutoff    = (Get-Date).AddDays(-$DaysOld)
    [int64]$sizeBytes = $SizeThresholdMB * 1MB

    Write-Verbose "Scanning $Path for files older than $DaysOld days, larger
than $SizeThresholdMB MB"
$matches = Get-ChildItem -Path $Path -Recurse -File |
        Where-Object { $_.LastWriteTime -lt $cutoff -and $_.Length -gt
$sizeBytes } |
        Sort-Object Length -Descending

    $results = foreach ($file in $matches) {
        $category = switch -Wildcard ($file.Extension) {
            ".log" { "Log" }
            ".tmp" { "Temp" }
            ".bak" { "Backup" }
            default   { "Other" }
        }
```

```
        [PSCustomObject]@{
            Name          = $file.Name
            FullPath      = $file.FullName
            Category      = $category
            SizeMB        = [math]::Round($file.Length / 1MB, 2)
            LastWriteTime = $file.LastWriteTime
        }
    }

    $results | Export-Csv -Path "$ReportPath.csv" -NoTypeInformation
    $results | ConvertTo-Json -Depth 3 | Set-Content "$ReportPath.json"
Write-Verbose "Report written: $ReportPath.csv / $ReportPath.json
($($results.Count) files found)"
if ($DeleteAfterReport) {
        foreach ($file in $results) {
            try {
                Remove-Item -Path $file.FullPath -ErrorAction Stop
                Write-Verbose "Deleted $($file.FullPath)"
            } catch {
                Write-Warning "Couldn't delete $($file.FullPath):
$($_.Exception.Message)"
            } finally {
                Write-Verbose "Finished handling $($file.Name)"
            }
        }
    }

    $results
}

# A dry-run report only - no deletion, exactly one clean value returned
Get-OldFileReport -Path "D:\Logs" -DaysOld 30 -SizeThresholdMB 10 -Verbose

# Once the report looks right - report AND delete
Get-OldFileReport -Path "D:\Logs" -DaysOld 30 -SizeThresholdMB 10 -
DeleteAfterReport
```

Chapter Attribution

STEP	CHAPTER(S) APPLIED
1 — Checking the environment	Chapter 2 (`Get-PSDrive`), Chapter 3 (`Get-Command` discovery)
2 — The core pipeline	Chapter 1 (object pipeline), Chapter 4 (`Where-Object` / `Sort-Object`)
3 — Configuration	Chapter 5 (`[ordered]@{}`, type casting)
4 — Classification	Chapter 6 (`switch -Wildcard`)
5 — The function itself	Chapter 7 (`param()`, `Write-Verbose` over uncaptured output)
6 — The report	Chapter 8 (`Export-Csv` / `ConvertTo-Json`, never `Format-Table`)
7 — Safe deletion	Chapter 9 (`try` / `catch` / `finally`, `-ErrorAction Stop`)
8 — Running & keeping it	Chapter 10 (`Unblock-File`), Chapter 11 (`$PROFILE`)

WHAT THIS WHOLE COURSE WAS REALLY ABOUT

From Chapter 1's first `Get-ChildItem` to this capstone's last `Remove-Item`, not one step ever fell back to parsing text. Discovery (Chapter 3), filtering (Chapter 4), configuration (Chapter 5), classification (Chapter 6), a well-behaved function (Chapter 7), and a clean exported report (Chapter 8) all stayed real, typed objects the entire way through — the exact claim Chapter 1 opened this course with, now carrying a genuine production script rather than a single example line.

HONEST SCOPE NOTE

This capstone deliberately stops short of full automation. It doesn't cover scheduling `Get-OldFileReport` to run unattended (Task Scheduler is OS-level territory, not a PowerShell Fundamentals topic), running it against several remote servers at once (PowerShell Remoting is PowerShell Intermediate/Advanced's own Chapter 3), or delivering the report anywhere automatically (email/Teams integration is a separate API concern). What's here is a genuinely real, safe, reusable tool — not yet an unattended one.

Hands-On Exercises

EXERCISE 1

Explain why `Get-OldFileReport` uses `Write-Verbose` for every progress message instead of a bare string or `Write-Output`. What would go wrong with the function's return value if a stray `Write-Output` line were added inside the `foreach` loop, referencing Chapter 7's own explanation?

View solution

EXERCISE 2

Explain why the deletion step (Step 7) uses `-ErrorAction Stop` on `Remove-Item` inside its `try` block, referencing Chapter 9's own explanation of terminating vs. non-terminating errors. What would happen to a locked file if `-ErrorAction Stop` were left off?

View solution

EXERCISE 3

Dana's colleague emails the finished script, and running it fails until `Unblock-File` is used. Explain exactly why, referencing Chapter 10's own explanation of what actually makes a file count as "remote."

View solution

CHAPTER 12 QUICK REFERENCE — COURSE COMPLETE

- **8 build steps, 11 prior chapters** — one real, reusable admin tool, built in the order a genuine first attempt would actually hit each concept
- **This course's own throughline, closed out:** real, typed .NET objects flowing cleanly from discovery through filtering, configuration, classification, execution, reporting, and safe deletion — never once falling back to parsing text
- **Honest scope note:** no unattended scheduling, no multi-server remoting, no automated delivery — a real tool, not yet a fully unattended one
- **The PowerShell Fundamentals course is now complete** — 12/12 chapters