WINDOWS

PowerShell Intermediate/Advanced

Advanced Functions, Regex & Remoting

Parallel Execution, .NET/COM & CIM

APIs, Modules & Desired State Configuration

Testing with Pester & Real Productivity

Capstone: A Multi-System Automation & Reporting Tool

Philip Osztromok

Generated with Claude

Table of Contents

PowerShell Intermediate/Advanced — 12 Chapters

PowerShell Fundamentals covered functions honestly, but deliberately kept them simple: a `param()` block, a type, maybe `[Parameter(Mandatory)]`, and — as the capstone's own `Get-OldFileReport` showed — real discipline about never letting stray output pollute the return value. That's genuinely enough for scripts you write and run yourself. This course exists for the next step: functions built to behave like real, first-class cmdlets — with automatic parameters, real input validation, native pipeline support, and the same `-WhatIf` safety net `Remove-Item` itself has. It starts with the one attribute that unlocks all of it.

`[CmdletBinding()]`: Turning a Function Into a Real Cmdlet

Adding `[CmdletBinding()]` immediately after `function Name {` — before `param()` — promotes a plain function into what PowerShell calls an **advanced function**, and the built-in cmdlets used throughout Fundamentals have always secretly worked this way:

```
function Get-Square {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory)]
        [int]$n
    )
    $n * $n
}
```

That one attribute adds a whole family of **CommonParameters** automatically — `-Verbose`, `-Debug`, `-ErrorAction`, `-WarningAction`, `-ErrorVariable`, `-WarningVariable`, `-OutVariable`, `-OutBuffer`, and `-PipelineVariable` — with zero extra `param()` entries needed:

```
(Get-Command Get-Square).Parameters.Keys

# n
# Verbose
# Debug
# ErrorAction
# WarningAction
# ErrorVariable
#  ...and the rest, all for free
```

This is exactly what made `-Verbose` and `-ErrorAction Stop` feel like built-in language features throughout Fundamentals — every cmdlet you called there had `[CmdletBinding()]` (or its C#-level equivalent) applied the whole time.

Parameter Validation Attributes

Fundamentals' own `[Parameter(Mandatory)]` only checked that a value was supplied at all. Validation attributes check that the value supplied is actually *acceptable*, rejecting bad input before the function body ever runs:

ATTRIBUTE	CHECKS
[ValidateNotNullOrEmpty()]	Rejects `$null` or an empty string/collection
[ValidateRange(1, 100)]	Numeric value must fall within the given range
[ValidateSet("Low","Medium","High")]	Value must be one of a fixed, explicit list
[ValidatePattern('^\d{5}$')]	Value must match a regular expression
[ValidateScript({ Test-Path $_ })]	Value must pass an arbitrary custom script block
[ValidateCount(1, 5)]	An array parameter must have between the given number of elements

```
function Set-Priority {
    [CmdletBinding()]
    param(
        [ValidateSet("Low", "Medium", "High")]
        [string]$Level,

        [ValidateRange(1, 100)]
        [int]$Percent
    )
    "$Level at $Percent%"
}

Set-Priority -Level "Urgent" -Percent 50
# error, immediately — "Urgent" isn't in the ValidateSet list; the function body never even runs
```

Pipeline Input: `ValueFromPipeline` vs. `ValueFromPipelineByPropertyName`

A Fundamentals-style function only ever received arguments passed explicitly by name or position. An advanced function can accept objects piped directly in, in two genuinely different ways:

```
function Test-ByWholeObject {
    [CmdletBinding()]
    param(
        [Parameter(ValueFromPipeline)]
        [string]$Name # the ENTIRE piped-in value becomes $Name
    )
    process { "Got: $Name" }
}
"Alice", "Bob" | Test-ByWholeObject
function Test-ByProperty {
    [CmdletBinding()]
    param(
        [Parameter(ValueFromPipelineByPropertyName)]
        [string]$ProcessName # matched by PROPERTY NAME on the incoming object
    )
    process { "Got: $ProcessName" }
}
Get-Process | Test-ByProperty # works because Get-Process objects have a real
ProcessName property
```

`ValueFromPipeline` is right when the whole piped-in object *is* the value you want. `ValueFromPipelineByPropertyName` is right when you're receiving richer objects (like Chapter 3's own `Get-Process` output) and only want to pull one matching property off each one — exactly the kind of binding that makes so many built-in cmdlets pipe together without you ever manually extracting a property first.

Why Pipeline-Bound Functions Need a `process {}` Block

```powershell
# WITHOUT process{} — a real, common bug
function Show-NameBroken {
    [CmdletBinding()]
    param([Parameter(ValueFromPipeline)][string]$Name)
    "Name is: $Name" # NOT inside process{}
}
"Alice", "Bob", "Carol" | Show-NameBroken
# Name is: Carol      — only ONE line, only the LAST object
# WITH process{} — correct
function Show-NameFixed {
    [CmdletBinding()]
    param([Parameter(ValueFromPipeline)][string]$Name)
    process { "Name is: $Name" }
}
"Alice", "Bob", "Carol" | Show-NameFixed
# Name is: Alice
# Name is: Bob
# Name is: Carol
```

THE CENTRAL FACT THIS CHAPTER IS BUILT ON

A function body written as one plain block — with no `begin` / `process` / `end` — runs exactly **once**, no matter how many objects the pipeline sends it, and by the time that single run executes, a pipeline-bound parameter only still holds the *last* object received. `begin {}` runs once, before the first pipeline object arrives — the right place for one-time setup. `process {}` runs once *per* pipeline object — the only block that actually sees every item. `end {}` runs once, after the last object has been processed — the right place for a final summary. Fundamentals' own functions never needed this distinction because none of them declared `ValueFromPipeline` — the moment a parameter does, `process {}` stops being optional.

`SupportsShouldProcess`: Real `-WhatIf` / `-Confirm` Support

The Fundamentals capstone leaned on `Remove-Item`'s own built-in safety — this is how to give a function you write that exact same behavior:

```
function Remove-TempFile {
    [CmdletBinding(SupportsShouldProcess)]
    param(
        [Parameter(Mandatory, ValueFromPipeline)]
        [string]$Path
    )
    process {
        if ($PSCmdlet.ShouldProcess($Path, "Delete")) {
            Remove-Item -Path $Path -ErrorAction Stop
        }
    }
}

Remove-TempFile -Path "C:\temp\old.log" -WhatIf
# What if: Performing the operation "Delete" on target "C:\temp\old.log".
# Nothing is actually deleted — exactly like calling Remove-Item itself with -
WhatIf
```

`$PSCmdlet.ShouldProcess()` is only meaningful once `SupportsShouldProcess` is declared — it automatically wires up both `-WhatIf` (preview only, shown above) and `-Confirm` (an interactive yes/no prompt before each action), with zero extra parameter declarations of your own.

> **A FIRST PRACTICAL HABIT**
>
> Any function you write that deletes, overwrites, or otherwise changes something irreversibly is worth reaching for `[CmdletBinding(SupportsShouldProcess)]` on by default — the same instinct that made Fundamentals' own capstone wrap `Remove-Item` in `try` / `catch`, just built into the function itself this time.

A MISSING PROCESS{} BLOCK FAILS SILENTLY, NOT LOUDLY

`Show-NameBroken` above didn't throw an error — it just quietly processed one object instead of three, with no warning that anything was wrong. This is exactly the kind of bug that survives a quick manual test (where you might only ever pipe in one item) and then breaks the moment it meets a real, multi-item pipeline in production. Any parameter marked `ValueFromPipeline` or `ValueFromPipelineByPropertyName` is a signal to double-check that a `process {}` block actually exists.

Hands-On Exercises

EXERCISE 1

Explain why `Show-NameBroken` only prints one line ("Carol") when three names are piped into it, using this chapter's own explanation of how a plain function body executes versus a `process {}` block.

View solution

EXERCISE 2

Explain the difference between `ValueFromPipeline` and `ValueFromPipelineByPropertyName`. Using this chapter's own `Get-Process | Test-ByProperty` example, explain specifically why `ValueFromPipeline` wouldn't have worked correctly there instead.

View solution

EXERCISE 3

Write a function `Remove-OldLog` that accepts a mandatory, pipeline-bound `[string]$Path`, supports `-WhatIf` / `-Confirm` via `SupportsShouldProcess`, and correctly processes every object piped into it (not just the last one). Explain how your function avoids both gotchas covered in this chapter.

View solution

CHAPTER 1 QUICK REFERENCE

- `[CmdletBinding()]` — promotes a plain function to an advanced function; unlocks CommonParameters (`-Verbose`, `-Debug`, `-ErrorAction`, etc.) automatically
- **Validation attributes** — `[ValidateNotNullOrEmpty()]` `[ValidateRange()]` `[ValidateSet()]` `[ValidatePattern()]` `[ValidateScript()]` `[ValidateCount()]`, checked before the function body ever runs
- `ValueFromPipeline` — the entire piped object becomes the parameter's value
- `ValueFromPipelineByPropertyName` — binds by matching a property name on the incoming object to the parameter name
- `begin {} / process {} / end {}` — once before / once per pipeline object / once after; a pipeline-bound parameter needs `process {}` or it silently only sees the last item
- `[CmdletBinding(SupportsShouldProcess)]` + `$PSCmdlet.ShouldProcess()` — real, built-in `-WhatIf`/`-Confirm` support, the same safety `Remove-Item` itself has
- **Next chapter:** Regular Expressions & Advanced String/Text Processing

PowerShell Fundamentals 4 introduced `-match` as one comparison operator among several — enough to filter a pipeline, not enough to actually use regular expressions well. This chapter goes back to `-match` and gives it real depth: the automatic variable it quietly populates, capturing named pieces out of a match, regex-powered find-and-replace, and — the chapter's own central gotcha — exactly why `-match` alone can't find every match in a string, no matter how many times that string actually contains one.

`-match` & `$Matches`: A Comparison That Remembers What It Found

Unlike every other comparison operator from Fundamentals 4, a successful `-match` does something extra: it populates the `$Matches` automatic variable with the details of what actually matched.

```
"The year is 2024" -match '\d{4}' # True
$Matches[0]                       # "2024" — the whole match
```

Capture Groups: Numbered and Named

```
# Numbered groups — accessed by position, 1-based ($Matches[0] is always the
whole match)
"2024-08-05" -match '(\d{4})-(\d{2})-(\d{2})'
$Matches[1]   # 2024
$Matches[2]   # 08
# Named groups — (?<name>...) — accessed by name, often much more readable
"John Smith" -match '(?<first>\w+)\s(?<last>\w+)'
$Matches['first']   # John
$Matches['last']    # Smith
```

`-replace`: Regex-Based Replacement With Backreferences

```
"2024-08-05" -replace '(\d{4})-(\d{2})-(\d{2})', '$2/$3/$1'
# 08/05/2024 - $1/$2/$3 refer back to the numbered capture groups above
```

`-replace`'s replacement string reads the same capture groups `-match` would have populated into `$Matches` — just referenced as `$1`, `$2`, `$3` (or `${name}` for a named group) directly inside the replacement text itself, rather than through a separate variable.

Case Sensitivity: Insensitive by Default

Consistent with every comparison operator back in Fundamentals 4, `-match` and `-replace` are case-**insensitive** by default — a genuinely different default from most languages' own regex engines. A `-c`-prefixed variant exists for the rare case a real case-sensitive match is actually needed:

```
"HELLO" -match "hello" # True  — case-insensitive by default
"HELLO" -cmatch "hello" # False — the case-sensitive variant
"HELLO" -creplace "hello", "hi" # no match, nothing replaced
```

When `-match` Isn't Enough: `[regex]::Matches()`

```
$text = "Contact: alice@example.com or bob@example.org"
$text -match '\w+@\w+\.\w+'
$Matches[0]   # alice@example.com — only the FIRST match, the second one is
nowhere to be found
# [regex]::Matches finds EVERY match in the string, not just one
[regex]::Matches($text, '\w+@\w+\.\w+') | ForEach-Object { $_.Value }
# alice@example.com
# bob@example.org
```

THE CENTRAL FACT THIS CHAPTER IS BUILT ON

`-match` against a single string only ever tells you two things: whether a match exists at all, and — if so — the details of exactly **one** match, in `$Matches`. It was never designed to enumerate every occurrence inside a longer string, no matter how many times that pattern genuinely appears. `[regex]::Matches()` — the real .NET method underneath, reached via the `[regex]` type accelerator — returns a full collection of every match found, each with its own `.Value` and its own `.Groups`. Reach for `-match` when you're testing a whole string against a pattern; reach for `[regex]::Matches()` the moment "how many times does this appear" or "find every instance" is the actual question.

$MATCHES GETS SILENTLY OVERWRITTEN BY THE NEXT -MATCH

`$Matches` isn't scoped to one expression — it's a shared automatic variable that any later `-match` call quietly replaces:

```
"abc" -match 'a'
$first = $Matches[0]   # captured immediately — safe
"xyz" -match 'x'
# $Matches now holds the xyz result — $first is still fine only because it
was saved separately first
```

Reading `$Matches` any time after a second `-match` has run, expecting the first result to still be sitting there, is a real, easy mistake — always assign what you need out of `$Matches` immediately after the `-match` that produced it.

`-split` With a Real Regex Pattern

Fundamentals 5 used `-split` with a plain literal separator. `-split`'s right-hand side is always a regex, which becomes genuinely useful once the separator itself is a pattern rather than a fixed string:

```
"one1two22three333four" -split '\d+'
# one, two, three, four — split on any run of one or more digits, whatever their length
```

A FIRST PRACTICAL HABIT

Before reaching for `[regex]::Matches()` or a more complex pattern, try the simplified `-match` / `$Matches` form first on a single sample string — it's faster to write and read for the common "does this match, and what did it capture" case, and it's the same underlying regex engine either way.

Hands-On Exercises

EXERCISE 1

Explain why `"HELLO" -match "hello"` returns `$true`, tying it back to Fundamentals 4's own case-insensitivity of `-eq`. What operator would you use instead for a genuinely case-sensitive match?

View solution

EXERCISE 2

Given a string containing three email addresses, explain why `-match` and `$Matches` only ever surface one of them. What's the correct tool for finding all three, and how would you use it?

View solution

EXERCISE 3

Write a single `-replace` expression that reformats a date string from `"2024-08-05"` (YYYY-MM-DD) to `"08/05/2024"` (MM/DD/YYYY), using regex capture groups and backreferences.

View solution

CHAPTER 2 QUICK REFERENCE

- `-match` — regex comparison; populates `$Matches` on success
- `$Matches[0]` / `$Matches[N]` / `$Matches['name']` — whole match / numbered group / named group
- `-replace` — regex-based replacement; `$1` / `$2` / `${name}` in the replacement string reference capture groups
- **Case-insensitive by default** — `-cmatch` / `-creplace` / `-csplit` for case-sensitive variants
- `[regex]::Matches($text, $pattern)` — returns every match in a string; `-match` alone only ever reports one
- `$Matches` **is shared and gets overwritten** — assign what you need out of it immediately, before the next `-match` runs
- `-split` — its right-hand side is always a regex, not just a literal separator
- **Next chapter:** PowerShell Remoting (WinRM, Invoke-Command, PSSessions)

Everything since Fundamentals 1 has run against the local machine. This chapter is where PowerShell stops being a single-machine tool: `Invoke-Command` runs a script block on another computer entirely and brings the results back — but "brings the results back" hides a real, important asterisk. What comes back across the network isn't the same kind of live object Fundamentals 1 built this whole course around; understanding exactly what's lost in that trip is this chapter's own central lesson.

What Remoting Actually Is: WinRM

PowerShell Remoting runs over **WinRM** (Windows Remote Management), Microsoft's implementation of the WS-Management protocol — a separate transport from SSH, though PowerShell 7+ can optionally use SSH instead once it's configured on both ends. One-time setup, run as Administrator on the machine you want to remote *into*:

```
Enable-PSRemoting -Force
```

This starts the WinRM service, creates a listener for incoming remoting connections, and opens the necessary firewall rule — all three are required before any of the cmdlets below will succeed against that machine.

Invoke-Command: Running Code on a Remote Machine

```
Invoke-Command -ComputerName Server01 -ScriptBlock { Get-Service -Name Spooler }
```

The script block runs entirely on `Server01`, not on the local machine — and only the *results* travel back over the network, not a live connection to keep working with them afterward.

The Deserialization Gotcha: What Comes Back Isn't "Live"

```
$procs = Invoke-Command -ComputerName Server01 -ScriptBlock { Get-Process }

$procs[0].GetType().Name   # Deserialized.System.Diagnostics.Process — NOT the
real type anymore
$procs[0].ProcessName     # works fine — the data survived the trip
$procs[0].Kill()          # fails — Kill() is a METHOD, and methods don't
survive the trip
```

THE CENTRAL FACT THIS CHAPTER IS BUILT ON

A remote object gets serialized into XML to cross the network, then **deserialized** back into a PowerShell object on your local machine — but deserialization can only reconstruct *data* (properties), not *behavior* (methods). The real, live `System.Diagnostics.Process` object, and its ability to actually terminate a running process, never left `Server01` at all — what you're holding locally is a detailed, read-only snapshot, type-prefixed `Deserialized.` as an explicit signal of exactly that. This is a deliberate architectural boundary, not a limitation to work around: letting a remote object's live methods execute back on your machine (or worse, letting your local methods silently execute back on the remote one) would be a genuine security problem, not a convenience worth having.

Passing Local Data In: the `$using:` Scope Modifier

A remote script block runs in a completely separate process on a completely separate machine — it has no visibility into local variables by default, even ones defined on the very line above the call:

```
$serviceName = "Spooler"
# WITHOUT $using: — fails, or silently sees $null, because $serviceName doesn't
exist remotely
Invoke-Command -ComputerName Server01 -ScriptBlock { Get-Service -Name
$serviceName }

# WITH $using: — explicitly injects the LOCAL value into the remote script block
Invoke-Command -ComputerName Server01 -ScriptBlock { Get-Service -Name
$using:serviceName }
```

`$using:` is the explicit signal "reach across and grab this value from my local session" — without it, the remote script block only ever sees its own, entirely empty, local scope.

Persistent Connections: `New-PSSession` & State

```
$session = New-PSSession -ComputerName Server01

Invoke-Command -Session $session -ScriptBlock { $data = Get-Process }
Invoke-Command -Session $session -ScriptBlock { $data.Count }   # $data is STILL
there — the session remembered it
Remove-PSSession $session
```

-COMPUTERNAME IS STATELESS — EVERY CALL IS A BRAND-NEW, THROWAWAY CONNECTION

`Invoke-Command -ComputerName` opens a fresh connection, runs the script block, and tears the whole thing down again — every single time. A variable set in one `-ComputerName` call is completely gone by the next one, even against the identical computer, because there was never a persistent session for it to survive in:

```
Invoke-Command -ComputerName Server01 -ScriptBlock { $data = 1 }
Invoke-Command -ComputerName Server01 -ScriptBlock { $data }   # $null — a
completely new, unrelated connection
```

Reach for `New-PSSession` the moment state needs to survive between calls — it's the difference between one-shot remote commands and an actual ongoing remote working session.

`Enter-PSSession -ComputerName Server01` is the fully interactive version of the same idea — it drops your prompt directly into the remote machine's own PowerShell session, exactly as if you'd logged in locally, until `Exit-PSSession` (or plain `exit`) returns you home.

Multiple Computers at Once — Automatic Parallelism

```
Invoke-Command -ComputerName Server01, Server02, Server03 -ScriptBlock { Get-
Service Spooler }
# Runs against all three simultaneously by default, not one after another —
# -ThrottleLimit controls the maximum number running concurrently (default 32)
```

A FIRST PRACTICAL HABIT

Before relying on a property from a remote result, check `.GetType().Name` the way this chapter's own `$procs[0]` example did — seeing the `Deserialized.` prefix is an immediate, reliable signal that only data, not behavior, made the trip back.

Hands-On Exercises

EXERCISE 1

Explain why `$procs[0].Kill()` fails after `Invoke-Command -ComputerName Server01 -ScriptBlock { Get-Process }`, even though `$procs[0].ProcessName` works perfectly fine. Use this chapter's own deserialization explanation.

View solution

EXERCISE 2

Explain why `$serviceName` isn't visible inside a remote script block without `$using:serviceName`, even though it's clearly defined in the local session on the line right before the `Invoke-Command` call.

View solution

EXERCISE 3

Explain why a variable set inside one `Invoke-Command -ComputerName` call doesn't persist into a second `Invoke-Command -ComputerName` call against the same computer. What would you change to make it persist?

View solution

CHAPTER 3 QUICK REFERENCE

- **WinRM / WS-Management** — the protocol underneath PowerShell Remoting; `Enable-PSRemoting -Force` is the one-time setup
- `Invoke-Command -ComputerName / -ScriptBlock` — runs code remotely, returns only results, no persistent connection
- **Deserialized objects** — remote results keep their data (properties) but lose their behavior (methods); type-prefixed `Deserialized.`
- `$using:variableName` — explicitly injects a local value into a remote script block's otherwise-empty scope
- `New-PSSession` / `-Session` — a persistent connection; variables and state survive between multiple `Invoke-Command` calls
- `-ComputerName` **alone is stateless** — every call is a brand-new connection; nothing persists between calls
- `Enter-PSSession` — a fully interactive remote prompt; `Exit-PSSession` to return
- **Multiple computers run in parallel automatically** — `-ThrottleLimit` controls the maximum concurrent (default 32)
- **Next chapter:** Background & Parallel Execution (Start-Job, ForEach-Object -Parallel, a Runspaces intro)

Chapter 3's remoting already ran code somewhere other than your own immediate session — this chapter does the same thing on the local machine, two genuinely different ways. `Start-Job` is heavyweight but simple: a full separate process per job. `ForEach-Object -Parallel` is lighter and PowerShell 7+ only — but it introduces a real, easy-to-miss gotcha of its own the moment you try to collect results back into an ordinary shared variable.

`Start-Job`: True Background Execution, One Process Per Job

```
$job = Start-Job -ScriptBlock { Start-Sleep -Seconds 5; "Done" }
$job # returns immediately — State: Running — your prompt is never blocked
```

`Start-Job` launches an entirely separate `powershell.exe` process to run the script block — genuine isolation, but genuine overhead too: every job started this way costs a full new process, not a lightweight thread.

The Job Lifecycle: `Get-Job`, `Receive-Job` & the `-Keep` Gotcha

```
Get-Job # check State: Running / Completed / Failed
Wait-Job $job | Receive-Job # block until finished, then get the output
Remove-Job $job
```

RECEIVE-JOB DRAINS THE JOB'S OWN OUTPUT BUFFER BY DEFAULT

Calling `Receive-Job` a second time on the same job returns **nothing**, even though the job clearly produced output the first time:

```
Receive-Job $job # "Done" — and the output buffer is now empty
Receive-Job $job # nothing — already consumed once
Receive-Job $job -Keep   # leaves the output in the buffer, safe to read
again later
```

Add `-Keep` any time you might genuinely need to read a job's output more than once.

Passing Data In: `-ArgumentList` (and `$using:`)

A job's script block runs in its own separate process, exactly like Chapter 3's remote script blocks did — it has no automatic access to local variables either:

```
# The traditional, always-works approach — param() plus -ArgumentList
$job = Start-Job -ScriptBlock { param($n) $n * $n } -ArgumentList 5

# The same $using: scope modifier from Chapter 3 works here too, on modern
PowerShell
$n = 5
$job = Start-Job -ScriptBlock { $using:n * $using:n }
```

The Cost of a Job: Why Many Small Jobs Get Slow

Because each `Start-Job` is a genuinely separate `powershell.exe` process, starting 100 small jobs means starting 100 separate PowerShell processes — real, measurable startup overhead multiplied by however many jobs you launch. Fine for a handful of independent, possibly long-running tasks; a poor fit for "run this tiny operation 500 times, fast."

`ForEach-Object -Parallel`: Lighter-Weight Concurrency (PowerShell 7+)

```
1..10 | ForEach-Object -Parallel { $_ * $_ } -ThrottleLimit 5
```

Instead of a full process per item, each parallel iteration runs in its own lightweight **runspace** — much cheaper to create than a whole new process, which is why `-Parallel` scales to hundreds of small items far better than an equivalent pile of `Start-Job` calls. `-ThrottleLimit` caps how many run concurrently — a much lower default (5) than remoting's own default of 32, since these compete for local CPU rather than remote machines' resources.

Exactly like `Start-Job` and Chapter 3's remoting, a local variable isn't automatically visible inside the parallel block — `$using:` is required here too:

```
$multiplier = 3
1..5 | ForEach-Object -Parallel { $_ * $using:multiplier }
```

The Isolated-Runspace Gotcha: Why a Shared Variable Doesn't Update

```powershell
$results = @()
1..5 | ForEach-Object -Parallel { $results += $_ }

$results.Count   # 0 — completely empty, even though it clearly ran five times
```

> **THE CENTRAL FACT THIS CHAPTER IS BUILT ON**
>
> Every `-Parallel` iteration runs in its own genuinely isolated runspace — the same kind of scope separation Chapter 3's remoting had, just local instead of across a network. `$results` inside each iteration isn't *the* `$results` from the caller's own scope; without `$using:`, it's simply undefined inside that isolated runspace, so `$results += $_` creates a brand-new, purely local variable in each of the five separate runspaces — five short-lived copies, none of which is ever the original, and all five are discarded the moment their own iteration ends.

The fix requires a genuinely thread-safe collection, not a plain array — one built to handle multiple runspaces writing to it safely at the same time:

```powershell
$results = [System.Collections.Concurrent.ConcurrentBag[object]]::new()
1..5 | ForEach-Object -Parallel { ($using:results).Add($_ * $_) }

$results.Count   # 5 — correct, because a ConcurrentBag is genuinely safe to
share and write across runspaces
```

A Brief Word on Runspaces Directly

`ForEach-Object -Parallel` is itself built on PowerShell's own **Runspace API** — a runspace pool it manages for you behind the scenes. Managing raw runspaces directly (`[runspacefactory]::CreateRunspacePool()` and hand-rolling your own thread-safe result collection and synchronization) is real, substantially lower-level territory, genuinely useful for high-performance custom tooling but well beyond what this course builds from scratch — an honest boundary, not a gap to quietly paper over.

	ISOLATION UNIT	TYPICAL USE
Start-Job	A full separate process — heaviest overhead, strongest isolation	A handful of independent, possibly long-running tasks
ForEach-Object -Parallel	A lightweight runspace — much cheaper than a process (PowerShell 7+ only)	Many short, similar parallel iterations
Raw Runspaces	Manually managed — most control, most complexity	High-performance custom tooling; rarely needed directly

A FIRST PRACTICAL HABIT

Before reaching for either kind of parallel execution, ask whether the work is genuinely independent — if one iteration's result depends on another's, parallelizing it correctly gets a lot harder than either `-ArgumentList` or `$using:` alone can solve.

Hands-On Exercises

EXERCISE 1

Explain why calling `Receive-Job $job` twice in a row returns real output the first time and nothing the second time. What parameter fixes it?

View solution

EXERCISE 2

Explain why `$results.Count` is `0` after `1..5 | ForEach-Object -Parallel { $results += $_ }`, using this chapter's own isolated-runspace explanation. What's the actual fix?

View solution

EXERCISE 3

You need to process 200 small, independent items in parallel as fast as possible on PowerShell 7. Would you reach for `Start-Job` or `ForEach-Object -Parallel`? Justify your answer using this chapter's own weight/overhead comparison.

View solution

CHAPTER 4 QUICK REFERENCE

- `Start-Job` — a full separate process per job; heaviest overhead, strongest isolation
- `Receive-Job` — drains the job's output buffer by default; use `-Keep` to read it more than once
- `-ArgumentList` / `$using:` — how to pass local data into a job or parallel block's otherwise-empty scope
- `ForEach-Object -Parallel` **(PowerShell 7+)** — runs each iteration in a lightweight runspace, not a full process; `-ThrottleLimit` (default 5) caps concurrency
- **Isolated runspaces** — a plain shared variable modified inside `-Parallel` never updates the caller's copy; each iteration gets its own, discarded afterward
- `[System.Collections.Concurrent.ConcurrentBag[object]]::new()` — a genuinely thread-safe collection, the real fix for collecting results out of `-Parallel`
- **Raw Runspaces** — what `-Parallel` is built on; real, advanced territory, an honest boundary for this course
- **Next chapter:** Working with .NET Objects & COM Directly

Every object this entire course has ever touched — a process, a file, an error record — has secretly been a .NET object the whole time, per Fundamentals 5's own opening finding. Every one of them arrived pre-built, courtesy of a cmdlet. This chapter is about building and calling into .NET directly, without a cmdlet doing it for you first — plus its considerably older cousin, COM, which comes with a real, painful cleanup responsibility no other chapter in this course has needed to mention.

Creating Objects: `New-Object` vs. `[Type]::new()`

```
# Older syntax — works everywhere, more verbose
$sb1 = New-Object System.Text.StringBuilder

# Newer syntax (PowerShell 5+) — generally preferred, skips New-Object's own parameter-binding overhead
$sb2 = [System.Text.StringBuilder]::new()
```

Both lines create the identical type of object — `[Type]::new()` is simply the more direct, modern way to say the same thing.

Static vs. Instance Members

```
# Static — called on the TYPE itself, no object ever created
[Math]::Pow(2, 10)   # 1024
[Math]::PI          # 3.14159265358979
# Instance — called on a particular OBJECT you actually created
$sb2.Append("Hello")
$sb2.ToString()
```

`[Math]` exposes *only* static members — there's no such thing as "an instance of Math," which is why you'll never see `New-Object Math` or `[Math]::new()` anywhere. The `[Type]::Member` syntax always means "call this directly on the type"; a variable's own `.Member` always means "call this on the specific object that variable holds."

A Real Payoff: StringBuilder vs. String Concatenation at Scale

```
# Inefficient at scale — strings are immutable in .NET
$result = ""
foreach ($i in 1..10000) { $result += "line $i`n" }

# Efficient — StringBuilder mutates in place instead of copying
$sb = [System.Text.StringBuilder]::new()
foreach ($i in 1..10000) { [void]$sb.AppendLine("line $i") }
$result = $sb.ToString()
```

Every `+=` against a string doesn't modify it in place — it can't, because .NET strings are immutable. Each one silently builds an entirely *new* string containing a full copy of everything accumulated so far, plus the new piece. At 10,000 iterations that's not 10,000 small operations; it's roughly 10,000 progressively larger copies, real quadratic-ish cost that gets noticeably slow well before the loop ends. `StringBuilder` avoids all of that by mutating an internal buffer directly. The `[void]` cast on `AppendLine` isn't decoration — `Append`/`AppendLine` both return the `StringBuilder` itself (for chaining), and per Fundamentals 7's own central rule, an uncaptured return value would otherwise silently join the loop's own output.

Other Useful .NET Types

```
[System.IO.Path]::Combine("C:\Users\Philip", "Documents", "report.txt")
# C:\Users\Philip\Documents\report.txt - correct separators, no manual string-
joining needed

[System.IO.Path]::GetExtension("report.txt")                # .txt
[System.IO.Path]::GetFileNameWithoutExtension("report.txt")  # report
```

`[System.IO.Path]` is generally more robust than hand-built string concatenation for paths — it handles separator characters correctly regardless of trailing slashes on the inputs. `[regex]`, already covered in Chapter 2, is the same idea applied to pattern matching — both are ordinary .NET types, reached exactly the way `[Math]` and `[System.Text.StringBuilder]` are here.

COM Objects: `New-Object -ComObject`

COM (Component Object Model) predates .NET entirely, and it isn't part of the managed object model this chapter has covered so far — `New-Object -ComObject` is the *only* way in; there's no `[Type]::new()` equivalent, because a COM class isn't a .NET type in the first place.

```
$excel = New-Object -ComObject Excel.Application
$excel.Visible = $true
$workbook = $excel.Workbooks.Add()
$sheet = $workbook.Worksheets.Item(1)
$sheet.Cells.Item(1, 1) = "Hello from PowerShell"
$workbook.SaveAs("C:\temp\report.xlsx")
$excel.Quit()
```

The COM Cleanup Gotcha: Ghost Processes

SKIPPING .QUIT() CAN LEAVE EXCEL.EXE RUNNING INVISIBLY, INDEFINITELY

A managed .NET object's memory is cleaned up automatically by the garbage collector once nothing references it anymore — every object this course has used so far worked this way with no extra effort. A COM object doesn't get that same reliable, prompt cleanup: the underlying native application (`EXCEL.EXE`, here) can keep running as an orphaned background process even after `$excel` goes out of scope and the script finishes. Run a script like this repeatedly without `.Quit()`, and Task Manager quietly accumulates one ghost `EXCEL.EXE` process per run — invisible (since `Visible` only controlled the window, not the underlying process's lifetime the moment it's abandoned), silently consuming memory, never asked for.

```
$excel.Quit()
[System.Runtime.InteropServices.Marshal]::ReleaseComObject($excel) | Out-Null
Remove-Variable excel
```

THE CENTRAL FACT THIS CHAPTER IS BUILT ON

Every .NET object covered earlier in this chapter cleans itself up automatically the moment nothing references it — that's what "managed" means. COM sits entirely outside that system: it's your own explicit responsibility to call `.Quit()` (or the more general `Marshal.ReleaseComObject()`) every time, on every COM object you create, with no automatic safety net behind you. Skipping it doesn't error, doesn't warn, and doesn't fail the script — it just quietly leaves something running that should have stopped.

A FIRST PRACTICAL HABIT

Wrap any COM automation in `try` / `finally` (Fundamentals 9) with the cleanup calls inside `finally` — that guarantees `.Quit()` runs even if something in the middle of the script throws, which is exactly the situation most likely to leave a ghost process behind in practice.

Hands-On Exercises

EXERCISE 1

Explain the real performance difference between building a 10,000-line string with `+=` in a loop versus using `StringBuilder`, using this chapter's own explanation of string immutability.

View solution

EXERCISE 2

Explain why `[Math]::Pow(2, 10)` works with no object creation step first, while `$sb.Append("x")` requires `$sb` to already exist as a real, created object. What's the underlying distinction?

View solution

EXERCISE 3

Explain why forgetting to call `$excel.Quit()` after `New-Object -ComObject Excel.Application` can leave `EXCEL.EXE` running in the background indefinitely, even after the script finishes and `$excel` goes out of scope.

View solution

CHAPTER 5 QUICK REFERENCE

- `New-Object TypeName` / `[Type]::new()` — two ways to create a .NET object; `[Type]::new()` is the newer, generally preferred syntax
- **Static members (`[Type]::Member`)** — called on the type itself, no object required; e.g. `[Math]::Pow()`, `[Math]::PI`
- **Instance members (`$object.Member`)** — called on a specific, already-created object
- `StringBuilder` — mutates a buffer in place; far cheaper than repeated `+=` on an immutable string at scale
- `[System.IO.Path]::Combine()` — robust path-joining, correct separators regardless of trailing slashes
- `New-Object -ComObject` — the only way to create a COM object; no `[Type]::new()` equivalent exists for COM
- **COM cleanup is manual** — `.Quit()` and/or `Marshal.ReleaseComObject()`; skipping it can leave a ghost process (e.g. `EXCEL.EXE`) running indefinitely, with no automatic garbage collection to rescue you
- **Next chapter:** CIM/WMI: Querying and Managing Windows Systems

Windows exposes an enormous amount of its own state — hardware, OS details, disks, running processes, installed software — as a queryable, database-like management layer: **WMI** (Windows Management Instrumentation). PowerShell's modern way in is `Get-CimInstance`, and one detail from Chapter 3 turns out to matter here too: CIM queries travel over the exact same WinRM infrastructure Chapter 3's own remoting used, which is the real reason CIM replaced WMI's older cmdlet family rather than just renaming it.

Get-CimInstance: Querying System State

```
Get-CimInstance -ClassName Win32_OperatingSystem |
    Select-Object Caption, Version, OSArchitecture

Get-CimInstance -ClassName Win32_LogicalDisk |
    Select-Object DeviceID, @{Name="FreeGB"; Expression={
[math]::Round($_.FreeSpace / 1GB, 2) }}
```

`Win32_OperatingSystem`, `Win32_LogicalDisk`, `Win32_ComputerSystem`, `Win32_BIOS`, and `Win32_Process` are among the most commonly used WMI classes — each one a real, queryable table of system state, returned as ordinary PowerShell objects Chapter 3's `Where-Object` / `Select-Object` already know how to work with.

WQL: `-Filter` and `-Query`

```
# -Filter - a simplified, single-condition shorthand
Get-CimInstance -ClassName Win32_Process -Filter "Name='chrome.exe'"
# -Query - full WQL (WMI Query Language), SQL-like syntax, for anything more
complex
Get-CimInstance -Query "SELECT Name, ProcessId FROM Win32_Process WHERE
Name='chrome.exe'"
```

Invoking Methods: `Invoke-CimMethod`

WMI classes aren't just read-only data — many expose real methods too, invoked directly rather than through a dedicated cmdlet:

```
# Calling a class-level method, with arguments
Invoke-CimMethod -ClassName Win32_Process -MethodName Create -Arguments @{
CommandLine = "notepad.exe" }

# Calling a method on a specific instance you already queried
$os = Get-CimInstance -ClassName Win32_OperatingSystem
Invoke-CimMethod -InputObject $os -MethodName Reboot
```

Remote Queries: The Same WinRM Infrastructure as Chapter 3

```
Get-CimInstance -ClassName Win32_OperatingSystem -ComputerName Server01

# A reusable CIM session — the same pattern as Chapter 3's own New-PSSession
$cimSession = New-CimSession -ComputerName Server01
Get-CimInstance -ClassName Win32_LogicalDisk -CimSession $cimSession
```

THE CENTRAL FACT THIS CHAPTER IS BUILT ON

`Get-CimInstance` isn't just `Get-WmiObject` under a new name — it's a genuine protocol upgrade. CIM travels over WSMan, the exact same WinRM transport Chapter 3's remoting already uses, which is why remote CIM queries need the same one-time `Enable-PSRemoting` setup and the same firewall rule, rather than a separate, older set of exceptions. `Get-WmiObject`, by contrast, used **DCOM** — an older, largely firewall-unfriendly protocol that needs its own separate configuration to punch through most networks. `New-CimSession` even mirrors Chapter 3's own `New-PSSession` directly: a reusable, persistent connection instead of reconnecting from scratch on every single call.

`Get-WmiObject` vs. `Get-CimInstance`

	GET-WMIOBJECT (LEGACY)	GET-CIMINSTANCE (MODERN)
Protocol	DCOM — older, often firewall-unfriendly	WSMan/WinRM — the same as Chapter 3's remoting
Availability	Windows PowerShell 5.1 only	Available in PowerShell 7+ as well
Session reuse	No real persistent-session concept	`New-CimSession` — reusable, like `New-PSSession`

`Get-WmiObject` was fully removed from PowerShell 7+ — not deprecated-but-working, genuinely gone — which makes `Get-CimInstance` the only real option for any script expected to run on a modern PowerShell install.

WIN32_PRODUCT IS SLOW FOR A REAL, DOCUMENTED REASON — AVOID IT

Querying `Win32_Product` to list installed software is a well-known, Microsoft-acknowledged trap: the query itself triggers Windows Installer to run a consistency check against **every single installed MSI package** on the machine — genuinely slow, and capable of triggering unexpected repair prompts on machines with a lot of software installed. Reading installed programs from the registry's own `Uninstall` keys instead — the same `HKLM:` / `HKCU:` provider Fundamentals 2 already covered — is faster and has no such side effect: `Get-ItemProperty HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\*`.

A FIRST PRACTICAL HABIT

Reach for `-Filter` first for anything expressible as one simple condition — it reads closer to plain PowerShell. Save full `-Query` WQL for genuinely more complex conditions `-Filter` can't express on its own, mirroring the same "simplified first, script-block when needed" instinct Chapter 1's own `Where-Object` material already built.

Hands-On Exercises

EXERCISE 1

Explain why `Get-WmiObject` doesn't work at all in PowerShell 7+ while `Get-CimInstance` works fine, referencing the underlying protocol each one actually uses.

View solution

EXERCISE 2

Explain why running `Get-CimInstance -ClassName Win32_Product` to list installed software is a real, documented bad idea, and what alternative this chapter recommends instead.

View solution

EXERCISE 3

Write a `Get-CimInstance` query (using either `-Filter` or `-Query`) that finds all `Win32_Process` instances named `notepad.exe`. Explain which form you chose and why.

View solution

CHAPTER 6 QUICK REFERENCE

- `Get-CimInstance -ClassName` — queries a WMI class as ordinary PowerShell objects
- `-Filter` / `-Query` — a simplified single-condition shorthand, or full WQL for anything more complex
- `Invoke-CimMethod` — calls a real method on a WMI class or a queried instance
- **CIM uses WSMan/WinRM** — the same transport, setup, and firewall rule as Chapter 3's own remoting
- `New-CimSession` — a reusable remote connection, mirroring Chapter 3's own `New-PSSession`
- `Get-WmiObject` **is fully removed from PowerShell 7+** — used DCOM, an older, firewall-unfriendly protocol; `Get-CimInstance` is the only real modern option
- `Win32_Product` **is a real trap** — triggers a consistency check against every installed MSI; read the registry's `Uninstall` keys instead
- **Next chapter:** Calling APIs: Invoke-RestMethod & Working with JSON

Fundamentals 8 covered `ConvertTo-Json`/`ConvertFrom-Json` as a genuine structured interchange format — this chapter is where that format actually starts crossing a network, to and from a real REST API. It also surfaces a real, useful exception to one of Fundamentals 9's own central rules: unlike most cmdlets, a failed API call doesn't quietly print a red error and move on — it stops, immediately, whether you asked it to or not.

`Invoke-RestMethod` vs. `Invoke-WebRequest`

```
# Invoke-RestMethod — parses JSON automatically, hands back real objects
directly
$user = Invoke-RestMethod -Uri "https://api.github.com/users/octocat"
$user.name           # already a usable property — no ConvertFrom-Json step
needed
# Invoke-WebRequest — the raw HTTP response: status code, headers, AND the body
as text
$response = Invoke-WebRequest -Uri "https://api.github.com/users/octocat"
$response.StatusCode   # 200
$response.Headers      # the full response header collection
$response.Content | ConvertFrom-Json # the body is still just raw text — parse
it yourself
```

`Invoke-RestMethod` is really `Invoke-WebRequest` plus an automatic `ConvertFrom-Json` already applied — the right default whenever you just want the data. Reach for `Invoke-WebRequest` instead the moment you actually need the status code, the response headers, or a non-JSON body (HTML, plain text, a file download).

Sending JSON: POST Requests

```
$body = @{ name = "New Item"; price = 19.99 } | ConvertTo-Json
Invoke-RestMethod -Uri "https://api.example.com/items" `
    -Method Post -Body $body -ContentType "application/json"
```

Building the request body is exactly Fundamentals 8's own `ConvertTo-Json` — including the same default `-Depth 2` gotcha for any genuinely nested request body.

Authentication Headers

```
$headers = @{ Authorization = "Bearer $token" }
Invoke-RestMethod -Uri "https://api.example.com/secure" -Headers $headers
```

HTTP Errors Are Terminating by Default

```
try {
    Invoke-RestMethod -Uri "https://api.example.com/does-not-exist"
} catch {
    "Request failed: $($_.Exception.Message)"
}
# Caught with NO -ErrorAction Stop added anywhere — this just works
```

A REAL, USEFUL EXCEPTION TO FUNDAMENTALS 9'S OWN RULE

Fundamentals 9 spent an entire chapter establishing that most cmdlet errors are non-terminating by default, and that `-ErrorAction Stop` is almost always required before `try`/`catch` actually does anything. `Invoke-RestMethod` and `Invoke-WebRequest` don't follow that pattern — any non-2xx HTTP status code (a 404, a 500, an authentication failure) is treated as a genuinely terminating error automatically, no `-ErrorAction Stop` required. It's worth remembering specifically *because* it's the exception rather than the norm — assuming every cmdlet needs `-ErrorAction Stop` to be caught, out of habit, isn't wrong here, but it isn't necessary either.

Reading the Real API Error Message

```
try {
    Invoke-RestMethod -Uri "https://api.example.com/items" -Method Post -Body
$body -ContentType "application/json"
} catch {
    "HTTP failure: $($_.Exception.Message)" # a generic .NET message — "The
remote server returned an error: (400) Bad Request."
"API said: $($_.ErrorDetails.Message)" # the ACTUAL response body — often the
genuinely useful part
}
```

`$_.Exception.Message` only ever describes the HTTP failure in generic terms. Most real APIs put the actually useful explanation — *which* field was invalid, *why* the request was rejected — in the response body itself, and `$_.ErrorDetails.Message` is where PowerShell already parses that body out for you, no manual stream-reading required.

Pagination: Looping Through Multiple Pages

```
$allResults = @()
$uri = "https://api.example.com/items?page=1"
do {
    $response = Invoke-RestMethod -Uri $uri
$allResults += $response.items
    $uri = $response.nextPageUrl
} while ($uri)
```

Fundamentals 6's own `do`/`while` is a natural fit here — the loop needs to run at least once regardless, then keep going only as long as the API keeps handing back another page to fetch.

> **A FIRST PRACTICAL HABIT**
>
> Watch for a `429 Too Many Requests` response on any API called in a tight loop, and add a small `Start-Sleep` between calls proactively rather than reactively — most APIs document their own rate limits, and it's worth checking before writing the loop, not after it starts failing.

Hands-On Exercises

EXERCISE 1

Explain the difference between `Invoke-RestMethod` and `Invoke-WebRequest`. Describe a concrete situation where you'd need `Invoke-WebRequest` specifically, rather than the simpler `Invoke-RestMethod`.

View solution

EXERCISE 2

Explain why a plain `try`/`catch` around `Invoke-RestMethod` can catch an HTTP 404 with no `-ErrorAction Stop` anywhere, contrasting this directly with Fundamentals 9's own general rule about non-terminating errors.

View solution

EXERCISE 3

Write a POST request using `Invoke-RestMethod` that sends a JSON body built from a hashtable, includes a Bearer token in the `Authorization` header, and is wrapped in `try`/`catch` that reads the real API error message from the response body if the request fails.

View solution

CHAPTER 7 QUICK REFERENCE

- `Invoke-RestMethod` — parses JSON automatically, returns real objects; the right default for API data
- `Invoke-WebRequest` — the raw response: status code, headers, and body as text; needed for anything beyond parsed JSON data
- `-Body` / `-ContentType "application/json"` / `-Method Post` — sending a JSON request body, built with Fundamentals 8's own `ConvertTo-Json`
- `-Headers @{ Authorization = "Bearer $token" }` — the common bearer-token auth pattern
- **HTTP errors are terminating by default** — a real exception to Fundamentals 9's own non-terminating-by-default rule; no `-ErrorAction Stop` needed here
- `$_.ErrorDetails.Message` — the actual API response body text, already parsed out inside `catch`
- **Pagination** — a `do`/`while` loop (Fundamentals 6) following a "next page" link until none remains
- **Next chapter:** Building & Publishing Your Own Modules

Fundamentals 11 covered *consuming* modules — `Get-Module`, `Import-Module`, installing one from the Gallery. This chapter is about being on the other side of that: turning a folder of your own functions into something someone else could genuinely `Install-Module`, complete with the one file most people skip when they're just experimenting, and the reason skipping it eventually causes real problems.

Two Files: `.psm1` (Code) and `.psd1` (Manifest)

A **script module** is just a `.ps1` file renamed to `.psm1`, containing function definitions — nothing more is technically required to import and use one. A real, publishable module adds a **manifest**, a `.psd1` file: pure metadata (version, author, which functions are actually meant to be public, required PowerShell version, dependencies) that PowerShell can read without ever running any of the module's own code.

`Export-ModuleMember`: Controlling What's Actually Public

```
# MyTools.psm1
function Get-Greeting {
    param([string]$Name)
    "Hello, $Name!"
}

function Get-InternalHelper {
    # an internal helper — not meant to be called by whoever imports this module
"internal only"
}

Export-ModuleMember -Function Get-Greeting
```

WITHOUT EXPORT-MODULEMEMBER, EVERYTHING GETS EXPORTED BY DEFAULT

Skip `Export-ModuleMember` entirely, and **every** function defined in the `.psm1` — including `Get-InternalHelper`, which was never meant to be part of any public API — becomes fully visible and callable by anyone who imports the module, discoverable through the exact same `Get-Command` Fundamentals 3 already covered. Explicitly listing only the functions actually meant to be public is what keeps a module's real interface honest.

Module Folder Structure & `PSModulePath`

```
$env:PSModulePath -split ';' # the semicolon-separated list of folders
PowerShell searches for modules
# The folder name must match the module's own base name exactly:
# MyTools\
# ├── MyTools.psm1
# └── MyTools.psd1
```

A module named `MyTools` that doesn't sit inside a `MyTools\` folder, on one of the paths `$env:PSModulePath` lists, simply won't be found by `Import-Module` or the auto-loading Fundamentals 11 already relied on.

`New-ModuleManifest`: Generating a Real Manifest

```
New-ModuleManifest -Path .\MyTools\MyTools.psd1 `
    -RootModule "MyTools.psm1" `
    -ModuleVersion "1.0.0" `
    -Author "Philip Osztromok" `
    -FunctionsToExport @("Get-Greeting")
```

`New-ModuleManifest` generates a correctly structured `.psd1` with all the right fields — far more reliable than hand-writing one from scratch and risking a typo in a key PowerShell's own tooling actually checks.

Publishing to the PowerShell Gallery

```
Publish-Module -Path .\MyTools -NuGetApiKey $apiKey
```

Publishing requires an account and a NuGet API key from `powershellgallery.com` — the same PSGallery Fundamentals 11 already covered from the consuming side, via `Find-Module` / `Install-Module`.

EACH PUBLISHED VERSION IS IMMUTABLE — REPUBLISHING THE SAME VERSION FAILS

The Gallery treats every published version as permanent and unchangeable — running `Publish-Module` again with the identical `ModuleVersion` still set in the manifest fails outright, even for a genuinely tiny fix. The version has to be bumped first:

```
Update-ModuleManifest -Path .\MyTools\MyTools.psd1 -ModuleVersion "1.0.1"
Publish-Module -Path .\MyTools -NuGetApiKey $apiKey
```

Semantic versioning (`Major.Minor.Patch`) is the expected convention — a patch bump like `1.0.0` → `1.0.1` for a small fix, a minor bump for a genuinely new feature, a major bump for a breaking change.

`FunctionsToExport` vs. `Export-ModuleMember`: Why Explicit Beats Wildcard

Both the manifest's own `FunctionsToExport` and the `.psm1`'s `Export-ModuleMember` can restrict what's public — but leaving `FunctionsToExport` as the default wildcard (`'*'`) carries a real, system-wide performance cost, not just a style preference:

THE CENTRAL FACT THIS CHAPTER IS BUILT ON

A manifest with an explicit `FunctionsToExport` list is pure, static metadata — PowerShell can read exactly what a module exports directly from the `.psd1` file, without ever loading or running a single line of the module's own code. `FunctionsToExport = '*'` throws that advantage away: PowerShell has no choice but to actually load the module and introspect it to discover what it exports, every time module auto-discovery needs an answer. Multiply that across every module on a system using a wildcard, and Fundamentals 11's own "most modules just auto-load, no explicit `Import-Module` needed" convenience gets measurably slower for everyone — auto-loading only stays fast because well-behaved modules declare their exports explicitly, rather than making the shell do the work of finding out.

A FIRST PRACTICAL HABIT

Let `New-ModuleManifest -FunctionsToExport` match exactly the same list already passed to `Export-ModuleMember` in the `.psm1` — keeping both declarations in sync avoids a confusing mismatch between what the manifest promises and what the code actually exposes.

Hands-On Exercises

EXERCISE 1

Explain what happens to `Get-InternalHelper` if `Export-ModuleMember -Function Get-Greeting` is never added to `MyTools.psm1` at all, referencing this chapter's own default-export behavior.

View solution

EXERCISE 2

Explain why running `Publish-Module` a second time with the manifest's `ModuleVersion` left unchanged fails, and what you'd need to do first before it can succeed.

View solution

EXERCISE 3

Explain why `FunctionsToExport @('Get-Greeting')` in a manifest is genuinely better for system-wide performance than leaving it as `FunctionsToExport = '*'`, referencing this chapter's own explanation of module auto-discovery.

View solution

CHAPTER 8 QUICK REFERENCE

- `.psm1` — the module's actual code; `.psd1` — its manifest, pure metadata read without running any code
- `Export-ModuleMember -Function` — restricts what's public; without it, every function in the `.psm1` is exported by default
- **Folder name must match the module's base name**, and sit on a path listed in `$env:PSModulePath`
- `New-ModuleManifest` — generates a correctly structured `.psd1` rather than hand-writing one
- `Publish-Module -NuGetApiKey` — publishes to the PowerShell Gallery; requires an account and API key
- **Published versions are immutable** — republishing the same `ModuleVersion` fails; bump it with `Update-ModuleManifest` first
- **Explicit** `FunctionsToExport` **beats a wildcard** — lets PowerShell read exports as static metadata instead of loading the whole module just to find out, keeping system-wide auto-discovery fast
- **Next chapter:** Introduction to Desired State Configuration (DSC)

Every script written since Fundamentals 7 has been **imperative**: a sequence of steps, executed once, that you're responsible for making safe to re-run yourself — exactly the discipline the Fundamentals capstone's own `try`/`catch`-wrapped `Remove-Item` was built around. DSC is a genuine departure from that whole model: instead of writing steps, you declare the **end state** you want, and let DSC's own engine figure out — and keep checking — whether reality matches it.

The `Configuration` Keyword & Resources

```
Configuration MyWebServerConfig {
    Node "localhost" {
        WindowsFeature IIS {
            Ensure = "Present"
            Name   = "Web-Server"
        }
        Service W3SVC {
            Name      = "W3SVC"
            State     = "Running"
            DependsOn = "[WindowsFeature]IIS"
        }
    }
}
```

`Configuration` is its own distinct keyword, not a function — inside it, `Node` blocks target specific machines, and inside *those*, each named block (`WindowsFeature`, `Service`, `File`, `Registry`, and more) is a **resource** declaring a piece of desired state, almost always including an `Ensure = "Present"`/`"Absent"` property. `DependsOn` makes ordering explicit where it genuinely matters — here, IIS has to actually be installed before its own `W3SVC` service can be started.

Compiling a Configuration: `.mof` Files

Writing the block above doesn't apply anything by itself — `Configuration` defines a template that has to be **compiled** first, by calling it like a function:

```
MyWebServerConfig -OutputPath .\MyWebServerConfig
# Produces .\MyWebServerConfig\localhost.mof — one .mof file per targeted Node,
nothing applied yet
```

A `.mof` (Managed Object Format) file is the actual, portable artifact DSC works from — a real, easy-to-miss two-step process: define the configuration, *then* compile it, and only after that, apply it.

Applying It: `Start-DscConfiguration`

```
Start-DscConfiguration -Path .\MyWebServerConfig -Wait -Verbose
```

Only this step actually does anything to the target machine — checking each resource's current state and taking action only where it doesn't already match.

The Real Payoff: Idempotency, Built In

THE CENTRAL FACT THIS CHAPTER IS BUILT ON

Run `Start-DscConfiguration` a second time against a node that's already fully compliant, and nothing happens — no error, no duplicate install, no second attempt to start an already-running service. This isn't something you had to code yourself the way the Fundamentals capstone's own `Remove-Item` needed an explicit `try` / `catch` and careful "does this still exist" thinking — every DSC resource builds a "check the current state, only act if it's actually different" test into itself as a fundamental design principle. Declarative configuration means idempotency is the resource's own job, not yours.

Detecting and Correcting Drift

```
Test-DscConfiguration -Detailed
# Reports whether the LIVE system still matches the compiled configuration — no
changes applied, just a check
```

`Test-DscConfiguration` answers "has anything drifted?" without touching anything — a service someone stopped manually, a file someone edited by hand. A node's Local Configuration Manager (LCM) can even be set to periodically re-check and auto-correct drift on its own, turning a one-time configuration into an ongoing, self-healing guarantee — the real distinguishing value DSC has over a script that only ever runs once.

The Escape Hatch: The `Script` Resource

```
Script CustomCheck {
    GetScript  = { @{ Result = "n/a" } }
    TestScript = { Test-Path "C:\Reports\marker.txt" }
    SetScript  = { New-Item -Path "C:\Reports\marker.txt" -ItemType File }
}
```

For anything not covered by a built-in resource, `Script` lets you hand-write the three pieces DSC's own idempotency model needs: `TestScript` (is this already true?), `SetScript` (make it true, only called if `TestScript` says no), and `GetScript` (report the current state) — the same idempotent-by-design contract every built-in resource already follows, just written by hand.

HONEST SCOPE NOTE

This is genuinely an *introduction*. Real production DSC usage — class-based custom resources, pull servers, Azure Automation DSC, and the newer cross-platform DSC v3 rewrite — is a substantially larger topic than a single chapter can responsibly cover, matching the same honest boundary this course already drew around raw Runspaces in Chapter 4.

A FIRST PRACTICAL HABIT

Before writing a custom `Script` resource, check whether a built-in resource already covers what you need — `WindowsFeature`, `Service`, `File`, and `Registry` alone cover a large share of real configuration tasks, already idempotent, with no hand-written `TestScript` logic to get right yourself.

Hands-On Exercises

EXERCISE 1

Explain the core difference between a DSC `Configuration` block and an ordinary PowerShell function, using this chapter's own declarative-vs-imperative framing.

View solution

EXERCISE 2

Explain what actually happens when you call `MyWebServerConfig -OutputPath .\MyWebServerConfig`. What gets produced, and what hasn't happened yet at that point?

View solution

EXERCISE 3

Explain why running `Start-DscConfiguration` a second time against an already-compliant node does nothing, contrasting this directly with how the Fundamentals capstone had to manually check state before calling `Remove-Item`.

View solution

CHAPTER 9 QUICK REFERENCE

- **Declarative, not imperative** — describe the desired end state; DSC's engine figures out how to get there
- `Configuration` **/** `Node` **/ resources** — a distinct keyword; `Node` targets a machine; each resource block (`WindowsFeature` , `Service` , `File` , `Registry`) declares one piece of desired state
- **Compiling** — calling the configuration like a function produces a `.mof` file per node; nothing is applied yet at this step
- `Start-DscConfiguration` — the step that actually checks and applies state
- **Idempotency is built into the resource** — re-running against a compliant node does nothing; you never hand-code the "does this already exist" check yourself
- `Test-DscConfiguration` — checks for drift without applying anything
- `Script` **resource** — the escape hatch; `TestScript` / `SetScript` / `GetScript` reproduce the same idempotent contract by hand
- **Honest scope note:** class-based resources, pull servers, Azure Automation DSC, and DSC v3 are all genuinely beyond this introduction
- **Next chapter:** Testing PowerShell Code with Pester

Every function written across both courses has been verified by eye — run it, read the output, decide it looks right. **Pester** is PowerShell's own built-in testing framework, and it formalizes exactly that instinct into something repeatable: write the expected behavior down once, and let a single command confirm it's still true every time the code changes.

The Structure: `Describe`, `Context`, `It`

```
# Get-Square.Tests.ps1
Describe "Get-Square" {
    It "returns the square of a positive number" {
        Get-Square -n 4 | Should -Be 16
    }
    It "returns 0 for an input of 0" {
        Get-Square -n 0 | Should -Be 0
    }
}
```

`Describe` groups every test for one thing — usually one function. `It` is a single test case, named in plain English describing what's actually being checked. `Context` (not shown above) nests further, grouping related scenarios inside a `Describe` — "when the input is negative," "when the file doesn't exist" — for tests large enough to benefit from that extra layer.

`Should`: Making Assertions

ASSERTION	CHECKS
Should -Be	Equal (case-insensitive for strings)
Should -BeExactly	Equal, case-sensitive
Should -BeNullOrEmpty	Value is `$null` or empty
Should -Contain	A collection contains a given value
Should -Throw	The code block raises a terminating error
Should -Exist	A file or path actually exists

`Should -Be` vs. `Should -BeExactly`: A Familiar Theme, Again

```
It "matches case-insensitively by default" {
    "HELLO" | Should -Be "hello" # passes
}
It "BeExactly requires the same case" {
    "HELLO" | Should -BeExactly "hello" # fails — case must match exactly
}
```

THE CENTRAL FACT THIS CHAPTER IS BUILT ON

This is the exact same theme this whole course keeps returning to: Fundamentals 4's `-eq` was case-insensitive by default, with `-ceq` as the explicit case-sensitive variant. Chapter 2's `-match` was case-insensitive by default, with `-cmatch` as the explicit variant. `Should -Be` follows the identical pattern — case-insensitive comparison by default, with `Should -BeExactly` as the deliberate opt-in for anything genuinely case-sensitive. Once you've internalized it once, it applies consistently everywhere in PowerShell.

Testing Error Conditions: `Should -Throw`

```
It "throws when given a negative number" {
    { Get-Square -n -1 } | Should -Throw
}
```

The code under test has to be wrapped in a script block (`{ ... }`) for `Should -Throw` — running it directly would throw immediately and fail the test *before* `Should` ever gets a chance to check anything.

Mocking: Isolating a Function From Its Real Dependencies

```
Describe "Get-ConfigValue" {
    It "reads a value from the config file" {
        Mock Get-Content { return '{"setting": "value"}' }

        Get-ConfigValue -Name "setting" | Should -Be "value"
# Get-Content never touched a real file — Mock replaced it completely for this
test
    }
}
```

`Mock` replaces a real cmdlet's behavior for the duration of the test — `Get-ConfigValue` can be tested without a real config file existing anywhere on disk, and without the test depending on that file's contents staying the same over time. This is what actually makes a test trustworthy: it fails only when the function's own logic is wrong, never because of something unrelated in the surrounding environment.

Setup & Teardown: `BeforeAll` / `BeforeEach`, and `TestDrive:`

```
Describe "File Cleanup" {
    BeforeAll {
        New-Item -Path "TestDrive:\sample.txt" -ItemType File
    }
    It "the test file exists" {
        "TestDrive:\sample.txt" | Should -Exist
    }
}
```

`TestDrive:` is a real PSDrive — Fundamentals 2's own provider model at work — that Pester creates automatically for each test run and tears down completely afterward, so file-based tests never leave real files scattered on disk. `BeforeAll` runs once before every test in its `Describe`/`Context`; `BeforeEach` runs before *every single* `It`; `AfterAll`/`AfterEach` mirror both for cleanup.

Running Tests: `Invoke-Pester`

```
Invoke-Pester -Path .\Get-Square.Tests.ps1 -Output Detailed
```

THE *.TESTS.PS1 NAMING CONVENTION ISN'T OPTIONAL

Pester's own test discovery specifically looks for files ending in `.Tests.ps1` — a file named `GetSquareTests.ps1` (missing that literal dot before `Tests`) won't be picked up at all when `Invoke-Pester` is run against a folder with no explicit `-Path` to that one file. It's a real, load-bearing naming convention, not a style preference.

A FIRST PRACTICAL HABIT

Write the failing test first — `It "returns 0 for an input of 0" { Get-Square -n 0 | Should -Be 0 }` before `Get-Square` even handles that case correctly. Watching it fail, then pass once the code is fixed, is real proof the test is actually checking something, not just quietly passing regardless of what the code does.

Hands-On Exercises

EXERCISE 1

Explain why `"HELLO" | Should -Be "hello"` passes while `"HELLO" | Should -BeExactly "hello"` fails, tying this back to Fundamentals 4's own `-eq`/`-ceq` comparison behavior.

View solution

EXERCISE 2

Explain what `Mock Get-Content { return '...' }` actually does during a test, and why this matters for writing a test that doesn't depend on a real file existing on disk.

View solution

EXERCISE 3

Explain why a test file named `GetSquareTests.ps1` (without the dot before "Tests") wouldn't be picked up by `Invoke-Pester` run against a folder with no explicit `-Path` to that specific file.

View solution

CHAPTER 10 QUICK REFERENCE

- `Describe` / `Context` / `It` — group tests for a thing, group scenarios within that, and a single named test case
- `Should -Be` — case-insensitive equality; `Should -BeExactly` — case-sensitive, the same pattern as `-eq` / `-ceq` and `-match` / `-cmatch`
- `Should -Throw` — wrap the code in `{ ... }`; testing that it actually raises a terminating error
- `Mock` — replaces a real cmdlet's behavior for a test, isolating the function under test from real files/network/state
- `TestDrive:` — a real, auto-cleaned PSDrive for file-based tests (Fundamentals 2's provider model at work)
- `BeforeAll` / `BeforeEach` / `AfterAll` / `AfterEach` — setup and teardown at different scopes
- `Invoke-Pester` — runs tests; requires the `*.Tests.ps1` naming convention for automatic discovery
- **Next chapter:** Profiles, Customization & Productivity

Fundamentals 11 introduced `$PROFILE` just far enough to make the `ll` function persistent. That was only ever a quarter of the real picture — `$PROFILE` is actually **four** distinct scripts, and picking the wrong one is exactly why a customization can work perfectly in one PowerShell window and mysteriously not exist in another.

`$PROFILE` Isn't Just One File — It's Four

```
$PROFILE # same as CurrentUserCurrentHost — the one Fundamentals 11 used
$PROFILE.CurrentUserCurrentHost   # this user, this specific host only
$PROFILE.CurrentUserAllHosts      # this user, EVERY host — console, VS Code,
ISE, Windows Terminal
$PROFILE.AllUsersCurrentHost      # every user, this specific host only — needs
admin rights
$PROFILE.AllUsersAllHosts         # every user, every host — needs admin rights
```

"Host" here means the actual application running PowerShell — the classic console, VS Code's own integrated terminal, the ISE, Windows Terminal — and PowerShell treats each one as genuinely separate, even though the underlying engine is identical.

> **"IT WORKS IN THE CONSOLE BUT NOT IN VS CODE" IS A PROFILE-SCOPE PROBLEM**
>
> Editing `$PROFILE` (which really means `CurrentUserCurrentHost`) only affects **one specific host**. A customization saved there correctly, working perfectly in the classic console, can appear to simply not exist the moment the exact same account opens VS Code's integrated terminal instead — because that's a different host, reading a different file. This isn't a bug; it's four separate files behaving exactly as documented, once you know they're separate.

Which One Should You Actually Edit?

```
if (-not (Test-Path $PROFILE.CurrentUserAllHosts)) {
    New-Item -ItemType File -Path $PROFILE.CurrentUserAllHosts -Force
}
notepad $PROFILE.CurrentUserAllHosts
```

For anything meant to follow you everywhere — Fundamentals 11's own `ll` function among them — `CurrentUserAllHosts` is almost always the right target, not the plain `$PROFILE` shorthand.

Customizing the Prompt: `prompt` Is Just a Function

```
function prompt {
    "PS $($PWD.Path)> "
}
```

Nothing special about `prompt` beyond its exact reserved name — PowerShell calls it automatically before displaying every new command line, and whatever string it returns *becomes* the prompt text.

> **FUNDAMENTALS 7'S OWN RULE, NOW WITH REAL VISIBLE STAKES**
>
> Fundamentals 7 established that every uncaptured line inside a function joins its return value — here, that rule applies directly to the one function PowerShell calls constantly, in front of you, every single time. A stray `Write-Output "checking git ... "` left inside a custom `prompt` function wouldn't just quietly pollute a variable somewhere — it would visibly break your actual prompt, printing extra unwanted text before it, on every single command. This is the same gotcha as always, just with the highest-visibility consequence possible.

A slightly fancier version, showing the current git branch when there is one:

```
function prompt {
    $path = $PWD.Path
    $branch = git branch --show-current 2>$null
if ($branch) {
        "PS $path [$branch]> "
    } else {
        "PS $path> "
    }
}
```

PSReadLine: the Engine Behind Your Interactive Session

Tab completion, syntax highlighting as you type, and multi-line editing all come from **PSReadLine**, a module loaded by default in every modern PowerShell session:

```
Set-PSReadLineOption -PredictionSource History   # ghost-text suggestions pulled
from your own command history
Set-PSReadLineOption -EditMode Windows             # or Emacs, or Vi — controls
the actual keybinding scheme
Set-PSReadLineOption -Colors @{ Command = 'Cyan'; String = 'Green' }
```

Prediction, History Search & Key Handlers

```
Set-PSReadLineKeyHandler -Key Tab -Function MenuComplete
# Tab now shows a selectable completion menu instead of just cycling through
options one at a time
```

`Set-PSReadLineKeyHandler` remaps individual keys to built-in editing functions — a genuinely deep customization surface, useful even at this single, well-chosen example's level.

> **BEYOND THIS CHAPTER**
>
> A whole ecosystem of third-party prompt theming exists outside core PowerShell — Oh My Posh being the best-known example, adding icons, git status, and full visual themes well beyond what a hand-written `prompt` function alone typically attempts. Worth knowing it exists; genuinely outside this course's own PowerShell-core scope.

Putting It Together

```
# $PROFILE.CurrentUserAllHosts — applies everywhere, for this user
function ll { Get-ChildItem -Force @args }   # Fundamentals 11's own payoff, now
formalized here
function prompt {
    $branch = git branch --show-current 2>$null
if ($branch) { "PS $($PWD.Path) [$branch]> " } else { "PS $($PWD.Path)> " }
}

Set-PSReadLineOption -PredictionSource History
```

Hands-On Exercises

EXERCISE 1

A user reports that customizations saved to `$PROFILE` work perfectly in the regular PowerShell console but don't appear at all in VS Code's integrated terminal. Explain exactly why, and name the specific `$PROFILE` property that fixes it.

View solution

EXERCISE 2

Explain what the `prompt` function actually is, and why a stray `Write-Output` line inside a custom `prompt` function would visibly break what your prompt looks like — referencing Fundamentals 7's own uncaptured-output rule directly.

View solution

EXERCISE 3

Write a `function prompt` that shows the current path, and — only when inside a git repository — the current branch name in brackets right after it.

View solution

CHAPTER 11 QUICK REFERENCE

- `$PROFILE` **is four files** — CurrentUserCurrentHost (the default shorthand), CurrentUserAllHosts, AllUsersCurrentHost, AllUsersAllHosts
- **"Host"** — the actual application running PowerShell (console, VS Code terminal, ISE, Windows Terminal); each is genuinely separate
- `$PROFILE.CurrentUserAllHosts` — the right target for anything meant to follow you across every host
- `function prompt { ... }` — a specially-named function PowerShell calls before every command line; its return value becomes the prompt text
- **Fundamentals 7's uncaptured-output rule applies to** `prompt` **directly** — a stray line there visibly corrupts your actual prompt, every time
- **PSReadLine** — the module behind tab completion, syntax highlighting, and history-based prediction (`Set-PSReadLineOption`, `Set-PSReadLineKeyHandler`)
- **Next chapter:** Capstone — A Multi-System Automation & Reporting Tool

Dana's own `Get-OldFileReport`, from the Fundamentals capstone, checked one folder on one machine. The team has since grown to a small fleet of servers, and Dana's been asked for something that scales to all of them at once, reports out automatically, and — since other people now depend on it — is actually tested rather than just trusted. `Get-FleetHealthReport` is that tool, and building it touches every chapter in this course.

STEP 1 — VALIDATING SERVER NAMES BEFORE TOUCHING THE NETWORK

The function starts as a real advanced function (Chapter 1) — `[CmdletBinding(SupportsShouldProcess)]`, a mandatory, pipeline-bound, non-empty `-ComputerName` parameter. Before any server is ever queried, each name is checked against the team's own naming convention with a regex (Chapter 2) — a malformed entry is rejected immediately, with a clear warning, rather than discovered mid-query as a confusing connection failure.

STEP 2 — QUERYING EVERY SERVER AT ONCE, SAFELY

`ForEach-Object -Parallel` (Chapter 4) launches a `Get-CimInstance` query (Chapter 6) against every validated server concurrently — and because CIM travels over the same WinRM infrastructure Chapter 3's own remoting established, no separate connection setup is needed beyond what `Enable-PSRemoting` already provides on each target. Results land in a `ConcurrentBag`, specifically avoiding Chapter 4's own isolated-runspace gotcha where a plain shared array would have silently stayed empty.

STEP 3 — BUILDING THE REPORT: STRINGBUILDER AND A REAL EXCEL WORKBOOK

A quick verbose summary is assembled with `StringBuilder` (Chapter 5) rather than repeated string concatenation. The full report also gets written to a real Excel workbook via COM automation — with Chapter 5's own cleanup discipline applied without exception: `.Quit()` and `Marshal.ReleaseComObject()` sit inside a `finally` block, so no orphaned `EXCEL.EXE` is left behind even if writing the workbook fails partway through.

STEP 4 — REPORTING OUT VIA A WEBHOOK

The same results, converted with `ConvertTo-Json`, get posted to a monitoring webhook with `Invoke-RestMethod` (Chapter 7) — wrapped in `try`/`catch` that needs no `-ErrorAction Stop`, since HTTP failures are terminating by default, reading `$_.ErrorDetails.Message` for the real reason if the post fails.

The Complete Function

```
function Get-FleetHealthReport {
    [CmdletBinding(SupportsShouldProcess)]
    param(
        [Parameter(Mandatory, ValueFromPipeline)]
        [ValidateNotNullOrEmpty()]
        [string[]]$ComputerName,

        [string]$WebhookUri,
        [string]$ExcelPath = ".\FleetHealthReport.xlsx"
    )

    begin {
        $namePattern = '^[A-Z]{2,4}-\d{3}$'
$results = [System.Collections.Concurrent.ConcurrentBag[object]]::new()
    }

    process {
        foreach ($name in $ComputerName) {
            if ($name -notmatch $namePattern) {
                Write-Warning "Skipping '$name' - doesn't match the expected
naming pattern"
continue
            }

            $name | ForEach-Object -Parallel {
                $server = $_
$bag = $using:results
try {
                    $disk = Get-CimInstance -ClassName Win32_LogicalDisk -
ComputerName $server -Filter "DeviceID='C:'" -ErrorAction Stop
                    $svc  = Get-CimInstance -ClassName Win32_Service -
ComputerName $server -Filter "Name='Spooler'" -ErrorAction Stop
```

```
                    $bag.Add([PSCustomObject]@{
                        ComputerName = $server
                        FreeGB       = [math]::Round($disk.FreeSpace / 1GB, 2)
                        ServiceState = $svc.State
                    })
                } catch {
                    $bag.Add([PSCustomObject]@{
                        ComputerName = $server
                        FreeGB       = $null
                        ServiceState = "Unreachable: $($_.Exception.Message)"
                    })
                }
            } -ThrottleLimit 10
        }
    }

    end {
        $sb = [System.Text.StringBuilder]::new()
        foreach ($r in $results) {
            [void]$sb.AppendLine("$($r.ComputerName): $($r.FreeGB) GB free,
Spooler is $($r.ServiceState)")
        }
        Write-Verbose $sb.ToString()

        if ($PSCmdlet.ShouldProcess($ExcelPath, "Write Excel report")) {
            $excel = New-Object -ComObject Excel.Application
            try {
                $excel.Visible = $false
$workbook = $excel.Workbooks.Add()
                $sheet = $workbook.Worksheets.Item(1)
                $sheet.Cells.Item(1, 1) = "ComputerName"
$sheet.Cells.Item(1, 2) = "FreeGB"
$sheet.Cells.Item(1, 3) = "ServiceState"
$row = 2
                foreach ($r in $results) {
                    $sheet.Cells.Item($row, 1) = $r.ComputerName
                    $sheet.Cells.Item($row, 2) = $r.FreeGB
                    $sheet.Cells.Item($row, 3) = $r.ServiceState
                    $row++
                }
                $workbook.SaveAs($ExcelPath)
            } finally {
                $excel.Quit()

[System.Runtime.InteropServices.Marshal]::ReleaseComObject($excel) | Out-Null
            }
        }

        if ($WebhookUri) {
```

```
            $body = $results | ConvertTo-Json -Depth 3
            try {
                Invoke-RestMethod -Uri $WebhookUri -Method Post -Body $body -
ContentType "application/json"
            } catch {
                Write-Warning "Webhook post failed: $($_.ErrorDetails.Message)"
            }
        }

        $results
    }
}
```

STEP 5 — PACKAGING IT AS A REAL MODULE

The function moves into `FleetHealth.psm1`, with `Export-ModuleMember` and the manifest's own `FunctionsToExport` (Chapter 8) both restricted to just `Get-FleetHealthReport` — nothing internal leaks out as if it were part of the public API.

```
# FleetHealth.psm1 — ends with:
Export-ModuleMember -Function Get-FleetHealthReport

# Manifest:
New-ModuleManifest -Path .\FleetHealth\FleetHealth.psd1 `
    -RootModule "FleetHealth.psm1" -ModuleVersion "1.0.0" `
    -FunctionsToExport @("Get-FleetHealthReport")
```

STEP 6 — TESTING THE PARTS THAT DON'T NEED A REAL SERVER

A Pester test (Chapter 10) confirms the naming-pattern rejection actually works, by mocking `Get-CimInstance` and asserting it's never called for a malformed name — the test suite never touches a real server:

```
# FleetHealth.Tests.ps1
Describe "Get-FleetHealthReport" {
    It "never queries a name that fails the naming pattern" {
        Mock Get-CimInstance
        Get-FleetHealthReport -ComputerName "not-a-real-name" -
WarningAction SilentlyContinue | Out-Null
Should -Invoke Get-CimInstance -Times 0
    }
}
```

STEP 7 — MAKING IT INSTANTLY AVAILABLE

A reference to the module, plus a short convenience wrapper, goes into `$PROFILE.CurrentUserAllHosts` (Chapter 11) — available in every host, every session, from here on:

```
Import-Module FleetHealth
function fleet { Get-FleetHealthReport -ComputerName (Get-Content
.\servers.txt) }
```

Chapter Attribution

STEP	CHAPTER(S) APPLIED
1 — Validating names	Chapter 1 (CmdletBinding, ValidateNotNullOrEmpty, pipeline input), Chapter 2 (regex)
2 — Querying in parallel	Chapter 3 (WinRM foundation CIM relies on), Chapter 4 (ForEach-Object -Parallel, ConcurrentBag), Chapter 6 (Get-CimInstance)
3 — The report	Chapter 5 (StringBuilder, COM automation with proper cleanup)
4 — Webhook reporting	Chapter 7 (Invoke-RestMethod, terminating HTTP errors, ErrorDetails)
5 — Packaging	Chapter 8 (Export-ModuleMember, FunctionsToExport)
6 — Testing	Chapter 10 (Describe/It, Mock, Should -Invoke)
7 — Availability	Chapter 11 ($PROFILE.CurrentUserAllHosts)

WHAT THIS WHOLE COURSE WAS REALLY ABOUT

From Chapter 1's advanced parameters to this capstone's webhook post, real object fidelity never broke down at any step — a CIM result stayed a real object across a parallel runspace boundary, survived being written into Excel, and only became text at the one deliberate point (`ConvertTo-Json`) where crossing an actual network genuinely required it. That's the same throughline Fundamentals 1 opened this entire two-course series with, now carrying a tool that queries a fleet, reports automatically, and — because it's tested and packaged — other people can actually trust without reading its source first.

HONEST SCOPE NOTE

DSC (Chapter 9) is deliberately not part of this capstone — `Get-FleetHealthReport` reports on existing state, it doesn't enforce configuration, which is a genuinely different job DSC was built for, not an oversight. The worked example above is illustrative code, not a script run against real infrastructure in the writing of this chapter. And nothing here covers long-term monitoring-platform integration beyond the single webhook POST shown — a real alerting pipeline (thresholds, escalation, history) is its own, larger topic.

Hands-On Exercises

EXERCISE 1

Explain why `Get-FleetHealthReport` checks each computer name against a regex *before* the `ForEach-Object -Parallel` block runs, rather than letting a malformed name simply fail whatever CIM query eventually tries it.

View solution

EXERCISE 2

Explain why the Excel-writing block's `.Quit()` and `ReleaseComObject()` calls sit inside a `finally` block rather than just after the `try`. What could happen if they didn't?

View solution

EXERCISE 3

Explain what the Pester test in Step 6 is actually verifying, and why mocking `Get-CimInstance` — rather than running the function against a real server — is the right way to test that specific piece of behavior.

View solution

CHAPTER 12 QUICK REFERENCE — COURSE COMPLETE

- **7 build steps, 11 prior chapters** — one real, multi-system tool, tested and packaged, built in the order a genuine advanced automation project would actually hit each concept
- **This course's own throughline, closed out:** real object fidelity preserved across parallel runspaces, remote queries, .NET automation, and a network API call — the same discipline Fundamentals 1 began, now proven at fleet scale
- **Honest scope note:** no DSC (a genuinely different job), illustrative code only, no full monitoring-platform integration
- **The PowerShell Intermediate/Advanced course is now complete** — 12/12 chapters