Podstawy składni PowerShella

Poziom: podstawy **Dotyczy: Windows PowerShell 5.1, PowerShell 7+**

Czas: ~30 min

Konwencja nazw, zmienne, tablice i sześć poleceń potoku, które razem odpowiadają za dziewięćdziesiąt procent tego, co realnie piszemy w PowerShellu na co dzień.

Skróty użyte na tej stronie (6)

Skrót	Co znaczy
API	Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą
BOM	Byte Order Mark: znacznik na początku pliku tekstowego UTF-8, który dopisuje Excel
CSV	w Windowsie: Cluster Shared Volume, dysk widoczny naraz dla wszystkich węzłów klastra; w danych: Comma-Separated Values, plik z wartościami oddzielonymi przecinkami
JSON	JavaScript Object Notation: tekstowy format danych z klamrami i cudzysłowami
REST	sposób budowania API przez zwykłe żądania HTTP (GET, POST) zamiast osobnego protokołu
XML	eXtensible Markup Language: format tekstowy do zapisu ustrukturyzowanych danych w znacznikach, tutaj używany przez PowerShell do zapisu zrzutu obiektów poleceniem Export-Clixml

Fiszka

TYLKO ODCZYT

Znalezienie polecenia po fragmencie nazwy: Get-Command

```
Get-Command -Verb Get -Noun *service*
```

Szukamy poleceń pasujących do wzorca w rzeczowniku, bez konieczności znajomości pełnej nazwy — przydatne, gdy pamiętamy tylko, czego dotyczy polecenie, nie jak dokładnie się

nazywa.

Przykład:

```
CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Cmdlet          Get-Service                                        3.1.0.0
Microsoft.PowerShell.Management
Cmdlet          Get-CimAssociatedInstance                          1.0.0.0    CimCmdlets
```

TYLKO ODCZYT

Wyświetlenie pomocy do polecenia z przykładami: Get-Help -Examples

```
Get-Help Get-Service -Examples
```

Ograniczamy wynik do samych przykładów użycia, pomijając pełny, długi opis parametrów — najszybszy sposób, żeby zobaczyć, jak dane polecenie się realnie wywołuje.

Przykład:

```
NAME
    Get-Service

    ------- Example 1 -------
    PS C:\> Get-Service "s*"

    Gets services on the system whose service name begins with s.

    ------- Example 2 -------
    PS C:\> Get-Service | Where-Object {$_.Status -eq "Running"}

    Gets only the services with a status of "Running".
```

TYLKO ODCZYT

Wyświetlenie pełnej pomocy do polecenia: Get-Help -Full

```
Get-Help Get-Service -Full
```

Dostajemy komplet: opis, składnię ze wszystkimi zestawami parametrów, szczegółowy opis każdego parametru z typem i wartością domyślną oraz przykłady na końcu.

□□ ZMIENIA SYSTEM

Pobranie i zaktualizowanie plików pomocy: Update-Help

```
Update-Help -Force
```

> **Pojęcie: pliki pomocy.** PowerShell nie przechowuje treści `Get-Help` w samym silniku — pobiera ją osobno, z internetu, jako paczki XML przypisane do konkretnych modułów. Świeżo zainstalowany system bez `Update-Help` zwraca dla większości poleceń wyłącznie okrojoną pomoc wygenerowaną automatycznie, bez przykładów i bez szczegółowych opisów parametrów. Wymaga uprawnień administratora i działającego połączenia z internetem.

□□ **XML** = eXtensible Markup Language: format tekstowy do zapisu ustrukturyzowanych danych w znacznikach, tutaj używany przez PowerShell do zapisu zrzutu obiektów poleceniem Export-Clixml

□□ TYLKO ODCZYT

Wylistowanie wszystkich aliasów wskazujących na jedno polecenie: Get-Alias

```
Get-Alias -Definition Get-ChildItem
```

> **Pojęcie: alias.** Skrócona nazwa polecenia — `ls`, `dir`, `gci` to trzy różne aliasy tego samego polecenia `Get-ChildItem`, wygodne przy pracy interaktywnej, bo krótsze do wpisania. W skryptach ich nie używamy z dwóch powodów: alias może nie istnieć na maszynie, na której skrypt uruchomi ktoś inny (część aliasów pochodzi z modułów, które trzeba najpierw zaimportować), a poza tym czytelność kodu dla kogoś, kto nie zna wszystkich aliasów na pamięć, drastycznie spada — `gps | ? CPU -gt 50 | select Name` czyta się dużo gorzej niż `Get-Process | Where-Object CPU -gt 50 | Select-Object Name`.

Przykład:

```
CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Alias           dir -> Get-ChildItem
Alias           gci -> Get-ChildItem
Alias           ls -> Get-ChildItem
```

CPU = Central Processing Unit: procesor

TYLKO ODCZYT

Sprawdzenie typu wartości zmiennej: GetType()

```
$liczba = 42
$liczba.GetType().Name
```

Sprawdzamy realny typ .NET stojący za wartością — PowerShell wnioskuje typ automatycznie przy przypisaniu, jeśli nie zadeklarowaliśmy go jawnie.

Przykład:

```
Int32
```

TYLKO ODCZYT

Deklaracja zmiennej z jawnym typem

```
[int]$port = "8080"
```

Wymuszamy konwersję na konkretny typ .NET już w momencie przypisania — próba wpisania wartości niekonwertowalnej na ten typ (`[int]$port = "abc"`) kończy się błędem od razu, zamiast dopiero przy późniejszym użyciu zmiennej.

TYLKO ODCZYT

Sprawdzenie, czy zmienna jest pusta: $null

```
if ($null -eq $wynik) {
    Write-Host "brak wyniku"
}
```

Pojęcie: $null. Zmienna automatyczna reprezentująca brak wartości — odpowiednik `NULL` w bazach danych, nie to samo co pusty ciąg znaków (`""`) ani zero. Porównanie `$null -eq $wynik` zapisujemy z `$null` po lewej stronie celowo: jeśli `$wynik` okaże się tablicą, `$wynik -eq $null` porówna każdy element tablicy z osobna i może dać zaskakujący wynik, podczas gdy `$null -eq $wynik` zawsze zwraca pojedynczą wartość logiczną.

TYLKO ODCZYT

Utworzenie tablicy i odczyt elementu: @()

```
$serwery = @("fs01", "rds01", "wsus01")
$serwery[0]
```

Pojęcie: tablica. Uporządkowana lista wartości pod jedną zmienną, indeksowana od zera — `$serwery[0]` to pierwszy element, `$serwery[-1]` to ostatni, `$serwery[1..2]` to zakres. Standardowy typ tablicy w PowerShellu (`[object[]]`) przechowuje elementy dowolnego typu naraz, nie tylko jednego.

Przykład:

```
fs01
```

TYLKO ODCZYT

Utworzenie tablicy haszującej: @{}

```
$konfiguracja = @{
    Serwer = "docker01"
    Port   = 443
    SSL    = $true
}
$konfiguracja["Port"]
```

Pojęcie: tablica haszująca. Zbiór par klucz-wartość (`hashtable`), odpowiednik słownika — odczyt po kluczu jest natychmiastowy, niezależnie od liczby wpisów, w przeciwieństwie do przeszukiwania tablicy po wartości. Klucze i wartości mogą być

dowolnego typu, a same tablice haszujące są tym, czego używamy do parametrów wywołania funkcji przez `splatting`, opisanego na stronie Skrypty PowerShell: struktura i dobre praktyki.

Przykład:

```
443
```

SSL = Secure Sockets Layer: dawna nazwa szyfrowania, dziś formalnie TLS

TYLKO ODCZYT

Odwołanie do bieżącego elementu potoku: $_ oraz $PSItem

```
Get-Process | ForEach-Object { "$($_.ProcessName) zajmuje $($_.WorkingSet64 / 1MB) MB" }
```

Pojęcie: $_ / $PSItem. Zmienna automatyczna dostępna wewnątrz bloku skryptu przetwarzającego element potoku (`ForEach-Object`, `Where-Object`, blok `process` w funkcji zaawansowanej) — reprezentuje bieżący element, który akurat przepływa przez potok. `$PSItem` to dosłowny synonim `$_`, wprowadzony dla czytelności w zagnieżdżonych blokach, gdzie kilka `$_` z różnych poziomów mogłoby się mylić.

TYLKO ODCZYT

Filtrowanie elementów potoku: Where-Object

```
Get-Service | Where-Object Status -eq "Stopped"
```

Pojęcie: potok. Mechanizm przekazywania obiektów z wyjścia jednego polecenia na wejście kolejnego, opisany szerzej na stronie CMD, PowerShell, PowerShell Core: trzy powłoki, trzy różne światy. `Where-Object` przepuszcza dalej tylko te obiekty, dla których podany warunek jest prawdziwy — składnia uproszczona (`Status -eq "Stopped"`) działa dla prostych porównań jednej właściwości, składnia pełna z blokiem `{ $_.Status -eq "Stopped" }` jest potrzebna przy warunkach złożonych z kilku właściwości naraz.

Przykład:

```
Status   Name               DisplayName
------   ----               -----------
Stopped  wuauserv           Windows Update
Stopped  bits               Background Intelligent Transfer Service
```

CMD = skrót od polecenia command, dawna nazwa wiersza poleceń Windows sprzed PowerShella

TYLKO ODCZYT

Wybranie konkretnych właściwości obiektu: Select-Object

```
Get-Process | Select-Object -Property ProcessName, Id, CPU -First 5
```

Ograniczamy widoczne właściwości obiektu do wskazanych — obiekt oryginalny wciąż istnieje w pamięci ze wszystkimi właściwościami, ale dalej w potoku (i na ekranie) widać wyłącznie wybrane. `-First 5` ogranicza liczbę zwróconych obiektów, niezależnie od tego, ile ich faktycznie istnieje.

Przykład:

```
ProcessName        Id        CPU
-----------        --        ---
chrome           8420     412.34
explorer         3212      88.11
svchost          1044      12.05
MsMpEng          2104     198.77
dwm              1608      45.90
```

CPU = Central Processing Unit: procesor

TYLKO ODCZYT

Posortowanie wyników: Sort-Object

```
Get-Process | Sort-Object -Property WorkingSet64 -Descending | Select-Object -First 3
ProcessName, WorkingSet64
```

Sortujemy po realnej wartości właściwości numerycznej, nie po jej tekstowej reprezentacji — `Sort-Object` na liczbach porządkuje `2`, `10`, `100` poprawnie, podczas gdy sortowanie tekstowe umieściłoby `10` przed `2`.

Przykład:

```
ProcessName        WorkingSet64
-----------        ------------
chrome                487194112
sqlservr              412876800
MsMpEng               298765312
```

TYLKO ODCZYT

Policzenie, zsumowanie i uśrednienie wartości: Measure-Object

```
Get-ChildItem C:\Logs -File | Measure-Object -Property Length -Sum -Average -Maximum
```

> **Pojęcie: Measure-Object.** Polecenie liczące podstawowe statystyki na wskazanej właściwości numerycznej przepływających przez potok obiektów — bez parametrów zwraca tylko liczbę obiektów (`Count`), z `-Sum` / `-Average` / `-Maximum` / `-Minimum` dokłada odpowiednie agregaty.

Przykład:

```
Count             : 214
Average           : 1048576.34
Sum               : 224395897
Maximum           : 52428800
Minimum           : 128
Property          : Length
```

TYLKO ODCZYT

Grupowanie obiektów po wartości właściwości: Group-Object

```
Get-Process | Group-Object -Property Company | Sort-Object Count -Descending | Select-Object Count, Name -First 5
```

Zliczamy, ile obiektów w potoku ma tę samą wartość wskazanej właściwości — odpowiednik `GROUP BY` z SQL-a, ale bez konieczności pisania osobnego agregatu na każdą kolumnę.

Przykład:

```
Count Name
----- ----
   34 Microsoft Corporation
   12 Google LLC
    8
    3 Mozilla Corporation
```

□□ TYLKO ODCZYT

Wykonanie akcji na każdym elemencie potoku: ForEach-Object

```
Get-ChildItem C:\Logs -Filter *.log | ForEach-Object {
    Compress-Archive -Path $_.FullName -DestinationPath "$($_.FullName).zip"
}
```

`ForEach-Object` uruchamia podany blok skryptu raz na każdy obiekt, który przez niego przepływa — w przeciwieństwie do pętli `foreach` (bez myślnika), która wymaga wcześniej zebranej w pamięci kolekcji, `ForEach-Object` przetwarza obiekty w miarę ich napływania z potoku, jeden po drugim, co ma znaczenie przy bardzo dużych zbiorach danych.

□□ TYLKO ODCZYT

Formatowanie wyniku jako tabeli: Format-Table

```
Get-Service | Format-Table -Property Name, Status, StartType -AutoSize
```

> **Pojęcie: rodzina Format-*.** `Format-Table`, `Format-List`, `Format-Wide` i `Format-Custom` przekształcają żywe obiekty w gotowy do wyświetlenia tekst — od tego momentu w potoku nie ma już obiektów z właściwościami, jest wyłącznie sformatowany napis. Dlatego `Format-*` musi być ostatnim poleceniem w potoku: cokolwiek postawimy po nim (`Where-Object`, `Sort-Object`, `Select-Object`) będzie już operować na tekście, nie na strukturze danych, i albo zwróci błąd, albo da wynik zupełnie inny od zamierzonego.

□□ TYLKO ODCZYT

Formatowanie wyniku jako listy pionowej: Format-List

```
Get-Process -Name explorer | Format-List -Property *
```

Wypisujemy wszystkie właściwości obiektu jedna pod drugą zamiast w kolumnach — nieocenione przy obiektach z wieloma właściwościami albo z długą treścią w jednej z nich, które w tabeli łamałyby się na ekranie.

Przykład:

```
Name                : explorer
Id                  : 3212
CPU                 : 88.109375
WorkingSet64        : 92471296
StartTime           : 24.09.2026 07:12:08
Path                : C:\Windows\explorer.exe
```

□□ TYLKO ODCZYT

Otwarcie wyniku w interaktywnej tabeli graficznej: Out-GridView

```
Get-Process | Out-GridView -Title "Procesy" -PassThru
```

Otwieramy osobne okno z sortowalną, filtrowalną tabelą — dostępne wyłącznie w Windows PowerShell 5.1 na systemach z interfejsem graficznym (wymaga modułu `Microsoft.PowerShell.GraphicalTools` w PowerShell 7). Parametr `-PassThru` sprawia, że zaznaczone w oknie wiersze wracają do potoku po zamknięciu okna, zamiast okno być ślepym zaułkiem.

□□ TYLKO ODCZYT

Eksport do pliku CSV: Export-Csv

```
Get-Process | Select-Object ProcessName, Id, CPU | Export-Csv -Path
C:\Raporty\procesy.csv -NoTypeInformation -Encoding UTF8
```

`-NoTypeInformation` pomija pierwszą linię z nazwą typu .NET (`#TYPE System.Diagnostics.Process`), którą starsze wersje PowerShella dopisywały domyślnie — bez tego parametru większość programów odczytujących CSV (w tym Excel) dostaje dodatkowy, niepotrzebny wiersz na samej górze pliku.

CSV = w Windowsie: Cluster Shared Volume, dysk widoczny naraz dla wszystkich węzłów klastra; w danych: Comma-Separated Values, plik z wartościami oddzielonymi przecinkami ·**CPU** = Central Processing Unit: procesor

TYLKO ODCZYT

Konwersja obiektu na format JSON: ConvertTo-Json

```
Get-Service -Name wuauserv | Select-Object Name, Status, StartType | ConvertTo-Json
```

Zamieniamy obiekt na tekstową reprezentację JSON — przydatne przy przekazywaniu danych do API REST albo zapisywaniu konfiguracji w formacie czytelnym dla innych narzędzi niż tylko PowerShell.

Przykład:

```
{
    "Name":  "wuauserv",
    "Status":  4,
    "StartType":  3
}
```

Domyślna głębokość zagnieżdżenia to dwa poziomy — obiekty zagnieżdżone głębiej wymagają jawnego `-Depth`, inaczej zamieniają się w napis `System.Object[]` zamiast rozwiniętej struktury.

JSON = JavaScript Object Notation: tekstowy format danych z klamrami i cudzysłowami ·**API** = Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą ·**REST** = sposób budowania API przez zwykłe żądania HTTP (GET, POST) zamiast osobnego protokołu

TYLKO ODCZYT

Zapisanie wyniku do pliku tekstowego: Out-File

```
Get-Service | Out-File -FilePath C:\Raporty\uslugi.txt -Encoding UTF8
```

> **Pojęcie: kodowanie w Out-File.** Domyślne kodowanie `Out-File` różni się między silnikami — Windows PowerShell 5.1 domyślnie zapisuje w UTF-16LE z BOM, PowerShell 7 domyślnie w UTF-8 bez BOM. Skrypt, który ma działać identycznie pod obiema wersjami i którego wynik trafia do narzędzia wrażliwego na kodowanie (na przykład starszy parser CSV albo `cmd.exe`), powinien podawać `-Encoding` jawnie,

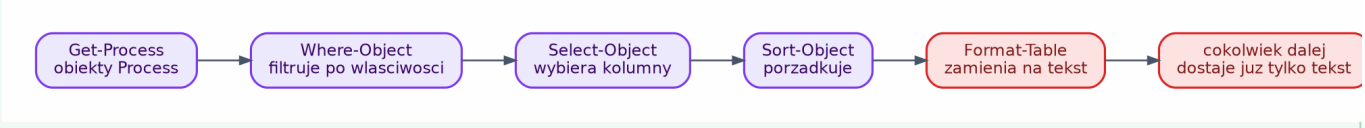
zamiast polegać na wartości domyślnej.

BOM = Byte Order Mark: znacznik na początku pliku tekstowego UTF-8, który dopisuje Excel · **CSV** = w Windowsie: Cluster Shared Volume, dysk widoczny naraz dla wszystkich węzłów klastra; w danych: Comma-Separated Values, plik z wartościami oddzielonymi przecinkami

TYLKO ODCZYT

Wszystkie polecenia potoku z tej strony

Polecenie	Rola
`Where-Object`	filtrowanie obiektów po warunku
`Select-Object`	wybór właściwości i ograniczenie liczby wyników
`Sort-Object`	sortowanie po właściwości
`Measure-Object`	liczenie, suma, średnia, minimum, maksimum
`Group-Object`	grupowanie po wartości właściwości
`ForEach-Object`	wykonanie akcji na każdym elemencie
`Format-Table` / `Format-List`	sformatowanie do wyświetlenia — zawsze na końcu potoku
`Out-GridView`	interaktywna tabela graficzna
`Export-Csv` / `ConvertTo-Json` / `Out-File`	zapis wyniku poza konsolę



Kolejne polecenia potoku przekazuja dalej obiekty, az Format-Table zamieni je na tekst i kolejne polecenie dostanie juz tylko napis (kliknięcie otwiera pełny rozmiar)

Uważaj

- `Format-*` na końcu potoku to reguła bez wyjątków — cokolwiek stoi po nim, operuje już na tekście, nie na obiektach.
- `ConvertTo-Json` bez `-Depth` obcina głęboko zagnieżdżone struktury do napisu `System.Object[]` bez ostrzeżenia.

- Aliasy (`%` dla `ForEach-Object`, `?` dla `Where-Object`, `select` dla `Select-Object`) są wygodne interaktywnie i niepożądane w skryptach współdzielonych z innymi osobami.

Z życia wzięte

Z ŻYCIA WZIĘTE

Tomek dostaje zadanie zrobienia szybkiego raportu z fs01 o dziesięciu największych plikach w udziale działu księgowości. Składa polecenie z `Format-Table` w środku, bo tak wygląda gotowo, a potem dokleja `Sort-Object` na końcu, żeby posortować po rozmiarze. Wynik wychodzi posortowany alfabetycznie po pierwszej kolumnie tekstu, nie po rozmiarze pliku, bo `Sort-Object` dostaje już gołe linijki, nie obiekty `FileInfo`.

Po co to istnieje

Pojęcie: konwencja Czasownik-Rzeczownik. Każde polecenie PowerShella (`cmdlet`) nazywa się parą słów rozdzieloną myślnikiem — czasownik z ustalonej, zamkniętej listy (`Get`, `Set`, `New`, `Remove`, `Start`, `Stop`, `Test`, `Invoke`...) i rzeczownik opisujący, na czym operuje (`Service`, `Process`, `ADUser`, `NetFirewallRule`). Dzięki temu `Get-Service`, `Set-Service`, `Restart-Service` i `Stop-Service` są od razu rozpoznawalne jako rodzina poleceń działających na tym samym obiekcie, a osoba, która nigdy nie widziała konkretnego modułu, potrafi się domyślić, czego szukać — jeśli istnieje `Get-ADUser`, prawie na pewno istnieje też `New-ADUser` i `Set-ADUser`. To jest zamierzona różnica względem basha, gdzie `ls`, `cp`, `rm`, `chmod` nie mają żadnej wspólnej konwencji nazewniczej między sobą.

Ta strona zbiera elementy składni, bez których żaden skrypt PowerShella się nie obejdzie — od znalezienia właściwego polecenia, przez podstawowe struktury danych, po sześć poleceń potoku, które w praktyce pokrywają większość codziennej pracy administracyjnej.

Jak to działa

`Get-Command` i `Get-Help` odpowiadają na dwa różne pytania. `Get-Command` mówi „jakie polecenie istnieje, które robi X" — przeszukuje zainstalowane moduły po wzorcu w czasowniku, rzeczowniku albo pełnej nazwie. `Get-Help` mówi „jak dokładnie użyć polecenia, które już znam z nazwy" — czyta

plik pomocy przypisany do konkretnego polecenia. Oba działają lepiej po jednorazowym `Update-Help`, bo świeżo zainstalowany PowerShell ma tylko automatycznie wygenerowaną, ubogą pomoc.

Tablice i tablice haszujące różnią się dostępem: tablica (`@()`) trzyma kolejność i odczytujemy z niej po pozycji liczbowej, tablica haszująca (`@{}`) nie gwarantuje kolejności (poza `[ordered]@{}`, który ją wymusza) i odczytujemy z niej po nazwanym kluczu. Wybór między nimi wynika z pytania: czy naturalnie myślimy o tych danych jako o liście, czy jako o zestawie nazwanych ustawień.

Częste błędy

Użycie aliasu w skrypcie, który trafia do repozytorium wspólnego z zespołem. `gci`, `%`, `?`, `select` działają identycznie jak pełne nazwy w danej chwili, ale utrudniają czytanie kodu osobie, która danego aliasu nie zna na pamięć, a przy migracji między silnikami część aliasów (zwłaszcza tych z modułów niewbudowanych) może w ogóle nie istnieć.

Postawienie `Where-Object` albo `Sort-Object` po `Format-Table`. Skrypt zwraca błąd albo, gorzej, wynik, który wygląda niepoprawnie, ale nie rzuca żadnego wyjątku — bo `Format-Table` już zamieniło obiekty na tekst, a kolejne polecenie potoku dostaje do przetworzenia gołe linijki znaków zamiast struktury danych.

`ConvertTo-Json` na strukturze zagnieżdżonej głębiej niż dwa poziomy bez `-Depth`. Wynik zawiera napis `"System.Object[]"` w miejscu, gdzie powinny być dane — łatwe do przeoczenia, jeśli nie sprawdzimy wyniku ręcznie przed wysłaniem go dalej.

□□ Ćwiczenia

> Wszystko poniżej wykonujemy we własnej sesji PowerShell, bez uprawnień administratora.

Ćwiczenie 1 — od surowych danych do raportu

```
Get-Process |
    Where-Object CPU -gt 10 |
    Select-Object ProcessName, Id, CPU, WorkingSet64 |
    Sort-Object CPU -Descending |
    Export-Csv -Path "$env:TEMP\procesy_cpu.csv" -NoTypeInformation -Encoding UTF8
```

```
Import-Csv "$env:TEMP\procesy_cpu.csv" | Select-Object -First 5
```

Budujemy pełny łańcuch: filtrowanie, wybór kolumn, sortowanie, zapis do pliku, a na końcu ponowny odczyt tego pliku.

Odpowiedź

`Import-Csv` odtwarza obiekty z pliku, ale wszystkie właściwości wracają jako tekst (`System.String`), nie jako liczby — `CPU` i `WorkingSet64` po ponownym imporcie nie da się już posortować numerycznie bez jawnej konwersji `[double]`/`[long]`, mimo że w oryginalnym potoku były prawdziwymi liczbami. To jest praktyczna konsekwencja tego, że CSV jako format nie przenosi typów danych, tylko sam tekst.

Ćwiczenie 2 — pułapka Format-Table na końcu potoku

```
Get-Service | Format-Table Name, Status | Where-Object Status -eq "Running"
```

Dlaczego to polecenie zwraca błąd albo pusty wynik, mimo że w systemie na pewno działają jakieś usługi?

Odpowiedź

`Format-Table` zamieniło strumień obiektów `ServiceController` na gotowy tekst tabeli, więc `Where-Object` dostaje do przetworzenia obiekty formatujące (`Microsoft.PowerShell.Commands.Internal.Format.FormatEntryData`), a nie usługi z właściwością `Status` — filtr po `Status -eq "Running"` nie znajduje takiej właściwości i albo nic nie przepuszcza, albo zgłasza błąd, w zależności od wersji silnika. Poprawna kolejność to filtrowanie i wybór właściwości najpierw, formatowanie na samym końcu.

Pytania kontrolne

1. Skąd PowerShell wie, że po wpisaniu `Get-` i naciśnięciu Tab ma podpowiedzieć `Get-Service`, `Get-Process` czy `Get-ChildItem`, mimo że nigdy wcześniej nie widzieliśmy pełnej listy poleceń w systemie?

Odpowiedź

Konwencja Czasownik-Rzeczownik jest wymuszona przez sam silnik PowerShella przy rejestracji poleceń w module — dopełnianie tabulatorem (`tab completion`) po prostu przeszukuje wszystkie zarejestrowane nazwy poleceń pasujące do wpisanego prefiksu, korzystając z tego samego mechanizmu, którym operuje `Get-Command`.

2. Dlaczego `Measure-Object -Property Length -Sum` na pustej kolekcji plików nie rzuca błędu, tylko zwraca `Sum` równe zero?

Odpowiedź

`Measure-Object` traktuje brak elementów w potoku jako prawidłowy, choć zerowy, zbiór danych do zagregowania — `Count` wynosi `0`, a suma po zerze elementów matematycznie wynosi zero, więc polecenie zwraca wynik zamiast błędu. To odróżnia je od operacji, które zakładają istnienie co najmniej jednego elementu, jak odwołanie do indeksu tablicy poza jej zakresem.

Powiązane: CMD, PowerShell, PowerShell Core: trzy powłoki, trzy różne światy · Polityka wykonywania i podpisywanie skryptów · Skrypty PowerShell: struktura i dobre praktyki

Źródła: `Get-Help about_Comparison_Operators` · `Get-Help about_Hash_Tables` · `Get-Help about_Arrays` · dokumentacja Microsoft Learn, „about PowerShell"

Revision #36
Created 2026-09-25 06:44:29 UTC by Tomasz Konieczny
Updated 2026-10-18 08:53:13 UTC by Tomasz Konieczny