Befehle

- Start-Sleep
- Join-Path
- Set-Alias

Start-Sleep

Das Cmdlet `Start-Sleep` pausiert die Ausführung eines Skripts oder einer Befehlssequenz für eine definierte Zeitspanne.

Es wird häufig verwendet, um:

- Wartezeiten einzubauen
- Prozesse zu synchronisieren
- Ressourcen zu schonen (z. B. bei Schleifen)

□□ Syntax

```
Start-Sleep [-Seconds] <Double> [<CommonParameters>]


Start-Sleep -Milliseconds <Int32> [<CommonParameters>]
```

□□ Parameter

`-Seconds`

- **Typ:** `Double`
- **Pflicht:** Ja (wenn `-Milliseconds` nicht verwendet wird)
- Gibt die Wartezeit in Sekunden an
- Dezimalwerte sind erlaubt

```
Start-Sleep -Seconds 2.5
```

`-Milliseconds`

- **Typ:** `Int32`
- **Pflicht:** Ja (wenn `-Seconds` nicht verwendet wird)

- Gibt die Wartezeit in Millisekunden an

```
Start-Sleep -Milliseconds 500
```

⚠ Hinweise zur Verwendung

- `-Seconds` und `-Milliseconds` **dürfen nicht gleichzeitig verwendet werden**
- Während der Ausführung blockiert `Start-Sleep` den aktuellen Thread vollständig
- Es erfolgt **keine Hintergrundverarbeitung** während der Pause

---

□□ Verhalten

Eigenschaft	Beschreibung
Blockierend	Ja
Rückgabewert	Keiner
Thread-Verhalten	Aktueller Thread wird pausiert
Genauigkeit	Abhängig vom System-Timer

□□ Beispiele

Einfaches Warten

```
Write-Host "Start"
Start-Sleep -Seconds 2
Write-Host "Ende"
```

Verwendung in Schleifen

```
for ($i = 1; $i -le 5; $i++) {
    Write-Host "Durchlauf $i"
    Start-Sleep -Seconds 1
}
```

Kurze Pause in Millisekunden

```
Start-Sleep -Milliseconds 200
```

Dynamische Wartezeit

```
$delay = 1.5
Start-Sleep -Seconds $delay
```

⚙ Typische Anwendungsfälle

- Polling (z. B. auf Datei oder Prozess warten)
- Debugging (Abläufe verlangsamen)
- UI-Skripte (z. B. kurz warten, bis Controls geladen sind)
- API-Rate-Limiting

□ Alternativen / Ergänzungen

`[System.Threading.Thread]::Sleep()`

```
[System.Threading.Thread]::Sleep(1000)
```

Unterschiede:

- Arbeitet nur mit Millisekunden
- Weniger PowerShell-idiomatisch
- Kein Support für `CommonParameters`

`Start-Sleep` vs. `Start-Job`

Szenario	Empfehlung
Einfaches Warten	`Start-Sleep`
Asynchrone Ausführung	`Start-Job`
UI-Responsiveness wichtig	Kein `Start-Sleep`

□□ Typische Fehler

1. Beide Parameter gleichzeitig verwenden

```
# □ Falsch
Start-Sleep -Seconds 1 -Milliseconds 500
```

---

2. UI einfrieren

```
# □ Problematisch in WinForms/WPF
Start-Sleep -Seconds 5
```

➡ Blockiert die UI komplett

---

3. Zu kurze Wartezeiten erwarten hohe Präzision

```
Start-Sleep -Milliseconds 1
```

➡ Systembedingt oft ungenau

□□ Best Practices

- Für UI-Anwendungen besser Timer verwenden statt `Start-Sleep`
- In Schleifen immer bewusst einsetzen, um CPU-Last zu reduzieren
- Wartezeiten so kurz wie möglich halten
- Für präzise Zeitsteuerung ggf. andere Mechanismen nutzen

Wenn du das Ding in einer WinForms-App benutzt, dann frierst du dir halt elegant dein UI ein und wunderst dich danach, warum alles tot wirkt. Klassischer Anfänger-Move, aber immerhin ein lehrreicher.

Join-Path

Das Cmdlet `Join-Path` kombiniert mehrere Pfadsegmente zu einem gültigen Dateisystempfad.

Es wird verwendet, um:

- Pfade plattformunabhängig zusammenzusetzen
- manuelles Hantieren mit Trennzeichen (`\` oder `/`) zu vermeiden
- Fehler durch doppelte oder fehlende Separatoren zu verhindern

---

□□ Syntax

```
Join-Path [-Path] <String> [-ChildPath] <String> [[-AdditionalChildPath] <String[]>]
[<CommonParameters>]
```

`-Path`

- **Typ:** `String`
- **Pflicht:** Ja
- Basispfad (z. B. Verzeichnis)

`-ChildPath`

- **Typ:** `String`
- **Pflicht:** Ja
- Pfadsegment, das an `-Path` angehängt wird

```
Join-Path -Path "C:\Temp" -ChildPath "Datei.txt"
```

`-AdditionalChildPath`

> *Verfügbar ab:* ***PowerShell 6+***

- **Typ:** `String[]`

- **Pflicht:** Nein
- Weitere Pfadsegmente, die nacheinander angehängt werden

```
Join-Path -Path "C:\Temp" -ChildPath "Logs" -AdditionalChildPath "2026","April"
```

⚠ Hinweise zur Verwendung

- Trennzeichen werden automatisch korrekt gesetzt (kein manuelles `\` nötig)
- Funktioniert providerübergreifend (z. B. Registry, Zertifikate)
- Mehrere Segmente werden sauber zusammengeführt
- Bestehende Trennzeichen im Input werden berücksichtigt (keine doppelten `\\`)

□□ Verhalten

Eigenschaft	Beschreibung
Plattformabhängigkeit	Berücksichtigt das jeweilige Dateisystem
Rückgabewert	`String` (zusammengesetzter Pfad)
Validierung	Keine Existenzprüfung des Pfades
Separator-Handling	Automatisch korrekt

□□ Beispiele

Einfaches Zusammenfügen

```
Join-Path -Path "C:\Temp" -ChildPath "Datei.txt"
```

Ergebnis: `C:\Temp\Datei.txt`

Mehrere Segmente

```
Join-Path -Path "C:\Temp" -ChildPath "Logs" -AdditionalChildPath "2026","April"
```

Ergebnis: `C:\Temp\Logs\2026\April`

Mit Variablen

```
$base = "C:\Temp"
$file = "report.txt"

Join-Path -Path $base -ChildPath $file
```

Ergebnis: `C:\Temp\report.txt`

Provider-unabhängig (z. B. Registry)

```
Join-Path -Path "HKCU:\Software" -ChildPath "Microsoft"
```

⚙ Typische Anwendungsfälle

- Dynamische Dateipfade erstellen
- Arbeiten mit temporären Verzeichnissen
- Plattformunabhängige Skripte schreiben
- Zusammenbau von Registry-Pfaden

Alternativen / Ergänzungen

`String`-Konkatenation

```
# □ Fehleranfällig
"C:\Temp\" + "Datei.txt"
```

- Fehleranfällig bei fehlenden oder doppelten Trennzeichen
- Nicht plattformunabhängig

`[System.IO.Path]::Combine()`

```
[System.IO.Path]::Combine("C:\Temp", "Datei.txt")
```

Unterschiede:

- .NET-Methode, nicht PowerShell-spezifisch
- Keine Provider-Unterstützung (nur Dateisystem)
- Kein Support für `CommonParameters`

□□ Best Practices

- Immer `Join-Path` statt String-Konkatenation verwenden
- Ab PowerShell 6 kann für mehrere Segmente `-AdditionalChildPath` verwendet werden.
- Kombination mit `Test-Path` für Existenzprüfung
- Variablen statt Hardcoding verwenden

Wenn du Pfade immer noch per String zusammenklebst, dann sabotierst du dich halt selbst mit Ansage. Funktioniert kurz, bricht später, und dann suchst du den Fehler wie ein Detektiv ohne Kaffee. Nimm einfach `Join-Path` und erspar dir das Drama.

Set-Alias

Das Cmdlet `Set-Alias` erstellt oder verändert einen Alias für ein Cmdlet, eine Funktion oder einen Befehl.

Ein Alias ist dabei einfach ein alternativer Kurzname für einen bestehenden Command.

Es wird häufig verwendet, um:

- häufig genutzte Befehle schneller aufzurufen
- eigene Kurzbefehle zu definieren
- bestehende Aliase umzubiegen oder anzupassen
- Shell-Umgebungen individueller zu gestalten

□□ Syntax

```
Set-Alias [-Name] <String> [-Value] <String>
          [-Description <String>]
          [-Option <ScopedItemOptions>]
          [-PassThru]
          [-Scope <String>]
          [-Force]
          [-WhatIf]
          [-Confirm]
          [<CommonParameters>]
```

□□ Parameter

`-Name`

- **Typ:** `String`
- **Pflicht:** Ja
- Name des Alias

```
Set-Alias -Name ll -Value Get-ChildItem
```

-Value

- **Typ:** `String`
- **Pflicht:** Ja
- Zielbefehl, auf den der Alias zeigen soll

```
Set-Alias -Name edit -Value notepad
```

-Description

- **Typ:** `String`
- **Pflicht:** Nein
- Fügt dem Alias eine Beschreibung hinzu

```
Set-Alias -Name gs -Value Get-Service -Description "Listet Dienste auf"
```

-Option

- **Typ:** `ScopedItemOptions`
- **Pflicht:** Nein
- Steuert das Verhalten des Alias

Mögliche Optionen:

Option	Bedeutung
`None`	Keine besondere Einschränkung
`ReadOnly`	Alias kann nur mit `-Force` geändert werden
`Constant`	Alias kann gar nicht mehr geändert werden
`Private`	Nur im aktuellen Scope sichtbar

```
Set-Alias -Name test -Value Get-Date -Option ReadOnly
```

-PassThru

- Gibt das erstellte Alias-Objekt zurück

```
Set-Alias -Name now -Value Get-Date -PassThru
```

`-Scope`

- Legt fest, in welchem Scope der Alias existiert

Beispiele:

Scope	Bedeutung
`Local`	Nur aktueller Scope
`Global`	Überall verfügbar
`Script`	Nur innerhalb des Skripts

```
Set-Alias -Name ll -Value Get-ChildItem -Scope Global
```

`-Force`

- Erzwingt das Überschreiben von `ReadOnly`-Aliases

```
Set-Alias -Name ls -Value Get-Process -Force
```

⚠ Hinweise zur Verwendung

- Aliase speichern **keine Parameter**
- Ein Alias ersetzt nur den Befehlsnamen
- Aliase gelten standardmäßig nur für die aktuelle Sitzung
- Für dauerhafte Aliase muss man sie ins Profil (`$PROFILE`) schreiben
- `Constant`-Aliases können nicht mehr entfernt werden

□□ Verhalten

Eigenschaft	Beschreibung
Rückgabewert	Standardmäßig keiner
Überschreibbar	Ja, außer `Constant`
Persistenz	Nur aktuelle Sitzung
Unterstützt Funktionen	Ja
Unterstützt EXE-Dateien	Ja

□□ Beispiele

Einfachen Alias erstellen

```
Set-Alias -Name ll -Value Get-ChildItem
```

Jetzt funktioniert:

```
ll
```

Alias für Programme

```
Set-Alias -Name np -Value notepad
```

Bestehenden Alias überschreiben

```
Set-Alias -Name ls -Value Get-Process -Force
```

Jetzt startet `ls` plötzlich Prozesse statt Dateien aufzulisten.
Ein hervorragender Weg, sich selbst drei Stunden später maximal zu verwirren.

Alias dauerhaft speichern

```
Add-Content -Path $PROFILE -Value 'Set-Alias -Name ll -Value Get-ChildItem'
```

Alias anzeigen

```
Get-Alias ll
```

⚙ Typische Anwendungsfälle

- Eigene Kurzbefehle erstellen
- Linux-ähnliche Befehle nachbauen
- Lange Befehlsnamen abkürzen
- Eigene Shell-Umgebungen konfigurieren
- Schnellzugriffe für Skripte

□ Alternativen / Ergänzungen

`New-Alias`

```
New-Alias -Name ll -Value Get-ChildItem
```

Unterschied zu `Set-Alias`:

Cmdlet	Verhalten
`New-Alias`	Erstellt nur neue Aliase
`Set-Alias`	Erstellt oder überschreibt

Funktionen statt Alias

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

Vorteil:

- Kann Parameter enthalten
- Deutlich flexibler
- Besser für komplexe Logik

□□ Typische Fehler

1. Denken, dass Aliase Parameter speichern

```
# □ Funktioniert NICHT wie erwartet
Set-Alias -Name ll -Value "Get-ChildItem -Force"
```

Ein Alias verweist nur auf einen Commandnamen.
Nicht auf eine komplette Befehlszeile.

➡ Dafür nutzt man Funktionen.

---

2. Alias nach Neustart weg

```
Set-Alias -Name test -Value Get-Date
```

Nach neuer PowerShell-Sitzung verschwunden.

➡ Alias ins `$PROFILE` schreiben.

---

3. Wichtige Standard-Aliase überschreiben

```
Set-Alias -Name cd -Value Get-Date
```

Technisch möglich.
Psychologisch fragwürdig.

□□ Best Practices

- Nur sinnvolle Kurzformen verwenden
- Standard-Aliase nicht leichtfertig überschreiben
- Für komplexe Logik lieber Funktionen nutzen
- Dauerhafte Aliase ins `$PROFILE` auslagern
- In Skripten sparsam mit Aliases umgehen, damit der Code lesbar bleibt