I122

Automatiser des procédures à l'aide de scripts

Module ICT-I122

Introduction

PowerShell est un outil en ligne de commande et un langage de script de Microsoft, conçu pour automatiser des tâches et administrer efficacement des systèmes.

Objectifs

À la fin de ce chapitre, vous serez capable de :

- Expliquer l'utilité de PowerShell et son historique.
- Identifier les principales versions et leurs différences.
- Préparer un environnement de travail (VM, policy d'exécution, éditeur).

Historique

- Avant PowerShell, l'automatisation sous Windows reposait surtout sur les fichiers **Batch** (`.bat`, `.cmd`) exécutés dans `command.com` sous MS-DOS puis `cmd.exe` sous Windows.
- Ces scripts permettaient d'enchaîner des commandes, mais restaient plus limités que les shells Unix/Linux comme **Bash**.
- En **2002–2003**, Microsoft lance le projet **Monad**, qui deviendra plus tard **Windows PowerShell** : un shell et un langage de script orientés administration système.
- En **2006**, **PowerShell 1.0** est publié officiellement. Il introduit une approche plus moderne que les fichiers Batch, avec des **cmdlets**, des **objets** et un pipeline orienté objets.
- En **2009**, **Windows 7** intègre nativement **Windows PowerShell 2.0**, qui apporte notamment l'exécution distante, les jobs en arrière-plan et de nombreuses améliorations pour l'administration.
- Par la suite, PowerShell évolue jusqu'à **Windows PowerShell 5.1**, qui reste la version Windows historique encore présente par défaut sur de nombreux systèmes Windows.
- À partir de **PowerShell 6**, le projet devient **open source**, basé sur **.NET Core**, et devient **multiplateforme** : Windows, macOS et Linux.

- Aujourd'hui, la branche moderne est **PowerShell 7**, installée séparément de Windows PowerShell 5.1 (`powershell.exe`) et conçue pour coexister avec elle. La documentation Microsoft référence actuellement **PowerShell 7.5** (`pwsh.exe`) comme version stable récente.

Utilité

Automatisation de tâches et procédures en utilisant des scripts.

- Gain de temps
- Minimisation des erreurs

Exemples d'utilisation :

- Création/suppression de comptes utilisateurs
- Modification et copie de fichiers d'un serveur à un autre pour les intégrer dans une GED
- Surveillance de services, d'espaces disques, process,... avec envoi d'emails/alertes
- Configuration d'un serveur sans interface graphique
- Collecte de données ou administration sur des machines physiques distantes
- Installation de systèmes ou programmes

Définition et informations

Interface développée par Microsoft, remplaçant l'invite de commandes et ajoutant des fonctionnalités permettant d'interagir avec le système et de scripter :

- Langage interprété et non typé
- Outil en lignes de commandes similaire à *Linux*
- Héritage du **Framework.Net**, version 2.0 requise au minimum
- Orienté objets

Versions natives

- Windows 7 et WS 2008 R2 ⇒ PowerShell 2.0

- Windows 8 et WS 2012 ⇒ PowerShell 3.0
- Windows 8.1 et WS 2012 R2 ⇒ PowerShell 4.0
- Windows 10 (version initiale) ⇒ PowerShell 5.0
- Windows 10 Anniversary Update, WS 2016, Windows 11 et WS 2022 ⇒ PowerShell 5.1

> **Info**
>
> PowerShell 7 n'est pas natif, il s'installe séparément et coexiste avec Windows PowerShell 5.1.

L'on peut trouver la version installée avec `$PSVersionTable.PSVersion`.

Sécurité d'exécution

Par défaut, **l'exécution de scripts PowerShell n'est pas autorisée sur les ordinateurs clients Windows**. La stratégie d'exécution par défaut est généralement **`Restricted`**, ce qui empêche le lancement des scripts `.ps1`. Sur **Windows Server**, la stratégie par défaut est en général **`RemoteSigned`**.

Les stratégies d'exécution ont pour but de **réduire les risques d'exécution accidentelle de scripts malveillants**. Elles ne constituent toutefois **pas une protection de sécurité complète**, mais plutôt un mécanisme de prévention et d'encadrement.

Parmi les stratégies les plus connues :

Stratégie	Description
`Restricted`	aucun script n'est exécuté
`RemoteSigned`	les scripts locaux peuvent être exécutés, mais les scripts téléchargés depuis Internet doivent être signés
`AllSigned`	tous les scripts doivent être signés par un éditeur approuvé
`Bypass`	aucune restriction ni avertissement
`Unrestricted`	les scripts peuvent s'exécuter, avec avertissement pour certains scripts téléchargés

Pour connaître la stratégie actuellement appliquée, on peut utiliser :

```powershell
Get-ExecutionPolicy
```

Pour afficher le détail selon les différentes portées (`Process`, `CurrentUser`, `LocalMachine`, etc.) :

```powershell
Get-ExecutionPolicy -List
```

Pour autoriser plus facilement l'exécution de scripts sur un poste de travail, on utilise souvent :

```powershell
Set-ExecutionPolicy RemoteSigned
```

Cette commande modifie la stratégie d'exécution sur Windows. Selon la portée choisie, elle peut s'appliquer uniquement à l'utilisateur courant ou à l'ensemble de la machine.

Installation et préparation de la VM

Dans ce cours vous utiliser une machine virtuelle sous VirtualBox afin de pouvoir bénéficier des droits d'administration.

- 📥 Installation VBox-Win10 (PDF)
- 📥 Installation VBox-Win11 (PDF)

> **Info**
>
> Sélectionnez la documentation ci-dessus selon l'OS votre machine.

Windows offre 2 interfaces différentes, en 32 et 64 bits :

- **CLI (Command Line Interface)** est la console en lignes de commande.
- **ISE (Integrated Scripting Environment)** permet l'édition de scripts et offre la mise en couleur pour les différentes parties de la commande et les listes déroulantes.

Suite à l'abandon de la mise à jour de ISE par MS, pour coder, il est préférable d'utiliser VS Code avec l'extension PowerShell.

Configuration de Visual Studio Code pour PowerShell :

- 📥 Configuration de VS pour PS (PDF)

> **Info**
>
> La mise à jour de l'aide intégrée de PowerShell se fait avec **Update-Help**.

Résumé

- PowerShell est un outil d'automatisation orienté objets.
- Windows PowerShell 5.1 est encore présent nativement sur Windows, mais PowerShell 7 constitue la version moderne et multiplateforme.
- L'environnement de travail (VM, policy d'exécution, éditeur) conditionne l'exécution et le développement des scripts.
- VS Code avec l'extension PowerShell est aujourd'hui préférable à ISE pour écrire des scripts.

Commandes

Powershell a uniformisé les commandes en les définissant avec un verbe, qui donne l'action, suivi d'un tiret, puis d'un nom : verbe-nom.

Objectifs

À la fin de ce chapitre, vous serez capable de :

- Reconnaître la convention **Verbe-Nom** des cmdlets.
- Utiliser **Get-Help** et **Get-Command** pour explorer les commandes et leurs paramètres.
- Chaîner des commandes avec le **pipe** (|).

Les Cmdlet

On les nomme Cmdlet (prononcée command-let) et l'écriture française commandelette :

- `Get-Service` ⇒ obtenir les services
- `Get-Process` ⇒ obtenir les processus
- `Get-Help` ⇒ obtenir l'aide
- `Get-Command` ⇒ obtenir les commandes
- `Stop-Service` ⇒ arrêter les services
- `Stop-Process` ⇒ arrêter les processus
- `New-Item` ⇒ Créer un objet
- `Set-ExecutionPolicy` ⇒ Définir la politique de sécurité
- `Clear-Host` ⇒ Efface le contenu de la console

Les verbes sont toujours les mêmes dans le sens où lorsque l'on veut obtenir une information c'est toujours le verbe **Get** ou **Stop**, si l'on veut arrêter.

C'est le même principe pour le nom, c'est toujours **Service** ou **Process**, ce n'est pas au pluriel ou en français!

Pour connaître tous les verbes utilisé (ou à utiliser), faites :

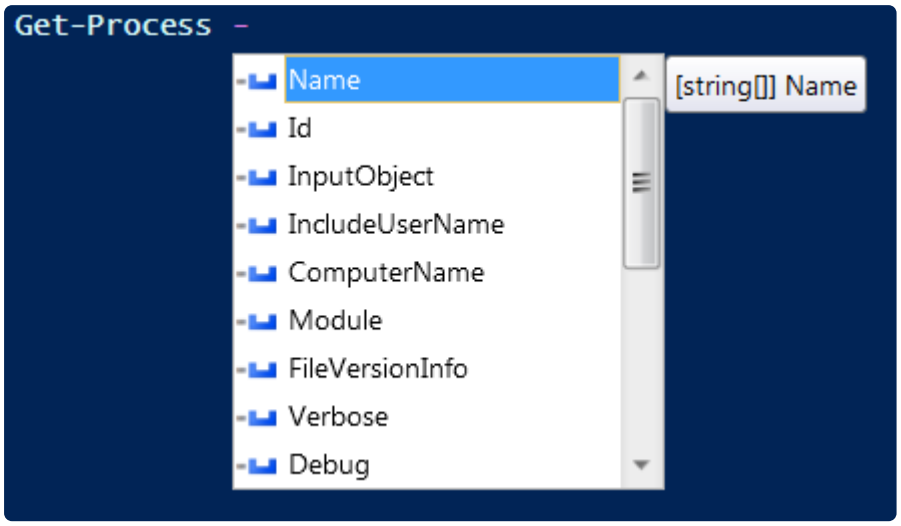
```powershell
Get-Verb | Sort-Object Verb
```

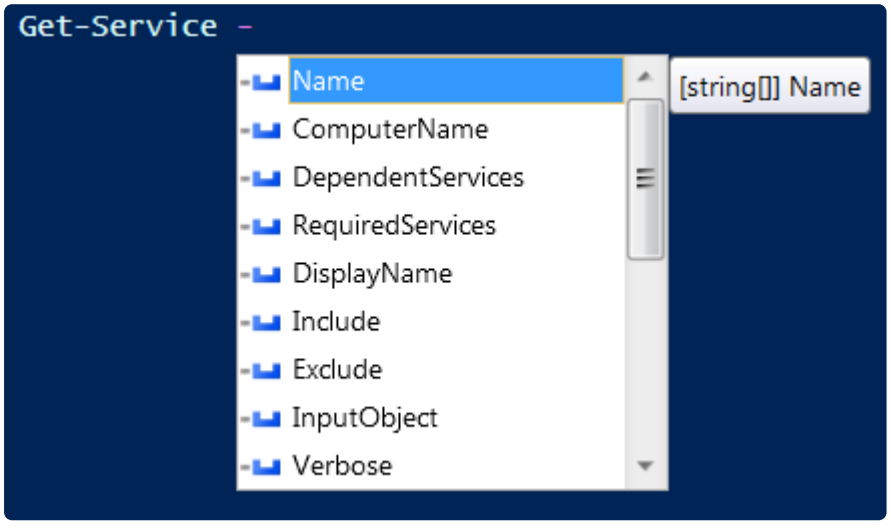
Elle affiche la liste des verbes approuvés à utiliser pour nommer correctement les fonctions et cmdlets PowerShell.

Des paramètres sont définis pour toutes les commandes : `-Confirm`, `-Debug`, `-Whatif`, `-ErrorAction`, `-Verbose`, `-Force`, etc...

Pour chaque cmdlet, il y a une liste de paramètres.

Exemple avec `Get-Process` et `Get-Service` :





Par exemple, obtenir les services qui ont comme première lettre **w** :

```powershell
Get-Service -Name w*
```

Ou bien, arrêter le service *Client Impero*, avec une confirmation :

```powershell
Stop-Process -Name ImperoClientSVC -Confirm
```

Un Cmdlet (**CommandType**) peut être de plusieurs type :

- **Cmdlet**
- **Function**
- **Alias**
- **Application**
- **Script**

Les Cmdlet sont écrites dans un langage de programmation .Net et sont compilées.

- `Remove-Item`
- `Start-Service`

Une Function est écrite en Powershell et n'est pas compilée. Il est possible d'ajouter ses propres fonctions dans celles du système.

- **ImportSystemModule**
- **C:**
- **Clear-Host**

Un alias est un raccourci, example :

- **ls** ⇒ `Get-ChildItem`

Il y a 3 commandes de bases à connaître.

```powershell
Get-Help
```

Donne les informations sur une commande ainsi que la syntaxe.

C'est identique à `Get-Service -?`

```powershell
Get-Help Get-Service
```

Get-Command

`Get-Command` permet de lister les commandes disponibles.

Par exemple, liste les commandes qui contiennent le nom service :

```powershell
Get-Command *service*
```

On fait la différence entre le paramètre `-Noun` et le `-Name` :

```
PS C:\Users\admin> Get-Command -Noun process

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Cmdlet          Debug-Process                                      3.1.0.0    Microsoft.PowerShe...
Cmdlet          Get-Process                                        3.1.0.0    Microsoft.PowerShe...
Cmdlet          Start-Process                                      3.1.0.0    Microsoft.PowerShe...
Cmdlet          Stop-Process                                       3.1.0.0    Microsoft.PowerShe...
Cmdlet          Wait-Process                                       3.1.0.0    Microsoft.PowerShe...


PS C:\Users\admin> Get-Command -Name *process*

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Function        Get-AppvVirtualProcess                             1.0.0.0    AppvClient
Function        Start-AppvVirtualProcess                           1.0.0.0    AppvClient
Cmdlet          Debug-Process                                      3.1.0.0    Microsoft.PowerShe...
Cmdlet          Enter-PSHostProcess                                3.0.0.0    Microsoft.PowerShe...
Cmdlet          Exit-PSHostProcess                                 3.0.0.0    Microsoft.PowerShe...
Cmdlet          Get-Process                                        3.1.0.0    Microsoft.PowerShe...
Cmdlet          Get-PSHostProcessInfo                              3.0.0.0    Microsoft.PowerShe...
Cmdlet          Start-Process                                      3.1.0.0    Microsoft.PowerShe...
Cmdlet          Stop-Process                                       3.1.0.0    Microsoft.PowerShe...
Cmdlet          Wait-Process                                       3.1.0.0    Microsoft.PowerShe...
Application     qprocess.exe                                       10.0.14... C:\Windows\system3...
```

Dans le cas de `-Noun`, la recherche sur la partie nom de la cmdlet. De la même façon on peut obtenir les commandes avec un verbe défini **-Verb** :

```powershell
Get-Command -Verb Write
```

Get-Member

Donne le type, les propriétés et les méthodes pour un objet Il y a cependant une particularité pour cette commande. Elle a besoin d'être appliquée sur un objet... donc on utilise le **pipe** ou | (AltGr + 7).

Par exemple, on a pour la cmdlet `Get-Service` :

```powershell
Get-Service | Get-Member
```

Pipe

Le **pipe** (tube en français) permet de transmettre le résultat d'une commande à une autre commande.

On obtient les services qui commence par la lettre w, affichés sous format d'une table à 2 colonnes avec le nom et l'état associé :

```powershell
Get-Service w* | Format-Table Name, Status
```

Un autre exemple est celui d'un tri sur une arborescence :

```
PS C:\Windows\system32> Get-ChildItem C:\test\test1


    Répertoire : C:\test\test1


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        26.01.2019     16:06                fr-FR
d-----        10.07.2015     10:28                Icons
d-----        26.01.2019     16:06                Skins
d-----        10.07.2015     10:28                Visualizations
-a----        10.07.2015     10:25         167936 mpvis.DLL
-a----        25.11.2018     14:21             12 toto.txt
-a----        10.07.2015     10:25         107008 WMPMediaSharing.dll
-a----        10.07.2015     10:25         507904 wmpnssci.dll
-a----        10.07.2015     10:25          18432 WMPNSSUI.dll


PS C:\Windows\system32> Get-ChildItem C:\test\test1 | sort


    Répertoire : C:\test\test1


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        26.01.2019     16:06                fr-FR
d-----        10.07.2015     10:28                Icons
-a----        10.07.2015     10:25         167936 mpvis.DLL
d-----        26.01.2019     16:06                Skins
-a----        25.11.2018     14:21             12 toto.txt
d-----        10.07.2015     10:28                Visualizations
-a----        10.07.2015     10:25         107008 WMPMediaSharing.dll
-a----        10.07.2015     10:25         507904 wmpnssci.dll
-a----        10.07.2015     10:25          18432 WMPNSSUI.dll
```

La commande Get-History permet de lister les commandes de la session active.

Résumé

- Les **Cmdlets** suivent une convention de nommage **Verbe-Nom**.
- **Get-Help** et **Get-Command** sont les outils clés pour apprendre une commande.
- Le **pipe** permet de transmettre des objets d'une commande à une autre.

Affichage

Il faut distinguer le formatage de l'affichage et les modes d'affichage.

Objectifs

À la fin de ce chapitre, vous serez capable de :

- Choisir le bon format d'affichage (**Format-List/Wide/Table**).
- Comprendre la différence entre *formatage* et *sortie*.
- Utiliser les cmdlets **Write-*** pour informer, diagnostiquer et signaler.

Le formatage va définir la façon dont l'information sera "mise en page", alors que le mode d'affichage définira où et quels types d'informations seront traitées.

Formatage

Il y a 4 formatages possibles :

- `Format-List` ⇒ liste
- `Format-Wide` ⇒ colonnes sur la largeur de page
- `Format-Table` ⇒ tableau
- `Format-Custom` ⇒ personnalisé

Liste

```powershell
Get-Service u* | Format-List DisplayName, Status
```

```
PS C:\Users\isastucki> Get-service u* | Format-List DisplayName, Status


DisplayName : Détection de services interactifs
Status      : Running

DisplayName : Redirecteur de port du mode utilisateur des services Bureau à distance
Status      : Running

DisplayName : Hôte de périphérique UPnP
Status      : Stopped

DisplayName : Gestionnaire de sessions du Gestionnaire de fenêtrage
Status      : Running
```

Tableau

```powershell
Get-Service u* | Format-Table DisplayName, Status
```

```
PS C:\Users\isastucki> Get-Service u* | Format-Table DisplayName, Status

DisplayName                                                          Status
-----------                                                          ------
Détection de services interactifs                                    Running
Redirecteur de port du mode utilisateur des services Bureau à distance Running
Hôte de périphérique UPnP                                            Stopped
Gestionnaire de sessions du Gestionnaire de fenêtrage                Running
```

Largeur de page, `-AutoSize` adapte le nombre de colonnes en fonction de l'espace

```powershell
Get-Service u* | Format-Wide DisplayName -Column 2
```

```
PS C:\Users\isastucki> Get-Service u* | Format-Wide DisplayName -Column 2


Détection de services interactifs                 Redirecteur de port du mode utilisateur des services B...
Hôte de périphérique UPnP                         Gestionnaire de sessions du Gestionnaire de fenêtrage
```

Modes

`Write-Host`

Affiche uniquement sur la console :

```powershell
Write-Host "bonjour toto"
```

`Write-Output`

Sortie avec redirection possible :

```powershell
Write-Output "bonjour toto"
```

Ici, les 2 **Write** donneront le même résultat :

Si on utilise la redirection dans un fichier. Le fichier *titi.txt* sera vide, car c'est un affichage sur l'écran seulement.

```powershell
Write-Host "bonjour titi" > d:\temp\titi.txt
```

Le fichier *tutu.txt* contiendra le texte, car cette Cmdlet permet la redirection dans un fichier :

```powershell
Write-Output "bonjour tutu" > d:\temp\tutu.txt
```

Si les informations que l'on veut ajouter dans le fichier sont seulement les erreurs ("canal" 1 correspond à standard console et 2 à standard erreur).

Le fichier *erreurs.txt* contiendra l'information du message d'erreur, car il n'y a pas de service qui s'appelle *web* :

```powershell
Get-Service web 2> d:\log\erreurs.txt
```

Le fichier sera vide car il trouve un service *WebClient* :

```powershell
Get-Service web* 2> d:\log\erreurs.txt
```

`Write-Verbose`

Affiche un message complémentaire en bleu par exemple pour information dans l'execution d'un script pas de redirection et le début de ligne du message commence par *COMMENTAIRES :*

```powershell
Write-Verbose Hello -Verbose
```

`Write-Debug`

Débogage d'un script, pas de redirection, commence par *DEBOGUER :*

```powershell
Write-Debug "point de debug" -debug
```

`Write-Warning`

Affiche un message d'avertissement en orange, pas de redirection et commence par *AVERTISSEMENT :*

```powershell
Write-Warning "hello"
```

`Write-Error`

Affiche un message en rouge, redirection seulement pour les erreurs :

```powershell
Write-Error "Erreur"
```

Résumé

- Le formatage concerne la présentation, pas les données.
- La sortie peut être orientée écran/fichier selon le besoin.
- Les cmdlets **Write-*** aident à structurer l'exécution et les messages.

Scripts de base

Un script est un ensemble de commandes permettant d'effectuer une tâche.

Objectifs

À la fin de ce chapitre, vous serez capable de :

- Comprendre la structure d'un script (en-tête, paramètres, fonctions, corps).
- Adapter la **stratégie d'exécution** et lancer un script de manière correcte.
- Appliquer des normes de codage (noms, variables, paramètres) et utiliser le débogage de base.

Script .ps1

Le fichier PowerShell aura une extension **.ps1** et un double-clique provoque l'ouverture du fichier avec **notepad** et non pas son exécution.

> **Info**
>
> Notepad ne supportant pas l'encodage UTF-8, préférer l'utilisation de Notepad++, ou encore mieux de VS Code.

L'execution

Pour exécuter un script PowerShell, on utilise son nom précédé de `.\` si l'on se trouve dans le dossier courant.

La commande la plus simple est :

```powershell
.\MonScript.ps1
```

Avec chemin complet :

```powershell
C:\Scripts\MonScript.ps1
```

Et si ton script attend des paramètres :

```powershell
.\MonScript.ps1 -Nom "John" -Age 25
```

L'on peut aussi l'exécuter le script depuis ISE, depuis une boite de commande ou en utilisant le planificateur de tâches. Dans certains cas, les droits administrateurs sont requis.

Info

Par défaut, la stratégie de sécurité de Microsoft bloque l'exécution des scripts (**Restricted**).

Info

On active l'exécution des scripts sur une machine avec la commande `Set-ExecutionPolicy` et il est nécessaire d'avoir les droits administrateur. La stratégie **Unrestricted** s'applique dans notre cas, car nous sommes dans une école. Ce ne serait pas prudent dans une entreprise.

La structure

Un script est constitué de plusieurs zones d'un script sont :

- l'en-tête;
- la zone pour les paramètres et les variables;
- les fonctions;
- et enfin le corps du script.

```
<#
    Zone d'en-tête, qui est aussi utilisée pour le Help du script
#>

<#
    Paramètres et variables
#>

<#
    Fonctions
#>

<#
    Corps du script
#>
```

L'en-tête

Pour les commentaires, on utilise le `#`, pour une seule ligne :

```powershell
# commentaire
Write-Host "HELLO"
```

Et `<#...#>` pour plusieurs lignes :

```powershell
<# commentaires
    commentaires
    commentaires
#>
```

Dans le canevas fourni dans la première page du cours, l'en-tête du fichier est composée de plusieurs parties.

L'aide de PowerShell utilise ces différentes parties : **.NOTES**, **.SYNOPSIS**, **.DESCRIPTION**,...

Isabelle-Give-Name.ps1

```
<#
.NOTES
    ********************************************************************************
    ETML
    Nom du script: Isabelle-Give-Name.ps1
    Auteur: Isabelle Stucki
    Date:   juillet 2023
    ********************************************************************************
    Modifications
    Date  : -
    Auteur: -
    Raisons: -
    ********************************************************************************
.SYNOPSIS
    Ecrit un message à l'écran

.DESCRIPTION
    Affiche sur la console un message de bienvenue à une personne définie par son prénom et son nom

.PARAMETER Firstname
    Le prénom de la personne

.PARAMETER Name
    Le nom de la personne

.EXAMPLE
    .\Isabelle-Give-Name.ps1 -Firstname Isabelle -Name Stucki
    Résultat : Bonjour Isabelle Stucki

.EXAMPLE
    .\Isabelle-Give-Name.ps1
    Résultat : Sans paramètre, affichage de l'aide

#>
```

Chaque partie correspond à une rubrique de l'aide. Il est impératif de respecter le point et les majuscules.

Pour la totalité des possibilités pour l'aide, voir sur le site de Microsoft : Powershell help ↗.

La partie du code qui affiche l'aide utilise `$MyInvocation.MyCommand.Path` pour obtenir le chemin complet du script en cours d'exécution, puis le passe à `Get-Help` :

```powershell
Get-Help $MyInvocation.MyCommand.Path
```

`$MyInvocation` est une variable automatique qui contient des métadonnées sur l'exécution en cours (chemin du script, numéro de ligne, etc.). C'est bien `Get-Help` qui interprète et affiche l'aide commentée.

Lorsque l'aide s'affiche, le résultat donne :

```
    PS C:\>.\Isabelle-Give-Name.ps1

    Résultat : Sans paramètre, affichage de l'aide



LIENS CONNEXES



PS C:\Users\ISI> Get-Help S:\Isabelle-Give-Name.ps1

NOM
    S:\Isabelle-Give-Name.ps1

RÉSUMÉ
    Ecrit un message à l'écran


SYNTAXE
    S:\Isabelle-Give-Name.ps1 [[-Firstname] <String>] [[-Name] <String>] [<CommonParameters>]


DESCRIPTION
    Affiche sur la console un message de bienvenue à une personne définie par son prénom et son nom


LIENS CONNEXES

REMARQUES
    Pour consulter les exemples, tapez : "get-help S:\Isabelle-Give-Name.ps1 -examples".
    Pour plus d'informations, tapez : "get-help S:\Isabelle-Give-Name.ps1 -detailed".
    Pour obtenir des informations techniques, tapez : "get-help S:\Isabelle-Give-Name.ps1 -full".
```

Les parties **.NOTES**, **.SYNOPSIS**, **.DESCRIPTION** et **.EXEMPLE** sont toujours présentes et les autres sont ajoutées si besoin.

Possible seulement 1 fois : **.NOTES**, **.SYNOPSIS**, **.DESCRIPTION**, **.OUTPUTS**

Possible plusieurs fois : **.PARAMETER**, **.EXEMPLE**, **.LINK**

Élément	Description
`.NOTES`	informations sur l'auteur, date
`.SYNOPSIS`	max 1 ligne, comme un titre
`.DESCRIPTION`	explications complètes sur ce que fait le script, ce qui se passe en cas d'erreur, le résultat attendu
`.PARAMETER`	le nom du paramètre, ce qu'il fait, son type, les contraintes,... et un seul à la fois
`.OUTPUTS`	ce qui est généré, fichier d'erreurs, création d'utilisateur(s), modification d'une clé,...
`.EXEMPLE`	ligne pour lancer le script et aussi un seul à la fois et bien si min 2 exemples
`.LINK`	lien sur un site ou prérequis comme nom d'un ou plusieurs script(s) (par exemple : utilisation de 2 scripts Create-User et Create-Group dans un script Add-UserGroup)

Info

A noter que l'on peut aussi utiliser **.INTPUTS** pour défini ce qui peut provenir du pipe.

Normes de codage

On respecte les normes de codage définies dans l'école.

- Pour le nom des scripts, on utilise **Firstname-Verb-Noun**, avec des majuscules pour la première lettre du verbe et du nom.
- On ajoute le prénom devant le nom du script, pour que l'enseignant puisse différencier les scripts lors de contrôles !!!
- Pour les noms des paramètres (après "param") et des fonctions dans le script, on utilise *UpperCamelCase*.
- Pour les variables, on utilise *lowerCamelCase*.
- La notion de **guard clauses** permet l'optimisation du code en évitant de le parcourir entièrement en cas d'informations fausses ou manquantes.
- Il faut éviter d'utiliser la partie *else* et plutôt ajouter un *exit* dans la partie *if* .

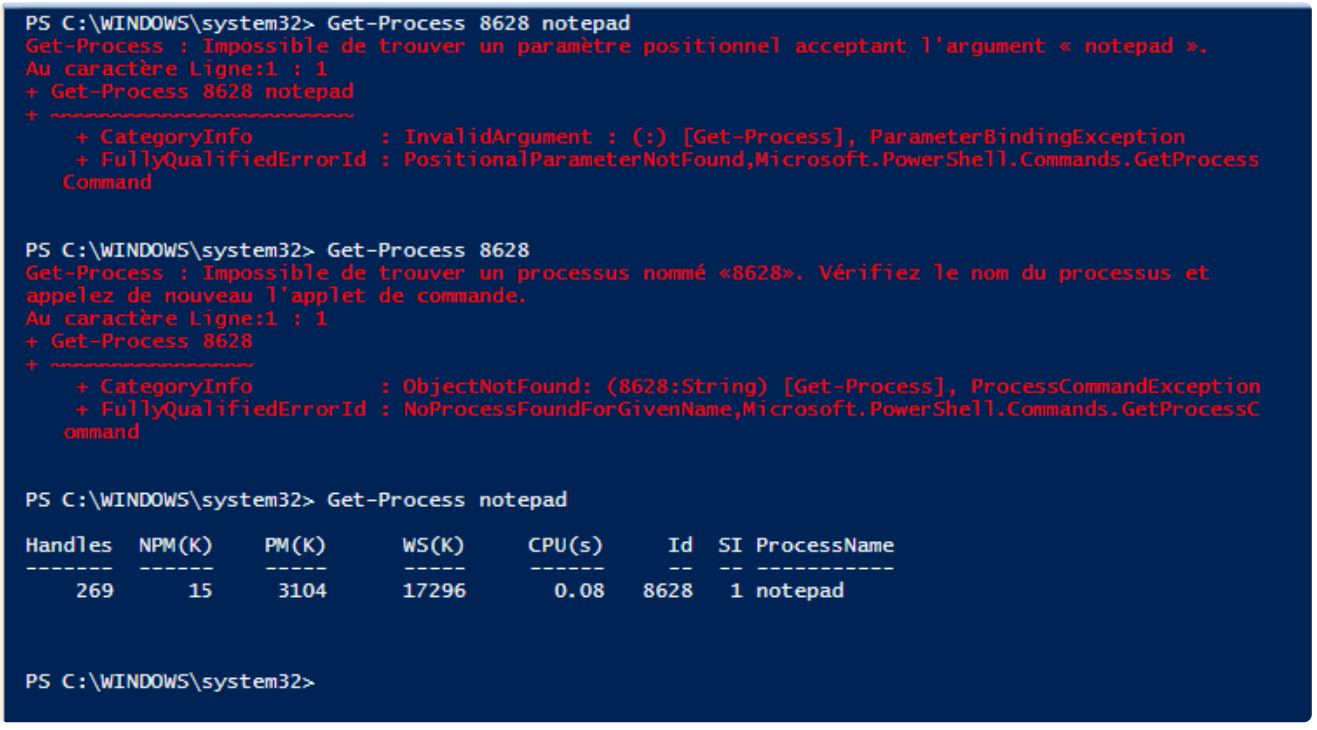
Exemple de test vérification des droits administrateur :

```powershell
# Vérification des droits administrateur
$estAdmin = ([Security.Principal.WindowsPrincipal] `
    [Security.Principal.WindowsIdentity]::GetCurrent()
).IsInRole([Security.Principal.WindowsBuiltInRole]::Administrator)

if (-not $estAdmin) {
    Write-Error "Ce script doit être exécuté avec des droits administrateur
    exit 1
}

Write-Host "Le script est bien exécuté en tant qu'administrateur."
```

Les paramètres

Dans le script, la première ligne de code doit être **param**

Dans le cas d'un script, les paramètres non explicitement sélectionnés seront pris dans l'ordre, mais dans le cas d'une cmdlet, il ne tient compte que du premier de la liste.



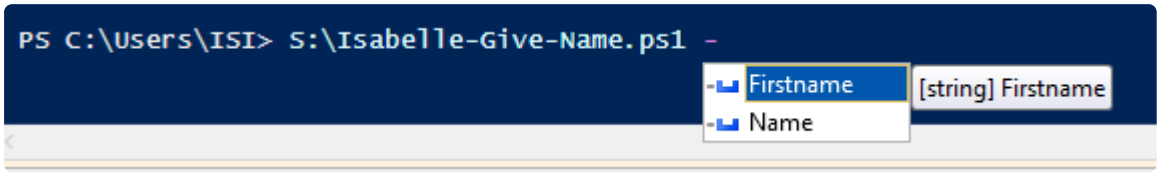
Utilisation

Si le script est exécuté avec les paramètres.

```
35
36  # La définition des paramètres $Firstname et $Name se trouve dans l'en-tête
37  param([string]$Firstname, [string]$Name)
38
39  ##############################################################################################
40  # Zone de définition des variables et fonctions, avec exemples
41  # Commentaires pour les variables
42
43
44  ##############################################################################################
45  # Zone de tests comme les paramètres renseignés ou les droits administrateurs
46
47  # Affiche l'aide si un ou plusieurs paramètres ne sont par renseignés, "safe guard clauses"
48  if(!$Firstname -or !$Name)
49  {
50      Get-Help $MyInvocation.Mycommand.Path
51      exit
52  }
53
54  ##############################################################################################
55  # Corps du script
56
57  # Ce que fait le script, ici, afficher un message
58  Write-Host "Bonjour" $Firstname $Name
59
```

```
PS C:\>.\Isabelle-Give-Name.ps1 -Firstname Isabelle -Name Stucki

Résultat : Bonjour Isabelle Stucki
```

Les paramètres d'un script que l'on a codé auront le même comportement, à savoir que lorsque l'on tape le **-** la liste des paramètres s'affiche dans un menu déroulant.



Dans l'aide, la liste des paramètres sera définie par **.PARAMETER**.

```
19
20  .PARAMETER Firstname
21      Le prénom de la personne
22
23  .PARAMETER Name
24      Le nom de la personne
25
```

Concernant le menu des paramètres, il est établi lorsque l'on définit les variables du script.

```
35
36  # La définition des paramètres $Firstname et $Name se trouve dans l'en-tête
37  param([string]$Firstname, [string]$Name)
38
```

Le corps du script se situe après la définition des paramètres et du test de la présence du(es) paramètre(s)

```powershell

##################################################################################################
# Zone de tests comme les paramètres renseignés ou les droits administrateurs

# Affiche l'aide si un ou plusieurs paramètres ne sont par renseignés, "safe guard clauses"
if(!$Firstname -or !$Name)
{
    Get-Help $MyInvocation.Mycommand.Path
    exit
}

##################################################################################################
# Corps du script

# Ce que fait le script, ici, afficher un message
Write-Host "Bonjour" $Firstname $Name

```

Ces attributs

On peut donner des caractéristiques particulières à un paramètre, dans le même esprit que pour typer une variable !

```powershell
param(
    [Parameter(Mandatory=$True, Position=0, ValueFromPipeline=$True)][strin
    $NotRequiredParam
)
```

Ci-dessus 2 paramètres sont déclarés `$RequiredParam` et `$NotRequiredParam`.

- Ici, seul `$RequiredParam` possède des attributs de configuration avancés.
- `$NotRequiredParam` : déclare un second paramètre non typé et non obligatoire.

Description de la configuration avancée :

Élément	Explication
`param(...)`	définit les paramètres du script ou de la fonction
`[Parameter(...)]`	ajoute des options de configuration au paramètre qui suit
`Mandatory=$True`	rend ce paramètre obligatoire
`Position=0`	permet de passer ce paramètre en premier, sans devoir écrire son nom
`ValueFromPipeline=$True`	autorise la réception de la valeur depuis le pipeline

Élément	Explication
`[string]$RequiredParam`	déclare un paramètre nommé `RequiredParam` de type **chaîne de caractères**

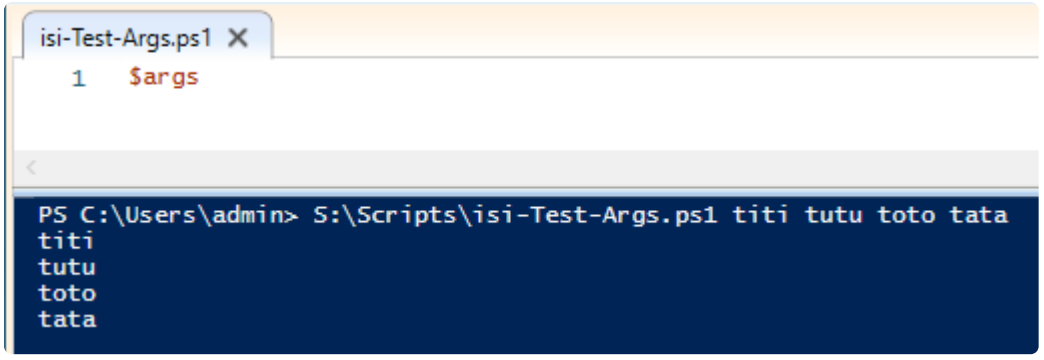
Les arguments

Les arguments seront mis "de manière libre" après le nom du script.

Dans la liste des variables automatiques, il y a `$Args` et c'est un tableau des arguments passés lors de l'exécution d'un script.

Ce tableau peut contenir plusieurs sortes d'éléments et sa longueur s'obtient avec `$Args.length`.

Un script qui ne contient que `$args`, affichera le contenu du tableau des arguments fournis lors du lancement du script.



Exemple de test des paramètres :

```powershell
# Test des paramètres du script
param(
    [string]$Nom,
    [int]$Age
)

if ([string]::IsNullOrWhiteSpace($Nom)) {
    Write-Error "Le paramètre -Nom est obligatoire."
    exit 1
}
```

```powershell
if ($Age -lt 0) {
    Write-Error "Le paramètre -Age doit être supérieur ou égal à 0."
    exit 1
}

Write-Host "Paramètres valides : Nom=$Nom, Age=$Age"
```

Debug

Dans ISE, il y a un menu Déboguer.

Dans ISE et dans VS Code, les raccourcis sont :

- **F5** : exécuter
- **F9** : ajouter un point de debug
- **F10** : pas à pas (step over)
- **F11** : pas à pas détaillé (step into)
- On peut voir le contenu d'une variable en passant dessus lentement avec le curseur.

Avec VS Code, il faut ajouter un fichier texte et le sauvegarder en choisissant le type *Powershell*, et surtout avec l'extension *.ps1*.

Résumé

- Un script PowerShell est structuré et documenté via l'en-tête (comment-based help).
- Les paramètres et leurs attributs rendent les scripts plus robustes et réutilisables.
- Les normes de codage et le débogage facilitent la maintenance et le diagnostic.

Répertoires et fichiers

PowerShell permet de gérer facilement les répertoires et les fichiers, que ce soit pour les parcourir, les créer, les copier, les déplacer ou les supprimer.

Objectifs

À la fin de ce chapitre, vous serez capable de :

- Manipuler les fichiers et dossiers (lister, créer, copier, déplacer, renommer, supprimer).
- Naviguer dans le système de fichiers avec **Get-Location** / **Set-Location**.
- Comprendre les bases de la **registry** et lire des propriétés avec les cmdlets associées.

Get-ChildItem

Permet de lister le contenu d'un répertoire. Exemple :

```powershell
Get-ChildItem D:\test
Get-ChildItem D:\test -Force
Get-ChildItem D:\test -Include *.txt -Recurse
```

Paramètres : `-Force` permet d'afficher les fichiers cachés et `-Recurse` le contenu de l'arborescence.

Get-Location

Retourne l'emplacement actuel en chemin absolu :

```powershell
Get-Location
$path = Get-Location
```

```
$path | Get-Member
$path = (Get-Location).path
```

La variable `$path` contiendra le chemin absolu du répertoire courant.

Set-Location

Permet de se déplacer dans une arborescence ou dans un autre disque :

```powershell
Set-Location C:
Set-Location D:\temp
Set-Location ..
Set-Location \
```

> **Info**
>
> Pour rappel `..` se réfère au répertoire parent et le `\` à la racine du disque courant.

New-Item

Permet la création de dossiers ou de fichiers. Le paramètre `-ItemType` définit `File` pour un fichier et **Directory** pour un répertoire.

```powershell
New-Item -ItemType Directory -Name titi -Path d:\toto
New-Item -ItemType File -Name tutu.txt -Path d:\toto
```

Remove-Item

Permet de supprimer des fichiers ou dossiers.

```powershell
Remove-Item -Path d:\toto\tutu.txt
```

```
Get-ChildItem d:\toto\tutu.txt -Include *.txt | Remove-Item
```

Paramètres : `-Force` pour les fichiers cachés, `-Whatif` permet de simuler le résultat et `-Confirm` demander confirmation.

Move-Item

Permet le déplacement d'un fichier ou répertoire d'une source à une destination.

```powershell
Move-Item -Path d:\temp\toto -Destination d:\temp\titi
Move-Item -Path d:\temp\toto -Destination d:\machin -Force
Move-Item -Path d:\temp\*.ini -Destination d:\temp\ini
```

Pour le dernier exemple le répertoire *ini* doit déjà exister et pour le 2e, c'est le *disque d:* qui doit exister. Le `-Force` permet la création du répertoire si besoin.

Rename-Item

Permet de renommer un dossier ou un fichier.

```powershell
Rename-Item -Path C:\temp\toto titi
Rename-Item -Path C:\temp\toto C:\temp\titi
Rename-Item -Path C:\temp\doc1.txt doc2.txt
```

Copy-Item

Permet de copier des fichiers ou des répertoires.

```powershell
Copy-Item -Path d:\temp\doc1.txt -Destination d:\docs
Copy-Item -Path d:\temp -Destination d:\backups -Recurse
```

Info

Paramètre `-Recurse` pour une arborescence, afin de copier aussi le contenu des répertoires et sous-répertoire se trouvant dans `d:\temp`.

Invoke-Item

Permet d'ouvrir un fichier avec l'exécution associée par défaut.

```powershell
Invoke-Item -Path d:\temp\planning.docx
```

Le fichier *planning.docx* sera ouvert avec *Office Word*.

Pour la base de registre

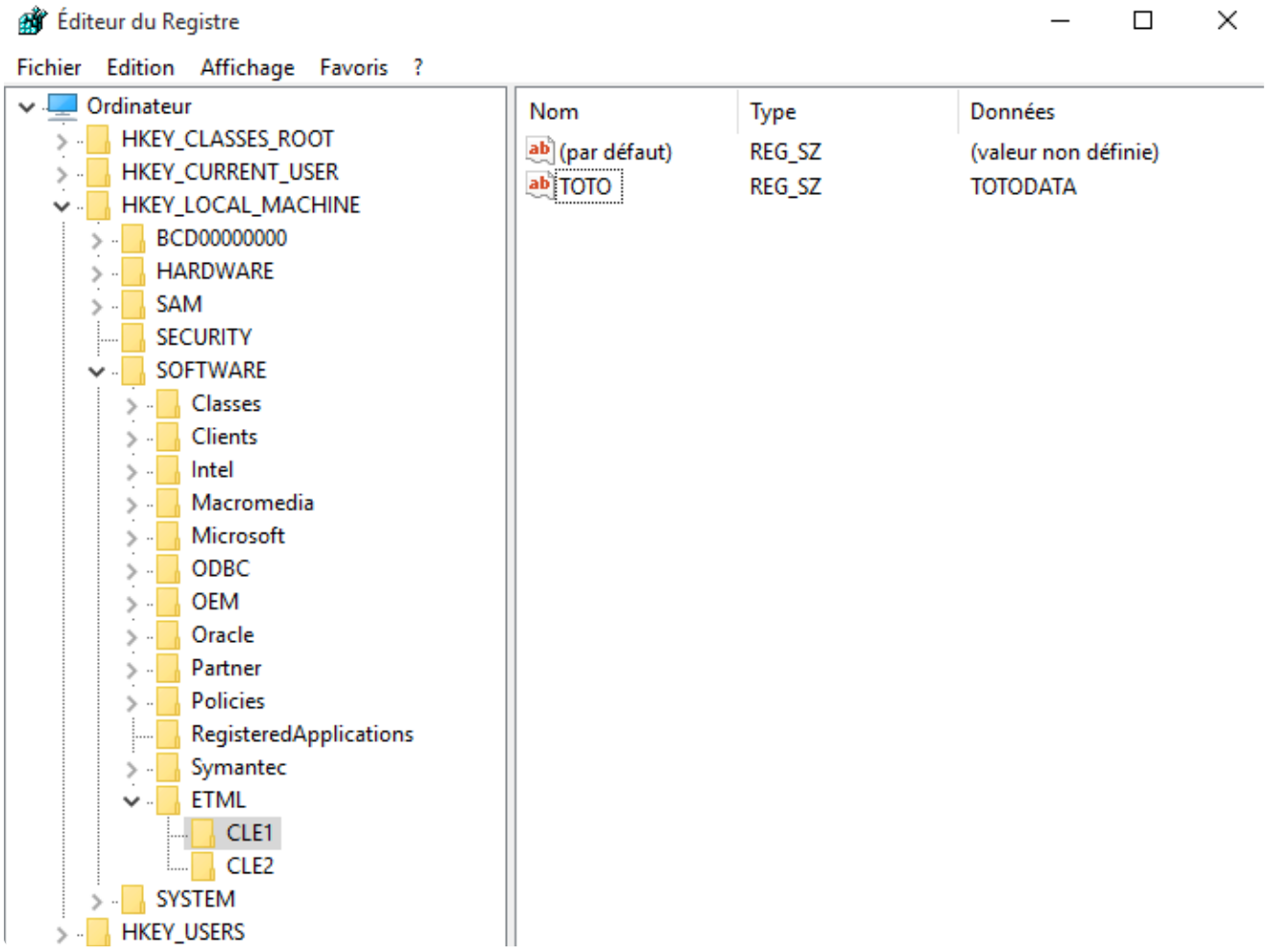
Pour rappel, la **base de registres** (**registry** en anglais) contient des informations essentielles au fonctionnement du système, il est donc impératif d'être prudent !

L'interface graphique se lance avec **regedit.exe** pour **Registry Editor**

Clé du Registre	Rôle principal
HKEY_CLASSES_ROOT	Contient les informations sur les applications et les associations entre types de fichiers et programmes.
HKEY_CURRENT_USER	Contient les paramètres de l'utilisateur actuellement connecté.
HKEY_LOCAL_MACHINE	Contient les informations générales de l'ordinateur : matériel, système et sécurité.
HKEY_USERS	Contient les paramètres de tous les utilisateurs du système.
HKEY_CURRENT_CONFIG	Contient les informations sur la configuration matérielle utilisée au démarrage actuel.

Pour avoir la racine de l'arborescence de la **registry**, on utilise **HKLM** et **HKCU**.

Dans l'exemple ci-dessous la registry a été modifiée avec l'ajout d'une clé *ETML* :



Get-ItemProperty

Permet d'obtenir des informations sur un fichier, un répertoire ou une clé de registre :

```powershell
Get-ItemProperty -Path HKLM:\Software\ETML\CLE1
```

Dans le cas d'un fichier, il n'y a pas de différence entre les 2 commandes, mais dans le cas d'une clé, les informations retournées par `Get-ItemProperty` sont plus complètes.

```
PS C:\Users\isastucki> Get-ItemProperty HKLM:\SOFTWARE\7-Zip


Path64       : C:\Program Files\7-Zip\
Path         : C:\Program Files\7-Zip\
PSPath       : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\7-Zip
PSParentPath : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE
PSChildName  : 7-Zip
PSDrive      : HKLM
PSProvider   : Microsoft.PowerShell.Core\Registry



PS C:\Users\isastucki> Get-Item HKLM:\SOFTWARE\7-Zip


    Hive: HKEY_LOCAL_MACHINE\SOFTWARE

Name                           Property
----                           --------
7-Zip                          Path64 : C:\Program Files\7-Zip\
                               Path   : C:\Program Files\7-Zip\
```

Get-ItemPropertyValue

Permet d'obtenir des informations sur une valeur de clé :

```powershell
Get-ItemPropertyValue -Path HKLM:\Software\ETML\CLE1 -Name TOTO
```

Les commandes de la partie **registry**, avec les `ItemProperty` seront utiles pour gérer les informations de la partie droite.

Pour gérer l'arborescence, on utilise les mêmes commandes que pour les arborescences des disques.

Il y a aussi sur le même principe les commandes ci-dessous :

```
CommandType     Name
-----------     ----
Cmdlet          Clear-ItemProperty
Cmdlet          Copy-ItemProperty
Cmdlet          Get-ItemProperty
Cmdlet          Get-ItemPropertyValue
Cmdlet          Move-ItemProperty
Cmdlet          New-ItemProperty
Cmdlet          Remove-ItemProperty
Cmdlet          Rename-ItemProperty
Cmdlet          Set-ItemProperty
```

WARNING

Pour des raisons de sécurité, avant de commencer les exercices, il faut ajouter la clé ETML, tel que dans la capture de l'exemple de l'éditeur de registres.

Résumé

- PowerShell fournit des cmdlets cohérentes pour gérer fichiers/dossiers.
- Les opérations courantes (copie, déplacement, suppression) s'appuient sur des paramètres comme **-Recurse**, **-Force**, **-WhatIf**.
- La registry se manipule comme une arborescence, avec des cmdlets dédiées pour lire les valeurs.

CIM / WMI

Pour administrer un poste Windows localement ou à distance, PowerShell s'appuie souvent sur CIM et WMI pour interroger les ressources du système.

Objectifs

À la fin de ce chapitre, vous serez capable de :

- Expliquer la différence entre **CIM** et **WMI** et pourquoi privilégier CIM.
- Interroger des informations système via **Get-CimClass** et **Get-CimInstance**.
- Utiliser des filtres et des requêtes (**WQL**) pour cibler les données.

Définition

CIM et WMI permettent de collecter des informations sur les ressources d'un ordinateur Windows local ou distant, par exemple la liste des utilisateurs, les services installés ou le nombre de cartes réseau. WMI correspond à l'implémentation de Microsoft fondée sur le standard CIM, qui définit un modèle commun pour représenter les informations de gestion système.

Le standard CIM a été défini par la **DMTF** (*Distributed Management Task Force*), un organisme industriel qui élabore des standards ouverts de gestion et d'interopérabilité pour les infrastructures informatiques.

La différence importante entre WMI et l'usage moderne des cmdlets CIM en PowerShell concerne surtout la communication à distance. Les cmdlets CIM utilisent par défaut **WS-Man** pour les connexions distantes, ce qui facilite souvent l'administration sur un réseau local, tandis que **DCOM** reste également possible dans certains cas.

Les avantages de CIM

Les avantages de **CIM** sur **WMI** sont :

- meilleures interopérabilité (Linux)
- facilitation de l'utilisation dans un LAN (pas de blocages sur les firewalls)
- compatible avec les systèmes Windows avant la version 3 avec RPC/DCOM

Les acronymes sont :

- **WMI** : Windows Management Instrumentation
- **CIM** : Common Interface Model

CIM permet la gestion de système Linux avec un framework comme **OMI = Open Management Infrastructure** (Microsoft) ou **OpenPegasus** (OpenGroup).

Il est préférable d'utiliser les commandes **CIM** que **WMI**, pour des systèmes à partir de Windows 7 SP1/Windows 8 et Server 2008R2 SP1/Server 2012.

CIM est disponible à partie de la version 3.0 de PowerShell, avant cela seul **WMI** était disponible. Dans des versions prochaines, PowerShell supprimera les cmdlets **WMI**.

> **Info**
>
> Dans ce cours, nous allons donc utiliser exclusivement les commandes **CIM** et le **CIM Explorer**. Les classes *WMI* sont toujours utilisables.

Les espaces de noms

Les classes dans **CIM Explorer** sont toutes contenues dans des espaces de noms. L'espace de nom par défaut est **root/cimv2**, il contient les classes pour le matériel de l'ordinateur et la configuration. Pour toutes les commandes qui suivent, l'espace de nom par défaut sera utilisé.

Les commandes

Nous utiliserons les commandes **CIM**. Il est possible d'utiliser directement les cmdlets ou d'utiliser une requête WQL(WMI Query Language).

Listes les classes :

```powershell
Get-CimClass
```

Lister une classe spécifique :

```powershell
Get-CimInstance -ClassName Win32_Process
```

Liste une classe spécifique en utilisant un filtre :

```powershell
Get-CimInstance -ClassName Win32_Process -Filter "Name = 'firefox.exe'"
```

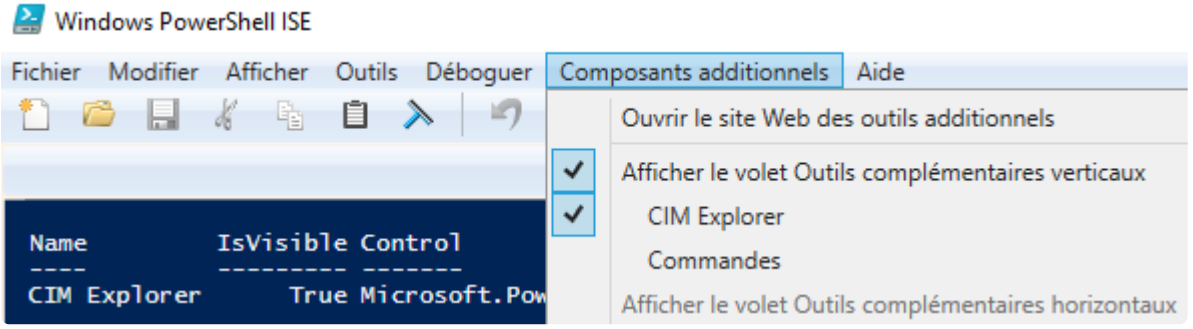
Liste une classe spécifique en utilisant une requête :

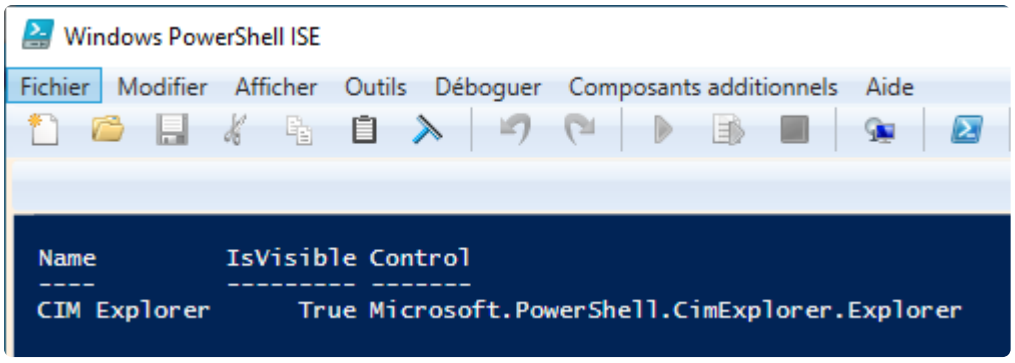
```powershell
Get-CimInstance -Query "SELECT * FROM Win32_Process WHERE Name = 'firefox.e
```

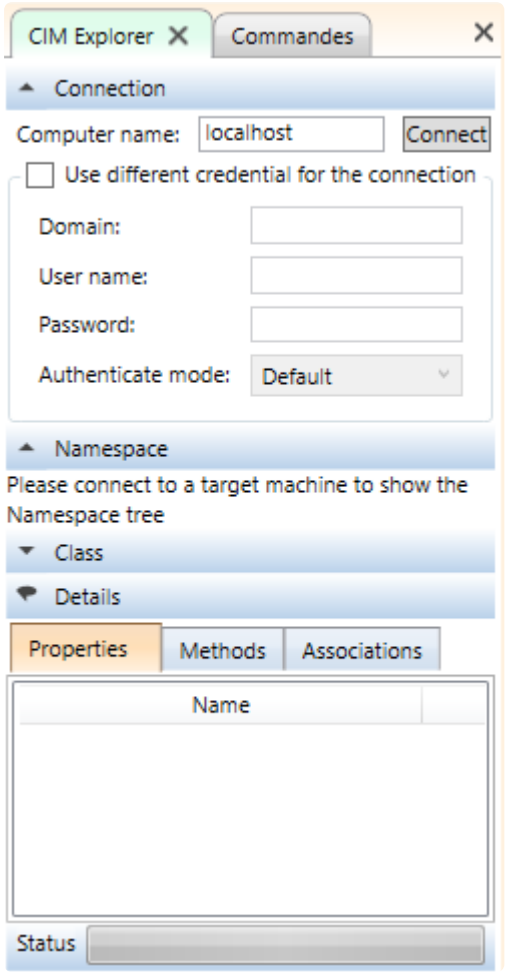
Les outils

CIM Explorer

Il y a un outil qui permet de visualiser les classes, directement dans ISE en ajoutant un onglet : **CIM Explorer**. Il faut l'extraire, et lancer le setup. Lorsque l'on lance ISE, il y a un composant additionnel.







WARNING

⚠️ Il faut noter que lorsque l'on ajoute le **CIMExplorer**, la politique d'exécution est modifiée avec **Unrestricted**. Dans notre cas, ce n'est pas un problème, mais dans une entreprise, il faudra la modifier selon les normes et prescriptions de l'entreprise.

La commande est **Get-CimInstance**. Elle permet d'obtenir aussi des informations sur des classes **Wmi** ou **Cim**

Avec VS Code

A noté qu'il n'a pas d'équivalant à CIM Explorer avec VS Code. Pour un usage moderne avec VS Code, utiliser directement le terminal intégré avec :

```powershell
Get-CimClass -Namespace root/cimv2
Get-CimClass -ClassName Win32_OperatingSystem
Get-CimInstance -ClassName Win32_OperatingSystem
Get-CimInstance -Namespace root/cimv2 -ClassName Win32_Process
```

Par exemple pour les informations systèmes :

```powershell
Get-CimInstance -ClassName Win32_OperatingSystem | Format-List
```

```
PS C:\Windows\system32> Get-CimInstance -ClassName CIM_OperatingSystem | Format-List


SystemDirectory : C:\Windows\system32
Organization    :
BuildNumber     : 18362
RegisteredUser  : admin
SerialNumber    : 00329-20000-00001-AA283
Version         : 10.0.18362
```

Résumé

- **CIM** (standard) et **WMI** (implémentation Windows) servent à collecter des infos système.
- Les cmdlets **CIM** s'appuient sur des protocoles adaptés aux réseaux modernes.
- **Get-CimInstance** permet d'interroger des classes avec filtres ou requêtes.

Variables & tableaux

Les variables et les tableaux sont essentiels en PowerShell, car ils permettent de stocker, organiser et manipuler efficacement les données utilisées dans un script.

Objectifs

À la fin de ce chapitre, vous serez capable de :

- Déclarer et manipuler des variables PowerShell (types, substitution, variables automatiques).
- Utiliser des chaînes (simples/doubles), here-strings et caractères d'échappement.
- Créer et parcourir des tableaux/collections selon le besoin.

Les variables

- Une variable commence toujours par un `$`.
- Le type est défini lors de chaque affectation, car le langage est non typé.
- On peut définir le type en ajoutant l'information devant la variable.

```powershell
[string]$texte = "Bonjour"
[int]$nombre = 123
[double]$prix = 19.95
[bool]$actif = $true
[char]$lettre = 'A'
[array]$tableau = @("rouge", "vert", "bleu")
[datetime]$date = Get-Date
```

Chaîne de caractères

Une variable peut être utilisée pour son nom ou pour son contenu :

```powershell
$a='Hello'
$b="World"
Write-Host '$a $b'
Write-Host "$a $b"
```

Il y a substitution avec des guillemets double :

```
PS C:\Users\isastucki> $a='Hello'

PS C:\Users\isastucki> $b="World"

PS C:\Users\isastucki> Write-Host '$a $b'
$a $b

PS C:\Users\isastucki> Write-Host "$a $b"
Hello World
```

La substitution de la propriété d'un objet se fait avec `$($objet.propriété)` :

```
PS C:\Windows\system32> $a=Get-ChildItem C:\config.sys

PS C:\Windows\system32> Write-Host "Taille fichier : $a.length octets"
Taille fichier : C:\config.sys.length octets

PS C:\Windows\system32> Write-Host "Taille fichier : $($a.length) octets"
Taille fichier : 10 octets

PS C:\Windows\system32>
```

Here-String

Chaîne de caractères sur plusieurs lignes, qui peut être contenue entre des guillemets simples ou doubles.

Avec **guillemets doubles** :

```powershell
$texte = @"
Bonjour,
Ceci est une chaîne
sur $number lignes.
"@
```

Avec **guillemets simples** :

```
$texte = @'
Bonjour,
```

```
Ceci est une chaîne
sur 3 lignes.
'@
```

Noté

Différence importante :

- `@" ... "@` : les variables sont interprétées
- `@' ... '@` : le texte est pris tel quel

Caractères d'échappement

Le backtick `` ` `` ou guillemet inversé, correspond au `\` du language C.

- Utilisé en fin de ligne indique que la commande continue sur la ligne suivante
- Si l'on veut que la variable `$a` ne soit pas substituée, alors on utilise `` `$a ``

Le tableau ci-dessous indique les principaux caractères d'échappement :

Caractères d'échappement	Résultat
`` `n ``	Nouvelle ligne
`` `r ``	Retour chariot
`` `t ``	Tabulation
`` `a ``	Alarm
`` `b ``	Backspace
`` `' ``	Guillemet simple
`` `" ``	Guillemet double
`` `0 ``	Null
`` `` ``	Backtick

On trouve le type d'une variable avec `Get-Member` ou `GetType` :

```powershell
$a | Get-Member
$a.GetType()
Get-Member -InputObject $a
```

Variables spéciales

Variables qui stockent des informations d'état "Automatic Variables" :

Variable	Description
`$_`	contenu du pipe, utilisé avec Where-Object, Foreach-Object, Filter, ...
`$?`	booléen sur le résultat de la dernière opération (Modifié par la dernière fonction avec exit0 : ok, exit1 : erreur ou exception : erreur bloquante pour l'exécution)
`$Args`	tableau des arguments passés à une fonction ou un script
`$Error`	tableau des erreurs, la dernière erreur sera donc `$Error[0]` et l'avant dernière `$Error[1]`, pour effacer $Error.Clear
`$Home`	Répertoire de l'utilisateur
`$Host`	Informations sur l'hôte
`$PWD`	Chemin absolu du répertoire courant

Constantes

C'est une variable avec une valeur définie que l'on ne peut pas changer

```
PS C:\Windows\system32> Set-Variable -Name const -Value 12 -Option Constant

PS C:\Windows\system32> $const
12
```

Opérateurs

- Arithmétiques : `+`, `-`, `*`, `/`, `%`, `System.Math`

```
PS C:\Users\isastucki> $a= 4

PS C:\Users\isastucki> if ($a%2 -eq 0){Write-Host "pair"}
pair

PS C:\Users\isastucki> $b=3

PS C:\Users\isastucki> if ($b%2 -eq 0){Write-Host "pair"}
```

- Comparaison ne respectant pas la casse : `-eq`, `-ne`, `-gt`, `-ge`, `-lt`, `-le`
- Comparaison respectant la casse : `-ceq`, `-cne`, `-cgt`, `-cge`, `-clt`, `-cle`

```
PS C:\Users\isastucki> "toto" -eq "Toto"
True

PS C:\Users\isastucki> "toto" -ceq "Toto"
False
```

- Logiques : `-and`, `-or`, `-not`, `!`, `-xor`

```
PS C:\Users\isastucki> $b
3

PS C:\Users\isastucki> ($b -ge 2) -and ($b -lt 5)
True
```

- Binaires : `-band`, `-bor`, `-bnot`, `-bxor`

```
PS C:\Users\isastucki> 5 -band 3
1
```

- Redirection : `>`, `>>`, `2>&1`, `2>`, `2>>`

```
PS C:\Users\isastucki> Write-Output "hello toto" > D:\temp\redirection.txt
```

Le fichier *redirection.txt* contient le texte :

```
Stop-Service toto 2>> t:\error.txt
```

Le fichier *error.txt* contient le message d'erreur :

- Types : `-is`, `-isnot`

```
PS C:\Users\isastucki> $a="toto"

PS C:\Users\isastucki> $a -is "string"
True

PS C:\Users\isastucki> $a -is "int"
False
```

- **Affectation :** `+=`, `-=`, `*=`, `/=`, `%=`, `++`, `--`

```
PS C:\Users\isastucki> $a=4

PS C:\Users\isastucki> $a+=3

PS C:\Users\isastucki> $a
7
```

- **Regex :** `-match`, `-notmatch`

```
PS C:\Users\isastucki> $regex1 = "to"

PS C:\Users\isastucki> "toto" -match $regex1
True

PS C:\Users\isastucki> "titi" -match $regex1
False

PS C:\Users\isastucki> "tito" -match $regex1
True

PS C:\Users\isastucki>
```

- **Sur les chaînes de caractères :** `-replace`, `-split`, `-join`, `-like`, `-notlike`

```
PS C:\Users\isastucki> $a
4

PS C:\Users\isastucki> $b
3

PS C:\Users\isastucki> $a -like $b
False
```

Tableaux à une dimension

Déclaration d'un tableau vide :

```powershell
$tabVide=@()
```

Déclaration avec initialisation :

```powershell
$tab1=10,20,30,40,50
$tab2=1..10
$tab3=@(1,2,3,4)
```

Déclaration avec type forcé :

```powershell
[int[]]$tab9=9,99,999
[double[]]$tab5=@()
```

Accès à une valeur par sont index :

```powershell
$tab1[2]
```

Accès à plusieurs valeurs :

```powershell
$tab9[0..2]
```

Affiche :

```
PS C:\Users\admin> [int[]]$tab9=9,99,999

PS C:\Users\admin> $tab9[0..2]
9
99
999
```

Affiche le tableau

```powershell
$tab3
```

Affiche :

```
PS C:\Users\eleve> $tab3=1,2,3,4

PS C:\Users\eleve> $tab3
1
2
3
4
```

On peut aussi parcourir le tableau avec une boucle **foreach** ou **for**

Affiche la dernière valeur :

```powershell
$tab3[-1]
```

Ajout de valeur :

```powershell
$tab3+=5
```

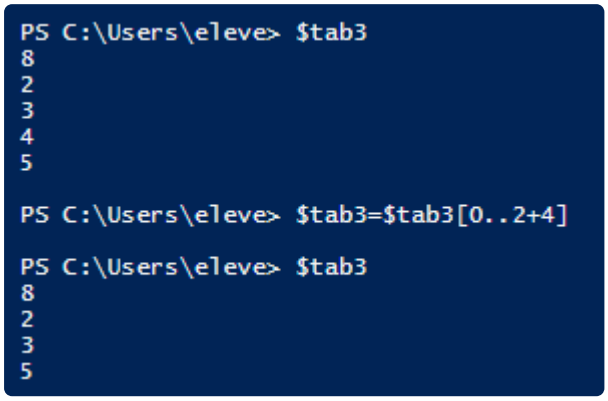
Modification par changement de valeur :

```powershell
$tab3[0]=8
```

Suppression par réaffectation :

```powershell
$tab3=$tab3[0..2+4]
```

Affiche :



Obtenir la longueur du tableau :

```powershell
$tab1.Length
$tab1.Count
```

Concaténation

Différence pour caractères et string :

```
PS C:\Users\admin> $hel='h', 'e', 'l'

PS C:\Users\admin> $lo= 'l', 'o'

PS C:\Users\admin> $hel+$lo
h
e
l
l
o

PS C:\Users\admin> $hel="hel"

PS C:\Users\admin> $lo= "lo"

PS C:\Users\admin> $hel+$lo
hello
```

Tableau à plusieurs dimensions

Déclaration avec initialisation :

```powershell
$tab11= (11,12,13),(21,22,23)
```

Accès à une ligne :

```powershell
$tab11[0]
```

Affiche : 11 12 13

Accès aux lignes :

```
PS C:\Users\eleve> foreach ($ligne in $tab11) {Write-Host "c'est la ligne :" $ligne}
c'est la ligne : 11 12 13
c'est la ligne : 21 22 23
```

Accès à une case :

```powershell
$tab11[0][1]
```

Affiche : 12

Tableaux associatifs

Initialiser un tableau vide :

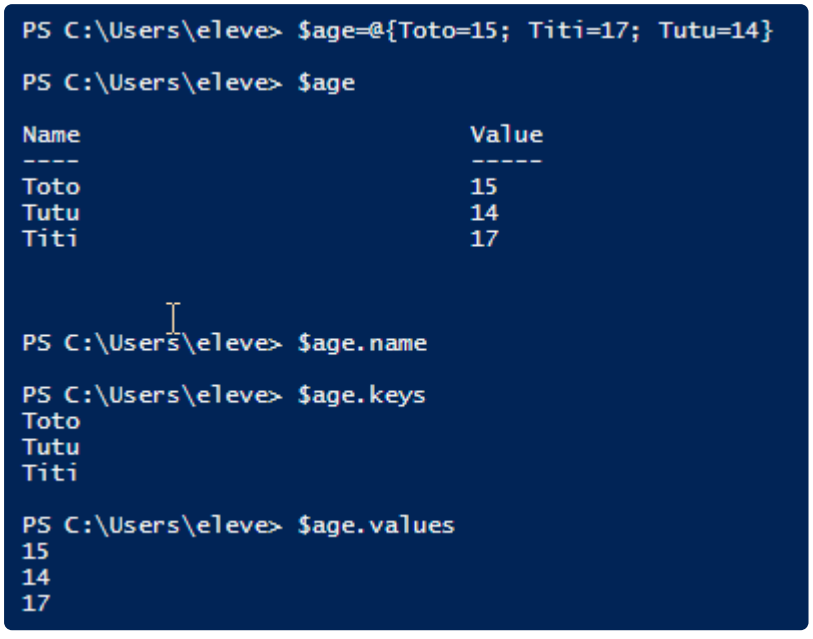
```powershell
$tabAssoVide={}
```

> **WARNING**
>
> Attention : un tableau `@()` utilise des `()` alors que un tableau associatif `@{}` des `{}`.

Dans un tableau associatif l'indice est une clé :

```powershell
$age=@{Toto=15; Titi=17; Tutu=14}
```

L'affichage utilise les propriétés des tableaux associatifs :



Les propriétés **Keys** et **Values** permettent d'avoir la liste des noms ou des valeurs.

Accéder à la valeur associée d'une clé :

```powershell
$age['Tutu']
```

Affiche : 14

Pour ajouter une clé/valeur à un tableau :

```powershell
$age+=@{Tata=16}
```

Résumé

- Les variables commencent par **$** et peuvent être typées explicitement.
- Les chaînes gèrent la substitution et les here-strings facilitent le multi-lignes.
- Les tableaux et variables automatiques sont essentiels pour traiter des données en script.

Scripts avancés

Ce chapitre traite des structures conditionnelles, des différentes boucles et des exceptions.

Objectifs

À la fin de ce chapitre, vous serez capable de :

- Écrire des **fonctions** et réutiliser du code.
- Utiliser des structures conditionnelles et des **boucles** pour automatiser des traitements.
- Gérer les erreurs avec **Try/Catch** et comprendre les erreurs terminées/non-terminées.

Fonctions

La syntaxe d'une fonction :

```powershell
function FuncHello()
{
    param($name)
    $date = (Get-Date).ToShortDateString()
    Write-Host "Salut $name, nous sommes le $date"
}
```

La fonction est appelée avec le nom en paramètre d'entrée :

```powershell
FuncHello("Toto")
```

Un autre exemple avec le résultat :

Isabelle-Test-FunctionDate.ps1*

```powershell
<#
.NOTES
    ********************************************************************************
    ETML
    Nom du script: Isabelle-Test-FunctionDate.ps1
    Auteur: Isabelle Stucki
    Date:   juillet 2023
    ********************************************************************************

.SYNOPSIS
    Ecrit la date à l'écran

.DESCRIPTION
    Affiche la date avec la couleur déinie dans la variable

.EXAMPLE
    .\Isabelle-Test-FunctionDate.ps1 si la variable $color="yellow"
    Résultat à l'affichage : Nous sommes le : 26.07.2023 11:04:27 en jaune

.EXAMPLE
    .\Isabelle-Test-FunctionDate.ps1 si la variable $color="red"
    Résultat à l'affichage : Nous sommes le : 26.07.2023 11:05:57 en rouge
#>

# Pas de paramètre

################################################################################
# Zone de définition des variables et fonctions
# La couleur pour l'affichage
$color="yellow"

# Fonction pour l'affichage de l'heure avec une couleur définie
function Write-Date ([string]$color)
{
    $date = Get-Date
    Write-Host "Nous sommes le :" $date -ForegroundColor $color
}

################################################################################
# Zone de tests comme les paramètres renseignés ou les droits administrateurs

################################################################################
# Corps du script

Write-Date -color $color
```

```
PS C:\Users\ISI> S:\Isabelle-Test-FunctionDate.ps1
Nous sommes le : 26.07.2023 12:29:09
```

Structure conditionnelle

Structure permettant un choix en fonction d'une expression booléenne.

powershell

```powershell
if (expression booléenne)
    {instructions}
elseif (expression booléenne)
    {instructions}
else
    {instructions}
```

Switch

Permet des choix multiples.

```powershell
switch (expression)
{
    valeur1
        {instructions}
    valeur2
        {instructions}
    default
        {instructions}
}
```

Boucles

While

L'expression booléenne est testée en premier.

```powershell
while (expression booléenne)
    {instructions}
```

Do-While

Il y aura au minimum un passage étant donné que l'expression booléenne est testée à la fin.

```powershell
do
    {instructions}
while (expression booléenne)
```

For

Permet de définir le point de départ et celui d'arrivée.

```powershell
for (expression initiale; expression booléenne; expression finale)
    {instructions}
```

L'expression initiale permet d'initialiser le compteur, alors que la finale permet de l'incrémenter.

Foreach-Object

Permet de parcourir les valeurs d'une collection (généralement d'un tableau).

```powershell
Foreach ($Element in $Collection)
    {instructions}
```

Il y a d'autres commandes utiles pour les objects.

```
PS C:\Windows\system32> Get-Command *object*

CommandType     Name

-----------     ----
Cmdlet          Compare-Object
Cmdlet          ForEach-Object
Cmdlet          Get-WmiObject
Cmdlet          Group-Object
Cmdlet          Measure-Object
Cmdlet          New-Object
Cmdlet          Register-ObjectEvent
Cmdlet          Remove-WmiObject
Cmdlet          Select-Object
Cmdlet          Sort-Object
Cmdlet          Tee-Object
Cmdlet          Where-Object
Application     NetCfgNotifyObjectHost.exe
```

Try-Catch

Cette structure permet d'essayer une instruction et de récupérer l'erreur en cas de problème.

```powershell
try
    {instructions}
```

```
catch
    {message}
```

Ces instructions fonctionnent correctement lorsque les erreurs sont des **erreurs de type terminées**.

Si elles sont **non-terminées** alors il faut spécifier l'arrêt de l'instruction :

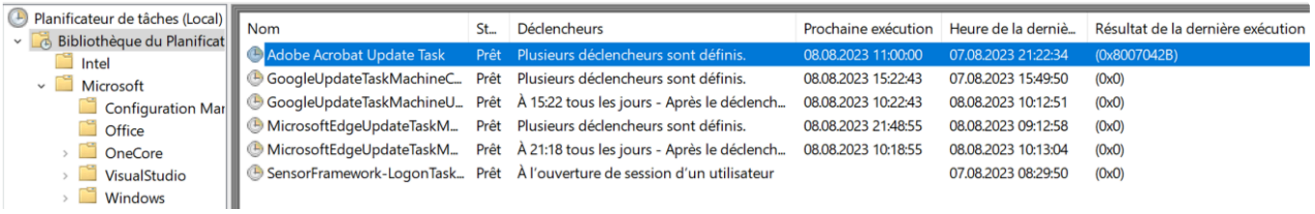
powershell

```
try
    {instructions} -Erroraction Stop
catch
    {message}
```

Il est possible d'avoir plusieurs blocs **Catch** et on peut spécifier le type d'erreur avec par exemple `catch[System.IO.IOException]`.

Pour trouver le nom du type complet et correct de l'erreur, il faut utiliser `$error[0].exception.gettype().fullname`.

Une fonction, un script ou une tâche peuvent se terminer « mal » par un exit1 ou une exception. Le résultat de la dernière exécution sera n'importe quelle valeur sauf le 0. Tout ce qui est bien terminé `exit = exit0`, aura un résultat à « 0x0 ». C'est ce que l'on observe dans le planificateur de tâches ci-dessous.



Ceci permet de faire des contrôles sur « les fichiers de log », que lorsque l'on sait qu'il y a eu un problème.

Résumé

- Les fonctions structurent et réutilisent le code.
- Les conditions et boucles permettent d'automatiser des décisions et répétitions.

- **Try/Catch** aide à contrôler le flux en cas d'erreur (avec `-ErrorAction Stop` si nécessaire).

Traitement de fichiers

Afin de pouvoir traiter les fichiers, il faut d'abord comprendre comment ils sont générés et organisés.

Objectifs

À la fin de ce chapitre, vous serez capable de :

- Créer, lire et compléter des fichiers texte avec **Out-File** / **Get-Content** / **Add-Content**.
- Importer et exporter des données tabulaires avec **Import-CSV** et **Export-CSV**.
- Vérifier l'existence de chemins et rechercher du contenu avec **Test-Path** et **Select-String**.

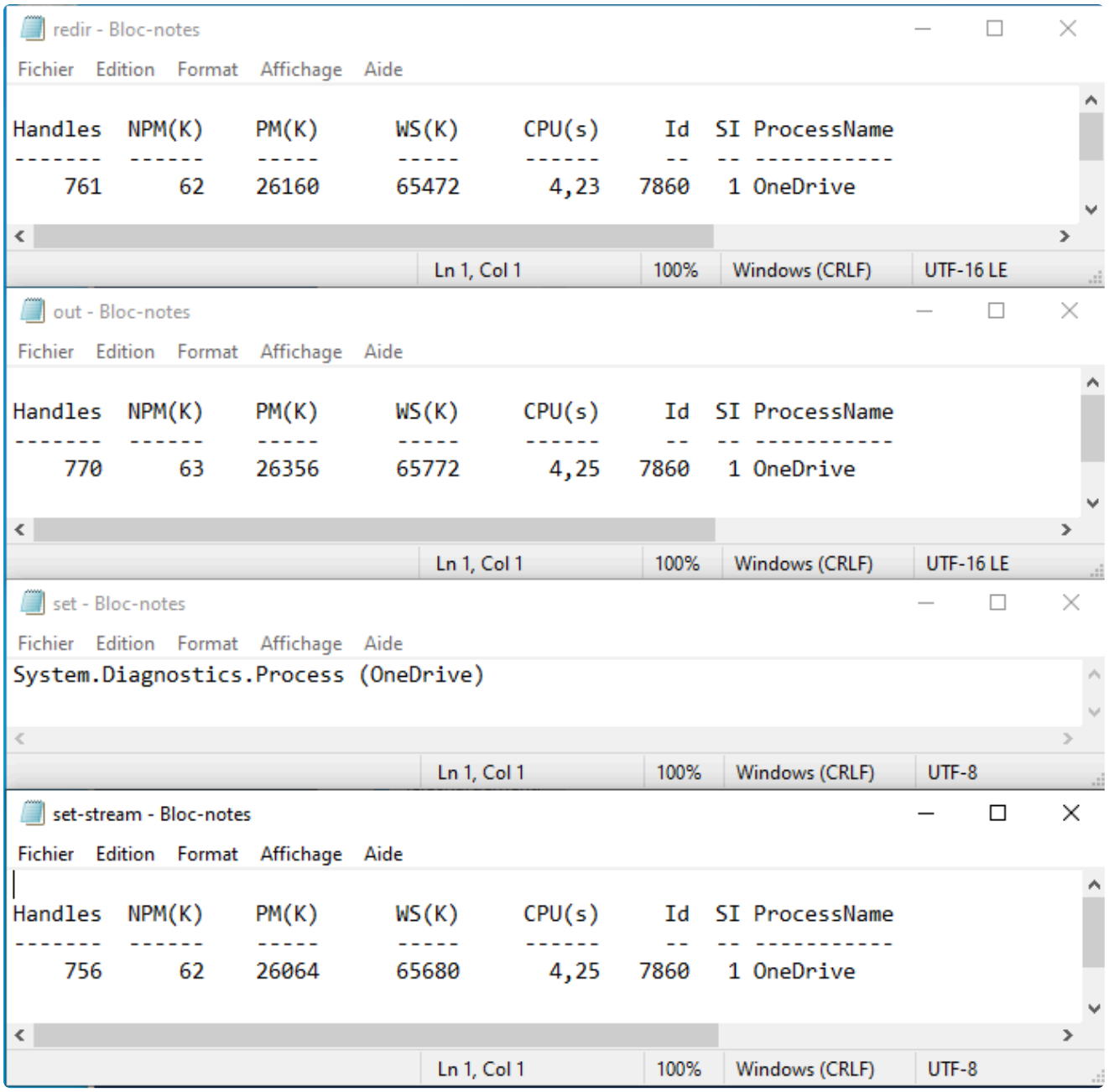
Les commandes permettant la création de fichiers sont `Out-File`, `Set-Content` et la redirection avec `>` ou `>>`.

Il y a cependant quelques différences entre ces possibilités :

- La redirection ou `Out-File`, donneront un fichier tel qu'affiché dans la console, alors que `Set-Content` aura comme résultat une représentation des objets.
- La commande `Out-String -Stream` convertit tous les objets sous forme de chaines. Les différentes commandes :

```
PS C:\Users\admin> Get-Process -Name OneDrive > T:\redir.txt
PS C:\Users\admin> Get-Process -Name OneDrive | Out-File T:\out.txt
PS C:\Users\admin> Get-Process -Name OneDrive | Set-Content T:\set.txt
PS C:\Users\admin> Get-Process -Name OneDrive | Out-String -Stream | Set-Content T:\set-stream.txt
```

Et les résultats :



Dans la pratique, on utilisera plutôt la redirection ou `Out-File` pour les fichiers textes et `Set-Content` pour les fichiers binaires.

La commande `Out-File` offre plus de possibilités avec les paramètres `-Encoding` (par défaut **UTF-16 LE** sous Windows PowerShell 5.1, et **UTF-8** sous PowerShell 7+) ou `-Width` pour le nombre de caractères max par ligne.

De plus avec `Out-File`, le retour chariot (`CR` = Carriage Return) et le saut de ligne (`LF` = Line Feed) sont pris en charge.

Out-File

Cette commande permet de créer un nouveau fichier qui sera semblable au contenu de la console

```powershell
Get-Service | Where-Object{$_.Status -eq "Running"} | Out-File -FilePath T:
```

Get-Content

Permet de mettre le contenu d'un fichier dans une variable de type tableau.

Le tableau sera composé des lignes du fichier et chaque ligne sera donc une chaîne de caractères.

```powershell
$a=Get-Content "d:\test\toto.txt"
```

Pour parcourir le fichier :

```powershell
foreach ($line in $a)
    {Write-Host $line}
```

Le contenu de `$line` est aussi un tableau.

Add-Content

Permet d'ajouter une ligne dans un fichier :

```powershell
$newLine = "ligne à ajouter"
Add-Content -Path "d:\truc\machin.txt" -Value $newLine
```

Info

A noter que `Clear-Content` effacera le contenu, mais pas le fichier.

Import-CSV

Rappelons d'abord qu'un fichier *.csv* (*Comma Separated Values*) est un fichier contenant des informations séparées par défaut, par des virgules... par exemple une liste extraite d'un fichier Excel.

Il est possible de choisir un autre séparateur.

Lorsque l'on utilise cette commande, le fichier sera aussi sous forme de tableau avec des cases contenant les lignes du fichier, mais en plus chaque case sera un tableau associatif des éléments du fichier.

```powershell
$users= Import-CSV "x:\temp\liste-utilisateurs.csv"
```

Un exemple d'affichage des informations d'un fichier CSV :

```
12
13  $filePath = "T:\test-classe.csv"
14
15  $eleves = Import-Csv $filePath
16
17  for ($i=0; $i -lt $eleves.Length ;$i++)
18  {
19      Write-Host $eleves[$i].prenom $eleves[$i].hobby
20  }
21
```

```
PS C:\Windows\system32> S:\Scripts\ISI-Test-CSV.ps1
Monia tennis
Yann curling
Baptiste ski
Quentin tennis
Anthony foot
Franck ski
Marco bad
Romain bad
```

Export-CSV

Permet d'exporter des informations directement sous format **.csv** :

```powershell
Get-Service | where{$_.status -eq "running"} | Export-CSV x:\test2.csv
Get-Service | where{$_.status -eq "running"} | Out-File x:\test3.csv
```

Il faut noter que les fichiers *test2.csv* et *test3.csv* n'auront pas le même contenu.

Avec `Out-File`, l'affichage est simplement redirigé vers un fichier, quelle que soit son extension, alors que pour `Export-CSV` le fichier contient toutes les données séparées par des virgules.

Version encore plus proche de ton modèle initial :

ConvertTo-Json

Permet d'exporter des informations directement sous format **.json** :

```powershell
Get-Service | where{$_.status -eq "running"} | Select-Object -First 3 | Conv
Get-Service | where{$_.status -eq "running"} | Select-Object -First 3 | Out
```

Il faut noter que les résultats obtenus n'auront pas le même contenu.

Avec `Out-File`, l'affichage est simplement redirigé vers un fichier, quelle que soit son extension, alors que pour `ConvertTo-Json` le contenu est converti en structure JSON.

Test-Path

Vérifier l'existence d'un fichier ou répertoire.

Exemple de vérification d'un répertoire :

```powershell
if (Test-Path "d:\temp")
    {instructions si le répertoire existe}
else
    {instructions si le répertoire n'existe pas}
```

Select-String

Permet d'extraire les phrases qui contiennent l'information recherchée.

```powershell
Select-String -Path T:\test\toto.txt -Pattern "toto"
```

Le paramètre `-Quiet` permet d'obtenir un booléen et `-CaseSensitive` pour que la recherche prenne en compte la casse.

Il est aussi possible de chercher avec une expression régulière.

```
PS C:\Users\admin> Select-String -Path T:\test\services-w.txt -Pattern 'Windows'

T:\test\services-w.txt:5:Running  WbioSrvc           Service de biométrie Windows
T:\test\services-w.txt:6:Running  Wcmsvc             Gestionnaire des connexions Windows
T:\test\services-w.txt:9:Running  WinDefend          Service antivirus Windows Defender
T:\test\services-w.txt:11:Running  Winmgmt            Infrastructure de gestion Windows
T:\test\services-w.txt:12:Running  WLMS               Windows Licensing Monitoring Service
T:\test\services-w.txt:16:Running  WSearch            Windows Search
T:\test\services-w.txt:18:Windows
```

Get-Date

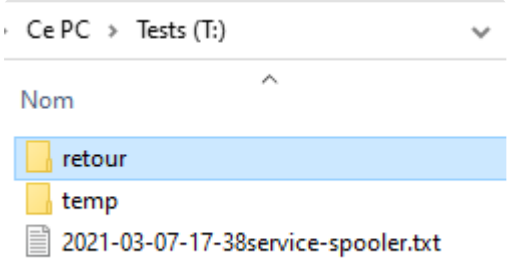
Dans plusieurs cas, il est nécessaire d'avoir des fichiers qui ont un nom unique.

Par exemple, pour des fichiers *log* ou pour organiser des documents dans un dossier.

Une façon simple consiste à ajouter la date à un nom de fichier.

```powershell
$date = Get-Date -Format "yyyy-MM-dd-HH-mm"
$filePath = "T:\"
$fileName = "service-spooler.txt"
Get-Service *spooler* >> "$filePath$date$fileName"
```

Et le fichier de résultat :



Info

Noter que la commande `Get-Date` est utilisée avec le paramètre `-Format`, ce qui permet de choisir la mise en forme de la date.

Résumé

- La création/écriture de fichiers se fait via redirection, **Out-File** ou **Set-Content** selon le type.
- Les cmdlets **Import-CSV** / **Export-CSV** manipulent des objets, pas seulement du texte.
- **Test-Path** et **Select-String** sont utiles pour automatiser des contrôles et recherches.

Bases de données

Manipuler une base de données avec Powershell.

Objectifs

À la fin de ce chapitre, vous serez capable de :

- Décrire les 3 étapes d'accès à une base de données (connexion, commande, récupération).
- Identifier les classes principales utilisées côté PowerShell/.NET.
- Lire et adapter un exemple de script pour exécuter une requête SQL.

Pour accéder à une base de données, il y a 3 étapes : la connexion, la commande et la récupération du résultat de la requête.

Pour ce faire on utilise les classes *SqlConnection*, *SqlCommand*, *SqlDataAdapter* du namespace *.NET*.

> **Namespace .NET**
>
> Les exemples ci-dessous utilisent `System.Data.SqlClient` (namespace historique, toujours fonctionnel sous Windows PowerShell 5.1). Pour les projets modernes avec PowerShell 7+ / .NET 6+, préférer `Microsoft.Data.SqlClient` qui reçoit les mises à jour de sécurité actives.

SqlCommand	Represents a Transact-SQL statement or stored procedure to execute against a SQL Server database. This class cannot be inherited.
SqlCommandBuilder	Automatically generates single-table commands that are used to reconcile changes made to a DataSet with the associated SQL Server database. This class cannot be inherited.
SqlConnection	Represents a connection to a SQL Server database. This class cannot be inherited.
SqlConnectionStringBuilder	Provides a simple way to create and manage the contents of connection strings used by the SqlConnection class.
SqlCredential	SqlCredential provides a more secure way to specify the password for a login attempt using SQL Server Authentication. SqlCredential is comprised of a user id and a password that will be used for SQL Server Authentication. The password in a SqlCredential object is of type SecureString. SqlCredential cannot be inherited. Windows Authentication (`Integrated Security = true`) remains the most secure way to log in to a SQL Server database.
SqlDataAdapter	Represents a set of data commands and a database connection that are used to fill the DataSet and update a SQL Server database. This class cannot be inherited.

Pour plus de détails : Doc MS SQLClient

Connexion à une base de donnée

Pour pouvoir connecter une base de données, il faut plusieurs informations : l'url, le nom et l'utilisateur de la base ainsi que le mot de passe.

L'ouverture de la connexion à la base :

```powershell
# Informations pour la connexion à la base de données
# Cette base est locale, il est cependant possible de donner une adresse IP
$sqlServer = "localhost\SQLEXPRESS"
```

```powershell
# Le nom de la base créée dans SQLExpress
$sqlDBName = "dbTestPS"
# Le login et pwd, toujours contrôler que l'utilisateur a bien les droits su
$user = "sa"
$pwd = ".etm1-"

# Connexion à la base de données
$sqlConnection = New-Object System.Data.SqlClient.SqlConnection
$sqlConnection.ConnectionString = "Server = $sqlServer; Database = $sqlDBNa

# ouvrir la connexion avec la base, toujours ouvrir et fermer la connexion
$sqlConnection.Open()
```

Création de la commande

Il faut noter que la requête est définie dans une variable, donc un objet et qu'elle est utilisée pour la création de la commande.

Dans l'exemple, on fait une recherche sur les surnoms débutant avec les lettres *DU*.

Toutes les requêtes SQL sont possibles :

powershell

```powershell
# La requete sql que l'on veut effectué sur la table, ici la recherche des
$sqlQuery = "SELECT * FROM [db.TestPS].[dbo].[Table_1] WHERE surname like '

# Lorsque la connexion est ouverte, on peut créer la commande pour la requê
$sqlCmd = New-Object System.Data.SqlClient.SqlCommand($sqlQuery,$sqlConnecti
```

Récupération du résultat

Pour avoir les données résultant de la requêtes, on utilise la commande définie précédemment ainsi qu'une variable, qui peut contenir un nombre important d'informations !

On n'oublie pas la fermeture de la connexion :

```powershell
# Pour récupérer les informations en retour
# La commande SqlDataAdapter permet mettre les résultats en mémoire sous fo
$sqlAdapter = New-Object System.Data.SqlClient.SqlDataAdapter($sqlCmd)
$dataSet = New-Object System.Data.DataSet

# Le résultat sera contenu dans la variable $dataSet. Le résultat sera reto
$sqlAdapter.Fill($dataSet)

# Fermeture de la connexion à la base
$sqlConnection.Close()
```

- 5.- Créer une DB contenant une table avec nom, prénom et surnom, qui sont les 2 premiers caractères du nom et prénom. Écrire un script avec 1 paramètre d'entrée pour les 2 premières lettres de la recherche sur le surnom et qui affichage le résultat selon l'exemple ci-dessous.

```
PS C:\Users\admin> Y:\scripts\Get-DBInfos.ps1 -firstCars DU
3

name   firstName surname
----   --------- -------
Dupont Jean      DUJE
Durant Pierre    DUPI
Duncan Tim       DUTI
```

Résumé

- L'accès à une base passe par une connexion, une commande SQL, puis la récupération des résultats.
- Les classes .NET (connexion/commande/adaptateur) structurent l'exécution.
- La fermeture de la connexion fait partie du « bon réflexe ».

Scheduling

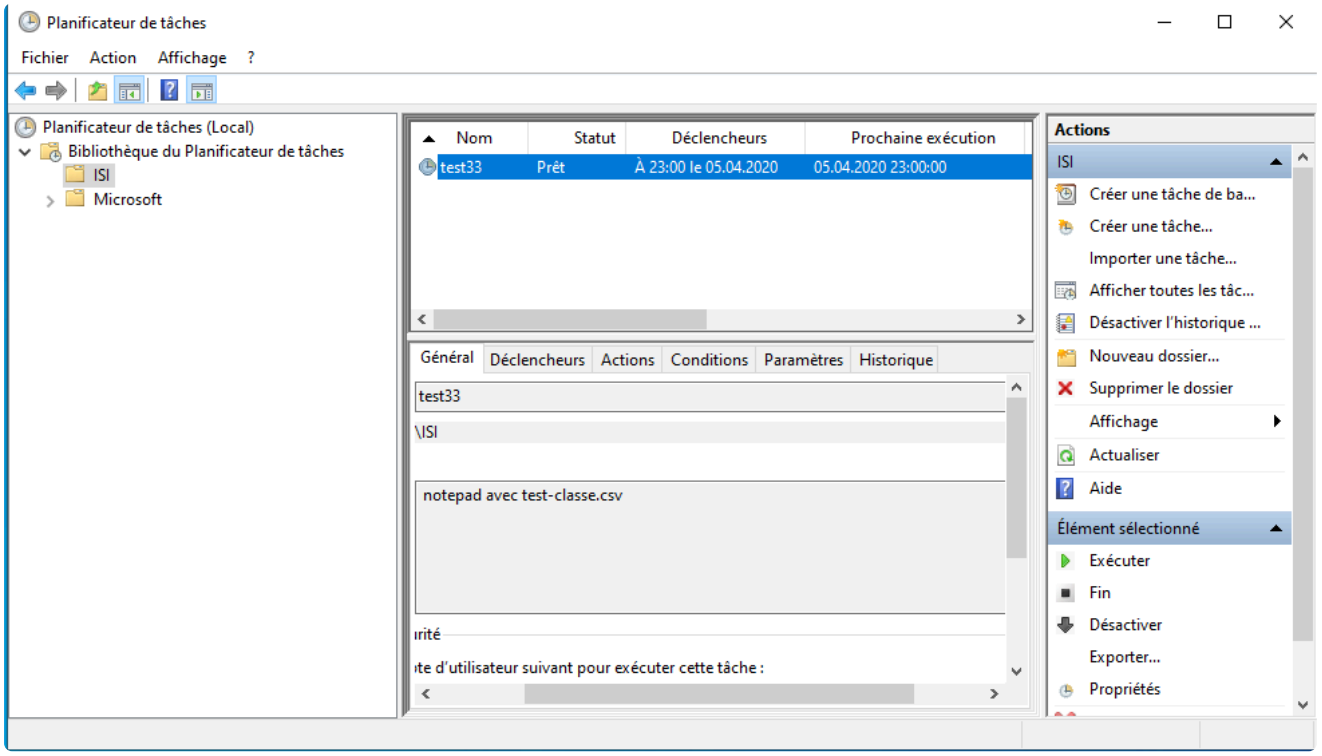
Le scheduler, planificateur de tâches en français, permet d'exécuter une action à un moment défini.

Objectifs

À la fin de ce chapitre, vous serez capable de :

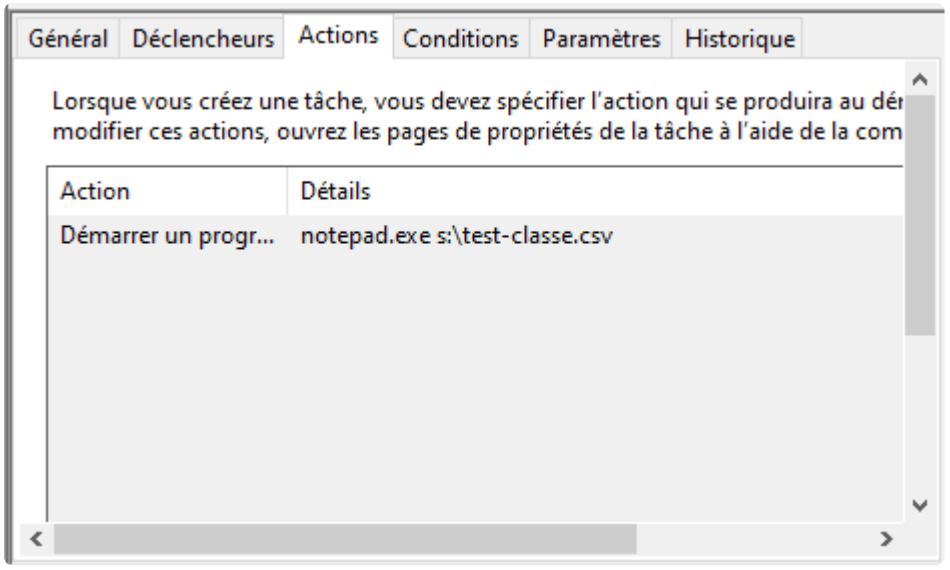
- Expliquer le principe d'une tâche planifiée (action + déclencheur + enregistrement).
- Créer une action et un déclencheur via les cmdlets **ScheduledTasks**.
- Enregistrer une tâche avec **Register-ScheduledTask**.

Dans l'exemple ci-dessous, l'ouverture du fichier **test-classe.csv** avec **notepad.exe**, doit s'exécuter une fois à 17h



Pour pouvoir ajouter une tâche dans le planificateur avec un script, il y a plusieurs étapes à effectuer

La première est de définir une **action** (**action**)

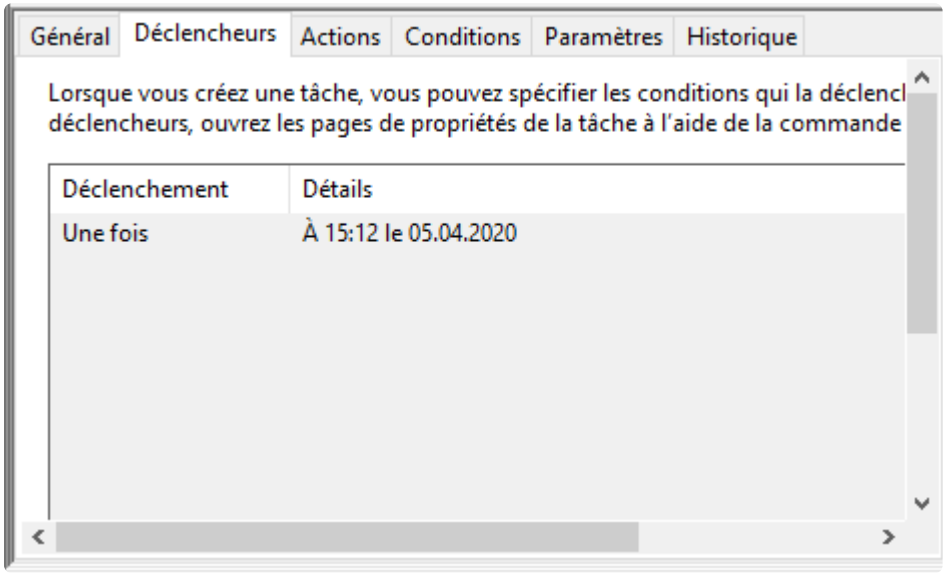


Pour le code, on doit définir une variable

```powershell
$action=New-ScheduledTaskAction -Execute 'notepad.exe' -Argument 's:\test-c
```

Si on veut lancer un script, on utilisera **powershell.exe** et le script avec son chemin comme **argument**

La seconde est de définir le **déclencheur** (**trigger**)

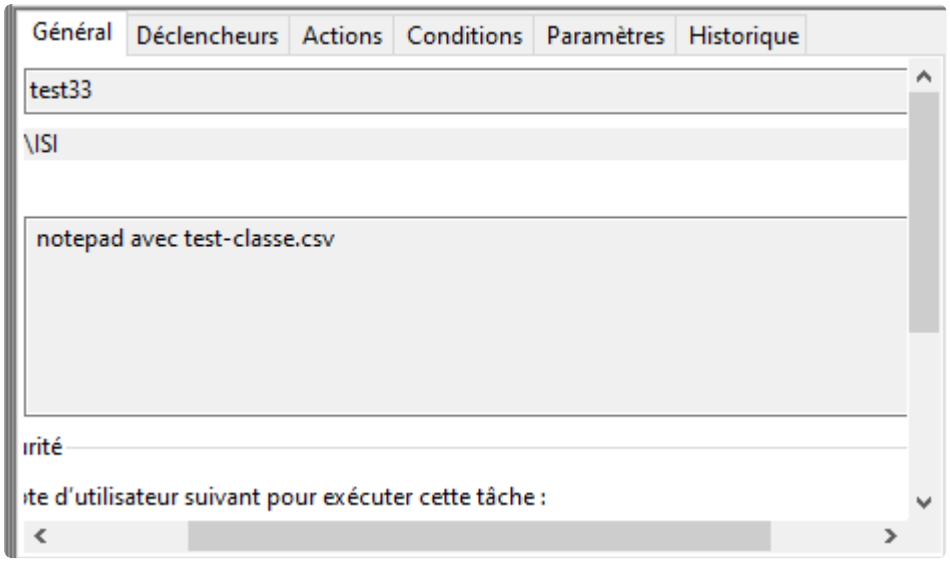


Pour le code, on définit aussi une variable

```powershell
$trigger=New-ScheduledTaskTrigger -Once -At $hour
```

Dans cet exemple, la tâche s'exécutera une fois et l'heure d'exécution est définie dans les paramètres du script **$hour**

Enfin pour la troisième étape, il faut enregistrer la tâche avec les variables définies



Pour le code, on utilise la cmdlet **Register-ScheduledTask** pour l'enregistrement de la tâche

```powershell
Register-ScheduledTask -Action $action -Trigger $trigger -TaskName $taskNam
```

Info

Le nom, le chemin et la description sont définis dans des variables, mais on pourrait aussi les passer en paramètres!

Dans ISE, le lancement du script donne :

```
PS C:\Users\admin> S:\Scripts\ISI-Open-File.ps1 -file S:\test-classe.csv -hour 15:12

TaskPath                                 TaskName                                 State
--------                                 --------                                 -----
\ISI\                                    test33                                   Ready
```

Résumé

- Une tâche planifiée combine une **action**, un **trigger**, puis un enregistrement.
- Les cmdlets **New-ScheduledTaskAction** / **New-ScheduledTaskTrigger** construisent les objets nécessaires.
- **Register-ScheduledTask** finalise la création avec nom, chemin et description.

Remoting

Le remoting permet d'agir sur une autre machine à distance. C'est l'essence même du Powershell.

Objectifs

À la fin de ce chapitre, vous serez capable de :

- Identifier les prérequis réseau/sécurité pour le remoting PowerShell.
- Activer et configurer **WinRM** pour permettre l'exécution distante.
- Comprendre la logique d'un script qui exécute une action sur une machine distante.

Il y a quelques prérequis à vérifier sur les 2 machines :

- Dans VBox, réseau, il faut choisir réseau interne
- Win10 dans WORKGROUP et changer le nom de machine pour PC1 et PC2
- Désactiver le par-feu (possible dans le cas d'un labo mais pas dans une entreprise)
- IPV6 désactivé et IPV4 avec adresse fixe - les 2 VM doivent pouvoir « ping » le nom de l'autre
- Compte faisant partie du groupe administrateurs sur la machine distante (controler l'utilité de l'utilisateur qui lance le script avec droit admin)
- Autorisation pour l'exécution des scripts

Afin de pouvoir faire du remoting, il faut activer le service winrm (Windows Remote Managment) et définir les hôtes à distance

Sur les 2 PC exécuter en tant qu'admin :

- `Enable-PSRemoting -SkipNetworkProfileCheck -Force`
- `Set-Item WSMan:\\localhost\\Client\\TrustedHosts *`
- `Restart-Service winrm redémarrer le service winrm`
- Redemarrer les machines

Le code du script lancé sur PC1 pour la création distante d'un fichier sur PC2 :

```powershell
# Le PC de destination, doit être accessible avec un ping
$compName="PC2"

# Va contenir le nom d'utilisateur et mot de passe SecureString
$cred=Get-Credential

# Ouverture de la session sur la machine de destination
$sess=New-PSSession -ComputerName $compName -Credential $cred

# Avec l'accès à la machine de destination, on envoie la commande pour ajou
Invoke-Command -Session $sess -ScriptBlock{New-Item -Path C:\Users\ISI\Desk

# On supprime la session
Remove-PSSession $sess
```

Variables dans un ScriptBlock distant

Les variables locales ne sont **pas** automatiquement disponibles dans le `ScriptBlock` distant. Deux approches :

- **`$using:nomVariable`** (recommandé, PowerShell 3+) : préfixe la variable locale pour l'injecter dans le bloc distant.
- **`-ArgumentList`** : passe les valeurs comme paramètres, à réceptionner avec `param(…)` dans le bloc.

```powershell
$dest = "C:\Users\ISI\Desktop"
$name = "toto.txt"

# Avec $using:
Invoke-Command -Session $sess -ScriptBlock {
    New-Item -Path $using:dest -Name $using:name
}

# Avec -ArgumentList
Invoke-Command -Session $sess -ArgumentList $dest, $name -ScriptBlock {
    param($path, $n)
    New-Item -Path $path -Name $n
}
```

Voici un exemple simple, basé sur le même principe, mais cette fois pour exécuter une commande à distance sur le PC2 :

```powershell
# Le PC de destination doit être accessible
$compName = "PC2"

# Demande les identifiants
$cred = Get-Credential

# Ouvre une session distante
$sess = New-PSSession -ComputerName $compName -Credential $cred

# Exécute une commande sur la machine distante
Invoke-Command -Session $sess -ScriptBlock {
    Get-ComputerInfo | Select-Object CsName, WindowsProductName, OsVersion
}

# Ferme la session
Remove-PSSession $sess
```

Et voici une exemple qui copie un fichier du PC2, sur le PC1 depuis le PC1 :

```powershell
# Le PC de destination doit être accessible
$compName = "PC2"

# Demande les identifiants
$cred = Get-Credential

# Ouvre une session distante
$sess = New-PSSession -ComputerName $compName -Credential $cred

# Copie un fichier distant vers le PC local
Copy-Item -Path "C:\Temp\infos.txt" -Destination "C:\Temp\infos_copie.txt"

# Ferme la session
Remove-PSSession $sess
```

Info

Dans ces exemples l'on utilise une authentification simple par login avec `Get-Credential`.

Résumé

- Le remoting s’appuie sur **WinRM** et nécessite une configuration cohérente des deux côtés.
- Les prérequis (réseau, pare-feu, droits admin, policy) conditionnent le succès des commandes distantes.
- Les scripts distants doivent gérer correctement paramètres et authentification.

Informations complémentaires

📥 Support remoting pour PS (PDF) ↗

Diagramme de flux

Un diagramme de flux aide à comprendre rapidement le déroulement logique d'un script PowerShell, étape par étape.

Objectifs

À la fin de ce chapitre, vous serez capable de :

- Expliquer l'intérêt d'un diagramme de flux pour analyser un processus.
- Reconnaître les symboles principaux et leur rôle.
- Lire un diagramme et en déduire les étapes d'un traitement.

Pourquoi utiliser un diagramme de flux

- Visualiser les étapes principales d'un processus / d'une fonction / d'une séquence.
- Moyen de communication pour comprendre, analyser, standardiser et améliorer le processus en réduisant les délais et les étapes superflues.

Comment le représenter

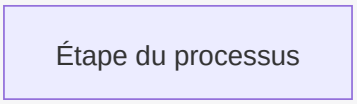
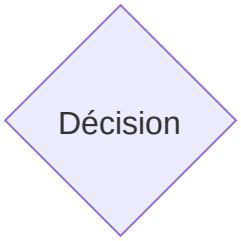
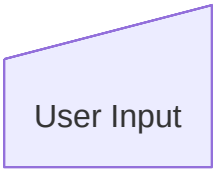
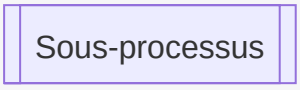
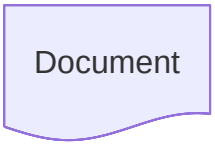
Le diagramme de flux se représente à l'aide de symboles.

Voici un **fichier Markdown complet** prêt à copier dans VitePress, avec un **tableau HTML** dans le Markdown :

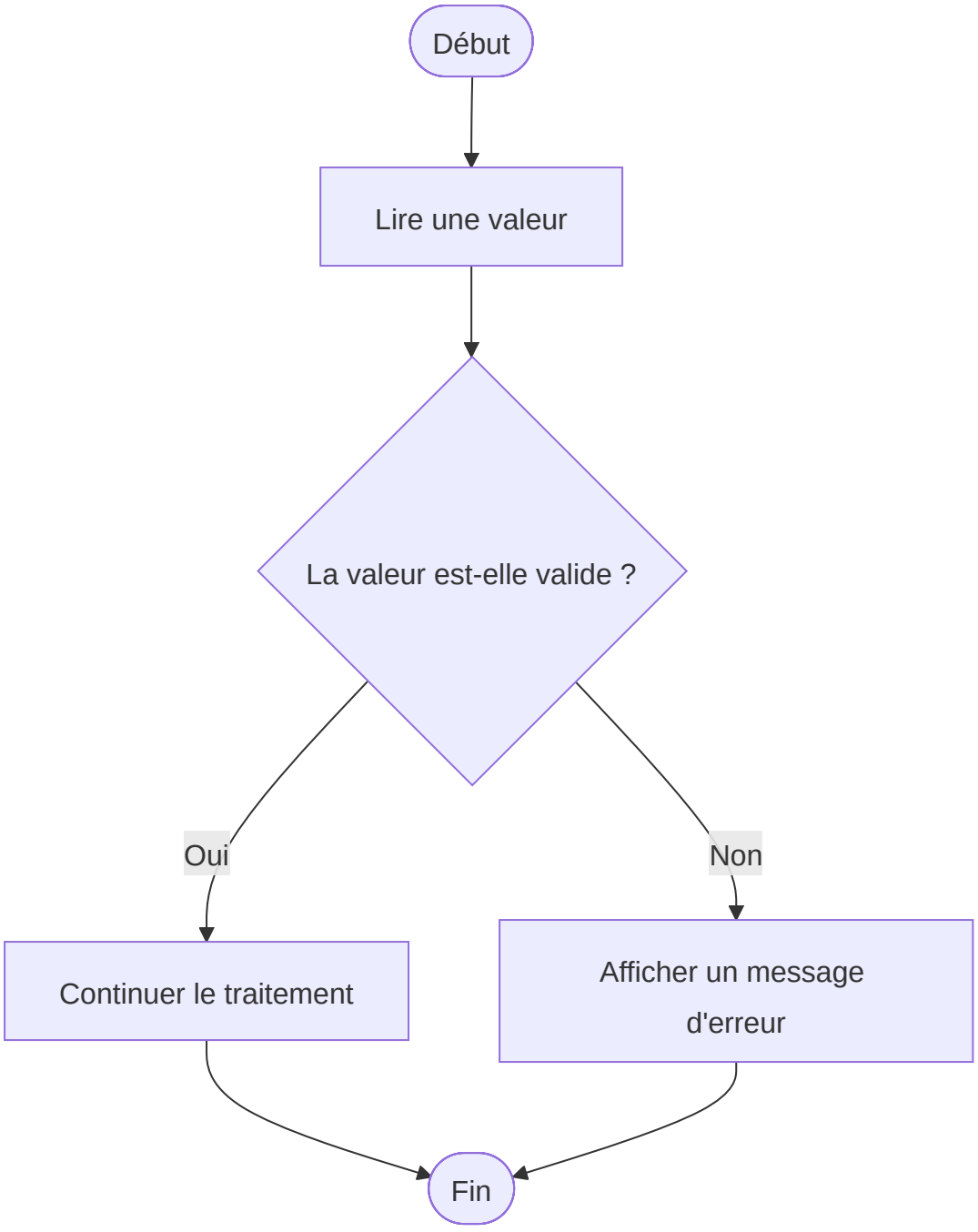
Symboles

Le tableau ci-dessous présente quelques symboles courants de diagramme de flux :

Symbole	Description
	Première et dernière étape du processus.

Symbole	Description
	Représente une action ou une opération du processus.
	Représente un questionnement ou un test dans le processus.
	Informations qui entrent dans le processus ou en sortent.
	Entrée de l'utilisateur.
	Ensemble d’étapes définies plus en détail ailleurs.
	Représente une étape qui découle sur un document (CSV, JSON, TXT).
	L'étape suivante (ou précédente) sur trouvant ailleurs sur le dessin.

Exemple de base

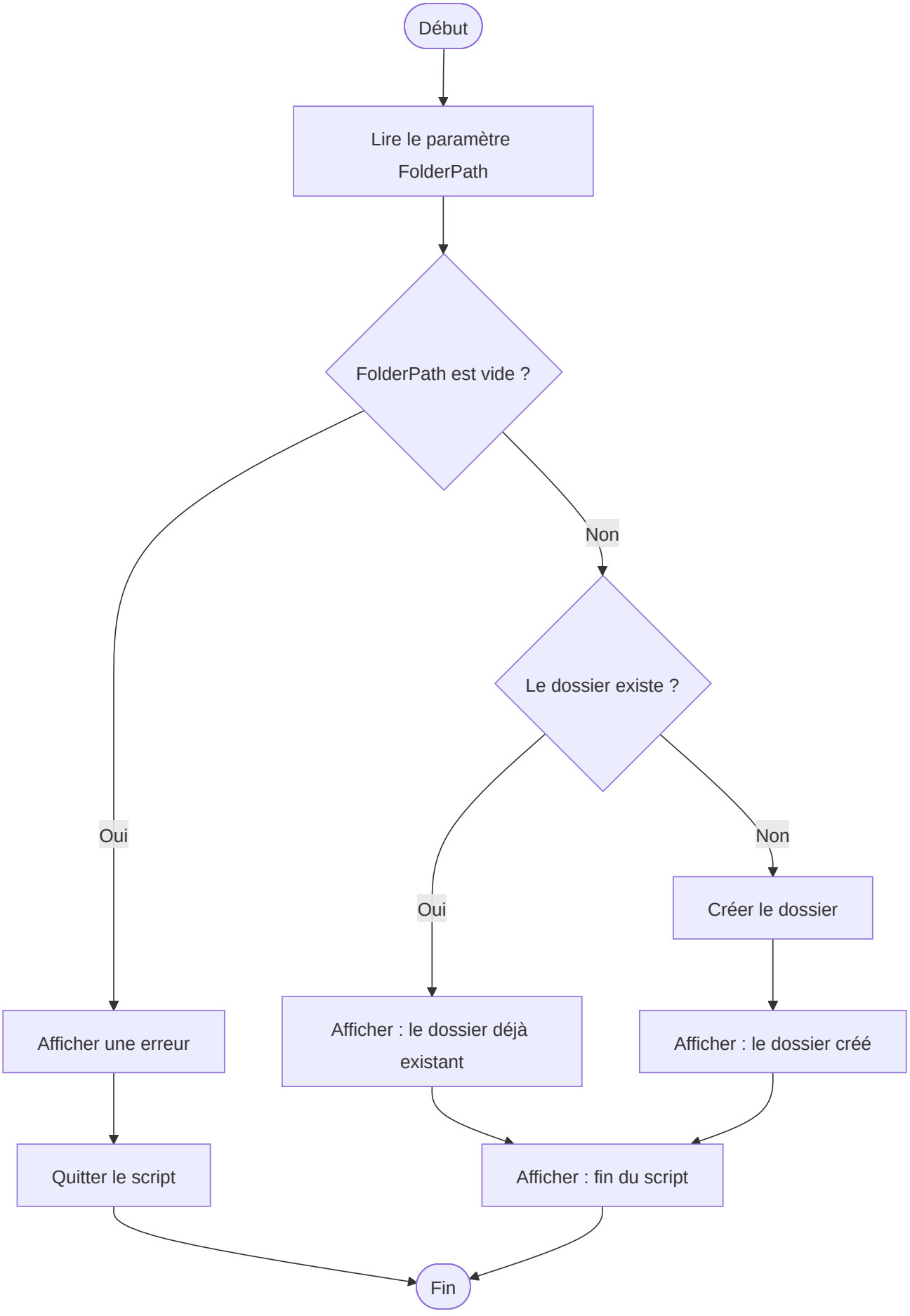


Exemple d'un script

Ce script PowerShell reçoit en paramètre un chemin de dossier (`FolderPath`), vérifie d'abord que ce paramètre a bien été renseigné, puis contrôle si le dossier existe déjà. Si le dossier n'existe pas, il le crée ; sinon, il affiche simplement un message d'information. Enfin, il indique la fin de l'exécution du script.

```powershell
param(
    [string]$FolderPath
)

if ([string]::IsNullOrWhiteSpace($FolderPath)) {
    Write-Error "Le paramètre FolderPath est obligatoire."
    exit 1
}

Write-Host "Vérification du dossier : $FolderPath"

if (Test-Path -Path $FolderPath) {
    Write-Host "[INFO] Le dossier existe déjà."
}
else {
    New-Item -Path $FolderPath -ItemType Directory | Out-Null
    Write-Host "[OK] Le dossier a été créé."
}

Write-Host "Fin du script."
```

Représentation en diagramme de flux du script ci-dessus :


Début
Lire le paramètre FolderPath
FolderPath est vide ?
Oui
Non
Le dossier existe ?
Oui
Non
Afficher une erreur
Afficher : le dossier déjà existant
Créer le dossier
Afficher : le dossier créé
Quitter le script
Afficher : fin du script
Fin

Résumé

- Le diagramme de flux aide à visualiser, communiquer et améliorer un processus.
- Il repose sur des symboles standardisés pour représenter actions et décisions.
- Un exemple concret facilite la compréhension avant l'exercice.

Les Objets avec PowerShell

L'une des grandes forces de PowerShell est sa capacité à manipuler des objets plutôt que du simple texte, ce qui facilite l'analyse, le tri, le filtrage et la réutilisation des données.

Objectif

À la fin de ce chapitre, vous saurez :

- ce qu'est **un objet** en PowerShell (et pourquoi c'est différent du texte)
- inspecter un objet avec `Get-Member`
- lire et manipuler **propriétés** et **méthodes**
- exploiter le pipeline avec `Where-Object`, `Select-Object`, `Sort-Object`, `Group-Object`
- créer vos propres objets (`[pscustomobject]`) et produire une sortie "propre" pour des scripts

---

Le principe clé

PowerShell manipule des **objets**.

Dans beaucoup de shells, la sortie d'une commande est surtout du **texte**. En PowerShell, la sortie est (souvent) une **collection d'objets**.

Un objet = une « chose » avec :

- des **propriétés** (des données) : ex. `Name`, `Id`, `Status`
- des **méthodes** (des actions) : ex. `ToString()`, `Kill()`, ...

Exemple :

```powershell
$p = Get-Process | Select-Object -First 1
$p
```

Info

`$p` n'est pas une ligne de texte. C'est un objet .NET (souvent un `System.Diagnostics.Process`).

`Get-Member` (alias `gm`)

La commande la plus importante pour découvrir les objects retourné par les cmdlets :

```powershell
Get-Process | Select-Object -First 1 | Get-Member
```

Vous obtenez :

- le **TypeName** (le type réel de l'objet)
- la liste des **propriétés** et **méthodes** disponibles

Pour ne voir que les propriétés :

```powershell
Get-Process | Select-Object -First 1 | Get-Member -MemberType Property
```

Accéder aux propriétés - la « syntaxe point »

On lit une propriété d'un objet avec un `.` :

```powershell
$p = Get-Process | Select-Object -First 1
$p.Name
$p.Id
```

Objets en liste (pipeline)

```powershell
Get-Process | Select-Object -First 5 -Property Name, Id
```

Vous remarquez : `Select-Object` **fabrique** un nouvel objet « réduit » qui ne contient que les propriétés demandées, ici `Name` et `Id`.

Appeler des méthodes

Une méthode s'appelle avec des parenthèses :

```powershell
"bonjour".ToUpper()
( Get-Date ).AddDays(7)
```

Beaucoup d'objets ont des méthodes utiles, mais vous n'avez pas besoin de toutes les mémoriser : `Get-Member` vous les montre.

Le pipeline : on filtre, on transforme, on trie

Le pipeline `|` passe des **objets** de commande en commande.

Filtrer avec : `Where-Object`

```powershell
# services en cours d'exécution
Get-Service | Where-Object Status -eq 'Running'

# processus consommant plus de 200 MB (WorkingSet en octets)
Get-Process | Where-Object WorkingSet -gt 200MB
```

> **Info**
>
> La forme « courte » `Where-Object Property -op valeur` marche quand le cas est simple.

Forme *scriptblock* est plus flexible :

```powershell
Get-Process | Where-Object { $_.Name -like 'p*' }
```

Ici `$_` représente **l'objet courant** qui passe dans le pipeline.

Transformer avec : `Select-Object`

```powershell
Get-Process | Select-Object -First 10 Name, Id, CPU
```

Propriété calculée - quand vous voulez une colonne « sur mesure » :

```powershell
Get-Process |
  Select-Object -First 10 Name,
    @{ Name = 'RAM_MB'; Expression = { [math]::Round($_.WorkingSet / 1MB, 1
```

Ici l'on utilise `[math]::Round()`, pour arrondir la valeur retournée.

Trier avec : `Sort-Object`

```powershell
Get-Process | Sort-Object CPU -Descending | Select-Object -First 10 Name, C
```

Grouper avec : `Group-Object`

```powershell
Get-Service | Group-Object Status
```

Important : formatage ≠ données

`Format-Table` / `Format-List` servent à **afficher**. Une fois que vous avez formaté, vous perdez souvent la possibilité de réutiliser l'objet correctement dans le pipeline.

Mauvaise habitude (à éviter) :

```powershell
Get-Process | Format-Table Name, Id | Export-Csv processes.csv
```

Bonne habitude : manipuler les objets d'abord, puis formater seulement à la fin :

```powershell
Get-Process |
  Select-Object Name, Id |
```

```
Export-Csv -NoTypeInformation processes.csv
```

Les objets ont un type (et ça compte)

Vous pouvez vérifier le type comme ceci :

```powershell
$p = Get-Process | Select-Object -First 1
$p.GetType().FullName
```

Créer vos propres objets avec : `[pscustomobject]`

Quand vous écrivez un script, vous voulez souvent produire un résultat structuré (facile à trier, filtrer, exporter).

```powershell
$info = [pscustomobject]@{
  ComputerName = $env:COMPUTERNAME
  UserName     = $env:USERNAME
  When         = Get-Date
}

$info | Format-List
```

Vous pouvez ensuite exporter vers CSV/JSON :

```powershell
$info | ConvertTo-Json
```

Manipulations courantes « style script »

Sélection + export

```powershell
Get-Service |
  Select-Object Name, Status, StartType |
  Export-Csv -NoTypeInformation services.csv
```

Trouver « un seul objet »

```powershell
$svc = Get-Service -Name 'w32time'
$svc.Status
```

Vérifier l'existence (pattern utile)

```powershell
$svc = Get-Service -Name 'w32time' -ErrorAction SilentlyContinue
if ($null -eq $svc) {
  "Service introuvable"
} else {
  $svc.Status
}
```

Exemple plus complet

Objectif : produire un petit « rapport » RAM/CPU des 8 processus les plus gourmands.

```powershell
Get-Process |
  Where-Object { $_.CPU -ne $null } |
  Sort-Object CPU -Descending |
  Select-Object -First 8 Name, Id,
    @{ Name = 'CPU_s';  Expression = { [math]::Round($_.CPU, 1) } },
    @{ Name = 'RAM_MB'; Expression = { [math]::Round($_.WorkingSet / 1MB, 0
```

Vous obtenez une table d'objets que vous pouvez ensuite :

- exporter (`Export-Csv`)
- filtrer à nouveau
- comparer d'un poste à l'autre

Mini-lab (à faire)

1. Liste 10 processus (`Get-Process`) et affiche uniquement `Name`, `Id`, `StartTime` (attention : certains processus n'ont pas de `StartTime` accessible).
2. Fais un tri décroissant sur la RAM.
3. Crée un objet `[pscustomobject]` avec : `When`, `ComputerName`, `TopProcessName`, `TopProcessRAMMB`.
4. Exporte ce résultat en JSON.

Résumer

- PowerShell est un shell orienté **objets**.
- `Get-Member` vous dit tout sur ce que vous manipulez.
- Manipule des objets (`Where-Object`, `Select-Object`, `Sort-Object`) avant d'afficher (`Format-*`).
- `[pscustomobject]` est la base pour produire des résultats propres dans vos scripts.