



Scripting Powershell Partie 2



Plan

- 1 - Wildcards et regex
- 2- Les structures conditionnelles
- 3 - Les structures itératives
- 4 - Les tableaux





Wildcards et regex



Les wildcards

Les **wildcards** (ou **caractères génériques**) sont des symboles qui permettent de remplacer un ou plusieurs caractères dans une chaîne de caractères.



Exemples de wildcards

- * remplace zéro ou plusieurs caractères :

Get-ChildItem C:*.txt

⇒ recherche dans C:\ tous les fichiers txt

- ? remplace exactement un caractère :

Get-ChildItem C:\file?.txt

⇒ recherche tous les fichiers file1.txt, file2.txt, etc., dans C:\



Les regex

Les **regex** (ou **expressions régulières**) en fournissent un moyen puissant de manipuler des chaînes de caractères.

On peut les utiliser avec des opérateurs comme **-match**, **-replace**, et des cmdlets comme **Select-String**.



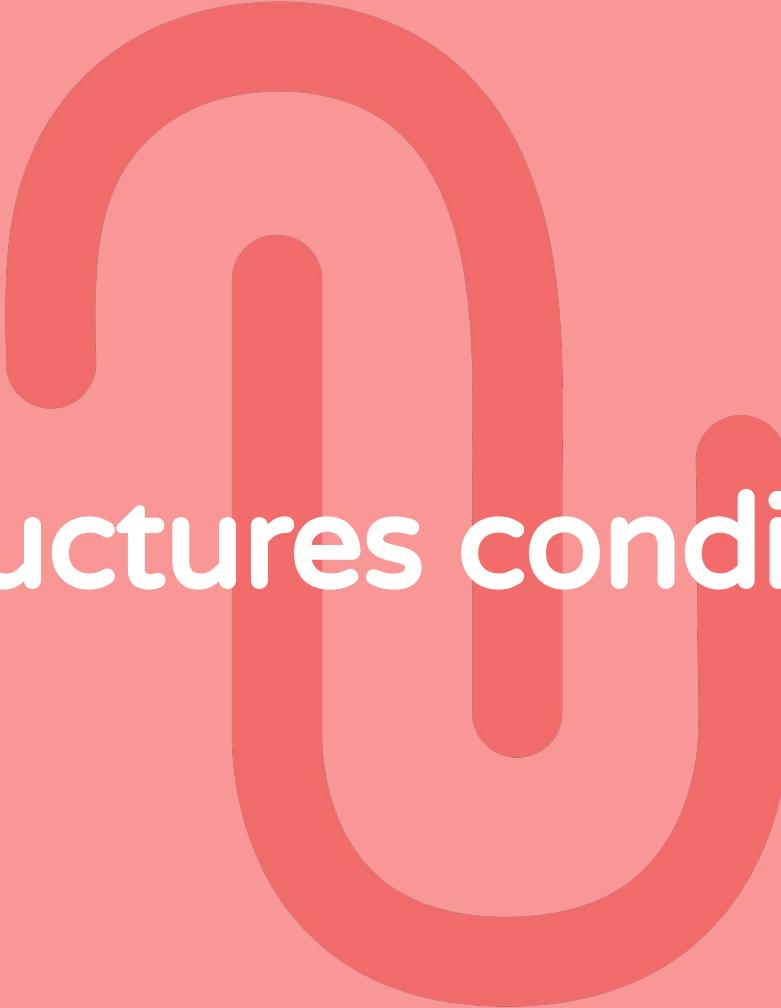
Exemples de regex

```
PS C:\Lab> "123" -match "\d" # \d correspond à tous les chiffres  
True
```

```
PS C:\Lab> "Hello" -match "^H" # ^ correspond à un début de chaîne  
True
```

```
PS C:\Lab> "Hello boy" -replace "(boy|girl)", "everybody"
```

[Autre exemple](#)



Les structures conditionnelles



Rappel du cours Bash

En algorithmique, on appelle **structure conditionnelle**, une construction d'un langage qui permet la création d'instructions optionnelles

C'est à dire de portions de code dont l'exécution va dépendre d'une **condition** (on dit aussi **test**)



Les tests

En PowerShell, un test est :

- **Vrai** s'il vaut **True**
- **Faux** s'il vaut **False**

Ainsi, le code de sortie (**status code**) d'une commande qui a réussi équivaut à vrai et celui d'une commande qui a échouée équivaut à faux.

Les tests se font avec des **opérateurs de comparaison**.



Codes de sortie

Rappel :

\$? permet de récupérer le code de sortie de la dernière commande

```
PS C:\Lab> New-Item -Name NewDir -ItemType Directory
```

Répertoire : C:\Lab

| Mode | LastWriteTime | Length | Name |
|------|------------------|--------|--------|
| -d— | 16/06/2022 10:20 | | NewDir |

```
PS C:\Lab> $?  
True
```

```
PS C:\Lab> New-Item -Name NewDir -ItemType Directory  
New-Item : Il existe déjà un élément avec le nom spécifié  
C:\Lab\NewDir
```

```
PS C:\Lab> $?  
False
```



Les opérateurs de comparaison

-eq (equal to) et **-ne** (not equal to)

-gt (greater than) et **-lt** (less than)

-ge (greater than or equal to) et **-le** (less than or equal to)

-like et **-notLike** (avec les wildcards)

-match et **-notMatch** (avec les expressions régulières)

-not ou ! inverse le code de sortie, (NON logique)



Comparaison de chaînes

Supposons 2 chaînes s1 et s2

s1 -eq s2 : vrai si les chaînes sont identiques

s1 -ne s2 : vrai si les chaînes sont différentes

[String]::IsNullOrEmpty(s1) : vrai si s1 est vide

![String]::IsNullOrEmpty(s1) : vrai si s1 n'est pas vide

Note : Attention aux espaces !

```
PS C:\Lab> 'identique' -eq 'identique'  
True
```

```
PS C:\Lab> 'identique' -eq 'différent'  
False
```

```
PS C:\Lab> 'identique' -ne 'différent'  
True
```

```
PS C:\Lab> [String]::IsNullOrEmpty("")  
True
```

```
PS C:\Lab> "" -eq $Null  
False
```



Comparaison de nombres

Supposons 2 nombres n1 et n2

n1 -eq n2 : vrai si les nombres sont égaux

n1 -ne n2 : faux si les nombres sont différents

n1 -lt n2 : $n1 < n2$

n1 -le n2 : $n1 \leq n2$

n1 -gt n2 : $n1 > n2$

n1 -ge n2 : $n1 \geq n2$

```
PS C:\Lab> $trois = 3  
PS C:\Lab> $trois -eq 3  
True
```

```
PS C:\Lab> 2 -ne $trois  
True
```

```
PS C:\Lab> $deux = 2  
PS C:\Lab> $deux -lt $trois  
True
```



Opérateurs logiques booléens

Supposons c1 et c2 des conditions

! c1 : NON logique (vrai si c1 est faux et vice versa)

c1 -and c2 : ET logique (vrai si c1 et c2 vrai)

c1 -or c2 : OU logique (vrai si l'une des 2 conditions ou les 2 sont vraies)

c1 -xor c2 : OU exclusif logique (vrai si uniquement l'une des 2 conditions est vraies)

```
PS C:\Lab> $trois = 3  
PS C:\Lab> !$trois -eq 3  
False
```

```
PS C:\Lab> 2 -lt $trois -and $trois -lt 4  
True
```

```
PS C:\Lab> $trois -eq 3 -or $trois -eq 4  
True
```

```
PS C:\Lab> $trois -eq 3 -or $trois -lt 4  
True
```

```
PS C:\Lab> $trois -eq 3 -xor $trois -lt 4  
False
```



Opérateurs sur les chemins

Supposons p un chemin/un fichier/un dossier

Test-Path p : vrai si p existe

```
PS C:\Lab> Test-Path -Path C:\Windows  
True
```



Si ... Sinon

Structure conditionnelle if

```
If (condition)
{
    instructions
}
Else
{
    instructions
}
```

```
If (New-Item -ItemType Directory -Name NewDir -ErrorAction SilentlyContinue)
{
    Write-Host "Création dossier succès"
}
else
{
    Write-Host "Création dossier échec" -ForegroundColor Red
}
```

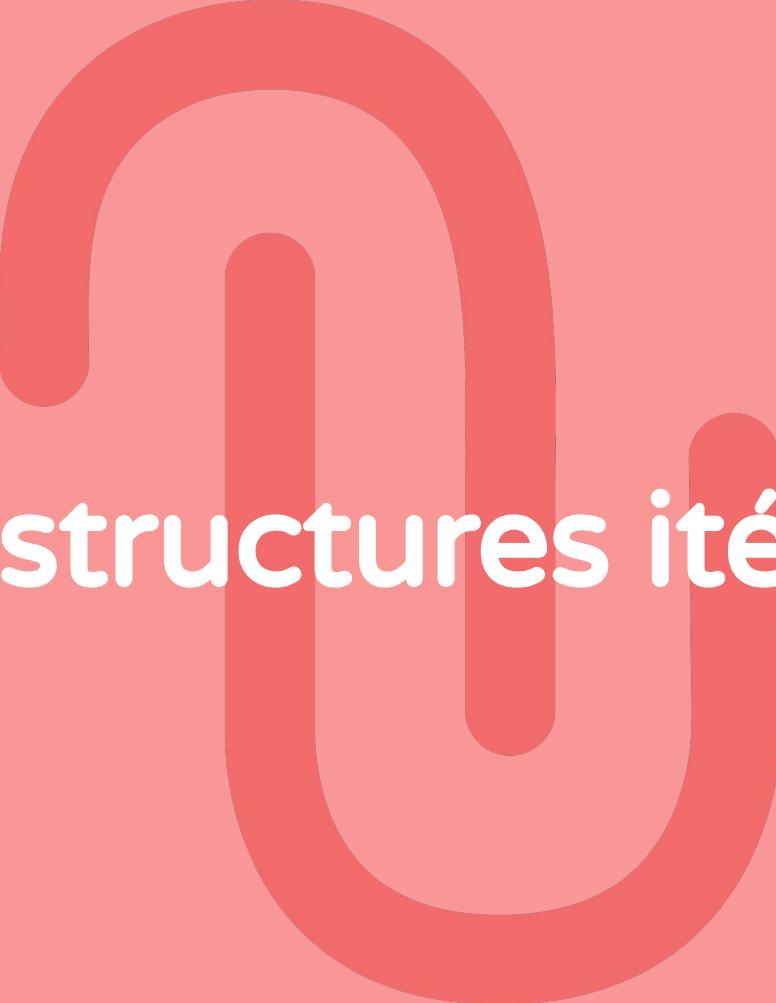


Switch

Structure conditionnelle Switch

```
Switch (condition)
{
    valeur1 {ScriptBlock1}
    valeur2 {ScriptBlock2}
    ...
    default {ScriptBlock par défaut}
}
```

```
$Condition = 5
Switch ($Condition)
{
    1 {Write-Host "hello"}
    2 {Write-Host "2"}
    5 {Write-Host "5"}
    default {Write-Host "default"}
}
```



Les structures itératives



Rappel du cours Bash

En algorithmique, on appelle **structure itérative**, une construction d'un langage qui permet la répétition d'instructions

C'est à dire de portions de code dont l'exécution va être effectuée un nombre de fois donné ou tant qu'une **condition** est remplie.

Il est courant de les qualifier de **boucles**



Les structures itératives

La notion d'unaire

Les opérateurs unaire sont souvent utilisés pour incrémenter des variables dans les boucles.

On utilise **++** et **--** pour incrémenter ou décrémenter une variable de 1

```
PS C:\Lab> $i  
PS C:\Lab> $i = $i + 1  
PS C:\Lab> $i  
1
```

```
PS C:\Lab> $i++  
PS C:\Lab> $i  
2
```

```
PS C:\Lab> $i--  
PS C:\Lab> $i  
1
```



Sens d'incrémantation unaire

Écrire le script ci-dessous et l'exécuter.

Dans le 1^{er} cas, \$i est affiché, puis est incrémenté.

Dans le 2^{ème} cas, \$j est incrémenté, puis affiché.

```
Clear-Host  
$i = 1  
" `\$i vaut $($i++) "  
" `\$i maintenant vaut $i `n"
```

```
$j = 1  
" `\$j vaut $($++$j) "  
" `\$j maintenant vaut $j "
```

```
$i vaut 1  
$i maintenant vaut 2
```

```
$j vaut 2  
$j maintenant vaut 2
```



Boucle For

La **boucle for** est une boucle que l'on initialise et qui a une fin définie.

For (initialisation; condition; mise-à-jour)
{
 bloc d'instructions
}

```
For ($i=0; $i -le 10; $i++)  
{  
    Write-Host "Valeur: $i"  
}
```



Boucle For à conditions multiples

```
For (( $i=0 ), ($j=0) ; $i -le 20 -and ($i+$j) -le 15; $i++, $j++)
{
    Write-Host "Valeur de `\$i: \$i`nValeur de `\$j: \$j`nValeur de `\$i+\$j: $($i+$j)`n"
}
```



Boucle Foreach

La **boucle foreach** est une boucle qui est utilisée pour la manipulation de collection de données ou tableaux. Elle va lire chaque ligne à chaque boucle.

Foreach (element in collection)

```
{  
    bloc d'instructions  
}
```

```
$Services = Get-Service  
foreach ($Service in $Services)  
{  
    Write-Host "$($Service.Name) -->  
$($Service.Status)"  
}
```



Boucle Foreach derrière un pipe

```
Get-Service | ForEach {Write-Host "$($_.Name) --> $($_.Status)"}
```

Foreach est l'alias de **ForEach-Object**

On peut également le remplacer par %



Boucle Foreach (ex avec un switch)

```
$Services = Get-Service
$Count = 1
Foreach ($Service in $Services)
{
    Switch ($Service.Status)
    {
        "Stopped" {Write-Host "$Count - Service: $($Service.Name) ($($Service.DisplayName)) --> $($Service.Status)" -ForegroundColor Red}
        "Running" {Write-Host "$Count - Service: $($Service.Name) ($($Service.DisplayName)) --> $($Service.Status)" -ForegroundColor Green}
        default {Write-Host "$Count - Service: $($Service.Name) ($($Service.DisplayName)) --> $($Service.Status)" -ForegroundColor Blue}
    }
    $Count++
}
```



Boucle While

La boucle **while** exécute le bloc d'instructions **tant que la condition est vérifiée.**

While (condition)

{

bloc d'instructions

}

```
$Count = 0
while ($Count -le 10)
{
    Write-Host "Compteur égal à $Count"
    $Count++
}
```



Boucle Do While

La boucle **do while** est comme la boucle while, sauf que la condition est réalisée à la fin, donc **il y a au moins 1 passage dans la boucle.**

```
Do  
{  
    bloc d'instructions  
}  
While (condition)
```

```
$Count = 0  
  
do  
{  
    Write-Host "Compteur égal à $Count"  
    $Count++  
}  
While ($Count -le 10)
```



Boucle Do Until

La boucle **do until** exécute le bloc de script **jusqu'à** ce que la condition soit réalisée.

```
Do  
{  
    bloc d'instructions  
}  
Until (condition)
```

```
$Count = 0  
do  
{  
    Write-Host "Compteur égal à $Count"  
    $Count++  
}  
Until ($Count -eq 10)
```



Les tableaux



Définition

En programmation, les **tableaux** (ou **collections**) sont des structures de données qui contiennent plusieurs éléments.

Le tableau est créé sous la forme d'un bloc séquentiel de mémoire dans lequel chaque valeur est stockée juste à côté de l'autre.

Pour accéder aux différents éléments, 3 méthodes :

- Avec un index
- Avec une boucle
- Avec une clé



Initialisation de tableau

L'initialisation d'une variable en type tableau change sa structure de données.

Pour un tableau \$Tab

\$Tab = @() : initialisation de tableau

\$Tab = @(valeur1, valeur2,...)

\$Tab = valeur1, valeur2, ...

\$Tab = ValeurInit..ValeurFinale

```
PS C:\Lab> $Tab
PS C:\Lab> $Tab -eq $Null
True
PS C:\Lab> $Tab = @()
PS C:\Lab> $Tab -eq $Null

PS C:\Lab> $Tab.GetType()
IsPublic     IsSerial      Name        BaseType
-----       -----       ----       -----
True         True          Object[]    System.Array

PS C:\Lab> $Tab.Count -gt 0
False
PS C:\Lab> If ($Tab2 -eq $Null) { "Oui" } else { "Non" }
Oui
PS C:\Lab> 1..5
1
2
...
```



Mon premier tableau

Écrire le bloc d'instruction,
l'exécuter, et écrire les
commandes en console.

Pour un tableau \$Tab
\$Tab.count : nombre d'éléments

```
$Tab = @("Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi", "Samedi", "Dimanche")
```

```
PS C:\Lab> $Tab.Count
7
PS C:\Lab> $Tab
Lundi
Mardi
Mercredi
Jeudi
Vendredi
Samedi
Dimanche
```



Tableaux : recherche par index

Repartons du \$Tab précédent.
Exécuter les commandes en
console.

Pour un tableau \$Tab
\$Tab[n] : n ième élément
Attention n commence à 0

```
$Tab = @("Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi", "Samedi", "Dimanche")
```

```
PS C:\Lab> $Tab[0]
```

Lundi

```
PS C:\Lab> $Tab[1,3,5]
```

Mardi

Jeudi

Samedi

```
PS C:\Lab> $Tab[-1]
```

Dimanche

```
PS C:\Lab> $Tab+="JourEnPlus"
```

```
PS C:\Lab> $Tab[-1]
```

JourEnPlus



Tableaux : recherche par boucle

Repartons du \$Tab précédent.
La boucle Foreach est souvent utilisée dans l'exploitation des tableaux.

```
$Tab = @("Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi", "Samedi", "Dimanche")
$Count = 1
Foreach ($Date in $Tab)
{
    Write-Host "Jour N°$Count de la semaine : $Date"
    $Count++
}
```



Tableaux : recherche par clé

Les tableaux avec des clés sont appelés **table de hachage** (ou dictionnaire ou tableau associatif). Ce sont des structures de données qui stockent une ou plusieurs paires clé/valeur.

@{clé1=valeur1;clé2=valeur2;...}

```
$HashTable = @{1 = "Lundi";2 = "Mardi";3 = "Mercredi";4 = "Jeudi";5 = "Vendredi";6 = "Samedi";7 = "Dimanche"}
```

```
PS C:\Lab> $HashTable.Count  
7
```

```
PS C:\Lab> $HashTable[2]  
Mardi
```

```
PS C:\Lab> $HashTable.Keys
```

```
PS C:\Lab> $HashTable.Values
```

```
PS C:\Lab> $HashTable.Add("8","Jour d'après")
```

```
PS C:\Lab> $HashTable.GetEnumerator()
```



En conclusion

- Regex
- Si...Sinon et Switch
- Les différentes boucles
- Le système des tableaux

