WILD CODE SCHOOL

# Scripting Powershell
# Partie 3

# Plan

1 - Les fonctions

2 - Remote PowerShell

3 - Exécution parallèle

WILD CODE SCHOOL

# Les fonctions

# Rappel du cours Bash

Une **fonction** est un bloc de code nommé qu'on déclare pour pouvoir l'utiliser plus tard, éventuellement plusieurs fois

Elles permettent de structurer son code, de favoriser sa réutilisation, sa maintenance, etc.

# Les bases

Déclaration de fonction

```
function nom
{
    instructions
}
```

Attention : les fonctions doivent être déclarée avant d'être appelée

```
Clear-Host
function Hello
{
   Write-Host "Hi folks !"
}
Hello
Write-Host "and again"
Hello
```

```
Hi folks !
and again
Hi folks !
```

# Fonctions et paramètres

Un appel de fonction peut être suivi d'arguments

On les récupère dans la fonction comme les paramètres d'un script

**$args** contient tous les arguments du script
**$args[n]** est le n$^{ième+1}$ argument
**n** commence à 0

```
function Hello
{
   Write-Host "Hi folks !"
}
function greet
{
   param ([Array]$ArgumentsList)
   If ($ArgumentsList.Count -gt 0)
   {
       Write-Host "Hi $($ArgumentsList[0])"
   }
   else
   {
       Hello
   }
}
greet -ArgumentsList $args
greet
```

# Fonctions avancées

Pour aller plus loin

On peut utiliser des attributs de paramètres pour améliorer une fonction.
Quelques exemples:
**[ValidateSet('VALUE1', 'VALUE2')]** : liste de choix
**[Parameter(Mandatory=$false)]** détermine si le paramètre est nécessaire
**[ValidateRange(MIN, MAX)]** permet de n'accepter qu'un entier situer entre MIN et MAX

```
function Conversion
{
    param ( [Parameter(Mandatory=$True)]
            [ValidateRange(0, 255)]
            [Int32]$Number,

            [Parameter(Mandatory=$True)]
            [ValidateSet('Binaire', 'octal')]
            [String]$Calcul )

    Switch ($Calcul)
    {
        'Binaire'
{[convert]::ToString([int]$Number,2)}
        'Octal'
{[Convert]::ToString([int]$Number,8)}
    }
}
```

WILD CODE SCHOOL

# Remote PowerShell

# Powershell à distance

On peut exécuter du PowerShell à distance de différentes manières :

- Cmdlet de commandes à distance
- Session interactive à distance
- Exécution de commandes à distance

# Cmdlet de commandes à distance

Quelques cmdlet possèdent le paramètre ComputerName :

**Restart-Computer**
**Test-Connection**
**Clear-EventLog**
**Get-EventLog**
**Get-HotFix**
**Get-Process**
**Get-Service**
**Set-Service**
**Get-WinEvent**
**Get-WmiObject**

```
PS C:\Lab> Stop-Computer -ComputerName client1

PS C:\Lab>  Test-Connection -ComputerName client2

Source     Destination     IPV4Address      IPV6Address      Bytes    Time(ms)
------     -----------     -----------      -----------      -----    --------
DC1          client2       172.16.1.101                       32       0
DC1          client2       172.16.1.101                       32       0
DC1          client2       172.16.1.101                       32       0
DC1          client2       172.16.1.101                       32       0
```

# Session interactive à distance

Une exécution de code à distance peut se faire avec le cmdlet **Enter-PSSession**.
Le service **WinRM** doit être démarré sur l'ordinateur distant.
**Exit-PSSession** clos une session distante.

```
PS C:\Lab>  Enter-PSSession -ComputerName client1

[client1]: PS C:\Users\administrator\Documents> Set-Location -path c:\

[client1]: PS C:\> New-Item -Path C:\ -ItemType Directory -Name "00_test"

Répertoire : C:\

Mode        LastWriteTime          Length      Name
----        -------------          ------      ----
d-----      21/06/2022     00:17               00_test

[client1]: PS C:\> Exit-PSSession

PS C:\Lab>
```

# Exécution de commandes à distance

Le cmdlet **Invoke-Command** permet d'exécuter du code à distance.

```
PS C:\Lab>  PS C:\Users\Administrator> Invoke-Command -ComputerName client1
-ScriptBlock {Get-ChildItem -Path C:\}


    Directory: C:\


Mode        LastWriteTime         Length Name                 PSComputerName
----        -------------         ------ ----                 --------------
d-----      21/06/2022    00:17          00_test              client1
d-----      07/12/2019    10:14          PerfLogs             client1
d-r---      07/03/2022    22:41          Program Files        client1
d-r---      06/10/2021    15:36          Program Files (x86)  client1
d-r---      08/03/2022    14:10          Users                client1
d-----      07/03/2022    22:26          Windows              client1
```

WILD CODE SCHOOL

# Exécution parallèle

# Définition

L'exécution en parallèle, une capacité clé dans la programmation, permet l'exécution simultanée de multiples tâches pour améliorer les performances et l'efficacité.
En PowerShell, cette fonctionnalité est mise en œuvre à travers des méthodes telles que :

- Les jobs
- Les runspaces
- Les workflows
- L'exécution parallèle dans la boucle Foreach

# Jobs

- Pour exécuter des commandes ou des scripts dans des processus distincts de manière asynchrone, sans bloquer l'exécution du script principal
- Utiles pour des taches longues ou intensives en ressources, comme des appels à des services distants ou des traitements de fichiers volumineux

# Runspaces

- Pour contrôler finement la gestion des threads et des ressources
- Mieux que les jobs pour les tâches légères ou pour les scripts nécessitant un parallélisme plus finement réglé

# Workflow

- Pour définir des processus de travail structurés impliquant des étapes qui peuvent être exécutées en parallèle ou séquentiellement
- Utiles pour l'automatisation de processus métier complexes impliquant des interactions entre plusieurs systèmes ou services

# Parallel.ForEach

- Pour traiter des éléments d'une collection en parallèle de manière simple et efficace
- Utile pour des tâches de traitement de données dans des tableaux ou la manipulation d'objets dans des listes où chaque élément peut être traité de manière indépendante

# Détails sur les commandes des jobs

**Get-Job** : Obtient la liste des jobs existants
**Receive-Job** : Récupère le résultat d'un job
**Remove-Job** : Supprime un job
**Start-Job** : Débute l'exécution d'un job
**Stop-Job** : Arrête un job en cours d'exécution, passe en "Stopped"
**Wait-Job** : Attend la fin d'exécution d'un job
**Suspend-Job** : Suspend l'exécution d'un job, passe en "Suspended"
**Resume-Job** : Reprend l'exécution d'un job suspendu, passe en "Running"

# Exécution séquentielle

```
Write-Host "Début du script : $(Get-Date)"
$StartJobTime1 = Get-Date
Start-Sleep -Seconds 10
Write-Host "Job1 : $StartJobTime1 --> $(Get-Date)"
$StartJobTime2 = Get-Date
Start-Sleep -Seconds 10
Write-Host "Job2 : $StartJobTime2 --> $(Get-Date)"
$StartJobTime3 = Get-Date
Start-Sleep -Seconds 10
Write-Host "Job3 : $StartJobTime3 --> $(Get-Date)"
Write-Host "Fin du script : $(Get-Date)"
```

# Exécution parallèle avec les jobs

```
Write-Host "Début du script : $(Get-Date)"
$Job1 = Start-Job -ScriptBlock {
    $Start = Get-Date
    Start-Sleep -Seconds 10
    $End = Get-Date
    Write-Host "Job 1 : $Start --> $End"
}
$Job2 = Start-Job -ScriptBlock {
    $Start = Get-Date
    Start-Sleep -Seconds 10
    $End = Get-Date
    Write-Host "Job 2 : $Start --> $End"
}
$Job3 = Start-Job -ScriptBlock {
    $Start = Get-Date
    Start-Sleep -Seconds 10
    $End = Get-Date
    Write-Host "Job 3 : $Start --> $End"
}
```

```
Get-Job | Wait-Job

Receive-Job $Job1
Receive-Job $Job2
Receive-Job $Job3

Remove-Job -Id $Job1.Id
Remove-Job -Id $Job2.Id
Remove-Job -Id $Job3.Id

Write-Host "Fin du script : $(Get-Date)"
```

# En conclusion

- Les fonctions et leurs paramètres
- Le PowerShell à distance
- La notion de job