

# POWERSHELL INITIATION

<b>1) INTRODUCTION A POWERSHELL .....</b>	<b>4</b>
A. QU'EST CE QU'UN SCRIPT ? .....	4
B. POURQUOI UN SCRIPT ? .....	4
C. HISTORIQUE.....	4
D. POWERSHELL : ORIENTE OBJETS .....	5
<i>Exemple 1 : un camion.....</i>	<i>5</i>
<i>Exemple 2 : les variables sous MS-DOS et sous Powershell.....</i>	<i>5</i>
<i>Exemple 3 : traitement du résultat d'une commande .....</i>	<i>6</i>
<b>2) PRISE EN MAIN DE POWERSHELL .....</b>	<b>7</b>
A. INSTALLATION .....	7
B. LA CONSOLE.....	7
C. CONSOLE ISE .....	8
D. COMMANDES POWERSHELL .....	11
E. QUELQUES COMMANDES UTILES .....	11
<i>Get-Command .....</i>	<i>11</i>
<i>Get-Alias .....</i>	<i>11</i>
<i>Get-Member .....</i>	<i>11</i>
<i>Get-Help .....</i>	<i>11</i>
<i>Mise à jour de l'aide .....</i>	<i>12</i>
F. EXERCICE 1 .....	12
G. EXERCICE 2 .....	13
<b>3) GESTION DES FICHIERS ET DES REPERTOIRES .....</b>	<b>14</b>
A. GET-CHILDITEM .....	14
<i>Exercice 3.....</i>	<i>15</i>
B. WHERE-OBJECT.....	15
<i>Rappel : Get-Member .....</i>	<i>16</i>
<i>Opérateurs de comparaison .....</i>	<i>16</i>
<i>Exercice 4.....</i>	<i>16</i>
C. NEW-ITEM .....	16
<i>Exercice 5.....</i>	<i>17</i>
D. COPY-ITEM.....	17
<i>Exercice 6.....</i>	<i>17</i>
<i>Conclusion .....</i>	<i>18</i>
<b>4) BASES DU SCRIPTING .....</b>	<b>19</b>
A. TEST-PATH, IF-ELSE, WRITE-HOST ET VARIABLES.....	20
<i>Test-Path .....</i>	<i>20</i>
<i>Exercice 7.....</i>	<i>21</i>
<i>If-Else .....</i>	<i>21</i>
<i>Exercice 8.....</i>	<i>22</i>
<i>Variables.....</i>	<i>22</i>
<i>Exercice 9.....</i>	<i>23</i>
B. IF-ELSE IMBRIQUES.....	23
<i>Exercice 10.....</i>	<i>24</i>
<i>Exercice 11.....</i>	<i>25</i>
C. LA NON-CONDITION (DOUBLE NEGATIF = POSITIF).....	25
<i>Exercice 12.....</i>	<i>26</i>
D. VARIABLES INT, TABLEAU ET BOUCLE WHILE .....	26

<i>While</i> .....	26
<i>Tableau</i> .....	27
<i>Exercice 13</i> .....	27
<i>Intégration du tableau dans la boucle While</i> .....	27
<i>Int comme un compteur</i> .....	28
<i>Exercice 14</i> .....	28
<i>Exercice 15</i> .....	29
<i>Exercice 16</i> .....	29
E. LES FONCTIONS.....	29
<i>Exercice 17</i> .....	29

# 1) Introduction à Powershell

## a. Qu'est ce qu'un script ?

Un script = une suite de commandes, d'instructions.

Architecture d'un script = organigramme, structure conditionnelle, boucle, fonction, gestion des erreurs.

## b. Pourquoi un script ?

Pour automatiser des tâches fastidieuses et répétitives.

Les + = exécutions identiques, pas d'erreurs humaines ...

Les - = programmation, tests, débogage, syntaxe ...

## c. Historique

Microsoft	Unix/Linux	Autres
	1968 : 1ers environnements d'exécutions de scripts (natifs systèmes) – Bourne Shell, C Shell, Korn Shell, Bash Shell, ...	
1981 : scripts batch (.bat) avec MS-DOS 1.0. Tâches limitées.		
	1987 : PERL pour Unix (objectifs : manipulation de fichiers, de données, de processus)	
		1995 : Javascript (NetScape)
1996 : JScript (très ressemblant à JavaScript)		
1997 : PERL intégré dans un kit de ressources pour NT 4.0 + kiXtart		1997 : Normalisation de JavaScript et JScript = VBScript
1998 : Windows Script Host : environnement de scripts VBScript ; association avec kiXtart		
2006 : Powershell v1.0. Les systèmes Microsoft sont depuis orientés lignes de commandes pour l'administration systèmes. Interfaces graphiques construites sur une « couche Powershell.		
Aujourd'hui V7.5.0 de Powershell. Produits Microsoft et tiers construits dessus et disposant de leurs propres jeux de commandes.		

## d. Powershell : orienté objets

### Présentation de PowerShell

PowerShell est à la fois un interpréteur de commandes et un puissant langage de scripts. Il tire sa puissance en grande partie grâce au Framework .NET sur lequel il s'appuie. Bien connu des développeurs, le Framework .NET l'est en revanche beaucoup moins des administrateurs système et autres développeurs de scripts ; ce qui est normal... Pour vulgariser en quelques mots, le Framework .NET est une immense bibliothèque de classes à partir desquelles nous ferons naître des objets ; objets qui nous permettront d'agir sur l'ensemble du système d'exploitation en un minimum d'effort. Tous ceux qui ont goûté à la puissance du Framework .NET ne tariront pas d'éloges à son égard. C'est donc grâce à ce dernier que PowerShell tire toute son intelligence ainsi que sa dimension objet. Et c'est d'ailleurs cette faculté à manipuler les objets qui fait de PowerShell un shell d'exception !

La nouveauté, c'est que Powershell est orienté objet, et permet l'accès aux objets .NET, COM et WMI (on verra les types d'objets plus tard).

### Objet = Type – Méthodes – Propriétés

#### Exemple 1 : un camion

Le camion est un objet unique.

Il fait partie de la catégorie des véhicules : c'est le type.

On peut faire faire des actions au camion : avancer, reculer, tourner à droite, etc ... : ce sont les méthodes.

Enfin, on peut obtenir plein d'informations sur ce camion : son modèle, sa couleur, son niveau d'essence actuel, etc ... : ce sont les propriétés.

#### Exemple 2 : les variables sous MS-DOS et sous Powershell

Sous MS-DOS, créons une variable, appelée A, qui contient le texte Bonjour :

```
C:\Users\Administrateur>set A = Bonjour
C:\Users\Administrateur>set A
A = Bonjour
ALLUSERSPROFILE=C:\ProgramData
APPDATA=C:\Users\Administrateur\AppData\Roaming
```

En l'état, à part afficher son contenu sous format texte, je ne peux pas en faire grand-chose.

La même chose sous Powershell :

```

PS C:\Users\Administrateur> $A = 'Bonjour'
PS C:\Users\Administrateur>
PS C:\Users\Administrateur> $A.ToUpper()
BONJOUR
PS C:\Users\Administrateur>
PS C:\Users\Administrateur> $A.ToLower()
bonjour
PS C:\Users\Administrateur>
PS C:\Users\Administrateur> $A.length
7

```

La variable qu'on vient de créer (**\$A = 'Bonjour'**) est un objet. Comme c'est un objet, il dispose de méthodes et de propriétés.

**\$A.ToUpper()** renvoie comme résultat BONJOUR. ToUpper() est une méthode, on a fait faire à l'objet un passage en majuscules.

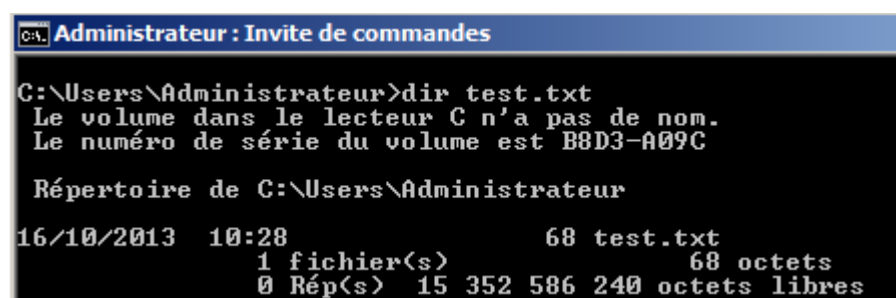
Même chose pour **\$A.ToLower()**, qui renvoie comme résultats bonjour. ToLower() est une méthode, on a fait faire à l'objet un passage en minuscules.

**\$A.length** renvoie 7 (nombre de caractères) ; length est une propriété de l'objet.

### Exemple 3 : traitement du résultat d'une commande

On souhaite obtenir la taille d'un fichier test.txt.

Sous MS-DOS, la commande DIR permet d'afficher un résultat :



```

C:\Users\Administrateur>dir test.txt
Le volume dans le lecteur C n'a pas de nom.
Le numéro de série du volume est B8D3-A09C

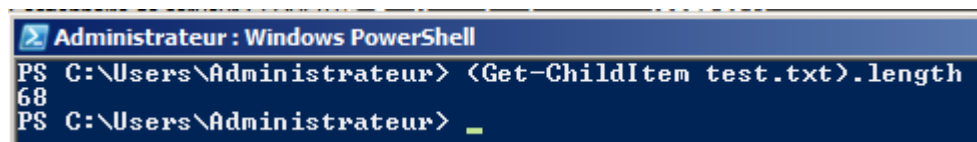
Répertoire de C:\Users\Administrateur

16/10/2013  10:28                68 test.txt
               1 fichier(s)                68 octets
               0 Rép(s)  15 352 586 240 octets libres

```

Par défaut, la commande affiche tout un tas d'information, et c'est à nous d'aller récupérer celle qui nous intéresse (le fichier fait 68 octets).

Sous Powershell, le résultat des commandes est un objet ; comme tout objet, il dispose de méthodes et propriétés et on peut donc afficher directement et utiliser ce qui nous intéresse :



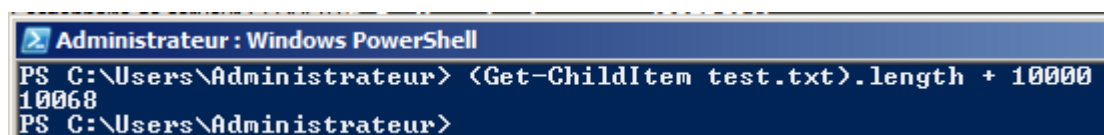
```

PS C:\Users\Administrateur> <Get-ChildItem test.txt>.length
68
PS C:\Users\Administrateur>

```

On a directement le résultat : 68.

Ce résultat étant un objet, je peux le combiner à d'autres fonctions si j'en ai besoin :



```

PS C:\Users\Administrateur> <Get-ChildItem test.txt>.length + 10000
10068
PS C:\Users\Administrateur>

```

La commande **Get-Member** permet de connaître le type d'objet, ainsi que les méthodes et propriétés de cet objet. Par exemple, la commande suivante permet d'afficher toutes les propriétés dont on dispose sur l'objet créé avec la résultat de la commande **Get-ChildItem test.txt** :

```

Administrateur : Windows PowerShell
PS C:\Users\Administrateur> (Get-ChildItem test.txt) | get-member -membertype property

TypeName : System.IO.FileInfo

Name      MemberType Definition
-----
Attributes Property  System.IO.FileAttributes Attributes {get;set;}
CreationTime Property  datetime CreationTime {get;set;}
CreationTimeUtc Property  datetime CreationTimeUtc {get;set;}
Directory  Property  System.IO.DirectoryInfo Directory {get;}
DirectoryName Property  string DirectoryName {get;}
Exists     Property  bool Exists {get;}
Extension  Property  string Extension {get;}
FullName   Property  string FullName {get;}
IsReadOnly Property  bool IsReadOnly {get;set;}
LastAccessTime Property  datetime LastAccessTime {get;set;}
LastAccessTimeUtc Property  datetime LastAccessTimeUtc {get;set;}
LastWriteTime Property  datetime LastWriteTime {get;set;}
LastWriteTimeUtc Property  datetime LastWriteTimeUtc {get;set;}
Length     Property  long Length {get;}
Name       Property  string Name {get;}

```

## 2) Prise en main de Powershell

### a. Installation

Vous n'avez rien à faire si vous travaillez sur Windows 8 ou Windows 2012 Server ; Powershell 3.0 est directement intégré.

Si vous êtes sous Windows 7 ou Windows 2008 R2, vous disposez de la version 2 de Powershell. La version 3 compte environ 10 fois plus de commandes que la version 2, alors autant utiliser la V7.5.0

Sur un Windows 2008 R2 :

- Installez la fonctionnalité Windows Powershell Integrated Scripting Environment (ISE), sinon elle ne sera pas mise à jour par la suite.
- Installer la version 4.0 du Framework .NET.
- Installer Windows Management Framework 3.0.

### b. La console

Quasi identique à celle de MS-DOS (hormis le fond bleu).

Prend en charge la complétion automatique avec la touche TAB.

Quelques autres touches pratiques :

- Echap : pour effacer la ligne de commande en cours
- F7 : pour afficher l'historique des commandes effectuées, puis sélection avec les touches flèche haut/bas ou avec F9 + n° de la commande.

```

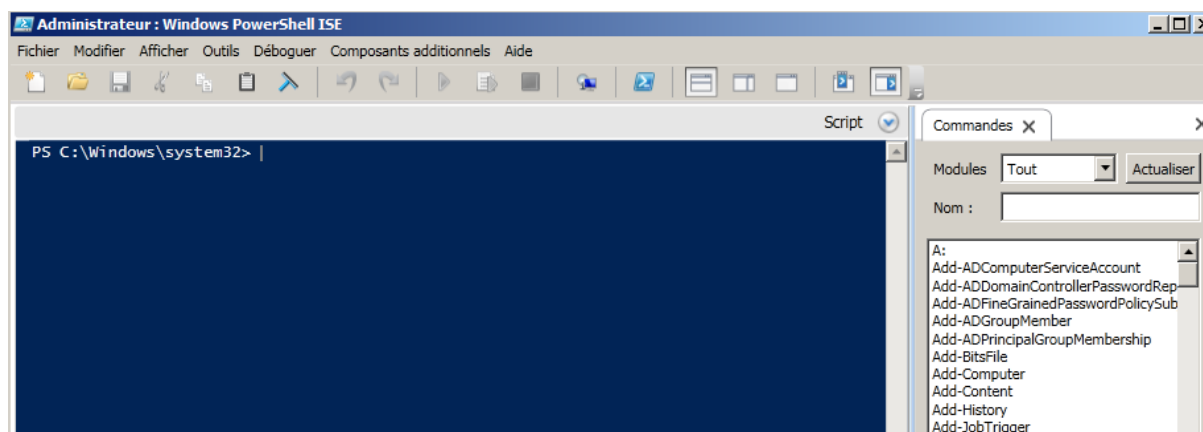
lastWriteTime (get;set,?)
3: $A.length
9: get-childitem test.txt ! $_.length
10: (Get-ChildItem test.txt).length
11: g
12: { Numéro de commande : h
13: { length
14: cls
15: (Get-ChildItem test.txt).length
16: (Get-ChildItem test.txt).length + 10
17: cls

```

- CTRL + C pour quitter l'instruction en cours

### c. Console ISE

De base, on se retrouve avec la console Powershell, agrémentée d'un volet à droite où sont récapitulées toutes les commandes disponibles :

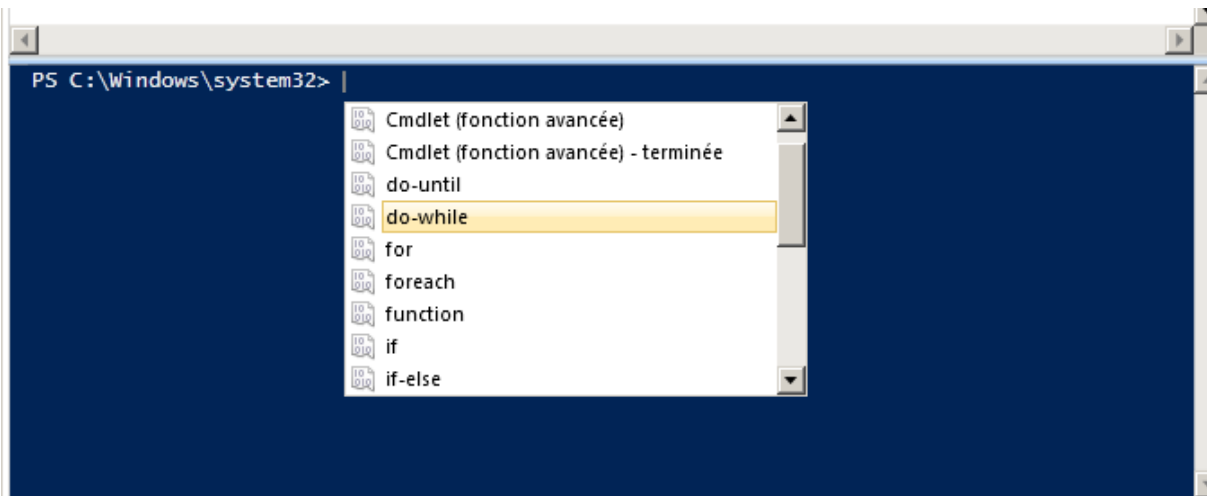


Le champ Nom permet de filtrer les commandes selon ce qu'on recherche (liste toutes les commandes contenant le mot service par exemple).





A noter que la combinaison de touches CTRL + J permet d'intégrer directement une structure de code :



---

## d. Commandes Powershell

---

Les commandes par défaut de Powershell sont appelées **cmdlets** (prononcer commandelettes) et sont enrichies à chaque fois qu'un nouveau module est installé (Active Directory, Exchange, SharePoint, Hyper-V, etc ...).

Elles disposent d'alias qui permettent, outre de raccourcir la commande, de faire le lien avec le monde MS-DOS et Linux. Ainsi, la commande DIR affiche la liste des fichiers et dossiers du répertoire en cours, tout comme la commande LS. DIR et LS sont des alias vers la « vraie » commande Powershell : Get-ChildItem.

Les commandes Powershell sont toutes structurées de la même façon : verbe-nom.

Verbe	Nom
Get	Command
Set	Alias
Add	Item
Remove	Object
Convert	User
Enable	Computer
Disable	Account
...	Path
	Variable
	...

Toutes les combinaisons verbe-nom ne sont bien évidemment pas possible, mais cela permet quand même de mémoriser plus facilement les principales commandes au fur et à mesure.

---

## e. Quelques commandes utiles

---

### Get-Command

Affiche la liste de toutes les cmdlets (+ alias et fonctions Powershell).

### Get-Alias

Affiche la liste de tous les alias

### Get-Member

Affiche les méthodes et propriétés d'un objet (cf. p6 pour exemple)

### Get-Help

Quasiment la commande la plus importante, elle peut être utilisée de plusieurs sortes.

**Get-Help laCommande** : affichera un aide synthétique.

**Get-Help laCommande -detailed** : affichera une aide détaillée

**Get-Help laCommande -examples** : affichera des exemples d'utilisation

On peut obtenir de l'aide via d'autres syntaxes :

**Help** laCommande

laCommande **-help**

### Mise à jour de l'aide

L'aide complète n'est disponible qu'en anglais (sauf si vous êtes sur un Windows 2012 Server). Toutes les commandes ne disposent donc pas d'une aide complète.

Deux solutions pour pallier à ce problème :

- Get-Help laCommande -online : vous amène sur le site de Microsoft, directement à la rubrique d'aide complète (en français)
- Ou l'utilisation de l'aide en anglais (un script est à exécuter pour forcer cette utilisation – demandez à votre formateur).

## f. Exercice 1

A l'aide de la commande Get-Alias, trouvez quelles sont les commandes Powershell les alias indiqués dans le tableau. Essayez d'utiliser les arguments de la commande Get-Alias.

CD	
DIR	
MD	
REN	
COPY	
LS	
CLS	
ECHO	
KILL	
MAN	
MOUNT	
PS	
PWD	
START	

## **g. Exercice 2**

---

Avec Get-Command, indiquez la commande pour afficher :

- Uniquement les cmdlets Powershell :
- Uniquement les commandes qui contiennent le mot New :
- Uniquement les commandes du module Active Directory :

### 3) Gestion des fichiers et des répertoires

Set-Location	Se positionner dans l'arborescence de fichiers
Get-ChildItem	Affiche les fichiers et les dossiers présents dans le système de fichiers
New-Item	Création de fichiers et de dossiers
Copy-Item	Copie de fichiers et de dossiers
Move-Item	Déplacement de fichiers et de dossiers
Rename-Item	Renommer un fichier et un dossier
Remove-Item	Supprimer des fichiers et des dossiers
(le pipe)	Permet de connecter des commandes ; la sortie d'une commande devient l'entrée d'une autre commande
Where-Object	Filtre qui s'utilise toujours avec le pipe ; utilise le résultat de la commande précédente et filtre le résultat selon ce qui nous intéresse

#### a. Get-ChildItem

Cette cmdlet affiche les fichiers et les dossiers présents dans le système de fichiers :

```

Administrateur : Windows PowerShell
PS C:\> Get-ChildItem

Répertoire : C:\

Mode                LastWriteTime         Length Name
----                -
d-----         14/07/2009         05:20      PerfLogs
d-r--         14/10/2013         14:47      Program Files
d-r--         17/10/2013         10:49      Program Files (x86)
d-----         14/10/2013         16:20      test
d-r--         14/10/2013         14:05      Users
d-----         17/10/2013         13:40      Windows
  
```

5 attributs existents :

- **D** : pour Directory
- **A** : pour Archive
- **R** : pour ReadOnly
- **H** : pour Hidden
- **S** : pour System

Il est possible de n'afficher les fichiers et dossiers qu'en fonction de leur attribut :

**Get-ChildItem -Attributes Directory** : n'affiche que les répertoires

**Get-ChildItem -Attributes System** : n'affiche que les fichiers systèmes

Il est possible de combiner les attributs avec des opérations logiques :

- **+** : est l'équivalent du ET logique
- **,** : est l'équivalent du OU logique
- **!** = est l'équivalent de la négation

Get-ChildItem -Attributes !Directory : n'affiche que les fichiers, pas les répertoires (utilisation de la négation !).

Get-ChildItem -Attributes !Directory+System : n'affiche que les fichiers systèmes (utilisation de la négation ! et du ET logique + ; les 2 conditions doivent être respectées)

### Exercice 3

---

Quelle commande permet de :

- Se positionner dans le répertoire c:\Windows\ :
- N'affichez que les répertoires cachés :
- N'affichez que les répertoires ou les fichiers systèmes :
- N'affichez que les fichiers cachés (et pas les répertoires) :

### b. Where-Object

---

La commande **Where-Object** est un filtre, qui ne s'utilise qu'avec le pipe, et qui permet de filtrer les résultats selon les valeurs d'un attribut. Elle utilise une syntaxe précise :

**Get-ChildItem | Where-Object {\$\_.Length -gt 2KB}**

**Get-ChildItem** est la première commande

| permet d'utiliser la sortie de la première commande comme entrée de Where-Object.

**\$\_** est une variable qui permet justement de stocker cette sortie

**.Length** est l'attribut sur lequel on veut filtrer la réponse.

**-gt** est un opérateur de comparaison (cf. ci-après) ; ici, il signifie supérieur à (greater than).

Et enfin, la taille en KiloOctets, le tout entre accolades { }

### Rappel : Get-Member

Utilisez la commande Get-Member pour obtenir les propriétés d'un objet. Dans l'exemple précédent, on a utilisé .Length pour la taille du fichier, mais il existe d'autres attributs. Cf p6 de ce document pour un exemple.

### Opérateurs de comparaison

Ces opérateurs permettent de comparer des données entre elles. Ils sont valables pour comparer des nombres, des chaînes de caractères, ou encore effectuer des recherches sur un tableau.

Vous aurez besoin dans un premier temps des comparateurs suivants :

- **-eq** : pour equal (égal à)
- **-ne** : pour non equal (différent de)
- **-gt** : pour greater than (strictement supérieur à)
- **-ge** : pour greater equal (supérieur ou égal à)
- **-lt** : pour less than (strictement inférieur à)
- **-le** : pour less equal (inférieur ou égal à)

### Exercice 4

Se positionnez dans C:\Windows si ce n'est déjà fait.

Quelle commande permet de :

- Afficher les fichiers dont la taille est supérieur à 60 Ko :
- Afficher les fichiers et les dossiers créés le 1<sup>er</sup> août 2013 et après (attention, la date est toujours exprimée en anglais au format MM/JJ/AAAA) :
- N'afficher que les répertoires auxquels on n'a pas accédés aujourd'hui :

### c. New-Item

---

Cette cmdlet permet de créer des fichiers et des dossiers :

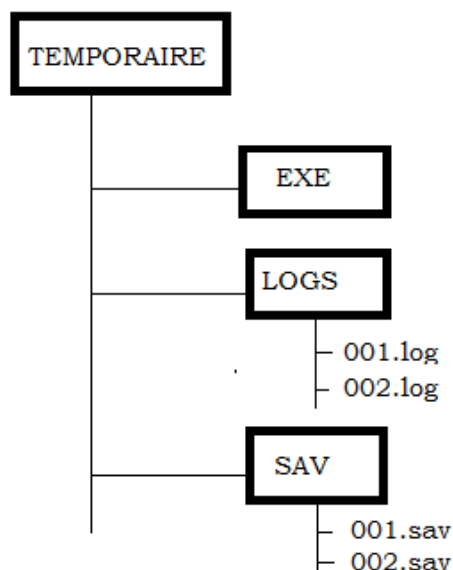
- **-ItemType file** : créé un fichier
- **-ItemType Directory** : créé un dossier
- **-Name** : donne le nom du fichier ou du dossier
- **-Path** : indique l'emplacement.



## Exercice 5

---

Créez la structure de répertoires et fichiers suivantes (sous C:\)



Notez les commandes utilisées :

### d. Copy-Item

---

Cette cmdlet copie les fichiers et dossiers d'un emplacement à un autre. En utilisation classique, sa syntaxe n'a rien de particulier : `copy-Item -path « fichier ou dossier à copier » -destination « nouvel emplacement »`.

## Exercice 6

---

Pour chaque demande, notez la commande.

- Copier tous les fichiers .log de C:\Windows dans c:\Temporaire\Log\
- Copier tous les fichiers (pas les répertoires), qui ne sont pas des archives, de C:\Windows plus ceux des sous-répertoires de C:\Windows, dont la taille est inférieure ou égale à 1Ko dans le répertoire C:\Temporaire\Sav\

Une fois les manipulations effectuées, effacez le répertoire C:\Temporaire et tout son contenu :

## **Conclusion**

Voilà pour la 1<sup>ère</sup> approche de Powershell 3.0 et la gestion des fichiers et dossiers avec les principales commandes et le passage de paramètre avec le | et le filtre Where-Object.

Mais Powershell est bien un langage de scripting ; l'objectif est d'automatiser les process, pas de taper du code en permanence.

Dans la suite de ce document, nous allons donc refaire ces premières manipulations, mais de façon automatique, en créant un 1<sup>er</sup> script.

Pour cela, vous avez besoin de connaître d'autres notions de scripting, comme les variables, les opérateurs, les boucles, les structures conditionnelles et les fonctions Powershell.

## 4) Bases du scripting

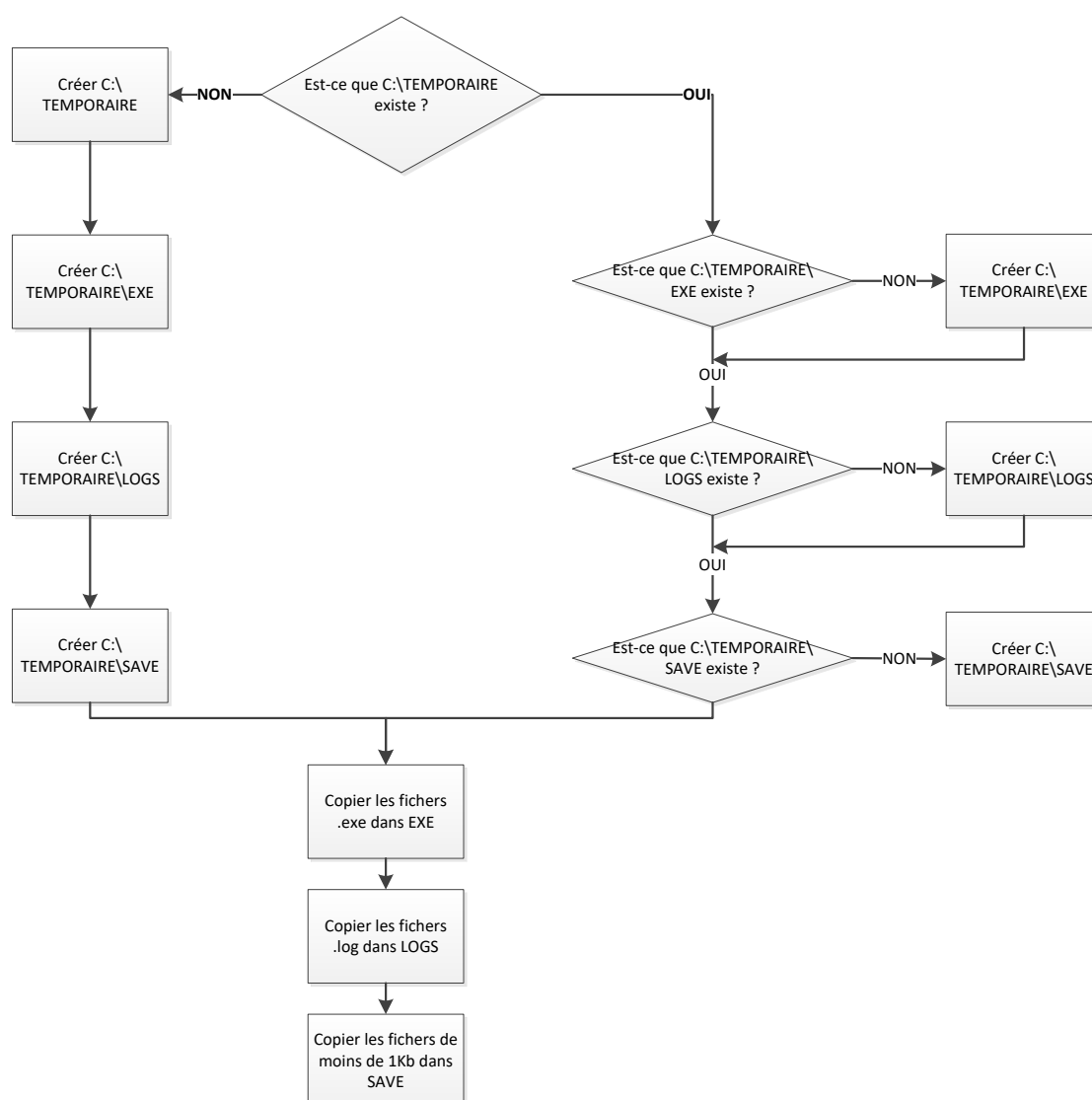
La base de toute réalisation de scripts consiste d'abord à savoir ce qu'on veut faire et à la poser sous forme d'organigramme. Sinon, point de salut !

Si on reprend les exercices réalisés précédemment, que veut-on faire exactement ?

On veut copier les fichiers .exe de C:\Windows vers c:\Temporaire\Exe, puis les fichiers .log de c:\Windows vers c:\Temporaire\logs, et enfin tous les fichiers de c:\Windows et sous-dossiers qui ne sont pas des archives et qui font moins de 1Kb vers C:\Temporaire\Save.

Pour cela, on va d'abord devoir vérifier que les répertoires Temporaire, Exe, Logs et Save sont bien présents, et sinon, les créer, avant d'initier la copie des fichiers.

Ce qui pourrait se traduire sous cette forme d'organigramme :



Le pas à pas reste la meilleure méthode pour apprendre le scripting Powershell et la structure des scripts. C'est ce que nous allons commencer par faire.

Avant toute chose, par défaut, l'exécution des scripts Powershell n'est pas autorisée dans les environnements Microsoft. Il faut donc les autoriser :

- Ouvrez la console ISE en mode administrateur (être loggé en tant qu'administrateur ne suffit pas).
- Entrez la commande **Set-ExecutionPolicy –ExecutionPolicy Unrestricted**
- Et répondre OK aux messages de mises en garde.

Ouvrez un nouveau script dans la console ISE, et c'est parti ...

## a. Test-Path, If-Else, Write-Host et Variables

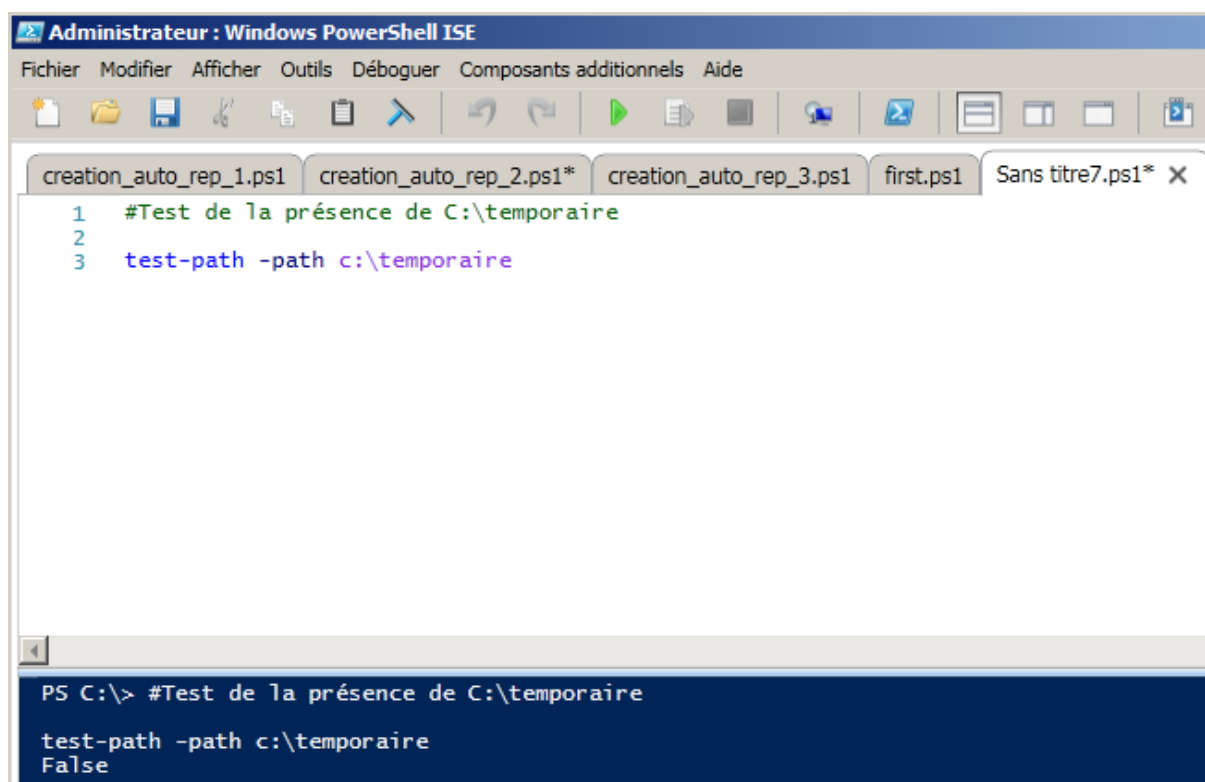
### Test-Path

Il existe une commande Powershell pour à peu près tout ce sur quoi vous voulez intervenir sur votre système.

La commande **Test-Path** teste la présence de fichiers ou de dossier. Sa syntaxe est relativement simple :

**Test-Path –path chemin+nomrep**

Elle renvoie une valeur de type booléen : **True** ou **False**.



The screenshot shows the Windows PowerShell ISE interface. The title bar reads "Administrateur : Windows PowerShell ISE". The menu bar includes "Fichier", "Modifier", "Afficher", "Outils", "Débuguer", "Composants additionnels", and "Aide". The toolbar contains various icons for file operations and execution. The script editor shows a file named "creation\_auto\_rep\_2.ps1\*" with the following code:

```
1 #Test de la présence de C:\temporaire
2
3 test-path -path c:\temporaire
```

The console window at the bottom shows the execution of the script:

```
PS C:\> #Test de la présence de C:\temporaire
test-path -path c:\temporaire
False
```

## Exercice 7

---

- Entrez la commande Test-Path dans votre script
  - Exécutez-le
  - Notez le résultat
- 
- Créez le répertoire C:\Temporaire
  - Relancez le script
  - Notez le résultat
- 

NB : n'oubliez pas de commenter votre script ; rien de plus pénible que de devoir débogger un script non commenté. Pour ce, entrez le caractère # devant chaque ligne de commentaires.

Les booléens ne sont pas très parlant ; on peut améliorer la réponse à notre question, de sorte que s'affiche « Oui le répertoire existe », ou « Non le répertoire n'existe pas » plutôt que True ou False. Mais pour cela, il va falloir utiliser If-Else ; c'est ce qu'on appelle une structure conditionnelle.

## If-Else

---

Cette structure conditionnelle est dérivée de la structure If :

### If (condition)

```
{
    #Bloc d'instructions
}
```

Si la condition est vraie, alors les instructions vont se dérouler. Par exemple :

### If (10 –lt 100)

```
{
    Write-Host 'Bien sûr que 10 est inférieur à 100'
}
```

L'exemple est stupide, mais parlant. Notez au passage l'utilisation de la commande **Write-Host**, qui permet d'afficher du texte en sortie ; le texte à afficher doit être placé entre cote ' le texte'.

Le **If-Else** est calquée sur cette structure :

### If (condition)

```
{
    #Bloc d'instructions n°1
}
Else
{
    #Bloc d'instructions n°2
}
```

Si la condition est vraie, alors le bloc d'instructions n°1 s'exécute, sinon, c'est le bloc d'instructions n°2 qui s'exécute.

C'est cette structure dont nous avons besoin pour afficher clairement 'Oui le répertoire existe' ou 'Non le répertoire n'existe pas' lorsqu'on teste la présence de C:\temporaire.

## Exercice 8

- Modifiez votre script initial en utilisant la structure If – Else
- La condition sera la commande Test-Path
- Si le répertoire existe, on veut afficher 'Oui le répertoire existe'
- Sinon, on veut afficher 'Non le répertoire n'existe pas'
- Testez-le

## Variables

Pour le moment, on n'a testé que l'existence de c:\Temporaire, mais il va aussi falloir tester la présence des sous-répertoires Exe, Logs et Save. Si on veut afficher 'Oui le répertoire existe' ou 'Non le répertoire n'existe pas', on va pouvoir utiliser des variables plutôt que de renseigner à chaque fois cette information dans le bloc d'instructions.

L'avantage avec Powershell, c'est qu'il est capable de définir le type d'une variable automatiquement, et qu'on n'a pas besoin de passer par une ligne de commande interminable.

Plusieurs types de variables sont couramment utilisées : le type **string** pour les chaînes de caractères, le type **int** pour les nombres entiers, et le type **tab** pour un tableau.

Par exemple :

**\$MaVariable1 = 15**

**\$MaVariable2 = 20**

Je viens de créer deux variables de type **int**, qui ont pour valeur 15 et 20. Je vais pouvoir les utiliser pour réaliser des opérations :

**\$MaVariable1 + \$MaVariable2** donne pour résultat **35**.

Idem pour les variables de type **String** :

**\$MaPhrase = 'Oui le répertoire existe'**

**\$MaPhrase** donne pour résultat 'Oui le répertoire existe'.

```
PS C:\> $MaPhrase = 'Oui le répertoire existe'
PS C:\> $MaPhrase
Oui le répertoire existe
PS C:\>
```

Plus besoin du Write-Host pour afficher le contenu de la variable.

Les variables se définissent généralement au tout début du script :

```

Administrateur : Windows PowerShell ISE
Fichier  Modifier  Afficher  Outils  Débuguer  Composants additionnels  Aide

creation_auto_rep_1.ps1  creation_auto_rep_2.ps1*  creation_auto_rep_3.ps

1  $MaVariable1 = 10
2  $MaVariable2 = 25
3  $MaPhrase1 = 'c est supérieur à 30'
4  $MaPhrase2 = 'c est inférieur à 30'
5
6  If (($MaVariable1 + $MaVariable2) -gt 30)
7  {
8      $MaPhrase1
9  }
10 Else
11 {
12     $MaPhrase2
13 }

PS C:\> C:\Users\Administrateur\Documents\Sans titre8.ps1
c est supérieur à 30

PS C:\> |
  
```

## Exercice 9

Modifiez votre script :

- Déclarez 2 variables \$Message1 et \$Message2 qui vont contenir vos messages.
- Remplacez les Write-Host 'messages' par vos variables.
- Testez-le.

## b. If-Else imbriqués

Arrivés à cette étape, on est capables de tester la présence ou pas du répertoire C:\Temporaire.

Si ce répertoire existe, il faut aussi tester la présence ou pas des répertoires Exe, Logs et Sav.

Pour cela, il va falloir imbriquer les structures If-Else :

**If (condition1)**

{

**If (condition 2)**

    {

**#Blocs Instructions**

    }

**Else**

    {

**#Blocs Instructions**

    }

```

}
Else
{
    #Blocs Instructions
}

```

Ce qui pourrait se traduire dans notre cas par :

```

If (C:\temporaire existe)
{
    Afficher le répertoire existe
    If (C:\Temporaire\exe existe)
    {
        Afficher Oui le repertoire existe
    }
    Else
    {
        Afficher Non le repertoire n existe pas
    }
    ... ce à répéter pour chaque sous-répertoires
}
Else
{
    Afficher le répertoire n'existe pas
}

```

## Exercice 10

---

- Modifiez votre script pour tester la présence du répertoire C:\Temporaire et des sous-répertoires Exe, Logs et Sav.
  - Afficher à chaque fois si oui ou non le répertoire existe (utilisez les variables \$Message1 et \$Message2).
  - Testez votre script (pour tous les cas de figure)
- 

On a donc testé la présence des répertoires et afficher des messages selon leurs existences ou pas. Maintenant que vous maîtrisez les If-Else, on va pouvoir passer à la création des répertoires s'ils n'existent pas. Pour cela, rien de plus simple, il suffit d'insérer les commandes New-Item qui vont bien dans les blocs d'instructions (à la place de l'affichage du message. A vous ...



## Exercice 11

---

- Modifiez votre script pour créer les répertoires s'ils n'existent pas
  - Vous pouvez supprimer les messages pour ne pas alourdir le script.
- 

Ca commence à faire beaucoup de If-Else imbriqués tout ça ; le script s'alourdit avec les commandes New-Item et encore, on n'a pas renseigné les commandes Copy-Item (qui sont le but à atteindre, pour rappel).

De plus, si on regarde bien la structure du script, les actions ne sont à effectuer que si les conditions ne sont pas respectées (dans la partie Else).

Plutôt que de tester l'existence des répertoires et agir en conséquence, il peut être judicieux de tester la non-existence des répertoires et agir en conséquence.

### c. La non-condition (double négatif = positif)

---

Il existe un argument **-not** qui permet de tester la non-condition dans les structures conditionnelles. Son utilisation et sa syntaxe est simple, mais elle oblige à repenser l'organisation du script à l'envers.

Prenons l'exemple d'un script qui teste l'existence de c:\test, et qui, si il existe, ne faire rien, et si il n'existe pas, créé le répertoire (c'est notre structure de départ) :

**If (test-path -path c:\test)**

```
{
}
```

**Else**

```
{
```

**New-Item -ItemType Directory c:\test**

```
}
```

Avec le test de la non condition, on inverse la logique :

**If (-not (test-path -path c:\test))**

```
{
```

**New-Item -ItemType Directory c:\test**

```
}
```

**Else**

```
{
```

```
}
```

Comme le Else ne sert plus à rien, il suffit de le supprimer pour utiliser une structure conditionnelle simple : le If.

## Exercice 12

---

- Modifier votre script en utilisant le `-not` sur les conditions
- Supprimez les Else inutiles
- Testez le script

## d. Variables int, tableau et boucle While

---

### While

---

On a vu qu'en utilisant le test de non-condition, on pouvait simplifier considérablement notre script.

Cependant, dans notre exemple, on ne teste la présence que de 3 sous-répertoires (utilisation de 3 structures conditionnelles If). Imaginez qu'il vous faille tester la présence d'une cinquantaine de répertoires ... le script deviendrait rapidement illisible ... Toujours mieux que de les tester un par un en ligne de commande, mais pas vraiment efficace en automatisation, sans compter le risque de fautes de frappes à l'intérieur du script.

Pour simplifier encore le script, on peut utiliser des boucles ; les boucles sont des structures répétitives qui permettent d'exécuter plusieurs fois les instructions qui se trouvent à l'intérieur d'un bloc d'instructions. Au lieu de taper 50 fois les commandes Test-Path et New-Item, on aimerait mieux la taper une fois et faire passer automatiquement 50 paramètres.

La boucle While (traduire par Tant que) permet ceci :

### While (condition)

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

En pseudo-langage, on pourrait le traduire par :

### Tant que (les répertoires n'existent pas)

```
{
    Créer les répertoires
}
```

Plus simple à gérer qu'une succession de If et If-Else ... mais comment faire passer les paramètres ? Dans notre exemple, il faut en 1<sup>er</sup> lieu tester la présence de c:\temporaire, puis, dans notre boucle While, on aimerait tester successivement la présence des sous-répertoires Exe, Logs et Sav (les paramètres), les uns après les autres.

## Tableau

Pour cela, avant même d'utiliser While, on va être obligé de créer ce qu'on appelle un tableau à 1 dimension. C'est un type de variable à laquelle il suffit d'affecter plusieurs valeurs séparées par des virgules.

**\$tab = 'A','B','C','D','E'**

Pas d'espace entre les virgules ; le texte doit être mis entre cotes ` `.

Peut se traduire par un tableau simple :

A	B	C	D	E
---	---	---	---	---

La 1<sup>ère</sup> case est située à l'indice 0, la seconde à l'indice 1, la 3<sup>ème</sup> à l'indice 2, etc ...

A	B	C	D	E
Indice 0	Indice 1	Indice 2	Indice 3	Indice 4

Pour afficher le contenu de la case 0 :

**\$tab[0]**

Pour afficher le contenu de la case 3 :

**\$tab[3]**

Pour afficher les contenus des cases 1 à 4 :

**\$tab[1..4]**

## Exercice 13

- Créez un nouveau script dans lequel vous allez déclarer un tableau de caractères (ça peut être des mots ou des phrases) à 4 indices.
- Affichez la valeur de la 1<sup>ère</sup> case.
- Affichez la valeur des cases 3 à 4.

## Intégration du tableau dans la boucle While

Je peux donc récupérer le contenu d'une case d'un tableau et le faire passer en paramètres, ce qui peut potentiellement nous aider pour nos tests d'existences des sous-répertoires ; exe, logs et sav seront contenus dans un tableau.

Mais je ne vais pas écrire 50 lignes de \$tab[0] et \$tab[1] et \$tab[2], etc ... dans mon script ; ce serait franchement contreproductif. De même, il va être impossible de forcer la boucle While à lire toutes les cases du tableau. La notation \$tab[0..49] me direz-vous ... oui mais ... mais la boucle While se termine

dès que la condition est fausse. Imaginez que le répertoire Exe existe ; le script passe une 1<sup>ère</sup> fois dans la boucle While, teste la présence du répertoire Exe (tab[0]) et, comme il existe, la boucle se termine ... peu importe que les répertoires logs et sav (et éventuellement les 50 suivants) n'existent pas !

Il faut donc impérativement forcer la boucle While à lire le tableau en entier ; ceci est possible si on initialise une sorte de compteur.

### Int comme un compteur

---

C'est l'utilisation classique de la combinaison entre tableau et boucle While, avec une variable de type int (un nombre entier qui sert de compteur).

Prenons pour exemple un script qui affiche toutes les valeurs d'un tableau, en utilisant un compteur :

```
$tab = 'A','B','C','D','E','F'
```

```
$compteur = 0
```

```
While ($compteur -lt $tab.Length)
```

```
{
```

```
    $tab[$compteur]
```

```
    $compteur = $compteur+1
```

```
}
```

1<sup>er</sup> passage dans la boucle ; \$compteur est à 0 ; \$tab.length est égal à 6 (la longueur du tableau). La condition est vraie, le bloc d'instructions s'exécute ; on affiche la valeur de la 1<sup>ère</sup> case (\$tab[\$compteur] équivaut à \$tab[0] qui équivaut à la case 1 du tableau). Puis on incrémente la variable \$compteur qui du coup, passe à 1.

2<sup>nd</sup> passage de boucle ; \$compteur est à 1 ; \$tab.length toujours à 6 ; la condition est vraie, on affiche la valeur de la 2<sup>nd</sup> case (\$tab[\$compteur] équivaut à \$tab[1] qui équivaut à la 2<sup>nde</sup> case du tableau. Puis on incrémente \$compteur qui passe à 2, etc ...

Jusqu'à ce que \$compteur passe à 6 ; dans ce cas, la condition n'est pas respectée (6 n'est pas strictement inférieur à 6) et la boucle s'arrête).

### Exercice 14

---

- Reprenez le script de l'exercice 13
- Créez une variable \$compteur à 0
- Créez une boucle While qui affiche :
  - Le contenu de la case 1 est : votre\_contenu
  - Le contenu de la case 2 est : votre\_contenu
  - Etc pour toutes les cases du tableau.

## Exercice 15

---

- Reprenez le script de création automatique des répertoires.
- Modifiez-le pour ajouter des boucles While au lieu des If imbriqués.
- Il vous faudra pour cela définir un tableau, un compteur, garder une structure conditionnelle If-Else et utiliser des boucles while (voir des boucles while imbriquées).
- Bon courage !

## Exercice 16

---

Ne reste plus qu'à intégrer les commandes qui vont permettre de copier les fichiers dans ces nouveaux répertoires. Pour rappel, on souhaite :

- copier les fichiers .exe de C:\Windows vers c:\Temporaire\Exe
- copier les fichiers .log de c:\Windows vers c:\Temporaire\logs
- copier tous les fichiers de c:\Windows et ses sous-dossiers qui ne sont pas des archives et qui font moins de 1Kb vers C:\Temporaire\Sav

## e. Les fonctions

---

Les fonctions sont un ensemble d'instructions auxquelles on donne un nom. Dans un script, il arrive souvent qu'on répète les mêmes suites d'instructions dans plusieurs boucles (l'instruction new-Item -ItemType Directory -name \$tab[\$compteur] dans votre script par exemple).

Pour simplifier encore le script, on peut créer en début de script une fonction, que l'on appellerait CreerRep, et qui comporterait cette instruction.

Ensuite, dans le script, au lieu de taper la suite d'instructions complète, il suffirait de taper CreerRep et hop, le tour serait joué. Une fonction se déclare de cette manière :

### Function Nom\_Fonction

```
{
    #Bloc d'instructions
}
```

Puis pour appeler la fonction dans le script :

### Nom\_Fonction

## Exercice 17

---

Vous avez compris le principe ...

- Modifiez votre script en créant une fonction CréerRep qui contient l'instruction new-Item -ItemType Directory -name \$tab[\$compteur]
- Remplacez l'instruction par l'appel de votre fonction dans le script.