

# FORMATION QUALITE LOGICIELLE

Module 2 : Script shell



Ecole de la Qualité Logicielle

# SCRIPT SHELL

## Objectif du cours



### Linux

- Connaître la structure Unix/Linux
- Attribuer des droits particuliers sur des fichiers



### Commandes utiles

- Utiliser des lignes de commande
- Connaître et utiliser des commandes fondamentales



### Scripting

- Comprendre le fonctionnement et la structure d'un script
- Rédiger des scripts simples pour administrer un système Unix
- Utiliser des variables et des algorithmes simples

HIGH  
LEVEL

### Scripting avancé

Utiliser les commandes **awk** et **sed** (hors support)  
Création d'un script de planificateur de tâches (hors support)

# Partie 1 : INTRODUCTION

# SCRIPT SHELL

## Introduction : d'Unix à Linux

### UNIX

- ✓ Unix est un système d'exploitation
- ✓ Unix existe depuis 1970.
- ✓ **Multi-couches** → différentes couches composent le système. Ex : matériel, noyau, make, le shell...
- ✓ **Multi-tâches** → plusieurs logiciels peuvent fonctionner simultanément sur une même machine
- ✓ **Multi-utilisateurs** → plusieurs utilisateurs et groupes d'utilisateurs peuvent travailler sur une même machine.
- ✓ Son historique est disponible dans l'annexe de ce document

### LINUX

- ✓ Linux est système d'exploitation basé sur Unix
- ✓ Linux existe depuis 1991
- ✓ Comme Unix, il est multi-couches, multi-tâches et multi-utilisateurs
- ✓ Linux est basé sur une structure définie, régie par des droits
- ✓ Dans Linux tout est fichier (même les répertoires)
- ✓ Contrairement à la légende, Linux peut être payant et « boguer »

# SCRIPT SHELL

## Introduction : Arborescence de fichiers Linux

### Exemple d'arborescence d'un système Linux

/ symbolise la racine du système Linux

**/bin** → contient les binaires de certaines commandes utiles

**/etc** → contient les fichiers de configuration et les systèmes de bdd

**/dev** → contient des fichiers qui représentent des périphériques physiques

**/home** → **contient les dossiers des utilisateurs (le plus important pour nous)**

**/mnt** → permet de monter un volume (par exemple une clé USB)

**/lib** → contient des librairies système

**/root** → le dossier de l'administrateur système (superutilisateur root)

**/usr** → contient par exemple les applications installées

**/var** → contient des éléments variables et temporaires

- Plusieurs arborescences composent Linux mais / est toujours la racine.

# SCRIPT SHELL

## Introduction : Les utilisateurs et groupes d'utilisateur(s)

### Utilisateurs et groupes

- Les **utilisateurs** standards → vous, votre voisin, une autre personne, une application qui simule un utilisateur factice.
- Le superutilisateur **root** → il est l'administrateur du système Linux. Par exemple, il gère l'accès à certains répertoires et fichiers, il gère la configuration du système, il ouvre le système Linux sur d'autres systèmes...
- Les **groupes** → certains utilisateurs standards sont affiliés à des groupes d'utilisateurs.



« L'utilisateur root » possède tous les droits même celui de détruire le système Linux. Il est fortement recommandé de ne l'utiliser que pour des tâches d'administration système. Toutes autres tâches doivent être réalisées avec un utilisateur dédié!!!

# SCRIPT SHELL

## Introduction : Les droits (autorisations) dans Linux

### Linux fonctionne sur un système de droits (autorisations)

**Droits** d'accès aux fichiers et dossiers (souvenez-vous, sous Linux tout est fichier!)

- Les droits de lecture (**read**) --> lire un contenu
- Les droits d'écritures (**write**) --> écrire un contenu dans un fichier
- Les droits d'exécution (**execute**) --> exécuter un contenu (par exemple script)

### Règles sur les droits

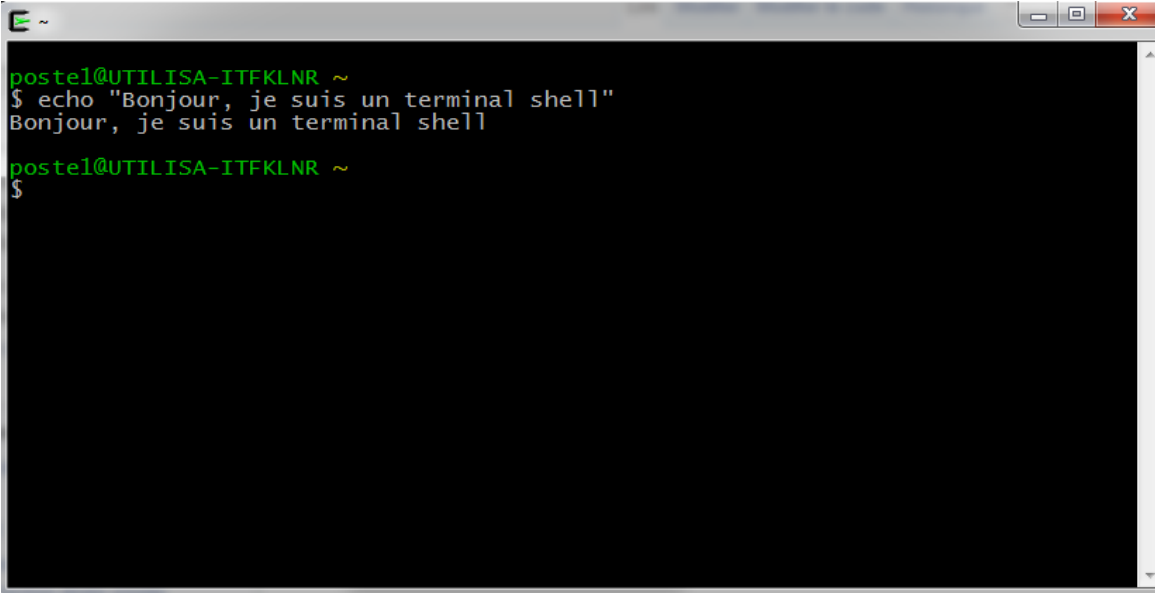
- Ces droits sont applicables à tous les utilisateurs (même root), aux groupes et à des utilisateurs extérieurs
- Un système Linux possède des droits généralisés à tout le système, c'est le umask

# SCRIPT SHELL

## Introduction : Qu'est ce qu'un shell Unix? (1/2)

### Un shell Unix c'est :

- ✓ Une des couches qui compose un système d'exploitation type Unix (comme Linux!!)
- ✓ Un interpréteur de commandes destiné aux systèmes type Unix. L'utilisateur peut ainsi interagir avec les fonctionnalités internes du système d'exploitation
- ✓ Il se présente sous la forme d'une interface en ligne de commandes accessibles par exemple depuis une console ou un terminal.



```
postel@UTILISA-ITFKLNR ~  
$ echo "Bonjour, je suis un terminal shell"  
Bonjour, je suis un terminal shell  
postel@UTILISA-ITFKLNR ~  
$
```



# SCRIPT SHELL

## Introduction : Qu'est ce qu'un shell Unix? (2/2)

### Différencier une GUI d'une ligne de commande

#### Interface graphique (GUI)

- L'interface utilisateur est dessinée à l'écran
- Les objets qui composent l'écran sont par exemple manipulables avec une souris

#### Ligne de commande

- L'interface utilisateur n'est plus dessinée à l'écran
- Les actions avec la machine s'effectuent via des commandes informatiques. Ces commandes sont simplement des programmes que l'utilisateur exécute. Par exemple, la commande *je\_lance\_un\_process\_proc* exécute le programme *proc*. Dans le cadre d'une interface graphique, l'utilisateur aurait par exemple cliqué sur une icône nommée *je\_lance\_un\_process\_proc*.
- Une ligne de commande peut s'assimiler à un ou plusieurs clics sur un ou plusieurs processus.

# SCRIPT SHELL

## Introduction : Présentation du WSL (1/3)

### **WSL (*Windows Subsystem for Linux*)**

- ✓ C'est l'application avec laquelle nous allons utiliser Linux
- ✓ C'est une implémentation d'un système (distribution) Linux sur un système Windows.
- ✓ Utiliser WSL sur Windows équivaut à utiliser une machine Linux
- ✓ WSL permet d'avoir un environnement shell basé sur UNIX sur un système hôte Windows

### **Installation:**

1. L'installation se fait grâce à un script qui se situe sur le réseau: <\\form-dc02\1-Logiciels\1 - Installateurs\installdeb.bat>
2. Copier-coller ce fichier sur votre poste Windows (sur le *Bureau* ou dans *Téléchargements*)
3. Exécuter le script en tant qu'administrateur avec clic droit, puis « Exécuter en tant qu'administrateur »
4. À la fin de l'installation, tapez « Debian » dans le menu Démarrer et lancez l'application quand vous voyez l'icône apparaître.

### Configurer le WSL

Une fois démarré, l'installation se poursuit toute seule.

Puis le système vous demande d'entrer un login et un mot de passe.

Mettre **formation** comme login et **@FCEQL1** comme mot de passe (comme sur votre poste Windows)

Alternatives pour accéder au WSL

- Dans une fenêtre DOS ou Powershell, taper **debian**
- Double clic sur le raccourcis de l'application Debian
- Dans une fenêtre DOS ou Powershell, taper **bash** (cela ouvre un shell BASH en lieu et place du shell DOS, avec vos paramètres de session WSL)

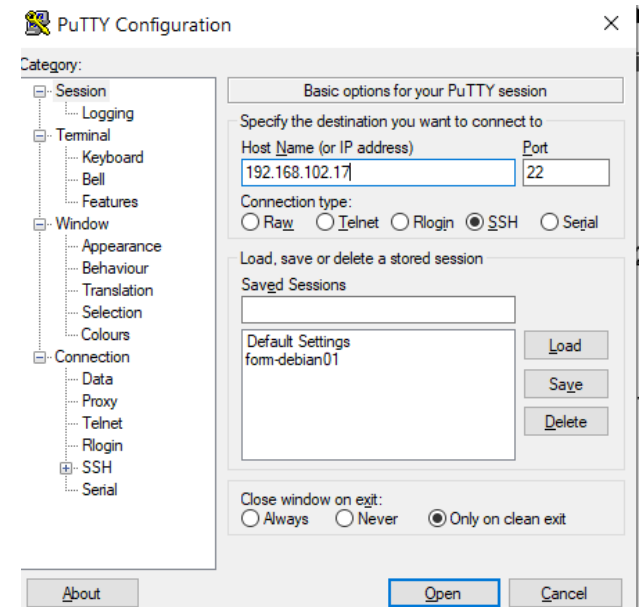
# SCRIPT SHELL

## Introduction : Présentation de Putty

**Putty** est un logiciel qui permet **d'ouvrir un Shell à distance**. C'est lui qui vous permet de **vous connecter à votre terminal Linux**.

C'est un **logiciel client** car il se connecte à un serveur.

1. Ouvrir Putty
2. Entrez l'adresse IP du serveur 192.168.102.17
3. Cliquez sur Open
4. Acceptez la signature du serveur
5. Entrez votre login [**eleveX**] et tapez sur Entrée
6. Entrez votre mot de passe [**linuxisthbest**] et tapez sur Entrée



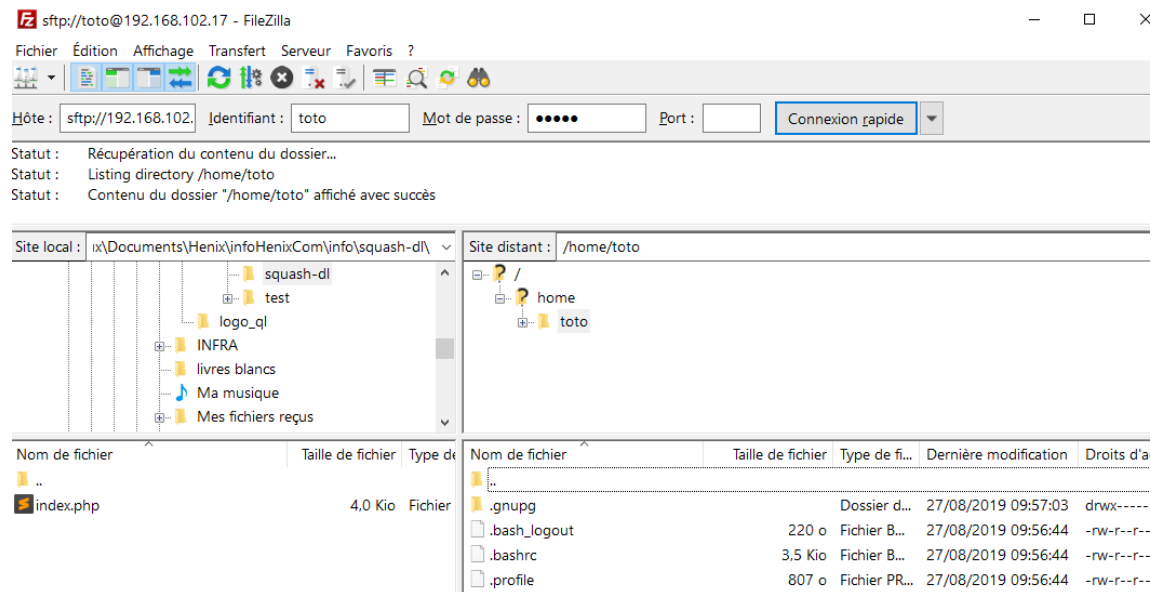
# SCRIPT SHELL

## Introduction : Présentation de Filezilla

**Filezilla** est un logiciel qui permet **d'échanger des fichiers à distance**.  
C'est lui qui vous permet de **vous connecter à votre système de fichier sur Linux**.

C'est un **logiciel client** car il se connecte à un serveur.

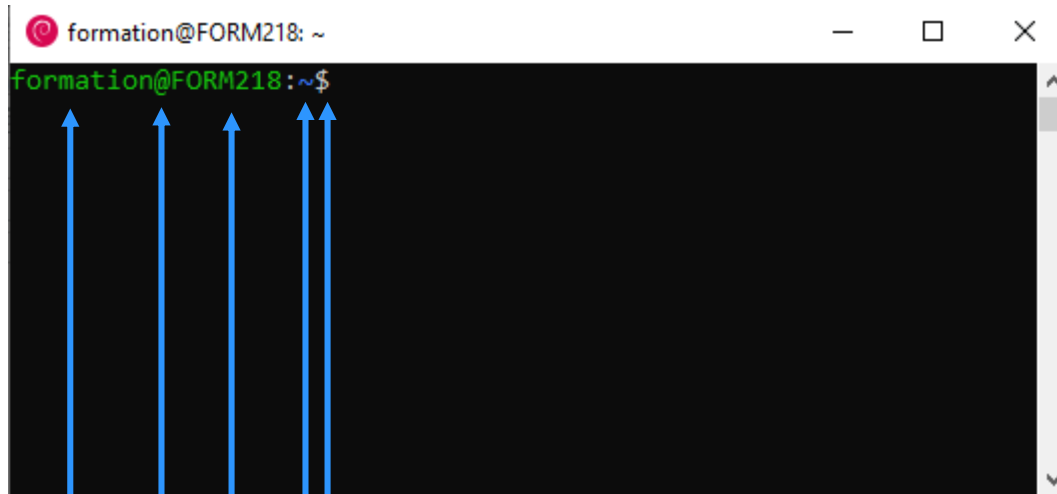
1. Ouvrir Filezilla
2. Entrez l'adresse IP du serveur 192.168.102.17, vos identifiants et le port [22] ou bien mettez sftp://
3. Cliquez sur **Connexion rapide**





# SCRIPT SHELL

## Introduction : Configuration du WSL (3/3)



```
formation@FORM218: ~  
formation@FORM218:~$
```

\$ : indique que vous le shell attend une commande

~ : répertoire courant

FORM218 : nom de la machine (*hostname*)

@ : séparateur (permet de différencier l'utilisateur et la machine)

formation : login de l'utilisateur courant du shell

# Partie 2 : Commandes utiles





### Les commandes utiles :

- ✓ Sont les commandes que l'on utilise quotidiennement sous Linux
- ✓ Servent par exemple :
  - ✓ à **naviguer** dans Linux (système d'exploitation basé sur environnement Unix).
  - ✓ à **créer**, **modifier** et **supprimer** des répertoires et des fichiers
  - ✓ à **copier** / **couper** / **coller** des répertoires, des fichiers et leurs contenus
  - ✓ à **extraire** des données de fichiers
  - ✓ à **modifier** des droits, **rediriger** des flux, **automatiser** des tâches (composées d'une ou plusieurs commandes)

# SCRIPT SHELL

## Commandes utiles : structure d'une commande shell (1/2)

### Exercice 3 :

#### Analyse de la commande ls (list → liste l'ensemble du contenu d'un dossier)

1. Dans votre shell, saisir la commande `ls`. Que constatez-vous?
2. Saisir la commande `ls -a`. Que constatez-vous?
3. Saisir la commande `ls -la`. Que constatez-vous?

### Exercice 4 : Comprendre les résultats

- ✓ `ls` est la commande pour lister le contenu d'un répertoire
- ✓ `ls -a` est la commande list suivi de l'option a.
- ✓ `ls -la` est la commande list suivi de deux options a et l. Les options se mettent à la suite sans espace.
- ✓ La commande `man ls` montre que les options `-a` signifie `-all` et `-l` affiche le résultat de `ls` sous forme d'une liste.



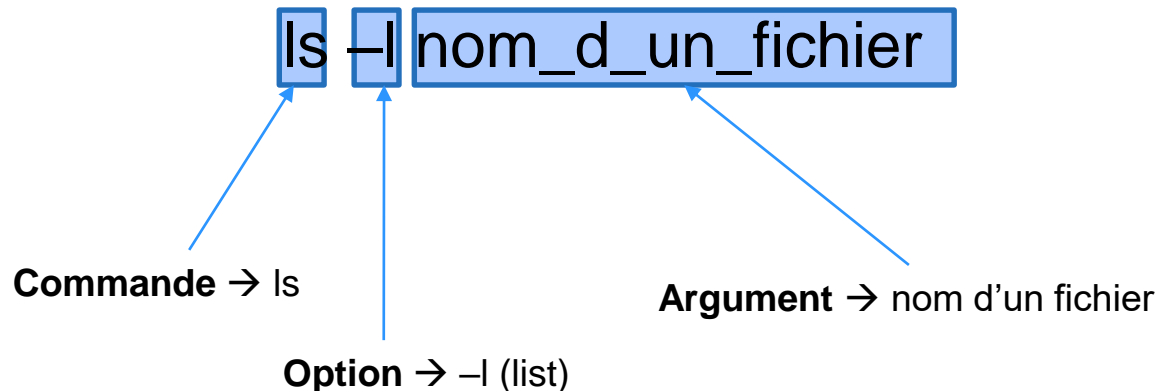
Les dossiers et fichiers préfixés d'un point sont des items cachés. C'est pour cette raison que la commande `ls -l` ne les montre pas. L'option `-a` révèle tout.

# SCRIPT SHELL

## Commandes utiles : structure d'une commande shell (2/2)

### Ajouter un argument à une commande

- Il est souvent nécessaire, à la suite d'une commande avec ou sans option, d'ajouter un ou des arguments.
- Par exemple : `ls -l nom_d_un_fichier`



- Il est possible d'ajouter plusieurs arguments en les séparant par un simple espace. Exemple : `ls -l nom_d_un_fichier1 nom_d_un_fichier2`



Ces règles de syntaxe fonctionnent pour toutes les commandes que nous verrons

# SCRIPT SHELL

## Commandes utiles : lister le contenu d'un dossier avec la commande ls

**Commande ls**, liste l'ensemble du contenu d'un répertoire

- Syntaxe : `ls`
- Principales options :
  - `-a` → montre tous le contenu d'un dossier
  - `-R` → montre récursivement le contenu d'un dossier
  - `-r` → inverse l'ordre d'affichage du contenu d'un dossier
  - `-h` → indique la taille d'un fichier/dossier
  - `-l` → créer une simple liste
  - `-d` → n'affiche que les dossiers
- Exemple : `ls -l`

### Exercice 5 :

1. Indiquer les dimensions des fichiers `.bash_history` et `.bash_profile`
2. Afficher chronologiquement une simple liste de vos dossiers cachés.

# SCRIPT SHELL

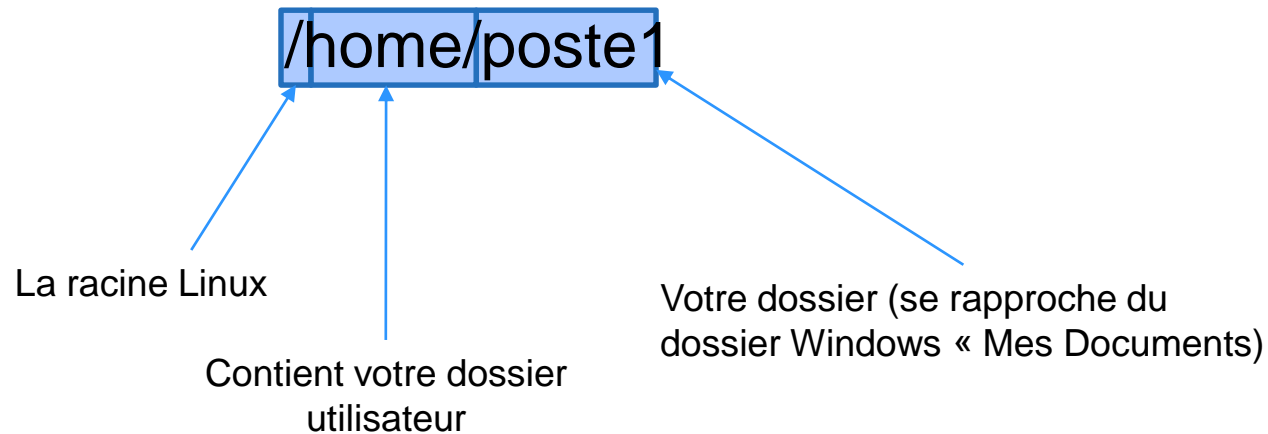
## Commandes utiles : savoir « où je suis » avec pwd

**Commande pwd** (*print working directory*), vous indique dans quel répertoire vous vous trouvez

➤ Syntaxe : pwd

### Exercice 6 :

1. Tapez la commande pwd
2. Détaille du résultat



# SCRIPT SHELL

Commandes utiles : besoin d'info sur les commandes? La commande man!

**Commande man** (manual), indique le manuel d'utilisation des commandes Linux

- Cette commande est très utile pour connaître les syntaxes, options et arguments applicables aux commandes utilisables sous Linux.
- Syntaxe basique : `man Le_nom_de_La_commande_souhaitée`
- La commande man peut contenir des options

## Exercice 7 :

1. Dans un shell bash, saisir la commande man. Que constatez-vous?
2. Afficher le manuel de la commande ls.

# SCRIPT SHELL

## Commandes utiles : gagner en mobilité avec la commande cd (1/2)

**Commande cd** (change directory), permet de changer de répertoire

- Syntaxe : `cd nom_du_répertoire_de_destination`
- Exemples :
  - `cd -` → revient dans le répertoire précédent
  - `cd` → se déplace dans votre \$HOME
  - `cd ..` → remonte d'un répertoire
  - `cd ../../` → remonte de deux répertoires (etc...)

### Chemin relatif et chemin absolu

- **Le chemin relatif** indique un chemin **depuis le répertoire dans lequel vous vous trouvez**. Par exemple depuis votre répertoire my\_home : `cd nom_du_répertoire_de_destination`
- **Le chemin absolu** indique un chemin **depuis la racine /**. Par exemple : `cd /home/my_home/nom_du_répertoire_de_destination`
- Ces règles sur les chemins fonctionnent avec toutes les commandes qui peuvent utiliser des chemins.

# SCRIPT SHELL

## Commandes utiles : gagner en mobilité avec la commande cd (2/2)

### Exercice 8 :

1. Depuis votre shell, se rendre dans le répertoire /tmp
2. Depuis le répertoire /tmp, indiquez quatre commandes différentes qui vous permettrait de retourner dans votre répertoire \$HOME (poste1).  
Tester les commandes et assurez-vous de vous trouver dans votre \$HOME



**\$HOME** est une variable d'environnement qui contient le chemin de votre dossier de travail. Tapez **echo \$HOME**  
Pour afficher toutes les variables d'environnement, tapez **env**



# SCRIPT SHELL

## Commandes utiles : créer et supprimer un répertoire

**Commande mkdir** (*make directory*), crée un répertoire

➤ Syntaxe : `mkdir nom_du_répertoire_à_creer`

**Commande rmdir** (*remove directory*), supprime un répertoire **VIDE**

➤ Syntaxe : `rmdir nom_du_répertoire_à_supprimer`

**Commande rm -Rf** *nom\_du\_répertoire\_à\_supprimer*, supprime un répertoire qui contient d'autres répertoires et des fichiers



La ligne de commande `rm -Rf` supprime récursivement et définitivement tout le contenu d'un répertoire (donc aussi les sous répertoires et fichiers). Cette commande mal exécutée peut détruire tout un environnement Linux sans aucune solution de retour arrière.

- Eviter d'exécuter cette commande avec le superutilisateur (root)
- Eviter d'utiliser `*` à la place du nom du répertoire à supprimer. `*` signifie tout le contenu présent dans le répertoire où vous exécutez la commande.

# SCRIPT SHELL

## Commandes utiles : créer et supprimer un fichier

**Commande touch**, crée un simple fichier

➤ Syntaxe : `touch nom_du_fichier_à_creer`

**Commande rm** (remove), supprime le fichier vide ou plein

➤ Syntaxe : `rm nom_du_fichier_à_supprimer`

### Exercice 9 :

1. Dans votre \$HOME (à vérifier avec pwd), créer un répertoire tmp\_directory
2. **Depuis votre \$HOME**, créer en une seule ligne de commande 3 répertoires tmp1, tmp2 et tmp3
3. Se déplacer dans le répertoire tmp3 et créer le fichier tmp\_file
4. **Retourner dans votre \$HOME** et supprimer le fichier tmp\_file
5. Supprimer **directement** le répertoire tmp\_directory

# SCRIPT SHELL

## Commandes utiles : déplacer des répertoires et fichiers

**Commande mv** (move), coupe et colle un répertoire et/ou un fichier

- Syntaxe : `mv SOURCE DESTINATION`
- Exemple : `mv fichier1 /tmp/fichier1`

**Commande cp** (copy), copie un fichier ou un répertoire vide (si le répertoire contient des données, la commande ne fonctionnera pas).

- Syntaxe : `cp SOURCE DESTINATION`
- Exemple : `cp fichier1 fichier2`

**Commande cp -r** (copy), copie un répertoire et son contenu

- Syntaxe : `cp -r SOURCE DESTINATION`
- Exemple : `cp -r /home/user1 /home/user2`



Toutes ces commandes ont la même syntaxe de base:

`mv|cp|ln [option] argument1 argument2`

# SCRIPT SHELL

## Commandes utiles : gérer les droits sur les fichiers et répertoires (1/3)

La commande `ls -l` affiche les différents droits des fichiers et répertoires

- ✓ La 1<sup>ère</sup> colonne indique le type de fichier : '-' fichier, 'd' directory, 'l' link
- ✓ Nous retrouvons rwx pour read, write, execution (utilisateurs, groupes, les autres)

-rwxr-xr-x	1	postel	None	1,9K	5 déc. 13:24	.inputrc
-rw-----	1	postel	None	37	8 déc. 15:13	.lessht
-rw-r--r--	1	postel	None	28	6 déc. 08:26	.minttyrc
-rwxr-xr-x	1	postel	None	1,3K	5 déc. 13:24	.profile
drwxr-xr-x+	1	postel	None	0	6 déc. 11:56	.vim

Droits lecture:écriture/exécution pour utilisateur (ex : vous)	Droits lecture:écriture/exécution pour groupe	Droits lecture:écriture/exécution pour les autres	Nom du propriétaire du fichier/répertoire	Groupe d'appartenance du propriétaire	Taille du fichier/répertoire	Date de création du fichier/répertoire	Nom du fichier/répertoire
--	---	---	---	---------------------------------------	------------------------------	--	---------------------------

**.vim est un répertoire, l'utilisateur à tous les droits, le groupe et les autres peuvent l'ouvrir**

# SCRIPT SHELL

## Commandes utiles : gérer les droits sur les fichiers et répertoires (2/3)

La commande **chmod** s'utilise de manière « relative » ou « absolue »

### Manière relative :

- ✓ Modifie **simplement** des droits **tout en conservant les anciens droits**

#### non modifiés

- u : user
- g : group
- o : other
- a : all
- r : read
- w : write
- x : execute

#### Exemples :

- Donner au groupe des droits d'exécution : `chmod g+x nom_fichier`
- Donner au groupe des droits d'écriture et retirer aux autres des droits de lecture : `chmod g+w o-r nom_fichier`
- Donner à tout le monde tous les droits : `chmod a+rwx nom_fichier`

### Manière absolue :

- ✓ **Ecrase les droits existants** par de **nouveaux droits**
- ✓ La manière « absolue » est basée sur une conversion binaire/décimale
- ✓ Par exemple : je donne tous les droits à user, droits de lecture à group et droits de lecture aux autres
  - ✓ User : `rwx` → 111 → 7
  - ✓ Group : `r--` → 100 → 4
  - ✓ Other : `r--` → 100 → 4

Dec	Bin	Droits
1	001	x
2	010	w
3	011	wx
4	100	r
5	101	rx
6	110	rw
7	111	rwx

- ✓ Soit : `chmod 744 nom_fichier`
- Module 2 : Script shell

# SCRIPT SHELL

## Commandes utiles : gérer les droits sur les fichiers et répertoires (3/3)

**Commande chmod** (change mode), gère les droits sur les répertoires et les fichiers

- Syntaxe : `chmod droits nom_du_dossier_OU_fichier`
- Principales options
  - -R → récursif

### Exercice 9 :

1. Donner les commandes absolues pour les permissions suivantes :
  - a. je souhaite que tous aient des droits d'exécution et de lecture
  - b. r et x pour u, w pour g, rw pour o
  - c. lecture exécution utilisateur, écriture exécution groupe, rien pour les autres
- d. Donner les commandes relatives pour les permissions suivantes :
  - a. rwx r-x r-x
  - b. 755
  - c. tous les droits pour tous

# SCRIPT SHELL

## Commandes utiles : modifier utilisateur et groupe

**Commande chown** (change the owner), modifie l'utilisateur et le groupe qui possède le fichier ou le répertoire

- Syntaxe : `chown utilisateur:group nom_du_dossier_OU_fichier`
- Principales options
  - `-R` → récursif

# SCRIPT SHELL

## Commandes utiles : recherche

**Commande find**, recherche tous les fichiers nommés *nom\_fichier*

- Syntaxe : `find -name nom_fichier`
- Exemple : `find -name *.txt` --> recherche tous les fichiers .txt

**Commande look**, recherche tous les mots commençant par un préfixe

- Syntaxe : `look prefixe`
- Exemple : `look toto` → remonte tous les mots commençant par le préfixe toto

**Commande ls -lrt**, liste les fichiers par date

**Commande grep**, affiche une ligne de correspond à un pattern (soit dans un fichier, soit dans le résultat d'une ligne de commande)

- Syntaxe : `grep pattern nom_de_mon_fichier`
- Exemple : `grep bash tmp.txt` → affiche en console le pattern bash présent dans un fichier nommé tmp.txt



# SCRIPT SHELL

## Commandes utiles : Qui est là? Commandes whoami, who

**Commande whoami**, affiche l'identité de l'utilisateur (user) connecté

- Syntaxe : whoami

**Commande who**, affiche la liste des utilisateurs connectés ainsi que le terminal utilisé

- Syntaxe : who
- Principales options
  - -a → tous

# SCRIPT SHELL

## Commandes utiles : commandes d'entrée et de sortie standard

**Commande echo**, affiche du texte à l'écran ou dans un fichier

- Syntaxe : `echo « mon_texte »`

**Commande read**, permet de saisir du texte

- Syntaxe : `read`

**Combinaison des commandes echo et read**

```
echo "Quel est votre prenom ? "
```

```
Read
```

**Ou bien ...**

```
read -p
```

# SCRIPT SHELL

## Commandes utiles : date et heure

**Commande cal** (calendar), affiche le calendrier

- Syntaxe : `cal`
- Exemples :
  - `cal` → affiche le calendrier
  - `cal 3 2017` → affiche le calendrier de mars 2017
  - `cal -3` → affiche le calendrier sur 3 mois

**Commande date**, affiche la date système

- Syntaxe : `date`

**Commande time**, indique la durée d'exécution d'une commande

- Syntaxe : `time nom_d_une_commande`

# SCRIPT SHELL

## Commandes utiles : archive et compression

**Commande tar**, archive et compresse des répertoires

- Syntaxe : `tar -options nom_de_larchive nom_répertoire`
- **Création d'une archive**  
`tar -cf archive.tar nom_répertoire`
- **Création d'une archive compressée**  
`tar -zcf archive.tar.gz nom_répertoire`
- **Décompression d'une archive**  
`tar -xzf archive.tar.gz nom_répertoire`
- Principales options
  - -v → mode verbose, affiche le détail de la compression/décompression/archive

# SCRIPT SHELL

## Commandes utiles : internet

**Commande links** (lynx-like), ouvre le navigateur web links

- Syntaxe : `links`

**Commande wget**, permet le téléchargement de données depuis une URL

- Syntaxe : `wget URL`

# SCRIPT SHELL

Commandes utiles : effectuer deux commandes sur une même ligne avec ;

**Il est possible d'effectuer deux commandes DIFFERENTES sur une même ligne en utilisant le symbole ;**

- Syntaxe : *commande\_1;commande\_2*
- Exemple : `ls;pwd`

**Essayer la commande. Que constatez-vous?**

# SCRIPT SHELL

## Commandes utiles : opérateurs binaires (1/2)

**Il est possible de lier l'action de deux commandes en effectuant un tube (pipe)**  
**L'action redirige le résultat d'une commande vers l'entrée de la commande suivante**

- Syntaxe : `commande_1 | commande_2`
- Exemple : `ps aux | grep bash`

L'exemple ci-dessus liste les processus auxiliaires de votre système et affiche les lignes qui correspondent au pattern « bash ». Tester la commande.

# SCRIPT SHELL

## Commandes utiles : opérateurs binaires (2/2)

**Il est possible d'enchaîner deux commandes à la suite si et seulement la première commande est un succès complet**

- Syntaxe : `commande_1 && commande_2`
- Exemple : `pwd && exit`

L'exemple ci-dessus indique le répertoire dans lequel vous vous trouvez puis quitte le shell. Effectuer la commande suivante `pwd && exit` et observez le résultat.



### Partie 3 : Les flux

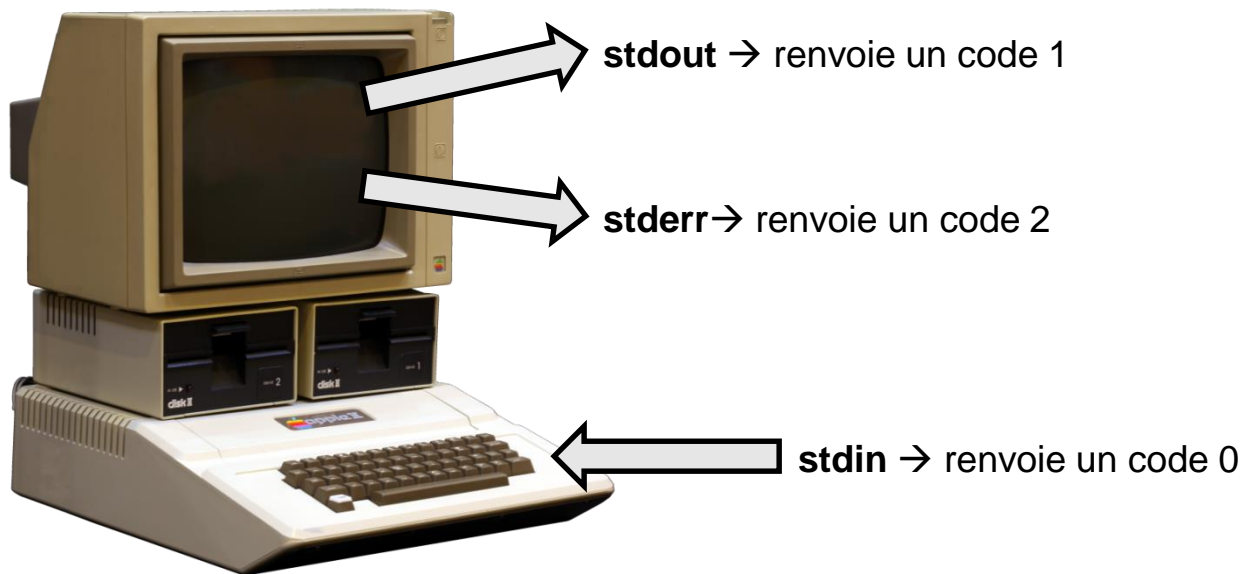


# SCRIPT SHELL

## Les flux : entrées et sorties de processus (1/2)

Chaque commande possède trois flux standards pour communiquer avec l'utilisateur

- l'**entrée** standard nommée **stdin** (identifiant 0) : il s'agit par défaut du clavier
- la **sortie** standard nommée **stdout** (identifiant 1) : il s'agit par défaut de l'écran
- la **sortie** d'erreur standard nommée **stderr** (identifiant 2) : il s'agit par défaut de l'écran



# SCRIPT SHELL

## Les flux : entrées et sorties de processus (2/2)

- Lorsque l'on exécute une commande, le shell redirige le résultat de la commande à l'écran :

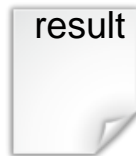
ls -l



```
postel@UTILISA-ITFKLNR ~  
$ ls -l  
total 0  
-rwxrwx-wx 1 postel None 0  9 déc.  14:08 tmp  
postel@UTILISA-ITFKLNR ~  
$
```

- Il est possible de rediriger le résultat de la commande dans un fichier en utilisant le signe >

ls -l > result



total 0  
-rwxrwx-wx 1 poste1 None 0 9 déc.  
14:08 tmp

- L'intérêt est par exemple de conserver une trace du résultat de la commande dans un fichier texte persistant.

# SCRIPT SHELL

## Les flux : redirection et concaténation de flux dans un fichier

### ➤ Rediriger la sortie standard :

*commande* > resultat

Exemple : `ls > resultat`

### ➤ Rediriger la sortie standard en identifiant le flux à rediriger

*commande* 1>resultat

Exemple : `ls 1>resultat`

### ➤ Rediriger la sortie d'erreur standard

*commande* 2>resultat

Exemple : `ls 2>resultat`

### Rediriger un résultat vers /dev/null (« trou noir »)

*commande* > /dev/null

### ➤ Concaténer les redirections → utilisation du symbole >>

*commande* >> resultat

Exemple : `ls >> resultat`

### Exercice 10 :

1. Dans votre `$HOME` (à vérifier), créer un répertoire `tmp_directory` et ajouter la commande `echo` pour indiquer la création du répertoire dans un fichier `creation.txt`.
2. Concaténez le résultat de la commande `cal` dans le fichier `creation.txt` . Sur la même ligne, réorientez la sortie d'erreur vers `/dev/null`.
3. Sauvegarder les lignes de code pour l'exercice 12.

# SCRIPT SHELL

## Exercices (1/2)

**Exercice 11 (20 minutes) : Exercices sur les commandes utiles Linux (rappel) :**

**En ligne de commande :**

1. Afficher le nom de votre système Linux
2. Fermer le terminal (avec raccourci clavier)
3. Afficher le manuel de la commande grep
4. Lister le contenu (avec les fichiers cachés) du répertoire /var/log
5. Dans votre répertoire \$HOME, en utilisant le chemin absolu, créer un répertoire tmp\_directory (s'il existe le supprimer puis le recréer)
6. Afficher dans la console le contenu du fichier /etc/profile (attention la destruction de ce fichier détruit votre machine)
7. Trouver les fichiers appelés .profile (avec une commande et deux syntaxes différentes)
8. Compter le nombre de lignes du fichier /etc/profile (chercher la commande à utiliser)
9. Indiquez le répertoire où vous vous trouvez
10. Listez les processus tournant
11. Listez les processus liés à l'application bash

### **Exercice 11 (20 minutes) (suite) : Exercices sur les commandes utiles Linux (rappel) :**

12. Dans votre répertoire \$HOME, créer deux répertoires puis les supprimer
13. Créer un fichier texte puis le copier dans le répertoire \$HOME/tmp\_directory
14. Créer en une seule ligne de commande trois nouveaux fichiers et les couper/coller dans le répertoire \$HOME/tmp\_directory
15. Indiquez la taille totale du répertoire tmp\_directory
16. Supprimer le répertoire tmp\_directory
17. Indiquer l'espace disque utilisé
18. Effacer les commandes affichées en console
19. Afficher l'arbre des processus linux
20. Lancer la commande ftp et l'interrompre en utilisant le raccourci clavier

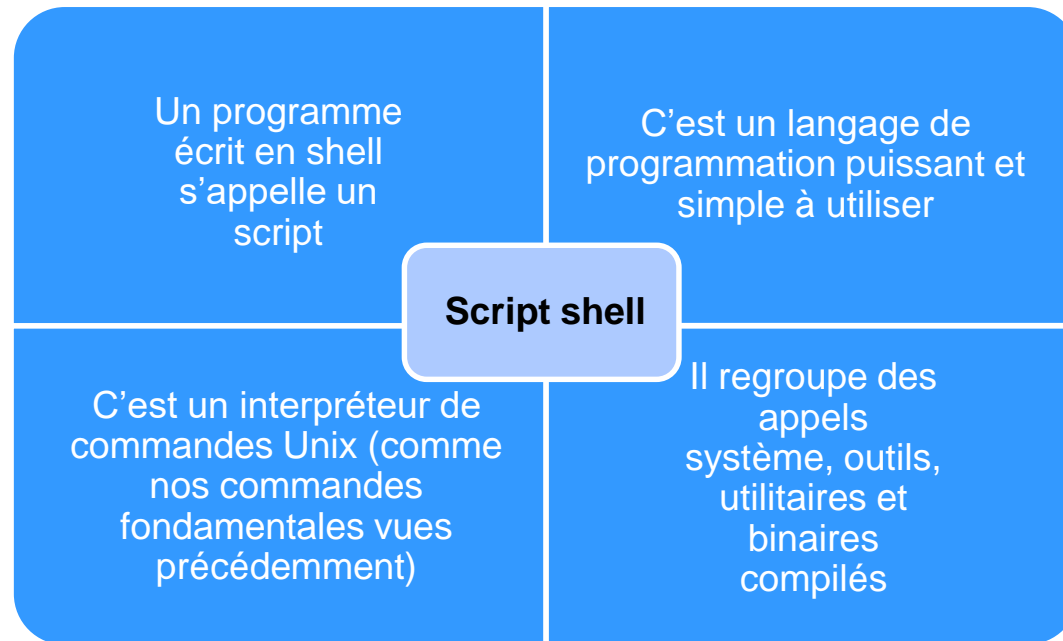
### Partie 3 : Scripting shell





# SCRIPT SHELL

## Scripting shell : C'est quoi un script?



### Les scripts shell sont utilisés pour :

- ✓ Des tâches d'administration système
- ✓ Des routines répétitives qui peuvent être automatisées (programmes simples et faciles à maintenir)

# SCRIPT SHELL

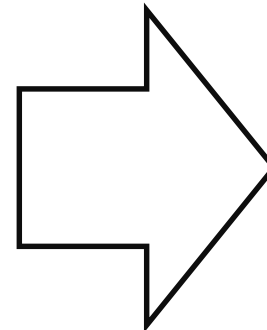
## Scripting shell : créer un script shell basic

**La structure d'un script shell est très simple :**

- ✓ Un **she-bang** en entête : indique au système quel interpréteur shell vous utilisez. Nous utiliserons toujours : `#!/bin/bash`
- ✓ Le corps du script → rien ou des commandes
- ✓ L'enregistrer au format `.sh` (exemple : `mon_script.sh`)
- ✓ C'est tout!!!

mon\_script.sh

```
#!/bin/bash  
  
ma_commande1  
ma_commande2  
ma_commande3
```



mon\_script.sh  
(simple fichier texte  
avec des commandes)

# SCRIPT SHELL

## Scripting shell : exécuter un script shell

### Exécution d'un script shell :

- ✓ Rendre le script shell exécutable en utilisant la méthode relative de `chmod` soit : `chmod +x mon_script.sh`
- ✓ Dans votre terminal exécuter le script de la manière suivante :  
`./mon_script.sh`

### EXERCICE 12 :

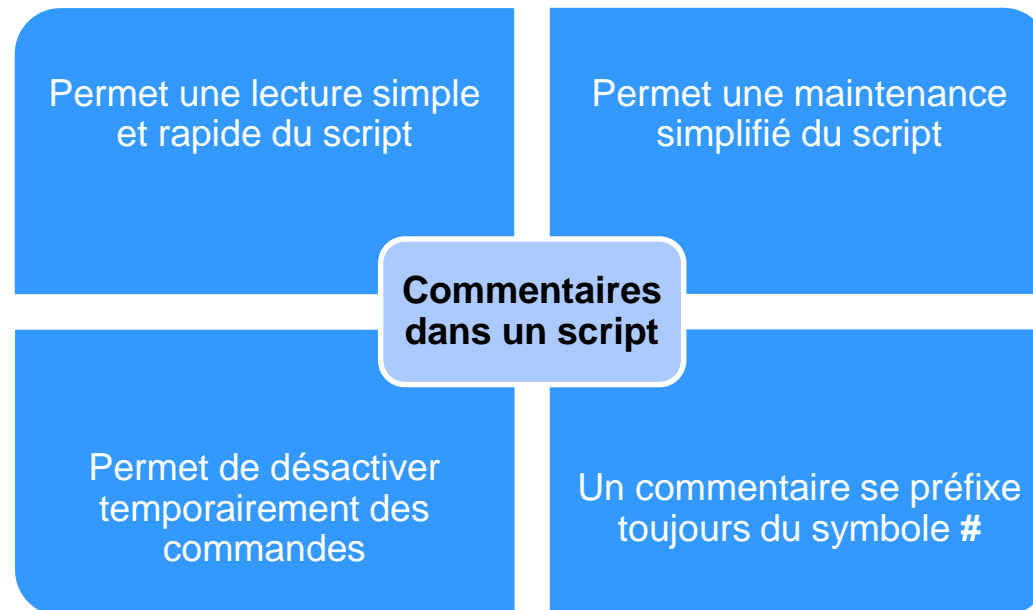
Sous Notepad ++ :

1. Créer un nouveau fichier (dont vous choisissez le nom)
2. Dans l'onglet « Edition », choisissez « convertir les sauts de ligne » puis « Convertir en format UNIX »
3. Dans l'onglet « Langage », choisissez « shell »
4. Taper votre sha-bang
5. Réutiliser le code de l'exercice 10
6. Enregistrer votre script. Exécuter le!

# SCRIPT SHELL

## Scripting shell : ajouter des commentaires dans votre script shell

La bonne pratique veut qu'un script bien fait soit correctement commenté.



# SCRIPT SHELL

## Scripting shell : ajouter des variables dans votre script shell

**Une variable est une valeur qui peut avoir différente(s) valeur(s)**

Par exemple, vous connaissez l'équation  $ax+b=c$ . Toutes les lettres sont des variables où par exemple  $a=1$ ,  $b=2$  et  $c=2$  soit :  $1x+2=2$  donc  $x=0$

- ✓ Dans un script shell, nous pouvons utiliser des variables qui fonctionnent de la même manière.  
Par exemple :  $a = \log_1$  et  $b = \log_2$
- ✓ Une variable permet de modifier une valeur rapidement, d'être mutualisé, de récupérer une valeur
- ✓ Je peux déclarer mes variables en début de script soit :  $a = \log_1$
- ✓ J'utilise mes variables dans le script avec le symbole  $\$$  soit  $\$a$

```
#!/bin/bash
#Variable
a=log_1

#Mes commandes
ls -l > $a
cat $a
```

Testez ce script

Indication : la commande cat affiche dans le terminal le contenu d'un fichier texte.

# SCRIPT SHELL

## Scripting shell : Algorithmique

Un **algorithme** est une suite finie et non ambiguë d'opérations et/ou d'instructions permettant de résoudre un problème (source : Wikipedia).

En shell, les algorithmes sont utilisés dans des scripts et permettent par exemple de créer des **conditions** (si, alors, sinon si...), des **boucles** (while, for, select...)...

L'**intérêt** est par exemple d'offrir plusieurs solutions à un problème (conditions) et de « boucler » jusqu'à que le résultat attendu apparaisse (boucle).

**Pour rappel, voir en annexe (pour votre culture d'informaticien) :**

- Calcul base 2, base 10 et base 16
- Connaitre les puissances de 2 usuelles
- Un Bit et un octet (uniquement pour votre culture)
- Algèbre Boole : et, ou, ou exclusif

# SCRIPT SHELL

## Scripting shell : la condition if

### La structure if :

- La condition if correspond à « si »
- Elle se compose de 5 termes, dont trois obligatoires :
  - if (si) → la condition à remplir
  - then (alors) → les conséquences de la condition à remplir
  - elif (sinon, si) → (facultatif) une conséquence alternative
  - else (sinon) → (facultatif) la condition à remplir si la condition if n'est pas remplie
  - fi → clos la condition si

### Exemple :

```
if condition_1
    then commandes_1
elif condition_2
    then commandes_2
else commandes_3
fi
```

# SCRIPT SHELL

## Scripting shell : les conditions --> les tests (1/5)

La condition **if/then** permet de tester si l'état de sortie d'une liste de commandes vaut 0 (succès) ou autre chose (échec sauf si le code retour est défini autrement).

Le caractère **[** (crochet gauche) est un synonyme de la commande **test**.

Cette commande considère ses arguments comme des expressions de comparaison ou comme des tests de fichiers. Il renvoie un état de sortie qui correspond au résultat de la comparaison : 0 pour succès et 1 pour échec

Syntaxe : `[ $a comparaison_souhaitée $b ]` Refaire cet exemple :

```
#!/bin/sh
```

```
read -p "a = " a
```

```
read -p "b = " b
```

```
# deux variables $a et $b (par exemple deux entiers)
```

```
if [ $a -eq $b ]
```

```
then
```

```
    echo « a est égal à b »
```

```
else
```

```
    echo « a est différent de b »
```

```
fi
```



# SCRIPT SHELL

## Scripting shell : les conditions --> les tests (2/5)

### Tester une variable :

Syntaxe : [ \$a = toto]

### Tests sur les objets du système de fichier :

[ -e \$FILE ] → vrai si l'objet désigné par \$FILE existe dans le répertoire courant

[ -s \$FILE ] → vrai si l'objet désigné par \$FILE existe dans le répertoire courant et si sa taille est supérieure à zéro

[ -f \$FILE ] → vrai si l'objet désigné par \$FILE est un fichier dans le répertoire courant

[ -r \$FILE ] → vrai si l'objet désigné par \$FILE est un fichier lisible dans le répertoire courant

[ -w \$FILE ] → vrai si l'objet désigné par \$FILE est un fichier inscriptible dans le répertoire courant

[ -x \$FILE ] → vrai si l'objet désigné par \$FILE est un fichier exécutable dans le répertoire courant

[ -d \$FILE ] → vrai si l'objet désigné par \$FILE est un répertoire dans le répertoire courant.

# SCRIPT SHELL

## Scripting shell : les conditions --> les tests (3/5)

### Tester des chaînes de caractère :

- [ c1 = c2 ] → vrai si c1 et c2 sont égaux
- [ c1 != c2 ] → vrai si c1 et c2 sont différents
- [ -z c ] → vrai si c est la chaîne vide (*Zero*)
- [ -n c ] → vrai si c n'est pas la chaîne vide (*Non zero*)

### Tests sur les nombres :

- [ n1 -eq n2 ] → vrai si n1 et n2 sont égaux (*EQual*)
- [ n1 -ne n2 ] → vrai si n1 et n2 sont différents (*Not Equal*)
- [ n1 -lt n2 ] → vrai si n1 est strictement inférieur à n2 (*Less Than*)
- [ n1 -le n2 ] → vrai si n1 est inférieur ou égal à n2 (*Less or Equal*)
- [ n1 -gt n2 ] → vrai si n1 est strictement supérieur à n2 (*Greater Than*)
- [ n1 -ge n2 ] → vrai si n1 est supérieur ou égal à n2 (*Greater or Equal*).

### Tester le code retour de la dernière commande (gestion d'erreur) :

- [ \$? -gt 0 ] → la variable \$? retourne code retour de la dernière commande exécutée. Si le code retour est strictement supérieur à 0, alors il y a une erreur. Si le code retour est égal à 0 alors la dernière commande est correcte.

# SCRIPT SHELL

## Scripting shell : les conditions --> les tests (4/5)

### Exercice 13 :

#### Spécifications :

Votre client souhaite un script qui compare deux valeurs entières strictement supérieures à 0 et savoir si elles sont égales, supérieures ou inférieures.

**Règle 1 :** L'utilisateur doit pouvoir saisir deux entiers en entrée (pas de chaîne de caractère)

**Règle 2 :** Le script doit vérifier que les deux entiers saisis sont strictement supérieurs à 0 soit  $a > 0$  et  $b > 0$ . Si ce n'est pas le cas, le traitement s'interrompt.

**Règle 3 :** Le traitement compare a et b et indique si  $a = b$ ,  $a > b$  et  $a < b$

**Règle 4 :** Les utilisateurs doivent être informés de l'avancée du script

# SCRIPT SHELL

## Scripting shell : les conditions --> les tests (5/5)

**Solution proposée pour l'exercice 13 (plusieurs solutions sont possibles)**

```
#!/bin/sh

#Saisie des entiers a et b

echo "Saisissez deux entiers strictement supérieurs à 0 "
read a
read b

#Vérifie que a et b sont deux entiers strictement supérieur à 0
if [ $a -gt 0 ] && [ $b -gt 0 ]; then echo "$a et $b sont des nombres
entiers"
    #Si oui, les entiers a et b sont comparés
    if [ $a -eq $b ] then echo "$a est égal à $b"
    elif [ $a -gt $b ] then echo "$a est plus grand que $b "
    elif [ $a -lt $b ] then echo "$a est plus petit que $b "
    fi #Fin de de la seconde condition si
#Si a et b ne sont pas des entiers strictement positifs, alors le
traitement s'interrompt
else echo "$a et $b ne sont pas des entiers, fin du traitement "
fi #fin de la première condition si
```

# SCRIPT SHELL

## Scripting shell : les conditions select/case (1/2)

### La structure case :

La condition case est utilisée lorsqu'une condition offre plusieurs possibilités.

➤ Syntaxe :

```
case ma_condition in
    "cas_1 ") une_commande_facultative;;
    "cas_2 ") une_commande_facultative;;
esac
```

# SCRIPT SHELL

## Scripting shell : les conditions select/case (1/2)

### La structure case :

La condition case est utilisée lorsqu'une condition offre plusieurs possibilités.

➤ Syntaxe :

```
case ma_condition in
    "cas_1 ") une_commande_facultative;;
    "cas_2 ") une_commande_facultative;;
esac
```

```
read -p "Voulez vous continuer le programme ? " reponse
case $reponse in
    [yYo0]*) echo "Ok, on continue";;
    [nN]*) echo "$0 arrete suite a la mauvaise volonte de l'utilisateur ;-)"
            exit 0;;
    *) echo "ERREUR de saisie"
       exit 1;;
esac
```

# SCRIPT SHELL

## Scripting shell : les boucles --> while

**La boucle while EXECUTE** la série d'instructions entre les mots clés do et done tant que la condition est vrai. La série d'instructions n'est pas exécutée si la condition n'a jamais été vraie

**La boucle until** est identique mais tant que la condition est fausse.

Pour toutes les boucles, un `|` correspond a 'et', deux `||` correspond à 'ou'

➤ Syntaxe :

```
while [test]; do
    echo 'Action en boucle'
done
```

➤ Exemple :

```
#!/bin/bash
while [ -z $reponse ] || [ $reponse != 'OK' ]; do
    read -p 'Dites OK : ' reponse
done
```

# SCRIPT SHELL

## Scripting shell : les boucles --> for (1/3)

La boucle **for** boucle **AUTANT DE FOIS** que permet de traiter la même série d'instructions pour une liste de valeurs. Cette liste de valeurs est précisée dans la commande **for**

Syntaxe en précisant une liste de valeurs :

```
for variable in valeur1 valeur2  
do  
    echo « La variable vaut $variable »  
done
```

➤ Résultats :

La variable vaut valeur1

La variable vaut valeur2



# SCRIPT SHELL

## Scripting shell : les boucles --> for (2/3)

### Exercice 15 :

Créer une boucle for qui liste tous les fichiers contenus dans votre répertoire \$HOME



# SCRIPT SHELL

## Scripting shell : le débogage

Pour déboguer un script, on peut placer la commande `set -x` en amont du code à déboguer. Pour arrêter le débogage, on entre la commande `set +x`.

```
set -x

set +x

#!/bin/sh

#Saisie des entiers a et b
echo "Saisissez deux entiers strictement supérieurs à 0 "

set -x
read a
set +x
read b

#Vérifie que a et b sont deux entiers strictement supérieur à 0
if [ $a -gt 0 ] && [ $b -gt 0 ]; then echo "$a et $b sont des nombres entiers"
    #Si oui, Les entiers a et b sont comparés
    if [ $a -eq $b ] then echo "$a est égal à $b"
    elif [ $a -gt $b ] then echo "$a est plus grand que $b "
    elif [ $a -lt $b ] then echo "$a est plus petit que $b "
    fi #Fin de de la seconde condition si
#Si a et b ne sont pas des entiers strictement positifs, alors le traitement s'interrompt
else echo "$a et $b ne sont pas des entiers, fin du traitement "
fi #fin de la première condition si
```

# SCRIPT SHELL

## Scripting shell : les boucles --> for (3/3)

### Exercice 15 :

Créer une boucle for qui liste tous les fichiers contenus dans votre répertoire \$HOME



# SCRIPT SHELL

## Scripting shell : les boucles --> for (3/3)

### Exercice 15 :

Créer une boucle for qui liste tous les fichiers contenus dans votre répertoire \$HOME

### Solution possible :

```
#!/bin/bash
```

```
liste_fichiers=`ls -a`
```

```
for fichier in $liste_fichiers
```

```
do
```

```
    echo « Fichier(s) trouvé(s) : $fichier »
```

```
done
```

### Partie 4 : Mini projet Station

### Spécifications :

#### 1. Création du répertoire d'exploitation de l'application :

- ✓ Si le répertoire du jeu existe ne pas le créer
- ✓ Si le répertoire du jeu n'existe pas, le créer
- ✓ L'utilisateur doit pouvoir choisir le nom du répertoire

#### 2. Création des questions qui composent le jeu

- ✓ Le questionnaire se compose de trois questions à choix multiples
- ✓ La question 1 possède deux solutions, une solution a et une solution b
- ✓ Si la réponse est a, alors on pose la question 2a
- ✓ Si la réponse est b, alors on pose la question 2b
- ✓ Les questions 2a et 2b ont deux réponses chacune
- ✓ Question 2a : la réponse 1 renvoie « Vous êtes impatient », la réponse 2 renvoie la commande `sl -l`
- ✓ Question 2b : la réponse 1 renvoie « Vous êtes en retard »; la réponse 2 renvoie `sl -a`
- ✓ Tous les résultats aux questions doivent être conservés dans un fichier `result.txt`

### Enoncé :

1. Réaliser le script `station.sh` du jeu et assurez-vous qu'il fonctionne

# SCRIPT SHELL

## Projet Station

### Indications :

#### 1. Essayer d'exécuter le script depuis un script extérieur

#### 1. Création du répertoire d'exploitation de l'application :

- ✓ Vous pouvez utiliser la condition if et la boucle until pour vérifier si le répertoire existe et proposer à l'utilisateur de choisir le nom du répertoire
- ✓ Niveau + : Vous pouvez intégrer cette partie dans une fonction make\_directory

#### 1. Création des questions qui composent le jeu :

- ✓ Vous pouvez utiliser la boucle select
- ✓ Les deux conditions if et case sont utilisables
- ✓ Utiliser la concaténation pour rediriger les réponses dans un fichier texte.

#### Pour aller plus loin (niveau + + +):

- ✓ Ajouter une interface graphique avec la commande dialog
- ✓ Séquencer le script dans des fonctions (function) et les utiliser dans un algorithme

# Partie 5 : Annexes





En 1969, Ken Thompson qui travaille alors pour les laboratoires Bell développa la première version d'un système d'exploitation mono-utilisateur sous le nom de New Ken's System. Il réalisa ce travail sur un mini-ordinateur PDP-7 (Programmed Data Processor) de marque DEC animé par General Comprehensive Operating System[réf. nécessaire] et rédigea le nouveau logiciel en langage d'assemblage. Le nom Unics fut suggéré par Brian Kernighan suite à un jeu de mot « latin » avec Multics; « Multi- car Multics faisait la même chose de plusieurs façons alors qu'Unics faisait chaque chose d'une seule façon ». Ce nom fut par la suite contracté en Unix (pour au final être déposé sous le nom UNIX par AT&T), à l'initiative de Brian Kernighan. Un décret datant de 1962 interdisait à l'entreprise AT&T, dont dépendait Bell Labs, de commercialiser autre chose que des équipements téléphoniques ou télégraphiques. C'est la raison pour laquelle la décision fut prise en 1973 de distribuer le système UNIX complet avec son code source dans les universités à des fins éducatives, moyennant l'acquisition d'une licence au prix très faible

(Source Wikipédia)

# SCRIPT SHELL

## Annexe : Calcul base 2, base 10 et base 16 (1/2)

### La base 2 (binaire) :

- ✓ Souvenez-vous de la commande chmod (méthode absolue)
- ✓ Le système binaire est un système de numérotation utilisant la base 2
- ✓ Par convention , les valeurs sont notées 0 et 1
- ✓ « Vulgairement », le binaire fonctionne comme un interrupteur :
  - 1 j'allume
  - 0 j'éteins



- La calculatrice Windows avec affichage programmeur permet de calculer rapidement en base 2, 10 et 16.

### EXERCICE 12 :

1. Ouvrez la calculatrice Windows
2. Sélectionner Affichage → Programmeur
3. Saisir 10 (en base 10 (Déc))
4. Cocher la case Bin, qu'obtenez vous?

Dec	Bin
1	0
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010

# SCRIPT SHELL

## Annexe : Calcul base 2, base 10 et base 16 (2/2)

### La base 10 (décimale)

- ✓ C'est la base de calcul que nous utilisons au quotidien. Exemple : 1, 2, 3, 4, ..., 1000, ...

### La base 16 (hexadécimale)

- ✓ C'est un moyen pratique de représenter les nombres avec 16 symboles.
- ✓ Le système hexadécimal est un compromis entre le code binaire des machines et une base de numérotation utilisable par l'Homme.
- ✓ Chaque chiffre hexadécimal correspond exactement à quatre chiffres binaires.

<b>0</b> <sub>hex</sub> = <b>0</b> <sub>dec</sub> = <b>0</b> <sub>oct</sub>	0	0	0	0
<b>1</b> <sub>hex</sub> = <b>1</b> <sub>dec</sub> = <b>1</b> <sub>oct</sub>	0	0	0	1
<b>2</b> <sub>hex</sub> = <b>2</b> <sub>dec</sub> = <b>2</b> <sub>oct</sub>	0	0	1	0
<b>3</b> <sub>hex</sub> = <b>3</b> <sub>dec</sub> = <b>3</b> <sub>oct</sub>	0	0	1	1
<b>4</b> <sub>hex</sub> = <b>4</b> <sub>dec</sub> = <b>4</b> <sub>oct</sub>	0	1	0	0
<b>5</b> <sub>hex</sub> = <b>5</b> <sub>dec</sub> = <b>5</b> <sub>oct</sub>	0	1	0	1
<b>6</b> <sub>hex</sub> = <b>6</b> <sub>dec</sub> = <b>6</b> <sub>oct</sub>	0	1	1	0
<b>7</b> <sub>hex</sub> = <b>7</b> <sub>dec</sub> = <b>7</b> <sub>oct</sub>	0	1	1	1
<b>8</b> <sub>hex</sub> = <b>8</b> <sub>dec</sub> = <b>10</b> <sub>oct</sub>	1	0	0	0
<b>9</b> <sub>hex</sub> = <b>9</b> <sub>dec</sub> = <b>11</b> <sub>oct</sub>	1	0	0	1
<b>A</b> <sub>hex</sub> = <b>10</b> <sub>dec</sub> = <b>12</b> <sub>oct</sub>	1	0	1	0
<b>B</b> <sub>hex</sub> = <b>11</b> <sub>dec</sub> = <b>13</b> <sub>oct</sub>	1	0	1	1
<b>C</b> <sub>hex</sub> = <b>12</b> <sub>dec</sub> = <b>14</b> <sub>oct</sub>	1	1	0	0
<b>D</b> <sub>hex</sub> = <b>13</b> <sub>dec</sub> = <b>15</b> <sub>oct</sub>	1	1	0	1
<b>E</b> <sub>hex</sub> = <b>14</b> <sub>dec</sub> = <b>16</b> <sub>oct</sub>	1	1	1	0
<b>F</b> <sub>hex</sub> = <b>15</b> <sub>dec</sub> = <b>17</b> <sub>oct</sub>	1	1	1	1

Source : wikipédia

### L'algèbre booléenne :

- ✓ Structure algébrique qui ne contient que deux éléments : 1 ou 0

### Opérateurs logiques (1/2) :

- ✓ L'opérateur **OU** :

- $a \text{ OU } b$  est VRAI si et seulement si  $a$  est VRAI **ou**  $b$  est VRAI
- On le symbolise par un OR ou  $||$  (en algèbre de Boole par un  $+$ )

- ✓ L'opérateur **XOR** (OU exclusif)

- $a \text{ OU } b$  est VRAI exclusivement si et seulement si  $a$  est VRAI **ou**  $b$  est VRAI
- On le symbolise par un XOR

- ✓ L'opérateur **ET** :

- $a \text{ ET } b$  est VRAI si et seulement si  $a$  est VRAI et  $b$  est VRAI
- On le symbolise par un AND ou  $\&\&$  (en algèbre de Boole par un  $.$ )

### Opérateurs logiques (2/2) :

- ✓ L'opérateur **NXOR** (NON OU exclusif)
  - $a \text{NXOR} b$  si et seulement si  $a$  et  $b$  sont dans le même état logique
  - On le symbolise par un NXOR
- ✓ L'opérateur **NON** :
  - $a$  est VRAIE si et seulement si  $a$  est FAUX
  - On l'utilise souvent dans des conditions par un
  - L'opérateur NON est souvent représenté par un !
    - ✓ L'opérateur **NON ET**
  - combo des opérateurs NON et ET
    - ✓ L'opérateur **NAND** (NON OU)
  - combo des opérateurs NON et OU

### EXERCICE 13 :

- Déterminer par écrit la 'Table de vérité' des opérateurs suivants : OU, ET, NON, XOR, NXOR, NAND, NOR et NON

#### Indications :

- L'opérateur NON n'a qu'une condition soit a et deux résultats attendus
- Les autres opérateurs ont deux conditions soit a et b
- Il existe 4 possibilités pour chaque opérateur (que deux pour NON) :

a	b	a opérateur b
0	0	Résultat 1
0	1	Résultat 2
1	0	Résultat 3
1	1	Résultat 4

# SCRIPT SHELL

## Annexe : un Bit, un octet (uniquement pour votre culture)

### Un Bit

- ✓ Est la contraction de **B**inaryDigit
- ✓ C'est la plus petite unité d'information manipulable par une machine numérique
- ✓ En binaire, cela équivaut à 0 et 1
- ✓ Ne pas confondre Bit et byte (un byte = 8 bits)

### Un octet

- ✓ Un octet correspond à 8 bits (soit 1111 1111 = 1 octet = 255 (en décimal))
- ✓ En anglais on dit *byte* pour octet

### Abréviations

- Mo = MégaOctet
- MB = MégaByte
- Mb = MégaBit

# SCRIPT SHELL

## Annexe : créer sa propre commande shell

### Créer sa propre commande shell :

1. Créer un script.sh
2. Le copier dans le répertoire /usr/local/bin
3. Taper le nom sans ./ Le script est maintenant considéré comme une commande à part entière