

# Introduction à la programmation Shell

Pape Abdoulaye BARRO

---

Enseignant-chercheur  
Expert en Télémétrie et Systèmes Intelligents  
UIDT, EPT, Réseaux des E-Lab

# PLAN

---

- Généralités
- Exécution et environnement Shell
- Manipulation des variables
- Tests: les instructions if et case
- Boucles: les instructions for, while, et until
- Les tableaux
- Les fonctions

# PLAN

---

- Généralités
- Exécution et environnement Shell
- Manipulation des variables
- Tests: les instructions if et case
- Boucles: les instructions for, while, et until
- Les tableaux
- Les fonctions

# Généralités

## Définitions

- Le Shell est un interpréteur de commandes (un programme informatique) destiné aux *systèmes d'exploitation Unix* (type Unix) et permettant d'accéder aux fonctionnalités du système.
  - Il se présente sous la forme d'une interface en *ligne de commande* accessible depuis un terminal.
    - L'utilisateur lance des *commandes* sous forme d'une entrée texte exécutée ensuite par le Shell.
    - Shell est un mot anglais signifiant enveloppe en français pour simplement désigner la couche la plus haute de toutes les interfaces des systèmes Unix (Linux, macOS).
      - À l'origine, aux tout débuts de l'informatique, avant l'utilisation des interfaces graphiques (macOS ou Windows), le shell (sh) était la seule interface utilisateur disponible sur un système de type Unix. Aujourd'hui, nous assistons à la naissance de nombreuses variantes, telles que le csh (C Shell), étendu en tcsh, ou ksh (qui tourne aussi sur windows), le bash plus répandu (qui s'inspire de sh, ksh, et csh), et autres(zsh, ash, ...).
      - Le Shell peut être utilisé de manière avancée en combinant des *commandes* soit de manière interactive ou sur un fichier *exécutable* d'extension .sh appelé script, dans le but d'automatiser certains tâches.

# Généralités

## Définitions

- Un **système d'exploitation** (SE) encore appelé Operating System (OS) en anglais est un ensemble de programmes qui dirige l'utilisation des ressources (processeurs, mémoire et périphériques) d'un appareil électronique (ordinateur) par des logiciels (ou applications). Il joue le rôle d'ordonnanceur entre l'utilisateur, les ressources disponibles et les applications compatibles.
  - On les retrouve:
    - Dans les ordinateurs pour faire fonctionner les claviers, les souris, les écrans, l'utilisation des mémoires, ainsi que les différents calculs à effectués.
    - Dans d'autres appareils tels que les appareils photo numériques et autres afin de faire fonctionner tous les différents mécanismes ou fonctionnalités qu'ils constituent.
  - Les systèmes d'exploitation les plus répandus sont Windows (PC), Mac OS (PC), Linux (PC et serveurs) et Unix (pour les serveurs) pour les PC et/ou serveur. Nous avons aussi Android, iOS, Symbian, Windows Phone pour les téléphones et Raspbian pour Raspberry Pi.
  - Nous chercherons à installer Ubuntu ou Debian pour des besoins de tests.

# Généralités

## Définitions

---

● **Unix** est un système d'exploitation multi-utilisateurs et multitâches, permettant à un ordinateur de faire exécuter simultanément plusieurs programmes par un ou plusieurs utilisateurs.

- Il a été initié par Ken Thompson et Dennis Ritchie et est écrit en C.
- Initialement à but non commercial, Unix a été développé vers les années 70 dans les laboratoires Bell (AT&T) et à donner naissances à plusieurs distributions dont les plus célèbres sont **Linux**, macOS, iOS, et Android.
- Il possède un ou plusieurs interpréteurs de commandes (dont shell) ainsi qu'un grand nombre de commandes et de nombreux utilitaires (assembleur, compilateurs pour de nombreux langages, traitements de texte, messagerie électronique, ...).

# Généralités

## Définitions

- Une **ligne de commande** en Unix désigne une séquence de texte taper par un utilisateur, dans un terminal, interprétée par un Shell, pour interagir avec un système informatique et non à l'aide d'une interface graphique.
  - Avant l'émergence des PC, l'informatique était encore faite de lignes de commande.
  - La ligne de commande est analysée et découpée en arguments en utilisant des caractères spéciaux comme séparateurs.
- Une **commande** est donc une instruction qu'un utilisateur envoie au système d'exploitation afin de lui faire exécuter une tâche donnée.
  - Il existe un très grand nombre de commande selon la complexité du système d'exploitation et peut être accompagnée d'options et de paramètres.
- Un **exécutable Shell**, encore appelé **Script Shell** est un fichier contenant des commandes écrit en Shell.
  - Il peut être utilisé pour automatiser un certains nombres de taches. Il peut également être utiliser avec Python, PHP, JSP ou autres.

# Généralités Historiques

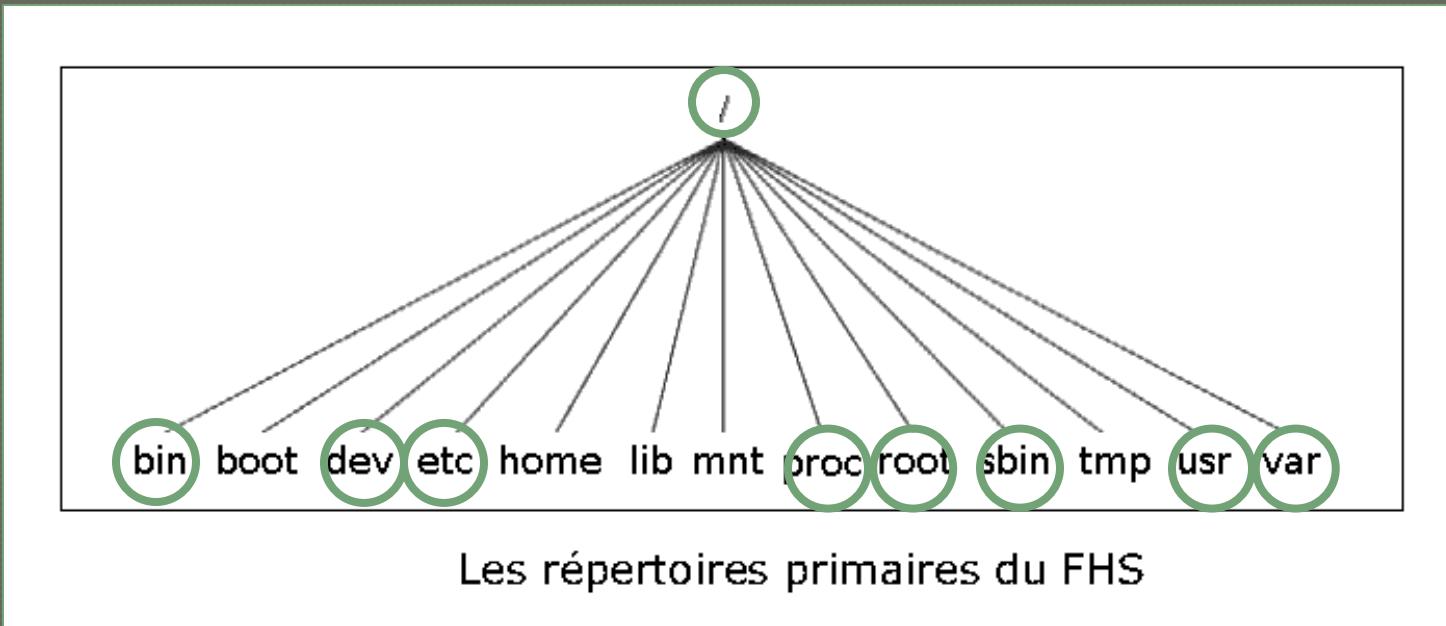
---

- En 1971, Ken Thompson, l'un des créateurs d'Unix, a écrit le tout premier **shell** (avec la première version d'Unix) qui est remplacé en 1977 par le **Bourne shell** ou **sh**, écrit par Stephen Bourne (avec la version 7 d'Unix).
- En 1978, Bill Joy (étudiant à l'époque), crée le **C shell** ou **csh**, une évolution du shell qui ressemble syntaxique au C. Nous avons aussi **tcsh** autre version plus moderne.
- En 1983, David Korn proposa le **Korn shell** ou **ksh**, qui reprend certaines fonctionnalités de **csh** en y ajoutant quelques fonctions avancées.
- En 1988, nous assistons à la naissance du **bash** (Le **Bourne-Again shell**), une version écrit par Brian Fox dans le cadre du projet GNU. C'est le shell le plus utilisé aujourd'hui.
- En 1990, nous avons le **Z shell** ou **zsh**, écrit par Paul Falstad (étudiant à l'époque), qui est juste une évolution du sh avec des retouches sur les autres (csh, ksh et tcsh).

# Généralités

## Linux et arborescences

- Linux est une version libre d'UNIX, qui s'est beaucoup inspiré des autres distributions payant pour mettre en place un système d'exploitation robuste.
- Les systèmes Linux respectent la norme FHS (File Hierarchy Standard) afin d'assurer sa compatibilité ainsi que sa portabilité. Cette *hiérarchie de base* est la suivante :



# Généralités

## Linux et arborescences

- **/** - la racine ou root: racine du système de fichiers, ne contient habituellement aucun fichier.
- **/boot** – il contient la base même du système, kernels et boot maps, ainsi que les fichiers de configuration pour grub et les fichiers de ressources pour lilo.
- **/bin** – contient les binaires exécutables des utilitaires de base du système. Des commandes pour démarrer en 'single'. Ces binaires sont compilés en dynamique, ils nécessitent donc la présence des 'libraries' correspondantes.
- **/sbin** - pour Static Binaries, contient des programmes nécessaires au fonctionnement du système, beaucoup sont réservés à 'root' : mkfs, fsck, lilo, agetty.
- **/lib** - 'Shared libraries' (libc.so, libtermcap.so, ...). Librairies partagées nécessaires aux programmes de démarrage situés dans /bin. Il contient aussi les modules du kernel.
- **/dev** – le descripteurs de périphériques (devices).
- **/etc** – on y trouve des fichiers de configuration, des données pour les programmes: passwd, group, inittab, fstab, profile, ..., et différents sous-réertoires, dont :
  - **/rc.d** – contenant des scripts 'run commands': rc.S, rc.local, rc.keymap, ...
  - **/skel** – pour Skeleton. Squelette des scripts de configuration 'users': .inputrc, .bash\_profile... recopiés dans le HOMEDIR d'un 'new user' à la création d'un nouveau compte avec la commande 'adduser'.
  - **/X11** – fichier de configuration pour Xwindow, Xfree86, ...
  - **/modules** – contenant les fichiers de configuration des modules du kernel

# Généralités

## Linux et arborescences

- ⦿ **/usr** – pour Unix System Resources. Contient l'essentiel des ressources de votre système. Il contient une arborescence complète de données que les utilisateurs peuvent se partager. Il peut être le point de montage d'un disque autre que celui contenant la racine. Il peut être monté en réseau et partagé entre plusieurs machines. Ses principaux sous-répertoires sont:

- **/include** – qui contient les fichiers 'headers' .h pour gcc: print.h, malloc.h, gpm.h, ...
- **/lib** – qui contient les 'Libraries' associées aux exécutables de /usr/bin
- **/lib/X11** - lien symbolique vers /usr/X11R6/lib/X11 /kbd/keytables, consolefonts, ...
- **/share** – qui contient les ressources partagées pour les exécutables dans /usr/bin
- **/bin** – qui contient les exécutables, compilés en statique, des programmes utilisables sur le système. C'est juste des utilitaires associés à votre distribution et installés par votre gestionnaire de paquetages.
- **/bin/X11** - Lien symbolique vers /usr/X11R6/bin qui contient les exécutables pour Xwindow
- **/local** - Reproduit l'arborescence de /usr mais il n'est pas géré par le gestionnaire de paquetages. Il contient les programmes installés à partir de sources ou disposant de leur propre procédure d'installation. Programmes spécifiques au système ou au site et indépendants d'une distribution.
- **/src** – il contient les sources des applications installées, **/usr/src/linux** contient les sources du kernel.
- **/man** – manuels en ligne: pages man
- **/spool** – contient les fichiers intermédiaires: cron, lpd, uucp, mqueue, gestion des imprimantes
- **/etc** – qui contient des fichiers de configuration pouvant être partagés entre plusieurs machines, généralement des liens symboliques vers /etc/...
- **/X11R6** - Dédié à Xwindow, un lien symbolique est habituellement créé vers /usr/X11, parfois /usr/X386 ,depuis ce répertoire
- **/X11R6/bin** – contenant les exécutables pour le système Xwindow
- **/X11R6/lib** – contenant les librairies pour les programmes situés dans /usr/X11R6/bin
- **/X11R6/lib/X11** – contenant les librairies utiles au démarrage du serveur Xwindow
- **/X11R6/include/X11** – contenant les fichiers 'header' pour le développement d'applications X11

# Généralités

## Linux et arborescences

- ◎ **/var** – contient des données variables, par exemple: des logs de l'activité système, queues, crontabs des users, games scores, ... Ses principaux sous-répertoires sont:
  - **/lock** – qui contient les fichiers de verrouillage: lock, LCK..
  - **/log** – Journal de l'activité système
  - **/spool** – Files d'attente
  - **/spool/cron** - Entrées de crontab pour l'automatisation des tâches
  - **/spool/lpd** – qui contient en attente d'impression
  - **/spool/mail** – Mailboxes et messages des utilisateurs
- ◎ **/mnt** – c'est le point de montage par la commande 'mount' , avec différents sous- répertoires:
  - **/floppy** - pour le floppy, par exemple /dev/fd0
  - **/cdrom** - pour le CDROM ou DVD
  - **/disk** - pour un disque amovible (mémoire USB, carte SD)
- ◎ **/média** – (dans les distributions basées sur Debian) c'est le point de montage des supports de données amovibles
- ◎ **/tmp** – qui contient les fichiers temporaires
- ◎ **/root** – c'est le répertoire personnel de l'administrateur système. Il contient les programmes et scripts écrits par root pour le système.
- ◎ **/home** – c'est la racine des répertoires personnels des utilisateurs
- ◎ **/opt** – c'est le répertoire des logiciels optionnels, il a sa propre structure. Chaque application y a son propre sous-répertoire lequel contient tout ce qui est nécessaire à son exécution.
- ◎ **/proc** – c'est réservé au système. Le kernel place ici des infos relatives à l'état du système et des process. Il n'occupe aucun espace sur le disque.
- ◎ **/lost+found** - strictement réservé au système aussi. Il est utilisé pour la reconnection d'inodes en cas de crash pendant une création de fichier.

# Généralités

## Permissions et droits d'accès

- Les permissions UNIX constituent un système de définition des droits d'accès aux ressources, représentées par des fichiers disponibles sur un système informatique. Elles restent le moyen le plus utilisé pour définir les droits des utilisateurs sur les systèmes de type UNIX.
  - Toute personne ou programme devant interagir avec un système UNIX doit être authentifié par un utilisateur ou user.
  - Un utilisateur est reconnu par son nom et un numéro unique. Cette correspondance nom/numéro est stockée dans le fichier `/etc/passwd`.
- Etant donné que tous les utilisateurs n'ont pas les même droit, c'est à dire ils ne peuvent pas faire la même chose, la gestion et la sécurité sont renforcées.
- Certains utilisateurs peuvent ne pas s'authentifier sur l'ordinateur et accéder à un interpréteur de commandes. Il leur sera possible de lire ou écrire des fichiers mais cela nécessite que le super-utilisateur démarre un programme pour cet utilisateur.
- Sur tout système UNIX, il y a un super-utilisateur, appelé *root*, qui a tous les pouvoirs. Il peut accéder librement à toutes les ressources de l'ordinateur, y compris à la place d'un autre utilisateur.

# Généralités

## Permissions et droits d'accès

- Un utilisateur UNIX appartient à un ou plusieurs **groupes**. Ils servent à rassembler des utilisateurs afin de leur attribuer des droits communs. Par exemple, sur un système doté d'une carte son, il y a souvent un groupe audio qui regroupe les utilisateurs autorisés à en faire usage.
- En UNIX, tout fichier possède un propriétaire. Seuls le propriétaire du fichier ou le super utilisateur (root) peuvent changer les droits.
- Un fichier UNIX peut aussi appartenir à un groupe. On définit ainsi les actions du groupe sur ce fichier.
- Les droits sur un fichier UNIX s'attribuent sur trois « actions » différentes possibles:
  - **r(read)** – lecture: on peut lire le fichier. Si ce droit est alloué à un répertoire, il autorise l'affichage du contenu du répertoire.
  - **w(write)** – écriture: on peut modifier le contenu du fichier. S'il est alloué à un répertoire, il autorise la création, la suppression et le changement de nom des fichiers qu'il contient, quels que soient les droits d'accès des fichiers de ce répertoire.
  - **x(execute)** – exécution: on peut exécuter un fichier exécutable. Lorsqu'il est attribué à un répertoire, il autorise l'accès au répertoire.

# Généralités

## Permissions et droits d'accès

- Si on désir connaitre les permissions sur les fichiers d'un répertoire, on peut d'abord les lister en faisant: `ls -l`

- Les droits d'accès apparaîtront alors sous forme d'une liste de symboles. Exemple: `drwxr-xr-x`
- Le premier symbole peut être `-`, `d`, soit `l`, `entres autres`. Ces symboles indiquent la nature du fichier:
  - `-` : fichier classique,
  - `d` : directory : répertoire,
  - `l` : link : lien symbolique,
  - `c` : character : périphérique de type caractère,
  - `b` : block : périphérique de type bloc,
  - `p` : pipe : tube, tuyau ou file (d'attente),
  - `s` : socket
- Les symboles qui suivent, regroupés en 3 groupes de 3, indiquent si le fichier (ou répertoire) est autorisé en lecture, en écriture ou en exécution. Ces 3 groupes correspondent, dans cet ordre, aux droits du propriétaire, du groupe puis du reste des utilisateurs. Si la permission n'est pas accordée, la lettre en question est remplacé par « `-` ».
- D'après l'exemple précédent: `drwxr-xr-x`
  - `d` : c'est un répertoire.
  - `rwx` pour le 1er groupe de 3 symboles : son propriétaire peut lire, écrire et exécuter.
  - `r-x` pour le 2e groupe de 3 symboles : le groupe peut uniquement lire et exécuter le fichier, sans pouvoir le modifier.
  - `r-x` pour le 3ème groupe de 3 symboles : le reste du monde peut uniquement lire et exécuter le fichier, sans pouvoir le modifier.

# Généralités

## Permissions et droits d'accès

- Sachant qu'un fichier a un propriétaire et un groupe, il est possible de changer ces permissions. En ligne de commande, nous avons:
  - On peut utiliser la commande `chown` pour changer le propriétaire:
    - `sudo chown mademba nomFichier` # le fichier appartient désormais à l'utilisateur mademba.
  - On peut utiliser la commande `chgrp` pour changer le groupe:
    - `sudo chgrp mesAmis notreFichier` # le fichier appartient désormais au groupe mesAmis.
  - On peut, avec la commande `chown`, simultanément changer le propriétaire et le groupe:
    - `sudo nouveauPropriétaire:nouveauGroupePropriétaire nomFichier`

# Généralités

## Permissions et droits d'accès

- On peut aussi utiliser la commande `chmod` pour modifier les permissions sur un fichier. Il peut s'employer de deux façons équivalentes:
  - soit en ajoutant ou en retirant des permissions à une ou plusieurs catégories d'utilisateurs à l'aide des symboles **r**, **w** et **x**, que nous avons présentés plus haut. Chaque droit peut être gérer séparément:
    - À qui s'applique le changement
      - **u** (**user**, utilisateur) représente la catégorie "propriétaire" ;
      - **g** (**group**, groupe) représente la catégorie "groupe propriétaire" ;
      - **o** (**others**, autres) représente la catégorie "reste du monde" ;
      - **a** (**all**, tous) représente l'ensemble des trois catégories.
    - La modification que l'on veut faire
      - **+** : ajouter
      - **-** : supprimer
      - **=** : affectation
    - Le droit que l'on veut modifier
      - **r** : `read` ⇒ lecture
      - **w** : `write` ⇒ écriture
      - **x** : `execute` ⇒ exécution
      - **X** : `eXecute` ⇒ exécution (concerne uniquement les répertoires et les fichiers qui ont déjà une autorisation d'exécution pour l'une des catégories d'utilisateurs).

Exemple:

- `chmod o-w fichier` # enlèvera le droit d'écriture pour les autres.
- `chmod a+x` # ajoutera le droit d'exécution à tout le monde.

# Généralités

## Permissions et droits d'accès

- soit en précisant les permissions de manière octale, à l'aide de chiffres. Dans ce cas, chaque « groupement » de droits (pour user, group et other) sera représenté par un chiffre et à chaque droit correspond une valeur :

- **r** (read) = 4
- **w** (write) = 2
- **x** (execute) = 1
- **-** = 0

### Exemple:

- Pour **rwx**, on aura :  $4+2+1 = 7$
- Pour **rw-**, on aura :  $4+2+0 = 6$
- Pour **r--**, on aura :  $4+0+0 = 4$
- Cette permission, **drwxr-w---**, donnera en octal, **750**.
- Sur un répertoire, on peut faire: **chmod 750 monRépertoire**

### les combinaisons :

- 0 : --- (aucun droit)
- 1 : - - x (exécution)
- 2 : - w - (écriture)
- 3 : - w x (écriture et exécution)
- 4 : r - - (lecture seule)
- 5 : r - x (lecture et exécution)
- 6 : r w - (lecture et écriture)
- 7 : r w x (lecture, écriture et exécution)

# Généralités

## Commandes usuelles

- On peut lister quelques commandes principales:
  - `ls` permet de lister le contenu d'un répertoire. Il embarque beaucoup d'options à savoir:
    - `-a` pour afficher tous les fichiers, y compris les fichiers cachés;
    - `-l` pour afficher un listing détaillé;
    - `-R` pour afficher les fichiers récursivement;
    - `-d` pour afficher uniquement les répertoires et non leur contenu;
    - `-S` pour trier par taille;
    - `-t` pour trier par date de dernière modification;
    - `-X` pour trier par ordre alphabétique de l'extension;
    - `-r` pour trier en ordre inverse.
  - `cd` permet de changer de répertoire. Il s'emploie avec quelques options aussi:
    - `cd` pour revenir au répertoire de l'utilisateur;
    - `cd -` pour revenir au répertoire précédent;
    - `cd ..` pour revenir au répertoire parent.
  - `mkdir` pour créer un nouveau répertoire
  - `rmdir` permet de supprimer un répertoire. Ses options sont:
    - `-f` pour forcer la suppression;
    - `-R` pour supprimer récursivement;

# Généralités

## Commandes usuelles

- **cp** permet de copier un fichier;
- **mv** permet de déplacer un fichier;
- **rm** permet de supprimer un fichier. Il a pour options:
  - **-f** pour forcer la suppression;
  - **-R** pour supprimer récursivement;
- **pwd** permet d'afficher le chemin complet du répertoire de travail en cours;
- **passwd** permet de changer le mot de passe de l'utilisateur;
- **cat** permet de concaténer deux fichiers et d'afficher le résultat sur la sortie standard;
- **more** permet d'afficher le contenu du fichier avec des pauses;
- **file** permet d'afficher le type présumé du fichier spécifié;
- **chmod** (déjà abordée)
- **clear** permet d'effacer les lignes affichées sur le terminal;
- **traceroute** permet de tracer le chemin entre la machine locale et la machine visée;
- **ping** permet de savoir si une machine répond sur le réseau;
- **talk** permet de parler à un utilisateur connecté;
- **mesg** permet d'autoriser ou non la commande talk:
  - **mesg n** : Empêche la réception de messages talk;
  - **mesg y** : Permet la réception de messages talk;
- **exit** pour se déconnecter

# Généralités

## Utilisation de vi

- Vi (**visual**) est un éditeur de fichiers qui contiennent des lignes de texte et fonctionnant en mode écran. Il embarque :

- quelques commandes essentielles pour :
    - le démarrage de l'éditeur: **vi nomFichier**;
    - la sauvegarde : **:w nomFichier**;
    - quitter l'éditeur en sauvegardant le fichier: **:x**
    - quitter sans sauvegarder : **:q!**
  - des commandes de base pour:
    - Entrer en mode insertion
      - **a** pour ajouter du texte à la droite du curseur;
      - **i** pour insérer du texte à la gauche du curseur;
      - **o** pour intercaler une ligne vide au-dessous du curseur;
      - **O** pour intercaler une ligne vide au-dessus du curseur;
      - **ECHAP** (ou **ASC**) pour revenir au mode commande
    - Pour remplacer du texte on fait:
      - **r** (ici, le caractère tapé sera remplacé par le caractère pointé par le curseur);
      - **R** remplace plusieurs caractères ;
      - **ECHAP** (ou **ESC**) pour revenir au mode commande;

# Généralités

## Utilisation de vi

- Pour déplacer le curseur dans le texte:
  - On peut utiliser les flèches pour se déplacer d'un caractère vers la gauche ou la droite, ou d'une ligne vers le haut ou le bas [ou utiliser les touches **h** (gauche), **j** (bas), **k** (haut), **l** (droite)]
  - Par ligne:
    - **0** pour se positionner au début de la ligne;
    - **\$** pour se positionner à la fin de la ligne;
    - **retour** pour se positionner au premier mot de la ligne suivante
  - D'un écran :
    - **CTRL** et **f** pour avancer d'un écran;
    - **CTRL** et **b** pour reculer d'un écran;

# Généralités

## Utilisation de vi

➤ Pour aller à une ligne en particulier :

- **1G** permet de positionner le curseur au début du fichier;
- **#G** permet de positionner le curseur à la ligne #;
- **G** permet de positionner le curseur à la dernière ligne du fichier;
- **CTRL** et **g** permet de révéler le numéro de la ligne courante;
- **:set nu** permet d'afficher les numéros de lignes;

➤ Pour enlever, remplacer ou copier une partie du texte :

- **X** permet de détruire le caractère pointé par le curseur et de le placer dans le tampon;
- **#X** permet de détruire # caractères et de les placer dans le tampon;
- **dd** permet de détruire la ligne courante et de le placer dans le tampon;
- **#dd** permet de détruire # lignes à partir de la ligne courante et de le placer dans le tampon;
- **yy** permet de copier la ligne courante dans le tampon;
- **#yy** permet de copier # lignes consécutives dans le tampon;
- **p** permet d'insérer le contenu du tampon à la droite du curseur;
- **P** permet d'insérer le contenu du tampon à la ligne précédente;

# Généralités

## Les commandes internes et externes

---

En Unix, on distingue deux sortes d'implémentation pour les commandes: les commandes internes (`cd` , `echo` , `for` , `pwd`, ...) et les commandes externes (`ls`, `mkdir`, `cat`, `sleep`, ..., et les fichiers Shell).

- La commande interne est exécutée en priorité et est la plus rapide. Avec la commande `type`, on peut s'avoir si la commande est interne mais ne renseigne pas si elle est externe aussi.
  - La commande `echo` est une commande interne:
    - `type -t echo` (builtin -> echo est une commande interne du Shell)
- Une commande externe est une commande dont le code se trouve dans un fichier ordinaire. Le Shell crée un processus pour exécuter une commande externe.
  - La commande `sleep` est une commande externe :
    - `type -t sleep` (file -> sleep est une commande externe)

# Généralités

## L'affichage à l'écran

- On peut utiliser la commande `echo` pour afficher un message à l'écran:
  - `echo hello word.`
- Si la phrase doit contenir un apostrophe, il va falloir l'échapper avec un antislash:
  - `echo c\'est pour moi.`
- Il faut aussi l'échapper s'il doit contenir des guillemets:
  - `echo il habit a \"Kirikoro\"`
- Le double antislash "`\\"`" permet d'afficher un antislash:
  - `echo -e "je suis un antislash \\\"`
- Le caractère "`\n`" sert à provoquer un saut de ligne:
  - `echo -e "je suis \n un saut de ligne\\\"`
- Ci-dessous, la liste de quelques caractères spéciaux:
  - `\\" Antislash`
  - `\a Bip`
  - `\b Effacement du caractère précédent`
  - `\c Suppression du saut de ligne en fin de ligne`
  - `\f Saut de page`
  - `\n Saut de ligne`
  - `\r Retour chariot`
  - `\t Tabulation horizontale`
  - `\v Tabulation verticale`

# Généralités

## Les processus en avant-plan et en arrière-plan

- En Unix, par défaut, une commande est exécuté en avant-plan. Dans ce cas, le Shell crée un processus enfant et attend qu'il se termine. Ce qui veut dire que les exécutions se font *séquentiellement*.
- Une commande peut aussi être utilisée en arrière-plan en utilisant un "&" à la fin de la commande. Les exécutions se font *simultanément*.
- Si on lance un processus prenant du temps à s'exécuter, et on ferme la console ou on se déconnecte de la session, le processus (en avant-plan ou en arrière-plan) s'arrête automatiquement car elle est attaché à la console. Pour éviter cela, il faut utiliser la commande **nohup** en tache de fond (arrière-plan).
  - **nohup sleep 30000 &**
- On peut vérifier les processus à l'aide de la commande **ps**.
- Si la commande persiste et que ce n'est pas lancer en arrière-plan, or on a besoin de continuer sur autre chose. On peut interrompre son exécution avec la combinaison de touche **Ctrl+Z** et la relancer en arrière plan, à l'aide de la commande **bg**.
  - **sleep 30000**
  - **jobs** (pour vérifier si la taches est en avant-plan ou en arrière-plan)
    - ❖ **Ctrl+Z** (pour arrêter l'exécution)
  - **jobs** (pour vérifier si la taches est en avant-plan ou en arrière-plan)
  - **bg** (pour basculer en arrière-plan)
  - **jobs**

# Généralités

## Le regroupement des commandes

---

- On peut exécuter séquentiellement des commandes en les séparent par des ";":
  - `ls ; pwd ; date`
- On effectuer un regroupement avec "()" ou avec "{}" pour rediriger la sortie standard des commandes au même endroit:
  - `( date ; ls ) > myFile`
  - `cat myFile`
- Si on souhaite imbriquer des commandes, il est possible d'utiliser "\$()" pour cela. Dans ce cas, la commande sera remplacée par son résultat:
  - `echo nous sommes le $(date)`

# Généralités

## cas pratique

### Promenade dans le système de fichiers:

- Entrer la commande `pwd` (pour « print working directory », c'est-à-dire, afficher le nom du répertoire courant) et noter ce qui est imprimé à l'écran : c'est le chemin absolu de votre home, répertoire personnel;
- Entrer successivement les commandes `cd ..` (avec un espace entre `cd` et `..`) et `pwd`, jusqu'à ce que le résultat ne change plus. Commenter.
- Entrer la commande `cd` (sans argument), puis `pwd`. Commenter.
- Entrer la commande `cd /`, puis `pwd` et `ls`.
- Entrer la commande `cd /usr/include`. Utiliser la commande `ls`. À quoi semble servir ce répertoire ?
- La commande `cat` (pour « concatenate ») permet d'afficher un ou plusieurs fichiers donnés en argument (à la suite) dans le terminal. La commande `wc` (pour « word count ») affiche (dans cet ordre) le nombre de lignes, de mots et de caractères des fichiers donnés en argument, puis, s'il y en a plusieurs, les sommes de ces nombres pour tous les fichiers. Afficher le contenu le nombre de lignes du fichier `stdlib.h`. Donner le nombre total de lignes des fichiers `stdlib.h` et `stdio.h`.
- Entrer les commandes `cd ..`, `pwd` puis `ls`.
- Entrer les commande `cd share/man`, puis `pwd` et `ls`. Pouvez-vous deviner ce que désignent certains des résultats affichés ?
- Entrer la commande `ls /bin`. Certains noms vous sont-ils familiers ?
- Entrer la commande `echo ~`, puis la commande `cd ~`. Qu'a fait le shell au caractère `~` ?

# PLAN

---

- Généralités
- Exécution et environnement Shell
- Manipulation des variables
- Tests: les instructions if et case
- Boucles: les instructions for, while, et until
- Les tableaux
- Les fonctions

# Exécution et environnement Shell

## définition

---

- Un script est un fichier (exécutable) contenant une ou plusieurs commandes qui seront exécutées de manière séquentielle.
  - Il permet de regrouper :
    - des commandes simples pour réaliser des actions complexes;
    - des actions que l'on désire exécuter plus d'une fois.
  - Il permet également de réaliser un outil à mettre à la disposition de plusieurs personnes.
- Plusieurs interpréteurs de commandes existent pour commander des systèmes ou autres en utilisant des langages différents comme Perl, Python, Bash, Tcsh, ... il va falloir alors indiquer au système de quel genre de script s'agit il pour lui permettre de le reconnaître comme exécutable et de pouvoir interpréter les commandes. Il suffit dans ce cas, de préciser au tout début du script (une ligne nommée "*shebang*") la nature du script. Il existe de nombreux interpréteurs possibles, par exemple:

- `#!/bin/sh`
- `#!/bin/bash`
- `#!/bin/tcsh`
- `#!/usr/bin/perl`
- `#!/usr/bin/tcl`

# Exécution et environnement Shell

## exécution

- Pour pouvoir exécuter un fichier sous Unix, il faut d'abord lui attribuer le droit d'exécution. En utilisant la commande `chmod`, on pourra faire:
  - `chmod +x script.sh`
  - `ls -l` (pour vérifier les droits)
- Une fois que nous avons notre exécutable, le fichier pourra alors être exécuter de plusieurs manières différentes.
  - En utilisant `./` (si l'on se trouve dans le répertoire du script) :
    - `./script.sh`
  - En spécifiant le chemin absolu (si le script se trouve dans le répertoire `/home/pape`):
    - `/home/pape/script.sh`
  - En modifiant la variable d'environnement `PATH` et d'y faire figurer le répertoire qui contient le script à exécuter:
    - `PATH=$PATH:/home/pape/`
    - `script.sh`
  - Ou en appelant directement l'interpréteur et lui transmettre le script à exécuter:
    - `bash script.sh`

# Exécution et environnement Shell

## exemple de script

- Exemple:

```
➤#!/bin/bash  
➤# test1.sh : ceci est un commentaire  
➤echo "Hello World"  
➤exit
```

# Exécution et environnement Shell

## LES CARACTÈRES SPÉCIAUX

- Des caractères spéciale sont très utilisées lors de la création de scripts et joue le rôle de variables prépositionnées. Ils permettent:
  - De récupérer les paramètres transmis sur la ligne de commande;
  - De savoir si une commande a échoué ou pas;
  - D'automatiser le traitement de tous paramètres.
- **Ci-dessous, la liste de paramètres prépositionnées:**
  - \$0 : nom du script. Plus précisément, il s'agit du paramètre 0 de la ligne de commande;
  - \$1, \$2, ..., \$9 : respectivement premier, deuxième, ..., neuvième paramètre de la ligne de commande;
  - \$\* : tous les paramètres vus comme un seul mot;
  - \$@ : tous les paramètres vus comme des mots séparés : "\$@" équivaut à "\$1" "\$2" ...
  - \$# : nombre de paramètres sur la ligne de commande
  - \$- : options du shell
  - \$? : code de retour de la dernière commande
  - \$\$ : PID du shell
  - \$! : PID du dernier processus lancé en arrière-plan
  - \$\_ : dernier argument de la commande précédente

# Exécution et environnement Shell

## exemple de script

- Exemple:

```
>#!/bin/bash  
># test2.sh  
>echo "Nom du script: $0"  
>echo "premier paramètre: $1"  
>echo "second paramètre: $2"  
>echo "PID du shell :" $$  
>echo "code de retour: $"?  
>exit
```

# PLAN

---

- Généralités
- Exécution et environnement Shell
- Manipulation des variables
- Tests: les instructions if et case
- Boucles: les instructions for, while, et until
- Les tableaux
- Les fonctions

# Manipulation des variables

## Variables internes

- Une variable est un symbole qui associent un nom (identifiant) à une valeur ou un objet qui sera implantée dans la mémoire du système. Il peut contenir une valeur qui peut varier au cours de l'exécution du programme.
- En Shell, contrairement à d'autres langages de programmation, les variables ne sont pas classées par type mais pour l'essentiel, suivant le contexte, nous avons des chaînes de caractères et des chiffres. Les opérations arithmétiques, les comparaisons et autres sont autorisées.
- Pour pouvoir effectuer une affectation, on utilise l'opérateur "=" et pour le déréférencement ou appel, on utilise "\$".
  - VARIABLE="valeur"
  - echo \$VARIABLE # ou bien echo \${VARIABLE}
  - **Exemple:**
    - #!/bin/bash
    - # test3.sh
    - PRENOM="pape"
    - echo "mon répertoire perso est: /home/\${PRENOM}"
    - exit

# Manipulation des variables

## typage des variables

- Les commandes **declare** ou **typeset** (ils sont synonymes) permettent de restreindre les propriétés des variables. Quelques options:

- **-r** (pour la lecture seule): **declare -r variable**

On ne peut que lire ce type de variable. Une tentative de modification de la valeur entraînera un message d'erreur.

- **-i** (entier): **declare -i variable**

Les occurrences suivantes de variable seront traitées comme un entier.

- **Exemples:**

```
>#!/bin/bash
># test4.sh
>declare -i n
>n=3
>echo "la valeur est: $n" # on lit: la valeur est 3
>n="trois"
>echo "la valeur est: $n" # on lit: la valeur est 0
>exit
```

```
>#!/bin/bash
># test5.sh
>n=8/2
>echo "n = $n"      # n = 8/2
>declare -i n
>n=8/2
>echo "n = $n"      # n = 4
>exit
```

# Manipulation des variables

## Interaction avec l'utilisateur

Les commandes suivantes permettent d'interagir avec l'utilisateur:

- La commande `echo` pose une question;
- La commande `read` lit les valeurs entrées au clavier et les stocke dans une variable.
  - `echo "prenom ?"`
  - `read val`
  - `echo $val`
- Pour aller plus vite, on peut se passer de la commande `echo` et utiliser uniquement la commande `read -p` (cela permettra de faire sortir du texte et d'attendre une valeur en entrée).
  - `read -p "prenom ?" val`
  - `echo $val`
- Si aucune variable n'est fournie lors de l'appel de `read`, la valeur entrée est stockée dans la variable `REPLY`.
  - `echo "prenom ?"`
  - `read # alphonse`
  - `echo $REPLY # alphonse`

# Manipulation des variables

## cas pratiques

### ➊ Incrémentation:

- `a=2334`
- `let "a += 1"`
- `echo "a = $a"`

### ➋ Remplacer "23" par "BB":

- `b=${a/23/BB}`
- `echo "b = $b"`
- `declare -i b`
- `echo "b = $b"`
- `let "b += 1"`
- `echo "b = $b"`

### ➌ Remplacer "BB" par "23"

- `c=BB34`
- `d=${c/BB/23}`
- `echo "d = $d"`
- `let "d += 1"`
- `echo "d = $d"`

### ➍ Variables nulles:

- `e=""`
- `echo "e = $e"`
- `let "e += 1"`
- `echo "e = $e"`

### ➎ Variables non déclarées:

- `echo "f = $f"`
- `let "f += 1"`
- `echo "f = $f"`

# Manipulation des variables

## cas pratiques

### Utilisation de echo et printf:

- `#!/bin/bash`
- `PI=3.14159265358979`
- `Msg="Hello world"`
  
- `echo`
- `printf "Pi avec deux décimales = %1.2f" $PI`
- `echo`
- `printf "Pi avec neuf décimales = %1.9f" $PI`
- `printf "\n"`
- `printf "%s \n" $Msg`
- `echo`
  
- `exit 0`

### Utilisation de read:

- `#!/bin/bash`
  
- `echo -n "Entrez une valeur : "`
- `read val`
- `echo "val = $val"`
- `echo`
- `echo -n "Entrez deux valeurs:"`
- `read vall val2`
- `echo "vall = $vall et val2 = $val2"`
- `echo`
- `echo -n "Saisissez une nouvelle valeur : "`
- `read`
- `val3="$REPLY"`
- `echo "valeur=$val3"`
  
- `exit 0`

# PLAN

---

- Généralités
- Exécution et environnement Shell
- Manipulation des variables
- Tests: les instructions if et case
- Boucles: les instructions for, while, et until
- Les tableaux
- Les fonctions

# Tests: les instructions if et case

## l'instruction if

Le Shell possède des structures de contrôle telles qu'il en existe dans les langages de programmation d'usage général. Dans cette section, nous allons voir les instructions conditionnelles:

- l'instruction if,
- la commande test qui la complète,
- l'instruction case.

### ○ L'instruction if:

Elle présente trois variantes qui correspondent aux structures sélectives à une, deux ou n alternatives.

- La sélection à une alternative : if... then... fi

```
if [ condition ]
then
    commande1
    commande2
    ...
fi
```

- Les commandes sont exécutées si la condition renvoie un code retour nul ( \$? = 0 )

# Tests: les instructions if et case

## l'instruction if

- Exemples:

```
>#!/bin/bash  
># test21.sh  
>read -p "Enter la valeur de A: " a  
>read -p "Enter la valeur de B: " b  
>if [ $a -lt $b ]  
    >then echo "$a est inférieur à $b"  
>fi
```

```
>#!/bin/bash  
># test22.sh  
>read -p "Enter un nombre: " x  
>if [ $((x%2)) == 0 ]  
    >then echo "$x est paire"  
>fi
```

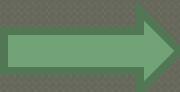
- Les crochets ( [ ] ) font référence à la commande test dans lequel ont pourra effectuer toutes les opérations tests. Ci-dessous, quelques opérateurs:

- ! Expression : La négation
- n la\_chaine : La taille de la\_chaine est plus grand que zéro
- z la\_chaine : La taille de la\_chaine est null
- chaîne1 = chaîne2 : Chaîne1 est identique à la chaîne2
- chaîne1 != chaîne2 : Chaîne1 est différent de la chaîne2
- entier1 -eq entier2 : entier1 est numériquement égal à l'entier2
- entier1 -gt entier2 : entier1 est numériquement plus grand que l'entier2
- entier1 -lt entier2 : entier1 est numériquement plus petit que l'entier2

# Tests: les instructions if et case

## l'instruction if

- **&& ou bien -a:** ET logique
  - **|| ou bien -o:** Ou logique
  - **-d fichier :** le fichier existe et est un répertoire;
  - **-e fichier :** le fichier existe;
  - **-r fichier :** le fichier existe et est autoriser pour lecture;
  - **-s fichier :** le fichier existe et il n'est pas vide;
  - **-w fichier :** le fichier existe et est autoriser pour écriture;
  - **-x fichier :** le fichier existe et est autoriser pour exécution;
- Exemples:



```
>#!/bin/bash
># test21a.sh
>read -p "Enter un nombre: " x
>if [[ $x -lt 10 || $x -ge 100 ]]
    >then echo "la valeur entree n'est pas a deux chiffres"
>fi
```



```
>#!/bin/bash
># test21b.sh
>read -p "Enter un nombre: " x
>if [[ $x -lt 10 o $x -ge 100 ]]
    >then echo "la valeur entree n'est pas a deux chiffres"
>fi
```

# Tests: les instructions if et case

## l'instruction if

- Attention:

Dans [ condition ], ne pas oublier le caractère espace entre [ et condition et entre condition et ]. Si then est sur la même ligne, il doit être séparé du ] par un espace et un caractère ; .

- La sélection à deux alternatives : if... then... else... fi

```
if [ condition ]
    then commandes1
    else commandes2
fi
```

- Les commandes commandes1 sont exécutées si la condition renvoie un code retour nul, sinon ce sont les commandes2 qui sont exécutées.
- Exemples:

```
>#!/bin/bash
># test23.sh
>read -p "Enter un nombre: " x
>if [ $((x%2)) == 0 ]
    >then echo "le nombre est paire"
>else
    >echo "le nombre n'est pas paire"
>fi
```

# Tests: les instructions if et case

## l'instruction if

- La sélection à n alternatives : if... then.... elif... then... fi

```
if [ condition1 ]
    then commandes1
elif [ condition2 ]
    then commandes_2
elif [ condition3 ]
    then commandes_3
...
else commandes_n
fi
```

- Les commandes commandes1 sont exécutées si la condition1 renvoie un code retour nul, sinon la condition2 est testée et les commandes2 sont exécutées si elle renvoie un code de retour nul, ainsi de suite jusqu'à la dernière qui exécute les commandes\_n si la condition\_n-1 ne renvoie pas un code de retour nul.

# Tests: les instructions if et case

## l'instruction if

- Exemples:

```
>#!/bin/bash  
># test24a.sh  
  
>echo -n "Enter un nombre: "  
>read x  
>if [ $x == 0 ] ;then  
    >echo "le nombre est zero"  
>elif [ $((x%2)) == 0 ];then  
    >echo "le nombre est paire"  
>else  
    >echo "le nombre est impaire"  
>fi
```

```
>#!/bin/bash  
># test24b.sh  
  
>read -p "Enter un nombre: " x  
>if [[ $x -lt 0 ]]  
    >then echo "le nombre est positif"  
>elif [[ $x-gt 0 && $x-lt 10 ]];then  
    >    echo "le nombre est un chiffre"  
>elif [[ $x-gt 9 && $x-lt 100 ]];then  
    >    echo "le nombre est a deux chiffres"  
>else  
    >echo "le nombre est a plusieurs chiffres"  
>fi
```

# Tests: les instructions if et case

## l'instruction case

L'instruction case est une instruction très puissante et très commode pour effectuer un choix multiple dans un fichier de commandes.

- La syntaxe est la suivante:

```
case chaine in
    motif1) commandes 1 ;;
    motif2) commandes 2 ;;
    ....
    motifn) commandes n ;;
esac
```

- Le Shell recherche, parmi les différentes chaînes de caractères **motif1**, **motif2**,..., **motifn** proposées, la première qui correspond à **chaine** et il exécute les commandes correspondantes.
- Un double point virgule (**;;**) termine chaque choix.
- La chaine dans un case peut prendre diverses formes :
  - un chiffre,
  - une lettre ou un mot,
  - des caractères spéciaux du Shell,
  - une combinaisons de ces éléments.

# Tests: les instructions if et case

## l'instruction case

- La chaîne peut être lue, passée en paramètre ou être le résultat d'une commande exécutée avec l'opérateur backquote `` ou \$( ). Dans les différentes chaînes motif à n, on peut utiliser les caractères spéciaux (\*, ?, ...). De plus, pour regrouper plusieurs motifs dans une même alternative, on utilise le caractère | .
- Exemples:

```
>#!/bin/bash
># test25a.sh

>echo -n "Entrer un caractere"
>echo "1 - Bleue"
>echo "2 - Rouge"
>echo "3 - Jaune"
>echo "4 - Verte"
>echo "5 - Orange"
>read couleur;
>case $couleur in
>1) echo "Bleue est votre premiere couleur.";;
>2) echo "Rouge est votre premiere couleur.";;
>3) echo "Jaune est votre premiere couleur.";;
>4) echo "Verte est votre premiere couleur.";;
>5) echo "Orange est votre premiere couleur.";;
>*) echo "cette couleur n'existe pas. Faites le
bon choix svp!";;
>esac
```

```
>#!/bin/bash
># test25b.sh

>read -p "Enter un mois: " mois
>case $mois in
>Fevrier) echo "il y a 28 ou 29 jours en $mois .";;
>Avril | Juin | Septembre | Novembre ) echo "il y a
30 jours en $mois .";;
>Janvier | Mars | Mai | Juillet | Aout | Octobre |
Decembre ) echo "il y a 31 jours en $mois .";;
>*) echo "mois inconnu! Entrer un nom correct
svp!" ;;
>esac
```

# Tests: les instructions if et case

## l'instruction case

- Exemples:

```
>#!/bin/bash
># test26.sh

>echo -n "Entrer un caractere : "
>read caractere
>case $caractere in
>[[:lower:]] echo "Tu as entre un caractere minuscule.";;
>[[:upper:]] echo "Tu as entre un caractere majuscule.";;
>[0-9]) echo "Tu as entre un nombre a un chiffre.";;
>?) echo "Tu as entre un caractere speciale.";;
>*) echo "Tu as entre plusieurs caractere.";;
>esac
```

- Il est possible de définir une plage de valeur pour délimiter les valeurs possibles.
- Le ? caractère couvre les caractères qui ne sont pas des **minuscules**, des **majuscules** ou des **chiffres**. Il remplace un seul caractère, par opposition à \* qui remplace tout le reste non couvert par les conditions ci-dessus.

# Tests: les instructions if et case

## Exercices

---

- Crée un script qui demande à l'utilisateur de saisir une note et qui affiche un message en fonction de cette note :
  - « très bien » si la note est entre 16 et 20 ;
  - « bien » lorsqu'elle est entre 14 et 16 ;
  - « assez bien » si la note est entre 12 et 14 ;
  - « moyen » si la note est entre 10 et 12 ;
  - « insuffisant » si la note est inférieur à 10.
  
- Crée un script Shell nommé "nombreJours" qui affichera le nombre de jours du mois courant. "nombreJours" affichera pour février 2007 le message "28 jours en février 2007".

# PLAN

---

- Généralités
- Exécution et environnement Shell
- Manipulation des variables
- Tests: les instructions if et case
- Boucles: les instructions for, while, et until
- Les tableaux
- Les fonctions

# Boucles: les instructions for, while, et until

## la boucle for

---

La présence des instructions itératives dans le Shell en fait un langage de programmation complet et puissant. Le Shell dispose de trois structures itératives : for, while et until.

➤ Itération bornée : La boucle for

Trois formes de syntaxe sont possibles :

□ Forme 1

```
for variable in chaine1 chaine2... chaine_n  
do  
commandes  
done
```

□ Forme 2

```
for variable  
do  
commandes  
done
```

□ Forme 3

```
for variable in *  
do  
commandes  
done
```

# Boucles: les instructions for, while, et until

## la boucle for

- Pour chacune des trois formes, les commandes placées entre *do* et *done* sont exécutées pour chaque valeur prise par la variable du Shell *variable*. Ce qui change c'est l'endroit où *variable* prend ses valeurs.
  - Pour la forme 1, les valeurs de *variable* sont les chaînes de *chaine1* à *chaine\_n*.
  - Pour la forme 2, *variable* prend ses valeurs dans la *liste des paramètres du script*.
  - Pour la forme 3, la *liste des fichiers du répertoire* constitue les valeurs prises par *variable*.
- Exemples:*

```
>#!/bin/bash  
># test31a.sh  
  
>for i in {"un","deux","trois"}; do  
>    echo $i;echo "\n"  
>done
```

```
>#!/bin/bash  
># test31b.sh  
  
>for i ; do  
>    echo $i;echo "\n"  
>done
```

```
>#!/bin/bash  
># test31c.sh  
  
>echo "le repertoires courant est: `pwd` "  
>for i in * ; do  
>    echo $i;echo "\n"  
>done
```

# Boucles: les instructions for, while, et until

## la boucle for

- *Exemples:*

```
>#!/bin/bash  
># test32a.sh  
  
>for i in {1..5}; do  
>if [[ "$i" == '2' ]]; then  
>    continue  
>fi  
>echo "Nombre: $i"  
>done
```

```
>#!/bin/bash  
># test32b.sh  
  
>echo "liste des repertoires sous `pwd`"  
>echo "....."  
>for i in *  
>do  
>if [ d $i ] ; then  
>    echo $i ":repertoire"  
>else  
>    echo $i ":dossier"  
>fi  
>done  
>echo "....."
```

# Boucles: les instructions for, while, et until

## la boucle while et until

### ➤ Itération non bornées : *while* et *until*

Syntaxes:

*while condition  
do commandes  
done*

*until condition  
do commandes  
done*

- Les commandes sont exécutées tant que (while) ou jusqu'à ce que (until) la condition retourne un code nul (la condition est vraie).
- En until, lorsque l'expression est évaluée à FALSE, le bloc de commandes est exécuté de manière itérative. Pour la première fois, lorsque l'expression est évaluée à VRAI, la boucle est interrompue.

# Boucles: les instructions for, while, et until la boucle while et until

## ➤ Exemple avec la boucle while:

Il faut demander à l'utilisateur d'entrer une chaîne **oui** ou la chaîne **non** avec un contrôle de saisie c'est-à-dire si il entre un autre mot différent des deux, le script lui demandera à nouveau de saisir le bon mot jusqu'à ce que le mot soit correct.

```
➤#!/bin/bash
➤# test33a.sh

➤echo "entrer un mot"
➤read mot
➤while [ "$mot" != "oui" -a "$mot" != "non" ]
➤do
➤  read mot
➤done

➤if [ "$mot" = "oui" ]
➤then
➤  echo "vous avez saisi OUI"
➤elif [ $mot = "non" ]
➤then
➤  echo " vous avez saisi NON"
➤fi
```

```
➤#!/bin/bash
➤# test33b.sh

➤while read -p "Entrer le mot(oui/non) : " mot
➤do
➤  case $mot in
➤    oui) echo ' vous avez saisi OUI' ;break ;;
➤    non) echo ' vous avez saisi NON' ;break ;;
➤  esac
➤done
```

# Boucles: les instructions for, while, et until

## la boucle while et until

### ➤ Exemple avec la boucle until:

On initialise compteur à 0 et on boucler tant que le compteur reste en dessous de 5.

```
➤#!/bin/bash  
➤# test34a.sh  
  
➤compteur=0  
  
➤until [ $compteur -gt 5 ]  
➤do  
➤ echo Compteur:$compteur  
➤ ((compteur++))  
➤done  
  
➤exit 0
```

On peut boucler tant que le nombre saisi n'est pas égal à 35.

```
➤#!/bin/bash  
➤# test34b.sh  
  
➤nombre=0  
  
➤until (( nombre == 35 ))  
➤do  
➤ echo -e "Saisir 35 :\c"  
➤ read nombre  
➤done  
  
➤exit 0
```

# Boucles: les instructions for, while, et until

## la boucle while et until

On peut utiliser les instructions **break** et **continue** pour modifier le flux de la boucle.

- **break** : cette instruction termine la boucle en cours et passe le contrôle du programme aux commandes suivantes.
- **continue** : cette instruction termine l'itération actuelle de la boucle, en sautant toutes les commandes restantes en dessous et en commençant l'itération suivante.

On utilise la boucle infinie.

- Si le nombre est égal à 5, continuez en ignorant le reste du corps de la boucle.
- Si le nombre est égal ou supérieur à 10 la boucle se termine.

```
>#!/bin/bash
># test34c.sh

>compteur=0
>until false
>do
> ((compteur++))
> if [[ $compteur -eq 5 ]]
> then
>   continue
> elif [[ $compteur -ge 10 ]]
> then
>   break
> fi
> echo "Compteur = $compteur"
>done
>exit 0
```

# PLAN

---

- Généralités
- Exécution et environnement Shell
- Manipulation des variables
- Tests: les instructions if et case
- Boucles: les instructions for, while, et until
- Les tableaux
- Les fonctions

# Les tableaux

## Définition

Les tableaux sont bien pratiques dans des scripts Shell avancés.

**Nous en distinguons deux types:** les tableaux indicés (dont les clefs sont des entiers) et les tableaux associatifs (dont les clefs sont des chaînes de caractères).

- ◎ Création de tableau
- Un tableau indicé peut être créé en initialisant ses éléments misent entre parenthèses:

```
monTableau=(val1 val2 .... valN)
```

```
monTableau=("alpho" "beta" "gamma")
```

- Ici, les indices sont automatiquement assignés et ils commencent par 0.

On peut initialiser un tableau avec des indices imposés. Par exemple :

```
monTableau=("alpho" "beta" [5]="gamma")
```

- Un tableau associatif est aussi créer des la même façon mais pour ce cas de figure, il va falloir préciser une clef pour chaque élément.

```
monTableau=(['un']="alpho" ['deux']="beta" ['trois']="gamma")
```

# Les tableaux

## Définition

---

- Un tableau peut être créé de façon explicite à l'aide du mot-clef **declare**, suivi de l'option **-a** pour un tableau indicé, et **-A** pour un tableau associatif :

```
declare -a monTableauIndic
```

```
declare -A monTableauAssoc
```

- Ils peuvent aussi être initialisé en même temps qu'il est déclaré :

```
declare -a monTableauIndic=("alpha" "beta" "gamma")
```

```
declare -A monTableauAssoc(['un']="alpha" ['deux']="beta" ['trois']="gamma")
```

- Ils peuvent aussi être déclarés en lecture seule soit avec le mot-clef **declare** avec l'option **-r** ou avec le mot-clef **readonly**.

```
declare -r -a monTableau=("alpha" "beta" "gamma")
```

```
readonly -a monTableau=("alpha" "beta" "gamma")
```

# Les tableaux

## Affichage

---

### ○ Affichage

- Pour afficher les valeurs d'un tableau, la syntaxe est :

`echo ${monTableau[INDICE]}`. Exemple:

```
declare -a monTableau=("alpha" "beta" "gamma")
```

```
echo ${monTableau[0]} # vaut alpha
```

- Si l'indice est remplacé par @ ou \*, tous les éléments sont affichés. Par exemple :

```
monTableau=("alpha" "beta" "gamma")
```

```
echo ${monTableau[@]} # vaut alpha beta gamma
```

# Les tableaux

## Affichage

- Il est possible d'obtenir la liste des clefs d'un tableau à l'aide de la syntaxe suivante:

`${!monTableau[@]}`. Exemple:

```
declare -a monTableau1=("alpha" "beta" "gamma")
declare -A monTableau2=(['un']="alpho" ['deux']="beta"
['trois']="gamma")
echo ${!monTableau1[@]} # 0 1 2
echo ${!monTableau2[@]} # un deux trois
```

- il est possible de connaître la taille ou le nombre d'élément du tableau avec la syntaxe suivante:

`${#monTableau[@]}`. exemple:

```
declare -a monTableau=("alpha" "beta" "gamma")
echo ${#monTableau[@]} # 3
```

# Les tableaux

## Modification

---

### ○ Modification

- Un élément peut être assigné selon la syntaxe suivante:

monTableau[indice]=valeur. Exemple :

```
declare -a monTableau=("alpha" "beta" "gamma")
```

# ajouter un élément

```
monTableau[3] = delta
```

```
echo ${monTableau[3]} # vaut delta
```

- Si le tableau n'existe pas, il sera créé comme un tableau indicé.

# Les tableaux

## Modification

- On peut supprimer un élément du tableau avec la commande suivante:

`unset monTableau[indice]. Exemple :`

- Pour un tableau indicé:

```
declare -a monTableau=("alpha" "beta" "gamma", "delta")
echo ${monTableau[@]} # alpha beta gamma delta
unset monTableau[1]
echo ${monTableau[@]} # alpha gamma delta
```

- Pour un tableau associatif:

```
declare -A monTableau=(['un']="alpha" ['deux']="beta" ['trois']="gamma" ['quatre']="delta")
echo ${monTableau[@]} # alpha beta gamma delta
unset monTableau['deux']
echo ${monTableau[@]} # alpha gamma delta
```

- On peut également supprimer entièrement un tableau à l'aide de la commande `unset`. Pour cela, il existe 3 façons d'y parvenir:

- `unset tableau`
- `unset tableau[@]`
- `unset tableau[*]`

# Les tableaux

## cas pratique 1

```
> #! /bin/bash
> # test41.sh

> tab1[0]=un
> tab1[1]=1
> echo ${tab1[0]}
> echo ${tab1[1]}

> tab2=( un deux trois )
> echo ${tab2[0]}
> echo ${tab2[2]}

> tab3=( [6]=six [12]=12 )
> echo ${tab3[6]}
> echo ${tab3[12]}

> echo "Maintenant, il faut donner les elements du tableau 4"
> read -a tab4
> for i in "${tab4[@]}"
> do
>     echo $i
> done

> exit 0
```

# Les tableaux

## cas pratique 2

```
> #! /bin/bash
> # test42.sh

> tab=( pomme pigeon chat chien elephant grenouille )

> #affichage du premier element
> echo ${tab[0]}
> echo ${tab:0}

> #affichage de tous les elements
> echo ${tab[@]}
> echo ${tab[@]:0}

> #affichage de tous les elements à exception du premier
> echo ${tab[@]:1}

> #affichage des elements dans un intervalle
> echo ${tab[@]:1:4}

> #la taille du premier element
> echo ${#tab[0]}
> echo ${#tab}

> #le nombre d'elements du tableau
> echo ${#tab[*]}
> echo ${#tab[@]}

> #remplacement d'un caractere
> echo ${tab[@]//a/A}

> exit 0
```

# Les tableaux

## exercices

---

- Ecrire un programme qui lit un entier  $n$ , puis  $n$  autres entiers positifs dans un tableau, l'affiche puis calcul la somme, le max, et le min de ses éléments.
- Ecrire un programme qui construit la table de multiplication des entiers entre 1 et 9 puis l'affiche.
- On dispose de deux tableaux A et B (de dimensions respectives  $N$  et  $M$ ), triés par ordre croissant. Fusionner les éléments de A et B dans un troisième tableau FUS trié par ordre croissant.

# PLAN

---

- Généralités
- Exécution et environnement Shell
- Manipulation des variables
- Tests: les instructions if et case
- Boucles: les instructions for, while, et until
- Les tableaux
- Les fonctions
- Les fichiers
- Regex

# Les fonctions

## Définition

---

- Comme tout langage de programmation, l'utilisation des fonctions facilite le développement et la structuration de scripts.
- Il existe deux (2) méthodes différentes de déclarer une fonction.
  - Soit avec l'utilisation du mot clé **function**:

```
function maFonction()  
{  
    instructions  
}
```

- Soit sans le mot clé *function* (compatible avec les shells Bourne et Korn):

```
maFonction()  
{  
    instructions  
}
```

# Les fonctions

## Appel

- Pour appeler une fonction, on utilise la syntaxe suivante:

maFonction # sans paramètre

maFonction param\_1 param\_2 ... param\_n # avec paramètres

Exemples:

#sans paramètre

```
function hello {  
    echo "Hello World!"  
}  
hello
```

#avec paramètres

```
function carre {  
    echo "Le carré de $1 est $((\$1*\$1))"  
}  
carre 4
```

# Les fonctions

## récupération du résultat

- Pour pouvoir récupérer le résultat de l'exécution d'une fonction, on utilise la sortie standard de celle-ci en lui affectant à une variable.

Exemple:

```
function carre {  
    echo $($1*$1)  
}  
valeur=$(carre 4)  
echo "Le carre de 4 est $valeur"
```

```
function hello {  
    echo "bonjour"  
}  
x=$(hello)  
echo $x
```

- Une fonction en Bash, utilise le mot clé **return** pour passer les valeurs des variables locales au programme principal. la valeur renvoyée est stockée dans la variable par défaut **\$?** . Exemple:

```
function somme {  
    som=$[ $1 + $2 ]  
    return $som  
}  
somme 7 9  
echo "la somme est $"?
```

# Les fonctions

## Portée des variables

- Par défaut dans un script shell, les variables sont déclarées comme étant **globales**. Pour déclarer une variable localement, il faut la faire précéder du mot clé **local** et sera utilisable que dans la fonction qui la déclare durant l'exécution de celle-ci.

```
> #! /bin/bash
> # test511.sh

> function maFonction()
> {
>     local varlocal="variable locale de la fonction"
>     echo "$varlocal"
>     echo "Nombres de paramètres : $#"
>     echo $@
>     echo $*
>     echo $1
>     echo $2
> }

> maFonction "Hello" "World!"
> exit 0
```

# Les fonctions

## cas pratiques

```
> #! /bin/bash
> # test52.sh

> var1="valeur 1"

> function func {
>   echo "dans la fonction, je vaux var1=$var1";
>   var1="valeur 2";
>   echo "dans la fonction, apres modification, je vaux var1=$var1";
> }

> echo "Avant l'appel de la fonction, je vaux var1=$var1"

> func

> echo "Apres appel de la fonction, je vaux var1=$var1"

> exit 0
```

# Les fonctions

## cas pratiques

```
> #! /bin/bash
> # test531.sh

> add(){
>     som=$(( $1 + $2 ))
>     echo $som
> }

> read -p "Entrez un nombre:" n1
> read -p "Entrez un autre nombre :" n2

> valeur=$(add $n1 $n2)
> echo "Le résultat est :" $valeur

> exit 0
```

```
> #! /bin/bash
> # test532.sh

> function plusgrand {
>     if [[ $1 -gt $2 ]]
>     then
>         return 1
>     else
>         return 0
>     fi
> }

> read -p "Entrez un nombre:" n1
> read -p "Entrez un autre nombre :" n2

> plusgrand $n1 $n2
> echo "Le résultat est :" $?

> exit 0
```

# Les fonctions

## cas pratiques

```
> #! /bin/bash
> # test54.sh

> x=1
> y="hello"
> function modif {
>     x=2
>     local y="bonjour"
> }

> modif
> echo $x
> echo $y

> exit 0
```

# Les fonctions

## exercices

---

- ➊ Créer un script qui prend en paramètre deux valeurs décimale et qui affiche leur somme, produit et différence. Penser à utiliser les fonction.



**feedback**

pape.abdouaye.barro@gmail.com