Introduction aux scripts shell

➢ Un script shell est un programme qui permet d'automatiser l'exécution d'une série d'opérations.

➢ Les scripts shell se présente sous la forme d'un fichier contenant une ou plusieurs instructions écrites en langage interprété (**bash**, **csh**, **ksh**, **php**, **perl**, …) qui sont exécutées Séquentiellement.

➢ Un langage interprété est un langage de programmation dont les codes sources sont interprétés, autrement dit, chaque ligne du code source est analysée et exécutée par l'interpréteur.

➢ Les langages interprétés sont portables, autrement dit, ils sont indépendants du système d'exploitation utilisé.

➢ **Autrement dit, le shell est :**

- ✓ l'interpréteur de commande ;
- ✓ l'interface entre l'utilisateur et les commandes ;
- ✓ un langage de programmation (interpréteur), il permet donc de réaliser de nouvelles commandes ;
- ✓ un gestionnaire de processus ;
- ✓ un environnement de travail configurable.

➢ **Le fichier qui contient un script, doit commencer par l'instruction :**

#! /(chemin absolu de l'interpréteur utilisé)

➢ Exemples :

- ✓ #!/bin/sh Toutes les instructions qui suivent, seront analysées par l'interpréteur sh
- ✓ #!/bin/bash Toutes les instructions qui suivent, seront analysées par l'interpréteur bash
- ✓ #!/bin/csh Toutes les instructions qui suivent, seront analysées par l'interpréteur csh
- ✓ #!/bin/tcsh Toutes les instructions qui suivent, seront analysées par l'interpréteur tcsh
- ✓ #!/usr/bin/perl Toutes les instructions qui suivent, seront analysées par l'interpréteur perl
- ✓ ……

➢ **Exécuter un script :**

✓ Un script doit être écrit dans un fichier texte, qui peut être créer soit en utilisant des éditeurs de textes graphiques (**emacs**, **gedit**, **textedit**, …) ou bien dans un terminal, (**vi**, …).

✓ Généralement, les fichiers sous **UNIX** sont crées avec un **mask=22** (par défaut), autrement dit ces fichiers ne sont pas exécutables, dans ce cas vous pouvez exécuter votre script, en tapant :

le_nom_de_l'interpréteur Nom_de_script

Si non, il faudra utiliser la fonction **chmod** pour ajouter le droit d'exécution :

chmod +x Nom_de_script

Dans ce cas, vous pouvez exécuter votre script en tapant :

./Nom_de_script ou bien /chemin_absolu/*Nom_de_script*

Vous pouvez aussi modifier la valeur de la variable **PATH** (**export PATH=$PATH:$HOME/Scripts**) afin de faciliter l'exécution.

➢ **Les variables d'un script Shell**

$? : Renvoie **1** si le nom du fichier du scripts Shell est connu, sinon **0**;

$0 : Renvoie le nom du fichier du scripts Shell (erreur si inconnu);

$*** : Renvoie la liste **$argv[*] des arguments;

$n ou **${n}** : Renvoie le **n**ème argument **$argv[n]** du scripts Shell;

$?var : Renvoie **1** si la variable var a une valeur, sinon **0** ;

$$: Renvoie le numéro de processus du Shell père ;

$# : Renvoie le nombre d'argumentsLes paramètres de réglage

➢ Les paramètres de réglage traduisent la position des composantes d'une saisie à l'intérieur d'une ligne de commande. Ils se composent du caractère signalant une variable **$** et un numéro d'ordre dans la saisie. Il peut y avoir au maximum 10 paramètres de réglage en commençant par **$0**

Prog.sh	**fichier1**	**fchier2**	**fichier3**
$0	**$1**	**$2**	**$3**

✓ Pour pouvoir traiter des commandes qui comportent plus de 10 paramètres de réglage. Les positions supérieures à 10 peuvent être demandées si la ligne de commandes est déplacée vers la gauche à l'aide de shift.

Sortie :

AA BB CC DD EE FF GG HH II KK LL MN NN
$0 $1 $2 $3 $4 $5 $6 $7 $8 $9

shift

Sortie :

BB CC DD EE FF GG HH II KK LL MM NN
$0 $1 $2 $3 $4 $5 $6 $7 $8 $9

Le langage interprété de Bourne Shell
SH (BASH, …)

Ce langage a été programmé pour la première fois en 1977 par Stephan Bourne (laboratoire d'AT&T Bell), il est devenue le shell par défaut pour les systèmes UNIX.

Les objets du langage:

I. Les variables

a. Déclaration et type de variables :
   i. Il n'est pas nécessaire de déclarer une variable avant de l'utiliser.
   ii. Les objets possédés par les variables sont d'un seul « type » : chaîne de caractères.

II. Affectation d'une valeur à une variable

a. Syntaxe : ***nom-de-variable = chaîne-de-caractères*** , (évitez les espaces).
   i. Création et modification en BASH :
      1.
         a. var1=5
         b. **var2=7**
      2. On peut déclarer des variables vides :
         a. var3=
         b. **var3=""**
      3. Affectation et interprétation des blancs : si la *chaîne-de-caractères* à affecter à la variable comporte des blancs, il faut en faire un seul mot à l'aide des quotes.

         var='En Normandie, il fait toujours beau !'

      4. Créer une variable en lecture seule :

         a. readonly variable=valeur

         b. Exemple : readonly MaVariable=7

      5. créer une variable locale (dans une fonction) :

         a. local variable=valeur

III. Variables dynamiques

a. Il est possible d'utiliser le contenu d'une variable pour en appeler une autre en remplaçant le caractère $ par un !
   1. Var="Test" ; Toto=Var
   echo $Toto ==> Var # Affiche le contenu de la variable **Toto**

      1. **echo ${Toto}** ==> bash: **echo${$Toto}:** bad substitution # Erreur

      2. **echo ${!Toto}** ==> Test # !Toto retourne le contenu de la variable **Var ==> echo $Var**
      (ou bien **${Var}**)

IV. Afficher la longueur d'une variable ou le nombre d'éléments d'un tableau avec

i. Soit la variable : Var=abc.gm3@insa.GM

➡ La longueur de cette variable est : **echo ${#Var}** ==> 15 # Il y a 15 caractères

V. Retourner une valeur par défaut :

a. Si la variable n'existe pas "-" ou si la variable est vide ":-"
   i. ${Var-VarParDefaut}
      1. Exemple1 : VarParDefaut="Dimanche"
         a. echo ${Var-$VarParDefaut} ==> Dimanche
         b. echo ${Var:-$VarParDefaut} ==> Dimanche
      2. Exemple2: VarParDefaut="Dimanche" ; Var=""
         a. echo ${Var-$VarParDefaut} ==>
         b. echo ${Var:-$VarParDefaut} ==> Dimanche

VI. Opérations sur les contenus des variables # Ces opérations ne sont pas valides sur certaines version de bash

a. **Conversion minuscules <==> MAJUSCULES**
   i. **minuscules ==> MAJUSCULES**
      1. **Première lettre en MAJUSCULE ${Var ^}**
         a. **Chaine="un test"; echo ${Chaine^} ==> Un test**
      2. **La chaîne en MAJUSCULE ${Var^^}**
         a. **Chaine="un test"; echo ${Chaine^^} ==> UN TEST**
   ii. **MAJUSCULES ==> minuscules**
      1. **Première lettre en minuscule ${Var,}**
         a. **Chaine="UN TEST"; echo ${Chaine,} ==> uN TEST**
      2. **La chaîne en minuscule ${Var,,}**
         a. **Chaine="UN TEST"; echo ${Chaine,,} ==> un test**

---

b. **Substitutions :**
   i. **Remplacer la première occurence avec / ==> ${Var/Cherche_Motif/Remplace_Motif}**
      1. **Var="Mon premier script"**
         a. **echo ${Var/M/S} ==> Son premier script**
         b. **echo ${Var/Mon/Notre} ==> Notre premier script**
         c. **echo ${Var/premier/\?} ==> Mon ? script**
         d. **echo ${Var/p*m/?} ==> Mon ?ier script**
         e. **...**

---

   ii. **Remplacer toutes les occurences avec // ==> ${Var/Cherche_Motif/Remplace_Motif}**
      1. **Var="Mon premier script"**
         a. **echo ${Var//r/?} ==> Mon p?emie? sc?ipt**
         b. **echo ${Var//p/#} ==> Mon #remier scri#t**
      2. **Var="toto:x:309:50080:Monsieur Toto:/bin/bash**
         a. **echo ${Var//:/ } ==> toto x 309 50080 Monsieur Toto /bin/bash**

---

   iii. **Remplacer un motif du début ou fin de chaine "/#" ou "/%"**

      1. **Var="Mon premier script"**
         a. **echo ${Var/#Mon/Notre} ==> Notre premier script**
         b. **echo ${Var/#on/Notre} ==> # il n'y a pas de substitution, "on" n'est pas au début**
      2. **Var="Mon premier scriptSHELL"**
         a. **echo ${Var/%SHELL/ SHELL} ==> Notre premier script SHELL**
         b. **echo ${Var/%t/t SHELL} <==> echo ${Var/%SHELL/ SHELL}**

c. ...

c. **Conversion " binaire, octal, hexadécimale" vers la base décimale : B#Valeur**

i. **Conversion de binaire vers décimal**

1. **echo $((2#1)) ==> 1**
2. **echo $((2#10)) ==> 2**
3. **...**
4. **echo $((2#1010)) ==> 10**
5. **...**

ii. **Conversion de octal vers décimal**

1. **echo $((8#1)) ==> 1**
2. **...**
3. **echo $((8#7)) ==> 7**
4. **echo $((8#10)) ==> 8**
5. **...**
6. **echo $((8#12)) ==> 10**
7. **...**

iii. ....

Variables systèmes (extrait)	
$!	PID de la dernière commande lancée en arrière-plan
$$	PID du shell en cours (vérifiable avec la commande ps)
$_	Dernier paramètre de la dernière commande exécutée après substitution
!$	Dernière commande avant toute substitution
$BASH	Chemin d'accès de l'interpréteur bash
$BASH_VERSION	Version du bash (voir aussi le tableau $BASH_VERINFO)
$COLUMNS	Nombre de colonnes affichable par le terminal
$HOME	Dossier home de l'utilisateur (cd $HOME = cd ~)
$HOSTNAME	Nom de la machine
$HOSTTYPE	Informations sur la machine (ex x86_64) Voir aussi $MACHTYPE
$LINES	Nombre de lignes affichable par le terminal
$OLDPWD	Dossier précédent (cd $OLDPWD = cd ~-)
$PATH	Chemins de recherche des exécutables
$PIPESTATUS	Tableau contenant les codes de retour des commandes d'un pipe
$PPID	PID du processus père
$PWD	Dossier en cours (cd $PWD = cd ~+)
$RANDOM	Chiffre aléatoire entre 1 et 32237 (en bash seulement)
$SECONDS	Nombre de secondes écoulées depuis le lancement de l'interpréteur, donc du script
$SHELL	Nom du shell de login
$TMOUT	Nombre de secondes avant la déconnexion du script
$UID	UID de l'utilisateur en cours ($EUID donne l'identifiant réel)
$USER	Nom de l'utilisateur

VII. Les Tableaux

a) La création d'un tableau

- **En BASH, les tableaux commencent à l'index 0 !** (et à 1 en CSH)

✓ **Pour indiquer au bash la création ou la modification, il suffit d'utiliser des parenthèses !**

Exemple de création :

Tableau=(Un DeuxTrois Quatre)

✓ L'affichage se fait avec **${LeNomDuTableau[Index]}**

✓ Remplacer l'index par le caractère * ou @ pour afficher l'intégralité du tableau

☑ **Exemples d'utilisation :**

Montableau=(Janvier février Mars Avril Mai Juin Juillet Aout Septembre Octobre Novembre Décembre)

- Afficher le premier élément du tableau : **echo** $Montableau ==> Janvier
- Afficher le deuxième élément du tableau : **echo** ${Montableau[1]} ==> février
- Afficher l'intégralité du tableau : **echo** ${Montableau[@]} ==> Janvier février ... Décembre
- Ou bien **echo** ${Montableau[*]} ==> Janvier février ... Décembre
- Afficher la longueur du 1er élément du tableau : **echo** ${#Montableau} ==> 7
- Afficher la longueur du i ème élément du tableau : **echo** ${#Montableau[i]}
- Afficher le nombre d'éléments du tableau : **echo** ${#Montableau[*]} ==> 12
- Ajouter un nouveau élément au tableau : Montableau+=("2021")
- Afficher les indices utilisées du tableau : **echo** ${!Montableau[*]}

☑ **Suppression d'un élément du tableau** (pour supprimer tout le tableau :

- **unset Montableau** **# Effacer le contenu du tableau**
- **unset Montableau[0]** **# Effacer le premier élément du tableau**
- **unset Montableau[i]** **# Effacer le l'élément à l'indexe i+1 du tableau**

☑ **Ajout d'un élément en fin de tableau**

- **Tableau+=(Cinq); echo ${Tableau[4]} ==> Cinq # Affiche l'élément à l'index 4 du tableau (MonTableau contient bleu noir cyan), on utilise ici un calcul avec $((…))**
- **echo ${Tableau[*]:$((${#tableau[*]}-1))}** ==> # renvoie le dernier élément du tableau

☑ **Ajoute un élément a un indice spécifique (pour laisser des valeurs nulles par exemple)**

- **Tableau=(Un Deux Trois Quatre [9]=Tata)** Un, Deux, Trois, Quatre, puis à l'index 9 on trouve Tata
- Affiche les index utilisés du tableau remplis précédemment

echo ${!Tableau[@]} ==> 0 1 2 3 9

VIII. Affectation et substitutions

NomDuTableau=($(commande)) ou bien **NomDuTableau=`commande`**

- **Ou bien : Tab=($(cat NomDuFichier))** <==> **Tab=`cat NomDuFichier`**

a. On peut affecter à une variable le résultat d'exécution d'une commande :

i. **D=$(date)** ou bien **D=`date`**

1. **echo $D ==> Dim 31 jan 2021 17:45:41 CET**
2. **echo ${D[*]} ==> Dim 31 jan 2021 17:45:41 CET**

ii. **Soit le fichier SeR qui contient le texte "J'aime programmer des scripts SHELL"**

1. **Tab=($(cat SeR) ou bien Tab=`cat SeR`**
2. **echo ${Tab[*]} ==> J'aime programmer des scripts SHELL**

b. On peut aussi affecter à une variable le contenu d'une autre variable (substitution), sauf * :

i. U=$USER

ii. **F=*** On affecte à la variable **F** le caractère ***** et non pas la liste des noms des fichiers.

IX. Autres possibilités d'utilisation de l'opérateur $:

a. **Avec une variable :**

i. Var=azerty.qwerty@coucou.com

1. **echo ${#var} ==> 24** # Il y a 24 caractères

b. **Avec un tableau :**

i. **Tableau=(azerty . qwerty @ coucou . com)**

1. **echo ${#Tableau[*]} ==> 7** # Il y a 7 éléments

X. Découpage de chaines de caractères avant ou après un motif *m* avec #, ##, % et %%

a. **Elimine ce qui se trouve AVANT le premier motif**

i. Var=azerty.qwerty@coucou.com

1. **echo ${Var#*.}** ==> qwerty@coucou.com
2. **echo ${Var#*c}** ==> **oucou.com**

b. **Elimine ce qui se trouve APRES le dernier motif**

i. **Var=azerty.qwerty@coucou.com**

1. **echo ${Var%.*} ==> azerty.qwerty@coucou**
2. **echo ${Var#@*} ==> alerte.qwerty**

c. **Elimine ce qui se trouve AVANT le dernier motif**

i. **Var=azerty.qwerty@coucou**

1. **echo ${Var##*.} ==> qwerty@coucou**

2. **echo ${Var##*@} ==> coucou**

XI. Elimine ce qui se trouve APRES le premier motif m

i. **Var=azerty.qwerty@coucou**

1. **echo ${Var%.*} ==> azerty**

2. **echo ${Var%%@*} ==> azerty.qwerty**

XII. Récupérer une partie d'une chaine

i. **Var=azerty.qwerty@coucou**

1. **echo ${Var:0:5} ===> azert**

2. **echo ${Var:6:10} ==> qwerty@co**

3. **echo ${Var:(-4):3} ==> uco**

XIII. Découpage d'un tableau

i. **Tab**=(1 2 3 4 5 6)
   1. **echo ${Tab[*]:0:3}** ==> 1 2 3

XIV. Lire la saisie d'un utilisateur au clavier

- La fonction **read** permet de permet d'arrêter une procédure pour lire et y faire entrer une ou plusieurs saisies de l'entrée standard (normalement le clavier).

Syntaxe : read <option> <var1> <var2> ...

Options :

-n **NbCar** ==> Stoppe la saisie après NbCar caractères

Exemple : **read -n** 5 var

-p **Texte** = Affiche le texte avant la saisie

Exemple : **read -p** '*entrer 4 valeurs séparées par un espace* ' **-a** Tab

-t **NbSec** = Attends NbSec secondes avant d'annuler la saisie

Exemple : **read -t** 20 var

.....

XV. LES DIRECTIVES INTERNES DU SHELL

Syntaxe : test-[rwxfdcbugkstzn] <argument> commande interactive

a. Commande: test

i.	**-r <nom>**	**le fichier existe avec autorisation de lire**
ii.	**-w <nom>**	**le fichier existe avec autorisation d'écrire**
iii.	**-x <nom>**	**le fichier existe avec autorisation d'exécuter**
iv.	**-f <nom>**	**fichier existant**
v.	**-d <nom>**	**répertoire de fichiers existant**
vi.	**-s <nom>**	**le fichier existe et n'est pas vide**
vii.	**-b <nom>**	**le fichier est orienté blocs et existe**
viii.	**-c <nom>**	**le fichier est orienté caractères et existe**
ix.	**-z <nom>**	**la longueur de la suite de caractères est 0 (vide)**

b.	**-n** <nom>	longueur de la suite de caractères > 0
c.	**<s1> = <s2>**	les chaînes s1 et s2 sont identiques (espaces !)
d.	**<s1> != <s2>**	s1 est différent de s2 (espaces !)
e.	**<s1> = <s2>**	les chaînes s1 et s2 sont identiques (espaces !)
f.	**<s1> != <s2>**	s1 est différent de s2 (espaces !)
g.	**!<Condition>**	négation de la condition
h.	**<cond1> -a <cond2>**	ET logique
i.	**<cond1> -o <cond2>**	OU logique (inclusif)
j.	**\(cond1 cond2 cond3\)**	conditions à réunir dans un groupe

k. Comparaison algébrique val1 -opérateur val2

i.	**<val1> -eq <val2>**	**val1 = val2**
ii.	**<val1> -ne <val2>**	**val1 <> val2**
iii.	**<val1> -gt <val2>**	**val1 > val2**
iv.	**<val1> -ge <val2>**	**val1 >= val2**
v.	**<val1> -lt <val2>**	**val1 < val2**
vi.	**<val1> -le <val2>**	**val1 <= val2**

l. Test de variables numériques (exemple bash)

i. AA=5; AA=`expr $AA + 1`

```
echo $? ==> 0
echo $AA ==> 6
```

La variable AA passe de 5 à 6 sans soucis Le code de retour est bien de zéro.

ii. AA="M"; AA=`expr $AA + 1`

```
==> expr: not a decimal number: 'M'.
# Provoque une erreur de expr !
echo $? ==> 2
```

XVI. Les structures Conditionnelles

a. Les tests simples

```
if <condition> ; then
Action(s)
fi
```

b. Les tests avancés

```
if <condition1> ; then
Action1(s)
elif <Condition2> ; then
Action2(s)
……
else
Actionn(s)
fi
```

Exemples :

Ex1.sh :

```
#!/bin/sh
if test $PWD = $HOME ; then
echo "Vous êtes dans votre répertoire d'accueil."
elif test $PWD = $HOME/SE ; then
        echo "Vous êtes dans le répertoire SE."
elif test $PWD = $HOME/Documents ; then
        echo "Vous êtes dans le répertoire Documents"
elif test $PWD = $HOME/Images ; then
        echo "Vous êtes dans le répertoire Images."
else
        echo "Vous êtes dans un autre répertoire"
fi
```

Ex2.sh : tester l'existence des paramètres

```
#!/bin/sh
if [ ! $1 ] ; then
echo "aucun paramètre"
elif [ ! $2 ] ; then
echo "Vous avez un seul paramètre"
else
echo "Vous avez $# paramètres"
fi
```

VII. La structure conditionnelle case

La construction **case** permet de vérifier si une expression *(variable $VAR, $i ou une constante)* figure dans la liste de modèles.

```
case <expression> in
modele1)                    action;;
modele2 | modele3)          action;;
modele4)                    action
esac
```

Exemple1 :

```
case $1 in
    lundi)                  echo " ..... ";;
    mardi)                  echo " ..... ";;
    mercredi)               echo " ..... ";;
    jeudi)                  echo " ..... ";;
    vendredi)               echo " ..... ";;
    samedi | dimanche)      echo " ..... ";;
    *)                      echo " ..... ";;
esac
```

- Exemple2 :

```
#!/bin/bash
echo entrez une chaine de caractères
read var
case $var in
        [0-9]*)
                if [[ $var = +([0-9]) ]] ; then
                echo "$var est une chaîne numérique"
                else
                echo "$var est une chaîne de caractères commençant par un nombre."
                fi;;
    [a-zA-Z]*)
                if [[ $var = +([a-zA-Z]) ]] ; then
                echo "$var est mot"
                else
                echo "$var est une chaîne de caractères commençant par un mot."
                fi;;
            *)  echo "$var n'est ni un nombre ni un mot.";;
esac
```

cas où les paramètres sont fournit par l'utilisateur.
Exemple 3 :

```
#!/bin/bash
case $1 in
  *[!0-9]*) echo "$1 n'est pas un nombre";;
          *) echo "$1 est  un nombre"
esac
```

Exemple 4 :

case sur un lecture clavier ex_case.sh:

```
#!/bin/sh
        echo "Voulez vous continuer le programme ?"
        read rep
        case $rep in
              [yYoO]) echo "On continue"
                      ./ex_case.sh;;
               [nN]) echo "$0 s'arrête"
               exit 0;;
                    *) echo "ERREUR de saisie"
               exit 1;;
         esac
```

XVII. Les paramètres d'un script ou une fonction :

a. Gestion des paramètres passés à un script

i. exemple, dans la commande : **ls -l Scripts**, nous considérons que "**-l**" et "**Scripts**" sont deux paramètres passés à la commande **ls.**

Variables	En tableau	Signification
$#	**$#argv**	Nombre de paramètres
$0	**$argv[0]**	Nom du script
$n	**$argv[n]**	Nième paramètre
$*	**$argv[*]**	La totalité des paramètres

Rappel : la commande **shift** décale les paramètres (**$n+1** devient **$n**), $0 est épargné

XVIII. Création de paramètres :

La commande **set** affiche (entre autre) la liste des variables et leurs contenus.

i. En bash cette fonction peut être utilisée pour remplir les paramètres : $1 à $n

1. set un deux trois quatre ; echo $3 ==> trois

2. Var="a b c d e"; set $Var; echo $2 ==> b

3. Afficher le dernier paramètre :

Si la commande date affiche : Ven 14 fév 2020 12:04:42 CET

La commande: set $(date); echo ${@:$#} ==> CET

Opérations :	**Incrémenter une valeur : méthodes diverses (bash)**
Commande externe EXPR	`a=$(expr $a + 1))`
Evaluation arithmétique	`((a=a+1)) <==> let a=a+1 (Instruction KSH)`
Evaluation arithmétique	`((a ++)). <==> let a=a+1 (Instruction KSH)`
Commande interne LET	`let "a=a+1"  (Instruction KSH)`

Exemples :
`A=5`
`echo $A ==> 5`
`((A++))`
`echo $A ==> 6`
`echo $((A++)) ==> 7`
`echo $((++A)) ==> 8`
	
	La variable contient bien **5**
	Incrémentation
	La variable contient **6**
	Incrémentation PUIS affichage, A contient **7**
	Incrémentation PUIS affichage, A contient **8**

XIX. La structure itérative for

Une boucle for avec une fin permet d'exécuter la même action pour une liste d'objets.

for *variable* in liste de valeurs

do

actions

done

Exemple 1 : chercheclient.bash

```
# recherche des clients dans des fichiers d'adresses
for client in agenda_1 agenda_2 agenda_3
        do
                grep $1 $client
        done
```

Exemple 2 :

Opérer sur le contenu d'un répertoire :

```
#!/bin/bash
for i in *
do
   echo $i
done
```

XX. Compteur de boucles seq

```
#!/bin/bash
for i in `seq 1 n`
do
        echo Instruction $i
done
```

XXI. Autres syntaxes :

```
for ((e1;e2;e3))
        do instruction(s)
done
```

- e1, e2 et e3 sont des expressions arithmétiques.

Exemple :

```
#!/bin/bash
for ((i=0 ; 10 - $i ; i++))
do
        echo $i
done
```

XXII. Les boucles while et until

La boucle **while** permet d'exécuter un bloc d'instructions tant qu'une certaine condition est vraie, cette boucle s'arrêt lorsque cette condition devient fausse.

Si la condition est toujours vraie, dans ce cas, cette boucle peut effectuer un nombre indéterminé ou infini de tours de boucles.

While/until condition
    do instruction(s)
done

Exemple:
"Tant que la chaine entrée en paramètre est différente d'une chaine vide faire, insérer la nouvelle chaine à la fin du fichier "fich_dyn"

```
while test "$1" != ""
    do
        cat $1 >> fich_dyn
    done
```

XXIII. Les directives: break (exit), continue

Les directives agissent sur les boucles for et while.

break permet de terminer prématurément une boucle (exit);

continue permet de parcourir encore une fois une boucle.

exit permet de terminer une procédure du shell

Exemple: vérification du type de la saisie: fichier ou répertoire.

```
#!/bin/bash
if test -f "$l"
        then exit 2;
 elif test -d "$1";
        then exit 3;
else exit 4
fi
case $? in
    2) cat  $1;  echo"fichier $1";;
    3)  cd  $1; 1s -l;    cd ..;;
    4)  echo "ni fichier, ni répertoire"
```

Le SHELL de Korn

➢ **Le Shell de Korn est un langage de programmation "interpréteur de commandes", il permet donc de réaliser de nouvelles commandes pour UNIX.**

➢ **Il a été développé par David G. Korn des Laboratoires Bell.**

➢ Avantages du Korn Shell

➡ **Le Korn shell regroupe les fonctions du shell Bourne et du C shell, tout en apportant de nouvelles propriétés et nouvelles fonctions, afin d'obtenir un shell plus convivial, plus puissant et plus rapide.**

✓ Le Korn shell a les possibilités supplémentaires suivantes :

- Un historique des commandes peut-être mis en place et utilisé (vi ou emacs) ;
- Une compatibilité Bourne shell ;
- Le shell Korn possède une arithmétique de nombres entiers incorporée, « Incorporée » signifie que les calculs et les comparaisons sont exécutés directement par le shell
- Des commandes Bourne avec de nouvelles fonctionnalités ;
- Des sélections par menu possible ;
- Des messages d'erreur plus significatifs ;
- ...

Les différences entre le Korn shell et le Bash sont suffisamment faibles, les versions actuelles de bash incluent un grand nombre de fonctions KSH.
On va s'intéressé uniquement pendant ce cours à la partie spécifique au KSH.

➳ Calculs arithmétiques :

- Le fonction "**let**" développée à l'origine pour KSH, intégrée actuellement dans bash permet d'effectuer les calculs directement pas le(s) SHELLs.

let *expression*

let i=1 ==> La valeur 1 est attribuée à la variable i.

incorrect : **let i=i + 1**

correct : **let i=i+1**

correct : **let "i=i + 1"**

Écritures abrégées. La commande ((*expression*)) correspond à « let *expression* ».

Les écritures ci-dessus peuvent se remplacer par:

((i=i+1))

ou

((i+=1))

- La fonction **expr** : cette fonction est externe aux SHELLs, autrement dit, les demandes de calculs se fait par des appels externes aux SHELLs.

Bien évidemment, effectuer les calculs directement par les fonctions intégrées au SHELL est plus beaucoup plus rapide que les fictions externes au SHELL.

Exemple1 : calcul.bash

```
#! /bin/bash
#nombre.bash
# compter de 1 a 10000
 i=1
while [ $i -le 10000 ]
  do
    echo $i
    i=`expr $i + 1`
  done
```

Exemple2 : calcul.ksh

```
#! /bin/ksh
#nombre.bash
# compter de 1 a 10000
integer i=1
while [ $i -le 10000 ]
  do
    echo $i
    let i=i+1
  done
```

L'exécution de deux programmes, dans les mêmes conditions done :

time calcul.bash ==> 0,116 s
time calcul.bash ==> 6,537 s

Si on compare les deux résultats, on constate que l'utilisation de la fonction interne au SHLL est presque 56 fois plus rapide que la fonction externe (sur des opérations simples !)

➳ **Les attribues des variables :**

- **Typeset :** Cette fonction permet de positionner, réinitialiser ou affecter les variables selon différentes options.

Syntaxe :

typeset -attribut variable [...]

attribut=filrux

i	**la variable est de type entier**
l	**majuscules transformées en minuscules**
u	**minuscules transformées en majuscules**
f	**la variable est une fonction**
x	**variable exportée (export)**
r	**la variable ne sera accessible qu'en lecture**

typeset -in *variable* <==> **integer** *variable*

Exemple1:

integer x=l6; Print $x ===> 16
typeset -il6; print $x ===> 10

Exemple2

typeset -u chaine=*"petit et GRAND" # minuscule ==> majuscule*
print $chaine ==> PETIT ET GRAND

Exemple2

typeset -l chaine *# majuscule ==> minuscule*
print $chaine ==> petit et grand

Exemple3

typeset -r var=valeur1 # définir une variable protégée
print $var ==> valeur1

typeset -r var=*valeur2* # modifier la variable valeur1
/bin/ksh: var: is read only # modification impossible

Exemple4 : Afficher les résultats de conversions des entiers de 1 au 32 dans les bases : Décimale, Binaire, octale et hexadécimale

```
#! /bin/ksh
# affichage en décimal, binaire, octal et hexadécimal

print "Decimal\t\t Binaire\t\t Octal\t\t Hexadecimal\t\t"
print "=========================================================="

k=1
while test $k -le 32
  do

  typeset -i10 k
  print -n "$k\t\t"
  typeset -i2 k
  print -n "$k\t\t"
  typeset -i8 k
  print -n "$k\t\t"
  typeset -i16 k
  print -n "$k\t\t"
  echo
  let k=k+1

  done
```

Extrait du résultat d'exécution :

```
Decimal.      Binaire              Octal        Hexadecimal
==========================================================
1             2#1                  8#1          16#1
2             2#10                 8#2          16#2
9             2#1001               8#11         16#9
10            2#1010               8#12         16#a
13            2#1101               8#15         16#d
14            2#1110               8#16         16#e
15            2#1111               8#17         16#f
16            2#10000              8#20         16#10
...             ...                ...          ...
32            2#100000             8#40         16#20
```

➽ *Les tableaux*

L'affectation à un élément se fait par:

nom[*index*]=*valeur*

La directive -A permet d'affecter séquentiellement à un tableau toute une liste de valeurs:

set -A tableau *un deux trois quatre*

Exemple :

set -A tableau *un deux trois quatre*
print ${tableau[*1*]} ==> deux

tableau[*10*]=*100*
print ${tableau[*]} ==> un deux trois quatre 100

***a=3*; print ${tableau[$a]}** ==> quatre

➽ Programmation de menus

✓ **Des menus peuvent être programmés simplement par la commande:**

```
select  var [ in texte ... ]
do
        commandes
done
```

Exemple :

```
select choix list edit quit
do
        case $choix
        list) more $*;;
        edit) vi $*;;
        quit) exit;;
        esac
done
```

➽ Les fonctions

Les fonctions permettent la division des scripts en petites parties, lesquelles sont plus faciles à tester et à réutiliser. Avant d'exister, une fonction doit être définie, de la façon suivante:

```
function nom
{
    commandes
}
```

Exemple 1: soit le script nommé carre.ksh

☑ La valeur est lue par la fonction **read**

```
#! /bin/ksh

function carre
{
read -p 'Rentrez un nombre '  val
let m=$val*$val
echo  $m
}

carre
```

☑ La valeur est donnée en **paramètre**

```
#! /bin/ksh

function carre
{
let m=$1*$1
echo  $m
}

carre $1
```

Exemple 2:

Lire un liste de nombres, et tester la parité

☑ Les nombres sont donnés en **paramètre**

➞ **Première Version !**

```
#!/bin/ksh
        # Fonction qui teste la parité d'une valeur

function pair_impair
{
  test $(($1%2)) -eq 0   # <==> test `expr $1 % 2` -eq 0
}


# Pour chaque nombre passé au programme

for n in $*     # "$*" représente la liste de paramètres
do
        # Vérification de la parité de ce nombre

    if pair_impair $n
            then
    echo "$n est pair"
            else
    echo "$n  est impair"
    fi
done
```

 # Cette version est compatible avec BASH mais pas avec SH

➞ **Deuxième version !**

```
#!/bin/ksh
   # Fonction qui teste la parité d'un nombre

pair_impair()
{
  test $(($1%2)) -eq 0   # <==> test `expr $1 % 2` -eq 0
}

        # Pour chaque nombre passé au programme

for n in $*
do
     # Vérification de la parité de ce nombre

      if pair_impair $n
  then
     echo "$n est pair"
  else
     echo "$n est impair"
  fi
done
```

 # La deuxième version est compatible avec BASH et SH