



Les systèmes d'exploitation

Filière: Génie Informatique

S2

Chapitre 7

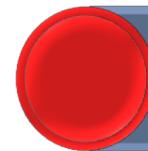
Les scripts shell



Qu'est-ce qu'un shell ?

Définition

Le **shell** est un programme qui sert d'interface en mode texte entre le noyau et l'utilisateur. Le **shell** est un interpréteur de commande et un langage de programmation . Le **shell** est une interface en mode texte dont le clavier est l'entrée et l'écran la sortie.



Qu'est-ce qu'un shell ?

Les différents environnements console (shells)

sh : Bourne Shell. L'ancêtre de tous les shells.

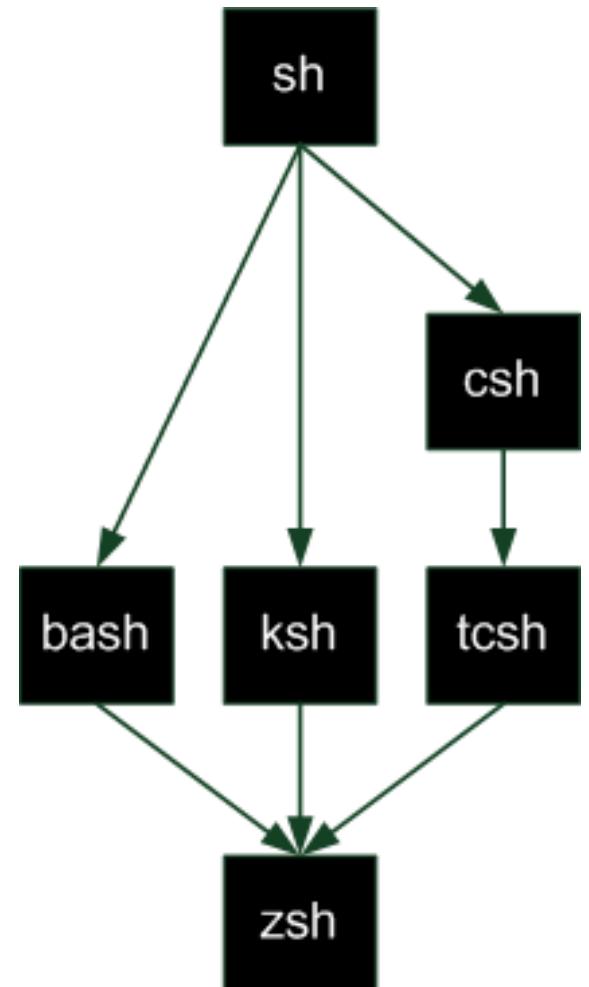
bash : Bourne Again Shell. Une amélioration du Bourne Shell, disponible par défaut sous Linux et Mac OS X.

ksh : Korn Shell. Un shell puissant assez présent sur les Unix propriétaires, mais aussi disponible en version libre, compatible avec bash.

csh : C Shell. Un shell utilisant une syntaxe proche du langage C.

tcsh : Tenex C Shell. Amélioration du C Shell.

zsh : Z Shell. Shell assez récent reprenant les meilleures idées de bash, ksh et tcsh.





Premier programme

Bonjour.sh

```
#!/bin/sh
# Ceci est un commentaire!
echo "Hello World" # Ceci est un commentaire aussi!
```

/bin/sh. C'est l'emplacement standard du *Bourne shell* sur à peu près tous les systèmes Unix.

- La deuxième ligne commence par un symbole spécial: **#**. Cela marque la ligne en tant que commentaire et il est complètement ignoré par le shell.
- La troisième ligne exécute la commande **echo** avec le paramètre **"Hello World"**.



Premier programme

Jusqu'à maintenant, le fichier Bonjour.sh ne possède pas la permission d'exécution. Donc, il faut lui donner le cette permission par la commande:

CHMOD 755

Pour exécuter le script il faut juste taper:

./Bonjour.sh* ou *bash Bonjour.sh

```
ubuntu@ubuntu-VirtualBox:~/Bureau$ ./script.sh
Hello World
```



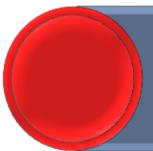
Le chemin ./

Quand vous tapez une commande ("ls" par exemple), le shell regarde dans le PATH qui lui indique où chercher le code de la commande.

Pour voir à quoi ressemble votre PATH, tapez dans votre console:

```
ubuntu@ubuntu-VirtualBox:~/Bureau$ echo $PATH  
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

C'est à dire que le shell va aller voir si la définition de la commande tapée ("ls" pour continuer sur le même exemple) se trouve dans **/usr/local/sbin** puis dans **/usr/local/bin**... jusqu'à ce qu'il la trouve.



Le chemin . /

Quand vous tapez une commande, le shell regarde dans le PATH qui lui indique où chercher le code de la commande.

Pour voir à quoi ressemble votre PATH, tapez dans votre console:

```
ubuntu@ubuntu-VirtualBox:~/Bureau$ echo $PATH  
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

C'est à dire que le shell va aller voir si la définition de la commande tapée ("ls" pour continuer sur le même exemple) se trouve dans **/usr/local/sbin** puis dans **/usr/local/bin**... jusqu'à ce qu'il la trouve.

Pour ajouter notre répertoire dans \$PATH

```
ubuntu@ubuntu-VirtualBox:~/Bureau$ export PATH=$PATH:$HOME/Bureau
```

Maintenant, on peut taper directement le nom du fichier exécutable
Pour une variable globale



Variables

Notez qu'il ne doit y avoir aucun espace autour du signe "=":

VAR=value fonctionne;

VAR = value ne fonctionne pas.

Dans le premier cas, le shell voit le symbole "=" et traite la commande comme une affectation de variable.

Dans le second cas, le shell suppose que VAR doit être le nom d'une commande et tente de l'exécuter.

Notez aussi que nous avons besoin des guillemets autour de la chaîne Hello World.

Le signe **\$** pour afficher la valeur d'une variable

```
1 #!/bin/sh
2 MON_MESSAGE="Hello World"
3 echo $MON_MESSAGE
```

Variables

Le shell ne se soucie pas des types de variables; ils peuvent stocker des chaînes, des entiers, des nombres réels - tout ce que vous voulez.

```
1 #!/bin/sh
2 MY_MESSAGE="Hello World"
3 MY_SHORT_MESSAGE=hi
4 MY_NUMBER=1
5 MY_PI=3.142
6 MY_OTHER_PI="3.142"
7 MY_MIXED=123abc
```

Variables

Les quotes

Il est possible d'utiliser des quotes pour délimiter un paramètre contenant des espaces. Il existe trois types de quotes :

- les apostrophes '' (simples quotes) ;
- les guillemets "" (doubles quotes) ;
- les accents graves `` (back quotes), qui s'insèrent avec Alt Gr + 7 sur un clavier AZERTY français.

Selon le type de quotes que vous utilisez, la réaction de bash ne sera pas la même.



Variables

Les quotes

Les simples quotes ''

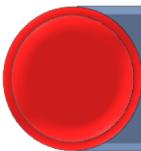
```
1 #!/bin/sh
2 message='Bonjour tout le monde'
3 echo 'Le message est : $message'
```

```
ubuntu@ubuntu-VirtualBox:~/Bureau$ ./script.sh
Le message est : $message
```

Les doubles quotes ""

```
1 #!/bin/sh
2 message='Bonjour tout le monde'
3 echo "Le message est : $message"
```

```
ubuntu@ubuntu-VirtualBox:~/Bureau$ ./script.sh
Le message est : Bonjour tout le monde
```



Variables

Les quotes

Les back quotes ``

Un peu particulières, les back quotes demandent à bash d'exécuter ce qui se trouve à l'intérieur.

```
1 #!/bin/sh
2 message=`pwd`
3 echo "Vous êtes dans le dossier $message"
```

```
ubuntu@ubuntu-VirtualBox:~/Bureau$ ./script.sh
Vous êtes dans le dossier /home/ubuntu/Bureau
```



Variables

Commande comme variable

On peut utiliser le résultat d'une commande comme suit:

```
1#!/bin/sh  
2nbre_lignes=$(wc -l < fichier.ext)
```

nbre_lignes contiendra le nombre de lignes contenu dans fichier.ext

Variables

Commande comme variable

Il existe des variables un peu spéciales

Nom	Fonction
\$*	contient tous les arguments passés à la fonction
\$#	contient le nombre d'argument
\$?	contient le code de retour de la dernière opération
\$0	contient le nom du script
\$n	contient l'argument n, n étant un nombre
\$!	contient le PID de la dernière commande lancée



Variables

Commande comme variable

Exemple : créer le fichier *script.sh* avec le contenu qui suit:

```
1 #!/bin/bash
2 echo "Nombre d'argument $"#
3 echo "Les arguments sont $"*
4 echo "Le second argument est $"2
5 echo "Et le code de retour du dernier echo est $"?
```

Lancez ce script avec un ou plusieurs arguments et vous aurez

```
ubuntu@ubuntu-VirtualBox:~/Bureau$ script.sh 1 2 3
Nombre d'argument 3
Les arguments sont 1 2 3
Le second argument est 2
Et le code de retour du dernier echo est 0
```



Variables

Les tableaux

```
#!/bin/bash
tab=("Mohamed" "Fatima zahra")
echo "Bonjour ${tab[0]} et ${tab[1]}"
tab[0]="Ahmed"
tab[1]="Zineb"
echo "Bonjour ${tab[0]} et ${tab[1]}
```

```
ubuntu@ubuntu-VirtualBox:~/Bureau$ script.sh
Bonjour Mohamed et Fatima zahra
Bonjour Ahmed et Zineb
```

Pour compter le nombre d'éléments du tableau :

len=\${#tab[*]} ou **echo \${#tab[@]}**

Pour afficher un élément :

echo \${tab[1]}

Pour afficher tous les éléments :

echo \${tab[@]}

ou bien

for i in \${!tab[@]}; do echo \${tab[i]}; done

ou encore

for ((i=0; i < \${#tab[@]}; i++)); do echo \${tab[i]}; done

Variables

Les arguments en ligne de commande

Pour passer des arguments en ligne de commande c'est encore une fois très simple. Chaque argument est numéroté et ensuite on l'appelle par son numéro :

```
#!/bin/bash  
echo "Bonjour $1 et $2"
```

```
ubuntu@ubuntu-VirtualBox:~/Bureau$ script.sh  
Bonjour et  
ubuntu@ubuntu-VirtualBox:~/Bureau$ script.sh Mohamed Fatima zahra  
Bonjour Mohamed et Fatima  
ubuntu@ubuntu-VirtualBox:~/Bureau$ script.sh Mohamed "Fatima zahra"  
Bonjour Mohamed et Fatima zahra  
ubuntu@ubuntu-VirtualBox:~/Bureau$
```

Commandes de base

Opérateur de test sur les fichiers

Elle permet de faire un test et de renvoyer 0 si tout s'est bien passé ou 1 en cas d'erreur

Syntaxe	Fonction réalisée
-e fichier	renvoie 0 si fichier existe.
-d fichier	renvoie 0 si fichier existe et est un répertoire.
-f fichier	renvoie 0 si fichier existe et est un fichier 'normal'.
-w fichier	renvoie 0 si fichier existe et est en écriture.
-x fichier	renvoie 0 si fichier existe et est exécutable.
f1 -nt f2	renvoie 0 si f1 est plus récent que f2.
f1 -ot f2	renvoie 0 si f1 est plus vieux que f2.



Commandes de base

Opérateur de test sur les fichiers

Exemple

```
#!/bin/bash  
test -f $1  
echo $?
```

Ou encore

```
#!/bin/bash  
[ -f $1 ]  
echo $?
```

```
ubuntu@ubuntu-VirtualBox:~/Bureau$ script.sh Fichier1  
0  
ubuntu@ubuntu-VirtualBox:~/Bureau$ script.sh Fichier2  
1
```

Donc: **test -f *Fichier*** est équivalent à **[-f *Fichier*]**

N'oubliez pas les espaces

Commandes de base

Opérateurs de comparaison numériques

Syntaxe1	Syntaxe2	Fonction réalisée
[\$a -lt 5]	((\$a < 5))	renvoie 0 si \$a est strictement inférieur à 5
[\$a -le 5]	((\$a <= 5))	renvoie 0 si \$a est inférieur ou égal à 5
[\$a -gt 5]	((\$a > 5))	renvoie 0 si \$a est strictement supérieur à 5
[\$a -ge 5]	((\$a >= 5))	renvoie 0 si \$a est supérieur ou égal à 5
[\$a -eq 5]	((\$a == 5))	renvoie 0 si \$a est égal à 5
[\$a -ne 5]	((\$a != 5))	renvoie 0 si \$a est différent de 5



Commandes de base

Opérateurs de comparaison numériques

Exemple

```
#!/bin/bash  
[ $1 -gt 0 ]  
echo $?
```

ou

```
#!/bin/bash  
(( $1>0 ))  
echo $?
```

```
#!/bin/bash  
[ $1 -eq 0 ]  
echo $?
```

ou

```
#!/bin/bash  
(( $1==0 ))  
echo $?
```

```
ubuntu@ubuntu-VirtualBox:~/Bureau$ script.sh 3  
0  
ubuntu@ubuntu-VirtualBox:~/Bureau$ script.sh -4  
1
```

```
ubuntu@ubuntu-VirtualBox:~/Bureau$ script.sh 0  
0  
ubuntu@ubuntu-VirtualBox:~/Bureau$ script.sh 7  
1
```

Commandes de base

Les opérateurs logiques

Ces opérateurs permettent d'exécuter ou non une commande en fonction du code de retour d'une autre commande. L'évaluation est faite de gauche à droite.

Évaluation de l'opérateur &&

Syntaxe

commande1 **&&** commande2

- La deuxième commande est exécutée uniquement si la première commande renvoie un code vrai.
- L'expression globale est vraie si les deux commandes renvoient vrai.

Opérateur	Signification
&&	ET logique
 	OU logique

Évaluation de l'opérateur ||

Syntaxe

commande1 **||** commande2

- La deuxième commande est exécutée uniquement si la première commande renvoie un code faux.
- L'expression globale est fausse si les deux commandes renvoient faux.

Commandes de base

Les opérateurs logiques

Exemple

Si le fichier donné au paramètre existe, alors [-f \$1] renvoie 0, donc [! -f \$1] renvoie 1. et par conséquence, la commande **echo** est exécutée

```
#!/bin/bash
[ ! -f $1 ] && echo "Fichier '$1' inexistant"
```

```
ubuntu@ubuntu-VirtualBox:~/Bureau$ script.sh noms.txt
Fichier 'noms.txt' inexistant
```

Si le fichier donné au paramètre existe, alors [-f \$1] renvoie 0, donc la commande **echo** est exécutée

```
#!/bin/bash
[ -f $1 ] || echo "Fichier '$1' existant"
```

```
ubuntu@ubuntu-VirtualBox:~/Bureau$ script.sh noms.txt
Fichier 'noms.txt' existant
```



Commandes de base

Combinaison des conditions logiques

$(\$1>0)$ et $(\$2>0$ ou $\$3>0)$ est équivalent à:

```
#!/bin/bash
[ \$1 -gt 0 ] -a [ \$2 -gt 0 -o \$3 -gt 0 ]
echo $?
```

↑ ↑
ET OU

Ou encore

```
#!/bin/bash
(( \$1 > 0 )) && (( \$2 > 0 || \$3 > 0 ))
echo $?
```



Commandes de base

Syntaxe à double crochets

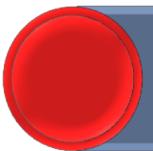
Ces conditions permettent tout ce qu'offrent les conditions à simples crochets et plus. En revanche, elles ne sont pas compatibles avec sh. Elles prennent la forme suivante.

```
#!/bin/bash
[[ -x $1 ]]
echo $?
```

Ces conditions améliorées proposent l'usage du wildcard comme en bash ainsi que des expressions régulières. Ainsi, il est possible d'avoir des conditions comme cela :

```
#!/bin/bash
[[ $1 == *at* ]]
echo $?
```

```
ubuntu@ubuntu-VirtualBox:~/Bureau$ script.sh objet
1
ubuntu@ubuntu-VirtualBox:~/Bureau$ script.sh Math
0
```



Commandes de base

L'arithmétique

La commande **Expr**

Syntaxe

expr \$nbr1 opérateur \$nbr2

expr \$chaine : expression régulière

NB: Attention, certains opérateurs ayant une signification particulière pour le shell, ceux ci doivent être protégés par un antislash.

Opérateurs	Signification
Opérateurs arithmétiques	
\$nb1 + \$nb2	Addition
\$nb1 - \$nb2	Soustraction
\$nb1 * \$nb2	Multiplication
\$nb1 / \$nb2	Division
\$nb1 % \$nb2	Modulo
Opérateurs de comparaison	
\$nb1 \> \$nb2	VRAI si \$nb1 est strictement supérieur à \$nb2
\$nb1 \>= \$nb2	VRAI si \$nb1 est supérieur ou égal à \$nb2
\$nb1 \< \$nb2	VRAI si \$nb1 est strictement inférieur à \$nb2
\$nb1 \<= \$nb2	VRAI si \$nb1 est inférieur ou égal à \$nb2
\$nb1 = \$nb2	VRAI si \$nb1 est égal à \$nb2
\$nb1 != \$nb2	VRAI si \$nb1 est différent de \$nb2



Commandes de base

L'arithmétique

La commande Expr

Syntaxe

expr \$nbr1 opérateur \$nbr2

expr \$chaine : expression régulière

NB: Attention, certains opérateurs ayant une signification particulière pour le shell, ceux ci doivent être protégés par un antislash.

Opérateurs	Signification
Opérateurs logiques	
\$chaine1 \& \$chaine2	VRAI si les 2 chaines sont vraies
\$chaine1 \ \$chaine2	VRAI si l'une des 2 chaines est vraie
Opérateurs divers	
-\$nb1	Opposé de \$nb1
\(expression \)	Regroupement
\$chaine : expression_reguliere	Compare la chaine avec l'expression régulière

Commandes de base

L'arithmétique

La commande Expr

Exemple

```
#!/bin/bash
nb=3
expr $nb + 5
expr $nb \* 6
expr $nb / 2
nb=10
expr $nb % 3
expr $nb - -5
```

```
ubuntu@ubuntu-VirtualBox:~/Bureau$ script.sh
8
18
1
1
15
```

Récupérer le résultat dans une variable

```
#!/bin/bash
nb=10
nb2=`expr $nb - 2`
echo $nb2
```



Commandes de base

L'arithmétique

La commande `(())`

Syntaxe

`((expression_arithmétique))`

La commande `(())` dispose de nombreux avantages par rapport à la commande `expr`

- Opérateurs supplémentaires
- Les arguments n'ont pas besoin d'être séparés par des espaces
- Les variables n'ont pas besoin d'être préfixées par le symbole \$
- Les caractères spéciaux du shell n'ont pas besoin d'être protégés par des antislashes
- Les affectations se font dans la commande
- Son exécution est plus rapide



Commandes de base

L'arithmétique

La commande `(())`

Une grande partie des opérateurs proviennent du langage C

Opérateurs	Signification
Opérateurs arithmétiques	
<code>nbr1 + nbr2</code>	Addition
<code>nbr1 - nbr2</code>	Soustraction
<code>nbr1 * nbr2</code>	Multiplication
<code>nbr1 / nbr2</code>	Division
<code>nbr1 % nbr2</code>	Modulo

Opérateurs	Signification
Opérateurs travaillant sur les bits	
<code>~nbr1</code>	Complément à 1
<code>nbr1 >> nbr2</code>	Décalage sur nbr1 de nbr2 bits à droite
<code>nbr1 << nbr2</code>	Décalage sur nbr1 de nbr2 bits à gauche
<code>nbr1 & nbr2</code>	ET bit à bit
<code>nbr1 nbr2</code>	OU bit à bit
<code>nbr1 ^ nbr2</code>	OU exclusif bit à bit



Commandes de base

L'arithmétique

La commande (())

Une grande partie des opérateurs proviennent du langage C

Opérateurs	Signification
Opérateurs de comparaison	
<code>nbr1 > nbr2</code>	VRAI si nbr1 est strictement supérieur à nbr2
<code>nbr1 >= nbr2</code>	VRAI si nbr1 est supérieur ou égal à nbr2
<code>nbr1 < nbr2</code>	VRAI si nbr1 est strictement inférieur à nbr2
<code>nbr1 <= nbr2</code>	VRAI si nbr1 est inférieur ou égal à nbr2
<code>nbr1 == nbr2</code>	VRAI si nbr1 est égal à nbr2
<code>nbr1 != nbr2</code>	VRAI si nbr1 est différent de nbr2



Commandes de base

L'arithmétique

La commande (())

Opérateurs	Signification
Opérateurs logiques	
!nbr1	Inverse la valeur de vérité de nbr1
&&	ET
	OU
Opérateurs divers	
-nbr1	Opposé de nbr1
nbr1 = expression	Assignement
(expression)	Regroupement
nbr1 <i>binop</i> = nbr2	<i>binop</i> représente l'un des opérateurs suivants : +, -, /, *, %, >>, <<, &, , ^. Equivalent à nbr1 = nbr1 <i>binop</i> nbr2



Commandes de base

L'arithmétique

La commande (())

Exemples

Ajouter 10 à nbr1
(2 méthodes différentes)

```
#!/bin/bash
nbr1=10
((nbr1=nbr1+10))
echo $nbr1
nbr1=10
((nbr1+=10))
echo $nbr1
```

Test si nbr1 est supérieur
à nbr2 et inversement

```
#!/bin/bash
nbr1=5
nbr2=6
((nbr1>nbr2))
echo $?
((nbr1<nbr2))
echo $?
```

Regroupements et tests logiques

```
#!/bin/bash
nbr1=2
nbr2=5
if (( (nbr1>0) && (nbr2>nbr1) ))
then
echo "nbr1 entre 0 et nbr2"
else
echo "nbr1 = 0 ou > à nbr2"
fi
```



Commandes de base

L'arithmétique

La commande LET

La commande let est équivalente à ((expression))

Syntaxe

let "expression"

Exemples

Multiplier nbr1 par 3

```
#!/bin/bash
nbr1=5
let "nbr1=nbr1*3"
echo $nbr1
```

Calculer le modulo de nbr1 par 2 et l'affecter à la variable nbr2

```
#!/bin/bash
nbr1=5
let "nbr2=nbr1%2"
echo $nbr1
echo $nbr2
```

Commandes de base

La commande IF

La structure de contrôle conditionnelle if permet d'effectuer une action en fonction d'une expression logique.

Syntaxe 1

```
if condition  
then  
    instructions;  
fi
```

Syntaxe 2

```
if condition  
then  
    instructions;  
else  
    instructions;  
fi
```

Syntaxe 3

```
if condition1  
then  
    instructions1;  
elif condition2  
then  
    instructions2;  
else  
    instructions4;  
fi
```

Commandes de base

La commande IF

Exemples

```
#!/bin/bash
if [[ $1 == [sS]ystème ]]
then
    echo "Accès autorisé"
else
    echo "Accès non autorisé"
fi
```

```
ubuntu@ubuntu-VirtualBox:~/Bureau$ script.sh système
Accès autorisé
ubuntu@ubuntu-VirtualBox:~/Bureau$ script.sh Système
Accès autorisé
ubuntu@ubuntu-VirtualBox:~/Bureau$ script.sh System
Accès non autorisé
```

```
#!/bin/bash
if [[ $1 == 1 ]];then echo "Choix 1";
elif [[ $1 == 2 ]];then echo "Choix 2";
elif [[ $1 == 3 ]];then echo "Choix 3";
else echo "Aucun choix";
fi
```

```
ubuntu@ubuntu-VirtualBox:~/Bureau$ script.sh 3
Choix 3
ubuntu@ubuntu-VirtualBox:~/Bureau$ script.sh 2
Choix 2
```



Commandes de base

La commande CASE

La structure de contrôle **case** permet elle aussi d'effectuer des tests. Elle permet d'orienter la suite du programme en fonction d'un choix de différentes valeurs.

Quand il y a un nombre important de choix, la commande **case** est plus appropriée que la commande **if**.



Commandes de base

La commande CASE

Syntaxe

```
case $variable in
    modèle1) commande1
    ...
    ;;
    modèle2) commande2
    ...
    ;;
    modèle3 | modèle4 | modèle5 ) commande3
    ...
    ;;
esac
```

Le shell compare la valeur de la variable aux différents modèle renseignés.

Lorsque la valeur correspond au modèle, les commandes faisant partie du bloc sont exécutées.

Les caractères `;;` permettent de fermer le bloc et de mettre fin au **case**.

Le shell continue à la première commande située sous **esac**.



Commandes de base

La commande CASE

Syntaxe

```
case $variable in  
    modele1) commande1
```

```
    ...
```

```
    ;;
```

```
    modele2) commande2
```

```
    ...
```

```
    ;;
```

```
    modele3 | modele4 | modele5 ) commande3
```

```
    ...
```

```
    ;;
```

```
esac
```

Le shell compare la valeur de la variable aux différents modèle renseignés.

Lorsque la valeur correspond au modèle, les commandes faisant partie du bloc sont exécutées.

Les caractères `;;` permettent de fermer le bloc et de mettre fin au **case**.

Le shell continue à la première commande située sous **esac**.

Il ne faut surtout pas oublier les caractères `;;` car cela engendrera une erreur.



Commandes de base

La commande CASE

Les caractères spéciaux :

Caractères spéciaux pour modèles de chaînes de caractères	Signification
Caractères spéciaux valables dans tous les shells :	
*	0 à n caractères
?	1 caractère quelconque
[abc]	1 caractère parmi ceux inscrits entre les crochets
[!abc]	1 caractère ne faisant pas partie de ceux inscrits entre les crochets

Commandes de base

La commande CASE

Les caractères spéciaux :

Caractères spéciaux non valides en Bourne Shell.

En bash, il faut activer l'option extglob (shopt -s extglob)

?(expression)	de 0 à 1 fois l'expression
*(expression)	de 0 à n fois l'expression
+(expression)	de 1 à n fois l'expression
@(expression)	1 fois l'expression
!(expression)	0 fois l'expression
?(expression1 expression2 ...)	alternatives
*(expression1 expression2 ...)	
+(expression1 expression2 ...)	
@(expression1 expression2 ...)	
!(expression1 expression2 ...)	

Commandes de base

La commande SELECT

Le select est une extension du case. La liste des choix possibles est faite au début et on utilise le choix de l'utilisateur pour effectuer un même traitement :

```
#!/bin/bash

select sys in "Windows" "Linux" "Mac OS" "Autre"
do
    echo "vous avez chosie le système" $sys
    break
done
```

```
smi@ubuntu:~$ ./scr7.sh
1) Windows
2) Linux
3) Mac OS
4) Autre
#? 2
vous avez chosie le système Linux
smi@ubuntu:~$ █
```



Commandes de base

La commande CASE

Travail à faire

Le script suivant permet de créer, modifier, visualiser et supprimer un fichier dans le répertoire d'exécution du script.

Il prend en argument un nom de fichier et affiche un menu.

Utilisation de case avec imbrication de if.

1(Creer)
2(Editer)
3(Afficher)
4(Supprimer)
Votre choix :



Commandes de base

La boucle FOR

Répéter l'action pour chaque élément de liste.

Syntaxe

```
#!/bin/bash
for VAR in LISTE
do
    # actions
done
```

Exemple

```
#!/bin/bash
for i in 1 2 3
do
    echo "i=$i"
done
```

Exemple

```
i=1
i=2
i=3
```



Commandes de base

La boucle FOR

Répéter l'action pour chaque valeur de i selon. Le contrôle d'arrêt est défini par une condition de sortie de la boucle.

Syntaxe

```
#!/bin/bash
for ((initialisation de VAR; contrôle de VAR; modification de VAR))
do
# actions
done
```

Exemple

```
#!/bin/bash
for ((i = 10; i >= 0; i -= 1))
do
echo $i
done
```

Sortie

Affichage des nombres de 10 à 0
i -= 1 équivalent à i = i - 1

Commandes de base

La boucle WHILE

Répéter l'action tant que la condition est vérifiée

Syntaxe

```
#!/bin/bash
while CONDITION
do
# actions
done
```

Exemple

```
#!/bin/bash
i=0
while [ $i -le 10 ]
do
echo $i
let i=1+$i
done
```

ou

```
#!/bin/bash
i=0
while ((i <= 10))
do
echo $i
((i += 1))
done
```

Sortie

Affichage des nombres de 0 à 10

Commandes de base

La boucle WHILE

Exemple: Analyser et interpréter les scripts suivants

```
#!/bin/bash
nbr=0
while ((nbr!=53))
do
    echo -e "Saisir 53 : \c"
    read nbr
done
exit 0
```

```
#!/bin/bash
cpt=0
while ((cpt<10))
do
    echo "Le compteur vaut : $cpt"
    ((cpt+=1))
done
exit 0
```



Commandes de base

La boucle WHILE

Exemple: Analyser et interpréter les scripts suivants

```
#!/bin/bash
somme=0
echo "Saisir un nombre, ^d pour afficher la somme"
while read nombre
do
    if [[ $nombre != +([0-9]) ]]
    then
        echo "$nombre n'est pas un nombre"
        continue
    fi
    ((somme+=nombre))
done
echo "La somme est de : $somme"
exit 0
```

Le mot clé **continue** permet de remonter aussitôt à la boucle while sans exécuter la commande suivante

Commandes de base

La boucle WHILE

Exemple: Analyser et interpréter les scripts suivants

```
#!/bin/bash
cpt=0
while ((cpt<10))
do
    echo "Le compteur vaut : $cpt"
    done
exit 0
```

Ce script provoque une boucle infinie car il manque l'incrémentation du compteur

```
#!/bin/bash
while true
do
    echo "Boucle infinie"
    done
exit 0
```

faire une boucle infinie



Commandes de base

La boucle UNTIL

A l'inverse de **while**, la commande **until** exécute les commandes situées entre le **do** et le **done** tant que la commande située à droite du **until** retourne un code faux.

Syntaxe

```
#!/bin/bash
until commande1
do
    commande2
    ...
done
```

Exemple

```
#!/bin/bash
nbr=0
until ((nbr==53))
do
    echo -e "Saisir 53 : \c"
    read nbr
done
exit 0
```

\c : pour ne pas retourner à la ligne

Commandes de base

BREAK et CONTINUE

Les commandes **break** et **continue** peuvent s'utiliser à l'intérieur des boucles for, while, until et select.

La commande **break** permet de sortir d'une boucle.

La commande **continue** permet de remonter à la condition d'une boucle.

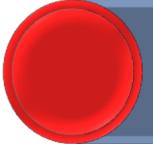
Syntaxe

Break: Quitter la boucle de premier niveau

break n: Quitter la boucle de niveau n

Continue: Remonter à la condition de la boucle de premier niveau

continue n: Remonter à la condition de la boucle de niveau n



Commandes de base

BREAK et CONTINUE

Les commandes **break** et **continue** peuvent s'utiliser à l'intérieur des boucles for, while, until et select.

La commande **break** permet de sortir d'une boucle.

La commande **continue** permet de remonter à la condition d'une boucle.

Syntaxe

Break: Quitter la boucle de premier niveau

break n: Quitter la boucle de niveau n

Continue: Remonter à la condition de la boucle de premier niveau

continue n: Remonter à la condition de la boucle de niveau n

Fin de chapitre 7