LES SCRIPTS

- Scripts

LES SCRIPTS EN SHELL

Il existe plusieurs environnements console : les shells

les fonctionnalités offertes par l'invite de commandes peuvent varier en fonction du **shell** que l'on utilise.

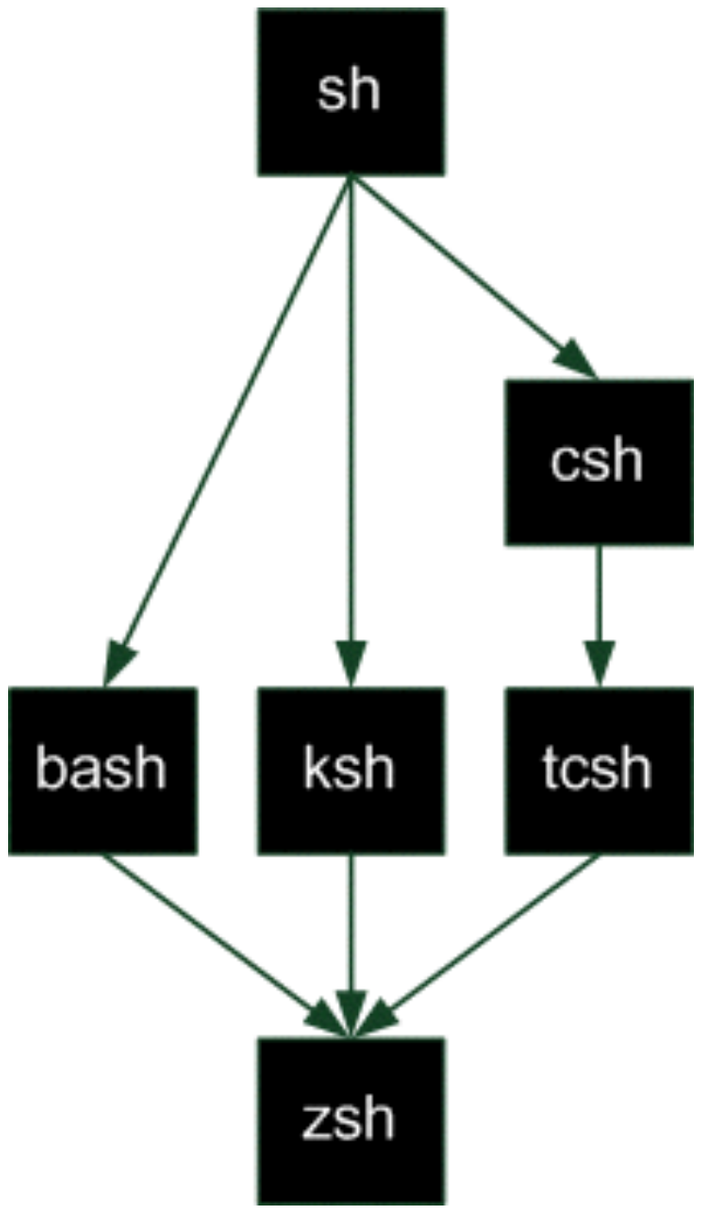
- **sh** : *Bourne Shell*. L'ancêtre de tous les shells.
- **bash** : *Bourne Again Shell*. Une amélioration du *Bourne Shell*, disponible par défaut sous Linux et Mac OS X.

- **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.


sh
csh
bash
ksh
tcsh
zsh

À QUOI SERT UN SHELL ?

Le shell est le programme qui gère l'invite de commandes. C'est donc le programme qui attend que vous rentriez des commandes

le bash dans ce cours car :

- on le **trouve par défaut sous Linux** et Mac OS X
- il rend l'écriture de scripts **plus simple que sh** ;
- il est plus **répandu** que ksh et zsh sous Linux.

En clair, le bash est un bon compromis entre sh (le plus compatible) et ksh / zsh (plus puissants).

CRÉATION DU FICHIER

Commençons par créer un nouveau fichier pour notre script. Le plus simple est d'ouvrir Vim en lui donnant le nom du nouveau fichier à créer :

```
$ vim essai.sh
```

INDIQUER LE NOM DU SHELL UTILISÉ PAR LE SCRIPT

Vim est maintenant ouvert et vous avez un fichier vide sous les yeux.

La première chose à faire dans un script shell est d'indiquer… quel shell est utilisé. En effet, comme je vous l'ai dit plus tôt, la syntaxe du langage change un peu selon qu'on utilise sh, bash, ksh, etc.

En ce qui nous concerne, nous souhaitons utiliser la syntaxe de bash, plus répandu sous Linux et plus complet que sh. Nous indiquons où se trouve le programme bash :

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

Exécution de commandes

Après le sha-bang, nous pouvons commencer à coder.
Le principe est très simple : il vous suffit d'écrire les commandes que vous souhaitez exécuter. Ce sont les mêmes que celles que vous tapiez dans l'invite de commandes !

- `ls` : pour lister les fichiers du répertoire.
- `cd` : pour changer de répertoire.
- `mkdir` : pour créer un répertoire.
- `grep` : pour rechercher un mot.
- `sort` : pour trier des mots.
- etc.

COMMENTAIRE

```
#!/bin/bash

# Affichage de la liste des fichiers
ls
```

Commencez par enregistrer votre fichier et fermez votre éditeur. Sous Vim, il suffit de taper `:wq` ou encore `:x`.
Vous retrouvez alors l'invite de commandes.

Donner les droits d'exécution au script

Si vous faites un `ls -l` pour voir votre fichier qui vient d'être créé, vous obtenez ceci :

```
$ ls -l
total 4
-rw-r--r-- 1 root root 17 2015-03-13 14:33 essai.sh
```

Ce qui nous intéresse ici, ce sont les droits sur le fichier : `-rw-r--r--`.
Si vous vous souvenez un petit peu du chapitre sur les droits, vous devriez vous rendre compte que notre script peut être lu par tout le monde (`r`), écrit uniquement par nous (`w`), et n'est pas exécutable (pas de `x`).

EXÉCUTION DU SCRIPT

Le script s'exécute maintenant comme n'importe quel programme, en tapant « ./ » devant le nom du script :

```
$ ./essai.sh
essai.sh
```

EXÉCUTION DE DÉBOGAGE

Plus tard, vous ferez probablement de gros scripts et risquerez de rencontrer des bugs. Il faut donc dès à présent que vous sachiez comment déboguer un script. Il faut l'exécuter comme ceci :

```
$ bash -x essai.sh
```

Le PATH est une variable système qui indique où sont les programmes exécutables sur votre ordinateur. Si vous tapez `echo $PATH` vous aurez la liste de ces répertoires.

Il vous suffit donc de déplacer ou copier votre script dans un de ces répertoires, comme `/bin`, `/usr/bin` ou `/usr/local/bin` (ou encore un autre répertoire du PATH). Notez qu'il faut être root pour pouvoir faire cela.

Une fois que c'est fait, vous pourrez alors taper simplement `essai.sh` pour exécuter votre programme et ce quel que soit le répertoire dans lequel vous vous trouverez !

CRÉER UNE VARIABLE ET L'AFFICHER

```
#!/bin/bash
message='Bonjour tout le monde'
echo 'Le message est : $message'
```

- http://openclassrooms.com/courses/reprenez-le-controle-a-l-aide-de-linux/afficher-et-manipuler-des-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.

```
message='Bonjour tout le monde'
echo 'Le message est : $message'
Le message est : $message
```

```
message='Bonjour tout le monde'
echo "Le message est : $message"
Le message est : Bonjour tout le monde
```

```
message=`pwd`
echo "Vous êtes dans le dossier $message"
Vous êtes dans le dossier /home/
```

- https://fr.wikipedia.org/wiki/Bourne-Again_shell

read : demander une saisie

```
#!/bin/bash

read nom
echo "Bonjour $nom ! »
```

```
#!/bin/bash

read nom prenom
echo "Bonjour $nom $prenom !"
```

`-p` : afficher un message de prompt

Bon : notre programme n'est pas très clair et nous devrions afficher un message pour que l'utilisateur sache quoi faire. Avec l'option `-p` de `read`, vous pouvez faire cela :

```
#!/bin/bash
read -p 'Entrez votre nom : ' nom
echo "Bonjour $nom !"
```

`-n` : limiter le nombre de caractères

Avec `-n`, vous pouvez au besoin couper au bout de X caractères si vous ne voulez pas que l'utilisateur insère un message trop long.

Exemple :

```
#!/bin/bash

read -p 'Entrez votre login (5 caractères max) : ' -n 5 nom
echo "Bonjour $nom !"
Entrez votre login (5 caractères max) : mathiBonjour mathi !
```

Notez que le bash coupe automatiquement au bout de 5 caractères sans que vous ayez besoin d'appuyer sur la touche `Entrée`. Ce n'est pas très esthétique du coup, parce que le message s'affiche sur la même ligne. Pour éviter cela, vous pouvez faire un `echo` avec des `\n`, comme vous avez appris à le faire plus tôt :

```
#!/bin/bash

read -p 'Entrez votre login (5 caractères max) : ' -n 5 nom
echo -e "\nBonjour $nom !"
Entrez votre login (5 caractères max) : mathi
Bonjour mathi
```

`-t` : LIMITER LE TEMPS AUTORISÉ POUR SAISIR UN MESSAGE

Vous pouvez définir un *timeout* avec `-t`, c'est-à-dire un nombre de secondes au bout duquel le `read` s'arrêtera.

```
#!/bin/bash

read -
p 'Entrez le code de désamorçage de la bombe (vous avez 5 s
econdes) : '  -t 5 code
echo -e "\nBoum !"
```

`-s` : ne pas afficher le texte saisi

Probablement plus utile, le paramètre `-s` masque les caractères que vous saisissez. Cela vous servira notamment si vous souhaitez que l'utilisateur entre un mot de passe :

```
#!/bin/bash

read -p 'Entrez votre mot de passe : ' -s pass
echo -
e "\nMerci ! Je vais dire à tout le monde que votre mot de pa
sse est $pass ! "
Entrez votre mot de passe :
Merci ! Je vais dire à tout le monde que votre mot de passe e
st supertopsecret38 !
```

EFFECTUER DES OPÉRATIONS MATHÉMATIQUES

```
#!/bin/bash
let "a = 5"
let "b = 2"
let "c = a + b"
echo $c
```

Les opérations utilisables sont :

- l'addition : + ;
- la soustraction : - ;
- la multiplication : * ;
- la division : / ;
- la puissance : ** ;
- le modulo (renvoie le reste de la division entière) : %.

QUELQUES EXEMPLES :

```
let "a = 5 * 3" # $a = 15
let "a = 4 ** 2" # $a = 16 (4 au carré)
let "a = 8 / 2" # $a = 4
let "a = 10 / 3" # $a = 3
let "a = 10 % 3" # $a = 1
```

- 10 / 3 = 3 car la division est entière (la commande ne renvoie pas de nombres décimaux) ;
- 10 % 3 renvoie 1 car le reste de la division de 10 par 3 est 1. En effet, 3 « rentre » 3 fois dans 10 (ça fait 9), et il reste 1 pour aller à 10.

Les variables d'environnement : env

Les variables d'environnement sont des variables que l'on peut utiliser dans n'importe quel programme. On parle aussi parfois de **variables globales**. Vous pouvez afficher toutes celles que vous avez actuellement en mémoire avec la commande `env` :

```
TERM_PROGRAM=Apple_Terminal
SHELL=/bin/bash
TERM=xterm-256color
TMPDIR=/var/folders/mr/y17qq2md487656pgypvzykkh0000gn/T/
Apple_PubSub_Socket_Render=/tmp/launch-VHuiu8/Render
TERM_PROGRAM_VERSION=326
TERM_SESSION_ID=9127AC63-CEC4-4B9D-86A8-C72236EC2A25
USER=raoufbabari
SSH_AUTH_SOCK=/tmp/launch-M7ZUJM/Listeners
__CF_USER_TEXT_ENCODING=0x1F5:0:1
PATH=/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin
__CHECKFIX1436934=1
PWD=/Users/raoufbabari
LANG=fr_CA.UTF-8
SHLVL=1
HOME=/Users/raoufbabari
LOGNAME=raoufbabari
_=/usr/bin/env
OLDPWD=/bin
```

ON TROUVE :

- `SHELL` : indique quel type de shell est en cours d'utilisation (sh, bash, ksh…) ;
- `PATH` : une liste des répertoires qui contiennent des exécutables que vous souhaitez pouvoir lancer sans indiquer leur répertoire. Nous en avons parlé un peu plus tôt. Si un programme se trouve dans un de ces dossiers, vous pourrez l'invoquer quel que soit le dossier dans lequel vous vous trouvez ;
- `EDITOR` : l'éditeur de texte par défaut qui s'ouvre lorsque cela est nécessaire ;
- `HOME` : la position de votre dossier `home` ;
- `PWD` : le dossier dans lequel vous vous trouvez ;
- `OLDPWD` : le dossier dans lequel vous vous trouviez auparavant.

COMMANDES LINUX

- Travail maison :
- Tail, heap, cut sort grep

Les variables des paramètres

Comme toutes les commandes, vos scripts bash peuvent eux aussi accepter des paramètres. Ainsi, on pourrait appeler notre script comme ceci :

```
./variables.sh param1 param2 param3
```

En effet, des variables sont automatiquement créées :

- **`$#` : contient le nombre de paramètres ;**
- `$0` : contient le nom du script exécuté (ici `./variables.sh`) ;
- `$1` : contient le premier paramètre ;
- `$2` : contient le second paramètre ;
- … ;
- **$9** : contient le 9e paramètre.

Essayons :

```
#!/bin/bash
echo "Vous avez lancé $0,y a $# paramètres"
echo "Le paramètre 1 est $1 »
#shift
```

```
$ ./variables.sh param1 param2 param3
Vous avez lancé ./variables.sh,
 il y a 3 paramètres
Le paramètre 1 est param1
shift est généralement utilisé dans une boucle
qui permet de traiter les paramètres un par un.
```

TABLEAU

```
tableau=('valeur0' 'valeur1' 'valeur2')
```

Cela crée une variable `tableau` qui contient trois valeurs (`valeur0`, `valeur1`, `valeur2`).

Pour accéder à une case du tableau, il faut utiliser la syntaxe suivante :

```
${tableau[2]}
```

… ceci affichera le contenu de la case n° 2 (donc `valeur2`).

```
#!/bin/bash

tableau=('valeur0' 'valeur1' 'valeur
2')
tableau[5]='valeur5'
echo ${tableau[1]}
```

À votre avis, que va afficher ce script ?

Réponse :

```
valeur1
```

Vous pouvez afficher l'ensemble du contenu du tableau d'un seul coup en utilisant `${tableau[*]}` :

```
#!/bin/bash

tableau=('valeur0' 'valeur1' 'valeur2')
tableau[5]='valeur5'
echo ${tableau[*]}
valeur0 valeur1 valeur2 valeur5
```

TESTS

```
#!/bin/bash
nom="Bruno"
if [ $nom = "Bruno" ]
then
        echo "Salut Bruno !"
fi
```

À la place du mot `test`, il faut indiquer votre test. C'est à cet endroit que vous testerez la valeur d'une variable, par exemple. Ici, nous allons voir un cas simple où nous testons la valeur d'une chaîne de caractères,

Faisons quelques tests sur un script : `conditions.sh` :

```
#!/bin/bash
nom="Bruno"
if [ $nom = "Bruno" ]
then
        echo "Salut Bruno !"
else
        echo "J'te connais pas, ouste !"
fi
```

le premier paramètre (`$1`) envoyé au script :

```
#!/bin/bash
if [ $1 = "Bruno" ]
then
        echo "Salut Bruno !"
else
        echo "J'te connais pas, ouste !"
fi
```

On peut reprendre notre script précédent et l'adapter pour utiliser des `elif` :

```
#!/bin/bash
if [ $1 = "Bruno" ]
then
        echo "Salut Bruno !"
elif [ $1 = "Michel" ]
then
        echo "Bien le bonjour Michel"
elif [ $1 = "Jean" ]
then
        echo "Hé Jean, ça va ?"
else
        echo "J'te connais pas, ouste !"
fi
```

LES DIFFÉRENTS TYPES DE TESTS

Il est possible d'effectuer trois types de tests différents en bash :

- des tests sur des chaînes de caractères ;
- des tests sur des nombres ;
- des tests sur des fichiers.

TESTS SUR DES CHAÎNES DE CARACTÈRES

En bash toutes les variables sont considérées comme des chaînes de caractères.

Il est donc très facile de tester ce que vaut une chaîne de caractères. Vous trouverez les différents types de tests disponibles sur le tableau suivant.

Vérifions par exemple si deux paramètres sont différents :

```
#!/bin/bash
if [ $1 != $2 ]
then
        echo "Les 2 paramètres sont différents !"
else
        echo "Les 2 paramètres sont identiques !"
fi
```

COMPARAISON DE CHAINES

`$chaine1 = $chaine2`
Vérifie si les deux chaînes sont identiques. **Notez que bash est sensible à la casse** : « b » est donc différent de « B ».

Il est aussi possible d'écrire « == » pour les habitués du langage C.

`$chaine1 != $chaine2`
Vérifie si les deux chaînes sont différentes.

`-z $chaine`
Vérifie si la chaîne est vide.

`-n $chaine`
Vérifie si la chaîne est non vide.

```
#!/bin/bash
if [ -z $1 ]
then
        echo "Pas de paramètre"
else
        echo "Paramètre présent"
fi
$ ./conditions.sh
Pas de paramètre
$ ./conditions.sh param
Paramètre présent
```

COMPARAISON DES NOMBRES

`$num1 -eq $num2`
Vérifie si les nombres sont égaux (**eq***ual*). À ne pas confondre avec le « = » qui, lui, compare deux chaînes de caractères.
`$num1 -ne $num2`
Vérifie si les nombres sont différents (**n***on***e***qual*).
Encore une fois, ne confondez pas avec « != » qui est censé être utilisé sur des chaînes de caractères.
`$num1 -lt $num2`
Vérifie si `num1` est inférieur (<) à `num2` (**l***ower***t***han*).
`$num1 -le $num2`
Vérifie si `num1` est inférieur ou égal (<=) à `num2` (**l***ower***or e***qual*).
`$num1 -gt $num2`
Vérifie si `num1` est supérieur (>) à `num2` (**g***reater***t***han*).
`$num1 -ge $num2`
Vérifie si `num1` est supérieur ou égal (>=) à `num2` (**g***reater***or e***qual*).

```
#!/bin/bash
if [ $1 -ge 20 ]
then
        echo "Vous avez envoyé 20 ou plus"
else
        echo "Vous avez envoyé moins de 20"
fi
$ ./conditions.sh 23
Vous avez envoyé 20 ou plus
$ ./conditions.sh 11
Vous avez envoyé moins de 20
```

TESTS SUR DES FICHIERS

`-e $nomfichier`
Vérifie si le fichier existe.
`-d $nomfichier`
Vérifie si le fichier est un répertoire. sous Linux, tout est considéré comme un fichier, même un répertoire !
`-f $nomfichier`
Vérifie si le fichier est un… fichier. Un vrai fichier cette fois, pas un dossier.
`-L $nomfichier`
Vérifie si le fichier est un lien symbolique (raccourci).
`-r $nomfichier`
Vérifie si le fichier est lisible (r).
`-w $nomfichier`
Vérifie si le fichier est modifiable (w).
`-x $nomfichier`
Vérifie si le fichier est exécutable (x).
`$fichier1 -nt $fichier2`
Vérifie si `fichier1` est plus récent que `fichier2` (**n***ewer***t***han*).
`$fichier1 -ot $fichier2`
Vérifie si `fichier1` est plus vieux que `fichier2` (**o***lder***t***han*).

Je vous propose de faire un script qui demande à l'utilisateur d'entrer le nom d'un répertoire et qui vérifie si c'en est bien un :

```
#!/bin/bash
read -p 'Entrez un répertoire : ' repertoire
if [ -d $repertoire ]
then
        echo "Bien, vous avez compris ce que j'ai dit !"
else
        echo "Vous n'avez rien compris..."
fi
Entrez un répertoire : /home
Bien, vous avez compris ce que j'ai dit !
Entrez un répertoire : rienavoir.txt
Vous n'avez rien compris…
```

Notez que bash vérifie au préalable que le répertoire existe bel et bien.

Effectuer plusieurs tests à la fois

Dans un `if`, il est possible de faire plusieurs tests à la fois. En général, on vérifie :

- si un test est vrai **ET** qu'un autre test est vrai ;
- si un test est vrai **OU** qu'un autre test est vrai.
- **&&** : signifie « et » ;
- **||** : signifie « ou ».

exemple :

```
#!/bin/bash
if [ $# -ge 1 ] && [ $1 = 'koala' ]
then
        echo "Bravo !"
        echo "Vous connaissez le mot de passe"
else
        echo "Vous n'avez pas le bon mot de passe"
fi
```

Le test vérifie deux choses :

- qu'il y a au moins un paramètre (« si `$#` est supérieur ou égal à 1 ») ;
- que le premier paramètre est bien `koala` (« si `$1` est égal à `koala` »).

Si ces deux conditions sont remplies, alors le message indiquant que l'on a trouvé le bon mot de passe s'affichera.

```
$ ./conditions.sh koala
Bravo !
Vous connaissez le mot de passe
```

INVERSER UN TEST

Il est possible d'inverser un test en utilisant la négation. En bash, celle-ci est exprimée par le point d'exclamation « `!` ».

```
if [ ! -e fichier ]
then
        echo "Le fichier n'existe pas"
fi
```

Vous en aurez besoin, donc n'oubliez pas ce petit point d'exclamation.

case : tester plusieurs conditions à la fois

avec un `case` :

```
#!/bin/bash
case $1 in
        "Bruno")
                echo "Salut Bruno !"
                ;;
        "Michel")
                echo "Bien le bonjour Michel"
                ;;
        "Jean")
                echo "Hé Jean, ça va ?"
                ;;
        *)
                echo "J'te connais pas, ouste !"
                ;;
esac
```

Analysons la structure du `case` !

```
case $1 in
```

Tout d'abord, on indique que l'on veut tester la valeur de la variable `$1`. Bien entendu, vous pouvez remplacer `$1` par n'importe quelle variable que vous désirez tester.

```
"Bruno")
```

Là, on teste une valeur. Cela signifie « Si `$1` est égal à Bruno ». Notez que l'on peut aussi utiliser une étoile comme joker : « `B*` » acceptera tous les mots qui commencent par un B majuscule.

Si la condition est vérifiée, tout ce qui suit est exécuté jusqu'au prochain double point-virgule :

```
;;
```

Important, il ne faut pas l'oublier : le double point-virgule dit à bash d'arrêter là la lecture du `case`. Il saute donc à la ligne qui suit le `esac` signalant la fin du `case`.

```
*)
```

C'est en fait le « else » du `case`. Si aucun des tests précédents n'a été vérifié, c'est alors cette section qui sera lue.

```
esac
```

Marque la fin du `case` (`esac`, c'est « case » à l'envers !).

Nous pouvons aussi faire des « ou » dans un `case`. Dans ce cas, petit piège, il ne faut pas mettre deux | | mais un seul ! Exemple :

```
#!/bin/bash

case $1 in
        "Chien" | "Chat" | "Souris")
                echo "C'est un mammifère"
                ;;
        "Moineau" | "Pigeon")
                echo "C'est un oiseau"
                ;;
        *)
                echo "Je ne sais pas ce que c'est"
                ;;
esac
```

EN RÉSUMÉ

- On effectue des tests dans ses programmes grâce aux `if`, `elif`, `else`, `fi`.
- On peut comparer deux chaînes de caractères entre elles, mais aussi des nombres. On peut également effectuer des tests sur des fichiers : est-ce que celui-ci existe ? Est-il exécutable ? Etc.
- Au besoin, il est possible de combiner plusieurs tests à la fois avec les symboles `&&` (ET) et `||` (OU).
- Le symbole `!` (point d'exclamation) exprime la négation dans une condition.
- Lorsque l'on effectue beaucoup de tests sur une même variable, il est parfois plus pratique d'utiliser un bloc `case in… esac` plutôt qu'un bloc `if… fi`.

MENUS

```
#!/bin/bash

echo Enter 1 to exit
echo Enter 2 for Hello
echo Enter 3 for date
echo Enter 4 for globals
echo Enter 5 for all
echo Enter 6 for locals
opt=""
while [ 1 ]; do
read opt
if [ $opt -eq 1 ]; then
echo Exiting...
exit
elif [ $opt -eq 2 ]; then
echo Hello World!
echo I am alive.
elif [ $opt -eq 3 ]; then
echo `date`            # Format for date/time
elif [ $opt -eq 4 ]; then
echo Global environment variables are...
printenv
elif [ $opt -eq 5 ]; then
echo All environment variables are ...
set    # List all the variables
else
echo bad option
fi
done
```

AUTRE EXEMPLE

```
#!/bin/bash
# A menu driven shell script sample template
## ----------------------------------
# Step #1: Define variables
# ----------------------------------
RED='\033[0;41;30m'
STD='\033[0;0;39m'

# ----------------------------------
# Step #2: User defined function
# ----------------------------------
pause(){
    read -p "Press [Enter] key to continue..." EnterKey
}

one(){
    echo "one() called"
    pause
}

# do something in two()
two(){
    echo "two() called"
    pause
}

# function to display menus
show_menus() {
    clear
    echo "~~~~~~~~~~~~~~~~~~~~~"
    echo " M A I N - M E N U"
    echo "~~~~~~~~~~~~~~~~~~~~~"
    echo "1. Set Terminal"
    echo "2. Reset Terminal"
    echo "3. Exit"
}
```

```
# read input from the keyboard and take a action
# invoke the one() when the user select 1 from the menu option.
# invoke the two() when the user select 2 from the menu option.
# Exit when user the user select 3 form the menu option.
read_options(){
    local choice
    read -p "Enter choice [ 1 - 3] " choice
case $choice in
    1) one ;;
    2) two ;;
    3) exit 0;;
    *) echo -e "${RED}Error...${STD}" && sleep 2
    esac
}
# ----------------------------------
# Step #4: Main logic - infinite loop
# ----------------------------------
while true
do
show_menus
read_options
done
```

```
#!/bin/bash
clear
echo "This is information provided by mysystem.sh.  Program starts now."

echo "Hello, $USER"
echo

echo "Today's date is `date`, this is week `date +"%V"`."
echo

echo "These users are currently connected:"
w | cut -d " " -f 1 - | grep -v USER | sort -u
echo

echo "This is `uname -s` running on a `uname -m` processor."
echo

echo "This is the uptime information:"
uptime
echo

echo "That's all folks!"
```

EXEMPLE MENU

```
 1 #!/bin/bash
 2 opt=""
 3 enter=""
 4 while [ 1 ]; do
 5 clear
 6 echo \#############################
 7 echo \#  Entrer 1 pour quitter    \#
 8 echo \#  Entrer 2 pour Hello      \#
 9 echo \#  Entrer 3 pour afficher les utilisateurs      \      \#
10 echo \#############################
11 read opt
12 # Choix 1
13 if [ $opt -eq 1 ]; then
14 echo Bye Bye
15 exit
16 #Choix 2
17 elif [ $opt -eq 2 ]; then
18 echo Hello!
19 read enter
20 # Choix 3
21 elif [ $opt -eq 3 ]; then
22 echo Utilisateurs!
23 w
24 read enter
25 # Mauvais choix
26 else
27 echo Mauvais Choix
28 read enter
29 fi
30 done
31 # fin du script
```