

Cours des Systèmes d'Exploitation LINUX

ENSAM – Casablanca
2022-2023

Chapitre 6

Programmation en Shell

Introduction

Le Shell est plus qu'un interpréteur de commandes : c'est également un puissant langage de programmation. Cela n'est pas propre à Linux ; tout système d'exploitation offre cette possibilité d'enregistrer dans des fichiers des suites de commandes que l'on peut invoquer par la suite. Mais aucun système d'exploitation n'offre autant de souplesse et de puissance que le Shell Linux dans ce type de programmation. De plus, l'existence de plusieurs Shells conduit à plusieurs langages. Sous Linux, un fichier contenant des commandes est appelé script et nous n'emploierons plus que ce terme dans la suite. De même nous utiliserons le terme Shell pour désigner à la fois l'interpréteur de commandes et le langage correspondant.

Quelques consignes

Le Shell est un langage interprété ; en conséquence tout changement dans le système sera pris en compte par un script lors de sa prochaine utilisation (il est inutile de “ recompiler ” les scripts).

Il est tout à fait possible d'écrire et d'invoquer des scripts dans un certain Shell tout en utilisant un autre shell en interactif. En particulier, il est très fréquent (mais non obligatoire) d'utiliser un TC-shell en tant que “login shell” et le Bourne-shell ou un autre shell pour l'écriture des scripts.

Les scripts les plus simples (listes de commandes) seront identiques quel que soit le Shell, mais dès que des instructions de tests ou d'itérations sont nécessaires, les syntaxes du Bourne-shell, du Bash et du C-shell diffèrent.

Quelques consignes

Un script Shell n'est rien d'autre qu'un fichier texte dans lequel sont inscrites un certain nombre de commandes compréhensibles par votre interpréteur de commandes. Etant donné que Linux ne prend pas en compte les extensions des fichiers, vous êtes libre de nommer vos fichiers de script comme vous voulez. Mais il est souhaitable de mettre l'extension « .sh », afin qu'il soit reconnaissable par vous-même et par les éditeurs.

Pour le Shell que nous allons utiliser pendant ce cours, ça sera le Bash.

Pour l'éditeur vous êtes libre de choix :

- ❑ Le plus facile serait d'utiliser un éditeur graphique.
- ❑ Le plus intéressant et instructive serait le **Vim**, **emacs** en mode CLI,
- ❑ Ceux qui marchent à tout les coup : **nano**, **gedit**

Les éditeurs en ligne de commandes

Emacs : éditeur de texte très puissant, extensible et personnalisable. Emacs peut servir d'environnement de développement pour beaucoup de langages (**LaTeX** avec l'extension auctex, html...).

Nano : éditeur de texte en ligne de commande très simple, installé par défaut sur Ubuntu.

Vi/Vim : est très apprécié des développeurs pour toutes ses fonctions qui en font un très bon IDE (coloration syntaxique de 200 langages, complétion automatique, comparaison de fichiers, recherche évoluée, ...) et est extensible par des scripts.

Les éditeurs graphique

Gedit: éditeur de texte par défaut d'Ubuntu.

Gvim : vim avec une interface graphique.

TEA : un éditeur multiplate-forme qui propose de nombreuses fonctionnalités.

Emacs : Emacs peut aussi être utilisé avec une interface graphique si on installe les paquets spécifiques.

Premiers exemples

Le premier exemple de script Shell :

```
#!/bin/bash  
echo Bonjour # Un commentaire  
# Un autre commentaire
```

La première ligne du script doit être exclusivement utilisée pour indiquer qu'il s'agit bien d'un script et non pas d'un fichier texte classique, donc elle doit toujours commencer par le chemin de l'interpréteur de commandes « `#!/bin/Bash` »

Les commentaires du code sont précédés de `#`, tout ce qui s'écrit sur la ligne après est considéré comme commentaire.

Éditez ce script puis enregistrez-le sous le nom **scr1.sh**. Pour rendre le fichier exécutable, il faut ajouter le droit d'exécution :

```
chmod u+x scr1.txt
```

Pour exécuter le script depuis la ligne de commande :

```
$/scr1.sh ou $bash scr1.sh
```


Premiers exemples

Maintenant que nous avons essayé un script basique avec la commande **echo**, nous pouvons ajouter à la suite autant de commandes shell que l'en veut. Par exemple :

```
#!/bin/bash
clear
echo -n Vous utilisez le système
uname
echo
echo -n La version de votre noyau est
uname -r
```

L'exécution de ce script donne le résultat :

```
smi@ubuntu:~$ ./scr2.sh

Vous utilisez le système :Linux
la version de noyau est :3.5.0-17-generic
```

Les variables - système

On en distingue trois types : utilisateur, système et spéciales.

Les variables système :

Le **bash** et le système Linux d'une manière générale utilise ses propres variables pour fonctionner. Elles comportent des informations sur l'environnement du système (peuvent être visualisées en tapant la commande **env**). Les variables système sont nommées en majuscules pour éviter d'entrer en conflit avec les variables que vous pourriez créer.

Exemples de ces variables:

HOME, LOGNAME, USER, SHELL, PATH, PWD, LANG, MAIL ...

Les variables - utilisateur

Les variables utilisateur :

Le principe est de pourvoir affecter un contenu à un nom de variable, généralement un chaîne de caractère, ou des valeurs numériques. Pour les variables utilisateur, Il n'y a pas besoin de déclarer les variables comme en **langage C**, par exemple. On peut directement affecter une valeur à une variable de cette façon :

```
a=100
```

```
nom= khalid
```

Pour utiliser le contenus des variables qui ont été affectées, on utilise le signe \$.

```
echo $nom
```

Un nom de variable obéit à certaines règles :

- ❑ Il peut être composé de lettres minuscules, majuscules, de chiffres, de caractères de soulignement
- ❑ Le premier caractère ne peut pas être un chiffre
- ❑ Le taille d'un nom est en principe illimité (il ne faut pas abuser non plus)
- ❑ Les conventions veulent que les variables utilisateur soient en minuscules pour les différencier des variables système. [REDACTED] e l'utilisateur.

Les variables – spéciales

Les variables spéciales :

Le Shell prédéfinit des variables facilitant la programmation en fournissant des informations spéciales au moment de l'exécution du script:

0 contient le nom sous lequel le script est invoqué,

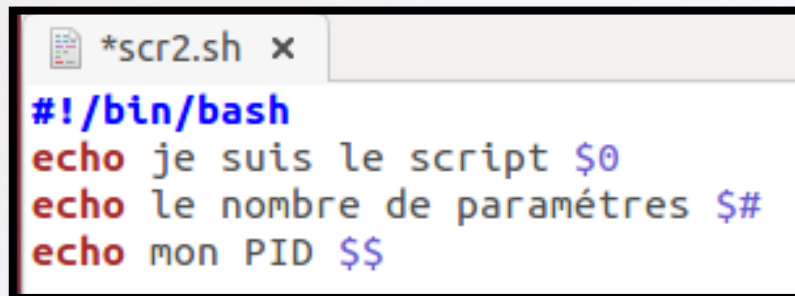
contient le nombre de paramètres passés en argument,

* contient la liste des paramètres passés en argument,

? contient le code de retour de la dernière commande exécutée,

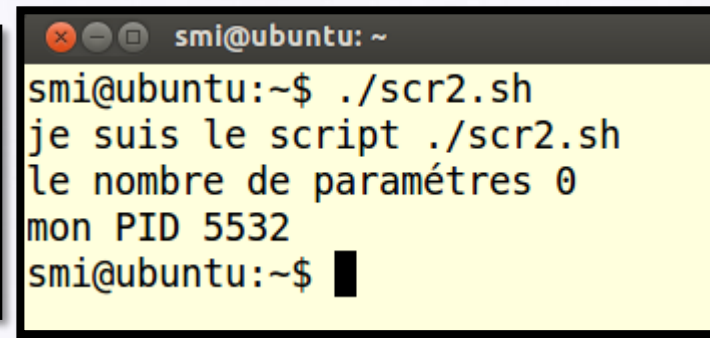
\$ contient le numéro de processus (PID) du Shell (décimal).

Exemple:



A screenshot of a text editor window showing the contents of a script file named `*scr2.sh`. The script starts with a shebang `#!/bin/bash` and contains three `echo` commands that use special shell variables: `$0` for the script name, `$#` for the number of arguments, and `$$` for the shell's PID.

```
*scr2.sh x
#!/bin/bash
echo je suis le script $0
echo le nombre de paramètres $#
echo mon PID $$
```



A screenshot of a terminal window with the prompt `smi@ubuntu: ~`. The user has executed the command `./scr2.sh`. The script's output is displayed on the following lines: `je suis le script ./scr2.sh`, `le nombre de paramètres 0`, and `mon PID 5532`. The prompt returns to `smi@ubuntu:~$`.

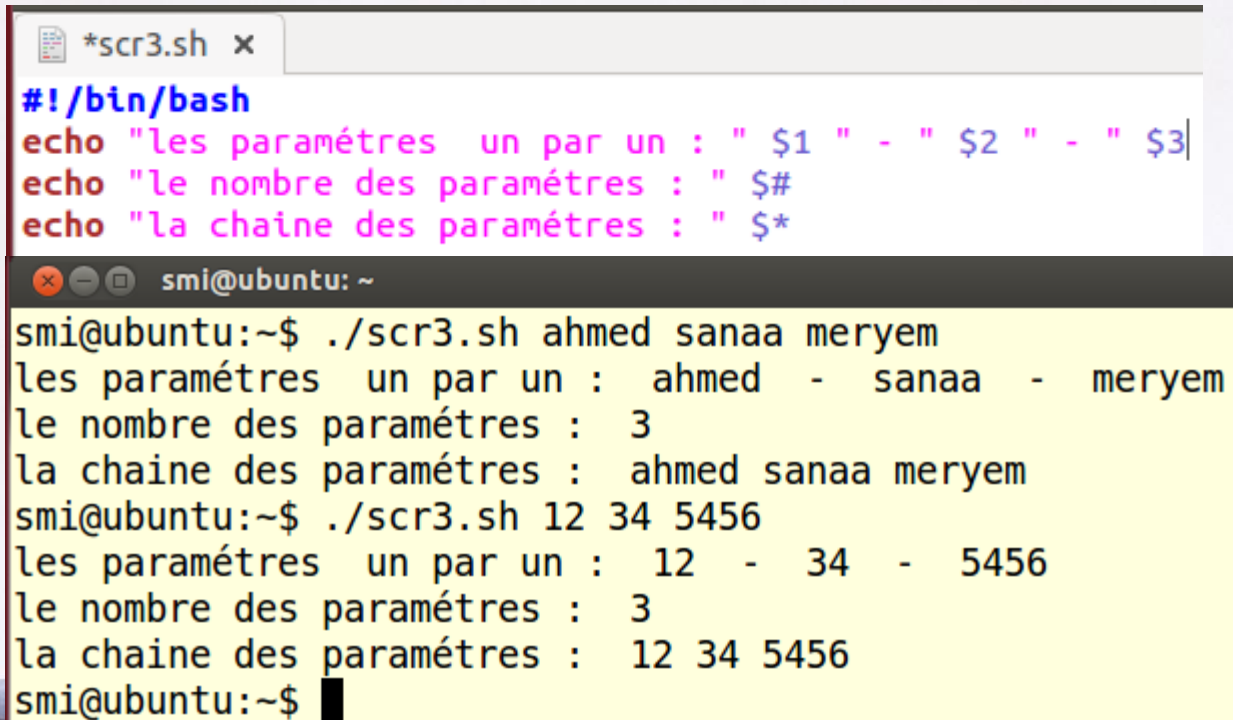
```
smi@ubuntu: ~
smi@ubuntu:~$ ./scr2.sh
je suis le script ./scr2.sh
le nombre de paramètres 0
mon PID 5532
smi@ubuntu:~$
```

Les variables – spéciales

Le passage des paramètres

On peut rendre un script plus interactif en lui transmettant des arguments lors de l'invocation.

Pour ce faire, les variables 1, 2, ..., 9 permettent de désigner respectivement le premier, le deuxième, ..., le neuvième paramètre associés à l'invocation du script.



The screenshot shows a terminal window with a tab labeled `*scr3.sh`. The script content is as follows:

```
#!/bin/bash
echo "les paramètres un par un : " $1 " - " $2 " - " $3
echo "le nombre des paramètres : " $#
echo "la chaine des paramètres : " $*
```

Below the script content, the terminal shows the execution of the script with two different sets of arguments:

```
smi@ubuntu: ~
smi@ubuntu:~$ ./scr3.sh ahmed sanaa meryem
les paramètres un par un : ahmed - sanaa - meryem
le nombre des paramètres : 3
la chaine des paramètres : ahmed sanaa meryem
smi@ubuntu:~$ ./scr3.sh 12 34 5456
les paramètres un par un : 12 - 34 - 5456
le nombre des paramètres : 3
la chaine des paramètres : 12 34 5456
smi@ubuntu:~$
```

Les variables – spéciales

Le passage des paramètres - Généralisation

Le nombre de paramètres passés en argument à un script n'est pas limité à 9. Toutefois seules les neuf variables 1, ..., 9 permettent de désigner ces paramètres dans le script.

La commande **shift** permet de contourner ce problème. Après shift, le *i*^{ème} paramètre est désigné par **\$i-1**.

Comme nous le pouvons remarquer sur cet exemple :

```
*scr4.sh x
#!/bin/bash
echo liste des paramètres $1 $2 $3
shift; shift; shift
echo "liste des paramètres après shift" $1 $2 $3
```

```
smi@ubuntu: ~
smi@ubuntu:~$ ./scr4.sh 10 20 30 40 50 60 70 80
liste des paramètres 10 20 30
liste des paramètres après shift 40 50 60
smi@ubuntu:~$
```


Les instructions de lecture et d'écriture

Ces instructions permettent de créer des scripts interactifs par l'instauration d'un dialogue sous forme de questions/réponses. La question est posée par l'ordre **echo** et la réponse est obtenue par l'ordre **read** à partir du clavier.

read *variable_1 variable_2... variable_n*

read lit une ligne de texte à partir du clavier, découpe la ligne en mots et attribue aux variables *variable_1* à *variable_n* ces différents mots. S'il y a plus de mots que de variables, la dernière variable se verra affecter le reste de la ligne.

```
scr5.sh x
#!/bin/bash
echo votre nom
read mot1
echo votre prénom
read mot2
echo age votre et votre numéro de tél :
read age num
echo vous êtes $mot1 $mot2 vous avez $age
echo et votre numéro est $num
```

```
smi@ubuntu: ~
smi@ubuntu:~$ ./scr5.sh
votre nom
Fatim ezzahra
votre prénom
Hachimi
age votre et votre numéro de tél :
22 0672829209
vous êtes Fatim ezzahra Hachimi vous avez 22
et votre numéro est 0672829209
smi@ubuntu:~$
```

Opérations et opérateurs numériques

Pour faire des opérations sur des variables, nous pouvons utiliser la commande **expr**.

Cette commande s'utilise avec des variables de type numérique, par exemple :

```
som = expr 3 "+" 5
```

```
pro = expr $a "*" $b
```

le symbole \$ permet d'accéder au contenu de la variable.

Néanmoins la Bash a spécialement un format d'écriture plus clair que cette forme classique (sh). En effet, en mettant l'expression entre double parenthèses, cela permet d'évaluer le résultat d'opérations arithmétiques tel que vous les connaissez :

```
som = $(3+5)
```

```
pro = $(a+b)
```

```
n3=$(17%3) # % : le reste de la division : n3=2
```

```
n1=$((n2*(n1+27)-5)) # n1=127
```

Les structures de contrôle

Le shell possède des structures de contrôle telles qu'il en existe dans les langages de programmation d'usage général :

- ❑ instructions conditionnelles (if.. then.. else, test, case).
- ❑ itérations bornées.
- ❑ itérations non bornées.

Les instructions conditionnelles

Pour la programmation des actions conditionnelles, nous disposons de trois outils :

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

Les instructions conditionnelles

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 commande  
then commandes  
fi
```

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

Exemple :

```
if [ $age -ge 18 ];  
then echo Vous etes majeur(e)  
fi
```

Les instructions conditionnelles

L'instruction if

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

```
if commande  
then commandes1  
else commandes2  
fi
```

Les commandes **commandes1** sont exécutées si la commande_condition **commande** renvoie un code retour nul, sinon ce sont les **commandes2** qui sont exécutées.

Exemple :

```
if [ $age -ge 18 ];  
then echo Vous etes majeur(e)  
else echo Vous etes mineur(e)  
fi
```


Les instructions conditionnelles

L'instruction if

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

```
if commande1
then commandes1
elif commande2
then commandes2
elif commande3
then commandes3
...
else
commandes0
fi
```

Le **elif** (else if) permet d'imbriquer plusieurs **if** les uns dans les autres pour pouvoir traiter tous les cas possibles :

```
if [ $feu = rouge » ];
then echo N'avancez pas
elif [ $feu = "orange" ];
then echo Ralentissez
else echo Allez-y
fi
```

Les instructions conditionnelles

Formuler les conditions

Plusieurs conditions peuvent être liées par les connecteurs logiques. Par exemple pour vérifier que l'âge entré par l'utilisateur est situé entre 0 et 100 :

```
if [ $age -le 0 ] -o [ $age -ge 100 ];  
then echo 1\'age entré n\'est pas correct  
fi
```

Les expressions peuvent être niées par l'opérateur logique de **négation** ! et combinées par les opérateurs **ou logique** -o et **et logique** -a.

Les instructions conditionnelles

Opérateurs de comparaison

Opérateurs sur des fichiers :

Opérateur	Description	Exemple
Opérateurs sur des fichiers		
<code>-e fichier</code>	vrai si <i>fichier</i> existe	<code>[-e /etc/shadow]</code>
<code>-d fichier</code>	vrai si <i>fichier</i> est un répertoire	<code>[-d /tmp/trash]</code>
<code>-f fichier</code>	vrai si <i>fichier</i> est un fichier ordinaire	<code>[-f /tmp/glop]</code>
<code>-L fichier</code>	vrai si <i>fichier</i> est un lien symbolique	<code>[-L /home]</code>
<code>-r fichier</code>	vrai si <i>fichier</i> est lisible (r)	<code>[-r /boot/vmlinuz]</code>
<code>-w fichier</code>	vrai si <i>fichier</i> est modifiable (w)	<code>[-w /var/log]</code>
<code>-x fichier</code>	vrai si <i>fichier</i> est exécutable (x)	<code>[-x /sbin/halt]</code>
<code>fichier1 -nt fichier2</code>	vrai si <i>fichier1</i> plus récent que <i>fichier2</i>	<code>[/tmp/foo -nt /tmp/bar]</code>
<code>fichier1 -ot fichier2</code>	vrai si <i>fichier1</i> plus ancien que <i>fichier2</i>	<code>[/tmp/foo -ot /tmp/bar]</code>

Les instructions conditionnelles

Opérateurs de comparaison

Opérateurs sur les chaînes :

<code>-z chaine</code>	vrai si la <i>chaine</i> est vide	<code>[-z "\$VAR"]</code>
<code>-n chaine</code>	vrai si la <i>chaine</i> est non vide	<code>[-n "\$VAR"]</code>
<code>chaine1 = chaine2</code>	vrai si les deux chaînes sont égales	<code>["\$VAR" = "totoro"]</code>
<code>chaine1 != chaine2</code>	vrai si les deux chaînes sont différentes	<code>["\$VAR" != "tonari"]</code>

Opérateurs de comparaisons numériques :

<code>num1 -eq num2</code>	égalité	<code>[\$nombre -eq 27]</code>
<code>num1 -ne num2</code>	inégalité	<code>[\$nombre -ne 27]</code>
<code>num1 -lt num2</code>	inférieur (<)	<code>[\$nombre -lt 27]</code>
<code>num1 -le num2</code>	inférieur ou égal (<=)	<code>[\$nombre -le 27]</code>
<code>num1 -gt num2</code>	supérieur (>)	<code>[\$nombre -gt 27]</code>
<code>num1 -ge num2</code>	supérieur ou égal (>=)	<code>[\$nombre -ge 27]</code>

Les instructions conditionnelles

Format arithmétique des conditions

Les manipulations arithmétiques sont très incommodes sous cette forme (**sh** classique) de syntaxe.

En **Bash**, l'arithmétique entière est plus facile, grâce à une notation adaptée : le double parenthésage ((...)).

Entre des doubles parenthèses, le bash interprète les caractères < > () * % ... selon leur signification arithmétique usuelle, et le caractère \$ n'est pas nécessaire devant un nom de variable. Le parenthésage y est possible, et sans parenthésage, la priorité des opérateurs arithmétiques est la priorité usuelle. Cette notation offre un cadre cohérent pour l'évaluation arithmétique et le test arithmétique. L'exemple précédant devient plus compréhensible :

```
#!/bin/bash
age=$1
if (((age<0) || (age>100)));
then echo l'age entré n'est pas correct
fi
```

Les instructions conditionnelles

La structure case

La structure **case** permet de faire un traitement différent en fonction de la valeur d'une variable, par exemple :

```
scr6.sh ✕
#!/bin/bash
echo Quel OS vous préférez
echo "1- Win 2- Linux 3- Mac OS 4- Autre"
read choix
case "$choix" in
1) echo "vous préférez Windows" ;;
2) echo "vous préférez Linux" ;;
3) echo "vous préférez Mac OS" ;;
4) echo "Vous préférez un autre système" ;;
*) echo "vous devez taper un choix entre 1 et 4" ;;
esac
```

```
smi@ubuntu:~$ ./scr6.sh
Quel OS vous préférez
1- Win 2- Linux 3- Mac OS 4- Autre
3
vous préférez Mac OS
smi@ubuntu:~$
```


Les instructions conditionnelles

La structure 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 :

```
scr7.sh ✕  
#!/bin/bash  
  
select sys in "Windows" "Linux" "Mac OS" "Autre"  
do  
    echo "vous avez choisie le système" $sys  
    break  
done
```

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

Les itérations

La boucle while - tant que

Les boucles servent à répéter des instructions un certain nombre de fois. Dans le Bash les boucles servent à:

- ❑ Vérifier qu'une information saisie par l'utilisateur est correcte et lui faire recommencer la saisie tant que ce n'est pas correct
- ❑ Recommencer un certain nombre de fois la même suite de commandes

D'autre part elles fonctionnent toujours avec ces trois critères :

- ❑ Une valeur de départ
- ❑ Une condition d'entrée ou de sortie
- ❑ Une incrémentation

Syntaxe

```
While [condition] ; do  
    commandes  
    incrémentation  
done
```

Les itérations

La boucle while - tant que

Exemple :

```
scr8.sh ✕  
#!/bin/bash  
  
i=1 # Valeur de départ 1  
while [ $i -le 5 ]; do # Condition de sortie : i > 5  
    echo tour de boucle n° $i  
    i=`expr $i + 1`  
done
```

```
smi@ubuntu:~$ ./scr8.sh  
tour de boucle n° 1  
tour de boucle n° 2  
tour de boucle n° 3  
tour de boucle n° 4  
tour de boucle n° 5  
smi@ubuntu:~$
```


Les itérations

La boucle until – jusqu'à

until signifie jusqu'à, ce qui veut dire que la boucle sera exécutée jusqu'à ce que la condition soit respectée.

Exemple :

```
*scr9.sh X
#!/bin/bash
continuer=0
until [ $continuer = "n" ] ; do # condition de sortie : n
echo "Voulez-vous recommencer ? o/n"
read continuer # Nouvelle valeur de continuer
done # (qui remplace l'incrément)
```

```
smi@ubuntu:~$ ./scr9.sh
Voulez-vous recommencer ? o/n
o
Voulez-vous recommencer ? o/n
o
Voulez-vous recommencer ? o/n
n
smi@ubuntu:~$ █
```

Les itérations bornées

La boucle for

A priori, la boucle for est utilisée quand on veut exécuter un ensemble de commandes un nombre précis de fois. Trois formes de syntaxe sont possibles :

Forme 1

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

Forme 2

```
for variable
do
commandes
done
```

Forme 3

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

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 **chainen**. 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**.

Les itérations bornées

Exemple forme 1

A priori, la boucle for est utilisée quand on veut exécuter un ensemble de commandes un nombre précis de fois.

Exemple :

```
smi@ubuntu:~$ ./scr10.sh
Combien voulez-vous d'étoiles p
14
*****smi@ubuntu:~$
```

Ainsi ici nous avons utilisé la forme 1 de for, `seq $nombre` permet de créer une séquence successive de nombre.

```
ex.sh ✕
#!/bin/bash
echo Combien voulez-vous d\'étoile?
read nombre
for i in $(seq $nombre)
do
echo -n \*
done
```


Les itérations bornées

Exemple forme 2

La seconde forme de for permet de parcourir la liste d'arguments passés au script :

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

```
smi@ubuntu:~$ ./ex.sh
smi@ubuntu:~$ ./ex.sh az ze er rt
az
ze
er
rt
smi@ubuntu:~$
```

Les itérations bornées

Exemple 3ème forme

En shell la boucle for est beaucoup utilisée pour traiter les fichiers, par exemple :

```
ex.sh ✕
#!/bin/bash
echo "liste des repertoires sous " $(pwd)
echo "=====
for i in *
do
if [ -d $i ]
then
echo $i " :repertoire"
fi
done
echo "=====

smi@ubuntu:~$ ./ex.sh
liste des repertoires sous  /home/smi
=====
Desktop :repertoire
Documents :repertoire
Downloads :repertoire
Music :repertoire
Pictures :repertoire
Public :repertoire
Templates :repertoire
test_cmd :repertoire
Videos :repertoire
=====
smi@ubuntu:~$
```

Les itérations bornées

Exemple 3ème forme

Un autre exemple de la 3ème forme de for, qui permet de renommer tout les fichiers du répertoire depuis lequel le script a été lancé :

```
#!/bin/bash
for fichier_src in *
do
    if [ ! -d $fichier_src ]; then
        fichier_dest=$USER--$fichier_src
        mv $fichier_src $fichier_dest
    fi
done
```

Dangereux ! : exécuter ce script dans un répertoire isolé et NON à partir de la racine ou du répertoire personnel

```
smi@ubuntu:~/test$ chmod u+x *
smi@ubuntu:~/test$ ls -l
total 8
-rwxrw-r-- 1 smi smi  0 Nov 22 20:40 cours.doc
-rwxrw-r-- 1 smi smi  0 Nov 22 20:40 Fichier1.txt
-rwxrw-r-- 1 smi smi  0 Nov 22 20:40 image.jpg
-rwxrw-r-- 1 smi smi  0 Nov 22 20:40 music.mp3
-rwxrw-r-- 1 smi smi 193 Nov 22 20:44 scr11.sh
-rwxrw-r-- 1 smi smi 180 Nov 22 20:41 scr11.sh~
smi@ubuntu:~/test$ ./scr11.sh
Ce script va renommer tous les fichiers en y ajoutan
t votre nom
smi@ubuntu:~/test$ ls -l
total 8
-rwxrw-r-- 1 smi smi  0 Nov 22 20:40 smi--cours.doc
-rwxrw-r-- 1 smi smi  0 Nov 22 20:40 smi--Fichier1.
txt
-rwxrw-r-- 1 smi smi  0 Nov 22 20:40 smi--image.jpg
-rwxrw-r-- 1 smi smi  0 Nov 22 20:40 smi--music.mp3
-rwxrw-r-- 1 smi smi 193 Nov 22 20:44 smi--scr11.sh
-rwxrw-r-- 1 smi smi 180 Nov 22 20:41 smi--scr11.sh~
smi@ubuntu:~/test$
```