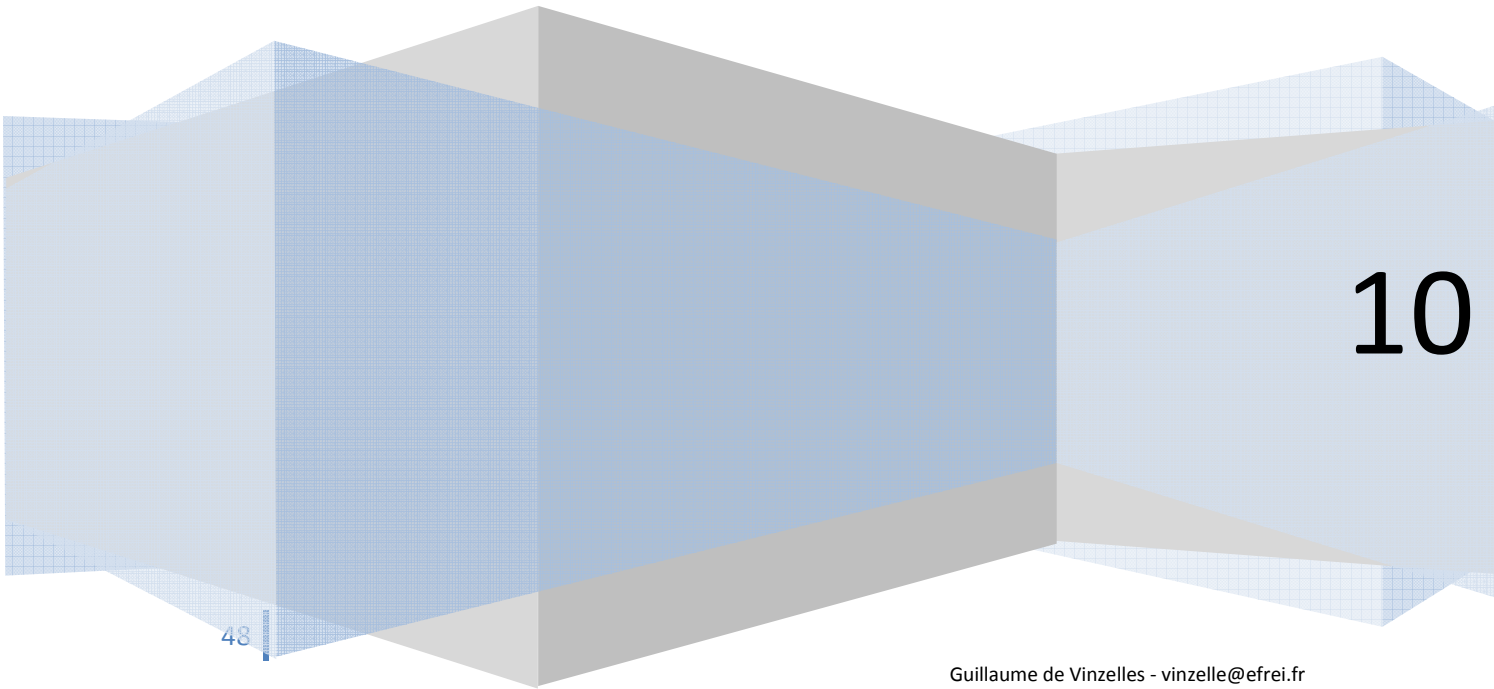
EFREI

TD Unix pour l'utilisateur

Partie 4

Guillaume de Vinzelles - vinzelle@efrei.fr



10

Les réponses aux exercices devront être écrites dans un fichier texte - pas de document Open Office ou de document Word, un simple fichier texte - dont le nom sera composé de :

- La lettre désignant votre groupe
- Les noms des membres de votre binôme
- La date

Voici un exemple de nom de fichier :
`B_edward_alphonse_19052008.txt`
N'hésitez pas à illustrer votre compte rendu avec des copiés/collés venant directement de votre terminal (pas des images, seulement le texte).
N'oubliez pas, pour chaque question, de préciser son numéro.

A la fin du TP, il conviendra de m'envoyer ce fichier par mail à l'adresse *vinzelle@efrei.fr*, en précisant le nom du fichier dans le champ 'Sujet' du mail.

13 La programmation en Shell

13.1 Le premier script

Avant de rentrer dans le détail de l'implémentation des scripts shells, passons en revue quelques méthodes permettant d'exécuter de tels scripts.

13.1.1 Invocation de l'interpréteur

Ecrivons un script extrêmement simple, permettant d'afficher une simple ligne.

```
vinzelle@pommard:~$ echo "echo Ceci est mon premier essai" > essai.sh
vinzelle@pommard:~$ sh essai.sh
Ceci est mon premier essai
```

Nous avons appelé le script en précisant de manière explicite l'interpréteur de commande que nous allons utiliser, ici `sh`.

Le suffixe `.sh` a une valeur purement informative. En effet, le programme fonctionnera aussi bien si on le supprime.

```
vinzelle@pommard:~$ mv essai.sh essai
vinzelle@pommard:~$ sh essai
Ceci est mon premier essai
```

13.1.2 Appel direct

Plutôt que de devoir préciser l'interpréteur à utiliser, supposons que nous souhaitons appeler notre script comme un programme Unix traditionnel.

Voici ce qui se passe si nous appelons le script `essai.sh` directement.

```
vinzelle@pommard:~$ essai.sh
-bash: essai.sh: command not found
```

Nous avons vu dans le chapitre 7.2 que le shell recherchait dans les chemins contenus dans la variable `$PATH` les différentes commandes que l'on pouvait lui demander d'exécuter.

```
vinzelle@pommard:~$ echo $PATH
/users/guest/vinzelle/bin:/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games
```

Le chemin contenant notre script n'est pas inclus dans la variable `$PATH`, ce qui explique que le shell ne trouve pas notre script.

Plusieurs solutions s'offrent alors à nous :

- Ajouter notre script dans un des répertoires présents dans la variable `$PATH`
- Ajouter le . dans notre variable `$PATH`
- Spécifier le chemin qui permet au shell d'exécuter le script

C'est souvent la dernière solution qui est plébiscitée surtout en cours de développement du script.

Malheureusement, voici ce qui se passe si on tente d'exécuter notre script de cette manière.

```
vinzelle@pommard:~$ ./essai.sh
-bash: ./essai.sh: Permission non accordée
```

Il est en effet nécessaire de rendre exécutable le script ! Reportez vous au chapitre 5.4 pour plus de détails.

Voici la méthode à utiliser :

```
vinzelle@pommard:~$ ./essai.sh
-bash: ./essai.sh: Permission non accordée
vinzelle@pommard:~$ chmod u+x essai.sh
vinzelle@pommard:~$ ./essai.sh
Ceci est mon premier essai
```

Nous avons maintenant un script exécutable simplement.

13.1.3 Ligne shebang

Dans la section 14.1.1, nous avons vu qu'il était possible d'exécuter un script en le passant en paramètre d'un interpréteur de commandes.

Dans le cas où nous utilisions la méthode d'appel direct de la section 14.1.2, comment faire pour spécifier l'interpréteur de commandes ?

Il faut pour cela utiliser ce qu'on appelle une ligne *shebang*.

Si l'on veut utiliser le shell *Bash* pour exécuter un script, on le précisera dans cette ligne *shebang* en respectant cette syntaxe :

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

Il faut bien sûr s'assurer que le `bash` se trouve bien dans le répertoire `/bin`.

```
vinzelle@pommard:~$ which bash
/bin/bash
```

Notre script contiendra donc les lignes suivantes :

```
vinzelle@pommard:~$ cat essai.sh
#!/bin/bash
echo Ceci est mon premier essai
vinzelle@pommard:~$ ./essai.sh
Ceci est mon premier essai
```

13.2 Les variables

13.2.1 Généralités

A la différence des langages compilés habituels, une variable n'a pas à être déclarée explicitement, elle commence à exister dès qu'on lui affecte une valeur.
Cette affectation prend la forme `variable=valeur`, sans espaces.
Le contenu de la variable est alors accessible grâce à l'opérateur `$`.
Voici un exemple d'affectation simplissime :

```
vinzelle@pommard:~$ variable=888
vinzelle@pommard:~$ echo $variable
888
```

Attention au cas particulier où l'on souhaite faire figurer un espace dans le contenu de la variable.

```
vinzelle@pommard:~$ variable=abc def
-bash: def: command not found
vinzelle@pommard:~$ variable="abc def"
vinzelle@pommard:~$ echo $variable
abc def
```

13.2.2 Précisions sur l'opérateur $

On consulte le contenu d'une variable grâce à l'opérateur `$`.
Cet opérateur est très versatile et a de nombreux usages.

13.2.2.1 Délimitation du nom de variable

Voici un problème que l'on peut rencontrer lorsque l'on souhaite concaténer le contenu de deux variables.

```
vinzelle@pommard:~$ var1=abc
vinzelle@pommard:~$ var2=abc
vinzelle@pommard:~$ var3=$var1$var2
vinzelle@pommard:~$ echo $var3
abcabc
vinzelle@pommard:~$ var3=$var1.$var2
vinzelle@pommard:~$ echo $var3
abc.abc
vinzelle@pommard:~$ var3=$var1_$var2
vinzelle@pommard:~$ echo $var3
abc
vinzelle@pommard:~$ var3=${var1}_${var2}
vinzelle@pommard:~$ echo $var3
abc_abc
```

L'*underscore* n'est pas un caractère séparateur pour le shell, à la différence du point, ce qui fait que lors de la concaténation :

```
vinzelle@pommard:~$ var3=$var1_$var2
```

le shell ne parvient pas à trouver la variable `$var1_`.
Pour résoudre ce problème il convient de délimiter ses variables avec les caractères { et }.

13.2.2.2 Extraction de sous-chaînes et recherche de motifs

Les shells récents offrent une possibilité d'extraction automatique de sous-chaînes de caractères au sein d'une variable. Cette extraction se réalise grâce à l'expression `${variable:début:longueur}`.
En voici un exemple :

```
vinzelle@pommard:~$ variable=123456789
vinzelle@pommard:~$ echo ${variable:3:2}
```

```
45
```

Il est de plus possible de manipuler le contenu d'une variable grâce à des expressions utilisant le caractère #.
L'expression `${variable#motif}` est remplacée par la valeur de la variable de laquelle on ôte le plus court préfixe qui corresponde au motif.

```
vinzelle@pommard:~$ variable=azertyuiopazertyuiop
vinzelle@pommard:~$ echo ${variable#aze}
rtyuiopazertyuiop
```

Il est également possible d'utiliser les wildcards :

```
vinzelle@pommard:~$ echo ${variable#*z}
ertyuiopazertyuiop
```

De plus les crochets [et] encadrant une liste de caractères représentent n'importe quel caractère contenu dans cette liste. La liste peut contenir un intervalle indiqué par un tiret : `[A-Z]`.
Le caractère ^ ou ! placé en tête de liste indique que la correspondance se fait avec n'importe quel caractère qui n'appartient pas à l'ensemble.

```
vinzelle@pommard:~$ echo ${variable#*[op]}
pazertyuiop
```

L'expression `${variable##motif}` sert à éliminer le plus long préfixe correspondant au motif transmis.
Voici les mêmes exemples illustrant le comportement de cet opérateur.

```
vinzelle@pommard:~$ echo ${variable##aze}
rtyuiopazertyuiop
vinzelle@pommard:~$ echo ${variable##*z}
ertyuiop
vinzelle@pommard:~$ echo ${variable##*[op]}
```

Symétriquement, les expressions `${variable%motif}` et `${variable%%motif}` correspondent au contenu de la variable indiquée, qui est débarrassé respectivement, du plus court et du plus long suffixe correspond au motif transmis.

```
vinzelle@pommard:~$ echo ${variable%aze*}
azertyuiop
vinzelle@pommard:~$ echo ${variable%%t*}
azer
```

Il existe enfin un opérateur qui permet de remplacer la première occurrence du motif par une chaîne fournie, grâce aux expressions :

- `${variable/motif/remplacement}` qui remplace la première occurrence du motif
- `${variable//motif/remplacement}` qui remplace toutes les occurrences du motif

```
vinzelle@pommard:~$ echo ${variable/aze/wxc}
wxcrtyuiopazertyuiop
vinzelle@pommard:~$ echo ${variable//aze/wxc}
wxcrtyuiopwxcrtyuiop
```

13.2.2.3 Longueur de chaîne

L'opérateur `$` offre aussi la possibilité de calculer la longueur d'une chaine de caractères grâce à sa forme `${#variable}`.

```
vinzelle@pommard:~$ echo ${#variable}
20
```

13.2.2.4 Code de retour

La variable `$?` permet de récupérer le code de retour de la dernière commande exécutée, ce qui est très pratique pour s'assurer qu'une commande s'est correctement déroulée.

```
vinzelle@pommard:~$ cd plop
-bash: cd: plop: Aucun fichier ou répertoire de ce type
vinzelle@pommard:~$ echo $?
1
vinzelle@pommard:~$ cd Unix/
vinzelle@pommard:~/Unix$ echo $?
0
```

Le plus souvent le code retour 0 correspond à un déroulement normal de la commande, alors qu'un code de retour >0 signifie une erreur. Les codes retours sont en général décrits dans les *manpages*.

13.2.2.5 PID du shell

La variable $$ permet de récupérer le PID du shell.

```
vinzelle@pommard:~$ ps | grep $(echo $$) | grep -v grep
12687 pts/0    00:00:00 bash
```

13.2.2.6 Invocation de commande

L'opérateur $ permet également de placer automatiquement le contenu de la sortie standard d'une commande dans une variable, grâce à cette syntaxe : `variable=$(ls -l)`.
Cette syntaxe est équivalente à celle utilisant des *backquotes* : `variable=`ls -l``. Cette dernière syntaxe est néanmoins déconseillée car moins lisible et plus difficile à imbriquer.

```
vinzelle@pommard:~$ machine=$(uname -a)
vinzelle@pommard:~$ echo $machine
Linux pommard 2.6.24.2 #1 SMP Mon Feb 11 16:20:10 CET 2008 i686 GNU/Linux
```

13.3 Le passage de paramètres

13.3.1 Paramètres positionnels

Les paramètres positionnels sont utilisés pour accéder aux informations qui sont fournis sur la ligne de commande lors de l'invocation d'un script, mais aussi aux arguments transmis à une fonction. Les arguments sont placés dans des paramètres qui peuvent être consultés avec la syntaxe `$1`, `$2`, `$3`, etc.…
Pour consulter le contenu d'un paramètre qui contient plus d'un chiffre, il faut l'encadrer par des accolades : `${10}`.
Par convention, l'argument numéro 0 contient toujours le nom du script tel qu'il a été invoqué.
Reprenons notre script du précédent chapitre.

```
vinzelle@pommard:~$ cat essai.sh
#!/bin/bash
echo Ceci est mon premier essai
echo Nom du script : $0
echo Contenu du 8° argument : $8
vinzelle@pommard:~$ ./essai.sh a b c d e f g h i j k l m
Ceci est mon premier essai
Nom du script : ./essai.sh
Contenu du 8° argument : h
```

A noter qu'il est possible de décaler l'ensemble des paramètres à l'aide de l'instruction `shift`.

```
vinzelle@pommard:~$ cat essai.sh
#!/bin/bash
echo Ceci est mon premier essai
echo Nom du script : $0
echo Contenu du 8° argument : $8
shift
echo Contenu du 9° argument : $8
vinzelle@pommard:~$ ./essai.sh a b c d e f g h i j k l m
Ceci est mon premier essai
Nom du script : ./essai.sh
Contenu du 8° argument : h
Contenu du 9° argument : i
```

13.3.2 Paramètres spéciaux

Nos exemples d'affichage d'arguments comportent une lacune : nous ne pouvons déterminer dynamiquement le nombre de paramètres passés à notre script.
Le paramètre spécial `$#` nous renseigne sur cette valeur.

```
vinzelle@pommard:~$ cat essai.sh
#!/bin/bash
echo Ceci est mon premier essai
echo $# paramètres ont été passés à ce script
echo Nom du script : $0
echo Contenu du 8° argument : $8
shift
echo Contenu du 9° argument : $8
vinzelle@pommard:~$ ./essai.sh a b c d e f g h i j k l
Ceci est mon premier essai
12 paramètres ont été passés à ce script
Nom du script : ./essai.sh
```

```
Contenu du 8° argument : h
Contenu du 9° argument : i
```

Il est souvent utile de pouvoir manipuler en une seule fois l'ensemble des paramètres positionnels, afin de les transmettre à une fonction ou à un autre script, par exemple.
Le paramètre spéciale `$@` donne cette possibilité.

```
vinzelle@pommard:~$ cat essai.sh
#!/bin/bash
echo Ceci est mon premier essai
echo $# paramètres ont été passés à ce script
echo Nom du script : $0
echo Contenu du 8° argument : $8
shift
echo Contenu du 9° argument : $8
echo Ensemble des paramètres passés en argument : $@
vinzelle@pommard:~$ ./essai.sh a b c d e f g h i j k l
Ceci est mon premier essai
12 paramètres ont été passés à ce script
Nom du script : ./essai.sh
Contenu du 8° argument : h
Contenu du 9° argument : i
Ensemble des paramètres passés en argument : b c d e f g h i j k l
```

13.4 Les instructions de lecture et d'écriture

Les instructions `read` et `echo` permettent de créer des fichiers de commandes interactifs sous forme de questions/réponses.

```
vinzelle@pommard:~$ cat interactif.sh
#!/bin/bash
echo "Quel est votre nom ?"
read name
echo Bonjour $name
vinzelle@pommard:~$ ./interactif.sh
Quel est votre nom ?
Guillaume
Bonjour Guillaume
```

13.5 Exécutions séquentielles et parallèles

Voici un tableau récapitulant les différents opérateurs permettant d'organiser entres elles plusieurs opérations :

Symbole	Connexion	Détail
;	Séquencement	La seconde opération ne commence qu'après la fin de la première.
&	Parallélisme	La première opération est lancée à l'arrière plan et la seconde démarre simultanément ; elles n'ont pas d'interactions entre elles.
&&	Dépendance	La seconde opération n'est exécutée que si la première a renvoyé un code de retour 0, ce qui par convention signifie « succès ».
||	Alternative	La seconde opération n'est exécutée que si la première a renvoyé un code de retour non nul, ce qui par convention signifie « échec ».

13.6 Les structures de contrôle

13.6.1 Sélection d'instructions

13.6.1.1 Construction if-then-else

Voici la syntaxe complete d'une construction `if-then-else` :

```
if condition_1
then
   commande_1
elif condition_2
then
   commande_2
else
   commande_n
fi
```

A noter qu'il est également courant de rencontrer cette syntaxe :

```
if condition ; then
   commande
fi
```

13.6.1.2 Conditions et tests

La condition dans une structure de test est représentée par une fonction dont le code de retour correspond à une valeur vraie (0) ou fausse (retour > 0).
Pour effectuer la plupart des comparaisons dont nous avons besoin dans nos scripts, le shell nous offre un opérateur nommé `test`.
Cet opérateur peut être remplacé de manière transparente par la commande interne [ ].

Option	Condition
(EXPRESSION)	EXPRESSION is true
! EXPRESSION	EXPRESSION is false
EXPRESSION1 -a EXPRESSION2	both EXPRESSION1 and EXPRESSION2 are true
EXPRESSION1 -o EXPRESSION2	either EXPRESSION1 or EXPRESSION2 is true
-n STRING	the length of STRING is nonzero
STRING	equivalent to -n STRING
-z STRING	the length of STRING is zero
STRING1 = STRING2	the strings are equal
STRING1 != STRING2	the strings are not equal
INTEGER1 -eq INTEGER2	INTEGER1 is equal to INTEGER2
INTEGER1 -ge INTEGER2	INTEGER1 is greater than or equal to INTEGER2
INTEGER1 -gt INTEGER2	INTEGER1 is greater than INTEGER2
INTEGER1 -le INTEGER2	INTEGER1 is less than or equal to INTEGER2
INTEGER1 -lt INTEGER2	INTEGER1 is less than INTEGER2
INTEGER1 -ne INTEGER2	INTEGER1 is not equal to INTEGER2
FILE1 -ef FILE2	FILE1 and FILE2 have the same device and inode numbers
FILE1 -nt FILE2	FILE1 is newer (modification date) than FILE2
FILE1 -ot FILE2	FILE1 is older than FILE2
-b FILE	FILE exists and is block special
-c FILE	FILE exists and is character special
-d FILE	FILE exists and is a directory
-e FILE	FILE exists
-f FILE	FILE exists and is a regular file
-g FILE	FILE exists and is set-group-ID
-G FILE	FILE exists and is owned by the effective group ID
-h FILE	FILE exists and is a symbolic link (same as -L)
-k FILE	FILE exists and has its sticky bit set
-L FILE	FILE exists and is a symbolic link (same as -h)
-O FILE	FILE exists and is owned by the effective user ID
-p FILE	FILE exists and is a named pipe
-r FILE	FILE exists and read permission is granted
-s FILE	FILE exists and has a size greater than zero
-S FILE	FILE exists and is a socket
-t FD	file descriptor FD is opened on a terminal
-u FILE	FILE exists and its set-user-ID bit is set
-w FILE	FILE exists and write permission is granted
-x FILE	FILE exists and execute (or search) permission is granted

Voici quelques exemples d'utilisation des tests.

```
vinzelle@pommard:~$ variable="itsalive"
vinzelle@pommard:~$ if [ variable ]; then echo $variable; fi
itsalive
vinzelle@pommard:~$ unset variable
```

```
vinzelle@pommard:~$ if [ variable ]; then echo $variable; fi

vinzelle@pommard:~$ touch test
vinzelle@pommard:~$ if [ -r test ]; then echo "Le fichier test est lisible"; fi
Le fichier test est lisible
vinzelle@pommard:~$ if [ -s test ]; then echo "Le fichier test contient des données"; else
echo "Le fichier test est vide"; fi
Le fichier test est vide
vinzelle@pommard:~$ echo "plop" > test
vinzelle@pommard:~$ if [ -s test ]; then echo "Le fichier test contient des données"; else
echo "Le fichier test est vide"; fi
Le fichier test contient des données
```

13.6.1.3 Construction case-esac

Cette construction simplifie énormément les structures conditionnelles

```
case expression in
   motif_1) commande_1 ;;
   motif_2) commande_2 ;;
   …
esac
```

Voici un exemple de script utilisant la structure `case` permettant de déterminer le type de noyau installé sur la machine, pour par exemple automatiser des installations conditionnelles.

```
#!/bin/bash
i=$(uname -r)
i=${1:-$i}
case "$i" in
        2.6.* | 2.7.* ) type="2.6" ;;
        2.4.* | 2.5.* ) type="2.4" ;;
        2.2.* | 2.3.* ) type="2.2" ;;
        *) type="inconnu" ;;
esac
echo Votre noyau est de type ${type}
```

Voici le résultat de quelques exécutions de ce script sur une des machines de l'Efrei :

```
vinzelle@pommard:~$ bash case.sh
Votre noyau est de type 2.6
vinzelle@pommard:~$ bash case.sh 2.2.2
Votre noyau est de type 2.2
vinzelle@pommard:~$ bash case.sh 2.4.2
Votre noyau est de type 2.4
vinzelle@pommard:~$ bash case.sh 2.6.2
Votre noyau est de type 2.6
vinzelle@pommard:~$ bash case.sh 2.8
Votre noyau est de type inconnu
vinzelle@pommard:~$ bash case.sh 2.8.3
Votre noyau est de type inconnu
```

13.6.2 Itérations d'instructions

13.6.2.1 Répétitions while-do et until-do

Les deux structures `while-do-done` et `until-do-done` servent à répéter une séquence d'instructions jusqu'à ce qu'une condition soit vérifiée. Elles se présentent ainsi :

```
while condition
do
   commandes
done
```

et

```
until condition
do
   commandes
done
```

La première construction répète les commandes tant que la condition est vraie, alors que la deuxième les répète jusqu'à ce que la condition soit vraie.

En voici un exemple :

```
vinzelle@pommard:~$ cat compteur.sh
#!/bin/bash

i=1
```

```
while [ $i -le 10 ]; do
        echo $i
        i=$((i+1))
done
vinzelle@pommard:~$ bash compteur.sh
1
2
3
4
5
6
7
8
9
10
```

13.6.2.2 Ruptures de séquence avec break et continue

Dans certains cas, les instructions dans le corps de la boucle influent sur la condition utilisée pour limiter les itérations. Les ruptures de séquences sont introduites par les mots clés `break` et `continue`. Le mot clé `break` sort immédiatement de la boucle en cours, alors que le mot clé `continue` permet de renvoyer le contrôle au début de la boucle.

Voici un exemple de l'utilisation des mots clé `break et continue`.

```
vinzelle@pommard:~$ cat break.sh
#!/bin/bash

while true; do
        echo "Devinez le chiffre auquel je pense"
        read reponse
        if [ -z $reponse ] ; then
                echo "Saisie invalide"
                continue
        fi
        if [ $reponse = "8" ] ; then
                echo "Bonne réponse !"
                break
        else
                echo "Allons faites un effort"
        fi
done
vinzelle@pommard:~$ bash break.sh
Devinez le chiffre auquel je pense
0
Allons faites un effort
Devinez le chiffre auquel je pense

Saisie invalide
Devinez le chiffre auquel je pense
8
Bonne réponse !
```

13.6.2.3 Construction for-do

La boucle `for` disponible avec le shell est très différente de la boucle `for` classique disponible dans de nombreux langages de programmation. La boucle `for-do` se rapproche plus d'une boucle `foreach` disponible en PHP, Python, Perl ou C#.

Voici la structure d'une boucle `for-do` :

```
for variable in liste_de_mots
do
   commandes
done
```

La variable va prendre successivement comme valeur tous les mots de la liste suivant `in` et le corps de la boucle sera répété pour chacune de ces valeurs. Voyons un exemple, qui calcule des carrés.

```
vinzelle@pommard:~$ cat carres.sh
#!/bin/bash

for i in 1 2 3 5 7 11 13 ; do
        echo "$i² = $((i*i))"
done
vinzelle@pommard:~$ bash carres.sh
1² = 1
```

```
2² = 4
3² = 9
5² = 25
7² = 49
11² = 121
13² = 169
```

13.7 Fonctions

De manière générale une fonction se présente ainsi :

```
function nom_de_la_fonction
{
   commandes
}
```

Les arguments transmis à la fonction sont placés dans des paramètres positionnels, à l'image de ce qui est fait pour un script.

Il est possible de déclarer une variable qui ne soit que locale à la fonction à l'aide du mot clé `local`.

```
vinzelle@choam:~/scripts$ cat fonction.sh
#!/bin/bash

function fct
{
        globale=$1
        local locale=$2
        echo Le premier paramètre transmis est $globale
        echo Le deuxième paramètre transmis est $locale
}

fct 1 2
echo Après appel de la fonction, le premier paramètre est : $globale
echo Le deuxième est : $locale
vinzelle@choam:~/scripts$ bash fonction.sh
Le premier paramètre transmis est 1
Le deuxième paramètre transmis est 2
Après appel de la fonction, le premier paramètre est : 1
Le deuxième est :
```

13.8 Exercices

13.8.1 Exercice

Ecrire un script shell qui décrit sur sa sortie standard les messages suivants :

Mon nom est xxx

Je suis appelé avec yyy arguments

qui sont 111 222 333 444

(*xxx* sera remplacé par le nom sous lequel ce script aura été invoqué, *yyy* par le nombre d'arguments et *111*, *222*, etc. par les arguments en question). Quand ce script fonctionnera correctement, invoquez-le avec les cinq arguments : *Bienvenue dans le monde Unix*, puis avec un seul argument contenant la chaîne de caractères : « *Bienvenue dans le monde Unix* ».

13.8.2 Exercice

Ecrire deux scripts démontrant que le shell fils hérite de son père, mais que le père n'hérite pas de son fils. Il faut pour cela écrire un script père, appelons le `Vador.sh`, qui appelle un script fils, appelons le `Luke.sh`. `Vador.sh` affiche quelques variables système, et `Luke.sh` les modifie. `Vador.sh` affiche alors à nouveau ces variables.

13.8.3 Exercice

Ecrivez un script qui affiche le nom de chaque fichier du répertoire dans lequel il se trouve, en le faisant précéder d'un numéro d'ordre, comme dans l'exemple suivant :

```
vinzelle@pommard:~/scripts$ bash 14.8.3.sh
1) 14.8.3.sh
2) break.sh
3) carres.sh
4) case.sh
5) compteur.sh
6) fonction.sh
```

13.8.4 Exercice

Ecrivez un script qui affiche à cinq reprises le contenu de sa ligne de commande, en dormant une seconde entre chaque affichage. Pour endormir un script, on invoque la commande `sleep`.

Invoquez alors trois exemplaires de ce script en parallèle (sur la même ligne de commande), chacun avec un argument différent pour vérifier que les exécutions sont bien simultanées.

13.8.5 Exercice

En utilisant exclusivement les commandes `cd` et `echo`, écrire le schell script `recurls` réalisant la même fonction que la commande `ls -R`. La commande `recurls rep1` devra lister les noms de tous les fichiers et répertoires situés sous le répertoire `rep1`, y compris les sous répertoires et les fichiers qu'ils contiennent. Pensez récursivité pour ce script.

13.8.6 Exercice

Ecrivez le script `rename` permettant de renommer un ensemble de fichiers. Par exemple `rename '.c' '.bak'` aura pour effet de renommer tous les fichiers d'extension `.c` en `.bak`. Utilisez la commande `basename`.

13.8.7 Exercice

Dans le script case.sh du paragraphe 14.6.1.3, à quoi sert la ligne `i=${1:-$i}` ?

13.8.8 Exercice

Ecrivez un script, nommé `killprog`, qui permet d'envoyer le signal SIGKILL à un processus désigné non pas par son PID mais par son nom.

Il faudra prendre garde au fait que plusieurs processus différents peuvent porter le même nom, en présenter clairement la liste, et toujours demander confirmation avant d'envoyer le signal.