Nous allons créer un nouveau script que nous appellerons `variables.sh` :

console
```
$ vim variables.sh
```

La première ligne de tous nos scripts doit indiquer quel shell est utilisé, comme nous l'avons appris plus tôt. Commencez donc par écrire :

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

Cela indique que nous allons programmer en bash.

Maintenant, définissons une variable. Toute variable possède un nom et une valeur :

console
```
message='Bonjour tout le monde'
```

Dans le cas présent :

- la variable a pour **nom** `message` ;
- … et pour **valeur** `Bonjour tout le monde`.

Ne mettez pas d'espaces autour du symbole égal « = » ! Le bash est très pointilleux sur de nombreux points, évitez par conséquent de le vexer.
Je vous signalerai systématiquement les pièges à éviter, car il y en a un certain nombre !

Si vous voulez insérer une apostrophe dans la valeur de la variable, il faut la faire précéder d'un antislash \. En effet, comme les apostrophes servent à délimiter le contenu, on est obligé d'utiliser un **caractère d'échappement** (c'est comme ça que cela s'appelle) pour pouvoir véritablement insérer une apostrophe :

console
```
message='Bonjour c\'est moi'
```

Bien, reprenons notre script. Il devrait à présent ressembler à ceci :

console
```
#!/bin/bash

message='Bonjour tout le monde'
```

Exécutez-le pour voir ce qui se passe (après avoir modifié les droits pour le rendre exécutable, bien sûr) :

console
```
$ ./variables.sh
$
```

Il ne se passe rien !

Que fait le script, alors ?

Il met en mémoire le message `Bonjour tout le monde`, et c'est tout ! Rien ne s'affiche à l'écran !

Pour afficher une variable, il va falloir utiliser une commande dont je ne vous ai pas encore parlé…

Avant de commencer à parler de **variables**, il y a une commande que j'aimerais vous présenter : `echo`. J'aurais pu en parler avant que l'on commence à faire des scripts bash, mais vous n'en auriez pas vu l'utilité avant d'aborder ce chapitre.

Son principe est très simple : elle affiche dans la console le message demandé. Un exemple :

console

```
$ echo Salut tout le monde
Salut tout le monde
```

Comme vous le voyez, c'est simple comme bonjour. Les guillemets ne sont pas requis.

Mais… comment est-ce que cela fonctionne ?

En fait, la commande `echo` affiche dans la console tous les paramètres qu'elle reçoit. Ici, nous avons envoyé quatre paramètres :

- `Salut` ;
- `tout` ;
- `le` ;
- `monde`.

Chacun des mots était considéré comme un paramètre que `echo` a affiché. Si vous mettez des guillemets autour de votre message, celui-ci sera considéré comme étant un seul et même paramètre (le résultat sera visuellement le même) :

console

```
$ echo "Salut tout le monde"
Salut tout le monde
```

Si vous voulez insérer des retours à la ligne, il faudra activer le paramètre `-e` et utiliser le symbole :

console

```
$ echo -e "Message\nAutre ligne"
Message
Autre ligne
```

Afficher une variable

Pour afficher une variable, nous allons de nouveau utiliser son nom précédé du symbole dollar `$` :

console

```
#!/bin/bash

message='Bonjour tout le monde'
echo $message
```

Comparez les lignes 3 et 4 : lorsque l'on **déclare** la variable à la ligne 3, on ne doit pas mettre de `$` devant. En revanche, lorsqu'on l'**affiche** à la ligne 4, on doit cette fois mettre un `$` !

Résultat :

console

```
Bonjour tout le monde
```

Maintenant, supposons que l'on veuille afficher à la fois du texte et la variable. Nous serions tentés d'écrire :

console

```
#!/bin/bash

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

Le problème est que cela ne fonctionne pas comme on le souhaite car cela affiche :

console

```
Le message est : $message
```

Pour bien comprendre ce qui se passe, intéressons-nous au fonctionnement de ce que l'on appelle les « quotes ».

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.

Les simples quotes ' '

Commençons par les simples quotes :

console

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

console

```
Le message est : $message
```

Avec de simples quotes, la variable n'est pas analysée et le $ est affiché tel quel.

Les doubles quotes " "

Avec des doubles quotes :

console

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

console

```
Le message est : Bonjour tout le monde
```

… ça fonctionne ! Cette fois, la variable est analysée et son contenu affiché.

En fait, les doubles quotes demandent à bash d'analyser le contenu du message. S'il trouve des symboles spéciaux (comme des variables), il les interprète.
Avec de simples quotes, le contenu était affiché tel quel.

Les back quotes ` `

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

Un exemple valant mieux qu'un long discours, **regardez la première ligne** :

console

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

console

```
Vous êtes dans le dossier /home/mateo21/bin
```

La commande `pwd` a été exécutée et son contenu inséré dans la variable `message` ! Nous avons ensuite affiché le contenu de la variable.

Cela peut paraître un peu tordu, mais c'est réellement utile. Nous nous en resservirons dans les chapitres suivants.

Vous pouvez demander à l'utilisateur de saisir du texte avec la commande `read`. Ce texte sera immédiatement stocké dans une variable.

La commande `read` propose plusieurs options intéressantes. La façon la plus simple de l'utiliser est d'indiquer le nom de la variable dans laquelle le message saisi sera stocké :

console
```
read nomvariable
```

Adaptons notre script pour qu'il nous demande notre nom puis qu'il nous l'affiche :

console
```
#!/bin/bash

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

Lorsque vous lancez ce script, rien ne s'affiche, mais vous pouvez taper du texte (votre nom, par exemple) :

console
```
Mathieu
Bonjour Mathieu !
```

Notez que la première ligne correspond au texte que j'ai tapé au clavier.

Affecter simultanément une valeur à plusieurs variables

On peut demander de saisir autant de variables d'affilée que l'on souhaite. Voici un exemple de ce qu'il est possible de faire :

console
```
#!/bin/bash

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

console
```
Deschamps Mathieu
Bonjour Deschamps Mathieu !
```

`read` lit ce que vous tapez mot par mot (en considérant que les mots sont séparés par des espaces). Il assigne chaque mot à une variable différente, d'où le fait que le nom et le prénom ont été correctement et respectivement assignés à `$nom` et `$prenom`.
Si vous rentrez plus de mots au clavier que vous n'avez prévu de variables pour en stocker, la dernière variable de la liste récupèrera tous les mots restants. En clair, si j'avais tapé pour le programme précédent « Nebra Mathieu Cyril », la variable `$prenom` aurait eu pour valeur « Mathieu Cyril ».

–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 :

console
```
#!/bin/bash

read -p 'Entrez votre nom : ' nom
echo "Bonjour $nom !"
```

Notez que le message `'Entrez votre nom'` a été entouré de quotes. Si on ne l'avait pas fait, le bash aurait considéré que chaque mot était un paramètre différent !

Résultat :

console
```
Entrez votre nom : Mathieu
Bonjour Mathieu !
```

C'est mieux !

`-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 :

console
```
#!/bin/bash

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

console
```
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 , comme vous avez appris à le faire plus tôt :

console
```
#!/bin/bash

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

console
```
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.

console
```
#!/bin/bash

read -p 'Entrez le code de désamorçage de la bombe (vous avez 5 secondes) : '  -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 :

console
```
#!/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 passe est $pass ! :)"
```

console

```
Entrez votre mot de passe :
Merci ! Je vais dire à tout le monde que votre mot de passe est supertopsecret38 ! :)
```

Comme vous pouvez le constater, le mot de passe que j'ai entré ne s'affiche pas lors de l'instruction `read`.

En bash, les variables sont toutes des chaînes de caractères. En soi, le bash n'est pas vraiment capable de manipuler des nombres ; il n'est donc pas capable d'effectuer des opérations.

Heureusement, il est possible de passer par des commandes (eh oui, encore). Ici, la commande à connaître est `let`.

console

```
let "a = 5"
let "b = 2"
let "c = a + b"
```

À la fin de ce script, la variable `$c` vaudra 7. Testons :

console

```
#!/bin/bash

let "a = 5"
let "b = 2"
let "c = a + b"
echo $c
```

console

```
7
```

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 :

console

```
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
```

Une petite explication pour les deux dernières lignes :

- 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.

Notez qu'il est possible aussi de contracter les commandes, comme cela se fait en langage C.
Ainsi :

console

```
let "a = a * 3"
```

… équivaut à écrire :

```
let "a *= 3"
```

Actuellement, les résultats renvoyés sont des nombres entiers et non des nombres décimaux. Si vous voulez travailler avec des nombres décimaux, renseignez-vous sur le fonctionnement de la commande `bc`.

Le type de condition le plus courant est le `if`, qui signifie « si ».

Si

Les conditions ont la forme suivante :

console

```
SI test_de_variable
ALORS
------> effectuer_une_action
FIN SI
```

Bien entendu, ce n'est pas du bash. Il s'agit juste d'un schéma pour vous montrer quelle est la forme d'une condition.

La syntaxe en bash est la suivante :

console

```
if [ test ]
then
        echo "C'est vrai"
fi
```

Le mot `fi` (`if` à l'envers !) à la fin indique que le `if` s'arrête là. Tout ce qui est entre le `then` et le `fi` sera exécuté uniquement si le test est vérifié.

Vous noterez — c'est très important — qu'il y a des espaces à l'intérieur des crochets. On ne doit pas écrire `[test]` mais `[ test ]` !

Il existe une autre façon d'écrire le `if` : en plaçant le `then` sur la même ligne. Dans ce cas, il ne faut pas oublier de rajouter un point-virgule après les crochets :

console

```
if [ test ]; then
        echo "C'est vrai"
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, puis nous apprendrons à faire des tests plus compliqués un peu plus loin dans le chapitre.

Faisons quelques tests sur un script que nous appellerons `conditions.sh` :

console

```
#!/bin/bash

nom="Bruno"

if [ $nom = "Bruno" ]
then
        echo "Salut Bruno !"
fi
```

Comme `$nom` est bien égal à « Bruno », ce script affichera :

```
Salut Bruno !
```

Essayez de changer le test : si vous n'écrivez pas précisément « Bruno », le `if` ne sera pas exécuté et votre script n'affichera donc rien.

Notez aussi que vous pouvez tester deux variables à la fois dans le `if` :

console

```
#!/bin/bash

nom1="Bruno"
nom2="Marcel"

if [ $nom1 = $nom2 ]
then
        echo "Salut les jumeaux !"
fi
```

Comme ici `$nom1` est différent de `$nom2`, le contenu du `if` ne sera pas exécuté. Le script n'affichera donc rien.

Sinon

Si vous souhaitez faire quelque chose de particulier quand la condition n'est **pas** remplie, vous pouvez rajouter un `else` qui signifie « sinon ».

En français, cela s'écrirait comme ceci :

console

```
SI test_de_variable
ALORS
------> effectuer_une_action
SINON
------> effectuer_une_action
FIN SI
```

console

```
if [ test ]
then
        echo "C'est vrai"
else
        echo "C'est faux"
fi
```

Reprenons notre script de tout à l'heure et ajoutons-lui un `else` :

console

```
#!/bin/bash

nom="Bruno"

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

Bon : comme la variable vaut toujours la même chose, le `else` ne sera jamais exécuté, ce n'est pas rigolo. Je vous propose plutôt de vous baser sur le premier paramètre (`$1`) envoyé au script :

```
#!/bin/bash

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

Testez maintenant votre script en lui donnant un paramètre :

console

```
$ ./conditions.sh Bruno
Salut Bruno !
```

Et si vous mettez autre chose :

console

```
$ ./conditions.sh Jean
J'te connais pas, ouste !
```

Notez que le script plante si vous oubliez de l'appeler avec un paramètre. Pour bien faire, il faudrait d'abord vérifier dans un `if` s'il y a au moins un paramètre. Nous apprendrons à faire cela plus loin.

Sinon si

Il existe aussi le mot clé `elif`, abréviation de « else if », qui signifie « sinon si ». Sa forme ressemble à ceci :

console

```
SI test_de_variable
ALORS
------> effectuer_une_action
SINON SI autre_test
ALORS
------> effectuer_une_action
SINON SI encore_un_autre_test
ALORS
------> effectuer_une_action
SINON
------> effectuer_une_action
FIN SI
```

C'est un peu plus compliqué, n’est-ce pas ?

Sachez que l'on peut mettre autant de « sinon si » que l'on veut ; là, j'en ai mis deux. En revanche, on ne peut mettre qu'un seul « sinon », qui sera exécuté à la fin si aucune des conditions précédentes n'est vérifiée.

Bash va d'abord analyser le premier test. S'il est vérifié, il effectuera la première action indiquée ; s'il ne l'est pas, il ira au premier « sinon si », au second, etc., jusqu'à trouver une condition qui soit vérifiée. Si aucune condition ne l’est, c'est le « sinon » qui sera lu.

Bien ! Voyons comment cela s'écrit en bash :

console

```
if [ test ]
then
        echo "Le premier test a été vérifié"
elif [ autre_test ]
```

```
then
        echo "Le second test a été vérifié"
elif [ encore_autre_test ]
then
        echo "Le troisième test a été vérifié"
else
        echo "Aucun des tests précédents n'a été vérifié"
fi
```

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

console

```
#!/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
```

Vous pouvez tester ce script ; encore une fois, n'oubliez pas d'envoyer un paramètre sinon il plantera, ce qui est normal.

Voyons maintenant un peu quels sont les tests que nous pouvons faire. Pour l'instant, on a juste vérifié si deux chaînes de caractères étaient identiques, mais on peut faire beaucoup plus de choses que cela !

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.

Nous allons maintenant découvrir tous ces types de tests et les essayer. 

Tests sur des chaînes de caractères

Comme vous devez désormais le savoir, 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 :

console

```
#!/bin/bash

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

console

```
$ ./conditions.sh Bruno Bernard
Les 2 paramètres sont différents !
```

console

```
$ ./conditions.sh Bruno Bruno
Les 2 paramètres sont identiques !
```

Condition	Signification
`$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.

On peut aussi vérifier si le paramètre existe avec `-z` (vérifie si la chaîne est vide). En effet, si une variable n'est pas définie, elle est considérée comme vide par bash. On peut donc par exemple s'assurer que `$1` existe en

faisant comme suit :

console

```
#!/bin/bash

if [ -z $1 ]
then
        echo "Pas de paramètre"
else
        echo "Paramètre présent"
fi
```

console

```
$ ./conditions.sh
Pas de paramètre
```

console

```
$ ./conditions.sh param
Paramètre présent
```

Tests sur des nombres

Bien que bash gère les variables comme des chaînes de caractères pour son fonctionnement interne, rien ne nous empêche de faire des comparaisons de nombres si ces variables en contiennent. Vous trouverez les différents types de tests disponibles sur le tableau suivant.

Condition	Signification
`$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 $num2`	Vérifie si `num1` est inférieur ou égal (<=) à `num2` (**l***owero***r e***qual*).
`$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***reatero***r e***qual*).

Vérifions par exemple si un nombre est supérieur ou égal à un autre nombre :

console

```
#!/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
```

console

```
$ ./conditions.sh 11
Vous avez envoyé moins de 20
```

Tests sur des fichiers

Un des avantages de bash sur d'autres langages est que l'on peut très facilement faire des tests sur des fichiers : savoir s'ils existent, si on peut écrire dedans, s'ils sont plus vieux, plus récents, etc. Le tableau suivant présente les différents types de tests disponibles.

Condition	Signification
`-e $nomfichier`	Vérifie si le fichier existe.
`-d $nomfichier`	Vérifie si le fichier est un répertoire. N'oubliez pas que 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 :

console

```
#!/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 !
```

console

```
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.

Les deux symboles à connaître sont :

- **&&** : signifie « et » ;
- || : signifie « ou ».

Il faut encadrer chaque condition par des crochets. Prenons un exemple :

console

```
#!/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.

console

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

Notez que les tests sont effectués l'un après l'autre et seulement s'ils sont nécessaires. Bash vérifie d'abord s'il y a au moins un paramètre. Si ce n'est pas le cas, il ne fera pas le second test puisque la condition ne sera de toute façon pas vérifiée.

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.

On a vu tout à l'heure un `if` un peu complexe qui faisait appel à des `elif` et à un `else` :

console

```
#!/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
```

Ce genre de « gros `if` qui teste toujours la même variable » ne pose pas de problème mais n'est pas forcément très facile à lire pour le programmeur. À la place, il est possible d'utiliser l'instruction `case` si nous voulons.

Le rôle de `case` est de tester la valeur d'une même variable, mais de manière plus concise et lisible.

Voyons comment on écrirait la condition précédente avec un `case` :

console

```
#!/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
```

Cela fait beaucoup de nouveautés d'un coup.
Analysons la structure du `case` !

console

```
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.

console

```
"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`.

console

```
*)
```

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.

console

```
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 :

console

```
#!/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`.

Le type de boucle que l'on rencontre le plus couramment en bash est `while`.

Le principe est de faire un code qui ressemble à ceci :

console

```
TANT QUE test
FAIRE
------> effectuer_une_action
RECOMMENCER
```

En bash, on l'écrit comme ceci :

console

```
while [ test ]
do
        echo 'Action en boucle'
done
```

Il est aussi possible, comme pour le `if`, d'assembler les deux premières lignes en une, à condition de mettre un point-virgule :

console

```
while [ test ]; do
echo 'Action en boucle'
done
```

On va demander à l'utilisateur de dire « oui » et répéter cette action tant qu'il n'a pas fait ce que l'on voulait. Nous allons créer un script `boucles.sh` pour l'occasion :

console

```
#!/bin/bash

while [ -z $reponse ] || [ $reponse != 'oui' ]
do
        read -p 'Dites oui : ' reponse
done
```

On fait deux tests.

1. Est-ce que `$reponse` est vide ?
2. Est-ce que `$reponse` est différent de `oui` ?

Comme il s'agit d'un OU (`||`), tant que l'un des deux tests est vrai, on recommence la boucle. Cette dernière pourrait se traduire par : « Tant que la réponse est vide ou que la réponse est différente de `oui` ».
Nous sommes obligés de vérifier d'abord si la variable n'est pas vide, car si elle l'est, le second test plante (essayez, vous verrez).

Essayons ce script :

console

```
Dites oui : euh
Dites oui : non
Dites oui : bon
Dites oui : oui
```

Comme vous pouvez le voir, il ne s'arrête que lorsque l'on a tapé `oui` !

Il existe aussi le mot clé `until`, qui est l'exact inverse de `while`. Il signifie « Jusqu'à ce que ». Remplacez juste `while` par `until` dans le code précédent pour l'essayer.

Avertissement pour ceux qui ont déjà fait de la programmation : le `for` en bash ne se comporte pas de la même manière que le `for` auquel vous êtes habitués dans un autre langage, comme le C ou le PHP. Lisez donc attentivement.

Parcourir une liste de valeurs

La boucle `for` permet de parcourir une liste de valeurs et de boucler autant de fois qu'il y a de valeurs.

Concrètement, la forme d'un `for` est la suivante :

console
```
POUR variable PRENANT valeur1 valeur2 valeur3
FAIRE
------> effectuer_une_action
VALEUR_SUIVANTE
```

La variable va prendre successivement les valeurs `valeur1`, `valeur2`, `valeur3`. La boucle va donc être exécutée trois fois et la variable vaudra à chaque fois une nouvelle valeur de la liste.

En bash, la boucle `for` s'écrit comme ceci :

console
```
#!/bin/bash

for variable in 'valeur1' 'valeur2' 'valeur3'
do
        echo "La variable vaut $variable"
done
```

Ce qui donne, si on l'exécute :

console
```
La variable vaut valeur1
La variable vaut valeur2
La variable vaut valeur3
```

Vous pouvez donc vous servir du `for` pour faire une boucle sur une liste de valeurs que vous définissez :

console
```
#!/bin/bash

for animal in 'chien' 'souris' 'moineau'
do
        echo "Animal en cours d'analyse : $animal"
done
```

console
```
Animal en cours d'analyse : chien
Animal en cours d'analyse : souris
Animal en cours d'analyse : moineau
```

Toutefois, la liste de valeurs n'a pas besoin d'être définie directement dans le code. On peut utiliser une variable :

console
```
#!/bin/bash

liste_fichiers=`ls`
```

```
for fichier in $liste_fichiers
do
        echo "Fichier trouvé : $fichier"
done
```

Ce script liste tous les fichiers trouvés dans le répertoire actuel :

console

```
Fichier trouvé : boucles.sh
Fichier trouvé : conditions.sh
Fichier trouvé : variables.sh
```

On pourrait faire un code plus court sans passer par une variable `$liste_fichiers` en écrivant :

console

```
#!/bin/bash
for fichier in `ls`
do
        echo "Fichier trouvé : $fichier"
done
```

Bien entendu, ici, on ne fait qu'afficher le nom du fichier, ce qui n'est ni très amusant ni très utile. On pourrait se servir de notre script pour renommer chacun des fichiers du répertoire actuel en leur ajoutant un suffixe `-old` par exemple :

console

```
#!/bin/bash

for fichier in `ls`
do
        mv $fichier $fichier-old
done
```

Essayons de voir si l'exécution du script renomme bien tous les fichiers :

console

```
$ ls
boucles.sh  conditions.sh  variables.sh
$ ./boucles.sh
$ ls
boucles.sh-old  conditions.sh-old  variables.sh-old
```

À vous de jouer ! Essayez de créer un script `multirenommage.sh`, reposant sur ce principe, qui va rajouter le suffixe `-old`… uniquement aux fichiers qui correspondent au paramètre envoyé par l'utilisateur !

console

```
./multirenommage.sh *.txt
```

Si aucun paramètre n'est envoyé, vous demanderez à l'utilisateur de saisir le nom des fichiers à renommer avec `read`.

Un `for` plus classique

Pour les habitués d'autres langages de programmation, le `for` est une boucle qui permet de faire prendre à une variable une suite de nombres.

En bash, comme on l'a vu, le `for` permet de parcourir une liste de valeurs. Toutefois, en trichant un peu à l'aide de la commande `seq`, il est possible de simuler un `for` classique :

console

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

```
for i in `seq 1 10`;
do
        echo $i
done
```

Explication : `seq` génère tous les nombres allant du premier paramètre au dernier paramètre, donc 1 2 3 4 5 6 7 8 9 10.

console

```
1
2
3
4
5
6
7
8
9
10
```

Si vous le voulez, vous pouvez changer le pas et avancer de deux en deux par exemple. Dans ce cas, il faut écrire `seq 1 2 10` pour aller de 1 à 10 en avançant de deux en deux ; cela va donc générer les nombres 1 3 5 7 9.

En résumé

- Pour exécuter une série de commandes plusieurs fois, on utilise des boucles.
- `while` permet de boucler tant qu'une condition est remplie. Le fonctionnement des conditions dans les boucles est le même que celui des blocs `if` découverts dans le chapitre précédent.
- `for` permet de boucler sur une série de valeurs définies. À l'intérieur de la boucle, une variable prend successivement les valeurs indiquées.