Operating systems

Adil ENAANAI

adil.enaanai@gmail.com
Computer Science Department
Faculty of Science -Tetouan-

Chapter 7

Shell scripts

What is a shell?

Definition

The **shell is** a program that acts as a text-mode interface between the kernel and the user. The **shell is** a command interpreter and programming language. The **shell is a text-mode** interface with the keyboard as input and the screen as output.

What is a shell?

The various console environments (shells)

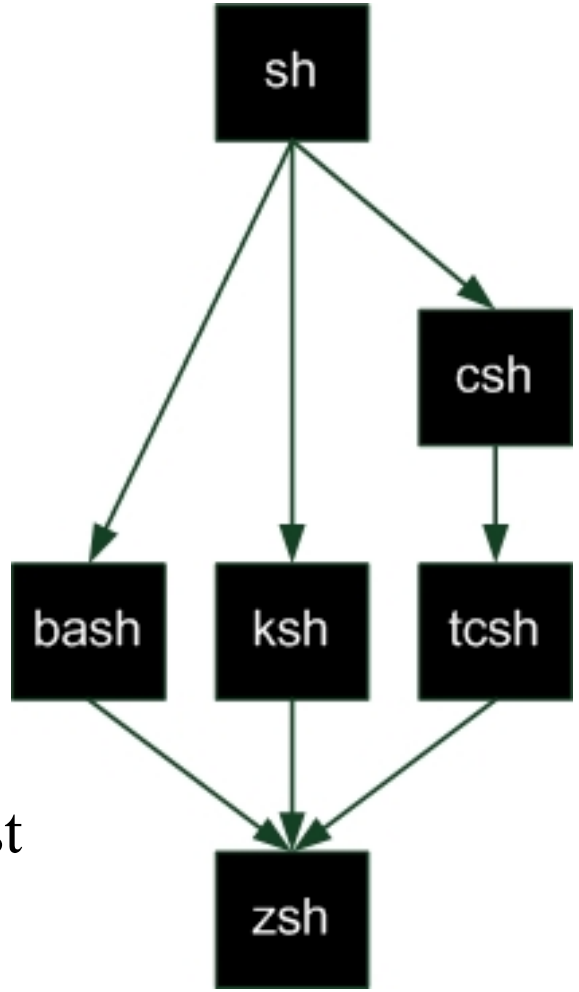
sh: Bourne Shell. The ancestor of all shells. **bash:** Bourne Again Shell. An enhancement to Bourne Shell, available by default on Linux and Mac OS X.

ksh: Korn Shell. A powerful shell common on proprietary Unixes, but also available in a free, bash-compatible version.

csh: C Shell. A shell using a syntax similar to language.

tcsh: Tenex C Shell. C Shell enhancement.

zsh: Z Shell. A fairly recent shell incorporating the best ideas from bash, ksh and tcsh.



First program

Hello.sh

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

- The first line tells Unix that the file is to be executed by **/bin/sh**. This is the standard *Bourne shell* location on just about all Unix systems.
- The second line begins with a special symbol: #. This marks the line as a comment and is completely ignored by the shell.
- The third line executes the **echo** command with the "**Hello World**" parameter.

First program

So far, the Bonjour.sh file does not have execution permission. It must therefore be given this permission with the command:

CHMOD 755

To execute the script, just type:

./Hello.*sh* or bash ***Hello.sh***

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

The ./ path

When you type a command ("ls" for example), the shell looks in the PATH, which tells it where to find the command code.
To see what your PATH looks like, type into your console:

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

In other words, the shell will check whether the definition of the command typed ("ls" to continue with the same example) is in **/usr/local/sbin** then **/usr/local/bin**... until it finds it.

The ./ path

When you type a command, the shell looks in the PATH, which tells it where to look for the command code.

To see what your PATH looks like, type into your console:

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

In other words, the shell will check whether the definition of the command typed ("ls" to continue with the same example) is in **/usr/local/sbin** then **/usr/local/bin**... until it finds it.

To add our directory to $PATH

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

Now you can type the name of the executable file directly

For a global variable

some environment variables to be aware of: HOME, USER, PATH, IFS, etc.

Variables

Note that there must be no spaces around the "=" sign:
VAR=value *works*;
VAR = value *does not work*.
In the first case, the shell sees the "=" symbol and treats the command as a variable assignment.
In the second case, the shell assumes that VAR must be the name of a command and attempts to execute it.
Note also that we need quotation marks around the Hello World string.

The $ sign to display the value of a variable

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

Variables

The shell doesn't care about variable types; they can store strings, integers, real numbers - you name it.

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

Variables

Quotes

You can use quotes to delimit a parameter containing spaces. There are three types of quotes:

- apostrophes ' ' (simple quotes) ;
- double quotes ;
- back quotes, which are inserted with Alt Gr + 7 on a French AZERTY keyboard.

Depending on the type of quotes you use, bash will react differently.

Variables

Quotes

Simple quotes ' '

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

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

Simple quotes " "

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

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

Variables

Quotes

Back quotes ` `

A little unusual, back quotes ask bash to execute what's inside.

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

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

Variables

Control as variable

The result of a command can be used as follows:

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

nbre_lignes will contain the number of lines contained in fichier.ext

Variables

Control as variable

Some variables are a bit special

Name	Function
$*	contains all arguments passed to the
$#	contains the number of arguments
$?	contains the return code of the last operation
$0	contains the script name
$n	contains argument n, where n is a number
$!	contains the PID of the last command run

Variables

Control as variable

Example: create a ***script.sh*** file with the following contents:

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

Run this script with one or more arguments and you'll get

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

Variables

Tables

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

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

To count the number of elements in the array: To `len=${#tab[*]}` ou `echo ${#tab[@]}`

display an element : `echo ${tab[1]}`

To display all items: or `echo ${tab[@]}`

or even

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

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

Variables

Command line arguments

Passing arguments on the command line is again very simple. Each argument is numbered and then called by its number:

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

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

Basic commands File test

operator

It can be used to perform a test and return 0 if all went well, or 1 if there was an error.

Syntax	Function performed
-e file	returns 0 if file exists.
-d file	returns 0 if file exists and is a directory.
-f file	returns 0 if file exists and is a 'normal' file.
-w file	returns 0 if file exists and is writable.
-x file	returns 0 if file exists and is executable.
f1 **-nt** f2	returns 0 if f1 is more recent than f2.
f1 **-ot** f2	returns 0 if f1 is older than f2.

Basic commands File test

operator

Example

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

Or

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

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

Therefore: **test** -f *File* is equivalent to **[** -f *File* **]**.

Don't forget the spaces

Basic controls

Numerical comparison operators

Syntax1	Syntax2	Function performed
[$a -lt 5]	(($a<5))	returns 0 if $a is strictly less than 5
[$a -le 5]	(($a<=5))	returns 0 if $a is less than or equal to 5
[$a -gt 5]	(($a>5))	returns 0 if $a is strictly greater than 5
[$a -ge 5]	(($a>=5))	returns 0 if $a is greater than or equal to 5
[$a -eq 5]	(($a==5))	returns 0 if $a equals 5
[$a -ne 5]	(($a!=5))	returns 0 if $a is not equal to 5

Basic controls

Numerical comparison operators

Example

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

or

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

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

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

or

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

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

Basic commands

Logic operators

These operators allow you to decide whether or not to execute a command based on the return code of another command. They are evaluated from left to right.

Operator	Meaning
&&	AND logic
||	OR logic

Operator evaluation &&.

Syntax

order1 && order2

- The second command is only executed if the first command returns a true code.
- The global expression is true if the

Evaluation of the || operator

Syntax

order1 || order2

- The second command is executed only if the first command returns a false code.
- The global expression is false if the two commands return false.

Basic commands Logic

operators

Example

If the file given in the parameter exists, then [-f $1] returns **0**, so [! -f $1]. returns **1**. and therefore the **echo** command is executed

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

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

If the file given in the parameter exists, then [-f $1] returns 0, so the **echo** command is executed

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

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

Basic controls

Combining logical conditions

($1>0) and ($2>0 or $3>0) is equivalent to:

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

AND OR

Or

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

Basic commands

Double bracket syntax

These conditions offer everything that single-hook conditions and more do. However, they are not compatible with sh. They take the following form.

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

These improved conditions feature wildcard usage as in bash, as well as regular expressions. Thus, it is possible to have conditions like :

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

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

Basic commands

Arithmetic

The Expr command

Syntax

expr *$nbr1* operator *$nbr2*

expr *$chaine* **:** *regular expression*

NB: Be careful, as some operators have a special meaning for the shell, they must be protected by a backslash.

Operators	Meaning
Arithmetic operators	
$nb1 + $nb2	Addition
$nb1 - $nb2	Subtraction
$nb1 * $nb2	Multiplication
$nb1 / $nb2	Division
$nb1 % $nb2	Modulo
Comparison operators	
$nb1 \> $nb2	**TRUE** if $nb1 is strictly greater than $nb2
$nb1 \>= $nb2	**TRUE** if $nb1 is greater than or equal to $nb2
$nb1 \< $nb2	**TRUE** if $nb1 is strictly less than $nb2
$nb1 \<= $nb2	**TRUE** if $nb1 is less than or equal to $nb2
$nb1 = $nb2	**TRUE** if $nb1 is equal to $nb2
$nb1 != $nb2	**TRUE** if $nb1 is different from $nb2

Basic commands

Arithmetic

The Expr command

Syntax

expr *$nbr1* operator *$nbr2*

expr *$chaine* **:** *regular expression*

NB: Be careful, as some operators have a special meaning for the shell, they must be protected by a backslash.

Operators	Meaning
Logic operators	
$chaine1 \& $chaine2	**TRUE** if both strings are true
$chaine1 | $chaine2	**TRUE** if one of the 2 strings is true
Various operators	
-$nb1	Opposite of $nb1
\(expression \)	Grouping
$chaine : regular_expression	Compares the string with the regular expression

Basic commands

Arithmetic

The Expr command

Example

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

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


Retrieving the result in a variable

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

Basic controls

Arithmetic

The (( )) command

Syntax

((arithmetic_expression))

The (( )) command has a number of advantages over the **expr** command

- Additional operators
- Arguments need not be separated by spaces
- Variables do not need to be prefixed with the $ symbol.
- Shell special characters need not be protected by backslashes
- Assignments are made using the
- Faster execution

Basic commands

Arithmetic

The (( )) command

A large number of operators come from the C language

Operators	Meaning
Arithmetic operators	
nbr1 + nbr2	Addition
nbr1 - nbr2	Subtraction
nbr1 * nbr2	Multiplication
nbr1 / nbr2	Division
nbr1 % nbr2	Modulo

Operators	Meaning
Operators working on bits	
~nbr1	Complement to 1
nbr1 >> nbr2	Shift nbr1 by nbr2 bits to the right
nbr1 << nbr2	Shift nbr1 by nbr2 bits to the left
nbr1 & nbr2	AND bit by bit
nbr1 | nbr2	OR bitwise
nbr1 ^ nbr2	Exclusive bitwise OR

Basic commands

Arithmetic

The (( )) command

A large number of operators come from the C language

Operators	Meaning
Comparison operators	
nbr1 > nbr2	**TRUE** if nbr1 is strictly greater than nbr2
nbr1 >= nbr2	**TRUE** if nbr1 is greater than or equal to nbr2
nbr1 < nbr2	**TRUE** if nbr1 is strictly less than nbr2
nbr1 <= nbr2	**TRUE** if nbr1 is less than or equal to nbr2
nbr1 == nbr2	**TRUE** if nbr1 is equal to nbr2
nbr1 != nbr2	**TRUE** if nbr1 is different from nbr2

Basic commands

Arithmetic

The (( )) command

Operators	Meaning
Logic operators	
!nbr1	Inverts the truth value of nbr1
&&	AND
||	OR
Various operators	
-nbr1	Opposite of nbr1
nbr1 = expression	Assignment
(expression)	Grouping
nbr1 binop= nbr2	*binop* represents one of the following operators: +, -, /, *, %, >>, <<,

Basic controls

Arithmetic

The (( )) command

Examples

Add 10 to nbr1 (2 different methods)

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

Test if nbr1 is greater than nbr2 and vice versa

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

Grouping and logic tests

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

Basic commands

Arithmetic

The LET command

The let command is equivalent to ((expression))

Syntax

let "expression"

Examples

Multiply nbr1 by 3

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

Calculate the modulo of nbr1 by 2 and assign it to the variable nbr2

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

Basic commands

The IF command

The if conditional control structure allows you to perform an action based on a logical expression.

Syntax 1

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

Syntax 2

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

Syntax 3

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

Basic commands

The IF command

Examples

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

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

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

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

Basic commands

The CASE command

The **case** control structure can also be used to perform tests. It can be used to direct the program sequence according to a choice of different values.

When there are a large number of choices, the **case** command is more appropriate than the **if** command.

Basic controls

The CASE command

Syntax

```
case $variable in
modele1) commande1
    ...
    ;;
model2) order2
    ...
    ;;
model3 | model4 | model5 ) order3
    ...
    ;;
esac
```

The shell compares the value of the variable with the various models entered.
When the value matches the model, the commands in the block are executed.
The **;;** characters are used to close the block and end the **case**.
The shell continues with the first command under **esac**.

Basic commands

The CASE command

Syntax

```
case $variable in
modele1) commande1
    ...
    ;;
model2) order2
    ...
    ;;
model3 | model4 | model5 ) order3
    ...
    ;;
esac
```

The shell compares the value of the variable with the various models entered.
When the value matches the model, the commands in the block are executed.
The **;;** characters are used to close the block and end the **case**.
The shell continues with the first command under **esac**.

Don't forget the characters ;; because this will generate an error.

Basic commands

The CASE command

Special characters :

Special characters for string templates	Meaning
Special characters valid in all shells :	
*	0 to n characters
?	1 any character
[abc]	1 character among those entered in square brackets
[!abc]	1 character not included between square brackets

Basic commands

The CASE command

Special characters :

Special characters not valid in Bourne Shell. In bash, activate the extglob option (shopt -s extglob)	
?(expression)	from 0 to 1 times the expression
*(expression)	from 0 to n times the expression
+(expression)	from 1 to n times the expression
@(expression)	1 times the expression
!(expression)	0 times the expression
?(expression1 | expression2 | ...) *(expression1 | expression2 | ...) +(expression1 | expression2 | ...) @(expression1 | expression2 | ...) !(expression1 | expression2 | ...)	alternatives

Basic commands The

SELECT command

Select is an extension of case. The list of possible choices is made at the beginning and the user's choice is used to perform the same processing:

```
#!/bin/bash

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

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

Basic commands

The CASE command

Work to do

The following script allows you to create, modify, view and delete a file in the script execution directory.
It takes a file name as argument and displays a menu. Use of case with if nesting.

```
1(Create)
2(Edit)
3(View)
4(Delete)
Your choice
```

Basic commands

The FOR loop

Repeat the action for each list item.

Syntax

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

Example

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

Example

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

Basic commands

The FOR loop

Repeat the action for each value of i according to. Stop control is defined by a loop exit condition.

Syntax

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

Example

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

Output

Display numbers from 10 to 0 i -= 1 equivalent to i = i - 1

Basic controls The

WHILE loop

Repeat action as long as condition is met

Syntax

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

Example

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

or

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

Output

Displaying numbers from 0 to 10

Basic controls The

WHILE loop

Example: Analyze and interpret the following scripts

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

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

Basic controls The

WHILE loop

Example: Analyze and interpret the following scripts

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


The continue keyword allows you to return to the while loop without executing the following command

Basic controls The

WHILE loop

Example: Analyze and interpret the following scripts

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

This script causes an infinite loop because the counter is not incremented.

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

make an infinite loop

Basic commands

The UNTIL loop

Unlike **while**, the **until** command executes the commands between **do** and **done** until the command to the right of **until** returns **false** code.

Syntax

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

Example

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

\c: not to return to the line

Basic BREAK and

CONTINUE

The **break** and **continue** commands can be used inside for, while, until and select loops.

The **break** command is used to exit a loop.

The **continue** command is used to go back to the condition of a loop.

Syntax

Break: Exit the first-level loop

break n: Exit level n loop

Continue: Go back to first-level loop condition

continue n: Go back to level n loop condition

Basic BREAK and

CONTINUE

The **break** and **continue** commands can be used inside for, while, until and select loops.

The **break** command is used to exit a loop.

The **continue** command allows you to go back to the condition of a loop.

Syntax

Break: Exit the first-level loop

break n: Exit level n loop

Continue: Go back to first-level loop condition

continue n: Go back to level n loop condition

End of chapter 7