Conditional execution

Conditional statements:

- ★If … then
- ★ If … then … else
- ★ If … then … elif
- ★ case

Conditional statement “if...then”

The if construction allows you to specify different courses of action to be taken in a shell script, depending on the success or failure of a command.

The most compact syntax of the if command is:

```
if TEST-COMMANDS; then COMMANDS; fi
```

Which is the same, less compact:

```
if TEST-COMMANDS
    then COMMANDS
fi
```

The TEST-COMMAND list is executed, and if its return status is one (True), the COMMANDS are executed. The return status is the exit status of the last command executed, or zero if the condition tested is False.

Example of conditional statement “if...then”

- Testing exit status

The ? variable holds the exit status of the previously executed command (the most recently completed foreground process).

Example

Test to check if a command has been successfully executed:

```
ls -l
if [ $? -eq 0 ]
   then echo 'That was a good job!'
fi
```

- Numeric comparisons

The example below use numerical comparisons:

```
num=$(less work.txt | wc -l)
echo $num
if [[  "$num" -gt "150"  ]]
then echo ; echo "you've worked hard enough for today."
fi
```

Main conditional operators

Relational operators

- -lt (<) lower-than
- -gt (>) greather-then
- -le (<=) lower-equal
- -ge (>=) greather-equal
- -eq (==) equal
- -ne (!=) not equal

Boolean operators

- && and
- || or
- | not

Files operators:

- if [-x "$filename"]; then # if filename is executable
- if [-e "$filename"]; then # if filename exists
- ……

Condition check

The [[ ]] construct is the more versatile Bash version of [ ].
This is the extended test command.

No filename expansion or word splitting takes place between [[and]], but there is parameter expansion and command substitution.

```
file=/etc/passwd
if [[ -e $file ]]
then
  echo "Password file exists."
fi
```

Using the [[...]] test construct, rather than [...] can prevent many logic errors in scripts. For example, the &&, ||, <, and > operators work within a [[ ]] test, despite giving an error within a [ ] construct.

Exercise: True and false result

```
a=3

(($a>10))
echo $?          # print 1 because the condition is false

(($a>2))
echo $?          # print 0 because the condition is true
```

Strings comparison example (try)

```
#!/bin/bash
s1='string'
s2=''
if [ ${s1} != ${s2} ]
then
        echo "s1 ('${s1}') is not equal to s2 ('${s2}')"
fi
if [ ${s1} = ${s1} ]
then
        echo "s1('${s1}') is equal to s1('${s1}')"
fi
```

Some issue? Try the [[ ]] construct

Check if a variable is empty example

Try:

```
if [[ X == X${variable_to_check} ]]
  then
      echo “variable is empty”
 else
      echo “variable value is ${variable_to_check}”
fi
```

Then try:

```
variable_to_check=”I_am_not_empty”
if [[ X == X${variable_to_check} ]]
  then
      echo “variable is empty”
 else
      echo “variable value is ${variable_to_check}”
fi
```

Nested conditional if...then statement

```
a=3

if [ "$a" -gt 0 ]
then
  if [ "$a" -lt 5 ]
  then
    echo "The value of \"a\" lies somewhere between 0 and 5."
  fi
fi

# Same result as:

if [ "$a" -gt 0 ] && [ "$a" -lt 5 ]
then
  echo "The value of \"a\" lies somewhere between 0 and 5."
fi
```

Conditional statement “if...then...else”

```
if [ condition-true ]
then
   command 1
   command 2
   ...
else  # Adds default code block executing if original condition tests false.
   command 3
   command 4
   ...
fi
```

Note:
When if and then are on same line in a condition test, a semicolon must terminate the if statement. Both if and then are keywords. Keywords (or commands) begin statements, and before a new statement on the same line begins, the old one must terminate.

Exercise: “if...then...else”

Write a simple example of the construct if...then...else

Suggestion:
Basic example of if .. then ... else:

```
#!/bin/bash
if [ "foo" = "foo" ]; then
   echo expression evaluated as true
else
   echo expression evaluated as false
fi
```

Example of condition with variables:

```
#!/bin/bash
t1="foo"
t2="bar"
if [ "$t1" = "$t2" ]; then
    echo expression evaluated as true
else
    echo expression evaluated as false
fi
```

Loops

Loop statements:

- ★ for
- ★ while
- ★ until

for loop

Executes an iteration on a set of words.
It is slightly different from other languages (like C) where the iteration is done respect to a numerical index.

Syntax:
```
for CONDITION; do
    COMMANDS
done
```

Examples:
```
#!/bin/bash
for i in $( ls ); do
    echo item: $i
done
```

C-like for:
```
#!/bin/bash
for i in $(seq 1 10)
do
        echo $i
done
```

for loop examples (try)

Counting:

```
#!/bin/bash
for i in {1..25}
do
    echo $i
done
```

or:

```
#!/bin/bash
for ((i=1 ; i<=25 ; i++))
do
    echo $i
done
```

Counting on "n" steps

```
#!/bin/bash
for i in {0..25..5}
do
    echo $i
done
```

That will count with 5 to 5 steps.

Counting backwards

```
#!/bin/bash
for i in {25..0..-5}
do
    echo $i
done
```

Acting on files

```
#!/bin/bash
for file in ~/*.txt
do
echo $file
done
```

That example will just list all files with "txt" extension. It is the same as ls *.txt

Calculate prime numbers

```
#!/bin/bash
read -p "How many prime numbers ?:
" num
c=0
k=0
n=2

while [ ${k} -ne ${num} ]
do
    for i in $(seq 1 ${n})
    do
        r=$((n % i))

if [ ${r} -eq 0 ]
then
            c=$((c + 1))
fi
    done

    if [ ${c} -eq 2 ]
    then
        echo "${i}"
k=$((k + 1))
    fi
    n=$((n + 1))
    c=0
done
```

break statement in for loop

break statement is used to break the loop before it actually finish executing.
You are looking for a condition to be met, you can check the status of a variable for that condition.
Once the contidition is met, you can break the loop. Pseudo-code example:

```
for i in [series]
do
        command 1
        command 2
        command 3
        if (condition) # Condition to break the loop
        then
                command 4 # Command if the loop needs to be broken
                break
        fi
        command 5 # Command to run if the "condition" is never true
done
```

With the use of if ... then you can insert a condition, and when it is true, the loop will be broken with the break statement

continue statement in for loop

continue stop the execution of the commands in the loop and jump to the next value in the series.
It is similar to continue which completely stop the loop.

Pseudo-code example:

```
for i in [series]
do
    command 1
    command 2
    if (condition) # Condition to jump over command 3
            continue # skip to the next value in "series"
    fi
    command 3
done
```

break statement in iteration

break command is used to exit out of current loop completely before the actual ending of loop.
Break command can be used in scripts with multiple loops. If we want to exit out of current working loop whether inner or outer loop, we simply use break but if we are in inner loop & want to exit out of outer loop, we use break 2.
Example

```
#!/bin/bash
# Breaking outer loop from inner loop
for (( a = 1; a < 5; a++ )); do
    echo “outer loop: $a”
    for (( b = 1; b < 100; b++ )); do
      if [ $b –gt 4 ]; then
        break 2
      fi
    echo “Inner loop: $b ”
    done
done
```

The script start with a=1 & move to inner loop and when it reaches b=4, it break the outer loop.

Exercise:
In this same script, use break instead of break 2, to break inner loop & see how it affects the output.

continue statement in iteration

continue command is used in script to skip current iteration of loop & continue to next iteration of the loop.

Example

```
#!/bin/bash
# using continue command
for i in 1 2 3 4 5 6 7 8 9
do
   if [[ $i == 5 ]]
   then
       echo “skipping number 5”
       continue
   fi
   echo “I is equal to $i”
done
```

while loop

Executes one or more instructions while a condition is true.
It stops when the control condition is true or when the execution is intentionally stopped by the programmer with an explicit interruption instruction (break or continue)

Syntax:

```
while CONDITION; do
    COMMANDS
done
```

Example:

```
#!/bin/bash
counter=0
while [  ${counter} -lt 10 ]; do
    echo The counter is ${counter}
    let counter=counter+1
done
```

Example of break statement in while loop

Interrupt the loop at number … (try)

```
#!/bin/bash

num=1
while [ $num -lt 10 ]
do
   if [ $num -eq 5 ]
   then
      echo “$num equal to 5 so I interrupt the loop”
   break
   fi
   echo $num
   let num+=1
done
echo “Loop is complete”
```

until loop

Executes one or more instructions until a condition is false.

Syntax:

```
until CONDITION; do
    COMMANDS
done
```

Example:

```
#!/bin/bash
counter=20
until [  $counter -lt 10 ]; do
    echo counter $counter
    let counter-=1
done
```

until vs. while

Until is similar to while, but it is a slightly difference:
Until is executed while the condition is false,
While is executed while the condition is true.
What means it?

Try the following code and check the output:

```
num=1
while [[ $num -lt 10 ]]
do
  if [[ $num -eq 5 ]]
  then
    break
  fi
  echo $num
  let num=num+1
done
echo “Loop while is complete”
```

```
num1=1
until [[ $num1 -lt 10 ]]
do
  if [[ $num1 -eq 5 ]]
  then
    break
  fi
  echo $num1
  let num1=num1+1
done
echo “Loop until is complete”
```

Loop through array elements

${myArray[@]} return all the elements of an array
replace the numeric index with the @ symbol can be thought as standing for all.

Example: Loop on all elements of the array:

```
myArray=(1, 3, 5, ”try” , ”this” ,1)

for t in ${myArray[@]}; do
  echo array element ${t}
done
```

Loop through array indices

${!allThreads[@]} returns all the indexes in an array.

Example: Loop on all indexes of the array:

```
myArray=(1, 3, 5, ”try” , ”this”, 1)

for i in ${!myArray[@]}; do
  echo “Array element ${i} is = ${myArray[$i]}”
done
```