BASH SCRIPTING

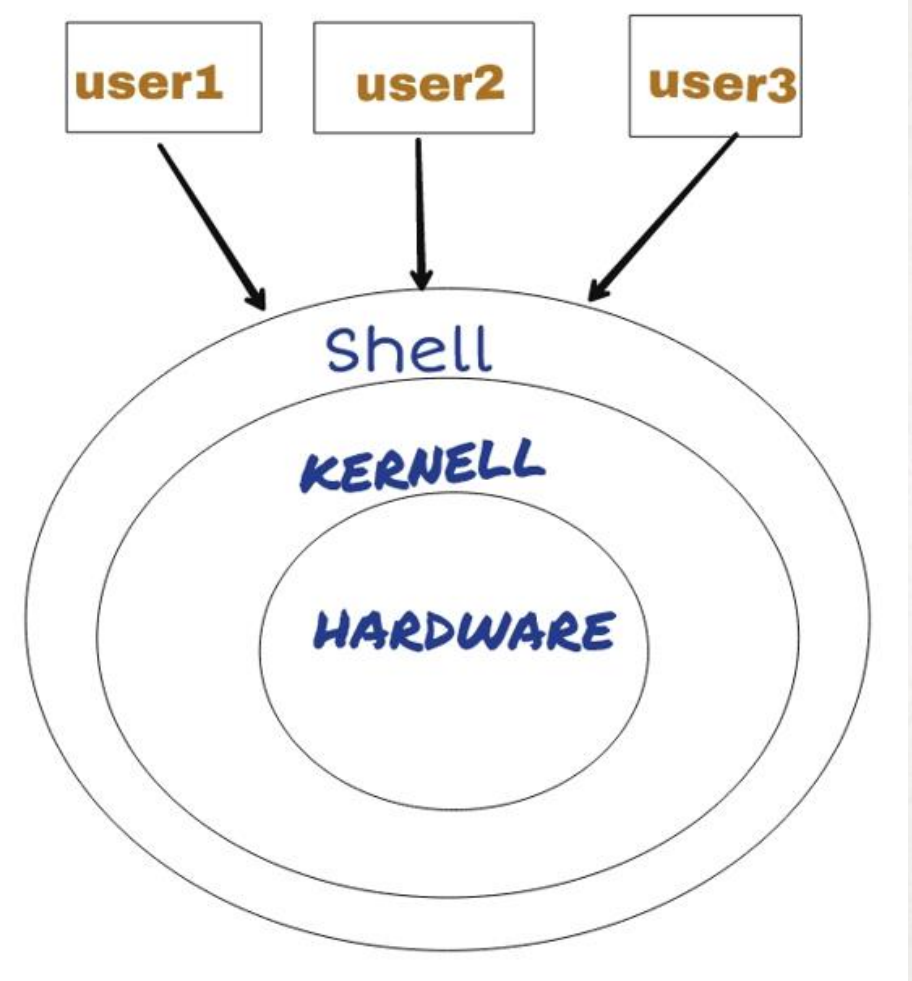
BY / ISLAM SALAH

COURSE AGENDA

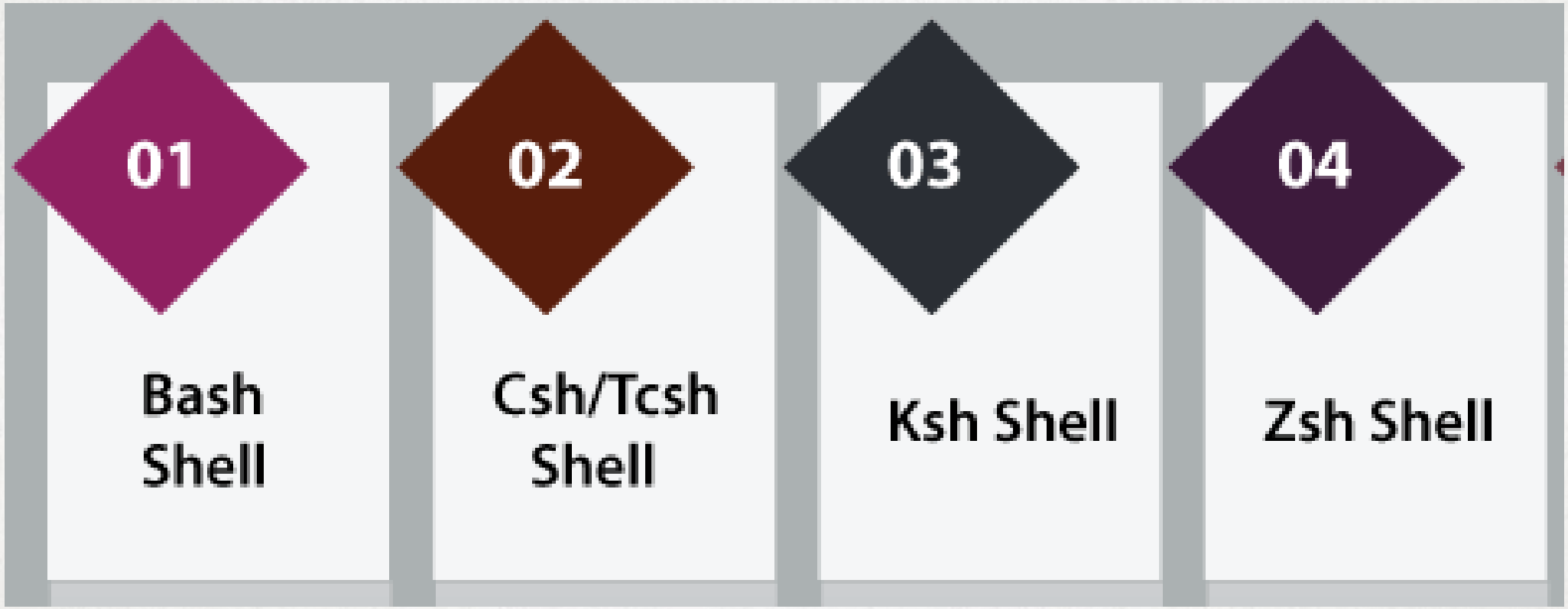
- What is Shell ?
- Types of Shell
- What is Shell Scripting?
- Benefits of Shell Scripting
- Basic Linux Commands
- Basic Bash Scripting
- Useful Shell commands
- Conditionals
- Loops
- Functions

WHAT IS SHELL ?

- A shell accepts and interprets commands.
- A shell is an environment in which commands, programs, and shell scripts are run.
- There are many types of Linux shells available. Bash is one of them, and this course discusses it further.



MOST COMMON SHELL TYPES



MOST COMMON SHELL TYPES

Shell Type	Description
Bash or Bourne Again Shell	This is the most common shell available on all Linux and debian based systems. It is open source and freeware.
CSH or C Shell	This Shell scripting program uses the C programming's shell syntax and its almost similar to C.
KSH or Korn Shell	This shell is quite advanced and it's a high level programming language, used in Oracle.
TCSH	TCSH is an advanced version of Berkeley Unix C shell. It again supports C style syntax

WHAT IS BASH?

- Bash is the default shell in Linux.
- It offers an efficient environment for interacting with the operating system and scripting.
- Bash is a programming language for running commands. Bash is the default shell in Linux operating systems. It is widely used, so some familiarity with Bash is expected in many systems or development roles.

BASIC BASH SCRIPTING

Type of Variables:

- Local Variables
- Environment Variables
- Predefined Variables

BASIC BASH SCRIPTING

Local Variables:

- There are rules for defining or creating variables in the shell.
- When defining a variable, the variable name must be prefixed with the dollar ($) symbol.
- The variable must contain no spaces or special characters within the variable name. A variable name can contain only letters (a to z or A to Z), numbers (0 to 9), or the underscore character (_), and they are usually capitalized (e.g. VARIABLE).

name="Salah"
Age = "28"

BASIC BASH SCRIPTING

Display shell Variables:

```
name="John"
echo ${name}
```

BASIC BASH SCRIPTING

Environment Variables:

Environment Variables	Description
$HOME	Defines the user's home directory
$PATH	Indicates the location of the commands
$SHELL	Defines the login shell type
$USER	Contains the user's system name

BASIC BASH SCRIPTING

Predefined Variables:

Predefined variables are variables known to the shell and their values are assigned by the shell.

`$#`	Number of arguments
`$*`	List of all arguments
`$0`	Script name
`$1, $2,..`	First argument, second argument,..
`$?`	Return code of the last command

HANDS ON AND DEMONSTRATION

1. Open the Bash shell in a Linux environment.

2. At the command prompt, enter the echo command and an environment variable from the following list:

- echo $HOME
- echo $SHELL
- echo $USER
- echo $PATH

3. env command is a shell command for Linux. You use this command to print a list of environment variables or run another utility in an altered environment.

4. Create variable and echo it

THE ALIAS COMMAND

By using aliases, you can define new commands by substituting a long command with a short one.

Aliases can be set temporarily in the current shell, but it is more common to set them in the user's .bashrc file so that they are permanent.

alias alias_name='command'

$ alias ll='ls –la'

$ unalias ll

WHAT ARE SCRIPTS?

- Scripts are text files of commands and related data.
- When the text file is processed, the commands are run.
- Scripts can be set as scheduled tasks by using cron.
- Automation allows scripts to run more quickly than if they are run manually.

WHAT IS SHELL SCRIPTING?

- A shell script is a list of commands in a computer program that is run by the Unix shell to execute commands.

- Shell scripting is scripting in *any* shell, whereas Bash scripting is scripting specifically for Bash.

NEED OF SHELL SCRIPTING

- Creating backup jobs
- Archiving log files
- Configuring systems and services
- Simplifying repetitive task
- Automating tasks

CHARACTERISTIC OF SHELL SCRIPTING

- Shell scripts don’t have to be boring
- Reusing
- Always available
- Readability
- Shell scripting is powerful

BASH SCRIPT

- Create script file.
- chmod command to give execute permission
- Run/Executing script

EXAMPLE BASH SCRIPTING

$ touch backup_script.sh
$ chmod +x backup_script.sh
$ vi backup_script.sh

```
#!/usr/bin/bash

#Script to backup home directory
tar -cf backup_home.tar  /home/salah

echo "backup task done successfully"
```

$ ./backup_script.sh

- #! is referred to as a shebang.

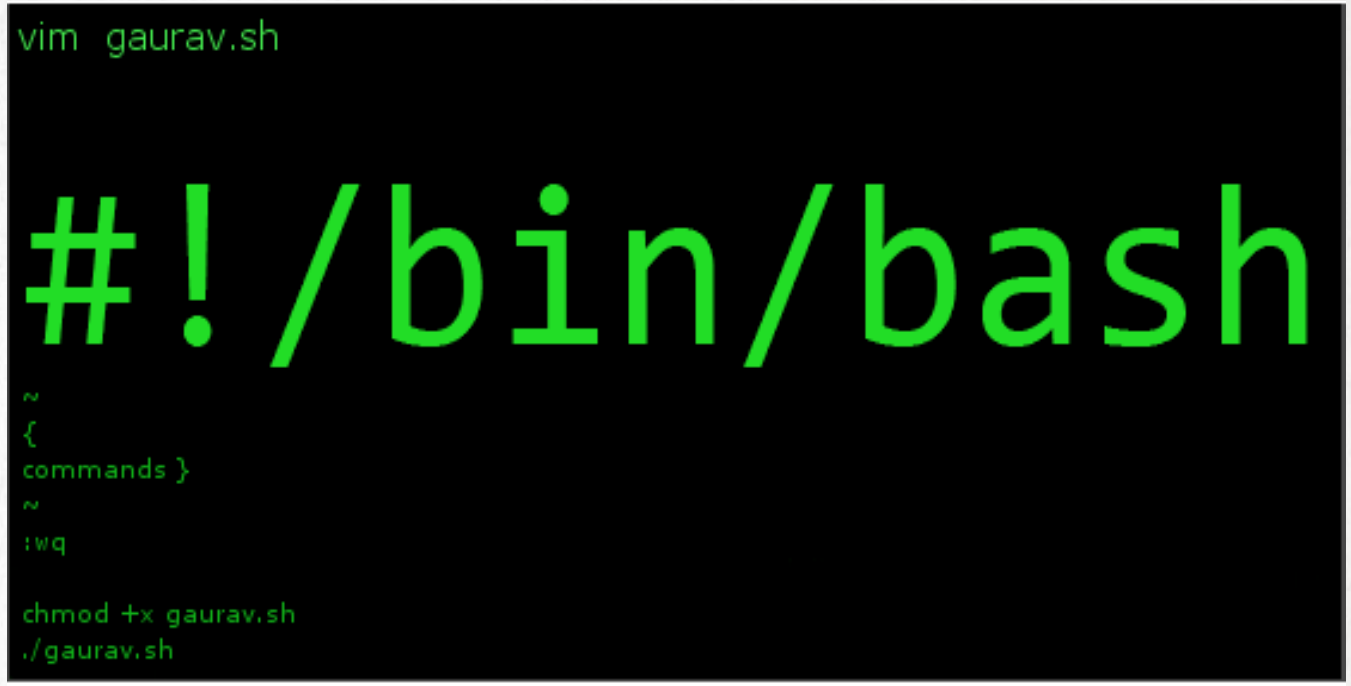
- The first line defines the interpreter to use (gives the path and name of the interpreter).

- Scripts must begin with the directive for which shell will run them.

- The location and shell can be different.

- Each shell has its own syntax, which tells the system what syntax to expect.

- Example: #!/bin/bash



SCRIPT DOCUMENTATION

- Some administrators create a script template, which contains all the relevant information and sections.

- The template might include the following:
  - Title
  - Purpose
  - Author's name and contact information
  - Special instructions or examples

- There is no exact format definition that scripts should follow.

```
#/bin/bash

#.....................
#Author : Jane Doe
# Date : 06/15/2021

#Description :Here is how you can document a script
#
#
#Usage :
# ./myScript.sh param1 [param2]
#param 1:
#param2:
#.....................
#Version: 2.0.1

#Declared variables

#.....................
#Scrit body
```

USEFUL LINUX COMMANDS

Command	Description
echo	Displays information on the console
read	Reads a user input
subStr	Gets the substring of a string
+	Adds two numbers or combine strings
file	Opens a file
mkdir	Creates a directory
cp	Copies files
mv	Moves or renames files
chmod	Sets permissions on files
rm	Deletes files, folders, etc.
ls	Lists directories

USEFUL LINUX COMMANDS

pwd Get the full path of the current working directory.
cd ~ or just cd Navigate to the current user's home directory.
cd .. Go to the parent directory of current directory (mind the space between cd and ..)
ls List the files and directories in the current directory.
cp Will copy the file from source to destination.
mv moves/renames the file1 to file2.
rm -R dir-name Will remove the directory dir-name recursively.
mkdir dir-name - Create a directory dir-name.
touch filename - Create a file filename, if it doesn't exist.
chmod <specification> filename - Change the file permissions.
chown owner1 filename - Change ownership of a file to user owner1.

DECLARE BASH VARIABLE

```
#!/bin/bash

name="Islam Salah“
echo ${name}
echo ${name/J/j} (substitution)
echo ${name:0:2} (slicing)
echo ${name::2} (slicing)
echo ${name::-1} echo ${name:(-1)} (slicing from right)
length=2 echo ${name:0:length}
```

BASIC BASH SCRIPTING

Reading User Input:

```
echo “What is your full name?”
read name
print “hello $name”

print “what is your age?“
read   age
print “I am $age years old”
```

SPECIAL BASH VARIABLE

1	**$0** The filename of the current script.
2	**$n** These variables correspond to the arguments with which a script was invoked.
3	**$#** The number of arguments supplied to a script.
4	**$*** All the arguments are double quoted. If a script receives two arguments, $* is equivalent to $1 $2.
5	**$@** All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2.
6	**$?** The exit status of the last command executed.
7	**$$** The process number of the current shell. For shell scripts, this is the process ID under which they are executing.
8	**$!** The process number of the last background command.

DEMO FOR INTERACTIVE AND SILENT SCRIPT

- Create script to print welcome message, in silent style
- Create script to get name and print message to welcome him, in interactive style

HANDS ON ACTIVITY

- Create script to get name and print it
- Create variable and print it
- Run the script

USEFUL SHELL COMMAND

- grep
- cut
- sed
- awk
- sort

USING | GREP

- grep is commonly used after another command, along with a pipe (|).

- Examples:
- ps -ef | grep sshd
- cat /var/log/secure | grep fail

USING | CUT

The cut command requires the user to specify bytes, fields, or characters to extract.
When using a field, you must specify the delimiter of the file.

Command options are listed as follows.
-b: Byte
-c: Character
-f: Field
-d: Delimiter

Examples are as follows.
• cut -d ',' -f 1 names.cvs
Extracts the first field of each record.
The separator is the comma (,).

• cut –c 1-2 names.csv
Extracts the first two characters of each line.

USING | SED

- A non-interactive text editor
- Edits data based on the rules that are provided (can insert, delete, search, and replace)
- Supports regular expression

- Addressing is used to determine which lines to be edited.
- The addressing format can be:
  - Number (represents a line number)
  - Regular expression
  - Both
- The sed commands tell sed what to do with the line:
  - Print it
  - Remove it
  - Change it

- The sed format
  - sed ‘command’ filename

USING | SED | EXAMPLE

To print lines contain the pattern root
$sed ‘/root/p’ myfile

To suppresses the default behavior of the sed
$sed –n ‘/root/p’ /etc/passwd

To print lines from salah to root
$sed –n ‘/salah/,/root/p’ /etc/passwd

To print lines from 2 to the line that begins with us
$sed –n ‘2,/^us/p’ /etc/passwd

To substitute text
$sed ‘s/salah/fargany/g’ myfile
$sed –n ‘s/salah/fargany/2’ myfile

USING | SED | EXAMPLE

To delete lines from 1 to 3 (including 3rd line)
$sed ‘1,3d’ myfile

To delete from line 3 to the end
$sed ‘3,$d’ myfile

To delete lines containing root pattern
$sed ‘/root/d’ myfile

To delete the third line
$sed ‘3d’ myfile

To delete the last line
$sed ‘$d’ myfile

To issue multi command change
$sed –e ‘2d’ –e ‘s/salah/islamsalah/g’ myfile

USING | AWK

- awk is a programming language used for manipulating data and generating reports.
- awk scans a file line by line, searching for lines that match a specified pattern performing selected actions
- awk stands for the first initials in the last names of each authors of the language, Alfred Aho, Peter Weinberger, and Brian Kernighan
- The awk utility consists of:

  $awk ‘instructions’ inputfile

USING | AWK

By default, each line is called a record and terminated with a new line.

Record separators are by default carriage return, stored in a built-in variables ORS and RS.

The $0 variable is the entire record.
The NR variable is the record number.

Each record consists of words called fields which by default separated by white spaces.
NF variables contains the number of fields in a record
FS variable holds the input field separator, space/tab is the default.

awk [option] file

USING | AWK | EXAMPLE

$awk –F: ‘{print $1}’ /etc/passwd

$awk -F : '{print "Logname:",$1}' /etc/passwd

To display the whole record (cat)
$awk ‘{print $0}’ /etc/passwd

To display the file numbered (cat -n)
$awk ‘{print NR,$0}’ /etc/passwd

To display number of fields (words) in each record (line)
$awk –F: ‘{print $0,NF}’ /etc/passwd

USING | AWK | EXAMPLE

The program is print $3: Prints the third field on each record in the customer.txt file
awk –F , '{ print $3 }' customers.txt

The field separator is @ so the first field becomes everything that is before that @ instead of the first name
awk –F @ ‘{print $1}’ customers.txt

Selects only records for which the second field is > 35
awk -F : '$3 > 35 {print $1,$2}' names.csv

This program prints Start Processing, then displays the first field of each record, and finally displays Done!:
awk 'BEGIN { print "Start Processing." }; { print $1 }; END { print “Done ! " }' names.csv

USING | SORT

Sorts file contents in a specified order: alphabetical, reverse order, number, or month

• Examples:
-o outputs the result to a file
(sort file.txt –o sortedfile.txt is like sort file.txt > sortedfile.txt)
-r sorts in reverser order
-n sorts numerically if the file contains numbers
-k sorts according to the kth column (if the file is formatted as a table)
-u removes duplicates
-t for delimiter

USING | SORT| EXAMPLE

sort names.csv

sort -r names.csv

sort -o names.csv sorted_names.csv

sort -k 2n salary.txt

sort -u names.csv

sort -t ":" -nk3 names.csv

OPERATORS

```
#!/usr/bin/bash

sum=$(($1 + $2))
echo " $1 + $2 equals $sum"
```

Operator	Meaning
<	Less than
<=	Less than and equal
==	Equal to
!=	Not equal to
>=	Greater than and equal
>	Greater than
~	Match regular expression
!~	Not matched by regular expression

TESTING AND LOGICAL OPERATIONS

- String Testing

String1 = string2	string1 is equal string2
String1 != string2	string1 is not equal string2
-z string	Length of string is zero
-n string	Length of string is nonzero

- Integer Testing

int1 –eq int2	equal to
int1 –ne int2	not equal to
int1 –gt int2	greater than
int1 –ge int2	greater than or equal
int1 –lt int2	less than
int1 –le int2	less than or equal

TESTING AND LOGICAL OPERATIONS

-a	and operator
-o	or operator
-f filename	file existence
-h filename	symbolic link
-r filename	readable
-w filename	writable
-x filename	executable

CONDITIONAL STATEMENTS | IF

```
if command
then
        … commands …
fi
```

```
if [ expression ]
then
        … commands …
fi
```

```
Nested condition
if command
      Then
              … commands …
              if command
                    Then
                          … commands …
               fi
fi
```

CONDITIONAL STATEMENTS | IF

```
if [ expression ]
then
        ... commands ...
elif [ expression ]
then
        ... commands ...
else
        ... commands ...
fi
```

Or inline:

```
if <condition>; then <command> fi
```

Note:

- The semicolon (;) is not mandatory if not writing inline.
- Indentation is used for better readability but is not required.

CONDITIONAL STATEMENTS | EXAMPLE

```
#!/usr/bin/bash

if  test  -f   file1
then
                        cat file1
fi
```

```
if  [  -f   file1  ];
then
            cat file1 > new_file.txt
fi
```

CONDITIONAL STATEMENTS | EXAMPLE

```
echo “Are you ok?”
read answer

if [ $answer = Y –o $answer = y ]
then
        echo “Glad to hear that ”
fi
```

```
if [ cp    file1     /tmp ];
then
        rm file1
fi
```

HANDS ON IF STATEMENT

INTEGER COMPARISON OPERATORS

Comparison operators compare two variables or quantities.

- `-eq` is equal to: `if [ "$a" -eq "$b" ]`
- `-ne` is not equal to: `if [ "$a" -ne "$b" ]`
- `-gt` is greater than: `if [ "$a" -gt "$b" ]`
- `-ge` is greater than or equal to: `if [ "$a" -ge "$b" ]`
- `-lt` is less than: `if [ "$a" -lt "$b" ]`
- `-le` is less than or equal to: `if [ "$a" -le "$b" ]`
- < is less than (within double parentheses): `(("$a" < "$b"))`
- <= is less than or equal to (within double parentheses): `(("$a" <= "$b"))`
- > is greater than (within double parentheses): `(("$a" > "$b"))`
- >= is greater than or equal to (within double parentheses): `(("$a" >= "$b"))`

INTEGER COMPARISON OPERATORS| EXAMPLE

```
#!/bin/bash
#Compares integer $1 and $2

if [   "$1"    -gt   "$2“   ];
then
            echo ".."
fi

if ((    "$1“    >  "$2“    ))
then
            echo ".."
fi

if ((    $1   ==   $2  ));
then
            echo ".."
fi
```

INTEGER COMPARISON OPERATORS| EXAMPLE

```
if [ $1   -gt    $2 ]
then
              echo ".."
fi
```

```
if [[ $1   -gt   $2   ]]
then
              echo ".."
fi
```

CONDITIONAL STATEMENTS | EXAMPLE

```
#!/bin/bash
#Compares $1 and $2

if [    $1    -gt    $2    ];
then
                echo "the first number is greater then the second number"
elif [   $1    -lt   $2    ];
then
                echo "the second number is greater then the first number"
else
                echo " the numbers are equal"
fi
```

STRING COMPARISON OPERATORS

- = or == is equal to
  - if ["$a" = "$b"]
  - if ["$a" == "$b"]
- != is not equal to
  - `if [ "$a" != "$b" ]`
  - This operator uses pattern matching within a `[[ ... ]]` construct.
- < is less than, in ASCII alphabetical order
  - `if [[ "$a" < "$b" ]]`
  - `if [ "$a" \< "$b" ]`
  - Note that the < must be escaped in a [ ] construct
- > is greater than, in ASCII alphabetical order
  - `if [[ "$a" > "$b" ]]`
  - `if [ "$a" \> "$b" ]`
  - Note that the > must be escaped in a [ ] construct.
- -z string is null (that is, it has zero length)
- -n string is not *null*

STRING COMPARISON OPERATORS | EXAMPLE

```
#!/bin/bash
#Compares letters $1 and $2

if [ "$1" == "salah" ];
then
	echo "words are the same"

else
	echo "$1 is not equal salah"
fi
```

STRING COMPARISON OPERATORS | EXAMPLE

```
#!/bin/bash

echo "Are you ok?"
read var
if [  "$var"  =  "Y“  ] || [  "$var“  =  "y"  ]
then
        echo "Glad to hear that"
else
         echo “Sorry to hear that"
fi
```

CONDITIONAL STATEMENTS | CASE

```
case word in
  pattern1)
            Statement(s) to be executed if pattern1 matches
             ;;
  pattern2)
            Statement(s) to be executed if pattern2 matches
            ;;
  pattern3)
             Statement(s) to be executed if pattern3 matches
            ;;
  *)
            Default condition to be executed
            ;;
esac
```

CONDITIONAL STATEMENTS | CASE

```
case variable in
value1)
                Command(s)
                ;;
value2)
                Command(s)
                ;;
*)
                Command(s)
                ;;
esac
```

CONDITIONAL STATEMENTS | EXAMPLE

```
#!/bin/bash

echo “enter your favorite fruit: apple or banana or kiwi ”
read fruit

case "$fruit" in
  "apple")
            echo "Apple pie is quite tasty."
            ;;
  "banana")
            echo "I like banana nut bread."
             ;;
  "kiwi")
            echo "New Zealand is famous for kiwi."
            ;;
     *)
            echo “your answer is out of scope”
esac
```

CONDITIONAL STATEMENTS | EXAMPLE

```
#!/bin/bash

echo "Enter the name of a month."
read month

case $month in
        February)
                echo "There are 28/29 days in $month.";;
        April | June | September | November)
                echo "There are 30 days in $month.";;
        January | March | May | July | August | October | December)
                echo "There are 31 days in $month.";;
        *)
                echo "Unknown month. Please check if you entered the correct month name: $month";;
esac
```

CONDITIONAL SELECT| EXAMPLE

```
select choice in Ahmed Adel Tamer Quit
do
case $choice in
        Ahmed)
                echo Ahmed is good boy
                ;;
        Adel)
                echo Adel is the best
                ;;
        Tamer)
                echo Tamer is a bad boy
                ;;
        *)
                echo $REPLY is not one of the choices.
                ;;
esac
done
```

CONDITIONAL SELECT| EXAMPLE

Output:
1)Ahmed
2)Adel
3)Tamer
#? 1
Ahmed is a good boy
1)Ahmed
2)Adel
3)Tamer
#?5
5 is not one of the choices
1)Ahmed
2)Adel
3)Tamer

LOOP STATEMENTS

- **While**
- **Until**
- **For**

LOOP STATEMENTS | WHILE

```
while command
do
                    … command …
done
```

Inline:

```
while command; do  command; done
```

Examples:

```
num=0
while [ $num –lt 10 ]
do
                    echo “hello it’s number: $num”
                    num=$(($num+1))
done
```

LOOP STATEMENTS | WHILE

```
echo "please enter your name:"
read name

while [[    "$name" != +([a-zA-Z])    ]]
do
            echo enter valid input please
            read name
done


echo     “ you entered right data”
```

LOOP STATEMENTS | WHILE

Write a bash script game that asks the executer to guess the number and exit only when he gets the correctly number

LOOP STATEMENTS | UNTIL

```
until command
do
command(s)
done
```

Example:

```
hour=1
until [ $hour –lt 24 ]
do
        case $hour in
                [6-11] ) echo good morning ;;
                12) echo lunch time ;;
                [1-4]) echo work time ;;
                *) echo Good Night ;;
        esac
         hour=$hour+1
done
```

LOOP STATEMENTS | FOR

- It is used to execute commands a finite number of times on a list of items (files/usernames)

```
for (( initializer; condition; step ))
do
        command
done
```

```
for VARIABLE in val1 val2 val3 val4 val5 .. N
do
        command1
        command2
        commandN
done
```

LOOP STATEMENTS | FOR | EXAMPLE

Example:

```
for name in mona ahmed maha
do
        echo hi $name
done
```

Example:

```
for ((a=1; a <= 5 ; a++))
do
  echo "Welcome $a times."
done
```

Example:

```
for i in $(seq 5)
do
 echo "Welcome $i times"
done
```

LOOP STATEMENTS | FOR | EXAMPLE

Example:

```
for i in {1..5}
do
  echo "Welcome $i times"
done
```

Example:

```
for i in {0..10..2}
do
   echo "Welcome $i times"
done
```

Example:

```
BOOKS=('Python' 'Java' 'Bash' 'Go' 'Rubyonrails')
for book in "${BOOKS[@]}";
do
 echo "Book is: $book"
done
```

LOOP CONTROL

- Used with loops to better manage their conditions:

- break: Stop running the entire loop.

- continue: If a condition is met, break out the current iteration of loop.
  The loops keeps running

LOOP CONTROL | BREAK

```
#!/bin/bash
#break statement
counter=1
while [ $counter -le 10 ];
do
        echo $counter
        ((counter++))
        if [ $counter == 5 ];
        then
        echo "condition met"
        break
        echo "after break control"
        fi
echo "loop keeps going"
done

echo "loop exited"
echo "counter equals $counter"
```

LOOP CONTROL | CONTINUE

```
#!/bin/bash
#The continue statement
counter=1
while [ $counter -le 10 ];
do
        echo $counter
        ((counter++))
        if [ $counter == 5 ];
        then
        echo "condition met"
        continue
        echo "after continue"
        fi
echo "loop keeps going"
done

echo "loop exited"
echo "counter equals $counter"
```

COMMAND SUBSTITUTION

- Commands can be placed in the syntax of other commands.
- Commands are surrounded by backticks (`).
- Commands can be useful in scripts

Example:

```
#!/bin/bash

echo “Hello $USER!”
echo “Today’s date is : `date`”
```

FUNCTION

- A Function is a collection of commands that can be executed simply by entering the function's name.

- Functions must be defined before it is used.

- Local variables can be declared in the function using typeset.

- Format:
- function function-name {commands; commands;}

- function_name

- Return value $?

FUNCTION | EXAMOLE

```
function F1()
{
   retval='I like programming‘

}

retval='I hate bash‘

echo $retval

F1

echo $retval
```

FUNCTION | EXAMPE

```
#! /bin/bash

function increment {

      (( sum = $1 + 1 ))

            return $sum
}

echo  "The sum is:"
increment 5
out=$?
Ret_val=$(increment 5)
echo "$out“
echo  “$Ret_val”
```

LET'S GET CONNECTED!

www.linkedin.com/in/islam-salah-25b4a4a4

010 68-280-165