Part 1 - Introduction to Shell Scripting

About Shell -

- It is an interface between user and system
- Shell the executes the user’s input and displays the output
- Shell is an environment where we can execute

❑ Commands

❑ Programs

❑ Shell Scripts

Shell Scripting -

- It is a group of unix commands and shell keywords
- These are executed in Sequence of order
- These are not complied but interpreted by O.S
- It is always advisable to use #sign (comment) to describe about the shell

Purpose of Shell Scripting -

- To handle text files
- Create new commands
- Automate the system administration tasks
- To perform the Repetitive tasks .. etc

Various Types of Shells -

SHELL NAME	BY	PROMPT	INTERPRETER NAME	DEFAULT SHELL
Bourne shell	Stephen bourne	$	sh	Sco-Unix, Solaris, HP- UX
Korn shell	David korn	$	ksh	IBM AIX
C shell	Bill joy	%	csh	IRIX
Bash Shell	Stephen Bourne	$	bash	Linux
Z shell	Paul	$	zsh	--

Q. How to know what shell scripting supported by system ?

- # cat /etc/shells
- Execute the command, it will show all the shell scripts supported by system

Q. How to shift to various shells ?

```
# ksh

# sh
```

Q. How to check current Child Shell or Subshell ?

```
# echo $0
```

Q. How to exit from a shell ?

```
# exit
```

GENERIC WORKFLOW OF BASH SCRIPTING

Step 1 - Create a script file with .sh extension

Note : Extension .sh is not mandatory, however it is recommend to use standard conventions

Step 2 - Write the script content

Step 3 - Change the permission to script file

Step 4 - Execute the Script file

FILE PERMISSIONS ON SCRIPT FILE

- Make sure that script does have read and execute permission at a user level as a convention
- However, x permission is mandatory to script file
- Use chmod command to provide read and execute permissions

Example :

```
# chmod u+rx <script_filename.sh>
```

EXECUTE SCRIPT FILE

Method 1 - ./<script_filename.sh>

Here, script file needs to be at current path and generally used for relative path execution of script file

Method 2 - # sh <script_filename.sh>

Here, this command can be applied when the file is at current path or located at Absolute path location and generally used for Absolute Path execution of script file

EXERCISE ON CREATING A SAMPLE SCRIPT FILE -

- Create a file called sample.sh
- Then add the below content

```
date
ls-l
```

- Save the file
- Change the permissions
- Execute the script file

Expected Interview Questions -

Q. Purpose of Script file ?

Q. How to execute script file ?

Q. Do we need to change any permissions to execute script file ?

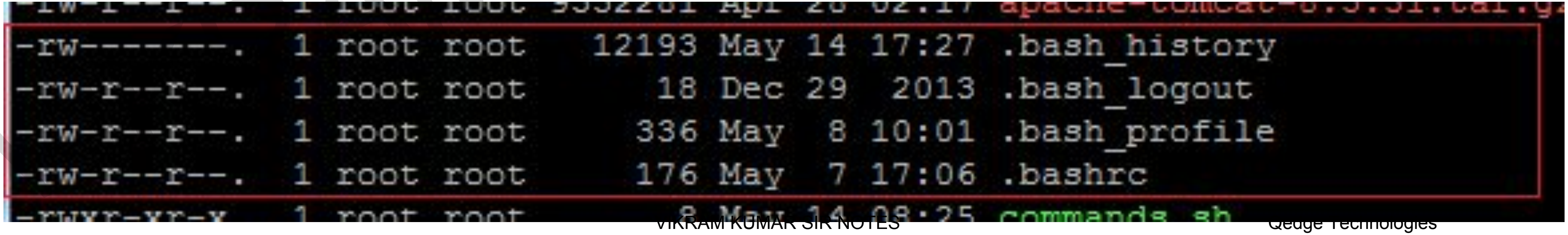
THANK YOU

Part 2 - Understanding of Profile scripts

PROFILE SCRIPTS - at startup

- **.bash_profile** is a predefined file that executes when the user login into system
- .bash_profile is a hidden file
- We can put code of script that needs to be executed at startup of user login
- .bash_profile impacts only current logged in user only
- Any changes that are made to .bash_profile need to execute below command to impact changes

# source .bash_profile



```
-rw-r--r--.  1 root root 9552281 Apr 28 02:17 apache-tomcat-8.5.31.tar.gz
-rw-------.  1 root root   12193 May 14 17:27 .bash_history
-rw-r--r--.  1 root root      18 Dec 29  2013 .bash_logout
-rw-r--r--.  1 root root     336 May  8 10:01 .bash_profile
-rw-r--r--.  1 root root     176 May  7 17:06 .bashrc
-rwxr-xr-x.  1 root root       8 May 14 08:25 commands.sh
```

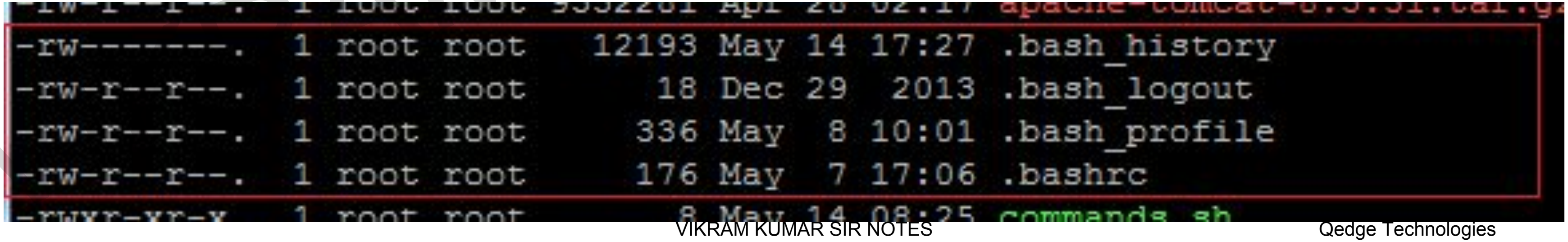
Exercise Activity -

Write a sample code for startup messages in .bash_profile -

PROFILE SCRIPTS - at logout

- **.bash_logout** is a predefined file that executes when user logs out of system
- .bash_logout is a hidden file
- We can put code of script that needs to be executed when user logs out of system
- .bash_logout impacts only current logged in user only
- Any changes that are made to .bash_logout need to execute below command to impact below changes

# source .bash_logout



```
-rw-------.  1 root root   12193 May 14 17:27 .bash_history
-rw-r--r--.  1 root root      18 Dec 29  2013 .bash_logout
-rw-r--r--.  1 root root     336 May  8 10:01 .bash_profile
-rw-r--r--.  1 root root     176 May  7 17:06 .bashrc
-rwxr-xr-x   1 root root       8 May 14 08:25 commands.sh
```

Exercise Activity -

Write a sample code for logout messages in .bash_logout -

- Use below code

Echo “ Bye. Have a Good Day”
Sleep 2

THANK YOU

Part 3 - Bash Scripting - Shebang, comment, variables

Use of Shebang or hashbang

Step 1 : find the location of bash shell using below command

```
# which bash
```

Note - make a note of the location of bash

Step 2 : create a file with .sh extension to create a script file called helloworld.sh

Write the below code

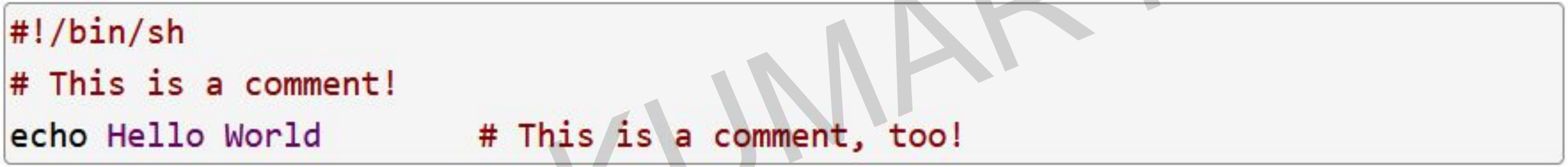
```
#! /usr/bin/bash
Echo “Hello World”
```

Step 3 - save and exit the script file, then give execute permission to script file

Step 4 - execute the script file

How to comment in Bash Scripting

- To comment in Bash Scripting use #!

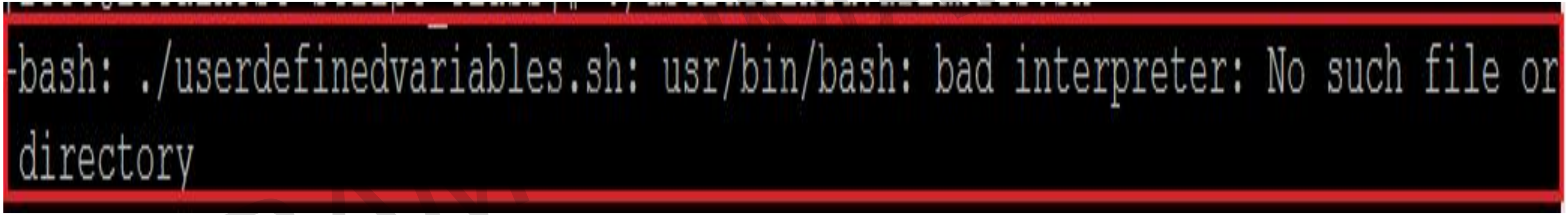


```
#!/bin/sh
# This is a comment!
echo Hello World        # This is a comment, too!
```

Case Study - Bad interpreter error messages

- We get below error message when shebang information in script file is not correct

- When the path of shell location is incorrect



```
-bash: ./userdefinedvariables.sh: usr/bin/bash: bad interpreter: No such file or
directory
```

Activity :

- Write a script to perform commenting in a script file

VARIABLES

- Variables store the any type data in them
- There are no data types in Shell Scripting
- The value of variable can be assigned inbuilt or can be assigned at the execution time
- There are two types of variables -

1. User Defined Variables

2. System Defined Variables (Environment Variables)

Difference between System Variables and User Defined Variables

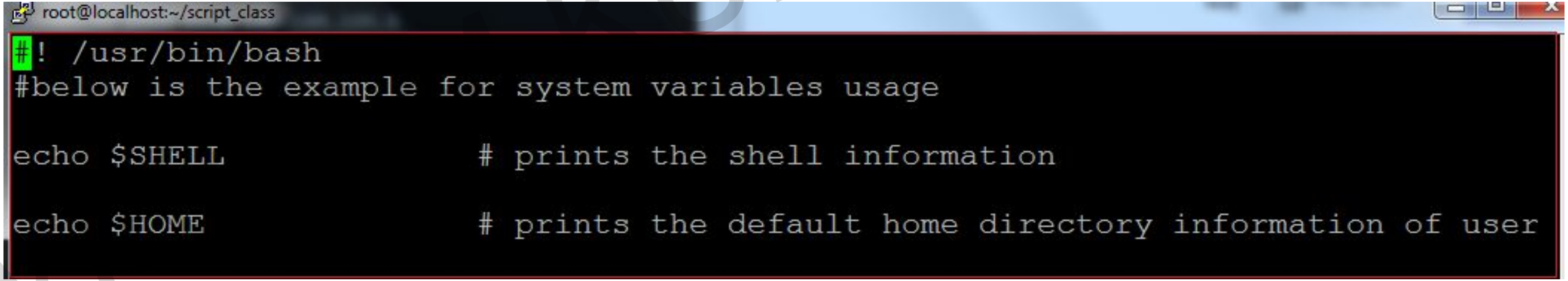
System Variables	User Defined Variables
● Created and maintained by Linux system	● Created by user
● These are used by system	● Mostly created in lower format but also can use CAPS
● These are mostly in CAPS format	● However, as a convention need to use lower case. ● * user defined variables are case sensitive

Examples of System variables

- echo $SHELL
- echo $HOME

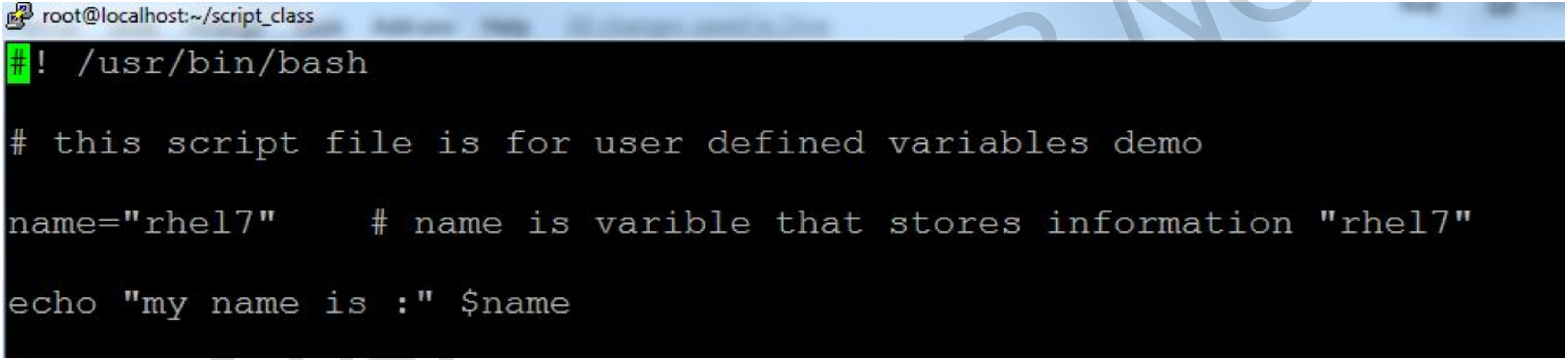
.. etc

Example of system variables -



```
#! /usr/bin/bash
#below is the example for system variables usage

echo $SHELL                # prints the shell information

echo $HOME                 # prints the default home directory information of user
```

Examples of User Defined variables



```
#! /usr/bin/bash

# this script file is for user defined variables demo

name="rhel7"    # name is varible that stores information "rhel7"

echo "my name is :" $name
```

Naming Conventions to declare a variable -

- The name of a variable can contain only be

❏ letters (a to z or A to Z)

❏ numbers (0 to 9)

❏ underscore character (_)

❏ Variable name cannot start with number

Syntax for Variable Declaration :

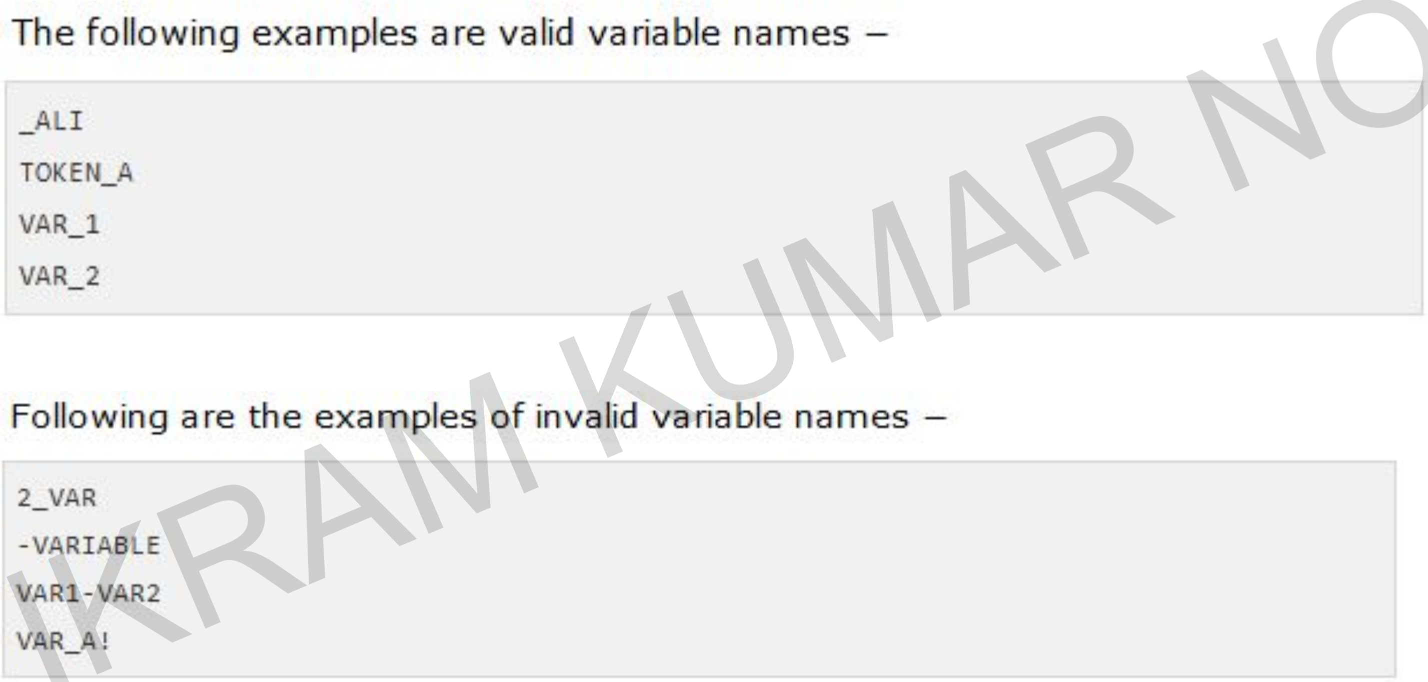
variable=value

** Note that there must be no spaces around the "=" sign

VAR=value works

VAR = value doesn't work

Case study on declaration of Valid and Invalid Variables



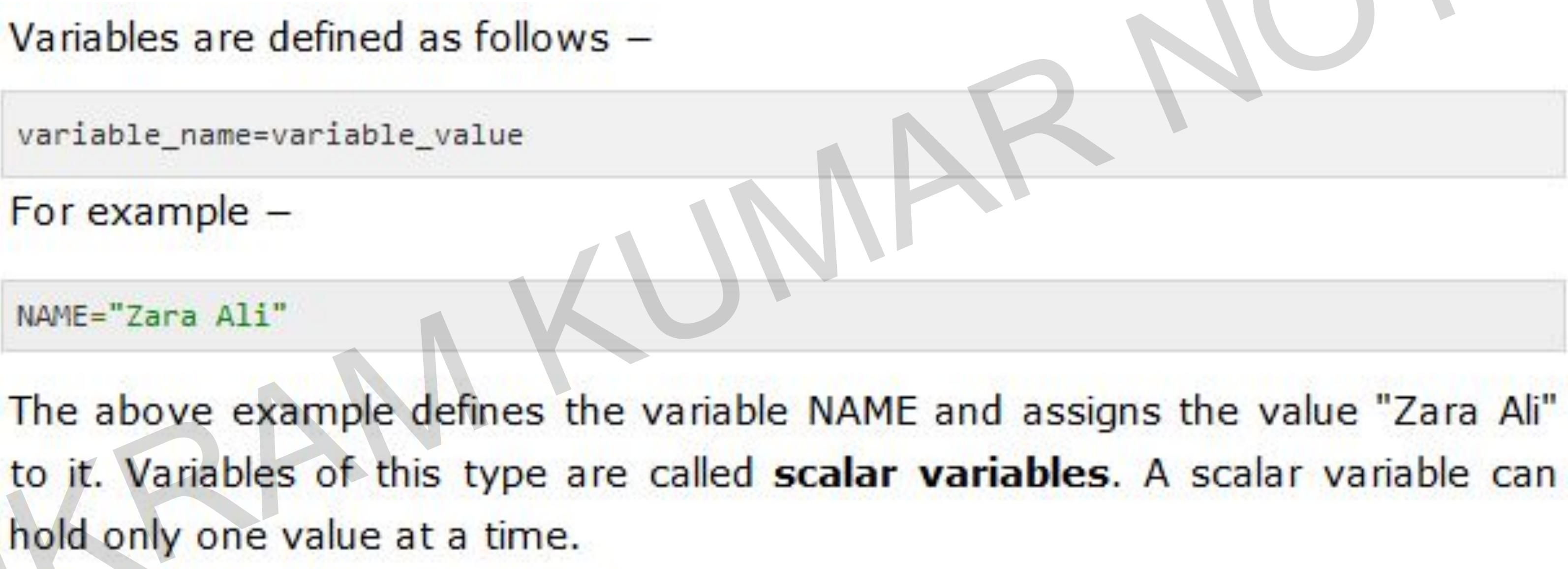
The following examples are valid variable names −

```
_ALI
TOKEN_A
VAR_1
VAR_2
```

Following are the examples of invalid variable names −

```
2_VAR
-VARIABLE
VAR1-VAR2
VAR_A!
```

SCALAR VARIABLES -



Variables are defined as follows –

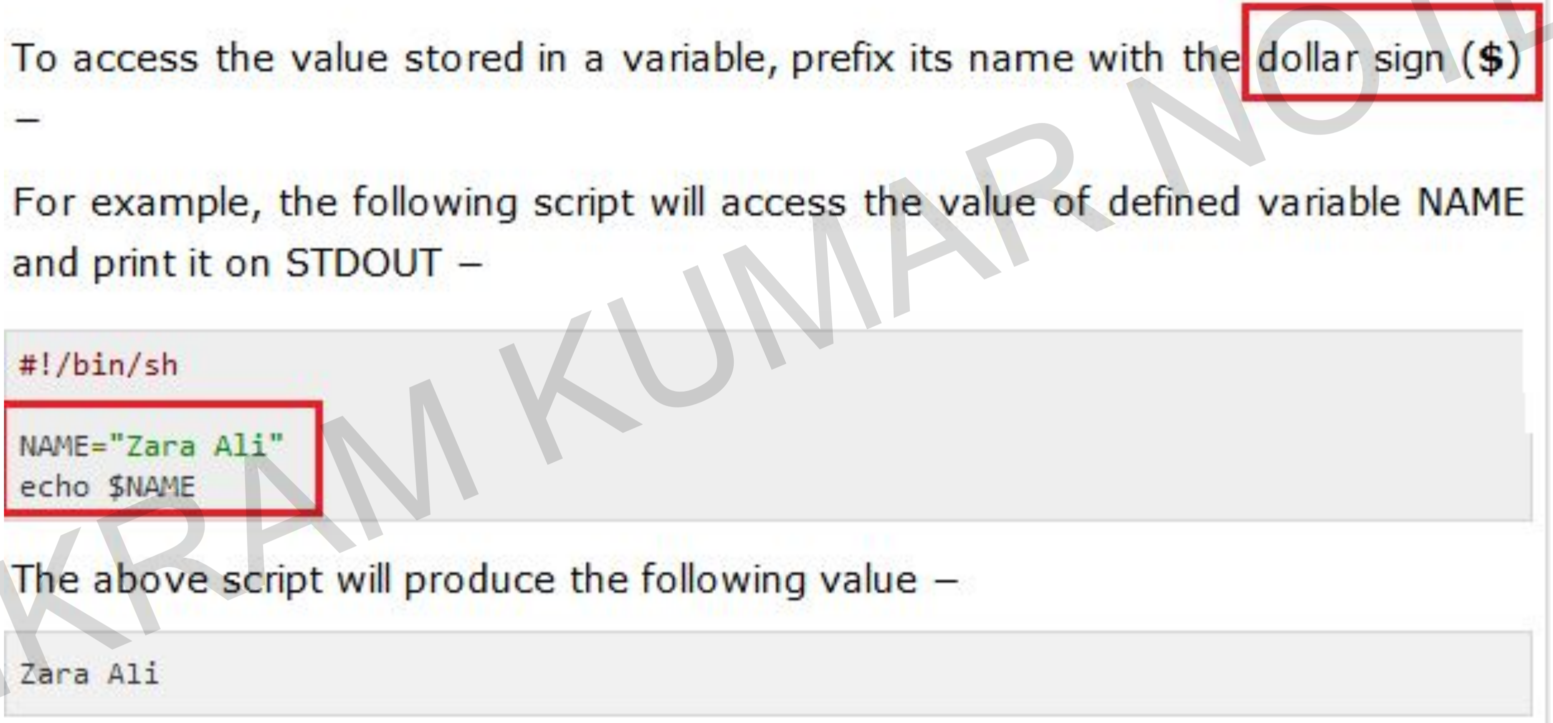
```
variable_name=variable_value
```

For example –

```
NAME="Zara Ali"
```

The above example defines the variable NAME and assigns the value "Zara Ali" to it. Variables of this type are called **scalar variables**. A scalar variable can hold only one value at a time.

ACCESSING VARIABLES



To access the value stored in a variable, prefix its name with the dollar sign ($) –

For example, the following script will access the value of defined variable NAME and print it on STDOUT –

```
#!/bin/sh

NAME="Zara Ali"
echo $NAME
```

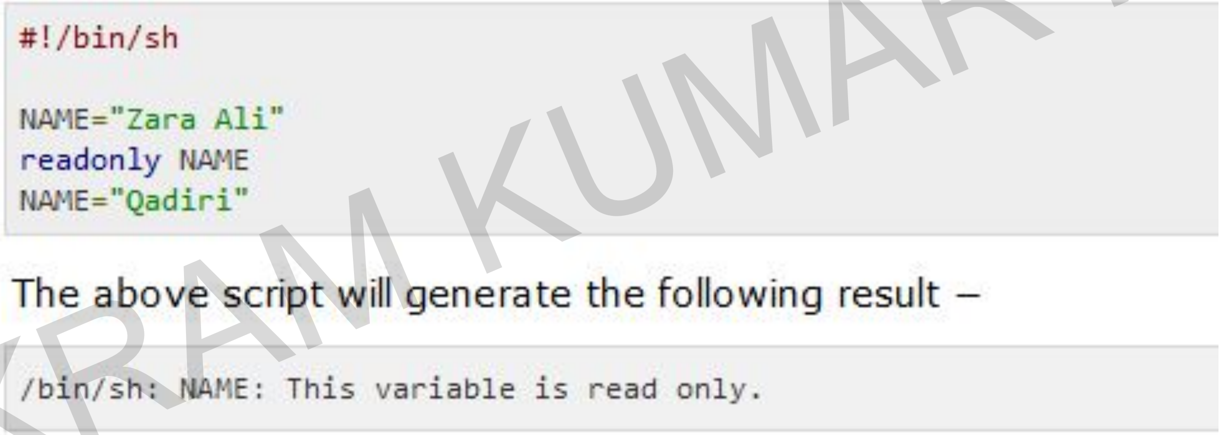
The above script will produce the following value –

```
Zara Ali
```

Read Only Variables -

- Once a variable is set to READ ONLY Mode, the value of variable cannot be changed

Example :



```
#!/bin/sh

NAME="Zara Ali"
readonly NAME
NAME="Qadiri"
```

The above script will generate the following result –

```
/bin/sh: NAME: This variable is read only.
```

Expected Interview Questions -

Q. My script fails to execute, it reports error bad interpreter what could be the issue?

Q. How do you access the variables ?

Q. Difference between System Defined Variables and User Defined Variables ?

THANK YOU

Part 4 - Various approaches to read input, arguments, arrays

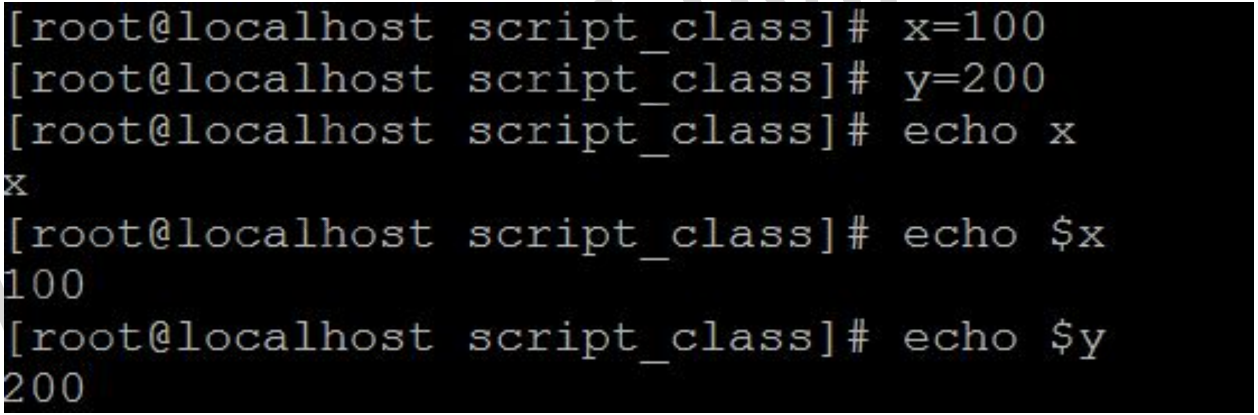
USER DEFINED VARIABLES -

- User defined variables are classified under three types

1. Local Variables
2. Constant Variables
3. Global Variables

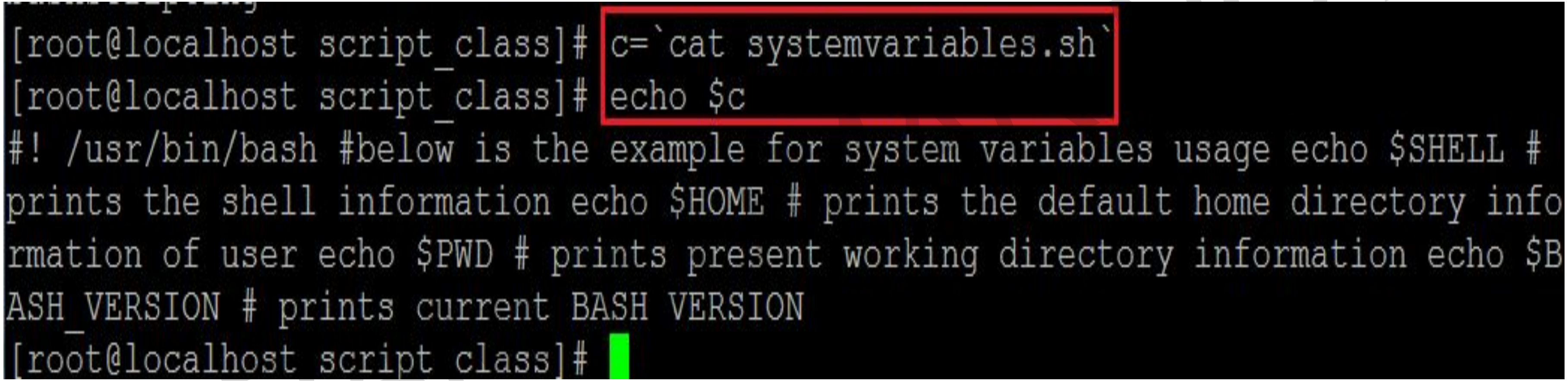
Local Variables

- These type of variables are present within the current instance of the shell
- It is not available to programs that are started by the shell
- They are set at the command prompt



```
[root@localhost script_class]# x=100
[root@localhost script_class]# y=200
[root@localhost script_class]# echo x
x
[root@localhost script_class]# echo $x
100
[root@localhost script_class]# echo $y
200
```

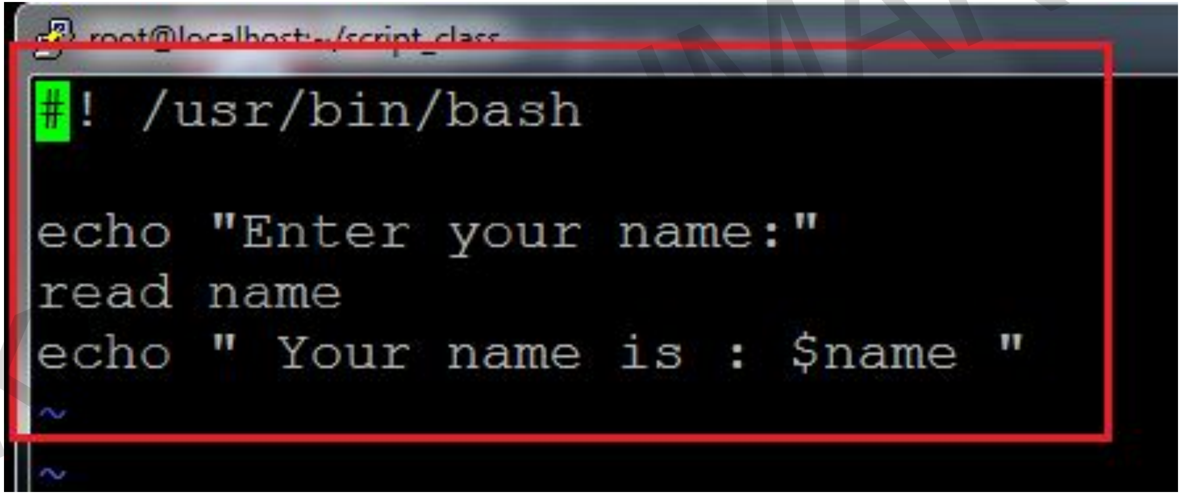
Example - local variables assign commands



```
[root@localhost script_class]# c=`cat systemvariables.sh`
[root@localhost script_class]# echo $c
#! /usr/bin/bash #below is the example for system variables usage echo $SHELL #
prints the shell information echo $HOME # prints the default home directory info
rmation of user echo $PWD # prints present working directory information echo $B
ASH_VERSION # prints current BASH VERSION
[root@localhost script_class]#
```

READING INPUT FROM PROMPT

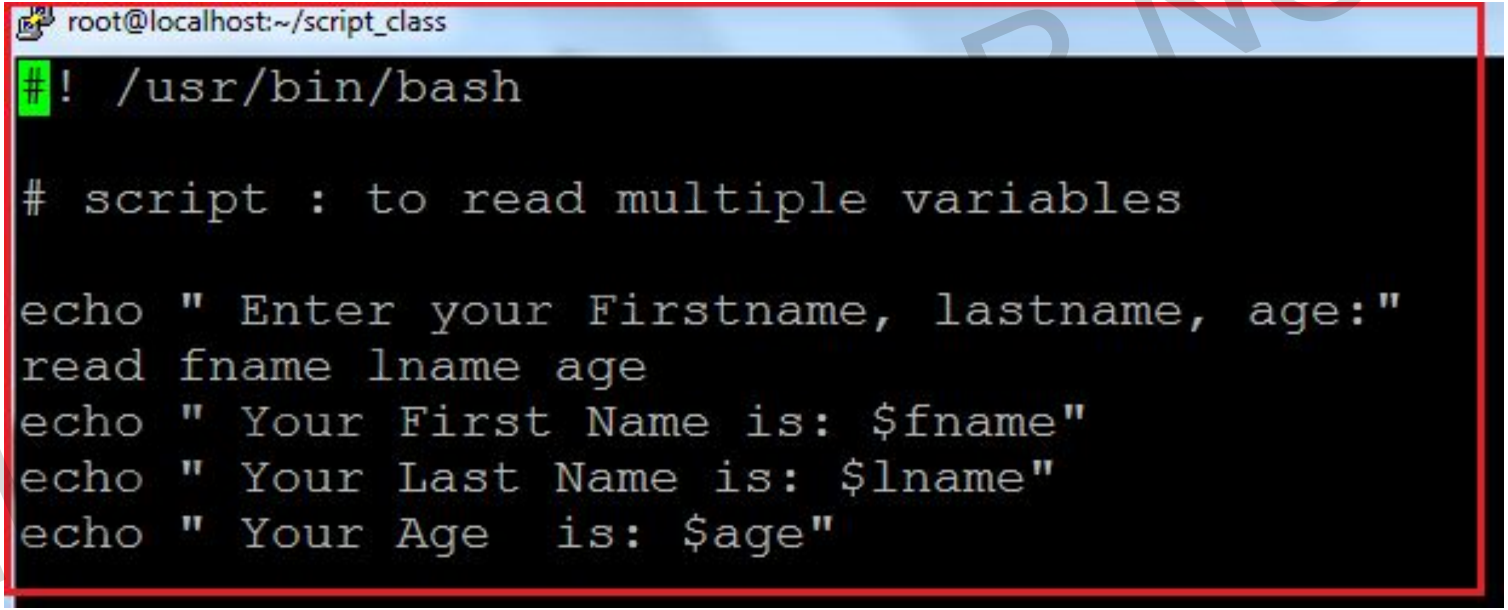
- To read the input from standard input need to use read Command
- Below is the example for reading single variable



```
#! /usr/bin/bash

echo "Enter your name:"
read name
echo " Your name is : $name "
~
~
```

Reading Multiple Variables

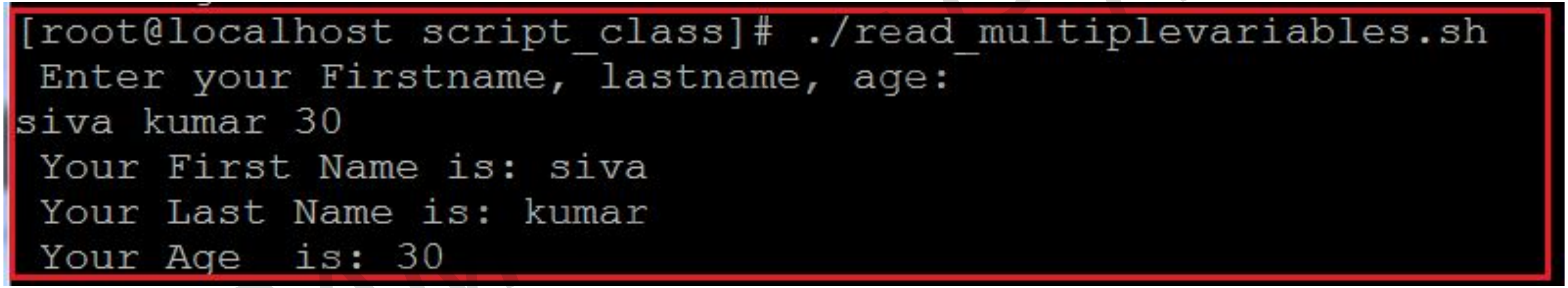
- Enter the data for multiple variables at runtime using spaces between them



root@localhost:~/script_class

```
#! /usr/bin/bash

# script : to read multiple variables

echo " Enter your Firstname, lastname, age:"
read fname lname age
echo " Your First Name is: $fname"
echo " Your Last Name is: $lname"
echo " Your Age  is: $age"
```

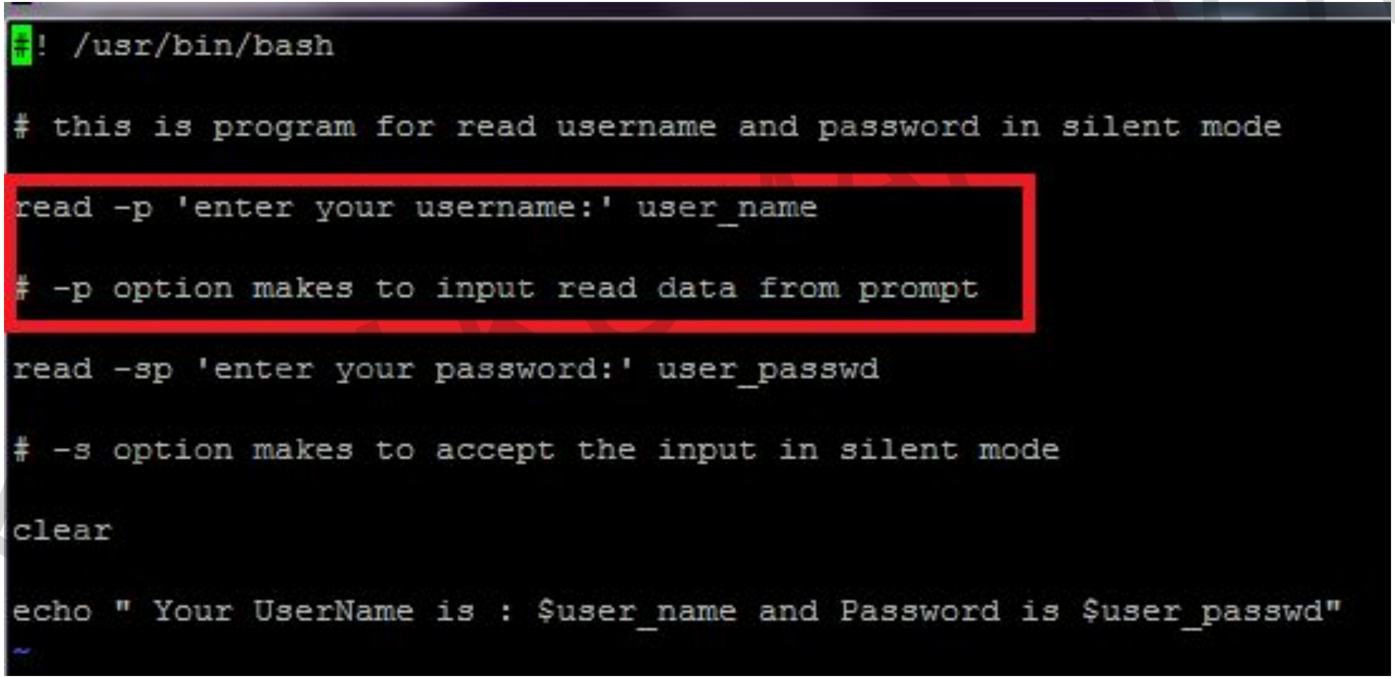
output



```
[root@localhost script_class]# ./read_multiplevariables.sh
 Enter your Firstname, lastname, age:
siva kumar 30
 Your First Name is: siva
 Your Last Name is: kumar
 Your Age  is: 30
```

Reading Variables from input prompt

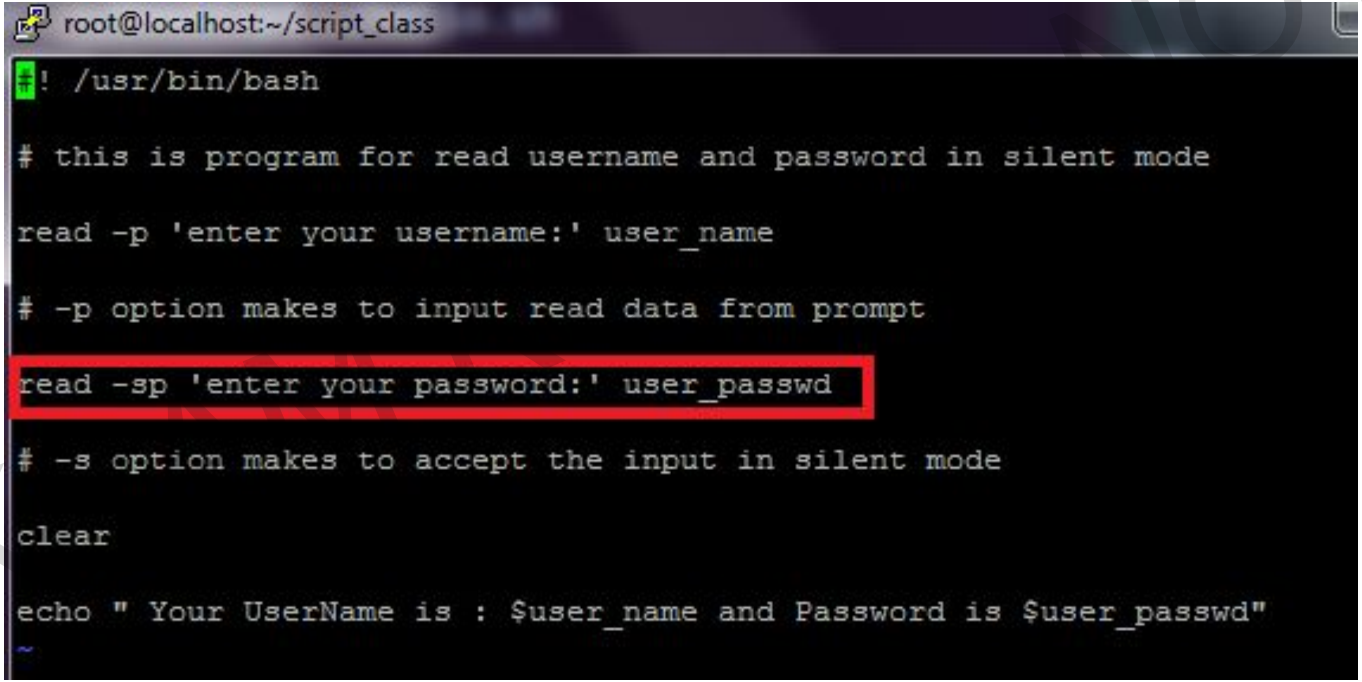
- Use -p option as below



```
#! /usr/bin/bash

# this is program for read username and password in silent mode

read -p 'enter your username:' user_name

# -p option makes to input read data from prompt

read -sp 'enter your password:' user_passwd

# -s option makes to accept the input in silent mode

clear

echo " Your UserName is : $user_name and Password is $user_passwd"
~
```

Reading Variables in Silent Mode

- Use -s option as below



```
#! /usr/bin/bash

# this is program for read username and password in silent mode

read -p 'enter your username:' user_name

# -p option makes to input read data from prompt

read -sp 'enter your password:' user_passwd

# -s option makes to accept the input in silent mode

clear

echo " Your UserName is : $user_name and Password is $user_passwd"
~
```

Reading Variables in form of Array

- While reading an array use the option -a

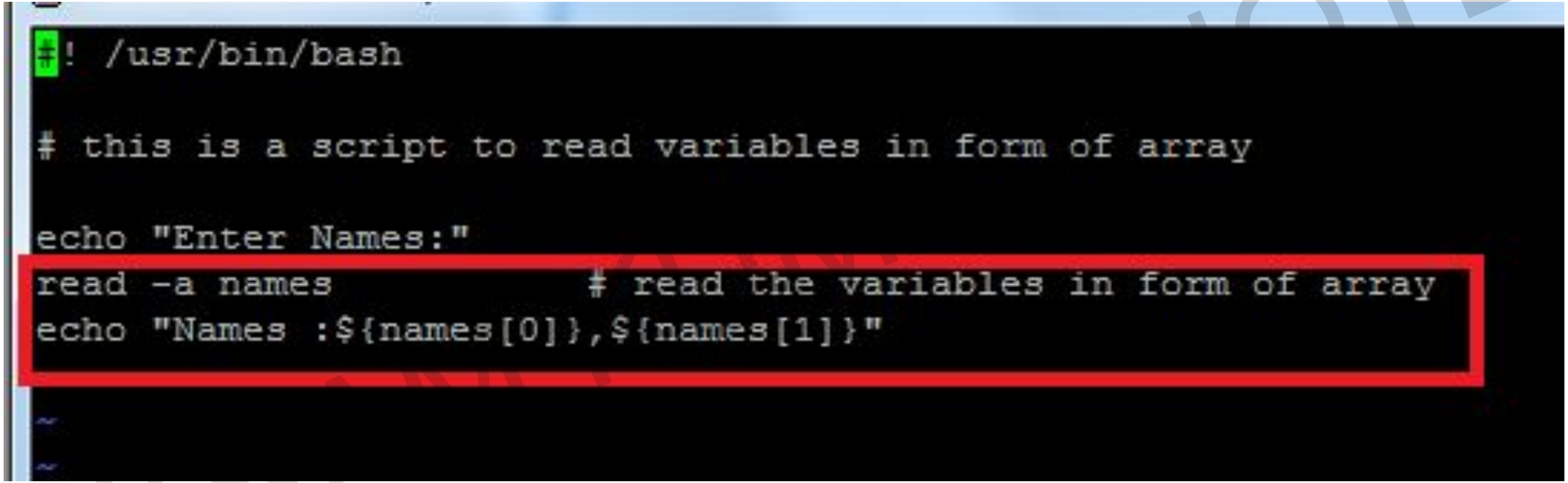
Example - read -a ....

- While print the data from an array, call the array element using index value
- Index value starts from Zero

Syntax - ${array_variable_name[index_value]}

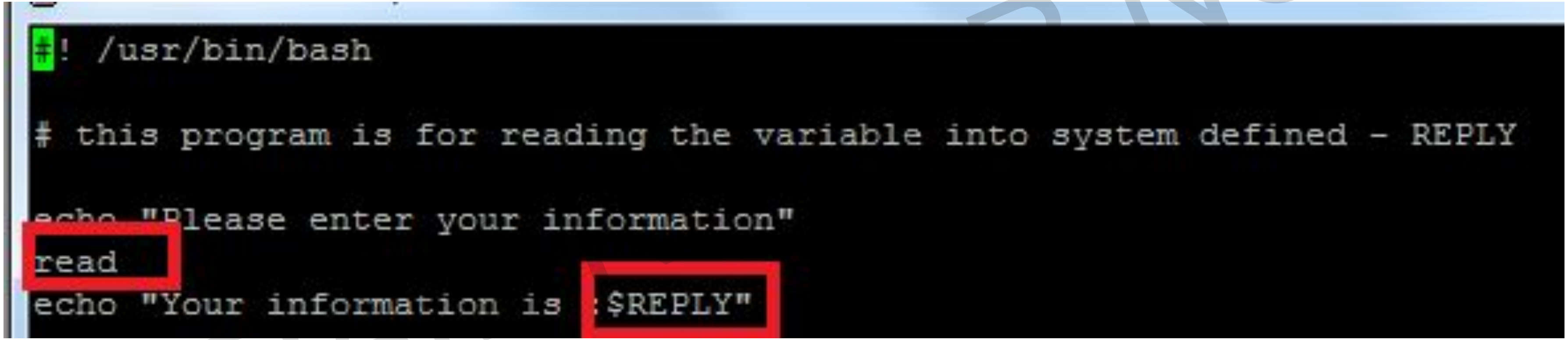
Example - echo “names are:” ${names[0]}

Example - reading the variable inform of array



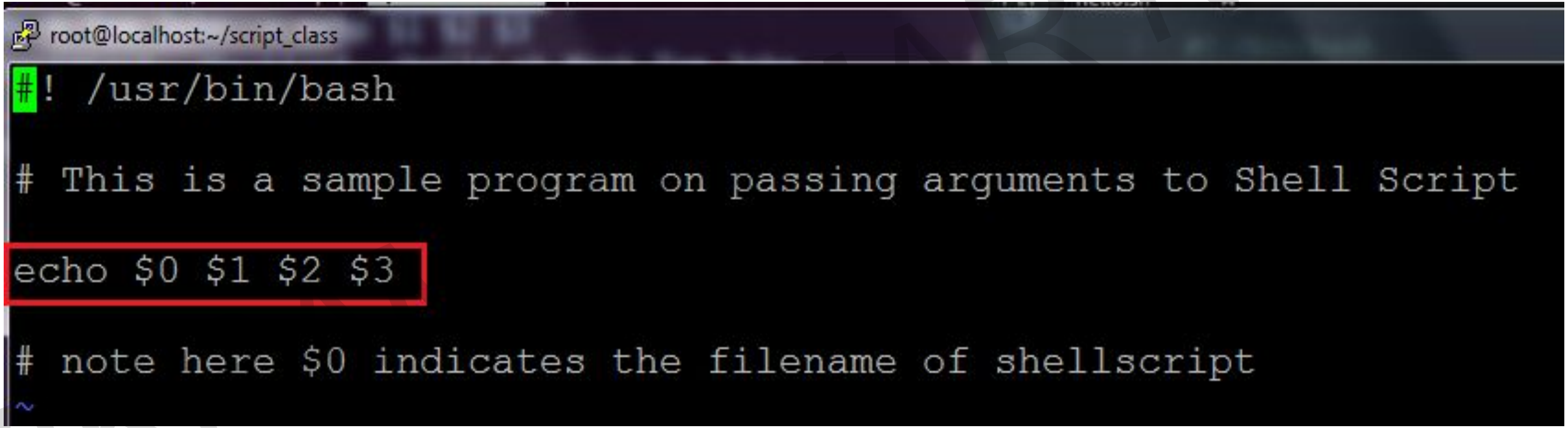
```
#! /usr/bin/bash

# this is a script to read variables in form of array

echo "Enter Names:"
read -a names                # read the variables in form of array
echo "Names :${names[0]},${names[1]}"

~
~
```

Reading input into System Defined Variables- REPLY



```
#! /usr/bin/bash

# this program is for reading the variable into system defined - REPLY

echo "Please enter your information"
read
echo "Your information is :$REPLY"
```

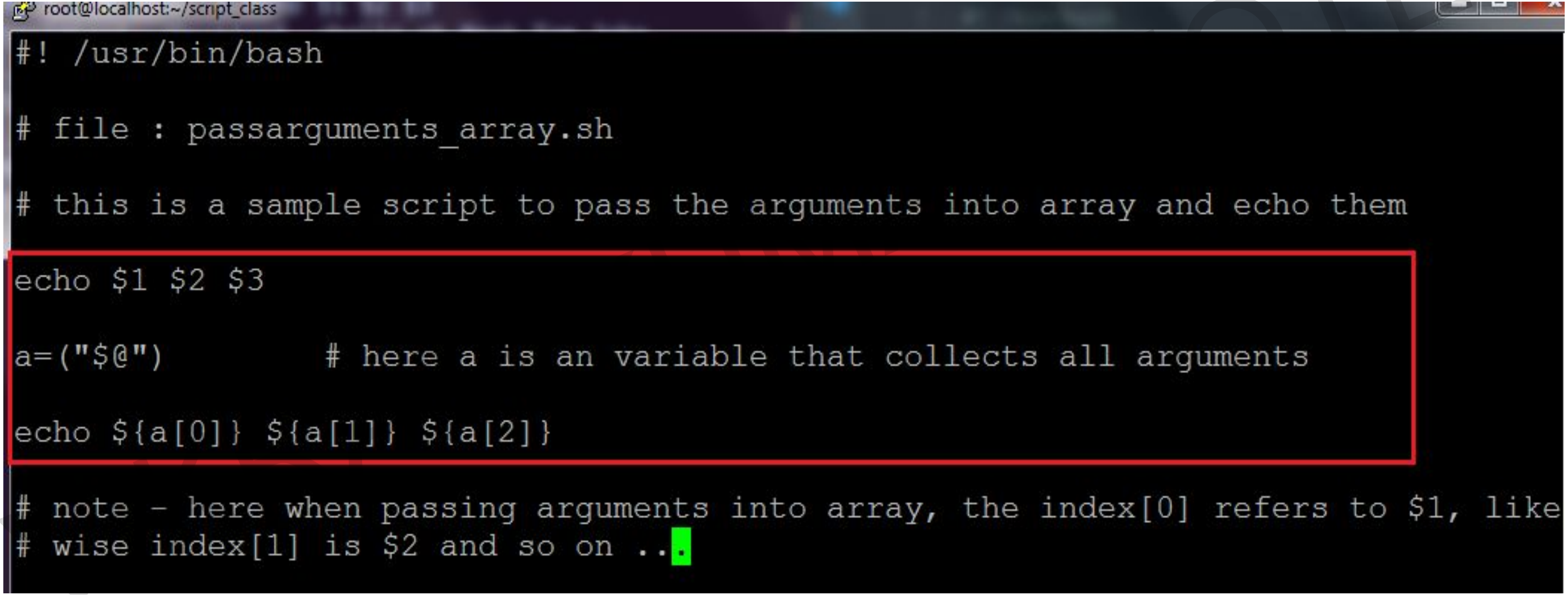
Passing Arguments to Shell Script



root@localhost:~/script_class

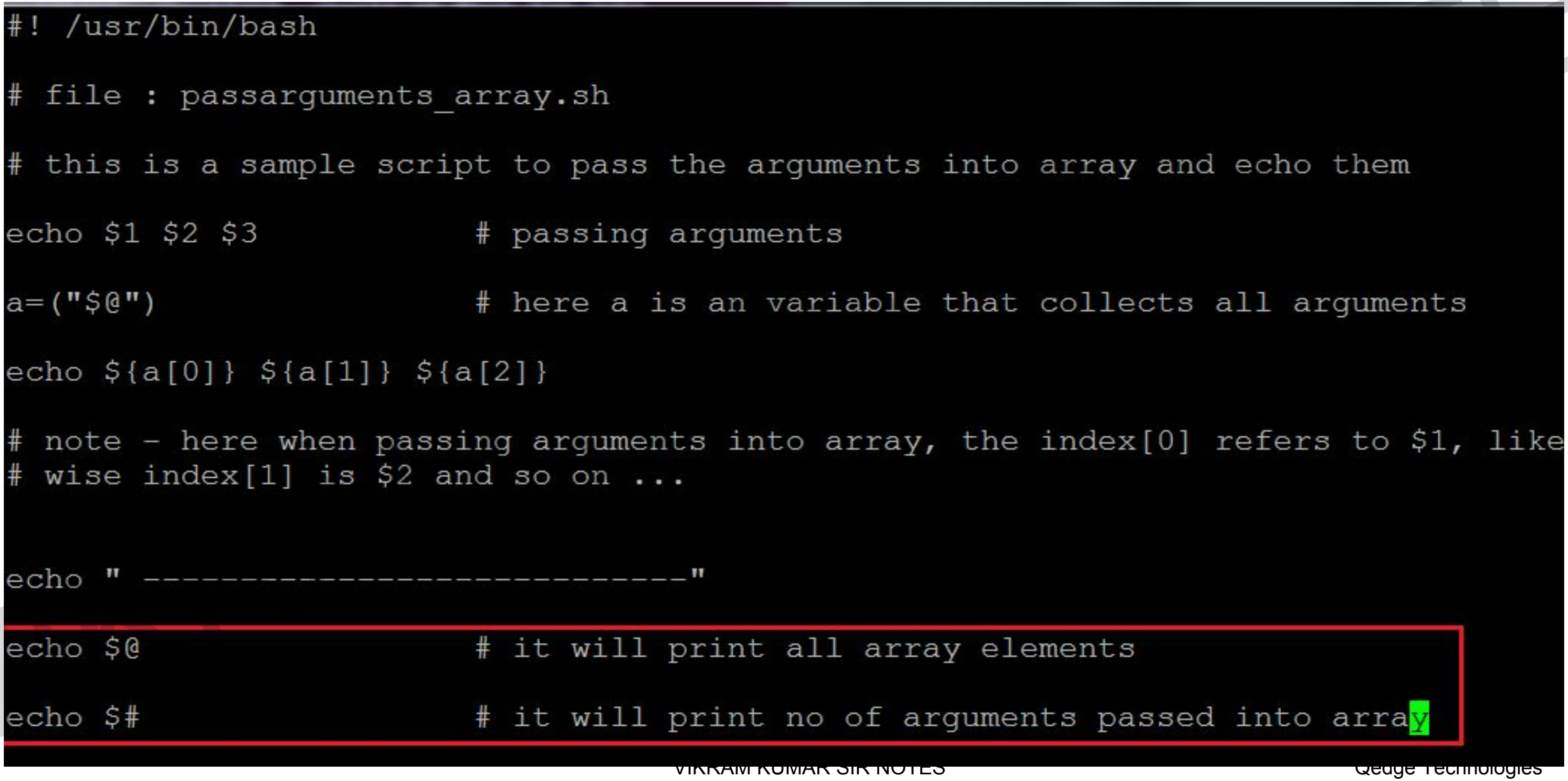
```
#! /usr/bin/bash

# This is a sample program on passing arguments to Shell Script

echo $0 $1 $2 $3

# note here $0 indicates the filename of shellscript
~
```

Passing arguments into Array -



```
#! /usr/bin/bash

# file : passarguments_array.sh

# this is a sample script to pass the arguments into array and echo them

echo $1 $2 $3

a=("$@")          # here a is an variable that collects all arguments

echo ${a[0]} ${a[1]} ${a[2]}

# note - here when passing arguments into array, the index[0] refers to $1, like
# wise index[1] is $2 and so on ...
```

Print all array arguments and count no. of array arguments



```
#! /usr/bin/bash

# file : passarguments_array.sh

# this is a sample script to pass the arguments into array and echo them

echo $1 $2 $3                # passing arguments

a=("$@")                     # here a is an variable that collects all arguments

echo ${a[0]} ${a[1]} ${a[2]}

# note - here when passing arguments into array, the index[0] refers to $1, like
# wise index[1] is $2 and so on ...


echo " ----------------------------"

echo $@                      # it will print all array elements

echo $#                      # it will print no of arguments passed into array
```

Expected Interview Questions -

Q. How to read the variables in silent mode ?

Q. How to pass arguments to shell script ?

Q. what is use echo $@

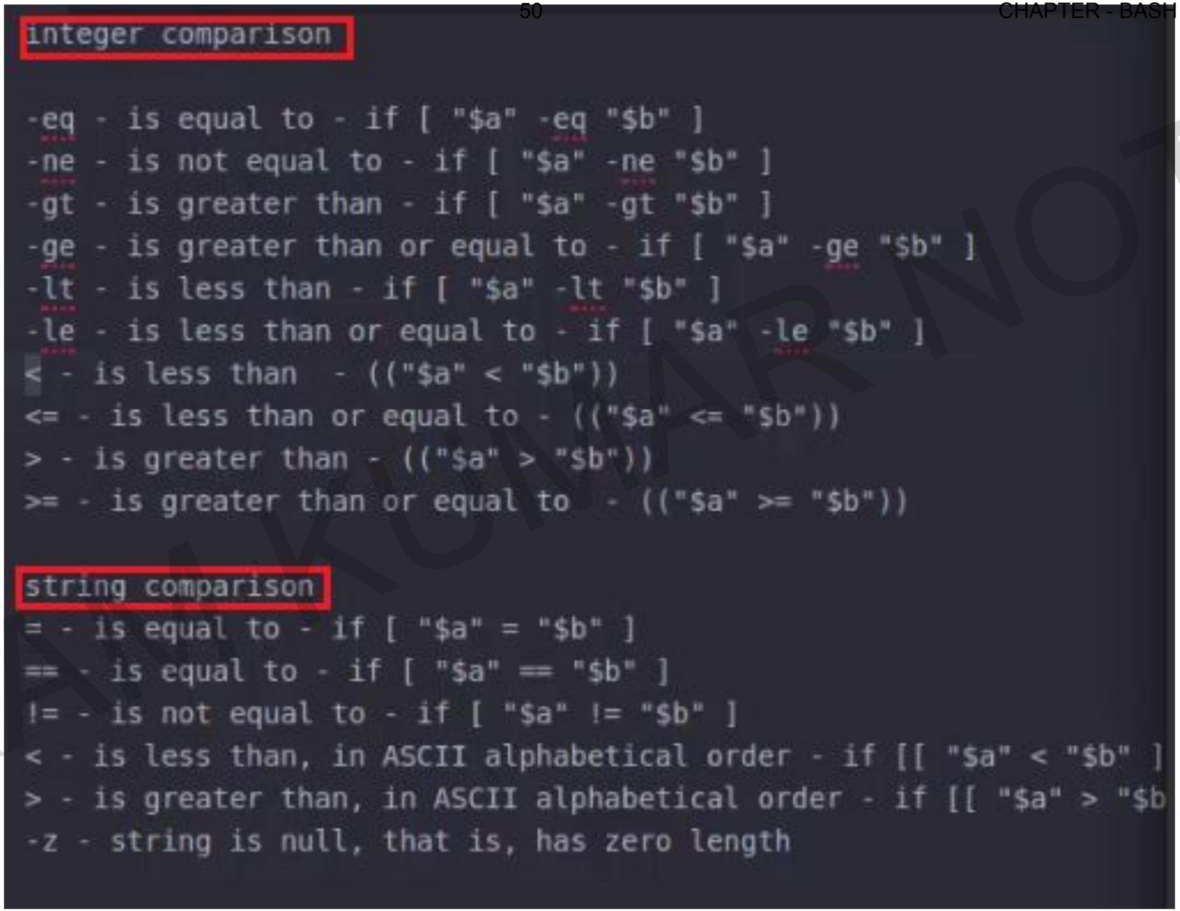
Q. What is use of echo $#

Q. Difference between passing arguments to shell script and arguments to array

THANK YOU

Part 5 - if, if-else, case, file operations

Comparison



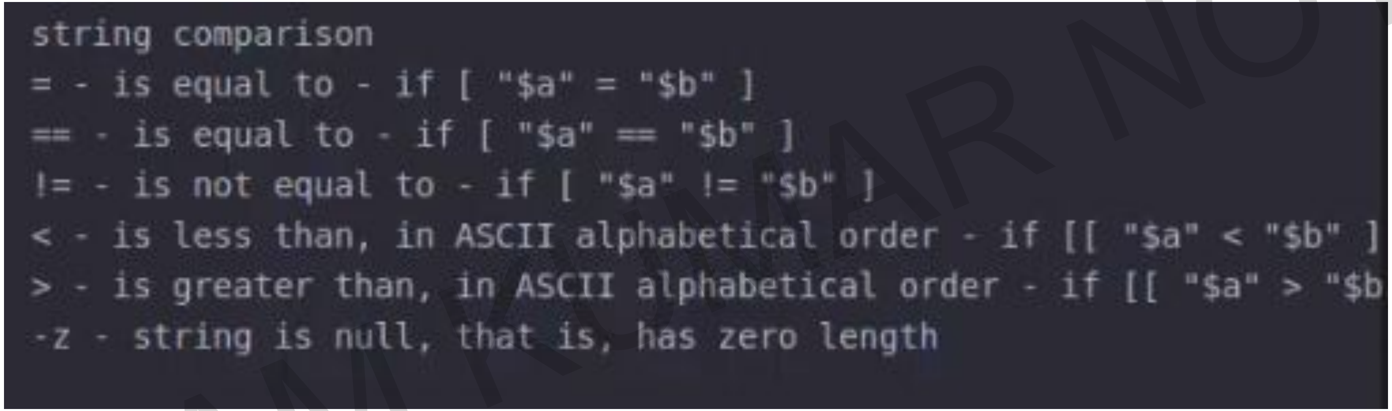
```
integer comparison

-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  - (("$a" < "$b"))
<= - is less than or equal to - (("$a" <= "$b"))
> - is greater than - (("$a" > "$b"))
>= - is greater than or equal to  - (("$a" >= "$b"))

string comparison
= - is equal to - if [ "$a" = "$b" ]
== - is equal to - if [ "$a" == "$b" ]
!= - is not equal to - if [ "$a" != "$b" ]
< - is less than, in ASCII alphabetical order - if [[ "$a" < "$b" ]
> - is greater than, in ASCII alphabetical order - if [[ "$a" > "$b
-z - string is null, that is, has zero length
```

Integer Comparisons -

-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	if (("$a" < "$b"))
<=	is less than or equal to	if (("$a" <= "$b"))
>	is greater than	if (("$a" > "$b"))
>=	is greater than or equal to	if (("$a" >= "$b"))

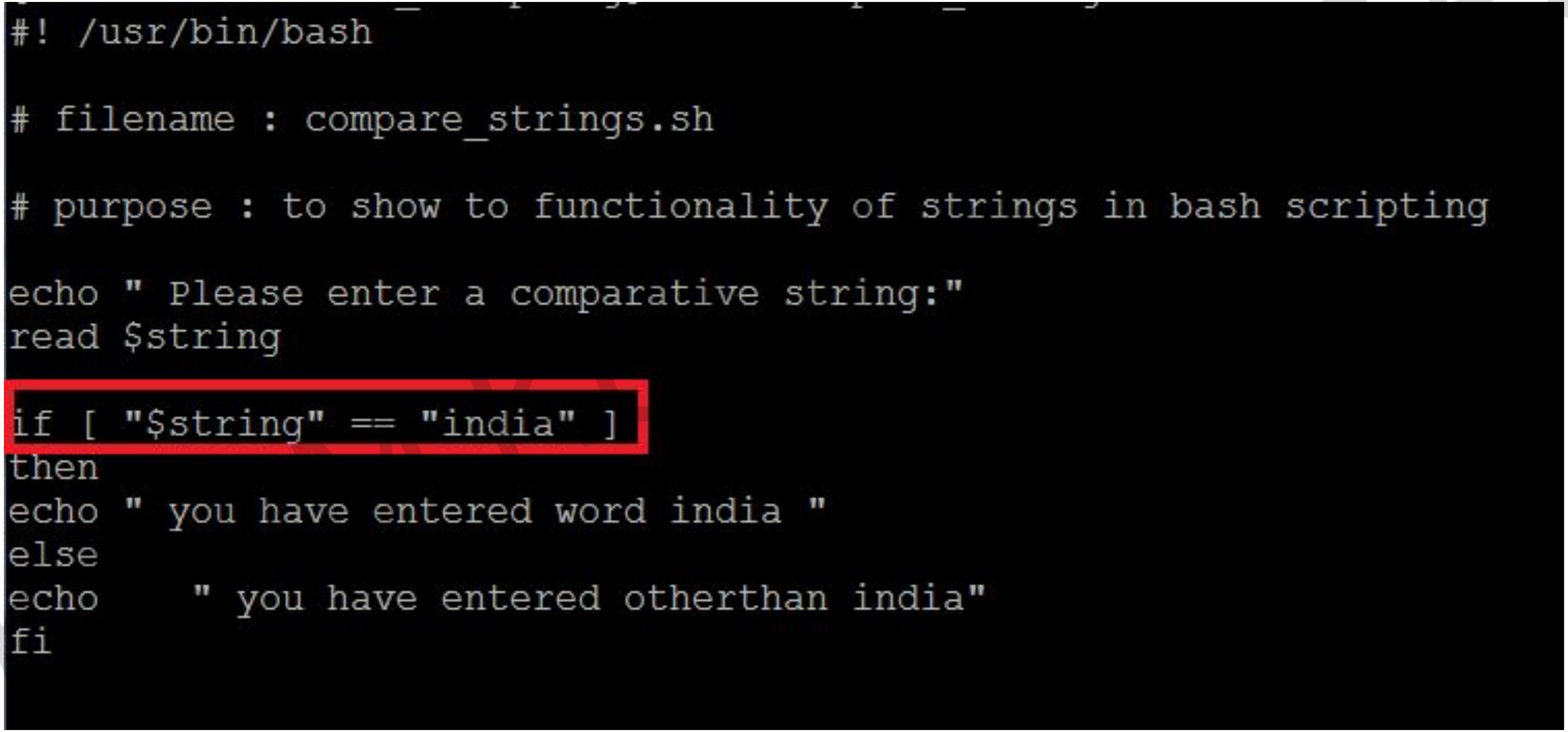
String Comparisons -



```
string comparison
= - is equal to - if [ "$a" = "$b" ]
== - is equal to - if [ "$a" == "$b" ]
!= - is not equal to - if [ "$a" != "$b" ]
< - is less than, in ASCII alphabetical order - if [[ "$a" < "$b" ]
> - is greater than, in ASCII alphabetical order - if [[ "$a" > "$b
-z - string is null, that is, has zero length
```

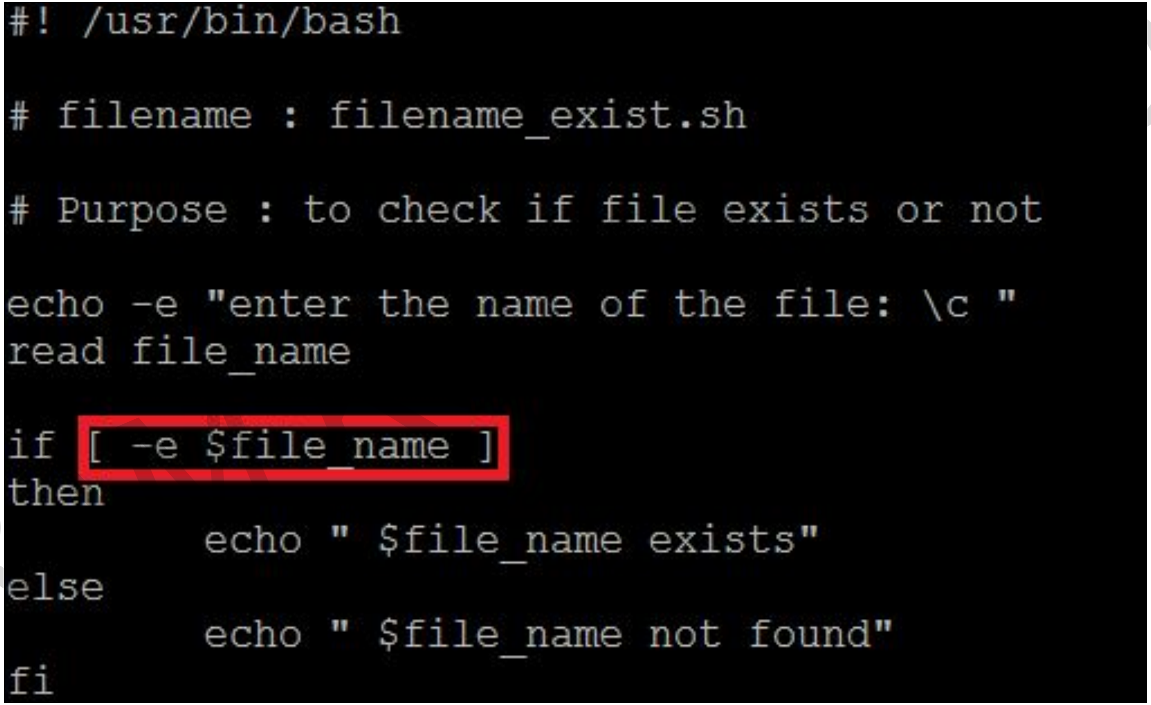
=	is equal to	if ["$a" = "$b"]
=	is equal to	if ["$a" == "$b"]
!=	is not equal to	if ["$a" != "$b"]
<	is less than in ASCII ORDER	if [["$a" < "$b"]]
>	is greater than in ASCII ORDER	if [["$a" > "$b"]]
-z	is string is NULL or string is of Zero Length	

Compare Strings in Bash Scripting

- 

```
#! /usr/bin/bash

# filename : compare_strings.sh

# purpose : to show to functionality of strings in bash scripting

echo " Please enter a comparative string:"
read $string

if [ "$string" == "india" ]
then
echo " you have entered word india "
else
echo     " you have entered otherthan india"
fi
```

File Operations using option -e to check existence of file



```
#! /usr/bin/bash

# filename : filename_exist.sh

# Purpose : to check if file exists or not

echo -e "enter the name of the file: \c "
read file_name

if [ -e $file_name ]
then
        echo " $file_name exists"
else
        echo " $file_name not found"
fi
```

Other File operations

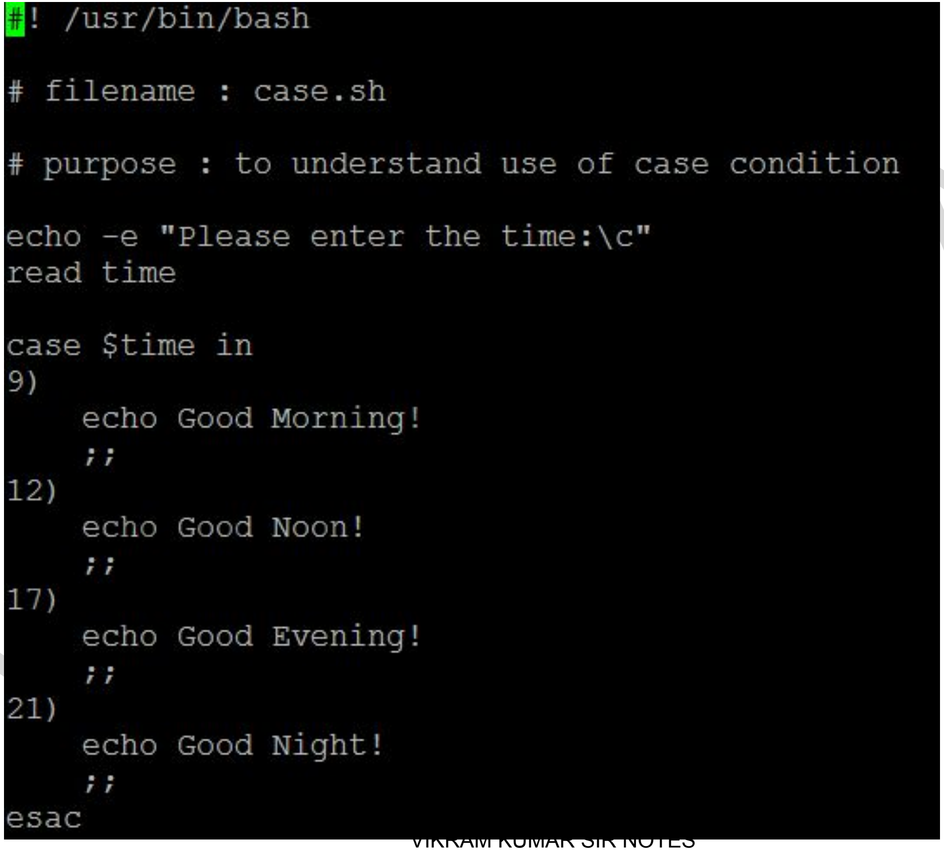
-f	to check if file is regular file type or not
-d	to check if the expression is directory or not
-b	to check if the file is Block special file or not
-s	to check if the file is empty or not
-r	to check read permission of the file
-w	to check write permissions of file
-x	to check execute permission of the file

Case Condition

Syntax :

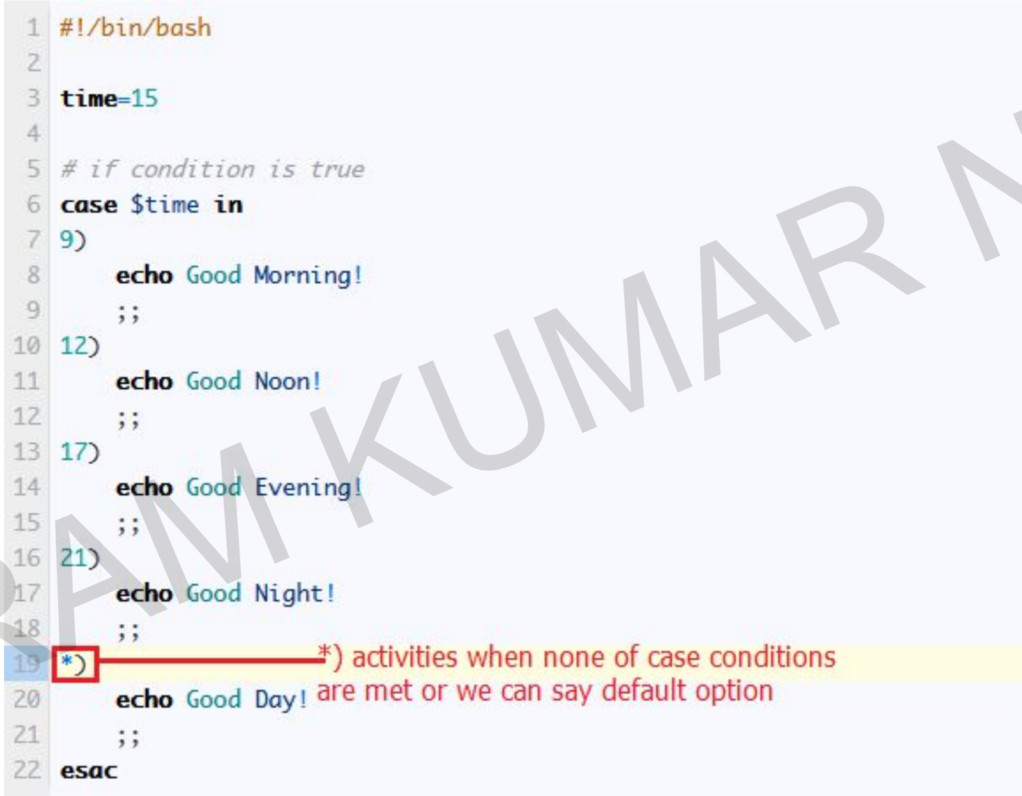
```
case EXPRESSION in

 case1)
                COMMAND-LIST
                 ;;
case2)
                COMMAND-LIST
                 ;;


casen)
                COMMAND-LIST
                 ;;
*)
                Command-list
                ;;
esac
```

Example for Case Condition -



```
#! /usr/bin/bash

# filename : case.sh

# purpose : to understand use of case condition

echo -e "Please enter the time:\c"
read time

case $time in
9)
    echo Good Morning!
    ;;
12)
    echo Good Noon!
    ;;
17)
    echo Good Evening!
    ;;
21)
    echo Good Night!
    ;;
esac
```

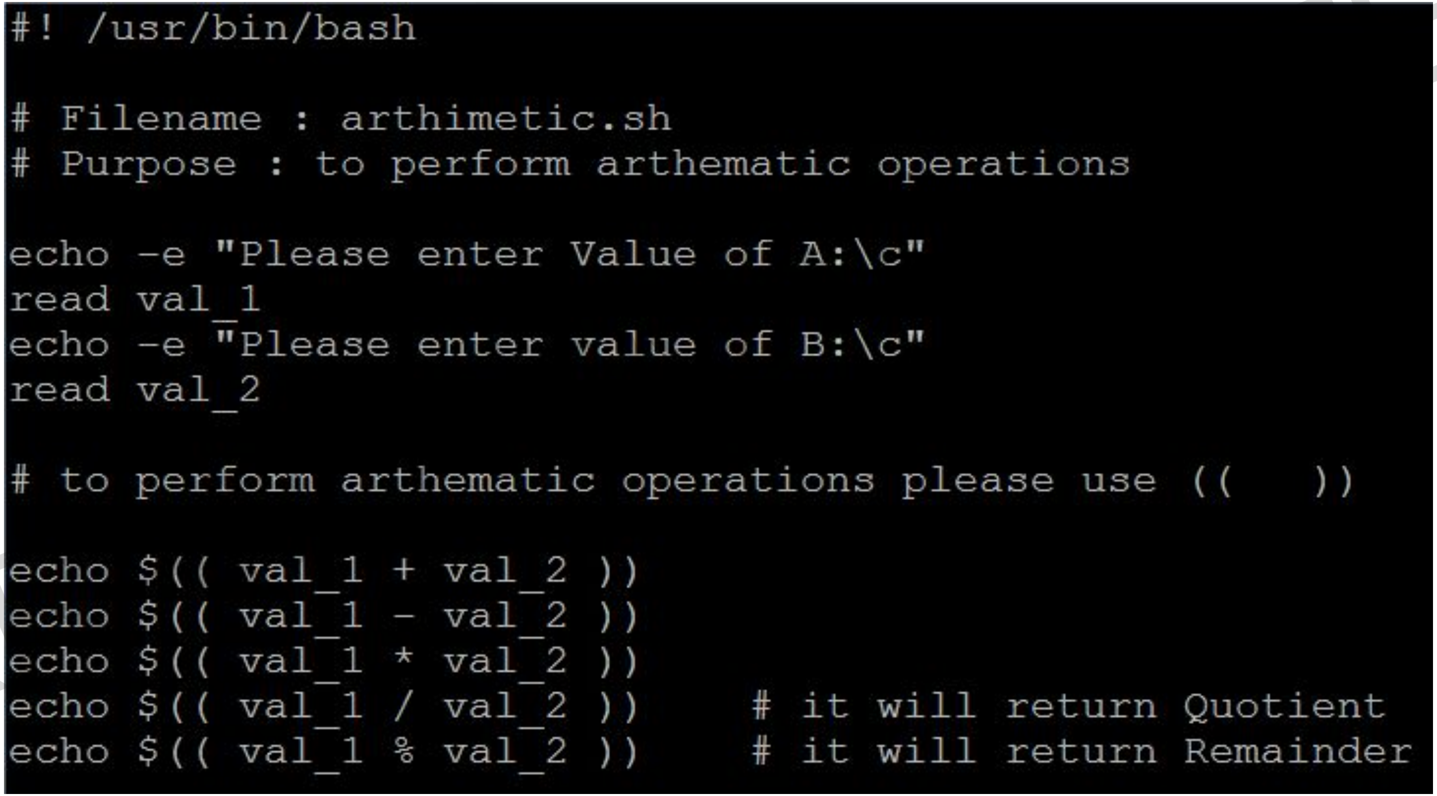
Case with default option



```
#!/bin/bash

time=15

# if condition is true
case $time in
9)
    echo Good Morning!
    ;;
12)
    echo Good Noon!
    ;;
17)
    echo Good Evening!
    ;;
21)
    echo Good Night!
    ;;
*)
    echo Good Day!
    ;;
esac
```

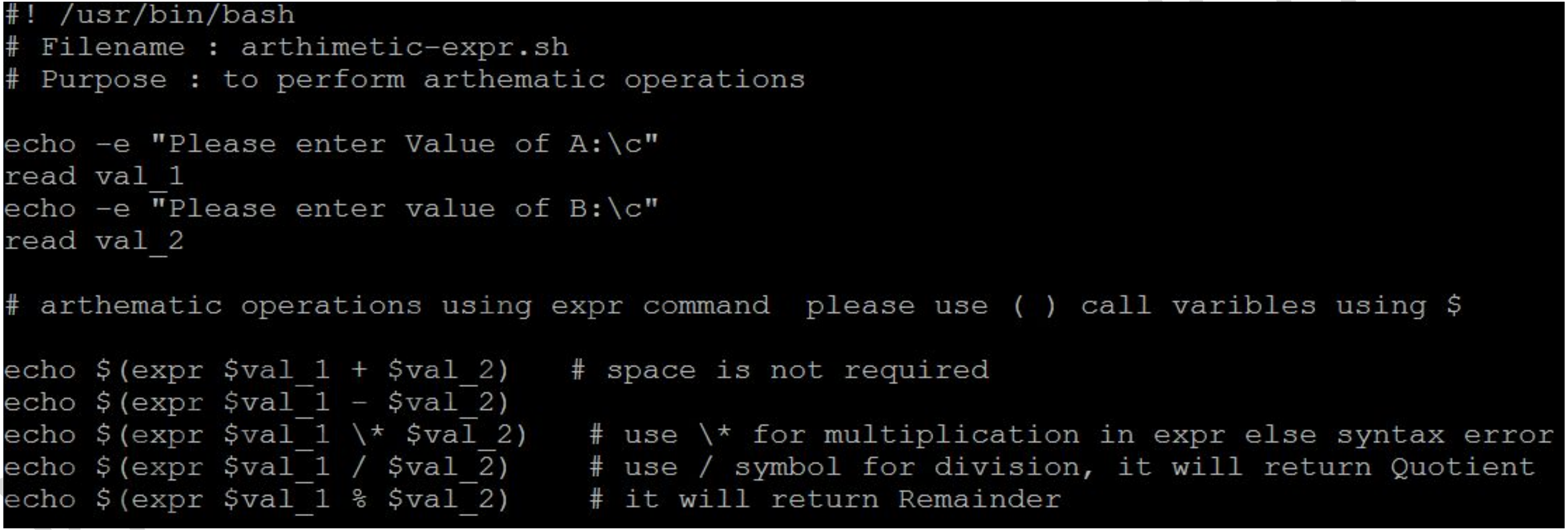
*) activities when none of case conditions are met or we can say default option

Arithmetic Operations -



```
#! /usr/bin/bash

# Filename : arthimetic.sh
# Purpose : to perform arthematic operations

echo -e "Please enter Value of A:\c"
read val_1
echo -e "Please enter value of B:\c"
read val_2

# to perform arthematic operations please use ((   ))

echo $(( val_1 + val_2 ))
echo $(( val_1 - val_2 ))
echo $(( val_1 * val_2 ))
echo $(( val_1 / val_2 ))    # it will return Quotient
echo $(( val_1 % val_2 ))    # it will return Remainder
```

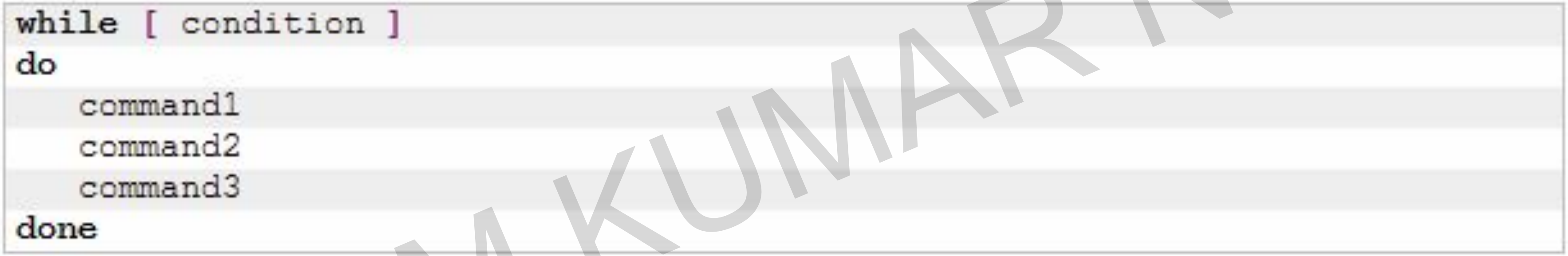
Arithmetic Operations using expr



```
#! /usr/bin/bash
# Filename : arthimetic-expr.sh
# Purpose : to perform arthematic operations

echo -e "Please enter Value of A:\c"
read val_1
echo -e "Please enter value of B:\c"
read val_2

# arthematic operations using expr command  please use ( ) call varibles using $

echo $(expr $val_1 + $val_2)    # space is not required
echo $(expr $val_1 - $val_2)
echo $(expr $val_1 \* $val_2)    # use \* for multiplication in expr else syntax error
echo $(expr $val_1 / $val_2)    # use / symbol for division, it will return Quotient
echo $(expr $val_1 % $val_2)    # it will return Remainder
```

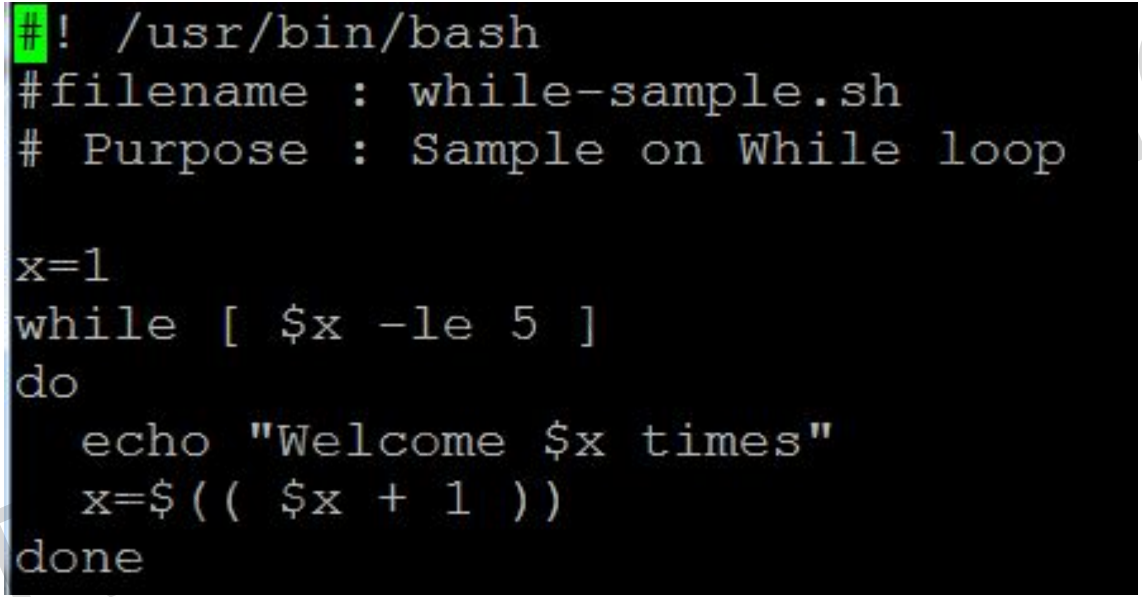
While Loop -

Syntax -



```
while [ condition ]
do
    command1
    command2
    command3
done
```

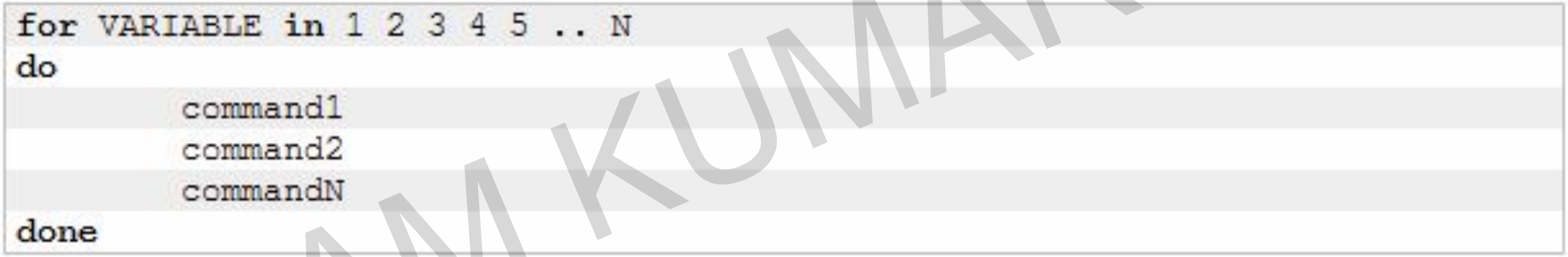
While loop sample program -



```
#! /usr/bin/bash
#filename : while-sample.sh
# Purpose : Sample on While loop

x=1
while [ $x -le 5 ]
do
  echo "Welcome $x times"
  x=$(( $x + 1 ))
done
```

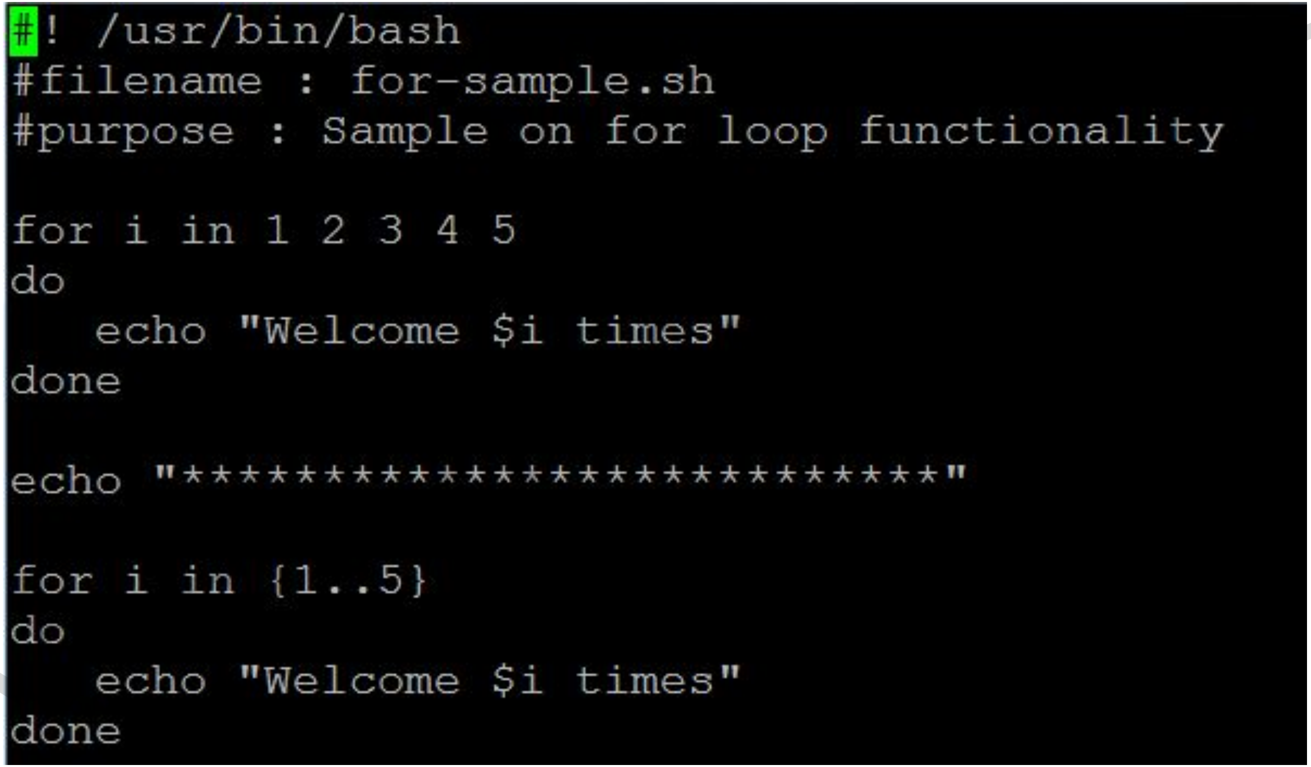
for Loop -

Syntax -



```
for VARIABLE in 1 2 3 4 5 .. N
do
        command1
        command2
        commandN
done
```

for loop sample program -



```
#! /usr/bin/bash
#filename : for-sample.sh
#purpose : Sample on for loop functionality

for i in 1 2 3 4 5
do
   echo "Welcome $i times"
done

echo "***************************"

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

THANK YOU