**Course:** Operating Systems
**Instructor:** Dr. Raheel Ahmed Memon
**Lab-4 and 5**

## Objectives

1. Shell Scripting (*zero to hero*)

# ***SHELL SCRIPTING***

## Theory Part-1

**Common Shells**.
**C-Shell - csh** : The default on teaching systems Good for interactive systems Inferior programmable features
**Bourne Shell - bsh or sh - also restricted shell - bsh** : Sophisticated pattern matching and file name substitution
**Korn Shell** : Backwards compatible with Bourne Shell Regular expression substitution emacs editing mode
**Thomas C-Shell - tcsh** : Based on C-Shell Additional ability to use emacs to edit the command line Word completion & spelling correction Identifying your shell.

**General Things**
**The shbang line**
The "shbang" line is the very first line of the script and lets the kernel know what shell will be interpreting the lines in the script. The shbang line consists of a #! followed by the full pathname to the shell, and can be followed by options to control the behavior of the shell.
**E*XAMPLE***

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

**Comments**
Comments are descriptive material preceded by a # sign. They are in effect until the end of a line and can be started anywhere on the line.
**E*XAMPLE***

```
# this text is not
```

**Assigning values to variables**
A value is assigned to a variable simply by typing the variable name followed by an equal sign and the value that is to be assigned to the variable. For example, if you wanted to assign a value of 5 to the variable count, you would enter the following command:

```
count=5
```

**Accessing variable values**

To access the value stored in a variable precede the variable name with a dollar sign ($). If you wanted to print the value stored in the count variable to the screen, you would do so by entering the following command:

```
echo $count
```

If you omitted the $ from the preceding command, the echo command would display the word count on-screen.

**Positional parameters**

The shell has knowledge of a special kind of variable called a positional parameter. Positional parameters are used to refer to the parameters that were passed to a shell program on the command line or a shell function by the shell script that invoked the function. When you run a shell program that requires or supports a number of command-line options, each of these options is stored into a positional parameter. The first parameter is stored into a variable named 1, the second parameter is stored into a variable named 2, and so forth. These variable names are reserved by the shell so that you can't use them as variables you define. To access the values stored in these variables, you must precede the variable name with a dollar sign ($) just as you do with variables you define.

The following shell program expects to be invoked with two parameters. The program takes the two parameters and prints the second parameter that was typed on the command line first and the first parameter that was typed on the command line second.

#program reverse, prints the command line parameters out in reverse #order

```
echo "$2" "$1"
```

If you invoked this program by entering reverse `hello there`

The program would return the following output:

```
there hello
```

**READ Statement :**

To get the input from the user.

`read x y` (no need of commas between variables)

**The if Statement**

All three shells support nested if...then...else statements. These statements provide you with a way of performing complicated conditional tests in your shell programs. The syntax of the if statement is the same for bash and pdksh and is shown here:

```
if [ expression ]
 then
    commands
 elif [ expression2 ]
   then
     commands
  else
    commands
fi
```

**The case Statement**

The case statement enables you to compare a pattern with several other patterns and execute a block of code if a match is found.

```
case string1 in
str1)
  commands;;
str2)
  commands;;
*)
  commands;;
esac
```

String1 is compared to str1 and str2. If one of these strings matches string1, the commands up until the double semicolon (;;) are executed. If neither str1 nor str2 matches string1, the commands associated with the asterisk are executed. This is the default case condition because the asterisk matches all strings.

The shell languages also provide several iteration or looping statements. The most commonly used of these is the for statement.

**The for Statement**

The ‘for’ statement executes the commands that are contained within it a specified number of times. The ‘bash’ and ‘pdksh’ have two variations of the for statement.

```
for var1 in list
do
  commands
done
for i in {2..10}
do
 echo "output: $i"
done
```

**The while Statement**

Another iteration statement offered by the shell programming language is the while statement. This statement causes a block of code to be executed while a provided conditional expression is true.

```
while expression
do
  statements
done
```

Care must be taken with the while statements because the loop will never terminate if the specifies condition never evaluates to false.

**ARRAYS and for LOOP**

```
NAMES="Hamza Junaid Ali Ahmed Niyaz"
for NAME in $NAMES
do
  echo $NAME
done
```

**Functions**
The shell languages enable you to define your own functions. These functions behave in much the same way as functions you define in C or other programming languages. The main advantage of using functions as opposed to writing all of your shell code in line is for organizational purposes. Code written using functions tends to be much easier to read and maintain and also tends to be smaller, because you can group common code into functions instead of putting it everywhere it is needed.

```
fname () {
shell commands
}
```

**####RELATIONAL OPERATORS TO KNOW###**
# val1 -eq val2 Returns true if the values are equal
# val1 -ne val2 Returns true if the values are not equal
# val1 -gt val2 Returns true if val1 is greater than val2
# val1 -ge val2 Returns true if val1 is greater than or equal to val2
# val1 -lt val2 Returns true if val1 is less than val2
# val1 -le val2 Returns true if val1 is less than or equal to val2

**File Operations:**
A file can be read like:
```
FILE=”test.txt”
```
**Check if file exist?**
```
 if [ -e "$FILE" ]
```

**####FILE TEST OPERATORS TO KNOW####**

| | |
|---|---|
| -d file | True if the file is a directory |
| -e file | True if the file exists (note that this is not particularly portable, thus -f is generally used) |
| -f file | True if the provided string is a file |
| -g file | True if the group id is set on a file |
| -r file | True if the file is readable |
| -s file | True if the file has a non-zero size |

# Practice Part-2

```
#! /bin/bash

 #ECHO COMMAND
 echo Hello World!
```

**INTRODUCTING VARIABLES**

```
# VARIABLES
# Uppercase by convention
# Letters, numbers, underscores
NAME="Bob"
 echo "My name is $NAME"
 echo "My name is ${NAME}"
```

**INPUT FROM USER**

```
 # USER INPUT
 read -p "Enter your name: " NAME
 echo "Hello $NAME, nice to meet you!"
```

**CONDITIONAL IF STATEMENT AND VARIATIONS**

```
# SIMPLE IF STATEMENT
 if [ "$NAME" == "Brad" ]
 then
   echo "Your name is Brad"
 fi
```

**COMPARISON**

```
 NUM1=31
 NUM2=5
 if [ "$NUM1" -gt "$NUM2" ]
 then
   echo "$NUM1 is greater than $NUM2"
 else
   echo "$NUM1 is less than $NUM2"
 fi
```

**######RELATIONAL OPERATORS TO KNOW######**

# val1 -eq val2 Returns true if the values are equal
# val1 -ne val2 Returns true if the values are not equal
# val1 -gt val2 Returns true if val1 is greater than val2
# val1 -ge val2 Returns true if val1 is greater than or equal to val2
# val1 -lt val2 Returns true if val1 is less than val2
# val1 -le val2 Returns true if val1 is less than or equal to val2

## OPERATIONS WITH FILE AND DIRECTORIES

A file can be read like:

```
FILE=”test.txt”
```

**Check if file exist?**

```
 if [ -e "$FILE" ]
```

**####FILE TEST OPERATORS TO KNOW####**

| | |
|---|---|
| -d file | True if the file is a directory |
| -e file | True if the file exists (note that this is not particularly portable, thus -f is generally used) |
| -f file | True if the provided string is a file |
| -g file | True if the group id is set on a file |
| -r file | True if the file is readable |
| -s file | True if the file has a non-zero size |
| -u | True if the user id is set on a file |
| -w | True if the file is writable |
| -x | True if the file is an executable |

## ARRAYS and for LOOP

```
NAMES="Raheel Ali Ahmar Sarwech Junaid"
for NAME in $NAMES
do
  echo $NAME
done
```

## CASE STATEMENT

```
#CASE STATEMENT
 read -p "Are you 18 or over? Y/N " ANSWER
 case "$ANSWER" in
   [yY] | [yY][eE][sS])
     echo "You can make your National ID Card"
     ;;
   [nN] | [nN][oO])
     echo "Sorry, you still have to survive on Form B :("
     ;;
   *)
     echo "Please enter y/yes or n/no"
     ;;
 esac
```

## WHILE LOOP

```
# WHILE LOOP - READ THROUGH A FILE LINE BY LINE
```

```
LINE=1
while read -r CURRENT_LINE
  do
    echo "$LINE: $CURRENT_LINE"
    ((LINE++))
done < "./new-1.txt"
```

## FUNCTIONS IN SHELL SCRIPTING

```
FUNCTION
function sayHello() {
  echo "Hello World"
}
sayHello


FUNCTION WITH PARAMS
function greet() {
  echo "Hello, I am $1 and I am $2"
}

greet "Raheel" "36"
```

# Task Part-03

**TASK-1**
USE *else* and *elif* with logic to get input and decide the statement to execute

**TASK -2**
Check if a particular filename (string) exist?

**TASK -3**
Check if a particular file exists?

**TASK-4**
Input the names in an array from console and greet each name separately (assume that one name is of one word)

**TASK-5**
Use for loop to rename a couple of existing files in a folder,
For example existing files in a particular folder are:
`file1.txt, file2.txt, file3.txt`
After running your script, these should be renamed to:
`new-file1.txt, new-file2.txt, new-file3.txt`

**TASK-6**
Use for loop to rename a number of files on the basis of choices mentioned in case A or B. let's say append "new" in case A or "old" in case B as a prefix of file name if a file date of modification/creation is today or before. Where, today is prefixed with "new" and whatever before is prefixed with "old".

**TASK-7**
Make a function to crawl through the subfolders of current directory and display all the available files with .txt extension.