# 1 Introduction to Shell Programming

This lecture is prepared for beginners who wish to learn the basics of shell scripting/programming plus introduction to power tools such as *awk, sed,* etc.

## 1.1 What is Linux Shell?

- In Operating System, there is special program called **Shell**. Shell accepts your instruction or commands in English (mostly) and if its a valid command, it is passed to kernel.
- Shell is a user program or it's a environment provided for user interaction.
- Shell is an command language interpreter that executes commands read from the standard input device (keyboard) or from a file.
- Shell is not part of system kernel, but uses the system kernel to execute programs, create files etc.
- To find all available shells in your system type following command:

  ```
  $ cat /etc/shells
  ```

- Note that each shell does the same job, but each understand a different command syntax and provides different built-in functions.
- In MS-DOS, Shell name is COMMAND.COM which is also used for same purpose, but it's not as powerful as our Linux Shells are!
- To find your current shell type following command

  ```
  $ echo $SHELL
  ```

- To use shell (You start to use your shell as soon as you log into your system) you have to simply type commands.

## 1.2 What is Shell Script ?

- Normally shells are interactive. It means shell accept command from you (via keyboard) and execute them.
- But if you use command one by one (sequence of 'n' number of commands) , the you can store this sequence of command to text file and tell the shell to execute this text file instead of entering the commands.
- Why to Write Shell Script ?
  - Shell script can take input from user, file and output them on screen.
  - Useful to create our own commands.
  - Save lots of time.
  - To automate some task of day today life.
  - System Administration part can be also automated.

## 1.3 Getting started with Shell Programming

You are introduced to shell programming, how to write script, execute them etc.

**How to write shell script**

- Use any editor to write shell script.
- After writing shell script set execute permission for your script as follows:

```
$ chmod +x your-script-name
or
$ chmod 755 your-script-name
```

- Execute your script as;

```
$ bash your-script-name
$ sh your-script-name
$ ./your-script-name
```

- In the last syntax ./ means current directory.

- But only . (dot) means execute given command file in current shell without starting the new copy of shell as follows;

```
$ . your-script-name
```

Now you are ready to write first shell script that will print "Knowledge is Power" on screen. first

```
#
# My first shell script
#
clear
echo "Knowledge is Power"
```

After saving the above script, you can run the script as follows:

```
$ ./first
```

This will not run script since we have not set execute permission for our script first; to do this type command

```
$ chmod 755 first
$ ./first
```

- Script to print user information who currently login , current date and time.ginfo

```
#
#Script to print user information who currently login , current date
# & time
#
clear
echo "Hello $USER"
echo -e "Today is \c ";date
echo -e "Number of user login : \c" ; who | wc -l
echo "Calendar"
cal
exit 0
```

At the end why statement exit 0 is used?

**Variables in Shell**

- In Linux (Shell), there are two types of variable:

  1. *System variables* - Created and maintained by Linux itself. This type of variable defined in CAPITAL LETTERS.
     - We have learned the command **printenv** to see environment variables. You can see system variables by giving command like some of the important System variables are:

Table 1: Some of the important System variables

| System | Variable Meaning |
|---|---|
| BASH | Our shell name |
| BASH_VERSION | Our shell version name |
| COLUMNS | No. of columns for our screen |
| LINES | No. of columns for our screen |
| PS1 | Our prompt settings |
| HOME | Our home directory |
| OSTYPE | Our Os type |
| PATH | Our path settings |
| PWD | Our current working directory |
| SHELL | Our shell name |
| LOGNAME | Our logging name |
| USERNAME | User name who is currently login to this PC |

  2. *User defined variables* - Created and maintained by user. This type of variable defined in lower letters. The following script defines three variables; variscript

```
#
# Script to test MY knowledge about variables!
#
myname=Vivek
myos = TroubleOS
myno=5
echo "My name is $myname"
echo "My os is $myos"
echo "My number is myno, can you see this number ?"
```

**Shell Arithmetic**

- Use to perform arithmetic operations. Syntax:

```
expr op1 math-operator op2
```

Examples:

```
$ expr 1 + 3
$ expr 2 - 1
$ expr 10 / 2
$ expr 20 % 3
$ expr 10 \* 3
$ echo `expr 6 + 3`
```

  - Multiplication use  and not $*$ since its wild card.
  - Before **expr** keyword we used ` (back quote) sign not the (single quote i.e. ') sign. **expr** is also end with ` i.e. back quote.
  - If you use double quote or single quote, it will NOT work.

- There are three types of quotes

  1. Double quotes (") - Anything enclose in double quotes removed meaning of that characters (except \ and $).
  2. Single quotes (') - Enclosed in single quotes remains unchanged.
  3. Back quote (`) - To execute command

```
$ echo "Today is date"
$ echo "Today is `date`".
```

**Exit Status**

- By default in Linux if particular command/shell script is executed, it return two type of values which is used to see whether command or shell script executed is successful or not.

  1. If return value is zero (0), command is successful.
  2. If return value is nonzero, command is not successful or some sort of error executing command/shell script.

- This value is know as *Exit* Status.

- To find out exit status of command or shell script; use $? special variable of shell.

```
$ rm unknownfile
$ echo $?
$ ls
$ echo $?
$ expr 1 + 3
$ echo $?
$ echo Welcome
$ echo $?
$ something nonse
$ echo $?
$ date
$ echo $?
```

### The read Statement

- Use to get input (data from user) from keyboard and store (data) to variable.
- Following script first ask user, name and then waits to enter name from the user via keyboard. Then user enters name from keyboard (after giving name you have to press ENTER key) and entered name through keyboard is stored (assigned) to variable *fname*. sayH

```
#
#Script to read your name from key-board
#
echo "Your first name please:"
read fname
echo "Hello $fname, Lets be friend!"
```

### Why Command Line arguments required

- Telling the command/utility which option to use.
- Informing the utility/command which file or group of files to process (reading/writing of files).
- Following script is used to print command line argument and will show you how to access them: demo

```
#!/bin/sh
#
# Script that demos, command line args
#
echo "Total number of command line argument are $#"
echo "$0 is script name"
echo "$1 is first argument"
echo "$2 is second argument"
echo "All of them are :- $* or $@"
```

**Redirection of Standard output/input**

- Mostly all commands give output on screen or take input from keyboard, but it's also possible to send output to file or to read input from file.
- There are three main redirection symbols $>, >>, <$
  1. $>$ Redirector Symbol. Syntax:
     Linux-command $>$ filename
     To output Linux-commands result (output of command or shell script) to file. Note that if file already exist, it will be overwritten else new file is created.

     ```
     $ ls > myfiles
     ```

  2. $>>$ Redirector Symbol. Syntax:
     Linux-command $>>$ filename
     To output Linux-commands result (output of command or shell script) to END of file. Note that if file exist , it will be opened and new information/data will be written to END of file, without losing previous information/data, And if file is not exist, then new file is created.

     ```
     $ date >> myfiles
     ```

  3. $<$ Redirector Symbol. Syntax:
     Linux-command $<$ filename
     To take input to Linux-command from file instead of keyboard.

     ```
     $ cat < myfiles
     ```

## 1.4 Shells (bash) structured Language Constructs

This section introduces to the bash's structured language constructs such as:

1. Decision making
2. Loops

### 1.4.1 Decision making

- **if condition**
  - If condition which is used for decision making in shell script, If given condition is true then **command1** is executed.
    Syntax:

```
if condition
then
        command1 if condition is true or if exit status
        of condition is 0 (zero)
        ...
        ...
fi
```

  - Condition is nothing but comparison between two values.
  - For compression you can use **test** or [ **expr** ] statements or even exist status can be also used.
  - An expression is nothing but combination of values, relational operator (such as >, <, <> etc) and mathematical operators (such as +, −, / etc ). showfile

```
#!/bin/sh
#
#Script to print file
#
if cat $1
then
        echo -e "\n\nFile $1, found and successfully echoed"
fi
```

- **test command or [ expr ]**
  - **test** command or [ **expr** ] is used to see if an expression is true, and if it is true it return zero (0), otherwise returns nonzero for false.
    Syntax:

```
test expression OR [ expression ]
```

- Following script determine whether given argument number is positive. ispositive

```
#!/bin/sh
#
# Script to see whether argument is positive
#
if test $1 -gt 0
then
        echo "$1 number is positive"
fi
```

- **test** or [ **expr** ] works with
  1. Integer ( Number without decimal point)
  2. File types
  3. Character strings

- **if...else...fi**
  - If given condition is true then **command1** is executed otherwise **command2** is executed. Syntax:

```
if condition
then
        condition is zero (true - 0)
        execute all commands up to else statement
else
        if condition is not true then
        execute all commands up to fi
fi
```

  - Script to see whether argument is positive or negative; isnumpn

```
#!/bin/sh
#
# Script to see whether argument is positive or negative
#
if [ $# -eq 0 ]
then
        echo "$0 : You must give/supply one integers"
        exit 1
fi
if test $1 -gt 0
then
```

```
        echo "$1 number is positive"
else
        echo "$1 number is negative"
fi
```

- **Nested if-else-fi**
  - You can write the entire **if-else** construct within either the body of the **if** statement of the body of an **else** statement. This is called the nesting of **if**s. Syntax:

```
if condition
then
        if condition
        then
                do this
        else
                do this
        fi
else
        do this
fi
```

nestedif

```
#!/bin/sh
#
# Script to see nesting of ifs
#
osch=0

echo "1. Unix (Sun Os)"
echo "2. Linux (Red Hat)"
echo -n "Select your os choice [1 or 2]? "
read osch

if [ $osch -eq 1 ] ;
then
        echo "You Pick up Unix (Sun Os)"
        else # nested if i.e. if within if #
        if [ $osch -eq 2 ] ;
        then
                echo "You Pick up Linux (Red Hat)"
        else
                echo "What you don't like Unix/Linux OS."
```

```
            fi
    fi
```

- **Multilevel if-then-else** Syntax:

```
if condition
then
        condition is zero (true - 0)
        execute all commands up to elif statement
elif condition1
then
        condition1 is zero (true - 0)
        execute all commands up to elif statement
elif condition2
then
        condition2 is zero (true - 0)
        execute all commands up to elif statement
else
        None of the above condition,condition1,condition2 are true (i.e.
        all of the above nonzero or false)
        execute all commands up to fi
fi
```

For multilevel **if-then-else** statement try the following script: elf

```
#!/bin/sh
# Script to test if..elif...else
#
if [ $1 -gt 0 ];
then
        echo "$1 is positive"
elif [ $1 -lt 0 ]
then
        echo "$1 is negative"
elif [ $1 -eq 0 ]
then
        echo "$1 is zero"
else
        echo "Opps! $1 is not number, give number"
fi
```

Integer comparison is expected.

### 1.4.2 Loops

- **Loops in Shell Scripts**

- Loop defined as: Computer can repeat particular instruction again and again, until particular condition satisfies. A group of instruction that is executed repeatedly is called a loop.
- Bash supports:
  1. for loop
  2. while loop
- Note that in each and every loop,
  * First, the variable used in loop condition must be initialized, then execution of the loop begins.
  * A test (condition) is made at the beginning of each iteration.
  * The body of loop ends with a statement that modifies the value of the test (condition) variable.

- **for Loop**
  Syntax:

```
for { variable name } in { list }
do
        execute one for each item in the list until the list is
        not finished (And repeat all statement between do and done)
done
```

Try the following script: testfor

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

Also try the following script: mtable

```
#!/bin/sh
#
#Script to test for loop
#
#
if [ $# -eq 0 ]
then
        echo "Error - Number missing form command line argument"
        echo "Syntax : $0 number"
        echo "Use to print multiplication table for given number"
        exit 1
```

```
fi
n=$1
for i in 1 2 3 4 5 6 7 8 9 10
do
        echo "$n * $i = `expr $i \* $n`"
done
```

Syntax:

```
for (( expr1; expr2; expr3 ))
do
        repeat all statements between do and
        done until expr2 is TRUE
Done
```

testfor2

```
#!/bin/sh
#
#Script to test for loop 2
#
#
for (( i = 0 ; i <= 5; i++ ))
do
        echo "Welcome $i times"
done
```

- **Nesting of for Loop**
  - As you see the **if** statement can nested, similarly **loop** statement can be nested.
  - To understand the nesting of for loop see the following shell script. nestedfor

```
#!/bin/sh
#
# Nesting of for loop
#
#
for (( i = 1; i <= 5; i++ )) ### Outer for loop ###
do
        for (( j = 1 ; j <= 5; j++ )) ### Inner for loop ###
        do
```

```
                echo -n "$i "
        done
        echo "" #### print the new line ###
done
```

Here, for each value of i the inner loop is cycled through 5 times, with the variable j taking values from 1 to 5. The inner for loop terminates when the value of j exceeds 5, and the outer loop terminates when the value of i exceeds 5.

– Following script is quite interesting, it prints the chess board on screen. chessboard

```
#!/bin/sh
#
# Chessboard
#
#
for (( i = 1; i <= 9; i++ )) ### Outer for loop ###
do
        for (( j = 1 ; j <= 9; j++ )) ### Inner for loop ###
        do
                tot=‘expr $i + $j‘
                tmp=‘expr $tot % 2‘
                if [ $tmp -eq 0 ];
                then
                        echo -e -n "\033[47m "
                else
                        echo -e -n "\033[40m "
                fi
        done
echo -e -n "\033[40m " #### set back background color to black
echo "" #### print the new line ###
done
echo -e -n "\033[47m " #### set back background color to white
echo "" #### print the new line ###
echo "" #### print the new line ###
echo "" #### print the new line ###
echo "" #### print the new line ###
```

- **while loop**

  – Syntax:

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

```
        ..
        ..
done
```

– Loop is executed as long as given condition is true. Above for loop program (shown in last section of for loop) can be written using while loop as; nt1

```
#!/bin/sh
#
#Script to test while statement
#
#
if [ $# -eq 0 ]
then
        echo "Error - Number missing form command line argument"
        echo "Syntax : $0 number"
        echo " Use to print multiplication table for given number"
        exit 1
fi
n=$1
i=1
while [ $i -le 10 ]
do
        echo "$n * $i = `expr $i \* $n`"
        i=`expr $i + 1`
done
```

- **The case Statement**

  – The **case** statement is good alternative to multilevel **if-then-else-fi** statement.

  – It enable you to match several values against one variable. Its easier to read and write. Syntax:

```
case $variable-name in
        pattern1) command
                ..
                command;;
        pattern2) command
                ..
                command;;
        patternN) command
                ..
                command;;
        *) command
                ..
```

```
                command;;
esac
```

- The *$variable-name* is compared against the patterns until a match is found.
- The shell then executes all the statements up to the two semicolons that are next to each other.
- The default is *) and its executed if no match is found. Example; car

```
#!/bin/sh
#
# if no vehicle name is given
# i.e. -z $1 is defined and it is NULL
#
# if no command line arg
if [ -z $1 ]
then
        rental="*** Unknown vehicle ***"
elif [ -n $1 ]
then
        # otherwise make first arg as rental
        rental=$1
fi
case $rental in
        "car") echo "For $rental Rs.20 per k/m";;
        "van") echo "For $rental Rs.10 per k/m";;
        "jeep") echo "For $rental Rs.5 per k/m";;
        "bicycle") echo "For $rental 20 paisa per k/m";;
        *) echo "Sorry, I can not get a $rental for you";;
esac
```

### How to de-bug the shell script?

- While programming shell sometimes you need to find the errors (bugs) in shell script and correct the errors (remove errors - debug).
- For this purpose you can use -v and -x option with sh or bash command to debug the shell script. Syntax:

```
sh option { shell-script-name }
OR
bash option { shell-script-name }
Option can be
-v Print shell input lines as they are read.
-x After expanding each simple-command, bash displays the expanded
   value of system variable,followed by the command and its expanded
    arguments.
```

Example; dsh1

```
#!/bin/sh
#
# Script to show debug of shell
#
tot=‘expr $1 + $2‘
echo $tot
```

execute as

```
$ ./dsh1 4 5
9
$ sh -x dsh1 4 5
++ expr 4 + 5
+ tot=9
+ echo 9
9
$ sh -v dsh1 4 5
#!/bin/sh
#
# Script to show debug of shell
#
tot=‘expr $1 + $2‘
expr $1 + $2
echo $tot
9
```

## 1.5 Advanced Shell Scripting Commands

After learning basis of shell scripting, its time to learn more advance features of shell scripting/command.

### Local and Global Shell variable (export command)

- Normally all our variables are local.
- Local variable can be used in same shell, if you load another copy of shell (by typing the /bin/bash at the $ prompt) then new shell ignored all old shell's variable.
- Global shell defined as: "You can copy old shell's variable to new shell (i.e. first shells variable to seconds shell), such variable is know as Global Shell variable."
- To set global varible you have to use export command. Syntax:

```
export variable1, variable2,.....variableN
```

Examples:

```
$ vech=Bus
$ echo $vech
Bus
$ export vech
$ /bin/bash
$ echo $vech
Bus
$ exit
$ echo $vech
Bus
```

### Conditional execution i.e. && and ||

- The control operators are && (read as AND) and || (read as OR). The syntax for AND list is as follows;

  ```
  command1 && command2
  ```

  *command2* is executed if, and only if, *command1* returns an exit status of zero.

- The syntax for OR list as follows

  ```
  command1 || command2
  ```

  *command2* is executed if and only if *command1* returns a non-zero exit status.

- You can use both as follows

  ```
  command1 && comamnd2 (if exist status is zero) || command3
  (if exit status is non-zero)
  ```

  if *command1* is executed successfully then shell will run *command2* and if *command1* is not successful then *command3* is executed. Example:

  ```
  $ rm myf && echo "File is removed successfully" || echo "File
  is not removed"
  ```

### 1.5.1 User Interface and dialog utility

- Good program/shell script must interact with users. You can accomplish this as follows:

  1. Use command line arguments (args) to script when you want interaction i.e. pass command line args to script as :

     ```
     $ ./sutil foo 4
     ```

     where foo and 4 are command line args passed to shell script *sutil*.
  2. Use statement like echo and read to read input into variable from the prompt. userinte,
  3. Even you can create menus to interact with user, first show menu option, then ask user to choose menu item, and take appropriate action according to selected menu item, this technique is show in following script; menuui.

### Dialog Utility

- User interface usually includes, menus, different type of boxes like info box, message box, Input box etc.
- In Linux shell (i.e. bash) there is no built-in facility available to create such user interface, But there is one utility supplied with Red Hat Linux version 6.0 called **dialog**, which is used to create different type of boxes like info box, message box, menu box, Input box etc.
- To show some information on screen; dial.

  ```
  #!/bin/sh
  #
  # Script to show some information on screen
  #
  dialog --title "Linux Dialog Utility Infobox" --backtitle "Linux
  Shell Script Tutorial" --infobox "This is dialog box called infobox,
  which is used\
  to show some information on screen, Press any key. . . " 7 50 ;
  read
  ```

  - Here 7 and 50 are height-of-box and width-of-box respectively.
  - "Linux Shell Script Tutorial" is the backtitle of dialog show on upper left side of screen and below that line is drawn.

  Use dialog utility to Display dialog boxes from shell scripts. Syntax:

```
dialog --title {title} --backtitle {backtitle} {Box options}
where Box options can be any one of following
--yesno {text} {height} {width}
--msgbox {text} {height} {width}
--infobox {text} {height} {width}
--inputbox {text} {height} {width} [{init}]
--textbox {file} {height} {width}
--menu {text} {height} {width} {menu} {height} {tag1} item1}...
```

- To show some information on screen which has also Ok button; dial2.

```
!/bin/sh
#
# Script to show some information on screen which has also Ok button
#
dialog --title "Linux Dialog Utility Msgbox" --backtitle "Linux Shell
Script Tutorial" --msgbox "This is dialog box called msgbox, which is
used\ to show some information on screen which has also Ok button,
Press any key. . . " 9 50
```

- **yesno** box using dialog utility; dial3.
- Input Box (inputbox) using dialog utility; dial4.
- Putting it all together. Its time to write script to create menus using dialog utility, following are menu items
  - Date/time
  - Calendar
  - Editor

### 1.5.2 getopts command

- This command is used to check valid command line argument are passed to script. Usually used in while loop. Syntax:

```
getopts {optstring} {variable1}
```

**getopts** is used by shell to parse command line argument.

- Each time it is invoked,getopts places the next option in the shell variable variable1,
- When an option requires an argument, getopts places that argument into the variable OPTARG.

- Example; We have script called ani. which has syntax as

```
ani -n -a -s -w -d
Options: These are optional argument
-n name of animal
-a age of animal
-s sex of animal
-w weight of animal
-d demo values (if any of the above options are used their values
are not taken)
```

For executing;

```
$ ani -n Lassie -a 4 -s Female -w 20Kg
$ ani -a 4 -s Female -n Lassie -w 20Kg
$ ani -n Lassie -s Female -w 20Kg -a 4
$ ani -w 20Kg -s Female -n Lassie -a 4
$ ani -w 20Kg -s Female
$ ani -n Lassie -a 4
$ ani -n Lassie
$ ani -a 2
```

- See because of **getopts**, we can pass command line argument in different style

Table 2: Operator in Shell Script

| Mathematical | Operator in Shell |
|---|---|
| -eq | is equal to 5 == 6; if test 5 -eq 6 ; if [ 5 -eq 6 ] |
| -ne | is not equal to 5! = 6; if test 5 -ne 6; if [ 5 -ne 6 ] |
| -lt | is less than 5 < 6; if test 5 -lt 6; if [ 5 -lt 6 ] |
| -le | is less than or equal to 5 <= 6; if test 5 -le 6; if [ 5 -le 6 ] |
| -gt | is greater than 5 > 6; if test 5 -gt 6; if [ 5 -gt 6 ] |
| -ge | is greater than or equal to 5 >= 6; if test 5 -ge 6; if [ 5 -ge 6 ] |
| String Comparisons | Operator Meaning |
| string1 = string2 | string1 is equal to string2 |
| string1 != string2 | string1 is NOT equal to string2 |
| string1 | string1 is NOT NULL or not defined |
| -n string1 | string1 is NOT NULL and does exist |
| -z string1 | string1 is NULL and does exist |
| File and directory | Test Meaning |
| -s file | Non empty file |
| -f file | Is File exist or normal file and not a directory |
| -d dir | Is Directory exist and not a file |
| -w file | Is writeable file |
| -r file | Is read-only file |
| -x file | Is file is executable |
| Logical Operators | Operator Meaning |
| ! expression | Logical NOT |
| expression1 -a expression2 | Logical AND |
| expression1 -o expression2 | Logical OR |