Unit : 4 **Text Editing with vi and nano Editor, Shell Programming**

Part 1 :

- Introduction of vi editor
- Modes in vi
- Switching mode in vi
- Cursor movement
- Screen control commands
- Entering text, cut, copy, paste in vi editor
- Introduction of nano editor

Introduction of vi editor

The default editor that comes with the Linux/UNIX operating system is called **vi** (visual editor). Using vi editor, we can edit an existing file or create a new file from scratch. we can also use this editor to just read a text file. The advanced version of the vi editor is the **vim** editor.

No matter what work you do with the UNIX system, you will eventually write some C programs or shell scripts. You may have to edit some of the system files at times. For all this you must learn to use an editor, and UNIX providers a very versatile one VI editor, VI editor is a screen editor, where a portion of the file is displayed on the terminal screen, and the cursor can be moved around the screen to indicate where you want to make changes. You can select which part of the file you want to have displayed. Screen editors are also called display editors, or visual editors. VI is one of the more popular screen editors that run on the UNIX system.

How to Open VI Editor

vi [file_name]

Example 1: Creating a new file with `file_name` = a.txt

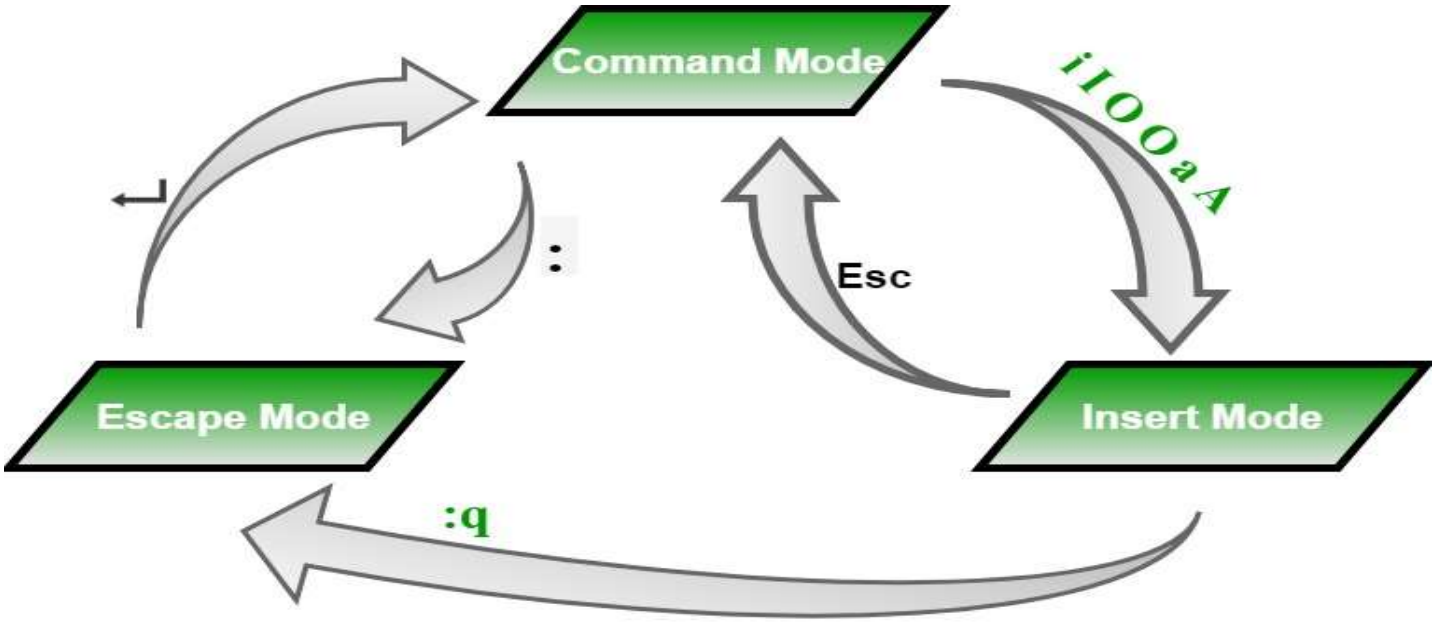
```
vi a.txt
```

Example 2: Opening a preexisted file with `file_name` = a.txt

```
vi a.txt
```

Vi Command Mode :

One of the most important aspects to remember about **vi** is that most of the commands fall into one of three modes:



1) **Command Mode:** is one of its fundamental modes. When you first open Vi/Vim, you are in command mode by default. This mode allows you to navigate, delete, copy, paste, and manipulate text.

This is the default mode of VI editor. This mode is used to give some command for navigation, edition and copy or cut. This is the base mode of VI editor if user want to switch this mode to insert or input mode then user has to press i or I or a or A. If user wants to switch execute mode from this mode then user has to press:

To enter insert mode from command mode:

- i → Insert before the cursor
- I → Insert at the beginning of the line
- a → Append after the cursor
- A → Append at the end of the line
- o → Open a new line below
- O → Open a new line above

2) Input or Insert Mode: This mode is used to enter data in VI editor. If user wants to insert some text to the editor then first user has to select this mode. This mode will be selected from command mode pressing I or i or a or A.

To enter insert mode from command mode:

- i → Insert before the cursor

- I → Insert at the beginning of the line
- a → Append after the cursor
- A → Append at the end of the line
- o → Open a new line below
- O → Open a new line above

3) Execute Mode: This mode is used to save or quit from VI editor. What ever the changes user has done in file using VI editor if user wants to save, find any particular string then using this mode user can save or find the string and also whenever user wants to quit from the VI editor this mode will supports user to quit from the VI editor. To switch to this mode user has to press: or / on command mode

How to Enter Execute Mode?

- **Press :** (colon) in Command Mode → For executing commands like save, quit, and replace.
- **Press /** (forward slash) in Command Mode → For searching forward in the text.
- **Press ?** (question mark) in Command Mode → For searching backward in the text.

Switching mode in vi

While you start VI editor at that time you will be at vi mode that will ready to accept defined command on that particular key but not any input. If you want to input text into the file you will have to go to input mode for that you will have to press “i”.

If you will press “i” at VI mode you will be at input mode here you can input any data into the file from this mode if you want to switch to vi mode then you will have to press “Esc” key. If you will press “Esc” key at input mode you will be able to switch to vi mode.

If you want to switch to command mode from the VI mode then you will have to press “:” or “/” that will support you to switch you from VI mode to command mode.

If you want to switch to command mode from the input mode then you will have to first come to the VI mode then you will be able to switch to command mode. To switch from input mode to command mode then you first will have to press “Esc” key and then after you will have to press “:” or “/” key that will support you to switch from input mode to command mode via VI mode.

Cursor movement in VI editor

Whatever the command you are giving that will work on VI mode only.

To move around in the file, use the arrow keys: the up arrow will move the cursor up one line, the down arrow down one line. The right arrow will cause the cursor to move one position to the right. From the end of a line, it will go to the beginning of the next line.

For terminals with no functional arrow keys, four keys will move the cursor around:

h left arrow
j down arrow
k up arrow
l right arrow

Screen control commands

Whatever the command you are giving that will work on vi mode only.

ctrl + f

This will move the user forward one screen in the file.

ctrl+b

This will move the user backward one screen in the file.

G

This will move the cursor to the end of the file. Note, again that vi, like any other UNIX utility, is always case sensitive.

num

This will bring the cursor to line defined line number.

Entering text, cut, copy, paste in vi editor

- **How to Enter Text in Vi Editor**

  1) Enter Insert Mode

  To start typing, press one of the following keys in **Command Mode**:

Command	Action
i	Insert before the cursor
I	Insert at the beginning of the line
a	Append after the cursor
A	Append at the end of the line
o	Open a new line below the current line
O	Open a new line above the current line

2) Type Your Text
Once in Insert Mode, you can type normally as you would in any text editor.

3) Exit Insert Mode
To return to Command Mode (so you can save, delete, or copy text), press:
Esc

4) Save and Exit Vi Editor
After typing your text, you can save and exit using Execute Mode:
:w → Save (write) the file
:q → Quit (only if no unsaved changes)
:wq or :x → Save and quit
:q! → Quit without saving

Example

- Open Vi:
  vi filename
- Press i to enter Insert Mode.
- Type your text.
- Press Esc to return to Command Mode.
- Save and exit with :wq.

- **How to Cut Text in Vi Editor**
  In **Vi/Vim**, cutting text is done using the **delete (**d**)** command. Anything deleted is stored in a buffer, so it can be pasted elsewhere.

  1) Cut a Single Line (This deletes (cuts) the entire current line.)
     dd

  2) To cut a single word:
     dw

  3) To cut from the cursor to the **end of the word**:
     dE

  4) If you want to cut everything from the **cursor position to the end of the line**:
     d$

5) If you want to cut everything from the **cursor position to the beginning of the line**:
   d0

6) Cut the Entire File (To cut all text in a file:)
   ggVGd

- gg → Move to the beginning of the file.
- V → Select entire lines.
- G → Move to the end of the file.
- d → Cut (delete) the selection.

- **How to Copy Text in Vi Editor**

1) This copies the entire current line.
   yy

2) To copy a single word:
   yw

3) To copy from the cursor to the **end of the word**:
   yE

4) If you want to copy everything from the **cursor position to the end of the line**:
   y$

5) If you want to copy everything from the **cursor position to the beginning of the line**:
   y0

6) To copy all text in a file:
   ggVGy

- gg → Move to the beginning of the file.

- V → Select entire lines.
- G → Move to the end of the file.
- y → Yank (copy) the selection.

- **How to Paste Text in Vi Editor**

1) Paste After the Cursor (or Below the Line)
p

2) Paste Before the Cursor (or Above the Line)
P

3) Copy and Paste a Line
Move to the line you want to copy: yy
Move to where you want to paste and press: p

4) Cut and Paste a Line
Move to the line you want to cut: dd
Move to the new location and press: p

Introduction of nano editor

Nano is a simple, user-friendly, and lightweight command-line text editor available on Linux and Unix systems. It is an alternative to Vi/Vim and is designed to be easy to use for beginners.
Key Features of Nano:

- ☑ Simple and easy to use
- ☑ Displays commands at the bottom of the screen
- ☑ Supports syntax highlighting
- ☑ Allows basic text editing (cut, copy, paste, search, etc.)
- ☑ Available by default in most Linux distributions

How to Open Nano Editor
nano filename

Basic Nano Commands

Command	Description
`Ctrl + X`	Exit Nano
`Ctrl + S`	Save the file
`Ctrl + O`	Write changes to the file
`Ctrl + K`	Cut a line
`Ctrl + U`	Paste a line
`Ctrl + W`	Search for text
`Ctrl + G`	Show help menu
`Ctrl + R`	Open another file in Nano

Saving and Exiting

1. After editing, press Ctrl + X to exit.
2. If changes were made, Nano will ask **"Save modified buffer?"**
   - Press Y (Yes) to save or N (No) to discard.
3. Press Enter to confirm the filename and save.

PART : 2

- Shell Keywords
- Shell Variables
- System variables
  PS2, PATH, HOME,LOGNAME, MAIL, IFS, SHELL, TERM, MAILCHECK
- User variables
  set, unset and echo command with shell variables
- Positional Parameters
- Interactive shell script using read and echo
- Decision Statements
  - o if then fi
  - o if then else fi
  - o if then elif else fi
  - o case esac
- test command
- Logical Operators
- Looping statements
  - o for loop
  - o while loop
  - o until loop
  - o break, continue command
- Array
- Function
- Various shell script examples

Shell Keyword

In Linux shell scripting, a keyword is a reserved word that has a special meaning to the shell. These keywords are part of the shell's syntax and cannot be used as variable names or function names.

Shell Keywords	
Keyword	Description
if	Starts an if conditional statement
then	Used in an if statement to define the action
else	Defines an alternative action in an if statement
elif	Else-if condition in an **if** statement
fi	Ends an if statement
for	Starts a for loop
while	Starts a while loop
until	Starts an until loop (runs until a condition is true)
do	Starts a loop block (for, while, until)
done	Ends a loop block (for, while, until)
case	Starts a multi-way conditional statement
esac	Ends a case statement
function	Defines a function in Bash
return	Exits from a function and returns a value
exit	Terminates a script or shell session
break	Exits a loop prematurely
continue	Skips the current iteration of a loop
readonly	Declares a variable as read-only (cannot be changed)
export	Makes a variable available to child processes

Shell variables

Shell variables are used to store data (such as strings or numbers) that can be used within a shell session or script. They are essential for managing system settings, user preferences, and script automation.

Rules for variable definition

A variable name could contain any alphabet (a-z, A-Z), any digits (0-9), and an underscore (_). However, a variable name must start with an alphabet or underscore. It can never start with a number. Following are some examples of valid and invalid variable names:

- **Valid Variable Names**
- ABC
  _AV_3
  AV232

- **Invalid variable names**
- 2_AN
  !ABD
  $ABC
  &QAID

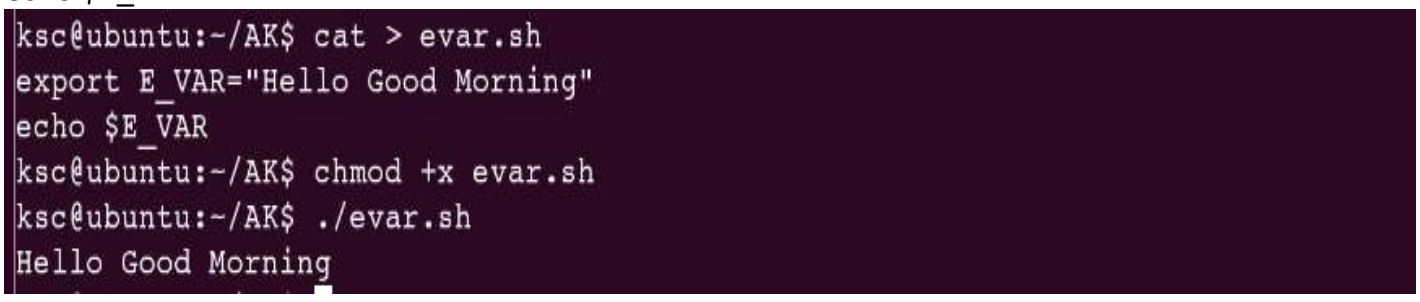
In shell scripting there are three main types of variables are present. They are –

1. Local Variables
2. Global Variables or Environment Variables
3. Shell Variables or System Variables

1. **Local Variables :** Defined within the shell session or script and exist only in that session. Variables which are specific to the current instance of shell. They are basically used within the shell, but not available for the program or other shells that are started from within the current shell.

a="Hello Dear Students"
echo $a

2. **Environment Variables :** Available to all processes started from that shell.
   These variables are commonly used to configure the behavior script and programs that are run by shell. Environment variables are only created once, after which they can be used by any user.
   export E_VAR="Hello Good Morning"
   echo $E_VAR



3. **Special Variables (Shell Built-in Variables) :** Predefined by the shell and used for system functions.
   Example: $HOME, $PATH, $USER, $PWD, $SHELL
   $PWD` = Stores working directory
   `$HOME` = Stores user's home directory

`$SHELL` = Stores the path to the shell program that is being used.

```
ksc@ubuntu:~/AK$ echo $HOME
/home/ksc
ksc@ubuntu:~/AK$ echo $SHELL
/bin/bash
ksc@ubuntu:~/AK$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
ksc@ubuntu:~/AK$ echo $USER
ksc
ksc@ubuntu:~/AK$ echo $PWD
/home/ksc/AK
```

- **System variables:** system variables are **environment variables** that the editor can use to configure settings, interact with the operating system, and run shell commands. These variables come from the **shell environment** and can be accessed or modified within Vim.

User & Environment Variables

Variable	Description
$HOME	The home directory of the current user.
$USER	The username of the logged-in user.
$SHELL	The shell being used (e.g., /bin/bash).
$PATH	The system's executable search path.
$PWD	The present working directory.

```
ksc@ubuntu:~/AK$ echo $LOGNAME
ksc
ksc@ubuntu:~/AK$ echo $PS2
>>
ksc@ubuntu:~/AK$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
ksc@ubuntu:~/AK$ echo $HOME
/home/ksc
ksc@ubuntu:~/AK$ echo $MAIL
/home/ksc/mailbox
ksc@ubuntu:~/AK$ echo $IFS

ksc@ubuntu:~/AK$ echo $SHELL
/bin/bash
ksc@ubuntu:~/AK$ echo $TERM
xterm
ksc@ubuntu:~/AK$ echo $MAILCHECK
60
```

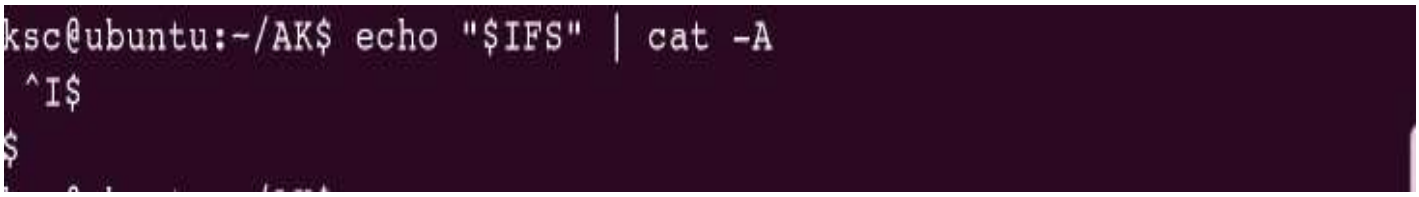
PS2: PS2 is an **environment variable** in Unix/Linux shells (like Bash, Zsh) that defines the **secondary prompt string**. It appears when a command is not complete, such as when entering a multi-line command.

MAIL: The $MAIL environment variable stores the **path to the current user's mailbox**, where incoming emails are stored. This is mainly used in Unix/Linux systems for local mail delivery (e.g., system notifications, cron job reports).

LOGNAME: $LOGNAME is an environment variable that stores the current logged-in user's username. It is automatically set by the system when you log in and is often used in scripts and applications.

IFS: IFS (Internal Field Separator) is a special environment variable in Unix/Linux that defines how the shell splits words and lines into fields. By default, IFS includes:

- Space (' '): Separates words
- Tab ('\t'): Recognizes tab-separated values
- Newline ('\n'): Treats each line separately



You may see **spaces, tabs (^I), and newlines ($)**.

TERM : $TERM is an **environment variable** that defines the **type of terminal** being used. It tells applications (like Vim, bash, tmux, etc.) how to handle display settings, colors, and key bindings based on the terminal type.

MAILCHECK: $MAILCHECK is an environment variable in Unix/Linux that **controls how often the shell checks for new mail** in the mailbox specified by $MAIL.

User variables

1. **set :** The set command is used to display or modify shell environment settings and variables.
   -x : It prints the commands and their arguments in the same sequence as they got executed.

Example 1:

```
ksc@ubuntu:~$ cat > usrvar.sh
set -x
echo "Hello"
echo "Students"
```

Output:

```
ksc@ubuntu:~$ chmod +x usrvar.sh
ksc@ubuntu:~$ ./usrvar.sh
++ echo Hello
Hello
++ echo Students
Students
```

Example 2

set.sh

```
set Apple Banana Mango Orange
echo $1
echo $2
echo $3
echo $4
```

Output:

```
ksc@ubuntu:~/AK$ chmod +x set.sh
ksc@ubuntu:~/AK$ ./set.sh
Apple
Banana
Mango
Orange
ksc@ubuntu:~/AK$
```

2. **Unset :** The unset command removes variables or functions.

unset.sh

```
set Apple Banana Mango Orange
set --
echo $1
echo $2
echo $3
echo $4
echo "Hello"
```

```
ksc@ubuntu:~/AK$ chmod +x unset.sh
ksc@ubuntu:~/AK$ ./unset.sh



Hello
```

The unset command removes variables or functions.

unset.sh ✖ unset2.sh ✖

```
a="Hello"
echo "$a"
unset a
echo "$a"
```

a="Hello"
echo "$a" # Outputs: Hello
unset a
echo "$a" # Outputs nothing (variable is removed)

```
ksc@ubuntu:~/AK$ chmod +x unset2.sh
ksc@ubuntu:~/AK$ ./unset2.sh
Hello
```

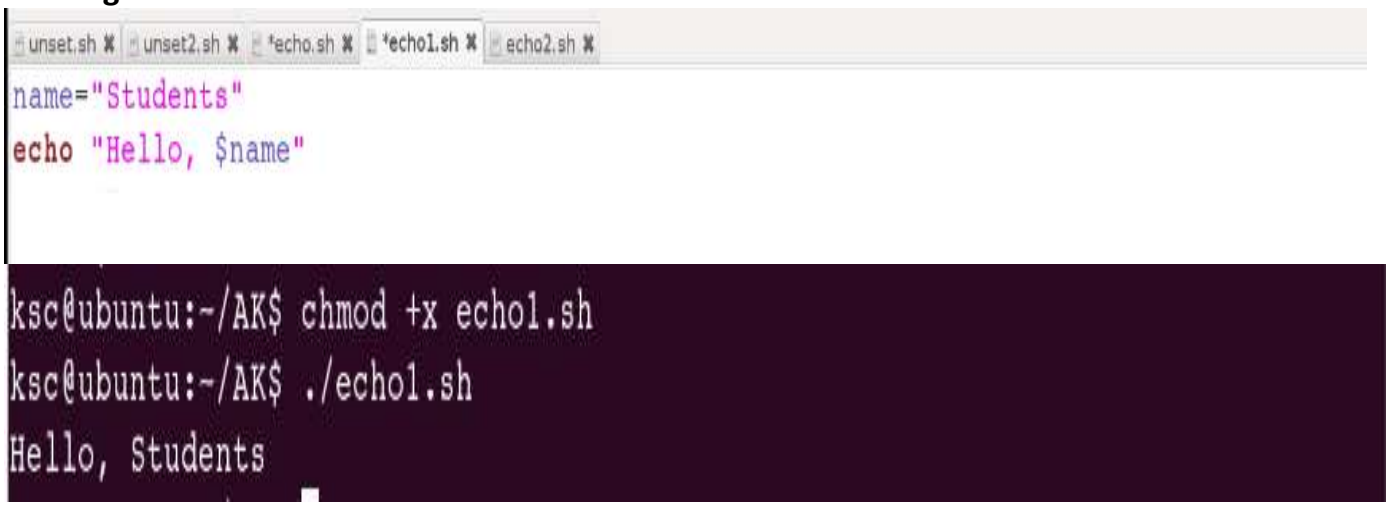
3. **echo :** The echo command prints text or variable values.
   Printing a String :

unset.sh ✖ unset2.sh ✖ *echo.sh ✖ echo1.sh ✖ echo2.sh ✖

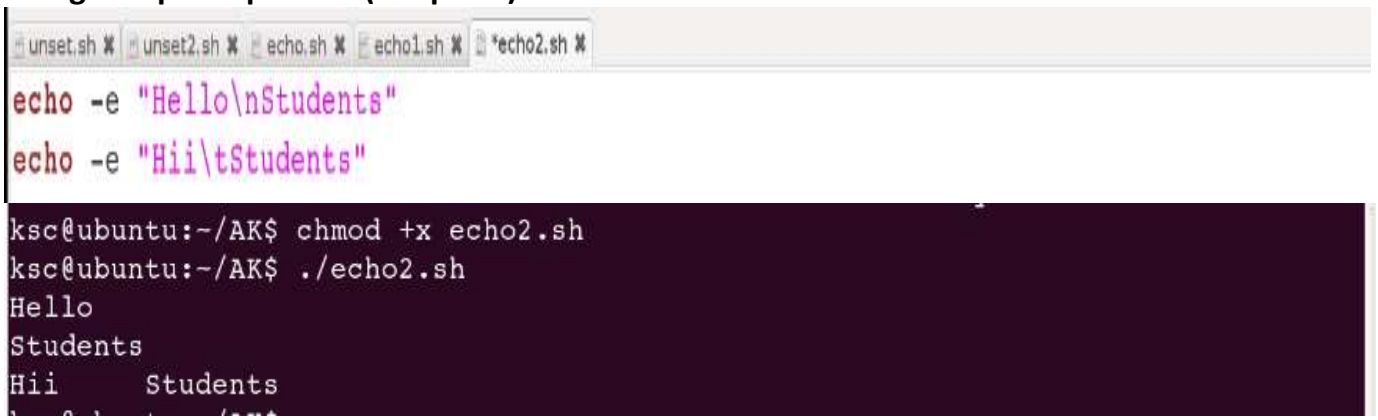
```
echo "Hello Dear Students"
```

```
ksc@ubuntu:~/AK$ chmod +x echo.sh
ksc@ubuntu:~/AK$ ./echo.sh
Hello Dear Students
```

Printing a Variable Value :



Using Escape Sequences (-e option)



Example of set, unset, echo

```
ksc@ubuntu:~/AK$ chmod +x sue.sh
ksc@ubuntu:~/AK$ ./sue.sh
++ a='Hello, Students'
++ echo 'Hello, Students'
Hello, Students
++ unset a
++ echo 'After unset: '
After unset:
++ set +x
```

Positional Parameters

A positional parameter is a variable that refers to a specific position in a list of arguments.

When a shell script is executed with arguments, those arguments are stored in **positional parameters**.

Special Positional Parameters

Symbol	Description
$0	Name of the script
$1, $2, ...	Arguments passed to the script
$#	Total number of arguments
$@	All arguments as separate words
$*	All arguments as a single string
"$@"	Preserves argument boundaries (preferred over $*)
"$*"	Treats all arguments as a single string
$$	Process ID (PID) of the script
$?	Exit status of the last command
$!	PID of the last background process

unset.sh | unset2.sh | echo.sh | echo1.sh | echo2.sh | sue.sh | positional.sh | positional1.sh

```
echo "Name of Script: $0"
echo "1st Argument passed: $1"
echo "2nd Argument passed: $2"
echo "3rd Argument passed: $3"
echo "All Argument as seprate words: $@"
echo "Total No. of Argument: $#"
echo "All argument as a single string: $*"
echo "Preserves Argument Boundaries: "$@""
echo "Treats all Arguments as a single string: "$*""
echo "Process ID of the script: $$"
echo "Exit Status of Last Command: $?"
echo "Process ID of the last background process: $!"
```

```
ksc@ubuntu:~/AK$ ./positional1.sh Hello Dear Students
Name of Script: ./positional1.sh
1st Argument passed: Hello
2nd Argument passed: Dear
3rd Argument passed: Students
All Argument as seprate words: Hello Dear Students
Total No. of Argument: 3
All argument as a single string: Hello Dear Students
Preserves Argument Boundaries: Hello Dear Students
Treats all Arguments as a single string: Hello Dear Students
Process ID of the script: 2276
Exit Status of Last Command: 0
Process ID of the last background process:
```

Interactive Shell Script using read and echo

An interactive shell script allows users to input values during execution using the read command. The echo command displays messages or user input.

Example 1:

```
*read.sh
echo "Enter Your Name"
read nm
echo "Hello, $nm! Welcome To Kamani Science College"
```

```
ksc@ubuntu:~/AK$ chmod +x read.sh
ksc@ubuntu:~/AK$ ./read.sh
Enter Your Name
Student
Hello, Student! Welcome To Kamani Science College
```

Example 2:

```
read1.sh
echo "Enter Your First Name, Middle Name and  Last  name:"
read fnm mnm lnm
echo "Your Name is: $fnm $mnm $lnm"

echo "My First Name is: $fnm"
echo "My Middle Name is: $mnm"
echo "My Last Name is: $lnm"
```

```
ksc@ubuntu:~/AK$ chmod +x read1.sh
ksc@ubuntu:~/AK$ ./read1.sh
Enter Your First Name, Middle Name and  Last  name:
Hi Hii Hiii
Your Name is: Hi Hii Hiii
My First Name is: Hi
My Middle Name is: Hii
My Last Name is: Hiii
```

Example 3:

You can use the -p flag to show the prompt in the same line.

```
read -p "Enter Your Age:" age
echo "You are $age years Old"
```

```
ksc@ubuntu:~/AK$ chmod +x read2.sh
ksc@ubuntu:~/AK$ ./read2.sh
Enter Your Age:95
You are 95 years Old
```

Example 4:

The -s flag hides the input (useful for passwords).

```
read1.sh  *read2.sh  *read3.sh
read -sp "Enter Password:" psw
echo -e "\n Password Saved Successfully"
```

```
ksc@ubuntu:~/AK$ chmod +x read3.sh
ksc@ubuntu:~/AK$ ./read3.sh
Enter Password:
 Password Saved Successfully
```

Decision Statements

- o if then fi
- o if then else fi
- o if then elif else fi
- o case esac

1. **if then fi :** The if...fi statement is the fundamental control statement that allows Shell to make decisions and execute statements conditionally.

Syntax:

```
if [ expression ]
then
   statement
fi
```

Example 1: Write a shell program to check a number is Equal

if.sh

```
echo "Enter No1:"
read a
echo "Enter No2:"
read b

if [ $a == $b ]
then
    echo "Both are Equal"
fi
```

```
ksc@ubuntu:~/AK$ ./if.sh
Enter No1:
10
Enter No2:
10
Both are Equal
ksc@ubuntu:~/AK$ ./if.sh
Enter No1:
20
Enter No2:
10
```

2. **if then else fi :** The if-then-else-fi construct is used for decision-making in shell scripts. It allows executing one block of code if a condition is **true** and another if it is **false**.

Syntax:

```
if [ expression ]
then
   statement1
else
   statement2
fi
```

Example 1: Write a shell program to check a number is Equal or Not

equal.sh ✖ if.sh ✖

```
echo "Enter No1:"
read no1
echo "Enter No2:"
read no2

if [ $no1 == $no2 ]
then
    echo "Both are Equal"
else
    echo "Both are Different"
fi
```

```
ksc@ubuntu:~/AK/if_else$ ./equal.sh
Enter No1:
20
Enter No2:
10
Both are Different
```

Example : 2 Write a shell program to check a number is Even or Odd

file.sh ✖ odd.sh ✖

```
read -p "Enter No:" no
if [ $no%2==0 ]
then
echo "No is Even"
else
echo "No is Odd"
fi
```

```
ksc@ubuntu:~/AK/if_else$ chmod +x odd.sh
ksc@ubuntu:~/AK/if_else$ ./odd.sh
Enter No:20
No is Even
ksc@ubuntu:~/AK/if_else$
```

Example 3: Write a shell program to check a file exists or not

positive.sh | file.sh

```
read -p "Enter your File Name:" file
if [ -f "$file" ]
then
echo "File Exists"
else
echo "File does not Exists"
fi
```

```
ksc@ubuntu:~/AK/if_else$ chmod +x file.sh
ksc@ubuntu:~/AK/if_else$ ./file.sh
Enter your File Name:positive.sh
File Exists
ksc@ubuntu:~/AK/if_else$ chmod +x file.sh
ksc@ubuntu:~/AK/if_else$ ./file.sh
Enter your File Name:negative.sh
File does not Exists
```

Example 4: Write a shell program to check a Eligible for voting or not

if.sh | if_else_vote.sh

```
echo "Enter your age: "
read age
echo "your Age is $age"
 if [ $age -ge 18 ]
 then
        echo "You are Eligible for Voting"
 else
        echo "You are Not Eligible for Voting"
fi
```

```
ksc@ubuntu:~/AK$ ./if_else_vote.sh
Enter your age:
20
your Age is 20
You are Eligible for Voting
```

if then elif else fi : The if-then-elif-else-fi construct is used when multiple conditions need to be checked in sequence. It allows for branching logic where different conditions lead to different actions.

Syntax:

```
if [ expression1 ]
then
   statement1
   statement2
   .
   .
elif [ expression2 ]
then
   statement3
   statement4
   .
   .
else
   statement5
fi
```

Example 1: Write a shell program to check a number is Positive, Negative OR Zero

positive.sh

```
read -p "Enter Number:" no
if [ $no -gt 0 ]
then
  echo "Your No $no is Positive"
elif [ $no -lt 0 ]
then
   echo "Your No $no is Negative"
else
   echo "Your No $no is Zero"
fi
```

```
ksc@ubuntu:~/AK/if_else$ ./positive.sh
Enter Number:20
Your No 20 is Positive
```

Example 2: Write a shell program to check a file and directory exists or not

file.sh

```
read -p "Enter File or Directory Name :" a
if [ -f $a ]
then
echo "Your File : $a"
elif [ -d $a ]
then
echo "Your Directory : $a"
else
echo "Invalid Data"
fi
```

```
ksc@ubuntu:~/AK/if_elif$ ./file.sh
Enter File or Directory Name :file.sh
Your File : file.sh
```

Example 3: Write a shell program to create marksheet

result.sh

```
echo "Enter C# Marks : "
read m1
echo "Enter JAVA Marks : "
read m2
echo "Enter OS Marks : "
read m3
echo "Enter SEO Marks : "
read m4
echo "Enter IOT Marks : "
read m5
echo "Enter DE Marks : "
read m6

total=`expr $m1 + $m2 + $m3 + $m4 + $m5 + $m6`
echo "Total: " $total

per=`expr $total / 5`
echo "Percentage: " $per

if [ $per -gt 90 ]
then
echo "Grage : A+"
elif [ $per -gt 80 -a $per -lt 90 ]
then
echo "Grade : B+"
elif [ $per -gt 70 -a $per -lt 80 ]
then
echo "Grade : B"
elif [ $per -gt 60 -a $per -lt 70 ]
then
echo "Grade : C+"
elif [ $per -gt 50 -a $per -lt 60 ]
then
echo "Grade : C"
elif [ $per -gt 35 -a $per -lt 50 ]
then
echo "Grade : D"
else
echo "Fail"
fi
```

```
ksc@ubuntu:~$ cd AK/if_elif
ksc@ubuntu:~/AK/if_elif$ chmod +x result.sh
ksc@ubuntu:~/AK/if_elif$ ./result.sh
Enter C# Marks :
80
Enter JAVA Marks :
90
Enter OS Marks :
83
Enter SEO Marks :
95
Enter IOT Marks :
63
Enter DE Marks :
12
Total:  423
Percentage:  84
Grade : B+
```

Example 4: Write a shell program to convert Celsius to Fahrenheit or Fahrenheit to Celsius

result.sh | week.sh | cel.sh

```
echo "1. Convert Celsius to fehrenheit"
echo "2. Convert fehrenheit to Celsius"
echo -n "select your choice: "
read ch
if [ $ch -eq 1 ]
then
echo -n "Enter temperature in celsius: "
read c
f=$(echo "scale=2; ((9/5) * $c) +32" | bc )
echo "$c C = $f F"
elif [ $ch -eq 2 ]
then
echo -n "Enter celsius in temperature: "
read f
c=$(echo "scale=2; ((5/9) * $f) -32" | bc )
echo "$f F = $c C"
else
echo "Enter valid choice"
fi
```

```
ksc@ubuntu:~/AK/case$ chmod +x cel.sh
ksc@ubuntu:~/AK/case$ ./cel.sh
1. Convert Celsius to fehrenheit
2. Convert fehrenheit to Celsius
select your choice: 1
Enter temperature in celsius: 35
35 C = 95.00 F
ksc@ubuntu:~/AK/case$
```

If then else if then fi fi : Nested if statements allow multiple levels of condition checking

A **nested** if **statement** in shell scripting is an if condition inside another if condition. It is used when multiple conditions need to be checked in a structured way

Syntax:

```
if [ expression1 ]
then
   statement1
   statement2
   .
else
   if [ expression2 ]
   then
      statement3
      .
   fi
fi
```

Example 1 :Write a shell program to check a number is Positive, Negative and Odd, Even

senior.sh ✖ odd.sh ✖

```
read -p "Enter Number :" no
if [ $no%2==0 ]
then
echo "No is Even"
   if [ $no -lt 0 ]
   then
    echo "No is Negative"
   else
     echo "No is Positive"
   fi
else
 echo "No is Odd"
   if [ $no -gt 0 ]
    then
    echo "No is Positive"
    else
    echo "No is Negative"
  fi
fi
```

```
ksc@ubuntu:~/AK/Nested_if$ ./odd.sh
Enter Number :20
No is Even
No is Positive
ksc@ubuntu:~/AK/Nested_if$ ./odd.sh
Enter Number :-50
No is Even
No is Negative
```

Example 2: Write a shell program to check a person is Adult, minor or senior citizen

senior.sh

```
echo "Enter Your Age: "
read age
if [ $age -gt 18 ]
  then
    if [ $age -gt 60 ]
      then
      echo "You are Senior Citizen"
   else
       echo "You are Adult"
    fi
else
   echo "You are minor"
fi
```

```
ksc@ubuntu:~/AK/Nested_if$ ./senior.sh
Enter Your Age:
90
You are Senior Citizen
ksc@ubuntu:~/AK/Nested_if$ ./senior.sh
Enter Your Age:
35
You are Adult
ksc@ubuntu:~/AK/Nested_if$ ./senior.sh
Enter Your Age:
12
You are minor
```

case esac : The case-esac construct in shell scripting is used for multi-way branching, similar to switch-case in other programming languages. It simplifies handling multiple conditions compared to using multiple if-elif-else statements.

Syntax:

```
case  in
   Pattern 1) Statement 1;;
   Pattern n) Statement n;;
esac
```

Example 1 : Write a shell program to perform different command

```
echo "1.ls"
echo "2. pwd"
echo "3. who"
echo "4. date"
echo "5. cal"
read -p "Enter your choice: " a

case $a in
1) echo -e "List File And Folder\n" $(ls);;
2) echo -e "Current directory path\n" $(pwd) ;;
3) echo -e "User Information\n" $(who);;
4) echo -e "Current date\n" $(date) ;;
5) echo -e "Calender\n" $(cal) ;;
*) echo -e "Invalid Command" ;;
esac
```

```
ksc@ubuntu:~/AK/case$ ./command.sh
1.ls
2. pwd
3. who
4. sort
Enter your choice: 2
Current directory path
 /home/ksc/AK/case
```

Example 2 : Write a shell program to check file extension

extension.sh | arith.sh | command.sh

```
read -p "Enter File Name :" file
case $file in
*.txt) echo "Text file" ;;
*.sh) echo "Shell File" ;;
*.png) echo "Image File" ;;
*) echo "Invalid file" ;;
esac
```

```
ksc@ubuntu:~/AK/case$ chmod +x extension.sh
ksc@ubuntu:~/AK/case$ ./extension.sh
Enter File Name :arith.sh
Shell File
```

Example 3: Write a shell program to perform Arithmetic operator

extension.sh ✖ arith.sh ✖ command.sh ✖

```
read -p "Enter No 1: " no1
read -p "Enter No 2: " no2
read -p "Enter any one operator (+ - * / %): " arith
case $arith in
+) echo "Addition $(($no1 + $no2))";;
-) echo "Substraction $(($no1 - $no2))";;
*) echo "Multiplication $(($no1 * $no2))";;
/) echo "Division $(($no1 / $no2))";;
%) echo "Modulo $(($no1 % $no2))";;
*) echo "Invalid Operator";;
esac
```

```
ksc@ubuntu:~/AK/case$ ./arith.sh
Enter No 1: 20
Enter No 2: 10
Enter any one operator (+ - * / %)*
Multiplication 200
```

Example 4: Write a shell program to check Weekday Checking

```
read -p "Enter Number of Day(1 - 7) :" day
case $day in
1)echo "Sunday" ;;
2)echo "Monday" ;;
3)echo "Tuesday" ;;
4)echo "Wednesday" ;;
5)echo "Thursday" ;;
6)echo "Friday" ;;
7)echo "Saturday" ;;
*)echo "Invalid Choid Plz enter number between 1 to 7" ;;
esac
```

```
ksc@ubuntu:~/AK/case$ chmod +x week.sh
ksc@ubuntu:~/AK/case$ ./week.sh
Enter Number of Day(1 - 7) :5
Thursday
```

Example 5: Write a shell program to convert km to meter, cm and inch

```
echo "Enter kilometer: "
read km

echo "Enter Your Choice:"
echo "1. kilometer to meter"
echo "2. kilometer to centimeter"
echo "3. kilometer to Inch"
read -p "Your choice: " c

case $c in
1)r=$(echo "$km*1000" | bc )
  echo "$km km = $r Meter" ;;
2)r=$(echo "$km*100000" | bc )
  echo "$km km = $r centimeter" ;;
3)r=$(echo "$km*39370.1" | bc )
  echo "$km km = $r Inch" ;;
*) echo "Invalid Choice"
esac
```

```
20 km = 2000000 centimeter
ksc@ubuntu:~/AK/case$ ./dis.sh
Enter kilometer:
50
Enter Your Choice:
1. kilometer to meter
2. kilometer to centimeter
3. kilometer to Inch
Your choice2
50 km = 5000000 centimeter
```

test command

The test command in a shell script is used to evaluate expressions related to files, strings, and numbers.

odd.sh

```
read -p "Enter No:" no
if test $no%2==0
then
echo "No is Even"
else
echo "No is Odd"
fi
```

```
ksc@ubuntu:~$ cd AK/test
ksc@ubuntu:~/AK/test$ chmod +x odd.sh
ksc@ubuntu:~/AK/test$ ./odd.sh
Enter No:20
No is Even
```

Logical Operators

Logical operators help in combining multiple conditions in shell scripts.

1. AND (&& or -a) : Returns true only if both conditions are true.

Example 1: Write a shell script program to Check if a File Exists and is Writable

```
File  Edit  View  Search  Tools  Documents  Help
Open  Save  Undo
and.sh  a-.sh
echo "Enter File Name: "
read file
if [ -f $file ]&&[ -w $file ]
then
echo "File is exists and editable"
else
echo "No available"
fi
```

```
ksc@ubuntu:~/AK/Logical_operator$ ls -l or.sh
---x--x--x 1 ksc ksc 0 2025-03-31 19:47 or.sh
ksc@ubuntu:~/AK/Logical_operator$ ./a-.sh
Enter File Name:
or.sh
No available
```

Example2:

```
and.sh
read -p "Enter Your Age :" age
if [ $age -ge 18 -a $age -lt 100 ]
then
echo "Eligible for votting"
fi
```

```
ksc@ubuntu:~/AK/Logical_operator$ ./and.sh
Enter Your Age :79
Eligible for votting
```

Example 3: Write a shell script program to create marksheet

```
echo "Enter 3 Sub MArks :"
read m1 m2 m3

total=`expr $m1 + $m2 + $m3`
echo "Total :" $total

per=`expr $total / 3`
echo "Percentage :" $per

if [ $per -gt 90 ]
then
echo "Grade: A+"
elif [ $per -gt 80 ]&&[ $per -le 90 ]
then
echo "Grade A"
elif [ $per -gt 70 ]&&[ $per -le 80 ]
then
echo "Grade B+"
elif [ $per -gt 60 ]&&[ $per -le 70 ]
then
echo "Grade B"
elif [ $per -gt 50 ]&&[ $per -le 60 ]
then
echo "Grade C"
elif [ $per -gt 35 ]&&[ $per -le 50 ]
then
echo "Grade D"
else
echo "Fail"
fi
```

```
ksc@ubuntu:~$ chmod +x mark.sh
ksc@ubuntu:~$ ./mark.sh
Enter 3 Sub MArks :
70 80 90
Total : 240
Percentage : 80
Grade B+
ksc@ubuntu:~$
```

2. **OR (|| or -o) :** Returns true if at least one condition is true.

Example 1: Write a shell script program to Check if a File Exists OR a Directory Exists

```
and.sh ✖ a-.sh ✖ or.sh ✖
echo "Enter File Name: "
read fd
if [ -f $fd ]||[ -d $fd ]
then
echo "File or Directory"
else
echo "No available"
fi
```

```
ksc@ubuntu:~/AK/Logical_operator$ ./or.sh
Enter File Name:
and.sh
File or Directory
```

Example 2: Write a shell script program to Check if a File is Readable OR Writable

```
and.sh ✖ a-.sh ✖ or.sh ✖
echo "Enter File Name: "
read f
if [ -f $f ]
then
  if [ -r $f ] || [ -w $f ]
  then
    if [ -r $f ]
    then
  echo "$f is a Readable"
  elif [ -w $f ]
  then
  echo "$f is a editable"
 fi
fi
else
  echo "Not exists"
fi
```

```
ksc@ubuntu:~/AK/Logical_operator$ ./or.sh
Enter File Name:
and.sh
and.sh is a Readable
```

Example 3: Write a shell script program to check if user to enter a file or directory name, checks whether it is a file or a directory

```
echo "Enter File Name: "
read fd
if [ -f $fd ] || [ -d $fd ]
then
 if [ -f $fd ]
  then
  echo "$fd is a File not Directory"
  elif [ -d $fd ]
  then
  echo "$fd is a Directory not File"
 fi
else
  echo "No File or Directory"
fi
```

```
ksc@ubuntu:~/AK/Logical_operator$ ./or.sh
Enter File Name:
and.sh
and.sh is a File not Directory
```

3. **NOT (!) :** Reverses the condition (true → false, false → true).

```
echo "Enter File Name: "
read f
if [ -f $f ]&&[ ! -w $f ]
  then
    echo "Read or execute"
else
  echo "Write"
fi
```

```
ksc@ubuntu:~/AK/Logical_operator$ ./or.sh
Enter File Name:
and.sh
Write
```

Example 2: Leap Year

leap.sh

```
read -p "Enter Year: " year
#divisible by 400 or 4 ->leap year divisible by 100->not
leap year
if [ $year%400==0 ] || [ $year%4==0 ]&&[ $Year%100 != 0 ]
then
   echo "Leap Year"
else
 echo "Not Leap Year"
fi
```

```
ksc@ubuntu:~/AK/Logical_operator$ ./leap.sh
Enter Year: 2024
Leap Year
```

Looping statements

for loop : The for loop in shell scripting is used to iterate over a list of values or a range of numbers. A for loop is used when you need to repeat a set of commands for a specific number of times or over a list of items.

List Loop	**for item in a b c; do ... done**
Number Range	**for i in {1..5}; do ... done**
seq Command	**for i in $(seq 1 2 10); do ... done**
C-Style Loop	**for ((i=1; i<=5; i++)); do ... done**
File Loop	**for file in *.txt; do ... done**

Syntax

```
for var in word1 word2 ... wordN
do
   Statement(s) to be executed for every word.
done
```

Example 1: Loop Using a Range of Numbers

range.sh

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

```
ksc@ubuntu:~/AK/for$ chmod +x range.sh
ksc@ubuntu:~/AK/for$ ./range.sh
1
2
3
4
5
```

Example 2: Loop Using seq Command

range.sh | seq.sh

```
for i in $(seq 1 5)
do
echo $i
done
```

```
ksc@ubuntu:~/AK/for$ chmod +x seq.sh
ksc@ubuntu:~/AK/for$ ./seq.sh
1
2
3
4
5
```

Example 3: Even Number

range.sh | seq.sh | c.sh | even.sh

```
for ((i=1; i<=10; i++))
do
 if((i%2==0))
  then
   echo $i
 fi
done
```

```
ksc@ubuntu:~/AK/for$ ./even.sh
2
4
6
8
10
```

Example 4: Odd Number

```
: <<EOF
for i in {1..20..2}
do
echo $i
done
EOF

for i in $(seq 1 2 20)
do
echo $i
done

for i in {1..20}
do
 if ((i%2==1))
  echo $i
 fi
done
```

```
ksc@ubuntu:~/AK/for$ chmod +x odd.sh
ksc@ubuntu:~/AK/for$ ./odd.sh
1
3
5
7
9
11
13
15
17
19
```

Example 5:

range.sh | seq.sh | c.sh

```
for ((i=1; i<=5; i++))
do
echo $i
done
```

```
ksc@ubuntu:~/AK/for$ chmod +x c.sh
ksc@ubuntu:~/AK/for$ ./c.sh
1
2
3
4
5
```

Example 6:

tri.sh | square.sh

```
read -p "Enter no of Row: " r
for ((i=1; i<=r; i++))
do
 for ((j=1; j<=r; j++))
 do
   echo -n "* "
 done
 echo " "
done
```

```
ksc@ubuntu:~/AK/for/pattern$ ./square.sh
Enter no of Row: 5
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

Example 7: Print a Hollow Square

tri.sh | square.sh | bsquare.sh

```
echo "Enter No of rows: "
read r
echo "Enter No of cols: "
read c

for ((i=1; i<=r; i++))
do
 for ((j=1; j<=c; j++))
 do
  if [ $i -eq 1 ]||[ $i -eq $r ]||[ $j -eq 1 ]||[ $j -eq $c ]
  then
   echo -n "*" #print border with *
   else
 echo -n " " #print space for inner area
 fi
done
echo #print new line
done
```

```
Enter No of rows:
5
Enter No of cols:
5
*****
*   *
*   *
*   *
*****
```

Example 08: Print a Right-Angled Triangle

tri.sh

```
read -p "Enter no of Row: " r
for ((i=1; i<=r; i++))
do
 for ((j=1; j<=i; j++))
 do
   echo -n "* "
 done
 echo " "
done
```

```
ksc@ubuntu:~/AK/for/pattern$ chmod +x tri.sh
ksc@ubuntu:~/AK/for/pattern$ ./tri.sh
Enter no of Row: 5
*
* *
* * *
* * * *
* * * * *
```

Example 09: Print an Inverted Right-Angled Triangle

tri.sh Untitled Document 1

```
read -p "Enter no of Row: " r
for ((i=r; i>=1; i--))
do
 for ((j=1; j<=i; j++))
 do
   echo -n "* "
 done
 echo " "
done
```

```
ksc@ubuntu:~/AK/for/pattern$ ./tri.sh
Enter no of Row: 5
* * * * *
* * * *
* * *
* *
*
```

Example 10: Print a Pyramid Pattern

tri.sh pyramid.sh

```
read -p "Enter no of Row: " r
# outer loop (Height of pyramid)
for ((i=1; i<=r; i++))
do
# inner loop for space before *
 for ((j=i; j<r; j++))
 do
   echo -n " " #print space
 done
#Inner loop for *
 for ((j=1; j<=(2*i-1); j++))
 do
   echo -n "*" #print * with space
 done
 echo #move next line
done
```

```
ksc@ubuntu:~/AK/for/pattern$ ./pyramid.sh
Enter no of Row: 5
    *
   ***
  *****
 *******
*********
```

Example 11: Print a Diamond Pattern

tri.sh pyramid.sh

```
read -p "Enter no of Row: " r
# outer loop for rows
for ((i=1; i<=r; i++))
do
# inner loop for space before *
 for ((j=i; j<r; j++))
 do
   echo -n " " #print space
 done
#Inner loop for *
 for ((j=1; j<=(2*i-1); j++))
 do
   echo -n "*" #print * with space
 done
 echo #move next line
done
# outer loop for rows
for ((i=r-1; i>=1; i--))
do
# inner loop for space
 for ((j=r; j>i; j--))
 do
   echo -n " " #print space
 done
#Loop for *
 for ((j=1; j<=(2*i-1); j++))
 do
   echo -n "*" #print * with space
 done
 echo #move next line
done
```

```
ksc@ubuntu:~/AK/for/pattern$ ./pyramid.sh
Enter no of Row: 5
    *
   ***
  *****
 *******
*********
 *******
  *****
   ***
    *
```

Example 12: Print Number Pyramid

pyramid.sh | tri.sh | left.sh | demo.sh

```
read -p "Enter no of Row: " r
for ((i=1; i<=r; i++))
do
 for ((j=1; j<=i; j++))
 do
   echo -n "$j "
 done
 echo
done
```

```
ksc@ubuntu:~$ ./demo.sh
Enter no of Row: 5
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

Example 13: Print Pascal's Pyramid

pyramid.sh | tri.sh | left.sh | *demo.sh

```
read -p "Enter no of Row: " r
for ((i=0; i<r; i++))
do
 n=1
 for ((j=0; j<=i; j++))
 do
   echo -n "$n "
   n=$((n*(i-j)/(j+1)))
 done
 echo
done
```

```
Enter no of Row: 5
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

Example 14: Print Floyd's Pyramid

pyramid.sh tri.sh left.sh demo.sh

```
read -p "Enter no of Row: " r
for ((i=1; i<=r; i++))
do
 n=1
 for ((j=1; j<=i; j++))
 do
   echo -n "$n "
   n=$((n+1))
 done
 echo
done
```

```
ksc@ubuntu:~$ ./demo.sh
Enter No of Rows: 5
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15
```

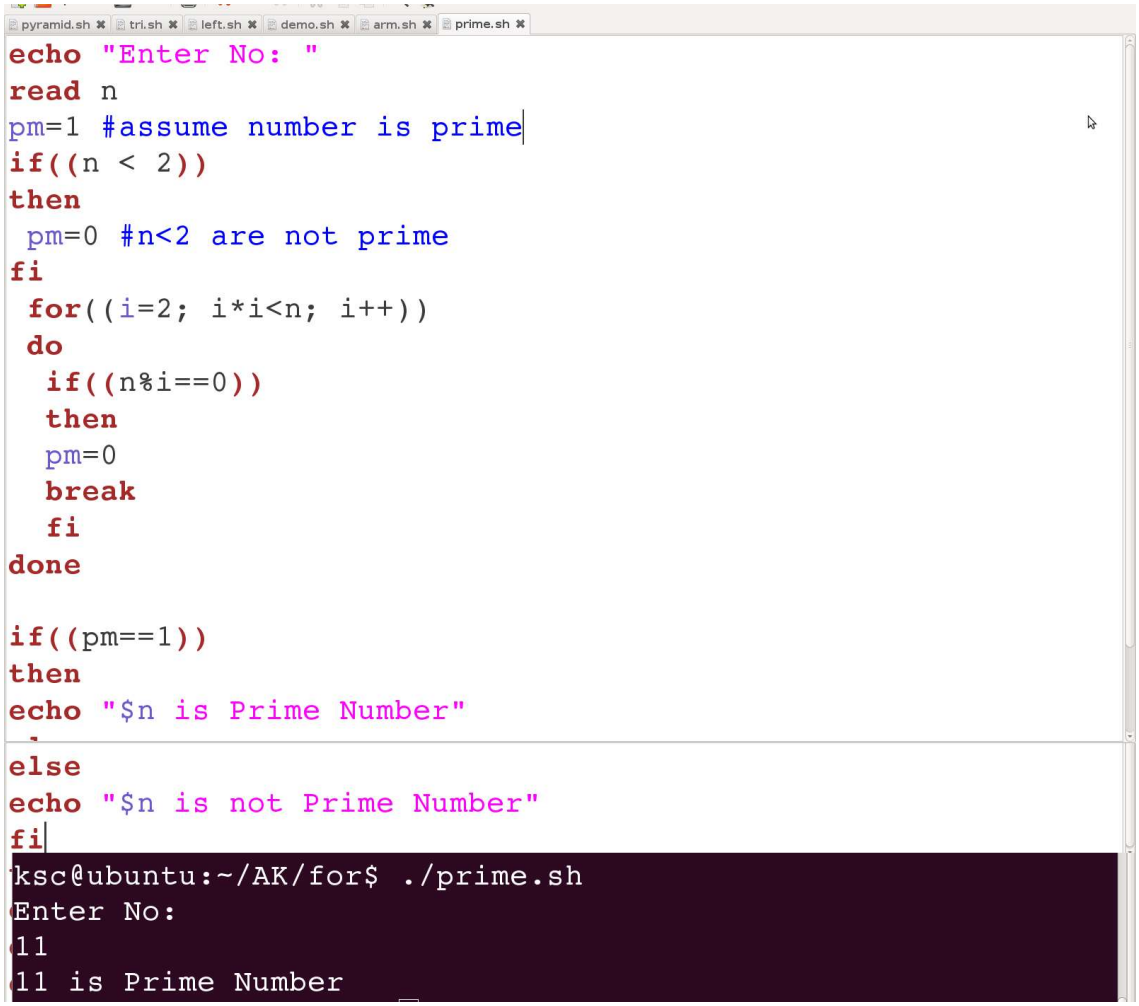
Example 15: Print Your name Pyramid

pyramid.sh tri.sh left.sh *demo.sh

```
read -p "Enter your Name: " nm
len=${#nm} #calculate length of string
for ((i=1; i<=len; i++))
do
 echo "${nm:0:i}" #i controls how many character print each
iteration
done
```

```
ksc@ubuntu:~/AK/for$ ./pyramid.sh
Enter Your Name:
Student
S
St
Stu
Stud
Stude
Studen
Student
```

Example 16: Prime Number



Example:17 Prime Number

pyramid.sh | tri.sh | left.sh | demo.sh | arm.sh | prime.sh

```
echo "Enter Start No: "
read sn
echo "Enter End NO: "
read en
for((n=sn; n<=en; n++))
do
pm=1 #assume number is prime
if((n < 2))
then
 pm=0 #n<2 are not prime
fi
 for((i=2; i*i<n; i++))
 do
  if((n%i==0))
  then
  pm=0
  break
  fi
done
if((pm==1))
then
echo "$n "
fi
done
echo ""
```

```
ksc@ubuntu:~/AK/for$ ./prime.sh
Enter Start No:
1
Enter End NO:
5
2
3
4
5
```

Example 18: Print Your name Reverse Pyramid

leap.sh pyramid.sh

```
echo "Enter Your Name: "
read nm

len=${#nm}
for((i=len; i>=1; i--))
do
echo "${nm:0:i}"
done
```

```
ksc@ubuntu:~/AK/for$ ./pyramid.sh
Enter Your Name:
Students
Students
Student
Studen
Stude
Stud
Stu
St
S
```

while loop : A while loop in a shell script (like in Bash) allows you to execute a block of code repeatedly as long as a given condition is true. The while loop is used in shell scripting to repeat a set of commands as long as a certain condition is true.

Syntax

```
while command
do
   Statement(s) to be executed for every word.
done
```

Example 1:

arm.sh basic.sh

```
echo "Enter Start No: "
read n
read -p "Enter End No: " e
while [ $n -le $e ]
do
 echo "Count: $n"
 ((n++)) #increment
done
```

```
ksc@ubuntu:~/AK/while$ ./basic.sh
Enter Start No:
6
Enter End No: 8
Count: 6
Count: 7
Count: 8
```

Example 2: Fibonacci Series

arm.sh | basic.sh | fib.sh

```
read -p "Enter No: " n
a=o #First Fibonacci number
b=1 # 2nd Fibonacci No
c=0 #counter to track the no of terms print
echo "Fibonacci Series......"
while [ $c -lt $n ]
do
 echo -n "$a " #Print without new line using -n
fib=$((a+b)) #calculate
a=$b #a takes value of b
b=$fib #b takes new fibonacci no
((c++)) #Increment
done
echo #moves to new line
```

```
ksc@ubuntu:~/AK/while$ ./fib.sh
Enter No: 12
Fibonacci Series......
o 1 1 2 3 5 8 13 21 34 55 89
```

Example 3: Palindrome Number

arm.sh | basic.sh | fib.sh | menu.sh | palindrome.sh

```
read -p "Enter No: " no
r=0
tmp=$no #store original no
while [ $tmp -gt 0 ]
do
d=$((tmp % 10)) #Extract last digit
r=$((r * 10 + d)) #append digit to reverse
tmp=$((tmp / 10)) #Remove last digit
done
if [ $no -eq $r ]
then
echo "Palindrome"
else
echo "Not palindrome"
fi
```

```
ksc@ubuntu:~/AK/while$ ./palindrome.sh
Enter No: 121
Palindrome
```

Example 4: Menu driven

arm.sh | basic.sh | fib.sh | menu.sh

```
n=0 #loop runs at least once
while [ $n != "3" ] #loop runs n is not 3
do
 echo "1. Date"
 echo "2. Current Path"
 echo "3. Exit"
read -p "Enter Choice: " n
case $n in
1) date;;
2) pwd;;
3) echo "Exit" ;;
*) echo "Inavalid Choice";;
esac
done
```

```
ksc@ubuntu:~/AK/while$ ./menu.sh
1. Date
2. Current Path
3. Exit
Enter Choice: 2
/home/ksc/AK/while
1. Date
2. Current Path
3. Exit
Enter Choice: 3
Exit
```

Example 5:

arm.sh | basic.sh | fib.sh | menu.sh | palindrome.sh | prime.sh | timer.sh

```
echo "Enter NO: "
read n
while [ $n -ge 1 ]
do
 echo "Count: $n"
 ((n--)) #Decrement
  sleep 3
done
echo "Time's up"
```

```
ksc@ubuntu:~/AK/while$ chmod +x timer.sh
ksc@ubuntu:~/AK/while$ ./timer.sh
Enter NO:
2
Count: 2
Count: 1
Time's up
```

Example 6: Armstrong Number

arm.sh

```
echo "Enter No: "
read no
tmp=$no
sum=0
n=${#no} #count no of digit
while [ $tmp -gt 0 ]
do
    d=$((tmp % 10)) #Extract last digit
    sum=$((sum + d ** n)) #Add d^n to sum
    tmp=$((tmp / 10)) #remove last digit
done

if [ $sum -eq $no ]
 then
 echo "Armstrong"
else
 echo "Not Armstrong"
fi
```

```
ksc@ubuntu:~/AK/while$ chmod +x arm.sh
ksc@ubuntu:~/AK/while$ ./arm.sh
Enter No:
153
Armstrong
```

until loop: The until loop in shell scripting is used when you want to keep executing a set of commands until a certain condition becomes true — in other words, it loops while the condition is false.

Syntax

```
until command
do
   Statement(s) to be executed for every word.
done
```

Example 1:

arm.sh basic.sh fib.sh menu.sh palindrome.sh prime.sh timer.sh count.sh table.sh

```
echo "Enter No: "
read no
until [ $no -lt 1 ] #stop when no<1
do
echo "No: $no"
sleep 2
no=$((no - 1))
done
echo "Time's up"
```

```
ksc@ubuntu:~/AK/until$ ./count.sh
Enter No:
3
No: 3
No: 2
No: 1
Time's up
```

Example 2:

arm.sh basic.sh fib.sh menu.sh palindrome.sh prime.sh timer.sh count.sh table.sh

```
echo "Enter no: "
read no
i=1
until [ $i -gt 10 ] #stop i>10
do
echo "$no x $i = $((no * i))"
i=$((i+1)) #Inccrement i
done
```

```
ksc@ubuntu:~/AK/until$ chmod +x table.sh
ksc@ubuntu:~/AK/until$ ./table.sh
Enter no:
6
6 x 1 = 6
6 x 2 = 12
6 x 3 = 18
6 x 4 = 24
6 x 5 = 30
6 x 6 = 36
6 x 7 = 42
6 x 8 = 48
6 x 9 = 54
6 x 10 = 60
```

break and continue command :

Command	Effect
`break`	Exits the loop entirely
`continue`	Skips to the next iteration of the loop

break in Shell Scripts

Usage: Exits the current loop entirely (useful to stop a loop based on a condition).

Can also break out of nested loops with a number argument (e.g., break 2 breaks two levels deep).

1.sh 2.sh 3.sh 4.sh 1.sh

```
for i in {1..5}
do
echo "No: $i"
if [ $i -eq 4 ]
then
 echo "Break loop"
break
fi
done
echo "Loop End"
```

```
ksc@ubuntu:~/AK/break$ ./1.sh
No: 1
No: 2
No: 3
No: 4
Break loop
Loop End
```

continue in Shell Scripts

Usage: Skips the rest of the loop for the current iteration only, then continues with the next iteration.

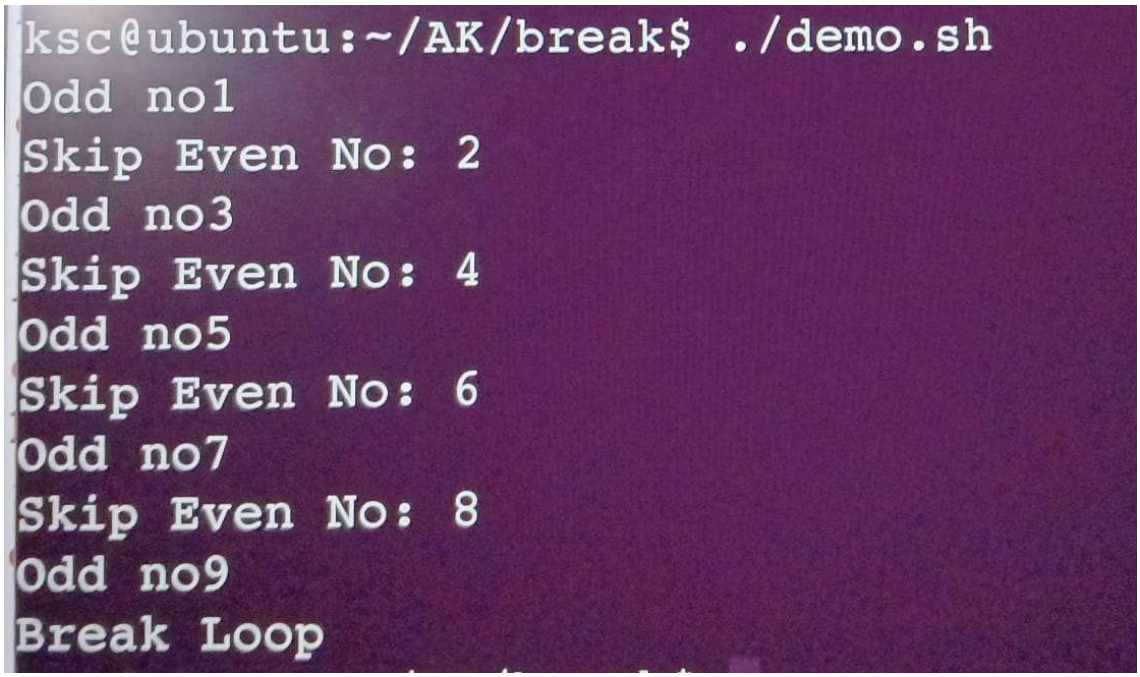
Also accepts a numeric argument for nested loops (e.g., continue 2 skips to the next iteration of the second enclosing loop).

1.sh 2.sh 3.sh 4.sh 1.sh

```
for i in {1..5}
do
echo "No: $i"
if [ $i -eq 4 ]
then
 echo "Break loop"
continue
fi
done
echo "Loop End"
```

```
ksc@ubuntu:~/AK/break$ chmod +x 1.sh
ksc@ubuntu:~/AK/break$ ./1.sh
No: 1
No: 2
No: 3
No: 4
Break loop
No: 5
Loop End
```

```
for i in {1..10}
do
#skip even no
if((i%2==0))
then
 echo "Skip Even No: "$i
 continue #skip loop body and go next iteration so won't
process the echo or break check after this
fi
echo "Odd no"$i
#stop loop when no = 9
if [ $i -eq 9 ]
then
echo "Break Loop"
break
fi
done
```



Array: An array in shell scripting (especially in Bash) is a variable that can hold multiple values under a single name, where each value is accessed using an index.

Think of it like a list of items — like a basket holding apples, bananas, and cherries — where each item has a position (index) you can refer to.

Type	Index Type	Declaration	Access Example
Indexed Array	Numeric (0, 1, 2...)	`array=(val1 val2 val3)`	`${array[0]}`
Associative Array	String (key-value)	`declare -A array`	`${array["key"]}`

Example 1: Index Array

leap.sh pyramid.sh pos.sh sum.sh user.sh tab.sh

```
fruits=("Apple" "Banana" "Mango" "Orange" "Cherry") #define
array
echo "All Fruits: ${fruits[@]}" #print all element
echo "Total: ${#fruits[@]}" #array length
```

```
ksc@ubuntu:~/AK/array$ ./pos.sh
All Fruits: Apple Banana Mango Orange Cherry
Total: 5
```

Example 2: sum of element

sum.sh

```
no=(10 20 30 40 50) #element
s=0 #use to collect the total
for n in "${no[@]}" #each element in the numbers array
do
((s+=n)) # s=s+10  s=s+20 and so on
done
echo "Numbers: ${no[@]}" #print all value in number array
echo "Sum: $s" #print total of number
```

```
ksc@ubuntu:~/AK/array$ ./sum.sh
Numbers: 10 20 30 40 50
Sum: 150
```

Example 3: Associative Array

sum.sh user.sh

```
declare -A cap
cap[Gujarat]="Gandhinagar"
cap[Haryana]="Chandigarh"
cap[Assam]="Dispur"
cap[Goa]="Panji"
cap[MP]="Bhopal"

echo "Capital List:"
for st in "${!cap[@]}"
do
echo "$st : ${cap[$st]}"
done
```

```
ksc@ubuntu:~/AK/array$ ./user.sh
Capital List:
Haryana : Chandigarh
Gujarat : Gandhinagar
MP : Bhopal
Assam : Dispur
Goa : Panji
```

Example 4:

sum.sh user.sh

```
declare -A cap
cap[Gujarat]="Gandhinagar"
cap[Haryana]="Chandigarh"
cap[Assam]="Dispur"
cap[Goa]="Panji"
cap[MP]="Bhopal"

echo -e "\nState\t\t|Capital"
echo "........................."
for st in "${!cap[@]}"
do
echo -e "$st\t\t|${cap[$st]}"
done
```

```
ksc@ubuntu:~/AK/array$ ./user.sh

State           |Capital
.........................
Haryana         |Chandigarh
Gujarat         |Gandhinagar
MP              |Bhopal
Assam           |Dispur
Goa             |Panji
```

Example 5: Tabular format data print

leap.sh pyramid.sh pos.sh sum.sh user.sh tab.sh

```
names=("ABC" "DEF" "GHI") #index array
ages=(12 17 22) #index array
#%-10s -> left align string with width 10 characters
printf "%-10s | %-5s\n" "Name" "Age"
echo "-------------------------"
#length of array names
for((i=0; i<${#names[@]}; i++))
do
#get name and match age
printf "%-10s | %-5s\n" "${names[$i]}" "${ages[$i]}"
done
```

```
ksc@ubuntu:~/AK/array$ ./tab.sh
Name       | Age
-------------------------
ABC        | 12
DEF        | 17
GHI        | 22
```

Function: A **function** is a named block of code that performs a specific task and can be reused whenever needed. Instead of writing the same code over and over again, you can define it once as a function and call it whenever required.

Example 1:

1.sh

```
add() #define function
{
sum=$(($1 + $2))
echo "Sum of $1 + $2 = $sum"
}
add 20 10
```

```
ksc@ubuntu:~/AK/Function$ ./1.sh
Sum of 20 + 10 = 30
```

Example 2:

```
user()
{
echo "Today's Date : $(date)"
}
echo "Enter your name: "
read nm
user
```

```
ksc@ubuntu:~/AK/Function$ ./2.sh
Enter your name:
Ayushi
Today's Date : Sun Apr  6 20:33:47 RET 2025
```

Example 3:

```
list()
{
echo "Files in $1:"
ls $1
}
list "/home/ksc"
```

```
ksc@ubuntu:~/AK/Function$ ./3.sh
Files in /home/ksc:
aa.sh        demo.sh          marksheet.sh  set1.sh
Add.sh       demo.txt         Music         set.sh
AK           Desktop          new file      student
```

Example 4:

```
count()
{
if [ -f "$1" ] #file exists or not
then
wc -w "$1" #count no of word
else
echo "Not found"
fi
}
count "/home/ksc/AK/Function/2.sh"
```

```
ksc@ubuntu:~/AK/Function$ ./4.sh
19 /home/ksc/AK/Function/1.sh
```