Unit 3.
Shell Programming

3.1 What is shell programming:

The shell provides you with an interface to the UNIX system. It gathers input from you and executes programs based on that input.
When a program finishes executing, it displays that program's output.

A shell is an environment in which we can run our commands, programs, and shell scripts. There are different flavors of shells, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions.

VI Editor - Introduction

There are many ways to edit files in Unix. Editing files using the screen-oriented text editor **vi** is one of the best ways. This editor enables you to edit lines in context with other lines in the file.
An improved version of the vi editor which is called the VIM has also been made available now. Here, VIM stands for Vi IMproved.

vi is generally considered the de facto standard in Unix editors because −

- It's usually available on all the flavors of Unix system.
- Its implementations are very similar across the board.
- It requires very few resources.
- It is more user-friendly than other editors such as the ed or the ex.
- You can use the vi editor to edit an existing file or to create a new file from scratch. You can also use this editor to just read a text file.

- vi stands for Vim Improved.
- Vi uses number of internal commands to navigate to any point in a text file and edit text there.
- It also allows us to copy and move the text within the file and also from one file to another.
- We can use internal commands for editing work.
- It makes complete use of keyboard where practically every key has a function.
- **To create any vi file**,

  $ vi <some text or filename>
- If file doesn't exist, vi provides us a full screen with the filename shown at the bottom with the qualifier [New File].
- **The cursor is positioned at top and all remaining lines of the screen (Except last) show a ~. We can't take cursor there because they are nonexistent lines.**
- The last line is reserved for commands that we can enter to act on the text. This line is also used by the system to display the message.

General Command Information

As mentioned previously, vi uses letters as commands.
It is important to note that in general vi commands:

- are case sensitive - lowercase and uppercase command letters do different things
- are not displayed on the screen when you type them
- Generally do not require a Return after you type the command.

You will see some commands which start with a colon (:). These commands are ex commands which are used by the ex editor. Ex is the true editor which lies underneath vi -- in other words, vi is the interface for the ex editor.

Limitations of VI editor

vi editor has weak environment s there are some disadvantages of it as below:

1. vi is case-sensitive.
2. It don't display error message when something is going wrong. At that moment, only beep sound of speaker which informs you there is something wrong.
3. There is no online help available in vi.
4. It works on 3 different modes. Same keys can create different effect on each mode.

VI Editor - MODES

While working with the vi editor, we usually come across the following two modes −

- **Command mode −**

  By default, any file open in vi is in command mode.

  This is default mode of vi.

  This mode enables you to perform administrative tasks such as saving the files, executing the commands, moving the cursor, cutting (yanking) and pasting the lines or words, as well as finding and replacing. In this mode, whatever you type is interpreted as a command.

  When user is in command mode and press Esc key then internal speaker of terminal will beep.

  This is a mode where we can pass commands to act on text, using keys of the keyboard. Pressing a key doesn’t show it on screen but may perform a function like moving cursor to the next line or deleting a line.
  We can’t use Command Mode to enter or replace text.

- **Insert mode −**

  This mode enables you to insert text into the file.

  Everything that's typed in this mode is interpreted as **input and placed in the file**.

  When you are in insert mode the same letters of the keyboard will type or edit text.

  For text editing, vi uses 24 of the 25 lines that are normally available in a terminal.

To enter text, we must switch to Input Mode. First press key marked i and we are ready to input text.

We can start inserting a few lines of text followed by [enter].

[Notes: vi always starts out in command mode.
When you wish to move between the two modes, keep these things in mind.
You can **type i** to enter the insert mode.
If you wish to leave insert mode and return to the command **mode, hit the ESC key.**
If you're not sure where you are, hit ESC a couple of times and that should put you back in command mode.]

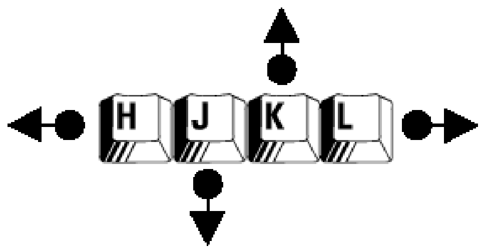


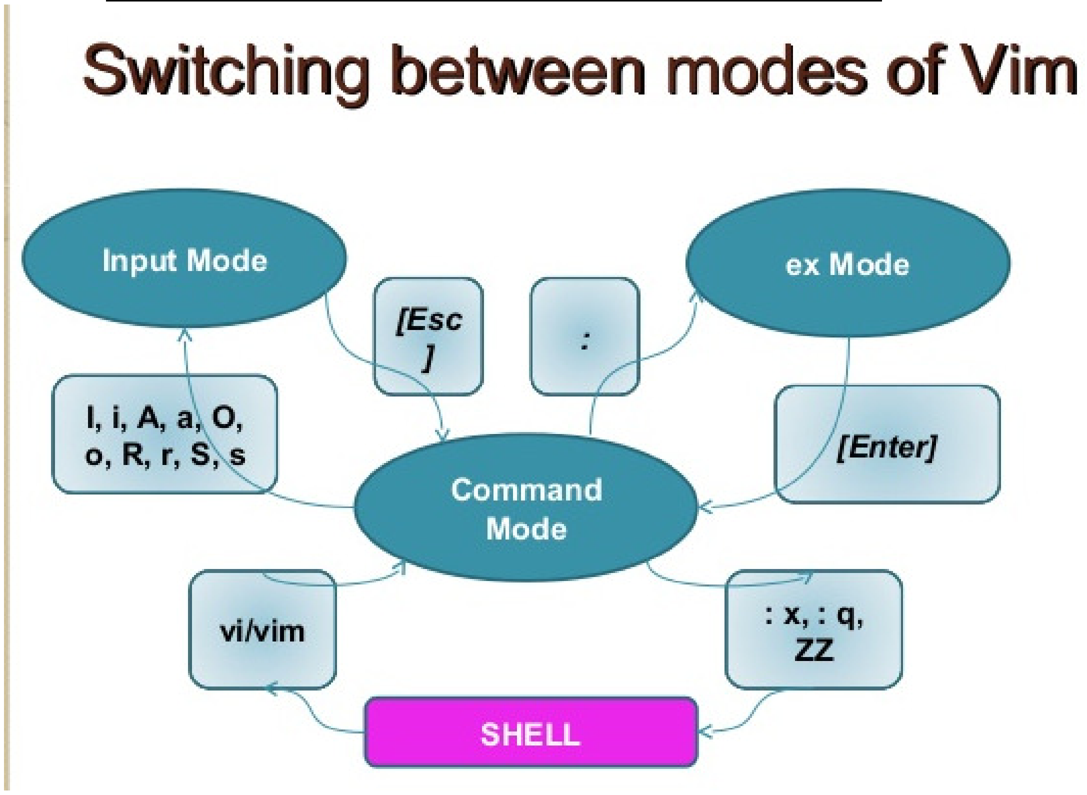
Table 7.1: Summary of `vi` Commands.

`vi` mode	`input` mode	`command` mode
	< ESC > to end input	
← ↓ ↑ → - cursor	i - insert	:q - quit
h j k l - cursor	a - append	:q! - quit no save
CTL-f - forward screen	A - append at EOL	:w - write
CTL-b - backward screen	O - open line	:wq - write and quit
G - end of file	r - replace character	:*num* goto line *num*
x - delete character	R - overwrite	/*str* - find *str*
dw - delete word		:set all - vi settings
dd - delete line		:r *file* - import file
yy - copy line in buffer		
D - delete to EOL		
p - paste/put buffer		
u - undo last command		
CTL-r - redo last undo (linux/`vim`)		
. - repeat last editing command		
n - find next occurrence of string		
cw - change word		
# command - repeate command # times		

- **Last Line mode –**
  The last vi mode is known as vi last line mode.
  After text entry is complete, the cursor is positioned on the last character of the last line.
  This is known as **Current Line** and the character is stationed is the Current Cursor Position.
  You can only get to last line mode from command mode, and you get into last line mode by pressing the colon key, like this:
  ':'
  After pressing this key, you'll see **a colon character** appear at the beginning of the last line of your vi editor window, and your cursor will be moved to that position.
  This indicates that vi is ready for you to type in a "last line command".
  If you want to remove something then can use [Backspace] key.

  If a word has been misspelled then use [Ctrl+w] to erase entire word.
  Press [Esc] key to revert to Command Mode.
  The text that you entered hasn't been saved on disk yet. The entered text exists in some temporary storage called Buffer.
  To save the entered text, we must switch to the ex mode or Last Line Mode (3rd mode of VI).
  To invoke ex Mode from the Command Mode by entering a colon (:) which shows on last line. Press x and then [Enter].
  Now, the file is stored on the disk and vi returns shell prompt.
  To modify this file, again write vi <some text or filename> again.

Representation the 3 different modes of vi-editor.



Inserting or Adding Text

The following commands allow you to insert and add text. Each of these commands puts the vi editor into insert mode; thus, the <Esc> key must be pressed to terminate the entry of text and to put the vi editor back into command mode.

*	`i`	*insert text before cursor, until <Esc> hit*
	`I`	*insert text at beginning of current line, until <Esc> hit*
*	`a`	*append text after cursor, until <Esc> hit*
	`A`	*append text to end of current line, until <Esc> hit*
*	`o`	*open and put text in a new line below current line, until <Esc> hit*
*	`O`	*open and put text in a new line above current line, until <Esc> hit*
	rch	*replace single character under cursor (no* <Esc> *needed)*
	R	*replace characters, starting with current cursor position, until* <Esc> *hit*
	s	*Replaces the current character with the character you type. Afterward, you are left in the insert mode.*
	S	*Deletes the line the cursor is on and replaces it with the new text. After the new text is entered, vi remains in the insert mode.*

Last-line mode command:

The purpose of these commands is to save a file, exit from a file with or without saving it and so on. Sometimes it is also known as file handling commands.

Commands	Significance
:wq or :x	It saves a file, quiets from vi editor and return to shell prompt.
:q	It quits from the editing mode, when no changes are made to file, and return to shell prompt.
:w	It saves a file and remain in editing mode, i.e. stay in vi editor
:q!	It quits from the editing mode, without changes are made to file, and return to shell prompt.
:w file1	It saves buffer content into file1.
:w! file1	As above but over write existing file.
:w>>file1	It appends current/open file contents into file file1.
:n1,n2w file1	It writes n1 to n2 lines of open file into file file1.
:.w file1	It writes current line of open file into file file1. here (.) stands for current line of open file.
:$w file1	It writes last line of open file into file file1. here ($) stands for last line of open

	file.
:sh	It temporary exit from vi and return to shell prompt. type <ctrl+d> to return back to URL.
Ctrl-z	It suspends current session and escape to unix shell. type fg at shell prompt to return back to URL.

Command mode command:

VI editor supports two types of command mode commands:

- **Cursor movement commands**
  The purpose of these commands is to perform navigate or to move cursor from one place to another place in an editor. The VI editor has its own set of commands.

Commands	**Significance**
H or (backspace)	It moves cursor one-character left.
j	It moves cursor one-line down.
k	It moves cursor one-line up.
l OR (cursor)	It moves cursor one-character right.
[return]	It moves cursor to the beginning of the next line.
$	It moves cursor last column/character (or end of line) on the current lines.
0(zero) or | or ^	It moves cursor first column/character (or end of line) on the current lines.
w	It moves cursor forward to the beginning of the next word or punctuation mark.
b	It moves cursor to the beginning of the previous word or punctuation mark.
e	It moves cursor forward to end of next word or punctuation mark.
G	It moves cursor to the begging of last line in a file.

example :

Movement

h, j, k, l	left, down, up, right
$	To the end of the line
^	To the beginning of the line
G	To the end of the file
:1	To the beginning of the file
:47	To line 47

- **page scroll commands**
  Sometimes a user wish to move from one page to another page in forward or backward direction then scrolling command is used.

Commands	**Significance**

Ctrl-f	It scrolls full page forward.
Ctrl-b	It scrolls full page backward.
Ctrl-d	It scrolls half page forward.
Ctrl-u	It scrolls half page backward.
Ctrl-l	It redraw a screen.

example:
5ctrl-f :it scrolls 5-full pages forward.
5ctrl-b :it scrolls 5-full pages backward.
2ctrl-d :it scrolls 2-half pages or 1 page forward

VI Editor - OPERATORS

A user can operates various tasks like delete, copy, paste, and search and replace a text.
Operator is a single letter command that performs an action on the text described by the object.
The operators (to be described below) are:

1. **d deletion operator**
2. **c change operator**
3. **y yank (copy) operator**
4. **! filter operator**

Editing Commands:
A user can use above operators with commands to perform editing operations in a file that is opened in an editor.
To edit the file, you need to be in the insert mode.

Command & Description
X:Deletes the character under the cursor location
X:Deletes the character before the cursor location
J: it joins current line with next line.
Dw: Deletes word from the current cursor location .
d^: Deletes from the current cursor position to the beginning of the line
D4g or 4dG: it deletes character from cursor position to 4th line.
dfch : it deletes character from cursor position to 1st occurrences of character 'ch'.
d/str: it delete characters from cursor position to 1st occurrences to string 'str' in forward direction.

D?str: it delete characters from cursor position to 1st occurrences to string 'str' in backward direction.
D or d$: Deletes character from the cursor position to the end of the current line
Dd: Deletes the line the cursor on current line.
Cc: Removes the contents of the line, leaving you in insert mode.
Cw: Changes the word the cursor is on from the cursor to the lowercase w end of the word.
c$ or C: it changes the character from the cursor position to the end of the current line
c4g or 4cG: it changes character from cursor position to 4th line.
r: Replaces the character under the cursor. vi returns to the command mode after the replacement is entered.
R: Overwrites multiple characters beginning with the character currently under the cursor. You must use Esc to stop the overwriting.
S: Replaces the current character with the character you type. Afterward, you are left in the insert mode.
S: Deletes the line the cursor is on and replaces it with the new text. After the new text is entered, vi remains in the insert mode.
Yy oy Y: Copies the current line.
Yw: Copies the current word from the character the lowercase w cursor is on, until the end of the word.
y$: it yanks character from cursor position to end of line.
Y4G or 4Gy: it copy character from cursor position to 4th line.
Yfch: it yanks cursor position to 1st occurrences of character 'ch' in forward direction.
y/str: it yanks from cursor position to 1st occurrences to string 'str' in reverse direction.

y?str: it delete characters from cursor position to 1st occurrences to string 'str' in forward direction.
P: Puts the copied text after the cursor.
P: Puts the yanked text before the cursor.
~ : it reverse the case of character under the cursor position.
. (dot): it repeats last ending instruction.
u: it undoes last ending instructions
U: it undoes all changes made in current line. **Ctrl-r : it redoes previous undo.**

<u>Search and replacing Commands:</u>

Command & Description
/pat : It search pattern *pat* in forward direction. **?pat : It search pattern *pat* in backward direction.** **n : it repeats search in same directions along with which previous search was made.** **N: it repeats search in opposite directions along with which previous search was made.** **fch : it removes cursor forward to 1st occurrences of character 'ch' in current line.** **Fch : it removes cursor backward to 1st occurrences of character 'ch' in current line.** **tch : it moves cursor forward but before 1st occurrences of character 'ch' in current line.** **Tch : it moves cursor backward but before 1st occurrences of character 'ch' in current line.** **;(semi-colon) : it repeats the search in the same direction made with f,F,t or T command.** **, (comma) : : it repeats the search in the opposite direction made with f,F,t or T command.** **:n1,n2 s/s1/s2: it replace 1st occurrences of string or regular expression s1 with string s2 in line n1 to n2.** **:n1,n2 s/s1/s2/g: it replace all occurrences of string or regular expression s1 with**

string s2 in line n1 to n2. **:s : it repeats the last substitution on the current line.**

Handling multiple files:
VI uses last line mode to handle multiple files and buffers.
A user can open as many buffers as required and switch from one to other.
The commands are as below for multiple file handling.

Command & Description
: r fname : it reads(/insert) file fname **below the current line.** **:r !cmd : it reads(/insert) output of 'cmd' command below the current line.** **:e fame: it stop editing current file, and edits the file fname.** **:e !fname: it is similar to :e fname but after discarding chsnges made to current file.** **:e! : it loads the last saved edition of current file.** **Ctrl-^ or :e# : it returned most recently edited file.** **:n : it edits next file.** **:n! : it permits editing of next file without saving the current file.** **:rew : it rewinds the file list to start editing first file.** **:rew! : it permits editing to the 1st file in command line without saving the current file.** **:f : it displays name of the current file.** **:args : it displays name of all files in the buffer, the name of the current file is enclosed within square brackets.** **:sp : it splits the existing window into 2 separate windows.** **Ctrl-w/W : it cycles through window.**

Customized VI Editor:
A user can customize environment of VI editor. For that last line mode commands are used.

Command & Description
:set ic/ :set ignorecase -Ignores the case when searching **:set noic/ :set noignorecase -it does not**Ignores the case when searching

:set ai/ :set autoindent -Sets auto indent **:set noai-**Unsets autoindent
:set nu/ :set number-Displays lines with line numbers on the left side **:set nonu/ :set nonumber-does not** Displays lines with line numbers on the left side
:set showmode -it display mode in which user working. **:set noshowmode -**it does not display mode in which user working:
:set aw/autowrite -it automatically writes buffer contents to disk before switching to next file during multiple file handling. **:set noaw/noautowrite -**it dont writes buffer contents to disk before switching to next file during multiple file handling.

3.2 Environmental & user defined variables

Variable Names:

The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (_). By convention, Unix shell variables will have their names in UPPERCASE. It is case sensitive. you cannot use special characters.

Two types of variables in unix : **User defined variables and system/environmental varisbles.**

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!
```

The reason you cannot use other characters such as !, *,&,$ or - is that these characters have a special meaning for the shell.

1. User defined variables: A variable defined by user known as user defined variables.

Defining Variables : Variables are defined as follows −

```
variable_name=variable_value/expression/command
```

For example −:

```
$myvar = 1234                          #No space on either side of equal sign
```

By default any variable created in shell are of **string type.** so, the values are stored in ASCII rather than binary.

variables created in shell are automatically removed as soon as shell is expired.

when shell reads the command line, it interprets any word preceded by $ as a variable and replace the word by the value of the variable.

To display content , a user can use **echo command**:

$echo $myvar <enter>

ans: 1234

$

[note: LINK Scanned documnet : variables]

Shell enables you to store any value you want in a variable. For example −

```
VAR1="Zara Ali"

VAR2=100
```

Accessing Values

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

Shell provides a way to mark variables as read-only by using the read-only command. After a variable is marked read-only, its value cannot be changed.

For example, the following script generates an error while trying to change the value of NAME

```
#!/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.
```

3.4 Shell's interpretation at prompt:

when user enters a command on command line shell perform some internal steps before the command directs its execution to a utility or s program.
Following are the steps:

1. **PARSING:** in this stage, the shell divides the command line into words, if it is not quoted or escaped. The shell uses system variable IFS (internal field separator) to delimiter the value of word delimiter, which is space and tab.
   When shell encounters 2 or more spaces or tabs in the command line then they are replaced with a single space.
   for example, a=10
   $echo value   of   variable   a=$a <enter> //multiple spaces

   $\downarrow$ parsing

   $ echo value of variable a=$a //spaces are replaced.

2. **Variable Evaluation:** in this stage, the shell is looking for variable names.
   for example, a=10
   $echo value of variable a=$a <enter> //$a is consider as variable

   $\downarrow$ variable evaluation

   $ echo value of variable a=10 //value is assigned.

3. **Command Substitution:** Any Unix command enclosed within back quotes is executed by the shell and its output is substituted as argument for another command into a command line.
   E.g.: echo "Today's date: 'date'" // date command is enclosed within back quote ' '

   $\downarrow$ Command Substitution

   echo "Today's date : MON Feb 21 14:03:03 IST 2018"
   // The output of date command is become an argument for echo command.

4. **Redirection:** in this stage, shell scans for the characters >,< and >> and open a file associated with this characters in a particular mode.

   e.g.: wc <f1 here, shell opens a file instead of command and display number of lines, words and character in file f1.

5. **Wild-card interpretation:** A number of characters are interpreted by the Unix shell before any other action takes place.
   These characters are known as wildcard characters.
   Usually these characters are used in place of filenames or directory names.

*	An asterisk matches any number of characters in a filename, including none.
?	The question mark matches any single character.
[ ]	Brackets enclose a set of characters, any one of which may match a single character at that position.
-	A hyphen used within [ ] denotes a range of characters.
~	A tilde at the beginning of a word expands to the name of your home directory. If you append another user's login name to the character, it refers to that user's home directory.

Here are some examples:

1. cat c* displays any file whose name begins with c including the file c, if it exists.
2. ls *.c lists all files that have a .c extension.
3. cp ../rmt?. copies every file in the parent directory that is four characters long and begins with rmt to the working directory. (The names will remain the same.)
4. ls rmt[34567] lists every file that begins with rmt and has a 3, 4, 5, 6, or 7 at the end.
5. ls rmt[3-7] does exactly the same thing as the previous example.
6. ls ~ lists your home directory.
7. ls ~hessen lists the home directory of the guy1 with the user id hessen.

6. **PATH evaluation:** finally, it examine the system variable PATH to search a command into the sequence of directories stored in it. If it is found then load an executable of command into memory and then execute it otherwise it sows error message on standard output.

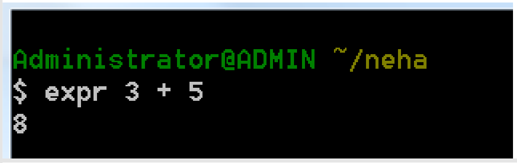
3.5 Arithmetic expression evaluation

1. **expr :** this command has 2 functions : it performs arithmetic operations on integer and string manipulation on strings.
   Syntax: expr [expression]

   expr prints the value of EXPRESSION to standard output. A blank line below separates increasing precedence groups.

EXPRESSION may be:

ARG1 | ARG2	**ARG1 if it is neither null nor 0, otherwise ARG2.** **$x=3;y=5** **$expr $x\\\|$y ans: 3**
ARG1 & ARG2	ARG1 if neither argument is null or 0, otherwise 0.
ARG1 < ARG2	ARG1 is less than ARG2.
ARG1 <= ARG2	ARG1 is less than or equal to ARG2.
ARG1 = ARG2	ARG1 is equal to ARG2.
ARG1 != ARG2	ARG1 is unequal to ARG2.
ARG1 >= ARG2	ARG1 is greater than or equal to ARG2.
ARG1 > ARG2	ARG1 is greater than ARG2.

	$x=3;y=5 **$expr $x\>$y ans: 0**
ARG1 + ARG2	arithmetic sum of ARG1 and ARG2. $x=3;y=5 **$expr $x + $y ans: 8**  Administrator@ADMIN ~/neha $ expr 3 + 5 8
ARG1 - ARG2	arithmetic difference of ARG1 and ARG2. $x=3;y=5 **$expr $x - $y ans: -2**
ARG1 * ARG2	arithmetic product of ARG1 and ARG2. $x=3;y=5 **$expr $x*$y ans: 15**
ARG1 / ARG2	arithmetic quotient of ARG1 divided by ARG2. $x=3;y=5 **$expr -5/2 ans: -2**
ARG1 % ARG2	arithmetic remainder of ARG1 divided by ARG2. $x=3;y=5 **$expr -3%5 ans: 3** **Or** $x=3;y=5 **$expr 3%-5 ans: 5**
STRING : REGEXP	anchored pattern match of regular expression REGEXP in STRING.
match STRING REGEXP	same as STRING : REGEXP.
substr STRING POSLENGTH	substring of STRING, POS counted from 1.
index STRING CHARS	index in STRING where any CHARS is found, or 0.
length STRING	length of STRING.
+ TOKEN	interpret TOKEN as a string, even if it is a keyword like 'match' or an operator like '/'.
(EXPRESSION)	value of EXPRESSION.

<u>String manipulation Expression</u>

Length STRING	Return the length of the string
Substr STRING POS LEN	It returns LEN-characters of STRING from POS position.
Imdex STRING CHARS	It returns the index of CHARS if found in a string otherwise 0.

Example:

$ expr length "TYBCA"   ans :5
$ expr substr "TYBCA" 3 2   ans:BCA
$ expr substr "TYBCAmkics" 0 6   // display nothing
$ expr substr "TYBCAmkics" 7 12   ans:kics
$ expr substr "TYBCAmkics" -1 6   // display nothing
$ expr index "TYBCAmkics" A   ans:5

2. <u>bc command: IT stands for basic calculator.</u>

SYNTAX :bc [-options] [*file ...*] OPTIONS

-h, --help :Print the usage and exit.
-i, --interactive: Force interactive mode.
-l, --mathlib: Define the standard math library.
-w, --warn: Give warnings for extensions to POSIX **bc**.
-s, --standard: Process exactly the POSIX **bc** language.
-q, --quiet: Do not print the normal GNU bc welcome.
-v, --version: Print the version number and copyright and quit.

It allows to perform various tasks.

- Arithmetic calculations on integers as well as on real numbers.
- conversion of numbers from one base to another.(i.e. decimal to binary and so on.)
- calculate square roots, logarithms and trigonometric calculus such as sin, cos etc.

bc commands work on modes [1] interactive mode [2] command-line mode.

<u>[1] Interactive mode :</u> when you invoke bc without any option or argument, it display some header lines and cursor keeps on blinking and nothing seems to happen. this is the interactive mode of bc command.

```
$bc <enter>
#----------display some header lines---------
12 + 5 <enter>
17
<ctrl+d> or quit or halt
$
```

- **<u>bc uses four special variables : scale, ibase, obase and last.</u>**

I. **scale:**

✦ By default, bc performs integer division so that result will be integer, the fractional part of the result is truncated.

✦ To show a fraction part, a user has to set a special variable called scale.

```
scale=2 <enter>
17/7 <enetr>\
2.42
```

II. **ibase (input base) :**

✦ bc is quit useful in converting numbers from one base to another, set ibase before you provide the number.

```
ibase=2 <enter>   #default input and output base is  10(decimal)
101 <enter>    #input is binary number
5              #output is in decimal base
```

III. obase (output) :

- ✦ it defines conversion base of output.

```
obase=2 <enter>          #now output base is binary
5 <enter>                #input base is decimal
101                      #binary of decimal
```

IV. last :

- ✦ it contains value of last printed number.

```
4+5 <enter>
9
last <enter>
9
```

- bc also supports in-built functions like sqrt, cosine, sine, tangent, exponent etc.
- To use these function in bc, a user has to pre-load a math library using –l option. i.e. bc -l.
- This library has default scale to 20. table shows the list of functions:

Table : math function

Function	**Meaning**
s(x)	it calculates sine of x, where x is in radian.
c(x)	it calculates cosine of x, where x is in radian.
a(x)	it calculates arctangent of x, where x is in radian.
l(x)	it calculates natural logarithm of x.
e(x)	it calculates e raise to x, where e=2.718281.
sqrt(x)	it calculates square root of x.

```
$bc -l
sqrt(4)
2
sqrt(11)
3
e(l)
2.71828182845904523536
scale=2
sqrt(11)
3.31
```

- **option used with bc command are as follow:**
  (i) -h : it display usage and exit from the bc.
  (ii) -l : it defines standard math library functions.
  (iii) -q : it suppresses initial messages i.e. header information that is displayed during interactive mode.

<u>**[2] command line mode : A**</u> user can also execute bc command at command-line.
in this mode, argument file contains numbers or expressions. for example, consider an input file is as follow:

```
$ cat num
s(45*3.14/180) //convert into degree
```

```
scale=2
4/3
sqrt(5)
e(l)
1.5+4
quit

$ bc -lq num
.70682518110536592374
1.33
2.33
2.71
5.5
$
```

- ❖ Here, num file contain expression and functions.
- ❖ if you apply a command at prompt, be executes commands line-by-line and display its output on a screen.
- ❖ last statement quit is used to exit from bc.
- ❖ A user can assign output of bc command to another variable like this:

```
$ a=`echo "scale=2;5/3"|bc`
$ echo $a                    //ans:1.66
```

- ❖ **A user can convert decimal number into octal at command line as follow:**

```
 $echo "obase=8;12" | bc
14                              #converts to octal value
$
```

Note: in merged notes 'be' is written in place of bc,pls check it

3. **factor :** It is math related command. [link : facor(bharat)]

- Works in interactive or command line mode.

  SYNTAX: **Factor [NUMBER(S)]**
- If positive number given as input, less than 2^46,it will factorize the number and display prime factors.
- Use **<ctrl+d>** to come out of interactive mode
- If no **NUMBER** is specified on command line, "factor" reads numbers from standard input delimited by space, tabs or new line.

e.g.: $ factor

```
12                    # takes std. input,until <ctrl+d>
12: 2 2 3
35
35: 5 7
```

```
42
42: 2 3 7
100
100: 2 2 5 5
13 10           # Takes 2 inputs at a time
13: 13
10: 2 5
```

$ factor 12 10 9 //factor command works on command line like this

```
12: 2 2 3
10: 2 5
9: 3 3
```

4. **units:**

- Converts quantities expressed in one scale to its equivalents in other scale.

  Syntax: units[FROM-UNIT[TO-UNIT]]

- In absence of from unit and to unit, the program will use interactive mode.
- "units" command will not work in bash shell, as:
  $ units
- -bash: units: command not found.
- Example:

```
$ units
you have:10 meter    # at you have prompt, give input
you want:feet        # at you want prompt gives wanting unit
*32.808399           #output shown in interactive mode
/0.03048
you have:            # after output input prompt again will come
```

first line of output: 10 meters equals to 32.8 feet
second line of output: 1 foot equals to 0.03 decameter(10 meter)

- ➔ (ctrl+d) is used to quit from interactive mode of units command.
- ➔ Here both units are given as:

```
$ units  10meters  feet
*32.808399
/0.03048
$
```

- ➔ if only one unit given with command then ,definition of the given unit will be displayed on the output screen.

3.5 Control Structure:

control structure alters the flow of execution of shell script. it supports 2 ways of control structure.

1. **Decision making &**
2. **Loop control**

1. **Decision making : It contains 2 decision making statements.**

1.1. If statement

If else statements are useful decision-making statements which can be used to select an option from a given set of options.

Unix Shell supports following forms of if…else statement –

- **if-then-fi statement**
- **if-then-else-fi statement**
- **if-then-elif-else-fi statement**

Syntax :
```
If  control-command
           then
           statements
      fi
```

here if,then and fi are keywords.

example :
```
#script name :sh1
echo -e "Enter name of first file :\c"
read fname1
echo -e "Enter name of second file :\c"
read fname2

if cmp -s $fname1 $fname2
then echo "file contents are similar"
fi
```

Other forms,

Double decision:

```
if <condition>
then
   ### series of code if the condition is satisfied
else
   ### series of code if the condition is not satisfied
fi
```

example :
```
#script name :sh2
echo -e "Enter name of first file :\c"
read fname1
echo -e "Enter name of second file :\c"
read fname2
```

```
if cmp -s $fname1 $fname2
then
echo "file contents are similar"
else
echo "file contents are not similar"
fi
```

Multiple if condition:

```
if <condition1>
then
  ### series of code for condition1
elif <condition2>
then
  ### series of code for condition2
else
  ### series of code if the condition is not satisfied
fi
```

1.2. case-esac statement:

You can use multiple if...elif statements to perform a multiway branch.

However, this is not always the best solution, especially when all of the branches depend on the value of a single variable.

Unix Shell supports **case...esac** statement which handles exactly this situation, and it does so more efficiently than repeated if...elif statements.

There is only one form of **case...esac** statement which has been described in detail here −

case...esac statement

The case...esac statement in the Unix shell is very similar to the switch...case statement we have in other programming languages like C or C++ and PERL, etc.

Syntax of bash case statement.

```
case expression in
  label1 )
    statements ;;
  label2 )
    statements ;;
  ...
esac
```

Following are the key points of bash case statements:

- Case statement first expands the expression and tries to match it against each label.
- When a match is found all of the associated statements until the double semicolon (;;) are executed.

- After the first match, case terminates with the exit status of the last command that was executed.
- If there is no match, exit status of case is zero.

example:

```
echo "enter no."
read num
 case $num in
         1) echo "one"
                 ;;
         2) echo "Two"
                 ;;
                 ...
         9)echo "Nine"
                 ;;
          *) echo "please ,enter a number between 1 and 9"
          ;;
esac
```

2. Loop control structure : It contains 3 control structures.

2.1. while loop :To repeat a part of program fixed number of times then while loop is used.

Syntax:

```
while command1 ;              # this is loop1, the outer loop
do
  Statement(s)      //to be executed if command1 is true
done
```

```
Example:

a=0
while [ "$a" -le 10 ]            # this is loop1
do
    echo $n
     n=`expr $n + 1`
  done
Ans: by this you can display 1 to 10
```

2.2. until loop : It is a complement loop of while.

The while loop is perfect for a situation where you need to execute a set of commands while some condition is true. Sometimes you need to execute a set of commands until a condition is true.

```
Syntax:
until command
do
  Statement(s)                          //to be executed until command is true
done
```

Here the Shell command is evaluated. If the resulting value is false, given statement(s) are executed. If the command is true then no statement will be executed and the program jumps to the next line after the done statement.

Example

Here is a simple example that uses the until loop to display the numbers zero to nine −

```
n=1
until [  $n -gt 10 ]
do
  echo $n
  n=`expr $a + 1`
done
```

Answer:

Upon execution, you will receive the following result −

```
0
1
2
3
4
5
6
7
8
9
```

2.3. for loop : The for loop operates on lists of items. It repeats a set of commands for every item in a list.

Syntax

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

Here var is the name of a variable and word1 to wordN are sequences of characters separated by spaces (words). Each time the for loop executes, the value of the variable var is set to the next word in the list of words, word1 to wordN.

Example

Here is a simple example that uses the for loop to span through the given list of numbers −

```
for i in 0 1 2 3 4
do
  echo $i
done
```

Upon execution, you will receive the following result −

```
0
1
```

```
2
3
4
```

3. **Un conditional jump Statements : Break & Continue.**

3.1 break:

The break statement is used to terminate the execution of the entire loop, after completing the execution of all of the lines of code up to the break statement. It then steps down to the code following the end of the loop.

Syntax

The following break statement is used to come out of a loop −

```
break
```

The break command can also be used to exit from a nested loop using this format −

```
break n
```

Here n specifies the nth enclosing loop to the exit from.

Example

Here is a simple example which shows that loop terminates as soon as a becomes 5 −

```
a=0

while [ $a -lt 10 ]
do
  echo $a
  if [ $a -eq 5 ]
  then
    break
  fi
  a=`expr $a + 1`
done
```

3.2 Continue :

The continue statement is similar to the break command, except that it causes the current iteration of the loop to exit, rather than the entire loop.
This statement is useful when an error has occurred but you want to try to execute the next iteration of the loop.

Syntax

```
continue
```

Like with the break statement, an integer argument can be given to the continue command to skip commands from nested loops.

```
continue n
```

Here n specifies the nth enclosing loop to continue from.

Example

The following loop makes use of the continue statement which returns from the continue statement and starts processing the next statement −

```
NUMS="1 2 3 4 5 6 7"

for NUM in $NUMS
do
  Q=`expr $NUM % 2`
  if [ $Q -eq 0 ]
  then
    echo "Number is an even number!!"
    continue
  fi
  echo "Found odd number"
done
```

Upon execution, you will receive the following result −

```
Found odd number
Number is an even number!!
Found odd number
Number is an even number!!
Found odd number
Number is an even number!!
Found odd number
```

Logical Operators

This operators are used to join conditions.
shell offers 3 types of logical operators.

1. -a (Logical AND) : used to join 2 or more conditions.
2. -0 (Logical OR) : also, used to join 2 or more conditions.
3. -! (Logical NOT) : negates the value of expression. i.e. : it makes non-zero to zero and vise-versa.

test Command

it is used to evaluate expression and returns either true or false exit status which is used by if statement to make decision.
There are 3 major functionalities of test command.

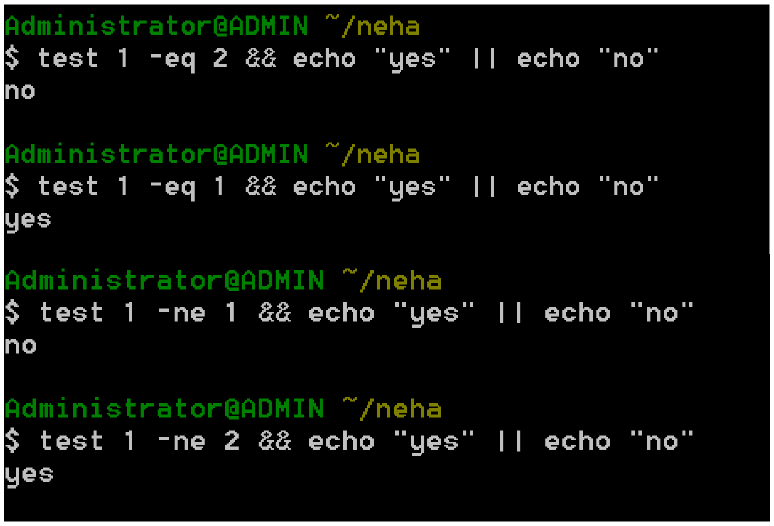
1. compare 2 numbers
2. compare 2 strings or a single or null value
3. checks a file attributes

1. **Numeric comparison: It performs comparison between 2 nos.**

    -eq - does value 1 equal value 2
    -ge - is value 1 greater or equal to value 2
    -gt - is value 1 greater than value 2
    -le - is value 1 less than or equal to value 2
    -lt - is value 1 less than value 2
    -ne - does value 1 not equal value 2

    Examples:

test 1 -eq 2 && echo "yes" || echo "no"
(displays "no" to the screen because 1 does not equal 2)



```
Administrator@ADMIN ~/neha
$ test 1 -eq 2 && echo "yes" || echo "no"
no

Administrator@ADMIN ~/neha
$ test 1 -eq 1 && echo "yes" || echo "no"
yes

Administrator@ADMIN ~/neha
$ test 1 -ne 1 && echo "yes" || echo "no"
no

Administrator@ADMIN ~/neha
$ test 1 -ne 2 && echo "yes" || echo "no"
yes
```

test 1 -ge 2 && echo "yes" || echo "no"
(displays "no" to the screen because 1 is not greater or equal to 2)

test 1 -gt 2 && echo "yes" || echo "no"
(displays "no" to the screen because 1 is not greater than 2)

test 1 -le 2 && echo "yes" || echo "no"
(displays "yes" to the screen because 1 is less than or equal to 2)

test 1 -lt 2 && echo "yes" || echo "no"
(displays "yes" to the screen because 1 is less than or equal to 2)

test 1 -ne 2 && echo "yes" || echo "no"
(displays "yes" to the screen because 1 does not equal 2)

2. String comparison:

If you are comparing elements that parse as strings you can use the following comparison operators:

- = - does string 1 match string 2
- != - is string 1 different to string 2
- -n - is the string length greater than 0
- -z - is the string length 0

Examples:
test "string1" = "string2" && echo "yes" || echo "no"
(displays "no" to the screen because "string1" does not equal "string2")

test "string1" != "string2" && echo "yes" || echo "no"

(displays "yes" to the screen because "string1" does not equal "string2")

test -n "string1" && echo "yes" || echo "no"
(displays "yes" to the screen because "string1" has a string length greater than zero)

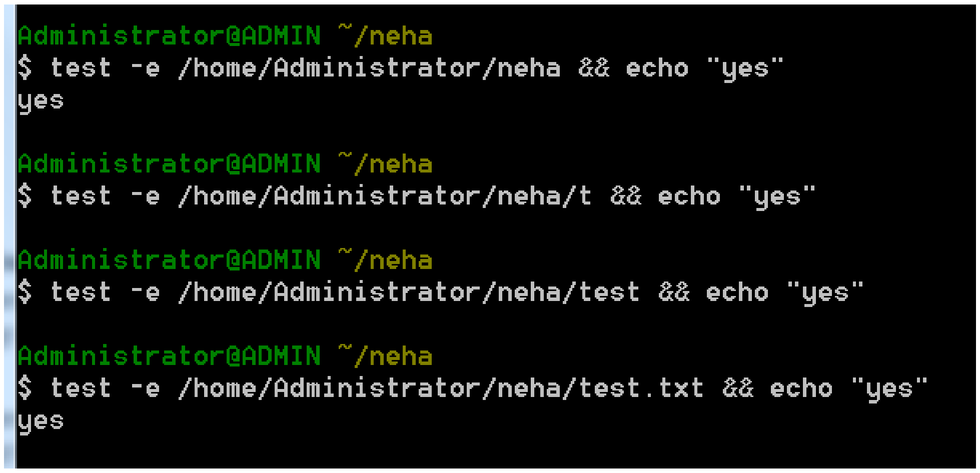
test -z "string1" && echo "yes" || echo "no"
(displays "no" to the screen because "string1" has a string length greater than zero)

3. checks a file attributes :

If you are comparing files you can use the following comparison operators:

- -ef - Do the files have the same device and inode numbers (are they the same file)
- -nt - the first file newer than the second file
- -ot - the first file older than the second file
- -b - The file exists and is block special
- -c - The file exists and is character special
- -d - The file exists and is a directory
- -e - The file exists
- -f - The file exists and is a regular file
- -g - The file exists and has the specified group number
- -G - The file exists and owner by the user's group
- -h - The file exists and is a symbolic link
- -k - The file exists and has its sticky bit set
- -L - The same as -h
- -O - The file exists you are the owner
- -p - The file exists and is a named pipe
- -r - The file exists and is readable
- -s - The file exists and has a size greater than zero
- -S - The file exists and is a socket
- -t - The file descriptor is opened on a terminal
- -u - The file exists and the set-user-id bit is set
- -w - The file exists and is writable
- -x - The file exists and is executable

Examples:
test /path/to/file1 -n /path/to/file2 && echo "yes"
(If file1 is newer than file2 then the word "yes" will be displayed)



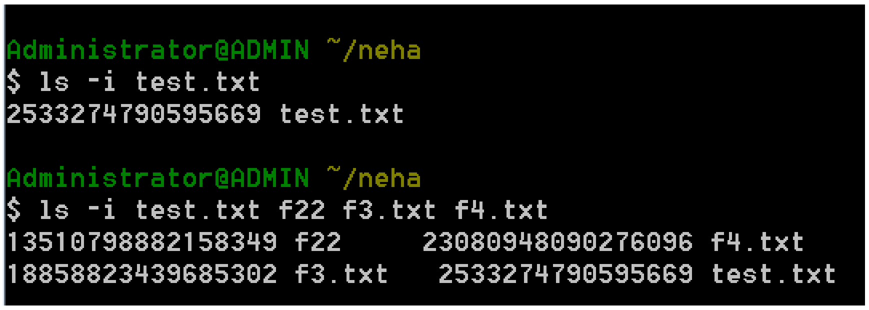
```
Administrator@ADMIN ~/neha
$ test -e /home/Administrator/neha && echo "yes"
yes

Administrator@ADMIN ~/neha
$ test -e /home/Administrator/neha/t && echo "yes"

Administrator@ADMIN ~/neha
$ test -e /home/Administrator/neha/test && echo "yes"

Administrator@ADMIN ~/neha
$ test -e /home/Administrator/neha/test.txt && echo "yes"
yes
```

test -e /path/to/file1 && echo "yes"
(if file1 exists the word "yes" will be displayed)

test -O /path/to/file1 && echo "yes"
(if you own file1 then the word "yes" is displayed")

Managing file links : ln

- Link means a single file having many aliases.
- Like pointers in any programming languages, links in UNIX are pointers pointing to a file or a directory.
- Creating links is a kind of shortcuts to access a file.
- Links allow more than one file name to refer to the same file, elsewhere.
- When we create a file or dir, a system allocates a unique number to it known as inode number; hence system defines any files by their inode-number.
- Directory contains a list of i-node numbers of files and other directory.
- A user can view inode number of any file or dir by ls-i as
  $ ls –i f1<enter>
  1733089 f1
  $



```
Administrator@ADMIN ~/neha
$ ls -i test.txt
2533274790595669 test.txt

Administrator@ADMIN ~/neha
$ ls -i test.txt f22 f3.txt f4.txt
13510798882158349 f22     23080948090276096 f4.txt
18858823439685302 f3.txt   2533274790595669 test.txt
```

- When we make a copy of a file, we are having 2 different files with different names and same content. In this file occupies separate space on a disk.

- But when we make a link of a file, we are having a single file, different names and same contents. In this no separate space is occupied on a disk.
- In case of copy a new file new i-node number is created. But in case of link the files have the same i-node number.

Links have following advantages:

- If a file has 2 links and if one link is removed accidently then your file is save i.e. it just remove the link not the content.
- One file is shared among several users instead of giving each user a separate copy of the same file.
- The ln command is used to create multiple links of a file. The general form is:

Syntax: ln [option] filename link filename

- Unix system offers 2 types of link

1. Hard link
2. Soft link

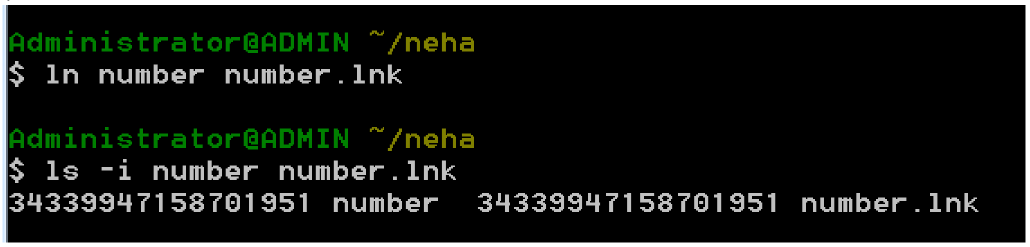
1. HARD-LINK:

- Create a hard link to a file f1 by giving the command as follows:

```
$ ln f1  f1.lnk<enter>
$
```

- Verify links as:

```
$ ls –i f1  f1.lnk
1733089  f1  1733089  f1.lnk
$
```



```
Administrator@ADMIN ~/neha
$ ln number number.lnk

Administrator@ADMIN ~/neha
$ ls -i number number.lnk
3433994715870195l number  3433994715870195l number.lnk
```

- If we do changes in f1 then it will also be in f1.lnk
- f1 and f1.lnk are same files.
- They are known as hard link because they create a direct link to i-node.
- If we delete any one from f1 and f1.lnk using rm, we are actually deleting only one link.
- A file is strictly deleted from the file system only when it has no links.
- When we create a file it will have just one link.
- If we have multiple links to a file, the command rm will delete only a single link.
- In directories we consider 2 hard links:”..” (A link pointing to the parent directory) and “.” (A link pointing to itself)
- The parent of root dir is the root dir itself.

2. SYMBOLIC LINK(SOFT LINK):

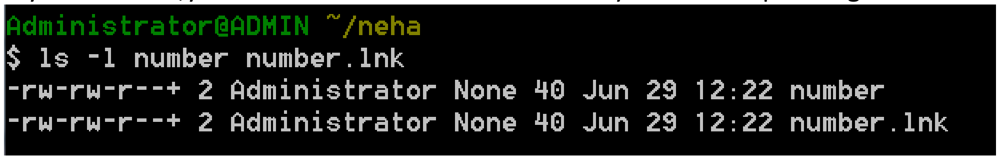
- Symbolic link allows you to give a file, another name, but does not link the file by i-node.
- We can create Symbolic link to the file using ln with –s option as follows:

  $ ln -s a1 a1.lnk

- It display nothing means you created a symbolic link named a1.lnk that points to the file a1.
- If ls –i is used the 2 files have different i-nodes.

  $ ls -i a1 a1.lnk
  309686 a1   309589 a1.lnk

- If you use ls –l ,you will see that the file a1.lnk is a symbolic link pointing to a1



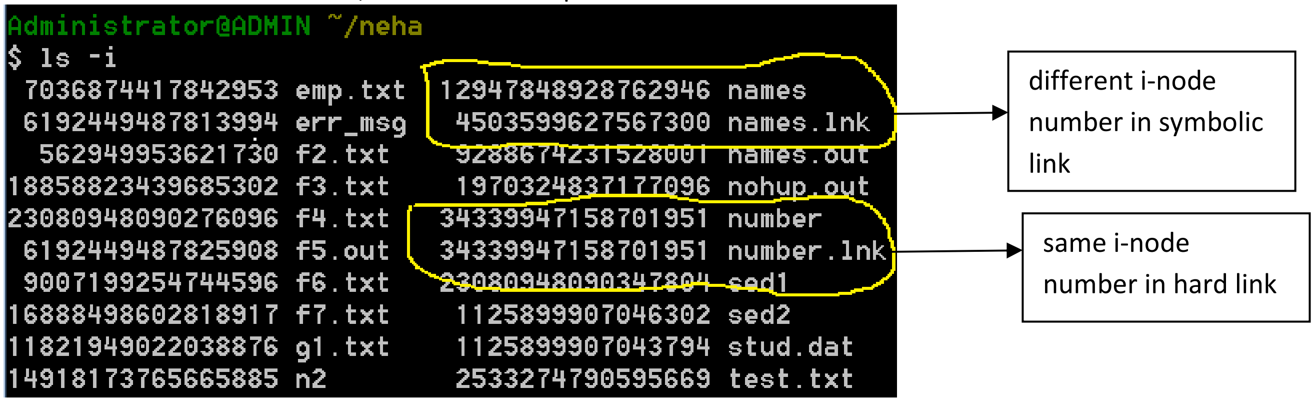
```
Administrator@ADMIN ~/neha
$ ls -l number number.lnk
-rw-rw-r--+ 2 Administrator None 40 Jun 29 12:22 number
-rw-rw-r--+ 2 Administrator None 40 Jun 29 12:22 number.lnk
```

The result shows that a symbolic link file a1.lnk will not use the permissions of input file a1.

- They always have the permission rwxrwxrwx.
- On the other hand permission of hard link is same as that of original file.
- Drawback of symbolic link is that if we remove source file, we lose the file containing the data.
- In this case, the link file points to a nonexistent file.
- So we call it dangling symbolic link.

DIFFERENCE BETWEEN HARD-LINK AND SOFT-LINK

- Symbolic links identify the file they point to. But with hard link, there is no easy way to determine which files are linked to the same i-node.
- You can create hard-link files only when they are on the same file system; symbolic links not have this restriction.
- We can create symbolic links between 2 directories, but not hard link.
- Symbolic link file have different i-node number, file size and file permission whereas hard link file have same i-node number, file size and file permission.



```
Administrator@ADMIN ~/neha
$ ls -i
 7036874417842953 emp.txt   12947848928762946 names
 6192449487813994 err_msg    4503599627567300 names.lnk
  562949953621730 f2.txt     9288674231528001 names.out
18858823439685302 f3.txt     1970324837177096 nohup.out
23080948090276096 f4.txt    34339947158701951 number
 6192449487825908 f5.out    34339947158701951 number.lnk
 9007199254744596 f6.txt    23080948090347804 sed1
16888498602818917 f7.txt     1125899907046302 sed2
11821949022038876 g1.txt     1125899907043794 stud.dat
14918173765665885 n2         2533274790595669 test.txt
```

different i-node number in symbolic link
same i-node number in hard link

- A user can create a symbolic link to a file that does not exist. The same is not possible with hard-links.

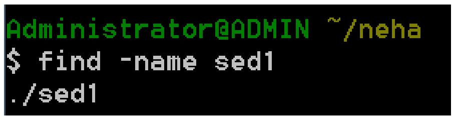
Find command

it searches files in the directory hierarchy.
Syntax:

Find [path-list] selection-criteria [action]

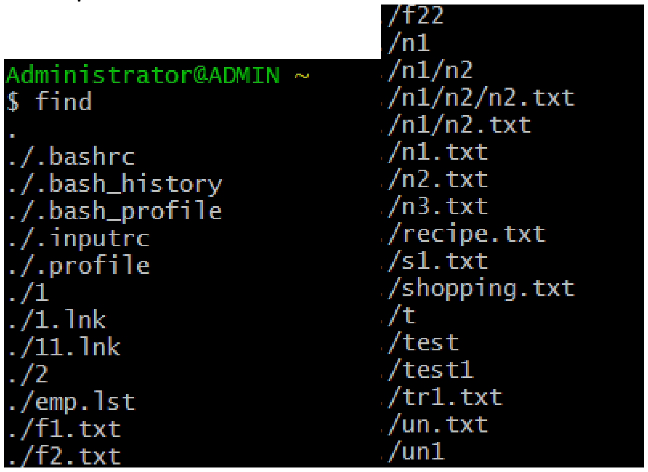
Path_list: it is one or more sub-directories separated by white space.
On omitting this, current directory will be taken as default path-list.

Selection-criteria: it is one or more operators as shown below.
selection criteria begins with hyphen.
we can join one or more operator using logical **operator –a and –o.**



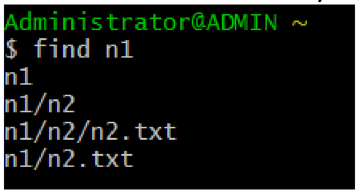
```
Administrator@ADMIN ~/neha
$ find -name sed1
./sed1
```

1. A simple find command:



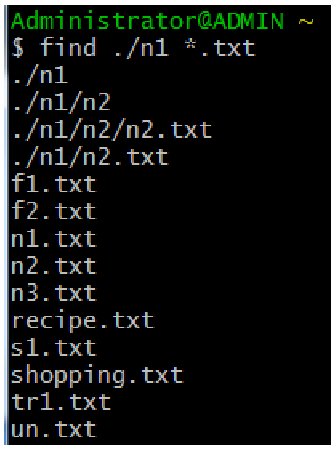
```
Administrator@ADMIN ~
$ find
.
./.bashrc
./.bash_history
./.bash_profile
./.inputrc
./.profile
./1
./1.lnk
./11.lnk
./2
./emp.lst
./f1.txt
./f2.txt
/f22
/n1
/n1/n2
/n1/n2/n2.txt
/n1/n2.txt
/n1.txt
/n2.txt
/n3.txt
/recipe.txt
/s1.txt
/shopping.txt
/t
/test
/test1
/tr1.txt
/un.txt
/un1
```

2. find with file or directory name:



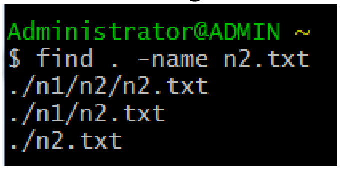
```
Administrator@ADMIN ~
$ find n1
n1
n1/n2
n1/n2/n2.txt
n1/n2.txt
```

3. **Search for a file by the name abc in the current directory and its hierarchy**
   $ find -name abc
4. Search a file with specific name.



```
Administrator@ADMIN ~
$ find ./n1 *.txt
./n1
./n1/n2
./n1/n2/n2.txt
./n1/n2.txt
f1.txt
f2.txt
n1.txt
n2.txt
n3.txt
recipe.txt
s1.txt
shopping.txt
tr1.txt
un.txt
```

5. Find Files Using Name in Current Directory



```
Administrator@ADMIN ~
$ find . -name n2.txt
./n1/n2/n2.txt
./n1/n2.txt
./n2.txt
```

6. Find Files Using Name and Ignoring Case

Find all the files whose name is tecmint.txt and contains both capital and small letters in /home directory.

```
# find /home -iname tecmint.txt
./tecmint.txt
./Tecmint.txt
```

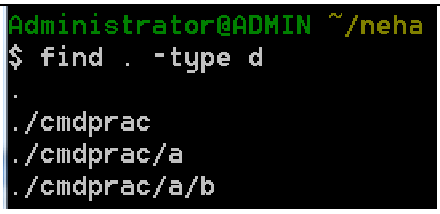
7. **Find Directories Using Name**

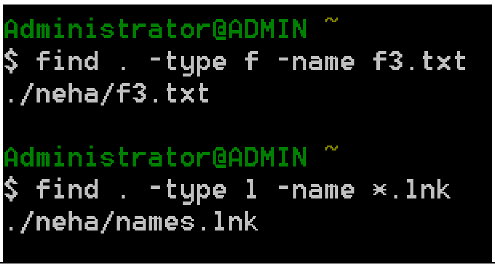
Find all directories whose name is Tecmint in / directory.

```
# find / -type d -name Tecmint
/Tecmint
```

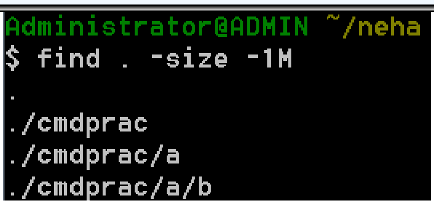
Operators used with find

Operator	Significance
-name flname	It selects file flname
-user uname	It selects files owned by uname
-group gname	It selects files owned by group user gname
-type ftype	It selects files of type ftype. where ftype should be any of the following: f(regular file),d(directory file),l(link file),c(character file),b(block file),p(FIFO file),s(socket file).
-type f	It selects ordinary/regular files
-type d	It selects directory files

	
	```
	Administrator@ADMIN ~/neha
	$ find . -type d
	.
	./cmdprac
	./cmdprac/a
	./cmdprac/a/b
	```
	 Finding hidden directories `find -type d -name ".*"`

-type l	It selects linked files 
	```
	Administrator@ADMIN ~
	$ find . -type f -name f3.txt
	./neha/f3.txt

	Administrator@ADMIN ~
	$ find . -type l -name *.lnk
	./neha/names.lnk
	```

-links n|-n|+n	It selects file having n links(-n for less than n links and +n for greater than n links)
inum n|-n|+n	It selects file having i-node number n (-n for less than n number and +n for greater than n number)
-size x|-x|+x	It selects file if size equal to x blocks(-x for less than x blocks and +x for greater than x blocks) How to find files based on the size? **a.** Finding files whose size is exactly 10M `find . -size 10M` **b.** Finding files larger than 10M size `find . -size +10M` **c.** Finding files smaller than 10M size `find . -size -10M`

	
	```
	Administrator@ADMIN ~/neha
	$ find . -size -1M
	.
	./cmdprac
	./cmdprac/a
	./cmdprac/a/b
	```

-atime x|-x|+x	It selects file if accessed in x days (-x for less than x days and +x for greater than x days) Print the files which are accessed within 1 day. `find . -atime -1`
-amin x|-x|+x	It selects file if accessed in x minutes (-x for less than x minutes and +x for greater than x minutes) Print the files which are accessed within 1 hour.

	`find . -amin -60`

-mtime x|-x|+x	It selects file if modified in x days
	(-x for less than x days and +x for greater than x days)
	
	Find the files which are modified within 1 day.
	`find . -mtime -1`
	
	How to find the files which are modified 1 day back.
	`find . -not -mtime -1`

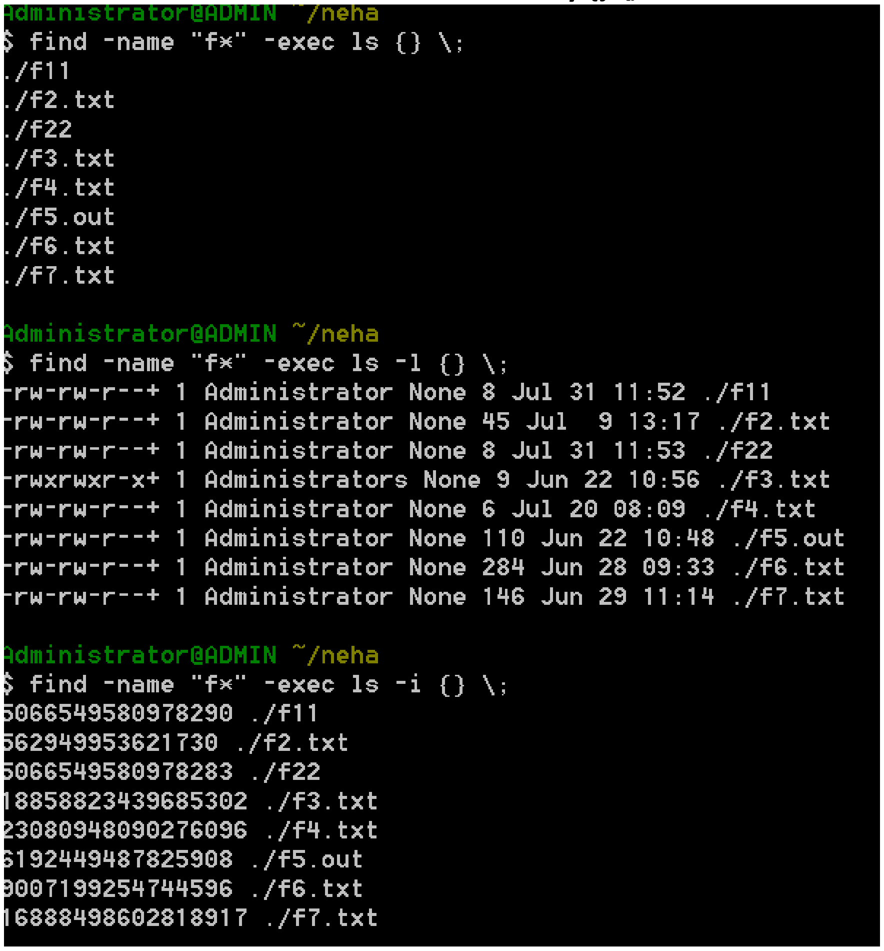
-mmin x|-x|+x	It selects file if modified in x minutes(-x for less than x minutes and +x for greater than x minutes)
	
	Find the files which are modified within 30 minutes.
	`find . -mmin -30`
	
	How to find the files which are modified 30 minutes back
	`find . -not -mmin -30`

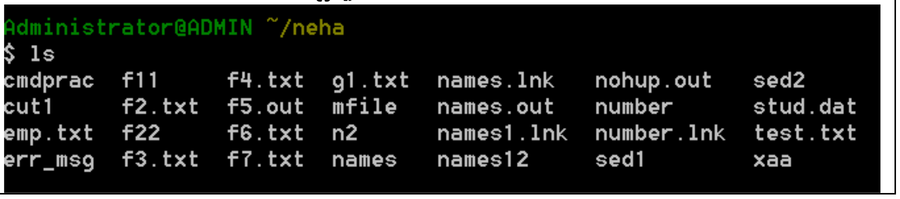
-newer flname	It selects file if modified after flname
	
	How to find the files which are modified after the modification of a give file.
	`find -newer "sum.java"`
	This will display all the files which are modified after the file "sum.java"
	
	Display the files which are accessed after the modification of a give file.
	`find -anewer "sum.java"`
	
	Display the files which are changed after the modification of a give file.
	`find -cnewer "sum.java"`

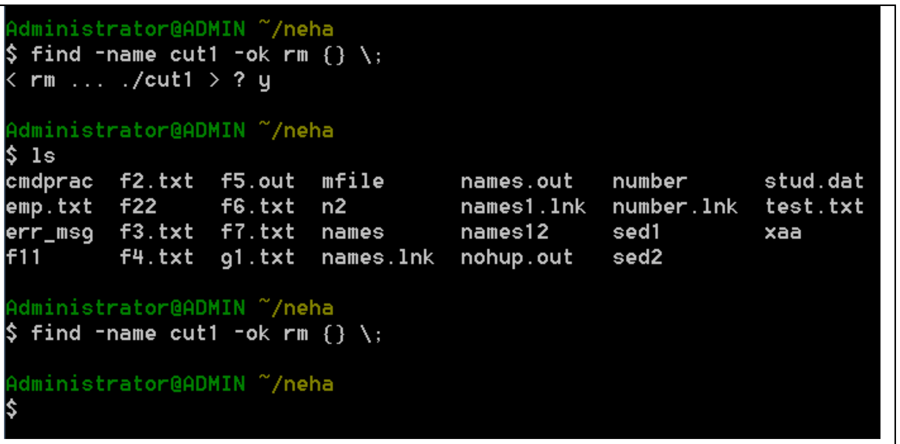
-perm nnn	It selects file if octal permission is nnn How to find the files based on the file permissions? `find . -perm 777` This will display the files which have read, write, and execute permissions. To know the permissions of files and directories use the command "ls -l".
-maxdepth n	It selects files upto n levels
Find Read Only Files	Find all Read Only files. **`# find / -perm /u=r`**

Actions: actions performed with find command.

Action	Significance

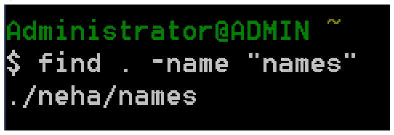
-exec cmd	**it executes unix command cmd followed by {} \;**
	
	`Administrator@ADMIN ~/neha`
	`$ find -name "f*" -exec ls {} \;`
	`./f11`
	`./f2.txt`
	`./f22`
	`./f3.txt`
	`./f4.txt`
	`./f5.out`
	`./f6.txt`
	`./f7.txt`
	
	`Administrator@ADMIN ~/neha`
	`$ find -name "f*" -exec ls -l {} \;`
	`-rw-rw-r--+ 1 Administrator None 8 Jul 31 11:52 ./f11`
	`-rw-rw-r--+ 1 Administrator None 45 Jul  9 13:17 ./f2.txt`
	`-rw-rw-r--+ 1 Administrator None 8 Jul 31 11:53 ./f22`
	`-rwxrwxr-x+ 1 Administrators None 9 Jun 22 10:56 ./f3.txt`
	`-rw-rw-r--+ 1 Administrator None 6 Jul 20 08:09 ./f4.txt`
	`-rw-rw-r--+ 1 Administrator None 110 Jun 22 10:48 ./f5.out`
	`-rw-rw-r--+ 1 Administrator None 284 Jun 28 09:33 ./f6.txt`
	`-rw-rw-r--+ 1 Administrator None 146 Jun 29 11:14 ./f7.txt`
	
	`Administrator@ADMIN ~/neha`
	`$ find -name "f*" -exec ls -i {} \;`
	`5066549580978290 ./f11`
	`5629499953621730 ./f2.txt`
	`5066549580978283 ./f22`
	`18858823439685302 ./f3.txt`
	`23080948090276096 ./f4.txt`
	`6192449487825908 ./f5.out`
	`9007199254744596 ./f6.txt`
	`16888498602818917 ./f7.txt`
	
	Another examples are
	find . –name f1 –exec rm{}\;
	#; is necessary
	find . –name f1 -type f –exec rm{}\;

-ok cmd	**similar to –exec, except that command is executed after user's confirmation**
	
	find . –name f1 –ok rm {}\;
	
	`Administrator@ADMIN ~/neha`
	`$ ls`
	`cmdprac  f11     f4.txt  g1.txt  names.lnk    nohup.out   sed2`
	`cut1     f2.txt  f5.out  mfile   names.out    number      stud.dat`
	`emp.txt  f22     f6.txt  n2      names1.lnk   number.lnk  test.txt`
	`err_msg  f3.txt  f7.txt  names   names12      sed1        xaa`

	 ```Administrator@ADMIN ~/neha``` ```$ find -name cut1 -ok rm {} \;``` ```< rm ... ./cut1 > ? y``` ```Administrator@ADMIN ~/neha``` ```$ ls``` ```cmdprac f2.txt f5.out mfile names.out number stud.dat``` ```emp.txt f22 f6.txt n2 names1.lnk number.lnk test.txt``` ```err_msg f3.txt f7.txt names names12 sed1 xaa``` ```f11 f4.txt g1.txt names.lnk nohup.out sed2``` ```Administrator@ADMIN ~/neha``` ```$ find -name cut1 -ok rm {} \;``` ```Administrator@ADMIN ~/neha``` ```$```
-print	**it prints selected files on standard output(it is the default action)**

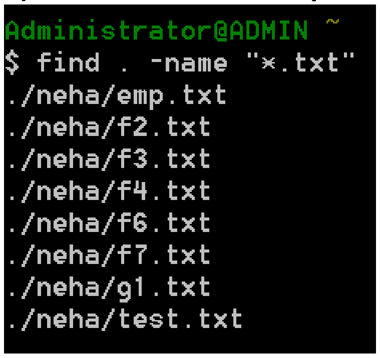
Examples: you want to locate all files names file1 under current directory hierarchy, then command is :

$find . –name f1 –print
./d1/d2/f1
./d1/f1
./f1
$



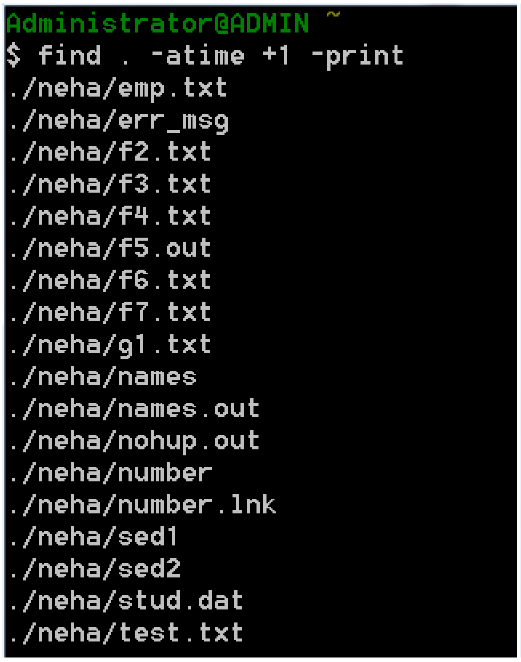
```
Administrator@ADMIN ~
$ find . -name "names"
./neha/names
```

2)find . –name "*.sh" –print

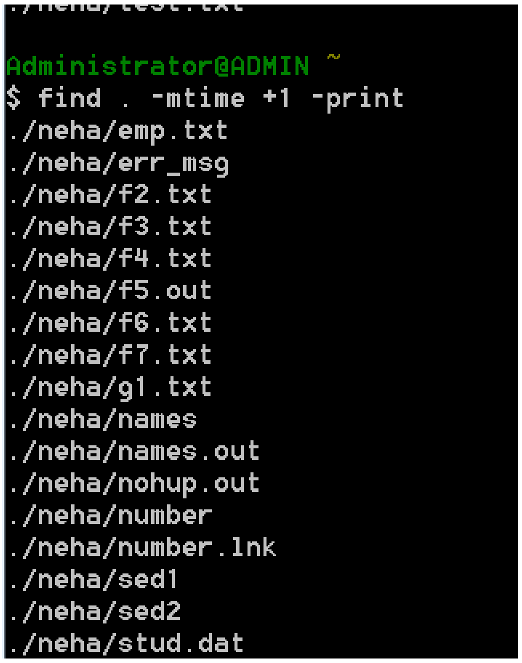


```
Administrator@ADMIN ~
$ find . -name "*.txt"
./neha/emp.txt
./neha/f2.txt
./neha/f3.txt
./neha/f4.txt
./neha/f6.txt
./neha/f7.txt
./neha/g1.txt
./neha/test.txt
```

3)$find . –mtime 5 –print



```
Administrator@ADMIN ~
$ find . -atime +1 -print
./neha/emp.txt
./neha/err_msg
./neha/f2.txt
./neha/f3.txt
./neha/f4.txt
./neha/f5.out
./neha/f6.txt
./neha/f7.txt
./neha/g1.txt
./neha/names
./neha/names.out
./neha/nohup.out
./neha/number
./neha/number.lnk
./neha/sed1
./neha/sed2
./neha/stud.dat
./neha/test.txt
```



```
./neha/test.txt

Administrator@ADMIN ~
$ find . -mtime +1 -print
./neha/emp.txt
./neha/err_msg
./neha/f2.txt
./neha/f3.txt
./neha/f4.txt
./neha/f5.out
./neha/f6.txt
./neha/f7.txt
./neha/g1.txt
./neha/names
./neha/names.out
./neha/nohup.out
./neha/number
./neha/number.lnk
./neha/sed1
./neha/sed2
./neha/stud.dat
```

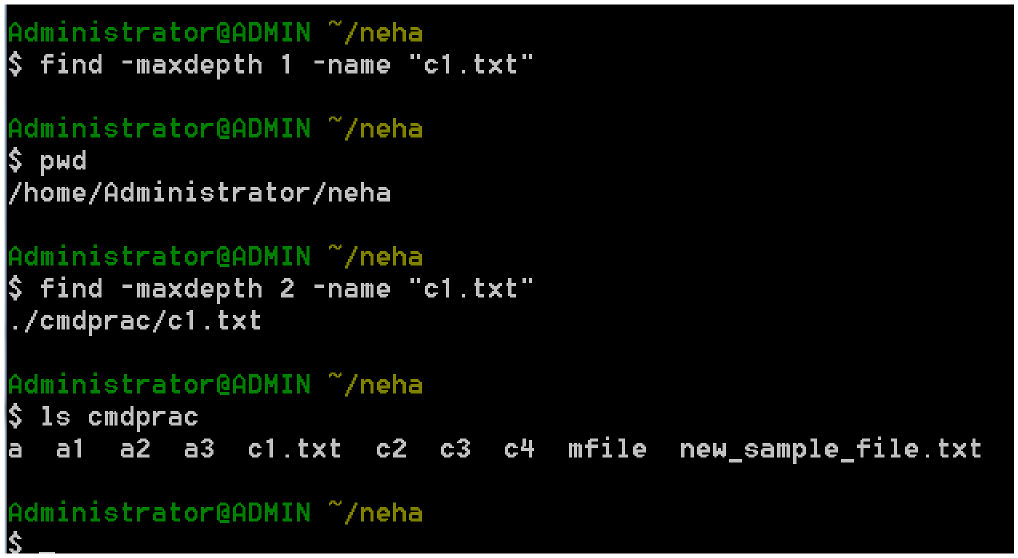
4) $find /user/b1 –name “f?” –print

8)$find . –maxdepth 1 –name f1 –print

# search file named f1 in current directory only

9)find . –maxdepth 2 –name f1 –print

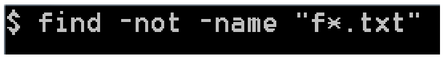
# search file named f1 in current directory & its sub-directory only



```
Administrator@ADMIN ~/neha
$ find -maxdepth 1 -name "c1.txt"

Administrator@ADMIN ~/neha
$ pwd
/home/Administrator/neha

Administrator@ADMIN ~/neha
$ find -maxdepth 2 -name "c1.txt"
./cmdprac/c1.txt

Administrator@ADMIN ~/neha
$ ls cmdprac
a  a1  a2  a3  c1.txt  c2  c3  c4  mfile  new_sample_file.txt

Administrator@ADMIN ~/neha
$ _
```

How to find the files whose name are not "sum.java"?

find -not -name "sum.java"



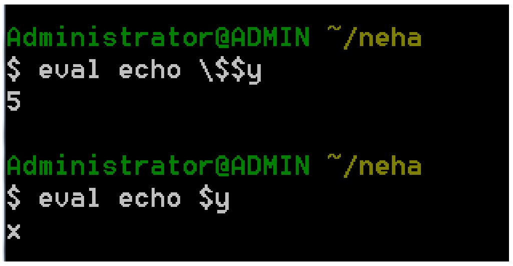
```
$ find -not -name "f*.txt"
```

Evaluate command : eval

This command evaluates the command twice.

In the first pass, it ignores character followed by backslash(\) and make a substitution if any. It executes command in 2nd pass only.

Let us consider an example:
x=5
y=x
eval echo \$$y



```
Administrator@ADMIN ~/neha
$ eval echo \$$y
5

Administrator@ADMIN ~/neha
$ eval echo $y
x
```

In this example during the 1st pass $ symbol after \ remains as it is and $y replaces by 'x'. where as in 2nd pass $x replaced by 5 and then **echo** command print on screen.

Eval command is useful when a user wish to create a command line inside a variables. It is useful to assign a value of a variable to another variable as follow:

```
$a="hello"
$apte=a
$eval $aptr = "world"
$echo $a
```

Executing a command stored in a variable

```
$ c="ls|more"
$ eval $c
1
11
1cd
201403152214.45
22
a
……..output of more ahead
```

3) Extracting the last positional parameter

```
$ set unix
$ eval echo \$$#
unix
```

4) using pipe stored in a variable and this variable is used for executing commands

```
$ x="|"
```

```
$ls $x wc
ls: |: No such file or directory
ls: wc: No such file or directory
$ eval ls $x wc
    79      79     395
```

5)
```
$ ls f*
f1  f1234  f1copy  f1h  f1s  f2  ft.sh
$ wc<f*
-bash: f*: ambiguous redirect
```
" \<" this holds redirection which is to be executed first and executes it later with eval ,first f* is executed which is to be executed to be second.
```
$ eval wc \< f*
     0      0      0 f1234
     1      1      5 f1copy
     1      1      5 f1h
     4      8     31 f1s
     4      8     31 f2
     8     40    154 ft.sh
    18     58    226 total
```

6)
```
$x=5
$ y=x
$ y=4
$ echo $x
5
```
assigns value of one variable to other variable.
```
$ x=5
$ y=x
$ eval $y=4
$ echo $x
4
```

7)
```
$ x=5
$ y=x
$ eval echo \$$y
5
```

User mask : umask

umask, as the man page says, stands for User file creation mask which is used for determining the default permission for a new file creation.
umask command is a shell built-in meaning it is an internal command.

There are three general classes of users:

- The user who owns the file ("User").
- Users belonging to the file's defined ownership group ("Group").

- Everyone else ("Other").

The three file permission attributes are read, write and execute.
These 3 are mapped to octal values as shown below:

read	- 4
write	- 2
execute	- 1

The umask command is used to set this mask, or to show you its current value.
The system default settings for both file and directory is rw-rw-rw (i.e. Octal 666) and rwxrwxrwx (i.e. Octal 777) respectively.

In UNIX, the default file creation value is 666. 6 is 4+2(read + write).
Permission 666 means 6 for the User, 6 for the group and 6 for others.
Hence, a new file creation by default is meant to have read and write permission for User, group and others.
This is the place where the umask comes into the picture.
It is a kind of filter wherein we can choose to retain or block some of the default permissions from being applied on the file.

Say, the umask value is 0022. umask is by default displayed in Octal form, and hence the first 0 in the umask value is the indication for octal value. So, the actual umask is 022. This value together with the default file value(666) decides the final permission to be given to the file.

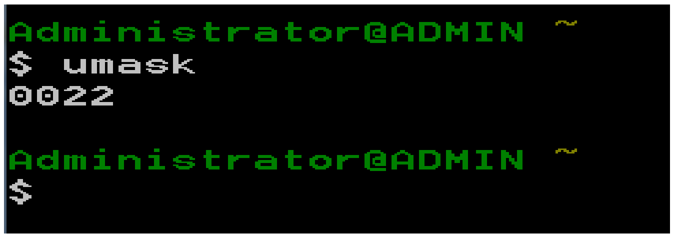
Assume we create a file say "file1". The permissions given for this file will be the result coming from the substraction of the umask from the default value :

```
  Default: 666
  umask : 022
---------------
Result :   644
```

644 is the permission to be given on the file "file1". 644 means read and write for the User(644), read only for the group(644) and others(644).

- **What is umask?**
  umask is a number which defines the default permissions which are not to be given on a file. A umask of 022 means not to give the write permission to the group(022) and others(022) by default.

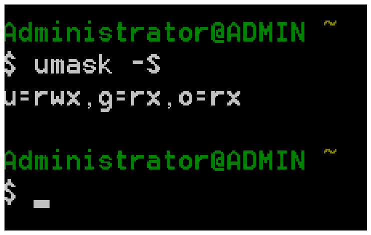
2. How to find out the umask value?



```
Administrator@ADMIN ~
$ umask
0022

Administrator@ADMIN ~
$
```

The option -S gives in more readable format.

This means umask, at the max, allows all permissions for the user, read and execute alone for the group and others.



```
Administrator@ADMIN ~
$ umask -S
u=rwx,g=rx,o=rx

Administrator@ADMIN ~
$ _
```

Example:-

If user creates a regular file then the default permission is calculated as below:

Default Permission of regular file

= system's default setting for regular file – user mask

= 666 – 002

= 664

So, Default Permission of regular file is assigned as 664(666 - 002) i.e. rw-rw-r--

[6 = read + right, 4= read, 2= write, 7= read + write+ execute, 1= execute]

If user creates a new directory then the default permission is calculated as below:

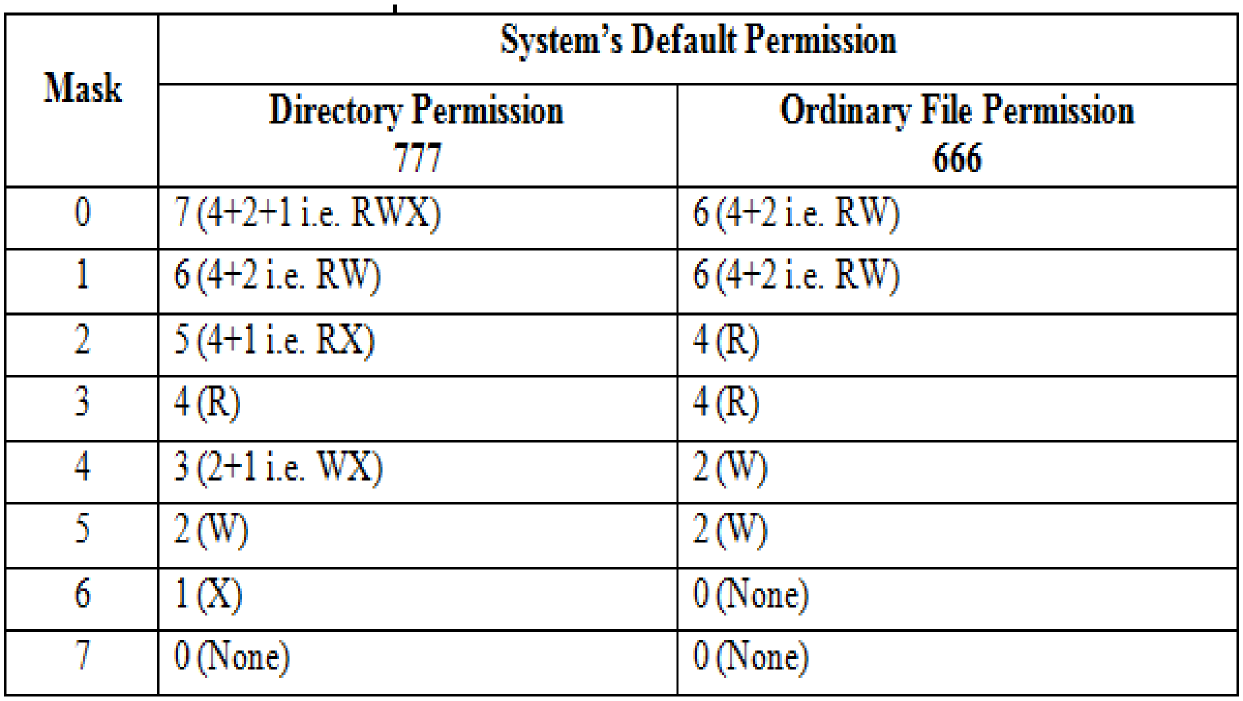
Default Permission of directory

= System's default setting for directory – user mask

= 777- 002

= 775

- So, default permission to new directory is assigned as 755 (777 – 002) i.e. rwxrwxr-x
- You can change user mask settings as follows:
  $umask 222



Mask	System's Default Permission	
	Directory Permission 777	Ordinary File Permission 666
0	7 (4+2+1 i.e. RWX)	6 (4+2 i.e. RW)
1	6 (4+2 i.e. RW)	6 (4+2 i.e. RW)
2	5 (4+1 i.e. RX)	4 (R)
3	4 (R)	4 (R)
4	3 (2+1 i.e. WX)	2 (W)
5	2 (W)	2 (W)
6	1 (X)	0 (None)
7	0 (None)	0 (None)

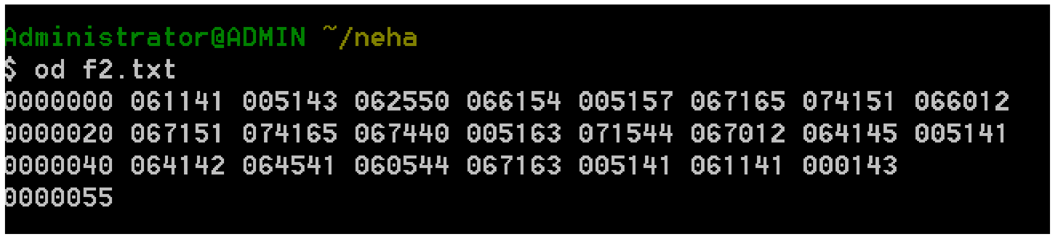
- ❖ If mask value 1 means remove 1 from system's default directory and file permission.
- ❖ For directory, the default permission is 7 (i.e. 4+2+1), so we remove 1 then the result is 6 (4+2).
- ❖ The default for a file is 6 (4+2), so we remove 1 then result is 6 (4+2).

Octal dump : od

Most of the unix commands dont display invisible characters such as tab, space, new line etc.
od is a program for displaying ("dumping") data in various human-readable output formats.
The name is an acronym for "octal dump" since it defaults to printing in the octal data format.
It can also display output in a variety of other formats, including hexadecimal, decimal, and ASCII.
It is useful for visualizing data that is not in a human-readable format, like the executable code of a program.

od syntax

```
od [OPTION]... [FILE]...
```

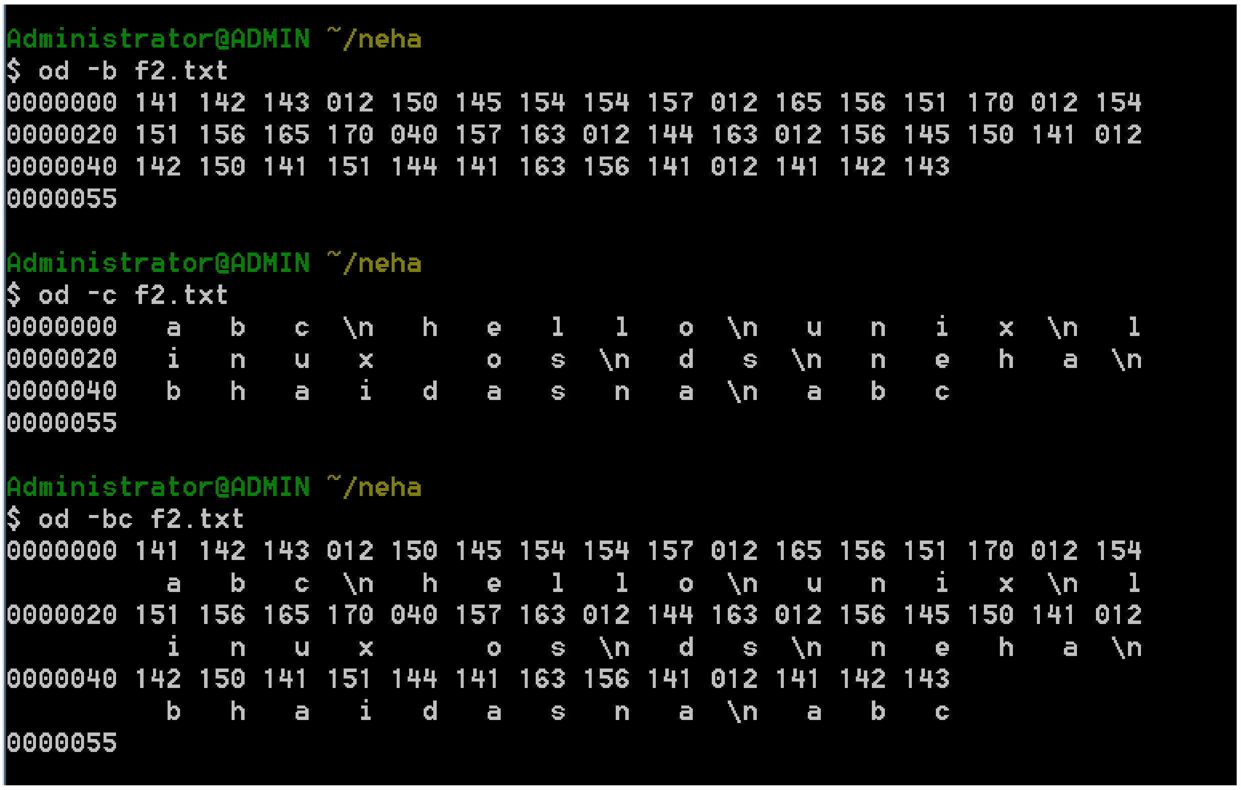


```
Administrator@ADMIN ~/neha
$ od f2.txt
0000000 061141 005143 062550 066154 005157 067165 074151 066012
0000020 067151 074165 067440 005163 071544 067012 064145 005141
0000040 064142 064541 060544 067163 005141 061141 000143
0000055
```

So we see that output was produced in octal format. The first column in the output of od represents the byte offset in file.

byte offset is the number of character that exists counting from the beginning of a line. ... The **byte offset** is the count of **bytes** starting at zero.

Display contents of file in character format using -c option
Display contents of file in character in octal format using -b option



```
Administrator@ADMIN ~/neha
$ od -b f2.txt
0000000 141 142 143 012 150 145 154 154 157 012 165 156 151 170 012 154
0000020 151 156 165 170 040 157 163 012 144 163 012 156 145 150 141 012
0000040 142 150 141 151 144 141 163 156 141 012 141 142 143
0000055

Administrator@ADMIN ~/neha
$ od -c f2.txt
0000000   a   b   c  \n   h   e   l   l   o  \n   u   n   i   x  \n   l
0000020   i   n   u   x       o   s  \n   d   s  \n   n   e   h   a  \n
0000040   b   h   a   i   d   a   s   n   a  \n   a   b   c
0000055

Administrator@ADMIN ~/neha
$ od -bc f2.txt
0000000 141 142 143 012 150 145 154 154 157 012 165 156 151 170 012 154
          a   b   c  \n   h   e   l   l   o  \n   u   n   i   x  \n   l
0000020 151 156 165 170 040 157 163 012 144 163 012 156 145 150 141 012
          i   n   u   x       o   s  \n   d   s  \n   n   e   h   a  \n
0000040 142 150 141 151 144 141 163 156 141 012 141 142 143
          b   h   a   i   d   a   s   n   a  \n   a   b   c
0000055
```

chown and chgrp

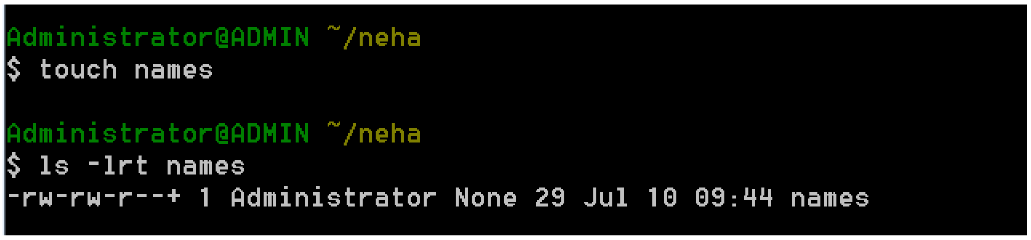
It is possible to change ownership and group name of file or directory.
The chown command changes ownership of files and directories in a unix file system.

Syntax : chown **[option]** ***new-user File(s)***

Change modification & access time of file : Touch command

➔This command is used to create the empty files in Unix system. The size of the file created using touch command in UNIX will be 0 bytes. We will be able to add the contents into the file using vi command.

➔Ex: touch TestFile.txt


```
Administrator@ADMIN ~/neha
$ touch names

Administrator@ADMIN ~/neha
$ ls -lrt names
-rw-rw-r--+ 1 Administrator None 29 Jul 10 09:44 names
```

There are three time stamps associated with UNIX file:

- Last modification date and time.---**mtime**
- Last access date and time.---**atime**
- Last inode change date and time. ---**ctime**

  [A timestamp is information associated with a file that identifies an important time in the file's history.
  A file can have multiple timestamps, and some of them can be "forged" by setting them manually.
  Internally, the operating system stores these times as time elapsed since an arbitrary date called the epoch.]

➔ When user make changes in a file then modification time (mtime) is changed by the kernel.
➔ When we read, write and execute a file then access time (atime) of file is changed by the kernel.
➔ But ctime changes a few extra times. For example, it will change if you change the owner or the permissions on the file.

A user can display last modification time by: ls –l as:

```
$ ls -l y11
---x--x--x   1 neha   neha       33 Jul  5 03:07 y11
```

A user can display last access time & date by: ls –lu as:

```
 $ ls -lu y11
---x--x--x   1 neha   neha   33 Jul  9 04:32 y11
```

A user can display last inode change time & date by: ls –lc as:

```
$ ls -lc y11
---x--x--x   1 neha   neha   33 Jul 11 03:18 y11
```

- Touch command allows you to change modification and access time of file. The general form of touch command is:

Syntax: *Touch[option][expression]file(s)*

- It updates the access and modification times of each file to the current time.

- When touch is used without option and expression, it changes both times of file to current time. If file does not exist then it creates an empty file.
- Using touch command, you can update modification and access time of file f1 to current time as follow:

```
$touch f1
```

  The result shows that both the time of file f1 have been changed to the current time.
- In the syntax of the touch command, an expression consists of the form **[[cc]yy]MMDDhhmm[.ss]**.
- The meaning of each symbol is given in table:

Symbol	Meaning
cc	It denotes first two digits of 4-digits year.
yy	It denotes last two digits of 4-digits year.
MM	It denotes month of the year(1-12).
DD	It denotes day of the month(1-31).
hh	It denotes hour(0-23).
mm	It denotes minutes(0-59).
ss	It denotes seconds(0-59).

Consider a file f2 as follow:

```
$ls-l f2 ; ls-lu f2
-rwxrwxrwx 1 bharat bharat 23 feb 28 13:37 f2
-rwxrwxrwx 1 bharat bharat 23 mar 11 16:31 f2
$
```

- You can change both modification and access time of file f2 using **–t option** as follow:

```
$touch –t 201403152214.45 f2
$ls -l f2; ls -lu f2
-rwxrwxrwx 1 bharat bharat 23 mar 15 22:14 f2
-rwxrwxrwx 1 bharat bharat 23 mar 15 22:14 f2
$
```

- The **–a option** is used to change only access time of file.

```
$ touch -a 03171414 f2
touch: warning: `touch 03171414' is obsolete(outdated); use `touch -t 201603171414.00'
$ ls -lu f2
 -rw-rw-r--    1 nidhi    nidhi         0 Mar 17 14:14 f2
```

- Similarly, **-m option** is used to change only modification time of a file. Use **–t option with –m** option you can suppress warning message.

```
$ touch -m -t 03171414 f2
$ ls -lu f2
-rw-rw-r--    1 nidhi    nidhi          0 Mar 17 14:14 f2
```

- Consider a file f2 , you can change modification time of file f2 to current time as :
  $ touch –m f2

- If a user doesn't want to create an empty file if file does not exist then **–c option** is used.

```
$ touch –c  nofile1
$ ls nofile1
Ls : nofile1: no such file or directory
$
```

Result shows that there is no nofile1 in a directory. It means that touch command does not create an empty file even though a file does not exist.

Shell Meta characters

The meta characters are special characters which are interpreted by the shell.
The shell expand their meaning at a time of execution.

They are characterized as follow:

- **Filename substitution**
- **Redirection**
- **Piping**
- **Conditional Execution**
- **Process execution**
- **Quoting & escaping**
- **Positional Parameters**
- **Variable & command execution**

- **Filename substitution : There are mainly 3 meta characters for file name substitution.**

Item	Description
*	Matches any string, including the null string
?	Matches any one character
[. . .]	Matches any one of the characters enclosed in square brackets

1. **Asterisk (*) :** The asterisk is a wildcard that matches for zero or more of any character in a filename.
   Example
   1 $ ls *   abc abc1 abc122 abc123 abc2 file1 file1.bak file2 file2.bak
   none   nonsense noone nothing nowhere one
   2 $ ls *.bak   file1.bak file2.bak 3 $ print a*c   abc

   EXPLANATION
   1. The asterisk expands to all of the files in the present working directory. All of the files are passed to ls and displayed.

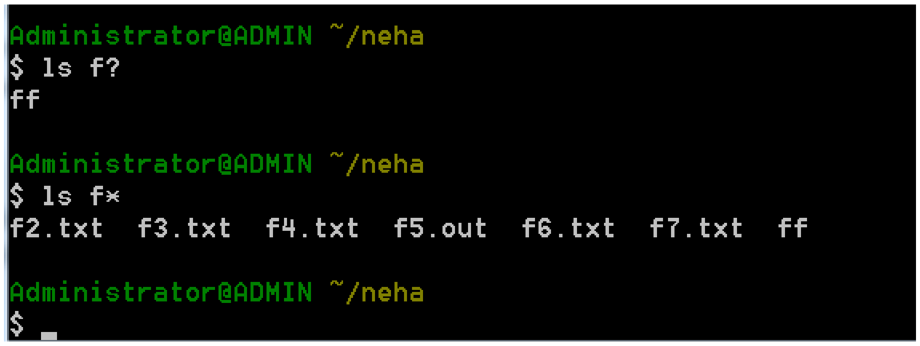
like **$ls***

2. All files starting with zero or more characters and ending with .bak are matched and listed.
3. All files starting with a , followed by zero or more characters, and ending in c are matched and passed as arguments to the print command.

2. **Question-mark(?):**The question mark represents a single character in a filename. When a filename contains one or more question marks, the shell performs filename substitution by replacing the question mark with the character it matches in the filename.

   Example

```
 1  $  ls   abc abc1 abc122 abc123 abc2 file1 file1.bak file2 file2.bak
none nonsense noone nothing nowhere one
2  $  ls a?c?   abc1 abc2
3  $  ls ??   ?? not found
4  $  print abc???   abc122 abc123  5  $  print ??   ??
```



```
Administrator@ADMIN ~/neha
$ ls f?
ff

Administrator@ADMIN ~/neha
$ ls f*
f2.txt  f3.txt  f4.txt  f5.out  f6.txt  f7.txt  ff

Administrator@ADMIN ~/neha
$ 
```

EXPLANATION

1. The files in the current directory are listed.
2. Filenames containing four characters are matched and listed if the filename starts with an a , followed by a single character, followed by a c and a single character.
3. Filenames containing exactly two characters are listed. There are none, so the two question marks are treated as literal characters. Because there is no file in the directory called ?? , the shell sends the message ?? not found .
4. Filenames containing six characters are matched and printed, starting with abc and followed by exactly three of any character.
5. The ksh print function gets the two question marks as an argument. The shell tries to match for any filenames with exactly two characters. There are no files in the directory that contain exactly two characters. The shell treats the question mark as a literal question mark if it cannot find a match. The two literal question marks are passed as arguments to the print command.

3. **Character class [..] :** Brackets are used to match filenames containing one character from a set or range of characters.

Example

```
1  $  ls   abc abc1 abc122 abc123 abc2 file1 file1.bak file2 file2.bak   none nonsense none
           nothing nowhere one
2  $  ls abc[123]   abc1 abc2
3  $  ls abc[13]   abc1 abc2
4  $  ls [az][az][az]   abc one
5  $  ls [!fz]???   abc1 abc2
6  $  ls abc12[2-3]   abc122 abc123
```

EXPLANATION

1. All of the files in the present working directory are listed.
2. All four-character names are matched and listed if the filename starts with abc , followed by 1 , 2 , or 3 . Only one character from the set in the brackets is matched for a filename.
3. All four-character filenames are matched and listed if the filename starts with abc , and is followed by a number in the range from 1 to 3 .
4. All three-character filenames are matched and listed, if the filename contains exactly three lowercase alphabetic characters.
5. All four-character files are listed if the first character is not a letter between f and z , followed by three of any character (???).
6. Files are listed if the filenames contain abc12 , followed by 2 or 3 .

Character class uses 2 another meta characters ! (bang or exclamation)and -(hyphen) to specify the range inside the classes e.g.: [1-4].
! used to reverse the matching criteria.

3.7 Redirection

- Redirection is a feature in unix such that when executing a command, you can change the standard input/output devices.
- Most Unix system commands take input from your terminal and send the resulting output back to your terminal.
- A command normally reads its input from the standard input, which happens to be your terminal by default.
- Similarly, a command normally writes its output to standard output, which is again your terminal by default.
- Every program you run from the shell opens three files:
    - **Standard input,**
    - **standard output, and**
    - **standard error.**
- The files provide the primary means of communications between the programs, and exist for as long as the process runs.
- With redirection, the above standard input/output can be changed.

Understanding I/O streams numbers: The Unix / Linux standard I/O streams with numbers:

Handle	Name	Description

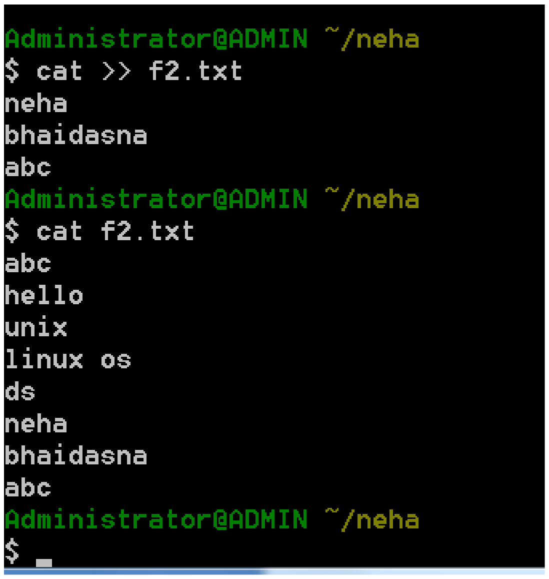
0	stdin	Standard input
1	stdout	Standard output
2	stderr	Standard error

1. **Output Redirection:**
   The output from a command normally intended for standard output can be easily diverted to a file instead.

   This capability is known as output redirection.

   If the **notation > file** is appended to any command that normally writes its output to standard output, the output of that command will be written to file instead of your terminal.

   - e.g.: **$ cat file1 > file2** [A user can create a duplicate file using output redirection]
   - **$ cat file1 > file2** #command using file descriptor for o/p
   - **$wc f1 >> f2** #appends o/p in f2.
   - **$ cat > newfile <enter>** #output redirection creates a new file
     This is new file <enter>
     <ctrl+d>
     $



```
Administrator@ADMIN ~/neha
$ cat >> f2.txt
neha
bhaidasna
abc
Administrator@ADMIN ~/neha
$ cat f2.txt
abc
hello
unix
linux os
ds
neha
bhaidasna
abc
Administrator@ADMIN ~/neha
$ _
```

   - If you want to override the option and replace the current file
     Contents with new output, you must use the **redirection override operator, greater than bar(>|).**
     $ who >| exist_file

2. **Input Redirection:**
   - Just as the output of a command can be redirected to a file, so can the input of a command be redirected from a file.
   - As the greater-than character > is used for output redirection,

- the **less-than character < is** used to redirect the input of a command.
- The commands that normally take their input from the standard input can have their input redirected from a file in this manner.
- We can apply **input from keyboard**.
- For example, to **count the number of lines** in the file users generated above, you can execute the command as follows −

```
$ wc -l users
2 users
$
```

- Upon execution, you will receive the following output.
- You can count the number of lines in the file by redirecting the standard input of the wc command from the file users −

```
$ wc -l < users
2
$
```

```
$ wc            #takes std input until ctrl+d then give o/p
hiii
hello
(ctrl+d)
 2    2    11
```

note: here newline "\n" is also calculated.

- **$ wc < f1.txt** # take i/p from redirection.
- **$cat f1 – f2** # 3 inputs f1 ,std i/p and f2
- **Cat 0< file1** # reads file and display its content on screen
- **Cat < file1** # same as *cat 0<file1*
- **Cat file1** # not use input redirection

3. Error Redirection:

- Standard error ("stderr") is like standard output and standard input, but it's the place where error messages go.
- Error re-direction is one of the very popular features of Unix/Linux.
- Frequent UNIX users will reckon that many commands give you massive amounts of errors.
- For instance, while searching for files, one typically gets permission denied errors. These errors usually do not help the person searching for a particular file.
- While executing shell scripts, you often do NOT want error messages cluttering up the normal program output.

Example 1

```
$ myprogram 2 > errorsfile
```

Above we are executing a program names myprogram.
The file descriptor for standard error is 2.
Using "2>" we re-direct the error output to a file named "errorfile"
Thus, program output is not cluttered with errors.

example:

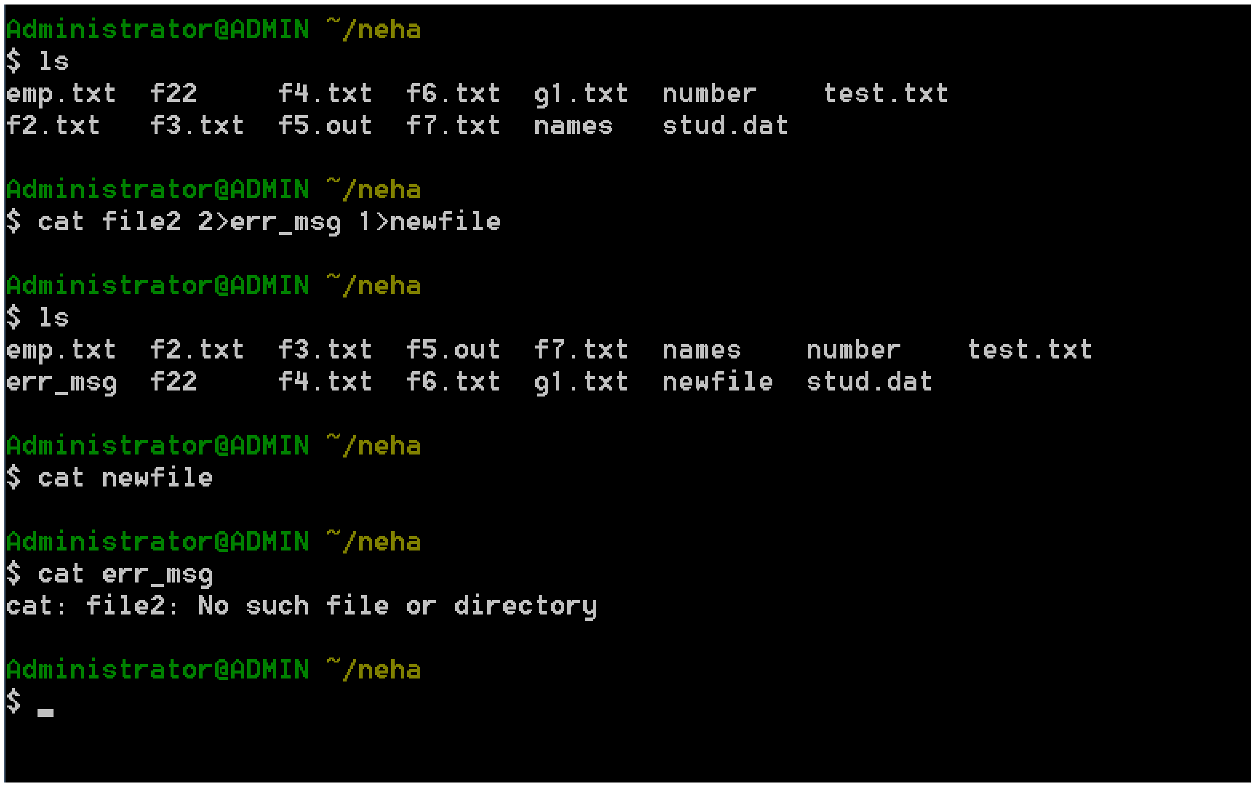
To write an output of command or error message into a same file, we must use redirection substitution operator(>&)

$ ls –l file1 file2 file3   1> out_file   2>&1

Here, Error file is same as output file

$ cat file1 file2 file3   2>err_file 1>&2

Here output file is same as error file.



```
Administrator@ADMIN ~/neha
$ ls
emp.txt  f22     f4.txt  f6.txt  g1.txt  number    test.txt
f2.txt   f3.txt  f5.out  f7.txt  names   stud.dat

Administrator@ADMIN ~/neha
$ cat file2 2>err_msg 1>newfile

Administrator@ADMIN ~/neha
$ ls
emp.txt  f2.txt  f3.txt  f5.out  f7.txt  names    number    test.txt
err_msg  f22     f4.txt  f6.txt  g1.txt  newfile  stud.dat

Administrator@ADMIN ~/neha
$ cat newfile

Administrator@ADMIN ~/neha
$ cat err_msg
cat: file2: No such file or directory

Administrator@ADMIN ~/neha
$ _
```

Redirection Commands

Following is a complete list of commands which you can use for redirection –

Sr.No.	Command & Description
1	**pgm > file** **Output** of pgm is redirected to file
2	**pgm < file** Program pgm **reads its input** from file
3	**pgm >> file** Output of pgm is **appended** to file
4	**n > file** Output from stream with descriptor **n redirected** to file
5	**n >> file** Output from stream with descriptor **n appended** to file
6	**n >& m**

	Merges output from stream **n** with stream **m**
7	**n <& m** Merges input from stream **n** with stream **m**
8	**<< tag** Standard input comes from here through next tag at the start of line
9	**|** Takes output from one program, or process, and sends it to another

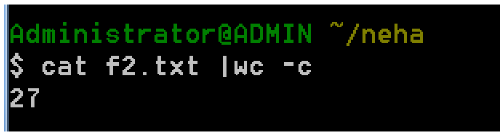
Piping Mechanism:

- **Piping** connects the output of one process to the input of a second.
- Piping uses the symbol "|".
- Pipe is used to combine two or more command and in this the output of one command act as input to another command and this command output may act as input to next command and so on.
- It can also be visualized as a temporary connection between two or more commands/ programs/ processes.
- The command line programs that do the further processing are referred to as filters.
- **A filter** is a process which is between two pipes.
- It simply changes the information coming down the pipe.
- Standard input for a process can be drawn from the script file itself.
- This uses the special redirection symbol "<<".
- The name which follows the redirection symbol is a tag for the end of the input.
- The Unix/Linux systems allow stdout of a command to be connected to stdin of another command.
- **Syntax :**

```
command_1 | command_2 | command_3 | .... | command_N
```

Lets us consider a command as follow:

```
$cat f1|wc -c
123
$
```



```
Administrator@ADMIN ~/neha
$ cat f2.txt |wc -c
27
```

here the o/p of cat command can be used as standard i/p of wc command.
So, the result shows the no. of characters in a file f1.

Advantages of pipe feature:

- There is no need to create intermediate temporary files to perform complex task.
- As compare to redirection it is faster.

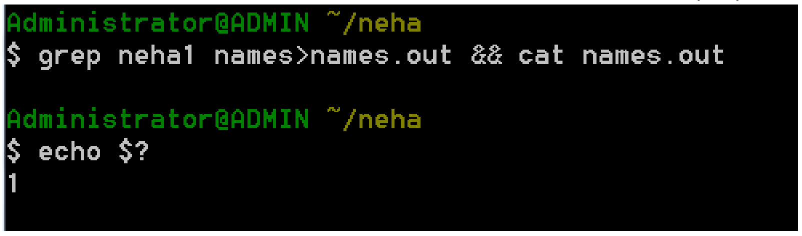
Conditional Execution:

- There are 2 conditional execution meta characters : && and ||.
- These meta characters can execute a command depending on the screen or failure of the previous command.
- The shell provides meta-character && which executes the **second command** only if the first command succeeds.

Syntax: COMMAND 1 && COMMAND 2 OR
COMMAND 1 || COMMAND 2
e.g.:

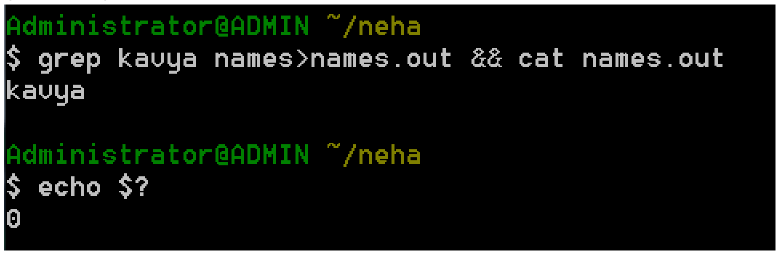
```
$ cat y11
hhhhhhhhhuh+
ds
hello
linux
unix
```

$ grep neha1 y11 > y11.out && cat y11.out
Here we not get 'neha1 pattern so, prompt will come
$ echo $? # $? Shows exit status of last command,1 ,so unsuccessful



```
Administrator@ADMIN ~/neha
$ grep neha1 names>names.out && cat names.out

Administrator@ADMIN ~/neha
$ echo $?
1
```

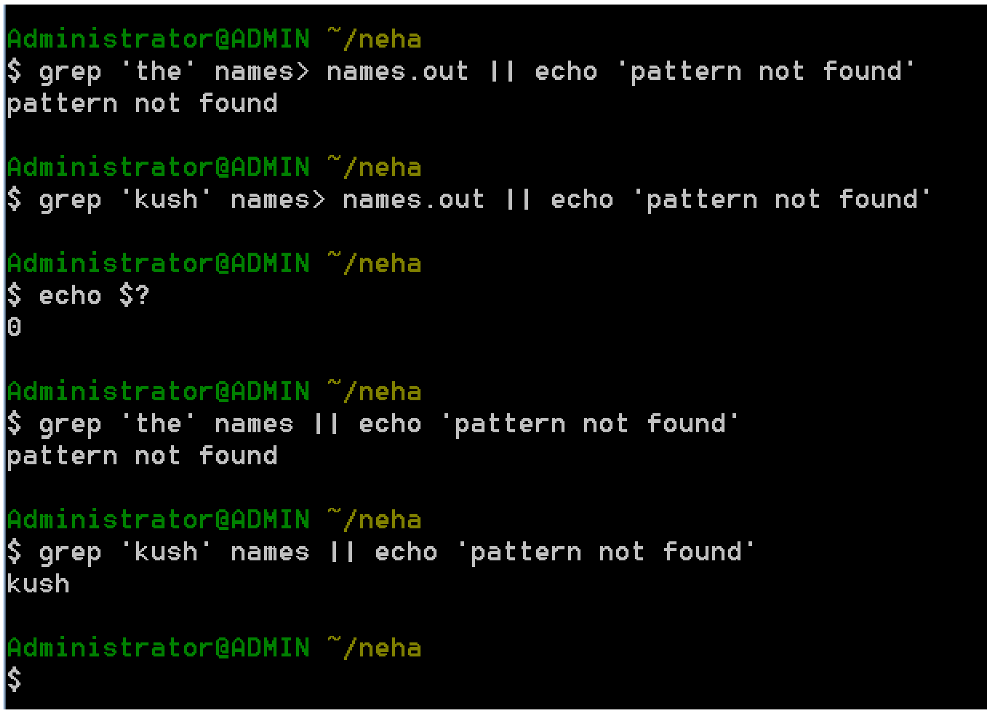
$ grep 'hello' y11 > y11.out && cat y11.out
Hello #pattern found so command successful
$ echo $? #exit status 0 shows success of last command



```
Administrator@ADMIN ~/neha
$ grep kavya names>names.out && cat names.out
kavya

Administrator@ADMIN ~/neha
$ echo $?
0
```

➔ **Shell provides meta character || which executes second command only if the first command fails.**
Syntax : Command 1 || command 2

- **$ grep 'the' y11 > y11.out || echo 'pattern not found'**
  pattern not found
- **$ grep 'hello' y11 > y11.out || echo 'pattern not found'**
  $ #output
- **$ grep 'the' y11 || echo 'pattern not found'**
  pattern not found



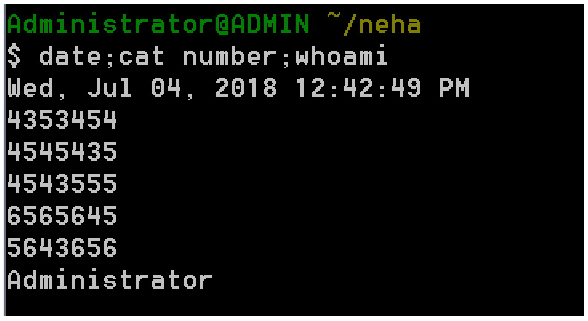
```
Administrator@ADMIN ~/neha
$ grep 'the' names> names.out || echo 'pattern not found'
pattern not found

Administrator@ADMIN ~/neha
$ grep 'kush' names> names.out || echo 'pattern not found'

Administrator@ADMIN ~/neha
$ echo $?
0

Administrator@ADMIN ~/neha
$ grep 'the' names || echo 'pattern not found'
pattern not found

Administrator@ADMIN ~/neha
$ grep 'kush' names || echo 'pattern not found'
kush

Administrator@ADMIN ~/neha
$
```

COMMAND GROUPING:

This feature allows shell to execute commands in different way on command line.
There are 4 different forms of this: Sequence command, Group of commands, Chain of commands, Conditional commands.

- **Sequence of commands**:
  sometimes a user wants to execute one or more commands all together at shell prompt in one hit then process execution metacharacter (;) is used.
  cmd 1;cmd 2;……;cmd N

- **$ date;cat y11;who am i**

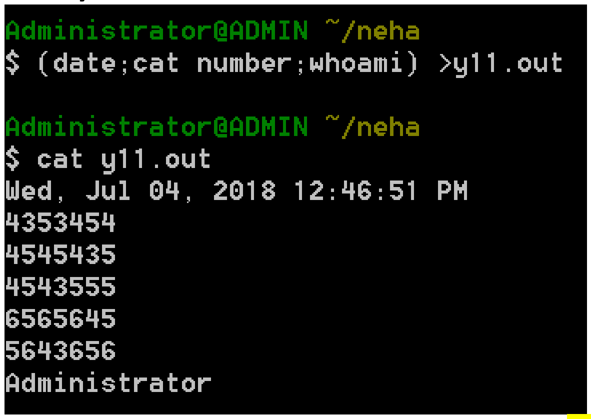


```
Administrator@ADMIN ~/neha
$ date;cat number;whoami
Wed, Jul 04, 2018 12:42:49 PM
4353454
4545435
4543555
6565645
5643656
Administrator
```

- **Group of commands : (cmd 1;cmd 2;……;cmd N)**
  To send output of more than one command into a single file then we have to make a group of these commands.
  A meta-character parenthesis is used to make a group of commands.
  Syntax:
  (cmd 1;cmd 2;……;cmd N)

Example:

- **$ (date;cat y11;who am i) > y11.out**
- **$ cat y11.out**



```
Administrator@ADMIN ~/neha
$ (date;cat number;whoami) >y11.out

Administrator@ADMIN ~/neha
$ cat y11.out
Wed, Jul 04, 2018 12:46:51 PM
4353454
4545435
4543555
6565645
5643656
Administrator
```

when shell gets parenthesis then it creates sub-shell.
So all commands in parenthesis will be executed in a sub-shell.
Sub-shell will be terminated as soon as closing parenthesis is encountered.

- **Chain of commands:** joining of commands by |(pipe)
- **Conditional commands:** joining of commands by && and || meta characters

H-W
read /dev/null file
read /dev/tty file
from bharat 7.10.1 and 7.10.2-PAGE NO: 118

/dev/null file
sometime, user want to ignore un-necessary output and error messages generated by any command or programs, and do not want to save this output/error messages, then /dev/null file is used.
When we redirect any output to this file, its size always remains zero.
For example:
$ cat f123
cat: f123: No such file or directory #error message
$ cat f123 > f1234 2> /dev/null
$ cat /dev/null
$ #prompt will come means nothing stored in /dev/null

- **Quoting & escaping**

Problem with rm a* or ls a*

To protect special characters (including wild cards) so the shell is not able to interpret

- **ESCAPING**-provides a **\(backslash)** before the wild card to remove(or turn off) its special meaning
- **QUOTING**-enclosing the wild card, or even entire pattern, within quotes. ex: 'chap*' (i.e. pairs of quotes ".." and '..')

➢ $ cat > a* Hiii

- $ cat > a1
hello
- $ ls a*
a* a1
- $ cat a*
hiii
hello
- $ cat a*
hiii

ESCAPING:

- rm chap* #doesn't remove chap1,chap2
- ls chap0\[1-3\] # chap0[1-3]
- rm chap0\[1-3\]
- $ echo \\ #gives \ in output, in absence of escaping takes std. I/p and display it in O/p.

```
$ echo \
> aaa
aaa
```

- Pass the **-e parameter** to enable interpretation of backslash escapes.
- Example:

```
$ echo -e "hello\nworld"
hello
world
```

```
$ echo -e "hello\tworld"
hello   world
```

- $ echo "hello\nworld"

```
hello\nworld
```

- $ echo -e hello\nworld

```
hellonworld
```

```
$ echo -e hello\\nworld
hello
world
```

QUOTING:

- Escaping turns to be tedious when there are too many characters to protect. At that time quoting will be used.
- Single quote protects all special characters
- Double quote also protect all except the $ and `(backquote)
- $ echo $SHELL
/bin/bash
- $ echo "$SHELL"
/bin/bash
- $ echo '$SHELL'
$SHELL

- echo '\' #displays ans \
- echo "\" #creates file
- rm "my document.doc" # removes file my document.doc

- $echo 'the path searched is $PATH'

the path searched is $PATH

- $ echo "the path searched is $PATH"
  Ans : the path searched is /usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/nidhi/bin

- **double quote protect single quote and vice versa.**
  Ex: echo " 'hiiii' "   or   echo ' "hello" '
- **Single quote protect all special characters**
- Single quote protects system variables not double quote as:
  $ echo '$SHELL'
  $SHELL
- Single quote also protects ``(backquote)from command substitution.

- <u>**Variable & command execution**</u>

What is Substitution? - **The shell performs substitution when it encounters an expression that contains one or more special characters.**

<u>**Variable Substitution :**</u>

Variable substitution enables the shell programmer to manipulate the value of a variable based on its state.

Here is the following table for all the possible substitutions −

Sr.No.	Form & Description
1	**${var}** Substitute the value of *var*.
2	**${var:-word}** If *var* is null or unset, *word* is substituted for **var**. The value of *var*does not change.
3	**${var:=word}** If *var* is null or unset, *var* is set to the value of **word**.
4	**${var:?message}** If *var* is null or unset, *message* is printed to standard error. This checks that variables are set correctly.
5	**${var:+word}** If *var* is set, *word* is substituted for var. The value of *var* does not change.

<u>**COMMAND SUBSTITUTION:**</u>

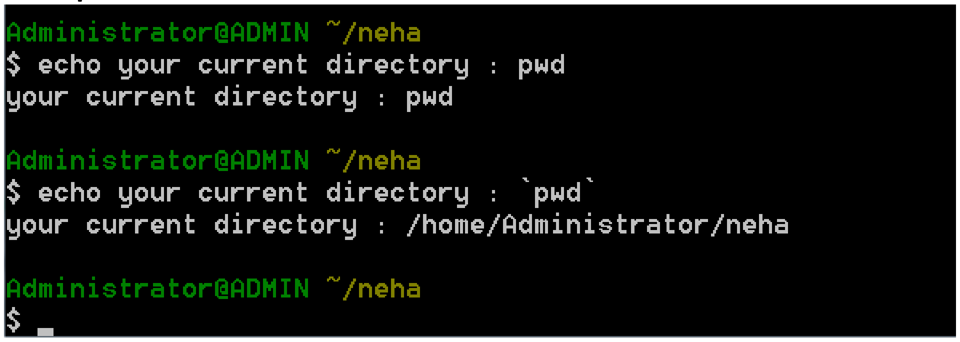
Command substitution is the mechanism by which the shell performs a given set of commands and then substitutes their output in the place of the commands.

Syntax

```
`command`
```

When performing the command substitution make sure that you use the backquote, not the single quote character.

Example1:



```
Administrator@ADMIN ~/neha
$ echo your current directory : pwd
your current directory : pwd

Administrator@ADMIN ~/neha
$ echo your current directory : `pwd`
your current directory : /home/Administrator/neha

Administrator@ADMIN ~/neha
$ _
```

Example2:

Command substitution is generally used to assign the output of a command to a variable.
Each of the following examples demonstrates the command substitution –

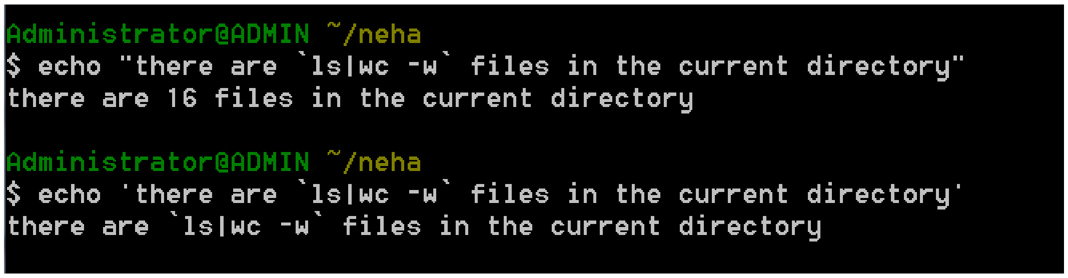
```
#!/bin/sh
DATE=`date`
echo "Date is $DATE"

USERS=`who | wc -l`
echo "Logged in user are $USERS"

UP=`date ; time`
echo "Uptime is $UP"
```

Upon execution, you will receive the following result –

```
Date is Thu Jul  2 03:59:57 MST 2009

Logged in user are 1
Uptime is Thu Jul  2 03:59:57 MST 2009
03:59:57 up 20 days, 14:03,  1 user,  load avg: 0.13, 0.07, 0.15
```

3. **$ echo "there are `ls|wc -w` files in the current directory"**
Output: there are 60 files in the current directory

$ echo 'there are `ls|wc -w` files in the current directory'
#single quote protects special meaning of `(backquote)
Output: there are `ls|wc -w` files in the current directory



```
Administrator@ADMIN ~/neha
$ echo "there are `ls|wc -w` files in the current directory"
there are 16 files in the current directory

Administrator@ADMIN ~/neha
$ echo 'there are `ls|wc -w` files in the current directory'
there are `ls|wc -w` files in the current directory
```