```
File Protection
---------------

Users        Owner, Group, Others/World

Permissions rEAD, wRITE, eXECUTE

How to see ownership?
     ls -l
     ls -ld

1 File type:
          - = file, d = directory, l = symbolic link, p = named pipe,
c = character device, b = block device

2-4        Read, write, and execute permissions for the file's owner
5-7        permission (rwx) for  file's group members
8-10       permission (rwx) for other users

History of commands
-------------------

     history           Displays the history of commands issued

     history -c  Deletes/Clears the history

     history N   Displays last N number of commands from your command history
record

     !$          Last parameter from the previous command
     !*          All parameters from the previous command

                 $ ls a b c
                 a b c
                 $ rm !$
                 $ ls c
                 c not found
                 $ ls x y z
                 x y z
                 $rm !*
                 $ ls x y z
                 x y z not found

Editor vi(m)
------------

     Modes       Insert and command mode
                 i to insert
                 esc (Escape key) to get back to command mode

     Moving around    Arrow keys in command mode

     Beginning of file 1G
     End of file      G
     Any line x       xG
     Beginning of Line 0
     End of Line      $
     w                word
     n                next
     j                Down arrow
     k                Up arrow
```

```
       l                  Left arrow
       h                  Right arrow
       /string                   Search for given string


       Substitute         :s/old/new/

       Therefore, if you need to change a word, command would be cw

       D          Delete upto end of line from where you are
       R          Replace until you wish from where you are
       x          Delete current character
       X          Delete previous character


       .          apply the last change once more


       u          undo last change


       dd         Delete current line


       ndd        Delete n number of lines


       yy         Copy current line in the unnamed buffer


       nyy        Copy n number of lines from current line into unnamed buffer
       p          Put the content of the buffer from the current position
onwards


       P          Put the content of the buffer just before the current position


       :w         Save changes to the current file


       :q         Quit the edit session, in case, changes are done and are not
to be saved use :q!



       Named registers   a b c so on upto z, 0-9, A-Z



       Show line numbers :set number
                    (In short, :se nu)


       Indent                :set autoindent
                    :se ai


       All setting       :set all
                    Displays current settings


       Search at the
       beginning of line /^string


       Search at the
       end of line       /string$


       Blank line search /^$


       Deleting blank line    :map v /^$ dd
       First map v
       and now issue v command
========================================================
```

Session 3

31 What is shell?
32 Different types of Linux shells
33 Bourne Again Shell (BASH)
34 Shell variables (environment and user defined)
35 Shell files (.bashrc, .profile, .bash_profile, .bash_logout)
36 Wild cards (* and ?)
37 I/O redirection and tee command
38 Shell meta characters
39 Command line expansion

Session 4

40 Command line arguments
41 Arithmetic in shell scripts
42 Read and echo commands in shell scripts
43 Taking decisions:
     o if-then-fi
     o if-then-else-fi
     o The test command (file tests, string tests)
     o Nested if-else
     o The case control structure
===

Session 3
31 What is shell?

     Command line interface to the operating system services
     Shell controlled environment
     Program execution
     Variable and file name substitution
     I/O Redirection
     Pipeline hookup
     Interpreted programming language
     Environment control
     Has built-in commands
     (cd, export, etc.)
     Provides mechanisms to control jobs/processes created by the user

32 Different types of Linux shells

     Bourne Shell   /bin/sh
     C Shell   /bin/csh
     K Shell   /bin/ksh
     bash (Bourne Again Shell)   /bin/bash

     Other shells

     rsh
     zsh
     tcsh - c like scripting

     Implementation features of C, K and bash differ on different Unix flavors
and hence it is advisable to write your shell programs (called typically as
shell scripts) in Bourne Shell only.

     First line in your shell scripts acts as a guideline on which shell to be
used for running your script. Should be

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

The above line clearly indicates that your script is to be executed using bourne shell.

33 Bourne Again Shell (BASH)

   BASH provides additional features compared to bourne shell.

   command history & editing
   alias
   Brace expansion
   Process substitution
   Wildcard expansion
   I/O redirection
   Pipes

34 Shell variables (environment and user defined)

   printenv

   Explain some of the environment variables

   Variable names starts with alphabet character and can be combination of
   alphanumeric, is case sensitive

   ```
   $ a=25
   $ A=26
   $ echo "a = $a, A = $A"
   a = 25, A = 26
   ```

   Concept of parent process and child process in relation with export

   When you export a variable, its COPY only is available to all the child
processes of this parent process. Child process even if they alter value of such
exported variables, the changes will not be reflected into parent process
variable.

   Demonstrate exporting of a variable and its consequences
   Use /tmp/vartest

35 Shell files (.bashrc, .profile, .bash_profile, .bash_logout)

   .profile   file gets executed when you login using bourne shell
   .bash_profile   when you log in to bash shell
   .bash_logout   when you log out from bash shell
   .bashrc   gets executed when you log in to bash shell and also when
you start another bash session

   Demonstrate .bashrc changes and its effect on the current and new session.

36 Wildcards (* and ?)

   A wildcard is a character that can stand for all members of some class of
characters. Characters like *, ? are treated differently by the shell.

```
*       matches 0 or more occurrences of any character

?       matches exactly one occurrence of any character

[set]       Any single character in the given set ,
        most commonly a sequence of characters,
        like [aeiouAEIOU] for all vowels, or a range with a
        dash, like [A-Z] for all capital letters

[^set]      Any single character not in the given set ,
        such as [^0-9] to mean any non-digit

[!set] Same as [^set]
```

Brace expansion

```
Similar to wildcards, expressions with curly braces also expand
to become multiple arguments to a command. The comma separated
expression:

{X,YY,ZZZ}

expands first to X, then YY, and finally ZZZ within a command
line, like this:

$ echo sand{X,YY,ZZZ}wich
sandXwich sandYYwich sandZZZwich

Important::

Braces work with any strings, unlike wildcards which expand
only if they match existing filenames.
```

Quotes

```
Three types of quotes

Single quote              ' '
Double quotes             " "
Back quotes           ` `

Text included in back-quotes is evaluated as command by shell

Text included in double quotes preserves white spaces and interprets $
```
prefixed to variable names

```
Text included in single quotes preserves whitespaces and does not
```
interpret characters like $, *, ?

```
    Examples
   --------

 $ echo hello                    world
 hello world

 Please note shell has removed extra whitespace in the above example

 $ echo 'hello                    world'
```

```
    hello                       world

    Please note shell has preserved white space.

    $ x=25
    $ echo 'x = $x'
    x = $x

    Please note $ is not interpreted

    $ echo "x = $x"
    x = 25

    Please note $x has been interpreted

    Back-quote

    $ x=`date`
    $ echo $x
    Sat Sep 15 14:13:05 IST 2018
```

Escape sequences

An escape sequence is a sequence of characters that does not represent itself
when used inside a character or string literal, but is translated into another
character or a sequence of characters that may be difficult or impossible to
represent directly.

The built-in echo command is an older form of printf . Bash provides it for
compatibility with the Bourne shell.

echo does not use a format string: It displays all variables as if "%s\n"
formatting was used.

It can sometimes be used as a shortcut when you don't need the full features of
printf .

$ echo "$BASH_VERSION"   ********* Smart quote !! Problem
2.05a.0(1)-release

A line feed is automatically added after the string is displayed. It can be
suppressed with the -n (no new line) switch.

$ echo -n "This is " ; echo "one line."
This is one line

If the -e (escape) switch is used, echo interprets certain escape sequences as
special characters.

\a —A beep ("alert")
\b —A backspace
\c —Suppresses the next character; at the end of the string, suppresses the
trailing line feed
\E —The escape character
\f —A form feed
\n —A line feed (new line)
\r —A carriage return
\t —A horizontal tab
\v —A

vertical tab
\\ —A backslash
\num —The octal ASCII code for a character

Example

```
$ echo "\101"
\101
$ echo -e "\101"
A

$ echo -e  "x \t yz \n x \n yz \n x \r yz \n" > /tmp/test.txt
$ cat /tmp/test.txt
x       yz
 x
 yz
 x yz

$ od -b /tmp/test.txt
0000000 170 040 011 040 171 172 040 012 040 170 040 012 040 171 172 040
0000020 012 040 170 040 015 040 171 172 040 012 012
0000033
```

The -E switch turns off escape sequence interpretation.This is the default
setting.

37 I/O redirection and tee command

        Three files

        Filename   fd(File Descriptor)
        stdin   0
        stdout   1
        stderr   2

        Output redirected to a file
        $ who > /tmp/users

        Input redirected from a file
        $ /tmp/input_format < /tmp/input_test

        Errors are displayed on the screen but it is not stdout
        $ gcc /tmp/boss.c
          try to redirect the errors

        $ gcc /tmp/boss.c 2> /tmp/errors.txt

        Piping

        Output of one program is given to another program as input.

        Use /tmp directory to illustrate piping.

           Introduce tr command

           tee command

```
tee [options] files
```

Like the cat command, the tee command copies standard input to standard
output unaltered.
Simultaneously, however, it also copies that same standard input to one or
more files. tee is most often found in the middle of pipelines, writing some
File Text Manipulation intermediate data to a file while also passing it to the
next command in the pipeline:

Example

```
$ who | tee original_who | sort
barrett pts/1 Sep 22 21:15
byrnes pts/0 Sep 15 13:51
silver :0 Sep 23 20:44
silver pts/2 Sep 22 21:18
```

This command line produces the sorted output of who on
screen, but also writes the original, unsorted output of who to
the file original_who:

```
$ cat original_who
barrett pts/1 Sep 22 21:15
byrnes pts/0 Sep 15 13:51
silver :0 Sep 23 20:44
silver pts/2 Sep 22 21:18
```

and then passes along that same output to the rest of the pipe□
line (sort), producing sorted output on screen.

Useful options
-a Append instead of overwriting files.
-i Ignore interrupt signals.

Advanced example

```
$ wget http://example.com/some.iso
$ sha1sum some.iso

$ wget http://example.com/some.iso && sha1sum some.iso
```

Both the above versions read the some.iso file two times.

Now, same.iso is read only once
```
$ wget http://example.com/some.iso|tee >(sha1sum > dvd.sha1) > some.iso
```

Example (Process substitution)

Lets say, we have set of files as given below.

file1.jpg file1.txt file2.jpg file2.txt ...

```
→ ls *.jpg | cut -d. -f1 > /tmp/jpegs
→ ls *.txt | cut -d. -f1 > /tmp/texts
→ diff /tmp/jpegs /tmp/texts
```

With process substitution, you can perform the same task with

a single command and no temporary files:

    → diff <(ls *.jpg|cut -d. -f1) <(ls *.txt|cut -d. -f1)

    Each <() operator stands in for a filename on the command
    line, as if that "file" contained the output of ls and cut.

      ----

38 Shell meta-characters

    Normally, the shell treats whitespace simply as separating the
    words on the command line.

    If you want a word to contain whitespace (e.g., a filename with a space in
it), surround it with single or double quotes to make the shell treat it as a
unit.

    Single quotes treat their contents literally, while double quotes let
    shell constructs be evaluated, such as variables:

    $ echo 'The variable HOME has value $HOME'
    The variable HOME has value $HOME
    $ echo "The variable HOME has value $HOME"
    The variable HOME has value /home/mmj

    Backquotes cause their contents to be evaluated as a shell command.
    The contents are then replaced by the standard output of the command:

    $ date +%Y

    Print the current year
    2018

    → echo This year is `date +%Y`
    This year is 2018

    A dollar sign and parentheses are equivalent to backquotes:

    → echo This year is $(date +%Y)
    This year is 2018
    but are superior because they can be nested:
    → echo Next year is $(expr $(date +%Y) + 1)
    Next year is 2019

    Escaping

    If a character has special meaning to the shell but you want it used
literally (e.g., * as a literal asterisk rather than a wildcard), precede the
character with the backward slash "\" character. This is called escaping the
special character:

    $ echo a*
    aardvark adamantium apple
    (As a wildcard, matching "a" filenames)

    $ echo a*
    a*
    (As a literal asterisk)

```
    $ echo "I live in $HOME"
    I live in /home/mmj
    (Print a variable value)

    $ echo "I live in \$HOME"
    I live in $HOME
    (A literal dollar sign)
```

39 Command line expansion

```
      Brace expansion
      Tilde expansion
      Shell Parameter & variable expansion
      Command substitution (` ` or (...))
      Arithmetic expansion
      Process substitution
      Word splitting
      File name expansion
```
LAB::
man bash
search for EXPANSION
Read the section

Combining commands

To invoke several commands in sequence on a single command line, separate them
with semicolons:
$ command1 ; command2 ; command3

To run a sequence of commands as before, but stop execution if any of them
fails, separate them with && ("and") symbols:

$ command1 && command2 && command3

To run a sequence of commands, stopping execution as soon as one succeeds,
separate them with || ("or") symbols:

$ command1 || command2 || command3
===

Quotes (Advanced)
======

When the shell sees the first single quote, it ignores any special characters
that follow until it sees the matching closing quote.

```
$ echo one     two three                          four
one two three four
$ echo 'one     two three
four'
one     two three
four
```

Note:: Shell removed extra whitespace characters in the first instance. In the
second case, the moment it saw ', it decided to skip its processing until the
closing '.

```
$ file=/users/steve/bin/progl
$ echo $file
/users/steve/bin/progl
```

```
$ echo '$file'
$file
```

Note: $ not interpreted

```
$ pwd
/tmp
$ echo *
```

Note: It prints List of files

```
$ echo '*'
*
```

Note: No interpretation of *

```
$ echo '< > | ; ( ) { } >> " &'
< > | ; ( ) { } >> " &
```

Even the Enter key will be retained as part of the command argument if it's
enclosed in single quotes:

```
$ echo 'How are you today,
> Mr. Mallya'
How are you today,
Mr. Mallya
$
```

```
$ echo '* means all files in the directory'
* means all files in the directory
```

```
$ text='* means all files in the directory'
$ echo $text
```

What is the output of the above?

How to fix it?

```
echo "$text"
```

Double Quotes
----

single quotes tell the shell to ignore all enclosed characters,

double quotes say to ignore most. Therefore, double quotes are less protective.

In particular, the following three characters are not ignored inside double
quotes:
```
 $    Dollar sign
 `    Back quote
 \    Backslash
```

Note::
 "smart quotes" are generated by word processors like Microsoft Word and curl
"inward" towards the material they surround, making it much more attractive when
printed.
The problem is, they will break your shell scripts, so be alert!

Example illustrating the difference between double quotes and no quotes:

```
$ address="39 East 12th Street
> New York, N. Y. 10003"
$ echo $address
39 East 12th Street New York, N. Y. 10003

$ echo "$address"
39 East 12th Street
New York, N. Y. 10003
$
```

Singlequote and Doublequote mixed::

```
$ x="' Hello,' she said"
$ echo $x
'Hello,' she said
$ article=' "nice song" as per pappu'
$ echo $article
"nice song" as per pappu
```

Backslash

The backslash (used as a prefix) is equivalent to placing single quotes around a single character, though with a few minor exceptions.

The backslash escapes the character that immediately follows it. The general format is \c where c is the character you want to quote.

Any special meaning normally attached to that character is removed.

```
$ echo >
syntax error: 'newline or ;' unexpected
$ echo \>
>
$
```

In the first usage, the shell sees the > and thinks that you want to redirect echo 's output to a file. As it doesn't filename, the shell issues the error message.

In the next usage, the backslash escapes the special meaning of the > , so it is passed along to echo as a character to be displayed.

One more example ==>

```
$ x=*
$ echo \$x
$x

$ echo \\
\
$
```

using single quotes to accomplish the above task:

```
$ echo '\'
\
```

```
$

The exception
-------------

\c is essentially equivalent to 'c' . The one exception to this rule is when the
backslash is used as the very last character on the line


$ lines=one'
> 'two
$ echo "$lines"
one
two
$ lines=one\
> two
$ echo "$lines"
onetwo
$

When a backslash is the last character of a line of input, the shell treats it
as a line continuation character.

It removes the newline character, it's as if it wasn't even typed.

***
This construct is often used for entering long commands across multiple lines.
***

Backslash in the Double Quotes
------------------------------

Backslash is one of the three characters interpreted by the shell inside double
quotes.

$ echo "\$x"
$x

$ echo "\ is the backslash character"
\ is the backslash character


$ x=5
$ echo "The value of x is \"$x\""
The value of x is "5"
$


Difficult one
-------------

$ x=22

Let's say that you want to display the following line at the terminal wherein 22
is result of $x:

<<< echo $x >>> displays the value of x, which is 22

$
$ echo <<< echo $x >>> displays the value of x, which is $x
syntax error: '<' unexpected
```

```
$
```

If you put the entire message inside single quotes, the value of x won't be substituted at the end. If you enclose the entire string in double quotes, both occurrences of $x will be substituted.

???

Two different ways to properly quote the above::

```
$ echo "<<< echo \$x >>> displays the value of x, which is $x"

$ echo '<<< echo $x >>> displays the value of x, which is ' $x
```

Explanation::

In the first case, everything is enclosed in double quotes, and the backslash is used to prevent the shell from performing variable substitution in the first instance of $x .

In the second case, everything up to the last $x is enclosed in single quotes but the variable that should be substituted is added without quotes around it.

Word of caution:: If x=* then the second form would be a problem!!
Solution:: include $x in DQs

Backquotes

There are two ways in the shell to perform command substitution: by enclosing the command in back quotes or surrounding it with the $(...) construct.

```
$ echo Date and time is `date`
Date and time is Sat Mar 18 21:47:15 IST 2017

$ echo Date and time is $(date)
Date and time is Sat Mar 18 21:48:13 IST 2017
```

Which form is better?

Complex commands that use combinations of forward and back quotes can be difficult to read, particularly if the typeface you're using doesn't visually differentiate between single and back quotes;

$(...) constructs can be easily nested, allowing command substitution within command substitution.

Nesting can also be performed with back quote, but its tricky.

How many users are logged in?

```
$ echo There are $(who|wc -l) users logged in

$ filelist=$(ls)
$ echo $filelist
$ echo "$filelist"
```

Any difference?

Command substitution is often used to change the value stored in a shell
variable
```
$ name="M M Joshi"
$ name=$(echo $name | tr '[a-z]' '[A-Z]')
$ echo $name
```

???

Next::

```
$ filename=/users/steve/memos
$ filename=$(echo $filename | tr "$(echo $filename | cut -c1)" "^")
$ echo $filename
```

???

==

Session 4

40 Command line arguments
==========================
42 Read and echo commands in shell scripts
==

Shell scripts can accept command-line arguments and options
just like other Linux commands. Within your shell script, you can refer
to these arguments as $1 , $2 , $3 , and so on upto $9. After 9th one, we need
to use {}. So argument 10th can be accessed by ${10}.

$0 represents command itself.

$# provides total number of arguments.
$* provides all the arguments at one go

shift built-in command shifts the arguments by one position to left.

Thus, if your program has been given following arguments,

$ prog one two three

then $1=one, ... $3=three

but if you use shift command once in your prog, then

$1=two and $2=three and $3=

You can shift n number of positions using syntax shift n

Example
----

cla.sh

41 Arithmetic in shell scripts
===============================

```
expr command
------------

expr command must see each operator and operand as a separate argument.

Incorrect usage
$ expr 1+2

$ expr 1 + 2
3
$

$ expr 10 + 20 / 2
20
$

$ expr 17 * 6
expr: syntax error
$

$ expr "17 * 6"
17 * 6
$

Use backslash!

$ expr 17 \* 6
102
$

Quiz::

One of the most frequently used is the : operator in the expr command.

It is used to match characters in the first operand against a regular expression
given as the second operand. By default, it returns the number of characters
matched.

$ file="My name is "
$ expr "$file" : ".*"

Output???

Another route
-------------

The format for arithmetic expansion is

$((expression))

where expression is an arithmetic expression using shell variables and
operators.

$ echo $(( 8#100 ))
64
$ echo $(( 2#101010101010101010 ))
174762

The result of computing expression is substituted on the command line.

For example,
```

```
$ echo $((i+1))
```

adds one to the value in the shell variable i and prints the result.

Notice that the variable i doesn't have to be preceded by a dollar sign because the shell knows that the only valid elements that can appear in arithmetic expansions are operators, numbers, and variables.

If the variable is not defined or contains a NULL string, its value is assumed to be zero.

So if we have not assigned any value yet to the variable a , we can still use it in an integer expression:

```
$ echo $a
$
echo $(( a = a + 1 ))
1
```
Parentheses may be used freely inside expressions to force grouping, as in

```
$ echo $((i = (i + 10) * j))
```

If you want to perform an assignment without echo or some other command, you can move the assignment before the arithmetic expansion.

So to multiply the variable i by 5 and assign the result back to i you can write

```
i=$(( i * 5 ))
```

OR

```
$(( i *= 5 ))
```

```
$(( i++ ))
```

Finally, to test to see whether i is greater than or equal to 0 and less than or equal to 100, you can write

```
result=$(( i >= 0 && i <= 100 ))
```

which assigns result the value of 1 (true) if the expression is true or 0 (false) if it's false:

```
$ i=$(( 100 * 200 / 10 ))
$ j=$(( i < 1000 ))
$ echo $i $j
2000 0
$
```


43 Taking decisions:
     o if-then-fi

     Syntax::

     if command
     then
     command
     ...
```

```
     fi

     o if-then-else-fi

     if command
     then
     command
     ...
     else
     command
     ...
     fi

     o The test command (file tests, string tests)

test expression

test must see all operands and operators as separate.

Example::

$ day=monday
$ if test $day = monday
> then
> echo "TRUE"
> else
> echo "FALSE"
> fi
TRUE
$
Now set day to no value (NULL)

$ day=

What do you expect?

What is the solution?
placeholder

String operations using test
============================

Operator                       Returns TRUE (exit status of 0) if
========                       ==================================
string1 = string2              string1 is identical to string2
string1 != string2             string1 is not identical to string2
string                         string is not null
-n string                      string is not null, length > 0
-z string                      string is null (zero length)

$ blanks="    "
$ test $blanks
$ echo $?
1
$ test "$blanks"
$ echo $?
0
$

The -n operator returns an exit status of zero if the argument that follows is
```

not null. This operator is testing for nonzero length.

```
$ nullvar=
$ nonnullvar=abc
$ test -n "$nullvar"
$ echo $?
1
$ test -n "$nonnullvar"
$ echo $?
0
$ test -z "$nullvar"
$ echo $?
0
$ test -z "$nonnullvar"
$ echo $?
1
```

test expression
This can also be expressed in the alternative format as
[expression]

```
Operator               Returns TRUE (exit status of 0) if
-------                ---------------------------------
int1 -eq int2          int1 is equal to int2
int1 -ge int2          int1 is greater than or equal to int2
int1 -gt int2          int1 is greater than int2
int1 -le int2          int1 is less than or equal to int2
int1 -lt int2          int1 is less than int2
```

```
$ x1="005"
$ x2=" 10"
$ [ "$x1" = 5 ]
$ echo $?
1
$ [ "$x1" -eq 5 ]
$ echo $?
0
$ [ "$x2" = 10 ]
$ echo $?
1
$ [ "$x2" -eq 10 ]
$ echo $?
0
```

```
Operator        Returns TRUE (exit status of 0) if
--------        ---------------------------------
-d file         file is a directory
-e file         file exists
-f file         file is an ordinary file
-r file         file is readable by the process
-s file         file has nonzero length
-w file         file is writable by the process
-x file         file is executable
-L file         file is a symbolic link
------------------------------------------------------------
```

The Logical Negation Operator !

The Logical AND Operator –a

The Logical OR Operator –o
The -o operator has lower precedence than the -a operator.

You can use parentheses in a test expression to alter the order of evaluation as
needed. Make sure that the parentheses themselves are quoted!

Example::
a=0
b="003"
c="10"

$ ["$a" -eq 0 -o "$b" -eq 2 -a "$c" -eq 10]
echo $?
???

      o Nested if-elses

      if command 1
      then
        command
        ...
      elif command 2
      then
        command
        ...
      elif command n
      then
        command
        ...
      else
        command
        ...
      fi

      o The case control structure

       case value in
       pattern1)
       command
       ...
       command;;
       pattern2)
       command
       ...
       command;;
       ...
       patternn)
       command
       ...
       command;;
       esac

Exit Status

Built in variable $? contains status of previous command's execution in terms of

success/failure. If failure, probable causes of failure.

Refer to es.sh