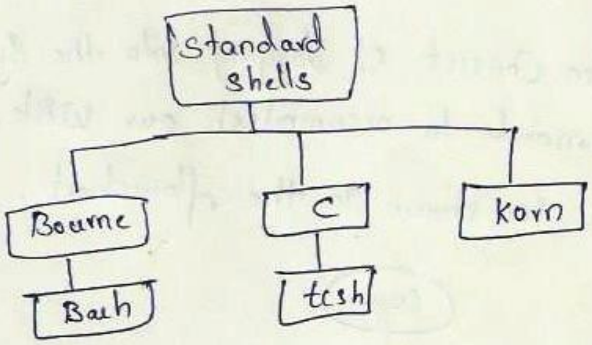
Unit - II

UNIX operating System Contains four distinct parts:

1. The Kernel
2. The shell
3. Utilities
4. Applications programs

The shell is the part of UNIX that is most visible to user.
→ It receives and interprets the Commands entered by the user, It the most important component of UNIX structure.



There are two major parts to a shell. The first is the Interpreter which reads your Commands & works with the Kernel to execute them.

Second part of the shell is a programming Capability, that allows you to write a shell (command) script

A shell script is a file that Contains shell Commands the perform a useful function. It is also known as shell program.

Three traditional shells are Osed in UNIX today the Bourne shell is developed by steve Bourne at the AT&T dabs, is the oldest.

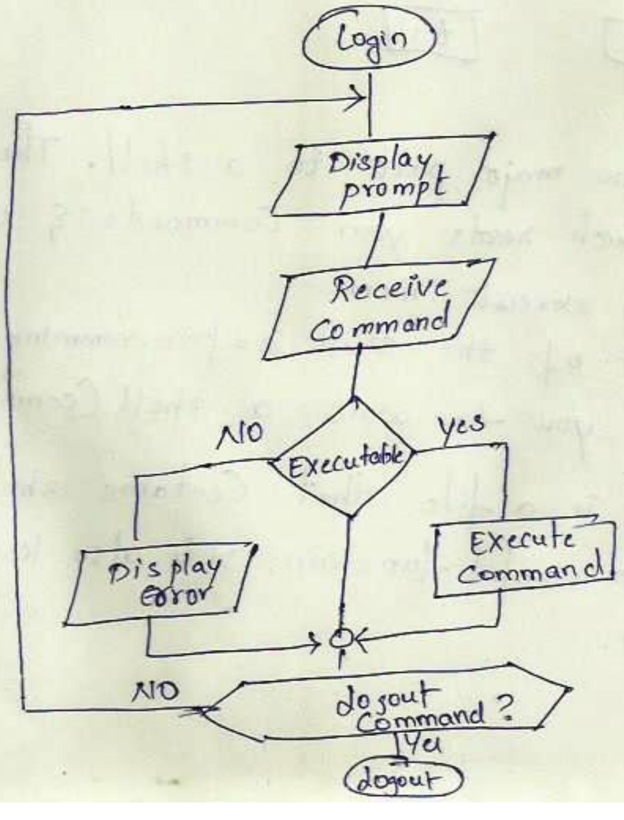
An enchanced Version of the Bourne shell is called Bash (Bourne again shell) is Used in Linux.

The C shell developed in Berkeley by Bill Joy. & Korn shell developed by David korn in AT&T dabs.

UNIX Session

A UNIX Session Consists of dogging into the System and then executing Commands to accomplish our work.

The work flow is shown in the flowchart.



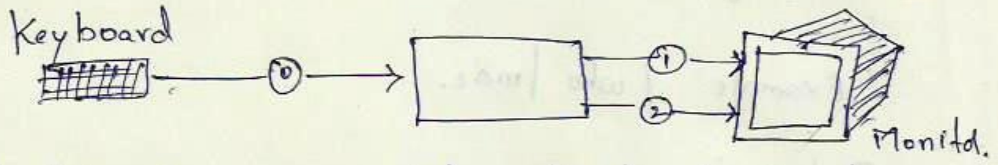
Standard Streams

UNIX defines three standard streams that are used by Command.

→ Each Command takes its input from a stream known as standard input.

→ Commands that create output send it to a stream known as standard output.

If an executing Command encounter an error, the error message is sent to standard error.



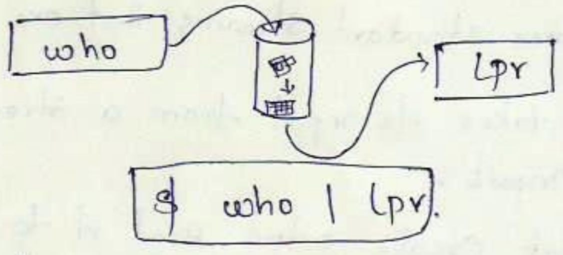
Standard input is associated with the keyboard, standard output is associated with the Monitor & standard error

*** We can change the default file association using pipes (or) redirection.

Pipes

Pipe is an operator that temporarily saves the output of one command in a buffer that is being used at the same time as the input of the next Command.

→ The first command must be able to send its output to standard output, the second Command must be able to read its input from standard input.



i.e The left Command must be able to Send data to standard output & the right Command must be able to receive data from standard input.

The Symbol of pipe is Vertical bar (|).

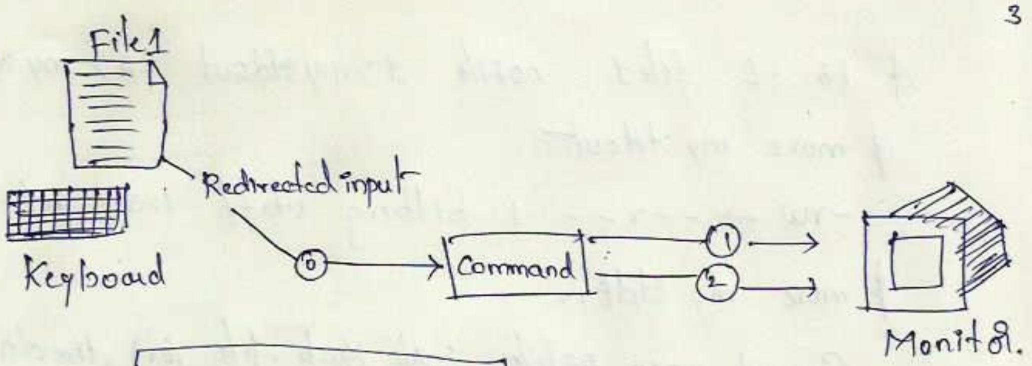
Example $ who | more.

Redirection

Redirection is the process by which we specify that a file is to be used in place of one of standard files.
With input file, we call it input redirection, with output files, we call it output redirection. & with the error file we call it error redirection.

Input Redirection

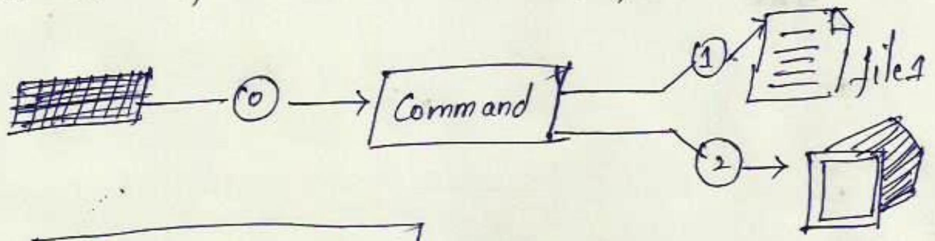
Redirecting standard input from the keyboard to any text file.
The input redirection operator is the less-than character (<).
An arrow pointing to a Command, meaning that the Command is to get its input from the designated file.



$\boxed{\text{Command < file1.}}$

Output Redirection

We redirect standard output, that the Command's output is copied to a file rather than displayed on the monitor



```
Command > file1
Command >| file1
Command >> file1
```

If noclobber option is turned on, it prevents redirected output from destroying an existing file.

```
$ who > whooct2
$ who >| who oct2
$ who >> whooct2.
$ ls -l file1  nofile  1>mystdout  2> mystderr
$ more mystdout
  -rw -r--r-- 1 bilberg staff 1234 oct 2 15:16 files
$ more my stdErr
   Cannot open nofile : No Such file (or) directory.
```

Here document

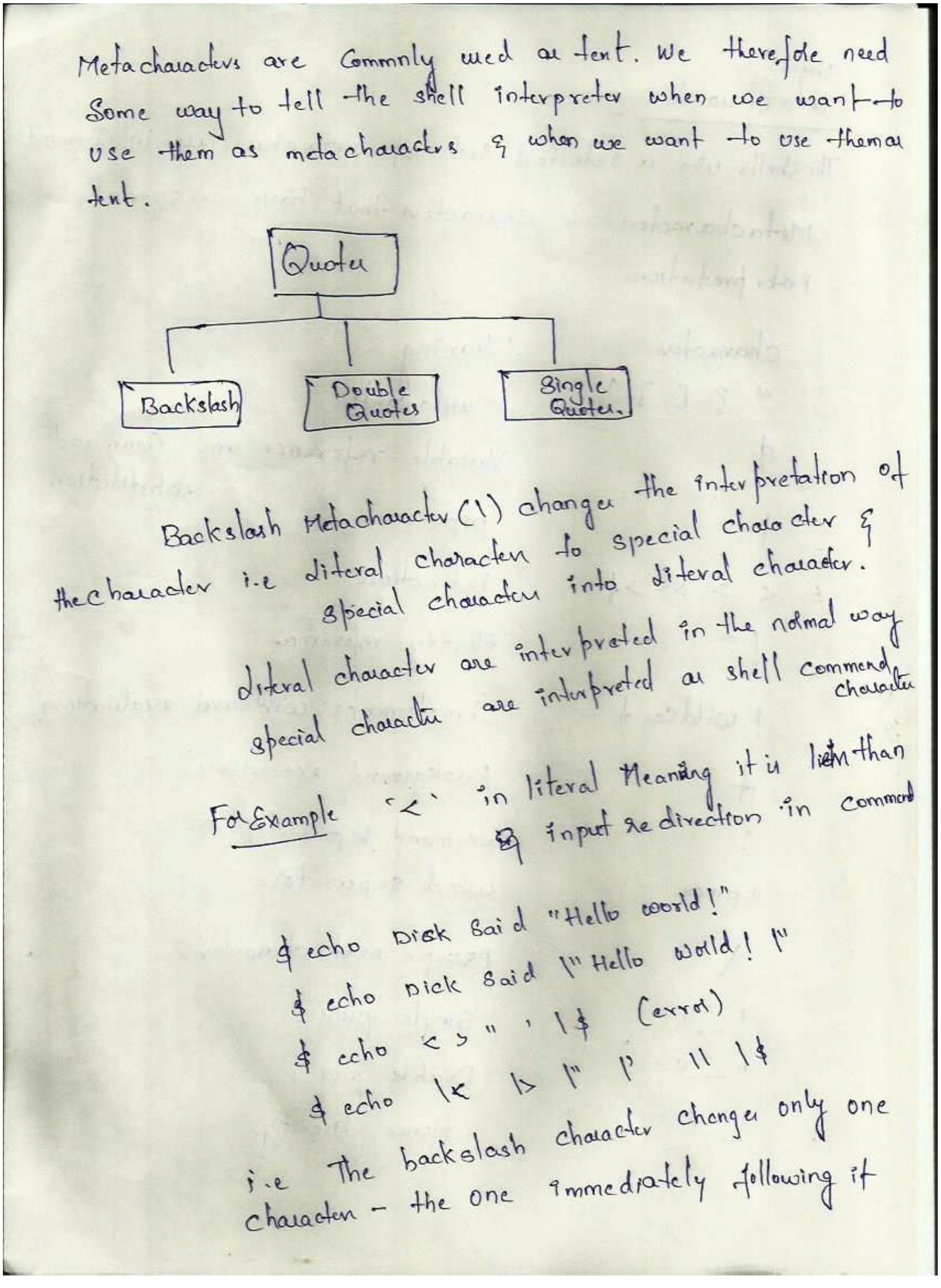
when we want to include the text in the Script itself rather than read it from a file. This is done with the Here document operator (<<).

Shell Metacharacters :-

TheShells use a selected set of metacharacters in Command.

Metacharacters are characters that have a special interpretation.

Character	Meaning
* ? [ ] ^	wild cards
$	Variable reference and Command substitution
|	pipe
< > >> >&	Redirection
! ^	History references
! wildcard	Complements wildcard evaluation.
&	Background execution
;	Command separator
space	word separator
\	Escape next character.
'...'	Single quotes
"..."	Double quotes.
~	Home directory.



Metacharacters are Commnly used as text. We therefore need Some way to tell the shell interpreter when we want to use them as metacharacters & when we want to use them as text.

Quotes
Backslash
Double Quotes
Single Quotes.

Backslash Metacharacter (\) changes the interpretation of the character i.e literal characters to special character & special characters into literal character.

special characters are interpreted in the normal way

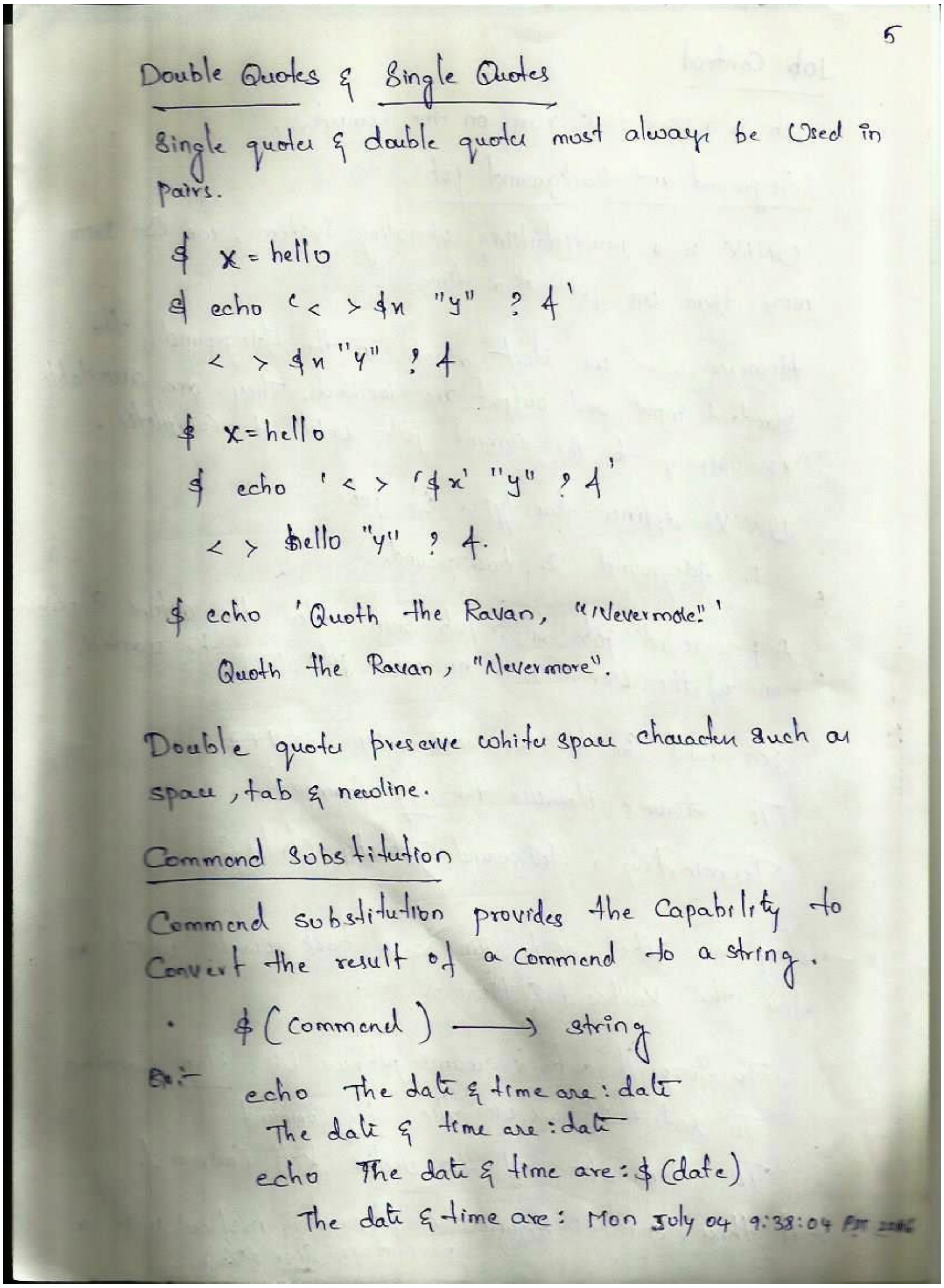
literal character are interpreted as shell command character

special character are interpreted

For Example '<' in literal Meaning it is less than & input redirection in command

```
$ echo Dick Said "Hello world!"
$ echo Dick Said \"Hello world! \"
$ echo < > " ' \$    (error)
$ echo \< \> \" \' \\ \$
```

i.e The backslash character changes only one character - the one immediately following it



Double Quotes & Single Quotes

Single quotes & double quotes must always be used in pairs.

```
$ x = hello
$ echo '< > $x "y" ? 4'
  < > $x "y" ? 4
$ x=hello
$ echo '< > '$x' "y" ? 4'
  < > hello "y" ? 4.
$ echo 'Quoth the Ravan, "Nevermore."'
    Quoth the Ravan, "Nevermore".
```

Double quotes preserve white space characters such as space, tab & newline.

Command substitution

Command substitution provides the capability to convert the result of a command to a string.

- $(command) ⟶ string

Ex:-

```
echo The date & time are: date
 The date & time are: date
echo The date & time are: $(date)
   The date & time are: Mon July 04 9:38:04 PST 2016
```

Job Control

Job is a user task run on the Computer.

Foreground and Background jobs.

UNIX is a multitasking operating System, we Can run more than one job at a time.

However, when we start a job in the foreground, the Standard input and output are locked. They are available exclusively to the current job until it completes.

UNIX defines two types of jobs
1. foreground 2. background.

A Foreground job is a job run under the active Supervision of the user. And no other jobs may be started.

Foreground Can be Suspended by ctrl+z.
To resume it again use the fg Command
Terminating a foreground job by ctrl+c

→ Long time jobs Can be run has a background. which are free the keyboard & Monitor.

To Suspend a back ground job, we Use stop Command.
To restart it, we use the bg Command
To terminate it, we use the kill Command.

All three Commands require the job number, prefaced with a percentage Sign (%).

moving Between Background & Foreground.

To move a job between the foreground and background the job must be suspended.

One the job is Suspended, we can move it from the Suspended stat to the background with the bg command

To move a back ground job to the foreground we use the fg Command.

```
$ longJob.scr
^z
[1] + stopped   longJob.scr
$ bg
[1]   longJob.scr
$ fg %1
dongJob.scr.
```

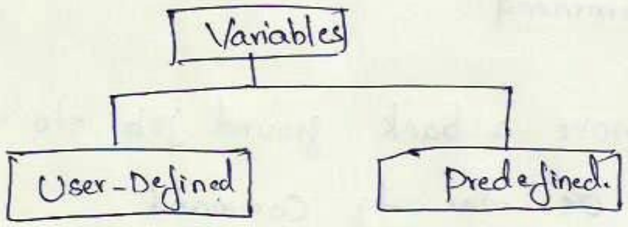
• To list the Current jobs and their status, we use the jobs Command

```
$ jobs.
```

This Command lists all jobs whether (or) not they are running (or) stopped.

Shell Variables

A Variable is a location in memory where values can be stored. Each shell Variable must have a name. The name of a Variable must start with an alphabetic (or) underscore (_) character followed by zero (or) more alphanumeric (or) underscore characters.



User-Defined Variables

	Syntax
Assignment	Variable = Value
Reference	$Variable.

It uses assignment operator "=" to store values in a Variable.

Example

```
$ X=23
$ echo $x
23
$ x=hello
$ echo $x
hello.
$ x= "Go Don's"
$ echo $x
Go Don's.
```

Here DocumentA **here document** is a form of quoting that allows shell variables to be substituted. It's a special form of redirection that starts with **<<WORD** and ends with **WORD** as the only contents of a line. In the Bourne shell you can prevent shell substitution by escaping **WORD** by putting a \ in front of it on the redirection line, i.e. **<<\WORD**, but not on the ending line.

The following scripts illustrate this,

for the **Bourne shell**:

```
#!/bin/sh
does=does
not=""
cat << EOF
This here document
$does $not
do variable substitution
EOF
cat << \EOF
This here document
$does $not
do variable substitution
EOF \EOF
```

the output:

```
This here document
does
do variable substitution
This here document
$does $not
do variable substitution
```

In the top part of the example the shell variables **$does** and **$not** are substituted. In the bottom part they are treated as simple text strings without substitution.

PREDEFINED LOCAL VARIABLES:

Some predefined local variables in Bourne shell having special meaning are listed below:

$@ : an individually quated list of all the positional parameters

$# : the number of positional parameters

$? : the exit value of th last command

$! : the process id of the lst background command

$- : the current shell option assigned from the command line

expr *expression*

The command **expr** evaluates **expression** and sends the result to the standard output. All of the components of the expression must be seperated by blanks, and allof the shell metacharacters must be escaped by a \. In an expression the following operators may be used.

*** / %**	: the number of positional parameters
+ -	: the exit value of the last command
= \> \>= \< \<= !=	: the comparison operators
\&	: logical "and"
\\|	: logical "or"

Escaped parantheses **(** and **)** may be used to explicitly control the order of evalution.

CONDITIONAL EXPRESSIONS

The utility **test** returns a 0 exit status if the given expression evaluates to true; it returns a non-zero exit status otherwise. The exit status of the test command is typically used by the shell control structures for branching purposes. The syntax is as follows:

test *expression*

or equivalently the following may be used instead of the above form

[*expression***]**

The expression may be written in the following forms

str1=*str2* : true if *str1* is equal to *str2*
*str1***!=***str2* : true if *str1* is not equal to *str2*
string : true if *string* is not null
int1 **–eq** *int2* : true if *int1* is equal to *int2*
int1 **–ne** *int2* : true if *int1* is not equal to *int2*
int1 **–gt** *int2* : true if *int1* is greater than *int2*
int1 **–ge** *int2* : true if *int1* is greater or equal to *int2*
int1 **–lt** *int2* : true if *int1* is less than *int2*
int1 **–le** *int2* : true if *int1* is less or equal to *int2*
!*expr* : true if *expr* is false
expr1 **–a** *expr2* : true if *expr1* and *expr2* are both true
expr1 **–o** *expr2* : true if *expr1* or *expr2* is true
\**(***expr*\**)** : escaped parantheses are used for grouing expressions

CONTROL STRUCTURES:

Conditional if: The **conditional if** statement is available in both shells, but has a different syntax in each.

if condition1
then
command list if condition1 is true
[***elif*** condition2
then command list if condition2 is true]
[***else***
command list if condition1 is false]
fi

The conditions to be tested for are usually done with the ***test***, or ***[]*** command .
The **if** and **then** must be separated, either with a <newline> or a semicolon (**;**).

```
#!/bin/sh
if [ $# -ge 2 ]
then
echo $2
elif [ $# -eq 1 ]; then
echo $1
else
echo No input
fi
```

There are required spaces in the format of the conditional test, one after **[** and one before **]**. This script should respond differently depending upon whether there are zero, one or more arguments on the command line. First with no arguments:

```
$ ./if.sh
No input
```

Now with one argument:

```
$ ./if.sh one
one
```

And now with two arguments:

```
$ ./if.sh one two
Two
```

while ... do ... done
The **while** command executes the commands in *list2* as long as the last command in *list1* succeeds.
while *list1*
do
list2
done

The following commands can be used to control loops
break: causes the loop to end immediately
loop: causes the loop jump immediately to the next iteration

until ... do ... done
The **until** command executes the commands in *list2* as long as the last command in *list1* fails.
until *list1*
do
list2
done

```
$ cat until.sh
x=1
until [$x –gt] 3
do
```

case ..in ...esac
The case command supports multi-way branching based on the value of a single string and has the following syntax
case *expression* **in**
*pattern***{|***pattern***}*)**
list
;;
Esac

for ... do ... done

The for comman allows a list of commnds to be executed several times, using a differnt value of the loop
avriable during each iteration.

for *name* **[in {***word***}*]**
do
list
done

The **for** command loops the value of the variable name through each word in the word list, evaluating the commands list after each iteration. İf no word list is supplied, $@ (i.e. all positional parameters) is used instead.

SHELL SCRIPT EXAMPLES:

1.$ cat for.sh

```
for color in red yelow blue
do
echo one color is $color
done
```

Output:

```
$ for.sh
one color is red
one color is yellow
```

2. **$ cat if.sh**

```
echo –n ’enter a number:’
read number
if [$number –lt 0]
then
echo negative
elif [$number –eq 0]
then
echo zero
else
echo positive
fi
```

Output:

```
$ if.sh
enter a number: 1
positive
$ if.sh
enter a number: -1
negative
```