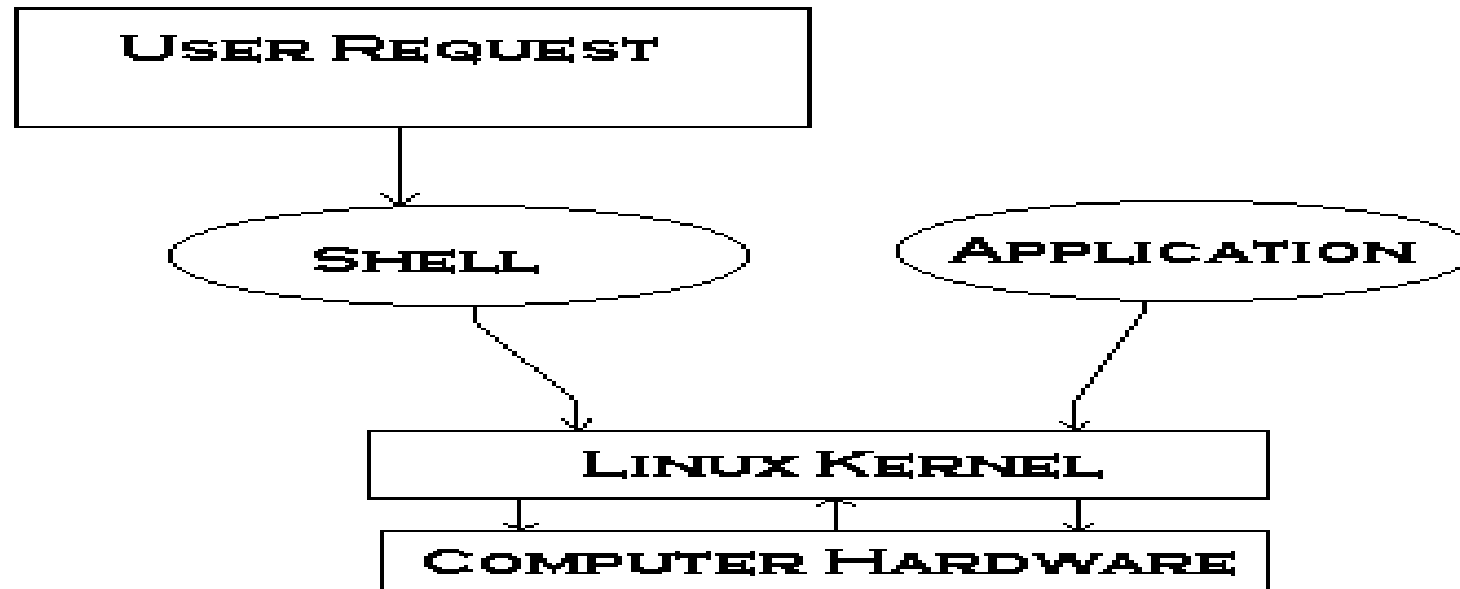# What is shell?

A shell is a program that takes commands typed by the user and calls the operating system to run those commands.
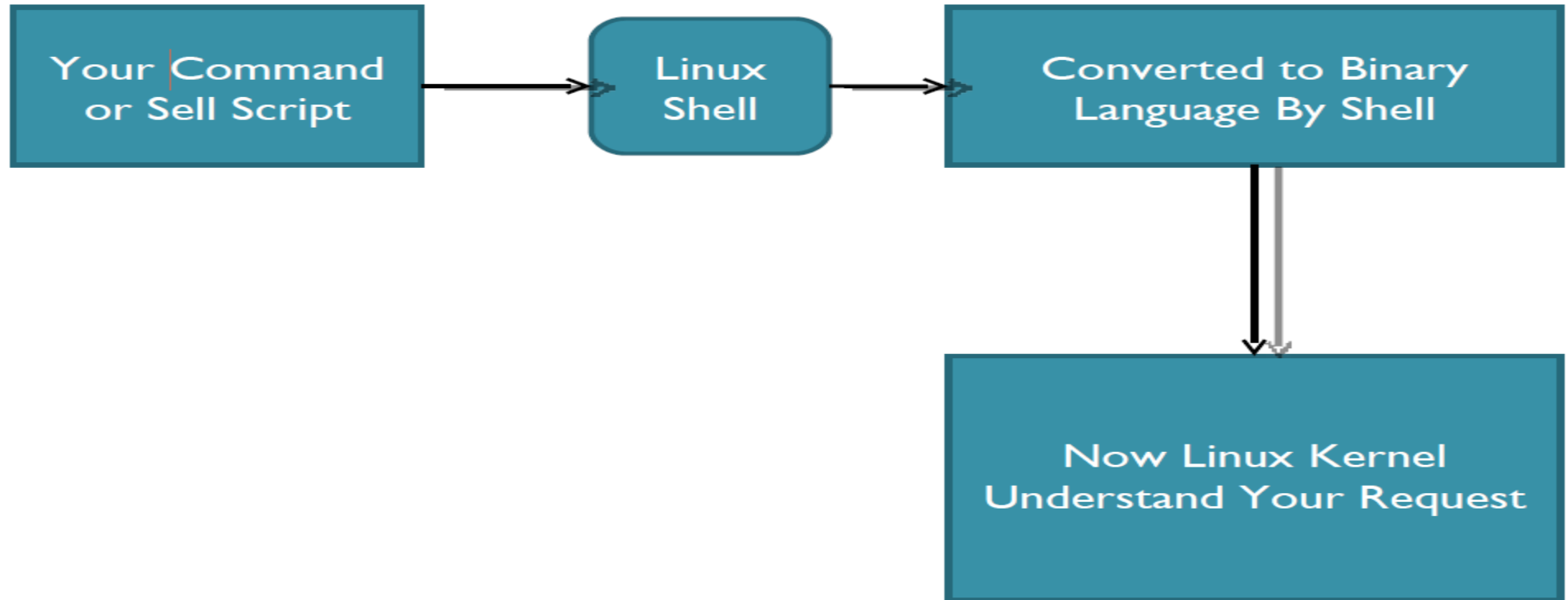
Shell accepts your instruction or commands in English and translate it into computers native binary language.
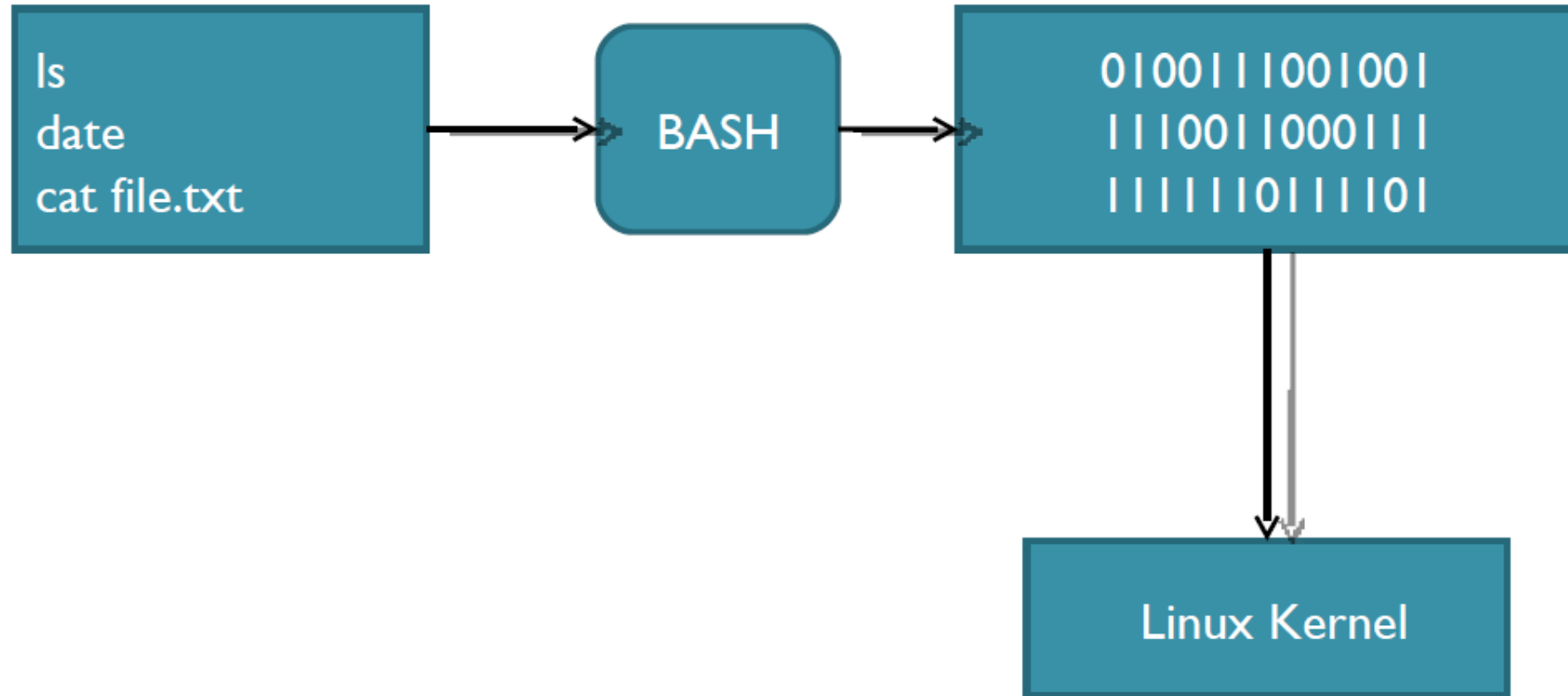
```
┌─────────────────────────┐
│     USER REQUEST        │
└─────────────────────────┘
            │
            ▼
      (  SHELL  )        ( APPLICATION )
            │                    │
            ▼                    ▼
┌─────────────────────────────────────┐
│          LINUX KERNEL               │
└─────────────────────────────────────┘
┌─────────────────────────────────────┐
│        COMPUTER HARDWARE            │
└─────────────────────────────────────┘
```

Kind of Shells

- *Bourne Shell*
- *C Shell*
- *Korn Shell*
- *Bash Shell*
- *Tcsh Shell*

This is what Shell Does for US

Example


```
ls
date
cat file.txt
```

BASH

```
0100111001001
1110011000111
1111110111101
```

Linux Kernel

Shell is an command language interpreter that executes commands read from the standard input device (keyboard) or from a file.

Changing Your Default Shell

Tip: To find all available shells in your system type following command:

$ cat /etc/shells

The basic Syntax :

chsh username new_default_shell

The administrator can change your default shell.

Shell Scripting

Shell script is a series of command(s) stored in a plain text file.

A shell script is similar to a batch file in MSDOS, but is much more powerful.

Practical examples where shell scripting actively used:

1. Monitoring your Linux system.

2. Data backup and creating snapshots.

3. Find out what processes are eating up your system resources.

4. Find out available and free memory.

5. Find out all logged in users and what they are doing.

6. Find out if all necessary network services are running or not.

Create a script

As discussed earlier shell scripts stored in plain text file, generally one command per line.

◦ *vi myscript.sh*

Make sure you use .bash or .sh file extension for each script. This ensures easy identification of shell script.

Setup executable permission

- Once script is created, you need to setup executable permission on a script. Why?

- Without executable permission, running a script is almost impossible.

- Besides executable permission, script must have a read permission.

- Syntax to setup executable permission:

◦ *$ chmod +x your-script-name.*
◦ *$ chmod 755 your-script-name.*

Run a script (execute a script)

Now your script is ready with proper executable permission on it. Next, test script by running it.

◦ *bash your-script-name*

◦ *sh your-script-name*

◦ *./your-script-name*

 Examples

◦ $ bash bar

◦ $ sh bar

◦ $ ./bar

Example

```
$ vi first
```

```
#
# My first shell script
#
clear
echo "This is my First
script"
```

```
$ chmod 755 first


$ ./first
```

Using Exit code($?)

Every Linux or Unix command executed by the shell script or user has an exit status.

Exit status is an integer number. 0 exit status means the command was successful without any errors.

A non-zero (1-255 values) exit status means command was a failure.

Example

```
date                    test 1 –gt 0 ;
echo $?                 echo $?
```

Variables in Shell

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

○ **System variables** - Created and maintained by Linux itself. This type of variable defined in CAPITAL LETTERS.

○ **User defined variables (UDV)** – Created and maintained by user. This type of variable defined in lower letters.

System Variables

Variable	Description
PATH	This variable contains a colon (:)-separated list of directories in which your system looks for executable files. The search path.
USER	The username
HOME	Default path to the user's home directory
EDITOR	Path to the program which edits the content of files
UID	User's unique ID
TERM	Default terminal emulator
SHELL	Shell being used by the user

Export command- One can change the shell variable into environment or system variable using export command.

```
[ubantu@localhost ~]$ echo $EXPORT

[ubantu@localhost ~]$ export EXPORT="New Value
"
[ubantu@localhost ~]$ echo $EXPORT
New Value
```

HISTSIZE vs HISTFILESIZE

`HISTSIZE` is the number of lines or commands that are stored in memory in a history list while your bash session is ongoing.

`HISTFILESIZE` is the number of lines or commands that (a) are allowed in the history file at startup time of a session, and

(b) are stored in the history file at the end of your bash session for use in future sessions.

- echo $HISTSIZE
- O/P- 1000
- HISTSIZE=10
- Echo $HISTSIZE
- o/p-10

Similarly HISTFILESIZE

Alias command

- Used to customize the shell variables.
- Aliases are shorthand for longer expressions.

alias myls= "ls -l"

User defined variables (UDV)

To define UDV use following syntax:

◦ *variable name=value*

◦ *$ no=10*

Rules for Naming variable name

- Variables must begin with a letter.

- Spaces are not allowed.

- Underscore can be allowed.

- No special character in variable name.

- Variables are case-sensitive.

Print or access value of UDV

To print or access UDV use following

syntax :

◦ *$variablename.*

  Examples:

◦ $vech=Bus

◦ $ n=10

◦ $ echo $vech

◦ $ echo $n

Don't try

- ***$ echo vech***
- it will print vech instead its value 'Bus'.
- ***$ echo n***
- it will print n instead its value '10'.

"You must *use $ followed by variable name.*"

Class work

1. Define variable x with value 10 and print it on screen.

2. Define variable xn with value SUST and print it on screen.

3. print sum of two numbers, let's say 6 and 3 .

Example of UDV

A=10

Ba=20

BA=30

HOSTNAME=$ (hostname)

DATE=`date`

123var=333

wrong@var=False

Hyphen-var=Falsehyphen value

Positional Variables(Parameters)

- These are also called command line arguments.
- $0 name of the script
- $1 first argument
- $2 second argument
- $3 third argument
- $# total number of arguments
- $* value of all arguments

Positional Variables(Parameters)

$vim filename.sh

touch $1 $2

echo "files created"

echo "Total number of arguments passed $#"

echo "vale of the arguments passed is $*"

Save this and make it executable and run

$./filename.sh abc def (abc and def are file names which are passed as arguments)

Shell Arithmetic

Syntax:

- *expr op1 math-operator op2*

  Examples:

- $ expr 1 + 3

- $ expr 2 – 1

- $ expr 10 / 2

- $ expr 20 % 3

- $ expr 10 * 3

- $ echo `expr 6 + 3`

Shorthand

Shorthand	Meaning
$ ls *	will show all files
$ ls a*	will show all files whose first name is starting with letter 'a'
$ ls *.c	will show all files having extension .c
$ ls ut*.c	will show all files having extension .c but file name must begin with 'ut'.
$ ls ?	will show all files whose names are 1 character long
$ ls fo?	will show all files whose names are 3 character long and file name begin with fo
$ ls [abc]*	will show all files beginning with letters a,b,c

The read Statement

- Use to get input (data from user) from keyboard and store (data) to variable.
- *Syntax:*
  - read variable1, variable2,...variableN

```
echo "Your first name please:"
read fname

echo "Hello $fname, Lets be friend!"
```

Shell Relational

Math- ematical Operator in Shell Script	Meaning	Normal Arithmetical/ Mathematical Statements	But in Shell	
			For test statement with if command	For [expr] statement with if command
-eq	is equal to	5 == 6	if test 5 -eq 6	if expr [5 -eq 6]
-ne	is not equal to	5 != 6	if test 5 -ne 6	if expr [5 -ne 6]
-lt	is less than	5 < 6	if test 5 -lt 6	if expr [5 -lt 6]
-le	is less than or equal to	5 <= 6	if test 5 -le 6	if expr [5 -le 6]
-gt	is greater than	5 > 6	if test 5 -gt 6	if expr [5 -gt 6]
-ge	is greater than or equal to	5 >= 6	if test 5 -ge 6	if expr [5 -ge 6]

Test command or [expr]

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

Syntax: test expression OR [expression]

The following examples demonstrate the use of Bash's test command syntax and Bash's numeric comparison operators.

```
[user@host ~]$ [ 1 -eq 1 ]; echo $?
0
[user@host ~]$ [ 1 -ne 1 ]; echo $?
1
[user@host ~]$ [ 8 -gt 2 ]; echo $?
0
[user@host ~]$ [ 2 -ge 2 ]; echo $?
0
[user@host ~]$ [ 2 -lt 2 ]; echo $?
1
[user@host ~]$ [ 1 -lt 2 ]; echo $?
0
```

The following examples demonstrate the use of Bash's string comparison operators.

```
[user@host ~]$ [ abc = abc ]; echo $?
0
[user@host ~]$ [ abc == def ]; echo $?
1
[user@host ~]$ [ abc != def ]; echo $?
0
```

Shell Logical Expressions

!   Not   <u>not</u> must be enclosed within [   ]

&&   and

|| or   <u>and, or</u>

                           must be enclosed within

                           [[   ]]

Example: Using the ! Operator

```bash
#!/bin/bash

read -p "Enter years of work: "Years

if [ ! "$Years" -lt 20 ]; then

   echo "You can retire now."

else

   echo "You need 20+ years to
retire"

fi
```

Example: Using the (−a) or && Operator

The logical AND (-a) operator will give true if both the operands are true. Otherwise, false.

In the following example we will check if the number is even and greater than 10.

```
#!/bin/sh
# take a number from the user
echo "Enter a number: "
read a
# check
if [ `expr $a % 2` == 0 -a $a -gt 10 ]
then
  echo "$a is even and greater than 10."
else
  echo "$a failed the test."
fi
```

Example: Using the (–o) or || Operator

The logical OR -o operator will give true if any one of the operand is true. If both operands are false then it will return false.

In the following example we will check if entered number is either odd or less than 10.

```
#!/bin/sh
# take a number from the user

echo "Enter a number: "
read a
# check

if [ `expr $a % 2` != 0 -o $a -lt 10 ]
then
  echo "$a is either odd or less than 10."
else
  echo "$a failed the test."
fi
```

File Test Operators

- d : True if the file exists and is a directory.

- e : True if the file exists.

- f : True if the file exists and is a regular file.

- r : True if the file exists and is readable.

- s : True if the file exists and has a size greater than zero.

- w : True if the file exists and is writable.

- x : True if the file exists and is executable.

- L/h : True if the file exists and is a symbolic link.

if condition

Syntax:

if condition

then

command1 if condition is true or if

exit status

of condition is 0 (zero)

fi

Example

```
$ vim myscript.sh

read choice
if [ $choice -gt 0 ]; then
echo "$choice number is positive"
else
echo "$ choice number is negative"
fi
```

psacct.service-The **psacct service** is responsible for starting and stopping process accounting at system boot time and at system shutdown.

The following code section demonstrates the use of an `if/then/else` statement to start the psacct service if it is not active and to stop it if it is active.

```
[user@host ~]$ systemctl is-active psacct > /dev/null 2>&1
[user@host ~]$ if  [ $? -ne 0 ]; then
> sudo systemctl start psacct
> else
> sudo systemctl stop psacct
> fi
```

Use of if/then/elif/then/else

The following code section demonstrates the use of an `if/then/elif/then/else` statement to run the **mysql** client if the `mariadb` service is active, run the **psql** client if the `postgresql` service is active, or run the **sqlite3** client if both the `mariadb` and `postgresql` services are not active.

```
[user@host ~]$ systemctl is-active mariadb > /dev/null 2>&1
MARIADB_ACTIVE=$?
[user@host ~]$ sudo systemctl is-active postgresql > /dev/null 2>&1
POSTGRESQL_ACTIVE=$?
[user@host ~]$ if  [ "$MARIADB_ACTIVE" -eq 0 ]; then
> mysql
> elif  [ "$POSTGRESQL_ACTIVE" -eq 0 ]; then
> psql
> else
> sqlite3
> fi
```

```
# The following THREE if-conditions
produce the same result

* DOUBLE SQUARE BRACKETS
read -p "Do you want to continue?"
reply
if [[ $reply = "y" ]]; then
    echo "You entered " $reply
fi

* SINGLE SQUARE BRACKETS
read -p "Do you want to continue?"
reply
if [ $reply = "y" ]; then
    echo "You entered " $reply
fi

* "TEST" COMMAND
read -p "Do you want to continue?"
reply
if test $reply = "y"; then
    echo "You entered " $reply
```

Nested if-else-fi

```
$ vi nestedif.sh

echo "1. Unix (Sun Os)"
echo "2. Linux (Red Hat)"
echo -n "Select your os choice [1 or 2]? "
read osch
if [ $osch -eq 1 ] ; then
        echo "You Pick up Unix (Sun Os)"
else
        if [ $osch -eq 2 ] ; then
                echo "You Pick up Linux (Red Hat)"
        else
                echo "What you don't like Unix/Linux OS."
        fi
fi
```

Loops in Shell Scripts

Bash supports:

1. for loop.

2. while loop.

Note that in each and every loop:

◦ First, the variable used in loop condition must be initialized, then execution of the loop begins.

◦ A test (condition) is made at the beginning of each iteration.

◦ The body of loop ends with a statement that modifies the value of the test (condition) variable.

for Loop

Syntax:

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

Example

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

```
[user@host ~]$ for HOST in host1 host2 host3; do echo $HOST; done
host1
host2
host3
[user@host ~]$ for HOST in host{1,2,3}; do echo $HOST; done
host1
host2
host3
[user@host ~]$ for HOST in host{1..3}; do echo $HOST; done
host1
host2
host3
[user@host ~]$ for FILE in file*; do ls $FILE; done
filea
fileb
filec
[user@host ~]$ for FILE in file{a..c}; do ls $FILE; done
filea
fileb
filec
```

for Loop

Syntax:

for ((expr1; expr2; expr3))

do

      repeat all statements between

      do and done until expr2 is TRUE

Done

Example

```
for (( i = 0 ; i <= 5; i++ ))
do
        echo "Welcome $i times"
done
```

Nesting of for Loop

$ vi nestedfor.sh

```
for (( i = 1; i <= 5; i++ ))
do
        for (( j = 1 ; j <= 5; j++ ))
do
echo -n "$i "
done
```

while loop

Syntax:

while [condition]

do

> *command1 command2*
>
> *command3 .. ....*

done

Example
i=1
while [$i -le 10]
do
echo "$n * $i = `expr $i \* $n`"
i=`expr $i + 1`
done

The case Statement

Syntax:

case $variable-name in

pattern1) command…..;;

pattern2) command…..;;

pattern N) command….;;

.

*) command ;;

esac

Example
read var
case $var in
1) echo "One";;
2) echo "Two";;
3) echo "Three";;
4) echo "Four";;
*) echo "Sorry, it is bigger than
Four";;
esac

Array

- Shell supports a different type of variable called an **array variable**. This can hold multiple values at the same time.

- **Defining Array Values**

  array—name[index]=value

Examples

Name[0]="ABC"

Name[1]="PQR"

Name[2]="XYZ"

Name[3]="GOD"

- **Accessing Array Values**

- **${array—name[index]}**

- You can access all the items in an array in one of the following ways –

- **${array-name[*]}**

- **${array-name[@]}**

Array Script

- echo "enter name"
- read -a names

- echo "names :${names[0]}, ${names[1]}

Select command

- Constructs simple menu from word list
- Allows user to enter a number instead of a word
- User enters sequence number corresponding to the word

Syntax:

```
select WORD in LIST
do
        RESPECTIVE-COMMANDS
done
```

Loops until end of input, i.e. ^d (or ^c)

Select example

```
#! /bin/bash
select var in alpha beta gamma
do
     echo $var
done
```
Prints:
```
       1) alpha
       2) beta
       3) gamma
     #? 2
     beta
     #? 4
     #? 1
     alpha
```

The until Loop

Purpose:

To execute commands in "command-list" as long as "expression" evaluates to false

<u>Syntax:</u>

```
until [ expression ]
do
        command-list
done
```

Example: Using the until Loop

```bash
#!/bin/bash


COUNTER=20
until [ $COUNTER -lt 10 ]
do
    echo $COUNTER
    let COUNTER-=1
done
```
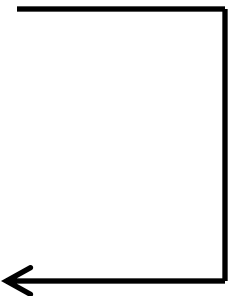
break and continue

The break statement

- transfer control to the statement AFTER the done statement
- terminate execution of the loop

The continue statement

- transfer control to the statement TO the done statement
- skip the test statements for the current iteration
- continues execution of the loop

The break command

```
while [ condition ]
do
        cmd-1
        break
        cmd-n
done
echo "done"
```

This iteration is over and there are no more iterations

The continue command

```
while [ condition ]
do
        cmd-1
        continue
        cmd-n
done
echo "done"
```

This iteration is over; do the next iteration

Example:

```
for index in 1 2 3 4 5 6 7 8 9 10
do
        if [ $index -le 3 ]; then
                echo "continue"
                continue
        fi
        echo $index
        if  [ $index -ge 8 ]; then
                echo "break"
                break
        fi
done
```

Grep: matches a pattern throughout a file.

- Grep <string> <filename>
- Grep <string> <filename> <filename>
- Options
- -i: ignore case
- -v: invert match.To select non-matching lines.
- -c: counting no. of matches

Basic Regular expressions

Symbol	Descriptions
.	matches a single character
^	matches start of string
$	matches end of string
*	matches up zero or more times the preceding character
\	Represent special characters
()	Groups regular expressions
?	Matches up exactly one character

[a1v]---any one character out of a or 1 or v

[^a1v]---any character except a, 1 and v

[[:alnum:]]---Alphanumeric [a-z A-Z 0-9]

[[:alpha:]]---Alphabetic [a-z A-Z]

[[:blank:]]---Blank characters (spaces or tabs)

[[:digit:]]---Numbers [0-9]

[[:lower:]]---Lowercase letters [a-z]

[[:upper:]]---Uppercase letters [A-Z]

[[:xdigit:]]---Hex digits [0-9 a-f A-F]