Unit-5 "Bash Shell Programming"

- Introduction to Shell :
  Bourne, C, Korn and Bash Shell with their difference
- Shell & Kernel
- Introduction to Vi Editors
  - Types of Mode
  - Vi Commands
  - Features of Vi
  - Disadvantages of Vi
- Shell Script :
  - How to write shell script?
  - Shell Keywords
  - Shell Comments
- Variables in shell
  - System Variable & User define variable
  - Variable naming rules
- To print or access values in shell, echo command & read statement.
- Shell arithmetic, commands used with script
- Operators used in Shell Script
- Structured language construct: if, else, else – if, case statement
- Looping Statements in shell
- Arrays
- Command line argument.

Introduction to SHELL (Bourne, C, Korn, Bash Shell)

- Shell contributes an important role in unix/linux Operating System. Shell is a program by which commands given by user are identified and executed then after.
- In other words we can say that, shell is a command interpreter which recognizes and executes the commands which are given on the command prompt by users.
- Shell sets three things which include standard input, standard output and standard error. Now-a-days, different shells are available as mentioned below.

❖ Bourne Shell

- Bourne shell developed by Dr. Steven Bourne at AT & T Bell Lab.
  - **Feature of Shell**

    i. Use of wild cards
    ii. Input and output variables for customizing the shell environment
    iii. A set of shell variables for customizing the shell environment
    iv. Background execution of commands
    v. Command set, loop and conditional statement support.

❖ C Shell

- C shell developed by Bill Joy at University of California.
  - **Feature of Shell**

    i. Use of wild cards
    ii. Input and output redirection operations
    iii. A set of shell variables for customizing the shell environment
    iv. Integer arithmetic
    v. Alias for used commands
    vi. History mechanism.

❖ Korn Shell

- Korn shell developed by David Korn at AT & T Lab.
  - **Feature of Shell**

    i. Alias for used commands
    ii. History mechanism.
    iii. Pattern matching for filename
    iv. Interactive editing facility using editor
    v. Home directory can be represented by a tilde sign.(~)

❖ Bash Shell

- Bash shell developed by Brian Fox & Chet Ramey. It is released of GNU. This shell used by Linux Operating System.
    - **Feature of Shell**
      i. Alias for used commands
      ii. History mechanism.
      iii. Pattern matching for filename
      iv. Interactive editing facility using editor
      v. Home directory can be represented by a tilde sign(~).

Shell & Kernel

- Shell is a program which allows the user to access the computer system. Shell is an interface between the user and computer system.
- Kernel is the only way through which the programs (all programs including shell) can access the hardware. It's a layer between the application programs and hardware. It is the core of most of the operating systems and manages everything including the communication between the hardware and software.
- KERNEL is the core part of operating system. It contains modules like device modules and other modules etc. Kernel is written in C language. Basically kernel is mediator between hardware and Operating System. But SHELL is an interface between users and operating system. Both are mediator but work is totally different.
- For example, User give a command to Shell through input device like keyboard and see that command on video Device like monitor, but in actual concept is user give a command to shell, then this shell transfer that command to kernel when kernel have module of that command then it transfer to hardware like CDROM.
- After that hardware behave as the module in kernel and then kernel again transfer the output to Shell. And finally shell transfer that output to user.

Introduction to Vi Editor

- The vi editor (short for visual editor) is a screen editor which is available on almost all Unix systems. Once you have learned vi, you will find that is is a fast and powerful editor. Vi has no menus but instead uses combinations of keystrokes in order to accomplish commands.
- It also remains one of the most widely used editor in Linux.
- Keyboard-based editors use a keyboard for two different operations: to specify editing commands and to receive character input. Used for editing commands, certain keys perform deletions, some execute changes, and others perform cursor movement. Used for character input, keys represent characters that can be entered into the file being edited.
- Usually, these two different functions are divided among different keys on the keyboard. Alphabetic keys are reserved for character input, while function keys and control keys specify editing commands.

Starting Vi Editor:

To start using vi, at the Linux prompt type vi followed by a file name. If you wish to edit an existing file, type in its name, if you are creating a new file type in the name you wish to give to the new file.

```
$vi filename
```

Then press Enter. You will see a screen similar to the one below which shows blank lines with tildes and the name and status of the file.

Example:

```
$vi myfile <enter>
~
~
~
"myfile"  [New file]
```

- Vi editor has three separate modes of operation for the keyboard: command mode, input mode and execution mode.

✶ Types of mode :

Command mode:

- It is the mode where you will be giving commands to the vi editor.
- When you invoke vi, this is the mode where you will be placed initially.

Input (insert) mode:

- In this mode, user can add text to the file.
- It is also called editing mode.
- When you want to insert at the current cursor position, give command i in the command mode.
- You will now get placed in the input mode and you can perform adding text.
- No command can be given while you are in the input mode.
- To move back to the command mode press "ESC" key.

Ex mode:

- It enables you to give commands at the command line.
- To move to the ex mode, press ":" (colon) if you are in command mode.
- If you are in the input mode. First press "ESC" key and then press ":" to move to the ex mode.

Basically, 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, press the **Esc** key. If you are not sure where you are, press **Esc** a couple of times and that should put you back in command mode.

General Command Information:

- Vi uses letters as commands. It is important to note that in general vi commands. Vi commands are case sensitive (make a difference between lower case and upper case) and those are not displayed on the screen when you type them. Generally they do not require a return after you type the command.

✶ Vi Editor Commands

❖ Closing and Saving Files:

When you edit a file in vi, you are actually editing a copy of the file rather than the original. The following selections describe methods you might use when closing a file, quitting vi, or both.

Quitting and Saving a file

- The commands ZZ (in uppercase) will allow you to quit vi and save the edits made to a file. You will then return to Linux prompt. Note that, you can also use the following commands.

Command	Function
:w	To save your file but not quit vi.
:q	To quit if you have not any edits.
:wq	To quit and save edits. (basically same as ZZ)

Quitting without Saving edits

- To erase all edits made to the file and either start over or quit. To do so, you can choose from the following two commands.

Command	Function
:e!	Reads the original file back in so that you can start over.
:q!	Wipes out all edits and allows you to exit from vi.

❖ Cursor Movement

You must be in command mode if you wish to move the cursor to another position in your file. If you have just finished typing text, you are still in insert mode and will nedd to press Esc to return to the command mode.

Command	Function
h	move left (backspace)
j	move down
k	move up
l	move right (spacebar)
w	Moves the cursor forward one word.
b	Moves the cursor backward one word.
e	Moves to the end of word.

❖ Shortcuts

Two short cuts for moving quickly on a line include the **$** and **0** (zero) keys. The $ key will move you to the end of a line, while the 0 will move you quickly to the beginning of a line.

❖ Screen Movements

To move the cursor to a line within your current screen uses the following keys.

Command	Function
H	move cursor to the top line of the screen
M	move cursor to the middle line of the screen
L	move cursor to the bottom line of the screen

To scroll through the file and see other screen use:

Command	Function
Ctrl-f	Scrolls down one screen
Ctrl-b	Scrolls up one screen
Ctrl-u	Scrolls up a half a screen
Ctrl-d	Scrolls down a half a screen

❖ Inserting

Command	Function
r	replace character under cursor with next character typed
R	keep replacing character until [esc] is hit
i	insert before cursor
a	append after cursor
A	append at end of line
O	open line above cursor and enter append mode

❖ Deleting

To delete a character, first place your cursor on that character. Then you may use any of the following command.

Command	Function
x	To delete character under cursor
X	To delete the character to the left of your cursor.
Dd	To delete all the current line.
dw	To delete from the character selected to the end of the word.
D	To delete from the current character to the end of the line.

❖ Misc Commands

Command	Function
Crtl G	Gives the line number of current position in the buffer and modification status of the line
.(DOT)	Repeats the action performed by the last command.
u	Undoes the effect of the last command.
U	Restores all changes to the current line since you moved the cursor to the line.
J	Joins the line immediately below, the current line with the current line.
~	Changes character at current cursor position from upper case to lower and vice-versa
:sh	Temporarily returns to the shell to perform some shell commands. Type exit to return Vi.
Crtl L	Clears and redraws the current window.

❖ Copying Text with Yank

If you wish to make a duplicate copy of existing text, you may use the Yank and put commands to accomplish this function. **Yank** copies the selected text into a buffer and holds it until another yank or deletion occurs. **Yank** is usually used in combination with a word or line object such as the ones shown below.

Command	Function
Yw	Copies a word into a buffer (7yw copies 7 words)
yy	Copies a line into a buffer (3yy will copy 3 lines)

Cutting (Yanking) & Pasting Text :

Command	Function
yw	Yanks current word
yy	Yanks current line
y$	Yanks to the end of the line
Y	Yanks the current line
P	Pasting from the cursor position

❖ Command for Quit Vi

Command	Function
ZZ	Writes the buffer to the file and quits vi.
:wq	Writes the buffer to the file and quits vi.
:w filename and :q	Writes the buffer to the filename and quits vi
:w! filename and :q	Overwrites existing the filename with the content of the buffer and quits vi.
:q!	Quits vi whether or not changes made to the buffer since the last write command
:q	Quits vi if changes made to the buffer were written to a file.

❖ General Startup

To use vi: vi filename
To exit vi and save changes: ZZ or :wq
To exit vi without saving changes: :q!
To enter vi command mode: [esc]

✶ Vi Features

- It is present in almost every Linux Unix system, even the most minimal.
- It is very small. In fact, some versions have a total code size of less than 100KB. This makes it easy to include vi on even the tiniest versions of Linux, such as those in embedded systems and those that run from a single floppy disk.

- It is typist-friendly, at least once you get used to it. For example, the commands are very short, usually just a few keystrokes. And because vi does not use the mouse, there is never any need to remove one's hands from the keyboard. This can speed up editing substantially.
- It is very powerful, as just a few very short commands can make broad changes to large documents. In fact, vi is more powerful than most of its users realize, and few of them know more than just fraction of all the commands.

✶ Disadvantages of Vi

- The guy who wrote vi didn't believe in help, so there isn't any online help available in vi.
- The user is always kept guessing. There are no self explanatory error messages.
- If anything goes wrong no error message appear, only speaker beeps to inform you that something went wrong.
- There are three modes. Under each mode the same key press creates different effects.
- Vi is obsessively case-sensitive. You have to differ the command with upper and lower case.

Shell Script

- 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.
- **"Shell script is series of commands written in plain text file. Shell script is just like batch file in MS-DOS but have more power than the MS-DOS batch file."**

- Normally shells are interactive. It means shell accept command from you (via keyboard) and execute them. But if you use command one by one (sequence of N numbers of commands), then you can store this sequence of commands to text file and tell the shell to execute this text file instead of entering the commands. This is known as Shell Script.

✶ How to write shell script?

Following steps are required to write shell script:

1) Use any editor like vi, gedit to write shell script.
2) Open terminal type **$vi <script name>.sh** Then Press Insert Key to transfer from command to insert Mode.
3) Write your script
4) To store your script press **ESC** key and write **: wq**<Enter>
5) Now execute your script like this **$sh <script name>.sh**

→ ***Shell script demo to print your name on screen.***

```
$ vi myscript
#
# My first shell script
#
clear
echo  "Chinkal Trivedi "
```

[Press Esc key the :wq & Execute script by the command : $sh myscript.sh]

✶ Shell Keywords

- Keywords are the words whose meaning has already been explained to the shell. The keywords cannot be used as variable names because of it is a reserved words with containing reserved meaning.

List of shell keywords:

echo	**read**	**set**	**unset**	**readonly**	**shift**
export	**if**	**fi**	**else**	**while**	**do**
done	**for**	**until**	**case**	**esac**	**break**
continue	**exit**	**return**	**trap**	**wait**	**eval**
exec	**ulimit**	**umask**			

✶ Shell Comments

You can put your comments in your script by # symbol.

Example :

```
# This is a statement not to execute.
```

✶ Shell Variables

Variables are an important part of any program or script. A variable is an element to refer a block of data in memory that can be modified. A Linux or Unix variable can be assigned any type of value, such as text, string or number. In Linux (Shell), there are two types of variables. (1) System/environment variable (2) User defined variable.

1) **System Variable/ Environment variable**

An environment / system variable is a variable that is available to any child process of shell. Usually, these variables are defined and needed by the programs that it runs. It is created and maintained by the Linux itself. This type of variable defined in CAPITAL LETTERS. Some of the system variables are shown below.

System Variable	**Meaning**
PWD (/home/students)	Current working directory
USERNAME (student)	Username who is currently login to this PC
SHELL (/bin/bash)	Shell name
BASH (/bin/bash)	Shell name
BASH_VERSION (1.14.7)	Shell version name
COLUMNS (80)	No. of columns for screen
LOGNAME (students)	Students logging name

You can print the value of system variable in two ways :

$ echo $USERNAME

OR

$ echo USERNAME

2) User defined variable

This type of variables are created and maintained by the user. And can be defined in lower case letter.

Syntax : **variable_name=value**

'value' is assigned to variable name and must be on right side = sign.

Example : **$no=10**

$name=Chinkal

Rules for naming variable name:

(1) Variable name must begin with alphanumeric character or underscore character. E.g name, std, col_name
(2) Don't put spaces on either side of equal to sign when assigning the value to the variable. For example $no=10 is valid, where $no =10 or $no= 10 is invalid.
(3) Variables are case sensitive in Linux.
(4) You can define NULL variable as follows.
$name=""
(5) Do not use?, * to name your variable.

echo command

- echo is one of the most commonly and widely used built-in command for Linux bash and C shells, that typically used in scripting language and batch files to display a line of text/string on standard output or a file.

Option	Use
-n	Do not print the trailing new line.
-e	Enable interpretation of backslash escapes.
\b	Back space
\n	New line
\t	Horizontal tab
\v	Vertical tab
\r	Carriage return

Examples:

1. To display standard output.
echo "Chinkal Trivedi" OR echo Chinkal Trivedi

2. The '**-e**' option in Linux acts as interpretation of escaped characters that are backslashed. Using option '**\b**' – backspace with backslash interpretor '**-e**' which removes all the spaces in between.

echo –e "Hello \bStudentsOf \bTYBCA "
Output : HelloStudentsOfTYBCA

3. Using option '**\n**' – New line with backspace interpretor '**-e**' treats new line from where it is used.
echo –e "Hello Students\n We are learning \n Linux."
Output :
```
Hello Students
We are learning
Linux.
```

4. Using option '**\t**' – horizontal tab with backspace interpretor '**-e**' to have horizontal tab spaces.
echo –e "Hello Students.. \t We are learning \t Linux. "
Output :
```
Hello Students..        We are learning      Linux.
```

5. To use option new Line '**\n**' and horizontal tab '**\t**' simultaneously.
echo –e "\n\t Hello Students \n\t We are learning \n\t Linux."
Output :
```
             Hello Students
             We are learning
             Linux.
```

6. Using option '**\v**' – vertical tab with backspace interpretor '**-e**' to have vertical tab spaces.
echo –e "\vHello Students \vWe are learning \vLinux."
Output :
```
       Hello Students
                  We are learning
                              Linux.
```

7. To use option new Line '**\n**' and vertical tab '**\v**' simultaneously.
echo –e "\n\v Hello Students \n\v We are learning \n\v Linux.
Output:

```
Hello Students

We are learning

Linux.
```

read command

- Read command or statement is used to input/get the value to store in variable from the keyboard.

Syntax :

```
read variablename1, variablename2, …… variablenameN
```

Example :

```
read name
```

- This will create no string type variable and input value of number by keyboard.

#Write a shell script to enter user's complete name and display on terminal.

```
echo "Input First Name : "
read fname
#read –p "Input First Name" fname
echo "Input Last Name : "
read lname
#read –p "Input First Name" fname
echo "Hello I am $fname $lname"
```

Shell Arithmetic

- Shell arithmetic us used to perform arithmetic operations with shell programming. For doing this, 'expr' keyword is used to perform the arithmetic like Addition, Subtraction, Multiplication, Division, Module etc.

Syntax :

expr Operator1 Math Operator Operator2

Syntax :

```
echo "Addition : " `expr 6 + 4`

echo "Subtraction : `expr 10 – 5` "

echo "Multiplication : `expr 10 \* 5 ` "
```

Commands used with shell script

- Any Linux command used in terminal can be used with shell script also.

```
#Write a shell script to print today's date.

        echo  "Today is : `date` "
```

- To run more than one command with one command line:

```
#Write a shell script to print today's date with calender.

        echo  "Today is : `date;cal` "1
```

```
#Write a shell script to print today's date with users who are currently
logged on.
        echo  "Today's date and logged in members : `date;who` "
```

Operators used in Shell Script

- Linux shell programming has the following operators: they compare or evaluate mathematical, logical and relational expressions.

→ Arithmetic Operators:

Operator	Meaning
+	Used to perform Addition
-	Used to perform Subtraction
*	Used to perform Multiplication
/	Used to perform Division
%	Used to perform Return Reminder
^	Used to for power value

→ Relational/Comparison Operators:

Operator	Meaning
-eq	Is equal to (==)
-ne	Is not equal to (!=)
-lt	Is less than (<)
-le	Is less than or equal to (<=)
-gt	Is greater than (>)
-gt	Is greater than or equal to (>=)

→ Logical Operators: [To combine two or more condition at a time]

Operator	Meaning	Use
!	Logical NOT	!expression
-a	Logical AND	expression1 –a expression2
-o	Logical OR	expression1 -o expression2

→ String Operators:

Operator	Meaning
=	To check the value of two operands are equal or not. If yes, then condition will be true. (String1=String2)
!=	To check the value of two operands are equal or not. If they are not equal, then condition will be true. (String1!=String2)
-z	To check if the given string operand size is zero. If it is zero length, then it returns true. (-z string1)
-n	To check if the given string operand size is non-zero. If it is non-zero length, then it returns true. (-n string1)
str	To check if str is not empty string. If it is an empty string, then it returns false.
	String is not null or not defined. (string1)

Evaluate argument as an "expr"

Syntax : **expr expression**

Expression	**Operator**	**Examples**
Addition	**+**	$ expr 10 + 5
Subtraction	**-**	$ expr 10 - 5
Multiplication	**\***	$ expr 10 \* 5
Division	**/**	$ expr 10 / 5
Modulation	**%**	$ expr 10 % 5
Substring of length	**expr <string> <position> <length>**	$ expr "Hello Friends" 7 6 (Output will be "Friend"

Decision Making Statements in Shell Script / Conditional Statements

There are the following kinds of statements available in shell programming as decision making statements.

- ✶ **simple if**
- ✶ **if… else**
- ✶ **if…elif…else**
- ✶ **nested if**
- ✶ **case statement**

→ *Simple if*

- Simple if is used for decision making in shell script. If the given condition is true then it will execute the set of code that you have allocated to that block.

Simple if – Syntax

```
if  [ condition ]
then
        Execute the statements
        "true" condition.
fi
```

```
if  test  <condition>
then
        Execute the statements
        "true" condition.
fi
```

Simple if – Example

```
#To find bigger no.
echo "Input No1 : "
read n1
echo "Input No2 : "
read n2

if  [ $n1 -gt   $n2 ]
then
        echo "no1 is bigger.."
fi
```

```
#To find bigger no.
echo "Input No1 : "
read n1
echo "Input No2 : "
read n2

if  test $n1 -gt   $n2
then
        echo "no1 is bigger.."
fi
```

Note : Either the test command or just an expression in square brackets [ ] can be used to evaluate whether an expression is true (zero) or false (non zero).

→ *If...else*

- If… else is used for decision making in shell script where the given condition is true then it will execute the set of code that you have allocated to that block otherwise you can execute the rest of code for the false condition.

if…else – Syntax

```
if  [ condition ]
then
        Execute the statements
        "true" condition.
else
        Execute the statements
        "false" condition.
fi
```
	```
	if  test  <condition>
	then
	        Execute the statements
	        "true" condition.
	else
	        Execute the statements
	        "false" condition.
	fi
	```

If…else – Example

```
#To find bigger no.
echo "Input No1 : "
read n1
echo "Input No2 : "
read n2

if  [ $n1 –gt   $n2 ]
then
        echo "no1 is bigger.."
else
        echo "no2 is bigger.."
fi
```

```
#To find bigger no.
echo "Input No1 : "
read n1
echo "Input No2 : "
read n2

if  test $n1 –gt   $n2
then
        echo "no1 is bigger.."
else
        echo "no2 is bigger.."
fi
```

→ *If…elif…else*

- It is possible to create compound conditional statements by using one or more else if (elif) clause. If the 1st condition is false, then subsequent elif statements are checked. When an elif condition is found to be true, the statements following that associated parts are executed.

if…elif..else – Syntax

```
if  [ condition1 ]
then
        Execute the statements
        For "true" condition1.
elif  [ condition2 ]
then
        Execute the statements
        For "true" condition2.
elif  [ conditionN ]
then
        Execute the statements
        For "true" conditionN.

else
        Execute the statements
        For "false" condition.
fi
```

```
if  test <condition1>
then
        Execute the statements
        For "true" condition1.
elif  test  <condition2>
then
        Execute the statements
        For "true" condition2.
elif  test <conditionN>
then
        Execute the statements
        For "true" conditionN.

else
        Execute the statements
        For "false" condition.
fi
```

If…elif..else – Example

```
#To find bigger no.
echo "Input No1 : "      read n1
echo "Input No2 : "      read n2
if  [ $n1 –gt   $n2 ]
then
        echo "no1 is bigger.."
elif [ $n1 –lt $n2 ]
then
        echo "no2 is bigger.."
else
        echo "Both are same.."
fi
```

```
#To find bigger no.
echo "Input No1 : "      read n1
echo "Input No2 : "      read n2
if  test  $n1 –gt   $n2
then
        echo "no1 is bigger.."
elif  test  $n1 –lt $n2
then
        echo "no2 is bigger.."
else
        echo "Both are same.."
fi
```

→ *Nested if*

- If statement and else statement can be nested in bash shell programming. The keyword “fi” indicates the end of the inner if statement and all if statement should end with “fi”.

Nested if – Example

```
#To verify the username & password.
echo “Input Username : “
read Uname
echo “Input Password : “
read Pass
if [ $Uname = “Chinkal” ]
then
    if [$Pass = “Trivedi” ]
    then
        echo “Login Correctly..”
    else
        echo “Invalid Password..”
     fi
else
    echo “Invalid Username..”
fi
```

```
#To verify the username & password.
echo “Input Username : “
read Uname
echo “Input Password : “
read Pass
if  test  $Uname = “Chinkal”
then
    if  test  $Pass = “Trivedi”
    then
        echo “Login Correctly..”
    else
        echo “Invalid Password..”
     fi
else
    echo “Invalid Username..”
fi
```

→ *case…esac statement*

- The case statement is good alternative to Multilevel if-then-else-fi statement. It enables you to match several values against one variable. Its easier to read and write multiple if conditions.

case…esac – Syntax	case…esac – Example1

```
case $variable in
value1)
        Statements executed for
          matching value1
         ; ;
value2)
        Statements executed for
          matching value2
         ; ;
value3)
        Statements executed for
          matching value3
         ; ;
valueN)
        Statements executed for
          matching valueN
         ; ;
* )
      Statements executed for
      not matching above values
        ; ;
esac
```
	```
	#To check character, number or special
	symbol.

	read –p "Input Single Value : " ch
	case $ch in
	[A-Z] ) echo "It is a Character."
	         ; ;
	[0-9] ) echo "It is a Number."
	         ; ;
	* ) echo "It is a special symbol. "
	        ; ;
	esac
	```

Looping statements in shell programming

- For shell script, the particular instruction can be repeated again and again until particular condition is satisfied. A group of instruction that is executed repeatedly is called "Loop". Bash shell programming provides the following kinds of loop.

→ for loop

There are two methods to use for loop in shell programming.

❖ Bash for loop using "in".

for loop Syntax	for loop Example1

for Variable in List
do
    Command1
    Command2
    ……
    CommandN
done
	#To print number as per list of values
	i=1
	for i in 1 2 3 4 5
	do
	    echo $i
	done

- Here, for, in, do, done are the keywords. "List" indicates the list of values.
- The list can be a variable's value that contains several words separated by spaces. Variable name can be any bash variable.

for loop Example2	for loop Example3

#To print number in given range
no=1
for no in {1..10}
do
    echo "Number : " $i
done
	#To print Weekday
	i=1
	for day in Mon Tue Wed
	do
	    echo "Weekday : $((i++)) $day "
	done
	
	Output : Weekday :1 Mon
	            Weekday :2 Tue

❖ Bash for loop using c programming syntax:

for loop Syntax

for ((expr1; expr2; expr3))
do
    Repeat all statements between do and done until 'expr2' will be true.
done

for loop Example2	for loop Example3

#To print 1 to 10
for ((i=1; i<=10; i++))
do
        echo $i
done
	#To print Factorial of given number
	read –p "Number " Num
	fact=1
	for ((i=$num;i>=1;i--))
	do
	   fact=`expr $fact \* $i`
	done
	echo "Factorial : $fact"

→*while loop*

- The while loop enables you to execute set of commands repeatedly until some condition occurs. It is usually used when you need to manipulate the value of a variable repeatedly.

While loop syntax	While loop example

while [condition]
do
     Command
     Command
     ....
     [break/continue]
done
	#To print 1 to 10
	i=1
	while [$i –le 10]
	do
	    echo $i
	    i=`expr $i +1`
	done

→until loop

- The until loop is useful when you need to execute a set of commands until a condition is true.

until loop syntax	until loop example

until [condition]
do
    Command
    Command
    ….
    [break/continue]
done
	#To print until get 10
	i=1
	while [! $i –lt 10]
	do
	   echo $i
	   i=\`expr $i +1\`
	done

Array

- A shell variable is capable enough to hold single or multiple values. Shell supports a different type of variable called "Array variable" to store multiple values at the same time with giving a single name to variable.

Declare an Array:

An array can be explicitly declared by the ***declare*** shell-built in.

declare -a ARRAYNAME

Defining array values:

Syntax : Array_Name [Index] = Value

Here, Array name is the name of array variable, Index is the item number to set and value is the item that you want to set for array.

Accessing array values:

Syntax : **${Array_Name [Index]}**

Example:

```
Arr[0]="Hello"
Arr[1]="Chinkal"
Arr[2]="Trivedi"
echo ${Arr[1]}
Output : Chinkal
```

Note : "*" is used as index number to print the rest of the elements from the array.

```
Arr[0]="Hello"
Arr[1]="Chinkal"
Arr[2]="Trivedi"
echo ${Arr[0]}
echo ${Arr[*]}
Output :   Hello
           Chinkal
           Trivedi
```

Array Example To print array elements according to size of array

```
read –p "Input Array Size : " N

for ((i=0;i<$N;i++))
do
        read arr[$i]
done

for ((i=0;i<$N;i++))
do
        echo ${arr[$i]}
done
```

Try Yourself: Write a shell script to input key value from the user and check your entered key value is found in your array or not.

Script to find the maximum elements from the array.

```
read -p "Input array size : " no
for ((i=0;i<$no;i++))
do
        read arr[$i]
done

max=0
for((i=0;i<$no;i++))
do
      if test ${arr[$i]}  -gt  $max
        then
                max=${arr[$i]}
        fi
done

echo "Max : " $max
```

Script to print array elements in reverse order

```
read -p "Input Size : " size
for ((i=0;i<size;i++))
do
        read arr[$i]
done

echo "Reverse array"
echo
j=`expr $size - 1`
for ((i=0;i<size;i++))
do
      echo ${arr[$j]}
        j=`expr $j - 1`
done
```

❖ Two Dimensional Array:

We can easily represent a 2-dimensional matrix using a 1-dimensional array. We can represent M*N array for M number of rows and N number of columns. For doing this, we have to declare arrays with "declare –A arr1 arr2 arr3" like this.

Read array elements

```
declare -A arr

for((i=0;i<2;i++))
do
        for((j=0;j<2;j++))
        do
                read arr[$i,$j]
        done
done
```

Access array elements

```
for((i=0;i<2;i++))
do
      for((j=0;j<2;j++))
      do
            echo -n "   " ${arr[$i,$j]}
      done
        echo -e
done
```

Command Line Argument

- To input arguments into bash shell programming is like any normal command line program, where you can give the arguments while you are executing script. This can be done with special variables set aside for storing the arguments.
- The arguments are stored in variables with a number in the order of argument starting at 1 as $0 variable is the name of your script.
- The total numbers of arguments is stored in **$#** and to view arguments passed to the script you can use **'$@'**.

Your Script Name : $0

First Argument : $1

Second Argument : $2

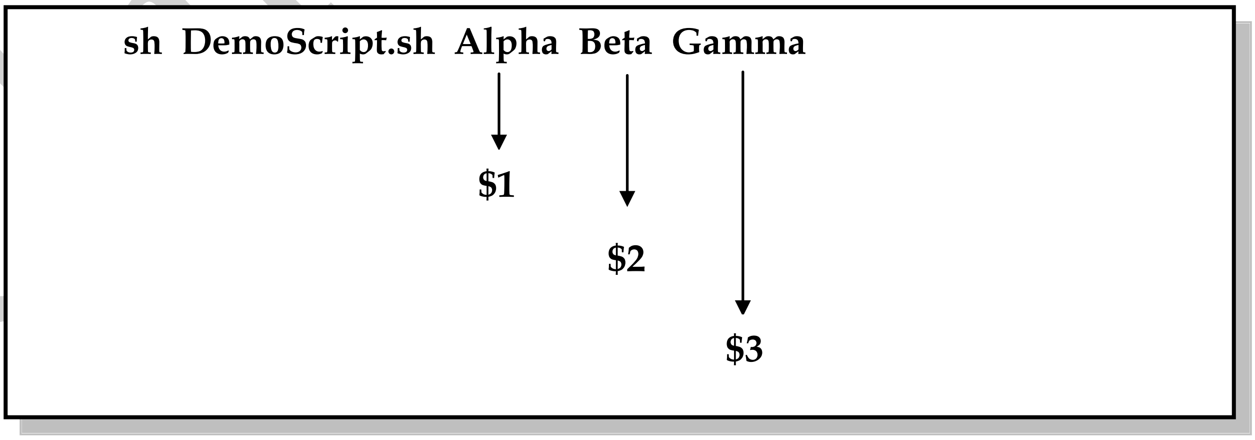
Third Argument : $3

……

Arguments passed to the script: $@

Total number of arguments: $#

- For example, you want to give three arguments to your script then, passed them while executing the script.


sh DemoScript.sh Alpha Beta Gamma
$1
$2
$3