

## Unit-3

### 1. What is Shells?

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

### Shell Prompt

- The prompt, \$, which is called the **command prompt**, is issued by the shell. While the prompt is displayed, you can type a command.
- Shell reads your input after you press **Enter**. It determines the command you want executed by looking at the first word of your input. Spaces and tabs separate words.

### Shell Scripts

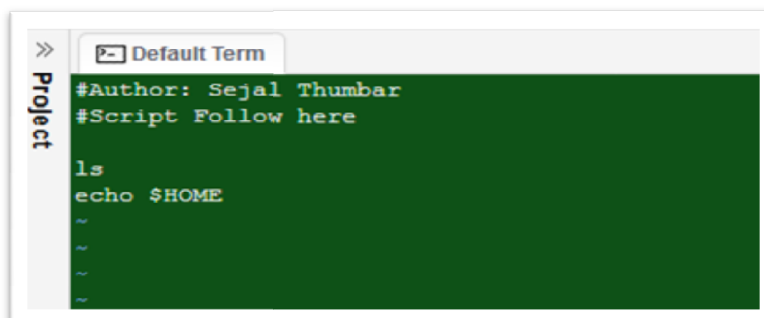
- The basic concept of a shell script is a list of commands, which are listed in the order of execution. A good shell script will have comments.
- There are **conditional tests**, such as value A is greater than value B, loops allowing us to go through massive amounts of data, files to read and store data, and variables to read and store data, and the script may include functions.
- Shell **scripts and functions** are both **interpreted**. This means they are not compiled.

**Example Script:** - Assume we create a test.sh script.

```
echo $HOME
WHOAMI
~
~
~
~
```

### Shell Comments

You can put your comments in your script as follows –



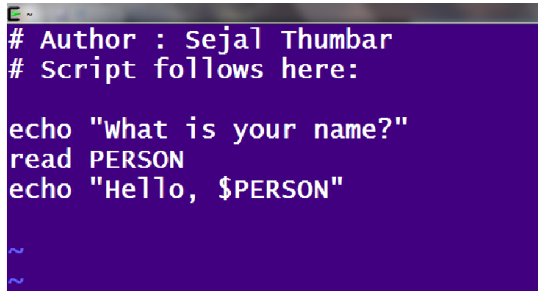
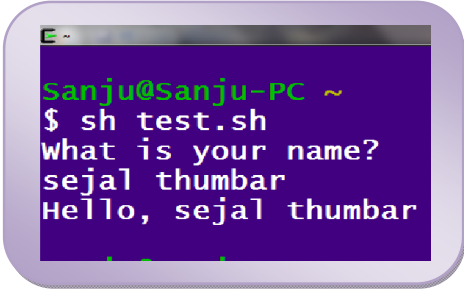
```
>> Default Term
Project
#Author: Sejal Thumbar
#Script Follow here

ls
echo $HOME
~
~
~
~
```

## Unit-3

### Extended Shell Scripts

- Shell scripts have several required constructs that tell the shell environment what to do and when to do it. Of course, most scripts are more complex than the above one.
- The following script uses the **read** command which takes the input from the keyboard and assigns it as the value of the variable **PERSON** and finally prints it on STDOUT.

vi test.sh	OUTPUT
 <pre># Author : Sejal Thumbar # Script follows here:  echo "What is your name?" read PERSON echo "Hello, \$PERSON"  ~ ~</pre>	 <pre>sanju@sanju-PC ~ \$ sh test.sh What is your name? sejal thumbar Hello, sejal thumbar</pre>

## 2. Shell Variables

- "A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data."
- A variable is nothing more than a pointer to the actual data. The shell enables you to create, assign, and delete variables.

### Variable Names

- The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (\_).
- By convention, Unix shell variables will have their names in UPPERCASE.
- The reason you cannot use other characters such as !, \*, or - is that these characters have a special meaning for the shell.

### Defining Variables

Variables are defined as follows –

variable\_name=variable\_value

For example :- **NAME="Sejal Thumbar"**

### Accessing Values

To access the value stored in a variable, prefix its name with the dollar sign (\$)

**NAME="Sejal Thumbar"**

**echo \$NAME**

## Unit-3

### Read-only Variables

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

**syntax:-** `readonly variable_name`

#### vi readonly

```
NAME="Sejal Thumbar"
readonly NAME
NAME="Kajal"
~
~
~
```

#### output

```
Sanju@Sanju-PC ~
$ sh readonly
readonly: line 3: NAME: readonly variable
```

### Unsetting Variables

- Unsetting or deleting a variable directs the shell to remove the variable from the list of variables that it tracks.
- Once you unset a variable, you cannot access the stored value in the variable.
- You cannot use the unset command to **unset** variables that are marked **readonly**.

**syntax:-** `unset variable_name`

### Variable Types

When a shell is running, **three main types** of variables are present

- Local Variables** – A local variable is a variable that is **present within the current instance of the shell**. It is not available to programs that are started by the shell. They are set at the command prompt.
- Environment Variables** – An environment variable is **available to any child process** of the shell. Some programs need environment variables in order to function correctly. Usually, a shell script defines only those environment variables that are needed by the programs that it runs.
- Shell Variables** – A shell variable is a **special variable that is set by the shell** and is required by the shell in order to function correctly. **Some of these variables are environment variables** whereas others are local variables.

### 3. System Variable:

- "An important Unix concept is the **environment**, which is defined by environment variables. Some are set by the system, others by you, yet others by the shell, or any program that loads another program."
- A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data.
- For example:- **\$ TEST="Unix Programming"**  
**\$ echo \$TEST**

## Unit-3

- Note that the environment variables are set without using the \$ sign but while accessing them we use the \$ sign as prefix. These variables retain their values until we come out of the shell.

### Common environment variables in Unix

#### 1. PS1

It is used for Primary prompt string like \$ or % if you want to change your prompt string then write:

**PS1="\u@\h\w"** OR **export PS1="\u@\h\w"**

Sr.No.	Escape Sequence & Description
1	<b>\t</b> :- Current time, expressed as HH:MM:SS
2	<b>\d</b> :- Current date, expressed as Weekday Month Date
3	<b>\n</b> :- Newline
4	<b>\s</b> :- Current shell environment
5	<b>\W</b> :- Working directory
6	<b>\w</b> :- Full path of the working directory
7	<b>\u</b> :- Current user's username
8	<b>\h</b> :- Hostname of the current machine
9	<b>\#</b> :- Command number of the current command. Increases when a new command is entered
10	<b>\\$</b> :- If the effective UID is 0 (that is, if you are logged in as root), end the prompt with the # character; otherwise, use the \$ sign

#### 2. PS2

PS2 is Secondary prompt, it is used when the command entered on the first line was not able to completed, **the default prompt is > .** If you want to change your prompt string then write:

**"PS2="Con =>"** OR **export PS2="con ->"**

#### 3. PATH

Gives the default path for command. **echo \$PATH**

#### 4. HOME

Indicates the home directory of the current user: **echo \$HOME**

#### 5. LOGNAME

It gives login name which is stored in passwd file. This variable shows your user name only while **who** and **whoami** display all the features.

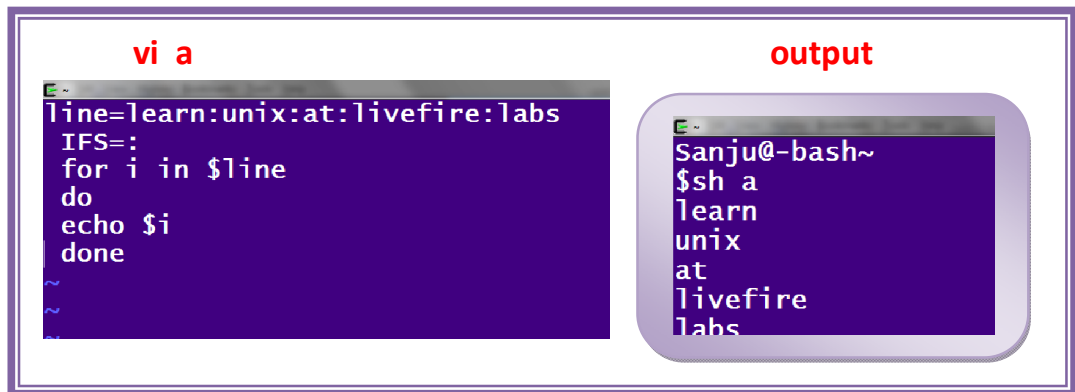
#### 6. MAIL

It displays absolute path name of user's mailbox and determine where the incoming mail to the addressed user it is to be stored.

#### 7. IFS

This variable contains internal field separator . The shell uses the value stored in IFS, which is the space, tab, and newline characters by default, to delimit words for the read and set commands, when parsing output from command substitution, and when performing variable substitution.

## Unit-3



8. SHELL  
This variable contains the pathname of current the login shell.
9. TERM  
It indicates the terminal types.
10. MAILCHECK  
IT contains interval between the tests for new email.

#### 4. User variables set and echo command with shell variables

##### SET Command

- **set** is a built-in function of the [Bourne shell \(sh\)](#), [C shell \(csh\)](#), and [Korn shell \(ksh\)](#), which is used to define and determine the values of the system [environment](#).
- Change the value of shell attributes and positional parameters, or display the names and values of shell variables.  
**Syntax:-** set var=value;

##### echo Command

- **echo** command in linux is used to display line of text/string that are passed as an argument . This is a built in command that is mostly used in shell scripts and batch files to output status text to the screen or a file.

**Syntax :** echo [option] [string]

##### Options

Tag	Description
-n	Do not output a trailing newline.
-e	Enable interpretation of backslash escape sequences (see below for a list of these).
\a	An alert (The BELL character).

## Unit-3

\b	Backspace
\c	Produce no further output after this.
\e	The escape character; equivalent to pressing the escape key.
\n	A newline.
\r	A carriage return
\t	A horizontal tab.
\v	A vertical tab.

### 5. Positional Parameters

"A positional parameter is a variable within a shell program; its value is set from an argument specified on the command line that invokes the program. Positional parameters are numbered and are referred to with a preceding ``\$'': \$1, \$2, \$3, and so on."

We understood how to be careful when we use certain non-alphanumeric characters in variable names. This is because those characters are used in the names of special Unix variables. These variables are reserved for specific functions.

**For example**, the \$ character represents the process ID number, or PID, of the current shell

```
$echo $$
```

```
The above command writes the PID of the current shell  
29949
```

The following table shows a number of special variables that you can use in your shell scripts –

Sr.No.	Variable & Description
1	<b>\$0</b>  <b>The filename</b> of the current script.
2	<b>\$n</b>  These variables correspond to the arguments with which a script was invoked. Here <b>n is a positive decimal number</b> corresponding to the position of an argument (the first argument is \$1, the second argument is \$2, and so on).
3	<b>\$#</b>  <b>The number of arguments</b> supplied to a script.
4	<b>\$*</b>  <b>All the arguments are double quoted.</b> If a script receives two arguments, \$* is equivalent to \$1 \$2.
5	<b>\$@</b>  <b>All the arguments are individually double quoted.</b> If a script receives two arguments, \$@ is equivalent to \$1 \$2.

## Unit-3

6	<code>\$?</code>  <b>The exit status of the last command executed.</b> As a rule, most commands return an exit status of 0 if they were successful, and 1 if they were unsuccessful.
7	<code>\$\$</code>  The process <b>number of the current shell</b> . For shell scripts, this is the process ID under which they are executing.
8	<code>\$!</code>  The process number of <b>the last background command</b> .

### Command-Line Arguments

The command-line arguments \$1, \$2, \$3, ...\$9 are positional parameters, with \$0 pointing to the actual command, program, shell script, or function and \$1, \$2, \$3, ...\$9 as the arguments to the command.

Vi test.sh	Output
<pre>echo "File Name: \$0" echo "First Parameter : \$1" echo "Second Parameter : \$2" echo "Quoted Values: \$@" echo "Quoted Values: \$*" echo "Total Number of Parameters : \$#"</pre>	<pre>Sanju@Sanju-PC ~ \$ sh test.sh Ram Sharma File Name: test.sh First Parameter : Ram Second Parameter : Sharma Quoted Values: Ram Sharma Quoted Values: Ram Sharma Total Number of Parameters : 2</pre>

### Special Parameters \$\* and \$@

There are special parameters that allow accessing all the command-line arguments at once. \$\* and \$@ both will act the same unless they are enclosed in double quotes, "".

Both the parameters specify the command-line arguments. However, the "\$\*" special parameter takes the entire list as one argument with spaces between and the "\$@" special parameter takes the entire list and separates it into separate arguments.

Vi test.sh	Output
<pre>for TOKEN in \$* do     echo \$TOKEN done ~ ~ ~</pre>	<pre>Sanju@Sanju-PC ~ \$ sh test.sh 1 2 3 1 2 3</pre>

## Unit-3

### 6. Shell Decision Making

- While writing a shell script, there may be a situation **when you need to adopt one path out of the given two paths.**
- So you need to make use of conditional statements that allow your program to **make correct decisions and perform the right actions.**
- Unix Shell supports conditional statements which are used to perform different actions based on different conditions. We will now understand two decision-making statements here –
  1. The **if...else** statement
  2. The **case...esac** statement

#### 1. The if...else statements

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

Unix Shell supports following forms of **if...else** statement –

- i. **if...fi** statement
- ii. **if...else...fi** statement
- iii. **if...elif...else...fi** statement

##### i. **if...fi** statement

- ❖ The **if...fi** statement is the fundamental control statement that allows Shell to **make decisions and execute statements conditionally.**

##### Syntax

```
if [ expression ]
then
    Statement(s) to be executed if expression is true
fi
```

- ❖ If the resulting value is *true*, given *statement(s)* are executed.
- ❖ If the **expression is false** then no statement would be executed. Most of the times, comparison operators are used for making decisions.
- ❖ It is recommended to **be careful with the spaces between braces and expression**. No space produces a syntax error.
- ❖ If **expression** is a shell command, then it will be assumed true if it returns **0** after execution. If it is a Boolean expression, then it would be true if it returns true.

```
a=10
b=20

if [ $a == $b ]
then
    echo "a is equal to b"
fi
```



## Unit-3

### ii. if...else...fi

- ❖ The **if...else...fi** statement is the next form of control statement that allows Shell to execute statements in a controlled way and make the right choice.

Syntax:-

```
if [ expression ]  
then  
    Statement(s) to be executed if expression is true  
else  
    Statement(s) to be executed if expression is not true  
fi
```

- ❖ The Shell *expression* is evaluated in the above syntax. If the resulting value is *true*, given *statement(s)* are executed. If the *expression* is *false*, then no statement will be executed.

```
a=10  
b=20  
  
if [ $a == $b ]  
then  
    echo "a is equal to b"  
else  
    echo "a is not equal to b"  
fi
```

### iii. if...elif...fi statement

- ❖ The **if...elif...fi** statement is the one level advance form of control statement that allows Shell to make correct decision out of several conditions.

Syntax

```
if [ expression 1 ]  
then  
    Statement(s) to be executed if expression 1 is true  
elif [ expression 2 ]  
then  
    Statement(s) to be executed if expression 2 is true  
else  
    Statement(s) to be executed if no expression is true  
fi
```

- ❖ This code is just a series of if statements, where each if is part of the else clause of the previous statement.
- ❖ Here statement(s) are executed based on the true condition, if none of the condition is true then else block is executed.

## Unit-3

```
a=10
b=20

if [ $a == $b ]
then
    echo "a is equal to b"
elif [ $a -gt $b ]
then
    echo "a is greater than b"
elif [ $a -lt $b ]
then
    echo "a is less than b"
else
    echo "None of the condition met"
fi
```

### 2. The case...esac Statement

- You can use **multiple if...elif statements to perform a multi-way branch**. However, this is not always the best solution, especially when all of the branches depend on the value of a single variable.
- Unix Shell supports case...esac statement which handles exactly this situation, and it does so more efficiently than repeated if...elif statements.
- The basic syntax of the **case...esac** statement is to give an expression to evaluate and to execute several different statements based on the value of the expression.
- The interpreter checks each case against the value of the expression until a match is found. If nothing matches, a default condition will be used.
- There is only one form of case...esac statement which has been described in detail here –

#### Syntax:

```
case Var in
    pattern1)
        Statement(s) to be executed if pattern1 matches ;;

    pattern2)
        Statement(s) to be executed if pattern2 matches ;;

    pattern3)
        Statement(s) to be executed if pattern3 matches ;;

    *)
        Default condition to be executed ;;
esac
```

## Unit-3

---

- Here the string word is compared against every pattern until a match is found. The statement(s) following the matching pattern executes.
- If no matches are found, the case statement exits without performing any action. There is no maximum number of patterns, but the minimum is one.
- When statement(s) part executes, the command ;; indicates that the program flow should jump to the end of the entire case statement. This is similar to break in the C programming language.

```
FRUIT="kiwi"
case "$FRUIT" in

    "apple") echo "Apple pie is quite tasty." ;;
    "banana") echo "I like banana nut bread." ;;
    "kiwi") echo "New Zealand is famous for kiwi." ;;

esac
```

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

### 7. test command

- The **test** command checks for various properties of files, strings and integers. It produces no output (except error messages) but returns the result of the test as the exit status
- Test is used in conditional execution. It is used for:
  1. File attributes comparisons
  2. Perform string comparisons.
  3. Basic arithmetic comparisons.
- **test** exits with the status determined by **EXPRESSION**. Placing the **EXPRESSION** between square brackets ([ and ]) is the same as testing the **EXPRESSION** with **test**.
- **Exit status :-** **0** for true, **1** for false. Anything greater than 1 indicates an error or malformed command.

#### **Syntax:**

test **EXPRESSION**

## Unit-3

```
Sanju@Sanju-PC~$:===test -h /dev/stdin; echo $?  
0  
Sanju@Sanju-PC~$:===|
```

### 8. Shell Basic Operators

- There are various operators supported by each shell. We will discuss in detail about Bourne shell (default shell) **We will now discuss the following operators** –
  - Arithmetic Operators
  - Relational Operators
  - Boolean Operators
  - String Operators
  - File Test Operators

The following **points need to be considered** while adding –

- There must be spaces between operators and expressions. For example, 2+2 is not correct; it should be written as 2 + 2.
- The complete expression should be enclosed between ‘```’, called the backtick.

#### 1. Arithmetic Operators

- The following arithmetic operators are supported by Bourne Shell.
- Assume variable **a** holds 10 and variable **b** holds 20 then –

Operator	Description	Example
<b>+</b> (Addition)	Adds values on either side of the operator	<code>`expr \$a + \$b`</code> will give 30
<b>-</b> (Subtraction)	Subtracts right hand operand from left hand operand	<code>`expr \$a - \$b`</code> will give -10
<b>*</b> (Multiplication)	Multiplies values on either side of the operator	<code>`expr \$a * \$b`</code> will give 200
<b>/</b> (Division)	Divides left hand operand by right hand operand	<code>`expr \$b / \$a`</code> will give 2
<b>%</b> (Modulus)	Divides left hand operand by right hand operand and returns remainder	<code>`expr \$b % \$a`</code> will give 0
<b>=</b> (Assignment)	Assigns right operand in left operand	<code>a = \$b</code> would assign value of b into a
<b>==</b> (Equality)	Compares two numbers, if both are same then returns true.	<code>[ \$a == \$b ]</code> would return false.
<b>!=</b> (Not Equality)	Compares two numbers, if both are different then returns true.	<code>[ \$a != \$b ]</code> would return true.

- It is very important to understand that all the conditional expressions should be inside square braces with spaces around them, for example `[ $a == $b ]` is correct whereas, `[ $a==$b ]` is incorrect.
- All the arithmetical calculations are done using long integers.

## Unit-3

```
a=10
b=20

val=`expr $a + $b`
echo "a + b : $val"

val=`expr $a - $b`
echo "a - b : $val"

val=`expr $a \* $b`
echo "a * b : $val"

val=`expr $b / $a`
echo "b / a : $val"

val=`expr $b % $a`
echo "b % a : $val"

if [ $a == $b ]
then
    echo "a is equal to b"
fi

if [ $a != $b ]
then
    echo "a is not equal to b"
fi
```

### OUTPUT

```
a + b : 30
a - b : -10
a * b : 200
b / a : 2
b % a : 0
a is not equal to b
```

## 2. Relational Operators

- Bourne Shell supports the **following relational operators that are specific to numeric values**. These operators do not work for string values unless their value is numeric.
- For example, following operators will work to check a relation between 10 and 20 as well as in between "10" and "20" but not in between "ten" and "twenty".
- Assume variable **a** holds 10 and variable **b** holds 20 then –

Operator	Description	Example
-eq	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[ \$a -eq \$b ] is not true.
-ne	Checks if the value of two operands are equal or not; if values are not equal, then the condition becomes true.	[ \$a -ne \$b ] is true.
-gt	Checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true.	[ \$a -gt \$b ] is not true.
-lt	Checks if the value of left operand is less than the value of right operand; if yes, then the condition becomes true.	[ \$a -lt \$b ] is true.
-ge	Checks if the value of left operand is greater than or equal to the value of right operand; if yes, then the condition becomes true.	[ \$a -ge \$b ] is not true.
-le	Checks if the value of left operand is less than or equal to the value of right operand; if yes, then the condition becomes true.	[ \$a -le \$b ] is true.

## Unit-3

- It is very important to understand that **all the conditional expressions should be placed inside square braces with spaces around them.**
- For example, [ \$a <= \$b ] is correct whereas, [\$a <= \$b] is incorrect.

```
a=10
b=20

if [ $a -eq $b ]
then
    echo "$a -eq $b : a is equal to b"
else
    echo "$a -eq $b: a is not equal to b"
fi

if [ $a -ne $b ]
then
    echo "$a -ne $b: a is not equal to b"
else
    echo "$a -ne $b : a is equal to b"
fi
```

### OUTPUT

```
10 -eq 20: a is not equal to b
10 -ne 20: a is not equal to b
```

### 3. Boolean Operators

- The following Boolean operators are supported by the Bourne Shell.
- Assume variable **a** holds 10 and variable **b** holds 20 then –

Operator	Description	Example
!	This is logical negation. This inverts a true condition into false and vice versa.	[ ! false ] is true.
-o	This is logical <b>OR</b> . If one of the operands is true, then the condition becomes true.	[ \$a -lt 20 -o \$b -gt 100 ] is true.
-a	This is logical <b>AND</b> . If both the operands are true, then the condition becomes true otherwise false.	[ \$a -lt 20 -a \$b -gt 100 ] is false.

```
a=10
b=20

if [ $a != $b ]
then
    echo "$a != $b : a is not equal to b"
else
    echo "$a != $b: a is equal to b"
fi

if [ $a -lt 100 -a $b -gt 15 ]
then
    echo "$a -lt 100 -a $b -gt 15 : returns true"
else
    echo "$a -lt 100 -a $b -gt 15 : returns false"
fi
```

### OUTPUT

```
10 != 20 : a is not equal to b
10 -lt 100 -a 20 -gt 15 : returns true
```

## Unit-3

### 4. String Operators

- The following string operators are supported by Bourne Shell.
- Assume variable **a** holds "abc" and variable **b** holds "efg" then –

Operator	Description	Example
=	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[ \$a = \$b ] is not true.
!=	Checks if the value of two operands are equal or not; if values are not equal then the condition becomes true.	[ \$a != \$b ] is true.
-z	Checks if the given string operand size is zero; if it is zero length, then it returns true.	[ -z \$a ] is not true.
-n	Checks if the given string operand size is non-zero; if it is nonzero length, then it returns true.	[ -n \$a ] is not false.
str	Checks if <b>str</b> is not the empty string; if it is empty, then it returns false.	[ \$a ] is not false.

```
a="abc"
b="efg"

if [ $a ]
then
    echo "$a : string is not empty"
else
    echo "$a : string is empty"
fi
```

#### OUTPUT

abc : string is not empty

### 5. File Test Operators

- We have a few operators that can be used to test various properties associated with a Unix file.
- Assume a variable **file** holds an existing file name "test" the size of which is 100 bytes and has **read**, **write** and **execute** permission on –

Operator	Description	Example
-b file	Checks if file is a block special file; if yes, then the condition becomes true.	[ -b \$file ] is false.
-c file	Checks if file is a character special file; if yes, then the condition becomes true.	[ -c \$file ] is false.
-d file	Checks if file is a directory; if yes, then the condition becomes true.	[ -d \$file ] is not true.
-f file	Checks if file is an ordinary file as opposed to a directory or special file; if yes, then the condition becomes true.	[ -f \$file ] is true.
-r file	Checks if file is readable; if yes, then the condition becomes true.	[ -r \$file ] is true.
-w file	Checks if file is writable; if yes, then the condition becomes true.	[ -w \$file ] is true.
-x file	Checks if file is executable; if yes, then the condition becomes true.	[ -x \$file ] is true.
-s file	Checks if file has size greater than 0; if yes, then condition becomes true.	[ -s \$file ] is true.
-e file	Checks if file exists; is true even if file is a directory but exists.	[ -e \$file ] is true.

## Unit-3

```
file="unix.txt"

if [ -e $file ]
then
    echo "File exists"
else
    echo "File does not exist"
fi
```

### OUTPUT

File exists

### 9. Looping statements

- A loop is a powerful programming tool that **enables you to execute a set of commands repeatedly**. In this chapter, we will examine the following types of loops available to shell programmers –
  - 1) [The while loop](#)
  - 2) [The for loop](#)
  - 3) [The until loop](#)
- You will use different loops based on the situation. For example, the **while** loop executes the given commands until the given condition remains true; the **until** loop executes until a given condition becomes true.

#### 1) The while Loop

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

#### Syntax

```
while command
do
    Statement(s) to be executed if command is true
done
```

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

```
a=0

while [ $a -lt 5 ]
do
    echo $a
    a=`expr $a + 1`
done
```

### OUTPUT

0  
1  
2  
3  
4

- Each time this loop executes, the variable **a** is checked to see whether it has a value that is less than 10. If the value of **a** is less than 10, this test condition has an exit status of 0. In this case, the current value of **a** is displayed and later **a** is incremented by 1.



## Unit-3

### 2) The for loop

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

#### Syntax

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

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

```
for var in 0 1 2 3 4 5 6 7
do
    echo $var
done
```

#### Output

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

### 3) The until Loop

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

#### Syntax

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

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

```
a=0
until [ ! $a -lt 5 ]
do
    echo $a
    a=`expr $a + 1`
done
```

#### OUTPUT

```
0
1
2
3
4
5
```

## Unit-3

### Shell Loop Control

We will learn following two statements that are used to control shell loops:

- The **break** statement
- The **continue** statement

#### 1. The break Statement

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

##### Syntax

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

**break**

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

**break n**

Here **n** specifies the **n<sup>th</sup>** enclosing loop to the exit from.

```
a=0
while [ $a -lt 10 ]
do
    echo $a
    if [ $a -eq 5 ]
    then
        break
    fi
    a=`expr $a + 1`
done
```

##### Output

```
0
1
2
3
4
5
```

Here is a simple example of nested for loop. This script breaks out of both loops if **var1** equals 2 and **var2** equals 0 –

```
for var1 in 1 2 3
do
    for var2 in 0 5
    do
        if [ $var1 -eq 2 -a $var2 -eq 0 ]
        then
            break 2
        else
            echo "$var1 $var2"
        fi
    done
done
```

##### Output

```
1 0
1 5
```

## Unit-3

### The continue statement

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

**Syntax:-** `continue`

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

**continue n**

Here **n** specifies the **n<sup>th</sup>** enclosing loop to continue from.

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

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

#### OUTPUT

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