

# NETWORKING & SYSTEM ADMINISTRATION LAB

## SHELL SCRIPTING



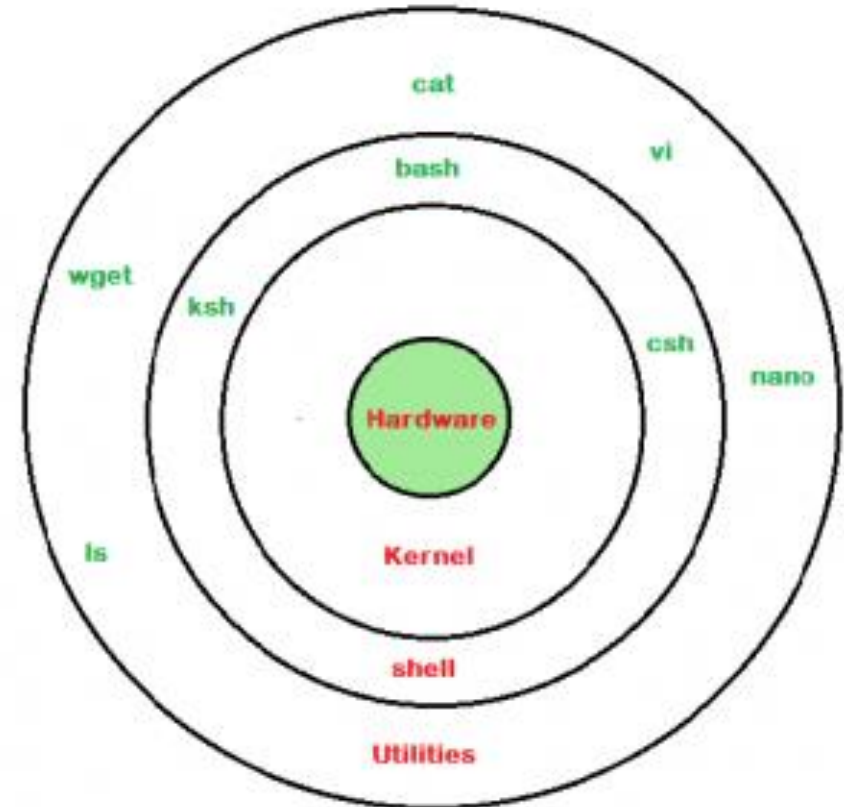
Syllabus: Shell scripting: study bash syntax, environment variables, variables, control constructs such as if, for and while, aliases and functions, accessing command line arguments passed to shell scripts. Study of startup scripts, login and logout scripts, familiarity with systemd and system 5 init scripts is expected

## Introduction to Linux Shell and Shell Scripting

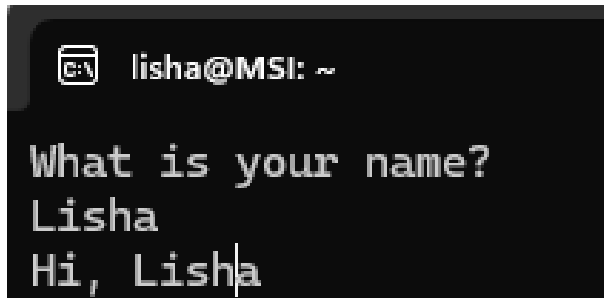
- If you are using any major operating system you are indirectly interacting to **shell**.
- If you are running Ubuntu, Linux Mint or any other Linux distribution, you are interacting to shell every time you use terminal.
- The **kernel** is a computer program that is the core of a computer's operating system, with complete control over everything in the system. It manages following resources of the Linux system –
  - File management
  - Process management
  - I/O management
  - Memory management
  - Device management etc.

# Shell

- A shell is special user program which provide an interface to user to use operating system services.
- Shell accept human readable commands from user and convert them into something which kernel can understand.
- It is a command language interpreter that execute commands read from input devices such as keyboards or from files. The shell gets started when the user logs in or start the terminal.
- Shell is broadly classified into two categories
  - Command Line Shell
  - Graphical shell



- **Command Line Shell** : Shell can be accessed by user using a command line interface. A special program called Terminal in linux/macOS or Command Prompt in Windows OS is provided to type in the human readable commands such as “cat”, “ls” etc. and then it is being execute. The result is then displayed on the terminal to the user.
- **Graphical Shells** : Graphical shells provide means for manipulating programs based on graphical user interface (GUI), by allowing for operations such as opening, closing, moving and resizing windows, as well as switching focus between windows. Window OS or Ubuntu OS can be considered as good example which provide GUI to user for interacting with program.



```
lisha@MSI: ~  
What is your name?  
Lisha  
Hi, Lisha
```



## Shell Types

**Bourne shell** – If you are using a Bourne-type shell, the \$ character is the default prompt.

**C shell** – If you are using a C-type shell, the % character is the default prompt.

The Bourne Shell has the following subcategories –

- Bourne shell (sh)
- Korn shell (ksh)
- Bourne Again shell (bash)
- POSIX shell (sh)

The different C-type shells follow –

- C shell (csh)
- TENEX/TOPS C shell (tcsh)

## Shell Scripting

- As shell can also take commands as input from file we can write these commands in a file and can execute them in shell to avoid this repetitive work. These files are called **Shell Scripts** or **Shell Programs**.
- Each shell script is saved with **.sh** file extension eg. **myscript.sh**
- A shell script comprises following elements –
  - Shell Keywords – if, else, break etc.
  - Shell commands – cd, ls, echo, pwd, touch etc.
  - Functions
  - Control flow – if..then..else, case and shell loops etc.

**Example 1:** Shell script to display your name.

```
#!/bin/bash  
echo "wht is your name?"  
read person  
echo "hello, $person"
```

**Example 2:** Shell script to display date.

**Example 3:** Shell script to display date, pwd, ls commands.



## 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.
- The name of a variable can contain only letters (a to z or A to Z), numbers ( 0 to 9) or the underscore character ( \_).

```
_ALI  
TOKEN_A  
VAR_1  
VAR_2
```

valid variable names

```
2_VAR  
-VARIABLE  
VAR1-VAR2  
VAR_A!
```

invalid variable names

- The reason you cannot use other characters such as !, \*, or - is that these characters have a special meaning for the shell.
  - **Syntax:**  
**variable\_name=variable\_value**

## Variable Types

When a shell is running, three main types of variable

**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.

## Special Variables

- The `$` character represents the process ID number, or PID, of the current shell

```
Lisha@MSI:~$ echo $$  
1194
```

- `$0` - The filename of the current script.
- `$n` - These variables correspond to the arguments with which a script was invoked. Here `n` is a positive decimal number corresponding to the position of an argument (the first argument is `$1`, the second argument is `$2`, and so on).
- `$#` - The number of arguments supplied to a script.
- `$*` - All the arguments are double quoted. If a script receives two arguments, `$*` is equivalent to `$1 $2`.
- `$@` - All the arguments are individually double quoted. If a script receives two arguments, `$@` is equivalent to `$1 $2`.
- `$?` - The exit status of the last command executed.
- `$$` - The process number of the current shell. For shell scripts, this is the process ID under which they are executing.
- `#!` - The process number of the last background command.

## 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.

### Example 4: Shell script to demonstrate variable

## Example 5: Shell script to count lines and words in a file

```
#!/bin/bash
file_path="/home/meera/test.sh"
countlines=`wc --lines < $file_path`
countwords=`wc --word < $file_path`
echo "Number of lines: $countlines"
echo "Number of words: $countwords"
```

```
Number of lines: 4
Number of words: 11
```

## Shell Arrays

- Shell supports a different type of variable called an **array variable**. This can hold multiple values at the same time.
- Arrays provide a method of grouping a set of variables.
- Instead of creating a new name for each variable that is required, you can use a single array variable that stores all the other variables.

```
NAME1="Anu"  
NAME2="Binu"  
NAME3="Cinu"  
NAME4="Dinu"  
NAME5="Linu"
```

**Scalar Variable**

```
NAME[0]="Anu"  
NAME[1]="Binu"  
NAME[2]="Cinu"  
NAME[3]="Dinu"  
NAME[4]="Linu"
```

**Array Variable**

**Syntax: array\_name[index]=value**

**Accessing Array Values**

**`${array_name[index]}`**

## Example 6: Shell Script to display array index

```
lisha@MSI: ~  
NAME[0]="Anu"  
NAME[1]="Binu"  
NAME[2]="Dinu"  
NAME[3]="Linu"  
NAME[4]="Ginu"  
echo "First Method: ${NAME[*]}"  
echo "Second Method: ${NAME[@]}"
```

```
lisha@MSI:~$ ./test.sh  
First Method: Anu Binu Dinu Linu  
Second Method: Anu Binu Dinu Linu
```

## Basic Operators in Shell Scripting

- 5 basic operators in bash/shell scripting:
  - Arithmetic Operators
  - Relational Operators
  - Boolean Operators
  - Bitwise Operators
  - File Test Operators

**1. Arithmetic Operators:** These operators are used to perform normal arithmetics/mathematical operations. There are 7 arithmetic operators:

- **Addition (+):** Binary operation used to add two operands.
- **Subtraction (-):** Binary operation used to subtract two operands.
- **Multiplication (\*):** Binary operation used to multiply two operands.
- **Division (/):** Binary operation used to divide two operands.
- **Modulus (%):** Binary operation used to find remainder of two operands.
- **Increment Operator (++):** Unary operator used to increase the value of operand by one.
- **Decrement Operator (--):** Unary operator used to decrease the value of a operand by one



Assume variable **a** holds 10 and variable **b** holds 20 then

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

### Example 7: Shell Script to add two numbers

Shell script reads two numbers as command line parameters and performs the addition operation.

```
lisha@MSI:~$ vi sum.sh
lisha@MSI:~$ ./sum.sh
-bash: ./sum.sh: Permission denied
lisha@MSI:~$ chmod +x sum.sh
lisha@MSI:~$ ./sum.sh
Total value : 4
```

Write a shell script to initialize two numeric variables. Then perform an addition operation on both values and store results in the third variable.

Shell script, which takes input from the user at run time. Then calculate the sum of given numbers and store to a variable and show the results.

## Example 8: Shell Script to demonstrate arithmetic operators

```
#!/bin/bash
read -p "Enter a: " a
read -p "Enter b: " b
add=$(( a + b ))
echo "Addition of a and b are $add"
sub=$(( a - b ))
echo "Subtraction of a and b are $sub"
mul=$(( a * b ))
echo "Multiplication of a and b are $mul"
div=$(( a / b ))
echo "Division of a and b are $div"
mod=$(( a % b ))
echo "Modulus of a and b are $mod"
if [ $a == $b ]
then
    echo "a is equal to b"
fi
if [ $a != $b ]
then
    echo "a is not equal to b"
fi
(( ++a ))
echo "Increment operator on a $a"
(( --b ))
echo "Decrement operator on b $b"
```

```
Enter a: 13
Enter b: 10
Addition of a and b are 23
Subtraction of a and b are 3
Multiplication of a and b are 130
Division of a and b are 1
Modulus of a and b are 3
a is not equal to b
Increment operator on a 14
Decrement operator on b 9
meera@STMeera:~$ ./arithmetic.sh
Enter a: 13
Enter b: 13
Addition of a and b are 26
Subtraction of a and b are 0
Multiplication of a and b are 169
Division of a and b are 1
Modulus of a and b are 0
a is equal to b
Increment operator on a 14
Decrement operator on b 12
```

**2. Relational Operators:** Relational operators are those operators which define the relation between two operands. They give either true or false depending upon the relation. They are of 6 types:

- **'==' Operator:** Double equal to operator compares the two operands. It returns true if they are equal otherwise returns false.
- **'!=' Operator:** Not Equal to operator returns true if the two operands are not equal otherwise it returns false.
- **'<' Operator:** Less than operator returns true if first operand is less than second operand otherwise returns false.
- **'<=' Operator:** Less than or equal to operator returns true if first operand is less than or equal to second operand otherwise returns false.
- **'>' Operator:** Greater than operator returns true if the first operand is greater than the second operand otherwise returns false.
- **'>=' Operator:** Greater than or equal to operator returns true if first operand is greater than or equal to second operand otherwise returns false.

## Example 9: Shell Script to demonstrate relational operators

```
#!/bin/bash
read -p "Enter a " a
read -p "Enter b " b
if(( $a==$b ))
then
    echo "a is equal to b"
else
    echo "a is not equal to b"
fi
if(( $a!= $b ))
then
    echo "a is not equal to b"
else
    echo "a is equal to b"
fi
if(( $a<$b ))
then
    echo "a is less than b"
else
    echo "a is not less than b"
fi
if(( $a<=$b ))
then
    echo "a is less than or equal to b"
else
    echo "a is not less than or equal to b"
fi
if(( $a>$b ))
then
    echo "a is greater than b"
else
    echo "a is not greater than b"
fi
if(( $a>=$b ))
then
    echo "a is greater than or equal to b"
else
    echo "a is not greater than or equal to b"
fi
```

```
Enter a 12
Enter b 10
a is not equal to b
a is not equal to b
a is not less than b
a is not less than or equal to b
a is greater than b
a is greater than or equal to b
```

```
#!/bin/bash
a=12
b=10

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

if [ $a -gt $b ]
then
    echo "$a -gt $b: a is greater than b"
else
    echo "$a -gt $b: a is not greater than b"
fi

if [ $a -lt $b ]
then
    echo "$a -lt $b: a is less than b"
else
    echo "$a -lt $b: a is not less than b"
fi

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

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

```
12 -eq 10: a is not equal to b
12 -ne 10: a is not equal to b
12 -gt 10: a is greater than b
12 -lt 10: a is not less than b
12 -ge 10: a is greater or equal to b
12 -le 10: a is not less or equal to b
```

**3. Logical Operators** : They are also known as **boolean operators**. These are used to perform logical operations. They are of 3 types:

- **Logical AND (&&)**: This is a binary operator, which returns true if both the operands are true otherwise returns false.
- **Logical OR (||)**: This is a binary operator, which returns true if either of the operand is true or both the operands are true and return false if none of them is false.
- **Not Equal to (!)**: This is a unary operator which returns true if the operand is false and returns false if the operand is true.

## Example 10: Shell Script to demonstrate logical operators

```
#!/bin/bash

read -p 'Enter a : ' a
read -p 'Enter b : ' b

if(( $a == "true" & $b == "true" ))
then
    echo Both are true.
else
    echo Both are not true.
fi

if(( $a == "true" || $b == "true" ))
then
    echo Atleast one of them is true.
else
    echo None of them is true.
fi

if(( ! $a == "true" ))
then
    echo "a" was initially false.
else
    echo "a" was initially true.
fi
```

```
Enter a : True
Enter b : True
Both are true.
Atleast one of them is true.
a was initially true.
```



## Example 11: Shell Script to check a number is even or odd

```
#!/bin/bash
echo "shell script to check a number is even or odd"
read -p "Enter a number: " number
if(( $number % 2 == 0 ))
then
    echo "$number is even"
else
    echo "$number is Odd"
fi
```

```
shell script to check a number is even or odd
Enter a number: 57
57 is Odd
```

## Example 12: Shell script to check whether a number is positive or negative

```
#!/bin/bash
echo "Check a number is positive or negative"
read -p "Enter a number " num
if [ $num -lt 0 ]
then
    echo "$num is Negative"
elif [ $num -gt 0 ]
then
    echo "$num is Positive"
else
    echo "Neither Positive Nor Negative"
fi
~
```

```
Check a number is positive or negative
Enter a number -12
-12 is Negative
```

```
Check a number is positive or negative
Enter a number 12
12 is Positive
```

```
Check a number is positive or negative
Enter a number 0
Neither Positive Nor Negative
```

**Example 13: Shell script for finding greatest of two numbers**

**Example 14: Shell script to find the greatest of three numbers**

## String Operators

Assume variable **a** holds "abc" and variable **b** holds "efg" then

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

## Example 15: Shell Script to demonstrate String Operators

```
abc = efg: a is not equal to b
abc != efg : a is not equal to b
-z abc : string length is not zero
-n abc : string length is not zero
abc : string is not empty
```

```
#!/bin/bash
a="abc"
b="efg"

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

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

if [ -z $a ]
then
    echo "-z $a : string length is zero"
else
    echo "-z $a : string length is not zero"
fi

if [ -n $a ]
then
    echo "-n $a : string length is not zero"
else
    echo "-n $a : string length is zero"
fi

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

# Bitwise Operators

- A bitwise operator is an operator used to perform bitwise operations on bit patterns. They are of 6 types:
- **Bitwise And (&):** Bitwise & operator performs binary AND operation bit by bit on the operands.
- **Bitwise OR (|):** Bitwise | operator performs binary OR operation bit by bit on the operands.
- **Bitwise XOR (^):** Bitwise ^ operator performs binary XOR operation bit by bit on the operands.
- **Bitwise complement (~):** Bitwise ~ operator performs binary NOT operation bit by bit on the operand.
- **Left Shift (<<):** This operator shifts the bits of the left operand to left by number of times specified by right operand.
- **Right Shift (>>):** This operator shifts the bits of the left operand to right by number of times specified by right operand.

## Example 16: Shell Script to demonstrate Bitwise Operators

```
Enter a : 10
Enter b : 8
Bitwise AND of a and b is 8
Bitwise OR of a and b is 10
Bitwise XOR of a and b is 2
Bitwise Compliment of a is -11
Left Shift of a is 20
Right Shift of b is 4
```

```
#!/bin/bash
read -p 'Enter a : ' a
read -p 'Enter b : ' b
bitwiseAND=$(( a&b ))
echo Bitwise AND of a and b is $bitwiseAND
bitwiseOR=$(( a|b ))
echo Bitwise OR of a and b is $bitwiseOR
bitwiseXOR=$(( a^b ))
echo Bitwise XOR of a and b is $bitwiseXOR
bitiwiseComplement=$(( ~a ))
echo Bitwise Compliment of a is $bitiwiseComplement
leftshift=$(( a<<1 ))
echo Left Shift of a is $leftshift
rightshift=$(( b>>1 ))
echo Right Shift of b is $rightshift
```

**File Test Operator:** These operators are used to test a particular property of a file.

- **-b operator:** This operator check whether a file is a block special file or not. It returns true if the file is a block special file otherwise false.
- **-c operator:** This operator checks whether a file is a character special file or not. It returns true if it is a character special file otherwise false.
- **-d operator:** This operator checks if the given directory exists or not. If it exists then operators returns true otherwise false.
- **-e operator:** This operator checks whether the given file exists or not. If it exists this operator returns true otherwise false.
- **-r operator:** This operator checks whether the given file has read access or not. If it has read access then it returns true otherwise false.
- **-w operator:** This operator check whether the given file has write access or not. If it has write then it returns true otherwise false.
- **-x operator:** This operator check whether the given file has execute access or not. If it has execute access then it returns true otherwise false.
- **-s operator:** This operator checks the size of the given file. If the size of given file is greater than 0 then it returns true otherwise it is false.



## Example 17: Shell Script to demonstrate File Test Operators

```
Enter file name : Stringoperators.sh
File Exist
The given file is not empty.
The given file has read access.
The given file has write access.
The given file has execute access.
```

```
#!/bin/bash
read -p 'Enter file name : ' FileName
if [ -e $FileName ]
then
    echo File Exist
else
    echo File doesnot exist
fi
if [ -s $FileName ]
then
    echo The given file is not empty.
else
    echo The given file is empty.
fi
if [ -r $FileName ]
then
    echo The given file has read access.
else
    echo The given file does not has read access.
fi
if [ -w $FileName ]
then
    echo The given file has write access.
else
    echo The given file does not has write access.
fi
if [ -x $FileName ]
then
    echo The given file has execute access.
else
    echo The given file does not has execute access.
fi
```

# Shell Decision Making/Conditional Statements

1. **if statement**
2. **if-else statement**
3. **if..elif..else..fi statement (Else If ladder)**
4. **if..then..else..if..then..fi..fi..(Nested if)**
5. **switch statement**

## if statement

This block will process if specified condition is true.

### Syntax

```
if [ expression ]  
then  
    statement  
fi
```

### Example 18: Shell Script to check two numbers are equal using if statement

```
#!/bin/bash  
a=10  
b=20  
if [ $a == $b ]  
then  
    echo "a is equal to b"  
fi  
if [ $a != $b ]  
then  
    echo "a is not equal to b"  
fi
```

```
a is not equal to b
```

## if-else statement

If specified condition is not true in if part then else part will be execute.

### *Syntax*

```
if [ expression ]  
then  
    statement1  
else  
    statement2  
fi
```

**Example 19: Shell Script to check two numbers are equal using if else statement**

```
#!/bin/bash  
a=20  
b=20  
if [ $a == $b ]  
then  
    echo "a is equal to b"  
else  
    echo "a is not equal to b"  
fi
```

```
a is equal to b
```

## if..elif..else..fi statement (Else If ladder)

To use multiple conditions in one if-else block, then elif keyword is used in shell. If expression1 is true then it executes statement 1 and 2, and this process continues. If none of the condition is true then it processes else part.

### Syntax

```
if [ expression1 ]
then
    statement1
    statement2
    .
    .
elif [ expression2 ]
then
    statement3
    statement4
    .
    .
else
    statement5
fi
```

### Example 20: Shell Script to check the range of a numbers using else if ladder

```
#!/bin/bash
read -p 'Enter number: ' num
if (( $num>=0 && $num<=10 ))
then
    echo "Number is in between 0 and 10"
elif (( $num>=11 && $num <=20 ))
then
    echo "Number is in between 11 and 20"
elif (( $num>=21 && $num <=30 ))
then
    echo "Number is in between 21 and 30"
else
    echo " Number is greater 30"
fi
```

```
Enter number: 5
Number is in between 0 and 10
```

## if..then..else..if..then..fi..fi..(Nested if)

Nested if-else block can be used when, one condition is satisfies then it again checks another condition. In the syntax, if expression1 is false then it processes else part, and again expression2 will be check.

**Syntax:**

```
if [ expression1 ]
then
    statement1
    statement2
.
else
    if [ expression2 ]
    then
        statement3
        .
    fi
fi
```

**Example 21: Shell Script to analyze people of certain age groups who are eligible for getting a suitable job if their condition and norms get satisfied using nested if statement.**

```
#!/bin/bash
read -p 'Enter age: ' age
if (( $age<18 ))
then
    echo "Belongs to Minor,Not eligible for job"
else
    if (( $age>=18 && $age <=50 ))
    then
        echo "Eligible for job, Please fill details and apply for job"
    else
        echo "Age above 50, Not eligible as per organization norms"
    fi
fi
```

```
Enter age: 12
Belongs to Minor,Not eligible for job
```