Mail server

What is a mail server?

A mail server -- also known as a mail transfer agent, or MTA; mail transport agent; mail router; or internet mailer -- is an application that receives incoming email from local users and remote senders and forwards outgoing messages for delivery. A computer dedicated to running these applications is also called a mail server

A mail server works with other programs to create a messaging system. A messaging system includes all the applications necessary to keep email moving smoothly. When an email is sent, a program, such as Microsoft Outlook, forwards the message to a mail server. The mail server then forwards the message to either another mail server or to a holding area on the same server to be forwarded later.

What are the types of mail servers?

Mail servers can be divided into two categories:

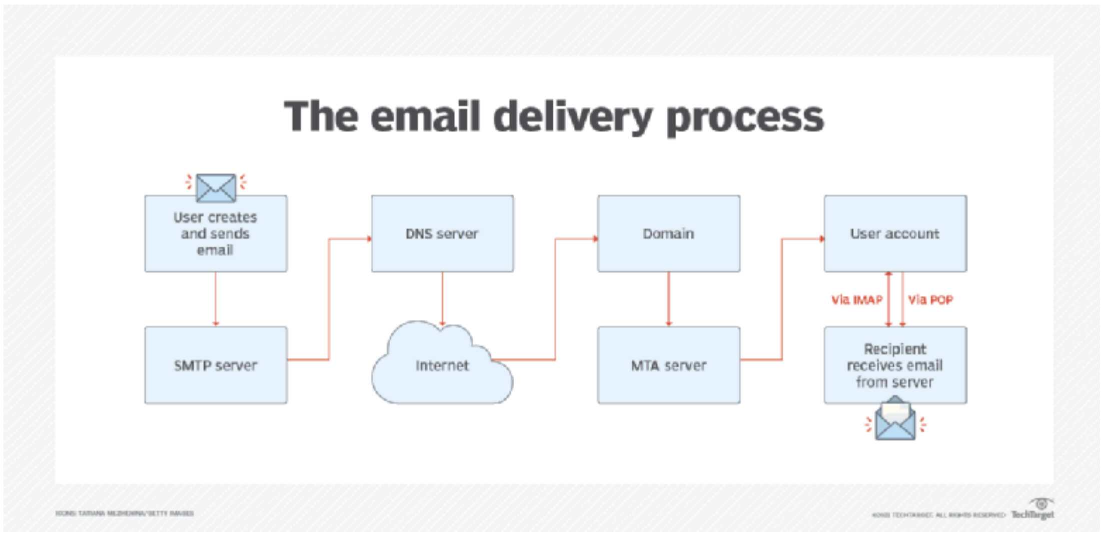
- incoming mail servers
- outgoing mail servers

An incoming mail server stores mail and sends it to a user's inbox. Post Office Protocol 3 (POP3) and Internet Message Access Protocol (IMAP) are the two main types of incoming mail servers.

POP3, for example, downloads email from a server and stores incoming email messages on a single device until the user opens the email client. Once the user downloads the email, it is automatically deleted from the server, unless the "keep mail on server" setting is enabled. Many internet service providers offer their users POP3 email accounts, as they are more space efficient.

IMAP servers enable users to preview, delete and organize emails before transferring them to multiple devices from the email server. Copies of emails are left on the server until the user deletes them.

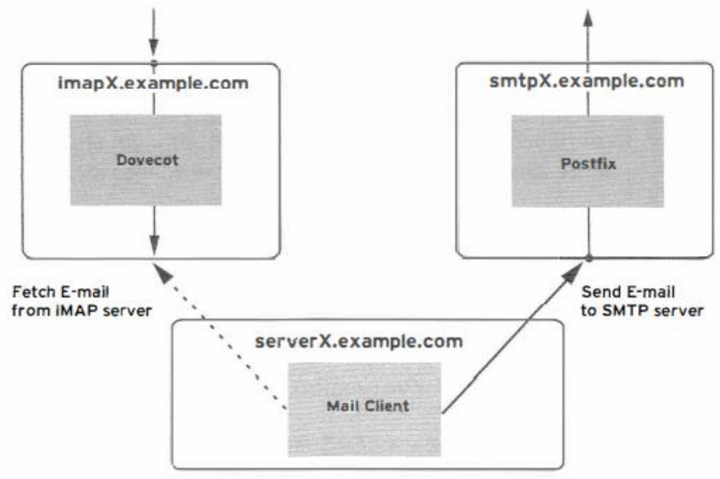
An Outgoing mail server operates by having a user's machine communicate with Simple Mail Transfer Protocol (SMTP), which handles the email delivery process. SMTP servers work with other types of mail servers, namely POP3 or IMAP, to send emails from email clients.


The email delivery process
User creates and sends email
SMTP server
DNS server
Internet
Domain
MTA server
User account
Via IMAP
Via POP
Recipient receives email from server

Transmission of an email message

To send an email, in most cases the mail client communicates with an outgoing mail server, which will help relay that message to its final destination. The mail client transmits messages to the mail server using the Simple Mail Transfer Protocol (SMTP). The outgoing mail relay may require no authentication from internal clients, in which case the server listens on port 25/TCP. In that case, the relay will restrict which hosts ca n relay through IP address-based restrictions or firewall rules.

The outgoing mail relay then uses DNS to look up the MX record identifying the mail server that accepts delivery for messages sent to the recipient's domain. The relay then uses SMTP on port 25/TCP to transmit the email to that server. The recipient's mail service may provide a POP3 or IMAP server, such as Dovecot or Cyrus, to allow a dedicated mail client to download their messages. Frequently, the mail service provides a web-based interface, allowing clients to use a web browser as a mail client. The following image illustrates how an email client retrieves incoming mail from an I MAP server and sends outgoing mail through an SMTP server. The mail client on serverX.example.com fetches incoming mails from the IMAP server imapX.example.com. Outgoing mails are sent to smtpX. example.com. An MX DNS record defines smtpX.example.com as the responsible mail server for the desktopX.example.com domain.



Email client communication

Mail client

"Email client," "mail client," "mail program," and "mail reader," it provides the ability to send and receive email messages and file attachments.

Popular email clients include Microsoft Outlook, Mozilla Thunderbird, macOS Mail, IncrediMail, Mailbox and iOS Mail. The most popular web-based email client is Gmail; others include Yahoo! Mail and Outlook.com

Rain Loop is a free webmail application based on PHP, it’s free and open source, has modern user interface to handle large number of email accounts without the need of any database connectivity, besides non database connectivity it holds both SMTP and IMAP protocols to easily send/receive emails without any trouble.

Protocol	**Port number**
POP3	110
IMAP	143
SMTP	25

Shell scripting

Many simple day-to-day system administration tasks can be accomplished by the numerous Linux command-line tools available to administrators. However, tasks with greater complexity often require the chaining together of multiple commands. In these situations, Linux command-line tools can be combined with the offerings of the Bash shell to create powerful shell scripts to solve real-world problems. In its simplest form, a Bash shell script is simply an executable file composed of a list of commands. However, when well-written, a shell script can itself become a powerful command-line tool when executed on its own, and can even be further leveraged by other scripts. Proficiency in shell scripting is essential to the success of Linux system administrators in all operational environments. Working knowledge of shell scripting is especially crucial in enterprise environments, where its use can translate to improved efficiency and accuracy of routine task completion.

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. Shell scripts are similar to the batch file in MS-DOS. Each shell script is saved with .sh file extension e.g. myscript.sh. A shell script has syntax just like any other programming language. If you have any prior experience with any programming language like Python, C/C++ etc. it would be very easy to get started with it.
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

Why do we need shell scripts?

There are many reasons to write shell scripts –

- To avoid repetitive work and automation
- System admins use shell scripting for routine backups
- System monitoring
- Adding new functionality to the shell etc.

Advantages of shell scripts

- The command and syntax are exactly the same as those directly entered in command line, so programmer do not need to switch to entirely different syntax
- Writing shell scripts are much quicker
- Quick start
- Interactive debugging etc.

Disadvantages of shell scripts

- Prone to costly errors, a single mistake can change the command which might be harmful
- Slow execution speed
- Design flaws within the language syntax or implementation
- Not well suited for large and complex task
- Provide minimal data structure unlike other scripting languages. Etc

Basic Operators in Shell Scripting

There are **5** basic operators in bash/shell scripting:

1. Arithmetic Operators
2. Relational Operators
3. Boolean Operators
4. Bitwise Operators
5. File Test Operators

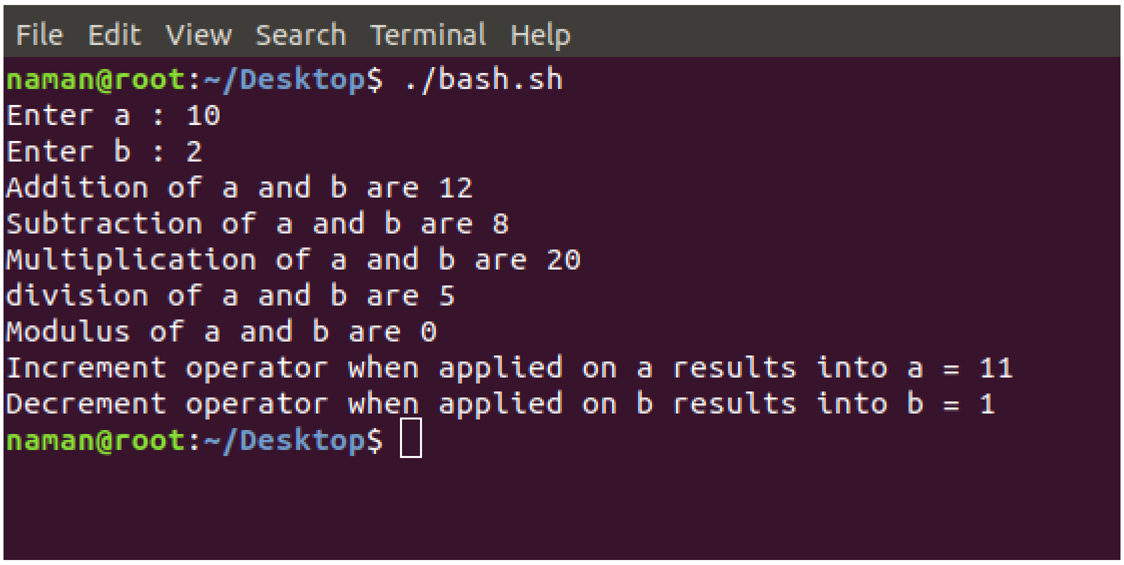
1. Arithmetic Operators: These operators are used to perform normal arithmetic/mathematical operations. There are 7 arithmetic operators:

- **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 an operand by one

```
#!/bin/bash
#reading data from the user
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
((++a))
echo Increment operator when applied on "a" results into a = $a
((--b))
echo Decrement operator when applied on "b" results into b = $b
```

Output:



```
naman@root:~/Desktop$ ./bash.sh
Enter a : 10
Enter b : 2
Addition of a and b are 12
Subtraction of a and b are 8
Multiplication of a and b are 20
division of a and b are 5
Modulus of a and b are 0
Increment operator when applied on a results into a = 11
Decrement operator when applied on b results into b = 1
naman@root:~/Desktop$
```

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. Its returns true is they are equal otherwise returns false.
- **'!=' Operator**: Not Equal to operator return true if the two operands are not equal otherwise it returns false.

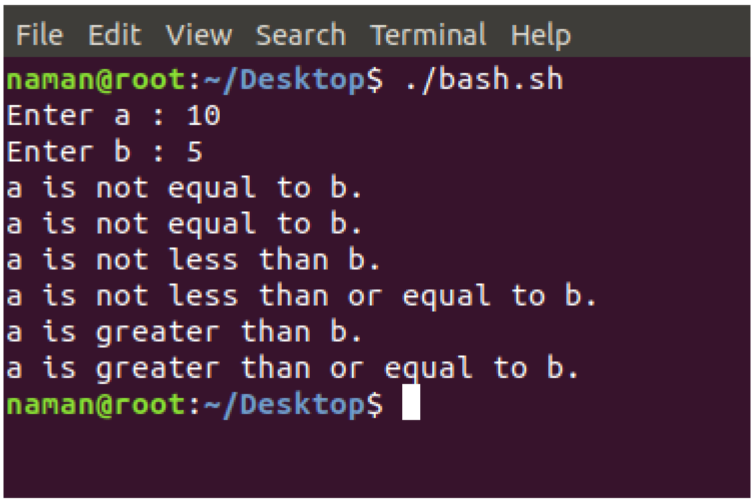
- **‘< ‘Operator**: Less than operator returns true if first operand is less than second operand otherwise returns false.
- **‘<=’ Operator**: Less than or equal to operator returns true if first operand is less than or equal to second operand otherwise returns false
- **‘>’ Operator**: Greater than operator return true if the first operand is greater than the second operand otherwise return false.
- **‘>=’ Operator**: Greater than or equal to operator returns true if first operand is greater than or equal to second operand otherwise returns false

```
#!/bin/bash

#Reading data from the user

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
```

Output:



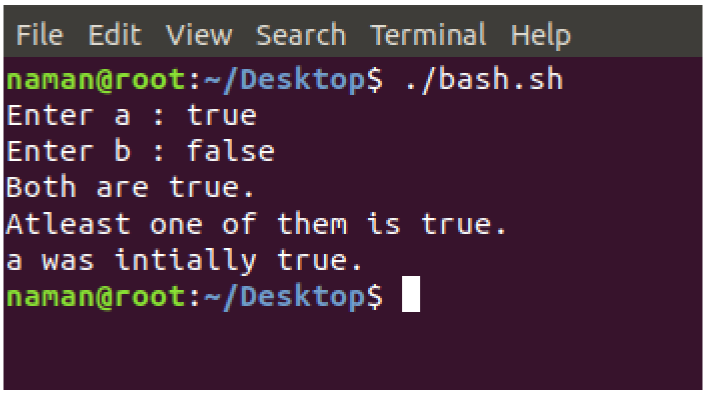
```
naman@root:~/Desktop$ ./bash.sh
Enter a : 10
Enter b : 5
a is not equal to b.
a is not equal to b.
a is not less than b.
a is not less than or equal to b.
a is greater than b.
a is greater than or equal to b.
naman@root:~/Desktop$
```

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 is 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.

```
#!/bin/bash

#reading data from the user

read -p 'Enter a : ' a

read -p 'Enter b : ' b

if(($a == "true" & $b == "true" ))

then

echo Both are true.

els

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
```

Output:



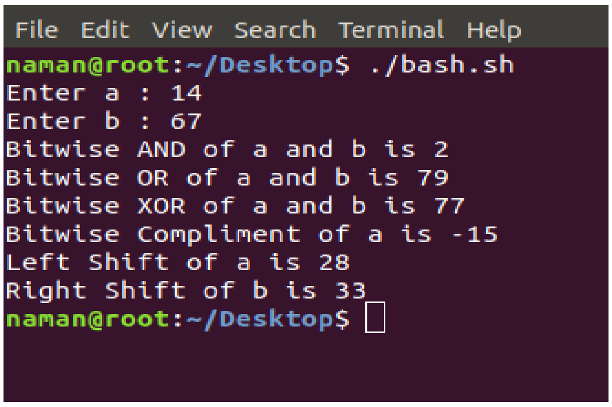
```
naman@root:~/Desktop$ ./bash.sh
Enter a : true
Enter b : false
Both are true.
Atleast one of them is true.
a was intially true.
naman@root:~/Desktop$
```

4. 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.

```
#!/bin/bash

#reading data from the user

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
```

Output:



```
naman@root:~/Desktop$ ./bash.sh
Enter a : 14
Enter b : 67
Bitwise AND of a and b is 2
Bitwise OR of a and b is 79
Bitwise XOR of a and b is 77
Bitwise Compliment of a is -15
Left Shift of a is 28
Right Shift of b is 33
naman@root:~/Desktop$
```

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

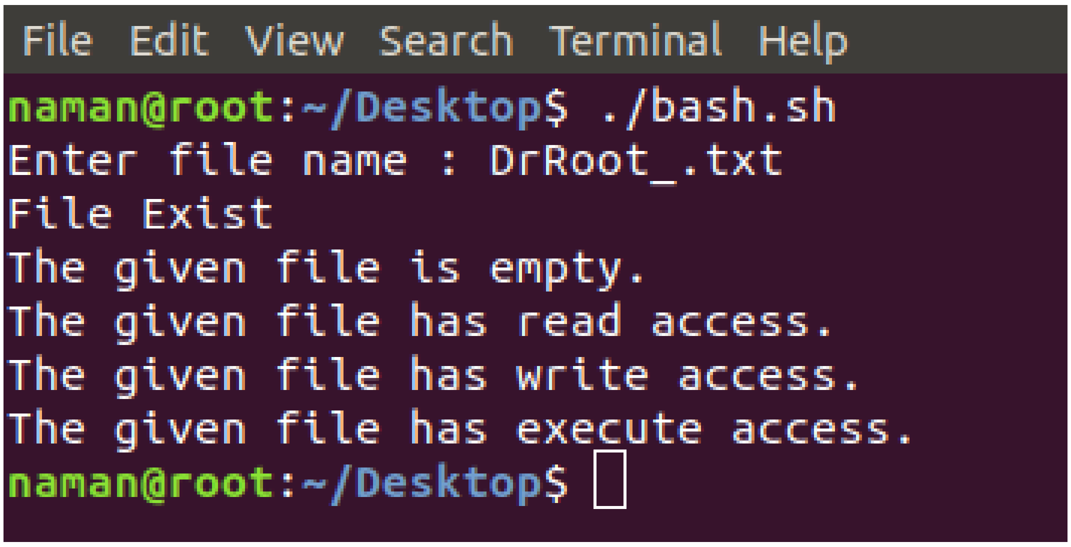
- **-b operator**: This operator checks whether a file is a block special file or not. It returns true if the file is a block special file otherwise false.
- **-c operator**: This operator checks whether a file is a character special file or not. It returns true if it is a character special file otherwise false.
- **-d operator**: This operator checks if the given directory exists or not. If it exists then operators return true otherwise false.
- **-e operator**: This operator checks whether the given file exists or not. If it exits 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 checks whether the given file has written access or not. If it has written then it returns true otherwise false.
- **-x operator**: This operator checks whether the given file has executed access or not. If it has executed 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.

```
#!/bin/bash

#reading data from the user

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
```

Output:



```
naman@root:~/Desktop$ ./bash.sh
Enter file name : DrRoot_.txt
File Exist
The given file is empty.
The given file has read access.
The given file has write access.
The given file has execute access.
naman@root:~/Desktop$
```

Conditional Statements

There are total 5 conditional statements which can be used in bash programming

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

Their description with syntax is as follows:

1. **If statement**

   This block will process if specified condition is true.

   Syntax:

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

2. **if-else statement**

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

   Syntax

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

3. **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
```

4. **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
```

5. **switch statement**

   case statement works as a switch statement if specified value match with the pattern, then it will execute a block of that particular pattern
   When a match is found all of the associated statements until the double semicolon (; ;) is executed.
   A case will be terminated when the last command is executed.
   If there is no match, the exit status of the case is zero.

Syntax:

```
case in
Pattern 1) Statement 1;;
Pattern n) Statement n;;
esac
```

Example Programs

Example 1: Implementing if statement

```
#Initializing two variables
a=10
b=20

#Check whether they are equal
if [ $a == $b ]
then
    echo "a is equal to b"
fi

#Check whether they are not equal
if [ $a != $b ]
then
    echo "a is not equal to b"
fi
```

Output

```
$bash -f main.sh

a is not equal to b
```

Example2*:* Implementing if.else statement

```
#Initializing two variables
a=20
b=20
```

```
if [ $a == $b ]
then
   #If they are equal then print this
   echo "a is equal to b"
else
   #else print this
   echo "a is not equal to b"
fi
```

Output

```
$bash -f main.sh

a is equal to b
```

Example3: Implementing switch statement

```
CARS="bmw"

#Pass the variable in string
case "$CARS" in
   #case 1
   "mercedes") echo "Headquarters - Affalterbach, Germany" ;;

   #case 2
   "audi") echo "Headquarters - Ingolstadt, Germany" ;;

   #case 3
   "bmw") echo "Headquarters - Chennai, Tamil Nadu, India" ;;
esac
```

Output

```
$bash -f main.sh

Headquarters - Chennai, Tamil Nadu, India.
```

Note: Shell scripting is a case-sensitive language, which means proper syntax has to be followed while writing the scripts.