A shell script is a computer program designed to be run by the Unix/Linux shell which could be one of the following:

- The Bourne Shell
- The C Shell
- The Korn Shell
- The GNU Bourne-Again Shell

A shell is a command-line interpreter and typical operations performed by shell scripts include file manipulation, program execution, and printing text.

Extended Shell Scripts

Shell scripts have several required constructs that tell the shell environment what to do and when to do it.

The shell is, after all, a real programming language, complete with variables, control structures, and so forth. No matter how complicated a script gets, it is still just a list of commands executed sequentially.

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.

```
#!/bin/sh
echo "What is your name?"
read PERSON
echo "Hello, $PERSON"
```

Here is a sample run of the script −

```
$./test.sh
What is your name?
Zara Ali
Hello, Zara Ali
$
```

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. A word is an unbroken set of characters. Spaces and tabs separate words.

Following is a simple example of the **date** command, which displays the current date and time −

$date
Thu Jun 25 08:30:19 MST 2009

You can customize your command prompt using the environment variable PS1 explained in the Environment tutorial.

Shell Types

In Unix, there are two major types of shells −

- **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)

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, preceded by **#** sign, describing the steps.

Example Script

Assume we create a **test.sh** script. Note all the scripts would have the **.sh** extension. Before you add anything else to your script, you need to alert the system that a shell script is being started. This is done using the **shebang** construct. For example −

```
#!/bin/sh
```

This tells the system that the commands that follow are to be executed by the

Bourne shell. *It's called a shebang because the # symbol is called a hash, and the ! symbol is called a bang*.

To create a script containing these commands, you put the shebang line first and then add the commands −

```
#!/bin/bash
pwd
ls
```

Shell Comments

You can put your comments in your script as follows −

```
#!/bin/bash

# Author : Zara Ali
# Copyright (c) Tutorialspoint.com
```

```
# Script follows here:
pwd
ls
```

Save the above content and make the script executable −

```
$chmod +x test.sh
```

The shell script is now ready to be executed −

```
$./test.sh
```

Shell 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 following examples are valid variable names −

```
_ALI
TOKEN_A
VAR_1
VAR_2
```

Following are the examples of invalid variable names −

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

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="Zara Ali"
```

The above example defines the variable NAME and assigns the value "Zara Ali"

to it. Variables of this type are called **scalar variables**. A scalar variable can hold only one value at a time.

Shell enables you to store any value you want in a variable. For example −

```
VAR1="Zara Ali"
VAR2=100
```

Accessing Values

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

For example, the following script will access the value of defined variable NAME and print it on STDOUT −

```
#!/bin/sh

NAME="Zara Ali"
echo $NAME
```

The above script will produce the following value −

Zara Ali

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.

For example, the following script generates an error while trying to change the value of NAME −

```
#!/bin/sh

NAME="Zara Ali"
readonly NAME
NAME="Qadiri"
```

The above script will generate the following result −

/bin/sh: NAME: This variable is read only.

Command line Argument

Command-line arguments are values passed to a script or command when it's executed. In Bash, these arguments are accessible through the special variables *$1*, *$2*, *$3*, up to *$9*. In this case, *$1* represents the first argument, *$2* represents the second argument, and so forth. These arguments are also contained in the argument array, represented by the special variable $@.

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.

$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.

```
#add two numbers using command line argument
echo "File name : $0"
echo "First number : $1"
echo "2nd number : $2"
sum=$(($1+$2))
echo "Sum=$sum"

echo "Total number of parameters : $#"
```

Array

The difference between an array variable and a scalar variable can be explained as follows.

Suppose you are trying to represent the names of various students as a set of variables. Each of the individual variables is a scalar variable as follows −

```
NAME01="Zara"
NAME02="Qadir"
NAME03="Mahnaz"
NAME04="Ayan"
NAME05="Daisy"
```

We can use a single array to store all the above mentioned names. Following is the simplest method of creating an array variable. This helps assign a value to one of its indices.

array_name[index]=value

Here *array_name* is the name of the array, *index* is the index of the item in the array that you want to set, and value is the value

you want to set for that item.

As an example, the following commands −

```
NAME[0]="Zara"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
```

If you are using the **bash** shell, here is the syntax of array initialization −

array_name=(value1 value2 value3 ... valuen)

Accessing Array Values

After you have set any array variable, you access it as follows −

${array_name[index]}

Here *array_name* is the name of the array, and *index* is the index of the value to be accessed.

Following is an example to understand the concept −

```
#!/bin/sh

NAME[0]="Zara"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
echo "First Index: ${NAME[0]}"
echo "Second Index: ${NAME[1]}"
```

The above example will generate the following result −

```
$./test.sh
First Index: Zara
Second Index: Qadir
```

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

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

Here **array_name** is the name of the array you are interested in. Following example will help you

understand the concept −

```
#!/bin/sh

NAME[0]="Zara"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
echo "First Method: ${NAME[*]}"
echo "Second Method: ${NAME[@]}"
```

The above example will generate the following result –

```
$./test.sh
First Method: Zara Qadir Mahnaz Ayan Daisy
Second Method: Zara Qadir Mahnaz Ayan Daisy
```

Shell Basic Operators

```
#!/bin/sh

val=`expr 2 + 2`
echo "Total value : $val"
```

The above script will generate the following result −

```
Total value : 4
```

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.

Arithmetic Operators

The following arithmetic operators are supported by Bourne Shell.

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

Show Examples

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.

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.

Relational Operators

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.

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.

Boolean Operator

The following Boolean operators are supported by the Bourne Shell.

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

Show Examples

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

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:

if statement

This block will process if specified condition is true.

Syntax:

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

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
```

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
```

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
```

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
```

```
#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

#Switch case program

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"
                echo “hiiiii ”;;
esac
```

The bash **case** statement is the simplest form of the if elif else conditional statement.
The **case** statement simplifies complex conditions with multiple different choices. This statement is easier to maintain and more readable than nested **if** statements.

The **case** statement tests the input value until it finds the corresponding pattern and executes the command linked to that input value. Thus, it is an excellent choice for creating menus where users select an option which triggers a corresponding action.

The bash **case** statement takes the following syntax:

```
case $variable in
        pattern-1)   echo “hi this ia pattern 1”;;
        pattern-2)  commands;;
        pattern-3)  commands;;
        pattern-N)  commands;;
                *)  commands;;
esac
```

The **case** statement starts with the **case** keyword followed by the **$variable** and the **in** keyword. The statement ends with the **case** keyword backwards - **esac**.

$variable

- The script compares the input **$variable** against the patterns in each clause until it finds a match.

Patterns

- A pattern and its commands make a **clause**, which ends with **;;**.
- Patterns support special characters.
- The **)** operator terminates a pattern list.
- The | operator separates multiple patterns.
- The script executes the commands corresponding to the first pattern matching the input **$variable**.
- The asterisk ***** symbol defines the default case, usually in the final pattern.

Example 1:

```
#!/bin/bash
echo "Which color do you like best?"
echo "1 - Blue"
echo "2 - Red"
echo "3 - Yellow"
echo "4 - Green"
echo "5 - Orange"
read color;
case $color in
        1) echo "Blue is a primary color.";;
        2) echo "Red is a primary color.";;
        3) echo "Yellow is a primary color.";;
        4) echo "Green is a secondary color.";;
        5) echo "Orange is a secondary color.";;
        *) echo "This color is not available. Please choose a different one.";;
esac
```

Example 2:

```
#!/bin/bash
echo "Enter the name of a month."
read month
case $month in
        February)  echo "There are 28/29 days in $month.";;
        April | June | September | November)  echo "There are 30 days in $month.";;
        January | March | May | July | August | October | December)  echo "There are 31 days in $month.";;
        *)  echo "Unknown month. Please check if you entered the correct month name: $month";;
esac
```

Example 3:

```
#!/bin/bash
echo "Choose a contact to display information:"
echo "[C]hristina Th."
echo "[J]ames warner"
echo "[S]arah Paonam"
echo "[R]ose Armstrong"
```

```
read person
case "$person" in
  "C" | "c" ) echo "Christina Th."
                    echo "Imphal west"
                    echo "Sagolband Meino Leirak";;
  "J" | "j" ) echo "James Gardner"
                    echo "Imphal West"
                    echo "Kwakeithel ";;
  "S" | "s") echo "Sarah Paonam"
                    echo "Imphal East";;
  "R" | "r") echo "Rose Armstrong"
                    echo "Imphal West";;
  *) echo "Contact doesn't exist.";;
esac
```

Looping Statements in Shell Scripting: There are total 3 looping statements that can be used in bash programming.

While statement:

Here the command is evaluated and based on the resulting loop will execute, if the command is raised to false then the loop will be terminated **that.**

Syntax:

```
#/bin/bash
while control_command
do
   statement1
   ...
   statementN
done
```

where,
control_command can be any command that exits with a success or failure status.
The **statements** in the body of the while loop can be any utility commands, user programs, shell scripts, or shell statements.

When a while statement is executed, the control_command is evaluated. If the control_command succeeds, all the statements between the do and done are executed, and then the controlling command is again executed.

As long as the control_command succeeds, the loop body continues to execute. As soon as the control_command fails, the statement following the done statement is executed.

Example 1:

```
COUNT=1

while [ $COUNT -le 10 ]
do
   echo "Loop count is $COUNT"
   COUNT=$((COUNT+1))
done

echo "Done"
```

The for Loop Structure

Using the for loop in shell scripts is reasonably straightforward, and you can manipulate the structure to achieve different goals.
The basic structure is as follows:

```
for item in [LIST]
do
      [COMMANDS]
done
```

```
for VARIABLE in 1 2 3 4 5 .. N
do
  command1
  command2
  commandN
done
```

You can define the number of iterations in the first line. This way, you'll mention the starting value and the ending value.
The number of iterations is determined by the values you specify, while the code following the **do** statement is the resulting loop value.

Print Integers Using for Loops

1. Loop Code to Print a Set of Numbers

```
#!/usr/bin/bash

for i in  1 2 3

do
     echo "Current # $i"
done
```

Where:

- **i** = variable name to store the iterated values
- **1 2 3** = number of times the for loop in shell script iterates
- **do** = command to perform a certain set of actions
- **echo** = print the results defined alongside
- **done** = end of the loop

2. Alternate Way to Print a Set of Numbers

There are alternate ways to define a for loop in a shell script. You can also specify the starting and ending value of the loop's iterations using curly brackets.

```
for i in {1..10} # a for loop defines a variable and how many iterations you want to make through a loop
do
    echo "Current # $i: Example 2"
done
```

3. Loop Code Using Step Values

You can define the step values in your loop if you want to move nonsequentially through the iterations. Depending on the value specified, the output will have a fixed gap.

```
for i in {1..10..2}
do
     echo "Number = $i"
done
```

Where:

- **i** = variable to store the iterations
- **1..10** = number of iterations to run the loop
- **2** = step value
- **do** = command to print the output
- **echo** = print command

Print Character Values Using for Loops

For loops in shell scripting isn't restricted to just integers. In Bash, you can use a for loop to effectively iterate through characters and string values.

1. Looping Through Strings

Here's a basic example of how you can loop through some string values (defined in the for statement):

```
for name in  John Jack Mary
do
        echo "My name is $name"
done
```

Where:

- **name** = variable to store the string values
- **do** = command to print the output
- **echo** = print command
- **done** = exit command for the loop
-

```
#!/bin/bash
# For loop with letter range
for i in {a..f}
do
    echo "Element $i"
done
```

```
#!/bin/bash
# For loop with reverse number range
for i in {5..0}
do
    echo "Element $i"
done
```

```
#!/bin/bash
# For loop with reverse range increment numbers
for i in {10..0..2}
do
    echo "Element $i"
done
```

Bash scripts allow C-style three parameter for loop control expressions. Add the expression between double parentheses as follows:

```
#!/bin/bash
# For loop C-style
for (( i=0; i<=5; i++ ))
do
  echo "Element $i"
done
```

Arrays

Arrays store a list of elements. The **for** loop provides a method to go through arrays by element. For example, define an array and loop through the elements with:

```
#!/bin/bash
# For loop with array
array=(1 2 3 4 5)
for i in ${array[@]}
do
      echo "Element $i"
done
```

Nested Loops

To loop through or generate multi-dimensional arrays, use nested **for** loops.

```
#!/bin/bash
# Nested for loop
for (( i = 0; i <= 2; i++ ))
do
   for (( j = 0 ; j <= 2; j++ ))
   do
         echo -n " * "
   done
  echo ""
done
```

Files

The **for** loop combined with proximity searches helps list or alter files that meet a specific condition.For example, list all Bash scripts in the current directory with a **for** loop:

```
#!/bin/bash
# For loop with files
for f in *.sh
do
      echo $f
done
```

Command Substitution

The **for** loop accepts command substitution as a list of elements to iterate through.

```
#!/bin/bash
# For loop with command substitution
list=`cat dev1.sh`
# Alternatively, use list=$(cat list.txt)
for i in $list
do
      echo $i
done
```