

# What is Shell Scripting

In Linux, shells like bash and korn support programming construct which are saved as scripts. These scripts become shell commands and hence many Linux commands are script.

A system administrator should have a little knowledge about scripting to understand how their servers and applications are started, upgraded, maintained or removed and to understand how a user environment is built.

## How to determine Shell

You can get the name of your shell prompt, with following command :

### Syntax:

```
echo $SHELL
```

A terminal window with a dark purple background. The prompt is 'sssit@JavaTpoint: ~'. The user enters 'echo \$SHELL' and the output is '/bin/bash'. The prompt then returns to 'sssit@JavaTpoint:~\$'.

Look at the above snapshot, with the help of above command we got the name of our shell which is '**bash**'.

The \$ sign stands for a shell variable, echo will return the text whatever you typed in.

## Steps to write and execute a script

- Open the terminal. Go to the directory where you want to create your script.
- Create a file with **.sh** extension.
- Write the script in the file using an editor.
- Make the script executable with command **chmod +x <fileName>**.
- Run the script using **./<fileName>**.

**Note:** In the last step you have to mention the path of the script if your script is in other directory.

# Shell Scripting She-bang

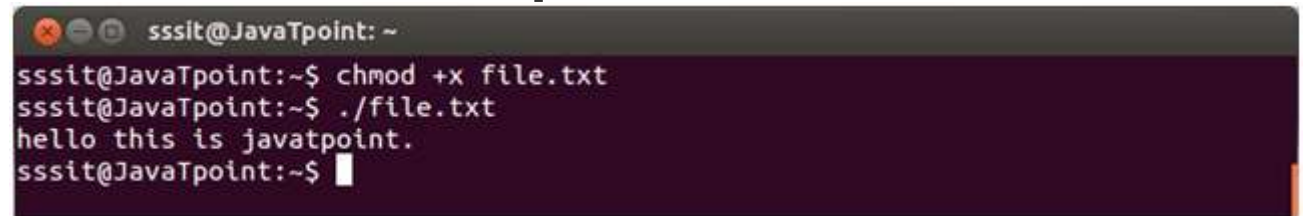
The sign **#!** is called she-bang and is written at top of the script. It passes instruction to program **/bin/sh**.

To run your script in a certain shell (shell should be supported by your system), start your script with **#!** followed by the shell name.

## Example:

```
#!/bin/bash
echo Hello World
#!/bin/ksh
echo Hello World
```

# Execute Shell Scripts

A terminal window titled 'sssit@JavaTpoint: ~' showing the execution of a script. The user runs 'chmod +x file.txt' and then './file.txt'. The output is 'hello this is javatpoint.'.

```
sssit@JavaTpoint: ~
sssit@JavaTpoint:~$ chmod +x file.txt
sssit@JavaTpoint:~$ ./file.txt
hello this is javatpoint.
sssit@JavaTpoint:~$
```

# Shell Scripting Variables

Scripts can contain variables inside the script.

A terminal window titled 'sssit@JavaTpoint: ~' showing a script with variable assignments and an echo command. The script starts with '#!/bin/bash', assigns 'var1=Hello' and 'var2=Jai', and then echoes the values using '\$var1' and '\$var2'.

```
#!/bin/bash
var1=Hello
var2=Jai
#
echo "$var1 $var2"
```

Look at the above snapshot, two variables are assigned to the script **\$var1** and **\$var2**.

As scripts run in their own shell, hence variables do not survive the end of the script.

```
sssit@JavaTpoint: ~  
sssit@JavaTpoint:~$ ./exm.sh  
Hello Jai  
sssit@JavaTpoint:~$ echo $var1  
  
sssit@JavaTpoint:~$ echo $var2  
  
sssit@JavaTpoint:~$
```

Look at the above snapshot, **var1** and **var2** do not run outside the script.

## Troubleshooting a shell script

There is one more way other than script execution to run a script in a different shell. Type bash with the name of the script as parameter.

### Syntax:

```
bash <fileName>
```

### Example:

```
sssit@JavaTpoint: ~  
sssit@JavaTpoint:~$ bash exm  
Hello  
sssit@JavaTpoint:~$
```

Look at the above snapshot, it displays the **exm** script content with bash command.

```
sssit@JavaTpoint: ~  
#!/bin/bash  
var1=Hello  
#  
echo $var1
```

Look at the above snapshot, this is the exm script we have written. By expanding **bash** command with **-x**, shell allows us to see commands that the shell is executing.

```
sssit@JavaTpoint: ~  
sssit@JavaTpoint:~$ bash -x exm  
+ var1=Hello  
+ echo Hello  
Hello  
sssit@JavaTpoint:~$
```

Look at the above snapshot, with command **bash -x**, we can see the shell expansion.

# Reading Input

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

Following script use the **read** command which takes the input from the keyboard and assigns it as the value of the variable PERSON and finally prints it.

```
#!/bin/sh

# Author : Zara Ali
# Script follows here

echo "What is your name?"
read PERSON
echo "Hello, $PERSON"
```

Here is sample run of the script:

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

## Using 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 would 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"
```

Above example defines the variable `NAME` and assigns it the value `"Zara Ali"`. Variables of this type are called scalar variables. A scalar variable can hold only one value at a time.

The 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 ( `$` ):

```
#!/bin/sh  
  
NAME="Zara Ali"  
echo $NAME
```

This would produce following value:

```
Zara Ali
```

## Read-only Variables

The shell provides a way to mark variables as read-only by using the `readonly` command. After a variable is marked read-only, its value cannot be changed. For example, following script would give error while trying to change the value of `NAME`:

```
#!/bin/sh
NAME="Zara Ali"
readonly NAME
NAME="Qadiri"
```

This would produce following result:

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

## Unsetting Variables

Unsetting or deleting a variable tells the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you would not be able to access stored value in the variable.

Following is the syntax to unset a defined variable using the **unset** command:

```
unset variable_name
```

Above command would unset the value of a defined variable. Here is a simple example:

```
#!/bin/sh
NAME="Zara Ali"
unset NAME
echo $NAME
```

Above example would not print anything. You cannot use the `unset` command to **unset** variables that are marked **readonly**.

## 7.12 Special Variables

These variables are reserved for specific functions. For example, the \$\$ character represents the process ID number, or PID, of the current shell:

```
$echo $$
```

Above command would write PID of the current shell:

```
29949
```

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

Variable	Description
\$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.
\$\$	The process number of the current shell. For shell scripts, this is the process ID under which they are executing.

## 7.13 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.

Following script uses various special variables related to command line:

```
#!/bin/bash
echo "File Name: $0"
echo "First Parameter : $1"
echo "First Parameter : $2"
echo "Total Number of Parameters : $#"
```

Here is a sample run for the above script:

```
$/test.sh Zara Ali
File Name : ./test.sh
First Parameter : Zara
Second Parameter : Ali
Total Number of Parameters : 2
```

## Special Parameters \$\*

There are special parameters that allow accessing all of the command-line arguments at once. \$\* special parameter takes the entire list as one argument with spaces between them.

We can write the shell script shown below to process an unknown number of command-line arguments with \$\*

```
#!/bin/bash

echo "All Parameters : $*"

```

There is one sample run for the above script:

```
$/test.sh Zara Ali 10 Years Old
Zara Ali 10 Years Old
```

## Unix - 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 right actions.

Unix Shell supports conditional statements which are used to perform different actions based on different conditions. Here we will explain following two decision making statements:

- The **if...else** statements
- The **case...esac** statement

### 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.



- 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
```

Here Shell *expression* is evaluated. If the resulting value is *true*, given *statement(s)* are executed. If *expression* is *false* then no statement would be not executed. Most of the times you will use comparison operators while making decisions.

Give you attention on the spaces between braces and expression. This space is mandatory otherwise you would get syntax error.

## Example:

```
#!/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
```

This will produce following result:

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

- if...else...fi statement

The **if...else...fi** statement is the next form of control statement that allows Shell to execute statements in more controlled way and making decision between two choices.

## 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
```

Here Shell *expression* is evaluated. If the resulting value is *true*, given *statement(s)* are executed. If *expression* is *false* then no statement would be not executed.

## Example:

If we take above example then it can be written in better way using *if...else* statement as follows:

```
#!/bin/sh

a=10
b=20

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

This will produce following result:

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

- **if...elif...else...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
elif [ expression 3 ]
then
    Statement(s) to be executed if expression 3 is true
else
    Statement(s) to be executed if no expression is true
fi
```

There is nothing special about this code. It 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.

## Example:

```
#!/bin/sh

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
```

This will produce following result:

```
a is less than b
```

# The case...esac Statement

You can use multiple **if...elif** statements to perform a multiway 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.

## Syntax

The basic syntax of the **case...esac** statement is to give an expression to evaluate and several different statements to execute 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.

```
case word 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
    ;;
esac
```

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 program flow should jump to the end of the entire case statement. This is similar to **break** in the C programming language.

## Example:

```
#!/bin/sh

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
```

This will produce following result:

```
New Zealand is famous for kiwi.
```