Shell Scripting Tutorial

Shell Scripting tutorial provides basic and advanced concepts of Shell Scripting. Our Shell Scripting tutorial is designed for beginners and professionals.

Shell Scripting is an open-source operating system.

Our Shell Scripting tutorial includes all topics of Scripting executing scripting, loops, scripting parameters, shift through parameters, sourcing, getopts, case, eval, let etc. There is also given Shell Scripting interview questions to help you better understand the Shell Scripting operating system.

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.

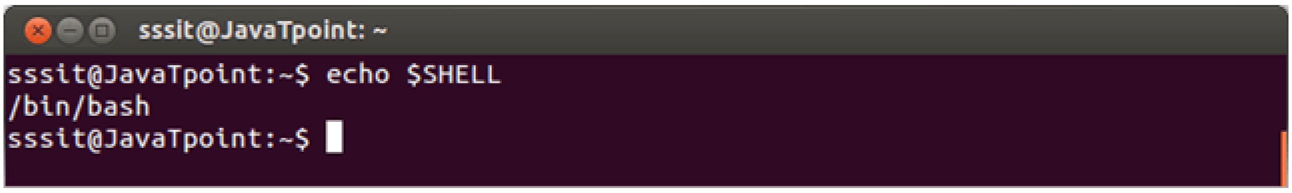
A system administrator should have a little knowledge about scripting to understand how their servers and applications are started, upgraded, maintained or removed and to understand how a user environment is built.

How to determine Shell

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

Syntax:

1. echo $SHELL



```
sssit@JavaTpoint: ~
sssit@JavaTpoint:~$ echo $SHELL
/bin/bash
sssit@JavaTpoint:~$ █
```

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.

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.**

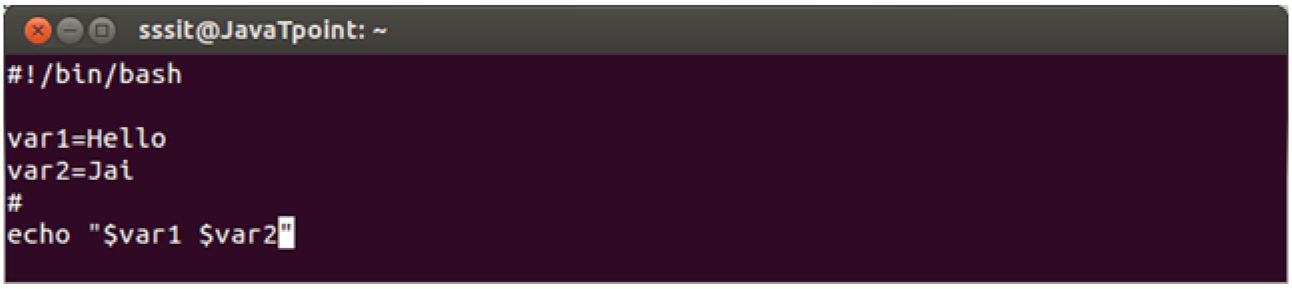
To run your script in a certain shell (shell should be supported by your system), start your script with #! followed by the shell name.

Example:

1. #!/bin/bash
2. echo Hello World
3. #!/bin/ksh
4. echo Hello World

Shell Scripting Variables

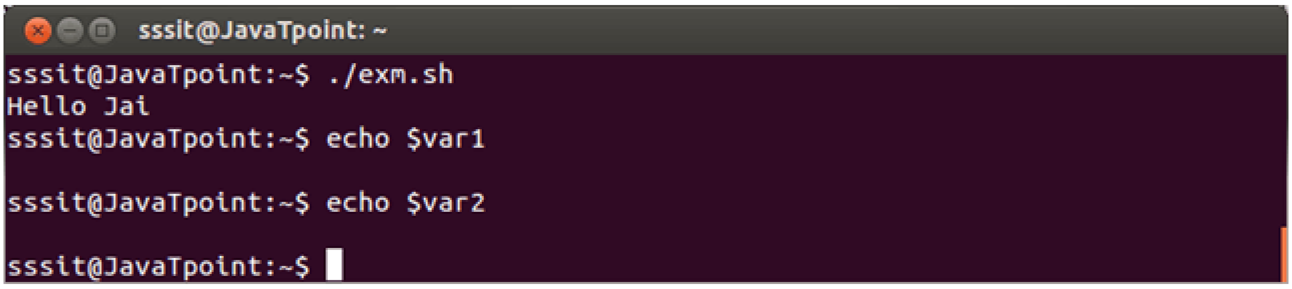
Scripts can contain variables inside the script.



```
#!/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.

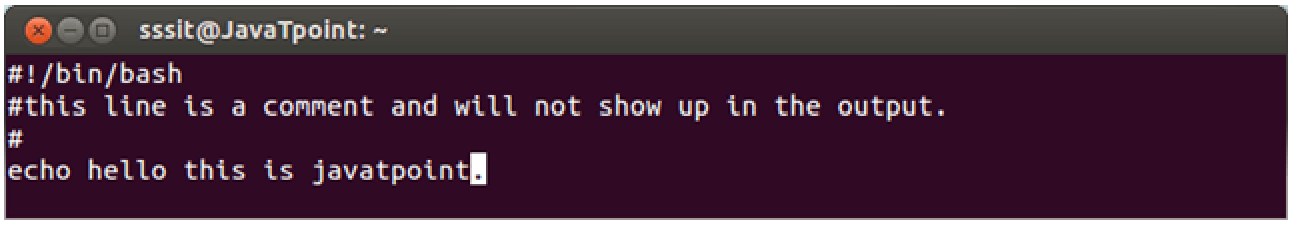


```
sssit@JavaTpoint:~$ ./exm.sh
Hello Jai
sssit@JavaTpoint:~$ echo $var1

sssit@JavaTpoint:~$ echo $var2

sssit@JavaTpoint:~$
```

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

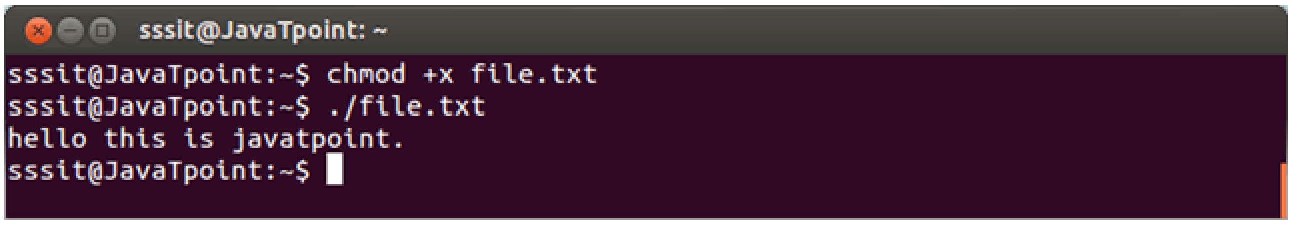
Shell Scripting Comments

Any line starting with a hash (#) becomes comment. Comment means, that line will not take part in script execution. It will not show up in the output.



```
sssit@JavaTpoint: ~
#!/bin/bash
#this line is a comment and will not show up in the output.
#
echo hello this is javatpoint.
```

Look at the above snapshot, lines after the # are commented.



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

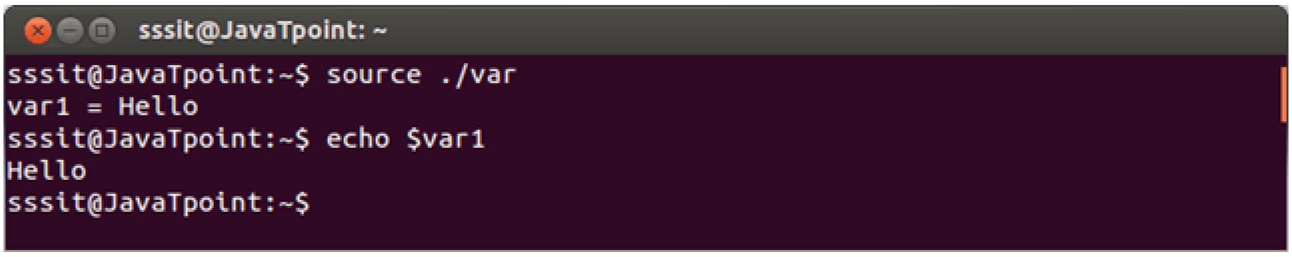
Look at the above snapshot, commented lines are not displayed in the output.

Shell Scripting Sourcing a file

A file is sourced in two ways. One is either writting as **source** <**fileName**> or other is writting as . ./<**filename>** in the command line. When a file is sourced, the code lines are executed as if they were printed on the command line.

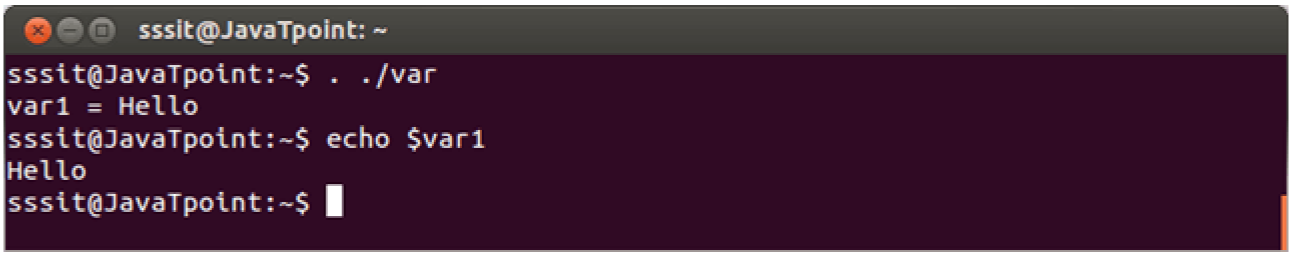
The difference between sourcing and executing a script is that, while executing a script it runs in a new shell whereas while sourcing a script, file will be read and executed in the same shell.

In sourcing, script content is displayed in the same shell but while executing script run in a different shell.



```
sssit@JavaTpoint: ~
sssit@JavaTpoint:~$ source ./var
var1 = Hello
sssit@JavaTpoint:~$ echo $var1
Hello
sssit@JavaTpoint:~$
```

Look at the above snapshot, we have sourced the file **var** with one of the method.



```
sssit@JavaTpoint: ~
sssit@JavaTpoint:~$ . ./var
var1 = Hello
sssit@JavaTpoint:~$ echo $var1
Hello
sssit@JavaTpoint:~$
```

Look at the above snapshot, we have sourced the file **var** with another method.

Steps to write and execute a script

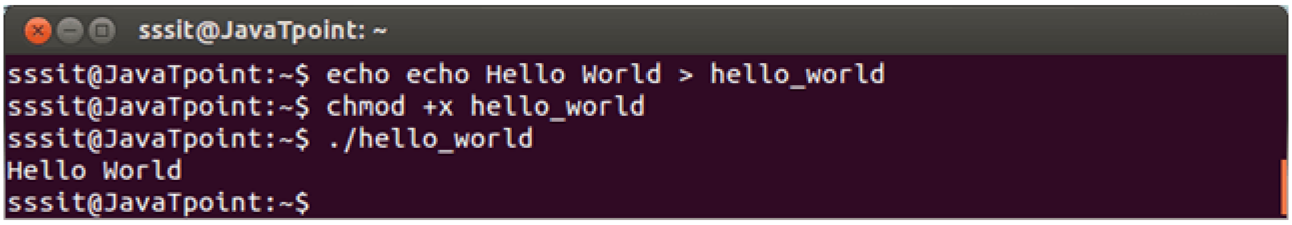
- Open the terminal. Go to the directory where you want to create your script.
- Create a file with **.sh** extension.
- Write the script in the file using an editor.
- Make the script executable with command **chmod +x <fileName>**.
- Run the script using ./<**fileName**>.

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

Hello World script

Here we'll write a simple programme for Hello World.

First of all, create a simple script in any editor or with echo. Then we'll make it executable with **chmod +x** command. To find the script you have to type the script path for the shell.



```
sssit@JavaTpoint: ~
sssit@JavaTpoint:~$ echo echo Hello World > hello_world
sssit@JavaTpoint:~$ chmod +x hello_world
sssit@JavaTpoint:~$ ./hello_world
Hello World
sssit@JavaTpoint:~$
```

Look at the above snapshot, script **echo Hello World** is created with echo command as **hello_world.** Now command **chmod +x hello_world** is passed to make it executable. We have given the command **./hello_world** to mention the hello_world path. And output is displayed.

Shell Script Parameters

A bash shell script have parameters. These parameters start from **$1** to **$9.**

When we pass arguments into the command line interface, a positional parameter is assigned to these arguments through the shell.

The first argument is assigned as $1, second argument is assigned as $2 and so on...

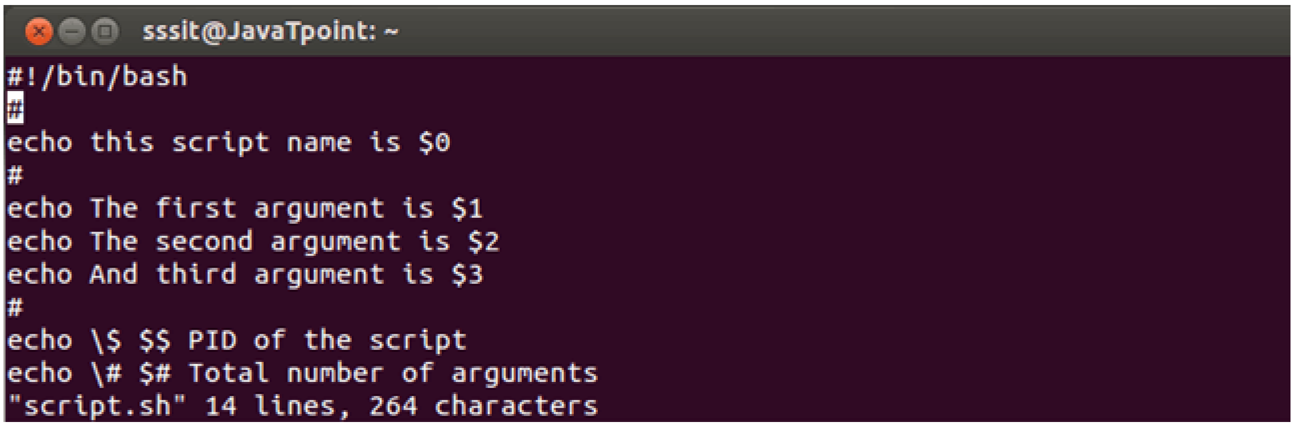
If there are more than 9 arguments, then **tenth** or onwards arguments can't be assigned as $10 or $11.

You have to either process or save the $1 parameter, then with the help of **shift** command drop parameter 1 and move all other arguments down by one. It will make $10 as $9, $9 as $8 and so on.

Shell Parameters

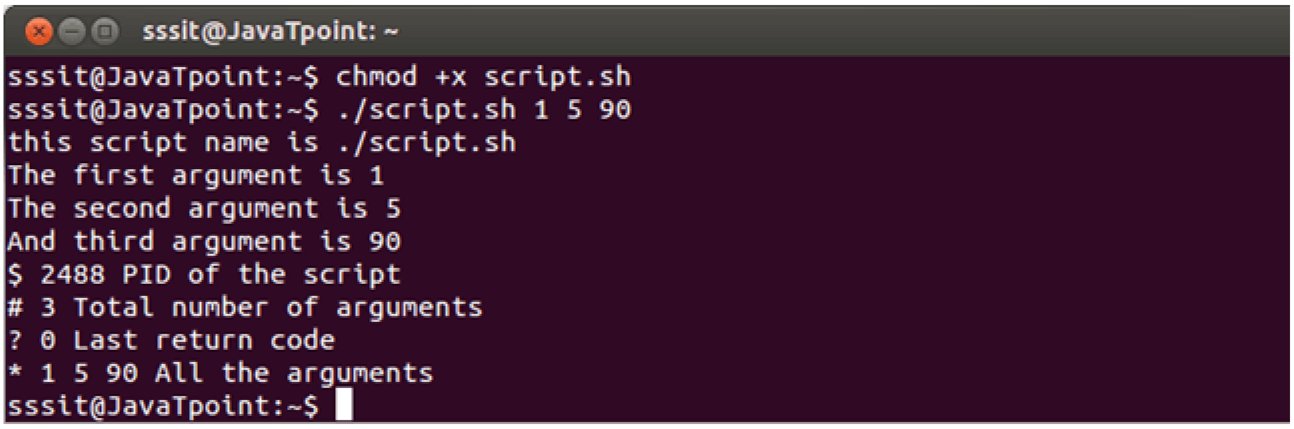
Parameters	Function
$1-$9	Represent positional parameters for arguments one to nine
${10}-${n}	Represent positional parameters for arguments after nine
$0	Represent name of the script
$∗	Represent all the arguments as a single string
$@	Same as $∗, but differ when enclosed in (")
$#	Represent total number of arguments
$$	PID of the script
$?	Represent last return code

Example:



```
sssit@JavaTpoint: ~
#!/bin/bash
#
echo this script name is $0
#
echo The first argument is $1
echo The second argument is $2
echo And third argument is $3
#
echo \$ $$ PID of the script
echo \# $# Total number of arguments
"script.sh" 14 lines, 264 characters
```

Look at the above snapshot, this is the script we have written to show the different parameters.



```
sssit@JavaTpoint: ~
sssit@JavaTpoint:~$ chmod +x script.sh
sssit@JavaTpoint:~$ ./script.sh 1 5 90
this script name is ./script.sh
The first argument is 1
The second argument is 5
And third argument is 90
$ 2488 PID of the script
# 3 Total number of arguments
? 0 Last return code
* 1 5 90 All the arguments
sssit@JavaTpoint:~$
```

Look at the above snapshot, we have passed arguments **1, 5, 90**. All the parameters show their value when script is run.

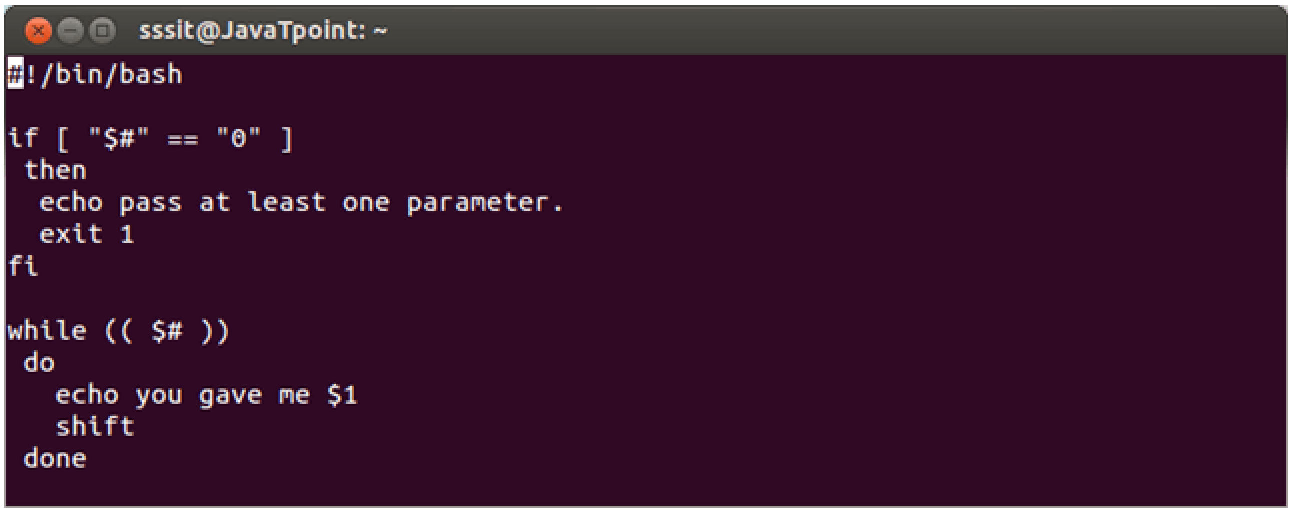
Shell Scripting Shift Through Parameters

Shift command is a built-in command. Command takes number as argument. Arguments shift down by this number.

For example, if number is 5, then $5 become $1, $6 become $2 and so on.

Example:

The shift command is mostly used when arguments are unknown. Arguments are processed in a while loop with a condition of **(($#))**. this condition holds true as long as arguments are not zero. Number of arguments are reduced each time as the shift command executes.



```
sssit@JavaTpoint: ~
#!/bin/bash

if [ "$#" == "0" ]
 then
  echo pass at least one parameter.
  exit 1
fi

while (( $# ))
 do
   echo you gave me $1
   shift
 done
```

Look at the above snapshot, this is our script.

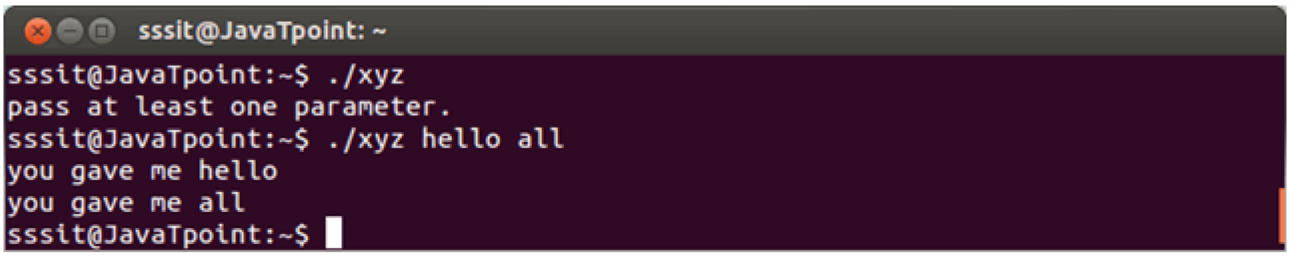
Shell Scripting Sourcing a config file

Many programs use external configuration files. Use of external configuration files prevents a user from making changes to a script. Config file is added with the help of **source** command.

If a script is shared in many users and every user need a different configuration file, then instead of changing the script each time simply include the config files.

Example:

We have two files, one is parent file **(main.sh)** and other is configuration file **(config.sh)**. We have to source this configuration file into our parent file.



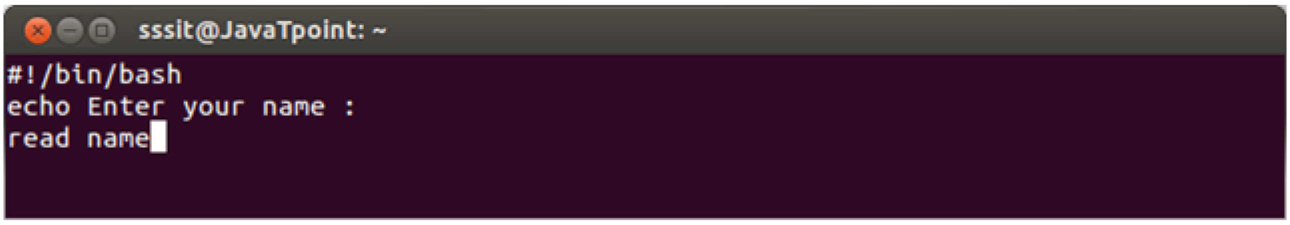
sssit@JavaTpoint: ~

```
sssit@JavaTpoint:~$ ./xyz
pass at least one parameter.
sssit@JavaTpoint:~$ ./xyz hello all
you gave me hello
you gave me all
sssit@JavaTpoint:~$
```

Look at the above snapshot, this is the output of the above script.

read command

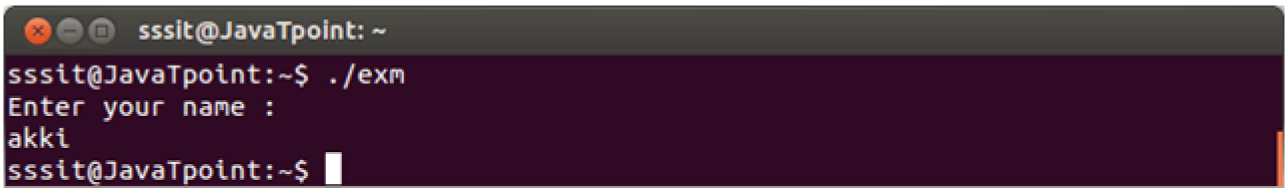
The read command allows a user to provide the runtime input.



sssit@JavaTpoint: ~

```
#!/bin/bash
echo Enter your name :
read name
```

Look at the above snapshot, this is our script using read command.

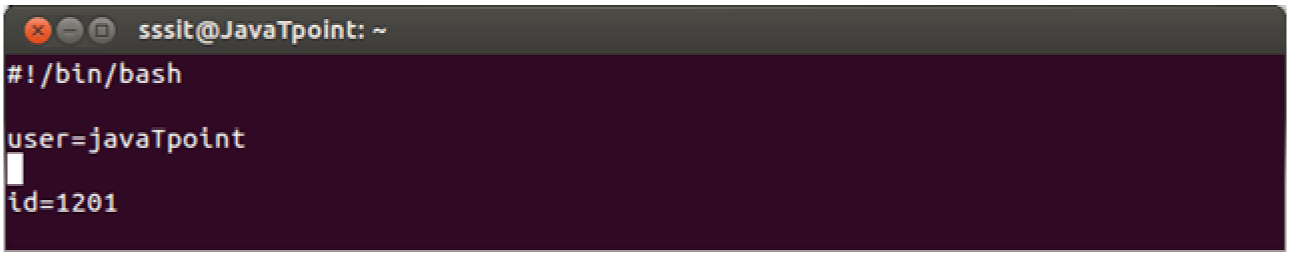


sssit@JavaTpoint: ~

```
sssit@JavaTpoint:~$ ./exm
Enter your name :
akki
sssit@JavaTpoint:~$
```

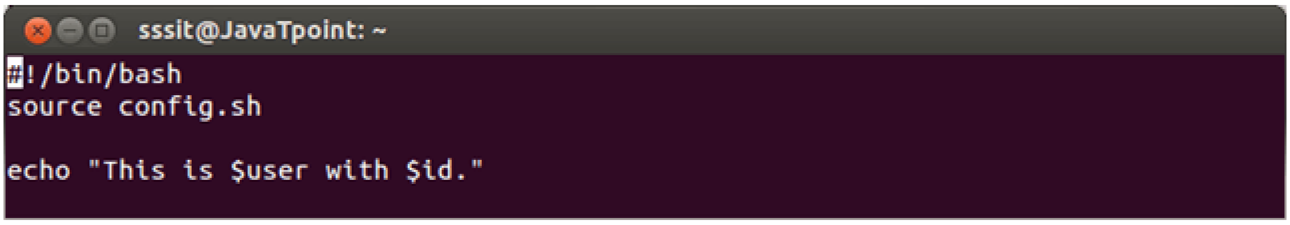
Look at the above snapshot, a user can enter the name in the shell.

Script for config.sh



```
#!/bin/bash

user=javaTpoint

id=1201
```

Script for main.sh

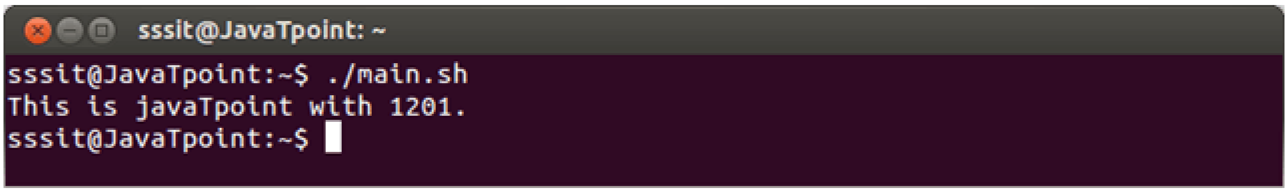


```
#!/bin/bash
source config.sh

echo "This is $user with $id."
```

Look at the above snapshot, we've included config.sh file with source command.

Note: We can also use **(. config.sh)** command instead of **(source config.sh)** .

Now on running main.sh file, config.sh file is included.



```
sssit@JavaTpoint:~$ ./main.sh
This is javaTpoint with 1201.
sssit@JavaTpoint:~$
```

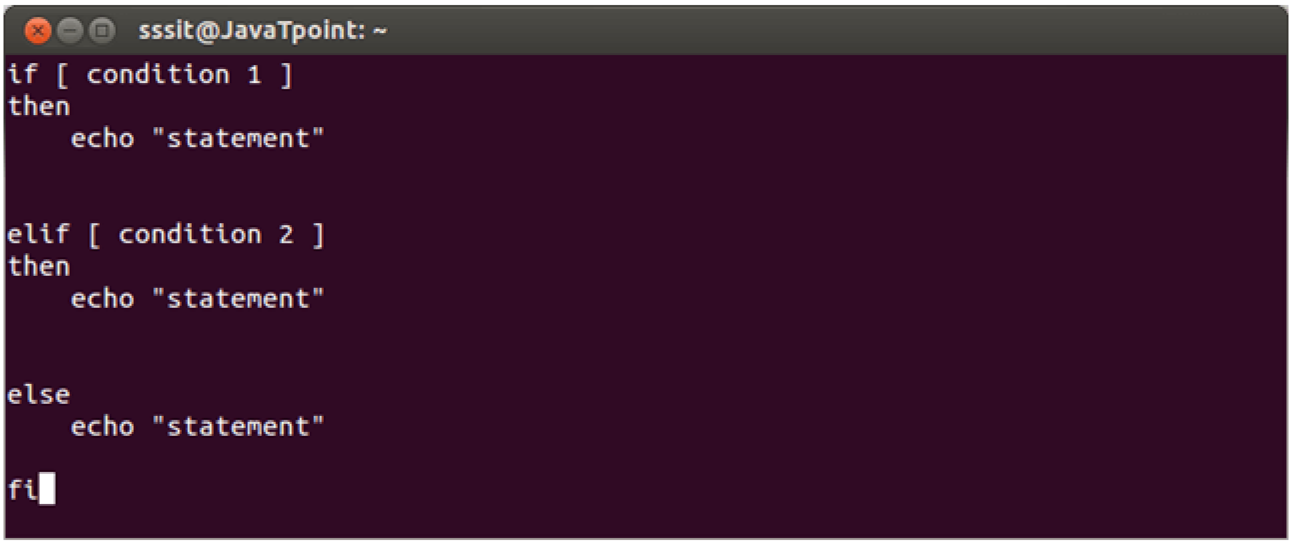
Look at the above snapshot, on running main.sh file, content of config.sh file is imported via source command.

Shell Scripting if then elif

A new if can be nested inside an elif.

Syntax:

Syntax of if then elif is shown in the snapshot below,



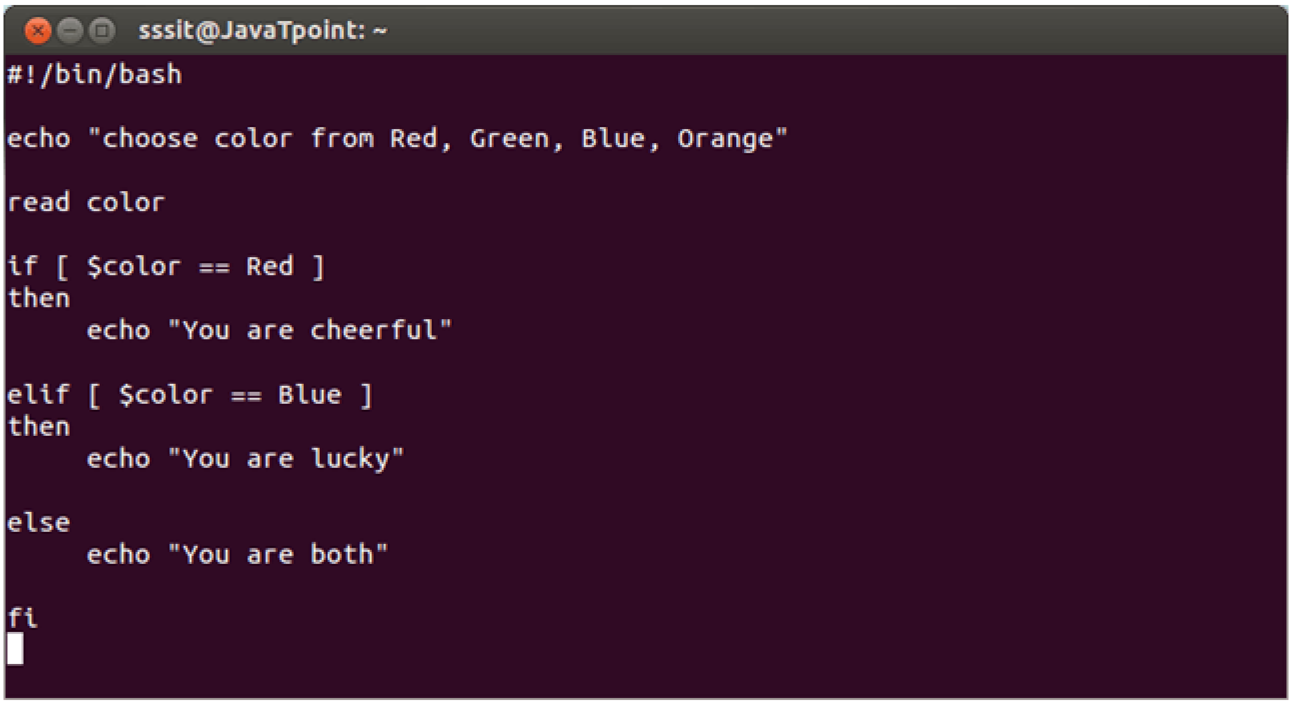
```
sssit@JavaTpoint: ~
if [ condition 1 ]
then
    echo "statement"


elif [ condition 2 ]
then
    echo "statement"


else
    echo "statement"

fi
```

Example if then elif:

We have shown the example of choosing **color.**

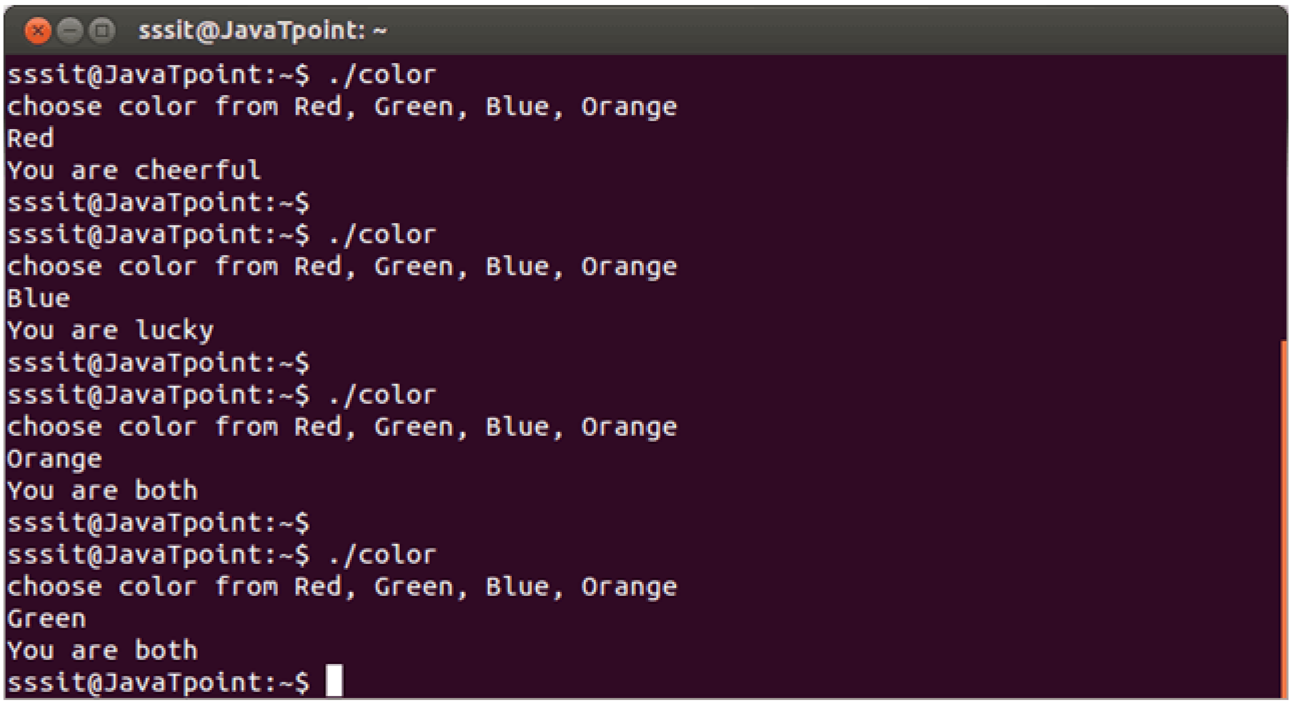
Condition:

1. if [$color == Red]
2. elif [$color == Blue]



```
sssit@JavaTpoint: ~
#!/bin/bash

echo "choose color from Red, Green, Blue, Orange"

read color

if [ $color == Red ]
then
     echo "You are cheerful"

elif [ $color == Blue ]
then
     echo "You are lucky"

else
     echo "You are both"

fi
```

Look at the above snapshot, we have shown the script.



```
sssit@JavaTpoint:~$ ./color
choose color from Red, Green, Blue, Orange
Red
You are cheerful
sssit@JavaTpoint:~$
sssit@JavaTpoint:~$ ./color
choose color from Red, Green, Blue, Orange
Blue
You are lucky
sssit@JavaTpoint:~$
sssit@JavaTpoint:~$ ./color
choose color from Red, Green, Blue, Orange
Orange
You are both
sssit@JavaTpoint:~$
sssit@JavaTpoint:~$ ./color
choose color from Red, Green, Blue, Orange
Green
You are both
sssit@JavaTpoint:~$
```

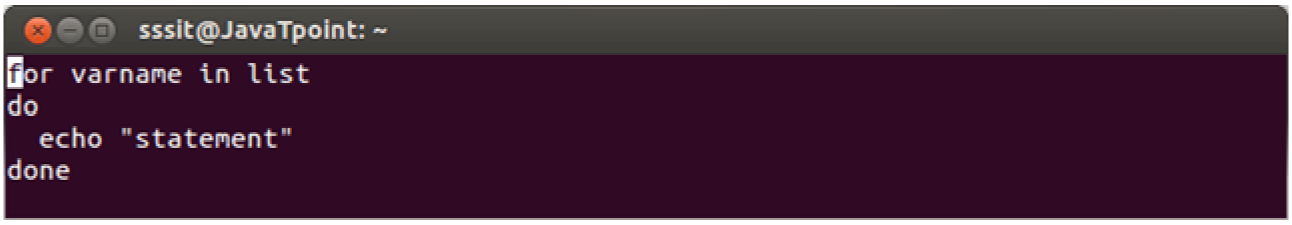
Look at the above snapshot, on **Red** color it goes to **if** part, on **Blue** color it goes to **elif** part and on other colors it goes to **else** part.

Shell Scripting for loop

The for loop moves through a specified list of values until the list is exhausted.

1) Syntax:

Syntax of for loop using **in** and list of values is shown below. This for loop contains a number of variables in the list and will execute for each item in the list. For example, if there are 10 variables in the list, then loop will execute ten times and value will be stored in varname.



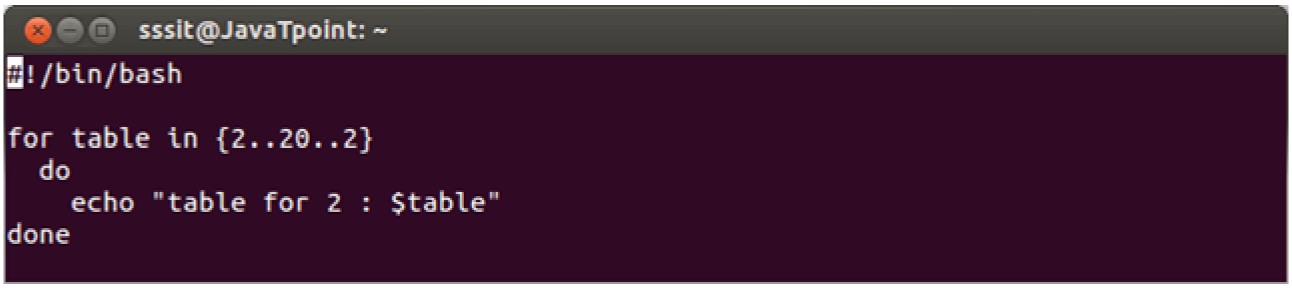
```
for varname in list
do
  echo "statement"
done
```

Look at the above syntax:

- Keywords are for, in, do, done
- List is a list of variables which are separated by spaces. If list is not mentioned in the for statement, then it takes the positional parameter value that were passed into the shell.
- Varname is any variable assumed by the user.

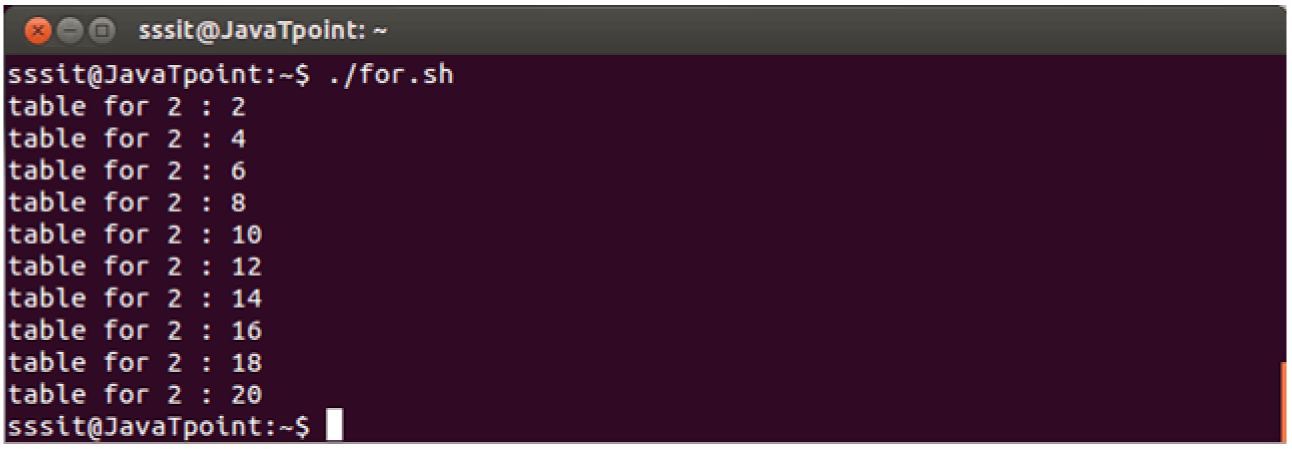
Example for:

We have shown an example to count 2's table within for loop.



```
#!/bin/bash

for table in {2..20..2}
  do
    echo "table for 2 : $table"
done
```

Look at the above snapshot, our varname is **table**, list is specified under curly braces. Within the curly braces, first two will initialize the table from 2, 20 represents maximum value of $table and last 2 shows the increment by value 2.

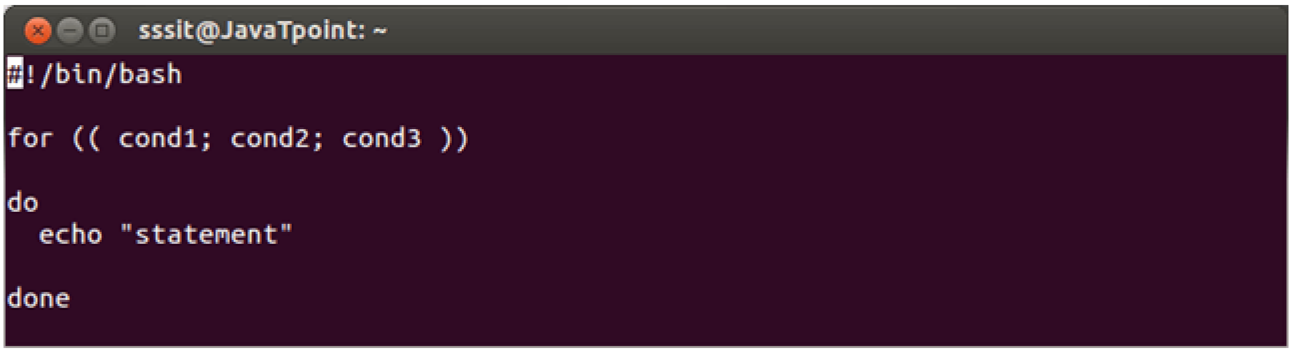


```
sssit@JavaTpoint:~$ ./for.sh
table for 2 : 2
table for 2 : 4
table for 2 : 6
table for 2 : 8
table for 2 : 10
table for 2 : 12
table for 2 : 14
table for 2 : 16
table for 2 : 18
table for 2 : 20
sssit@JavaTpoint:~$ 
```

Look at the above snapshot, it displays the 2's table as the output.

2) Syntax:

Syntax of for like C programming language.

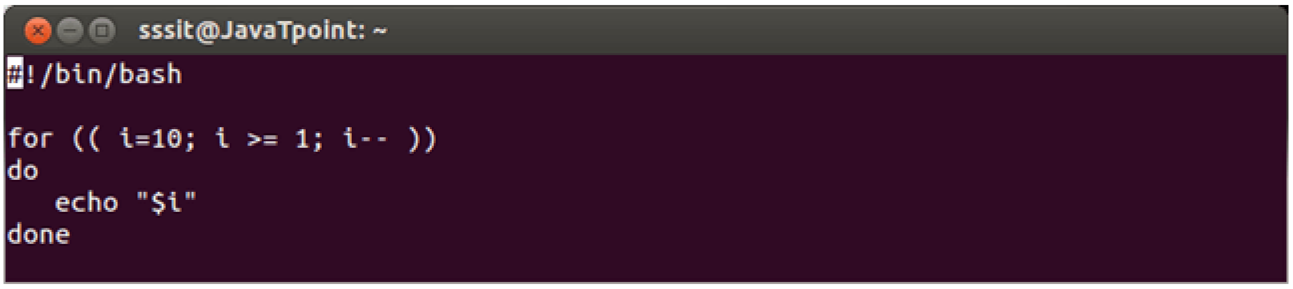


```
#!/bin/bash

for (( cond1; cond2; cond3 ))

do
  echo "statement"

done
```

Look at the above snapshot, condition1 indicates **initialization**, cond2 indicates **condition** and cond3 indicates **updation.**

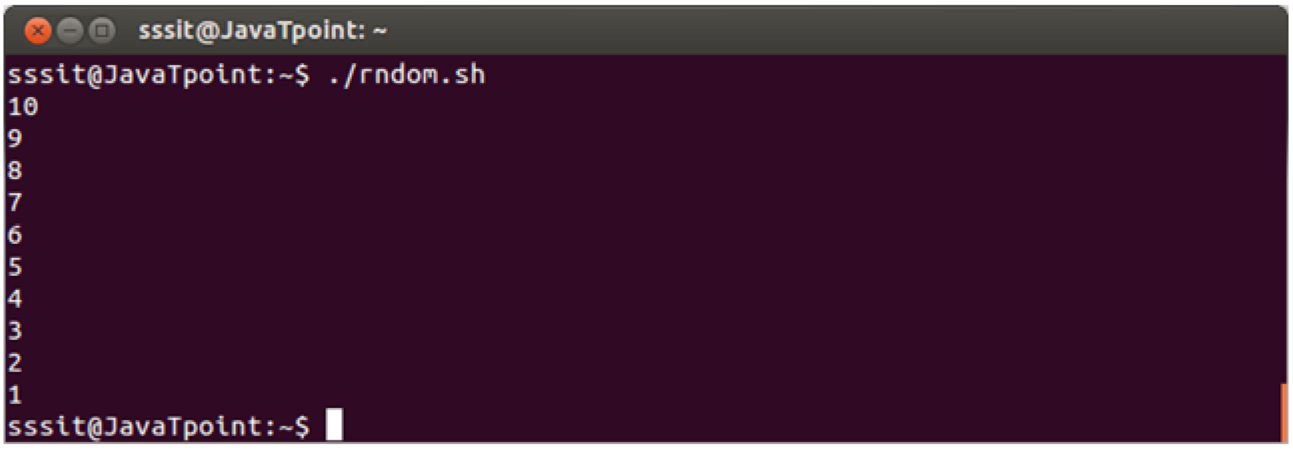
Example for:

We have shown an example to count the number in reverse direction.



```
#!/bin/bash

for (( i=10; i >= 1; i-- ))
do
   echo "$i"
done
```

Look at the above snapshot, this is the loop script. $i will initialize with 10 and will go till 1, decrementing with 1 value.



```
sssit@JavaTpoint:~$ ./rndom.sh
10
9
8
7
6
5
4
3
2
1
sssit@JavaTpoint:~$
```

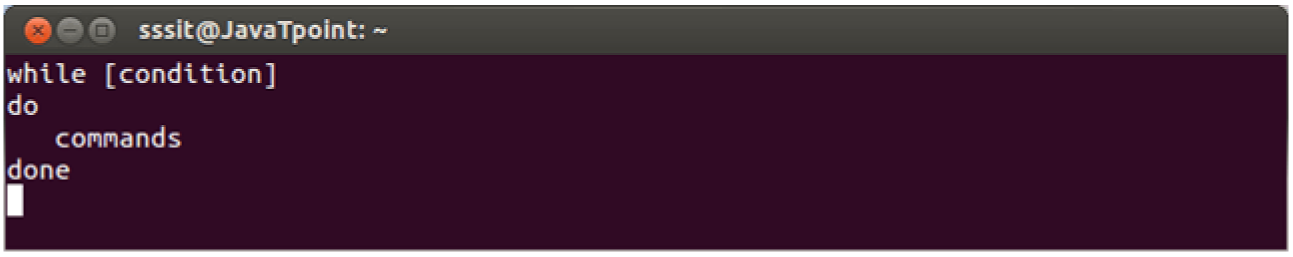
Look at the above snapshot, this is the output of the script

Shell Scripting while loop

Linux scripting while loop is similar to C language while loop. There is a condition in while. And commands are executed till the condition is valid. Once condition becomes false, loop terminates.

Syntax:

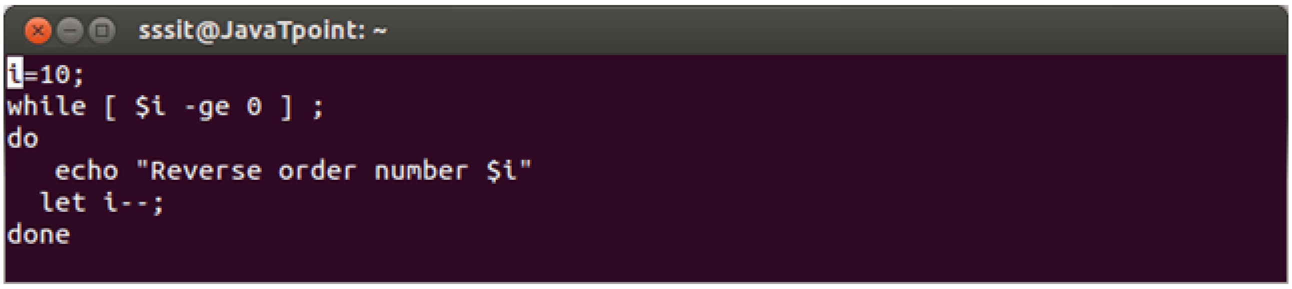
Syntax of while loop is shown in the snapshot below,



```
while [condition]
do
   commands
done
```

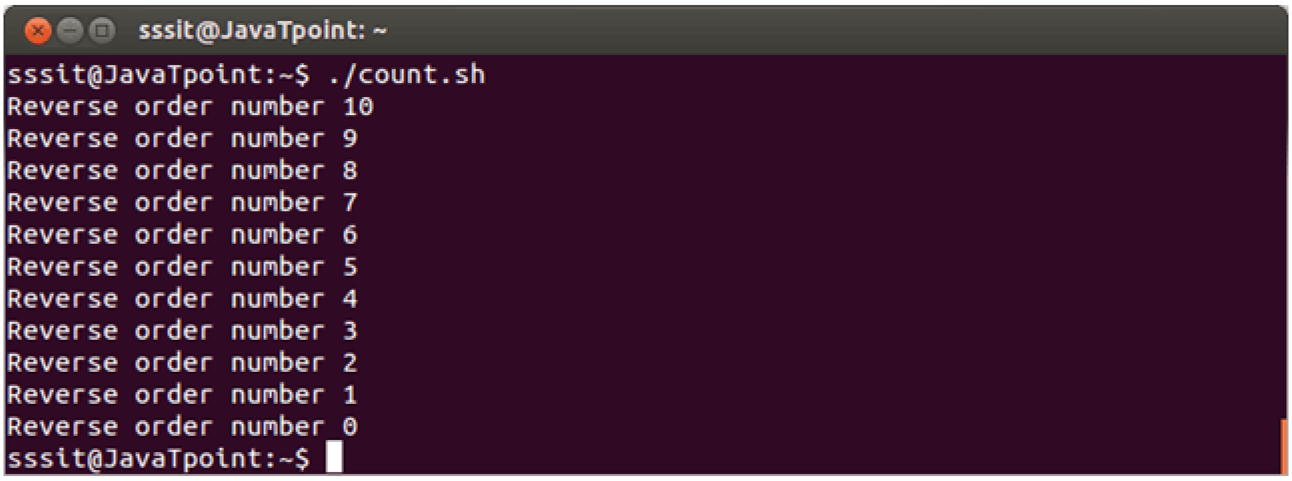
Example:

We have shown the example of printing number in reverse order.



```
i=10;
while [ $i -ge 0 ] ;
do
   echo "Reverse order number $i"
  let i--;
done
```

Output is displayed in the below snapshot,

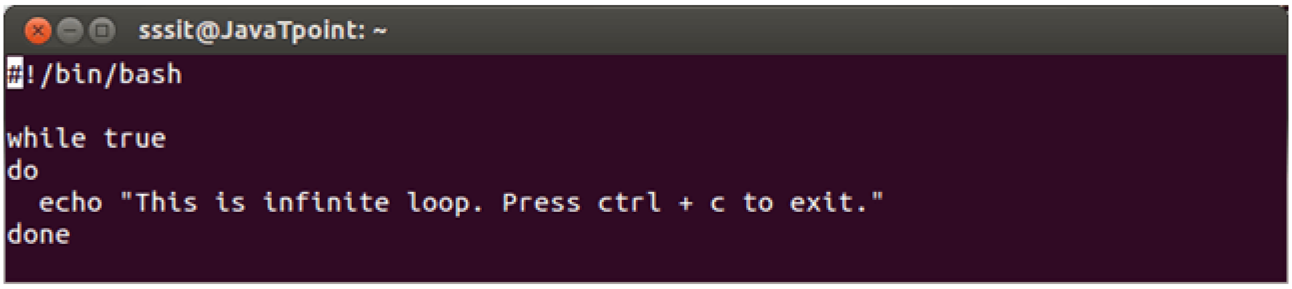


```
sssit@JavaTpoint:~$ ./count.sh
Reverse order number 10
Reverse order number 9
Reverse order number 8
Reverse order number 7
Reverse order number 6
Reverse order number 5
Reverse order number 4
Reverse order number 3
Reverse order number 2
Reverse order number 1
Reverse order number 0
sssit@JavaTpoint:~$
```

---

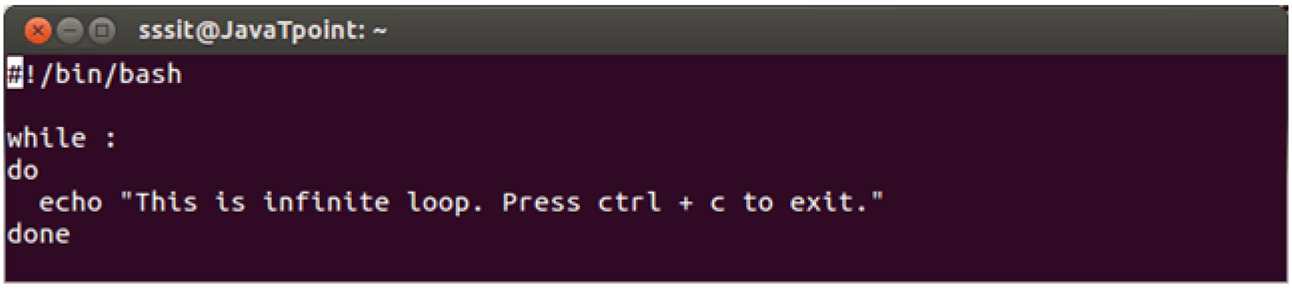
while infinite loop:

Infinite loop is also called endless loop. It is made with **while true** (it means condition will always be true) or **while** : (it means an empty expression), where colon (:) is equivalent to no operation.



```
#!/bin/bash

while true
do
  echo "This is infinite loop. Press ctrl + c to exit."
done
```

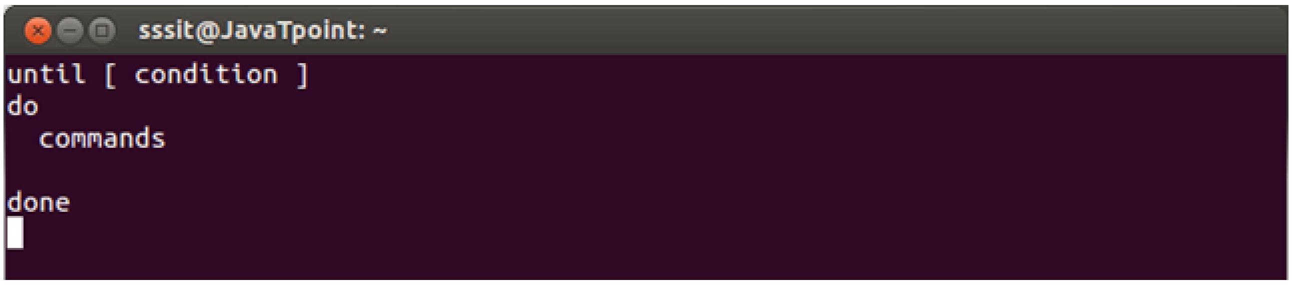
Look at the above snapshot, this script includes **while true**syntax.



```
#!/bin/bash

while :
do
  echo "This is infinite loop. Press ctrl + c to exit."
done
```

Look at the above snapshot, this script includes **while:** syntax.

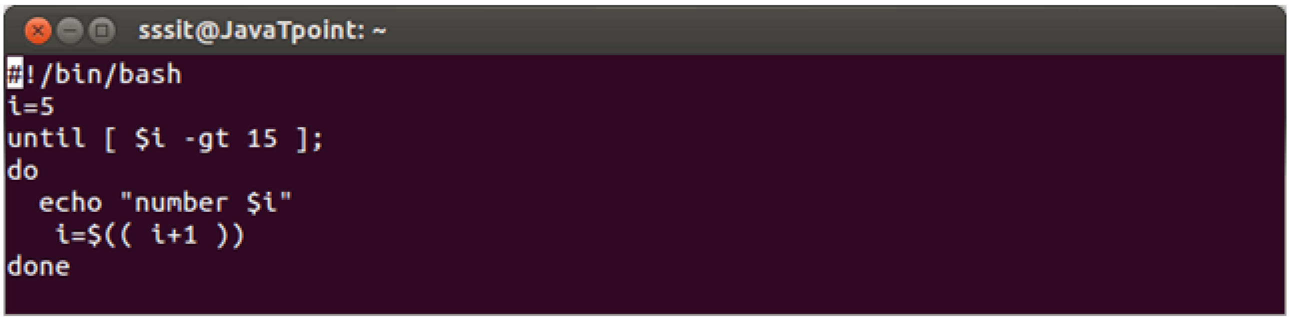
Both of them display the same output



```
until [ condition ]
do
  commands

done
```

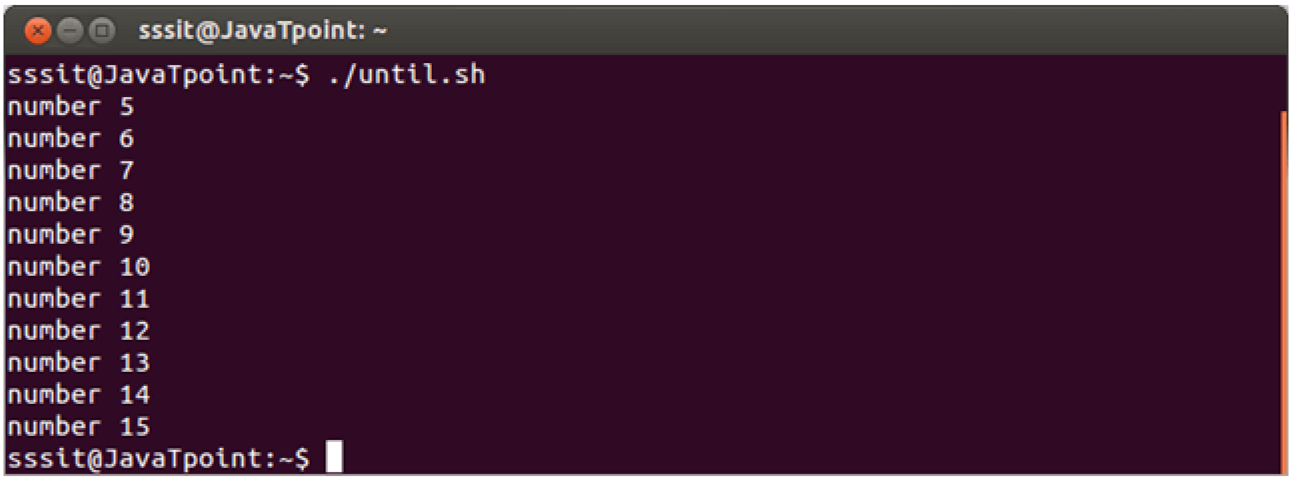
Example:

We have shown an example to display number from 5 to 15.



```
#!/bin/bash
i=5
until [ $i -gt 15 ];
do
  echo "number $i"
   i=$(( i+1 ))
done
```

Look at the above snapshot, it shows the script.



```
sssit@JavaTpoint:~$ ./until.sh
number 5
number 6
number 7
number 8
number 9
number 10
number 11
number 12
number 13
number 14
number 15
sssit@JavaTpoint:~$
```

Look at the above snapshot, it displays the output until the condition is false