Shell Script

Shell is a program that takes commands from the keyboard and gives them to the operating system to perform. LS / pwd

On most Linux systems a program called bash (which stands for Bourne Again SHell, an enhanced version of the original Unix shell program, sh, written by Steve Bourne) acts as the shell program. Besides Bourne shell, there are other shell programs that can be installed in a Linux system. These include **Korn Shell (ksh), Boune Again Shell (bash), and C Shell (csh)**

Bourne Shell (sh)

Developed by Stephen Bourne, from Bell Labs (AT&T from where Unix came from), this was, through several years, the most used Shell over the Unix operating system. Also known as "Standard Shell", because for couple of years it was the only available Shell. Even nowadays it's the most popular and present on all Unix and Linux distributions.

Boune Again Shell (bash)

This is the most powerful Shell whose user's number grows all around the world, maybe because it's the Linux default Shell .This shell is called *superset* of the Bourne shell, a set of add-ons and plug- ins. This means that the Bourne Again shell is compatible with the Bourne shell: commands that work in **sh**, also work in **bash**. However, the reverse is not always the case.

Korn Shell (ksh)

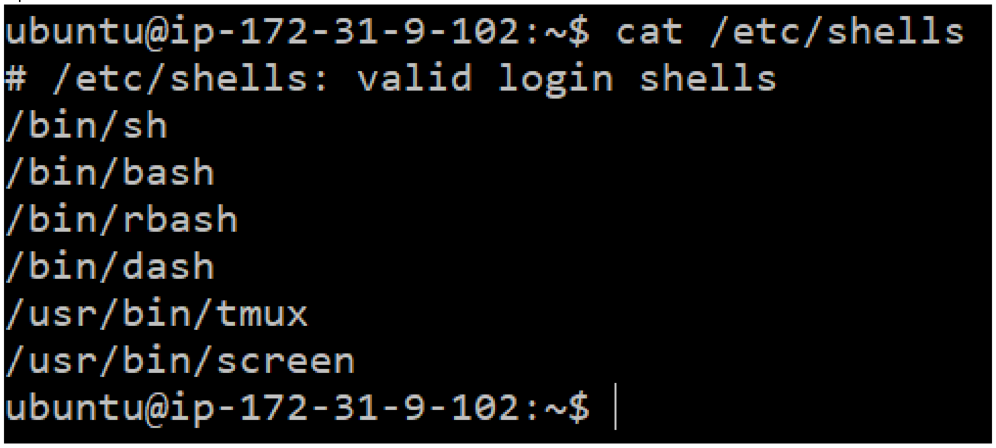
Developed by David Korn, also from Bell Labs, it's a superset derived from sh, which means it has all features that sh does and also a bunch of improvements. As it is 100% compatible with sh, it is taking over a lot of shell programers and users. Not to mention, it's the one we will base our course on.

C Shell (csh)

Developed by Bill Joy from Berkley University is the most common Shell on *BSD/Xenin environments. Commands structure is very similar to the C programming language. Its major fault was that it is not compatible with the sh Shell.

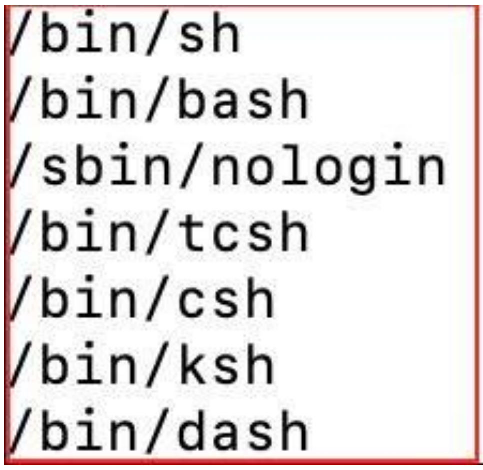
tcsh or TENEX C shell: a superset of the common C shell, enhancing user-friendliness and speed. That is why some also call it the Turbo C shell.

The file /etc/shells gives an overview of known shells on a Linux system:



```
ubuntu@ip-172-31-9-102:~$ cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/bash
/bin/rbash
/bin/dash
/usr/bin/tmux
/usr/bin/screen
ubuntu@ip-172-31-9-102:~$
```

Your default shell is set in the /etc/passwd file, like this line for user ***Test***.



```
/bin/sh
/bin/bash
/sbin/nologin
/bin/tcsh
/bin/csh
/bin/ksh
/bin/dash
```

Test:x:510:512:Test ID:/home/Test:/bin/bash

Comparison between Shells

Feature	Bourne	Bash	Korn	C	TC
Aliases	no	yes	yes	yes	yes
Advanced Pattern Matching	no	yes	yes	no	no
Command Line Editing	no	yes	yes	no	yes
Directory Stacks(pushd,popd)	no	yes	no	yes	yes
Filename Completion	no	yes	yes	yes	yes
Functions	yes	yes	yes	no	no
History	no	yes	yes	yes	yes
JobControl	no	yes	yes	yes	yes
Key Binding	no	yes	no	no	yes
Prompt Formatting	no	yes	no	no	yes
Spelling Correction	no	yes	no	no	yes

What is Shell Script?
Ans) In the simplest terms, a shell script is a file containing a series of commands. The shell reads this file and carries out the commands as though they have been entered directly on the command line.
Normally the shell scripts has the file extension .sh or .sc

To create a shell script, you use a *text editor*. There are many text editors available for your Linux system, both for the command line environment and the GUI environment. Here is a list of some common ones:

1) vi or vim and nano is the command line interface text editor.
2) gedit is the GUI text editor.

Following steps are required to write shell script:

Step 1) Use any editor like vi or gedit to write shell script.
Syntax: vi << Script file name >>

Example:
```
#vi hello.sh // Start vi editor
#gedit hello.sh
```

Step 2) After writing shell script set execute permission for your script as follows
Syntax: chmod << permission >> <<script file name >>

Examples:
```
# chmod +x script file name
# chmod 755 script file name
```

Note: This will set read, write and execute (7) permission for owner, for group and other permissions are read and execute only (5).

Step 3) Execute your script as
Syntax: bash script file name
sh script file name
./script file name

Example for writing script:

```
vi hello.sh // Start vi editor
#!/bin/sh
echo “Hello welcome to Shell Script!”
```
Give the permissions as follows
```
chmod +x hello.sh
```
Run the hello.sh script file as follows.
```
./hellow.sh
```

Output:
Hello welcome to Shell Script!”

First Shell script

File Name: hello.sh

```
#!/bin/bash
#Purpose : To display the message.
                                        . cat

echo “Hello Welcome to Shell script!”
```

The first line is called a shebang (#!/bin/sh) or a "bang" line. It is nothing but the absolute path to the Bash interpreter. It consists of a number sign and an exclamation point character (#!), followed by the full path to the interpreter such as /bin/bash.
If you do not specify an interpreter line, the default is usually the /bin/sh.

In order to execute the shell script make sure that you have execute permission. Give execute permissions as follows.

```
#chmod +x hello.sh (OR) chmod 111 hello.sh
```

There are different ways to run a shell script:
1) bash hello.sh
2) sh hello.sh
3) ./hello.sh
4) . hello.sh

Case Sensitivity
As you know Linux is case sensitive, the file names, variables, and arrays used in shell scripts are also case sensitive.

Check out this Example:

```
#!/bin/bash
string1=10
String1=20
echo "The value of string1 is $string1 "
echo "The value of String1 is $String1 "
```

File Naming conventions:
1) A file name can be a maximum of 255 characters
2) The name may contain alphabets, digits, dots and underscores
3) System commands or Linux reserve words can not be used for file names.

4) File system is case sensitive.
5) Some of valid filenames in Linux are

landmark.txt
mylandmark.txt
Anna.sh Mithun.sh
frida_08112013.sh
Henry05.sh

Comments

- Comments are used to escape from the code.
- This part of the code will be ignored by the program interpreter.
- Adding comments make things easy for the programmer, while editing the code in future.
- Single line comments can be do using #
- Multiline comments can be using HERE DOCUMENT feature as follows

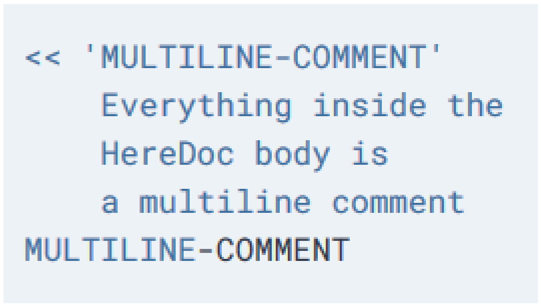
```
<<COMMENT1
   your comment 1
   comment 2
   blah
COMMENT1
```

File name : comments.sh

```
#! /bin/bash
# This is basic shell script.
echo "Hello, welcome to ShellScripting..." # It will display the message
```

File name : multiline_comments.sh

```
#!/bin/bash
echo "multiple lines comment start"
<<COMMENT1
This is a  multi line comment bash shell script.
Author: Landmark
Date: July 2020
Version: 1.0.0
feature
COMMENT1
echo "Multi-line comment start"
echo "Multi-line comment done"
```



```
<< 'MULTILINE-COMMENT'
    Everything inside the
    HereDoc body is
    a multiline comment
MULTILINE-COMMENT
```

Variables

There are two types of variables in Linux shell script.

Those are 1) System variables   and 2) User Define Variable (UDV)

2) User defined variables (UDV).

Naming conventions for variables:

1) Variable name must begin with alphanumeric character or underscore character (_). Following are the few examples for variable.

```
tools
training
```

By convention, environment variables (PAGER, EDITOR, ..) and internal shell variables (SHELL, BASH_VERSION, ..) are capitalized. All other variable names should be lower case.

Remember that variable names are case-sensitive; this convention avoids accidentally overriding environmental and internal variables.

Do not put spaces on either side of the equal sign when assigning value to variable. For example, the following is valid variable declaration:

```
no=10
```

However, any of the following variable declaration will result into an error such as *command not found*:

```
no  =10
no=  10
no  =  10
```

2) Variables names are case-sensitive, just like filenames.

```
no=10
No=11
NO=20
nO=2
```

All are different variable names, to display value 20 you've to use $NO variable:

```
echo "$no" # print 10 but not 20
echo "$No" # print 11 but not 20
echo "$nO" # print 2 but not 20
echo "$NO" # print 20
```

You can define a NULL variable as follows (NULL variable is variable which has no value at the time of definition):

```
vech=
vech=""
```

How to access the value of
UDV? Use a $ symbol to
access

Try to print its value by issuing the following command:

```
echo $vech
```

Do not use ?,* and other special characters, to name your variable.

```
?no=10 #invalid
out*put=/tmp/filename.txt #invalid
_GREP=/usr/bin/grep #valid
echo "$_GREP"
```

System defined Variables :

Created and maintained by Linux bash shell itself. This type of variable is defined in CAPITAL LETTERS.
There are many shell inbuilt variables which are used for administration and writing shell scripts.
To see all system variables, type the following command at a console / terminal:
env or printenv

Use echo command to display variable value as follows.
File name : system_variables.sh

```
#! /bin/bash
echo 'BASH='$BASH
echo 'BASH_VERSION='$BASH_VERSION
echo 'HOSTNAME=' $HOSTNAME
echo 'TERM='$TERM
echo 'SHELL='$SHELL
echo 'HISTSIZE='$HISTSIZE
echo 'SSH_CLIENT='$SSH_CLIENT
echo 'QTDIR='$QTDIR
echo 'QTINC='$QTINC
echo 'SSH_TTY='$SSH_TTY
echo 'RE_HOME='$JRE_HOME
echo 'USER='$USER
echo 'LS_COLORS='$LS_COLORS
echo 'TMOUT='$TMOUT
echo 'MAIL='$MAIL
echo 'ATH='$PATH
echo 'PWD='$PWD
echo 'JAVA_HOME='$JAVA_HOME
echo 'LANG='$LANG
echo 'SSH_ASKPASS='$SSH_ASKPASS
echo 'HISTCONTROL='$HISTCONTROL
echo 'SHLVL='$SHLVL
echo 'HOME='$HOME
echo 'LOGNAME='$LOGNAME
echo 'TLIB='$QTLIB
echo 'CVS_RSH='$CVS_RSH
echo 'SSH_CONNECTION='$SSH_CONNECTION
echo 'LESSOPEN='$LESSOPEN
echo '_BROKEN_FILENAMES='$G_BROKEN_FILENAMES
echo 'OLDPWD='$OLDPWD
```

Following are some system variables and those meanings

System Variable	Meaning	Value
BASH		/bin/bash
BASH_VERSION	To get bash version. Useful for controlling the shell script flow.	3.2.25(1)-release
COLUMNS		168
EUID	Display UserID.	0
GROUPS	To get GID info.	0
HOME	Give you user's home directory.	/root
HOSTNAME	Displays hostname.	ralbz2072.cloud.dst.ibm.com
HOSTTYPE	To get host architecture ie 32bit or 64 bit.	x86_64
HISTSIZE	Gives the size of number of Commands that can be accommodated in History file	1000
HISTFILE	To get the history file location	/root/.bash_history
LINES		45
LOGNAME		root
MACHTYPE		x86_64-redhat-linux-gnu
OSTYPE	Detect OS, such as gnu Linux, sun sol etc. Useful for controlling the shell script flow.	linux-gnu
PATH	Get path to all the binaries ie commands	/usr/kerberos/sbin: /usr/kerberos/bin: /usr/local/sbin: /usr/local/bin:/sbin: /bin:/usr/sbin: /usr/bin:/root/bin
PIPESTATUS		0
PPID		1313
PS1,PS2,PS3 and PS4		[\u@\h \W]\$ > +
PWD	Display's current working Directory.	/root
SHELL	Gives present shell	/bin/bash
TERM	Gives you terminal name.	xterm
TMOUT	Exit the shell, if no activity was there on terminal, useful for securing the server.	
USERNAME	User name who is currently login to this PC	

User defined Variables:

Created and maintained by user. This type of variable defined may use any valid variable name, but it is good practice to avoid all uppercase names as many are used by the shell.

```
Ex: name="Esia Legah“
      id=05211P
```

File name : user_defined_variables.sh

```
#! /bin/bash
trainingCourse=DevOps

echo 'Displaying the user defined the varibale (trainingCourse) value is: '$trainingCourse
```

Command Line arguments

During shell script execution, values passing through command prompt is called as command line arguments.

For example, while running a shell script, we can specify the command line arguments as “sh scriptfile.sh arg1 arg2 arg3”

While using command line arguments follow the below important points.

- We can specify n number of arguments, there is no limitation.
- Each argument is separated by space.

Following example describes the command line arguments.
File name: commandlineargs.sh

```
#!/bin/sh
#Number of arguments on the command line.
echo '$#:' $#
#Process number of the current process.
echo '$$:' $$
#Display the 3rd argument on the command line, from left to right.
echo '$3:' $3
#Display the name of the current shell or program.
echo '$0:' $0
#Display all the arguments on the command line using * symbol.
echo '$*:' $*
#Display all the arguments on the command line using @ symbol.
echo '$*:' $@
```

Run: `# sh commandlineargs.sh Github Bluemix UCD Jenkins Java Linux WebSphereApplicationServer`
Output:

```
$#: 4
```

```
$$: 16955
$3: WebSphereApplicationServer
$0: commandlineargs.sh
$*: Java Linux WebSphereApplicationServer Android
$@: Java Linux WebSphereApplicationServer Android
```

Difference between $* and $@

The collection of arguments in $* is treated as one text string, whereas the collection of arguments in $@ is treated as separate strings.

Escape character

When using echo -e, you can use the following special characters:

\\	backslash
\a	alert (BEL)
\b	backspace
\c	suppress trailing newline
\f	form feed
\n	new line
\r	carriage return
\t	horizontal tab
\v	vertical tab

Example:

File Name: escape_characters.sh

```
echo -e "Displaying backslash : \\ "

echo -e "Making sound(alert) : \a "

echo -e "Displaying backspace : \b"

echo -e "Displaying single quote : \' "

echo -e "Displaying double quote : \" "

echo -e "New Line: \n "
```

Strings

Strings are scalar and there is no limit to the size of a string.
Any characters, symbols, or words can be used to make up your string.

How to define a string?
We use single or double quotations to define a string.
"**Double Quotes**" - Anything enclose in double quotes removed meaning of that characters (except \ and $).
'**Single quotes**' - Enclosed in single quotes remains unchanged.
`Back quote` - To execute command

Example:

FileName: quotes.sh

```
#!/bin/bash
single='Single quoted'
double="Double quoted"
echo $single
echo $double
```

Save the above example script as quotes.sh

```
chmod 755 quotes.sh
./quotes.sh
```

String Formatting

Character	Description
-n	Do not output the trailing new line.
-e	Enable interpretation of the following backslash escaped characters in the strings:
\a	alert (bell)
\b	backspace
\c	suppress trailing new line
\n	new line
\r	carriage return
\t	horizontal tab
\\	backslash

Example:

```
#!/bin/bash
echo -e "Hello \t Welcome to Linux Shell Scripting"
```

Arithmetic Operations

We use the keyword " expr " to perform arithmetic operations.

Syntax:
expr op1 math-operator op2 **ech**

Example:

FileName: arithmetic_operations.sh

```
$ expr 3 + 2
$ expr 3 - 2
$ expr 10 / 2
$ expr 20 % 3
$ expr 10 \* 3
$ echo `expr 3 + 2`
```

In shell scripting we will be using the echo command to print the result of the arithmetic operations.

Note:
If you use echo command, before expr keyword we use ` (back quote) sign.
Here both double and single quote will not give you the desired result.
Try the following commands:

```
$ echo "expr 3 + 2" # Result expr 3 + 2
$ echo 'expr 3 + 2' # Result expr 3 + 2
$ echo `expr 3 + 2` # Result 5
```

From the above example, we see that if we use double or single quote, the echo command directly prints " expr 3 + 2 ".
To get the result of this expression, we need to use the back quote.

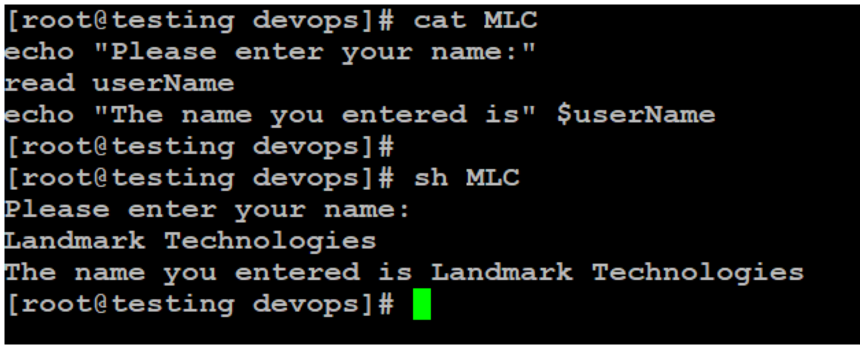
User Interaction using read command

In some cases the script needs to interact with the user and accept inputs.
In shell scripts we use the read statement to take input from the user.

read : read command is used to get the input from the user (Making scripts interactive).

Following shell script demonstrates the take the input from user and display that back to user.

File Name: readName.sh



```
[root@testing devops]# cat MLC
echo "Please enter your name:"
read userName
echo "The name you entered is" $userName
[root@testing devops]#
[root@testing devops]# sh MLC
Please enter your name:
Landmark Technologies
The name you entered is Landmark Technologies
[root@testing devops]#
```

echo The name you entered is $userName

Run the above script as follows:
#./readName.sh

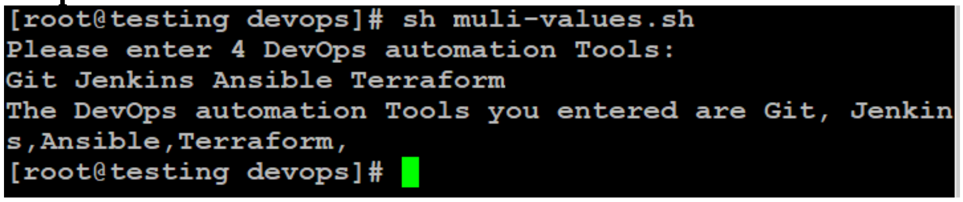
Output:
Please enter your name: Anna Lanmark
The name you entered is Anna Landmark

File Name: readMultipleValues.sh
#!/bin/bash
#Author: Mylandmark .
#Date: 7th June 2020.

echo "Please enter 4 DevOps automation Tools:"
read devOpsTool1 devOpsTool2 devOpsTool3 devOpsTool4

echo The DevOps Tools you entered are $devOpsTool1 , $devOpsTool2 , $devOpsTool3 , $devOpsTool4

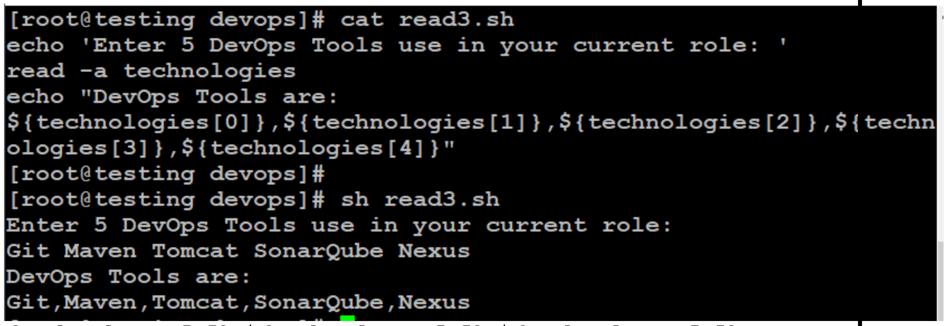
Output:



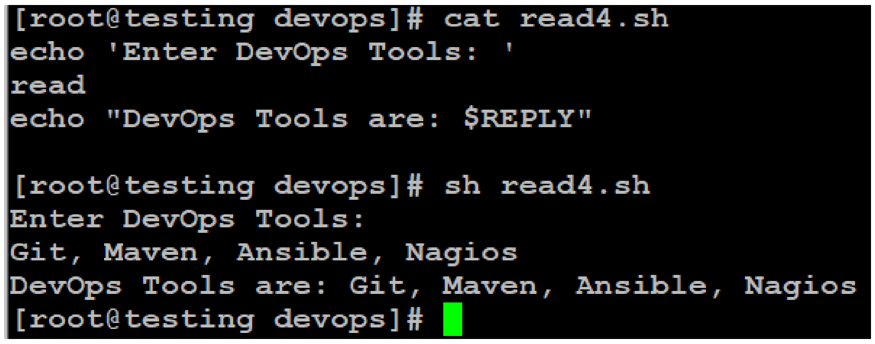
```
[root@testing devops]# sh muli-values.sh
Please enter 4 DevOps automation Tools:
Git Jenkins Ansible Terraform
The DevOps automation Tools you entered are Git, Jenkin
s,Ansible,Terraform,
[root@testing devops]#
```

read command with different options.
File Name: readWithOptions2.sh
#Author: Landmark Technologies. .
#Date: 7th June 2020.

echo 'Enter DevOps Tools: '
read -a technologies
echo "DevOps Tools are: ${technologies[0]},${technologies[1]},${technologies[2]},${technologies[3]},${technologies[4]}"



```
[root@testing devops]# cat read3.sh
echo 'Enter 5 DevOps Tools use in your current role: '
read -a technologies
echo "DevOps Tools are:
${technologies[0]},${technologies[1]},${technologies[2]},${techn
ologies[3]},${technologies[4]}"
[root@testing devops]#
[root@testing devops]# sh read3.sh
Enter 5 DevOps Tools use in your current role:
Git Maven Tomcat SonarQube Nexus
DevOps Tools are:
Git,Maven,Tomcat,SonarQube,Nexus
```



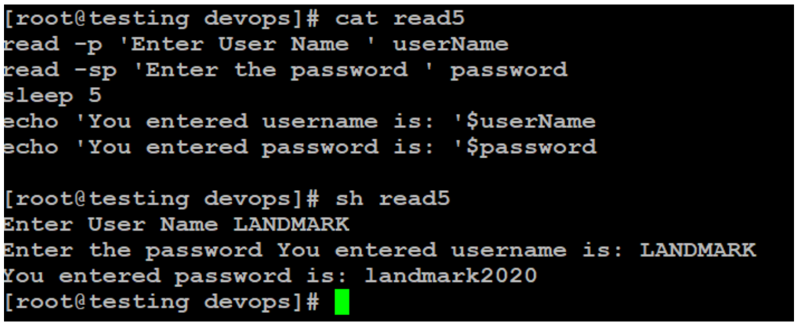
```
[root@testing devops]# cat read4.sh
echo 'Enter DevOps Tools: '
read
echo "DevOps Tools are: $REPLY"

[root@testing devops]# sh read4.sh
Enter DevOps Tools:
Git, Maven, Ansible, Nagios
DevOps Tools are: Git, Maven, Ansible, Nagios
[root@testing devops]#
```

File Name: readWithOptions1.sh
#Author: Landmark Technologies.
#Date: 7th June 2020.

read -p 'Enter User Name ' userName
read -sp 'Enter the password ' password

echo '   '
echo 'You entered username is: '$userName
echo 'You entered password is: '$password



```
[root@testing devops]# cat read5
read -p 'Enter User Name ' userName
read -sp 'Enter the password ' password
sleep 5
echo 'You entered username is: '$userName
echo 'You entered password is: '$password

[root@testing devops]# sh read5
Enter User Name LANDMARK
Enter the password You entered username is: LANDMARK
You entered password is: landmark2020
[root@testing devops]#
```

Debugging

For debugging the shell we can use –v, -x and –n options. General syntax is as follows.

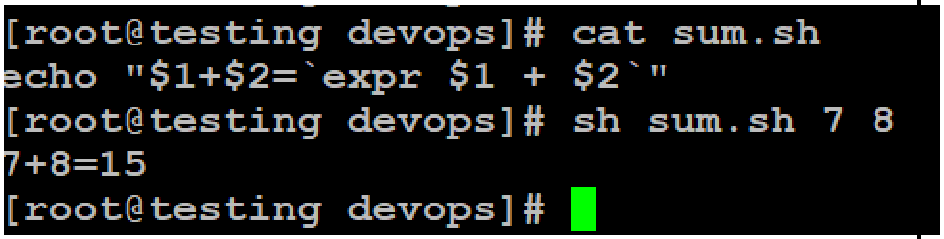
#sh << options >> << script name >>
(OR)
#bash << options >> << script name >>

-x : Display commands and their arguments as they are executed.
-v : Display shell input lines as they are read.
-n : Read commands but do not execute them. This may be used to check a shell script for syntax errors
Following shell script will accept the numbers from command prompt and, it will display the sum of those two numbers.

File Name: debug.sh

#! /bin/sh
#Purpose: Addition of two numbers
#Author: Landmark Technologies
#Date: June 24th 2020

sum=`expr $1 + $2`
echo $sum



```
[root@testing devops]# cat sum.sh
echo "$1+$2=`expr $1 + $2`"
[root@testing devops]# sh sum.sh 7 8
7+8=15
[root@testing devops]#
```

echo "$1+$2=`expr $1 + $2`"

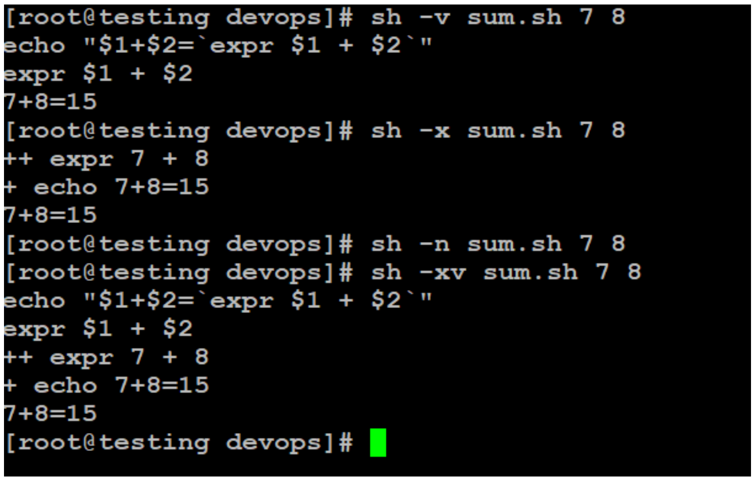
Run the above script as follows.

#sh -v sum.sh 7 8

#sh -x sum.sh 7 8

#sh -n sum.sh 7 8

#sh -xv sum.sh 7 8



```
[root@testing devops]# sh -v sum.sh 7 8
echo "$1+$2=`expr $1 + $2`"
expr $1 + $2
7+8=15
[root@testing devops]# sh -x sum.sh 7 8
++ expr 7 + 8
+ echo 7+8=15
7+8=15
[root@testing devops]# sh -n sum.sh 7 8
[root@testing devops]# sh -xv sum.sh 7 8
echo "$1+$2=`expr $1 + $2`"
expr $1 + $2
++ expr 7 + 8
+ echo 7+8=15
7+8=15
[root@testing devops]#
```

Bash shell offers debugging options which can be turned on or off using set command as follows.

```
#!/bin/sh
#Purpose: Addition of two numbers
#Author: Landmark Technologies
#Date: June 24th 2020

### Turn on debug mode ###

set -x
//set -v
//set -n

sum=`expr $1 + $2`
echo $sum
```

Run the above script as follows.

```
#sh debug.sh 1 2
```

Input - Output redirection in Shell Scripts

Using shell scripts, we can redirect - the output of a command to a file
or
- redirect an output file as an input to other commands.

In Shell script there are mainly 3 types of redirect symbols as follows.

1. > Redirect standard output

Example:
```
ls > ls-file.txt
```

The above command will redirect the output of the " ls " to the file " ls-file ".
If the file " ls-file " already exist, it will be overwritten. Here you will loose the existing data.

2. >> Append standard output

Example:
```
date >> ls-file.txt
```

The output of the date command will be appended to the file " ls-file ".
In this case you will not loose any data. The new data gets added to the end of the file.

3. < Redirect standard input

Example:
cat < ls-file.txt

This redirection symbol takes input from a file.
In the above example the cat command takes the input from the file " ls-file " and displays the "ls-file" content.

Some of the forms of redirection for the Bourne shell family are:

Character	Action
2>	Redirect standard error
2>&1	Redirect standard error to standard output

Examples:

```
#date > output.txt
#whoami >> output.txt
#(pwd; whoami; date) > output.txt
# pwd; whoami; date > output1.txt ---> It will redirect only date to output1.txt file.
```

We can suppress redirected output and/or errors by sending it to the null device, /dev/null. The example shows redirection of both output and errors:

```
#who >& /dev/null

# sh debug.sh > myout 2> myerror
```

Control commands

if control statement:

Syntax:

```
if condition
then
      Display commands list if condition is true.
else
      Display commands list if condition is false.
fi
```

Note: if and then must be separated, either with a << new line >> or a semicolon (;).
Termination of the if statement is **fi**.

Example:

Following example demonstrates the find biggest number.

File Name: FindBiggestNumber.sh

```
if [ $# -ne 3 ]
   then
      echo "$0: number1 number2 number3 are not given" >&2
      exit 1
   fi

if [ $n1 -gt $n2 ] && [ $n1 -gt $n3 ]
then
      echo "$n1 is the biggest t number"
   elif [ $n2 -gt $n1 ] && [ $n2 -gt $n3 ]
   then
      echo "$n2 is the biggest t number"
   elif [ $n3 -gt $n1 ] && [ $n3 -gt $n2 ]
   then
      echo "$n3 is the biggest number"
   elif [ $1 -eq $2 ] && [ $1 -eq $3 ] && [ $2 -eq $3 ]
   then
      echo "All the three numbers are equal"
   else
      echo "I can not figure out which number is bigger"
   fi
```

Run:

```
sh findbiggestNumber.sh 1 2 3
```

Output:

```
3 is the biggest number
```

File Name: FileExists.sh

```
#! /bin/bash

echo -e "Enter the name of the file: \c"
read file_name

if [ -f $file_name ]
 then
        if [ -w $file_name ]
        then
                echo "Type something, To Quit type Ctrl +d"
                cat >> $file_name
        else
                echo "The file do not have write permissions"
        fi
 else


         echo "$file_name not exists"
fi
```

----

for loop

Syntax:

```
        for (condition )
        do
              execute here all command/script until the condition is
              not satisfied.(And repeat all statement between do and done)

        done
```

Example:

FIleName: for_loop.sh

```
echo "Can you see the following:"

for (( i=1; i<=5; i++ ))
do
  echo $i
  echo ""
done
```

Output:

```
Can you see the following:
1

2

3

4

5
```

```
while [ condition ]
           do
                 command1
                 command2
                 command3
                 ..
                 ....
            done
```

Example:

```
i=5
while test $i != 0
do

       echo "$i"
       echo " "
        i=`expr $i - 1`
```

Output:

```
5

4

3

2

1
```

- -

switch case:

The case statement is good alternative to multilevel if-then-else-fi statement. It enables you to match several values against one variable. It’s easier to read and write.

Syntax:

```
           case $variable-name in
                pattern1) command
                                ...
                                ..
                                 command;;
                pattern2)  command
                                ...
                                ..
                                  command;;
```

```
            patternN)  command
                       ...
                       ..
                       command;;
            *)         command
                       ...
                       ..
                       command;;
        esac
```

```
Filename: switch_case.sh
#!/bin/sh

echo "Enter a number between 1 and 10. "
read NUM

case $NUM in
        1) echo "You entered is one" ;;
        2) echo "You entered is two" ;;
        3) echo "You entered is three" ;;
        4) echo "You entered is four" ;;
        5) echo "You entered is five" ;;
        6) echo "You entered is six" ;;
        7) echo "You entered is seven" ;;
        8) echo "You entered is eight" ;;
        9) echo "You entered is nine" ;;
        10) echo "You entered is ten" ;;
        *) echo "INVALID NUMBER!" ;;
esac
```

Functions:

When your scripts start to become very large, you may tend to notice that you are repeating code more often in your scripts. You have the ability to create functions inside of your script to help with code reuse. Writing the same code in multiple sections of your script can lead to severe maintenance problems. When you fix a bug in a section of code you need to be sure that all sections of code that are repeated will also have those fixes. A function is a block of code that can be called from other parts of your script. It can have parameters passed to it as if it were a separate script itself. As an example, we will create a function called log it, which will take two parameters, a level and a message. The level will be a number between 1 and 3 that will indicate the severity of the log message. The level of messages that you want to view will be passed in the command line of the script.

Syntax:

```
function_name() {

Commands to be execute here...

}
```

Example:

Write a function to display Welcome to Shell script! message.

FileName: FunctionExample.sh

```
#!/bin/bash


greetfn(){

echo "Welcome to Shell script! "

}

echo "Calling greetfn() ! "

greetfn
```

FileName: addFunction.sh

```
#!/bin/bash


addfn(){

echo `expr $1 + $2`

}
echo "Calling greetfn() ! "

addfn 1 2
```

Pipes:

A pipe is nothing but a temporary storage place where the output of one command is stored and then passed as the input for second command. Pipes are used to run more than two commands (Multiple commands) from same command line.

Pipelines connect the standard output of one command directly to the standard input of another. The pipe symbol (|) is used between the commands:

Example:
Creating a new file " friends ".

```
cat > friends
esia
eric
mercy
Evolie
```
Press CTRL + D to save.

Now issue the following command:
```
sort friends | tr "[a-z]" "[A-Z]" > FRIENDS
cat FRIENDS
```

Result:

```
esia
eric
mercy
Evolie
```

The above command has sorted the file " friends " and using pipe the output was translated to upper-case letters and then redirected its output to the file " FRIENDS ".

Filters

A filter is a program that accepts input, transforms it or does something useful with it and outputs the transformed data. Some important filter commands are awk, tee, grep, sed, spell, and wc.

Example:

```
ls -l | grep -a "^d" | tee dir.lst | wc -l
```

what the above command does:
The ls command lists all the files and directories. The grep command takes only the directories. Using tee command, we write the result to the file " dir.list " and the wc -l prints the total number of lines in that file.

We have used 3 filters in the above example:
The grep, tee and wc

.

Process

What is Process?
Process is kind of program or task in execution.
Each command, program or a script in linux is a process.

Managing a process
You might have seen some process taking long time to execute. You can run such process in the background.
Example:

```
ls / -R | wc -l
```

The above command can be used to find the total count of the files on your system. This takes a long a time, so to get the command prompt and to start an other process, you can run the above command in background

Running a process in background

```
ls / -R | wc -l &
```

The ampersand (&) at the end of command tells the shell to run the command in background. You will a number printed on your screen. It is the process Id.
So for each process you will have a process Id.

Some of the important commands assosiated with processes.

ps -> display currently running process
kill {PID} -> to kill a particular process
killall {Process-name} -> to kill all the process having the same name.
ps -ag -> to get information about all running process
kill 0 -> to kill all process except your shell
linux-command & -> start a process in background
ps aux -> to get all the details regarding the running process.

ps ax | grep process-name -> to check a particular process.
top -> to see currently running processes and there memory usage.
pstree -> to see currently running processes in a tree structure.