# OS Lab 3
# Shell Scripting

## What is “Shell”?

- The “Shell” is simply *another program* which provides a basic human-OS interface.
  - It is a command interpreter
    - Built on top of the kernel
    - Enables users to run services provided by the UNIX/Linux OS
  - In its simplest form, a series of commands in a file is a shell program that saves having to retype commands to perform common tasks.
- How to know what shell you use
  - `echo $SHELL`

## What is “Shell”?

- ❑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
- ❑Current location of the running shell/bash? Type:
  - ❑Which bash -> e.g, /bin/bash

## Linux Shells

- ❑sh Bourne Shell (Original Shell)
- ❑bash Bourne Again Shell (*GNU Improved Bourne Shell*)
- ❑csh C-Shell (C-like Syntax)(*Bill Joy of Univ. of California*)
- ❑ksh Korn-Shell (Bourne+some C-shell)(*David Korn of AT&T*)
- ❑tcsh  Turbo C-Shell  (More User Friendly C-Shell).
- ❑To check shell:
  - ❑$ `echo  $SHELL` (shell is a pre-defined variable)

## Pipes

- ❑ Connect processes using pipe „|‟ operator
- ❑ Processes connected by pipes can run simultaneously and are automatically scheduled as data flows between them
- ❑Using „sort‟ command to sort output from “ls‟ command

```
# ls | sort
```

## What is shell script?

- A **shell script** is a script written for the shell
- 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.
- All the scripts would have the **.sh** extension

## What is shell script?

- 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*
  - This is the first line of the script above. The hash exclamation mark ( #! ) character sequence is referred to as the Shebang. Following it is the path to the interpreter (or program) that should be used to run (or interpret) the rest of the lines in the text file. (For Bash scripts it will be the path to Bash, but there are many other types of scripts and they each have their own interpreter.)

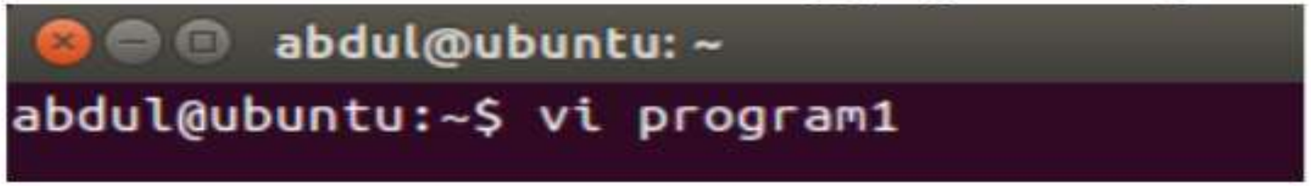
## How to write shell scripts?

- ❑Use any editor (kwrite, kate, gedit, vi etc) to write shell script
- ❑Set the execute permission for your script
  - ❑ through graphical interface
  - ❑chmod +x script-name
  - ❑chmod 755 script-name

Execute your script as:

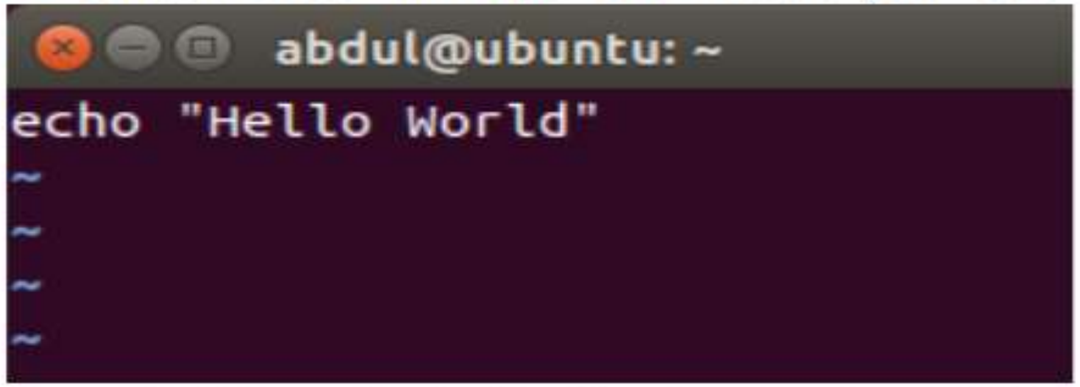
./script-name

Go to the terminal and create a file e.g. program1 using vi editor as



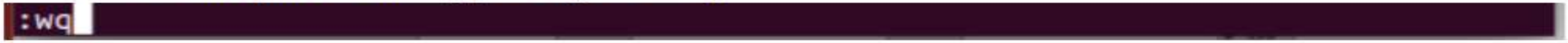
```
abdul@ubuntu: ~
abdul@ubuntu:~$ vi program1
```

Vi console editor will open the file and you have to write hello world string message as



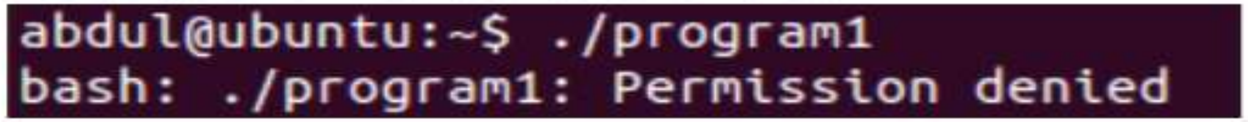
```
abdul@ubuntu: ~
echo "Hello World"
~
~
~
~
```

Save the file and quit vi editor by pressing ESC :wq



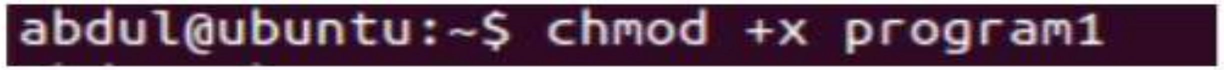
```
:wq
```

Now execute the script by writing ./scriptname , before execution, you have to assign execute permission to the script otherwise it gives an error as



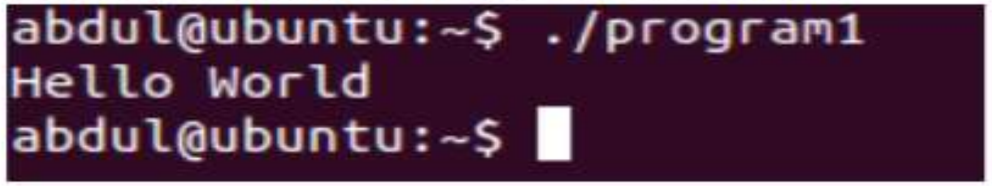
```
abdul@ubuntu:~$ ./program1
bash: ./program1: Permission denied
```

Assigning execute permission write chmod +x scriptname as



```
abdul@ubuntu:~$ chmod +x program1
```

Now your script is ready for execution.



```
abdul@ubuntu:~$ ./program1
Hello World
abdul@ubuntu:~$ 
```

## Commenting

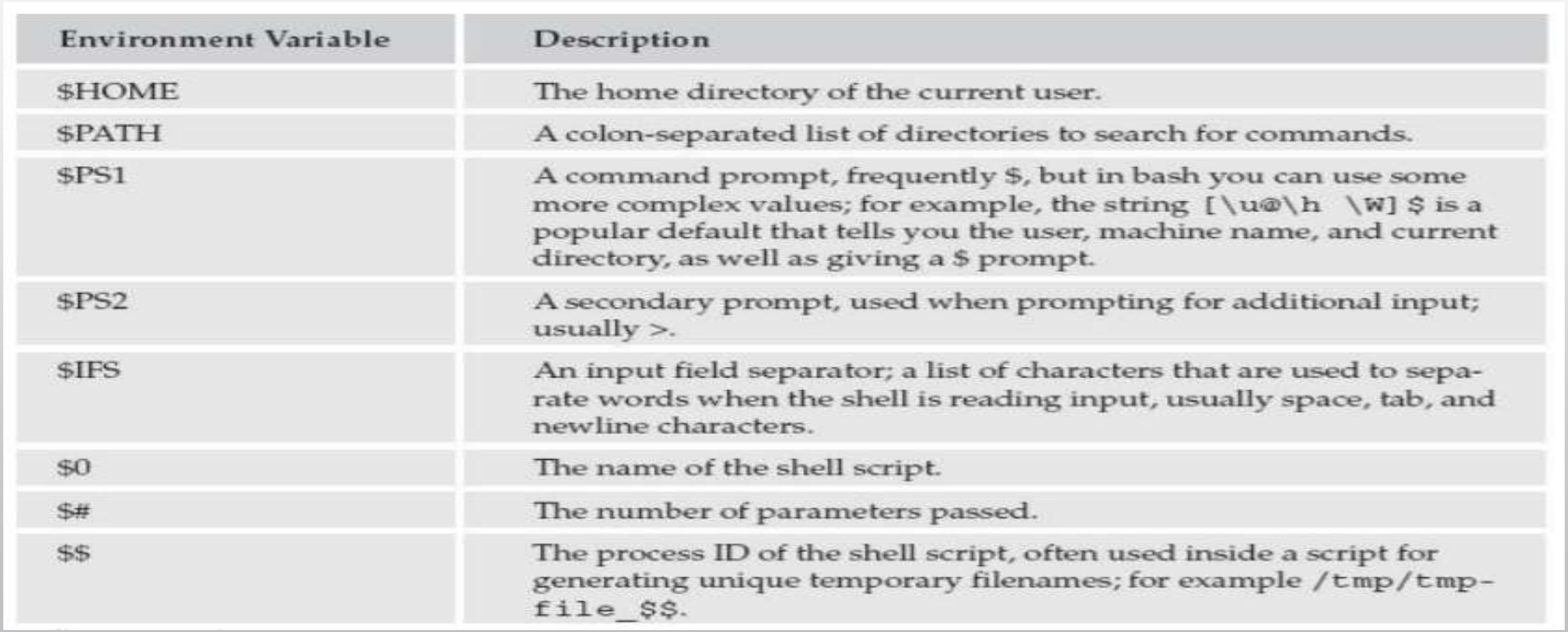
- ❑Lines starting with # are comments except the very first line where #! indicates the location of the shell that will be run to execute the script.
- ❑On any line characters following an unquoted # are considered to be comments and ignored.
- ❑Comments are used to;
  - ◦ Identify who wrote it and when
  - ◦ Identify input variables
  - ◦ Make code easy to read
  - ◦ Explain complex code sections
  - ◦ Version control tracking
  - ◦ Record modifications

## Variables in Shell

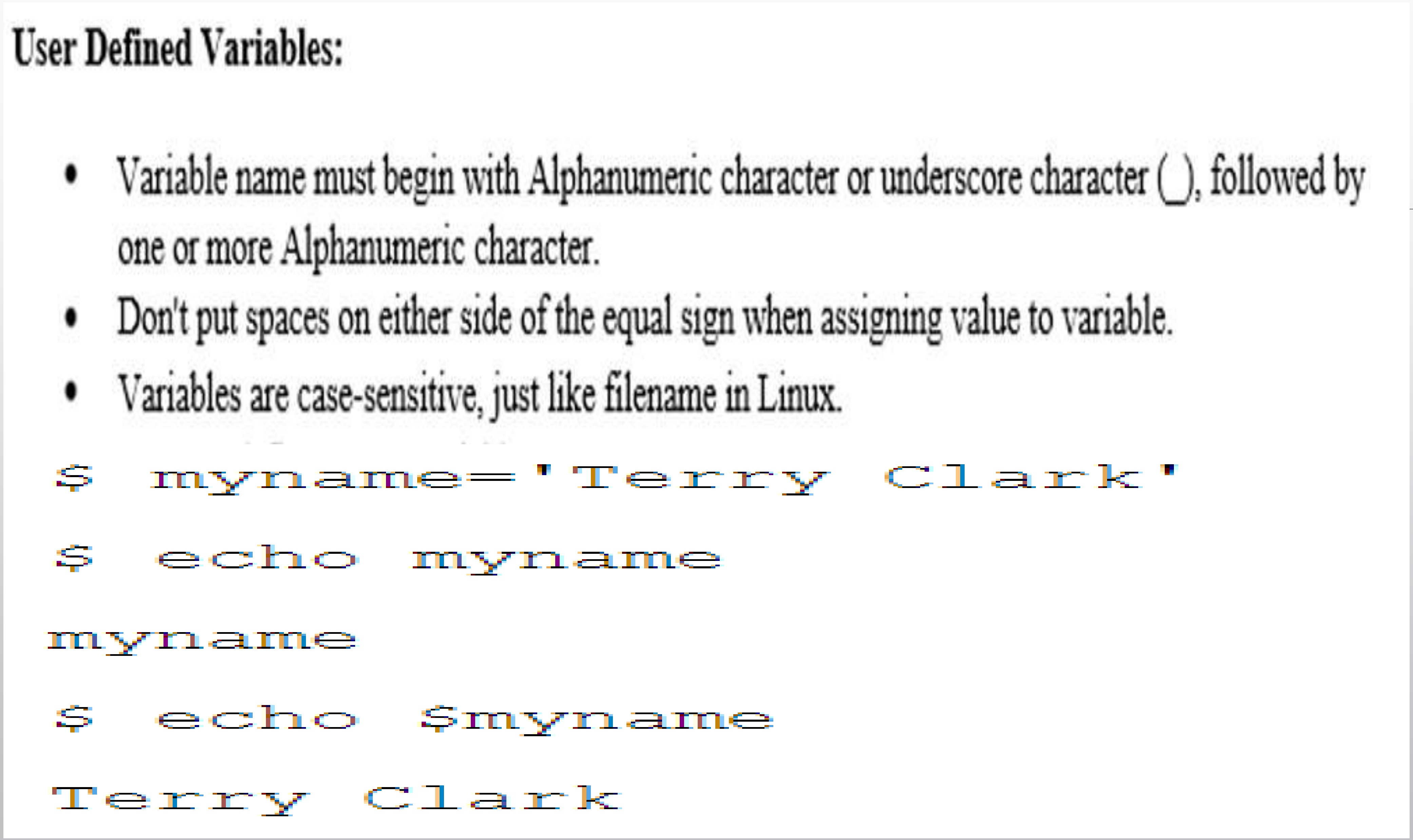
In Linux shell there are three types of variables

- ❑System or Environment variables
- ❑ User defined variables
- ❑Parametric variables

```
numan@numan:~$  printenv
```



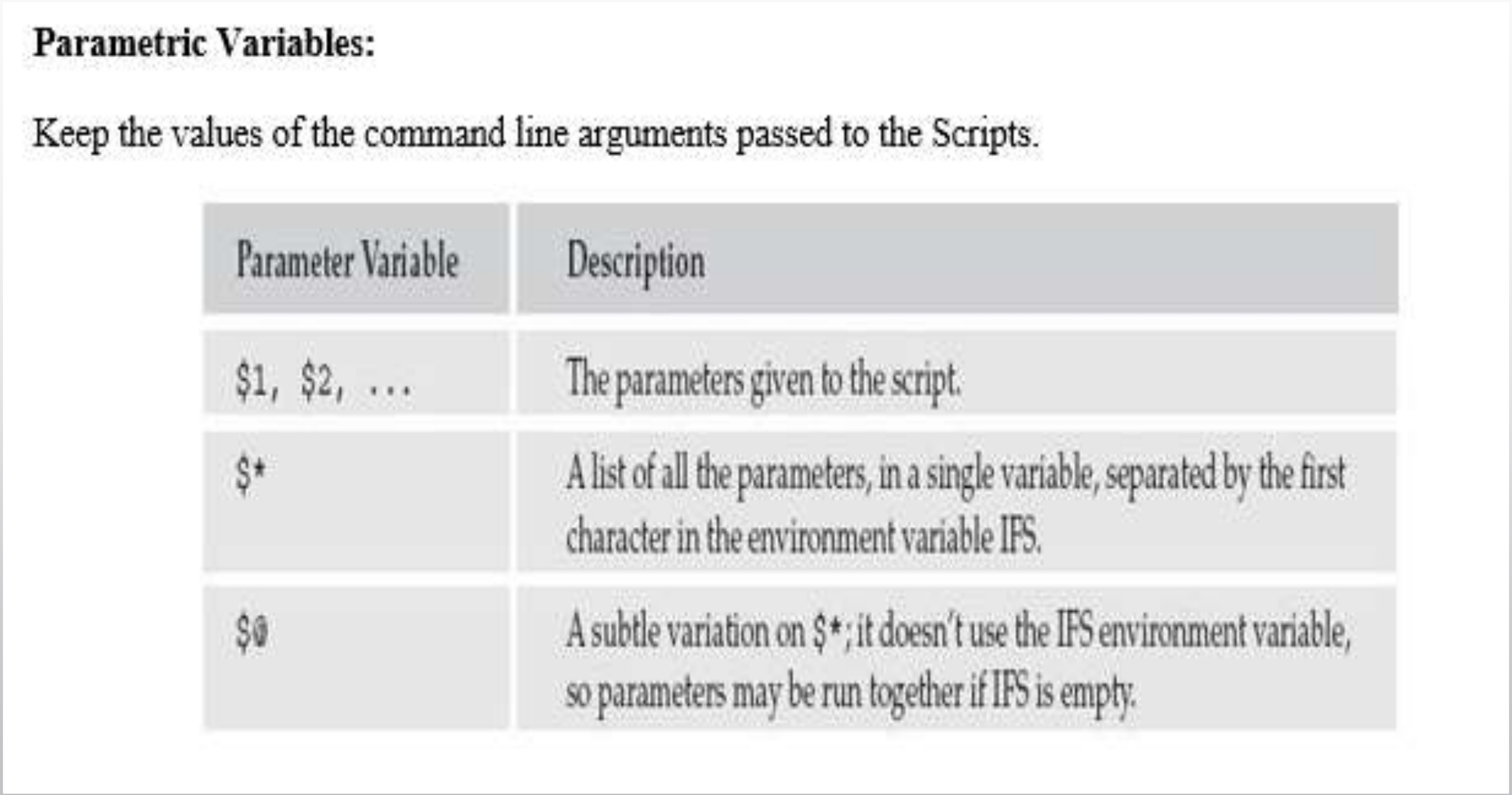
| Environment Variable | Description |
|---|---|
| $HOME | The home directory of the current user. |
| $PATH | A colon-separated list of directories to search for commands. |
| $PS1 | A command prompt, frequently $, but in bash you can use some more complex values; for example, the string `[\u@\h \W] $` is a popular default that tells you the user, machine name, and current directory, as well as giving a $ prompt. |
| $PS2 | A secondary prompt, used when prompting for additional input; usually >. |
| $IFS | An input field separator; a list of characters that are used to separate words when the shell is reading input, usually space, tab, and newline characters. |
| $0 | The name of the shell script. |
| $# | The number of parameters passed. |
| $$ | The process ID of the shell script, often used inside a script for generating unique temporary filenames; for example `/tmp/tmp-file_$$`. |



## User Defined Variables:

- Variable name must begin with Alphanumeric character or underscore character (_), followed by one or more Alphanumeric character.
- Don't put spaces on either side of the equal sign when assigning value to variable.
- Variables are case-sensitive, just like filename in Linux.

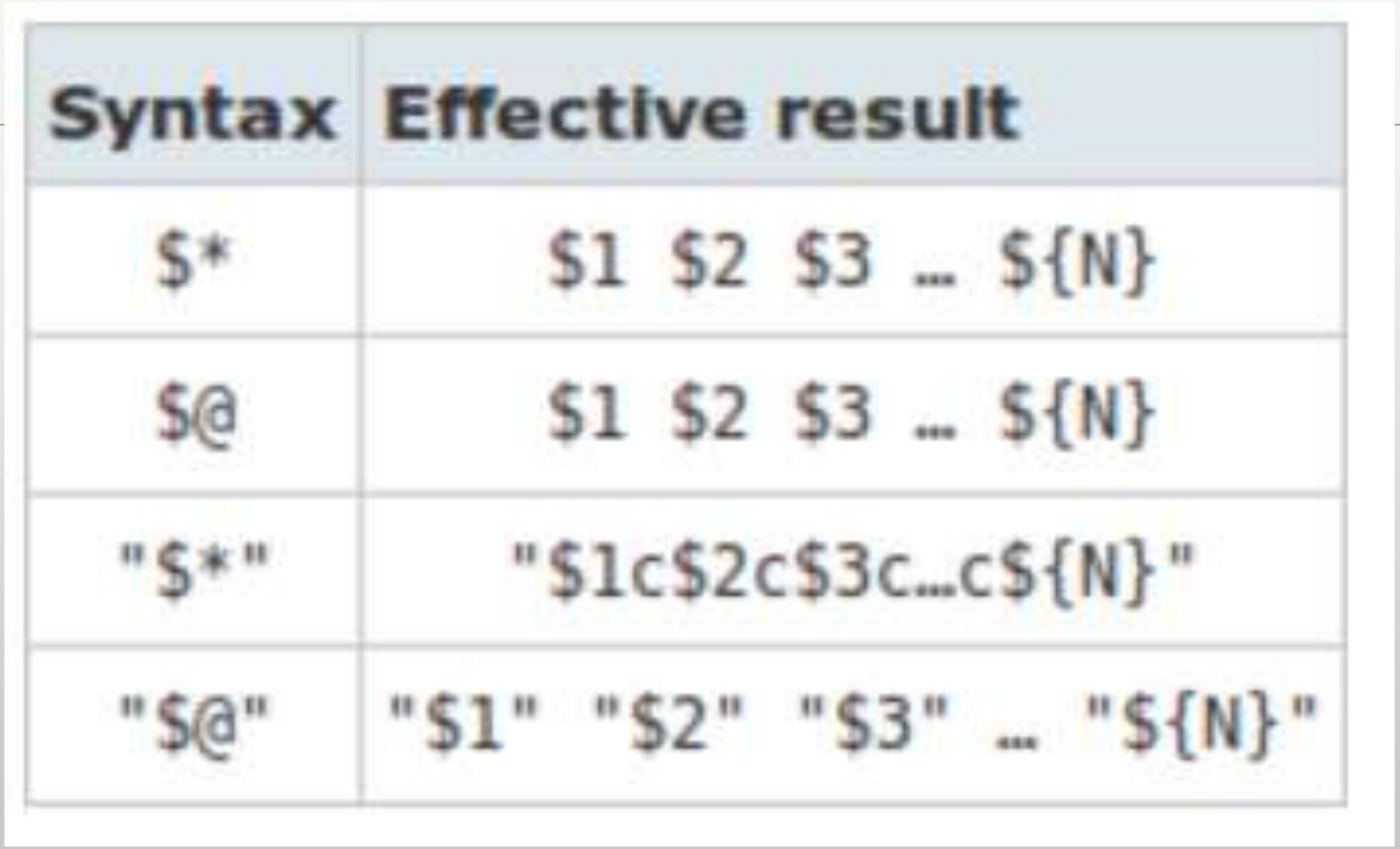
```
$ myname='Terry Clark'
$ echo myname
myname
$ echo $myname
Terry Clark
```



**Parametric Variables:**

Keep the values of the command line arguments passed to the Scripts.

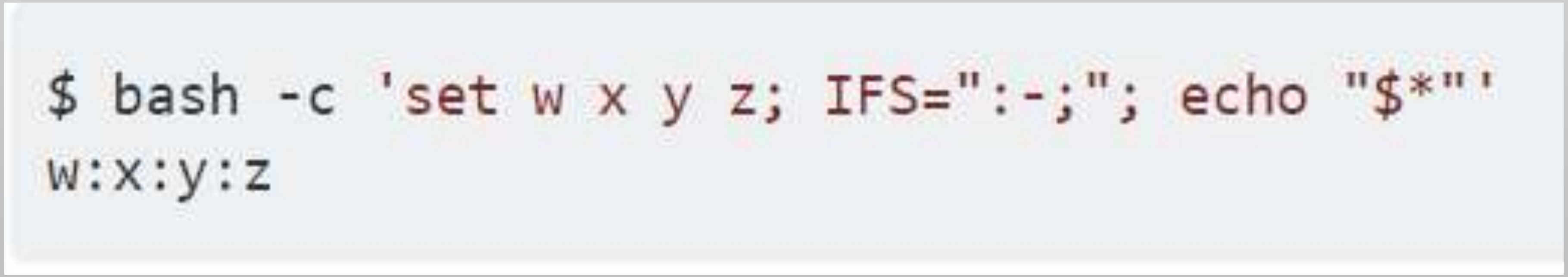
| Parameter Variable | Description |
|---|---|
| `$1, $2, ...` | The parameters given to the script. |
| `$*` | A list of all the parameters, in a single variable, separated by the first character in the environment variable IFS. |
| `$@` | A subtle variation on `$*`; it doesn't use the IFS environment variable, so parameters may be run together if IFS is empty. |



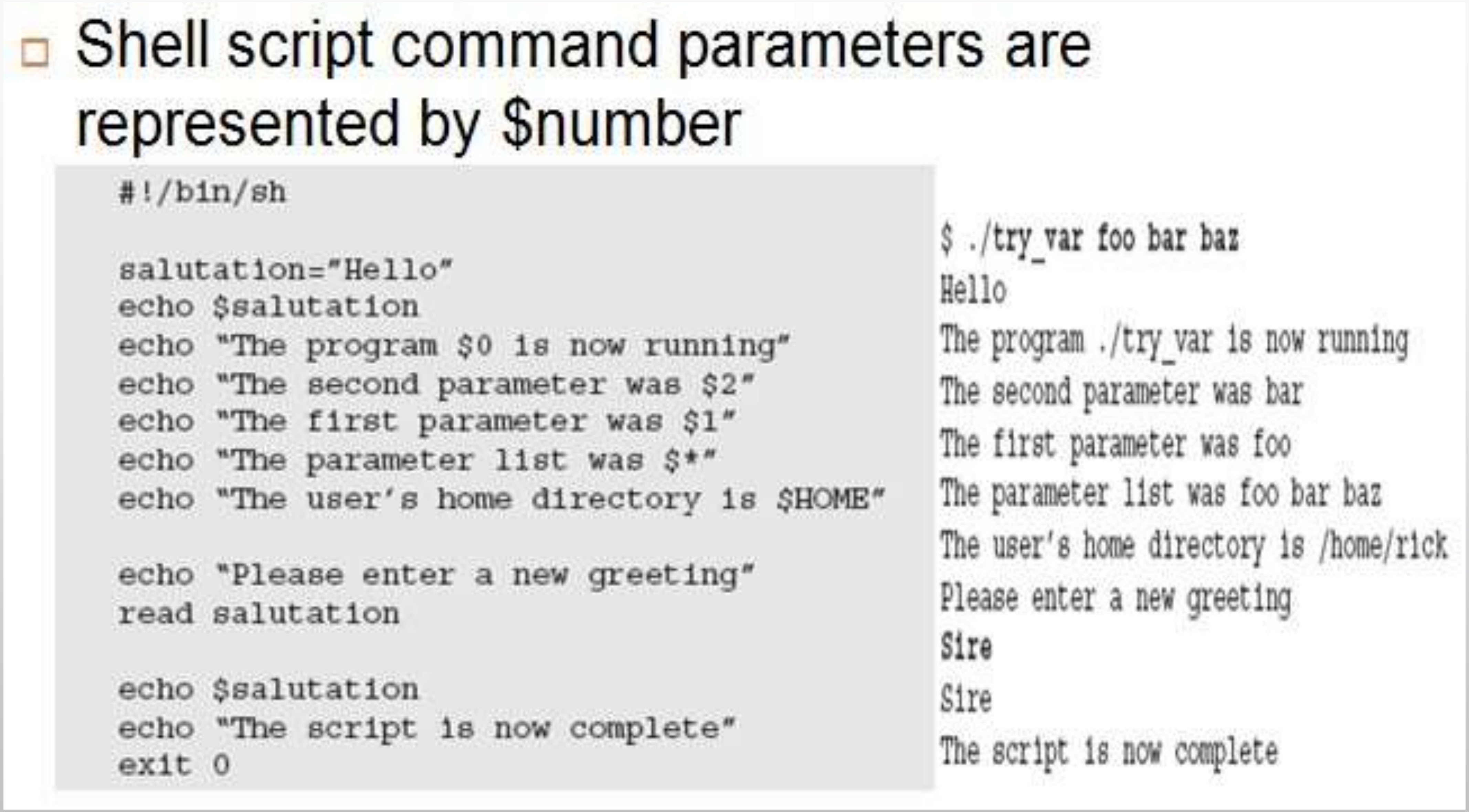
| Syntax | Effective result |
|---|---|
| $* | $1 $2 $3 … ${N} |
| $@ | $1 $2 $3 … ${N} |
| "$*" | "$1c$2c$3c…c${N}" |
| "$@" | "$1" "$2" "$3" … "${N}" |

## IFS Environment

- ❑For many command line interpreters (“shell”) of Unix/Linux operating systems, the internal field separator (abbreviated **IFS**) refers to a **variable** which **defines** the character or characters used to separate a pattern into tokens for some operations.
- ❑**IFS** typically includes
  - ❑Space
  - ❑Tab
  - ❑newline



```
$ bash -c 'set w x y z; IFS=":-;"; echo "$*"'
w:x:y:z
```



- Shell script command parameters are represented by $number

```
#!/bin/sh

salutation=”Hello”
echo $salutation
echo “The program $0 is now running”
echo “The second parameter was $2”
echo “The first parameter was $1”
echo “The parameter list was $*”
echo “The user’s home directory is $HOME”

echo “Please enter a new greeting”
read salutation

echo $salutation
echo “The script is now complete”
exit 0
```

```
$ ./try_var foo bar baz
Hello
The program ./try_var is now running
The second parameter was bar
The first parameter was foo
The parameter list was foo bar baz
The user’s home directory is /home/rick
Please enter a new greeting
Sire
Sire
The script is now complete
```

## A Shell Script Example

```
#!/bin/bash
# use ‘#” to add comments in shell script
#Author: Numan
#shell scripting practice problem
#script follows here:
pwd
Ls
```

## Shell Scripting

Tell Linux that the script file is executable

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

Execute the shell-script

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

## My First Shell Script

```
#! /bin/bash
# The first example of a shell script
directory=`pwd`
 echo Hello World!
 echo The date today is `date`
 echo The current directory is $directory
```

```
$ chmod +x myfirstscript.sh
$ ./myfirstscript.sh
  Hello World!
  The date today is Mon Mar 8 15:20:09 EST 2010
  The current directory is /netscr/shubin/test
```

## User Input

As shown on the hello script input from the standard input location is done via the read command.

Example

```
echo "Please enter three filenames:"
read  filea fileb filec
echo "These files are used:$filea  $fileb  $filec"
```

Each read statement reads an entire line. In the above example if there are less than 3 items in the response the trailing(losing) variables will be set to blank ' '.

Three items are separated by one space.

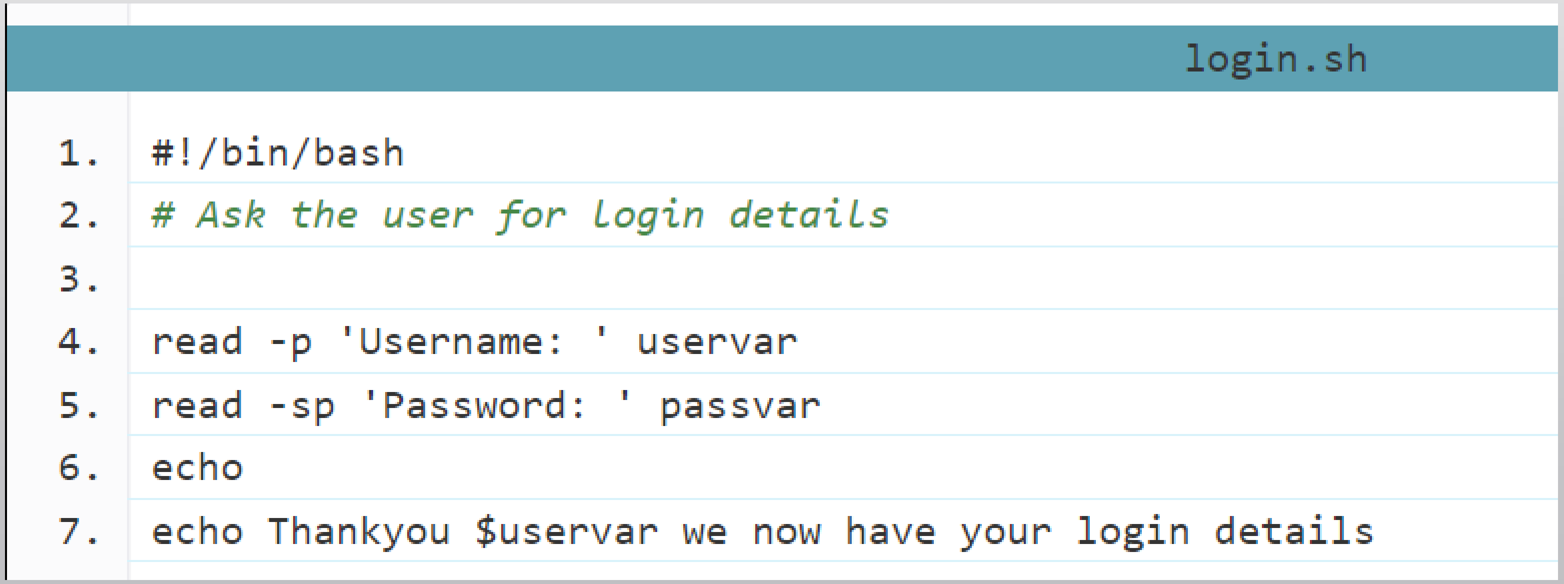
## Hello script

The following script asks the user to enter his name and displays a personalised hello.

```
#!/bin/sh
  echo "Who am I talking to?"
  read user_name
  echo "Hello $user_name"
```

# More with Inputs

- You can alter the behavior of **read** with a variety of command line options.
- Two commonly used options however are **-p** which allows you to specify a prompt and **-s** which makes the input silent.
- This can make it easy to ask for a username and password combination like the example below:



login.sh

```bash
#!/bin/bash
# Ask the user for login details

read -p 'Username: ' uservar
read -sp 'Password: ' passvar
echo
echo Thankyou $uservar we now have your login details
```

## Arithmetic operations

- ❑**let expression -> let a=2+3**

Make a variable equal to an expression.

- ❑**expr expression**

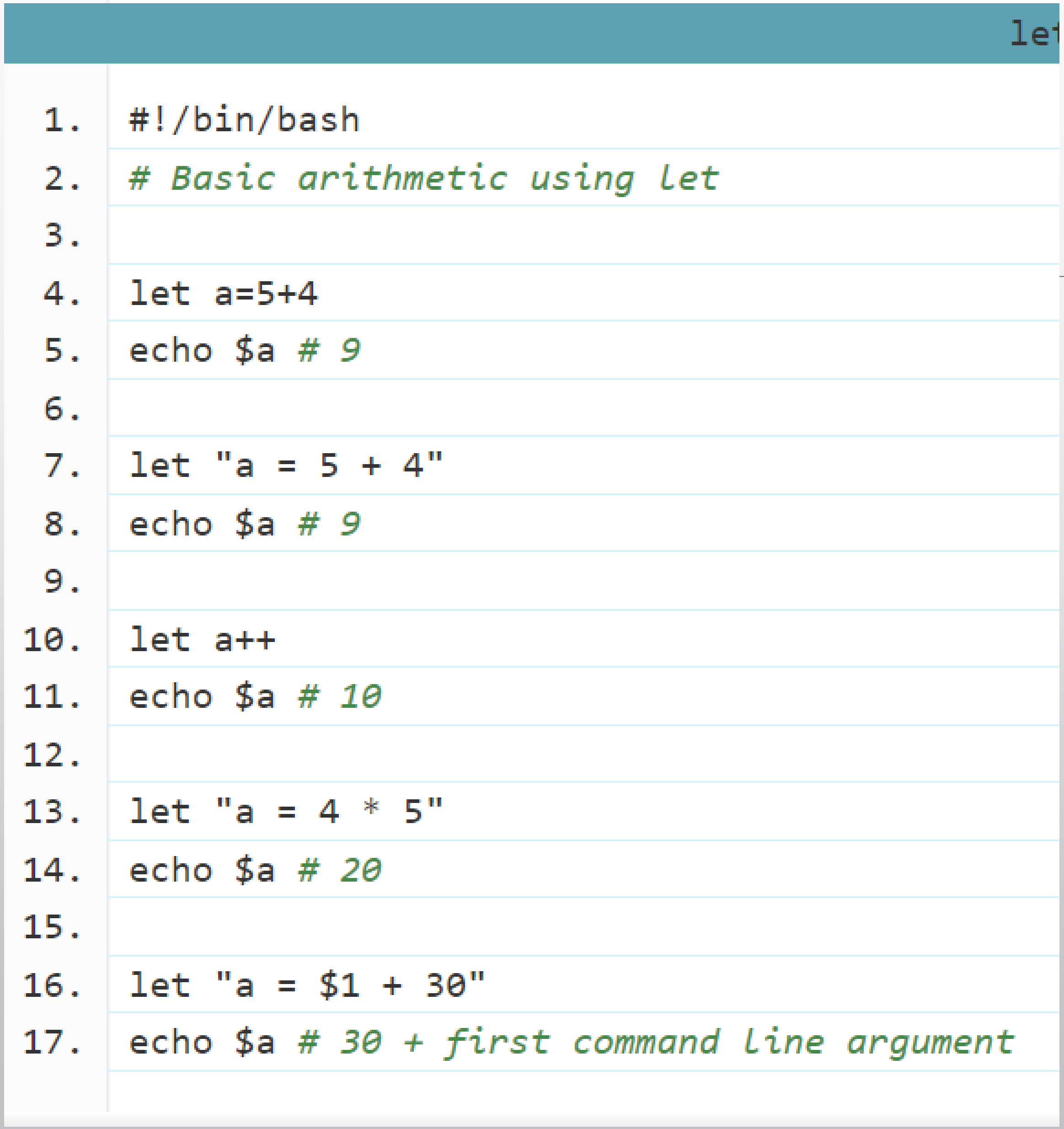
print out the result of the expression.

- ❑**$(( expression )) var=$((a=2+3))**

Return the result of the expression.

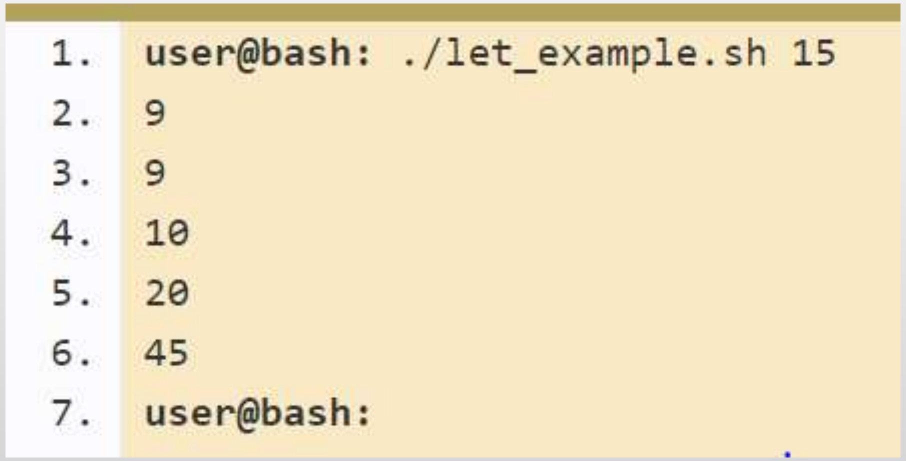
- ❑**${#var}**

Return the length of the variable var.

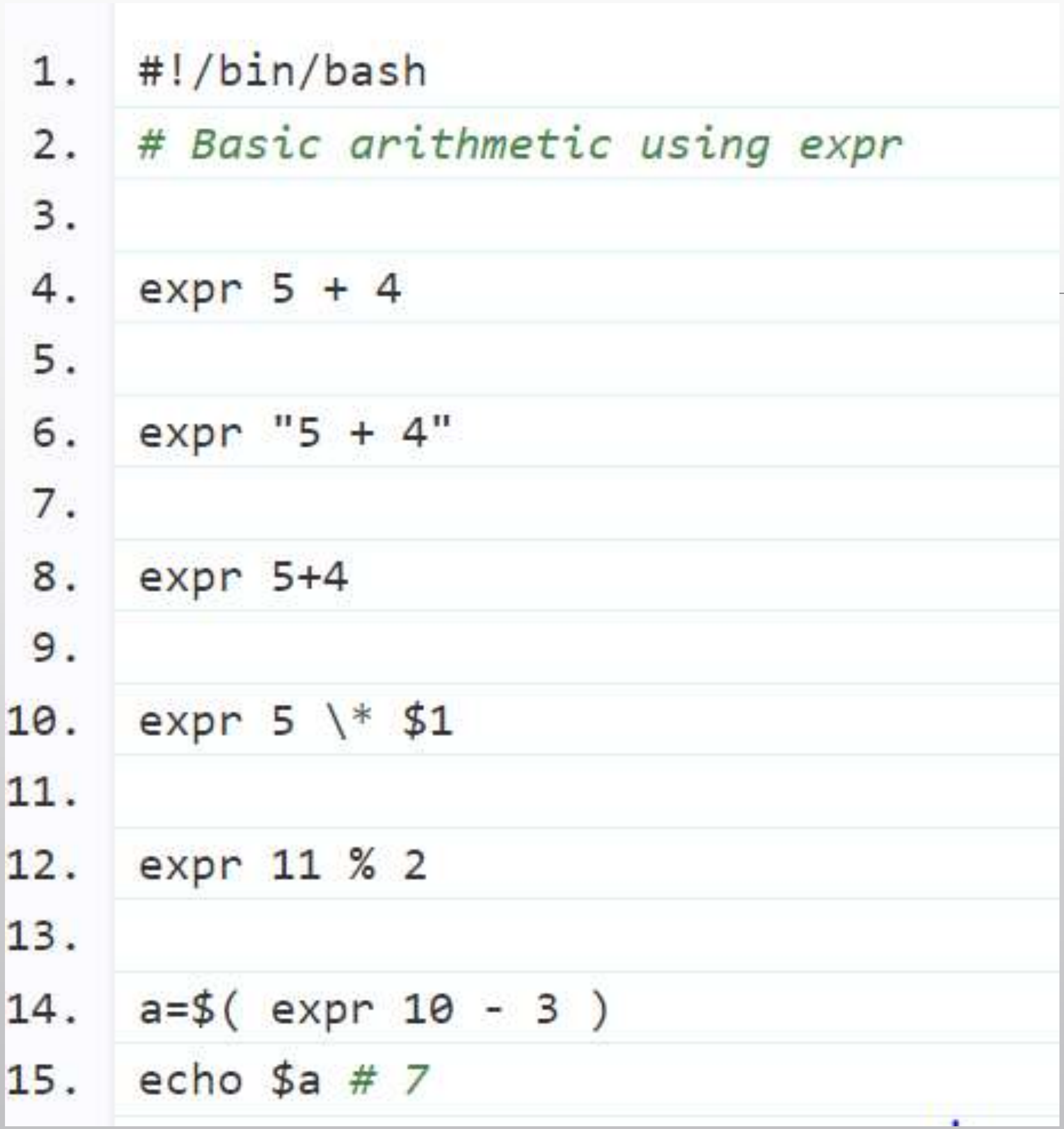


let

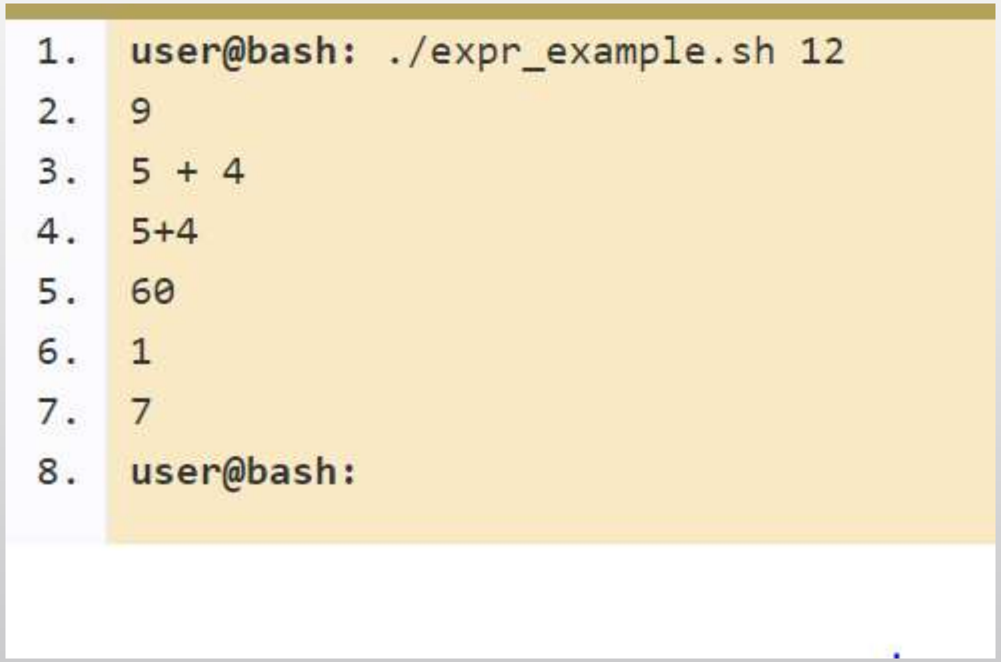
```bash
#!/bin/bash
# Basic arithmetic using let

let a=5+4
echo $a # 9

let "a = 5 + 4"
echo $a # 9

let a++
echo $a # 10

let "a = 4 * 5"
echo $a # 20

let "a = $1 + 30"
echo $a # 30 + first command line argument
```



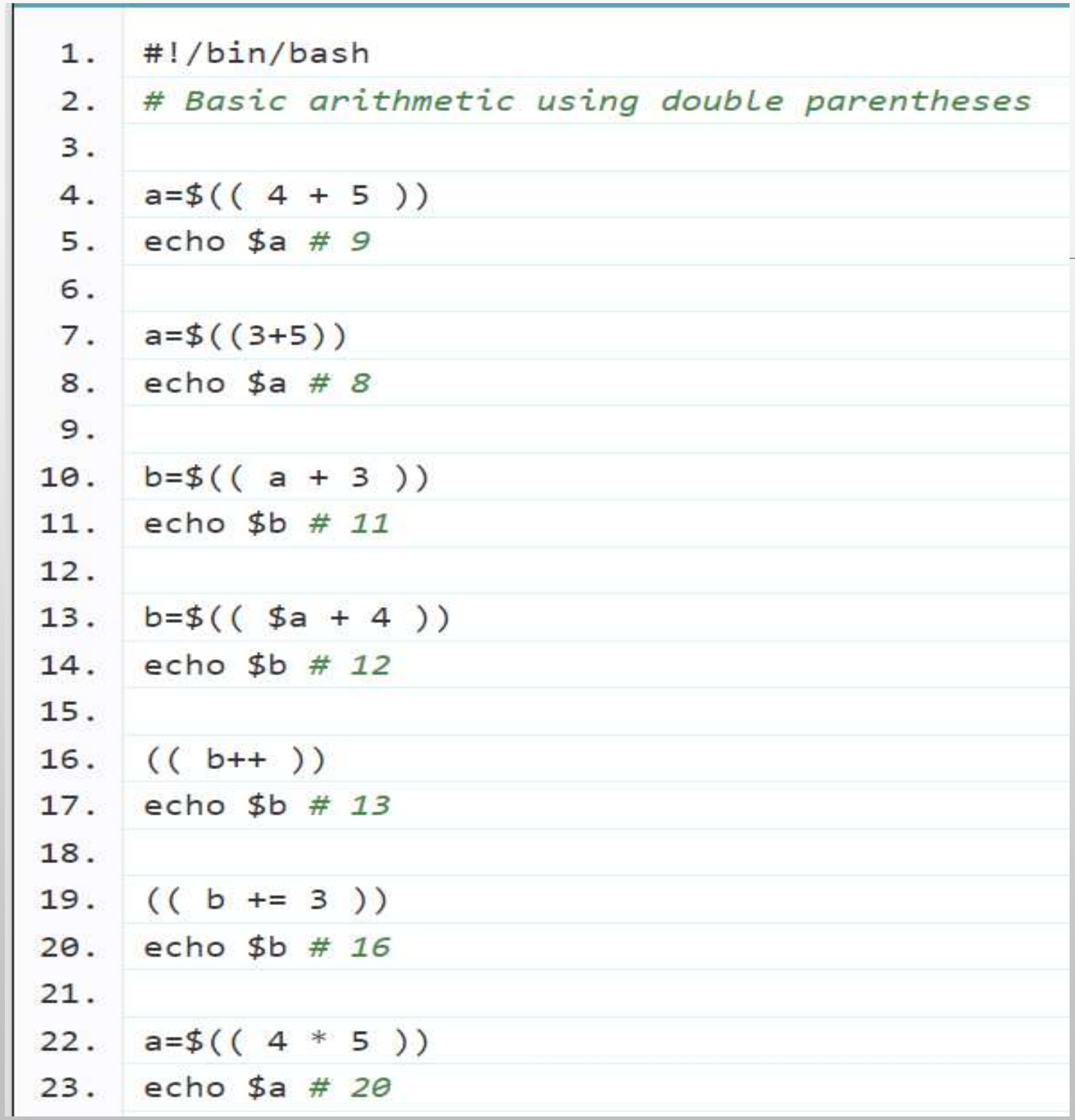
```
user@bash:  ./let_example.sh 15
9
9
10
20
45
user@bash:
```



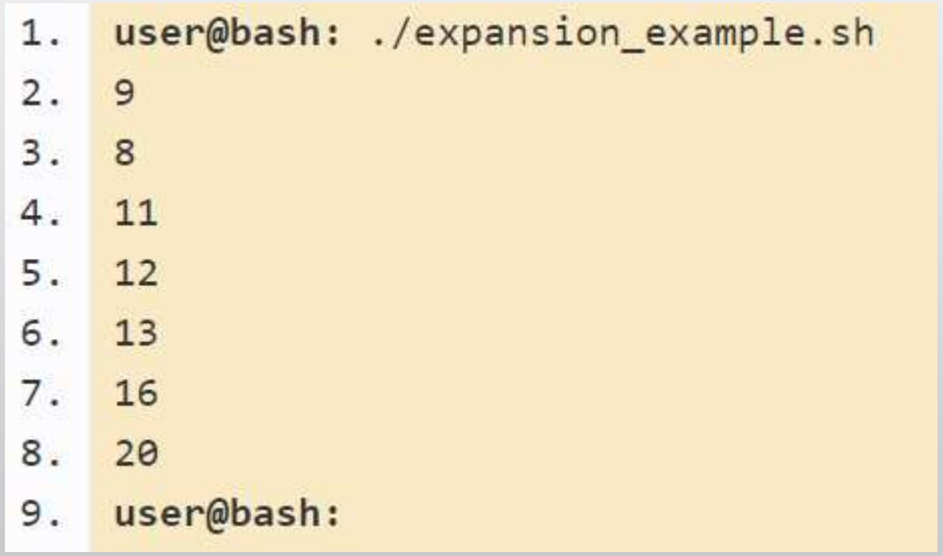
```
#!/bin/bash
# Basic arithmetic using expr

expr 5 + 4

expr "5 + 4"

expr 5+4

expr 5 \* $1

expr 11 % 2

a=$( expr 10 - 3 )
echo $a # 7
```



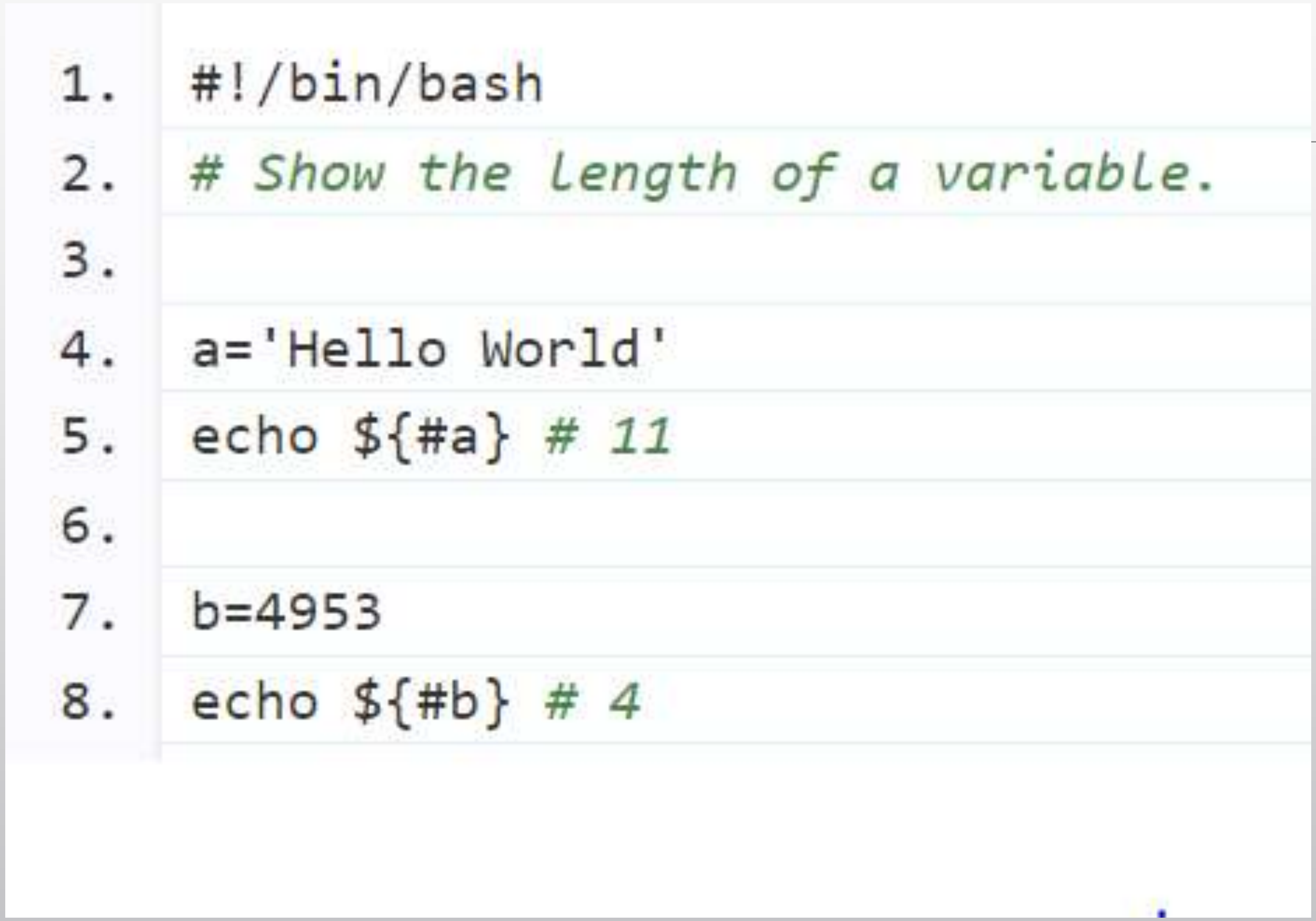
```
user@bash: ./expr_example.sh 12
9
5 + 4
5+4
60
1
7
user@bash:
```



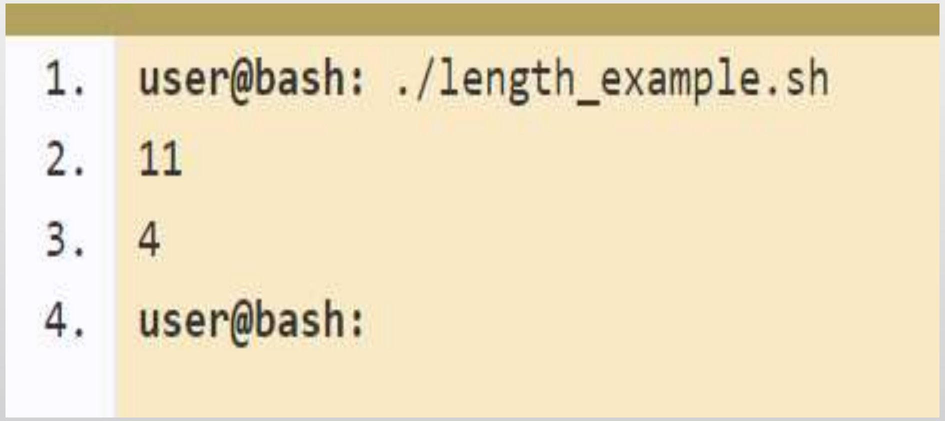
```bash
#!/bin/bash
# Basic arithmetic using double parentheses

a=$(( 4 + 5 ))
echo $a # 9

a=$((3+5))
echo $a # 8

b=$(( a + 3 ))
echo $b # 11

b=$(( $a + 4 ))
echo $b # 12

(( b++ ))
echo $b # 13

(( b += 3 ))
echo $b # 16

a=$(( 4 * 5 ))
echo $a # 20
```



```
user@bash: ./expansion_example.sh
9
8
11
12
13
16
20
user@bash:
```

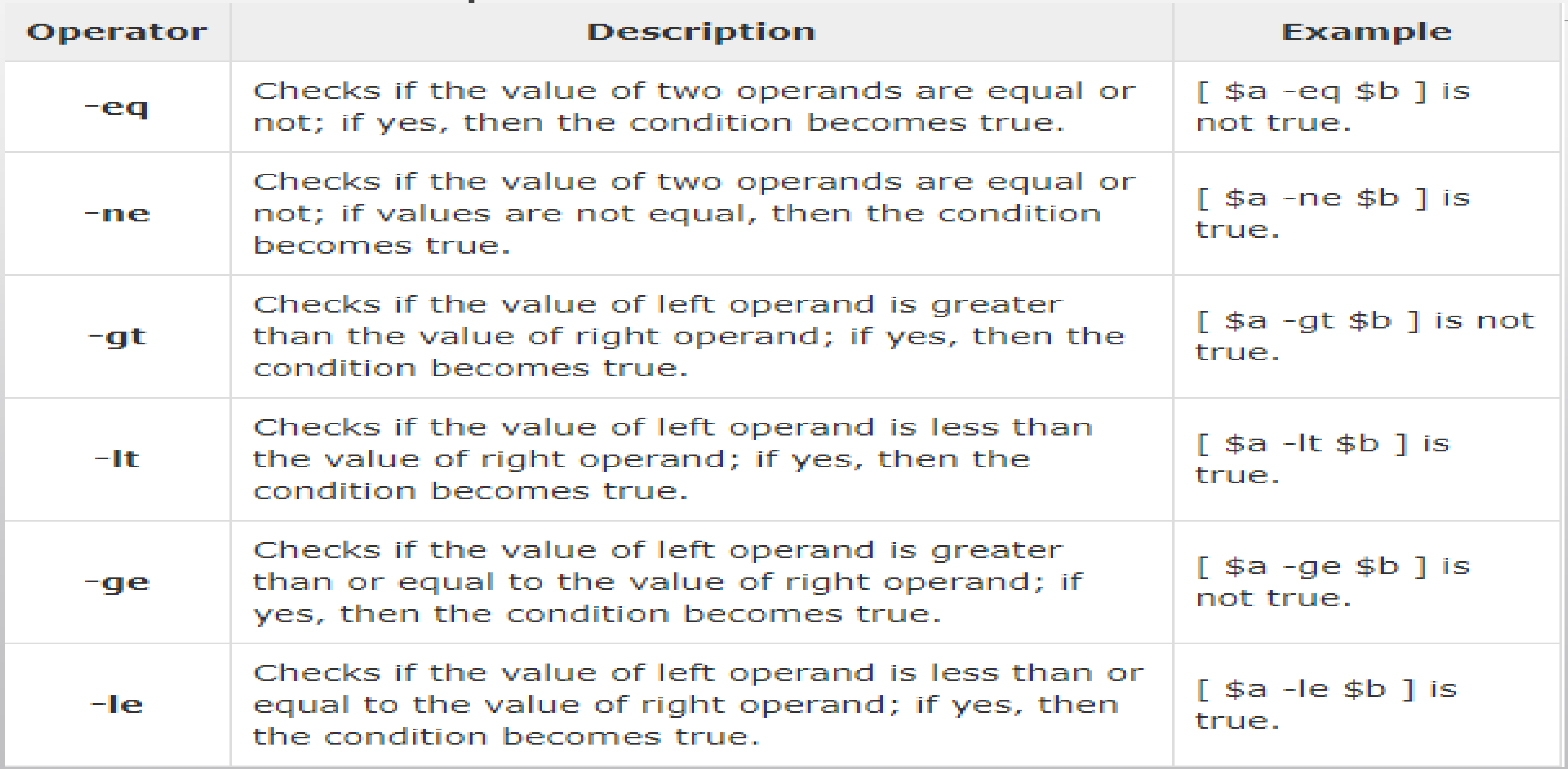


```bash
#!/bin/bash
# Show the length of a variable.

a='Hello World'
echo ${#a} # 11

b=4953
echo ${#b} # 4
```



```
user@bash: ./length_example.sh
11
4
user@bash:
```

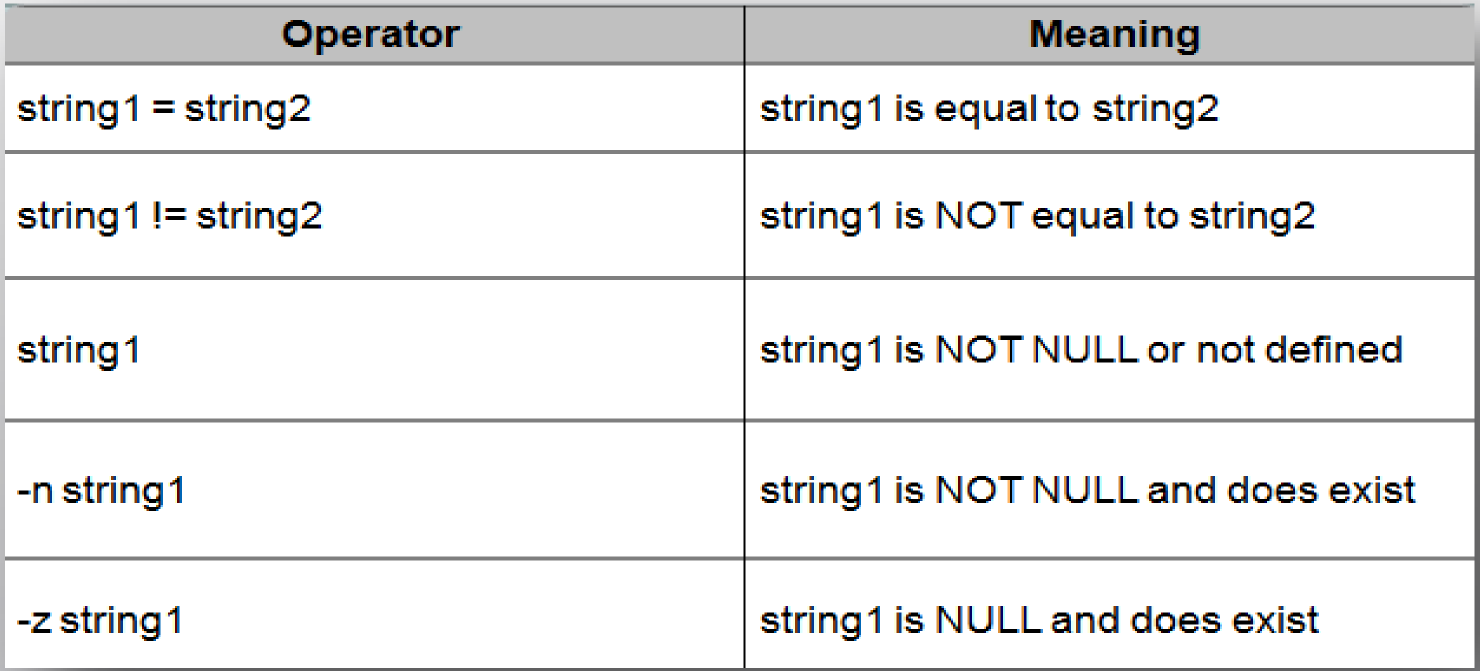
# Relational operators



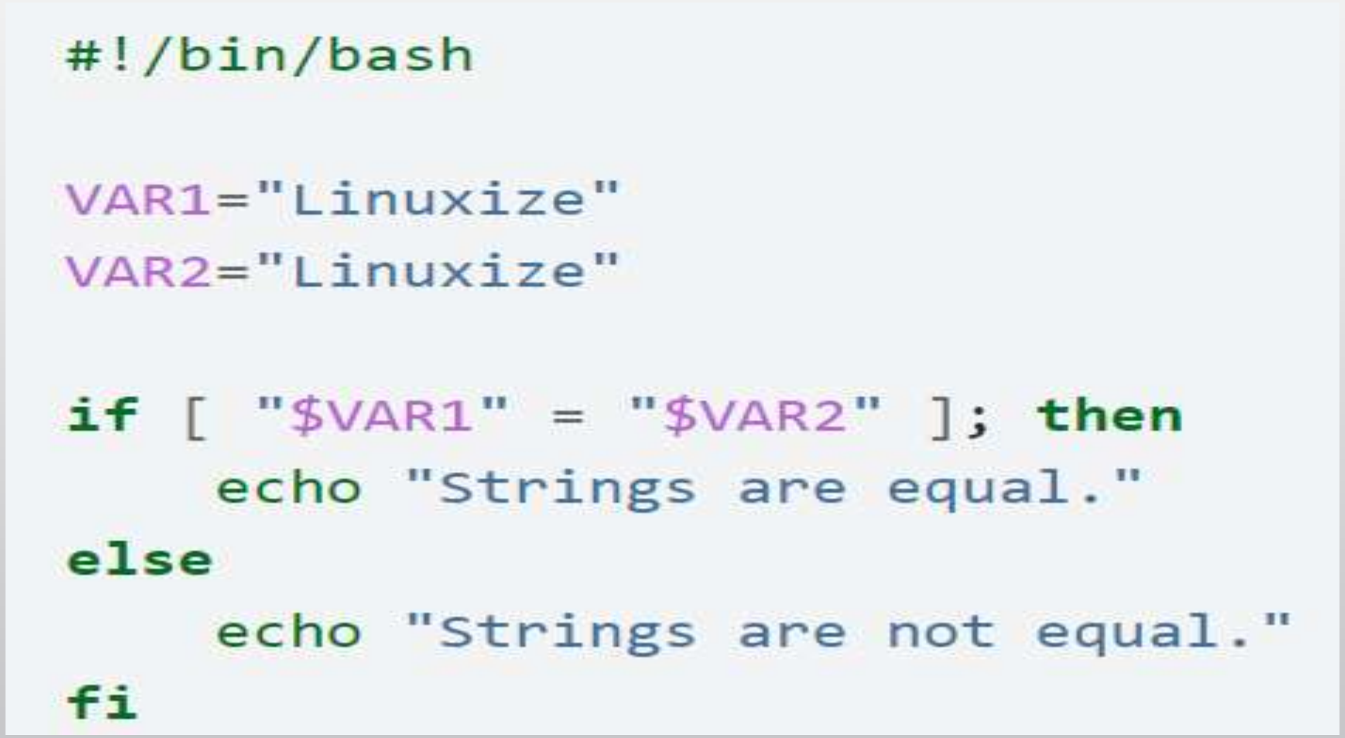
| Operator | Description | Example |
|---|---|---|
| -eq | Checks if the value of two operands are equal or not; if yes, then the condition becomes true. | [ $a -eq $b ] is not true. |
| -ne | Checks if the value of two operands are equal or not; if values are not equal, then the condition becomes true. | [ $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. |

```
#!/bin/sh

a=10
b=20

if (( $a == $b ))
then
   echo "$a -eq $b : a is equal to b"
else
   echo "$a -eq $b: a is not equal to b"
fi

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

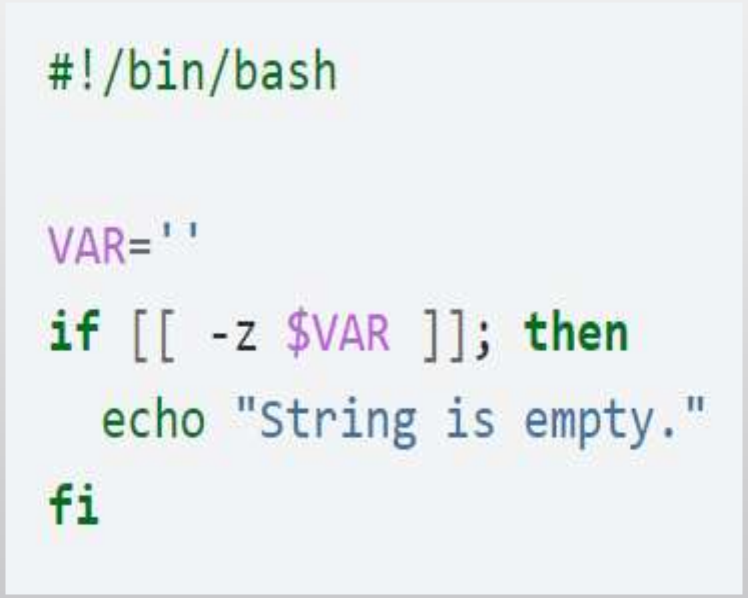
# String Comparison



| Operator | Meaning |
|---|---|
| string1 = string2 | string1 is equal to string2 |
| string1 != string2 | string1 is NOT equal to string2 |
| string1 | string1 is NOT NULL or not defined |
| -n string1 | string1 is NOT NULL and does exist |
| -z string1 | string1 is NULL and does exist |

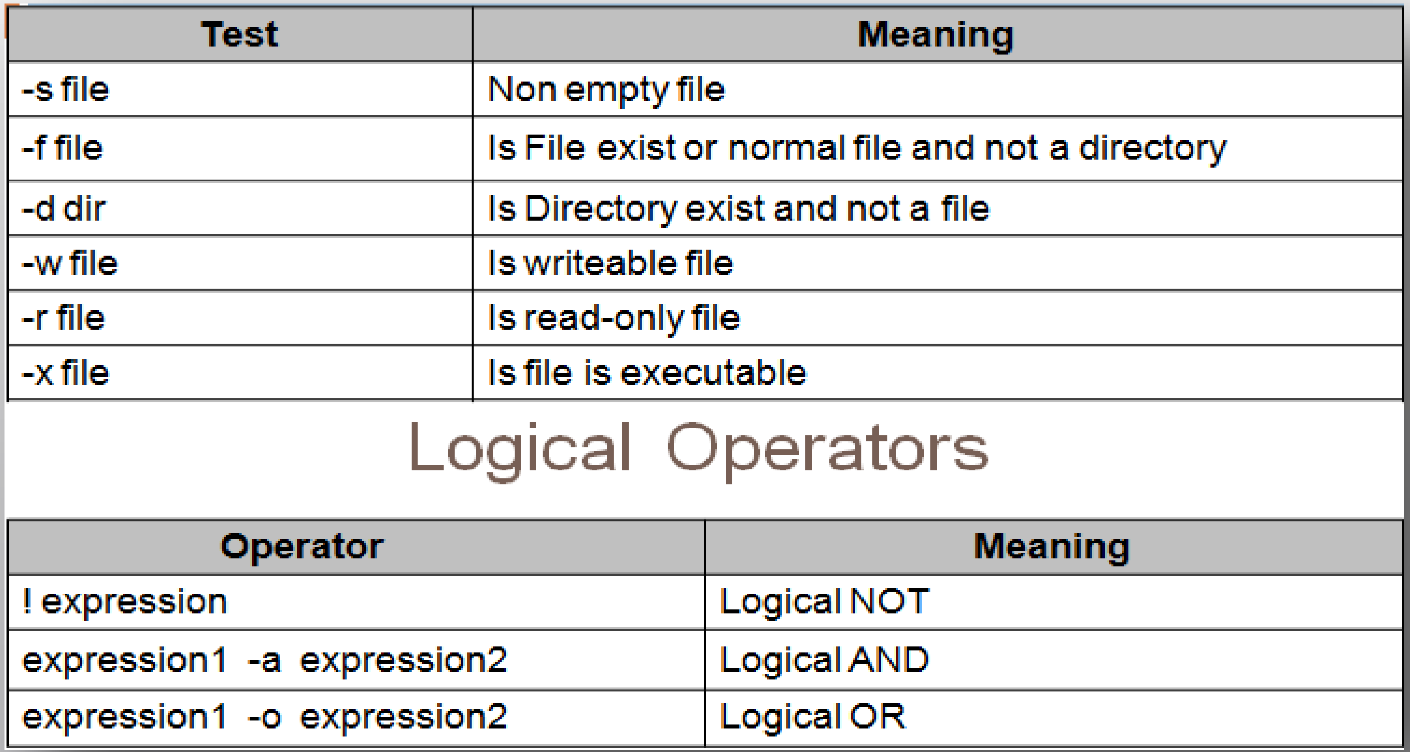


```bash
#!/bin/bash

VAR1="Linuxize"
VAR2="Linuxize"

if [ "$VAR1" = "$VAR2" ]; then
    echo "Strings are equal."
else
    echo "Strings are not equal."
fi
```



```bash
#!/bin/bash

VAR=''
if [[ -z $VAR ]]; then
  echo "String is empty."
fi
```

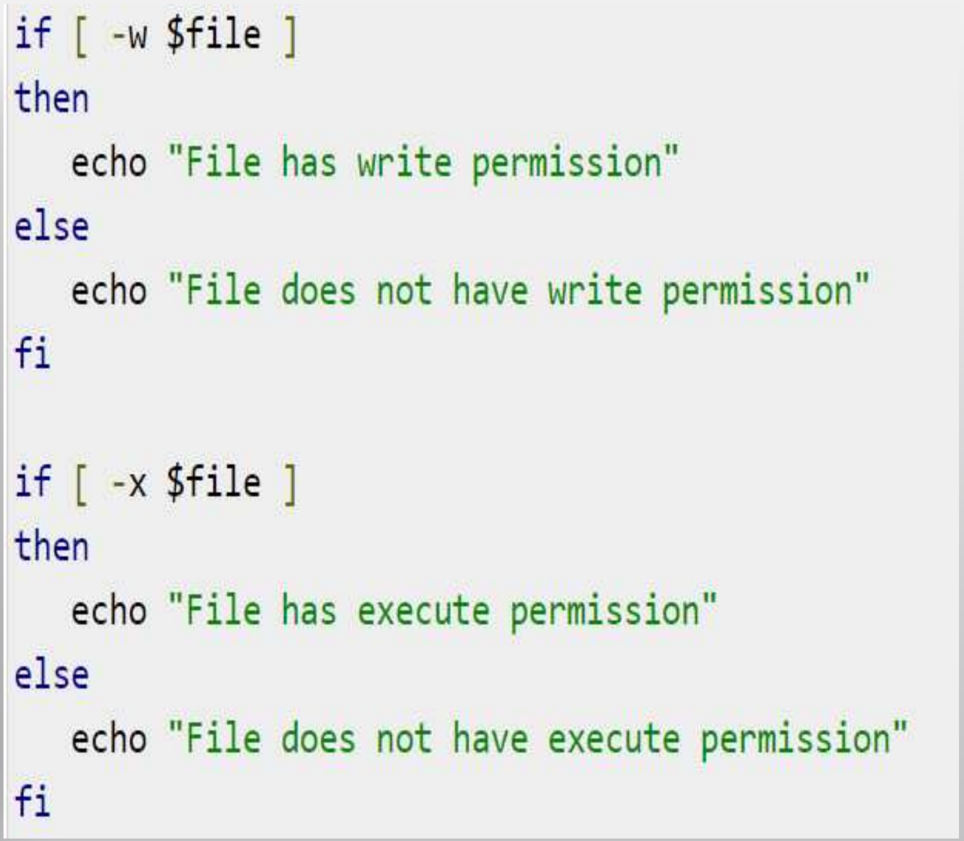
## Other operators



| Test | Meaning |
|---|---|
| -s file | Non empty file |
| -f file | Is File exist or normal file and not a directory |
| -d dir | Is Directory exist and not a file |
| -w file | Is writeable file |
| -r file | Is read-only file |
| -x file | Is file is executable |

Logical Operators

| Operator | Meaning |
|---|---|
| ! expression | Logical NOT |
| expression1 -a expression2 | Logical AND |
| expression1 -o expression2 | Logical OR |



```
if [ -w $file ]
then
   echo "File has write permission"
else
   echo "File does not have write permission"
fi

if [ -x $file ]
then
   echo "File has execute permission"
else
   echo "File does not have execute permission"
fi
```