

# Linux administration with Bash. Lection 2

# Bash

- Scripting
- Q&A.

## Bash. Scripting parameters.

A **bash** shell script can have parameters. The numbering you see in the script below continues if you have more parameters. You also have special parameters containing the number of parameters, a string of all of them, and also the process id, and the last return code. The **man** page of **bash** has a full list.

```
#!/bin/bash
```

```
echo The first argument is $1
```

```
echo The second argument is $2
```

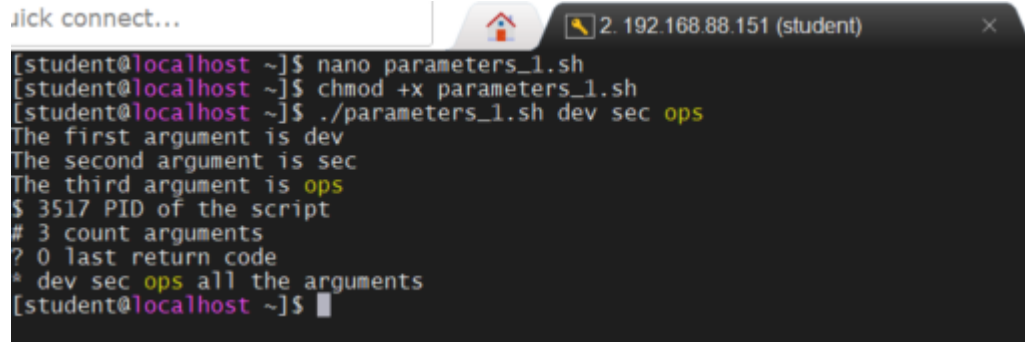
```
echo The third argument is $3
```

```
echo \$$ $$_ PID of the script
```

```
echo \# $# count arguments
```

```
echo \? $? last return code
```

```
echo \* $* all the arguments
```



```
jick connect... 2. 192.168.88.151 (student)
[student@localhost ~]$ nano parameters_1.sh
[student@localhost ~]$ chmod +x parameters_1.sh
[student@localhost ~]$ ./parameters_1.sh dev sec ops
The first argument is dev
The second argument is sec
The third argument is ops
$ 3517 PID of the script
# 3 count arguments
? 0 last return code
* dev sec ops all the arguments
[student@localhost ~]$
```

## Bash. Scripting parameters.

Once more the same script, but with only two parameters

```
quick connect...
[student@localhost ~]$ ./parameters_1.sh dev sec
The first argument is dev
The second argument is sec
The third argument is
$ 3581 PID of the script
# 2 count arguments
? 0 last return code
* dev sec all the arguments
[student@localhost ~]$
```

Here is another example, where we use \$0.  
The \$0 parameter contains the name of the script.

```
#!/bin/bash
echo This script is called $0
echo The first argument is $1
echo The second argument is $2
echo The third argument is $3
echo \$$ $$ PID of the script
echo \# $# count arguments
echo \? $? last return code
echo \* $* all the arguments
```

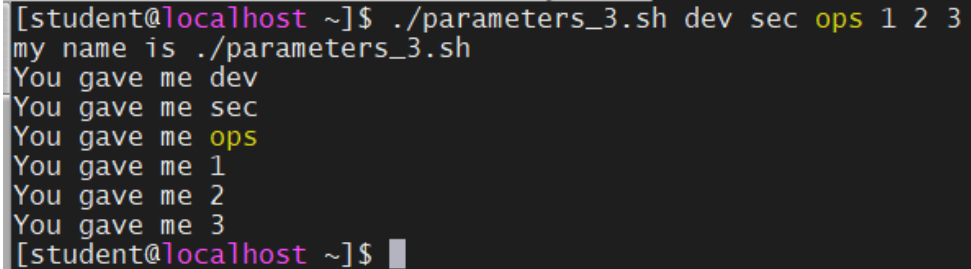
```
quick connect...
[student@localhost ~]$ ./parameters_2.sh dev
This script is called ./parameters_2.sh
The first argument is dev
The second argument is
The third argument is
$ 3606 PID of the script
# 1 count arguments
? 0 last return code
* dev all the arguments
[student@localhost ~]$
```

## Bash. Shift through parameters.

The shift statement can parse all parameters one by one. This is a sample script.

```
#!/bin/bash  
echo my name is $0  
if [ "$#" == "0" ] then  
    echo You have to give at  
least one parameter.  
    exit 1  
fi  
while (( $# ))  
do  
    echo You gave me $1  
    shift  
done
```

quick connect...



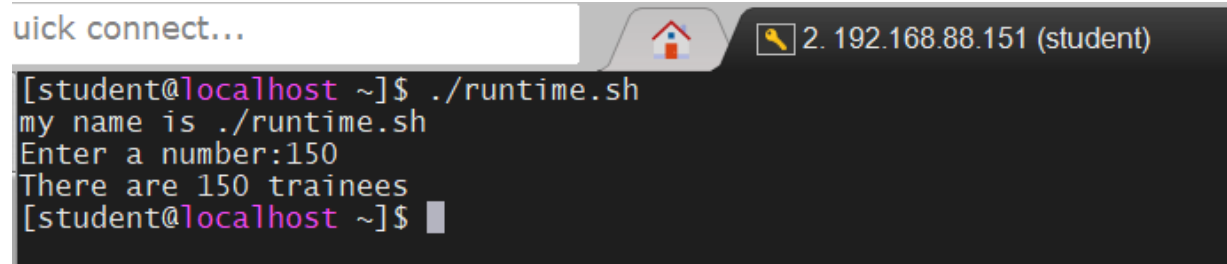
A terminal window titled "quick connect..." showing a remote connection to IP 2.192.168.88.151 (student). The user runs the command `./parameters_3.sh dev sec ops 1 2 3`. The script outputs: "my name is ./parameters\_3.sh", "You gave me dev", "You gave me sec", "You gave me ops", "You gave me 1", "You gave me 2", and "You gave me 3". The prompt returns to `[student@localhost ~]$`.

```
[student@localhost ~]$ ./parameters_3.sh dev sec ops 1 2 3  
my name is ./parameters_3.sh  
You gave me dev  
You gave me sec  
You gave me ops  
You gave me 1  
You gave me 2  
You gave me 3  
[student@localhost ~]$
```

## Bash. Runtime input.

You can ask the user for input with the *read* command in a script

```
#!/bin/bash
echo my name is $0
echo -n Enter a number:
read number
echo There are $number trainees
```

A terminal window showing the execution of a script. The window title is "uick connect...". The top bar shows a home icon, a search icon, and the IP address "2. 192.168.88.151 (student)". The terminal content shows the user running the script, which outputs the script's name, prompts for a number, and then outputs the number of trainees.

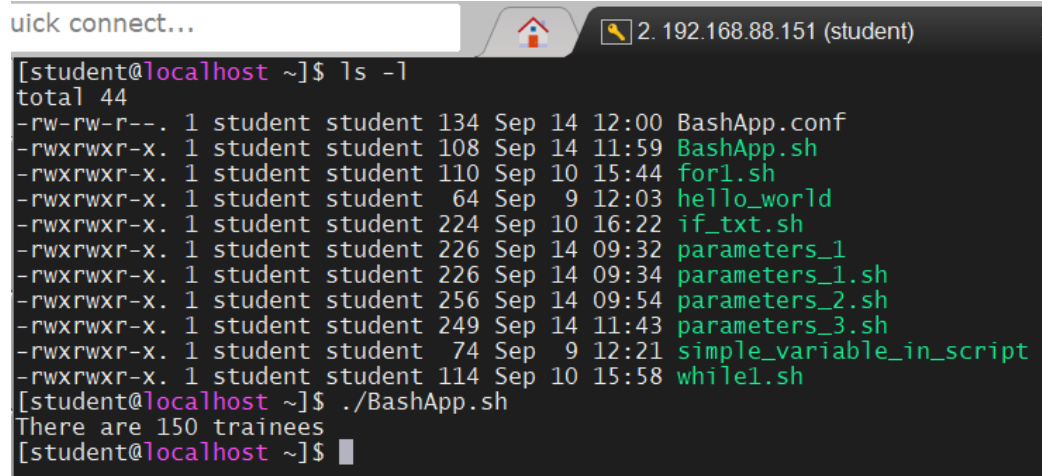
```
uick connect...
[student@localhost ~]$ ./runtime.sh
my name is ./runtime.sh
Enter a number:150
There are 150 trainees
[student@localhost ~]$
```

# Bash. Sourcing a config file.

The **source** can be used to source a configuration file. Below a sample configuration file for an application

```
# The config file of BashApp  
# Enter the path here  
BashAppPath=/home/student/myApp  
# Enter the number of trainees here  
trainees=150
```

Output:



```
uick connect... 2. 192.168.88.151 (student)  
[student@localhost ~]$ ls -l  
total 44  
-rw-rw-r--. 1 student student 134 Sep 14 12:00 BashApp.conf  
-rwxrwxr-x. 1 student student 108 Sep 14 11:59 BashApp.sh  
-rwxrwxr-x. 1 student student 110 Sep 10 15:44 for1.sh  
-rwxrwxr-x. 1 student student 64 Sep 9 12:03 hello_world  
-rwxrwxr-x. 1 student student 224 Sep 10 16:22 if_txt.sh  
-rwxrwxr-x. 1 student student 226 Sep 14 09:32 parameters_1  
-rwxrwxr-x. 1 student student 226 Sep 14 09:34 parameters_1.sh  
-rwxrwxr-x. 1 student student 256 Sep 14 09:54 parameters_2.sh  
-rwxrwxr-x. 1 student student 249 Sep 14 11:43 parameters_3.sh  
-rwxrwxr-x. 1 student student 74 Sep 9 12:21 simple_variable_in_script  
-rwxrwxr-x. 1 student student 114 Sep 10 15:58 while1.sh  
[student@localhost ~]$ ./BashApp.sh  
There are 150 trainees  
[student@localhost ~]$
```

And here an application that uses this file:

```
#!/bin/bash  
#  
# Welcome to the BashApp  
application  
#  
../BashApp.conf  
echo There are $trainees trainees
```

## Bash. Get script options with getopt.

The **getopts** function allows you to parse options given to a command. The following script allows for any combination of the options a, b and c

```
#!/bin/bash
while getopts ":abc" option;
do
  case $option in
    a)
      echo received -a ;;
    b)
      echo received -b ;;
    c)
      echo received -c ;;
    *)
      echo "invalid option -$OPTARG" ;;
  esac
done
```

Output:

```
quick connect...
[student@localhost ~]$ ./options.sh -abc
received -a
received -b
received -c
[student@localhost ~]$ ./options.sh -defcl
invalid option -d
invalid option -e
invalid option -f
received -c
invalid option -l
[student@localhost ~]$ ./options.sh -abcdef
received -a
received -b
received -c
invalid option -d
invalid option -e
invalid option -f
[student@localhost ~]$ ./options.sh -abde
received -a
received -b
invalid option -d
invalid option -e
[student@localhost ~]$
```



## Bash. Get script options with getopt.

You can also check for options that need an argument, as this example shows.

```
#!/bin/bash
while getopt ":ab:c:" option;
do
case $option in
a)
echo received -a ;;
b)
echo received -b with $OPTARG ;;
c)
echo received -c with $OPTARG ;;
:)
echo "option -$OPTARG needs an argument" ;;
*)
echo "invalid option -$OPTARG" ;;
esac
done
```

Output:

```
quick connect... 2. 192.168.88.151 (student)
[student@localhost ~]$ ./arg_options.sh -a -b dev -c ops
received -a
received -b with dev
received -c with ops
[student@localhost ~]$ ./arg_options.sh -abc ops
received -a
received -b with c
[student@localhost ~]$ ./arg_options.sh -abcops
received -a
received -b with cops
[student@localhost ~]$ ./arg_options.sh -azcnbcops
received -a
invalid option -z
received -c with nbcops
[student@localhost ~]$ ./arg_options.sh -nmopab devops -c cool
invalid option -n
invalid option -m
invalid option -o
invalid option -p
received -a
received -b with devops
received -c with cool
[student@localhost ~]$
```

## Bash. Additional scripting elements.

*eval* reads arguments as input to the shell (the resulting commands are executed). This allows using the value of a variable as a variable.

```
> answer=42  
> word=answer  
> eval x=\$$word ; echo $x  
> 42
```

In bash the arguments can be quoted

```
> answer=42  
> word=answer  
> eval "y=\$$word" ; echo $x  
> 42
```

## Bash. Additional scripting elements.

Sometimes the *eval* is needed to have correct parsing of arguments. Consider this example where the *date* command receives one parameter **1 week ago**

```
[student@localhost ~]$ date --date="1 week ago"  
Mon Sep 7 23:38:02 EEST 2020
```

When we set this command in a variable, then executing that variable fails unless we use *eval*

```
[student@localhost ~]$ lastweek='date --date="1 week ago"'  
[student@localhost ~]$ $lastweek  
date: extra operand 'ago'  
Try 'date --help' for more information.  
[student@localhost ~]$ eval $lastweek  
Mon Sep 7 23:44:33 EEST 2020  
[student@localhost ~]$
```

## Bash. Additional scripting elements.

The `(( ))` allows for evaluation of numerical expressions

```
> (( 42 > 33 )) && echo true || echo false
> true
> (( 42 > 1201 )) && echo true || echo false
> false
> var42=42
> (( 42 == var42 )) && echo true || echo false
> true
> (( 42 == $var42 )) && echo true || echo false
> true
> var42=33
> (( 42 == var42 )) && echo true || echo false
> false
```

## Bash. Additional scripting elements.

The let built-in shell function instructs the shell to perform an evaluation of arithmetic expressions.

```
[student@localhost ~]$ let x="3 + 4" ; echo $x
7
[student@localhost ~]$ let x="10 + 100/10" ; echo $x
20
[student@localhost ~]$ let x="10-2+100/10" ; echo $x
18
[student@localhost ~]$ let x="10*2+100/10" ; echo $x
30
```

There is a difference between assigning a variable directly, or using let to evaluate the arithmetic expressions (even if it is just assigning a value).

The shell can also convert between different bases.

```
[student@localhost ~]$ let x="0xFF" ; echo $x
255
[student@localhost ~]$ let x="0xC0" ; echo $x
192
[student@localhost ~]$ let x="0xA8" ; echo $x
168
[student@localhost ~]$ let x="8#70" ; echo $x
56
[student@localhost ~]$ let x="8#77" ; echo $x
63
[student@localhost ~]$ let x="16#c0" ; echo $x
192

[student@localhost ~]$ dec=15 ; oct=017 ; hex=0x0f
[student@localhost ~]$ echo $dec $oct $hex
15 017 0x0f
[student@localhost ~]$ let dec=15 ; let oct=017 ; let hex=0x0f
[student@localhost ~]$ echo $dec $oct $hex
15 15 15
```

## Bash. Additional scripting elements.

You can sometimes simplify nested **if** statements with a **case** construct

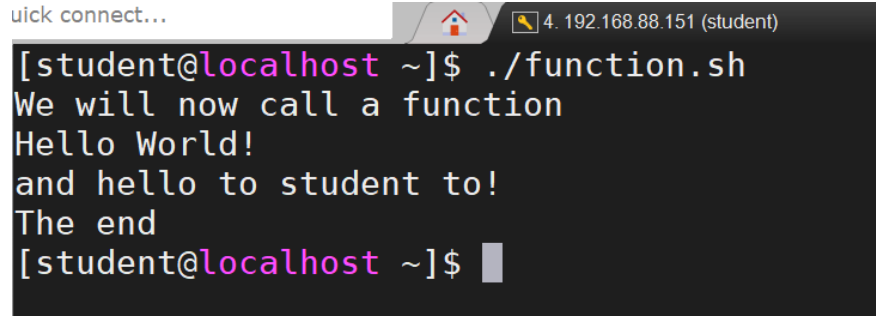
```
#!/bin/bash
# Job Helpdesk Advisor :-)
echo -n "What job do you want ? "
read job
case $job in
    "devops")
        echo "Excellent"
    ;;
    "dev")
        echo "Good"
    ;;
    "test")
        echo "not bad."
    ;;
    "frontend")
        echo "Really???"
    ;;
    *)
        echo "Make your choise once more from: devops, dev, test and frontend"
    ;;
esac
```

## Bash. Additional scripting elements.

Shell *functions* can be used to group commands in a logical way.

```
#!/bin/bash  
function greetings {  
echo Hello World!  
echo and hello to $USER to!  
}  
echo We will now call a function  
greetings  
echo The end
```

uick connect...

A terminal window with a dark background. The title bar shows a home icon, a signal strength icon, and the IP address '4. 192.168.88.151 (student)'. The terminal content shows a user at a localhost prompt running a script. The script's output is displayed in white text.

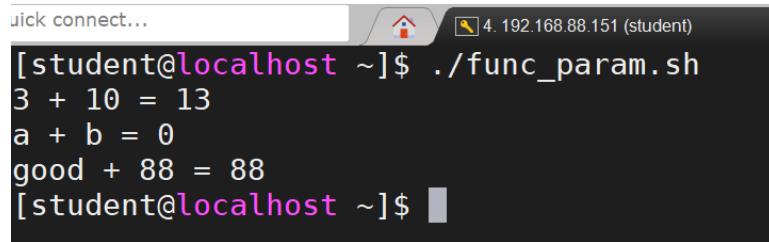
```
[student@localhost ~]$ ./function.sh  
We will now call a function  
Hello World!  
and hello to student to!  
The end  
[student@localhost ~]$ █
```

# Bash. Additional scripting elements.

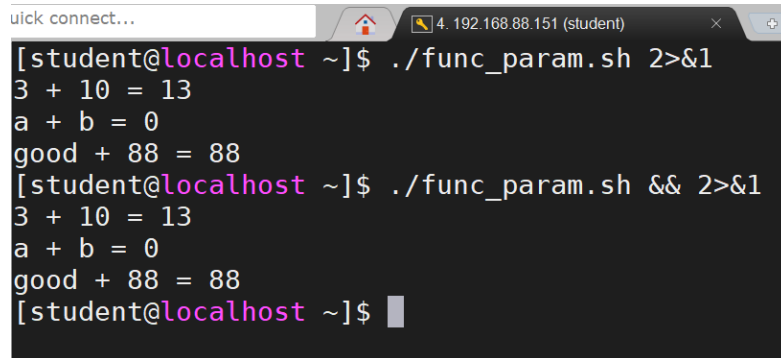
A shell **function** can also receive parameters

```
#!/bin/bash
function plus {
  let result="$1 + $2"
  echo $1 + $2 = $result
}
```

```
plus 3 10
plus a b
plus good 88
```



```
quick connect... 4. 192.168.88.151 (student)
[student@localhost ~]$ ./func_param.sh
3 + 10 = 13
a + b = 0
good + 88 = 88
[student@localhost ~]$
```



```
quick connect... 4. 192.168.88.151 (student)
[student@localhost ~]$ ./func_param.sh 2>&1
3 + 10 = 13
a + b = 0
good + 88 = 88
[student@localhost ~]$ ./func_param.sh && 2>&1
3 + 10 = 13
a + b = 0
good + 88 = 88
[student@localhost ~]$
```



## Bash. Shell expansions.Quotes

Notice that double quotes still allow the parsing of variables, whereas single quotes prevent this.

```
$ MyVar=555  
$ echo $MyVar  
555  
$ echo "$MyVar"  
555  
$ echo '$MyVar'  
MyVar
```

The bash shell will replace variables with their value in double quoted lines, but not in single quoted lines.

```
$ city=Burtonville  
$ echo "We are in $city today."  
We are in Burtonville today.  
$ echo 'We are in $city today.'  
We are in $city today.
```

## Bash. Shell expansions. Backticks or single quotes

Single embedding can be useful to avoid changing your current directory. The screenshot below uses backticks instead of dollar-bracket to embed.

```
$ echo `cd /etc; ls -d * | grep pass`  
passwd passwd- passwd.OLD  
$
```

Placing the embedding between backticks uses one character less than the dollar and parenthesis combo. Be careful however, backticks are often confused with single quotes. The technical difference between ' and ` is significant!

```
$ echo `var1=5;echo $var1`  
5  
$ echo 'var1=5;echo $var1'  
var1=5;echo $var1  
$
```

# Bash. Shell expansions.

## Read a File:

You can read any file line by line in bash by using loop. Create a file named, '**read\_file.sh**' and add the following code to read an existing file named, '**book.txt**'.

```
GNU nano 2.3.1 File: read_file.sh
#!/bin/bash
file='books.txt'
while read line; do
echo $line
done < $file
```

```
uick connect... 2. 192.168.88.151 (student)
student@localhost~$ bash read_file.sh
Project Phoenix
Continious integration
Working with Jenkins
Ansible forever
student@localhost~$
```



**Thank you!**