OPerating Systems Laboratory
OPS*Lab*

PART III
Bash Scripting

Prof. Marco Autili
University of L'Aquila
marco.autili@univaq.it
http://people.disim.univaq.it/marco.autili/

"Lasciate ogni speranza, voi ch'intrate" 😮

Scripts

- Scripts permit to define a series of actions/commands in a non-interactive way
- Scripts are interpreted by the shell
  - we will keep on using the Bash shell
- A script is a plain-text file
- Useful when repetitive task must be accomplished
- Useful when recurrent problems must be solved
- . . .

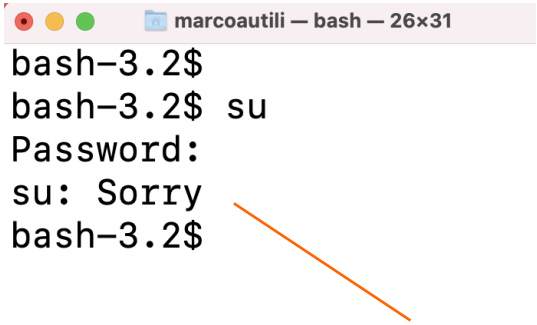
Scripting VS Permission

- It can happen that if you write your script by using a text editor (e.g., Sublime, TextEdit) you cannot save it unless you open the editor as a super-user, e.g.,
  - sudo open /Applications/Sublime\ Text\ 2.app
  - sudo open /System/Applications/TextEdit.app/

- After saving the script file, it can also happen that you do not have the right permissions to execute the script
  - for example, if you want to enable the owner to write or modify the script and everyone else to execute the script only, you can use the shorthand 755 with chmod (see next slides)

Gaining Privileges: su

- The **su** command is mainly used to **s**witch to some other **u**ser during an existing session (the default user is the superuser)
- In other words, the command lets you assume the identity of some other user without having to logout and then login (as that user)
- The su command is mostly used to switch to the superuser account (i.e., root account), as root privileges are frequently required while working on the command line, but you can use it to switch to any other, non-root user as well
- The su command requires you to enter the password of the target user (the root user by default)
- After the correct password is entered, the command starts a sub-session inside the existing session on the terminal, precisely, a new shell belonging to the target user is executed

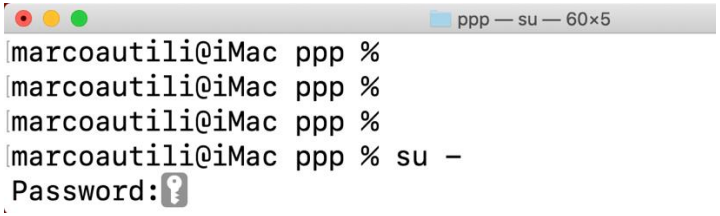


In macOS, the root user is disabled by default, therefore su will not work

It's better to use sudo (next slides)

Gaining Privileges: su

- There's another way to switch to the root user: su - or, equivalently, su -l



- Roughly speaking, the difference is that (see man su for a detailed description):
  - 'su' keeps the environment of the old/original user (generally, with the exception of USER, HOME, and SHELL)
  - 'su –' and, equivalently, 'su –l' both create a new environment similar to the case when you explicitly log in as root user from the login screen
    - In practice, depending on the system, the above may translate into executing, e.g., the ~/.bashrc or the /etc/profile and ~/.profile also

```
su -l foo
       Simulate a login for user foo.
su - foo
       Same as above.
su -   Simulate a login for root.
```

More on su

A Pluggable Authentication Module (PAM) is a mechanism to integrate multiple low-level authentication schemes into a high-level application programming interface (API). PAM allows programs that rely on authentication to be written independently of the underlying authentication scheme.

DESCRIPTION

The **su** utility requests appropriate user credentials via PAM and switches to that user ID (the default user is the superuser). A shell is then executed.

PAM is used to set the policy su(1) will use. In particular, by default only users in the ``admin'' or ``wheel'' groups can switch to UID 0 (``root''). This group requirement may be changed by modifying the ``pam_group'' section of /etc/pam.d/su. See pam_group(8) for details on how to modify this setting.

By default, the environment is unmodified with the exception of USER, HOME, and SHELL. HOME and SHELL are set to the target login's default values. USER is set to the target login, unless the target login has a user ID of 0, in which case it is unmodified. The invoked shell is the one belonging to the target login. This is the traditional behavior of **su**.

The options are as follows:

-f If the invoked shell is csh(1), this option prevents it from reading the ``.cshrc'' file.

-l Simulate a full login. The environment is discarded except for HOME, SHELL, PATH, TERM, and USER. HOME and SHELL are modified as above. USER is set to the target login. PATH is set to ``/bin:/usr/bin''. TERM is imported from your current environment. The invoked shell is the target login's, and **su** will change directory to the target login's home directory.

- (no letter) The same as **-l**.

-m Leave the environment unmodified. The invoked shell is your login shell, and no directory changes are made. As a security precaution, if the target user's shell is a non-standard shell (as defined by getusershell(3)) and the caller's real uid is non-zero, **su** will fail.

The **-l** (or **-**) and **-m** options are mutually exclusive; the last one specified overrides any previous ones.

Gaining Privileges: sudo

sudo [-u user] [-i | -s] [command] (stands for "Substitute User do", originally "superuser do")

- sudo execute a command as another user. It allows a permitted user to execute a command as the superuser or another user, as specified by the security policy

- The default user is the superuser, thus root is the default value of the [user] parameter

- Unlike the similar command su, users must, by default, supply their own password for authentication, rather than the password of the target user

- The default security policy is sudoers, which is generally configured via the file /etc/sudoers
  - thus, after authentication, the permissions to run commands are granted according to the configuration file /etc/sudoers... see also /private/etc/sudoers (editable with visudo – next slide)

Gaining Privileges: sudo

- visudo is a command-line utility that allows editing the sudo configuration file in a fail-safe manner. It prevents multiple simultaneous edits with locks and performs sanity and syntax checks

```
# root and users in group wheel can run anything on any machine as any user

root ALL = (ALL) ALL

%admin ALL = (ALL) ALL

%users localhost=/usr/sbin/shutdown -h now
```

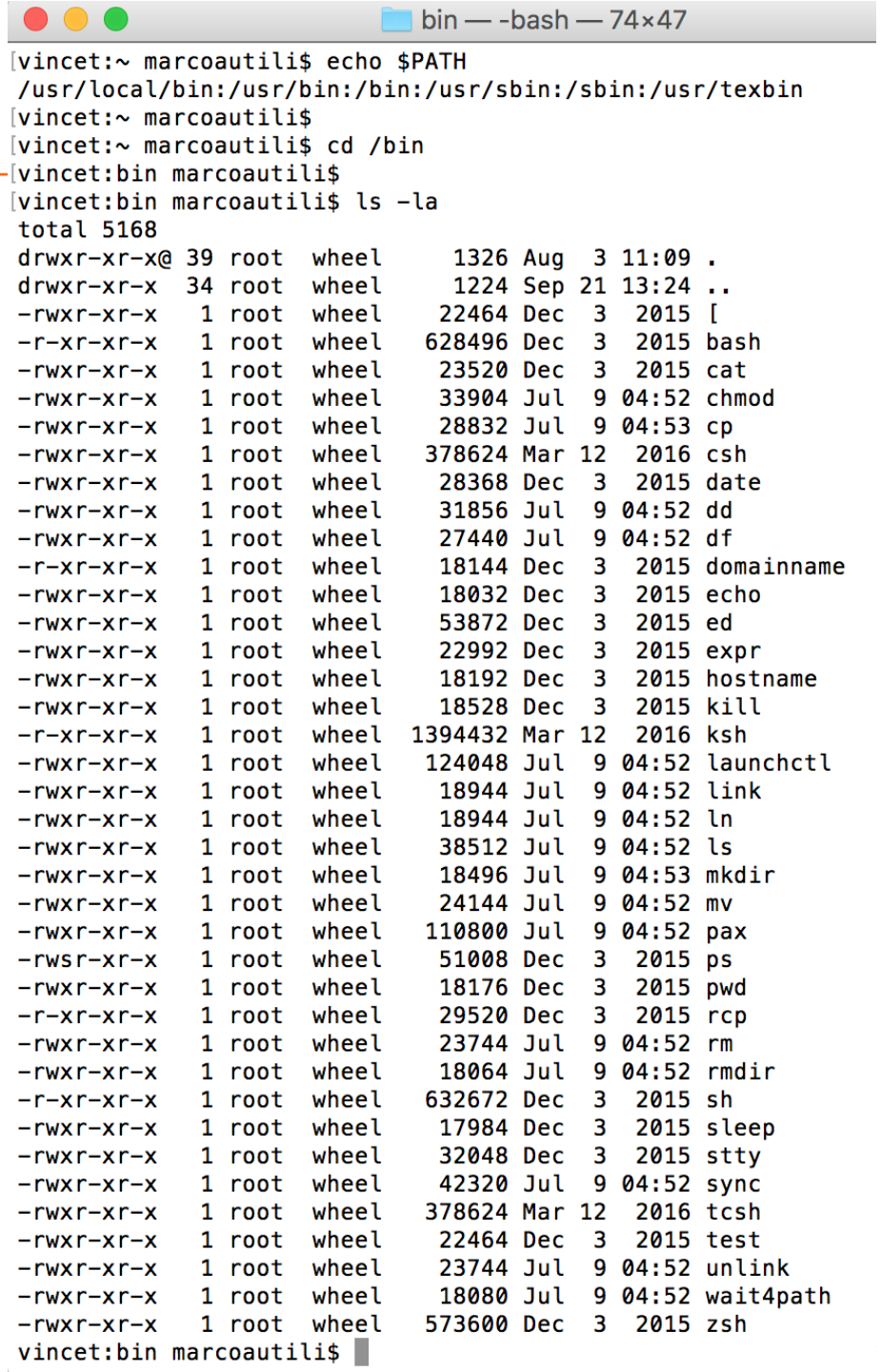
- Can be used to get a shell prompt
  - sudo -s – run the shell specified by the SHELL environment variable if it is set or the shell specified by the invoking user's password database entry
  - sudo -i or sudo --login – similar to real login, run the shell specified by the target user's password database entry as a login shell and initializes the shell environment. This means that login-specific resource files such as .profile, .bash_profile or .login will be read by the shell

$PATH (from PART 1)

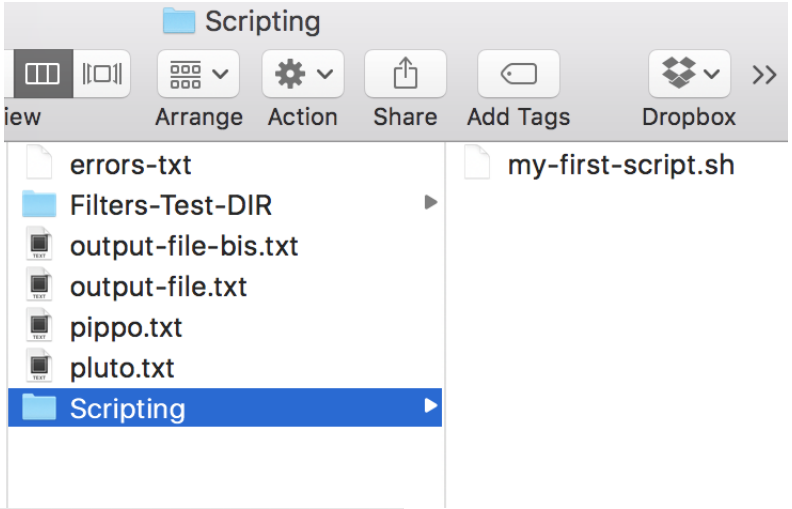
- When we type a command or script or program on the terminal, the system searches for it in a predefined set of directories

- The predefined set of directories to be searched is specified by the variable $PATH

- The system will not look inside the current directory, that's why when we want to run our own script from within the current directory, we need to use "./"
  - ./myfirstscript.sh



Scripting VS Permission




```
Scripting — -bash — 74×14
desktop-jisqlks:Scripting marcoautili$ ls -la
total 8
drwxr-xr-x@  3 marcoautili  staff  102 Sep 22 13:09 .
drwxr-xr-x@ 10 marcoautili  staff  340 Sep 22 13:09 ..
-rw-r--r--@  1 marcoautili  staff   52 Sep 22 13:04 my-first-script.sh
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$ chmod 755 my-first-script.sh
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$ ls -la
total 8
drwxr-xr-x@  3 marcoautili  staff  102 Sep 22 13:09 .
drwxr-xr-x@ 10 marcoautili  staff  340 Sep 22 13:09 ..
-rwxr-xr-x@  1 marcoautili  staff   52 Sep 22 13:04 my-first-script.sh
desktop-jisqlks:Scripting marcoautili$
```

Running a script

```
Scripting — -bash — 57×5
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$ which bash
/bin/bash
desktop-jisqlks:Scripting marcoautili$
```

Tell us where the bash shell is located

In general, which <program>

#! - called Shebang
Must be in the very first line

my-first-script.sh

```
1  #!/bin/bash
2  # This is the first script I am showing
3  # Marco A. dd/mm/yyyy
4
5  echo
6  echo Before listing
7  echo
8  ls ..
9  echo
10 echo After listing... see you next time!
11 echo
12
```

```
Scripting — -bash — 63×12
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$ ./my-first-script.sh

Before listing

Filters-Test-DIR        output-file-bis.txt     pluto.txt
Scripting               output-file.txt
errors-txt              pippo.txt

After listing... see you next time!

desktop-jisqlks:Scripting marcoautili$
```

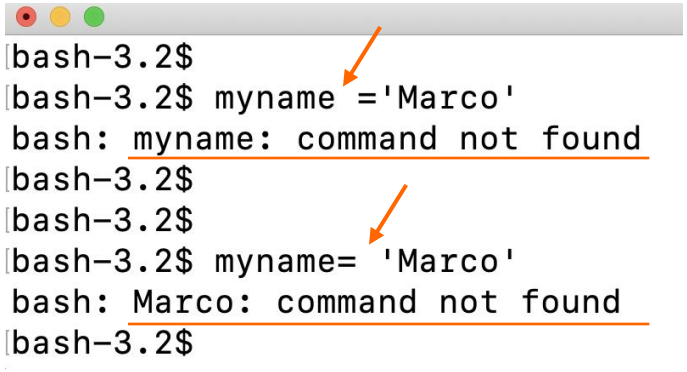
Variables and command line arguments

myname='Marco'

echo My name is $myname

variable=value

No space before nor after "="



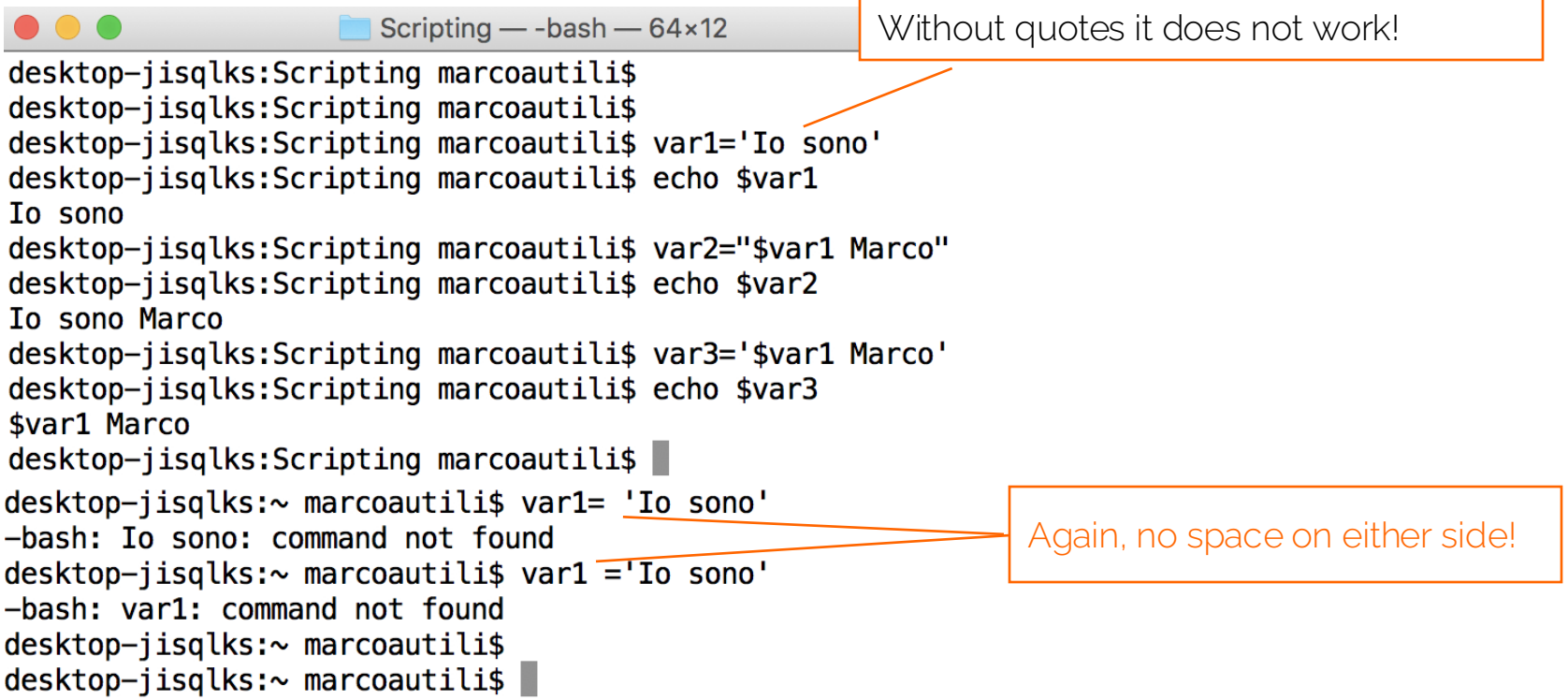
- $0 - The name of the script
- $1 - $9 - Any command line arguments given to the script: $1 is the first argument, $2 the second and so on
- $# - How many command line arguments were given to the script.
- $* or $@ - All of the command line arguments passed to the script
- $? - The exit status of the most recently run process
- $$ - The process ID of the current script
- $USER - The username of the user running the script
- $HOSTNAME - The hostname of the machine the script is running on
- $SECONDS - The number of seconds since the script was started
- $RANDOM - Returns a different random number each time is it referred to
- $LINENO - Returns the current line number in the Bash script

Type env on the command line to see other available variables

Single quotes VS double quotes

'Single quotes' will treat every character literally

"Double quotes" will allow you to do substitution



Variables and commands

The output of commands can be saved into variables by using back ticks: `command`

variables-sample-script.sh

```
#!/bin/bash
# This is the first script about variables
# Marco A. dd/mm/yyyy

echo
echo Let\'s play with variables...
echo The name of this script is: \"$0\"
echo The first argument passed to this script is: \"$1\"
echo The second argument passed to this script is: \"$2\"
echo

p1from30to39=`egrep '3.?' $1`
p2from30to39=`egrep '3.?' $2`

echo Persons from 30 to 39 years old in \"$1\": $p1from30to39
echo Persons from 30 to 39 years old in \"$2\": $p2from30to39

echo
echo Ciao ciao... See you next time!
echo
```

persone1.txt

```
Nome; Cognome; Età

Mario; Rossi; 35
Mario; Bianchi; 36
Paolo; Rossi; 20
Pippo; Verdi; 44
Pluto; Verdone; 21
Minnie; Rosa; 28

```

Line 3, Column 4 Tab

persone2.txt

```
Nome; Cognome; Età

Pippo; Rossi; 35
Carlo; Bianchi; 36
Pluto; Rossi; 20
Pippo; Giallo; 44
Pasquale; Verdone; 31
Rosalia; Rosa; 28

```

Line 2, Column 1 Tab Size:

Later on, you will understand the meaning of ` `

PART 3 - Scripting — -bash — 160×31

```
iMac-di-User:PART 3 - Scripting marcoautili$
iMac-di-User:PART 3 - Scripting marcoautili$ ./variables-sample-script.sh persone1.txt persone2.txt

Let's play with variables...
The name of this script is: "./variables-sample-script.sh"
The first argument passed to this script is: "persone1.txt"
The second argument passed to this script is: "persone2.txt"

Persons from 30 to 39 years old in "persone1.txt": Mario; Rossi; 35 Mario; Bianchi; 36
Persons from 30 to 39 years old in "persone2.txt": Pippo; Rossi; 35 Carlo; Bianchi; 36 Pasquale; Verdone; 31

Ciao ciao... See you next time!

iMac-di-User:PART 3 - Scripting marcoautili$
```

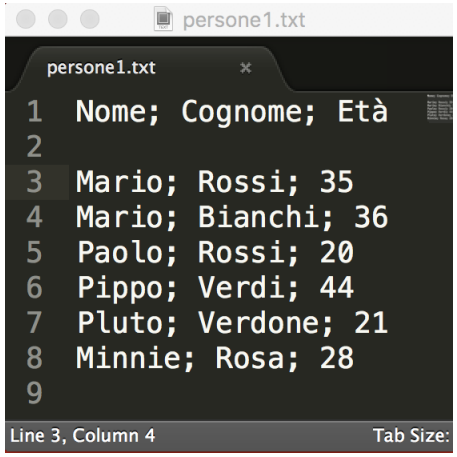
Command substitution

`COMMAND` or ` COMMAND `

is equivalent to

$(COMMAND) or $(COMMAND)

It can be used for saving command output (stdout) in variables rather than printing on the stdout




```
Scripting — -bash — 76×10
[desktop-jisqlks:Scripting marcoautili$
[desktop-jisqlks:Scripting marcoautili$ var4=$( egrep '3.?' persone1.txt )
[desktop-jisqlks:Scripting marcoautili$
[desktop-jisqlks:Scripting marcoautili$ echo $var4
Mario; Rossi; 35 Mario; Bianchi; 36
[desktop-jisqlks:Scripting marcoautili$
[desktop-jisqlks:Scripting marcoautili$ egrep '3.?' persone1.txt
Mario; Rossi; 35
Mario; Bianchi; 36
desktop-jisqlks:Scripting marcoautili$
```

No need to add blank spaces. . .

When saving the output of commands in variables (the end of) lines are removed

Command substitution

- Bash executes the command in a subshell environment and replaces the command substitution with the (standard) output of the command, with any trailing newlines deleted

- The second form `COMMAND` is more or less obsolete for Bash, since it has some trouble with nesting and escaping characters (e.g., "inner" backticks need to be escaped)

- Use $(COMMAND), it is also POSIX ☺

  Just not to forget

  - Portable Operating System Interface (POSIX) is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems
  - POSIX defines the application programming interface (API), along with command line shells and utility interfaces, for software compatibility with variants of Unix and other operating systems

Command substitution preferred syntax

```
echo `echo `ls``      # INCORRECT
echo `echo \`ls\``    # CORRECT
echo $(echo $(ls))    # CORRECT
echo "$(echo "$(ls)")" # CORRECT
```

- $( ) should be the preferred syntax
  - it's clean syntax
  - it's intuitive syntax
  - it's more readable
  - it's nestable
  - its inner parsing is separate

Now let us discuss some peculiar syntactic aspects.
We need to introduce arithmetic expansion first

Arithmetic expansion

- Arithmetic expansion provides a powerful tool for performing (integer) arithmetic operations in scripts
- Arithmetic expansion allows the evaluation of an arithmetic expression and the substitution of the result
- Translating a string into a numerical expression is relatively straightforward using *backticks*, *double parentheses*, or *let* (more on that later…)

```
Scripting — -bash — 76×16
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$ a=$(( 4 * 5 ))
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$ echo $a
20
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$ a=$((4 * 5))
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$ echo $a
20
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$ a=$( (4 * 5) )
-bash: 4: command not found
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$
```

Arithmetic expansion is "slightly different" from *double parentheses.*

More about that later

Now it's time to introduce commands grouping…

Creating an explicit subshell by grouping commands (1/5)

A subshell is a child process launched by a shell (or a shell script)

- each running script is, in effect, a subprocess (child process) of the parent shell

As we will see later, the parentheses are operators "( )" that are indeed a way for grouping commands and execute them in a subshell environment

For know, let us execute that script, see what happens and then keep on comparing this syntax with command substitution and arithmetic expansion

com-group-subshell.sh

```
#!/bin/bash
# This is the first script on commands grouping
# by using the parentheses operators ()
# Marco A. dd/mm/yyyy

(
# Commands group inside parentheses, and therefore a subshell process ...

while [ 1 ]   # Endless loop.
do
    sleep 1
    echo "Subshell running ..."
done

echo "Subshell process after the while [ 1 ] ..."

)
# This script wait "here" for the subshell process...
# Thus, it will run forever, or at least:
# -- until it is terminated by a Ctrl-C
# -- until its subshell process is killed

echo
echo "Finally, I got here... I can now exit!"
echo

exit $?  # End of script

```

Creating an explicit subshell by grouping commands (2/5)

Run the script from a terminal

```
PART 3 - Scripting — bash — 64×29
bash-3.2$
bash-3.2$ ./com-group-subshell.sh
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
^C
bash-3.2$
bash-3.2$
```

The script with PID 42612 launched the subshell with PID 42613

Open another terminal and type:

```
marcoautili — bash — 86×23
bash-3.2$
bash-3.2$ ps -fe | egrep subshell.sh
  501 42612 42434   0 12:03PM ttys000    0:00.00 /bin/bash ./com-group-subshell.sh
  501 42613 42612   0 12:03PM ttys000    0:00.00 /bin/bash ./com-group-subshell.sh
  501 42620 42445   0 12:03PM ttys001    0:00.00 egrep subshell.sh
bash-3.2$
bash-3.2$ ps -fe | egrep subshell.sh
  501 42639 42445   0 12:03PM ttys001    0:00.00 egrep subshell.sh
bash-3.2$
```

After typing Ctrl-C (i.e., SIGINT) on the other terminal to stop the parent process

Note that, the subshell process is also terminated (see the same code with "**trap**" in next slide)

```
bash-3.2$ ps -fe
  UID   PID  PPID   C STIME   TTY           TIME CMD
```

NOTE THAT the usual
"UID PID PPID C STIME TTY TIME CMD"
line printed as first by the ps -ef command has been filtered out by the "grep" command

Creating an explicit subshell by grouping commands (3/5)

Run the script from a terminal

```
PART 3 - Scripting — bash — 64×29
bash-3.2$
bash-3.2$ ./com-group-subshell.sh
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
./com-group-subshell.sh: line 12: 42547 Killed: 9
( while [ 1 ]; do
    sleep 1; echo "Subshell running ...";
done )

Finally, I got here... I can now exit!

bash-3.2$
bash-3.2$
```

Open another terminal and type:

```
marcoautili — bash — 86×23
bash-3.2$
bash-3.2$ ps -fe | egrep subshell.sh
  501 42546 42434   0 11:57AM ttys000    0:00.00 /bin/bash ./com-group-subshell.sh
  501 42547 42546   0 11:57AM ttys000    0:00.00 /bin/bash ./com-group-subshell.sh
  501 42557 42445   0 11:57AM ttys001    0:00.00 egrep subshell.sh
bash-3.2$
bash-3.2$
bash-3.2$ kill -9 42547
bash-3.2$
```

- After killing the subshell process from this terminal…
- the parent process can continue its execution and, hence, it can exit normally

More on commands grouping later…

Creating an explicit subshell by grouping commands (4/5)

com-group-subshell-with-trap.sh

```
1  #!/bin/bash
2  # This is the first script on commands grouping
3  # by using the parentheses operators () and the trap command for handling SIGINT
4  # Marco A. dd/mm/yyyy
5
6  # Handling SIGINT from within the "parent code"
7  trap 'echo " --> HELLO: the parent process handled Ctrl^C"' 2
8
9  (
10 # Commands group inside parentheses, and therefore a subshell process ...
11
12 # Handling SIGINT from within the "subshell code"
13 # trap 'echo " --> HELLO: the subshell process handled Ctrl^C"' 2
14
15 while [ 1 ]   # Endless loop.
16 do
17     sleep 1
18     echo "Subshell running ..."
19 done
20
21 echo "Subshell process after the while [ 1 ] ..."
22
23 )
24 # This script wait "here" for the subshell process...
25 # Thus, it will run forever, or at least:
26 # -- until it is terminated by a Ctrl-C
27 # -- until its subshell process is killed
28
29 echo
30 echo "Finally, I got here... I can now exit!"
31 echo
32
33 exit $?  # End of script
34
```

Using "trap" in the parent …

PART 3 - Scripting — -zsh — 50×24

```
^C
marcoautili@iMac PART 3 - Scripting %
marcoautili@iMac PART 3 - Scripting %
marcoautili@iMac PART 3 - Scripting %
marcoautili@iMac PART 3 - Scripting % ./com-group-
subshell-with-trap.sh
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
^C --> HELLO: the parent process handled Ctrl^C

Finally, I got here... I can now exit!

marcoautili@iMac PART 3 - Scripting %
marcoautili@iMac PART 3 - Scripting %
marcoautili@iMac PART 3 - Scripting %
```

…
- the parent is not terminated
- the subshell process is terminated

Creating an explicit subshell by grouping commands (5/5)

com-group-subshell-with-trap.sh

```
1  #!/bin/bash
2  # This is the first script on commands grouping
3  # by using the parentheses operators () and the trap command for handling SIGINT
4  # Marco A. dd/mm/yyyy
5
6  # Handling SIGINT from within the "parent code"
7  # trap 'echo " --> HELLO: the parent process handled Ctrl^C"' 2
8
9  (
10 # Commands group inside parentheses, and therefore a subshell process ...
11
12 # Handling SIGINT from within the "subshell code"
13 trap 'echo " --> HELLO: the subshell process handled Ctrl^C"' 2
14
15 while [ 1 ]   # Endless loop.
16 do
17     sleep 1
18     echo "Subshell running ..."
19 done
20
21 echo "Subshell process after the while [ 1 ] ..."
22
23 )
24 # This script wait "here" for the subshell process...
25 # Thus, it will run forever, or at least:
26 # -- until it is terminated by a Ctrl-C
27 # -- until its subshell process is killed
28
29 echo
30 echo "Finally, I got here... I can now exit!"
31 echo
32
33 exit $?  # End of script
34
```

Using "trap" in the subshell process …

PART 3 - Scripting — sleep ‹ com-group-subshell-with-trap.sh — 50×24

```
marcoautili@iMac PART 3 - Scripting %
marcoautili@iMac PART 3 - Scripting % ./com-group-
subshell-with-trap.sh
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
^C --> HELLO: the subshell process handled Ctrl^C
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
```

…
- Neither the parent process nor the subshell process is now terminated
- Use kill to terminate them. Try yourself…

Summing up

- Take care when grouping commands (COMMANDs) inside the command substitution $(COMMANDs)
  - this way is wrong: $((COMMANDs))

- It is wrong because it collides with the Double Parentheses syntax for arithmetic expansion, e.g., ((3+4))
  - $((EXPRESSIONs)) is arithmetic expansion (evaluation + substitution), not to be confused with command substitution

- When you need to group commands inside a command substitution, you need to separate the command substitution $(COMMANDs) from the inner commands group (COMMANDs)
  - $((COMMANDs))

Add blank spaces on both sides. SEE NEXT SLIDE...

Command substitution VS arithmetic expansion VS commands grouping

```
Scripting — -bash — 85×16
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$ var=$( (ls) )
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$ echo $var
break-script.sh case-script.sh continue-script.sh double-parentheses-script.sh expr-s
cript.sh functions-return-script.sh functions-script.sh if-script.sh let-script.sh lo
ops-script.sh my-first-script.sh persone1.txt persone2.txt read-input-script.sh selec
t-script.sh variables-exporting-script1.sh variables-exporting-script2.sh variables-s
ample-script.sh
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$ var=$((ls))
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$ echo $var
0
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$
```

Blank spaces → Command substitution for the command "**(ls)**", i.e., "**ls**" executed by a subshell

No blank spaces
→ Arithmetic expansion

N.B.: "**ls**" and "**pippo**" are now considered as unset variables, hence evaluate to 0

More on Double Parentheses later on these slides

```
marcoautili — -bash — 47×6
iMac-di-User:~ marcoautili$
iMac-di-User:~ marcoautili$ var=$((pippo))
iMac-di-User:~ marcoautili$
iMac-di-User:~ marcoautili$ echo $var
0
iMac-di-User:~ marcoautili$
```

Command substitution VS arithmetic expansion VS commands grouping VS …

marcoautili — bash —

```
bash-3.2$
bash-3.2$ var=(ls)
bash-3.2$ echo $var
ls
bash-3.2$ var=(ls -la)
bash-3.2$ echo $var
ls
bash-3.2$ var=(pippo pluto)
bash-3.2$ echo $var
pippo
bash-3.2$ var=('pippo pluto')
bash-3.2$ echo $var
pippo pluto
bash-3.2$ (ls -la)
total 136
drwxr-xr-x+   47 marcoautili  staff    1504 Oct 29 10:23 .
drwxr-xr-x     6 root         admin     192 Sep 29  2019 ..
-rw-r--r--@    1 marcoautili  staff      16 Nov 12  2017 .CF89AA64
-r--------     1 marcoautili  staff       7 Apr 14  2020 .CFUserTextEncoding
-rw-r--r--@    1 marcoautili  staff   12292 Oct 24 11:48 .DS_Store
drwx------@ 9660 marcoautili  staff  309120 Oct 29 10:15 .Trash
-rw-------     1 marcoautili  staff   14812 Oct 19 15:51 .bash_history
drwx------    67 marcoautili  staff    2144 Oct 19 15:51 .bash_sessions
drwxr-xr-x     5 marcoautili  staff     160 Apr 24  2020 .config
```

Command substitution VS arithmetic expansion VS commands grouping VS …

```
bash-3.2$
bash-3.2$ var=(pippo pluto minnie paperino)
bash-3.2$ echo $var
pippo
bash-3.2$ echo ${var}
pippo
bash-3.2$ echo ${var[0]}
pippo
bash-3.2$ echo ${var[1]}
pluto
bash-3.2$ echo ${var[2]}
minnie
bash-3.2$ echo ${var[3]}
paperino
bash-3.2$
bash-3.2$
```

It's an array!! ☺

These are equivalent …

More on that later…

… and it does not end here…

Command substitution VS … STDERR

```
marcoautili — -bash — 71×12
Last login: Tue Oct 11 14:35:21 on ttys000
desktop-jisqlks:~ marcoautili$
desktop-jisqlks:~ marcoautili$
desktop-jisqlks:~ marcoautili$ MV_OUTPUT=$(mv file.txt file-copy.txt)
mv: rename file.txt to file-copy.txt: No such file or directory
desktop-jisqlks:~ marcoautili$
desktop-jisqlks:~ marcoautili$
desktop-jisqlks:~ marcoautili$ echo $MV_OUTPUT

desktop-jisqlks:~ marcoautili$
desktop-jisqlks:~ marcoautili$
desktop-jisqlks:~ marcoautili$
```

ATTENTION: by default, STDERR is redirected to STDOUT. . .

. . . that's why MV_OUTPUT is empty!

```
Scripting — -bash — 85×6
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$ MV_OUTPUT=$(mv file.txt file-copy.txt 2>&1)
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$ echo $MV_OUTPUT
mv: rename file.txt to file-copy.txt: No such file or directory
desktop-jisqlks:Scripting marcoautili$
```

NOTE THAT we are now redirecting mv STDERR to its STDOUT target, because command substitution only catches STDOUT

More on command substitution

```
marcoautili — bash — 145×
bash-3.2$
bash-3.2$ cat pippo-file.txt

pippo-file content1
pippo-file content2
pippo-file content3
pippo-file content4

bash-3.2$
bash-3.2$
bash-3.2$
bash-3.2$ pippovar=$(cat pippo-file.txt)
bash-3.2$
bash-3.2$ echo $pippovar
pippo-file content1 pippo-file content2 pippo-file content3 pippo-file content4
bash-3.2$
bash-3.2$
bash-3.2$ pippovar=$(< pippo-file.txt)
bash-3.2$
bash-3.2$
bash-3.2$ echo $pippovar
pippo-file content1 pippo-file content2 pippo-file content3 pippo-file content4
bash-3.2$
bash-3.2$
bash-3.2$ pippovar=`< pippo-file.txt`
bash-3.2$
bash-3.2$
bash-3.2$ echo $pippovar
pippo-file content1 pippo-file content2 pippo-file content3 pippo-file content4
bash-3.2$
```

Four lines

One line!
Do you remember why?

Compound commands

It's now time to introduce Compound commands

Compound commands are shell programming language constructs

Each construct begins with a reserved word or control operator and is terminated by a corresponding reserved word or operator

- Command Grouping - Ways to group commands
- Looping Constructs - Shell commands for iterative action
- Conditional Constructs - Shell commands for conditional execution

Back to commands grouping

Bash provides two ways to group a list of commands to be executed as a unit

- When commands are grouped, redirections may be applied to the entire command list
  - For example, the output of all the commands in the list may be redirected to a single stream

(list of commands)

- Placing a list of commands between parentheses causes a subshell environment to be created, and each of the commands in list to be executed in that subshell (see previous slides)
- Since the list is executed in a subshell, variable assignments do not remain in effect after the subshell completes

{ list of commands; }

- Placing a list of commands between curly braces causes the list to be executed in the current shell context, no subshell is created
- The semicolon (or newline) following the list is required

Back to commands grouping

Parentheses operators "{ }"

As before but now using the curly brace operators "{ }", which are indeed a way for grouping commands and execute them in the current shell environment

No subshell is created

com-group-no-subshell.sh

com-group-no-subshell.sh 〉 No Selection

```
1  #!/bin/bash
2  # This is the first script on commands grouping
3  # by using the parentheses operators {}
4  # Marco A. dd/mm/yyyy
5
6  {
7  # Commands group inside parentheses {}, and therefore NO subshell process ...
8  while [ 1 ]   # Endless loop.
9  do
10     sleep 1
11     echo "Subshell running ..."
12 done
13 }
14 # No subshell has been created...
15
16 echo
17 echo "I will never got here... I cannot exit!"
18 echo "Please, terminate me with Ctrl-C or kill me :("
19 echo
20
21 exit $?  # End of script
22
```

Back to commands grouping

1. Run the script from a terminal

PART 3 - Scripting — bash — 64×29

```
bash-3.2$
bash-3.2$ ./com-group-no-subshell.sh
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
^C
bash-3.2$
```

3. Ctrl-C to terminate

2. Open another terminal and type:

marcoautili — bash — 86×23

```
bash-3.2$
bash-3.2$ ps -fe | egrep subshell.sh
  501 42763 42434   0 12:20PM ttys000    0:00.00 /bin/bash ./com-group-no-subshell.sh
  501 42770 42445   0 12:20PM ttys001    0:00.00 egrep subshell.sh
bash-3.2$
```

Note that no subshell process has been now created

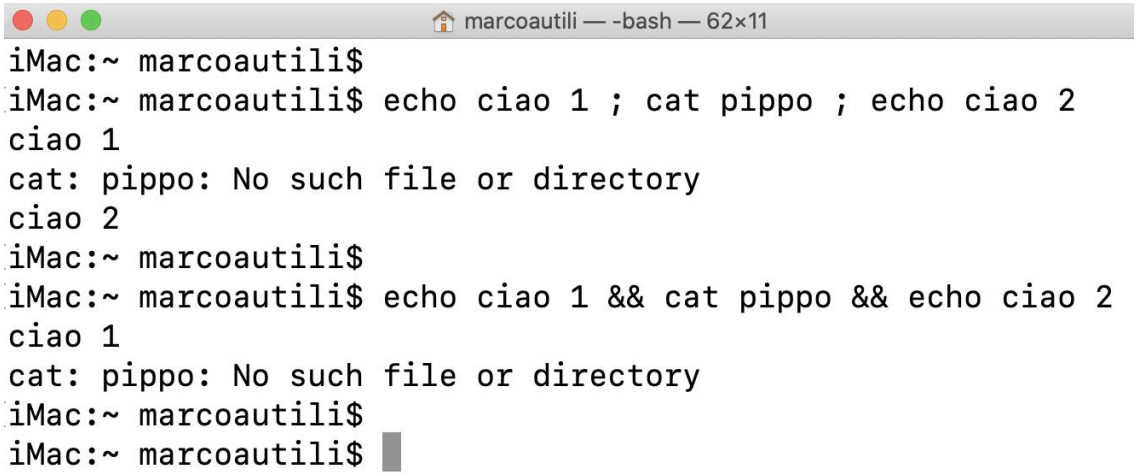
Run commands in sequence (from Part 2)

Semicolons ";"

- run each command, regardless if the preceding ones fail
- com1 ; com2 ; com 3

Double-ampersands "&&"

- run the next command if the preceding command succeeded, and all the commands correctly set exit codes
- com1 && com2 && com 3



Run commands in sequence

Double-pipe "||"

- com2 is executed if, and only if, com1 returns a non-zero exit status
- com1 || com2

Single-ampersands "&"

- If a command is terminated by the control operator "&", the shell executes the command asynchronously in a subshell
- This is known as executing the command in the background, and these are referred to as asynchronous commands
- The shell does not wait for the command to finish, and the return status is 0 (true).

```
$ com1 &
```

- The operators "&&" and "||" have equal precedence, followed by ";" and "&", which have equal precedence

Commands grouping and asynchrony

A shell allows execution of commands, both synchronously and asynchronously

- The shell waits for synchronous commands to complete before accepting more input
- The shell does not wait for asynchronous commands and asynchronous commands continue to execute in "parallel" with the shell while it reads and executes additional commands
- Thus, back to our previous example, explicit subshells can also be used to do "parallel" processing, executing multiple subtasks "simultaneously"

(Note that I have used double quotes for the words: "parallel" and "simultaneously"... You now know the reason ☺)

com-group-asynch-subshell.sh

com-group-asynch-subshell.sh 〉 No Selection

```
1  #!/bin/bash
2  # This is the first script on commands grouping
3  # by using the parentheses operators ()
4  # Marco A. dd/mm/yyyy
5
6  (
7  # Commands group inside parentheses, and therefore a subshell
       process ...
8  while [ 1 ]   # Endless loop.
9  do
10     sleep 1
11     echo "Subshell running ..."
12 done
13 ) &
14 # This script DOES NOT wait "here" for the subshell process...
15
16 i=0
17 while [ $i -lt 4 ]   # Endless loop.
18 do
19     sleep 1
20     echo "Parent script running - Cycle number: $i "
21     ((i++))
22 done
23
24 echo
25 echo "I completed my job... I can now exit!"
26 echo
27
28 exit $?  # End of script
```

Note that I have now added the control operator "&" so to execute the subshell asynchronously

Commands grouping and asynchrony

```
PART 3 - Scripting — -zsh — 86×33
marcoautili@iMac PART 3 - Scripting % ./com-group-asynch-subshell.sh
Subshell running ...
Parent script running - Cycle number: 0
Subshell running ...
Parent script running - Cycle number: 1
Subshell running ...
Parent script running - Cycle number: 2
Subshell running ...
Parent script running - Cycle number: 3

I completed my job... I can now exit!

marcoautili@iMac PART 3 - Scripting % Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
Subshell running ...
```

Importantly, the **parent script** and the **subshell** are interleaving their execution

After performing the four cycles, the parent script terminates its execution

```
marcoautili — bash — 99×23
bash-3.2$
bash-3.2$ ps -fe | egrep subshell.sh
  501 43607 42434   0  3:22PM ttys000    0:00.00 /bin/bash ./com-group-asynch-subshell.sh
  501 43608 43607   0  3:22PM ttys000    0:00.00 /bin/bash ./com-group-asynch-subshell.sh
  501 43614 42445   0  3:22PM ttys001    0:00.00 egrep subshell.sh
bash-3.2$
bash-3.2$
bash-3.2$ ps -fe | egrep subshell.sh
  501 43608     1   0  3:22PM ttys000    0:00.00 /bin/bash ./com-group-asynch-subshell.sh
  501 43623 42445   0  3:22PM ttys001    0:00.00 egrep subshell.sh
bash-3.2$
bash-3.2$
bash-3.2$ kill -9 43608
bash-3.2$
```

After the parent script termination, the subshell process has a different PPID. Its is now "**1**" ! We will be back on that later…

Killing the subshell process…

Note that, typing Ctrl-C from this shell, now, does not have any effect on the subshell

Exporting Variables

- varname=value
  - the scope of the variable varname is restricted to the shell or script and is not available to any other process. Used for, e.g., loop variables, temporary variables, etc.

- export varname=value
  - the variable varname is available to *any process* you run from that shell process. If you want a process to make use of this variable, use export, and run the process from that shell

  - Note that exporting a variable does not make it available to parent processes. That is, specifying and exporting a variable from within a child process does not make it available in the process that launched it

Exporting Variables

variables-exporting-script1.sh

```
1  #!/bin/bash
2  # Script to set the environment variable PATH when needed
3  # Marco A. dd/mm/yyyy
4
5  export NEW_LOCATION=/usr/local/mynewbin
6  echo $NEW_LOCATION
7  export PATH=$PATH:$NEW_LOCATION
8  echo The content of the variable PATH is: $PATH
9
10
```

Scripting — -bash — 113×12

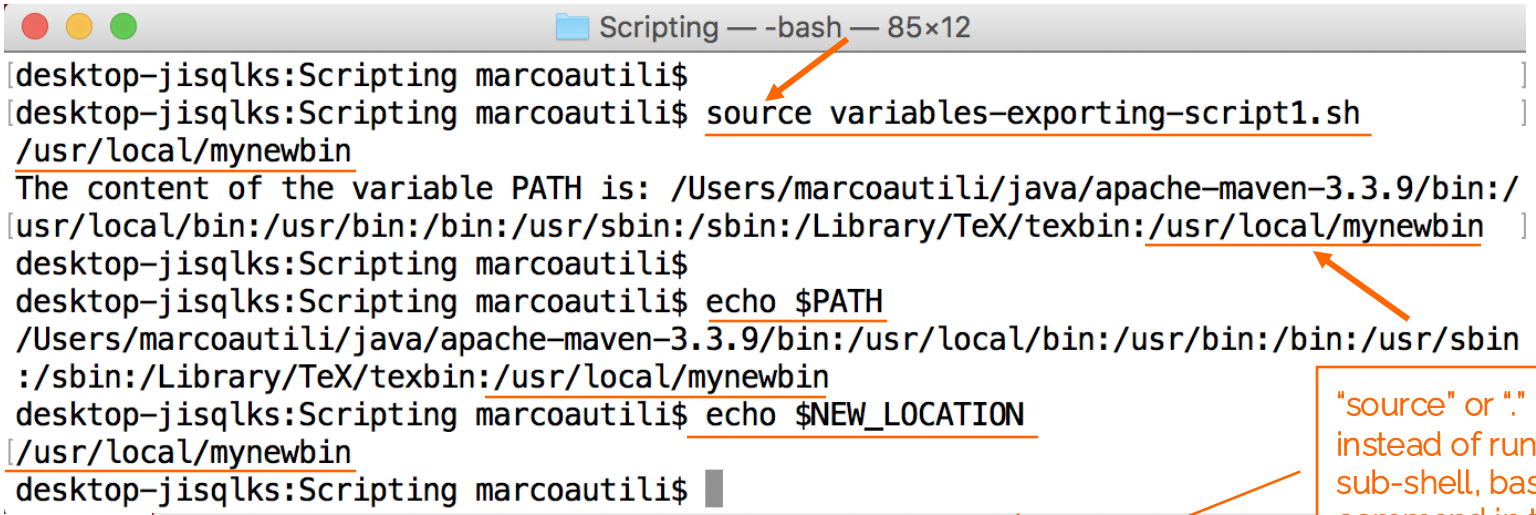
```
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$ ./variables-exporting-script1.sh
/usr/local/mynewbin
The content of the variable PATH is: /Users/marcoautili/java/apache-maven-3.3.9/bin:/usr/local/bin:/usr/bin:/bin:
/usr/sbin:/sbin:/Library/TeX/texbin:/usr/local/mynewbin
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$ echo $PATH
/Users/marcoautili/java/apache-maven-3.3.9/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Library/TeX/texbin
desktop-jisqlks:Scripting marcoautili$
```

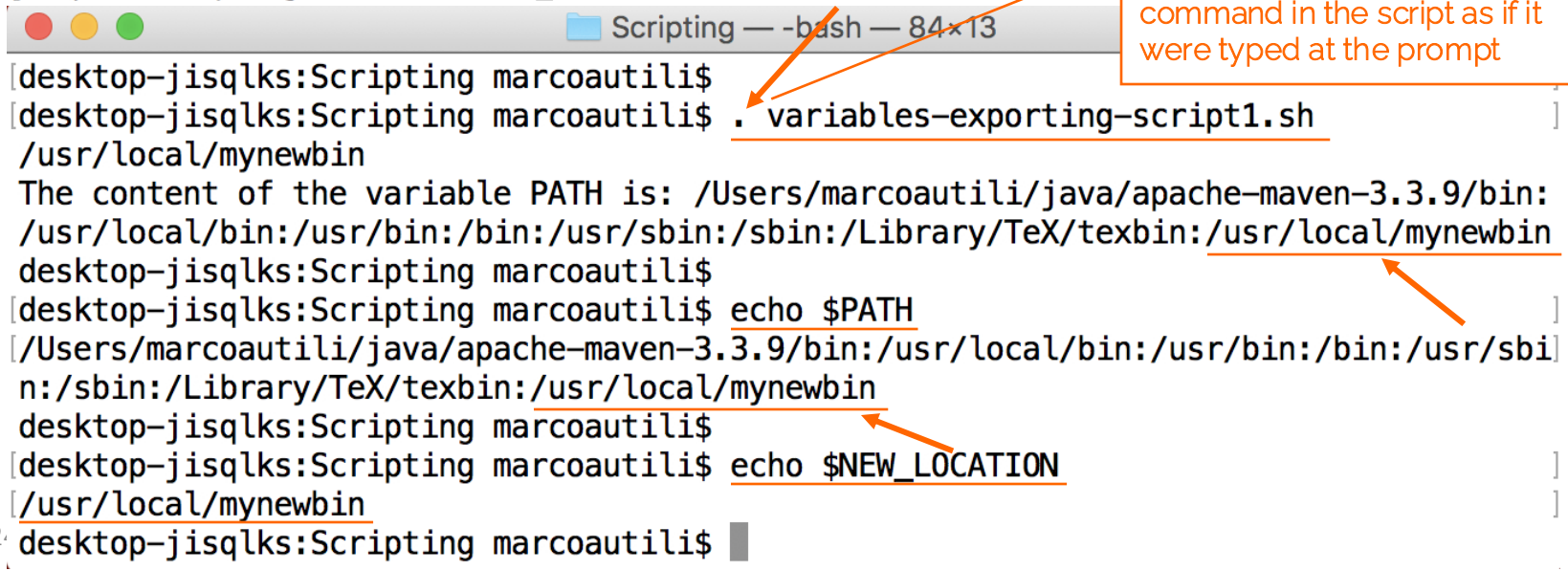
It does not work since

- "export" exports the variable assignment to sub-shells, i.e., shells which are started as child processes of the shell containing the export directive
- the problem is that the command-line environment is the parent of the script's shell, so it does not see the variable assignment
- ... SEE NEXT SLIDE

Exporting Variables

- You can use the source bash command or "." to execute the script commands in the current shell environment and it will work…





"source" or "." → instead of running the script in a sub-shell, bash will execute each command in the script as if it were typed at the prompt

Reading input

read <var1> ... <varn>

```
Scripting — -bash — 62×18
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$ ./read-input-script.sh

What's your name?
Marco

Hi Marco !!

What is your nikname?  Mik
What is your password?
Mik
My password is 0000

Mik ... You should never print the password!!!

Ciao ciao... See you next time!

desktop-jisqlks:Scripting marcoautili$
```

read-input-scrip

read-input-script.sh

```
1  #!/bin/bash
2  # This is the first script for readi
3  # Marco A. dd/mm/yyyy
4
5  echo
6  echo What\'s your name?
7  read varname
8  echo
9  echo Hi $varname !!
10 echo
11
12 read -p 'What is your nikname?  ' nicknamevar
13
14 read -sp 'What is your password?  ' passvar
15 echo
16 echo $nicknamevar
17
18 echo "My password is $passvar"
19 echo
20 echo "$nicknamevar ... You should never print the password!!!"
21
22 echo
23 echo Ciao ciao... See you next time!
24 echo
25
```

Line 16, Column 18 Tab Size: 4 Shell Script (Bash)

Useful options are:

- -p to specify a prompt (e.g., "What is your nikname? ")
- -s to make the input silent

See the man page for further options

Let

let <arithmetic expression>

let is a Bash built-in function that allows us to do simple arithmetic

let-script.sh

```
#!/bin/bash
# Arithmetic expression with let
# Marco A. dd/mm/yyyy

echo

# No spaces are allowed if double quotes are not used
let a=5+4
let b=$a*4
# Spaces are allowed if double quotes are used
let "c = 5/4"

let d=50*4
let "f = $a % 6"

let a++

let g=50*-4
let g=g+5

echo "a = $a; b = $b; c = $c; d = $d; f = $f; g = $g;"

...o... See you next time!
```

Scripting — -bash — 56×9

```
[desktop-jisqlks:Scripting marcoautili$
[desktop-jisqlks:Scripting marcoautili$ ./let-script.sh

a = 10; b = 36; c = 1; d = 200; f = 3; g = -195;

Ciao ciao... See you next time!

[desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$
```

Expr

expr is similar to let

- Instead of saving the result to a variable, expr prints the answer to stdout
- Differently from let, quotes are not required…
  - However, spaces between the terms of the expression must be used
- Commonly, expr is used within a command substitution to save the output to a variable

Expr

Only integer arithmetic (no floating point) by using the:

- Unix expr utility
- Bash $[N op M] syntax

Try also
echo $[1 + 2] and var=$[4 + 2]

expr-script.sh

```
#!/bin/bash
# Arithmetic expression with expr
# Marco A. 24/10/2016

echo

# Spaces must be used
expr 5 + 4

expr 5+ 4
expr 5 + 4
expr "5 + 4"

# The tiimes operator "*"" must be escaped
expr 5 \* 12

xpr 5 * 12

xpr $1 + 1000

cho a=$(expr $1 + 2000)
cho a=$( expr $1 + 2000 )

cho
cho Ciao ciao... See you next time!
cho
```

Scripting — -bash — 62×16

```
desktop-jisqlks:Scripting marcoautili$ ./expr-script.sh  88

9
expr: syntax error
9
5 + 4
60
expr: syntax error
1088
a=2088
a=2088

Ciao ciao... See you next time!

desktop-jisqlks:Scripting marcoautili$
```

Double Parentheses

double-parentheses-script.sh

```bash
#!/bin/bash
# Arithmetic  with double parentheses
# Marco A. dd/mm/yyyy

echo

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

echo
echo Ciao ciao... See you next time!
echo
```

Line 4, Column 1 Tab Size: 4

- The ((compound command (aka Double Parentheses) is similar to the let command, construct permits arithmetic expansion and evaluation
- In its simplest form, a=$((5 + 3)) sets a to 8
- However, it is also a mechanism for allowing C-style manipulation of variables in Bash, for example, ((var++)) or a=$((var++))

Scripting — -bash — 71×14

```
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$ ./double-parentheses-script.sh

9
8
11
12
13
16
20

Ciao ciao... See you next time!

desktop-jisqlks:Scripting marcoautili$
```

Floating point arithmetic with AWK

- AWK operators and functions
  - raising a number to a power (**,^)
  - the square root (sqrt(x))
  - the natural logarithm (log(x))
  - the sine function (sin(x))
  - cosine function (cos(x))
  - the arctangent (atan2(y,x))

As you know, we will not study AWK

This is just a plus

- AWK has the advantage that it is very old and is almost always available on Unix and Unix-like systems

* http://math-blog.com/floating-point-arithmetic-in-the-bourne-again-shell-bash/

Floating point arithmetic

- You can also use
  - bc arbitrary precision numeric processing language
    - syntax is similar to C, but differs in many part
  - Perl programming language
  - Python programming language
  - Ruby programming language

We will not study any of these languages

Floating point arithmetic

float-numbers-script.sh

```
#!/bin/bash
# Arithmetic with float numbers
# Marco A. dd/mm/yyyy
echo

#It doesn't work because bash doesn't support floating point
let var0=2.8+2.6
echo var0 = $var0

#It doesn't work because bash doesn't support floating point
let var1=2,8+2,6
echo var1 = $var1

# Using bc
var2=$(bc <<< "2.5+2.5")
echo var2 = $var2

# Using awk
var3=$(awk "BEGIN {print 2.5+2.5; exit}")
echo var3 = $var3

# Using perl
var4=$(perl -e "print 2.5+2.5")
echo var3 = $var4

# Using python
var5=$(python -c "print 2.5+2.5")
echo var5 = $var5
echo
```

Comma "," has another meaning in this case: it's a command separator

This command is equivalent to following three commands:

- let var1=2
- let 8+2
- let 6

As mentioned in Part 2, this another type of redirection, called Here String.
The string after <<< is passed as input to bc.
More on that later...

Scripting — -bash — 45×16

```
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$ ./floa
t-numbers-script.sh

./float-numbers-script.sh: line 8: let: var0=
2.8+2.6: syntax error: invalid arithmetic ope
rator (error token is ".8+2.6")
var0 =
var1 = 2
var2 = 5.0
var3 = 5
var3 = 5
var5 = 5.0

desktop-jisqlks:Scripting marcoautili$
```

Floating point arithmetic

Scripting — -bash — 71×17

```
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$ ./float-numbers-bis-script.sh

3.3
-1.1
2.42
2.459014118
.50000000000000000000
.52380952380952380952
3
./float-numbers-bis-script.sh: line 26: 2^8: command not found

10
256
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$
```

Remember that this is interpreted as a command (because of blank spaces)

Pay attention!

It is the bitwise XOR

... next slide

float-numbers-bis-script.sh

```
1  #!/bin/bash
2  # Arithmetic with float numbers
3  # Marco A. dd/mm/yyyy
4  echo
5
6  echo 1.1 + 2.2 | bc -l
7
8  echo 1.1 - 2.2 | bc -l
9
10 # This is wrong!
11 # echo 1.1 * 2.2 | bc -l
12
13 # This is right!
14 echo 1.1 \* 2.2 | bc -l
15
16 # This is right!
17 echo '1.1 * 2.23546738' | bc -l
18
19 echo '1.1 / 2.2' | bc -l
20
21 myvar=`echo '1.1 / 2.1' | bc -l`
22 echo $myvar
23
24 echo $[1 + 2]
25
26 echo $( (2^8) )
27
28 echo $((2^8))
29
30 echo $((2**8))
```

Bitwise operators (a brief account of)

<< bitwise left shift

- (multiplies by 2 for each shift position)

<<= left-shift equal

- multiply by 2 for each shift position
- for example, let "var <<= 3" results in var left-shifted 3 bits (i.e., multiplied by 8)

>> bitwise right shift

- divide by 2 for each shift position

>>= right-shift equal (inverse of <<=)

& bitwise AND

&= bitwise AND-equal

| bitwise OR

|= bitwise OR-equal

~ bitwise NOT

^ bitwise XOR

^= bitwise XOR-equal

http://www.tldp.org/LDP/abs/html/ops.html#FTN.AEN4242

Bitwise operators (a brief account of)

```
marcoautili — bash –
bash-3.2$
bash-3.2$ let a=4
bash-3.2$ echo "obase=2;$a" | bc
100
bash-3.2$ let a=9
bash-3.2$ echo "obase=2;$a" | bc
1001
bash-3.2$ let a=7
bash-3.2$ echo "obase=2;$a" | bc
111
bash-3.2$
bash-3.2$ echo $(($a << 1))
14
bash-3.2$ echo $(($a << 2))
28
bash-3.2$ echo $a
7
bash-3.2$ echo $(($a <<= 2))
bash: 7 <<= 2: attempted assignment to non-variable (error token is "<<= 2")
bash-3.2$ echo $((a <<= 2))
28
bash-3.2$ echo $a
28
bash-3.2$ echo $((a >>= 2))
7
bash-3.2$ echo "obase=2;$a" | bc
111
bash-3.2$ echo $((a <<= 2))
28
bash-3.2$ echo "obase=2;$a" | bc
11100
bash-3.2$
```

Note that $a is not modified here

Bitwise XOR

- The truth table for the XOR operator is:
  either one or the other
- First, the operands are being converted to their respective binary forms

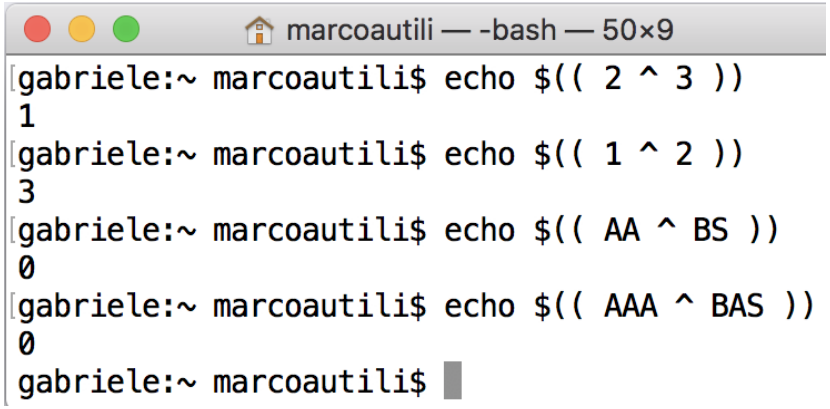
  1 → 001

  2→ 010

  1^2→011 = 3

A	B	A^B
0	0	0
0	1	1
1	0	1
1	1	0

- In general, if the operands are in the form of n ^ (n+1) and also n is an even number, then the result is always 1
- The xor operation between two strings of arbitrary length is undefined



http://tldp.org/LDP/abs/html/ops.html

http://www.unix.com/faq-submission-queue/179027-analysis-bitwise-xor.html

Bitwise operators

- Bitwise operators seldom make an appearance in shell scripts
- Their chief use seems to be manipulating and testing values read from ports or sockets
- "Bit flipping" is more relevant to compiled languages, such as C and C++, which provide direct access to system hardware
- However, see Vladz's ingenious use of bitwise operators in his base64.sh (Example A-54) script

http://www.tldp.org/LDP/abs/html/ops.html
http://www.tldp.org/LDP/abs/html/contributed-scripts.html#BASE64

Variable length

${#variable} – the length of the variable in terms of numbers of characters

a='Marco'

${#a} is equal to 5

b=1024

${#b} is equal to 4

Back to “here string” redirection

[n]<<< string

- It is another type of redirection called Here String
- The string undergoes tilde expansion, parameter and variable expansion, command substitution, arithmetic expansion, and quote removal. Pathname expansion and word splitting are not performed. The result is supplied as a single string, with a newline appended, to the command on its standard input (or file descriptor n if n is specified)

e.g., ~ → /Users/marcoautili

```
marcoautili — -zsh — 80×24
marcoautili@iMac ~ %
marcoautili@iMac ~ % sed 's/a/b/g' <<< "aaa"
bbb
marcoautili@iMac ~ % myname='Marco'
marcoautili@iMac ~ % mysurname='Autili'
marcoautili@iMac ~ % sed 's/a/@/g' <<< "$myname $mysurname"
M@rco Autili
marcoautili@iMac ~ % sed 's/a/@/g' <<< "$myname $mysurname ~./"
M@rco Autili ~./
marcoautili@iMac ~ % sed 's/a/@/g' <<< "$myname $mysurname $((2+2))"
M@rco Autili 4
marcoautili@iMac ~ %
```

Back to “here string” redirection

You can also feed a string to a command's stdin as follow: echo "$string" | command

However, in bash, introducing a pipe means the individual commands are run in subshells

bash

```
marcoautili — bash — 80×24
bash-3.2$
bash-3.2$ first="mondo" ; second="ciao"
bash-3.2$ echo "hello world" | read first second
bash-3.2$ echo $second $first
ciao mondo
bash-3.2$
```

A curiosity: in zsh shell it behaves differently…
See next slide

zsh

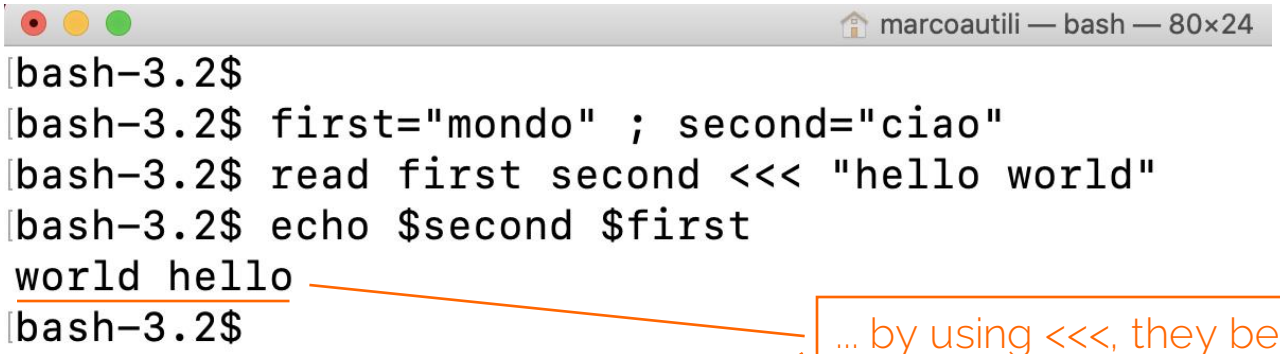
```
marcoautili — -zsh — 64×16
marcoautili@iMac ~ %
marcoautili@iMac ~ % first="mondo" ; second="ciao"
marcoautili@iMac ~ % echo "hello world" | read first second
marcoautili@iMac ~ % echo $second $first
world hello
marcoautili@iMac ~ % first="mondo" ; second="ciao"
marcoautili@iMac ~ % echo "hello world" | (read first second)
marcoautili@iMac ~ % echo $second $first
ciao mondo
marcoautili@iMac ~ %
```

If you force an explicit subshell,
then it will behave as in the bash shell

Back to “here string” redirection

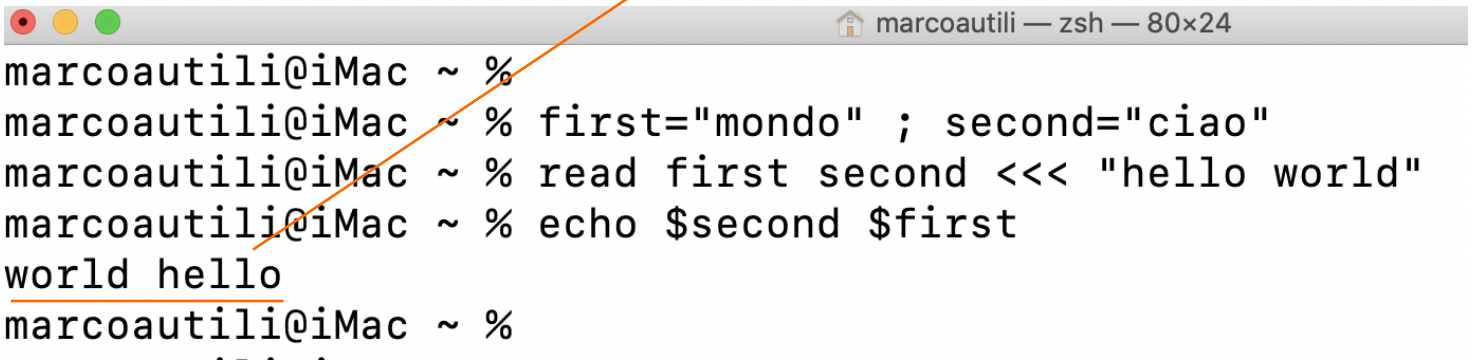
Here string can be of help in this case…

bash



… by using <<<, they behave the same ☺

zsh



If statements

```
Scripting — -bash — 97×31
desktop-jisqlks:Scripting marcoautili$ ./if-script.sh persone1.txt 30to39
persone1.txt 30to39
persone1.txt 30to39

You want to select the persons "30to39" listed in the file "persone1.txt"

The persons from 30 to 39 years old in "persone1.txt" are: Mario; Rossi; 35 Mario; Bianchi; 36

Ciao ciao... See you next time!

desktop-jisqlks:Scripting marcoautili$ ./if-script.sh persone1.txt 40to49
persone1.txt 40to49
persone1.txt 40to49

You want to select the persons "40to49" listed in the file "persone1.txt"

The persons from 40 to 49 years old in "persone1.txt" are: Pippo; Verdi; 44

Ciao ciao... See you next time!

desktop-jisqlks:Scripting marcoautili$ ./if-script.sh persone1.txt 50to59
persone1.txt 50to59
persone1.txt 50to59

You want to select the persons "50to59" listed in the file "persone1.txt"

You can choose either 30to39 or 40to49 !!!

Ciao ciao... See you next time!

desktop-jisqlks:Scripting marcoautili$
```

persone1.txt

```
1  Nome; Cognome; Età
2
3  Mario; Rossi; 35
4  Mario; Bianchi; 36
5  Paolo; Rossi; 20
6  Pippo; Verdi; 44
7  Pluto; Verdone; 21
8  Minnie; Rosa; 28
9
```

Line 1, Column 1 Tab

if-script.sh

```
1  #!/bin/bash
2  # Exercising with the if statement
3  # Marco A. 22/09/2016
4
5  echo $*
6  echo $@
7  echo
8
9  echo You want to select the persons \"$2\" listed in the file \"$1\"
10
11 if [ {#$2} == 0 ]
12 then
13     echo ciaoooo
14     exit
15 else
16     echo
17 fi
18
19 p1from30to39=`egrep '3.?' $1`
20 p1from40to49=`egrep '4.?' $1`
21
22
23 if [ $2 == '30to39' ]
24 then
25     echo The persons from 30 to 39 years old in \"$1\" are: $p1from30to39
26 else
27     if [ $2 == '40to49' ]
28     then
29         echo The persons from 40 to 49 years old in \"$1\" are: $p1from40to49
30     else
31         echo You can choose either 30to39 or 40to49 !!!
32     fi
33 fi
34
35 echo
36 echo Ciao ciao... See you next time!
```

If statements and test command

- Within if statements the conditional part between [ ] refers to the test command

(*EXPRESSION* **)**	*EXPRESSION* is true
! *EXPRESSION*	*EXPRESSION* is false
EXPRESSION1 **-a** *EXPRESSION2*	both *EXPRESSION1* and *EXPRESSION2* are true
EXPRESSION1 **-o** *EXPRESSION2*	either *EXPRESSION1* or *EXPRESSION2* is true
-n *STRING*	the length of *STRING* is nonzero
STRING	equivalent to **-n** *STRING*
-z *STRING*	the length of *STRING* is zero
STRING1 **=** *STRING2*	the strings are equal
STRING1 **!=** *STRING2*	the strings are not equal
INTEGER1 **-eq** *INTEGER2*	*INTEGER1* is equal to *INTEGER2*
INTEGER1 **-ge** *INTEGER2*	*INTEGER1* is greater than or equal to *INTEGER2*
INTEGER1 **-gt** *INTEGER2*	*INTEGER1* is greater than *INTEGER2*

If statements and test command

INTEGER1 **-le** *INTEGER2*	*INTEGER1* is less than or equal to *INTEGER2*
INTEGER1 **-lt** *INTEGER2*	*INTEGER1* is less than *INTEGER2*
INTEGER1 **-ne** *INTEGER2*	*INTEGER1* is not equal to *INTEGER2*
FILE1 **-ef** *FILE2*	*FILE1* and *FILE2* have the same device and inode numbers
FILE1 **-nt** *FILE2*	*FILE1* is newer (modification date) than *FILE2*
FILE1 **-ot** *FILE2*	*FILE1* is older than *FILE2*
-b *FILE*	*FILE* exists and is block special
-c *FILE*	*FILE* exists and is character special
-d *FILE*	*FILE* exists and is a directory
-e *FILE*	*FILE* exists
-f *FILE*	*FILE* exists and is a regular file
-g *FILE*	*FILE* exists and is set-group-ID
-G *FILE*	*FILE* exists and is owned by the effective group ID

If statements and test command

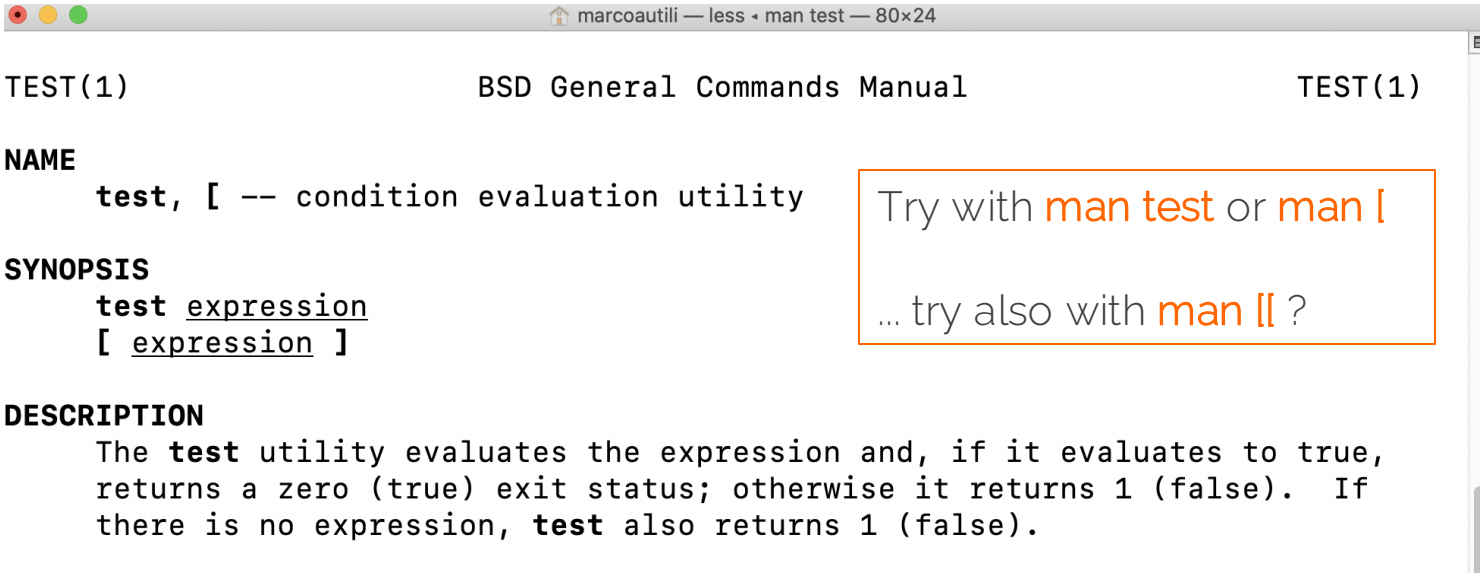
-h *FILE*	*FILE* exists and is a symbolic link (same as **-L**)
-k *FILE*	*FILE* exists and has its sticky bit set
-L *FILE*	*FILE* exists and is a symbolic link (same as **-h**)
-O *FILE*	*FILE* exists and is owned by the effective user ID
-p *FILE*	*FILE* exists and is a named pipe
-r *FILE*	*FILE* exists and read permission is granted
-s *FILE*	*FILE* exists and has a size greater than zero
-S *FILE*	*FILE* exists and is a socket
-t *FD*	file descriptor *FD* is opened on a terminal
-u *FILE*	*FILE* exists and its set-user-ID bit is set
-w *FILE*	*FILE* exists and write permission is granted
-x *FILE*	*FILE* exists and execute (or search) permission is granted

What is the difference between test, [and [[?

- The [(aka test) command and
  the [[...]] test construct are both used to evaluate expressions

Note the difference:
- a command versus
- a construct/keyword (or more correctly, compound command)

- [[...]] works only in the Korn shell (where it originates), Bash, Zsh, and recent versions of Yash and busybox sh (if enabled at compilation time), and is more powerful
- [and test are POSIX utilities (generally builtin)
- POSIX doesn't specify the [[...]] construct (which has a specific syntax with significant variations between implementations) though allows shells to treat [[as a keyword



Try with man test or man [

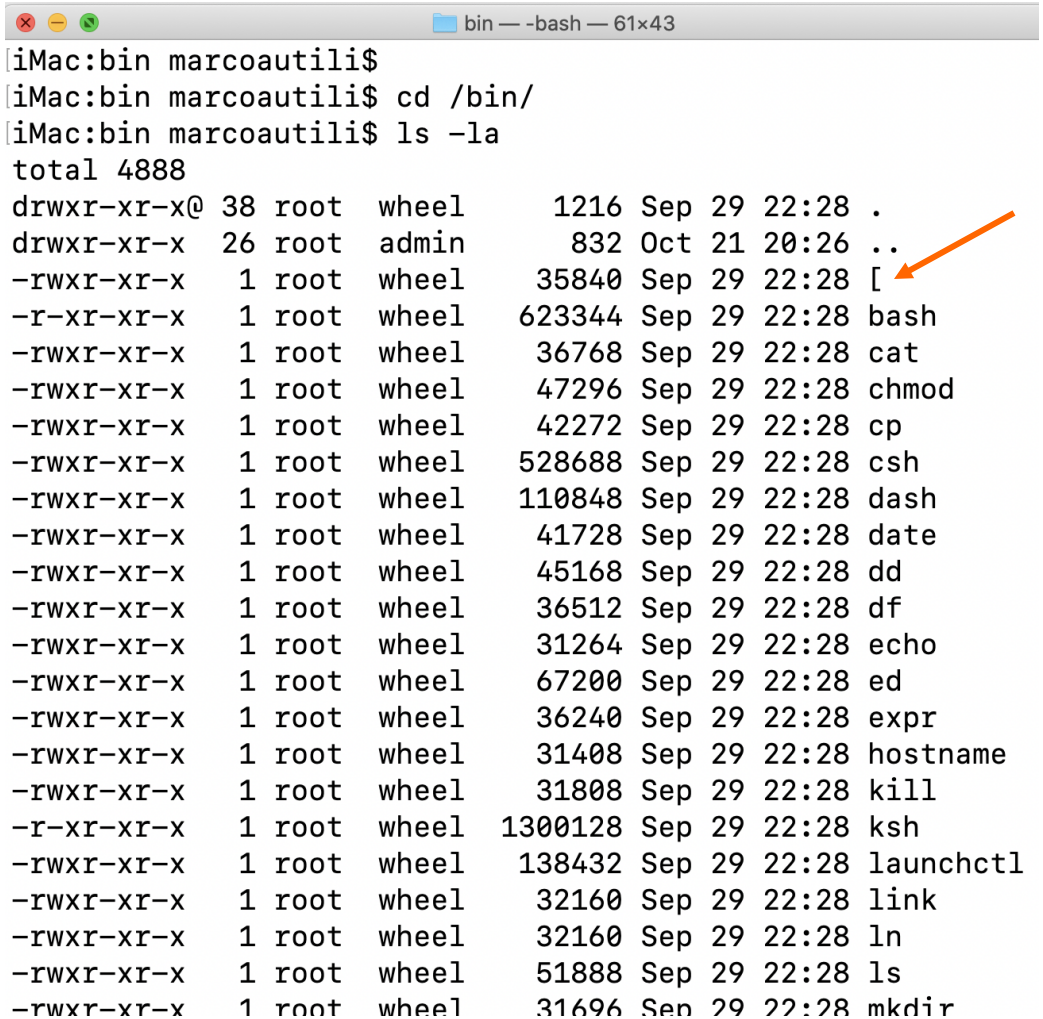
... try also with man [[?

What is the difference between test, [and [[?

Summing up

- test implements the old, portable syntax of the command. In almost all shells (the oldest Bourne shells are the exception), [is a synonym for test (but requires a final argument of])
- Although all modern shells have built-in implementations of [, there usually still is an external executable of that name, e.g., /bin/[(see next slide)
- POSIX defines a mandatory feature set for [, but almost every shell offers extensions to it. So, if you want portable code, you should be careful not to use any of those extensions
- [[is a new, improved version of it, and it is a keyword rather than a program. This makes it easier to use, as shown in next slide
- Although [and [[have much in common and share many expression operators like "-f", "-s", "-n", and "-z", there are some notable differences...

External executable /bin/[



[and [[notable differences

Feature	new test [[	old test [	Example
string comparison	>	\> (*)	`[[ a > b ]] || echo "a does not come after b"`
	<	\< (*)	`[[ az < za ]] && echo "az comes before za"`
	= (or ==)	=	`[[ a = a ]] && echo "a equals a"`
	!=	!=	`[[ a != b ]] && echo "a is not equal to b"`
integer comparison	-gt	-gt	`[[ 5 -gt 10 ]] || echo "5 is not bigger than 10"`
	-lt	-lt	`[[ 8 -lt 9 ]] && echo "8 is less than 9"`
	-ge	-ge	`[[ 3 -ge 3 ]] && echo "3 is greater than or equal to 3"`
	-le	-le	`[[ 3 -le 8 ]] && echo "3 is less than or equal to 8"`
	-eq	-eq	`[[ 5 -eq 05 ]] && echo "5 equals 05"`
	-ne	-ne	`[[ 6 -ne 20 ]] && echo "6 is not equal to 20"`
conditional evaluation	&&	-a (**)	`[[ -n $var && -f $var ]] && echo "$var is a file"`
	||	-o (**)	`[[ -b $var || -c $var ]] && echo "$var is a device"`
expression grouping	(...)	\(... \) (**)	`[[ $var = img* && ($var = *.png || $var = *.jpg) ]] && echo "$var starts with img and ends with .jpg or .png"`
Pattern matching	= (or ==)	(not available)	`[[ $name = a* ]] || echo "name does not start with an 'a': $name"`
RegularExpression matching	=~	(not available)	`[[ $(date) =~ ^Fri\ ...\ 13 ]] && echo "It's Friday the 13th!"`

(*) This is an extension to the POSIX standard; some shells may have it, others may not.

(**) The -a and -o operators, and (...) grouping, are defined by POSIX but only for strictly limited cases, and are marked as deprecated. Use of these operators is discouraged; you should use multiple [commands instead:

- `if [ "$a" = a ] && [ "$b" = b ]; then ...`
- `if [ "$a" = a ] || { [ "$b" = b ] && [ "$c" = c ];}; then ...`

[and [[notable differences

- Special primitives that [[is defined to have, but [may be lacking (depending on the implementation)

Description	Primitive	Example
entry (file or directory) exists	`-e`	`[[ -e $config ]] && echo "config file exists: $config"`
file is newer/older than other file	`-nt` / `-ot`	`[[ $file0 -nt $file1 ]] && echo "$file0 is newer than $file1"`
two files are the same	`-ef`	`[[ $input -` `ef $output ]] && { echo "will not overwrite input file: $input"; exit 1; }`
negation	`!`	`[[ ! -u $file ]] && echo "$file is not a setuid file"`

Back on that in PART 4

If Elif Else

```
if [ <some test> ]
then
        <commands>
elif [ <some test> ]
then
        <different commands>
else
        <other commands>
fi
```

Case Statements

```
case <variable> in

<pattern 1>)

        <commands>

        ;;

<pattern 2>)

        <other commands>

        ;;

esac
```

case-script.sh

```
 4
 5  echo
 6  echo What\'s your name?
 7  read varname
 8  echo
 9  echo Hi $varname !!
10  echo
11
12  read -p 'What is your nikname?  ' nicknamevar
13
14  read -p 'Do you want to print your "name" or your "nikname"?  ' answer
15  echo
16
17  case $answer in
18   name)
19      echo "The name you have entered is $varname
20      ;;
21
22  nikname)
23      echo "The nikname you have entered is $nicknamevar
24      ;;
25
26      *)
27      echo "Please, enter \"name\" or \"nikname\""
28      ;;
29  esac
30
31  echo
32  echo Ciao ciao... See you next time!
33  echo
34
```

Line 22, Column 7 Tab Size: 4 Shell S

While, Until, For

```
while [ <some test> ]
do
        <commands>
done
```

```
until [ <some test> ]
do
        <commands>
done
```

```
for var in <list>
do
        <commands>
done
```

While, Until, For

```
Scripting — -bash — 58×30
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$ ./loops-script.sh

1
2
3
--- reverse
4
3
2
1
Pippo
Pluto
Paperino
1
2
3
File ./case-script.sh copied!
File ./double-parentheses-script.sh copied!
File ./expr-script.sh copied!
File ./if-script.sh copied!
File ./let-script.sh copied!
File ./loops-script.sh copied!
File ./my-first-script.sh copied!
File ./persone1.txt copied!
File ./persone2.txt copied!
File ./read-input-script.sh copied!
File ./variables-sample-script.sh copied!
```

loops-script.sh

```
 7  counter=1
 8  while [ $counter -le 3 ]
 9  do
10      echo $counter
11      ((counter++))
12  done
13
14  echo --- reverse
15
16  until [ $counter -lt 1 ]
17  do
18      echo $counter
19      ((counter--))
20  done
21
22  names='Pippo Pluto Paperino'
23  for name in $names
24  do
25      echo $name
26  done
27
28  # Range
29  for value in {1..3}
30  do
31      echo $value
32  done
33
34  mkdir ./copydir
35  # For all file having an extension specified
36  for value in ./*.*
37  do
38      cp $value ./copydir/copy-of-$( basename $value )
39      echo File $value copied!
40  done
41
42  echo
43  echo Ciao ciao... See you next time!
44  echo
```

In the current folder

See man basename

The prefix added to the copied files

Break and Continue

- Built-in functions of the shell that, within a loop, allow for:

Break [n] - exit from a for, while, until, or select loop

- If n is supplied, the n^{th} enclosing loop is exited
- The return status is zero unless n is not greater than or equal to 1

Continue [n] - resume the next iteration of an enclosing for, while, until, or select loop

- If n is supplied, the execution of the n^{th} enclosing loop is resumed
- The return status is zero unless n is not greater than or equal to 1

n must be greater than or equal to 1

n is 1 by default

Break

```
marcoautili — bash — 60×24
bash-3.2$
bash-3.2$ ./break-script.sh
0
1
2
3
4
5
--- Let's now try break with the option n ---
1 0
1 1
1 2
1 3
1 4
2 0

Ciao ciao... See you next time!

bash-3.2$
```

Note that this is the 2nd enclosing loop

Try with n equal to 1

break-script.sh

```
1  #!/bin/bash
2  # This is the first script showing
3  # the break statement
4  # Marco A. dd/mm/yyyy
5
6  a=0
7  while [ $a -lt 10 ]
8  do
9     echo $a
10    if [ $a -eq 5 ]
11    then
12       break
13    fi
14    a=$(( $a + 1 ))
15 done
16
17 echo --- Let\'s now try break with the option n ---
18
19 for var1 in 1 2 3
20 do
21    for var2 in 0 1 2 3 4
22    do
23       if [ $var1 -eq 2 -a $var2 -eq 1 ]
24       then
25          break 2
26       else
27          echo "$var1 $var2"
28       fi
29    done
30 done
31
32 echo
33 echo Ciao ciao... See you next time!
34 echo
```

Continue

continue-script.sh

```
#!/bin/bash
# This is the first script showing
# the continue statement
# Marco A. dd/mm/yyyy

i=0
while [ $i -lt 5 ]
do
    echo "i = $i"
    i=`expr $i + 1`

    if [ $i -gt 2 ]
    then
        continue
    fi
    echo "Just after the continue command..."
done
```

Line 6, Column 2 Tab Size: 4 Shell Script (Bash)

Scripting — -bash — 63×10

```
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$ ./continue-script.sh
i = 0
Just after the continue command...
i = 1
Just after the continue command...
i = 2
i = 3
i = 4
desktop-jisqlks:Scripting marcoautili$
```

Try continue [n] yourself

Select

The select command is useful for doing iterations indefinitely in shell scripts

- built-in command that allows for the easy construction of a menu
- e.g., with the select command, you can present some data/options to users for interactive shell scripts
- e.g., depending on user inputs, the select command runs a certain option and gives back the prompt to the user with options once again

```
select var in <list>
do
        <commands>
done
```

- PS3 prompt is useful in shell scripts along with the select command to provide a custom prompt for the user to select a value

Select

```
marcoautili — bash — 60×25
bash-3.2$
bash-3.2$ ./select-script1.sh

This script shows that the select statement
allows for defining indefinite loops.
Choose a number followed by [ENTER]
or simply type [ENTER]

1) pippo
2) pluto
3) paperino
4) minnie
5) ziopaperone
#? 2
You have selected: pluto
#? 4
You have selected: minnie
#?
1) pippo
2) pluto
3) paperino
4) minnie
5) ziopaperone
#? ^C
bash-3.2$
```

select-script1.sh

```
1  #!/bin/bash
2  # This is the first script showing
3  # the select statement
4  # Marco A. dd/mm/yyyy
5
6  echo
7  echo "This script shows that the select statement "
8  echo "allows for defining indefinite loops."
9  echo "Choose a number followed by [ENTER]"
10 echo "or simply type [ENTER]"
11 echo
12
13 select var1 in pippo pluto paperino minnie ziopaperone
14 do
15     echo "You have selected: $var1"
16 done
17
```

Select

select-script2.sh

```
#!/bin/bash
# This is the first script showing
# the select statement
# Marco A. dd/mm/yyyy

echo
echo "This script shows the usage of PS3 "

PS3="Please enter a number from the above list: "

select var1 in pippo pluto paperino minnie ziopaperone
do
    echo "You selected: $REPLY) "
    echo "and the value of var1 is $var1"
done

```

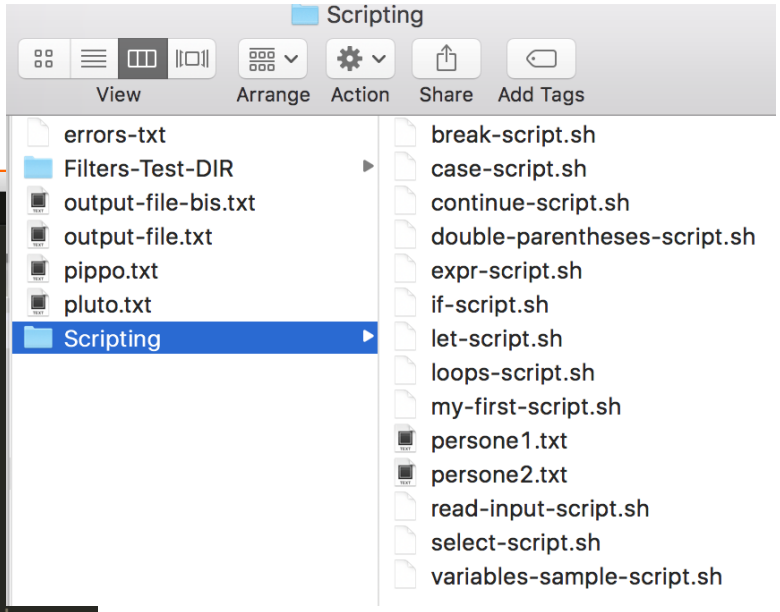
marcoautili — bash — 60×

```
bash-3.2$
bash-3.2$ ./select-script2.sh

This script shows the usage of PS3
1) pippo
2) pluto
3) paperino
4) minnie
5) ziopaperone
Please enter a number from the above list: 2
You selected: 2)
and the value of var1 is pluto
Please enter a number from the above list: 5
You selected: 5)
and the value of var1 is ziopaperone
Please enter a number from the above list: ^C
bash-3.2$
```

Select

select-script3.sh

```
#!/bin/bash
# This is another script showing the select statement
# together with a case statement
# Marco A. dd/mm/yyyy

echo
echo "This script can make a copy of a file in the current directory"
echo ---

# Meaningful prompt

PS3="--- Choose the number of the file to be copied: "
# Temporary file aptly created so to have an option to exit the script
TEMP_FILE_TO_QUIT="QUIT THIS PROGRAM - No more copies are needed."
touch "$TEMP_FILE_TO_QUIT"

# All the files in the current folder (... togther with $TEMP_FILE_T
select FILENAME in *;
do
  case $FILENAME in
       "$TEMP_FILE_TO_QUIT")
       echo
       echo "--- Closing the script..."
       break
       ;;
       *)
       echo
       echo ---
       echo "You selected the file $FILENAME by chosing $REPLY"
       cp "$FILENAME" "$FILENAME-copy-copy"
       ;;
  esac
done
# Remove the temporary file $TEMP_FILE_TO_QUIT
rm "$TEMP_FILE_TO_QUIT"

```




```
Scripting — -bash — 63×26
desktop-jisqlks:Scripting marcoautili$
desktop-jisqlks:Scripting marcoautili$ ./select-script.sh
This script can make a copy of a file in the current directory
---
 1) QUIT THIS PROGRAM - No more copies are needed.
 2) break-script.sh
 3) case-script.sh
 4) continue-script.sh
 5) double-parentheses-script.sh
 6) expr-script.sh
 7) if-script.sh
 8) let-script.sh
 9) loops-script.sh
10) my-first-script.sh
11) persone1.txt
12) persone1.txt-copy-copy
13) persone2.txt
14) read-input-script.sh
15) select-script.sh
16) variables-sample-script.sh
--- Choose the number of file to be copied: 11
---
You selected the file persone1.txt by chosing 11
--- Choose the number of file to be copied: 1
--- Closing the script...
desktop-jisqlks:Scripting marcoautili$
```

Bash functions

```
function_name () {
    <commands>
}
```

or

```
function function_name {
    <commands>
}
```

- Unlike functions in most programming languages, Bash functions "do not allow you to return an arbitrary value" to the caller and, e.g., assign it directly to a variable

- To "actually return" arbitrary values, you can:
  - set a global variable with the result
  - use command substitution
  - you can pass as input the name of a variable to be used as the result variable

- See the next slides…

Bash functions

- Although bash functions has a return statement, the only thing you can specify with it is the function's status, which is a numeric value assigned to the variable $? (like the value specified in an exit statement)

- The return statement terminates the function, and the specified value is stored into the $? variable
  - This mechanism permits script functions to have a "return value" similar to C functions, though limited in scope
    - zero for success
    - non-zero in the 1 - 255 range for failure

- If a function does not contain a return statement, its status is anyway stored in $? based on the status of the last statement executed in the function

```
#!/bin/bash

my_function () {
  echo "some result"
  return 55
}

my_function
echo $?
```

```
Output
some result
55
```

Bash functions

- The simplest way (not the best) to return a value from a bash function is to just set a global variable to the result
- Since all variables in bash are global (within the script scope) by default, this is easy

```
function myfunc()
{
    myresult='some value'
}

myfunc
echo $myresult
```

- The code above uses the "global" variable myresult as the function result
- However, when using global variables, particularly in large programs, it can be difficult to find bugs

Bash functions

- A better approach is to use local variables in your functions
- The problem then becomes how do you get the result to the caller?
    - One mechanism is to use command substitution

```
function myfunc()
{
    local  myresult='some value'
    echo "$myresult"
}

result=$(myfunc)    # or result=`myfunc`
echo $result
```

- Here the result is output to the stdout and the caller uses command substitution to capture the value in a variable

Bash functions

- The other way to return a value is to write your function so that it accepts a variable name as part of its "command line" and then set that variable to the result of the function

```
function myfunc()
{
    local  __resultvar=$1
    local  myresult='some value'
    eval $__resultvar="'$myresult'"
}

myfunc result
echo $result
```

No interpretation
'$myresult' is a string

First interpretation (substitution/expansion)
result='some value' (it is also a string)

Second interpretation (assignement)
some value is assigned to result

- Since we have the name of the variable to set stored in a variable, we can't set the variable directly, we have to use eval to actually do the setting
- eval [arguments] is a builtin command. The arguments are concatenated together into a single command, which is then read and executed, and its exit status is returned as the exit status of eval. If there are no arguments or only empty arguments, the return status is zero
- It is useful when you have a command stored in a variable and you want to execute it
- The eval statement basically tells bash to interpret the line twice

Next slide

Eval

- To execute a command stored in the string

  cmd='ls –la'

  eval $cmd

  As output, you will have the list of files in the current folder

- To print the value of a variable which is again a variable with value assigned to it

```
bash-3.2$
bash-3.2$ a=10
bash-3.2$ b=a
bash-3.2$ echo $b
a
bash-3.2$ c='$'$b
bash-3.2$ echo $c
$a
bash-3.2$ eval c='$'$b
bash-3.2$ echo $c
10
bash-3.2$ echo $a
10
bash-3.2$
```

NOTE that the dollar sign must be escaped with '$'

Eval

eval-script.sh

eval-script.sh

eval-script.sh 〉 No Selection

```
1  #!/bin/bash
2  # This is the first script about "eval"
3  # Marco A. dd/mm/yyyy
4
5  myfile="persone1.txt"
6
7  echo '************************'
8  # By using single quotes, you are preventing the shell from interpreting $myfile
9  mycmd='cat $myfile'
10 echo $mycmd; echo "$mycmd" # Both print out: cat $myfile
11 echo '************************'
12
13 # Let's us evaluate $mycomd
14 eval $mycmd   # Prints out the content of persone1.txt
15 echo '************************'
16
17 # By using double quotes, you are allowing the shell to interpret $myfile
18 mycmd="cat $myfile"
19 echo $mycmd; echo "$mycmd" # Prints out: cat persone1.txt
20 echo '************************'
21
22 # Let's us evaluate $mycomd again
23 eval $mycmd   #Still it prints out the content of persone1.txt
24 echo '************************'
25
26 myfile="persone2.txt"
27 eval $mycmd   #Still prints out persone1.txt and not persone2.txt
28 echo '************************'
29
```

Bash functions

- … back to the previous example: make sure the name of the local variable won't be (unlikely to be…) used by the caller
- If the caller chooses the same name for its result variable as you use for storing the name, the result variable will not get set
- For example, the following does not work

```
function myfunc()
{
    local  result=$1
    local  myresult='some value'
    eval $result="'$myresult'"
}

myfunc result
echo $result
```

- The reason it doesn't work is because when eval does the second interpretation and evaluates result='some value', result is now a local variable in the function, and so it gets set rather than setting the caller's result variable

Bash functions

- For more flexibility, you may want to write your functions so that they combine both result variables and command substitution
- If no variable name is passed to the function, the value is output to the standard output
- Remember that:
  - both [("test" command) and [[("new test" command) are used to evaluate expressions
  - [and test are available in POSIX shells

```
function myfunc()
{
    local  __resultvar=$1
    local  myresult='some value'
    if [[ "$__resultvar" ]]; then
        eval $__resultvar="'$myresult'"
    else
        echo "$myresult"
    fi
}

myfunc result
echo $result
result2=$(myfunc)
echo $result2
```

See the script: "functions-return-script.sh"

Scope of variables

functions-var-scoping-script.sh

```
#!/bin/bash
# This is the first script showing the scope of variables within functions
# Marco A. dd/mm/yyyy

# No need to specify formal paramters
test_var_scope () {
    local var1="local $1"

    var2="changed to global $1"

    echo Inside the function: var1 is $var1 : var2 is $var2

    return 0
    #return pippo # Try this
}

var1='global 1'
var2='global 2'

echo
echo Before function call: var1 is $var1 : var2 is $var2
echo

test_var_scope 2000

return_value=$?

echo
echo After function call: var1 is $var1 : var2 is $var2
echo

echo
echo The return value is $return_value
```

Typing variables: declare or typeset

- The declare or typeset builtins, which are exact synonyms, permit modifying the properties of variables
- This is a very weak form of the typing available in certain programming languages
- The declare builtin command is specific to version 2 or later of Bash
- The typeset builtin command is supplied in Bash for compatibility with the Korn shell (it is a synonym for the declare command)
- Both declare and typeset works fine in both bash and ksh (see next slide)

Typing variables: declare or typeset

variables-declare-typeset-script.sh

```
#!/bin/bash
# Script about declaring (or typesetting) variables
# Marco A. dd/mm/yyyy

declare -r var1=1 # or readonly -r var1=1
echo "var1 = $var1"   # var1 = 1
(( var1++ ))          # Error: readonly variable


declare -i number
# Subsequent occurrences of "number" will treated as an integer.

number=3
echo "Number = $number"     # Number = 3

number=three
# Tries to evaluate the string "three" as an integer.
# Note that there will not be error
echo "Number = $number"     # Number = 0

# Certain arithmetic operations are permitted for declared integer
# variables without the need for expr or let.
n=6/3
echo "n = $n"       # n = 6/3

declare -i n
n=6/3
echo "n = $n"       # n = 2

# The variable indices will be treated as an array.
# Newer versions of Bash support one-dimensional arrays.
declare -a array

array[11]=23 # Array members need not be consecutive or contiguous.
array[13]=37
array[51]=UFOs #

echo "array[11] = " ${array[11]}    #  {curly brackets} needed.
echo "array[13] = " ${array[13]}
```

Bash variables are untyped

Unlike many other programming languages, Bash does not “segregate” its variables by “type”

Essentially, Bash variables are character strings but, depending on context, Bash permits arithmetic operations and comparisons on variables

- The determining factor is whether the value of a variable contains only digits

To lighten the burden of keeping track of variable types in a script, Bash does permit declaring variables

Checkout the entire script from the svn

Typing variables: declare or typeset

Typing a variable means to classify it and restrict its properties

- For example, a variable declared or typed as an integer is no longer available for string operations

bash

```
bash-3.2$
bash-3.2$ declare -i intvar
bash-3.2$ intvar=8
bash-3.2$ echo $intvar
8
bash-3.2$ intvar=pippo
bash-3.2$ echo $intvar
0
bash-3.2$ typeset -i intvarbis
bash-3.2$ intvarbis=16
bash-3.2$ echo $intvarbis
16
bash-3.2$
```

In both shells they work fine

zsh

```
marcoautili — zsh — 70×24
bash-3.2$
bash-3.2$ zsh
marcoautili@iMac ~ %
marcoautili@iMac ~ %
marcoautili@iMac ~ % typeset -i intvar
marcoautili@iMac ~ % intvar=32
marcoautili@iMac ~ % echo $intvar
32
marcoautili@iMac ~ % intvar=pluto
marcoautili@iMac ~ % echo $intvar
0
marcoautili@iMac ~ %
marcoautili@iMac ~ % declare -i intvarbis
marcoautili@iMac ~ % intvarbis=64
marcoautili@iMac ~ % echo $intvarbis
64
marcoautili@iMac ~ %
```

Another use for declare

- The declare command can be helpful in identifying variables, environmental or otherwise.

  This can be especially useful with arrays

```
marcoautili — bash — 101×24
bash-3.2$
bash-3.2$ declare | egrep HOME
HOME=/Users/marcoautili
JAVA_HOME='/Library/Internet Plug-Ins/JavaAppletPlugin.plugin/Contents/Home'
bash-3.2$
bash-3.2$
bash-3.2$ pippo=64
bash-3.2$ declare | egrep pippo
pippo=64
bash-3.2$
bash-3.2$
bash-3.2$ cartoon_characters=([0]="pippo" [1]="pluto" [2]="paperino" [3]="minnie")
bash-3.2$ echo ${cartoon_characters[@]}
pippo pluto paperino minnie
bash-3.2$ typeset | egrep cartoon_characters
cartoon_characters=([0]="pippo" [1]="pluto" [2]="paperino" [3]="minnie")
bash-3.2$
```

Try also with the following and switch between bash and zsh

declare –p or declare –F or declare –f

While switching, try also with typeset

No manual entry for declare and typeset 😕

bash

```
bash-3.2$
bash-3.2$ man declare
No manual entry for declare
bash-3.2$ man typeset
No manual entry for typeset
bash-3.2$
```

zsh

```
bash-3.2$ zsh
marcoautili@iMac ~ % man declare
No manual entry for declare
marcoautili@iMac ~ % man typeset
No manual entry for typeset
marcoautili@iMac ~ %
```

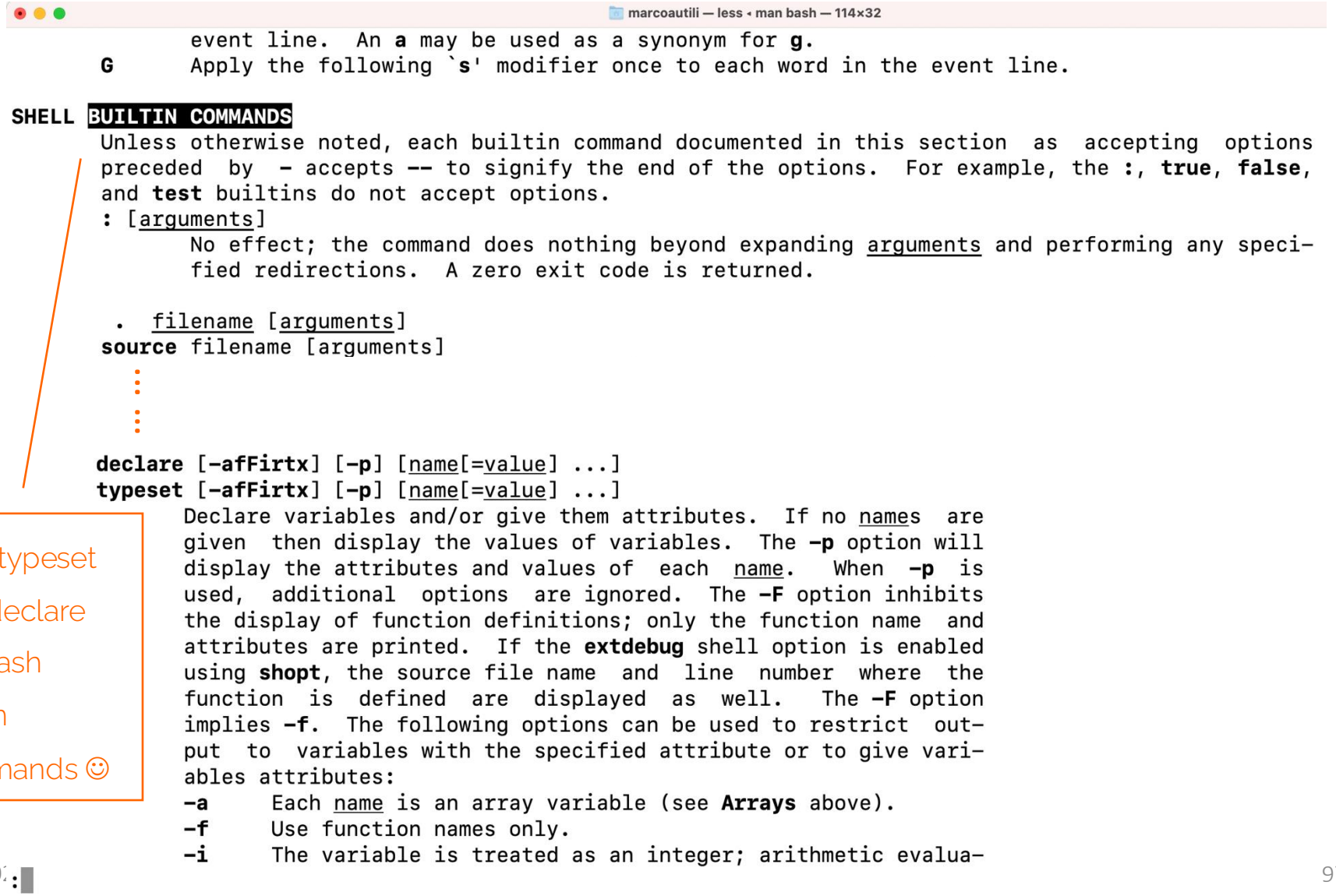
Do you know why?

Or do you remember why?

See next slide …

man page for builtin commands

man bash (then, search for, e.g., /BUILTIN COMMANDS)



Both typeset and declare are Bash builtin commands ☺

Additional material

- Learn Shell Programming

  https://www.learnshell.org

- The Shell Scripting Tutorial

  https://www.shellscript.sh

- The Bash Academy

  http://www.bash.academy

- Advanced Bash-Scripting Guide

  http://tldp.org/LDP/abs/html/