



# SYSTEM MANAGEMENT AND BASIC SCRIPTING

SOFTWARE ENGINEERING

# CONTENTS

- System Information
- Managing Services
- Package management
- Basic shell scripting concepts

# SYSTEM INFORMATION COMMANDS

- To know **only the system name**, you can use the **uname command** without any switch that will print system information or the **uname -s command** will print the kernel name of your system.
- To view your Linux network hostname, use the '**-n**' switch with the **uname command** as shown.
- To get information about the Linux kernel version: **uname -v**
- To get the information about your Linux kernel release: **uname -r**
- To print your Linux hardware architecture name: **uname -m**
- All this information can be printed at once by running the '**uname -a**' command

# DF/DU

- The "disk free" (df) command tells you the total disk size, space used, space available, usage percentage, and what partition the disk is mounted on.
- The "disk usage" (du) is used you need to see the size of a given directory or subdirectory. It runs at the object level and only reports on the specified stats at the time of execution.

```
ahmedmady@HP:~/Documents$  
ahmedmady@HP:~/Documents$ df  
Filesystem      1K-blocks    Used Available Use% Mounted on  
tmpfs            200596     1448   199148   1% /run  
/dev/sda2        25623780  14537968  9758864  60% /  
tmpfs            1002960       0  1002960   0% /dev/shm  
tmpfs             5120       8    5112   1% /run/lock  
tmpfs            200592      96  200496   1% /run/user/1000  
/dev/sr0          5047188  5047188       0 100% /media/ahmedmady/Ubuntu 23.10.1  
  amd64  
ahmedmady@HP:~/Documents$ du Files  
4      Files/folder/hi  
8      Files/folder  
100    Files  
ahmedmady@HP:~/Documents$
```

# MANAGING SERVICES AND DAEMONS

- An operating system requires programs that run in the background called services. In a Linux system, these services are called daemons. They are managed using an init system like systemd.
- In Unix-based computer operating systems, init (short for initialization) is the first process started during booting of the operating system. Init is a daemon process that continues running until the system is shut down.

# PROCESS VS SERVICE

- A process is an instance of a running program. When you execute a program, it becomes a process.
- Processes are the basic units of execution in a Linux system.
- Each process has a unique process ID (PID) assigned to it.
- Processes have their own memory space, file descriptors, and other resources.

# PROCESS VS SERVICE

- A service is a background process or daemon that runs on a system to provide specific functionality or perform specific tasks.
- Services often start when the system boots up and continue running in the background, waiting for specific events or requests.
- Services are usually managed by an init system like systemd, which can start, stop, restart, and manage their lifecycle, there are others, but system is the most used.

# MANAGING SERVICES AND DAEMONS

- Most modern Linux systems use `systemd` – an init system and service manager for controlling daemons. It is a drop-in replacement for older distributions' init processes: `pstree` → started by `systemd`
- `Systemd` has the `systemctl` command, which lets users manage their system and service configurations. For example, use it to list all unit files in your Linux server: `daemons = units = services`
- `sudo systemctl list-unit-files --type service --all`

```
ahmedmady@HP:~/Documents$ sudo systemctl list-unit-files --type service --all
[sudo] password for ahmedmady:
UNIT FILE                                     STATE   PRESET
accounts-daemon.service                       enabled enabled
alsa-restore.service                          static  -
alsa-state.service                           static  -
alsa-utils.service                          masked  enabled
anacron.service                            enabled enabled
apparmor.service                           enabled enabled
apport-autoreport.service                   static  -
apport-forward@.service                     static  -
apport.service                            enabled enabled
apt-daily-upgrade.service                  static  -
apt-daily.service                          static  -
apt-news.service                           static  -
autovt@.service                           alias   -
avahi-daemon.service                      enabled enabled
bluetooth.service                         enabled enabled
bolt.service                             static  -
brltty-udev.service                        static  -
brltty.service                            disabled enabled
cloud-config.service                      enabled enabled
cloud-final.service                       enabled enabled
cloud-init-hotplugd.service                static  -
cloud-init-local.service                  enabled enabled
Lines 1-23...skipping...
UNIT FILE                                     STATE   PRESET
accounts-daemon.service                       enabled enabled
```

# SUDO SYSTEMCTL LIST-UNIT-FILES --TYPE SERVICE -- ALL

- Enabled – active services running in the background.
- Disabled – disabled services that users can enable directly using the start command.
- Masked – stopped services that can only be started by removing the masked property.
- Static – services that only run when another program or unit requires them.
- Failed – inactive services that can't load or operate properly.

# MANAGING SERVICES AND DAEMONS

- To stop a service: `sudo systemctl stop [SERVICE]`
- To get the status of a service: `sudo systemctl status [SERVICE]`
- To start a service: `sudo systemctl start [SERVICE]`
- To restart a service: `sudo systemctl restart [SERVICE]`

# PACKAGE MANAGEMENT

- When it comes to package management on Linux systems, two popular tools are YUM and APT. YUM, short for Yellowdog Updater Modified, is commonly used in Red Hat-based distributions like CentOS and RHEL. On the other hand, APT, which stands for Advanced Packaging Tool, is widely used in Debian, Ubuntu, and their derivatives. Understanding the differences between these package managers can help you effectively manage software installations and updates on your Linux system.

# YUM VS. APT: MANAGING SOFTWARE PACKAGES ON LINUX

- YUM and APT are package managers that simplify the installation, upgrade, and configuration of software packages on Linux systems. While they serve the same purpose, there are some notable differences between them.

# APT

- APT uses .deb files as the package format and is primarily used in Debian, Ubuntu, and related distributions.
- APT provides several commonly used commands, such as update, upgrade, install, remove, purge, list, and search.
- APT organizes options into functional groups and stores them in the /etc/apt/apt.conf file, which is organized in a tree structure.

# APT

- When you install an application using apt in Ubuntu, the package manager (apt) does not contain all the data for the application itself. Instead, apt relies on repositories, which are online databases of software packages maintained by Ubuntu and its community.
- When you instruct apt to install a package, it searches the repositories configured on your system to find the package and its dependencies. If the package and its dependencies are found, apt downloads them from the repository over the internet and installs them on your system.
- The repositories contain metadata about the packages, such as their names, versions, descriptions, dependencies, and download locations. apt uses this metadata to locate and install the requested packages.
- So, while apt itself doesn't contain all the data for each application, it acts as a tool to efficiently manage the installation and removal of software packages from online repositories.

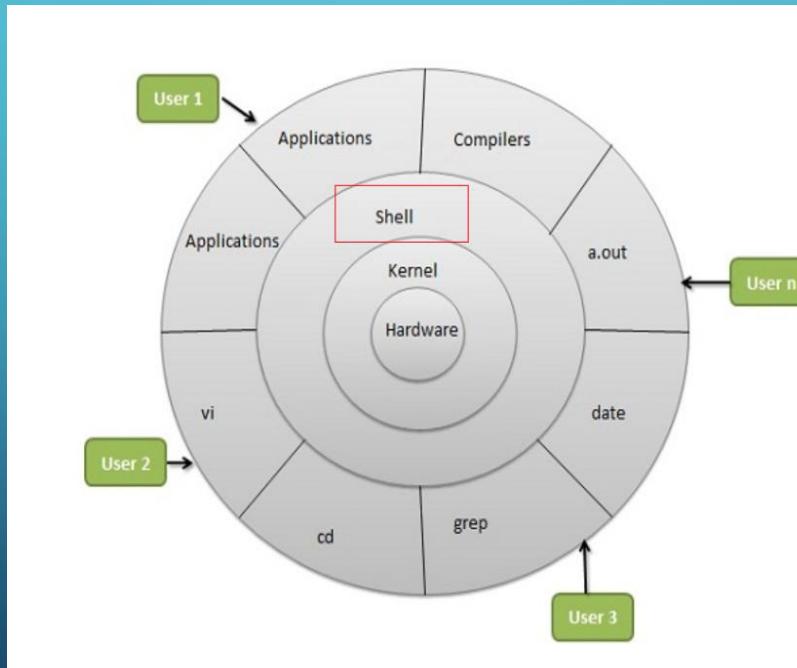
```
ahmedmady@HP:~$ sudo apt-get install -y mypaint
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following NEW packages will be installed:
  mypaint
0 upgraded, 1 newly installed, 0 to remove and 22 not upgraded.
Need to get 700 kB of archives.
After this operation, 3,558 kB of additional disk space will be used.
Get:1 http://eg.archive.ubuntu.com/ubuntu mantic/universe amd64 mypaint amd64 2.0.1-9 [700 kB]
Fetched 700 kB in 2s (433 kB/s)
Selecting previously unselected package mypaint.
(Reading database ... 223676 files and directories currently installed.)
Preparing to unpack .../mypaint_2.0.1-9_amd64.deb ...
Unpacking mypaint (2.0.1-9) ...
Setting up mypaint (2.0.1-9) ...
Processing triggers for gnome-menus (3.36.0-1.lubuntu1) ...
Processing triggers for man-db (2.11.2-3) ...
Processing triggers for mailcap (3.70+nmulubuntu1) ...
Processing triggers for desktop-file-utils (0.26-1lubuntu5) ...
ahmedmady@HP:~$ sudo apt-get remove -y mypaint
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following packages were automatically installed and are no longer required:
  libblas3 libgfortran5 liblapack3 libmypaint-1.5-1 libmypaint-common mypaint-brushes mypaint-data
    mypaint-data-extras python3-distutils python3-numpy
Use 'sudo apt autoremove' to remove them
```

# YUM

- YUM uses .rpm files and is commonly used in Red Hat-based distributions like CentOS, RHEL, Fedora, and OpenSUSE.
- YUM offers commands like install, remove, search, info, and update.
- YUM allows options to be set with global and repository-specific effects, and the configuration is managed in the /etc/yum.conf file

# BASIC SHELL SCRIPTING CONCEPTS

- Shell is an interpreter for the applications/commands of a user to make the kernel manage the hardware.



# TYPES OF SHELL

Types of shell with varied features

- o **sh** o the original Bourne shell.
- o **ksh** o one of the three: Public domain ksh (pdksh), AT&T ksh or mksh
- o **bash** o the GNU Bourne-again shell. It is mostly Bourne-compatible, mostly POSIX-compatible, and has other useful extensions. It is the default on most Linux systems.
- o **csh** o BSD introduced the C shell, which sometimes resembles slightly the C programming language.
- o **tcsh** o csh with more features. csh and tcsh shells are NOT Bourne- compatible.

# SHELL COMPARISON

Software	sh	csh	ksh	bash	tcsh
Programming language	y	y	y	<b>y</b>	y
Shell variables	y	y	y	<b>y</b>	y
Command alias	n	y	y	<b>y</b>	y
Command history	n	y	y	<b>y</b>	y
Filename completion	n	y*	y*	<b>y</b>	y
Command line editing	n	n	y*	<b>y</b>	y
Job control	n	y	y	<b>y</b>	y

\*: not by default

- Bash supports all because it was created by the linux foundation(non-profit organization) → time and resources are abundant

# WHAT CAN YOU DO WITH A SHELL

- File Management, Directory Management
- Process Management, compile and run applications
- Network Management
- Shell Scripting

# SHELL

- List available shells on the system: `cat /etc/shells`
- To check the current shell you are using: `echo $0`

```
ahmedmady@HP:~$ cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/usr/bin/sh
/bin/bash
/usr/bin/bash
/bin/rbash
/usr/bin/rbash
/bin/dash
/usr/bin/dash
ahmedmady@HP:~$ echo $0
bash
ahmedmady@HP:~$
```

# SHELL SCRIPTING

- Script: a program written for a software environment to automate execution of tasks
- A series of shell commands put together in a file
- When the script is executed, those commands will be executed one line at a time automatically
- Shell script is interpreted, not compiled.

# WHEN NOT TO USE SHELL SCRIPTING

- Performance/Security-Critical Applications
- Complex Data Structures and Algorithms
- Cross-Platform Development
- Large Software Projects

# HELLO

```
#!/bin/bash
# A script example
echo "Hello World!" # print something
```

1. `#!`: "Shebang" line to instruct which interpreter to use. In the current example, bash. For tcsh, it would be: `#!/bin/tcsh`
1. All comments begin with `#`.
2. Print "Hello World!" to the screen.

```
$ ./hello_world.sh # using default Hello World!
$ bash hello_world.sh # using bash script to run the /bin/bash
Hello World!
```

```
GNU nano 7.2
#!/bin/bash
echo "Hello G14"
```

```
Feb 2 11:27 PM ahmedmady@HP:~/Documents/Files
ahmedmady@HP:~/Documents/Files$ nano hello.sh
ahmedmady@HP:~/Documents/Files$ cat hello.sh
#!/bin/bash
echo "Hello G13 Software Engineering"
ahmedmady@HP:~/Documents/Files$ ./hello.sh
bash: ./hello.sh: Permission denied
ahmedmady@HP:~/Documents/Files$ ls -l
total 32
-rw-rw-r-- 1 ahmedmady ahmedmady 474 Feb  2 13:23 file1.cpp
-rwx--xr-- 1 ahmedmady ahmedmady    0 Jan 28 13:03 file2.py
-rw-rw-r-- 1 ahmedmady ahmedmady    0 Jan 28 13:03 file3.cpp
-rw-rw-r-- 1 ahmedmady ahmedmady 138 Jan 31 18:45 file.tar.gz
drwxrwxr-x 3 ahmedmady ahmedmady 4096 Jan 28 13:08 folder
-rw-rw-r-- 1 ahmedmady ahmedmady   50 Feb  2 23:25 hello.sh
-rw-rw-r-- 1 ahmedmady ahmedmady    6 Feb  2 10:45 list
-rw-rw-r-- 1 ahmedmady ahmedmady  656 Feb  2 10:48 records.txt
-rw-rw-r-- 1 ahmedmady ahmedmady   71 Feb  1 14:48 text_file.txt
-rw-rw-r-- 1 ahmedmady ahmedmady  244 Feb  2 20:04 unix
ahmedmady@HP:~/Documents/Files$ chmod 700 hello.sh
ahmedmady@HP:~/Documents/Files$ ls -l
total 32
-rw-rw-r-- 1 ahmedmady ahmedmady 474 Feb  2 13:23 file1.cpp
-rwx--xr-- 1 ahmedmady ahmedmady    0 Jan 28 13:03 file2.py
-rw-rw-r-- 1 ahmedmady ahmedmady    0 Jan 28 13:03 file3.cpp
-rw-rw-r-- 1 ahmedmady ahmedmady 138 Jan 31 18:45 file.tar.gz
drwxrwxr-x 3 ahmedmady ahmedmady 4096 Jan 28 13:08 folder
-rwx----- 1 ahmedmady ahmedmady   50 Feb  2 23:25 hello.sh
-rw-rw-r-- 1 ahmedmady ahmedmady    6 Feb  2 10:45 list
-rw-rw-r-- 1 ahmedmady ahmedmady  656 Feb  2 10:48 records.txt
-rw-rw-r-- 1 ahmedmady ahmedmady   71 Feb  1 14:48 text_file.txt
-rw-rw-r-- 1 ahmedmady ahmedmady  244 Feb  2 20:04 unix
ahmedmady@HP:~/Documents/Files$ ./hello.sh
Hello G13 Software Engineering
ahmedmady@HP:~/Documents/Files$
```

# INTERACTIVE VS NONINTERACTIVE SHELL

- An **interactive shell** refers to a command-line interface that allows users to interact with the computer's operating system by typing commands in real-time and receiving immediate feedback. In an interactive shell, users can enter commands, execute programs, and perform various tasks by typing text-based commands.
- A **non-interactive shell** is a shell session that runs without direct interaction with a user through a command-line interface. In a non-interactive shell, commands are often executed from scripts or other automated processes, and there is typically no user input or interaction during the execution. A shell running a script is always a non-interactive shell.

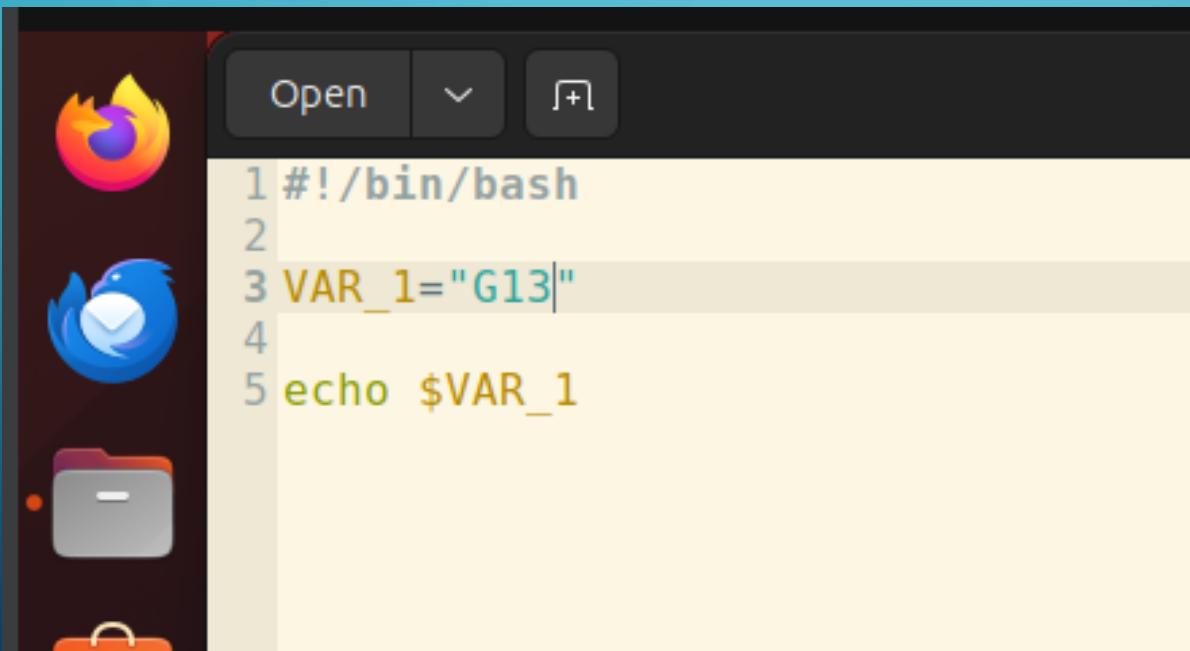
# SUBSHELL

- A subshell is a child shell that is spawned by the main shell (also known as the parent shell). It is a separate process with its own set of variables and command history, and it allows you to execute commands and perform operations within a separate environment.



# VARIABLES IN SHELL SCRIPTING

- A symbolic name for a chunk of memory to which we can assign values, read and manipulate its contents.



A screenshot of a terminal window on a Mac OS X desktop. The window title bar says "Open". The desktop icons visible include a Firefox icon, a Mail icon, a Finder icon, and a Home icon. The terminal window contains the following shell script code:

```
1 #!/bin/bash
2
3 VAR_1="G13"
4
5 echo $VAR_1
```

# VARIABLES

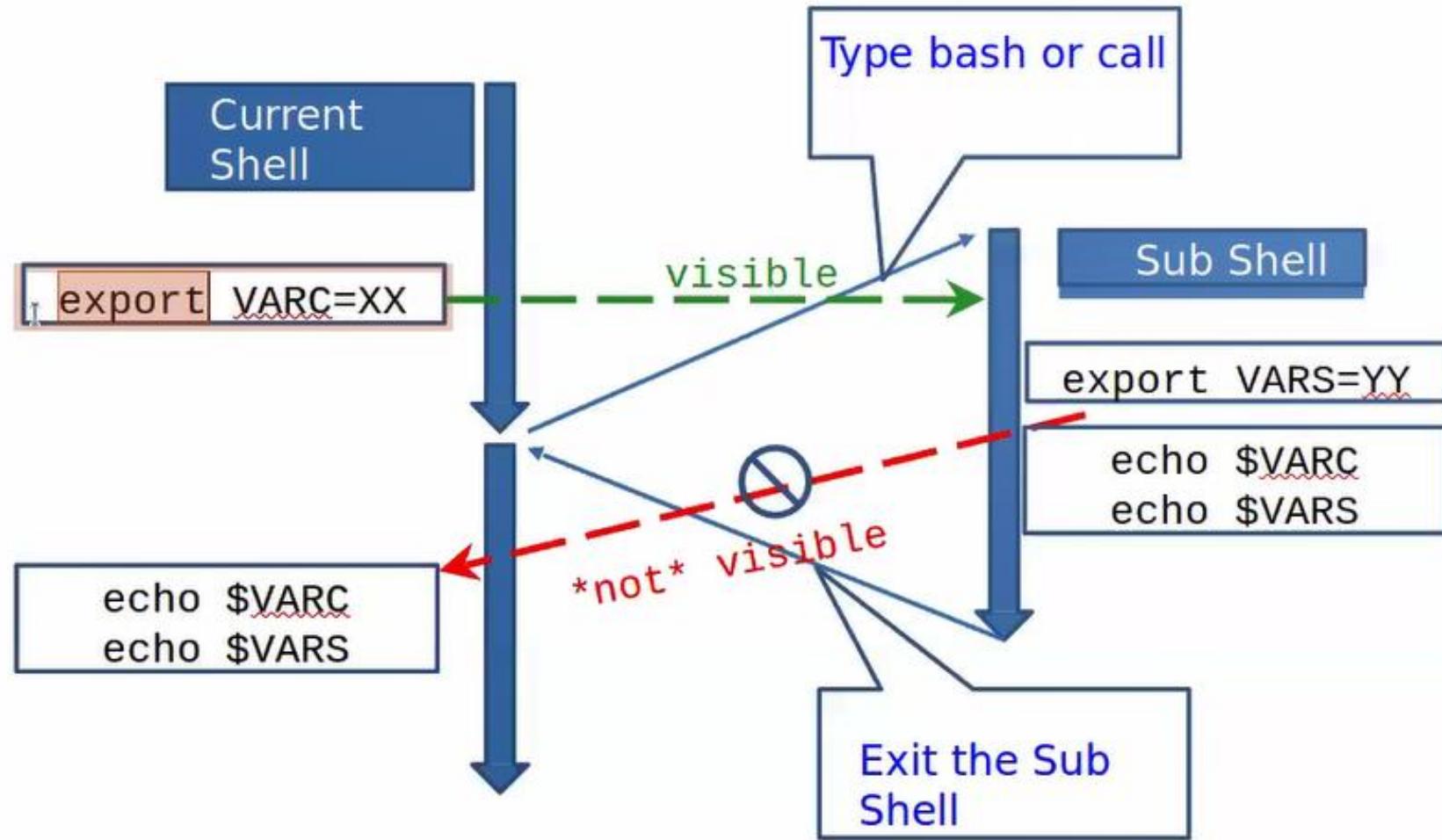
- Must start with a letter or underscore
- Number can be used anywhere else
- Do not use special characters such as @,#,%,\$
- Case sensitive
- Allowed: VARIABLE, VAR1234able, var\_name, \_VAR
- Not allowed: 1var, %name, \$myvar, var@NAME, myvar-1
- To reference a variable, prepend \$ to the name of the variable
- Example: \$PATH, \$LD\_LIBRARY\_PATH, \$myvar etc.

# LOCAL AND GLOBAL VARIABLES

- Variables created and used inside a shell are only local to the shell and can't be seen outside this shell.
- If a variable is created in the terminal and used in a process for example, this variable is local to this terminal and can't be seen elsewhere.
- Global variables are variables that are accessible and can be modified throughout the entire script, regardless of their initial declaration.
  - Like: PATH, LD\_LIBRARY\_PATH, DISPLAY
  - cmake -version: export name=value(directory)

# Global and Local Variables

## - current shell and subshell



GNU nano 7.2 global.sh

```
#!/bin/bash

export USER_NAME="G13"
```

GNU nano 7.2 call\_global.sh

```
#!/bin/bash

echo $USER_NAME
```

ahmedmady@HP:~/Documents/Files\$ ./call\_global.sh

ahmedmady@HP:~/Documents/Files\$ source global.sh

ahmedmady@HP:~/Documents/Files\$ ./call\_global.sh

G13

ahmedmady@HP:~/Documents/Files\$

ahmedmady@HP:~/Documents/Files\$ nano global.sh

ahmedmady@HP:~/Documents/Files\$ nano call\_global.sh

ahmedmady@HP:~/Documents/Files\$ . global.sh

ahmedmady@HP:~/Documents/Files\$ source global.sh

ahmedmady@HP:~/Documents/Files\$ chmod 700 call\_global.sh

ahmedmady@HP:~/Documents/Files\$ ./call\_global.sh

G13

ahmedmady@HP:~/Documents/Files\$

# List of Some Environment Variables

PATH	A list of directory paths which will be searched when a command is issued
LD_LIBRARY_PATH	colon-separated set of directories where libraries should be searched for first
HOME	indicate where a user's home directory is located in the file system.
PWD	contains path to current working directory.
OLDPWD	contains path to previous working directory.
TERM	specifies the type of computer terminal or terminal emulator being used
SHELL	contains name of the running, interactive shell.
PS1	default command prompt
PS2	Secondary command prompt
HOSTNAME	The systems host name
USER	Current logged in user's name
DISPLAY	Network name of the X11 display to connect to, if available.

# QUOTATIONS

- Single quotation: Enclosing characters in single quotes ('') preserves the literal value of each character within the quotes. A single quote may not occur between single quotes, even when preceded by a backslash.
- Double quotation: Enclosing characters in double quotes ("") preserves the literal value of all characters within the quotes, with the exception of '\$', ``', ``\''

# QUOTATION

Str1='echo \$USER'

Echo "\$str1" → echo \$USER

Str2="echo \$USER"

Echo "\$str2" → echo YOUR\_USERNAME

Str3='`echo \$USER`'

Echo \$str3 → YOUR\_USERNAME

Str4=\$(echo \$USER)

Echo "\$str4" → YOUR\_USERNAME

```
ahmedmady@HP:~$ echo $USER  
ahmedmady  
ahmedmady@HP:~$ str1='echo $USER'  
ahmedmady@HP:~$ echo "$str1"  
echo $USER  
ahmedmady@HP:~$ str2="echo $USER"  
ahmedmady@HP:~$ echo "$str2"  
echo ahmedmady  
ahmedmady@HP:~$ str3=`echo $USER`  
ahmedmady@HP:~$ echo $str3  
ahmedmady  
ahmedmady@HP:~$ str4=$(echo $USER)  
ahmedmady@HP:~$ echo "$str4"  
ahmedmady  
ahmedmady@HP:~$ █
```

# SPECIAL CHARACTERS

#	Start a comment line.
\$	Indicate the name of a variable.
\	Escape character to display next character literally
{}	Enclose name of variable
;	Command separator. Permits putting two or more commands on the same line.
;;	Terminator in a case option
.	“dot” command, equivalent to <code>source</code> (for bash only)
	Pipe: use the output of a command as the input of another one
>	Redirections ( <code>0&lt;:</code> standard input; <code>1&gt;:</code> standard out; <code>2&gt;:</code> standard error)
<	

\$?	Exit status for the last command, 0 is success, failure otherwise
\$\$	Process ID variable.
[]	Test expression, eg. if condition
[[ ]]	Extended test expression, more flexible than [ ]
\$[], \$ (( ))	Integer expansion
, &&, !	Logical OR, AND and NOT

# INTEGER ARITHMETIC OPERATIONS

- `$((expression))`
- `$(( n1+n2 ))`
- `$(( n1/n2 ))`
- `$(( n1-n2 ))`
- Addition, Subtraction, Multiplication, Division, Exponentiation, Modulus

GNU nano 7.2

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

```
x="2"
```

```
y="3"
```

```
add=$((x+y))
```

```
sub=$((y-x))
```

```
echo add: "$add"
```

```
echo sub: "$sub"
```

```
ahmedmady@HP:~/Documents/Session4$ nano variables.sh
ahmedmady@HP:~/Documents/Session4$ chmod 700 variables.sh
ahmedmady@HP:~/Documents/Session4$ ./variables.sh
add: 5
sub: 1
ahmedmady@HP:~/Documents/Session4$
```

# FLOATING-POINT ARITHMETIC OPERATIONS

GNU basic calculator (bc) external calculator

- Add two numbers

```
echo "3.8 + 4.2" | bc
```

- Divide two numbers and print result with a precision of 5 digits:

```
echo "scale=5; 2/5" | bc
```

- Convert between decimal and binary numbers

```
echo "ibase=10; obase=2; 10" | bc
```

- Call bc directly:

```
bc <<< "scale=5; sqrt(2)"
```

Operation	bash	Operation	bash
File exists	if [ -e test ]	Equal to	if [ 1 -eq 2 ]
File is a regular file	if [ -f test]	Not equal to	if [ \$a -ne \$b ]
File is a directory	if [ -d /home ]	Greater than	if [ \$a -gt \$b ]
File is not zero size	if [ -s test ]	Greater than or equal to	if [ 1 -ge \$b ]
File has read permission	if [ -r test ]	Less than	if [ \$a -lt 2 ]
File has write permission	if [ -w test ]	Less than or equal to	if [ \$a -le \$b ]
File has execute permission	if [ -x test ]	Operation	Example
<b>Operation</b>		! (NOT)	if [ ! -e test ]
Equal to		&& (AND)	if [ -f test] && [ -s test ] [[ -f test && -s test ]] if ( -e test && ! -z test )
Not equal to			if
Zero length or null		(OR)	if [ -f test1 ]    [ -f test2 ] if [[ -f test1    -f test2 ]]
Non zero length			

# CONDITIONAL STATEMENTS

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

```
if [ condition ]; then  
# code to be executed if the condition is true  
fi
```

```
#!/bin/bash  
  
x=2  
  
if [ "$x" -eq 2 ]; then  
    echo "x is 2"  
fi
```

# LOOPS

- A loop is a block of code that iterates a list of commands as long as the loop control condition stays true
- Loop constructs
- for, while and until

```
#!/bin/bash

for((i=1;i<=10;i=i+1)); do
    echo "$i"
done
```

```
ahmedmady@HP:~/Documents/Session4$ nano loop.sh
ahmedmady@HP:~/Documents/Session4$ chmod 700 loop.sh
ahmedmady@HP:~/Documents/Session4$ ./loop.sh
1
2
3
4
5
6
7
8
9
10
ahmedmady@HP:~/Documents/Session4$ █
```

# While Loop

- The while construct test for a condition at the top of a loop and keeps going as long as that condition is true.
- In contrast to a for loop, a while is used when loop repetitions is not known beforehand.

```
#!/bin/bash

echo "Enter counter: "
read counter
while [ $counter -ge 0 ]
do
    counter=$((counter - 1))
    echo "$counter"
done
```

```
ahmedmady@HP:~/Documents/Session4$ nano while.sh
ahmedmady@HP:~/Documents/Session4$ chmod 700 while.sh
ahmedmady@HP:~/Documents/Session4$ ./while.sh
Enter counter:
3
2
1
0
-1
ahmedmady@HP:~/Documents/Session4$ █
```

# Until Loop

- The until construct test a condition at the top of a loop, and stops looping when the condition is met (opposite of while loop)

```
#!/bin/bash

echo "Enter counter: "
read counter
until [ $counter -eq 0 ]
do
    counter=$((counter - 1))
    echo "$counter"
done
```

```
ahmedmady@HP:~/Documents/Session4$ nano until.sh
ahmedmady@HP:~/Documents/Session4$ chmod 700 until.sh
ahmedmady@HP:~/Documents/Session4$ ./until.sh
Enter counter:
3
2
1
0
```

# Functions

- A function is a code block that implements a set of operations. Code reuse by passing parameters,

- Syntax:

```
function_name () {  
    command...  
}
```

- By default all variables are global.
- Modifying a variable in a function changes it in the whole script.
- Create local variables using the local command, which is invisible outside the function

```
local var=value
```

```
#!/bin/bash

print_function()
{
echo "Hello G15"
}

print_function
```

```
ahmedmady@HP:~/Documents/Session4$ nano function1.sh
ahmedmady@HP:~/Documents/Session4$ chmod 700 function1.sh
ahmedmady@HP:~/Documents/Session4$ ./function1.sh
Hello G15
ahmedmady@HP:~/Documents/Session4$ █
```

# FUNCTIONS

```
#!/bin/bash  
  
# Basic function  
  
print_something () {  
    echo Hello I am a function  
}  
  
print_something  
print_something
```

# PASSING ARGUMENTS

- Within the function they are accessible as \$1, \$2, etc.

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

*# Passing arguments to a function*

```
print_something () {
```

```
    echo Hello $1
```

```
}
```

```
print_something Mars
```

```
print_something Jupiter
```

```
#!/bin/bash

print()
{
echo $1
}

print Mars
print Jupiter
```

# RETURN VALUES

```
#!/bin/bash  
  
# Setting a return status for a function  
  
print_something () {  
    echo Hello $1  
    return 5  
}  
  
print_something Mars  
print_something Jupiter  
echo The previous function has a return value of $?
```

```
#!/bin/bash  
  
print()  
{  
    return 5  
}  
  
print  
echo "$?"
```

# TASK1

- Implement a function that takes 2 inputs and performs
  - 1. addition of the 2 numbers
  - 2. Print the result
  - 3. loops over the resulting addition and decrementing the value printing the result inside the loop

# ARRAYS

- myArray=("cat" "dog" "mouse" "frog")  
for str in \${myArray[@]}; do  
    echo \$str  
done

```
#!/bin/bash  
  
array=("1" "2" "3" "4")  
for str in ${array[@]}; do  
    echo $str  
done
```

Print the whole array  
    \${my\_array[@]}

Length of array  
    \${#my\_array[@]}

# ASSIGNMENT

- Create a directory called dip that has 5 files: f1 f2 f3 f4 f5
- Implement a bash script that lists all the files in this directory and deletes the directory with its content using a LOOP

# POSIX

- A POSIX (Portable Operating System Interface) system refers to an operating system that adheres to a set of standards specified by the IEEE (Institute of Electrical and Electronics Engineers) in the POSIX family of standards. The POSIX standards define a set of APIs (Application Programming Interfaces) and other conventions for ensuring compatibility between different Unix-like operating systems.
- Shell and Utilities, File System Structure, Process Control, User and Group IDs, IO Operations, Networking, threads, and system administration