SYSTEM MANAGEMENT AND BASIC SCRIPTING

SOFTWARE ENGINEERING

ROBOTICS CORNER

CONTENTS

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

ROBOTICS CORNER

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

ROBOTICS CORNER

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.

ROBOTICS CORNER

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.

ROBOTICS CORNER

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.

ROBOTICS CORNER

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.

ROBOTICS CORNER

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

ROBOTICS CORNER

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.

ROBOTICS CORNER

MANAGING SERVICES AND DAEMONS

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

ROBOTICS CORNER

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.

ROBOTICS CORNER

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.

ROBOTICS CORNER

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.

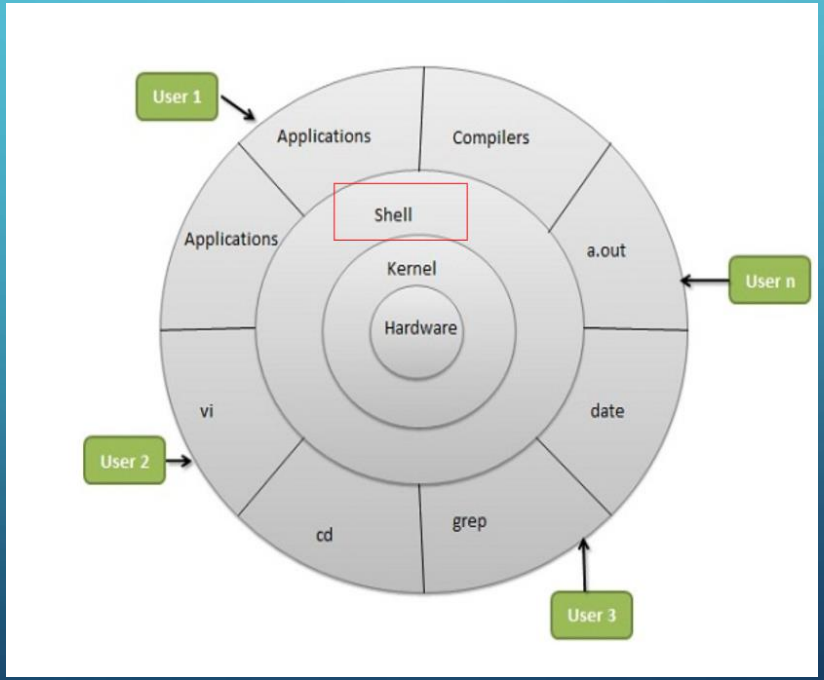
ROBOTICS CORNER

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.



ROBOTICS CORNER

TYPES OF SHELL

Types of shell with varied features

- sh — the original Bourne shell.
- ksh — one of the three: Public domain ksh (pdksh), AT&T ksh or mksh
- bash — 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.
- csh — BSD introduced the C shell, which sometimes resembles slightly the C programming language.
- tcsh — csh with more features. csh and tcsh shells are NOT Bourne- compatible.

ROBOTICS CORNER

SHELL COMPARISON

Software	sh	csh	ksh	bash	tcsh
Programming language	y	y	y	y	y
Shell variables	y	y	y	y	y
Command alias	n	y	y	y	y
Command history	n	y	y	y	y
Filename autocompletion	n	y*	y*	y	y
Command line editing	n	n	y*	y	y
Job control	n	y	y	y	y

*: not by default

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

ROBOTICS CORNER

WHAT CAN YOU DO WITH A SHELL

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

ROBOTICS CORNER

SHELL

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

ROBOTICS CORNER

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.

ROBOTICS CORNER

WHEN NOT TO USE SHELL SCRIPTING

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

ROBOTICS CORNER

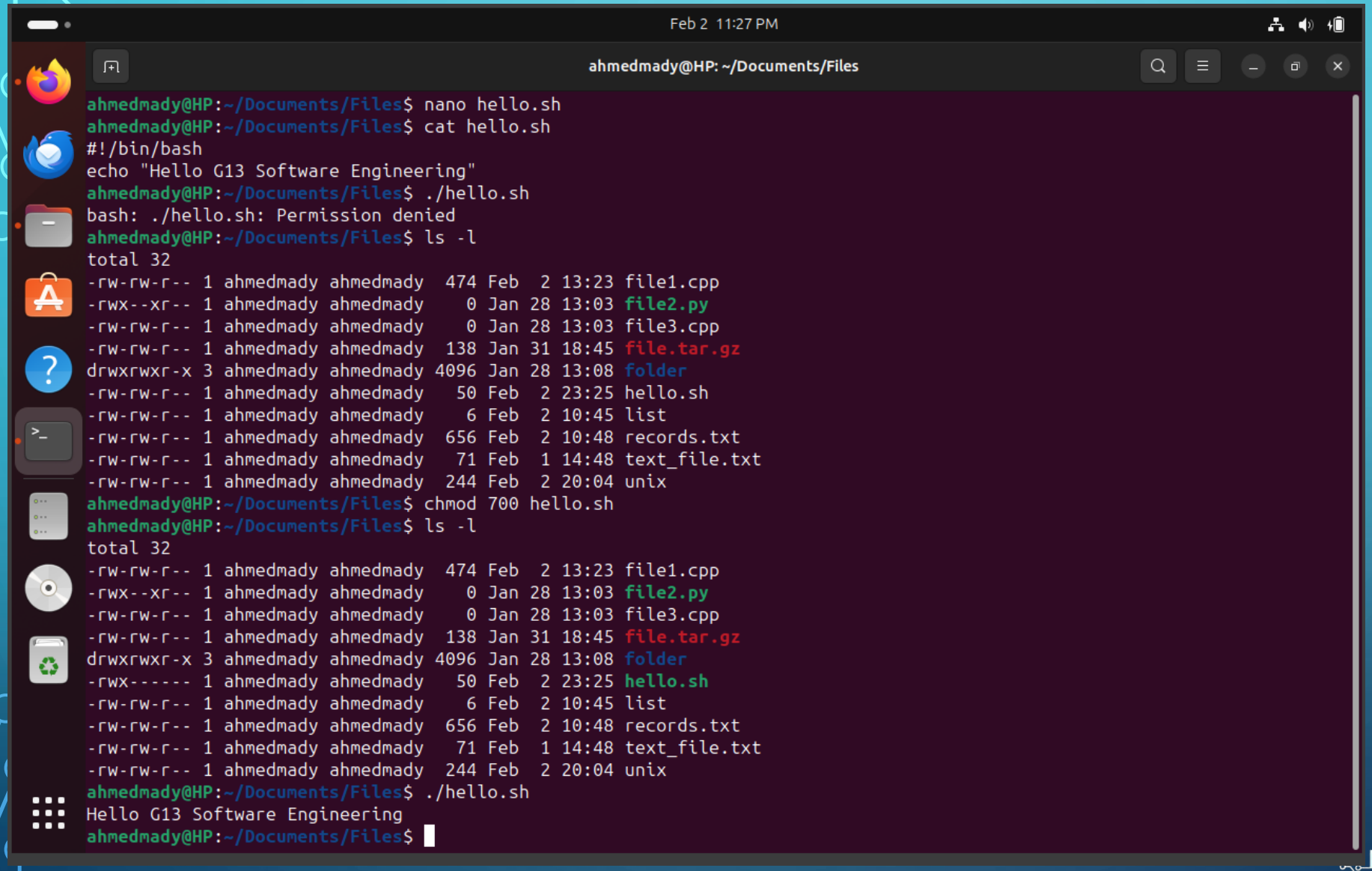
HELLO

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

ROBOTICS CORNER



ROBOTICS CORNER

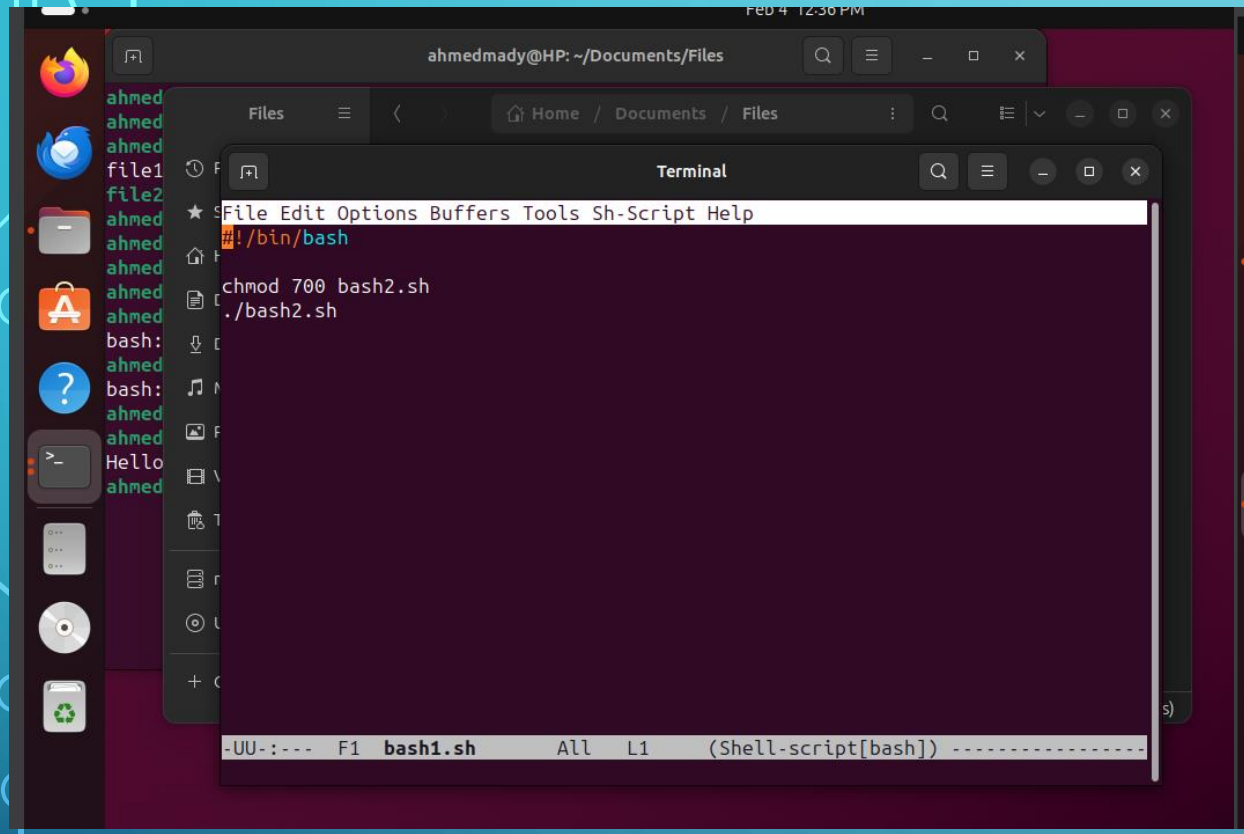
INTERACTIVE VS NONINTERACTIVE SHELL

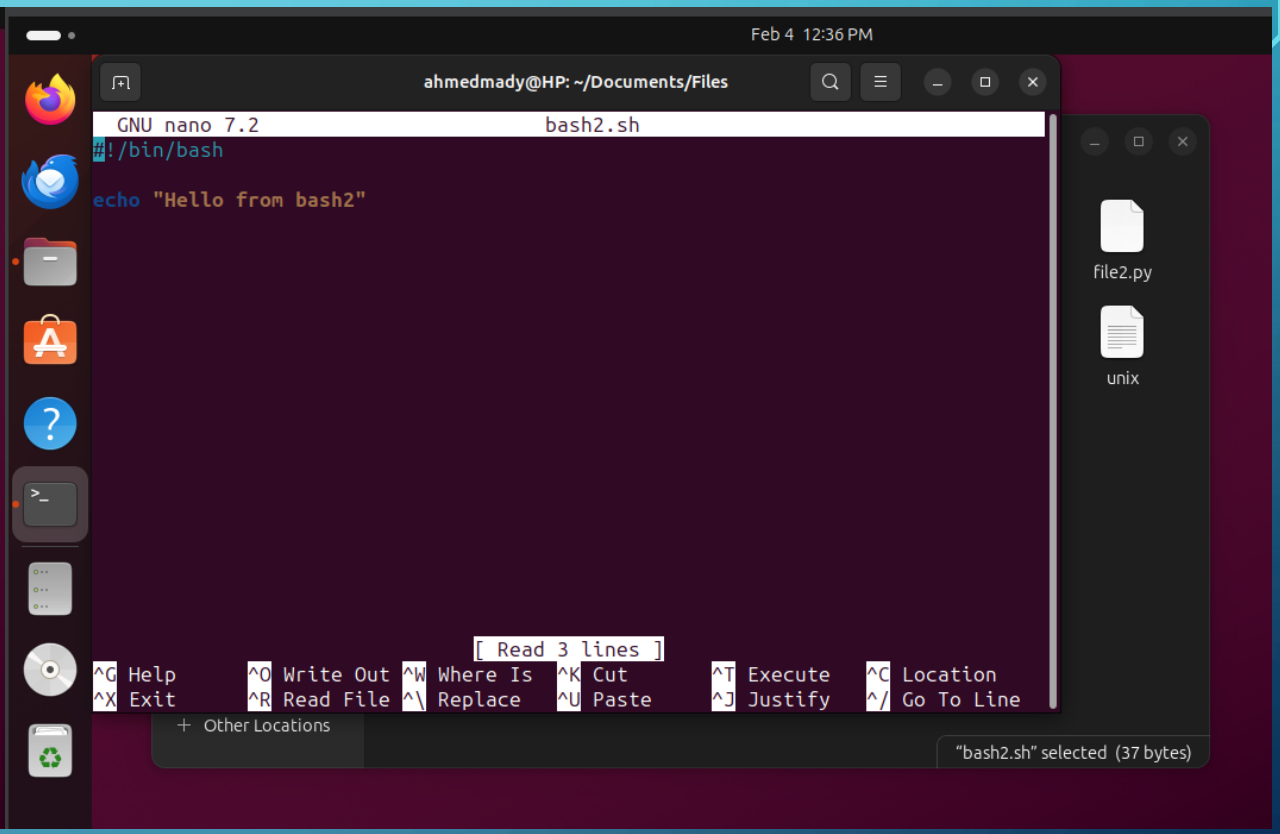
- An interactive shell refers to a command-line interface that allows users to interact with the computer's operating system by typing commands in real-time and receiving immediate feedback. In an interactive shell, users can enter commands, execute programs, and perform various tasks by typing text-based 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.

ROBOTICS CORNER

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.





ROBOTICS CORNER

VARIABLES IN SHELL SCRIPTING

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

ROBOTICS CORNER

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.

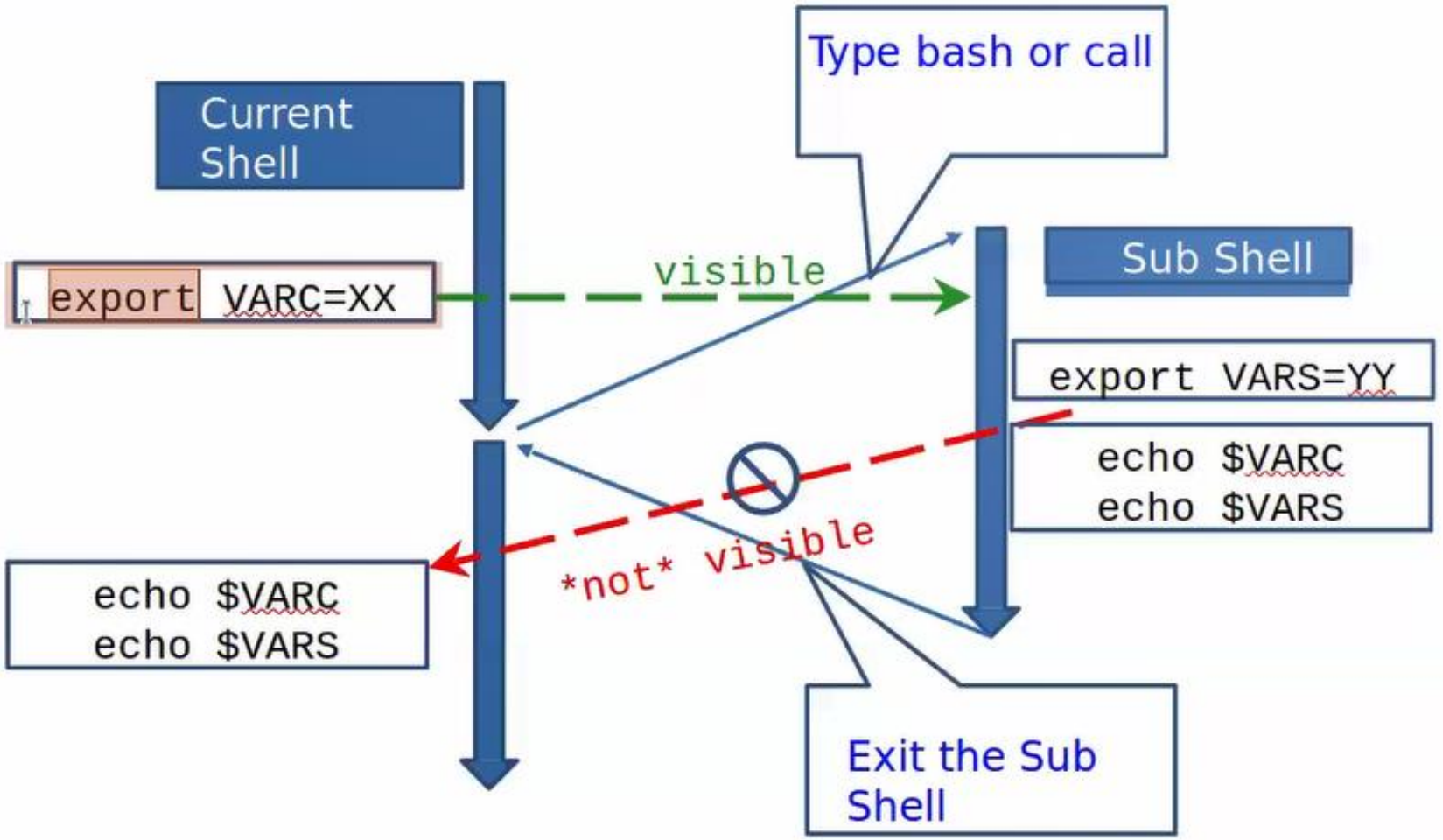
ROBOTICS CORNER

LOCAL AND GLOBAL VARIABLES

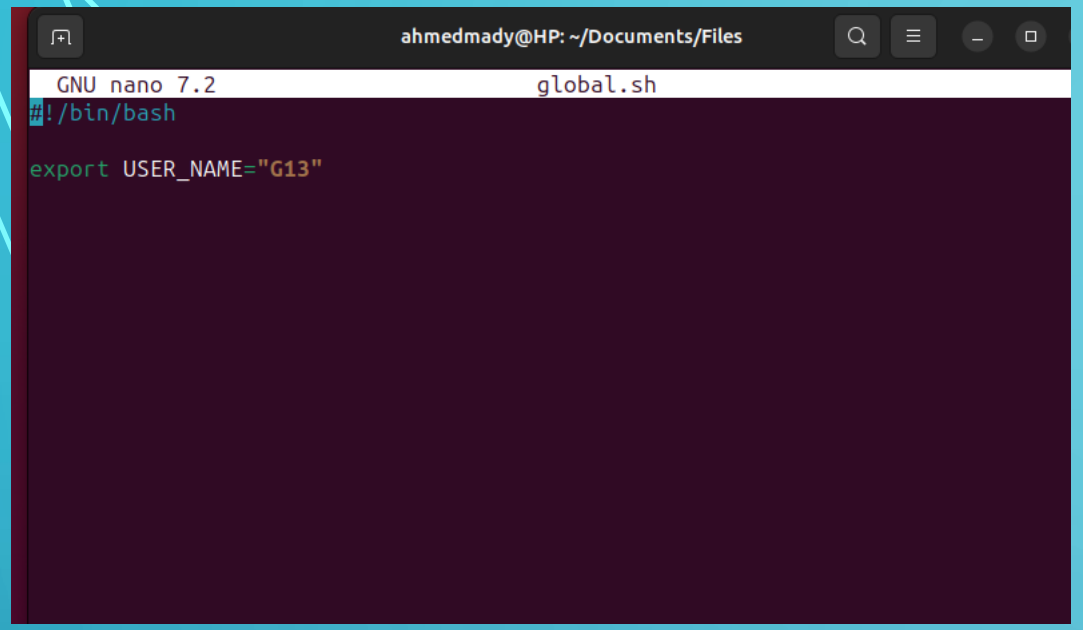
- Variables created and used inside a shell are only local to the shell and can't be seen outside this shell.
- If a variable is created in the terminal and used in a process for example, this variable is local to this terminal and can't be seen elsewhere.
- Global variables are variables that are accessible 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)

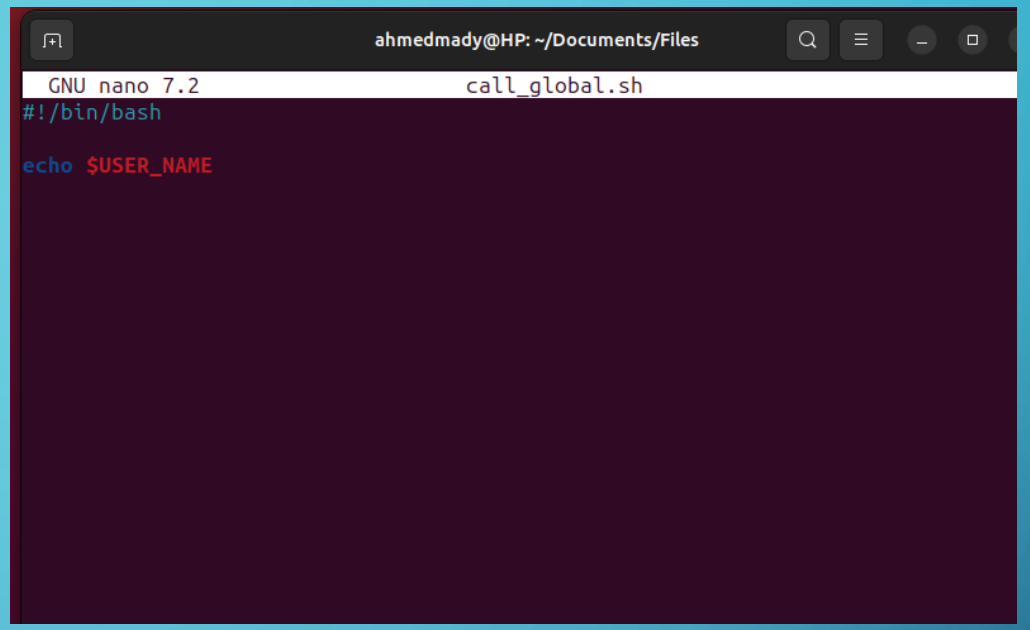
ROBOTICS CORNER

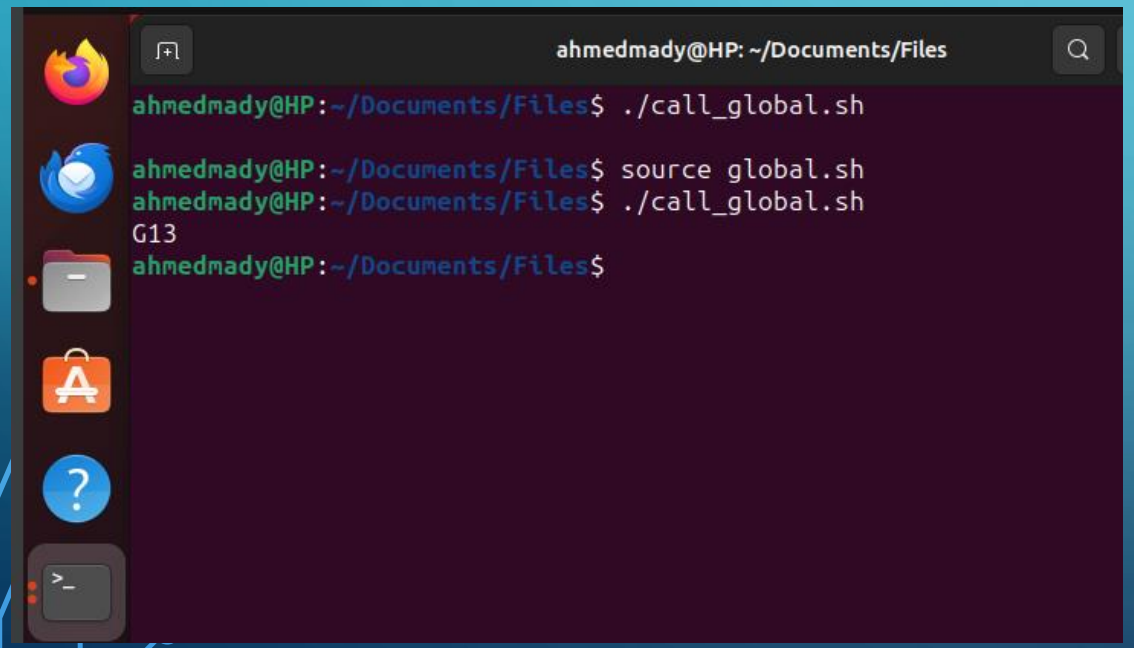
Global and Local Variables
- current shell and subshell

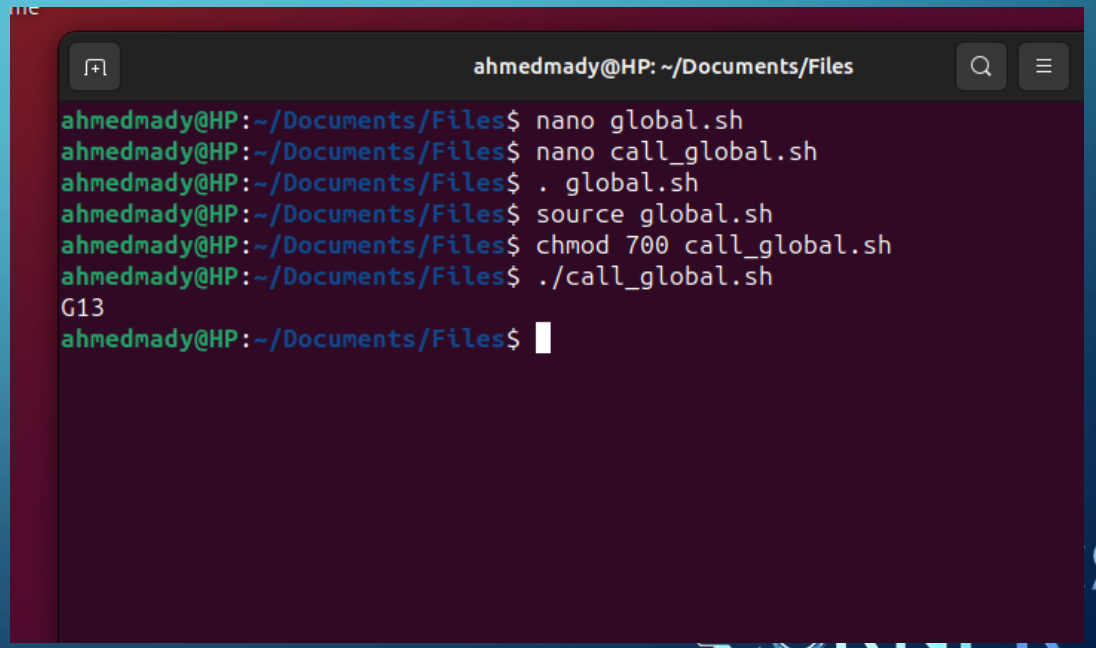


ROBOTICS CORNER


ahmedmady@HP: ~/Documents/Files
GNU nano 7.2 global.sh
#!/bin/bash
export USER_NAME="G13"


ahmedmady@HP: ~/Documents/Files
GNU nano 7.2 call_global.sh
#!/bin/bash
echo $USER_NAME


ahmedmady@HP: ~/Documents/Files
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
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 ‘$’, ‘`’, ‘\’

ROBOTICS CORNER

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

ROBOTICS CORNER

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 `source` (for bash only)
|	Pipe: use the output of a command as the input of another one
> <	Redirections (`0<`: standard input; `1>`: standard out; `2>`: standard error)

ROBOTICS CORNER

$?	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

ROBOTICS CORNER

INTEGER ARITHMETIC OPERATIONS

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

ROBOTICS CORNER

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)"
```

CONDITIONAL STATEMENTS

```
#!/bin/bash

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

ROBOTICS CORNER

Operation	bash
File exists	if [-e test]
File is a regular file	if [-f test]
File is a directory	if [-d /home]
File is not zero size	if [-s test]
File has read permission	if [-r test]
File has write permission	if [-w test]
File has execute permission	if [-x test]

Operation	bash
Equal to	if [$a == $b]
Not equal to	if [$a != $b]
Zero length or null	if [-z $a]
Non zero length	if [-n $a]

Operation	bash
Equal to	if [1 –eq 2]
Not equal to	if [$a –ne $b]
Greater than	if [$a –gt $b]
Greater than or equal to	if [1 –ge $b]
Less than	if [$a –lt 2]
Less than or equal to	if [$a –le $b]

Operation	Example
! (NOT)	if [! –e test]
&& (AND)	if [-f test] && [-s test] if [[-f test && -s test]] if (-e test && ! –z test) if
|| (OR)	if [-f test1] || [-f test2] if [[-f test1 || -f test2]]

ROBOTICS CORNER

if condition examples

Example 1:

```
read input
if [ $input == "hello" ]; then
    echo hello;
else echo wrong ;
 fi
```

Example 2

```
touch test.txt
if [ -e test.txt ]; then
    echo “file exist”
elif [ ! -s test.txt ]; then
    echo “file empty”;
fi
```

What happens after

```
echo “hello world” >>
test.txt
```

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

ROBOTICS CORNER

for loop examples

Exmaple1:

```
for arg in `seq 1 4`
 do
   echo $arg;
   touch test.$arg
done
```

How to delete test files using a loop?

```
rm test.[1-4]
```

Example 2:

```
for file in `ls /home/$USER`
 do
   cat $file
 done
```

ROBOTICS CORNER

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.

```
read counter
while [ $counter -ge 0 ]
 do let
counter--
   echo $counter
 done
```

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)

```
read counter
until [ $counter -lt 0 ]
 do let
counter--
   echo $counter
 done
```

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 a local variables using the local command, which is invisible outside the function

```
local var=value
```

FUNCTIONS

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

ROBOTICS CORNER

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
```

ROBOTICS CORNER

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 $?
```

ROBOTICS CORNER

ARRAYS

- myArray=("cat" "dog" "mouse" "frog")

```
for str in ${myArray[@]}; do
  echo $str
done
```

```
Print the whole array
    ${my_array[@]}
Length of array
    ${#my_array[@]}
```

ROBOTICS CORNER

TASK

- 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

ROBOTICS CORNER

```
#!/bin/bash

# Set the directory path
directory="/Documents/dip"

# Change to the specified directory
cd "$directory" || { echo "Error: Directory not found"; exit 1; }

# Get the list of files in the directory
files=(*)

# Loop through the array and delete each file
for file in "${files[@]}"; do
    if [ -f "$file" ]; then
        rm "$file"
        echo "Deleted: $file"
    fi
done
```

ROBOTICS CORNER

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 adminstration

ROBOTICS CORNER