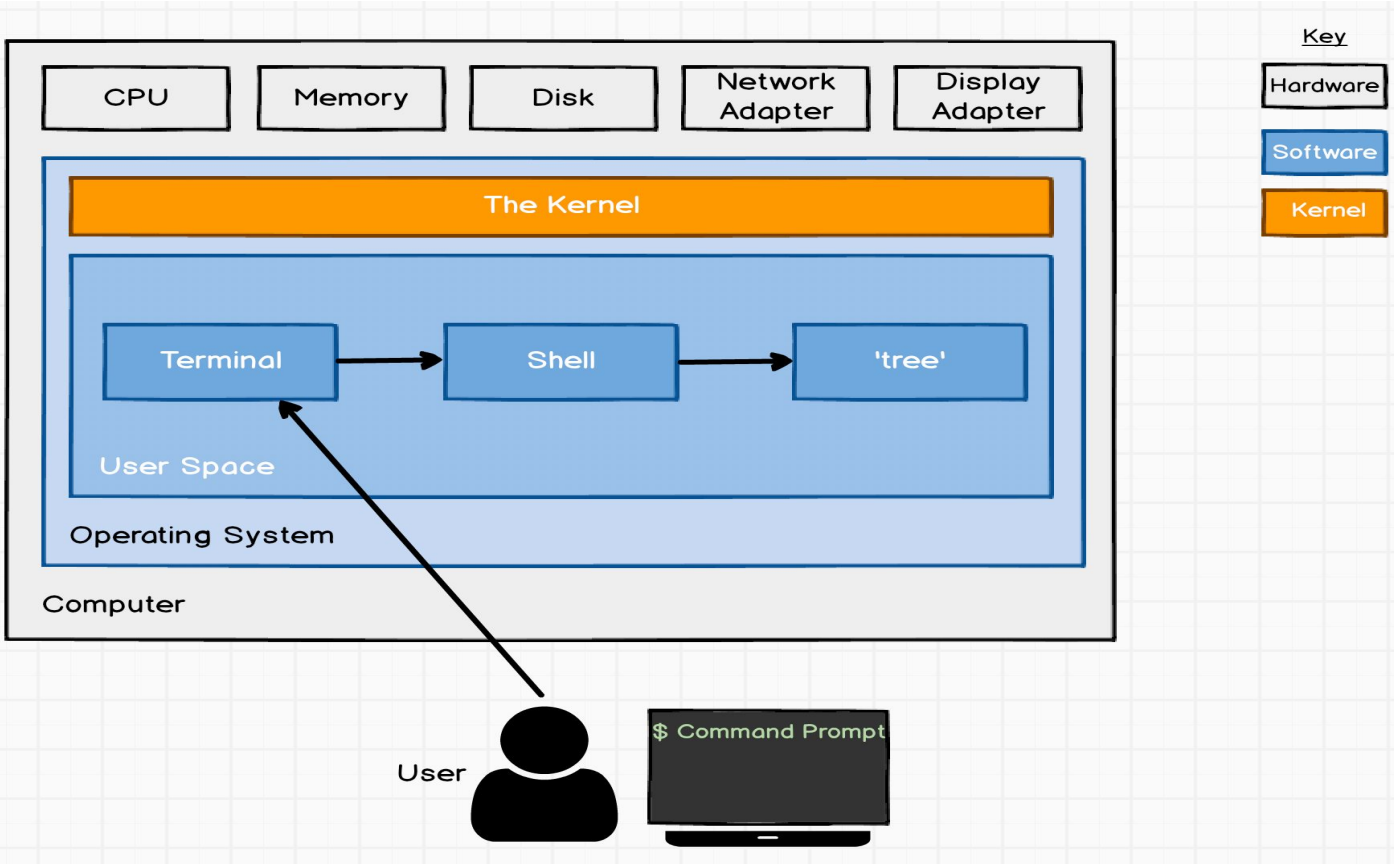
Bash Scripting

Session 4

Table of contents

01

Step back!


CPU
Memory
Disk
Network Adapter
Display Adapter
The Kernel
Terminal
Shell
'tree'
User Space
Operating System
Computer
User
$ Command Prompt
Key
Hardware
Software
Kernel

Terminal

A terminal is a text-based interface that provides a window or access point to a shell. It is the visual environment where **users type commands** and view the output from the shell and the underlying system.

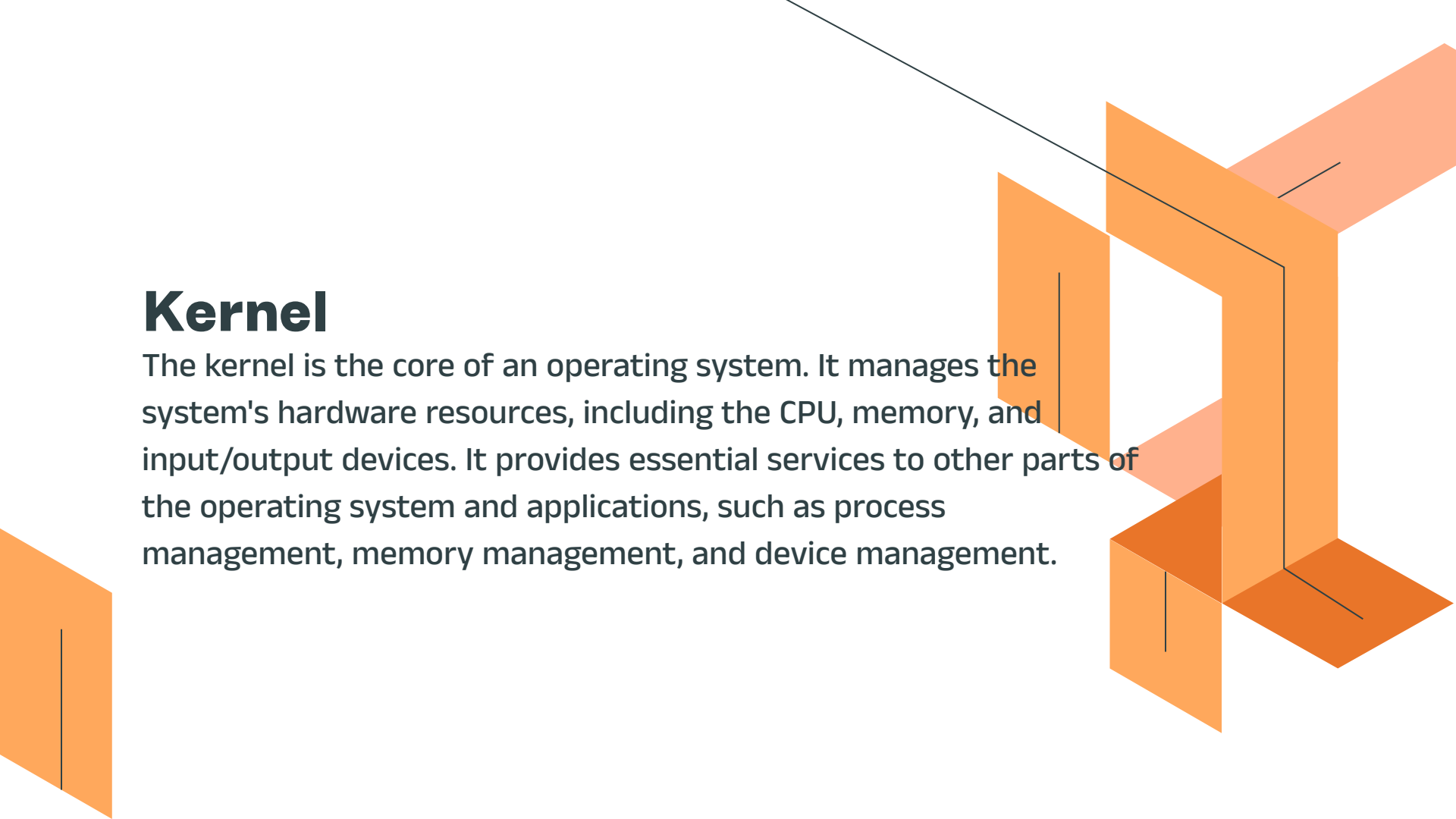
Shell

A shell is a program that provides a user interface to the operating system's kernel. It acts as **a command-line interpreter**, allowing users to interact with the system by typing commands. The shell interprets these commands and translates them into system calls that the kernel can execute. Popular shells include **Bash**, **Zsh**.

.

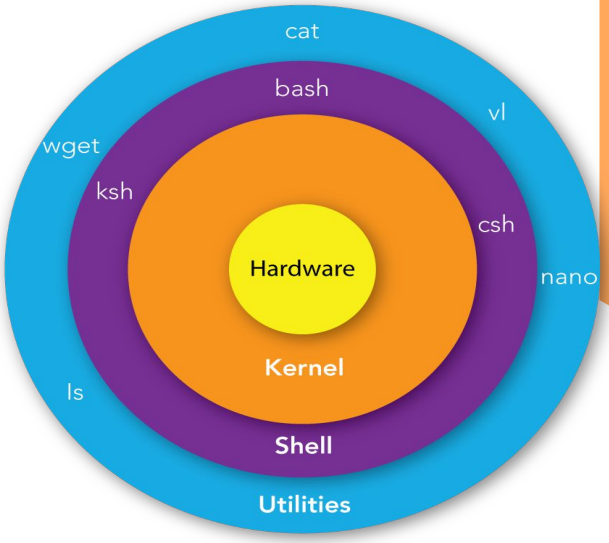
Kernel

The kernel is the core of an operating system. It manages the system's hardware resources, including the CPU, memory, and input/output devices. It provides essential services to other parts of the operating system and applications, such as process management, memory management, and device management.



Command Life cycle!-

In a very simple way! You use the **Terminal** to type commands into the **Shell,** which then **interprets** those commands and communicates with the **Kernel** to **perform actions on** the computer's **hardware and resources**.



Shell

The Shell is a program that acts as an interface between the user and the kernel. When you type a command in the terminal and press Enter, the shell takes over. It **interprets the command, checks its syntax**, and if it's valid, **converts it into a form the kernel can understand.** If the command is incorrect, it returns an error message. In essence, the shell **translates user commands into system actions**. There are different types of shells like **Bash** (the default on most Linux and macOS systems), TCSH, KSH, and others. Overall, the shell is a **command interpreter** that helps users access system services by converting commands into system calls.

Command not found, where??

PATH variable

defining directories where executables are searched

```
nasai@NASAI:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:
/usr/games:/usr/local/games:/snap/bin:/snap/bin
```

If command written is not in one of these directories, “Command not found” will appear-

02

What is Bash Scripting ?

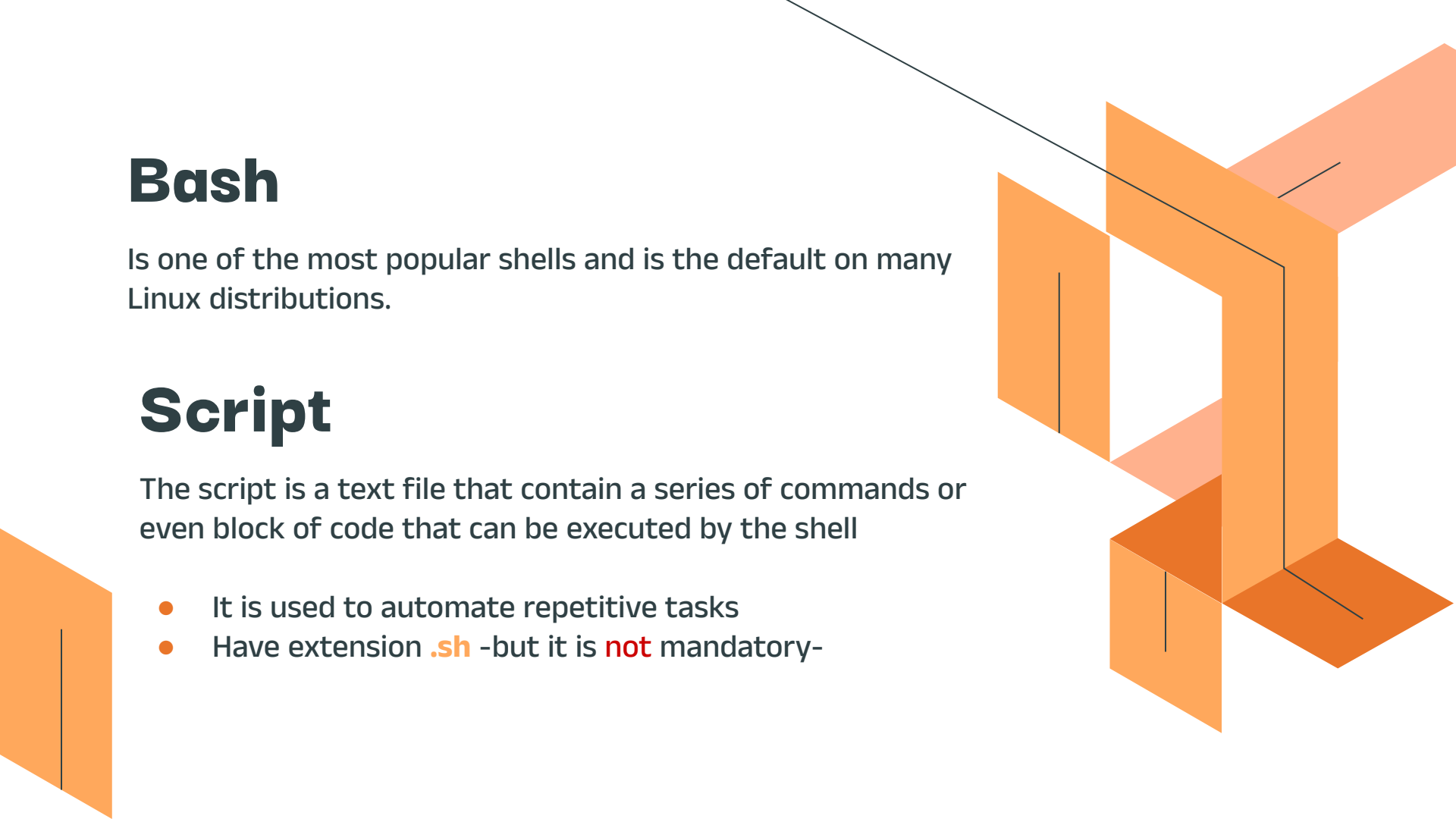
Bash

Is one of the most popular shells and is the default on many Linux distributions.

Script

The script is a text file that contain a series of commands or even block of code that can be executed by the shell

- It is used to automate repetitive tasks
- Have extension .sh -but it is not mandatory-

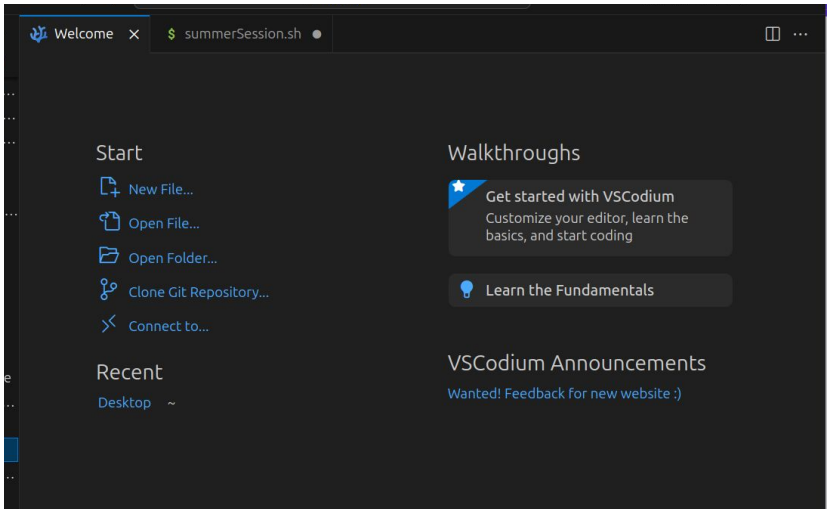


03

How to Write & Execute script?

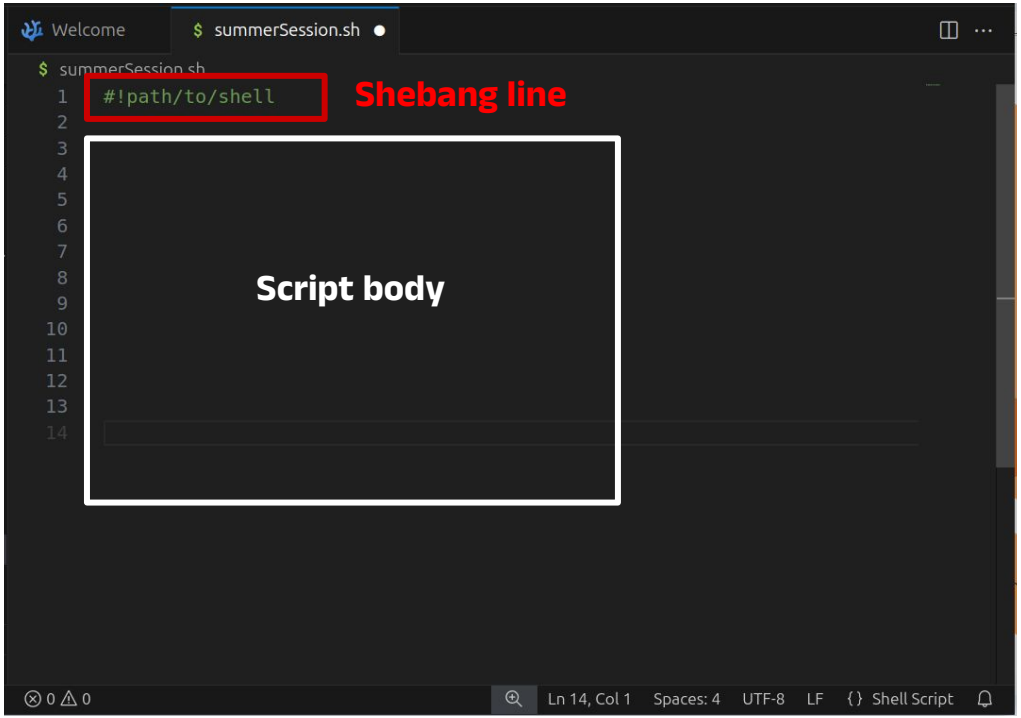
How to create your own script

- Open your text editor, choose the folder you want to keep your script in
- Create a file with extension .sh



Steps to create your own script

- Script file structure should be as follow:



Shebang [#!]

- The [#!path/to/shell] line
- Should be the first line in your script
- It is used to tell the operating system which shell to use to run this script
- By typing this line you do not need to specify the shell at the command line before executing the file.

Ignoring Shebang : If you do not specify an interpreter line, the default is usually the /bin/sh. But, it is recommended that you set #!/bin/bash line for your bash scripts.

How to execute your script?

First way:

- Make your script file executable

  chmod +x pathToScriptFile

- Run your script by specifying its path

  pathToFile/scriptFile.sh

```
nasai@NASAI:~$ chmod +x sessionScript1.sh
nasai@NASAI:~$ ./sessionScript1.sh
This is My first script to execute!
```

First way complete scenario:

- The script was not executable, so when i tried to run it, it gave me “Permission denied”
- I changed permissions of script so it became executable
- Run again?
- Script output appears!

```
nasai@NASAI:~$ ls -l sessionScript1.sh
-rw-rw-r-- 1 nasai nasai 55 Jul 24 12:19 sessionScript1.sh
nasai@NASAI:~$ ./sessionScript1.sh
bash: ./sessionScript1.sh: Permission denied
nasai@NASAI:~$ chmod +x sessionScript1.sh
nasai@NASAI:~$ ./sessionScript1.sh
This is My first script to execute!
```

How to execute your script?

Second way:

- By specifying the shell explicitly

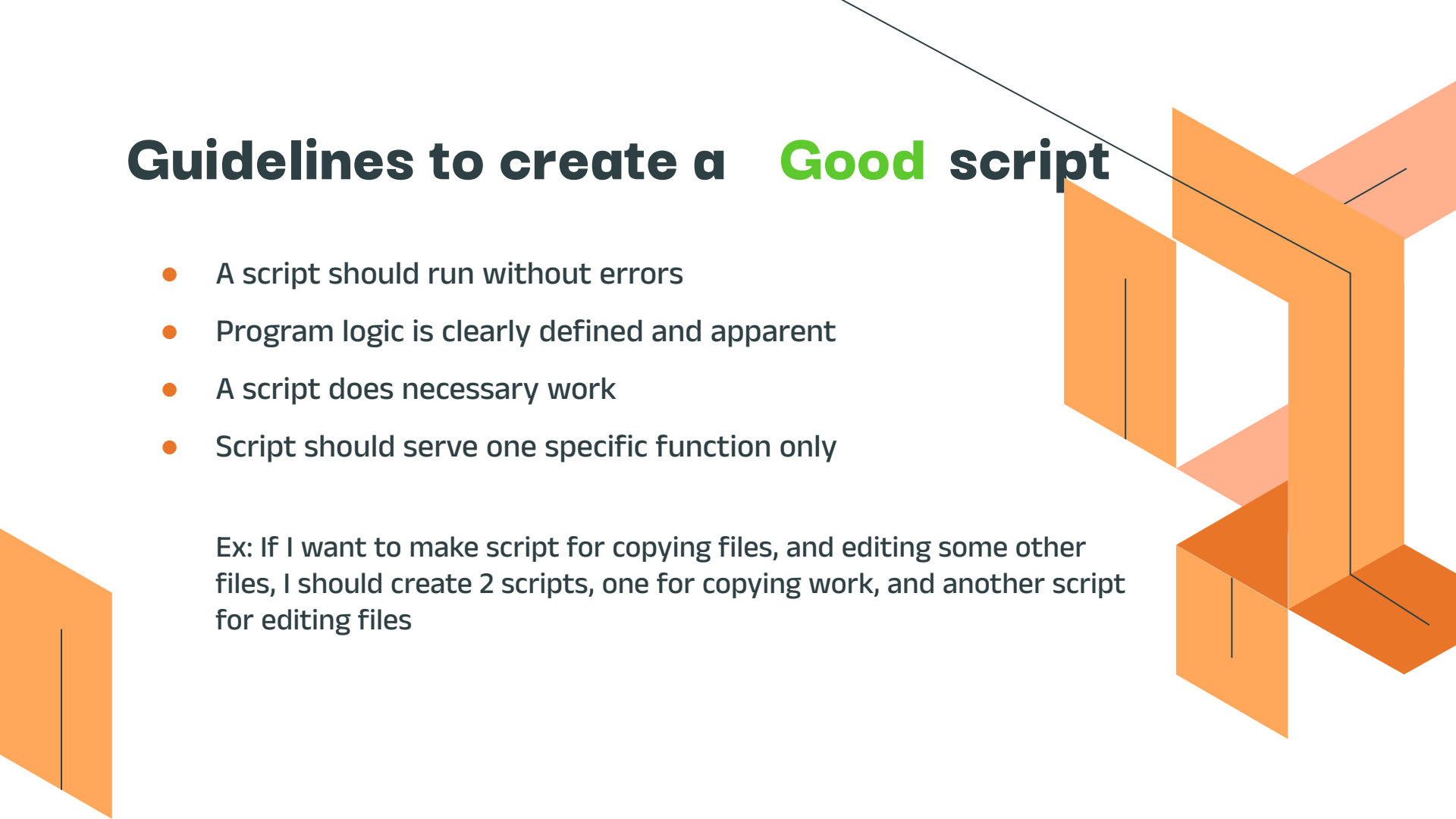
bash pathToFile/scriptFile.sh

```
nasai@NASAI:~$ ls -l sessionScript1.sh
-rw-rw-r-- 1 nasai nasai 55 Jul 24 12:19 sessionScript1.sh
nasai@NASAI:~$ bash sessionScript1.sh
This is My first script to execute!
```

Guidelines to create a Good script

- A script should run without errors
- Program logic is clearly defined and apparent
- A script does necessary work
- Script should serve one specific function only

Ex: If I want to make script for copying files, and editing some other files, I should create 2 scripts, one for copying work, and another script for editing files



04

Variables

Shell Variable

A shell variable is used to store some value. It could be an integer, filename, string, or some shell command itself.

Declaring Variables

To declare a variable, you write the variable name followed by = sign then assign the value wanted

- **without** any spaces before and after the = sign

Accessing the variable

To access the variable you add a dollar sign $ before the variable name

- ex: $varname

Declaring and Accessing Variables

This examples declares the difference between accessing the variable with and without the dollar sign

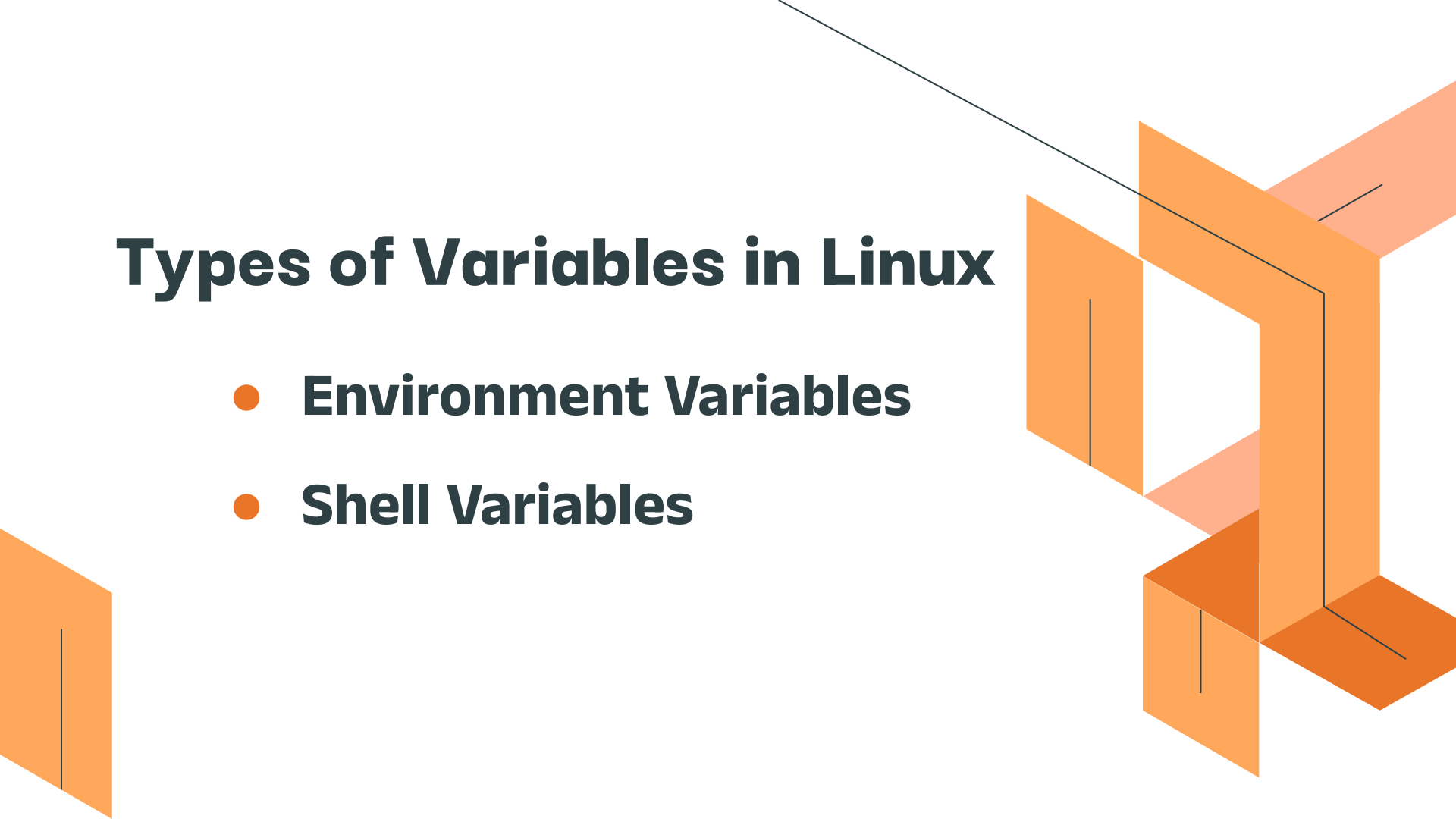
```
var=123

echo "first trail with dollar sign $"
echo "var value = $var"

echo " "

echo "Second trial without dollar sign"
echo "var value = var"
```

Output

```
first trail with dollar sign $
var value = 123

Second trial without dollar sign
var value = var
```

Types of Variables in Linux

- Environment Variables
- Shell Variables



Environment Variables

Purpose:

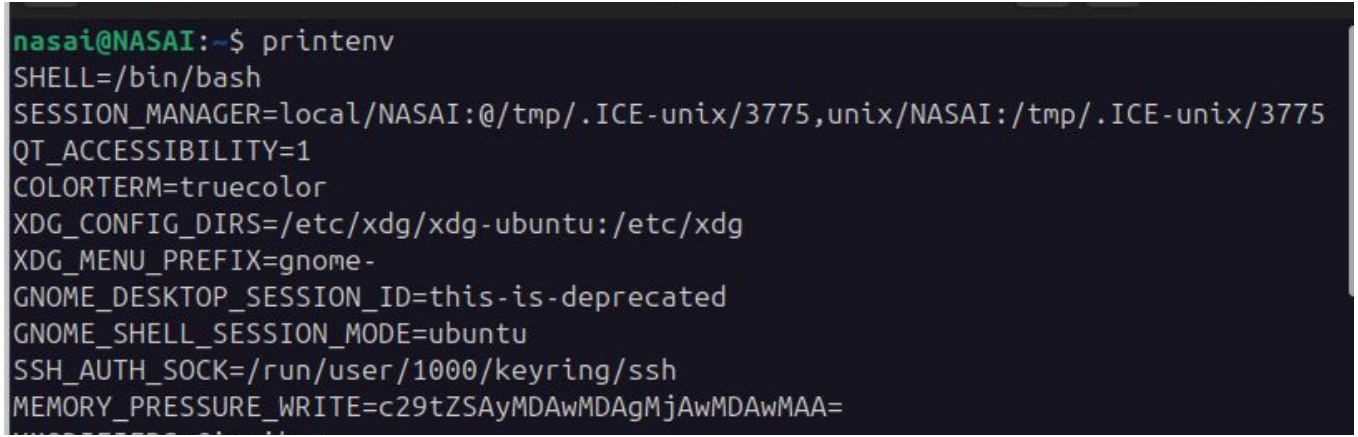
- used to **store system-wide or user-specific configuration** information that needs to be accessible by various programs and processes.
- Examples: **PATH** (defining directories where executables are searched), **HOME** (user's home directory), and **SHELL** (default shell program).

Scope:

- **Globally available** to the **current shell and any child processes** or shells spawned from it. They are inherited by subsequent processes.

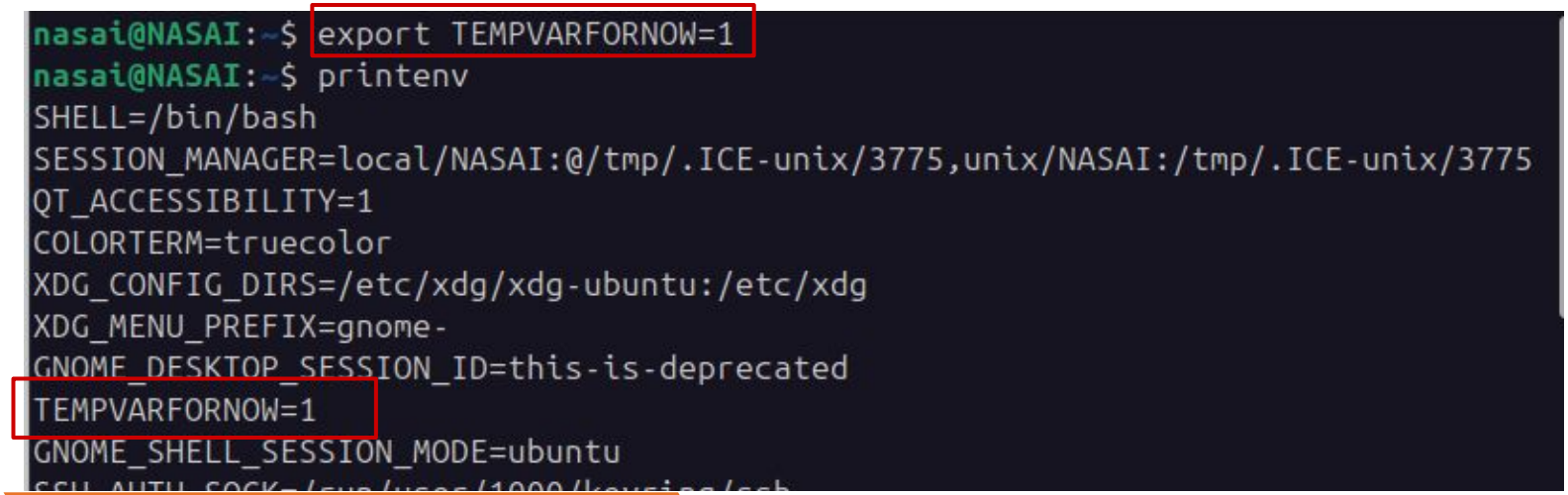
Environment Variables Management

- To list environment variables, use **`env`** or **`printenv`**.



Environment Variables Management

- To set a temporary environment variable for the current session, use **`export VARIABLE_NAME=value`**.



Shell Variables

Purpose:

- Are often used for **temporary storage of data within a script** or for managing settings specific to the current interactive shell session.

Scope:

- These variables are **local** and **apply only to the current shell instance** in which they are defined. They are not inherited by child processes.

Rules to name the variable in our shell

- Variable names are case sensitive so: var != Var
- could contain any alphabet (a-z, A-Z), any digits (0-9), and an underscore (_)
- Must start with an alphabet or underscore, never start with a number
- You can not use [! , - , *] in variable names as they have special meaning to the shell
- **Valid Variable Names**

  `abc`

  `_Abc`

  `abc123`

- **Invalid Variable Names**

  `2ABC` `&ABC`

  `!abc`

  `$Abc`

Variable Types

Shell variables are “Typeless”, as they accept any kind of data with any values, like that!

- varname="Hello World"
- varname=hi
- varname=1
- varname=3.142
- varname="3.142"
- varname=123abc

Command Substitution

You can assign a command to a variable, so the variable will be able to store the output of this command

variable=$(command)

Example

This Example stores the output of date command in var varibale

```
var=$(date)

echo "Var value now is= $var"
```

Output

```
Var value now is= Thu Jul 24 12:58:28 PM EEST 2025
```

Constant Variables

To declare constant variable in the shell you write **readonly** before the variable name

- This make the value of this variable is **unchangeable**

Example

Try to change variable [var] value, but will not success.

```
readonly var=12

var=13

echo "var value =$var"
```

Output

```
sessionScript1.sh: line 5: var: readonly variable
var value =12
```

How to Unset Variables?

The unset command directs a shell to **delete a variable and its stored data**

unset variableName

Example

Here I wanted to delete [var1] variable

```
var1="Devil"
var2=23
echo $var1 $var2

unset var1

echo $var1 $var2
```

Output

```
Devil 23
23
```

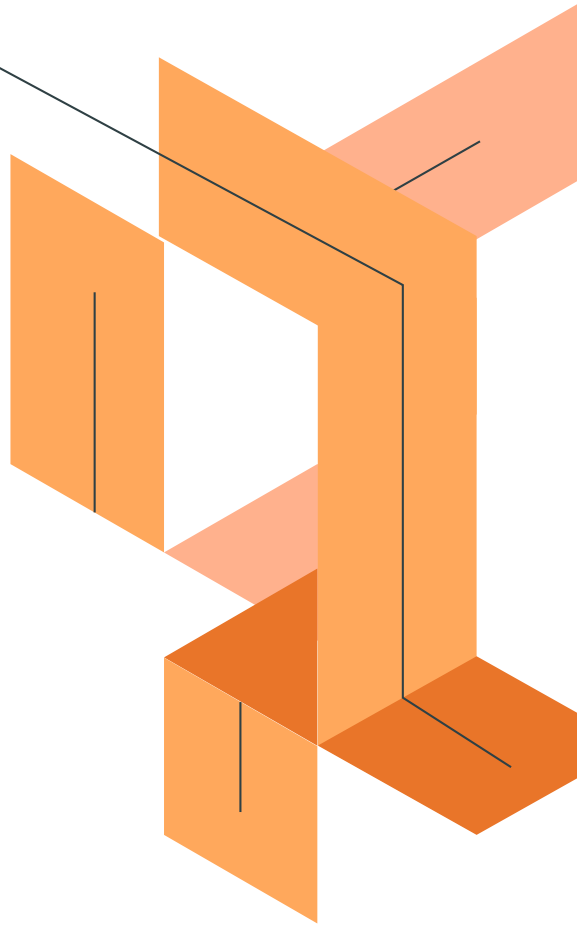
Do you think that your script is just a hard-coded block?-

06

Take User Input

Take user input by 2 ways

- Positional Parameters
- Read Command



Positional Parameters

Positional parameters are a **series of special variables** ($0 through $N) that **contain the contents of the command line.**

They are **accessed** inside the script by dollar sign **$+ number** (from 0 to 9), where:

- **$1** : first argument
- **$2** : second argument
- **$3** : third argument, etc..

QUESTION!!

-> What if I passed **more** than 9 params?

Answer: Argument will be accessed by dollar sign **$ + { }**

- **${10}** : Tenth argument

Positional Parameters

Special parameters:

- **$0** : returns the **Name of script**
- **$#** : returns total **number of arguments** (without script name)
- **$*** : returns all **arguments passed** as **one word** (without script name)

Positional Parameters Example

```
nasai@NASAI:~$ bash sessionScript1.sh Nouran OSC Linux
```

```
#!/bin/bash

echo "The total no of args are: $#"
echo "The script name is : $0"
echo "The first argument is : $1"
echo "The second argument is: $2"
echo "[using *] The total argument list is: $*"
```

```
The total no of args are: 3
The script name is : sessionScript1.sh
The first argument is : Nouran
The second argument is: OSC
[using *] The total argument list is: Nouran OSC Linux
```

Positional Parameters

Maximum number of arguments?

- No maximum number of arguments here, but we are limited by The system's **maximum command-line length**
- We can check the max length by

getconf ARG_MAX

Note that:

- Spaces between arguments take place in size count
- As well as null terminators

Example:

```
./myscript.sh one two three
```

The total bytes include:

- "./myscript.sh" (the command itself)
- "one", "two", "three" (arguments)
- The spaces between them (used by the shell to separate arguments)
- The null terminators after each argument
- All environment variables at the time

read Command

read Command is used to read input from user during the script execution

```
read  <options>  <arguments>
```

Command options:

- **-p <prompt>** : Outputs the prompt string before reading user input.
- **-s** : Does not echo the user's input.

read Command Example

This script asks the user to enter his first and second name using read command -without options-

```
#!/bin/bash

echo "Enter your first name"
read fname

echo "Enter your second name"
read sname

echo "Name entered is: $fname $sname"
```

Output

```
nasai@NASAI:~$ bash sessionScript1.sh
Enter your first name
Nouran
Enter your second name
Ahmed
Name entered is: Nouran Ahmed
```

read Command Example

This script asks the user to enter his first and second name using read command -with options-

```
#!/bin/bash

read -p "Enter your username: " username

#compination of p & s options
read -sp "Enter your password: " password

echo
echo "Your username is: $username"
echo "Your password is: $password"
```

Output

```
nasai@NASAI:~$ bash sessionScript1.sh
Enter your username: Nouran
Enter your password:
Your username is: Nouran
Your password is: 123
```

07

Difference between

Single quotes ‘ ‘ & Double quotes “ “

Single quotes ‘ ‘

- Wherever you wrote between single quotes, you will get it as it is

What you see is what you get

- **No variable expansion**, “$var” is echoed as “$var”, variable will not be substituted by its value
- **No command substitution**

```
person="Nouran"

singleQ_variable='Hello $person'

echo "This is the single quote example"
echo $singleQ_variable
```

```
This is the single quote example
Hello $person
```

Double quotes “ “

- The variable is substituted by its value
  -> Ex: “$var” will be echoed as “23” (its value)
- Command substitution takes place
- **+ve extra point:**
  -> Escape characters are recognized : \n (new line) - \t (tab)

```
person="Nouran"

doubleQ_variable="Hello $person"

echo "This is the double quote example"
echo $doubleQ_variable
```

```
This is the double quote example
Hello Nouran
```

Guess the Answer!

Question 1

What happens when I write a command in the Terminal and press Enter?

Question 2

What is the extension of script file?

Question 3

First Line of Bash Script?

Question 4

What is the output?

```
var=”Session “
var2=4
echo “var var2”
```

Question 5

What is the output?

```
readonly var=”Hello”
var=”OSCian”
echo “$var”
```

Question 6

What is the output?

```
var=”Session “
var2=4
echo “var var2”
```

Question 7

What is the output?

```
var=”Hello “
var2=”OSCian”
echo ‘ “$var $var2” ’
```

Question 8

What is the Output?

```
~$ var=100
~$ bash
~$ var=45
~$ exit
~$ echo “Var value is: $var”
```

Question 9

Will the “VARFORTEST” appear in output of printenv?

```
~$ export VARFORTEST=4
~$ printenv
```

Let's write our first script together

Open your laptops!!

- Write a welcome msg for your script user
- Ask for his/her name -use read cmd options
- Ask for the age
- Thank the user then echo his/her info

Break

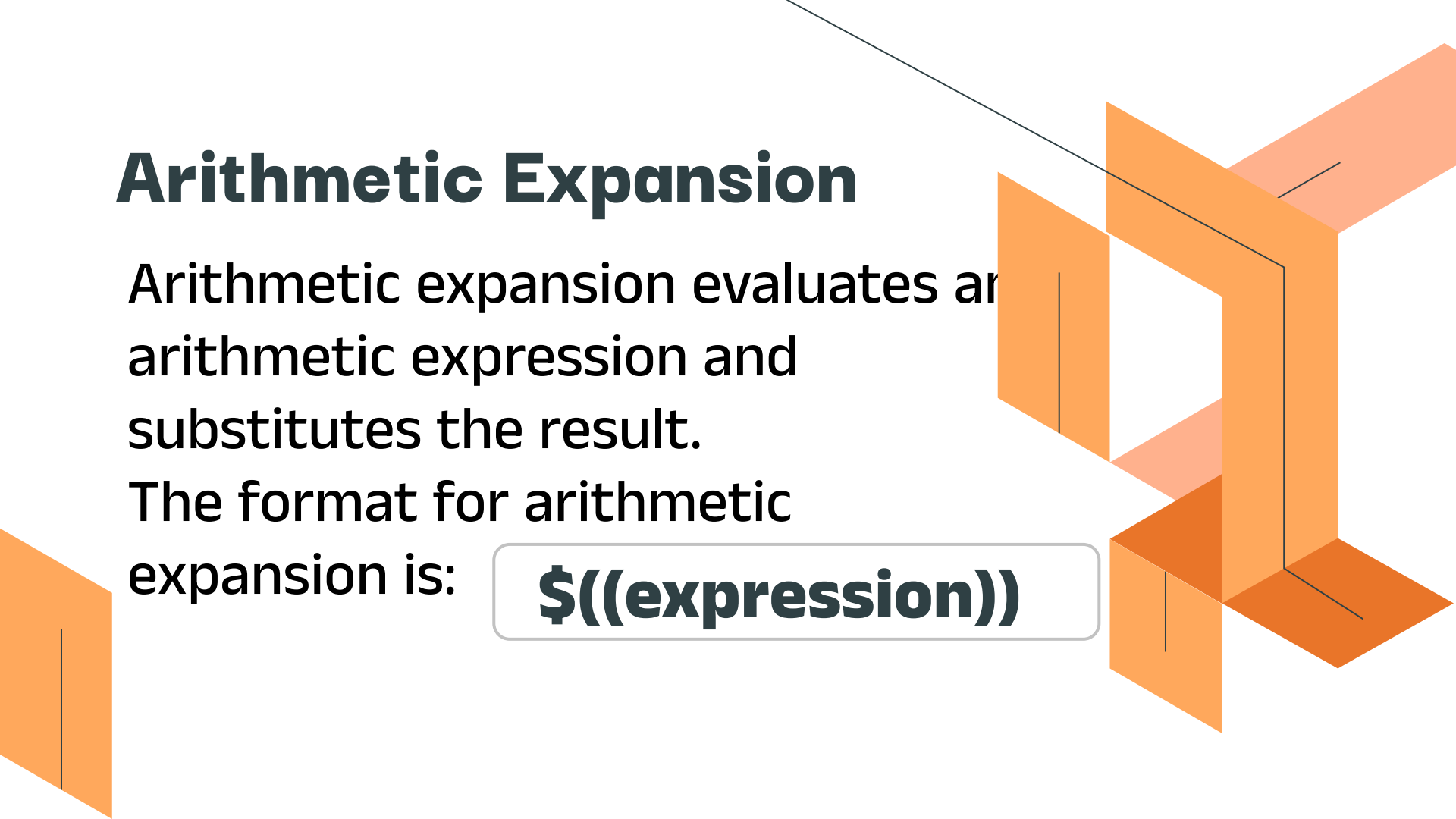
07

Arithmetic Expansion

Arithmetic Expansion

Arithmetic expansion evaluates ar
arithmetic expression and
substitutes the result.
The format for arithmetic
expansion is:

$((expression))



Arithmetic operations you can do in BASH

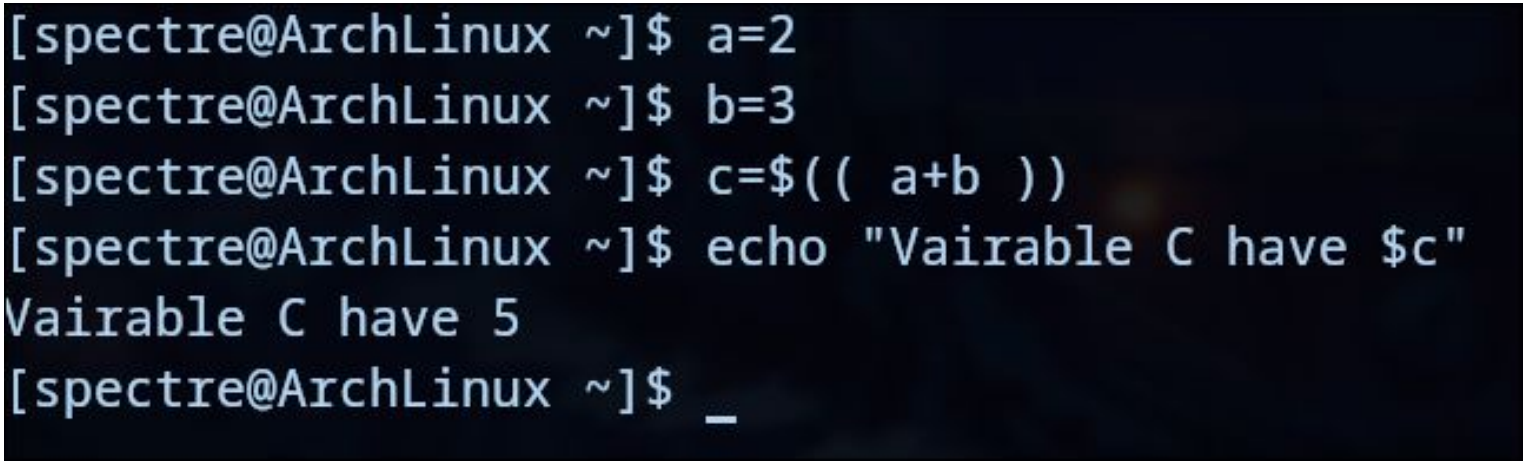
Addition	a + b	a plus b
subtraction	a - b	a minus b
Multiplication	a * b	a times b

Arithmetic operations you can do in BASH

Integer Division	a / b	a divided by b
Modulo	a % b	The integer remainder of a divided b
Exponentiation	a ** b	a^b a to the power of b

Arithmetic Expansion Example

This example assign an arithmetic expression into a variable then return the output of it



Arithmetic Expansion Example

```
[spectre@ArchLinux ~]$ a=2
[spectre@ArchLinux ~]$ b=3
[spectre@ArchLinux ~]$ c=$(( a+b ))
[spectre@ArchLinux ~]$ echo "Vairable C have $c"
Vairable C have 5
[spectre@ArchLinux ~]$ echo 'Vairable C have $c'
Vairable C have $c
[spectre@ArchLinux ~]$ _
```

08

Control Structures in Bash

Control Structures in Bash

1. If Conditions
2. Case Statements
3. Loops
4. Functions

if conditions

if condition

If condition used to run some code only when a condition (a test) is true

basic syntax:

Method 1

```
if [[ condation ]]
then
   # some code to do
fi
```

Method 2

```
if [[ condation ]] ; then
   # some code to do
fi
```

if condition

If else if syntax

```
If [[ condition ]] ; then
    # some stuff to do
else
    # other stuff to do
fi
```

If else if syntax

```
if [[ condition ]] ; then
    # some stuff to do
elif [[[ condition ]] ; then
    # other stuff to do
fi
```

Conditions

A test we make to execute some commands based on it

conditional expressions should be placed inside square braces [[Conditions]] with spaces around them.

Conditions

What we will discuss in Conditions ?

- Comparing String Variables
- Comparing Numerical Variables
- File Conditions

Comparing String Variables

Expression in C	Expression in BASH	Description
a == b	$a = $b or $a == $b	Checks if **a** is equal to **b**
a != b	$a != $b	Checks if **a** is not equal to **b**
a < b	$a < $b	Checks if **a** is less than **b**
a > b	$a > $b	Checks if **a** is greater than **b**

Comparing Numerical Variables

Expression in C	Expression in BASH	Description
a == b	$a -eq $b	Checks if a is equal to b
a != b	$a -ne $b	Checks if a is not equal to b
a < b	$a -lt $b	Checks if a is less than b
a > b	$a -gt $b	Checks if a is greater than b

Comparing Numerical Variables

Expression in C	Expression in BASH	Description
a >= b	$a -ge $b	Checks if a is **greater than** or equal to b
a <= b	$a -le $b	Checks if a is **less than** or equal to b

File Conditions

Expression in BASH	Description
-d $file	Checks if file is a directory
-f $file	Checks if the file exists and is an ordinary file (i.e., not a directory or a special file).
-e $file	Checks if file/directory exists

File Conditions

Expression in BASH	Description
-r $file	Checks if file is **readable**
-w $file	Checks if file is **writable**
-x $file	Checks if file is **executable**
-L $file	Checks if file is **Symbolic Link**

Comparing String Variables Example

Make a Bash Script that take **name** from user and output **welcome message** based on it's name

$ welcome.sh

```
#!/bin/bash
echo "Enter Name……"
read name

if [[ $name == "samir" ]] ; then
    echo "Welcome SPECTRE"
elif [[ $name == "ahmed" ]] ; then
    echo "Welcome admin"
else
    echo "Welcome $name"
fi
```

Comparing Numerical Variables Example

Code a Bash script asks the user to input a **number** and checks whether the number is:

- greater than 10
- equal to 10
- less than 10

then outputs the appropriate message.

$ test10.sh

```
#!/bin/bash

read -p "Enter a number: " num
if [[ $num -gt 10 ]] ; then
    echo "greater than"
elif [[ $num -eq 10 ]] ; then
    echo "equal"
else
    echo "less than"
fi
```

Let’s Practice!

$ exist.sh

Code a Bash Script called exist.sh that take a file as argument check if file exist, then output it a directory or file, if not print file not exist

```
[spectre@ArchLinux samples]$ bash exist.sh welcome.sh
"welcome.sh" exist
"welcome.sh" is a ordinary file
[spectre@ArchLinux samples]$ bash exist.sh me
"me" not exist
```

```
#!/bin/bash
readonly file=$1

if [[ -e $file ]] ; then
    echo "$file exist"

    # conditions on the fly
    [[ -f $file ]] && echo "\"$file\" is ordinary file"
    [[ -d $file ]] && echo "\"$file\" is directory"
else
    echo "$file not exist"
fi
```

Case Statements

Case Statements

more elegant way to handle multiple conditions compared to using multiple if statements.

They are especially useful when you have a variable that could have multiple specific values.

Case Statements Syntax

```
case expression in
    pattern1)
        # some commands to excute
        ;; # break
    pattern2)
        # other commands to excute for pattern2
        ;;
    *)
        # commands to do if any pattern matches
        ;;
esac
```

Let’s make welcome.sh but with case

$ welcomeV2.sh

```
#!/bin/bash

read -p "Enter your name: " name
case $name in
    "samir")
        echo "allowed" ;;
    "ahmed")
        echo "readonly" ;;
    *)
        echo "$name not allowed"
esac
```

Loops

For Loop

A for loop is used when you want to iterate over a list of items (e.g., numbers, strings, files) or just loop for some times.

Syntax:

```
for variable in list ; do
   # Command to execute
done
```

For Loop Examples

```
for i in {1..5} ; do
    echo "number: $i"
done
```

➔ For Loop Example (Another way like C++)

```
n=5
for (( i=1; i <= $n; i++ )) ; do
    echo "Number: $i"
done
```

While Loop

The while loop enables you to execute a set of commands repeatedly until some condition occurs. It is usually used when you need to manipulate the value of a variable repeatedly.

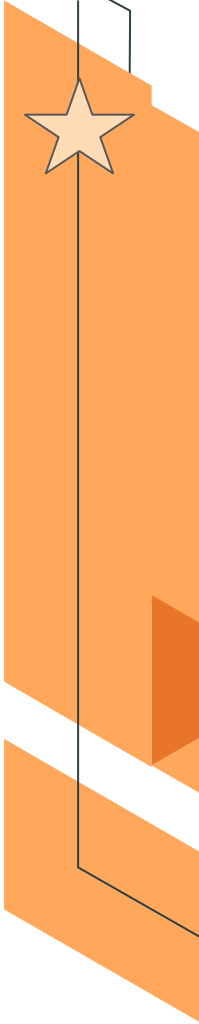
```
while [[ condition ]]
do
   # some stuff to execute
done
```

```
while [[ condition ]] ; do
   # some stuff to execute
done
```

While Loop Example

This script is uses a while loop to count from 1 to 5.

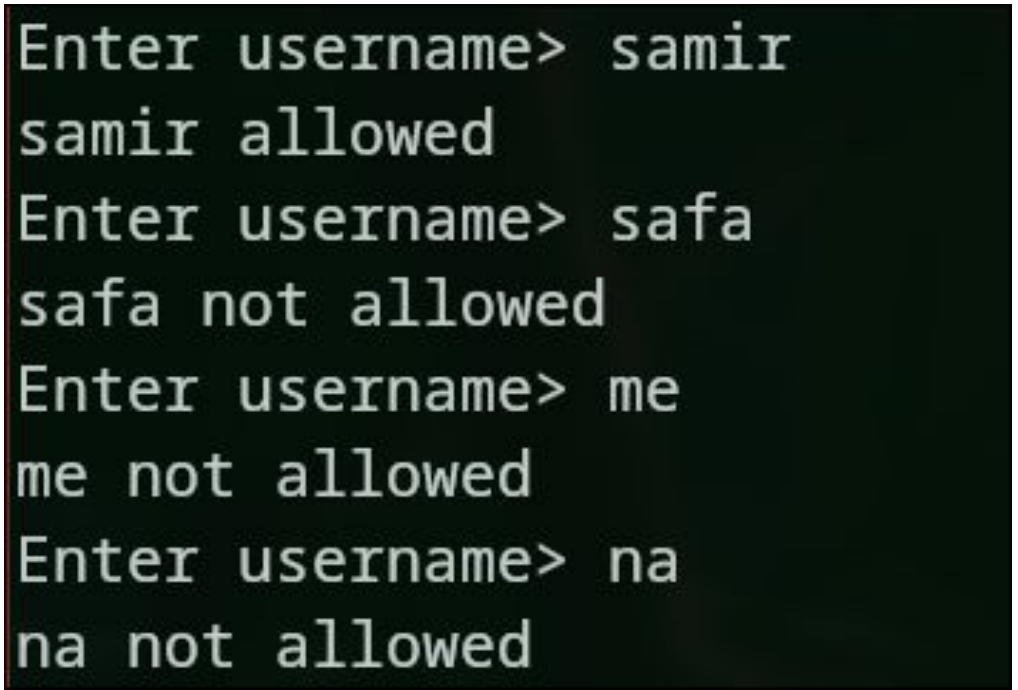
```
#!/bin/bash
count=1

while [[ $count -le 5 ]] ; do
    echo "Counter: $count"
    (( count++ ))
done
```



Hands on

$ login.sh

Code a Bash Script using case and loops, repeat accept users until enter **exit** or enter 3 unallowed users



$ login.sh

```
#!/bin/bash
counter=0
while [[ $counter -lt 3 ]] ; do
    read -p "Enter username> " user
    case $user in
        "samir")
            echo "$user allowed" ;;
        "admin")
            echo "$user allowed" ;;
        "exit")
            echo "exiting......"
            exit ;;
        *)
            echo "$user not allowed"
            counter=$((counter + 1))
    esac
done
```

Functions

Functions in shell scripts allow you encapsulate a block of code that can be reused throughout your script.

Functions can take arguments and return values, making them powerful tools for structuring scripts.

Syntax

```
function NAME # Function Definition
{
    # Doing a things
}
NAME # Function Call
```

Syntax another way

```
NAME() # Function Definition
{
    # Doing a things
}

NAME # Function Call
```

$ hello.sh

```
#!/bin/bash
hello()
{
    for i in seq 1 3 ; do
        echo "Hello !"
    done
}

hello
```

Arguments of Function

You can passing argument by write them after function call
To use the arguments as variables, you can access their values by using $n where n is the order of the argument passed to the function, ex

- $1 first argument
- $2 second argument
- $3 third argument
- $# number of arguments

$ add.sh

```
#!/bin/bash
function add
{
    echo "$1 + $2 = $(($1 + $2))"
}

add 7 3
```


BASH SCRIPT
BASH SCRIPT EVERYWHERE


PathAnalyzer

$ pathAnalyzer.sh pseudocode

```
# main process
    # read input, inial counter for valid paths
    # handle inputs path,
    # handle commands ex (-exit, -ps)
    # -ps echo counter
    # -exit print number of valid paths and exit
# check (function)
    # check if path exist using if
    # check type of path
    # print the meta data using stat cammand
    # increase counter
```

```
#!/bin/bash
counter=0
echo "Welcome to path Analyzer :)"
check() {
    path=$1
    if [[ -e $path ]] ; then
        ((counter++))
        [[ -d $path ]] && echo "Oh! $path is a directory"
        [[ -f $path ]] && echo "Oh! $path is a file"
        stat $path
    else
        echo "$path not exist"
    fi
}
```

```
while [[ $path != "-exit" ]] ; do
    read -p "PATH> " path
    case $path in
        "-help")
            echo "enter path to analysis"
            echo "commands {-help, -clear, -ps, -exit}" ;;
        "-exit")
                echo "========================"
                echo "Valid paths: $counter" "BYE!....."
                echo "========================"
            exit ;;
        "-vp")
            echo "Valid paths: $counter" ;;
        *)
            check $path ;;
    esac
done
```

Power of Scripting !

Thanks!

Do you have any questions?