



ondia

The logo for 'ondia' is centered on a white background. The word is written in a lowercase, rounded sans-serif font. The letters 'o', 'n', and 'd' are a medium purple, while 'i' and 'a' are a darker blue. A light blue and teal graphic element, resembling a stylized 'd' or a corner bracket, is positioned behind the 'd'. The corners of the image are decorated with purple geometric shapes: a triangle in the top-left, a triangle in the top-right, and a larger shape in the bottom-left and bottom-right.



# **Linux Plus for AWS and DevOps**

# Table of Contents



## ▶ Review

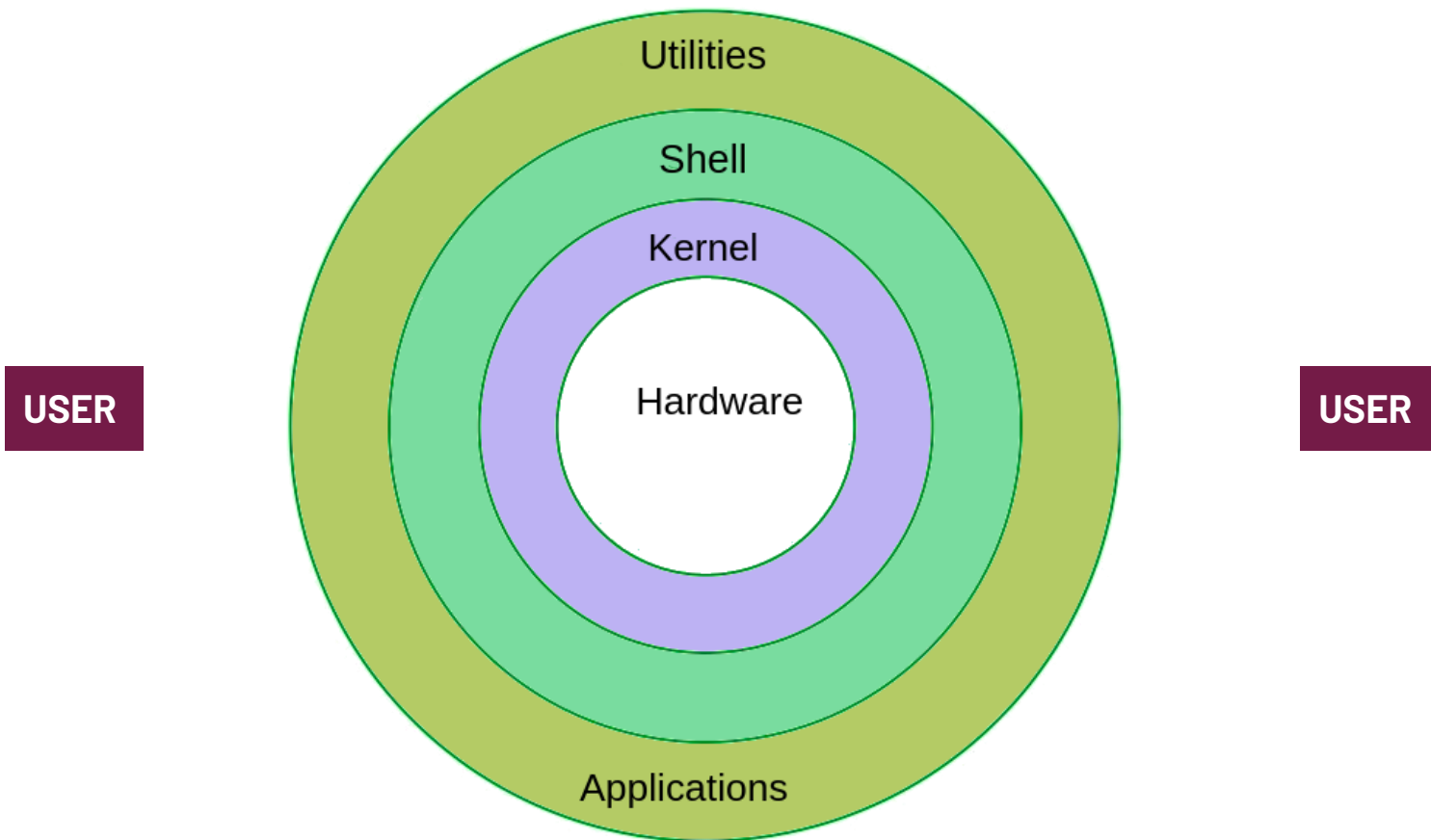
- Shell
- Bash

## ▶ Bash Prompt

## ▶ Shell Scripts



# Components of Linux

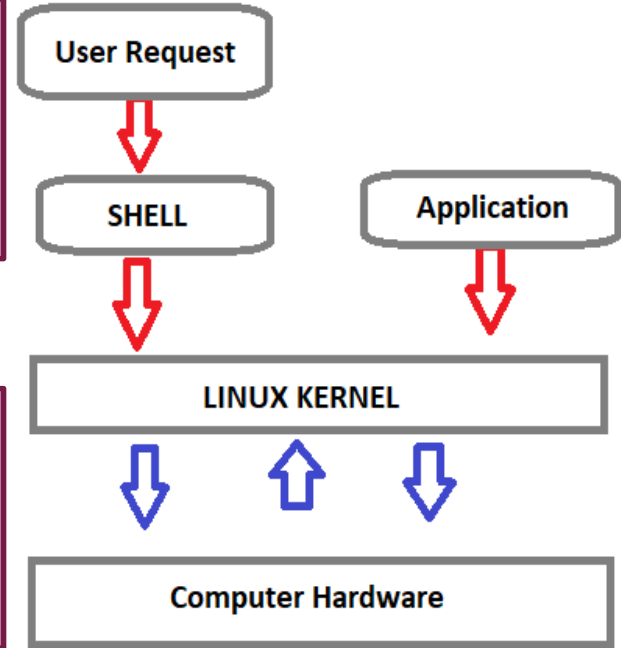


# What is SHELL?

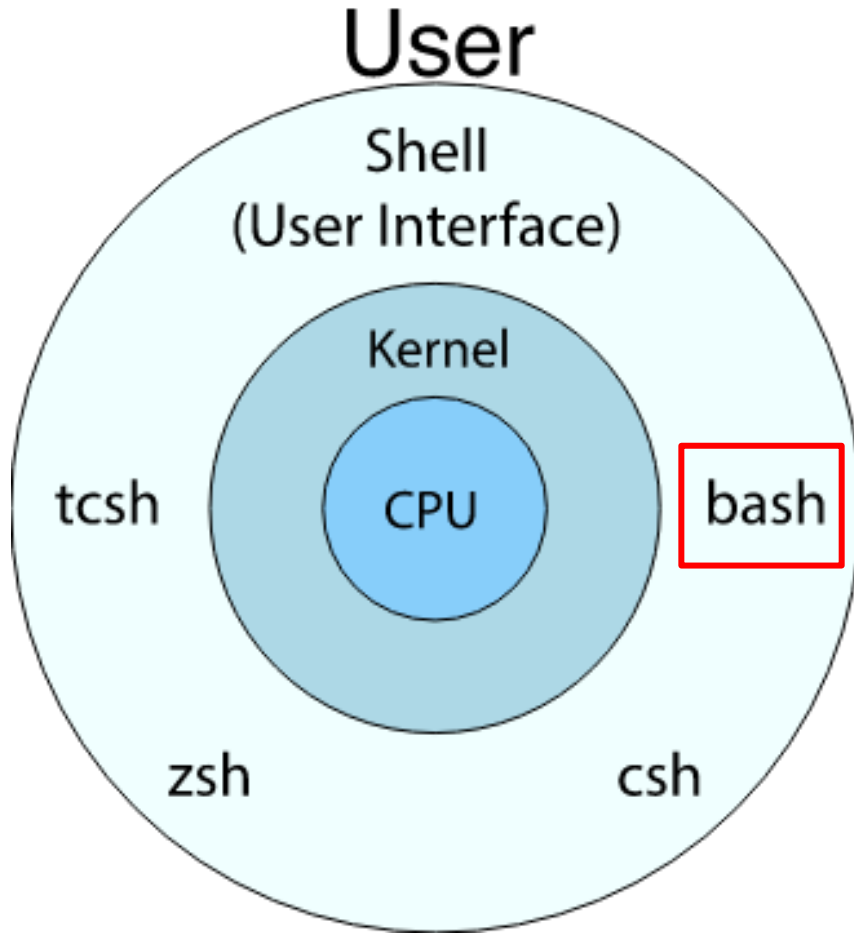


Shell is a **program** that **receives** the user's **commands** and **gives** them to the **operating system** to **process** and displays the output.

The standard Linux shell is both a **command-line interpreter** and a **programming language**.



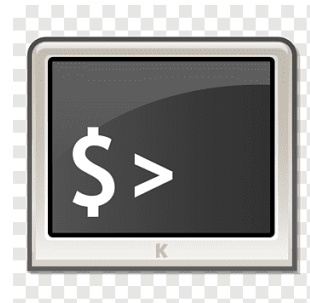
# Shell



SH



**Bourne-Again SHell**





1

# Shell Scripts



## What is Shell Scripting?

Shell Scripting is an open-source computer program designed to be run by the Unix/Linux shell which could be one of the following:

- The Bourne Shell
- The C Shell
- The Korn Shell
- The GNU Bourne-Again Shell





## What is Shell Scripting?

- Typical activities that can be done in a shell, such as file manipulation, program execution, and printing text, can also be done with the shell script.
- Lengthy and repetitive sequences of commands can be combined into a single script that can be stored and executed anytime.

# Shell Scripts



```
aslan@AslanTurker:~/linuxplus$ vim class.sh
aslan@AslanTurker:~/linuxplus$ chmod +x class.sh
aslan@AslanTurker:~/linuxplus$ ./class.sh
Hello world
aslan@AslanTurker:~/linuxplus$
```

```
#!/bin/bash

echo "Hello world"
~
~
~
~
~
~
~
~
~
~
```

Shebang (#!)

#!

# ▶ Shell Scripts



```
aslan@AslanTurker:~/linuxplus$ vim class.sh
aslan@AslanTurker:~/linuxplus$ chmod +x class.sh
aslan@AslanTurker:~/linuxplus$ ./class.sh
Hello world
aslan@AslanTurker:~/linuxplus$
```

```
#!/bin/bash

echo "Hello world"
~
~
~
~
~
~
~
~
~
```



**chmod**

# Shell Scripts

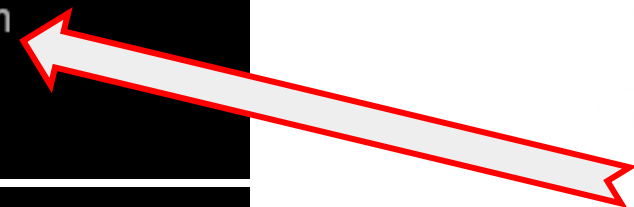


```
aslan@AslanTurker:~/linuxplus$ vim class.sh
aslan@AslanTurker:~/linuxplus$ chmod +x class.sh
aslan@AslanTurker:~/linuxplus$ ./class.sh
Hello world
aslan@AslanTurker:~/linuxplus$
```

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

```
echo "Hello world"
```

```
~
~
~
~
~
~
~
~
~
~
```



"/

# Shell Scripts



```
#!/bin/bash

echo "Hello world"
date
echo "So i can use commands in shell!"
~
~
```

```
aslan@AslanTurker:~/linuxplus$ ./class.sh
Hello world
Thu Jan 30 17:19:05 +03 2025
So i can use commands in shell!
aslan@AslanTurker:~/linuxplus$
```

# Exercise

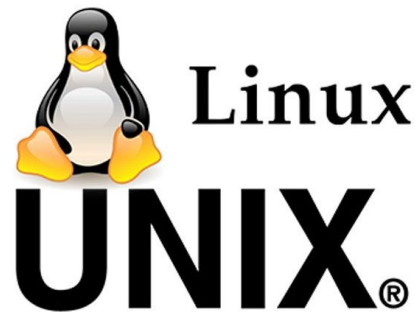


1. Create a script named: **“my-first-script.sh”**

It should print: **“This is my first script.”**

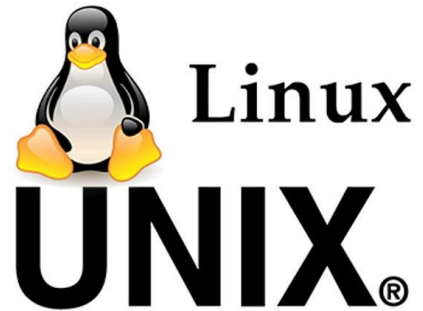
1. Make the script executable.

2. Execute the script.





Create an environment that you don't need to provide “./”  
before your scripts while executing them.



# Variables



- A variable is pointer to the actual data. The shell enables us to create, assign, and delete variables.
- The name of a variable can contain only letters (a to z or A to Z), numbers ( 0 to 9) or the underscore character (\_) and beginning with a letter or underscore character.
- The reason you cannot use other characters such as !, \*, or - is that these characters have a **special meaning for the shell**.

```
$VARIABLE=value
$echo $VARIABLE
value
$
$my_var=my_value
$echo $my_var
my_value
$
$my-var=my-value
my-var=my-value: command not
found
$
$myvar?=my-value
myvar?=my-value: command not
found
```



# Variables

## variable=value

This is one of those areas where formatting is important. Note there is **no space** on either side of the equals ( = ) sign. We also leave off the \$ sign from the beginning of the variable name when setting it.

```
sampledir=/etc  
ls $sampledir
```

```
$ myvar='Hello World'  
$ echo $myvar  
Hello World  
$ newvar="More $myvar"  
$ echo $newvar  
More Hello World  
$ newvar='More $myvar'  
$ echo $newvar  
More $myvar  
$
```

# ▶ Console input



`read [variable-name]`

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

```
echo "Enter your name: "
```

```
read name
```

```
echo Hello $name
```

```
~
```

```
~
```

```
[ec2-user@ip-172-31-36-108 ~]$ ./run.sh
```

```
Enter your name:
```

```
Raymond
```

```
Hello Raymond
```

```
[ec2-user@ip-172-31-36-108 ~]$
```

# Console input



## read

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

```
read -p "Enter Your Name: " username  
echo "Welcome $username!"
```

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

```
read -s -p "Enter Password: " pswd  
echo $pswd
```

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

```
read -sp "Enter Password: " pswd  
echo $pswd
```

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

```
echo "What cars do you like?"
```

```
read car1 car2 car3
```

```
echo Your first car was: $car1  
echo Your second car was: $car2  
echo Your third car was: $car3
```

# Command Line Arguments

Built-in variables



**\$0** - The name of the Bash script.

**\$1 - \$9** - The first 9 arguments to the Bash script.

**\$#** - How many arguments were passed to the Bash script.

**\$@** - All the arguments supplied to the Bash 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 it is referred to.

**\$LINENO** - Returns the current line number in the Bash script.



# ▶ Command Line Arguments



The diagram illustrates the mapping of command line arguments to shell variables. It consists of three rows. The top row contains the command `./script.sh` followed by ten arguments: `ARG1`, `ARG2`, `ARG3`, `ARG4`, `ARG5`, `ARG6`, `ARG7`, `ARG8`, `ARG9`, and `ARG10`. The middle row contains ten red arrows pointing downwards from each argument to its corresponding variable. The bottom row contains the shell variables: `$0`, `$1`, `$2`, `$3`, `$4`, `$5`, `$6`, `$7`, `$8`, `$9`, and `${10}`.

Command	ARG1	ARG2	ARG3	ARG4	ARG5	ARG6	ARG7	ARG8	ARG9	ARG10
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
<code>\$0</code>	<code>\$1</code>	<code>\$2</code>	<code>\$3</code>	<code>\$4</code>	<code>\$5</code>	<code>\$6</code>	<code>\$7</code>	<code>\$8</code>	<code>\$9</code>	<code>\${10}</code>

# Simple Arithmetic



**expr** command **print** the value of expression to **standard output**.

```
expr item1 operator item2
```

**let** is a builtin function of Bash that helps us to do simple arithmetic. It is similar to **expr** except instead of printing the answer **it saves the result to a variable**.

```
let <arithmetic expression>
```

We can also evaluate arithmetic expression with double parentheses.

```
$((arithmetic expression))
```

# Arithmetic Expressions



```
expr item1 operator item2
```

```
#!/bin/bash
first_number=8
second_number=2

echo "SUM="`expr $first_number + $second_number`
echo "SUB="`expr $first_number - $second_number`
echo "MUL="`expr $first_number \* $second_number`
echo "DIV="`expr $first_number / $second_number`
```

```
$ chmod +x cal.sh
$ ./cal.sh
SUM=10
SUB=6
MUL=16
DIV=4
```

# Arithmetic Expressions



**let [expression]**

```
#!/bin/bash

number1=8
number2=2

let total=number1+number2
let diff=number1-number2
let mult=number1*number2
let div=number1/number2

echo "Total = $total"
echo "Difference = $diff"
echo "Multiplication = $mult"
echo "Division = $div"
```

```
$ ./run.sh
Total = 10
Difference = 6
Multiplication = 16
Division = 4
```



# “num++” “++num” “num--” “--num”

number++ (Post Increment) →>> First **assigned** to new number then **increase** number

++number (Pre Increment) →>> First **increase** number then **assigned** to new number

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

```
number=10
```

```
let new_number=number++
```

```
echo "Number = $number"
```

```
echo "New number = $new_number"
```

```
number=10
```

```
let new_number=--number
```

```
echo "Number = $number"
```

```
echo "New number = $new_number"
```

```
~
```

```
[ec2-user@ip-172-31-91-206 ~]$ ./run.sh
```

```
Number = 11
```

```
New number = 10
```

```
Number = 9
```

```
New number = 9
```

```
[ec2-user@ip-172-31-91-206 ~]$
```

# Arithmetic Expressions



`$ ( (Expression) )`

`((Expression))`

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

```
number1=8
```

```
number2=2
```

```
echo "Total = $((number1+number2))"
```

```
((total=number1+number2))
```

```
echo "Total = $total"
```

```
█
```

```
~
```

```
[ec2-user@ip-172-31-91-206 ~]$ ./run.sh
```

```
Total = 10
```

```
Total = 10
```

```
[ec2-user@ip-172-31-91-206 ~]$ █
```

# Exercise



1. Ask user to enter two numbers to variables **num1** and **num2**.
2. Calculate the total of 2 numbers.
3. Print the **total** number and increase it by 1.
4. Print the new value of the **total** number.
5. Subtract **num1** from the **total** number and print result.
6. Change the **num1** and **num2** variables to be passed from the **Command line arguments** instead of receiving them from the user



1. Create a script named **calculate.sh**:

Create a variable named **base\_value** with default value of **5**

Request another number from user and assign it to **user\_input** variable

Add **user\_value** to the **base\_value** and assign it to **total** variable

Print **total** to the screen with the message “**Total value is:** ”

1. Make the script executable.
2. Execute the script.

# THANKS!

**Any questions?**

