SHELL SCRIPTING

INTRODUCTION TO SHELL SCRIPTING

Definition:
Shell scripting is writing a **series of commands in a file** (script) that the Linux shell can execute automatically.

Why Shell Scripting is Needed (Purpose)

- Automates **repetitive tasks**
- Simplifies **system administration**
- Speeds up **production support tasks**
- Reduces **human error**

Key Points (Core Concepts)

- A **shell script** is a text file containing commands
- Executed using **bash** or other shells (sh, zsh)
- Supports **variables, loops, conditions, functions**
- Must have **execute permission**

Basic Structure of a Shell Script

```
#!/bin/bash       # Shebang line
# Comment
echo "Hello World"
```

- #!/bin/bash → tells system which shell to use
- echo → prints output

NOTE: Even without a shebang, scripts may run because Bash handles the execution fallback. However, the OS kernel requires a shebang to know which interpreter to use, so it is mandatory for portability and production systems.

Making a Script Executable

```
chmod +x script.sh
./script.sh
```

Variables

```
name="Cognizant"
echo "Welcome $name"
```

Conditional Statements

```
if [ $age -ge 18 ]; then
  echo "Adult"
else
  echo "Minor"
fi
```

Loops

For loop example:

```
for i in 1 2 3; do
  echo "Number $i"
done
```

While loop example:

```
count=1
```

```
while [ $count -le 5 ]; do
  echo $count
  count=$((count+1))
done
```

Functions

```
greet() {
  echo "Hello $1"
}
greet "Shaik"
```

Real-Life Example

Automate backup of logs:

```
#!/bin/bash
cp /var/log/app.log /backup/app_$(date +%F).log
echo "Backup completed"
```

Technical Example (Production Support)

- Check if a service is running:

```
#!/bin/bash
if systemctl status apache2 | grep "running"; then
  echo "Apache is running"
else
  echo "Apache is stopped"
fi
```

Interview Explanation (How to Say It)

"Shell scripting is writing command sequences in a file to automate tasks, monitor systems, and simplify administration in Linux."

Common Interview Questions

- What is a shell script?
- Difference between sh and bash?
- How to pass arguments to a script?
- How to schedule scripts?

Common Mistakes Freshers Make

- Forgetting shebang line
- Not giving execute permission
- Hardcoding values instead of variables
- Ignoring error handling

One-Line Summary

Shell scripting automates Linux tasks using sequences of commands with variables, loops, and conditions.

BASIC SELL SCRIPTING CONCEPTS

Definition:

Basic shell scripting concepts are the foundational ideas needed to **write, understand, and execute scripts** in Linux for automation and system tasks.

Why Learn Shell Scripting Concepts (Purpose)

- Automates repetitive tasks
- Speeds up production support
- Simplifies file, process, and system management
- Essential for DevOps and Linux-based roles

Core Concepts

3.1 Shebang (#!)

- First line in a script
- Specifies the shell to execute the script

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

3.2 Comments

- Lines starting with # are ignored by the shell

```
# This is a comment
```

3.3 Variables

- Store data to reuse in scripts
- No spaces around =

```
name="Shaik"
echo "Hello $name"
```

3.4 Input & Output

- echo → print to screen
- read → get user input

```
read -p "Enter your name: " name
echo "Hello $name"
```

3.5 Operators

- Arithmetic: + - * / %
- Comparison: -eq -ne -gt -lt -ge -le
- Logical: && || !

3.6 Conditional Statements

```
if [ $age -ge 18 ]; then
  echo "Adult"
else
  echo "Minor"
fi
```

3.7 Loops

- **For Loop:**

```
for i in 1 2 3; do
  echo $i
done
```

- **While Loop:**

```
count=1
while [ $count -le 5 ]; do
  echo $count
  count=$((count+1))
done
```

3.8 Functions

- Encapsulate reusable logic

```
greet() {
  echo "Hello $1"
}
greet "Shaik"
```

3.9 Command-line Arguments

- $0 → script name
- $1, $2... → arguments

```
echo "Script name: $0"
echo "First argument: $1"
```

Exit Status

- $? stores the **exit status** of last command
- 0 → success, non-zero → failure

Real-Life Example

Backup script with input:

```
#!/bin/bash
read -p "Enter filename: " file
cp /var/log/$file /backup/$file_$(date +%F)
echo "Backup completed"
```

Interview Explanation (How to Say It)

"Basic shell scripting concepts include variables, loops, conditions, functions, input/output, and arguments to automate tasks efficiently in Linux."

Common Interview Questions

- What is a shell script?
- Difference between $0 and $1?
- How to use variables in shell scripts?
- How to loop in shell scripting?

Common Mistakes Freshers Make

- Forgetting shebang
- Misusing spaces in variable assignments
- Ignoring exit status
- Not testing scripts before running

One-Line Summary

Shell scripting concepts provide the foundation to automate tasks using commands, variables, loops, and conditions in Linux.

CONTROL STRUCTURES IN SHELL SCRIPTING

Definition:

Control structures are statements that **control the flow of execution** in shell scripts based on conditions or repetitions.

Why Control Structures Are Needed (Purpose)

- To make scripts **dynamic and flexible**
- To **execute tasks conditionally**
- To **repeat tasks efficiently**
- Essential for **automation and production support**

Types of Control Structures

3.1 Conditional Statements

- **If-Else**: Execute commands based on a condition

```
if [ $age -ge 18 ]; then
  echo "Adult"
else
  echo "Minor"
fi
```

- **If-Elif-Else**: Multiple conditions

```
if [ $score -ge 90 ]; then
  echo "A grade"
elif [ $score -ge 75 ]; then
  echo "B grade"
else
  echo "C grade"
fi
```

3.2 Case Statement

- Used for multiple choices

```
read -p "Enter a fruit: " fruit
case $fruit in
  Apple) echo "Red";;
  Banana) echo "Yellow";;
  *) echo "Unknown";;
esac
```

3.3 Loops

- **For Loop:** Iterates over a list

```
for i in 1 2 3 4 5; do
  echo "Number $i"
done
```

- **While Loop:** Executes while condition is true

```
count=1
while [ $count -le 5 ]; do
  echo $count
  count=$((count+1))
done
```

- **Until Loop:** Executes until condition becomes true

```
count=1
until [ $count -gt 5 ]; do
  echo $count
  count=$((count+1))
done
```

Break & Continue

- **break** → exit loop early
- **continue** → skip current iteration

```
for i in 1 2 3 4 5; do
  if [ $i -eq 3 ]; then
    continue
  fi
  echo $i
done
```

Real-Life Example

- Checking if a server service is running and restarting it:

```
if systemctl status apache2 | grep "inactive"; then
  systemctl start apache2
  echo "Service started"
else
  echo "Service running"
fi
```

Technical Example

- Loop through log files and count errors:

```
for file in /var/log/*.log; do
  echo "$file : $(grep -c ERROR $file)"
done
```

Interview Explanation (How to Say It)

“Control structures in shell scripting like if-else, case, and loops control the flow of execution to perform conditional or repetitive tasks.”

Common Interview Questions

- Difference between while and until loop?
- When to use case over if-else?
- How to exit a loop early?
- Difference between break and continue?

Common Mistakes Freshers Make

- Forgetting fi for if
- Wrong comparison operators (= vs -eq)
- Infinite loops due to wrong conditions
- Misusing break and continue

One-Line Summary

Control structures make shell scripts dynamic by handling conditions and repeating tasks efficiently.

COMMAND-LINE ARGUMENTS IN SHELL SCRIPTING

Definition:

Command-line arguments are **inputs passed to a shell script when it is executed,** allowing scripts to be **dynamic and reusable.**

Why Command-Line Arguments Are Needed (Purpose)

- To **pass data dynamically** without editing the script
- To make scripts **flexible and reusable**
- Essential for **automation and production support tasks**

Positional Parameters

Parameter	Meaning
$0	Script name
$1, $2…	First, second, etc., arguments
$#	Number of arguments passed

Parameter	Meaning
$@	All arguments as separate words
$*	All arguments as a single string
$?	Exit status of last command
$$	Process ID of current script

Basic Example

```
#!/bin/bash
echo "Script name: $0"
echo "First argument: $1"
echo "Second argument: $2"
echo "Total arguments: $#"
```

Run:

```
./script.sh arg1 arg2
```

Output:

```
Script name: ./script.sh
First argument: arg1
Second argument: arg2
Total arguments: 2
```

Loop Through Arguments

```
#!/bin/bash
echo "All arguments:"
for arg in "$@"; do
  echo $arg
done
```

Real-Life Example

Backup script with dynamic filename:

```
#!/bin/bash
cp /var/log/$1 /backup/$1_$(date +%F)
echo "Backup of $1 completed"
```

Run:

```
./backup.sh app.log
```

Technical Example (Production Support)

Check service status using argument:

```
#!/bin/bash
service=$1
if systemctl status $service | grep "running"; then
  echo "$service is running"
else
  echo "$service is stopped"
fi
```

Interview Explanation (How to Say It)

“Command-line arguments allow passing dynamic inputs to a shell script, making it reusable and flexible for different scenarios.”

Common Interview Questions

- What is $0 vs $1?
- Difference between $* and $@?
- How to count arguments passed?
- How to use arguments in loops?

Common Mistakes Freshers Make

- Confusing $* and $@
- Not checking if arguments are provided
- Hardcoding values instead of using arguments
- Forgetting to quote $@ in loops

One-Line Summary
Command-line arguments make shell scripts dynamic and reusable by allowing input at runtime.

FUNCTIONS IN SHELL SCRIPTING

Definition:
Functions are **reusable blocks of code** in a shell script that perform a specific task and can be called multiple times.

Why Functions Are Needed (Purpose)

- Avoid **code repetition**
- Make scripts **organized and modular**
- Simplify **complex scripts**
- Essential for **production support and automation**

Syntax

```
function_name() {
  # commands
  return value  # optional
}
```

- Call a function by its name:

```
function_name
```

Example 1 – Simple Function

```
#!/bin/bash
greet() {
  echo "Hello, $1"
}

greet "Shaik"
greet "Cognizant"
Output:
Hello, Shaik
Hello, Cognizant
```

Example 2 – Function with Return Value

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

result=$(add 5 10)
echo "Sum is $result"
```

Example 3 – Using Exit Status

```
#!/bin/bash
check_file() {
  if [ -f "$1" ]; then
    return 0  # success
  else
    return 1  # failure
  fi
}

check_file "/etc/passwd"
if [ $? -eq 0 ]; then
  echo "File exists"
else
  echo "File does not exist"
fi
```

Real-Life Example

- Reusable function to **backup logs**:

```
backup() {
  cp /var/log/$1 /backup/$1_$(date +%F)
  echo "$1 backed up"
}

backup app.log
backup sys.log
```

Technical Example (Production Support)

- Function to **check and restart service**:

```
check_service() {
  svc=$1
  if systemctl status $svc | grep "running"; then
    echo "$svc is running"
  else
    systemctl start $svc
    echo "$svc started"
  fi
}

check_service apache2
check_service mysql
```

Interview Explanation (How to Say It)

“Functions in shell scripts allow grouping of commands to perform a specific task, making scripts modular, reusable, and easier to maintain.”

Common Interview Questions

- How to pass arguments to a function?
- Difference between return value and exit status?
- Can a function call another function?
- Why use functions in shell scripts?

Common Mistakes Freshers Make

- Forgetting to call the function

- Confusing return value with echo output
- Using global variables incorrectly
- Overcomplicating simple tasks without functions

One-Line Summary
Functions make shell scripts modular, reusable, and easier to maintain by grouping commands for specific tasks.

TEXT PROCESSING IN LINUX

Definition:
Text processing in Linux is the use of **commands and utilities to manipulate, filter, and analyze text files** efficiently.

Why Text Processing Is Needed (Purpose)

- To **search and filter logs**
- To **extract useful information** from files
- To **analyze large datasets** quickly
- Essential for **production support and automation**

Key Commands

Command	**Purpose**	**Example**
cat	View full file	cat file.txt
less / more	View large files	less file.txt
head	First n lines	head -n 5 file.txt
tail	Last n lines	tail -n 10 file.txt
grep	Search text	grep "ERROR" app.log
cut	Extract columns	cut -d',' -f2 file.csv
sort	Sort lines	sort names.txt
uniq	Remove duplicates	`sort names.txt
wc	Count lines/words/characters	wc -l file.txt
awk	Pattern scanning & processing	awk '{print $2}' file.txt
sed	Stream editor for substitution	sed 's/old/new/g' file.txt

Real-Life Example
Count error occurrences in log:

```
grep "ERROR" /var/log/app.log | wc -l
```

Technical Example (Production Support)
Extract unique IP addresses from a log file:

```
cat access.log | awk '{print $1}' | sort | uniq
```

Combining Commands (Pipes & Filters)

```
tail -f /var/log/app.log | grep "CRITICAL" | wc -l
```

- Monitor logs in real-time
- Filter critical errors
- Count them

Interview Explanation (How to Say It)

“Text processing in Linux uses commands like grep, awk, sed, cut, sort, and wc to filter, extract, and analyze text efficiently.”

Common Interview Questions

- Difference between grep and awk?
- How to count unique occurrences?
- How to replace text in a file using command line?
- How to extract a specific column from a CSV?

Common Mistakes Freshers Make

- Not using sort before uniq
- Forgetting quotes around patterns in grep
- Using cat unnecessarily (UUOC – Useless Use of Cat)
- Misunderstanding field separators in cut or awk

One-Line Summary

Text processing allows filtering, extracting, and analyzing text efficiently using Linux commands and pipelines.

ERROR HANDLING & DEBUGGING IN SHELL SCRIPTING

Definition:

Error handling is the process of **detecting and responding to errors** in a shell script, while **debugging** is the process of **finding and fixing errors** in scripts.

Why Error Handling & Debugging Are Needed (Purpose)

- To **prevent script failures**
- To **identify the root cause of issues**
- To ensure **reliable automation**
- Essential for **production support and system maintenance**

Key Concepts

3.1 Exit Status

- Every command returns an **exit status**
- 0 → Success
- Non-zero → Failure

```
ls /tmp
echo $?  # Shows exit status of last command
```

3.2 Conditional Error Handling

```
#!/bin/bash
mkdir /tmp/testdir
if [ $? -eq 0 ]; then
  echo "Directory created successfully"
else
  echo "Failed to create directory"
fi
```

3.3 set -e & set -u

- set -e → Exit script on first error
- set -u → Treat unset variables as error

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

```
set -e
cp file1.txt /tmp
echo "This will not run if cp fails"
```

3.4 Using trap

- Execute commands on **script exit or error**

```
#!/bin/bash
trap 'echo "An error occurred. Exiting..."' ERR
cp file1.txt /tmp
```

Debugging Options

Option	Purpose
-x	Print commands as they are executed (bash -x script.sh)
-v	Print shell input lines as read (bash -v script.sh)

Example:

```
#!/bin/bash
set -x  # Enable debugging
echo "Debugging example"
ls /nonexistent
```

Real-Life Example

Check if a service is running and restart if failed:

```
#!/bin/bash
trap 'echo "Error occurred while managing service"' ERR
service apache2 status || systemctl start apache2
```

Technical Example (Production Support)

Backup script with error handling:

```
#!/bin/bash
set -e
trap 'echo "Backup failed at $(date)"' ERR
cp /var/log/app.log /backup/app_$(date +%F).log
echo "Backup completed successfully"
```

Interview Explanation (How to Say It)

"Error handling in shell scripts involves checking exit statuses, using set options, and traps, while debugging uses flags like -x or -v to trace and fix issues."

Common Interview Questions

- How do you check if a command failed in a script?
- What is the purpose of trap in shell scripting?
- Difference between set -e and set -u?
- How to debug a shell script?

Common Mistakes Freshers Make

- Ignoring exit statuses
- Not using set -e for critical scripts
- Not testing scripts before production
- Confusing debugging flags -x and -v

One-Line Summary

Error handling and debugging ensure shell scripts run reliably by detecting, handling, and tracing errors effectively.