Shell Scripting (Bash)

From Basics to Practical Automation

Roadmap

In this chapter we will cover:

1. What is scripting?
2. How Bash executes programs
3. Writing and running scripts
4. Variables and arguments
5. Input and output
6. Conditionals (if, elif, case)

7. Loops (for, while, until)
8. Arrays and strings
9. Functions
10. Exit status and debugging
11. Mini practical exercises

Section 1

What is Shell Scripting?

In This Section

- Script vs program
- Interpreter model
- Why automation matters
- Real-world use cases

What is a Script?

A script is a file containing commands executed sequentially.

Analogy

- A cooking recipe
- A checklist for packing luggage
- Morning routine instructions

The shell follows instructions step by step.

Interpreter Model

Bash reads one line
Executes it
Moves to next line

Unlike C:

- C → compiled once, run many times
- Bash → interpreted every run

Downside: Interpreted programs are slower — each line is translated at runtime.

Where is Bash Used?

- Any user required automations
- Server automation
- Backup scripts
- Many more

Summary — Section 1

- Scripts automate command sequences
- Bash is an interpreter
- Used heavily in system administration

Section 2

Creating and Running Scripts

In This Section

- Script structure
- Shebang
- Permissions
- Execution flow
- Common scripting errors

Basic Script Structure

```
echo "Hello World"
```

- Save it as `hello.sh`

```
bash hello.sh
```

Adding the Shebang

```
#!/bin/bash
echo "Hello World"
```

`#!/bin/bash` tells the system which interpreter to use.

Required for `./hello.sh` style execution.

Not needed when calling `bash hello.sh` directly.

Making It Executable

```
chmod +x hello.sh
./hello.sh
```

Why `./`?

Because current directory is not automatically searched.

How Shell Finds Commands

Shell checks:

1. Built-ins
2. PATH variable directories

Check PATH:

```
echo $PATH
```

Common Scripting Errors

Spaces in assignment:

```
FULL_NAME=Frank Zappa   # ❌ Error: tries to run "Zappa"
FULL_NAME="Frank Zappa" # ✅ Correct
```

Permission error:

```
wc -l /etc/shadow   # ❌ Permission denied
```

Bash runs the script even if errors exist — all errors appear in output alongside results.

Class Exercise

1. Create script `info.sh`
2. Print:
    - Current user
    - Current directory
3. Use commands:
    - `whoami`
    - `pwd`

Reflect

What happens if you remove execute permission?

Summary — Section 2

- Shebang is needed for direct `./script.sh` execution
- Execute permission required for `./` style execution
- PATH controls command lookup
- Bash reports all errors at runtime — no compile-time check

Section 3

Variables and Arguments

In This Section

- Variable assignment
- Single vs double quotes
- `declare` keyword
- Positional parameters
- Special variables

Variables

```
name="Alice"
echo $name
```

⚠️ No spaces around `=`

```
name = "Alice"    # ❌ Error: tries to run command "name"
name="Alice"      # ✅ Correct
```

Single vs Double Quotes

```
name="Alice"
echo "Hello $name"    # → Hello Alice  (variable expanded)
echo 'Hello $name'    # → Hello $name  (no expansion)
```

- **Double quotes** `" "` — variables and special characters expand
- **Single quotes** `' '` — everything treated as a literal string

The `declare` Keyword

Variables can be declared with attributes:

```
declare -r PI=3.14          # read-only (constant)
declare -i COUNT=0          # integer
declare -x PATH_VAR=/tmp    # exported to child processes
declare -a NAMES            # array
```

You don't need to declare — but it helps write safer scripts.

Special Variables

- `$1` → first argument
- `$2` → second argument
- `$#` → number of arguments
- `$0` → script name
- `$?` → last command exit status
- `$@` → all arguments as separate strings

Example Script

```
#!/bin/bash
echo "Script name: $0"
echo "Arguments count: $#"
echo "First argument: $1"
```

Class Exercise

Write script that:

- Takes two numbers as arguments
- Prints their sum

Hint:

```
$((a + b))
```

Think About This

Why does Bash not need type declarations?

Summary — Section 3

- Variables store data; no spaces around `=`
- Double quotes expand variables; single quotes do not
- `declare` adds optional type/attribute safety
- Arguments passed via `$1`, `$2`; `$@` for all
- Arithmetic via `$(( ))`

Section 4

Input and Output

Reading Input

```
read name
echo "Hello $name"
```

Better — with a prompt:

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

Redirecting Output

```
echo "Hello" > file.txt    # overwrite
echo "More"  >> file.txt   # append
```

`>` overwrites the file (creates if not exists)

`>>` appends to the file (creates if not exists)

Input Redirection

```
wc -l < file.txt
```

Feeds `file.txt` as input to `wc -l` instead of keyboard.

Class Exercise

Create script that:

- Takes a filename as argument
- Prints number of lines in the file

Hint:

```
Search for wc?
```

Reflect

What is the difference between `>` and `>>`?

Summary — Section 4

- `read -p "prompt" var` takes interactive input
- `>` overwrites; `>>` appends
- `<` redirects a file as input
- Scripts can process files passed as arguments

Section 5

Conditional Statements

If Statement

```
if [ $1 -gt 10 ]
then
  echo "Greater than 10"
fi
```

If-Else

```
if [ $1 -eq 0 ]
then
  echo "Zero"
else
  echo "Non-zero"
fi
```

elif — N-way Selection

```
if [ $score -ge 90 ]
then
  echo "Grade: A"
elif [ $score -ge 80 ]
then
  echo "Grade: B"
elif [ $score -ge 70 ]
then
  echo "Grade: C"
else
  echo "Grade: F"
fi
```

`elif` = else + if chained together.

Logical Operators

Inside `[ ]`:

- `-a` → AND (older style)
- `-o` → OR (older style)

Preferred modern style using `[[ ]]`:

```
if [[ $x -gt 0 && $x -lt 10 ]]
then
  echo "Between 1 and 9"
fi
```

File Conditions

```
if [ -f file.txt ]
then
  echo "Regular file exists"
fi

if [ -d /tmp ]
then
  echo "Directory exists"
fi
```

Common operators: `-f` (file), `-d` (directory), `-e` (exists), `-r` (readable), `-x` (executable)

The `case` Statement

An alternative to long `elif` chains:

```
case $color in
  red)   echo "Stop" ;;
  green) echo "Go" ;;
  yellow) echo "Slow down" ;;
  *)     echo "Unknown color" ;;
esac
```

`*)` is the default (like `else`). Each option ends with `;;`.

When to Use `case` vs `elif`

Use `case` when:

- Comparing one variable to many fixed values
- Options are discrete strings or patterns

Use `elif` when:

- Conditions involve ranges or comparisons (`-gt`, `-lt`)
- Multiple variables are involved

Class Exercise

Write script that:

- Checks if a file given as argument exists
- Prints "File found" or "File not found"

Think Deeper

- Why must there be spaces inside `[ ]`?
- Difference between `[` nad `[[` style?

Number Comparators ([ ] or [[ ]])

Operator	Meaning
`-eq`	equal
`-ne`	not equal
`-gt`	greater than
`-ge`	greater than or equal
`-lt`	less than
`-le`	less than or equal

String Comparators

Operator	Meaning
`=`	equal
`!=`	not equal
`<`	less than (ASCII order)
`>`	greater than (ASCII order)
`-z`	string is empty
`-n`	string is not empty

Arithmetic Style (inside (( )))

Operator	Meaning
`==`	equal
`!=`	not equal
`>`	greater than
`<`	less than
`>=`	greater or equal
`<=`	less or equal

File Comparators

Operator	Meaning
`-e`	exists
`-f`	regular file
`-d`	directory
`-s`	not empty
`-r`	readable

File Comparators

Operator	Meaning
`-w`	writable
`-x`	executable
`-nt`	newer than
`-ot`	older than
`-ef`	same file

Logical Operators

Operator	Meaning
`&&` / `-a`	AND
`||` / `-o`	OR
`!`	NOT

Summary — Section 5

- `if/elif/else` handles multi-way decisions
- `case` is cleaner for matching one variable against many values
- File conditions: `-f`, `-d`, `-e`, `-r`, `-x`
- Use `[[ ]]` with `&&` / `||` for compound conditions

Section 6

Loops

For Loop — Iterator Style

```
for i in {1..5}
do
  echo $i
done
```

For Loop — Counter Style (C-style)

```
for (( i=0; i<5; i++ ))
do
  echo $i
done
```

Useful when you need the index value (e.g., accessing array elements).

While Loop

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

Until Loop

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

`until` loops while the condition is **false** — opposite of `while`.

Loop Over Files

```
for file in *.txt
do
  echo $file
done
```

Class Exercise

Write script that:

- Prints even numbers from 1 to 20

Reflect

When is `while` better than `for`?

Summary — Section 6

- `for in` for known lists or ranges
- `for (( ))` for counter-controlled loops (C-style)
- `while` for condition-based repetition (pre-test)
- `until` loops while condition is false

Section 7

Arrays and Strings

Arrays — Zero Indexed

```
arr=("one" "two" "three")
echo ${arr[0]}   # → one
echo ${arr[1]}   # → two
```

⚠️ Arrays are **zero-indexed** — first element is at index 0.

Useful Array Notations

```
arr=("a" "b" "c")
echo ${#arr[@]}    # → 3 (number of elements)
echo ${arr[@]}     # → a b c (all elements)
echo "${arr[@]}"   # → "a" "b" "c" (each individually quoted)
echo "${arr[*]}"   # → "a b c" (all as one string)
```

Looping Arrays

```
for item in "${arr[@]}"
do
  echo $item
done
```

Adding to an Array

```
arr=("one" "two")
arr+=("three")
echo ${arr[@]}   # → one two three
```

String Length and Substrings

```
name="Alice"
echo ${#name}          # → 5 (length)
echo ${name:0:3}       # → Ali (substring: start 0, length 3)
echo ${name:2:2}       # → ic
```

Indexing starts at **0** for substrings.

String Replacement

```
str="Hello World"
echo ${str/World/Bash}     # → Hello Bash (replace first)
echo ${str//l/L}           # → HeLLo WorLd (replace all)
```

Class Exercise

Create array of 5 numbers.
Print their sum using a loop.

Summary — Section 7

- Arrays are zero-indexed
- `${arr[@]}` expands all elements; `${#arr[@]}` gives count
- `arr+=("val")` appends to an array
- Substrings: `${string:start:length}` (zero-indexed)
- String replacement: `${str/old/new}`

Section 8

Functions

Function Definition

```
square() {
  echo $(($1 * $1))
}
```

Call:

```
square 5    # → 25
```

Local Variables

```
greet() {
  local message="Hello $1"
  echo $message
}

greet "Alice"
echo $message   # → (empty — local to function)
```

Use `local` to avoid accidentally changing variables outside the function.

Function Return Values

Functions communicate results in two ways:

```
# 1. Echo output (capture with $())
double() {
  echo $(($1 * 2))
}
result=$(double 5)
echo $result   # → 10

# 2. Return exit code (0=success, non-zero=error)
check_positive() {
  if [ $1 -gt 0 ]; then return 0; else return 1; fi
}
check_positive 5
echo $?   # → 0 (success)
```

Why Functions?

- Code reuse
- Cleaner structure
- Easier debugging
- `local` variables prevent side effects

Class Exercise

Create function that:

- Accepts 3 numbers
- Prints the largest number

Summary — Section 8

- Functions accept arguments via `$1`, `$2`
- Use `local` for variables that should not leak outside
- Return data via `echo` (captured with `$()`) or via exit code (`return`)
- Functions improve modularity and reduce duplication

Section 9

Exit Status and Debugging

Exit Codes

Every command returns:

- `0` → success
- Non-zero → error

Check:

```
echo $?
```

Using Exit Status — Best Practice

```
# Preferred: check command directly
if grep -q "error" log.txt
then
  echo "Errors found"
fi

# Fragile: $? can be overwritten by another command
grep -q "error" log.txt
echo $?   # ← $? is now from echo, not grep
```

⚠️ Check `$?` immediately after the command, or use the command directly in `if`.

Setting Exit Codes in Scripts

```
#!/bin/bash
if [ ! -f "$1" ]
then
  echo "File not found"
  exit 1    # non-zero = failure
fi
echo "Processing $1"
exit 0      # zero = success
```

Debugging Mode

```
bash -x script.sh
```

Shows each line before it executes — useful for tracing logic errors.

Think Deeper

Why is exit code 0 considered success?

Summary — Section 9

- Commands return exit code: `0` = success, non-zero = error
- Use commands directly in `if` rather than checking `$?` after the fact
- `exit N` sets the exit code of your script
- `bash -x` traces execution step-by-step

Mini Project (In-Class)

Create a script:

1. Ask user for a directory path
2. Check if the directory exists (`-d`)
3. Count number of `.txt` files inside it
4. Print the result
5. Exit with code `1` if directory not found, `0` if successful

Summary

You learned:

- Script execution model and common errors
- Variables, quotes, and `declare`
- Input/output redirection
- Conditionals: `if/elif/else` and `case`
- Loops: `for`, `while`, `until`

Summary

- Arrays, substrings, and string replacement
- Functions with `local` variables and return values
- Exit codes and debugging with `bash -x`

Bash is small, simple, powerful and widely used in real systems.

Final Reflection

- What daily task can you automate?
- Where would loops save time?
- How would you debug a failing script?

Thank you!

Any questions?

Thejesh GN