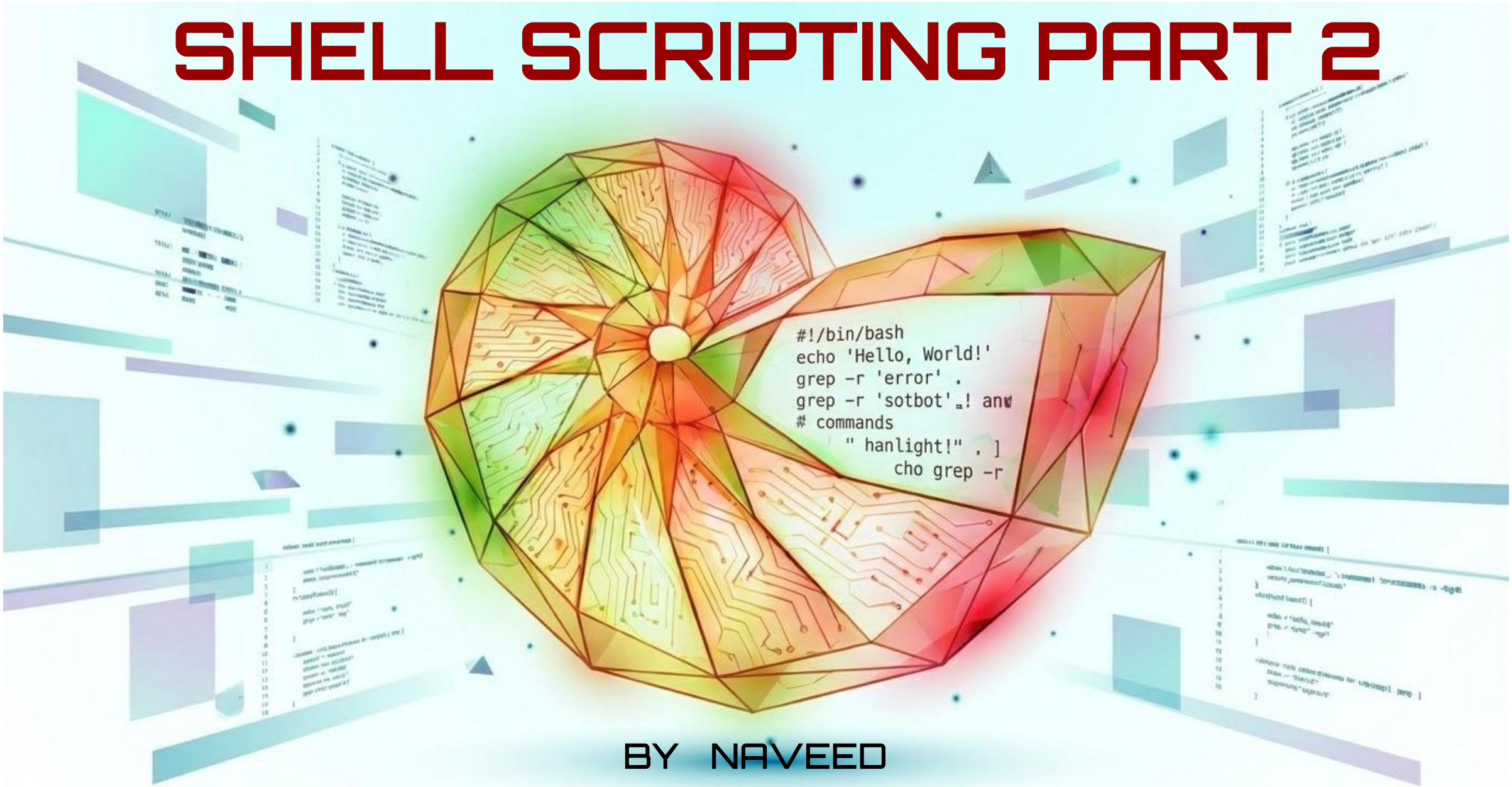
SHELL SCRIPTING PART 2



BY NAVEED

1. EXIT STATUS: THE CONCEPT

In Linux, every single command returns a hidden number called the **Exit Status** (or Return Code) when it finishes.

- This integer ranges from **0 to 255**.
- **0** represents **Success**.
- **Non-Zero (1-255)** represents various types of **Failure**.
- The shell stores the exit status of the *last executed command* in the special variable $?.
- Checking $? immediately is crucial for error handling in automation.

By
Naveed

EXIT STATUS: SUCCESS EXAMPLE

Let's see what happens when a command runs successfully.

- Here, ls /tmp is a valid command (assuming /tmp exists).
- We redirect output to /dev/null to keep the terminal clean.
- We immediately echo $? to see the result.
- The result 0 confirms the command worked perfectly.

success.sh

```
# Run a valid command
ls /tmp > /dev/null

# Check the exit status
echo "Exit Status: $?"
```

```
Exit Status: 0
```

By
Naveed

EXIT STATUS: FAILURE EXAMPLE

Now, let's force an error to see a non-zero code.

- The false command in Linux does nothing but return an exit code of **1**.
- Similarly, trying to ls a file that doesn't exist returns **2** (No such file or directory).
- Scripts use these codes to decide whether to stop or retry.

fail.sh

```
false # Built-in command to fail
echo "False Status: $?"

ls /non/existent/file
echo "LS Error Status: $?"

# False Status: 1 ls: cannot access ... LS Error Status: 2
```

By
Naveed

CUSTOM EXIT CODES

You can manually set the exit code of your **own script** using the exit command.

- If your script encounters a critical error (like missing arguments), use exit 1 (or any number > 0).
- This tells the parent process (or user) that your script failed.
- Always end a successful script with exit 0 (though it's often implied).

custom_exit.sh

```
if [ -z "$1" ]; then  echo "Error: No name
provided!"  exit 2 # Custom error code fi
 echo "Hello $1" exit 0 # Success
```

By
Naveed

2. VARIABLES: GLOBAL SCOPE

By default, **all variables** in Bash are **Global**.

- If you define a variable anywhere in the script (even inside a function), it is visible and modifiable by the entire script.
- This can cause bugs if different functions try to use the same variable name (like i or count).

global.sh

```
MY_VAR="Original"

change_it() {
MY_VAR="Changed!"
}

change_it
echo "$MY_VAR"
```

```
Changed!
```

By Naveed

VARIABLES: LOCAL SCOPE

To prevent side effects, use the local keyword inside functions.

- local var=value creates a variable that exists **only** within that function.
- It shadows any global variable with the same name.
- Once the function finishes, the local variable is destroyed, and the global one remains untouched.

local.sh

```
my_func() {
local temp="Invisible outside"
echo "Inside: $temp"
}

my_func
echo "Outside: $temp"
```

```
Inside: Invisible outside Outside:
```

By Naveed

VARIABLE SCOPE COLLISION

This example proves why local is a best practice.

- We have a global NAME "Naveed".
- The function defines a local NAME "Guest".
- Inside the function, it sees "Guest".
- After the function, the global "Naveed" is perfectly preserved.

collision.sh

```
NAME="Naveed"
test_scope() {
local NAME="Guest"
echo "In Func: $NAME"
}

test_scope
echo "Global: $NAME"
```

```
In Func: Guest Global: Naveed
```

By
Naveed

3. FUNCTIONS: DEFINITION

Functions allow you to group commands into a reusable block.

- **Syntax:** name() { commands... }
- You can also use the keyword function name { ... }.
- **Important:** You do NOT put parentheses () when *calling* the function, only when defining it.
- Define functions at the top of your script before you call them.

define.sh

```
# Definition
welcome() {
echo "Welcome to DevOps!"
echo "Let's script."
}

# Invocation (No parentheses!)
welcome
```

```
Welcome to DevOps! Let's script.
```

By
Naveed

FUNCTIONS: ARGUMENTS

Functions have their own set of **Positional Parameters**.

- Inside a function, $1 is the first argument passed *to the function*, not the script.
- $2 is the second argument, and so on.
- $# is the number of arguments passed to the function.
- This allows functions to be dynamic and reusable.

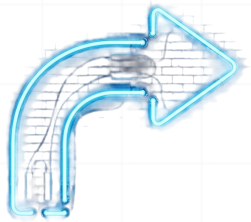
args.sh

```
greet() {
# $1 is the first argument
echo "Hello, $1!"
}

# Pass "Naveed" as argument
greet "Naveed"
greet "World"
```

```
Hello, Naveed! Hello, World!
```

By Naveed



FUNCTIONS: RETURN STATUS

The return keyword in a function sets the **Exit Status** (0-255).

- It does **NOT** return a value/string like Python or JavaScript.
- It is used to indicate Success (0) or Failure (non-zero).
- You capture it with $? immediately after the function call.

return_code.sh

```
is_valid_user() {
if [ "$1" == "Naveed" ]; then
return 0 # Success
else
return 1 # Failure
fi
}

is_valid_user "Bob"
echo "Result: $?"
```

```
Result: 1
```

By Naveed

FUNCTIONS: RETURN OUTPUT

To get data (strings/numbers) out of a function, you must **echo** it to standard output.

- The caller captures this output using **Command Substitution**: $(function_name).
- This assigns everything the function printed to the variable.
- This is the standard way to "return" values in Bash.

return_val.sh

```
get_time() {
date +%H:%M
}

# Capture output into variable
NOW=$(get_time)

echo "Current time is $NOW"
```

```
Current time is 14:05
```

By Naveed

4. CASE STATEMENT

The case statement is a cleaner alternative to multiple if-elif-else blocks.

- It matches a variable against several patterns.
- Syntax ends with esac (case spelled backwards).
- Each pattern block ends with double semicolons ;;.
- *) acts as the "default" or "else" catch-all block.

syntax.sh

```
case "$VAR" in  "pattern1")  # Commands for pattern1  ;;  "pattern2")  # Commands for pattern2 ;;  *)  # Default commands  ;; esac
```

By Naveed

CASE STATEMENT EXAMPLE

Here is a real-world example: Handling command-line arguments to start or stop a service.

- We check the value of $1.
- We handle "start", "stop", and "restart".
- Any other input triggers the Usage message.
- This pattern is used in almost every system init script.

service.sh

```
case "$1" in
start) echo "Starting app ... " ;;
stop) echo "Stopping app ... " ;;
restart) echo "Restarting ... " ;;
*) echo "Usage: $0 {start|stop}" ;;
esac

$ ./service.sh start Starting app ...
```

By Naveed

5. LOOPS: FOR LOOP (LIST)

The classic for loop iterates over a list of items.

- Useful for processing a list of strings, filenames, or server names.
- The variable item takes the value of each element in the list one by one.
- You can perform the same operation on every item.

for_list.sh

```
# Iterate over explicit strings
for fruit in Apple Banana Cherry; do
echo "I like $fruit"
done
```

```
I like Apple I like Banana I like Cherry
```

By Naveed

LOOPS: FOR LOOP (C-STYLE)

Bash supports C-style loops for numeric iteration.

- Syntax: ((initial; condition; increment)).
- Best used when you need a specific number of iterations or an index counter.
- Note the double parentheses (( )) which are specific to Bash arithmetic.

c_style.sh

```
# Count from 1 to 3
for (( i=1; i<=3; i++ )); do
echo "Counter: $i"
done
```

```
Counter: 1 Counter: 2 Counter: 3
```

By Naveed

LOOPS: WHILE LOOP

The while loop continues running **as long as** the condition is True.

- Useful for reading files line-by-line.
- Useful for creating "daemon" scripts that run forever (while true).
- Remember to update the condition variable (like count) to avoid infinite loops!

while.sh

```
count=1
while [ $count -le 3 ]; do
echo "Count: $count"
count=$((count + 1))
done
```

```
Count: 1 Count: 2 Count: 3
```

By Naveed

LOOPS: UNTIL LOOP

The until loop is the opposite of while.

- It runs **until** the condition becomes True.
- In other words, it runs as long as the condition is **False**.
- Great for waiting for a resource to become available (like waiting for a server to ping).

until.sh

```
count=1
# Run UNTIL count > 3
until [ $count -gt 3 ]; do
echo "Wait: $count"
count=$((count + 1))
done
```

```
Wait: 1 Wait: 2 Wait: 3
```


By Naveed

LOOPS: SELECT (MENUS)

select allows you to create interactive menus easily.

- It automatically prints a numbered list of options.
- It prompts the user to enter a number.
- Often combined with case to handle the choice.
- Set PS3 to change the prompt string.

menu.sh

```
PS3="Choose: "
select opt in Run Quit; do
case $opt in
Run) echo "Running..." ;;
Quit) break ;;
*) echo "Invalid" ;;
esac
done
```

```
1) Run 2) Quit Choose:
```

By Naveed

6. DEBUGGING OVERVIEW

Scripts will fail. Debugging is about finding out *why* and *where*.

- **Trace Mode:** Prints commands before executing them.
- **Exit on Error:** Stops script immediately if a command fails.
- **Syntax Check:** Checks code without running it.
- **Linting:** Use external tools like shellcheck.

By
Naveed

DEBUGGING: SET MODES

The set command controls shell options.

- set -x (xtrace): Prints each command to stderr before execution. Great for tracing logic.
- set -e (errexit): Aborts script if *any* command returns a non-zero exit status.
- set -u (nounset): Treats unset variables as an error (prevents "rm -rf /" disasters).

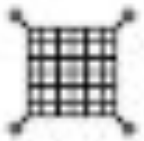
debug_modes.sh

```
# Enable trace & exit on error
set -xe

NAME="Naveed"
echo "Hello $NAME"

ls /nonexistent # Script dies here
echo "This won't run"
```

```
+ NAME=Naveed + echo 'Hello Naveed' Hello Naveed
+ ls /nonexistent (Error: Script exits)
```



DEBUGGING: TRAP

trap allows you to catch signals and errors to perform cleanup.

- Catch **ERR** to run code when a command fails.
- Catch **EXIT** to run code when the script finishes (successfully or not).
- Commonly used to remove temporary files or log errors.

trap.sh

```
trap 'echo "Error at line $LINENO!"' ERR

echo "Running..."
ls /missing_file
echo "Done"
```

```
Running...  ls: cannot access...  Error at line 3!
```

By
Naveed

7. SED (STREAM EDITOR)

sed is a powerful tool for parsing and transforming text in a stream (file or pipeline).

- Most common use: **Find and Replace**.
- Basic Syntax: s/search_pattern/replacement/flags.
- By default, it prints the modified text to the screen (Standard Output).
- It does not modify the original file unless you use a specific flag.

basic_sed.sh

```
echo "Hello World" | sed 's/World/Naveed/'
```

```
Hello Naveed
```

SED: INLINE EDIT & DELETION

- -i: Edit file **In-place** (save changes to file).
- d: **Delete** matching lines.
- Warning: -i is destructive. Test your regex without it first!

advanced_sed.sh

```
# Replace OLD with NEW in file sed -i
's/OLD/NEW/g' config.txt
 # Delete lines containing "error" sed -i
'/error/d' log.txt
```

By
Naveed

THANKS FOR READING! 🙏

You've leveled up to Intermediate Shell Scripting.

Follow **Naveed Ibrahim A** for more daily DevOps bites.