Shell scripting notes

1. Hello world shell script

```
#!/bin/bash
# This is a simple shell script that prints "Hello, World!" to the terminal
echo "Hello, World!"
```

Hello world shell script explained

```
#!/bin/bash
##!, called the shebang (pronounced “sha-bang”).
#Use the Bash shell (located at /bin/bash) to interpret and execute all commands in this script.
# This is a simple shell script that prints "Hello, World!" to the terminal

echo "Hello, World!”
```

Other Common Shebang Options:

Shebang	Interpreter	Use Case
`#!/bin/bash`	Bash shell	Most common; full-featured scripting shell.
`#!/bin/sh`	Basic shell (often linked to `dash` on Ubuntu)	Faster, POSIX-compliant, but lacks some Bash features.
`#!/usr/bin/env bash`	Finds Bash from the user’s `$PATH`	More portable across systems where Bash may not be in `/bin/`.
`#!/bin/zsh`	Z shell	Used if you prefer Zsh scripting features.
`#!/usr/bin/python3`	Python interpreter	Used for Python scripts.
`#!/usr/bin/perl`	Perl interpreter	Used for Perl scripts.

2.

Display Current Date and Time

File: `current_time.sh`

```
#!/bin/bash
echo "Current date and time: $(date)"
```

Check Disk Usage

File: `disk_usage.sh`

```
#!/bin/bash
echo "Disk usage report:"
df -h
```

List Files in a Directory

File: list_files.sh

```
#!/bin/bash
echo "Files in current directory:"
ls -l
```

Check if a File Exists

File: file_check.sh

```
#!/bin/bash
read -p "Enter filename: " file
if [ -f "$file" ]; then
  echo "File '$file' exists."
else
  echo "File '$file' does not exist."
fi
```

Add Two Numbers

File: add_numbers.sh

```
#!/bin/bash
read -p "Enter first number: " a
read -p "Enter second number: " b
sum=$((a + b))
echo "Sum: $sum"
```

Print System Uptime

File: system_uptime.sh

```
#!/bin/bash
echo "System has been up for:"
uptime -p
```

Check User Login

File: check_user.sh

```
#!/bin/bash
read -p "Enter username to check: " user
if id "$user" &>/dev/null; then
  echo "User '$user' exists."
else
  echo "User '$user' not found."
fi
```

Backup a Directory

File: backup_dir.sh

```
#!/bin/bash
read -p "Enter source directory: " src
```

```
read -p "Enter backup directory: " dest
tar -czf "$dest/backup_$(date +%F).tar.gz" "$src"
echo "Backup completed successfully!"
```

Check Internet Connectivity

File: check_internet.sh

```
#!/bin/bash
if ping -c 1 8.8.8.8 &>/dev/null; then
  echo "Internet is working."
else
  echo "No internet connection."
fi
```

Find the Length of a String

File: string_length.sh

```
#!/bin/bash
read -p "Enter a string: " str
echo "Length of string: ${#str}"
```

${str} — Parameter Expansion

- In Bash, variables are referenced using **${variable_name}.**
- ${str} retrieves the **value** stored in the variable str.

Example:

```
str="hello"
echo ${str}
# Output: hello
```

So, ${str^^} is a **modified version** of ${str} — not the raw value.

${str^^} — Case Conversion (Uppercase)

This is a **special Bash 4+ syntax** that changes the case of letters inside a variable.

Syntax options:

Expression	Description	Example
${var^}	Converts **first character** to uppercase	"hello" → "Hello"
${var^^}	Converts **entire string** to uppercase	"hello" → "HELLO"
${var,}	Converts **first character** to lowercase	"HELLO" → "hELLO"
${var,,}	Converts **entire string** to lowercase	"HELLO" → "hello"

N.B: below code will not work with sh <filename> as sh <filename> executes it on shell and not bash so change the permission and execute it as ./<file>

Convert String to Uppercase

File: uppercase.sh

```
#!/bin/bash
read -p "Enter a string: " str
echo "Uppercase: ${str^^}"
```

Convert String to Lowercase

File: lowercase.sh

```
#!/bin/bash
read -p "Enter a string: " str
echo "Lowercase: ${str,,}"
```

Reverse a String

File: reverse_string.sh

```
#!/bin/bash
read -p "Enter a string: " str
rev_str=$(echo "$str" | rev)
echo "Reversed string: $rev_str"
```

Bash specific code aga
sh <filename> will not work.

Check if Two Strings are Equal

File: compare_strings.sh

```
#!/bin/bash
read -p "Enter first string: " str1
read -p "Enter second string: " str2

if [[ "$str1" == "$str2" ]]; then
  echo "Strings are equal."
else
  echo "Strings are not equal."
fi
```

Bash specific code

Extract a Substring

File: substring_extract.sh

```
#!/bin/bash
read -p "Enter a string: " str
read -p "Enter starting position: " pos
read -p "Enter length: " len
echo "Substring: ${str:$pos:$len}”
```

/bin/sh vs /bin/bash

Feature	/bin/sh	/bin/bash
Name	Bourne Shell (or POSIX shell)	Bourne Again Shell
Speed	Slightly faster (lighter)	Slightly slower (more features)

Portability	Very portable (exists on all UNIX systems)	Common but not guaranteed everywhere
Features	Basic, POSIX-compliant	Rich: arrays, functions, string ops, brace expansion, [[ ]], ==, etc.
Use case	Simple, portable scripts	Complex, interactive, feature-rich scripts

---

Example 1 — Bash-specific feature that fails in /bin/sh

Script: example.sh

```
#!/bin/bash
name="vilas"
echo "Uppercase: ${name^^}"
```

Output:

```
Uppercase: VILAS
```

Now change the first line to:

```
#!/bin/sh
```

and run again.

Output:

```
example.sh: 3: Bad substitution
```

Why it failed:

- /bin/sh doesn't understand the ${var^^} syntax (Bash-only feature).
- So the script works in Bash, but breaks in sh.

Example 2 — Arrays

Bash version (/bin/bash):

```
#!/bin/bash
colors=("red" "green" "blue")
echo "First color: ${colors[0]}"
echo "All colors: ${colors[@]}"
```

Output:

```
First color: red
All colors: red green blue
```

If you run the same script with /bin/sh:

```
syntax error: "(" unexpected
```

/bin/sh doesn't support arrays.

Example 3 — Conditional Expressions

Bash version (/bin/bash):

```
#!/bin/bash
str="devops"
if [[ $str == devops ]]; then
  echo "Match"
fi
```

Works fine — [[...]] is a Bash keyword.

In /bin/sh:

```
[: not found
```

/bin/sh only supports single brackets [...].

Example 4 — Command substitution and arithmetic

Both /bin/sh and /bin/bash support:

```
#!/bin/sh
echo $((3 + 4))
```

Works fine — because arithmetic expansion $((...)) is POSIX-compliant.
So if you stick to basic syntax, /bin/sh is enough.

Practical Guidelines

Situation	Which to use	Why
Simple, portable scripts (init, cron, Docker entrypoint)	/bin/sh	Works across all Unix/Linux systems, lightweight
Scripts using arrays, regex, string manipulation, or color output	/bin/bash	Full Bash features available
Scripts meant for automation in modern Linux distros	/bin/bash	Bash is default on most systems
Minimalist or embedded systems (like Alpine)	/bin/sh (linked to ash)	

---

shell script to create a user without following standards

```
#!/bin/bash
USERNAME="$1"
GROUPNAME="$2"

groupadd "$GROUPNAME"
useradd -m -s /bin/bash -g "$GROUPNAME" "$USERNAME"

# Add welcome message to .bashrc
echo "echo \"Welcome, $USERNAME!!\"" >> /home/$USERNAME/.bashrc

echo "Setup complete! When '$USERNAME' logs in, they'll see a welcome message."
```

Same code in standard format

```
#!/bin/bash
# ================================================
# Script Name: create_user.sh
# Description: Create a new Linux user and group,
#              set home directory and shell.
# Usage: sudo ./create_user.sh <username> <groupname>
# ================================================

# Exit immediately if a command fails
set -e

# Check if script is run as root (user management needs root)
if [[ $EUID -ne 0 ]]; then
  echo "This script must be run as root (use sudo)"
  exit 1
fi

# Check for required arguments
```

```
if [[ $# -ne 2 ]]; then
  echo "Usage: $0 <username> <groupname>"
  exit 1
fi

USERNAME="$1"
GROUPNAME="$2"

# Step 1: Create group (if not exists)
if getent group "$GROUPNAME" > /dev/null; then
  echo " Group '$GROUPNAME' already exists."
else
  echo "Creating group '$GROUPNAME'..."
  groupadd "$GROUPNAME"
  echo " Group '$GROUPNAME' created."
fi

# Step 2: Create user (if not exists)
if id "$USERNAME" &>/dev/null; then
  echo " User '$USERNAME' already exists."
else
  echo " Creating user '$USERNAME'..."
  useradd -m -s /bin/bash -g "$GROUPNAME" "$USERNAME"
  echo " User '$USERNAME' created with home directory /home/$USERNAME"
fi

# Step 3: (Optional) Add user to additional groups
# Example: Add to 'sudo' if needed
# usermod -aG sudo "$USERNAME"

# Step 4: Display user info
echo "----------------------------------------"
echo "User Information:"
id "$USERNAME"
echo "----------------------------------------"

# Step 5: Set password (optional prompt)
read -p "Do you want to set a password for $USERNAME? (y/n): " choice
if [[ "$choice" == "y" || "$choice" == "Y" ]]; then
  passwd "$USERNAME"
fi

echo " User setup completed successfully!"
```

Assignment question

```
### Objective
Create a reusable utility script and a main script that automate project setup by creating
uniquely named directories with timestamps.
This exercise will help you practice **functions, sourcing scripts, argument handling, and
timestamp-based directory creation** in bash.

---

### Instructions
1. **Create the Utility Library (`utils.sh`)**
   - Define a `bash` function named `create_timestamped_dir()` that:
     - Takes **one argument**: the project name (e.g., "webapp").
     - Creates a directory in `/tmp` named `[project_name]-[timestamp]`, with the timestamp in
`YYYYMMDD-HHMMSS` format.
     - Prints the **full path** of the created directory to the console.

2. **Create the Main Script (`setup_project.sh`)**
   - Make it **executable** with the correct `bash` shebang.

3. **Implement the Main Logic**
   - `source` the `utils.sh` file to import the `create_timestamped_dir` function.
   - Call `create_timestamped_dir` with the project name `"my-new-app"` to create the directory.

---

### Important Notes
- Use `source "$(dirname "$0")/utils.sh"` to import functions from the library script reliably.
```

- Use `date +%Y%m%d-%H%M%S` to generate timestamps in the desired format.
- Access function arguments inside the function using `$1`.
- Use `mkdir -p` to safely create directories.

Outcome
By completing this lab, you will be able to:
- Write reusable bash functions and import them across scripts.
- Dynamically create directories with unique timestamps.
- Use arguments and command substitution to make scripts flexible and automated.

```
#!/bin/bash
# utils.sh — Utility functions for project setup

# Function to create a timestamped directory
create_timestamped_dir() {
    local project_name="$1"

    # Check if project name is provided
    if [ -z "$project_name" ]; then
        echo "Error: Project name not provided."
        return 1
    fi

    # Generate timestamp (format: YYYYMMDD-HHMMSS)
    local timestamp
    timestamp=$(date +%Y%m%d-%H%M%S)

    # Construct directory path under /tmp
    local dir_path="/tmp/${project_name}-${timestamp}"

    # Create the directory
    mkdir -p "$dir_path"

    # Print the full path of the created directory
    echo "Directory created: $dir_path"
}
```

```
#!/bin/bash
# setup_project.sh — Main script to set up a new project

# Source the utility script (located in the same directory)
source "$(dirname "$0")/utils.sh"

# Call the utility function with project name
create_timestamped_dir "my-new-app"
```

Additional topic added based on students request

1. What are ACLs?

ACLs allow you to define **fine-grained permissions** for multiple users or groups **on the same file or directory** — something not possible with basic UNIX permissions.

Example:

Normally you can assign access to:

- One **owner**
- One **group**
- Everyone else (**others**)

With **ACLs**, you can say things like:

> "User *john* can read this file, *mary* can write it, and *devgroup* can execute it."

---

2. Checking if ACLs are enabled

Most modern filesystems (ext4, xfs, btrfs) support ACLs by default.

To confirm:

```
mount | grep acl
```

If you see acl in the mount options, it's enabled.
If not, remount with ACL support:

```
sudo mount -o remount,acl /home
```

Or add acl to /etc/fstab for persistence:

```
UUID=xxxx /home ext4 defaults,acl 0 2
```

---

3. Viewing ACLs

To view ACLs on a file or directory:

```
getfacl filename
```

Example:

```
$ getfacl project.txt
# file: project.txt
# owner: root
# group: root
user::rw-
user:john:r--
group::r--
mask::r--
other::—
```

This shows normal owner/group permissions **plus** extra ACL entries.

---

4. Setting ACLs

Use setfacl to assign ACL entries.

Give a user read access:

```
setfacl -m u:john:r-- project.txt
```

Give a group write access:

```
setfacl -m g:developers:rw- project.txt
```

Verify:

```
getfacl project.txt
```

---

5. Removing ACL entries

Remove one entry:

```
setfacl -x u:john project.txt
```

Remove all ACLs:

```
setfacl -b project.txt
```

---

6. Setting ACLs on directories (and recursive)

Example:

```
setfacl -R -m u:john:rwX /var/www/
```

- -R → recursive

- X → execute permission only on directories (smart mode)

---

7. Default ACLs (for newly created files)

Default ACLs apply to **new files/directories** created inside a directory.

Example:

```
setfacl -d -m u:john:rwX /var/www/
```

Now any new file in /var/www/ will automatically give John read/write access.

Check:

```
getfacl /var/www/
```

---

8. Understanding the “mask” field

When multiple ACL entries exist, Linux uses the **mask** to define the *maximum allowed permissions* for all users/groups other than the owner.

You can modify it with:

```
setfacl -m m::r-- file.txt
```

> Even if an ACL entry grants rw, if the mask is r--, the user effectively gets **read-only** access.

---

9. Integrating ACLs with scripts

Example: give a team shared access automatically:

```
#!/bin/bash
SHARE_DIR="/data/team"
USER_LIST="alice bob charlie"

for user in $USER_LIST; do
  setfacl -m u:$user:rwx $SHARE_DIR
done
```

Make it executable:

```
chmod +x set_team_acl.sh
```