Introduction To Shell Scripting

What is Shell Scripting?

Shell scripting is writing a series of **commands** in a text file to automate tasks in a **Linux/Unix shell** (like bash).

Think of it like writing a recipe: Instead of typing commands one-by-one, you **save them in a .sh file**, and run them all together.

Example:

```
#!/bin/bash
echo "Hello, Connections!"
date
```

Run it:

1. Save as script.sh
2. Make it executable: chmod +x script.sh
3. Run: ./script.sh

Who invented Shell Scripting?

Shell scripting wasn't "invented" by one person like a tool — it **evolved** with Unix.

But here's the key timeline:

- **Ken Thompson** (1971) – Created the **first Unix shell** called **sh** (Bourne shell).
- **Stephen Bourne** (1979) – Wrote the improved **Bourne Shell (sh)**, which made scripting possible.

So, **Stephen Bourne** is mostly credited for **popularizing shell scripting**.

🔁 Later came other shells:

- bash (Bourne Again Shell)
- zsh, ksh, etc.

Q: Why is Shell Scripting important for DevOps?

Shell scripting is **super important** in DevOps because it helps automate stuff — which is what DevOps is all about.

Here's why it matters:

1. **Automation:** Run builds, deployments, tests, cleanups — all with one script.
2. **CI/CD Pipelines**: Shell scripts are often used in tools like **Jenkins, GitHub Actions, GitLab CI** to automate pipelines.
3. **Server Management**: Start/stop services, backup logs, check disk space — easily scripted.
4. **No Extra Tools Needed**: Works out-of-the-box on any Linux system (which most servers run).
5. **Glue Language**: Connects tools like Docker, Git, Kubernetes, etc.

In DevOps, if you can't automate, you can't scale. Shell scripts = your automation superpower.

Q: How do I write and run my first shell script?

Step 1: Create the script file

Open terminal and type:

```
nano hello.sh
```

Step 2: Write this code

```
#!/bin/bash
echo "Hello, Connections!"
date
```

Save & Exit (in nano): Press CTRL + X, then Y, then Enter.

Step 3: Make it executable

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

Step 4: Run the script

Run	Output
`./hello.sh`	`Hello, Connections!` `Sat Jun 14 16:05:00 IST 2025`

Q: What is #!/bin/bash and explain the syntax of the previous script?

Let’s break down the script line-by-line:

```
#!/bin/bash
echo "Hello, Connections!"
date
```

Line 1: #!/bin/bash

- Called **shebang**.
- Tells the system: **"Use the Bash shell to run this script."**
- If you skip it, the system might run your script with the **wrong shell** (like sh, dash, etc.).

Line 2: echo "Hello, Connections!"

- echo = prints text on the screen.
- "Hello, Connections!" = string being printed.

Output: Hello, Prashant!

Line 3: date

- A built-in command that shows the **current date and time**.

Output: e.g., Sat Jun 14 16:10:45 IST 2025

Q: What are variables in Shell Scripting and how to use them?

Variables store data — like names, numbers, paths — to use later in the script.

Example Script with Variables:

```
#!/bin/bash

name="Tanvir Mulla"
age=21

echo "My name is $name"
echo "I am $age years old"
```

Syntax Rules:

- **No spaces** around =
  - ✅ name="Tanvir Mulla"
  - ❌ name = "Tanvir Mulla"
- To use a variable: add $ before the name → $name

Output	

```
My name is Tanvir Mulla
I am 21 years old
```

You can even take input like this:

```
read username
echo "Welcome, $username!"
```

Q: What are comments in Shell Scripting?

Comments are lines that **explain the code** — they're **ignored by the shell**.

✅ Syntax:

```
# This is a comment
```

🧠 **Where to use:**

- Above commands to explain
- In complex logic
- To disable code (temporarily)

Example:

```
#!/bin/bash

# This script greets the user
name="Tanvir Mulla"     # Storing name
echo "Hello, $name" # Printing greeting
```

💡 Use comments wisely — not for obvious stuff, but for **clarity and logic**.

Q: What is if-else in Shell Scripting?

if-else is used to **make decisions** — like "if this is true, do that, else do something else."

Basic Syntax:

```
if [ condition ]
then
   # commands if true
else
   # commands if false
fi
```

fi = "if" reversed → marks the end of if block.

Example:

```
#!/bin/bash

echo "Enter your age:"
read age

if [ $age -ge 18 ]
then
    echo "You are an adult."
else
    echo "You are a minor."
fi
```

Breakdown:

- read age → takes user input.
- [$age -ge 18] → checks if age ≥ 18.
- -ge = greater than or equal.

Common operators:

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

Q: What is elif in Shell Scripting?

elif stands for **"else if"** — used when you have **multiple conditions** to check, not just if and else.

Syntax

```
if [ condition1 ]
then
   # do this
elif [ condition2 ]
then
   # do that
else
   # do something else
fi
```

Example

```
#!/bin/bash

echo "Enter your marks:"
read marks

if [ $marks -ge 90 ]
then
    echo "Grade: A+"
elif [ $marks -ge 75 ]
then
    echo "Grade: A"
elif [ $marks -ge 60 ]
then
    echo "Grade: B"
else
    echo "Grade: C or below"
fi
```

Use elif to avoid writing too many nested ifs. Cleaner and readable.

Q: What are loops in Shell Scripting?

Loops are used to **repeat a block of code** multiple times — very useful for automation and repetition.

Types of Loops in Shell Scripting

1. for loop

Runs for a **fixed list or range** of values.

Input	Output
`#!/bin/bash` `for i in 1 2 3 4 5` `do` `  echo "Number: $i"` `done`	`Number: 1` `Number: 2` `...`

2. while loop

Runs **while a condition is true**.

Input	Output
`#!/bin/bash` `count=1` `while [ $count -le 5 ]` `do` `  echo "Count is: $count"` `  count=$((count + 1))` `done`	`Count is: 1` `Count is: 2` `...`

3. until loop

Runs **until the condition becomes true** (opposite of while).

Input	Output
`#!/bin/bash` `x=1` `until [ $x -gt 5 ]` `do` `  echo "x is: $x"` `  x=$((x + 1))` `done`	`x is: 1` `x is: 2` `...`

Summary Table:

Loop	Runs When...	Use Case
for	For each value in list	Known items/ranges
while	Condition is **true**	Repeat till false
until	Condition is **false**	Repeat till true

Q: What are functions in Shell Scripting?

Functions are **reusable blocks of code**. You define them once and **call them anytime**.

Syntax	
`function_name() {` `  # code here` `}`	`function_name {` `  # code here` `}`

Input	Output
`#!/bin/bash` `greet() {` `  echo "Hello, Connections!"` `}` `greet  # Calling the function`	`Hello, Connections!`

Functions with arguments	
Input	**Output**
`say_hello() {` `  echo "Hello, $1!"` `}` `say_hello "Tanvir"`	`Hello, Tanvir!`

Q: What are arrays in Shell Scripting?

Arrays let you **store multiple values** in a single variable — super handy when dealing with lists!

1. Declare an Array

```
fruits=("apple" "banana" "cherry")
```

2. Access elements

```
echo ${fruits[0]}  # apple
echo ${fruits[1]}  # banana
```

3. Length of Array

```
echo ${#fruits[@]}  # 3
```

4. Loop Through Array

```
for fruit in "${fruits[@]}"
do
  echo "$fruit"
done
```

```
apple
banana
cherry
```

5. Add elements

```
fruits+=("orange")
```

6. Remove elements:

You can't remove directly, but you can unset:

```
unset fruits[1]  # removes banana
```

Use arrays when working with multiple items like file names, user inputs, package lists, etc.

Q: What are Strings in Shell Scripting?

A **string** is just a **sequence of characters** — like text, names, paths, etc.

`name="Tanvir"` `echo "Hello, $name"`	`Hello, Tanvir`

String with spaces:

Always use quotes if your string has spaces:

```
greet="Hello World"
echo "$greet"
```

String Operations	
Length	`echo ${#name}   # 8`
Concatenation	`full="$name is learning Shell"` `echo "$full"`
Compare Strings	`if [ "$name" == "Tanvir" ]; then` `  echo "Match!"` `fi`

Q: What is Substring in Shell Scripting?

A **substring** is a part of a string. You can extract it using **${variable:position:length}**.

Syntax	Example
`${string:start:length}`	`str="DevOpsRocks"` `echo ${str:0:6}  # DevOps` `echo ${str:6:5}  # Rocks`

Breakdown:

- str: variable name
- 0: starting index (0-based)
- 6: number of characters to extract

Extract till end:

```
echo ${str:6}       # From index 6 to end → Rocks
```

Negative indexing isn't supported directly:

Want last 2 chars? Use:

```
echo ${str: -2}     # ks  (space after `:` is required!)
```

Q: What is file handling in Shell Scripting?

File handling lets you **read, write, or modify files** directly from a script — super useful in DevOps automation.

1. Reading a File Line by Line

``bash #!/bin/bash filename="file.txt" while read line do echo "Line: $line" done < "$filename" ```	``` Line: Hello Line: World Line: Tanvir ```

2. Writing to a File

```bash
echo "This is a new line" > output.txt
# Overwrites the file
```

3. Appending to a File

```bash
echo "Another line" >> output.txt
# Adds to the end without deleting existing content.
```

4. Checking if File Exists

```bash
if [ -f "file.txt" ]; then
  echo "File exists"
else
  echo "File not found"
fi
```

Q: What are some useful keyboard shortcuts in Linux terminal?

Here are some powerful shortcuts to speed up terminal work:

Shortcut	Action Description
Ctrl + K	Delete from cursor to **end of the line**
Ctrl + U	Delete from cursor to **start of the line**
Ctrl + W	Delete the **word before** the cursor
Ctrl + R	**Search previous commands** in history
Ctrl + L	**Clear** the terminal screen (same as clear command)
Ctrl + S	**Pause** output to the screen
Ctrl + Q	**Resume** output (if paused with Ctrl + S)
Ctrl + C	**Terminate** the running command
Ctrl + Z	**Suspend** current command and send to background

These are very helpful when writing shell scripts or debugging — they save a lot of time!