CHAPTER 11 :SHELL SCRIPTING

BY HANSIKA BHAMBHANEY

OBJECTIVES:

- Understand how a shell interprets commands and scripts.
- Learn to comment and document scripts properly.
- Use variables and constants safely and correctly.
- Know how the shell environment is set up at startup.
- Identify and avoid common scripting problems.
- Recognize when not to use bash for certain tasks.

11.1 What does the shell do?

The shell is a special program that acts as an interface between the user and the operating system kernel. It interprets the commands you type and converts them into actions by interacting with system calls, file systems, and running programs.

When you log into a Linux or Unix system, the shell starts automatically. It waits for you to type a command, then executes it. This process is called a command-line interface (CLI) interaction. Bash is one of the most popular shells because it is powerful, flexible, and supports advanced features like scripting and command completion.

A shell can do much more than simply run single commands. It can execute sequences of commands, handle input and output redirection, run commands in the background, and manage processes. More importantly, it can execute scripts, which are files containing lists of commands to be run as a batch, helping to automate tasks and simplify repetitive operations.

Example

When you type:

```
ls -l
```

The shell interprets `ls` as a command to list files, with `-l` as an option to display details. The shell then finds where `ls` is located (by searching the directories in `$PATH`), executes it, and displays the output on your terminal.

The shell as a programming language

Besides running individual commands, the shell can also:

- Use variables to store values temporarily.
- Support loops (`for`, `while`) and conditionals (`if`, `case`) to make decisions.
- Allow function definitions for modular, reusable code.
- Handle signals and errors gracefully.

Because of these capabilities, bash is not just a command runner but also a **scripting language** that allows you to build complex workflows.

Interactive vs non-interactive

- **Interactive shell**: When you type commands manually and get immediate feedback.
- **Non-interactive shell**: When running scripts automatically without user interaction (e.g., startup scripts, cron jobs).

11.1.1 Comments

When writing bash scripts, **comments** play a crucial role in making your code readable, understandable, and easier to maintain. Comments allow you to explain what each part of your script does so that others — or even you, after some time — can understand the logic without guessing.

In bash, a comment starts with the hash symbol #. Anything written after # on that line is ignored by the shell when executing the script.

Example of a comment above a line

```
#!/bin/bash

# Print a welcome message to the user
echo "Welcome to my script!"
```

Example of an inline comment

```
count=5   # Set the initial value of count to 5
```

Clarity: Comments explain why you did something, not just what you did.

Collaboration: If others read your script, they can understand the steps without needing extra explanations.

Maintenance: When you return to your own script after a long time, comments help you remember the logic and structure.

Debugging: You can temporarily comment out a line of code to disable it for testing.

Shebang line

The only exception is the very first line in the script:

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

Although it starts with #, it is **not** a comment. This line is called the **shebang** and tells the system which interpreter to use to execute the script.

11.1.2 Variables and constants

Variables

Variables are used in bash scripts to **store values** that can be reused and manipulated throughout the script. They make your scripts dynamic and flexible, allowing you to easily update values without changing code everywhere.

In bash, you do not need to declare a type for variables — everything is treated as a string unless used in arithmetic contexts.

The syntax is simple:

```
variable_name=value
```

There should be **no spaces** on either side of the `=` sign.

Example:

```
name="Hansika"
echo "Welcome, $name"
```

Here, `name` holds the value "Hansika", and `$name` is used to refer to its value.

Using variables

When you want to access the value stored in a variable, you put a `$` before the variable name:

```
echo $name
```

Assigning command output to a variable

You can also store the output of a command in a variable using **command substitution**:

```
today=$(date)
echo "Today's date is $today"
```

Constants (readonly variables)

Sometimes, you want to ensure that a variable's value cannot be changed after it is set. You can do this by marking it as **readonly**, making it act like a constant.

```
pi=3.14159
readonly pi
```

After this, if you try to change `pi`, bash will throw an error.

```
pi=3.14  # This will give an error because pi is readonly
```

Exporting variables

If you want a variable to be available in child processes or subshells, you need to **export** it.

```
path="/usr/local/bin"
export path
```

1.Variable names are case-sensitive (`Name` and `name` are different).

2.By convention, environment variable names are often written in uppercase.

3.Always quote variable expansions (e.g., `"$name"`) when they might contain spaces or special characters to avoid unexpected behavior.

11.2 Starting up

When you start a shell session in Linux or Unix, **several configuration files are executed automatically** to set up your environment. These files define variables, set paths, establish default editors, and apply user-specific or system-wide settings. Understanding this startup process is essential for writing scripts and managing a consistent working environment.

Login shell vs non-login shell

A **login shell** is started when you log in to the system directly (e.g., through a terminal, SSH, or console). It runs certain files meant to initialize your environment fully.

A **non-login shell** is started when you open a new terminal window inside a graphical session or run a script. It does not execute all login initialization files.

Files executed at startup

For login shells

- `/etc/profile` — The system-wide configuration file for all users.

- One of these user-specific files (whichever exists):
  - `~/.bash_profile`
  - `~/.bash_login`
  - `~/.profile`

These files typically set environment variables (like `PATH`), define shell options, and execute other startup scripts.

For non-login interactive shells

- `~/.bashrc`

This file is usually used to set aliases, functions, prompt styles, and other interactive features.

Linking files together

Often, users make sure that `~/.bash_profile` sources `~/.bashrc`, so all configurations are applied regardless of how the shell starts.

Example:

```
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi
```

This means: "If the file `.bashrc` exists, run (source) it.

When running scripts, especially in non-login shells, **some environment settings (like custom variables or paths) might not be available** if they are only defined in `.bash_profile` or `/etc/profile`.

This is why it is a good practice to **explicitly define or export variables** within scripts, rather than assuming they'll always be inherited from your interactive environment.

Examples of what gets set up

- Default text editor (`export EDITOR=vim`)
- Custom PATH settings (`export PATH=$PATH:/custom/scripts`)
- Prompt appearance (`export PS1="[\u@\h \W]\$ "`)

POSSIBLE OPTIONS :

Long option	Option	Examples where used
—long	-l	ls
—line	-l	wc
—links	-L	cpio, ls
—output	-o	cc, sort
—quiet	-q	who
—recursive	-r	rm
—recursive	-R	ls
—silent	-s	Synonym for -quiet
—unique	-u	*sort*

Long option	Option	Examples where used
—all	-a	*du*, *ls*, *nm*, *stty*, *uname*, *unexpand*
—append	-a	etags, tee, time
—binary	-b	cpio, diff
—blocks	-b	head, tail
—date	-d	touch
—directory	-d	cpio
—exclude-from	-X	tar
—file	-f	fgrep
—help	-h	man

11.3 Potential problems

While bash scripting is powerful and flexible, it also comes with **common pitfalls** that can lead to unexpected behavior, errors, or even system issues. Understanding these potential problems helps you write more reliable and secure scripts.

Permission issues

One of the most frequent problems is forgetting to set the correct **execute permission** on a script. By default, new files do not have execute permissions, so a script won't run directly.

Example solution:

```
chmod +x myscript.sh
```

Path and environment differences

Scripts might rely on certain commands or files being in the system's `PATH`. When running interactively, your environment might include custom paths, but non-login or non-interactive shells (like cron jobs or automated tasks) might have a minimal environment.

Example fix: Always specify full paths in scripts.

```
/usr/bin/ls
```

Or explicitly set `PATH` at the beginning of the script:

```
export PATH=/usr/local/bin:/usr/bin:/bin
```

Variable mistakes

A common error is **forgetting to quote variables**, especially when they contain spaces or special characters. This can cause unexpected splitting or interpretation.

Incorrect example:

```
filename=$1
cat $filename
```

If `$filename` is "my file.txt", this will fail.

Correct example:

```
cat "$filename"
```

Unintended overwriting

Scripts that create or modify files might overwrite important data if filenames or paths are not handled carefully. Always check whether a file exists before overwriting, or use backup strategies.

Example:

```
if [ -f "$file" ]; then
    echo "File exists, backing up..."
    cp "$file" "$file.bak"
fi
```

Infinite loops

Mistakes in loop conditions can lead to **infinite loops**, consuming system resources and potentially freezing terminals.

Example faulty loop:

```
while [ "$input" != "quit" ]; do
    echo "Running..."
done
```

If `$input` is never updated, this loop never exits.

Debugging challenges

Scripts often fail silently if errors aren't handled explicitly. It's helpful to use bash debugging options:

```
set -x  # Print each command as it runs
set -e  # Exit script immediately if any command fails
```

Differences across systems

Bash scripts might work perfectly on one system but fail on another because of:

- Different bash versions
- Missing utilities
- Different default shells

To improve portability, avoid system-specific features when possible and test scripts in different environments.

11.4 Don’t use bash

While bash is a powerful and widely used shell scripting language, there are situations where **using bash may not be the best choice**. This section encourages you to recognize the **limitations of bash** and understand when it's better to use another language or tool.

Portability concerns

Bash scripts are **not fully portable** across all Unix-like systems.
Some systems, such as older versions of Solaris or embedded Linux distributions, may not include bash by default. They might only support **POSIX-compliant shells** like `sh`.

If you write a script using bash-specific features (like `[[ ]]`, arrays, or advanced string manipulation), it might **fail to run on systems that only support `/bin/sh`**.

In such cases, writing scripts using **strict POSIX syntax** is safer — or using other scripting languages that are more universally supported.

Performance limitations

Bash is excellent for running and combining small system commands, but **it's not ideal for heavy computation or processing large data sets**.

For example:

- Tasks involving complex logic
- Processing large log files
- Performing mathematical operations or working with data structures

In these situations, using a **more efficient language** like Python, Perl, or even compiled languages like C or C++ will be faster and more scalable.

Poor handling of structured data

Bash has **no native support for arrays of objects, dictionaries, or JSON**. Handling structured data becomes messy and error-prone.

For example, parsing JSON in bash often requires external tools like `jq`, whereas in Python you could do it natively with:

```
import json
data = json.loads(json_string)
```

If your script interacts with APIs, handles nested data, or requires complex data handling, bash is not the best tool.

Readability and maintainability

Bash scripts can become **difficult to read and maintain** as they grow. Nested loops, long chains of conditionals, and lack of proper error handling can make scripts error-prone.

Languages like Python offer:

- Clear syntax
- Better exception handling
- Readable indentation
- Built-in modules for common tasks

So for long-term maintainability, choosing a modern scripting language is often better.

When not to use bash

Avoid using bash for:

- Writing complex business logic
- GUI applications
- Advanced file manipulation and parsing
- Scientific computation
- Multi-threaded operations

CONCLUSION

Chapter 11 emphasizes that while bash is a powerful and versatile tool for automating tasks, managing files, and simplifying system administration, it must be used thoughtfully. By learning to write and execute scripts, effectively using comments, variables, and constants, and understanding how the shell starts up and handles environments, users can create reliable and maintainable scripts. However, it is equally important to recognize common pitfalls such as permission issues, variable mismanagement, and portability problems. Finally, the chapter reminds us that bash is best suited for simpler automation and system tasks, and not ideal for complex, performance-heavy, or highly portable applications — encouraging users to choose the right tool for the right job.