Bash

The Linux command line

Roadmap

- What is a **shell**? What is **Bash**?
- Reading the **prompt**
- Typing commands: command / options / parameters
- Getting help: `man`, `--help`, `help`
- Bash features: history, variables, aliases, editing
- Redirection (`>`, `>>`)
- Editing with `vi` (minimum survival set)
- Interpreters: why Bash is one

Section 1

Introduction

What is Bash?

- **Bash** is a *shell*
- A shell is a program that:
  - reads what you type
  - runs other programs
  - shows results

Real-life analogy:
A receptionist who routes your request to the right department.

Why use the command line?

- Fast for repeated tasks
- Precise and scriptable
- Works even without a graphical desktop

Analogy:
GUI is like “tap a button”. CLI is like “tell exact instructions”.

Think About This

- A GUI often asks “Are you sure?”
- A CLI often assumes you mean it

What habits make CLI safer?

Section 1: Summary

- Bash is a shell (a middleman)
- You use it to run programs by typing text commands
- CLI is powerful, so safe habits matter

Section 2

Entering Linux Commands

The prompt: “who + where”

A prompt often shows:

- your **username**
- your **computer name**
- your **current folder**
  Additionally:
- `~` Home folder of user
  Example:

```
user@pc:~/projects$
```

Demo/Lab

- Access the machine
- Prompt and home

Simple Linux commands

Try these (safe):

```
- `pwd` → where am I?
- `whoami` → current user
- `hostname` → machine name
```

Demo/Lab

- Lets go through those commands
- There are no inputs other than commands themselves

Simple Linux commands

Meaning:

- `ls .` → what's here?
- `cd ..` → go up one folder

Additionally:

- `.` current folder
- `..` one folder up

Demo/Lab

- Lets go through those commands
- You can supply additional inputs

Real-life analogy: folders

- A **folder** is like a **drawer**
- `pwd` tells you which drawer you opened
- `ls` shows what is inside that drawer
- `cd` opens a different drawer

Command shape

Now that we have seen, Most commands look like:

```
command [options] [parameters]
```

- command: the program name
- options: switches that change behavior
- parameters: targets (files/folders/values)

Example: command + parameter

```
ls
```

Lists current folder.

```
ls /etc
```

Lists the folder `/etc`.

Analogy:
“Show me this shelf” vs “show me that shelf”.

Options (short and long)

Options often start with:

- `-` short form (example `-l`)
- `--` long form (example `--help`)

Example:

```
ls -l
ls --help
```

Common options you will see

Many commands reuse meanings:

- `-h` / `--help` → show help
- `-i` / `--interactive` → ask before action
- `-f` / `--force` → do it without asking
- `-r` / `--recursive` → include subfolders

Parameters: multiple targets

You can pass multiple parameters:

```
ls /home /etc /var/log
```

One command, many targets.

Analogy:
One shopping trip, multiple items.

Reflect

- Why do you think short options exist?
- When are long options safer for beginners?

Section 2: Summary

- Prompt gives context (who/where)
- Beginner commands: `pwd`, `ls`, `cd`, `whoami`, `hostname`
- Command format: command + options + parameters

Section 3

Forms of Linux Help

`man` pages (manual pages)

```
man ls
```

- Full manual for a command
- Usually very detailed

Analogy:

The official instruction book.

Navigating `man` (survival keys)

Inside `man`:

- `Space` → next page
- `b` → previous page
- `/word` → search
- `n` → next match
- `q` → quit

Quick help: `--help`

Many commands support:

```
rmdir --help
```

Often:

- shorter than `man`
- shows common options + usage

Shell built-in help: `help`

Some commands are built into Bash:

```
help
help cd
```

- `help` is for shell built-ins
- `man` is for many system programs

Demo/Lab

- Using help

Think Deeper

You have a problem. What order is best?

1. `command --help`
2. `man command`
3. search inside man (`/keyword`)

Why?

Section 3 Summary

- `--help` → quick hints
- `man` → full reference
- `help` → for Bash built-ins

Section 4

Bash Features

4.1 History: recall old commands

```
history
```

Why it helps:

- re-run long commands
- learn from your past commands

Analogy:
Your “sent messages” list.

Think About This

If you can recall commands easily:

- how does that change how you learn?
- can it also hide mistakes (re-running wrong command)?

4.2 Variables: stored values

Example:

```
echo $USER
```

- `$USER` expands to your username

Analogy:
A labeled box: label = name, inside = value.

Setting a variable (simple demo)

```
COURSE="Intro to Computers"
echo $COURSE
```

Explain:

- `NAME="value"` assigns
- `$NAME` reads the value

Demo/Lab

- Using history, variables and echo

4.3 Aliases: command nicknames

```
alias ll='ls -l'
ll
```

- `ll` becomes a shortcut for `ls -l`

Analogy:

Saving a phone number under a short contact name.

4.4 Command-line editing

Bash lets you edit the current line:

- fix typos without retyping
- move cursor left/right

Analogy:
Editing a message before you send it.

4.5 Redirection: save output to a file

Normally output goes to the screen.

```
ls > list.txt
```

- `>` writes output to `list.txt`
- overwrites if file already exists

Analogy:
Instead of reading notes aloud, you write them into a notebook.

Append instead of overwrite

```
echo "first"  > notes.txt
echo "second" >> notes.txt
```

- `>` overwrite
- `>>` append

Analogy:

Replace a page vs add a new line at the end.

Reflect

- Why is `>` powerful?
- What can go wrong with `>` if you forget the filename?

4.6 Other useful features (quick taste)

Depending on setup, you may have:

- Tab completion (type part of a name, press `Tab`)
- Wildcards like `*` (matches many names)

Example:

```
ls *.txt
```

Demo/Lab

- Go through the features

Section 4 Summary

- History reduces typing
- Variables store values (`$USER`)
- Aliases make shortcuts
- Editing fixes typos quickly
- Redirection saves output (`>`, `>>`)

Section 5

Tailoring Our Environment

Tailoring = make the shell “yours”

Common customizations:

- prompt look
- useful aliases
- environment variables

Analogy:
Arranging your desk so common tools are within reach.

Practical starter kit (safe)

```
alias ll='ls -l'
alias la='ls -a'
```

What they do:

- `-l` shows details
- `-a` shows hidden files

Demo/Lab

- Using alias

Think About This

- What 2 aliases would help you every day?
- When can too many shortcuts become confusing?

Section 5 Summary

- Tailoring means customizing the shell
- Aliases and variables are common tools for that

Section 6

Editing Files

Why we need a text editor

Many tasks need editing text:

- config files
- notes
- scripts

In Linux, editors often run **inside the terminal**.

Analogy:
Terminal editor = writing inside a notebook, not a word processor.

nano: beginner-friendly editor

- Simple
- No modes
- Commands shown on screen
- Installed on most Linux systems

Analogy:
nano is like **Notepad** inside the terminal.

Opening nano

```
nano notes.txt
```

What happens:

- File opens if it exists
- New file created if it doesn’t

Typing text in nano

- Just start typing
- No special mode
- Arrow keys work normally

Analogy:

Like typing in a simple text editor.

nano shortcuts (must-know)

Shortcuts use `Ctrl` key
(`^` on screen means Ctrl)

- `Ctrl + O` → save (Write Out)
- `Ctrl + X` → exit
- `Ctrl + K` → cut line
- `Ctrl + U` → paste
- `Ctrl + W` → search

Demo/Lab

1. Open file:

```
nano hello.txt
```

2. Type text

Demo/Lab

3. Save:

```
Ctrl + O  → Enter
```

4. Exit:

```
Ctrl + X
```

Think About This

Why does nano show shortcuts on screen?

- Good for beginners?
- Good for memory?
- Any downsides?

vi exists (just a heads-up)

- vi is very powerful editor
- Uses **modes**
- Available almost everywhere
- `man` pages use similar keys

But:

- Steeper learning curve
- Easy to get “stuck” at first

Section 6 Summary

- Text editors are needed in the terminal
- nano is simple and beginner-friendly
- Shortcuts are visible and easy
- vi exists and you’ll meet it later

Section 7

Interpreters

Compiled vs interpreted

- **Compiled**: translate whole program → then run
- **Interpreted**: read and run step-by-step

Analogy:

- Compiled = translate a whole book first
- Interpreted = translate each sentence while reading

Interpreters in shells

Shell usage is interactive:

- type a command
- get output now
- decide next command

So step-by-step interpretation is natural.

The Bash interpreter (concept)

When you press Enter, Bash:

- reads the line
- expands variables/aliases
- runs the command
- prints output/errors
- returns to the prompt

Think Deeper

If Bash expands aliases and variables:

- how does that boost productivity?
- how can it cause surprises?

Section 7 Summary

- Interpreters run step-by-step
- Shells are interactive, so interpreters fit well
- Bash interprets your command line before executing

Big takeaways

- Bash is a shell (command interpreter)
- Command format: command + options + parameters
- Help: `--help`, `man`, `help`
- Features: history, variables, aliases, editing, redirection
- nano basics
- Bash behaves like an interpreter

Exit ticket (2 minutes)

1. What does `pwd` show?
2. Option vs parameter: what is the difference?
3. How do you quit `man`?
4. What does `>` do?

Thank you!

Any questions?

Thejesh GN