Introduction to Git

Part 1 - Theory

- What version control is
- Why Git
- Git states
- Install and setup
- First local workflow

Part 2 - Lab

- Clone a remote repo
- Make commits
- Push changes
- Pull changes
- Merge changes
- Small team exercise

Section 1 — Why version control?

We will learn:

- What problem Git solves
- Why manual file copies fail
- Why teams need history

Version control

Version control records changes to files over time.

It helps you:

- Go back to an older version
- Compare changes
- Know who changed what
- Work with other people safely

Real-life analogy

Without version control, many people do this:

- `notes_v1.txt`
- `notes_v2.txt`
- `notes_final.txt`
- `notes_final_final.txt`

Think About This

What problems happen when two people edit the same file and save different copies?

Exercise

Create a text file and save 3 versions manually.

Then discuss:

- Which one is latest?
- What changed?
- Who changed it?

Think

Have you ever lost work because you overwrote a file?

Any questions?

Section 2 — Types of version control

We will see:

- local
- centralized
- distributed

Local version control

Everything stays on one computer.

Good

Simple

Bad

If the laptop dies, history may die too.

Centralized version control

One main server stores the official history.

Good

Easy central control

Bad

If the server is down, work stops.

Distributed version control

Everyone can have the full repository.

Good

- fast
- offline work
- many backups

Reflect

Why is "many full copies" safer than "one central copy"?

Any questions?

Section 3 — Git

We will learn:

- What is Git
- snapshots
- local speed
- integrity
- three states

What is Git

- Git is a free and open source *distributed* version control system designed to handle everything from small to very large projects with speed and efficiency. Git Project Site.
- It started as a tool needed to manage Linux Kernel

Git stores snapshots

- Many older systems think in terms of file differences.
- Git thinks in snapshots.

Analogy

- Git takes a picture of the project at each commit.

Most Git work is local

Git can:

- show history fast
- compare versions fast
- let you commit offline

Git has integrity

- Git uses hashes to identify content.
- Even a small change creates a different hash.
- That helps Git detect corruption and track exact content.

Key term: Repository

A **repository** (or **repo**) is a folder that Git is tracking.

It stores:

- your project files
- the full history of every change
- who changed what and when

> When you start using Git on a folder, that folder becomes a repository.

Key term: Commit

A **commit** is a saved snapshot of your project at a specific moment.

Each commit has:

- a message describing what changed
- a unique ID (hash)
- the author and timestamp

> Think of a commit as pressing "save" in a video game — you can always return to this point.

Tracked vs. Untracked files

When Git watches a folder, files are either:

- **Untracked** — Git sees the file but is not recording changes to it yet.
- **Tracked** — Git is actively watching this file and recording its history.

> A brand new file starts as **untracked**. You use `git add` to tell Git to start tracking it.

The three states

Files can be:

- Modified
- Staged
- Committed

Analogy for three states

- Working directory → your desk
- Staging area → ready-to-submit folder
- Repository → permanent archive

> Staging lets you choose which changes go into the next commit.

```
Edit -> Stage -> Commit
```

Any questions?

Section 4 — Install and setup

We will learn:

- Install Git
- Set name and email
- Check config

Install Git

Ubuntu / Debian

```
sudo apt install git
```

Windows / Mac

Use the Git installer or system tools.

First-time setup

```
git config --global user.name "Thejesh GN"
git config --global user.email "i@thejeshgn.com"
```

Check config

```
git config --list
```

Live exercise — check

Everyone should:

```
git --version
git config --global user.name
git config --global user.email
```

Check:

- Is Git installed?
- Is your identity set?

Summary

You learned:

- What version control is
- Why Git matters
- Git's three states
- How to install and set it up

What is Git Remote

A remote is another copy of a Git repository that lives on another computer or server.

It lets you share and synchronize your work with others.

Examples:

- Codeberg
- GitLab
- GitHub

Create Online Account

- Go to https://codeberg.org
- Create an account using your official email ID
- Remember the username and password, we will use it in the lab later

Any questions?

Part 2 - Lab

Section 1 — Working with a real remote repository

We will use this class repository:

`https://codeberg.org/thejeshgn/our_class_work.git`

We will use it for:

- clone
- commit
- push
- pull
- merge

Two ways to start a repo

Starting from scratch (your own new project):

```
git init
```

Creates a new empty repo in the current folder.

Joining an existing project (what we do today):

```
git clone <url>
```

Downloads the full repo from a server.

> Today we use `clone` because we are joining a project that already exists.

Section 2 — Clone the class repository

We will learn:

- What cloning means
- How to get a local copy
- What appears after clone

Clone

```
git clone https://codeberg.org/thejeshgn/our_class_work.git
```

This creates a local folder with:

- project files
- full Git history
- remote called `origin`

> **`origin`** is the name Git automatically gives to the server you cloned from. It is just a shortcut name. You will see it in push and pull commands.

Go into the repo

```
cd our_class_work
git remote -v
git status
```

Live exercise — clone and inspect

Run:

```
git clone https://codeberg.org/thejeshgn/our_class_work.git
cd our_class_work
git status
git remote -v
```

Questions:

- What is the repo folder name?
- What is the remote name?
- Is the working tree clean?

What `git status` looks like

After cloning, a clean repo:

```
On branch main
nothing to commit, working tree clean
```

Section 3 — Make your first class change

We will learn:

- Create a file
- Stage it
- Commit it

Create a class work file

Example:

```
mkdir students
nano students/name.txt
# add your NAME to the file
```

What `git status` looks like

After creating a new file:

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    students/name.txt
```

Stage

```
git add students/name.txt
```

What `git status` looks like

After `git add`:

```
Changes to be committed:
    new file:   students/name.txt
```

Commit

```
git commit -m "Add NAME class file"
```

What `git status` looks like?

Check history

```
git log --oneline -5
```

Live exercise — first commit

Steps:

```
mkdir students
nano students/name.txt
# add your NAME to the file, save and exit
git add students/name.txt
git commit -m "Add NAME class file"
git log --graph --oneline -5
```

Check:

- Did the commit happen?
- Can you see your commit at the top?

Section 4 — Push to the remote

We will learn:

- What push does
- How local and remote history connect

Push

If the default branch is already set up:

```
git push
```

Or explicitly:

```
git push origin main
```

What push means

Push sends your local commits to the remote server.

It does **not** send:

- unstaged changes
- uncommitted changes

Only commits are pushed.

Live exercise — push

Try:

```
git push
```

If that fails, inspect:

```
git branch
git remote -v
```

Then try the explicit command for your branch.

If you see a **permission** or **authentication** error, your instructor will help you set up access to the remote.

Discussion:
Why does Git require a commit before push?

Section 5 — Pull and fetch

We will learn:

- fetch
- pull
- update local copy

Fetch

```
git fetch
```

Gets remote changes, but does not merge them into your current branch.

Pull

```
git pull
```

Gets remote changes and tries to merge them.

Simple classroom use

After others push to the class repo:

```
git pull
ls students
```

You should see more student files.

Live exercise — pull others' work

After a few classmates push:

```
git pull
git log --graph --oneline -10
ls students
```

Questions:

- What new files appeared?
- Which commits came from others?

Section 6 — Merge

We will learn:

- what merging means
- simple merge flow
- conflict idea

Key term: Branch

A **branch** is a separate line of work in the same repository.

- The default branch is usually called `main` or `master`
- You can create a new branch to try something without affecting the main version
- When you are done, you **merge** the branch back

> Think of it like a draft copy. You write in the draft, and when it is ready, you bring the changes into the final version.

What is a merge?

A merge combines work from different lines of history.

Simple case:

- your branch has one change
- another branch has another change
- Git combines them

Merge demo

Create a new branch:

```
git checkout -b demo-branch
```

Edit a file and commit:

```
echo "branch line" >> demo.txt
git add demo.txt
git commit -m "Add line from demo branch"
```

Go back and merge:

```
git checkout main
git merge demo-branch
```

If your repo uses `master`, replace `main` with `master`.

What a merge conflict is

If two branches change the same part of the same file, Git may need help.

Then Git stops and asks you to resolve the conflict.

Live demo — local merge practice

Use a local-only merge first.

```
git checkout -b branch-a
echo "A" >> merge_practice.txt
git add merge_practice.txt
git commit -m "Add merge practice file"
git checkout main
git merge branch-a
```

Goal: See a successful simple merge.

Live demo — Conflict demo

1. Create two branches from the same point.
2. Change the same line in the same file on both branches.
3. Merge them.

Then inspect:

- conflict markers
- resolved file
- final commit

Good commit messages

Use short, clear messages.

Good:

- `Add Anu class file`
- `Update intro notes`
- `Fix typo in README`

Bad:

- `changes`
- `done`
- `update`

Think Deeper 🤔

Why is `Add NAME class file` better than `done`?

`.gitignore`

Sometimes you do not want Git to track certain files — like log files, temp files, or system files.

Create a file called `.gitignore` in your repo:

```
*.log
*.tmp
.DS_Store
```

Git will ignore any file matching those patterns.

> You do not need this today, but you will want it in your own projects.

Glossary

Term	Meaning
Repository (repo)	A folder Git is tracking, with full history
Commit	A saved snapshot with a message and ID
Branch	A separate line of work in the same repo
Remote	A copy of the repo on another server

Term	Meaning
Clone	Download a full copy of a remote repo
Origin	The default name for the remote you cloned from
Tracked	A file Git is watching and recording
Untracked	A file Git sees but is not yet recording
Staging area	Where you prepare changes before committing

Quick command sheet

```
git clone https://codeberg.org/thejeshgn/our_class_work.git
cd our_class_work
git status
mkdir students
nano students/name.txt
git add students/name.txt
git commit -m "Add NAME class file"
git push
git pull
git log --graph --oneline
```

Think

1. What is the difference between `add` and `commit`?
2. What is the difference between `fetch` and `pull`?
3. Why does Git need a merge?

Any questions?

Thank you!

Any questions?

Thejesh GN