Init & first commit
```
git init
git add .
git commit -m "message"
```

Branching
```
git checkout -b my-branch      # create + switch
git switch my-branch           # (newer form)
```

Remotes
```
git remote add origin <url>
git push -u origin main
git pull origin main
```

Status / diff / history
```
git status
git diff
git log --oneline --graph --decorate --all
```

Merging & conflicts
```
git merge <branch>
# edit files to resolve markers (<<<<<<< ======= >>>>>>>)
git add <resolved-file>
git commit
```

Undo / safety
```
git restore <file>            # discard unstaged changes to file
git restore --staged <file>    # unstage
git revert <commit>           # make a new commit that undoes that commit
```

```
git stash                    # shelve local WIP
```

A **normal repo** = working tree (your files) + `.git` database.
A **bare repo** = only the `.git` database.
Collaboration uses the bare r

Demo

```
cd ~
rm -rf git-class-lab
mkdir -p git-class-lab && cd git-class-lab

# We'll simulate a remote and two collaborators (alice and bob)
git init --bare remote.git

# --- Alice creates the project -----------------------------------------
git clone remote.git alice
cd alice
git config user.name "Alice"
git config user.email "alice@example.com"

# 1) init project; first commit
echo "# Demo Project" > README.md
echo "print('hello world')" > app.py
git add .
git commit -m "init: README and app.py"
git branch -M main
git remote add origin "$(cd ..; pwd)/remote.git"
git push -u origin main

# 2) feature branch, some commits
git checkout -b feature/greet
echo "def greet(name):\n    return f'Hello, {name}!'" >> app.py
git add app.py
git commit -m "feat: add greet(name)"
```

```
# 3) make a change on the SAME line on main later to force a conflict
#    We'll prep Alice's feature, but not merge yet.
git push -u origin feature/greet

# --- Bob clones and works on main ----------------------------------------
cd ..
git clone remote.git bob
cd bob
git config user.name "Bob"
git config user.email "bob@example.com"

# 4) Bob edits the same place in app.py differently -> future conflict
echo "# Bob's utilities" > utils.py
# overwrite app.py's last line to a different version
printf "print('hello world')\n# Bob tweaks greeting baseline\n" > app.py
git add .
git commit -m "chore: add utils and tweak app.py baseline"
git push origin main

# --- Alice tries to merge her feature ------------------------------------
cd ../alice
# Make a conflicting change on feature branch to the same area
printf "print('hello world')\n# Alice adds greet feature\n\ndef greet(name):\n    return f'Hi,
{name}!'\n" > app.py
git add app.py
git commit -m "feat: implement greet with 'Hi'"
git push

# Attempt to merge feature into main locally
git checkout main
git pull --rebase origin main
git merge --no-ff feature/greet || true   # expect conflict, continue script

# Show status so students see the conflict
echo "------ GIT STATUS (expect CONFLICT) ------"
git status

# 5) Resolve the conflict (keep both ideas nicely)
#    Find conflict markers and resolve by writing a final version:
cat > app.py <<'EOF'
print('hello world')
# Merged greeting: combine ideas from Alice and Bob
```

```
def greet(name):
    # Alice said "Hi", let's keep that but also nod to Bob's tweak
    return f'Hi, {name}! (welcome)'
EOF

git add app.py
git commit -m "merge: resolve conflict in app.py keeping 'Hi' variant"
git log --oneline --graph --decorate -n 8

# 6) Push merged main, Bob pulls and sees result
git push origin main

cd ../bob
git pull origin main
python3 - <<'PY'
from pathlib import Path
print("app.py contents after pull:\n")
print(Path("app.py").read_text())
PY

echo “Done. Lab folder at: $(cd ..; pwd)/git-class-lab"
```

Demo’s Details:

Clean start / workspace

- `cd ~`
  Go to your home directory.

- `rm -rf git-class-lab`
  Delete any old copy of the lab folder (if it exists).

- `mkdir -p git-class-lab && cd git-class-lab`
  Create the lab folder (no error if it exists) and enter it.

Make a central “remote” (like GitHub), but local

- `git init --bare remote.git`
  Create a **bare** repo (no working files, only Git database). This is the shared hub

everyone pushes to/pulls from.

Alice sets up her repo and first commit

- `git clone remote.git alice`
  Make Alice's working copy from the central remote.

- `cd alice`
  Enter Alice's repo.

- `git config user.name "Alice"`
  Set commit author name for this repo.

- `git config user.email "alice@example.com"`
  Set commit author email for this repo.

- `echo "# Demo Project" > README.md`
  Create README with a title.

- `echo "print('hello world')" > app.py`
  Create a tiny Python app.

- `git add .`
  Stage all new files for the next commit.

- `git commit -m "init: README and app.py"`
  Make the first commit with a message.

- `git branch -M main`
  Rename current branch to `main`.

- `git remote add origin "$(cd ..; pwd)/remote.git"`
  (Redundant after a clone.) Alice already has `origin` set from `git clone`. This line attempts to add it again and may error ("remote origin already exists"). Safe to **remove/skip**.

- `git push -u origin main`
  Push the `main` branch to the central repo and set upstream tracking.

Alice creates a feature branch and commits work

- `git checkout -b feature/greet`
  Create and switch to `feature/greet`.

- `echo "def greet(name):\n return f'Hello, {name}!'" >> app.py`
  Append a `greet()` function to `app.py`.

- `git add app.py`
  Stage the change.

- `git commit -m "feat: add greet(name)"`
  Commit the feature work.

- `git push -u origin feature/greet`
  Publish the feature branch to the central repo and track it.

Bob clones and works on `main`

- `cd ..`
  Go back to the lab folder.

- `git clone remote.git bob`
  Make Bob’s working copy from the same central repo.

- `cd bob`
  Enter Bob’s repo.

- `git config user.name "Bob"`
  Set Bob’s name for commits.

- `git config user.email "bob@example.com"`
  Set Bob’s email for commits.

- `echo "# Bob's utilities" > utils.py`
  Create a new file for Bob’s utilities.

- `printf "print('hello world')\n# Bob tweaks greeting baseline\n" > app.py`
  Overwrite `app.py` with a slightly different version—this sets up a **future conflict** with Alice's edits.

- `git add .`
  Stage changes.

- `git commit -m "chore: add utils and tweak app.py baseline"`
  Commit Bob's changes to `main`.

- `git push origin main`
  Push Bob's `main` to the central repo.

Alice tries to merge her feature into updated `main`

- `cd ../alice`
  Back to Alice's repo.

- `printf "print('hello world')\n# Alice adds greet feature\n\ndef greet(name):\n return f'Hi, {name}!'\n" > app.py`
  Alice rewrites `app.py` with her own variant (conflicts with Bob's change).

- `git add app.py`
  Stage the change.

- `git commit -m "feat: implement greet with 'Hi'"`
  Commit it on her feature branch.

- `git push`
  Push the feature branch updates (remote/tracking already set).

- `git checkout main`
  Switch to `main` to prepare the merge.

- `git pull --rebase origin main`
  Update local `main` with the latest from the central repo (applies your local commits on top—here there probably aren't any).

    - **Fetch** the latest commits from `origin`'s `main`
    - **Rebase** your *current branch* so your local commits are replayed **on top of** `origin/main`

- `git merge --no-ff feature/greet || true`
  Try to merge the feature into `main`. `--no-ff` forces a merge commit (useful for teaching history).
  If a conflict occurs, the `|| true` prevents the script from stopping—so the demo continues.

Show the conflict

- `echo "------ GIT STATUS (expect CONFLICT) ------"`
  Print a banner.

- `git status`
  Show conflicted files and next steps.

Resolve the conflict (choose a final version)

- The `cat > app.py <<'EOF' ... EOF` block
  Replace the conflicted file with a **manually merged** version that keeps both ideas and finalizes the function.

- `git add app.py`
  Mark the conflict as resolved by staging the fixed file.

- `git commit -m "merge: resolve conflict in app.py keeping 'Hi' variant"`
  Complete the merge with a commit message.

- `git log --oneline --graph --decorate -n 8`
  Show a compact graph of recent history so students can see the merge structure.

Share the resolved result; Bob pulls it down

- `git push origin main`
  Push the merged `main` to the central repo.

- `cd ../bob`
  Back to Bob's repo.

- `git pull origin main`
  Bob downloads the merged result.

- The `python3 - <<'PY' ... PY` block
  Simple Python snippet that prints the contents of `app.py` so the class can see the final merged code in Bob's copy.

- `echo "Done. Lab folder at: $(cd ..; pwd)/git-class-lab"`
  Friendly "all done" message showing where the lab lives on disk.

Quick notes to remember

- **Bare repo (`remote.git`)** = central hub; no files visible, just Git objects.

- **Clone** gives each collaborator a working copy with files.

- **Conflict** happens when the same lines change differently—Git asks a human to decide.

- After a **merge commit**, pushing and pulling sync everyone to the final agreed state.

- That `git remote add origin ...` in Alice's section is **unnecessary after a clone** and can be removed to avoid the "origin already exists" error.

Here's the **high-level flow** your demo runs, in plain steps:

1. **Clean workspace**
   Go to your home folder, delete any old lab, make a fresh `git-class-lab` folder.

2. **Create a central repo (the "hub")**
   `git init --bare remote.git` makes a **bare** repository—no working files, just Git's database—acting like a tiny local GitHub everyone pushes to/pulls from.

3. **Alice starts the project**

   - Clones the hub into `alice/`, sets her name/email.

   - Creates `README.md` and `app.py`, stages with `git add .`, makes the first **commit**, renames branch to **main**, and **pushes** `main` to the hub.
     (Note: the `git remote add origin ...` line is redundant after `git clone` and can be removed.)

4. **Alice opens a feature branch**

   - `git checkout -b feature/greet` creates/switches to a **feature branch**.

   - Adds a `greet()` function, commits, and **pushes** the branch so it's shared.

5. **Bob joins and works on `main`**

   - Clones the hub into `bob/`, sets his identity.

   - Adds `utils.py` and **changes the same part of `app.py`** that Alice is touching—this sets up a future **merge conflict**.

   - Commits and **pushes** to `main`.

6. **Alice advances her feature**

   - Still on `feature/greet`, she edits `app.py` in a way that will **conflict** with Bob's change and **pushes** her branch.

7. **Alice updates her `main` and tries to merge**

   - `git checkout main` then `git pull --rebase origin main` to bring local `main` up to date **without a merge commit** (clean history).

   - `git merge --no-ff feature/greet` tries to combine the feature into `main` therefore, **conflict occurs** (they changed the same lines differently).

8. **Resolve the conflict**
    - Open the conflicted file, choose/compose the final desired code, `git add` it, then `git commit` to finish the **merge**.
    - `git log --oneline --graph` shows the history with the merge.
9. **Share the resolved result**
    - Alice **pushes** the merged `main` to the hub.
10. **Bob syncs and verifies**

- Bob **pulls** the latest `main` and prints `app.py` to confirm he now has the merged, conflict-free version.