What is Git?

Simple Definition

Git is a **Version Control System** that tracks changes in your code.

Real Life Analogy

Google Docs history:

- You can go back to old versions
- You can see who changed what
- You can work together

Git does same for code.

Why Git is needed

- Save versions
- Undo mistakes
- Collaborate
- Backup
- Track history

Diagram Explanation (Draw on board)

Your Laptop (Local Repo) ----> Git

Version 1 -> Version 2 -> Version 3

---

2. What is GitHub?

Simple Definition

GitHub is a **cloud platform** that stores Git repositories online.

Analogy

- Git = Camera
- GitHub = Google Drive for photos

Why GitHub

- Online backup
- Team collaboration
- Show projects (portfolio)
- CI/CD, Issues, PRs

Diagram

Your PC (Git) <----> GitHub Server (Cloud)

---

3. Git vs GitHub

Git	GitHub
Software	Website / Cloud platform
Works locally	Works online
Tracks versions	Hosts repositories
Command line tool	Web interface
Open source	Company (Microsoft)

Interview Line:

"Git is a version control tool, GitHub is a hosting service for Git repositories."

---

4. Creating GitHub Account

Steps (Show Live):

1. Go to github.com
2. Click Sign Up
3. Username (professional)
4. Email verification
5. Profile setup

Explain:

- Public vs Private repos
- Why public for portfolio

---

5. Creating Repository on GitHub

Steps:

1. Click ➕ → New Repository
2. Repository Name: first-repo
3. Public
4. Add README
5. Create Repository

Explain:

- What is README.md
- Why README is important
- Repository URL

---

6. Git Commands

6.1 git init

Purpose:

Converts a normal folder into a Git repository.

Command:

git init

Explain:

- Creates hidden .git folder
- Git starts tracking

6.2 git status

Purpose:

Shows current state of files.

git status

Explain states:

- Untracked
- Modified
- Staged
- Committed

Diagram:

Working Area → Staging Area → Repository

6.3 git add <file>

Stage specific file:

git add index.html

Meaning:

Move file from working area to staging area.

6.4 git add .

Stage all files:

git add .

Explain:
. means current directory

Difference:

Command	Meaning
git add file	Add one file
git add .	Add all changes

6.5 git pull

Purpose:

Download latest code from GitHub to local.

git pull

Explain:

- Used when teammate pushed code
- Avoid conflicts
- Always pull before push

6.6 git push

Purpose:

Upload local commits to GitHub.

git push origin main

Explain:

- origin = remote name
- main = branch

---

7. Push Related Errors & Fixes (15 minutes)

Error 1: "rejected - fetch first"

Reason:

Remote has changes you don't have.

Fix:

git pull

git push

Error 2: Authentication Failed

Reason:

- Wrong credentials
- Token issue

Fix:

- Generate GitHub Personal Access Token
- Use token instead of password

Error 3: No upstream branch

Error:

fatal: The current branch main has no upstream branch

Fix:

git push --set-upstream origin main

Error 4: Permission denied (publickey)

Reason:

- SSH not configured

Fix:

- Use HTTPS instead of SSH
- Or setup SSH key

Error 5: Nothing to push

Reason:

- No commits

Explain need of:

git commit -m "first commit"

---

8. Complete Flow (Show Once End-to-End)

git init

git status

git add .

git commit -m "first commit"

git remote add origin <repo_url>

git push origin main

👷 **PART 1: Owner Creates a Repository**

Step 1: Create a new GitHub repository

1. Go to **github.com**
2. Click **New Repository**
3. Give it a name (example: css-collab-demo)
4. Set it to **Public**
5. Click **Create Repository**

Step 2: Connect repo with local folder

Open terminal and run:

mkdir css-collab-demo

cd css-collab-demo

git init

git remote add origin https://github.com/username/css-collab-demo.git

---

Step 3: Create HTML & CSS files

touch index.html style.css

Add basic content:

index.html

```
<!DOCTYPE html>
<html>
<head>
 <link rel="stylesheet" href="style.css">
</head>
<body>
 <h1>Hello CSS Collaboration</h1>
 <div class="box">Box</div>
</body>
</html>
```

style.css

```
body {
 font-family: Arial;
}

.box {
 width: 200px;
 height: 200px;
 background: skyblue;
}
```

Step 4: Push to GitHub

```
git add .
git commit -m "Initial HTML and CSS"
git branch -M main
git push -u origin main
```

Step 5: Share Repository URL

Owner sends the GitHub repo link to partner.

Example:
https://github.com/aniket/css-collab-demo

👷 PART 2: Cloner Clones the Repo

Step 6: Clone using git clone

In terminal:

git clone https://github.com/aniket/css-collab-demo.git

cd css-collab-demo

Now the full project is on their system.

Step 7: Cloner adds new CSS

Open style.css and add:

```
h1 {
  color: darkblue;
  text-align: center;
}

.box {
  border-radius: 20px;
  box-shadow: 0 0 10px gray;
}
```

Step 8: Cloner pushes changes

git add .

git commit -m "Added styles and effects"

git push

👷 PART 3: Owner Pulls Updates

Step 9: Owner pulls latest code

git pull origin main

Step 10: Run and See Changes

Open index.html in browser →
You'll now see:

- Colored heading
- Rounded box
- Shadow effect

🎯 **What Students Learn**

Command	Purpose
git clone	Copy project from GitHub
git push	Send changes to GitHub
git pull	Receive teammate's changes
GitHub Repo	Central shared codebase

What is .gitignore?

.gitignore is a file that tells Git:

"Do NOT track these files/folders.
Do NOT upload them to GitHub."

Used for:

- node_modules
- build files
- .env (passwords, API keys)
- OS junk files
- IDE settings

🛠 **How to Create .gitignore**

In your project folder:

touch .gitignore

Open it in VS Code and write:

node_modules/

.env

.DS_Store

dist/

🧠 How It Works

If a file name matches .gitignore, Git:

- Will not show it in git status
- Will not commit it
- Will not push it

🧪 Example for HTML + CSS Project

Folder:

project/

├— index.html

├— style.css

├— .gitignore

├— notes.txt

├— secrets.txt

.gitignore

secrets.txt

notes.txt

Now:

git status

These files will be hidden from Git.

⚠ Important Rule (Common Mistake)

If file is **already committed**, .gitignore won't work.

Fix:

git rm --cached secrets.txt

git commit -m "Remove secret file from tracking"

🧩 Folder Ignore Example

Ignore all images:

images/

Ignore all .log files:

*.log

Ignore all .txt except one:

*.txt

!important.txt

🧑‍🏫 Teaching Analogy

Tell students:

.gitignore is like a “No Entry List” for Git.
Files written there are invisible to version control.

Fast-forward Merge

What it means

A *fast-forward* happens when the target branch has **no new commits** since the feature branch was created.
So Git simply **moves the pointer forward** — no new merge commit is created.

Situation

A---B---C (main)

\

D---E (feature)

If main has not changed after C, and you merge feature:

A---B---C---D---E (main)

Command

git checkout main

git merge feature

Key Points

- No extra merge commit.
- Linear history.
- Clean and simple.
- Only possible when branches never diverged.

When used

- Small features
- Solo development
- Clean history preferred

---

2) Three-way Merge

What it means

A *three-way merge* happens when **both branches have new commits**.
Git uses **three snapshots**:

1. Common ancestor
2. Current branch HEAD
3. Merging branch HEAD

Then it creates a **new merge commit**.

Situation

```
    D---E  (main)
   /
A---B---C
   \
    F---G  (feature)
```

After merge:

```
    D---E
   /    \
A---B---C      M   (merge commit)
   \    /
    F---G
```

Command

git checkout main

git merge feature

(automatically becomes three-way when history diverges)

Key Points

- Creates a merge commit.
- Preserves full branch history.
- Required when branches diverge.
- Can cause merge conflicts.

When used

- Team collaboration
- Parallel development
- Long-running branches

Rebasing in Git — Concept, Pros & Cons (with Examples)

Since you're a software engineer and already comfortable with Git, I'll explain this in a clean, interview-ready + teaching style.

---

1. What is Rebasing?

Rebasing means:

Moving your branch's base to another commit, and *replaying* your commits on top of it.

In simple words:
Instead of merging two branches and keeping a messy history, rebase rewrites history to make it look linear.

1.1 Visual Concept

Suppose history is:

main: A --- B --- C

\

feature: D --- E

Now main has new commits F and G:

main: A --- B --- C --- F --- G

\

feature: D --- E

If you **rebase feature onto main**:

git checkout feature

git rebase main

Result becomes:

main: A --- B --- C --- F --- G

\

feature: D' --- E'

Notice:

- D and E are **replayed** as D' and E'
- Commit IDs change (history is rewritten)
- Graph becomes linear

---

2. Why Rebase is Used

2.1 To Keep History Clean

Merge creates this:

D --- E

/ \

A --- B --- C --- F --- G

Rebase creates this:

A --- B --- C --- F --- G --- D' --- E'

Much easier to:

- Read logs
- Bisect bugs
- Understand timeline

---

3. Rebase Example (Real Commands)

Step 1: Create branches

git checkout -b feature

make commits

git commit -m "Add login UI"

git commit -m "Fix validation"

Meanwhile main advances:

git checkout main

git commit -m "Update API"

Step 2: Rebase

git checkout feature

git rebase main

Now your feature commits sit on top of latest main.

---

4. Interactive Rebase (Important in Interviews)

git rebase -i HEAD~3

Allows you to:

- Squash commits
- Reword commit messages
- Drop commits
- Reorder commits

Example:

pick a1 Add button

pick b2 Fix typo

pick c3 Improve style

Change to:

pick a1 Add button

squash b2 Fix typo

squash c3 Improve style

Becomes one clean commit.

---

5. Pros of Rebasing

✅ **1. Clean, Linear History**

No unnecessary merge commits.

✅ **2. Easier Debugging**

git bisect works better with linear history.

✅ **3. Professional Commit Structure**

Perfect for PR before merging.

✅ **4. Better Code Review**

Small logical commits instead of many noisy ones.

---

6. Cons of Rebasing

❌ **1. Rewrites History (Dangerous on Shared Branches)**

Never rebase a branch that others have pulled:

git rebase main

git push --force # can break teammates' repos

❌ **2. Commit IDs Change**

Old commits:

D (abc123)

After rebase:

D' (xyz789)

Any references become invalid.

❌ **3. Conflict Resolution Can Repeat**

If many commits touch same lines, you may resolve conflicts multiple times.

---

7. Rebase vs Merge (Interview Table)

Feature	**Rebase**	**Merge**
History	Linear	Branchy
Commit IDs	Rewritten	Preserved
Safe on shared branches	❌ No	✅ Yes
Conflict resolution	Replayed per commit	Once
Best for	Feature cleanup	Integrating public branches

---

8. Golden Rules

Rule 1: Never Rebase Public Branches

Public = main, develop, shared feature branches.

Rule 2: Rebase Your Own Local Feature Branch Only

Best workflow:

git checkout feature

git fetch origin

git rebase origin/main

Rule 3: Squash Before PR

git rebase -i origin/main

---

9. One-Line Definition (For Exams / Interviews)

Rebasing is a Git operation that reapplies a sequence of commits onto a new base commit, rewriting history to create a clean and linear project timeline.