# **📋 Git Notes Summary**

# **✅ 1. Git Setup**

git config --global user.name "Your Name"

git config --global user.email "you@example.com"

* Sets your identity for all your Git commits.

## **📁 2. Initialize a Git Repo**

git init

* Starts a new Git repository in your folder.

## **📄 3. Tracking Files**

* Create a file (e.g. hello.txt)

Stage it:  
  
 git add hello.txt

Commit it:  
  
 git commit -m "Add hello.txt"

## **🧠 4. Git Areas**

| **Area** | **Purpose** |
| --- | --- |
| Working Directory | Where you write/edit your files |
| Staging Area (Index) | Files marked for the next commit |
| Git Repository (HEAD) | Latest saved snapshot (commits) |

## **↻ 5. Re-staging After Changes**

* If you change a file after staging it, you must git add it again before committing.

## **❌ 6. Unstage or Restore**

Unstage a file:  
  
 git reset <file>

Restore file to last commit (⚠ discards changes):  
  
 git restore <file>

## **🔍 7. Check Status**

git status

* Shows file states: staged, modified, untracked, etc.

## **📊 8. See Changes with git diff**

| **Command** | **Compares** |
| --- | --- |
| git diff | Working directory ↔ Staging area |
| git diff --cached | Staging area ↔ Last commit |

You're now comfortable with creating, staging, restoring, and diffing files — the core of how Git tracks your work. ✅

**📘 Git Branches & Merging – Notes**

**🌱 What is a Git Branch?**

* A branch is a lightweight pointer to a commit.
* It lets you work on new features or fixes without affecting the main code (usually main or master).
* Common pattern: create a new branch, work on it, then merge it back.

**Common Commands:**

* git branch → List all branches
* git branch <name> → Create a new branch
* git switch <name> → Switch to an existing branch
* git switch -c <name> → Create and switch to a new branch

**🔁 Merging in Git**

Merging brings changes from one branch into another.  
 Run the merge command from the branch **you want to merge into**:

* git merge <branch-name>

**✅ Fast-Forward Merge**

* Happens when the current branch has no new commits since the two branches split.
* Git simply moves the pointer forward — no new commit is created.

**Example:**

* Before merge:  
  + main: A → B
  + feature: B → C → D
* After git merge feature:  
  + main: A → B → C → D

Result: A straight-line history, no merge commit.

**🔀 3-Way Merge**

* Happens when both branches have new commits.
* Git compares:  
  + The common ancestor
  + Your current branch
  + The branch you are merging

Git then creates a new **merge commit** that combines both changes.

**Example:**

* Common ancestor: B
* main: B → C → D
* feature: B → E → F
* After merge: B → C → D → M (merge commit)

**⚠️ Merge Conflicts**

* Occur when the **same part** of a file is changed in both branches.
* Git stops the merge and asks you to resolve manually.

**Steps to resolve conflicts:**

1. Open the conflicting files and edit them.
2. Save your changes.
3. Run: git add <filename>
4. Run: git commit

To cancel a merge with conflicts:

* git merge --abort

**🧠 Git Merge Rules Summary**

| **Ancestor File State** | **Branch A** | **Branch B** | **Merge Result** |
| --- | --- | --- | --- |
| File exists | Unchanged | Unchanged | ✅ No change (file stays the same) |
| File exists | Modified | Unchanged | ✅ File is updated with Branch A's changes |
| File exists | Modified | Modified (different lines) | ✅ Git auto-merges both changes |
| File exists | Modified | Modified (same lines) | ⚠️ Conflict – manual resolution needed |
| File exists | Modified | Deleted | ⚠️ Conflict – choose to keep or delete |
| File exists | Deleted | Unchanged | ✅ File is deleted |
| File exists | Deleted | Deleted | ✅ File is deleted |
| File doesn't exist | New file added | Unchanged | ✅ File is added |
| File doesn't exist | New file added | New file added (same name, different content) | ⚠️ Conflict |
| File doesn't exist | New file added | New file added (same name, same content) | ✅ File is added |

**🧠 Best Practices**

* Use feature branches for new work.
* Always pull latest changes before merging.
* Check git status before and after a merge.
* Don’t panic during a conflict — carefully resolve.
* Use git log --graph --oneline to visualize branch history.

**🌐 Git Remote & Collaboration Notes**

## **🚀 What is a Remote Repository?**

A **remote** in Git is a version of your project hosted on the internet or another network.

* Usually it's a GitHub, GitLab, or Bitbucket repo
* You can collaborate by pushing and pulling code from this remote

## **📦 Common Remote Commands**

### **Add a Remote**

git remote add origin <remote-URL>

Example:

git remote add origin https://github.com/username/project.git

### **View Remotes**

git remote -v

Shows list of remotes with URLs.

## **📤 git push**

Sends your commits from local to remote.

git push origin main # push main branch

git push -u origin feature # push feature branch and set upstream

* -u sets the tracking relationship between local and remote branches

## **📥 git fetch**

Downloads new data (commits/branches) from the remote, but **does not merge**.

git fetch

* Use git branch -r to see fetched remote branches
* Use git fetch --all to fetch from all remotes

## **🤀 git pull**

Does git fetch + git merge

git pull origin main

* Brings in changes from GitHub into your current branch

Use with rebase for cleaner history:

git pull --rebase origin main

## **📋 git clone**

Copy a remote repo to your local machine

git clone <repo-url>

Clones all branches and sets origin remote by default

## **🌐 Remote Branches**

To see all remote branches:

git branch -r

To create a local branch that tracks a remote one:

git checkout -b feature origin/feature

## **📘 Real-Life Workflow**

1. Clone the repo:

git clone <repo-url>

1. Create a branch:

git checkout -b new-feature

1. Work and commit:

echo "work" > file.txt

git add .

git commit -m "Add new feature"

1. Push the branch:

git push -u origin new-feature

1. Teammates can fetch and check out your branch

## **🔐 Summary Table**

| **Command** | **Purpose** |
| --- | --- |
| git remote -v | List remote URLs |
| git push | Send commits to remote |
| git fetch | Download changes (no merge) |
| git pull | Download & merge changes |
| git pull --rebase | Download & rebase your changes on top |
| git clone | Copy entire repo from remote |
| git branch -r | List remote branches |

Let me know if you'd like to simulate remote collaboration with a second clone or push to GitHub!

# 📝 Git Notes – `stash`, `squash`, `tags`, `clean`, `rebase`

---

## 📦 `git stash` – Temporarily Save Work

\*\*Purpose:\*\* Save uncommitted changes without committing.

### Common Commands:

```bash

git stash # Save staged and unstaged changes

git stash -u # Include untracked files (not ignored)

git stash list # View all saved stashes

git stash pop # Reapply and delete most recent stash

git stash apply # Reapply most recent stash (keep it)

git stash drop # Remove a specific stash

git stash clear # Remove all stashes

🧹 git clean – Delete Untracked Files

Purpose: Remove files not tracked by Git (not staged or committed).

Common Commands:

bash

Copy

Edit

git clean -n # Dry run – preview what would be deleted

git clean -f # Delete untracked files

git clean -fd # Delete untracked files and directories

git clean -fx # Delete untracked AND ignored files (danger!)

🏷️ git tag – Label Specific Commits

Purpose: Create named references to specific commits (e.g., for releases).

Common Commands:

bash

Copy

Edit

git tag v1.0 # Lightweight tag

git tag -a v1.0 -m "Release v1.0" # Annotated tag with message

git tag # List all tags

git show v1.0 # Show tag details

git push origin v1.0 # Push a specific tag to remote

git push origin --tags # Push all local tags to remote

🧹 git squash – Combine Multiple Commits

Purpose: Combine several commits into a single one for a cleaner history.

How to squash commits:

bash

Copy

Edit

git rebase -i HEAD~3 # Interactively rebase the last 3 commits

Then in the editor, change:

sql

Copy

Edit

pick abc123 First commit

squash def456 Fix 1

squash ghi789 Fix 2

Save and write a new commit message.

If you already pushed, run: git push --force

🔁 git rebase – Reapply Commits on a New Base

Purpose: Rewrite commit history to apply your changes on a new base branch.

Common Commands:

bash

Copy

Edit

git rebase main # Replay your branch onto main

git rebase -i HEAD~4 # Interactively rebase last 4 commits

git rebase --abort # Cancel the rebase

git rebase --continue # After resolving conflicts

Rebase vs Merge:

Feature Merge Rebase

History Preserved Rewritten

Timeline Branched Linear

Use Case Team integrations Cleanup before PRs

✅ Pro Tips

Use git stash before switching or resetting branches

Squash before merging PRs for a clean history

Use git clean -n before any -f command

Annotated tags (-a) are best for versioning

Prefer rebase for solo work, merge for team-based workflows