📘 Git Basics – Day 2 & 3 & 4

🎯 Topic: Semantic Versioning

Semantic versioning (or *SemVer*) is a standard for version numbers in software releases. It follows the format:

MAJOR.MINOR.PATCH

Each part of the version number has a specific meaning:

🔄 Update Type	📈 Version Change	💡 Description
PATCH	`18.2.0` → `18.2.1`	Bug fixes only – no new features or breaking changes.
MINOR	`18.2.1` → `18.3.0`	New backward-compatible features added.
MAJOR	`18.3.0` → `19.0.0`	Incompatible API changes (breaking changes).

📌 Example Breakdown

If your software version is `3.4.7`:

- 3 is the **MAJOR** version.
- 4 is the **MINOR** version.
- 7 is the **PATCH** version.

🧠 Why This Matters in Git Projects:

- When you tag releases using Git (`git tag v1.2.3`), it's good practice to follow SemVer.
- It helps users and collaborators know what kind of changes they can expect from one version to another.
- Useful for **package versioning**, **release notes**, **dependency management**, etc.

🧭 Centralized vs Decentralized Version Control Systems (VCS)

1️⃣ Centralized Version Control System (CVCS)

In a **centralized** system:

- **One central server** stores the entire project history.
- All developers **pull (download)** the latest version from this central server.
- Developers **commit (upload)** changes back to that same server.
- If the server goes down, no one can collaborate or retrieve the history.

🔁 Example Tools:

- Subversion (SVN)
- Perforce

📌 Key Characteristics:

✅ Pros	⚠️ Cons
Simple to set up and understand	Single point of failure
Easier to manage in small teams	Must be online to commit or view history
One source of truth (central server)	Slower due to constant communication with server

---

2️⃣ Decentralized Version Control System (DVCS)

In a **decentralized** system:

- Every developer's machine has a **full copy** of the project — including **all history**.
- Developers can **work offline**, make commits, and experiment locally.
- Changes are **pushed** to or **pulled** from a **remote repository** (like GitHub) only when ready.
- There's no single point of failure — any copy can restore the entire project.

🔁 Example Tools:

- **Git** ✅ (most popular)
- Mercurial
- Fossil

📌 Key Characteristics:

✅ Pros	⚠️ Cons
Full local history — can work offline	Slightly steeper learning curve
Better performance for local operations	Potential merge conflicts in teams
No single point of failure	Requires occasional syncing

🤔 Why Git?

> "**Why should we use Git?**"

Git is the most popular **decentralized version control system**. Here's why:

- ✅ Each team member has **full access to the project history**, so they can **work independently**.
- ✅ Team members can **share updates** with others easily by **pushing to or pulling from** a shared repository (like GitHub).
- ✅ Git supports **branching and merging**, making parallel development (e.g., multiple features or bugfixes) simple and safe.
- ✅ With Git, collaboration becomes smooth, traceable, and **reversible** (you can go back in history if needed).

💡 Real-World Benefit:

If you and your team store everything in Git, all members can:

- Access the **latest code version** anytime.
- Collaborate effectively without overwriting each other's work.
- Track **who changed what, when, and why**.

Fantastic! You've collected a ton of valuable Git commands and concepts — now let's turn this into a well-organized, readable reference guide.

I'll structure it into logical sections, clean up the explanations, and preserve all your terminal commands, examples, and notes. Here's Part 1 of your **Git Reference Guide**:

📦 Cloning & Initial Setup

🔄 Clone a Remote Repository

git clone <repo-url> <directory-name>

Clone a repository into a specific folder.

🔍 Inspect Git Internals

ls -la .git/

List the contents of the hidden `.git/` directory to view internal Git files (history, config, etc.).

⚙️ Git Configuration

🌐 Global vs Local Config

- **Global config file**: `~/.gitconfig`
- **Local config file** (per-repo): `.git/config`

🛠️ Common Config Commands

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

git config --global user.email "your@email.com"

Set your global identity (stored in `~/.gitconfig`).

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

git config --local user.email "your@email.com"

Set your identity for just this repository (stored in `.git/config`).

git config --list

View all Git configuration settings.

📄 File Lifecycle in Git

🔍 Status

git status

Check file status: untracked, modified, staged, etc.

➕ Add / Stage Files

git add <filename>

git add . # stage all changes

git add --help # explore more options

Staging = preparing files for commit.

✅ Commit

git commit -m "C1: main Add hello.md"

Smart commit message convention: `Commit#:BranchName Action FileName`.

git commit --amend -m "C2: main add amend.md"

Change last commit message or content without creating a new commit.
⚠️ *Amend rewrites history — avoid if others have already pulled the commit.*

🌲 Viewing History

```
git log                # full history
git log -p             # show diffs per commit
git log -p <hash>      # show diff for a specific commit
git log -p -2          # last 2 commits with patch
git log --stat         # changes + summary
git log --pretty=oneline
git log --pretty=format:"%h - %an, %ar : %s"
```

🌿 Branching

📄 Manage Branches

```
git branch             # list local branches
git branch -r          # list remote branches
git branch <name>      # create new branch
git checkout <name>    # switch branch
git checkout -b <name>  # create & switch to new branch
git checkout -         # return to previous branch
git branch -D <name>   # delete a branch
```

🔁 HEAD and Branch Info

cat .git/HEAD # see which branch HEAD points to

🌐 Working with Remotes

git remote -v

See remote connections.

git remote add origin <ssh-url>

Add a remote repository named `origin`.

git push origin main

git pull origin main

- `push`: upload local commits to remote
- `pull`: fetch + merge remote changes into local

🎭 Staging & Stashing

📥 Stash Changes

git stash save "more changes" # stash with message

git stash list # see stashes

git stash apply # reapply most recent stash

git stash apply stash@{1} # apply a specific stash

git stash drop stash@{1} # delete a stash

git stash pop # apply + drop = pop

Stash = temporarily save your uncommitted changes without committing.

🔄 Comparing Changes

git diff # view unstaged changes

git reset HEAD <file> # unstage file but keep changes

🧬 Advanced: Commits, Cherry-Pick, Revert

git cherry-pick <hash>

Apply a commit from another branch into your current branch.

git revert <hash>

Undo a commit by creating a new "inverse" commit. (Safe alternative to `reset` or `amend`)

⚔️ Merge Conflicts

Conflict Markers

When merging causes a conflict, Git marks them like this:

<<<<<<< HEAD

your local changes

=======

their remote changes

>>>>>>> origin/main

Accept Conflict Changes

git checkout --ours . # keep local version

git checkout --theirs . # keep remote version

📁 Ignore Files

cat .DS_Store > .gitignore

Use `.gitignore` to exclude files/folders (e.g. system or temp files) from version control.

🧱 Concepts Recap

- **Stage** = ready to commit (`git add`)
- **Commit** = save to Git history

- **Working Tree** = uncommitted changes in your working directory
- **origin/main** = main branch on the remote server
- **HEAD** = pointer to your current commit or branch
- **Amend** = edit the last commit (dangerous if already pushed)
- **Stash** = temporary storage for uncommitted changes
- **Pull** = `fetch` + `merge`

💬 Pro Tip

> Commits made in a different branch aren't visible in other branches until merged.

♻️ Undoing Things / Amending Commits

🔄 `git commit --amend`

Use this command to modify the most recent commit without creating a new one.

✨ When to Use

- You forgot to include a file in the last commit.
- You want to fix a typo in the commit message.
- You want to change both the message and the content of the last commit.

> ⚠️ **Warning**: Amending **changes the commit hash** — don't amend if the commit has already been pushed and shared with others.

🔧 Example

Initial commit:

bash

CopyEdit

```
git log
```

sql

CopyEdit

```
commit 46e75a2f4d16ee3cd1c78e91d7a2bbdaaa5419ac (HEAD -> master)

Author: urmi <urmi@appscode.com>

Date:   Wed Jun 4 09:56:31 2025 +0600


    c3 master: add amend.md
```

Now amend the commit message:

bash

CopyEdit

```
git commit --amend -m "c4 master: add amend.md"
```

Check the log again:

bash

CopyEdit

```
git log
```

sql

CopyEdit

```
commit f2154cec3d299a0d3b650b112947193e3b4d7040 (HEAD -> master)

Author: urmi <urmi@appscode.com>

Date:   Wed Jun 4 09:56:31 2025 +0600


    c4 master: add amend.md
```

Notice:

- The commit **message changed** from `c3` to `c4`.
- The **hash changed** (`46e75a2` → `f2154ce`), confirming it's a new commit replacing the old one.

🧼 Unstaging a Staged File (`git reset`)

Sometimes we stage files for commit using `git add`, but then realize we **don't want to commit all of them yet**. In such cases, we can **unstage** specific files.

🔄 Use Case

You staged two files:

- `amend.md`
- `contribution.md`

Now, you want to **unstage only `contribution.md`** — keeping its changes in the working directory but removing it from the staging area.

🧪 Real Example: Step-by-Step

🔍 1. Initial `status` (both files modified but not staged)

git status

Output:

Changes not staged for commit:

    modified: amend.md

    modified: contribution.md

➕ 2. Stage all files

git add *

📋 3. Now both files are staged

git status

Output:

Changes to be committed:

    modified: amend.md

    modified: contribution.md

🔄 4. Unstage `contribution.md` only

git reset HEAD contribution.md

Output:

Unstaged changes after reset:

M	contribution.md

✅ 5. Final `status`

git status

Output:

Changes to be committed:

	modified: amend.md

Changes not staged for commit:

	modified: contribution.md

💡 Summary

Action	Command
Stage a file	`git add <file>`
Unstage a file	`git reset HEAD <file>`
Unstage all files	`git reset`
Discard changes in working dir	`git restore <file>`

Discard changes AND unstage	`git reset --hard` (⚠️ dangerous)

🧠 Key Concept

> `git reset HEAD <file>` moves the file **from the staging area back to the working directory**, preserving your edits.

🧽 Discarding Local Changes (Unmodifying a File)

🔄 `git checkout -- <file>` (Legacy Way)

Use this command when you want to **revert a modified file** back to the last committed version — discarding all uncommitted changes in that file.

📌 Scenario: Revert `contribution.md` to Its Last Committed State

You had two files:

- `amend.md`: Staged
- `contribution.md`: Modified but **not staged**

You decided to **discard the local changes** in `contribution.md`.

💻 Step-by-Step

🔍 1. Check current status

bash

CopyEdit

```
git status
```

Output:

yaml

CopyEdit

```
Changes to be committed:

      modified:   amend.md


Changes not staged for commit:

      modified:   contribution.md
```

🔁 **2. Revert the modified file**

bash

CopyEdit

```
git checkout -- contribution.md
```

This **discards all changes** made to `contribution.md` since the last commit.

> ⚠️ No undo button — be careful! You will lose any unsaved work in that file.

✅ **3. Final `status` check**

bash

CopyEdit

```
git status
```

Output:

yaml

CopyEdit

```
Changes to be committed:

      modified:   amend.md
```

Now `contribution.md` is **gone from the status**, meaning it's **back to its last committed state**.

📘 Tip: Modern Alternative

The `git checkout -- <file>` command is now replaced by:

bash

CopyEdit

```bash
git restore <file>
```

So this does the same:

bash

CopyEdit

```bash
git restore contribution.md
```

🧠 Summary

Task	Command

Revert a file to last commit (old way)	`git checkout -- <file>`
Revert a file to last commit (new way)	`git restore <file>`
Unstage a file (keep changes)	`git reset HEAD <file>`
Discard staged file (both staging + edit)	`git restore --staged <file>`

🔄 Unstaging a File Using `git restore --staged`

📌 Goal:

You staged multiple files (e.g. using `git add *`), but now want to **unstage only one** — keeping its changes in the working directory.

🧪 Scenario: Unstage `contribution.md` but keep changes

You added everything:

bash
CopyEdit
```
git add *
```

1.

You checked the status:

bash
CopyEdit
```
git status
```

Output:

yaml
CopyEdit
```
Changes to be committed:

    modified:   amend.md

    modified:   contribution.md
```

2.
3. You realized you don't want to commit `contribution.md` just yet.

You **unstaged** it with:

bash
CopyEdit
```
git restore --staged contribution.md
```

4.

Checked the status again:

bash
CopyEdit
```
git status
```

Output:

yaml
CopyEdit
```
Changes to be committed:

    modified:   amend.md



Changes not staged for commit:

    modified:   contribution.md
```

5.

Now:

- ✅ `amend.md` is **still staged**
- ✅ `contribution.md` is **no longer staged**, but **you didn't lose your edits**

🧠 Quick Comparison

Task	Old Way	Modern Way
Unstage a file	`git reset HEAD <file>`	`git restore --staged <file>`
Discard working dir changes	`git checkout -- <file>`	`git restore <file>`

🔁 Summary Table

Situation	Command
Stage file for commit	`git add <file>`
Unstage file (keep changes) — 🆕 way	`git restore --staged <file>`
Unstage file (keep changes) — 🧓 old way	`git reset HEAD <file>`
Discard changes in working directory	`git restore <file>`

🔁 Unmodifying a Modified File with `git restore`

🧠 What it does:

`git restore <file>` resets the working directory version of a file to the **last committed state**, **discarding any local modifications**.

📌 Scenario: Revert changes in `contribution.md`

1. Your working directory had uncommitted changes in `contribution.md`.

You ran:

bash
CopyEdit
```
git restore contribution.md
```

2.
3. Outcome:
    - ❌ Changes in `contribution.md` are **discarded**
    - ✅ File is now identical to the last committed version
    - ✅ Safe to commit or ignore as needed

💻 Example

Before:

bash

CopyEdit

```
git status
```

Shows:

yaml

CopyEdit

```
Changes not staged for commit:

  modified: contribution.md
```

Then:

bash

CopyEdit

```
git restore contribution.md
```

After:

bash

CopyEdit

```
git status
```

Shows:

yaml

CopyEdit

```
No longer modified: contribution.md
```

🔂 Related Commands

Action	Command
Unstage a file (keep changes)	`git restore --staged <file>`
Discard working directory changes	`git restore <file>`
Old way (before `restore`)	`git checkout -- <file>`

> 📝 **Warning:** `git restore <file>` deletes local changes. Use with caution if you haven't backed up or committed.

🌐 Working with Remote Repositories in Git

✅ Adding a New Remote

bash

CopyEdit

```
git remote add <name> <url>
```

Example:

bash

CopyEdit

```
git remote add pb git@github.com:paulboone/ticgit.git
```

✅ Adds a second remote named `pb`
✅ Now you can pull from Paul Boone's repository as well

🔍 Viewing Remotes

bash

CopyEdit

```
git remote -v
```

Shows:

scss

CopyEdit

```
origin     git@github.com:schacon/ticgit.git (fetch)
origin     git@github.com:schacon/ticgit.git (push)
pb        git@github.com:paulboone/ticgit.git (fetch)
pb        git@github.com:paulboone/ticgit.git (push)
```

📥 Fetching from a Remote

bash

CopyEdit

```
git fetch <remote-name>
```

Example:

bash

CopyEdit

```
git fetch pb
```

✅ Downloads branches from the `pb` remote
✅ Adds them locally as `pb/master`, `pb/ticgit`, etc.

📋 Inspecting Remote Details

bash

CopyEdit

```
git remote show <remote-name>
```

Example:

bash

CopyEdit

```
git remote show origin
```

Gives detailed info:

- Remote URL (fetch & push)
- Default branch
- Tracked branches
- Pull/Push tracking info

🔄 Renaming a Remote

bash

CopyEdit

```
git remote rename <old-name> <new-name>
```

Example:

bash

CopyEdit

```
git remote rename pb paul
```

✅ Changes `pb` → `paul`
✅ All references updated

❌ Removing a Remote

bash

CopyEdit

```
git remote remove <name>
```

Example:

bash

CopyEdit

```
git remote remove paul
```

✅ Deletes the remote config and all its refs (e.g. `paul/master`)

📤 Pushing to a Remote

bash

CopyEdit

```
git push origin master
```

✅ Uploads your local `master` branch to the `origin` remote
✅ Only works if you have write access

🔖 Git Tags: Creating, Viewing, Sharing, and Deleting Versions

📃 Listing Tags

bash

CopyEdit

```
git tag
```

Shows all existing tags (e.g.):

CopyEdit

```
v1.0

v2.0
```

🎯 Filter Tags by Pattern

bash

CopyEdit

```
git tag -l "v1.8.5*"
```

Example output:

python-repl

CopyEdit

```
v1.8.5

v1.8.5-rc0

v1.8.5.1

...
```

✍️ Creating Tags

🏷️ Annotated Tag (Recommended)

Includes tagger name, message, date, and can be GPG-signed:

bash

CopyEdit

```
git tag -a v1.4 -m "my version 1.4"
```

🪶 Lightweight Tag

Just a pointer to a commit, with no extra metadata:

bash

CopyEdit

```
git tag v1.4-lw
```

🔍 Viewing Tag Details

bash

CopyEdit

```
git show <tagname>
```

Example:

bash

CopyEdit

```
git show v1.2
```

Shows tag metadata and the commit it points to.

🕰️ Tagging Past Commits

bash

CopyEdit

```
git tag -a <tagname> <commit-id>
```

Example:

bash

CopyEdit

```
git tag -a v1.2 189325a
```

✅ Use this if you forgot to tag an older release.

📤 Sharing Tags

By default, `git push` does **not** push tags.

Push a single tag:

bash

CopyEdit

```
git push origin v1.5
```

Push all tags:

bash

CopyEdit

```
git push origin --tags
```

❌ Deleting Tags

Locally:

bash

CopyEdit

```
git tag -d v1.4-lw
```

On the Remote (you must have push access):

bash

CopyEdit

```
git push origin --delete v1.4-lw
```

or

bash

CopyEdit

```
git push origin :refs/tags/v1.4-lw
```

📂 Checking Out Tags

bash

CopyEdit

```
git checkout <tagname>
```

> ⚠️ This puts you in **"detached HEAD"** state
> If you commit now, your changes will not belong to any branch

✅ **To work from a tag safely, create a branch:**

bash

CopyEdit

```
git checkout -b version1.2 v1.2
```

🧾 Git Aliases: Shortcuts for Common Commands

Create a shorthand for a command using:

bash

CopyEdit

```
git config --global alias.<shortcut> <git-command>
```

Example:

bash

CopyEdit

```
git config --global alias.ci commit
```

Usage:

bash

CopyEdit

```
git ci -m "C1 master: add abc.md"
```

🌿 Git Branching: Creating, Switching, and Understanding HEAD

🧠 What Is `HEAD` in Git?

- In Git, `HEAD` is a special pointer that tells you **which branch you're currently on**.
- It always points to the **latest commit on your current branch**.
- Unlike SVN or CVS, Git's `HEAD` is not a static revision — it's dynamic and branch-aware.

🌱 Creating a New Branch

bash

CopyEdit

```
git branch <branch-name>
```

> ⚠️ This does **not** switch to the new branch, it only **creates** it.

Example:

bash

CopyEdit

```
git branch testing
```

Now both `version1.2` and `testing` point to the same commit.

🔍 Viewing Branch Decorations

bash

CopyEdit

```
git log --oneline --decorate
```

Example output:

csharp

CopyEdit

```
be9cc48 (HEAD -> version1.2, testing) C1 master: add abc.md
```

✅ Shows which branches point to each commit.

🌲 Making a Commit on a Branch

After switching to `version1.2`, you added and committed a new file:

bash

CopyEdit

```
echo "acd" > acd.md
```

```
git add .

git commit -m "c2 version1.2 : add acd.md"
```

Now the `version1.2` branch has moved ahead, while `testing` still points to the earlier commit.

bash

CopyEdit

```
git log --oneline --decorate
```

Output:

csharp

CopyEdit

```
8c2890e (HEAD -> version1.2) c2 version1.2 : add acd.md

be9cc48 (testing) C1 master: add abc.md
```

🔄 Switching Between Branches

bash

CopyEdit

```
git checkout <branch-name>
```

Example:

bash

CopyEdit

```
git checkout testing
```

Now `HEAD` points to `testing`, and the working directory reflects its state.

bash

CopyEdit

```
git log --oneline --decorate
```

Output:

csharp

CopyEdit

```
be9cc48 (HEAD -> testing) C1 master: add abc.md
```

Notice how `acd.md` is not in this branch — because it was committed only in `version1.2`.

📝 Summary

Action	Command	Notes
Create a branch	`git branch <branch>`	Does **not** switch
Switch branches	`git checkout <branch>`	Updates `HEAD`
View branches and commits	`git log --oneline --decorate`	Shows what branch points where

Commit stays in current branch	`git commit -m "msg"`	Only moves the branch you're on

🔀 Git Divergence, Merging, and Branch Management

🧠 What is “Divergent History”?

Divergence happens when two branches (or the same-named branch in different repos/remotes) share a common ancestor but each develop **different commits**.

Example:

bash

CopyEdit

```
# Start from master

git checkout -b testing

git commit -a -m 'Make a change on testing'



# Switch back to master

git checkout master

git commit -a -m 'Make other changes on master'
```

> Now both `master` and `testing` have **unique commits** — their histories have diverged.

🔍 View Divergence

bash

CopyEdit

```
git log --oneline --decorate --graph --all
```

Visual Example:

markdown

CopyEdit

```
* c2b9e (HEAD -> master) Make other changes
| * 87ab2 (testing) Make a change
|/
* f30ab Shared starting commit
```

✅ Why Divergence is Powerful in Git

- Branches are **cheap and fast** in Git.
- Encourages **parallel development** (features, bugfixes, experiments).
- Helps maintain a **clean history** through merging or rebasing.

🔄 Merging Diverged Branches

To combine changes:

bash

CopyEdit

```
git checkout master
git merge testing
```

- If the histories are **compatible**, Git will auto-merge.
- If **conflicts** exist, Git will pause for manual resolution.

🔍 Checking Merge Status

List branches that **are fully merged** into the current branch:

bash

CopyEdit

```
git branch --merged
```

List branches that **are not merged**:

bash

CopyEdit

```
git branch --no-merged
```

❌ Deleting a Branch

Safe deletion (only if fully merged):

bash

CopyEdit

```
git branch -d branch-name
```

Force deletion (even if not merged):

bash

CopyEdit

```
git branch -D branch-name
```

Example:

bash

CopyEdit

```
git branch -d version1.2    # Fails if not merged

git branch -D version1.2    # Forces deletion
```

🏷️ Renaming a Branch

bash

CopyEdit

```
git branch -m old-name new-name
```

Example:

bash

CopyEdit

```
git branch --move new-feature new
```

🌐 Working with Remotes (Team Simulation)

You're practicing with:

Remote	Simulates	Example Command
`origin`	Central repo (e.g., GitHub)	`git push origin main`
`teamone`	Team repo or fork	`git remote add teamone ../teamone`
`standalone`	Personal repo/fork	`git remote add standalone ../standalone`

🔄 Fetch & Compare Divergence Across Remotes

1. Fetch updates:

bash

CopyEdit

```
git fetch teamone

git fetch standalone
```

2. Compare branches with same name:

bash

CopyEdit

```
git log serverfix..teamone/serverfix --oneline

git log teamone/serverfix..serverfix --oneline
```

⛏️ Merge Divergent Remote Branches

To bring in changes from `teamone`'s version of `serverfix`:

bash

CopyEdit

```
git checkout serverfix

git merge teamone/serverfix
```

🚦 Summary: Key Commands

Action	Command Example
Create + switch to new branch	`git checkout -b branch-name`
View all history with branches	`git log --oneline --decorate --graph --all`
Merge a branch	`git merge branch-name`
List merged branches	`git branch --merged`
Delete a branch	`git branch -d branch-name` or `-D`

Rename a branch	`git branch -m old-name new-name`
Compare branch divergence	`git log branch1..branch2`
Merge remote branch	`git merge remote/branch`

📘 Git Rebase: Step-by-Step Practice & Notes

Git rebasing lets you **move or reapply commits** on top of another base. This helps you keep a clean, linear history — especially useful when working on feature branches.

✅ What You'll Learn

1. Basic rebasing
2. Fast-forward merge after rebase
3. Rebasing with `--onto`
4. When **not** to rebase
5. Pulling with `--rebase` and force-pushes

🧪 Step 1: Setup — Simulate Divergence

bash

CopyEdit

```
mkdir rebase-practice

cd rebase-practice

git init
```

Create an initial commit:

bash

CopyEdit

```
echo "start" > file.txt

git add file.txt

git commit -m "Initial commit"
```

Make a change on `master`:

bash

CopyEdit

```
git checkout -b master

echo "master change" >> file.txt

git commit -am "Master update"
```

Make a different change on `feature`:

bash

CopyEdit

```
git checkout -b feature

echo "feature work" >> file.txt
```

```
git commit -am "Feature update"
```

Now:

- `master` has one change
- `feature` has another
- The histories have **diverged**

🔁 Step 2: Rebase `feature` onto `master`

bash

CopyEdit

```
git checkout feature

git rebase master
```

✅ Git will:

- Move `feature` to the tip of `master`
- Reapply the `Feature update` commit on top

Check the history:

bash

CopyEdit

```
git log --oneline --graph --all
```

Expected output:

sql

CopyEdit

```
* 3456abc (feature) Feature update

* 1234def (master) Master update

* 9abc000 Initial commit
```

🚀 Step 3: Fast-Forward Merge to Master

bash

CopyEdit

```
git checkout master

git merge feature
```

Since `feature` is now on top of `master`, Git does a **fast-forward** merge — no new merge commit.

🎯 Step 4: Use `--onto` to Change Base

Let's say you want to rebase a branch **off a different base** than where it originally branched from.

Create new branches:

bash

CopyEdit

```
git checkout -b server master

echo "server" >> file.txt
```

```
git commit -am "Server commit"
```

```
git checkout -b client
echo "client1" >> file.txt
git commit -am "Client commit 1"
echo "client2" >> file.txt
git commit -am "Client commit 2"
```

Now:

- `client` branched off `server`
- But we want to rebase it onto `master` instead

Run:

bash

CopyEdit

```
git rebase --onto master server client
```

✅ Git:

- Finds the commits on `client` that are **not** on `server`
- Reapplies them **onto `master`**

⚠️ Step 5: Danger — Rewriting Public History

Let’s simulate pushing and **force-pushing after rebase**:

Set up a fake remote

bash

CopyEdit

```
mkdir ../main-repo

cd ../main-repo

git init --bare
```

Back in your working repo:

bash

CopyEdit

```
cd ../rebase-practice

git remote add origin ../main-repo

git push -u origin master

git push origin client
```

Now, rebase and **force-push**:

bash

CopyEdit

```
git checkout client

git rebase -i master  # squash or reword if needed

git push --force
```

⚠️ If others pulled the old history of `client`, they now have a conflict.

They can recover using:

bash

CopyEdit

```
git fetch origin

git rebase origin/client
```

Git will try to patch intelligently, but this can cause confusion.

🔧 Step 6: Pull with Rebase

When working on a shared branch (like `main` or `develop`), prefer:

bash

CopyEdit

```
git pull --rebase
```

This keeps the history linear by **replaying your local commits** on top of the updated remote branch.

Set this as default:

bash

CopyEdit

```
git config --global pull.rebase true
```

🧠 Key Rules to Remember

✅ Do	❌ Don't
Rebase before pushing (if changes are local)	Rebase commits already pushed/shared with others
Use `pull --rebase` for a clean history	Force-push without telling teammates
Use `--onto` for advanced rebase control	Rebase merge commits (they're skipped)
Squash/reword with `rebase -i` for tidy PRs	Rebase branches used by others without coordination

🧠 Git Command Reference: Rebasing, Pushing, Merging, Remotes

🔁 Handling Upstream Updates Before Pushing

If your remote (e.g., `origin/main`) is updated and you want to push your own work:

✅ **With Rebase:**

bash

CopyEdit

```
git fetch origin

git rebase origin/main

git push
```

```
# or in one step:
git pull --rebase
git push
```

✅ **With Merge:**

bash

CopyEdit

```
git fetch origin
git merge origin/main
git push
# or:
git pull
git push
```

🧼 Resetting `main` to an Older Commit (e.g., main is "locked")

bash

CopyEdit

```
git checkout main
git reset --hard <commit-id>   # e.g., git reset --hard c1
```

> ⚠️ Use `--hard` carefully — it discards local changes.

📤 Pushing Multiple Local Branches to Remote Main

Example: Rebase and push `side1` onto `origin/main`:

bash

CopyEdit

```
git rebase origin/main side1

git push origin side1:main
```

🔀 Merging Two Branches

To merge `side1` into `main`:

bash

CopyEdit

```
git checkout main       # HEAD should be on target branch

git merge side1         # Merge source into target
```

🌱 Creating a Local Branch Tracking `origin/main`

bash

CopyEdit

```
git checkout -b foo origin/main

# or if 'foo' already exists:

git branch -u origin/main foo
```

🚀 Remote Push Syntax Breakdown

Basic Push (push current branch to same remote name)

bash

CopyEdit

```
git push origin main
```

Push to Different Remote Branch Name

bash

CopyEdit

```
git push origin main:newBranch
```

Push to Remote Branch Using Local Ref's Parent

bash

CopyEdit

```
git push origin foo^:main

# ⚠️ Destructive: moves remote 'main' to parent of foo
```

Delete Remote Branch

bash

CopyEdit

```
git push origin :foo
```

📥 Fetching Specific Refs

Fetch a remote commit and create a local branch:

bash

CopyEdit

```
git fetch origin <remote-commit>:<local-branch>

# Example:

git fetch origin c2:bar
```

> This moves `bar` in local without creating `origin/bar`.

🛠️ Special `git pull` Forms

bash

CopyEdit

```
git pull origin foo

# Same as:

git fetch origin foo

git merge origin/foo
```

bash

CopyEdit

```
git pull origin bar:bugFix

# Same as:

git fetch origin bar:bugFix
```

```
git merge bugFix
```

✅ Quick Tips

Action	Command Example
Rebase your work on remote	`git fetch; git rebase origin/main`
Merge remote before push	`git fetch; git merge origin/main`
Reset local branch	`git reset --hard <commit-id>`
Track remote branch	`git checkout -b foo origin/main`
Delete remote branch	`git push origin :branchName`
Push under different name	`git push origin localBranch:remoteBranch`