Git Tutorial — Foundations

Introduction to Git

Git is a **version control system** used to track and manage changes to files over time.
It is the **most widely used version control system** in the software industry today.

Git helps individuals and teams work on the same project without losing work or overwriting each other's changes.

What Is Version Control?

Version control is software that keeps a **history of changes** made to a project.

It allows you to:

- Track how files evolve over time
- Restore earlier versions when needed
- Compare different versions of files
- Undo mistakes safely

Version control becomes essential as projects grow in size and complexity.

Git

Git is a **version control system**.

It is:

- Free and open source
- Fast and efficient
- The most widely used VCS today

Git is specifically a **distributed** version control system.

Centralized Version Control System (CVCS)

A **CVCS** has:

- One central server that stores the project history
- Clients that depend on this central server

Examples:

- Subversion (SVN)
- CVS

Limitations:

- Requires network access
- Central server is a single point of failure

Distributed Version Control System (DVCS)

A **DVCS** allows:

- Every developer to have a full copy of the repository
- Local commits without internet access
- Better performance and reliability

Git is a **DVCS**.

Key Differences (At a Glance)

Term	Meaning
VCS	Software for tracking file changes
CVCS	Centralized version control system
DVCS	Distributed version control system
Git	A popular DVCS

Why Version Control Matters

Modern software projects involve:

- Large codebases
- Many developers working in parallel
- Continuous daily changes

Without version control:

- Changes get lost
- Files overwrite each other
- Mistakes are difficult to fix

Git solves these problems by providing a structured way to manage and merge changes.

What Git Helps You Do

Git allows you to:

- Save snapshots of your project at different points in time
- See who made a change and why
- Compare changes between versions
- Roll back to a stable state when something breaks
- Share and combine work with others

Think of Git as a **safety net for your code**.

Git Compared to Other Version Control Systems

Git is not the only version control system.

Other tools include:

- Subversion (SVN)
- CVS
- Mercurial

While these tools have similar goals, Git stands out because it is:

- Faster
- More flexible
- Better suited for collaboration
- Scalable from small to very large projects

As a result, Git has become the industry standard.

Popularity and Industry Adoption

Git is used by:

- Individual developers
- Startups
- Large technology companies like Google and Facebook

Industry surveys show that **almost all professional developers use Git** today.

Learning Git is considered a **basic requirement** for software-related roles

How Git Describes Itself

Git is described as:

- Free and open source
- Distributed
- Efficient for projects of all sizes

In practice, this means:

- Anyone can use Git
- Every developer has a complete project history
- Git performs well even for large repositories

Key Concept Summary

Git:

- Is a version control system
- Tracks changes over time
- Helps manage collaboration
- Allows safe experimentation and recovery
- Is the dominant tool used in the industry

This foundation is critical before learning Git commands.

Git	GitHub

Git is a **version control system**. It is: ● A tool installed on your computer ● Used from the command line or IDE ● Responsible for tracking changes in files ● Responsible for creating checkpoints (versions) Git works **locally** on your machine.	GitHub is a **hosting platform for Git repositories**. It is: ● A web-based service ● Used to store Git projects online ● Built on top of Git GitHub does **not replace Git** — it uses Git.
You can use Git: ● Without internet ● Without GitHub ● Completely on your own system	GitHub adds features around Git, such as: ● Remote storage for repositories ● Collaboration between developers ● Code review via pull requests ● Issue tracking ● Project documentation ● Access control and permissions Think of GitHub as a **collaboration layer** on top of Git.
Git helps you: ● Track file changes ● Save project history ● Move between versions ● Create parallel versions of a project ● Combine changes safely Git is the **engine** that manages versions.	With GitHub, you get a **remote repository**: ● Lives on GitHub's servers ● Acts as a shared central copy Git helps you move changes between: ● Local repository ● Remote repository (GitHub)
Git handles **versioning**.	GitHub handles **sharing and collaboration**.
Git: ● Version control tool ● Runs locally	GitHub: ● Online platform ● Hosts Git repositories

• Tracks history	• Enables collaboration

How Git and GitHub Work Together

A common workflow:

- You write code locally
- Git tracks your changes
- You create checkpoints locally
- You send your changes to GitHub
- Others can review and pull your changes

GitHub Is Not the Only Option

GitHub is popular, but it is not the only Git hosting service.

Other platforms include:

- GitLab
- Bitbucket
- Azure Repos

All of them:

- Use Git underneath
- Provide similar collaboration features

Learning Git makes it easy to switch between platforms

Common Beginner Misconceptions

“GitHub stores my code”
→ Git stores the history, GitHub hosts a copy.

“I need GitHub to use Git”
→ Git works perfectly without GitHub.

“GitHub is Git”
→ GitHub is a service built around Git.

Clearing this confusion early is very important.

What Is a Git Client?

A **Git client** is a tool that allows you to interact with Git.

It acts as:

- An interface to run Git commands
- A way to view Git history, changes, and branches
- A helper for performing Git operations more easily

All Git clients ultimately use **Git underneath**.

Types of Git Clients

Git clients generally fall into three categories:

- Command-line clients
- Graphical (GUI) clients
- IDE-integrated clients

Each type serves different user preferences and workflows.

Command-Line Git Client

The command-line client is the **official and most powerful** way to use Git.

Characteristics:

- Uses terminal commands (e.g., `git status`, `git commit`)
- Available on all platforms
- Provides full access to Git features
- Works consistently across environments

This is considered the **industry baseline**.

Why the Command Line Matters

Using Git from the command line:

- Builds strong foundational understanding
- Matches documentation and tutorials
- Avoids tool-specific limitations
- Works the same on any system

Most advanced Git workflows assume command-line knowledge.

Graphical (GUI) Git Clients

GUI clients provide a visual interface for Git.

They typically offer:

- Buttons instead of commands
- Visual diffs and commit graphs
- Easier onboarding for beginners

GUI clients still execute Git commands behind the scenes.

Common GUI Git Clients

Popular GUI tools include:

- GitHub Desktop
- Sourcetree
- GitKraken
- Tower

These tools can:

- Simplify common tasks
- Improve visualization
- Reduce command memorization initially

Limitations of GUI Clients

GUI clients:

- Do not always expose all Git features
- Can hide what Git is actually doing
- May behave differently across tools

Relying only on GUIs can make debugging harder later.

IDE-Integrated Git Clients

Many code editors and IDEs include Git support.

Examples:

- Visual Studio Code
- IntelliJ IDEA
- Eclipse

These integrations allow:

- Running Git actions from within the editor
- Viewing changes alongside code
- Managing commits and branches visually

IDE Clients in Practice

IDE Git tools are:

- Convenient for daily development
- Good for basic workflows
- Often built on top of command-line Git

They work best when combined with command-line knowledge.

Installing Git on Windows

To use Git on Windows, you need to install the official **Git for Windows** package.
This installs Git along with Git Bash, which provides a Unix-like terminal for Git commands.

Downloading Git for Windows

Git is distributed officially for Windows via **Git for Windows**.

Steps:

- Open a web browser
- Go to the official Git website: https://git-scm.com
- The website automatically suggests a Windows installer
- Download the `.exe` installer

Always download Git from the official source to avoid issues.

Verifying the Installation

After installation:

- Open **Git Bash** from the Start menu
- Run the command:

```
git --version
```

Git Bash vs Command Prompt vs PowerShell

Git for Windows provides **Git Bash**, which:

- Supports Unix-style commands
- Matches most Git tutorials online
- Is recommended for learning Git

Git can also be used in:

- Command Prompt
- PowerShell
- Windows Terminal

Git Bash is the easiest option for beginners.

Configuring Username in Git

Understanding `git config --global user.name "Vishal Shah"`

This command sets your **name** in Git configuration.
Git uses this name to identify **who created a commit**.

What This Command Does

```
git config --global user.name "Vishal Shah"
```

This tells Git:

- Your name is **Vishal Shah**
- Attach this name to every commit you create
- Use this name across all Git repositories on your system

Every commit will record this information as metadata.

Git Repository

A **Git repository** is where Git stores:

- Your project files
- The complete history of changes

It is the basic unit Git works with.

Key Points

- A repository tracks a project over time
- Git stores history inside a hidden `.git` folder
- Without `.git`, a folder is not a Git repository
- Repositories can be **local** or **remote** (e.g., GitHub)

Remember

A repository is **your project with memory**.

The Three Core Areas in Git

Git works using **three logical areas** that help manage changes safely.

Understanding these is **critical** before learning Git commands.

Working Directory

The **working directory** is:

- The folder where you edit files
- Where you write code, add content, or delete files

Changes here are:

- Not yet tracked by Git
- Just regular file edits on your computer

Staging Area

The **staging area** is:

- A temporary holding area for changes
- Used to prepare what will go into the next checkpoint

Only staged changes:

- Are included in the next commit
- Give you control over what gets saved

Think of it as a **review area**.

Repository (Commit History)

The **repository** is:

- Where Git permanently stores checkpoints (commits)
- The project's full history

Once changes reach the repository:

- They are safely recorded

- They can be revisited later

Simple Flow to Remember

`Working Directory → Staging Area → Repository`

- Edit files → Working Directory
- Select changes → Staging Area
- Save checkpoint → Repository

Why Git Uses Three Areas

This design allows you to:

- Make many changes freely
- Choose exactly what to commit
- Avoid committing unfinished work

This is one of Git's biggest strengths.

File States in Git

File States in Git

Git tracks files by assigning them a **state** based on where they are in the workflow.

Understanding file states explains **why Git behaves the way it does**.

Untracked Files

An **untracked file** is:

- A new file Git does not know about
- Not yet part of the repository

Git will not save this file until you explicitly tell it to.

Modified Files

A **modified file** is:

- A file Git is already tracking
- Changed in the working directory
- Not yet staged

Changes exist only locally at this point.

Staged Files

A **staged file** is:

- A file added to the staging area
- Ready to be included in the next commit

Only staged files are saved when you commit.

Committed Files

A **committed file** is:

- Safely stored in the repository
- Part of Git's history
- Restorable at any time

This represents a saved checkpoint.

File State Flow

`Untracked → Modified → Staged → Committed`

Creating a Repository — `git init`

What `git init` Does

`git init` turns a normal folder into a **Git repository**.

It:

- Creates a hidden `.git` directory
- Enables version control for that folder
- Does **not** track files automatically

Example

You have a new project folder:

```
my-project/
```

Run:

```
git init
```

Result:

- Git creates `.git/`
- The folder is now a Git repository
- No files are tracked yet

Checking File States — `git status`

What `git status` Does

`git status` shows:

- Which files are untracked
- Which files are modified
- Which files are staged
- What Git is ready to commit

This is the **most-used Git command**.

Example

Create a file:

```
my-project/
 └── index.html
```

Run:

```
git status
```

Output (simplified):

```
Untracked files:
  index.html
```

Meaning:

- Git sees the file

- Git is not tracking it yet

Staging Changes — `git add`

What `git add` Does

`git add` moves changes from:

- **Working Directory → Staging Area**

It does **not** create a commit.

Staging means:

- "Include this in the next checkpoint"

Example: Staging One File

```
git add index.html
```

Check status again:

```
git status
```

Output:

```
Changes to be committed:
  index.html
```

Meaning:

- The file is staged
- Ready to be committed

Example: Modified File

Edit `index.html`.

Run:

```
git status
```

Output:

```
Changes not staged for commit:
  index.html
```

Meaning:

- File was changed
- Git is tracking it
- Changes are not staged yet

Stage it again:

```
git add index.html
```

Example: Staging Multiple Files

Add everything at once:

```
git add .
```

This stages:

- All new files
- All modified files

Use carefully in real projects.

Visual Flow (Important)

```
git init      → creates repository
edit files    → working directory
git status    → checks file states
git add       → stages changes
```

Creating Your First Checkpoint — `git commit`

What `git commit` Does

`git commit` creates a **checkpoint** in the repository.

It:

- Saves the staged changes permanently
- Records who made the change and when
- Adds the changes to Git's history

Only **staged files** are included in a commit.

Example

After staging files:

```
git commit -m "Add initial project files"
```

Result:

- A new checkpoint is created
- Git stores a snapshot of the staged changes
- The working directory becomes clean

What a Commit Represents

A commit is:

- A snapshot of the project at a point in time
- A logical unit of work
- A reference you can return to later

Commits allow Git to “time travel.”

Writing Good Commit Messages

Why Commit Messages Matter

Commit messages explain:

- What changed
- Why the change was made

They help:

- You in the future
- Teammates reviewing history
- Debugging and auditing changes

Characteristics of a Good Commit Message

A good commit message is:

- Short and clear
- Written in the present tense
- Focused on **what** was done, not how

Examples:

- `Add login page`
- `Fix navbar alignment`
- `Update README with setup steps`

Common Commit Message Mistakes

Avoid messages like:

- `fix`
- `changes`
- `final`
- `updated stuff`

These provide no useful context.

Viewing Commit History — `git log`

What `git log` Does

`git log` shows the **history of commits**.

It displays:

- Commit IDs
- Author names
- Dates
- Commit messages

This helps you understand how the project evolved.

Example

```
git log
```

Output (simplified):

```
commit a1b2c3d4
Author: Vishal Shah
Date:   Jan 20 2026

    Add initial project files
```

Each entry represents one checkpoint.

Understanding Commit IDs

Commit IDs:

- Are unique hashes
- Identify each commit permanently
- Allow Git to reference specific versions

You don’t need to memorize them—Git tools handle that.

Typical Beginner Workflow

```
edit files
git status
git add
git commit
git log
```

This is the core Git cycle used daily.

Committing Basics Exercise

1. Create a new folder called `Shopping`
2. Initialize a new git repo inside of the `Shopping` folder (make sure you're not already inside of a Git repo!)
3. Make a new file called `yard.txt`
4. Make another new file called `groceries.txt`
5. Make a commit that includes both empty files. The message should be "create yard and groceries lists"
6. In the `yard.txt` file, add the following changes:

```
None
- 2 bags of potting soil
- 1 bag of worm castings
```

7. In the `groceries.txt` file, add the following:

```
None
- 4 tomatoes
- 6 shallots
- 1 fennel bulb
```

8. Make a new commit, including **ONLY the changes from the `groceries.txt` file.** The commit message should be "add ingredients for tomato soup"
9. Make a second commit including **ONLY the changes to the `yard.txt` file.** It should have the commit message "add items needed for garden box"
10. Next up, add the following line to the end of `groceries.txt`

```
None
- couple of seed potatos
```

11. In the `yard.txt` file, change the first line so that it says "**3** bags of potting soil" instead of "2 bags of potting soil"

```
None
- 3 bags of potting soil
- 1 bag of worm castings
```

12. **Make a commit that includes the changes to BOTH files.** The message should read "add items needed to grow potatoes"
13. **Use a Git command to display a list of the commits. You should see 4!**

Amending the Last Commit

Amending a commit allows you to **modify the most recent checkpoint** instead of creating a new one.

This is commonly used to:

- Fix mistakes
- Add missing files
- Correct commit messages

When Should You Amend a Commit?

Use amending when:

- The commit is **local**
- The commit has **not been pushed**
- The change is small and related

Avoid amending shared commits.

Command Used to Amend

```
git commit --amend
```

This command:

- Replaces the latest commit
- Creates a new commit ID
- Keeps history clean

Ignoring Files with `.gitignore`

Why Ignore Files in Git?

Not all files in a project should be tracked by Git.

Some files are:

- Generated automatically

- Environment-specific
- Temporary or local to your machine

Git provides a way to **ignore these files**.

What Is `.gitignore`?

`.gitignore` is a file that tells Git:

- Which files or folders to ignore
- Which files should not be tracked or committed

Ignored files:

- Do not appear in `git status`
- Are not added accidentally
- Are excluded from commits

Common Files to Ignore

Typical examples include:

- Build outputs
- Dependency folders
- Log files
- Environment configuration files
- Editor and OS-generated files

These files change often and are not part of the source code.

Creating a `.gitignore` File

Steps:

- Create a file named `.gitignore` in the repository root
- Add patterns for files to ignore
- Save the file

Git automatically applies the rules.

Example `.gitignore`

```
node_modules/
.env
```

```
*.log
dist/
```

This ignores:

- Dependency folders
- Environment files
- Log files
- Build artifacts

How `.gitignore` Works

`.gitignore` uses **patterns**:

- File names
- Folder names
- Wildcards

Git checks these rules before tracking files.

Important Rule (Very Common Mistake)

`.gitignore` only works for **untracked files**.

If a file is already tracked:

- Adding it to `.gitignore` will NOT remove it
- Git will continue tracking it

Tracked files must be removed manually.

ranches and Branching

What Is a Branch in Git?

A **branch** is a separate line of development in a Git repository.

It allows you to:

- Work on new features
- Fix bugs
- Experiment safely
  without affecting the main codebase.

Branches make **parallel work** possible.

The Main (or Master) Branch

The **main branch** (formerly called `master`) is:

- The default branch in a repository
- The primary line of development
- Usually contains stable, production-ready code

Modern Git repositories use `main` instead of `master`.

Why Branches Exist

Branches allow you to:

- Isolate work
- Avoid breaking stable code
- Develop features independently
- Merge changes when ready

Without branches, collaboration would be chaotic.

Understanding `HEAD`

`HEAD` is a **pointer** that tells Git:

- Where you currently are in the repository
- Which branch (or commit) you are working on

Normally:

- `HEAD` points to a branch
- The branch points to a commit

How to View All Branches

To see all local branches:

```
git branch
```

Output example:

```
* main
  feature-login
  bugfix-header
```

- `*` indicates the current branch
- This is where `HEAD` is pointing

Creating a New Branch

To create a new branch:

```
git branch feature-login
```

This:

- Creates a new branch
- Does NOT switch to it

You remain on the current branch.

Switching Branches (Traditional Way)

Using `git checkout`:

```
git checkout feature-login
```

This:

- Moves `HEAD` to the new branch
- Updates the working directory

`git checkout` is powerful but overloaded.

Creating and Switching in One Step

Using checkout:

```
git checkout -b feature-login
```

This:

- Creates the branch
- Switches to it immediately

`git checkout` vs `git switch`

Git introduced `git switch` to reduce confusion.

`git checkout`

- Old, multipurpose command
- Used for branches and files
- Still widely used

`git switch`

- Newer, clearer command
- Used **only** for branches
- Easier for beginners

Example:

```
git switch feature-login
```

Create and switch:

```
git switch -c feature-login
```

Recommended for learning: `git switch`

Switching Branches with Unstaged Changes

If you have **unstaged changes**:

- Git allows switching **only if changes don't conflict**

If conflicts exist:

- Git blocks the switch
- You must commit or stash changes first

This prevents accidental data loss.

Deleting a Branch

Delete a local branch:

```
git branch -d feature-login
```

Git allows deletion only if:

- The branch has been merged
- No work will be lost

Force delete (dangerous):

```
git branch -D feature-login
```

Renaming a Branch

Rename the current branch:

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

Rename another branch:

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

Useful when:

- Renaming `master` to `main`
- Fixing naming conventions

How Git Stores Branches Internally

Internally:

- A branch is just a **pointer to a commit**
- It moves forward as new commits are added

Git does **not** copy files for branches.

Branches are:

- Lightweight
- Cheap
- Fast

How `HEAD` and Branches Work Together

Conceptually:

```
HEAD → branch → commit
```

When you:

- Switch branches → `HEAD` moves
- Commit → branch pointer moves forward
- Merge → pointers are updated

This is why branching is so efficient in Git.

Branching Exercise

1. Make a new folder called `Patronus`
2. Make a new git repo inside the folder (make sure you're not in an existing repo)
3. Create a new file called `patronus.txt` (leave it empty for now)
4. Add and commit the empty file, with the message "add empty patronus file"
5. Immediately make a new branch called `harry` and another branch called `snape` (both based on the master branch)
6. Move to the `harry` branch using the "new" command to change branches.
7. In the `patronus.txt` file, add the following:

```
None
HARRY'S PATRONUS


   /|       |\\

`__\\\\       //__'

   ||      ||

 \\__`\\     |'__/

   `_\\\\   //_'

   _.,:---;,._

   \\_:     :_/

     |@. .@|

     |     |

     ,\\.-./ \\

     ;;`-'   `---__________-----.-.

     ;;;                        \\_\\

     ';;;                        |
```

```
      ;    |                      ;
       \\   \\     \\         |      /
        \\_, \\    /         \\     |\\
          |';|  |,,,,,,,,/ \\    \\ \\_
          |  |  |           \\   /   |
          \\  \\  |            |  / \\  |
           | || |           | |   | |
           | || |           | |   | |
           | || |           | |   | |
           |_||_|           |_|   |_|
          /_//_/           /_/   /_/
```

8. Add and commit the changes, with the commit message "add harry's stag patronus"
9. Move over to the `snape` branch using the "older" command to change branches.
10. Put the following text in the `patronus.txt` file:

```
None
```

```
SNAPE'S PATRONUS

                    .     _,
                    |`\\__/ /
                    \\  . .(
                     | __T|
                    /   |
      _.----------'    |
```

```
     //               {}
    `|      ,   ,     {}
     \\      /___;    ,'
      ) ,-;`    `\\  //
     | / (        ;||
     ||`\\\\        |||
     ||  \\\\       |||
     )\\   )\\      )||
     `"   `"       `""
```

11. Add and commit the changes on the `snape` branch with the commit message "add snape's doe patronus"
12. Next, create a new branch based upon the `snape` branch called `lily`
13. Move to the `lily` branch
14. Edit the `patronus.txt` file so that it says `LILY'S PATRONUS` at the top instead of `SNAPE'S PATRONUS` (leave the doe ascii-art alone)
15. Add and commit the change with the message "add lily's doe patronus"
16. Run a git command to list all branches (you should see 4)
17. **Bonus:** delete the `snape` branch (poor Snape)

Merging and Conflicts

What Is Merging in Git?

Merging is the process of combining changes from one branch into another.

It is commonly used to:

- Bring feature work into `main`
- Combine parallel development
- Finalize completed changes

Merging integrates **branch histories**, not just files.

Fast-Forward Merge

What Is a Fast-Forward Merge?

A **fast-forward merge** happens when:

- The target branch has no new commits
- Git can simply move the branch pointer forward

No merge commit is created.

Example

```
main ── A ── B
              \
               C ── D  (feature)
```

Merge `feature` into `main`:

```
git checkout main
git merge feature
```

Result:

```
main ── A ── B ── C ── D
```

Git just **moves the pointer**.

IMPORTANT:

- We merge branches, not specific commits
- We always merge the current HEAD branch
- Switch to the branch you want to merge the changes into
- Use `git merge` command to merge changes from a specific branch into the current branch.

Merge Commit (Non–Fast-Forward Merge)

When Does Git Create a Merge Commit?

A **merge commit** is created when:

- Both branches have new commits

- Histories have diverged

Git combines both histories explicitly.

Example Command

```
git merge feature
```

Git creates:

- A new commit
- With two parents
- Representing the merge point

Why Merge Commits Matter

Merge commits:

- Preserve full branch history
- Show where branches were combined
- Are useful for auditing and reviews

They make collaboration history explicit.

Git Merging Exercise

Start by creating a new repo. Make a file or two in the repo for you to work on.

If you need some inspiration...I'll be working in a file called `greetings.txt` It will contain greetings in different languages.

Part 1: Fast Forward Merge

Your goal is to generate a fast forward merge. Demonstrate that you understand how FF merges work by creating one on your own!

Make a new branch. Do some work in the repo such that when you merge the new branch into master, it results in a fast forward merge. Merge that branch into master and see if you were right!

Part 2: Merge Commit (No Conflicts)

Your goal is to generate a merge commit with NO MERGE CONFLICTS.

Create a new branch. Make some changes to the repo such that when you merge the new branch into master, it results in a merge commit. The merge should not result in any conflicts. Merge that branch into master and see if you were right!

Part 3: Conflicts!

Your goal is to generate merge a conflict!

Create a new branch. Make some changes to the repo such that when you merge the new branch into the master branch, it results in a merge conflict. Merge that branch into master and see if you were right! Resolve the conflict!

Understanding `git diff`

What Is `git diff`?

`git diff` shows **what has changed** in your project.

It compares:

- File content between states

- Lines added, removed, or modified

It answers the question:
“What exactly did I change?”

Why `git diff` Is Important

`git diff` helps you:

- Review changes before committing
- Avoid committing mistakes
- Understand how files evolved
- Debug issues by inspecting differences

It is a **read-only** command — it never changes files.

Basic Usage — Working Directory vs Staging Area

```
git diff
```

This shows:

- Changes in the **working directory**
- That are **not yet staged**

Use this before `git add`.

Viewing Staged Changes

```
git diff --staged
```

(or `git diff --cached`)

This shows:

- Changes already added to the staging area

- What will go into the next commit

Use this before `git commit`.

Comparing with the Last Commit

```
git diff HEAD
```

This shows:

- All changes since the last commit
- Both staged and unstaged differences

Useful to review overall progress.

Comparing Between Commits

```
git diff commit1 commit2
```

This shows:

- Differences between two checkpoints
- Useful for debugging or code reviews

Commits are identified using commit hashes.

Comparing Branches

```
git diff main feature
```

This shows:

- What changes the `feature` branch introduces
- Compared to `main`

Very useful before merging branches.

How to Read `git diff` Output

Example:

```
- console.log("Hello")
+ console.log("Hello World")
```

Meaning:

- `-` line was removed
- `+` line was added

Unchanged lines are omitted for clarity.

File-Level Summary (Quick View)

```
git diff --stat
```

This shows:

- Which files changed
- Number of lines added and removed
- No detailed line-by-line output

Good for high-level review.

Common Beginner Mistakes

- Expecting `git diff` to show committed changes (it doesn't by default)
- Forgetting `--staged` for staged files
- Not reviewing diffs before committing

When to Use `git diff`

Use it:

- Before staging (`git add`)
- Before committing
- Before merging
- During debugging

Working with `git stash`

What Is `git stash`?

`git stash` temporarily saves **uncommitted changes** so you can:

- Switch branches safely
- Work on something urgent
- Come back and continue later

It acts like a **clipboard for your working directory**.

Why Do We Need `git stash`?

You need `git stash` when:

- You're in the middle of work
- Changes are not ready to commit
- You must switch branches immediately

Without stashing:

- Git may block branch switching
- You risk losing or mixing changes

What `git stash` Saves

By default, `git stash` saves:

- Modified tracked files
- Staged changes

It does **not** save:

- Untracked files (unless specified)

- Ignored files

Saving Changes to Stash

Basic Stash

```
git stash
```

This:

- Saves changes
- Cleans the working directory
- Leaves you with the last committed state

Stash with a Message

```
git stash save "WIP: navbar redesign"
```

This helps identify the stash later.

Applying Stashed Changes

`git stash pop`

```
git stash pop
```

This:

- Applies the most recent stash
- Removes it from the stash list

Use when you are done with temporary work.

`git stash apply`

```
git stash apply
```

This:

- Applies the stash
- Keeps it in the stash list

Useful when:

- You want to reuse the stash
- You are unsure about conflicts

Working with Multiple Stashes

Git supports **multiple stashes**.

List all stashes:

```
git stash list
```

Example output:

```
stash@{0}: WIP on main: navbar update
stash@{1}: WIP on feature-login: form validation
```

Each stash has an index.

Applying a Specific Stash

```
git stash apply stash@{1}
```

This applies a specific stash without removing it.

Dropping a Stash

Remove a specific stash:

```
git stash drop stash@{0}
```

Remove the most recent stash:

```
git stash drop
```

Cleaning All Stashes

Remove all stashes:

```
git stash clear
```

⚠️ This action is permanent.

Handling Conflicts While Stashing

- Conflicts may occur when applying a stash
- Git marks conflicts like a merge
- You must resolve them manually

After resolving:

```
git add .
```

Best Practices

- Use stash for short-term work
- Add meaningful stash messages
- Don’t use stash as long-term storage
- Clear old stashes regularly

`git checkout` a Commit

What Does “Checkout a Commit” Mean?

Checking out a commit means:

- Moving your working directory to a **specific point in history**
- Viewing the project exactly as it was at that commit

You are **not switching branches**—you are visiting a past snapshot.

Why Checkout a Commit?

You checkout a commit to:

- Inspect old code
- Debug when an issue was introduced
- Compare behavior between versions
- Verify what existed at a specific time

This is **read-only exploration**, not normal development.

Basic Command

```
git checkout <commit-hash>
```

Example:

```
git checkout a1b2c3d
```

Git updates:

- Files in your working directory
- `HEAD` to point directly to that commit

Detached HEAD State (Important)

After checking out a commit:

- `HEAD` is **detached**
- You are not on any branch

This means:

- New commits won't belong to a branch
- They can be lost if you don't save them

Git will warn you about this state.

What You Can and Should Do in Detached HEAD

Safe actions:

- Read code
- Run the project
- Compare files
- Use `git diff`

Avoid:

- Making long-term commits
- Doing feature development

Returning to a Branch

To go back to normal development:

```
git checkout main
```

(or any branch name)

This:

- Reattaches `HEAD`
- Restores the latest branch state

Creating a Branch from a Commit (Safe Practice)

If you want to keep changes from a past commit:

```
git checkout -b hotfix-from-old-commit
```

This:

- Creates a new branch at that commit
- Attaches `HEAD` to the new branch
- Makes commits safe

`git checkout` vs `git switch`

- `git checkout <commit>` → used for commits
- `git switch` → used for branches only

For commits, `git checkout` is still required.

Visual Model

```
main —— A —— B —— C

                ^

            HEAD (checkout commit)
```

`HEAD` points directly to a commit, not a branch.

Common Mistakes

- Forgetting you are in detached HEAD

- Making commits without creating a branch
- Panicking when files “disappear” (they haven’t)

Git is just showing a different snapshot.

Understanding `git restore`

What Is `git restore`?

`git restore` is used to **undo changes** in your working directory or staging area.

It allows you to:

- Discard unwanted changes
- Revert files to a previous state
- Unstage files safely

It does **not** affect commit history.

Why `git restore` Exists

Earlier, `git checkout` was used for:

- Switching branches
- Restoring files

This caused confusion.

`git restore` was introduced to:

- Clearly handle file restoration
- Separate file operations from branch operations

Restoring a Modified File

Scenario

You modified a file but want to discard the changes.

```
git restore index.html
```

This:

- Reverts `index.html` to the last committed version
- Discards all unstaged changes

⚠️ This action cannot be undone.

Restoring All Modified Files

```
git restore .
```

This restores:

- All modified tracked files
- Back to their last committed state

Unstaging a File

If a file is staged by mistake:

```
git restore --staged index.html
```

This:

- Removes the file from the staging area
- Keeps the changes in the working directory

Restoring from a Specific Commit

To restore a file from a previous commit:

```
git restore --source=<commit-hash> index.html
```

This replaces the file with its version from that commit.

Common Use Cases

Use `git restore` when:

- You want to discard local changes
- You staged the wrong file
- You need a file from an older commit

What `git restore` Does NOT Do

- It does not delete commits

- It does not rewrite history
- It does not affect other branches

It operates at the **file level only**.

Visual Flow

```
Commit → restore → Working Directory
```

Un-staging Changes with `git restore`

Un-staging means:

- Removing changes from the **staging area**
- Keeping the changes in your **working directory**

So you're not losing work — you're just saying:

> "Don't include this in the next commit."

The Command to Un-stage

```
git restore --staged <file>
```

Example

You staged a file by mistake:

```
git add index.html
```

Check status:

```
git status
```

You'll see something like:

```
Changes to be committed:
  index.html
```

Now un-stage it:

```
git restore --staged index.html
```

Check again:

```
git status
```

Now it shows:

```
Changes not staged for commit:
  index.html
```

Meaning:

- It's no longer staged
- Your edits are still there

Un-stage Multiple Files

```
git restore --staged file1 file2
```

Un-stage Everything

```
git restore --staged .
```

This removes *all* staged changes, but keeps them as modified in the working directory.

Undoing Commits with `git reset`

What `git reset` Does

`git reset` moves the current branch pointer (and usually `HEAD`) **back to an earlier commit**.

You use it to "undo commits" locally by:

- Removing commits from the branch history (rewriting history)
- Optionally keeping or discarding the actual file changes

⚠️ **Reset rewrites history**. Avoid using it on commits that are already pushed/shared.

Three Reset Modes (Most Important Part)

Git reset has three main modes, and each answers a different question:

`--soft` → "Undo commit, keep changes staged"

- Commit is removed
- Changes remain in the **staging area**
- You can recommit immediately (maybe with a better message)

```
git reset --soft HEAD~1
```

Use when:

- Commit message is wrong
- You want to squash/recommit quickly

`--mixed` (default) → "Undo commit, keep changes unstaged"

- Commit is removed
- Changes remain in the **working directory**
- Files are **not staged**

```
git reset HEAD~1
# same as:
git reset --mixed HEAD~1
```

Use when:

- You committed too early
- You want to re-stage selectively

`--hard` → "Undo commit and delete changes"

- Commit is removed
- Changes are discarded completely
- Working directory is reset to the target commit

```
git reset --hard HEAD~1
```

Use only when:

- You are 100% sure you don't need the changes

Example Scenario (Undo Last Commit)

You made a commit you regret:

```
git commit -m "Add feature X"
```

Option A: Keep changes staged (soft)

```
git reset --soft HEAD~1
```

Now:

- Commit is undone
- Changes are still staged
- You can do:

```
git commit -m "Better message"
```

Option B: Keep changes but unstaged (mixed)

```
git reset HEAD~1
```

Now:

- Commit is undone
- Changes are in working directory
- You can stage selectively:

```
git add file1
git commit -m "Commit only file1"
```

Option C: Delete everything (hard)

```
git reset --hard HEAD~1
```

Now:

- Commit is undone
- Changes are gone

Reset to a Specific Commit

If you want to reset to a particular commit:

```
git reset --mixed <commit-hash>
```

or:

```
git reset --hard <commit-hash>
```

How to Verify After Reset

Use:

```
git status
git log --oneline
```

- `git log` confirms commits moved back
- `git status` confirms where your changes are (staged/unstaged/clean)

Critical Safety Rule

✅ Safe:

- Reset on local commits you have not pushed

❌ Risky:

- Reset after pushing to a shared branch
- Because others may already have those commits.

If it's already pushed, prefer **`git revert`** (history-safe).

Key Takeaways

- `git reset` “undoes commits” by moving the branch pointer
- `--soft` keeps changes staged
- `--mixed` keeps changes unstaged (default)
- `--hard` deletes changes
- Don’t reset shared/pushed history

Undoing Commits with `git revert`

What Is `git revert`?

`git revert` is used to **undo a commit safely** by creating a **new commit**.

Instead of deleting history, it:

- Reverses the changes introduced by a commit
- Preserves the commit history
- Is safe for shared and pushed branches

This makes `git revert` ideal for collaboration.

Why Use `git revert`?

Use `git revert` when:

- A commit is already pushed
- Others may be using the branch
- You want to undo changes without rewriting history

It is the **safe alternative** to `git reset`.

How `git revert` Works

- Git analyzes the selected commit
- Creates a new commit that does the opposite
- History remains intact and readable

Nothing is deleted.

Basic Usage

```
git revert <commit-hash>
```

Example:

```
git revert a1b2c3d
```

Result:

- A new commit is created
- The changes from `a1b2c3d` are undone

Reverting the Most Recent Commit

```
git revert HEAD
```

This:

- Undoes the last commit
- Creates a new “revert” commit

Reverting Multiple Commits

```
git revert commit1 commit2
```

Or a range:

```
git revert HEAD~3..HEAD
```

Each revert creates a separate commit.

Handling Conflicts During Revert

Conflicts may occur if:

- The code has changed since the commit
- Git cannot apply the reverse cleanly

Resolution steps:

- Fix conflicts manually
- Stage the resolved files
- Continue the revert:

```
git revert --continue
```

To cancel:

```
git revert --abort
```

`git revert` vs `git reset`

Feature	git revert	git reset
Rewrites history	❌ No	✅ Yes
Creates new commit	✅ Yes	❌ No
Safe for pushed commits	✅ Yes	❌ No
Collaboration-friendly	✅ Yes	❌ Risky

Common Mistakes

- Expecting `git revert` to delete commits
- Using `git reset` on shared branches
- Forgetting that revert creates a new commit

Key Takeaways

- `git revert` undoes changes safely
- History is preserved
- Best for shared branches
- Preferred undo method in teams

Final Rule to Remember

> **Reset rewrites history. Revert records history.**

Undoing Things Exercise

This exercise is based around the Beatles' song Yesterday, and the evolution of its lyrics over time.

- Clone the repo I've created for you, using the following command (make sure you're not already in a repo when you run it!)

git clone <https://github.com/Colt/yesterday-exercise>

- Change into the newly created directory. You should see a file called `lyrics.txt`
- Take a look at the commit history. You should see 8 commits on the master branch.
- Go "back in time" by checking out the very first commit. Remember, this puts you in detached HEAD. Take a look at the `lyrics.txt` file to verify things have changed. Then, leave detached HEAD by returning to the master branch.
- Go back to the commit where the original version of the lyrics was completed. The commit has the message "finish original lyrics". Create a new branch called `scrambled-eggs` based upon this commit.
- Go back to the `master` branch
- Delete everything in the `lyrics.txt` file. Save the file.
- Oh no, that was a mistake! USING A GIT COMMAND, discard the changes you made to `lyrics.txt` since the last commit. We saw a couple of ways of achieving this.

Suddenly you get a creative itch! You want to write your own parody lyrics! Replace the contents of `lyrics.txt` with the following lyrics (from this website)

```
 404
Guess this ain't the page you're looking for
On this website there are thousands more
With no error 404

Suddenly
This is not the page you thought you'd see
But it's not an error 403
Yes, 404s come easily
```

- Add and commit the changes on the `master` branch
- Add this to the bottom of the `lyrics.txt` file:

```
    Why it has to show, I don't know

Or what it's for
You typed something wrong?
```

```
Here's no song - it's 404

404
Adding 1% to twenty score
(I put that line in 'cause it scans, no more)
"Goodbye" from error 404
```

- Add and commit the changes to the `master` branch

- After more consideration, you realize that you actually don't want the previous two commits (the 404 parody lyrics) on the master branch. You want to move them to a new branch!

- Use a git command to "undo" the prior two commits on master. Make sure you **KEEP THE CHANGES IN YOUR WORKING DIRECTORY.** Just undo the two git commits. Your `lyrics.txt` file should still contain the 404 lyrics.

- Create a new branch called 404, switch to it, and add and commit the 404 lyrics in your working directory.

- Switch back to master and make sure `lyrics.txt` contains the actual lyrics to Yesterday

Understanding GitHub

GitHub is an **online platform** that hosts Git repositories and enables people to **collaborate on projects**.

GitHub does **not replace Git**. Instead, it provides a **central place on the internet** where Git repositories can live and be shared.

Why GitHub Exists

Git works locally on your computer.
But software development is rarely a solo activity.

GitHub exists to solve problems like:

- Sharing code with others
- Collaborating with teams
- Reviewing changes before merging
- Maintaining a single source of truth online

GitHub turns Git from a **local tool** into a **collaboration system**.

GitHub as a Remote Repository

A GitHub repository is a **remote Git repository**.

This means:

- It stores the same Git history as your local repo
- It lives on GitHub's servers
- It can be accessed by others (based on permissions)

Your local Git repository and GitHub repository stay in sync using Git commands.

How Git and GitHub Work Together

Typical workflow:

- You write code locally
- Git tracks changes and commits
- You push commits to GitHub
- Others pull or review your changes

Git handles:

- Version control
- History
- Branching and merging

GitHub handles:

- Sharing
- Collaboration
- Visibility

Collaboration on GitHub

GitHub provides structured collaboration features.

Pull Requests

- Propose changes from one branch to another
- Enable discussion and review before merging
- Enforce quality checks

Code Reviews

- Comment on specific lines of code

- Suggest improvements
- Approve or request changes

Issues

- Track bugs
- Plan features
- Discuss tasks and ideas

These features are essential for team-based development.

GitHub for Open Source

GitHub is the largest platform for **open-source projects**.

It allows anyone to:

- View public code
- Fork repositories
- Contribute improvements
- Learn from real-world projects

Many popular technologies are developed openly on GitHub.

GitHub for Personal Use

For individuals, GitHub is often used to:

- Store personal projects
- Back up code online
- Build a public portfolio
- Demonstrate skills to recruiters

Your GitHub profile often reflects:

- Code quality
- Consistency
- Learning progress

GitHub for Teams and Organizations

Companies use GitHub to:

- Manage multiple repositories
- Control access levels
- Enforce review workflows

- Protect critical branches

GitHub supports:

- Small teams
- Large enterprises
- Global collaboration

Public vs Private Repositories

GitHub repositories can be:

- **Public** → anyone can view
- **Private** → restricted access

Private repositories are commonly used for:

- Company projects
- Confidential code
- Client work

GitHub Is Not the Only Platform

GitHub is popular, but alternatives exist:

- GitLab
- Bitbucket
- Azure Repos

All of them:

- Use Git internally
- Offer similar collaboration features

If you understand Git, switching platforms is easy.

Common Beginner Confusions (Clarified)

- GitHub does not store versions by itself → Git does
- GitHub is not required to use Git
- GitHub authentication is separate from Git configuration
- GitHub does not automatically track your files

Understanding this prevents early confusion.

Understanding `git clone`

`git clone` is used to **create a local copy of an existing Git repository**.

It:

- Downloads the entire repository
- Includes all commits, branches, and history
- Sets up a connection to the remote repository

This is usually the **first step** when working with an existing project.

Why Do We Use `git clone`?

You use `git clone` when:

- A project already exists on a remote platform like **GitHub**
- You want to start working on that project locally
- You want a full copy of the project and its history

Instead of starting from scratch, you clone the repository.

What Happens When You Clone a Repository?

When you run `git clone`:

- A new folder is created
- Git initializes a repository inside it
- All files and commit history are downloaded
- A remote connection called `origin` is set automatically

You are ready to work immediately.

Basic Syntax

```
git clone <repository-url>
```

Example:

```
git clone https://github.com/user/project.git
```

This creates:

```
project/
 ├── .git/
 ├── files...
```

Cloning via HTTPS vs SSH

HTTPS Clone

```
git clone https://github.com/user/project.git
```

- Easier for beginners
- Requires username/password or token

SSH Clone

```
git clone git@github.com:user/project.git
```

- Requires SSH key setup
- More convenient for frequent use

Both achieve the same result.

Cloning into a Custom Folder Name

```
git clone https://github.com/user/project.git my-folder
```

This:

- Clones the repository
- Saves it in `my-folder` instead of the default name

Cloning a Specific Branch (Optional)

```
git clone -b feature-login https://github.com/user/project.git
```

This:

- Clones the repository
- Checks out the specified branch immediately

Understanding `git remote` and `git remote -v`

What Is a Git Remote?

A **Git remote** is a **reference to a remote repository**.

It tells Git:

- Where your remote repository lives (URL)
- Where to push changes
- Where to fetch updates from

Most commonly, this remote repository is hosted on **GitHub**.

Why Git Remotes Are Needed

Git works locally by default.
Remotes allow Git to:

- Share code with others
- Sync changes between machines
- Collaborate with teams

Without remotes, your work stays only on your computer.

The Default Remote: `origin`

When you clone a repository:

```
git clone <repo-url>
```

Git automatically:

- Creates a remote named `origin`
- Links it to the cloned repository URL

`origin` is just a **name**, not a keyword.

Viewing Configured Remotes — `git remote`

```
git remote
```

This command:

- Lists all remote names
- Does not show URLs

Example output:

```
origin
```

Meaning:

- One remote is configured
- Its name is `origin`

Viewing Remote URLs — `git remote -v`

```
git remote -v
```

This shows:

- Remote names
- Their URLs
- Which URLs are used for fetch and push

Example output:

```
origin  https://github.com/user/project.git (fetch)
origin  https://github.com/user/project.git (push)
```

Understanding Fetch vs Push URLs

- **Fetch URL** → where Git downloads changes from
- **Push URL** → where Git uploads changes to

Most of the time:

- Both URLs are the same

They can be different in advanced setups.

Adding a New Remote (Optional)

To add a remote manually:

```
git remote add upstream https://github.com/other/project.git
```

Now:

- `origin` → your fork
- `upstream` → original repository

This is common in open-source workflows.

Renaming a Remote

```
git remote rename origin main-origin
```

Useful when:

- Managing multiple remotes
- Clarifying repository roles

Removing a Remote

```
git remote remove origin
```

This:

- Deletes the remote reference
- Does NOT delete any code
- Does NOT affect local commits

Common Confusions (Cleared)

- `git remote -v` does NOT contact GitHub
- It only shows saved configuration
- Removing a remote does NOT delete the repository online
- Remote names are local aliases

Understanding `git push`

`git push` is used to **send your local commits to a remote repository**.

It:

- Uploads commits from your local branch
- Updates the corresponding branch on the remote
- Makes your work visible to others

Until you push, your commits exist **only on your machine**.

Why `git push` Is Needed

Git is a **distributed system**:

- You commit locally
- Git does not automatically sync with the remote

You need `git push` to:

- Share your work with teammates
- Back up your commits online
- Trigger collaboration workflows (PRs, reviews, CI)

Without pushing, collaboration cannot happen.

Basic Syntax

```
git push <remote> <branch>
```

Example:

```
git push origin main
```

This means:

- Push commits
- From local `main`
- To remote `origin/main`

What Happens During `git push`

When you run `git push`:

- Git checks which commits the remote does not have
- Sends only the missing commits
- Updates the remote branch pointer

Git never re-sends existing history.

First Push to a New Repository

When pushing a branch for the first time:

```
git push -u origin main
```

The `-u` (or `--set-upstream`) flag:

- Links local `main` to `origin/main`
- Allows future pushes with just `git push`

After this, you can simply run:

```
git push
```

Pushing a New Branch

If you create a new branch locally:

```
git switch -c feature-login
```

Push it:

```
git push -u origin feature-login
```

Now:

- The branch exists on the remote
- Others can fetch or review it

Push Rejected (Very Common Case)

You may see an error like:

```
! [rejected] main -> main (fetch first)
```

This happens when:

- The remote branch has new commits
- Your local branch is behind

Fix:

```
git pull
git push
```

Git requires your branch to be **up to date** before pushing.

Force Push (Dangerous)

```
git push --force
```

This:

- Overwrites remote history
- Deletes commits on the remote

Use only when:

- You fully understand the consequences
- Working on your own branch
- Fixing rewritten history intentionally

❗ Never force-push to shared branches like `main`.

Push vs Commit (Important Distinction)

Action	Scope
`git commit`	Local only
`git push`	Sends commits to remote

Github Basics Exercise

1. Create a new repository locally on your machine.
2. Create a new Github repository. Name it whatever you want.
3. Connect your local repo to the Github repo.
4. Optional: rename the default branch from master to main.
5. Make a new file called `favorites.txt` Leave it empty. Make your first commit on the `main` branch.
6. Push up your `main` branch to Github! Make sure you see your empty `favorites.txt` file on Github.
7. Next, create two branches: `foods` and `movies`
8. Switch to the `foods` branch. Add three (or more) of your favorite foods to the `favorites.txt` file. Add and commit your changes on the `foods` branch.
9. Switch to the `movies` branch and add three or more of your favorite movies to the `favorites.txt` file. Add and commit your changes on the movies branch.
10. Push up your `foods` branch to Github. Make sure you see it on Github!
11. Push up your `movies` branch to Github. Make sure you see it on Github!
12. Merge the `foods` branch into the `main` branch. Then merge the `movies` branch into the `main` branch. If necessary, resolve conflicts so that you end up with your favorite foods and favorite movies in the same `favorites.txt` file.
13. Push up the latest work on your `main` branch to Github.

Understanding `git fetch` and `git pull`

Why Fetch and Pull Exist

In Git:

- Your **local repository** and **remote repository** are independent
- Other people can push changes at any time

You need a way to:

- See what changed on the remote
- Bring those changes into your local work

That's where **fetch** and **pull** come in.

What Is `git fetch`?

`git fetch` **downloads changes from the remote repository**
but **does not modify your working directory**.

It:

- Retrieves new commits
- Updates remote-tracking branches
- Keeps your current branch untouched

Think of it as **"check for updates"**.

Basic Usage — `git fetch`

```
git fetch
```

This:

- Contacts the remote (usually `origin`)
- Downloads new data
- Updates branches like `origin/main`

Your files do **not** change.

What Happens After `git fetch`

After fetching:

- You can inspect remote changes safely
- Your local branch may still be behind
- You decide when and how to integrate changes

Example:

```
git log origin/main
```

This lets you **review before merging**.

When to Use `git fetch`

Use `git fetch` when:

- You want to see what others pushed
- You want full control before merging
- You're in the middle of work
- You want to avoid surprises

Fetch is **safe and non-destructive**.

What Is `git pull`?

`git pull` is a **shortcut command**.

It does:

1. `git fetch`
2. followed by a merge (or rebase)

In other words:

```
git pull = fetch + merge
```

Basic Usage — `git pull`

```
git pull
```

This:

- Fetches remote changes
- Automatically merges them into your current branch
- Updates your working directory

Your files **will change**.

Example Scenario

Remote has new commits:

```
origin/main —— A —— B
local/main  —— A
```

Run:

```
git pull
```

Result:

```
local/main —— A —— B
```

Your branch is now up to date.

Fetch vs Pull (Critical Difference)

Command	What It Does
`git fetch`	Downloads changes only
`git pull`	Downloads **and merges** changes
Risk	Very low
File changes	❌ No

Merge Conflicts During `git pull`

A pull can cause conflicts when:

- You modified the same code
- Remote changes overlap with yours

Git will:

- Pause the pull
- Ask you to resolve conflicts
- Require a commit to finish

This is why many teams prefer **fetch first**.

Recommended Best Practice

Safe workflow:

```
git fetch
git status
git diff origin/main
git merge origin/main
```

Quick workflow:

```
git pull
```

Choose based on:

- Confidence
- Complexity of changes
- Team standards

Pull with Rebase (Advanced)

```
git pull --rebase
```

This:

- Replays your commits on top of remote changes
- Keeps history cleaner
- Avoids merge commits

Often used in professional teams.

Common Beginner Mistakes

- Using `git pull` without checking changes
- Panicking when conflicts appear
- Assuming fetch updates files (it doesn't)

Key Takeaways

- `git fetch` downloads changes safely
- `git pull` fetches **and merges**
- Fetch gives control, pull gives speed
- Pull can cause conflicts
- Understanding both is essential for teamwork

Final Rule to Remember

> **Fetch to see. Pull to apply.**

Understanding Rebasing

What Is Rebasing in Git?

Rebasing is a way to **move or replay commits from one branch onto another**.

Instead of merging histories, rebasing:

- Takes commits from one branch
- Re-applies them on top of another branch
- Creates a **linear commit history**

Why Rebasing Exists

Rebasing is used to:

- Keep history clean and linear
- Avoid unnecessary merge commits
- Make commit history easier to read
- Prepare branches before merging

It is mainly about **history cleanliness**, not functionality.

Rebase vs Merge (Core Difference)

Merge

- Combines two histories

- Creates a merge commit
- Preserves branch structure

Rebase

- Rewrites commit history
- Moves commits to a new base
- No merge commit (usually)

Visual Comparison

Before Rebase

```
main     A —— B
                \
feature           C —— D
```

After Rebase

```
main     A —— B —— C' —— D'
```

- `C` and `D` are replayed as `C'` and `D'`
- Commit IDs change

Basic Rebase Command

```
git rebase main
```

Run this **from the feature branch**.

Meaning:

- Take my commits

- Replay them on top of `main`

Step-by-Step Example

You are on `feature-login`:

```
git switch feature-login
git rebase main
```

Result:

- Your feature commits now sit on top of latest `main`
- History looks linear
- No merge commit created

What Happens During a Rebase

During rebasing:

- Git replays commits one by one
- Conflicts may occur
- Git pauses and asks you to resolve them

This is similar to merge conflicts, but happens **per commit**.

Resolving Conflicts During Rebase

If a conflict occurs:

- Fix the conflict manually

- Stage the file

Then continue:

```
git rebase --continue
```

To abort:

```
git rebase --abort
```

`git merge` vs `git rebase`

Both **merge** and **rebase** are used to **integrate changes from one branch into another**.

The difference is **how Git handles history**.

`git merge` — Preserve History

What `git merge` Does

`git merge`:

- Combines two branches
- Preserves their individual histories
- Usually creates a **merge commit**

It shows **when and where branches were joined**.

Visual Example (Merge)

Before merge:

```
main      A ── B
                \
feature           C ── D
```

After merge:

```
main      A ── B ── M
                /    \
feature         C ── D
```

`M` = merge commit

When to Use Merge

Use merge when:

- Working on shared branches
- Collaborating with a team
- You want a complete, honest history
- You don't want to rewrite commits

Merge is **safe by default**.

`git rebase` — Rewrite History

What `git rebase` Does

`git rebase`:

- Takes commits from one branch
- Replays them on top of another branch
- Creates a **linear history**
- Changes commit IDs

History looks cleaner, but is rewritten.

Visual Example (Rebase)

Before rebase:

```
main     A —— B
               \
feature          C —— D
```

After rebase:

```
main     A —— B —— C' —— D'
```

`C'` and `D'` are **new commits**.

When to Use Rebase

Use rebase when:

- Working on your own local branch
- Preparing a branch before merging
- Cleaning commit history
- You want linear logs

Never rebase shared history.

Key Differences at a Glance

Aspect	`git merge`	`git rebase`
History	Preserved	Rewritten
Commit IDs	Unchanged	Changed
Merge commit	Yes (usually)	No
Safety	Very safe	Risky if misused
Team usage	Recommended	Avoid on shared branches
Log readability	More complex	Linear and clean

Conflict Handling Differences

Merge Conflicts

- Happen once during merge
- Resolved in one step

Rebase Conflicts

- Can happen **per commit**
- Require resolving multiple times

What Happens to `HEAD`

- **Merge** → `HEAD` moves forward with a merge commit
- **Rebase** → `HEAD` moves as commits are replayed

Practical Rule for Students (Very Important)

If the branch is shared → use MERGE
If the branch is local → REBASE is okay

Common Beginner Mistakes

- Rebasing after pushing
- Rebasing `main`
- Using rebase to “fix” mistakes in shared branches
- Thinking rebase is “better” than merge (it’s not)

Mental Model to Remember

- **Merge** = “Bring the branch in as it is”

- **Rebase** = “Pretend my work started later”

Batch Exercise Workflow: Branch → Code → Push → PR → Pull Main

Repository Setup (You do this once)

Repo name suggestion: `java-basics-batch-exercises`

Project skeleton (simple Maven is ideal):

```
java-basics-batch-exercises/
  README.md
  pom.xml
  src/main/java/com/batch/exercises/
```

Add a `README.md` section “How to contribute” (you can paste the workflow below).

Participant Workflow (everyone follows this)

1) Clone the repo

```
git clone <repo-url>
cd java-basics-batch-exercises
```

2) Create a new branch (naming convention)

Use your name + exercise id:

```
git switch -c <firstname>-ex<NN>
# example: git switch -c vishal-ex03
```

3) Pull latest main before coding

```
git switch main
git pull
git switch <your-branch>
```

4) Create ONE new Java class for your assigned exercise

Location:
`src/main/java/com/batch/exercises/`

Class naming convention:
`ExerciseNN_<ShortName>.java`
Example: `Exercise03_FizzBuzzPlus.java`

5) Code + comments requirements

Inside the class, include:

- **Problem statement** (comment header)
- **Approach** (short comment)
- Inline comments only where helpful
- A `main()` method for demo input/output (no need for Scanner if you prefer hardcoded examples)

6) Check status → add → commit

```
git status
git add .
git commit -m "Add solution for Exercise NN: <ShortName>"
```

7) Push branch

```
git push -u origin <your-branch>
```

8) Create PR to `main` on GitHub

PR title format:
`Exercise NN - <Your Name>`

PR description should include:

- What you implemented
- Example output (copy-paste console output)

9) After PR is merged → everyone syncs

```
git switch main
```

```
git pull
```

Rules (to prevent chaos)

- One person = one exercise = one java file.
- Don’t modify other people’s files.
- Don’t modify `pom.xml` unless instructed.
- Keep code readable and commented.
- If your PR shows merge conflicts: **pull latest main and resolve**, then push again.