GIT & GITHUB SERIES · COURSE 1

Git & GitHub Foundations

Commits, branching, pull requests, undoing mistakes, and two real-world
scenarios — moving a project safely, and surviving your first conflict.

10 Chapters

osztromok.com

CHAPTER 1: WHAT GIT & GITHUB ARE

COURSE 1 · CH 1

What Git & GitHub Are — and Why They Matter

The difference between git and GitHub, installing git on your machine, and setting up a GitHub account

Before writing a single command, it's worth being completely clear on one point that trips up almost every beginner: **git** and **GitHub** are not the same thing. Git is the tool that tracks changes to your files. GitHub is a website that hosts copies of git projects online and adds collaboration features on top. You can use git without ever touching GitHub — but GitHub is what makes git genuinely useful for working with other people, or simply backing your work up somewhere other than your own hard drive.

Git vs GitHub

📦 Git

A **version control system** — a program that runs on your own computer. It records snapshots of your project over time (called **commits**), so you can see exactly what changed, when, and revert to any previous point if something breaks.

Git works entirely offline. You could use git for the rest of your life and never create a GitHub account — every project would just live on your own machine, with full history.

🌐 GitHub

A **website and service** that hosts git projects ("repositories") online. It adds things git itself doesn't have: a visual interface, issue tracking, pull requests for reviewing changes, and a place for other people to find and contribute to your code.

GitHub is one option among several — GitLab and Bitbucket are direct competitors that also host git repositories with similar features. This series focuses on GitHub since it's by far the most widely used.

A USEFUL ANALOGY

Git is like the autosave-with-full-history feature of a word processor — running locally, tracking every version of your document. GitHub is like uploading that document to a shared drive where colleagues can see it, comment on it, and propose their own edits for you to accept or reject.

Why Use Version Control at All?

If you've ever ended up with files named `project_final.zip`, `project_final_v2.zip`, and `project_final_v2_ACTUALLY_FINAL.zip`, you've already invented a crude form of version control — manually, and without any way to see exactly what changed between them. Git solves the same problem properly:

- **Full history, automatically.** Every commit is a complete, named snapshot — no more guessing which zip file had the working version.
- **See exactly what changed.** Git can show you the precise lines added, removed, or modified between any two points in history.
- **Safe experimentation.** Branches (covered in Chapter 4) let you try something risky without touching your working version at all.
- **Collaboration without overwriting each other.** Multiple people can work on the same project and git helps combine everyone's changes safely — this is the subject of several scenario chapters later in this series.
- **A backup that lives off your machine.** Once pushed to GitHub, your project survives a hard drive failure, a lost laptop, or — as covered in Chapter 9 — a folder move gone wrong.

Installing Git

Git is free, open-source, and available on every major operating system. Installation takes a couple of minutes.

🪟 Windows

Download the installer from `git-scm.com` and run it. The defaults are fine for almost everyone — the one setting worth checking is the default editor for commit messages (Vim is the default; switch it to Notepad or VS Code if you're not comfortable with Vim yet).

This installs **Git Bash**, a terminal that behaves like a Linux/Mac shell — the commands throughout this series will work identically there.

🍎 macOS

Easiest route: install Homebrew, then run `brew install git`. Alternatively, installing Xcode Command Line Tools (`xcode-select --install`) bundles a working git as well, though typically an older version.

Linux

Use your distribution's package manager — `sudo apt install git` on Debian/Ubuntu, `sudo dnf install git` on Fedora. Git is so fundamental that most distros offer it as a single, simple package.

Verifying the install

```
$ git --version
git version 2.45.1
```

Any version starting with 2.x is fine for everything in this series. If the command isn't recognised, restart your terminal — installers sometimes need a fresh shell to pick up the new PATH entry.

One-time setup — telling git who you are

Every commit records an author name and email. Set these once, globally, and every project on your machine will use them automatically:

```
$ git config --global user.name "Your Name"
$ git config --global user.email "you@example.com"
# Verify it took effect:
$ git config --global user.name
Your Name
```

USE THE SAME EMAIL AS YOUR GITHUB ACCOUNT

If the email in your git config doesn't match an email registered to your GitHub account, your commits won't be linked to your GitHub profile — they'll still work, but won't show your avatar or count toward your contribution graph. Worth setting correctly from the start.

Setting Up Your GitHub Account

1. **Create an account at github.com**

   Pick a username carefully — it appears in every repository URL you create and is genuinely awkward to change later once you have projects, stars, and followers attached to it.

2. **Verify your email address**

   Required before you can create repositories or interact with most features. Use the same address you set in `git config --global user.email` above where possible.

3. **Enable two-factor authentication**

   Settings → Password and authentication → Two-factor authentication. GitHub accounts are a common target for credential theft because of what they grant access to — an authenticator app (not SMS) is the recommended method.

4. **Fill in your profile (optional, but worthwhile)**

   A profile photo, bio, and a pinned repository or two make your account look active and trustworthy — relevant the moment you start collaborating with anyone else.

YOU DON'T NEED TO CONNECT GIT TO GITHUB YET

Creating an account is enough for this chapter. Chapter 5 covers connecting your local git installation to GitHub properly — SSH keys, personal access tokens, and your first `git push`. For now, the goal is just having both pieces installed and ready.

CHAPTER 1 QUICK REFERENCE

- **Git** = local version control tool, runs entirely on your machine, works offline
- **GitHub** = a website that hosts git projects online and adds collaboration features
- **Check install:** `git --version`
- **One-time identity setup:** `git config --global user.name` / `user.email`
- **Use the same email** in git config as your GitHub account for commits to link to your profile
- **GitHub account setup:** verified email → two-factor authentication (authenticator app, not SMS) → basic profile
- **Next chapter:** core git concepts — working directory, staging area, and your first commit

CHAPTER 2: CORE CONCEPTS

COURSE 1 · CH 2

Core Concepts — Working Directory, Staging Area, Commits

git init, git add, git commit — the three commands behind every single thing git does

Almost everything in git, no matter how advanced, is built on three commands you'll use constantly: `git init`, `git add`, and `git commit`. This chapter is entirely about understanding what each one actually does — not just memorising the syntax, but understanding the three-stage model underneath them, because that model is what makes the rest of git make sense.

The Three Stages

Every file in a git project exists in one of three places at any given moment. Understanding this is the single most important concept in the entire series — branching, merging, and almost every git command is really just moving files between these three areas.

Working Directory

The actual files on your disk, as you see them in your editor or file explorer. Edits here aren't tracked by git until you explicitly tell it to look.

`(just edit files normally)`

→

Staging Area

A holding area for changes you've decided should go into the *next* commit. Lets you build a commit out of exactly the changes you want, even if you've edited several unrelated files.

`git add <file>`

→

Repository (History)

A permanent, named snapshot of everything in the staging area at that moment. Once committed, that snapshot exists forever in the project's history.

`git commit -m "message"`

WHY A SEPARATE STAGING AREA AT ALL?

It feels like an extra step at first, but it's genuinely useful: if you've fixed a bug *and* made unrelated formatting changes in the same session, you can stage and commit just the bug fix first, then stage and commit the formatting separately — keeping your history clean and each commit focused on one thing.

Starting a New Project — git init

`git init` turns an ordinary folder into a git repository. It creates a hidden `.git` folder inside it — that hidden folder *is* the entire repository: every commit, every branch, the full history. Delete `.git` and you've deleted the repository (the files in your working directory remain, but all history is gone — this is exactly the situation we ran into during the website→debserver reorg, where a `.git` folder went missing and broke a nested repository link).

```
$ mkdir my-project
$ cd my-project
$ git init
Initialized empty Git repository in /home/you/my-project/.git/
```

ONE REPOSITORY PER PROJECT, NOT NESTED

Running `git init` inside a folder that's already inside a git repository creates a nested repository — git will treat the inner folder strangely, often showing it as a single unexplained "modified" entry rather than its actual file changes. If you only ever run `git init` once per project, at the top level, this never comes up.

Checking Status — your most-used command

`git status` tells you, at any moment, which files are in the working directory only, which are staged, and which haven't changed since the last commit. You will run this command more than any other — get comfortable reading its output.

```
$ git status
On branch master
No commits yet

Untracked files:
(use "git add <file>..." to include in what will be committed)
index.html
style.css
```

"Untracked" means git has noticed these files exist but isn't watching them for changes yet — they haven't been staged or committed.

Staging Changes — git add

`git add` moves a file's current state from the working directory into the staging area. It's a snapshot of that file *right now* — if you edit it again after staging, you need to run `git add` again to update what's staged.

```
$ # Stage one specific file
$ git add index.html

$ # Stage everything that's changed
$ git add .

$ git status
Changes to be committed:
new file: index.html
new file: style.css
```

BE DELIBERATE WITH `GIT ADD .`

`git add .` stages every changed file in the current directory and below — convenient, but it can accidentally stage files you didn't mean to commit (temporary files, local config, accidentally-saved secrets). `git add <specific-file>` or reviewing with `git status` first is the safer habit, especially on a shared project. Chapter 6 covers `.gitignore`, which prevents this class of mistake entirely for files that should never be tracked.

Recording a Snapshot — git commit

`git commit` takes everything currently in the staging area and saves it permanently as a new point in the project's history, along with a message describing what changed and why.

```
$ git commit -m "Add initial homepage and stylesheet"
[master (root-commit) a1b2c3d] Add initial homepage and stylesheet
2 files changed, 48 insertions(+)
```

```
 create mode 100644 index.html
 create mode 100644 style.css
```

That `a1b2c3d` is the commit's unique ID (a SHA-1 hash) — every commit gets one, and it's how git refers to that exact snapshot anywhere else in the system (covered more in Chapter 8 when undoing commits, and Chapter 3 when viewing history).

Writing a good commit message

- **Summarise what changed, in the imperative mood:** "Add login form" not "Added login form" or "Adding login form" — this matches git's own generated messages (merge commits, reverts) and reads cleanly in history.
- **Keep the first line under ~50 characters** where practical — many tools (including GitHub's interface) truncate longer summaries.
- **One logical change per commit.** "Fix login bug and update homepage copy" should usually be two commits, not one — it keeps history searchable and makes problems easier to isolate later.
- **For more detail, omit `-m` entirely** — running plain `git commit` opens your configured editor, where you can write a short summary line, a blank line, then a longer explanation.

Command Reference So Far

COMMAND	WHAT IT DOES
`git init`	Turns the current folder into a new git repository
`git status`	Shows what's changed, staged, or untracked right now
`git add <file>`	Moves a file's current state into the staging area
`git add .`	Stages all changed files in the current directory and below
`git commit -m "msg"`	Permanently records everything staged as a new snapshot

A REPOSITORY WITH ZERO COMMITS IS PERFECTLY NORMAL

After `git init`, you'll see "No commits yet" in `git status` — that's expected, not an error. The repository exists and is tracking the folder; it simply has no history yet until your first commit.

CHAPTER 2 QUICK REFERENCE

- **Three stages:** Working Directory → Staging Area → Repository (History)
- **git init** — creates the hidden .git folder; run once, at the project's top level
- **git status** — your most-used command; shows untracked, staged, and unstaged changes
- **git add <file>** — stage a specific file; **git add .** — stage everything (use deliberately)
- **git commit -m "message"** — permanently records the staged snapshot
- **Commit messages:** imperative mood, under ~50 chars first line, one logical change per commit
- **Next chapter:** viewing your history — log, diff, status, and show in depth

CHAPTER 3: VIEWING HISTORY

COURSE 1 · CH 3

Viewing History — log, diff, status, show

Reading the story your commits already tell — and learning to actually trust what git is showing you

Once you have more than one commit, the real value of git starts to show: a complete, searchable record of exactly what changed, when, and why. This chapter covers the four commands you'll use to read that history — `git log`, `git diff`, `git status`, and `git show` — each answering a slightly different question about your project's past.

git log — The Story So Far

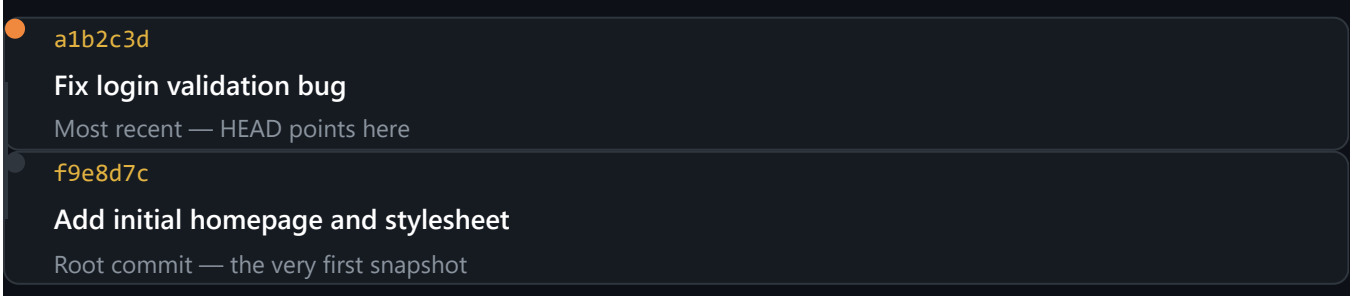
`git log` lists every commit in the current branch's history, most recent first. Each entry shows the commit hash, author, date, and message.

```
$ git log
commit a1b2c3d4e5f6...
Author: Philip Osztromok <you@example.com>
Date: Fri Jun 19 14:02:11 2026

Fix login validation bug

commit f9e8d7c6b5a4...
Author: Philip Osztromok <you@example.com>
Date: Fri Jun 19 11:30:45 2026

Add initial homepage and stylesheet
```



Useful log variations

```
$ # Compact one-line-per-commit view — the one you'll use most
$ git log --oneline
a1b2c3d Fix login validation bug
f9e8d7c Add initial homepage and stylesheet

$ # Visual branch graph — essential once you start branching (Ch4)
$ git log --oneline --graph --all

$ # Only the last 5 commits
$ git log -5

$ # Only commits touching one specific file
$ git log -- index.html
```

> **GIT LOG --ONELINE --GRAPH --ALL IS WORTH ALIASING**
>
> Most experienced git users set up a shortcut for this exact combination early on — it's the single most useful view once branches enter the picture. `git config --global alias.lg "log --oneline --graph --all"` lets you just type `git lg` from then on.

git diff — What Actually Changed

`git diff` shows line-by-line changes. Used without arguments, it shows changes in your working directory that *haven't* been staged yet — exactly what would be added if you ran `git add` right now.

```
$ git diff
```

```
diff --git a/index.html b/index.html
@@ -12,7 +12,7 @@
  <body>
-    <h1>Welcome</h1>
+    <h1>Welcome to my site</h1>
  </body>
```

Red lines (prefixed `-`) were removed; green lines (prefixed `+`) were added. Lines with no prefix are unchanged context, shown so you can see where the change sits.

Diff variations that matter

```
$ # Changes already staged, not yet committed
$ git diff --staged

$ # Difference between two specific commits
$ git diff a1b2c3d f9e8d7c

$ # Difference for one specific file only
$ git diff index.html
```

> **PLAIN GIT DIFF VS GIT DIFF --STAGED TRIPS UP ALMOST EVERYONE AT FIRST**
>
> Plain `git diff` shows *unstaged* changes only. If you've already run `git add`, those changes disappear from plain `git diff` output — not because they vanished, but because they moved to the staging area. Use `git diff --staged` to see them there instead.

git status — Revisited

Chapter 2 introduced `git status` for checking what's staged. With history now in the picture, it also tells you how your branch relates to the last commit and to any remote (covered in Chapter 5):

```
$ git status
On branch master
Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git restore <file>..." to discard changes in working directory)
modified: index.html
```

`git status` and `git diff` work well together: `status` tells you *which files* changed, `diff` tells you *exactly what* changed inside them.

git show — Inspecting One Specific Commit

`git show` displays the full details of a single commit — message, metadata, and the complete diff of everything that commit changed, in one view.

```
$ git show a1b2c3d
commit a1b2c3d4e5f6...
Author: Philip Osztromok <you@example.com>
Date: Fri Jun 19 14:02:11 2026

Fix login validation bug
```

```
diff --git a/login.js b/login.js
@@ -8,6 +8,6 @@
-  if (password.length > 6) {
+  if (password.length >= 8) {
```

Useful git show variations

```
$ # The most recent commit, without typing its hash
$ git show HEAD

$ # One commit before the most recent
$ git show HEAD~1

$ # Show only the file list changed, not the full diff
$ git show --stat HEAD
```

HEAD IS JUST A POINTER TO "WHERE YOU ARE RIGHT NOW"

`HEAD` always refers to your current commit — usually the tip of whichever branch you're on. `HEAD~1` means "one commit before that," `HEAD~2` means two before, and so on. This shorthand works in almost any git command that expects a commit, and saves having to copy-paste hashes for recent history.

Command Reference

COMMAND	WHAT IT DOES
`git log`	Full commit history, most recent first
`git log --oneline`	Compact one-line-per-commit view
`git log --graph --all`	Visual branch graph across all branches
`git diff`	Unstaged changes in the working directory
`git diff --staged`	Changes already staged, not yet committed
`git status`	What's staged, unstaged, or untracked right now
`git show <commit>`	Full details and diff for one specific commit
`HEAD / HEAD~1 / HEAD~2`	Shorthand for "current commit" and N commits before it

CHAPTER 3 QUICK REFERENCE

- **git log** — full history; **git log --oneline** for the compact view you'll use daily
- **git diff** — unstaged changes only; **git diff --staged** for what's already staged
- **git status** — quick check of what's changed before deciding what to stage/commit
- **git show <commit>** — full message + diff for one specific commit
- **HEAD** — your current commit; **HEAD~1**, **HEAD~2** — N commits back from there
- **Good habit:** run status → diff → log in that order when you're not sure what state a project is in
- **Next chapter:** branching — creating, switching, and merging, and what a branch actually is under the hood

CHAPTER 4: BRANCHING BASICS

COURSE 1 · CH 4

Branching Basics

What a branch really is, creating and switching between them, and merging your work back together

Branches are the feature that makes git genuinely powerful rather than just "version control with extra steps." A branch lets you work on something — a new feature, an experiment, a fix — completely isolated from your main project, then bring it back in safely once it's ready. This chapter demystifies what a branch actually is under the hood, because the mental model matters more than the commands themselves.

What a Branch Actually Is

This surprises most beginners: **a branch is just a movable label pointing at a commit.** That's it. It's not a copy of your files, not a separate folder, not a snapshot of the whole project — just a lightweight named pointer. This is exactly why creating a branch in git is instant, even on huge projects, while in older version control systems branching could take minutes and gigabytes of disk space.

🏷️ A branch = a label

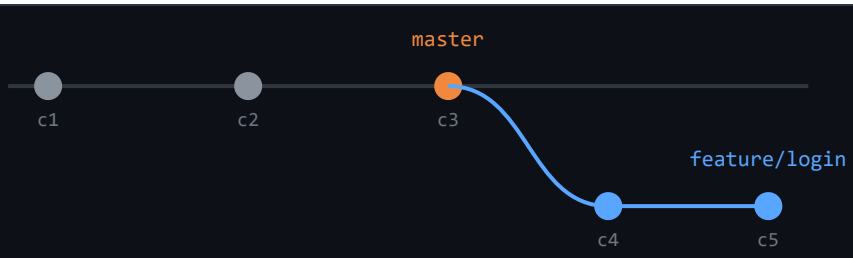
When you create a branch, git just writes a small file containing a commit hash. Switching branches moves that label, and updates your working directory to match whatever that commit looked like.

📍 HEAD = "where you are"

`HEAD` (from Chapter 3) is itself usually just a pointer to whichever branch label you currently have checked out. Switch branches, and HEAD moves to point at the new one.

➡️ The label moves automatically

Every time you commit while on a branch, that branch's label automatically moves forward to point at the new commit. You never move it manually — committing does it for you.



master stays at c3 while feature/login moves ahead through c4 and c5 — both branches share the same history up to c3.

Creating and Switching Branches

```
$ # Create a new branch (doesn't switch to it yet)
$ git branch feature/login

$ # List all branches — * marks the current one
$ git branch
feature/login
* master

$ # Switch to it
$ git switch feature/login
Switched to branch 'feature/login'

$ # Create AND switch in one step — the one you'll actually use most
$ git switch -c feature/login
```

GIT SWITCH VS THE OLDER GIT CHECKOUT

You'll see `git checkout feature/login` in a lot of older tutorials and Stack Overflow answers — it does the same job. `git switch` is newer (Git 2.23+) and was specifically introduced because `checkout` did too many unrelated things (switching branches, restoring files, detaching HEAD) bundled into one confusing command. Both still work; `switch` is the clearer choice going forward.

Naming branches well

- **Use a consistent prefix:** `feature/login-form`, `fix/header-overflow`, `docs/readme-update` — makes branch lists scannable at a glance, especially once a few accumulate.

- **Keep them short-lived where possible.** A branch that lives for months drifts further from `master` every day, making the eventual merge harder — Chapter 9 covers what happens when this goes wrong across a team.
- **No spaces, use hyphens.** Git branch names can't contain spaces; hyphens or underscores are the convention.

Doing Work on a Branch

Once switched, everything behaves exactly as in Chapters 2–3 — edit files, `git add`, `git commit` — except every commit you make extends *this* branch specifically, leaving `master` completely untouched in the meantime.

```
$ (feature/login) git add login.js
$ (feature/login) git commit -m "Add password strength validation"
$ (feature/login) git switch master
Switched to branch 'master'
# login.js reverts to how it looked on master — the feature/login changes are still safe, just not here
```

Merging — Bringing Branches Back Together

Once work on a branch is ready, `git merge` combines it back into another branch — typically, you switch to the destination branch first, then merge the feature branch into it.

```
$ (master) git switch master
$ (master) git merge feature/login
Updating f9e8d7c..a1b2c3d
Fast-forward
login.js | 4 ++++
1 file changed, 4 insertions(+)
```

MERGE TYPE	WHEN IT HAPPENS	WHAT YOU'LL SEE
Fast-forward	master hasn't moved at all since the branch was created — git just moves master's label forward	"Fast-forward" in the output, no new merge commit created

MERGE TYPE	WHEN IT HAPPENS	WHAT YOU'LL SEE
Three-way merge	master gained other commits while your branch was being worked on — git combines both histories	A new "merge commit" is created, with two parent commits
Merge conflict	both branches changed the *same lines* of the *same file* — git can't decide automatically	Git pauses and asks you to resolve it manually (full coverage in Course 2, Chapter 3)

FAST-FORWARD MERGES ARE THE EASY CASE — DON'T EXPECT EVERY MERGE TO LOOK LIKE THIS

The fast-forward example above only happens because nothing else changed on `master` while `feature/login` was being developed. The moment two branches have genuinely diverged, git needs an actual three-way merge, and if the same lines were touched by both, a conflict. This is completely normal and not a sign anything went wrong — Course 2 dedicates a full chapter to resolving conflicts confidently.

Cleaning up after a merge

```
$ # Once merged, the branch has done its job — delete it
$ git branch -d feature/login
Deleted branch feature/login (was a1b2c3d).
```

Deleting a merged branch only removes the label — every commit it pointed to is still in history, reachable from `master`. Nothing is lost; you're just tidying up branch names you no longer need.

Command Reference

COMMAND	WHAT IT DOES
`git branch`	List branches; current one marked with *
`git branch <name>`	Create a new branch (doesn't switch to it)
`git switch <name>`	Switch to an existing branch
`git switch -c <name>`	Create and switch to a new branch in one step

COMMAND	WHAT IT DOES
`git merge <name>`	Merge the named branch into your current branch
`git branch -d <name>`	Delete a branch (only its label — commits remain in history)

CHAPTER 4 QUICK REFERENCE

- **A branch is just a movable label** pointing at a commit — not a copy of your files
- **HEAD** points to whichever branch you currently have checked out
- **git switch -c <name>** — create and switch in one step (the one you'll use most)
- **git merge <name>** — run from the destination branch, merging the named branch in
- **Fast-forward** = simple case, no diverging history; **three-way merge** = both branches moved; **conflict** = same lines changed in both
- **git branch -d** — safe to delete a merged branch; its commits remain in history
- **Next chapter:** connecting to GitHub — remotes, push/pull/clone, and your first real backup off your own machine

CHAPTER 5: CONNECTING TO GITHUB

COURSE 1 · CH 5

Connecting to GitHub

Remotes, push/pull/clone, and choosing between SSH keys and personal access tokens

Everything so far has happened entirely on your own machine. This chapter connects that local repository to GitHub — turning it from a private history only you can see into something backed up, shareable, and ready for the collaboration features covered throughout the rest of this series.

What a "Remote" Actually Is

A **remote** is simply a saved URL pointing to a copy of your repository hosted somewhere else — almost always GitHub for this series, but it could be GitLab, Bitbucket, or even another folder on your own machine. Your local repository and the remote are two independent copies of the same history; git's job is keeping them in sync via `push` and `pull`.

"ORIGIN" IS JUST A NICKNAME, NOT A SPECIAL KEYWORD

By convention, the main remote is named `origin` — but that's purely a naming convention, not a git requirement. You could name it `github`, `upstream`, or anything else. Almost every tutorial and tool assumes `origin`, so it's worth sticking with unless you have a specific reason not to (Course 2, Chapter 4 covers fork workflows where a second remote, conventionally `upstream`, becomes genuinely useful).

Two Ways to Start — New Repo vs Existing Repo

Option A: You already have a local project (from Chapters 1–4)

```
$ # 1. Create an empty repository on github.com first (no README, no .gitignore — keep it empty)
$ # 2. Link your local repo to it
$ git remote add origin https://github.com/yourname/my-project.git
$ # 3. Push your existing history up
$ git push -u origin master
```

Option B: Starting from a repository that already exists on GitHub

```
$ git clone https://github.com/yourname/my-project.git
Cloning into 'my-project'...
remote: Enumerating objects: 12, done.
Receiving objects: 100% (12/12), done.
```

`git clone` downloads the full history and automatically sets up `origin` for you — no `git init` or `git remote add` needed. This is by far the most common way most people start working on an existing project.

Authenticating with GitHub — SSH vs Personal Access Tokens

GitHub no longer accepts your account password for git operations over HTTPS — you need either an SSH key pair or a personal access token (PAT). Both work; they suit slightly different situations.

🔑 SSH Keys

A key pair generated once on your machine. The **private key** stays on your computer, never shared; the **public key** is uploaded to your GitHub account. Once set up, authentication is automatic — no typing credentials, ever.

Best for: your own personal machine, used long-term. Slightly more setup effort upfront, but the smoothest experience afterward.

🎟️ Personal Access Tokens

A long random string that acts like a password, generated from GitHub's settings, used over HTTPS. Can be scoped to specific permissions and given an expiry date.

Best for: temporary or shared machines, CI/CD pipelines, or quick one-off setups where generating an SSH key feels like overkill.

Setting up SSH (the recommended approach for your own machine)

1. **Generate a key pair**

   `ssh-keygen -t ed25519 -C "you@example.com"` — accept the default file location, set a passphrase if you want extra protection.

2. **Copy the public key**

   The public key is the file ending in `.pub` (e.g. `id_ed25519.pub`) — never copy or share the file without `.pub`, that's your private key.

3. **Add it to GitHub**

   Settings → SSH and GPG keys → New SSH key. Paste the public key content in.

4. **Use the SSH URL, not HTTPS, when cloning**

   `git clone git@github.com:yourname/my-project.git` — GitHub shows both URL formats on every repository page; pick the SSH one.

> **ALREADY USING HTTPS AND TIRED OF TYPING YOUR TOKEN EVERY TIME?**
>
> Run `git config --global credential.helper store` (or `manager` on Windows, often set up automatically) to cache your token after the first use, instead of switching to SSH. Both authentication methods are completely valid long-term choices — this is about convenience, not correctness.

Pushing and Pulling

```
$ # Send your local commits up to GitHub
$ git push

$ # Bring down commits made elsewhere (another machine, a collaborator)
$ git pull
```

`git push` works only if your local branch is at or ahead of the remote's version — if someone else has pushed commits you don't have locally yet, git will reject the push and ask you to pull first. This exact scenario — and what to do about it — is the subject of Chapter 10's two-person collaboration scenario.

NEVER COMMIT SECRETS, EVEN TO A PRIVATE REPOSITORY

API keys, passwords, and `.env` files pushed to GitHub remain in history forever, even if you delete them in a later commit — anyone with repo access (or anyone who later finds a leaked private repo) can dig them out of old commits. Chapter 6 covers `.gitignore`, which prevents this category of mistake from happening in the first place.

Command Reference

COMMAND	WHAT IT DOES
`git remote add origin <url>`	Link an existing local repo to a GitHub repository
`git remote -v`	Show the URL(s) your remotes point to
`git clone <url>`	Download a full copy of a remote repo, with origin already set up
`git push`	Send local commits to the remote
`git push -u origin master`	Push and remember this remote/branch as the default for future pushes
`git pull`	Fetch and merge remote commits into your local branch

CHAPTER 5 QUICK REFERENCE

- **A remote** is just a saved URL to another copy of your repository — usually GitHub
- **origin** is the conventional name for your main remote, not a special keyword
- **git clone** sets up origin automatically; **git remote add** is for connecting an existing local repo
- **SSH keys** — best for your own long-term machine, no repeated credential prompts once set up
- **Personal access tokens** — best for temporary/shared machines and CI pipelines, scoped and expirable
- **git push / git pull** — send and receive commits; push fails if the remote has commits you don't have yet
- **Never commit secrets** — they persist in history even after deletion in a later commit

- **Next chapter:** .gitignore — exactly what should never be tracked, and how to enforce it automatically

CHAPTER 6: .GITIGNORE

COURSE 1 · CH 6

.gitignore — What Never to Commit

Secrets, build artifacts, and OS junk files — and how to stop them from ever being tracked in the first place

Not every file in a project folder belongs in git history. Some are generated automatically and can be recreated anytime; some are specific to your own machine; some are actively dangerous to publish. A `.gitignore` file tells git which files and folders to never even consider — they won't show up in `git status`, won't get staged by `git add .`, and won't accidentally end up in a commit.

What Should Never Be Tracked

🔐 Secrets & credentials

API keys, passwords, database connection strings, `.env` files. The single most important category — covered in detail below.

📦 Dependencies

`node_modules/`, Python's `venv/`, vendor folders. Huge, regenerable from a lockfile (`package-lock.json`, `requirements.txt`) — tracking them bloats the repo for no benefit.

🏗️ Build output

`dist/`, `build/`, compiled `.class` or

🖥️ OS & editor files

`.DS_Store` (macOS), `Thumbs.db` (Windows), `.vscode/` or

`.pyc`

files. Generated from source you already have tracked — committing the output as well just creates redundant, conflict-prone noise.

`.idea/`

personal editor settings. Specific to your machine, meaningless to anyone else cloning the repo.

📝 Logs & temp files

`*.log`

,

`*.tmp`

, cache directories. Generated during normal operation, churn constantly, and add no value to history.

💾 Large binary/media files

Large video files, database dumps, disk images. Git handles these poorly by default — bloats repo size dramatically (Course 3, Chapter 7 covers Git LFS for cases where you genuinely need to track large files).

Creating a .gitignore File

A `.gitignore` file is plain text, placed at the root of your repository, with one pattern per line. Git reads it automatically — no command needed to "enable" it.

```
# .gitignore — one pattern per line
node_modules/
.env
dist/
*.log
.DS_Store
```

Pattern syntax

PATTERN	MATCHES
`node_modules/`	The folder named node_modules, anywhere in the project (trailing slash = directory only)
`*.log`	Any file ending in .log, anywhere in the project
`.env`	A file named exactly .env in the root (and in any subfolder, since no leading slash)
`/build`	A folder named build, but only at the project root — leading slash anchors it
`temp/*`	Everything directly inside temp/, but not the temp/ folder itself

PATTERN	MATCHES
`!important.log`	An exception — un-ignores a file that an earlier pattern would otherwise have matched

> **USE GITIGNORE.IO OR GITHUB'S TEMPLATES INSTEAD OF WRITING FROM SCRATCH**
>
> gitignore.io generates a ready-made `.gitignore` for your exact stack (Node, Python, Java, VS Code, macOS, etc.) — paste in your tools and it builds the file for you. GitHub also offers starter templates when creating a new repository. Hand-writing one from memory is rarely necessary.

The .env File — Special Attention Required

Almost every modern project stores secrets (database passwords, API keys) in a `.env` file that gets loaded into environment variables at runtime. This file should **always** be in `.gitignore` — but the project still needs to communicate what variables are expected. The standard pattern:

```
.env.example ← tracked: shows variable names, no real values
.env ← ignored: contains your actual secrets
```

```
# .env.example — committed to the repo, safe to share
DATABASE_URL=postgres://user:password@localhost/dbname
API_KEY=your-api-key-here

# .env — never committed, real secrets, listed in .gitignore
DATABASE_URL=postgres://admin:Tr0ub4dor&3@prod-db.example.com/myapp
API_KEY=sk-live-a1b2c3d4e5f6g7h8i9j0
```

Anyone cloning the repo copies `.env.example` to `.env` and fills in their own real values — the structure is shared, the secrets never are.

.gitignore Only Works on Untracked Files

This catches almost everyone out at least once: adding a pattern to `.gitignore` does **nothing** for a file git is already tracking. If a file was committed before it was added to `.gitignore`, git keeps tracking it regardless.

```
$ # If config.json was already committed before being added to .gitignore:
$ git rm --cached config.json
rm 'config.json'
# Now it's untracked — .gitignore will take effect from here on
$ git commit -m "Stop tracking config.json"
```

`git rm --cached` removes the file from git's tracking *without deleting it from your disk* — exactly what you want when un-tracking a file you still need locally.

A SECRET COMMITTED IN THE PAST STAYS IN HISTORY FOREVER

`git rm --cached` stops tracking a file going forward, but every *previous* commit that included it still has it, fully recoverable by anyone with access to the repository's history — including a public repo that was briefly public, even if made private again later. If a real secret was ever committed, the correct response is to treat it as compromised: rotate/revoke the credential immediately. Rewriting history to remove it (covered in Course 3, Chapter 2) is good hygiene but does not undo the exposure on its own — assume it's been seen.

Command Reference

COMMAND	WHAT IT DOES
`.gitignore`	A file listing patterns git should never track, untracked files only
`git rm --cached <file>`	Stop tracking a file without deleting it from disk
`git check-ignore -v <file>`	Debug which .gitignore rule is (or isn't) matching a specific file

CHAPTER 6 QUICK REFERENCE

- **Never track:** secrets/.env, dependencies (node_modules), build output, OS/editor junk, logs, large binaries
- **.gitignore** — plain text, one pattern per line, at the repo root, no command needed to enable it
- **Generate, don't hand-write** — use gitignore.io or GitHub's templates for your stack
- **.env pattern:** commit .env.example (no real values), ignore .env (real secrets)
- **.gitignore only affects untracked files** — use **git rm --cached** to stop tracking something already committed

- **A committed secret is compromised the moment it's pushed** — rotate the credential; don't rely on deleting it later to undo the exposure
- **Next chapter:** pull requests 101 — making your first PR, getting reviewed, and merging on GitHub itself

CHAPTER 7: PULL REQUESTS 101

COURSE 1 · CH 7

Pull Requests 101

Making your first PR, understanding review comments, and merging your work on GitHub itself

A pull request (PR) is GitHub's mechanism for proposing changes and getting them reviewed before they land in the main branch. Everything covered in Chapters 4 and 5 — branching, pushing — leads here: a PR is what turns "I pushed a branch" into "this change is reviewed, discussed, and ready to merge."

What a Pull Request Actually Is

Despite the name, a pull request isn't really about "pulling" anything technically new — it's a **request** asking the maintainers of a repository to merge your branch into another branch (almost always into `master` or `main`). It wraps that merge in a conversation: a description, a diff view, comment threads, and a record of who approved it and when.

A PR IS GITHUB'S FEATURE, NOT GIT'S

Plain git has no concept of a "pull request" at all — it's purely a GitHub (and GitLab, and Bitbucket, under different names) feature built on top of ordinary branches and merges. You could do the exact same merge entirely from the command line with `git merge` — the PR just adds the review and discussion layer on top.

Making Your First Pull Request

1. **Create a branch and make your changes**

   `git switch -c fix/typo-homepage`, edit, commit — exactly the workflow from Chapter 4.

2. **Push the branch to GitHub**

   `git push -u origin fix/typo-homepage` — the branch now exists on GitHub, but nothing has been merged yet.

3 **Open the pull request on GitHub**

GitHub usually shows a "Compare & pull request" prompt right after a push. Otherwise: Pull requests tab → New pull request → choose your branch as the source.

4 **Write a clear title and description**

What changed and why — same principles as a good commit message (Chapter 2), but with room for more context, screenshots, or a linked issue.

5 **Request reviewers (if working with others)**

On a solo project this step doesn't apply — but it's exactly where Course 2's code review chapter and team workflows pick up.

Fix typo in homepage hero heading

Open fix/typo-homepage → master · 1 commit · 1 file changed

DESCRIPTION

Fixes "Welcom" → "Welcome" in the hero heading on the homepage. Spotted while reviewing the live site.

Reading and Responding to Review Comments

Reviewers can comment on the PR as a whole, or on specific lines of the diff. Line-level comments are the most useful — they let a reviewer point at exactly the code they're talking about.

Reviewer

Should this be a constant instead of a magic number? Might be worth naming it for clarity.

```
if (retries > 3) { ... }
```

> **↩ You**
>
> Good point — pushed a fix extracting it to `MAX_RETRIES`.

Responding to feedback the right way

- **Push new commits to the same branch** — they automatically appear in the same open PR. No need to close and reopen anything.

- **Reply directly under the relevant comment** rather than a general reply, so the conversation stays attached to the specific code it's about.
- **Mark resolved threads as resolved** once addressed — keeps the PR readable for anyone reviewing later, including the eventual approver.
- **Disagreeing is fine — explain your reasoning** rather than silently ignoring or silently complying with feedback you think is wrong. The discussion thread exists precisely for this.

Merging the Pull Request

Once a PR is approved (or, on a solo project, once you're satisfied with it yourself), GitHub offers three merge strategies — the choice affects what your history looks like afterward, covered in depth in Course 2's branching strategies chapter:

- **Merge commit** — preserves every individual commit from the branch, plus a new merge commit tying them together. Most information-preserving, but can clutter history with many small commits.
- **Squash and merge** — combines all commits on the branch into a single commit on the target branch. Clean history, but loses the individual commit-by-commit story.
- **Rebase and merge** — replays the branch's commits individually onto the target branch with no merge commit at all. Clean and linear, but rewrites commit hashes (relevant once Course 3 covers rebasing in depth).

```
# After merging on GitHub, clean up locally:
$ git switch master
$ git pull
# Brings the merged change down to your local master
$ git branch -d fix/typo-homepage
# Delete the now-merged local branch (Chapter 4)
```

A MERGED BRANCH ON GITHUB DOESN'T DISAPPEAR LOCALLY ON ITS OWN

GitHub offers a "Delete branch" button right after merging a PR — that only removes the remote copy. Your local branch (and the matching local copy on any collaborator's machine) still exists until deleted with `git branch -d`. Harmless to leave around, but tidying up keeps your branch list from accumulating clutter over time.

CHAPTER 7 QUICK REFERENCE

- **A PR** is GitHub's review/discussion layer on top of an ordinary git merge — not a git feature itself
- **Flow:** branch → push → open PR → write a clear title/description → (request review) → merge
- **Pushing new commits to the branch** updates the same open PR automatically
- **Reply to specific line comments** rather than general replies, to keep feedback attached to the relevant code
- **Three merge strategies:** Merge commit (preserves all), Squash (one clean commit), Rebase (linear, no merge commit)
- **After merging:** pull on your local master, then delete your local branch — GitHub's "Delete branch" only removes the remote copy
- **Next chapter:** undoing mistakes — restore, soft reset, revert vs reset, and amend

CHAPTER 8: UNDOING MISTAKES

COURSE 1 · CH 8

Undoing Mistakes — The Beginner Toolkit

git restore, soft reset, revert vs reset, and amend — fixing the four most common "I didn't mean to do that" moments

Almost every beginner's biggest fear with git is "what if I break something I can't undo." The reassuring truth: git is built to make almost everything reversible, as long as you know which tool fixes which mistake. This chapter covers the four you'll reach for constantly — undoing uncommitted changes, undoing staging, undoing commits two different ways, and fixing your most recent commit message.

Which Mistake Do You Have?

📝 "I edited a file and want to throw the changes away"

Changes are still in the working directory, not staged yet.

`git restore <file>`

📋 "I staged a file by mistake, didn't mean to commit it yet"

Already ran `git add`, want to unstage without losing the edit.

`git restore --staged <file>`

💬 "My last commit message has a typo"

Only the most recent commit, not yet pushed/shared.

`git commit --amend`

📦 "I committed something I didn't mean to, want it undone"

Need to remove an entire commit — see **revert vs reset** below, the choice depends on whether it's been pushed.

`git revert / git reset`

git restore — Undoing Uncommitted Changes

`git restore` (introduced alongside `git switch` in Chapter 4 — both replaced overlapping uses of the old `git checkout`) reverts a file in your working directory back to how it looked at

the last commit, discarding any edits since.

```
$ # Discard unstaged edits to one file
$ git restore index.html

$ # Unstage a file (keeps the edit, just removes it from staging)
$ git restore --staged index.html
```

GIT RESTORE (WITHOUT --STAGED) PERMANENTLY DISCARDS THE EDIT

Unlike unstaging, plain `git restore <file>` throws the actual content away — there's no staging-area safety net here, because the change was never committed anywhere git could recover it from. If you're not sure you want to lose the edit, `git diff` (Chapter 3) first to see exactly what you'd be discarding.

git commit --amend — Fixing Your Last Commit

`--amend` replaces your most recent commit entirely — useful for a typo'd message, or for adding a file you forgot to stage before committing.

```
$ # Just fix the message
$ git commit --amend -m "Fix login validation bug"

$ # Forgot to add a file? Stage it, then amend
$ git add forgotten-file.js
$ git commit --amend --no-edit
```

NEVER AMEND A COMMIT YOU'VE ALREADY PUSHED AND SHARED

Amending creates a brand-new commit hash to replace the old one. If anyone else has already pulled the original commit, amending and pushing again creates a conflicting history between your repo and theirs — exactly the kind of situation Course 2's "multiple users on the same branch" scenario chapter exists to help you recover from. Amend freely before pushing; be cautious after.

Revert vs Reset — Undoing Whole Commits

Both undo a commit's effects, but in fundamentally different ways — choosing the wrong one is the single most consequential mistake in this chapter.

COMMAND	WHAT IT DOES	SAFE AFTER PUSHING?
`git revert <commit>`	Creates a **new commit** that undoes the changes from the target commit. History is preserved — both the original mistake and the fix are visible.	✅ Yes — safe even on shared branches
`git reset --soft <commit>`	Moves your branch pointer back, but keeps all the changes staged — as if you'd never committed, but the edits remain.	⚠️ Only if not yet pushed
`git reset --hard <commit>`	Moves your branch pointer back AND discards all changes since, in the working directory too. Most destructive option in this chapter.	❌ Never on shared/pushed work

```
$ # Undo a commit that's already been pushed and shared — safe
$ git revert a1b2c3d
[master 9f8e7d6] Revert "Add experimental feature flag"

$ # Undo a LOCAL-ONLY commit, keep the edits to redo properly
$ git reset --soft HEAD~1

$ # Throw away a local-only commit AND its changes entirely
$ git reset --hard HEAD~1
```

THE SIMPLE RULE: HAS IT BEEN PUSHED?

Not pushed yet, only on your machine → `reset` is fine, rewrite away. Already pushed, possibly pulled by someone else → use `revert` instead, which adds a new commit rather than rewriting history anyone else might be relying on. When genuinely unsure, `revert` is always the safer default.

Recovering from a hard reset mistake

Even `git reset --hard` isn't always unrecoverable — git keeps a record of where your branch pointers have recently been, called the **reflog**. This is genuinely advanced-rescue territory and gets full treatment in Course 3, Chapter 3 ("Reflog and disaster recovery") — but it's worth knowing now that `git reflog` exists as a safety net, even for mistakes that feel catastrophic in the moment.

CHAPTER 8 QUICK REFERENCE

- **git restore <file>** — discard unstaged edits (permanent, no staging-area safety net)
- **git restore --staged <file>** — unstage, keeping the actual edit
- **git commit --amend** — fix the message or contents of your most recent commit (only safe before pushing/sharing)
- **git revert <commit>** — undoes a commit by adding a new one; safe even after pushing
- **git reset --soft** — rewinds the branch pointer, keeps changes staged; local-only commits
- **git reset --hard** — rewinds AND discards changes entirely; most destructive, local-only commits
- **Golden rule:** not pushed yet → reset is fine; already pushed/shared → use revert
- **Safety net:** git reflog can often recover even from a hard reset — full coverage in Course 3
- **Next chapter (Scenario):** moving your project folder or machine — what breaks, and how to do it correctly

CHAPTER 9: SCENARIO - MOVING YOUR PROJECT

COURSE 1 · CH 9 · SCENARIO

Moving Your Project Folder or Machine

What actually breaks when you move, rename, or relocate a git repository — and how to do it without losing anything

Sooner or later, every project gets moved — renamed, reorganised into a new folder structure, or copied to a different computer entirely. Git handles this far better than most tools, but only if you understand what's actually stored where. This chapter walks through the common ways a move goes wrong, and the correct way to do each one.

A REAL EXAMPLE — THIS ACTUALLY HAPPENED

While reorganising a website project folder (renaming it and moving content into a new structure), a nested git repository inside one of the subfolders lost its `.git` directory during the move — but the parent repository still had a recorded reference to it as a submodule-like "gitlink," pointing at a commit that no longer existed anywhere. Running `git status` afterward showed the entire subfolder collapsed into a single confusing `M website` line instead of the actual file changes inside it — because git could no longer see into a folder it expected to be its own repository.

The fix was straightforward once diagnosed: the broken reference simply needed removing from the parent repo's tracking (or the nested `.git` needed restoring, if a backup existed). The real lesson is the diagnosis step — recognising *why* a folder was behaving strangely, rather than assuming something was randomly broken.

What Actually Lives Where

Understanding a move starts with knowing exactly what makes a folder a git repository in the first place — covered back in Chapter 2, but worth restating here because it's the entire key to this chapter: **the hidden `.git` folder is the repository.** Everything else — your actual files — is just the working directory, recreatable from history at any time.

📁 .git/ folder

Every commit, branch, tag, and the full history. This is the only piece that can't be recreated if lost — guard it.

📄 Your actual files

The working directory — fully recoverable from .git/ history at any point, as long as it's been committed at least once.

🔗 The remote URL

Just a saved string inside .git/config — doesn't change on its own when you move a folder, but can become wrong if the remote itself moves.

Scenario: Moving a Folder on the Same Machine

This is the safest move of all, and usually completely uneventful — **as long as you move the entire folder, including the hidden `.git` directory, together, as one unit.**

```
$ # Safe — moves the folder and its .git together, nothing breaks
$ mv old-project-folder/ new-location/project/

$ cd new-location/project
$ git status
On branch master
nothing to commit, working tree clean
```

⚠️ The folder gets reorganised, not just moved

A file explorer "cut and paste" sometimes copies visible files but leaves hidden folders (like `.git`) behind, depending on the OS and settings — exactly what likely happened in the incident above.

Fix: verify .git exists in the new location BEFORE deleting the old folder. If it's missing, the move wasn't actually complete.

⚠️ Only part of the project gets moved

If a subfolder of a larger project (itself its own git repository) gets relocated separately from the rest, the parent's tracking of that subfolder can become inconsistent.

Fix: move nested repositories as complete units, and check the parent repo's status immediately afterward.

Verifying a move went cleanly

1. **Confirm .git is present at the new location**
   `git rev-parse --show-toplevel` from inside the moved folder — if it returns the new path correctly, the repository moved with it.

2 **Run git status and git log**

A clean `git status` and a full, intact `git log` confirm history survived the move completely.

3 **Don't delete the old location until both checks pass**

The single most important habit in this entire chapter — confirm the new copy is genuinely complete before removing the only other copy you have.

Scenario: Moving to a Different Machine

Moving between machines is where people most often reach for risky manual copying. There's almost always a cleaner option already covered in this course:

✅ **Best option: clone fresh**

If the project is already pushed to GitHub, the new machine just needs `git clone` (Chapter 5) — full history, correctly configured remote, zero risk of partial copies.

⚠️ **Riskier: copying the folder manually**

USB drive, cloud sync (OneDrive, Dropbox), or network share copies — works, but risks an incomplete copy if interrupted, and cloud sync tools sometimes mishandle the hidden `.git` folder or large numbers of small files inside it.

CLOUD-SYNCED FOLDERS (ONEDRIVE, DROPBOX, GOOGLE DRIVE) AND GIT CAN CLASH

Cloud sync tools were built for documents, not for the thousands of small internal files git stores inside `.git/objects/`. Sync conflicts, partial uploads, or sync pauses mid-write can corrupt a repository in ways that are hard to diagnose later. If a project lives inside a synced folder, treat GitHub (or another remote) as the real source of truth, and don't rely on the sync tool itself as your backup strategy.

Scenario: Renaming a Repository

Renaming the local folder is identical to moving it — `.git` doesn't care what the containing folder is named. Renaming the GitHub repository itself is different and needs one extra step:

```
$ # After renaming the repo on GitHub's settings page:
$ git remote set-url origin https://github.com/yourname/new-repo-name.git
$ git remote -v
```

```
origin https://github.com/yourname/new-repo-name.git (fetch)
origin https://github.com/yourname/new-repo-name.git (push)
```

GitHub actually redirects the old URL automatically for a good while after a rename — but updating it explicitly avoids relying on that redirect indefinitely, and avoids confusion for anyone else with the repo cloned.

Diagnosing a Broken Move

If something looks wrong after a move, work through this in order rather than guessing:

```
$ # Is this actually still a git repository, and where's its real root?
$ git rev-parse --show-toplevel

$ # Is the remote URL still correct?
$ git remote -v

$ # Is history intact?
$ git log --oneline -5

$ # Is a subfolder behaving like a broken gitlink rather than showing real changes?
$ git ls-files -s <suspicious-folder>
160000 90bafe80c9... 0 suspicious-folder
# Mode 160000 = git is tracking this as a submodule/gitlink, not an ordinary folder
```

That last command is exactly what diagnosed the real incident described at the top of this chapter — mode `160000` in `git ls-files` output is the unambiguous sign that git considers a folder its own separate repository, whether or not that's actually still true on disk.

CHAPTER 9 QUICK REFERENCE

- **The .git folder IS the repository** — everything else is recreatable from it
- **Moving on the same machine:** move the whole folder including .git as one unit; verify before deleting the old copy
- **Moving to a new machine:** prefer git clone over manual copying whenever the project is already on GitHub

- **Cloud-synced folders** (OneDrive/Dropbox) can mishandle .git's many small files — treat a real remote as your source of truth, not the sync tool
- **Renaming on GitHub:** update with git remote set-url origin <new-url> afterward
- **Diagnosing a broken move:** git rev-parse --show-toplevel → git remote -v → git log → git ls-files -s on any suspicious folder
- **Mode 160000** in git ls-files output = git is treating that folder as a separate repository (gitlink) — the exact signature of the broken-move incident in this chapter
- **Next chapter (Scenario):** two people on the same branch — push rejections, simple conflicts, and staying in sync

CHAPTER 10: SCENARIO - TWO PEOPLE, ONE BRANCH

COURSE 1 · CH 10 · FINAL CHAPTER · SCENARIO

Two People on the Same Branch

Push rejections, fast-forward pulls, and resolving your first simple merge conflict — what happens the moment a second person joins your repo

Every chapter so far assumed you're the only person touching the repository. The moment a second person commits to the same branch, a few new situations become possible — none of them are mistakes, all of them are completely normal, and this final chapter of Course 1 walks through exactly what each one looks like and how to handle it calmly.

The Core Rule: Git Won't Let You Silently Overwrite Someone Else's Work

This is the single most important thing to understand before working with anyone else: git is specifically designed to *reject* a push that would erase commits it doesn't recognise. You cannot accidentally destroy a collaborator's work just by pushing — git stops you and asks you to reconcile first.

Scenario A: Someone Else Pushed Before You

	YOU	YOUR COLLABORATOR
09:00	Pull latest `master`, both start at the same commit	Pull latest master, same commit
09:30	Editing `header.css` locally	**Commits and pushes** a fix to footer.css
09:45	Commits header.css, tries to push	—

```
$ git push
! [rejected] master -> master (fetch first)
error: failed to push some refs to 'github.com:you/project.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally.
```

This is not an error in the sense of something being broken — it's git correctly noticing that the remote has a commit (your collaborator's footer fix) that your local branch doesn't have. The fix is simply to pull first:

```
$ git pull
Merge made by the 'ort' strategy.
footer.css | 3 +++
$ git push
To github.com:you/project.git
a1b2c3d..9f8e7d6 master -> master
```

Because you and your collaborator edited *different files* (header.css vs footer.css), git merges them automatically with no conflict at all — this is the most common outcome by far in day-to-day collaboration.

> **PULL BEFORE YOU START WORK, NOT JUST BEFORE YOU PUSH**
>
> Pulling at the start of a working session, not just when a push gets rejected, reduces how often your local branch drifts far from the remote — smaller, more frequent catch-ups produce smaller, easier-to-resolve differences than one big gap.

Scenario B: You Both Edited the Same Lines — a Real Conflict

Sometimes the overlap is real: you and your collaborator both changed the exact same line of the exact same file. Git can't guess which version is correct, so it pauses and asks you to decide.

```
$ git pull
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Opening `index.html` shows git's conflict markers directly in the file:

```
  <header>
<<<<<<< HEAD
    <h1>Welcome to my site</h1>
=======
    <h1>Welcome, friend!</h1>
>>>>>>> 9f8e7d6 (their commit)
  </header>
```

Everything between `<<<<<<< HEAD` and `=======` is your version; everything between `=======` and `>>>>>>>` is theirs. Resolving means manually editing the file to keep whichever version is correct (or a combination of both) and removing the marker lines entirely.

1. **Open the conflicted file and decide what the final version should look like**

   Talk to your collaborator if it's unclear which version (or what combination) is correct — this is a conversation, not just a technical resolution.

2. **Delete the `<<<<<<<`, `=======`, and `>>>>>>>` marker lines**

   Leaving any marker lines in place will break the file — they're not valid HTML/code, just git's way of showing you both versions side by side.

3. **Stage the resolved file and commit**

   `git add index.html` then `git commit` — git pre-fills a "Merge branch..." message; usually fine to keep as-is.

4. **Push the resolved merge**

   A normal `git push` — the conflict is resolved locally, so this completes the sync.

IF YOU PANIC MID-CONFLICT, YOU CAN ALWAYS BACK OUT

`git merge --abort` (or `git pull --abort` if mid-pull) cancels the merge entirely and returns you to exactly where you were before — no harm done. There's no time pressure to resolve a conflict immediately; backing out and asking a collaborator for help is always a safe option.

Preventing Most Conflicts Before They Happen

- **Pull frequently** — at the start of each session, and before pushing, not just when forced to.
- **Use branches for anything non-trivial** (Chapter 4) — working in your own branch and merging via a reviewed PR (Chapter 7) catches most overlaps before they become a tense same-branch conflict.
- **Communicate about who's touching what** — a quick "I'm working on the header right now" message prevents most same-line conflicts before git ever needs to flag one.
- **Commit and push in smaller, more frequent chunks** — large, infrequent commits diverge further from the shared branch and produce bigger, harder conflicts when they finally meet.

> **THIS IS GENUINELY THE EASY VERSION OF THIS PROBLEM**
>
> Everything in this chapter assumes two people, working politely, on small changes. Course 2's "multiple users on the same branch" scenario chapter covers what happens at team scale — diverged history, force-push dangers, and branch protection rules — and Course 3 covers recovering when a history rewrite goes wrong across a team. This chapter is the foundation those build on.

CHAPTER 10 QUICK REFERENCE — AND COURSE 1 WRAP-UP

- **Push rejected ("fetch first"):** the remote has commits you don't — just **git pull**, then push again
- **Different files changed:** git merges automatically, no conflict, no manual intervention needed
- **Same lines changed:** a real conflict — git inserts `<<<<<<<` / `=======` / `>>>>>>>` markers for you to resolve manually
- **Resolving:** edit to the correct final version, delete all marker lines, git add, git commit, git push
- **Stuck or unsure?** git merge --abort backs out completely, no harm done — ask a collaborator if needed
- **Prevent conflicts:** pull often, use branches for non-trivial work, communicate, commit in smaller chunks
- **Course 1 recap:** install & setup (Ch1) → commits (Ch2) → history (Ch3) → branching (Ch4) → GitHub (Ch5) → .gitignore (Ch6) → pull requests (Ch7) → undoing mistakes (Ch8) → moving projects (Ch9) → collaboration basics (Ch10)

- **Next:** Course 2 — Git & GitHub for Teams (branching strategies, merge vs rebase, fork workflows, CI basics, and team-scale scenarios)