# Open Source Mein Pehla Qadam — Fork Se Pull Request Tak

Join a real project: fork, clone your fork, add the upstream remote, branch, change, push and open a pull request on someone else's repo.

## Introduction

Everything so far happened on YOUR repos. **Open source** flips it: the project belongs to someone else, you have no write permission — and yet you can still fix their bug and have your name in their history forever. The bridge is the **fork** from G.4 plus the **pull request** from G.5, joined by one new idea: your fork keeps a link to the original repo, called **upstream**, so you can pull the project's latest work into your copy before you propose changes back. This final lesson walks the whole journey once. *(Roman Urdu: Open source mein aap kisi aur ke project mein hissa lete hain — fork aap ki apni copy hai, pull request aap ki darkhwast.)*

## Key points

- **You cannot push to a repo you do not own — and you do not need to:** the fork gives you a copy you fully own. All your pushes go to the fork; the original receives only your pull request.
- **Two remotes, two jobs:** after cloning your fork, `origin` = YOUR fork (push here). You add `upstream` = the ORIGINAL repo (pull from here). Keeping upstream's main flowing into your fork is what stops your PR from being built on stale code.

  *(Roman Urdu: origin mera fork, upstream asal project — push origin ko, taaza kaam upstream se.)*
- **Read the project's rules before touching code:** most repos carry a `CONTRIBUTING.md` and open **issues** labelled `good first issue` — the maintainers' own list of beginner-sized tasks, with the project's style rules attached.
- **One PR, one purpose — and claim the issue first:** a comment like "I would like to work on this" stops two strangers duplicating the same fix. Small, focused PRs with a polite description are what maintainers merge.
- **Your PR is a portfolio piece forever:** merged or not, it is public proof of real collaboration — employers genuinely read these.

## Code examples

### Example 1 — Fork, clone YOUR fork, add upstream

```
$ git clone https://github.com/ayesha-khan/notes-app.git
Cloning into 'notes-app'...
Receiving objects: 100% (24/24), done.
$ cd notes-app
$ git remote add upstream https://github.com/codescope/notes-app.git
$ git remote -v
origin    https://github.com/ayesha-khan/notes-app.git (fetch)
origin    https://github.com/ayesha-khan/notes-app.git (push)
upstream  https://github.com/codescope/notes-app.git (fetch)
upstream  https://github.com/codescope/notes-app.git (push)
```

What it does: on GitHub you clicked **Fork** on the original project, creating `ayesha-khan/notes-app`. You clone THAT (so `origin` is yours to push). Then `git remote add upstream` saves the original's address. `git remote -v` now shows the full map: two remotes, four lines.

**What you will see:** the fork's page carries a line under its name — "forked from codescope/notes-app" — GitHub itself keeps the family record. Inside your clone, that one `upstream` line is the difference between a dead copy and a living one.

### Example 2 — Sync with upstream BEFORE you start work

```
$ git fetch upstream
remote: Enumerating objects: 9, done.
From https://github.com/codescope/notes-app
 * [new branch]      main       -> upstream/main
$ git switch main
$ git merge upstream/main
Updating m3n4o5p..o5p6q7r
Fast-forward
 README.md | 4 ++++
 1 file changed, 4 insertions(+)
$ git push origin main
To https://github.com/ayesha-khan/notes-app.git
   m3n4o5p..o5p6q7r  main -> main
```

What it does: the original project moved on while your fork sat still. `git fetch upstream` downloads its latest state (without touching your files), the merge fast-forwards your local main to match, and the push updates your fork on GitHub. NOW you branch (G.3) from fresh code — your PR will diff against the present, not the past.

**What you will see:** if `upstream/main` had NOT moved, the merge says "Already up to date." and you saved yourself a surprise later. Maintainers may ask you to do exactly this ("please rebase/merge latest main") when

a PR goes stale — now you know the three commands they mean.

**Example 3 — Branch, fix, push, and open the PR on THEIR repo**

```
$ git switch -c fix-readme-typo
Switched to a new branch 'fix-readme-typo'
$ git commit -am "Docs: README mein 'recieve' ko 'receive' theek kiya"
[fix-readme-typo p6q7r8s] Docs: README mein 'recieve' ko 'receive' theek kiya
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git push -u origin fix-readme-typo
To https://github.com/ayesha-khan/notes-app.git
 * [new branch]      fix-readme-typo -> fix-readme-typo
```

What it does: the fix is small and single-purpose — a typo the maintainers will thank you for. It is committed on a descriptively-named branch and pushed to YOUR fork. On GitHub, you open the pull request FROM your fork's branch TO the original repo's main (the repo page of your fork shows a banner; or use the original's **Pull requests** → **New** and choose "compare across forks").

**What you will see:** the PR lives on the ORIGINAL repo, titled with your commit message, your name as author. A maintainer reviews (G.5's flow, from the other chair now), may request a tweak, and merges — and the contributors list of a real project gains your name. That is the finish line of this whole series: from `git init` to contributing to the world's code.

## Quick recap

- Fork (your GitHub copy) → clone the fork → `upstream` points home. Push to origin, refresh from upstream.
- Sync first, branch second, one small purpose per PR — and claim the issue before you start.
- The PR is opened on the original repo and reviewed like any team PR; merged work carries your name in a public history.

## Common mistakes

1. **Cloning the ORIGINAL repo and trying to push** — "Permission denied" at push time, after all the work is done. The fix: fork first, clone the fork; the original is reached only through the PR. (Your local commits are not lost — add the fork as origin and push there.)
2. **Branching from a stale fork** — your PR then contains the project's last month of changes mixed with your fix, and the review stalls. The fix: the Example 2 sync is step zero of every contribution, before the branch exists.
3. **The drive-by giant PR** — a first-time contributor reformats 40 files "to help". The fix: touch only what the issue needs; in someone else's house, small and polite wins — formatting sweeps and renames are separate, discussed proposals, never surprises.

## Practice

1. Find any small public repo (a friend's project counts) and walk the full route with a tiny docs fix: fork → clone your fork → add upstream → sync → branch → fix → push → PR with a two-line polite description.
2. On a repo you follow, read three `good first issue` labels and the CONTRIBUTING file. Write down, for ONE of them: the exact files you would touch and the branch name you would use — the full plan, before any code. That planning habit is what separates merged PRs from closed ones.

## Practice quiz

**1. You cannot push to a repo you do not own. What is the open-source way in?**

- Clone the original and push anyway
- Fork it: the fork is your own copy on GitHub that you fully own; all your pushes go to the fork, the original receives only your pull request
- Ask for the owner's password
- Email your code to the owner

**Answer:** Fork it: the fork is your own copy on GitHub that you fully own; all your pushes go to the fork, the original receives only your pull request

**2. After cloning your fork, what are origin and upstream for?**

- origin is the original, upstream is your fork
- origin is YOUR fork (push here), upstream is the ORIGINAL repo (pull from here to stay fresh)
- Both point to your fork
- Both point to the original

**Answer:** origin is YOUR fork (push here), upstream is the ORIGINAL repo (pull from here to stay fresh)

**3. Before touching code in a new open-source project, what should you read and claim?**

- Nothing, just start coding
- CONTRIBUTING.md and issues labelled good first issue, and claim the issue first with a comment so two strangers do not duplicate the same fix
- Only the README's first line
- The project's licence only

**Answer:** CONTRIBUTING.md and issues labelled good first issue, and claim the issue first with a comment so two strangers do not duplicate the same fix

---

**Watch the Code Scope channel:** https://www.youtube.com/@CodeScopePK — this lesson's own video joins the channel when it is recorded.

**Read online:** https://scopetreasure.com/notes/git-open-source-first-step · More free tools and notes at https://scopetreasure.com