ScopeTreasure Courses

GIT & GITHUB COURSES · LESSON G.3 · CODE SCOPE SERIES

Branching — Parallel Kaam

Work in parallel without breaking the main line: branch, switch, merge — and fixing a merge conflict line by line when two changes collide.

Introduction

Your app works. Now you want to try a risky new login page — but if the experiment breaks everything, your working app breaks with it. **Branches** are Git's answer: a branch is a parallel line of commits where you can experiment freely while `main` stays safe and shippable. When the experiment succeeds, you **merge** it back. And when two lines change the same lines of the same file, Git stops and asks you to decide — a **merge conflict**, which this lesson fixes by hand, calmly. *(Roman Urdu: Branch aik parallel line hai — tajurba wahan karein, main ko mehfooz rakhein; kaam ban jaye to merge kar dein.)*

Key points

- **A branch is just a movable label on a commit:** creating one costs almost nothing — no files are copied. `main` is simply the default label every repo starts with.
- **`git switch` moves you between lines:** switching changes the files in your folder to match that branch's last commit. Modern Git uses `switch` (older tutorials use `checkout` — same idea).
  (Roman Urdu: Branch badalna matlab folder ki files us line ke aakhri commit jaisi ho jana.)
- **Commits on a branch do not touch `main`:** you can commit ten experimental checkpoints, and `main`'s history does not gain a single one until you merge.
- **Merging brings one line into another:** stand on the receiving branch (`main`), then `git merge feature-login`. If `main` has not moved since you branched, Git does a **fast-forward** — it simply slides the label forward, no new commit needed.
- **A conflict is a question, not a disaster:** Git marks the exact lines both sides changed differently, and waits. You edit the file to the correct final text, remove the markers, `git add`, and commit — that commit completes the merge.

Code examples

Example 1 — Create a branch and work on it

```
$ git switch -c feature-login
Switched to a new branch 'feature-login'
$ echo "<form>Login form</form>" > login.html
$ git add login.html
$ git commit -m "Login page ka form jora"
[feature-login b2c3d4e] Login page ka form jora
 1 file changed, 1 insertion(+)
 create mode 100644 login.html
$ git log --oneline --all
b2c3d4e (HEAD -> feature-login) Login page ka form jora
7c8d9e0 (main) Pehla commit: README jora
```

What it does: `git switch -c` creates the branch and moves you onto it in one step. The commit lands on `feature-login` only. `--all` draws every line at once, so you can see the fork: one shared commit, then the new branch stepping ahead.

What you will see: `git switch main` makes `login.html` **disappear** from your folder — it only exists on the branch. Switch back and it returns. Nothing was deleted; each branch simply shows its own version of the project.

Example 2 — A clean merge (fast-forward)

```
$ git switch main
Switched to branch 'main'
$ git merge feature-login
Updating 7c8d9e0..b2c3d4e
Fast-forward
 login.html | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 login.html
$ git branch -d feature-login
Deleted branch feature-login (was b2c3d4e).
```

What it does: standing on `main`, `git merge` pulls the branch's commits in. Because `main` had no new commits of its own, Git fast-forwards: the `main` label slides to the same commit — no merge commit, history stays a straight line. The finished branch is deleted with `-d` (safe delete: Git refuses if it is not merged).

What you will see: `login.html` is now part of `main`, and `git log --oneline` on main shows both commits in one straight story — as if the experiment had happened right here.

Example 3 — A merge conflict, fixed by hand

```
$ git merge feature-heading
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
$ cat index.html
<<<<<<< HEAD
<h1>Welcome to Notes App</h1>
=======
<h1>Notes App — Seekho Aur Likho</h1>
>>>>>>> feature-heading
$ git add index.html
$ git commit -m "Merge feature-heading: heading ka conflict hal kiya"
[main f5g6h7i] Merge feature-heading: heading ka conflict hal kiya
```

What it does: both branches changed the same line of `index.html`, so Git refuses to guess. It writes both versions into the file between markers: `<<<<<<< HEAD` (your current branch's version) to `=======`, then the incoming version to `>>>>>>> feature-heading`. You edit the file to the ONE correct final line — say `<h1>Notes App — Seekho Aur Likho</h1>` — deleting all markers, then `git add` + `git commit` (the last two commands above) complete the merge.

What you will see: during the conflict, `git status` lists the file under "Unmerged paths" and tells you the fix steps. After your add + commit, the merge is complete and the file contains only your chosen line — no markers anywhere (search the file for `<<<<<<<` to be sure before committing).

Quick recap

- A branch is a cheap label, not a copy: experiment there, keep `main` safe, merge when it works.
- `git switch -c name` creates + moves; merging from the receiving branch fast-forwards when that branch has not moved.
- Conflicts are marked questions: choose the final text, delete the markers, add, commit. Never commit a file that still contains `<<<<<<<`.

Common mistakes

1. **Committing on the wrong branch** — you meant the experiment to be on `feature`, but you were standing on `main`. The fix: glance at `git status` (its first line names the branch) before every commit — or run `git branch` and look for the `*`.
2. **Merging in the wrong direction** — standing on the feature branch and merging main into it, then wondering why main did not change. The fix: switch to the branch that should RECEIVE the work first, then merge the other one in.
3. **Panicking at a conflict and deleting the repo** — the most expensive reaction to the cheapest problem. The fix: a conflict is just a file with markers plus instructions in `git status`; and if you truly want out, `git`

`merge --abort` returns everything to the pre-merge state.

Practice

1. In your `my-subjects` repo, create a branch `add-urdu-notes`, add a file `urdu.txt` there and commit. Switch back to `main` — confirm `urdu.txt` vanishes — then merge the branch and confirm it returns. Delete the branch with `git branch -d`.
2. Manufacture a tiny conflict on purpose: on `main`, write line 1 of a file as "Version A"; branch, change it to "Version B", commit; back on main change it to "Version C", commit; merge. Fix the conflict by choosing a combined line, and complete the merge commit.

Practice quiz

1. What is a branch, really?

- A full copy of all your files
- Just a movable label on a commit; creating one costs almost nothing and copies no files
- A separate repository
- A backup on GitHub

Answer: Just a movable label on a commit; creating one costs almost nothing and copies no files

2. You stand on main and run git merge feature-login. If main has not moved since you branched, what does Git do?

- It always creates a new merge commit
- A fast-forward: it simply slides the main label forward, no new commit needed
- It deletes the feature branch's commits
- It refuses, you must be on the feature branch

Answer: A fast-forward: it simply slides the main label forward, no new commit needed

3. How do you finish resolving a merge conflict?

- Delete the repository and start over
- Edit the file to the correct final text, remove the markers, git add, and commit; that commit completes the merge
- Commit the file with the <<<<<<< markers still in it
- Run git merge --abort and give up forever

Answer: Edit the file to the correct final text, remove the markers, git add, and commit; that commit completes the merge

---

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-branching · More free tools and notes at https://scopetreasure.com