Working With Commits

Writing good commit messages, browsing history with git log/git show, git commit -am, and understanding that git commit --amend creates a NEW commit rather than editing the existing one in place. HEAD and HEAD~1/HEAD~2 references.

Beginner 25 min

Last updated: August 27, 2026
Read online: https://www.learnforgex.com/en/topics/working-with-commits

---

You now know how to create a commit. This lesson is about making your commits actually *useful* — to your future self, and to every teammate who will read `git log` trying to understand why the codebase looks the way it does. A repository's history is a form of documentation; a repository full of commits named "fix" and "update" documents nothing.

Writing Good Commit Messages

A good commit message answers one question: **why** was this change made? The *what* is already visible in `git diff` — the message shouldn't repeat it.

```
Bad:  fix bug
Bad:  update UserService.java
Good: Fix NullPointerException when user has no roles assigned
Good: Add retry logic to handle transient database timeouts
```

The convention most teams use is a short (~50 character) summary line, written in the imperative mood ("Add", "Fix", "Remove" — not "Added" or "Fixes"), optionally followed by a blank line and a longer explanation if the *why* needs more than one sentence.

One increasingly common convention worth knowing is **Conventional Commits** — prefixing the summary with a type:

```
feat: add password reset endpoint
fix: correct off-by-one error in pagination
docs: update README setup instructions
refactor: extract validation logic into a separate method
```

This isn't required by Git itself — it's a team/tooling convention (some projects use it to auto-generate changelogs) — but it's common enough in professional Java/Spring Boot teams that you should recognize it.

git log and Useful Log Options

`git log` has options that make it far more useful day-to-day than the default multi-line output:

```
git log --oneline              # one line per commit -- fast overview
git log --oneline -5           # only the last 5 commits
git log --author="Ada"         # only commits by a specific author
git log --since="2 days ago"   # only recent commits
git log -- UserService.java    # only commits that touched this file
```

`git log --oneline` in particular is something you'll run dozens of times a day once it becomes habit — it gives you a compact commit-hash + summary view that's easy to scan.

git show

While `git log` lists commits, `git show` shows you the **full content** of one specific commit — its message and its complete diff:

```
git show 4f2a1c9        # show a specific commit by (partial) hash
git show HEAD           # show the most recent commit
```

You only need enough of a hash's prefix for Git to uniquely identify it — usually 7 characters is plenty on a normal-sized project.

Creating Commits with git commit -am

For files Git is **already tracking**, `-am` combines staging and committing into one step:

```
git commit -am "Fix null check in UserService"
```

This is equivalent to `git add -u` (stage all *tracked, modified* files) followed by `git commit -m "..."`. The important word is "tracked" — `-am` will **not** stage brand-new, untracked files; you still need an explicit `git add` for those.

> `-am` stages every modified tracked file, not just the one you're thinking about. If you have unrelated changes sitting in your working tree, `-am` will bundle them into the same commit. Prefer explicit `git add <file>` when you want a focused commit.

Amending the Last Commit — git commit --amend

If you just committed and immediately noticed a mistake — a typo in the message, or a file you forgot to include — `--amend` lets you fix it without creating a separate "oops" commit:

```
git add ForgottenFile.java     # stage the file you forgot
git commit --amend             # opens editor to optionally edit the message
git commit --amend --no-edit   # keep the same message, just add the staged changes
```

```
#!/bin/sh
# Committing, noticing a forgotten file, and amending instead of creating a
# second "oops" commit.

git commit -m "Add UserService with findById method"
# Output:
# [main 8a1b2c3] Add UserService with findById method
#  1 file changed, 8 insertions(+)

# Oops -- forgot the matching test file.
git add UserServiceTest.java
git commit --amend --no-edit

# Output:
# [main a9f3e21] Add UserService with findById method
#  2 files changed, 24 insertions(+)

# Note the new commit hash (a9f3e21) -- it REPLACED 8a1b2c3, it did not edit
# it in place. `git log --oneline -1` now shows only the new commit.
git log --oneline -1
# Output:
# a9f3e21 Add UserService with findById method
```

Important concept: `--amend` does not modify the existing commit in place — Git commits are immutable. What actually happens is Git creates a **brand-new commit** that replaces the old one, and moves your branch to point at the new one. The old commit still technically exists for a while (recoverable via `git reflog`, covered later in this course), but it's no longer part of your branch's history.

This distinction matters because of what it implies: if you've already **pushed** that commit and someone else has pulled it, amending creates a divergence between your history and theirs — your branch now has a *different* commit at that position than the copy they have. This is exactly why the rule of thumb is: **only amend commits that are still local, that nobody else has pulled yet.** We'll cover the full danger of rewriting shared history when we get to `git push --force` later in this course.

Understanding HEAD

HEAD is Git's name for "whatever commit you currently have checked out" — it's a pointer, not a fixed thing. Most of the time HEAD points at the tip of your current branch, and moves forward automatically every time you commit.

```
git log -1 HEAD    # show the commit HEAD currently points to
```

You'll see `HEAD` referenced constantly in Git tooling and error messages — it always means "here, right now, in my checked-out history."

Commit References — HEAD~1, HEAD~2, etc.

You can refer to earlier commits *relative to* HEAD without knowing their hash:

```
git show HEAD~1     # the commit before the current one
git show HEAD~2     # two commits before the current one
git log HEAD~3..HEAD  # the 3 most recent commits
```

`HEAD~1` is often written `HEAD^` (they mean the same thing for a normal, single-parent commit chain — `^` and `~N` only start to differ once merge commits with multiple parents are involved, which we'll cover in the Merge lesson). You'll use these constantly in the next lesson, Undoing Changes, since commands like `git reset` almost always take a target expressed this way.

Common Mistakes

- **Amending a commit that's already been pushed and pulled by someone else.** This rewrites shared history — avoid it unless you're certain nobody else has that commit.
- **Using `-am` and accidentally committing unrelated changes** that happened to be sitting in the working tree.
- **Writing commit messages that describe *what* changed** ("changed 3 lines in UserService") instead of *why* — the diff already shows what changed.

Best Practices

- Keep commits small and focused — one logical change per commit, not a mix of unrelated fixes.
- Write the summary line in the imperative mood, under ~50 characters.
- Use `git commit --amend` freely for commits you haven't pushed yet — it keeps history clean instead of accumulating "fix typo" follow-up commits.

Summary, Cheat Sheet, and Glossary

Summary

- A commit message should explain *why*, not just *what* — the diff already shows *what*.
- `git log --oneline` and `git show` are your everyday tools for browsing history.
- `-am` stages and commits already-tracked files in one step, but never stages new, untracked files.

- `--amend` creates a *new* commit that replaces the last one — it never edits history in place, and should only be used on commits nobody else has pulled.
- `HEAD` always points at your currently checked-out commit; `HEAD~1` / `HEAD~2` reference earlier commits relative to it.

Cheat Sheet

```
git log --oneline              # compact history
git show <hash>                # full content of one commit
git commit -am "message"       # stage tracked changes + commit
git commit --amend             # replace the last commit
git commit --amend --no-edit   # replace it, keep the same message
git show HEAD~1                # the commit before the current one
```

Glossary

- **HEAD** — a pointer to the commit you currently have checked out.
- **Amend** — replacing the last commit with a new one that includes additional changes and /or a new message.
- **Conventional Commits** — a common convention of prefixing commit summaries with a type (`feat:` , `fix:` , etc.).