Undoing Changes

Undoing working tree changes with git restore, the difference between git reset's soft /mixed/hard modes, and safely undoing shared commits with git revert. The relationship between HEAD, staging area, and working tree, and the data-loss risk of git reset --hard.

Intermediate 30 min

Last updated: August 27, 2026
Read online: https://www.learnforgex.com/en/topics/undoing-changes

---

Mistakes are normal — you'll stage the wrong file, write a commit message you regret, or commit before you meant to. What matters is knowing exactly which "undo" command fixes which situation, because Git has several, and picking the wrong one can genuinely lose work. This lesson walks through them from the safest (working tree only) to the most destructive (`git reset --hard`), always in terms of the three areas you already know: working tree, staging area, and commit history.

Undoing Working Tree Changes — git restore

If you've edited a file but haven't staged it yet, and you want to throw the edit away and go back to the last committed version:

```
git restore UserService.java
```

This discards uncommitted changes in the **working tree only** — it never touches the staging area or commit history. It's the safest undo command in this lesson, but it's still destructive to your *uncommitted* edits: once discarded, that specific edit is gone (it was never committed, so there's no history to recover it from).

Unstaging Changes — git restore --staged

If you staged a file with `git add` but changed your mind about including it in the next commit — without wanting to discard the edit itself:

```
git restore --staged UserService.java
```

This moves the file back from the staging area to the working tree, unstaged. Your edits are completely untouched — you've only undone the `git add`, nothing else. This is the correct command for "I staged the wrong file" or "I'm not ready to commit this yet."

Introduction to git reset

`git reset` is more powerful (and more dangerous) than `restore` — it moves where your branch's HEAD points, and can optionally also change the staging area and working tree to match. It takes a commit reference (often `HEAD~1`, `HEAD~2`, or a specific hash) and three modes that control exactly how much it touches:

```
git reset --soft   →  moves HEAD only
git reset --mixed  →  moves HEAD + resets staging area   (this is the default)
git reset --hard   →  moves HEAD + resets staging area + resets working tree
```

Each mode is strictly more destructive than the one before it. Let's go through them one at a time.

git reset --soft

This is the answer to a very common real situation: **"I committed too early. How do I undo the commit while keeping my changes staged?"**

```
#!/bin/sh
# "I committed too early. How do I undo the commit while keeping my changes
# staged?" -- git reset --soft HEAD~1.

git commit -m "Add password validation"
# Output:
# [main 3c9d1a2] Add password validation
#  1 file changed, 12 insertions(+)

# Realized the commit was premature -- want to add more before committing again.
git reset --soft HEAD~1

git status
# Output:
# On branch main
# Changes to be committed:
#   (use "git restore --staged <file>..." to unstage)
#         modified:   PasswordValidator.java

# The commit is gone from history, but nothing was lost -- the exact same
# changes are sitting in the staging area, ready to be committed again
# (optionally combined with more work, or a better message).
```

`git reset --soft HEAD~1` moves your branch back one commit, but leaves the staging area and working tree completely untouched — everything that was in that commit is now sitting in the staging area, ready to be re-committed (perhaps combined with more changes, or with a better message). Nothing is lost; you've only "uncommitted" it.

git reset --mixed

This is `git reset`'s **default mode** (running `git reset HEAD~1` with no flag is the same as `git reset --mixed HEAD~1`). It moves HEAD back *and* resets the staging area to match — but leaves your working tree files untouched.

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

The practical effect: the commit is undone, and its changes are now sitting in your working tree as **unstaged** edits — as if you'd made the changes but never run `git add`. This is the right choice when you want to undo a commit *and* re-review/re-stage the changes from scratch (maybe split them into several smaller commits instead of one).

git reset --hard

This is the most destructive of the three modes: it moves HEAD back, resets the staging area, **and** overwrites your working tree files to match the target commit.

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

> `git reset --hard` **discards uncommitted work permanently** with no confirmation prompt. Anything in your working tree or staging area that isn't part of a commit is gone the instant this command runs. Always run `git status` first, and consider `git stash` (covered later in this course) if there's anything you might want to keep.

Resetting to Previous Commits

All three modes work with any commit reference, not just `HEAD~1` — you can jump back several commits at once:

```
git reset --soft HEAD~3    # undo the last 3 commits, keep everything staged
git reset --hard a1b2c3d   # discard everything back to a specific commit hash
```

The further back you reset, the more commits get "undone" in the same way — this is why it's worth being deliberate about which mode you use when resetting multiple commits at once.

git revert

`reset` rewrites history by moving HEAD backward — which is exactly what makes it dangerous on commits that have already been pushed and shared. `git revert` solves the

same problem ("undo what this commit did") a completely different way: it creates a **brand-new commit** whose changes are the exact opposite of the target commit, leaving all existing history untouched.

```
git revert HEAD           # create a new commit that undoes the most recent one
git revert a1b2c3d        # undo a specific older commit
```

Because `revert` only ever adds a new commit — it never removes or rewrites existing ones — it's completely safe to use on commits that have already been pushed and pulled by other people.

Reset vs Revert

- **How it undoes** — `git reset` moves HEAD backward, rewriting history; `git revert` adds a new commit that cancels out the old one.
- **Safe on pushed/shared commits?** — `git reset`: no, it rewrites shared history. `git revert`: yes, it never rewrites existing history.
- **Effect on commit count** — `git reset` makes commits disappear from the branch; `git revert` grows the count (adds an "undo" commit).
- **Typical use** — `git reset` for local, not-yet-pushed commits; `git revert` for any commit, especially already-shared ones.

Choosing the Right Undo Command

A quick decision guide, from least to most drastic:

- Discard an uncommitted edit → `git restore <file>`
- Unstage a file, keep the edit → `git restore --staged <file>`
- Undo a *local, unpushed* commit, keep changes staged → `git reset --soft HEAD~1`
- Undo a *local, unpushed* commit, re-review the changes → `git reset --mixed HEAD~1` (or just `git reset HEAD~1`)
- Discard a local commit *and* its changes completely → `git reset --hard HEAD~1`
- Undo a commit that's already **pushed and shared** → `git revert <hash>`

Common Mistakes

- **Reaching for `git reset --hard` out of habit** when `--soft` or `--mixed` would have kept your work. Once `--hard` runs, uncommitted changes are gone.
- **Using `git reset` on a commit someone else has already pulled**, silently rewriting shared history — use `git revert` instead in that situation.
- **Confusing `restore` and `reset`** — `restore` only ever touches the working tree (and staging area with `--staged`), while `reset` also moves HEAD.

Best Practices

- Run `git status` before any reset, especially `--hard`, so you know exactly what you're about to lose.
- Default to `git revert` for anything already pushed — it's the safe choice for shared history.
- Reach for `git reset --soft` specifically when you want to "un-commit but keep everything" — it's the least destructive way to fix an early commit.

Summary, Cheat Sheet, and Glossary

Summary

- `git restore` undoes working tree changes (and, with `--staged`, unstages files) — it never moves HEAD.
- `git reset --soft/--mixed/--hard` moves HEAD backward with increasing levels of destructiveness — soft keeps everything staged, mixed unstages it, hard discards it entirely.
- `git revert` undoes a commit's effect by adding a new commit, never rewriting history — the only safe undo option for already-shared commits.

Cheat Sheet

```
git restore <file>              # discard uncommitted working tree changes
git restore --staged <file>     # unstage a file, keep the edit
git reset --soft HEAD~1         # undo last commit, keep changes staged
git reset --mixed HEAD~1        # undo last commit, unstage the changes
git reset --hard HEAD~1         # undo last commit, discard everything (dangerous!)
git revert HEAD                 # safely undo the last commit with a new commit
```

Glossary

- **Reset** — moves HEAD (and optionally the staging area/working tree) backward, rewriting history.
- **Revert** — undoes a commit's changes by adding a new, opposite commit; never rewrites history.
- **Soft / Mixed / Hard** — the three `git reset` modes, in order of increasing destructiveness.