RABBIT HOLES

Accelerate · Manipal University Jaipur

4

PHASE

Undo

Every way to take it back, and the one way you cannot.

Part of **git started**, Accelerate's third workshop.
github.com/accelerate-muj/git-started · MIT licensed.

Contents

1 The premise 3

2 The three trees 3

3 Undoing work you have not committed 3

3.1 Throw away changes to a file 3

3.2 Unstage a file 4

3.3 Delete files git has never seen 4

4 Undoing the commit you just made 4

4.1 You want to change the message 5

4.2 You forgot to include a file 5

5 git reset, and its three modes 5

5.1 Undo the last commit but keep the work 6

5.2 Undo the last commit and unstage 6

5.3 Destroy the last commit entirely 6

6 Undoing something that is already public 7

7 The reflog 7

7.1 Recovering a deleted branch 8

7.2 Recovering from a rebase that went wrong 8

7.3 When the reflog is not enough 8

8 Stash 8

9 Choosing the right undo 10

10 What genuinely cannot be recovered 10

1 The premise

Most people are frightened of git for one reason. They believe that if they type the wrong thing, work disappears.

This is almost never true, and the gap between "almost never" and what people believe is where all the fear lives. Git is built to hoard. It keeps things you deleted, commits you abandoned, branches you removed, and the position of your `HEAD` pointer going back ninety days. Undoing something in git is usually not recovery at all, it is just moving a pointer back to something that never actually went anywhere.

This phase covers every undo git has. By the end you should be willing to try things, which matters more than any single command here.

> **WHY THIS EXISTS**
>
> Confidence is a real engineering skill. A developer who is afraid of their tools works slowly, avoids branches, commits huge unreviewable lumps, and never experiments. Learning the undos first is what buys you permission to be careless later.

2 The three trees

Every undo command in git is answering the same question: *which of the three trees am I changing?*

The working directory The files as they exist on disk right now. What your editor shows you.

The index, also called the staging area What will go into the next commit. `git add` copies from the working directory into here.

HEAD The commit you currently have checked out. The last recorded state.

When you run `git status` and it lists things in two groups, those groups are "working directory differs from index" and "index differs from HEAD". That is the whole output.

Hold on to this. Every command below is a different combination of which trees get touched, and if you know the three trees you can predict what any of them does without memorising anything.

3 Undoing work you have not committed

3.1 Throw away changes to a file

You edited a file, you made it worse, you want the version from the last commit.

```
git restore sonnet.md
```

The file on disk is replaced with the one in `HEAD`.

> **TRAP**
>
> This is the single most dangerous command in this document. The edits you made were never committed, so git never had a copy, so there is nothing to recover. This is one of the few genuine ways to lose work in git.
> Before running it on something you might want, either commit it or stash it. Committing on a scratch branch costs nothing and is always reversible.

3.2 Unstage a file

You ran `git add` on something you did not mean to include.

```
git restore --staged secrets.env
```

This copies the file from `HEAD` back into the index. Your edits on disk are untouched, the file is simply no longer queued for the next commit.

> **WHY THIS EXISTS**
>
> `git restore` was added in Git 2.23, in 2019, together with `git switch`. Before that both jobs were done by `git checkout`, which also changed branches, also restored files, and also created branches. One command, three unrelated jobs, and the flags that separated them were not memorable.
> Older tutorials will tell you `git checkout -- file` and `git reset HEAD file`. Both still work. `restore` and `switch` exist because the maintainers agreed the old interface was the problem, not you.

3.3 Delete files git has never seen

`git restore` only knows about tracked files. Build output, stray downloads, and half-finished experiments are untracked, so it ignores them.

```
git clean -n
git clean -fd
```

`-n` is a dry run and prints what would be deleted. `-f` actually does it, `-d` includes directories.

> **TRAP**
>
> Always run `git clean -n` first. Always. This command deletes files git has no record of, which means there is no version of them anywhere. It is the other genuine way to lose work.

4 Undoing the commit you just made

4.1 You want to change the message

```
git commit --amend -m "A message that is actually useful"
```

4.2 You forgot to include a file

```
git add the-file-you-forgot.md
git commit --amend --no-edit
```

`--no-edit` keeps the existing message.

Here is the thing to understand about `--amend`: it does not edit anything. Commits in git are immutable. What actually happens is that git builds a brand new commit from the current index, gives it the same parent as the old one, and moves your branch to point at the new commit. The old commit is still sitting in the object database, unreferenced, and the reflog still knows where it is.

> **TRAP**
>
> Amending rewrites history. If you have already pushed the commit, your local branch and the remote branch now disagree, and an ordinary push will be rejected. You will need `git push --force-with-lease`, and everyone who already pulled the old commit will have a bad afternoon.
> The rule: amend freely before you push, think hard afterwards.

5 git reset, and its three modes

This is the command people find genuinely confusing, and the confusion is entirely because it has three modes that do three different amounts of work. The three trees make it simple.

`git reset` moves your branch pointer to a different commit. The mode decides what happens to the index and the working directory.

Mode	What it changes
`--soft`	Moves `HEAD` only. Index and working directory untouched. Your changes are still staged, ready to be recommitted.
`--mixed`	Moves `HEAD` and resets the index. Your changes survive on disk but are unstaged. *This is the default.*
`--hard`	Moves `HEAD`, resets the index, and overwrites the working directory. Your changes are gone from all three trees.

Read that table as a ladder. Each mode does everything the one above it does, plus one more tree.

5.1 Undo the last commit but keep the work

The one you will use most often. You committed too early, or committed two unrelated things together.

```
git reset --soft HEAD~1
```

The commit is gone, the changes are staged exactly as they were. Fix the grouping, commit again properly.

`HEAD~1` means "one commit before HEAD". `HEAD~3` means three before. You will see `HEAD^` used the same way for the single step case.

5.2 Undo the last commit and unstage

```
git reset HEAD~1
```

Same as above but the changes come back unstaged. Useful when you want to re-add selectively with `git add -p`.

5.3 Destroy the last commit entirely

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

> **TRAP**
>
> `--hard` is the flag with a reputation, and it half deserves it. The *commit* is recoverable through the reflog, covered in a moment. But any uncommitted changes sitting in your working directory at the time are overwritten, and those were never in git, so those really are gone.
> Committed work: recoverable. Uncommitted work: not. That distinction is the entire danger and it is worth reading twice.

> **TRY IT**
>
> Do this in a scratch repo right now, because reading about it is not the same as watching it.
>
> ```
> mkdir reset-demo && cd reset-demo && git init
> echo one > f.txt && git add . && git commit -m "one"
> echo two >> f.txt && git add . && git commit -m "two"
>
> git reset --soft HEAD~1 && git status
> git reset --hard HEAD && cat f.txt
> git reflog
> ```
>
> Watch `git status` after the soft reset, and notice the change is still staged. Then look at the reflog and find the commit you just destroyed.

6 Undoing something that is already public

Everything above rewrites history, which is fine on your own machine and rude on a shared branch. When a bad commit is already on `main` and other people have pulled it, you do not remove it. You add a new commit that reverses it.

```
git revert a1b2c3d
```

This creates a fresh commit whose content is the exact inverse of `a1b2c3d`. History gains a commit rather than losing one. Nobody else's clone breaks, because nothing they already have has changed.

> **WHY THIS EXISTS**
>
> Rewriting shared history is antisocial for a mechanical reason, not a stylistic one. Your teammate's branch is built on commits that, after your rewrite, no longer exist. Their next pull produces a mess that they have to untangle and that they did not cause.
> The standard formulation: **rewrite your own unpushed history freely, never rewrite history other people are building on.** Reset for private work, revert for public work.

To revert a merge commit you have to tell git which side to keep, because a merge has two parents:

```
git revert -m 1 <merge-sha>
```

`-m 1` means "keep the first parent", which is the branch you merged into, normally `main`.

7 The reflog

This is the safety net under every other command in this document, and it is the single most valuable thing in this phase.

Git records every position `HEAD` has occupied. Every commit, checkout, merge, rebase, reset, and amend appends a line. Not just commits on branches, but every intermediate state, including ones you have since abandoned.

```
git reflog
```

```
a1b2c3d HEAD@{0}: reset: moving to HEAD~1
9f8e7d6 HEAD@{1}: commit: two
5c4b3a2 HEAD@{2}: commit: one
```

`HEAD@{1}` is where `HEAD` was one move ago. To go back:

```
git reset --hard HEAD@{1}
```

Or, better, put the recovered commit somewhere safe rather than moving your branch:

```
git switch -c rescued 9f8e7d6
```

7.1 Recovering a deleted branch

You deleted a branch you still needed.

```
git reflog
git switch -c the-branch-i-deleted <sha>
```

The branch name was only ever a label pointing at a commit. Deleting it removed the label. The commit was untouched, and the reflog remembers the number.

7.2 Recovering from a rebase that went wrong

A rebase leaves a trail. Find the commit from just before you started:

```
git reflog
git reset --hard HEAD@{5}
```

Git also stores the pre-rebase position in `ORIG_HEAD`, so this often works straight away:

```
git reset --hard ORIG_HEAD
```

7.3 When the reflog is not enough

If a commit is not in the reflog and not on any branch, it is still in the object database until garbage collection runs. Find orphans with:

```
git fsck --lost-found
```

This lists dangling commits and blobs. Inspect one with `git show <sha>`, and if it is what you wanted, `git switch -c recovered <sha>`.

> **DEEPER**
>
> Unreferenced objects survive for ninety days by default, controlled by `gc.reflogExpire` and `gc.pruneExpire`. Git runs `gc` automatically when loose objects pile up.
> Which means the honest version of "git never loses anything" is: git does not lose anything for about three months, as long as it was committed at least once. Everything in this phase depends on that word *committed*. Committing early and often is not a discipline for other people's benefit, it is what makes you recoverable.

8 Stash

Sometimes you do not want to undo work, you want to put it down for ten minutes.

```
git stash
git stash pop
```

`stash` takes your uncommitted changes, saves them, and gives you a clean working directory. `pop` puts them back and removes the saved copy.

Useful in practice:

Command	What it does
`git stash push -m "half a fix"`	Save with a label you will recognise
`git stash list`	Show everything you have stashed
`git stash show -p stash@{1}`	Look at one as a diff before applying it
`git stash apply`	Restore but keep the stash as well
`git stash pop`	Restore and delete the stash
`git stash -u`	Include untracked files
`git stash branch fix`	Make a new branch from a stash
`git stash drop stash@{0}`	Delete one
`git stash clear`	Delete all of them

TRAP

The stash is a stack and it is easy to treat as a filing cabinet. Six unlabelled stashes from three weeks ago are useless, because you will never work out which is which and you will never dare delete any of them.
If work is worth keeping for more than an hour, it is worth a branch and a commit. Branches have names.

NOTE

Stashes are reachable through `git fsck` even after `git stash drop`, because they are ordinary commit objects. A dropped stash is recoverable for the usual ninety days, though finding it takes effort.

9 Choosing the right undo

Situation	Command
Edited a file, want the committed version	`git restore <file>`
Staged something by mistake	`git restore --staged <file>`
Untracked junk to delete	`git clean -n` then `git clean -fd`
Bad commit message, not pushed	`git commit --amend`
Forgot a file in the last commit	`git add <f>` then `git commit --amend --no-edit`
Committed too early, want the work back	`git reset --soft HEAD~1`
Want the last commit gone completely	`git reset --hard HEAD~1`
Bad commit already pushed to main	`git revert <sha>`
Deleted a branch you needed	`git reflog` then `git switch -c <name> <sha>`
Rebase went wrong	`git reset --hard ORIG_HEAD`
Need to set work aside briefly	`git stash`
Cannot find a commit anywhere	`git fsck --lost-found`

10 What genuinely cannot be recovered

Short list, and worth knowing exactly:

1. **Changes never committed or stashed**, destroyed by `git restore`, `git checkout` or `git reset --hard`. Git never held a copy.
2. **Untracked files** removed by `git clean`. Same reason.
3. **Anything already garbage collected**, roughly ninety days after it became unreachable.

4. **A repository you deleted from disk**, unless it was pushed somewhere.

Every item on that list is a variation of the same sentence: git can only give you back what it was given. The habit that follows is commit early, commit often, push regularly. A commit costs nothing and buys you three months of insurance.

> **CHECK YOURSELF**
>
> 1. Name the three trees, and say which ones `--soft`, `--mixed` and `--hard` each touch.
> 2. You committed too early and want the changes back, staged. Which command?
> 3. Why is `git revert` correct for a pushed commit when `git reset` is not?
> 4. You deleted a branch an hour ago. Walk through recovering it.
> 5. Which two commands in this phase can genuinely destroy work, and what do they have in common?

Next: Phase 5, *Rewriting history*, where you stop undoing commits and start reshaping them: interactive rebase, squashing, splitting a commit in half, and moving a commit from one branch to another.