**Git Undoing Things:**

At any stage, you may want to undo something. Here, we’ll review a few basic tools for undoing changes that you’ve made. ***Be careful, because you can’t always undo some of these undos***. This is one of the few areas in Git where you may lose some work if you do it wrong.

***One*** *of the* ***common undos takes place*** *when you* ***commit too early*** *and* ***possibly forget to add some files***, or *you* ***mess up your commit message***. If you want to **redo** that **commit**, make the additional changes you forgot, stage them, and commit again using the --amend option:

* **git commit --amend**

This command ***takes your staging area*** and uses it for the commit. If you’ve made no changes since your last commit (for instance, you run this command immediately after your previous commit), then your snapshot will look exactly the same, and all you’ll change is your commit message.

The same commit-message editor fires up, but it already contains the message of your previous commit. You can edit the message the same as always, but it overwrites your previous commit.

As an example, if you commit and then realize you forgot to stage the changes in a file you wanted to add to this commit, you can do something like this:

* git commit -m 'initial commit'
* git add forgotten\_file
* git commit --amend

You end up with a single commit — the second commit replaces the results of the first.

|  |  |
| --- | --- |
| Note | It’s important to understand that when you’re amending your last commit, you’re not so much fixing it as replacing it entirely with a new, improved commit that pushes the old commit out of the way and puts the new commit in its place. Effectively, it’s as if the previous commit never happened, and it won’t show up in your repository history.  The obvious value to amending commits is to make minor improvements to your last commit, without cluttering your repository history with commit messages of the form, “Oops, forgot to add a file” or “Darn, fixing a typo in last commit”. |

**Unstaging a Staged File**

To work with your ***staging area*** and ***working directory changes***. The nice part is that the command you use to determine the state of those two areas also reminds you how to undo changes to them. For example, let’s say you’ve changed two files and want to commit them as two separate changes, but you accidentally type git add \* and stage them both. How can you unstage one of the two? The git status command reminds you:

* git add \*
* git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README

modified: CONTRIBUTING.md

Right below the “Changes to be committed” text, it says use git reset HEAD <file>... to unstage. So, let’s use that advice to unstage the CONTRIBUTING.md file:

* **git reset HEAD CONTRIBUTING.md**

Unstaged changes after reset:

M CONTRIBUTING.md

* git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: CONTRIBUTING.md

The command is a bit strange, but it works. The CONTRIBUTING.md file is modified but once again unstaged.

|  |  |
| --- | --- |
| Note | It’s true that git reset can be a dangerous command, especially if you provide the --hard flag. However, in the scenario described above, the file in your working directory is not touched, so it’s relatively safe. |

For now this magic invocation is all you need to know about the git reset command. We’ll go into much more detail about what reset does and how to master it to do really interesting things in [Reset Demystified](https://git-scm.com/book/en/v2/ch00/_git_reset).

**Unmodifying a Modified File**

What if you realize that you don’t want to keep your changes to the CONTRIBUTING.md file? How can you easily unmodify it — revert it back to what it looked like when you last committed (or initially cloned, or however you got it into your working directory)? Luckily, git status tells you how to do that, too. In the last example output, the unstaged area looks like this:

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: CONTRIBUTING.md

It tells you pretty explicitly how to discard the changes you’ve made. Let’s do what it says:

* **git checkout -- CONTRIBUTING.md**
* **git status**

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: README.md -> README

You can see that the changes have been reverted.

|  |  |
| --- | --- |
| Important | It’s important to understand that git checkout -- <file> is a dangerous command. Any local changes you made to that file are gone — Git just replaced that file with the most recently-committed version. Don’t ever use this command unless you absolutely know that you don’t want those unsaved local changes. |

If you would like to keep the changes you’ve made to that file but still need to get it out of the way for now, we’ll go over stashing and branching in [Git Branching](https://git-scm.com/book/en/v2/ch00/ch03-git-branching); these are generally better ways to go.

Remember, anything that is committed in Git can almost always be recovered. Even commits that were on branches that were deleted or commits that were overwritten with an --amend commit can be recovered (see [Data Recovery](https://git-scm.com/book/en/v2/ch00/_data_recovery) for data recovery). However, anything you lose that was never committed is likely never to be seen again.