Undoing Changes with Git

[(https://github.com/learn-co-curriculum/git-github-undoing-changes-with)](https://github.com/learn-co-curriculum/git-github-undoing-changes-with) [(https://github.com/learn-co-curriculum/git-github-undoing-changes-with/issues/new)](https://github.com/learn-co-curriculum/git-github-undoing-changes-with/issues/new)

Learning Goals

- Use Git commands to undo previous changes.

Introduction

Now that we know how to use the `git log` and `git diff` commands to explore our Git history, the next step is to learn how we can use these skills to undo or amend changes we've made. As with most things in Git, there are multiple commands and options we can use to go back in time and fix things. As always, the purpose here is not to memorize all (or even any!) of these commands, but to gain some understanding of what they do and to get some practice so you're comfortable using them when you need them.

Undoing Changes

One of the big advantages of using Git is that it gives you a great deal of control when you need to recover from making a mistake. It allows you to travel back in your Git history and undo changes, basically "resetting" your code to an earlier state. Here, again, you will see how beneficial it is to follow best practices for committing your code and using descriptive commit messages.

We will go over how to undo unstaged changes, staged changes, and changes that have been committed. Before we do that, however, we need to talk a bit about the `git checkout` command.

Git Checkout

The `git checkout` command is quite complex — if you take a look at the **Git documentation** [(https://git-scm.com/docs/git-checkout)](https://git-scm.com/docs/git-checkout), you can see just how many options and uses it has. However, `git checkout` is most often used for one of two purposes: checking out branches (which we will talk about in the next set of lessons) and checking out specific commits, for example to restore our code to an earlier state.

In both of these cases, we're simply pointing the HEAD to a different version of our code. However, because the two cases have very different purposes, the use of the same command for both can be confusing. As a result, in version 2.23, Git created two new commands that are a little more meaningfully named: `git switch` (used to switch branches) and `git restore` (https://git-scm.com/docs/git-restore).

Where applicable, we will include both versions — the original `checkout` command and the newer command — since you'll see both used. Ultimately, you are free to choose whichever option makes the most sense to you.

Unstaged Changes

When we want to undo changes that have not yet been staged or committed, it means we want to go back to the version we had in the most recent commit, i.e., to HEAD. We can do that using either `git checkout` or `git restore`.

With `git checkout` it would look like this:

```
$ git checkout HEAD <filename(s)>
```

For example, if we have two files we've made changes to and we want to discard the unstaged changes in both of them:

```
$ git checkout HEAD file1.txt file2.txt
```

This resets the state of the two files back to what they looked like in the last commit, and discards all the changes that have been made since then.

We can accomplish the same thing using `git restore`:

```
$ git restore <filename(s)>
```

Notice that the syntax for `restore` is a little simpler. We don't need to specify that we want to revert the code to HEAD — it does that automatically.

Note: In this case, because the changes we've made haven't been added or committed, we will lose them when we run either of the commands above. Only run one of these commands if you're sure you don't need any of the changes you've made since the last commit.

Let's go ahead and add some more birds to the backyard bird list we created in the previous lesson:

```
...
- American Flamingo
- Snail Kite
- Roseate Spoonbill
- Crested Caracara
```

We're getting ready to stage and commit our changes when we realize we've accidentally added birds from a recent vacation instead of birds seen in our yard. In this example, we could easily recover by just deleting or undoing the changes, but in cases where that isn't possible or would require a lot more work, we can use one of the commands above.

Go ahead and undo the changes using the `git restore` command. If you have the file open in VS Code, you'll see that those last four birds are no longer there.

Staged Changes

If you want to discard changes that you've staged but not yet committed, you can use `restore` to do that as well:

```
$ git restore --staged <filename(s)>
```

In this case, the changes you've made will not be lost. They will still be present in your working directory, but they will no longer be staged. This can be helpful if you decide you want to make additional changes before committing, or if you decide to split out changes in different files to separate commits.

Let's add some birds for day 3 to our list:

```
- American Crow
- Blue Jay
- Downy Woodpecker
- Yellow Warbler
```

Go ahead and stage the changes using `git add` , then run `git status` to verify that the file has been staged.

At this point, we realize we forgot a couple of birds. We could just make the additional changes then re-add the file (as we did in the last lesson), but if we want to make sure we don't accidentally commit the changes before we're ready, we can use `git restore --staged` to unstage the file:

```
$ git restore --staged bird_list.md
```

If you check the file you'll see that the four birds we added are still there, and if we run `git status` we'll see that our changes are no longer staged. We can add the birds we forgot, then re-stage and commit the file when we're done.

Top Tip: You don't need to memorize the restore commands! Any time one of them is applicable, Git shows syntax for using it when you run `git status` . To see this, go ahead and re-stage the file then run `git status` . You should see something like this:

```
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
```

If you have *unstaged* changes, it will look like this:

```
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   bird_list.md

no changes added to commit (use "git add" and/or "git commit -a")
```

Committed Changes

Finally, there are several commands that can be used for undoing changes that have been committed: [`git reset`](https://git-scm.com/docs/git-reset) (https://git-scm.com/docs/git-reset) , `git reset --hard` and [`git revert`](https://git-scm.com/docs/git-revert) (https://git-scm.com/docs/git-revert) .

The `git reset` command allows us to undo one or more commits **without discarding the changes we made.** In other words, the commit(s) are erased from our Git history, but the state of the code in our working directory is unchanged.

Earlier, we realized that we needed to add more birds *before* we committed, but that won't always be the case. Let's go ahead and re-stage our `bird_list.md` file with the four birds we added above (if you haven't already), then commit the changes:

```
$ git add bird_list.md
$ git commit -m "add third day's birds to list"
[main 06cc641] add third day's birds to list
 1 file changed, 4 insertions(+)
```

Now, to un-commit those changes, we run `git reset <SHA>` . The SHA we use in this command is the one we want to reset our code **to**. To find the one we need, we can use `git log` :

```
$ git log --oneline -2
584218a (HEAD -> main) add third day's birds to list
f9658e3 add second day's birds to list
```

We want to remove the last commit and reset our history to what it was after we added the second day birds, so we'll run:

```
$ git reset f9658e3
Unstaged changes after reset:
M         bird_list.md
```

If you look at `bird_list.md` , you will see that the changes are still present in the file, but those changes are now back to being unstaged. You can also see that if you run `git diff` . If you run `git log` , you will see that the "add third day's birds to list" commit is no longer part of our commit history.

The `reset` command can be helpful if you've committed some work on the `main` branch but decide you should be using a separate branch instead. Once you've run `git reset` to undo the commit (or commits), you can switch to a different branch and add and commit the changes there.

If you want to undo the commit **and** remove the changes, you would use the `--hard` option:

```
$ git reset --hard <SHA>
```

Note that using the `git reset` command can cause problems if you're working with collaborators because it **changes your Git history**. If the earlier commit(s) have been pushed up to GitHub and pulled down by anyone else, using `git reset` means that the Git history on your version of the code is different from the history other people have. This can lead to problems when they try to push their code up. Under these circumstances, there is a safer option: `git revert` (https://git-scm.com/docs/git-revert) .

Like `git reset` , `git revert` reverts the code to its state from an earlier commit, but **without** removing later commits. It simply creates a **new** commit in which the changes have been undone. Note that, while with `git reset` , you need to specify the SHA you want to reset **to**, with `git revert` you specify the SHA for the commit you want to undo:

```
$ git revert <SHA>
```

If you're reverting the most recent commit, you can use `HEAD` in place of the SHA.

When you use the `revert` command, Git creates a default commit title (`"Revert <commit message>"`) and message (`"This reverts commit <SHA>"`). Before the commit completes, it opens a file in the Git command line text editor that contains the default message so you can edit it if you choose. Edit it however you like, then save and close the file to complete the commit or, to use the default as is, simply close the file.

Note: The lesson "Maintaining a Clean, Well-documented Commit History" contains instructions for how to configure the text editor for Git. If you did not configure it at that time, we encourage you to go back and do it now. (See the "Mulitiline Commit Messages" section in that lesson for more details.)

Once you run `git revert` , the reverted changes will no longer be in your working directory, but, if you change your mind, you can still get them back through your Git history.

Exercise

1. Add and commit the list with the day 3 birds one more time.
2. Run `git log` to confirm that the commit has been made.
3. Run `git revert` to undo the commit. When the default commit message opens in your text editor, simply close the file to use the default message and complete the commit.

4. Run `git status` once more. You should see that the day 3 commit is still in your Git history, and that an additional commit has been added reverting that commit.
5. Use `git diff` to compare the file as it is now (`HEAD`) to the file from the day 2 commit. The command should not display anything.

Conclusion

The best way to avoid mistakes or problems with your code is by following good Git practices. But no matter how careful you are, there will be times when you make a mistake or change your mind. Git can be a powerful tool for recovering from these situations. It's important to have an understanding of the things Git allows you to do so you know how to access the information when you need it.