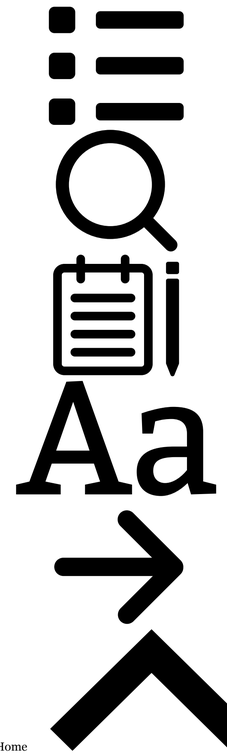

6

Gitignore After the Fact Written by Jawwad Ahmad & Chris Belanger

When you start a new software project, you might think that the prefab **.gitignore** you started with will cover every possible situation. But more often than not, you'll realize that you've committed files to the repository that you shouldn't have. While it seems that all you have to do to correct this is to reference that file in **.gitignore**, you'll find that this doesn't solve the problem as you thought it would.

In this chapter, you'll cover a few common scenarios where you need to go back and tell Git to ignore those types of mistakes. You're going to look at two scenarios to fix this locally: Forcing Git to assume a file is unchanged and removing a file from Git's internal index.

Getting started

To start, extract the repository contained in the starter **.zip** file inside the **starter** directory from this chapter's materials. Or, if you completed all of the challenges from the previous chapter, feel free to continue with that instead.

.gitignore across branches

Git's easy and cheap branching strategy is amazing, isn't it? But there are times when flipping between branches without a little forethought can get you into a mess.
Here's a common scenario to illustrate this.
Inside the **magicSquareJS** project, you should already be on the `main` branch. If you aren't, run `git checkout main`.
Pull up a listing of the top-level directory with `ls` and you'll see a file named **IGNORE_ME**. Print the contents of the file to the command line with **`cat IGNORE_ME`** and you'll see the following:

```
Please ignore this file. It's unimportant.
```

Now assume you have some work to do on another branch. Switch to the `yDoublyEven` branch with the following command:

```
git checkout yDoublyEven
```

Pull up a complete directory listing with the following command:

```
ls -la
```

You'll see that there's a **.gitignore** there, but there's no sight of the IGNORE_ME file. Looks like things are working properly so far.
Open up the **.gitignore** file in an editor and you'll see the following:

```
IGNORE_ME*
```

It looks like you're all set up to ignore that **IGNORE_ME** file. Therefore, if you create an **IGNORE_ME** file, Git should completely ignore it, right? Let's find out.
Create a file named **IGNORE_ME** in the current directory, and add the following text to that file:

```
Please don't look in here
```

Save your changes and exit.
You can check that Git is ignoring the file by executing `git status`:

```
On branch yDoublyEven
Your branch is up to date with 'origin/yDoublyEven'.

nothing to commit, working tree clean
```

So far so good. It looks like everything is working as planned.
Now switch back to `main` with the following command:

```
git checkout main
```

And at this point, Git shouldn't have anything to complain about, since it's ignoring that **IGNORE_ME** file. But open up that **IGNORE_ME** file and see what's inside:

```
Please ignore this file. It's unimportant.
```

Wait — shouldn't Git have ignored the change to that file and preserved the original `Please don't look in here` text you added on the other branch? Why did Git overwrite your changes, if it should have been ignoring any changes to this file?
Sounds like you should have a look at the **.gitignore** file in `main` to see what's going on. There *is* a **.gitignore** file in `main`, right?
Pull up a full directory listing with `ls -la` and you'll see that, in fact, there *is* no **.gitignore** on the `main` branch:

```
.
├── .git
├── .tools-version
├── IGNORE_ME
├── LICENSE
├── README.md
├── SECRETS
├── css
├── img
├── index.html
```

```
└── js
```

Oh. Well, that seems easy to fix. You'll just add a reference to **IGNORE_ME** to the **.gitignore** on `main` and everything should just sort itself out.
Create a **.gitignore** file in the current directory, and add the following to it:
```
IGNORE_ME*
```

Save your changes and exit. So Git should start ignoring any changes to **IGNORE_ME** now, right? It seems like you're safe to put your original change back in place.
Open up **IGNORE_ME** in an editor, and replace the contents of that file with the original content you wanted in there in the first place:
```
Please don't look in here
```

Save your changes and exit. Execute a quick `git status` to check that Git is actually ignoring that file, as you'd hoped:
```
git status
```

You'll see the following in your console, showing that Git is absolutely *not* ignoring that file:
```
On branch main
Your branch is ahead of 'origin/main' by 21 commits.
  (use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   IGNORE_ME

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore

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

Wait, what? You told Git to ignore that file, yet Git is obviously still tracking it. What's going on here? Doesn't putting something in **.gitignore**, gee, I don't know, tell Git to *ignore* it?
This is one of the more frustrating things about Git; however, once you build a mental model of what's happening, you'll see that Git's doing exactly what it's supposed to. And you'll also find a way to fix the situation you've gotten yourself into.

How Git tracking works

When you stage a change to your repository, you're adding the information about that file to Git's **index**, or **cache**. This is a binary structure on disk that tracks everything you've added to your repository.
When Git has to figure out what's changed between your working tree and the staged area, it simply compares the contents of the index to your working tree to determine what's changed. This is how Git "knows" what's unstaged and what's been modified.
But if you *first* add a file to the index, and *later* add a rule in your **.gitignore** file to ignore this file, this won't affect Git's comparison of the index to your working tree. The file exists in the index and it also exists in your working tree, so Git won't bother checking to see if it should ignore this file. Git only performs **.gitignore** filtering when a file is in your working tree, but *not yet* in your index.
This is what's happening above: You added the **IGNORE_ME** file to your index in `main` *before* you got around to adding it to the **.gitignore**. So that's why Git continues to operate on **IGNORE_ME**, even though you've referenced it in the **.gitignore**.
In fact, there's a handy command you can use to see what Git is currently ignoring in your repository. You've already used it quite a lot in this book, believe it or not! It's simply `git status`, but with the `--ignored` flag added to the end.
Execute this now to see what Git is ignoring in your repository:
```
git status --ignored
```

My output looks like the following:
```
On branch main
Your branch is ahead of 'origin/main' by 21 commits.
  (use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   IGNORE_ME

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore

Ignored files:
  (use "git add -f <file>..." to include in what will be committed)
        .DS_Store
        js/.DS_Store

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

So Git is ignoring .DS_Store files, as per my global **.gitignore**, but it's not ignoring **IGNORE_ME**. Fortunately, there are a few ways to tell Git to start ignoring files that you've already added to your index.

Updating the index manually

If all you want is for Git to ignore this file, you can update the index yourself to tell Git to assume that this file will never, ever change again. That's a cheap and easy workaround.
Execute the following command to update the index and indicate that Git should assume that when it does a comparison of this file, the file hasn't changed:
```
git update-index --assume-unchanged IGNORE_ME
```

Git won't give you any feedback on what it's done with this command, but run `git status --ignored` again and you'll see the difference:
```
On branch main
Your branch is ahead of 'origin/main' by 21 commits.
  (use "git push" to publish your local commits)

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore

Ignored files:
  (use "git add -f <file>..." to include in what will be committed)
        .DS_Store
        js/.DS_Store

nothing added to commit but untracked files present (use "git add" to track)
```

Git isn't ignoring it, *technically*, but for all intents and purposes, this method has the same effect. Git won't ever consider this file changed for tracking purposes.
To prove this to yourself, modify **IGNORE_ME** and add some text to the end of it, like below:
```
Please don't look in here. I mean it.
```

Save your changes, exit out of the editor, and then run `git status --ignored` again. You'll see that Git continues to assume that that file is unchanged.
This is useful for situations where you've added placeholders or temporary files to the repository, but you don't want Git tracking the changes to those temporary files during development. Or maybe you just want Git to ignore that file for now, until you get around to fixing it in a refactoring sprint later.
The issue with this workaround is that it's only a *local* solution. If you are working on a distributed repository, everyone else would have to do the same thing in their own clone if they want to ignore that file. Telling Git to assume a file is unchanged only updates the index on your local system. This means these file changes won't make it into a commit — but it also means that anyone else cloning this repo will still run into the same issues you did.
In fact, you might prefer to remove this file from the index entirely, instead of just asking Git to turn a blind eye to it.

Removing files from the index

When you implicitly or explicitly ask Git to start tracking a file, Git dutifully places that file in your index and starts watching for changes. If you're quite certain that you don't want Git to track this file anymore, you can remove this file from the index yourself.
After you remove a file from the index, Git follows the natural progression of checking the working tree against the index for changes, then looking to the **.gitignore** to see if it should exclude anything from the changeset.
You've already run across a command to remove files from Git's index: `git rm`. By default, `git rm` will remove files from *both* the index and your working tree. But in this case, you don't want to remove the file in your working tree — you want to keep it.
To remove a file from the index but leave it in your working tree, you can use the `--cached` option to tell Git to remove this file from the index only.
Execute the following command to instruct Git to remove **IGNORE_ME** from the index. Git will, therefore, stop tracking it:
```
git rm --cached IGNORE_ME
```

Git responds with a simple confirmation:
```
rm 'IGNORE_ME'
```

To see that this has worked, run `git status --ignored` again:
```
On branch main
Your branch is ahead of 'origin/main' by 21 commits.
```

```
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    IGNORE_ME

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore

Ignored files:
  (use "git add -f <file>..." to include in what will be committed)
        .DS_Store
        IGNORE_ME
        js/.DS_Store
```

IGNORE_ME now shows that it's both `deleted` and `ignored`. How can that be?
If you think about Git's perspective for a moment, this makes sense: `git status` compares the staging area, or index, to `HEAD` to see what the next commit should be. Git sees that **IGNORE_ME** is no longer in the index. Whether this file exists on disk is irrelevant to Git at this moment. So it sees that the next commit would delete **IGNORE_ME** from the repository.
Besides that, including the `--ignored` option on `git status` builds a list of what Git now knows to ignore, based on any files in your working tree that match any filters in the **.gitignore**.
IGNORE_ME is not in your current index, so when Git runs its ignore filter, it sees that you have a file named **IGNORE_ME** on disk and that file isn't present in your current index.
However, you've put this filter in your **.gitignore**, so Git adds this file to its list of files to ignore. Hence, **IGNORE_ME** is both in deleted status (as far as the index is concerned) and ignored status (as far as your **.gitignore** is concerned).
Since this seems to have cleared up the situation, you can now create your next commit. But wait — aren't you forgetting something? Something that got you into this mess in the first place?
Ah right — **.gitignore** is still untracked. Stage that file now:

```
git add .gitignore
```

And commit this change before you forget again:

```
git commit -m "Added .gitignore and removed unnecessary file"
```

Just remember that if someone else clones the repo after you've pushed this commit, they'll also lose that file in their clone. As long as that's your intent, that's fine.
Now, remember that this doesn't remove *all* traces of your file — there's still a whole history of commits in your repository that have this file fully intact. If someone really wanted to, they could go back in history and find what's inside that file.
To see this, run `git log` on the file in question:

```
git log -- IGNORE_ME
```

The second entry in that log shows the following:

```
commit 7ba2a1012e69c83c4642c56ec630cf383cc9c62b (origin/main, origin/HEAD)
Author: Yasmin <yasmin@example.com>
Date:   Mon Jul 3 17:34:22 2017 +0700

    Adding the IGNORE_ME file
```

Well, that doesn't seem to be a *huge* deal. So what if people can see that you added a file you later removed from the repository?
In this case, it's not that important. But often, people commit massive zip or binary files to a repo, and don't realize it until people complain about how long it takes to clone a repo to their local system.
More critically, what if you'd accidentally committed a file with API keys, passwords or other secrets inside? Then you *absolutely do care* about making sure you've purged the repository of any history about this file. If someone were to get your API keys or other secrets, they potentially have unlimited, unfettered access to some of your systems. Whoops.

Rebasing isn't always the solution

Assume you don't want anyone to know about the existence of **IGNORE_ME**. You've already learned one way to rewrite the history of your repository: Rebasing. But will this solve your current issue?
To see why rebasing isn't a great way to solve this problem, you'll work through an interactive rebase on the current repository. This will show you the situations where `git rebase` might not be the best choice to rewrite history.
You know that Yasmin added **IGNORE_ME** back in commit hash `7ba2a1012e69c83c4642c56ec630cf383cc9c62b`, as you saw above. So all you have to do is drop that particular commit, rebase everything else on top of the ancestor commit, and everything would be *just fine*, right?
But first: Did that commit *only* add **IGNORE_ME**? Or did it add any other files? You need to know that before you commit. You can't always trust someone's commit message.
Have a look at the patch for this commit to see what it actually contains:

```
git log -p -1 7ba2a10
```

You should see the following:

```
commit 7ba2a1012e69c83c4642c56ec630cf383cc9c62b
Author: Yasmin <yasmin@example.com>
Date:   Mon Jul 3 17:34:22 2017 +0700

    Adding the IGNORE_ME file

diff --git a/IGNORE_ME b/IGNORE_ME
new file mode 100644
index 0000000..28c0f4f
--- /dev/null
+++ b/IGNORE_ME
@@ -0,0 +1 @@
+Please ignore this file. It's unimportant.
```

OK, it seems that commit only added that file, as it said in the commit. Theoretically, you should be able to drop that commit from the history of the repo and everything should be *just* fine.
Start an interactive rebase with the following:

```
git rebase -i 7ba2a10~
```

The tilde ~ at the end of the commit hash means "start the rebase operation at the commit just prior to this one."
Git presents you with the interactive script for this rebase:

```
pick 7ba2a10 Adding the IGNORE_ME file
pick 883eb6f Adding methods to allow editing of the magic square
pick e632550 Adding ID to <pre> tag
pick f28af7a Adding ability to validate the inline square
pick c2cf184 Wiring up the square editing and validation
.
.
.
pick baf24aa Added .gitignore and removed unnecessary file
```

All you need to do is drop that first commit, right? Using your git-fu skills, type **cw** to cut the `pick` command on that first line, and in its place, put `drop`. Your rebase script should look like the following:

```
drop 7ba2a10 Adding the IGNORE_ME file
pick 883eb6f Adding methods to allow editing of the magic square
pick e632550 Adding ID to <pre> tag
pick f28af7a Adding ability to validate the inline square
pick c2cf184 Wiring up the square editing and validation
.
.
.
pick baf24aa Added .gitignore and removed unnecessary file
```

Press **Escape** to exit out of insert mode, and type **:wq** followed by **Enter** to save your work and carry on with the interactive rebase.
...and, of course, nothing is ever as simple as it seems. You've run into a merge conflict already, on **index.html**:

```
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
error: could not apply f985ed1... Centre align everything
```

Oh, right. Because Git is actually replaying all of the other commits as part of the rebase, you'll encounter merge conflicts in files that aren't related to **IGNORE_ME**.
What you failed to take into consideration is the ancestor of `7ba2a10 Adding the IGNORE_ME file` — and what's happened in the repository since then.
Execute the following command to see the full gory details of the origins of this commit:

```
git log --oneline --graph --all
```

Scroll *way* down and you'll see commit `69670e7 Adding a new secret`:

```
.
.
```

```
.
| * | | e632550 Adding ID to <pre> tag
| * | | 883eb6f Adding methods to allow editing of the magic square
| |/ /
* | | 7ba2a10 Adding the IGNORE_ME file
* | | 32067b8 Adding the structure to the generator
|/ /
* | 69670e7 Adding a new secret
.
.
.
```

`69670e7` is the ancestor of `7ba2a10`. And a lot has happened in the repository since that point. So when Git rewinds the history of the repository, it has to go all the way back to that ancestor and replay *every* commit that's a descendant of that ancestor and rebase it on top of `69670e7` — even commits that you've already merged back to `main`. Ugh. This really isn't what you bargained for, is it? You *could* go through each of these commits and resolve them, but that's a tremendous amount of work, and quite a bit of risk, just to get rid of a single file.
Abort this rebase in progress with the following command:
`git rebase --abort`

This resets your staging and working environment back to where you were before.
Note: For the purists out there, your working and staging area never actually changed during the rebase. Rebasing happens in a temporary detached HEAD space, which you can think of as a "virtual" branch that isn't spliced onto your repo until the rebase is complete. Aborting a rebase simply throws away that temporary space and puts you back into your unchanged working and staging area.
This isn't a scalable solution — not in the least. There's a better way to do this, and it's known as `git filter-branch`.

Using filter-branch to rewrite history

Let's put the issue with IGNORE_ME aside for the moment; you'll come back to it at the end of the chapter. Right now, you'll work through an issue with a similar file, **SECRETS**, that plays out the dreaded scenario above where you've committed files or other information that you never wanted to be public.
Print out the contents of the **SECRETS** file with the following command:
`cat SECRETS`

You'll see the following:

```
DEPLOY_KEY=THIS_IS_REALLY_SECRET
RAYS_HOTTUB_NUMBER=012-555-6789
```

Can you *imagine* the chaos if those two pieces of information hit the streets? You'll need to clean up the repository to remove all traces of that file — and also make sure the repository has been rewritten to remove any indication that this file was *ever* there in the first place.
The `filter-branch` command in Git lets you programmatically rewrite your repository. It's similar to what you tried to do with the interactive rebase, but it's far more flexible and powerful than trying to tweak things manually during an interactive rebase.
Although there are lots of ways to run `filter-branch`, you'll take the most direct route to remove this file: Rewrite your repository's staging area, or index.
A quick review, first: Do you recall how to remove a file from the index? That's right — `git rm --cached` removes the file from your staging area, as opposed to your working area. Remember this; you'll need it in just a moment.
There's another option to `git rm` that you'll need to know: `--ignore-unmatch`.
To see why you need this option, execute the following command at the command line to try to remove a non-existent file from the index:
`git rm --cached -- NoFileHere`

Git will respond with a `fatal` error:

```
fatal: pathspec 'NoFileHere' did not match any files
```

Since this is a fatal error, Git stops in its tracks and returns with what's known as a non-zero exit status; in other words, it errors out.
To prove this even further, execute the following chained Bash command, which will print `success!` if the first command succeeds:
`git rm --cached -- NoFileHere && echo 'success!'`

Git again responds with the single fatal error and halts; `echo 'success!'` is never executed. It's clear that if `git rm` doesn't match on a filename, it's done and halts execution immediately.
To get around this, `--ignore-unmatch` will tell Git to report a zero exit status — that is, a successful completion — even if it doesn't find any files to operate on. To see this in action, execute the following chained Bash command:
`git rm --cached --ignore-unmatch -- NoFileHere && echo 'success!'`

You'll see `success!` printed to the console, showing that `git rm` exited successfully.
Now — to put this knowledge to work.
Execute the following command to run `git filter-branch` to remove the offending file:
`git filter-branch -f --index-filter 'git rm --cached --ignore-unmatch -- SECRETS' HEAD`

Note: You'll see a warning about git-filter-branch having gotchas, and the message from Git itself recommends using an external tool called git-filter-repo (available at https://github.com/newren/git-filter-repo). However for our simple case git-filter-branch is sufficient and the command will automatically proceed a few seconds after showing the warning.
Taking that long command one bit at a time:
You execute `git filter-branch` to tell Git to start rewriting the repository history.
The `-f` option means "force"; this tells Git to ignore any internally-cached backups from previous operations. If you routinely use `filter-branch`, you'll want to use the `-f` option to avoid Git reminding you every time you run `filter-branch` that you have an existing backup from a previous operation.
You next specify the `--index-filter` option to tell Git to rewrite the index, instead of rewriting your working tree directly (more on that later).
You then specify the filter, or command, you want to run on each matching commit as Git rewrites history. In this case, you're performing `git rm --cached` to look up files in the index. `--ignore-unmatched` prevents Git from bailing out of `filter-branch` if it doesn't match any files. Finally, you indicate you want to remove the **SECRETS** file.
The final option indicates the revision list to operate on. Providing a single value here, in this case, `HEAD`, tells Git to apply `filter-branch` to all revisions from `HEAD` to as far back in history as Git can go with this commit's ancestors.
Git spits out multiple lines of output that tell you what it's doing. Here's one line from my output; yours may be slightly different:

```
Rewrite baf24aa0291b3364a0762c36509ab4c433527546 (39/40) (2 seconds passed, remaining 0 predicted)    rm 'SECRETS'
```

Git has stepped through every commit from `HEAD` back in time, performed the specified `git rm` command, and then re-committed the change. To prove this, look for the **SECRETS** file:
`git log -- SECRETS`

You'll get nothing back, telling you that Git's log knows nothing about this **SECRETS** file you're asking for.
Now, it seems like you've removed every single trace of this file, but there's one small clue that might tell someone you've removed something from the repository.
The original commit that added this file is still around. Execute **`git log --oneline --graph`**, scroll down, and you'll see the original commit that added this secret file:

```
* | dcbdf0c Adding a new secret
```

Then, look at the patch of the commit using the following command:

```
git log -p -1 dcbdf0c
```

Git shows you the metadata, but the patch itself is empty:

```
commit dcbdf0c2b3b5cf06eafd5dc6e441c8ab3a1d2ed5
Author: Will <will@example.com>
Date:   Mon Jul 3 14:10:59 2017 +0700

    Adding a new secret
```

Although no one can tell what the secret *was*, it would be nice to get rid of that commit entirely since it's empty. That's as simple as using another option to `filter-branch`: `--prune-empty`. If your author had had the foresight to tell you to use it in the first place, then you could have just tacked this on as an option to your original command.
But, Git is not a vengeful deity; you can run `filter-branch` again to clean things up. Execute the following command to run through your repository again and remove any "empty" commits:

```
git filter-branch --prune-empty -f HEAD
```

This simply runs through your repository, removing any commits that have an empty patch. Again, the `-f` command forces Git to perform `filter-branch`, disregarding any previous backups it may have saved from previous `filter-branch` operations.
Pull up your log again with `git log --oneline --graph` and scroll around; the commit is now gone.
Now that you're an expert on rewriting the history of your repository, it's time for your challenge for this chapter. It will bring things full circle and deal with that poor little **IGNORE_ME** file you were working with earlier.

Challenge: Remove IGNORE_ME from the repository

Now that you've learned how to eradicate any trace of a file from a repository, you can go back and remove all traces of **IGNORE_ME** from your repository.
You previously removed all traces of **SECRETS** from your repository, but that took you two steps. The challenge here is to do the same in one single command:
Use `git filter-branch`.
Use `--index-filter` to rewrite the index.
You can use a similar `git rm` command, but remember, you're filtering on a different file this time.
Use `--prune-empty` to remove any empty commits.
Remember that you want to apply this to all commits, starting at `HEAD` and going back.
You'll need to use `-f` to force this `filter-branch` operation, since you've already done a `filter-branch` and Git has stored a backup of that operation for you.
Note: If Git balks, check that the positioning of your options is correct in your command.
If you want to check your answer, or need a bit of help, you can find the answer to this challenge in the **challenge** folder included with this chapter.

Key points

.gitignore works by comparing files in the staging area, or index, to what's in your working tree.
.gitignore won't filter out any files already present in the index.
`git status --ignored` shows you the files that Git is currently ignoring.
`git update-index --assume-unchanged <filename>` tells Git to always assume that the file contained in the index will never change. This is a quick way to work around a file that isn't being ignored.

`git rm --cached <filename>` removes a file from the index but leaves the original file in your working tree.
`git rm --cached --ignore-unmatch <filename>` will succeed, returning an exit code of `0`, if `git rm` doesn't match on a file in the index. This is important when you use this command in conjunction with `filter-branch`.
`git filter-branch -f --index-filter 'git rm --cached --ignore-unmatch -- <filename>' HEAD` will modify any matching commits in the repository to remove `<filename>` from their contents.
The `--prune-empty` option will remove any commits from the repository that are empty after your `filter-branch`.

Where to go from here?

What you've learned in this chapter will usually serve you well when you've committed something to your repository that you didn't intend to be there.
The reverse case is fairly common, as well: You don't have something in your repository, but you know that bit of code or that file exists in another branch or even in another repository.
You've seen how you can selectively remove changes from your repository with `filter-branch`. Eventually, though, you'll hit a scenario where you mess something up, and you just need a good old-fashioned "undo" button to fix things. Again, Git has not one but several ways to "undo" what you've done – all of which you'll learn about in the next chapter.

7. The Many Faces of Undo 5. Rebasing to Rewrite History

Have a technical question? Want to report a bug? You can ask questions and report bugs to the book authors in our official book forum here.