4

Demystifying Rebasing Written by Jawwad Ahmad & Chris Belanger

Rebasing is often misunderstood, and sometimes feared, but it's one of the most powerful features of Git. Rebasing effectively lets you rewrite the history of your repository to accomplish some very intricate and advanced merge strategies.

Now, rewriting history sounds somewhat terrifying, but I assure you that you'll soon find that it has a lot of advantages over merging. You just have to be sure to rebase responsibly.

Why would you rebase?

Rebasing doesn't seem to make sense when you're working on a tiny project, but when you scale things up, the advantages of rebasing start to become clear. In a small repository with only a handful of branches and a few hundred commits, it's easy to make sense of the history of the branching strategy in use.

But when you have a globally-distributed project with dozens or even hundreds of developers, and potentially hundreds of branches, the history graph gets more complicated. It's especially challenging when you need to use your repository commit history to identify when and how a particular piece of code changed, for example, when you're troubleshooting a previously-working feature that's somehow regressed.

Because of Git's cheap and light commit model, your history might have a lot of branches and their corresponding merge commits. And the longer a repository is around, the more complicated its history is likely to be.

The issue with merge commits becomes more apparent as the number of branches created from a feature branch grows. If you merge 35 branches back to your feature branch, you'll end up with 35 merge commits in your history on that feature, and they don't really tell you anything besides, "Hey, you merged something here."

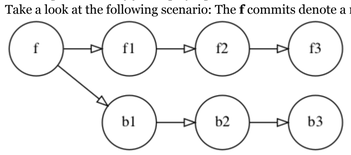
While that can often be useful, if the development workflow of your team results in fast, furious and short-lived branches, you might benefit from limiting merge commits and rebasing limited-scope changes instead. Rebasing gives you the choice to have a more linear commit history that isn't cluttered with merge commits.

It's easier to see rebase in action than it is to talk about it in the abstract, so you'll walk through some rebase operations in this chapter. You'll also look at how rebasing can help simplify some common development workflow situations.

What is rebasing?

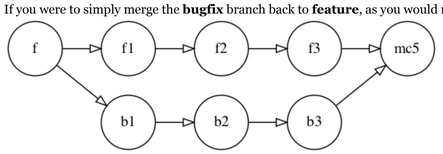
Rebasing is essentially just replaying a commit or a series of commits from history on top of a different commit in the repository. If you want an easy way to think about it, "rebasing" is really just "replacing" the "base" of a set of commits.

Take a look at the following scenario: The **f** commits denote a random **feature** branch, and the **b** commits denote a **bugfix** branch you created in order to correct or improve something inside the feature branch, without impeding work on the feature development. You've made a few commits along the way, and now it's time to merge the work in the **bugfix** branch back to **feature**.



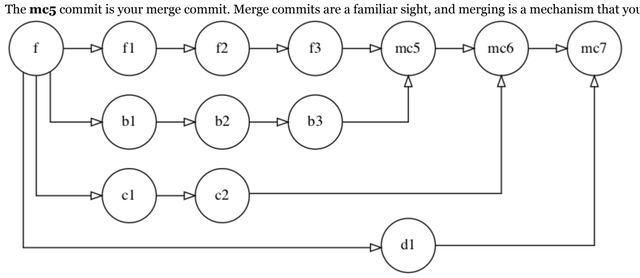
A simple branch (b) off of feature (f).

If you were to simply merge the **bugfix** branch back to **feature**, as you would normally tend to do, then the resulting history graph would look like this:



A simple branch (b) off of feature (f), merged back to feature with merge commit mc5.

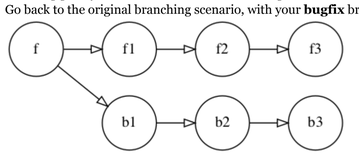
The **mc5** commit is your merge commit. Merge commits are a familiar sight, and merging is a mechanism that you and most everyone else who uses Git understands well. But as the size and activity of your repository grow, you can end up with a very complicated graph.



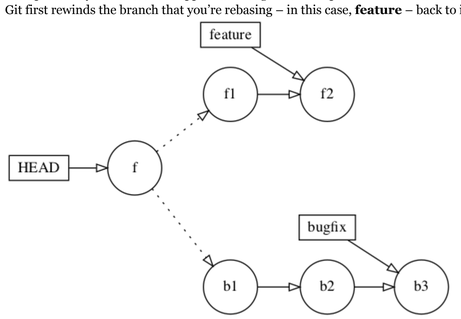
A more complex set of branches and merge commits, with merge commits mc5, mc6 and mc7.

Depending on the kind of work you're doing, you may not want to have your repository history show that you branched off, did some work and merged the changes back in. That little bit of extra cognitive overhead starts to add up as you try to make sense of months or even years of history graphs. Especially with small or trivial changes, you might prefer a linear history over seeing the code branched off and merged in again.

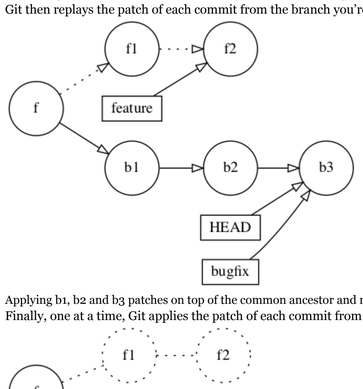
Rebasing gives you the freedom to avoid retaining branch history and merge commits. Instead, you can recreate your work as a linear commit progression.
Go back to the original branching scenario, with your **bugfix** branch off of **feature**:



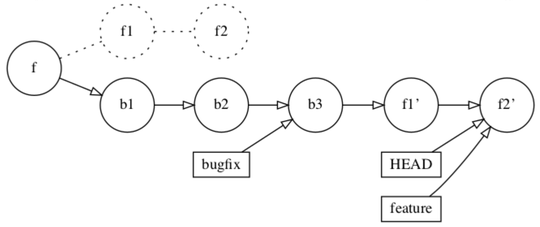
A simple branch (b) off of feature (f).
Rebasing uses a series of standard Git operations under the hood to accomplish rebasing. It isn't quite as straightforward as simply moving commits around as you'd move nodes in a tree data structure, for instance.
Let's presume you wanted it to appear as though the work performed on **feature** followed the work you did on **bugfix**. In Git parlance, you'd be rebasing **feature** on top of **bugfix**.
Git first rewinds the branch that you're rebasing – in this case, **feature** – back to its common ancestor with **bugfix**. The common ancestor is commit **f**:



Rewinding HEAD to the common ancestor of the feature and bugfix branches
Git then replays the patch of each commit from the branch you're rebasing on top of, in this case, **bugfix**, and moves the `HEAD` label along:



Applying b1, b2 and b3 patches on top of the common ancestor and moving labels along.
Finally, one at a time, Git applies the patch of each commit from the branch you're rebasing, in this case, **feature**, and moves the `HEAD` and `feature` labels along:



Applying f1, and f2 patches on top of the new base branch and moving labels along
At this point, you no longer have any real reference to those original commits from the **feature** branch. Git will eventually just collect those orphaned commits (or "loose" objects, as they're known) and clean them up in a regular garbage collection process. Although the orphaned commits still "know" who their parent is, you won't see these commits show up in the history graph.
It's like those commits never even existed. That's where the whole idea comes from that Git rebase is, quite literally, rewriting history. For all intents and purposes, anyone looking at the repository has no reason to believe that you didn't just make those commits on top of **feature** in the first place.
It's important to understand that Git is not just *moving* commits here; it's actually creating a *brand new* commit based on the contents of the patch it calculated at each commit in your branch.
Note: Choose to rebase only when the branch you're on is not shared with anyone else because, once again, you're rewriting the history of the repository. If you *must* rebase a shared branch, you'll have to coordinate this your team. It will be easier if everyone has pushed any and all changes to the branch and deleted it locally before you begin your work. Otherwise, *you're gonna have a bad time.*

Creating your first rebase operation

To start, you can continue using the your project from the previous chapter or use the project in the **starter** folder for this chapter.
You'll create an extremely trivial branch off of **wValidator**, make a change on that branch, and then rebase **wValidator** on top of your branch.
First, checkout the **wValidator** branch:

```
git checkout wValidator
```

Next create a new branch named **cValidator** from **wValidator**:

```
git checkout -b cValidator
```

Next, open up **README.md** and add **Chris** to the end of the `# Maintainers` section:

```
# Maintainers

This project is maintained by teamWYXZ:
- Will
- Yasmin
- Xanthe
- Zack
- Chris
```

Save your changes and exit the editor.
Now stage your changes:

```
git add .
```

And commit your staged changes with an appropriate message:

```
git commit -m "Added Chris as a new maintainer to README.md"
```

At this point, you have a branch **cValidator** with a commit containing changes to README.md. Now, you want to simulate someone creating more commits on the **wValidator** branch.
Switch back to the **wValidator** branch:

```
git checkout wValidator
```

Open **README.md** and add Chris's initial **C** to the end of the team name in the `# Maintainers` section:

```
# Maintainers

This project is maintained by teamWYXZC:
```

(A bit of alphabet soup, that is: “teamWXYZC”. You should petition the team to get a really cool name someday. But that’s for later.)
Save your changes and exit the editor.
Now you’ll stage and commit your changes, however instead of using `git add .` and `git commit -m` separately you’ll use `git commit -am`. As a reminder, the only difference is that `git commit -am` won’t commit newly added files.
Run the following to stage and commit your changes:

```
git commit -am "Updated team acronym to teamWYXZC"
```

Take a quick look at the current state of the repository in graphical form:

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

The top three lines show you what’s what:

```
* 9f5e491 (HEAD -> wValidator) Updated team acronym to teamWYXZC
| * 7f754d4 (cValidator) Added Chris as a new maintainer to README.md
|/
* 3574ab3 (origin/wValidator) Whoops—didn't need to call that one twice
```

You have a commit in a separate branch, **cValidator**, that you’d like to rebase **wValidator** on top of. While you could merge this in as usual with a merge commit, there’s really no need, since the change is so small and the changes in each branch are trivial and related to each other.
To rebase **wValidator** on top of **cValidator**, you need to be on the **wValidator** branch. Since you’re already there now, execute the rebase with the following command:

```
git rebase cValidator
```

Git shows a bit of output, telling you what it’s doing:

```
Successfully rebased and updated refs/heads/wValidator.
```

As expected, Git rewinds `HEAD` to the common ancestor — commit `3574ab3` in the graph shown above. It then applies each commit from the *branch you are on* — i.e., the branch that’s being rebased — on top of the end of the *branch you are rebasing onto*. In this case, the only commit from **wValidator** Git has to apply is `9f5e491 - Updated team acronym to teamWYXZC`.
Take a look at the history graph to see the end result by executing the following:

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

You’ll see the following linear activity at the top of the graph:

```
* 0e84d2b (HEAD -> wValidator) Updated team acronym to teamWYXZC
* 7f754d4 (cValidator) Added Chris as a new maintainer to README.md
* 3574ab3 (origin/wValidator) Whoops—didn't need to call that one twice
```

For a bit of perspective, you can look at the simple graphs at the start of the chapter for a visual reference to what’s happened here. But here’s the play-by-play to show you each of the steps:
Git rewound back to the common ancestor (`3574ab3`).
Git then replayed the **cValidator** branch commits (in this case, just `7f754d4`) on top of the common ancestor.
Git left the branch label `cValidator` attached to `7f754d4`.
Git then replayed the patches from each commit in **wValidator** on top of the commits from **cValidator** and moved the `HEAD` and `wValidator` labels to the tip of this branch.
You don’t need that **cValidator** branch anymore, nor is instructive to keep that label hanging around in the repository, so clean up after yourself with the following command:

```
git branch -d cValidator
```

As an aside, did you notice the difference in the commit hashes?
Old commit for `Updated team acronym to teamWYXZC`: `9f5e491`
New commit for `Updated team acronym to teamWYXZC`: `0e84d2b`
They’re different because what you have at the tip of **wValidator** is *a brand-new commit* — not just the old commit tacked onto the end of the branch.
You may be wondering where that old commit went, and you’ll dig into those details just a little further into this chapter as you investigate a more common scenario where you’ll encounter and resolve rebase conflicts.

A more complex rebase

Let’s go back to our Magic Square development team. Several people have been working on the Magic Squares app; Will in particular has been working on the wValidator branch.
Xanthe had branched off of Will’s wValidator branch to work on some refactoring, and it’s now time to bring everything back into the wValidator branch.
To see the history for just those 2 branches instead of all branches you can pass in the branch names instead of `--all` to the previous `git log --oneline --graph` command. So the command would be `git log --oneline --graph wValidator xValidator`. However this won’t quite work since you don’t have a local copy of the `xValidator` branch checked out.
So you have two choices. You can either checkout a local copy of `xValidator` or use the `origin/xValidator` remote tracking branch directly.
You don’t necessarily need a local `xValidator` branch, and you’d end up deleting it in the end anyway so you’ll just use the `origin/xValidator` remote tracking branch directly for the log and rebase commands.
Run the following command to take a look at the history for both branches.

```
git log --oneline --graph wValidator origin/xValidator
```

Here’s what the repository history looks like at this point:

```
* 0e84d2b (HEAD -> wValidator) Updated team acronym to teamWYXZC
* 7f754d4 Added Chris as a new maintainer to README.md
* 3574ab3 (origin/wValidator) Whoops—didn't need to call that one twice
* 43d6f24 check05: Finally, we can return true
| * 8ef01ac (origin/xValidator) Refactoring the main check function
| * 5fea71e Removing TODO
|/
* bf3753e check04: Checking diagonal sums
.
.
.
```

From this you can see that the common ancestor of both branches is `bf3753e`. There have been four commits to `wValidator` and two commits to `origin/xValidator` since then.
Although you could just merge all of Xanthe’s work into Will’s branch, you’d end up with a merge commit and clutter the history a little. And, conceptually, it makes sense to rebase in this situation, because the refactoring that Xanthe has done is within the logical context of Will’s work, so you might as well make it appear that the work has all taken place on a common branch.
Now, begin the rebase operation with the `git rebase` command, where you indicate which branch you want to rebase your current branch onto:

```
git rebase origin/xValidator
```

You’ll see the following merge conflict which you’ll learn how to resolve in the next section.

```
Auto-merging js/magic_square/validator.js
CONFLICT (content): Merge conflict in js/magic_square/validator.js
error: could not apply 43d6f24... check05: Finally, we can return true
Resolve all conflicts manually, mark them as resolved with
"git add/rm <conflicted_files>", then run "git rebase --continue".
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git rebase --abort".
Could not apply 43d6f24... check05: Finally, we can return true
```

Resolving errors

In the second line of the output from `git rebase` you’ll see that there’s a merge conflict in **js/magic_square/validator.js**.
Open up **js/magic_square/validator.js** in an editor and you’ll see the conflict that you need to resolve.
You’ll see three sections of code between the conflict markers:

```
<<<<<< HEAD
    Section 1: Xanthe's changes from origin/xValidator
|||||| parent of 43d6f24
    Section 2: Code before changes in the common ancestor
======
    Section 3: Will's changes on wValidator
>>>>>> 43d6f24
```

Section 1: Xanthe’s changes

The lines in the first section between `<<<<<<< HEAD` and `||||||| parent of 43d6f24` are Xanthe’s refactored changes from the `origin/xValidator` branch.
You might be confused here. If `HEAD` denotes the tip of the branch you’re on, and you were on `wValidator` before you started the rebase, why do these show changes from Xanthe’s `origin/xValidator` branch?
This is because in a rebase situation `HEAD` was first moved to the tip of the branch you’re rebasing on top of, i.e. the `origin/xValidator` branch.
As Git replays each commit onto this branch, `HEAD` moves along with each replayed commit.

Section 2: Code before changes

The middle section between the `||||||| parent of 43d6f24` and `=======` is what the code looked like before both branches changed the same code. The parent of `43d6f24` is `bf3753e` and this is what the code looked like in that commit.
Also recall that you are seeing this additional section because of the diff3 conflict style setting that was set in Chapter 2 via the `git config merge.conflictstyle diff3` command.

Section 3: Will's changes

Will's changes are in the third section between the `=======` and `>>>>>>>` conflict markers. If you take a closer look at the final conflict marker you'll see that it also tells you in which commit the conflict occurred:

```
>>>>>>> 43d6f24 (check05: Finally, we can return true)
```

For complex merge conflicts, this little bit of extra information can be quite useful.

Resolving the conflict

In this case you want to keep the changes on Xanthe's branch. Keep the code in the first section, i.e. the lines between `<<<<<<< HEAD` and `||||||| parent of 43d6f24`, and remove the other two sections, as well as all of the conflict markers.
When you're done, return to the command line and continue the rebase with the following command:

```
git rebase --continue
```

Oh, but Git won't let you continue. It gives you the following message:

```
js/magic_square/validator.js: needs merge
You must edit all merge conflicts and then
mark them as resolved using git add
```

Again, because you're working within the context of a single commit, you need to stage those changes. Git rebases each of the original commits one at a time, so you need to deal with and add the changes from each commit resolution one at a time.
Execute the following command to stage those changes to continue:

```
git add .
```

Then continue with the rebase:

```
git rebase --continue
```

But, frustratingly, Git *still* won't let you continue:

```
Auto-merging js/magic_square/validator.js
CONFLICT (content): Merge conflict in js/magic_square/validator.js
error: could not apply 3574ab3... Whoops—didn't need to call that one twice
Resolve all conflicts manually, mark them as resolved with
"git add/rm <conflicted_files>", then run "git rebase --continue".
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git rebase --abort".
Could not apply 3574ab3... Whoops—didn't need to call that one twice
```

This is one of those instances where Git can appear to be *completely* dense. Doesn't Git know that you *just* ran `git add`? Doesn't it *see* that you *just* resolved those commits? Gitdammit.
Feel free to vent for a second, and then take a closer look at the conflict message. In the third line you'll see that it says `could not apply 3574ab3... Whoops—didn't need to call that one twice`, whereas before the conflict was with `43d6f24... check05: Finally, we can return true`.
So this is actually a new conflict in a different commit.
In this case you also want to keep Xanthe's changes verbatim and disregard Will's changes completely.
While you could do the same thing you did before, there is actually an easier way to take the `HEAD` commit, i.e. Xanthe's commit verbatim with no changes, while discarding all the changes from the conflicting commit.
Run the following command to completely skip applying the `3574ab3 - Whoops—didn't need to call that one twice` commit:

```
git rebase --skip
```

Git then carries on and attempts to apply the second commit:

```
Successfully rebased and updated refs/heads/wValidator.
```

And you're done with that frustrating, yet enlightening journey through Git rebasing.
To see the result of your work from the perspective of Git, have a look at your history graph again for the two branches:

```
git log --oneline --graph wValidator origin/xValidator
```

You'll see the following:

```
* ed0c808 (HEAD -> wValidator) Updated team acronym to teamWYXZC
* d1ff29f Added Chris as a new maintainer to README.md
* 8ef01ac (origin/xValidator) Refactoring the main check function
* 5fea71e Removing TODO
* bf3753e check04: Checking diagonal sums
```

You'll see that the two conflicting commits from wValidator are gone (the commits with short hash `43d6f24` and `3574ab3`). You essentially skipped both of them. You skipped the first by manually keeping only Xanthe's changes, and you skipped the second one with the `git rebase --skip` command.
So does this mean you could have just run `git rebase --skip` on the first conflict as well? Indeed! And the next time you see yourself just keeping the changes from the `HEAD` commit verbatim, you'll remember that you can just run `git rebase skip`.
You can also see Xanthe's two commits (`5fea71e` and `8ef01ac`) are now neatly tucked in between the common ancestor `bf3753e` and your updates to README.md
If you'd just done a simple merge, you would have still seen them in the history of the repo. For example if you had originally ran `git merge origin/xValidator` while on the `wValidator` branch this is what the history would have looked like:

```
*   44194fa (HEAD -> wValidator) Merge remote-tracking branch 'origin/xValidator' into wValidator
|\
| * 8ef01ac (origin/xValidator) Refactoring the main check function
| * 5fea71e Removing TODO
* | 0e84d2b Updated team acronym to teamWYXZC
* | 7f754d4 Added Chris as a new maintainer to README.md
* | 3574ab3 (origin/wValidator) Whoops—didn't need to call that one twice
* | 43d6f24 check05: Finally, we can return true
|/
* bf3753e check04: Checking diagonal sums
```

Looking at this would have been confusing because you'd still see the original `43d6f24` and `3574ab3` commits in the history, which would have essentially been undone in the conflict resolution of the `44194fa` merge commit.
And if you hadn't rebased Chris's update, the `cValidator` branch would have looked even more confusing!

```
*   96f42e3 (HEAD -> wValidator) Merge remote-tracking branch 'origin/xValidator' into wValidator
|\
| * 8ef01ac (xValidator) Refactoring the main check function
| * 5fea71e Removing TODO
* |   b567a15 Merge branch 'cValidator' into wValidator
|\ \
| * | 9443e8d (cValidator) Added Chris as a new maintainer to README.md
* | | 76bacc5 Updated team acronym to teamWYXZC
|/ /
* | 3574ab3 Whoops—didn't need to call that one twice
* | 43d6f24 check05: Finally, we can return true
|/
* bf3753e check04: Checking diagonal sums
```

Orphaned commits

Even though the skipped commits are no longer shown in the log, you can still find these commits if you still have the original commit hash for them.
Execute the following command to see three commits, starting at the "Whoops—didn't need to call that one twice" commit of `3574ab3`:

```
git log --oneline -3 3574ab3
```

That shows original the history of the wValidator branch, from `3574ab3` back, as you understood it before you started rebasing:

```
3574ab3 (origin/wValidator) Whoops—didn't need to call that one twice
43d6f24 check05: Finally, we can return true
bf3753e check04: Checking diagonal sums
```

But where are those commits? Essentially, those commits are orphaned, or "loose" as Git refers to them. They are no longer referenced from any part of the repository tree, except for their mention in the Git internal logs.
You can see that the object still exists inside the .git directory:

```
git cat-file -p 3574ab3
```

Git returns with the commit metadata:

```
tree 1b4c07023270ed26167d322c6e7d9b63125320ef
parent 43d6f24d140fa63721bd67fb3ad3aafa8232ca97
author Will <will@example.com> 1499074126 +0700
committer Sam Davies <sam@razeware.com> 1499074126 +0700

Whoops—didn't need to call that one twice
```

But as you saw from the repository history tree above, that actual commit is no longer referenced anywhere. It's just sitting there until Git does its usual garbage collection, at which point Git will physically delete any loose objects that have been hanging around too long.
Note: Git generally tries to be as paranoid as possible when running garbage collection. It doesn't clean up every single loose object it finds, because there *might* be a chance that you made a mistake and really need the code from that commit.
In fact, even though the commit isn't referenced anywhere, as long as you know the hash of that commit from the logs, you can still check it out and work with the code inside. So Git, like any good developer, will keep those files hanging around for a while... *juuuuust* in case you need them later. Thanks, Git!

Merging vs rebasing

Although the politics and goals of your development team will dictate your approach to merging and rebasing, here are some pragmatic tips on when rebasing might be more appropriate over merging, and vice versa:
Choose to rebase when grouping the changes in a linear fashion makes contextual sense, such as Will's and Xanthe's work above that's contained to the same file.
Choose to merge when you've created major changes, such as adding a new feature in a pull request, where the branching strategy will give context to the history graph. A merge commit will have the history of both common ancestors, while rebasing removes this bit of contextual information.
Choose to rebase when you have a messy local commit or local branching history and you want to clean things up before you push. This touches on what's known as **squashing**, which you'll cover in a later chapter.
Choose to merge when having a complex history graph doesn't affect the day-to-day functions of your team.
Choose to rebase when your team frequently has to work through the history graph to figure out who changed what and when. Those merge commits add up over time!
There's a long, political history surrounding rebasing in Git, but hopefully, you've seen that it's simply another tool in your arsenal. Rebasing is most useful in your local, unpushed branches, to clean up the unavoidably messy business of coding.
But you've only begun your journey with rebasing. In the next chapter, you'll learn about **interactive rebasing**, where you can literally rewrite the history of the entire repository, one commit at a time.

Challenge: Rebase on top of another branch

You've discovered that Zach has also been doing a bit of refactoring on the **zValidator** branch with the range checking function:
Run the following to see the relationship between the two branches:

```
git log --oneline --graph wValidator origin/zValidator
```

You'll see that there is one commit on `origin/zValidator` after it branched from `wValidator`

```
...
* 64b1b51 check02: Checking the array contains the correct values
| * 136dc26 (origin/zValidator) Refactoring the range checking function
|/
* 665575c util02: Adding function to check the range of values
```

Your challenge is to rebase the work you've done on the `wValidator` branch on top of Zach's `zValidator` branch. Again, the shared context here and the limited scope of the changes mean you don't need a merge commit.
And don't worry, you won't have to resolve any merge conflicts!
As always, if you need help, or want to be sure that you've done it properly, you can always find the solution under the **challenge** folder for this chapter.

Key points

Rebasing "replays" commits from one branch on top of another.
Rebasing is a great technique over merging when you want to keep the repository history linear and as free from merge commits as possible.
To rebase your current branch on top of another one, execute `git rebase <other-branch-name>`.
You can resolve rebase conflicts just as you do merge conflicts.
To resume a rebase operation after resolving conflicts and staging your changes, execute `git rebase --continue`.
To skip rebasing a commit on top of the current branch, execute `git rebase --skip`.

5. Rebasing to Rewrite History 3. Stashes