5

Rebasing to Rewrite History Written by Jawwad Ahmad & Chris Belanger

As you saw in the previous chapter, rebasing provides you with an excellent alternative to merging. But rebasing also gives you the ability to reorganize your repository's history. You can reorder, rewrite commit messages and even squash multiple commits into a single, tidy commit if you like.

Just as you'd tidy up your code before pushing your local branch to a remote repository, rebasing lets you clean up your commit history before you push to remote. This gives you the freedom to commit locally as you see fit, then rearrange and combine your commits into a handful of semantically-meaningful commits. These will have much more value to someone (even yourself!) who has to comb through the repository history at some point in the future.

Note: Again, a warning: Rebasing in this manner is best used for branches that you haven't shared with anyone else. If you must rebase a branch that you've shared with others, then you must work out an arrangement with everyone who's cloned that repository to ensure that they all get the rebased copy of your branch. Otherwise, you're going to end up with a very complicated repository cleanup exercise at the end of the day.

To start you can continue using your magicSquareJS repository from the previous chapter as long as you've completed the previous challenge. Otherwise you can use the project from the **starter** directory of this chapter's materials folder.

Reordering commits

You'll start by taking a look at Will's `wValidator` branch. Execute the following to see what the current history looks like:

```
git log --oneline --graph wValidator
```

You'll see the following at the top of your history graph:

```
* dc24e14 (HEAD -> wValidator) Updated team acronym to teamWYXZC
* 72c4e8f Added Chris as a new maintainer to README.md
* 4c5274c Refactoring the main check function
* f0f212f Removing TODO
* d642b89 check04: Checking diagonal sums
* 791744c util06: Adding a function to check diagonals
* 556b640 check03: Checking row and column sums
* 75a6d1f util05: Fixing comment indentation
* 72ec86a util04: Adding a function to check column sums
* 34a656c util03: Adding function to check row sums
* 0efbaf2 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
* 0fc1a91 check01: checking that the 2D array is square
* 5ec1ccf util01: Adding the checkSqaure function
* 69670e7 Adding a new secret
```

It's not *terrible*, but this could definitely use some cleaning up. Your task is to combine those two trivial updates to **README.md** into one commit. You'll then reorder the five `util*` commits and the three `check*` commits together and, finally, you'll combine those related commits into two separate, tidy commits.

Interactive rebasing

First up: Combine the two top commits into one, inside the current branch. You're familiar with rebasing branches on top of other branches, but in this chapter, you'll rebase commits on top of other commits in the same branch.

In fact, since a branch is simply a label to a commit, rebasing branches on top of other branches really *is* just rebasing commits on top of one another.

But since you want to manipulate your repository's history along the way, you don't want Git to just replay commits on top of other commits. Instead, you'll use **interactive rebase** to get the job done.

First, get your game plan together. You want to combine, or **squash** those top two commits into one commit, give that new commit a clear message, and rebase that new squashed commit on top of the ancestor of the original commits. So your plan looks a little like the following:

Squash `dc24e14` and `72c4e8f`.

Create a new commit message for this squashed commit.

Rebase the resulting new commit on top of `4c5274c`.

To start an interactive rebase, you need to use the `-i` (`--interactive`) flag. Just as before, you need to tell Git where you want to rebase on top of; in this case, `4c5274c`.

You should already be on the `wValidator` branch. Execute the following to start your first Git interactive rebase:

```
git rebase -i 4c5274c
```

Git opens up the default editor on your system, likely Vim, and shows you the following:

```
pick 72c4e8f Added Chris as a new maintainer to README.md
pick dc24e14 Updated team acronym to teamWYXZC

# Rebase 4c5274c..dc24e14 onto 4c5274c (2 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup [-C | -c] <commit> = like "squash" but keep only the previous
#                    commit's log message, unless -C is used, in which case
```

```
#                   keep only this commit's message; -c is same as -C but
#                   opens the editor
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified); use -c <commit> to reword the commit message
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
```

Well, that's different! You've seen Vim in action before to create commit messages, but this is something new. What's going on?

Squashing in an interactive rebase

Here, Git's taken all of the commits past your rebase point, `72c4e8f` and `dc24e14`, and put them at the top of the file with some rather helpful comments down below.
What you're doing at this step is effectively creating a script of commands for Git to follow when it rebases. Git will start at the top of this file and work downwards, applying each action it finds, in order.
To perform a squash of commits, you simply put the `squash` command on the line with the commit you wish to squash into the previous one. In this case, you want to squash `dc24e14`, the last commit, into `72c4e8f`.
Note: Git interactive rebase shows all commits in ascending commit order. This is a different order than what you're used to seeing in with `git log`, so be careful that you're squashing things in the correct direction!
Since you're back in Vim, you'll have to use Vim commands to edit the file. Use **j** to move your cursor to the start of the `dc24e14` line and press the `c` key, followed by the `w` key — this is the "change word" command, and it essentially deletes the word your cursor is on and puts you into insert mode.
So type **squash** right there. The top few lines of your file should now look as follows:

```
pick 72c4e8f Added new maintainer to README.md
squash dc24e14 Updated team acronym
```

That's all you need to do, so write your changes and quit with the familiar **Escape** + **:wq** + **Enter** combination.
Git then throws you straight back into *another* Vim editor, this one a little more familiar:

```
# This is a combination of 2 commits.
# This is the 1st commit message:

Added Chris as a new maintainer to README.md

# This is the commit message #2:

Updated team acronym to teamWYXZC

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Sat Aug 21 23:20:17 2021 -0500
#
# interactive rebase in progress; onto 4c5274c
# Last commands done (2 commands done):
#    pick 72c4e8f Added Chris as a new maintainer to README.md
#    squash dc24e14 Updated team acronym to teamWYXZC
# No commands remaining.
# You are currently rebasing branch 'wValidator' on '4c5274c'.
#
# Changes to be committed:
#       modified:   README.md
#
```

Okay, this is a commit message editor, which you've seen before. Here, Git helpfully shares the messages of all commits affected by this rebase operation. You can choose to keep or edit any one of those commit messages, or you can choose to create your own.

Creating the squash commit message

In this case, you'll just create your own. Clear this file as follows:

. Type **gg** to ensure you're on the first line of the file.

. Type **dG** (that's a capital "G") to delete all of the following lines from the file.
You now have a nice, clean file for a commit message. Press **i** to enter **insert mode** and then add the following message:

```
Added Chris to README and updated team acronym
```

Then save your changes with **Escape** + **:wq** + **Enter** to continue the rebase operation.
Git carries on, emitting a little output with the success message at the end:

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

Execute the following to see what the repository history looks like now on your current branch:

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

Look at the top two lines and you'll see the following (your hashes will be different, of course):

```
* 6c07391 (HEAD -> wValidator) Added Chris to README and updated team acronym
* 4c5274c Refactoring the main check function
```

Git has done just what you asked; it's created a new commit from the two old commits and rebased that new commit on top of the ancestor. To see the combined effect of squashing those two commits into one, check out the patch Git created for the squashed commit with the following command:

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

Take a look at the bottom of that output and you'll see the following:

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

There's the combined effect of merging those two commits into one and rebasing that change on top of the ancestor commit.

Reordering commits

The asynchronous and messy nature of development means that sometimes you'll need to reorder commits to make it easier to squash a set of commits later on. Interactive rebase lets you literally rearrange the order of commits within a branch. You can do this as often as you need, to keep your repository history clean.
Execute the following to see the latest commits on your branch:

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

Take a look at the order of the last dozen commits or so:

```
* 6c07391 (HEAD -> wValidator) Added Chris to README and updated team acronym
* 4c5274c Refactoring the main check function
* f0f212f Removing TODO
* d642b89 check04: Checking diagonal sums
* 791744c util06: Adding a function to check diagonals
* 556b640 check03: Checking row and column sums
* 75a6d1f util05: Fixing comment indentation
* 72ec86a util04: Adding a function to check column sums
* 34a656c util03: Adding function to check row sums
* 0efbaf2 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
* 0fc1a91 check01: checking that the 2D array is square
* 5ec1ccf util01: Adding the checkSqaure function
* 69670e7 Adding a new secret
```

There's a collection of commits there that would make more sense if you arranged them contiguously. There's one set of check functions commits (the `check0x` commits) and another set of utility functions (the `util0x` commits). Before you merge these to `main`, you'd like to squash these related sets of commits into two commits to keep your repository history neat and tidy.
First, you'll need to start with the common ancestor of all of these commits. In this case, the base ancestor commit of the commits you're concerned with is `69670e7`. That commit will be the base for your interactive rebase.
Execute the following to start the interactive rebase on top of that base commit:

```
git rebase -i 69670e7
```

Once again, you'll be launched into Vim to edit the rebase script for the rebase operation:

```
pick 5ec1ccf util01: Adding the checkSqaure function
pick 0fc1a91 check01: checking that the 2D array is square
pick 665575c util02: Adding function to check the range of values
pick 136dc26 Refactoring the range checking function
pick 0efbaf2 check02: Checking the array contains the correct values
pick 34a656c util03: Adding function to check row sums
pick 72ec86a util04: Adding a function to check column sums
pick 75a6d1f util05: Fixing comment indentation
pick 556b640 check03: Checking row and column sums
pick 791744c util06: Adding a function to check diagonals
pick d642b89 check04: Checking diagonal sums
pick f0f212f Removing TODO
pick 4c5274c Refactoring the main check function
pick 6c07391 Added Chris to README and updated team acronym

# Rebase 69670e7..6c07391 onto 69670e7 (14 commands)
```

Since Git starts at the top of the file and works its way down in order, you simply need to rearrange the lines in this file in contiguous order to rearrange the commits.
Since you're in Vim, you might as well use the handy Vim shortcuts to move lines around:
To "cut" a line into the clipboard buffer, type **dd**.
To "paste" a line into the edit buffer underneath the current line, type **p**.
Use these two key combinations to do the following:

. Move the `util01` line to just above the `util02` line.

. Leave the `Refactoring the range checking function` after `util02`.

. Move the `util03` through `util06` lines, in order, to follow the `Refactoring the range checking function` commit.

When you're done, your rebase script should look as follows:

```
pick 0fc1a91 check01: checking that the 2D array is square
pick 5ec1ccf util01: Adding the checkSqaure function
pick 665575c util02: Adding function to check the range of values
pick 136dc26 Refactoring the range checking function
pick 34a656c util03: Adding function to check row sums
pick 72ec86a util04: Adding a function to check column sums
pick 75a6d1f util05: Fixing comment indentation
pick 791744c util06: Adding a function to check diagonals
pick 0efbaf2 check02: Checking the array contains the correct values
pick 556b640 check03: Checking row and column sums
pick d642b89 check04: Checking diagonal sums
pick f0f212f Removing TODO
pick 4c5274c Refactoring the main check function
pick 6c07391 Added Chris to README and updated team acronym
```

Then, save your changes with **Escape** + **:wq** + **Enter** to continue the rebase operation.
Git continues with a little bit of output to let you know things have succeeded:

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

Now, take a look at the log with `git log --oneline --graph` and you'll see that Git has neatly reordered your commits, and added new hashes as well:

```
* 0d6790c (HEAD -> wValidator) Added Chris to README and updated team acronym
* aed8cbb Refactoring the main check function
* 8e6d0a8 Removing TODO
* 734676e check04: Checking diagonal sums
* 798260c check03: Checking row and column sums
* 7dd347e check02: Checking the array contains the correct values
* 7a6b92d util06: Adding a function to check diagonals
* c2fae05 util05: Fixing comment indentation
* 680edea util04: Adding a function to check column sums
* 48b6bcf util03: Adding function to check row sums
* 5cdbb66 Refactoring the range checking function
* beb30fb util02: Adding function to check the range of values
* 92b5982 util01: Adding the checkSqaure function
* 6adde96 check01: checking that the 2D array is square
* 69670e7 Adding a new secret
```

I want to stress once again that these are *new* commits, not simply the old commits moved around. And it's not just the commits you moved around inside the instruction file that have new hashes: Every single commit from your rebase script has a new hash — *because they are new commits*.

Rewording commit messages

If you take a look at the `util01` commit message, you'll notice that it's misspelled as "Sqaure" instead of "Square". As a word nerd, I can't leave that the way it is. But I can quickly use interactive rebase to change that commit message.
Note: In my repo, the commit has the hash of `92b5982`, while in your system, it will likely be different. Simply replace the hash below with the hash of the commit you want to rebase on top of — that is, the commit just before the one you want to change, and things will work just fine.
Execute the following to start another interactive rebase, indicating the commit you want to rebase on top of. In this instance, you want to rebase on top of the `check01: checking that the 2D array is square` commit:

```
git rebase -i 6adde96
```

When Vim comes up, you'll see the commit you'd like to change at the top of the list:

```
pick 92b5982 util01: Adding the checkSqaure function
```

Ensure your cursor is on that line, and type **cw** to cut the word `pick` and change to insert mode in Vim. In place of `pick`, type `reword` there, which tells Git to prompt you to reword this commit as it runs the rebase script.
When you're done, the very first line in the script should look as follows:

```
reword 92b5982 util01: Adding the checkSqaure function
```

Note: You're not fixing the commit message in this step; rather, you'll wait for Git to prompt you to do it when the rebase script runs.
Save your work with **Escape** + **:wq** + **Enter** and you'll immediately be put back into Vim. This time, Git's asking you to actually modify the commit message.
Press **i** to enter insert mode, cursor over to that egregious misspelling, change the word `checkSqaure` to `checkSquare`, and save your work with **Escape** + **:wq** + **Enter**.
Git completes the rebase and drops you back at the command line.
You can see that Git has changed the commit message for you by executing `git log --oneline --graph` and scrolling down to find your new, rebased commit. Also note that all commits applied after that have new commit hashes as well.

```
// Hash updated from 5cdbb66 to 6a9a50d
* 6a9a50d Refactoring the range checking function

// Hash updated from beb30fb to 9923d78
* 9923d78 util02: Adding function to check the range of values

// Hash updated from 92b5982 to 0be4b0c and spelling updated
* 0be4b0c util01: Adding the checkSquare function

// Commit before rebase has same hash of 6adde96
* 6adde96 check01: checking that the 2D array is square
```

The fixed spelling is a small thing, to be sure, but it's a *nice* thing.

Squashing multiple commits

Now that you have your utility functions all arranged contiguously, you can proceed to squash these commits into one.
Again, you'll launch an interactive rebase session with the hash of the commit you want to rebase on top of. You want to rebase on top of the `Adding a new secret` commit, which is still `69670e7`. Remember: When you rebase *on top* of a commit, that commit doesn't change, so it still has the same hash as before. It's just the commits that follow that will get new hashes as each is rebased.
To start your adventure in squashing, execute the following to kick off another interactive rebase:

```
git rebase -i 69670e7
```

Once you're back in Vim, find the list of contiguous commits for the utility functions.
To squash a list of commits, find the first commit in the sequence you'd like to squash and leave that commit as it is. Then, on every subsequent line, change `pick` to `squash`. As Git executes this rebase script, each time it encounters `squash`, it will meld that commit with the commit on the previous line.
That's why you need to leave that first line unchanged: Otherwise, Git will squash *that* first commit into the previous commit, which isn't what you want. You want to squash this set of changes relevant to the utility functions as a nice tidy unit, not squash them into some random commit preceding them.
Note: You can use a bit of Vim-fu to speed things along here.
Type **cw** on the first commit you want to squash (the `util02` one) and change `pick` to `squash`.
Then press **Escape** to get back to command mode.
Cursor down to the start of the next commit you want to squash, and type `.` - a period. This tells Git "Do that same thing again, only on this line instead."
Continue on this way for all of the utility function commits. When you're done, your rebase script should look like the following:

```
pick 6adde96 check01: checking that the 2D array is square
pick 0be4b0c util01: Adding the checkSquare function
squash 9923d78 util02: Adding function to check the range of values
squash 6a9a50d Refactoring the range checking function
squash d761cfe util03: Adding function to check row sums
squash 98c44f8 util04: Adding a function to check column sums
squash 01688ee util05: Fixing comment indentation
squash fd28844 util06: Adding a function to check diagonals
pick 43a1bc1 check02: Checking the array contains the correct values
pick 60b55d2 check03: Checking row and column sums
pick 53a9b10 check04: Checking diagonal sums
pick 3a04bfb Removing TODO
pick d3eea6b Refactoring the main check function
pick 08c60cf Added Chris to README and updated team acronym
```

Save your changes with **Escape + :wq + Enter** and you'll be brought into another instance of Vim. This is your chance to provide a single, clean commit message for your squash operation.
Vim helpfully gives you a bit of context here, as it lists the collection of commit messages from the squash operation for context:

```
# This is a combination of 7 commits.
# This is the 1st commit message:

util01: Adding the checkSquare function

# This is the commit message #2:

util02: Adding function to check the range of values

# This is the commit message #3:

Refactoring the range checking function

# This is the commit message #4:

util03: Adding function to check row sums

# This is the commit message #5:

util04: Adding a function to check column sums

# This is the commit message #6:

util05: Fixing comment indentation

# This is the commit message #7:

util06: Adding a function to check diagonals
```

You could choose to reuse some of the above content for the squash commit message, but in this case, simply type **gg** to ensure you're on the first line and **dG** to clear the edit buffer entirely.
Press **i** to enter insert mode, and add the following commit message, to sum up your squash effort:

```
Creating utility functions for Magic Square validation
```

Save your changes with **Escape + :wq + Enter** and Git will respond with a bit of output to let you know it's done. Execute `git log --oneline` to see the result of your actions:

```
* f8d6e1b (HEAD -> wValidator) Added Chris to README and updated team acronym
* ededa27 Refactoring the main check function
* 2670142 Removing TODO
* d0bdabb check04: Checking diagonal sums
* 4424300 check03: Checking row and column sums
* 5e087af check02: Checking the array contains the correct values
* 1cb3ad3 Creating utility functions for Magic Square validation
* 6adde96 check01: checking that the 2D array is square
* 69670e7 Adding a new secret
```

Nice! You've now squashed all of the `util` commits into a single commit with a concise message.
But there's still a bit of work to do here: You also want to rearrange and squash the `check0x` commits in the same manner. And that, dear reader, is the challenge for this chapter!

Challenge 1: More squashing

You'd like to squash all of the `check0x` commits into one tidy commit. And you *could* follow the pattern above, where you first rearrange the commits in one rebase and then perform the squash in a separate rebase.
But you can do this all in one rebase pass:

. Figure out what your base ancestor is for the rebase.

. Start an interactive rebase operation.

. Reorder the `check0x` commits.

. Change the `pick` rebase script command to `squash` on all commits from the `check02` commit, down to and including the `Refactoring the main check function` commit.

. Save your work in Vim and exit.

. Create a commit message in Vim for the squash operation.

. Take a look at your Git log to see the changes you've made.

Challenge 2: Rebase your changes onto main

Now that you've squashed your work down to just a few commits, it's time to get wValidator back into the `main` branch. It's likely your first instinct is to merge `wValidator` back to `main`. However, you're a rebase guru by this point, so you'll rebase those commits on top of `main` instead:

. Execute `git rebase` with `main` as your rebase target.

. Crud — a conflict. Open README.md and resolve the conflict to preserve your changes, and move the changes to the `## Contact` section.

. Save your work.

. Stage those changes with `git add README.md`.

. Continue the rebase with `git rebase --continue`.

. Check the log to see where `main` points and where `wValidator` points.

. Check out the `main` branch.

. Execute `git merge` for `wValidator`. What's special about this merge that lets you avoid a merge commit?

. Delete the `wValidator` branch.

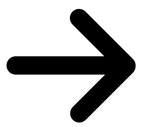
If you get stuck or need any assistance, you can find the solution for these challenges inside the **challenge** folder for this chapter.

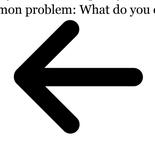
Key points

`git rebase -i <hash>` starts an interactive rebase operation.
Interactive rebases in Git let you create a "script" to tell Git how to perform the rebase operation
The `pick` command means to keep a commit in the rebase.
The `squash` command means to merge this commit with the previous one in the rebase.
The `reword` command lets you reword a particular commit message.
You can move lines around in the rebase script to reorder commits.
Rebasing creates new commits for each original commit in the rebase script.
Squashing lets you combine multiple commits into a single commit with a new commit message. This helps keep your commit history clean.

Where to go from here?

Interactive rebase is one of the most powerful features of Git because it forces you to think logically about the changes you've made, and how those changes appear to others. Just as you'd appreciate cloning a repo and seeing a nice, illustrative history of the project, so will the developers that come after you.
In the following chapter, you'll continue to use rebase to solve a terribly common problem: What do you do when you've already committed files that you want Git to ignore? If you haven't hit this situation in your development career yet, trust me, you will. And it's a beast to solve without knowing how to rebase!





6. Gitignore After the Fact 4. Demystifying Rebasing

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.