Up to date for Git 2.28

Advanced Git

FIRST EDITION

Understanding Git Internals & Commands

By the raywenderlich Tutorial Team

Jawwad Ahmad & Chris Belanger

Based on material by Sam Davies

Advanced Git

Jawwad Ahmad & Chris Belanger

Notice of Rights

Notice of Liability

Trademarks

Dedications

"For Russ and Skip."

— *Chris Belanger*

"To my parents, my wife, and three daughters, for their support and encouragement."

— *Jawwad Ahmad*

About the Author

Chris Belanger is an author of this book. He is the Editor-in-Chief of raywenderlich.com. If there are words to wrangle or a paragraph to ponder, he's on the case. In the programming world, Chris has over 25 years of experience with multiple database platforms, real-time industrial control systems, and enterprise healthcare information systems. When he kicks back, you can usually find Chris with guitar in hand, looking for the nearest beach, or exploring the lakes and rivers in his part of the world in a canoe.

Jawwad Ahmad is an author of this book. He is an iOS Developer that spends way too much time using the power of Git to attempt to craft the most ideal commits. He currently works as a Software Engineer at a technology company in the San Francisco Bay Area.

About the Editors

Bhagat Singh is a tech editor for this book. Bhagat started iOS Development after the release of Swift, and has been fascinated by it ever since. He likes to work on making apps more usable by building great user experiences and interactions in his applications. He also is a contributor in the Raywenderlich tutorial team. When the laptop lid shuts down, you can find him chilling with his friends and finding new places to eat. He dedicates all his success to his mother. You can find Bhagat on Twitter: @soulful_swift

Cesare Rocchi is a tech editor of this book. Cesare runs Studio Magnolia, an interactive studio that creates compelling web and mobile applications. He blogs at upbeat.it, and he's also building Podrover and Affiliator You can find him on Twitter at @_funkyboy.

Manda Frederick is an editor of this book. She has been involved in publishing for over ten years through various creative, educational, medical and technical print and digital publications, and is thrilled to bring her experience to the raywenderlich.com family as Managing Editor. In her free time, you can find her at the climbing gym, backpacking in the backcountry, working on poems, playing guitar and exploring breweries.

Sandra Grauschopf is an editor of this book. Sandra has over 20 years' experience as a writer, editor, copy editor, and content manager and has been editing tutorials at raywenderlich.com since 2018. She loves to travel and explore new places, always with a trusty book close at hand.

Aaron Douglas is the final pass editor for this book. He was that kid taking apart the mechanical and electrical appliances at five years of age to see how they worked. He never grew out of that core interest - to know how things work. He took an early interest in computer programming, figuring out how to get past security to be able to play games on his dad's computer. He's still that feisty nerd, but at least now he gets paid to do it. Aaron works for Automattic (WordPress.com, WooCommerce, Tumblr, SimpleNote) as a Mobile Lead primarily on the WooCommerce mobile apps. Find Aaron on Twitter as @astralbodies or at his blog at https://aaron.blog.

About the Artist

Vicki Wenderlich is the designer and artist of the cover of this book. She is Ray's wife and business partner. She is a digital artist who creates illustrations, game art and a lot of other art or design work for the tutorials and books on raywenderlich.com. When she's not making art, she loves hiking, a good glass of wine and attempting to create the perfect cheese plate.

Table of Contents: Overview

Table of Contents: Extended

Book License

By purchasing *Advanced Git*, you have the following license:

- You are allowed to use and/or modify the source code in *Advanced Git* in as many apps as you want, with no attribution required.
- You are allowed to use and/or modify all art, images and designs that are included in *Advanced Git* in as many apps as you want, but must include this attribution line somewhere inside your app: "Artwork/images/designs: from *Advanced Git*, available at www.raywenderlich.com".
- The source code included in *Advanced Git* is for your personal use only. You are NOT allowed to distribute or sell the source code in *Advanced Git* without prior authorization.
- This book is for your personal use only. You are NOT allowed to sell this book without prior authorization, or distribute it to friends, coworkers or students; they would need to purchase their own copies.

All materials provided with this book are provided on an "as is" basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

All trademarks and registered trademarks appearing in this guide are the properties of their respective owners.

Before You Begin

This section tells you a few things you need to know before you get started, such as what you'll need for hardware and software, where to find the project files for this book, and more.

What You Need

To follow along with this book, you'll need the following:

- **Git 2.28 or later.** Git is the software package you'll use for all the work in this book. There are installers for macOS, Windows and Linux available for free from the official Git page here: https://git-scm.com/downloads. We've tested this book on Git 2.28.0, but you can follow along with older versions of Git as well.

Book Source Code & Forums

Book source code

The materials for this book are all available in the GitHub repository here:

- https://github.com/raywenderlich/agit-materials/tree/editions/1.0

You can download the entire set of materials for the book from that page.

Forum

We've also set up an official forum for the book at https://forums.raywenderlich.com/c/books/advanced-git/. This is a great place to ask questions about the book or to submit any errors you may find.

About the Cover

Advanced Git

While not the most elegant or agile creature, the flightless penguin should not be underestimated. Very few other animals can boast the wide adaptability of these birds. Found in both global hemispheres, penguins are both animals of the land and the sea, spending half of their lives on each.

In water, they are independent, graceful swimmers and formidable hunters, feeding on fish, squid and other sea life as they swim and dive — sometimes up to depths of over 500 meters for up to 22 minutes at a time. On land — well, we know about penguins on land. Their colonies are a comical flurry of waddling, rock-hopping and belly sliding — but they are also social, gentle and maternal.

Like penguins, Git thrives in multiple environments and is incredibly adaptable, and its utility should not be underestimated. Though Git seems unassuming at first glance, not many other tools will allow you to leverage your work in so many environments, both independently and socially. And like these resilient birds who manage to slip and tumble, getting back up each time, Git will allow you to work knowing any mistake can be corrected. The key is just to keep waddling along.

It should also be noted that both penguins and the authors of this book look great dressed in tuxedos.

There are usually two reasons a person picks up a book about Git: one, they are unusually curious about how the software works at a deeper level; or two, they're frustrated and need something to solve their problems *now*.

Whatever situation brought you here, welcome! I'm happy to have you onboard. I came to write this book for *both* of the above reasons. I am a tinkerer and hacker by nature, and I love going deep into the internals of software to see what makes them tick. But I, like you, found Git at first to be an inscrutable piece of software. My brain, which had been trained in software development through the late 1990s, found version control packages like SVN soothing, with their familiar client-server architecture, Windows shell integration, and rather straightforward, albeit heavy, processes.

When I came to use Git and GitHub about seven years ago, I found it *inscrutable* at best; it seemed no matter which way I turned, Git was telling me I had a merge conflict, or it was merging changes from the master branch into my current branch, or quite often complaining about unstaged changes. And why was it called a "pull request", when clearly I was trying to *push* my changes into the master branch?

Little by little, I learned more about how Git worked; how to solve some of the common issues I encountered, and I eventually got to a point where I felt comfortable using it on a daily basis.

Enter the video courses

In early 2017, my colleague Sam Davies created a conference talk, titled "Mastering Git", and from that, two video courses at raywenderlich.com: "Beginning Git" and "Mastering Git". Those two courses form the basis of this book, but it always nagged me a little that, while Sam's video version of the material was quite pragmatic and tied nicely into using both the command line and graphical tools to solve common Git workflow problems, I always felt like there was a bit of detail missing; the kind of information that would lead a curious mind to say "I see the *how*, but I really want to know more about the *why*."

This book gives a little more background on the *why*: or, in other words, "*Why* the %^&$ did you do that to my repository, Git?!" Underneath the hood, you'll find that Git has a rather simple and elegant architecture, which is why it scales *so* well to the kinds of globally distributed projects that use Git as their version control software, via GitHub, GitLab, Bitbucket, or other cloud repository management solutions.

And while GUI-based Git frontends like Tower or GitHub Desktop are great at minimizing effort, they abstract you away from the actual guts of Git. That's why this book takes a command-line-first approach, so that you'll gain a better understanding of the various actions that Git takes to manage your repositories — and more importantly, you'll gain a better understanding of how to fix things when Git does things that don't seem to make much sense.

How to read this book

This book begins where the other Git book in our catalogue — *Git Apprentice* — ends.

That book covered Beginning Git. In that book, the chapters take you through concepts such as cloning, staging, committing, syncing, merging, viewing logs, and more. The very first chapter is a crash course on using Git, where you'll go through the basic Git workflow to get a handle on the *how* before you move into the *what* and the *why*.

The chapters work with a small repository that houses a simple ToDo system based on text files that hold ideas (both good and bad) ideas for content for the website. It's an ideal way to learn about Git without getting bogged down in a particular language or framework.

In this book, you will cover:

Section I: Advanced Git

If you've been using Git for a while, you may choose to start in this section first. If you know how to do basic staging, committing, merging and .gitignore operations, then you'll likely be able to jump right in here. This section walks you through concepts such as merge conflicts, stashes, rebasing, rewriting history, fixing .gitignore after the fact, and more.

If you've ever come up against a scenario where you feel you just need to delete your local repository and clone things fresh, then this section is just what you need to help you solve those sticky Git situations.

Section II: Workflows

This section takes a look at some common Git workflows, such as the feature branch workflow, Gitflow, a basic forking workflow, and even a centralized workflow. Because of the flexibility of Git, lots of teams have devised interesting workflows for their teams that work for them — but this doesn't mean that there's a single *right* way to manage your development.

Learning by doing

Above all, the best advice I can use is to *work* with Git: find ways to use it in your daily workflows, find ways to contribute to open-source projects that use Git to manage their repositories, and don't be afraid to try some of the more esoteric Git commands to accomplish something. There's little chance you're going to screw anything up beyond repair, and most developers learn best when they inadvertently back themselves into a technical rabbit-hole — then figure out how to dig themselves out.

A note on master vs. main

At the time that this book went to press, GitHub (and potentially other hosts) were proposing changing the name of the default repository branch to `main`, instead of `master`, in an attempt to use more culturally-aware language. So if you're working through this book and realize that some repos use `main` as the central reference branch, don't worry — simply use `main` in place of `master` where you need to in these commands. If the point comes when there seems to be a consensus on `main` vs `master` in the Git community, we'll modify the book to match.

I wish you all the best in your Git adventures. Time to *Git* going!

Section I: Advanced Git

This section dives into the inner workings of Git, what particular Git operations actually do, and will walk you through some interesting problem-solving scenarios when Git gets cranky. You'll build up some mental models to understand what's going on when Git complains about things to help you solve similar issues on your own in the future.

Chapter 1: How Does Git Actually Work?

By Chris Belanger

Git is one of those wonderful, elegant tools that does an amazing job of abstracting the underlying mechanism from the front-end workings. To pull changes from the remote down to the local, you execute `git pull`. To commit your changes in your local repository, you execute `git commit`. To push commits from your local repository to the remote repository, you execute `git push`. The front end does an excellent job of mirroring the mental model of what's happening to your code.

But as you would expect, a lot is going on underneath. The nice thing about Git is that you could spend your entire career not knowing how the Git internals work, and you'd get along quite well. But being aware of how Git manages your repository will help cement that mental model and give a little more insight into why Git does what it does.

To follow along, you can start with any repository. If you don't have one handy, you can use one of the repos provided with the materials for this book.

Everything is a hash

Well, not *everything* is a hash, to be honest. But it's a useful point to start when you want to know how Git works.

Git refers to all commits by their SHA-1 hashes. You've seen that many times over, both in this book and in your personal and professional work with Git. The hash is the key that points to a particular commit in the repository, and it's pretty clear to see that it's just a type of unique ID. One ID references one commit. There's no ambiguity there.

But if you dig down a little bit, the commit hash doesn't reference *everything* that has to do with a commit. In fact, a lot of what Git does is create references to references in a tree-like structure to store and retrieve your data, and its metadata, as quickly and efficiently as possible.

To see this in action, you'll dissect the "secret" files underneath the **.git** directory and see what's inside of each.

Dissecting the commit

Since the atomic particle of Git workflow is the commit, it makes sense to start there. You'll start walking down the tree to see how Git stores and tracks your work.

> **Note**: The commit hashes I'll use will be different than the ones in your repository. Simply follow the steps below, substituting in your hashes for the ones I have in my repository.

I'm going to pick one of my most recent commits that has a change that I made, as opposed to a merge, just to narrow down the set of changes I want to look at.

To get the list of the most recent five commits, execute the `git log` command as below:

```
git log -5 --oneline
```

My log result looks like the following:

```
f8098fa (HEAD -> master, origin/master, origin/HEAD) Merge
branch 'clickbait' with changes from crispy8888/clickbait
d83ab2b (crispy8888/clickbait, clickbait) Ticked off the last
item added
```

```
5415c13 More clickbait ideas
fed347d (from-crispy8888) Merge branch 'master' of https://
www.github.com/belangerc/ideas
ace7251 Adding debugging book idea
```

I'll select the commit with the short hash `d83ab2b` to start stepping through the tree structure. First, though, you'll need to get the long hash for this, instead of the short one. You'll see why this is in a moment.

You *could* simply run `git log` again without the `--oneline` option to get the long hash, but there's an easier way.

Converting short hash into long

Execute the command below to convert a short hash into its long equivalent, substituting your own short hash:

```
git rev-parse d83ab2b
```

Git responds with the long hash equivalent:
`d83ab2b104e4addd03947ed3b1ca57b2e68dfc85`.

Now, you need to start crawling through the Git tree to find out what this commit *looks* like on disk.

The inner workings of Git

Change to your terminal program and navigate to the main directory of your repository. Once you're there, navigate into the **.git** directory of your repository:

```
cd .git
```

Now, pull up a directory listing of what's in the **.git** directory, and have a look at the directories there. You should, at a minimum, see the following directories:

```
info/
objects/
hooks/
logs/
refs/
```

The directory you're interested in is the **objects** directory. In Git, the most common objects are:

- **Commits**: Structures that hold metadata about your commit, as well as the pointers to the parent commit and the files underneath.
- **Trees**: Tree structures of all the files contained in a commit.
- **Blobs**: Compressed collections of files in the tree.

Start by navigating into the **objects** directory:

```
cd objects
```

Pull up a directory listing to see what's inside, and you'll be greeted with the following puzzling list of directories:

```
02    14    39    55    6e    84    ad    c5    db    f8
05    19    3a    56    72    88    b4    c8    e0    f9
06    1a    3b    57    73    8b    b5    ca    e6    fb
0a    1c    3d    59    75    99    b8    ce    e7    fe
0b    24    3e    5d    76    9d    b9    cf    eb    ff
0c    29    43    5f    78    9f    ba    d2    ec    info
0d    2c    45    62    7a    a0    bb    d3    ed    pack
0e    33    47    65    7d    a1    be    d7    ee
0f    35    4e    67    7f    a4    bf    d8    f1
11    36    50    69    81    ab    c0    d9    f4
12    37    54    6c    83    ac    c4    da    f5
```

It's clear that this is a lookup system of some sort, but what does that two-character directory name mean?

The Git object repository structure

When Git stores objects, instead of dumping them all into a single directory, which would get unwieldy in rather short order, it structures them neatly into a tree. Git takes the first two characters of your object's hash, uses that as the directory name, and then uses the remaining 38 characters as the object identifier.

Here's an example of the Git **object** directory structure, from my repository, that shows this hierarchy:

```
objects
├── 02
│   ├── 1f10a861cb8a8b904aac751226c67e42fadbf5
```

```
│   └── 8f2d5e0a0f99902638039794149dfa0126bede
├── 05
│   └── 66b505b18787bbc710aeef2c8981b0e13810f9
├── 06
│   └── f468e662b25687de078df86cbc9b67654d938b
├── 0a
│   └── 795bccdec0f85ebd9411e176a90b1b4dfe2002
├── 0b
│   └── 2d0890591a57393dc40e2155bff8901acafbb6
├── 0c
│   └── 66fedfeb176b467885ccd1a1ec70849299eeac
├── 0d
│   └── dfac290832b19d1cf78284226179a596bf5825
├── 0e
│   └── 066e61ce93bf5dfaa9a6eba812aa62038d7875
├── 0f
│   └── a80ee6442e459c501c6da30bf99a07c0f5624e
├── 11
│   ├── 06774ed5ad653594a848631f1f2786a76a776f
│   ├── 92339da7c0831ba4448cb46d40e1b8c2bed12c
│   └── c1a7373df5a0fbea20fa8611f41b4a032b846f
.
.
.
```

To find the object associated with a commit, simply take the commit hash you found above:

```
d83ab2b104e4addd03947ed3b1ca57b2e68dfc85
```

Decompose that into a directory name and an object identifier:

- **Directory**: `d8`
- **Object identifier**: `3ab2b104e4addd03947ed3b1ca57b2e68dfc85`

Now you know that the object you want to look at is inside the **d8** directory. Navigate into that directory and pull up another listing to see the files inside:

```
.
.
.
d7
├── c33fdd7d35372cba78386dfe5928f1ba8dfb70
└── e92f9daeec6cd217fda01c6b726cb07866728c
d8
└── 3ab2b104e4addd03947ed3b1ca57b2e68dfc85
d9
└── 809bc1dafdec03f0d60f41f6c7f6cfc3228c80
da
```

```
├── 967ae1f60e59d2a223e37301f63050dca0cf6f
└── fe823560ecc5694151c37187f978b5cf3d5cf1
.
.
```

In my case, I only see one file: **3ab2b104e4addd03947ed3b1ca57b2e68dfc85**. You may see other files in there, and that's to be expected in a moderately busy repository.

You can't take a look at this object directly, though, as objects in Git are compressed. If you tried to look at it using `cat 3ab2b104e4addd03947ed3b1ca57b2e68dfc85` or similar, you'll probably see a pile of gibberish like so, along with a few chirps from your computer as it tries to read control characters from the binary object:

```
xu?Ko?0??51??Л
yB
    ??f?y?cBшo?{ɛ?|ьFL?:?@??_?0Td5?D2Br?D$??f?B??b?5W?HÁ?H*?&??
(fbd

dC!DV%?????D@?(???u0??8{?w????0?IULC1????@(<?s '
mO????????ze?S????>?K8                  89_vxm(#?jxOs?u?b?5m????
=w\l?
%?O??[V?t]?^??????G6.n?Mu?%
                              ??X?ʔX̧v??x?EX???:sys???G2?y??={X?ᑎe?
X?4u???????4o'G??^"qِ???$?Ccu?ml???vB_)?I?6?$?(?E9?z??nUmV?Em]?
p??3?`??????q?Ţqjw????VR?O? q?.r???e|lN?p??Gq?)?????#???85V?
W6?????
)|Wc*??8?1a?b?=?f*??pSvx3??;??3??^??O?S}??Z4?/?%J?
                                                    F?of??Oᔑ*??`
```

Viewing Git objects

Git provides a way to look at the contents of a compressed Git object: `git cat-file`. This decompresses the object and writes it out to your console in a human-readable form. You can simply pass it a short or long hash, and Git will write out the contents of that object in a human-readable form.

So take a look at the uncompressed form of the object file with the following command, substituting in the short or long hash from the commit that you want to look at:

```
git cat-file -p d83ab2b
```

The –p option here tells Git to figure out what type of object it's dealing with and to provide appropriately formatted output.

The commit object

In my case, Git tells me the details about my chosen commit object:

```
tree c0425d3b2aa2bfbbc0a08efda69ed00286dec6e4
parent 5415c13d2449f9719a8a8e84ee25105a1a587c5f
author cripsy8888 <chris@razeware.com> 1549849076 -0400
committer GitHub <noreply@github.com> 1549849076 -0400
gpgsig -----BEGIN PGP SIGNATURE-----

 wsBcBAABCAAQBQJcYNH0CRBK7hj4Ov3rIwAAdHIIABLgrn6UmK0fzh/
jqaIg7ax2

kie1Grd4EqLA+kuNT0jR+qTbc6x+OwlYt2PWZXOzfyOwY3UNKByHWhJDrhgzjLjB
 65CT7GGmMOKlGi7gis3W6jZetka+Lnauoeg9e/VnAu6q/
9JOv6ZyRN4j13wYpnK1

9wyooTbV2ipKMRFBs56DjL+6LkJcuIdD98rqluUzugGIvjFnGmIUCKF485l1bN3Q
 eZ+PsFGeqqIFHdWnXOyvBhzjVogoumR8K7WtQ8tGMXnAnwlBo0s+sikJa4tTmO/
o

feVt0ln+frS+j6zhnC1RHRPkucPDBV9DuVdrSiA4w1xmXCXmVZ26bCEHQkaf1Z0=
 =QrF9
 -----END PGP SIGNATURE-----


Ticked off last item added

No one would believe you could skew election results...
```

There's a wealth of information here, but what you're interested in is the **`tree`** hash.

The tree object

The tree object is a pointer to another object that holds the collection of files for this commit.

So execute `git cat-file` again to see what's inside *that* object, substituting your particular hash:

```
git cat-file -p c0425d3b2aa2bfbbc0a08efda69ed00286dec6e4
```

I get the following information about the tree object:

```
100644 blob 8b23445f4a55ae5f9e38055dec94b27ef2b14150    LICENSE
100644 blob f5c651739ff232f6226d686724f3c9618dd9f840
README.md
040000 tree d27f2eb006fff5b83fdc5d6639c7cfabdcf9fc37    articles
040000 tree 0b2d0890591a57393dc40e2155bff8901acafbb6    books
040000 tree 028f2d5e0a0f99902638039794149dfa0126bede    videos
```

Ah — that looks a *lot* like the working tree of your project, doesn't it? That's because that's precisely what this *is*: a compressed representation of your file structure inside the repository.

Now, again, this object is simply a pointer to other objects. But you can keep unwrapping objects as you go.

The blob object

For instance, you can see the state of the **LICENSE** file in this commit with `git cat-file`:

```
git cat-file -p 8b23445f4a55ae5f9e38055dec94b27ef2b14150
```

I see all that glorious legalese of the MIT license I added to my repository so many months ago:

```
MIT License

Copyright (c) 2019

Permission is hereby granted, free of charge, to any person
obtaining a copy
of this software and associated documentation files (the
"Software"), to deal
in the Software without restriction, including without
limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense,
and/or sell...
<snip>
```

You can dig further into the tree by following the references down. What's inside the **articles** directory in this commit? The following command will tell you that:

```
git cat-file -p d27f2eb006fff5b83fdc5d6639c7cfabdcf9fc37
```

I see the following files inside that directory:

```
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    .keep
100644 blob f8a69b62146eceef1b9078fed8788fbb6089f14f
clickbait_ideas.md
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391
ios_article_ideas.md
```

Looking inside **clickbait_ideas.md** with `git cat-file` again, I'll see the full contents of that file as I committed it:

```
# Clickbait Article Ideas

These articles shouldn't really have any content but need
irresistible titles.

- [ ] Top 10 iOS interview questions
- [ ] 8 hottest rumors about Swift 5 - EXPOSED
- [ ] Try these five weird Xcode tips to reduce app bloat
- [ ] Apple to skip iOS 13, eyes a piece of Android's pie
- [ ] 15 ways Android beats iOS into the ground and 7 ways it
doesn't
- [ ] I migrated my entire IT department back to Windows XP -
and then this happened
- [ ] The Apple announcement that should worry Swift developers
- [ ] iOS 13 to bring back skeuomorphism amidst falling iPhone
sales
- [x] Machine Learning to blame for skewed election results
```

You could keep digging further, but I'm sure you've seen enough to get an understanding of how Git stores commits, trees and the objects that represent the files in your project. It's turtles all the way down, man.

So you can see how easily Git can reconstruct a branch, based on a single commit hash:

1. You switch to a named branch, which is a label that references a commit hash.
2. Git finds that commit object by its hash, then it gets the tree hash from the commit object.
3. Git then recurses down the tree object, uncompressing file objects as it goes.
4. Your working directory now represents the state of that branch as it is stored in the repo.

That's enough mucking about under the hood of Git; navigate back up to the root directory of your project and let Git take care of its own business. You have more important things to attend to.

Key points

- Git uses the SHA-1 hash of content to create references to commits, trees and blobs.
- A commit object stores the metadata about a commit, such as the parent, the author, timestamps and references to the file tree of this commit.
- A tree object is a collection of references to either child trees or blob objects.
- Blob objects are compressed collections of files; usually, the set of files in a particular directory inside the tree.
- `git rev-parse`, among other things, will translate a short hash into a long hash.
- `git cat-file`, among other things, will show you the pertinent metadata about an object.

Where to go from here?

Git has quite an elegant and powerful design when you think about it. And the wonderful thing is that all of this is abstracted away from you at the command line, so you don't need to know *anything* about the mechanisms underneath if you're the type who thinks ignorance is bliss.

But for those of you who *do* want to know how things work, and for those people who find that development is messy and unpredictable, Git has a great tool for you: stashes.

Chapter 2: Merge Conflicts

By Chris Belanger

The reality of development is that it's a messy business; on the surface, it's simply a linear progression of logic, a smattering of frameworks, a bit of testing — and you're done. If you're a solo developer, then this may very well be your reality. But for the rest of us who work on code that's been touched by several, if not hundreds, if not thousands of other hands, it's inevitable that you'll eventually want to change the same bit of code that someone else has recently changed.

Imagine that your team's project contains the following bit of HTML:

```
<p>Head over to the following link to learn how to get started
with Git:</p>
<a href="http://guides.github.com/activities/hello-
world/">link</a>
```

You've been tasked with updating all of the text of the links to something more descriptive, while your teammate has been tasked with changing HTTP URLs in this particular project to HTTPS.

At 9:00 a.m., your teammate pushes the following change to the piece of code to the project repository, to update `http` to `https`:

```
<p>Head over to the following link to learn how to get started
with Git:</p>
<a href="https://guides.github.com/activities/hello-
world/">link</a>
```

At 9:01 a.m. (because you were a little farther back in the coffee lineup that morning), you attempt to push the following change to the repository:

```
<p>Head over to the following link to learn how to get started
with Git:</p>
<a href="http://guides.github.com/activities/hello-
world/">GitHub's Hello World project</a>
```

But, instead of Git committing your changes to the repository, you receive the following message instead:

```
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/
supersites/git-er-done.git'
hint: Updates were rejected because the remote contains work
that you do
hint: not have locally. This is usually caused by another
repository pushing
hint: to the same ref. You may want to first integrate the
remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help'
for details.
```

That's something you've probably seen before. The remote has your teammate's changes that you just haven't yet pulled down to your local system. "Easy fix," you think to yourself, so you execute `git pull` as suggested, and...

```
From https://github.com/supersites/git-er-done
```

```
   7588a5f..328aa94  master     -> origin/master
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the
result.
```

Well, that didn't go as planned. You were expecting Git to be smart to merge the contents of the remote, which contains the commits from your teammate, with your local changes. But, in this case, you and your teammate have changed the **same line**. And since Git, by design, doesn't know anything about the language you're working with, it doesn't know that your changes won't impact your teammate's changes — and vice-versa. So Git plays it safe and bails, and asks *you* to do the work to merge the two files manually.

Welcome to the wonderful world of **merge conflicts**.

What is a merge conflict?

As a human, it's fairly easy to see how two people modifying the same line of code in two separate branches could result in a conflict, and you could even argue that a halfway intelligent developer could easily work around that situation with a minimum of fuss. But Git can't reason about these things in a rational manner as you or I would. Instead, Git uses an algorithm to determine what bits of a file have changed and if any of those bits could represent a conflict.

For simple text files, Git uses an approach known as the **longest common subsequence** algorithm to perform merges and to detect merge conflicts. In its simplest form, Git find the longest set of lines in common between your changed file and the common ancestor. It then finds the longest set of lines in common between your teammate's changed file and the common ancestor.

Git aligns each pair of files along its longest common subsequence and then asks, for each pair of files, "What has changed between the common ancestor and this new file?" Git then takes those differences, looks again, and asks, "Now, of those changes in each pair of files, are there any sets of lines that have changed *differently* between each pair?" And if the answer is "Yes," then you have a merge conflict.

To see this in action, you'll start working through the sample project for this section of the book, and you'll merge in some of your team's branches in order to see that resolving merge conflicts isn't *quite* as scary or frustrating as it looks on the surface.

Handling your first merge conflict

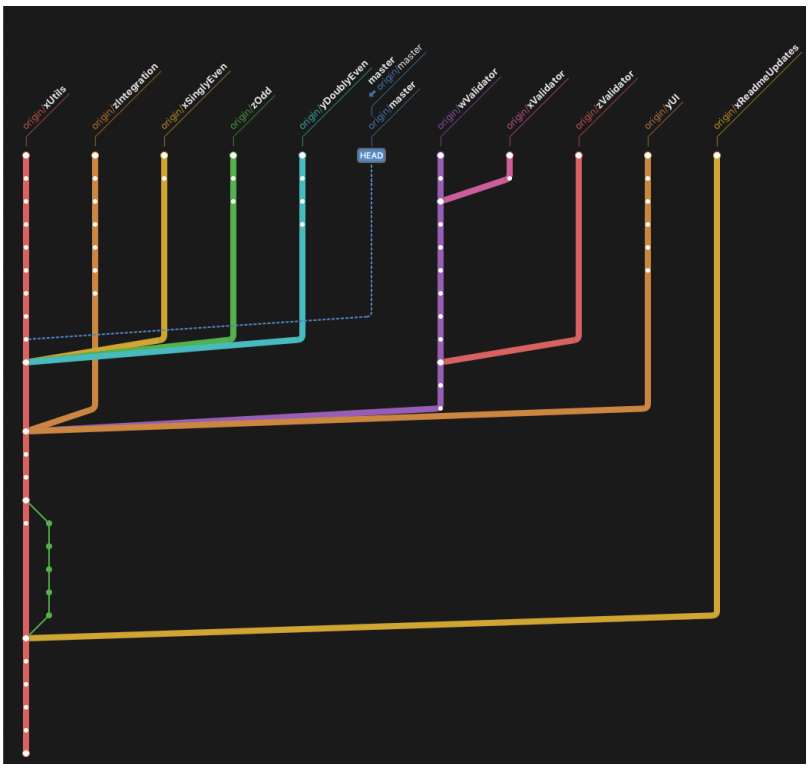
To get started, you'll need to clone the magicSquareApp repository that's used in this section of the book.

You can do this by way of the `git clone` command:

```
git clone https://github.com/raywenderlich/magicSquareJS.git
```

Once that's done, navigate into the directory into which you cloned it.

Now, here's the situation: Zach has been working on the front-end HTML of the magic square application to make it work with the back-end JavaScript. Zach isn't a designer, so Yasmin has offered to lend her design skills to the project UI and style the front end so that it looks presentable.



As the project lead, you're responsible for merging the various bits together and testing out the project. So, at this point, you'd like to verify that Zach's HTML works properly with Yasmin's UI. To do this, you'll have to merge Zach's work with Yasmin's work, and then test the project locally.

Merging from another branch

Zach has been doing his work in the **zIntegration** branch, while Yasmin has been working in the **yUI** branch. Your job is to merge Yasmin's branch with Zach's branch and resolve any conflicts.

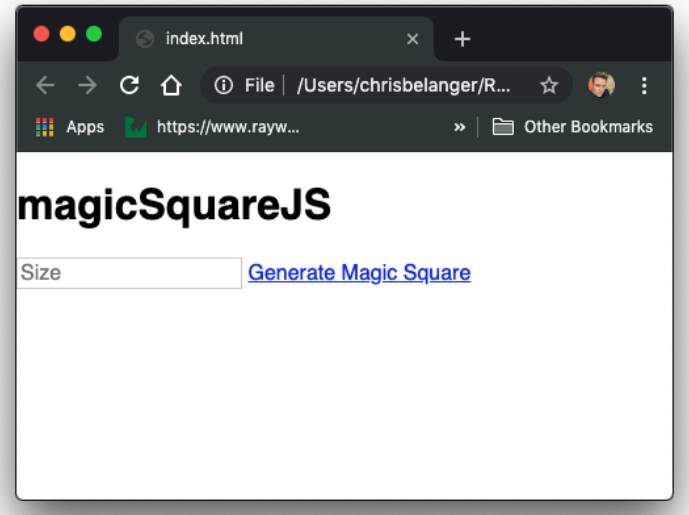
Pull down Yasmin's branch so you have a local, tracked copy of the branch:

```
git checkout yUI
```

First, switch to Zach's branch:

```
git checkout zIntegration
```

Open up **index.html** in a browser, to see what things look like in their current, pre-Yasminified state:



Well, it's clear that Zach is no designer. Good thing we have Yasmin.

Now you need to merge in Yasmin's UI branch:

```
git merge yUI
```

It appears that Zach and Yasmin's work wasn't *completely* decoupled, though, since Git indicates you have a merge conflict:

```
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the
result.
```

Helpfully, Git tells you above what file or files contain the merge conflicts. Open up **index.html** and find the following section:

```
  <body>
    <h1>magicSquareJS</h1>
<<<<<<< HEAD
    <section>
      <input type="text" placeholder="Size"
        id="magic-square-size" />
      <a href="#"
        id="magic-square-generate-button">
          Generate Magic Square</a>
      <pre id="magic-square-display">
=======
    <section class="box">
      <input type="text" class="flex-item" placeholder="Size"/>
      <a href="#"
        class="flex-item btn" >Generate Magic Square</a>
      <pre class="flex-item" >
>>>>>>> yUI
      </pre>
```

OK, you admit that your HTML is a *little* rusty, but you're pretty sure that `<<<<<<< HEAD` stuff isn't valid HTML. What on earth did Git do to your file?

Understanding Git conflict markers

What you're seeing here is Git's representation of the conflict in your working copy. Git compared Yasmin's file to the common ancestor, then compared Zach's file to the common ancestor and found this block of code that had changed differently in each case.

In this case, Git is telling you that the HEAD revision (i.e., the latest commit on Zach's branch) looks like the block between the `<<< HEAD` marker, and the `===` marker. The latest revision on Yasmin's branch is the block contained between the `===` line and the `>>> yUI` marker.

Git puts both revisions into the file in your working copy, since it expects you to do the work yourself to resolve this conflict. If you were intimately familiar with the code in question, you might know exactly how to combine Zach's and Yasmin's code to get the desired result. But you skipped a few too many project design meetings, didn't you?

No matter; you can ask Git to give you a few more clues as to what's happened here. Remember that a merge in Git is a three-way merge, but by default Git only shows you the two child revisions in a merge conflict; in this case, Yasmin and Zach's changes. It would be quite instructional to see the common ancestor for both of these child revisions, to figure out the intent behind each change.

Resolving merge conflicts

First, you need to return to the previous state of your working environment. Right now, you're mid-merge, and you only have two choices at this point: Either go forward and resolve the merge, or roll back and start over. Since you want to look at this merge conflict from a different angle, you'll roll back this merge and start over.

Reset your working environment with the following command:

```
git reset --hard HEAD
```

This reverts your working environment back to match HEAD, which, in this case, is the latest commit of your current branch, `zIntegration`.

A better way to view merge conflicts

Now, you can configure Git to show you the three-way merge data with the following command:

```
git config merge.conflictstyle diff3
```

> **Note**: If you ever wanted to change back to the default merge conflict tagging, simply execute `git config merge.conflictstyle merge` to get rid of the common ancestor tagging.

To see the difference in the merge conflict output, run the merge again:

```
git merge yUI
```

Git explains patiently that yes, there's still a conflict. In fact, this is a good time to see what Git's view of your working tree looks like, before you go in and fix everything up. Execute the `git status` command, and Git shows you its understanding of the current state of the merge:

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

You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Changes to be committed:

  new file:   css/main.css

Unmerged paths:
  (use "git add <file>..." to mark resolution)

  both modified:   index.html
```

Most of that output makes sense, but the last bit is rather odd: `both modified: index.html`. But there's only one **index.html**, isn't there? Why does Git think there is more than one?

A consolidated git status

Remember that Git doesn't always think about files, per se. In this case, Git is talking about both *branches* that are modified. To see this in a bit more detail, you can add the `-s` (`--short`) and `-b` (`--branch`) options to `git status` to get a consolidated view of the situation:

```
git status -sb
```

Git responds with the following:

```
## zIntegration...origin/zIntegration
A  css/main.css
UU index.html
```

The first two columns (showing `A` and `UU`) represent the "ours" versus "theirs" view of the code. The left column is your local branch, which currently is the mid-merge state of the original `zIntegration` branch mixed with the changes from the `yUI` branch. The right column is the remote branch. So this abbreviated `git status` command shows the following:

- You have one file added (`A`) on your local branch; this is **css/main.css** that Yasmin must have added in her work. But it's not in conflict with your work.
- On the other hand, you have not one, but *two* revisions of a file that are unmerged (`U`) in your branch. This is the original **index.html** from the `zIntegration` branch, and the **index.html** from your `yUI` branch.

These files are considered unmerged because Git has halted partway through a merge, and put the onus on you to fix things up. Once you've fixed them up, committing those changes will continue the merge.

Editing conflicts

Open up **index.html** and have a look at the conflicted block of code now, with the new `diff3` conflict style:

```
  <body>
    <h1>magicSquareJS</h1>
<<<<<<< HEAD
    <section>
      <input type="text" placeholder="Size" id="magic-square-
size" />
      <a href="#" id="magic-square-generate-button">Generate
Magic Square</a>
      <pre id="magic-square-display">
||||||| 69670e7
    <section>
      <input type="text" placeholder="Size"/>
      <a href="#">Generate Magic Square</a>
      <pre>
=======
    <section class="box">
      <input type="text" class="flex-item" placeholder="Size"/>
      <a href="#" class="flex-item btn" >Generate Magic Square</
```

```
a>
      <pre class="flex-item" >
>>>>>>>yUI
      </pre>
```

There's a new section in there: `||||||| 69670e7`. This shows you the hash of the common ancestor of both Yasmin and Zach's changes; that is, what the code looked like before each created their own branch. A visual comparison of HEAD (which is Zach's branch) against the common ancestry shows the following changes:

- Added `id="magic-square-size"` to the `input` tag
- Added `id="magic-square-generate-button"` to the `a` (anchor) tag
- Added `id="magic-square-display"` to the `pre` tag

A quick visual comparison of Yasmin's changes against the common ancestor shows the following changes:

- Added `class="box"` to the `section` tag
- Added `class="flex-item"` to the `input` tag
- Added `class="flex-item btn"` to the `a` tag
- Added `class="flex-item"` to the `pre` tag

It looks like it will be less work to migrate Zach's changes into Yasmin's code. So edit **index.html** by hand, moving Zach's new `id` attributes, from the first block of code in the conflicted section, into the third block in the conflicted section, which is Yasmin's code.

When you've moved those three `id` attributes into Yasmin's code, you can now delete the entire first two blocks from the conflicted section, from `<<< HEAD` all the way to `===`. Then, delete the `>>> yUI` line as well. When you're done, this section of code should look like the following:

```
  <body>
    <h1>magicSquareJS</h1>
    <section class="box">
      <input type="text" id="magic-square-size" class="flex-
item" placeholder="Size"/>
      <a href="#" id="magic-square-generate-button" class="flex-
item btn" >Generate Magic Square</a>
        <pre id="magic-square-display" class="flex-item" >
        </pre>
```

Save your work and return to the command line.

Completing the merge operation

You've finished resolving the conflict, so you can stage your changes with the following:

```
git add index.html
```

Execute the condensed version of `git status -sb` to see what Git thinks about your merge attempt:

```
## zIntegration...origin/zIntegration
A  css/main.css
M  index.html
```

There you are; one new file and one modified file. Git's noticed that you've resolved the outstanding conflicts, so all that's left to do to complete the merge is to commit your staged changes.

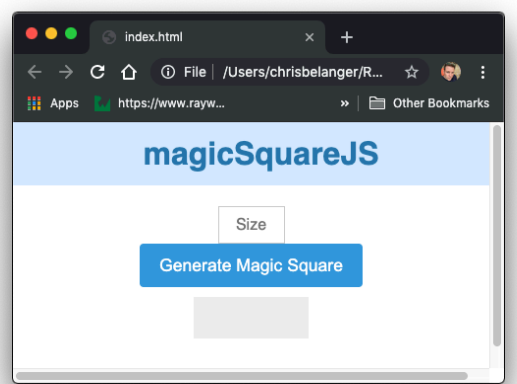
Commit those changes now, this time letting Git provide the merge message via Vim:

```
git commit
```

Type **:wq** inside of Vim to accept the preconfigured merge commit message, and Git dumps you back to the command line with a brief status, showing you that the merge succeeded:

```
[zIntegration af33aaa] Merge branch 'yUI' into zIntegration
```

Now, open **index.html** in a browser to see the changes:



That looks quite good. It's not fully functional at the moment, but you can see that Yasmin's styling changes are working. You're free to delete her branch and merge this work into `master`.

First, delete the `yUI` branch:

```
git branch -d yUI
```

Switch to the `master` branch:

```
git checkout master
```

Now, attempt a merge of the `zIntegration` branch:

```
git merge zIntegration
```

Git takes you straight into Vim, which means the merge had no conflicts. Type **:wq** to save this commit message and complete the merge. Git responds with the results of the merge:

```
Merge made by the 'recursive' strategy.
 css/main.css | 268 ++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++
 index.html   |  16 +++++++----
 js/main.js   |  85 ++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++
 3 files changed, 363 insertions(+), 6 deletions(-)
 create mode 100644 css/main.css
```

You're now able to delete the `zIntegration` branch, so do that now:

```
git branch -D zIntegration
```

> **Note**: The `-D` switch forces the local deletion of the branch regardless of its status. If you used the normal `-d` switch here, you'd see a warning about the branches changes not having been pushed to the remote.

Challenge

Challenge: Resolve another merge conflict

The challenge for this chapter is straightforward: resolve another merge conflict.

Xanthe has an old branch with some updates to the documentation; this work is in the `xReadmeUpdates` branch. You want to merge that work to `master`.

The steps are as follows:

- Check out the `xReadmeUpdates` branch and look at the README.md file to see Xanthe's version.
- Check out `master`, since this is the destination for your merge.
- Resolve any merge conflicts by hand.
- Stage your changes.
- Commit your changes.
- Delete the `xReadmeUpdates` branch.

If you get stuck, or want to check your solution, you can always find the answer to this challenge under the **challenge** folder for this chapter.

Key points

- Merge conflicts occur when you attempt to merge one set of changes with another, conflicting set of changes.
- Git uses a simple three-way algorithm to detect merge conflicts.
- If Git detects a conflict when merging, it halts the merge and asks for manual intervention to resolve the conflict.
- `git config merge.conflictstyle diff3` provides a three-way view of the conflict, with the common ancestor, "their" change, and "our" change.
- `git status -sb` gives a concise view of the state of your working tree.
- To complete a merge that's been paused due to a conflict, you need to manually fix the conflict, add your changes, and then commit those changes to your branch.

Where to go from here?

In practice, merge conflicts can get pretty messy. And it might seem that, with a bit of intelligence, Git could detect that adding HTML attributes to a tag is not really a conflict. And there are, in fact, lots of tools, such as IDEs and their plugins, that are language-aware and can resolve conflicts like this easily, without making you make all the edits by hand. But no tool can ever replace the insight that you have as a developer, nor can it replace your intimate understanding of your code and its intent. So even though you may come across tools that seem to do most of the work of resolving merge conflicts for you, at some point you'll find that there is no other way to resolve a merge conflict except by manual code surgery, so learning this skill now will serve you well in the future.

Up to now, your workflow has been constrained to the "happy path": you can create commits, switch between branches, and generally get along quite well without being interrupted. But real life isn't like that; you'll more often than not be partway through working on a feature or a fix, when you want to switch your local branch to take a look at something else. But because Git works at the atomic level of the commit, it doesn't like leaving things in an uncommitted state. So you need to stash the current state of your work somewhere, before you switch branches. And `git stash`, covered in the next chapter, does just that for you.

3 Chapter 3: Stashes

By Chris Belanger

After you've worked with Git for some time, you might start to feel a bit constrained. Sure, the staging, committing, branching and merging bits are all well and good, and these features undoubtedly support the nonlinear here-there-and-everywhere process of modern, iterative development. But the commit is still the atomic level of Git; there's nothing below it.

It can begin to feel like Git is forcing you into a workflow wherein you can't do anything outside of a commit, and that you have to limit yourself to building a working, documented commit each and every time you want to push your work to the repository. Talk about performance anxiety!

But Git recognizes that you can't always work to the level of a commit. Development is messy and unpredictable; you may go down a few parallel rabbit holes, chasing different possibilities, before finding the right solution for a bug. Or, quite commonly, you may be building out the next great feature for your app when, suddenly, you need to switch over to a bugfix branch and get a hotfix out the door in an hour. *Codus interruptus*, for sure.

So in situations like the above, what do you do with that unfinished and uncompilable code that isn't quite ready to be committed, but that you don't want to lose?

Well, if you're a paranoid coder like me, you may have developed a past habit of duplicating code directories that you want to keep for later, like so:

```
book-repo
├──git-book-before-edits
├──git-book-old
├──git-book-old-epub-test
├──git-book-older
└──git-book-version-with-jokes-removed
```

And if you bring that same thought process to working with Git, you'll often want to do a similar thing by creating "interim commits" in the branch you're working on, just so that you don't lose your work inside the confines of the Git repository:

```
| * b10bc04 It's the finallllllll countdooooooooown do-do-do-
doooooooooooo
| * 5f20836 Minor tweak
| * 36ff9ca fix lldb thing from errata forums
| * 0ce8469 Moved Mach-O section to DO_NOT_ADD_THIS… will keep
it around for next version
| * 79c465d removed TODO
| * cebc416 ok, time to wrap this crap up, Derek
| * 42d40dd wow that sucked
| * e7e8228 omg apple changing everything internally
| * 6d162de Partial Revert "wooooooooooooot done w/ code
signing"
| * 9676642 wooooooooooooot done w/ code signing
| * 986e288 ok, hopefully no more bugs on that script….
| * da5e21f Fix xattr bug
| * 52a2bc7 dsresign really looking good now
| * 6382f56 more tweaks
| * 66f0041 dsresign close to complete?
| * ba0bcf9 Added dsresign
| * 5ac42d4 meh
| * a5f08df Pre-delete a big section
| * 1d86a71 wow…. much words. Time to delete some content
.
.
.
```

(With apologies to the wonderful debugging guru Derek Selander at https://github.com/DerekSelander. Love ya, Derek.)

You can see how making interim commits to "save" your work can quickly get out of hand. It's clear that there *must* be a better way to quickly stash work in progress and retrieve it later, when you're done coding that duct-tape hotfix and are ready to return to refactoring the mess of code you left on a feature branch.

In fact, Git provides this feature out of the box, and it's called, unsurprisingly, `git stash`.

Introducing git stash

Let's return to your tightly knit team of Will, Xanthe, Yasmin and Zach. Things have been running smoothly for today, and you've been able to get some time to work on that all-important README file.

In the **magicSquareJS** repo, open **README.md** and add a bit of text to the bottom of the file:

```
## Contributing

To contribute to this project, simply
```

But before you can complete that thought, Xanthe pings you. "Can you take a quick look at the `xUtils` branch?" she asks. And, being the responsive, agile team member you are, you agree to have a look.

But you don't want to lose that work you've done. Granted, it's not a lot, but it does capture the state of your thoughts at that moment, and you don't really want to redo that work. So save your work to **README.md** and exit out of your text editor.

Back at your command prompt, switch to the `xUtils` branch:

```
git checkout xUtils
```

Git detects that you have changes in your working tree that haven't been captured anywhere, not at least as far as Git's concerned:

```
error: Your local changes to the following files would be
overwritten by checkout:
    README.md
Please commit your changes or stash them before you switch
branches.
Aborting
```

Again, Git nudges you in the right direction, here: Either stash your changes or commit them before you move on. And although your reflex might be to commit them — because they'd be *safe*, you know — stashing your changes is a much better option.

> **Note**: You will not get this error message if you did not complete the challenge from the last chapter.

How stashing works

In a manner that's quite similar to the way you created all of those "backup" directories of your code projects in the past, Git lets you take your current set of changes in your working tree and stash them in your local repository, to be retrieved later.

To see this in action, simply execute the following command:

```
git stash
```

Git will respond with a somewhat cryptic but reassuring message:

```
Saved working directory and index state WIP on master: 870aea1
Merge branch 'xReadmeUpdates' into master
```

It appears that Git has saved your current set of changes somewhere... but *where*? To see that, you'll dig into the internals of Git once again.

From the command prompt, navigate into the **.git/refs** directory:

```
cd .git/refs/
```

Inside that directory, you'll find a file named, simply, **stash**. Print the contents of that file out to the command line with the following command to inspect its contents:

```
cat stash
```

In my case, that file holds a single line, which is simply an object hash:

```
b6132364bdae71e8a5483e42584257aadbe49827
```

Just as you did before, you're going to use this object hash to look up the metadata and associated content of this object. Execute the following command to get the details about this object, substituting the actual value of your own hash for mine:

```
git cat-file -p b61323
```

Git tells me the following:

```
tree 1a79fa3410189b40dccb6b36f7eda8725c768627
parent 870aea10aa51d9103e6f6e37217b2cd077dd22bb
parent 68cde8e6fde76e21a7a93637f0bf5dbc3b36c242
author Chris Belanger <chris@razeware.com> 1556018137 -0300
committer Chris Belanger <chris@razeware.com> 1556018137 -0300

WIP on master: 870aea1 Merge branch 'xReadmeUpdates'
```

Well, that looks suspiciously like a commit. And, in fact, Git uses the same basic mechanism to stash your changes as it does to commit them. But instead of treating this as a commit, Git tracks this as a stash object.

I'll dive further into this stash, so feel free to try this out with your own metadata and hashes. I'll look up the `tree` hash of my stash with the following:

```
git cat-file -p 1a79fa
```

Git shows me the snapshot of my working tree at the moment I stashed it:

```
100644 blob 7b378be30685f019b5aa49dfbdd6a1c67001d73c    .tools-
version
100644 blob 28c0f4fd0553ffb10b0ead9cd9584a4d251b61c8
IGNORE_ME
100644 blob 9d47666971a2b201db4d89f0536d5766af389c7c    LICENSE
100644 blob 475ce3189a12efc860a75ab19d6e8f30533c723c
README.md
100644 blob feab599b6be0efb9d20ef20e749b3b8e70e8c69f    SECRETS
040000 tree d0a7cf32f8ee481267d545000ca99dc532ef0579    css
040000 tree 29a422c19251aeaeb907175e9b3219a9bed6c616    img
100644 blob 0ab31637631bfdf857b1e8ad0c2e2ae435db2352
index.html
040000 tree 3876f897b04b07c02dcbe321ad267b97d1db532f    js
```

And then I'll display the actual contents of the **README.md** file with the following:

```
git cat-file -p 475ce318
```

Git shows me the contents of that file, with my most recent change at the bottom (output truncated for brevity):

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

## Contact Info

For info on this project, please contact [Xanthe]
(mailto:xanthe@example.com).

## Contributing

To contribute to this project, simply
```

So, just as if I'd committed this file to the repo, I have a snapshot of my working tree at a particular point in time, but held as a stash, instead of as a commit.

That's enough playing around inside the Git internals; head back to the root of your project folder with the following command:

```
cd ../..
```

Now that you understand a little about how Git stores your stash, you can move on to retrieving what you've stashed.

Retrieving stashes

You've stashed your changes so that you can check out Xanthe's `xUtils` branch, so continue to do that now:

```
git checkout xUtils
```

Git doesn't complain this time, as it sees that you've created a stash of your current working tree and there are no unstashed or uncommitted changes.

For the purposes of this exercise, you'll assume that you've looked through Xanthe's changes and given her some advice on what she should do. With that task out of the way, you can now return to your changes that you'd like to complete on the `master` branch.

Switch back to `master` with the following command:

```
git checkout master
```

And take a look at the contents of **README.md** with the following:

```
cat README.md
```

Look at the end of the file, and you'll see that your changes aren't there. That makes sense, since Git switches you back to whatever the current state of `master` is on your local system. But how do you find your stashed changes and get them back into your working tree, since it's already been a long day and there's no *way* you're going to remember the hash of the stash you created earlier?

Listing stashes

Git lets you create more than one stash, and it keeps them in a stack structure so you can easily find the one you want to apply. But, first, you'll create another stash to illustrate this.

Imagine you don't think you need that **SECRETS** file lying around anymore, but you want to test this later to make sure you *really* don't need it. Delete that file from your working tree with the following:

```
rm SECRETS
```

And now create another stash, with the same command you used before:

```
git stash
```

Git gives you the same message as before; note that there's no mention from Git of how to identify your most recent stash.

```
Saved working directory and index state WIP on master: 870aea1
Merge branch 'xReadmeUpdates' into master
```

> **Note**: Since you're working with a stash, it seems that Git should let you use common stack operations, and, in fact, it does. `git stash` is a convenience alias for `git stash push`, for quickly creating a non-named stash on the stack. You also have access to common stack operators such as `pop`, `show` and `list`, as you'll see in the following sections.

To see the stack of stashes, use the `list` option on `git stash`:

```
git stash list
```

Git shows you all of the stashes it knows about:

```
stash@{0}: WIP on master: 870aea1 Merge branch 'xReadmeUpdates'
stash@{1}: WIP on master: 870aea1 Merge branch 'xReadmeUpdates'
```

Since this is a stack, the stash at index 0, denoted by the `stash@{0}` label, is the most recent, with the stash at index 1, being older as it's farther down the stack.

Now, the default stash message really isn't that descriptive; if you only work with one stash at a time, that might be sufficient. But you won't always remember what the most recent stash is, if they all have the same stash message. But just as you can provide a message when you create a commit, you can specify a message when you create a stash.

Adding messages to stashes

Make another change in your project working tree, and create a temporary file with the following stacked commands:

```
mkdir temp && touch temp/.keep && git add .
```

> **Note**: The && operator in Bash or Zsh lets you chain commands and execute them in order, but only if the preceding command succeeded (that is, had an exit code of `0`). So, in this case, you're trying to create a directory named **temp**; if that succeeds, you then create a **.keep** file in the **temp** directory so that Git recognizes this directory. If the file creation was successful, then you call `git add .` to stage this file so that Git can track it.

Now that you have a tracked addition to your working tree, you can create a stash with an appropriate message to help you keep track of what's what. Execute the following command:

```
git stash push -m "Created temp directory"
```

In this case, you've used the `push` operator, since `git stash` alone doesn't let you supply any arguments. Now, pull up the stash stack again with `git stash list` to see what the stack looks like now:

```
stash@{0}: On master: Created temp directory
stash@{1}: WIP on master: 870aea1 Merge branch 'xReadmeUpdates'
stash@{2}: WIP on master: 870aea1 Merge branch 'xReadmeUpdates'
```

That's a little more instructive, but your memory is short. If you can't recall what exactly you did in a particular stash, you can peek at the contents of that stack entry with the following command:

```
git stash show stash@{0}
```

Git tells you briefly what the changes are in this stashed snapshot:

```
 temp/.keep | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
```

In fact, you can use most of the options from `git log` on `git stash show`, since Git is simply using its own log to show the changes contained in your snapshot. You can check out the diff, or patch, of your very first stash using the `-p` option as you would with `git log`:

```
git stash show -p stash@{2}
```

Git then shows you the entire patch of your stash. Here's mine:

```
diff --git a/README.md b/README.md
index b7338e2..475ce31 100644
--- a/README.md
+++ b/README.md
@@ -18,3 +18,7 @@ This project is maintained by teamWYXZ:
 ## Contact Info

 For info on this project, please contact [Xanthe]
(mailto:xanthe@example.com).
+
+## Contributing
+
```

```
+To contribute to this project, simply
```

Here, you can see the final four lines added to the end of **README.md**. Speaking of which, you should probably finish those updates to the README file before you lose your creative inspiration.

Popping stashes

Now, you want to get back to the state represented by the stash at the bottom of your stack: `stash@{2}`. If you remember your stack operations from your data structures and algorithms courses, you know that you'd generally `pop` something off the top of this list. And in fact, you can do this with Git, using `git stash pop` to remove the top stash from the stack and apply that patch to your working environment.

In this case, however, the stash you want isn't at the top; rather, it's at the bottom of the stack. And you've decided that you don't really want to keep those other two stashes around, either.

Git lets you cherry-pick a particular stash out of the stack and apply it to your working tree. So to get back to the state where you created that first stash, that is, with your modification to the README file, first reset the state of your working tree, to get rid of any local changes:

```
git reset --hard HEAD
```

Now, you can use `git stash apply` to apply a particular stash to your working tree. In this case, you want the stash at the bottom of the stack — `stash@{2}`:

```
git stash apply stash@{2}
```

Git then reverts the state of your working tree to the snapshot captured in the stash. In this case, your stash doesn't have the **temp** directory, and it still has the **SECRETS** file, as you deleted that file *after* you created the stash. Pulling up a simple directory listing with `ls -la` will show you this:

```
drwxr-xr-x  12 chrisbelanger  staff    384 23 Apr 10:15 .
drwxr-xr-x   8 chrisbelanger  staff    256 16 Apr 05:42 ..
drwxr-xr-x  15 chrisbelanger  staff    480 23 Apr 10:16 .git
-rw-r--r--   1 chrisbelanger  staff      5 16 Apr 16:18 .tools-
version
-rw-r--r--   1 chrisbelanger  staff     43 16 Apr 16:18
IGNORE_ME
-rw-r--r--   1 chrisbelanger  staff  11343 16 Apr 05:42 LICENSE
```

```
-rw-r--r--   1 chrisbelanger  staff    538 23 Apr 10:15
README.md
-rw-r--r--   1 chrisbelanger  staff     65 23 Apr 08:47 SECRETS
drwxr-xr-x   5 chrisbelanger  staff    160 23 Apr 08:39 css
drwxr-xr-x   3 chrisbelanger  staff     96 16 Apr 05:42 img
-rw-r--r--   1 chrisbelanger  staff   1259 23 Apr 08:39
index.html
drwxr-xr-x   6 chrisbelanger  staff    192 23 Apr 08:39 js
```

Now, execute `git status` to see how Git interprets the situation, and it notes that you have one change not staged for commit:

```
On branch master
Your branch is ahead of 'origin/master' by 17 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 checkout -- <file>..." to discard changes in working
directory)

    modified:   README.md

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

If you think about it, this makes sense; you didn't stage that change, so Git's snapshot of the working tree in your stash also represents Git's *tracking* status of files, whether they were staged or unstaged. And when you apply a stash, you overwrite not just the working tree, but also the staging area of your repository. Remember: You staged your addition of the **temp/.keep** file *after* you created the stash you just applied.

Applying stashes

Applying a stash doesn't remove it from the stack; you can see this if you execute `git stash list`:

```
stash@{0}: On master: Created temp directory
stash@{1}: WIP on master: 870aea1 Merge branch 'xReadmeUpdates'
stash@{2}: WIP on master: 870aea1 Merge branch 'xReadmeUpdates'
```

`stash@{2}` is still sitting at the bottom of the stack; Git simply applied the patch resulting from this change and left that stash on the stack. In just a bit, you'll see how to remove elements from this stack.

Open up **README.md** in an editor and finish off that excellent line of documentation that you spent all night dreaming up:

```
## Contributing

To contribute to this project, simply create a pull request on
this repository and we'll review it when we can.
```

Save your changes, and exit the editor. Now, you can stage and commit your manifesto, again using the stacking option of && in Bash to do it all on one line:

```
git add . && git commit -m "Updates readme"
```

#lazygit for the win, kids! Oh, but wait. You really did want to add that **temp/.keep** file after all. Fortunately, that change was stashed at the top of the stash stack, so you can use the pop operator to simultaneously apply the patch of that stash and remove it from the top of the stack:

```
git stash pop
```

Git gives you a nice status message, combining the output you might expect from `git status` along with a little note at the end telling you which stash was popped from the stack:

```
On branch master
Your branch is ahead of 'origin/master' by 18 commits.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   temp/.keep

Dropped refs/stash@{0}
(fca102bbfc2e4dc51264ae46211f95164ac2c933)
```

You've applied that patch on top of your working tree, and because Git stashed the tracking information as well as the state of the files in your working tree, those changes have already been staged for commit.

Clearing your stash stack

Now, you might think you're done here, but you still have a few stashes hanging around that you don't really need. Use `git stash list` to see what's still hanging around in the stash stack:

```
stash@{0}: WIP on master: 870aea1 Merge branch 'xReadmeUpdates'
stash@{1}: WIP on master: 870aea1 Merge branch 'xReadmeUpdates'
```

You can see that Git popped the top of the stack (the `On master: Created temp directory` entry), so it's gone. But you can easily get rid of all entries with the `clear` option.

In your case, you don't want any of the stash entries. So execute the following command to clear the entire stash stack:

```
git stash clear
```

Execute `git stash list` now, and you'll see that there are no more stashes hanging around.

Merge conflicts with stashes

Since applying stashes looks a *lot* like merging commits from Git's perspective, you may be wondering if you can also get conflicts when merging a stash. Well, maybe you weren't wondering this, but I'll bet you are now!

The answer is yes — you can certainly experience merge conflicts when working with stashes. You'll set up a scenario now that will show you how to resolve a merge conflict with a stash.

Recall that a merge conflict occurs when Git detects a change to a common subsequence inside a file. In your case, the best way to illustrate this is to alter the same line in the README file between a branch and a stash.

You're currently on `master`, so create a small edit to **README.md** to keep this author happy with proper use of English. Change the shorthand term "info" to "information" as shown below:

```
For information on this project, please contact [Xanthe]
(mailto:xanthe@example.com).
```

Save your work, exit the editor, and add this change as a stash using the following command:

```
git stash
```

Now, switch to the `xUtils` branch, where you'll create a conflicting change:

```
git checkout xUtils
```

Open **README.md** in that branch and add the following line, immediately after the `# Maintainers` section:

```
## Contact Info

For info on this project, please contact [Xanthe]
(mailto:xanthe@example.com) or [Will](mailto:will@example.com).
```

Here, you've added Will as a contributor, and you'll notice that this line still has the shorthand "info" in use.

Save your work and exit the editor. Now, stage and commit those changes — remember, you can't merge into a "dirty" or uncommitted branch:

```
git add . && git commit -m "Added Will as a contributor"
```

Now attempt to apply the stash with the following command:

```
git stash pop
```

Git responds with a message noting the conflict:

```
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
The stash entry is kept in case you need it again.
```

Open **README.md** in an editor, and edit the conflict as you've done before so that the final section looks like the following:

```
## Contact Info

For information on this project, please contact [Xanthe]
(mailto:xanthe@example.com) or [Will](mailto:will@example.com).
```

Save your changes and exit the editor. Now you can commit those changes to the `xUtils` branch:

```
git add . && git commit -m "Corrected language usage"
```

Challenge

Challenge: Clean up the remaining stash

Is there anything left to do? Execute `git stash list` and you'll see there's still a stash there. But you popped that stash, didn't you? Of course you did.

Your challenge is twofold, and may require a little dive into the Git documentation:

1. Explain why there's still a stash in the stack, even though you executed `git stash pop`.
2. Remove the remaining entry from the stash without using `git stash pop` or `git stash clear`. There's one more way to remove entries from the stack with `git stash` — what is it?

If you get stuck, or want to check your solution, you can always find the answer to this challenge under the **challenge** folder for this chapter.

Key points

- Stashing lets you store your in-progress work and retrieve it later, without committing it to a branch.
- `git stash` takes a snapshot of the current state of your local changes and saves it as a stash.
- Git stores stashes as a stack; all successive stashes are pushed on the top of the stack.
- Git stashes work much like commits do, and it is captured inside the .git subdirectory.
- `git stash` is a convenience alias for `git stash push`.
- `git stash list` shows you the stack of stashes you've made.

- `git stash show <stashname>` reveals a brief summary of the changes inside a particular stash.
- `git stash show –p <stashname>` shows the patch, or diff, of the changes in a particular stash.
- `git stash apply <stashname>` merges the patch of a stash to your working environment.
- `git stash pop` pops the top stash off of the stack and merges it with your working environment.
- Merging a stash can definitely result in conflicts, if there are committed changes that touch the same piece of work. Resolve these conflicts in the same way as you would when merging two branches.

Where to go from here?

Stashes are an excellent Git mechanism that help make your life as a developer just a little bit easier. And what's nice is that, again, Git follows common workflows that come naturally to most developers, such as stashing not-quite-done-yet changes off to the side (which is an improvement over those terrible copies of directories I used to make).

The next two chapters deal with one of the more useful, yet widely misunderstood actions in Git: rebasing. There are mixed opinions out there as to whether merging or rebasing is a better strategy in Git. To help you make sense of the arguments on both sides, you'll look at rebasing in depth so you can form your own opinions about what makes the most sense for your team and your workflow.

Chapter 4: Demystifying Rebasing

By 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 off of 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."

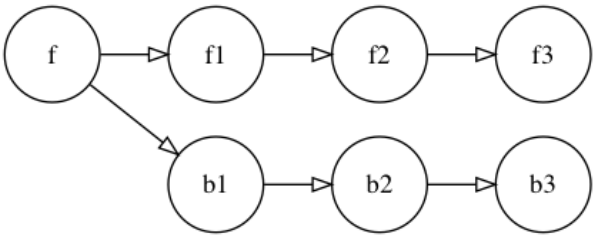
While that can often be useful, if the development workflow of your team results in fast, furious and short-lived branches, you might benefit from limiting merge commits and rebasing limited-scope changes instead. Rebasing gives you the choice to have a more linear commit history that isn't cluttered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

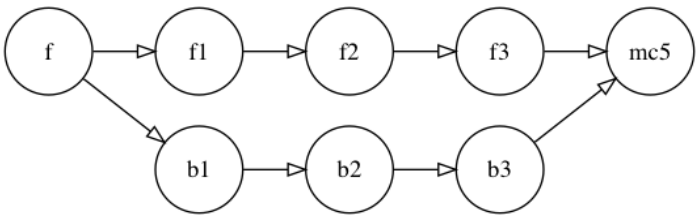
Rebasing is essentially just replaying a commit or a series of commits from history on top of a different commit in the repository. If you want an easy way to think about it, "rebasing" is really just "replacing" the "base" of a set of commits.

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**.



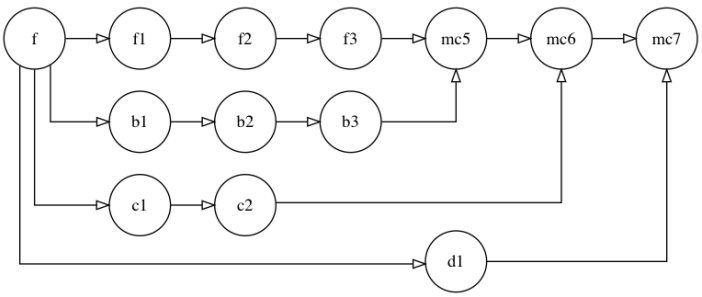
A simple branch (b) off of feature (f).

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.

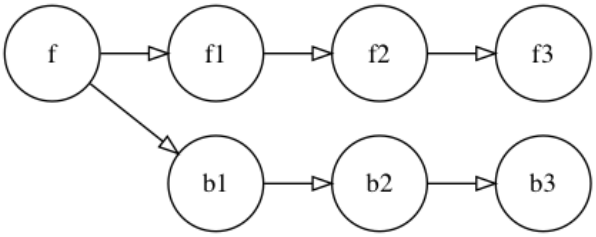


A more complex set of multiple branches and merge commits, with merge commits mc5, mc6 and mc7.

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**:

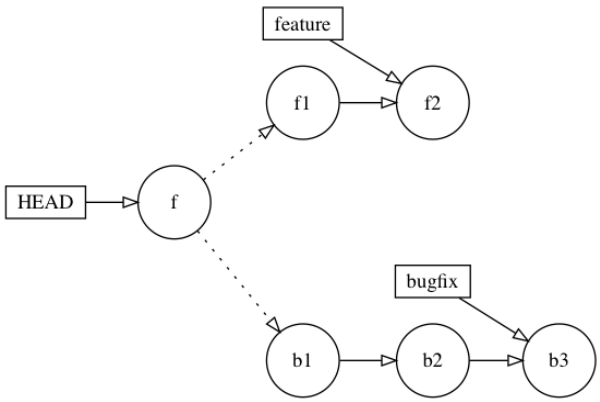


A simple branch (b) off of feature (f).

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.

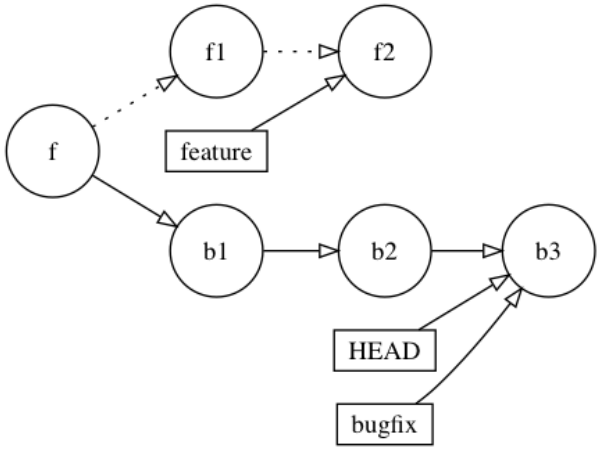
Let's presume you wanted it to appear as though the work performed on **feature** followed the work you did on **bugfix**. In Git parlance, you'd be rebasing **feature** on top of **bugfix**.

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**:



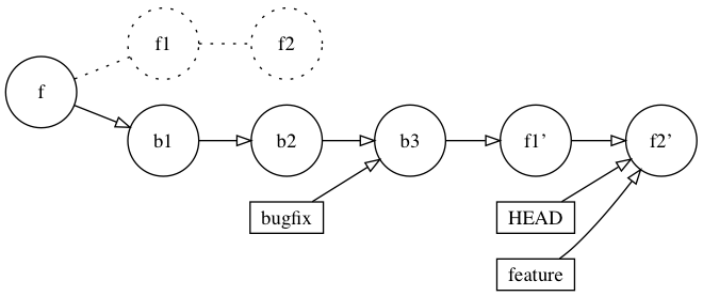
Rewinding HEAD to the common ancestor of the feature and bugfix branches

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` and `bugfix` labels 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" commits, as they're known) and clean them up in a regular garbage collection process. Although the loose 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 with your team to make sure that 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, find the starting repository for this chapter in the **starter** folder and unzip it to a working location.

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, check that you're on the correct branch for your repo:

```
git branch
```

You should be on the **wValidator** branch.

Create a new branch named **cValidator** from **wValidator**:

```
git checkout -b cValidator
```

Next, open up **README.md** and add your name 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.

Add your changes:

```
git add .
```

Commit your changes with an appropriate message:

```
git commit -m "Added 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 your initial 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, exit the editor and stage your changes:

```
git add .
```

Now create a commit with that change, using an appropriate message:

```
git commit -m "Updated team acronym"
```

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

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

The top three lines show you what's what:

```
* c628929 (HEAD -> wValidator) Updated team acronym
| * 2eb17a2 (cValidator) Added new maintainer to README.md
|/
* 3574ab3 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 (you're there now), and tell Git to 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 `78c60c3 – Updated team acronym`.

Take a look at the history graph to see the end result by executing the following:

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

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

```
* 17771e6 (HEAD -> wValidator) Updated team acronym
* 2eb17a2 (cValidator) Added new maintainer to README.md
* 3574ab3 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 `3f7969b`) on top of the common ancestor.
- Git left the branch label `cValidator` attached to `3f7969b`.
- 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`: `78c60c3`
- New commit for `Updated team acronym`: `f76b62c`

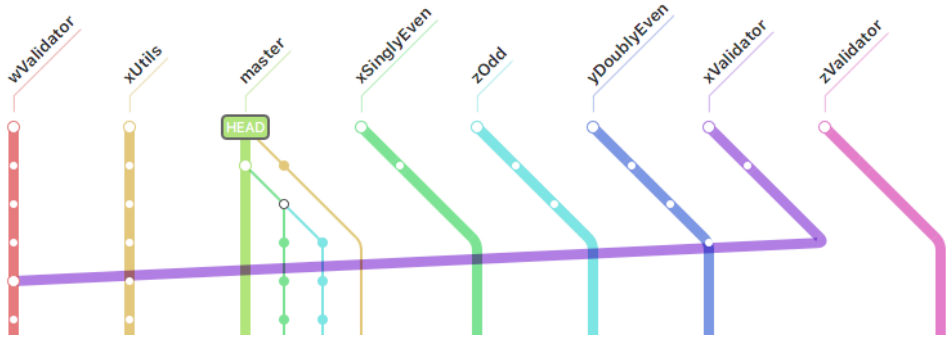
They're different because what you have at the tip of **wValidator** is *a brand-new commit* — not just the old commit tacked onto the end of the branch.

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 has also been busy refactoring on the xValidator branch.

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



The partial GitUp view of the repository, including the branches wValidator and xValidator.

Xanthe has branched off of Will's original branch to work on some refactoring, and it's now time to bring everything back into the wValidator branch. Because branching is cheap and easy in Git, these types of scenarios where developers branch off of existing branches is fairly common. Again, there's nothing saying that you always have to branch off of master or main — you can support any branching scheme you like, as long as you can keep track of things!

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.

First, check out the commits made since the common ancestor of wValidator and xValidator:

```
git log --oneline bf3753e~..
```

That last bit is new. What `bf3753e~..` means is: "limit `git log` to just this particular commit (inclusive) up to `HEAD`." Not providing the end commit hash indicates `HEAD`.

You'll see the following:

```
f76b62c (HEAD -> wValidator) Updated team acronym
3f7969b Added new maintainer to README.md
3574ab3 Whoops — didn't need to call that one twice
43d6f24 check05: Finally, we can return true
bf3753e check04: Checking diagonal sums
```

Those are the most recent commits on wValidator. Now, you know that xValidator branched from wValidator, so is it possible to view just what's changed on xValidator?

Absolutely. Execute the following to see what's happened since you branched xValidator from wValidator:

```
git log --oneline bf3753e~..xValidator
```

You'll see the following:

```
8ef01ac (xValidator) Refactoring the main check function
5fea71e Removing TODO
bf3753e check04: Checking diagonal sums
```

Your goal is to rebase the changes from wValidator on top of xValidator.

To start, ensure you've checked out the branch you want to merge your changes into with the following command:

```
git checkout wValidator
```

Git tells you you're already on that branch — no worries. It always pays to be sure.

Now, begin the rebase operation with the `git rebase` command, where you indicate which branch you want to rebase on top of your current branch:

```
git rebase xValidator
```

Resolving errors

`git rebase` provides quite a lot of verbose output, but if you look carefully through the output of your command, you'll see that there's a conflict you have to resolve in **js/magic_square/validator.js**.

Open up **js/magic_square/validator.js** and you'll see the conflict that you need to resolve. In this case, you want to keep the bits marked as `<<< HEAD`, since these are Xanthe's refactored changes that you want to keep.

> **Note**: You might be confused here. Why are you keeping the `HEAD` changes, if `HEAD` is the tip of the branch you're on — in this case, **wValidator**?
>
> In a rebase situation, `HEAD` refers to the tip of the branch you're rebasing on top of. As Git replays each commit onto this branch, `HEAD` moves along with each replayed commit.

Resolve the commit manually, removing the bits from the common ancestors and the original bits from Will's code. Save your work when you're done.

> **Note**: Did you notice the final separator line of the conflict?
>
> `>>>>>>> 43d6f24... check05: Finally, we can return true`
>
> Because rebasing works by replaying the commits of the other branch one by one on the current branch, Git helpfully tells you in which commit the conflict occurred. For complex merge conflicts, this little bit of extra information can be quite useful.

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 consider the situation from Git's perspective.

What you did above was to keep everything from the commit you are rebasing from xValidator. But Git is effectively expecting that there should be *some* change in the commit you're rebasing from xValidator, as this is the most likely case you'll encounter when rebasing work.

Imagine if Git just assumed that taking the commit verbatim was a completely normal situation; if you weren't paying attention to your commit resolution, you'd likely hit a point where you'd unwittingly clobber some work on the branch on which you're rebasing on top of. And then you'd spend countless hours, late at night, with too much coffee, trying to figure out where your rebase went so horribly wrong. In this situation, Git's error message would actually help you out.

But in this case, since you *are* taking the commit verbatim with no changes, you can simply execute the following command to carry on:

```
git rebase --skip
```

> **Note**: This may or may not be a great time to tell you that you didn't actually *need* to perform that conflict resolution in **validator.js**, since you took that commit as-is from xValidator. You could've just executed `git rebase --skip` straight away to tell Git that you had no interest in resolving the commit and to rebase the commit unchanged.

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

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

And you're done 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 since that common ancestor:

```
git log --oneline bf3753e~..
```

You'll see that the two original commits Will made at the end of wValidator are gone (the commits with short hash `3574ab3` and `43d6f24`), and Xanthe's commits are now neatly tucked in between the common ancestor and your updates to README.md, the xValidator branch label points to what was the tip of xValidator, and the wValidator branch label points to the tip as expected:

```
57f62b0 (HEAD -> wValidator) Updated team acronym
b14948d Added new maintainer to README.md
```

```
8ef01ac (xValidator) Refactoring the main check function
5fea71e Removing TODO
bf3753e check04: Checking diagonal sums
```

Stop just for a moment and consider what you've done here. Where did those commits from Will go? If you'd just done a simple merge, as you're used to doing, you would have still seen them in the history of the repo.

Even more confusingly, you can still find these commits in the logs. Execute the following command to see the three logged commits, starting at the "Whoops — didn't need to call that one twice" commit of `3574ab3`:

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

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

```
3574ab3 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!

Just for comparison purposes, check out what this entire scenario would have looked like from a merge perspective, as opposed to a rebase perspective:

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

You can see that the merge commit would result in the branch actions remaining in the repository history. Instead, the rebase action streamlined the commit history and gathered those changes as a cohesive linear operation. This is, arguably, clearer to the casual observer of your repository's history.

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

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:

```
| * 136dc26 (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 the zValidator branch. Again, the shared context here and the limited scope of the changes mean you don't need a merge commit.

Once you've rebased wValidator on top of zValidator, delete both the zValidator and xValidator branches, as you're done with them. Git might complain when you try to delete the branches. Explain why this is, and then figure out how to force Git to do it anyway.

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 <rebase-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`.

Chapter 5: Rebasing to Rewrite History

By 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, extract the compressed repository from the **starter** directory to a convenient location on your machine then navigate into that directory from the command line.

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 --all --decorate --oneline --graph
```

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

```
* 45f5b4f (HEAD -> wValidator) Updated team acronym
* 15233a5 Added new maintainer to README.md
* 783031e Refactoring the main check function
* 6396aa8 Removing TODO
* 8e39599 check04: Checking diagonal sums
* 199e71d util06: Adding a function to check diagonals
* a28b9e3 check03: Checking row and column sums
* bdc8bc7 util05: Fixing comment indentation
* a4d6221 util04: Adding a function to check column sums
* 59fd06e util03: Adding function to check row sums
* 5f53302 check02: Checking the array contains the correct
values
* 136dc26 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
```

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 `util*` commits and the `check*` commits together and, finally, to 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 `45f5b4f` and `15233a5`.
- Create a new commit message for this squashed commit.
- Rebase the resulting new commit on top of `783031e`.

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, `783031e`.

So, execute the following to start your first Git interactive rebase:

```
git rebase -i 783031e
```

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

```
pick 15233a5 Added new maintainer to README.md
pick 45f5b4f Updated team acronym

# Rebase 783031e..45f5b4f onto 783031e (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 <commit> = like "squash", but discard this commit's
log message
# 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.
#
# Note that empty commits are commented out
```

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, `15233a5` and `45f5b4f`, 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 `45f5b4f`, the last commit, into `15233a5`.

> **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. Cursor to the start of the `45f5b4f` 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 15233a5 Added new maintainer to README.md
squash 45f5b4f 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 new maintainer to README.md

# This is the commit message #2:

Updated team acronym

# Please enter the commit message for your changes. Lines
starting
# with '#' will be ignored, and an empty message aborts the
commit.
#
# Date:      Sun Jun 9 07:28:08 2019 -0300
#
# interactive rebase in progress; onto 783031e
# Last commands done (2 commands done):
#    pick 15233a5 Added new maintainer to README.md
#    squash 45f5b4f Updated team acronym
# No commands remaining.
# You are currently rebasing branch 'wValidator' on '783031e'.
#
# 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:

1. Type **gg** to ensure you're on the first line of the file.
2. 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:

```
Updates to README.md
```

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:

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

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

```
* 2492536 (HEAD -> wValidator) Updates to README.md
* 783031e 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 `2492536` 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 patches 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 in your repository:

```
git log --oneline
```

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

```
2492536 (HEAD -> wValidator) Updates to README.md
783031e Refactoring the main check function
6396aa8 Removing TODO
8e39599 check04: Checking diagonal sums
199e71d util06: Adding a function to check diagonals
a28b9e3 check03: Checking row and column sums
bdc8bc7 util05: Fixing comment indentation
a4d6221 util04: Adding a function to check column sums
59fd06e util03: Adding function to check row sums
5f53302 check02: Checking the array contains the correct values
136dc26 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 `master`, 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 5f53302 check02: Checking the array contains the correct
values
pick 59fd06e util03: Adding function to check row sums
pick a4d6221 util04: Adding a function to check column sums
```

```
pick bdc8bc7 util05: Fixing comment indentation
pick a28b9e3 check03: Checking row and column sums
pick 199e71d util06: Adding a function to check diagonals
pick 8e39599 check04: Checking diagonal sums
pick 6396aa8 Removing TODO
pick 783031e Refactoring the main check function
pick 2492536 Updates to README.md

# Rebase 69670e7..2492536 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:

1. Move the `util01` line to just above the `util02` line.
2. Leave the `Refactoring the range checking function` after `util02`.
3. 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 59fd06e util03: Adding function to check row sums
pick a4d6221 util04: Adding a function to check column sums
pick bdc8bc7 util05: Fixing comment indentation
pick 199e71d util06: Adding a function to check diagonals
pick 5f53302 check02: Checking the array contains the correct
values
pick a28b9e3 check03: Checking row and column sums
pick 8e39599 check04: Checking diagonal sums
pick 6396aa8 Removing TODO
pick 783031e Refactoring the main check function
pick 2492536 Updates to README.md
```

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` and you'll see that Git has neatly reordered your commits, and added new hashes as well:

```
35aab2b (HEAD -> wValidator) Updates to README.md
3899829 Refactoring the main check function
c8d5335 Removing TODO
5d16107 check04: Checking diagonal sums
5c9e64d check03: Checking row and column sums
4018013 check02: Checking the array contains the correct values
f7a31a0 util06: Adding a function to check diagonals
851663d util05: Fixing comment indentation
6c857e4 util04: Adding a function to check column sums
5ad299c util03: Adding function to check row sums
2575920 Refactoring the range checking function
96fb378 util02: Adding function to check the range of values
55d4ded util01: Adding the checkSqaure function
ded7caa 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 `55d4ded`, 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 ded7caa
```

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

```
pick 55d4ded 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 55d4ded 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` and scrolling down to find your new, rebased commit:

```
4f4e308 util01: Adding the checkSquare function
```

It's 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 ded7caa check01: checking that the 2D array is square
pick 4f4e308 util01: Adding the checkSquare function
squash 421c298 util02: Adding function to check the range of
values
```

```
squash 96dc840 Refactoring the range checking function
squash 19e90e9 util03: Adding function to check row sums
squash c9d8aa3 util04: Adding a function to check column sums
squash 30f164a util05: Fixing comment indentation
squash 0bda95b util06: Adding a function to check diagonals
pick d34c59b check02: Checking the array contains the correct
values
pick d235bf9 check03: Checking row and column sums
pick 00212f3 check04: Checking diagonal sums
pick ca6f8df Removing TODO
pick a4a05c0 Refactoring the main check function
pick a351e8a Updates to README.md
```

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:

```
858e215 (HEAD -> wValidator) Updates to README.md
d42dd03 Refactoring the main check function
499c6ac Removing TODO
7530a8f check04: Checking diagonal sums
c98bb17 check03: Checking row and column sums
eec2df9 check02: Checking the array contains the correct values
2207949 Creating utility functions for magic square validation
ded7caa 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!

Challenges

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:

1. Figure out what your base ancestor is for the rebase.
2. Start an interactive rebase operation.
3. Reorder the `check0x` commits.
4. Change the `pick` rebase script command for `squash` on all commits from the `check02` commit, down to and including the `Refactoring the main check function` commit.
5. Save your work in Vim and exit.
6. Create a commit message in Vim for the squash operation.
7. Take a look at your Git log to see the changes you've made.

Challenge 2: Rebase your changes onto master

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

1. Ensure you're on the wValidator branch.
2. Execute `git rebase` with `master` as your rebase target.
3. Crud — a conflict. Open README.md and resolve the conflict to preserve your changes, and move the changes to the `## Contact` section.
4. Save your work.
5. Stage those changes with `git add README.md`.
6. Continue the rebase with `git rebase --continue`.
7. Check the log to see where `master` points and where `wValidator` points.
8. Check out the `master` branch.
9. Execute `git merge` for `wValidator`. What's special about this merge that lets you avoid a merge commit?
10. 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!

Chapter 6: Gitignore After the Fact

By 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, ensure you're on `master` with `git checkout master`.

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
nothing to commit, working tree clean
```

So far so good. It looks like everything is working as planned.

Now switch back to `master` with the following command:

```
git checkout master
```

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 `master` to see what's going on. There *is* a **.gitignore** file in `master`, right?

Pull up a full directory listing with `ls -la` and you'll see that, in fact, there *is* no **.gitignore** on the `master` branch:

```
.
├── .DS_Store
├── .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 `master` 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 master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)

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

    modified:   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 master
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 master
Changes to be committed:
  (use "git reset HEAD <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
Author: Yasmin <yasmin@example.com>
Date:   Mon Jul 3 17:34:22 2017 +0700
an
    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 caret ^ 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 5d026f0 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 5d026f0 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 --all --decorate --oneline --graph
```

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 `master`. 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 stacked 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 stacked 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
```

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-history` 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 f28af7aad4f77da8deb28f1e0eb93b85ee755b43 (20/38) (1
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 --decorate`, 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 --decorate --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

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.

Chapter 7: The Many Faces of Undo

By Chris Belanger

One of the best aspects of Git is the fact that it remembers *everything*. You can always go back through the history of your commits with `git log`, see the history of your team's activities and cherry-pick commits from other places.

But one of the most frustrating aspects of Git is also that it remembers *everything*. At some point, you'll inevitably create a commit that you didn't want or that contains something you didn't intend to include.

While you can't rewrite shared history, you *can* get your repository back into working order without a lot of hassle.

In this chapter, you'll learn how to use the `reset`, `reflog` and `revert` commands to undo mistakes in your repository. While doing so, you'll find a new appreciation for Git's infallible memory.

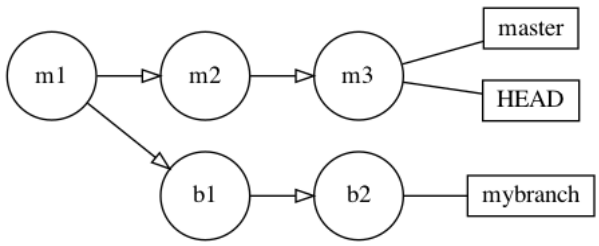
Working with git reset

Developers quickly become familiar with the `git reset` command, usually out of frustration. Most people see `git reset` as a "scorched earth" approach to fix a repository that's messed up beyond repair. But when you delve deeper into how the command works, you'll find that `reset` can be useful for more than a last-ditch effort to get things working again.

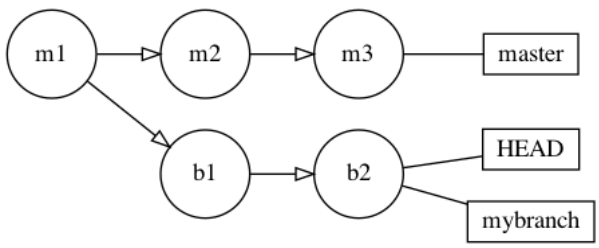
To learn how `reset` works, it's worth revisiting another command you're intimately familiar with: `checkout`.

Comparing reset with checkout

Take the example below, where the branch `mybranch` is a straightforward branch off of `master`:

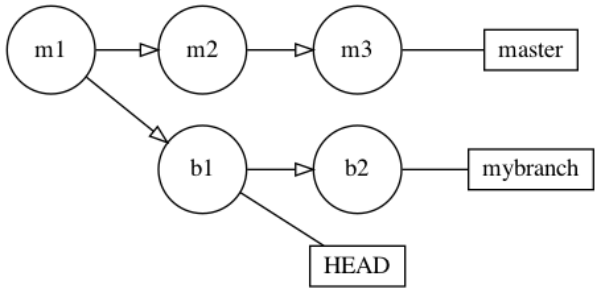


In this case, you're working on `master`, and `HEAD` is pointing to the hash of the last commit on the `master` branch. When you check out a branch with `checkout mybranch`, Git moves the `HEAD` label to point to the most recent commit on the branch:



So `checkout` simply moves the `HEAD` label between commits. But instead of specifying a branch label, you can also specify the hash of a commit.

For example, assume that instead of `checkout mybranch`, you wanted to check out the commit just before the one referenced by `HEAD` — in this case, `b1`:

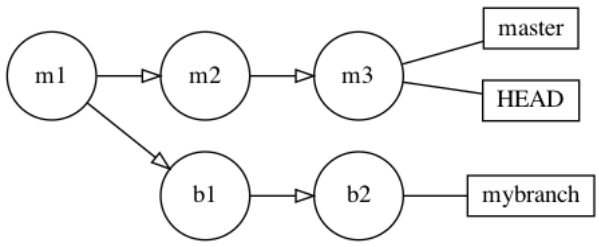


So your working directory now reflects the state of the repository represented by commit `b1`. This is a **detached `HEAD` state**, which simply means that `HEAD` now points to a commit that has no other label pointing to it.

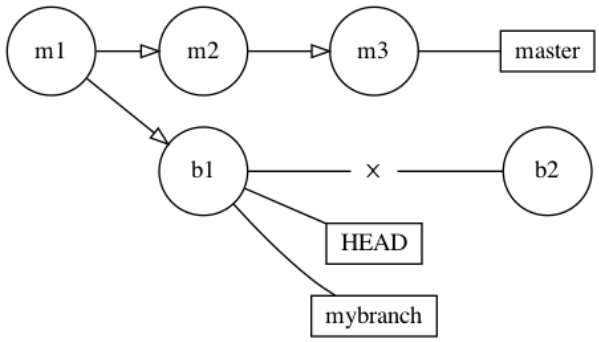
> **Note**: This is a weird (but totally permissible) state from Git's perspective. Git's normal workflow is to either work from the tip of a branch, denoted by the branch's name label, or to work from some other named label in the repository. A detached HEAD state is useful when you want to view the state of the repository at some earlier point in time, but it's not a state you'd be in as part of a normal workflow.

You've seen that `checkout` simply moves `HEAD` to a particular commit. `reset` is similar, but it also takes care of moving the branch's label to the *same* commit instead of leaving the branch label where it was. `reset`, in effect, returns your working environment — including your branch label — to the state a particular commit represents.

Consider again the example above, with a simple branch, `mybranch`, off of `master`:



This time, you execute a `reset` command with a target commit of `b1`:



Both `HEAD` and `mybranch` have now moved back to `b1`. This means you've effectively discarded the original tip commits of the `mybranch` branch, and stepped back to the `b1` commit.

But what happens to the tip commit that's now separated from the `b1` label?

From Git's perspective, it doesn't exist anymore. Git will collect it with its regular garbage collection cycle, and any commits you make on `mybranch` will now stem from the `b1` commit as their ancestor.

In this way, `reset` is quite useful when you're trying to "roll back" commits you've made, to get to an earlier point in your repository history. `reset` has a lot of different use cases for it, and with it several options to learn about that change its behavior.

Working with the three flavors of reset

Remember that Git needs to track three things: your working directory, the staging area and the internal repository index. Depending on your needs, you can provide parameters to `reset` to roll back either all those things or just a selection:

- **soft**: Leaves your working directory and staging area untouched. It simply moves the reference in the index back to the specified commit.
- **mixed**: Leaves your working directory untouched, but rolls back the staging area and the reference in the index.
- **hard**: Leaves nothing untouched. This rolls your working directory, your staging area and the reference in the index back to the specified commit.

To understand `reset` more fully, you'll work through a few scenarios in your repository to see how it affects each of the three areas above.

Start by extracting the compressed repository from the **starter** directory to a convenient location on your machine, then navigating into that directory from the command line.

Testing git reset –hard

`git reset ––hard` is most people's first introduction to "undoing" things in Git. The `––hard` option says to Git, "Please forget about the grievous things I've done to my repository and restore my entire environment to the commit I've specified".

The default commit for `git reset` is `HEAD`, so executing `git reset ––hard` is the equivalent of saying `git reset HEAD ––hard`.

To see how this can get you out of a sticky situation, you'll make some rather ill-considered changes to your repository, check the state of the index and staging area then execute `git reset ––hard` to see how Git can "undo" that mess for you.

Removing an utterly useless directory

Start by going to the command line and navigating to the root directory of your repository. Execute the following command to get rid of that pesky **js** directory, which doesn't look very important:

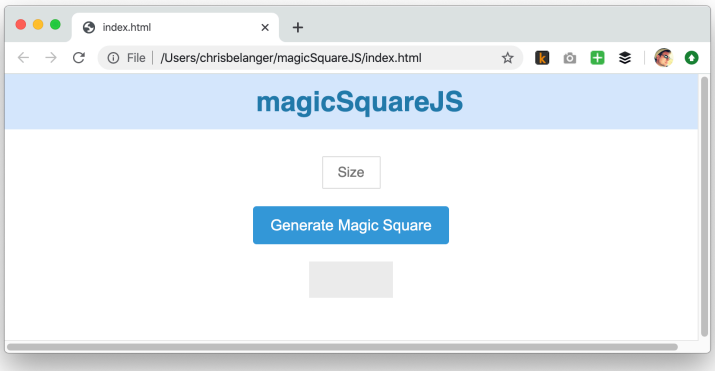
```
git rm –r js
```

This uses the `git rm` command to not only delete the directory, but also to update Git's staging area with the files deleted as well.

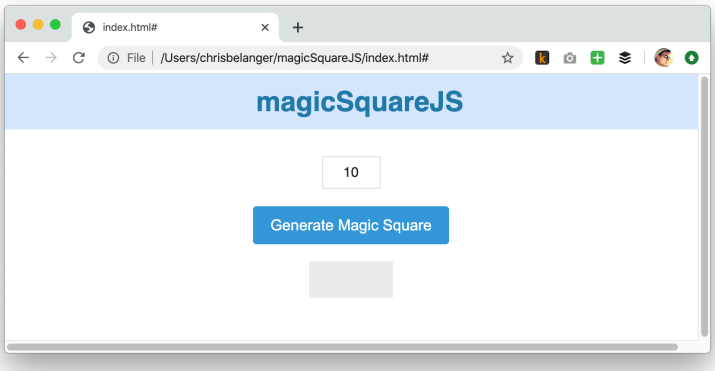
You're ultra-confident you don't need that directory, nor do you even need to test your changes (does anyone even *use* JavaScript anymore?), so you also commit your changes to the repository:

```
git commit –m "Deletes the pesky js directory"
```

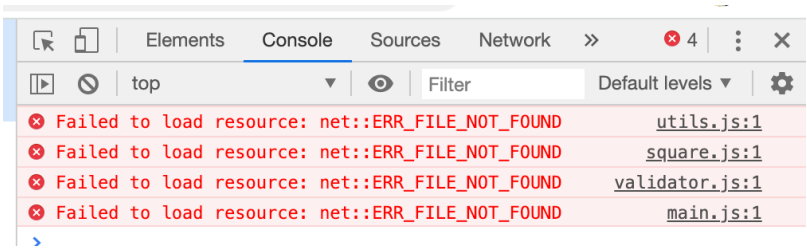
Now, open **index.html** in a browser and you'll find that the site still looks great, despite the loss of the **js** directory:



But enter a number in the field and click the **Generate Magic Square** button and you'll find that nothing happens at all:



Even worse, take a look at the developer console of the browser and you'll see the following JavaScript errors:



Whoops! Looks like you needed that directory after all. But you've gone and committed your work, haven't you? Yes, unfortunately, you have.

Execute the following command to see the commit history of your repository:

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

Sadly, you see your ill-advised commit sitting proudly at the tip of `master`:

```
* 6c5ecf1 (HEAD -> master) Deletes the pesky js directory
```

Oh no, no, no, no, no. How will you get that directory back now?

The first option is to panic, delete everything you're working on, and re-clone the repository.

Luckily, there's really no need to go to those lengths. Git remembers *everything*, so it's easy to get back to a previous state.

Restoring your directory

In this case, you want to return to the last commit before you made your blunder. You don't even need to know the commit hash; you can provide *relative* references to `git reset` instead.

Execute the following command to return your working directory, your staging area and your index to the previous commit:

```
git reset HEAD^ --hard
```

Here, the caret character, `^`, means "the first immediate ancestor commit just before `HEAD`".

Look at your working directory and you'll see that your **js** directory has reappeared. To be sure, open **index.html** in a browser and you'll see that your magic square generator now functions as it did before.

Whew! You dodged that bullet.

This situation allowed you to completely blow away everything in your working directory. But what if you had something in there that you wanted to keep?

That's where the other parameters for `git reset` come to the rescue.

Trying out git reset –mixed

Imagine that you're working on another software project. You're up late, the coffee ran out hours ago and you're tired. That never happens in real life, of course, but bear with me.

You want to create a temporary file to hold some login information. You're a responsible developer, so you'd *never* commit that sensitive information to the repository.

Create a file named **SECRETS** in your working directory with the command below:

```
touch SECRETS
```

Then add some ultra-secret information with the `echo` command:

```
echo 'password=correcthorsebatterystaple' >> SECRETS
```

Now, assume some time has passed and you've made lots of progress on your website. You want to get to bed as soon as you can, so you use the shortcut `git add` to add all your changes to the staging area:

```
git add .
```

And then you commit your changes, like the responsible developer you are:

```
git commit –m "Adds final styling for website"
```

No sooner have you pressed the **Enter** key, when you realize — with a start — that you committed **SECRETS**, too!

Well, you're fully awake now, thanks to that burst of adrenaline, and you're in quite a pickle.

Fortunately, you haven't pushed your changes to the remote yet, so that's one less mess to untangle. But you'd like to get **SECRETS** out of the commit history so you don't look like a total fool.

Removing your unwanted commit

You *could* use `git reset HEAD^ --hard`, as above, but that would blow away hours of hard work. Instead, use `git reset --mixed` to reset the commit index and the staging area, but leave your working directory alone.

This will let you add the **SECRETS** file to your .gitignore — which you should have done in the first place, silly — and preserve all your work.

Execute the following command to reset only the index and the staging area to the previous commit:

```
git reset HEAD^ --mixed
```

Now execute `git status` to see that **SECRETS** is now untracked:

```
On branch master
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:   SECRETS
```

There you are — you're back to the state right before you executed `git add .`. You're now free to add **SECRETS** to your .gitignore (lesson learned), then stage and commit all your hard work.

Execute the following commands to do just that:

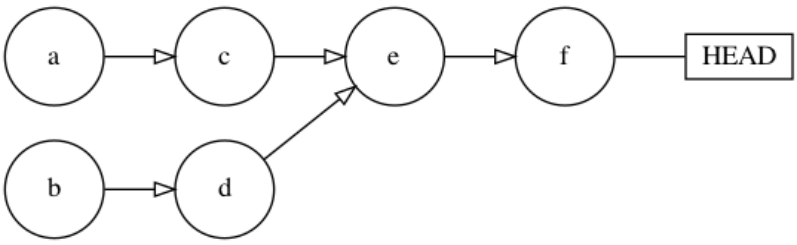
```
echo SECRETS >> .gitignore
git add .gitignore
git commit -m "Updates .gitignore"
```

Do you *always* have to use HEAD^; that is, do you always have to go to the previous commit?

No, not at all! It's what you'll use most of the time, in practice, but you can specify any commit in Git's history by using its commit hash.

There are other ways to specify a commit relative to `HEAD`. These are handy when you're going back farther than one level in history, or you're dealing with a commit that has more than one parent, like merge commits.

Take the image below, which shows a simple two-branch scenario that's been merged. In this diagram, from the perspective of time, commit `a` occurred first, commit `b` second, commit `c` happened third and so on.



Here's how you can use relative references to get to each point in this tree:

HEAD^1: References the first immediate ancestor of this commit in history: commit `e`. This is the simple case, since the commit referenced by `HEAD`, `f`, only has one ancestor. This is equivalent to the shorthand: `HEAD^`.

HEAD^^: References the immediate ancestor of the immediate ancestor of this commit in history: commit `c`. Because commit `e` has two ancestors — `c` and `d` — Git chooses the "oldest" or first ancestor of `e`: `c`.

HEAD^^2: References the second immediate ancestor of the immediate ancestor this commit in history: commit `d`. Because commit `e` has two ancestors — `c` and `d` — specifying `^2` (parent #2) chooses the "newer" or later ancestor of `e`: `d`.

HEAD^^^: References the first immediate ancestor, of the first immediate ancestor, of the first immediate ancestor of this commit in history: `a`. Here, you go back three generations, always following the "older" path first.

HEAD^^2^: References the first immediate ancestor, of the second immediate ancestor, of the first immediate ancestor of this commit in history: commit `b`. The first `^` tells Git to look back at the first ancestor of this commit. The next `^2` tells Git to look at the newer ancestor, and the final `^` looks to the ancestor of *that* commit: `b`.

If you'd like more information about trees and graph traversals, read up on the finer details of relative references, such as `HEAD~`, in the **Specifying Revisions** section of Git's `man` pages. This contains a more complex commit tree and instructions on how to traverse this tree with relative references.

Using git reset –soft

If you like to build up commits bit by bit, staging changes as you make them, then you may encounter a situation where you've staged various changes and committed them prematurely.

In that situation, use `git reset --soft` to roll back that commit while leaving your meticulously-built staging area intact.

You've dodged that bullet with the **SECRETS** file, but now you have a few more changes to make. You have two files to add as part of this commit: a configuration file and a change to README.md that explains how to set all the parameters in the configuration file.

Create the configuration file first:

```
touch setup.config
```

Now, stage that change:

```
git add setup.config
```

Next, execute the following command to add a line of text to the end of **README.md**:

```
echo "For configuration instructions, call Sam on 555-555-5309
any time" >> README.md
```

Making a mistake

Just before you add that to the staging area, Will and Xanthe call you excitedly with their plans for their next big project: to create a — wait for it — magic triangle generator. You humor them for a while, then turn your attention back to your project.

Did you add everything to the staging area? You're pretty sure you did, so you commit what's in the staging area:

```
git commit -m "Adds configuration file and instructions"
```

However, your keen eye notices the output message from Git:

```
[master c416751] Adds configuration file and instructions
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 setup.config
```

Zero insertions… that doesn't make sense. Wait, did you stage that change to **README.md**? No, you didn't, because you were distracted by Will and Xanthe.

Cleaning up your commit

So now, you need to clean up that commit so it includes both the change to **README.md** and the addition of **setup.config**.

All that's missing is that small change from **README.md**, so `git reset --soft` will roll back your commit and let you stage and commit that one change.

Execute the following to do a soft reset to the previous commit:

```
git reset HEAD^ --soft
```

Stage that small change from **README.md**:

```
git add README.md
```

Now, you can commit those changes again:

```
git commit -m "Adds configuration file and instructions"
```

Did it work? The output from Git confirms it did:

```
[master 297be58] Adds configuration file and instructions
 2 files changed, 2 insertions(+)
 create mode 100644 setup.config
```

You can use `git log` to see the actual contents of that commit with the following command:

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

The output tells you that yes, you've committed both **config.setup** and that change to **README.md**:

```
diff --git a/README.md b/README.md
index 331487d..fb18f7c 100644
```

```
.
.
.
+For configuration instructions, call Sam on 555-555-5309 any
time
diff --git a/setup.config b/setup.config
new file mode 100644
index 0000000..e69de29
```

There you go. You were able to salvage your carefully-crafted staging area without having to start over. Nice!

So that wraps up the situations where you created a commit that you didn't want in the first place. But what about the reverse situation, where you got rid of a commit that you *didn't* want to lose?

Using git reflog

You know that Git remembers *everything*, but you probably don't realize just how deep Git's memory goes.

Head to the command line and execute the following command:

```
git reflog
```

You'll get a *ton* of output. Here are the top few lines of mine:

```
297be58 (HEAD -> master) HEAD@{0}: commit: Adds configuration
file and instructions
6b51dc9 HEAD@{1}: reset: moving to HEAD^
c416751 HEAD@{2}: commit: Adds configuration file and
instructions
6b51dc9 HEAD@{3}: reset: moving to HEAD^
9142192 HEAD@{4}: commit: Adds final styling for website
6b51dc9 HEAD@{5}: reset: moving to HEAD^
6c5ecf1 HEAD@{6}: commit: Deletes the pesky js directory
6b51dc9 HEAD@{7}: filter-branch: rewrite
1bc3d71 (refs/original/refs/heads/master) HEAD@{8}: filter-
branch: rewrite
32281cf HEAD@{9}: filter-branch: rewrite
fdb857a HEAD@{10}: rebase -i (abort): updating HEAD
59f601b HEAD@{11}: rebase -i (pick): Linking to the main CSS
file
e725307 HEAD@{12}: rebase -i (pick): Creating basic CSS file
.
.
.
```

It looks a bit like a stash file, doesn't it? The Git **ref log** is like the world's most detailed play-by-play sports commentator. It's a running historical record of absolutely everything that's happened in your repo, from commits, to resets, to rebases and more.

Think of it as Git's undo stack — you can use it to get back to a particular point in time.

Press **Q** to exit the ref log. It's time to see how to resurrect commits that you assumed were long gone.

Finding old commits

You've rethought your changes above. Putting configuration elements in a separate file in the repo along with instructions isn't the best way to go about things. It obviously makes more sense to put those settings, along with Sam's mobile number, on the main wiki page for this project.

That means you can delete that last commit, and you might as well use `git reset --hard` to reset your working directory as well, to keep things clean.

Execute the following command to roll back to the previous commit:

```
git reset HEAD^ --hard
```

Now, check your commit history with the following command. You'll see that `HEAD` now points to the previous commit in the tree. No sign of your commit with the **Adds configuration file and instructions** message remains:

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

Just then, Yasmin pings you via DM. "Hey," she says, "Can you share those two files that have the setup configuration and Sam's mobile number? I'll stick them in the wiki myself. Thanks!"

But — you've gotten rid of that commit with `git reset`. How do you get it back?

Well, use the following command to take a look at Git's ref log to see if you can recover that commit:

```
git reflog
```

Here are the first two lines of my ref log:

```
6b51dc9 (HEAD -> master) HEAD@{0}: reset: moving to HEAD^
297be58 HEAD@{1}: commit: Adds configuration file and
instructions
```

Looking at these two lines in order, the `HEAD@{0}` reference is the `git reset` action you just applied, while the `HEAD@{1}` reference is your previous commit.

So you'll want to go back to the state that `HEAD@{1}` references to get those changes back. To get there, you'll use the `git checkout` command.

Recovering your commit with git checkout

Even though you usually use `git checkout` to switch between branches, as you saw way back at the beginning of this chapter, you can use `git checkout` and specify a commit hash, or in this case, a reflog entry, to create a detached `HEAD` state. You'll do that now.

First, execute `git checkout` with the reflog reference of the commit you want:

```
git checkout HEAD@{1}
```

Git will notify you that you're in a detached `HEAD` state:

```
Note: checking out 'HEAD@{1}'.

You are in 'detached HEAD' state. You can look around, make
experimental
changes and commit them, and you can discard any commits you
make in this
state without impacting any branches by performing another
checkout.

If you want to create a new branch to retain commits you create,
you may
do so (now or later) by using -b with the checkout command
again. Example:

  git checkout -b <new-branch-name>

HEAD is now at 297be58 Adds configuration file and instructions
```

Read through that above output before you go any further. Git has some excellent advice about what you might want to do in a detached `HEAD` state.

You definitely *do* want to retain any commits you create, so you'll need to create a branch to hold them, then merge those changes into `master`.

To see just how this looks in your commit tree, use `git log` to look at the top two commits of the tree:

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

You'll see some output, similar to the following:

```
* 297be58 (HEAD) Adds configuration file and instructions
* 2401800 (master) Updates .gitignore
```

You can tell from `git log` that this is a detached `HEAD` state, since it's not referencing any branch — it's just hanging out there on its own. If `HEAD` referenced a branch, you'd see it in the git log as `6c5ecf1 (HEAD -> master)` or similar.

To save your work in a detached `HEAD` state, use `git checkout` like this:

```
git checkout -b temp
```

That command creates a new branch, `temp`, based on what `HEAD` was pointing to. In this case, that's the detached commit you retrieved from Git's ref log. The command then updates `HEAD` to point to the tip of the `temp` branch.

Checking that your changes worked

Look at your commit tree again with `git log --all --oneline --graph` and you'll see something like this:

```
* 297be58 (HEAD -> temp) Adds configuration file and
instructions
* 2401800 (master) Updates .gitignore
```

`HEAD` now points to the `temp` branch, as you expected. If you pull a directory listing with `ls`, you'll notice that **setup.config** and your one-line change to **README.md** have both been preserved.

To prove that your changes are actually on a proper branch, switch back to `master` with `git checkout master`. Then, execute the following command to see what the tree looks like:

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

Git shows that your resurrected commit is on the `temp` branch and you're safely back on `master`:

```
* 297be58 (temp) Adds configuration file and instructions
* 2401800 (HEAD -> master) Updates .gitignore
```

Using git revert

In all of this work with `git reset` and `git reflog`, you haven't pushed anything to a remote repository. That's by design. Remember, you can't change *shared* history. Once you've pushed something, it's a lot harder to get rid of a commit since you have to synchronize with everyone else.

However, there's one final way to *mostly* undo what you've done in a commit. `git revert` takes the patchset of changes you applied in a specified commit, rolls back those changes, and then creates an entirely new commit on the tip of your branch.

To see this in action — and to learn why I say it can *mostly* undo your changes — you'll merge in the `temp` branch you created above, revert those changes then take a look at your commit history to see what you've done.

Setting up your merge

First, merge in that branch. Ensure you're on `master` to start:

```
git checkout master
```

Then, merge in the `temp` branch like so:

```
git merge temp
```

Git responds with what it's done:

```
Updating 6b51dc9..297be58
Fast-forward
 README.md    | 1 +
 setup.config | 0
 2 files changed, 1 insertion(+)
 create mode 100644 setup.config
```

OK, a fast-forward merge. That makes sense, since `temp` was a direct descendant of the changes on `master`.

Look at your commit history with `git log --all --oneline --graph` and you'll see something like the following:

```
* 297be58 (HEAD -> master, temp) Adds configuration file and
instructions
* 2401800 Updates .gitignore
```

There's `temp`, `master` and `HEAD`. Looks like your merge went fine.

You merrily push those changes to the remote… but then have second thoughts. You decide you *don't* want those changes, after all.

However, you just pushed those changes — and on `master`, of all places — so they're shared with everyone else.

Reverting your changes

While you can't change shared history, you can at least revert the changes you've made here to get back to the previous commit.

`git revert`, like most other Git commands, accepts a target commit: a label, a commit hash or other reference.

Again, you can use relative references to specify the commit you want to revert. In this case, however, you're simply reverting the last changes you made, so you'll use `HEAD` as a reference.

Execute the following command to revert the last change you made to `master`. `git revert` creates a new commit as the result of its actions. To avoid having to go into Vim and edit the message, you'll use the `--no-edit` switch to just accept the default revert message that Git provides:

```
git revert HEAD --no-edit
```

Git tells you what it's doing:

```
[master 82cfe6d] Revert "Adds configuration file and
instructions"
 2 files changed, 1 deletion(-)
 delete mode 100644 setup.config
```

If you compare that with the previous commit from earlier in this chapter that added these changes, you'll see that it's the exact inverse operation:

```
[master 297be58] Adds configuration file and instructions
 2 files changed, 1 insertion(+)
 create mode 100644 setup.config
```

Now, take a look at your commit history with `git log --all --oneline --graph` to see what happened:

```
* 82cfe6d (HEAD -> master) Revert "Adds configuration file and
instructions"
* 297be58 (temp) Adds configuration file and instructions
* 2401800 Updates .gitignore
```

You can see that `git revert` created a new commit at the tip of the `master` branch: `82cfe6d`. If you're still a little unsure what that commit actually did, use the `git log -p -1` command to see the contents of the patch for that commit:

```
diff --git a/README.md b/README.md
index fb18f7c..331487d 100644
--- a/README.md
+++ b/README.md
.
.
.
-For configuration instructions, call Sam on 555-555-5309 at
anytime
diff --git a/setup.config b/setup.config
deleted file mode 100644
index e69de29..0000000
```

The reason it *mostly* undoes your changes is that you still have the original commit that added these undesired changes in history.

If it offends you that the original commit is still in the history, use the techniques in Chapter 5, "Rebasing to Rewrite History" to fix that problem.

And with that, you've seen most of the ways you can undo your work in Git. Hopefully, you've learned some techniques to help you avoid relying on `git reset HEAD --hard` as a scorched earth technique to get your repository back in working order.

Key points

Congratulations on finishing this chapter! Here's a quick recap of what you've covered:

- A **detached HEAD** situation occurs when you check out a commit that no other branch or labeled reference points to.
- **git reset** updates your local system to reflect the state represented by `<commit>`. It also moves `HEAD` to `<commit>`, unlike `git checkout <commit>`.
- Git's regular garbage collection process will eventually clean up any commits left **unreferenced** due to `git reset`.
- **git reset –soft** leaves your working directory and staging area untouched, and simply moves the reference in the index back to the specified commit.
- **git reset –mixed** leaves your working directory untouched, but rolls back the staging area and the reference in the index.
- **git reset –hard** leaves nothing untouched. It rolls your working directory, your staging area and the reference in the index back to the specified commit.
- Use **relative references** to specify a commit, such as `HEAD^` and `HEAD~`.
- **git reflog** shows the entire history of all actions on your local repository and lets you pick a target point to revert to.
- **git revert** applies the inverse of the patch of the target commit to your working directory and creates a new commit.
- **git revert –no-edit** bypasses the need to edit the commit message in Vim.

Where to go from here?

You've already covered quite a lot in this chapter, but I recommend reading a bit more about how relative references work in Git.

Here are two good resources on relative references:

- https://stackoverflow.com/questions/2221658/whats-the-difference-between-head-and-head-in-git
- https://git-scm.com/docs/git-rev-parse#_specifying_revisions

In particular, they'll show you the difference between relative addressing using `HEAD~` and `HEAD^`. Knowing the difference will save you a *lot* of grief in the future when you're trying to fix a repo that seems beyond repair.

This brings an end to the in-depth exploration of the ins and outs of Git internals and the various commands you can use to achieve mastery over your repository.

However, Git is rarely used in isolation. You'll usually use Git in a team setting, so your team will have to collaborate and agree about which workflows to use to avoid stepping on each others' toes.

The next section of the book covers Git development workflows, so if you're struggling to figure out just how to implement Git across your teams, you'll find the upcoming chapters useful.

Section II: Git Workflows

Now that you understand how Git works and how to use some of the advanced features, you need to learn how to incorporate Git into your software development lifecycle. There are established best practices and several formal Git workflows out there.

Those formal Git workflows, well, they're all good, and in some cases, they're all bad. It depends what you want to accomplish in your repo, and how your own team works. GitFlow is one of the most popular branching strategies, but there are alternative models that work well in many situations. This section will introduce you to these workflows and branching models, and explain what problems they solve and what problems they create.

Chapter 8: Centralized Workflow

By Jawwad Ahmad

A centralized workflow is the simplest way to start with Git. With this system, you work directly on master instead of working in a branch and merging it with master when you're done.

Creating branches in Git is extremely easy, so you should only skip them when they're absolutely unnecessary.

In this chapter, you'll learn about scenarios where the centralized workflow is a good fit. You'll also learn how to handle common situations that arise when multiple developers are committing directly to master.

When to use the centralized workflow

One of the main reasons to first commit and push your code to a branch is to allow other developers to review your code before you push it to master. If the code doesn't need to be reviewed, the overhead of creating and pushing a separate branch is unnecessary. That's where the centralized workflow is a great fit.

Here are a few scenarios where a code review may not be necessary.

1. When working alone

If you're the sole developer on a project, you don't need the overhead of creating branches since there are no other developers to review your code.

Consider the commands you'd run if you were committing your feature to a branch before merging it to master:

```
git checkout -b my-new-feature # 1: Create and switch to branch
# Write the code
git add . && git commit -m "Adding my new feature"
git checkout master            # 2: Switch back to master
git merge my-new-feature       # 3: Merge branch into master
git branch -d my-new-feature   # 4: Delete branch
git push master
```

Compare that to how you'd handle the same update using a centralized workflow. You'd skip the four numbered commands above and end up with only:

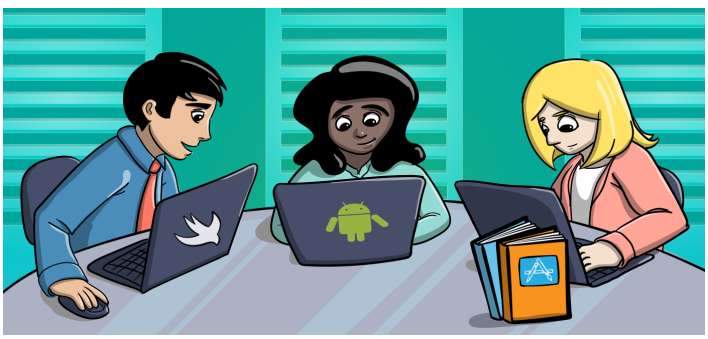
```
# Write the code
git add . && git commit -m "Adding my new feature"
git push master
```

Even when using the centralized workflow, there are still valid reasons to create branches. For example, if you have experimental or incomplete code that you aren't ready to commit to master, you can commit it to a branch and revisit it later.

In the centralized workflow creating branches is optional since you're allowed to push your commits directly to the master branch. This isn't the case in the feature branch workflow which you'll learn about in the next chapter. In that workflow creating branches is required since pushing to master directly is not allowed.

2. When working on a small team

If you're part of a small team where each team member has a specialized area of knowledge, a centralized workflow is a good choice. For example, if one developer works on backend code using one programming language and another works on front-end code in a different language, it's not always useful or practical for those team members to review code outside of their area of expertise.



Small team with non-overlapping expertise or code ownership

In another common scenario, each developer owns a specific area of the code. For example, in an iPhone app, one developer works on the search flow while another works on settings and account preferences. In this scenario, each member of the team is completely responsible for making the changes they need and ensuring their changes work correctly.

3. When optimizing for speed

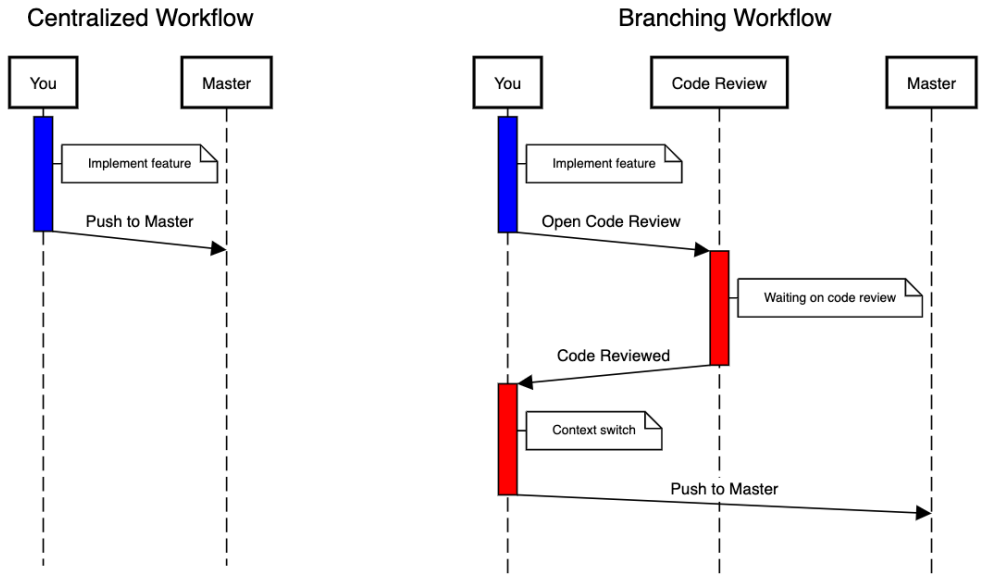
Code reviews are a great way to improve the code's quality before pushing it to the central repository, but every code review has some overhead.

After the author commits their change, they need to wait for someone to review it, which can block them from moving forward.

Furthermore, emails and alerts about code reviews are disruptive. Some team members might stop what they're doing to take a quick look at the code review request to see if they can review it immediately. If not, they need to devote time to do it later. Context switching is especially expensive when performing focused work, such as software development.

Any code that's pending review creates a mental burden for both the author and the rest of the review team.

The following sequence diagram illustrates some of the extra time and overhead required.



Centralized Workflow vs Branching Workflow

The first red section illustrates the overhead of waiting for your code to be reviewed. The second shows the context switch you have to make when you interrupt what you are currently working on to go back and merge the original code into master.

The longer a code review takes, the more likely it is that other people introduce conflicts that you'll have to resolve manually.

If you want to optimize for speed and reduce interruptions, your team can adopt a strategy where code doesn't have to be reviewed before the author pushes it to the remote master branch.

Keep in mind that not reviewing code before pushing it to master doesn't mean that the team can't review the code afterward. It just means that the code on master might not be clean and perfect the first time around.

On the other hand, even well-reviewed code is far from perfect. When optimizing for speed, it might make sense to allow for a bit more entropy for the sake of expediency.

This doesn't mean that you can't have your code reviewed. You can always create a branch to request an ad-hoc code review on a new or complex feature. It just means that there isn't a blanket policy to require a code review for every new feature.

4. When working on a new project

The need for expediency is often stronger when you're working on a new project with tight deadlines. In this case, the inconvenience of waiting for a code review may be especially high.

While bugs are undesirable in any context, unreleased projects have a higher tolerance for them since their impact is low. Thus, you don't have to scrutinize each commit as thoroughly before you push it to master.

Drop dead launch date! Must ship by the 8th!

Even if your new project doesn't start off using a centralized workflow, don't be surprised if your team lets you commit and push directly to master once the deadline approaches!

Centralized workflow best practices

Here are some best practices you can adopt to make using the centralized workflow easier. These are especially important when working in teams where multiple developers are committing to master.

Two important things to keep in mind are to rebase early and often and to prefer rebasing over creating merge commits. If you do accidentally create a merge commit, you can undo it as long as you haven't pushed it to the remote repository.

Rebase early and often

When using the centralized workflow in a team, you often have to rebase before pushing to master to avoid merge commits.

Even before you're ready to push your locally-committed code to the remote repository, you'll benefit from rebasing your work onto any newly-committed code that's available in master. You might pull in a bug fix or code you need for features that you're building upon.

The earlier you resolve conflicts and integrate your work-in-progress with the code on master, the easier it is to do. For example, if you're using a variable that was recently renamed, you'll have fewer updates to make if you pull it in sooner.

Remember, you want to use the `--rebase` option with the `git pull` command so you rebase any commits on your local `master` branch on `origin/master` instead of creating a merge commit. You'll work through an example of this shortly.

Undo accidental merge commits

At times, your local `master` branch may diverge from the remote `origin/master` branch. For example, when you have local commits that you haven't pushed yet, and the remote `origin/master` has newer commits pushed by others.

In this case, executing a simple `git pull` will create a merge commit. Merge commits are undesirable since they add an extra unnecessary commit and make it more challenging to review the Git history.

If you've accidentally created a merge commit, you can easily undo it as long as you haven't pushed it to master.

In the project, you'll work through an example to demonstrate this workflow and how to handle some of the issues you'll encounter when working directly on `master`.

Getting started

To simulate working on a team, you'll play the role of two developers, Alex and Beth!

Alex and Beth are working on an HTML version of a TODO list app called Checklists. They've just started work on the project, so there isn't much code.

And don't worry, you won't be adding much code to it throughout the next few chapters since your main focus will be to use it for learning various Git workflows.

Start by unzipping the **repos.zip** file from the **starter** folder for this chapter. You'll see the following unzipped directories within **starter**:

```
starter
└── repos
    ├── alex
    │   └── checklists
    ├── beth
    │   └── checklists
    └── checklists.git
```

At the top level, there are three directories: **alex**, **beth** and **checklists.git**. Within the **alex** and **beth** directories are checked-out copies of the **checklists** project.

What's unique about this setup is that **checklists.git** is configured as the remote origin for both Alex's and Beth's checked-out Git repositories. So when you push or pull from within Alex's or Beth's **checklists** repository, it will push to and pull from the local **checklists.git** directory instead of a repository on the internet.

The easiest way to work on the project is to have three separate terminal tabs open. Open your favorite terminal program, then open two additional tabs.

> **Note**: If you're on a Mac, **Command-T** opens a new tab in both **Terminal.app** and **iTerm2.app**, and **Command-Number** switches to the appropriate tab. For example, **Command-2** switches to the second tab.

Once you have three tabs open, cd to the **starter** folder and then to **repos/alex/checklists** in the first tab, **repos/beth/checklists** in the second tab and **repos/checklists.git** in the third tab.

```
cd path/to/starter/repos/alex/checklists # 1st Tab
cd path/to/starter/repos/beth/checklists # 2nd Tab
cd path/to/starter/repos/checklists.git  # 3rd Tab
```

To check what the remote origin repository is configured as, run the following command within **alex/checklists** or **beth/checklists**:

```
git config --get remote.origin.url # Note: The --get is optional
```

You'll see the following relative path, which indicates that the remote origin repository is the **checklists.git** directory:

```
../../checklists.git
```

If the remote repository were on GitHub, this URL would have started with either **https://github.com** or **git@github.com** instead of being a local path.

Alex's and Beth's respective projects have been configured with their name and email, so when you commit from within their checklists folder, the commit author will show as Alex or Beth.

While you could run `git config user.name`, and `git config user.email` to verify this, sometimes it's easier to just peek at the local `.git/config` file.

Run the following from within **alex/checklists** or **beth/checklists**:

```
cat .git/config
```

At the end of the file, you'll see their **user.name** and **user.email** settings:

```
...
[user]
    name = Alex Appleseed
    email = alex@example.com
```

Note: Your own name and email should already be configured in your global .gitconfig file. You can run `cat ~/.gitconfig` to verify this.

State of the project

The remote origin repository, **checklists.git**, contains four commits, which we'll refer to as A1, B1, A2 and B2 instead of with their commit hashes. Alex's and Beth's projects also have local commits that have not yet been pushed to the remote. Alex has one additional commit, A3, and Beth has two, B3 and B4.

In your terminal, switch to the **checklists.git** tab and run **`git log --oneline`**:

```
824f3c7 (HEAD -> master) B2: Added empty head and body tags
3a9e970 A2: Added empty html tags
b7c58f4 B1: Added index.html with <!DOCTYPE html> tag
a04ae7f A1: Initial Commit: Added LICENSE and README.md
```

You can see the four commits on `origin`: A1, B1, A2 and B2.

> **Note**: The **checklists.git** repository is a *bare* repo, which means that it only contains the history without a working copy of the code. You can run commands that show you the history, like `git log`, but commands that give you information about the state of the working copy, such as `git status`, will fail with an error, `fatal: this operation must be run in a work tree`.

Next switch to the **alex/checklists** tab and run **`git log --oneline`**:

```
865202c (HEAD -> master) A3: Added Checklists title within head
824f3c7 (origin/master, origin/HEAD) B2: Added empty head and...
3a9e970 A2: Added empty html tags
b7c58f4 B1: Added index.html with <!DOCTYPE html> tag
a04ae7f A1: Initial Commit: Added LICENSE and README.md
```

You can see A3 in addition to the four commits already on `origin/master`.

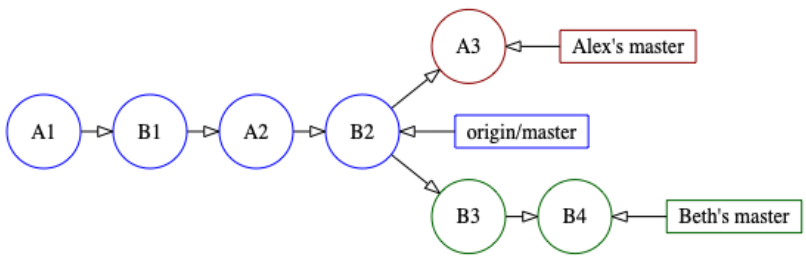
> **Note**: Some commit messages, such as for B2 above, will be shortened to end with an ellipsis (`...`) to fit them on a single line.

Finally, switch to the **beth/checklists** tab and run **`git log --oneline`**:

```
4da1174 (HEAD -> master) B4: Added "Welcome to Checklists!" w...
ed17ce4 B3: Added "Checklists" heading within body
824f3c7 (origin/master, origin/HEAD) B2: Added empty head and...
3a9e970 A2: Added empty html tags
b7c58f4 B1: Added index.html with <!DOCTYPE html> tag
a04ae7f A1: Initial Commit: Added LICENSE and README.md
```

You can see B3 and B4 in addition to the four commits already on `origin/master`.

Here is a combined view of the commits in the three repositories:



Relationship between origin/master and Alex and Beth's master branches

So while Alex and Beth are both working on `master`, their branches have diverged.

At this point, either Alex or Beth could push their commits to `origin`, but once one of them does, the other won't be able to.

For the remote to accept a push, it needs to result in a fast-forward merge of `master` on the remote. In other words, the pushed commits need to be direct descendants of the latest commit on `origin/master`, i.e. of B2.

Currently, both Alex's and Beth's commits qualify to be pushed. But once the remote's `master` branch is updated with one person's commits, the other won't be able to push without rebasing or creating a merge commit.

You'll have Beth push her commits to `origin` first.

Pushing Beth's commits to master

Switch to **beth/checklists** in your terminal and run **`git status`**. It should show the following to verify that it's ahead of `origin/master` by two commits:

```
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
...
```

Now, run **`git push`** to push Beth's commits to the remote `master` branch.

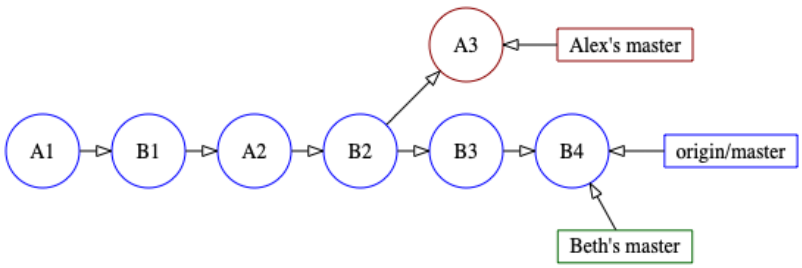
It'll successfully push both commits to the remote repository, i.e. to **checklists.git**.

Switch to the **checklists.git** tab and run **`git log --oneline`**:

```
4da1174 (HEAD -> master) B4: Added "Welcome to Checklists!" w...
ed17ce4 B3: Added "Checklists" heading within body
824f3c7 B2: Added empty head and body tags
...
```

You can see Beth's two additional commits B3 and B4, ahead of B2.

This is what it looks after Beth's push:



Relationship between origin/master and local master branches after Beth's push

Next, you'll attempt to push Alex's A3 commit to `master`.

Pushing Alex's commit to master

Switch to **alex/checklists** and run **`git status`**:

```
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
...
```

Alex's repository still thinks it's one commit ahead of `origin/master`. This is because he hasn't yet run a `git fetch` after Beth's push.

You'll run `git fetch` in a moment, but first, run **`git push`** to see what happens:

```
To ../../checklists.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to '../../checklists.git'
hint: Updates were rejected because the remote contains work
hint: that you do not have locally. This is usually caused by
hint: another repository pushing to the same ref. You may want
hint: to first integrate the remote changes (e.g.,
```

```
hint: 'git pull ...') before pushing again.
...
```

Uh oh. Take a look at the hint message piece by piece.

First, it says:

```
Updates were rejected because the remote contains work that you
do not have locally.
```

That's right, since it now contains the two additional commits from Beth: B3 and B4.

Then it says:

```
This is usually caused by another repository pushing to the same
ref.
```

Yes, that's exactly what Beth just did.

And finally, it suggests:

```
You may want to first integrate the remote changes (e.g., 'git
pull ...') before pushing again.
```

That's what you'll do next. But first, run **`git status`** again; you'll see that it still thinks Alex's branch is ahead of `origin/master` by one commit:

```
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
...
```

Although the origin repository rejected the changes, the local repository still hasn't fetched updates from `origin`.

Run **`git fetch`** to fetch updates from the remote. When you run **`git status`** now, it will correctly show that your local `master` branch has diverged from `origin/master`:

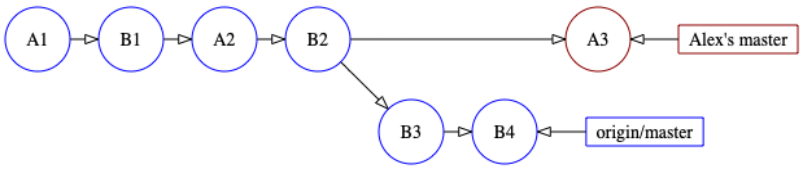
```
On branch master
Your branch and 'origin/master' have diverged,
and have 1 and 2 different commits each, respectively.
  (use "git pull" to merge the remote branch into yours)
```

Run **`git log --oneline --graph --all`** to see the log in graph format:

```
* 865202c (HEAD -> master) A3: Added Checklists title within ...
| * 4da1174 (origin/master, origin/HEAD) B4: Added "Welcome t...
| * ed17ce4 B3: Added "Checklists" heading within body
```

```
|/
* 824f3c7 B2: Added empty head and body tags
* 3a9e970 A2: Added empty html tags
* b7c58f4 B1: Added index.html with <!DOCTYPE html> tag
* a04ae7f A1: Initial Commit: Added LICENSE and README.md
```

Which is just a textual representation of the following:



Visual representation of the previous git log --oneline --graph --all command

> **Note**: Without the `--graph` option, it would have looked like the commit history was all on one branch. Without the `--all` option, it would only have shown you the commits on your current branch — that is, on `master` but not on `origin/master`. Try running the command without each of the options for comparison.

You can see that your local master has diverged from `origin/master`. You can't push to the remote repository in this state.

There are two ways you can resolve this issue:

1. The first and recommended way is to run `git pull` with the `--rebase` option to rebase any commits to your local master branch onto `origin/master`.
2. The second way is to create a merge commit by running `git pull`, committing the merge and pushing the merge commit to the remote.

Since it's easy to forget the `--rebase` option and simply run `git pull`, you'll use the non-recommended way first so you can also learn how to undo an accidentally-created merge commit.

Undoing a merge commit

Since Alex's `master` branch has diverged from `origin/master`, running a `git pull` will result in a merge commit.

This is because **`git pull`** is actually the combination of two separate commands: **`git fetch`** and **`git merge origin/master`**.

If Alex didn't have any local commits, then the implicit `git merge` part of the command would perform a fast-forward merge. This means that Alex's `master` branch pointer would simply move forward to where `origin/master` is pointing to. However, since `master` has diverged, this creates a merge commit.

1. Abort the merge commit

The easiest way to prevent a merge commit is to short-circuit the process by leaving the commit message empty.

From **alex/checklists**, run **`git pull`**. Vim will open with the following:

```
Merge branch 'master' of ../../checklists
# Please enter a commit message to explain why this merge is
# necessary, especially if it merges an updated upstream into
# a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message
# aborts the commit.
```

Take a look at the last line of the commit message template. It says:

```
Lines starting with '#' will be ignored, and an empty message
aborts the commit.
```

This means that you can enter **dd** to delete the first line and leave the remaining lines since they all start with a **#**.

However, there's something reassuring about clearing the complete commit message. Since it takes the same number of keystrokes, you'll do that instead. Enter **dG** to delete everything until the end and then **:wq** to exit.

Now, you'll see the following:

```
Auto-merging index.html
error: Empty commit message.
Not committing merge; use 'git commit' to complete the merge.
```

As the last line above indicates, you aborted the commit of the merge, but not the merge itself.

You can verify this by running a `git status`:

```
...
All conflicts fixed but you are still merging.
...
```

Run the following command to abort the merge itself:

```
git merge --abort
```

Congratulations, merge commit averted!

2. Hard reset to ORIG_HEAD

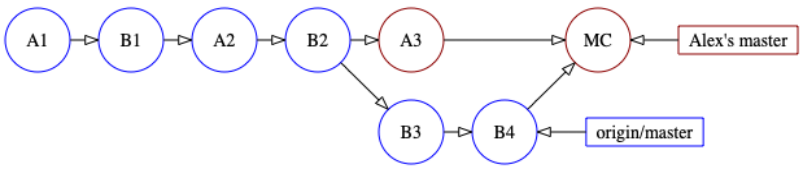
So what can you do if you accidentally created the merge commit? As long as you haven't pushed it yet, you can reset your branch to its original commit hash before the merge.

Run **`git pull`** again to trigger the merge. When Vim opens, type **`:wq`** to accept the default message and commit the merge.

Now run **`git log --oneline --graph`**:

```
*   fc15106 (HEAD -> master) Merge branch 'master' of ../../c...
|\
| * 4da1174 (origin/master, origin/HEAD) B4: Added "Welcome t...
| * ed17ce4 B3: Added "Checklists" heading within body
* | 865202c A3: Added Checklists title within head
|/
* 824f3c7 B2: Added empty head and body tags
* 3a9e970 A2: Added empty html tags
* b7c58f4 B1: Added index.html with <!DOCTYPE html> tag
* a04ae7f A1: Initial Commit: Added LICENSE and README.md
```

Visually, your repository is in the following state:



Now you have a merge commit, MC, that is a combination of all of the contents of `origin/master` that weren't in your branch yet. In this case, MC would contain the code from Beth's B3 and B4 commits.

As long as you haven't pushed the merge commit to master, you can undo it. First, however, you have to determine what the commit hash of Alex's `master` branch was before the merge, and then run `git reset --hard` using that commit hash.

One way to identify the commit hash is by looking at the commit log. You can visually see that **865202c** is the commit hash for the **A3** commit, which is where **master** was before the merge, so you could run `git reset --hard 865202c`.

There's also an easier way to identify the commit hash before the merge. When Git commits a merge operation, it saves the original commit hash before the merge into `ORIG_HEAD`.

If you're curious, you can run either of the following commands to see what the commit hash is for `ORIG_HEAD`:

```
git rev-parse ORIG_HEAD
```

or

```
cat .git/ORIG_HEAD
```

This shows the following:

```
865202c4bc2a12cc2fbb94f5980b00457d270113
```

Run the following command to perform the reset:

```
git reset --hard ORIG_HEAD
```

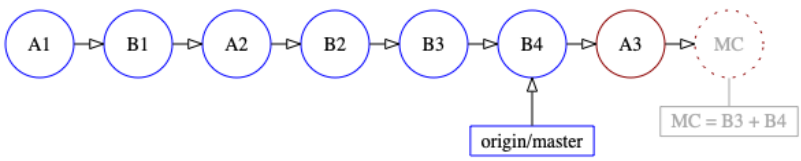
You should see the following confirmation message:

```
HEAD is now at 865202c A3: Added Checklists title within head
```

You're back to where you started, which is exactly what you wanted!

3. Rebase the merge commit

Another strategy you can adopt is to rebase the merge commit onto `origin/master`. This applies A3 and the merge commit on top of B4. Since `origin/master` already has B3 and B4, i.e., the contents of the merge commit, this removes the merge commit entirely.



Create the merge commit again by running `git pull` and then **:wq** to save the commit message.

Now run the following:

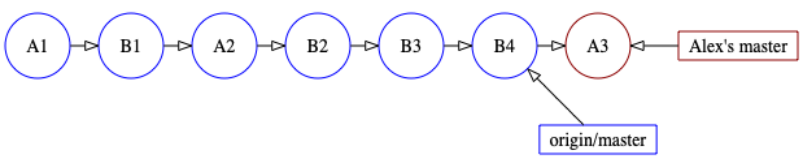
```
git rebase origin/master
```

Then run **`git log --oneline --graph`** to take a look at the commit history:

```
* 7988360 (HEAD -> master) A3: Added Checklists title within ...
* 4da1174 (origin/master, origin/HEAD) B4: Added "Welcome to ...
* ed17ce4 B3: Added "Checklists" heading within body
* 824f3c7 B2: Added empty head and body tags
* 3a9e970 A2: Added empty html tags
* b7c58f4 B1: Added index.html with <!DOCTYPE html> tag
* a04ae7f A1: Initial Commit: Added LICENSE and README.md
```

You can see that you rebased A3 on top of B4, and the merge commit has disappeared!

Visually, your repository is now in the following state:



This is the same outcome that you would have had with `git pull --rebase`, which is what you'll try next.

You could push at this point, but instead, you'll reset your branch again so you can try `git pull --rebase`. Since you rebased after the merge, you can no longer use `ORIG_HEAD`, so you'll reset to the commit hash directly. Resetting to `ORIG_HEAD` would have taken you back to the merge commit before the rebase.

Run the following:

```
git reset --hard 865202c
```

Then run **`git log --oneline --graph --all`** to verify that you've reset `master`.

Using git pull –rebase

You previously learned that **`git pull`** is the combination of two separate commands: **`git fetch`**, and **`git merge origin/master`**.

Adding the **`--rebase`** option to **`git pull`** essentially changes the second **`git merge origin/master`** command to **`git rebase origin/master`**.

Run **`git pull --rebase`**. You'll see the following:

```
First, rewinding head to replay your work on top of it...
Applying: A3: Added Checklists title within head
```

Then run **`git log --oneline --graph`** to take a look at the commit history:

```
* 4742353 (HEAD -> master) A3: Added Checklists title within ...
* 4da1174 (origin/master, origin/HEAD) B4: Added "Welcome to ...
* ed17ce4 B3: Added "Checklists" heading within body
...
```

You can see that you've now rebased your local A3 commit onto `origin/master`.

Reset your master branch one final time for the next exercise:

```
git reset --hard 865202c
```

Setting up automatic rebase

You may occasionally forget that you have local commits on `master` before you run `git pull`, resulting in a merge commit. Of course, this is no longer a terrible issue since you now know how to abort and undo merge commits.

But wouldn't it be swell if Git could automatically take care of this for you? And it can! By setting the **`pull.rebase`** option to **`true`** in your Git configuration, you can do just that.

Run the following command to set Git up to always rebase when you run `git pull`:

```
git config pull.rebase true
```

Now run **`git pull`**. It will automatically rebase your commit on top of `origin/master` instead of creating a merge commit.

Now, finally, the moment you've been working toward! Run **`git push`** to push Alex's newly rebased commit to the `master` branch of the remote.

```
git push
```

Voila! You can now `git pull` without having to remember to add the `--rebase` option.

One final point to keep in mind is that each developer on your team would have to configure this option for themselves. If there are common configuration options like this that would be useful for everyone on the team, consider adding them to something like a `setup_git_config.sh` file that you'd commit to the repository.

Key points

- The centralized workflow is a good fit when working alone or on small teams, when optimizing for speed or when working on a new, unpublished project.
- You can still create branches for in-progress code or for ad-hoc code reviews.
- Rebase frequently to incorporate upstream changes and resolve conflicts sooner.
- Prefer **`git pull --rebase`** instead of **`git pull`** to avoid creating merge commits.
- Set the **`pull.rebase`** option to **`true`** in your Git config to automatically rebase when pulling.
- There are multiple ways to undo accidental merge commits as long as you haven't pushed them to the remote repository.

Now that you have a good handle on using the centralized workflow, the next step in your Git journey is to branch towards the branching workflow. Proceed to the next chapter to get started!

Chapter 9: Feature Branch Workflow

By Jawwad Ahmad

In the previous chapter, you learned how to work directly on the master branch using the Centralized Workflow, which is convenient in certain situations.

Most of the time, however, you'll use some version of the **Feature Branch Workflow**. Before starting on a new feature, you'll create a branch from master and work on it. Once you're done, you'll merge the feature branch back into master.

Creating a feature branch essentially gives you your own frozen version of the master branch. It also allows you to delay pushing your commits to master until your feature is complete, which keeps the master branch in a more stable state for everyone.

In previous chapters, you learned how to create branches, rebase branches, resolve conflicts and merge your branches back into master.

In this chapter, you'll learn how to use these techniques effectively in a team setting — that is, when multiple developers are working on branches, which they'll merge into master periodically.

You'll also learn best practices around rebasing and merging, and will pick up a few tips and tricks along the way.

When to use the Feature Branch workflow

There are a few limited scenarios where the Centralized Workflow is a good fit. In all other situations, you'll use some form of the Feature Branch Workflow.

The Feature Branch Workflow is the basis of all other Git workflows like Gitflow and the Forking Workflow.

Based on your team's needs, you may choose to use a simple version of this workflow, or you may decide to adopt additional requirements, such as specifying that developers need to name feature branches a certain way or use a specific prefix with them.

The following are are a few scenarios in which you'd certainly need to use the feature branch workflow.

When developing features in parallel

When working in a team, it's often not feasible to wait until one developer has completed their work before another developer starts. Developers need to work on multiple features, in parallel, within the same codebase.

For example, one team might modify a page's design while another team adds additional content to it.

It's also not feasible for you to work on code that keeps changing while you're also changing it. The code you're working on needs to remain stable until you're ready to pull in other updates to it.

Even when working on your own, you might be in the middle of working on one feature when you have to switch to working on a different one. You'd need a way to store that in-progress code somewhere until you can come back to it.

Creating a feature branch allows a developer or a team to work on a certain snapshot of the code until they're ready to integrate it back into master.

When your code needs a review

Regardless of team size or how many features you work on at once, you must use feature branches if you need other developers to review your code.

If your code needs a review before you merge it into master, then by definition, you can't use the master branch to push your code for review!

When sharing code still in development

Feature branches allow you to share code before you merge it into master. For example, you might need code that another developer is currently working on, and so isn't available in master yet. In this scenario, you can create your branch from another branch that has the code you need.

Once you merge the other branch into master, you can rebase onto master, which will remove the other branch's commits from your branch. This allows you to start working with code that's still in development.

When collaborating on a feature

Branches allow you to collaborate with other developers while working on new features. Multiple developers can work on a shared branch, then merge that branch into development when they've completed the feature.

This allows master to remain stable while the feature is under development. Once the feature's development and testing phases are complete, it can be merged into master all at once.

Getting started

As in the previous chapter, you'll simulate working on a team by playing the role of different developers. However, you'll switch roles a bit more in this chapter.

A few things have happened since the last chapter. A new developer, Chad, has joined the team, and the team has switched to using the Feature Branch Workflow.

Start by unzipping **repos.zip** from the **starter** folder for this chapter. You'll now see a checked-out project for Chad within the **starter/repos** folder:

```
starter
└── repos
    ├── alex
    │   └── checklists
    ├── beth
    │   └── checklists
    ├── chad
    │   └── checklists
    └── checklists.git
```

As in the previous chapter, open four tabs in your Terminal app and open the following directories within each tab:

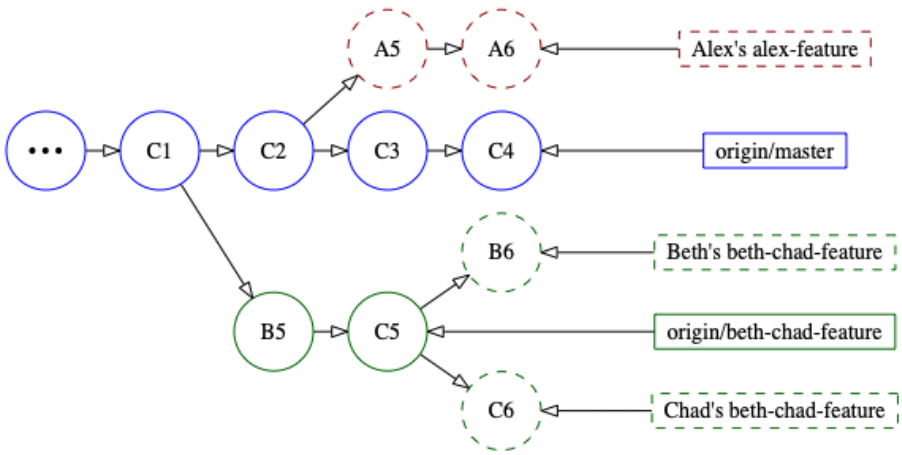
```
cd path/to/starter/repos/alex/checklists  # 1st Tab
cd path/to/starter/repos/beth/checklists  # 2nd Tab
cd path/to/starter/repos/chad/checklists  # 3rd Tab
cd path/to/starter/repos/checklists.git   # 4th Tab
```

Each developer's repository has local commits that they haven't pushed to the remote server. The following section will give you an overview of the branches you'll work with and the state of the commits on them.

Initial project state

The team has been hard at work on two feature branches. Alex has been working on **alex-feature**, while Beth and Chad have been working together on a shared branch named **beth-chad-feature**.

The following image gives you a combined view of the initial state of each developer's local repository and its relation to the origin remote. The solid-outlined commits have been pushed to the remote, while the dashed-outlined commits are still in each developer's local repository. The initial ellipsis (•••) node represents all commits on **master** before C1.



Solid nodes are on the remote, dashed nodes are local commits

Alex created his **alex-feature** branch when **master** was at C2 and has added two local commits, A5 and A6, on it. He hasn't pushed this branch to the remote yet, so the branch only exists in his local repository.

Beth and Chad created the shared **beth-chad-feature** branch when **master** was at C1. They've pushed its first two commits, B5 and C5, to the remote. Both Beth and Chad have one additional commit, B6 and C6 respectively, on their local version of the shared branch.

Just to confirm which commits have been pushed to the remote, switch to the **checklists.git** tab in Terminal and run the following:

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

You'll see the following confirming that **master** is at C4, as in the image above, and that **beth-chad-feature** is at C5 on the remote:

```
* b2deca5 (beth-chad-feature) C5: Added <footer> to <body>
* 4fbfda4 B5: Moved <h1> and <p> within <header>
| * 51bdc3c (HEAD -> master) C4: Updated section styling to u...
| * 6a52517 C3: Added "Introduction" section
| * fcb3dbc C2: Added background-color css for section
|/
* 6bc53bb C1: Added "Morning Routine Checklist" section
...
```

You won't need the fourth **checklists.git** tab for anything else in this chapter, so you can close it now to simplify things.

Since this chapter is a bit more involved, the following section will give you an overview of the tasks you'll perform in this chapter.

Project roadmap

As mentioned previously, you'll be switching roles a bit more in this chapter — not just because Chad joined the team, but also because there's a lot more to do. :]

The following is a quick roadmap of what you'll do in this chapter. You don't have to remember all of this. Its purpose is to give you an idea of the different tasks you'll perform so that you're mentally prepared for what's next.

You'll start by updating the **alex-feature** and **beth-chad-feature** branches with the new code on **master**. Since one branch is shared and the other isn't, you'll update each branch differently.

You'll also need to make sure your feature branches still work correctly after you update them with code from master. And, of course, there will be unintended side effects that you'll need to fix!

You'll fix an issue on **alex-feature**, and then will push the branch up for review. Then you'll review the branch as Beth and will merge it into master.

Then since master has been updated again, Chad will update the shared branch with the new code in master before pushing the shared branch up for review. And of course, this chapter wouldn't be complete without having to resolve a merge conflict!

Before you dive in, there's one final thing to learn: Why should you update feature branches with the latest code in master *before* you merge them if you're just going to merge them into master anyway? You'll cover that in the next section.

Importance of updating branches with master

There are two main reasons to update your branches with new code in master.

The first and most important is for a correct code review. Once you're done working on your branch and are ready to push it up for review, you'll want to ensure that your code will integrate properly with the newest code on the master.

There might be conflicts that you need to resolve or other changes that you need to make based on the latest changes on the master branch. Reviewing code based on an outdated version of master could lead to bugs once you merge the code into master.

The second reason is so your own code doesn't diverge too far from the master branch. If there are new updates in master that affect your branch, the sooner you integrate them, the fewer changes you'll have to make later.

How to update branches with master

There are two ways of updating your branches with master: You can either rebase your branch onto master or you can merge the master branch into your local branch.

If you're working on a local branch that you haven't pushed to the remote yet, rebasing is better. Rebasing your branch avoids merge commits and makes the history easier to review.

On the other hand, if you're working on a shared branch that's already been pushed to the remote, such as the **beth-chad-feature** branch, you should merge master into your branch instead. You should never rebase public branches that other developers are using.

One exception is if you're the only one working on a branch you've pushed to the remote. Sometimes developers will periodically push long-running branches to the remote as a backup.

If no one else is using your branch, you can rebase it. Since rebasing rewrites the branch history, you'll have to force-push it after you rebase.

Updating the two project branches

Since **alex-feature** hasn't been pushed to origin yet, you'll rebase it onto **master**. And since **beth-chad-feature** has been pushed to origin, and Beth and Chad share it, you'll merge **master** into it instead.

Each developer has already pulled master, so their local master branches are up to date with the remote ones.

Switch to the **alex/checklists** tab in Terminal and run the following to verify the current state of Alex's local repository:

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

You'll see the following:

```
* b2deca5 (origin/beth-chad-feature) C5: Added <footer> to <b...
* 4fbfda4 B5: Moved <h1> and <p> within <header>
| * 9f06a73 (HEAD -> alex-feature) A6: Added "Evening Routine...
| * 427b5ee A5: Added h2 color to style.css
| | * 51bdc3c (origin/master, origin/HEAD, master) C4: Update...
| | * 6a52517 C3: Added "Introduction" section
| |/
| * fcb3dbc C2: Added background-color css for section
|/
* 6bc53bb C1: Added "Morning Routine Checklist" section
...
```

You aren't really interested in the **origin/beth-chad-feature** reference, which is complicating the log — you just want to see the **alex-feature** branch in relation to **master**. Instead of using `--all`, you can specify a list of branches to include.

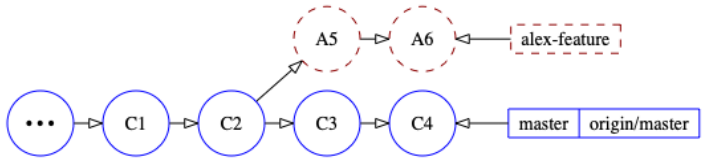
Run the previous command with **alex-feature master** instead of **–all**:

```
git log --oneline --graph alex-feature master
```

You'll see the following, which should look much better:

```
* 9f06a73 (HEAD -> alex-feature) A6: Added "Evening Routine C...
* 427b5ee A5: Added h2 color to style.css
| * 51bdc3c (origin/master, origin/HEAD, master) C4: Updated ...
| * 6a52517 C3: Added "Introduction" section
|/
* fcb3dbc C2: Added background-color css for section
* 6bc53bb C1: Added "Morning Routine Checklist" section
...
```

Visually, this is equivalent to the following:

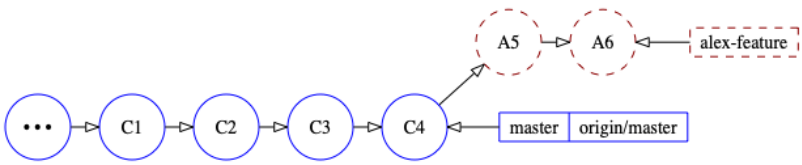


Current position of the alex-feature branch

Since **alex-feature** is the current branch, run the following to rebase it onto **master**:

```
git rebase master
```

The rebase should succeed without conflicts. The branch is now directly ahead of master:



Position of alex-feature after running: git rebase master

You also can run the previous log command again to verify the rebase:

```
git log --oneline --graph alex-feature master
```

You'll see that that the history is now linear:

```
* 28d0ae5 (HEAD -> alex-feature) A6: Added "Evening Routine C...
* b0f7244 A5: Added h2 color to style.css
* 51bdc3c (origin/master, origin/HEAD, master) C4: Updated se...
* 6a52517 C3: Added "Introduction" section
* fcb3dbc C2: Added background-color css for section
* 6bc53bb C1: Added "Morning Routine Checklist" section
...
```

Congratulations, you've successfully updated the **alex-feature** branch with changes in **master**. Next, you'll update the **beth-chad-feature** branch by merging **master** into it.

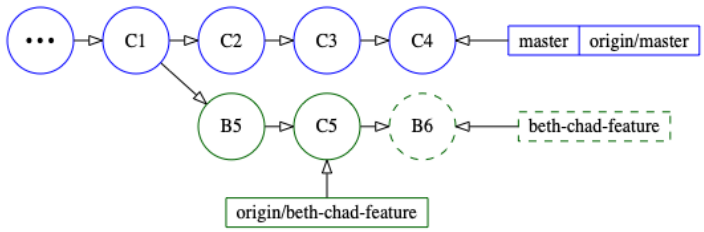
Switch to the **beth/checklists** tab in Terminal and run:

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

You'll see:

```
* 19f8c99 (HEAD -> beth-chad-feature) B6: Added <hr/> in <hea...
* b2deca5 (origin/beth-chad-feature) C5: Added <footer> to <b...
* 4fbfda4 B5: Moved <h1> and <p> within <header>
| * 51bdc3c (origin/master, origin/HEAD, master) C4: Updated ...
| * 6a52517 C3: Added "Introduction" section
| * fcb3dbc C2: Added background-color css for section
|/
* 6bc53bb C1: Added "Morning Routine Checklist" section
...
```

Visually, this is equivalent to the following:



State of Beth's local repository

Now, merge master into the **beth-chad-feature** branch, which is already checked out:

```
git merge master
```

Type **:wq** to accept the default commit message in Vim.

To verify the merge, run the previous log command again:

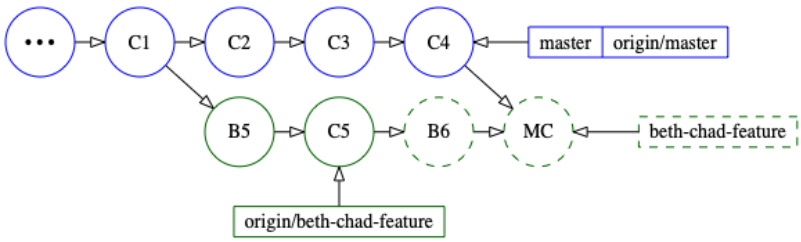
```
git log --oneline --graph --all
```

You'll see the following, which confirms the merge:

```
*   fbffe40 (HEAD -> beth-chad-feature) Merge branch 'master'...
|\
| * 51bdc3c (origin/master, origin/HEAD, master) C4: Updated ...
| * 6a52517 C3: Added "Introduction" section
| * fcb3dbc C2: Added background-color css for section
* | 19f8c99 B6: Added <hr/> in <header>
* | b2deca5 (origin/beth-chad-feature) C5: Added <footer> to ...
```

```
* | 4fbfda4 B5: Moved <h1> and <p> within <header>
|/
* 6bc53bb C1: Added "Morning Routine Checklist" section
...
```

Representing the merge commit with MC, you now have the following:

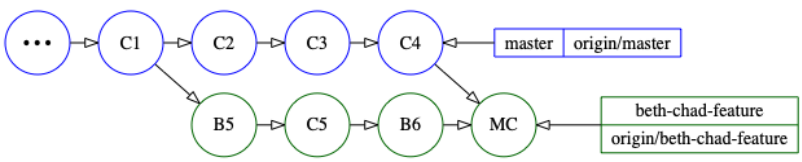


After merging master into beth-chad-feature

Next, Beth will push her local B6 commit and the merge commit to the remote branch. Run the following to push the branch to origin:

```
git push
```

Now, the origin version of **beth-chad-feature** has both C6 and the merge commit:



After pushing beth-chad-feature to origin

Next, you'll have Chad pull the newest changes on **beth-chad-feature** to get the updates that Beth pushed to the branch.

Switch to the **chad/checklists** tab in Terminal.

Before you fetch, run the same log command to view the state of the repository:

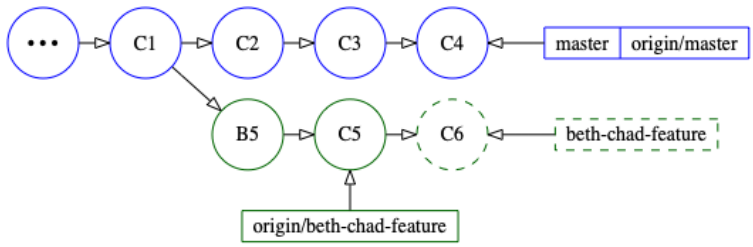
```
git log --oneline --graph --all
```

You'll see something similar to what Beth initially had:

```
* 347bcd3 (HEAD -> beth-chad-feature) C6: Removed "Routine" f...
* b2deca5 (origin/beth-chad-feature) C5: Added <footer> to <b...
* 4fbfda4 B5: Moved <h1> and <p> within <header>
| * 51bdc3c (origin/master, origin/HEAD, master) C4: Updated ...
| * 6a52517 C3: Added "Introduction" section
```

```
| * fcb3dbc C2: Added background-color css for section
|/
* 6bc53bb C1: Added "Morning Routine Checklist" section
...
```

Which is equivalent to the following:



Chad's local **beth-chad-feature** branch is still directly ahead of the remote tracking **origin/beth-chad-feature** branch since Chad hasn't fetched or pulled yet.

Run the following to fetch the latest updates from the remote

```
git fetch
```

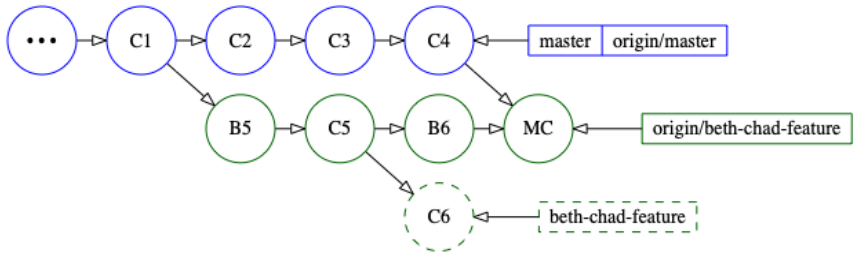
And then run the same log command:

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

You'll see the following:

```
*   fbffe40 (origin/beth-chad-feature) Merge branch 'master' ...
|\
| * 51bdc3c (origin/master, origin/HEAD, master) C4: Updated ...
| * 6a52517 C3: Added "Introduction" section
| * fcb3dbc C2: Added background-color css for section
* | 19f8c99 B6: Added <hr/> in <header>
| | * 347bcd3 (HEAD -> beth-chad-feature) C6: Removed "Routin...
| |/
|/|
* | b2deca5 C5: Added <footer> to <body>
* | 4fbfda4 B5: Moved <h1> and <p> within <header>
|/
* 6bc53bb C1: Added "Morning Routine Checklist" section
...
```

The following image will make it a bit easier to comprehend:



State of Chad's local repository after fetching Beth's updates

Chad's **beth-chad-feature** branch has diverged from **origin/beth-chad-feature**. If you recall from the previous chapter, this is the same situation that occurred when your **master** had local commits but another developer had updated **origin/master**.

In this situation, in which the remote branch has diverged, running a normal `git pull` will result in a merge commit, so you'll need to add the `--rebase` option.

Run the following:

```
git pull --rebase
```

Whoa, there's a merge conflict!

```
...
CONFLICT (content): Merge conflict in index.html
error: Failed to merge in the changes.
Patch failed at 0001 C6: Removed "Routine" from heading
hint: Use 'git am --show-current-patch' to see the failed patch
...
```

The changes Beth merged from master are causing this merge conflict.

Run the command from the hint above to see the change it was trying to apply:

```
git am --show-current-patch
```

You'll see the following:

```
C6: Removed "Routine" from heading
...
       <section>
-        <h2>Morning Routine Checklist</h2>
+        <h2>Morning Checklist</h2>
       </section>
...
```

It seems like a simple change. Chad removed the word **Routine** from **Morning Routine Checklist** to make it **Morning Checklist**.

Open **index.html** in a text editor to resolve the conflict. You'll see the conflict markers in the following area:

```
16     <main>
17 <<<<<<< HEAD
18       <section class="intro-section">
19         <h2>Introduction</h2>
20       </section>
21
22       <section class="checklist-section">
23         <h2>Morning Routine Checklist</h2>
24 =======
25       <section>
26         <h2>Morning Checklist</h2>
27 >>>>>>> C6: Remove "Routine" from headings
28       </section>
29     </main>
```

The conflict is because two different commits changed adjacent lines. In the merge commit from master, **class="checklist-section"** has been added to the `<section>` tag, to make it `<section class="checklist-section">`.

In Chad's change in C6, Chad updated the `<h2>Morning Routine Checklist</h2>` line below `<section>` to `<h2>Morning Checklist</h2>`.

Remove the old `<section>` tag line below `=======` (on line 25) and the old `<h2>` line above it (line 23) and then remove all three conflict marker lines.

It should now look like this:

```
16     <main>
17       <section class="intro-section">
18         <h2>Introduction</h2>
19       </section>
20
21       <section class="checklist-section">
22         <h2>Morning Checklist</h2>
23       </section>
24     </main>
```

To complete the rebase, run the following:

```
git add index.html
git rebase --continue
```

To verify that you resolved the conflict correctly, you can run `git show HEAD` or just `git show`, which will show you the contents of the latest commit on the current branch.

```
git show
```

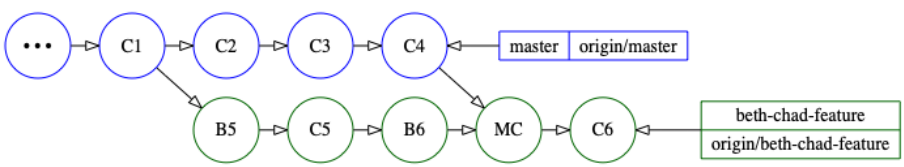
You'll now see the following changes in the diff:

```
C6: Removed "Routine" from heading
...
       <section class="checklist-section">
-        <h2>Morning Routine Checklist</h2>
+        <h2>Morning Checklist</h2>
       </section>
...
```

Now, run **git push** to push your rebased and conflict-free commit to the shared **beth-chad-feature** branch.

```
git push
```

Chad's repository is now in the following state:



State of Chad's repository after rebasing and pushing

Congratulations! You successfully incorporated the latest changes from the **master** branch into **alex-feature** and **beth-chad-feature**. You've also pushed Beth and Chad's local commits to the shared **beth-chad-feature** branch.

Merging the branches into master

Alex is ready to push his branch up for review. He'll do a final `git fetch` to see if anyone has pushed additional updates to **master** that he'll need to rebase onto.

Switch back to the **alex/checklists** tab in Terminal and run `git fetch`:

```
git fetch
```

You'll see that your remote **origin/beth-chad-feature** branch pointer has updated, but there aren't any additional updates to **master**.

```
...
From ../../checklists
 b2deca5..8aad97e  beth-chad-feature -> origin/beth-chad-feature
```

You could also run the previous branch-specific log command to verify this:

```
git log --oneline --graph alex-feature master
```

You can see that the **master** branch is up to date with **origin/master**:

```
* 28d0ae5 (HEAD -> alex-feature) A6: Added "Evening Routine C...
* b0f7244 A5: Added h2 color to style.css
* 51bdc3c (origin/master, origin/HEAD, master) C4: Updated se...
* 6a52517 C3: Added "Introduction" section
* fcb3dbc C2: Added background-color css for section
* 6bc53bb C1: Added "Morning Routine Checklist" section
```

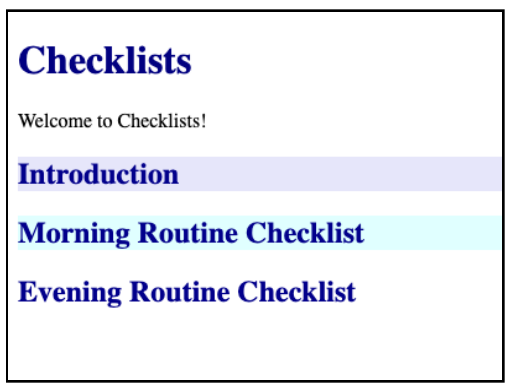
If you recall, there weren't any merge conflicts when rebasing Alex's branch.

However, the absence of a merge conflict doesn't guarantee that any updates Alex made will integrate correctly with any new code that has been added to **master**.

Run the following to open **index.html** in your default browser for a visual check:

```
open index.html
```

You'll see the following:



index.html on the rebased alex-feature branch

Note that the **Evening Routine Checklist** section heading doesn't have the light cyan background that the **Morning Routine Checklist** section heading has. That's because commit C4 changed how the CSS background-color is applied.

To show the content of C4, you could run `git show 51bdc3c`. However, since **master** is at C4, you can also use `git show master` to see the content of the latest commit on `master`.

Run the following to see the changes made in C4:

```
git show master
```

Here's an abbreviated version of the changes you'll see for **index.html**:

```
C4: Updated section styling to use a class
...
diff --git a/index.html b/index.html
     <main>
-      <section>
+      <section class="intro-section">
         <h2>Introduction</h2>
       </section>

-      <section>
+      <section class="checklist-section">
         <h2>Morning Routine Checklist</h2>
...
```

And you'll see the following for **style.css**:

```
diff --git a/style.css b/style.css
...
-section {
+.checklist-section {
     background-color: lightcyan;
 }
+
+.intro-section {
+    background-color: lavender;
+}
```

Prior to C4, the **background-color: lightcyan** style applied to a `<section>` tag. But C3 added an **Introduction** section, which required a different background color. In C4, the **background-color: lightcyan** style was updated to apply to any tags using the **checklist-section** class instead of to all `<section>` tags.

Even though the changes on Alex's branch were correct on their own, they were based on the earlier version of **master** at **C2**. That makes them incorrect after he integrated them with the updated **style.css** file in **master** at **C4**.

To fix this, Alex needs to add the **checklist-section** class to his newly-added **Evening Routine Checklist** section.

Open **index.html** in a text editor and add **class="checklist-section"** within the existing section tag on line 22 to make it **`<section class="checklist-section">`**.

Once you're done, run **git diff**, which should show the following change:

```
-      <section>
+      <section class="checklist-section">
         <h2>Evening Routine Checklist</h2>
```

Since Alex just added the **Evening Routine Checklist** section in A6, which is the latest commit, he can amend it to make this update part of the same commit.

Run the following to amend the current changes into the latest commit:

```
git add index.html
git commit --amend --no-edit
```

The `--no-edit` option allows you to accept the existing commit message. This lets you avoid the extra step of going to the Vim editor and typing **:wq**.

Just to make sure, run **git show** to see the contents of the latest commit.

```
git show
```

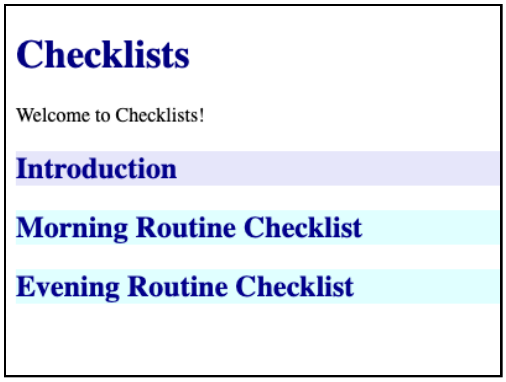
You should see the following change:

```
       </section>
+
+      <section class="checklist-section">
+        <h2>Evening Routine Checklist</h2>
+      </section>
     </main>
```

Now, open **index.html** again or refresh the existing page:

```
open index.html
```

This looks correct!



index.html on the rebased alex-feature branch with the amended commit

Alex is now ready to push his branch to origin for a review.

Run the following command:

```
git push -u origin head
```

In the `push` command, `-u` is shorthand for `--set-upstream`, which creates a remote tracking branch for your current branch. And using `head` is shorthand for using the same branch name. The more verbose command would have been:

```
git push --set-upstream origin alex-feature  # same as above
```

You should see the following confirmation:

```
...
To ../../checklists.git
 * [new branch]      head -> alex-feature
Branch 'alex-feature' set up to track remote branch 'alex-
feature' from 'origin'.
```

At this point, Alex pings the team and lets them know that the **alex-feature** branch is ready to be reviewed and merged.

You'll review the branch as Beth. Switch to your **beth/checklists** tab in Terminal and run **git fetch**:

```
git fetch
```

```
...
From ../../checklists
```

```
   fbffe40..8aad97e  beth-chad-feature -> origin/beth-chad-fe...
 * [new branch]      alex-feature      -> origin/alex-feature
```

You can see that **beth-chad-feature** has been updated from when Chad pushed it, and there's a new **alex-feature** branch.

While not strictly necessary, since you don't have any additional updates to make, you can run **git pull** to pull in the latest changes to **beth-chad-feature**:

```
git pull
```

Next, check out Alex's branch:

```
git checkout alex-feature
```

To review the changes, you'll use the `-p` or `--patch` option with `git log`. You'll also use a revision range specifier to only show the commits since **master**. The format for the range specifier is `<after>..<until>`. So you'll use `master..HEAD` or just `master..` for short.

Run the following command to see the changes on the current **alex-feature** branch, since **master**:

```
git log -p master..
```

The two commits look good, so now merge **alex-feature** into **master** and push it:

```
git checkout master
git merge alex-feature
git push
```

Also, delete the local branch and the remote branch with the following commands:

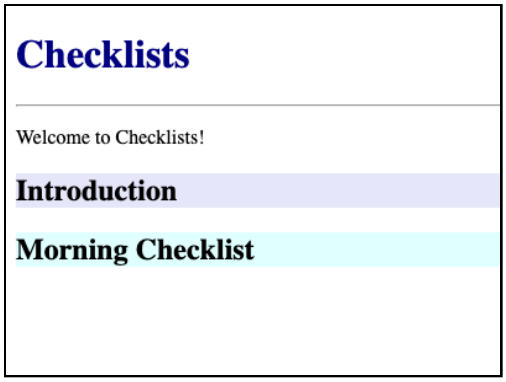
```
git branch -d alex-feature
git push origin --delete alex-feature
```

Congratulations! You've successfully merged the **alex-feature** branch into **master**.

Next, you'll push up the **beth-chad-feature** branch for review as Chad since he made the last commit on the branch.

Switch to the **chad/checklists** tab in Terminal and open **index.html** to make sure it looks correct:

```
open index.html
```



index.html in the beth-chad-feature branch

Chad's latest change in C6 was to remove the word **Routine** from **Morning Routine Checklist**, so that looks correct.

That's the commit with the conflict you resolved above. Now, run **git show** to take another look at the latest commit on the branch:

```
        <section class="checklist-section">
-         <h2>Morning Routine Checklist</h2>
+         <h2>Morning Checklist</h2>
        </section>
```

Since Chad is ready to push his branch for review, he'll do a final fetch to see if there have been any recent updates on master that he needs to integrate. Run **git fetch**.

```
git fetch
```

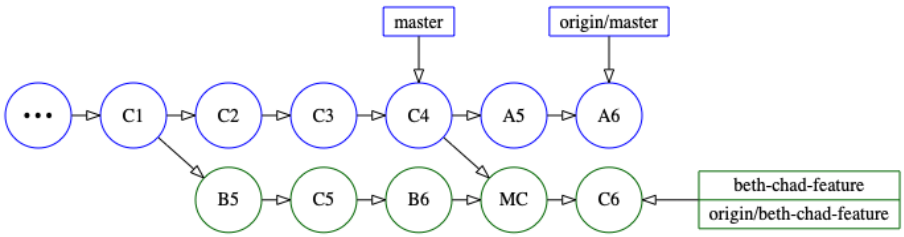
And indeed, the master branch has an update!

```
...
From ../../checklists
   51bdc3c..cca927c  master     -> origin/master
```

This is because **master** now contains the commits from Alex's **alex-feature** branch that Beth merged and pushed.

So now Chad needs to merge **master** into **beth-chad-feature**, make sure there are no issues, and then push the branch up for review.

Here's how Chad's repository currently looks:



If you run **git merge master** now it will say **Already up to date** because you haven't yet pulled the changes from **origin/master** into your local **master** branch.

Run the following commands to update **master**:

```
git checkout master
git pull
git checkout -
```

The dash in **`git checkout -`** takes you back to the previous branch you were on, similar to how `cd -` takes back you to the previous directory you were in.

It will also show you a confirmation that says:

```
Switched to branch 'beth-chad-feature'
```

Now, you're finally ready to merge **master** into your branch:

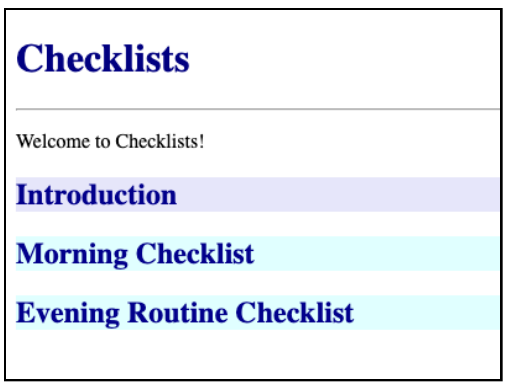
```
git merge master --no-edit
```

Again, `--no-edit` lets you use the default commit message and bypass Vim.

The merge was successful, but does that mean there aren't any issues with the merged-in code? It's best to check.

Once again, open **index.html** by running:

```
open index.html
```



Chad's index.html after merging master into beth-chad-shared

If you recall, in C6, Chad removed the word **Routine** from **Morning Routine Checklist**, which was based on an update from the design team. In the meantime, however, Alex added a section named **Evening Routine Checklist** based on the previous design.

Chad's changes aren't necessarily incorrect, but since he removed the word **Routine** from **Morning Routine Checklist**, he should probably remove it from **Evening Routine Checklist** as well.

Open **index.html** in a text editor and remove the word **Routine** from **Evening Routine Checklist**.

Run **git diff** to confirm you have the following changes:

```
        <section class="checklist-section">
-         <h2>Evening Routine Checklist</h2>
+         <h2>Evening Checklist</h2>
        </section>
```

It would be nice if Chad could amend the C6 commit where the other part of this change was made, but that's already been pushed to a shared branch, so amending it would rewrite its history.

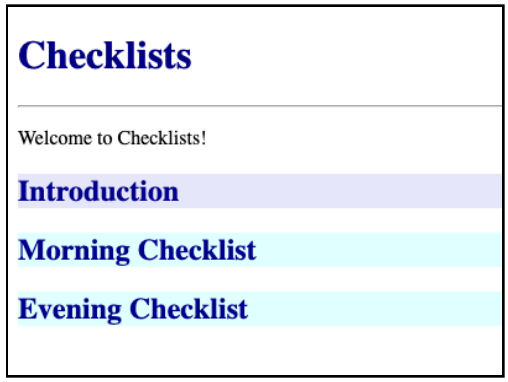
Additionally, the C6 commit is before the merge. It isn't possible to amend it since the changes you need to amend aren't available at that point.

So you'll just add an additional commit to the branch after the merge. Run the following to commit the change:

```
git commit -am "C7: Removed Routine from heading"
```

Open **index.html** one final time to confirm:

```
open index.html
```



It looks good! You're clear to push the latest changes up to the branch for review.

Run **git push**:

```
git push
```

Chad now lets Alex know that the branch is ready for him to review and merge.

Switch to the **alex/checklists** tab in terminal and run **git fetch**:

```
git fetch
```

Alex hasn't fetched since Beth merged his branch so he sees an update to master as well:

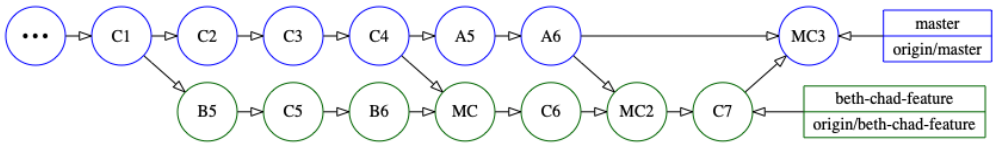
```
...
From ../../checklists
 8aad97e..cde0da3  beth-chad-feature -> origin/beth-chad-feature
 51bdc3c..cca927c  master            -> origin/master
```

Run the following to update **master**, then check out **beth-chad-feature**:

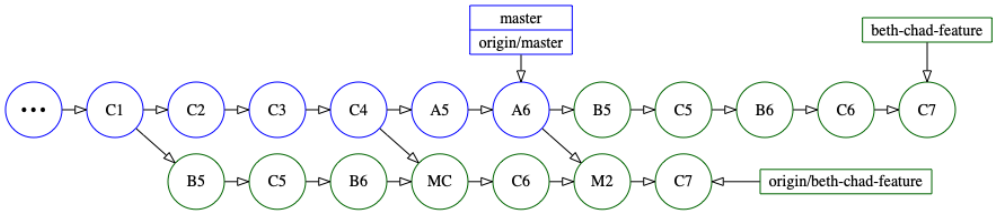
```
git checkout master
git pull
git checkout beth-chad-feature
```

Alex has reviewed the commits on the branch and is ready to merge the branch into master.

He has two options for doing this. In the first, he could merge it as-is, retaining the various merge commits. For example, he could run `git merge beth-chad-feature` and then `git push`, which would result in the following:

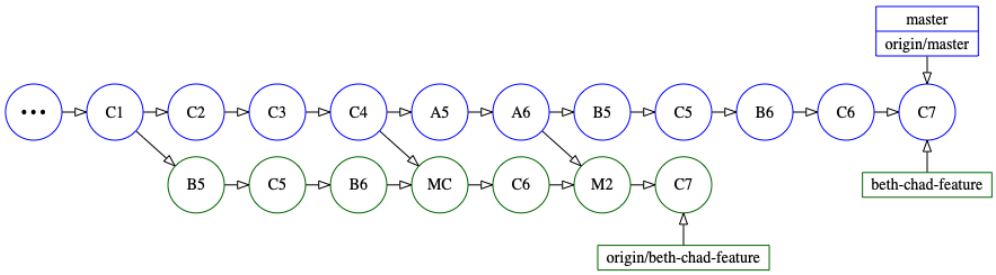


Alternatively, he could first rebase the **beth-chad-feature** branch against master to remove the merge commits:



And then merge the freshly-rebased version of the **beth-chad-feature** branch into **master**, which would result in a fast forward merge.

Pushing that would result in the following:



The reason no one rebased this branch before is that different developers were actively using it. Rebasing a branch while others are using it is never a good idea since it would need to be force-pushed.

However, this is essentially the end of life point for this branch. After he merges it, Alex will delete it. Its purpose as a shared branch is essentially defunct, so it no longer matters if you delete its history. And that's what you'll do next!

Run the following:

```
git checkout beth-chad-feature
git rebase master
```

You'll see that it applies the five commits:

```
First, rewinding head to replay your work on top of it...
Applying: B5: Moved <h1> and <p> within <header>
...
Applying: C5: Added <footer> to <body>
...
Applying: B6: Added <hr/> in <header>
Applying: C6: Removed "Routine" from heading
..
Applying: C7: Removed Routine from heading
```

In this case, the branch rebased cleanly. This won't always be the case if you had to resolve merge conflicts when you merged master into the shared branch. If that happens, you can decide whether to try to resolve the merge conflicts or just merge in the branch without rebasing it.

Now that you've rebased, there's one more optimization you can make. Recall that in C7, you made a change that you would have preferred to merge into C6, but couldn't because there was a merge commit in the way.

Run the following to take a look at the contents of the last two commits:

```
git log -p -2
```

You'll see the following:

```
C7: Removed Routine from heading
...
       <section class="checklist-section">
-        <h2>Evening Routine Checklist</h2>
+        <h2>Evening Checklist</h2>
       </section>
...

C6: Removed "Routine" from heading
...
       <section class="checklist-section">
-        <h2>Morning Routine Checklist</h2>
+        <h2>Morning Checklist</h2>
       </section>
...
```

Now that these commits are adjacent, you can squash them!

Run the following to do an interactive rebase on the last two commits:

```
git rebase -i head~2
```

You'll see:

```
pick 0383368 C6: Removed "Routine" from heading
pick 4652fdc C7: Removed Routine from heading
```

You can change the **pick** for the last commit to an **s** to squash it, which will also allow you to update the commit message. But if you're OK with using the previous commit's message and don't want to reword it, you can use **f** for "fix up". You'll do that here.

Change the **pick** for **C7** to an **f**:

```
pick 0383368 C6: Removed "Routine" from heading
f 4652fdc C7: Removed Routine from heading
```

Now, run **git show** to verify that both changes are in the latest commit:

```
git show
```

You'll see:

```
C6: Removed "Routine" from heading
...
       <section class="checklist-section">
-        <h2>Morning Routine Checklist</h2>
+        <h2>Morning Checklist</h2>
       </section>

       <section class="checklist-section">
-        <h2>Evening Routine Checklist</h2>
+        <h2>Evening Checklist</h2>
       </section>
...
```

You're now ready to merge the rebased branch into master and push it. Run the following:

```
git checkout master
git merge beth-chad-feature
git push
```

Now, delete the local **beth-chad-feature** branch as well as the one on the remote.

Even though you've merged the branch, using the safe delete **-d** option will give you an error. Try it out:

```
git branch -d beth-chad-feature
```

You'll see the following:

```
warning: not deleting branch 'beth-chad-feature' that is not yet
merged to 'refs/remotes/origin/beth-chad-feature', even though
it is merged to HEAD.
error: The branch 'beth-chad-feature' is not fully merged.
If you are sure you want to delete it, run 'git branch -D beth-
chad-feature'.
```

You can delete the branch using the `-D` option. Alternatively, you can delete the remote branch first. That will allow you to delete the branch using the safe `-d` option since if you delete the remote branch, the local branch is no longer associated with an unmerged remote branch.

Run the following:

```
git push origin --delete beth-chad-feature
git branch -d beth-chad-feature
```

Congratulations! You've learned the best practices of how to keep private and public or shared branches up to date with master.

You've also learned the importance of always updating your branch with master before pushing a branch up for review. And finally, that a conflict-free rebase or merge doesn't always mean your code will integrate with master properly, so you should always take a look at the code you integrate into your branch.

Key points

- In the Feature Branch Workflow, you create a branch any time you want to work on a new feature.
- You should update your branch periodically with new changes in master.
- You must update your branch with master before pushing it up for review.
- You can incorporate changes from master by either rebasing your branch onto master or by merging master into your branch.
- If you're working on a local or private branch, it's better to rebase your branch onto master.
- Never rebase public branches; instead, merge master into the branch to prevent rewriting history.

That's all. In the next chapter, you'll learn about a popular Git branching model named Gitflow.

Chapter 10: Gitflow Workflow

By Jawwad Ahmad

Gitflow is a workflow that Vincent Driessen introduced in his 2010 blog post, A successful Git branching model, https://nvie.com/posts/a-successful-git-branching-model.

At its core, Gitflow is a specialized version of the branching workflow. It introduces a few different types of branches with well-defined rules on how code can flow between them.

Vincent posted a ten-year update titled "Note of reflection" on March 5th, 2020, at the beginning of his original blog post. In his note, he recommends that you should consider whether this is the right workflow for you.

He notes that Gitflow is great for versioned software, but a simpler approach might work better in today's age of continuous deployment. He ends his note with the words: "Decide for yourself."

In this chapter, you'll learn about the rules that make up Gitflow, as well as the reasons behind them. This will allow you to decide if Gitflow is right for you.

When to use Gitflow

Gitflow is a good fit if you're building software that's explicitly versioned, especially if you need to support multiple versions of your software at the same time. For example, you might release a 2.0 version of a desktop app that's a paid upgrade, but still want to continue releasing minor bug fix updates for the 1.0 version.

Gitflow is also a good fit if your project has a regular release cycle. Its release branch workflow allows you to test and stabilize your release while normal day-to-day development continues on your main development branch.

Managing larger projects is easier with Gitflow since it has a well-defined set of rules for the movement of code between branches.

Gitflow is less ideal in scenarios that favor a continuous deployment model, such as web development. In these situations, Gitflow's release workflow might add unnecessary extra overhead.

Chapter roadmap

In this chapter, you'll first get a quick introduction to the basic concepts in Gitflow. You'll learn about the different long-lived and short-lived branches and the rules for how to create and merge them.

You'll then install the git-flow extensions, which aren't necessary to use the Gitflow workflow itself, but which make it easier to adopt. The term **git-flow** will be used to refer to the extensions, and **Gitflow** will be used to refer to the workflow itself.

Once installed, you'll learn how to use the git-flow extension commands to create and merge the various types of branches Gitflow uses.

Types of Gitflow branches

Gitflow uses two long-lived branches: **master** and **develop** and three main *types* of short-lived branches: **feature**, **release** and **hotfix**. While you never delete long-lived branches, you delete short-lived branches once you merge them into a long-lived branch.

There are well-defined rules about how and when to create short-lived branches, as well as rules for how to merge them back into the long-lived branches. You'll learn about these rules in a bit. But first you'll learn about the purpose of the various branch types.

Long-lived branches

Git itself uses a single long-lived branch, which is the **master** branch. Gitflow introduces the concept of an additional long-lived *production* branch.

Instead of introducing a new name for this production branch, Gitflow repurposes the **master** branch for this role. This means that the master branch can now only contain code that's been released to production, or that will be released to production when it's merged to master.

Consequently, Gitflow adds a **develop** branch for the role that the master branch had previously played, i.e., for normal day-to-day development.

Code can only be added to, and moved between the long-lived branches using short-lived branches which you'll learn about next.

Short-lived branches

The three main types of short-lived branches are **feature**, **release** and **hotfix**.

Feature branches: Just as in the normal branching workflow, you do all your new development here. You create a **feature** branch when you add new functionality to your app, such as a new settings screen or improvements to the login flow.

Release branches: Use these to prepare and test code on the **develop** branch for a release to production. If you find any bugs, you fix them on the **release** branch. These branches are also good for tasks like updating release notes and versions.

Hotfix branches: These are used to fix bugs already in production. Use these as *quick release* branches, creating them when you need to deploy an urgent bug to production as soon as possible.

There's also a less commonly-used type of branch known as a **support** branch. You use these only when you need to support previously-released versions of your software.

Rules for creating and merging branches

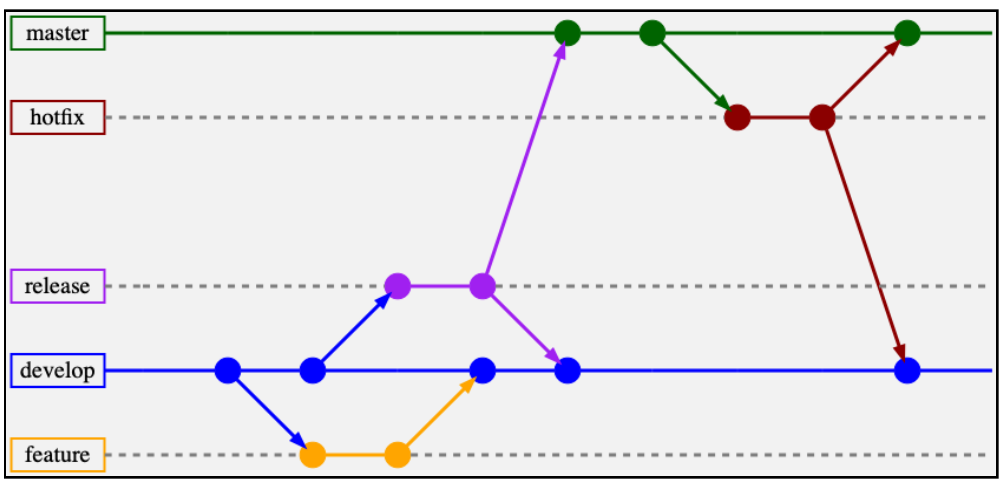
Here are the rules for creating and merging branches:

Feature branches are created only from **develop** and are only merged into **develop** since that's the branch you use for normal day-to-day development.

Release branches are created only from **develop** and are merged into both **master** and **develop**. The additional merge to **develop** ensures that any updates or bugfixes you commit on the **release** branch make their way back into **develop**.

Hotfix branches are created only from **master** and are merged into both **master** and **develop**. This ensures that bugs you fix in production are also fixed on develop.

Here's an image that displays an example of the branching and merging flow:



Gitflow's branching and merging flow for feature, release and hotfix branches

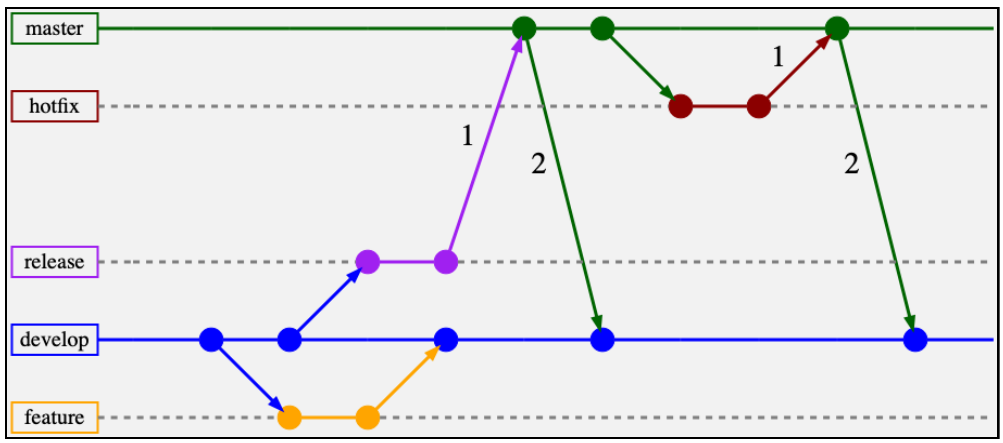
Solid lines represent long-lived branches, dashed lines represent short-lived branches and arrows represent the branch creation and merging flow.

Here are a few alternate ways of thinking about the branch creation and merging rules, which may help you grasp the flow better:

- The only branch that you can create from **master** is the **hotfix** branch since it's only meant to fix bugs in production. You can't create **hotfix** branches from **develop** because merging them would also deploy any additional features committed to **develop** since the last release.

- Any code merged to **master** that isn't already in **develop** needs to be added to **develop**. This is why, when you merge the **release** and **hotfix** branches to **master**, you also need to merge them into **develop**.
- An alternate way to update **develop** with code merged into **master** is to subsequently merge **master** into **develop**. So after you merge a **release** or **hotfix** branch to **master**, you then merge **master** into **develop**. Think of this as **back-merging** master into develop.

The flow of using this alternate approach of back-merging an updated **master** branch into **develop** looks like this:



An alternate flow for hotfix and release that back-merges master into develop

Both approaches are acceptable, but there's a slight benefit to the back-merging approach, which you'll learn about later.

Next you'll install the git-flow extensions and start playing with the various types of short-lived branches.

Installing git-flow

The git-flow extensions are a library of Git subcommands that make it easier to adopt the Gitflow workflow. For example, a single `git flow release finish` command will merge your release branch into master, tag the release, merge it back into develop and then finally delete the release branch.

Essentially, the git-flow extensions run sequences of Git commands that encode specific Gitflow workflows.

You have two options for installing the git-flow extensions. The first is from the creator of Gitflow, Vincent Driessen; you can find it at github.com/nvie/gitflow.

Unfortunately, Vincent Driessen no longer maintains this project, so installing this version isn't recommended. Its last release was in 2011, and you'll install that outdated version if you simply run `brew install git-flow` on macOS.

The recommended version to install is the AVH version by Peter van der Does, which is available at github.com/petervanderdoes/gitflow-avh, and is available on Homebrew as **git-flow-avh**.

> **Note**: If you've already installed the default `git-flow`, you can uninstall it via `brew uninstall git-flow`. Alternatively, if you've installed both, you can overwrite the older version with `brew link --overwrite git-flow-avh`.

If you need to install Homebrew, see https://brew.sh. To install git-flow-avh on Windows, see github.com/petervanderdoes/gitflow-avh/wiki/Installing-on-Windows.

Run the following to install the AVH version of git-flow:

```
brew install git-flow-avh
```

To verify the version you've installed, run the following:

```
git flow version
```

You should see **1.12.3 (AVH Edition)**. If you see **0.4.1** instead, you have the original, unmaintained version installed. See the note above on how to uninstall it.

If you have trouble installing git-flow, you can also use the manual Git commands that git-flow would run, which will also be provided in the chapter.

Next, you'll configure the starter project to use git-flow.

Initializing git-flow

Unzip **repos.zip** from the **starter** folder for this chapter. You'll only be working within the **alex/checklists** repository, so the **beth** and **chad** directories from previous chapters aren't included.

You'll see the following unzipped directories within **starter**:

```
starter
└── repos
    ├── alex
    │   └── checklists
    └── checklists.git
```

As in the previous chapter, **checklists.git** is the remote origin for the **alex/checklists** repository.

Open a terminal window and **cd** to the **alex/checklists** directory in **starter/repos**.

```
cd path/to/starter/repos/alex/checklists
```

Before you start using git-flow, you'll need to initialize a few configuration settings.

Run the following command to initialize a git-flow configuration for this repository:

```
git flow init
```

Press **Enter** to accept each of the defaults. However, for the **Version tag prefix?** question, use a lowercase **v** since this is a very common convention.

You'll see the following:

```
Which branch should be used for bringing forth production
releases?
   - master
Branch name for production releases: [master]
Branch name for "next release" development: [develop]

How to name your supporting branch prefixes?
Feature branches? [feature/]
Bugfix branches? [bugfix/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? [] v
Hooks and filters directory? [...]
```

If you accidentally miss the question, you can use `git flow init -f` to re-initialize it. Alternatively, you can manually edit the **.git/config** file by changing the **versiontag** = line to **versiontag** = **v** and saving it.

Gitflow uses branch prefixes to differentiate between different types of branches. These prefixes will automatically be added when using the various git-flow **start** commands to create the specified type of branch.

> **Note**: You may notice there is also a **bugfix** type of branch. This type of branch wasn't present in Vincent's implementation of git-flow or in his original workflow. This was added in git-flow-avh for fixing bugs on **develop**. It can be thought of as equivalent to a **feature** branch, but with an alternate prefix that more clearly indicates that the branch is for a bugfix instead of a feature.

Running `git flow init` will also create a **develop** branch from the **master** branch, if one doesn't exist already.

You're now ready to add a new feature using Gitflow which you'll do next.

Creating and merging a feature branch

Gitflow uses feature branches for work on new features, just as the branching workflow does. Since the day-to-day development branch in Gitflow is now **develop** instead of **master**, you create feature branches from **develop** and merge them back into **develop** when you're finished. You cannot create feature branches from **master** or any of the other short-lived branches.

Make sure you're still in the **checklists** folder and create a **feature** branch by running the following command:

```
git flow feature start update-h1-color
```

This is equivalent to running the following command from the develop branch:

```
git checkout -b feature/update-h1-color  # equivalent to above
```

You'll see the following **Summary of actions** that confirms what the command did:

```
Summary of actions:
- A new branch 'feature/update-h1-color' was created, based on
'develop'
- You are now on branch 'feature/update-h1-color'
```

Now, you'll make a minor change and commit this feature. Open **style.css** and, on the second line, change the color of the **h1** tag from **navy** to **blue**.

```
 h1 {
-    color: navy;
+    color: blue;
 }
```

Run the following command to commit the change:

```
git commit -am "Updated h1 color from navy to blue"
```

Now that you've completed the feature, you'll merge the **feature/update-h1-color** branch back into **develop** using git-flow.

While the previous `git flow feature start` command didn't save much typing over the actual checkout command, the **finish** version of this command does a bit more.

In a single command, it performed the following three actions:

```
1. Checkout the develop branch            # git checkout develop
2. Merge the feature branch  # git merge feature/update-h1-color
3. Delete the branch     # git branch -d feature/update-h1-color
```

Run the following command to finish the feature:

```
git flow feature finish
```

You'll see the following **Summary of actions** for it:

```
Summary of actions:
- The feature branch 'feature/update-h1-color' was merged into
'develop'
- Feature branch 'feature/update-h1-color' has been locally
deleted
- You are now on branch 'develop'
```

You saved some typing, and that's nice. But the individual commands aren't difficult to remember since they're common used in Git. Why bother, then?

The main benefit is that the git-flow extensions automatically enforce Gitflow's rules for you and prevent mistakes. For example, they'll prevent you from accidentally creating a feature branch from master or accidentally merging a completed feature branch into master.

When you're working on release and hotfix branches, the `git flow finish` commands will save you even more typing (and remembering) since you need to merge them into both develop and master. This is the perfect segue to learn about them in the next section!

Creating and merging a release branch

Release branches are where you prepare code for an upcoming release. They let you run tests and implement fixes while the day-to-day development continues on the development branch.

Since release branches hold release code under development, you create them from the **develop** branch and merge them into **master**. You merge them back into **develop** so that any additional bug fixes and updates committed to the release branch make it back into the **develop** branch.

You generally name release branches using a version number, then use that same version number to tag the release.

The AVH version of git-flow includes a number of improvements over the original, including a `--showcommands` option, which shows you the Git commands executed while running a git-flow command.

Run the following to create a new release branch and to see the command it uses:

```
git flow release start 1.0.0 --showcommands
```

Ignore the first `git config --local ...` line, which git-flow uses internally. On the second line, you'll see:

```
git checkout -b release/1.0.0 develop
```

The extra `develop` argument listed at the end is the start point (or the base) of the `release/1.0.0` branch. It's equivalent to running `git checkout develop && git checkout -b release/1.0.0`.

Now, you'll make an additional update to the release branch. There aren't any bug fixes to make, but you'll add a new VERSION file.

Run the following to add the new VERSION file and commit it:

```
echo '1.0.0' > VERSION
git add VERSION
git commit -m "Adding VERSION file for initial release"
```

Now the release branch is complete. It's time to merge into master to deploy it.

You also want to bring the commit that adds the VERSION file back into the develop branch. Recall that there are two ways to accomplish this: You can either merge `release/1.0.0` into `develop` or you can back-merge `master` into `develop`.

For your next step, you'll use the back-merge approach, which the AVH version of `git-flow` uses by default.

You'll do this by typing a single command, but if you were to manually perform these actions, the commands would be:

```
# Merge release into master
git checkout master
git merge --no-ff release/1.0.0

# Tag the release
git tag -a v1.0.0

# Merge master back to develop
git checkout develop
git merge --no-ff master

# Delete the branch
git branch -d release 1.0.0
```

> **Note**: To merge master back into develop, git-flow uses a reference to the tag instead of using master. The result is the same since both the tag and master resolve to the latest commit on master. However, using the tag as a reference results in a more specific commit message, since it includes the version.

With git-flow, run the following command:

```
git flow release finish --showcommands
```

Now, you'll need to save three commit messages. Type **:wq** to accept the default message for merging `release/1.0.0` into `master`. Next, for the tag message, type **Tag for 1.0.0 release** and save it with **:wq**.

Before accepting the last message, note that it says: **Merge tag 'v1.0.0' into develop**. If you had used `master`, it would have said: **Merge branch 'master' into develop**. It's nice to see the specific version that was merged in the merge message.

Type **:wq** once more to accept the default message for the merge.

Here's the last part of the output, with the third line confirming the back-merge of the tag:

```
Summary of actions:
- Release branch 'release/1.0.0' has been merged into 'master'
- The release was tagged 'v1.0.0'
- Release tag 'v1.0.0' has been back-merged into 'develop'
- Release branch 'release/1.0.0' has been locally deleted
- You are now on branch 'develop'
```

This is also reflected in the commands you ran:

```
git checkout master
...
git merge --no-ff release/1.0.0
...
git tag -a v1.0.0
git checkout develop
...
git merge --no-ff v1.0.0  # instead of: git merge --no-ff master
...
git branch -d release/1.0.0
```

As a final verification, you'll check that both master and the v1.0.0 tag point to the same commit. Run the following commands to get the latest commit for the tag and for master:

```
git -P log --oneline -1 v1.0.0
git -P log --oneline -1 master
```

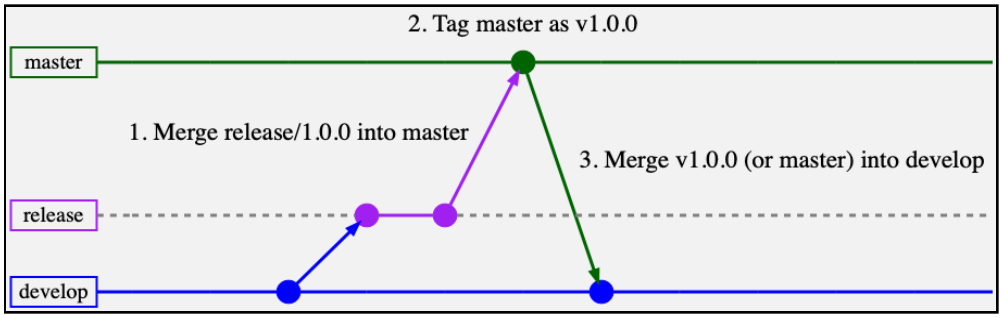
The `-P` or `--no-pager` option disables the pager, so you don't need to press **q** to quit it, and prints the output directly to the console. You'll see the same message and commit hash for both, something similar to:

```
85c2e4e (tag: v1.0.0, master) Merge branch 'release/1.0.0'
```

Next, you'll learn about some of the nitty-gritty details of the differences between back-merging master versus merging the release branch.

Back-merging master versus merging release

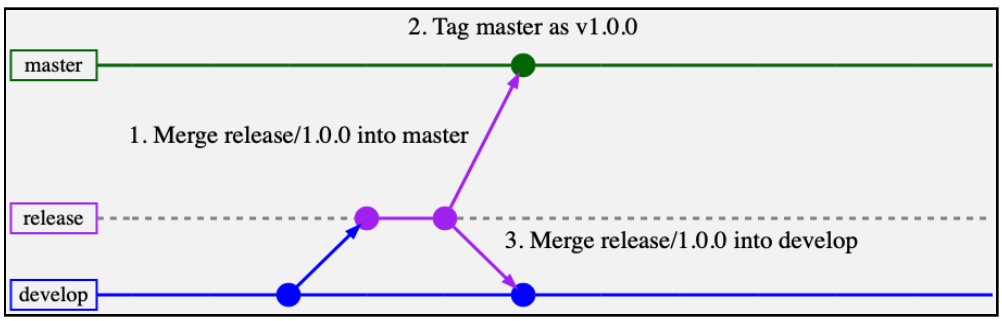
The following image shows the steps that took place when you back-merged **master** into **develop**:



Back-merging master into develop

When you merge master into develop, you really merge *everything* on master that wasn't already in develop. If you have additional commits on master that you *never* want to merge into develop — and you really shouldn't — then you can't use this strategy.

But in case you *really* need to, the AVH version of `git-flow` provides a non-back-merge fallback option using the **`--nobackmerge`** or **`-b`** flag. In that case, only the final merge step will be different, which you can see in the following image:



Merging the release branch into develop

One additional point to note is that with the `--nobackmerge` option, the tagged commit on master is not an ancestor of the commit that's merged into develop. This can create issues with commands like `git describe`, which finds the most recent tag reachable from a commit.

With the back-merge option, the commit tagged with `v1.0.0` is a parent of the merged commit on develop. This means `git describe` will be able to report the most recent tag as being `v1.0.0`, even when run from the develop branch.

With the `--nobackmerge` option, however, the tagged commit is not an ancestor of the merged commit on develop. `git describe` would only be able to find the `v1.0.0` tag when you run it from the **master** branch, but not from the **develop** branch.

In the next section, you'll finish a hotfix branch using `--nobackmerge`, to see this difference with the output of `git describe`.

Creating and merging a hotfix branch

You use **hotfix** branches to fix bugs in production, so you must create them from the **master** branch.

Even though the bug is also likely to be in the develop branch, you don't want to branch from develop to fix it since deploying that would prematurely deploy any additional code committed to develop since the last release.

You need to merge **hotfix** branches to both **master** and **develop** (or via back-merge from master). Like release branches, you name them with a version number, which git-flow will also use to tag the merge to master.

In a sense, hotfix branches are almost exactly like release branches, except that they're created from **master** instead of **develop**.

So it turns out that changing the color from **navy** to **blue** was a mistake and should have instead been changed to **midnightblue**. This is an urgent fix that needs to be deployed immediately and cannot wait to be included in the next release.

Run the following git-flow command to start a hotfix branch:

```
git flow hotfix start 1.0.1 --showcommands
```

You'll see that the command that actually executes is simply the following:

```
git checkout -b hotfix/1.0.1 master
```

Note that you didn't have to make sure you were on a specific branch before running the command. Are you starting to see the advantages of Gitflow? :]

Now, open **style.css** and change the color of the **h1** tag from **blue** to **midnightblue**:

```
 h1 {
-    color: blue;
+    color: midnightblue;
 }
```

Run the following command to commit the change:

```
git commit -am "Updated h1 color from blue to midnightblue"
```

Additionally, update the version number in the **VERSION** file to **1.0.1**. You can either edit the file manually or run the following command:

```
echo '1.0.1' > VERSION
```

Then commit the version update:

```
git commit -am "Updated VERSION to 1.0.1"
```

Now, you could run `git flow hotfix finish`, or simply `git flow finish`, to merge the hotfix branch you're on. However, this time you'll use the classic behavior of merging the **hotfix** branch into **develop** by using the `--nobackmerge` option.

```
git flow finish --nobackmerge --showcommands
```

Again, type **:wq** to accept the initial message for the merge to master and add **Tag for 1.0.1 release** for the tag message. Type **:wq** and then type **:wq** one final time to accept the message for the merge of the hotfix/1.0.1 branch to develop.

You'll see the following **Summary of actions**:

```
Summary of actions:
- Hotfix branch 'hotfix/1.0.1' has been merged into 'master'
- The hotfix was tagged 'v1.0.1'
- Hotfix branch 'hotfix/1.0.1' has been merged into 'develop'
- Hotfix branch 'hotfix/1.0.1' has been locally deleted
- You are now on branch 'develop'
```

Without using the `--nobackmerge` option, the third line would have said:

```
- Hotfix tag 'v1.0.1' has been back-merged into 'develop'
```

And you'll see this reflected in the commands as well:

```
git checkout master
...
git merge --no-ff hotfix/1.0.1
...
git tag -a v1.0.1
git checkout develop
...
git merge --no-ff hotfix/v1.0.1  # not: git merge --no-ff v1.0.1
...
git branch -d hotfix/1.0.1
```

Congratulations! You've used git-flow to adopt the Gitflow workflow and created and merged a feature branch, a release branch and a hotfix branch!

Now you'll cover some final details about how `git describe` won't be accurate when you use the classic approach using `--nobackmerge`. You'll also learn a bit about using help with `git-flow`, and then you'll be done with this chapter!

Using git describe

The `git describe` command shows you the most recent tag that's accessible from a commit. If the tag is on the current commit, `git describe` will show the tag itself. On the other hand, if the tag is on one of the ancestors, it will also show the number of additional commits and the commit hash in the following format:

```
{tag}-{number-of-additional-commits-from-tag}-g{commit hash}
```

Run **`git describe develop`** and you'll see something like the following, just with a different hash:

```
v1.0.0-4-g827ddd8
```

Now run **`git log --oneline -5 develop`** to get the five latest commits on develop:

```
827ddd8 (HEAD -> develop) Merge branch 'hotfix/1.0.1' into de...
b4990c4 Updated VERSION to 1.0.1
20df16b Updated h1 color from blue to midnightblue
404639f Merge tag 'v1.0.0' into develop
2249859 (tag: v1.0.0) Merge branch 'release/1.0.0'
```

In the output of `git describe`, **v1.0.0** is the tag, **4** is the number of additional commits after the tag, and **827ddd8** is the commit hash of the commit you ran `git describe` on — that is, **develop**.

This is misleading since the current version is 1.0.1 and the `git log` command above shows that the commit that updates the **VERSION** file to 1.0.1 is included in the develop branch.

The reason it even shows v1.0.0 is because you back-merged the v1.0.0 tag from master to develop, so develop can access it. If it hadn't been back-merged, it would just show an error.

You can replicate the error by running `git describe origin/master` since there are no tags accessible from the `origin/master` commit:

```
fatal: No tags can describe 'c0623652f3f7979f664918689fca42e9...
```

Now, run `git describe master` and you'll see the following:

```
v1.0.1
```

When the reference you used points to the same commit as the tag, running `git describe` prints the tag without the additional info.

> **Note**: You can suppress the additional info by using the **`--abbrev=0`** option. In that case, running `git describe develop --abbrev=0` would just show `v1.0.0`.

So one benefit of using the back-merge strategy to merge **master** into **develop** is that any tags on the **master** branch will also be accessible from the **develop** branch. That means you can run a `git describe` while on the **develop** branch to show the tag for the latest release.

Thus far you've only used the most commonly used commands from `git-flow`. Next you'll learn how to explore the various commands `git-flow` provides as well as the different options that can be used with each command.

Exploring the git-flow library

The `git-flow` library includes a few additional commands that can be helpful, such as `delete` for deleting a type of branch or `publish` for pushing it to the remote.

To see all subcommands run **git flow help**. You'll see the following:

```
Available subcommands are:
   init      Initialize a new git repo with support for the b...
   feature   Manage your feature branches.
   bugfix    Manage your bugfix branches.
   release   Manage your release branches.
   hotfix    Manage your hotfix branches.
   support   Manage your support branches.
   version   Shows version information.
   config    Manage your git-flow configuration.
   log       Show log deviating from base branch.

Try 'git flow <subcommand> help' for details.
```

Next use **help** with a specific type of subcommand. Run **`git flow release help`** to see the types of subcommands available for release branches:

```
$ git flow release help
usage: git flow release [list]
   or: git flow release start
   or: git flow release finish
   or: git flow release publish
   or: git flow release track
   or: git flow release delete

    Manage your release branches.

    For more specific help type the command followed by --help
```

Next use `--help` or `-h` with the specific type of sub-subcommand to see a description of what it does. You need the dashes otherwise it would use **help** as the branch name. For example, run **git flow release publish –help** and you'll see the following:

```
$ git flow release publish --help
usage: git flow release publish [-h] <name>

    Publish the release branch <name> on origin

    -h, --help            Show this help
    --showcommands        Show git commands while executing them
```

And you're done! You've not only learned about the Gitflow workflow, but also how to use and explore the various commands in the `git-flow` library.

Key points

- The master branch serves as the production branch.
- The develop branch is for normal day-to-day development.
- Feature branches are used for new feature development.
- Release branches are used to test, stabilize and deploy a release to production.
- Use hotfix branches to fix bugs you've already released to production.
- You create feature branches from develop and merge them to develop.
- You create release branches from develop and merge them to master and develop.
- You create hotfix branches from master and merge them to master and develop.
- Install the newer AVH version of git-flow with `brew install git-flow-avh`.

That's Gitflow for you! In the next chapter, you'll learn about the Forking Workflow.

Chapter 11: Forking Workflow

By Jawwad Ahmad

In this chapter, you'll learn all about the **Forking Workflow**. You use the Forking Workflow when you want to contribute to a project to which you only have read-only access. It's mainly used when contributing to open source projects, but you can also use it with private repositories.

When you don't have push access to a project, you'll need to push your changes to a public copy of the project. This personal, public copy of the project is called a **fork**. The original, or source, repository is conventionally referred to as the **upstream** repository.

To request that the upstream repository merge a branch from your fork, you then create a pull request with the branch that has your changes.

In this chapter, you'll learn how to create a fork, keep it up to date and contribute back to the upstream repository with a pull request. You'll also learn how to merge in open pull requests and branches from other forks.

Getting started

As a software developer, you've likely heard of FizzBuzz. In case you haven't, it's a programming task where, for numbers from 1 to 100, you print either the number itself or a word. For multiples of three, you print **Fizz**, for multiples of five you print **Buzz**, and for multiples of both three and five, you print **FizzBuzz**.

For example, here are the first fifteen items:

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Buzz
13
14
FizzBuzz
```

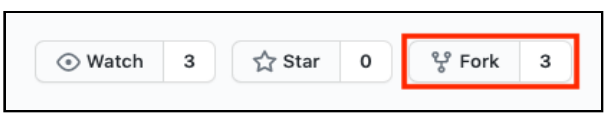
For this tutorial, you'll create a fork of a repository that implements FizzBuzz. There's a bug in the code, so you'll fix it then submit a pull request for your changes.

In a browser, open the following URL for the repository:

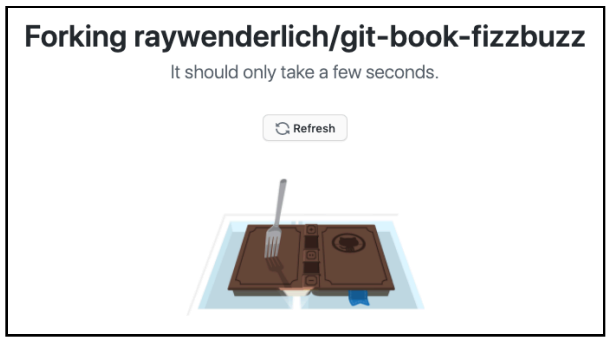
https://github.com/raywenderlich/git-book-fizzbuzz

> **Note**: The **git-book-fizzbuzz** repository uses **main** instead of **master** as the default branch.

Now, click the **Fork** button at the top-right corner of the page:

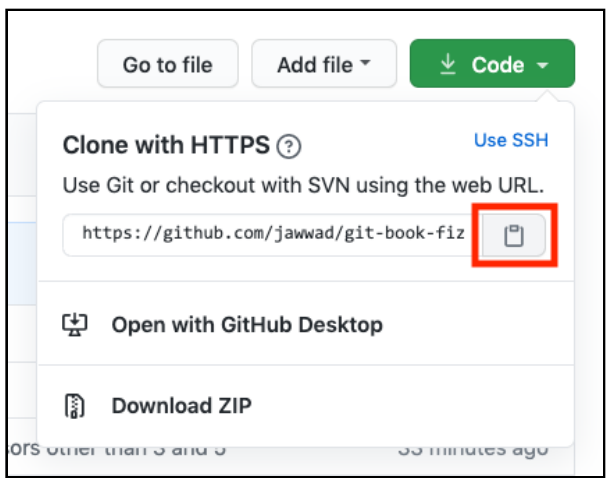


You'll see a progress screen indicating that GitHub is creating your fork:



Once GitHub finishes, it will redirect you to the newly-created fork, under your personal GitHub account. You'll see the URL of the page change to `https://github.com/{your-github-username}/git-book-fizzbuzz`.

Next, click on the **Code** button drop-down, then click the clipboard icon to copy the repository's URL:



Now, open Terminal and **cd** to the **starter** folder of this project:

```
cd {your/path/to}/forking-workflow/projects/starter
```

Next, type **git clone**, add a **space** and **paste** the copied repository URL.

You should have the following, with your GitHub username in place of `{username}`:

```
git clone https://github.com/{username}/git-book-fizzbuzz.git
```

Press **Enter** to execute the command. You'll see the following, confirming the clone:

```
Cloning into 'git-book-fizzbuzz'...
...
Resolving deltas: 100% (14/14), done.
```

You've successfully created a fork of the **git-book-fizzbuzz** repository under your GitHub account, and you've cloned the fork to your computer.

Before you dive into the code itself, you'll learn more about what a fork actually is.

A fork is simply a clone

In the previous section, you created a fork and then cloned it. So if a fork is just a clone, then you cloned your clone!

More specifically, a fork is a public, server-side clone of the project under your own account, which means you can push changes to it.

Forking is a workflow and not part of Git itself. There's no *git fork* command that will create a fork of a repository. When you create a fork in GitHub, it creates a server-side clone of the project under your account and enables certain features available only to forks, like the ability to create pull requests.

As far as Git is concerned, there's no difference between the upstream repository, your fork of the repository and the local clone of your fork.

To help you internalize this, you'll create a fourth local clone from upstream. This will show you how an upstream clone differs from a clone of the fork.

Make sure you're still in the **starter** folder and run the following command, all on a single line, to clone the upstream repository as **upstream-git-book-fizzbuzz**:

```
git clone https://github.com/raywenderlich/git-book-fizzbuzz.git
upstream-git-book-fizzbuzz
```

Now, run the following to compare the clone of your fork with the clone of the upstream, using a recursive diff:

```
diff -r git-book-fizzbuzz upstream-git-book-fizzbuzz -x logs -u
```

> **Note**: `-x logs` is short for `--exclude=logs`, which ignores timestamps in the logs directory. `-u` is short for `--unified`, which shows the diff in unified format — that is, with a - and + instead of < and >.

In the results, you'll see the following, which shows that the only difference is the remote origin URL:

```
...
 [remote "origin"]
-   url = https://github.com/{username}/git-book-fizzbuzz.git
+   url = https://github.com/raywenderlich/git-book-fizzbuzz.git
...
Binary files git-book-fizzbuzz/.git/index and upstream-git-book-
fizzbuzz/.git/index differ
```

The final line, telling you the `.git/index` files of the two branches are different, is inconsequential since this is a binary Git uses to keep track of staged changes.

You can even update the clone of the upstream repository to point to your fork by updating its origin URL.

Run the following, replacing `{username}` with your GitHub username:

```
cd upstream-git-book-fizzbuzz
git remote set-url origin https://github.com/{username}/git-
book-fizzbuzz.git
```

Run `cd ..` to go back to the starter folder, then execute the diff command again:

```
cd ..
diff -r git-book-fizzbuzz upstream-git-book-fizzbuzz -x logs -u
```

Now, you'll no longer see any differences other than the binary `.git/index` file.

With this, you see that there's absolutely no difference between a clone of your fork and a clone of the upstream repository other than the **origin** URL.

Now, delete the **upstream-git-book-fizzbuzz** clone since you no longer need it:

```
rm -rf upstream-git-book-fizzbuzz
```

Next, you'll explore the code, and then play a quick game of *find-the-bug*!

Exploring the code

Change to **git-book-fizzbuzz** and open **fizzbuzz.py** in an editor.

```
cd git-book-fizzbuzz
open fizzbuzz.py # or open manually in an editor of your choice
```

Start reading from the end of the file. The following lines mean that the `main()` method is executed when running this as a script:

```
if __name__ == "__main__":
    main()
```

And `main()` simply executes `fizzbuzz()`:

```
def main():
    fizzbuzz()
```

And above that, `fizzbuzz()` executes `fizzbuzz_for_num(n)` for each number `n` from 1 to 101 exclusive, which really means from 1 to 100.

```
def fizzbuzz():
    for n in range(1, 101):
        value = fizzbuzz_for_num(n)
        print(value)
```

Finally, `fizzbuzz_for_num(...)` contains the main logic that determines which string to return for a given number. It additionally allows using words other than **Fizz** and **Buzz**, and even allows you to use divisors other than **3** and **5**:

```
def fizzbuzz_for_num(
    n,
    fizz_divisor=3,
    fizz_word="Fizz",
    buzz_divisor=5,
    buzz_word="Buzz",
):
    should_fizz = n % 3 == 0
    should_buzz = n % 5 == 0
    if should_fizz and should_buzz:
        return fizz_word + buzz_word
    elif should_fizz:
        return fizz_word
    elif should_buzz:
        return buzz_word
    else:
        return str(n)
```

There is, however, a bug in the code above. See if you can spot it. The bug only manifests itself when using divisors other than 3 and 5.

If you haven't spotted it already, you certainly will when you take a look at the latest commit next.

Run **git show** and, in **fizzbuzz.py**, you'll see the following change:

```
 def fizzbuzz_for_num(
     n,
+    fizz_divisor=3,
     fizz_word="Fizz",
+    buzz_divisor=5,
     buzz_word="Buzz",
 ):
```

The commit added the `fizz_divisor` and `buzz_divisor` parameters to the method signature, but the code in the method itself was never updated to use the new parameters! Next, you'll fix this bug and open a pull request for it.

The second thing to notice in the `git show` output is that the commit also added tests to **test_fizzbuzz.py** in the **`test_with_alternate_divisors`** method.

Run the tests with the following command:

```
python test_fizzbuzz.py
```

You'll see the following three failures in the output:

```
...
    self.assertEqual(fizzbuzz_for_num(7, fizz_divisor=7,
buzz_divisor=11), "Fizz")
AssertionError: '7' != 'Fizz'
...
    self.assertEqual(fizzbuzz_for_num(11, fizz_divisor=7,
buzz_divisor=11), "Buzz")
AssertionError: '11' != 'Buzz'
...
    self.assertEqual(fizzbuzz_for_num(77, fizz_divisor=7,
buzz_divisor=11), "FizzBuzz")
AssertionError: '77' != 'FizzBuzz'
...
Ran 6 tests in 0.001s

FAILED (failures=3)
```

> **Note**: If you see an error that says: AttributeError: 'TestFizzBuzz' object has no attribute 'subTest', this means your default Python is Python 2. Try running `python3 test_fizzbuzz.py`; if that doesn't work, you can skip this step.

The output is saying that given `fizz_divisor=7`, and `buzz_divisor=11`:

1. For number 7, it should have returned **Fizz**, but it returned 7 instead.
2. For number 11, it returned 11 when it should have returned **Buzz**.
3. For number 77, it should have returned **FizzBuzz**, but it returned 77.

It's nice that the commit also included tests, but it looks like someone forgot to run them to verify that their code *actually* worked. :[

Though it's certainly helpful that there are tests so you can verify that your upcoming fix will work! :]

Fixing the custom divisors bug

Create a new branch for your fix named **fix-divisors-bug**:

```
git checkout -b fix-divisors-bug
```

Switch back to the editor where you have **fizzbuzz.py** open. On line 11, replace **3** with **fizz_divisor** and on line 12 replace **5** with **buzz_divisor**:

```
11)    should_fizz = n % 3 == 0  # replace 3 with fizz_divisor
12)    should_buzz = n % 5 == 0  # replace 5 with buzz_divisor
```

After saving the file, run `git diff` and confirm that you see only the following changes:

```
...
-    should_fizz = n % 3 == 0
-    should_buzz = n % 5 == 0
+    should_fizz = n % fizz_divisor == 0
+    should_buzz = n % buzz_divisor == 0
     if should_fizz and should_buzz:
...
```

And now for the moment of truth! Execute the tests with the following command:

```
python test_fizzbuzz.py
```

This time, all the tests should pass! :]

```
test_divisible_by_both (__main__.TestFizzBuzz) ... ok
test_divisible_by_five (__main__.TestFizzBuzz) ... ok
test_divisible_by_none (__main__.TestFizzBuzz) ... ok
test_divisible_by_three (__main__.TestFizzBuzz) ... ok
test_with_alternate_divisors (__main__.TestFizzBuzz) ... ok
test_with_alternate_words (__main__.TestFizzBuzz) ... ok

----------------------------------------------------------------------
Ran 6 tests in 0.001s

OK
```

Now, you can commit your changes.

It's a good idea for the commit message for your pull request to go into details about why you made the changes, how you fixed the bug, and how you tested your fix.

However, typing out long paragraphs in a tutorial is no fun. Instead, you'll use the message in **commit_message.txt**, which is in the **starter** folder.

Though you don't need to open the file manually. Run the following command to commit your changes with the message in **commit_message.txt**:

```
git commit -a --file=../commit_message.txt
```

Now, run **git show** to verify that the previous command added the message:

```
Fix bug in which alternate divisors were not used

This commit updates the code in the fizzbuzz_for_num method to
start using the fizz_divisor and buzz_divisor parameters that
were added to the method signature in a previous commit

Verified the fix by running existing tests in test_fizzbuzz.py
which were previously failing and now are all passing
```

Next, you'll push the **fix-divisors-bug** branch to your fork so you can open a pull request with it.

Opening a pull request

Run the following to push the current branch to your fork:

```
git push -u origin head
```

Specifying `head` tells Git to push the current branch, so the above is shorthand for:

```
git push --set-upstream origin fix-divisors-bug  # same as above
```

Now that the branch is available in your fork, there are a few different ways to reach the pull request creation page. The following are three ways that you can use:

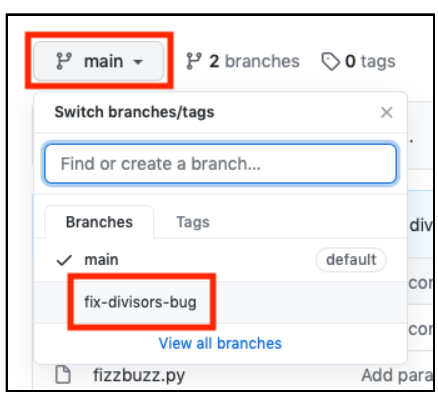
1. If you see a banner similar to the following appear on the GitHub page for your fork, you can click on the **Compare & pull request** button:

2. If you look at the output of the previous `git push` command you should see the following lines within the section prefixed with `remote:` that say:

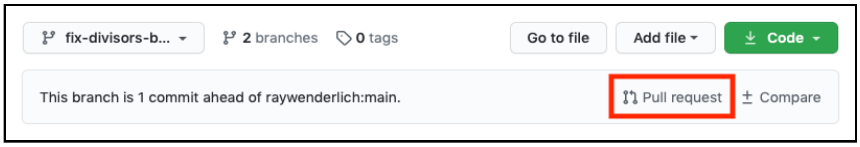
```
...
remote: Create a pull request for 'fix-divisors-bug' on GitHub
by visiting:
remote:      https://github.com/{username}/git-book-fizzbuzz/
pull/new/fix-divisors-bug
...
```

You can open the URL listed above to get to the pull request creation page.

3. On the page for your fork, click the **branches** drop-down and select the **fix-divisors-bug** branch:



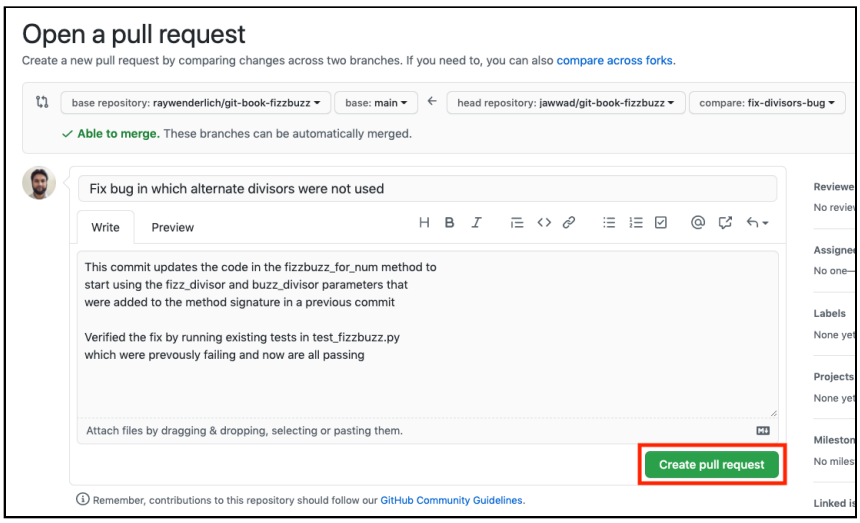
Then click on the **Pull request** button:



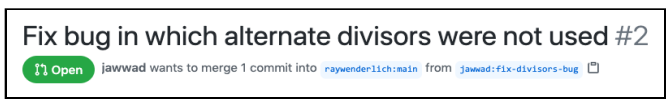
Each of these three methods will take you to the same **Open a pull request** page.

GitHub helpfully uses the first line of the commit message as the title of the pull request and the remaining lines as the body.

Finally, click on the **Create pull request** button to finish creating the pull request:



You've now created your pull request:



At this point, you'd normally just sit back, relax and wait for the maintainer of the upstream repository to merge your pull request. However, in this case, you still have the rest of the chapter to finish!

Although your pull request is amazing, I have a feeling that the maintainer of the upstream repository won't merge it, since this would change the tutorial for others. But feel free to leave it open since it lets me know you've read this chapter!

Next, you'll learn how to keep your fork up to date with any additional changes pushed to the **main** branch of the upstream repository.

Rewinding your main branch

Unfortunately, there won't be any updates to the upstream repository from the time that you cloned (or perhaps ever!), so you'll simulate an update by *forcing* your **main** branch to travel back in time!

Go back to your fork in GitHub, since creating the pull request would have taken you to the upstream **raywenderlich/git-book-fizzbuzz** repository.

First, note where it says: **This branch is even with raywenderlich:main.**

Now, run the following commands in Terminal to switch to your **main** branch and reset it back by two commits:

```
git checkout main
git reset head~2 --hard
```

You'll receive confirmation that the branch has been reset:

```
HEAD is now at 27e6f9a Move the "Fizz" and "Buzz" strings int...
```

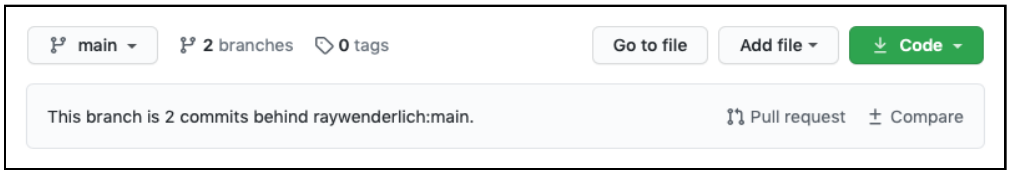
You also want to push this change to your fork. To do this, you'll do something that you were told never, ever to do... you'll force push the main branch! In this case, it's ok to do this since no one else would really be using your fork's main branch.

```
git push -f origin main
```

> **Note**: Just running `git push -f` would have accomplished the same thing. However, it's good practice to always specify the branch that you're force pushing so that you don't accidentally push the wrong branch.

Time travel complete! Now you can pretend that the upstream repository has two new commits since you forked the repository.

Refresh the page in GitHub and you'll see that it now says: **This branch is 2 commits behind raywenderlich:main.**



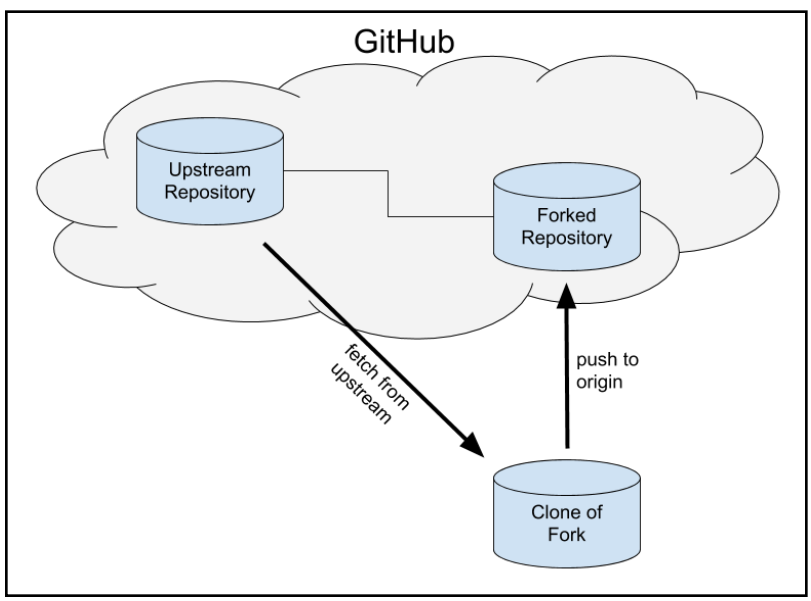
Next, you'll fetch those additional commits from the upstream remote.

Adding upstream and fetching updates

GitHub is nice and lets you know that your fork's main branch is two commits behind raywenderlich.com's main branch. But it doesn't actually give you a server-side option of updating your branch directly from upstream. Clicking a button would be too easy, right? :]

So you'll sync your fork with the upstream changes by using a triangular workflow. That is, you'll pull changes from one repository (upstream), then push those changes to another (your fork).

The following image represents this flow:



Run the following command to add the **upstream** remote:

```
git remote add upstream https://github.com/raywenderlich/git-
book-fizzbuzz.git
```

Now run `git remote -v` to list the remotes. You'll see the following:

```
origin https://github.com/{username}/git-book-fizzbuzz.git
(fetch)
origin https://github.com/{username}/git-book-fizzbuzz.git
(push)
upstream https://github.com/raywenderlich/git-book-fizzbuzz.git
(fetch)
upstream https://github.com/raywenderlich/git-book-fizzbuzz.git
(push)
```

Next, run the following to fetch updates from the **upstream** remote:

```
git fetch upstream
```

Since you added **upstream** as a remote, running `fetch` created a *remote tracking branch* named **upstream/main** that will update any time you run `git fetch upstream`:

```
From https://github.com/raywenderlich/git-book-fizzbuzz
 * [new branch]      main       -> upstream/main
```

Now run **git log –oneline –all** and you'll see that **upstream/main** is two commits ahead of **main** and **origin/main**:

```
d1dcc72 (origin/fix-divisors-bug, fix-divisors-bug) Fix bug i...
85ca623 (upstream/main) Add parameters to allow using divisor...
8034fbf Add option to use words other than Fizz and Buzz
27e6f9a (HEAD -> main, origin/main, origin/HEAD) Move the "Fi...
...
```

Now, merge **upstream/main** into **main** by running the following:

```
git merge upstream/main
```

Finally, push the updated branch to your fork:

```
git push
```

For your last step, refresh the GitHub page for your fork and you'll see that it once again says: **This branch is even with raywenderlich:main**.

Congratulations! You've updated your fork's **main** branch with the two additional commits from the upstream's **main** branch.

Fetching changes from other forks

You may occasionally want to merge feature branches from other forks into your fork. Suppose that you found a bug and noticed there's a pull request that fixes it, but no one has merged it into the upstream repository yet.

In this case, you can merge the branch from the pull request into your fork. It's not recommended that you merge anything other than **upstream/main** into your fork's **main** branch since yours should always mirror the upstream's **main** branch.

Run the following command to create a new **development** branch and merge your **fix-divisors-bug** branch into it:

```
git checkout -b development
git merge fix-divisors-bug
```

If you add an additional remote, fetching branches from that repository becomes easy. Running `git fetch remotename` will fetch all the remote branches and create remote tracking branches in the format **remotename/branchname**.

If you want to fetch a single branch from a different fork, adding the fork as an additional remote is overkill. You'd normally add remotes for forks that you want to fetch from more than once.

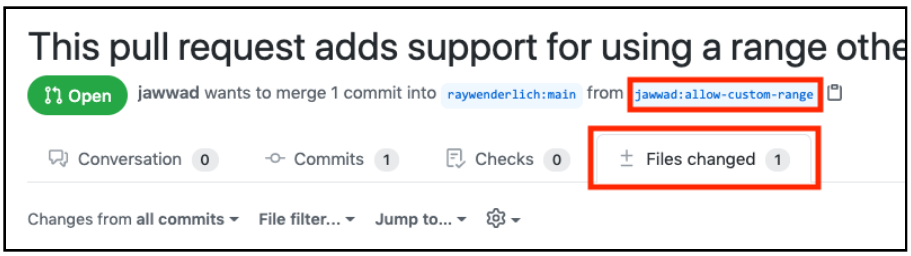
The feature branch you'll fetch already has a pull request open for it. It's for a minor feature that adds the ability to have **fizzbuzz.py** print a custom range instead of always using 1 to 100.

Navigate to the following page to see the pull request:

https://github.com/raywenderlich/git-book-fizzbuzz/pull/3

It looks like some user named **jawwad** opened it. The name sounds familiar but I can't quite place where I've heard it before. :]

The pull request is for the **allow-custom-range** branch on jawwad's fork. Click the **Files changed** tab to see the included changes:



You'll see the following:

```
-def fizzbuzz():
-    for n in range(1, 101):
+def fizzbuzz(start=1, end=100):
+    for n in range(start, end + 1):
         value = fizzbuzz_for_num(n)
         print(value)
```

This looks like a fairly simple update and something that might come in handy, so you'd like to merge it into your development branch.

There are three ways to do this. You can:

1. Fetch changes directly from the other fork using its repository URL.
2. Fetch changes from upstream using a special pull request reference.
3. Add the other fork an additional remote.

Next, you'll try out the first way by using the other fork's repository URL directly.

Fetching directly from a URL

To fetch from a URL, just use that URL in place of the remote name. So, for example, instead of `git fetch upstream` you'd run:

```
git fetch https://github.com/raywenderlich/git-book-fizzbuzz.git
```

However, `fetch` behaves differently on URLs than on named remotes. As you saw previously, running `git fetch upstream` created the remote tracking branch **upstream/main**. But if there isn't a named remote, there's no namespace to create remote tracking branches in.

So you'll have to give the command the branch name to create. But when you specify a branch name as an argument, that argument is actually for the remote branch it should fetch:

```
git fetch {remote_url} {remote_branch_name}
```

So you have to give it the local branch to fetch it into as well:

```
git fetch {remote_url} {remote_branch_name:local_branch_name}
```

So what happens if you leave off the `:local_branch_name` part? The best way to find out is to try it out. Run the following:

```
git fetch https://github.com/jawwad/git-book-fizzbuzz.git allow-
custom-range
```

You'll see the following:

```
From https://github.com/jawwad/git-book-fizzbuzz
 * branch            allow-custom-range -> FETCH_HEAD
```

So what's this `FETCH_HEAD` thing? It's actually a reference that contains the last commit hash that was fetched. Run the following to see what it contains:

```
cat .git/FETCH_HEAD
```

You'll see:

```
c7580ff4a6231bbcfd21b46ddbb204ef472f590b        branch 'allow-
custom-range' of https://github.com/jawwad/git-book-fizzbuzz
```

Now, create a new branch based on `FETCH_HEAD` with the following command:

```
git branch acr-from-fetch-head FETCH_HEAD
```

The `acr` prefix is just an abbreviation for `allow-custom-range`.

Run **git log –oneline –graph –all** to verify that the branch was created:

```
* d1dcc72 (HEAD -> development, origin/fix-divisors-bug, fix-...
| * c7580ff (acr-from-fetch-head) Add start and end parameter...
|/
* 85ca623 (upstream/main, origin/main, origin/HEAD, main) Add...
...
```

Next, you'll run the same command again with a specific local branch name.

Run the following command to fetch the **allow-custom-range** branch from jawwad's fork into a local branch with the same name:

```
git fetch https://github.com/jawwad/git-book-fizzbuzz.git allow-
custom-range:allow-custom-range
```

You'll see the following, indicating that the branch was created:

```
From https://github.com/jawwad/git-book-fizzbuzz
 * [new branch]      allow-custom-range -> allow-custom-range
```

Run **git log –oneline –graph –all** to confirm:

```
* d1dcc72 (HEAD -> development, origin/fix-divisors-bug, fix-...
| * c7580ff (allow-custom-range, acr-from-fetch-head) Add sta...
|/
* 85ca623 (upstream/main, origin/main, origin/HEAD, main) Add...
...
```

Before you merge this change, you'll learn how to fetch this branch directly from upstream, since it's part of a pull request.

Fetching a pull request

Any branches that are part of a pull request are available on the upstream repository in a special reference that uses the format: **pull/{ID}/head**. So for this pull request, it would be **pull/3/head**.

Run the following to create a local **acr-from-pull** branch from **pull/3/head**:

```
git fetch upstream pull/3/head:acr-from-pull
```

Then run the following command to verify a local acr-from-pull branch was created:

```
git log --oneline acr-from-pull
```

You'll see `acr-from-pull` on the same commit hash as `allow-custom-range`, indicating that **pull/3/head** also pointed to the same branch:

```
c7580ff (allow-custom-range, acr-from-pull, acr-from-fetch-head)
```

Before you actually merge this change, you'll learn how to add the **jawwad** fork as an additional remote so you can simply run `git fetch jawwad`. This will allow you to experience how remote tracking branches are automatically created when you have a named remote.

Adding an additional remote

Run the following to add jawwad's fork as an additional remote:

```
git remote add jawwad https://github.com/jawwad/git-book-
fizzbuzz.git
```

Run **git remote -v** to confirm its addition:

```
jawwad https://github.com/jawwad/git-book-fizzbuzz.git (fetch)
jawwad https://github.com/jawwad/git-book-fizzbuzz.git (push)
origin https://github.com/{username}/git-book-fizzbuzz.git (fe..
origin https://github.com/{username}/git-book-fizzbuzz.git (pu..
upstream https://github.com/raywenderlich/git-book-fizzbuzz.git
upstream https://github.com/raywenderlich/git-book-fizzbuzz.git
```

Now, run **git fetch jawwad** and you'll see that the fetch command also created the remote tracking branches — since there's now a `jawwad` namespace to create them in.

```
From https://github.com/jawwad/git-book-fizzbuzz
 * [new branch]      add-type-hints     -> jawwad/add-type-hints
 * [new branch]      allow-custom-range -> jawwad/allow-custo...
 * [new branch]      fix-divisors-bug   -> jawwad/fix-divisor...
 * [new branch]      main               -> jawwad/main
```

This fetches all branches from that fork. You can verify this by comparing them to the branches on the following page:

https://github.com/jawwad/git-book-fizzbuzz/branches/all

Finally, remove the **jawwad** remote with the following command:

```
git remote rm jawwad
```

The `git remote rm {remotename}` command deletes the remote tracking branches as well.

You've seen three different ways to fetch updates from other forks. Now, you're finally ready to merge them!

Merging the pull request

Run the following to merge the **allow-custom-range** branch:

```
git merge allow-custom-range --no-edit
```

Now, delete the other two branches:

```
git branch -d acr-from-pull acr-from-fetch-head
```

It's a good idea to keep the **allow-custom-range** branch, even though you've merged it — just in case you need to re-create your development branch from the different branches that you merged into it.

Finally, push your **development** branch up to your fork:

```
git push -u origin head
```

Congratulations! You learned how to fork a repo and keep a fork up to date. Plus, you learned various ways to fetch changes from forks and from pull requests.

Key points

- You use the Forking Workflow to contribute to repositories that you don't have push access to, like open-source repositories.
- Forking involves three main steps: Clicking Fork on GitHub, cloning your fork, and adding a remote named upstream.
- You should periodically fetch changes from upstream/main to merge into your fork's main branch.
- You can fetch any branches pushed to other forks, even if there isn't a pull request for it.
- To fetch all changes from a named remote, use `git fetch {remotename}`.
- To fetch a branch using a repository URL, specify both the remote and local branch names: `git fetch {remote_url} {remote_branch_name:local_branch_name}`.

12 Conclusion

We hope this book has helped you get up to speed with Git! You know everything you need to know to effectively use Git on any sized project and team.

Version control systems like Git are incredibly important to coordinate and collaborate with file-based projects. Git, at its core, is very simple once you understand those fundamental pieces of what is going on when you commit changes. When things go wrong it is important to know how to step through resolving those issues, which you now know how to do.

If you have any questions or comments as you continue to use Git, please stop by our forums at http://forums.raywenderlich.com.

Thank you again for purchasing this book. Your continued support is what makes the tutorials, books, videos and other things we do at raywenderlich.com possible — we truly appreciate it!

– Chris, Jawwad, Bhagat, Cesare, Manda, Sandra and Aaron

The *Advanced Git* team