Git for Authors

Git for Authors

Robert A. Beezer
University of Puget Sound

David Farmer
American Institute of Mathematics

DRAFT January 13, 2026 DRAFT

How to Read This Book

The essential concepts of `git` are treated in the first four chapters, they should be read thoroughly and carefully, in order. Chapter 2, Chapter 3, and Chapter 4, treat the cases, respectively, of one, several, and many collaborators, with very different organizational models, but the concepts are universal to all three models.

Eventually you will find more intricate topics in Chapter 5 and Chapter 6 necessary or useful, and maybe both. Chapter 7 explains ways to backup and correct mistakes. Chapter 8 is a grabbag of short topics that do not fit naturally in the early material and would have just been a distraction. At some point familiarize yourself with what is on-offer there. The appendices contain technical information that might change over time, in addition to reference material.

Contents

Chapter 1

Introduction

Git, henceforth `git`, is a **revision control system**. What's that? It is a tool to record changes to software, a tool to experiment with changes to software, and a tool to collaborate creating software. Why might an author want to use a tool designed to keep track of software? Presumably your writing is in an electronic form, and you may want to keep track of your work, you may want to experiment with different approaches to your writing, and you may collaborate with others, not least perhaps a copyeditor and an editor.

`git` has a powerful feature, know as **branching**, which allows you to branch off in two different directions with your writing. You can decide to tie the two branches back together later, with a process known as a **merge** or you can decide to keep one branch, and kill off the other. On a collaborative project, two different collaborators can work on two unrelated personal branches of the same project. These branches can be local to their own computers, with neither collaborator aware of, or concerned about, the other's work. If their work is independent, then both collaborator's additions may be merged into the project separately, or if their work overlaps, `git` can be used to resolve the overlaps.

Example 1.0.1 Heroine versus Hero. Suppose you are well along on writing the next Pulitzer Prize winning novel. But you cannot figure out how you want it to end? In one scenario, the hero dies, and in another scenario the heroine dies. You might choose to make two copies of your novel in different directories and work on separate endings, eventually choosing one over the other. But let us suppose that in the course of doing this work in two different directories, you think of a great new way to develop your two characters in the early chapters. So now you need to make the same edits to both copies? Or maybe your novel is spread across several files, one per chapter, and you only need to edit a few early files in one version and copy just those files over to the other directory (hopefully not overwriting some other work there). This sounds a bit tedious, maybe complicated, and likely error-prone.

Let us see how `git` would handle this. Originally you are working on the main branch, almost always named `master`. Once ready to create two different endings, you would split off from `master` along two branches, perhaps named `hero` and `heroine`. You can now add material to each of the two different branches, switching between them at will. When you get your idea to improve the early character development, you can switch back to the `master` branch, edit your early chapters in a way that is common to both of your endings, then return to alternately working on your `heroine` and `hero` branches. You only ever have one copy of your early work, and if you ever make overlapping

changes on your two branches, then there are procedures for managing those conflicts. □

That example is a bit contrived, but basically realistic. And there is necessarily some hand-waving in how `git` helps you. We will guide you carefully through a similar application of `git` in Checkpoint 2.2.1.

To use `git` you need to author with source files that are simple text. So that likely means using some markup language like LaTeX, MathBook XML, Markdown, reStructured Text, or a similar language such as those supported by `pandoc`. XML formats for Word files might work out acceptably, but will be less than perfect. We are also going to show you how to use `git` at the command-line in a terminal, so you might need to consult a tutorial on using a command-line and file management. We have some quickstart information on `git` itself in Appendix A.

`git` provides some powerful tools, and perhaps as a corollary, has a steep learning curve. There is no *right way* to use `git`, and the way you choose to employ it may in large part depend on social aspects of your project and its organization. We will organize our initial material according to the number of contributors and the organizational structure of the people in your project: solo contributor (Chapter 2), a few collegial contributors (Chapter 3), or many contributors with a central authority (Chapter 4). Read all of these chapters, and read them in order, even if you know which model you might want to eventually employ. The techniques, concepts and principles build on each other, and there are no firm rules about numbers, procedures or organizational models. It is even possible they will fluctuate over the lifetime of a project.

We will distill some of the essential concepts of `git` into a short series of eight **Principles**, summarized in Appendix F. Review them often, until you feel you understand them well, and it will be much easier to grasp the finer details of `git`.

Principle 1.0.2 Git is a Tool. `git` *is a tool that can be adapted to many purposes and projects, in many different ways.*

Discussions on the Internet about *how best* to use `git` can generate a lot of heat, but not much light. Ignore them. Understand the basic principles of how the tool functions, learn the commands that work for your projects, and employ it to make your real work more productive, efficient and enjoyable. Simply becoming a `git` expert is not the object.

You will find that `git` is like a garage door opener. Without one, you wonder why you might ever need one, but once you start using one, you decide you could not live without it. Or to extend the automotive motif, it is like a seat belt–you feel unsafe without it. Enjoy getting to know `git`.

Chapter 2

All By Yourself

`git` is at its best when people collaborate, but it can still be valuable to an individual, and you may gain collaborators or contributors in the later stages of a project. We can understand some basic concepts by considering first the simple situation of a single, solo contributor.

You will not fully appreciate all of our Principles on a first reading, but if you come back to review them, they may be more useful on each reading. Here is one such.

Principle 2.0.1 Git Manages Changes. `git` *manages collections of changes to your files. It does so in linear sequences of incremental changes that are always consistent.*

You may be very comfortable with organizing your writing as a collection of files, perhaps further organized in a series of directories or folders, perhaps even nested several levels deep. `git` works on, and manipulates, your files for you, which can be disconcerting at some point. The objects that `git` stores and tracks are collections of *changes* to your files. One such collection might make several small changes to several different files (perhaps you renamed a character throughout your novel), or the change may be to add a new file (an entire new chapter, say). As you instruct `git` to move between branches, you might see your character's name change back and forth, or you might see your new chapter entirely disappear, only to reappear later. The files in your project are the cumulative result of many changes applied in sequence, not some final state that never regresses to an earlier state. But don't panic, `git` has all your changes stored away safely.

> `git` manages changes to your files, and those changes are accumulated in files, whose state may change in ways you are not accustomed to.

If that sounds scary, realize that RAB is self-taught when it comes to `git` and has never lost any work. He has panicked. But he has gained valuable experience puzzling his way out of some jams (see Chapter 7). And once he ended up applying the same collection of changes twice, mysteriously getting two of everything in a chapter (see Section 8.3).

2.1 Commits

Collections of changes can be called **changesets**, but we will be more likely to call each such collection a **commit**. That is a noun, not a verb. If you have

some experience with other revision control systems, then you might be familiar with the notion of "committing", or "checking in." Try to avoid confusing the new noun with your old verbs.

How do you make a commit? Roughly, you edit your files, so that your directory of files (your **working directory**) is **dirty**. The dirty or **clean** directory is a good mental image as you start working with `git`. Edit your files, and save your files. Normally, you feel pretty secure at this point. You have made changes, and by saving the edited files, you feel like you have saved your changes. But from `git`'s perspective, your files are dirty and you have not made your changes known to `git` yet. Here is the drill, using two commands at the command line in a terminal.

List 2.1.1 Making a Commit

1. Edit some files and save them, making your working directory dirty.
2. `git add <file1> <file2> <file3>`
3. `git commit -m "Add the incident at the train station"`

You will get no reaction (output) from the `git add` command, but when you actually make the commit, you should get a response like

```
[master c0f19a2] Add the incident at the train station
```

OK, that is a basic recipe, but what actually happened? In the `add` command you would have listed some, or all, of the files you had edited and saved. If you only listed some, the commit would only contain some of your changes, and the remaining changes would contribute to keeping your working directory dirty. The `add` command moves your changes in the indicated files to a staging area, a sort of purgatory, called the **index**. We say those changes are **staged**. You can incrementally add changes to the index to form a coherent set of changes that will eventually become a commit. For example, above you could have run the `add` command three times, once for each file, to stage the same collection of changes. If you further edit a file after `git add`, you can add that file again to move the subsequent edits into the index.

Realize that `git add` does two similar things. If `git` is unaware of some file, then `add` will make it one of the files that `git` **tracks** and will put the current contents of that file into the index. And from now on, `git` will include relevant details about this file in reports. For example if the file is dirty, then certain reports will show the changes (see next paragraph). But "tracking" a file *does not* mean `git` automatically packages up changes. That is your job. You have control of exactly which changes `git` will manage, and when you want `git` to become aware of those changes. Subsequently, `git add` moves changes from a file into the index, and you can do this repeatedly to update which collection of changes are staged in the index.

With all this talk of a dirty directory, how can you tell if your directory is even dirty at all? The command is `git diff`. It takes no action and is merely informative. You can run it anytime you like and it is wise to do it often, especially when getting started. RAB often walks away from his writing with a dirty directory (not best practice). So it is a good habit he has to *always* run `git diff` when first returning to a project. The output of `git diff` is all of the changes in your working directory that are not staged into the index. It is organized by file (given in yellow on my computer), with red text being removed and green text being added. White text is unchanged and provides

context for changed text, in order to help `git` apply changes in the right places. Solid red squares or bars are extraneous whitespace that serves no purpose other than to potentially confuse `git`. It is a good idea to become comfortable understanding this information. When all your changes are in the index, your working directory is now clean, and `git diff` reports nothing.

`git diff` drops you into a simple program known as a **pager**. The down and up arrows work to scroll through the output, the spacebar advances by a screenful, and the `b` key takes you back a screenful. Press `h` for help on more commands, and use `q` to quit and exit.

As you add changes to the index, you can see what your future commit looks like by running `git diff --cached`, which will report the accumulated changes in the index, using the same format.

After all this `add`'ing and `diff`'ing, making the commit itself is straightforward. `git commit` will do the job—moving changes from the index into a single collection of changes, a changeset, to be stored, managed and manipulated by `git`. Technically, this is an irreversible action, but in practice there are many ways to back-up and have a do-over, especially when you are solo. So don't panic.

The `-m` switch allows you to make a **commit message** on the command line, which you should enclose in quotation marks (single or double, allowing use of the other kind in your message, if needed). Without it, `git` will dump you in your editor, a step we prefer to avoid. Either way, you will always want to include a commit message. They can have multiple lines, but in practice we like to keep them to one concise line, leading with a capitalized action verb, and not more than about sixty characters. These messages will help you find your way in your `git` repository, and they will be the first thing others see if they peruse your repository. We think they are worth some thought toward making them informative and helpful, rather than sloppy and uninformative. You are an author, no? Treat your commit messages much like the entries that form a Table of Contents.

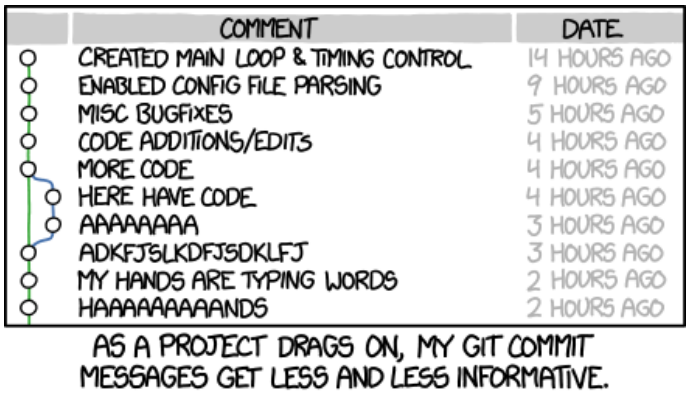

Figure 2.1.2 xkcd "Git Commit" (used with permission)

Unlike `git add`, the `git commit` command does acknowledge that something happened. The message shown above has the name of the branch (`master`) and the commit message. But what is `c0f19a2`? Every commit you ever create gets a hexadecimal identifier that is probabilistically unique across all the `git` repositories ever made and that will ever be made. And the first seven characters are usually good enough to uniquely identify commits within your repository.

Try the new command `git log`. It may not show much, but it will list information on every commit on your current branch. And you will see some

huge commit hashes. There are 268 435 456 different possibilities for seven hexadecimal digits. A full 40-digit commit hash has about 10^{48} possibilities. This is not a technical aside, we will see soon enough the critical role commit hashes play in a `git` repository (see Section 2.3). Even though your commit about the train station incident might often be shown shorthand as `c0f19a2`, it may in reality be

```
c0f19a223404c394d592661532747527038754e
```

which you would see in the log.

Here are two more useful diagnostic commands. `git status` will tell you which files are dirty, which files have changes staged in the index and destined for the next commit, and which files are lurking about in your directory, but which you have not ever told `git` about. This is a good command to run frequently, especially when you are beginning. Finally `git ls-files` will output all the files `git` has changes for. This one is interesting, but less useful day-to-day.

We will primarily teach by guiding you through exercises. They almost always have extra information, so read them just for that. Experiment with a scratch repository where you can try different things without the inevitable mistakes also being worrisome disasters. And realize you can always start over.

Checkpoint 2.1.3 My First Repository.

1. Consult Appendix A for instructions, setup `git`, and `init` an empty repository.

2. Make several commits, creating and adding at least three files into the repository. Use `git diff`, `git diff --cached`, `git status`, and `git log` liberally in the process.

   Put some non-trivial content into each file (though it does not need to be excessive). We will use this repository for future exercises, so do not get rid of it. Put a typographical mistake into one of your three files.

3. Do not experiment with branches, we will do that next.

4. Once completed, use `git show master` to see the changes in your last commit, in the diff format. Use `git show master~1` to see the changes in the commit just prior to that one. And `git show master~2` for the one before that. Try replacing the references (`master~N`) by the first seven or eight digits of the commit hash, which you can get from the output of `git log`, and see that this is the functional equivalent of using branch names with relative references.

When you are done, the logical arrangement of your commits might look like the following diagram. We list older commits at the bottom and do not include commit messages. We use a 4-digit hash, which will uniquely identify each commit. The name with the arrow points to the **tip** of the branch with that name. The commit at the bottom, the first commit ever, is known as the **root commit**. As your repository gains more branches, it will look more and more like a tree than a twig. This diagram should be similar in spirit to what `git log` reports for this simple first exercise.

Figure 2.1.4 Completed First Repository

2.2 Branches

The term **branch** gets used many ways in `git`, as a verb and a noun, likely because it is so fundamental. We will introduce you to this concept with an exercise, that will also give you some idea of the power of the idea. So even if you do not work the example right away, read through it, since it contains a lot of explanation.

Checkpoint 2.2.1 Hero and Heroine on Branches. Make sure your practice repository has at least three files in it, with one typographic mistake. So go ahead and create a new file and a new commit if you need to. We will wait.

We are going to pretend that we are writing a novel and we want to design two possible endings. In one, the hero dies, and in the other, the heroine dies.

Make sure your working directory is clean and type `git branch heroine`. Not much will appear to have just happened. So here is a new diagnostic command you will want to use all the time. Type `git show-branch`. This shows all of your possible branches, with an asterisk ("`*`") indicating the branch you are "on" and an exclamation mark indicating other branches ("`!`"). It is not very informative yes, since your repository is just beginning to take shape and you have only just *created* a branch. Be sure to keep this output visible in your terminal for an upcoming comparison.

Now we are going to move to you new branch. Ready? Type `git checkout heroine`. Now run `git show-branch` again and compare to the previous output. The big change is that the asterisk has moved to indicate the `heroine` branch. We say you are now **on** the `heroine` branch. If you have experience with other revision control systems, the word "checkout" is fraught with other meanings that are not accurate. Sorry, just get over it. In `git` the `checkout` command will change the files in your working directory to some possibly different state, based on a different sequence of changes in a sequence of commits constituting a different branch. (Remember, commits are collections of changes.)

Choose one of your two files without a typo and edit it to include some words about the heroine dying, and also remove some existing words at the same time. Use `git diff`, `git add`, `git commit` to form a new commit that is the changes you just made, and with a commit message about creating an ending where the heroine dies. You should have a clean working directory after the commit.

Now `show-branch` will show something interesting:

```
rob@lava:~/books/ppw$ git show-branch
* [heroine] Create an ending where the heroine dies
 ! [master] <message about last commit on master>
--
*  [heroine] Create an ending where the heroine dies
*+ [master] <message about last commit on master>
```

The top half lists all your branches, with color-coding and identifying symbols. The bottom half tries to describe to you how those branches relate. We will make it even more interesting shortly.

Close any files you have open in your editor, type `git checkout master`, and open the files again. Don't panic. You have restored your working directory to the state of all the accumulated changes up to the tip of the `master` branch, so you do not see your most recent change to your novel. The changes that constitute the brilliant ending about the death of the heroine is saved by `git` in a place that it is hard for you to find but easy for you to manage. Would you like your ending back? Close your files, type `git checkout heroine`, and

open your files. `git` has manipulated your files and applied the changes with the ending, so now your novel should contain the new ending.

Perhaps you now see why `git` can feel a bit foreign and why at some point you will have that "Oh, &\#*!" feeling. Don't panic. `git` manipulates the state of the files in your working directory, so in a way those files are ethereal. `git` stores your commits (collections of changes) safely and can rewind and replay them in ways consistent with your writing while on various branches, using your working directory as a sort of sandbox or laboratory.

Switch back and forth several times between the `master` and `heroine` branches with `git checkout`. Close and open your files in your editor as you let `git` manipulate your working directory. Run `git show-branch` liberally. When you are ready, leave a file open when you know a `checkout` is going to change it. How does your editor react? We don't know, so can't help. But a good editor might say something like "File on disk changed, do you want to reload it?" We use an editor that just silently reloads the file, unless there are unsaved edits in it by mistake. If a `git checkout` results in a file not even being present, our editor leaves it in a state we find very confusing. Sometimes this sort of behavior can be configured in a editor. Experiment until this is not confusing and in the meantime as you are learning, close and reopen files as you instruct `git` to manipulate the state of your working directory.

Ready for another branch? Make sure you are on `master` (by running `git checkout master`). Then make a new branch with `git branch hero`, and switch to it with `git checkout hero`. Run `git show-branch` as you go, studying carefully how the output is changing.

Open the file that is *different* from the one you edited before, and that is not the one with the typo. Remove some words, and author an ending where the hero dies. Create a commit with these changes, using a commit message about the hero dying. (We might now use "commit" as a verb, and say **commit your changes**.) Now `show-branch` will produce something like:

```
rob@lava:~/books/ppw$ git show-branch
* [hero] Made an ending where the hero is dead
 ! [heroine] Create an ending where the heroine dies
  ! [master] <message about last commit on master>
---
*   [hero] Made an ending where the hero is dead
 +  [heroine] Create an ending where the heroine dies
*++ [master] <message about last commit on master>
```

Now this is interesting. You have three branches, you are on the `hero` branch, and the `heroine` and `hero` branches contain divergent storylines based on a common beginning. (Everything up to `master` is common and not reported in this output.) Experiment checking out different branches, run `show-branch` liberally, and do not forget that you can do things like `git show hero` even if you are not presently on the `hero` branch. Get comfortable with the current state of your repository. Notice that each of our two new branches could each be a sequence of many commits, but we just need one commit each for the purposes of this exercise.

As you looked around, did you notice that typographical error you made in the third file when you first start writing? (If not, play along.) Checkout the `master` branch, open the file with the typo and and edit to correct the mistake. Commit those changes. You should now see:

```
rob@lava:~/books/ppw$ git show-branch
! [hero] Made an ending where the hero is dead
 ! [heroine] Create an ending where the heroine dies
  * [master] Correct a misspelling
---
  * [master] Correct a misspelling
+   [hero] Made an ending where the hero is dead
 +  [heroine] Create an ending where the heroine dies
++* [master^] <message about last commit on master>
```

Notice that the new commit has resulted in the tip of the `master` branch moving to be the commit with the typo fix. Every commit we have made so far has also moved the **branch pointer** to a new tip, but this is the first time it was really interesting and significant. Notice that through all these pointer movements we no longer have a pointer to the commit where all the branches originated. Notice too that this report still shows the last of our pre-exercise commits, since that is where our three branches diverged from common material.

The good news is that we have two versions of our story and only fixed the typo once, while the bad news is that the `master` branch has advanced and left the `hero` and `heroine` branches behind. Do you see that in the output above? As you checkout the three branches, you will see the effect of the three editing sessions, but they are all disjoint. Said differently, switching to the `hero` branch results in `git` manipulating your working directory by applying a sequence of commits that do not contain the commit with the typo fix. And similarly for the `heroine` branch. But not all is lost.

Here is where you begin to learn how to leverage `git`. We are going to rewind the `hero` branch back to the commit where the branch began, and then we are going to replay our changes at the new tip of the `master` branch. And `git` knows just how far to rewind. (This would be slightly more interesting if our `hero` branch had multiple commits beyond the old branch point.)

Ready? Make sure your files are closed, and checkout the `hero` branch, since this is the branch we are changing. Run `git log` now and save the output some place, because we want to see how the commit hashes behave. Now run `git rebase master`. This tells `git` to rewind the current branch (`hero`), checkout `master`, branch off `master`, and replay the commits that were just rewound (in the opposite order), leaving the `hero` branch pointer at the tip of the branch. Remember, `git` manipulates the files in your working directory by applying (replaying) commits in sequence.

You might be aware that in replaying the commits, `git` might get confused by intermediate changes that had not been present when the commits were based on the original branch point. These situations are known as **conflicts**. `git` can be pretty smart about these, but sometimes they require intervention by the author. If you have followed our instructions and used three files, then this will not happen in this exercise. We address conflicts in Chapter 3 and Chapter 4.

Open your files and you should see the hero dying, and you should also see the typo fixed. When I need to run this command, I say to myself "`rebase` the current branch onto `master`" to make sure I get it right and do the right thing. Examine the result with `git show-branch`.

Repeat this procedure to move the `heroine` branch up to the tip of `master`. This is an exercise, so we will not hold your hand on this one. Think carefully about each step while reviewing how we rebased the `hero` branch. When you are done, you should see:

```
rob@lava:~/books/ppw$ git show-branch
! [hero] Made an ending where the hero is dead
 * [heroine] Create an ending where the heroine dies
  ! [master] Correct a misspelling
---
+   [hero] Made an ending where the hero is dead
 *  [heroine] Create an ending where the heroine dies
+*+ [master] Correct a misspelling
```

Notice that *all* of the commits for the beginning of the story, prior to our typo fix are not shown in this output because they are now common to all the branches.

Before you experiment too much, checkout the `hero` branch, and as beforehand, run `git log`. Compare with the output you save a few steps back. You should see that the first commit (hero dies) now has a *totally different* commit hash, the second commit is the typo fix (which was not evident beforehand), and the present third commit is the previous second commit, with the *identical* commit hash. We will discuss this outside of the exercise. Now, explore your repository thoroughly with the various commands and diagnostics you have learned.

That is it, we are done. Notice that you are maintaining two versions of your novel, can easily switch between working on one or the other, and you can edit common material just once in a way that is reflected in each version. If you made a mess of this (which might be a good thing), just start over with a new repository and have a do-over. Or maybe repeat the exercise with extra commits on each branch for a fuller experience and to solidify your new skills.

The previous exercise begs the question: what to do once we decide which ending we want? The next exercise has the answer.

Checkpoint 2.2.2 Heroine Meets Her End. Having thoroughly explored the possibilities, we have decided to kill off the heroine as the ending of our novel. Briefly, we are going to take all the changes from the `heroine` branch and move them into the `master` branch permanently. Then we will deal with the `hero` branch.

First, checkout the `master` branch since that is the branch we will be changing. Then it is as simple as `git merge heroine`. Through this **merge** you have taken the changes from the `heroine` branch and brought them into the `master` branch. You should have output something like:

```
rob@lava:~/books/ppw$ git merge heroine
Updating 842f83c..c7fe82d
Fast-forward
 some-file.txt | 4 ++--
 1 file changed, 2 insertions(+), 2 deletions(-)
```

The file listed should be the one you edited with the story of the death of the heroine. For more diagnostics:

```
rob@lava:~/books/ppw$ git show-branch
! [hero] Made an ending where the hero is dead
 ! [heroine] Create an ending where the heroine dies
  * [master] Create an ending where the heroine dies
---
+   [hero] Made an ending where the hero is dead
 +* [heroine] Create an ending where the heroine dies
++* [hero^] Correct a misspelling
```

Now the `heroine` and `master` branches are identical, and the `master` pointer

has advanced one commit, so that the `hero` branch still splits off prematurely from `master`, but the `heroine` branch does not.

Running `git log` you will see that the second commit (the typo fix) was reported by the merge as a short commit hash in the output (`XXXX`) and the first commit (the heroine ending) is also reported by the merge via a short commit hash (`XXXX`). This is the movement of the `master` branch pointer, indicated by `..` between the hashes.

This is actually a very special type of merge, known as a **fast-forward** merge, as reported in the output. Since the `heroine` branch splits off from the tip of the `master` branch, it is trivial to rewind all the `heroine` commits back to `master` and then replay them onto `master`. Think about that for a minute. Not only is it trivial, it borders on silly.

Here is what really happens in a fast-forward merge. No commits are rewound and replayed. `git` simply moves the `master` branch pointer from its original location, so that it points to the same commit as the one that `heroine` points to. This has two implications. First, none of the commits on the `heroine` branch change in any way. Second, once the merge is completed `master` and `heroine` are redundant, as they point to the same commit, as you can see in the top half of the output from `show-branch` where the commit message is duplicated.

The previous paragraph is an important realization, but you might be wise to forget about it. In the long run, it is better to think of this merge as bringing (merging) the changes on `heroine` into `master`. That is the more general situation, and we remember how the simpler version of describing a fast-forward merge confused more general concepts later.

Now we have a bit of a mess on our hands. The `heroine` branch pointer is redundant, but really it is obsolete. Our trial death of the heroine is no longer experimental, we have decided that is the ending we want and now it is part of `master`. Let us kill the `heroine` branch pointer too.

```
rob@lava:~/books/ppw$ git branch -d heroine
Deleted branch heroine (was c7fe82d)
```

Note in the output the short-version commit hash for the commit that is the current tip of `master`, that is what `heroine` also pointed to. Most of your work will initially reside on branches, so deleting a branch pointer sounds like a dangerous thing to do, and it is. But `git` has your back. Try the following:

```
rob@lava:~/books/ppw$ git branch -d master
error: Cannot delete the branch 'master' which you are
    currently on.
```

That is good. You would not want to delete the branch you were on, as then you would be totally lost. So let us try cleaning up another part of our mess. We still have the `hero` branch, with the ending we did not choose. Let us delete that:

```
rob@lava:~/books/ppw$ git branch -d hero
error: The branch 'hero' is not fully merged.
If you are sure you want to delete it, run 'git branch -D
    hero'.
```

When we deleted the branch pointer to `heroine`, `git` knew it was redundant and happily deleted it at once with no questions asked. But the branch pointer `hero` lets us reference commits we cannot locate any other way, specifically the commit where we let the hero die. So `git` makes us be a bit more certain and we need the upper-case variant of the switch to get rid of it. Be careful, because if you lose track of commits, they can be hard to resurrect and eventually `git`

will **garbage collect** them when it does some automatic spring cleaning of your repository. For us, there is no real harm in leaving the `hero` branch in place, and it is instructive to wait a while before killing it. It is doing no damage where it is, we do not ever need to check it out, though eventually we will grow tired of seeing it in all our diagnostic reports.

Realize that the decision above to merge `heroine` into `master` should not be taken lightly, as it is quite difficult to reverse it, and not really in the spirit of how you use `git`. Branches like `hero` and `heroine` are sometimes called **topic branches**. Or in a nod to software development they may be called **feature branches**, since they might be used to create and test a new feature for a larger piece of software. I like to refer to a branch like `master` as the **mainline**. An essential aspect of working with `git` is to always work on a branch, and realize that there is little cost to making a branch, merging it, and then deleting the (temporary) branch pointer. I have made branches that only had a lifetime of five minutes. That is a principle.

Principle 2.2.3 Always Work on a Branch. *Always make a topic branch off your mainline when starting new work, and only merge once satisfied.*

2.3 Commit Hashes

In computer science, a **hash function** is a many-to-one function that takes lengthy input, massages it, and produces vastly shorter output of a fixed length, called the **hash**. While the output looks random, the function has no randomness, and the actual function is well-known to everyone. Identical inputs will always produce the same output. But if two inputs differ even slightly, they will have wildly different outputs—this is a design criteria for most hash functions. Similarly, we might require that if we know the hash (output), it is very hard to manufacture input to the hash function to produce that output. For the hash function used in `git` it is highly unlikely that two different inputs (commits) will produce the same output (hash). So the commit hash is similar to how we use fingerprints or retinal scans to identify humans. If you want to learn more, `git` uses the **SHA-1** hash function, whose tamperproof properties were designed for use in cryptographic applications. It is no longer considered secure for that purpose, but works fine for use in `git`.

The input `git` uses to form the hash include things like your name, the date, the commit message, the changes in the commit, and *most importantly*, the commit hash of the previous commit in the sequence on the branch. This is similar to the **blockchain** technology used in BitCoin, and lately a darling of fintech (financial technology). A sequence of blocks of information and their hashes, each formed with input including the previous hash, makes it practically impossible to tamper with any one piece of information without radically disturbing every subsequent hash.

Return to the fast-forward merge above to see that there was absolutely no change in the information for each commit and there was no disruption in the sequence (chain) of commits. As explained, the only thing that really happened was that the `master` block pointer was changed to point to a new commit, the tip of the `heroine` branch.

Contrast that to when we rebased the `hero` branch onto the `master` branch that contained the typo fix. `git` rewound the commits on `hero` up to the old branch point, but then replayed them onto a commit (the tip of `master`) with a different commit hash. All of the replayed commits from the `hero` branch (we just had one) will have their commit hashes recomputed and will be radically different since the hash of the tip of `master` is an input to the first commit of

the rewound branch. The previous hash(es) are meaningless (and lost to time). Notice that `git` updates the branch pointer `hero` to use a new hash from the tip of the replayed branch.

This is a principle that will be important once we get social and work with others.

Principle 2.3.1 A Rebase Changes Hashes, a Merge Does Not. *A rebase will always change some commit hashes, while a merge will never change any commit hashes.*

You now know, and have experience with, four of the six important concepts of working with `git`: committing, branching, rebasing and fast-forward merging. Only (general) merging and pull requests remains. But first, let us get social and begin collaborating with others, to realize some of the most powerful aspects of `git`.

Chapter 3

With a Few Friends

One of the real benefits of `git` is the ability to collaborate easily with others in a de-centralized manner across time and space. In this chapter we will experiment with a model that is appropriate for a small group of collegial collaborators. My dictionary (WordNet) defines "collegial" as "characterized by or having authority vested equally among colleagues," which is precisely the situation we will simulate. So, in this chapter we will explore the scenario where a small group of equals works on a writing project, on the assumption that everyone in the group is trusted to make any sort of change at any time.

We will *not* be using email attachments or Google Docs to communicate. Rather, an easy way to collaborate with `git` is to place a copy of the repository on a server where each collaborator has the right privileges to interact with the repository. If your project is secret and sensitive, this might be a server at your workplace, or a web host you control and trust. Or maybe the project is not so sensitive and a private account at GitHub is appropriate and easy to set-up. Or maybe your project has an open license and an open repository will eventually allow total strangers to contribute to your project (see Chapter 4). For exercises in this chapter, we will use GitHub, a (free) site that hosts `git` repositories along with tools supporting collaboration around a `git` repository.

Principle 3.0.1 Merge into Your Current Branch. *A merge integrates changes into your current branch, from a branch you specify.*

3.1 Collaborating across Time and Space

To get started, visit Appendix B, read the advice on making a GitHub account, and make an account. Then come back here to work the following example. To keep things simple, we will walk through an exercise with just two co-authors, but you might imagine up to about five individuals participating in the following.

Checkpoint 3.1.1 Alice and Bob Write Crypto. Alice and Bob are two professional cryptographers who have discovered a weakness in a critical algorithm underlying much of the world's electronic banking programs. They need to get the details out quickly as a research paper that they will host on the arXiv for the security community to vet. Alice and Bob have known each other for years. They trust each others technical skills and writing style, and even better, they both have GitHub accounts. In the best traditions of cryptography research, they decide to write their paper openly as a public GitHub repository, and they decide to host the repository in Alice's account. Everything else will

be discussed on GitHub.

Work this exercise playing the role of Alice. If you have a friend who can be Bob, all the better, but you can also play both sides of the collaboration yourself and get almost as much out of the exercise (if Bob is somebody else, then he need a GitHub account, but if you are playing both sides, then your one GitHub account is enough.). Alice (you!) will log into her GitHub account and initiate a new repository. Recall that in Chapter 2 we created a new repository on our local computer at the command-line with `git init`. Now Alice will let GitHub do that step since GitHub will automatically configure the repository for subsequent communication.

See Section B.2 and Section B.3 for instructions on the steps in this paragraph. Alice will create a new repository and name it `banking-paper`. She will make Bob a collaborator on the repository since she knows Bob's username on GitHub from their previous collaborations. So there is now a fresh repository on GitHub, which Alice and Bob can manipulate. We are going to call this the **definitive repository**, as it will hold the "official" version of their paper. In a minute we will setup Alice and Bob with local copies, but they have agreed that those are just their local workspaces and the repository on GitHub always holds the latest, and presumably best, version of their paper.

Section B.4 contains the necessary instructions for this paragraph, but are more general, so read them and this paragraph through completely before doing anything. In particular, ignore any discussion of "forks" until Chapter 4. Alice should make a copy of the fresh repository onto her work computer, and Bob should do the same. If you are playing both sides this exercise yourself, copy the repository once, and then rename the `banking-paper` directory to `alice-banking`. Then copy again and rename the resulting directory as `bob-banking`. These changes have zero effect on how your repository behaves, but you will need to mentally figure out which files you should be working with in the remainder of the exercise.

In principle, Alice and Bob are totally setup and organized, and never even need to visit the GitHub site ever again. But GitHub has some nice tools and Alice and Bob have decided to be 100% transparent in their work. A GitHub **issue** is like a topic on an online discussion forum. It is designed mostly for reporting and discussing bugs in software, or requesting and implementing new features in software. But they can also be used for planning and discussion. Alice and Bob would like to plan their writing as an open discussion on GitHub, deciding that Alice will concentrate on the introduction since she is the better overall writer, and Bob will therefore get started on the section with the details of the vulnerability. They will work more closely on the final section containing recommendations.

So in our exercise, Alice should create a branch off of `master` named `intro`, create a file `intoduction.txt`, add it to her branch, make some edits, commit the changes, and so on. Bob should do similarly but make a branch off `master` named `vulnerable` where he adds and edits a file `vulnerability.txt` as a series of commits. Recall that Principle 2.2.3 says Alice and Bob should do all of their work on branches.

Alice had the simpler task, so let us assume she finishes the introduction first. She does not know she is first, she does not even have any idea where Bob is in his writing. She has been doing her best to get the introduction right, and to not disturb Bob, who is presumably also working hard. So Alice suspects there are no new commits on the `master` branch, but does not really know. OK, Alice is going to update `master` with a **pull**, see no new commits there, do a fast-forward merge of her `intro` branch into `master` locally, and then **push** her `master` branch to GitHub. We will do the details carefully, but

recognize that the `push` and `pull` are the only new concepts we did not cover in Chapter 2.

But first, a bit of diagnostic work. Alice's repository was copied from GitHub and therefore is aware of its heritage.

```
alice@work:~/papers/banking-paper$ git remote -v
origin  https://github.com/alice-jones/banking-paper.git
    (fetch)
origin  https://github.com/alice-jones/banking-paper.git (push)
```

Alice's local version of the repository has a **remote** that carries the information necessary to communicate with the definitive repository. Since Bob made a similar copy he has an *identical* remote (remember the co-authors are sharing a definitive repository in Alice's account). The remote goes by the name `origin`, which is customary, similar to the `master` branch. You can add as many remotes as you like, putting your repository in contact with as many different copies as you can think of.

```
alice@work:~/papers/banking-paper$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
```

Notice that `git` seems to know something about the state of your local `master` in relation to the `origin/master` in the copy on GitHub. Let us check anyway.

```
alice@work:~/papers/banking-paper$ git pull
Already up-to-date.
```

So Alice attempted to update her local `master` branch from the definitive repository on GitHub, but there was nothing new to use as an update (her `master` is `up-to-date`). As we suspected (or hoped!), Bob is still working on the technical details locally. In case it was not obvious, we did not have to bother Bob with an email asking where he was with his task. Now Alice is going to merge her `intro` branch into `master`, which will be a fast-forward merge since `master` has not evolved beyond her original branch point for `intro`.

```
alice@work:~/papers/banking-paper$ git merge intro
Fast-forward
 introduction.txt | ++++++++++++
 1 file changed, 25 insertions(+), 0 deletions(-)
 create mode 100644 introduction.txt
```

Alice has incorporated her introduction to the `master` branch, but now will make it part of the definitive repository with a `push`. Note that Alice is still on her `master` branch.

```
alice@work:~/papers/banking-paper$ git push
Username for 'https://github.com': alice-jones
Password for 'https://alice-jones@github.com': xxxxxxxxxx
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 287 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/alice-jones/banking-paper.git
   80a9eb3..1aaedaf  master -> master
```

Notice that Alice must sign in to GitHub, since only she and Bob are allowed to modify the repository (the world can view it and copy it, not modify it). So Alice has placed her introduction in the definitive repository without any

additional coordination with Bob. Time is important, so Alice initiates an issue on GitHub to discuss their recommendations, which will now interrupt Bob, but they need to form a plan. However, without waiting for Bob's reply, Alice creates a new branch off `master` named `last-chapter` where she simply adds an empty file named `recommendations.txt`. She repeats the steps above and with a fast-forward merge, updates her local `master` and pushes it to the definitive repository on GitHub. Alice can now clean up by deleting her `intro` and `last-chapter` branch pointers that have become obsolete.

Now that Alice has communicated with a public repository, known to the world, it is the right time to introduce a principle that we will illustrate subsequently.

Principle 3.1.2 Never Alter a Public Commit. *Never, ever, alter in any way a commit that has been made available to anybody else.*

We discussed the nature of commit hashes in Section 2.3. We have seen how a local rebase changes commit hashes in Checkpoint 2.2.1. And we have a principle about a `rebase` changing hashes, while a `merge` does not (Principle 2.3.1). Alice can rebase her branch all she wants within her repository on her local computer, but the instant she pushes commits to the definitive repository, they become available to her co-author (Bob), and to the *entire world*. The commit hash for each of these commits is a globally unique ID (**GUID**). There are ways to modify these public commits in the definitive repository, but this would be tantamount to chopping off somebody's finger and replacing it with a new one with a different fingerprint. Don't do it!

Why not? All of `git`'s coordination is predicated on identical commits having identical ID (the commit hash). In Chapter 4 we will expand our circle of contributors to anybody in the world (don't panic, we will have a procedure for approving changes before they go into the definitive repository). Manipulating a public commit will totally confuse `git`, make a big mess, and infuriate your collaborators, whose copies of the repository are no longer consistent with your ill-advised action. This may be the only advice that all the Internet `git` commentators can agree on. If you follow the procedures we are describing, this will never be a danger. But when you push a commit to the master branch of the definitive repository and feel like you made a mistake, resist the temptation to go backwards locally and then do a "forced push" to the definitive repository. You might get away with it for a while, but eventually you will regret it. Just live with it (a misspelled commit message), or add another commit to fix your mistake (a grammatically poor sentence). And forget we even mentioned the possibility of changing public commits.

Back to our exercise, now Bob has finished up his section, so he wants to make it part of the definitive repository. He suspects Alice has finished the introduction, since she was eager to discuss the recommendations. So, just like Alice, he is going to update his `master` branch.

```
bob@laptop:~/publications/banking-paper$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
```

Notice that this particular `up-to-date` message is misleading. Bob will really check with a pull. Remember that Bob has a remote named `origin`, with connection information for the definitive repository. (Why doesn't a pull require a login?)

```
bob@laptop:~/publications/banking-paper$ git pull
remote: Counting objects: 2, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 2 (delta 0), reused 2 (delta 0), pack-reused 0
Unpacking objects: 100% (2/2), done.
From https://github.com/alice-jones/banking-paper
   1aaedaf..1912c1b  master     -> origin/master
Updating 80a9eb3..1912c1b
Fast-forward
 introduction.txt    | ++++++++++++
 recommendations.txt | 0
 2 files changed, 25 insertions(+), 0 deletions(-)
 create mode 100644 introduction.txt
 create mode 100644 recommendations.txt
```

Bob just picked up Alice's introduction section and empty recommendations section from the definitive repository. That's good. Do you see the subtlety? Think carefully about it for a minute, this is a major concept. The branch pointer `master` in Bob's local repository just advanced two commits forward from the branch point of the `vulnerable` branch he has been working on. Bob sees:

```
bob@laptop:~/publications/banking-paper$ git show-branch
* [master] Empty chapter for recommendations
 ! [vulnerable] Vulnerability section
--
*  [master] Empty chapter for recommendations
*  [master^] Introduction
 + [vulnerable] Vulnerability section
*+ [master~2] Initial commit
```

Before Bob pushes his technical section, he will rebase his `vulnerable` branch onto `master`. This would normally risk a **merge conflict**, but he knows Alice has work only in `introduction.txt`, his new work is only in `vulnerability.txt`, and `recommendations.txt` is empty. So Bob switches to `vulnerable`, rebases onto `master`, switches to `master`, merges vulnerable into `master` (which is a fast-forward merge, as expected), pushes `master` to `origin/master`, and cleans up by deleting the obsolete `vulnerable` branch pointer.

```
bob@laptop:~/publications/banking-paper$ git checkout vulnerable
Switched to branch 'vulnerable'
bob@laptop:~/publications/banking-paper$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: Vulnerability section
bob@laptop:~/publications/banking-paper$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
bob@laptop:~/publications/banking-paper$ git merge vulnerable
Updating 1912c1b..bceacaa
Fast-forward
 vulnerability.txt | ++++++++++++
 1 file changed, 90 insertions(+), 0 deletions(-)
 create mode 100644 vulnerability.txt
bob@laptop:~/publications/banking-paper$ git push
Username for 'https://github.com': bob-smith
Password for 'https://bob-smith@github.com': xxxxxxxxxxx
Counting objects: 2, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 279 bytes | 0 bytes/s, done.
Total 2 (delta 1), reused 0 (delta 0)
To https://github.com/alice-jones/banking-paper.git
   1912c1b..bceacaa  master -> master
bob@laptop:~/publications/banking-paper$ git branch -d vulnerable
Deleted branch vulnerable (was bceacaa).
```

That looks complicated, but you will do it over and over if you collaborate with this model. Update master from the definitive repository, rebase (advance) your working branch to the tip of master, fast-forward merge your working branch into your local master, then push your improved master to the definitive repository. The clear and present danger here is to forget the first step, updating your master. You may have little idea what has happened to the definitive repository if you have had your head down working, so you need to *always* remember to update before interacting with the definitive repository.

Where are we now? Alice and Bob both have the same introduction, and they both have an empty section destined to hold recommendations. Bob has pushed his technical section to the definitive repository, just now. However, Alice does not yet have the technical section on the vulnerability. Notice that at any time, Alice (or Bob) can make a commit on their present branch (so their working directory is clean), checkout `master`, and pull `origin/master` from the definitive repository. This will likely advance the branch pointer for `master` but there is no harm in that at all. It is good, since now Alice's local master has the latest changes from the definitive repository and she can see how the project is progressing.

Alice can return to her working branch, *exactly* as it was after she switched away from it. *At any time*, she can *elect* to rebase her working branch on a new tip of the `master` branch. And it would be best practice to rebase frequently and perhaps only encounter minor, easily resolvable merge conflicts regularly, rather than a head-in-the-sand approach that waits to do a single massive rebase before adding to the definitive repository. Even better would be to rebase onto an updated master immediately and then push immediately. Work can continue on a fresh local branch. That feels like a principle.

Principle 3.1.3 Rebase Working Branches Often. *Frequently rebase private working branches onto an up-to-date master branch.*

What does "private" mean here? We have seen that commit hashes form

a chain of repeated hashes all the way back to the root commit (Section 2.3), and that we can identify commits by the leading digits of a commit hash. Also, a rebase will always change some commit hashes (Principle 2.3.1). So while the present principle advocates frequent rebases, never perform a rebase that changes a commit hash that has been made available to somebody else. This is the advice contained in Principle 3.1.2, and now could be a good time to back and re-read the discussion that follows it.

We have seen how to pull from the `master` branch of the definitive repository, and how to push commits to the `master` branch of the definitive repository. So we have the tools for two-way communication between repositories. We can pull from public repositories at will, but need permission to push to repositories where we are trusted to make unilateral changes.

3.2 Conflicting Edits

Let us do one more exercise with Alice and Bob.

Checkpoint 3.2.1 Alice and Bob Edit Simultaneously. With agreement about the recommendations, and in a rush, both Alice and Bob are going to simultaneously perform conflicting edits in the recommendations section. We will purposely engineer a merge conflict as an exercise, so follow the instructions carefully.

Alice should make a new branch, named `rec`, for her work on the recommendations. She should write three short paragraphs in `recommendations.txt`, separated by blank lines, and she should commit those changes on her branch. Bob should make a new branch, named `recommend`, and should also author three short paragraphs in `recommendations.txt`, separated by blank lines, and he should commit those changes on his branch. However, Alice and Bob should collude (outside of GitHub) to have identical first and third paragraphs, but radically different second paragraphs. Of course, this is unrealistic, but this makes the exercise work. If you put Alice's name into her second paragraph someplace, and put Bob's name into his second paragraph, the exercise will be even more informative.

We will let Alice go first, since she got a head-start on Bob originally and she does not yet have the section with the technical details of the vulnerability. Alice updates her `master` branch from the definitive repository (`origin/master`), and in doing so, she will pickup Bob's technical section. Then she should rebase her `rec` branch onto master, which will go smoothly. Then she merges `rec` into her local `master` (which will be a fast-forward merge) and push the improved `master` to `origin/master`.

So the definitive repository has Alice's recommendation section, while Bob's recommendations are still local to his repository, and `origin/master` has advanced beyond Bob's local `master`. Here is the transcript for Bob (again).

```
bob@laptop:~/publications/banking-paper$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
bob@laptop:~/publications/banking-paper$ git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 3 (delta 1), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/alice-jones/banking-paper
   bceacaa..b7abe18  master     -> origin/master
Updating bceacaa..b7abe18
Fast-forward
 recommendations.txt | 5 +++++
 1 file changed, 5 insertions(+)
bob@laptop:~/publications/banking-paper$ git checkout recommend
Switched to branch 'recommend'
bob@laptop:~/publications/banking-paper$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: Bob recommends
Using index info to reconstruct a base tree...
M       recommendations.txt
Falling back to patching base and 3-way merge...
Auto-merging recommendations.txt
CONFLICT (content): Merge conflict in recommendations.txt
Failed to merge in the changes.
Patch failed at 0001 Bob recommends
The copy of the patch that failed is found in:
   ~/publications/banking-paper/.git/rebase-apply/patch

When you have resolved this problem, run "git rebase
    --continue".
If you prefer to skip this patch, run "git rebase --skip"
    instead.
To check out the original branch and stop rebasing, run "git
    rebase --abort".
```

Boom! The scariest thing that can happen in `git`. But you know we engineered this conflict to happen and we are not going to panic. (If you do want to panic, try `git rebase --abort` as suggested and when you collect yourself and settle down, come back to try the `rebase` again.)

Your hint about where the conflict lies is in the line

```
CONFLICT (content): Merge conflict in recommendations.txt
```

Details on resolving conflicts are in Chapter 5 so head there for details on what to do now, specifically see the instructions for a "rebase conflict" in List 5.0.1. Briefly, Bob will open `recommendations.txt` in his editor and see the two different second paragraphs adjacent to each other, with Alice's official second paragraph first, and his proposed second paragraph afterwards. Now there is no notion of official or not, Bob and Alice are equals. But Alice got there first, so Bob has the responsibility to decide which paragraph to keep, or to keep both, and in what order. If Alice does not like Bob's decisions, they can do another round of changes (perhaps after some discussion on a GitHub issue, including commit hashes in the discussion to reference the details of any disagreement).

Done resolving the conflict, Bob's `recommend` branch now comes off the tip of master and is positioned for a fast-forward merge that he can then push to `origin/master`.

```
bob@laptop:~/publications/banking-paper$ git show-branch
! [master] Recs by Alice
 * [recommend] Bob recommends
--
 * [recommend] Bob recommends
+* [master] Recs by Alice
bob@laptop:~/publications/banking-paper$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
bob@laptop:~/publications/banking-paper$ git merge recommend
Updating b7abe18..3bd7a4a
Fast-forward
 recommendations.txt | 4 +++-
 1 file changed, 3 insertions(+), 1 deletion(-)
bob@laptop:~/publications/banking-paper$ git push
Username for 'https://github.com': bob-smith
Password for 'https://bob-smith@github.com': xxxxxxxx
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 346 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/alice-jones/banking-paper.git
   b7abe18..3bd7a4a  master -> master
bob@laptop:~/publications/banking-paper$ git branch -d recommend
Deleted branch recommend (was 3bd7a4a).
bob@laptop:~/publications/banking-paper$ git show-branch
[master] Bob recommends
```

Of course, now Alice does not have the edits Bob made when adding his recommendations and resolving the conflict in the rebase. But by now this should be routine. Alice just pulls from `origin/master` and is up-to-date. Now, all three repositories have exactly the same changes on their respective `master` branches.

After these two exercises, Alice and Bob have a joint paper that they are ready to upload to the arXiv. They can **tag** the final commit with the identification information and revision number the arXiv assigns, they can digitally **sign** the tag with their GPG keys, attesting to the genuineness of the work (Section 8.4). Because the commit hashes are chained, any tampering with any of the commits will break the digital signature. The commit hash for this tag could be good information to include at the arXiv site as well.

Notice that Alice and Bob only had to communicate about whose account to use for hosting the paper and on decisions about content. Never did one have to wait on the other, nor did they pass around files with version numbers. And since Alice pushed her recommendations to the definitive repository first, it became Bob's responsibility to edit a conflict locally before he could push his idea of how the recommendations should go.

To summarize, the drill is always the same. Edit locally on a branch, update your master by pulling from master at the definitive repository, rebase your branch on master, while resolving any conflicts, merge your branch into master (fast-forward), push master to the definitive repository. We call this the Eight-Step C.0.1 and you can find a general version in our quick-reference appendix, Appendix C.

There is just one more general `git` concept left, the pull request, so continue on to Chapter 4 to see how to open up your collaborations to the long tail of contributors from the entire world.

Chapter 4

In Control

We now turn to the situation where your book may have numerous collaborators. Maybe your book is an anthology of short stories with thirty contributors, or maybe it is a human anatomy textbook with the potential for students to discover small errors for many, many years down the road. If the anatomy textbook has four authors, you might begin as colleagues as in Chapter 3, but later may have specialists reviewing and editing select chapters, until finally the book is published and students suggest changes and corrections for the inevitable Second Edition. `git` is flexible enough to help you at every step.

So the model here is a select few that are in control of the repository. They might be called authors or editors, but we will refer to them generically as **overlords** (WordNet: "a person who has general authority over others"). In our opinion many projects should be large, because you want to encourage people to contribute.

4.1 Creating a Pull Request

A **pull request** is a way for a contributor to provide changes to your project, such as new material, suggestions for changes, or corrections, in an "embargoed" way. This is perfect when the collaborator is a junior partner in your project, or even a total stranger. As an overlord you retain control, and as a contributor you have total freedom to craft your contribution exactly as you think it should be, and to make it extremely easy for an overlord to accept. For the rest of this section, we will stop supposing the project is yours, but instead, we adopt the point-of-view of a collaborator. Here is the suggested setup.

- As in Chapter 3 there is one "definitive" version of the project as a `git` repository. It resides on GitHub (see Appendix B). One or more people have control over the definitive repository (the overlords). You do not need to know who the overlords are. Everyone's goal is to have their changes become part of the definitive repository.
- Every person who wants to contribute to the project has a **fork**, which is a `git` repository under their own personal account on GitHub. (Those mysterious people who have control over the definitive repository: they also have their own forks in their own GitHub accounts.) A fork is a copy of the definitive repository, and your fork is completely under your control.

- Everyone who wants to contribute to the project also has a copy of their fork on their own computer. (We now know of three copies of the project: definitive, your fork on GitHub, your fork on your computer.) We will refer to this as "the version on your laptop," although your computer might be a desktop machine, or something in the cloud, or something else. (The technical term for "version on your laptop" is **clone**, but we will not use that terminology. Keep in mind that the version on your laptop was copied from your fork, not from the definitive repository.)
- The repository on each contributors's individual laptop knows about two repositories on GitHub:
  - Their own fork, which is called `origin`.
  - The definitive repository, which is called `upstream`.

  Both of these repositories are known and managed as **remotes** within the repository on the laptop.

To summarize, everyone has a repository on their laptop. Everyone has the same `upstream` remote: it is the definitive repository. Everyone has a different `origin`, which is their own personal fork (a repository) in their account on GitHub. Notice that in Chapter 3 the `origin` remote was the definitive repository, but now `origin` is a sort of intermediary between the repository on your laptop and `upstream`, the definitive repository. (See Appendix B for details on forking a project on GitHub and setting up your laptop with a copy.)

Once all of that is set up, you go through the exact same cycle every time you want to contribute changes to the project. You already know how to work with branches, and their usefullness in a personal project, or with a small group. Chapter 6 is devoted entirely to how you can work effectively with a branch.

In a big complicated project, like a calculus textbook, there are lots of different things that may need attention. Maybe you need to put in the solutions to the problems from Chapter 6, or maybe you need to add a new section on the chain rule to Chapter 4, or maybe you need to edit the introduction to Section 5.3.

With branches, you know that you could use your repository to productively work on all three tasks, switching between them at will. You would have three different branches, `solutions-chap-6`, `chain-rule-section`, `intro-section5-3`, and in each branch you would be doing something a bit different.

Why is this good? There are many reasons, two of which are: (i) You do not need to mess up the working version of the book, because you are just working on a copy, and (ii) independent changes can be evaluated independently. Item (ii) suggests the following principle.

Principle 4.1.1 Pull Requests Separate Creation and Approval. *The process of* suggesting *changes to the definitive repository of a project is separate from the process of* accepting *those changes.*

This may seem silly if you are thinking in terms of a single person writing a book. But if the writing and editing is a group effort, and many people are contributing, it makes perfect sense for all changes to require at least two people: the person who wrote the new material, and the person who agreed to add that material to the definitive version. Notice that the sections of this chapter are organized exactly according to this principle.

So, you make a branch when you are about to start working on some aspect of the document (Principle 2.2.3). You may work on that branch for just an hour, or for several days, or off-and-on for months. You may switch to working on other branches. If it sounds confusing at first, it will not be confusing once

you start doing it, and it is totally worth it. For example, suppose you have finished making the solutions to the problems for Chapter 6. Good, because now you can propose those changes to be included in the definitive repository, and it is no problem that you have not yet finished the other work you are going, because those tasks are on different branches. And if the overlords review your work promptly, that is nice, but if it takes them some time to do that, you are not hung up waiting for their feedback before you can go back to to editing your new section on the chain rule. Juggling several writing tasks has just become a whole lot easier.

One last bit of terminology: proposing that the changes in a branch go into the definitive version is called a **pull request**. As in, "I request that the overlords pull my changes into the definitive repository." Once the authorities accept your pull request, the changes from your branch are now incorporated into the definitive repository and part of the official version of the project.

Now that we know the purpose and workflow of a pull request, and how to setup our laptop, let us describe the procedure of making a pull request. Notice that we are leveraging what you already know about branches from Chapter 2 and what you know about pushing and pulling changes from Chapter 3. You will go through the following recipe every time you want to contribute to a project where an overlord will review your contribution.

List 4.1.2 Creating a Pull Request

1. `git checkout master`

   `git pull upstream master`

   Remember that `upstream` is the name your laptop has for the definitive repository. This checkout and pull will update your local `master` branch with all of the latest changes that constitute the official version of the project.

2. `git branch <branch_name>`

   Initate a new branch, named `branch_name`. Previously your branch names were disposable, known only to you. They will now becone part of the public workflow for you and the overlords, so concoct something short but descriptive. (You can change this name *before* making it public, Chapter 6.)

3. `git checkout <branch_name>`

   Switch to your new branch, where you are going to work (Principle 2.2.3).

4. `git add <file1> <file2> <file3>`

   `git commit -m "An informative message"`

   Edit files, and accumulate new commits, just like before. When your branch is just the way you would like it to look, it is time to go public. Ready?

5. `git pull upstream master`

   While on the `<branch_name>` branch, you are pulling in any new commits from the master branch of the definitive repository and merging them into your working branch.

There is the very real possibility of a conflict in this merge. But you would much, much rather discover this yourself now, in the privacy of your own laptop, instead of an overlord getting all geared up to evaluate your public pull request and being immediately stymied by discovering there is a merge conflict that will simply be handed right back to you to resolve. When this happens, the overlord is only going to let you know about it, and then head off to evaluate somebody else's pull request.

When you do have a conflict in the merge, head out to List 5.0.2 in Chapter 5 for instructions on resolving it. Now you have done all you can. Your work is all packaged up nicely and consistent with the absolute latest changes in the definitive repository, so you are ready to go public.

6. `git push origin <branch_name>`

   Remember that `origin` is the name your laptop has for your fork of the project in your GitHub account. Now your fork has a new branch, which you would like to see also incorporated into the definitive repository.

7. Go to your fork on GitHub.

   Use your browser to visit your GitHub account, and locate your fork of the project. GitHub knows that you just put a new branch on that fork, so you should see a prominent green button on the right which says `Compare & pull request`.

   Click that button and you will see a page with a text box where you can describe the changes you have made. It is considered polite to describe where those changes can be seen (in the final product, not in the source files) so that the reviewer can take a look at the effect of your changes. For example, say something like "New exercises for Chapter 6, see page 88."

   Click the green `create pull request` button and you are done. Easy, since GitHub knows who the overlords are and has already been in touch with them about your changes.

8. `git checkout master`

   `git pull upstream master`

   You are ready to start on something new while you wait on the overlords, so these two commands will reset the repository on your laptop to be ready to begin on something new, and update your repository with the latest changes from the definitive repository.

What might happen next? Hopefully someone with control over the definitive repository will accept your pull request. But maybe they want you to make some changes; perhaps they found a typo or other error. No problem: just checkout your branch again on your laptop, edit to make the corrections, save those changes as a new commit on the branch, and push your branch to your fork (`origin`). This is the same recipe as in List 4.1.2, except that you skip over the command that actually creates `branch_name`, since that branch already exists.

Your pull request will be automatically updated and the overlords notified.

Notice that we are not changing any commits, just adding to, and extending the branch. And anybody can look and see the record of your original proposal and your response to the overlord's request. There is a complete record of exactly what decisions were taken and `git` and GitHub make the mechanics of the collaboration between contributor and overlord very simple.

It may happen that you (the person who created the pull request) are also a person who has control over the definitive repository, i.e. an overlord. In that case you can actually go to the definitive version on GitHub and accept your own pull request. However, many projects adopt the policy that people are not supposed to accept their own pull request, so would frown on this. This sort of policy is a best practice, but is not enforced by `git` or `GitHub`, it is a social construct of the organization of your project.

Even if you never aspire to be an overlord, you will find reading the next section is helpful, since it is always a good idea to see what it is like when the shoe is on the other foot. It also explains what you should do as a contributor when the branch in your pull request has a conflict with the `master` branch due to some changes that were introduced to `master` between the time you pushed your branch and an overlord got interested in an evaluation.

4.2 Reviewing and Accepting a Pull Request

We now change our point-of-view, to assume that you, dear reader, are the owner of the definitive repository, or have control over a definitive repository. In other words, you are an overlord now, with responsibility for the official version that will be produced from the definitive repository. Other people want to contribute to your project, so they create pull requests that GitHub notifies you about. The pull request is GitHub's way of letting other people contribute to your project, while still allowing you to have complete control.

Here is an annotated description of the process of reviewing and accepting a GitHub pull request.

Locate a Pull Request on GitHub. GitHub has likely sent you a notification immediately after a contributor has created a pull request, and maybe that email has a link to take you directly there. But you also need to know how to find any given pull request later.

Go to the project's home on GitHub and see if there are any pending pull requests. Note that you are going to the definitive repository, not your own personal fork of the project. You have your manager-hat on now, not your contributor-hat.

It is common to see seven tabs across the top of the repository on GitHub:

```
Code  Issues  Pull requests  Wiki  Pulse  Graphs  Settings
```

The first three and the last one are used most often. For `Issues` and `Pull requests` there will be a little number telling you how many items need your attention. Click on `Pull requests`. Each pull request will have a title, and it will tell you who made the pull request and when they created it. Click on one of the pull requests.

Initial Review of a Pull Request. If the contributor did a good job, there will be a few sentences describing what they did, and an indication of how their proposed changes appear in the final product, such as "New exercises for Chapter 6, see page 88.". Let us assume the contributor gave a clear description, and what they describe sounds like something beneficial to your project.

Below the description you *should* see a check mark in a green disk, and the phrase

```
This branch has no conflicts with the base branch
```

What does that mean? A **merge conflict** occurs when two people modify the same part of the same file. This could happen, for example, if a definition had awkward wording, and two people decided to fix it at about the same time. If they both submit pull requests, and you accept one of the pull requests, then the other pull request will cause a merge conflict because there is no automated way to figure out how to apply the second set changes since the original context has been destroyed by the application of the first pull request. (In the unlikely case that both people use *the exact same wording* then there would not be a merge conflict.) Another common scenario is when two people add new exercises to the end of an existing list. Once you accept one of these pull requests, there is no automated way to determine where to put the second set of new exercises. Should the second group of exercises go *before* the first addition, or *after* the first addition? `git` cannot help you here, and therefore will not help you.

Another likely explanation is that the pull request has a branch with a long lifetime. It was created as a branch off the tip of `master`, but that was a long time ago. The contributor added commits at a leisurely pace, and the overlords took their time in getting to their pull requests (always a bad practice for an overlord, you want happy contributors!). During this time, conceivably many new commits have been added to the `master` branch and somewhere there is an unfortunate overlap with the contributor's edits or their context.

So a merge conflict requires a human to figure out how to **resolve** it. Usually it is simple: just look at both versions and choose one, or the other, or come up with some happy medium. Simple for a person, but impossible for a machine.

When GitHub says there is a merge conflict, it is because GitHub has done a trial test of merging your branch in the pull request into the `master` branch of the definitive repository and the trial ended badly. The usual thing to do is scroll down to the next item on the page, which is a comment box. Write a brief message to the contributor saying there is a merge conflict, and click the `Comment` button and an email will be sent to the contributor. You should expect the contributor to resolve the conflict (see next paragraph). Occasionally you may choose to fix a small conflict yourself, but let us not worry about that now.

The contributor can resolve the conflict by doing

```
git checkout <branch_name>
git pull upstream master
```

So the contributor is on the branch they offered in the pull request, and it is important to get this part right. Then the contributor pulls commits from the `master` branch of the definitive repository. Several things happen as a result. First, the contributor gets the very latest commits from the `master` branch of the definitive repository, so it very up-to-date. Then `git` will try to merge `master` into `<branch_name>`. This is going to end the same way as it did in `git`'s trial: badly. But we expected it, were prepared for it, and `git` will do everything it can to help us.

The merge of the commits pulled from master, into the contributor's branch *will* cause a merge conflict. See Chapter 5 for the general procedure for resolving a merge conflict. Once that is done, the contributor can now push their

branch to their fork at `origin` and GitHub will notify the overlords that the updated branch is ready for review.

Note: there is no need to click `Close pull request`. Only do that when you really intend to reject the request. And always leave a message so that it is clear you did not close the pull request by accident.

In-depth Review of a Pull Request. Now you are looking at a pull request with a description that sounds useful, and there are no conflicts. The next step is to look at the actual changes the contributor made and their effect on your project. There are three tabs below the title of the pull request:

```
Conversation    Commits    Files changed
```

Click on `Files changed`. Highlighted in red and green will be the lines which were deleted and added (respectively) as part of this pull request. You need to look carefully at what was written, because this is destined to become an official part of your project. All large successful projects have standards for writing the source material, and you should check that the author has done a good job. Suppose, for example, you see the following line added to your calculus textbook:

```
When finding a maximum, be sure to check {\em both} end points.
```

The author has not used the LaTeX markup language in the best way, so it would be reasonable to click back to the `Conversation` tab, and in the comments box put something like

```
Use \emph{...} instead of {\em ...} for emphasis.
```

It is a good idea to look through all of their changes and submit multiple comments (in the same comment box), otherwise both of you will become annoyed if you have to go back and forth several times.

Assuming you have looked through the contributor's changes and the format and content seems to look okay, now you need to actually check that their contribution performs as claimed. If the project is code, that means you need to compile and run their code. If it is a book, you need to produce the book with their changes and see that the output looks good. Here is the procedure for doing a preliminary review within your fork on your laptop (this is your personal sandbox) and then actually incorporating the changes into the definitive repository.

List 4.2.1 Reviewing and Accepting a Pull Request

1. `git checkout master`

   `git pull upstream master`

   You are preparing your fork on your laptop and updating to the most recent version of the definitive repository. You should be in the habit of doing that whenever you are about to start something new, so it shouldn't be a surprise that you are supposed to do that now.

2. Click back to `Conversation`, and next to the big green `Merge pull request` click on the blue words `command line instructions`. If the GitHub user `fredstro` made a pull request for your `calculus` repository as a branch named `solutions-chap-6`, then these commands would look similar to:

```
git checkout -b fredstro-solutions-chap-6 master
```

```
git pull https://github.com/fredstro/calculus.git
solutions-chap-6
```

You want to run these commands for your fork on your laptop– you could just cut them from your web browser and paste them into your terminal session.

As a result you will now have a branch in your laptop's fork named `fredstro-solutions-chap-6` with all of the changes `fredstro` is proposing. Now you can do a thorough check on the pull request. Produce your book, or run code, or whatever is appropriate for your project. Examine the output to see if the changes performed as expected. Notice that you have not endangered the definitive repository in any way, and eventually you can just delete the branch from your fork.

3. If the contribution fails to pass your discriminating review, then leave a comment under `Conversation` and the person will respond with further changes. That scenario is quite common: people often forget to actually verify that their changes behave as claimed before submitting a pull request. It can also happen that the overlord evaluating the pull request fails to actually check everything before accepting it. Then the real blame lies with the overlord who accepted the request, because they are in a position of responsibility for the project. So be very careful if you accept a pull request after only glancing at the `Files changed` on GitHub. Once the changes go into the definitive repository, you and your collaborators are responsible for them.

4. If everything looks good, then it is easy. Maybe too easy! Go back to the definitive repository on GitHub and click the big green `Merge pull request`. Modify the commit message if necessary, and then click `Confirm merge`. GitHub may then say something about deleting a branch which is no longer needed: that is a fine thing to do.

5. Note that your laptop is still on the temporary branch you created to check that pull request. So you should `checkout master` and `git pull upstream master`. The `master` branch on your fork on your laptop will now have the changes from the pull request you just accepted.

   If you handle a lot of pull requests you will acquire a lot of branches which are no longer needed. We have seen earlier how to delete branches. The temporary branches you make when you evaluate a pull request have distinctive names, which makes it easy to identify and delete them.

That is it! More than ninety percent of pull requests can be handled with those simple steps. If you encounter a complicated situation, seek help from an expert.

Large projects use a model similar to this one. Sage, an open source computer algebra system, has hundreds of contributors with a single **release manager** as the overlord. Reviews of proposed branches are distributed among the contributors themselves, so if the release manager is familiar with the reputations

and expertise of members of the community, it can be feasible for a single person to roll up fifty or so contributions on a weekly basis and still maintain the integrity of the project, and their sanity.

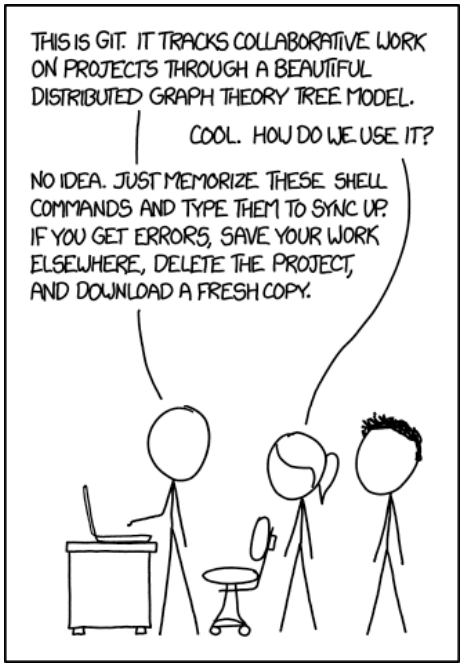

Figure 4.2.2 xkcd "Git" (used with permission)

Notice that with a large project, there are many people working asynchronously and the potential for conflicts is very real as the mainline branch may advance rapidly from where a contributor initiated it. In the Sage project, reviews are not assigned, and are voluntary. A diligent contributor might regularly merge the `master` branch into their proposed branch, creating merge commits every time, with or without any necessary edits, which are then pushed to the public equivalent of GitHub. In this way, the contributor's proposed branch is always consistent with a relatively recent version of the definitive repository. Due to this, and to the public nature of the commits on a pull request, there will be many merge commits in such a repository. The network of commits begins to look less like a tree, and more resembles a slow, broad, meandering river with many branches forming, coming back together, splitting again, reforming,… Mathematicians refer to this network as a directed acyclic graph, which you may see described as "the DAG". There is always an unambiguous path through the commits back to the original root commit.

If you have ever tried to collaborate on a writing project using email with attachments of traditional word-processing files (with or without "track changes"), then you may already be starting to appreciate how `git`'s branches and remotes, along with GitHub's pull requests and notifications, can free contributors and overlords to collaborate efficiently and concentrate on content with very little procedural friction.

Chapter 5

Merge Conflicts

How-to for resolving a merge conflict

git mergetool, meld

We present similar three-step recipes for a resolving conflicts that occur either during a rebase, or during a (non-fast-forward) merge.

List 5.0.1 Resolving a Rebase Conflict

1. In the output announcing the conflict, `git` will list exactly which files have merge conflicts. Open these files in your editor and search for `========`. Above and below a this line of equal signs you will see the text that `git` cannot resolve itself delimited with hints on where it comes from. Edit freely to make the text look the way you want it, and remove the markers: `<<<<<<<`, `========`, `>>>>>>>`.

2. `git add <file1> <file2> <file3>`

   You need to stage your changes in the index in preparation for a commit, as usual. At this point, `git status` will include

   ```
   (all conflicts fixed: run "git rebase --continue")
   ```

   so go ahead. Notice that you are not given an opportunity to change the commit message, and maybe your edits make this desirable. This can be done later, see interactive rebasing in Chapter 6.

3. `git rebase --continue`

   You have modified the commit that did not rewind smoothly, and so this command tells `git` to continue replaying commits from the branch, including the one it is in the midst of replaying. The remaining commits may apply smoothly, or they may present new conflicts. So you may go through this recipe several times.

List 5.0.2 Resolving a Merge Conflict

1. In the output announcing a merge conflict, `git` will list exactly which files have merge conflicts. Open these files in your editor and search for `========`. Above and below a this line of equal

signs you will see the text that `git` cannot resolve itself delimited with hints on where it comes from. Edit freely to make the text look the way you want it, and remove the markers: `<<<<<<<`, `========`, `>>>>>>>`.

2. `git add <file1> <file2> <file3>`

   You need to stage your changes in the index in preparation for a commit, as usual.

3. `git commit -m "Fixing merge conflict by..."`

   You now create one new commit, a **merge commit**. Remember, in a merge no commits change in any way. But here you do create an additional new commit that is slightly different in nature. It holds the changes you made to resolve the conflicts, but unlike other commits, it has two parents, not one. These are the tips of the two branches that are being merged. This what makes a regular merge different from a fast-forward merge. In `show-branch` it will be shown distinctively as a dash, not an asterisk or exclamation point.

Chapter 6

(*) Branch Management

Reference a principle, always working on a branch
- interactive rebase, total squash and rebuild commits, grafting
- relative references, merge commits, etc, seen already in solo chapter
- stash, "I'll just be a minute"
- branch rename, prior to public PR
- cherry-pick
- Rename a branch
- don't rebase through branch points, might lose one
- Fancier diff commands to see whole cumulative branch

Chapter 7

(*) Oops!

Sections with titles describing problems, content is fixes. General overarching advice (stop quickly, diagnostics,directory backup, reflog).

7.1 That is So Messed Up

Suppose you have really made a mess of some file, and you do not even really know just how it happened. No problem. With a clean working directory, you can `git checkout <ref>` where `ref` is any reference you can think of to refer to a certain commit anywhere in your repository. This reference could be a branch pointer, like `master` or `heroine`, or it could be the leading digits of a commit hash that you locate in the output of `git log`, like `39b56cc9`. This will convert the working directory to some previous state, where your file has what you want.

Now copy the acceptable version of the file in question to someplace outside of your repository. Then something like `git checkout giraffes` will bring you back to where you were. Now copy the file from outside the repository on top of the messed-up one. To `git` the changes will look indistinguisable to you typing at the keyboard to fix everything. Commit these changes in the usual way as a single changeset and get back to your writing.

Your history will contain a record of whatever mistake you made, but at least the collection of commits you had created in `git` allowed you to go back and re-create something acceptably close to your original work. Note that commiting frequently gives you greater latitude to pick just the right commit for the recovery.

Of course, if the damage is extensive and several files are affected, you can do the copy and replace with each one.

Chapter 8

Git Miscellany

This final chapter is a loose collection of tidbits that you may find useful, without getting too technical or arcane. We have not tried to cover everything, and we may have told you a few white lies along the way. When you exhaust what we have here, re-read Chapter 9, and send us a pull request if you have something useful (and not too arcane) for this chapter.

8.1 (*) Word Diff

```
git diff --word-diff
```

8.2 (*) Impersonating a Commiter

8.3 The Stash

The **stash** is a **stack**, a last-in, first-out arrangment, like a stack of plates in a warmer in a cafeteria. At any time you can **save** your changes in a dirty directory with the command `git stash save` and they are removed from the working directory, thus making the directory clean again, and are saved as a changese in the stash. This great if you just want to wander off some place else in your repository and you will be right back to your work-in-progress. So we talk of "stashing your changes" as a temporary measure.

When you are a beginner, the stash is very alluring. Resist the siren call. It is not a cure-all and there is usually a better way to juggle two things at once. Remember the three branches of Checkpoint 2.2.1? Here are some notes (which will not explain everything carefully). Since the stash manipulates the working directory all at once, you have less control than you do with commits, even if the stash seems less permanent.

- `git stash save -- "Reworking turtle chapter"` will include a message with the entry you place on the stack. Without it, you get a cryptic automatic message that includes the initialism `WIP` (work-in-progress). If you will be away more than ten minutes, a message is a good idea.

  This is like putting a plate into the warmer, perhaps with a special note scribbled onto it.

- `git stash show` will show you the diff of the changeset on top.

  This is like looking real closely at the top plate.

- `git stash list` will show you all the changesets currently in the stash, so it helps to provide good messages if you have many.

  This like like reading the notes on all the plates in the warmer.

- `git stash pop` will put your changes back into the working directory and remove the entry from the stack. It is up to you to make sure you are on the branch you want to be, and that your working directory is in the right state to accept these changes (or you may end up starting a merge you may not want).

  This is like taking a plate out of the warmer.

- `git stash apply` will take the changes on the top of the stack and put them back into the working dirtectory, but it will *leave the original entry on top of the stash.* RAB once ended up with duplicate copies of a set of changes in his working directory. He thinks he did an `apply` and subsequently did a `pop`. Maybe.

  This is like magically duplicating the plate on top, and removing it, leaving the original still in the warmer.

- `git stash drop` will remove the changes on top of the stack and throw them away. This can actually be very useful. You may `add` and `commit` a variety of changes to your branch, but still have some paragraphs you do not really like, and decided not to use, still polluting your working directory. Simple. Move the changes to the stash and immediately delete them, no message needed. Careful, think twice, you really are deleting changes, though they may be recoverable with advanced techniques.

  This is exactly like taking the top plate out and dropping it on the floor so it breaks into many unusable pieces.

- There are many more actions you can take with the stash, but the above should be sufficient for intermediate `git` use. Consult the usual sources for more advanced use. Note that prior to the time around `git` version 1.6.0 changesets in the stash expired, but it appers that behavior has changed. So don't panic if you see older posts that speak of expiration.

8.4 (*) Tagging Releases, Signing a Repository

8.5 (*) Who Did What, and When?

blame, log with diff-filter, "When a file was added to the repo",

8.6 (*) Where Did it All Go Wrong?

git bisect

8.7 (*) File Management

rm, mv

8.8 (∗) Binary Files

8.9 (∗) Everything: The reflog

8.10 (∗) README on GitHub

8.11 (∗) Log Display Options

`--pretty=oneline`, `--graph`

Chapter 9

Parting Shot

There are often many ways to accomplish the same thing in `git`, and some will be easier than others. You can find lots of advice on the Internet, some of it is even good. Sites where answers get upvoted or downvoted are often useful. As you gain a good understanding of the basic principles of how `git` works and the job it has been designed for, you will get better at locating and evaluating suggestions. We have tried to give you a headstart on that basic understanding. Take notes when you find good stuff that works for you. (Much of the later bits and pieces here are inspired by our own notes accumulated through our first three years of gaining valuable experience with `git`.)

Beware of dogma. Some will say you should *never* rewrite history–we do it *all* the time. Others will say you should always do fast-forward merges and obtain a perfectly linear history–impossible on a big public project. Use your independent judgement, and always remember the first of our principles, Principle 1.0.2, `git` is just a tool.

But most of all, have fun and don't panic!

Appendix A

Getting Started with Git

We generally prefer to use a large collection of very sharp tools to get our work done, and it is no different with `git`. There are pretty point-and-click programs that are suppose to make it easier to use and understand `git`, and even GitHub will offer to do certain tasks for you. In our limited experience, we just find these interfaces at best confusing, and downright impossible when things go south. Thus our decision to teach you how to interact with `git` at the command-line.

As it is, command-line `git` is already an interface to even more primitive commands. What you and I are using in these exercises are known as the **porcelain** commands, while the primitive versions are the **plumbing** commands. You can work out the metaphor. You can find tutorials that walk you through a sequence of plumbing commands to effect one of our porcelain commands like a commit or merge, and you can learn a lot from the exercise. (see [3, Chapter 10].) Don't panic if you see some terms below that you are not yet familiar with, they will be explained eventually.

The information in this appendix is accurate as of 2016-04-10. Corrections and updates are greatly appreciated. How about as a pull request?

A.1 Installing Command-Line Git

We are not going to be much help here, as we are not going to try to cover all the bases. Use your operating system's package manager or other tools to install command-line `git`. A good place to start is the download area of the canonical `git` site [1], or see the *Pro Git* book [3, Section 1.5].

You can verify a successful installation with:

```
rob@lava:~$ git --version
git version 2.5.0
```

Your version number will likely be different.

A.2 Configuring Git the First Time

This subsection has some one-time details you need to work through before using `git`.

First you need to identify yourself as the author of commits. Even if your repository will be private for a while, someday you may open it up to collaborators, and you would rather not go back and edit *all* of your commits.

Fortunately, `git` makes it easy. Use your real name, and use an email address that you expect to own for a long time. Together, these two pieces of information should identify you across all the repositories you will ever contribute to, and across all the repositories ever made.

```
rob@lava:~$ git config --global user.name "Robert A. Beezer"
rob@lava:~$ git config --global user.email beezer@pugetsound.edu
```

Notice that I use different values in practice, rather than the example here. In the earlier days of electronic mail, my institution used the domain name `ups.edu`, often confusing us with the shipping company, `ups.com`. So I prefer to use the older form that I was known by in other settings for many years. Notice the quotes around my name, in order to include the spaces as part of the text of my name.

`git` is inclined to drop you into an editor for various tasks. We try to avoid this, so often show you the `-m` switch, followed by a short quote-protected message you can supply as part of a command. But you will need to use an editor for various tasks (such as an interactive rebase) so you do not want to suddenly find yourself in a system-default editor (like `vi` or `emacs`) that you do not understand. I know the basics of an enhanced version of `vi`, known as `vim`, so I use the command-line name of this program in the setting below.

```
rob@lava:~$ git config --global core.editor vim
```

There are other settings, and you can check yours with `git config --list`. One last thing. If your version of `git` happens to be prior to version 2.0, then check your configurations for the value of `push.default` and ensure that it is `simple`. It will be important later.

More details, and especially for Windows systems, can be found in the *Pro Git* book [3, Section 1.6]

A.3 Initializing a Git Repository

To begin a new repository, create a directory where you want your files to live. We have been using `~/books/ppw` in our examples. Then make that the default directory in a command-line session.

```
rob@lava:~/books/ppw$ git init
Initialized empty Git repository in /home/rob/books/ppw/.git/
```

Naturally, there will be no commits in your repository.

```
rob@lava:~/books/ppw$ git show-branch
No revs to be shown.
```

But once you do make a commit, there will be a `master` branch and the root commit will get special identification as such. That's it, you are ready to go.

A.4 Ignore Temporary Files

You can save this subsection for later. When you run `git status` you will eventually become annoyed by intermediate or temporary files in your working directory that keep being reported as untracked, and worse case, they will obscure files you really need to know about. There is a `.gitignore` file (globally, or per-repository) where you can list filename patterns for `git` to ignore. You

can find lots of examples on the Internet (in addition to lots of custom configurations). I prefer not to use too much extra configuration, so that if I end up on an unfamiliar computer, commands continue to work as I expect.

Appendix B

Getting Started with GitHub

GitHub is a site that lets users host, and communicate with, their `git` repositories and the repositories of others. There are similar services, but GitHub appears to be the most popular, and so benefits from the network effects of a large number of users. You can make an unlimited number of public repositories, but must pay for private repositories. Here "public" means anybody can see the content of your repository and copy it, but you retain control over who can modify the repository.

The information in this appendix is accurate as of 2016-04-10. Corrections and updates are greatly appreciated. How about as a pull request?

B.1 Make An Account

This is routine, but we have one piece of advice. Choose your username with some thought. It will be part of the URL for your repositories and others will use it to identify you in discussions. It will also be part of a GitHub email address you may choose to use. And so on. If you are doing professional work, it might be best to use some variant of your real name, and if the base name of your canonical email is available, that might be the best. You will have a GitHub profile that you can use to disambiguate yourself.

Pick a strong password, since you would rather not have untrusted individuals changing your repositories, or worse, impersonating you on GitHub and destroying your hard-earned reputation.

B.2 Creating a New Repository on GitHub

Find a plus-sign button in the upper-right hand corner of a GitHub page, or maybe as a new user you see a big button for creating a new repository. This will take you to a set-up screen. You will give your repository a name. For books, we like initialisms (lower-case), so the book you are reading has the repository name `gfa`. Alice uses `banking-paper` in our first exercise. Like your username and commit messages, give this name some thought.

Fill in a human-readable description if you wish, and go ahead and tick the box for a README, which we will discuss elsewhere (Section 8.10). No need to mess with `.gitignore` now, and adding a license is your choice. You'll be taken to the main screen for your new repository and you will see much of the information you just provided.

B.3 Adding Collaborators to a GitHub Repository

In the seven tabs near the top, locate `Settings`, with the gear icon. This will take you to a page with five choices in the left sidebar. Go to `Collaborators`. Here you can search for and add a collaborator, who will then be able to add commits to your repository. So make sure you understand the trust you have placed in this person before you pull the trigger.

B.4 Copying a GitHub Repository (Forks, Clones)

For collaborators to work independently on a project, they need their own independent copy of a repository. On GitHub you can visit somebody else's repository and `fork` that repository. This will make an entire independent copy of that person's repository in your account (where you have permission to modify your copy), and GitHub will remember where your copy came from. In this case the copy is known as a `fork`. Yes, the word `fork` has just been used as a verb, "Cameron forked Drew's repo", and is also used as a noun, "Cameron's fork of Drew's repo." Making a fork of someone's project is a compliment, since it suggests interest, and the possibility of someday contributing back to the project, in the style of Chapter 4.

You can also, and often may prefer, to copy a repository to your local computer. You might like to write while on airplanes, on your desktop at work, or on your laptop at home in bed. Then you can even collaborate with yourself and keep all those repositories syncronized. So you could have copies of your project all over the place.

It is easy to copy a `git` repository generally, and dead-simple to copy a GitHub-hosted repository. To copy a GitHub repository, you need to know the URL. It is a bit hidden. On the main page for a project (yours, or somebody else's), locate and click on the green Code button, and the URL should be visible. You are looking for something like

```
https://github.com/alice-jones/banking-paper.git
```

which would fit our first GitHub example. Copy the URL to your clipboard.

Now open a terminal on your local machine where you can use the command-line. Navigate to a directory, where a new repository-specific directory makes sense for your work. In our first example, Alice might navigate to her existing `~/papers/` directory, where she expects to soon have a `~/papers/banking-paper` directory. OK, all set. Making a copy is known as a **clone**. (And a fork is just a special type of clone.)

```
alice@work:~/papers$ git clone
    https://github.com/alice-jones/banking-paper.git
Cloning into 'banking-paper'...
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
Checking connectivity... done.
```

The above may take a long time for a large repository (many large files, or a long history of many, many changes). The output above is for an essentially empty repository. Now verify the expected directory exists and has the expected files in it.

Appendix C

Quick Reference

Cheat sheets for common tasks

List C.0.1 The Eight-Step: Adding Commits to the Definitive Repository

We will describe the process of moving commits on a "topic" branch in a local repository to the master branch of a definitive repository where you have permission to push, and the repository is configured as a remote in your local repository. Eventually, the logic of this procedure will become second nature.

1. `git checkout master`

   Prepare to bring the local master branch up-to-date.

2. `git pull`

   Pull new commits from the definitive repository into the local master branch.

3. `git checkout topic`

   Move back to the topic branch (which will have a different name).

4. `git rebase master`

   Move the branch point of the topic branch to the tip of updated master. Resolve any conflicts which occur during the replay phase of the rebase.

5. `git checkout master`

   Move back to the local master branch in preparation for modifying it.

6. `git merge topic`

   Use a fast-forward merge to bring the topic branch into the master branch.

7. `git branch -d topic`

   The topic branch pointer is obsolete as it duplicates the master branch pointer.

8. `git push`

   Move new commits on the local master branch to the master branch of the definitive repository.

Appendix D

Cheat sheet for contributing to a project

You have forked a repository and cloned your fork to your laptop. You have `origin` as a remote to your fork, and `upstream` as a remote to the official repository for the project. Below are all the steps you typically need for contributing to the project.

List D.0.1

1. `git checkout master`
2. `git pull upstream master`

   Now you are on your master branch and have the latest version of the project.
3. `git branch <branchname>`
4. `git checkout <branchname>`

   Now you are on a new branch called <branchname>.
5. Make the changes you planned. When appropriate do

   `git add <file1> <file2> ...`

   `git commit -m "useful commit message"`
6. Check your work: run `latex`, or compile the code, or whatever is appropriate for the project. Save yourself the embarrassment of having a pull request rejected because you submitted something broken!
7. `git pull upstream master`

   Do that in case new material has been added since the last time you pulled. If you pulled in new material, it can't hurt to repeat step Item D.0.1:6.
8. `git push origin <branchname>`

   And then go to your fork on GitHub and make a pull request..

   Be sure to describe how to see your changes in the final product.
9. When in doubt, do `git status` to find out what branch you are on, and to see if there are uncommitted changes.

Appendix E

Cheat sheet for managing pull requests

You are one of the people in charge of a GitHub repository. Below are the typical steps to evaluate pull requests.

List E.0.1

1. Click on the `Pull requests` tab, and then choose a pull request.
2. If there are merge conflicts, write a brief message and click `Comment`.

   Now there is nothing else to do until you get an email either replying to your comment, or telling you that the pull request has been updated.
3. Click on the `Files changed` sub-tab, and carefully look at all the changes. If there are any errors or violations of the official style for the project, click back to `Conversation` and leave a helpful `Comment`.
4. If the changes look reasonable, then you have to actually check that the new material functions as advertised. Go to your laptop and do:

   ```
   git checkout master
   ```

   ```
   git pull upstream master
   ```

   Then click back to `Conversation` and click on `command line instructions`. Copy and paste those into the command line. They will look something like this:

   ```
   git checkout -b fredstro-solutions-chap-6 master
   ```

   ```
   git pull https://github.com/fredstro/calculus.git
   solutions-chap-6
   ```
5. Check their contribution: run `latex`, or compile the code, or whatever is appropriate.
6. If you aren't happy with what you see, leave a helpful comment.

   But if everything looks good, click `Merge pull request`, leave a comment if appropriate, and then click `Confirm merge`.

7. You are done, except don't forget to `git checkout master` and then pull in the new material.

Appendix F

List of Principles

Resources

[1] *Git Site*, https://git-scm.com/.

[2] *GitHub Site*, https://github.com/.

[3] *Pro Git*, Scott Chacon and Ben Straub, https://git-scm.com/book/en/v2.