Understanding Git (part 2) — Contributing to a Team

Understanding Git (part 1)—Explain it Like I'm Five

→ *Understanding Git (part 2)—Contributing to a Team*

Understanding Git (part 3)—Resolving Conflicts (stay tuned!)

Practically every major software company and open-source project uses some type of "version control" to track changes over time. Version control can help you coordinate teamwork and hunt down software bugs.

Git is one of the most popular version control systems today. See part 1 of this series for an "Explain it Like I'm Five" intro if you're not already familiar.

This guide covers how to use git to contribute to a team—whether at an office or online in the open-source community—and is intended for beginners. It'll explain the whole flow, from idea to Pull Request.

Any **jargon** will be highlighted in bold the first time it's described. Feel free to look up those terms in the official git glossary or reference guide for more details.

Starting from the trunk

Git uses a lot of “tree” analogies, so get ready for it. You can think of your main codebase as the **trunk** of a tree 🎄

Every time you add more changes (aka **commits**), your tree grows taller, straight up. Even if you delete code, that’s still considered a change and causes the tree to grow. It’s like how the “undo” history in a text editor saves your keystrokes, including backspace.

Git, by default, calls the trunk `master`. You can call it whatever you want; there’s nothing special about the word `master` other than that it’s conventional.

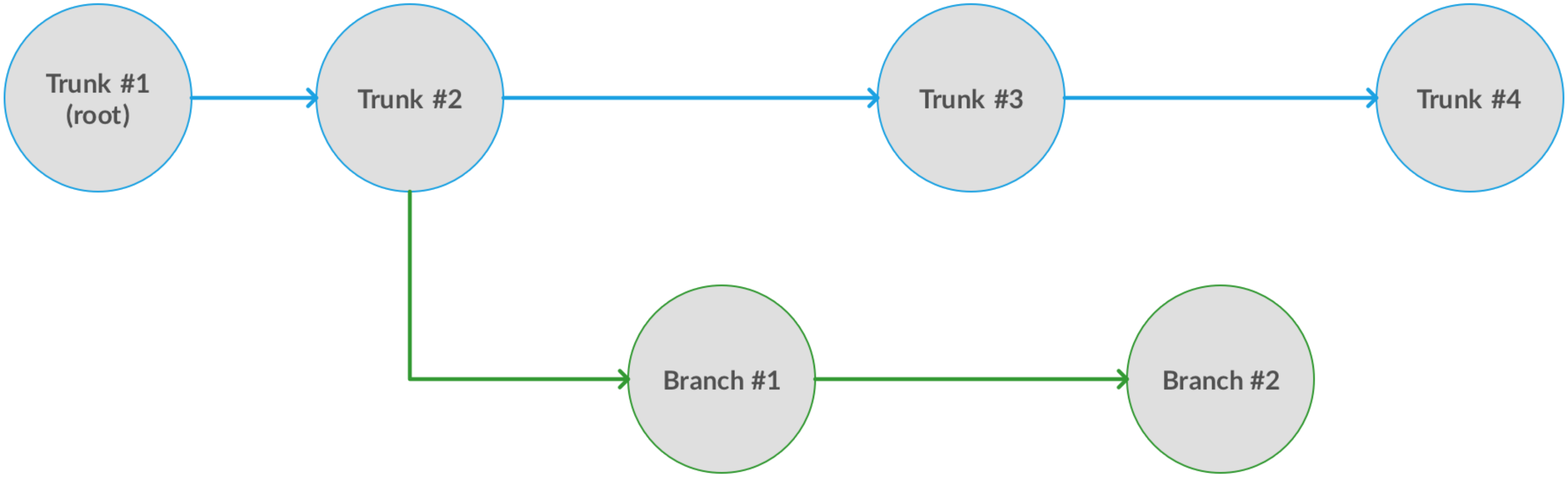
You can travel up and down the trunk—equivalent to going forward and backward in time—by **checking out** specific “checkpoints” as described in part 1.

Branching out

Most projects have a backlog of new features to add and bugs to fix. When you want to address one of these issues, one way would be to grow the tree taller and commit directly to the trunk (`master`). This works fine for small projects or projects where you’re the only person making changes, but what if multiple people are working at the same time? It’s too easy to get in each other’s way and end up with conflicting changes.

The solution is branching. Instead of committing to the trunk, you create your own personal **branch** (e.g. `my-cool-feature` instead of `master`) and work from there. Now you’re growing your *branch* taller instead of the trunk.

When visualizing branches, they’re often drawn sideways to save space. Imagine the following is an ugly tree, tipped over, with the roots on the left. Each circle is a commit. The farther to the right each circle is, the more recently it was committed:



I wouldn't plant this in my front yard.

There's a blue trunk and a green branch. There are several commits on each, shown chronologically from left to right. Your branch started at commit `T2`. While you were working on your branch (commits `B1` and `B2`), someone else worked directly on the trunk (commits `T3` and `T4`). Those commits aren't in your branch yet; your branch is out of date, which you'll read about soon.

Again, see part 1 of this series to understand the `branch` and `commit` commands, or see the free git book if you want more visualizations. You can also see real life branch visualizations on GitHub, or by typing `git log --all --decorate --oneline --graph` in your terminal.

Submitting your changes

Now you have your changes on a branch, but the eventual goal is to get them back onto the trunk as part of the “official” codebase.

Once you’ve tested your changes, you’ll need to share them with the team. Typically you’ll do this via a **pull request** (PR) or a **merge request** (MR)—they’re the same thing, the term just depends on what software you’re using (e.g. GitHub/Bitbucket/GitLab). You’re *requesting* that your changes be *pulled in and merged*. I’ll refer to these as PRs from here on.

Some people get nervous about making their first PR (I certainly did), but it’s really nothing to be afraid of. Most teams are happy to receive new PRs, even if the code needs a bit of work before being accepted. PRs are an important part of the open-source ecosystem.

Here’s a detailed description of exactly how to submit a PR, and some advice about PR etiquette. There’s even a whole community of “First Timers Only” for people who want to start contributing to open-source but don’t know where to begin.

The main thing to remember is to include a clear explanation of **why** you’re making the changes in order to give context.

Discussing and revising

Once you submit your PR, someone else on the team will need to look it over and leave feedback. They can ask questions and comment on specific lines of code, or they can give more general feedback about your changes. In some cases they may push their own changes directly to your branch, but usually they’ll ask you to make the changes yourself.

If you want to make changes based on the feedback, simply add more commits to your existing branch and push it to `origin` again. The PR will update automatically to reflect your changes.

Keeping up to date

If some time goes by before your PR is accepted, it might get “stale”, meaning it’s based on an older version of the trunk (like in the tree shown earlier). Your changes may have worked a week ago, but there’s no guarantee that they still work alongside other, more recent changes to the trunk.

To get up to date, you have two options:

1. You can “**merge** in” the changes using `git merge master`. This will apply any new changes from the trunk on top of your work.
2. You can “**rebase** on top of” the changes using `git rebase master`. This re-applies your work on top of any new changes on the trunk.

In either case, your own changes will still be the same—you’re effectively just moving your branch up to the top of the trunk to stay up to date with the latest code.

Different teams may prefer one method or the other; it’s usually safer to `merge` if you’re unsure.

Dealing with conflicts

When merging or rebasing, you'll occasionally run into **conflicts**. This means you changed a line of code that someone else also changed, and git doesn't know which version to keep.

When this happens, many people panic. The output from git looks really nasty, with weird `>>>>>>> ======= <<<<<<<` symbols, and you might think you've broken the whole thing. Don't worry! It looks strange, but with a bit of practice you'll understand it.

There's way too much to cover here, but check out this article to understand how conflicts work and how to resolve them. I also plan to focus on dealing with conflicts in part 3 of this series.

In essence, you just need to remove those weird symbols and manually combine the code in between them.

For example, the line:

```
print "Hello"
```

Might be changed on two separate branches, leading to the conflict:

```
>>>>>>>
print "Hello, world"

=======
print "Hello!"
<<<<<<<
```

Which, when resolved, might become:

```
print "Hello, world!"
```

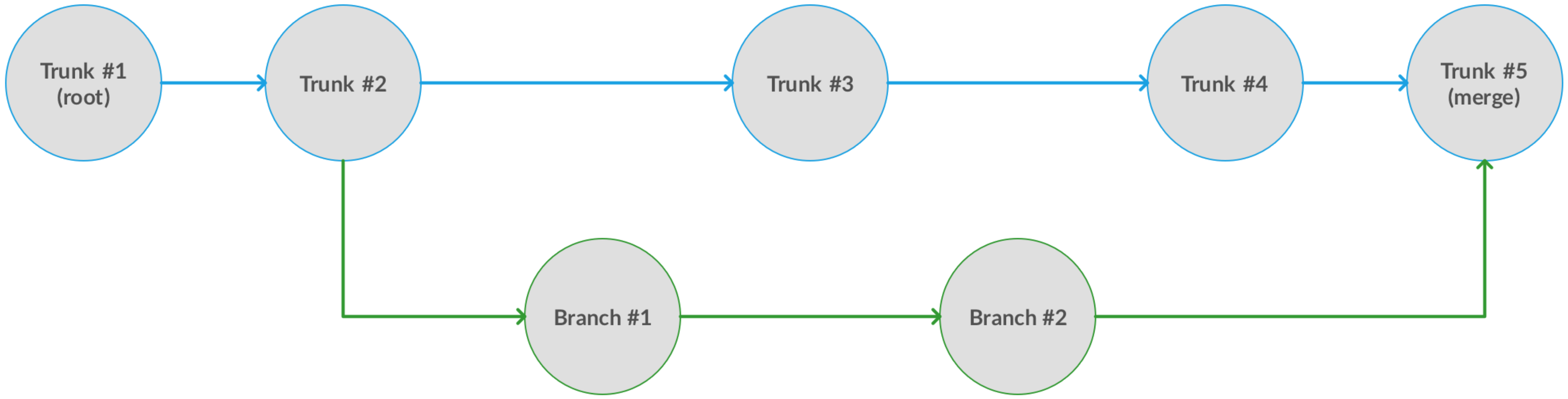
After manually merging the conflicting lines together—keeping both the “world” *and* the “!”, which had each been added separately.

Accepting your changes

Once all of the PR comments have been addressed and any conflicts have been resolved, your branch is ready to be merged!

An administrator of the codebase can accept the PR by merging your branch into the trunk—simply by pressing a button on GitHub—thus making your changes official.

Commit `T5` below is a “merge commit”, putting the changes from your green branch onto the trunk.



This isn’t how trees work in nature.

You’ve made a successful contribution! 🙌 You can view all the PRs you’ve ever submitted at https://github.com/pulls.

If you found this guide useful, please help by **tapping the 👏 button as many times as you’d like** so others can find it too.

Thanks for reading, and stay tuned for part 3. In the meantime, go watch this easy tutorial video and then tackle some First Timers issues!

2