**Git Commits**

A commit in a git repository records a snapshot of all the (tracked) files in your directory. It's like a giant copy and paste, but even better!

Git wants to keep commits as lightweight as possible though, so it doesn't just blindly copy the entire directory every time you commit. It can (when possible) compress a commit as a set of changes, or a "delta", from one version of the repository to the next.

Git also maintains a history of which commits were made when. That's why most commits have ancestor commits above them -- we designate this with arrows in our visualization. Maintaining history is great for everyone working on the project!

**Git Branches**

Branches in Git are incredibly lightweight as well. They are simply pointers to a specific commit -- nothing more. This is why many Git enthusiasts chant the mantra:

*branch early, and branch often*

Because there is no storage / memory overhead with making many branches, it's easier to logically divide up your work than have big beefy branches.

When we start mixing branches and commits, we will see how these two features combine. For now though, just remember that a branch essentially says "I want to include the work of this commit and all parent commits."

**Branches and Merging**

Great! We now know how to commit and branch. Now we need to learn some kind of way of combining the work from two different branches together. This will allow us to branch off, develop a new feature, and then combine it back in.

The first method to combine work that we will examine is git merge. Merging in Git creates a special commit that has two unique parents. A commit with two parents essentially means "I want to include all the work from this parent over here and this one over here, and the set of all their parents."

**Git Rebase**

The second way of combining work between branches is rebasing. Rebasing essentially takes a set of commits, "copies" them, and plops them down somewhere else.

While this sounds confusing, the advantage of rebasing is that it can be used to make a nice linear sequence of commits. The commit log / history of the repository will be a lot cleaner if only rebasing is allowed.

**Moving around in Git**

Before we get to some of the more advanced features of Git, it's important to understand different ways to move through the commit tree that represents your project.

Once you're comfortable moving around, your powers with other git commands will be amplified!

**HEAD**

First we have to talk about "HEAD". HEAD is the symbolic name for the currently checked out commit -- it's essentially what commit you're working on top of.

HEAD always points to the most recent commit which is reflected in the working tree. Most git commands which make changes to the working tree will start by changing HEAD.

Normally HEAD points to a branch name (like bugFix). When you commit, the status of bugFix is altered and this change is visible through HEAD.

Git checkout c4 , git checkout c2 – this will checkout from the commit and head will be pointing to that side.

**Relative Refs**

Moving around in Git by specifying commit hashes can get a bit tedious. In the real world you won't have a nice commit tree visualization next to your terminal, so you'll have to use git log to see hashes.

Furthermore, hashes are usually a lot longer in the real Git world as well. For instance, the hash of the commit that introduced the previous level is fed2da64c0efc5293610bdd892f82a58e8cbc5d8. Doesn't exactly roll off the tongue...

The upside is that Git is smart about hashes. It only requires you to specify enough characters of the hash until it uniquely identifies the commit. So I can type fed2 instead of the long string above.

Like I said, specifying commits by their hash isn't the most convenient thing ever, which is why Git has relative refs. They are awesome!

With relative refs, you can start somewhere memorable (like the branch bugFix or HEAD) and work from there.

Relative commits are powerful, but we will introduce two simple ones here:

* Moving upwards one commit at a time with ^
* Moving upwards a number of times with ~<num>

Git checkout c3

Git checkout c5

Git checkout HEAD, Git checkout HEAD^, Git checkout HEAD^^^ - this will create a

**The "~" operator**

Say you want to move a lot of levels up in the commit tree. It might be tedious to type ^ several times, so Git also has the tilde (~) operator.

The tilde operator (optionally) takes in a trailing number that specifies the number of parents you would like to ascend. Let's see it in action.

**Branch forcing**

You're an expert on relative refs now, so let's actually *use* them for something.

One of the most common ways I use relative refs is to move branches around. You can directly reassign a branch to a commit with the -f option. So something like:

git branch -f main HEAD~3

moves (by force) the main branch to three parents git HEAD.

**Reversing Changes in Git**

There are many ways to reverse changes in Git. And just like committing, reversing changes in Git has both a low-level component (staging individual files or chunks) and a high-level component (how the changes are actually reversed). Our application will focus on the latter.

There are two primary ways to undo changes in Git -- one is using git reset and the other is using git revert. We will look at each of these in the next dialog

**Git Revert**

While resetting works great for local branches on your own machine, its method of "rewriting history" doesn't work for remote branches that others are using.

In order to reverse changes and *share* those reversed changes with others, we need to use git revert. Let's see it in action.

**Moving Work Around**

So far we've covered the basics of git -- committing, branching, and moving around in the source tree. Just these concepts are enough to leverage 90% of the power of git repositories and cover the main needs of developers.

That remaining 10%, however, can be quite useful during complex workflows (or when you've gotten yourself into a bind). The next concept we're going to cover is "moving work around" -- in other words, it's a way for developers to say "I want this work here and that work there" in precise, eloquent, flexible ways.

This may seem like a lot, but it's a simple concept.

**Git Cherry-pick**

The first command in this series is called git cherry-pick. It takes on the following form:

* git cherry-pick <Commit1> <Commit2> <...>

It's a very straightforward way of saying that you would like to copy a series of commits below your current location (HEAD). I personally love cherry-pick because there is very little magic involved and it's easy to understand.

Let's see a demo!

***Git chery-pick c1 c3 c4*** (head should be there on which you want to paste the data )

**Git Revert (imp)**

If we have commit a certain change and it caused error so that we can remove it reverse it with the help of git revert and hence a new commit will be created and the changes made previously will be reversed.

With ***git revert***, a new commit is created with inverse changes. This cancels previous changes instead of making it as though the original commit never happened

Git revert c9 , git revert HEAD

**Git Interactive Rebase(imp)**

Git cherry-pick is great when you know which commits you want (*and* you know their corresponding hashes) -- it's hard to beat the simplicity it provides.

But what about the situation where you don't know what commits you want? Thankfully git has you covered there as well! We can use interactive rebasing for this -- it's the best way to review a series of commits you're about to rebase.

Let's dive into the details..

All interactive rebase means Git is using the rebase command with the -i option.

If you include this option, git will open up a UI to show you which commits are about to be copied below the target of the rebase. It also shows their commit hashes and messages, which is great for getting a bearing on what's what.

For "real" git, the UI window means opening up a file in a text editor like vim. For our purposes, I've built a small dialog window that behaves the same way.

When the interactive rebase dialog opens, you have the ability to do two things in our educational application:

* You can reorder commits simply by changing their order in the UI (via dragging and dropping with the mouse).
* You can choose to keep all commits or drop specific ones. When the dialog opens, each commit is set to be included by the pick button next to it being active. To drop a commit, toggle off its pick button.

*It is worth mentioning that in the real git interactive rebase you can do many more things like squashing (combining) commits, amending commit messages, and even editing the commits themselves. For our purposes though we will focus on these two operations above.*

Great! Let's see an example.

***git commit --amend*** allows us to modify and add changes to the most recent commit.

Git-amend rewrites the git history removing the previous commit and replacing it with the amended one.

**Locally stacked commits**

Here's a development situation that often happens: I'm trying to track down a bug but it is quite elusive. In order to aid in my detective work, I put in a few debug commands and a few print statements.

All of these debugging / print statements are in their own commits. Finally I track down the bug, fix it, and rejoice!

Only problem is that I now need to get my bugFix back into the main branch. If I simply fast-forwarded main, then main would get all my debug statements which is undesirable. There has to be another way...

Git rebase –i HEAD~3 --- Git checkout bugFix – git rebase bugFix

**Juggling Commits**

Here's another situation that happens quite commonly. You have some changes (newImage) and another set of changes (caption) that are related, so they are stacked on top of each other in your repository (aka one after another).

The tricky thing is that sometimes you need to make a small modification to an earlier commit. In this case, design wants us to change the dimensions of newImage slightly, even though that commit is way back in our history!!

We will overcome this difficulty by doing the following:

* We will re-order the commits so the one we want to change is on top with git rebase -i
* We will git commit --amend to make the slight modification
* Then we will re-order the commits back to how they were previously with git rebase -i
* Finally, we will move main to this updated part of the tree to finish the level (via the method of your choosing)

There are many ways to accomplish this overall goal (I see you eye-ing cherry-pick), and we will see more of them later, but for now let's focus on this technique. Lastly, pay attention to the goal state here -- since we move the commits twice, they both get an apostrophe appended. One more apostrophe is added for the commit we amend, which gives us the final form of the tree

That being said, I can compare levels now based on structure and relative apostrophe differences. As long as your tree's main branch has the same structure and relative apostrophe differences, I'll give full credit.

**Juggling Commits #2**

*If you haven't completed Juggling Commits #1 (the previous level), please do so before continuing*

**As you saw in the last level, we used rebase -i to reorder the commits.** Once the commit we wanted to change was on top, we could easily --amend it and re-order back to our preferred order.

The only issue here is that there is a lot of reordering going on, which can introduce rebase conflicts. Let's look at another method with git cherry-pick.

***The git commit --amend command is a convenient way to modify the most recent commit. It lets you combine staged changes with the previous commit instead of creating an entirely new commit. It can also be used to simply edit the previous commit message without changing its snapshot.***

*Git commit –amend*

*Git cherry-pick c2 c3 – now this will make a copy of these commits and put under the branch on which we are working currently*

**GIT tags**

As you have learned from previous lessons, branches are easy to move around and often refer to different commits as work is completed on them. Branches are easily mutated, often temporary, and always changing.

If that's the case, you may be wondering if there's a way to *permanently* mark historical points in your project's history. For things like major releases and big merges, is there any way to mark these commits with something more permanent than a branch?

You bet there is! Git tags support this exact use case -- they (somewhat) permanently mark certain commits as "milestones" that you can then reference like a branch.

More importantly though, they never move as more commits are created. You can't "check out" a tag and then complete work on that tag -- tags exist as anchors in the commit tree that designate certain spots.

Git tag v1 c2

Git tag v2 c4 – It applies a permanent tag on the given commit and can be considered as a milestone or a branch.

**Git Describe**

Because tags serve as such great "anchors" in the codebase, git has a command to *describe* where you are relative to the closest "anchor" (aka tag). And that command is called git describe!

Git describe can help you get your bearings after you've moved many commits backwards or forwards in history; this can happen after you've completed a git bisect (a debugging search) or when sitting down at the computer of a coworker who just got back from vacation.

Git describe takes the form of:

git describe <ref>

Where <ref> is anything git can resolve into a commit. If you don't specify a ref, git just uses where you're checked out right now (HEAD).

The output of the command looks like:

<tag>\_<numCommits>\_g<hash>

Where tag is the closest ancestor tag in history, numCommits is how many commits away that tag is, and <hash> is the hash of the commit being described.

Git describe <any branch or commit> - this command is used to find out the tags, if it is on the same branch used in the query then it will only display the name of that tag but if it is tag of the parent branch or commit then it will show the **(name of the tag – no of commits between it – commit name)**

Git describe main – **v0 \_2\_gc2 (tag-no of commits – name of commit)**

**Rebasing Multiple Branches**

Man, we have a lot of branches going on here! Let's rebase all the work from these branches onto main.

Upper management is making this a bit trickier though -- they want the commits to all be in sequential order. So this means that our final tree should have C7' at the bottom, C6' above that, and so on, all in order.

If you mess up along the way, feel free to use reset to start over again. Be sure to check out our solution and see if you can do it in fewer commands!

**Specifying Parents**

Like the ~ modifier, the ^ modifier also accepts an optional number after it.

Rather than specifying the number of generations to go back (what ~ takes), the modifier on ^ specifies which parent reference to follow from a merge commit. Remember that merge commits have multiple parents, so the path to choose is ambiguous.

Git will normally follow the "first" parent upwards from a merge commit, but specifying a number with ^ changes this default behavior.

Enough talking, let's see it in action

Git checkout HEAD~1^2~1 – git branch bugWork – with this we can move our head directly wherever we want to move onto. Like with this command first we have checkedout the HEAD on main then with ~ we moved it upward straightly then there are two different commit and we have to decide with which commit we have to proceed. If we want to carry out with the first one of it simple we can go with that modifier command.

**Branch Spaghetti**

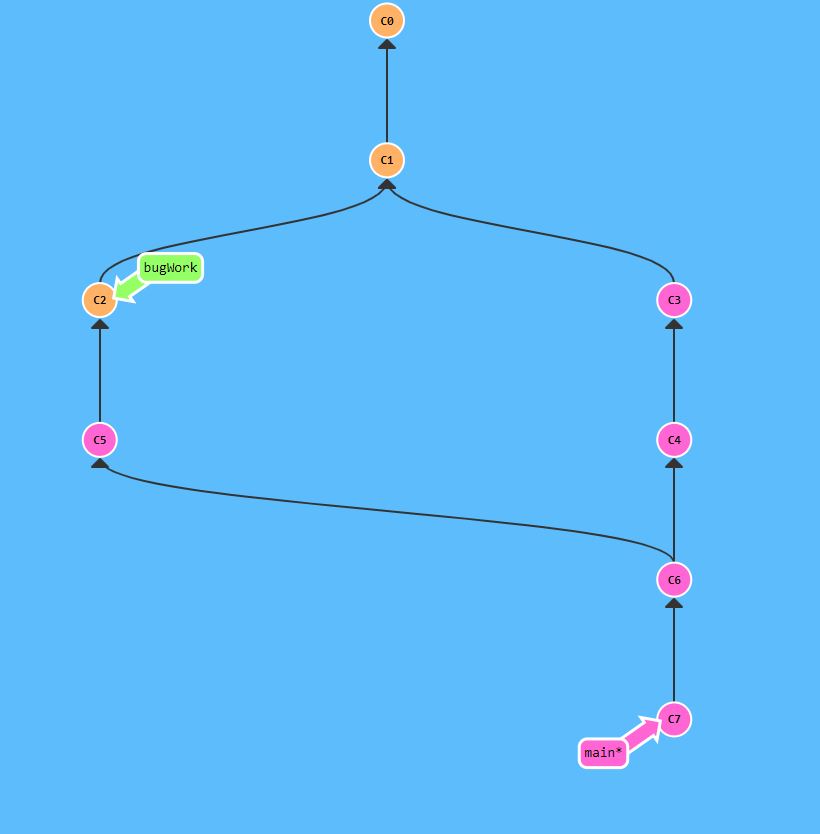
WOAHHHhhh Nelly! We have quite the goal to reach in this level.

Here we have main that is a few commits ahead of branches one two and three. For whatever reason, we need to update these three other branches with modified versions of the last few commits on main.

Branch one needs a re-ordering of those commits and an exclusion/drop of C5. Branch two just needs a pure reordering of the commits, and three only needs one commit transferred!

We will let you figure out how to solve this one -- make sure to check out our solution afterwards with show solution.

But if we want to proceed with the second commit or second parent than this command can be used ^, it will select the other parent or commit



**Git Remotes**

Remote repositories aren't actually that complicated. In today's world of cloud computing it's easy to think that there's a lot of magic behind git remotes, but they are actually just copies of your repository on another computer. You can typically talk to this other computer through the Internet, which allows you to transfer commits back and forth.

That being said, remote repositories have a bunch of great properties:

First and foremost, remotes serve as a great backup! Local git repositories have the ability to restore files to a previous state (as you know), but all that information is stored locally. By having copies of your git repository on other computers, you can lose all your local data and still pick up where you left off.

More importantly, remotes make coding social! Now that a copy of your project is hosted elsewhere, your friends can contribute to your project (or pull in your latest changes) very easily.

It's become very popular to use websites that visualize activity around remote repos (like [**GitHub**](https://github.com/)), but remote repositories *always* serve as the underlying backbone for these tools. So it's important to understand them!

**Our Command to create remotes**

Up until this point, Learn Git Branching has focused on teaching the basics of *local* repository work (branching, merging, rebasing, etc). However now that we want to learn about remote repository work, we need a command to set up the environment for those lessons. git clone will be that command.

Technically, git clone in the real world is the command you'll use to create *local* copies of remote repositories (from github for example). We use this command a bit differently in Learn Git Branching though -- git clone actually makes a remote repository out of your local one. Sure it's technically the opposite meaning of the real command, but it helps build the connection between cloning and remote repository work, so let's just run with it for now

**Git clone** -- it makes a replication of the repository or the area where you are working or tha tree

**Git Remote Branches**

Now that you've seen git clone in action, let's dive into what actually changed.

The first thing you may have noticed is that a new branch appeared in our local repository called o/main. This type of branch is called a *remote* branch; remote branches have special properties because they serve a unique purpose.

Remote branches reflect the *state* of remote repositories (since you last talked to those remote repositories). They help you understand the difference between your local work and what work is public -- a critical step to take before sharing your work with others.

Remote branches have the special property that when you check them out, you are put into detached HEAD mode. Git does this on purpose because you can't work on these branches directly; you have to work elsewhere and then share your work with the remote (after which your remote branches will be updated).

To be clear: Remote branches are on your *local* repository, not on the remote repository.

### What is o/?

You may be wondering what the leading o/ is for on these remote branches. Well, remote branches also have a (required) naming convention -- they are displayed in the format of:

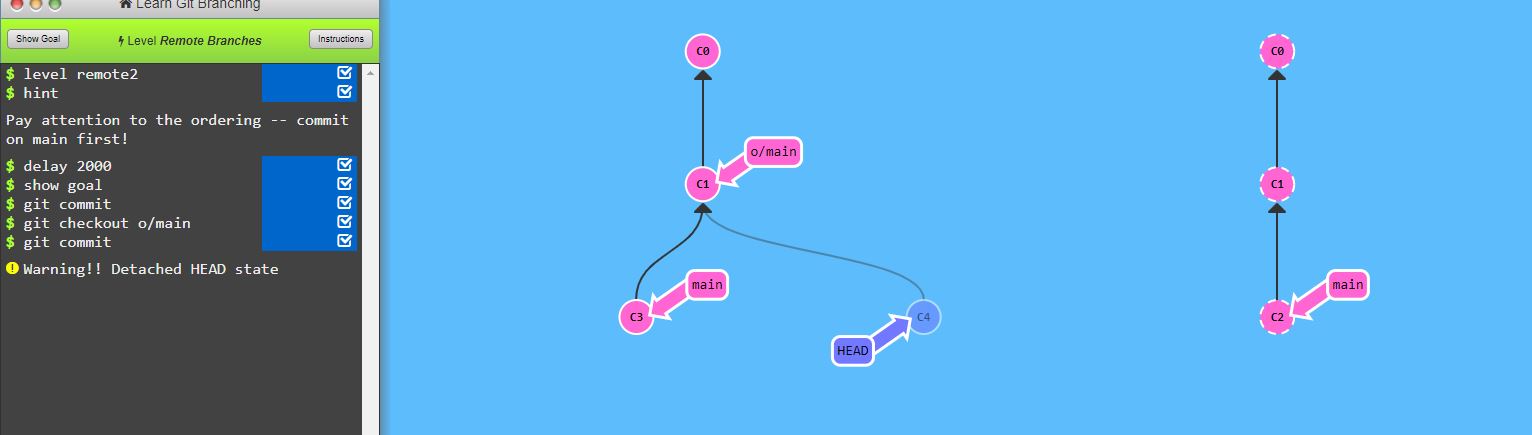
* <remote name>/<branch name>

Hence, if you look at a branch named o/main, the branch name is main and the name of the remote is o.

Most developers actually name their main remote origin, not o. This is so common that git actually sets up your remote to be named origin when you git clone a repository.

Unfortunately the full name of origin does not fit in our UI, so we use o as shorthand :( Just remember when you're using real git, your remote is probably going to be named origin!

That's a lot to take in, so let's see all this in action.



**Git Fetch**

Working with git remotes really just boils down to transferring data *to* and *from* other repositories. As long as we can send commits back and forth, we can share any type of update that is tracked by git (and thus share work, new files, new ideas, love letters, etc.).

In this lesson we will learn how to fetch data *from* a remote repository -- the command for this is conveniently named git fetch.

You'll notice that as we update our representation of the remote repository, our *remote* branches will update to reflect that new representation. This ties into the previous lesson on remote branches.

**What fetch does**

git fetch performs two main steps, and two main steps only. It:

* *Downlo ads the commits that the remote has but are missing from our local repository, and...*
* *updates where our remote branches point (for instance, o/main)*

git fetch essentially brings our *local* representation of the remote repository into synchronization with what the *actual* remote repository looks like (right now).

If you remember from the previous lesson, we said that remote branches reflect the state of the remote repositories *since* you last talked to those remotes. git fetch is the way you talk to these remotes! Hopefully the connection between remote branches and git fetch is apparent now.

git fetch usually talks to the remote repository through the Internet (via a protocol like http:// or git://).

Fit fetch – after using this command the main will fetch all the records from the remote one and adjust it where it is required. But keep in mind that it is just used to replicate the remote commits or branches

**Git Pull**

Now that we've seen how to fetch data from a remote repository with git fetch, let's update our work to reflect those changes!

There are actually many ways to do this -- once you have new commits available locally, you can incorporate them as if they were just normal commits on other branches. This means you could execute commands like:

* git cherry-pick o/main
* git rebase o/main
* git merge o/main
* etc., etc.

*In fact, the workflow of fetching remote changes and then merging them is so common that git actually provides a command that does both at once! That command is git pull.*

***Git pull*** *– this command can be used instead of merge and and fetch command – basically used for cloning the remote job/commits/project into the main as it is present in it already.*

*If there is any project and we want to replicate it to the main then we can use git pull*

*Git pull can be used to fetch the latest changes or updates made in the repo*

***git fetch*** *downloads the latest changes from a remote repository and updates the remote-tracking branches in your local repository to reflect those changes. However, it does not automatically merge those changes into your current branch. This allows you to review the changes and decide how to integrate them into your local branch.*

*On the other hand,* ***git pull*** *combines the* ***git fetch*** *command with an automatic merge. It fetches the changes from the remote repository and immediately merges them into your current branch. This simplifies the process by automatically incorporating the changes, but it may require manual conflict resolution if there are conflicting changes between your local branch and the remote branch.*

**Simulating collaboration**

So here is the tricky thing -- for some of these upcoming lessons, we need to teach you how to pull down changes that were introduced in the remote.

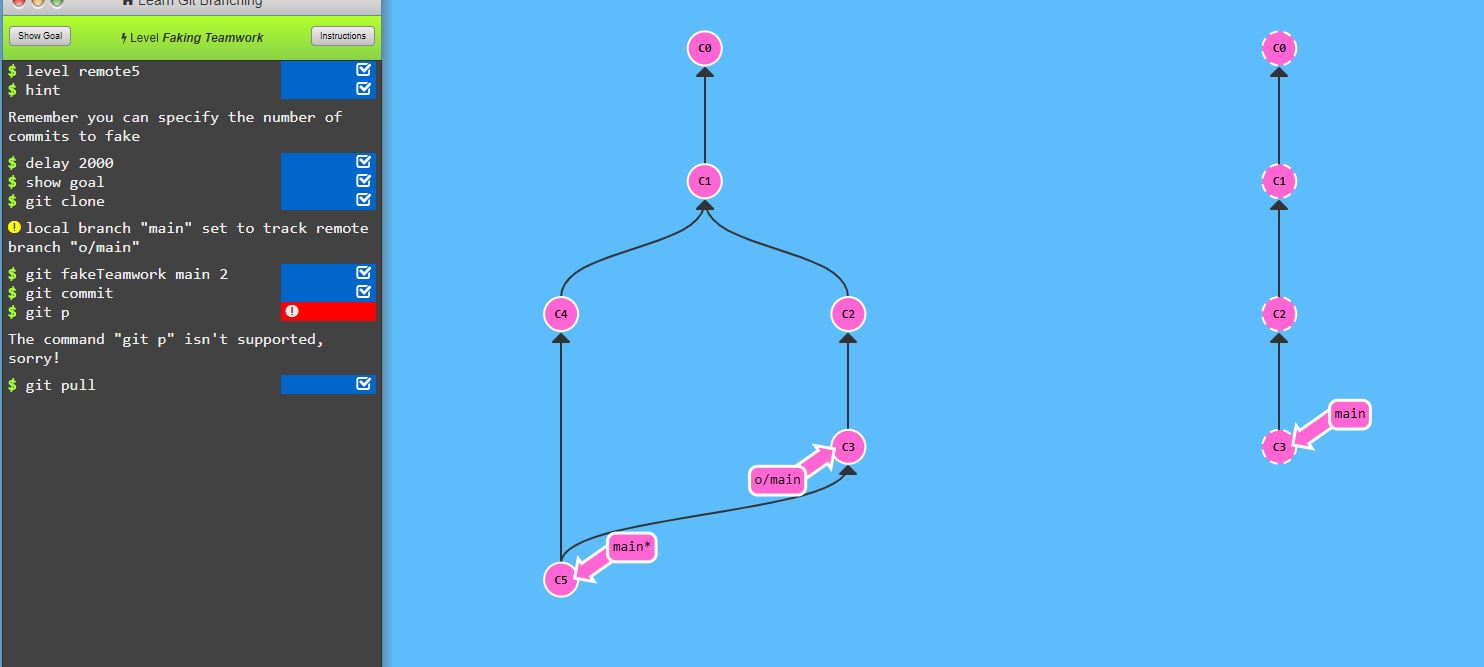
That means we need to essentially "pretend" that the remote was updated by one of your coworkers / friends / collaborators, sometimes on a specific branch or a certain number of commits.

In order to do this, we introduced the aptly-named command git fakeTeamwork! It's pretty self explanatory, let's see a demo...

Git faketeamwork is used to pretend/show some changes in the remote repos.

Git faketeamwork main 3 – this command will initiate 3 commits in the remote repo inorder for the real main to clone it or do someother things

Git faketeamwork abc 5 – this will initiate 5 commits below the given branch.



**Git Push**

Ok, so I've fetched changes from remote and incorporated them into my work locally. That's great and all... but how do I share *my* awesome work with everyone else?

Well, the way to upload shared work is the opposite of downloading shared work. And what's the opposite of git pull? git push!

git push is responsible for uploading *your* changes to a specified remote and updating that remote to incorporate your new commits. Once git push completes, all your friends can then download your work from the remote.

You can think of git push as a command to "publish" your work. It has a bunch of subtleties that we will get into shortly, but let's start with baby steps...

*note -- the behavior of git push with no arguments varies depending on one of git's settings called push.default. The default value for this setting depends on the version of git you're using, but we are going to use the upstream value in our lessons. This isn't a huge deal, but it's worth checking your settings before pushing in your own projects.*

Git push – it is used to push our main repo and where there are some changes made to the remote repo.

[git checkout](https://git-scm.com/docs/git-checkout) is effectively used to switch branches.

[git reset](https://git-scm.com/docs/git-reset#_examples) basically resets the repo, throwing away some changes. It’s somewhat difficult to understand, so reading the examples in the documentation may be a bit more useful.

There are some other useful articles online, which discuss more aggressive approaches to [resetting the repo](https://jwiegley.github.io/git-from-the-bottom-up/3-Reset/4-doing-a-hard-reset.html).

[git commit --amend](https://git-scm.com/docs/git-commit#Documentation/git-commit.txt---amend) is used to make changes to commits after-the-fact, which can be useful for making notes about a given commit.

[git revert](https://git-scm.com/docs/git-revert) makes a new commit which effectively rolls back a previous commit. It’s a bit like an undo command.

| **Command** | **Explanation & Link** |
| --- | --- |
| git branch | [Used to manage branches](https://git-scm.com/docs/git-branch) |
| git branch <name> | [Creates the branch](https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging) |
| git branch -d <name> | [Deletes the branch](https://git-scm.com/docs/git-branch#Documentation/git-branch.txt--D) |
| git branch -D <name> | [Forcibly deletes the branch](https://git-scm.com/docs/git-branch#Documentation/git-branch.txt--D) |
| git checkout <branch> | [Switches to a branch.](https://git-scm.com/docs/git-checkout) |
| git checkout -b <branch> | Creates a new branch and [switches to it](https://git-scm.com/docs/git-checkout#Documentation/git-checkout.txt--bltnewbranchgt). |
| git merge <branch> | [Merge joins branches together](https://git-scm.com/docs/git-merge). |
| git merge --abort | If there are merge conflicts (meaning files are incompatible), --abort can be used to abort the merge action. |
| git log --graph --oneline | [This shows a summarized view of the commit history for a repo](https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History). |

Git is a distributed version control system. Distributed means that each developer has a copy of the whole repository on their local machine.

GitHub is a web-based Git repository hosting service. On top of the version control functionality of Git, GitHub includes extra features like bug tracking, wikis, and task management. GitHub lets us share and access repositories on the web and copy or clone them to our local computer, so we can work on them. GitHub is a popular choice with a robust feature set, but it's not the only one. Other services that provide similar functionality are BitBucket, and GitLab.

There are various remote repository hosting sites:

* [GitHub](http://github.com/)
* [BitBucket](https://bitbucket.org/product)
* [Gitlab](https://gitlab.com/).

Follow the workflow at <https://github.com/join> to set up a free account, username, and password. After that, [these steps](https://help.github.com/articles/create-a-repo/) will help you create a brand new repository on GitHub.

Some useful commands for getting started:

| **Command** | **Explanation & Link** |
| --- | --- |
| git clone URL | [Git clone is used to clone a remote repository into a local workspace](https://git-scm.com/docs/git-clone) |
| git push | [Git push is used to push commits from your local repo to a remote repo](https://git-scm.com/docs/git-push) |
| git pull | [Git pull is used to fetch the newest updates from a remote repository](https://git-scm.com/docs/git-pull) |

| **Command** | **Explanation & Links** |
| --- | --- |
| git remote | [Lists remote repos](https://git-scm.com/docs/git-remote) |
| git remote -v | [List remote repos verbosely](https://git-scm.com/docs/git-remote#Documentation/git-remote.txt--v) |
| git remote show <name> | [Describes a single remote repo](https://git-scm.com/docs/git-remote#Documentation/git-remote.txt-emshowem) |
| git remote update | [Fetches the most up-to-date objects](https://git-scm.com/docs/git-remote#Documentation/git-remote.txt-emupdateem) |
| git fetch | [Downloads specific objects](https://git-scm.com/docs/git-fetch) |
| git branch -r | [Lists remote branches](https://git-scm.com/docs/git-branch#Documentation/git-branch.txt--r); can be combined with other branch arguments to manage remote branches |