LEARN GIT BRANCHING

Introduction Sequence

A nicely paced introduction to the majority of git commands

1. Introduction to Git Commits

Make two commits to complete the level

```
$ level intro1
$ hint
Just type in 'git commit' twice to finish!

$ git commit
$ git commit
```

2. Branching in Git

Make a new branch named bugFix and switch to that branch. By the way, here's a shortcut: if you want to create a new branch AND check it out at the same time, you can simply type git checkout -b [yourbranchname].

```
$level intro2
$ hint
Make a new branch with "git branch " and check it out with "git checkout "
$ git branch bugFix
$ git checkout bugFix
```

3. Merging in Git

To complete this level, do the following steps:

- Make a new branch called bugFix
- Checkout the bugFix branch with git checkout bugFix
- Commit once
- Go back to master with git checkout
- Commit another time
- Merge the branch bugFix into master with git merge

```
$ level intro3
$ hint
// Remember to commit in the order specified (bugFix before master)

$ git branch bugFix
$ git checkout bugFix
$ git commit
$ git checkout master
$ git commit
$ git merge bugFix
```

4. Rebase Introduction

To complete this level, do the following:

- Checkout a new branch named bugFix
- Commit once
- Go back to master and commit again
- Check out bugFix again and rebase onto master

```
$ level intro4
$ hint
// Make sure you commit from bugFix first

$ git branch bugFix
$ git checkout bugFix
$ git commit
$ git checkout master
$ git commit
$ git checkout bugFix
$ git rebase master
```

Ramping Up

The next serving of 100% git awesomes-ness. Hope you're hungry

1. Detach yo' HEAD

To complete this level, let's detach HEAD from bugFix and attach it to the commit instead. Specify this commit by its hash. The hash for each commit is displayed on the circle that represents the commit.

```
$ level rampup1
$ hint
// use the labek (hash) on the commit for help!

$ git checkout c4
```

2. Relative Refs (^)

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 ~
To complete this level, check out the parent commit of bugFix. This will detach HEAD.
You can specify the hash if you want, but try using relative refs instead!

```
$ level rampup2
$ hint
// use the caret (^)

$ git checkout bugFix(^)
```

3. **Relative Refs #2 (~)**

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.
Branch forcing: git branch -f master HEAD~3 moves (by force) the master branch to three parents behind HEAD.
To complete this level, move HEAD, master, and bugFix to their goal destinations shown.

```
$ level rampup3
$ hint
// You'll need to use at least one direct reference (hash) to complete this level

$ git branch -f master c6
$ git branch -f bugFix c0
$ git checkout HEAD^
```

4. **Reversing Changes in Git**

There are two primary ways to undo changes in Git -- one is using git reset and the other is using git revert. git reset will move a branch backwards as if the commit had never been made in the first place.
To complete this level, reverse the most recent commit on both local and pushed. You will revert two commits total (one per branch). Keep in mind that pushed is a remote branch and local is a local branch -- that should help you choose your methods.

```
$ level rampup4
$ hint
// Notice that revert and reset take different arguments.

$ git reset HEAD~1
$ git checkout pushed
$ git revert pushed
```

Moving Work Around

Get comfortable with modifying the source tree

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.

1. **Git Cherry-pick**

```
$ level move1
$ hint
// git cherry-pick followed by commit names!

git cheery-pick c3 c4 c7
```

2. **Git Interactive Rebase**

All interactive rebase means is using the rebase command with the -i option.
When the interactive rebase dialog opens, you have the ability to do 3 things:

- You can reorder commits simply by changing their order in the UI (in our window this means dragging and dropping with the mouse).
- You can choose to completely omit some commits. This is designated by pick -- toggling pick off means you want to drop the commit.
- Lastly, you can squash commits. Unfortunately our levels don't support this for a few logistical reasons, so I'll skip over the details of this. Long story short, though -- it allows you to combine commits.

To finish this level, do an interactive rebase and achieve the order shown in the goal visualization. Remember you can always undo or reset to fix mistakes :D

```
$ level move2
$ hint
// you can use either branches or relative refs (HEAD~) to specify the rebase target

$ git rebase -i HEAD~4
// omit c2, place C5 over c4
```

A Mixed Bag

A mixed bag of Git techniques, tricks, and tips

1. **Grabbing Just 1 Commit**

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 master branch. If I simply fast-forwarded master, then master would get all my debug statements which is undesirable. There has to be another way...

We need to tell git to copy only one of the commits over. This is just like the levels earlier on moving work around -- we can use the same commands:
`git rebase -i git cherry-pick` To achieve this goal.

This is a later level so we will leave it up to you to decide which command you want to use, but in order to complete the level, make sure master receives the commit that bugFix references.

```
$ level mixed1
$ hint
$ git checkout master
// Remember, interactive rebase or cherry-pick is your friend here
$ git rebase -i master // ommit c2 c3
$ git checkout master
```

```
$ git merge bugFix
```

2. Juggling Commits #1

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 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 master 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 master branch has the same structure and relative apostrophe differences, I'll give full credit

```
$ level mixed2 $ hint // The first command is git rebase -i HEAD~2
$ git rebase -i HEAD~2 // Reorder C3,C2
$ git commit --amend
$ git rebase -i HEAD~2 // Reorder C2'',C3'
$ git rebase caption master
```

3. 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

Remember that git cherry-pick will plop down a commit from anywhere in the tree onto HEAD (as long as that commit isn't an ancestor of HEAD).

```
So in this level, let's accomplish the same objective of amending C2 once but avoid using rebase -i. I'll leave it up to you to figure it out! :D
```

Remember, the exact number of apostrophe's (') on the commit are not important, only the relative differences. For example, I will give credit to a tree that matches the goal tree but has one extra apostrophe everywhere

```
$ level mixed3
$ hint
// Don't forget to forward master to the updated changes!
$ git checkout master
$ git cherry-pick C2
$ git commit --amend
$ git cherry-pick C3
```

4. 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.

Let's see what tags look like in practice.

Let's try making a tag at C1 which is our version 1 prototype

There! Quite easy. We named the tag v1 and referenced the commit C1 explicitly. If you leave the commit off, git will just use whatever HEAD is at

For this level just create the tags in the goal visualization and then check v1 out. Notice how you go into detached HEAD state -- this is because you can't commit directly onto the v1 tag.

In the next level we'll examine a more interesting use case for tags.

```
$ git tag v0 C1
$ git tag v1 C2
$ git checkout C2
```

5. 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 a coworkers computer who just got back from vacation.

Git describe takes the form of:
git describe
`Where 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:
`__g`
`Where tag is the closest ancestor tag in history, numCommits is how many commits away that tag is, and is the hash of the commit being described.`

```
$ level mixed5
$ hint
// Just commit once on bugFix when you're ready to move on
$ git commit // That's all!
```