**Introduction to GIT commits**

* Repository records all the files in the directory.
* The commit restores the versions of these repositories.
* Git maintains history of these commits. So every changed commit has an ancestor commit.

**GIT branches**

* Branches are lightweight
* They act as pointers to different commits
* It locally divides your work

**Merging in GIT**

* Combining work from two different branches is called merging.
* Merging creates a special commit

**GIT Rebase**

* Other way of combining work between branches is rebasing.
* 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.

**Ramping up**

Head

* HEAD is the symbolic name for the currently checked out commit
* HEAD always points to the most recent commit which is reflected in the working tree
* Detaching HEAD just means attaching it to a commit instead of a branch

Relative Refs

* Git is smart about hashes
* It only requires you to specify enough characters of the hash until it uniquely identifies the commit.
* Relative commits are powerful, but we will introduce two simple ones here: 1. Moving upwards one commit at a time with ^ 2. Moving upwards a number of times with ~<num>

Relative refs (~) -2

* The tilde operator (optionally) takes in a trailing number that specifies the number of parents you would like to ascend.
* 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.

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.
* 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.

**MOVING WORK AROUND**

* The first command in this series is called git cherry-pick. It takes on the following form:  
    
  git cherry-pick <Commit1> <Commit2> <...>
* Interactive rebase means is using the rebase command with the -i option.
* 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.

**A MIXED BAG**

* 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
* 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.
* 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)
* 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
* 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. 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.
* 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).

**ADVANCED TOPICS:**

* Upper management is making 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!
* Like the ~ modifier, the ^ modifier also accepts an optional number after it.
* Git will normally follow the "first" parent upwards from a merge commit, but specifying a number with ^ changes this default behavior.
* Here we have master that is a few commits ahead of branches one two and three.
* Branch one needs a re-ordering and a deletion of C5. two needs pure reordering, and three only needs one commit!