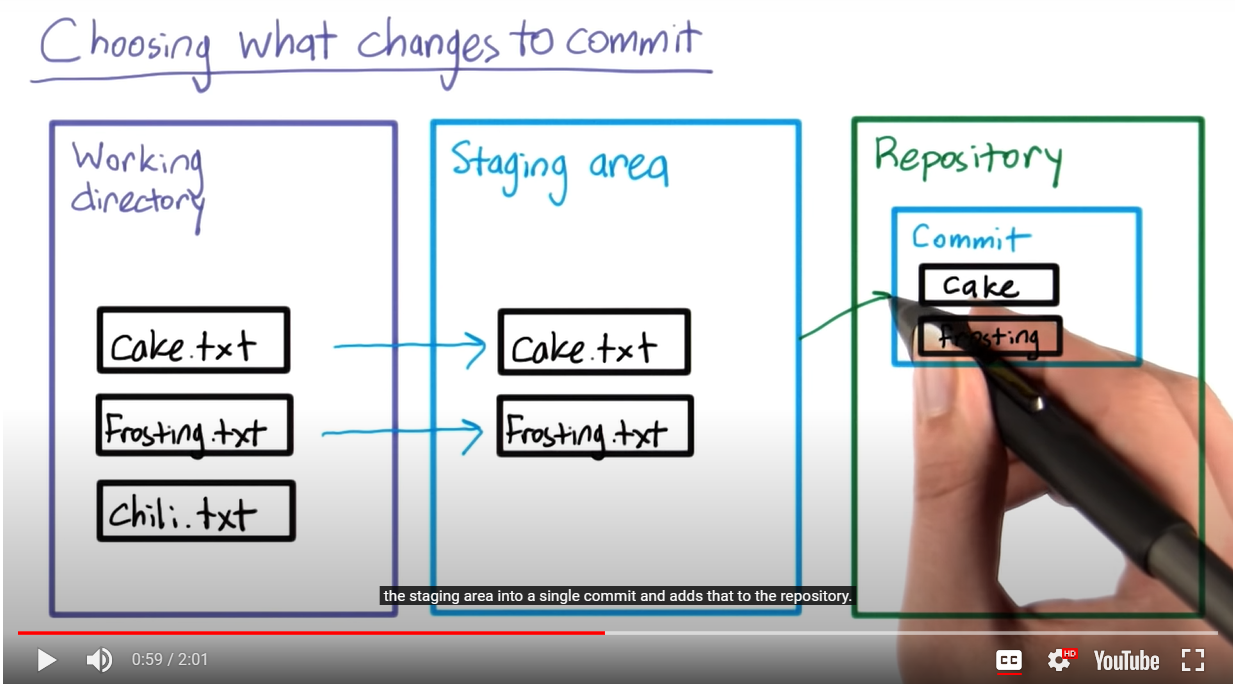
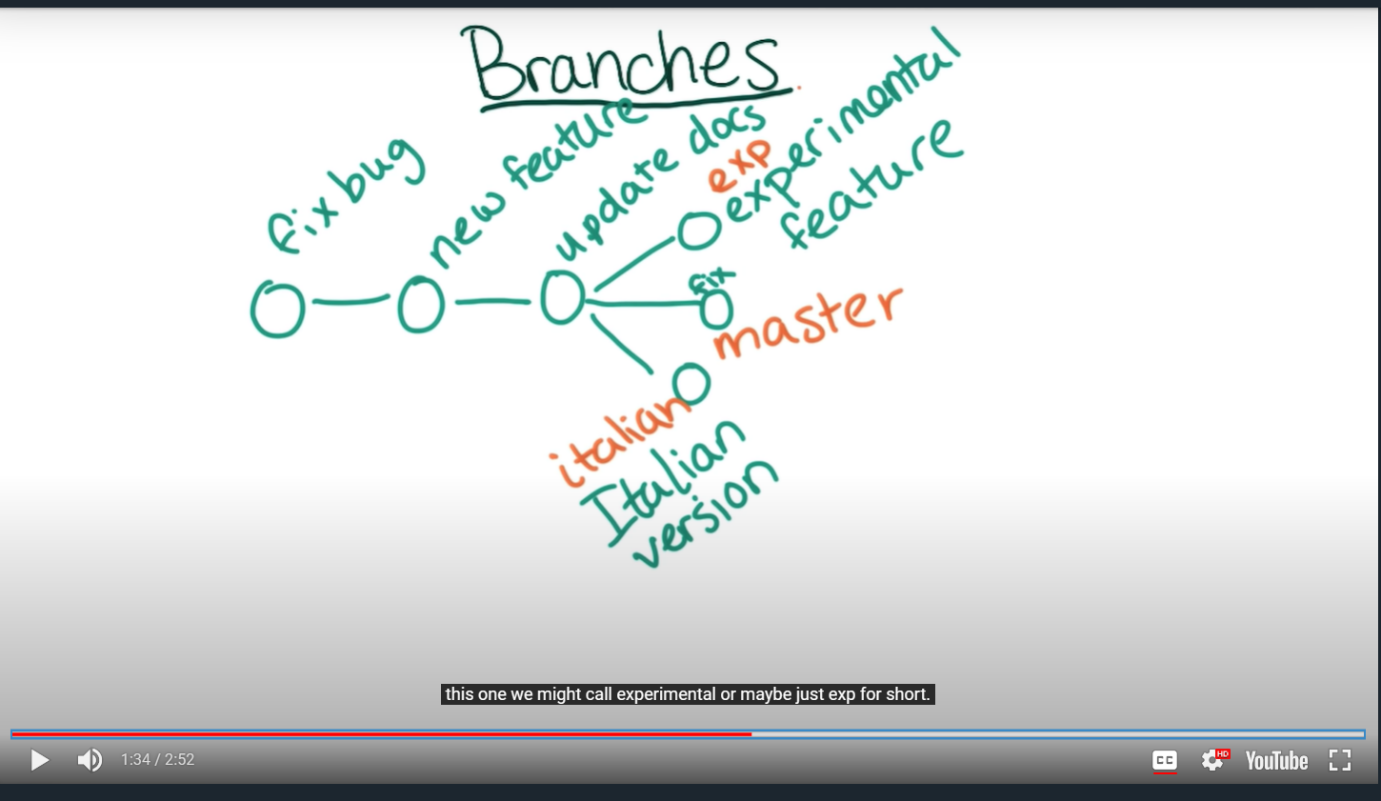
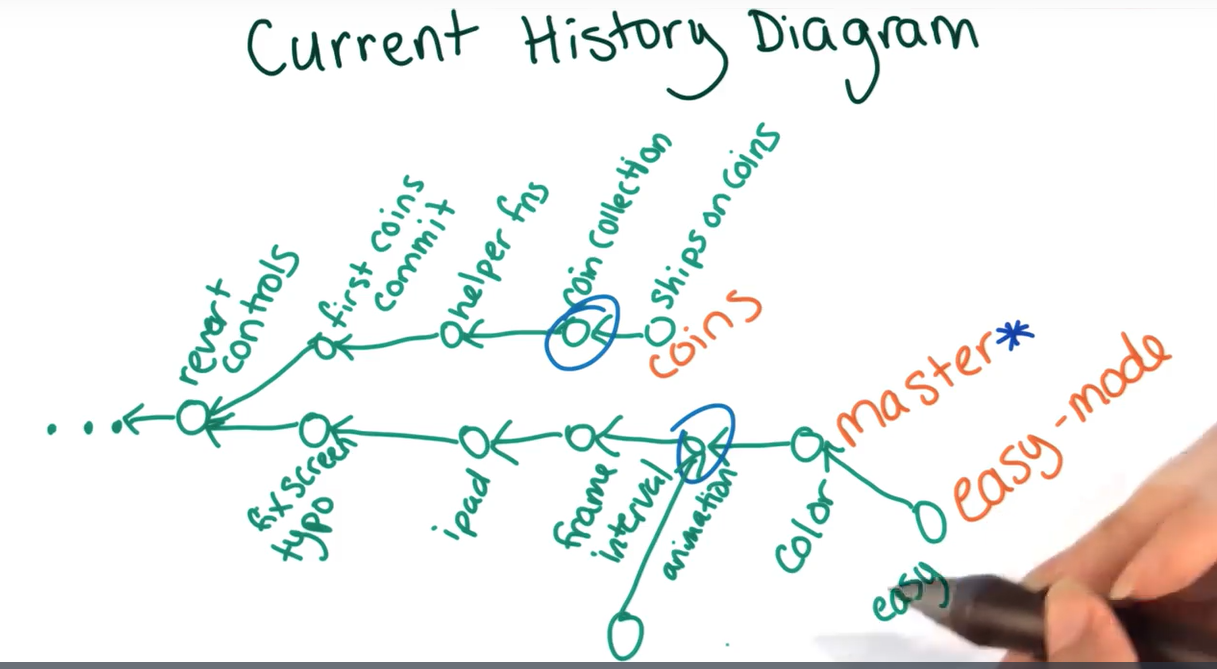
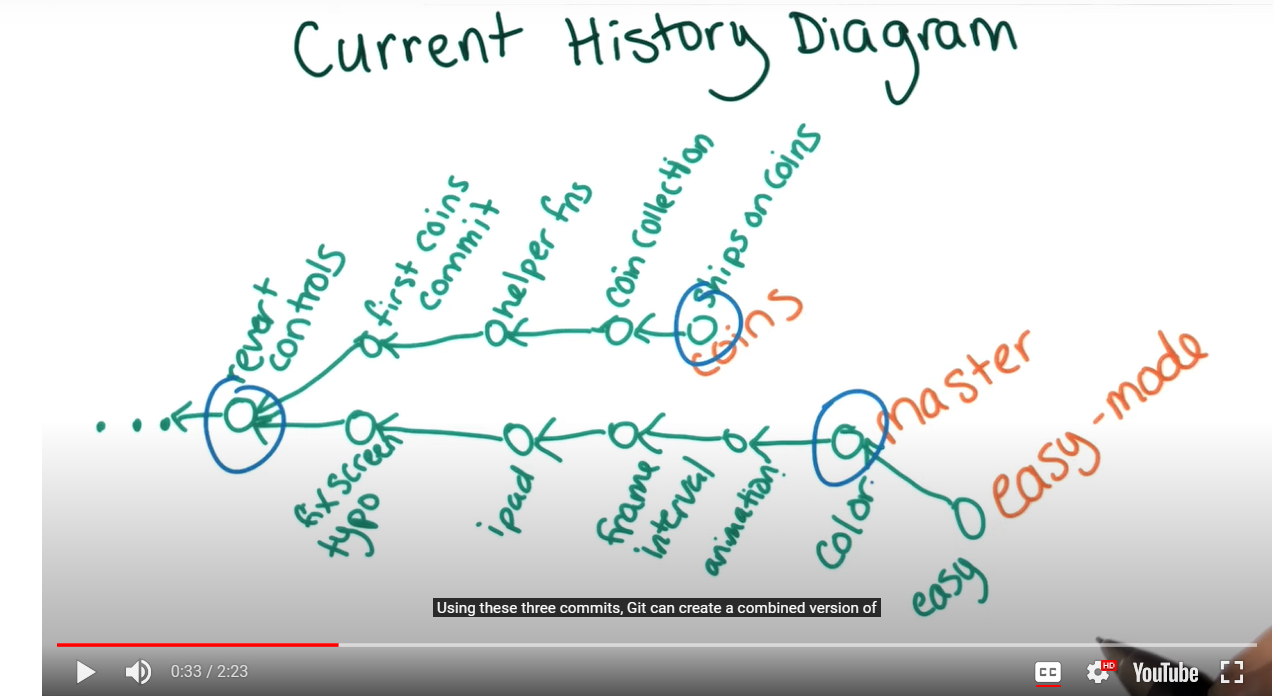
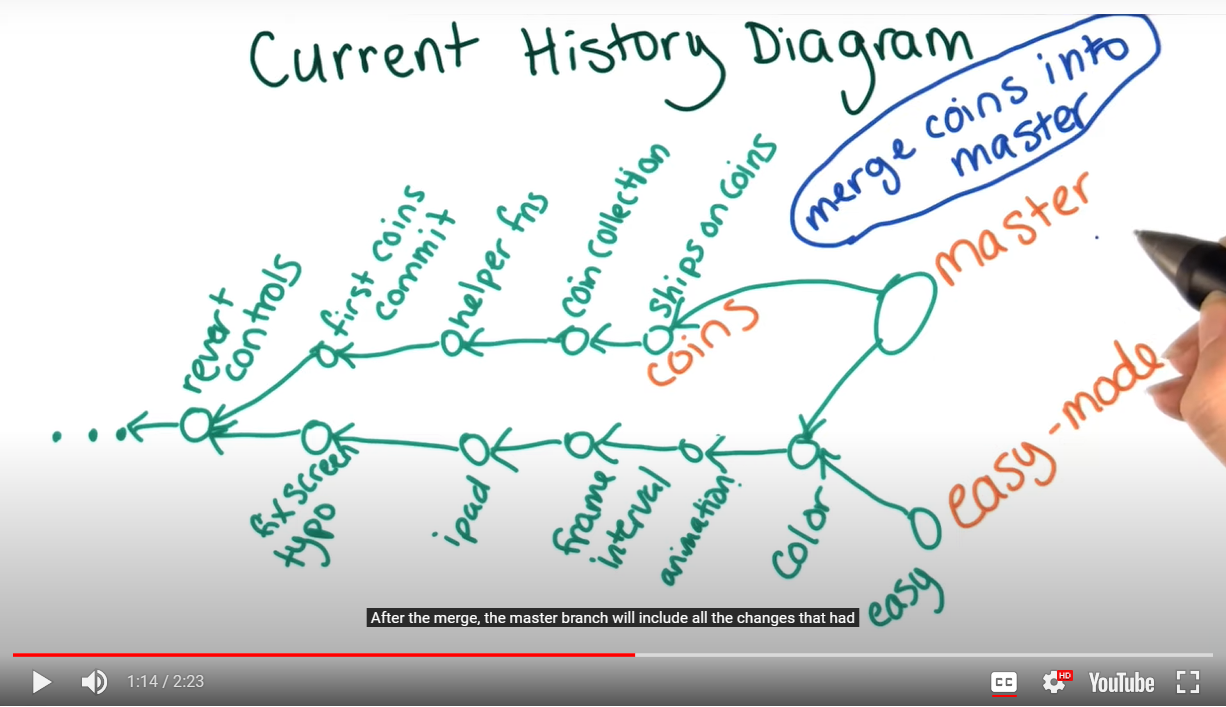
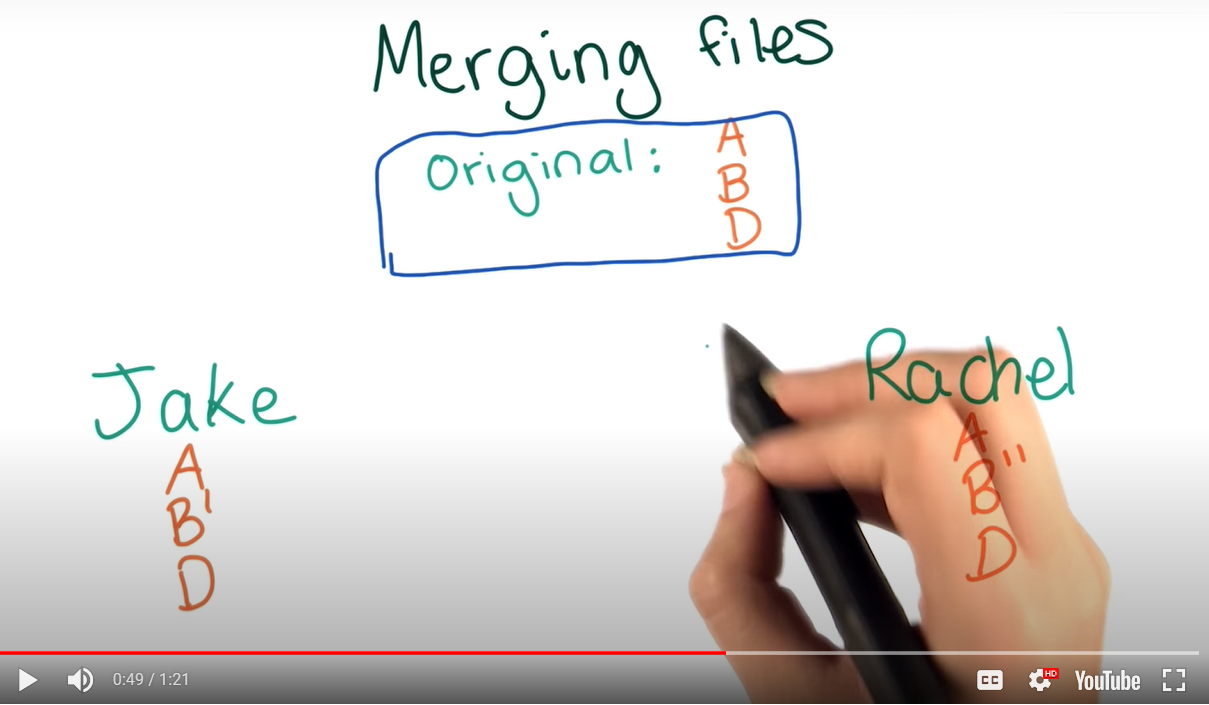
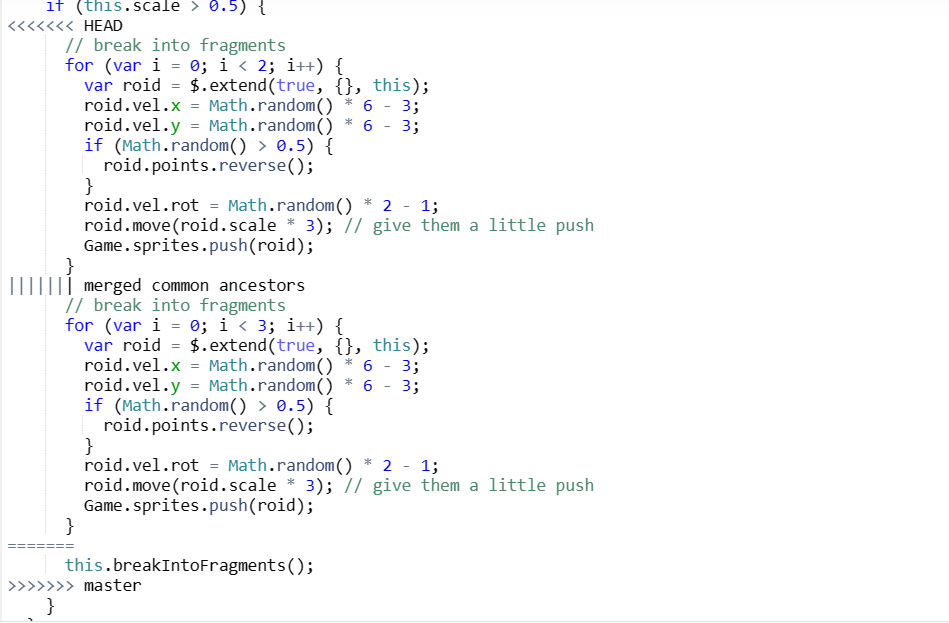
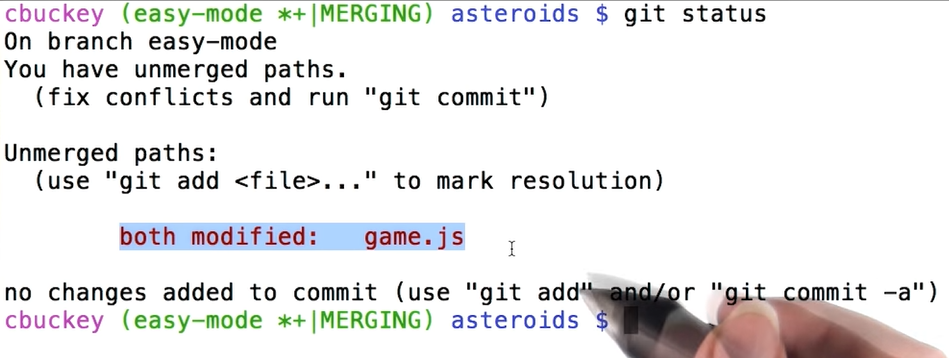
* Git stores metadata of a repository with the help of some hidden files. The filenames of these types usually start with a period, e.g. **.git**
* To create a git repository, go to the folder which you want to make a repository and type the command: **git init**
* You should be able to find a .git file in that folder after that.
* To list all the files (including hidden files) use the command: ls -a
* After creating a git repository to find the status of the repository use the command: git status
* Git allows the user to choose which files he wants to commit. After creating the repository, the user may move the two files into the staging area using the command: **git add <filename>**
* Now the two files which were added to the staging area will be tracked and will be added when a git commit is done. However, the third file will still remain untracked.
* 
* If you added some files accidentally to the staging area then you can revert it using the command **git reset**
* To commit the files in the staging area, use the command: **git commit**
* The default editor will appear after this command and you’ll have to type the commit message.
* In case you want to type the message in the command line, use the command: **git commit -m “commit message”**
* Staging area usually contains copy of the files from the most recent commit until and unless file changes have been added to it.
* To find the difference between the staging area and working directory use the command: **git diff**
* To find the difference between the staging area and the most recent commit use the command: **git diff –staged**
* To discard all the changes in the working directory and the staging area use the command: **git reset –hard**
* To go back to the latest commit, use the command: **git checkout master** (here there is no branching yet but the command may change when there are more than one branch).
* Branches are really useful while working on development.
* Git shows the “detached head” message when a commit is not labeled with a branch.
* When you checkout a branch and do a commit, the branch label automatically updates to the new commit.
* 
* The current last commit on a branch is called the “tip” of the branch.
* The process of combining two branches is called **merging**.
* To visualize the graph structure of the repository, use the command**: git log –graph –oneline master coins**
* The git log function shows the commits as a graph. The most recent commit traces its parent commit in the current branch and its parent traces its own parent and so on. SO, the concept of reachability comes into picture. To visualize this more clearly checkout the picture below -
* The detached head message is shown when a commit does not have a parent branch to follow. This way it won’t show up in the git log function call.
* To create a branch and checkout to it run the following commands: **git branch <new\_branch\_name>** and then **git checkout <new\_branch\_name>**
* To create a branch and checkout into it in a single command use the command: **git checkout -b new\_branch\_name**
* While merging two files it is important to know which line to add or to not add. For example, if a line appears in both codes, then it is obvious that the line should appear in the final code. However, if a line is present in only one code, then it is not confirmed whether it should be added in the final code or not. Maybe one of the developers have deleted that line from his code. In that case it would be frustrating to find that line in the final code. Having the initial code can help us whether to add the line of code or not as it would make it clear whether the line is previously present or not. If it was previously present and it is not in one of the developer’s codes it means that the line has been removed by that developer and it is unwanted.
* Git can merge two branches into one. However, to do this it is not enough to have tip of both the branches as it is evident from the previous point. To do this we must also have the point where both branches diverged so that we can tell what part of both branches we should keep and what no to. Look at the image below for explanation, the three required tips are circled down.
* Commits are the building blocks of git. So, the merge operation will also be a commit. But unlike other commits, merge commit will have both coins and master as its parent commits. After merging master branch will have all of its previous changes along with all the changes of coins branch. After the merge we can delete the coins branch. Note that while we talk about deleting a branch, we are talking about deleting the label. The commits will still be there in the history. However, if no branch can reach the commit, deleting a branch does effectively delete all its commits. After the merging the git log will show the commits according to the time when they were created. They could be in an interlinked fashion.
* If a branch is deleted and leaves some commits unreachable from existing branches, those commits will continue to be accessible by commit id, until Git’s garbage collection runs. This will happen automatically from time to time, unless you actively turn it off. You can also run this process manually with git gc.
* To merge coins and master branch first checkout to the master branch. Now run the command to merge coins and master: **git merge master coins**. Now if the git log command is run it will show commits from both the branches in an interlinked fashion. However now we don’t know which commit is parent of which. To find out how a commit differs from its parent commit run the command: **git show <commit id>.**
* Now if we want to delete the coins branch use the command: **git branch -d coins**. Note that it will only delete the label. The commits are still reachable through master branch. However, if the branch was deleted before merging, those commits would have been lost.
* git merge will also include the currently checked-out branch in the merged version. So, if you have branch1 checked out, and you run git merge branch2 branch3, the merged version will combine branch1 as well as branch2 and branch3. That’s because the branch1 label will update after you make the merge commit, so it’s unlikely that you didn’t want the changes from branch1 included in the merge. For this reason, you should always checkout one of the two branches you’re planning on merging before doing the merge. Which one you should check out depends on which branch label you want to point to the new commit.
* Since the checked-out branch is always included in the merge, you may have guessed that when you are merging two branches, you don't need to specify both of them as arguments to git merge on the command line. If you want to merge branch2 into branch1, you can simply git checkout branch1 and then type git merge branch2. The only reason to type git merge branch1 branch2 is if it helps you keep better mental track of which branches you are merging.
* Also, since the two branches are merged, the order in which they are typed into the command line does not matter. The key is to remember that git merge always merges all the specified branches into the currently checked out branch, creating a new commit for that branch.
* It is not true that git will always merge two branches automatically. For example, see the image below. Merge conflicts might occur when two developers have the same function in their respective branch but have different implementations. Git does not know which implementation to discard and which to keep. In this case it allows the developer to decide which changes to keep. This process is called resolving merge conflicts.
* Master has updated since the easy-mode branch was created. In this case, it has the addition of coins, which you might like to include in the easy-mode branch. In general, it’s very common that if you make a branch, either an experimental branch or to work on a new feature, you want to periodically merge master into that branch. This is because master usually contains the official version of the code, and it’s common to want experimental changes to include all of the changes to master.
* To incorporate the changes of master into the easy-mode branch first checkout to the easy mode branch and then run the command: **git merge master easy-mode**. This will show a message of merge conflicts. Now if we open the file with the merge conflicts it will look something like this. The first part under head is the changes made by the branch easy-mode. The part under merged common ancestor is the part the code had before the branches made the changes. The part above master is the code changes made in the master. There might be codes in other parts of the code which were changed by either master or easy-mode but not listed here as they produced no merge conflicts.
* Now after resolving the merge conflicts if we go and do git status, it will show something like this. It shows ‘both modified” instead of only “modified” as the changes were made by both master and easy-mode.
* Now add the changes in the staging area using **git add** and then commit the changes using **git commit**.