Git and Social Coding

## Git

Git is a free and open-source version control system, originally created by Linus Torvalds in 2005. Unlike older centralized version control systems such as SVN and CVS, Git is distributed: every developer has the full history of their code repository locally. This makes the initial clone of the repository slower, but subsequent operations such as commit, blame, diff, merge, and log dramatically faster.

Git also has excellent support for branching, merging, and rewriting repository history, which has led to many innovative and powerful workflows and tools. Pull requests are one such popular tool that allows teams to collaborate on Git branches and efficiently review each other's code. Git is the most widely used version control system in the world today and is considered the modern standard for software development.

## How Git works

Here is a basic overview of how Git works:

1. Create a "repository" (project) with a git hosting tool (like Bitbucket)
2. Copy (or clone) the repository to your local machine
3. Add a file to your local repo and "commit" (save) the changes
4. "Push" your changes to your main branch
5. Make a change to your file with a git hosting tool and commit
6. "Pull" the changes to your local machine
7. Create a "branch" (version), make a change, commit the change
8. Open a "pull request" (propose changes to the main branch)
9. "Merge" your branch to the main branch

**Merge Conflicts**

<https://stackoverflow.com/questions/11646107/you-have-not-concluded-your-merge-merge-head-exists>

I made a branch called 'f' and did a checkout to master. When I tried the git pull command I got this message:

You have not concluded your merge (MERGE\_HEAD exists).

Please, commit your changes before you can merge.

When I try the git status, it gave me the following:

On branch master

# Your branch and 'origin/master' have diverged,

# and have 1 and 13 different commit(s) each, respectively.

#

# Changes to be committed:

#

# modified: app/assets/images/backward.png

# modified: app/assets/images/forward.png

The problem is your previous pull failed to merge automatically and went to conflict state. And the conflict wasn't resolved properly before the next pull.

1. Undo the merge and pull again.

To undo a merge:

git merge --abort

git fetch --all

Then, you have two options:

git reset --hard origin/master

OR If you are on some other branch:

git reset --hard origin/<branch\_name>

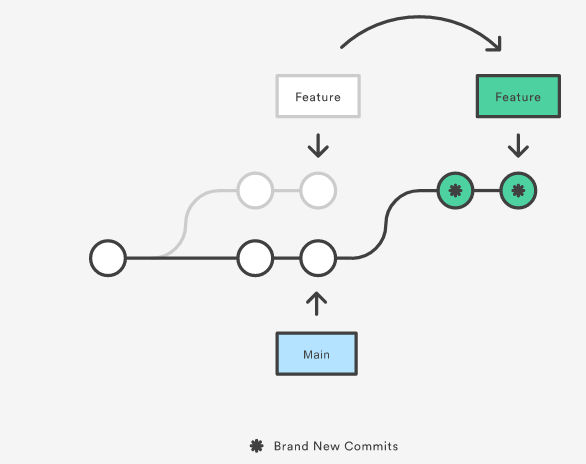
Abort did not work for me. So I had to continue and this is what worked for me:

git merge --continue

# git rebase

## What is git rebase?

Rebasing is the process of moving or combining a sequence of commits to a new base commit. Rebasing is most useful and easily visualized in the context of a feature branching workflow. The general process can be visualized as the following:



From a content perspective, rebasing is changing the base of your branch from one commit to another making it appear as if you'd created your branch from a different commit. Internally, Git accomplishes this by creating new commits and applying them to the specified base. It's very important to understand that even though the branch looks the same, it's composed of entirely new commits.

**Usage**

The primary reason for rebasing is to maintain a linear project history. For example, consider a situation where the main branch has progressed since you started working on a feature branch. You want to get the latest updates to the main branch in your feature branch, but you want to keep your branch's history clean so it appears as if you've been working off the latest main branch. This gives the later benefit of a clean merge of your feature branch back into the main branch. Why do we want to maintain a "clean history"? The benefits of having a clean history become tangible when performing Git operations to investigate the introduction of a regression. A more real-world scenario would be:

1. A bug is identified in the main branch. A feature that was working successfully is now broken.
2. A developer examines the history of the main branch using git log because of the "clean history" the developer is quickly able to reason about the history of the project.
3. The developer can not identify when the bug was introduced using git log so the developer executes a git bisect.
4. Because the git history is clean, git bisect has a refined set of commits to compare when looking for the regression. The developer quickly finds the commit that introduced the bug and is able to act accordingly.

# How to roll back Git code to a previous commit

With Git, IT teams can implement version control. Humans make mistakes, and consequently need to roll back to previous versions of content.

Let's look at how to roll back Git commits to a known-good version, as well as the implications and potential complications of doing so.

### What is git reset?

Every time an IT admin commits a Git deployment, that latest commit becomes the head, or tip, of the code tree -- in other words, the current version. The Git commit process provides a point-in-time snapshot ([PIT snapshot](https://www.techtarget.com/searchstorage/definition/point-in-time-snapshot-PIT-snapshot)) of the version-controlled files at the time of every change.

An administrator can roll back the code repository to a previous commit -- that point-in-time copy -- in several ways, depending on the end goal. One approach is the git reset command.

Before using this command, you must understand what git reset does. Outcomes can vary between command uses, and with which switches. Use the command with caution. There are two modes for git reset:

* **Soft**. This mode resets the code tree's head to the designated former commit instance. All the files between that PIT snapshot and now are set to staged -- ready to commit but not yet committed. This is the default mode.
* **Hard**. Use this mode with extreme caution. These changes can't be reverted. This command will [reset everything](https://www.theserverside.com/video/How-to-use-the-git-reset-hard-command-to-change-a-commit-history), move the head back to the indicated commit version and remove all changes added to the code tree after that specific version number. In effect, the git reset command instantiates a 'hard deletion' of all changes from now -- or point-in-time of code reversion -- to the designated former code commit. It resets the code tree to the version in question and deletes [unstaged files](https://www.theserverside.com/blog/Coffee-Talk-Java-News-Stories-and-Opinions/git-unstage-file-all-index-commit-folder-add-delete).

### How to undo a Git commit

First, decide how far back to go into the [version history](https://www.theserverside.com/video/Understand-the-Git-working-tree-status-command-for-easy-DVCS). To view the previous commits, use the git log –-oneline command. This provides the commit details.

Once the IT team chooses a code version to which their tree should revert, use the commit ID to execute the command. In the following example, **x12345** represents the commit ID, gained from the git log output:

git reset x12345