CAS CS 411
Lec 3

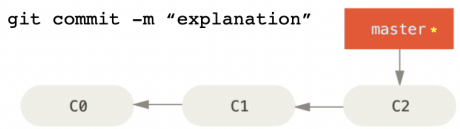
Version Control with Git

1. What we're trying to solve
    a. Track all changes to code base
    b. Rewindable history
    c. Collaboration
    d. Keep dev, test, prod code separate
    e. Work on features and fixes nondestructively
2. Two approaches
    a. Centralized repository (svn, cvs)
        i. One copy of code base on server
        ii. Devs check out a file to work on locally
        iii. No one else can work on that file
        iv. When done, dev checks file in to the server
    b. Advantages:
        i. Very clean separation of responsibility
        ii. Clean code history
    c. Disadvantages:
        i. Only one dev can work on a file at a time (and vacations!)
    d. Decentralized (git)
        i. Each dev has a copy of the code base
        ii. Concurrent work on a file is possible
        iii. Local versions (called branches) are embraced
    e. Advantages
        i. No locking of files
        ii. Concurrency
        iii. Simple branching
    f. Disadvantages
        i. Local branches must be merged ▪ History can become complex
3. Github
    a. Serves as a central repository (repo) for a project
    b. Since git is distributed, the copy on GitHub is canonical only by convention
    c. When GitHub is the canonical version, a distributed workflow can be built on it for concurrent development
    d. While there are no 'official' workflows, a few models have emerged that are commonly used
    e. We'll focus only on one that is appropriate for your team projects

4. Git concepts
    a. Git records local changes made to files in a directory (and its subdirectories)
    b. Those records are essentially snapshots of the state of all files at a given moment
    c. Multiple concurrent histories, called branches, are used to isolate specific work, for example a bug fix or new feature
    d. Two branches can be merged together, combining all of the changes made to both branches
    e. Local copies can be synchronized with other developers' local copies, or with branches stored on GitHub
5. Commits: Saving changes to a file
    a. Git only records changes when you tell it to, using the 'add' command
    b. 'add' is used to move the current state of a file into a staging area (it really should have been called 'stage' but wasn't)
    c. Changes that have been staged are recorded with the 'commit' command
    d. The workflow is
        i. edit -> stage (add) -> commit
6. Staging is a snapshot
    a. Staging happens when git add is executed and only then
    b. If you stage a file, then make more edits, they will not be included in the next commit unless you git add them again
7. Your best friend: git status
    a. The git status command provides details of your current state and advice on what to do next
    b.

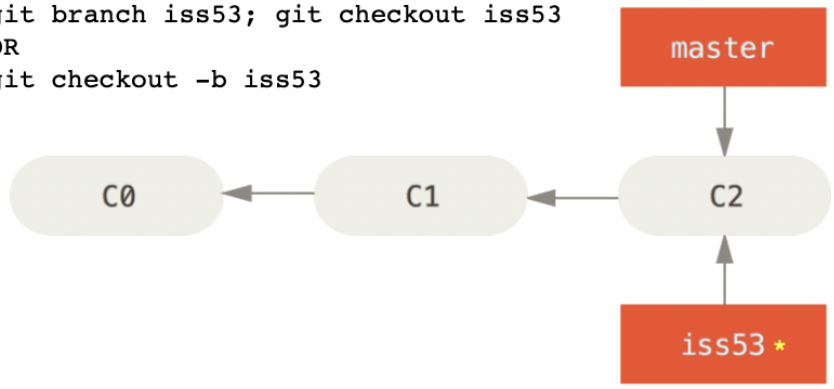
```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
$
```

8. Branching and Merging
    a. Branches are used to separate work from the main code base
    b. They let you work on new features, updates, bug fixes, and so on independent of the main code base
    c. In git it is common to create lots of new branches and delete them when you don't need them any more
    d. Work in a branch is rolled into the main line of code using the git merge command
    e. Work done on a single branch (master)

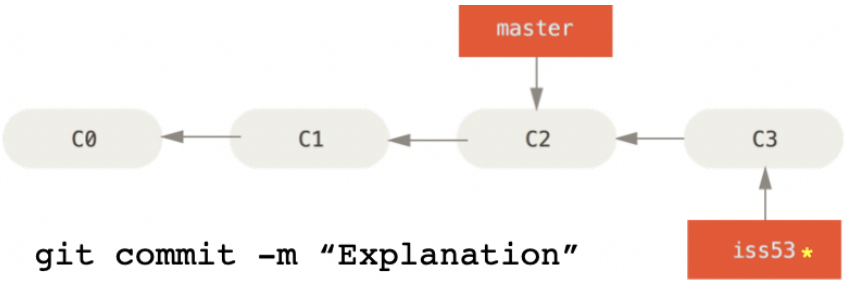


f. A new branch to work on issue 53

```
git branch iss53; git checkout iss53
OR
git checkout -b iss53
```

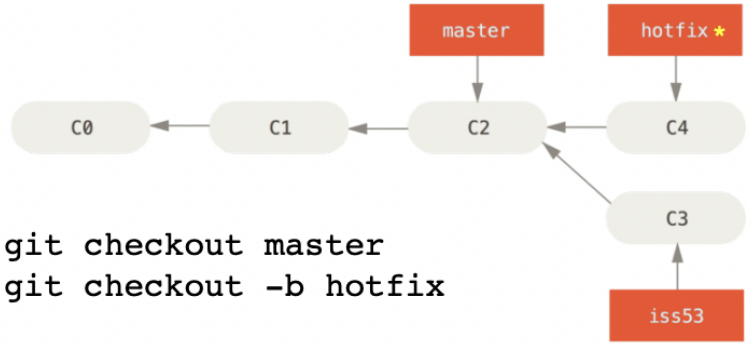


g. Add and commit changes in branch iss53




```
git commit -m "Explanation"
```

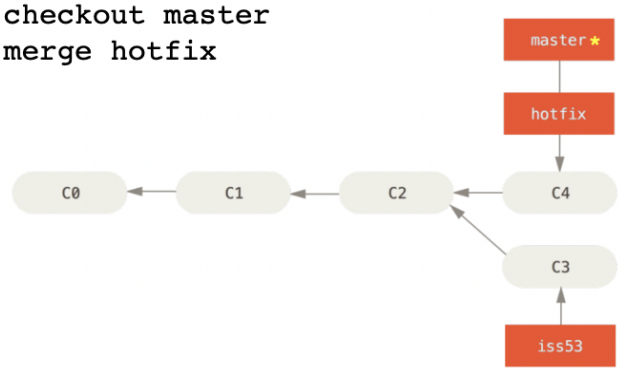
h. Move back to master, create a new branch to work on a hotfix



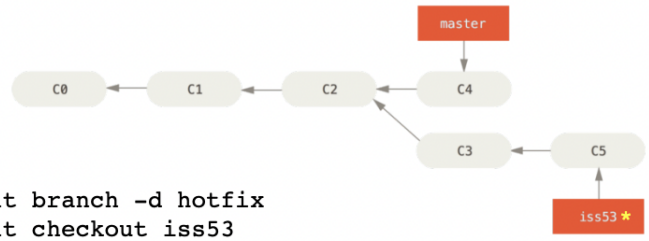

```
git checkout master
git checkout -b hotfix
```

i. Hotfix is merged into master

```
git checkout master
git merge hotfix
```



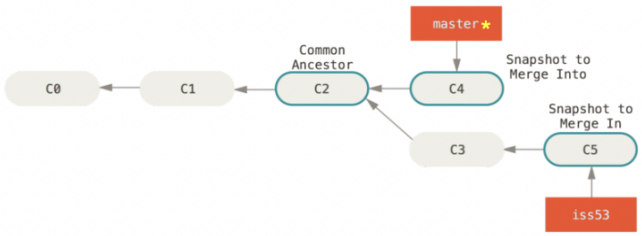
j. New commit on iss53…hotfix is no longer needed and is deleted




```
git branch -d hotfix
git checkout iss53
<some work is staged with git add>
git commit -m "fixing iss53 prob"
```

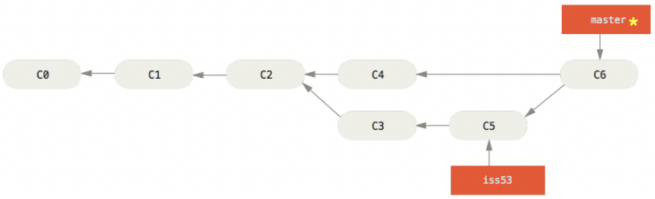
k. Getting ready to merge iss53 into master

```
git checkout master
```



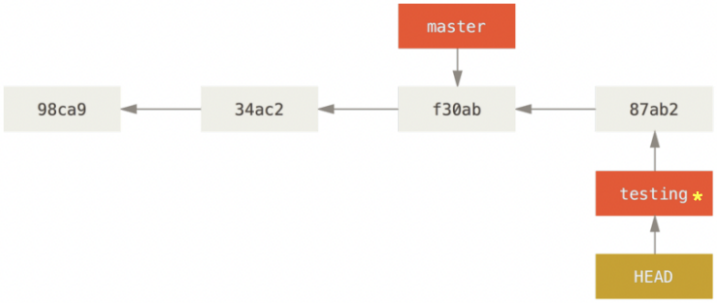
l. Merge iss53 fixes into master

```
git merge iss53
```



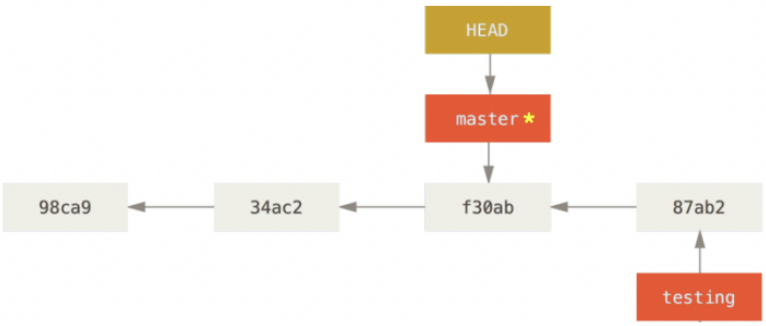
9. Basic flow
    a. Get up to date with the code you will be branching off of (often master but not always)
    b. Create a branch for a specific piece of functionality / bug fix
    c. Test your changes
    d. Merge your changes back into the main branch
    e. Delete the ‘feature’ branch
10. The HEAD pointer
    a. Git keeps a pointer, called HEAD, that refers to the commit that you are currently working at (that’s what the * was in the previous slides)
    b. Use git checkout to move HEAD around
    c. Normally we’re moving to the tip of a branch, but you can also move to a specific commit if you need to
    d. When you issue git branch, the branch is created from wherever HEAD is pointing to

e. After issuing git checkout testing



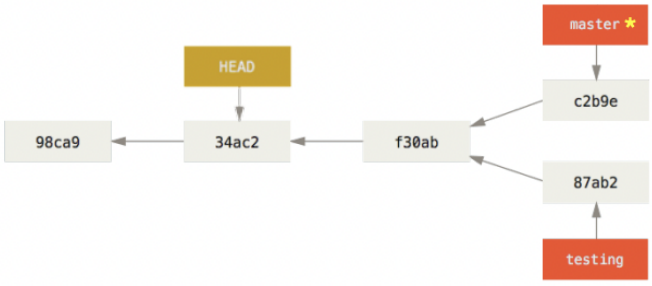
f.

g. Git checkout master moves back to the tip of master



h. Moving to a prior commit (this is called 'detached HEAD')

```
git checkout 34ac2
```



11. Stashing
    a. If you are working in a branch and have uncommitted changes, git will prevent you from switching
    b. This is because checking out a branch places all the files in your working directory in the state they were at when the branch was last committed
    c. That means that switching to a different branch when you have uncommitted changes in the current branch might overwrite those files
    d. To get around this, we use git stash to take a snapshot of those uncommitted changes

e. README.md has uncommitted changes

```
$ git commit -am "added Xs to README"
[quickTest 280414f] added Xs to README
 1 file changed, 1 insertion(+), 6 deletions(-)

$ vi README.md [make a change]

$ git checkout master
error: Your local changes to the following files would be overwritten by checkout:
    README.md
Please commit your changes or stash them before you switch branches.
Aborting

$
```

f. Git stash saves the uncommitted changes to a stack

```
$ git stash
Saved working directory and index state WIP on quickTest: 280414f added Xs to README

$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with ‘origin/master'.

$ git stash list
stash@{0}: WIP on quickTest: 280414f added Xs to README
$
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

nothing to commit, working tree clean
$


$
$ git checkout quickTest
Switched to branch 'quickTest'
$ git status
On branch quickTest
nothing to commit, working tree clean
$ head README.md
xxxx on quickTest

$ git stash list
stash@{0}: WIP on quickTest: 280414f added Xs to README

$ git stash pop   //or apply, which leaves stash on the stack
On branch quickTest
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
$ git add README.md
$ git commit -m "Forgot trailing Xs"
[quickTest 9bb1548] Forgot trailing Xs
 1 file changed, 1 insertion(+), 1 deletion(-)

$head README.md
xxxx on quickTest xxx
```

12. Conflicts
    a. Since git is distributed, it is possible (even likely) that two devs will work on the same file in different branches
    b. If the changes on the file conflict with each other, the conflict must be resolved
    c. This is usually a manual process
    d. The merge will pause to give you a chance to figure out which change to keep
    e. Once you are done, the merge resumes
13. Tools for managing conflicts
    a. A merge conflict creates a new file that marks the conflicting chunks
    b. You can open it with a text editor and resolve the conflict there, however it can get messy
    c. Most folks use a tool like mergetool (installed on MacOS when you install XCode) or gitKraken or others
        i. These tools give you a side-by-side view
        ii. They let you click-and-pick which part of the code to use or to drop
14. Both master and quickTest have Xs in line 1 but done differently … which is correct?

```
$ git stash drop
Dropped refs/stash@{0} (f2d78468f616bde989d34d7e322de7ce6d8d2b9f)
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

$ emacs README.md. [make a change]

$ git commit -am "Added Xs in master"
[master b93e76b] Added Xs in master
 1 file changed, 1 insertion(+), 6 deletions(-)
$
$ git merge quickTest
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
$
```

    a.
    b. README.md now has info about the conflict

```
$less README.md

<<<<<<< HEAD
*** These are in master ***
=======
xxxx on quickTest xxx
>>>>>>> quickTest
# `angular-seed` — the seed for AngularJS apps

This project is an application skeleton for a typical [AngularJS][angularjs] web
app. You can use it…
```

c. Fix the conflict (either manually or with a visual tool) and commit to complete the merge

```
$ emacs README.md [resolve conflicting code]

$ git commit -am "Fixed merge conflict in README, chose quickFix text"
[master a0e7b2b] Fixed merge conflict in README, chose quickFix text

$ git status
On branch master
Your branch is ahead of 'origin/master' by 4 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
$
```

15. Project workflow with gitHub
    a. There are two main ways to set up a repo on github
        i. Push existing local files to a new repo
        ii. Set up a new repo on github and clone it
    b. We'll look at the second method, which is the simpler of the two
        i. For the first method, a good tutorial is at https://www.digitalocean.com/community/tutorials/how-to-use-giteffectively
16. Project workflow: Setup for LEAD
    a. All members create gitHub account if it doesn't exist
    b. One team member (LEAD) creates a repo
    c. LEAD adds MEMBERs to collaborators list (add collabs in the Settings page (gear icon))
    d. MEMBERs accept email invite to be collaborator
    e. MEMBER navigates to github repo, click on green 'Clone or download' button, copy URL displayed
    f. MEMBER: From a terminal on your local machine, move to the directory you want your local repo to be
    g. MEMBER: git clone
17. Project workflow: Setup
    a. MEMBER creates a personal branch (i.e. perryd would do git checkout -b perry)
    b. MEMBER pushes personal branch to set up tracking (git push --set-upstream origin perryd)
18. Project workflow: doing work
    a. MEMBER: move to project directory on your machine
    b. Switch to your personal branch
        i. git checkout perryd
    c. Update with any changes made since last time you were working
        i. git pull origin master
    d. Create a new topic / feature branch to do work on a specific item
        i. git checkout -b oauth

e. After completing work on the topic branch, merge it into your personal branch git checkout perryd git merge oauth
f. Push your personal branch to the project's gitHub repo git push
g. Notify LEAD that your changes are ready to merge into the release branch with a pull request
    i. Log onto gitHub, navigate to project repo, click on New pull request
    ii. base: master <- compare: <your personal branch>
h. LEAD evaluates request, requests comments, merges into masterSave files on local branch to remote repo

19. Commands used in demo
    a. git init //create a new local repo (from current directory)
    b. git add . //add any existing files to local repo
    c. git commit -m "Message" file //commit local changes
    d. git remote add origin URL //connect to remote repo
    e. git remote -v //show remote repo connections
    f. git branch //display all branches
    g. git pull //fetch remote files