Git/GitHub Workshop

- What is Git?
- Anatomy of a git project
- Commits and the 3 file states
- .gitignore
- Branches
- Merging
- What is Github?
- SSH keys
- Remotes & Push/Pull
- Fork & Clone
- Pull Requests

*Sourced from the official git guide at git-scm.org

What is Git?

- Git is an open source **version control** system.
- Git is an application you install on your computer.
- Git itself is not Github, Github is a hosting service for git repositories. Basically, Cloud.
- type git —version into terminal to see if you have git installed and which version you have.

What is version control?

- Version control is a system that keeps track of changes to a file or a group of files over time so you can retrieve specific versions of the file(s) later.
- Git is considered a distributed version control system (DVCS) because any clients who check out a repository have a fully mirrored repository with all prior file histories. They essentially have a full backup.
- So, if any server or client dies, they can just check out the repo from another client or server who they were collaborating with and receive the full restore.
- A repo, or repository, is simply a place where the history of your work is stored.

Git Basics

- There are really 2 ways to get a git project on your computer: import an existing directory into git, or clone an existing repository from another server. We will talk about cloning later.
- git init - Creates an empty git repository or reinitializes an existing one in the directory you are currently in.
- After running git init, there will be a hidden .git file in the directory you are in. type ls -a to list all files including hidden files. You will see the .git file.
- After running git init in a directory, you now conceptually have a git project on your computer. So lets look at the anatomy of a git project.

The 3 sections

- Git uses 3 sections for a git project:

    1. **git directory** - where git stores the metadata and object database for your project. This is the most important part of git, and it's what is copied when you clone a repository from another computer/server. It is that hidden .git file.
    2. **working directory** - a single checkout of one version of the project. These files are pulled from the compressed database in the git directory and placed on disk for you to use. Any coding you do will be in the working directory.
    3. **staging area** - a simple file, generally contained in your git directory, that stores information about what will go into your next commit. Sometimes referred to as the index.

Working Directory

- Every file in your working directory has only 2 states - tracked and untracked.
- tracked means it was in the last snapshot, and untracked means it wasn’t.
- Once a file is tracked, it can then be one of 3 states: unmodified, modified, staged
- So when you first initialize a repository, nothing is being tracked. You will need to git add the files so they can be tracked by git.

Demo

Git Commits

- A git commit is synonymous with saving the state of your project in git.
- Every time you commit, git essentially takes a picture of what all of your files look like at that moment in time, and then stores a reference to that snapshot.
- For the sake of efficiency, if any files have not had any changes made to them since the last commit, git does not store those files again. It just stores a reference to the previous identical file it already has stored in a previous snapshot.

Commits are made locally

- A git commit is strictly a local operation.
- In fact, almost all operations in git are local, thats why its so much faster than previous version control systems.
- And since repositories have the entire history of the project, you can browse through the entire set of versions of all files while offline, without making any network calls. Technology rocks.

The 3 states

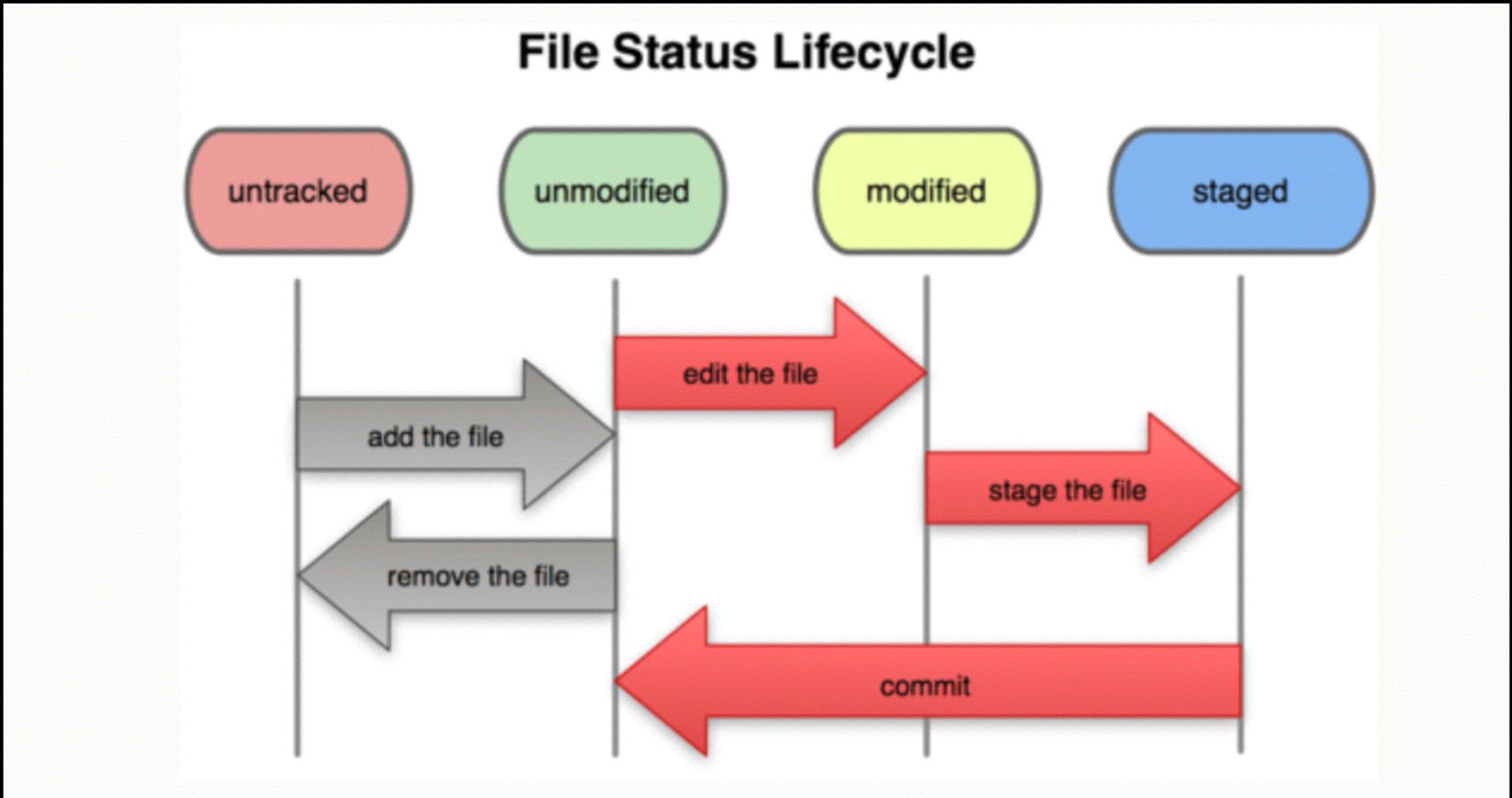
- Git has 3 main states your files can be in:

    1. Unmodified - data is safely stored in your local database. A tracked file without changes is in this state.
    2. Modified - you have changed the file but have not committed it to your database yet
    3. Staged - you have marked a file with changes to go into the next commit

Basic Git Workflow

1. You modify files in your working directory.
2. You stage the files, which adds snapshots of them to your staging area.
3. You do a commit, which takes files as they are in the staging area and stores snapshot permanently to your git directory.

Git File Status Lifecycle



git commit command

- Use the git commit once your staging area is set up the way you want it.
- Running just plane git commit will launch your text editor configured in your global git config.
- It does this because git wants a git commit message describing what this commit has changed.
- Alternatively, you can use the git commit -m “commit message” to enter your commit message inline.

demo

Checking the status

- Use the git status command to determine the state of your files in the git project
- It also tells you which branch you are on (more on branches later)
- If you see the message “nothing to commit, working directory clean” that means all files in the directory are tracked and have no changes to commit.
- if you are ever feeling confused or lost while working with a git project, git status is your best friend

Git add command

- It is important to understand that git add is a multi-purpose command:
  - used to begin tracking new files
  - to stage files
  - mark merge-conflicted files as resolved
- If you specify a directory instead of a single file with git add the commands adds all files in that directory recursively.
- git add -A runs the add command on all files in the repository

demo

.gitignore

- Often times you will have a set files that don’t want git to automatically add or even show you as being untracked.
- These are generally automatically generated logs, file system files, large SDKs or resources, or even secret API keys.
- In these cases, you can create a file called .gitignore that contains a list of the files or file patterns you want ignored by git.
- In this .gitignore file, you can list specific files, directories, or specific file types you want ignored.
- Use a # for comment lines
- github has a master list of best practice .gitignores for many programming languages.

objective c .gitignore

```
# Xcode
#
build/
*.pbxuser
!default.pbxuser
*.mode1v3
!default.mode1v3
*.mode2v3
!default.mode2v3
*.perspectivev3
!default.perspectivev3
xcuserdata
*.xccheckout
*.moved-aside
DerivedData
*.hmap
*.ipa
*.xcuserstate

# CocoaPods
#
# We recommend against adding the Pods directory to your .gitignore. However
# you should judge for yourself, the pros and cons are mentioned at:
# http://guides.cocoapods.org/using/using-cocoapods.html#should-i-ignore-the-pods-directory-in-source-control
#
# Pods/
```

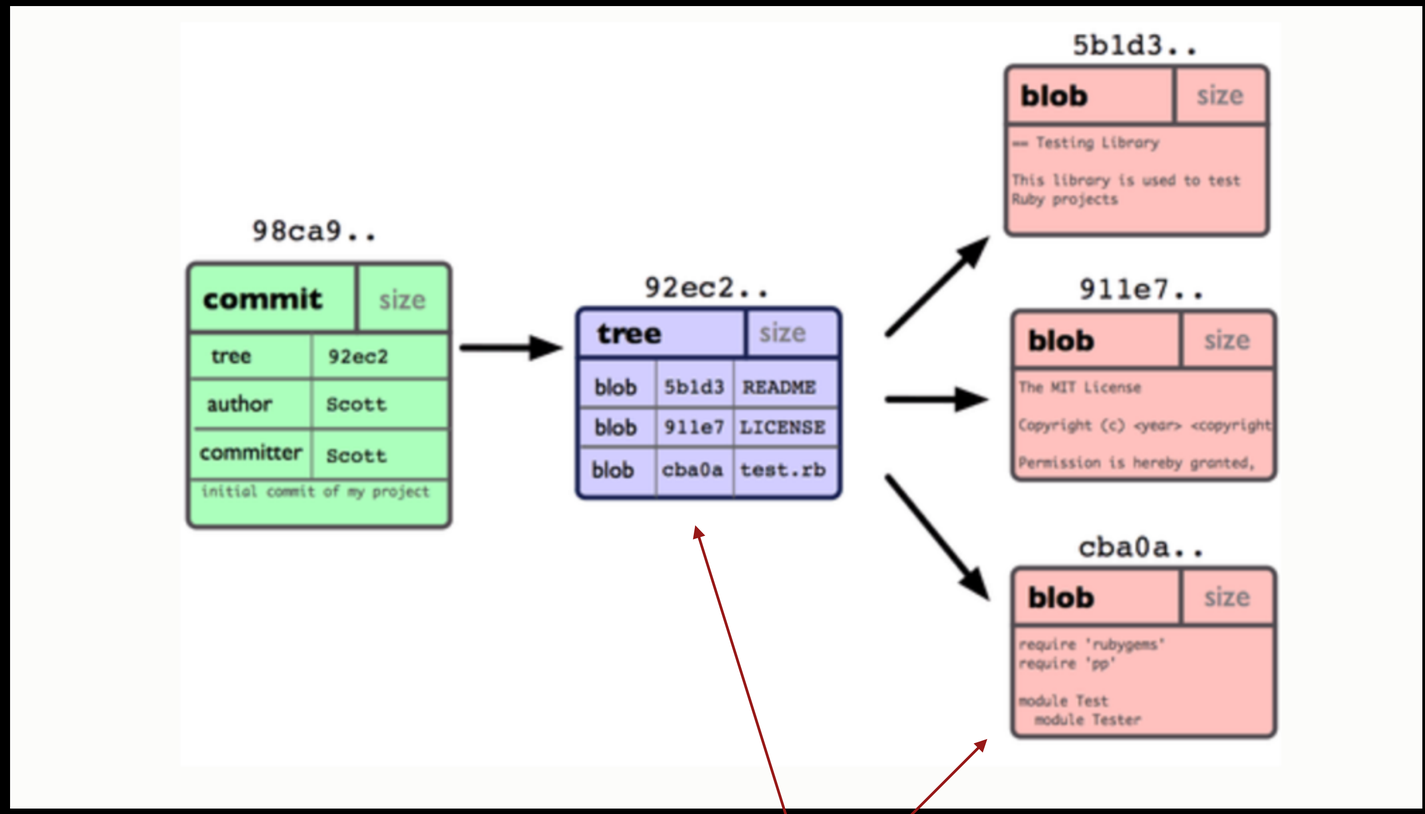
demo

Git Branches

- To understand what a branch is, you need to further understand how git works.
- Every commit is actually a commit object.
- that commit object has a pointer to the snapshot of the staged files.
- it also has a pointer to its direct parent commit:
  - zero parent pointers if its the first commit
  - one parent pointer if its a regular commit
  - two parent pointers if its a result of a merge

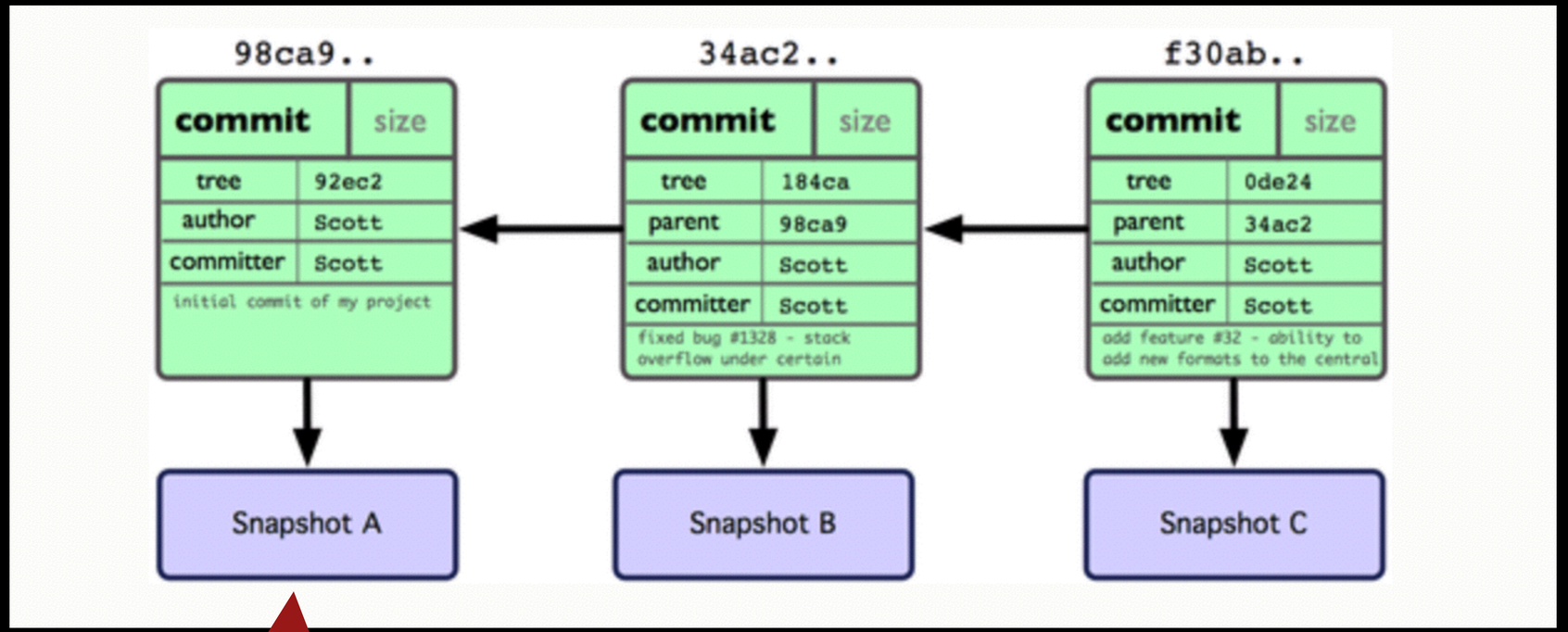
Anatomy of a Commit

- green: commit object containing metadata and pointer to tree
- purple: tree object lists the contents of the directory and specifies which files are stored as which blobs
- red: one blob file for every file in your project



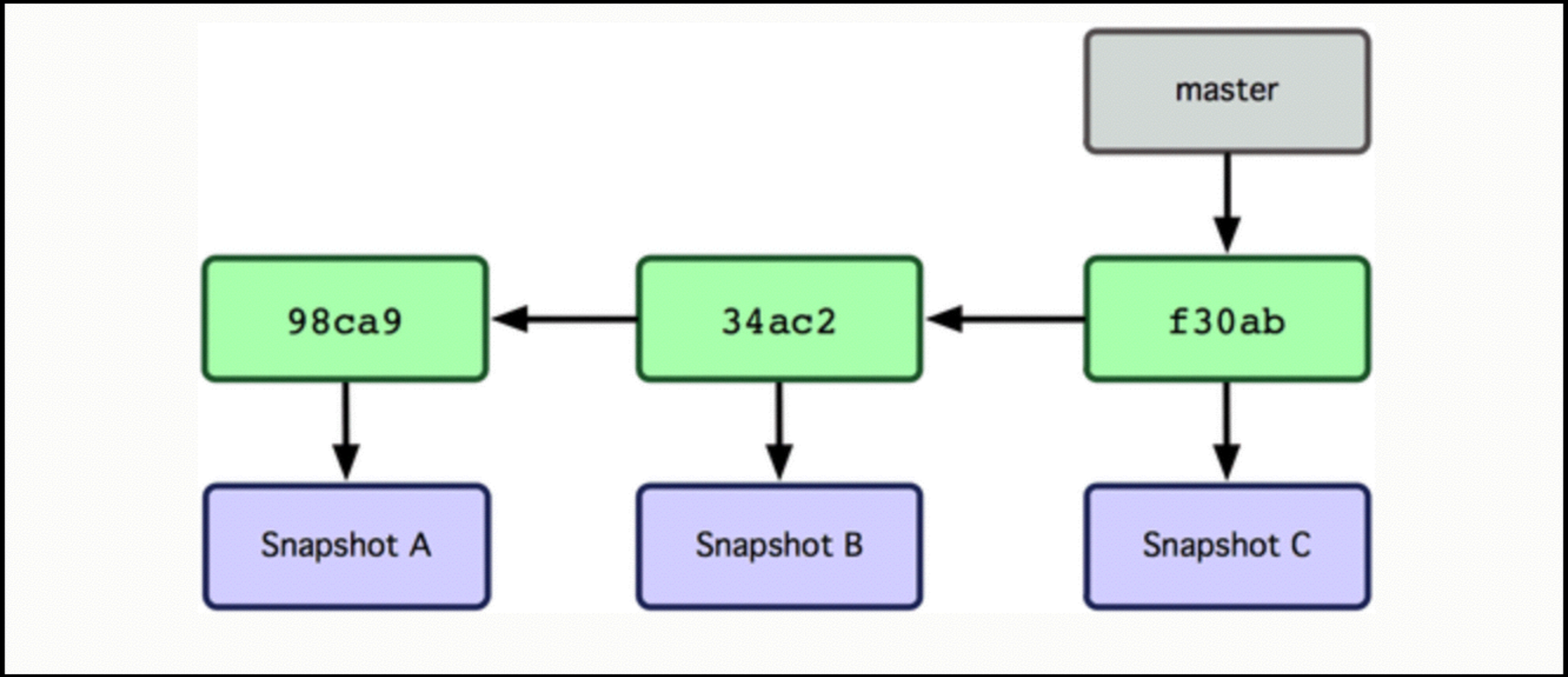
"Snapshot"

Multiple commits



initial commit

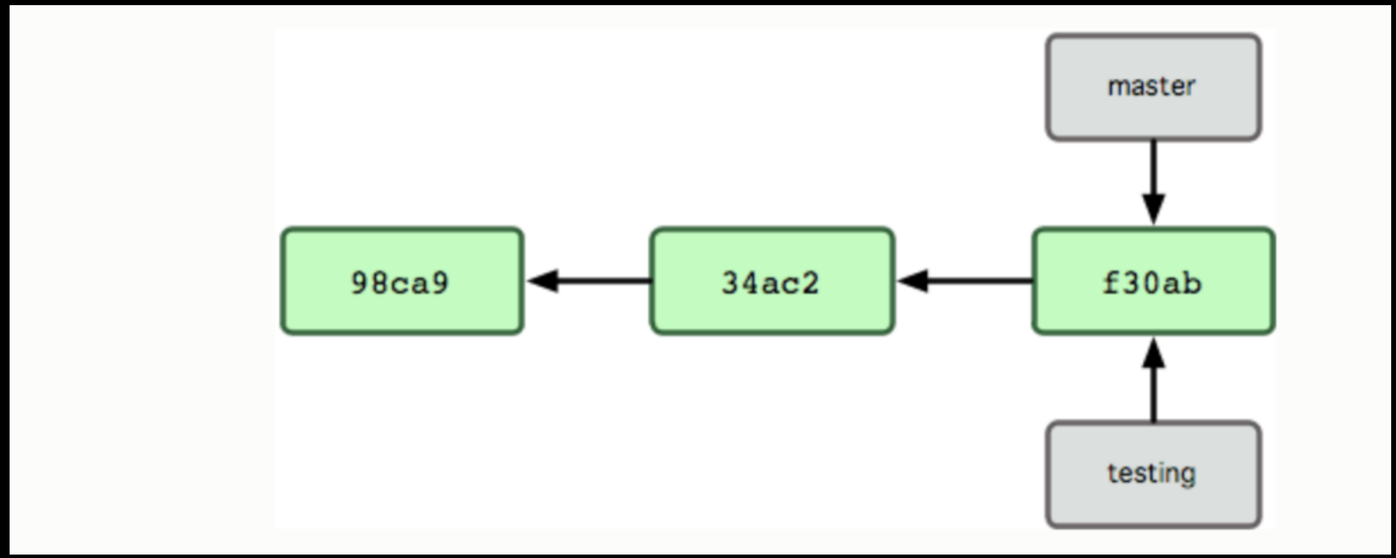
Branch == pointer



A branch is just a lightweight pointer to a commit

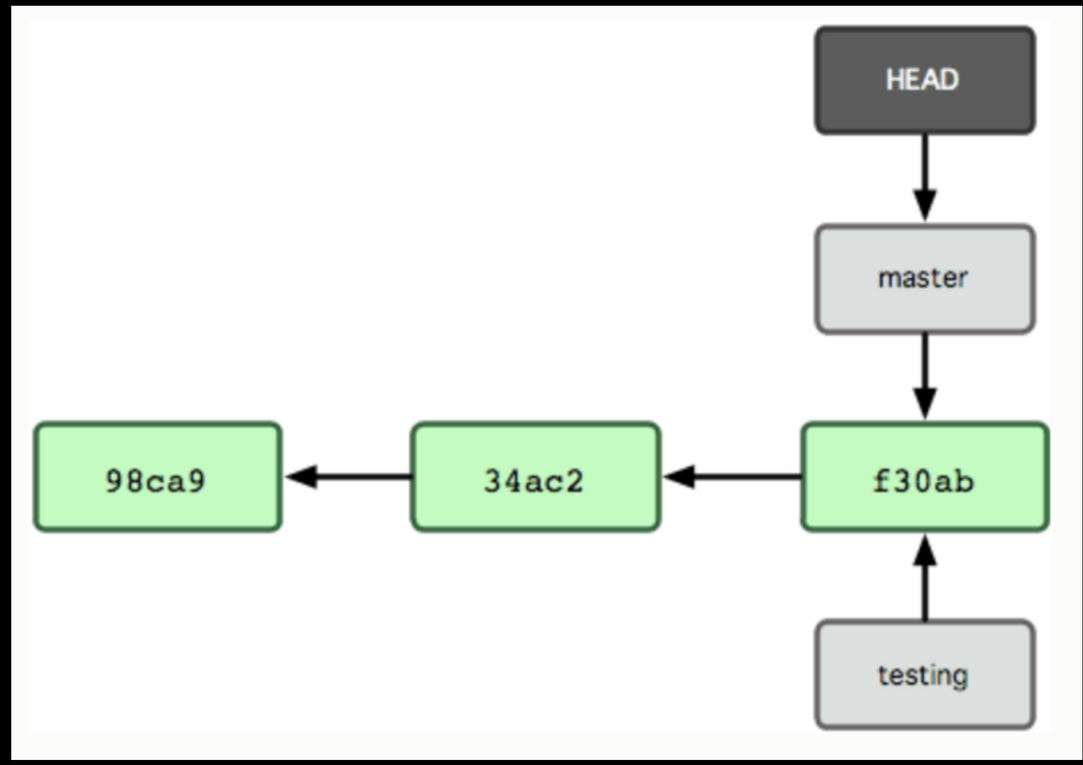
Creating a branch

- Use git branch <name> to create a new branch.
- Your new branch will point to the same commit of the branch you are currently on when you ran the that command



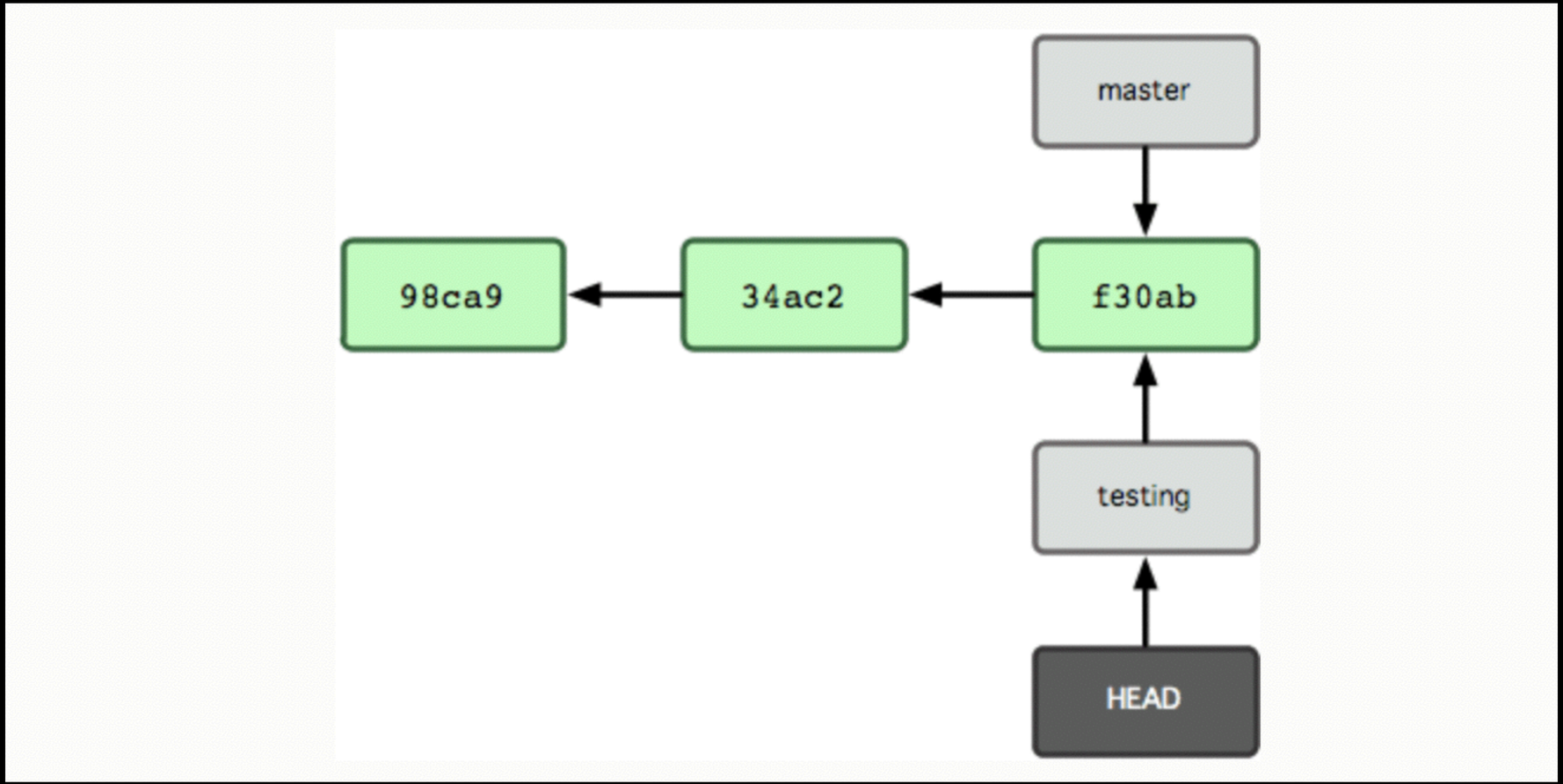
HEAD pointer

- Git keeps a special pointer called HEAD, which points to the current local branch you are on.
- Your working directory reflects the current commit your HEAD is pointing to.
- The branch command does not switch you to your new branch, it just creates it. So we are still on master here.



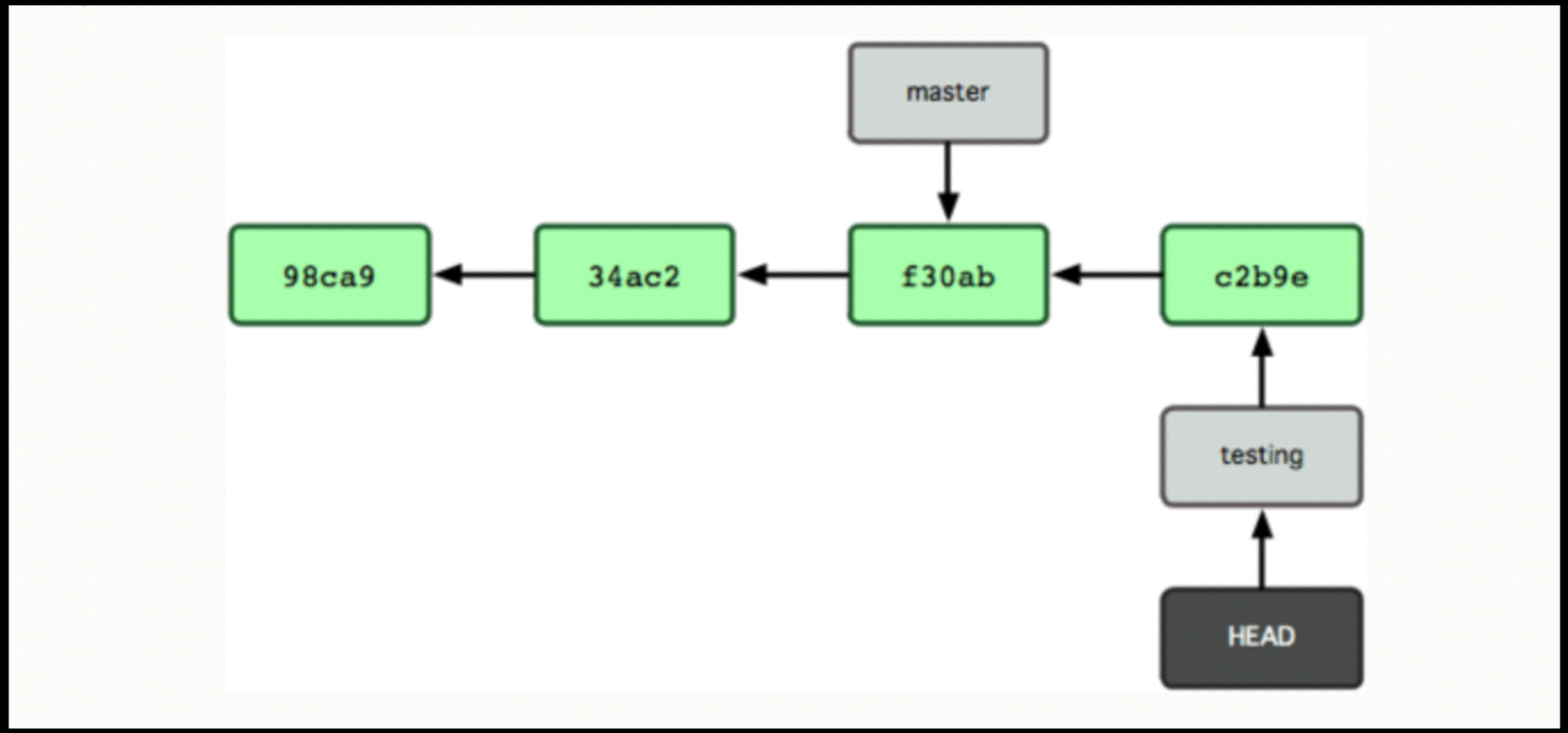
Git Checkout

- Running git checkout <branch name> switches HEAD to point to branch you specify



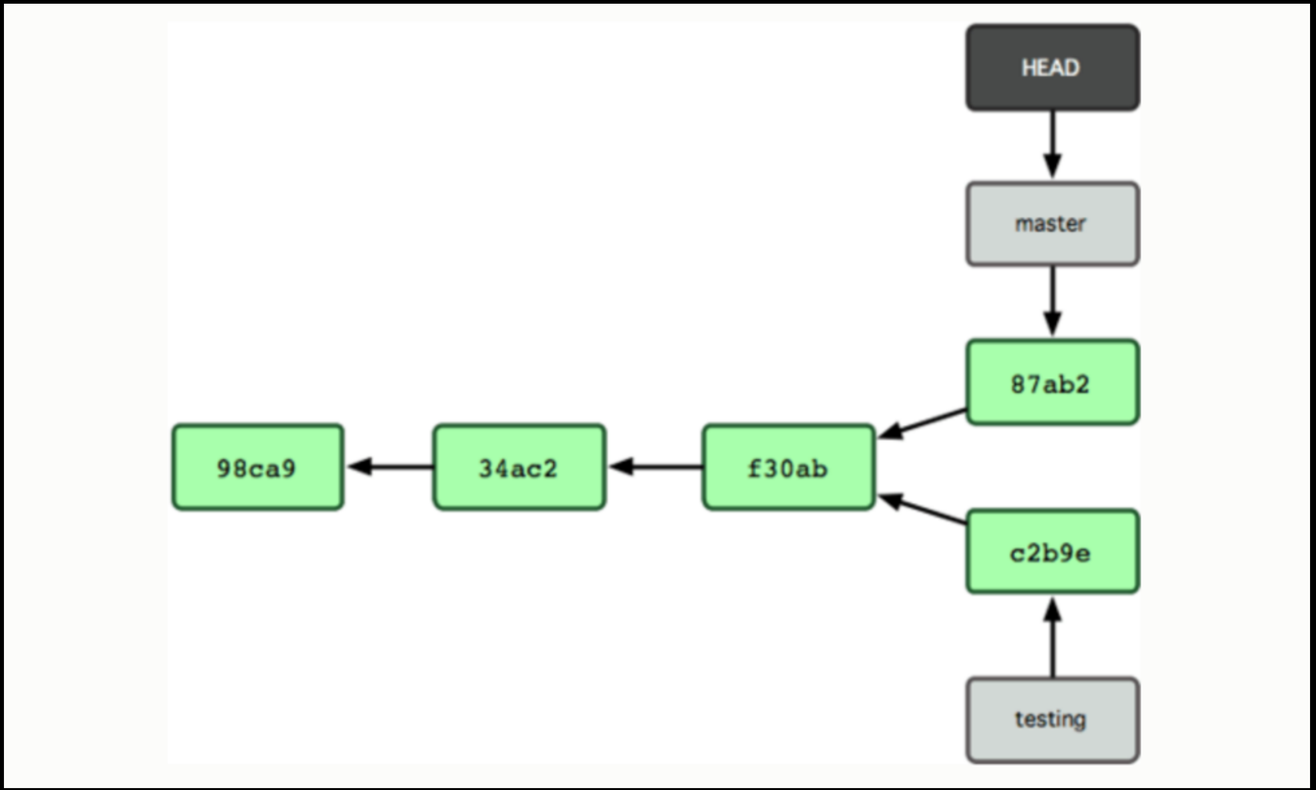
Moving forward

- If you commit now in your testing branch, testing moves forward while master stays put.



Fork in the road

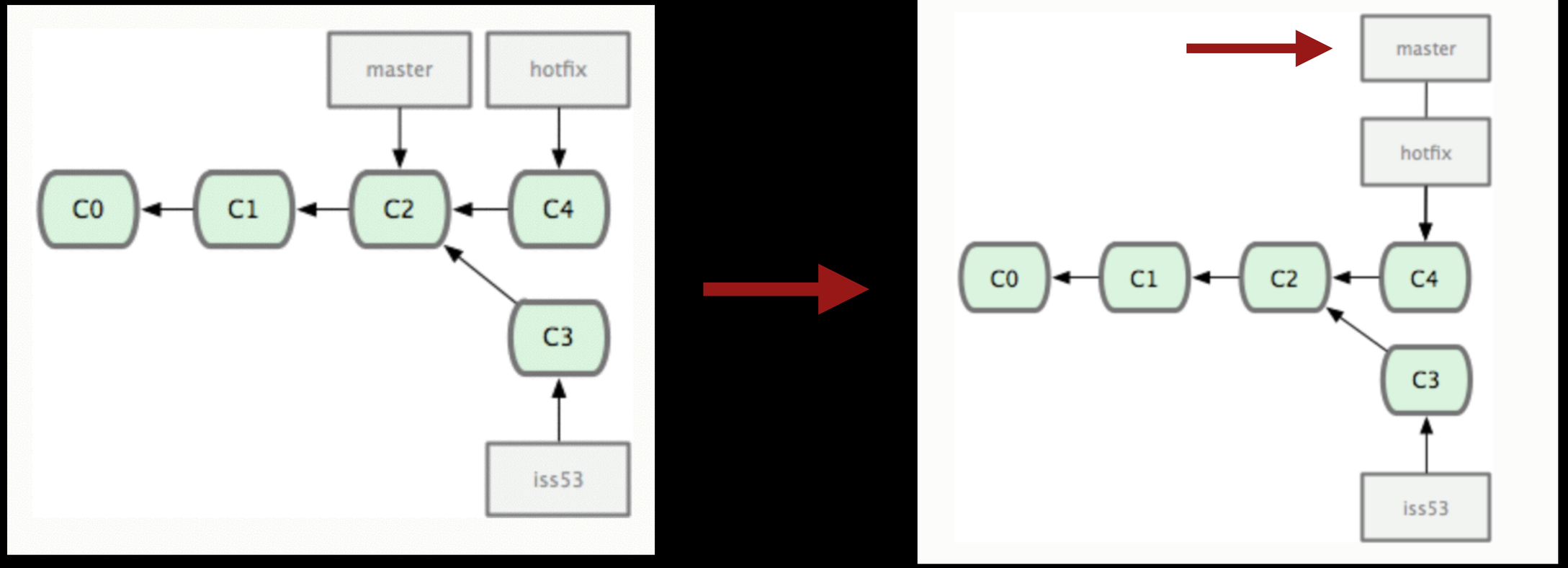
- Now if you checkout master and make commits, we have a divergence.



demo

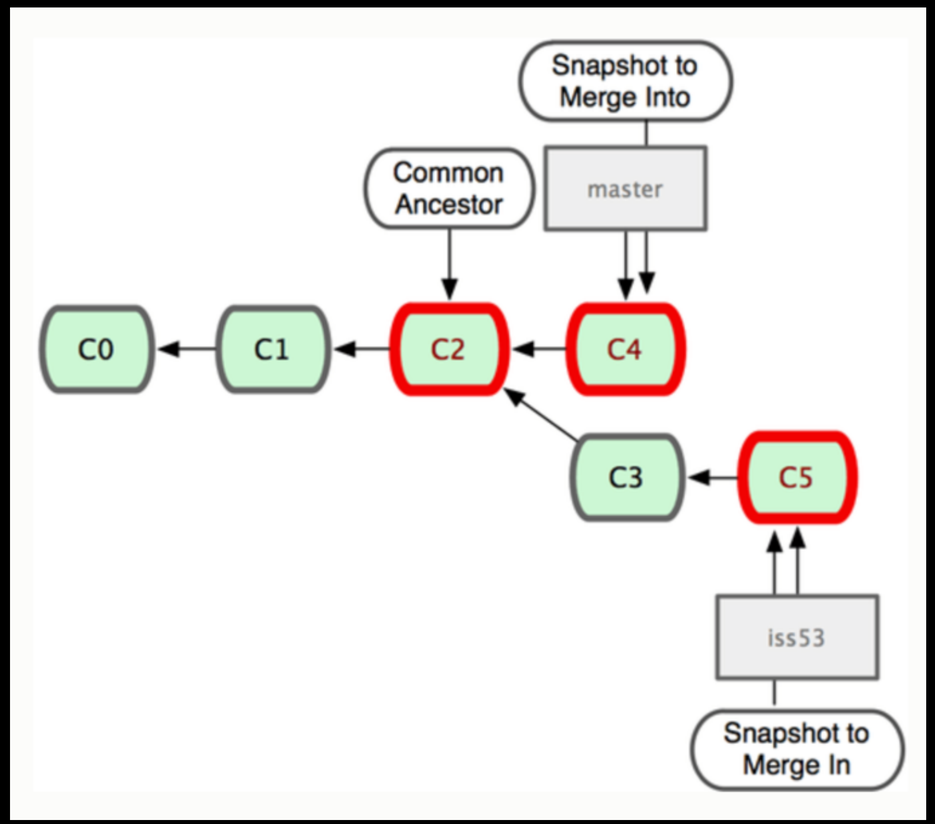
Merging

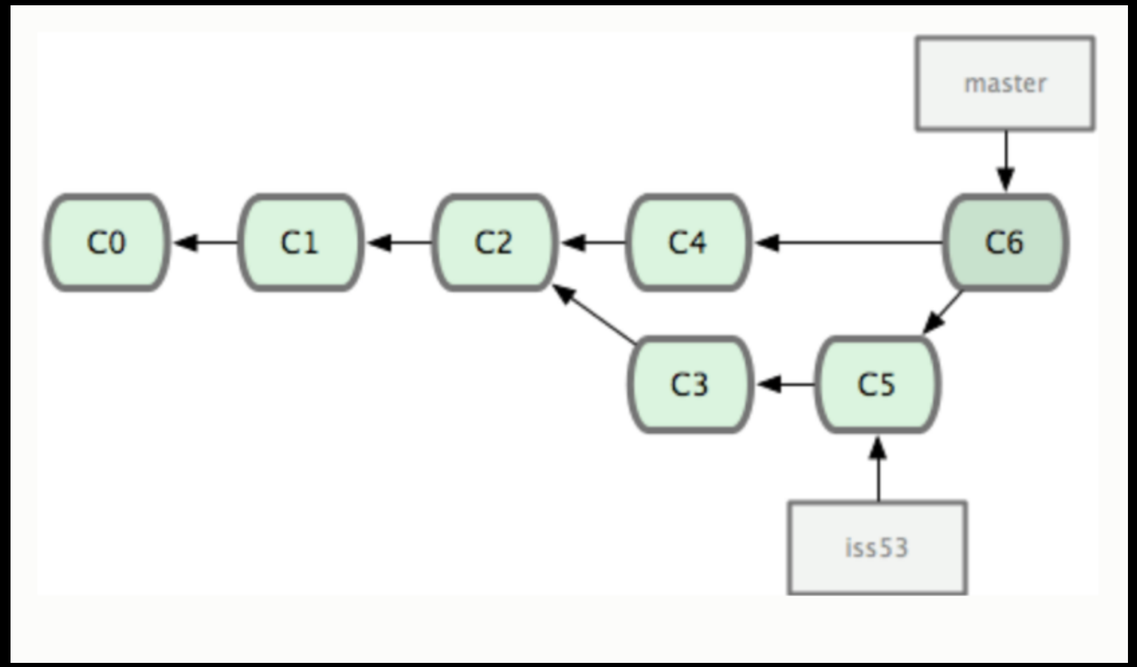
- Use the merge command to merge two branches pointing to different commits.
- A fast forward merge will not create a new commit:



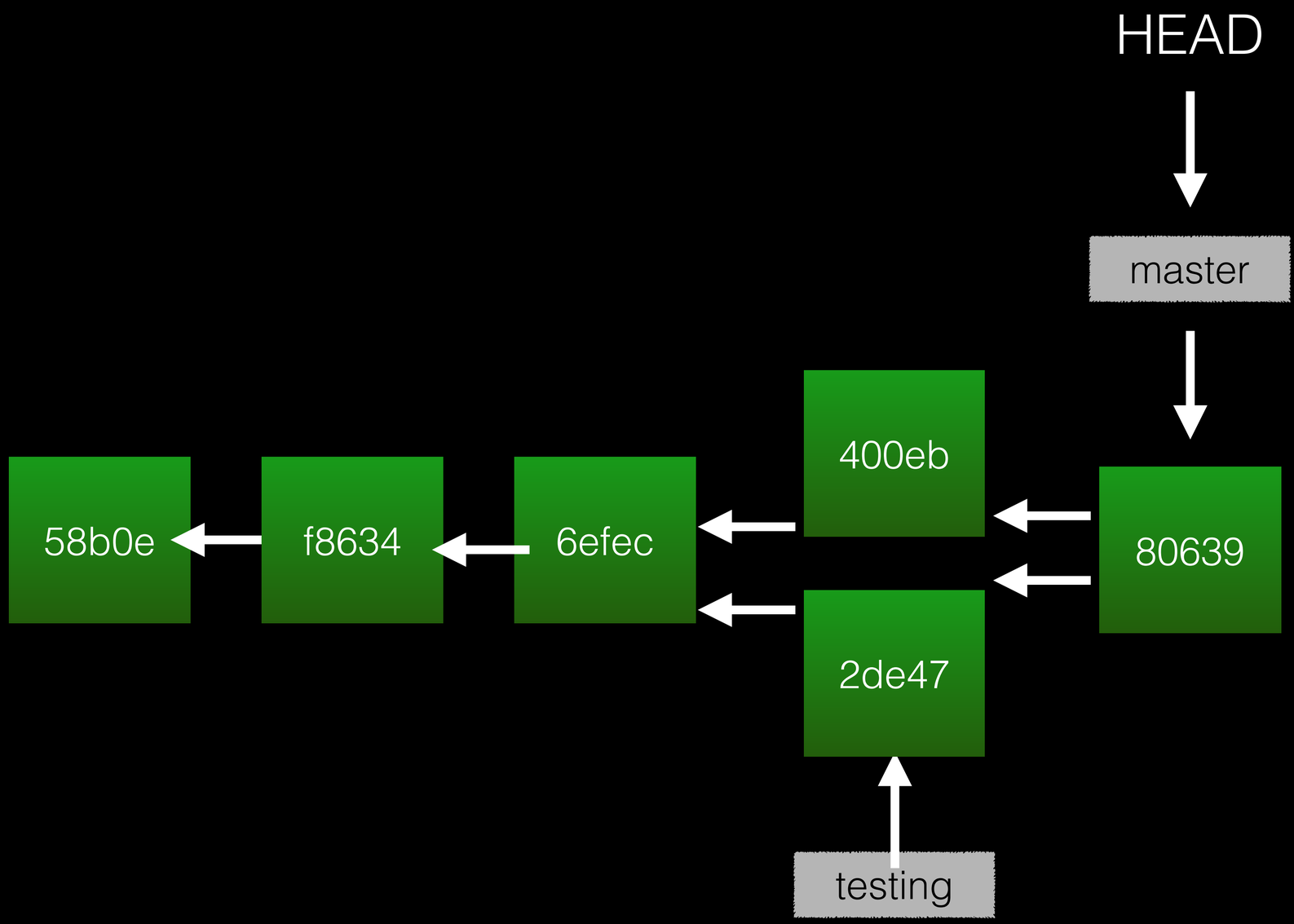
Merging

- A recursive merge happens when the two branches merging are not direct ancestors.
- A recursive merge will create a brand new commit as a result of the merge.





demo


HEAD
master
400eb
80639
58b0e
f8634
6efec
2de47
testing

Merge Conflicts

- Occasionally the merge process wont go as smoothly as we think it will.
- If you changed the same part of a file on the two branches you are merging, this will be a merge conflict.
- git will tell you theres conflicts in specific files, that the merge failed, and to fix the conflicts and then commit the results.
- Essentially git pauses the merge process until the conflicts are resolved.
- At anytime during a halted merge, you can run git status to see which files are still unresolved.

Resolving conflicts

- There are 2 ways to resolve the conflicts.
- Manually: Open each conflicted file and fix the conflicts line by line.
- Merge Tool: Use a merge tool that lets you choose which file's version of the conflicted code you want. This way is much less error prone.

Manual Resolution

- Git adds conflict-resolution markers to the files that have conflicts.
- Heres what they will look like when you open them manually:

```
<<<<<<< HEAD
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
  please contact us at support@github.com
</div>
>>>>>>> iss53
```

- The <<<<< HEAD denotes this is the beginning of the code that our local HEAD branch contains.
- The ====== signifies of the end of HEAD's version and the beginning of the branch we are trying to merge from.
- Finally, the >>>>>>iss53 signifies the end of the version of the code branch iss53 had
- Once we get rid of all the conflict markers (<<<<,======,>>>>>) in a file, we are ready to mark this file as resolved.
- Run git add on the file to mark it as resolved. staging the file tells git the conflicts have been resolved.

Merge Tool

- You can use a merge tool for a graphical interface based conflict resolution process
- use the git mergetool command to fire up the appropriate merge tool
- opendiff is the default merge tool if you havent configured git to use a different one.

demo

GitHub

- Github is a repository web-based hosting service.
- Github hosts people's repositories, and those repositories can be public or private.
- The repositories on Github are just like the repositories that you have on your own machine, except Github's web application has additional features that makes working in a team much easier.

- https://help.github.com/articles/set-up-git/

Remotes

- Remote repositories are versions of your project that are hosted on the Internet or network somewhere.
- So when you have a repository on Github, it is considered a remote repository.
- When you import a directory into a git project with git init, eventually you may want to create a remote repository on Github and push to it. This gives you a backup in the cloud, and also lets your work be seen by others.
- Or you can clone an already existing remote repository (your own or someone elses) to get a local copy of the remote repository on your machine.

GitHub and Git

- Github and your local git install have 2 ways of communicating:
  - https (recommended)
  - ssh
- Both of these forms of communication require authentication.
- For https, it is recommended you use a credential helper so you don't have to enter in your credentials every time you interact with a remote repository.
- For SSH, you can generate SSH keys on your computer and then register those SSH keys to your Github account.
- Lets all follow the steps at https://help.github.com/articles/set-up-git/ together to get our github & git properly setup.

demo

Git Remote Commands

- git remote -v shows you all the remotes you have configured for your local repository on your machine
- use git remote add <nickname> <url> to add a remote repo
- after committing, use git push <nickname> <branch> to push your committed changes to your remote
- use git pull to automatically fetch and merge a remote branch into your current local branch

Demo

Git Clone

- The git clone command clones a repository into a newly created directory, and creates remote tracking branches for each branch in the cloned repository (view them with git branch -r)
- **A remote branch is slightly different from a local branch.** The remote branches are just references to the state of branches on your remote repository. You don't move them yourself, they move automatically whenever you do any network communication with your remote.
- But the git clone command sets up remote tracking branches, which are local branches that have a direct relationship to the remote branches. You can push and pull from those tracking branches and they will automatically know which remote branch to work with.
- Cloning a repository gives you the complete history of that original repository.
- Cloning also automatically creates a remote that points to the repository you cloned from. Hooray!

GitHub Forking

- Forking a repository on Github is just making a copy of a repository.
- The forked repository is now your own repository.
- A common workflow for forking is:
  - Fork a repository
  - Clone your fork down to your machine
  - Fix a bug or add a feature
  - Push your changes up to your forked repository
  - Submit a pull request back to the project owner
- You will need to keep your force synced with the ‘upstream’, or original, repository. You can do this by fetching or pulling from upstream. You will need to manually add the upstream repository.

Demo

Github Pull Request

- Pull requests let you tell others about changes you've pushed to a GitHub repository.
- Once a pull request has been sent, the interested parties can review the set of changes, discuss potential modifications, and merge in or reject the submitted changes.

Pull Request Workflow

- 2 general work flows with pull requests:
  - Fork & Pull: Fork a repo, clone it and make your changes, push back up to your forked repo, then make a pull request back to the original repository.
  - Shared repository: everyone in a team gets collaboration rights (read & write) to the remote repository. People work on features in separate branches, and then do a pull request back to the main master branch once their feature is complete.

Fetch, Pull, and Push

- git fetch <remote> This command simply goes out to the remote project and pulls down all the data from that remote project that you don’t have yet. It does not do any merging.
- git pull <remote> <branch> Use git pull to automatically fetch from a remote branch and merge the remote branch into your current branch.
- git push <remote> <branch> When you have your project at a point where you want to share, you need to push it up to your remote.

Demo