

# Practical Git (and GitHub)

Ryan Morris



# Introductions



- ◉ Me
  - ◉ Reach me at [mr.morris@gmail.com](mailto:mr.morris@gmail.com)
- ◉ You
  - ◉ Any experience with version control?
  - ◉ Any experience with Git?
  - ◉ What are your goals for this class?

# Goals



- To work comfortably with Git from the **command line** (no guis)
- To be able to deal with the **typical frustration points**
- To get **exposure** to a broad range of Git functionality, as well as working with GitHub
- This class is geared towards **beginner-to-intermediate developers**

# Outline



- Day 1
  - Introductions
  - Git basics (A local repo)
  - Remotes
  - GitHub intro
  - Collaborating
- Day 2
  - Collaborating with git, continued
  - Git tools
  - Rebasing & advanced merging
  - Debugging (log, bisect)
  - Fixing (reset, revert)
  - Getting more out of git

# Class style



- Mix of lecture and labs
- I'll be working in the console, follow along
- Ask questions at any time
- And, some humble requests...
  - be on time after breaks
  - let me know if you need to duck out early
  - no cell-phones (take it outside)

# Resources



- The slides are available here:
  - <https://github.com/rm-training/resources>
- Grab and print a cheat sheet!
  - <https://training.github.com/kit/>
  - <http://zeroturnaround.com/rebellabs/git-commands-and-best-practices-cheat-sheet>
- Have some free time during labs? Read up!
  - The Git Parable
    - <http://bit.ly/1isB3K4>
  - Pro Git, 2<sup>nd</sup> edition (for free!)
    - <http://git-scm.com/documentation>



Now, before we begin...



- ◉ Forget everything you think you know about version control...

# Why Version Control?



# Why Version Control?



- ◉ **Keep track** of changes
- ◉ **Go back** to an older version
- ◉ **View a history** of changes

# Rudimentary version control!

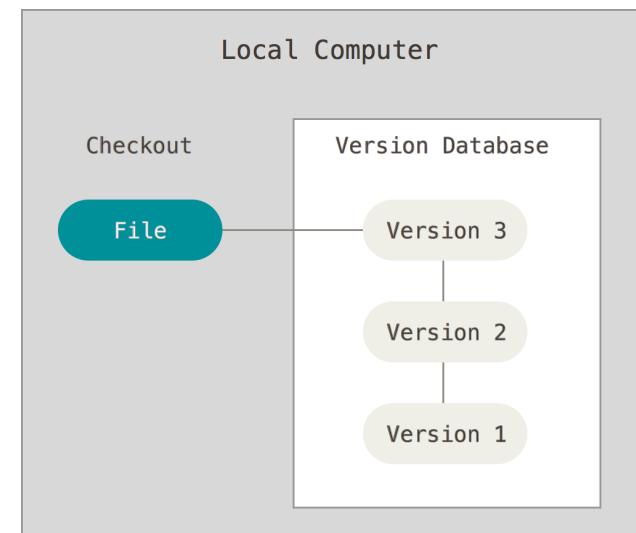


- ◉ Filename versioning
  - ◉ /my-files/myfile.v1
  - /my-files/myfile.v2
  - /my-files/myfile.v3.draft
- ◉ But...
  - ◉ Error prone
  - ◉ Single point of failure

# Centralized Version Control Systems



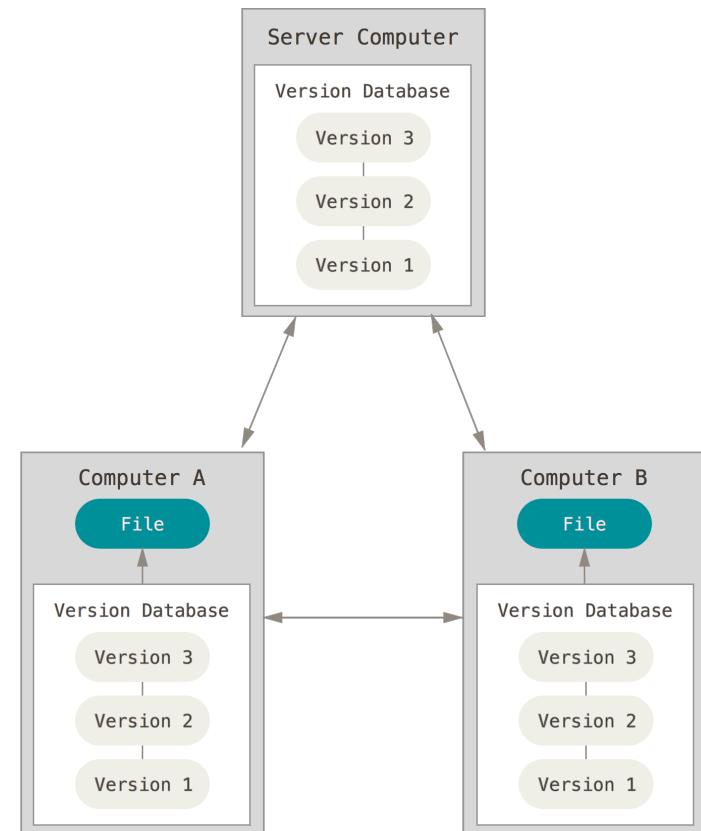
- A single, primary server manages all versions
  - CVS, Subversion, Perforce
- Fine-grained control and it is easy to see who is doing what
- But...
  - Still a single point of failure!
  - Branching + merging is a pain
  - Checkout locks are frustrating
  - Entire history is not available locally



# Distributed VCS

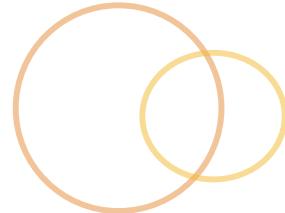
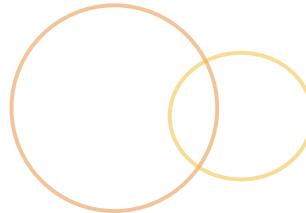


- All clients mirror the repository, *including* the entire history
  - Git, Mercurial, Bazaar
- All clients act as backups
- No single point of failure
- Supports many workflows





Git



- ◉ Distributed VCS
- ◉ Built by Linux maintainers
- ◉ Focuses on **speed, efficiency**, supporting **non-linear development** and very **large projects**
- ◉ At its core it is just a simple key-value data store
  - ◉ You can insert any content and it will give you a key you can use to retrieve the content

# What makes Git different



- **Snapshots**, not differences
- Nearly every operation is **local**
- **Data integrity**
  - Every file is *checksummed*
  - Data is never removed, always added
- **Branching** is easy
  - Yay for non-linear development!

# Module: Git Basics



- We'll cover...

- How to install git
- Creating a *local* repository
- The difference between staging and committing
- How to stage and commit files and changes
- Viewing a commit and your history
- Moving and removing files

# Install Git



- Install

- <http://git-scm.com/download/>
- Or... use a package manager like homebrew
- Or... install github's gui
  - <https://mac.github.com/>
  - <https://windows.github.com/>

- git --version

- Make sure you're on 2.0 or above

# Configure Git



- Use “git config” to set configuration vars
  - git config <key> <val>
  - git config --global user.name “Ryan Morris”
- Configs are stored in plaintext at one of three levels
  - System (all users) --system
    - /etc/gitconfig
  - Global (your user) --global
    - ~/.gitconfig or ~/.config/git/config
    - Windows: C:\Users\\$User\\$.Home\.gitconfig
  - Local (in a repo) --local
    - .git/config
- List configs
  - git config --list



# Mind the whitespace



- ◉ Windows uses CRLF, Mac/Linux use LF
  - ◉ Cross-platform work can be a pain
- ◉ `git config --global core.autocrlf <type>`
  - ◉ `input` (OSX/Linux)
    - ◉ Convert CRLF to LF on commit only
  - ◉ `true` (Windows)
    - ◉ Convert LF to CRLF when checking out a file
  - ◉ `false`
    - ◉ Do not perform any conversions
    - ◉ Use this if all collaborators are on the same system

# Lab: Setup



- Make sure git is installed
  - `git --version`
- Set up your identity
  - `git config --global user.name "Ryan Morris"`
  - `git config --global user.email me@gmail.com`
- And your CRLF setting
  - `git config --global core.autocrlf <val>`
- And *maybe* your text editor
  - defaults to vi or \$EDITOR
  - `git config --global core.editor emacs`
  - Or...
    - “sublime --n --w”
    - “atom --wait”
- Check your configurations
  - `git config --list`
  - `git config <key>`

# Creating a repository



- Make a root directory for the repo
  - `mkdir about-me`
  - `cd about-me`
- Initialize a new repository
  - `git init`
- Check the status and the log
  - `git status`
  - `git log`
- Check what git has initialized
  - `ls -la`
  - `ls -la .git`

# Our first commit



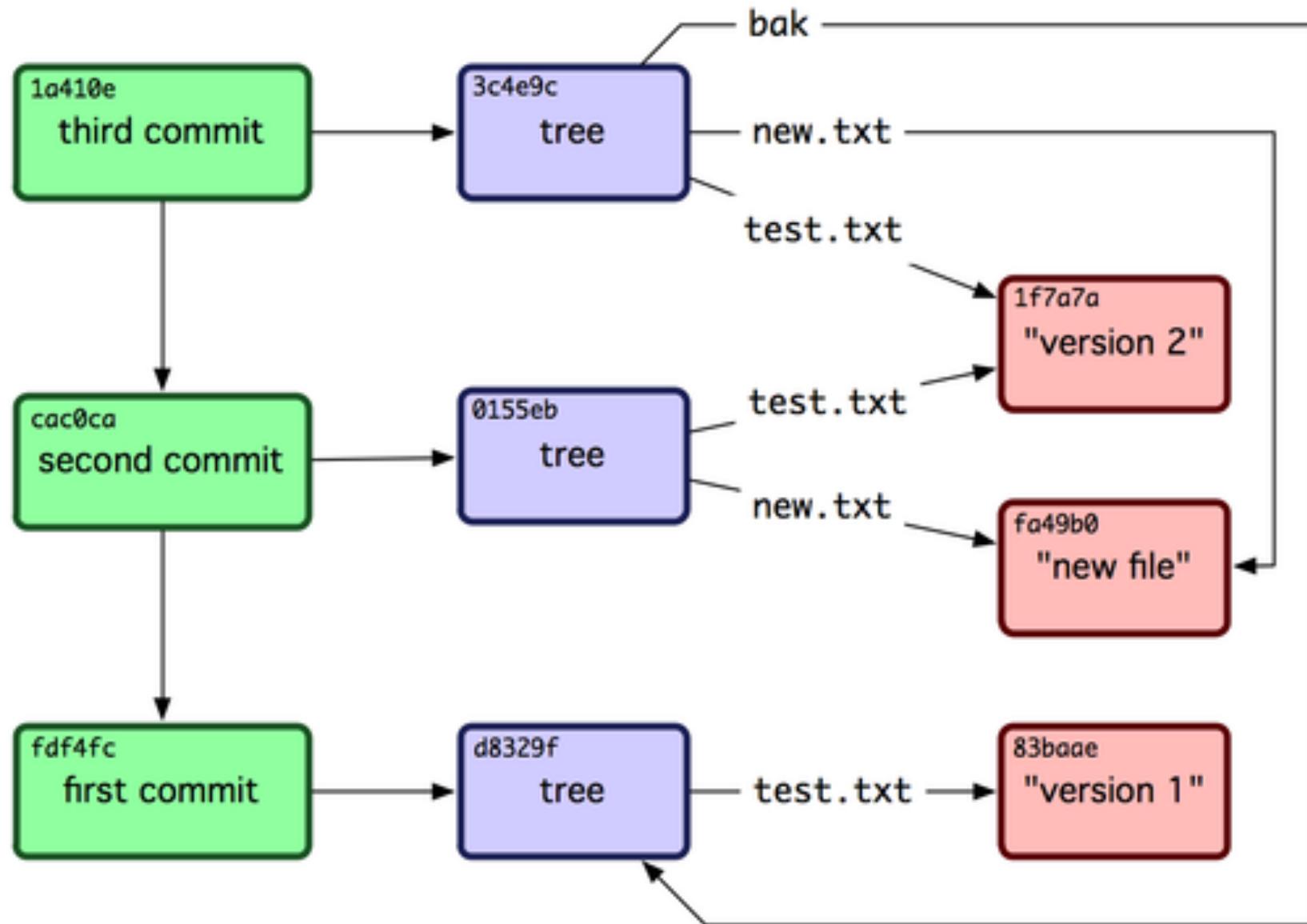
- ◉ Create a file, *stage* it, *commit* it
  - ◉ touch README
  - ◉ git add .
  - ◉ git commit -m 'Initial README'
- ◉ Check the status and the log
  - ◉ git status
  - ◉ git log

# What is a commit?



- Referenced by it's ***hash*** or ***id*** or ***sha***
- A **snapshot**
  - "the state of the files at a point in time"
- Each commit references
  - it's **parent commit**
  - **And the top-level directory** at that point in time

# Basic history of commits



# Tracked vs untracked



- ◉ Files are considered **tracked** or **untracked**
- ◉ Tracked
  - ◉ Any file the repository knows about
  - ◉ Staged, modified or otherwise
- ◉ Untracked
  - ◉ Any file that is new to the repository
  - ◉ Not-yet-staged

# Three trees

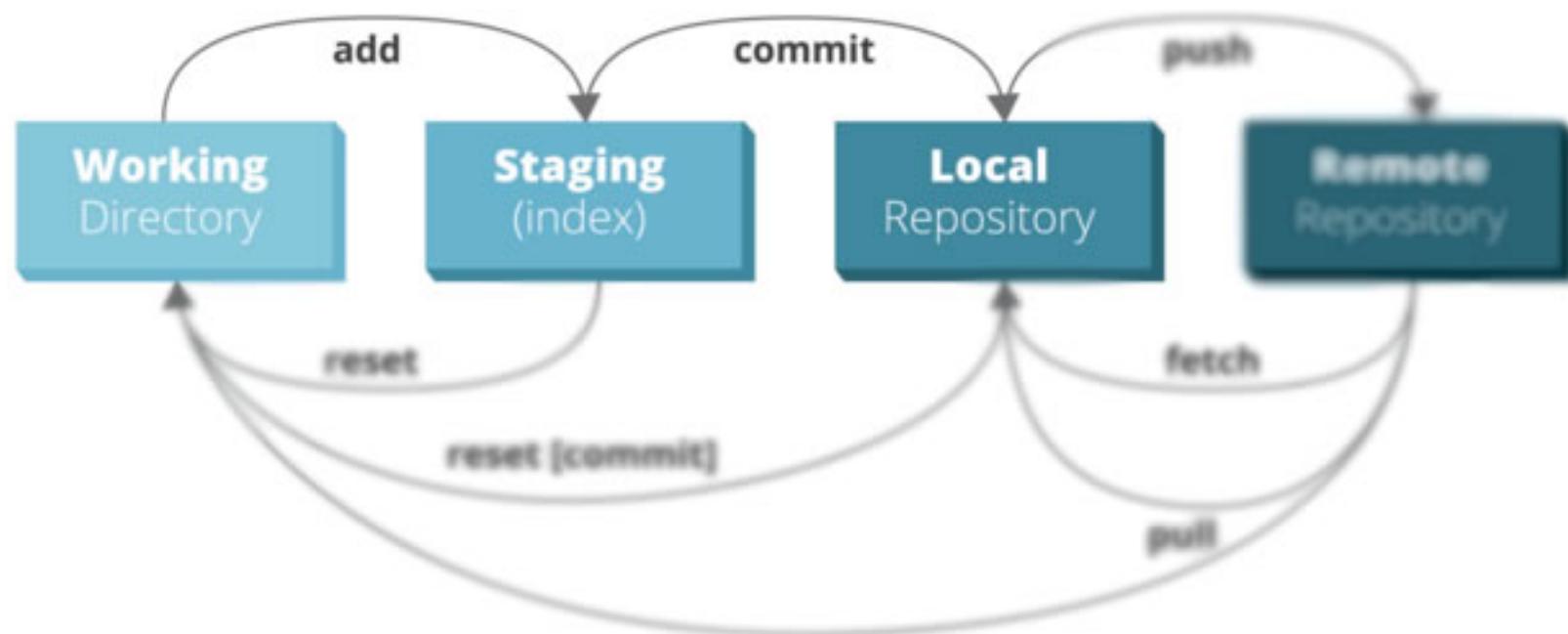


- The **working directory (WD)**
  - What you see in your filesystem
  - All files from your current commit/snapshot
  - *Along with modifications not yet staged*
- The **staging area (aka the index)**
  - Where you prepare your next commit
- The **repository**
  - All version data, every commit
  - A special pointer, **HEAD**, keeps track of your last commit

# The basic commit workflow



1. You **modify** files in your working directory
2. You **stage** files and changes that you want to be included in the next commit in the index
3. You **commit** the index as the next snapshot



# Git status



- `git status`

- Shows information about your **current branch**
- Changes added to the **staging area (index)**
- *Untracked files and tracked file modifications in the working directory*
- Some helpful tips for undoing things

# Staging files



- `git add <filename>`
  - To prepare a snapshot to commit, you'll add modifications and untracked files to the staging area with “git add”
- Stage a new file:
  - `touch NEWFILE`
  - `git add NEWFILE`
- Stage a modification:
  - `echo ‘Read Me’ > README`
  - `git add README`
- You can also add directories:
  - `git add <dirname>`
  - `git add <dir>/<subdir>`
- Use a wildcard:
  - `git add *.html`
- Or... just add everything:
  - `git add .`

# View your changes



- `git diff`
  - Shows diff output of what is in your **working directory**
  - Does not include untracked files, however
- `git diff --staged`
  - Shows diff output of what you have **staged**

# Committing



- ◉ `git commit`
  - ◉ Once files are staged, you can commit those changes as the next snapshot
  - ◉ Prompts for a commit message, which serves as the description
- ◉ `git commit -m "My commit message"`
  - ◉ A commit (as with many things) will abort if no message given
- ◉ `git commit -v`
  - ◉ Include file diff in the message meta
- ◉ `git commit -a`
  - ◉ Skip staging (auto-stage tracked file changes)

# Your commit(s)



- `git show <commit or branch>`
  - View commit info + diff
- Commits are referenced by their sha1 hash
  - aka commit id, hash, sha
    - ex: `42d484c401f0a19cc8a954c16240821329acefac`
    - `git show 42d484c401f0a19cc8a954c16240821329acefac`
- Can also reference in **abbreviated form**
  - `git show 42d4`

# Your history



- `git log`
  - View the history of commits
  - Defaults to from the current branch tip (HEAD)
- Plenty of options! We'll pick them up as we go...  
a few helpful ones
  - `--oneline`
  - `-<n>`
  - `--abbrev-commit`
  - `-p`
  - `--stat`

# Removing files



- `git rm <filename>`
  - Removes the file and auto-stages the removal
  - `-r` for recursive, like removing a directory
- If you simply **rm** the file, you must then **stage** the removal
  - `rm <filename>`
  - `git add <filename>`
- May only want to remove from staging w/out affecting WD (ie: to keep and ignore it)
  - `git rm --cached <filename>`

# Moving files



- `git mv <from> <to>`
  - Moves the file and stages the move
- But git doesn't track file movement, really... So this is basically equivalent to
  - `mv <from> <to>`
  - `git rm <from>`
  - `git add <to>`

# Recap



- We initialized a repository with **git init**
- Kept track of staging and working directory changes with **git status**
- Added changes to staging with **git add**
- Committed with **git commit -m “Message”**
- Checked diffs with **git diff**
- And then viewed our history with **git log**
- We also were able to move and delete files through git

# Lab: Basics



- First, **create** a new repository, “about-me”
- Then create a txt file named with your name
  - Touch <yourname>.txt
  - Check the status, check the diff
- Then **stage** and **commit** the file
  - Check the log
  - Use “git show” on the commit id you just created
- Then **edit** the file to add a short profile about you:
  - <Your Name>  
\* Born in: <where you were born>
  - Check the status, check the diff
- Then **stage** those changes
  - Check the status, check the diff
- Then **commit**
- All done?
  - Try creating a new file, “junk”, add then commit it
  - Try moving the “junk” to “junk.txt”, add then commit that change
  - Check out the git log

# Module: Basic undoing



- ◉ We'll learn about undoing some things like
  - ◉ Un-staging a file
  - ◉ Un-modifying a file in your working directory
  - ◉ And changing your last commit

# Undoing things



- ◉ First, some prep
  - ◉ Make a change to a file, add it
  - ◉ Make another change to the file
  - ◉ `git status`
- ◉ `git status` tells us how to undo things in staging and our working directory

# Undoing things



- Remove modifications from your working directory
  - `git checkout -- <file>`
- Remove something from your staging area
  - `git reset <file>`
  - Which is equivalent to: `git reset HEAD <file>`
- Can remove untracked stuff with
  - `git clean -f`

# Undoing things



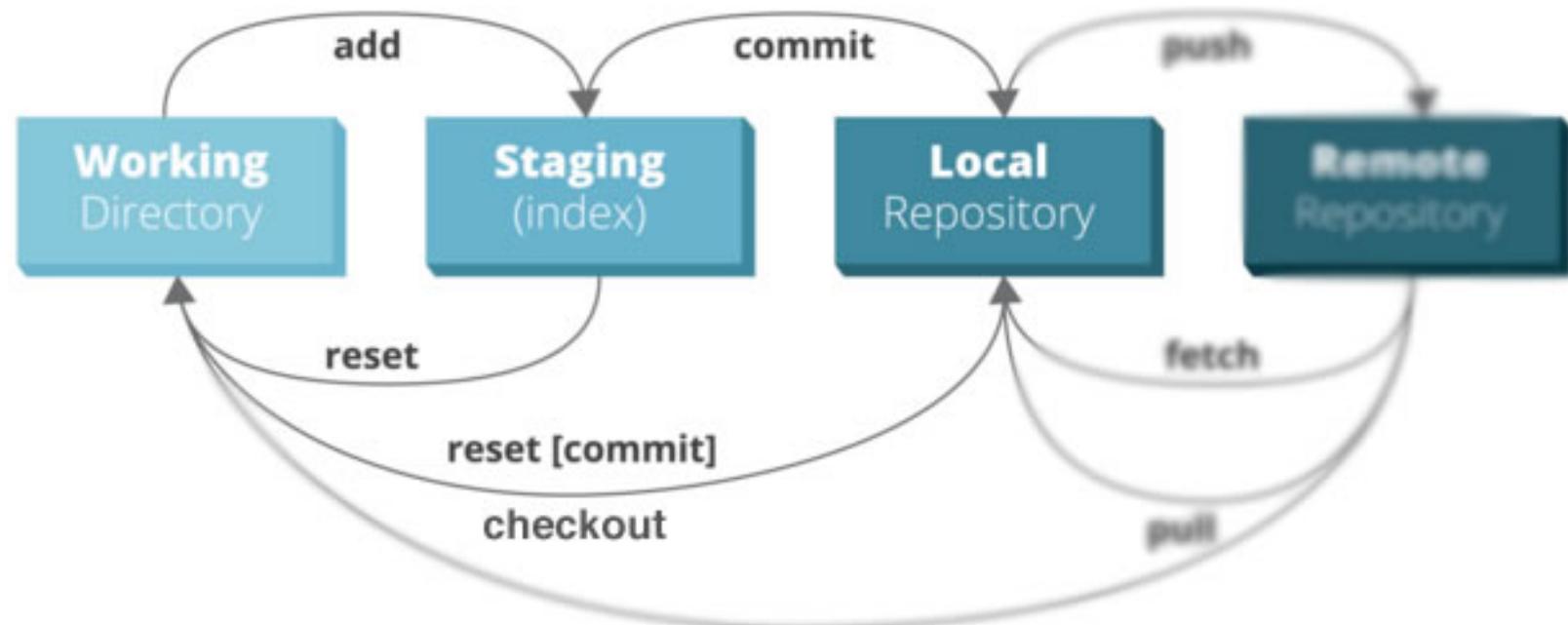
- Change the last commit
  - git commit --amend
  - To add forgotten changes
    - add them to your staging area then amend
  - To modify the commit message
    - just keep your staging area clean and amend
- To completely undo a commit
  - git reset HEAD^
  - We'll revisit this...

# Recap



- You can **un-stage** things with
  - `git reset <file>`
- You can **un-change** files in working directory with
  - `git checkout -- <file>`
- You can modify your last commit and commit message with
  - `git commit --amend`
- Remove untracked files with
  - `git clean -f`
- And undo the last commit with
  - `git reset HEAD^`

# The three trees, revisited



# Lab: Undoing things



- Take some time to practice editing files then ditching the changes with “git checkout” and un-staging them with “git reset”
  1. Create two new files, “file1” and “file2”
  2. Edit your existing “<yourname>.txt” file, adding a new point about yourself
  3. Stage all the changes
    - But don’t commit!
  4. Now... un-stage “file1” and the changes to “<yourname>.txt”
  5. Remove the changes in the working directory to “<yourname>.txt” and also delete “file1”
  6. Now you can commit (it should contain the new file, “file2”)
  7. Undo the commit then delete “file2”

# Module: Branches



- ◉ We'll learn about
  - ◉ **Branching** in git
  - ◉ Moving between branches with `git checkout`
  - ◉ What the `HEAD` is all about
  - ◉ **Merging** branches
  - ◉ **Deleting** branches

# Branches



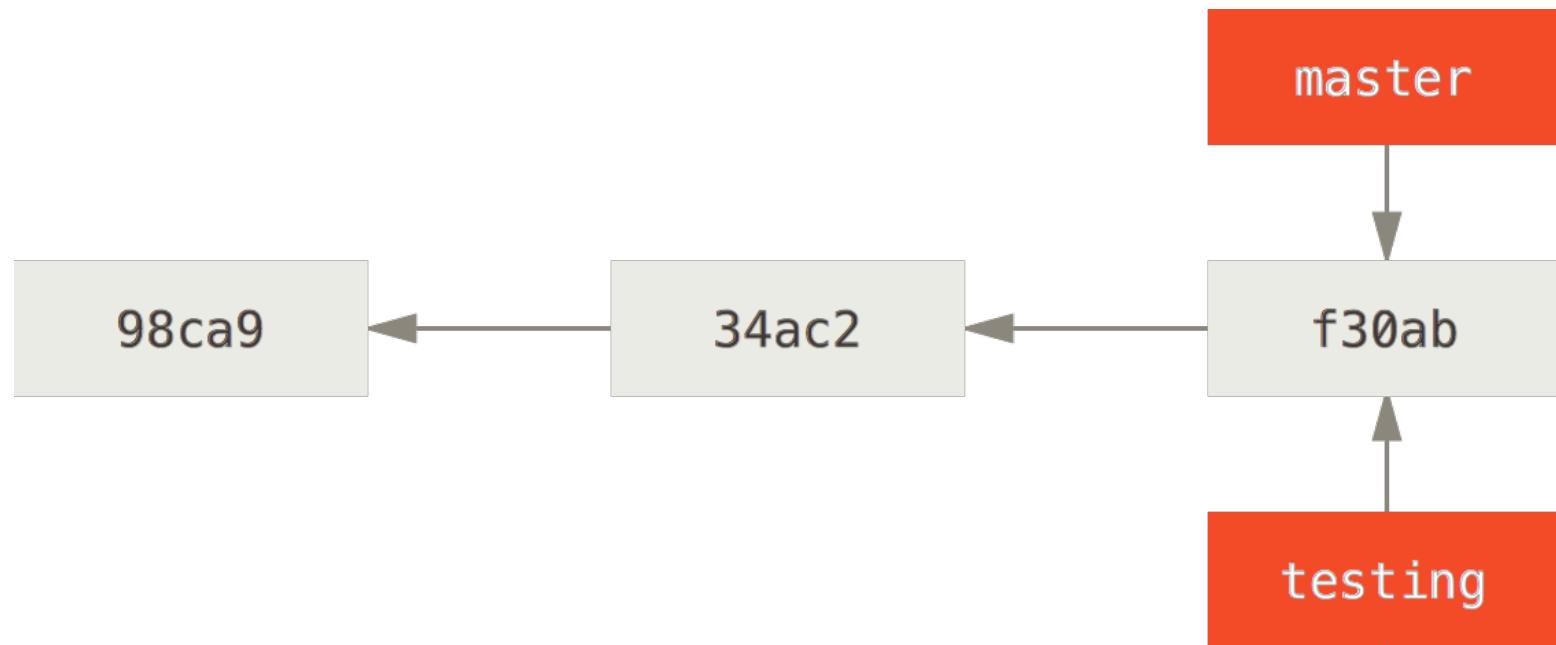
- ◉ Up until now we've been working on a single branch, **master**
- ◉ We'll begin using branches to work on many tracks of development at once
- ◉ Why branch?
  - ◉ Experimentation
  - ◉ Stability
  - ◉ Non-linear development
  - ◉ Diverging codebases or bucketing versions
  - ◉ Supports deployment workflows

# What is a branch in Git?

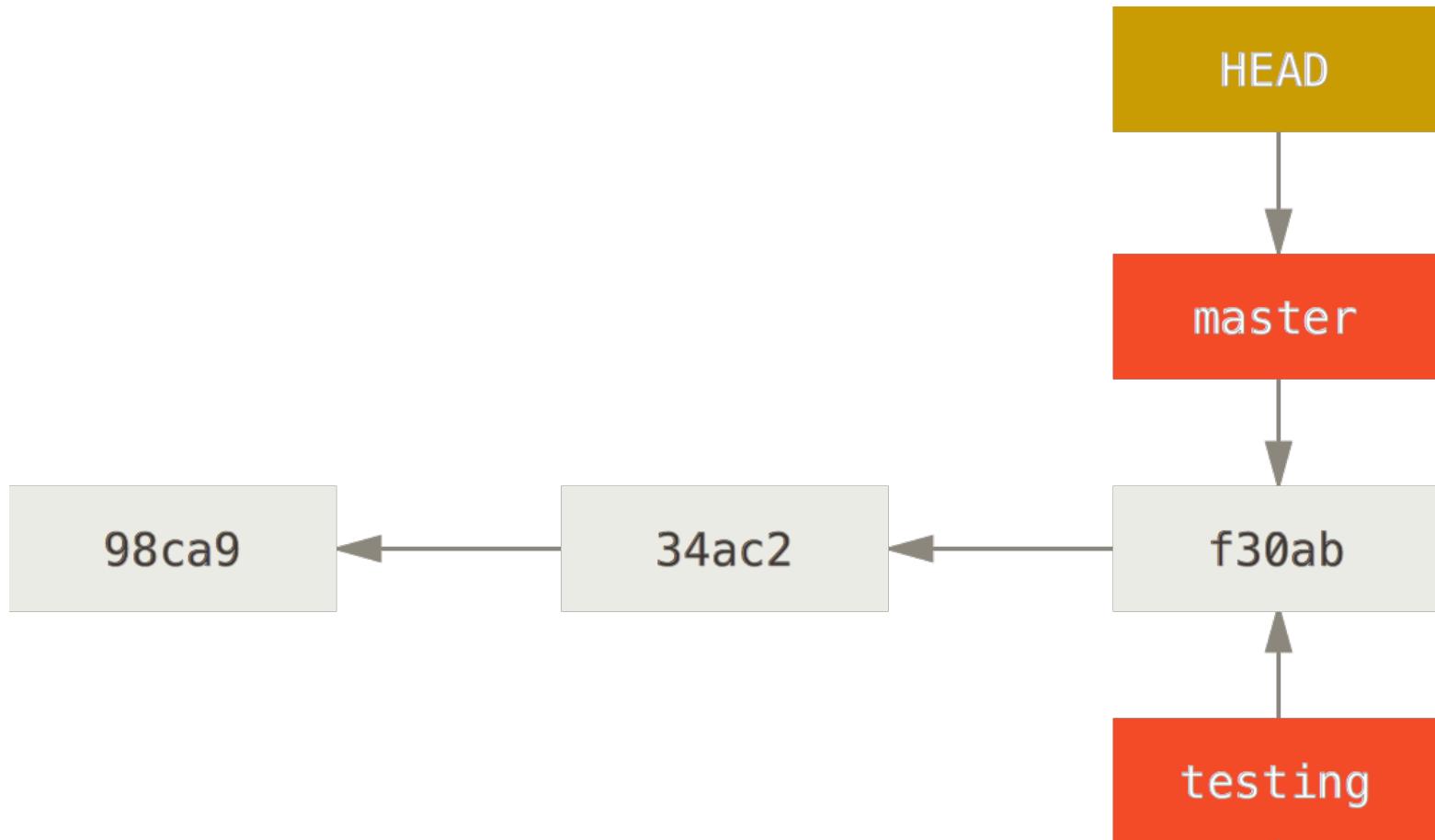


- A branch is a bookmark to a commit
  - The "tip" of the tree/history
- It's one line of text in the database, it's cheap!
- Creating a commit automatically moves the branch forward (by that commit)
- We can combine branches (later) by **merging** their histories

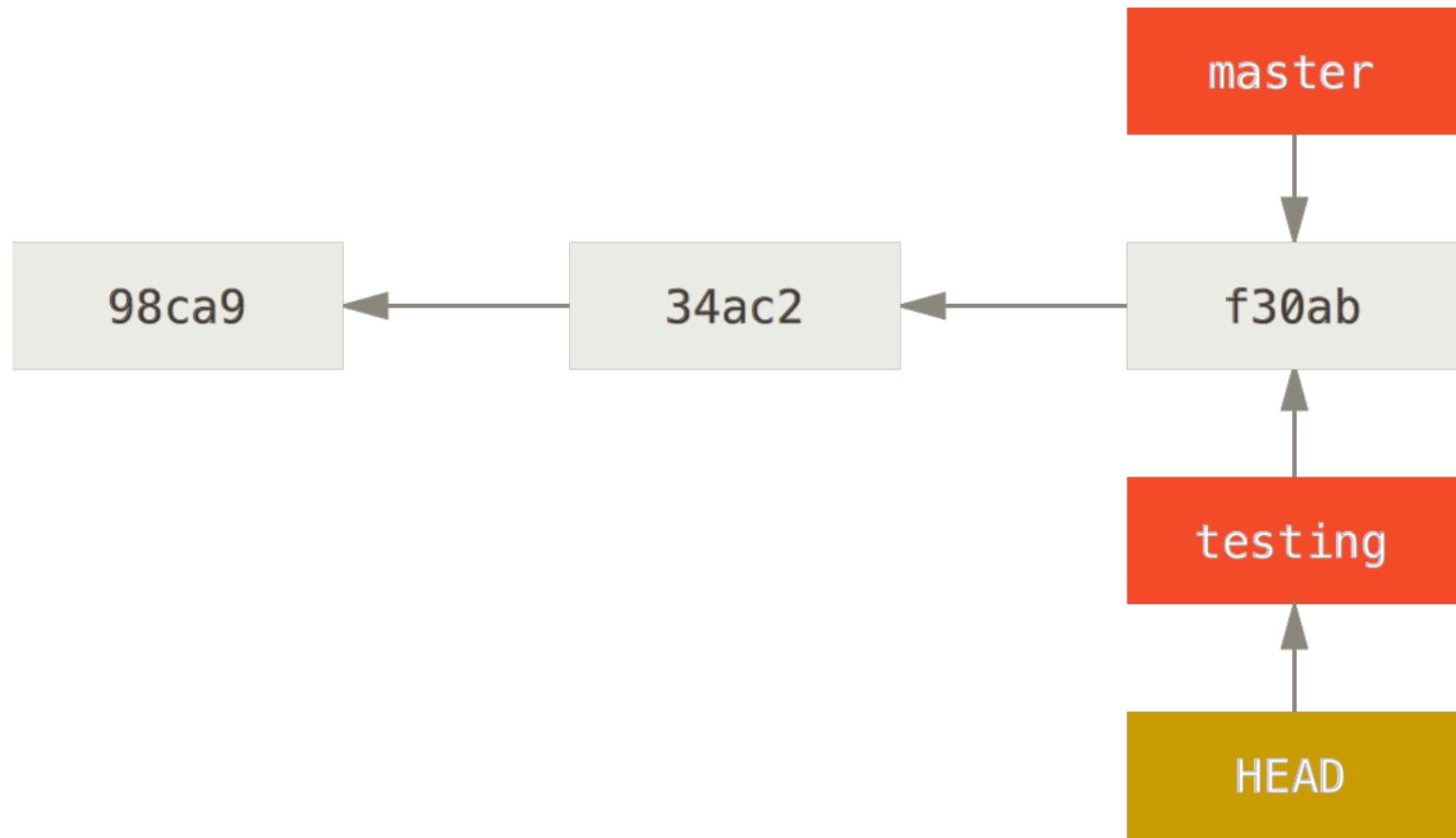
# 1) Create a branch



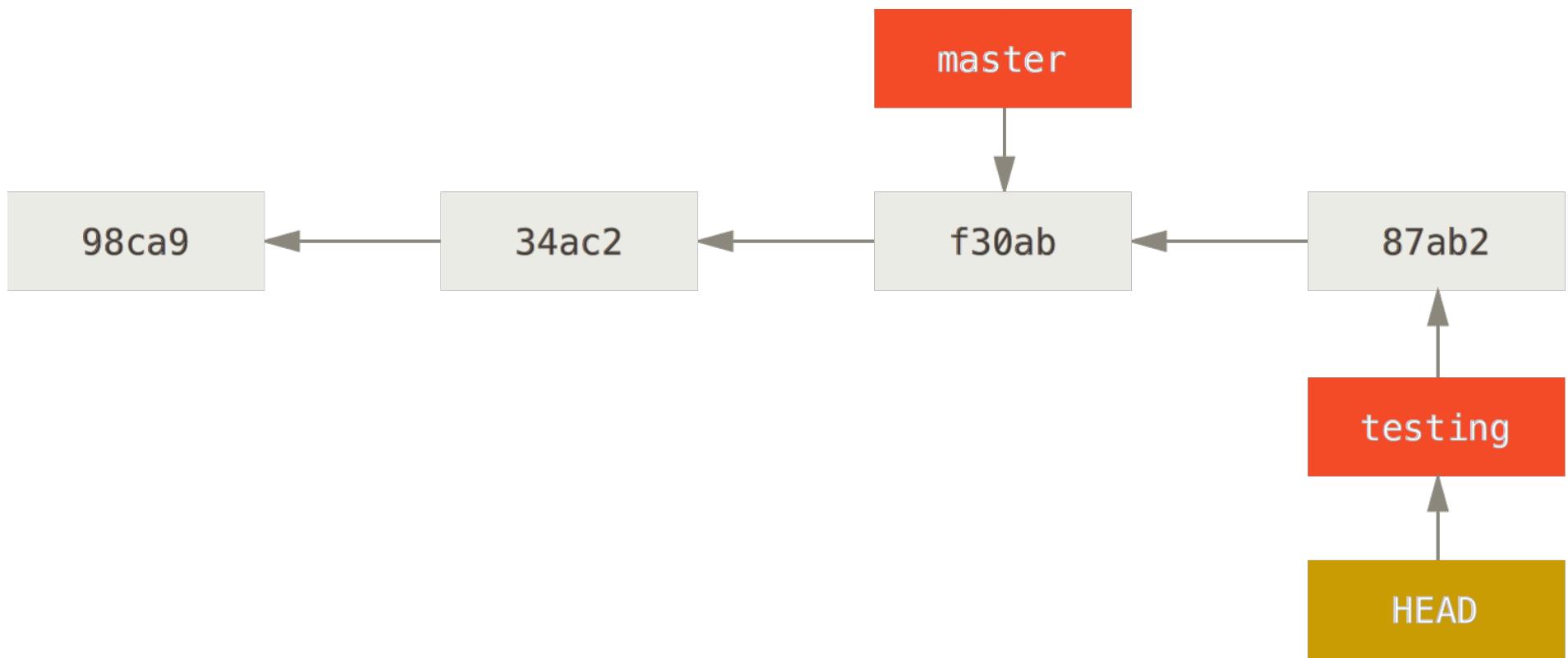
# Don't forget HEAD



## 2) Check out our new branch



### 3) Commit



# Git branch

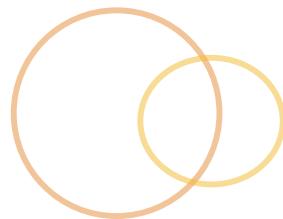
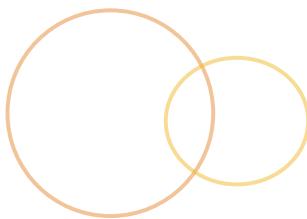


- List all your branches
  - `git branch`
  - `--list`
- Create a branch
  - `git branch <branch-name>`
- Switch to the the branch out (moves HEAD)
  - `git checkout <branch-name>`
- Create a branch and immediately check it out
  - `git checkout -b <branch-name>`

# Git checkout



- Updates your **working directory** in a *non-destructive way*
  - Undo changes in the WD
    - `git checkout -- <file>`
  - Switch branches
    - `git checkout <branch-name>`
  - And checkout a specific commit
    - `git checkout <commit-id>`
    - This puts HEAD in a “detached state”, which just means it is not pointing to a branch reference and commits can’t be made



- ◉ **HEAD** is a special pointer git uses to keep track of what branch you're currently on
  - ◉ And what the parent of the next commit will be...
- ◉ We've been working on one branch so far, “master”
  - ◉ `cat .git/HEAD`
    - ◉ “ref: refs/heads/master”
    - ◉ This is a pointer to the master branch reference
- ◉ `git checkout` updates where HEAD points to

# Log, revisited



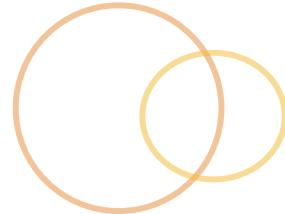
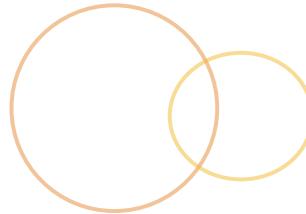
- Visualize the history through the log
  - --graph to show the tree
  - --decorate to show branch refs
  - --all to see all commits, including non-reachable
- `git log --oneline --decorate --graph --all`

# Lab: Branching



- View your branches
- Create a new branch off of master, “add-readme”, to do some work
  - This is sometimes referred to as a “topic branch”
- On the “add-readme” topic branch...
  - Create a new file, “README”, stage and commit it
- Then back on “master”...
  - Create a second branch, “add-fav-color”
- On “add-fav-color”...
  - Edit your <name>.txt file to add your favorite color to the list
  - Stage the change, commit it
- View the log --graph --oneline as you create new branches and switch between them

# Merging



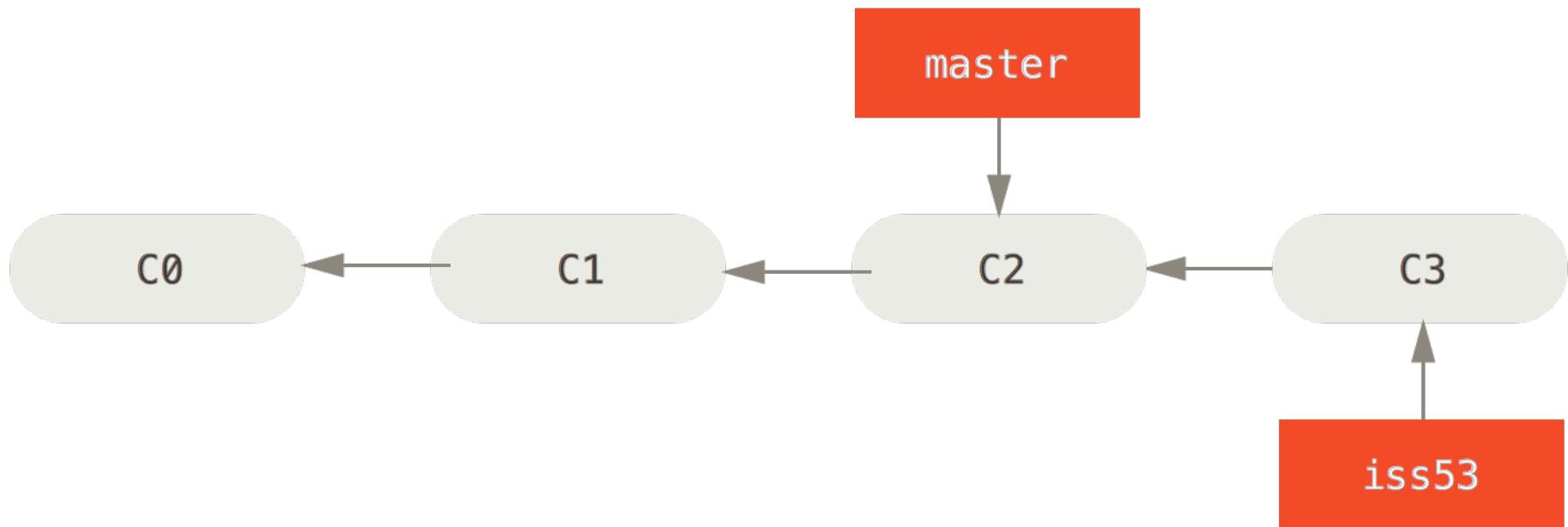
- ◉ Merging is how you get work from one branch into another
  - ◉ `git checkout <target-branch>`
  - ◉ `git merge <source-branch>`

# What is a merge?

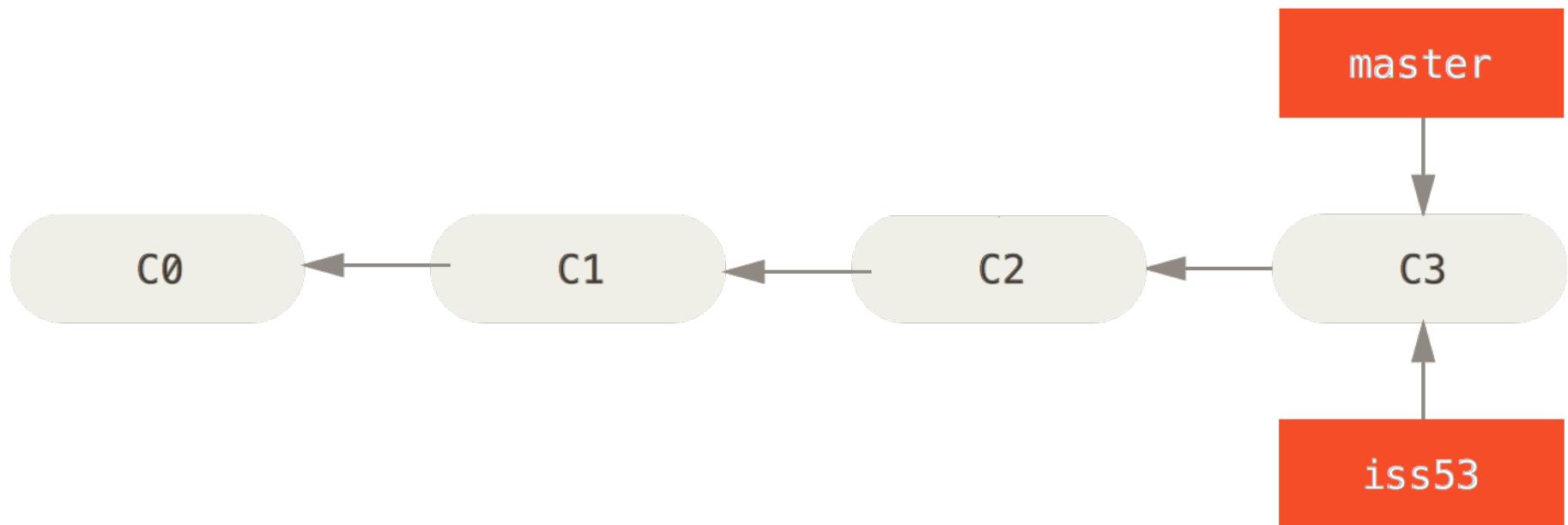


- ◉ A merge is an **action** you'll take to **combine the history** of two branches
- ◉ It can *sometimes* result in a new commit representing the merge
  - ◉ Fast-forward (no new commit)
  - ◉ 3-way merge (has a merge commit)
- ◉ It affects the **target branch** (merge into me) and not the source branch
  - ◉ `git merge <source-branch>`

# Fast-forward merge



# Fast-forward (completed)

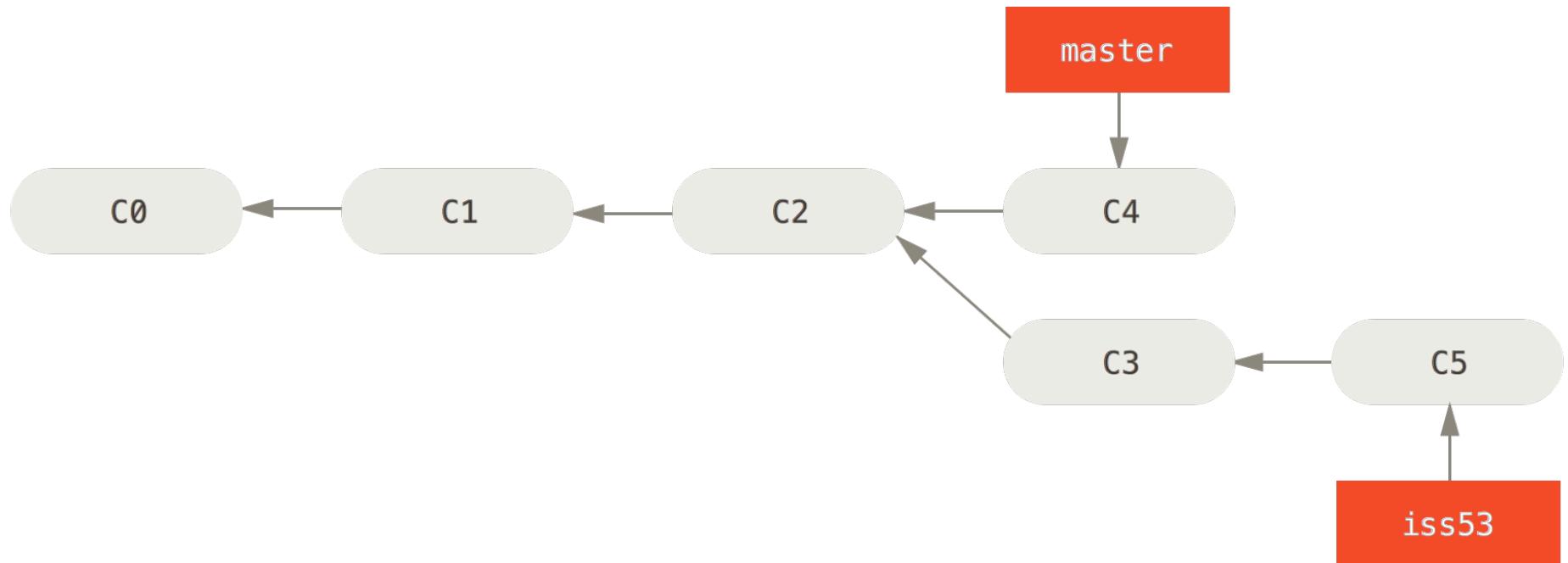


Now for the three-way

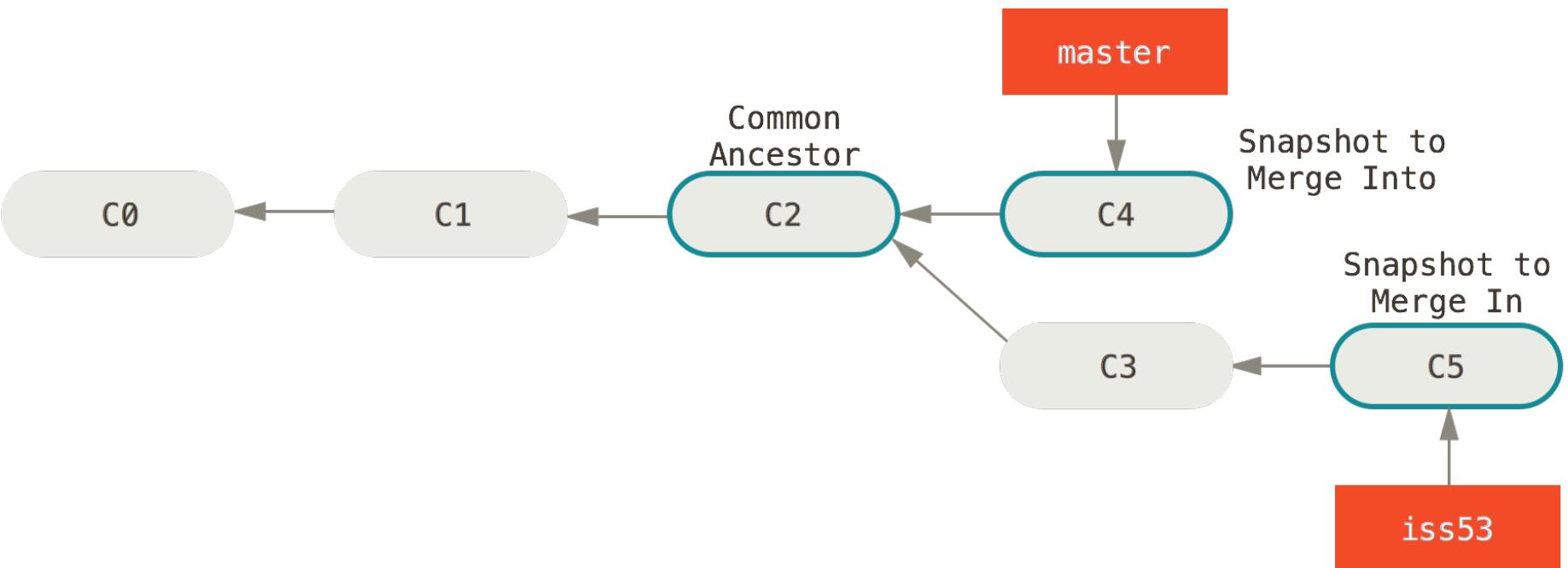


...  
...

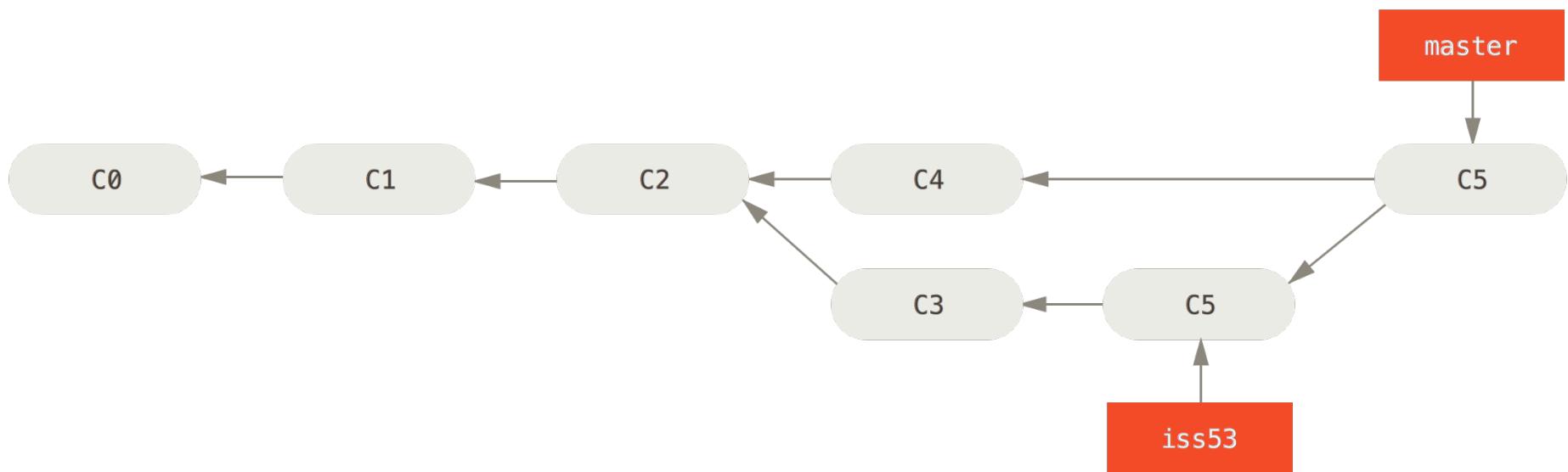
# The three-way merge



# Pre-merge



# Post-merge



# Basic merge strategies



- ◉ Fast-forward
  - ◉ When the branch ancestor is the same as the tip of the target branch
  - ◉ Results in a streamlined history
  - ◉ Can avoid this with --no-ff option
    - ◉ `git merge --no-ff <source-branch>`
- ◉ Three-way merge
  - ◉ If the commit on the branch you're merging is NOT a direct ancestor of the branch's tip you're merging into
  - ◉ Results in a new “merge” commit
  - ◉ Prompts for a commit message

# Branch Management



- You can see which branches have been merged
  - `git branch --merged`
- And see which branches are not yet merged
  - `git branch --no-merged`
- Also, don't forget to remove branches you're done with
  - `git branch -d testing`
- To remove an un-merged branch
  - `git branch -D testing`

# Basic branching workflow



- ◉ You're working on **master** and get a ticket to make a change
  - ◉ Instead of working directly on master...
- ◉ Create a new branch, based off master
- ◉ Do your work in the new branch
- ◉ Merge it back into master when ready

# Visualizing branches



- [http://pcottle.github.io/learnGitBranching/?  
NODEMO](http://pcottle.github.io/learnGitBranching/?NODEMO)

# Recap: Branching and Merging



- We saw how to view, create and delete branches with `git branch`
- Then switching between branches (and even commits) with `git checkout`
- `git merge` is how we'll integrate work from one branch into another
- And we briefly covered a branching workflow, in which new work is done in topic branches off of the stable master branch

# Lab: Merging



- Continuing from the branching lab...
- On master
  - Merge your “add-readme” branch
    - Review your log, --oneline --graph --all
    - What kind of merge did it perform?
  - View which branches are merged (--merged) and not merged (--no-merged)
  - Merge your “add-fav-color” branch
    - Review your log
    - What kind of merge did it perform?
- Then create a new branch, “add-license”
- On “add-license”
  - Add a “LICENSE” file, commit this.
  - Then edit the LICENSE file: “copyright 2015”. Add, commit.
- On “master”
  - Merge “add-license” as a non fast-forward merge with --no-ff
- Review the log, --oneline --graph --all
  - Notice the difference between the merges?
- Delete your merged branches

# Module: Merge issues



- We'll cover
  - Undoing a merge
  - Fixing conflicts from a merge

# Undoing a merge



- ◉ If you've just completed the merge and decided you didn't want to do that...
  - ◉ `git reset --hard ORIG_HEAD`
- ◉ A less destructive option
  - ◉ `git reset --merge ORIG_HEAD`
  - ◉ Resets staging but attempts to leave WD changes intact
- ◉ But if a merge has happened and it's already "out there"... you will have to revert instead
  - ◉ `git revert -m 1 <merge-commit>`
  - ◉ Creates a new commit to undo the merged branch

# Merge conflicts



- ◉ When a merge doesn't go smoothly, you've got a conflict
  - ◉ Like when the same part of a file has changed in two branches being merged together
- ◉ When a conflict happens
  - ◉ The merge is not committed
  - ◉ Git will ask you to resolve the conflict
  - ◉ You have to manually fix the conflict and commit the fixes
  - ◉ Then the conflict is resolved and the merge is considered complete

# Abort a merge during a conflict



- ◉ You can simply abort the merge in conflict
  - ◉ `git merge --abort`
- ◉ In older versions of git (still works)
  - ◉ `git reset --merge`
  - ◉ `git reset --hard HEAD`

# Resolving conflicts



- Git adds standard conflict-resolution markers to the files that have conflicts
  - <<<<<< HEAD  
original title  
=====
  - new title
  - >>>>> new-branch
- To resolve the conflict
  - Check for conflicted files with git status
  - Go fix the conflicts by hand (or a tool)
  - git add the resolved files
  - git commit to wrap it all up once all conflicts are resolved
- Or...
  - git mergetool

# Recap: Merge issues



- We saw how to undo a merge either by using `git reset` if the merge was just performed
- Or by using `git revert`, when the merge has been done some time in the past.
- We also got to see what a merge conflict is like and how to resolve it

# Lab: Resolving a conflict



- Let's create a conflict!
  - Create two branches off of master
  - In one branch, let's call it "red", edit your favorite color (in your <name>.txt file) to be "Red!"
  - In the other branch, call it "blue", edit your favorite color to be "No, Blue!"
- On "master"
  - Merge "red"
  - Then merge "blue" -- you should get a conflict
- Undo the merge! Just to get a feel for it
  - Then merge "blue" again
- Resolve the conflict

# Module: Tagging



- We'll learn about one of the last “objects” in git (along with branches and commits), **tags!**
- And how you can bookmark commits with a tag.

# Tags



- A **tag** is like a commit bookmark
  - release points
  - special commits
- Adding a tag is simple
  - `git tag <name> <optional-commit>`
- It can be annotated (signed vs lightweight)
  - `git tag -a <name> -m "Message"`
- Basic tag commands
  - List your tags
    - `git tag`
  - View a specific tag
    - `git show <tag-name>`
  - Or check it out
    - `git checkout <tag-name>`

# Lab: Tagging



- Tag your commit with a lightweight tag
  - `git tag current`
- List off your tags
  - `git tag`
- View info about your tag
  - `git show current`
- Try an annotated tag
  - `git tag -a v1.0`

# Module: Stashing, aliases, ignores



- We'll learn how to store work in progress by stashing through `git stash`
- As well as looking at two helpful repository utilities
  - We'll see how `git aliases` can help us simplify our workflow
  - And how we can tell our repository to ignore certain files

# Stashing



- Stashing will
  - Save all staged & working directory changes
  - Reset the staging area
  - Reset the working directory
- Helpful for...
  - Quickly storing work you want to revisit
  - Moving work you didn't want on branch A to branch B
- But usually a commit is fine, too

# Stashing



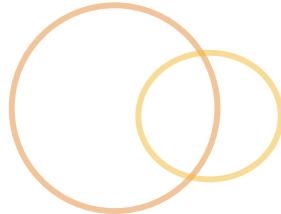
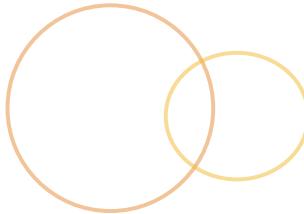
- To stash, just
  - `git stash`
- To re-apply the last stashed changes
  - `git stash pop`
- To see what is in the “stash”
  - `git stash list`
- To work from the list
  - `git stash apply <name>`
  - `git stash drop <name>`
  - `git stash pop <name>`
- Clear your stash
  - `git stash clear`

# Stashing (continued)



- You can give it a description
  - `git stash save "<description>"`
- Stash only working directory
  - `git stash --keep-index`
  - Stash only WD changes (nostaged changes)
- Stash untracked files as well
  - `git stash --include-untracked`
- Apply previously staged stuff
  - `git stash apply <name> --index`

# Aliases



- ◉ Command shortcuts
- ◉ *Lab: Let's set up a few*
  - ◉ `git config --global alias.co checkout`
  - ◉ `git config --global alias.unstage 'reset HEAD --'`
  - ◉ `git config --global alias.undo-merge 'reset --hard ORIG_HEAD'`
  - ◉ `git config --global alias.graph 'log --oneline --graph --decorate'`
- ◉ Can reference an external command with “!” prefix
  - ◉ `git config --global alias.visual '!gitk'`

# Ignoring files



- You can tell git to ignore certain files and folders
  - Set up a `.gitignore` file in the root of your project
    - `*.tmp`
    - `*.log`
    - `tmp/`
    - `/.build # only ".build" in current directory`
    - `logs/*.log`
  - Can use some basic glob patterns
- And, to stop tracking a file that is currently tracked
  - `git rm --cached <file>`
- Github has a lot of prefab `gitignores`
  - <https://github.com/github/gitignore>

# Recap: Stashing, aliases, ignores



- **Stashing** is helpful for quickly saving work in progress without having to affect history through a commit
- **Git aliases** are great for making command shortcuts
- You can easily avoid having junk/utility/config files and folders tracked in your repository by using **.gitignore**

# Lab: Stashing



- Let's imagine you need to work on the README file
- On "master"
  - Create a new branch, "readme-edits"
- On "readme-edits"
  - Add a line to the README file, "Read Me Introduction"
  - Add and commit this change
- Let's imagine you've walked away for lunch then came back and saw a ticket to update the LICENSE file...
  - Add a line to LICENSE, "Copyright 2015"
  - Add this change... ***Oh wait!***
  - It's unrelated to this branch's work effort
  - Stash it instead of committing... we should put it in its own branch
- On "master"
  - Create a new branch, "license-edits"
- On "license-edits"
  - Un-stash your edit
  - Add and commit it

# Pretty command line output



- ◉ At this point you may be lustng after my colors...
- ◉ “Sexy bash prompt”
  - ◉ [https://github.com/gf3/dotfiles/blob/v1.0.0/.bash\\_prompt](https://github.com/gf3/dotfiles/blob/v1.0.0/.bash_prompt)
  - ◉ Add this into your ~/.bash\_profile
  - ◉ There are alternatives out there...
- ◉ And my colors
  - ◉ Add these to your .gitconfig
  - ◉ <http://bit.ly/1SPPaXR>

# Let's regroup!



- ◉ You have the tools for basic git stuff on your local
- ◉ You can create a history of commits
- ◉ You can create branches and merge them
- ◉ You can view diffs
- ◉ You can deal with undoing basic changes
- ◉ You can deal with basic conflicts
- ◉ You can stash your work in progress
- ◉ Visualize?
  - ◉ <http://pcottle.github.io/learnGitBranching/?NODEMO>

# Module: Remote basics



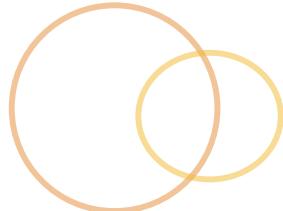
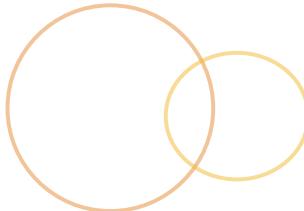
- ◉ We'll learn
  - ◉ About what a remote is and what it is for
  - ◉ How to set up a basic repository on GitHub
  - ◉ Connecting your local repository to a hosted repository through a “remote”
  - ◉ Sharing your work by pushing your branches to the hosted repository

# Collaboration!



- ◉ Working local is nice but what about collaborating with others?
- ◉ We probably want to..
  - ◉ Contribute to public projects
  - ◉ Have others contribute to our projects
  - ◉ Have a team work together on a project
  - ◉ Simply share our code
  - ◉ Keep our repository safe in case our HD dies
- ◉ First we will just get ourselves a hosted repo

# Day 2

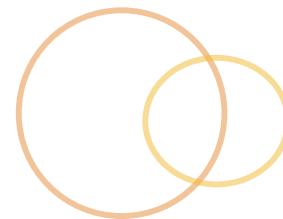
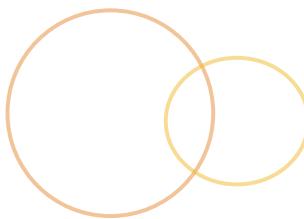


- ◉ We'll continue on collaboration in git, using GitHub and remote repositories
- ◉ Then some advanced topics like
  - ◉ Rebasing (altering history)
  - ◉ Searching the log
  - ◉ Recovering from problems in git with reflog
  - ◉ Basic hooks and integrations
  - ◉ Git plumbing (if we have time)
- ◉ Last 10 minutes for class review! :D
- ◉ Questions?

# Remotes



- **Remotes** are local references to other versions of the repository hosted elsewhere
- They are just copies of the repository!
  - Though their history may have diverged from yours by now...
- We'll be using **github** to host our repositories

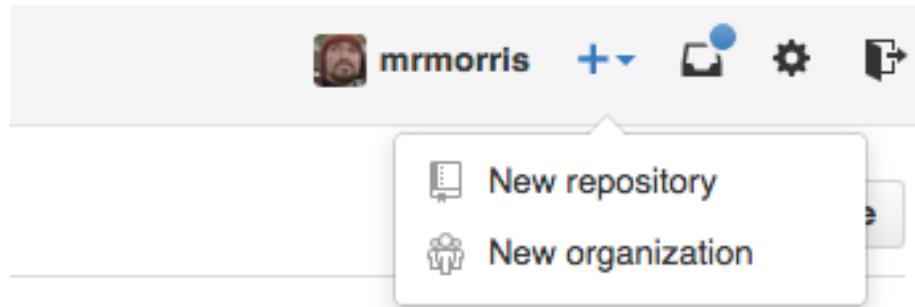


- ◉ Hosted Git repository service
  - ◉ Free for public repositories
  - ◉ Paid for private tiers
- ◉ And more project tools...
  - ◉ Web UI
  - ◉ GH Pages
  - ◉ Issues
  - ◉ Wiki
  - ◉ Integrations w/ services and hooks
- ◉ Alternatives...
  - ◉ Bitbucket
  - ◉ GitLab

# Lab: Hosting our repository



- Go to [github.com](https://github.com)
  - (got an account?)
- Add a new repository
  - Name it “about-me”
  - *don’t initialize it*
- Copy the remote (HTTPS) URL



Quick setup — if you've done this kind of thing before

[Set up in Desktop](#) or [HTTPS](#) [SSH](#) [git@github.com:mrmorris/about-me.git](https://github.com/mrmorris/about-me.git)

We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

- Add a **remote** to our local git repository
  - `git remote add origin <remote-url>`
  - `git push --set-upstream origin master`
- You will be prompted for your github credentials
- Now check out the repository on GitHub

# What just happened there?



- ◉ We created an empty repository on GitHub
- ◉ Locally, we added a remote reference to this repository and labeled it “origin”
  - ◉ `git remote add <name> <url>`
- ◉ We then pushed our local repository data to the remote repository
  - ◉ `git push <remote-name> <branch-name>`
- ◉ The `--set-upstream` (`-u`) flag set the remote as the permanent tracking remote for the branch
- ◉ Our local repo `<->` A GitHub remote

# Git protocols



- ◉ Or... how I learned to stop typing in my credentials every time I touched a remote
- ◉ HTTPS
  - ◉ Git will prompt you for your github creds
  - ◉ We can tell git to store our credentials (15 min)
    - ◉ `git config --global credential.helper cache`
  - ◉ Not working?
    - ◉ If you have two-factor authentication you'll need to use a GitHub "Personal Access Token" :p
- ◉ SSH...
  - ◉ Requires SSH keys to be set up
    - ◉ Generate your SSH key (Instructions coming)
    - ◉ Add it to your GitHub account

# How-to: SSH Keys + GitHub



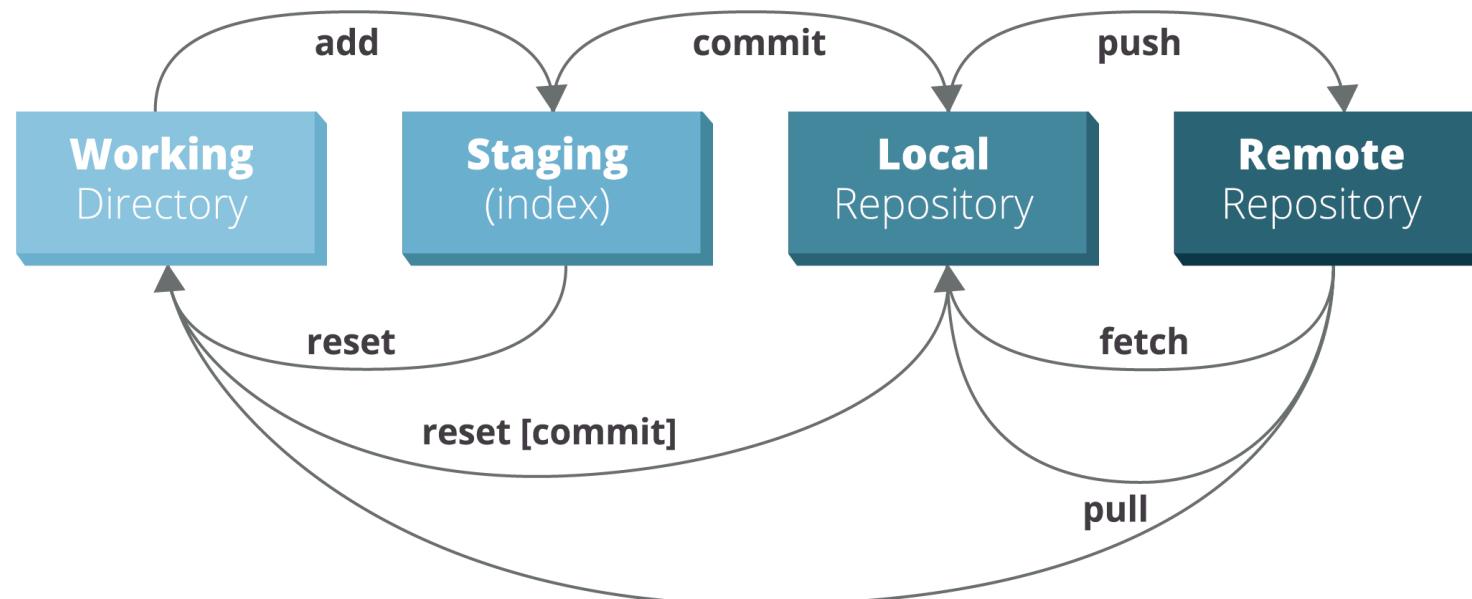
- ◉ Step-by-step
  - ◉ Check for keys you have already
    - ◉ `ls -al ~/.ssh`
    - ◉ Looking for `id_dsa.pub`, `id_ecdsa.pub`, `id_rsa.pub`
  - ◉ Not there? Generate a new one
    - ◉ `ssh-keygen -t rsa -b 4096 -C "your@email.com"`
    - ◉ [use defaults]
    - ◉ [enter a passphrase]
  - ◉ Turn on ssh-agent
    - ◉ Osx has it by default via keychain
    - ◉ Linux has it already
    - ◉ `eval "$(ssh-agent -s)" # start it in the bg`
    - ◉ `ssh-add ~/.ssh/id_rsa`
  - ◉ Add your SSH key to your GitHub account
    - ◉ Copy the `*.pub` file you generated
      - ◉ `pbcopy < ~/.ssh/id_rsa.pub`
    - ◉ Github.com > Settings > Add SSH Key > Paste > Add Key
      - ◉ <https://github.com/settings/ssh>
  - ◉ Test it!
    - ◉ `ssh -T git@github.com`
- ◉ **Or... GitHub has a very nice guide for all of this:**
  - ◉ <https://help.github.com/articles/generating-ssh-keys/>

# Working with remotes



- ◉ You'll be **fetching** remote updates to your local
  - ◉ `git fetch <remote-name>`
  - ◉ `git fetch --all`
  - ◉ `git remote update`
- ◉ You'll be **merging** updates from remotes
  - ◉ `git merge <remote-name>/<branch-name>`
  - ◉ `git pull <remote-name> <branch-name>`
    - ◉ A shortcut; fetch & merge in one
- ◉ And **pushing** your local branch updates to remote branches
  - ◉ `git push <remote-name> <branch-name>`

# How remotes fit in



# Fetch, pull and push



- **Fetch** updates remote references
  - Updated remote branch info + data is pulled down
  - Will NOT automatically merge updates.
  - **Pull** updates remote references and merges data
    - It is a fetch and then a merge
- **Push** sends your updated reference (branch) to the remote, along with all necessary data
  - Will fail when the remote branch is ahead of your local branch and a fast-forward merge is not possible
- This is how you **keep up to date** and **share changes**

# Sending tags



- ◉ To share **tags** you've added you need to push them to the remote as well
  - ◉ `git push <remote-name> <tag-name>`
  - ◉ `git push <remote-name> --tags`

# A workflow incorporating remotes



- You have a “stable” master branch
  - All new work is branched off master
- Before beginning a new branch off master, you check for updates from the remote and incorporate them into master
- You then branch off an updated master
  - You work in your branch
- When done with your work
  - You once again update master from the remote
  - Then you merge your branch to master
  - Then you push an updated master to the remote

# Remote management



- Adding remotes

  - `git remote add <name>`

- Removing remotes

  - `git remote rm <name>`

- Renaming

  - `git remote rename <orig-name> <new-name>`

- More info

  - `git remote show <name>`

- Listing

  - `git remote`

  - `git remote --verbose`

# Remote *branch* management



- View a list of branches on your remotes
  - `git branch --remote`
  - `git branch --all`
- You can check these out
  - `git checkout origin/master`
  - This will be in a detached HEAD state
- You can branch from them
  - `git checkout -b origin-master origin/master`
- You can push new remote branches
  - `git push origin origin-master`
- And delete remote branches
  - `git push origin :origin-master`

# Tracking branches



- You can tie a local branch to a remote branch so that it is “tracking” the remote
  - Fetch, Push, Pull, Merge, Rebase will automatically use the tracking branch
- Set up a local branch to track to a remote branch
  - `git branch --track <branch> <remote>/<branch>`
  - `git push --set-upstream <remote> <branch>`
  - `git branch --set-upstream-to <remote>/<branch>`
- Shortcut (*if remote/master exists*)
  - `git checkout master`
- You can view tracking branch info
  - `git branch -vv`

# Pruning remote (branches)



- ◉ There are potentially 3 versions of every remote branch
  - ◉ The actual branch on the remote repo
  - ◉ Your snapshot of that branch locally (in refs/remotes)
  - ◉ And a local branch that may be tracking the remote
- ◉ `git prune`
  - ◉ Removes references to remote branches that do not exist on the remote anymore
  - ◉ `git remote prune <remote>`
    - ◉ or
  - ◉ `git fetch --prune`

# Wait, what about collaborating?



- We'll begin to collaborate as we link team members up via our remotes and one (or many) hosted repositories

# Recap



- We just covered
  - Creating a hosted repository on GitHub
  - Linking to it from a local repo through `git remote`
  - Pushing changes on the master branch to the remote repository's master branch with `git push`
  - And we introduced the basics of a collaboration workflow in git using `git fetch` (to get updates) and `git push` (to share updates)
- Think of a remote as an **address/phone book** for another copy of the repo, which you are not hosting.  
When “up to date”:
  - you know the address (commit) of each branch it has
  - and the address (commit) of each tag it has
  - and all the data to be able to visit check those out locally

# Lab: Remote basics A



- Add a new remote with the same repo url as your about-me repository. Call it “github”
  - Check the branches that are on “github” -- you should at least see “master”. Forget to fetch?
- Create a new branch off master, call it “develop”
- Push this new branch to the “github” remote and set it up to be tracked
  - How do you confirm it is tracked?
- Check the decorated log.
  - `git log --decorate --oneline`
  - What’s different?
- Fetch updates to your “origin” repo

# Recap



- git status
  - git checkout and git reset to “undo” from WD and staging
- git log (and our alias git graph)
- git branch to see branches
  - git branch <branch> or git checkout -b <branch>
- git merge <branch-to-merge-in>
  - Merge can be fast-forward or 3-way
- git remote
  - git remote add <remote-name> <url>
  - git fetch <remote-name>
  - git merge <remote-name>/<branch-name>

# Recap



- Working locally
  - Branch off master
  - Work work work (**add, commit**, add, commit...)
  - Merge into master\*
    - \*Unless you're using pull requests or another review method!
- Staying up to date
  - Fetch from the remote
  - Merge remote branches into local branches
- Sharing
  - Fetch from the remote
  - Merge (see: stay up to date)
  - Push branches you want to share/update

# SVN way of life



- Revisions
  - Use the commit id, the latest revision is HEAD
- To restore a file from the last revision
  - `svn revert <path>`
  - `git checkout <path>`
- What change will I introduce?
  - `git diff <target-branch>..<my-branch>`
  - `git log <target-branch>..`
- What revision is a file at?
  - `git log -1 <filename>`
- Switch a file to a specific revision
  - `git checkout <commit or branch> -- <filename>`
- Did my *revision* go through to production?
  - `git branch --contains <commit-id>`
  - `git branch --remote --contains <commit-id>`
- Did my *change* go into another branch?
  - `git cherry -v <remote>/<branch>`



# Remotes and branches

**LOCAL**

master



# Remotes and branches

**LOCAL**

master

*add “origin” remote then fetch*

# Remotes and branches

LOCAL

master

origin/master

Origin (server)

master



# Remotes and branches

LOCAL

master

origin/master

Origin (server)

master

*create branch topic-1*



# Remotes and branches

LOCAL

master

topic-1

origin/master

Origin (server)

master



# Remotes and branches

LOCAL

master

topic-1

origin/master

Origin (server)

master

*git push origin topic-1*



# Remotes and branches



**LOCAL**

master

topic-1

origin/master

origin/topic-1

**Origin (server)**

master

topic-1

# Remotes and branches



**LOCAL**

master

topic-1

origin/master

origin/topic-1

**Origin (server)**

master

topic-1

*Someone else pushes a branch to origin*

# Remotes and branches



**LOCAL**

master

topic-1

origin/master

origin/topic-1

**Origin (server)**

master

topic-1

topic-20

# Remotes and branches



**LOCAL**

master

topic-1

origin/master

origin/topic-1

**Origin (server)**

master

topic-1

topic-20

*Then I fetch origin (again)*

# Remotes and branches



**LOCAL**

master

topic-1

origin/master

origin/topic-1

origin/topic-20

**Origin (server)**

master

topic-1

topic-20

*Then I fetch origin (again)*

# Remotes and branches



LOCAL

master

topic-1

origin/master

origin/topic-1

origin/topic-20

Origin (server)

master

topic-1

topic-20

*And if I want a copy of that branch locally...*

*I check it out (`git checkout origin/topic-20`)*

*Then create a branch off it (`git branch topic-20`)*

# Remotes and branches



**LOCAL**

master

topic-1

topic-20

origin/master

origin/topic-1

origin/topic-20

**Origin (server)**

master

topic-1

topic-20

# Module: GitHub Intro



- ◉ We'll get to know GitHub and learn about
  - ◉ Your account settings
  - ◉ Exploring & managing repositories
  - ◉ Introduction to issues and how they can be used in combination with pull requests

# What can we do within GitHub?



- We can do a lot directly in GH
  - Create repositories (of course)
  - Create branches (oh?)
  - Create/edit files and create commits (neat!)
  - Submit pull requests (essential)
- Along with project stuff...
  - Wiki, for documentation
  - Issues, bug/issue tracker
  - Analytics

# Let's browse GitHub



- My Account...
  - Add multiple emails and your SSH keys here
  - Manage notifications
  - Two-factor authentication?
- A repository
  - <https://github.com/twbs/bootstrap>
  - You can watch, star it
  - Fork it (clones it - you'll own the copy)
  - Review the project README, issues, pull requests, etc
- Adding a repository
  - Make it public or private
  - Initialize a few things, if you want
    - README
      - (any format, favor .md)
      - Will render as the landing page for your project
    - Pre-packaged .gitignore and license

# Let's browse: GitHub Repo Mgmt



- Set the default (master) branch
- Set repository features on/off
- Manage collaborators
- Set up service integrations
  - Integrate with external services like CI servers, bug tracking, chat systems, etc
- Set up hooks
  - Set up client-side git hooks
- Add deploy keys
  - For a single repository; typically an SSH key from your server that is deploying

# Let's Browse: GitHub Issues



- Built-in, per-project (repository) issue/bug tracker
- Ties into the pull-request workflow
- Use a GitHub flavored version of MarkDown

# GitHub MarkDown in Issues



- <https://guides.github.com/features/mastering-markdown/>
- Can set up task lists
  - [x] First task, complete
  - [ ] Second task, incomplete...
- Code style with fences

```
```javascript
var rad = function() {
```
- Reference and notify collaborators with @username mentions
- Reference issues and pull requests with #<num>
  - Issues and pull requests will not conflict
- Reference repositories and issues
  - username#<num>
  - username/repo#<num>
- Reference a commit by full sha1 hash (40 chars in full)
- Emoticons! :<meta>:
- Supports images (drag & drop)

# Recap so far



- GitHub is a place for us to **host our repository** and to **Maintain project collaboration**
- We can browse lots of information about our repo
- **Issues** are great for tracking bugs/updates (though you are not tied to them)
- **Pull requests** are essential for bringing changes into your project
- You can use pull requests to **submit patches** to other projects, too

# Lab: Join the team!



- Visit my “**about-us**” repository page
  - <https://github.com/rm-training/about-us>
- Create an **issue** requesting that I add you to the team, tagging me in the issue body
  - Ex: “Please add me to the team **@mrmorris**”
  - I will go through each issue and add you to the team
- Accept my invitation to join the team
  - You’ll get an email with the invitation
  - Otherwise check the **repository page** to see the invite
- All done?
  - Take a break
  - I'll set up a new team issue...

# Module: Pull requests



- ◉ We'll see how we can
  - ◉ create branches
  - ◉ make file edits
  - ◉ and submit pull requests
  - ◉ ...all from within GitHub

# Pull Requests



- ◉ GitHub term
- ◉ You can make a “merge request” to bring your work into another branch or project
  - ◉ “Please merge my branch into your branch”
- ◉ Works well with many different workflows
- ◉ Encourages early collaboration
- ◉ \*Will ALWAYS do a non-fast-forward merge

# I'll create a PR to resolve an issue



- <https://github.com/rm-training/about-us/issues>
- For me to do:
  - Create the branch
  - Add the index.html file
  - Push it to the remote (we're working directly in gh)
  - Create a pull request from the branch
    - Mention the issue #
  - Get it reviewed
  - Merge it
  - Delete the branch

# Lab: Resolving an issue with a PRO



- ◉ Note: *This is all to be done within GitHub (no local repository work yet)*
- ◉ Resolve the issue by creating a pull request with a new branch you create
  - ◉ Create a new branch, off of master
  - ◉ Add a new file in this branch for your profile (<your-name>.html)
    - ◉ Edit the file with the information required
  - ◉ Commit to your branch
  - ◉ Create a pull request, asking to merge your branch into master
- ◉ Relate your pull request to the issue
  - ◉ Reference the Issue number in the Pull Request body to auto-resolve the issue
    - ◉ Ex: “Resolves #5231”
  - ◉ Alternatively just reference your pull request number in the issue comments
- ◉ We'll go through and review (and merge) our pull requests together

# Recap so far



- So far we've been working as a team within a single repository, entirely within GitHub
- We're using **issues** as our ticket tracker
- And creating **new branches for new work**
- We then created **pull requests** to initiate a merge review to get our changes back into the main repository branch
- We kept our review discussion around our issue/pull-request in GitHub
- Ultimately, upon **approval**, our work was **merged**

# Module: Repository workflows



- We'll get a feel for the different ways you can organize your team's workflow
- We'll use git clone to copy down our hosted repo
- And we'll look at how you can keep up to date locally in your clone
- Along with your fork

# Remotes and branches => workflow



- ◉ How you structure your remotes and branches determine your workflow
- ◉ A project can be public or private
- ◉ Remote approaches
  - ◉ Single remote, all contributing
  - ◉ Many remotes (through forking), contributing up to single repo
- ◉ Branching approaches
  - ◉ master or main or production
  - ◉ master (stable) + develop (new work)
  - ◉ master + develop + hotfixes
  - ◉ master + integration + staging

# GitHub Flow



- GitHub supports a particular workflow
  - master is always stable
  - Branch any new work off master
  - Submit a pull request (early)
  - Discuss work in the pull request
  - Merge pull request to master when ready
- <https://guides.github.com/introduction/flow/>

# Taking it local



- ◉ It's all well and good to be able to do a lot of basic work directly in GitHub, but what about getting a local copy of that repository we've been working in?
- ◉ We can do that!

# Cloning



- Copy an existing repository to a new local
  - `git clone <remote-url> <optional-dir>`
- `git clone` will
  - Initialize a local git repo in the directory
  - Pull all data and remote branches down
  - Set up an initial remote, called “origin”
  - Set up the initial tracking branch for “master”
- If you are going to work on a repository that already exists then you will likely start by cloning
- *Forking* in GitHub is like cloning (but you own it)

# A word on authentication



- ◉ The repository URL protocol determines the authentication you'll use
  - ◉ HTTPS
    - ◉ Prompts for your username/pass
    - ◉ Available everyone (even behind proxies)
    - ◉ *If you use two-factor authentication* you'll have to set up a token to use as your password instead
  - ◉ SSH
    - ◉ Set up an SSH keypair and associate it with your GitHub account

# Staying up to date



- If we have a remote, we can stay up to date with
  - `git fetch <remote>`
- And bring updates from remote branches into our local branches
  - `git checkout <branch>`
  - `git merge <remote>/<branch>`
- We can push our own updates to the remote
  - `git push <remote> <branch>`

# Lab: Cloning

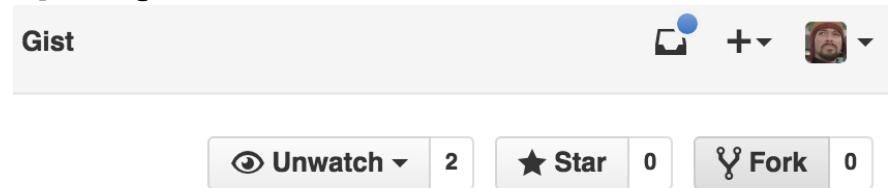


- Clone my repository!
  - **Clone** the repository locally
  - Get the clone url from my repo page:
    - <https://github.com/rm-training/about-us>
- Create a new **topic branch** to do some work
  - The task: Edit your profile page to add a paragraph about your pet preference (cat, dog, nothing?) and why.
  - Share your branch by **pushing** it to the remote repository
  - On GitHub, open a **pull request** to have your new branch merged into the master branch
- We'll review & merge together
- After we have merged them...
  - Update your local master branch using git fetch, then git merge
- What would happen if you didn't keep up to date and decided to branch off master?

# GitHub Forks



- You can Fork any public project in GitHub
  - It's a copy you own
- Why fork?
  - To submit work to a public project
  - Organizational safety net for larger teams
  - To enforce a subset of project owner(s)
- Forking workflow
  - You fork a project (**the upstream**)
  - You branch off of (and push branches to) your fork
  - You submit pull requests into the main, **upstream**, project
  - Project owner can review and merge



# Lab: Forking



- Now we're going to work within our own forks...
  - Each of us, as developers, will have our own fork and will push our work to that fork then submit Pull Requests to the main repository.
- Fork my repository via GitHub
  - <https://github.com/rm-training/about-us>
- Clone **your fork** to your local via git
  - Clone your fork into a new directory ("about-us-fork")?
    - `git clone <your-fork-url-from-github> ./`
  - Look at the branches that are set up and the remotes
- Create a new branch off of master to make an edit
  - Edit the index.html file to add a link to your profile page
    - `<a href="your-name.html">Your Name</a>`
  - Push your new branch to your fork
  - From GitHub, submit a pull request into my repository
- We will review and merge
- How do we keep our local (and our fork) up to date with the main repository?
- How do we deal with these... *conflicts*?

# Staying up to date (pt 2)



- When working on one remote, just fetch and merge from the “origin”
- When working on a **fork**, you will want to fetch updates from the “**upstream**”
  - Add an “**upstream**” remote (the main repository)
    - `git remote add upstream <main-repo-url>`
  - Fetch and merge updates from there
    - `git fetch upstream`
    - `git merge upstream/<branch>`
  - Push them to your fork as needed
    - `git push origin master`

# Lab: Staying up to date w/ forks



- Add a new “upstream” remote for the main repository
- **Fetch** the updates from upstream
- To keep master up to date:
  - Merge updates from upstream master to your local master
  - Push the updated local master to your fork’s master
- Then to deal with the conflicts:
  - Merge the updates from master into our topic branches
  - Re-push our topic branches to GitHub
  - Merge it from GitHub

# GitHub... wrap up



- There are GitHub UI's for working with git locally
- Integrations to take GitHub further
- And of course, a pretty meaty API

# Module: Collaboration Workflows



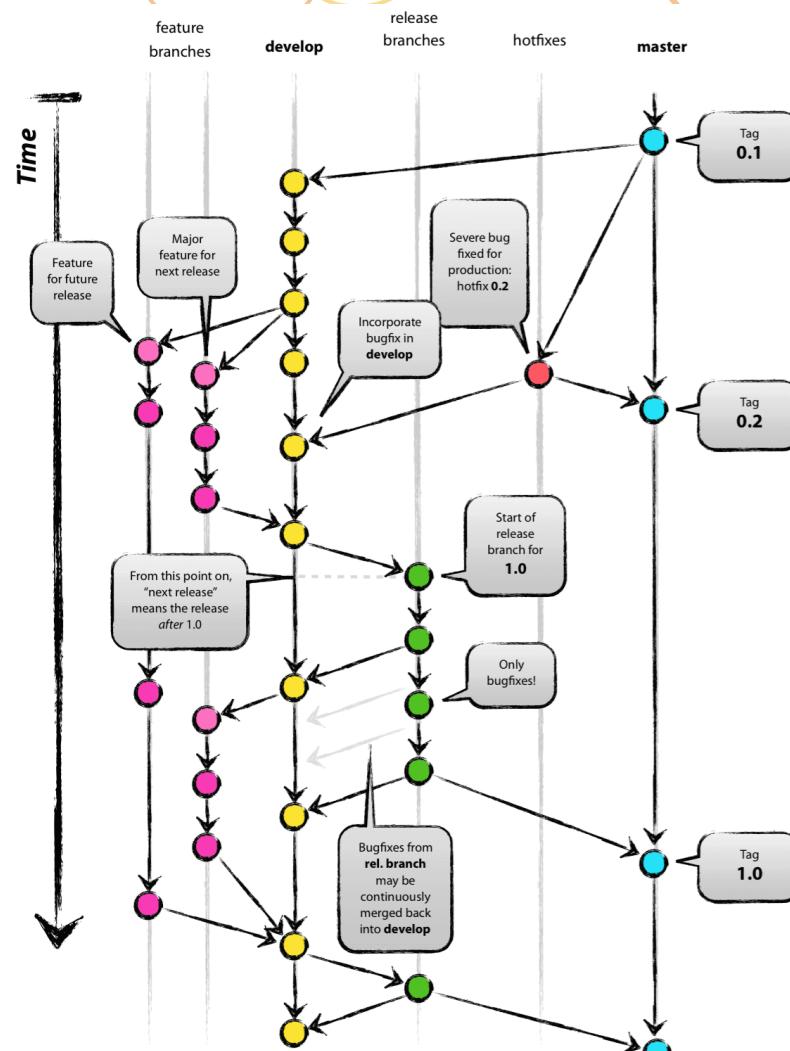
- We'll put together what we know and review some collaboration workflows
- As well as branch strategies

# Workflows, Teams, Branches



- Workflow styles
  - Centralized workflow
  - Integration-manager workflow
  - Dictator and lieutenants workflow
- Collaborator roles
  - Private, small team
  - Private, managed team
  - Public, forked project
- Branching strategy
  - Master as only branch
  - Stable (Master) and Unstable
  - Release, Topic, Hotfix branches

# “Git Flow”



- Source: <http://nvie.com/posts/a-successful-git-branching-model/>

# Tag Strategies



- ◉ Tag versions
  - ◉ V1.0
  - ◉ V1.0.1 (hotfixes)
- ◉ Keep versions in “active development” in a branch
- ◉ Otherwise, just use the tag

# Recap: Workflows

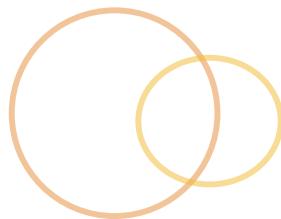
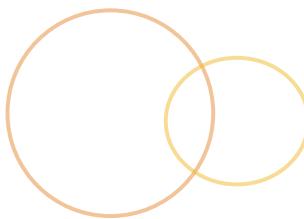


- ◉ How you structure your **team**, your **repositories** and your **branches** is up to you (and your team)
  - ◉ Pick a strategy and have everyone stick to it
- ◉ One main repo? Many forks?
- ◉ One primary branch, or several tracks of development?
- ◉ Public or private?

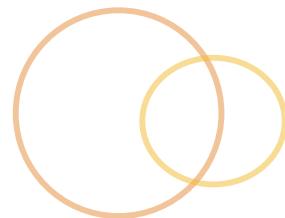
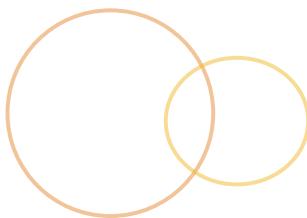
# Module: Git Tools



- We'll learn about the graphical tools git makes available to stage files and view your log
- As well as how you can go further with merging and diffing by using external visual tools



- GitK for a graphical display of the log and search
  - `gitk`
- Accepts most params that “git log” accepts
  - `gitk --all --decorate`
- Can also see what changes are in your staging and working directory



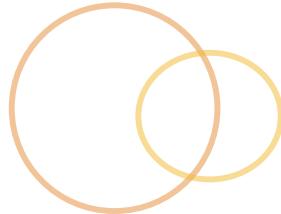
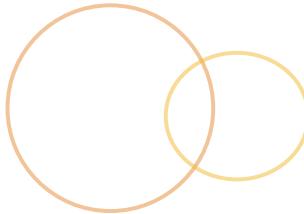
- ◉ A full UI for most Git functionality, but really boils down to...
- ◉ A tool for crafting commits
  - ◉ `git gui`
- ◉ You can stage and commit
  - ◉ Including patching (partials)

# Mergetool



- Perform merges using a visual tool by setting up “mergetool”
  - `git config --global merge.tool <toolname>`
- Then during a merge conflict
  - `git mergetool`
  - `git mergetool -t <toolname>`
- To see some options you may have already
  - `git mergetool --tool-help`
- Additional config
  - `git config --global mergetool.<toolname>.cmd "bla"`
  - `git config --global mergetool.trustExitCode false`
    - Whether the exit code indicates a successful merge
  - `git config --global mergetool.keepBackup false`

# Difftool



- You can use a similar visual tool for viewing diffs, too
  - `git config --global diff.tool <toolname>`
- Then just use it
  - `git difftool`
  - `git difftool -t <toolname>`
- To see some options you may have already
  - `git difftool --tool-help`
- More configs
  - `git config --global difftool.prompt false`

# Example: Kaleidoscope



- Grab it
  - <http://www.kaleidoscopeapp.com/>
- Can automatically tie it into git, or manually enter:

```
[difftool "Kaleidoscope"]
cmd = ksdiff --partial-changeset --relative-path
\"$MERGED\" -- \"\$LOCAL\" \"\$REMOTE\"

[mergetool "Kaleidoscope"]
cmd = ksdiff --merge --output \"\$MERGED\" --base
\"\$BASE\" -- \"\$LOCAL\" --snapshot \"\$REMOTE\" -
snapshot
trustexitcode = true
```
- Usage:
  - git mergetool -y -t Kaleidoscope
  - git difftool -y -t Kaleidoscope

# Recap: Git tools



- ◉ **GitK** is extremely useful for viewing and searching through your commit history
- ◉ Git **GUI** is *sort* of helpful, at least for visually patching files to staging
- ◉ Avoid painful merge/diff issues in the command line, set up a **mergetool** and **difftool** of choice.

# Lab: Set up our merge/diff tools



- Check what you have available
  - `git mergetool --tool-help`
- Then pick one and configure it
  - `git config --global merge.tool <toolname>`
  - `git config --global diff.tool <toolname>`
- Then test it by creating a conflict
  - Create a new repository (locally)
  - Add a new file, commit it (your first commit)
  - Create two branches off master
  - In both, edit the same line in the same file
  - Now: `difftool`
  - In both: add, commit
  - Merge one
  - Then attempt to merge the other
  - Now: `mergetool`

# Module: Altering history



- We'll look at an alternative to merging, **rebase**, which allows us to integrate changes with a modified history
- Through **rebase** we can avoid merge-commits as well as squash commits down into a single commit
- Finally, we'll use **cherry-pick** to grab a single commit from a branch and integrate it onto another

# Common merge types



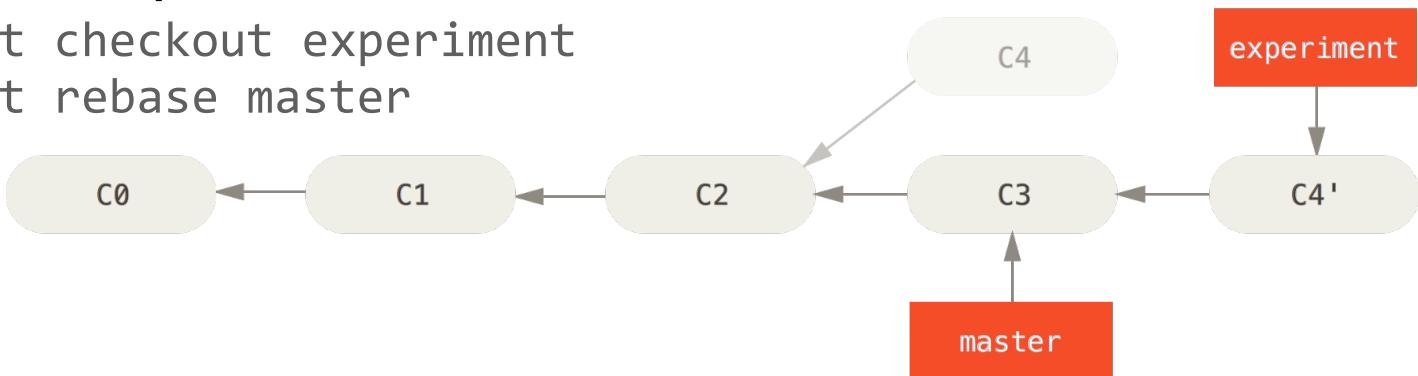
- ◉ When merging, git will attempt to use the most appropriate method of merge
- ◉ Fast-forward
  - ◉ If both branches share the **same parent**
  - ◉ Is like a non-destructive **rebase**
  - ◉ Can avoid: `git merge --no-ff <branch>`
- ◉ Recursive (3-way)
  - ◉ If the branches have diverged
  - ◉ Results in a new merge commit

# Rebase as a third option



- We can also “rebase” changes from one branch to another
- This takes the unique commits from the branch you are on and re-applies them as though the target branch is the new parent
- For example:

- `git checkout experiment`  
`git rebase master`



- Vis:
  - <http://pcottle.github.io/learnGitBranching/?NODEMO>

# Rebasing



- `git rebase <new-base>`
  - Sets **current branch** to point to **new base as its tip** then **re-applies** the unique commits on top of that
- Like a transplant, or grafting
  - Snip off the commits in current branch,
  - Re-apply one at a time to target branch
  - Current branch ref is set as the new tip
- It affects the current branch only
- It re-applies only the unique diffs on current
  - `git log <new-base>..`
- It is history-altering!

# Rebase to keep up to date



- ◉ When pulling changes from another branch (incl. remotes)
  - ◉ `git fetch origin`
  - ◉ `git rebase origin/master`
- ◉ But remember, this regenerates every *unique* commit in your current branch
  - ◉ Pushing to your remote will fail because it can't fast forward (indeed, the original parent has changed)
  - ◉ So you have to force it
    - ◉ `git push origin <branch> --force`
  - ◉ **But! Don't rebase (or force push) shared commits**

# Rebase to squash



- Using rebase “interactive” you can **squash** and modify commits
  - `git rebase -i origin/master`
  - Give you the option to “s” squash or “e” edit commits as it replays them
- Merge also has a squash option
  - `git merge <branch> --squash`  
#still need to commit  
`git commit`
  - Will squash all commits on other branch into one and stage it on the current branch

# Rebase conflicts



- ◉ Sometimes there's a conflict while it rebases...
- ◉ Resolve the conflict and
  - ◉ `git rebase --continue`
- ◉ Otherwise just abort the whole operation
  - ◉ `git rebase --abort`

# Rebase vs Merge



- Merge
  - Safe, easy, non-destructive
  - Easier to see branching activity
  - Easier to revert a merge (commit)
  - Noisy, lots of extra merge commits
  - Hard to follow the history
- Rebase
  - Clean, linear history
  - Can clean up lots of in-progress commits
  - Easier to navigate w/ log, bisect and gitk
  - Flexibility, can squash and edit commits
  - Unsafe, destructive
  - No traceability (ex: when was this feature merged?)
- So when should I use them?
  - **Merge** (--no-ff) completed work into master
  - **Rebase** to fetch updates into topic branches and into local master
  - Ultimately it is up to you and your team

# Cherry-pick



- You can select one commit at a time to rebase it into your current branch
  - `git cherry-pick r32fs32`
  - “Apply X commit as a new commit on my current branch”
- This does alter history
- Like rebase, you can continue/abort
  - `git cherry-pick --continue`
  - `git cherry-pick --abort`

# Workflow: Merge/History Strategies



- ◉ Always merge (never rebase)
- ◉ Always rebase (never merge)
- ◉ A mix
  - ◉ Merge branches to master (never fast-forward)
  - ◉ Rebase to update branches from the upstream
- ◉ To squash or not to squash?

# Recap: Altering History



- **Rebase** is a powerful (and dangerous) alternative to merging
- Keep a clean history by avoiding merge commits, and **squashing** messy work
- But **merge** still has a place to maintain a meaningful history
- **Cherry-picking** is good at grabbing one (or a couple) commits from one branch into another
- But in all history-altering operations be careful not to affect shared commits

# Lab: Changing history



- Clone my repository:
  - <https://github.com/rm-training/history-changer>
- Check the graph ( --all)
- 1) Using rebase to bring in updates
  - Update the topic-behind-1 branch with changes from master by using rebase
  - Note: the branches are all remote... you'll need to branch them locally to work on them locally
- 2) Using rebase to squash commits
  - checkout the messy-branch and view the log
  - Use rebase -i to squash it into one commit
  - Give it a more meaningful commit message
- 3) Cherry pick a commit
  - checkout diamond branch
  - Use git log to find the commit with the “super important patch”
  - Create a new branch, diamond-only, off master and use git cherry-pick to bring that *important* “super important patch” commit into the new branch
- 4) Extra time? Try to rebase topic-behind-error

# Module: More with merge



- We'll check out a few more advanced options when merging
- As well as some additional approaches for dealing with conflicts

# Merging options



- Merge and ignore whitespace
  - `git merge -Xignore-all-space <branch>`
    - Ignore all whitespace when comparing lines
  - `git merge -Xignore-space-change <branch>`
    - Ignore changes in amount of whitespace
- Avoid auto-committing the merge
  - `git merge --no-commit <branch>`

# Merge preference



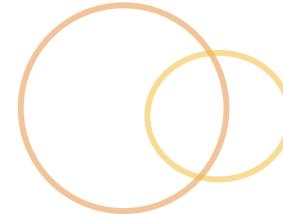
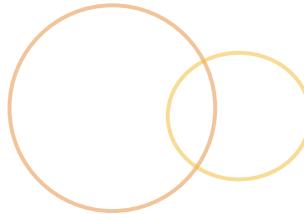
- ◉ When merging, we can tell git to prefer a specific branch
- ◉ Expecting a conflict?
  - ◉ `git merge -Xours <branch>`
    - ◉ Prefer the current branch for conflicts
  - ◉ `git merge -Xtheirs <branch>`
    - ◉ Prefer the branch you are merging in
- ◉ Or just opt for one entirely
  - ◉ `git merge -s ours <branch>`
    - ◉ Defer to current branch for *any* diffs
  - ◉ There used to be a “*theirs*” but it has been abandoned due to uselessness
  - ◉ Still results in a (dummy) merge commit

# When in conflict



- View a “merge” diff
  - `git diff --merge`
- Reset a *conflicted* file to the *conflicted* state
  - In case you screwed things up while resolving it
  - `git checkout -m <filename>`
- Opt for one version or another
  - Skips having to manually resolve the conflict
  - `git checkout --ours <filename>`
    - Defer to *current* branch version
  - `git checkout --theirs <filename>`
    - Defer to *merged* branch version

# ReReRe



- Reuse recorded resolution
- Tell git to track merge resolutions for re-use
  - `git config --global rerere.enabled true`
- It will remember how you resolved identical conflicted hunks and use that solution in the future

# Recap: More merging



- ◉ We saw a few merge options like
  - ◉ Ignoring whitespace
  - ◉ Conflict hints with `-Xours` and `-Xtheirs`
  - ◉ And having merge use one branch over another entirely with the (`-s`) `ours` strategy
- ◉ When in conflict you can pick between versions with checkout `--ours` and `--theirs`
- ◉ ReReRe is a neat helper to keep track of conflict resolutions
- ◉ Many other options and strategies

# Module: Comparing



- We'll check out the “..” and “...” operators for comparing commits and branches
- This is good for being able to see a collective diff output, or see the set of commits being introduced by a branch
- Let's use:
  - <https://github.com/rm-training/comparing>

# Comparing branches with log



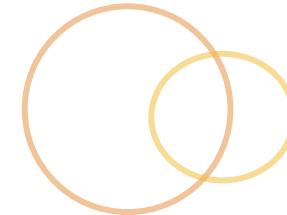
- ◉ What is in <B> that is not in <A>
  - ◉ “*Commits reachable by B but not A*”
  - ◉ `git log <A>..<B>`
  - ◉ `git log ..<B>`
- ◉ What is different in both <A> and <B> but not shared?
  - ◉ “*Commits reachable by either, but not both*”
  - ◉ `git log <A>...<B>`
  - ◉ To see which branch a commit comes from in the output
    - ◉ `git log <A>...<B> --left-right`

# Comparing branches with diff



- See the diffs between two branches
  - “*All differences between B and A*”
  - `git diff <A>..<B>`
- See the diff that `<B>` introduces to `<A>`
  - “*Difference between B and the common ancestor that B has with A*”
  - `git diff <A>...<B>`

# Recap: Comparing



- Easy enough!
- We can use log (with ..) to determine what commits are being introduced by a branch
- And diff (with ...) to see what changes a branch is introducing

# Lab: Comparing



- Clone my repo
  - <https://github.com/rm-training/comparing>
- While on master, view the log graph
- How can you check...
  - Which commits the “add-introduction” branch would introduce to master?
  - The difference in all commits between “add-introduction” and “add-profile”
    - Does it matter which order you check in?
  - The difference “add-profile” would introduce to “master”?

# Module: Debugging through Git



- We'll see how we can use git log to get lots of information about our repository history
- And how we can annotate a file with git blame to see who changed what.
- We'll also try out git bisect for finding where a bad change was introduced

# Debugging with log



- ◉ Use git log to find a commit you need
- ◉ Lots of options...
  - ◉ Just one liners please
    - ◉ --oneline
  - ◉ View the diff
    - ◉ -p
  - ◉ Abbreviated commit stats
    - ◉ --stat
  - ◉ Custom format
    - ◉ --pretty
    - ◉ `git log --pretty=format:"%an committed %h %ar: %s"`
  - ◉ Only show file names
    - ◉ --name-only
  - ◉ Include references
    - ◉ --decorate
  - ◉ Shortened commit id
    - ◉ --abbrev-commit
- ◉ <http://git-scm.com/docs/git-log>

# Limiting the Log



- Number of commits
  - -<n>
- By date
  - --since 2.days
  - --until "2012-01-01"
  - --before "yesterday"
  - --after "yesterday"
- Avoid merge commits
  - --no-merges
- Give a range
  - git log <since>..<until>
  - When given branches, it outputs the difference from <until> not in <since>

# Searching the Log



- Search by author
  - git log --author "Ryan"
- Search commit messages
  - git log --grep "Added"
- Search content (commits that add or remove a line matching the string) (use -G for regex version)
  - git log -S "myString"
  - git log -G "[0-9]+Rad"
- A path or file
  - git log -- file1 file2 path1 etc

# Git Bisect



- Binary search through commits and changes
- How to use it
  1. Start it up
    - `git bisect`
  2. Then tell bisect the known bad commit and last good commit
    - `git bisect bad <commit or defaults to current>`
    - `git bisect good <commit or tag when it was good>`
  3. It determines mid-point and allows you to re-verify
  4. Tell it if each commit is good or bad
    - `git bisect good`
    - `git bisect bad`
  5. Finally, it outputs the hash of the *first bad commit*
    - `git bisect reset`

# Git blame



- Annotates file with commit information for each line
  - `git blame <filename>`
- Limit the output
  - `git blame -L 12,22 <filename>`
- Track code movement
  - `git blame -C <filename>`

# Recap: Debugging in git



- The git log is a very powerful tool for parsing the repository history, and also debugging
- When in doubt, git bisect can help you search for where a change was introduced
- And finally, we saw how we can view who changed lines in a file with git blame

# Lab: Bisect and blame

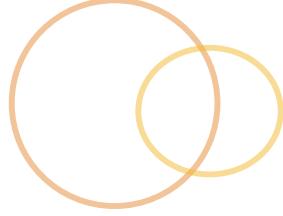
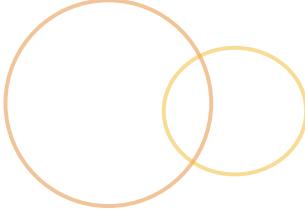


- Clone my repository
  - <https://github.com/rm-training/long-history>
- Using git bisect, determine in which commit the “*Long walks on the beach*” line was added to index.html
- Using git blame, determine who originally added that line.
- All done?
  - Experiment with git log
  - Search the log for the line using “git log -S”

# Module: Fixing issues with git



- We'll learn how we can search the history of what we've done in git using the reflog
- And come away with a better understanding of git's HEAD pointer
- We'll learn about navigating through your repository's commits
- Finally, we'll see how we can undo changes or make things right using git reset and git revert



**HEAD**

- Head is a symbolic reference to the branch you're on (or commit)
  - Actually a pointer to another reference
- For example...
  - `git checkout master`
  - `cat .git/HEAD`
    - Outputs: Ref: refs/heads/master
- HEAD is used
  - For new commits
    - A commit is given a parent id from HEAD
    - The branch ref is updated to point to the new commit
    - HEAD still points to the branch
  - When you checkout a commit or branch
    - HEAD is updated to point to that commit or branch

# The reflog



- Git keeps a log of where **HEAD** and **branch refs** have been over the past few months
  - `git reflog`
- You can use these references
  - `git show HEAD@{3}`
  - “Where HEAD was three moves ago”
- Branches also have ref logs
  - `git reflog master`
- And you can reference by date
  - `git show master@{yesterday}`
  - `git show master@{one.week.ago}`
- View by date
  - `git reflog --date=iso`

# Ancestral references



- For any commit, branch or tag, you can trace up through its heritage
- Direct parent
  - `git show HEAD^`
  - If it was a merge commit it has two parents, show second
    - `git show <merge_commit_hash>^2`
- Parent
  - `git show HEAD~`
    - 1<sup>st</sup> parent of HEAD
    - same as `HEAD^`
  - `git show HEAD~2`
    - 1<sup>st</sup> parent of the 1<sup>st</sup> parent of HEAD
    - Same as `HEAD^^`
  - `git show HEAD~n`
    - nth parent of HEAD

# Git reset



- git reset manipulates *the three trees* of git through three basic operations:
  - **Moving HEAD (soft)**
    - Changes the commit the current branch ref points to (via HEAD)
      - `git reset --soft HEAD~`
      - ie: undo the last commit without losing staging changes
    - **Moving HEAD and updating staging (mixed)**
      - The above *and* updates the index to match that commit
        - `git reset --mixed HEAD~`
    - **Moving HEAD, update staging and update the WD (hard)**
      - All the above *and* updates the working directory to match as well
        - `git reset --hard HEAD~`
      - This is destructive in that it will wipe out changes in your WD

# Git reset (continued)



- If you use a filename/path, however...
  - `git reset <filename>`
- It will behave like a `--mixed` reset
  - It can't move HEAD to a file, so it skips `--soft`
- Will put whatever `<filename>` looks like in the HEAD commit and put that in the Index
  - ie: unstage the file changes
- You can specify the commit version
  - `git reset <commit> <filename>`

# Checkout (in terms of reset)



- `git checkout` will change *what* reference HEAD points to
  - Unlike `reset`, it does not affect the reference itself
  - Updates index and WD to match, but it will not overwrite changes you have made
- `git checkout <filename>`
  - Will update the WD and index file from what HEAD points to

# Commit recovery (and undoing)



- Using log, reflog, reset and checkout, we can fix a lot of problems we may find ourselves in.
- Find a lost commit
- Find a lost branch
- Undo a merge
- Undo a rebase

# Git revert



- Apply an inverse of changes from a commit or set of commits
- If I want to undo commit ab3r230
  - `git revert ab3r230`
    - This will create a new commit applying changes that effectively reverse the changes in ab3r230
  - `git revert ab3r230 --no-commit`
    - Will create the revert changes but will not commit them
- Once reverted, I can't re-merge that commit
  - I can, however, revert a revert
- Good for undoing **public** commits

# Reverting a merge



- To revert a merged branch just revert the “merge commit”
  - `git revert 352e23 -m 1`
    - --mainline
    - You tell it (1 or 2) which branch is the mainline to revert
    - 1 is right-most, 2 is next to the left

```
$ git log --oneline --graph
* b83f729 Added third file
*   f757139 Merge branch 'newbranch'
|\ 
| * e66afed New file in branch
|/
* 8f80b81 New file added
* 1b08cb2 Added readme
```

# Fixing some common issues



- ◉ Branch won't merge cleanly
  - ◉ Rebase your branch
  - ◉ Or merge the target into your branch and resolve
- ◉ Accidentally rebased
  - ◉ Reset to the commit before the rebase
    - ◉ Use the ORIG\_HEAD
      - ◉ `git reset --hard ORIG_HEAD`
      - ◉ Or, use reflog to find the original HEAD
        - ◉ `git reflog <branch> -10`
  - ◉ Broke the master
    - ◉ Just check out the remote version again

# Whitespace got you down?



- ◉ Noticing “changed” files in working directory that you can’t reset with “git reset --hard HEAD”?
- ◉ It may be whitespace issues
  - ◉ 

```
git rm --cached -r .
# reset isn't enough on its own
git reset --hard
```
- ◉ Then set your appropriate autocrlf
  - ◉ And have your team do the same

# Recap: Fixing issues in git



- We witnessed the power of the reflog, which allows us to track our local history of actions and branch changes
- Using the reflog (and log) we can take control of our branches, and fix a lot of snafus, with git reset and git revert
- Hopefully we have a good understanding of what HEAD is
- And we know how to navigate through commit ancestry using ~ and ^

# Lab: Reflog and reset



- In the “about-us” repository, from **master**
  - Add two commits with arbitrary changes
- **Ack!**
  - We should have branched.
  - Use `git reflog` and `reset` to undo those commits w/out losing the changes
- Then create a new branch off **master**
  - Commit your changes
  - Create one more commit
- Merge the branch into **master**
  - Oops! Didn’t mean to merge...
  - Use `git reflog` and `reset` to undo the merge
- Now merge with --no-ff
  - **Ack!**, we didn’t want to do that, either
  - But what if this was already public?
  - Use `git revert` to undo the merge
- We have a messy master (compared to the upstream)
  - How would you fix it?

# Git hooks



- A way to fire off custom scripts when certain actions occur
  - Stored in .git/hooks
- To enable one, just put the appropriate named file in the hooks subdirectory
  - Check out the samples
    - ls .git/hooks
- Client-side (committer)
  - pre-commit, prepare-commit-msg, commit-msg, post-commit, pre-rebase, post-rewrite, post-checkout, post-merge, pre-push, pre-auto-gc, plus some email ones
- Server-side (on the remote)
  - pre-receive, update, post-receive
- They are local to a repo

# A sample pre-commit hook



- Add “prepare-commit-msg” file to .git/hooks
  - ```
#!/bin/sh
echo "# Please include a useful commit message!" > $1
```
- Don’t forget to make it executable
  - ```
chmod +x prepare-commit-msg
```
- Then try a commit!

# A sample server-side hook



- create a bare repository
  - git init --bare remote-sim
  - This will be our virtual remote
- Let's set up a post-receive on the “remote”
  - Create and edit “post-receive” in .git/hooks/
  - Add this content...
  - <https://gist.github.com/mrmorris/3e1c5087ed52303a9ea6>

# Now to set up a local repo



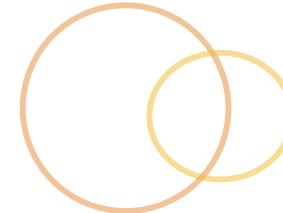
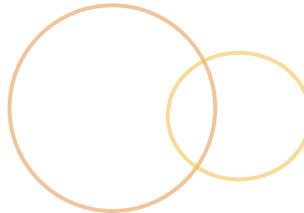
- Create another repository and add the bare repo as origin
  - cd ~/ && git init local-repo && cd local-repo
  - git remote add origin ../path/to/remote-sim
- This repo is our local, while the remote-sim repo is our remote server
- Create a commit then push master to the origin
- Check the result on the remote-sim repo

# Module: Getting more out of git



- We'll revisit some commands and expand on their options, to get more out of what git has to offer

# Patching



- Some git tools give the option of doing things as a patch
  - add, reset, stash
- Commit only partial changes with “--patch”
  - git add --patch
  - Or use git gui
  - Or use *interactive*
    - git add -i

# What else is there?



- Submodules
  - Have repositories inside your repository
- .gitattributes
  - Ability to set (and track) directory-specific file rules
- Git and SVN
  - There is a bridge available
  - GitHub has an importer

# Advanced branch management



- ◉ Tired of all those local branches?
  - ◉ Alias:
    - ◉ `git config --global alias.clean-branches = !sh -c 'git checkout master && git branch --merged master | grep -v master | xargs -n 1 git branch -d'`
  - ◉ Or a script
    - ◉ <http://rob.by/2013/remove-merged-branches-from-git>

# Git in your IDE



- Manage branches, checkouts, see annotations, built-in diffing, etc...
- Eclipse
  - Ships with a plugin, egit
  - Switch to the “git perspective”
    - Window > open perspective > other... select Git
- JetBrains
  - Enable “Git Integration” plugin
  - Make sure path to git is correct

# Git and Jenkins (or other CI systems)



- Install the Git plugin
- Create an SSH keypair for your jenkins user and add it to your git repo (on github)
- Set up a git user and email
- Connect to your git repo to test
- Create a new job in Jenkins
  - Use “git” for Source Code Management
  - Enter repo URL
  - Select the branch; jenkins will pull from here when a build is started
- Set up the post-receive hook in your repo (github)

# Best Practices in Git



- Commit early and often
- Useful commit messages
- Branch new work (don't work on master)
- Use remotes for people (not branches)
- Don't change published history
- Keep up to date
- Establish a branching and team workflow
- Tag your releases
- Don't commit configuration/secure stuff
- In an emergency, use the reflog

# Module: Git internals



- We'll look behind the curtains of git and how it stores the repository data
- You'll get a better sense of how git works and how commits, branches and tags come together
- These commands may not have practical use, but they can be helpful to firm up the concepts

# The .git directory



- `ls -la .git`
  - Refs contains our tags, branch and remote branch references
  - `cat .git/refs/heads/master`
    - Shows it as just a commit id!
  - `cat .git/HEAD`
    - Shows it as a reference to a branch
    - Unlike branches, it is not a direct link to a commit
  - If you checkout a commit, HEAD will contain the commit id and will be considered a “detached HEAD”; it does not coincide with a local branch

# Git 4 main objects



- **Tree** objects are representations of the snapshots, recording the state of a directory at a given point
- **Commit** object specifies the top-level tree for a snapshot of the project at that point (author + committer + message)
- **Blobs** represent file data
- **Tags** are objects that represent a commit; they can be annotated (and therefore objects in their own right) or just reference a commit (lightweight)

# Cat-file a commit



- `git cat-file <type> <object>`
  - Will output an object's info
- `git cat-file commit HEAD`
  - Commit info, like tree id, parent(s), author and committer information, commit message

# Git ls-tree a tree



- Trees contain binary data, so we can't use cat-file
- git ls-tree 61d35fb
  - Displays what looks like a directory listing

## Cat-file a blob



- From our tree listing, pick a blob
  - `git cat-file blob bef7f4ae`
  - Displays the entire file content
- Since all these objects are identified by their SHA-1 checksum, identical blobs are not duplicated (but are shared)
  - But, yes, a one-liner change results in two blobs

# Cat-file a tag



- `git cat-file tag v1.0`
  - Contains the commit id associated with it
  - Along with author, message information
- Only annotated tags are “objects”

# So what is a branch?



- `git cat-file -t master`
  - -t gets us the type of the object
  - “commit”!
- A branch is just a reference to a commit
  - `git cat-file commit master`
- `.git/refs` includes
  - Branches
  - Remote tracking branches
  - and Tag *references* (which point to tag *objects*)

# gc & packfiles



- Git cleans up duplicate data by packing it into “packfiles”
  - Initially, the data is called “loose” object format
- Tell git to run its garbage collection & packing
  - `git gc`
  - And packs up objects into `.git/objects/pack`
- Dangling files (not referenced by any commit) are not packed
- Packfiles compress down by looking for files that are named and sized similarly and store their deltas from one version to the next
- Newer versions of files are stored in full, while older versions in the pack are deltas (to optimize accessing the newer files)

# GitHub Pages



- An easy way to get a static (+js) page up via a repository
- Create “username.github.io” repository
  - Ex: mrmorris.github.io
  - Add an index.html file to your clone and push it up
  - Or use their layout bootstrapper to start
- Visit <http://username.github.io>
- Supports Jekyll installations
- Can set CNAME to use your own domain
- Has a nice layout bootstrapper

# Lab: GitHub Page Teams



- Group up into teams of 3-4. Each of you will be acting as a “Repo owner” of your own repo and “Team member” across your team’s repositories.
- As Repository Owners:
  - Add your team as collaborators
  - Create an initial issue: Add initial index page with a small profile about you (Assign to you)
  - Create an additional issue (at least one) per team member and assign to each team member. Need ideas?
    - Add a simple page about yourself and link to it from the index
    - Add an awesome animated gif to the index
    - Edit my index and bold all mentions of my name
    - Keep it light and simple
  - You are responsible for providing feedback on the issues in YOUR repository. You are providing approval to merge (or merging them yourself).
- Repository Team Members:
  - Beside that repo you own and manage...
  - You'll get a couple issues assigned to you. You are expected to:
    - Locally clone each repository that you are working on
    - Branch off master to submit your work to the repo
    - Create a pull request to resolve your issue
    - Get PR approval before merging
  - It would be nice of you to also review code changes across all repos you are involved in.
- Check out your progress at <http://<username>.github.io>