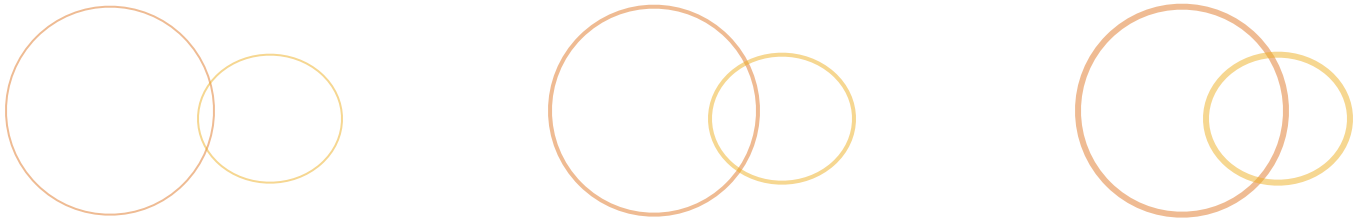


GIT FOUNDATIONAL

Ryan Morris

mr.morris@gmail.com


THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

Logistics/Introductions

- Class structure
- Introductions
  - Reach me at mr.morris@gmail.com
- You?
  - Any experience with version control?
  - Any experience with Git?
  - What are your goals for this class?

Class style

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

Goals

- To work comfortably with Git from the **command line** (no GUI)
- 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 remotes
- This class is geared towards **beginners**

Resources

- Slides
  - …
- Cheat sheets
  - https://training.github.com/kit/
  - http://zeroturnaround.com/rebellabs/git-commands-and-best-practices-cheat-sheet
- The Git Parable
  - http://bit.ly/1isB3K4
- Pro Git, 2nd edition (for free!)
  - http://git-scm.com/documentation
- Visualizing Git
  - http://pcottle.github.io/learnGitBranching

Planting the seeds

- **Stage** and **commit** files to your repository to keep track of changes over time
- Create **branches** to support non-linear projects, diverging ideas/experiments, versions, etc.
- **Merging** to combine branches together
- **Remotes** to link your repository to your team(s) and other copies of your repository so you can share work and work collectively on a project
- Consider how to manage your **history** and why that matters; **squashing** commits, **rebasing** when you merge, the nuances of "**reset**"

What is/why version control?

What is Version Control?

- A way to manage and organize changes in a set of documents, files, etc.
- From a software perspective, consider the codebase the crown jewels.
  - Many engineers are working on it
  - Different changes happening at different times, bug fixes, features…
  - Lots of mistakes over time, human error, etc…
  - The ability to organize changes and refer back to older versions is critical

Why Version Control?

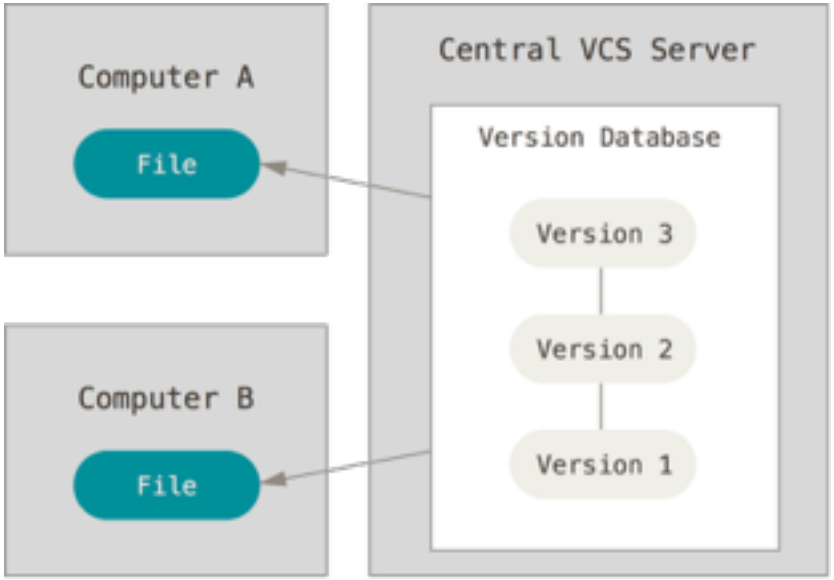
- ***Keep track*** of changes
- ***Go back*** to an older version
- ***View a history*** of changes
- ***Collaborate*** easily

And maybe… Automate operations like deployments, etc

Stone-age version control

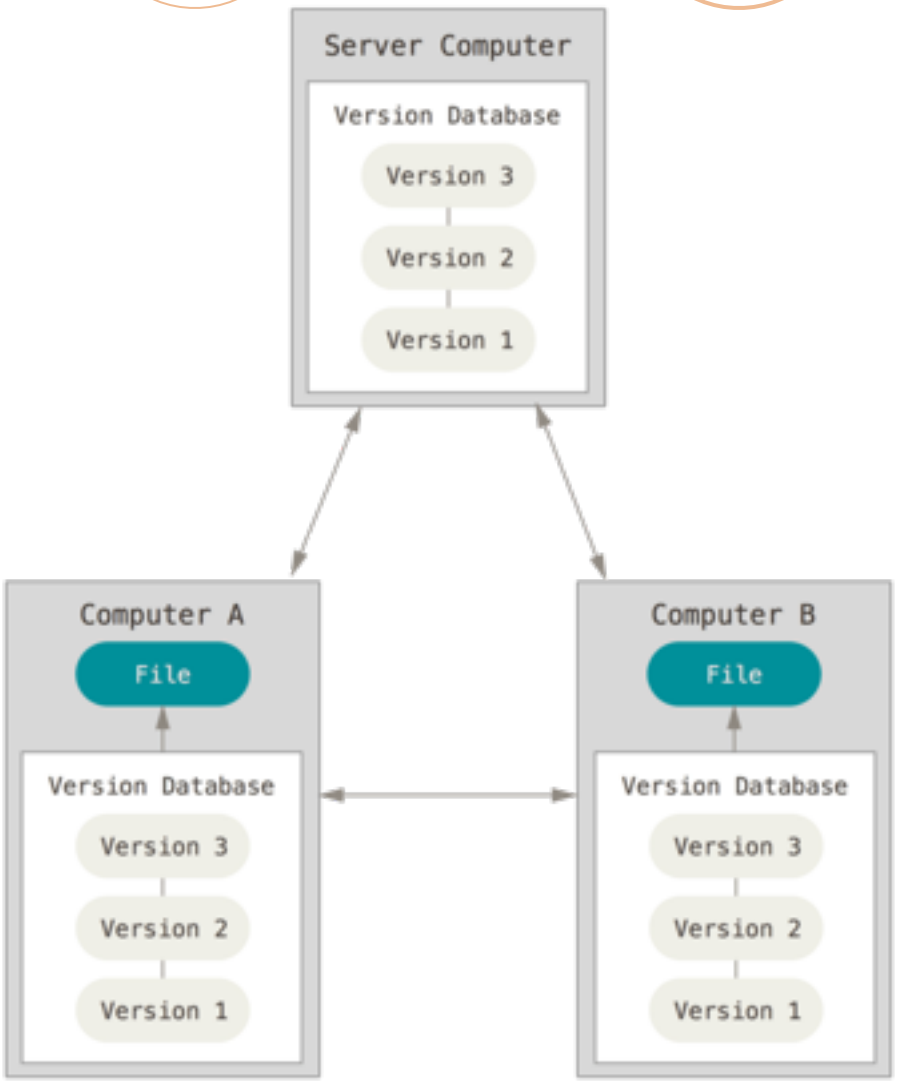
- You're editing file(s) and want to keep track of changes…
  - `/my-files/myfile.v1`
    `/my-files/myfile.v2`
    `/my-files/myfile.v3.draft`
- Advantages
  - Easy
- Disadvantages
  - Error prone
  - Single point of failure

Old World: Centralized Version Control



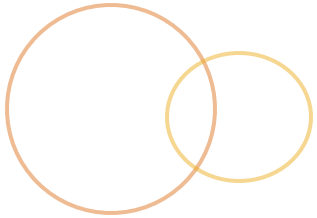
- Single central server manages all operations
- ex: CVS, Subversion, Perforce
- Advantages
  - Fine-grained control
  - Easy to see who is doing what
- Disadvantages
  - Single point of failure
  - History is not local
  - Branching/merging is a pain

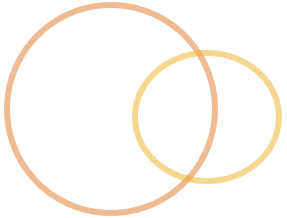
New World: Distributed Version Control



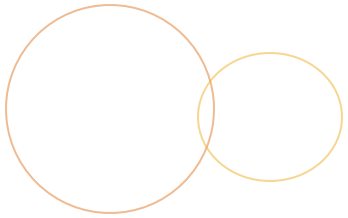
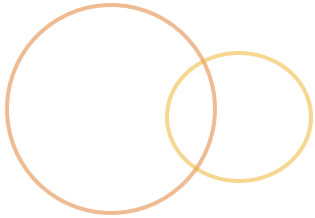
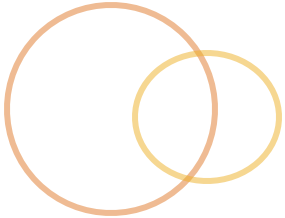
- All clients fully error the repository (incl. history)
- ex: Git, Mercurial, Bazaar
- Advantages
  - No single point of failure
  - Easy to collaborate
  - Flexible workflows
- Disadvantages?
  - Less control over access
  - No file locking

History





- Linux (1991-)
  - No real version control from 1991-2002
  - **bitkeeper** from 2002-2005
  - **git** created in 2005 after relationship w/ bitkeeper broke down
- Goals
  - speed
  - simplicity
  - support for non-linear development
  - distributed
  - support for large projects


module

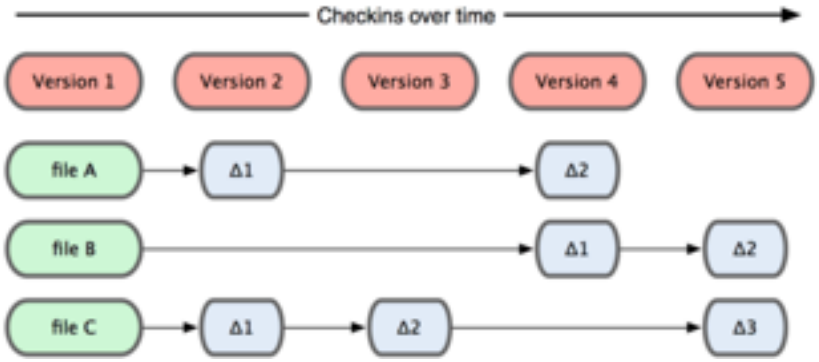
ABOUT GIT

What makes Git different

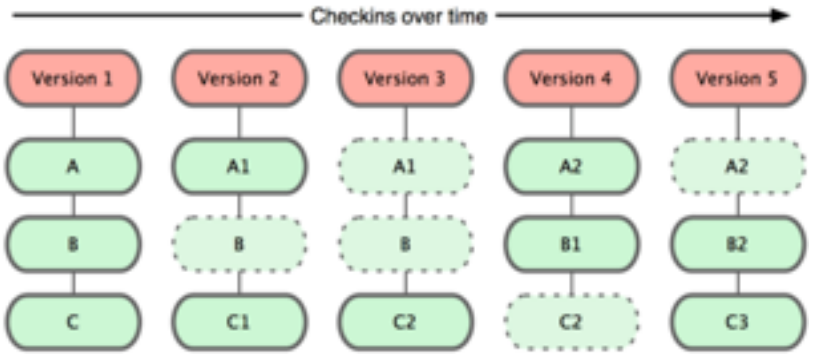
- *“Just a simple key-value data store”*
  - You can insert any content and it will give you a key you can use to retrieve the content
- Git stores **snapshots**, not differences
- Nearly every operation is **local**
- **Branching** is easy and encouraged

Snapshots vs Diffs

- Git stores data as snapshots of your filesystem in each commit, opposed to a delta or difference
  - A picture of what your files look like at that moment.
- Why is this a big deal?
  - It makes git more like a mini-filesystem with some powerful tools rather than simply a VCS



Diffs



Snapshots

Local operations

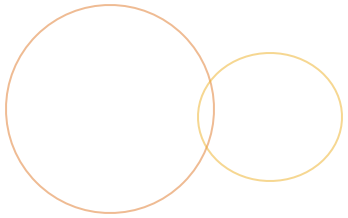
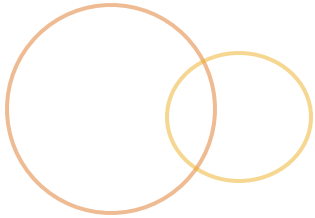
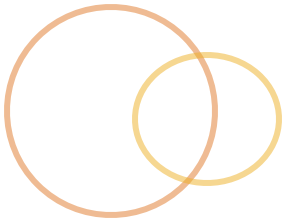
- Entire repo history is local so you don’t need to wait for network to check logs, etc.
- No dependency on a server to perform work, branch, commit

Git has (data) integrity

- Everything in Git is hashed (using the SHA-1 algorithm) before it's stored, and from then on is referred to by its hash
  - **`24b9da6552252987aa493b52f8696cd6d3b00373`**
  - Impossible to change or corrupt a file without Git knowing about it
- Commits (and other objects) are referenced by their hash
  - id, reference, hash…

Git generally *adds* data

- When you perform actions with Git, nearly all of them ADD data to the Git database
- Difficult to get Git to do anything that can't be undone or to erase data in any way
- As with any VCS, you can lose or overwrite uncommitted changes, but after committing, it's quite difficult to lose anything.


module

SETUP

Install Git

- Install it
  - 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/
- We'll be using the command line
  - Is everyone ok with **cli basics**?
- Make sure it's installed
  - `git --version`
  - Make sure you're on 1.7-ish or above

Git configuration

- Git's configuration is stored as a plain text file based on the configuration "level"
- **`~/.gitconfig`**
  - This is your **global** configuration, affecting all of your repositories
- **`.git/config`**
  - This is your **local** configuration, it affects only the repository in which it is located
- There is also a **system** configuration for all users on the system stored in **/etc/gitconfig**
- All levels are applied, with local overriding global overriding system

Set/View Configurations

- Set config values
  - `git config <key> <val>`
  - `git config --global user.name “Ryan Morris”`
  - `git config --local user.email “hi@me.com”`
- Check config values
  - `git config <key>`
  - `git config user.name`
- View all configs
  - `git config --list`
  - `git config --global --list`

Configure your text editor

- Text editing will default to **vi** or **$EDITOR**

- Emacs:
  - `git config --global core.editor emacs`
- Atom:
  - `git config --global core.editor "atom --wait"`
- Sublime (requires *subl*, cli command)
  - `git config --global core.editor "subl -n -w"`
- Text Wrangler:
  - `git config --global core.editor “edit -w”`
- Notepad:
  - `git config --global core.editor notepad`

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 *maybe* your text editor
  - `git config --global core.editor emacs`
- Double-check your configurations
  - `git config --list`
  - `git config <key>`
- Inspect the config file(s)
  - `cat ~/.gitconfig`

Creating a repository

- Make a root directory for the repo
  - `mkdir about-me`
- Then initialize it as a new repository
  - `cd about-me`
  - `git init`
- Or do it all in one step
  - `git init about-me`

Inspecting our repository

- Check the status and the log
  - `git status`
  - `git log`
- Check what git has initialized
  - `ls -la`
  - `ls -la .git`

Lab: Create your first repo

- Initialize a repository called "**first-repo**"
  - `git init first-repo`
- Check the status and log (*don't forget to cd in to the repo*)
  - `git status`
  - `git log`
- Find the git database?
  - `ls -la`

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?

- A **snapshot** of your filesystem (just the repo)
  - "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
- A commit is referenced by it's ***hash, id*** or ***sha***
  - ***42d484c401f0a19cc8a954c16240821329acefac***

Tracked vs untracked

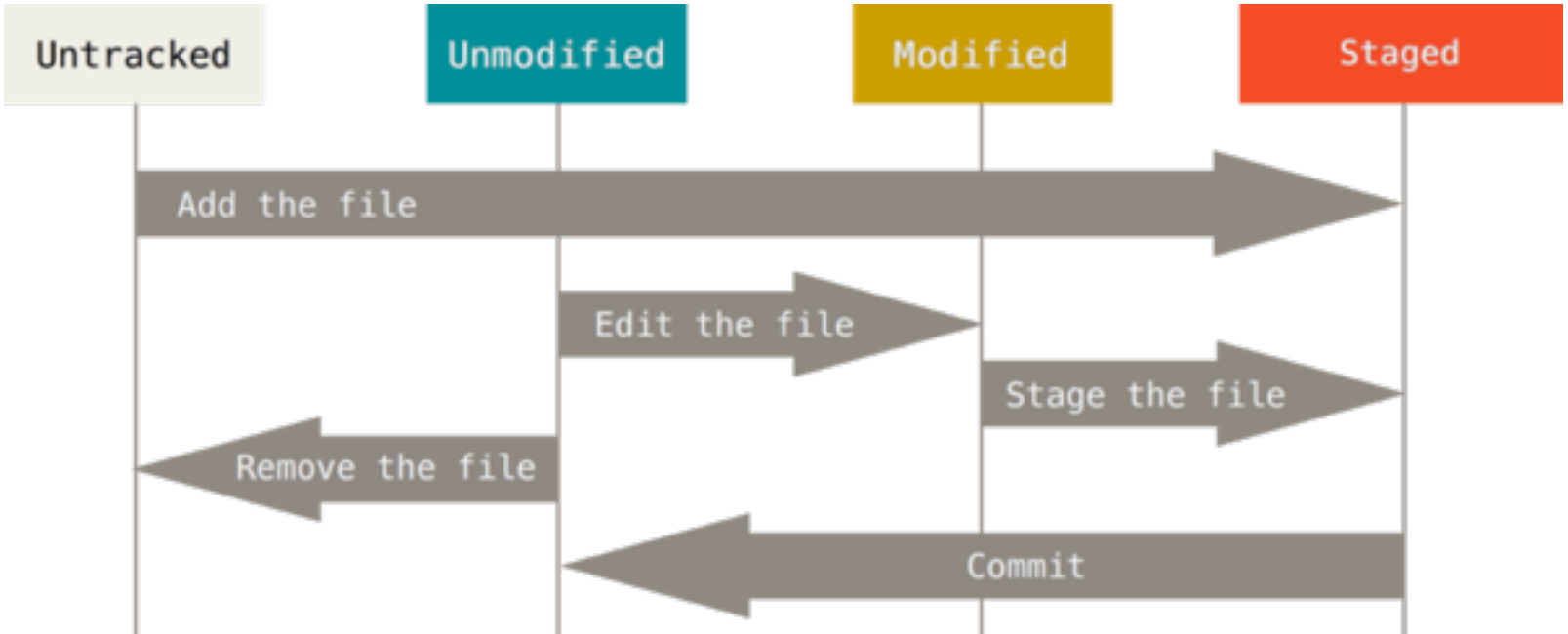
- To git, files are considered either:
  - **tracked**
    - a file that's been staged or committed
  - or **untracked**
    - a file that's new to in your working directory

Lab: Add something to your repo

- Create a file with an editor or…
  - `echo “Junk” > firstfile`
- Stage the file
  - `git add firstfile`
- Check the status
  - `git status`
- Commit the change
  - `git commit`
- Check the log
  - `git log`

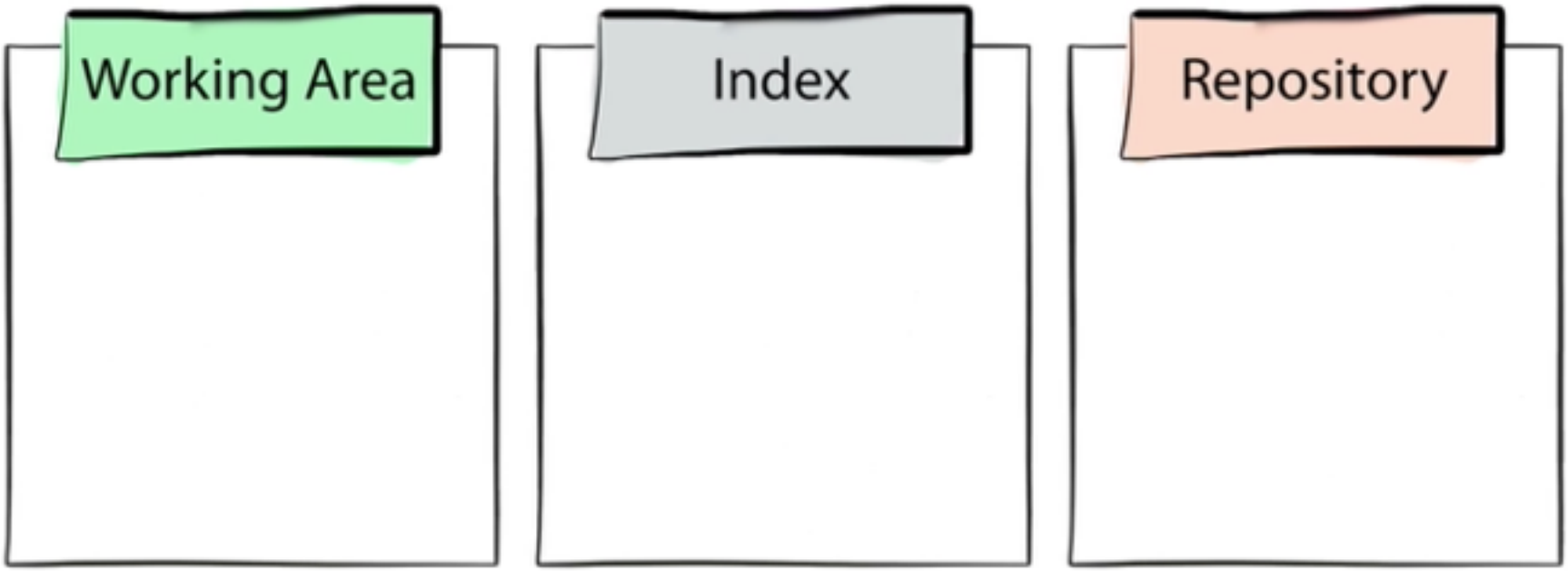
The basic workflow

1. **Add or modify** files in your working directory
2. **Stage** files (or changes) that you want to be included in the next commit in the index
3. **Commit** the index as the next snapshot, which stores the changes in the repository



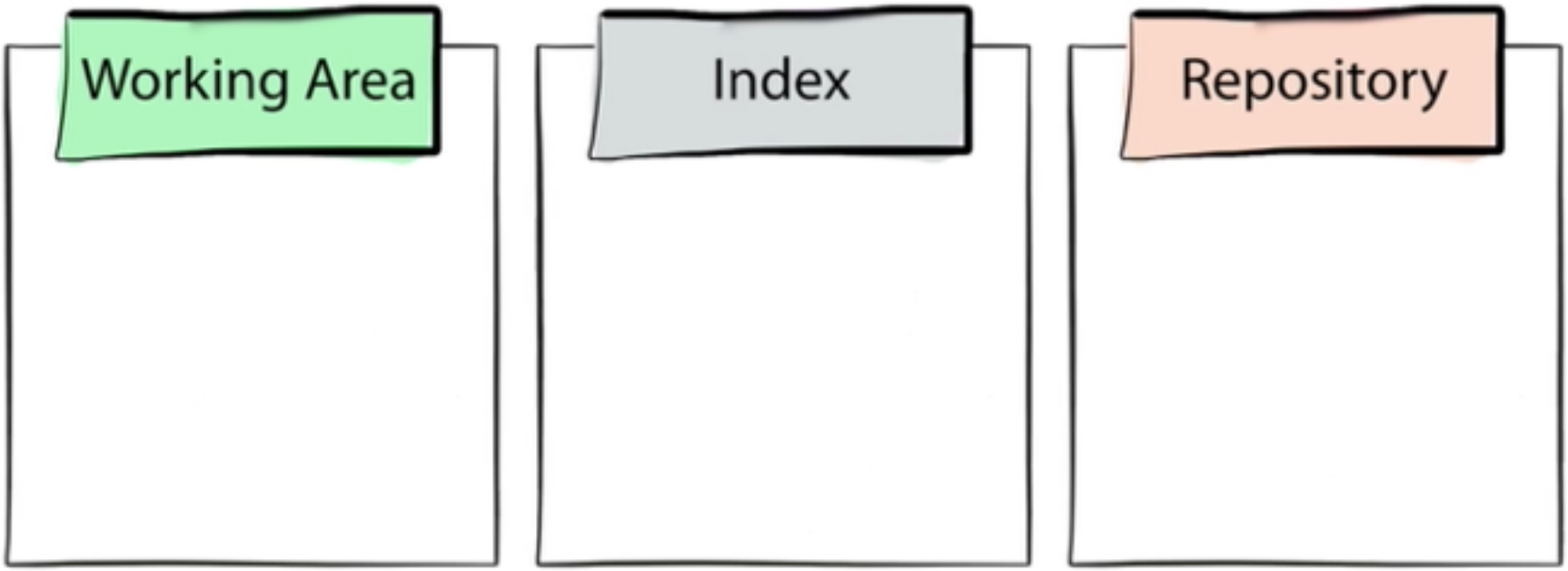
The three trees

- Git projects store things in three *areas*
  - **working area/directory (WD),** where you edit your files
  - **staging area (index),** where you prep the next commit
  - **repository,** where git stores your data (/.git)



Mastering Git

- To truly understand a git command, understand how it affects the three trees/areas…
  - **how does the command move data between areas?**
  - **how does the command affect the repository?**



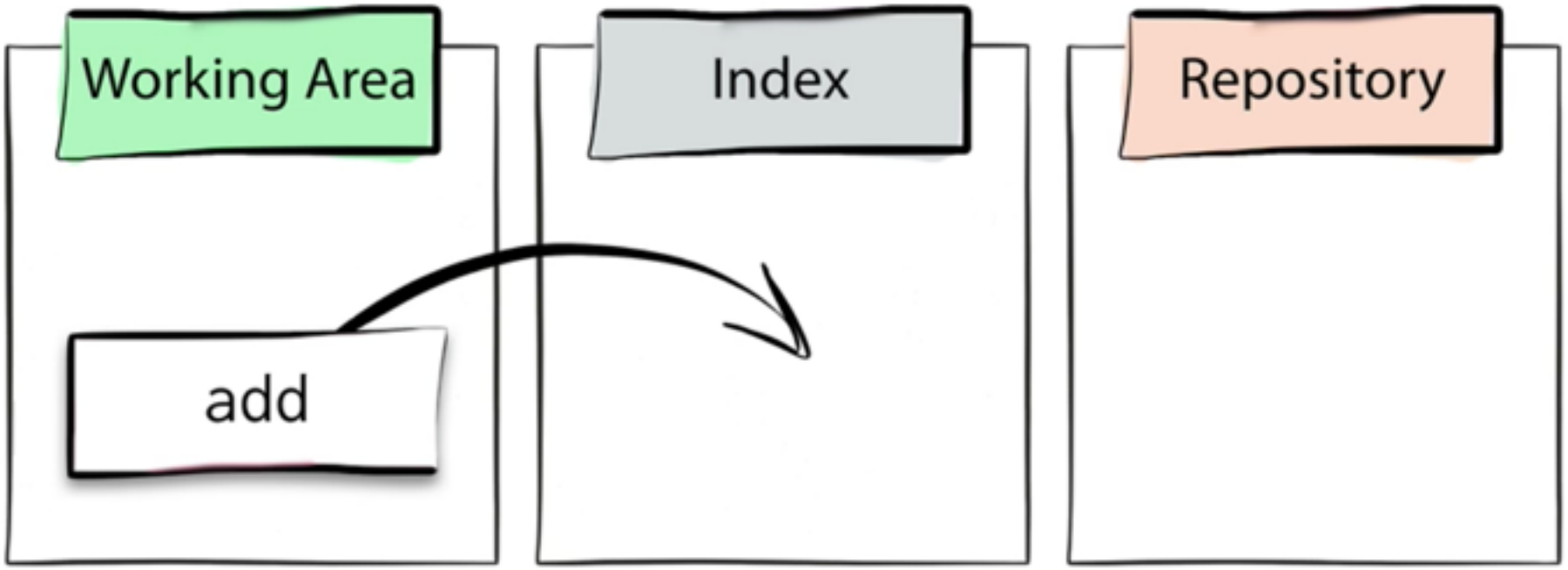
Git status

- `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

The Staging Area

- The **staging area**, aka ***index***, is where you prepare a set of changes as your next commit
  - A proposed commit
  - Forces you to craft your commit
  - Allows you to be selective about your changes
  - Staged changes are NOT part of the repository history…
- `git add <pattern, file or files>`
  - Stages the file(s) or changes
  - Begin tracking new files
  - **[pro]** Marking conflicts as resolved

Staging area

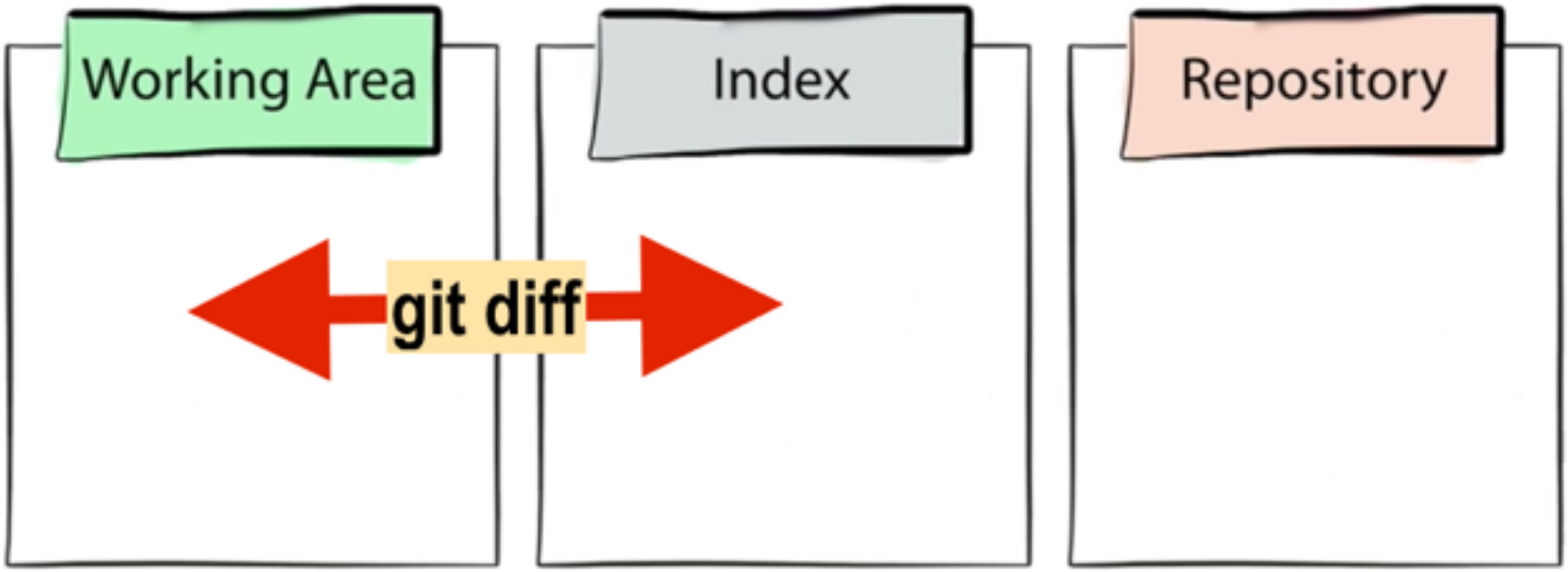


Staging with **git add**

- 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 .`

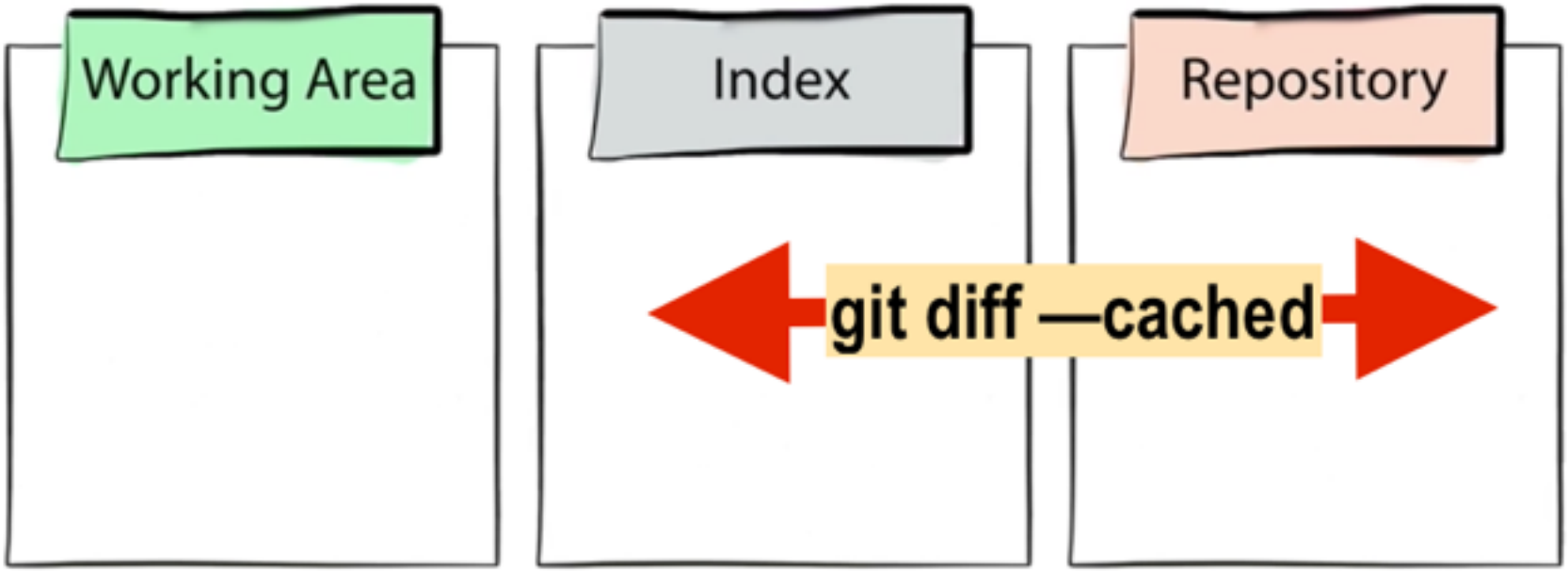
Seeing what has changed

- `git diff`
  - Shows difference between **working directory** and the index
  - Does not include untracked files



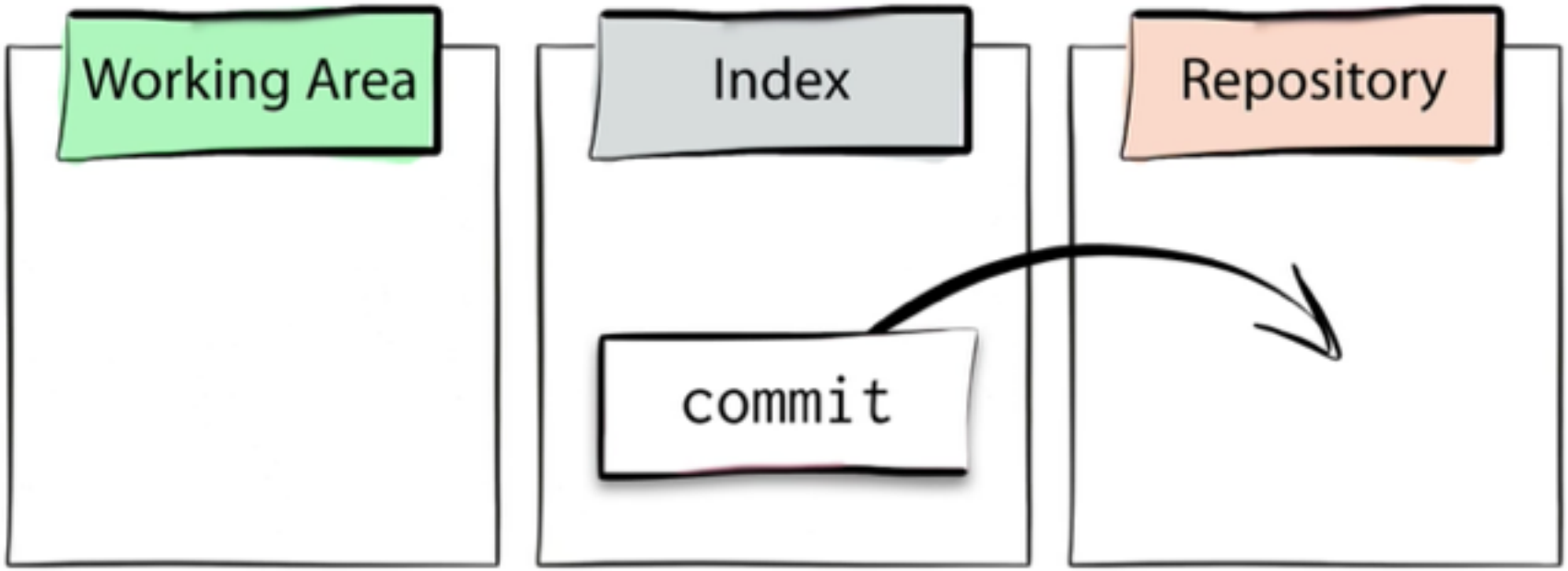
Seeing what has changed pt.2

- `git diff --staged`
  - Shows diff output of what you have **staged** compared to the repository
  - `git diff --cached`



Committing

- Once you stage files/changes you can commit them to the repository
- `git commit`
  - Applies files/changes from the staging area into the repo
  - …by creating a **commit object** in your git database



Committing

- `git commit`
  - Prompts for a commit message, which is required
- `git commit -m "My commit message"`
  - Shortcut to set a message as you commit
- `git commit -a`
  - Skip staging (auto-stage tracked file changes)
  - *I don't recommend using this*

Viewing commits

- `git show`
  - Show info about your latest commit
- `git show <commit>`
  - Show info about a specific commit by id
    - `git show 42d484c401f0a19cc8a954c16240821329acefac`
  - Can also reference in **abbreviated form**
    - `git show 42d4`

History, the git graph

- Each commit you create is related to the previous commit (parent-child)
  - In this way a history is generated
    - current commit, to previous, to previous, etc…

- `git log`
  - Show history from current commit, back
  - Lots of options
    - `--oneline`
    - `--graph`
    - `-<n>`

Removing and moving files

- You can do it however you normally would, then stage the change, that said…
- `git rm [-r] <file or dir>`
  - Removes a file/dir
  - Auto-stages the change
- `git mv <old name> <new name>`
  - Removes the original, adds the new
  - Auto-stages the change
  - Git doesn't explicitly track file movement

Recap

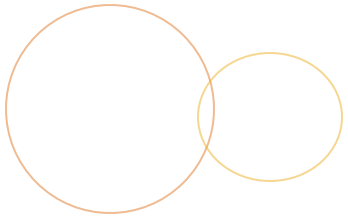
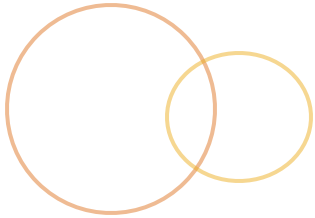
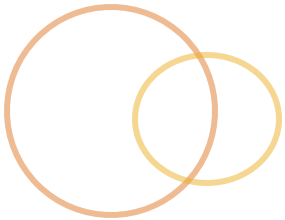
- **git init -** initialize a new repo
- **git status -** status of working directory, changes to stage or that have been staged
- **git add -** stage changes, prepare to commit
- **git commit -** commit changes, adds the change into the repository as a commit object
- **git diff -** see what you've been editing
- **git show -** to see info about a commit
- **git log -** view the history of commits

Lab: About me, a repository

1. **Create** a new repository, “about-me”
2. Then create a txt file named with your name
   - `Touch <yourname>.txt`
   - Check the status, check the diff
3. **Stage** and **commit** the file
   - Check the log
   - Use “`git show`” on the commit id you just created
4. **Edit** the file to add a short profile about you:
   - ```
     <Your Name>
     * Born in: <where you were born>
     ```
   - Check the status, check the diff
5. **Stage** those changes
   - Check the status, check the diff
6. 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`
     ```

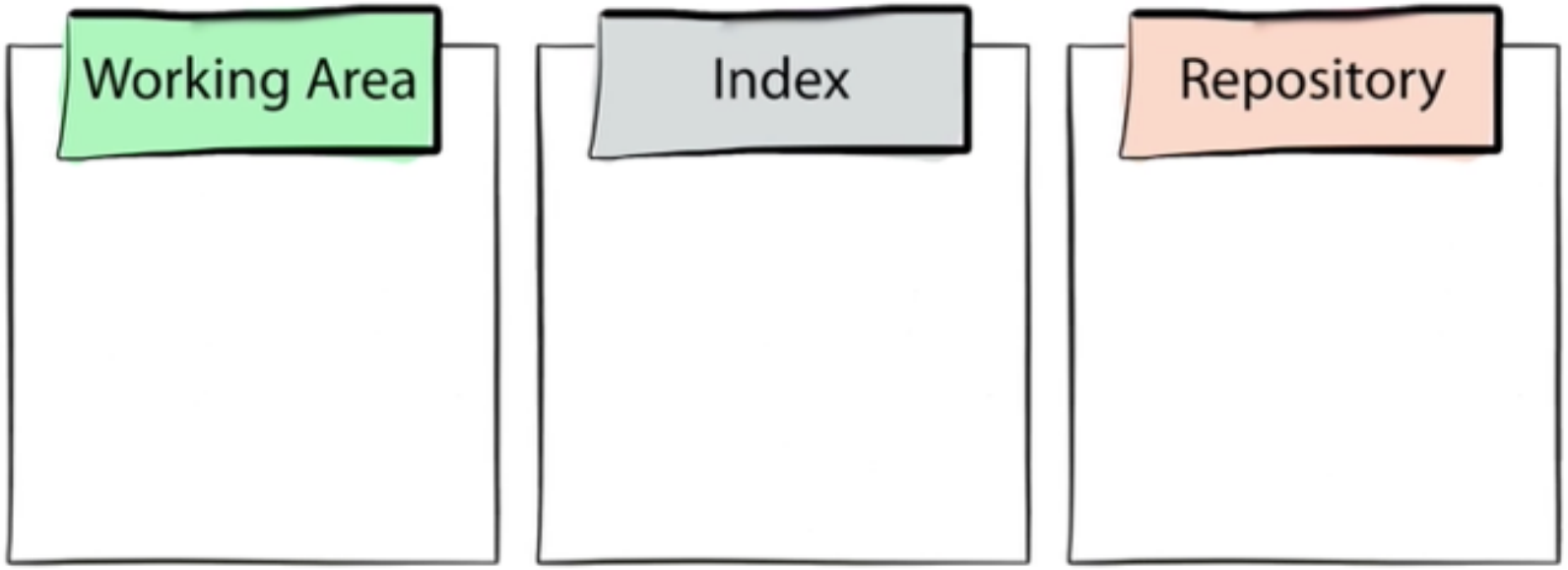





module

UNDOING

Undoing

- There are a lot of things you'll want to undo
- Undo edits things in your working directory
- Un-stage a file or change
- Undo a commit
- … among other things



Undoing (working directory changes))

- So… you've edited a file locally but want to throw away those changes
- `git checkout -- <file>`
  - Throw away local edits not yet staged
  - **Copies file from staging to working directory*
- **[pro]** You can also check out an older version
  - `git checkout <commit id> <file>`

Undoing (staged files)

- You've staged something with `git add` but want to undo the index
- `git reset <file>`
  - Remove a change from the index
  - Does not affect your working directory
  - **Copies file(s) from repository to the index*
  - `git reset HEAD <file>`

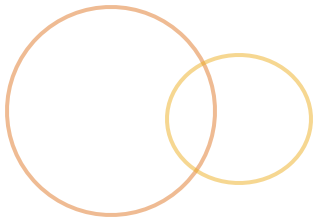
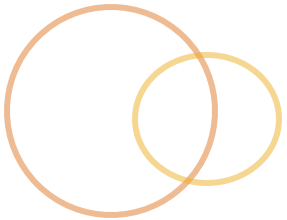
Fix (your last commit)

- You've just committed but meant to give a different commit message
  - `git commit --amend`
- You've just committed but forgot to include a change (or didn't mean to include a change)
  - *… make the changes you intended…*
  - `git add .`
  - `git commit --amend`
- To completely undo a commit
  - `git reset HEAD^`
  - *Write this down for later…*

Undo (your last commit)

- To completely undo a commit
  - `git reset HEAD^`
  - *Write this down for later…*

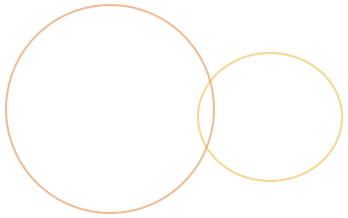
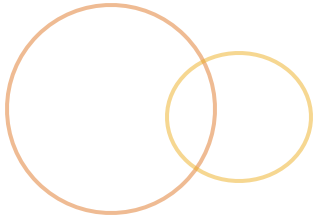
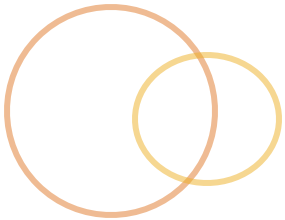
Recap


- 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`
- And undo the last commit with
  - `git reset HEAD^`

Lab: Undoing things

- Practice editing files then undoing the changes with “git checkout” and un-staging them with “git reset”
  1. Create two new files, “junk” and “LICENSE”
  2. Edit “<your-name>.txt” file, add something about yourself
  3. **Stage** all your changes
  4. Un-stage "junk" and "LICENSE"
  5. **Commit** only the change to “<your-name>.txt”
  6. **Amend** that last commit to edit the commit message
  7. **Stage** LICENSE and **commit** it
  8. Then **delete** “junk".
  9. Git **status** should show a clean repository


module

BRANCHING

Branches

- Up until now we've been working on a single branch, **master**
- We'll begin using **branches** to work on many tracks of work at once

What is a branch?

- A branch is like a fresh copy of all your files, which can have their own history diverging from the original history
- Use cases
  - ex: “I want to edit these three files to be in French, I’ll start a new branch called ‘french’ to do this”
  - ex: “I need to fix this bug, I’ll branch off master so I can make my fixes without worrying about losing my place or mixing my edits with other work I’m doing”
  - ex: “I want the team to incorporate this change but I can’t just upload it into the app via the FTP; Ill make a branch that has my changes and share that with the team so they can compare and review”

So, why branch?

- Experimentation
- Stability
- Collaborate with others
- Diverging codebases or bucketing versions
- Supports deployment workflows

Branches in git

- In git, a branch…
  - …is a movable bookmark for a commit
  - …references the "tip" of the history (tree)
  - …is one line of text in the database, it's cheap!
- A **commit** automatically moves the branch forward (to the new commit)
- Later, we'll combine branches back into master (or other branches) by **merging** them

- Visualize: http://pcottle.github.io/learnGitBranching/?NODEMO

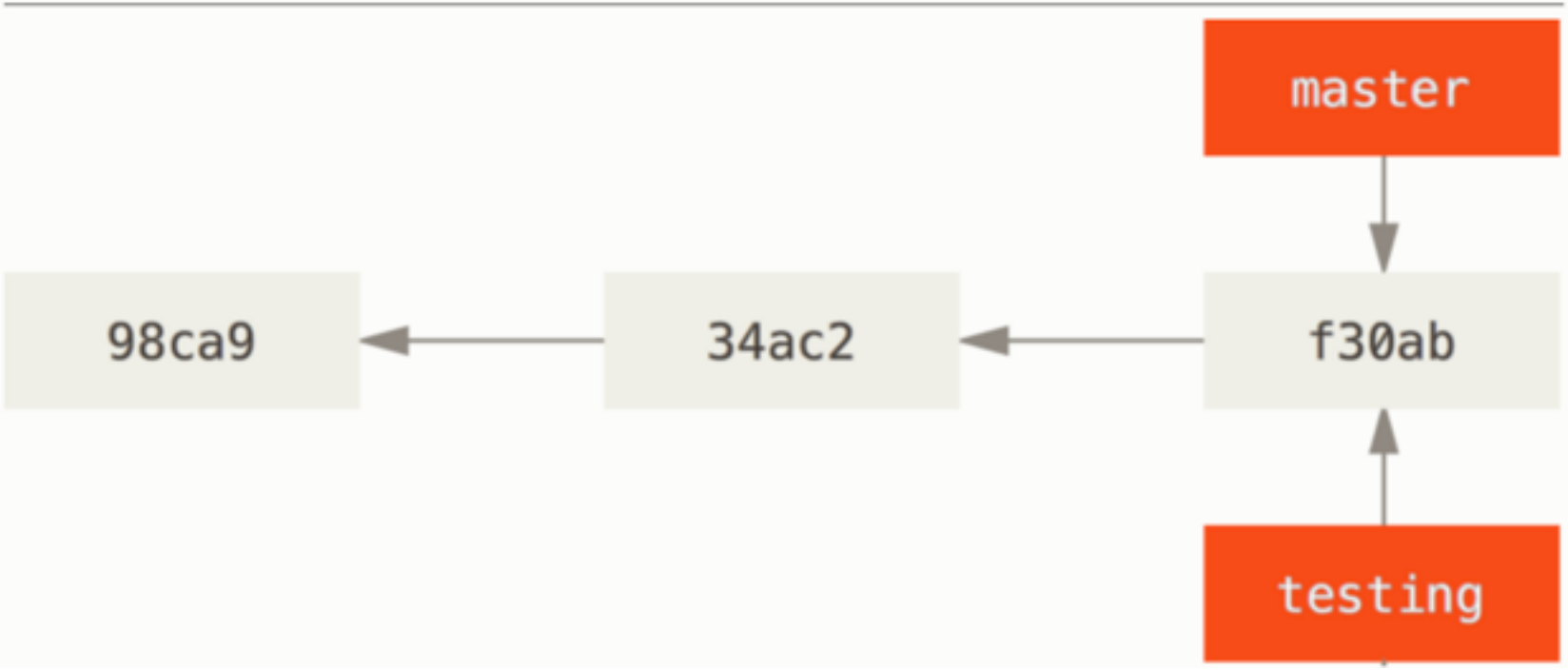
Git branch

- List all your branches
  - `git branch`
    - `--list`
- Create a branch
  - `git branch <branch-name>`
- Switch to a different branch
  - `git checkout <branch-name>`
- Create a branch and immediately switch to out
  - `git checkout -b <branch-name>`

Creating a branch

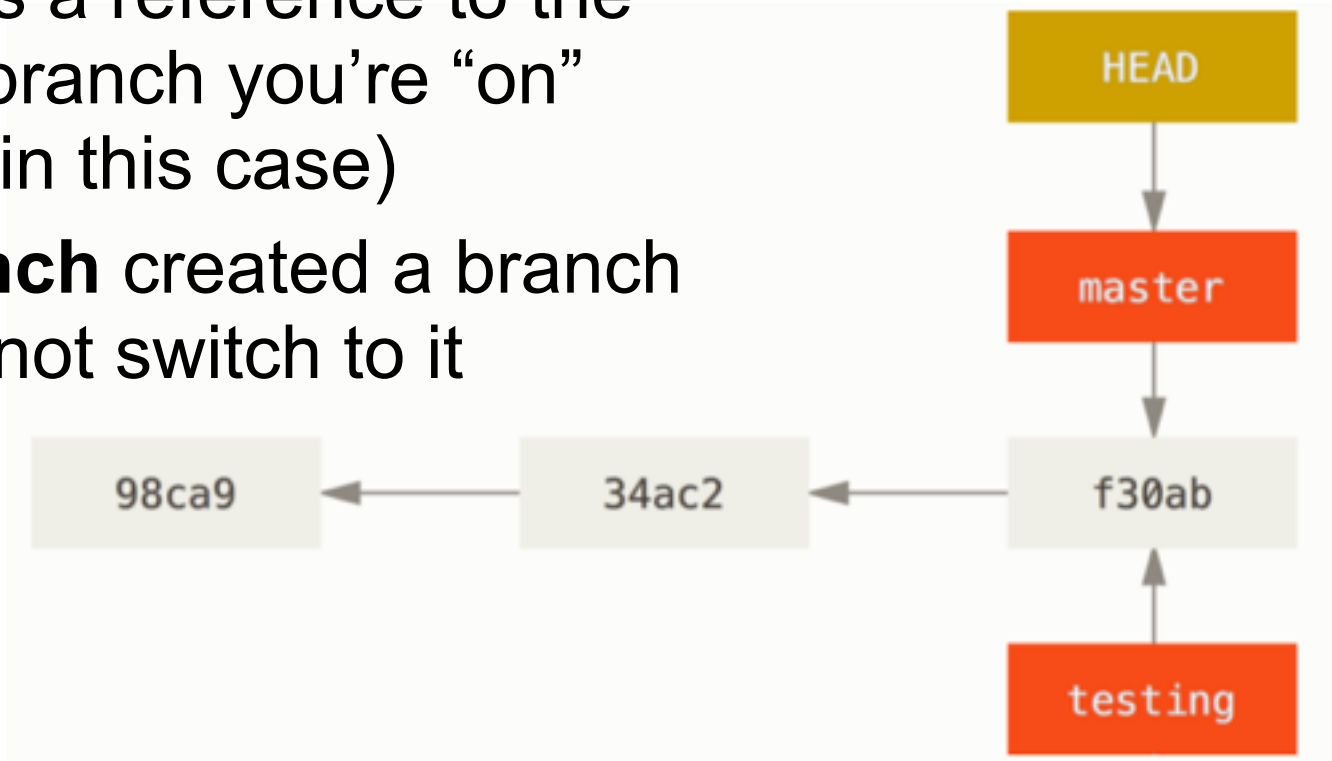
- Creating a new branch which references the same commit you're currently "on" (HEAD)
- It's not a copy of all the files, it's the same commit

```
$ git branch testing
```



HEAD

- How does git know what branch you're currently "on"?
  - **HEAD** is a reference to the current branch you're "on" (master in this case)
  - **git branch** created a branch but did not switch to it



HEAD (Cont'd)

```
$ git log --oneline --decorate -14
a1dc91c (HEAD -> master) A, B, and C all in one commit.
2c32bd8 Added new file based on the Gazornin protocol.
a0d137d Adding
f282e95 done
0539b46 7
c958298 3
69cf79b 2
86d1e62 1
b2f34d7 0
9943434 Solved Middle East peace problem. Next!
b65ff27 (b2) Adding conflict.txt
f2f5977 (b1) Adding conflict.txt
3b09dbe Merge branch 'newbranch' Preserve history even
47768e4 (newbranch) Here we go, in the branch
```

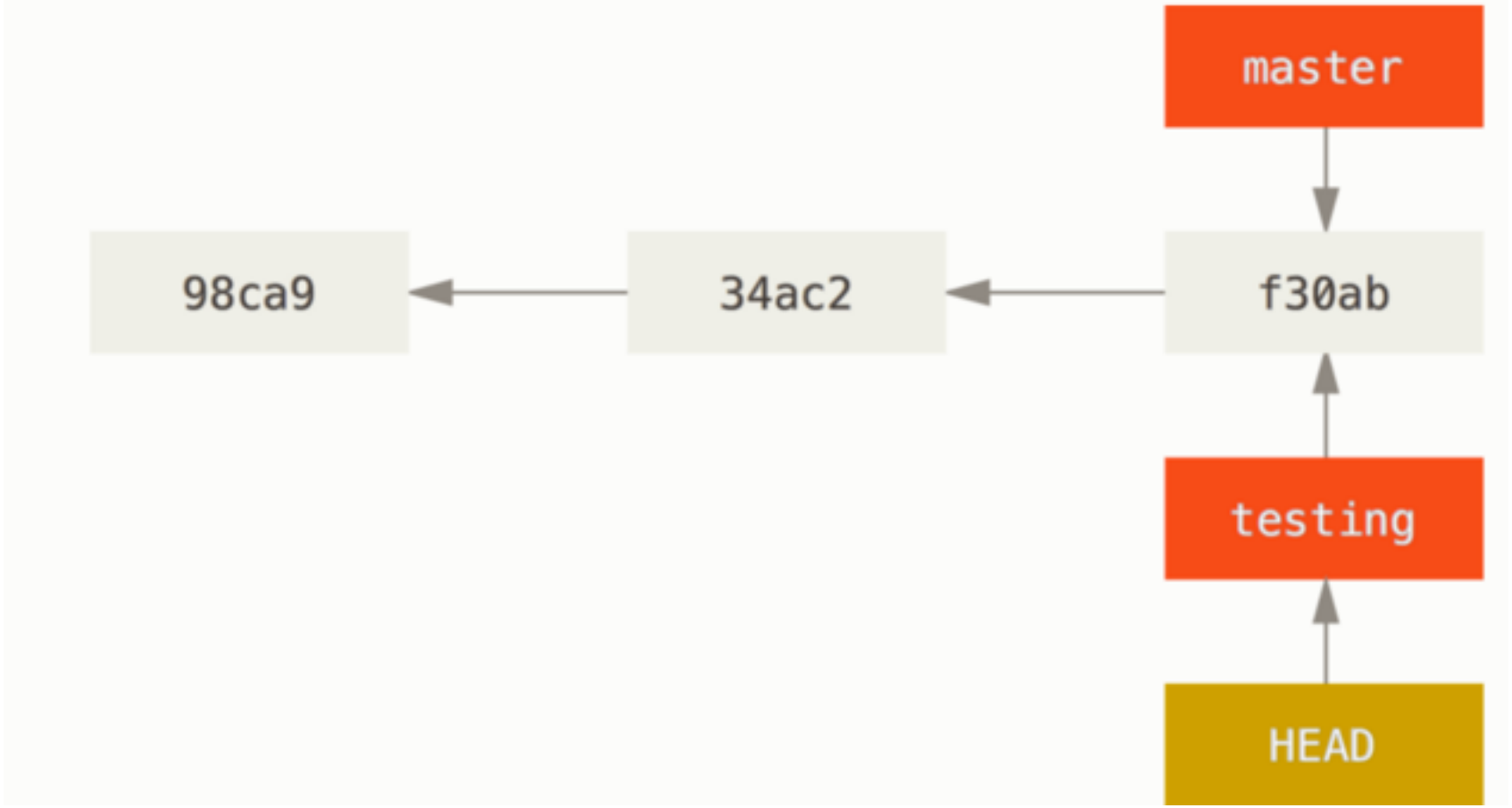
Git checkout

- Multi-purpose command that 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>`
    - Leaves you in a “detached state”, which just means it is not pointing to a branch reference and commits can’t be made

HEAD REDUX

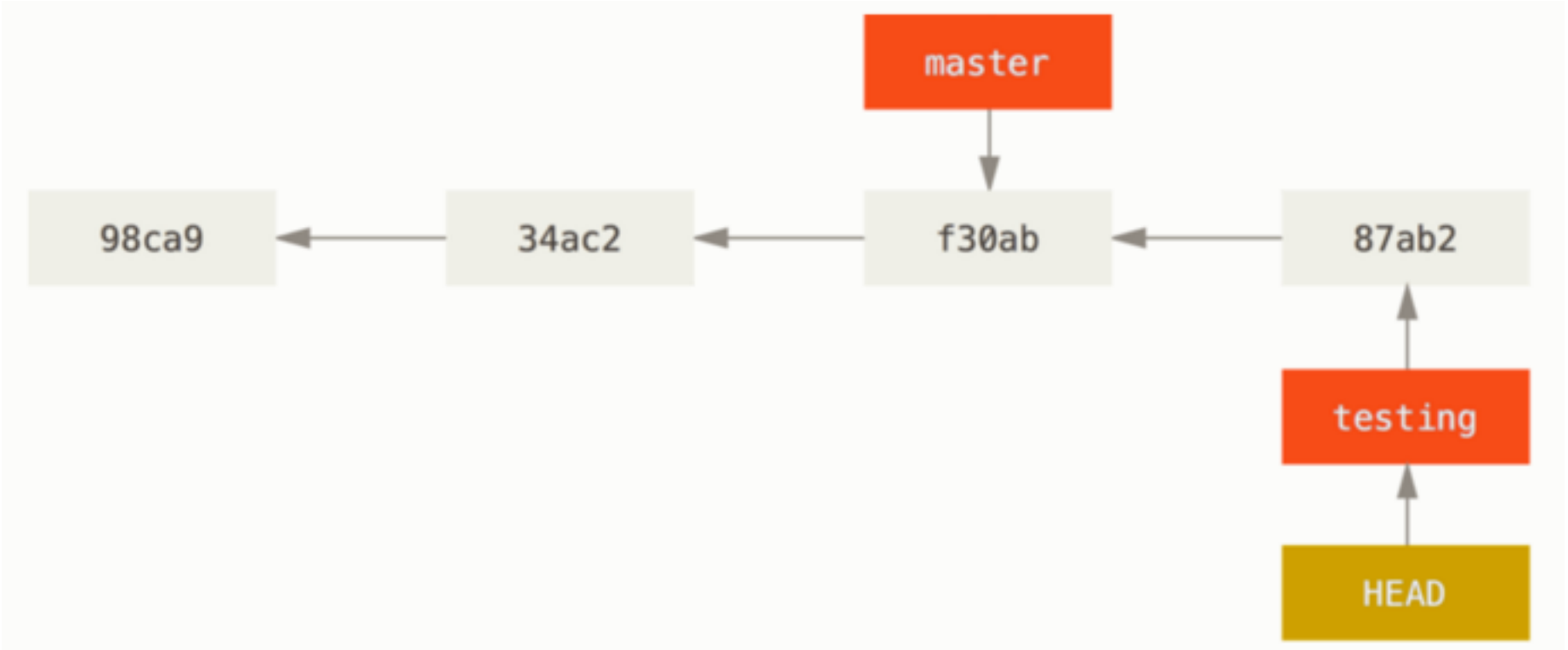
- Git checkout updates where HEAD points

```
$ git checkout testing
```



HEAD REDUX (Cont'd)

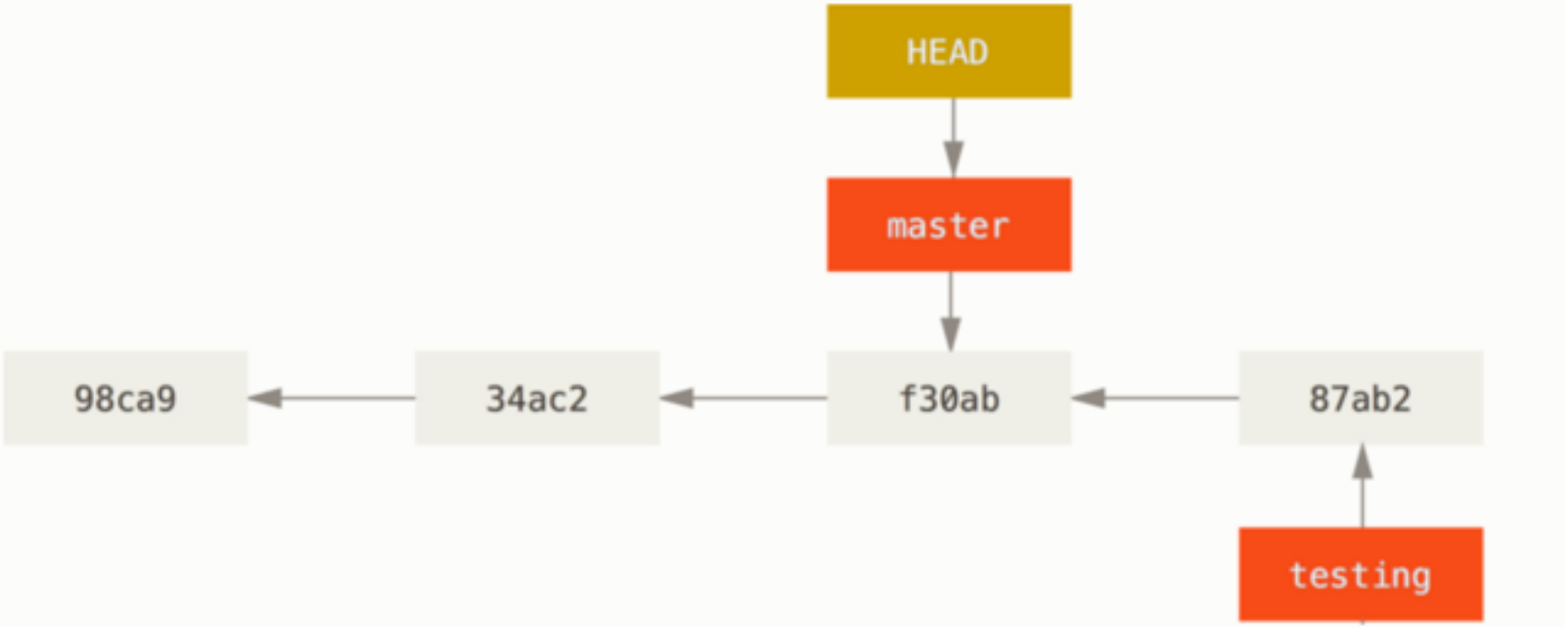
- Now another commit



HEAD REDUX (Cont'd)

- And finally, switch back to master

```
$ git checkout master
```

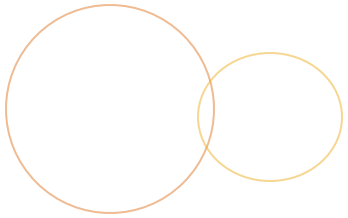
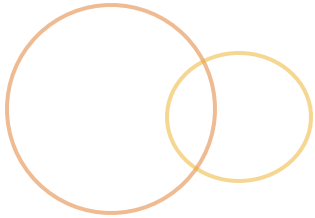
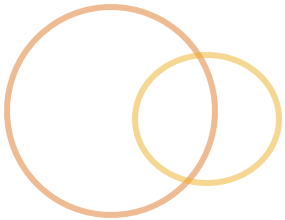


Log, revisited

- 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

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


module

MERGING

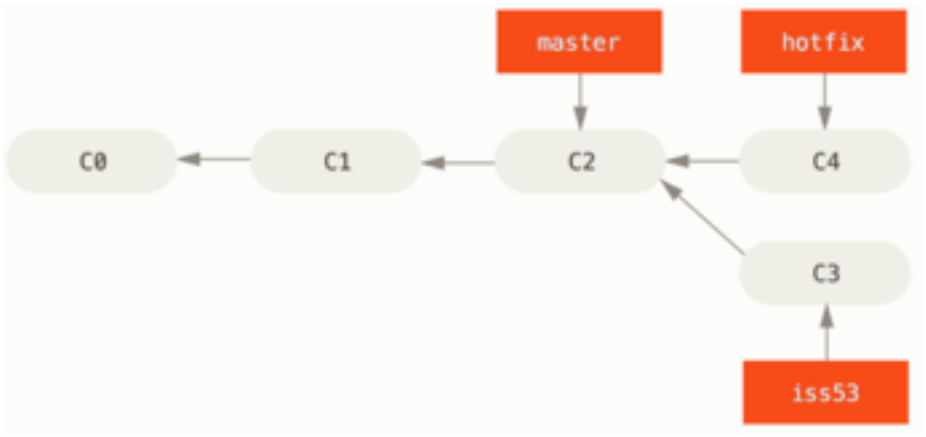
Merging

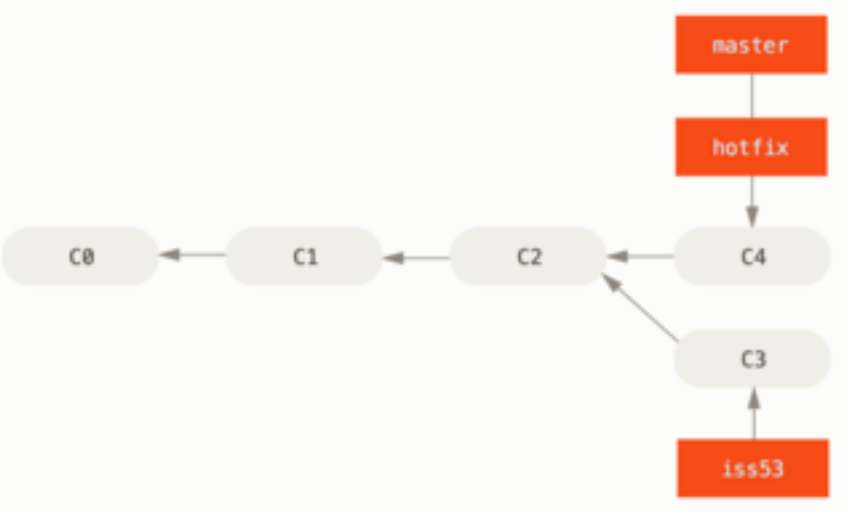
- A merge is an **action** you'll take to **combine the history** of two branches
- It's how you apply the changes you've made in one branch (ex: your bug fix, your test, your text edit) into another branch (ex: the main, master branch)
- `git merge <source branch>`
  - Applies the changes from <source branch> onto the current branch you're on
- It retains the history of both branches, but usually creates a new commit to represent the merging

Different merge results

- A merge can result in either:
  - No new commit, the target branch is updated cleanly, aka **fast-forward** merge
  - A new commit, representing the combining of histories (the merge), aka **3-way** merge
  - A **conflict**! The merge failed because git couldn't figure out how to resolve a difference between two files.
    - ex; The same line of the same file was edited in both branches

Fast-forward merge

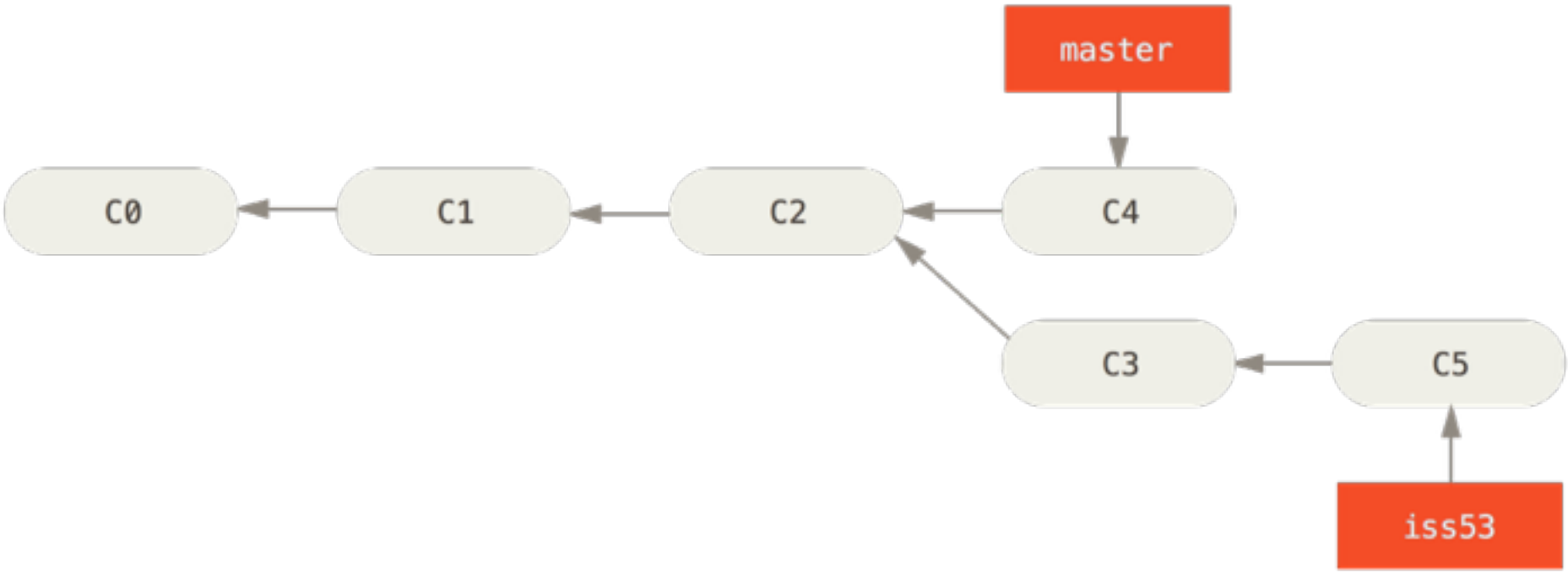




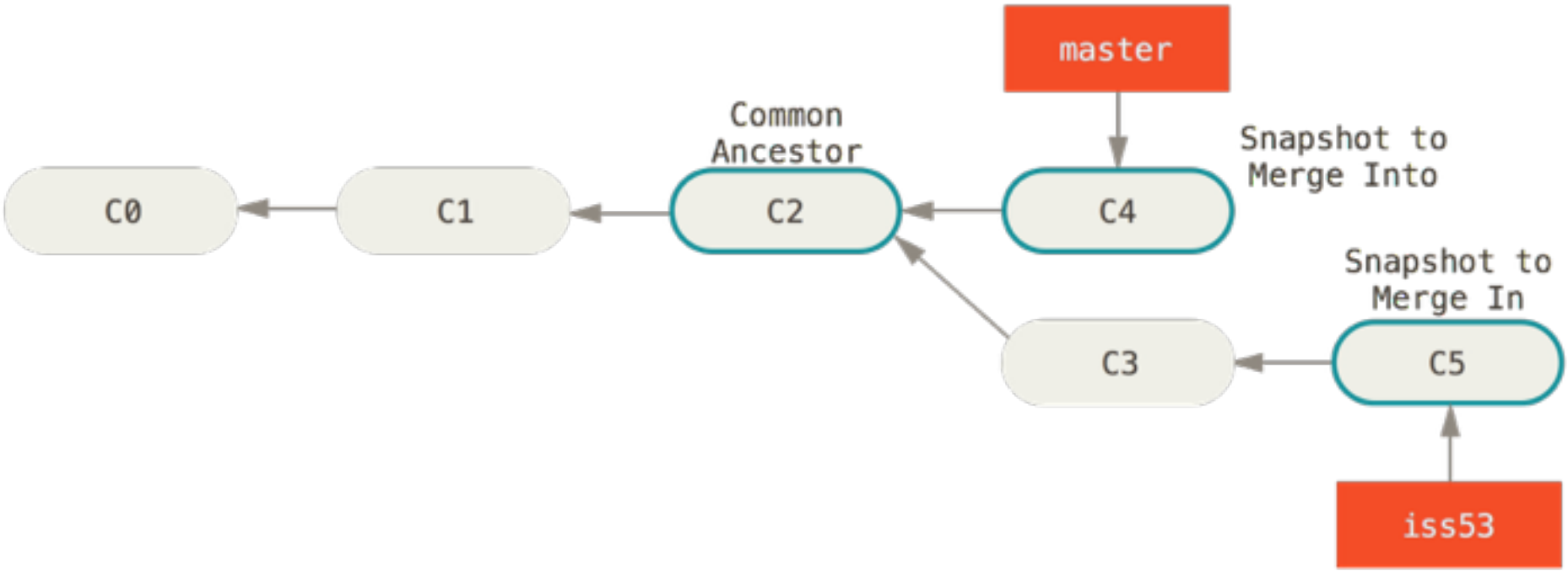
- When the history did not diverge
- Clean, streamlined history with no merge commit

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```

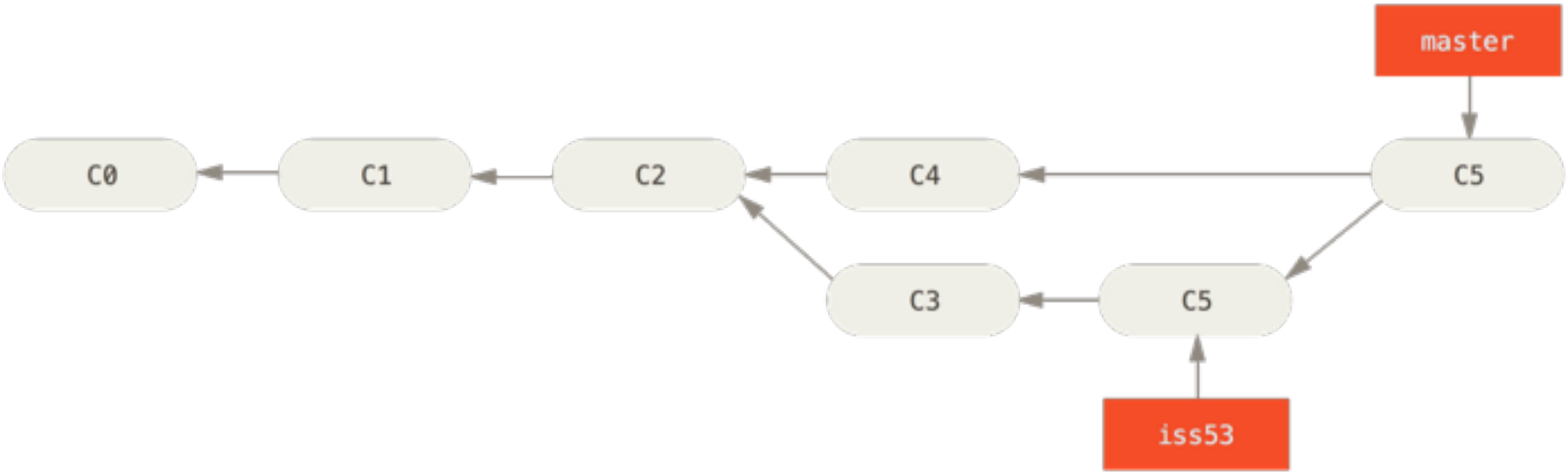
The three-way merge



Pre-merge


master
Common
Ancestor
Snapshot to
Merge Into
C0
C1
C2
C4
Snapshot to
Merge In
C3
C5
iss53

Post-merge



- History had diverged
- Merge commit needed to be created to represent the combined histories

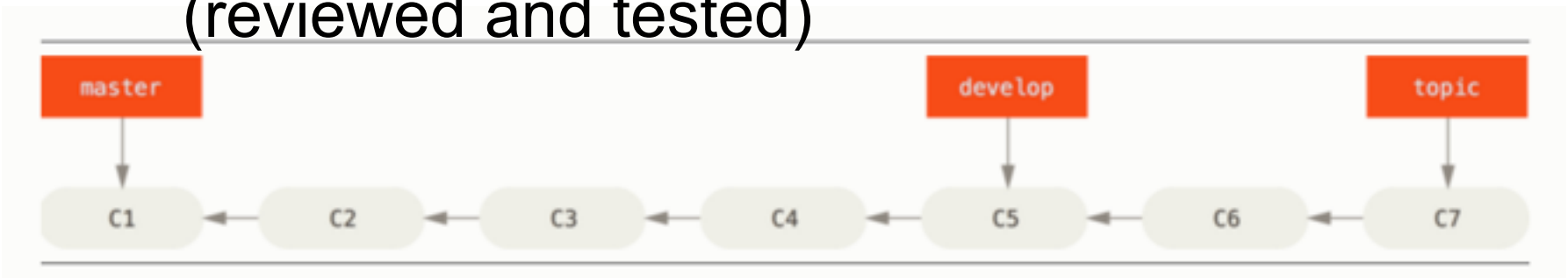
```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html |    1 +
1 file changed, 1 insertion(+)
```

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

- **Master** branch is typically considered stable, production-ready code
- Any time you want to make a new change, ex a bug fix or feature
  - Check out master
  - Create a new branch, based off master
  - Do your work in the new branch
  - Merge it back into master when ready (reviewed and tested)



Visualizing branches

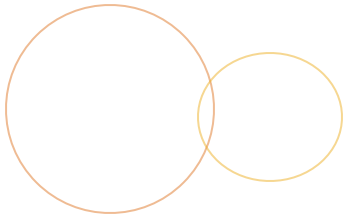
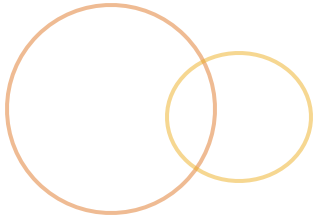
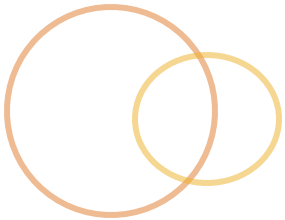
- http://pcottle.github.io/learnGitBranching/?NODEMO

Recap: Branching and Merging

- `git branch` – view, create, manage branches
- `git checkout` – switch between branches and commits
- `git merge` – 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”
- Review the log, `--oneline --graph --all`
  - Notice the difference between the merges?
- Delete your merged branches


module

MERGE ISSUES

Undoing a merge

- If you've just completed the merge and decided you didn't want to do that…
  - `git reset --hard ORIG_HEAD`

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 in limbo; it is not yet committed!
  - You must **resolve the conflict** or **abort the merge**
- Abort the merge during a conflict
  - `git merge --abort`

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

Merge tool

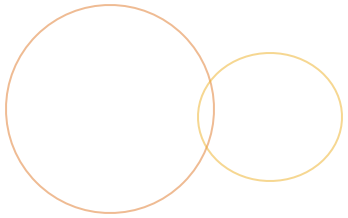
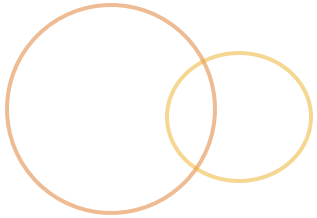
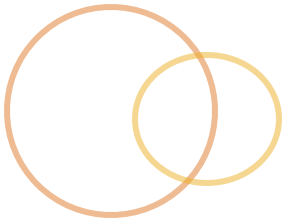
- Dealing with conflicts can be tedious, git allows you to set up a “mergetool” which makes it easier to view the conflict/diff and resolve it
- During a conflict…
  - `git mergetool`
  - Requires configuration
    - `git mergetool --tool-help`
    - `git mergetool -t <tool>`
    - `git config --global merge.tool <tool>`
- You’ll still need to add + commit after you resolve the conflict with your merge tool!

Recap: Merge issues

- We saw how to undo a merge either by using `git reset` if the merge was just performed
- We also got to see what a merge conflict is like and how to resolve it
  - abort it with `git merge --abort`
  - or resolve the conflict by hand or with `mergetool`

Lab: Resolving a conflict

- Let's create a conflict!
  - Create two branches off of master
    - Create first branch called "red"
      - Edit your favorite color (in your <name>.txt file) to be "Red!"
    - Go back to master
    - Create a second branch called "blue"
      - Edit your favorite color to be "No, Blue!"
- Checkout "master"
  - Merge "red"
  - Then merge "blue" -- you should get a conflict
- Abort or undo the merge! Just to get a feel for it
  - Then merge "blue" again
- Resolve the conflict


module

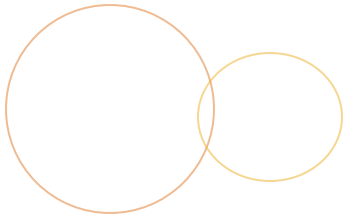
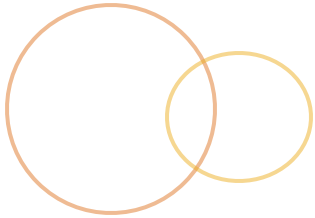
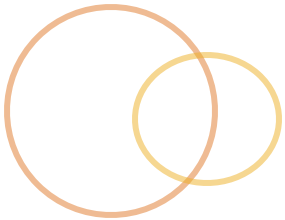
TAGS

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

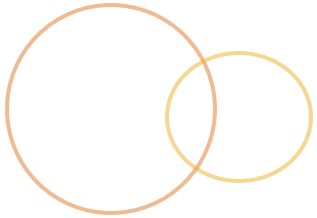
- Tag your current commit with a lightweight tag
  - `git tag current`
- List off your tags
  - `git tag`
- View info about your tag
  - `git show current`
- Check out the tag
  - `git checkout current`
- Check out master
- Try an annotated tag
  - `git tag -a v1.0`

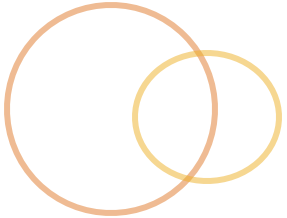





module

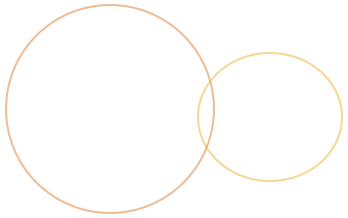
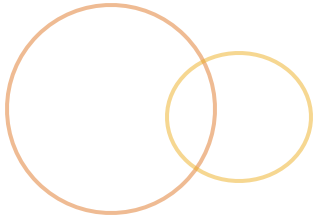
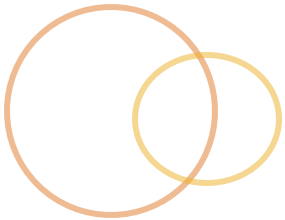
ALIASES

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'`


module

STASHING

Stashing

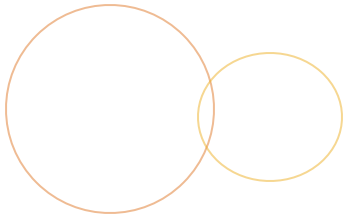
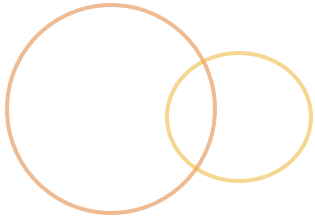
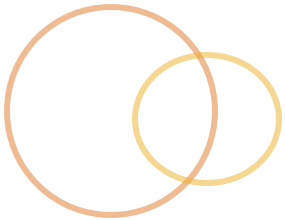
- Stashing is a way to quickly store work in progress without committing
  - Saves all staged & working directory changes
  - Clears the staging area
  - Clears the working directory
- Helpful for…
  - Quickly storing work you want to revisit
  - Moving work you didn't want on branch A to branch B
- 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`

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


module

IGNORES

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

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

review

DAY 2 REVIEW AND TEST!

Test: What does each command do?

Explain what each command does in context and how it affects the “three trees”

- git init
- touch me.html
- git commit
- git add me.html
- git status
- git reset me.html
- git checkout feature234
- git checkout
- git checkout -- profile.txt
- git checkout -b hotfix10
- git branch

Test: What does each command do?

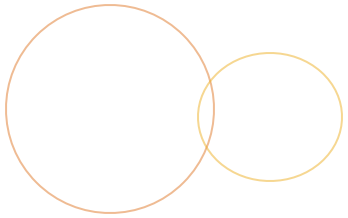
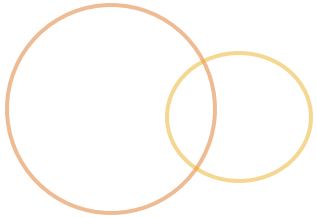
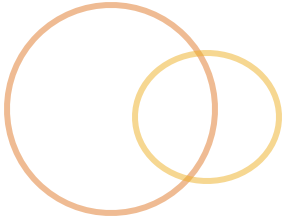
Explain what each command does in context and how it affects the “three trees”

- git branch ticket55
- git show ticket55
- git checkout 5d23eg
- git show 5d23eg
- git tag -a v1.5
- git rm debug.log
- git show HEAD
- git show HEAD^
- git show HEAD^^
- git reset HEAD^

Test

Let's map out a basic git graph and label the parts…

- Make three commits
- You're on… the master branch
- Make a "bug5" branch
- Where is HEAD
- Checkout bug5
- Where is HEAD
- Add a commit to bug5
- Where is HEAD, Master and Bug5?
- Can you undo it all and go back to the first commit?


module

REMOTES

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…
- Through hosted services and remotes we can
  - 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

Using Remotes

- Hosting our repository and sharing our work

I'll run through an example online

Lab: Hosting our repository

- *Using a hosting service and remotes to share our work*

- Go to github.com (or gitlab)
  - Make sure you have an account!
- Create a new repository
  - Call it "**about-me**"
  - Don't *initialize* it, we've already got an initialized repo
- Follow the directions they give or…
  - Copy the remote repository **url**
  - Then add a **remote** to our local git repository
    - `git remote add origin <remote-url>`
    - `git push --set-upstream origin master`
- You may be prompted for your credentials
- Now check out the repository on GitHub

What just happened there?

- We created an empty repository
- 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 hosted remote

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 **pulling** or **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>`

Fetch…

- **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
- `git fetch origin`

Pull..

- **Pull** is just a **fetch** and a **merge** in one
- `git pull origin master`
- `git pull --squash`

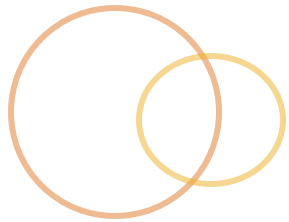
Push…

- **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
- `git push origin master`

Pushing 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`

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 push -u <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`

Lab: Sharing branches

- We're going to make a minor change in our repository and share it on the hosted remote
- In your local about-me repository
  - Create a new branch off master, call it "upper-name"
  - Edit your <name> file and change your name to uppercase (you can make any arbitrary edit you want)
  - Stage, commit
  - Push your branch to the remote
    - `git push origin upper-name`
- Go look at the branch that is now in your repository on github/lab
- In your local…
  - Make another change…. add your favorite noise?
  - Stage, commit
  - Push
- In your local…
  - Create a new branch *off master* and push it up to the origin
- **[adv]** Did you create them as tracking branches?

This is a typical workflow…

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`
  - `git push origin --delete <branch>`

Pruning remote (branches)

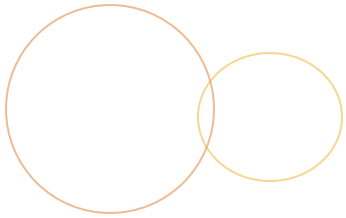
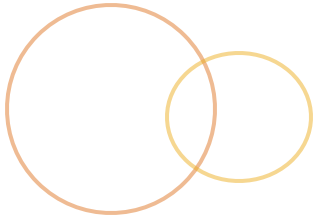
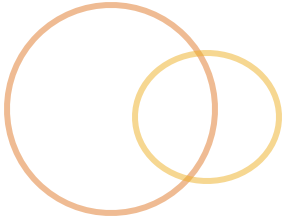
- There are potentially 3 versions of every remote branch
  - The actual branch on the remote repo
  - Your snapshot of that branch locally (in refs/remotes)
  - And a local branch that may be tracking the remote
- `git prune`
  - Removes references to remote branches that do not exist on the remote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

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

Recap

- Working locally
  - **Branch** off master
  - Work work work (**add**, **commit**, add, commit…)
  - **Merge** into master*
    - *Unless you're using merge 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


step-by-step

REMOTE VS LOCAL

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	Origin (server)
master	master
topic-1	topic-1
origin/master	
origin/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	Origin (server)
master	master
topic-1	topic-1
origin/master	topic-20
origin/topic-1	
origin/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

WORKING ONLINE

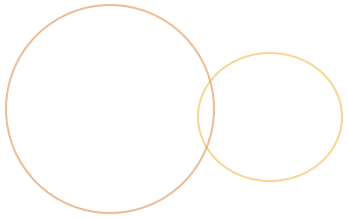
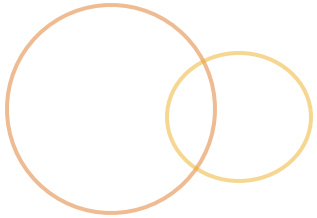
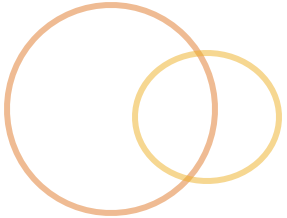
Hosting our repository

- Lot's of options
  - GitHub.com
  - GitLab.com
- A place to manage a project, collection of repositories
- Issue tracker integrated with code review and merging (when approved)

Let's check out the interface and the basics of what we can do in the service of our choice

Recap so far

- <service> 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)
- **Merge requests** are essential for bringing changes into your project
- You can use pull/merge requests to **submit patches** to other projects, too


module

MERGE/PULL REQUESTS

Lab: Join the team!

- How we get our team together depends on where it will be hosted, GitLab, GitHub, etc…
- Make sure you've signed up, share your username with me:
  - mr.morris@gmail.com
- Or… open an issue with my project

Merge/Pull Requests

- GitHub coined “pull requests”, GitLab uses “merge request”
  - “Please merge my branch into your branch”
- Works well with many different workflows
- Encourages early collaboration

Using issues and working online

- We can do a lot from the interface
  - Create branches
  - Edit files
  - Submit patch requests
  - Merge branches
- Let’s use the issue tracker and the interface to submit some changes

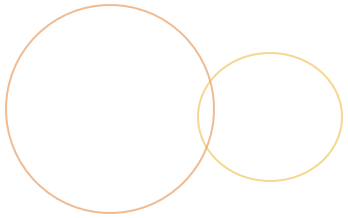
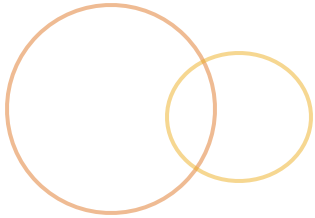
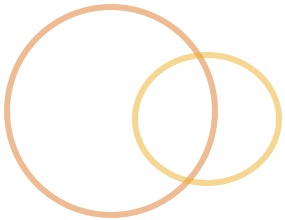
Lab: Working online (only)

*Note: This is all to be done within GitLab on the **about-us** repo (no local repository work yet)*

- **Create an issue(s) for us all to resolve**
- Resolve the issue by creating a merge request
  - Create a new branch, off of master
  - You're going to create a new file, your profile page, **<yourname>.html**
    - Add some basic info about you
  - Commit to your branch
  - Create a merge request, asking to merge your branch into `master`
- Relate your merge request to the issue
  - Reference the Issue number
    - Ex: "Resolves #5231"
- We'll go through and review (and merge) our merge requests together

Recap so far

- So far we've been working as a team within a single repository, entirely within GitHub/Lab
- 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/Lab
- Ultimately, upon **approval**, our work was **merged**


module

CLONING

Taking it local

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

Cloning

- `git clone <remote-url>`
  - Copies an existing repository
  - `git clone <remote-url> <dir>`
- What does this do exactly?
  - Initializes 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”

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>`

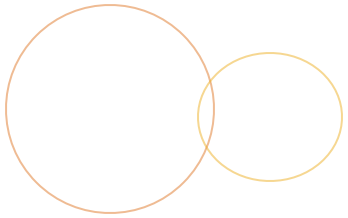
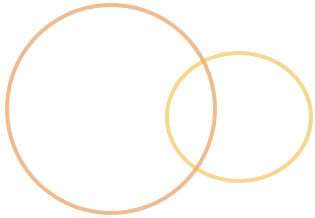
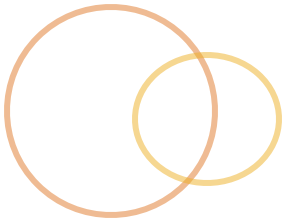
Lab: Cloning

- Start in a fresh directory - not in a previous repository!
- Clone the **about-us** repository
  - `git clone <git:url>`
- 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 GitLab, open a **merge request** to have your new branch merged into the master branch

Stop here

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?


module

FORKING

Forks

- You can Fork any public project in GitHub/Lab
  - 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/merge requests into the main, **upstream**, project
  - Project owner can review and merge

Lab: Forking

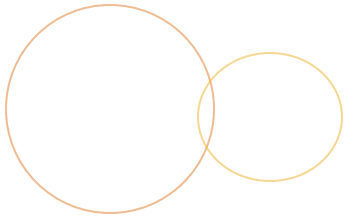
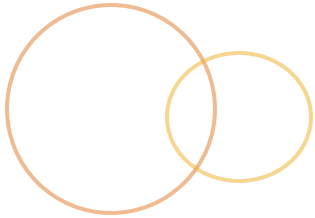
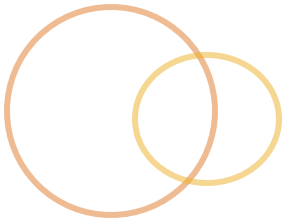
- Now we're going to work within our own forks…
  - Each of us, as developers, will have our own fork and will push our work to that fork then submit Pull Requests to the main repository.
- Fork my **about-us** repository
- Clone **your fork** to your local via git
  - Clone your fork into a new directory ("about-us-fork"?)
    - `git clone <your-fork-url> ./`
  - 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
  - Submit a merge 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/Lab
  - Merge it from GitHub/Lab

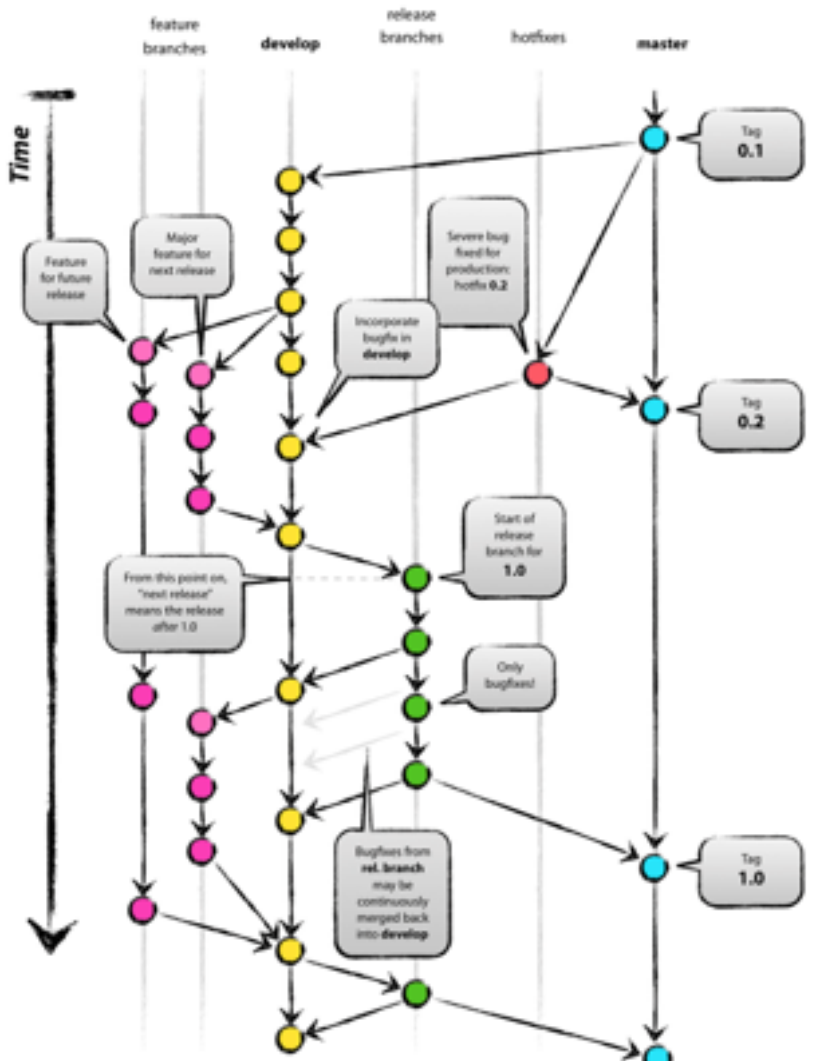

module

WORKFLOWS

Organizing

- How you organize your remotes and branches determine your workflow
- Consider
  - Private/public
  - Single remote, many remotes (forking)
  - Branch names
  - Branch stability
- Branching approaches
  - `master or main or production`
  - `master (stable) + develop (new work)`
  - `master + develop + hotfixes`
  - `master + integration + staging`

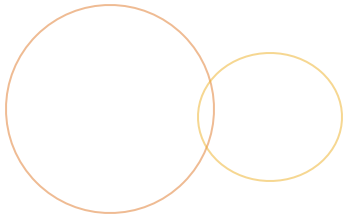
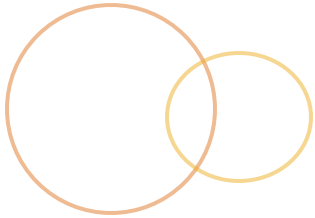
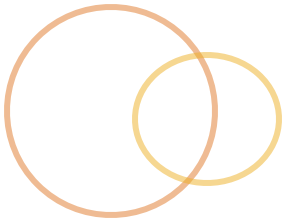
“Git Flow”



- 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


module

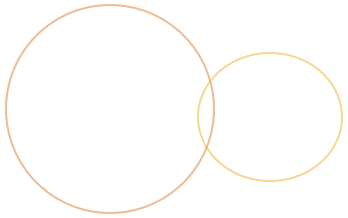
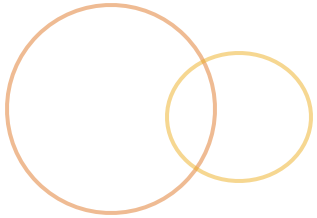
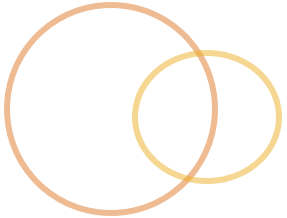
GIT ODDS AND ENDS

GitK

- 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

Git GUI

- 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)


module

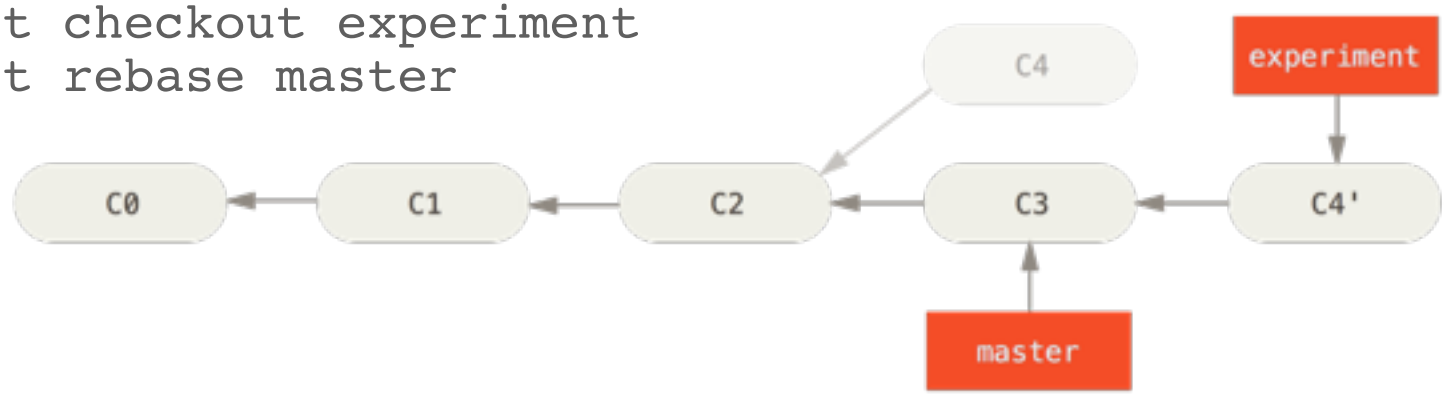
MANAGING HISTORY

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
    ```

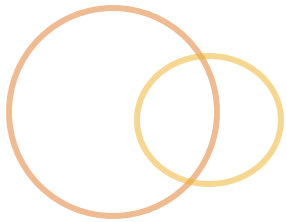
Rebasing

- `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`
    Or…
  - `git pull --rebase`
- 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?

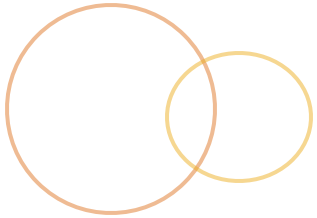
Recap: Altering History

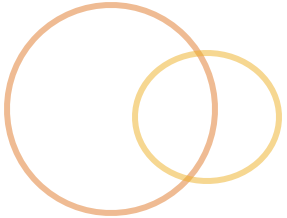
- **Rebase** is a powerful (and dangerous) alternative to merging
- Keep a clean history by avoiding merge commits, and **squashing** messy work
- But **merge** still has a place to maintain a meaningful history
- **Cherry-picking** is good at grabbing one (or a couple) commits from one branch into another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

Lab: Changing history

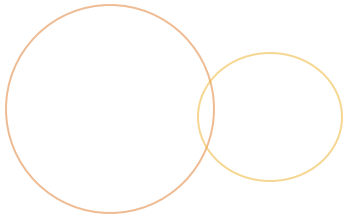
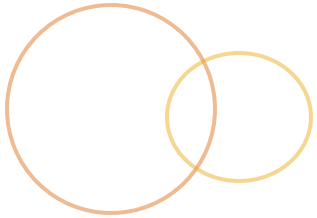
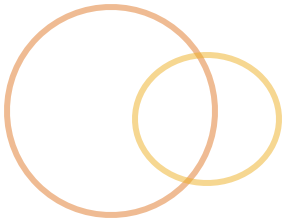
- Clone my repository:
  - https://github.com/rm-training/history-changer
- Check the graph (`--all`)
- 1) Using rebase to bring in updates
  - Update the topic-behind-1 branch with changes from master by using `rebase`
    - Note: the branches are all remote… you'll need to branch them locally to work 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

ReReReRe





- 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


module

COMPARING BRANCHES

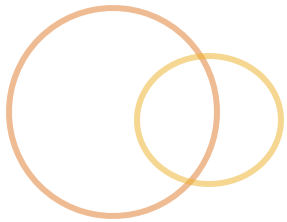
Comparing branches with log

- 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>`

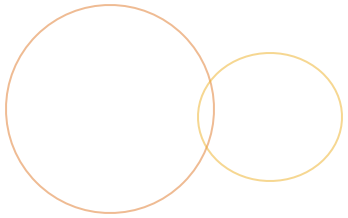
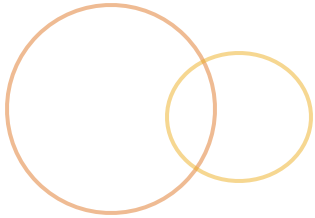
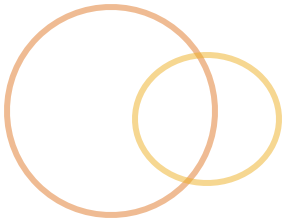
Recap: Comparing



- 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

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
  - `--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

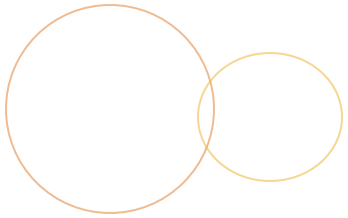
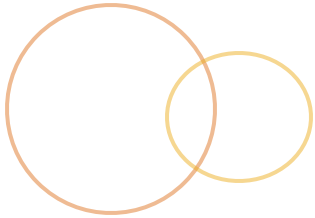
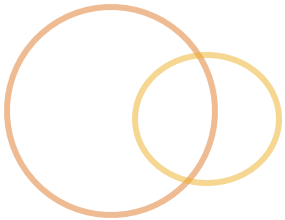
- Annotates file with commit information for each line
  - `git blame <filename>`
- Limit the output
  - `git blame -L 12,22 <filename>`
- Track code movement
  - `git blame -C <filename>`

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

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~`
    - 1st parent of HEAD
    - same as `HEAD^`
  - `git show HEAD~2`
    - 1st parent of the 1st 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)

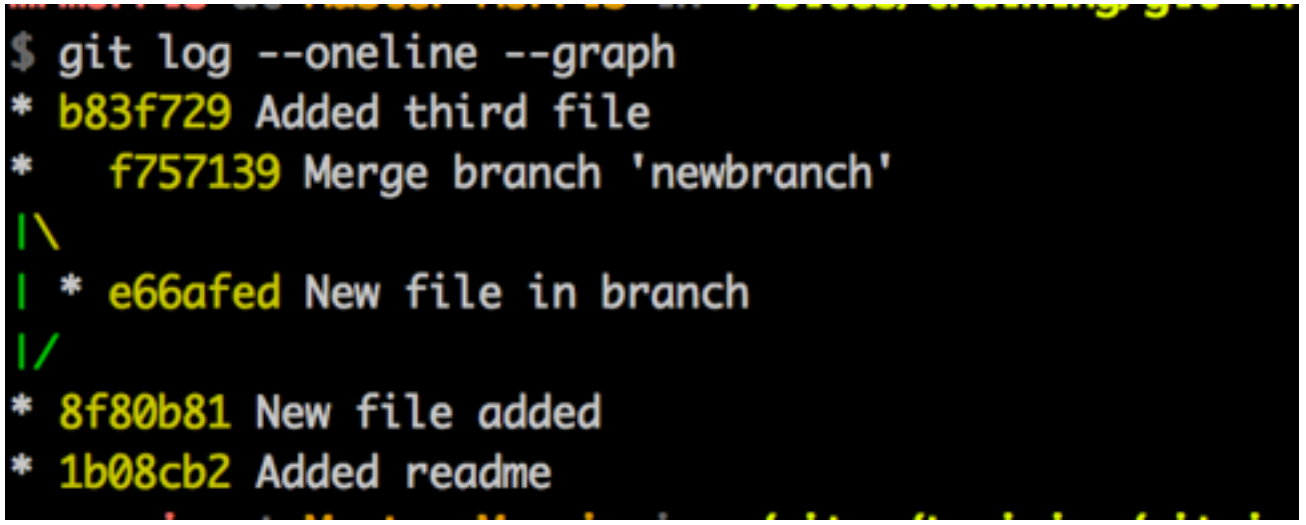
- Using log, reflog, reset and checkout, we can fix a lot of problems we may find ourselves in.
- Find a lost commit
- Find a lost branch
- Undo a merge
- Undo a rebase

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 brain is the mainline to revert
    - 1 is right-most, 2 is next to the left



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
    ```

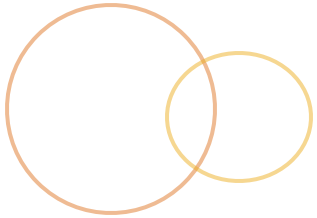
Recap: Fixing issues in git

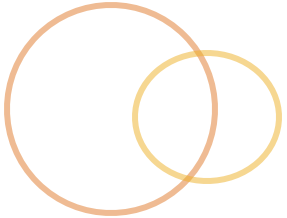
- We witnessed the power of the reflog, which allows us to track our local history of actions and branch changes
- Using the reflog (and log) we can take control of our branches, and fix a lot of snafus, with git reset and git revert
- Hopefully we have a good understanding of what HEAD is
- And we know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

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`

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