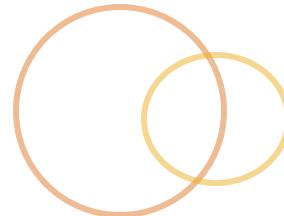
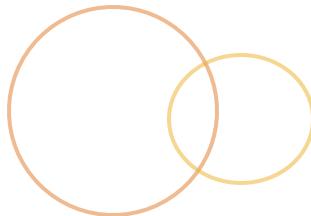
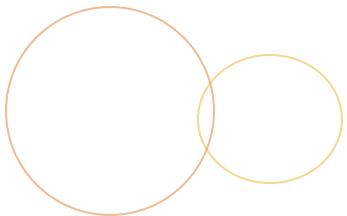




Git Fundamentals

Ryan Morris

mr.morris@gmail.com



module

INTRODUCTION

Git Fundamentals

Ryan Morris

mr.morris@gmail.com

Set up...

- Install Git CLI
 - <https://git-scm.org>
- Prove it
 - \$ git --version
git version 2.13.3
- Sign up at [github.com / gitlab](https://github.com)
 - Email me your username
- Download the Slides
 - <http://github.com/rm-training/resources>
- Grab a cheat sheet
 - [Github's](#)
 - [Atlassian's](#)

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 MEMORIZIZE 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.



Let's get started

● Introductions

- Me: Ryan Morris

- You?

- What is your experience with Version Control & Git?

● Class Style

- Informal

- Lecture & Labs

- One day intensive!

Plan for the next two days

Day 1

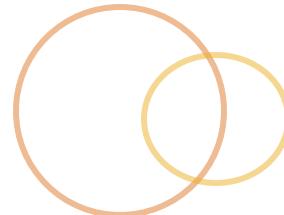
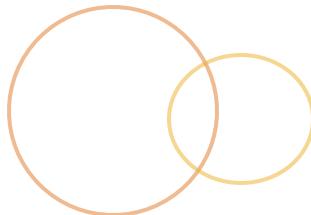
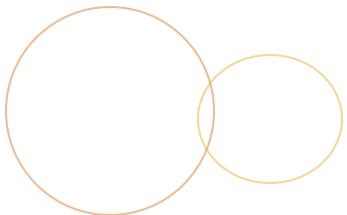
- Creating & Managing repositories
- Basic workflow of Git
 - Stage, commit
 - Branch, Merge
- Fixing Stuff
 - Undoing
 - Dealing with conflicts
- Odds & Ends
 - Stashing, tagging, etc...

Day 2

- Collaborating through Remotes
 - Staying up to date
 - Sharing your work
- Workflows
- Managing History
 - Rebase, Cherry-pick
- Reset

Resources!

- The Git Parable
 - <http://bit.ly/1isB3K4>
- Pro Git, 2nd edition (for free!)
 - <http://git-scm.com/documentation>
- Think like a Git
 - <http://think-like-a-git.net/>
- Visualizing Git
 - <http://pcottle.github.io/learnGitBranching>



module

VERSION CONTROL HISTORY

Why version control?



Why Version Control?

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

What things can we version control?



What things can we version control?

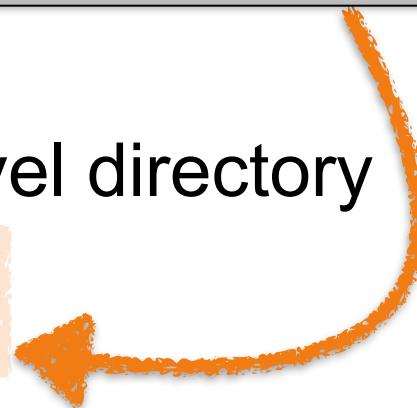
○ Any file!

- The code for our website
- My photo collection
- My mp3 collection
- My entire computer
- These slides

All files & folders within this directory will be part of my project

○ Our **project root** will be a top-level directory

/My Documents/projects/web-scrawler/



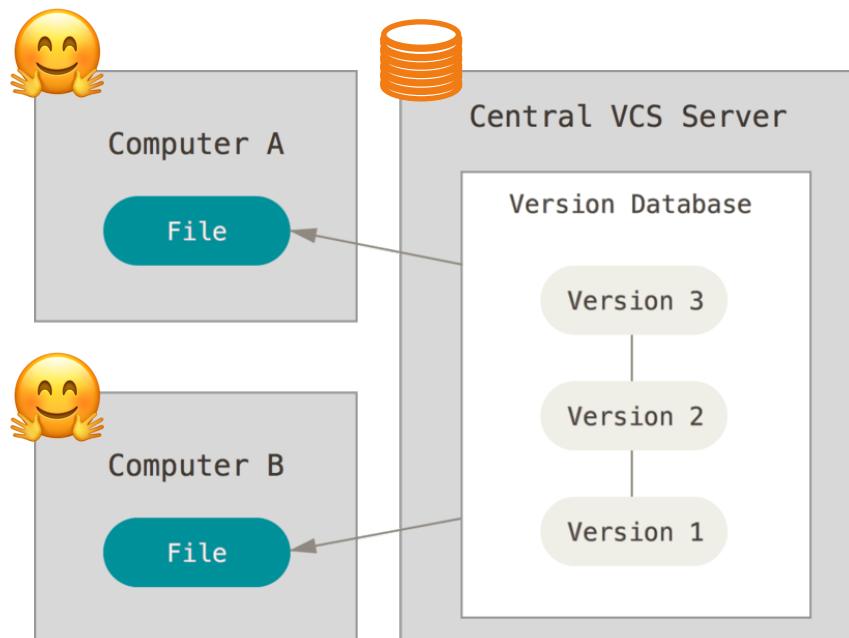
Stone-age version control



just me!

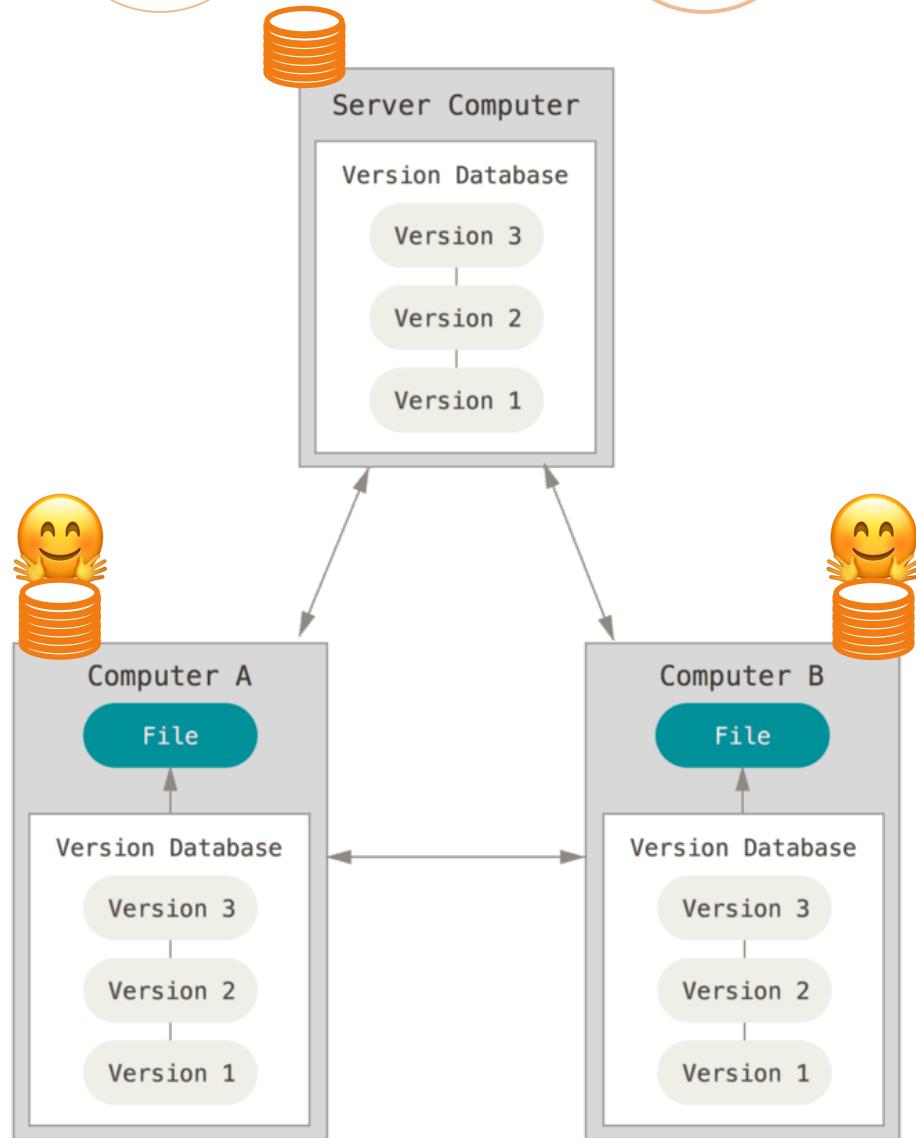
- A new file for each meaningful change
 - /my-files/myfile.v1
 - /my-files/myfile.v2
 - /my-files/myfile.v3.draft
- Advantages
 - Easy
- Disadvantages
 - Error prone
 - Single point of failure

Old World: Centralized Version Control



- ◉ 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

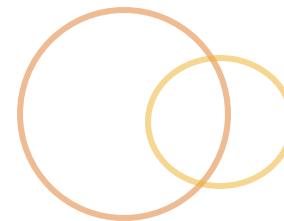
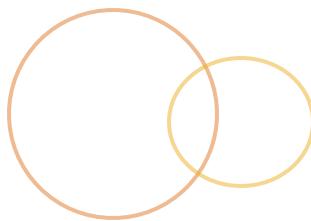
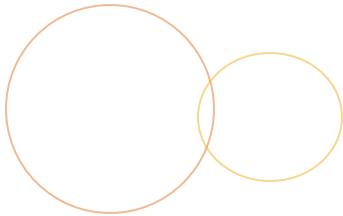
History

Linux (1991-)

- No real version control from 1991-2002
- **bitkeeper** from 2002-2005
- created **git** in 2005 after relationship w/ bitkeeper broke down

Goals

- speed
- simplicity
- support for non-linear development
- distributed
- support for large projects



module

UNDERSTANDING GIT

Git was made for

- Speed
- Large projects
- Large, distributed teams
- Frequent changes
- Lots of branching

Starts with a project root

/my-project/

./bin/

start.sh

./vendors/

jquery.js

datepicker.js

index.html

about.html

Each git project will track
A top-level directory (the root)



Nearly every operation is local

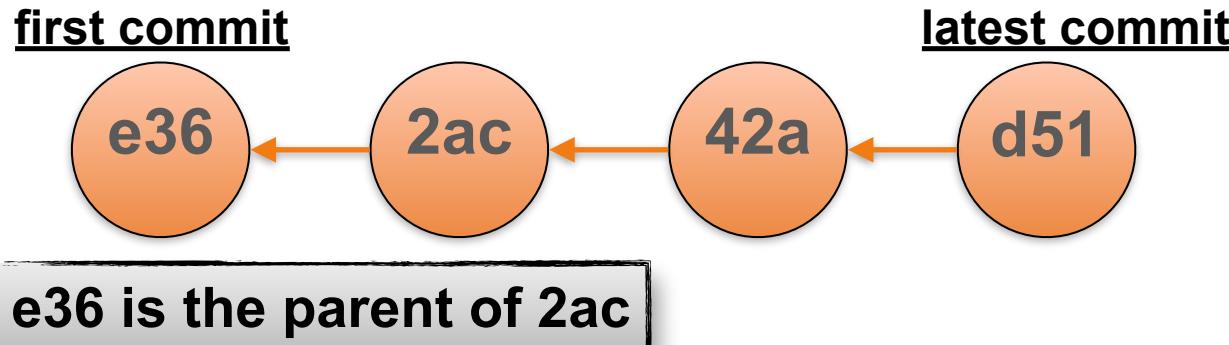
- Entire repository is available **locally**
 - Full history
 - All branches
- You can work offline!
 - Create/Check out branches
 - Commit changes
- You *do* need a network connection to collaborate

~~Versions => Commits~~

- Commits represent a "version"
 - No sequential identifiers
 - Identifier is a hash (sha1 or sha256)

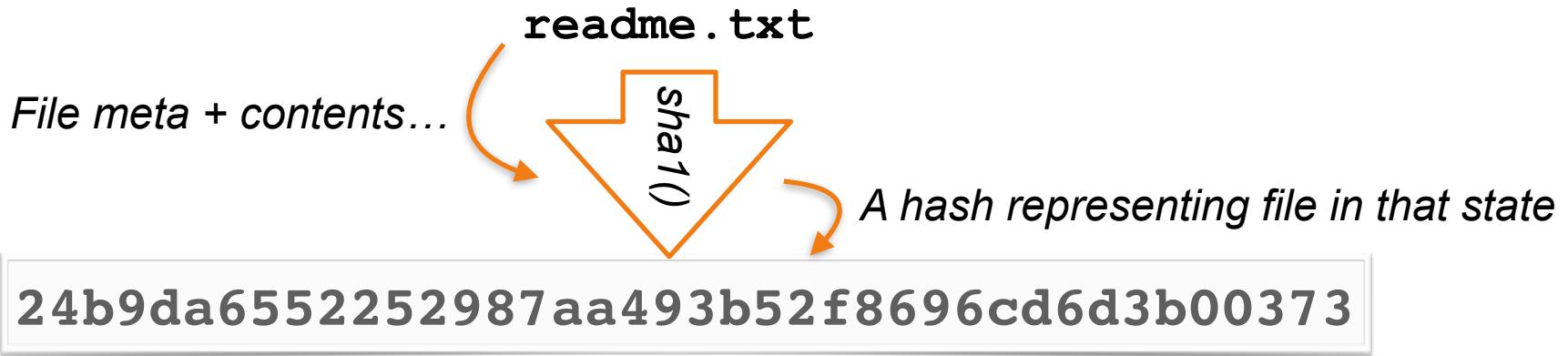
```
d842916416b7490e9b1b67b4a25748b1612465e8
```

- Each **commit** represents a "snapshot"
- Commits point to their parent commits



Hash all the things

- All *things* in git are represented by unique hashes
 - Files, directories, commits, tags...



- Changes are tracked in this way
- Tampering becomes obvious

Everything is hashed

/my-project/

./bin/

start.sh

./vendors/

jquery.js

datepicker.js

index.html

about.html

=> 5748b16

=> d84291

bin tree listing

=> 0dfb235

=> f8eb37f

=> d3f8321

vendors tree listing

=> 9a490c8

=> 0b992ca

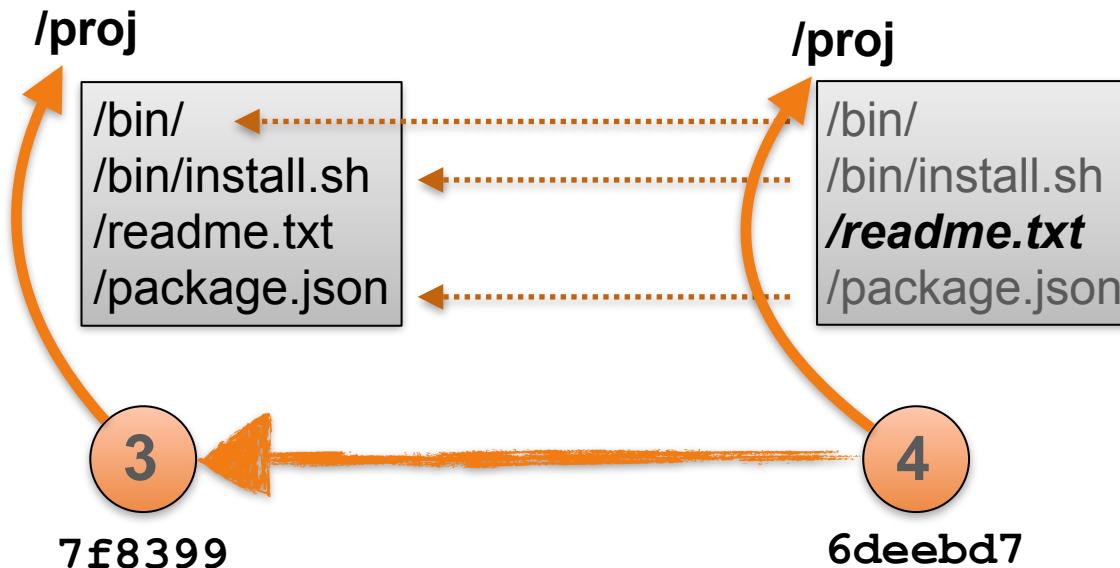
=> 0ed0cbc

=> 72e3c4d

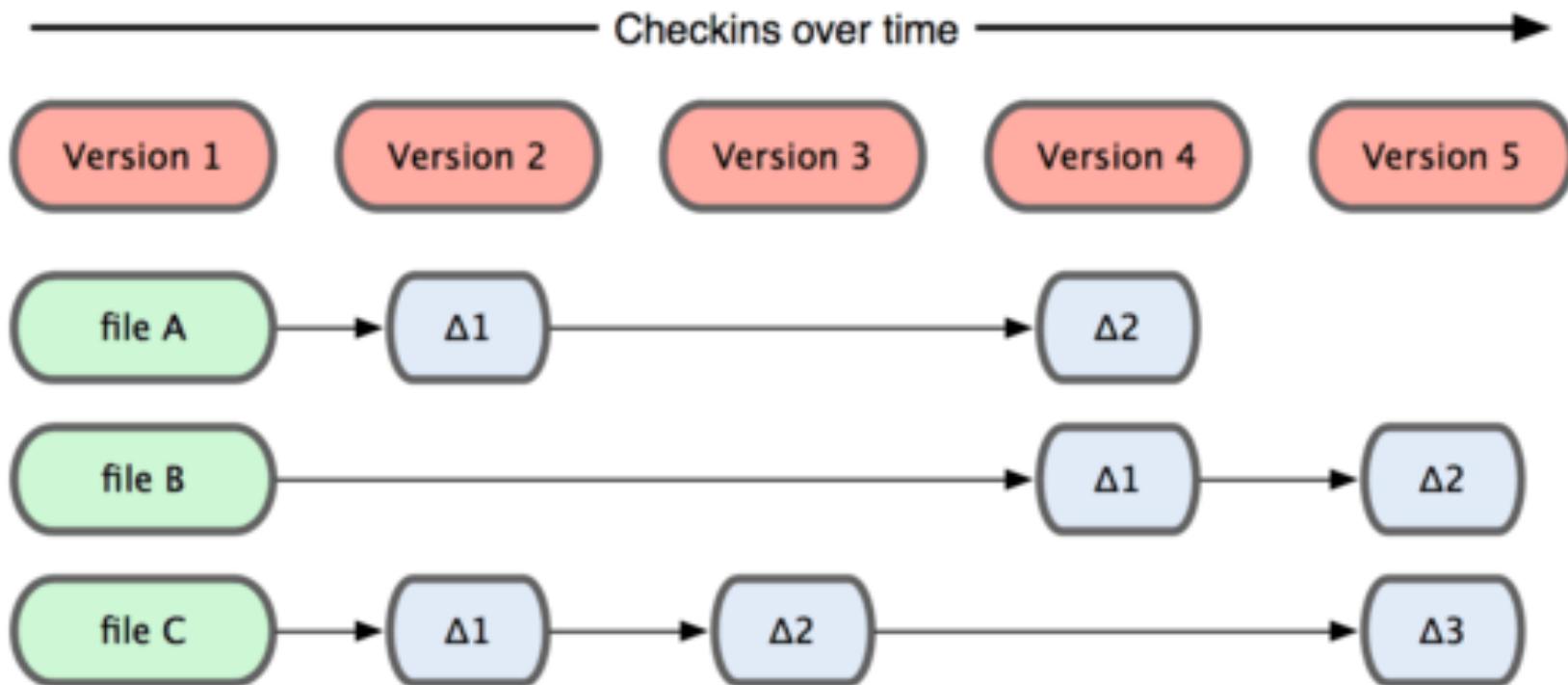
=> 4ba6045

Snapshots, not Deltas

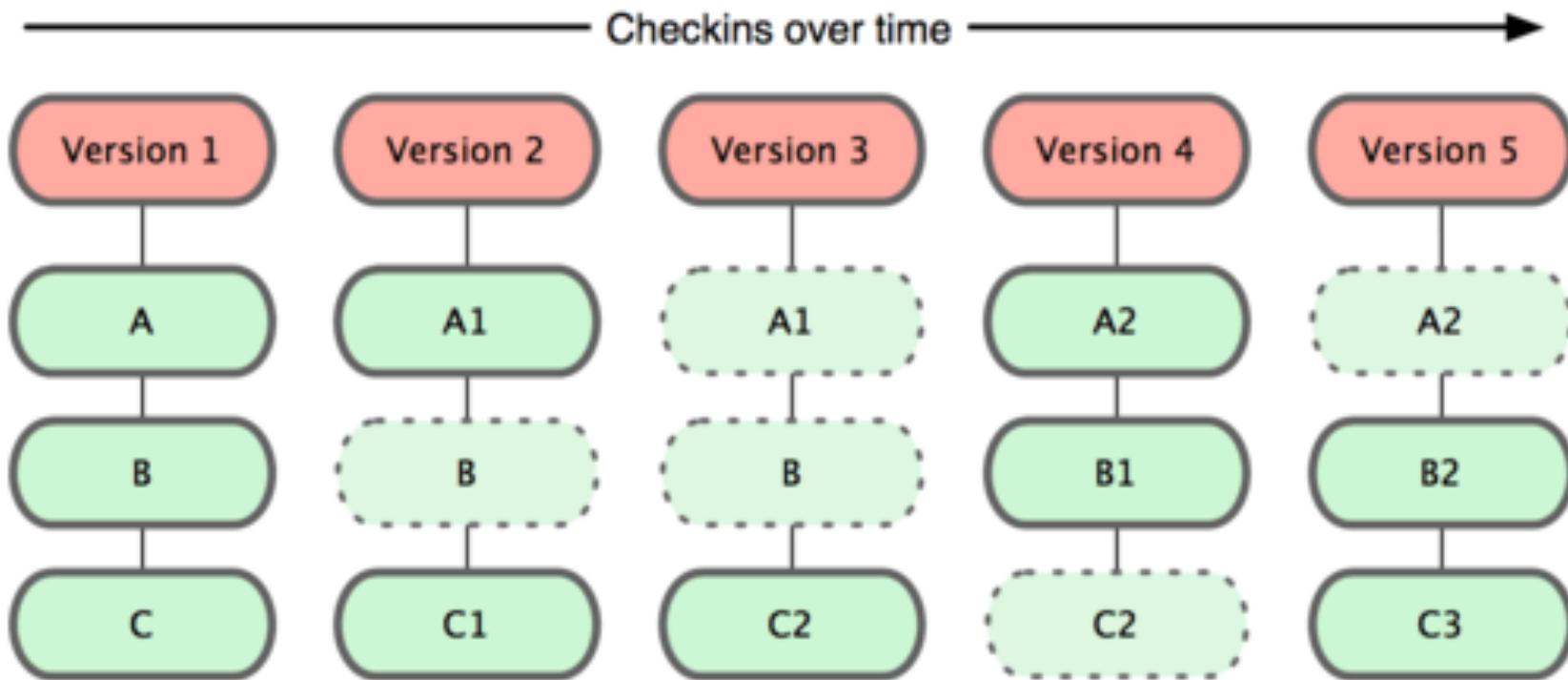
- A commit references *copies* of each file at the point in time the commit is made
- However, git will not keep copies of files that did not change



Diffs / Deltas (SVN, etc)



Snapshots (Git)



Loose vs Packed

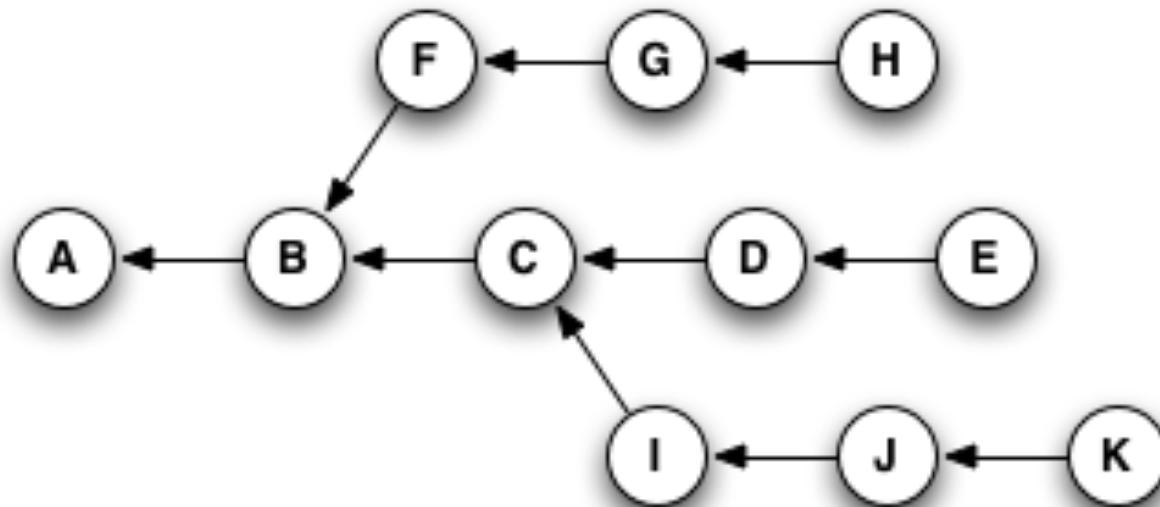
- ◉ One-liner change to a 1GB `readme.txt` file?
 - ◉ You now have two 1GB txt files in your repository!
- ◉ Git will clean things up...
 - ◉ combine similar **files** across commits into diffs
 - ◉ compresses **objects** and diffs
 - ◉ prunes unreachable **objects**

Git is defensive

- ◉ Nearly all actions **add** data to the Git database
 - ◉ Once something is committed it is part of the project history *forever*
- ◉ It's easy to recover from mistakes
 - ◉ Lost commits and deleted branches can *typically* be recovered
- ◉ However, git *does* garbage collect dangling commits

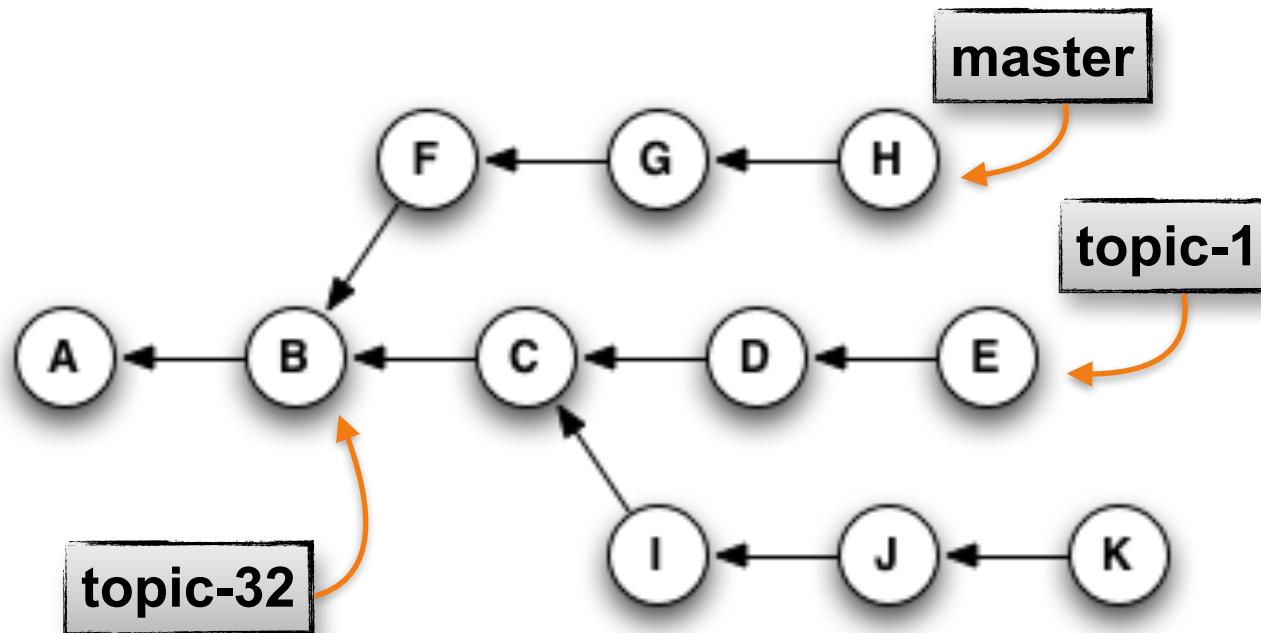
Git is building a Graph

Commits are the nodes



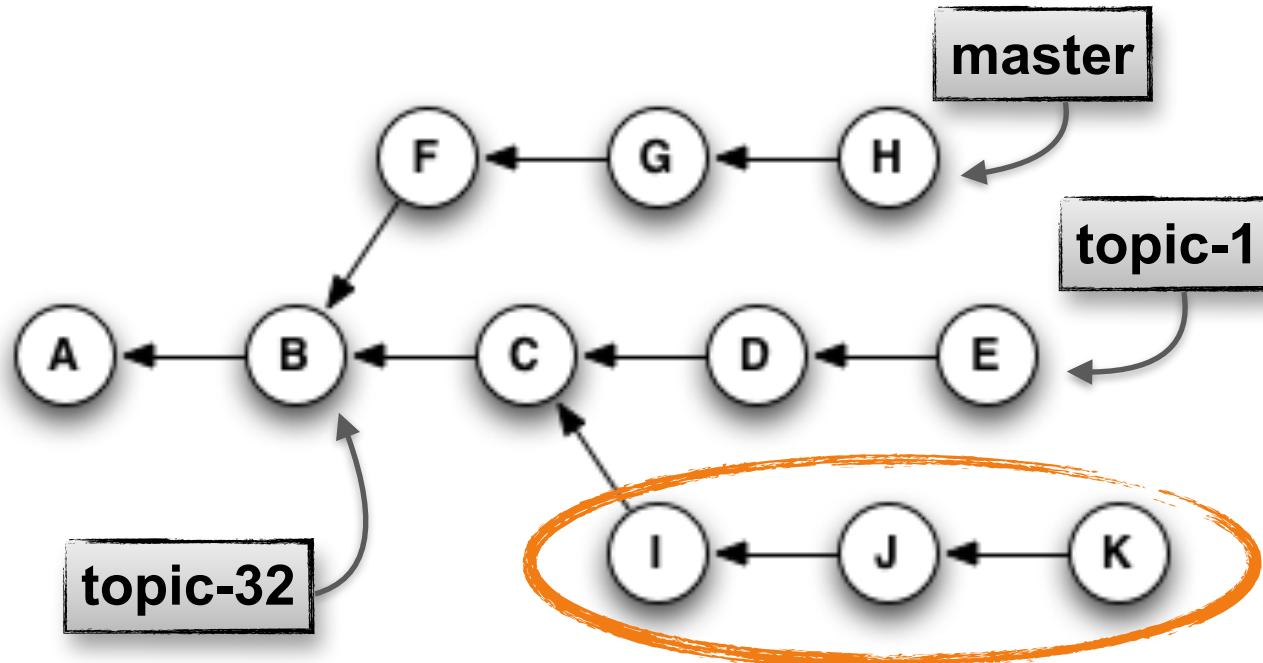
Git is building a Graph

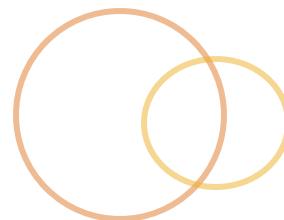
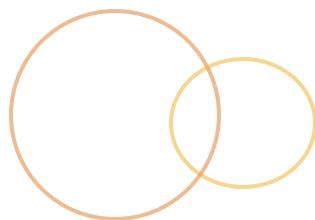
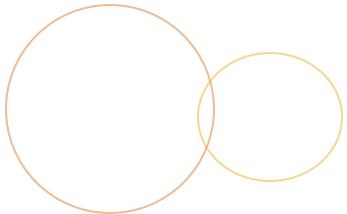
- Commits are the nodes
- We'll label special commits



Git is building a Graph

- Commits are the nodes
- We'll label special commits
- Unreachable stuff is still part of our graph





THE COMMAND LINE

Command line basics

◉ *How does everyone feel about using the cli?*

◉ *The Art of the Command Line*

◉ <https://github.com/jlevy/the-art-of-command-line/blob/master/README.md>

Core commands

cd <directory name>

change directory

cd ..

go back a directory

touch <filename>

create a file

mkdir <directory name>

create a directory

ls

list files in this directory

ls -la

list all files in this dir

rm <filename>

remove a file

rm -Rf <directory name>

remove a directory

clear

clear my console

pwd

what is the present dir

cat <filename>

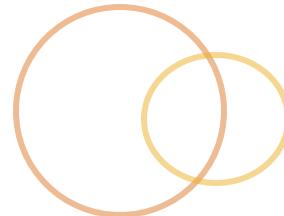
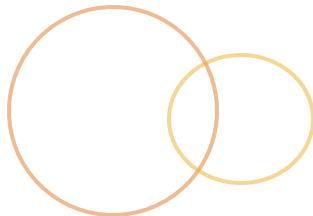
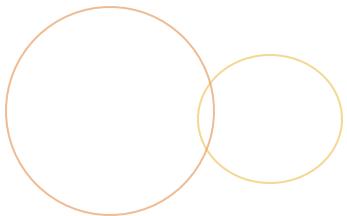
view file contents

open <filename>

open file with the main app

Dress up that CLI

- ◉ My command line prompt is formatted
- ◉ Yours may not be, don't be sad
- ◉ Options:
 - ◉ Grab one of many shell prompt formatters
 - ◉ Use a different shell like Zsh
 - ◉ Oh My Zsh
- ◉ All commands are the same



module

SETUP

Installed & Configured

Windows installs
Git bash and
Git CMD

- Install it: <http://git-scm.com/download/>

- Configure it

```
git config --global user.name "Ryan Morris"  
git config --global user.email "ryan@mnos.org"
```

- Don't want to affect all repos?

```
cd some/repo/directory  
git config --local user.name "Ryan Morris"
```

- View a key

```
git config --global user.name
```

Common Editor Settings

Emacs

```
git config --global core.editor emacs
```

Sublime (*requires "subl" cli installation*)

```
git config --global core.editor "subl -n -w"
```

Text Wrangler

```
git config --global core.editor "edit -w"
```

VS Code

```
git config --global core.editor "code --wait"
```

Windows Notepad

```
git config --global core.editor notepad
```

You might already have...

- Your name & email
- GPG Signing enabled
 - user.signingkey
 - gpg.program
 - commit.gpgsign
- A proxy defined
 - http.proxy
- Auto squash on pull
- pull.rebase

Set this to false
per repo to disable



Levels of Git Config

--system

/etc/gitconfig

Affects all users

Win:

in your git install
mingw32\etc

--global

~/.gitconfig

*Affects all repositories
for a single user*

Win:

C:\Users\your-user\

--local

per repository

myrepo/.git/config

Affects a single repo

*Overrides any other config
settings*

having trouble finding it?
git config --edit --local

Handy configs

- core.editor (vi, emacs, notepad, etc)

```
git config --global core.editor vi
```

- core.pager (less, more, ")
- push.followTags (true, false)
- pull.rebase (true, false)
- fetch.prune (true, false)
- And many more...

- Need to unset a key while experimenting?

```
git config --global --unset core.editor
```

Common Editor Settings

Emacs

```
git config --global core.editor emacs
```

Sublime (*requires "subl" cli installation*)

```
git config --global core.editor "subl -n -w"
```

Text Wrangler

```
git config --global core.editor "edit -w"
```

vs Code

```
git config --global core.editor "code --wait"
```

Windows Notepad

```
git config --global core.editor notepad
```

You might already have...

- Your name & email
- GPG Signing enabled
 - user.signingkey
 - gpg.program
 - commit.gpgsign
- A proxy defined
 - http.proxy
- Auto squash on pull
- pull.rebase

Set this to false
per repo to disable



.gitattributes

- Configuration file in a directory or subdirectory
 - Specifies behaviors for files/paths
- Such as dealing with binary files

```
# tell git to consider *.pbxproj as binaries
*.pbxproj binary

# tell git to use the "word" filter to diff
# this requires additional set up...
*.docx diff=word

# show exif data when diffing images
*.png diff=exif
```

Common Editor Settings

Emacs

```
git config --global core.editor emacs
```

Sublime (*requires "subl" cli installation*)

```
git config --global core.editor "subl -n -w"
```

Text Wrangler

```
git config --global core.editor "edit -w"
```

VS Code

```
git config --global core.editor "code --wait"
```

Windows Notepad

```
git config --global core.editor notepad
```

You might already have...

- Your name & email
- GPG Signing enabled
 - user.signingkey
 - gpg.program
 - commit.gpgsign
- A proxy defined
 - http.proxy
- Auto squash on pull
- pull.rebase

Set this to false
per repo to disable



Lab: Set up checkpoint

- Make sure git is installed

 - `git --version`

- Set up your identity

 - `git config --global user.name "your name"`
 - `git config --global user.email me@gmail.com`

- Optionally set your text editor

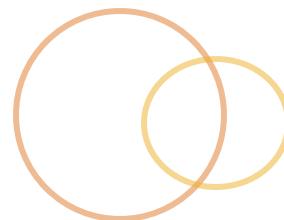
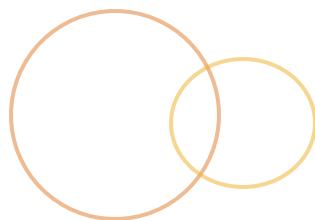
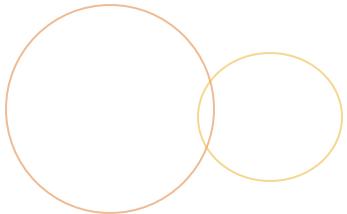
 - `git config --global core.editor emacs`
 - `git config -e`

- Double-check your configurations

 - `git config --list`
 - `git config <key>`

- Inspect the config file(s)

 - `cat ~/.gitconfig`



module

REPO CREATION

We'll learn about...

- **Creating** repositories with `init`
- **Tracked** vs **Untracked**
- **Staging & Committing** files changes
 - The typical workflow
- **The Git Database**

I'll run through creating a repo, inspecting, staging and creating commits

Creating a repository

● Initialize a repository

```
git init <directory>
```

● More examples

```
git init ./  
git init my-repo  
git init any-folder-name
```

Inspecting our repository

Check the status and the log

```
git status  
git log
```

Check what git has initialized

```
ls -la  
ls -la .git
```

Files are considered...

untracked

- A file your repository hasn't seen before
- The repository (git) is not going to touch it unless you "add/stage" it.

tracked

- A file your repository knows about
- Changes will be noticed
- Once you **stage** it... the file is *tracked*

Stage & Commit Workflow

- 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
```

Why Stage?

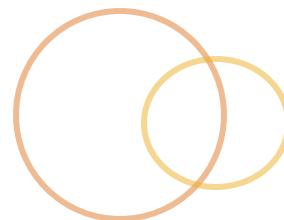
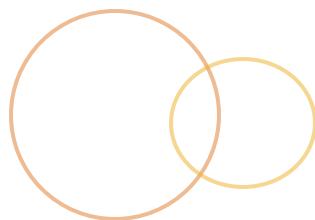
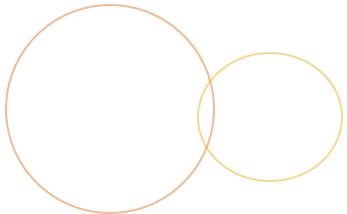
- Be thoughtful about your commit
 - Logically break up commits
 - Tell a story of your changes
- You can selectively prepare a commit
 - Don't commit everything at once
 - You can specify lines to commit

Lab: Create your first repo

- Initialize a repository called "first-repo"
 - `git init first-repo`
- Check the status and log
 - `git status`
 - `git log`
- Find the git database
 - `ls -la .git`
- Create a file with an editor or...
 - `echo "Hello World" > hello.txt`
- Stage the file change
 - `git add hello.txt`
- Check the status
 - `git status`
- Commit the change
 - `git commit`
- Check the log
 - `git log`

**Don't forget to
cd into the new
directory after
you initialize**





module

THE THREE TREES

What we'll learn...

- The Three Trees
- Commit shortcuts
- Viewing diffs
- Checking the project's history
- Moving and Removing files with git

I'll run through separation of staging from WD, log viewing, diffs, show, mv and rm

The Three Trees



Working Area

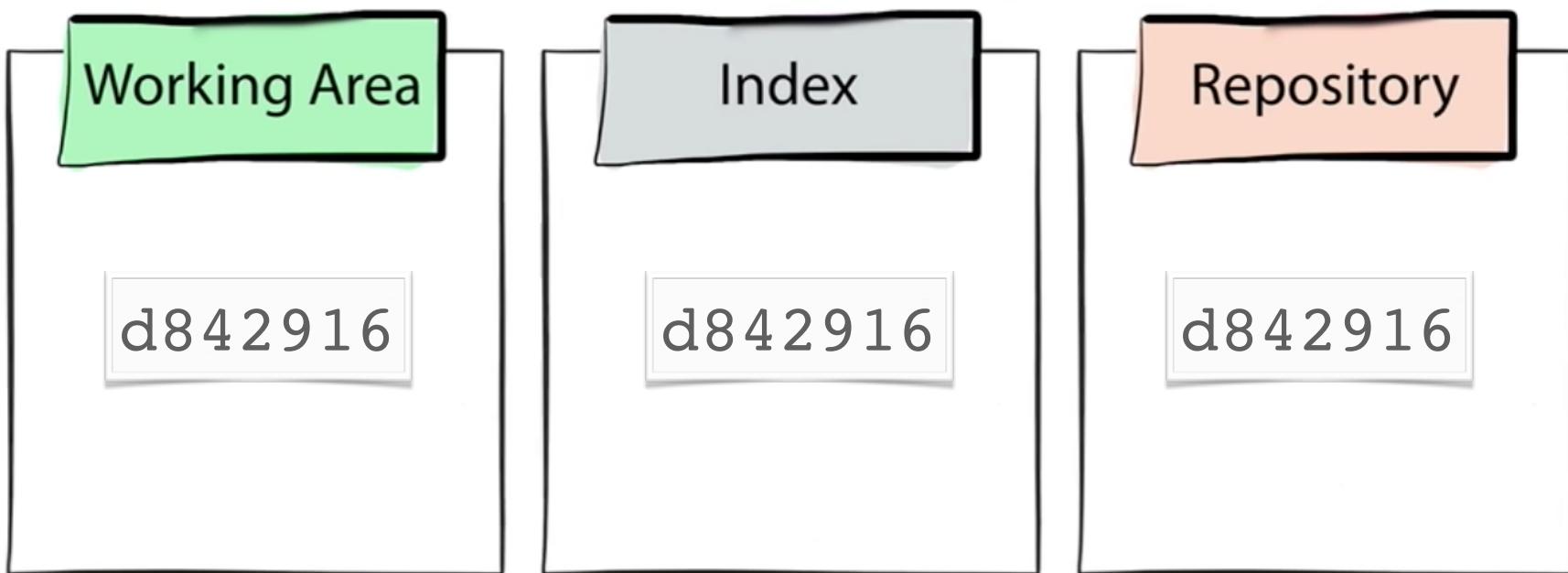
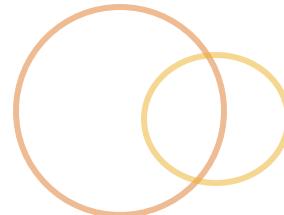
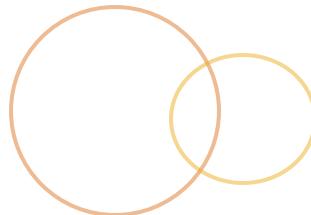
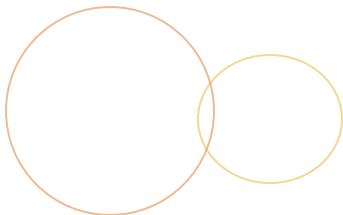
Your filesystem
or
The "Sandbox"

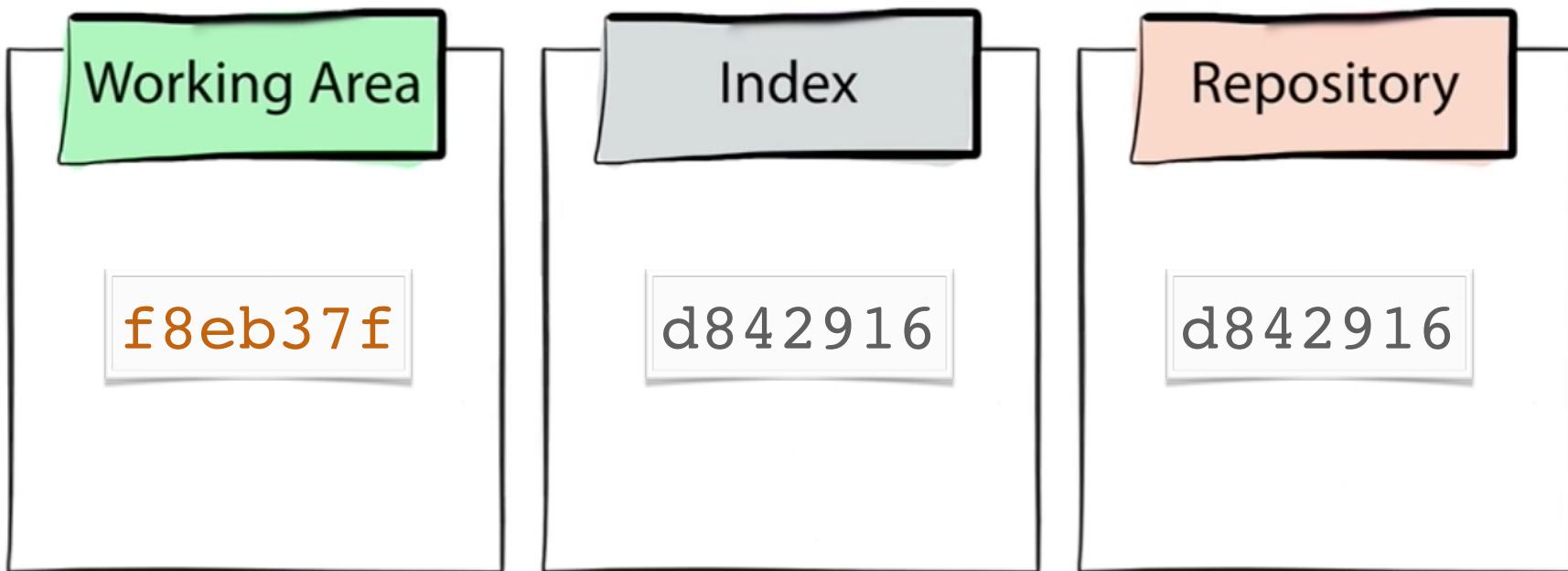
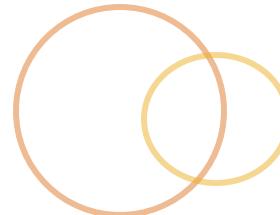
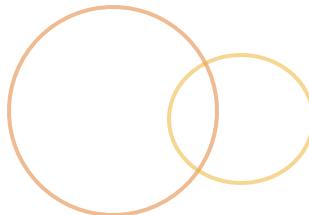
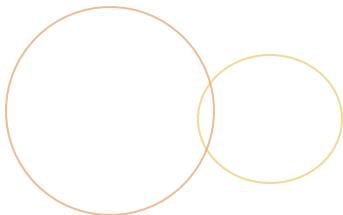
Index

"Next Commit"
or
Staging Area

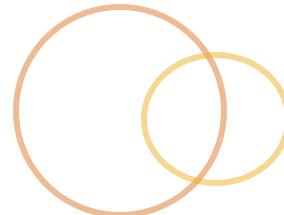
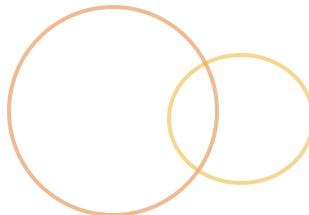
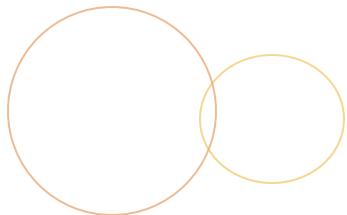
Repository

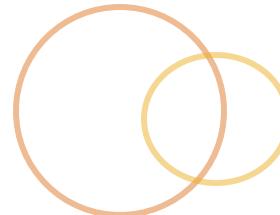
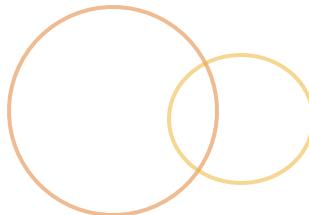
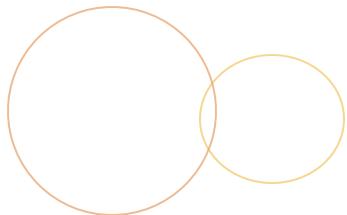
Current Version
(of branch)
or
Repo History





I make one or more edits
to my local files...





Git status

```
git status
```

- Info about your **current branch**
- Changes staged in the **staging area (index)**
- Changes in the **working directory**
- Hints for undoing things

```
# quick stats  
git status -s
```

Stage with add

```
git add <pattern, file or files>
```

- Stages the file(s) or changes
- Begin tracking new files
- Later... marking conflicts as resolved

```
# staging all changes in a directory  
git add bin
```

```
# staging with pattern matchers  
git add bin/*.sh  
git add *.html  
git add .
```

Skipping staging

- ➊ It is possible to commit w/out staging...

```
# auto-stage and commit  
git commit -a
```

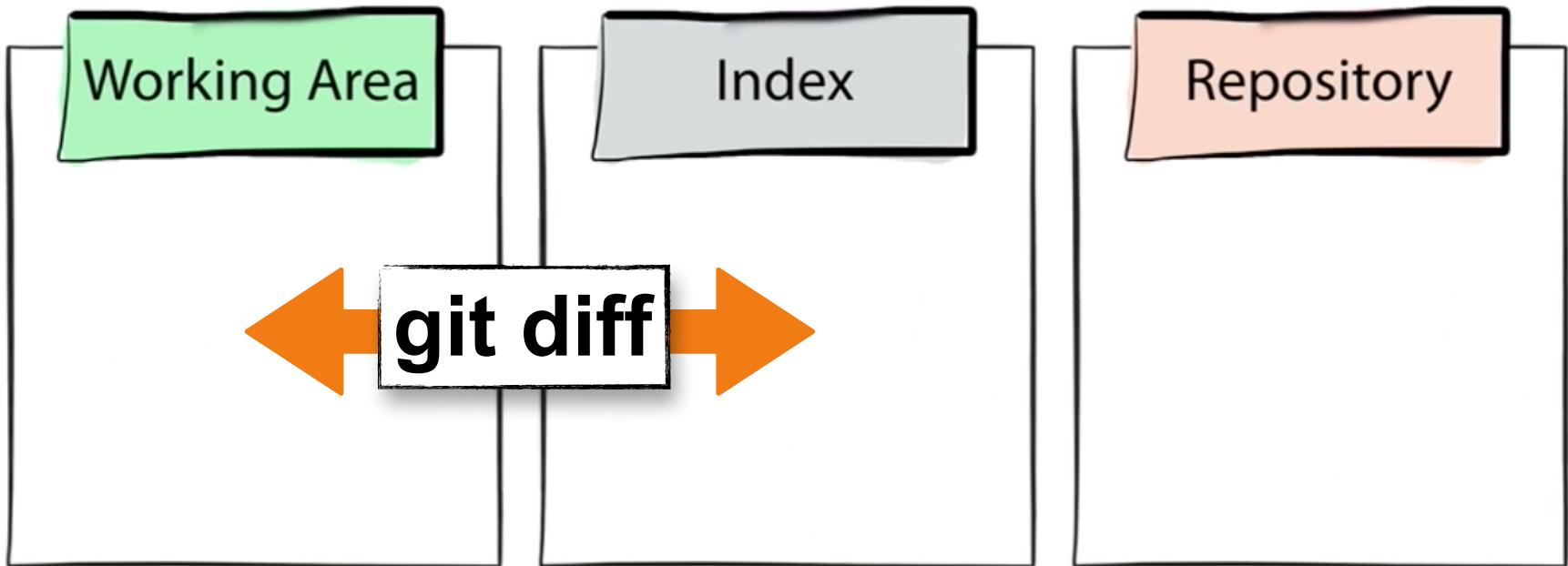
```
# or..  
# this will auto-stage and commit  
# commit changes to the filename  
git commit <filename>
```

- ➋ I don't recommend using this...

Seeing what has changed

git diff

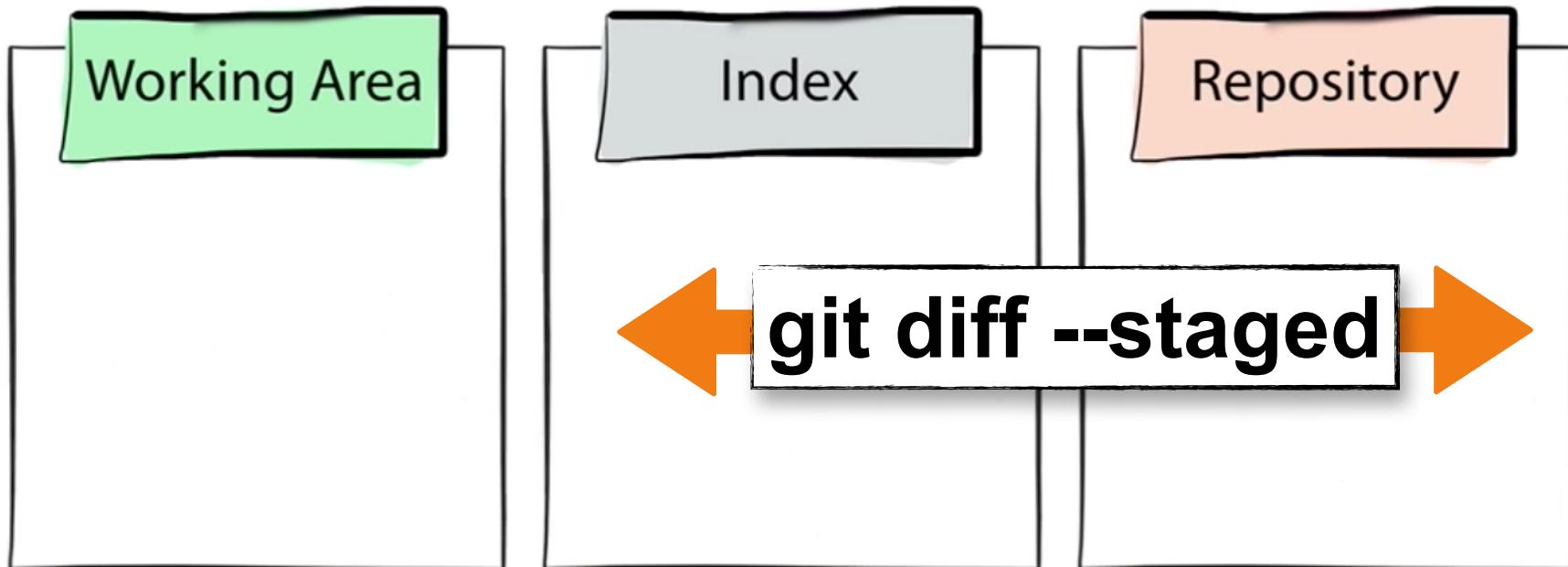
- Shows difference between **working directory** and **index**
- Does not include untracked files



Seeing what has changed pt.2

```
git diff --staged
```

- Compare changes between staging area and the latest version in the repository.
- What you will commit

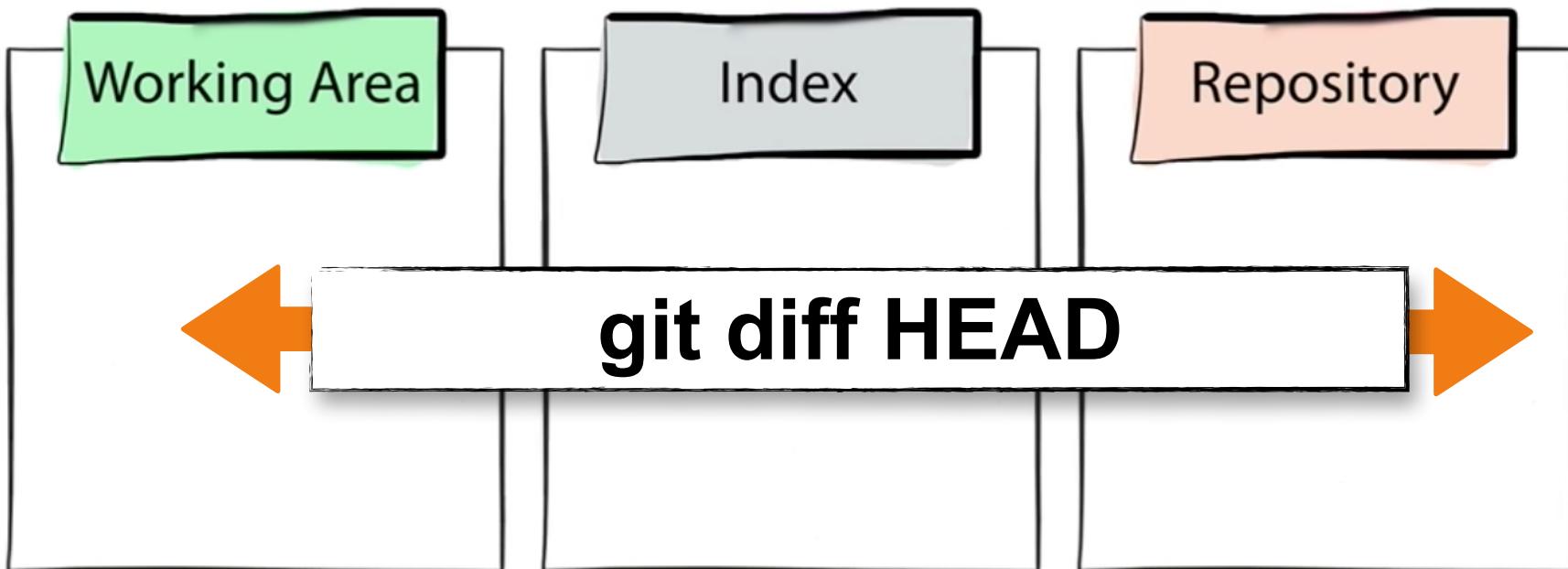


Seeing what has changed (pt3)

- Viewing all changes you've made, combined

```
git diff HEAD
```

"What is different between
my files (WD) and the last
commit (Repo)"



Committing

```
git commit
```

- Applies files/changes from the staging area into the repo
- ...by creating a **commit object** in your git database

Commit Messages

- All commits *require* a message
- You can commit with a message (shortcut)

```
git commit -m "My commit message"
```

- You can also commit an empty object

```
git commit --allow-empty
```

Commit Gotcha

- You can commit individual files changes

```
# ... edit my-file.txt  
git commit my-file.txt
```

What's wrong here?
1. forgot to 'add'
2. commit specified a file

- Skips staging the change
- Bypasses anything else staged
- It's actually a flag option (`--only`)
- ***Probably don't ever want to do this...***

Inspecting commits

```
git show
```

- Shows info about a commit

```
# latest commit
```

```
git show
```

```
# inspect a specific commit
```

```
git show 42d484c401f0a19cc8a954c16240821329acefac
```

```
# or use the abbreviated commit id
```

```
git show 4234
```

View your history

```
git log
```

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

Git rm and mv

○ Delete a file or directory

```
git rm [-r] <file or dir>
```

○ Move a file or directory

```
git mv <old> <new>
```

○ They are unnecessary, why bother with it?

- They auto-stage the change

- Which can reduce confusion

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" repo

After each step be
sure to check
status and the diff

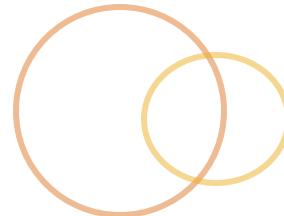
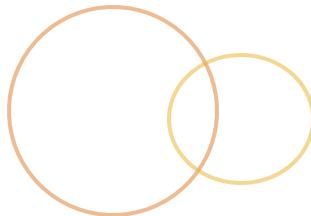
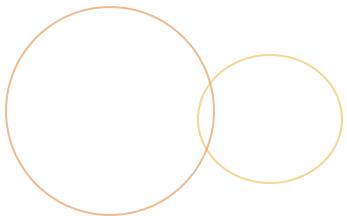
1. **Create** a new repository, "about-me"
2. Create a txt file named with your name
 - Touch <yourname>.txt
3. **Stage** the change
4. **Commit** the change
 - Check the log
 - Use "git show" on the commit id you just created
5. **Edit** the file to add a short profile about you:
6. **Stage** those changes
7. Then **commit**

All done?

- Try creating a new file, "test",
add then commit it
- Try renaming it to "rename-test",
add then commit that change

Toolkit

```
git init
git add
git status
git commit
git show
git log
git diff
git diff --staged
```



module

UNDOING

What we'll learn

- Undo edits in your **working directory**
- **Un-staging** changes to a file or directory
- Undoing a **commit**
- Learning more about what **HEAD** is
- Introduction to **reset**

I'll run through clearing my wd, staging area, un-doing a commit and amending

Throw out changes in my WD

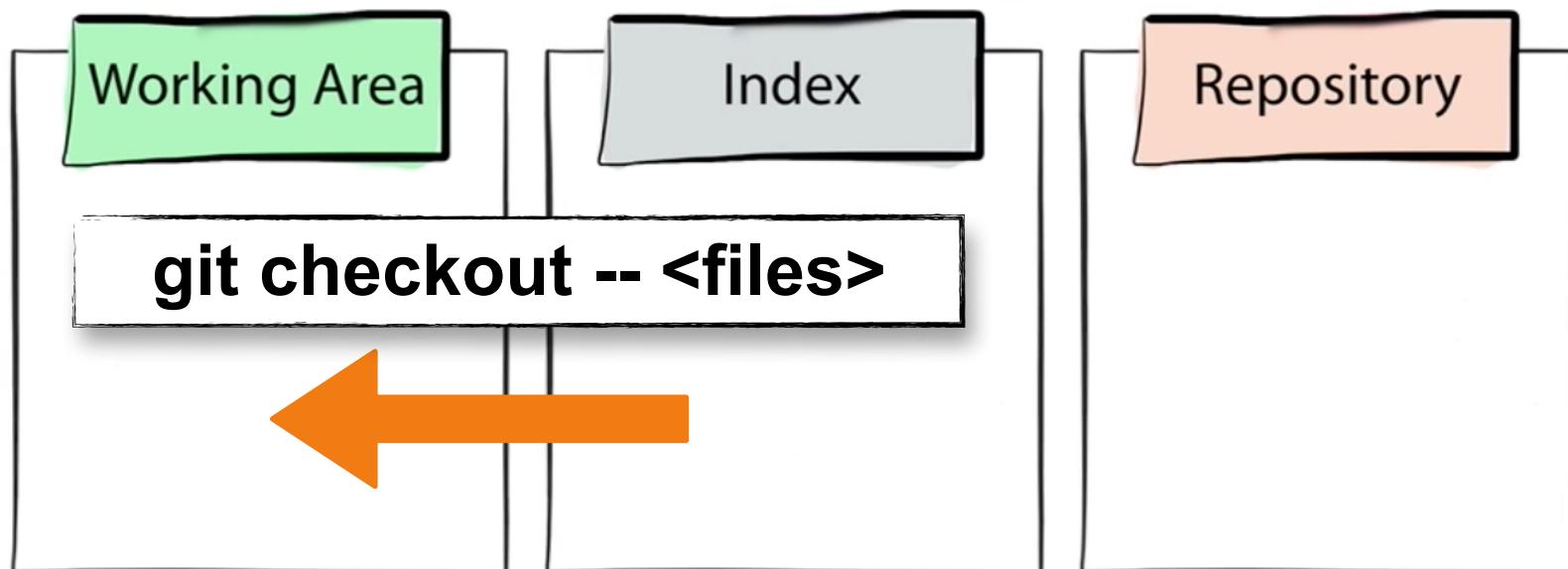
- Checkout can remove **un-staged** changes in your **working directory**
 - One of a few potentially lossy operations

```
git checkout -- readme.txt  
git checkout -- src/*.js  
git checkout -- .
```

- You can *also* use checkout to switch to a commit's version of a file, folder, etc. (Affects the WD only)

```
git checkout 145820e -- readme.txt
```

Checkout on files/folders



A staged change can't be "undone" with checkout...

Un-staging

- ◉ Reset can remove changes in your staging area
 - ◉ Totally safe, unless you find yourself doing --hard

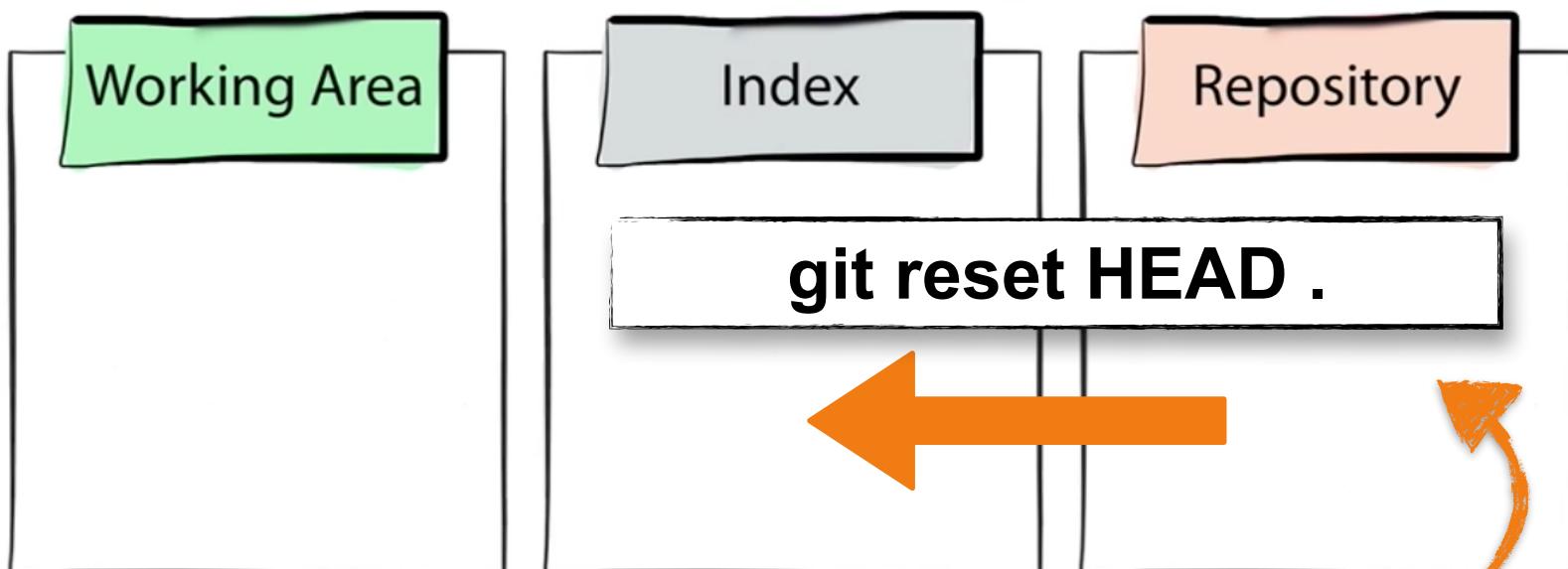
```
# stage it...
git add readme.txt
# now un-stage it...
git reset HEAD readme.txt
```

- ◉ You can also reset to a specific commit's version
(Affects staging area only -- this produces oddness)

```
git reset 145820e readme.txt
```

Reset w/ files

- In this scenario, `reset...` updates the index to be the 'version' the repository is currently at (HEAD)



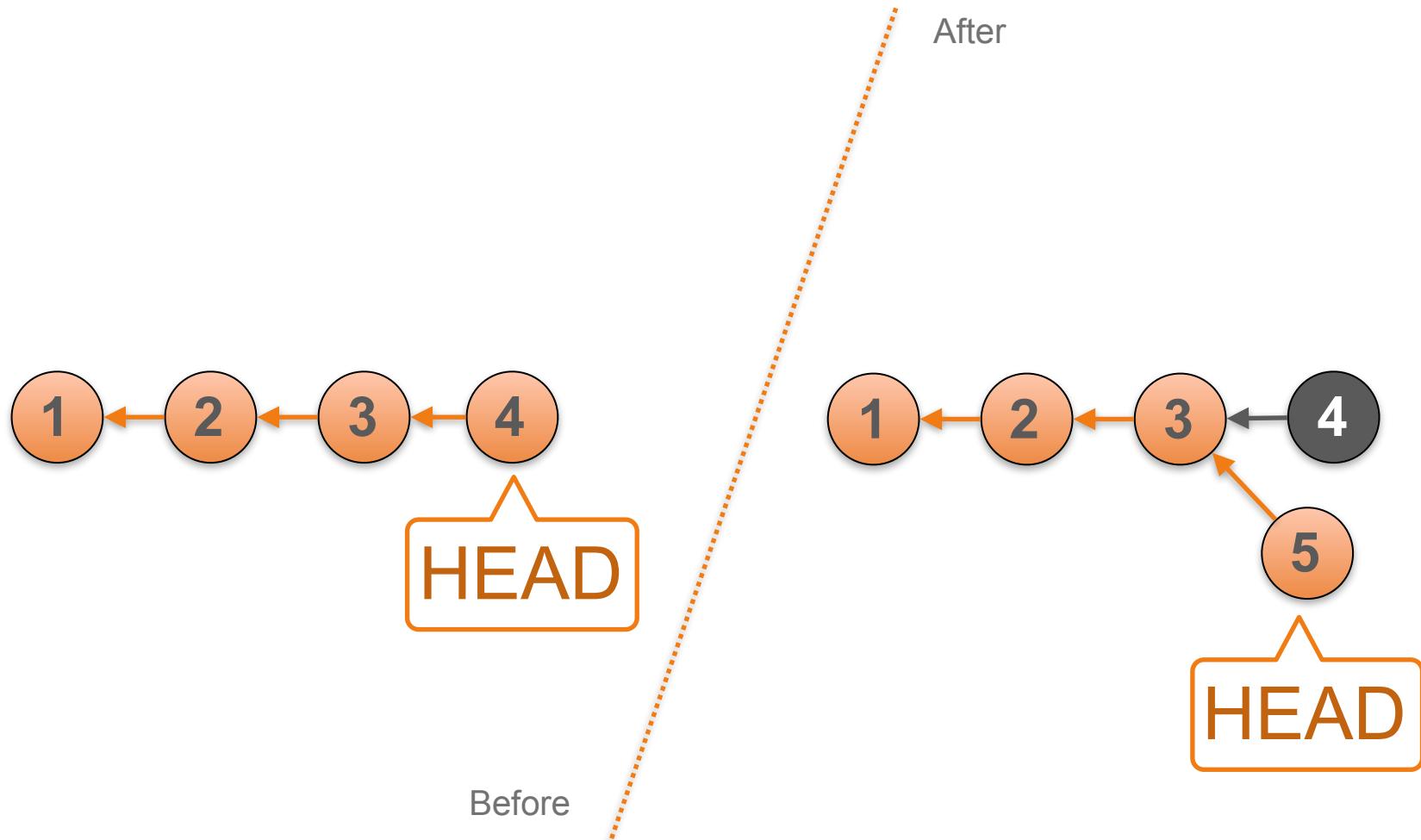
Think of HEAD as:
"The current version"
or "The latest version"

Amending the last commit

- Amend is useful to modify your last commit
 - Edit the message
 - Add or edit the changes you made
- Last commit?
 - Actually, you are amending the commit HEAD points to

```
git commit -m "Stuff!"  
# want to change that commit?  
# ok... optionally, edit files  
# then stage  
git add .  
# then amend...  
git commit --amend  
# git will allow you to edit the commit msg
```

What did --amend do?



Undo your last commit

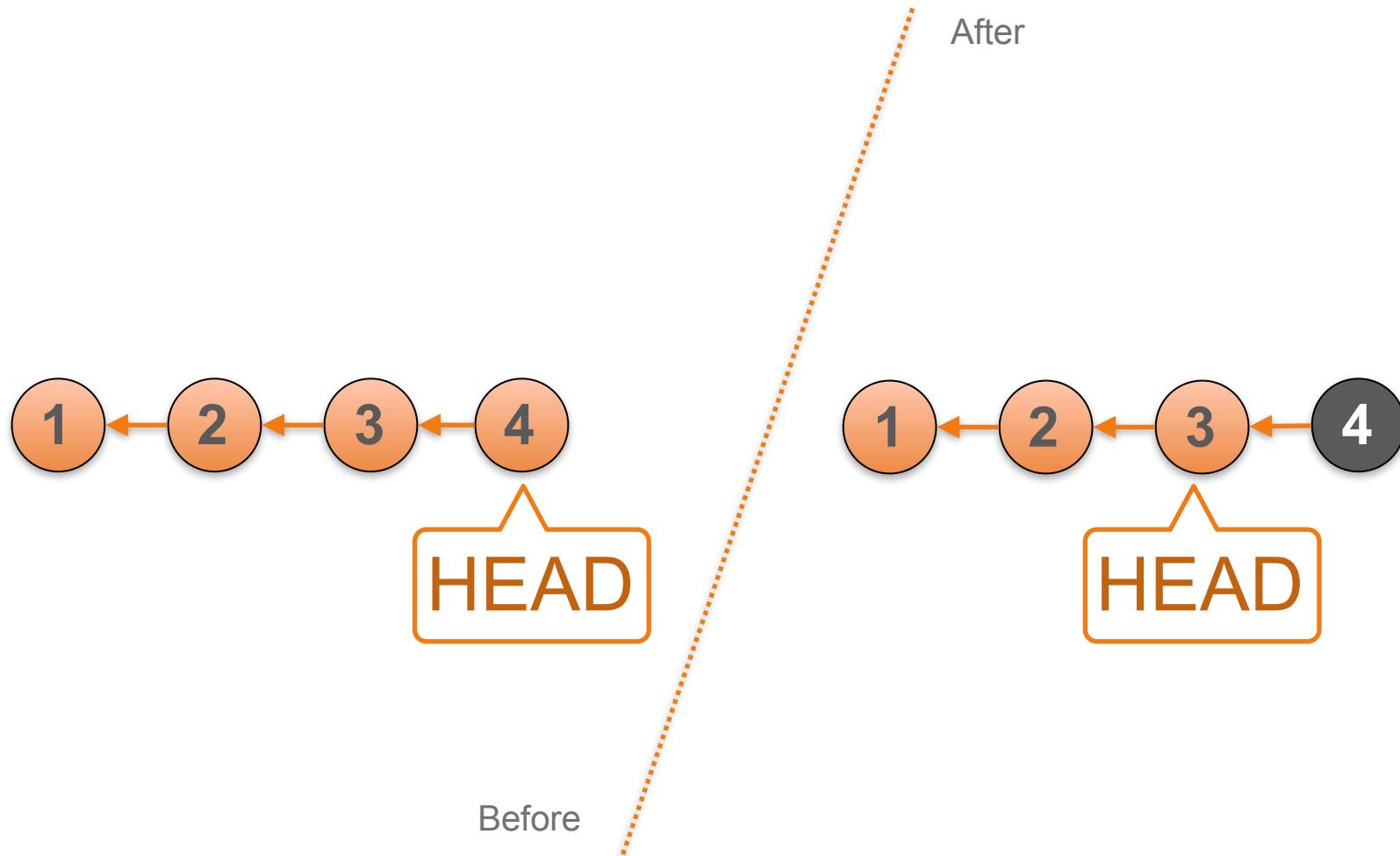
- Reset allows you to change your branch pointer

```
# move back by one commit  
git reset HEAD^
```

"Reset my BRANCH to point back one commit"

```
# same (ish) thing  
git reset HEAD~1
```

What did Reset HEAD[^] do?



Reset anywhere...

- ⌚ Yup, you can reset your history to anywhere

```
# moves history back 5 commits  
git reset HEAD~5
```

```
# moves history back to specific commit  
git reset ab3f9s
```

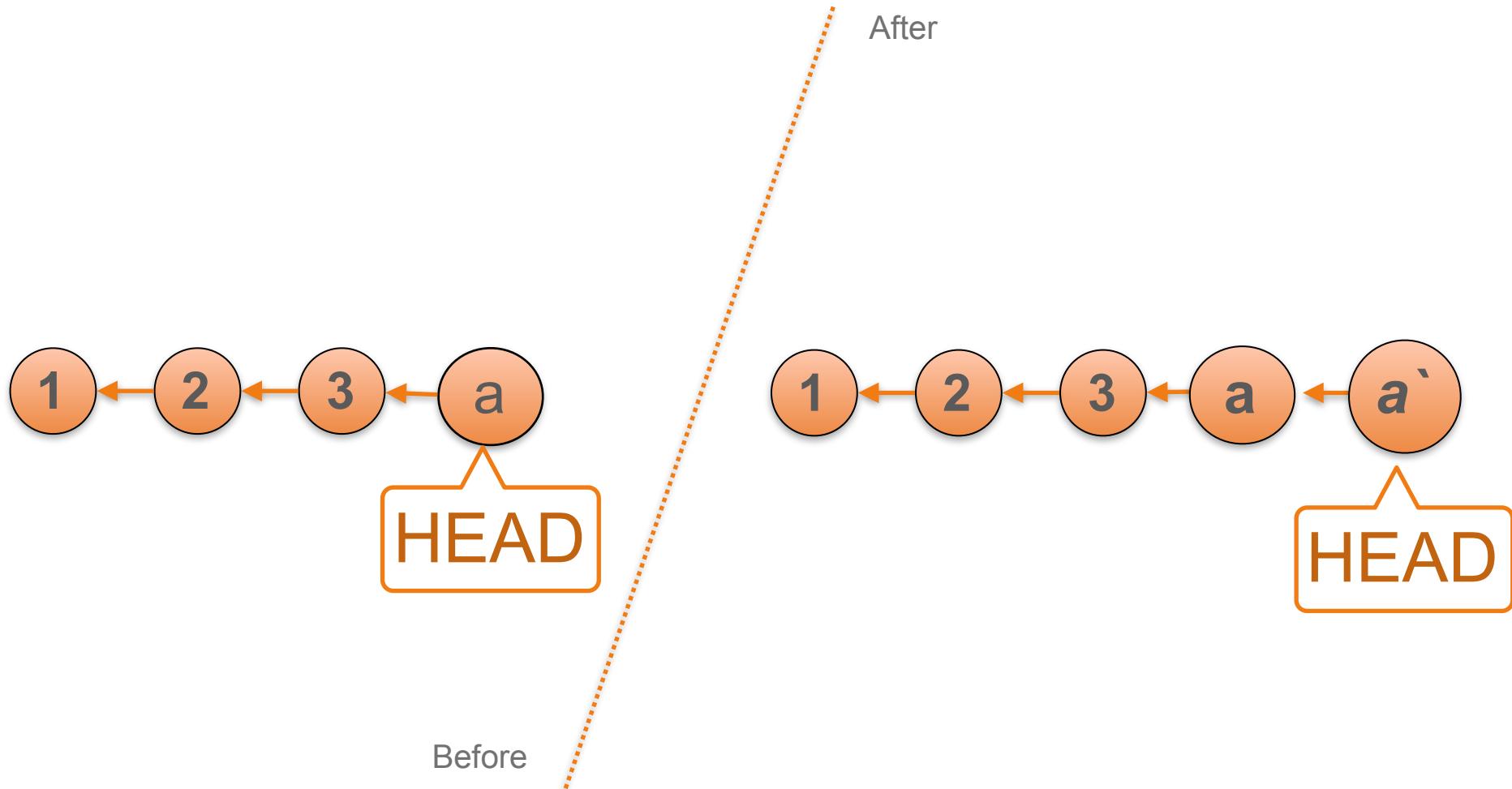
By *history* - I'm really talking about the current branch

Git revert

- Applies the ***inverse*** change set from a commit
- Good for undoing ***public*** commits

```
# undo commit ab3e230  
git revert ab3e230
```

What did revert ab3e230 do?



Git clean

- Unlike checkout, only affects untracked files

```
# throw out any untracked  
# files & directories  
# (except ignored files...)  
git clean -d
```

Recap

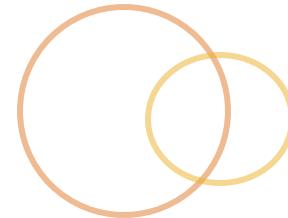
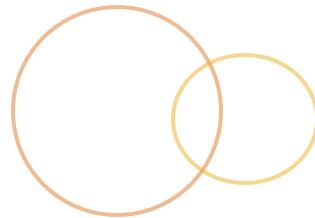
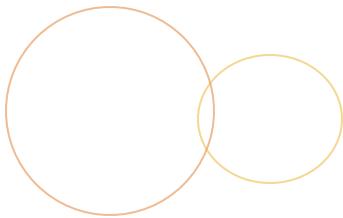
- **Un-stage** things (from **staging area**) with
 - `git reset <file>`
- **Un-change** files in **working directory** with
 - `git checkout -- <file>`
- **Modify** your **last commit** with
 - `git commit --amend`
- **Undo** the **last commit** with
 - `git reset HEAD^`
- **Reverse** any commit with **revert**
- **Clean** will throw out untracked stuff

Lab: Undoing things

1. Create two new files, "junk" and "LICENSE"
2. Edit "<your-name>.txt" file, ie: add your fav animal
3. **Stage** all your changes
 1. Un-stage "junk" and "LICENSE"
4. **Commit** only the change to "<your-name>.txt"
5. **Stage** the changes to LICENSE
 1. Then **commit**
6. Stage and commit "junk"
 1. Hmm, didn't mean to do that...
 2. Undo your last commit (with "junk")
 3. Then **delete** the file, "junk".
7. Git **status** should show a clean repository
8. Finally, **amend** your last commit and edit the commit message
9. **Bonus:** use **amend** again, but this time make an additional edit to the "LICENSE" file so that the change is included in the previous commit

Toolkit

```
git add  
git status  
git commit  
git checkout --  
git reset HEAD  
git commit --amend  
git reset HEAD^
```



module

BRANCHING (ONLY)

What we'll learn

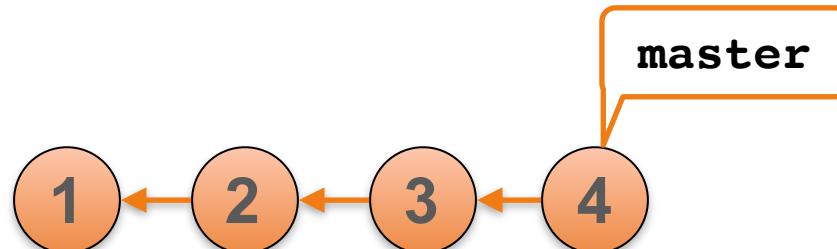
- What is a branch
- Creating and using branches
- Comparing branches
- View branching in the log

Branching in git

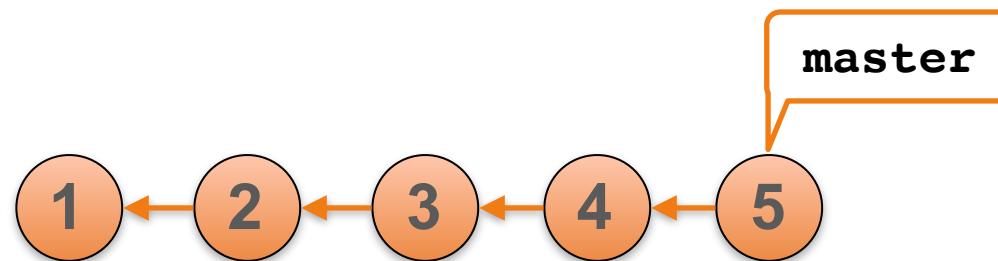
- Branches are cheap, use them frequently
 - Experimentation
 - Separate a unit of work like a ticket, feature, bug
 - Keeps related changes organized
 - Avoids noise/clutter in the mainline
- A branch is *like* a fresh copy of all your files
- **master** is our mainline (by convention only)

How branching works

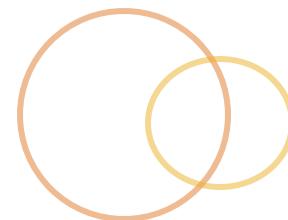
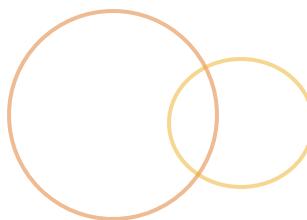
- A branch is like a *live bookmark* for a commit



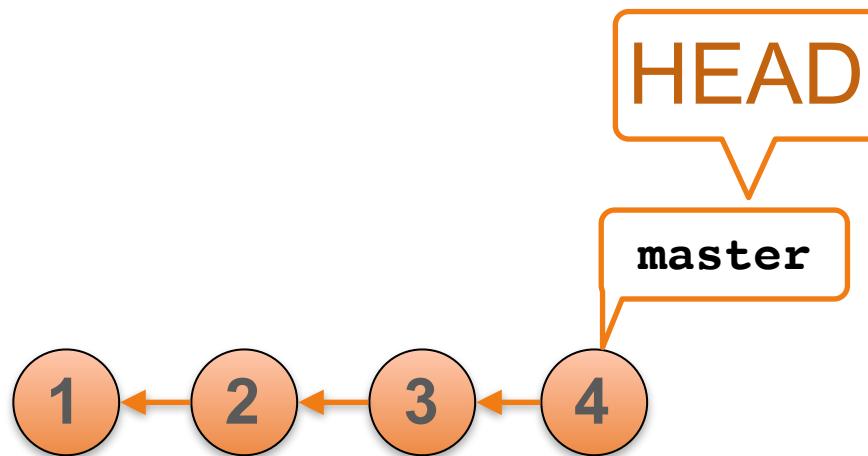
- A **commit** automatically moves the branch forward (to the new commit)



HEAD

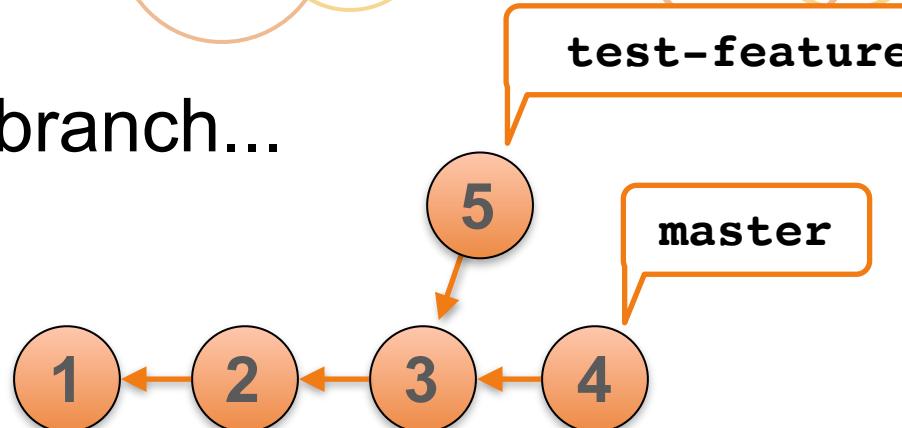


- ◉ HEAD is a pointer that keeps track of which branch you are on...

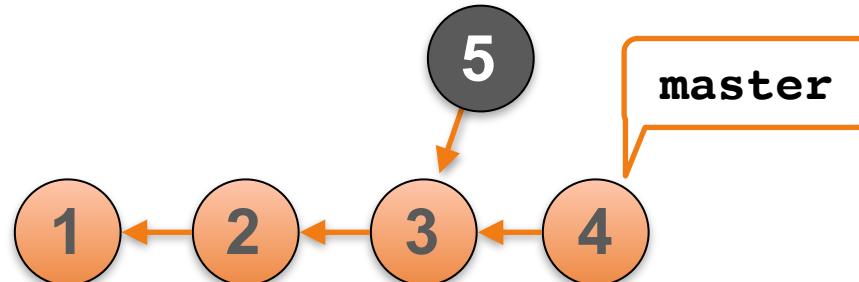


Branches make commits reachable

- Here is a branch...



- Without the branch, the commit is *unreachable*



Git branch commands

```
# list your branches  
git branch
```

```
# create a branch  
git branch <name>
```

```
# switch to a branch  
git checkout <name>
```

```
# create and switch shortcut  
git checkout -b <name>
```

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

```
# show history as a graph
# with all commits displayed
# and all branches labeled
git log --oneline --decorate --graph --all
```

Checkout

- Multi-purpose command that updates your **working directory** (*and index*) in a *non-destructive* way (overlays)

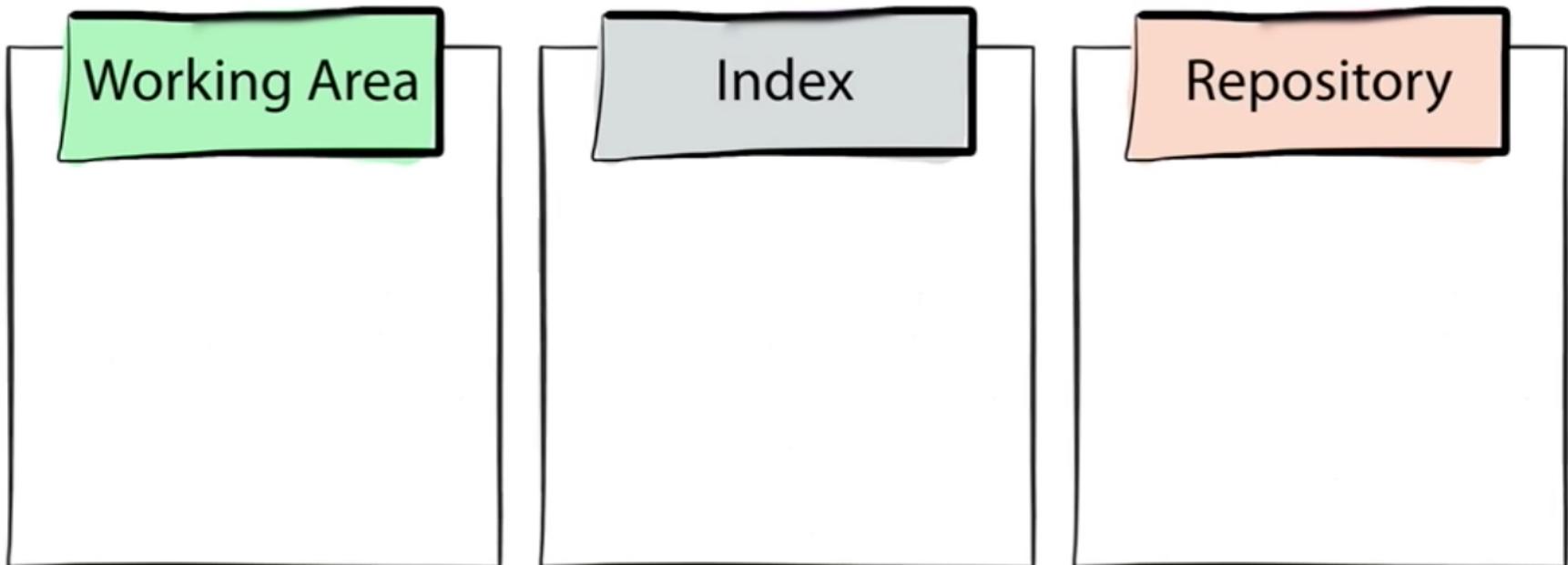
```
# undo changes to a file  
git checkout -- <file>
```

```
# checking out a branch  
git checkout test-branch
```

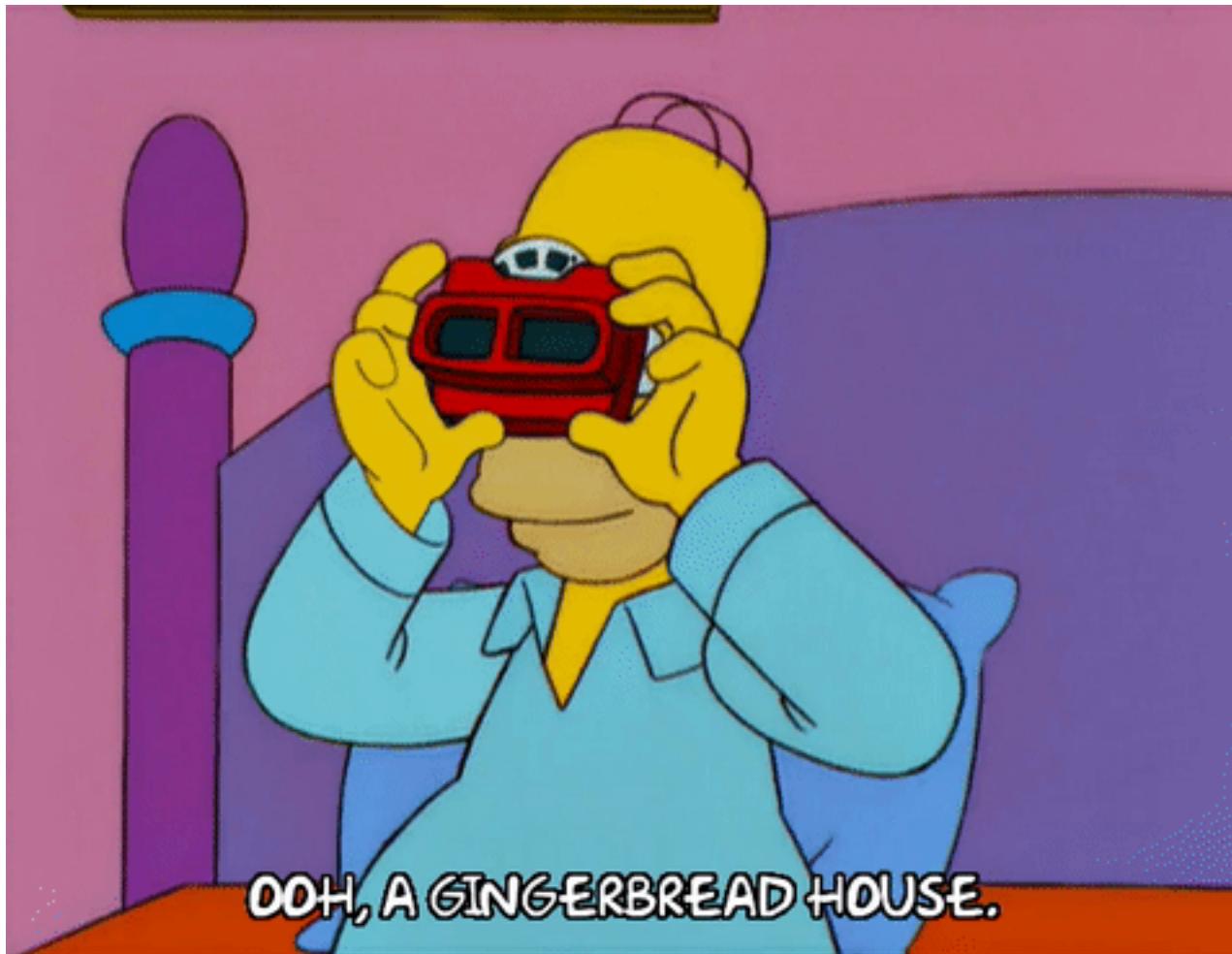
```
# checking out a specific commit  
git checkout abfecf3
```

Branch & Checkout

- So, how do **branch** and **checkout** affect the three trees?

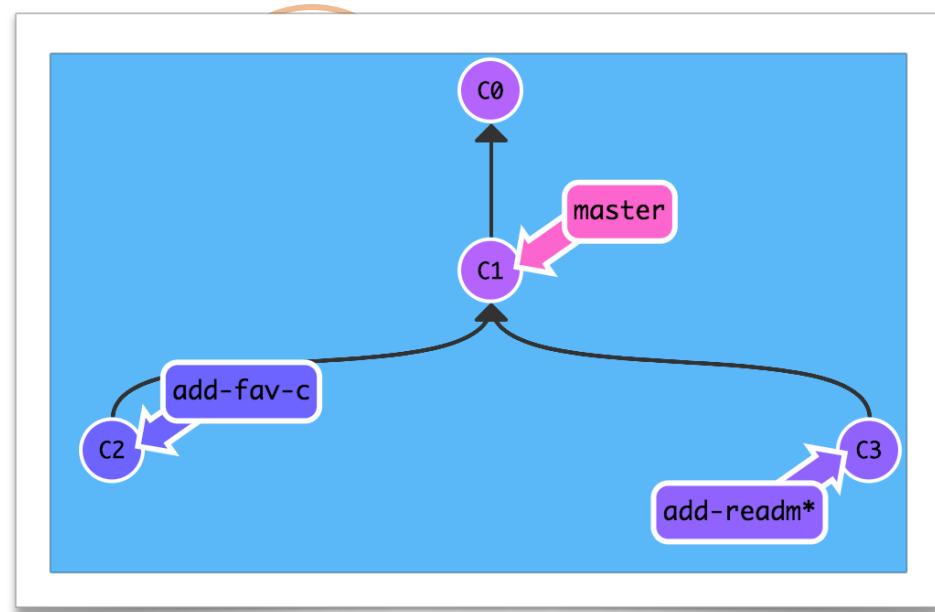


One staging area, many commits



Lab: Branching

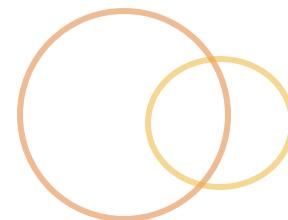
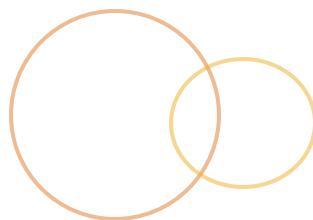
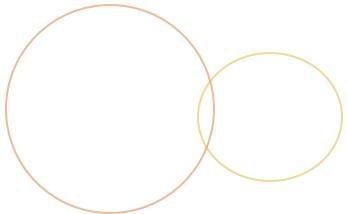
1. List your branches
2. Create a new branch off of master
 1. Label it "add-readme"
3. On the "add-readme" branch...
 1. Create a new file, "README"
 2. Stage and commit it
4. Back on "master"...
 1. Create a second branch, "add-fav-color"
5. On "add-fav-color"...
 1. Edit your <name>.txt file to add your favorite color to the list
 2. Stage the change and commit it
6. View the log as a graph, all commits



What we're going for

Toolkit

```
git status  
git add  
git commit  
git log  
git branch  
git checkout  
git checkout -b
```



module

COMPARING BRANCHES

What we'll learn

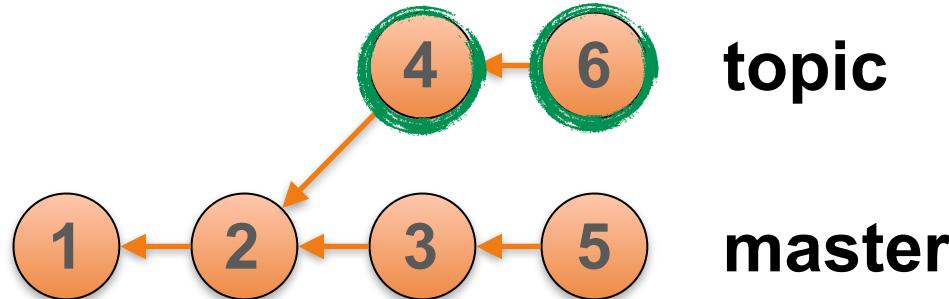
- We can compare commits & files (diffs)
 - **log, diff and range selections** (.. or ...)
- Can generate patches off these, if needed
- Generally use it to see what change you might introduce

New commits in a branch

- What are the new commits in "topic"

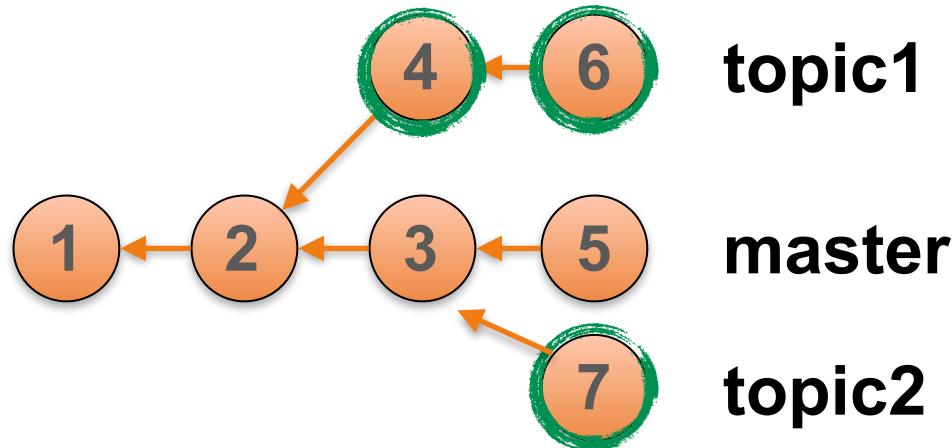
```
# reachable by topic, not master  
git log master topic  
git log master..topic  
git log ..topic
```

HEAD is implied



Commits in one but not another...

```
# reachable by topic1 and topic2  
# but not master  
git log topic1 topic2 --not master
```

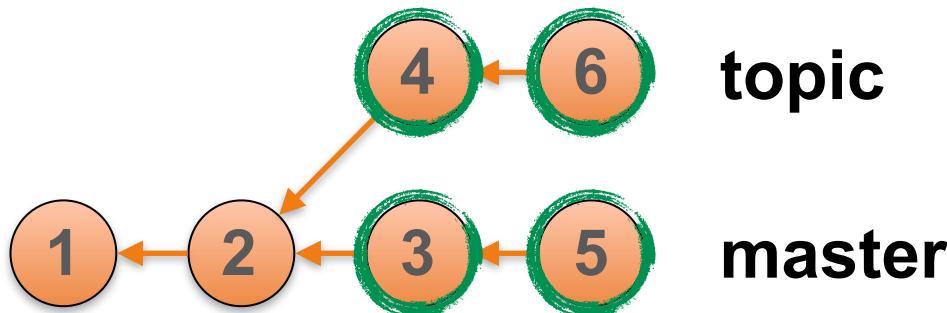


Differing commits

- What are the unique commits across both

```
# commits reachable by either, but not both  
git log master...topic  
git log ...topic
```

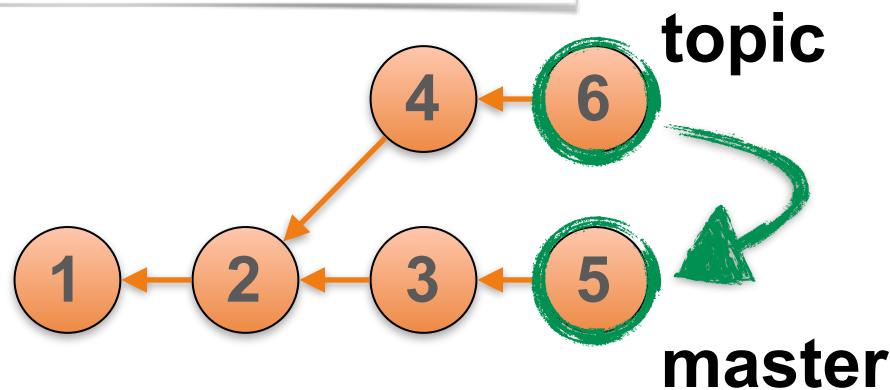
```
# display branch source  
git log master...topic --left-right
```



Combined file differences

- See the diffs between two branches
 - Compares between *tip* of each branch
 - "What patch can I apply to make master be like topic"

```
git diff master topic  
git diff master..topic  
git diff ..topic
```

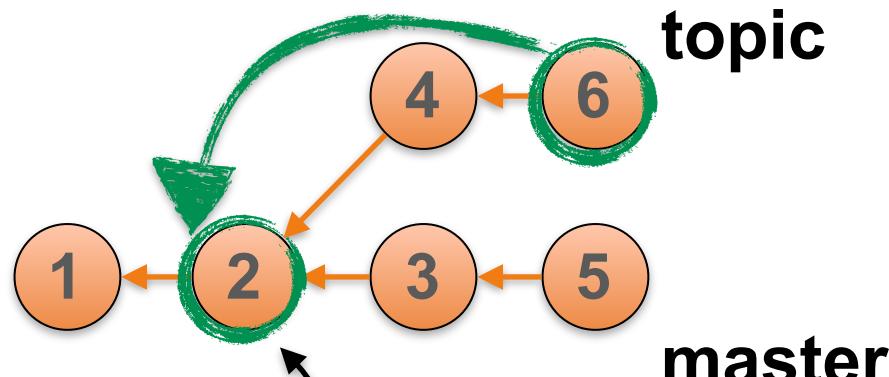


Changes from a branch

- Compares to **common ancestor (merge-base)**
 - "What change will B introduce to A"
 - or... "*What is new in B since it branched off A*"
 - Could create a patch from a branch this way

```
git diff master...topic
```

```
git diff ...topic
```

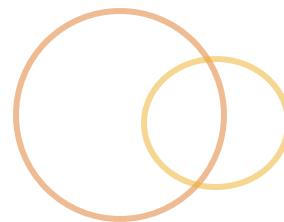
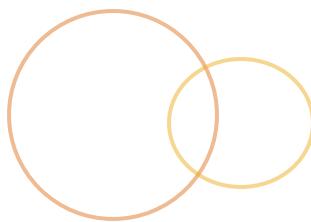


master

topic

```
git merge-base master topic
```

Exercise



What will these output?

git diff

git log

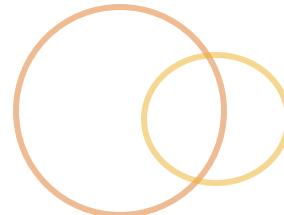
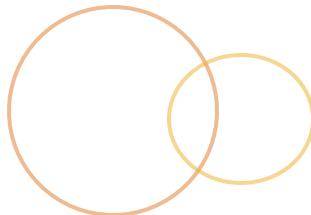
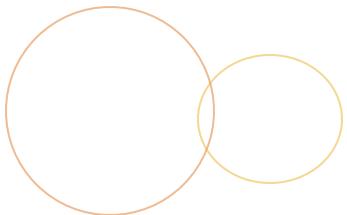
What commits will I merge into master...

git _____..featureA

What file changes will my

feature introduce to master

git _____ master_____featureA



module

MERGING

What we'll learn

- How to merge branches
- What are the basic merge strategies
- Branch management
- Our basic workflow takes form

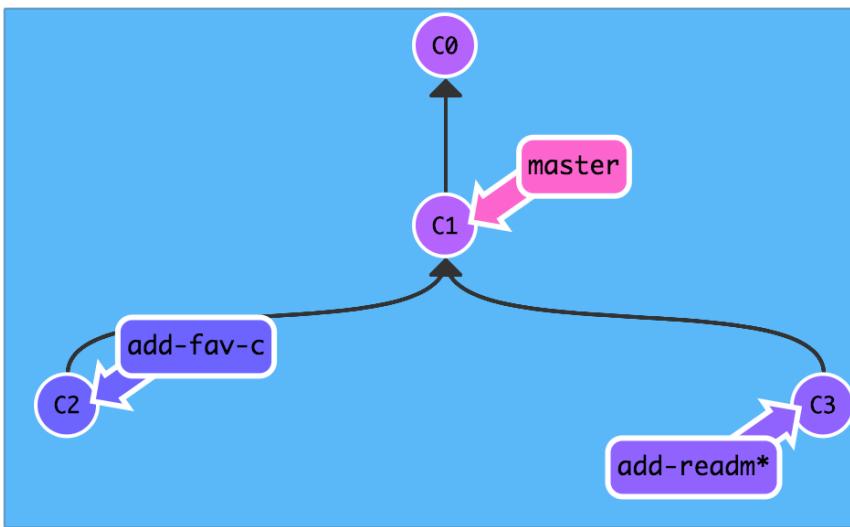
Merging

- A merge **combines the history** of two branches
 - It's how you get changes you've made in one branch into another branch (usually master)

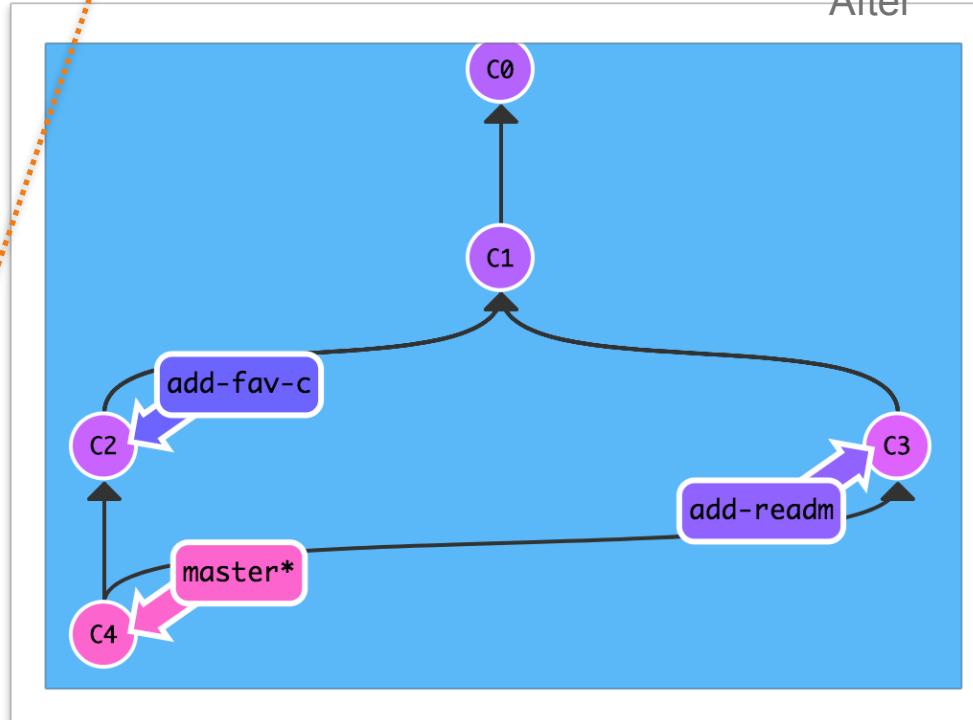
```
# Applies changes from <source>
# into the branch you're on
git merge <source branch>
```

- **Does not affect** the branch you merged
 - Only affects the branch **you are on**
- Git decides on a **merge strategy** to use

Before



After



```
$> git checkout master  
$> git merge add-fav-color  
$> git merge add-readme
```

Merge Strategies

○ **Fast-forward merge**

- Linear history (graph) maintained

○ **Recursive / 3-way merge**

- Merge commit created

○ **Octopus merge**

- More than 2 branches combined through a merge commit

Branch Management

```
# list branches already merged  
# into the branch you are on  
git branch --merged
```

Actually:
Which branches
are "in" the current
branch's history



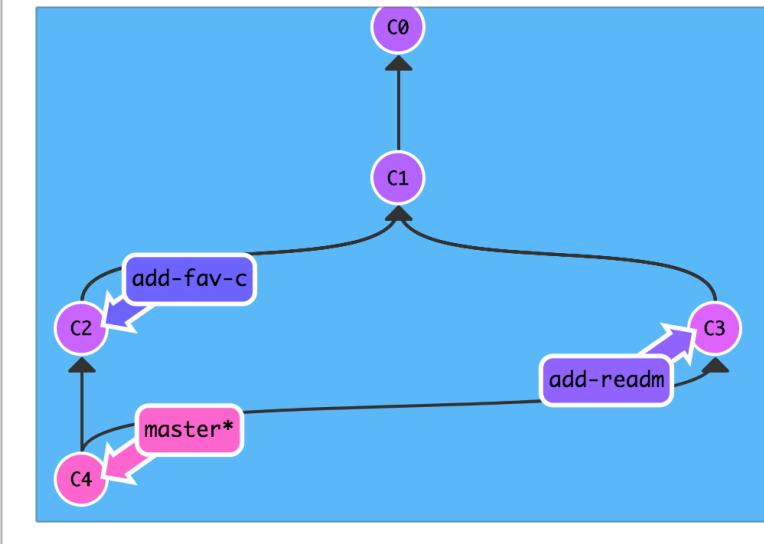
```
# list not-yet-merged branches  
git branch --no-merged
```

```
# delete a branch  
git branch -d <name>
```

```
# force delete a branch  
git branch -D <name>
```

Lab: Merging

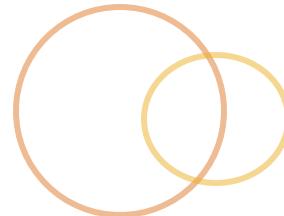
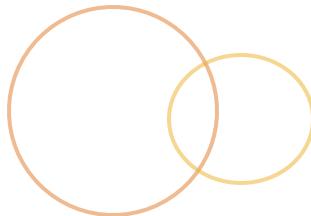
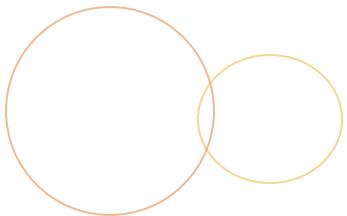
- On master
 - Merge your "add-readme" branch
 - Review your log...
 - What kind of merge did it perform?
 - List which branches are merged and not merged
 - Merge your "add-fav-color" branch
 - What kind of merge did it perform?
- Then create a new branch off master, "add-copyright"
- On "add-copyright"
 - Edit your <name>.txt file to include a copyright at the bottom
 - Stage & commit! **I will stop saying this**
- Back on "master"
 - Merge "add-copyright"
- Review the log
- Delete your merged branches



What we're going for

Toolkit

```
git status  
git add  
git commit  
git branch  
git checkout  
git log  
git branch --merged  
git branch --no-merged  
git branch -d  
git merge
```



module

MERGE ISSUES

What we'll learn

- What is a conflict
- How to abort a merge (avoiding the conflict)
- How to resolve merge conflicts
- Introduction to merge tools

Merge conflicts

- When performing a merge, if there are changes that git doesn't know how to combine you will end up in a "conflicted state".
 - Ex: The two branches contain a change to the same line of the same file
- What do I do when a conflict happens?
 - The merge is in limbo; it is not yet committed!
 - You must either:
 1. **abort the merge** or
 2. **resolve the conflict**

Abort a merge in progress

- Cancel the merge with --abort

```
# you'll remain on the branch you were on  
# and it is as though the merge never began  
git merge --abort
```

- But, you'll still want to merge the branch, eventually...

Resolving the conflict

```
# first, check which files are in conflict  
git status  
# then, fix the files in your editor
```

about me
hello!

profile
hello!

Conflict!

```
<<<<<< HEAD  
about me  
=====  
profile  
>>>>>> new-branch  
hello!
```

```
# then...  
git add .  
# finally, tell git to complete the merge  
git commit
```

Undoing a merge

- How can I undo a merge that I just did?
 - If you haven't done anything else
 - And you haven't left the target branch yet

```
# puts your branch back to where it was  
# before the merge happened
```

```
git reset --hard ORIG_HEAD
```

```
# back to... first parent  
git reset HEAD^
```

```
# back to... second parent  
git reset HEAD^2
```

Undoing a merge

```
# this won't work  
# how you expect  
# every time...  
git reset HEAD^
```

```
# there is a  
# second parent,  
# too!  
git reset HEAD^2
```

```
ryan@Mo-2 ~ /repos/ryans-project  
* 11536ca (HEAD -> master) Merge bran  
|\  
| * ce89db0 (edits) Three  
| * 5885008 Two  
| * 183b679 one  
* | 701fb6d file51  
* | 9377f78 file50  
|/  
* 9559c7c (origin/ticket-56, origin/n  
Merge pull request #2 from mrmorris/new  
|\  
| * b9ffde2 (origin/new-work-3) Merge  
| |\
```

Merge tool

Conflicts can be tedious, a GUI can help

```
# opens your default GUI  
git mergetool
```

Some great tools

- [sublime merge](#)
- [kaleidoscope](#)

```
# check which tools you have  
git mergetool --tool-help
```

```
# run a specific tool  
git mergetool -t <tool>
```

```
# configure it to always use one tool  
git config --global merge.tool <tool>
```

Common issues when resolving

- You commit the conflict-resolution markers
- You commit extra files created by your editor

```
$> ls -la  
filename  
filename.ORIG_B  
filename.ORIG_A
```

Lab: Resolve a conflict

Let's create a conflict!

- Create two branches off of master
- First branch will be called "red"
 - In this branch, edit your favorite color to be "Red!"
- Go back to master
- Create and checkout a second branch called "blue"
 - Edit favorite color to be "No, Blue!"
- Back on "master"
 - Merge "red" then "blue"
 - You should get a conflict
- Abort the merge!
 - Then merge "blue" again
- Resolve the conflict
 - When done, stage and commit

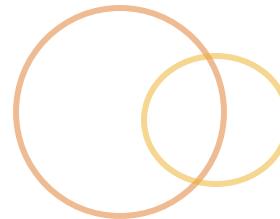
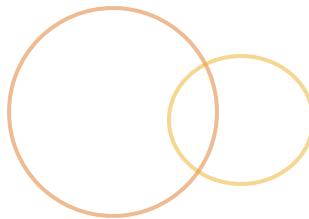
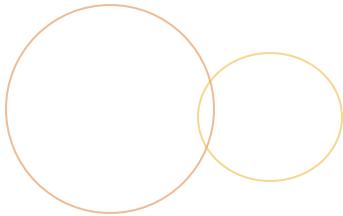
Filenames different?

Create a conflict by editing the same line of the same file in two different branches.



Toolkit

```
git status  
git add  
git commit  
git branch  
git checkout  
git log  
git merge  
git merge --abort  
git mergetool
```



module

CORE ODDS & ENDS

TAGS, ALIASES, STASHING AND IGNORE

What we'll learn

- Parent References
- Tags
- Aliases
- Stashing
- Ignoring

Tags

- A static bookmark for a special commit
- **Lightweight**, Annotated and **Signed**
- What should I tag?
 - Releases
 - Each x.x.x version
 - Important commits you may want to come back to

```
# add a new (lightweight) tag  
git tag <name> <optional-commit>
```

```
# add an annotated tag  
git tag -a <name> -m "Message"
```

```
# list tags  
git tag
```

You can
checkout a
tag just like a
branch, *but...*

Aliases

○ Command shortcuts you configure

```
git config --global alias.co checkout
```

○ What would you alias?

○ Maybe make "co" an alias to "checkout"

○ Or make "stage" an alias of "add"

○ Any command you want to simplify

○ Reference non-git commands with "!" prefix

```
git config --global alias.visual '!gitk'
```

Set up some aliases

```
# co
git config --global alias.co checkout

# unstage
git config --global alias.unstage 'reset HEAD --'

# undo-merge
git config --global alias.undo-merge 'reset --hard
ORIG_HEAD'

# graph
git config --global alias.graph 'log --oneline --graph --
decorate'
```

Stashing

- Quickly stores work in progress without committing -- local only

```
# saves changes in your stash  
git stash
```

```
# applies your most recently stashed work  
git stash pop
```

```
# list what is in your stash  
git stash list
```

```
# git stash apply <name>  
# git stash drop <name>  
# git stash pop <name>
```

Ignoring files

- You can tell git to ignore certain files and folders
 - Set up a `.gitignore` file in the root of your project
 - List files or patterns to ignore

```
*.tmp  
*.log  
  
# directories  
tmp/  
logs/*.*  
  
# negate a pattern  
!main.log
```

- Github has a lot of [prefab](#) `gitignores`

Empty folders

- Add a empty, .gitkeep, .keep file
- Optionally ignore it

```
# ignore files in logs folder  
logs/*
```

```
# but keep the folder  
!logs/.keep
```

Other special ignores

- Ignore a local file without sharing the "ignore"

```
#edit this file like .gitignore  
.git/info/exclude
```

- Global ignores (all your repos)

```
git config --global core.excludesfile ~/.gitignore_global
```

Lab: Day 1 (Stashing)

- Make sure you're on master
- Make some edits to your profile
 - Add your favorite city?
- Stage the changes

Shifting gears: You have to go fix the copyright ASAP

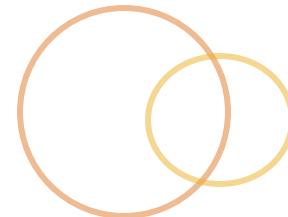
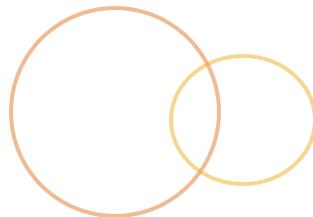
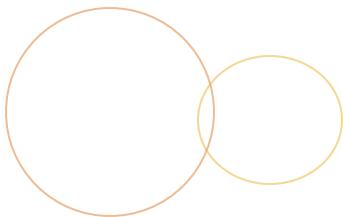
- Stash those changes
- Create a new branch off master, "fix-copyright"
 - On "fix-copyright", edit your copyright year to be 1901-2079
 - Stage & commit

Let's get back to what we were doing...

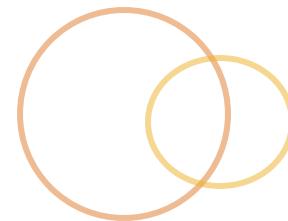
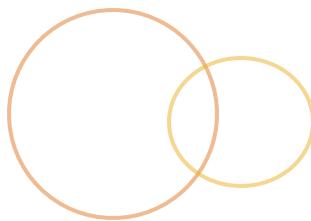
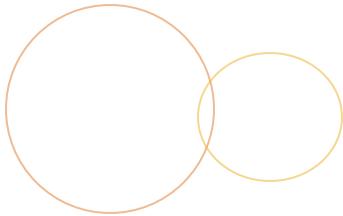
- Back on master...
- List your stash
- Un-stash the changes you were working on
- Make any further edits you want to your profile
- Create a new branch, "fav-city"
 - On "fav-city", stage and commit your profile edits
- Merge both "fav-city" and "fix-copyright" into master...
- View your log and delete merged branches

Toolkit

```
git status  
git stash  
git stash pop  
git stash list
```



End day 1



optional...

DEEP DIVE ON GIT

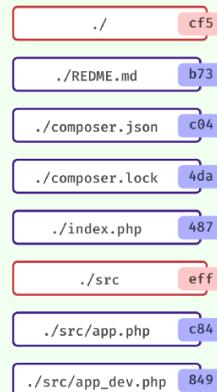
The trees (more detail)

Working Directory
Your filesystem
one "version" at
a time...

The Index/Staging
"Next commit"
and..
Tracks changes
across the two
< ----- >

The repository
a mess of objects:
commits
trees
blobs
tags

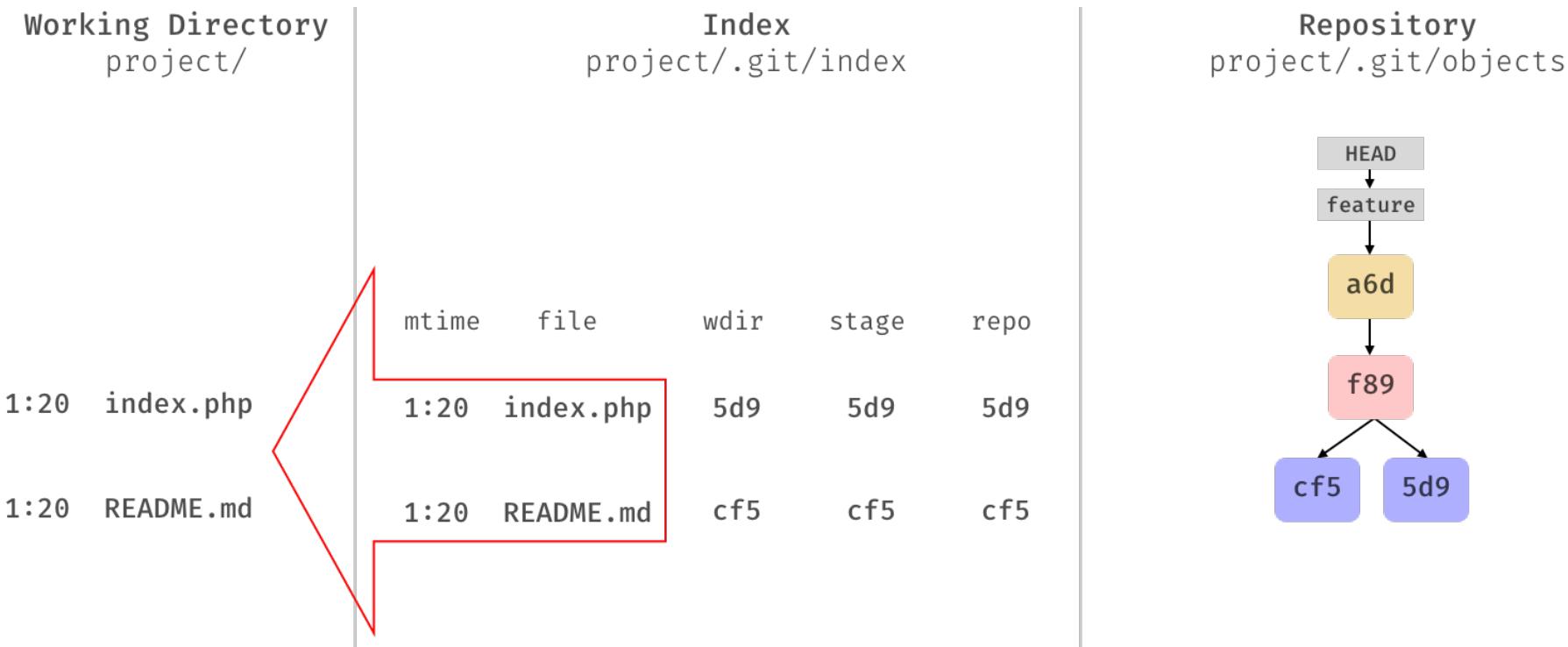
```
.  
├── README.md  
├── composer.json  
├── composer.lock  
└── index.php  
src  
└── app.php  
    └── app_dev.php
```



a6d	b73	c04	a6d	4da	487
c84	cf5	849	de2	eff	cf5
30b	b72	0b8	ca6	909	b09
ac9	038	f5e	d5a	f89	fae
f72	59d	058	af8	1a1	f63
a32	3a2	33b	b5f	d22	4fc
ca6	875	2e1	cbc	aa5	bc7

The index

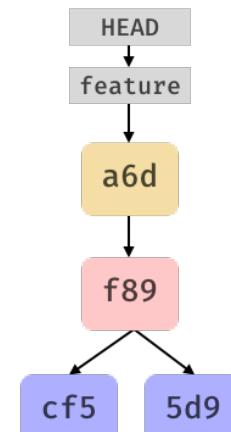
- Git keeps track of the three trees within .git/index
- Checkout updates the index and wd to match



Modifications

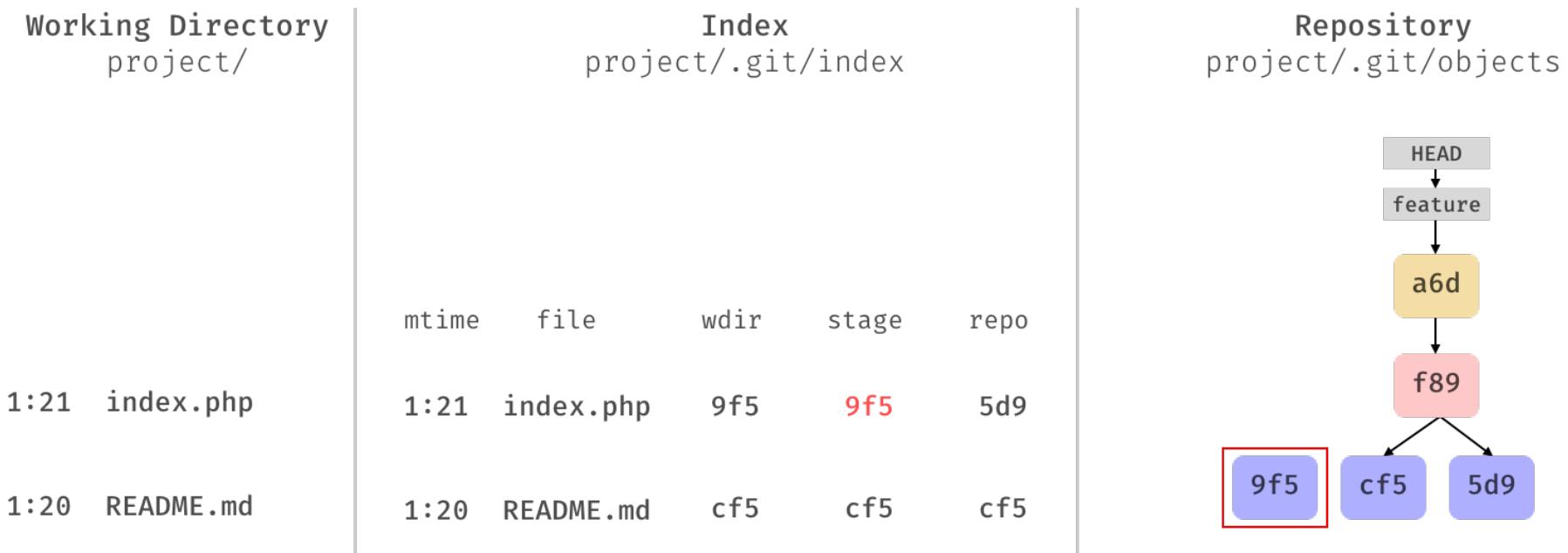
- Changes are recognized by git once you run `git status` -- which updates .git/index

Working Directory project/		Index project/.git/index			Repository project/.git/objects	
		mtime	file	wdir	stage	repo
1:21	index.php	1:21	index.php	9f5	5d9	5d9
1:20	README.md	1:20	README.md	cf5	cf5	cf5



Add

- Once you `git add` the change
- Git creates an object for it in `.git/objects`
- And update the `.git/index`



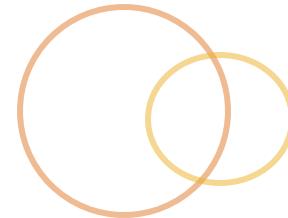
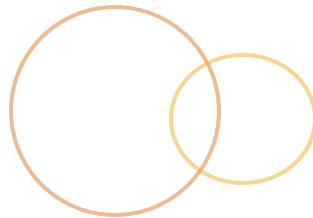
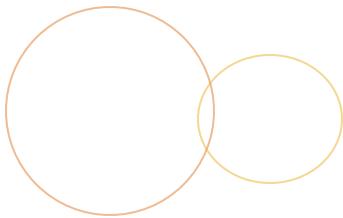
Commit

- Once you `git commit`, git
 - creates the commit and tree object
 - updates the branch pointer
 - updates the .git/index

Working Directory project/		Index project/.git/index					Repository project/.git/objects	
		mtime	file	wdir	stage	repo		
1:21	index.php	1:21	index.php	9f5	9f5	9f5	HEAD ↓ feature ↓ 536 ↓ 32b ↓ 9f5 cf5 5d9	a6d ↓ f89 ↓ 9f5 cf5 5d9
1:20	README.md	1:20	README.md	cf5	cf5	cf5		

Go deeper

- [Follow the trail](#)
- [The format of index per git docs](#)
- What I just ran through: [Understanding Git Index](#)
- [Introduction to Git \(Internals\)](#)
- Great video!



review

TEST YOUR BRAINS

Test: What does each command do?

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

- ⌚ git init
- ⌚ touch junk.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

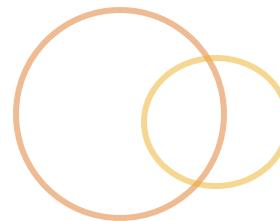
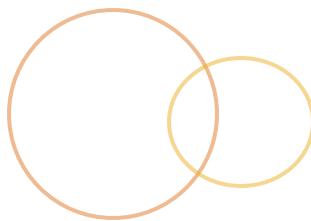
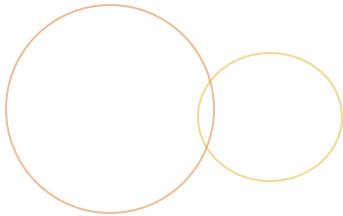
Test: What does each command do?

- git branch ticket55
- git show ticket55
- git checkout 5d23ef
- git show 5d23ef
- git tag -a v1.5
- git rm debug.log
- git show HEAD
- git show HEAD^
- git show HEAD^^
- git reset HEAD^

Test: Map the repository graph

- In a brand new repository...
- We create three commits
- We're on... the master branch
- Make a "bug-5" branch
- Where is HEAD
- Checkout bug-5
- Where is HEAD now
- Add a commit to bug-5
- Where is HEAD, master and bug-5?
- Can we undo it all and go back to the first commit?

**Map the
commits, label
the branches
and HEAD**



module

SHARING THROUGH A REMOTE

What we'll learn

- Repository hosting options / git servers
- What are remotes and how to use them
- Uploading our repository

Remotes

- **Remotes** are *local references to clones of the repository*
 - [GitHub.com](#) / [GitLab.com](#) / Bitbucket
 - Your own git server (Phabricator, Gerrit)
 - Your local system (git init --bare)
- And might include...
 - Workflow for introducing changes
 - Ticketing, automation integrations
- **This is how we collaborate and contribute**

Sharing our repository

- Created an empty repository on a host
- Added a **remote reference** to the hosted repository and labeled it: **origin**

```
git remote add <name> <url>
```

- Pushed local repository to the remote repository

```
git push <remote-name> <branch-name>
```

- Optionally, told git to "track" remote branches by using the **--set-upstream** (-u) flag

Lab: Sharing our repository

Uploading our repositories to a hosting platform

- On our hosting service (github.com)
 - Make sure you have an account!
- Create a new repository
 - Call it "**about-me**"
 - Don't *initialize* it, it is already initialized...
- Follow the directions they give or...
 - Copy the remote repository **url**
 - Then add a **remote** in our local git repository

```
git remote add origin <remote-url>
git push origin master --set-upstream
```

Issues with pushing? Try:

- moving --set-upstream after "origin master"
- disable 2-factor auth
- set up your ssh keys and use the ssh uri

- **List** your remotes
- **View all** your branches (including remote branches)
- Finally, **view** the repository on the hosting platform

Lab: Pushing Local

Uploading our repositories to a bare repository

- Create a bare repository

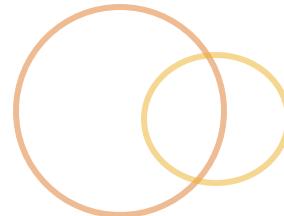
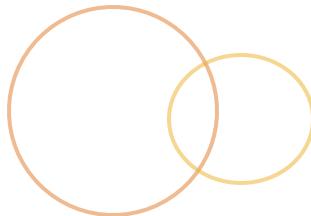
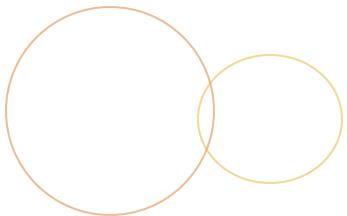
```
cd ..  
git init --bare ./about-me-bare
```

- Push your local 'about-me' to your local bare

- Copy the remote repository **url**
 - Then add a **remote** in our local git repository

```
cd about-me  
git remote add origin ../about-me-bare  
git push origin master --set-upstream
```

- List your remotes
- View all your branches (including remote branches)
- Finally, view the repository on the hosting platform



module

WORKING WITH REMOTES

What we'll learn

- Share work through branches
- Keeping up to date
- Fetch vs Pull
- Sharing tags you've created
- Setting up tracking branches
- General remote management

Push to share (or upload)

push to send branches to our remote(s)

- Typically one branch at a time
- We keep branch names in sync by convention
- Sends *data* necessary to recreate branch ref

```
git push <remote-name> <branch-name>
```

examples

```
git push origin bug-1
```

```
git push upstream master
```

```
git push origin ryan-fix:bug-25
```

```
git push origin :bug-1
```

Pull to sync (or download)

pull updates HEAD (local branch) & repo data

- One branch at a time
- Fetches all the latest data & remote refs
- Merges to the branch you are on

ie: "update my local branch with latest version of a remote branch"

```
git pull <remote-name> <branch-name>
```

for example

```
git checkout master  
git pull origin master
```



Make sure you're on the right branch, it will merge into HEAD

Fetch to update data w/out merging

- Fetch updates remote references

- *It does NOT automatically merge anything*

```
git fetch <remote-name>
```

```
# then maybe update your  
# local branches manually...  
# (pull would have done this)  
git checkout master  
git merge origin/master
```



This is a
remote branch
reference

Remote branches

- When we reference a remote git will also become aware of all the branches in the remote repository

```
git branch --remote  
git branch --all
```

- Remote refs behave like tags, they are not for work

```
git checkout origin/feature-5  
# also OK to merge these...  
git merge origin/feature-5
```

Puts us in
a detached
HEAD

- Checking out a remote branch by name only will auto-create a local copy with tracking

```
git checkout feature-5
```

Remote branch references

Local

Branches:

master
bug1
feature5

Remote Branch Refs:

origin/master
origin/bug1
origin/bug25

Remotes:

origin

Git Database:

data, commits, refs, etc

Origin Remote

Branches:

master
bug1
bug25

Remote Branch Refs:

Remotes:

Git Database:

data, commits, refs, etc

Tracking branches

- Relate a local branch to a specific remote branch
 - Pull, push (and a few other commands) will automatically fill in the branch name(s) for you

```
git checkout bug1
```

```
# before tracking  
git pull origin bug1
```

```
# add tracking relationship  
git branch --track bug1 origin/bug1
```

```
# after tracking  
git pull
```

Set up tracking

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; create local branch that tracks

```
# if origin/bug-22 exists
git checkout bug-22
```

You can view tracking branch info

```
git branch -vv
```

Sharing tags

- Tags need to be explicitly pushed

```
git push <remote-name> <tag-name>
git push <remote-name> --tags
```

```
# you can also ask git
# to push "reachable", annotated tags
git push origin --follow-tags
```

```
# and config git to always do this
git config push.followTags true
```

Tending remotes

⌚ Deleting remote branches (affects remote)

```
git push origin :master
```

or

```
git push origin --delete master
```

⌚ Prune remote branch references (affects local)

```
git fetch origin --prune
```

you can also config it

```
git config fetch.prune true
```

To summarize

Share changes

```
git push origin branch-name
```

Update your master

```
git checkout master  
git pull origin master
```

Update any branch

```
git checkout branch-name  
git pull origin branch-name
```

To summarize (pt 2)

- ➊ (more) update any branch with master's changes

```
git checkout branch-name  
git pull origin master
```

```
git checkout branch-name  
git fetch origin  
git merge origin/master
```

```
# update local master, then...  
git checkout branch-name  
git merge master
```

Lab: Pushing & Pulling (Team of 1)

1. Sharing your work

- In your local **about-me** repository
 - Create a new branch off **master**, call it **upper-name**
 - Edit your <name> file so your name in the file is UPPERCASED
 - Stage, commit, then **push** your branch to the remote

```
git push origin upper-name
```

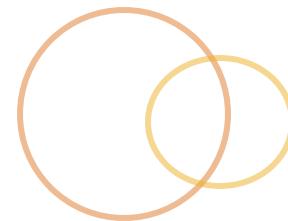
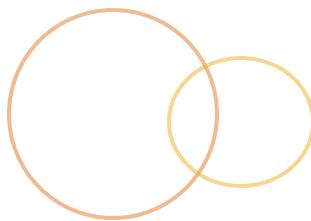
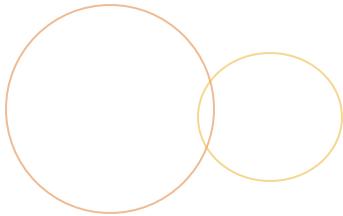
2. Integrating your work

- Create a pull/merge request on Github/Gitlab
 - To the remote's **master** branch
- Merge the PR

3. Keeping up to date

- Update your local master using **git pull**

This is a typical workflow...



step-by-step

REMOTE VS LOCAL

Remotes and branches

LOCAL

master

"remote add origin <uri>"

Remotes and branches

LOCAL

master

origin

Remotes and branches

LOCAL

master

origin

now "git push origin master"

Remotes and branches

LOCAL

master

origin/master

Origin (server)

master

Remotes and branches

LOCAL

master

origin/master

Origin (server)

master

create branch topic-1

Remotes and branches

LOCAL

master

topic-1

origin/master

Origin (server)

master

Remotes and branches

LOCAL

master

topic-1

origin/master

Origin (server)

master

git push origin topic-1

Remotes and branches

LOCAL

master

topic-1

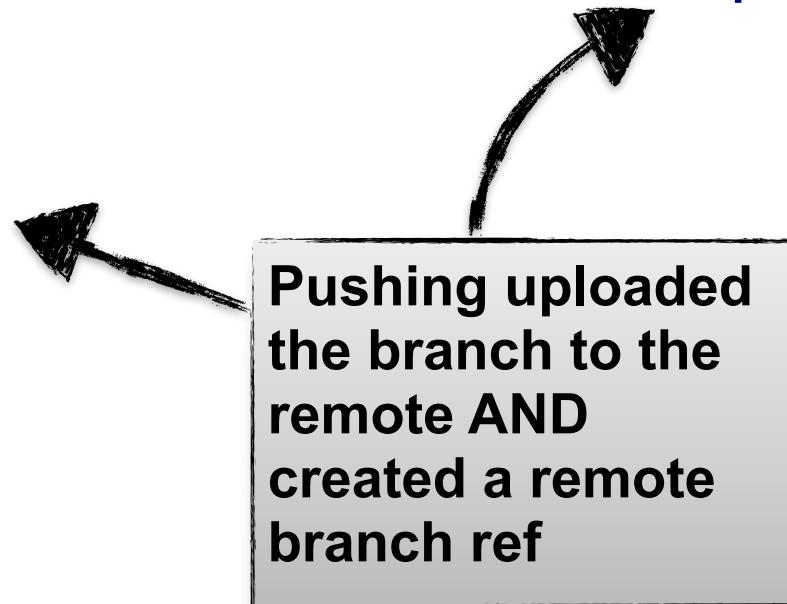
origin/master

origin/topic-1

Origin (server)

master

topic-1



Remotes and branches

LOCAL

master

topic-1

origin/master

origin/topic-1

Origin (server)

master

topic-1

Someone else pushes a branch to origin

Remotes and branches

LOCAL

master

topic-1

origin/master

origin/topic-1

Origin (server)

master

topic-1

topic-20



A wild branch
appears

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



Fetch made our
local repo
aware of the
new remote
branch

Remotes and branches

LOCAL

master

topic-1

origin/master

origin/topic-1

origin/topic-20

Origin (server)

master

topic-1

topic-20

To work on a branch locally, just check it out...

git checkout topic-20

Remotes and branches

LOCAL

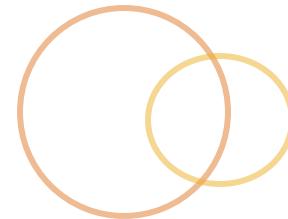
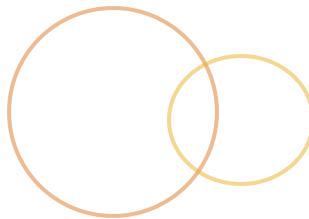
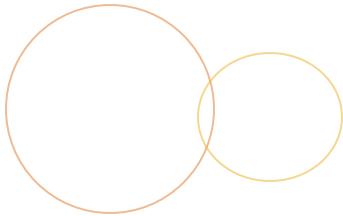
master
topic-1
topic-20
origin/master
origin/topic-1
origin/topic-20

Origin (server)

master
topic-1
topic-20

Git created a
local copy for
us, with
tracking!





module

COLLABORATION IN A SINGLE REPO

What we'll learn

- Intro to a git workflow with a single repository
 - Join - How to clone a repo
 - Do work - What are topic branches
 - Share work - Get it reviewed
 - Stay up to date (next module)
- Keeping up to date

I'll set up a single repo for us to work on as a team (& push my profile)

Join - Cloning

clone copies a repository

- Initializes the repo
- Pulls all data and remote branches
- Sets up an initial remote, called `origin`
- Sets up "master" with tracking

```
# copy the repo  
git clone <remote-url>
```

```
# copy into a specific dir  
git clone <remote-url> <dir>
```

Do work - Topic Branches

- **master** should be stable and production ready
- All **new work** should be done in a **new branch**
- Sometimes referred to **topic branches**, or "working" branch, or "feature" or "bug" or "hotfix" branch.

Share work - Get Reviewed

- ◉ Push your branch to the remote to share
- ◉ Get it reviewed
 - ◉ merge or pull request (hosted reviews)
 - ◉ or another review process / platform
- ◉ Get it merged
- ◉ Share early
- ◉ Your team should define:
 - ◉ review expectations; who does it?
 - ◉ merge expectations; who does it?
 - ◉ any tests/CI?
- ◉ Delete branches when done (local and remote)

Lab: Share your work

Start in a fresh directory - not in a previous repository!

○ **The task:** Add your profile page to the project

- Just a file w/ a list of your favorite things, name the file after you

○ **Clone the about-us repository**

```
git clone <git:url>
```

○ Create a new **topic branch** off master

- Add your profile, stage, commit, etc...

○ Share your branch by **pushing** it to the remote repository

○ On GitHub/Lab, open a **pull/merge request** to request that your new branch be merged into **master**

Don't merge...

We'll review & merge pull requests together

Keep up to date

○ Keep master up to date (typically)

```
# make sure you're on master  
git checkout master  
git pull  
# or, being explicit  
git pull origin master
```

○ Starting new work off master

```
git checkout master  
git checkout -b feature-25
```

Now I'll add an index page for us to all collaborate on...

Keep topic branches up to date

- Occasionally you'll want to update your topic branches

- Grab the latest changes/fixes

- Incorporate a teammate's changes

```
# always make sure you're on the branch  
git checkout feature-5
```

```
# incorporate changes  
# from master  
git pull origin master
```

```
# or do the merge manually  
git fetch origin  
git merge origin/master
```

```
# or incorporate changes from your teammate?  
git pull origin feature-5
```

Lab: Update & Collaborate

- **The Task:** Add your name to the list in the `index.html` file

- First: Make sure `master` is up to date

```
# use either  
git pull  
# or  
git fetch origin  
git merge origin/master
```

- Then: Do the work

- Branch off `master`, use a *branch name that won't collide with your team*

- Finally: Share your work

- Push your branch
 - Create a pull/merge request (DO NOT MERGE)

We'll merge later... I'll also add one more pull request to the mix...

Dealing with conflicts

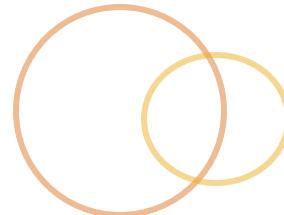
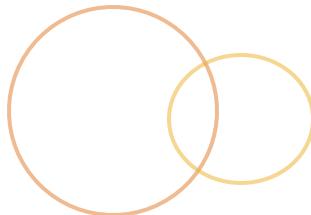
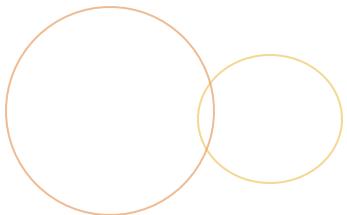
- When you get a conflict
 - ie: you can't merge your pull/merge request
- Merge the target branch into your topic branch
 - This will cause the conflict locally
- Resolve the conflict
 - Stage, commit (to complete the merge, per normal)
- Push your updated topic branch
- Now the request should merge cleanly

I'll create a conflict by being the first to edit the index ;)

Lab: Remote Conflicts

○ The Task: Fix the conflict so your branch will merge cleanly

1. Make sure **master** is up to date
2. Cause the conflict
 - Merge **master** into your local topic branch
 - We are merging in the conflicting changes to cause the conflict locally
3. Resolve the conflict
 - Fix, stage, commit
4. Share your updated branch
 - Push your branch
 - View your original pull/merge request, the conflict should be resolved



module

WORKFLOWS

What we'll learn

- Workflow options
 - Remote organization
 - Branching strategies
- Tag strategies

Team Workflows

- Centralized Repository
 - + trunk-based
- Feature Branch
- GitFlow
- Forking
- Mix & Match...

Centralized

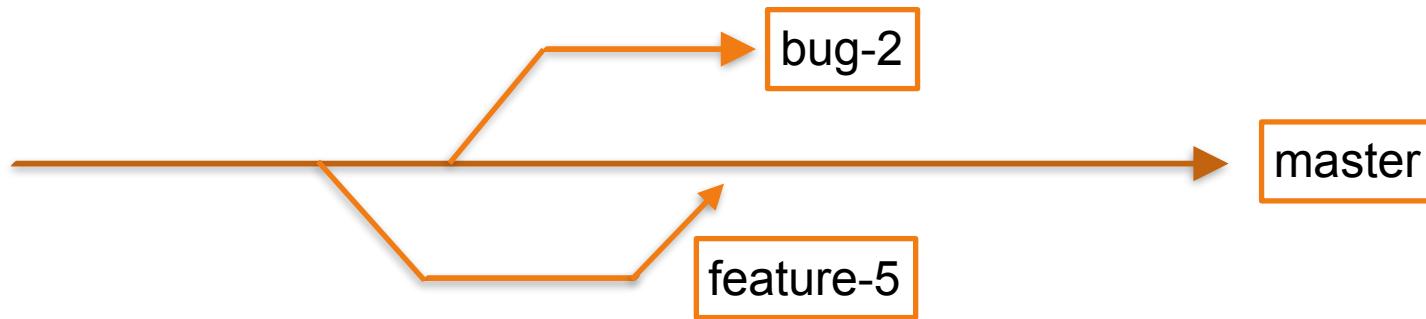
- Remote: One, primary
- Branches: One, master
- Everyone works on and pushes to master



- Might be rebasing to get updates
- Deal with conflicts frequently
- Might be sharing changes in singular "commits"
- ex: Gerrit

Feature Branches

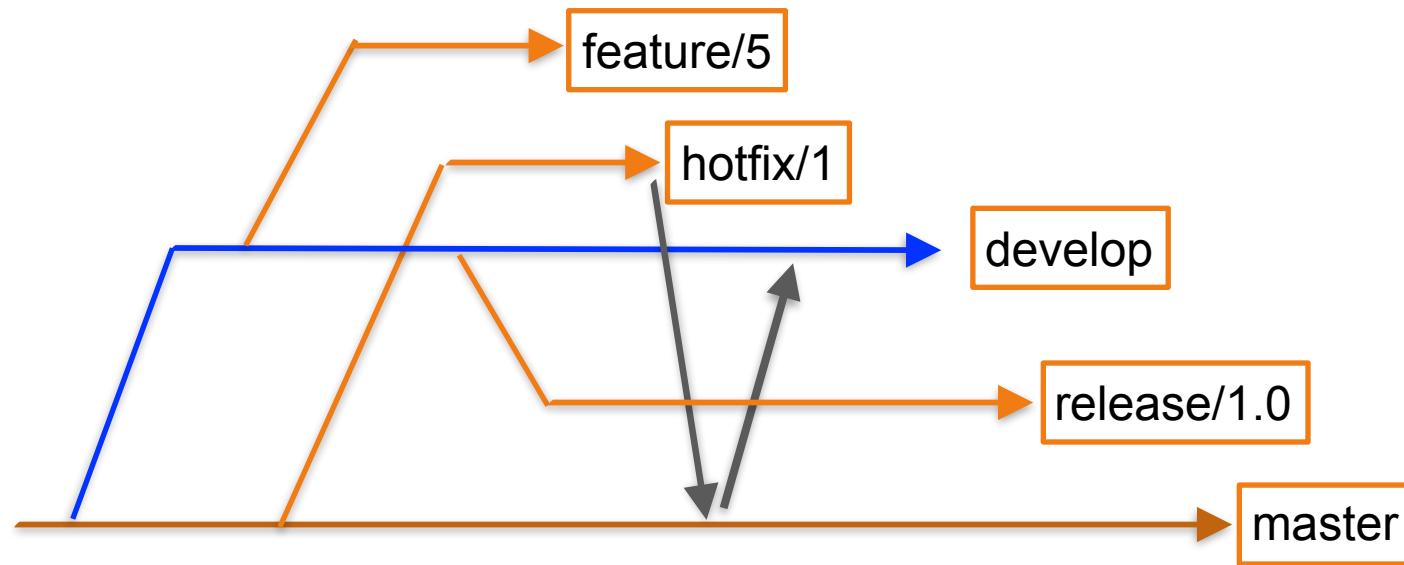
- Remote: One, primary
- Branches: master and many feature branches
- Everyone branches off master and works to get branches merged back into master



- Try to keep feature branches small, short-lived
- Might use "pull/merge requests" to get work back into master

"Git Flow"

- Remote: One, primary
- Branches: master, develop, features, hotfixes, releases
- Very rigid approach to branch naming, synchronization and merging



Forking

- Remotes: Many, with one primary everyone is working to get changes into
- Branches: variable; can work with many workflows
- Remotes could represent:
 - One per engineer
 - One per team within a larger team
 - One per module within a large project
 - One per contributor to an open source project

Other flows...

○ GitHub Flow

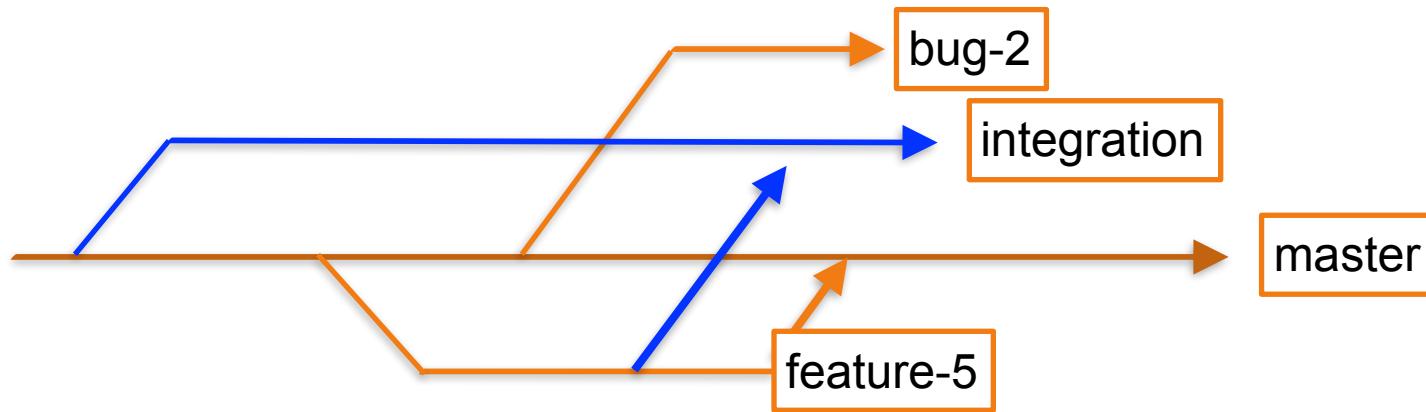
- We've been doing this, super simple
- Great for early and frequent collaboration
- Great for frequent deployers

○ GitLab Flow

- Advocates for a 'production' branch that is stable, while master is a test bed
- Encourages environment-specific branches

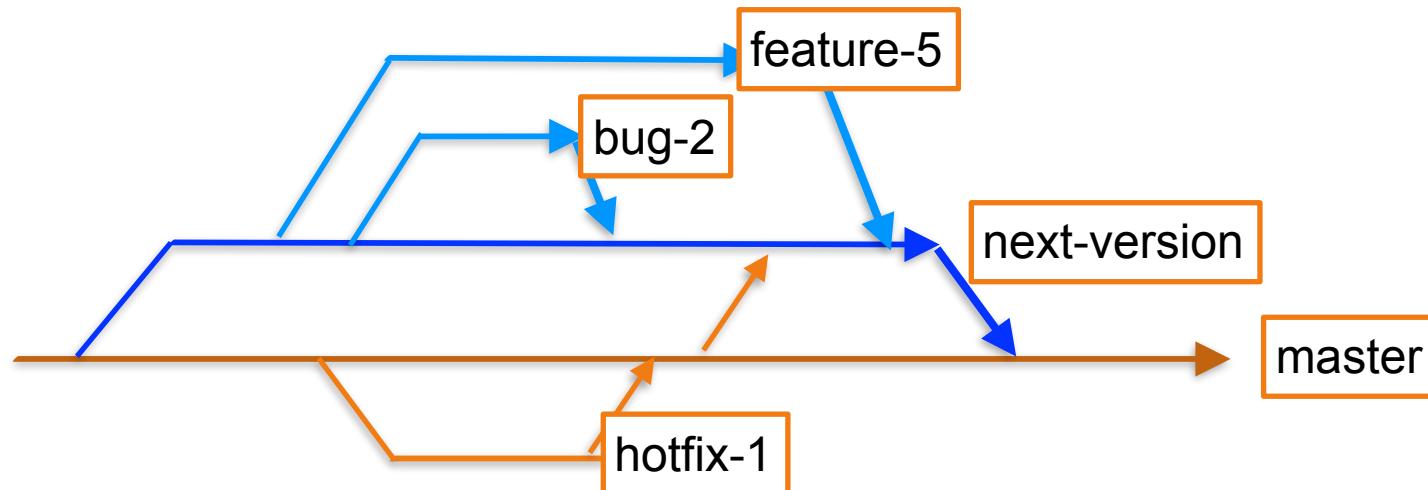
Testing-focused branch

- Branch off master, merge and test on integration, then merge topic branches into master once they pass tests, deploy master



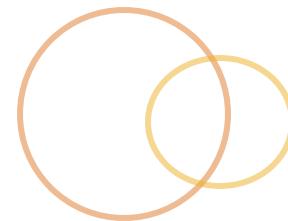
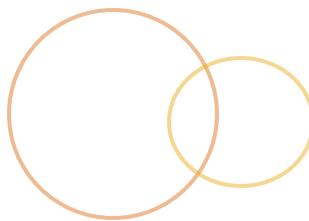
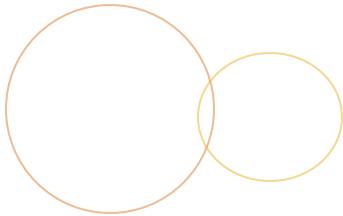
Next-version/release focused branch

- Branch off version branch, merge into version branch, merge full version branch into master once complete and deploy master.



Which workflow is right for me?

- Do you need a super stable master?
- Do you deploy frequently or in batches?
- How many environments do you release or deploy to?
- How do you want to be searching for bugs, sifting through the log?



module

COLLABORATION ACROSS MULTIPLE REPOS

What we'll learn

- What is a fork and when would you use it
- What does upstream mean
- How to collaborate with forked repositories
- Potential repository fork strategies

Fork a repository

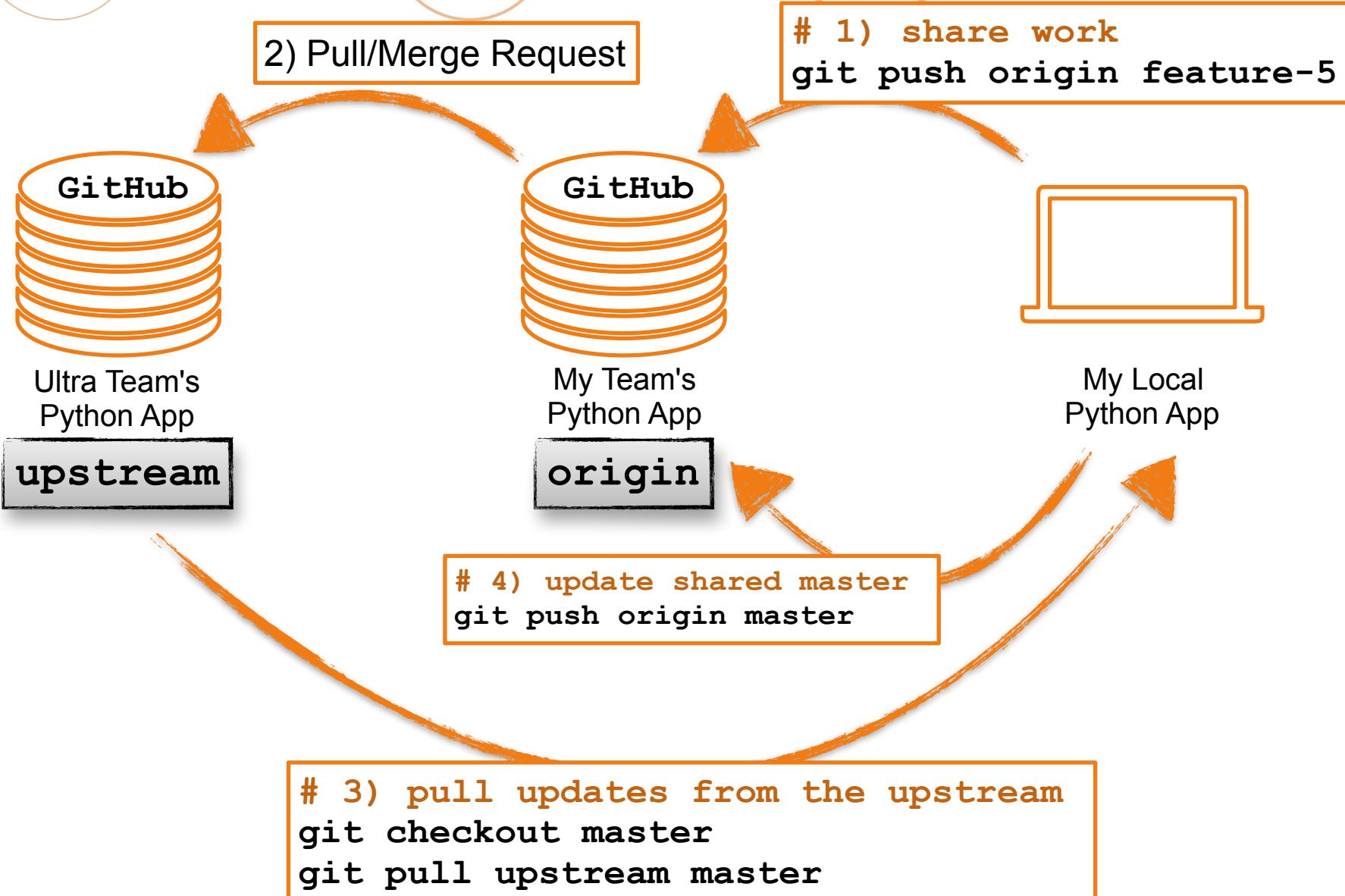
- A **fork** is a **copy** of a repository that remains within your hosting account
 - You can fork any public project in GitHub/Lab
 - It's a copy you own
- Why fork?
 - To submit changes to a public/open source project
 - Organizational safety net for larger teams
 - To split up a project or codebase amongst smaller teams



Fork Workflows

- The main repo is referred to as the **upstream**
- While your personal fork is the **origin**
- You **push** branches into your **origin**
- **Pull request** into the **upstream**
- And **pull** updates from the **upstream**

Fork Workflows



Lab: Working with forks

Share work

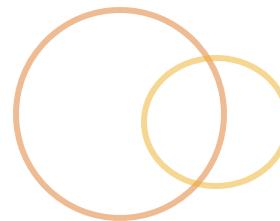
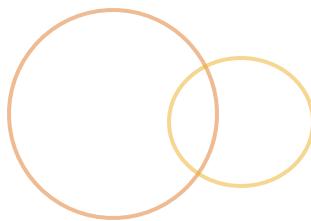
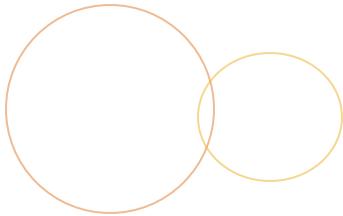
- Fork my repository
- Clone **your fork** to your local
- Submit a change
 - Branch off master
 - Make your edit(s) locally, stage, commit, etc...
 - Push to your **origin**
- Create a pull/merge request *into the main repository (mine)*

I'll begin merging...

Stay up to date with the fork

- Add a new remote for the main repo, call it **upstream**
- Update your local master

```
git remote add upstream <url>
git checkout master
git pull upstream master
```



module

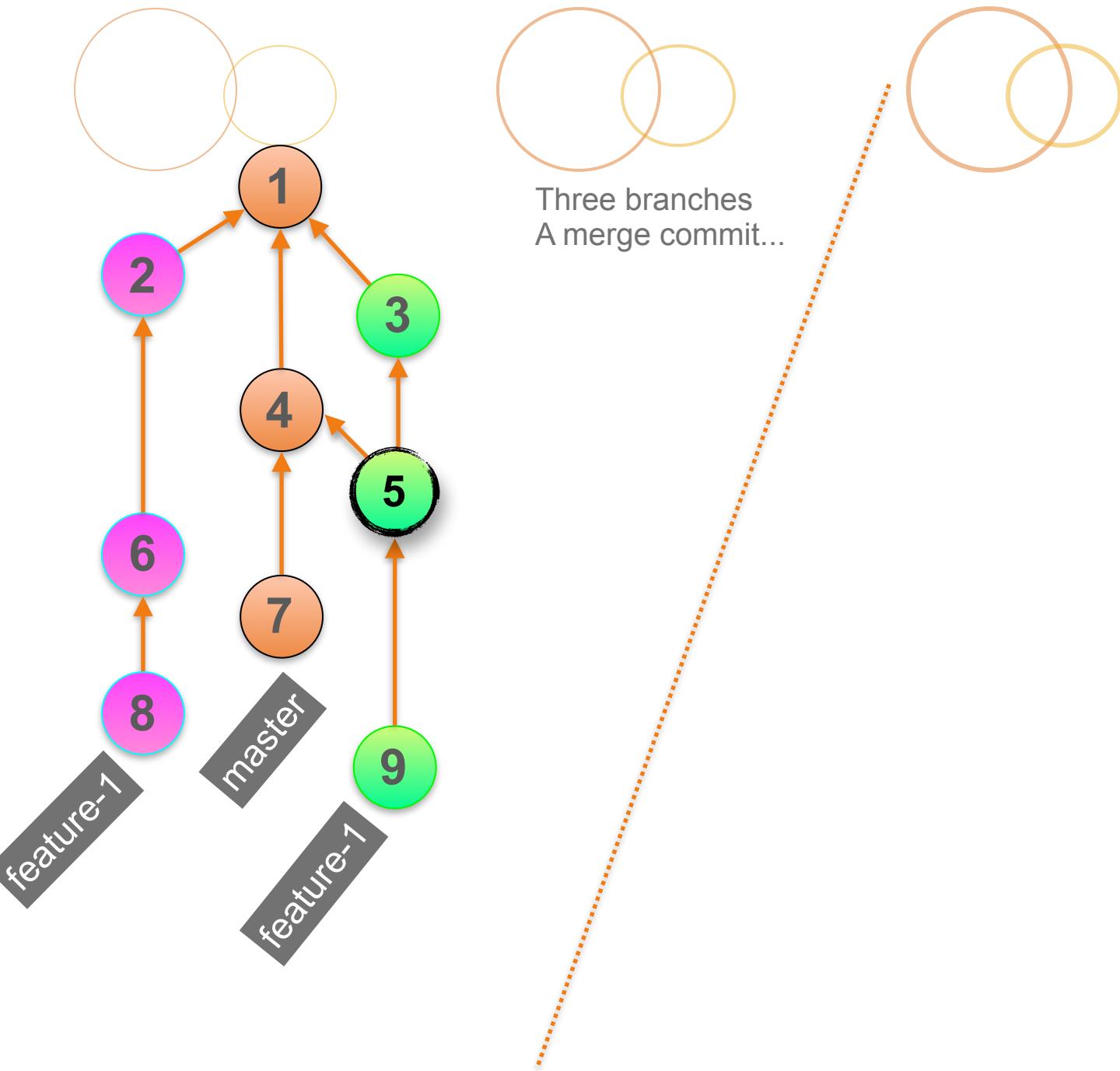
MANAGING HISTORY

What you'll learn

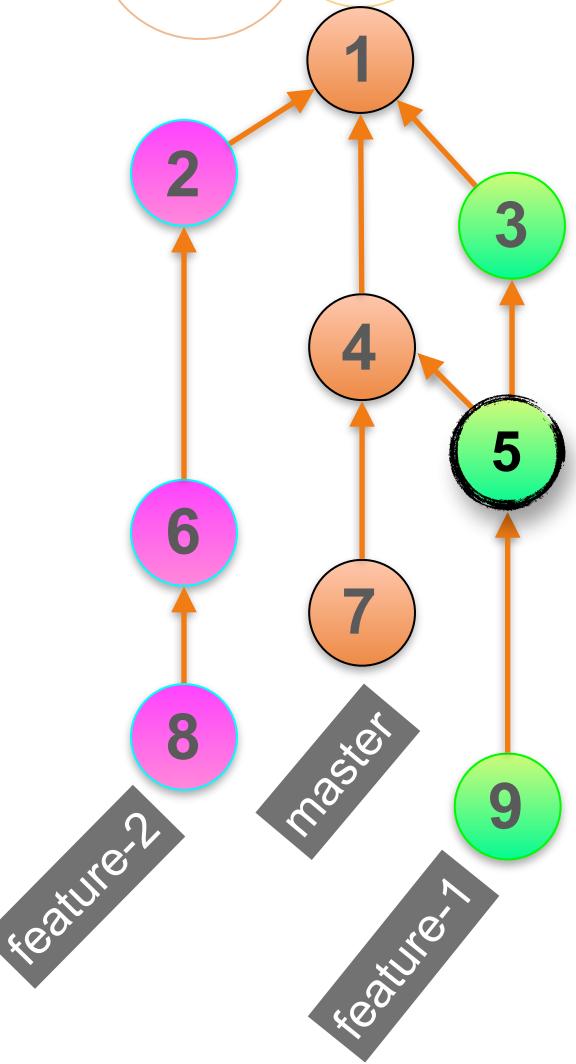
- Why bother *managing your repository history*
- How to **rebase** to stay up to date
- How to **squash** many commits into one
- How to **cherry picking** one or more commits

Manage History?

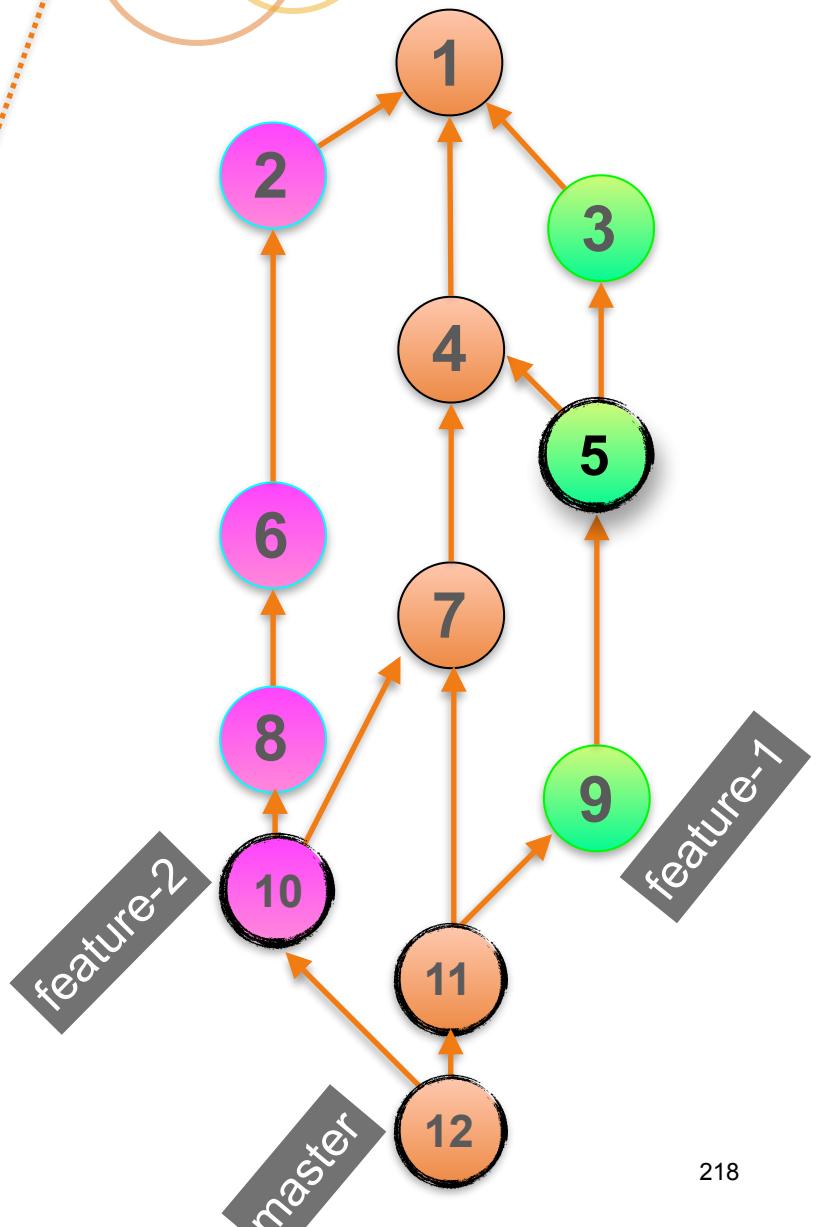
- ◉ Some teams want control over their graph
 - ◉ Keep commits in sequence
 - ◉ Reduce "merge noise"
 - ◉ Such as merges when someone is keeping up to date
 - ◉ Reduce number of commits
- ◉ Makes it easier:
 - ◉ to **debug** when looking at the log
 - ◉ easier to **hunt down bugs**, potentially
 - ◉ to **see related work** happening in sequence
 - ◉ to **see only meaningful work** (not mistakes)



Confetti Graph

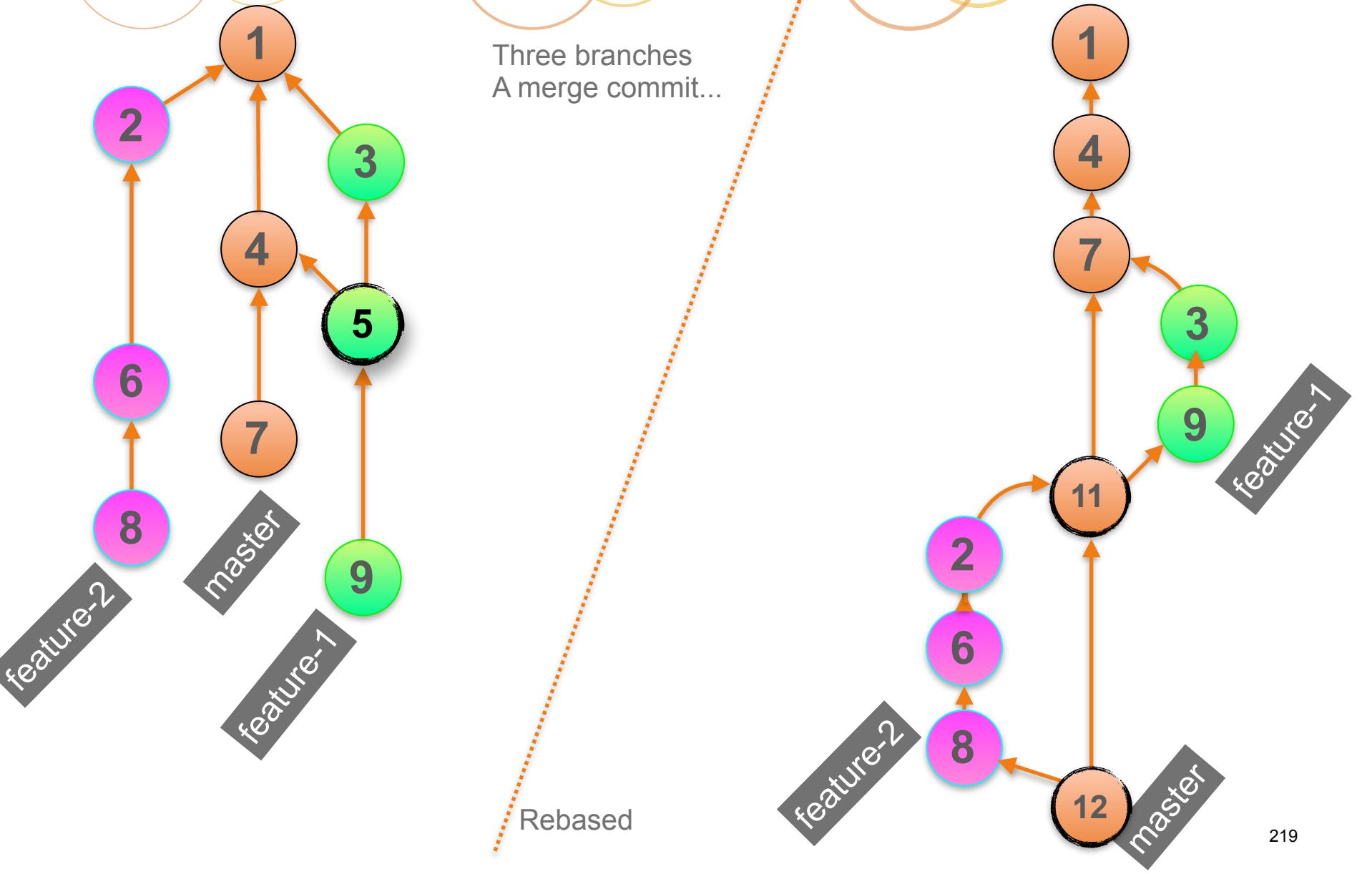


Three branches
A merge commit...



Confetti graph

Rebasing



Ways we control history

- Control the **merge type** that happens

```
git merge --no-ff <branch>
```

- Avoid merges altogether by **rebasing** on to master when keeping up to date

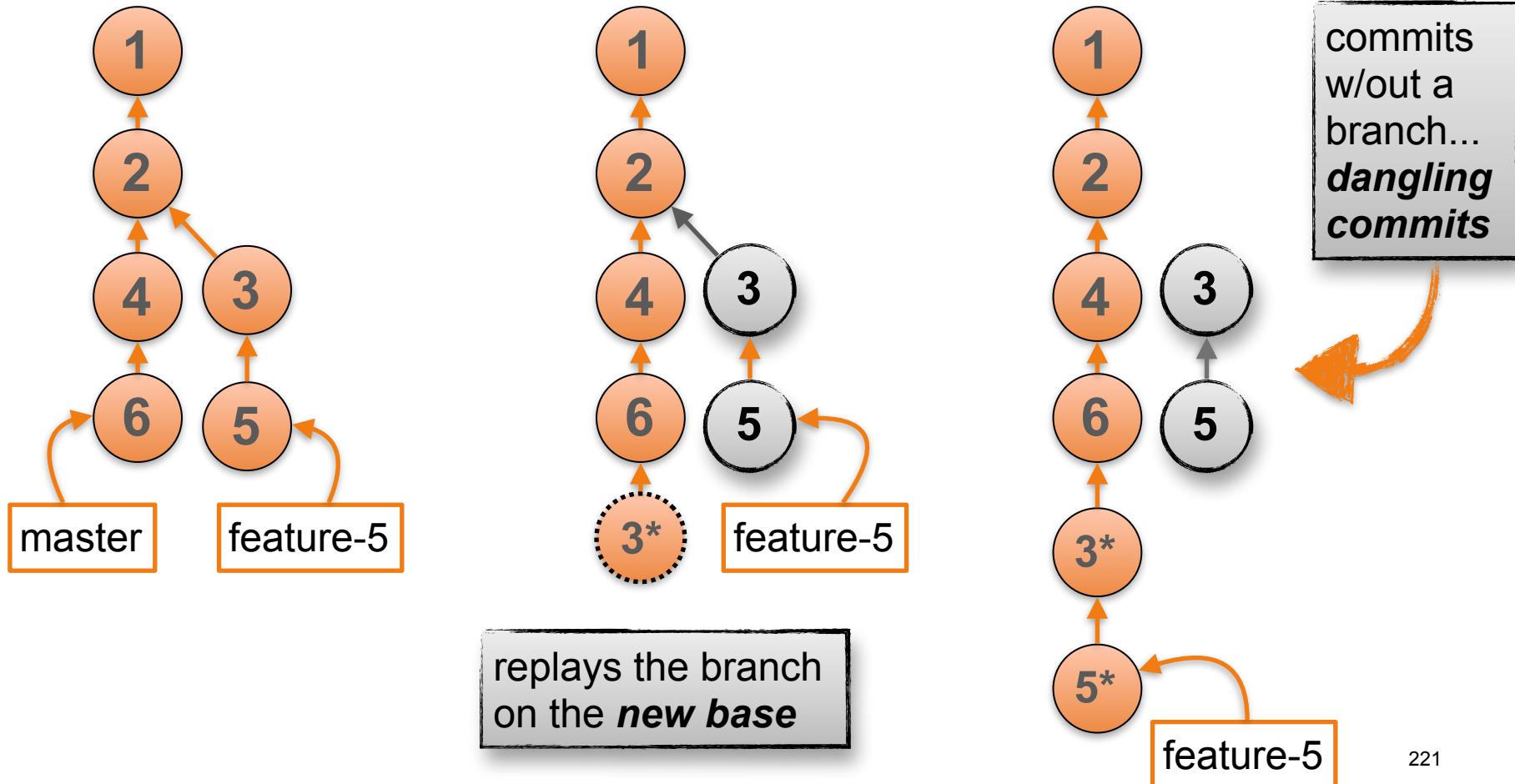
```
git rebase origin/master  
git pull origin master --rebase
```

- Squash** many commits into one when merging into master/releasing

```
git rebase origin/master -i  
git merge feature-1 --squash
```

What a rebase looks like

```
git checkout feature-5  
git rebase master
```



Squash many commits into one

- Combining many commits into fewer/one
 - Squashes into a new commit(s)

```
git rebase -i origin/master
```

- During the rebase we can tell git to
 - "s" squash commits
 - throw away commits
 - stop and let me edit the commit itself
 - stop and let me edit the commit message

Cherry-pick a single commit

- Apply (copy) changes from one commit

```
git cherry-pick 2629dbe
```

Don't alter shared history

- ◉ All these operations create new commit objects
 - ◉ Changes the HEAD of a branch
 - ◉ Duplicate changes, different commit
- ◉ Pushing to previously shared branch will fail
 - ◉ You can force push it but it will cause conflicts for your team if they are using your previous commits

```
# the semi-nuclear option  
git push origin feature-5 --force  
git push origin feature-5 -f
```

Conflicts

- **rebase** and **cherry-pick** can get conflicts
- Fix & stage the resolution then continue...

```
git rebase --continue  
git cherry-pick --continue
```

- Otherwise abort the whole operation

```
git rebase --abort  
git cherry-pick --abort
```

Lab: Changing history

- Clone my repository:

- [`https://github.com/rm-training/history-changer`](https://github.com/rm-training/history-changer)

- Check the log (`--all --graph --decorate`)

1. Using rebase to bring in updates

- Update the `topic-behind-1` branch with changes from master by using `rebase`

2. Using rebase to squash commits

- checkout the `messy-branch` and view the log

- Use `rebase -i` to squash it into one commit

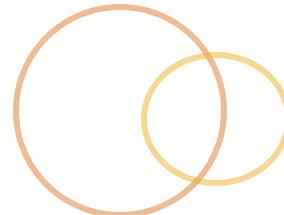
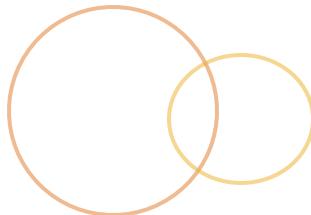
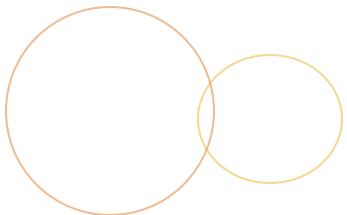
- Give it a more meaningful commit message

3. Cherry pick a commit

- checkout the `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



module

FIXING ISSUES

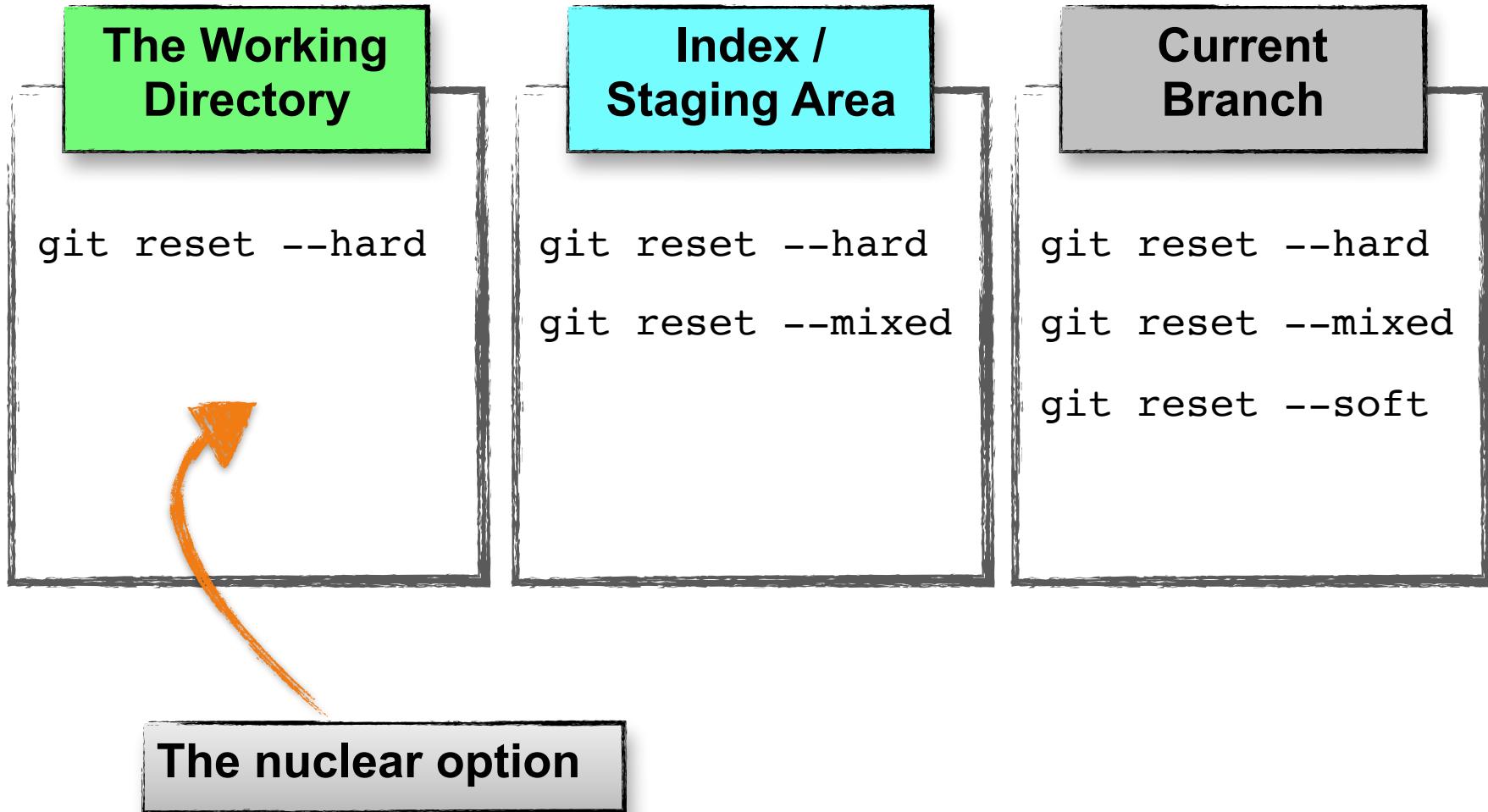
Reset - The utility knife

- ◉ **HEAD** keeps track of the branch you are on
- ◉ **Reset** can be used to modify branch pointers
 - ◉ Whereas **checkout** changes where HEAD points to
 - ◉ **Reset** changes which commit a branch points to

```
# reset HEAD to...
git reset HEAD~2
git reset 55d932f
```

```
# reset or checkout file to...
git reset 55d932f -- readme.txt
git checkout 55d932f -- readme.txt
```

Git Reset & the three trees



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`

Destroyed master?

Destroyed master?

Delete it and checkout a fresh copy

```
git checkout -b temp-branch  
git branch -D master  
git checkout master
```

Destroyed a branch?

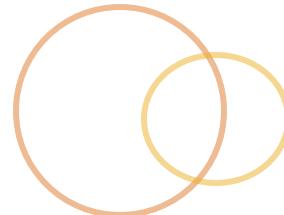
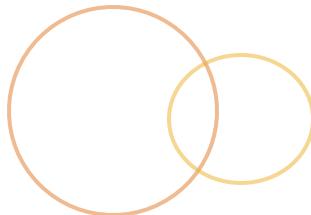
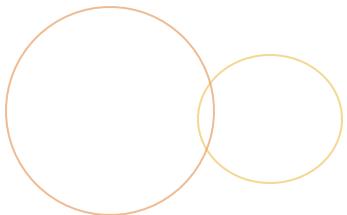
Find the "correct" commit and use git reset

```
git checkout branch-you-are-fixing  
git reflog  
git reset --hard <commit id>
```

Review common practices

1. Your master is all kinds of messed up
2. You want to work on your teammate's branch
3. Your branch won't merge cleanly into master
4. Your teammate has a change that you want to use in your branch
5. You start your day; working on a feature you've been building for several days
6. Your branch has work you want to keep, but it also has lots of other junk commits or changes...

Any other common scenarios, gotchas or questions?



module

MORE WITH GIT

Other commands

- git blame
- git gui
- gitk
- git bisect
- git mergetool
- git difftool

Commit Message Templates

- Agree on what is expected in a commit message

./gitmessage

```
# 50-char summary
```

```
# detailed explanation:
```

```
Resolves:
```

```
See also:
```

- Set up a template file and tell git about it

```
git config --local commit.template ~/.gitmessage
```

Hooks

- Trigger functionality during different git events
 - Pre/post commit, push, pull, etc

```
# take a look at the samples  
ls -la .git/hooks
```

- Local or Server-side hooks
- Our hosted repos can be set up to respond to branch updates, triggering a build, test suite, CI, etc...

Sharing hooks with your team

- .git/hooks directory is not version controlled
- So... we have two options:
 - 1.Add a setup script that copies from ./githooks into .git/hooks
 - 2.Define hooksPath in your local config (2.9+)

```
git config --local core.hooksPath .githooks
```

**Don't forget to make
newly added hooks executable**

And more (advanced stuff)

- submodules & subtrees (a portion)
 - repos in your repos
- Reuse Recorded Resolution
-