Git Fundamentals

Ryan Morris

mr.morris@gmail.com

Copyright © 2015-2019 Ryan Morris

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
```

- Download the Slides
  - [tbd]
- Grab a cheat sheet
  - Github's
  - Atlassian's

Hello

- I'm Ryan Morris
  - Reach me at mr.morris@gmail.com
    - https://www.linkedin.com/in/mrmorris/
  - Engineer with 15 years experience in start-ups
  - I am a **builder**, **leader**, **defender** and **mentor**
- You?
  - Any experience with the **command line**?
  - Any experience with **version control**?
  - Any experience with **Git**?
  - What are your **goals** for this class?

Class style

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

Goals

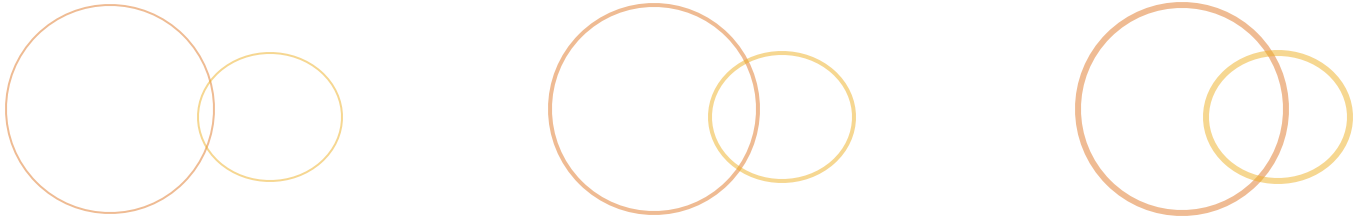
- This class is geared towards **beginners**
  - Comfort the **command line** (no GUI)
  - Deal with **typical frustration points**
  - **Exposure** to a broad range functionality
- Sprinkle of intermediate and advanced topics throughout

What we'll cover

- Today
  - Creating & Managing repositories
  - Staging & Committing
  - Branching & Merging
  - Dealing with conflicts
- Tomorrow
  - Remotes
  - Workflows
  - Managing History
  - Searching & Debugging

Resources

- The Git Parable
  - http://bit.ly/1isB3K4
- Pro Git, 2nd edition (for free!)
  - http://git-scm.com/documentation
- Visualizing Git
  - http://pcottle.github.io/learnGitBranching
- Sublime Merge
  - https://www.sublimerge.com/



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
- *And maybe… Automate operations like deployments, etc*

What things can we version control?

What things can we version control?

- **Any files!**
- Typically defined at a top-level directory
  - And includes everything underneath it

/My Documents/projects/web-scrawler/

This directory and all the files & folders inside it

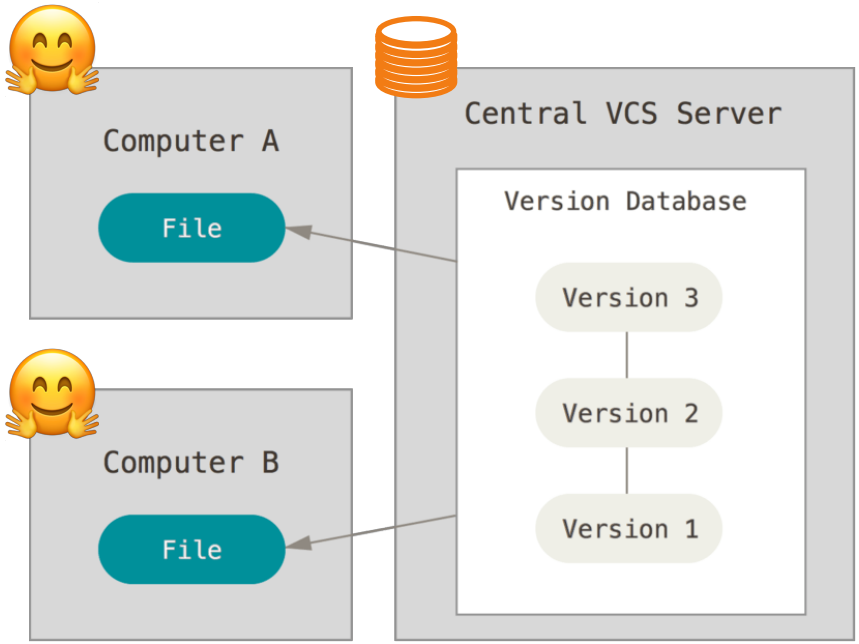
- So...
  - The code for our website
  - My photo collection
  - My mp3 collection
  - My entire computer
  - These slides
  - Configuration files shared across a team

Stone-age version control

just me!

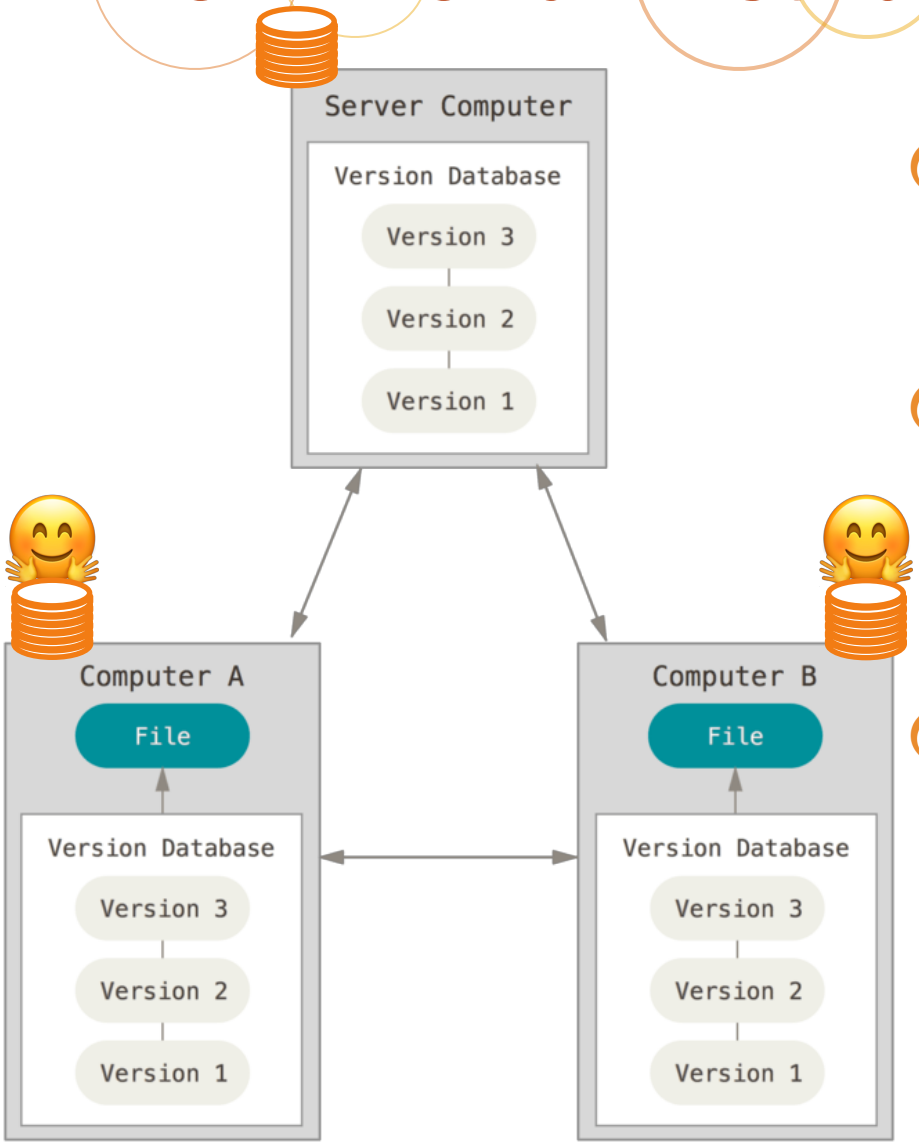
- A new file for each meaningful change
  - `/my-files/myfile.v1`
    `/my-files/myfile.v2`
    `/my-files/myfile.v3.draft`
- Advantages
  - Easy
- Disadvantages
  - Error prone
  - Single point of failure

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

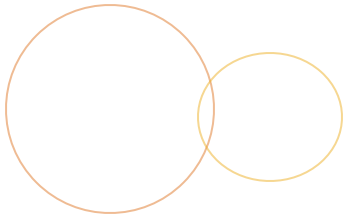
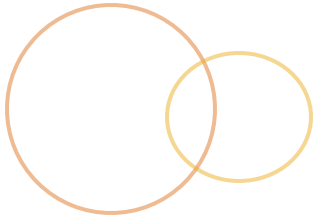
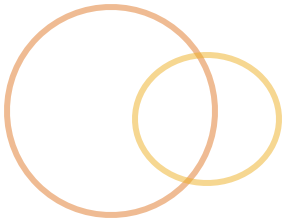
New World: Distributed Version Control



- 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
  - **git** created in 2005 after relationship w/ bitkeeper broke down
- Goals
  - speed
  - simplicity
  - support for non-linear development
  - distributed
  - support for large projects

module

ABOUT GIT

What makes Git different

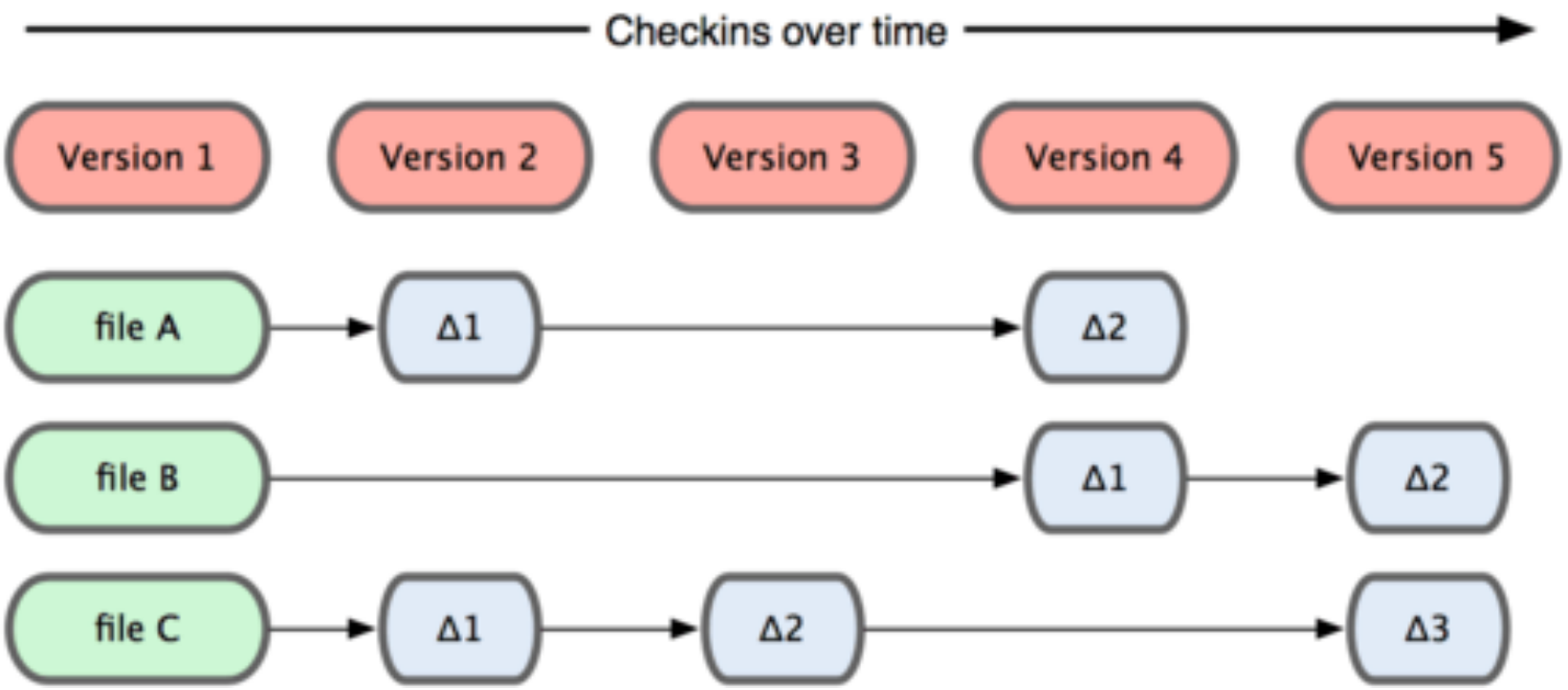
"Just a simple key-value data store"

- Git stores **snapshots**, not differences
- Nearly every operation is **local**
- **Data integrity** through hashing
- **Branching** is easy and encouraged

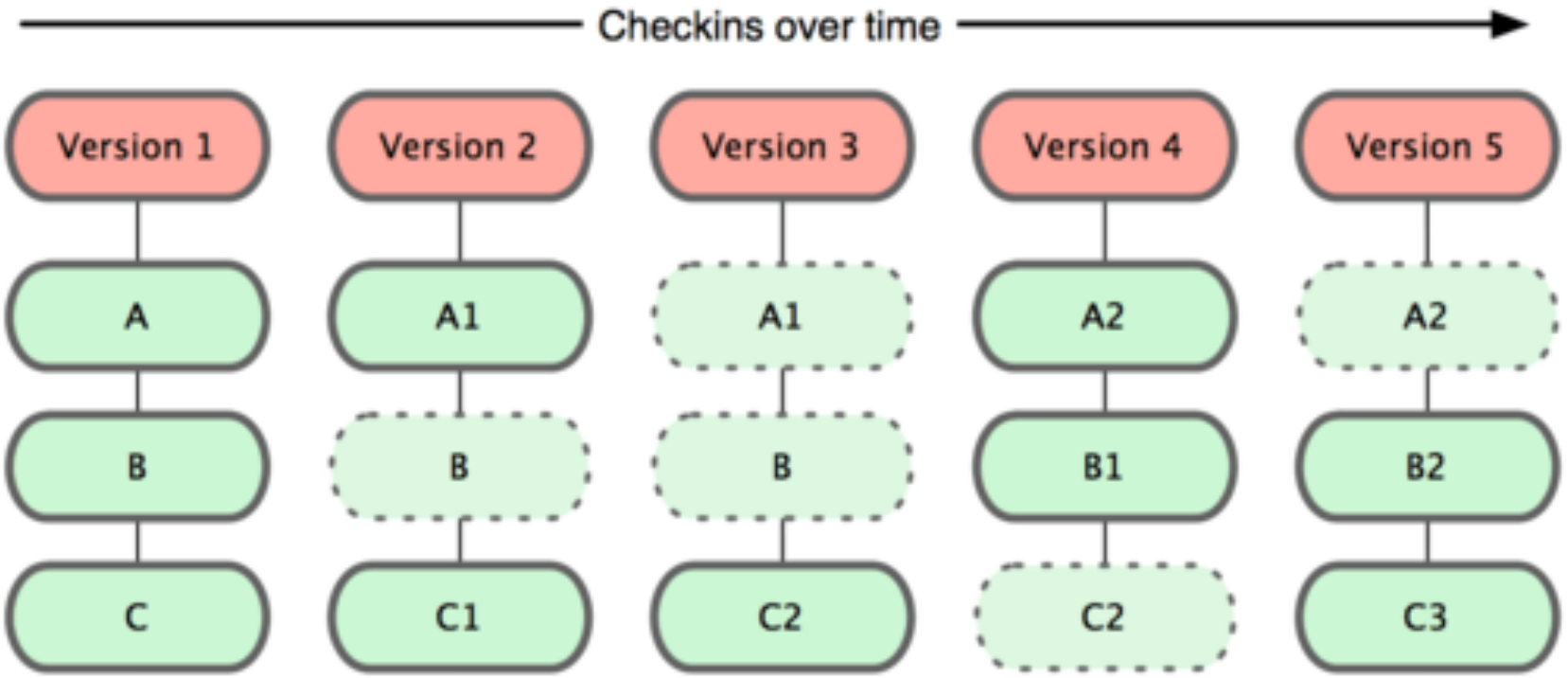
Git commits

- Each **commit** represents a "snapshot"
  - Traditional "version"
  - or, Video game save point
  - or, a photo of the folder & files at that point in time

Diffs / Deltas (SVN, etc)



Snapshops (Git)



Commits and Snapshots

/proj

/bin/
/bin/install.sh
/readme.txt
/package.json

I have a project in a folder - if I ***add all the files*** to my repository and commit...

Commits and Snapshots

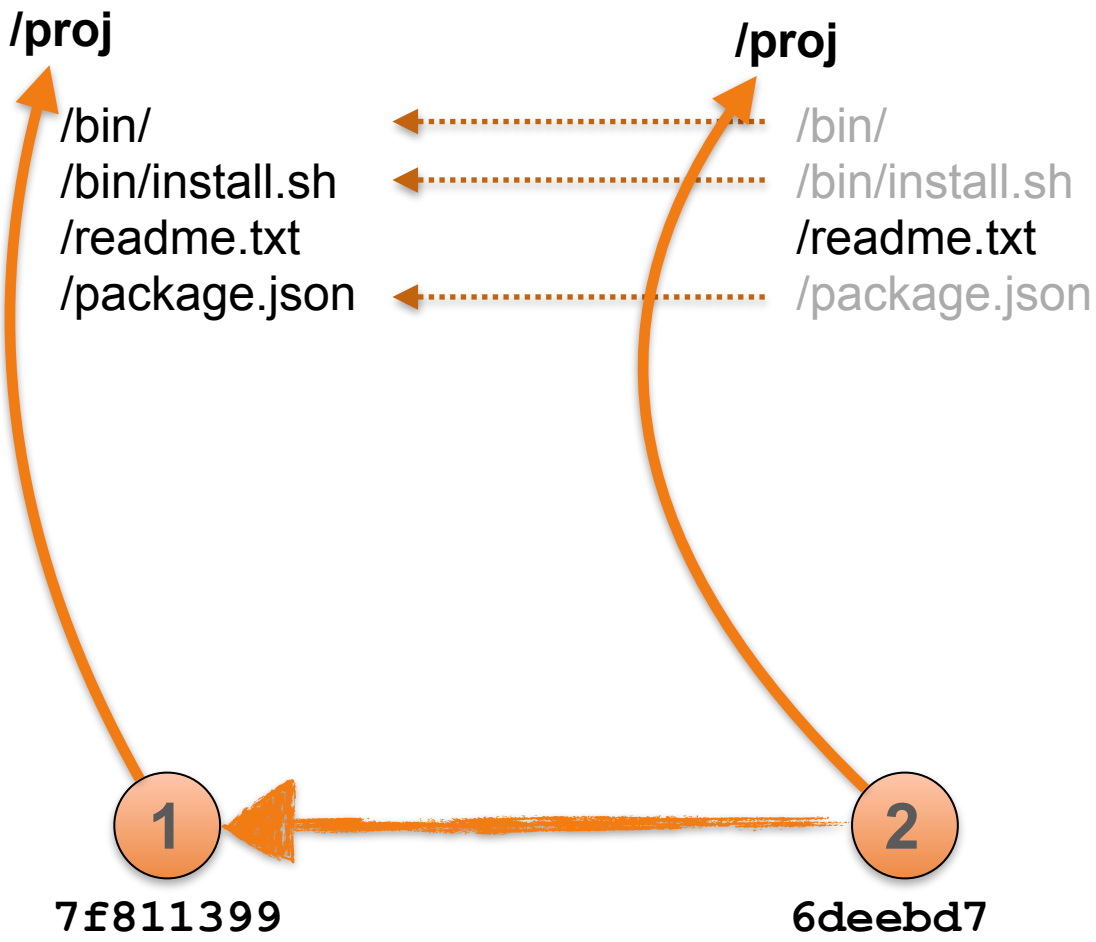
/proj

/bin/
/bin/install.sh
/readme.txt
/package.json

The "commit" represents all the files at the time I committed.

If I ***edit the readme.txt*** file and then commit this into my repository.

Commits and Snapshots



This commit points to a "new" version.

Each file is tracked in every version (think: photo/snapshot, not diff).

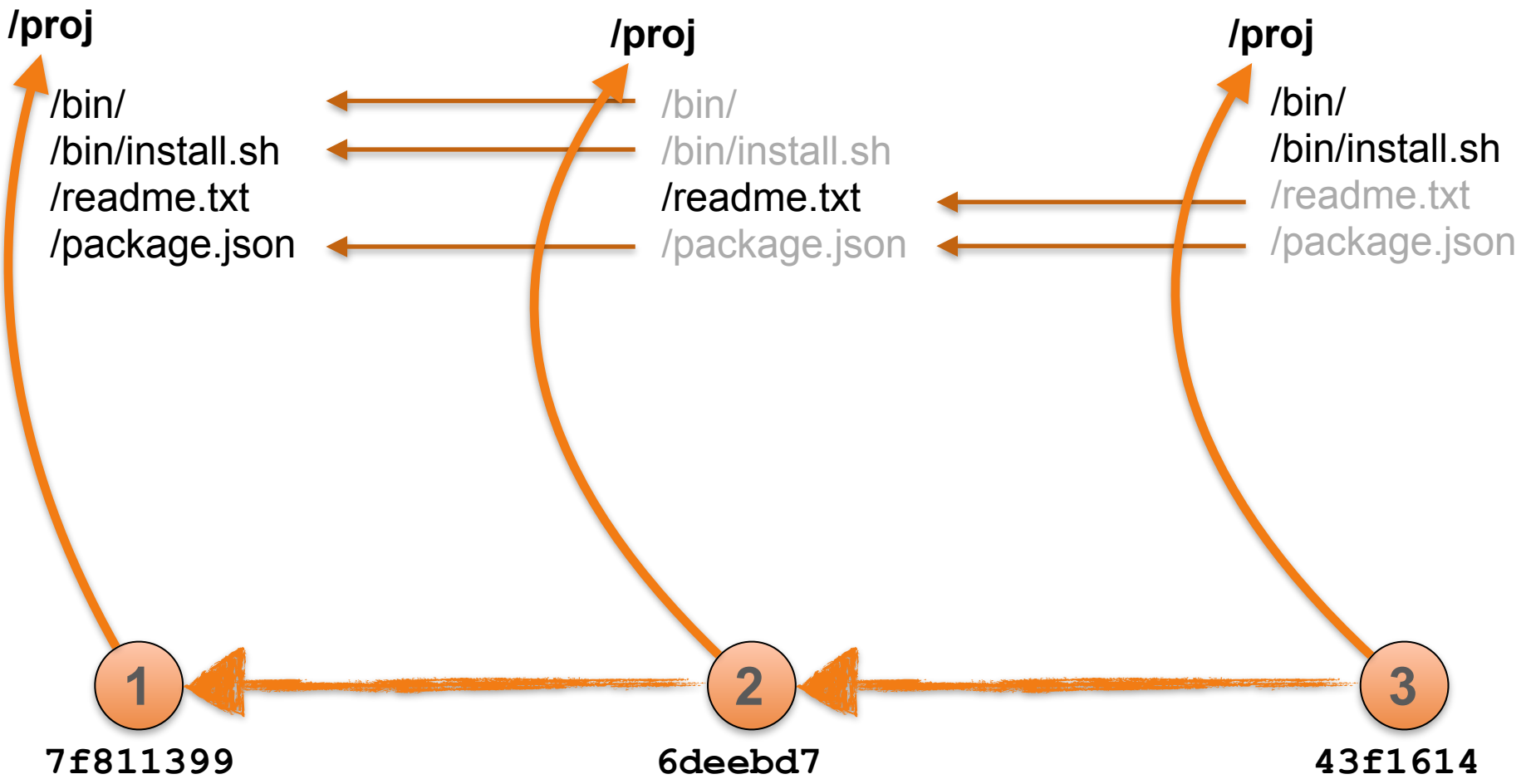
However, files or folders that do not change are not stored twice in the repository database.

Under the hood, files in a commit that did not change will point back to their previous version.

Let's make one more change... ***edit install.sh***

Commits and Snapshots

Notice the directory also "changed" here...

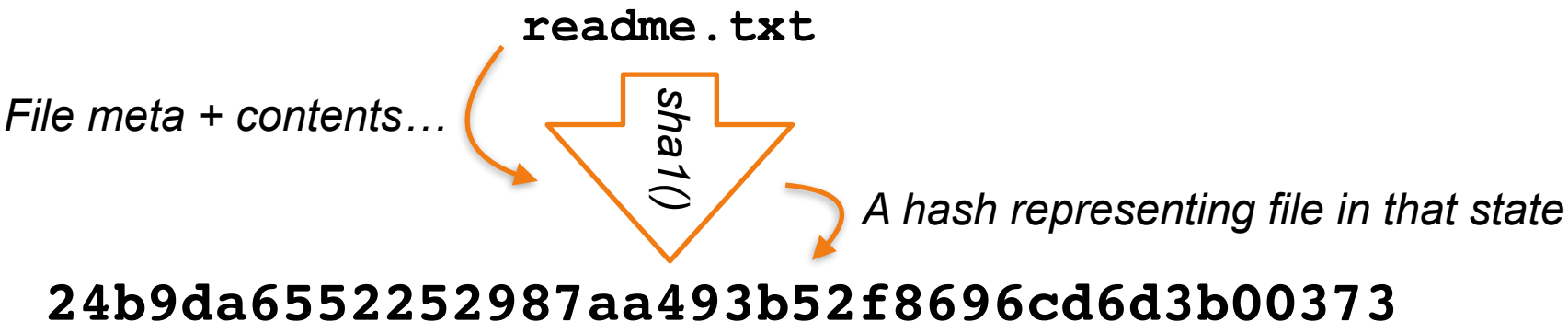


Local operations

- Entire repo is **local**
  - Full History
  - All* branches
  - Commit and Check out
  - No network connection needed
- *Working with **remotes** is the exception*

Data Integrity

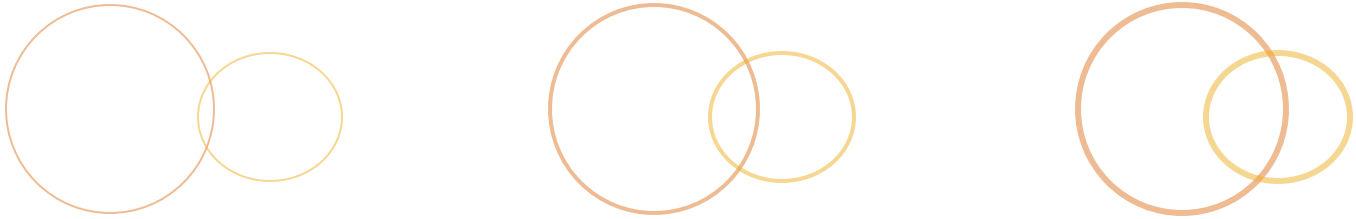
- All things in git are represented by unique hashes
  - Files, directories, commits…



- Changes are tracked in this way
- Tampering becomes obvious

Git generally *adds* data

- Nearly all actions ***add*** data to the Git database
- Almost everything can be ***recovered*** or ***undone***
- As with any VCS, you can lose or overwrite uncommitted changes, but after committing, it's quite difficult to lose anything.



THE COMMAND LINE

Command line basics

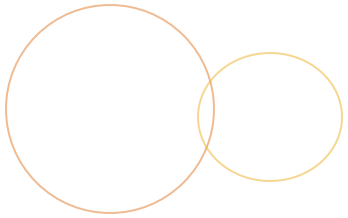
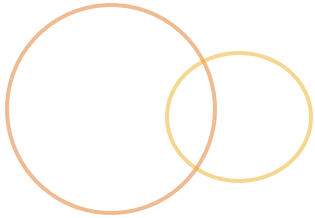
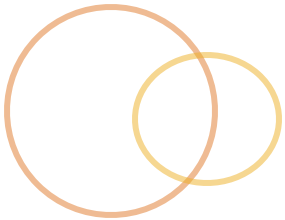
- *How does everyone feel about using the cli?*

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

Install Git

- Install it
  - http://git-scm.com/download/
  - Or… use a package manager like **homebrew**
  - Or… install github's gui
    - https://mac.github.com/
    - https://windows.github.com/
- Make sure it's installed
  - Make sure you're on 1.7-ish or above

```
git --version
```

Git configuration [beginner]

- Set **global** settings that affect all repositories
- Configure a few important values

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

- View **global** settings

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

Levels of Git Config

- Stored as a plain text file
- Three locations we can specify configuration
  - **Local** to *each* repository

    ```
    myrepo/.git/config
    ```

  - **Global** for *all* repositories of one user

    ```
    ~/.gitconfig
    ```

  - **System** for all user's
- Local > Global > System

Config source [advanced]

- Where was that config value set?

```
git config --list --show-origin
```

- Where is that config file!?

```
git config --list --global --show-origin
```

Configure your text editor

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

```
git config --global core.editor <editor>
```

- You can set this to an editor of your choice

```
# set as atom (if you have it installed)
git config --global core.editor "atom --wait"

# test it out
git config -e
```

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"

# Windows Notepad
git config --global core.editor notepad
```

https://help.github.com/articles/associating-text-editors-with-git/#platform-windows

Configure line endings

- If the team works across windows & linux-like systems then you may need to tell Git how to handle the difference between line endings

MacOS / Linux:

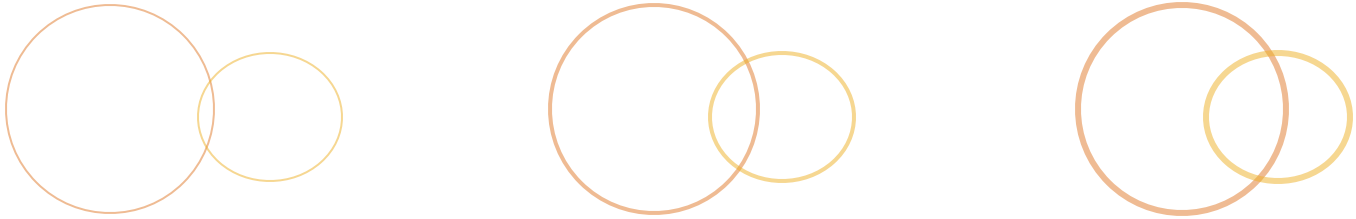
```
git config --global core.autocrlf input
```

Windows:

```
git config --global core.autocrlf true
```

Lab: Setup

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



module

OUR FIRST REPOSITORY

What we'll learn...

- **Creating** repositories
- **Staging** & **Committing** ~~files~~ changes
- **Tracked** vs **Untracked**
- Intro to ***the three trees***
- Peek inside the **Git Database**

I'll demo, no need to follow along...

Creating a repository

- Initialize a repository

```
git init <directory>
```

- More examples

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

What is a git repository?

- The files we want git to "**track**"
  - to retain changes and a history
- Like a "**project**"
- We create a repository in a **directory**
  - Which will then tell git to track the directory
  - And all files within it
  - And all sub-directories that I add to it
  - Forever…

Inspecting our repository

- Check the status and the log

```
git status
git log
```

- Check what git has initialized

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

Our first commit

- Create a file, *stage* it, *commit* it

```
touch README
git add .
git commit -m 'Initial README'
```

- Check the status and the log

```
git status
git log
```

What is a commit?

- A **snapshot** of your filesystem (just the repo)
  - "the state of the files at a point in time"
- Each commit references
  - it's **parent commit**
  - **And** the **top-level directory** at that point in time
- A commit is referenced by it's ***hash, id*** or ***sha***
  - ***42d484c401f0a19cc8a954c16240821329acefac***

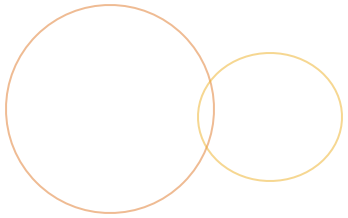
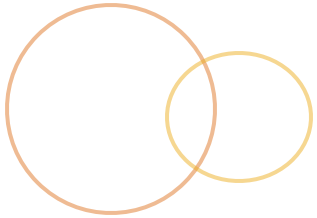
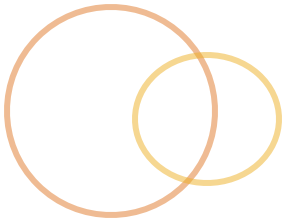
Tracked vs untracked

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

Lab: 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 "Junk" > firstfile`
- Stage the ~~file~~ change
  - `git add firstfile`
- Check the status
  - `git status`
- Commit the change
  - `git commit`
- Check the log
  - `git log`

Don't forget to cd **into the new directory**

module

BASIC FLOW & THE THREE TREES

What we'll learn...

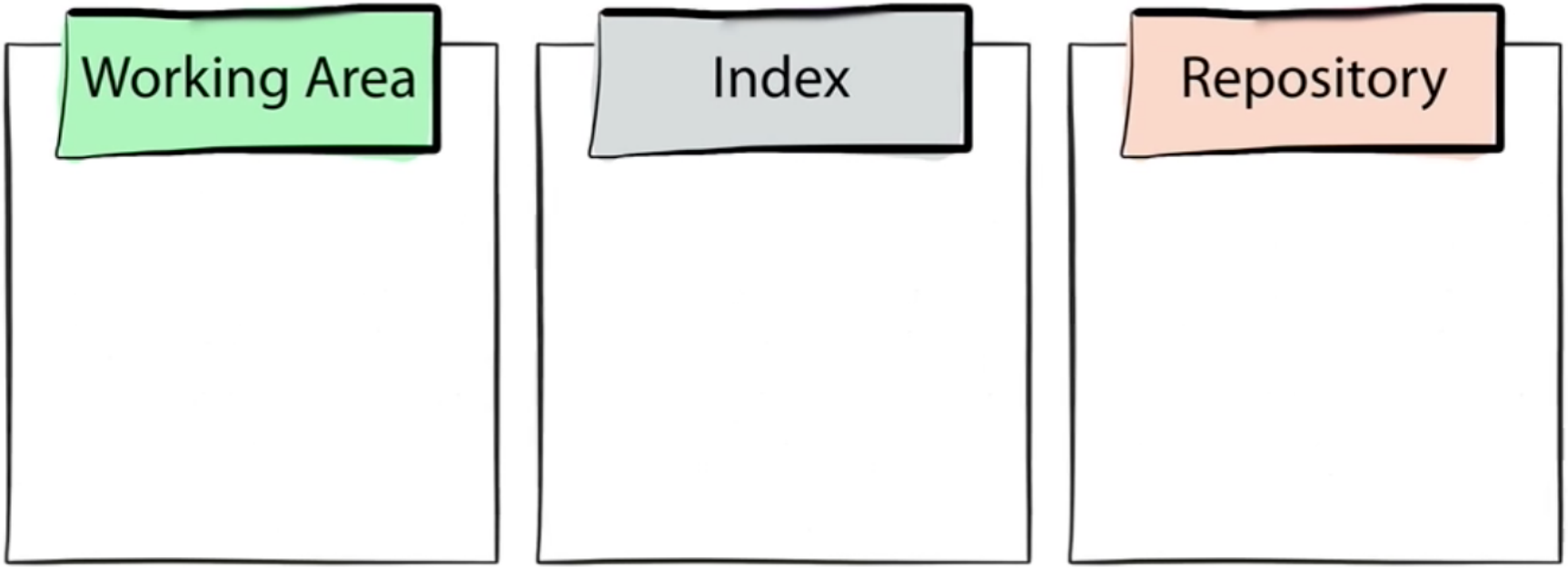
- The Three Trees
- What is staging... why bother?
- Commit shortcuts
- Viewing diffs
- Checking the project's history
- Moving and Removing files with git

Stage, Commit, Rinse & Repeat

- **Staging** changes
- **Commit** staged changes into your history
- Check your **status** often

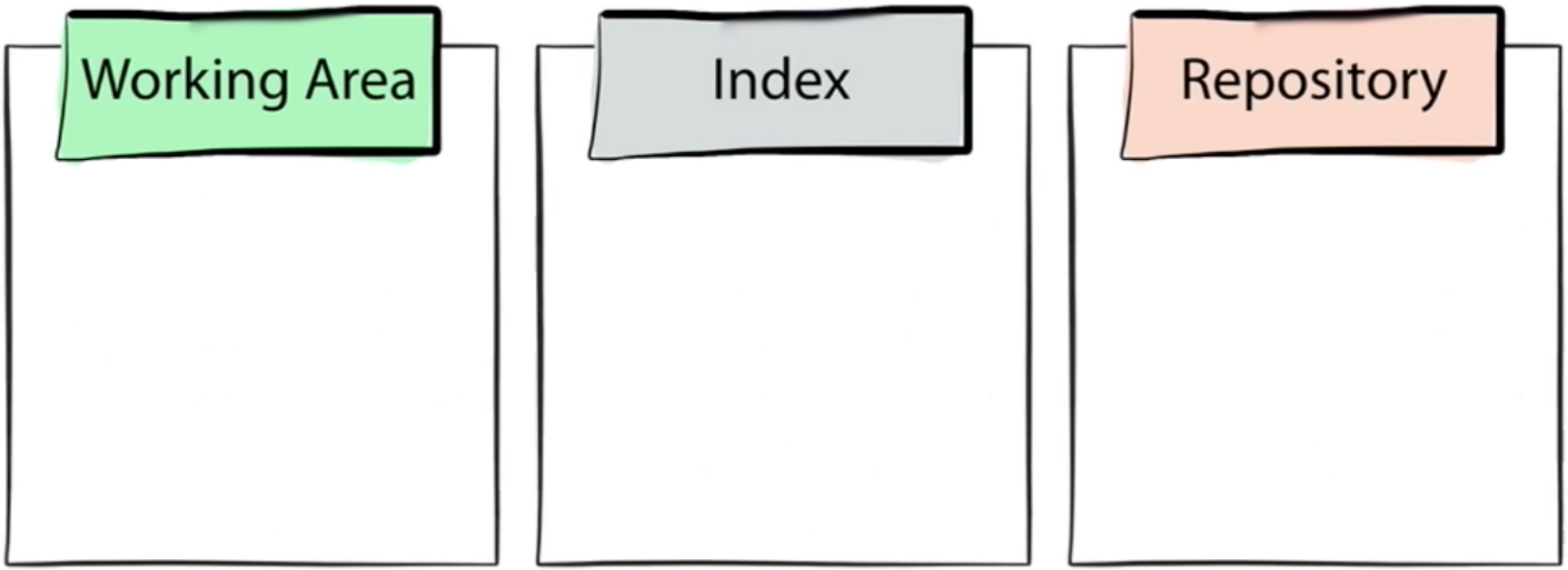
The Three Trees

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



Mastering Git

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



Git status

```
git status
```

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

The Staging Area

- The **staging area**, aka ***index***, is where you prepare a set of changes as your next commit
  - Staged changes are **not part** of the repository history…
  - Staging area is **not shared** across repos (or your team)
- Why stage?
  - Forces you to become a commit craftsperson
  - Allows you to be selective about your changes

Stage with add

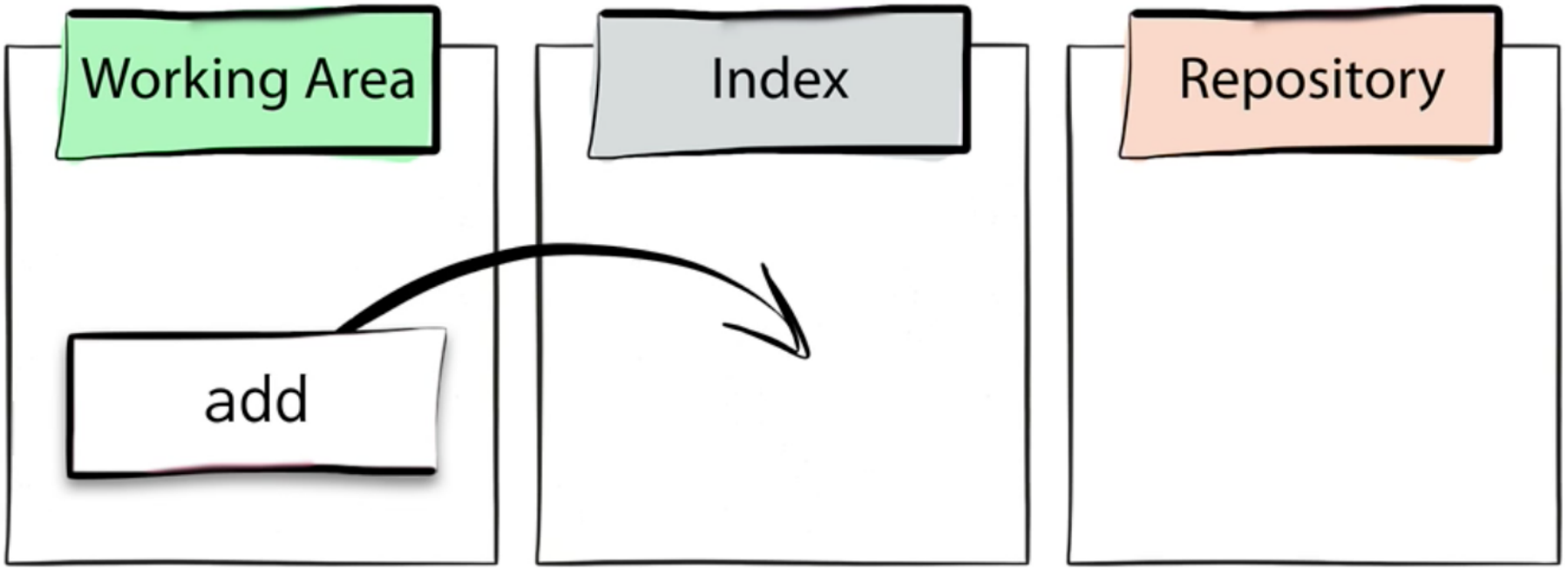
```
git add <pattern, file or files>
```

- 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 .
```

Staging area



Skipping staging

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

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

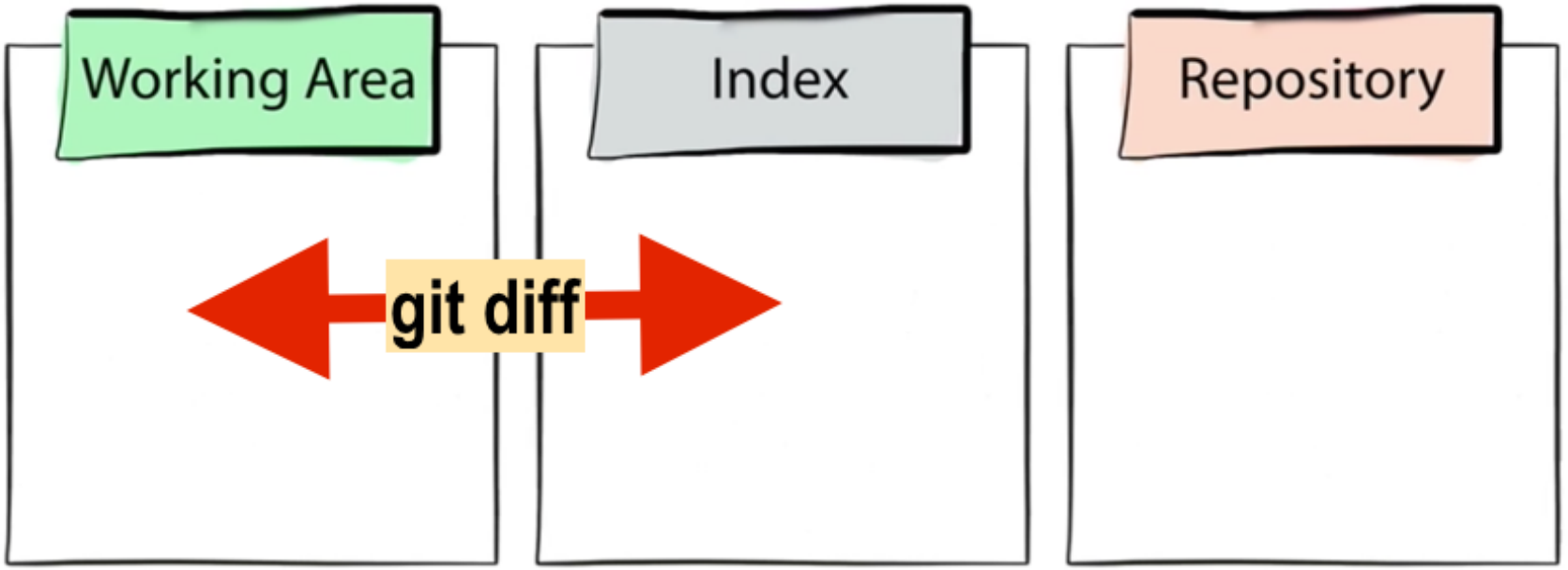
# or..
# this will auto-stage and commit
# commit changes to the filename
git commit <filename>
```

- *I don't recommend using this...*

Seeing what has changed

```
git diff
```

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

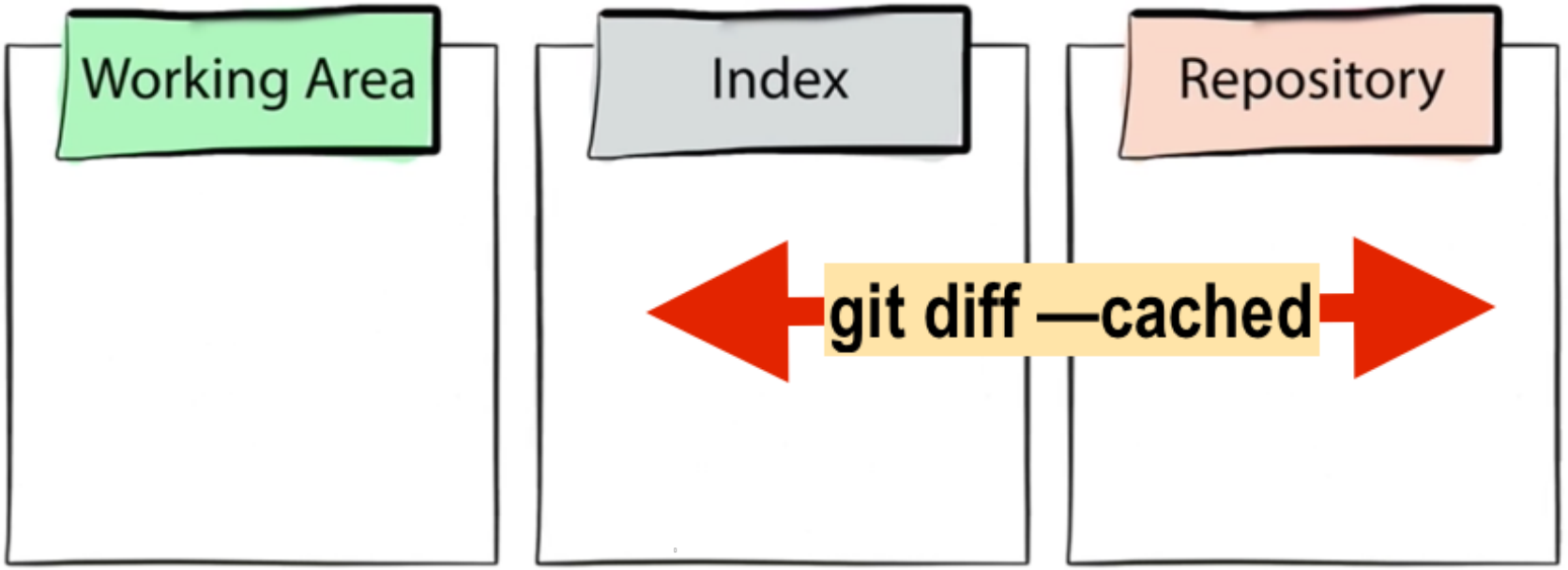


Seeing what has changed pt.2

```
git diff --staged
```

- Diff of what you have **staged** compared to the repository

```
# older git versions
git diff --cached
```



Diff in general

- Viewing a diff between WD and the REPO?

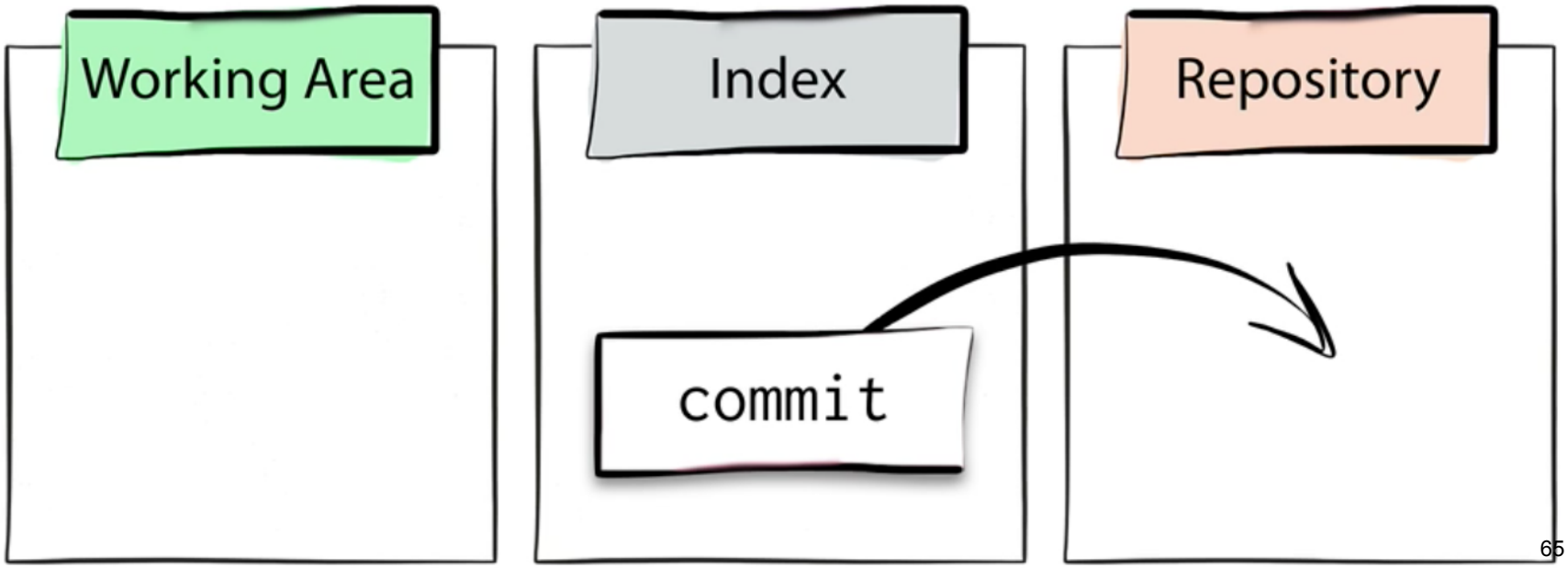
```
git diff HEAD
```

"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"
```

Viewing commits

```
git show
```

- Shows info about a commit

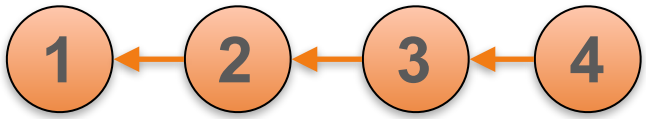
```
# latest commit
git show

# inspect a specific commit
git show 42d484c401f0a19cc8a954c16240821329acefac

# or use the abbreviated commit id
git show 4234
```

Project History

- Each commit you create is related to the previous commit (parent-child)



- In this way a history is generated

View your history

```
git log
```

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

Removing with git

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

- Deletes a file or directory
  - And auto-stages that removal

This is the only special bit

Moving with git

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

- Moves a file or directory
  - And auto-stages that move

This is the only special bit

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 a repository

After each step be sure to check status and the diff

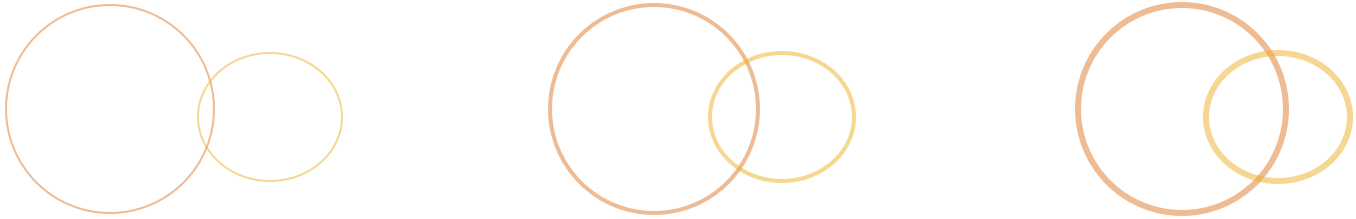
1. **Create** a new repository, "about-me"
2. Create a txt file named with your name
   - `Touch <yourname>.txt`
3. **Stage** the change
4. **Commit** the change
   - Check the log
   - Use "`git show`" on the commit id you just created
5. **Edit** the file to add a short profile about you:
6. **Stage**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

Undo with Checkout (WD)

- How do I throw away local edits?
  - Are they staged? *(No...)*
  - Are you OK with losing those changes? *(Yes...)*
  - Then...

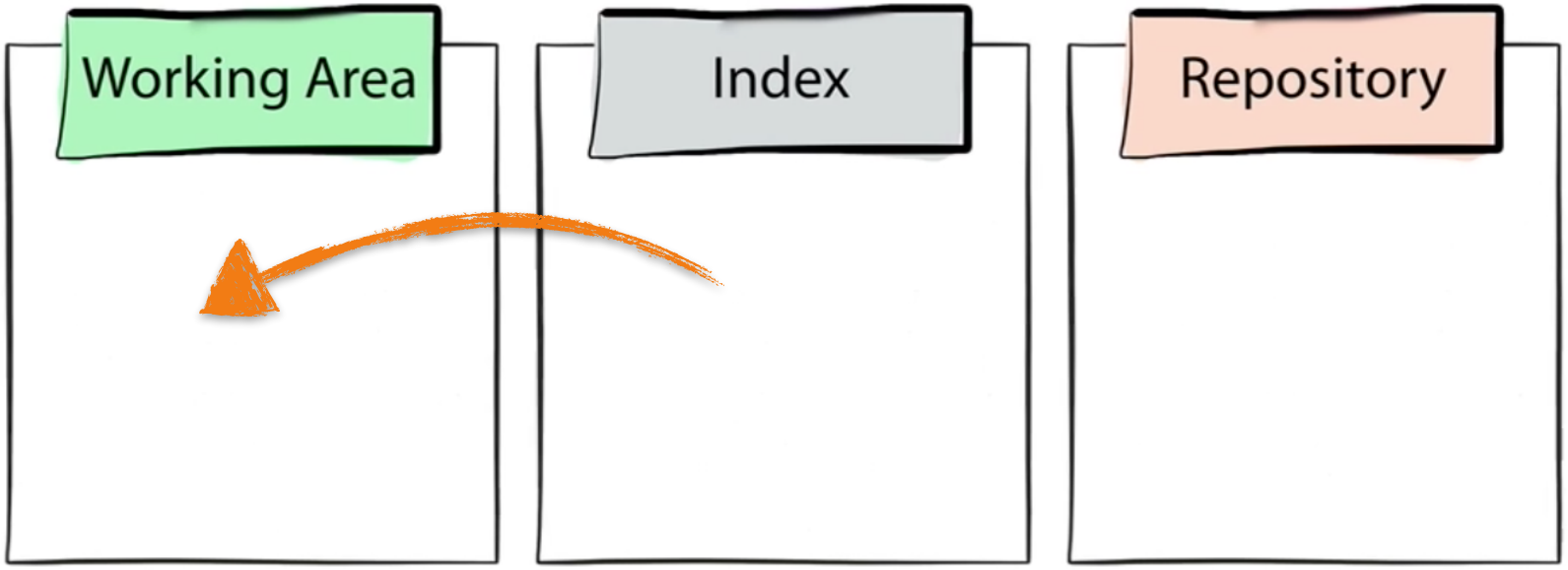
```
git checkout -- <file or pattern>
```

- Refreshes your files & folders with the current version, effectively throwing away any in progress edits...

Checkout w/ files

This is why a *staged* change can't be "undone" with checkout...

- *In this scenario,* **`checkout`***... updates WD files to be the 'version' staging (the index) is at*



- **[pro]** You can *also* checkout old versions of a file

```
git checkout <commit id> <file>
```

Undo with Reset (staging area)

- How do I un-stage a change?
  - Are they *actually* staged? *(Yes...)*
  - Then...

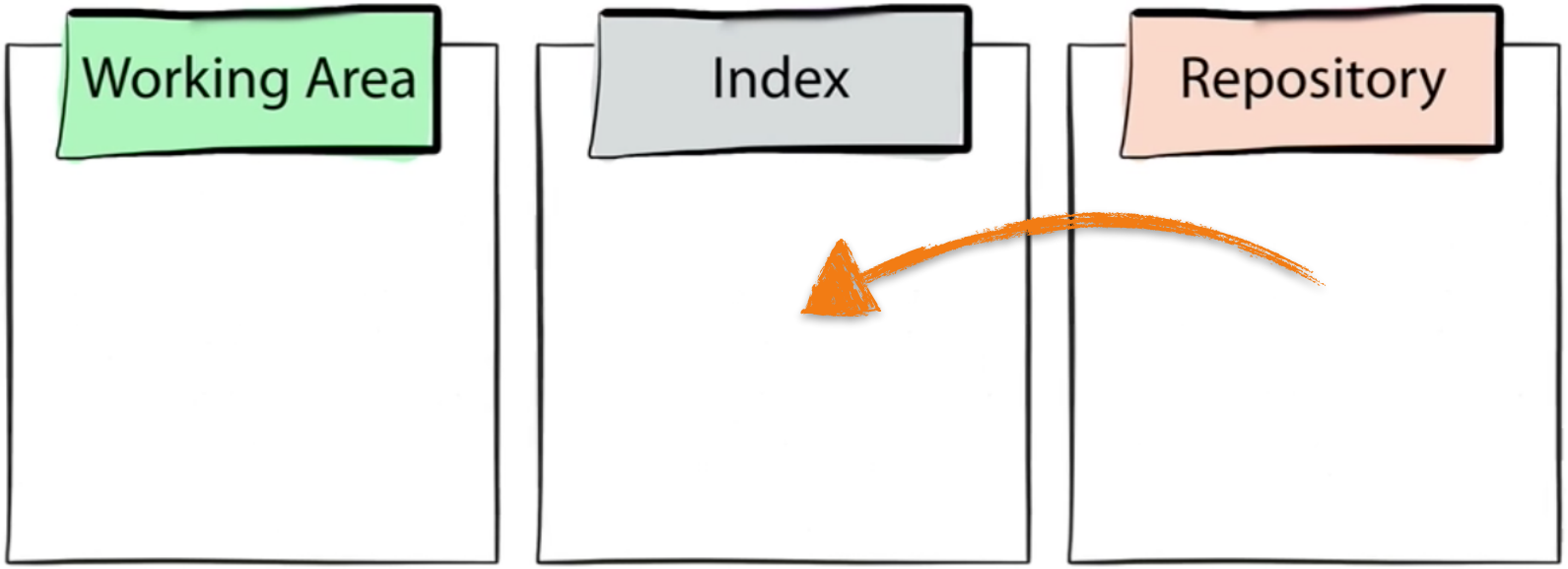
```
git reset <file or pattern>

# HEAD is implied
git reset HEAD <file or pattern>
```

- Refreshes staging with the current version, effectively throwing away any in staged edits...

Reset w/ files

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



- The staging area is more like a manifest

Amending commits

- How can I change my last commit message?

```
git commit --amend
# git will prompt you for a new message...
```

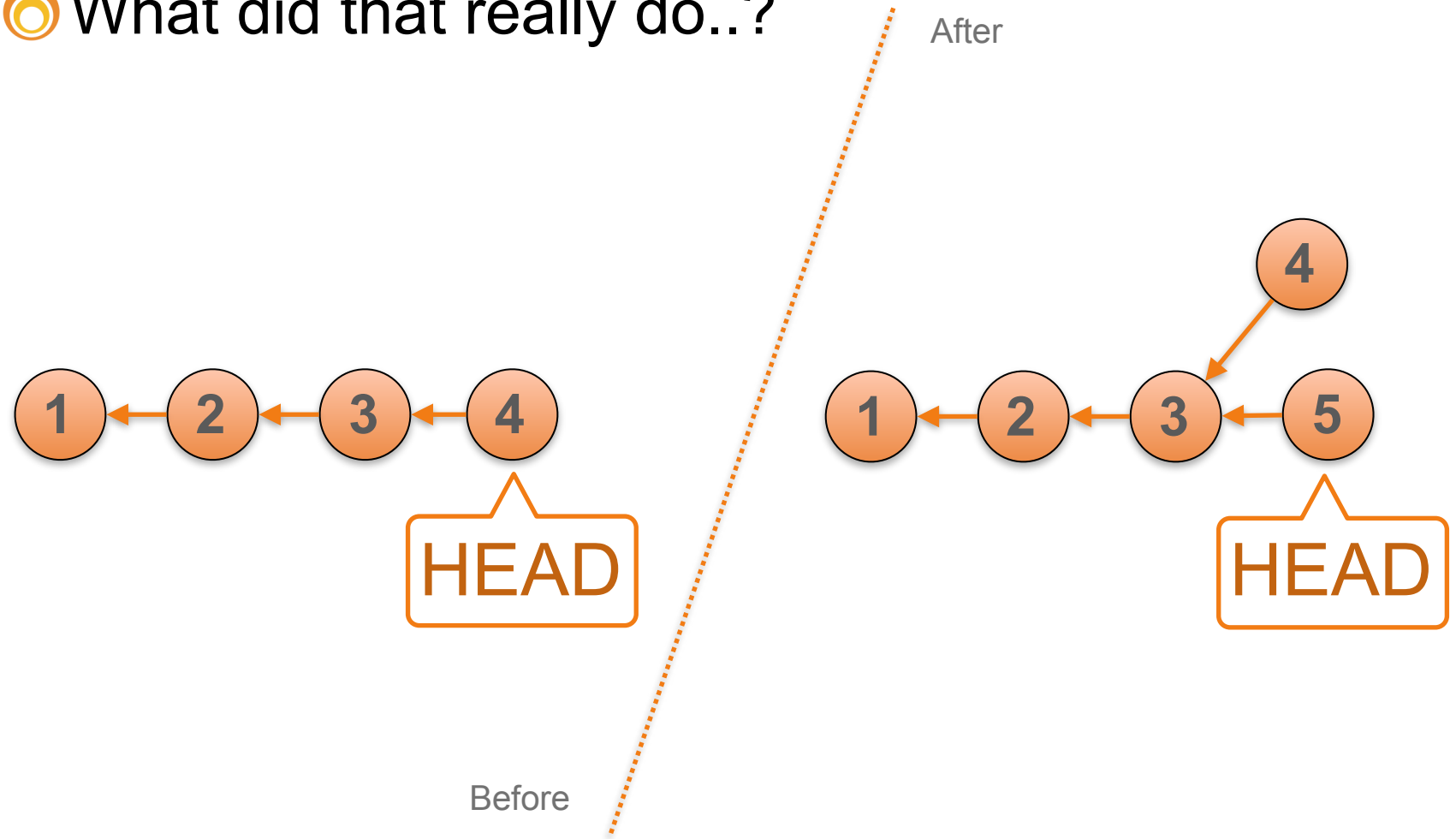
- How can I add some more changes to (or fix) my most recent commit?

```
# first make the changes to your files
# then stage the changes, then...
git commit --amend

# git will bundle all staged changes into
# the previous commit, making a new commit
```

Commit --amend

- What did that really do..?



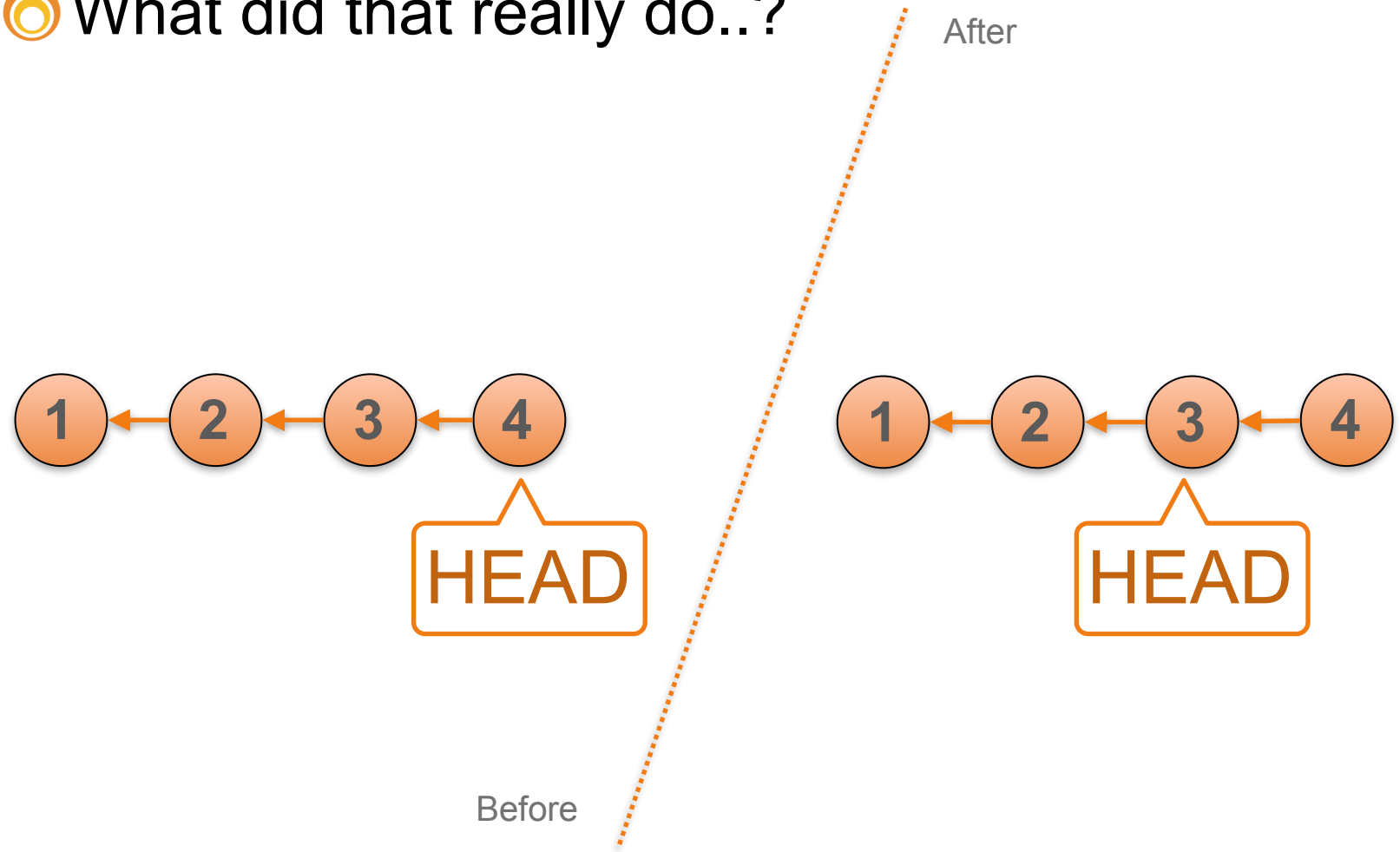
Reset w/ commits

- How do I undo my last commit?
  - Do you want to keep the changes or throw them away? (We can do both)
  - Did you share the commit with your team?
  - Just one back, or many...?

```
# moves history back by one commit
git reset HEAD^
```

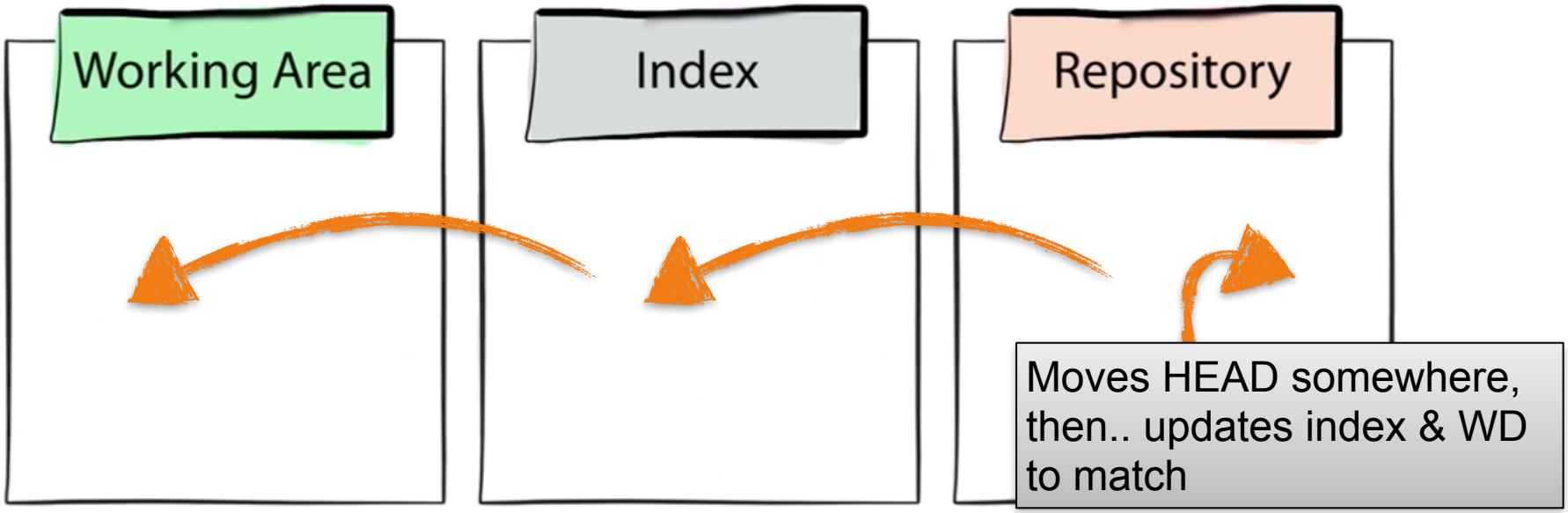
Reset HEAD^

- What did that really do..?



Reset and the three trees...

- **`reset`**... *when operating on commit references will update all three (depending on the type of reset)*



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
```

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^`

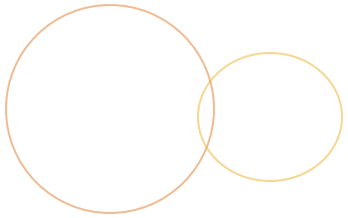
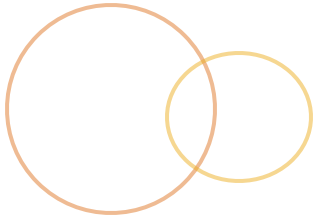
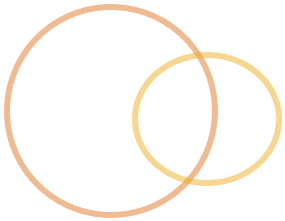
Lab: Undoing things

Un-staging and undoing changes with "git checkout" and un-staging them with "git reset"

1. Create two new files, "junk" and "LICENSE"
2. Edit "<your-name>.txt" file, ie: add your fav animal
3. **Stage** all your changes
4. Un-stage "junk" and "LICENSE"
5. **Commit** only the change to "<your-name>.txt"
6. Then **stage** the changes to LICENSE and **commit**
7. Then stage and commit "junk"
8. Undo that last commit
9. Now you can **delete** "junk".
10. Git **status** should show a clean repository
11. Bonus: **Amend** your last commit and edit the commit message

Toolkit

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

module

BRANCHING

What we'll learn

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

What is a branch?

- A separate track of history, allowing you to work on different tasks/tickets/ideas simultaneously w/ out overlap
- A branch is *like* a fresh copy of all your files

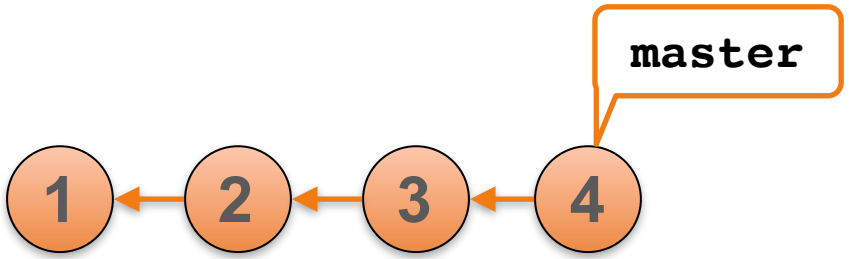
Why branch?

- Experimentation
- Stability
- Collaborate with others
- Diverging codebases or bucketing versions
- Supports deployment workflows
- *They are cheap!*

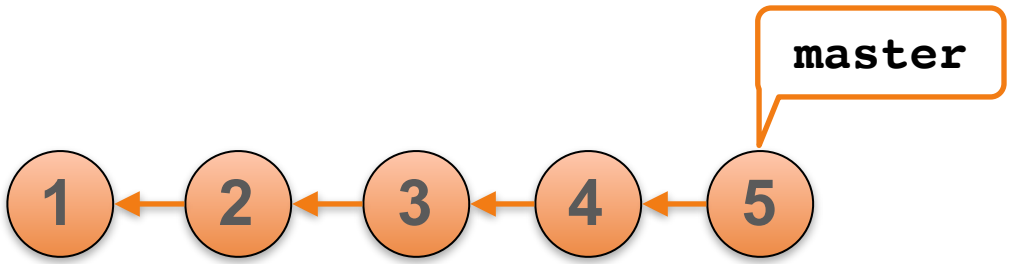
If you're starting something new, do it in a branch

How branching works

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



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



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

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>
```

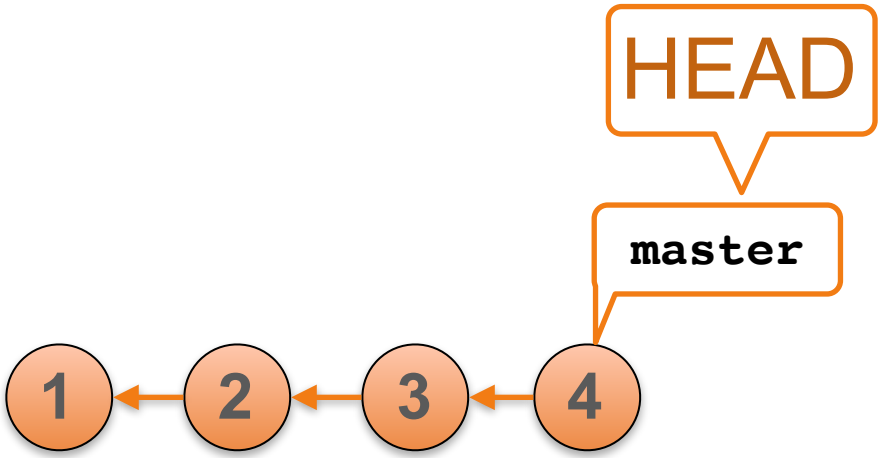
Log, revisited

- 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
```

HEAD

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



HEAD (Cont'd)

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

Checkout

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

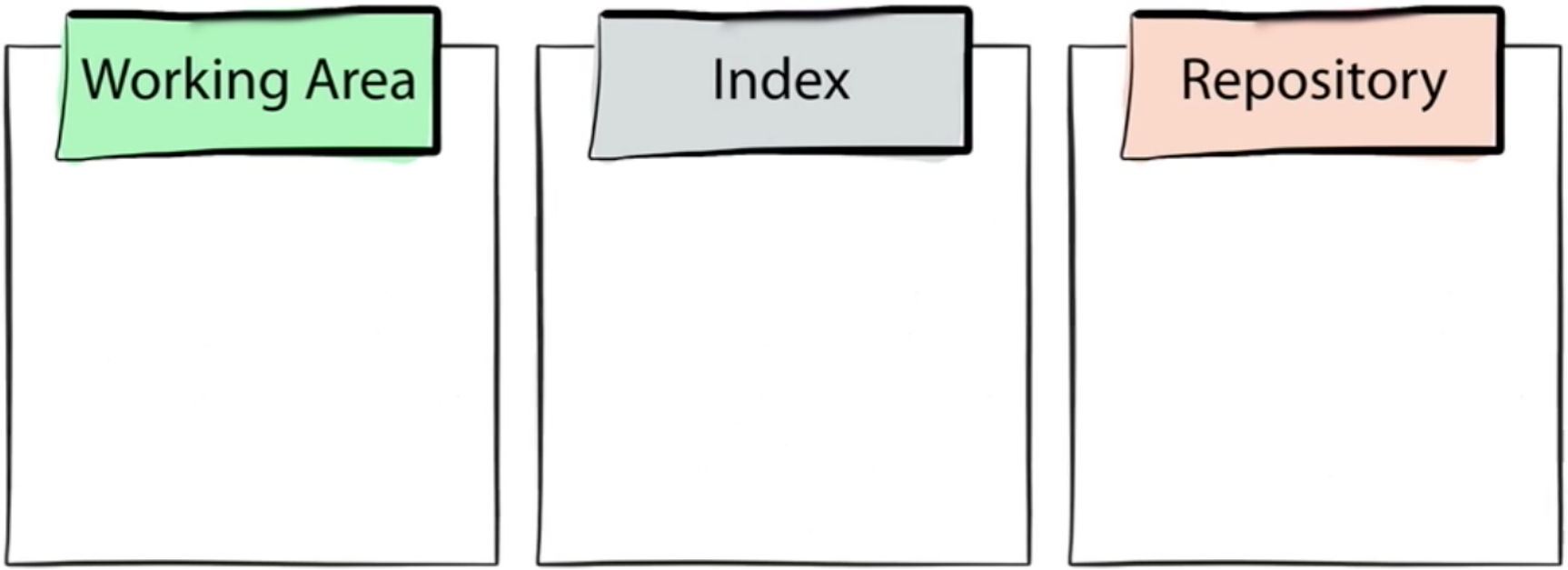
```
# undo changes to a file
git checkout -- <file>

# checking out a branch
git checkout test-branch

# checking out a specific commit
git checkout abfesef3
```

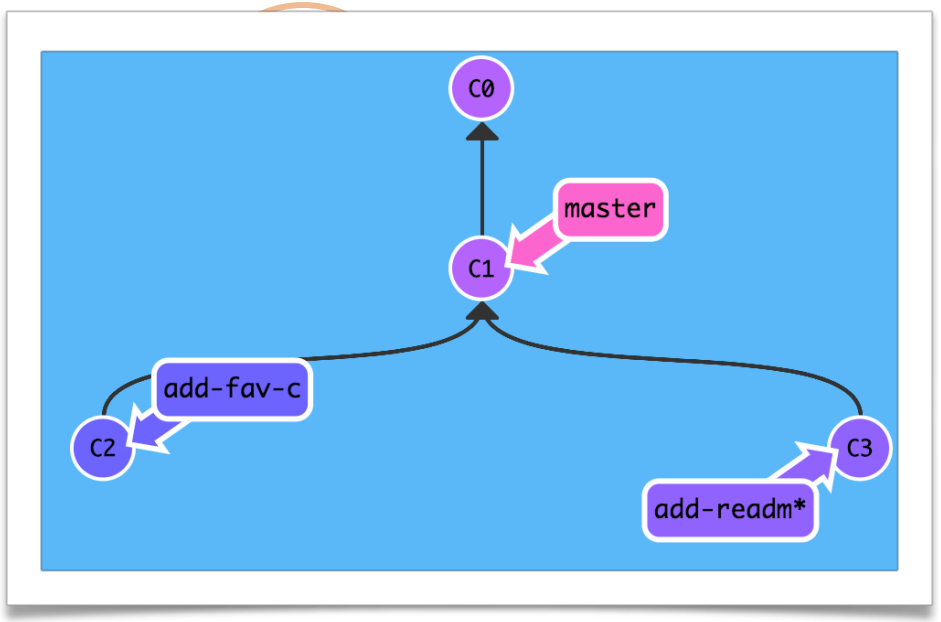
Branch & Checkout

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



Lab: Branching

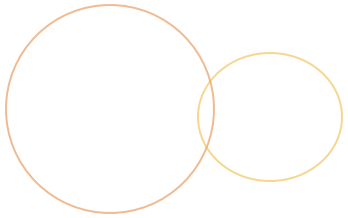
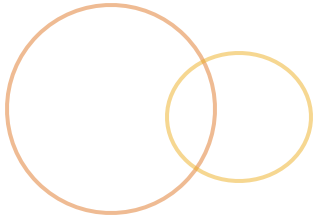
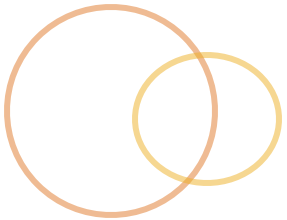
1. List your branches
2. Create a new branch off of master
   1. Label it "add-readme"
3. On the "add-readme" branch…
   1. Create a new file, "README"
   2. Stage and commit it
4. Back on "master"…
   1. Create a second branch, "add-fav-color"
5. On "add-fav-color"…
   1. Edit your `<name>.txt` file 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
```

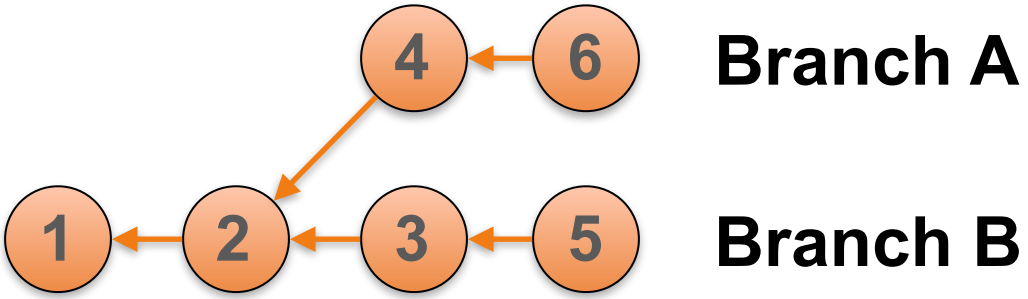
module

COMPARING BRANCHES

What we'll learn

- Comparing branches using commit ranges
- Determining commit differences across branches
- Determining file changes across branches

Comparing Commits (with Log)

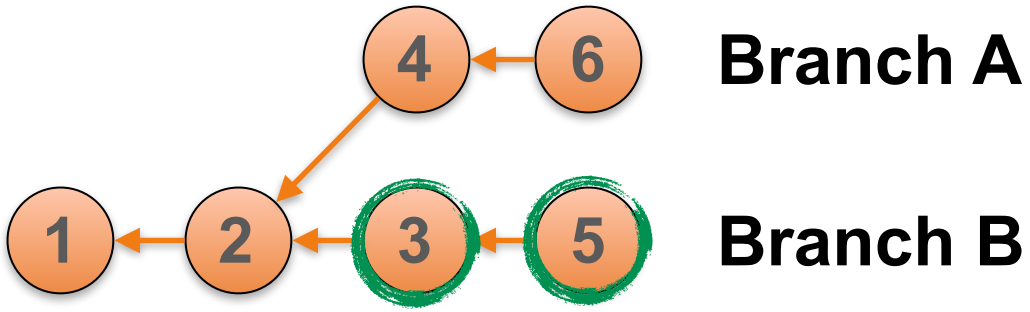


git log A..B

- What commits **are in B** that **are not in A**
  - *Commits reachable by B but not A*

```
git log A..B
git log ..B
```

HEAD is implied

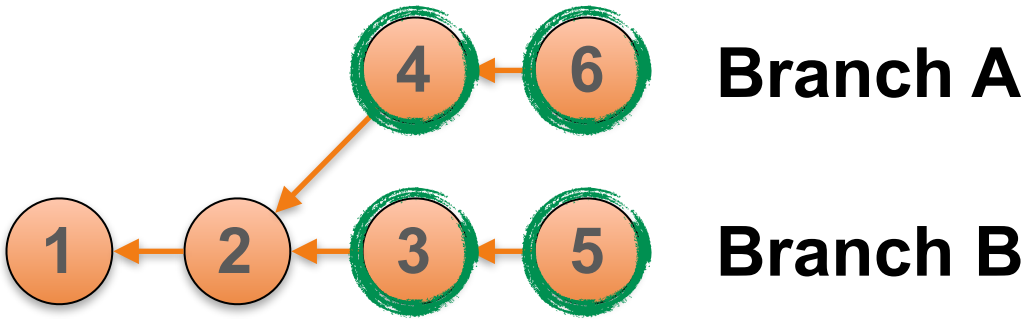


git log A...B

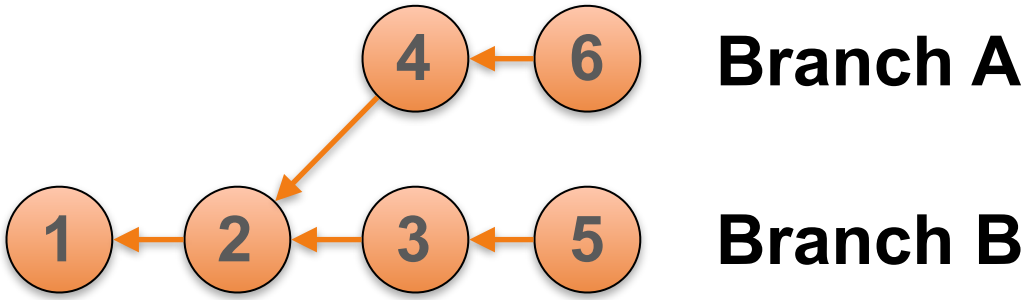
- What are the unique commits across both
  - *"Commits reachable by either, but not both"*

```
git log A...B

# display branch source
git log A...B --left-right
```



Comparing Changes (with Diff)

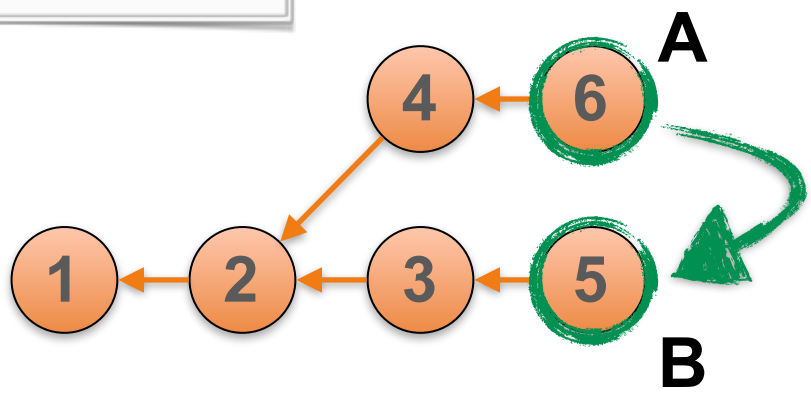


git diff A..B

- See the diffs between two branches
  - Compares between tip of A and B.
  - *"All differences between A and B"*

```
git diff A..B

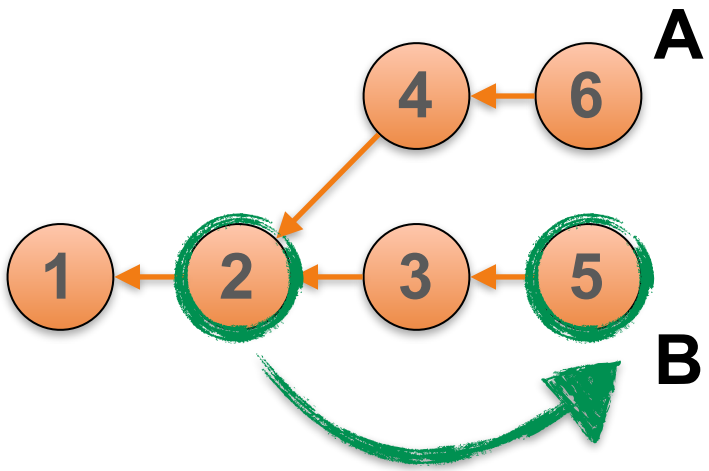
# same thing...
git diff A B
```



git diff A...B

- See diff that B would introduce to A
  - *"Difference between B and the common ancestor that B has with A"*

```
git diff A...B
```

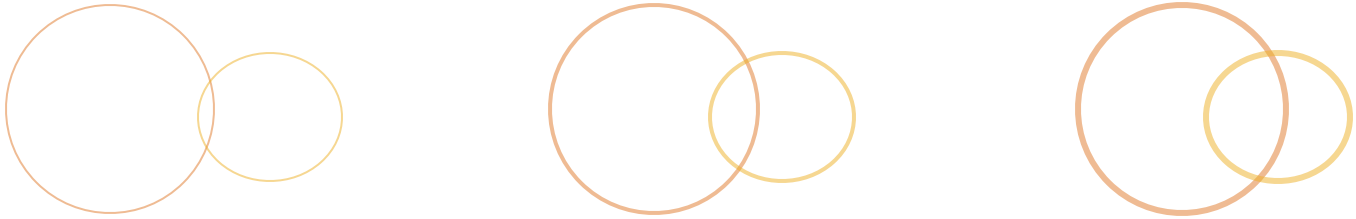


Lab: Comparing

- While on master, view the full log as a --graph
- Use "git log" and commit ranges to list...
  1. the different commits between your two branches, "add-fav-color" and add-readme"
  2. the commits "add-readme" has that "add-fav-color" does not.
  3. the commits that would be merged into master if you merged "add-fav-color".
- Use "git diff" and commit ranges to view...
  4. the difference between add-fav-color and add-readme
  5. the changes that would be merged into master if you merged "add-fav-color"

Solutions: Comparing

- Use "git log" and commit ranges to list...
  1. the different commits between your two branches, "add-fav-color" and add-readme"
     `git log add-fav-color...add-readme`
  2. the commits "add-readme" has that "add-fav-color" does not.
     `git log add-fav-color..add-readme`
  3. the commits that would be merged into master if you merged "add-fav-color".
     `git log master..add-fav-color`
- Use "git diff" and commit ranges to view...
  4. the difference between add-fav-color and add-readme
     `git diff add-fav-color..add-readme`
  5. the changes that would be merged into master if you merged "add-fav-color"
     `git diff master...add-fav-color`



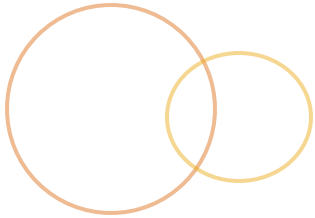
module

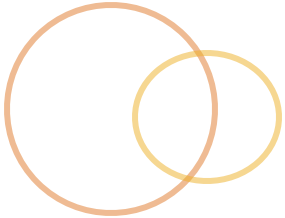
MERGING

What we'll learn

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

Merging





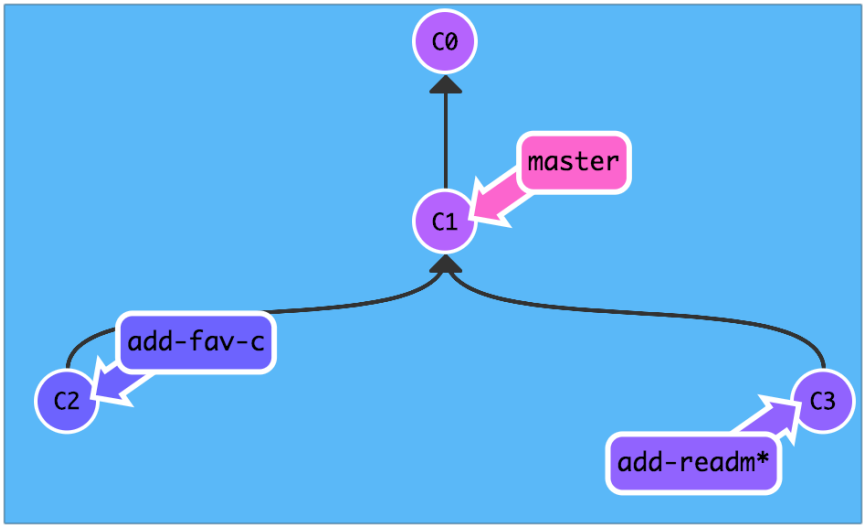
- A merge **combines the history** of two branches
  - It's how you get changes you've made in one branch into another branch (usually `master`)

```
# 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

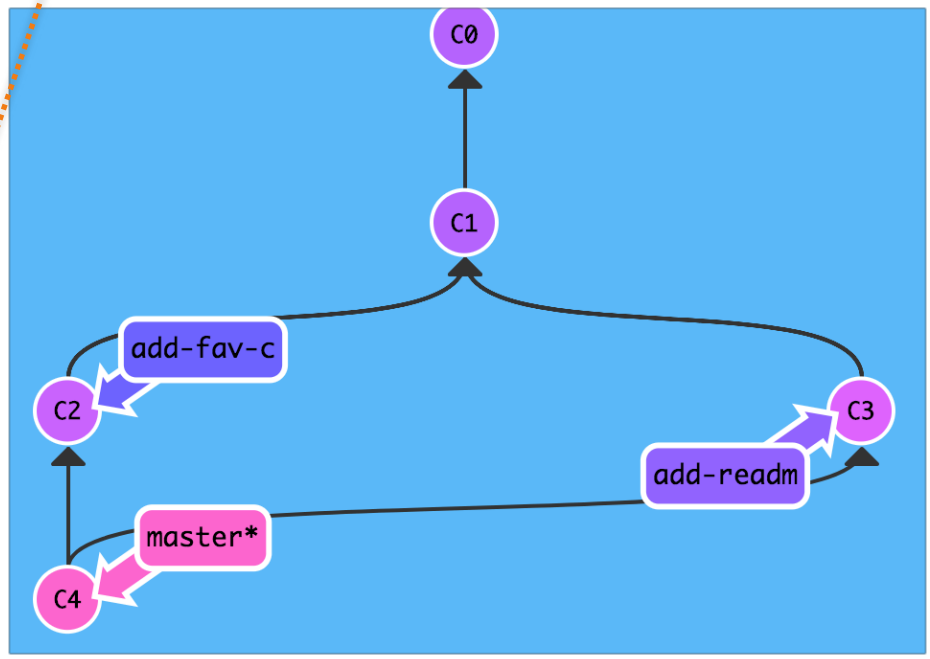
http://pcottle.github.io/learnGitBranching/?NODEMO

Before




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

After



Merge Strategies

- **Fast-forward** merge
  - When the target branch can be easily "moved forward" to the tip of the source branch

```
# we can tell git NOT to use this strategy
git merge <branch> --no-ff
```

- **Recursive** / **3-way** merge
  - When the two branches have diverged
  - Git creates a new commit to handle the combining of the two, plus any conflicts that come up

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

Branch Management

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

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

# delete a branch
git branch -d <name>

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

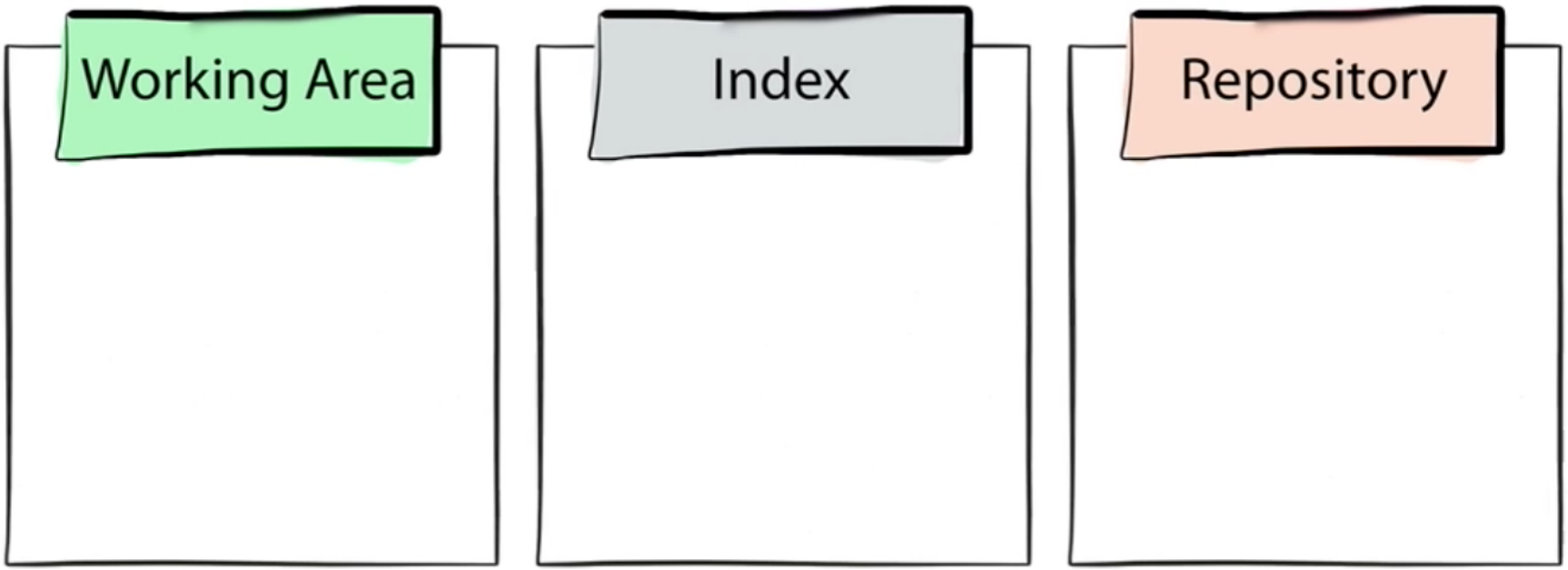
Actually:
Which branches are "in" the current branch's history

Our workflow / best practices

- **`master`** is stable, production-ready
- New work is *always* done in a new branch
  - Never commit directly to master
  - Branch off `master`
    - Do your work in that new branch
  - Merge back into `master` when done
- Remove branches when you're done with them
  - Usually once they are merged to `master`
- Write nice commit messages...
- Consider a branch naming convention

Merging

- So, how does **merge** affect the three trees?

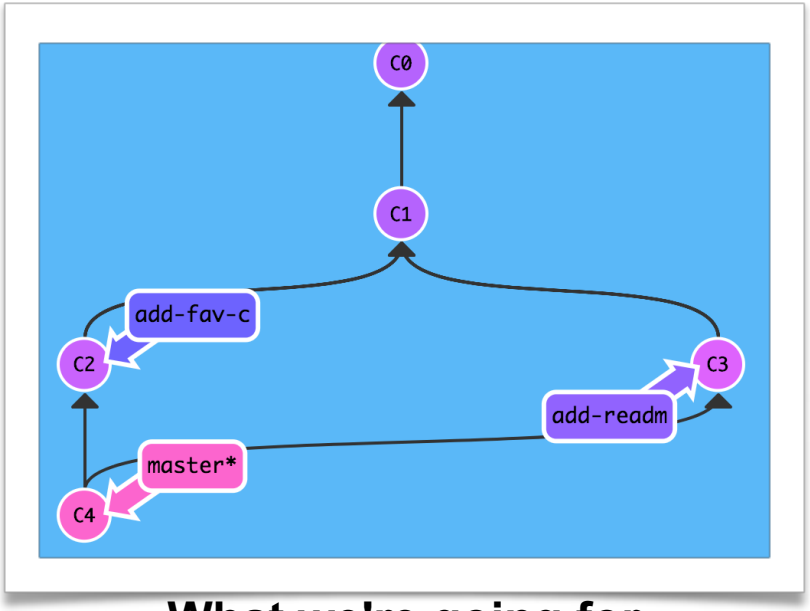


Recap: Merging

- `git branch` – view, create, manage branches
- `git checkout` – switch between branches and commits
- `git merge` – integrate work from one branch into another
- And we briefly covered a branching workflow, in which new work is done in topic branches off of the stable master branch

Lab: Merging

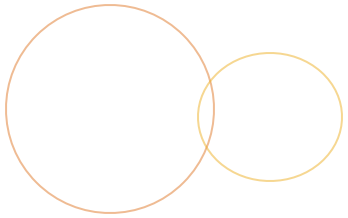
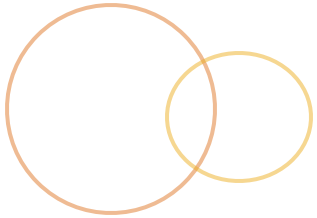
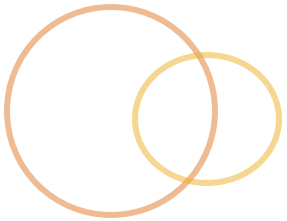
- On master
  - Merge your "add-readme" branch
    - Review your log . . .
    - What kind of merge did it perform?
  - List which branches are merged and not merged
  - Merge your "add-fav-color" branch
    - What kind of merge did it perform?
- Then create a new branch off master, "add-copyright"
- On "add-copyright"
  - Edit your `<name>.txt` file to include a copyright at the bottom
- Back on "master"
  - Merge "add-copyright"
    - Bonus: Merge w/out fast forwarding
- 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
git merge --no-ff
```

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

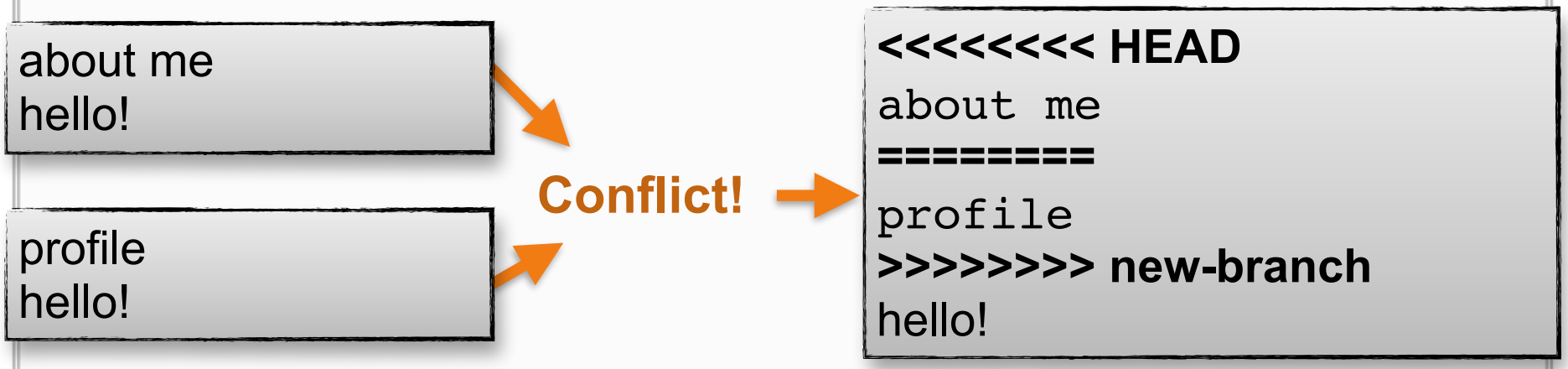
- Cancel the merge with --abort

```
# 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
```




```
# 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


# this won't work
# it goes back only 1 commit...
git reset HEAD^
```

Merge tool

- Conflicts can be tedious, a GUI can help

```
# opens your default GUI
git mergetool

# 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>
```

Some great tools
- sublime merge
- kaleidoscope

Recap: Merge issues

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

Lab: Resolve a conflict

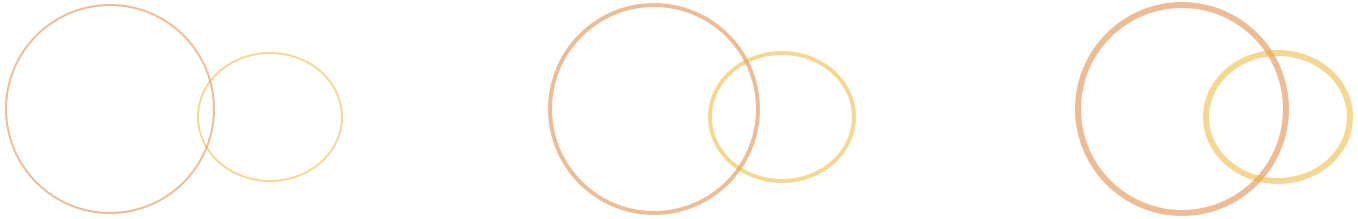
- Let's create a conflict!
  - Create two branches off of master
  - First branch will be called "red"
    - In this branch, edit your favorite color to be "Red!"
  - Go back to master
  - Create and checkout a second branch called "blue"
    - Edit favorite color to be "No, Blue!"
- Back on "master"
  - 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?
No problem. Create a conflict by editing the same line of the same file in two different branches.

Gotcha
Be sure to branch both off master!

Toolkit

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



module

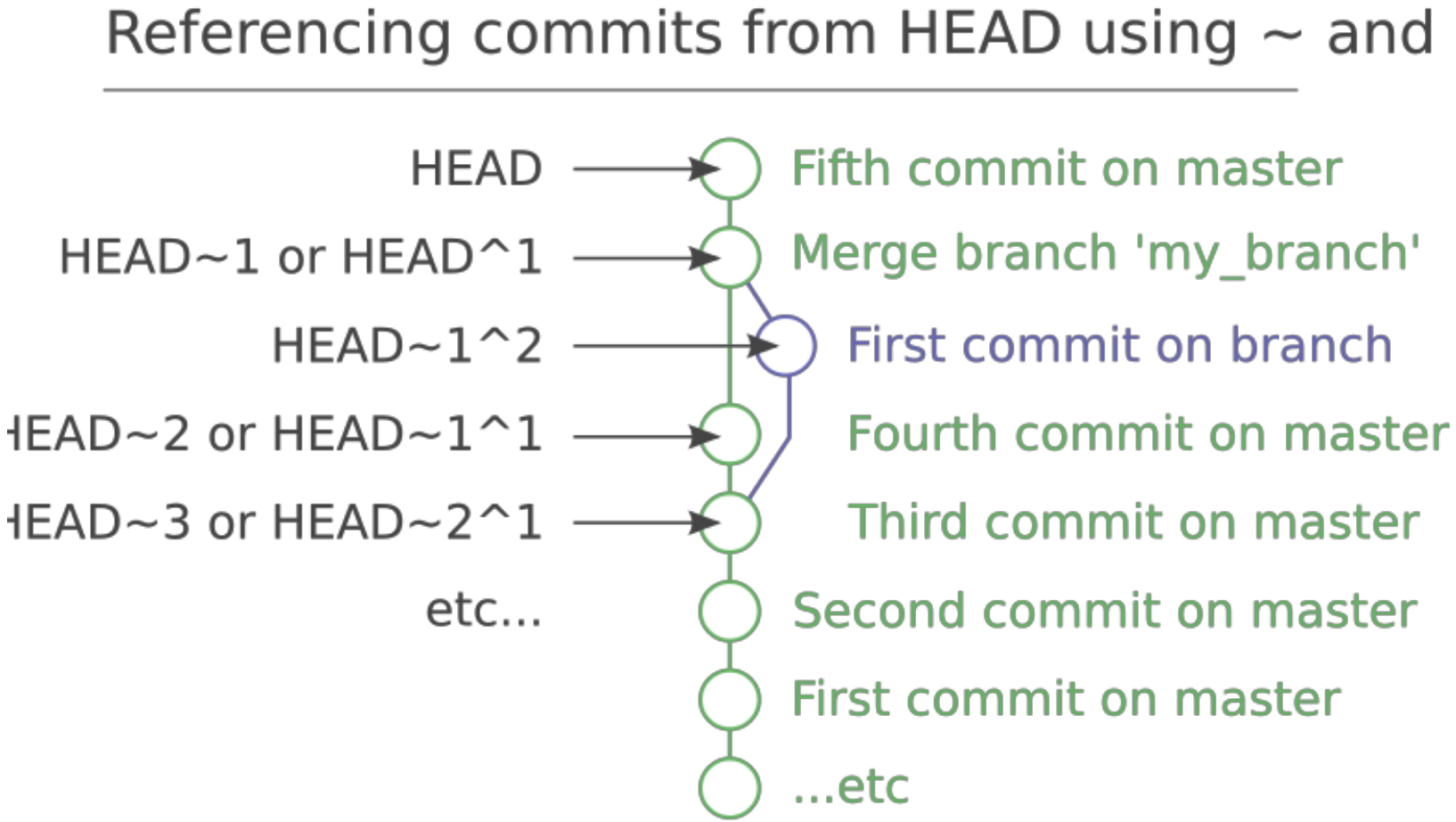
CORE ODDS & ENDS

TAGS, ALIASES, STASHING AND IGNORE

What we'll learn

- Parent References
- Tags
- Aliases
- Stashing
- Ignoring

Git parent references



http://www.paulboxley.com/blog/2011/06/git-caret-and-tilde

Tags

- A **tag** is a static bookmark for a specific commit
- What should I tag?
  - Important releases
  - Each x.x.x version
  - Important commits you may want to come back to

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

# 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 --'

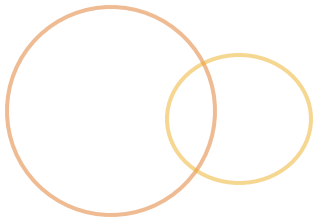
# undo-merge
git config --global alias.undo-merge 'reset --hard
ORIG_HEAD'

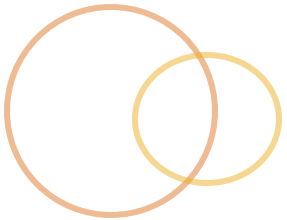
# graph
git config --global alias.graph 'log --oneline --graph --
decorate'
```

Stashing

- Quickly stores work in progress without committing
  - Saves any work in progress that is not committed
  - Clears the staging area & working directory of changes
- Why stash?
  - If you have edits you are working on and want to switch to a different task w/out committing
  - Quickly saves work you want to revisit
  - If you are unable to switch branches because the target branch would conflict w/ your current edits
- Often a commit is fine, too

Stashing





```
# 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>

# Clear your stash
git stash clear
```

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/*.log

# negate a pattern
!main.log
```

- Github has a lot of prefab gitignores

https://git-scm.com/docs/gitignore#_pattern_format

Empty folders

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

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

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

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...

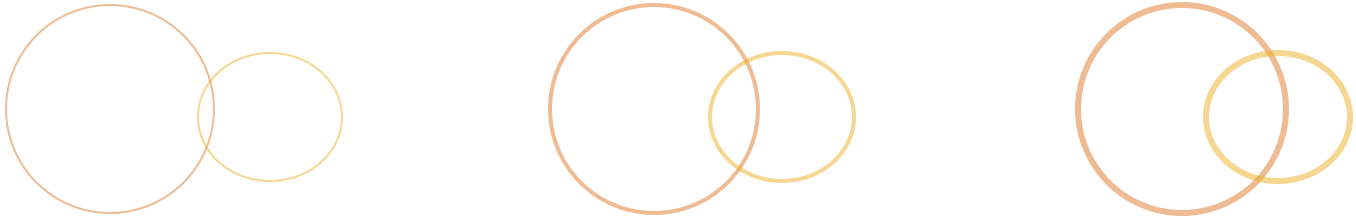
- Back on master...
- List your stash
- Un-stash the changes you were working on
- Make any further edits you want to your profile
- Create a new branch, "fav-city"
  - On "fav-city", stage and commit your profile edits
- Merge both "fav-city" and "fix-copyright" into master...
- View your log and delete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

Toolkit

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

Let's regroup!

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



optional...

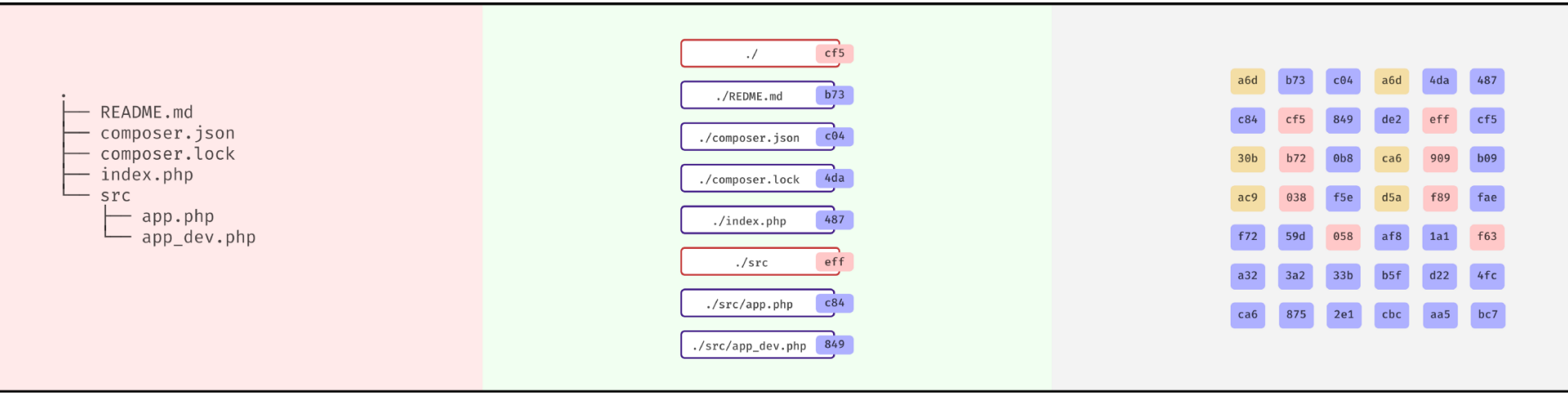
DEEP DIVE ON GIT

The trees (more detail)

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

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

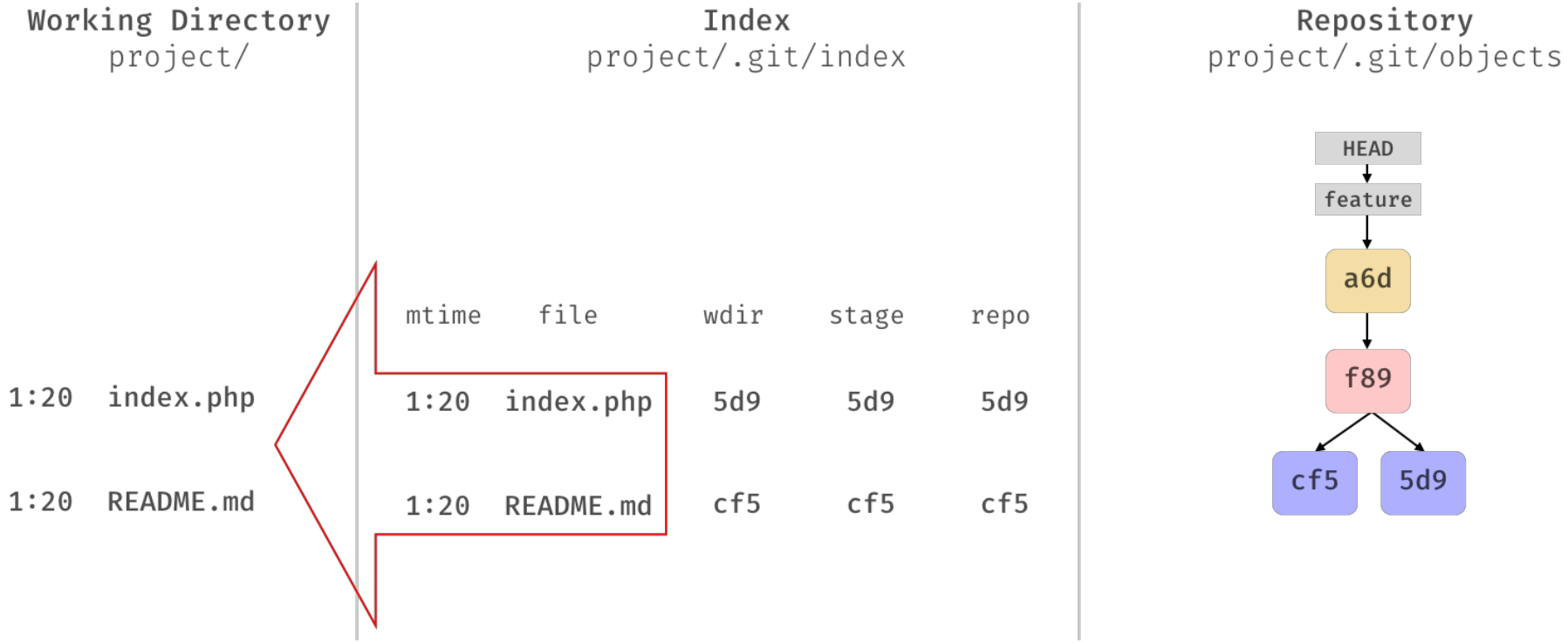
The repository
a mess of objects:
- commits
- trees
- blobs
- tags



Source: https://hackernoon.com/understanding-git-index-4821a0765cf

The index

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



Source: https://hackernoon.com/understanding-git-index-4821a0765cf

Modifications

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

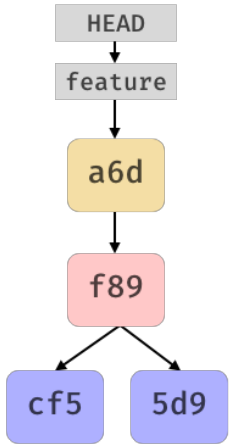
Working Directory
project/

1:21	index.php
1:20	README.md

Index
project/.git/index

mtime	file	wdir	stage	repo
1:21	index.php	9f5	5d9	5d9
1:20	README.md	cf5	cf5	cf5

Repository
project/.git/objects



Source: https://hackernoon.com/understanding-git-index-4821a0765cf

Add

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

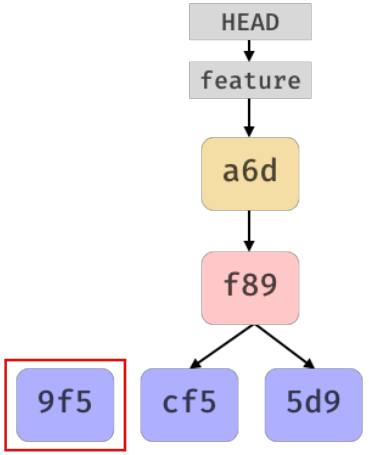
Working Directory
project/

1:21	index.php
1:20	README.md

Index
project/.git/index

mtime	file	wdir	stage	repo
1:21	index.php	9f5	9f5	5d9
1:20	README.md	cf5	cf5	cf5

Repository
project/.git/objects



Commit

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

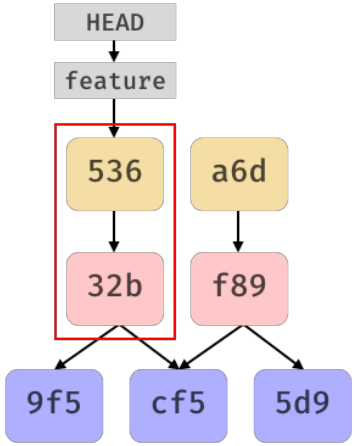
Working Directory
project/

mtime	file
1:21	index.php
1:20	README.md

Index
project/.git/index

mtime	file	wdir	stage	repo
1:21	index.php	9f5	9f5	9f5
1:20	README.md	cf5	cf5	cf5

Repository
project/.git/objects



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

DAY 2 REVIEW AND TEST!

Test: What does each command do?

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

- git init
- touch 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

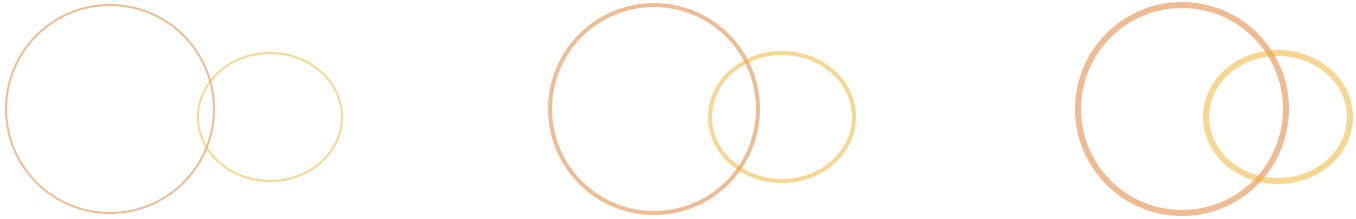
Test: What does each command do?

- 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

HOSTING A REPOSITORY

What we'll learn

- Repository hosting options
- What are remotes and how to use them
- Uploading our repository to a hosting service
- Pushing to share your changes
- Pulling to grab updates
- Intro to pull/merge requests

Hosting our repository

- Options
  - GitHub.com
  - GitLab.com
  - Bitbucket
- **They host a copy of our repository**
- And maybe...
  - Tools to manage our project
  - Manage issues
  - Offers workflows around merging
  - Access control

Let's check one out

Remotes

- **Remotes** are local references to other versions of the repository hosted elsewhere
  - They are just copies of the repository!
  - Although their history may diverge over time…
- Why do remotes exist?
  - **Contribute** to public projects
  - Have **others contribute** to our projects
  - Have a **team work together** on a project
  - Simply **share our code**
  - Keep our repository **safe** in case our HD dies

Using Remotes

- Hosting our repository and sharing our work

If I haven't already...
I'll run through getting my own
repository hosted online - then you'll get to try

Sharing our repository (recap)

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

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

- We then pushed our local repository data to the remote repository

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

- 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

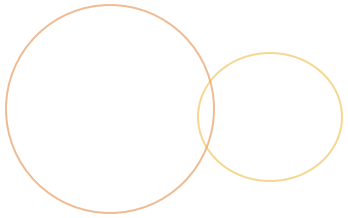
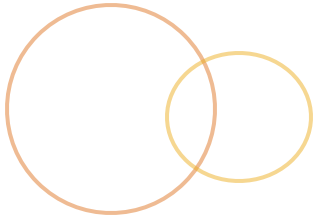
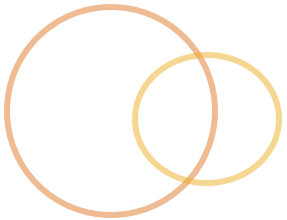
- On our hosting service (github.com)
  - Make sure you have an account!
- Create a new repository
  - Call it "**about-me**"
  - Don't *initialize* it, it is already initialized...
- Follow the directions they give or…
  - Copy the remote repository **url**
  - Then add a **remote** in our local git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

Issues with pushing? Try:

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

```
git remote add origin <remote-url>
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

- How to share your work (via branches)
- Keeping up to date
- Fetch vs Pull
- Sharing tags you've created
- Setting up tracking branches
- General remote management

Sharing work/changes

- Use the **`push`** command to send updates to our remote(s)
  - *Typically* one branch at a time
  - Local <branch> to the matching remote <branch>
  - Sends all repository data, too

```
git push origin <branch>

# for example
git push origin bug1
# explicit source to target
git push origin bug1:bug1
```

Sharing work [Advanced]

- We can send multiple branches

```
git push origin bug1 feature5 other-branch
```

- Or all branches

```
git push origin --all
```

- *Neither of these are part of a typical workflow*

Staying up to date

- We'll use the **`pull`** command to download updates from our remotes
  - One branch at a time
  - **Merges** from the remote <branch> to the ~~matching remote <branch>~~ **the <branch> you are on**
  - Fetches all repository data, too

```
git pull origin <branch>

# for example
git checkout bug1
git pull origin bug1
```

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

Update without merging

- **Fetch** updates your remote references
  - Updates all the repository data
  - Updates all remote branch refs
  - *But does NOT automatically merge anything*

```
git fetch origin

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

Remote update vs Fetch

```
# fetch updates all branch data
git fetch
git fetch --all

# or one at a time
git fetch origin master

# before that, we used remote update
# identical to "fetch --all"
git remote update
```

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
```

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
```

- These behave almost like normal branches, except they are not for working on (just for keeping track)

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

# you can delete them, too
git push origin :master
git push origin --delete master
```

Puts us in a detached HEAD

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
```

Pruning branches

- There are potentially three copies of a branch
  - Your local branch (ie: master)
  - The remote ref (ie: origin/master)
  - The remote's actual branch (ie: master, on the remote)
- You must tell git to clean up remote references to branches that no longer exist on the remote

```
# three variants -->
git fetch --prune

# you can also config it
git config fetch.prune true
```

 # or...
git prune
git remote prune origin

Remote management

```
# Adding remotes
git remote add <name>

# Removing remotes
git remote rm <name>

# Renaming
git remote rename <orig-name> <new-name>

# Listing
git remote
git remote --verbose
```

Our process thus far

- Master is considered stable
- We keep master up to date
- Start new work off master (in a new branch)
- Push to share your work
- Pull to keep up it up to date
- Once your work is merged in master (on the remote) you'll pull to update master again

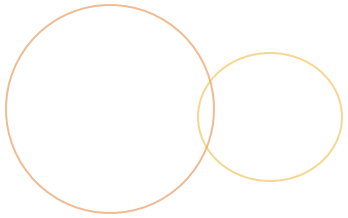
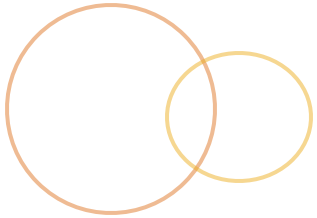
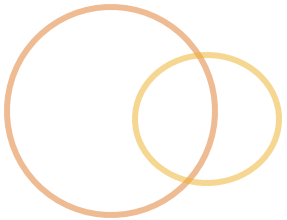
Lab: Pushing & Pulling (Team of 1)

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

Remotes and branches

LOCAL

master

add "origin" remote then fetch

Remotes and branches

LOCAL

master

origin/master

Origin (server)

master

Remotes and branches

LOCAL

master

origin/master

Origin (server)

master

create branch topic-1

Remotes and branches

LOCAL

master

topic-1

origin/master

Origin (server)

master

Remotes and branches

LOCAL

master

topic-1

origin/master

Origin (server)

master

git push origin topic-1

Remotes and branches

LOCAL

master

topic-1

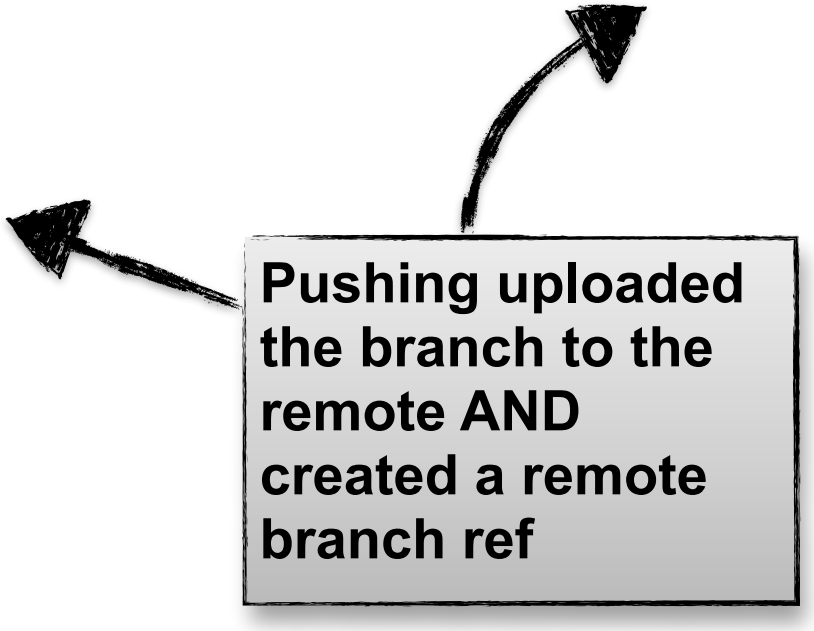
origin/master

origin/topic-1

Origin (server)

master

topic-1



Remotes and branches

LOCAL

master

topic-1

origin/master

origin/topic-1

Origin (server)

master

topic-1

Someone else pushes a branch to origin

Remotes and branches

LOCAL

master

topic-1

origin/master

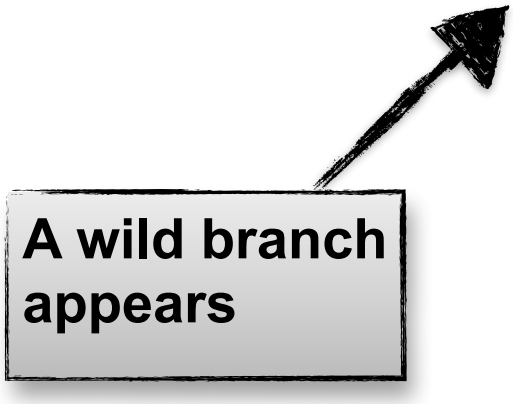
origin/topic-1

Origin (server)

master

topic-1

topic-20



Remotes and branches

LOCAL

master

topic-1

origin/master

origin/topic-1

Origin (server)

master

topic-1

topic-20

Then I fetch origin (again)

Remotes and branches

LOCAL

master

topic-1

origin/master

origin/topic-1

origin/topic-20

Origin (server)

master

topic-1

topic-20

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)

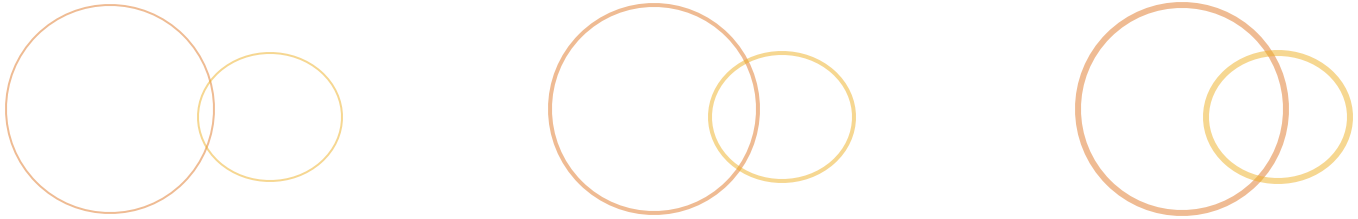
master

topic-1

topic-20

Lab: Join the team!

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



module

COLLABORATION IN A SINGLE REPO

What we'll learn

- Intro to a team workflow with a single repository
- Participate - How to clone a repo
- Do work - What are topic branches
- Sharing work - with pull/erge requests
- Keeping up to date

A single repository for a team

- We're going to work as a team using a single repository
- This is the most simple remote structure
  - Great for small to mid-size teams

I'll create an about-us repository
and then submit a change via a branch.

No need to follow along, you'll get your chance!

Participate - 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 - Pull/Merge Requests

- Push your branch to the remote to share
- Create a pull/merge request *early* to encourage collaboration and review
- Your team should define:
  - review expectations; who does it?
  - merge expectations; who does it?
  - any tests/CI?
- Delete branches on the remote when done

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`**

Stop here
We'll review & merge together

Keep `master` up to date

- Typically new work will be branched off **`master`**, so we should keep master up to date

```
# make sure you're on master
git checkout master
git pull

# or, being explicit
git pull origin master
```

- Starting new work at this point is simply

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

Keep branches up to date

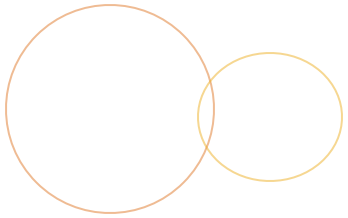
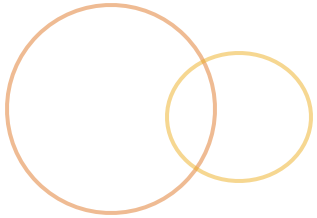
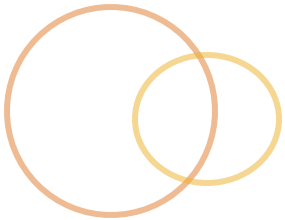
- Occasionally you'll want to update your topic branches
  - Grab the latest changes/fixes
  - Incorporate a teammates changes

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

# incorporate changes
# from master
git pull origin master

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

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

If I haven't already
I'll add an index page that we'll all work on simultaneously
We'll be adding our names to the index

Lab: Stay up to date

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

- Make sure **`master`** is up to date

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

- Do the work
  - Branch off **`master`**, *use a branch name that won't collide with your team*
- Share your work
  - Push your branch
  - Create a pull/merge request (DO NOT MERGE)

Dealing with conflicts

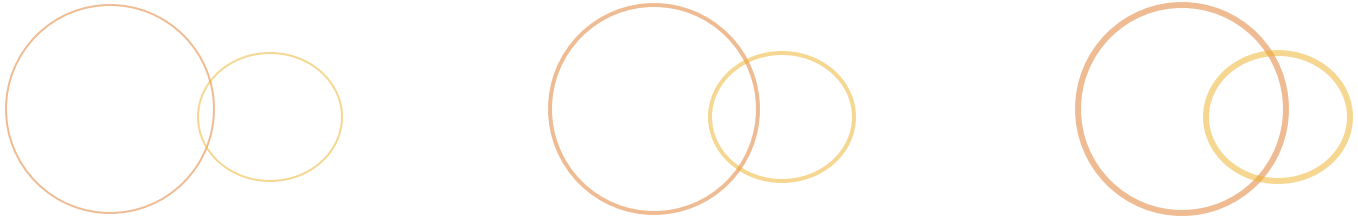
- When you get a conflict
  - ie: you can't merge your pull/merge request
- **Merge** the target branch into your topic branch
  - This will cause the conflict locally
- **Resolve** the conflict
  - Stage, commit (to complete the merge, per normal)
- **Push** your updated topic branch
- The pull/merge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

- Make sure `master` is up to date
- Cause the conflict
  - Merge `master` into your local topic branch (the branch with the conflict when trying to PR it into master on the remote)
- Resolve the conflict
  - Fix, stage, commit
- Share the update
  - Push your branch
  - View your original pull/merge request, the conflict should be resolved



module

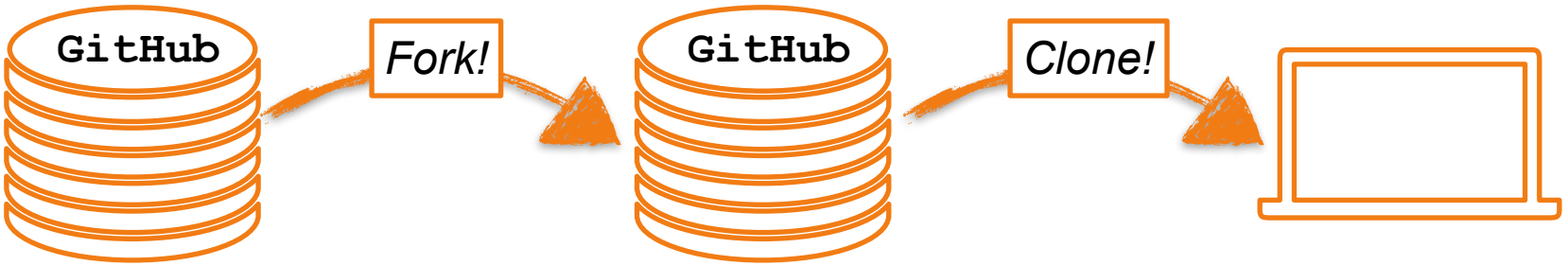
COLLABORATION ACROSS MULTIPLE REPOS

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

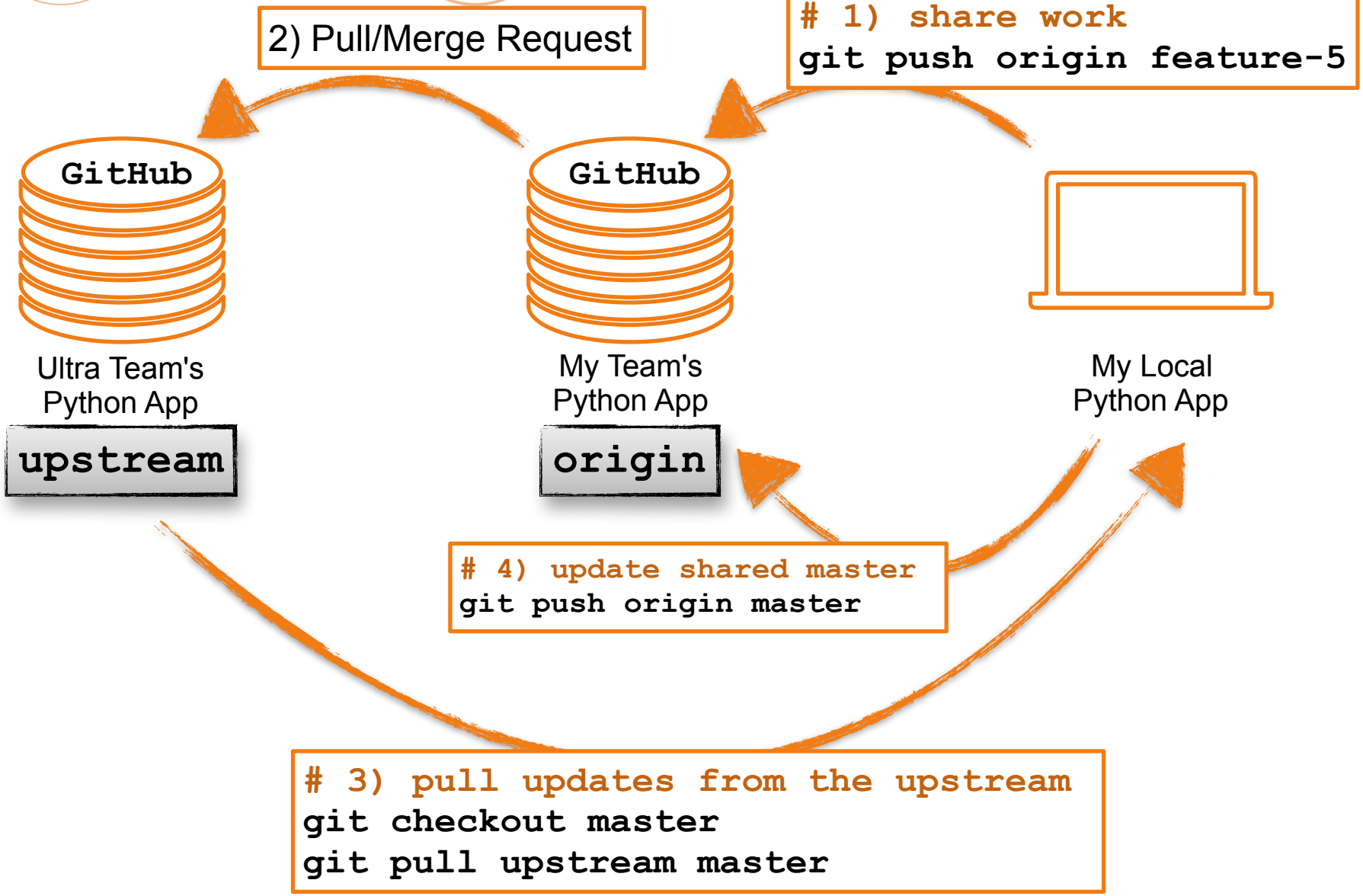
- A **fork** is a **copy** of a repository that remains within your hosting account
  - You can fork any public project in GitHub/Lab
  - It's a copy you own
- Why fork?
  - To submit changes to a public/open source project
  - Organizational safety net for larger teams
  - To split up a project or codebase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



Forking workflow

I'll run through it, just sip coffee and observe

github: https://github.com/rm-training/git-forked
gitlab: https://gitlab.com/rm-training/git-forked

Lab: Working with forks

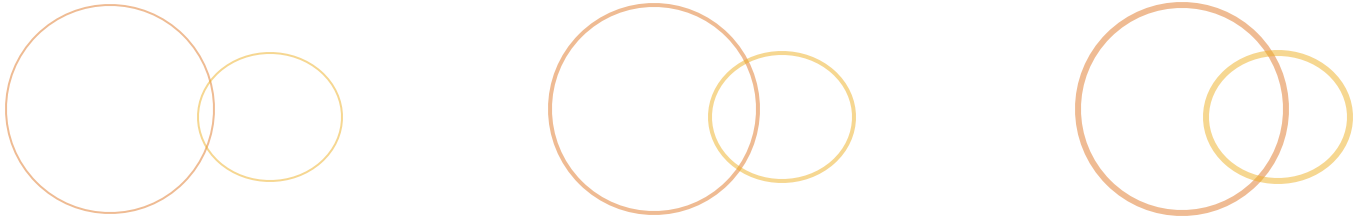
github: https://github.com/rm-training/git-forked
gitlab: https://gitlab.com/rm-training/git-forked

- **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

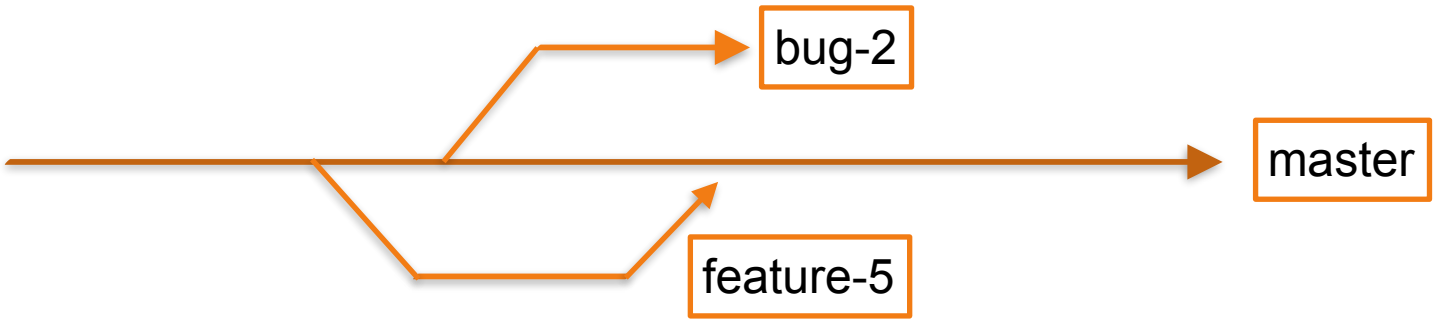
WORKFLOWS

What we'll learn

- What are our workflow options
- Branching strategies (the flows)
- Tagging strategies
- Remote strategies
- Team roles

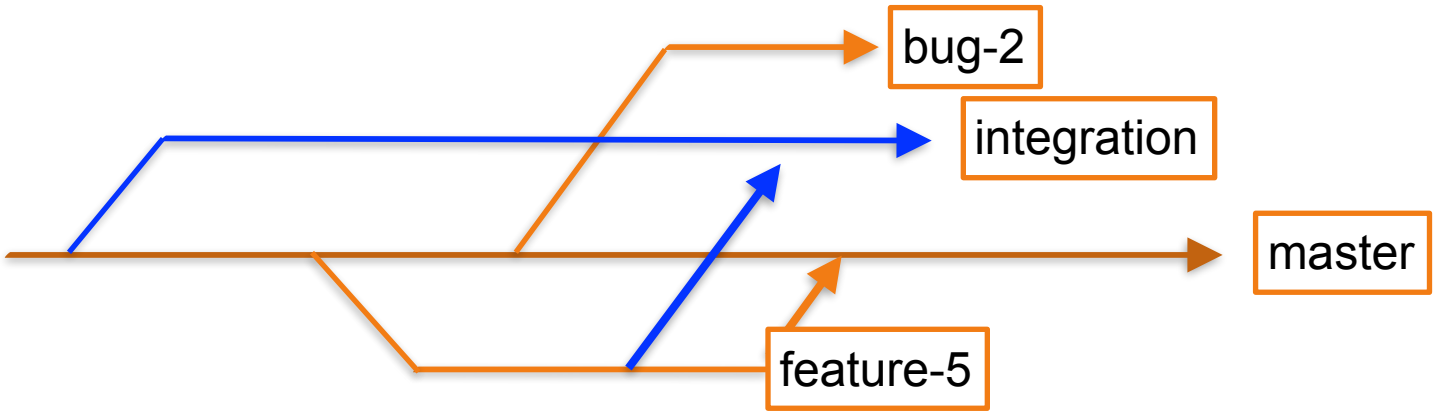
Branch Strategies

- Branch off of, merge to and deploy master as need



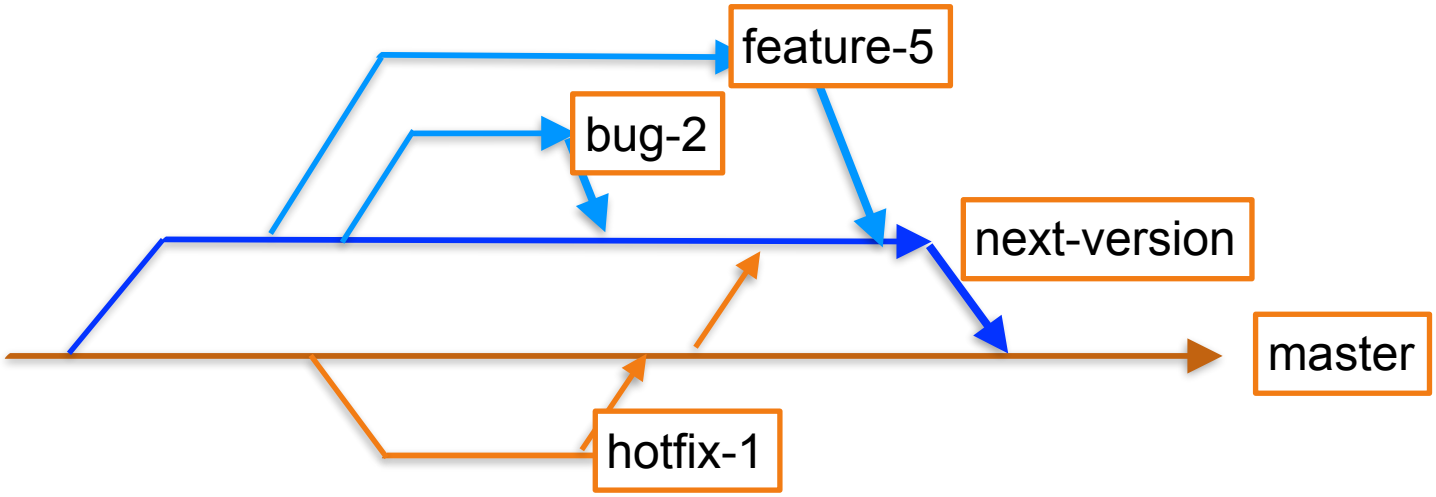
Branch Strategies

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



Branch Strategies

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



The Flows

- Git Flow
  - Well structured but complex
  - Develop, release, feature, bug and hotfix branches
- 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

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?

~~Remote~~ Maintainer Structure

- Single repository for a small team is best
- As teams grows, or code ownership splits across different *maintainers*, it may make sense to copy a repository into one or more remotes
- More remotes = More mental overhead
  - But more stability and control
- What is right for us?
  - How large is my team
  - How large is my codebase
  - Who are the maintainers?

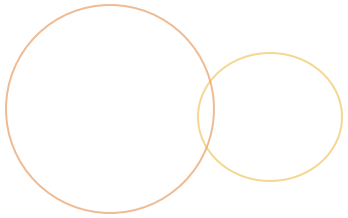
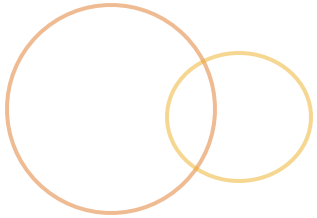
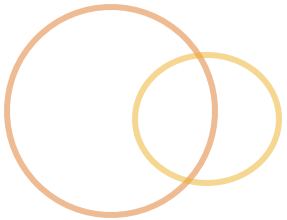
Tag Strategies

- Tag each release as a version
  - V1.0
  - V1.0.1 (hotfixes)
- Tags are static
- Versions in "active development" should be in a branch

Team Roles

- Maintainer
- Reviewer
- Contributor

- Who should review?
- Who should merge pull/merge requests?
- Who should deploy?

module

MANAGING HISTORY

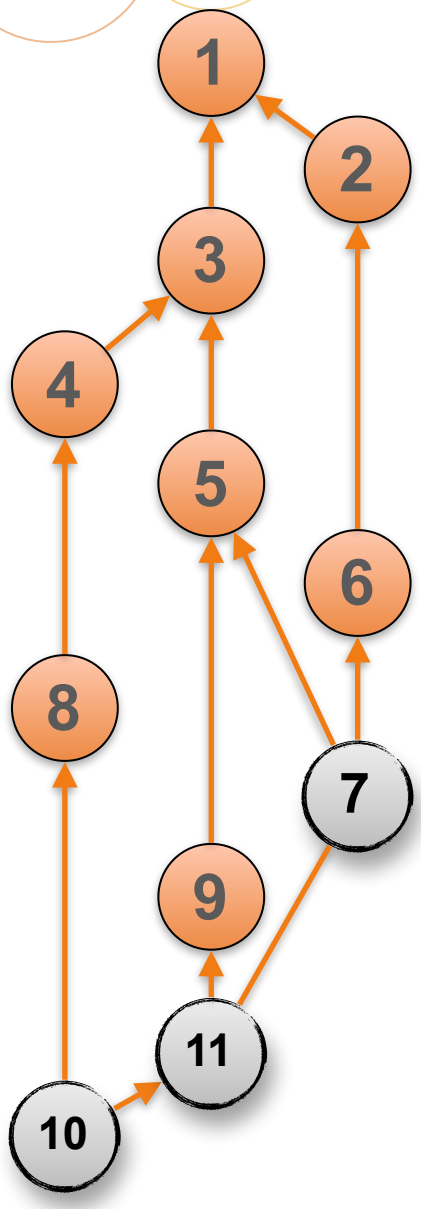
What you'll learn

- Why you may want to consider *managing your repository history*
- Different approaches to history management
- 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?

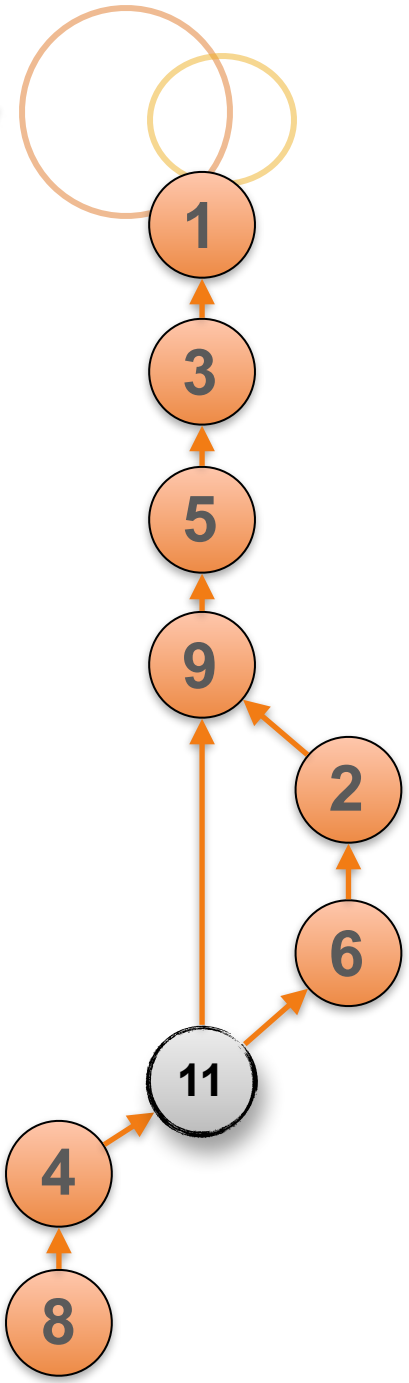
- Some teams want more control over their repository history
  - Reduce "merge commit" and "commit" noise
- Makes it easier:
  - to **debug** when looking at the log
  - to **see related work** happened
  - to **see only meaningful work** (not mistakes)
  - to **revert full branches** of work

Why maintain history



Kinda messy
and hard to follow

Easy to follow



Ways we control history

1. Control the merge type that happens
2. Avoid merges altogether by rebasing when keeping up to date
3. Squash many commits into one when merging into master/releasing
4. Copy a commit into my branch

1. Controlling the merge

- When merging, git will attempt to use the most appropriate method of merge
- Fast-forward
  - If both branches share the **same parent**
  - Is like a non-destructive **rebase**
- Recursive (3-way)
  - If the branches have diverged
  - Results in a new merge commit

Most of the time your hosting platform will handle this merge for you...

```
# typically, when merging into master
# we can prevent a fast-forward
git merge --no-ff <branch>
```

2. Rebase to stay up to date

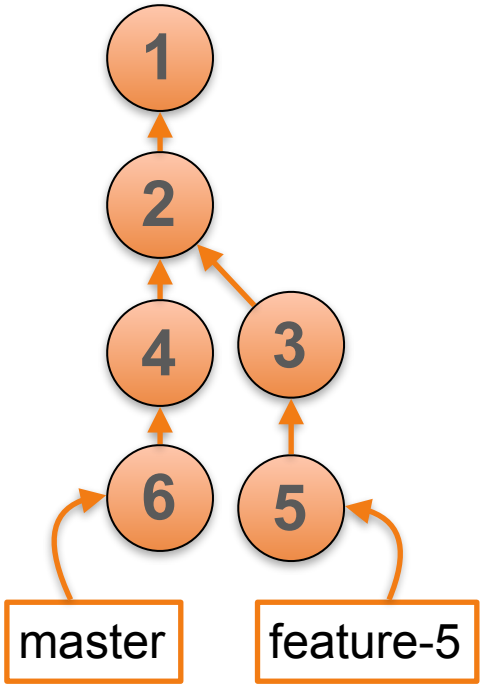
- **`rebase`** is like a branch transplant, or grafting
  - *Snips off* the commits in current branch
  - Re-plays the branch, one commit at a time, with the target branch tip as the new start of the branch
  - Updates the current branch ref

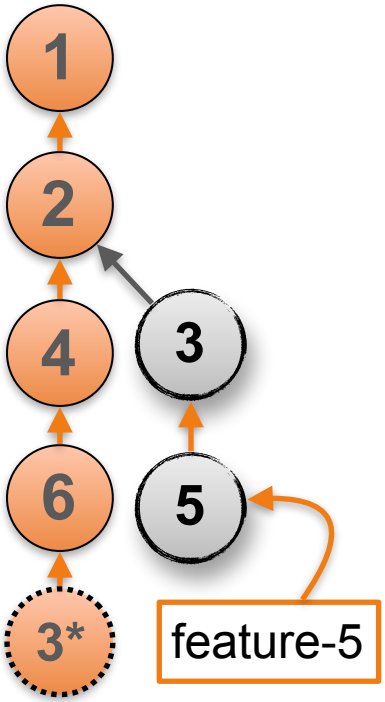
```
git checkout feature-5
git rebase master
```

"rebase `feature-5` onto `master`"

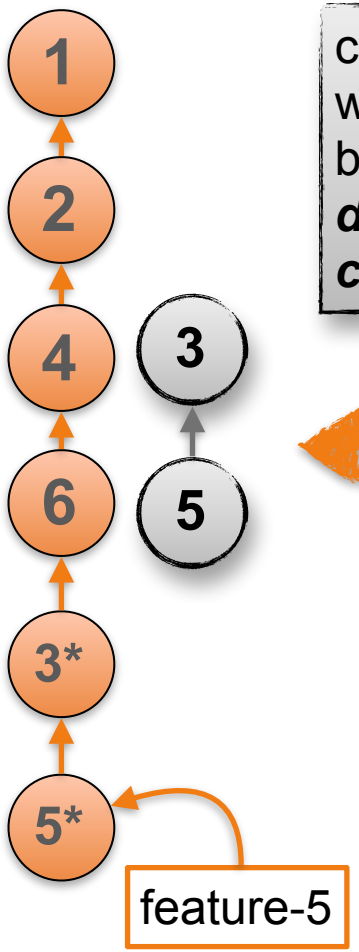
What a rebase looks like

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





replays the branch on the ***new base***



commits w/out a branch... ***dangling commits***

Rebasing to stay up to date (cont...)

- Rebase can be used instead of merge
- Why rebase?
  - Avoids merge commits
  - Keeps your branch commits in sequence
- Caution
  - Rebase re-creates each commit
    - That means commit ids change!
  - You'll need to "force push" a rebased branch

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

  - Don't rebase shared commits

3. Squash many commits into one

- Act of combining many commits into fewer/one
- A rebase *interactive mode* allows us to control the *rebase replay*

```
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

4. Cherry-pick a single commit

- We can copy single (or just a handful) of commits from one branch into another

```
git cherry-pick <commit id>
```

- Why cherry pick?
  - A branch has a edit you want to use but you don't want to merge the full branch
  - ie: A hot fix in your branch should be released, you can cherry-pick it into master (rather than your whole branch)
- Caution:
  - This does alter history

Conflicts

- Both rebase and cherry-pick can result in conflicts
- Fix the conflict then tell the command to continue

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

- Otherwise just abort the whole operation

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

Rebase vs Merge

- Merge
  - Safe, easy, non-destructive
  - Easier to see branching activity
  - Easier to revert a merge (commit)
  - Noisy, lots of extra merge commits
  - Hard to follow the history
- Rebase
  - Clean, linear history
  - Can clean up lots of in-progress commits
  - Easier to navigate w/ log, bisect and gitk
  - Flexibility, can squash and edit commits
  - Unsafe, destructive
  - No traceability (ex: when was this feature merged?)

History Strategies

- Stay up to date
  - Always rebase
  - or always merge
- When merging into master
  - Merge w/ a commit
  - or allow fast forwards
- Cleaning commits
  - Squash before merge into master
  - or leave as is

Consider: most workflows recommend committing frequently...

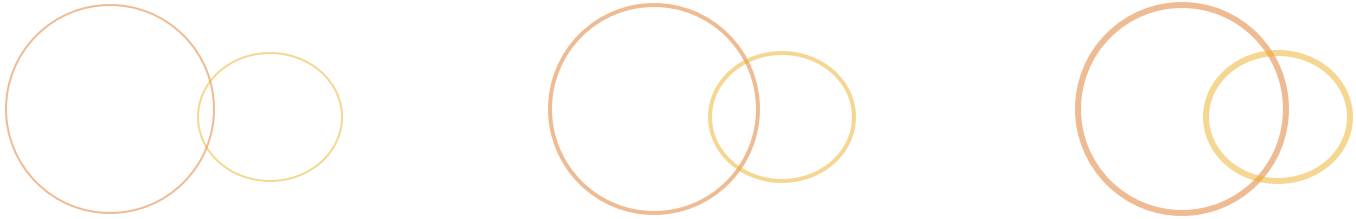
Recap: Altering History

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

Lab: Changing history

- Clone my repository:
  - https://github.com/rm-training/history-changer
  - Check the 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

GIT TOOLS & TOOLING

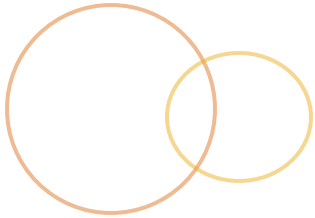
What we'll learn

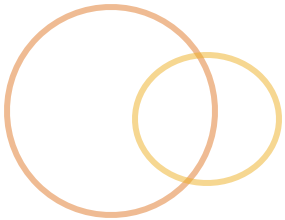
- Intro to some tools that come bundled in git
  - GitK, Git GUI
- Revisit merge & diffing tools
- Learn about hooks and how to use them

GitK

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

Git GUI





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

Merge & Diff tools

- We can configure git to use and open specialized diff viewing and merge conflict resolution applications

```
# opens the default tool
git mergetool
git difftool

# 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>
```

Some great tools
- sublime merge
- kaleidoscope

Git BFG

- A tool to help remove large, troublesome blobs
- Can also help you remove files or lines accidentally added to the repository history
  - Like passwords
- Not bundled with git
  - But uses **`git-filter-branch`**

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

Commit Message Templates

- Agree on what is expected in a commit message

./gitmessage

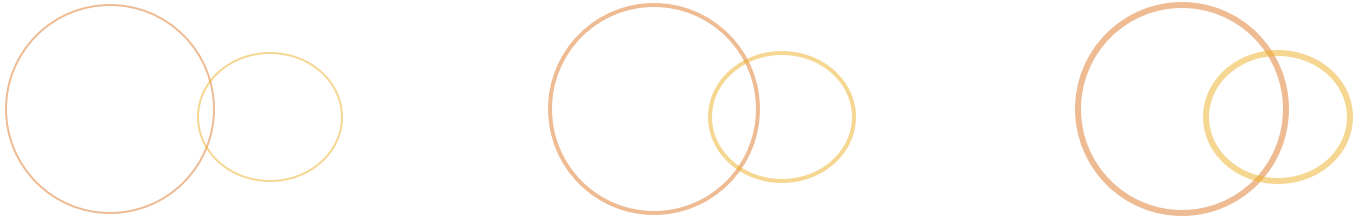
```
# 50-char summary

# detailed explanation:

Resolves:
See also:
```

- Set up a template file and tell git about it

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



module

DEBUGGING

What we'll learn

- How to debug issues in your project, using git
- Searching the log
- Using git blame to see who did what
- Git bisect to help search down bugs

Debugging with log

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

Limiting the Log

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

Searching the Log

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

Git Bisect

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

Git blame

- Ability to see who made changes in a file, line by line and in which commit

```
git blame <filename>

# limit the output
git blame -L 12,22 <filename>

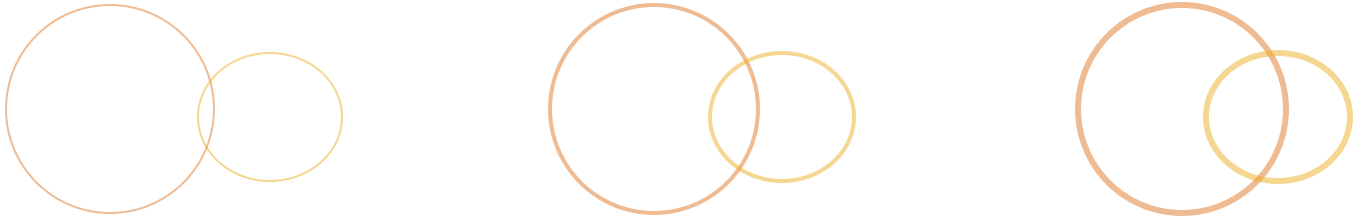
# track code movement
git blame -C <filename>
```

Recap: Debugging in git

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

Lab: Bisect and blame

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



module

FIXING ISSUES

HEAD

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

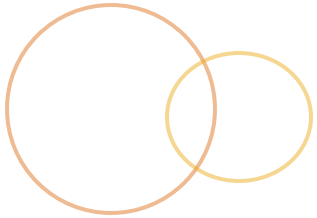
The reflog

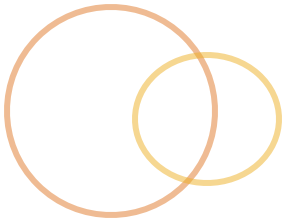
- Git keeps a log of where **HEAD** and **branch refs** have been over the past few months
  - `git reflog`
- You can use these references
  - `git show HEAD@{3}`
  - "Where HEAD was three moves ago"
- Branches also have ref logs
  - `git reflog master`
- And you can reference by date
  - `git show master@{yesterday}`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

Ancestral references

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

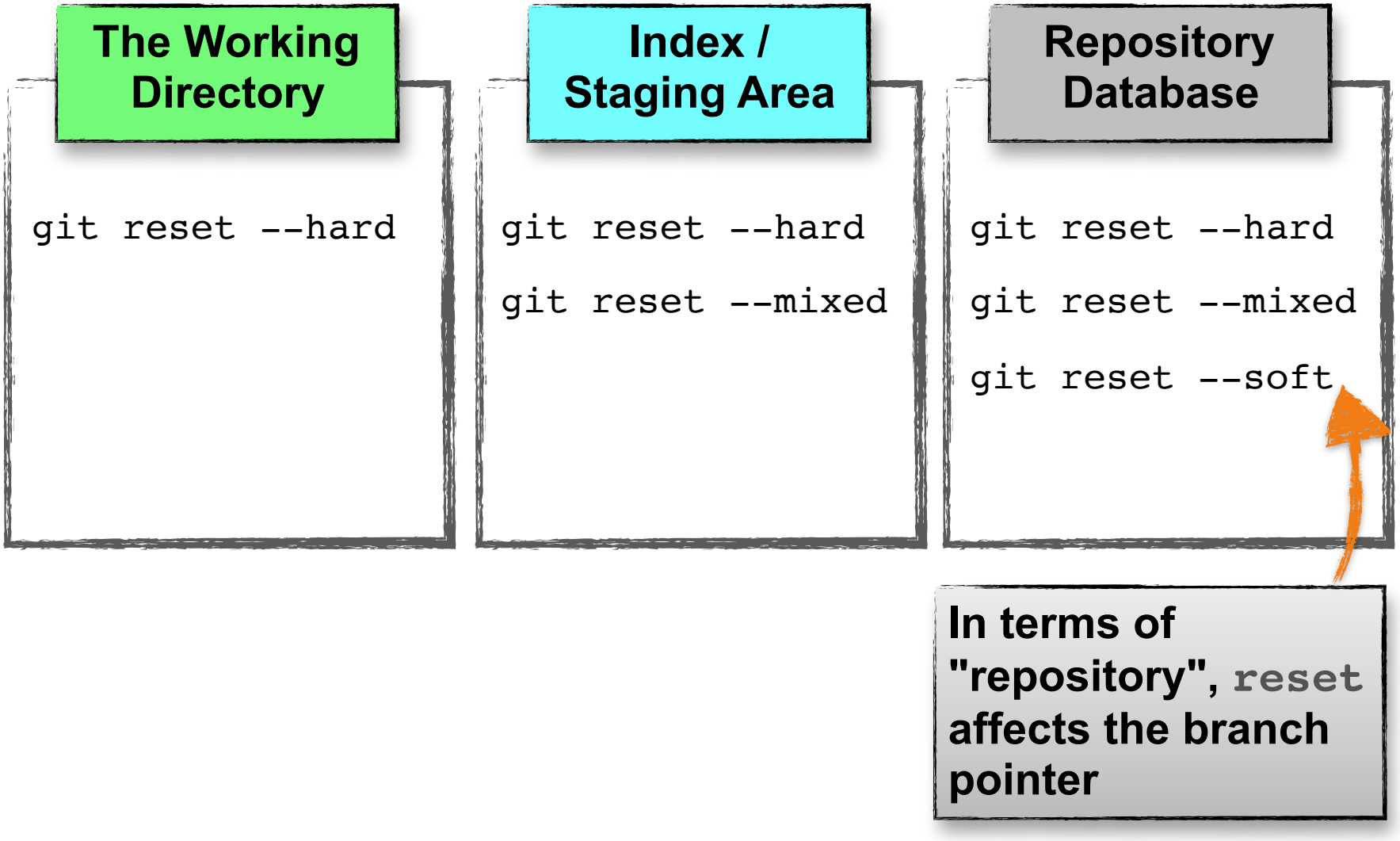
Git reset





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

Git Reset & the three trees



Git reset (continued)

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

Checkout (in terms of reset)

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

Commit recovery (and undoing)

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

Git revert

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

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>
```

Whitespace got you down?

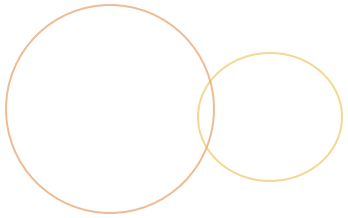
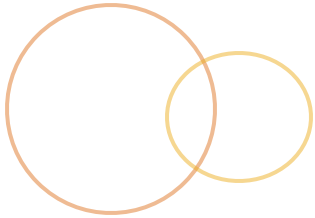
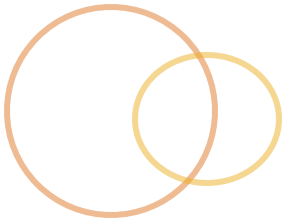
- Noticing "changed" files in working directory that you can't reset with "`git reset --hard HEAD`"?
- It may be whitespace issues
  - ```
    git rm --cached -r .
    # reset isn't enough on its own
    git reset --hard
    ```
- Then set your appropriate autocrlf
  - And have your team do the same
    ```

Recap: Fixing issues in git

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

Lab: Reflog and reset

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

module

MORE WITH GIT

Submodules & Subtrees

- Both offer a means to include other repositories within your own, *primary* repository
- **Submodule** is a pointer to a specific commit in another repository
  - You can easily push updates
  - Can be a bit more work to stay up to date
- **Subtree** is a copy of a repository that is pulled into a parent repository
  - Not easy to push updates to the original repo
  - Not the full history of the project you pull in

Git GUIs

- SourceTree
- Github Client
- Sublime Merge

Git Alternatives

- Workflow extensions like Gitflow
- Gitless

Git Worktree (v2.5+)

- "Checkout" a copy of the repository
  - Alternative to cloning a copy
  - Uses hard links; faster
  - Don't need a remote as a go-between to share changes across the two copies
- Why?
  - Run tests in parallel
  - Work on a different branch w/out existing current working directory

```
git worktree add ../new-worktree-dir some-existing-branch
```

Git enforced policies

- Use hooks to enforce a policy
  - Commit message format/content [commit-msg]
  - User access to specific files or directories
  - Code linting
  - Static analysis
  - Run tests
- How:
  - Set up integrations online (github/gitlab)
  - Local hooks
  - Server side hooks (ex: Git SCM)

Best Practices in Git

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

Copyright © 2015-2019 Ryan Morris

More Resources

- Git Immersion
- You could have invented git

Copyright © 2015-2019 Ryan Morris