

THIS IS GIT. IT TRACKS COLLABORATIVE WORK
ON PROJECTS THROUGH A BEAUTIFUL
DISTRIBUTED GRAPH THEORY TREE MODEL.

COOL. HOW DO WE USE IT?

NO IDEA. JUST MEMORIZE THESE SHELL
COMMANDS AND TYPE THEM TO SYNC UP.
IF YOU GET ERRORS, SAVE YOUR WORK
ELSEWHERE, DELETE THE PROJECT,
AND DOWNLOAD A FRESH COPY.



WORKING WITH / MASTERING GIT

DAVE WADE-STEIN

GOALS FOR THIS CLASS

Understand
how Git works
under the
hood so that
we may better
understand
the commands
we use

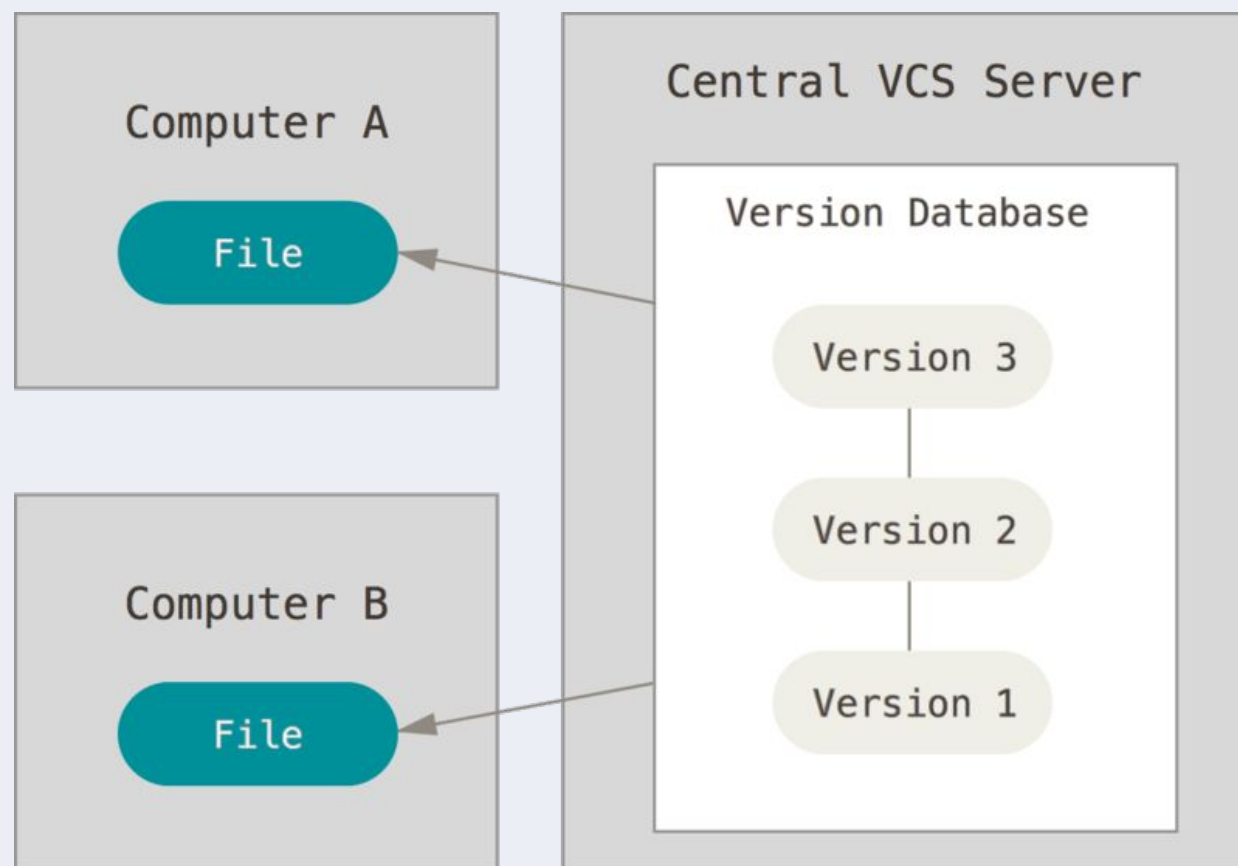
Learn some
new ways to
use
commands
we may
already be
familiar with

Learn some
new Git
commands

WHAT IS VERSION CONTROL?

- a system that records changes to a file or set of files over time
- why do we want this?
 - you can recall specific versions later
 - you can see the history of changes to the file(s)

OLD WORLD: VERSION CONTROL WITH A CENTRAL SERVER



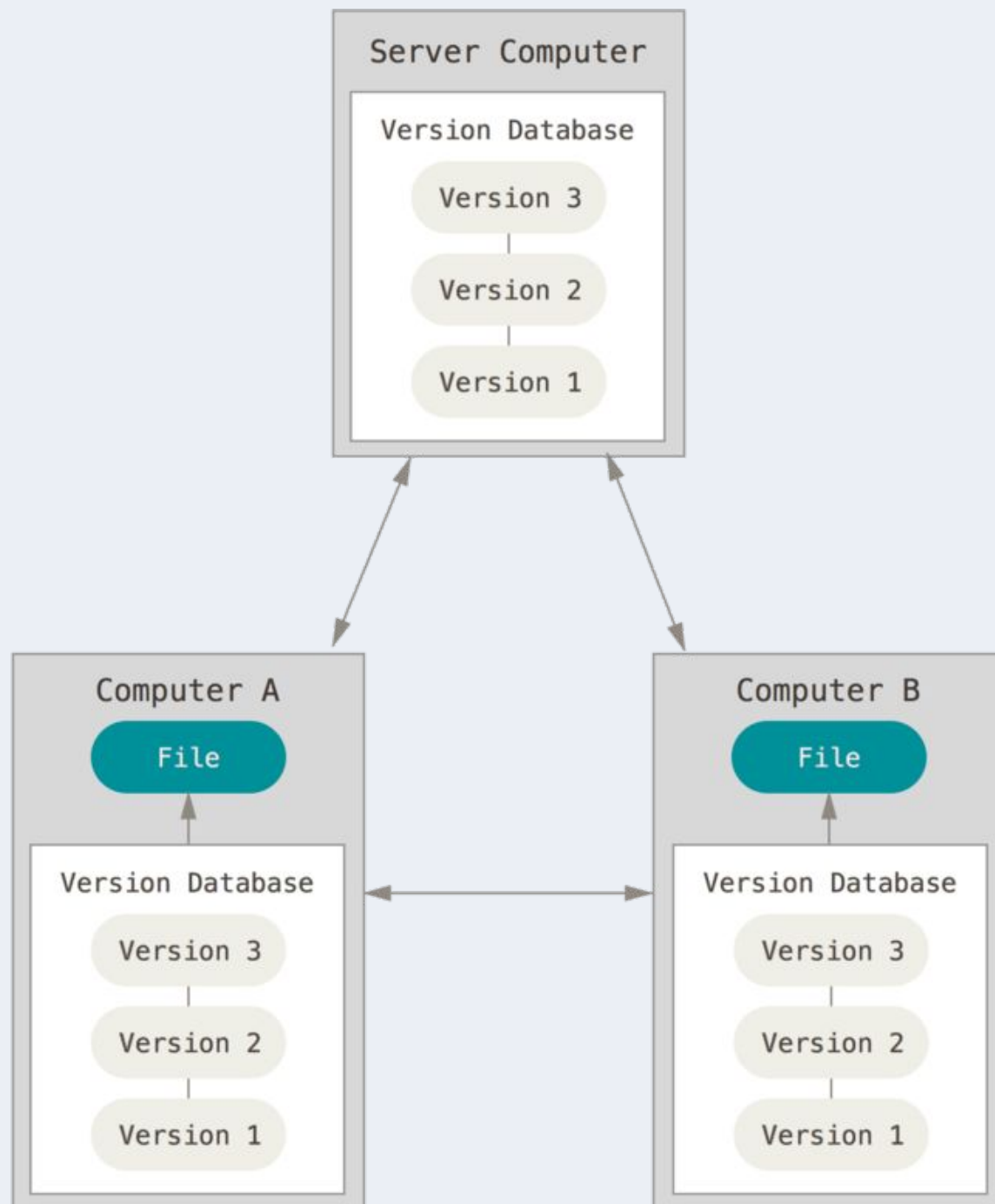
- single central server
- e.g., Subversion (svn), Perforce
- advantages
 - everyone knows what everyone else is doing*
 - fine-grained control
- disadvantages
 - history not available locally
 - single point of failure
 - branching/merging difficult

* ...for the most part

SOURCE:

<https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control>

NEW WORLD: DISTRIBUTED VERSION CONTROL



- All clients fully mirror the repository, including history
- e.g., Git, Mercurial, Bazaar
- All clients act as backups
 - ...no single point of failure!
- Collaboration easy
 - supports many workflows

BRIEF HISTORY OF GIT

- Linux (1991–)
 - No real version control from 1991-2002
 - **bitkeeper** (2002-2005)
 - **git** created in 2005 after relationship w/**bitkeeper** broke down
- Goals
 - speed!
 - simplicity
 - strong support for non-linear development
 - fully distributed
 - able to handle large projects, e.g., Linux kernel efficiently (in terms of both speed and size)

THIS IS GIT. IT TRACKS COLLABORATIVE WORK
ON PROJECTS THROUGH A BEAUTIFUL
DISTRIBUTED GRAPH THEORY TREE MODEL.

COOL. HOW DO WE USE IT?

NO IDEA. JUST MEMORIZE THESE SHELL
COMMANDS AND TYPE THEM TO SYNC UP.
IF YOU GET ERRORS, SAVE YOUR WORK
ELSEWHERE, DELETE THE PROJECT,
AND DOWNLOAD A FRESH COPY.



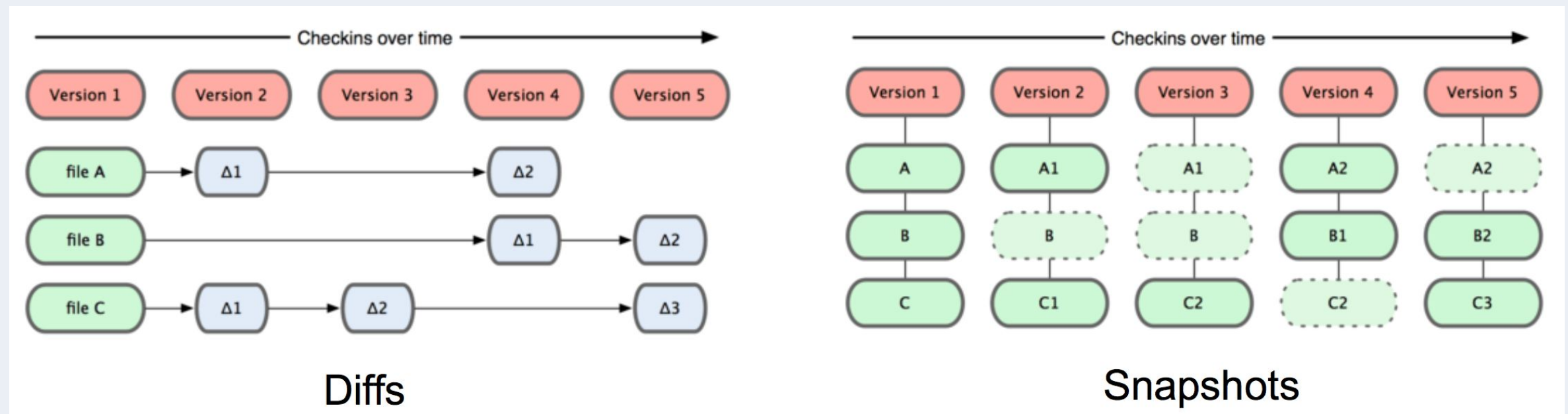
WHAT IS GIT?

WHAT IS GIT?

- Under the hood Git is simply a key/value store
 - You insert any content into Git, and Git will give you back a key that you can use to retrieve the content at any time
- Git stores *snapshots* of each file, not diffs between files
- Almost every operation is carried out on your local repo
- Branching is super easy and encouraged as part of your daily workflow

SNAPSHOTS VS. DIFFS

- Git stores data as snapshots of your repository files in each commit, as opposed to a delta, or difference



- In this sense, Git is more like a mini-filesystem with powerful tools to access it, rather than just a version control system

GIT HAS INTEGRITY

- Everything in Git is hashed (using the SHA-1 algorithm) before it's stored, and from then on is referred to by its hash (e.g., `24b9da6552252987aa493b52f8696cd6d3b00373`)
 - therefore, it's impossible to change contents of a file or have a file get corrupted w/o Git knowing about it
- Commits and other objects in the Git database are referenced by their hash
 - ...although we can usually avoid having to use the hash when interacting with Git

GIT GENERALLY ADDS DATA

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

FIRST-TIME GIT SETUP

- Git's configuration is stored as a plain text file based on the configuration "level"

`~/.gitconfig`

- this is your global configuration, i.e., it affects all of your repositories

`.git/config`

- this is your local configuration, i.e., it affects only the repo in which it is located
- There is also a systemwide configuration for all users in the file `/etc/gitconfig`, but these days we don't seem to have multiple users logged in to the same machine, do we?

SETTING CONFIG VALUES

- set config values

```
git config <key> <value>
```

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

```
git config --local ...
```

- check config values

```
git config --list
```

LAB: SETUP

- Set your name/email

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

```
git config --global user.email "dws@..."
```

- ...and your text editor setting if you want

- by default, Git uses `vi` or `$EDITOR`

- to use TextWrangler on Mac (after installing command line tools for TextWrangler)

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

LAB: SETUP (CONT'D)

- Check config

```
git config --list
```

```
git config <key> (e.g., git config core.editor)
```

- You can also simply look at the config file directly

```
vi/sublime .git/config
```

```
vi/sublime ~/.gitconfig
```


THIS IS GIT. IT TRACKS COLLABORATIVE WORK
ON PROJECTS THROUGH A BEAUTIFUL
DISTRIBUTED GRAPH THEORY TREE MODEL.

COOL. HOW DO WE USE IT?

NO IDEA. JUST MEMORIZE THESE SHELL
COMMANDS AND TYPE THEM TO SYNC UP.
IF YOU GET ERRORS, SAVE YOUR WORK
ELSEWHERE, DELETE THE PROJECT,
AND DOWNLOAD A FRESH COPY.



GIT BASICS

CREATING OUR FIRST REPO

- Create a directory for the repo

```
mkdir myrepo
```

```
cd myrepo
```

- Initialize the repo

```
git init
```

- Or do all three steps in one

```
git init myrepo
```

LAB: ADD SOMETHING TO OUR EMPTY REPO

- Create a file using an editor or
`echo something > firstfile`

`git add firstfile # STAGE THE FILE`

`git commit # COMMIT THE STAGED ITEMS`

`git log # CHECK THE LOG`

CLONE AN EXISTING REPO

- make a copy of an existing Git repo

- try...

`git clone`

<https://github.com/davewadestein/clone-me>

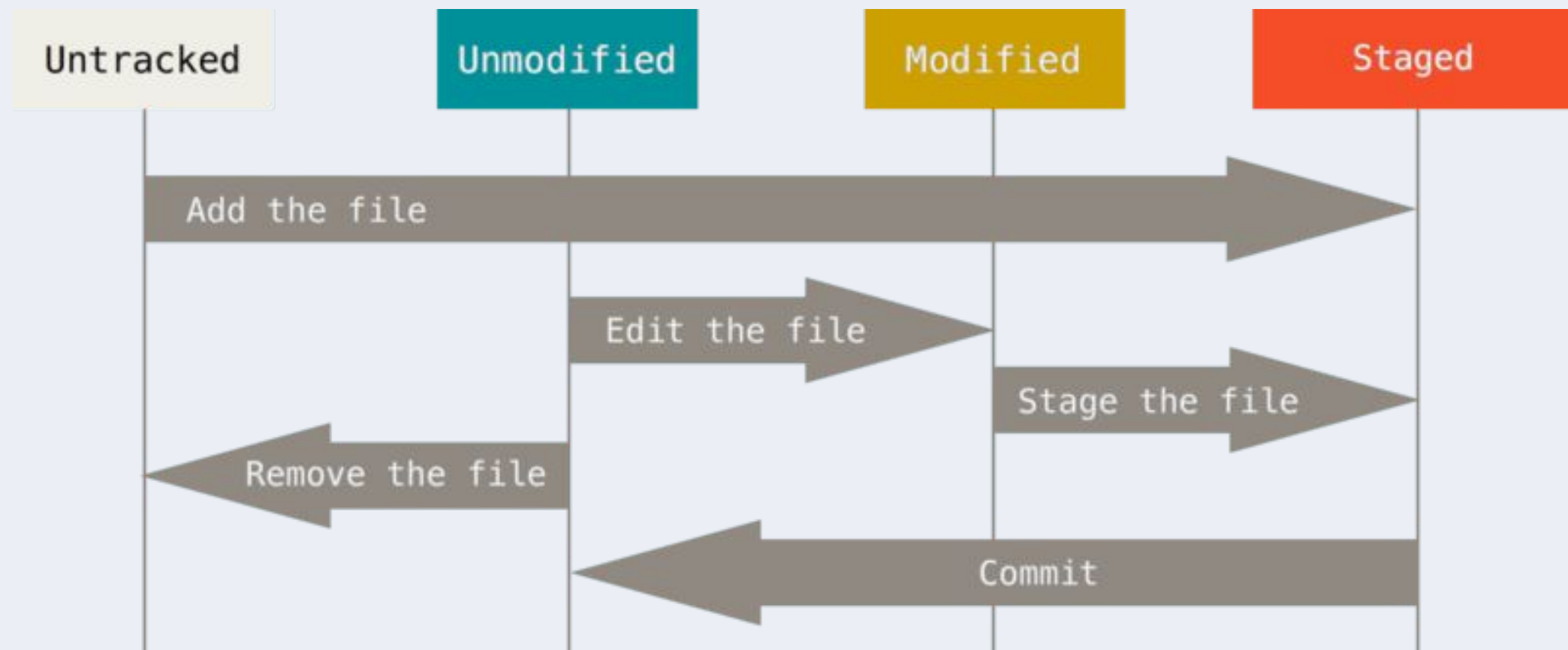
THE "WORKING COPY"

- Git's idea of a "working copy" is very different from the working copy you get by checking out from an SVN repo
 - Git makes no distinction between the working copy and the central repo—they're all full-fledged Git repos
 - this makes collaborating with Git fundamentally different than with SVN
 - SVN depends on the relationship between the central repo and the working copy, whereas Git's collaboration model is based on repo-to-repo interaction
 - Instead of checking a working copy into SVN's central repository, you push or pull commits from one repository to another

(UN)TRACKED FILES

- **tracked**
 - = a file that Git knows about
- **untracked**
 - = files in your working area that are not in the repo and are not staged

BASIC GIT WORKFLOW



- add/modify files in your working dir
- stage files, which adds snapshots to your staging area
- commit, which takes files from the staging area and stores the snapshot permanently in your repo (**.git** dir)
- we'll see these again when examine the 3 areas

GETTING STATUS

- **git status** returns info about your current branch
 - changes added to the staging area
 - untracked files
 - tracked files which have been modified
 - some helpful tips for undoing things
 - (hold that thought!)

SHORT (ABBREVIATED) STATUS

- `git status -s`
 - left column is for staging area
 - right column is for working dir
 - **A** = added, **M** = modified, **??** = untracked

```
$ git status -s
A  blankfile
MM foo.txt
M  newfile
?? somefile
```

USING GIT ADD TO "STAGE" FILES

- stage a new file
`echo something > newfile`
`git add newfile`
- stage a modified file
`echo stuff >> otherfile`
`git add otherfile`
- stage a directory: `git add <dirname>`
- wildcards
`git add *.txt`
`git add '*.txt'` (what's the difference?)
- add everything: `git add .`

IGNORING FILES

- we typically have files we want Git to *ignore*
- add a file named **.gitignore** either in root of project, or per directory
- **.gitignore** can include regular expressions (glob), e.g

***.pdf**

***.out**

logs/*.log

or...

tmp/

/.build # only ignore **.build** in current dir

- **<https://github.com/github/gitignore>**
(prefab ignores)

HOW TO GET HELP

```
git help <command>
```

```
git <command> --help
```

```
man git-<command>
```

e.g.,

```
git help status
```

```
git status --help
```

```
man git-status
```

- You can access the help even while offline!

RECAP: BASICS

- `git init` = initialize a repo
- `git status` = track staging area and working dir changes
- `git add` = stage files
- `git diff` = view differences between staging area and working area or between staging area and repo
- `git commit` = commit staged changes
- `git log` = view history

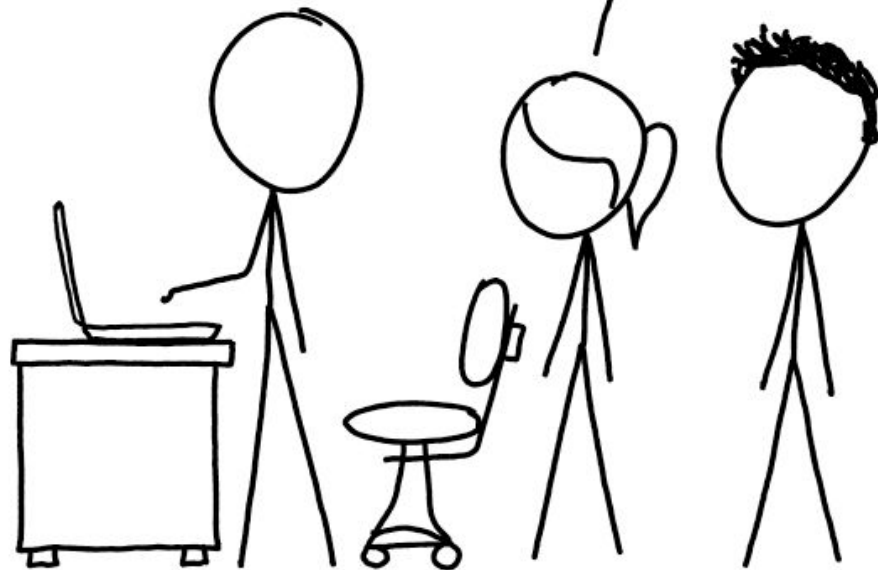
LAB: BASICS

- assuming you've been following along, not much to do here, but if you revisit these notes...
- create a new repo and a new file within it
- stage and commit the file
- check the log
- edit the file and add some text, then check status and diff
- stage your change, then check the status and diff
- make one more edit to the file, then check status and diff

THIS IS GIT. IT TRACKS COLLABORATIVE WORK
ON PROJECTS THROUGH A BEAUTIFUL
DISTRIBUTED GRAPH THEORY TREE MODEL.

COOL. HOW DO WE USE IT?

NO IDEA. JUST MEMORIZE THESE SHELL
COMMANDS AND TYPE THEM TO SYNC UP.
IF YOU GET ERRORS, SAVE YOUR WORK
ELSEWHERE, DELETE THE PROJECT,
AND DOWNLOAD A FRESH COPY.











GIT IN DEPTH

THE GIT OBJECT DATABASE

- Git stores four types of objects in the repository
 - **blob** = the contents of a file (either text or binary) at some point in the history of your project
 - **tree** = folders in your project
 - **commit** = a special object created whenever you commit
 - **tag** = "bookmarks" to let you remember a specific commit
- We will see examples of each!

LET'S TAKE A LOOK AT .GIT

Name	
▶ 	branches
	config
	description
	HEAD
▶ 	hooks
▶ 	info
▶ 	objects
▶ 	refs

- The entire repository is here
- **config** is the local or per-repo configuration
- **description** is a text file describing the repo
- We will look at the other file and folders shortly, but to begin with, we will focus on the **objects** folder

UNDER THE HOOD (CONT'D)

- there are actually two types of Git commands
 - "porcelain" commands
 - "plumbing" commands
- we don't normally use the plumbing commands, but we'll try a couple here so we can understand how Git stores things

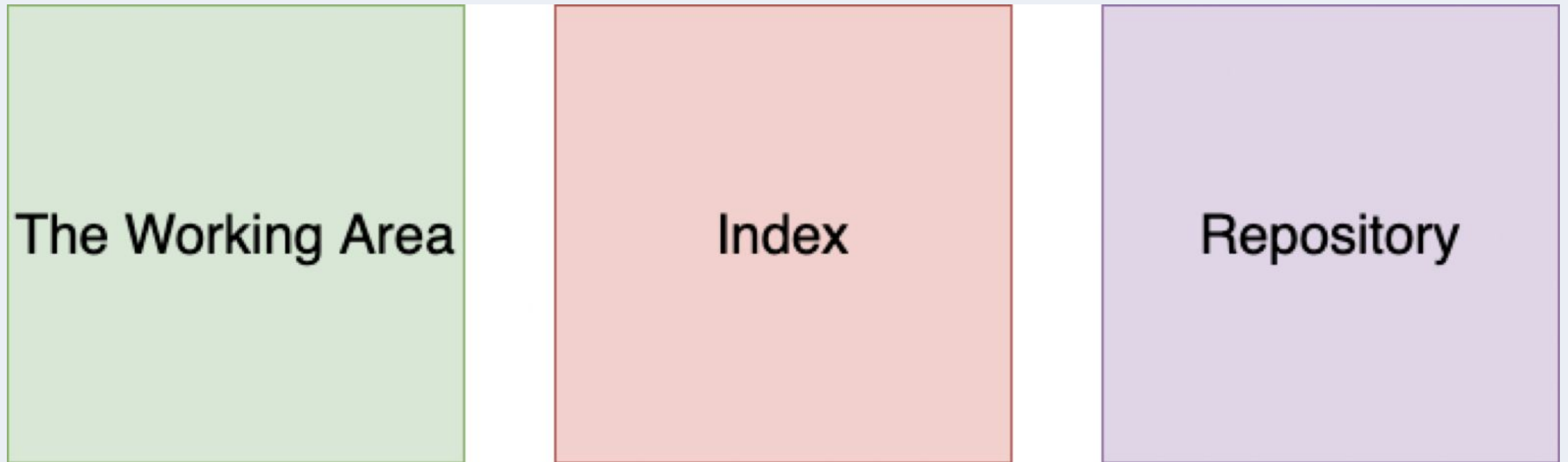
git hash-object computes the SHA-1 hash of an object

git cat-file dumps out contents of an object

git ls-tree dumps out contents of a tree object

git ls-files shows files in index and working tree

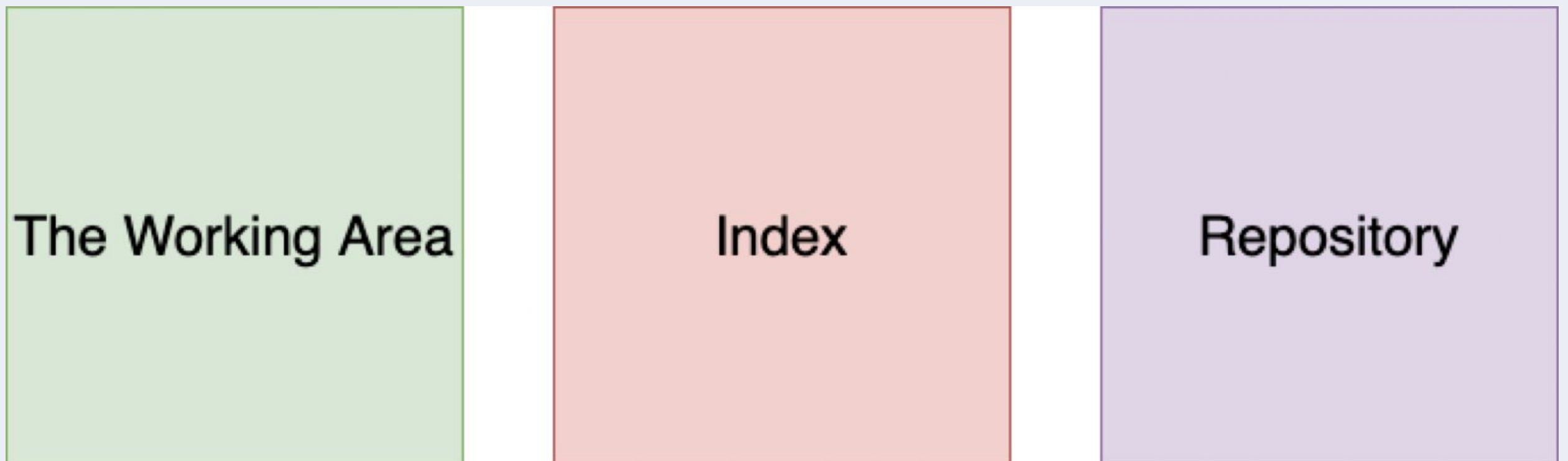
THE THREE FOUR AREAS



- Git projects store things in four separate areas
 - working area = where you edit your files
 - index (staging area) = what goes into next commit
 - repo(sitory) = where Git stores your data (.git)
- we will discuss the fourth area, the stash, later

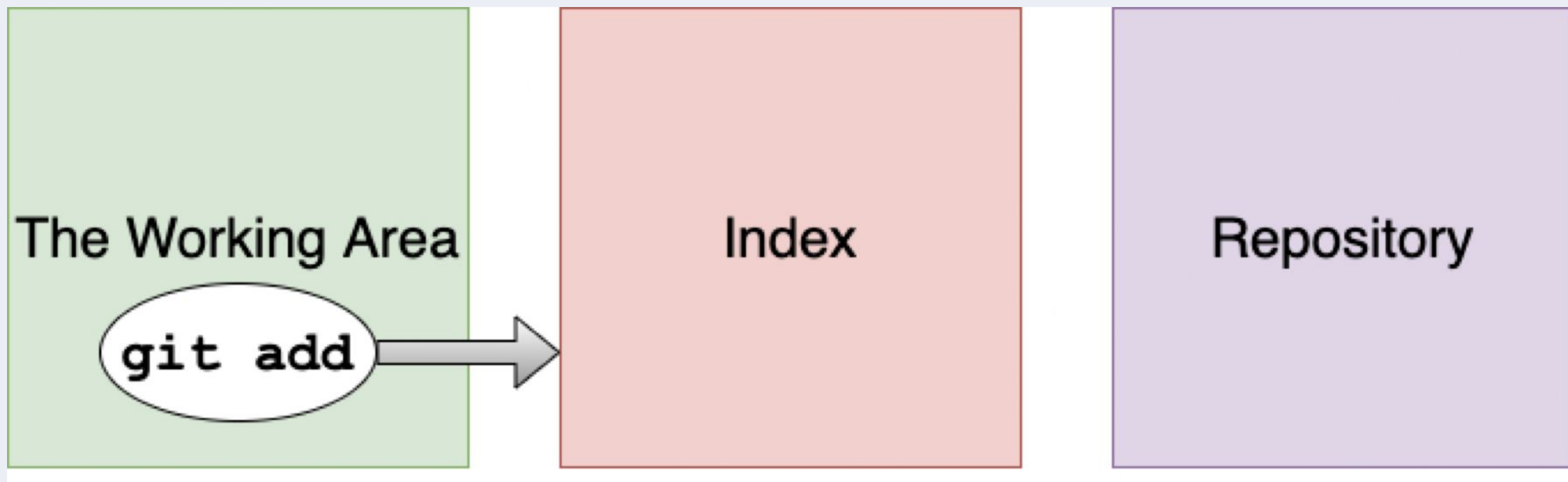
HOW DO GIT COMMANDS AFFECT THREE AREAS?

- To truly understand a `git` command, we need to understand two things:
 - ...how does it move data between the areas?
 - ...how does it change the repo?



STAGING FILES WITH GIT ADD

- staging a file means copying the file from the working area to the index, i.e., preparing the file for committing



GIT ADD DOES SEVERAL THINGS

- actually a multipurpose command used to
 - begin tracking new files
 - stage files
 - marking merge-conflicted files as resolved
- might be best to think of it as ***add this content to the next commit*** rather than ***add this file to the repo***

ADD SOME BUT NOT ALL CHANGES: `GIT ADD --PATCH/-p`

- interactively choose which of your changes ("hunks") to add to the index
- gives you a chance to review the difference before adding modified contents to the index
- if you stage some of your changes from a file and not others, you will end up with the same file being both staged for commit and NOT staged for commit
 - also happens if you stage a file for commit, and then modify it afterwards

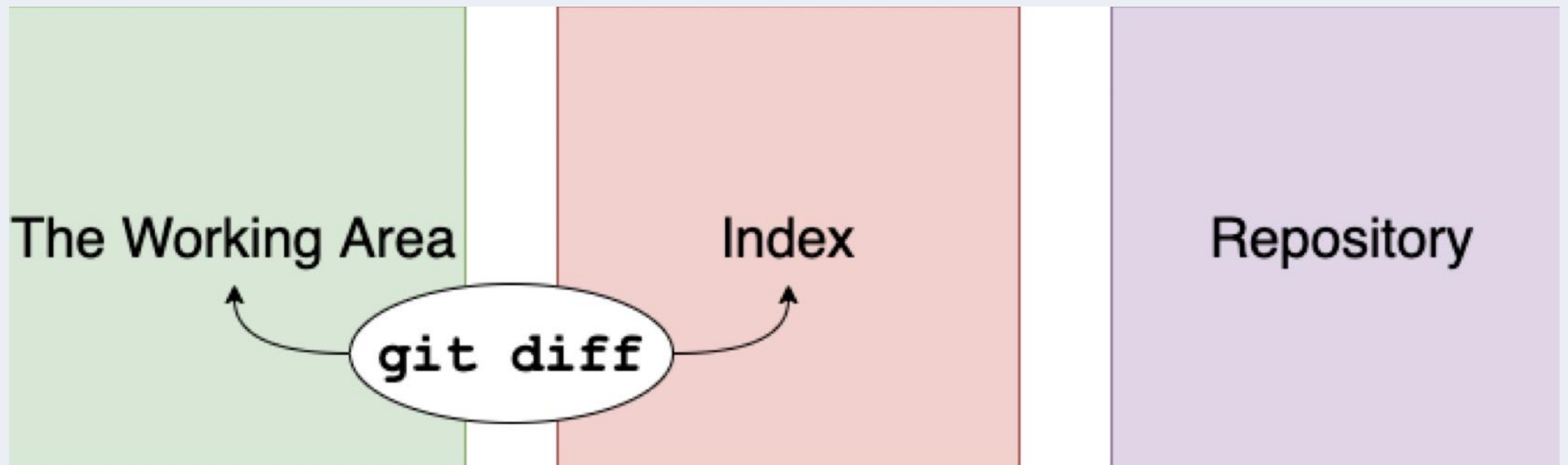
LAB: GIT ADD --PATCH/-p

- make 2+ changes to a single file, far apart from each other (if they're close, Git will consider them 1 change)
- use **git add -p** to interactively split the changes into separate hunks
- stage one or more hunks, but leave at least one hunk unstaged
- use **git status** to see that the file is both staged and unstaged
- commit

SEEING WHAT'S CHANGED PT. 1

`git diff`

- shows the difference between the working area and the staging area
 - `git status` can be used instead



SEEING WHAT'S CHANGED PT. 2

`git diff --staged` (or `--cached`)

- compares staged files against the repo, i.e., what is ready to commit but hasn't yet been committed?



COMMITTING

- Now that we've staged files, we can commit them
- Committing copies the files from the staging area into the repo



and...

COMMITTING (CONT'D)

- ...
 - creates blob objects in the Git object database which represent the new or changed files
 - creates a commit object in the Git object database
 - possibly creates tree objects in the Git object database

COMMITTING (CONT'D)

`git commit`

- prompts for a commit message
- recommended
- commit will abort if no message is given

`git commit -m 'My commit message'`

- not recommended (unless you edit history later)

`git commit -v`

- include diffs in the message

SKIP THE STAGING AREA

`git commit -a`

- let's you skip the staging step if it doesn't fit your workflow
- automatically stage every *tracked* file before doing the commit, i.e., letting you skip `git add`

REMOVING FILES

- To remove a file (and stage the removal)

```
git rm [-r] <filename>
```

(-r is recursive, just like regular **rm**)

- If you forget and simply **rm** the file, you must stage the removal manually:

```
rm <filename>
```

```
git add <filename> (!)
```

- Therefore, **git rm** removes data from the working directory and alters the index

REMOVING FILES (CONT'D)

- To have Git untrack a file without affecting working dir
`git rm --cached <filename>`
- In other words, remove a file from the index, but leave it in the working directory as an untracked file

VIEWING COMMITS

- show latest commit on your branch

```
git show
```

- show a specific commit

```
git show <commit>
```

- Commits are referenced by their SHA-1 hash

```
git show 7482e39151c341414431a06cb7e8e75...
```

- Obviously difficult to remember SHA-1 hashes, but you can use the abbreviated form
 - need to specify 4+ digits; enough to be unambiguous

```
git show 7482
```

LEARNING FROM HISTORY (THE LOG)

`git log`

- tons of options (kinda like the `ls` command)
 - `--oneline` = abbreviated hash, one commit per line
 - `--grep=<p>` = only show commit whose message matches pattern `<p>`
 - `--stat` = generate a diffstat, i.e., ratio of +ed/-ed lines
 - `--graph` = generate a text-based graph of commit history

LEARNING FROM HISTORY (THE LOG-CONT'D)

- <n> = limit output to last n commits
- p <path> = only show commits which touched <path>, and include the diff in the output
- main..**branchname** = show (unmerged) commits on **branchname** only (rather than continuing)
- if you're **in** the branch, you can shorten it to **main..**

MOVING/RENAMING FILES

- To move a file and auto-stage the move

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

- This is equivalent to

```
mv <old name> <new name>
```

```
git rm <old name>
```

```
git add <new name>
```

- Git doesn't explicitly track file movement
 - when you rename a file, no metadata is stored about the rename
 - Git is pretty smart about figuring it out, though

LAB: RENAME

- `mv firstfile secondfile`
 - check the status to see what Git noticed
 - `mv secondfile firstfile`
- use git to rename `firstfile` to `secondfile`
 - check the status, commit
 - `rm secondfile` (do not use `git rm`)
 - check the status
 - recover the file as Git suggests
- tell Git to untrack (remove from Git database) `secondfile`
 - check the status
 - unstage deletion from Git database as Git suggests

PROMPT/COLORS

- If you want the Git command completion you've seen me take advantage of...

`https://github.com/git/git/blob/master/contrib/completion/git-completion.bash`

- “Sexy bash prompt”

<https://git.io/v6QMn>

- Add it to your `~/ .bash_profile`

THIS IS GIT. IT TRACKS COLLABORATIVE WORK
ON PROJECTS THROUGH A BEAUTIFUL
DISTRIBUTED GRAPH THEORY TREE MODEL.

COOL. HOW DO WE USE IT?

NO IDEA. JUST MEMORIZE THESE SHELL
COMMANDS AND TYPE THEM TO SYNC UP.
IF YOU GET ERRORS, SAVE YOUR WORK
ELSEWHERE, DELETE THE PROJECT,
AND DOWNLOAD A FRESH COPY.



**THIS WILL
BE YOUR
UNDOING**

UNDO (AMEND/CLEAN)

- What we'll cover in this section:
 - How to amend your last commit
 - How to discard modifications to a file in your working directory
 - How to remove untracked files (not really an undo, but helpful to know at this point)

AMEND THE LAST COMMIT

`git commit --amend`

- amends/changes the commit message
- AND/OR add changes you missed to your staging area before executing this command and they will be committed and added to the latest commit

UN-STAGING A STAGED FILE

- let's modify/add a file, then stage the change
 - now run `git status`
 - what does it tell you to do?

`git restore --staged file`

- older versions of Git say: `git reset HEAD file`
- `git reset` is a difficult (and potentially dangerous) command which we will look at in depth soon

UN-MODIFYING A MODIFIED FILE

- Discard local changes to a file

`git restore file(s)`

- older versions of Git say: `git checkout -- file(s)`
 - copies file(s) from the index to the working area
- Remove all untracked files (be careful with this one!)

`git clean -f`

RECAP: UNDO

- Change last commit/commit message

```
git commit --amend
```

- Un-stage a staged file

```
git restore file
```

```
(git reset HEAD file)
```

- Discard changes in your working dir

```
git restore file
```

```
(git checkout -- <file>)
```

- Remove untracked files with

```
git clean -f
```

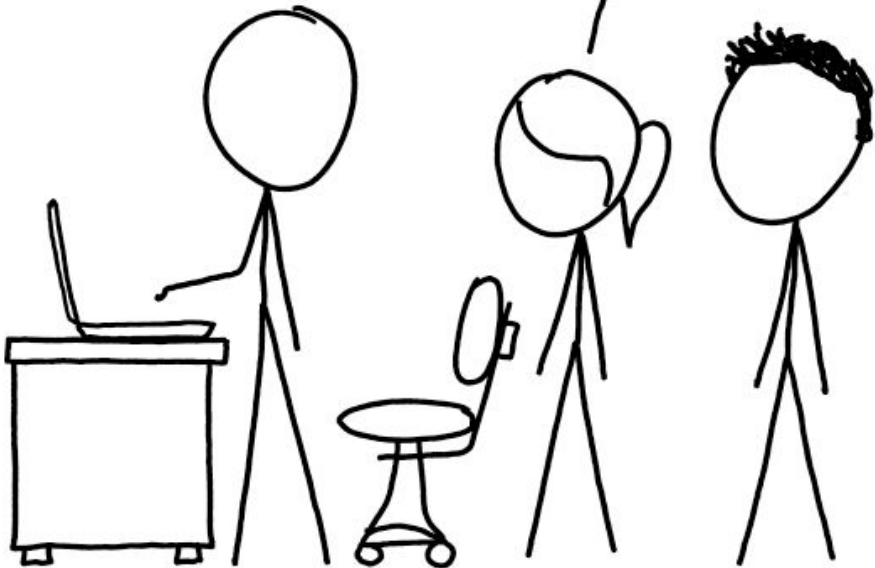
LAB: UNDO

- Create a new file
- Edit an existing file
- Discard the working directory changes
- Amend the log message from your last commit
- Create a new file, stage it, and amend it to last commit

THIS IS GIT. IT TRACKS COLLABORATIVE WORK
ON PROJECTS THROUGH A BEAUTIFUL
DISTRIBUTED GRAPH THEORY TREE MODEL.

COOL. HOW DO WE USE IT?

NO IDEA. JUST MEMORIZE THESE SHELL
COMMANDS AND TYPE THEM TO SYNC UP.
IF YOU GET ERRORS, SAVE YOUR WORK
ELSEWHERE, DELETE THE PROJECT,
AND DOWNLOAD A FRESH COPY.



BRANCHING OUT

BRANCHING

- Branching allows us to pursue a new line of development, separate and distinct from **main**
- Why branch?
 - try out new things
 - keep **main** stable
 - collaborate with others
- To really understand branching, we need to understand how Git stores data
 - ...recall that Git stores snapshots not changesets

COMMITTING

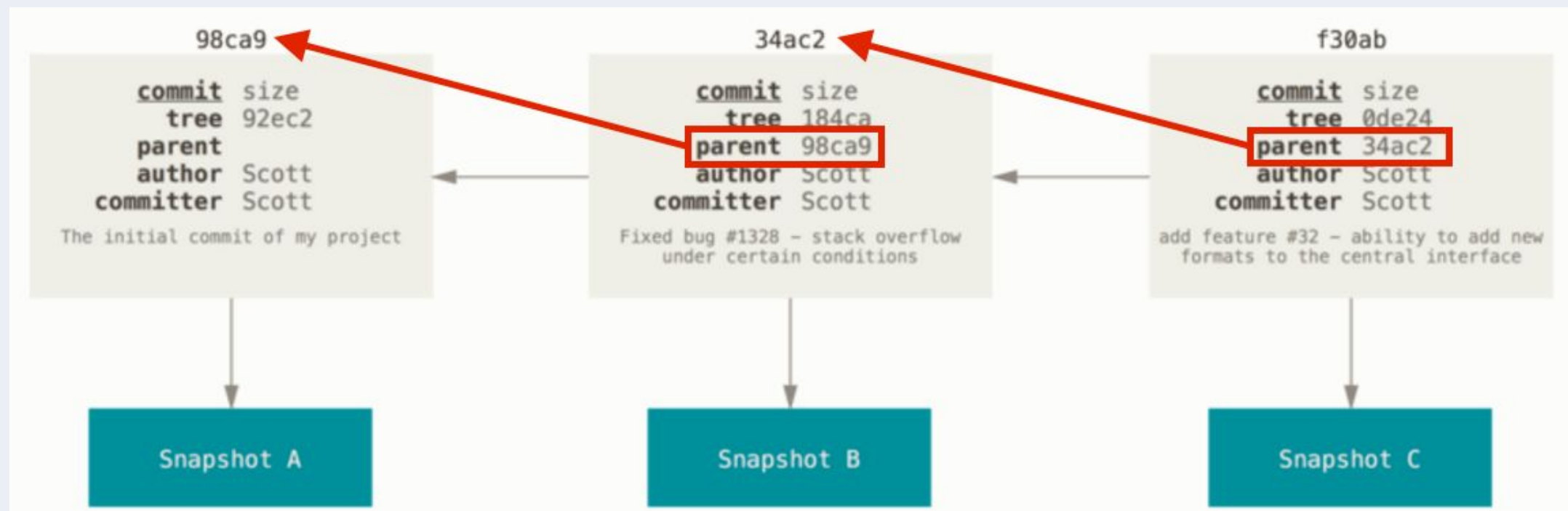
- Git stores a commit object containing a reference to the snapshot of the content you staged as well as...
 - author's name/email, commit message, and...
 - references to commit(s) that directly came before
 - 1 parent for normal commit (0 for initial commit)
 - 2+ parents for a merge of 2+ branches

```
ae668..
```

commit		size
tree	c4ec5	
parent	a149e	
author	Scott	
committer	Scott	
my commit message goes here and it is really, really cool		

WHAT IS A BRANCH?

- branch = simply a movable reference to a commit
- default branch name is **main** (just a name, nothing special about it)
- as you start making commits, Git creates a branch **main** that points to the last commit you made
- with each commit, pointer moves forward automatically



GIT BRANCH

- list all your branches

git branch

git branch -a (include remote branches)

git branch -v (verbose)

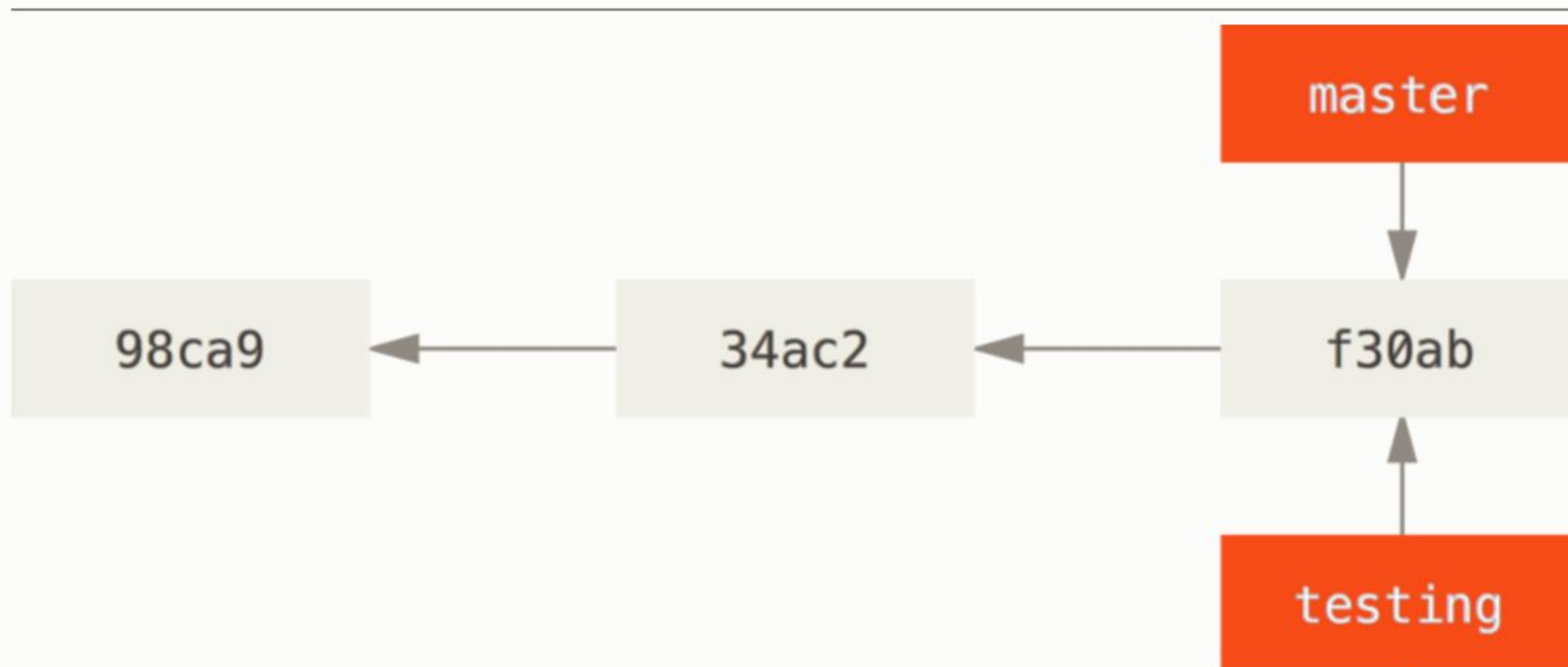
- create a branch

git branch <name>

CREATING A BRANCH

- creating a new branch creates a new reference to the same commit you're currently "on"
- it's fast! (all that's needed is to write the SHA1 hash to a file in your `.git` subdir...40 characters + newline!)

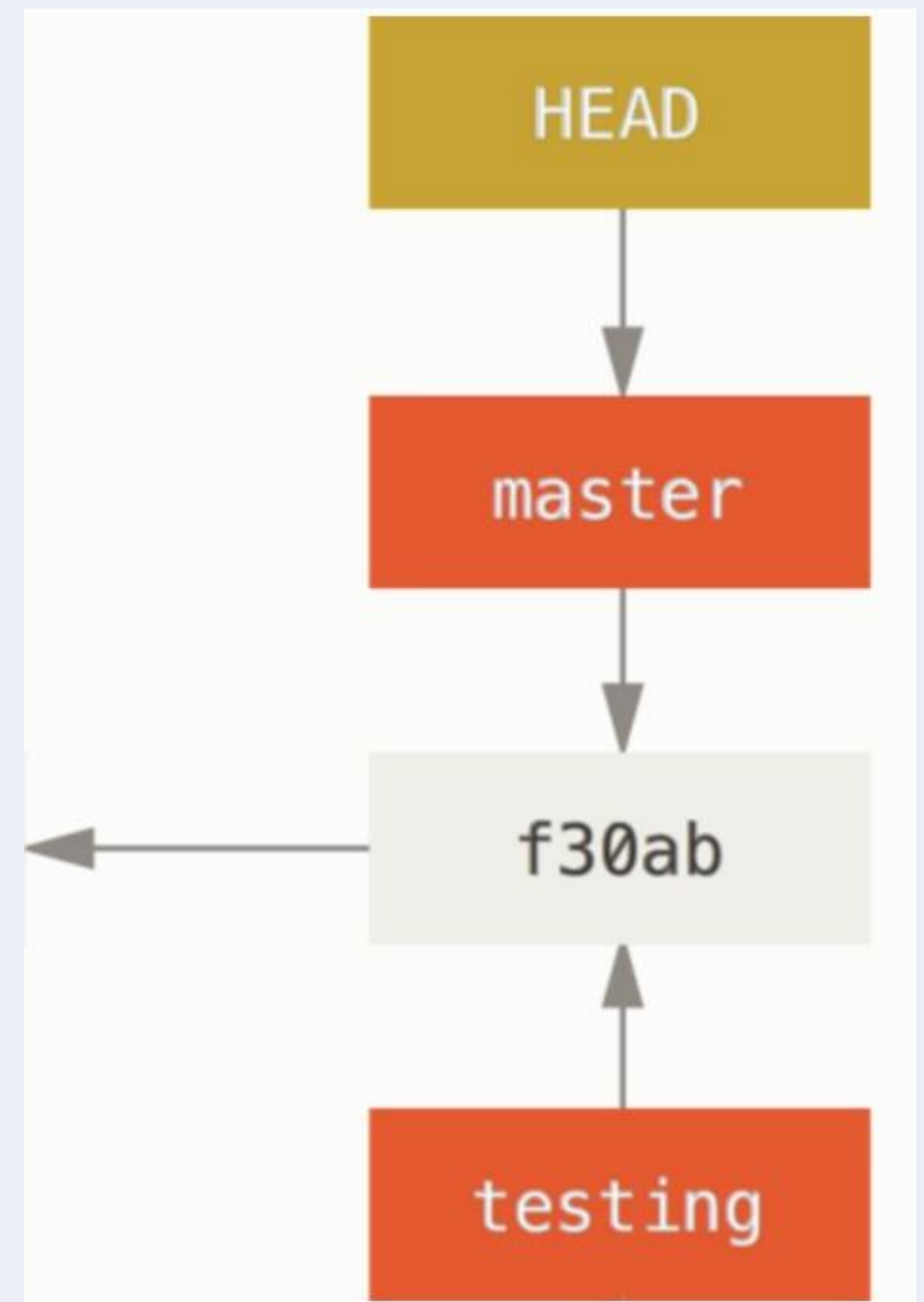
```
$ git branch testing
```



HEAD

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

git log --oneline --decorate



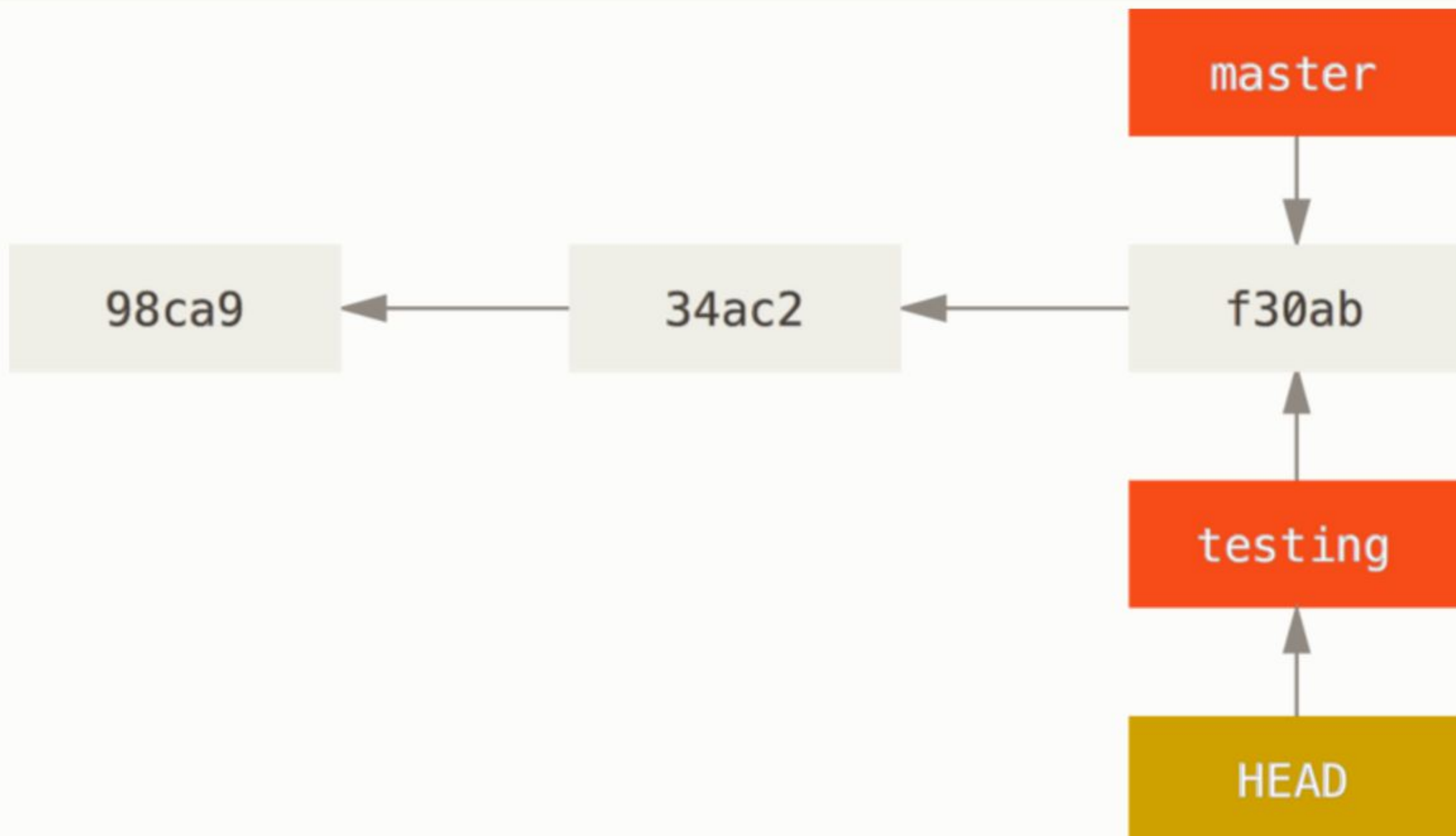
GIT SWITCH (CHECKOUT)

- does multiple things...new commands added to simplify
 - check out (“switch to”) specific branch/tag
git switch <branch/tag>
 - check out a specific commit (leaving you in “detached head” state)
git checkout <commit-id>
 - discard changes in the working dir
git checkout -- <file>
 - create a branch AND switch to it
git checkout -b <name>

HEAD REDUX

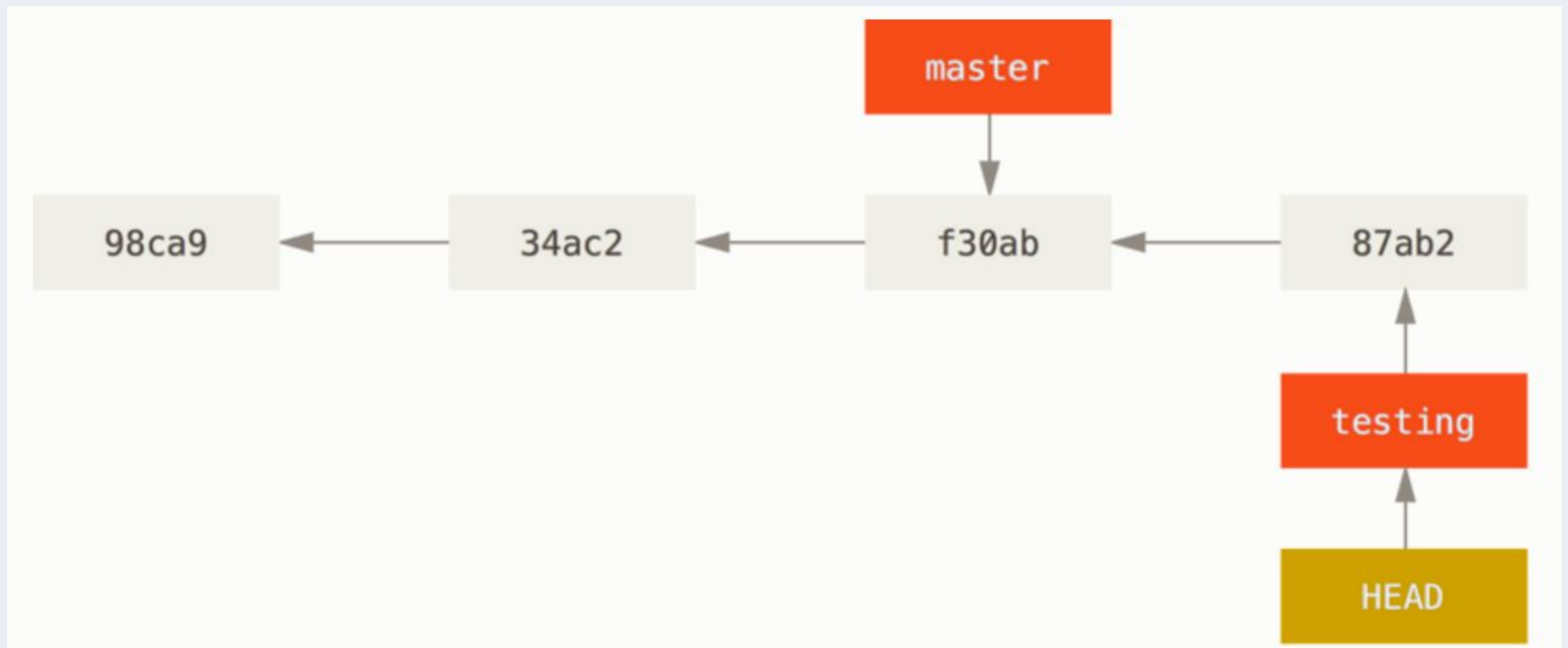
- `git switch/git checkout` updates where **HEAD** points

```
$ git checkout testing
```



HEAD REDUX (CONT'D)

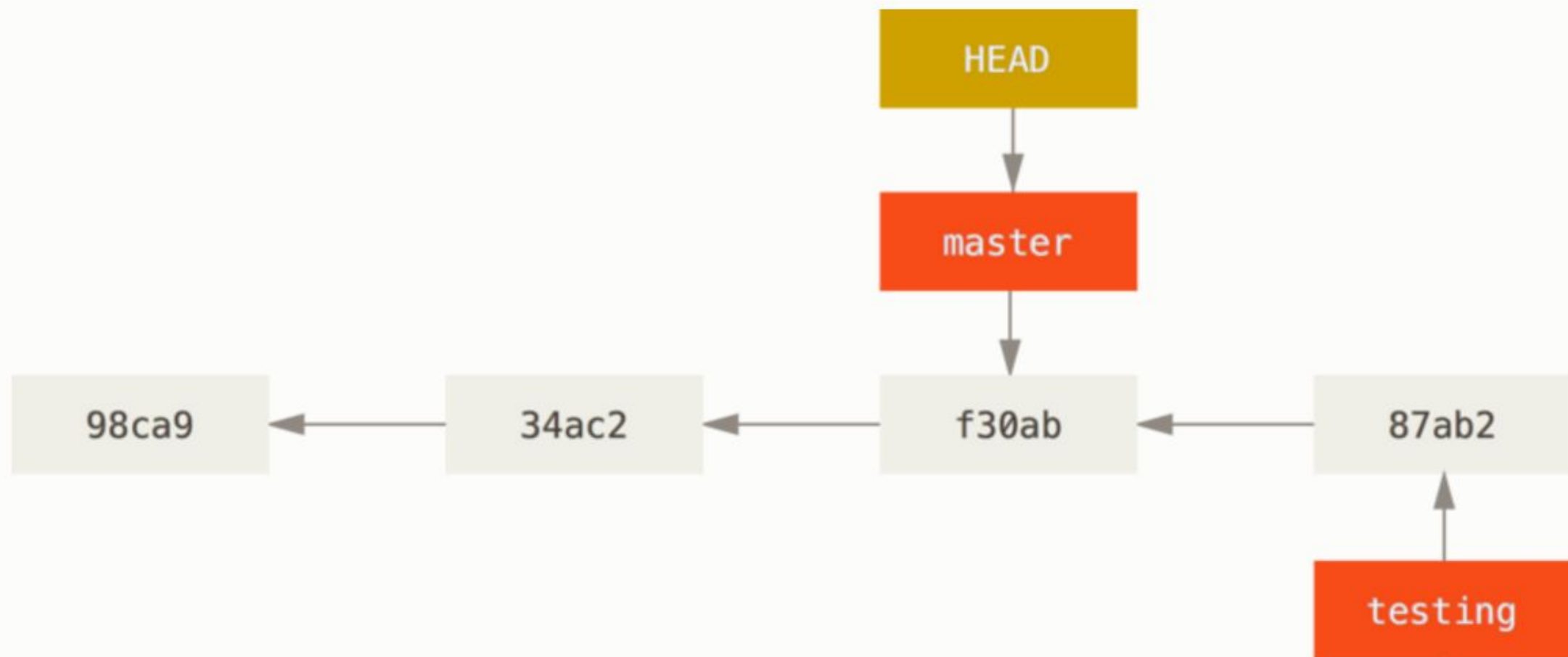
- Now let's do another commit...



HEAD REDUX (CONT'D)

- And finally, let's switch back to **master**

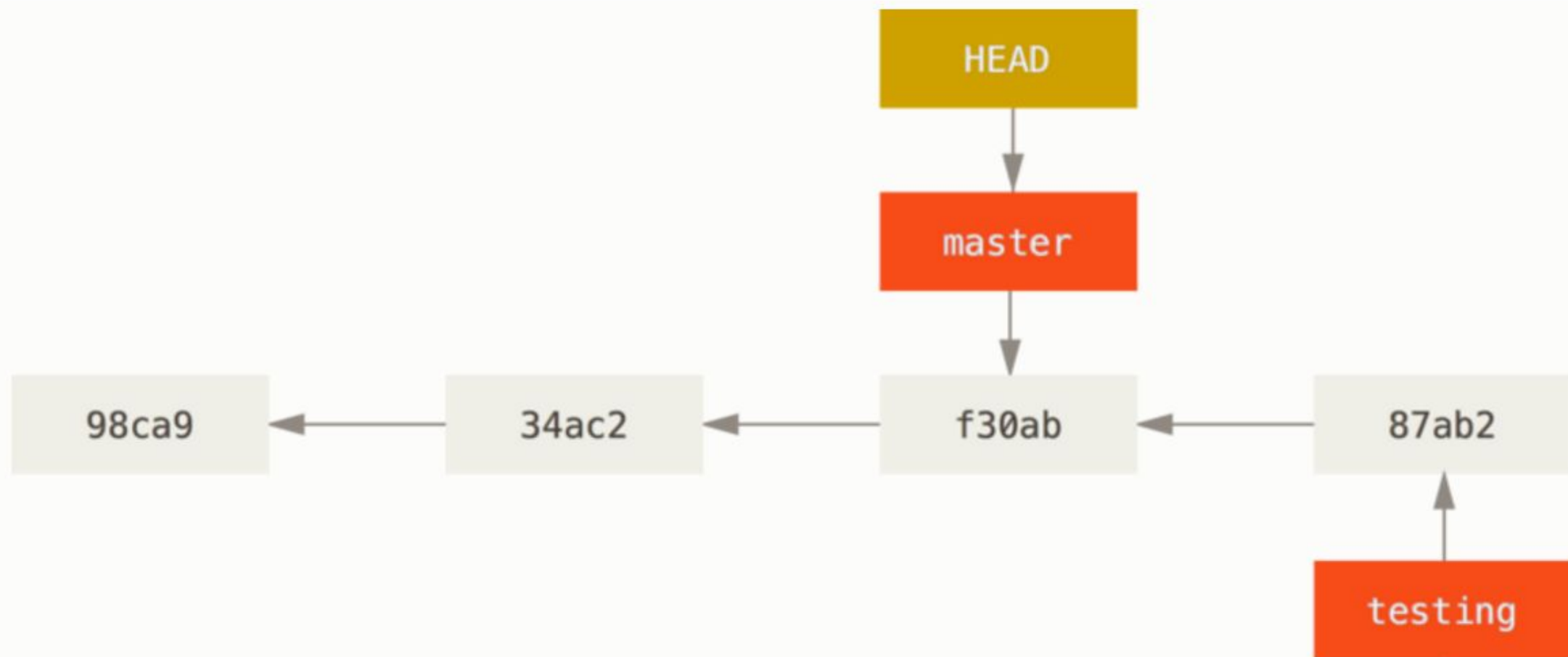
```
$ git checkout master
```



HEAD REDUX (CONT'D)

- And finally, let's switch back to **master**

```
$ git checkout master
```



HEAD REDUX (CONT'D)

- Let's see where we are in the `.git/objects` database
- The **refs** subdirectory contains references to the tip of each branch
- Let's use the Git visualizer tool:
<https://git-school.github.io/visualizing-git/>
- Another tool for learning branching (and merging):
https://learngitbranching.js.org/?locale=en_US

LAB: BRANCHING

- See what branches you have so far (probably none)
- Create two branches from **main** called **testing** and **hotfix**
- Use **git log --decorate --oneline** to verify that **main** and both branches point to the same commit
- Switch to **testing** branch and add a file, stage, and commit it
- Switch to **hotfix** branch and add a file, stage, and commit it
- Use **git log --graph --oneline** along the way to visualize the branches and commits

MERGING



MERGING ("INTO")

- Merging is the how we get the changes on one branch into another branch
- First, we check out the branch that is to receive the changes

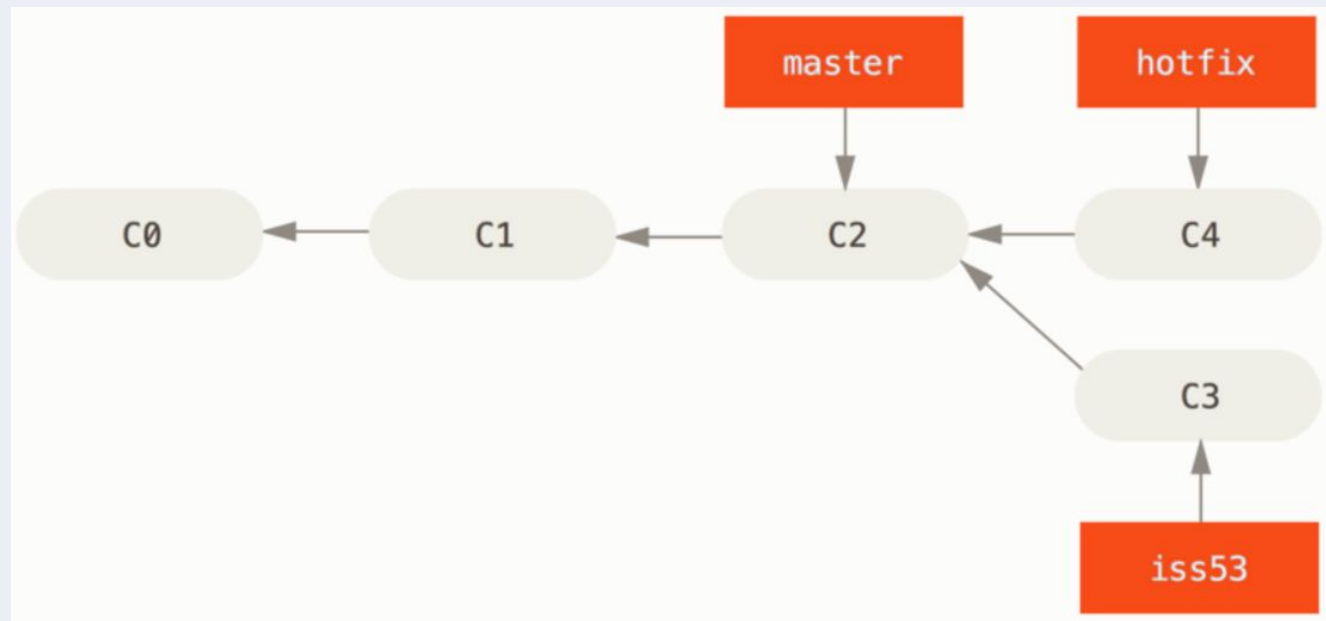
git checkout <target branch>

- Then, we merge the branch that contains the changes

git merge <source branch>

- There are two types of merges...

"FAST FORWARD" MERGE

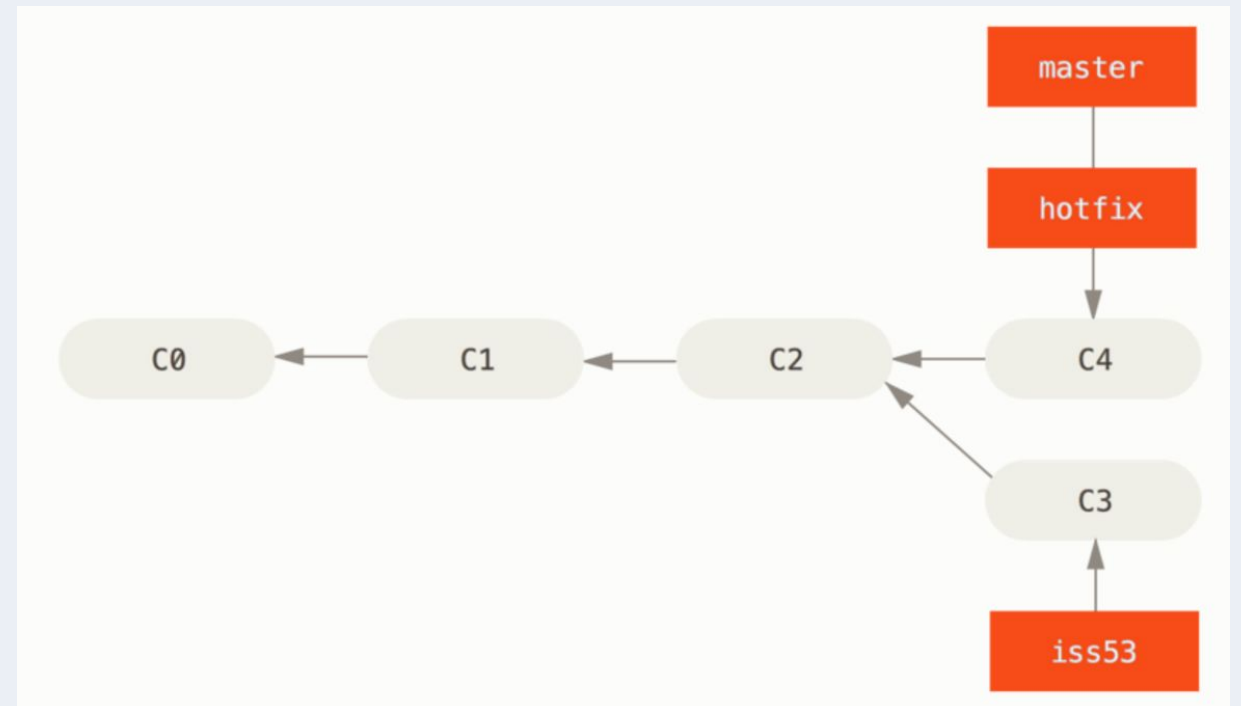
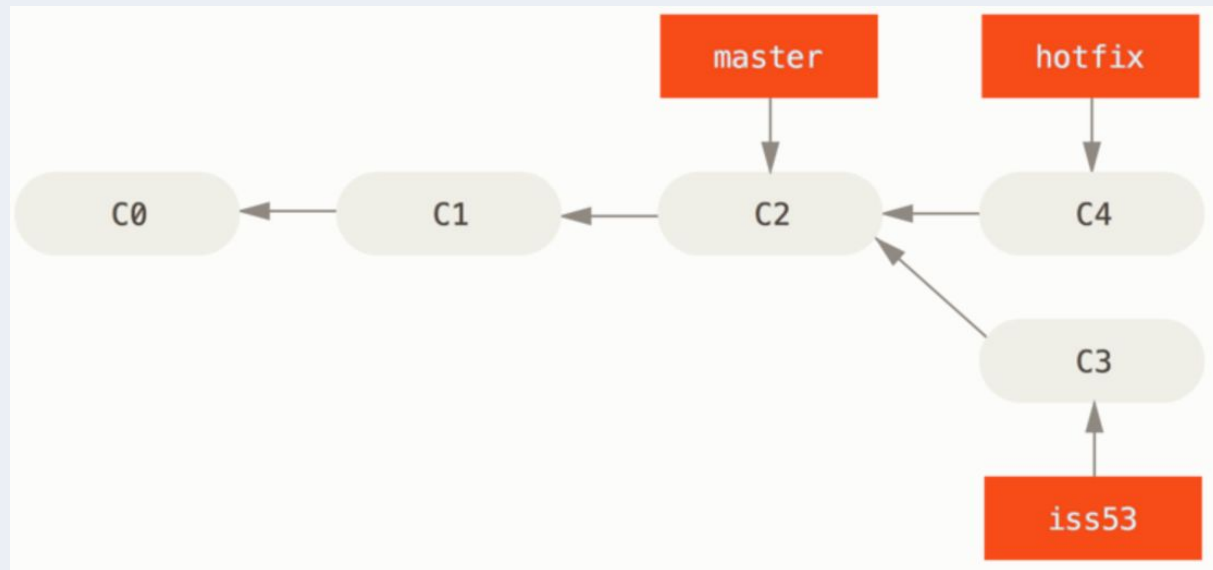


Consider the hotfix branch whose commit C4 fixes an issue in the master branch.

- When you merge one commit (C4, in this case) with a commit that can be reached by following the first commit's history (C2), git simplifies things by moving the pointer forward
- This is called a fast-forward merge

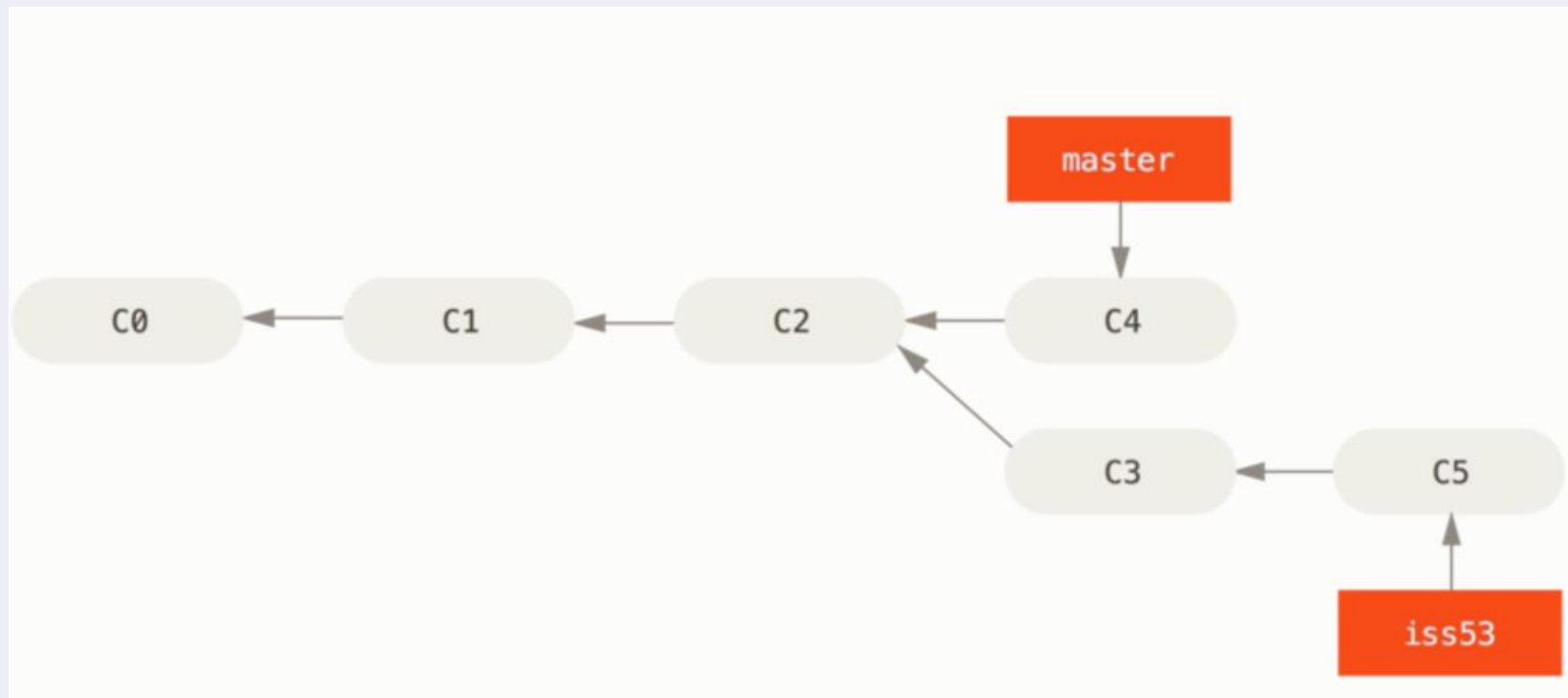
```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```

"FAST FORWARD" MERGE (CONT'D)



- A fast-forward merge results in a streamlined history without an explicit merge commit

"THREE WAY" MERGE

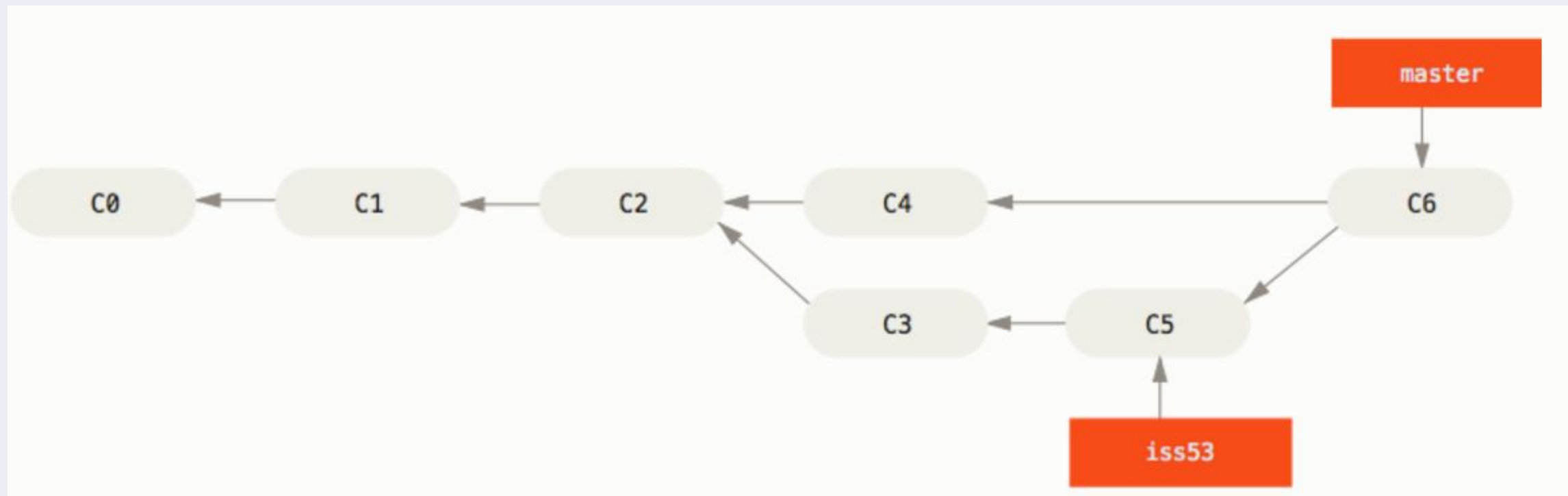
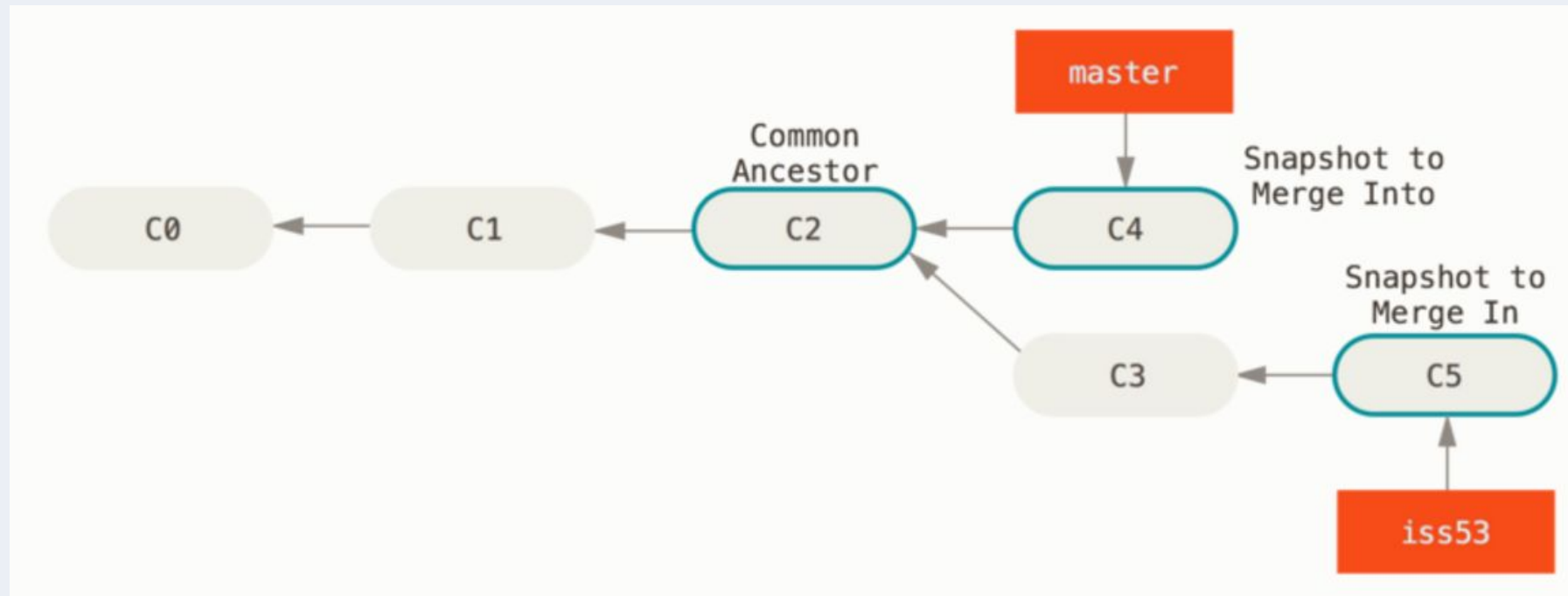


Look at **iss53** branch...C5 doesn't include changes from previous **hotfix** branch (C4)

- Git creates a new snapshot from this three-way merge and creates a new commit that points to it
- This is referred to as a merge commit, and is special in that it has more than one parent

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

"THREE WAY" MERGE (CONT'D)



BRANCH MANAGEMENT

- Which branches have been merged?

`git branch --merged`

- Which branches haven't been merged?

`git branch --no-merged`

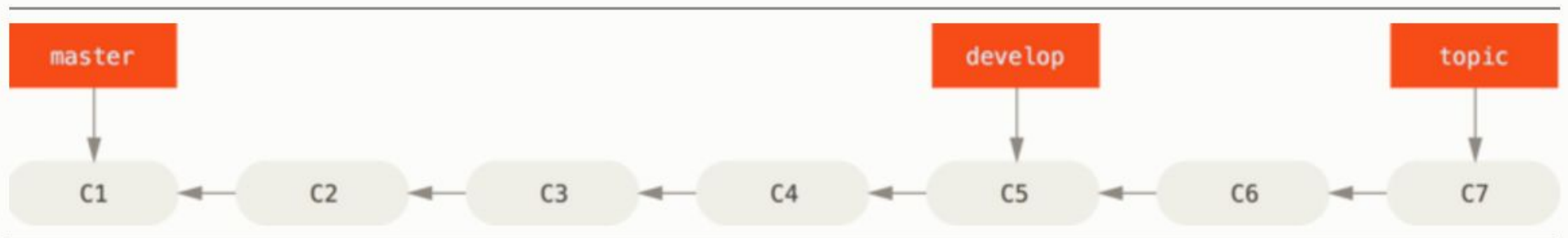
- Remove branches after they've been merged

`git branch -d <name>`

- What if you try to remove a branch you haven't merged?
 - use `-D` to force

TYPICAL BRANCHING WORKFLOW

- The **main/master** branch is typically stable code, so we want to avoid working in that branch
- For a bug fix, create a new branch from **main/master**
- Make fix(es) in your branch
- Merge back to **main/master** once your fixes have been tested!



RECAP: BRANCHING/MERGING

- **git branch** = viewed, created, or deleted branches
- **git switch/checkout** = switch between branches, tags, or commits
- **git merge** = apply changes from one branch "into" another

LAB: MERGING

- On **main**, merge the **hotfix** branch
 - It should have been a fast-forward merge
 - Check the log
 - Check which branches have been merged (and which haven't)
- Merge the **testing** branch
 - Review your log as before
- Create a new branch and switch to it
 - Add a file containing “Git is fun” and commit it

LAB: MERGING (CONT'D)

- On **main**, merge your new branch as a non-fast-forward merge using the flag **--no-ff**
- Review the log as before—notice the difference between the merges?
- Delete your branches

A photograph of two men standing on a cobblestone street, facing each other and shouting into yellow megaphones. The man on the left is wearing a white long-sleeved shirt, blue jeans, and green sneakers. The man on the right is wearing a light blue button-down shirt, dark trousers, and a red tie. The background is a textured, grey wall. In the center of the image, the words "CONFLICT (AVOIDANCE)" are written in a large, bold, black, hand-drawn font. The overall scene suggests a confrontation or a loud argument, which is the subject of the text overlay.

CONFLICT (AVOIDANCE)

MERGE CONFLICTS

- Occasionally, merging doesn't go smoothly and you get conflicts
 - for example, if the same part of a file was changed in both **main** and your branch
 - When a conflict occurs
 - the merge is not committed
 - you must resolve the conflict or...abort the merge
- git merge --abort**

MERGE CONFLICT (CONT'D)

- Git adds conflict-resolution markers to the conflicted files:

```
<<<<<<<< HEAD
```

```
    foo
```

```
=====
```

```
    bar
```

```
>>>>>>>> branch
```

CONFLICT RESOLUTION

- To resolve conflicts
 - Check for conflicted files with **git status**
 - Edit the conflicted files and fix the conflicts by hand
 - **git add** each resolved file
 - **git commit** to finish merge once all conflicts are resolved
- Or use **git mergetool**
 - config var **merge.tool** must be set up, or have a suitable default

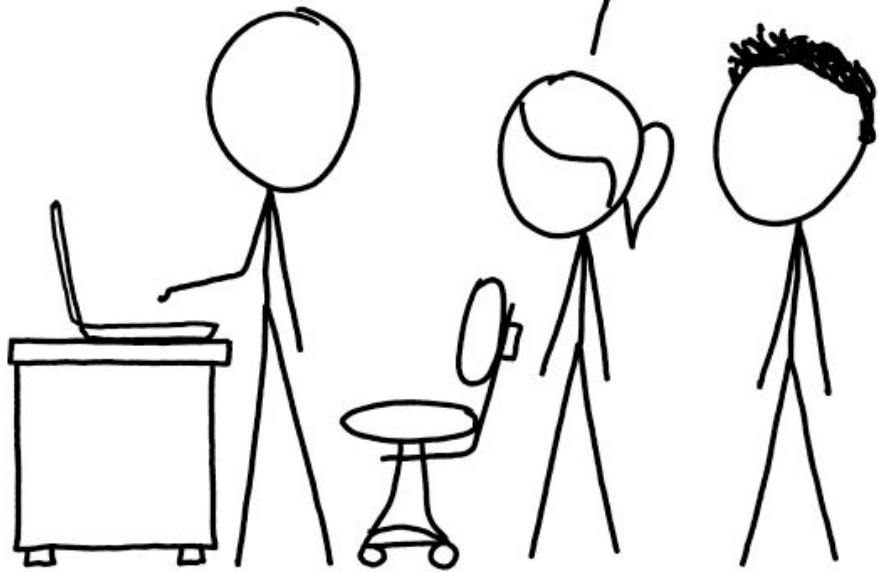
LAB: CONFLICT RESOLUTION

- Let's create a conflict which we resolve
- Create two branches off of **main**
 - In first branch, create a file called **conflict.txt** and add one line of text
 - In second branch, create the same file and add a different line of text
- On **main**, merge the first branch
- Then merge the second branch—you should get a conflict
- Abort the merge, then merge second branch again
- Resolve the conflict

THIS IS GIT. IT TRACKS COLLABORATIVE WORK
ON PROJECTS THROUGH A BEAUTIFUL
DISTRIBUTED GRAPH THEORY TREE MODEL.

COOL. HOW DO WE USE IT?

NO IDEA. JUST MEMORIZE THESE SHELL
COMMANDS AND TYPE THEM TO SYNC UP.
IF YOU GET ERRORS, SAVE YOUR WORK
ELSEWHERE, DELETE THE PROJECT,
AND DOWNLOAD A FRESH COPY.



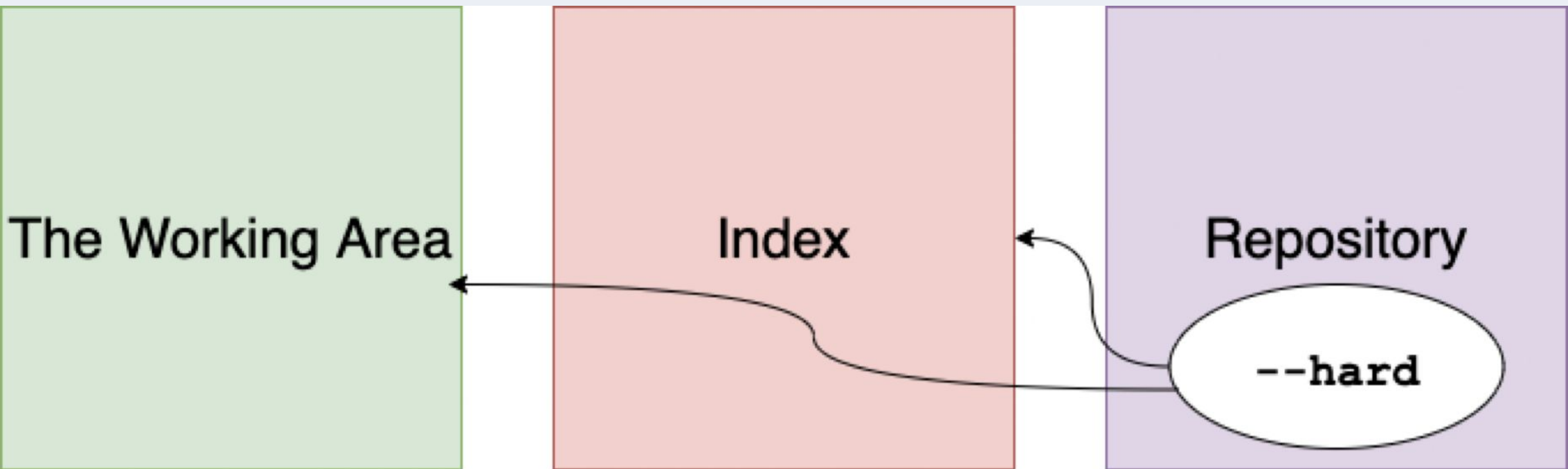
**GIT RESET:
SCALPEL OR
SLEDGE-
HAMMER**

GIT RESET

- this command does TWO things, each of which is optional (that is, it can be used to either of these things or both of them)
 - moves a branch reference to point at a specific commit
 - copies data from that specific commit to the other areas, depending on options

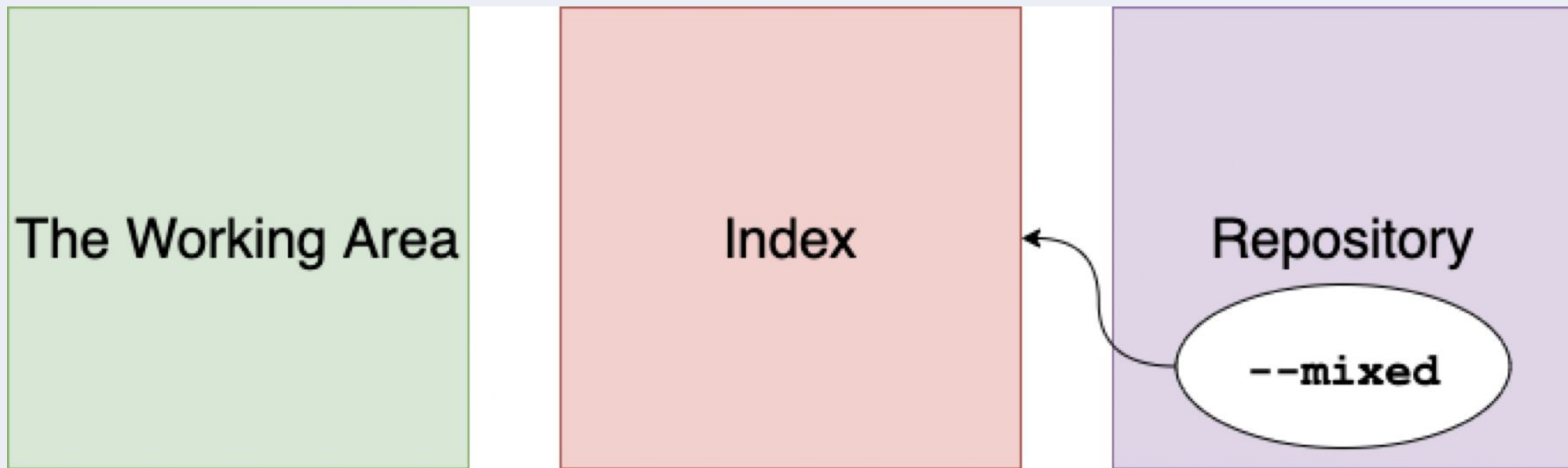
GIT RESET --HARD

- make index and working area the same as the commit



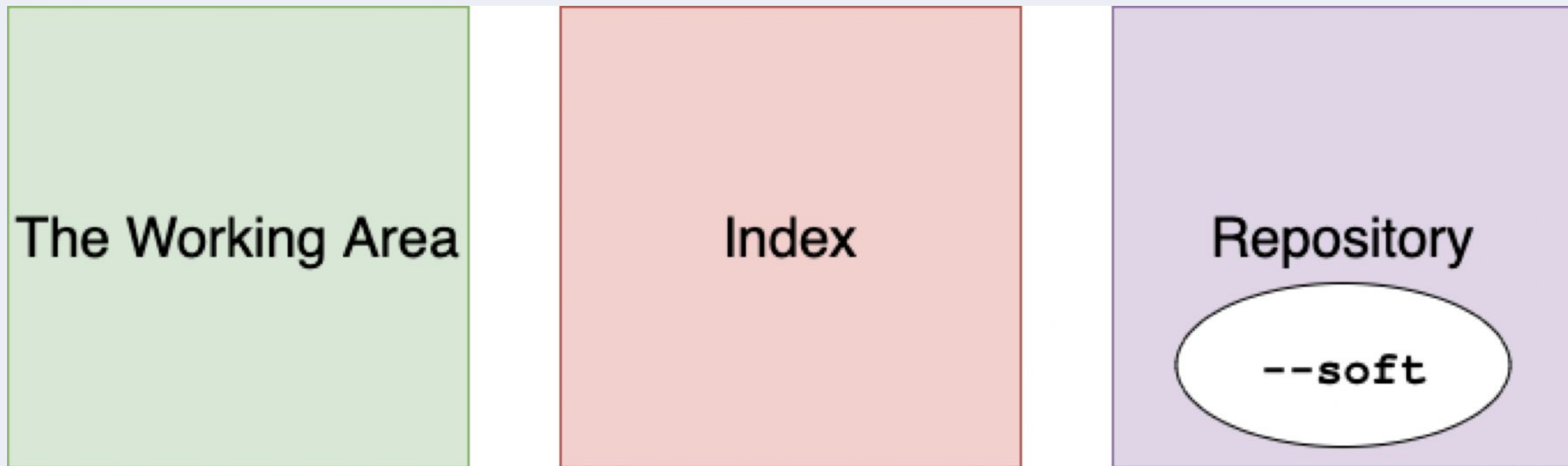
GIT RESET --MIXED (DEFAULT)

- make the index the same as the commit the branch was moved to, but preserve working area changes



GIT RESET --SOFT

- just move the branch but don't touch any of the areas



USING RESET TO REMOVE LAST COMMIT

```
git reset --hard HEAD^
```

- moves the **main** branch to point at the previous commit, thereby bypassing the previous commit
- **--hard** makes it so the working dir and index match that commit
- note the ^ character, which refers to the commit before the current **HEAD** (we could have used **HEAD~1** or the hash)
- note that this completely removes a commit
 - it's actually still in your object database, but it's dangling and eventually it will be garbage collected

USING RESET TO UN-STAGE CHANGES

```
git reset HEAD [file]
```

- Git moves the current branch to the commit pointed to by **HEAD**, in other words it doesn't move at all
- this is a mixed reset (the default), which moves data from the repo to the index, but NOT to the working dir

USING RESET TO DISCARD LOCAL CHANGES

```
git reset --hard HEAD [file]
```

- Git moves the current branch to the commit pointed to by **HEAD**, in other words it doesn't move at all
- as we saw, this moves data from the repo into BOTH the index and the working dir, discarding any local changes
- uncommitted/unstaged changes will be lost!

USING RESET TO DELETE AN ERRANT MERGE

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

- this command will discard any changes to working dir, but if you want to retain local changes, use...

```
git reset --merge ORIG_HEAD
```

- This command rewrites history, and we should only do this with a local repo which has not been shared

THIS IS GIT. IT TRACKS COLLABORATIVE WORK
ON PROJECTS THROUGH A BEAUTIFUL
DISTRIBUTED GRAPH THEORY TREE MODEL.

COOL. HOW DO WE USE IT?

NO IDEA. JUST MEMORIZE THESE SHELL
COMMANDS AND TYPE THEM TO SYNC UP.
IF YOU GET ERRORS, SAVE YOUR WORK
ELSEWHERE, DELETE THE PROJECT,
AND DOWNLOAD A FRESH COPY.



**TAG,
YOU'RE IT!**

TAGS

- tags give us a way to mark a commit, much like a bookmark
- why do this?
 - releases
 - work-in-progress
 - work done for a specific customer...
 - ...any time you want to take note of where the code base is NOW, so you can revisit it later
- they are fourth and final objects in the Git database (i.e., blobs, commits, trees, and tags)

LIGHTWEIGHT VS. ANNOTATED TAGS

- Adding a "lightweight" tag

```
git tag <name> <optional commit>
```

- let's take a look in the `.git` folder to see why it's called a "lightweight" tag
- Adding an "annotated" tag

```
git tag -a <name> -m "Message"
```

- what's different here?

TAGGING COMMANDS

- List your tags: `git tag`
- List tags matching a pattern: `git tag -l <pattern>`
e.g., `git tag -l 'v1.8.5*'`
- View a specific tag: `git show <tag name>`
- Or check it out: `git checkout <tag name>`
 - `git switch --detach <tag name>`
 - ...puts you in detached head state (but shouldn't matter)
-

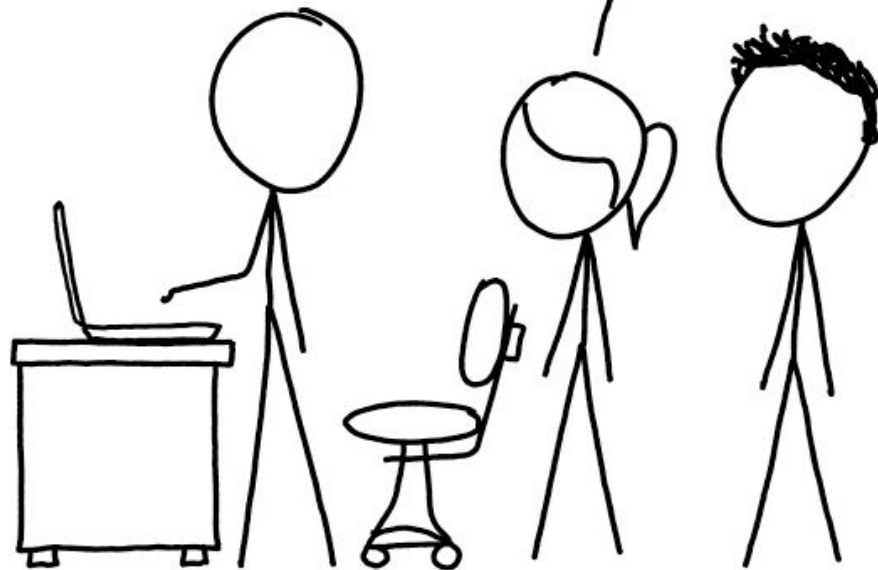
LAB: TAGS

- Tag your current commit
- List your tags
- Look at your tag using `git show`
- Tag some prior commit with an annotated tag
- List your tags
- Look at the tag you just made
- Check out the tag and verify that you're at the right revision, and you are in detached head state
- For fun, find the tags in your `.git` folder and be sure you understand the difference between a lightweight tag and an annotated tag

THIS IS GIT. IT TRACKS COLLABORATIVE WORK
ON PROJECTS THROUGH A BEAUTIFUL
DISTRIBUTED GRAPH THEORY TREE MODEL.

COOL. HOW DO WE USE IT?

NO IDEA. JUST MEMORIZE THESE SHELL
COMMANDS AND TYPE THEM TO SYNC UP.
IF YOU GET ERRORS, SAVE YOUR WORK
ELSEWHERE, DELETE THE PROJECT,
AND DOWNLOAD A FRESH COPY.



**WHERE'S
YOUR
STASH?**

STASHING

- What is stashing?
 - saves your work (staging area + working area)
 - resets the staging and working areas so they are the same as current repo, i.e., "clean state"
 - i.e., allows you to quickly set aside your work in progress
- Why stash?
 - to quickly store work you want to revisit
 - interruptions!

STASHING (CONT'D)

`git stash` (stash modified/staged files)

`git stash [-m message]`

`git stash list` (see what's stashed)

`git stash apply <name>` (apply, but leave in stash)

`git stash drop <name>` (delete without applying)

`git stash pop <name>` (apply and delete)

`git stash --keep-index` (keep staged stuff)

`git stash --all` (include untracked files)

`git stash apply --index` (re-stage stashed files)

`git stash clear` (remove everything!)

"CREATIVE" STASHING

- Suppose you've made a number of changes but only want to commit some of them
 - use **git stash --keep-index** to stash the files you haven't staged, then **git commit** only the staged files
- Suppose you want to choose what gets stashed and what doesn't
 - **git stash --patch** will interactively prompt you for which of the changes you want to stash and which you want to keep in your working directory

"CREATIVE" STASHING (CONT'D)

- if you stash work, leave it for a while, and continue working, you may have a problem reapplying the work you stashed
- you can create a new branch from the stashed work

git stash branch testchanges

- creates a new branch for you, checks out the commit you were on when you stashed, reapplies your work, then drops the stash if it applies successfully

STASHING VS. CLEANING

- `git clean` will remove untracked files, but be careful when doing this!
- `-d` can be used to remove any directories that became empty after removing untracked files, i.e.,
`git clean -d -f`
- `-n` will do a "dry run"
- a safer solution is `git stash --all`, which will stash everything, resulting in a clean working dir while still retaining a backup

LAB: STASHING PART 1

- Add an untracked file to your local dir
- Modify a tracked file in your repo
- Modify a second tracked file in your repo and stage it
- Check the status
- Stash everything except the staged file
- Commit
- Check that the status is clean
- Pop your stashed work
- Check the status
- Commit

LAB: STASHING PART 2

- Create a new branch **feature**
- In the branch, modify a file but don't stage or commit
- Imagine you get a request to fix an urgent bug
- Change a different file, but since you're not in the correct branch, stash it instead of committing
- Commit the original change, which does belong in this branch
- Now create a new branch **path-bug**
- In your branch, unstash the previously modified file and commit



**REMOTE
CONTROL**

GIT CLONE (REDUX)

- "cloning" means making a copy of an existing Git repo
- **git clone**
 - not a "checkout"—instead of getting just a working copy, cloning gives you a full copy of nearly all data the server has
 - every version of every file for the history of the project is copied into your repo by default
- Try cloning
<https://github.com/davewadestein/clone-me>
or something from your company server

REMOTES

- A remote is a local reference to other versions of the repo hosted elsewhere (i.e., a copy of the repo)
- when you clone a repo, Git automatically sets up a remote named **origin**
- to list your remotes

`git remote` (names only)

`git remote -v` (names + remote URLs)

`git ls-remote` (display all references in remote repo)

REMOTE-TRACKING BRANCHES

- references to state of remote branches
 - local references that you can't move
 - Git moves them when you do any network communication, ensuring they accurately represent the state of remote repo
 - like bookmarks, to remind you where branches in your remote repos were the last time you connected to them

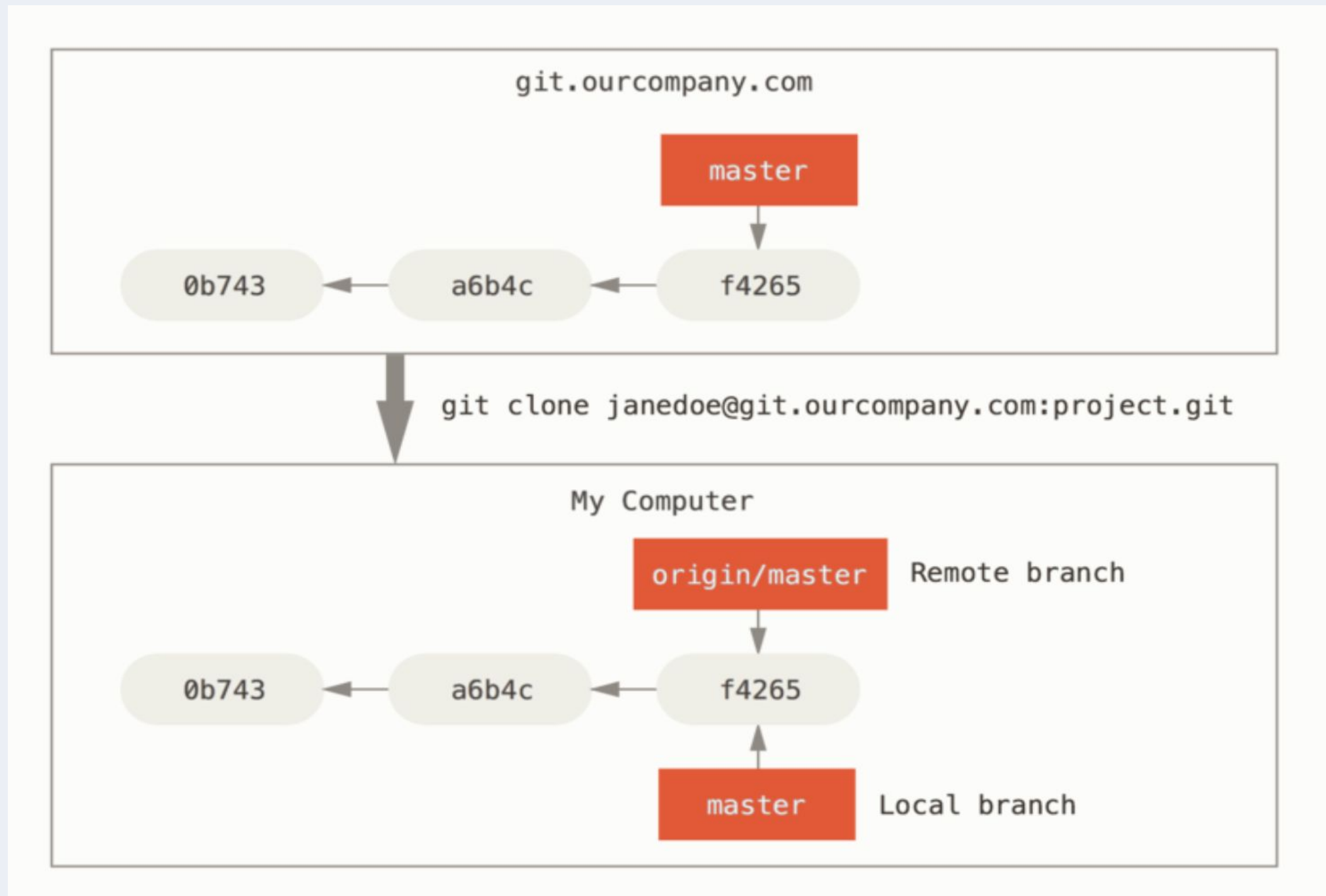
<remote>/<branch> e.g., **origin/master**

WHAT DOES GIT CLONE DO?

- create and initialize a local Git repo
- sets up an initial remote called **origin**
- sets up the initial remote tracking branch for **main**

REMOTE TRACKING BRANCHES (CONT'D)

- what happens after cloning?



SYNCHING W/REMOTE REPO

`git fetch` (origin is the default)

`git fetch <remote>` (e.g., `git fetch origin`)

- fetches data from remote that you don't yet have
- updates your local `.git` database, moving your `origin/master` pointer to its new up-to-date position

`git pull`

- fetch (`git fetch`) plus a merge (`git merge`)
- the "magic" of `git pull` can be confusing
 - we'll revisit this once we talk about rebasing...

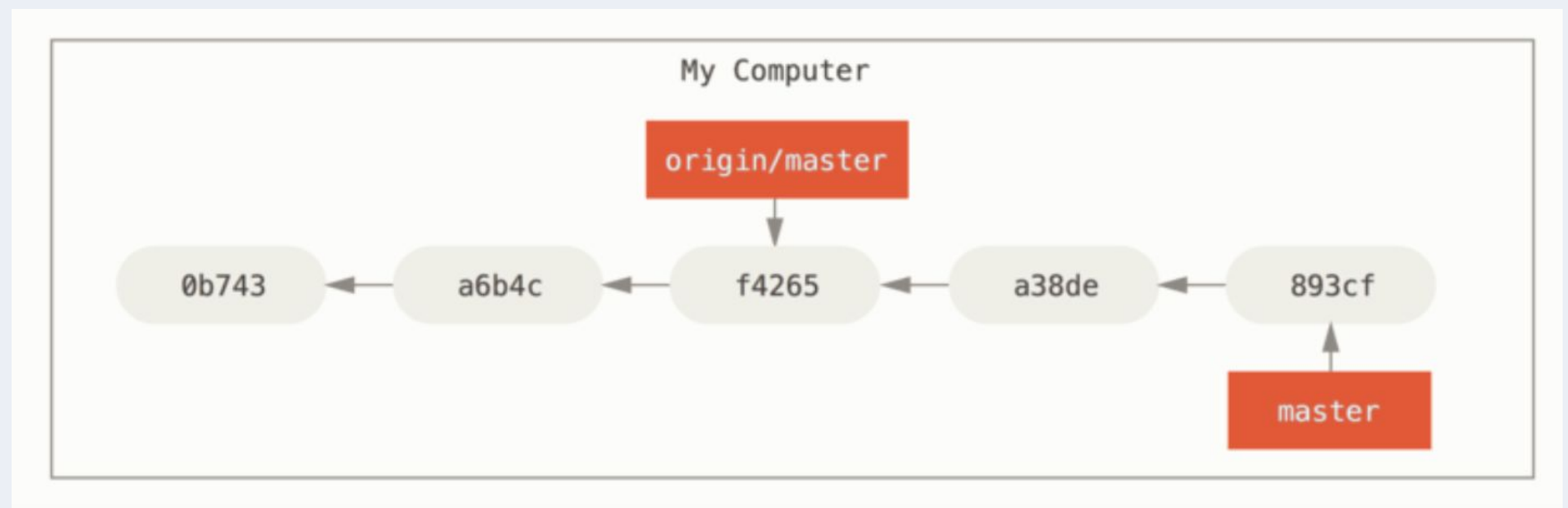
SYNCHING W/REMOTE REPO (CONT'D)

`git push`

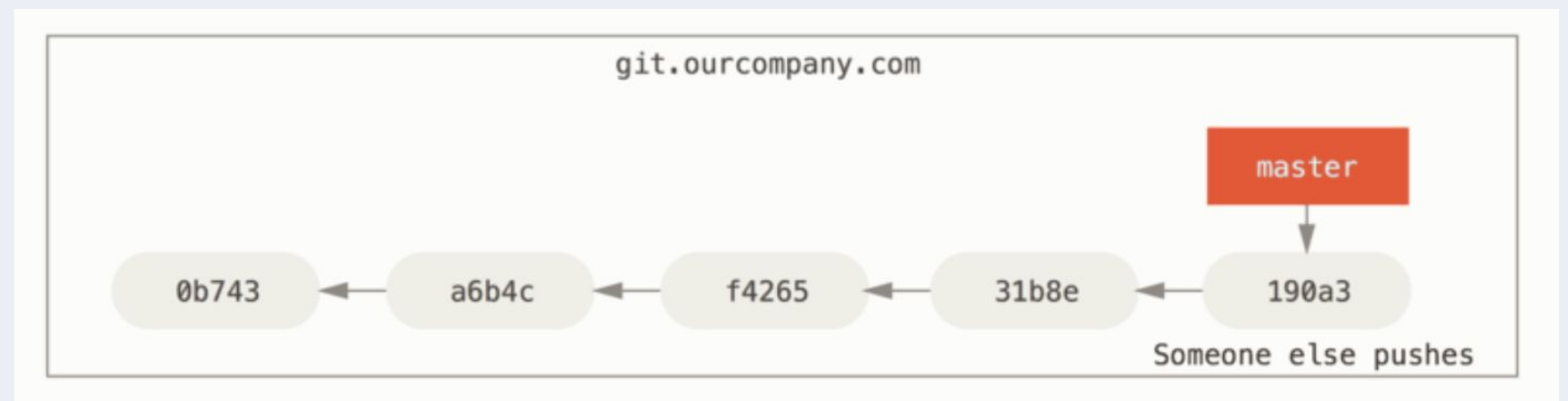
- share your work to the remote repo
- won't work someone else has pushed to the remote repo and it contains work that you don't have locally
- solution: use `git pull` to integrate those remote changes you don't have, then push again

REMOTE TRACKING BRANCHES (CONT'D)

- now you do some work locally...



- and someone else pushes to the repo...



- let's try it in the Git visualizer...

REMOTE TRACKING BRANCHES (CONT'D)

- to checkout a branch on a different remote, or to track a branch other than master

```
git checkout -b serverfix <remote>/serverfix
```

```
git checkout --track origin/serverfix
```

- so common that there's a shortcut for that shortcut
 - if branch 1) doesn't exist and 2) exactly matches a name on only one remote, Git will create a tracking branch for you

```
git checkout serverfix
```

- or if local branch already exists

```
git branch -u origin/serverfix
```

GIT BRANCH -v / -vv

- **-v**: show SHA1 and commit subject line for each head, along with relationship to upstream branch (if any)
- **-vv**: print the path of the linked worktree (if any) and the name of the upstream branch, as well

```
$ git branch -vv
iss53      7e424c3 [origin/iss53: ahead 2] forgot the brackets
master     1ae2a45 [origin/master] deploying index fix
* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] this should do it
testing    5ea463a trying something new
```

PUSHING TAGS

- the tags you create are local—to share them with your collaborators you must to **push** them to the remote

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

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

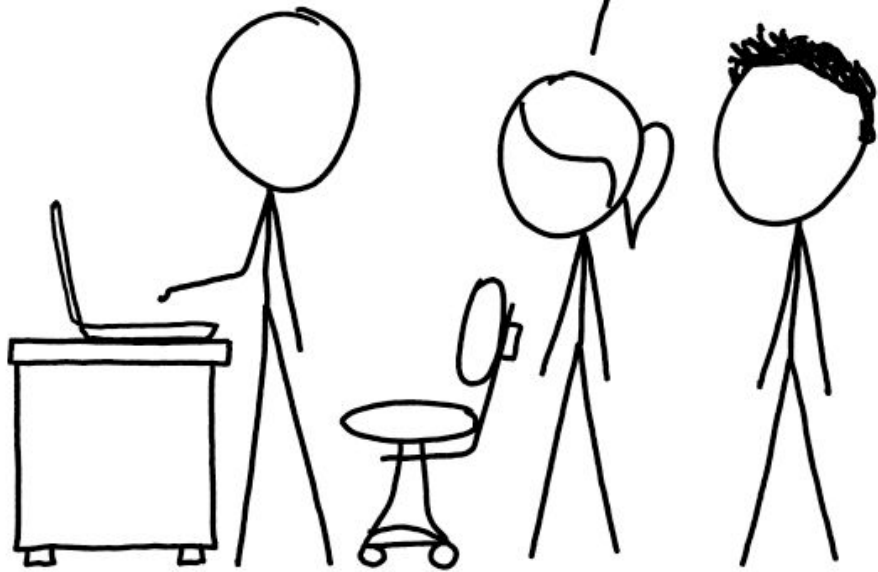

TYPICAL REMOTE WORKFLOW

- **main** is stable
 - topic branches, hotfixes, etc. branched off **main**
 - before branching from **main**, check for updates from **remote** and pull them down into **main**
 - branch from updated **main** and work in branch
- when done...
 - pull to update **main** from the remote
 - merge your branch to **main**
 - push your updated **main** to the remote

THIS IS GIT. IT TRACKS COLLABORATIVE WORK
ON PROJECTS THROUGH A BEAUTIFUL
DISTRIBUTED GRAPH THEORY TREE MODEL.

COOL. HOW DO WE USE IT?

NO IDEA. JUST MEMORIZE THESE SHELL
COMMANDS AND TYPE THEM TO SYNC UP.
IF YOU GET ERRORS, SAVE YOUR WORK
ELSEWHERE, DELETE THE PROJECT,
AND DOWNLOAD A FRESH COPY.



GIT REVERT: MEA CULPA

REVERT LAST COMMIT

- generates a new commit which reverses the operations in the last commit (not "undo", which is important when we revert a merge)
- does not remove the commit that was problematic (use reset for that)

`git revert HEAD`

- you can apply this command to an older commit, but of course there is always a chance it will cause a conflict
- use `--no-commit` to refrain from committing (good for seeing what will happen without committing)

REVERT-ING THE EFFECTS OF A MERGE

`git revert -m 1 HEAD`

- creates a new commit that reverses the merge (not an undo in that the merge commit remains, and therefore Git will still think the merge has been performed)
- the `-m 1` specifies which parent line you want to keep (1 means the left parent, usually **main**, i.e., you are discarding the branch merge and reverting back to **main**)
- If you later need to re-merge that branch, you'll need to revert the revert!
(<https://mirrors.edge.kernel.org/pub/software/scm/git/docs/howto/revert-a-faulty-merge.txt>)

THIS IS GIT. IT TRACKS COLLABORATIVE WORK
ON PROJECTS THROUGH A BEAUTIFUL
DISTRIBUTED GRAPH THEORY TREE MODEL.

COOL. HOW DO WE USE IT?

NO IDEA. JUST MEMORIZE THESE SHELL
COMMANDS AND TYPE THEM TO SYNC UP.
IF YOU GET ERRORS, SAVE YOUR WORK
ELSEWHERE, DELETE THE PROJECT,
AND DOWNLOAD A FRESH COPY.



**WE LEARN
FROM
HISTORY...**

MAINTAINING A GOOD HISTORY

- A "good" commit usually has four traits...**ACID**:
 - **A**tomic
 - **C**onsistent
 - **I**ncremental
 - **D**ocumented

ATOMIC

- a commit should represent one logical change
 - therefore commits should be relatively small
- semantically related changes should not be split across multiple commits (self-contained)
- all changes in a commit should be semantically related (coherent)

CONSISTENT

- each commit should leave the code in a consistent state, e.g., no compilation errors or failing tests
 - it should be possible to apply any commit to the working directory and build on them

INCREMENTAL

- our code evolves via a sequence of coherent modifications that build on each other incrementally
- order of our commits should be explanatory
- consider building a feature...
 - the order of commits should document the thought process involved in working on the code

DOCUMENTED

- the meaning of a change and the reasoning behind it should be conveyed in a good commit message
- a good commit message consists of two parts:
 - one-line summary (single sentence)
 - more detailed description (if needed)
- whatthecommit.com

THIS IS GIT. IT TRACKS COLLABORATIVE WORK
ON PROJECTS THROUGH A BEAUTIFUL
DISTRIBUTED GRAPH THEORY TREE MODEL.

COOL. HOW DO WE USE IT?

NO IDEA. JUST MEMORIZE THESE SHELL
COMMANDS AND TYPE THEM TO SYNC UP.
IF YOU GET ERRORS, SAVE YOUR WORK
ELSEWHERE, DELETE THE PROJECT,
AND DOWNLOAD A FRESH COPY.



ALTERING HISTORY

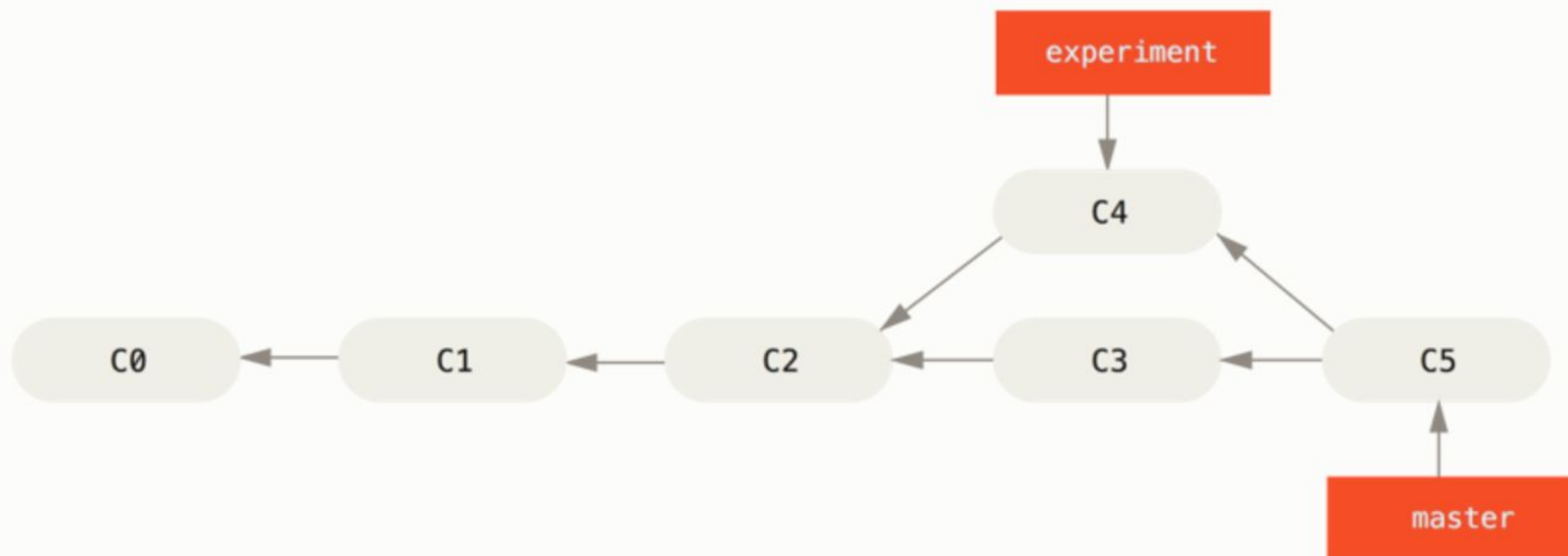
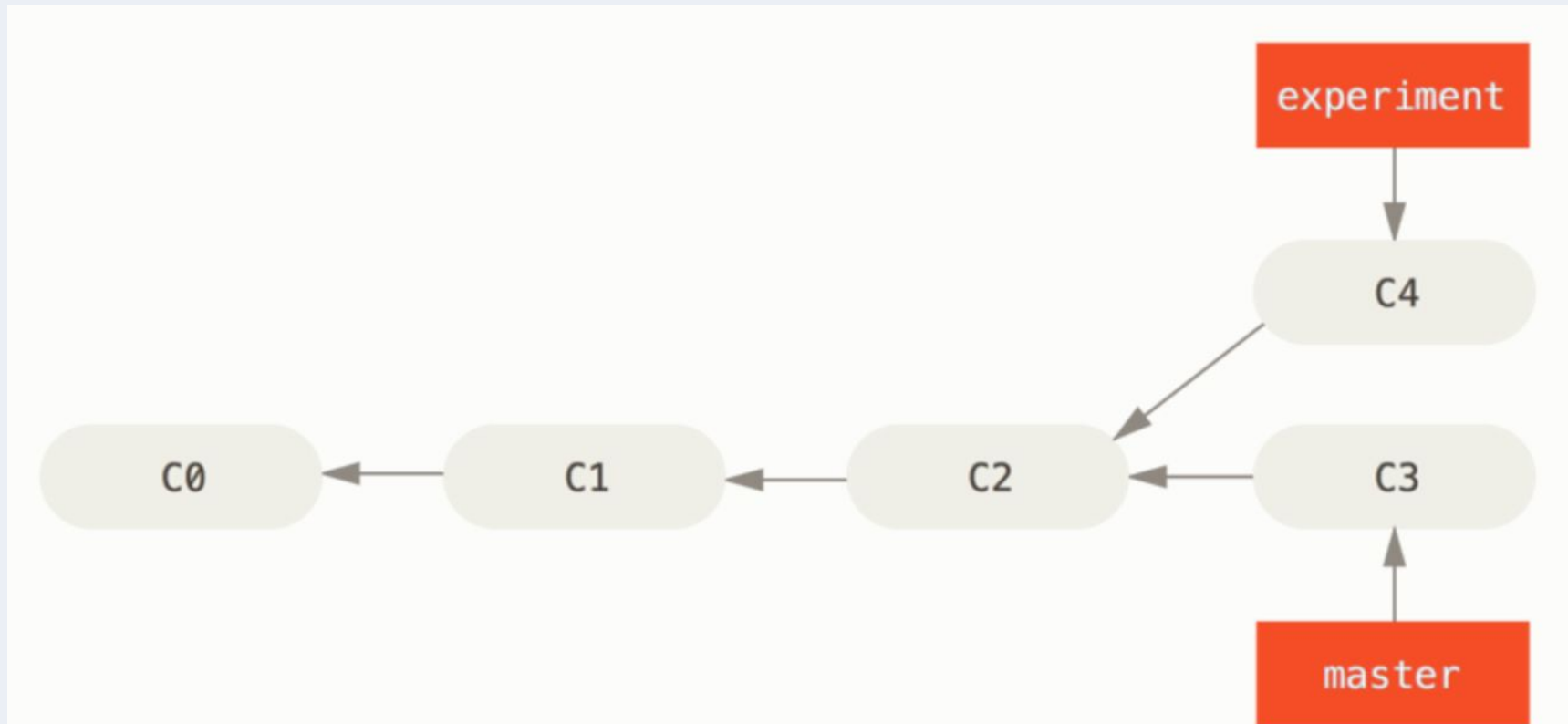
ALTERING HISTORY

- we previously saw examples of this process with **git reset**
- now we'll look at
 - rebasing
 - cherry picking

REBASE ("ONTO")

- rebasing is an alternative to merging which results in a cleaner history...by changing it
- the key idea is that we essentially cut off a branch and graft (or "re-base") it onto a different branch
- first, let's review merging...

MERGING



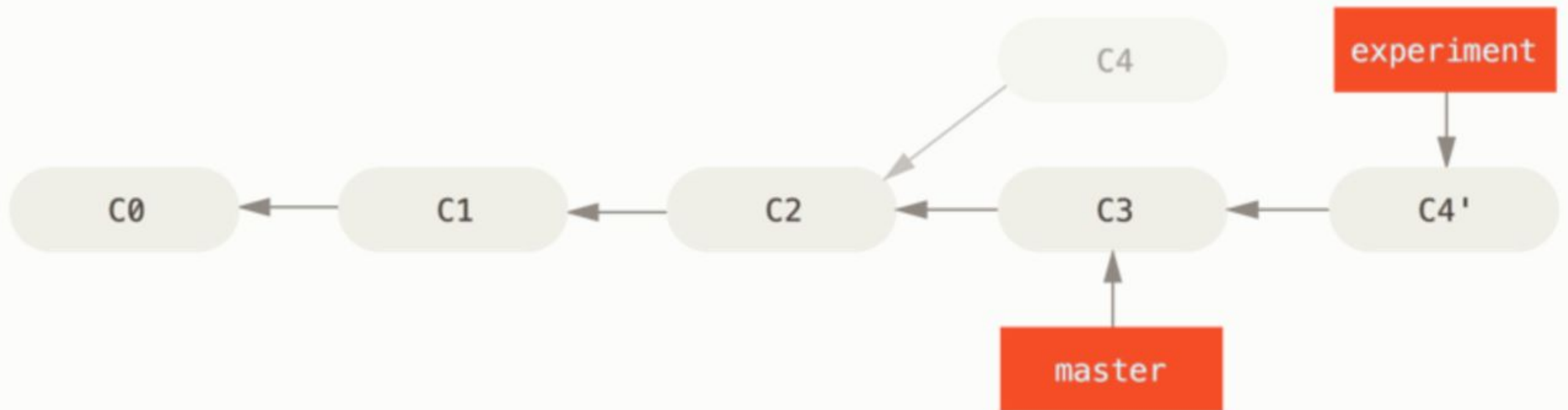
REBASE (CONT'D)

```
$ git checkout experiment
```

```
$ git rebase master
```

First, rewinding head to replay your work on top of it...

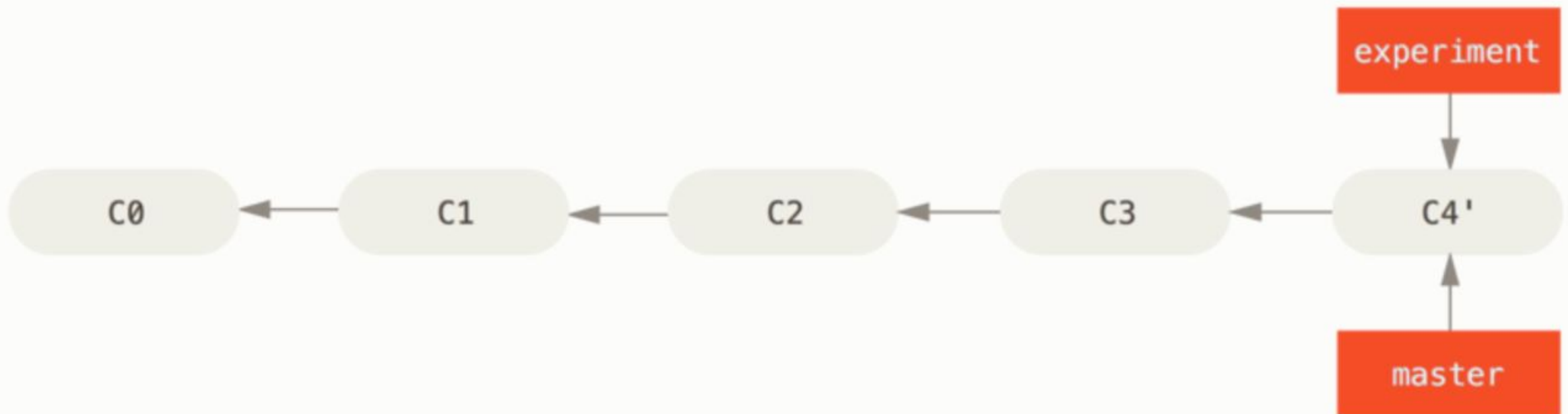
Applying: added staged command



REBASE (CONT'D)

- now we go back to master and do a fast-forward merge

```
$ git checkout master  
$ git merge experiment
```



REBASE TO KEEP YOUR BRANCH CURRENT

- When pulling changes from a remote branch they will be merged into your branch by default
 - do we really need a merge commit to saying that we kept our local branch current w/the remote?
 - instead, use **git pull --rebase**
- Here's a scenario that happens frequently when collaborating...
 - Alice creates topic branch A and works on it
 - Bob creates unrelated topic branch B, and works on it
 - Alice does **git checkout main && git pull**
 - **main** is already up to date.
 - Bob does **git checkout main && git pull**
 - **main** is already up to date.
 - Alice does **git merge topic-branch-A**

REBASE TO KEEP YOUR BRANCH CURRENT (CONT'D)

- Bob does `git merge topic-branch-B`
- Bob does `git push origin main` before Alice
- Alice does `git push origin main`, which is rejected because it's not a fast-forward merge.
- Alice looks at the `origin/main` log, and sees that the commit is unrelated to hers.
- Alice does `git pull --rebase origin main`
- Alice's merge commit is unwound, Bob's commit is pulled, and Alice's commit is applied after Bob's commit
- Alice does `git push origin main`, and everyone is happy they don't have to read a useless merge commit when they look at the logs in the future

REBASE TO KEEP YOUR BRANCH CURRENT (CONT'D)

"You and others are hacking away at the intended linear history of a branch. The fact that someone else happened to push slightly prior to your attempted push is irrelevant, and it seems counterproductive for each such accident of timing to result in merges in the history."

<https://stackoverflow.com/a/4675513/51048>

- You can configure to always rebase your pulls, like so
`git config --global pull.rebase preserve`

REBASE TO SQUASH MULTIPLE COMMITS

- With an interactive rebase you can squash and modify commits

git rebase -i <commit>

- lets you interactively edit commits from the specified commit (excluded) onward...
 - reword commit messages
 - squash commits
 - drop commits altogether

LAB: INTERACTIVE REBASE

- make an "errant" commit that will be OK to drop later (e.g., add a bogus file, make a bad change, etc.)
- make a good commit with a bad log message
- make several WIP commits
- perform an interactive rebase to
 - squash those commits
 - change one or more commit messages
 - drop the errant commit

GIT MERGE TO SQUASH

- Note that merge also has a squash option

```
git merge <branch> --squash
```

```
git commit
```

- Will squash all commits on other branch into one and stage it on the current branch

LAB: MERGE TO SQUASH

- create a "topic" branch with a few commits

git merge <branch> --squash

- examine log to verify all branch commits were squashed

REBASE CONFLICTS

- Similar to merge conflicts
- Either resolve the conflict and continue...

```
git rebase --continue
```

- ...or just abort

```
git rebase --abort
```


SQUASHING W/O REBASING

- suppose you have a topic branch into which you've been merging upstream changes to **main** (as opposed to be rebasing)–so the history is littered with merges you'd prefer not to be there
- it could be squashed upstream upon merging, but you may not be in control of that, so instead, you can create a temporary branch and squash into that:

```
git switch -c temp main (make new branch from main)
git merge --squash topic (merge messy topic into temp)
git commit
git switch topic
git reset --hard temp (nice use of reset)
git push --force (if already on remote)
```

<https://blog.oddbit.com/post/2019-06-17-avoid-rebase-hell-squashing-wi/>

MERGE VS. REBASE

- safe vs. unsafe
- non-destructive vs. destructive
- easy vs. difficult to understand(?)
- easy vs. difficult to revert
- ugly merge commits vs. linear history
- difficult vs. easy to view log
- keep vs. discard branching activity

MERGE GENERAL RULES

- When merging a public branch into another public branch, prefer recursive merges
- When merging a private branch into a public branch, prefer fast-forward merges
 - rebase your work onto public branch before merging
 - other people likely don't care that the work was done in a topic branch that only exists in your local repo

CHERRY PICKING

- Select one or more commits and rebase it/them onto current branch

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

- Conflicts are possible, so just as with merge and rebase, you can

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

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

CHERRY PICKING (CONT'D)

- Can also cherry pick a range

```
git cherry-pick ref1..ref2
```

- Not a merge because we've simply tacked one or more commit on to the end of our current branch
 - first commit in the range is excluded

```
git cherry-pick $(git merge-base master  
feature)..feature
```

- allows us to cherry pick all commits on a branch by starting from the first common ancestor of **master** and **feature**

LAB: CHERRY PICKING I

- in a branch, commit a new file, e.g, **newfile**
- add a line to the file and commit again
- make one or more additional commits
- check out **master** and cherry pick the first commit (the one that added the file)
- use **git reset** to discard the cherry pick
- cherry pick the second commit
- abort the cherry pick
- cherry pick both commits

LAB: CHERRY PICKING 2

- create a branch and add 3 commits to it
- switch to **main** and cherry pick all 3 commits from the branch
- how is this different from a rebase?

RECAP: ALTERING HISTORY

- Rebase keeps your history clean by avoiding merge commits, and squashing messy work
 - ...but merge when necessary
- Cherry picking is good for grabbing a few commits here and there
- DO NOT alter shared commits!

FROM THE HORSE'S MOUTH...

People can (and probably should) rebase their own work. That's a *cleanup*. But never other people's code. That's a "destroy history."

–Linus Torvalds

THIS IS GIT. IT TRACKS COLLABORATIVE WORK
ON PROJECTS THROUGH A BEAUTIFUL
DISTRIBUTED GRAPH THEORY TREE MODEL.

COOL. HOW DO WE USE IT?

NO IDEA. JUST MEMORIZE THESE SHELL
COMMANDS AND TYPE THEM TO SYNC UP.
IF YOU GET ERRORS, SAVE YOUR WORK
ELSEWHERE, DELETE THE PROJECT,
AND DOWNLOAD A FRESH COPY.



ALIASES

ALIASES

- aliases allow you to set up your own command shortcuts
- let's set some up as an interactive lab

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

```
git config --global alias.br branch
```

```
git config --global alias.ci commit
```

```
git config --global alias.unmerge "reset --hard ORIG_HEAD"
```

```
git config --global alias.graph "log --oneline --graph  
--decorate"
```

```
git config --global alias.llg "log --color --graph  
--pretty=format: '%C(bold white)%h %d%Creset%n%s%n%+b%C(bold  
blue)%an <%ae>%Creset %C(bold green)%cr (%ci)'"
```

- Can reference an external command with “!” prefix

```
git config --global alias.acm '!git add . && git commit -m'  
# add everything and commit with message...
```



**GIT
RESET
REDUX**

ADDITIONAL USES OF GIT RESET

- undo a commit, making into a new topic branch

```
git branch newtopic
```

```
git reset --hard HEAD^
```

```
git checkout newtopic
```

- in other words...
 - create a branch from the current commit
 - discard current commit
 - switch to branch and keep working

ADDITIONAL USES OF GIT RESET (CONT'D)

- you can safely pull/merge into a "dirty" working area if you know that the pull/merge will not affect the files you've modified...

```
git pull
```

Auto-merging...

Merge made by recursive.

...

- if you decide you want to undo

```
git reset --merge ORIG_HEAD
```

- note the **--merge**, which keeps your local changes by backing out the merge changes (whereas **--mixed** would leave working area unchanged)

ADDITIONAL USES OF GIT RESET (CONT'D)

- suppose you want to squash two WIP commits

```
git reset --soft HEAD~3
```

- rewind 3 commits (**HEAD~3** = 3 commits before current **HEAD**)
- ...but leave index and working area looking like the current commit
- then just commit, since working area reflects current state

ADDITIONAL USES OF GIT RESET (CONT'D)

- suppose you want to split the last commit into two commits

git reset --soft HEAD^

- rewind one commit, discarding latest commit
- ...but leave index and working area looking like the current commit
- use **git reset --patch** to unstage stuff which doesn't belong!
- commit

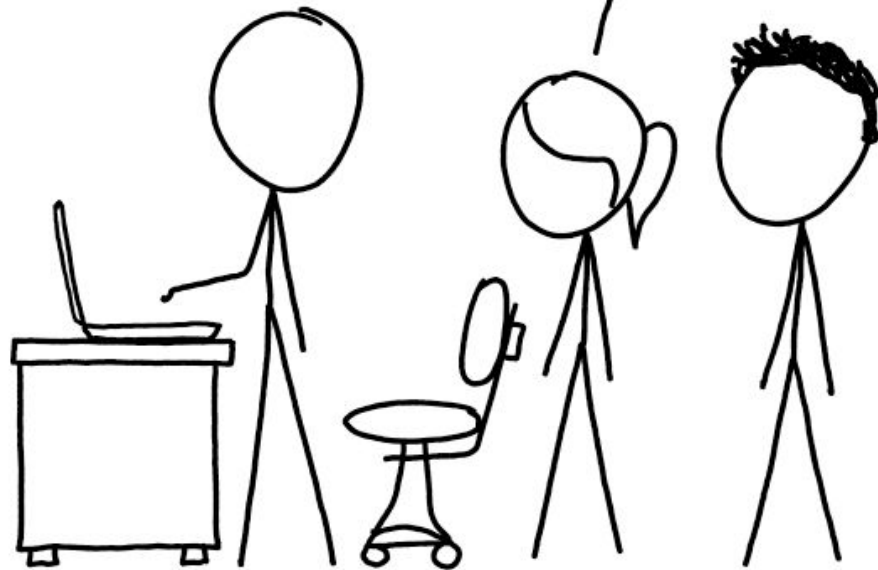
LAB: ADDITIONAL USES OF GIT RESET

- commit to master
- create a topic branch
- rewind master to discard your errant commit
- switch to branch and "keep working"
- use reset to squash some WIP commits (cf. interactive)
- make two unrelated changes and commit as one
- use git reset to rewind one commit, leaving index changes intact
- use git reset -p to unstage one of the changes
- re-commit, this time with only one change

THIS IS GIT. IT TRACKS COLLABORATIVE WORK
ON PROJECTS THROUGH A BEAUTIFUL
DISTRIBUTED GRAPH THEORY TREE MODEL.

COOL. HOW DO WE USE IT?

NO IDEA. JUST MEMORIZE THESE SHELL
COMMANDS AND TYPE THEM TO SYNC UP.
IF YOU GET ERRORS, SAVE YOUR WORK
ELSEWHERE, DELETE THE PROJECT,
AND DOWNLOAD A FRESH COPY.



**DOT DOT
(DOT)
NOTATION**

DOUBLE DOT WITH GIT DIFF..JUST A RANGE

- find the difference between the current HEAD and what it looked like 3 commits ago:

```
git diff HEAD~3..HEAD
```

- do the same thing for one file

```
git diff HEAD~3:filename..HEAD:filename
```

DOUBLE DOT WITH GIT LOG

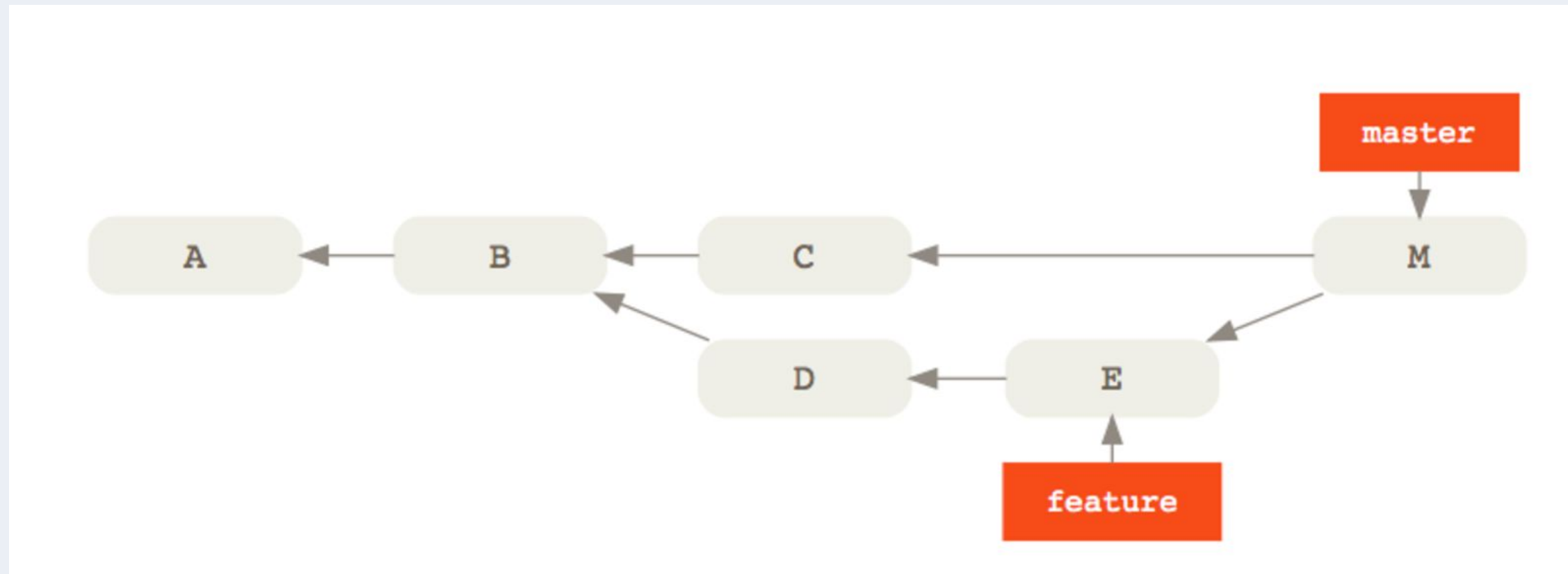


- find commits which are not reachable from the first reference, but are reachable from second reference

`git log master..feature`

- in other words...which commits are in the feature branch that aren't in master? (D and E)
- if we flip the references, i.e., **`git log feature..master`** we will get C

TRIPLE DOT (GIT LOG ONLY)



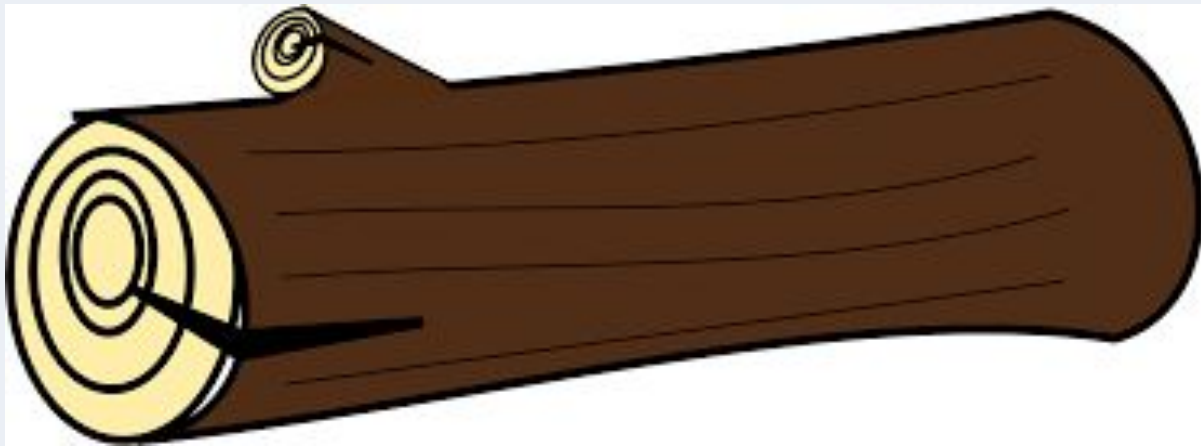
- find commits which are reachable from either master or feature, but not both

`git log master...feature`

- in other words, which commits are not shared by both **master** and **feature**



REFLOG



REFLOG

- Git keeps a log of where **HEAD** and branch refs have been

```
git reflog
```

```
git reflog branchname
```

- Instead of hashes, use the reflog references...

```
git show HEAD@{1}
```

- ...where was HEAD previously?

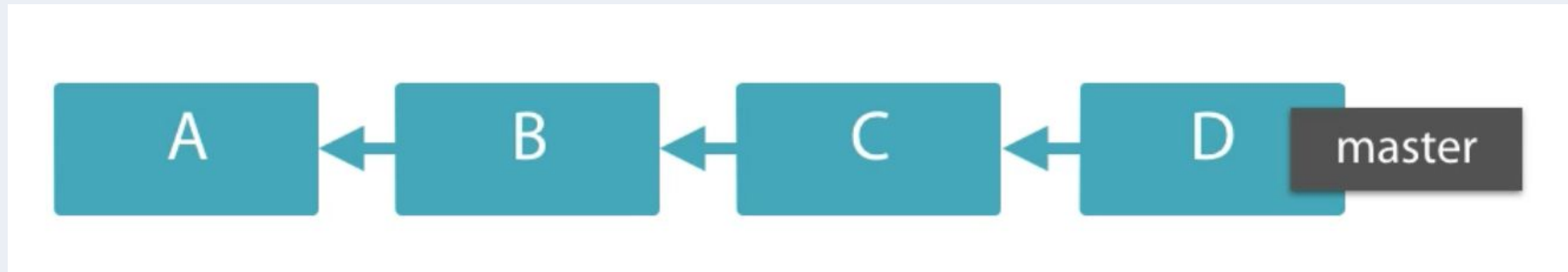
- You can use dates as well

```
git show main@{yesterday}
```

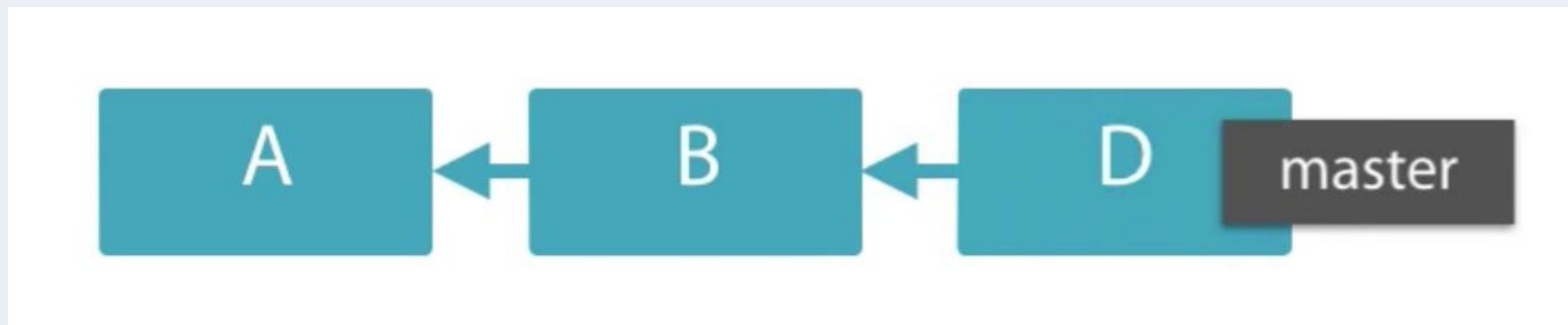
```
git show main@{one.week.ago}
```


USING REFLOG TO FIX MISTAKES

- suppose your history looks like this:



- ...and you accidentally remove commit C via an interactive rebase



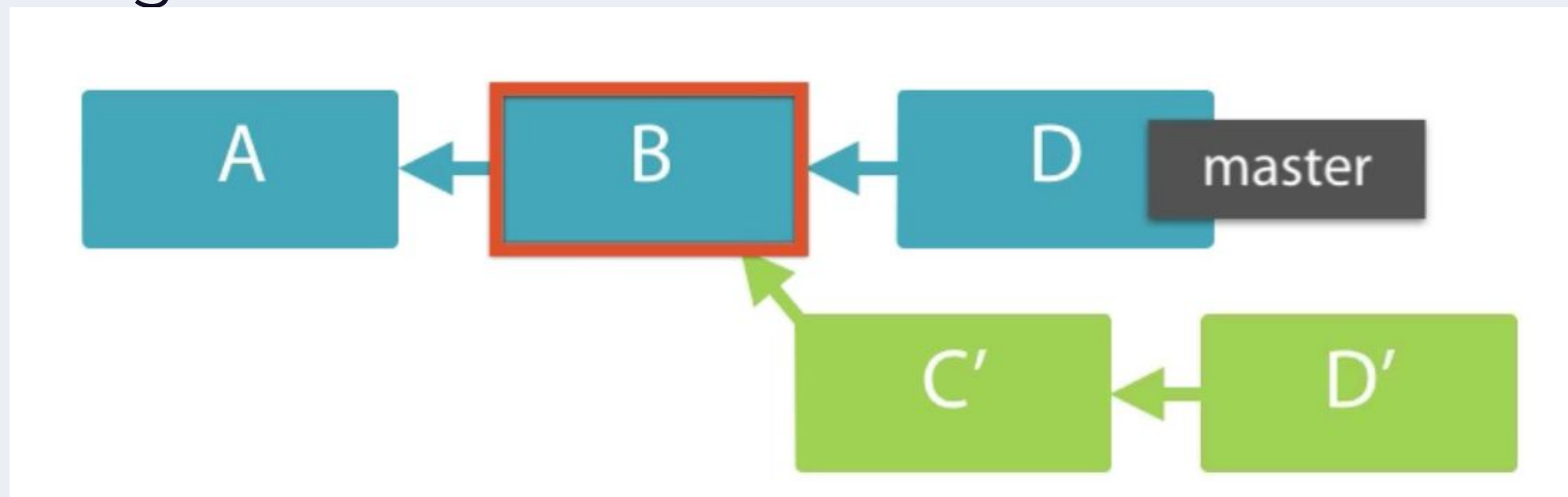
- let's use the **reflog** to fix this
 - we could simply find the lost commit in the reflog

USING REFLOG TO FIX MISTAKES (CONT'D)

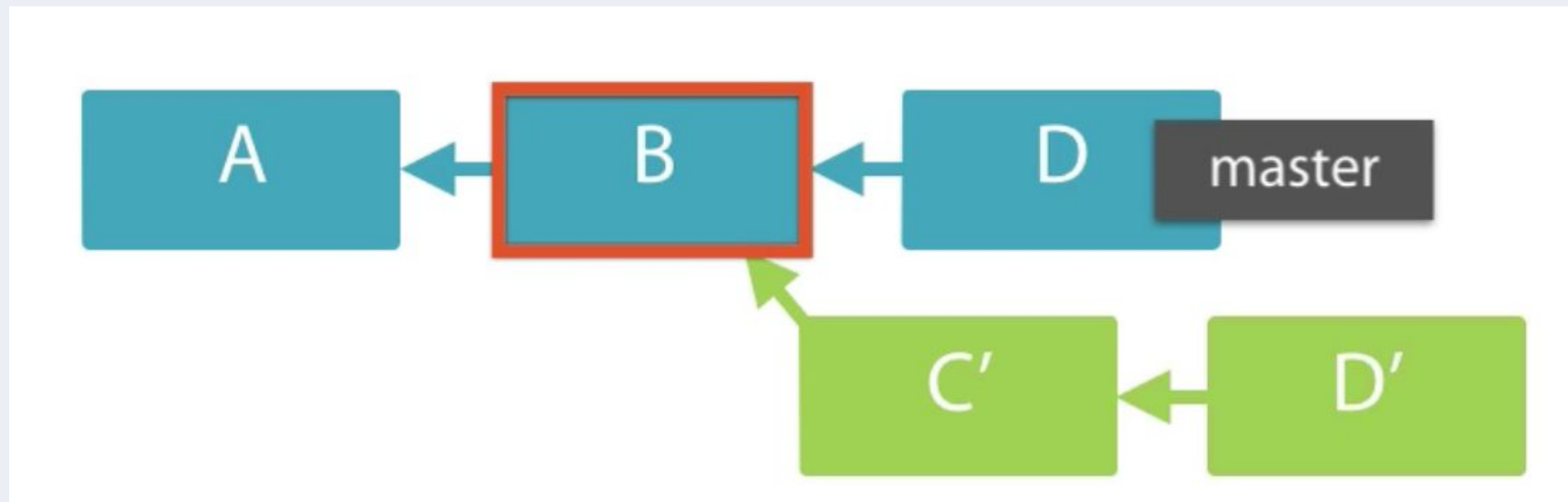
- or we can use the `--grep` option of `git log` to find the commit by commit message (in this case, "C" was our commit message)
- we also want the `--walk-reflogs` option to look through the `reflog` as well
- at this point, we can cherry pick the deleted commit, BUT we can't quite do that because that commit will be put onto the end of our current history
- so we back up one commit using `git checkout HEAD~1`
 - we use `checkout` instead of `reset` so we don't move `HEAD`

USING REFLOG TO FIX MISTAKES (CONT'D)

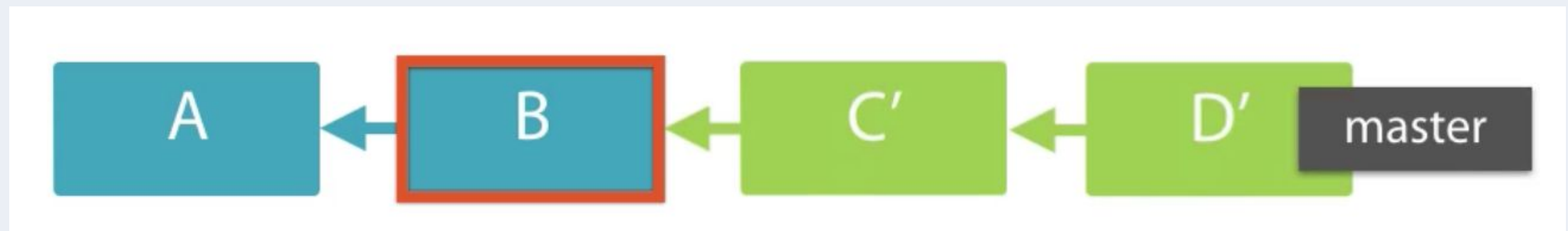
- we can now cherry pick this lost commit onto our current position
- since the **main** branch still points at the commit that should follow the one we cherry picked, we can cherry pick **main** to get that commit
- finally, we have to point master at **D'**, the cherry-picked commit which is patch-equivalent to **D**, which master is currently pointing to



USING REFLOG TO FIX MISTAKES (CONT'D)



- now we need to move main
`git branch -f main HEAD`
(-f is needed to set the branch to a specific commit)





**ODDS +
ENDS**

GIT RERERE

(REUSE RECORDED RESOLUTION)

- if we're doing multiple merges and keep getting identical conflicts, we can use this command to automatically resolve the conflict as we did previously
- e.g., long-lived topic branch
 - merge in **main** to get latest changes and check your work
 - because your branch and **main** touch the same file, you get a conflict and resolve it
 - now you back out the merge (so you don't have all these "merge from master" commits) and keep working
 - lather, rinse, repeat
- in order to use this command, we turn it on as follows:

```
git config --global rerere.enabled true
```


LAB: GIT RERE

1. create two branches, each of which creates a new file
2. force a conflict by changing the same line of file in each branch
3. merge first branch, then second branch, causing a conflict (you should see "**Recorded preimage for <filename>**")
4. resolve the conflict (you should see "**Recorded resolution for <filename>**")
5. delete the branches
6. repeat steps 1-3 (you should see "**Resolved <filename> using previous resolution**")
7. check the file and then commit

TRACKING AUTHORS

```
git blame <filename>
```

- Annotates file with latest commit which affected each line

```
git blame -L 70,85 <filename>
```

- Limit the output to a range of lines

```
git shortlog -s -n
```

- **-s** = sum total number of commits by each author
- **-n** = sort by that number

GIT BISECT

- if your code is broken, and you tell Git the last known good commit, it will find the point at which the error was introduced by bisecting (cutting in half) all the intervening commits

git bisect (start the process)

- then tell Git the bad and good commits

git bisect bad <commit or defaults to current>

git bisect good <commit or tag when last good>

- Git will then switch to midpoint
- tell Git whether good or bad, and it will continue until it identifies the bad commit

git bisect reset (to reset your HEAD back to original)

HOOKS

- Hooks allow you to execute custom scripts when certain things happen, such as committing
- Stored in **.git/hooks**
- To enable a hook, put the appropriate named file in the **hooks** subdirectory
- Check out the samples
`ls .git/hooks`
- Client-side
 - pre-commit, prepare-commit-msg, commit-msg, post-commit, pre-rebase, post-rewrite, post-checkout, post-merge, pre-push, pre-auto-gc, and more
- Server-side (on the remote) pre-receive, update, post-receive

HOOKS (CONT'D)

- Add `prepare-commit-msg` file to `.git/hooks`

```
#!/bin/sh
```

```
echo "# Please include a commit-message!" > $1
```

- Don't forget to make it executable

```
chmod +x prepare-commit-msg
```

- Then try a commit!

EXAMINING INDEX AND WORKING AREA

- of course we can use `git status`, but that doesn't tell us exactly what's in the index, just what's staged
- `git ls-files` will show us what's in the index
 - try it with the `-s` option
- `git ls-tree <commit>` will show us what files are in that snapshot

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

MERGETOOL

- Perform merges using a visual tool by setting up mergetool

```
git config --global merge.tool <toolname>
```
- Then during a merge conflict

```
git mergetool
```

```
git mergetool -t <toolname>
```
- To see some options you may have already

```
git mergetool --tool-help
```

DIFFTOOL

- You can use a similar visual tool for viewing diffs, too

```
git config --global diff.tool <toolname>
```

- Then just use it

```
git difftool
```

```
git difftool -t <toolname>
```

- To see some options you may have already

```
git difftool --tool-help
```

- More config

```
git config --global difftool.prompt false
```

GITK

- GitK for a graphical display of the log and search

gitk

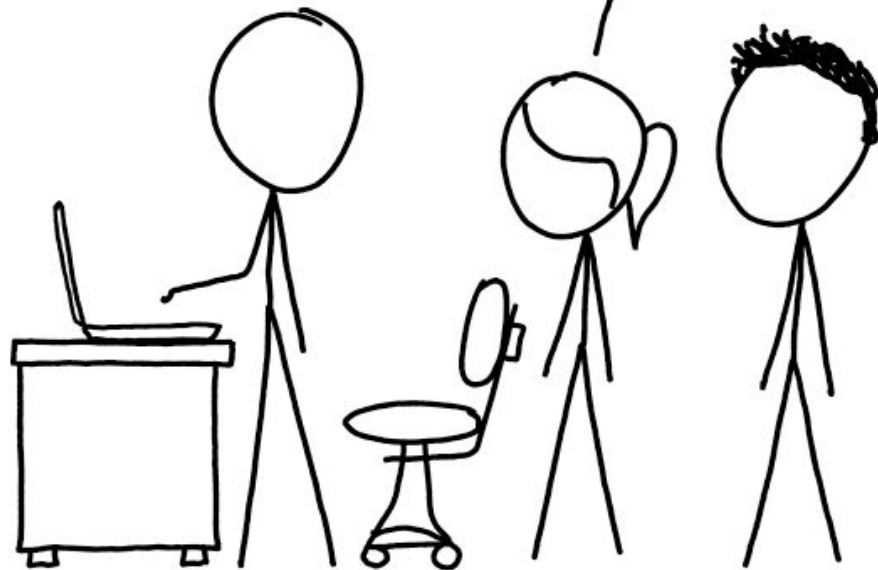
- Accepts most params that git log accepts

gitk --all --decorate

THIS IS GIT. IT TRACKS COLLABORATIVE WORK
ON PROJECTS THROUGH A BEAUTIFUL
DISTRIBUTED GRAPH THEORY TREE MODEL.

COOL. HOW DO WE USE IT?

NO IDEA. JUST MEMORIZE THESE SHELL
COMMANDS AND TYPE THEM TO SYNC UP.
IF YOU GET ERRORS, SAVE YOUR WORK
ELSEWHERE, DELETE THE PROJECT,
AND DOWNLOAD A FRESH COPY.



PARTING WORDS

GIT BEST PRACTICES

- Commit often
- Write useful commit messages (now or later)
- Branch new work (don't work on **main**)
- Stash to maintain a clean working area when you switch projects (or commit, since you can always edit history)
- Keep up to date
- Establish a branching and team workflow
- Tag your releases
- Don't change shared history!

IF YOU GET INTO TROUBLE

- Don't panic!
- Remember that deleted commits won't really be deleted for 60 days, so no need to rush and make things worse
- For not-yet-shared work, edit your history at will

git commit --amend

git rebase -i

git reset

- For shared work, use **git revert** and own up to your mistakes
- To recover lost work, the **reflog** is your friend
 - Also **git fsck** will let you see dangling and unreachable objects

RESOURCES

- Cheat sheet

<https://www.git-tower.com/blog/git-cheat-sheet/>

- The Git Parable:

<http://tom.preston-werner.com/2009/05/19/the-git-parable.html>

- Pro Git, 2nd Edition <https://git-scm.com/book/en/v2>

- Git documentation <https://git-scm.com/documentation>

- Visualizing Git <https://git-school.github.io/visualizing-git>

- Git Branching <http://pcottle.github.io/learnGitBranching>