now you `git` it! (day i of ii)

wintersession winter 2022, january 13 @ 10:00
by dev dabke (ddabke@princeton.edu)

me (@DbCrWk, ddabke)

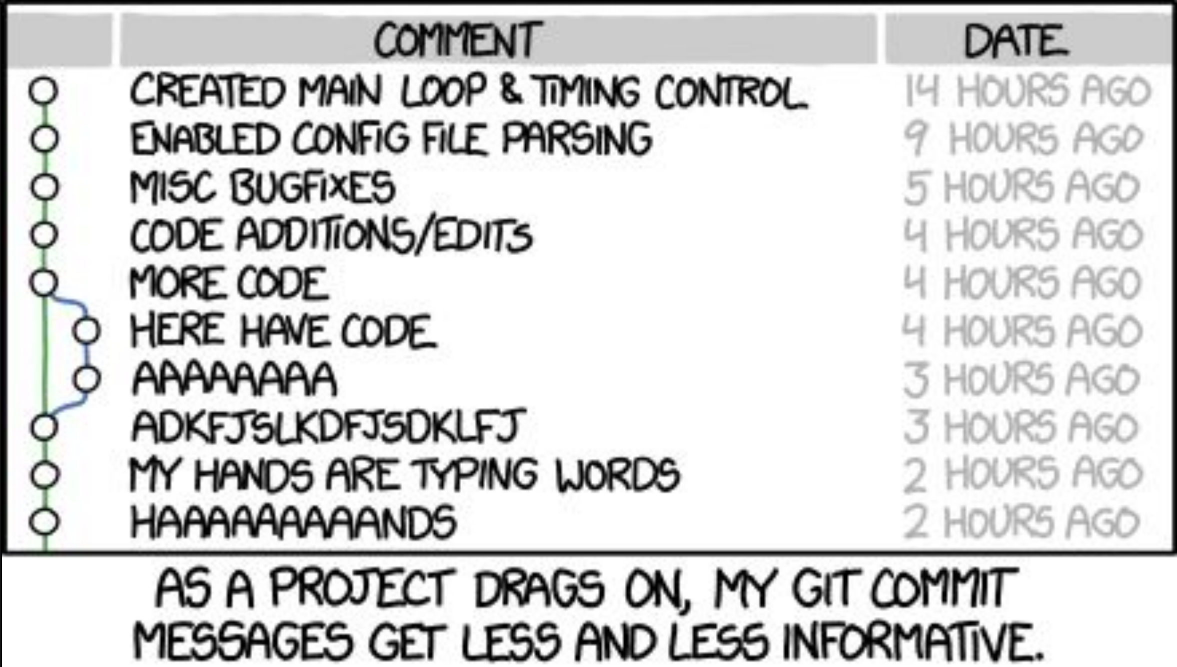
- he/him/his
- education
  - applied math phd @ Princeton
  - math/cs undergrad @ Duke
- industry
  - Facebook, NASA, BlackRock, Airtable
- started with svn, then realized how git works and why it’s awesome

shoutout to @jiaweizhang

- engineer
- electrical engineering undergrad @ Duke
- wrote part of this talk with me four years ago

1. introduction

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.



	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

In case of fire

1. git commit
2. git push
3. leave building

citation: xkcd, https://github.com/qw3rtman/git-fire

workshop structure

1. overall style
    a. feel free to jump in with questions at any time (don’t be afraid!)
    b. i will poll the audience a lot
2. start with basic concepts and theory; then practical exercises
    a. discuss basic ideas; play with a few
    b. learn git for working with ourselves and other people
3. high-level
    a. **day 1**: basics, toy examples, theory
    b. day 2: exercises; personal and team workflows
4. works cited: https://git-scm.com/book/en/v2

main points

1. use version control
2. be fearless
3. perfect is the enemy of the good
4. git is designed (see https://github.com/git/git)

what is version control (vcs)?

why version control?

why version control?

1. tracking changes over time

why version control?

1. tracking changes over time
2. working with other people

key vcs dichotomies

1. **source** vs. generated
2. **code** vs. *binary*
3. immutable vs. **mutable**
4. **distributed** vs. centralized

do we want everything under version control?

do we want everything under version control?

no!

- sometimes call `git` “source control”
- general rule: track the least info you can to generate everything else

source vs. binaries

- what is this file type?
  - “code” : direct human-readable text that can be edited
  - “binary” : data and content made for some computer program
- how is this file being used?
  - “source” : ground-truth information, data, or files
  - “generated” : secondary information, data, or files that can be wholly constructed from “source”

source vs. binaries, e.g.

	source	generated
code	SomeClass.c	SomeClass.o
binary	figure in a paper	pdf file from latex

centralized vs. decentralized

centralized

- one “master” repo
- central server or machine hosts the “canonical code”
- checkout / checkin to one place
- checkout only parts of a repo
- easy to manage permissions

decentralized

- everyone has the **entire** repo (yes, submodules exist in git, etc., but c’mon)
- checkout / checkin from anywhere
- “canonical code” is **established by convention**

immutable vs. mutable

immutable

- cannot rewrite the “history” or “record” of what has happened
- history reflects what actually happened
- safer

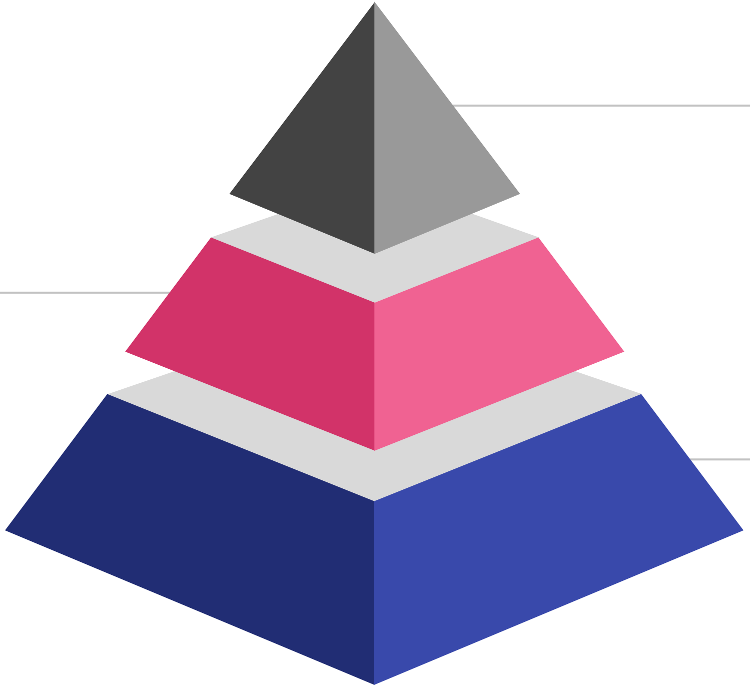
mutable

- possible to change history
- history can be beautified, changed to be easier to read for the future
- more flexible
- can be more dangerous
- **must establish convention**

isn’t github a centralized system, though? wait . . . what is github?

what is a convention in code and why do they matter?

assumptions & conventions at 3 levels



user-level

1 commit messages and branch names are meaningful

application-level (git)

2 we progress forward in time; collaborative coding

system-level

1 code needs to be tracked on different computers with multiple copies of code on each computer

demo 1: goals

1. commits: units of change
2. branches: connecting these units of change
   a. support **decentralization**
   b. support **mutability**

demo

commits and branches; go to: https://learngitbranching.js.org/?NODEMO

demo 1: takeaways

1. how decentralized, mutable VCS **has** to work
2. how to manipulate git assuming files are tracked

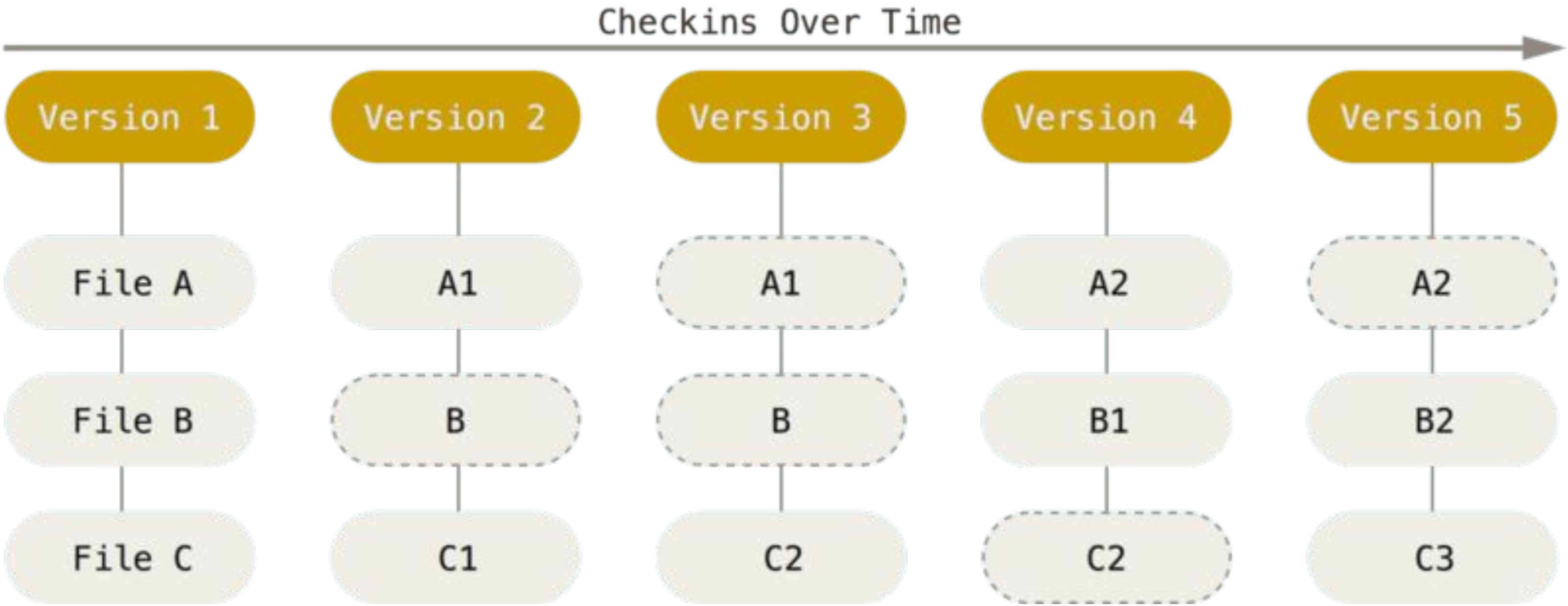
unanswered questions

1. how does git know what is a “unit of change?”
2. how does git connect to other computers?

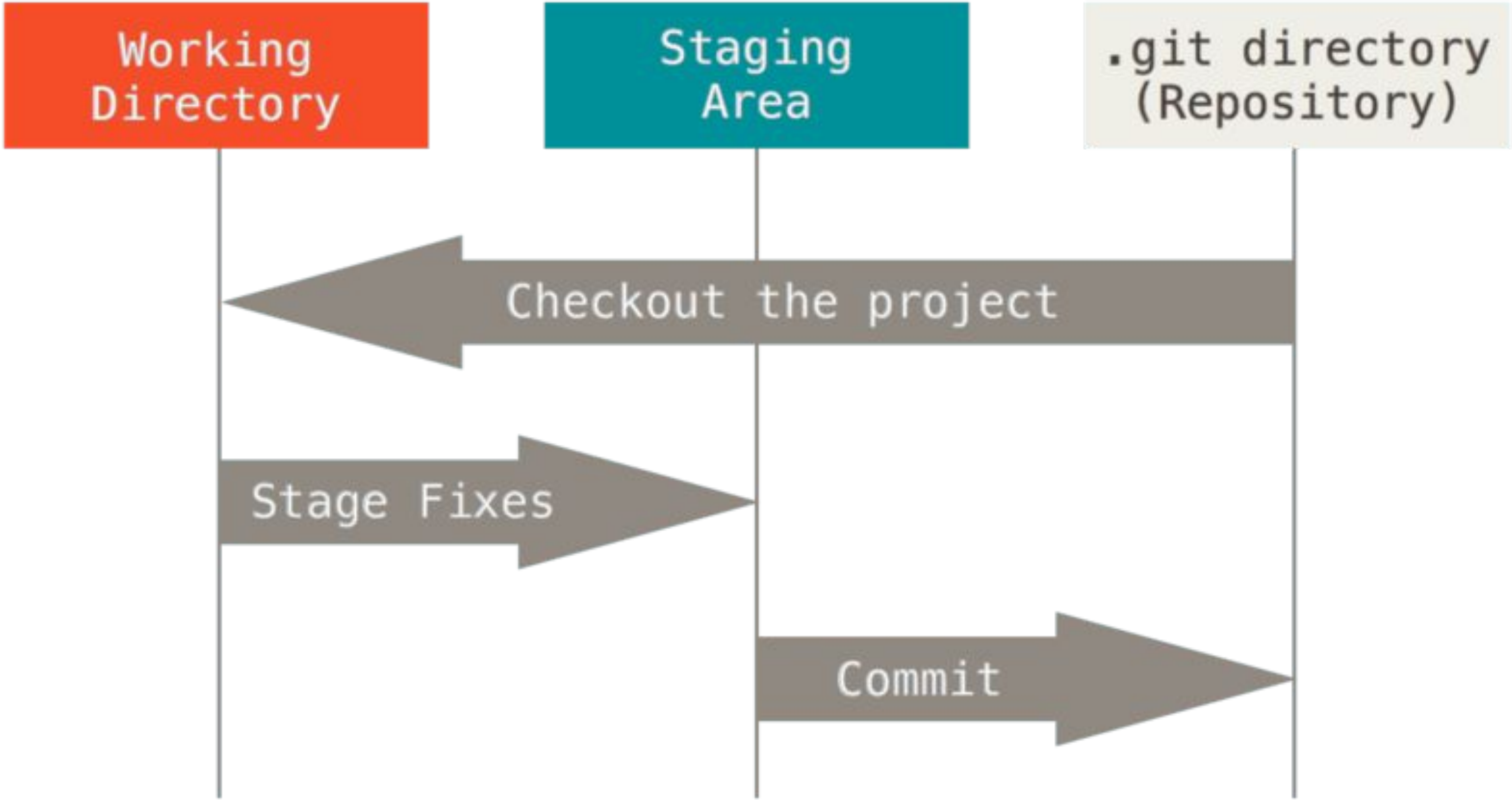
2. how does `git` track changes?

starting a git project

1. pick a folder (called a “directory”) on your computer
2. run `git init`
    a. tells git to track the directory
    b. for efficiency: tracks changes



`git` takes “snapshots” of your repositories (repos)
citation: https://git-scm.com/book/en/v2/Getting-Started-Git-Basics



citation: https://git-scm.com/book/en/v2

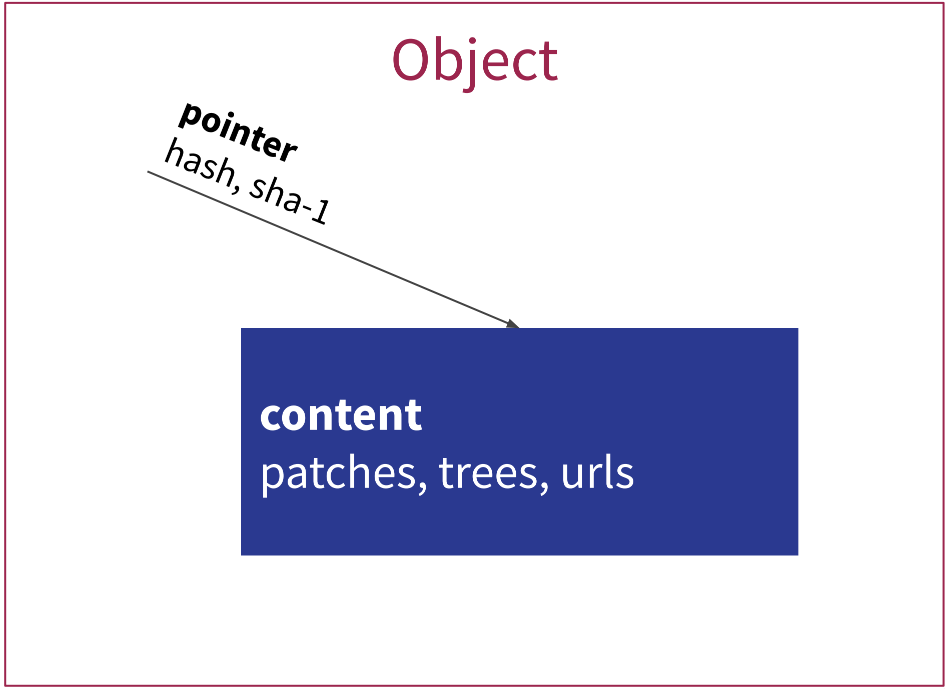
storing a unit of changes

1. commits: units of changes, based on convention
   a. **you** decide what changes are “meaningful” to package into a commit
2. simple git operations:
   a. `git add`
   b. `git commit`

3. technical details: clarifying conventions

okay, but how does `git` actually track these files?

everything is an object



Object

pointer
hash, sha-1

content
patches, trees, urls

what is a hash again?

okay, but how do i remember hashes?

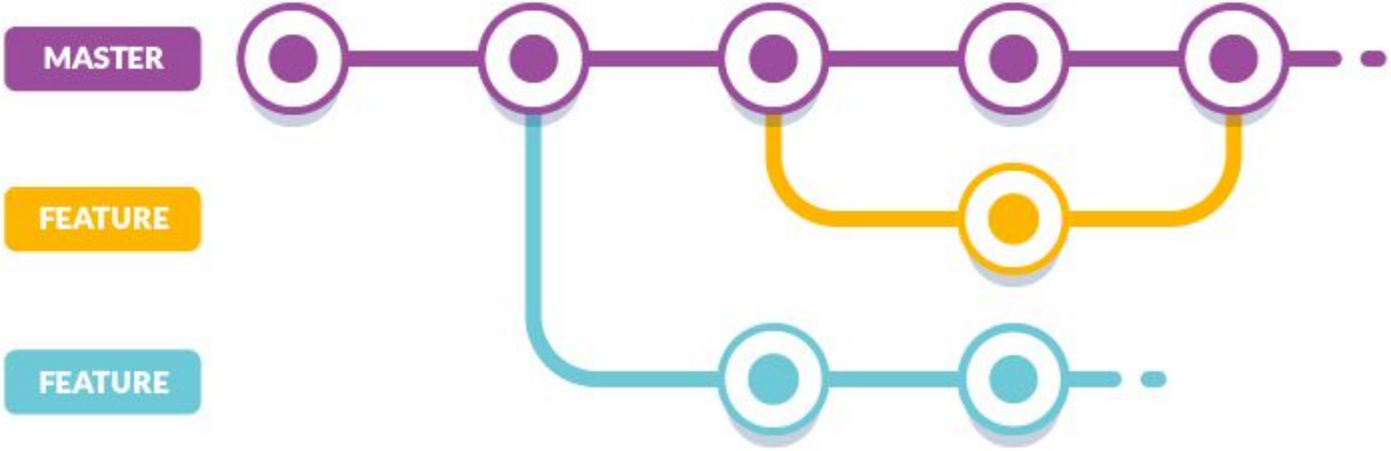
hashes are labelled

- humans cannot remember complicated hash values
- by convention: hashes use shortened value
- git allows us to label objects
  - commits: labelled by branches, tags, etc.
  - other computers: labelled by remotes

commit: the most important object

- **pointer**: some hash value
- **content**
  - author*
  - timestamp*
  - previous commit
  - patch
    - changes made from last commit
  - cryptographic signature (optional)

*`git` is non-adversarial and trusts the given information; these are not verified and be very easily faked



citation: https://buddy.works/blog/5-types-of-git-workflows

what is special about `master`?

nothing!

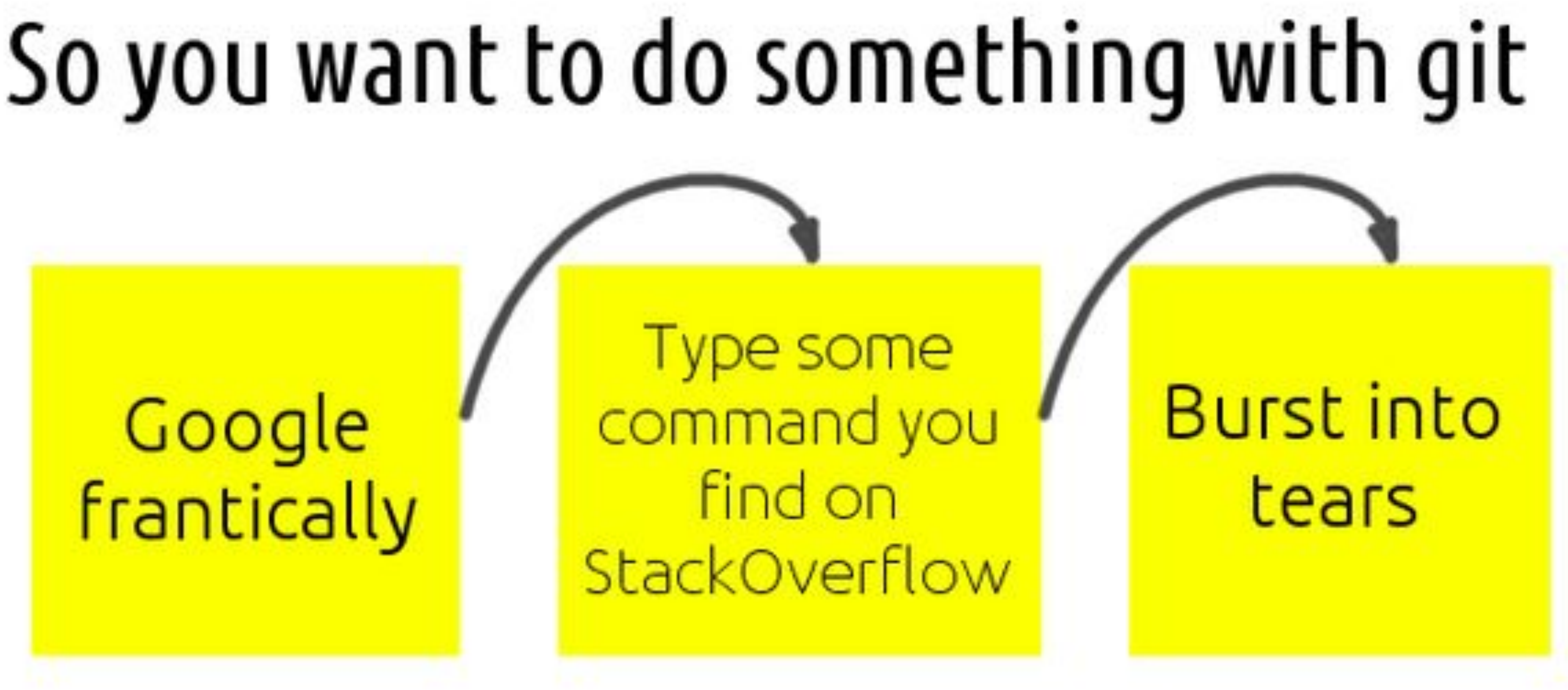
4. forming commits

preamble: using a command line

- unforgiving, but don’t panic!
    - case-sensitive
    - not optimized for readability
    - order of commands matters

preamble: git

- let’s make sure `git` is installed
- by operating system
  - linux: you probably already know what to do
  - mac: you are secretly running linux
  - windows: ¯_(ツ)_/¯ (ssh into adroit and use linux)
- there is an “activation energy” to using `git`



So you want to do something with git

Google frantically

Type some command you find on StackOverflow

Burst into tears

citation: somewhere on google

demo

on your computer this time

configure `git`

```
$ git config --global user.name “git fan”

$ git config --global user.email gitfan@git.git
```

use email you will use for “most” of your projects (and that you will link to github)

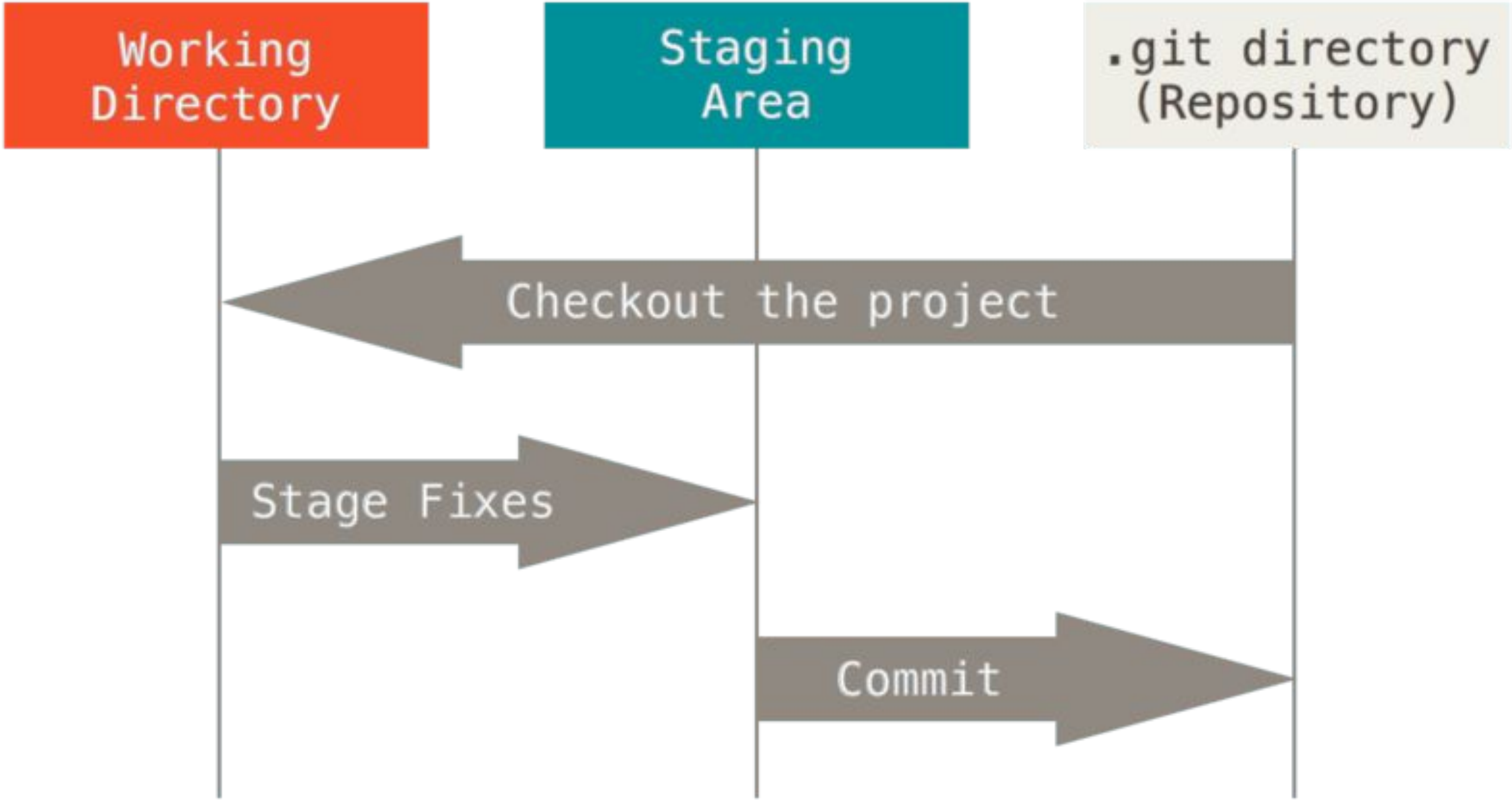
*reminder: git trusts whatever you tell is and does not verify; github verifies

git init

- pick a directory (a folder) to track
- once initialized: directory is called *repository*

`git add`

add file(s) to staging area



https://git-scm.com/book/en/v2/Getting-Started-Git-Basics

git commit

commit files in the staging area; mark these changes as one “unit”

git status

figure out what is happening in your repository

`git log`

see the **history** of git

5. connecting commits

`git checkout -b`

checkout a new branch

pop quiz: what is special about `master`?

git merge

merge a branch (or object) into another branch (or object)

merge conflicts

- what is a merge conflict?
- what does it represent?

`git reset`

remove stuff from your staging area that you added

git revert

safe “undo” by adding a new commit with old code, rather than modifying history

`git show`

shows an object

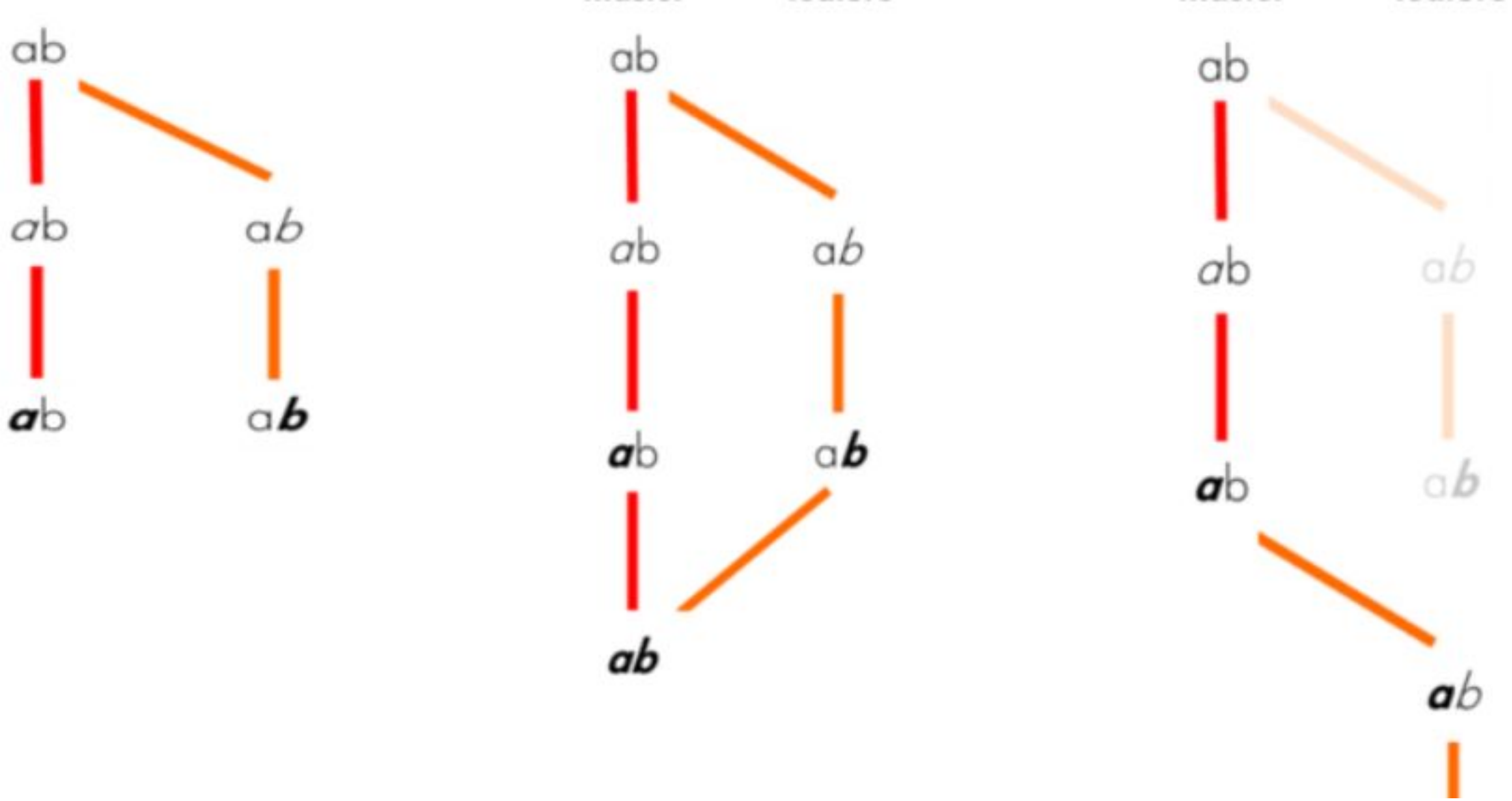
git diff

see the difference between two objects

6. advanced `git` stuff

`rebase` vs. `merge`

- `rebase`: opposite of merge
- case study: take commits of feature and place them on top of master
  - merge: git checkout master; git merge feature
  - rebase: git checkout feature; git rebase master; git checkout master; git merge feature



citation: https://medium.com/datadriveninvestor/git-rebase-vs-merge-cc5199edd77c

git add -p

more granular way to add content to the “staging area”

git clean

remove untracked changes that are not in the staging area

```
git clean -n # shows changes

git clean -f # deletes changes

git clean -fd # deletes changes that include directories
```

reflog

git magic that let’s you see where your refs have been pointing

what does this mean?

you can see **all** code/objects that you’ve seen on your computer

stash

- don’t really use this
- in case you need to quickly store your code without committing it
- don’t rely on this! it’s not real version control

gitconfig

- shoutout to @codebytere (she works at github)
- configure your git and aliases
- protip: put your **gitconfig** under git so you can easily set up a new environment

gitignore

tl;dr: tell git to **not** put something in version control

Samples

- `generated_binary_file.bin` (a specific file)
- `generated/binaries` (a directory)
- `*.pdf` (any pdf file)
- `!this_one_pdf.pdf` (an exception to a more generic rule)

references

https://git-scm.com/book/en/v2/

now you `git` it!