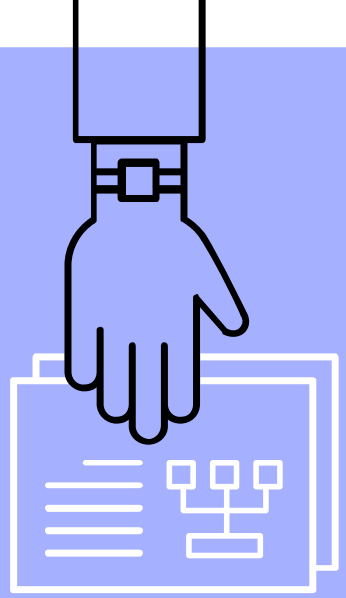
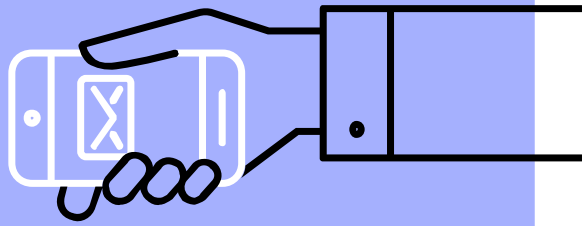


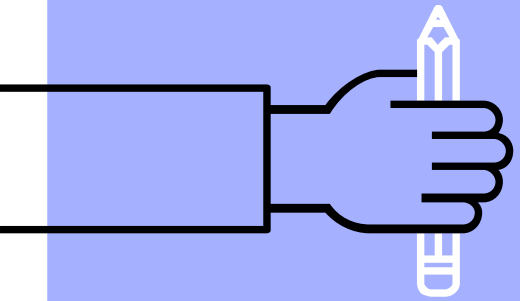


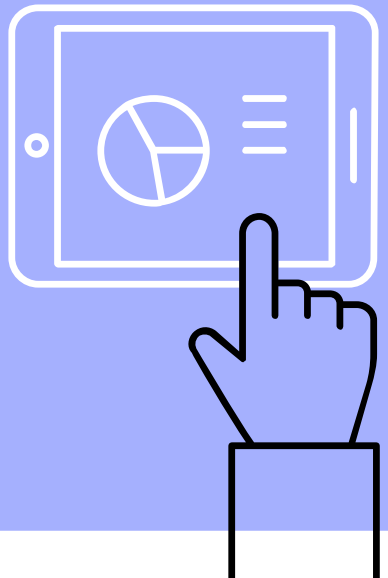
git-basics

ECS 124

Thursday, May 30

6:00 pm - 7:30 pm



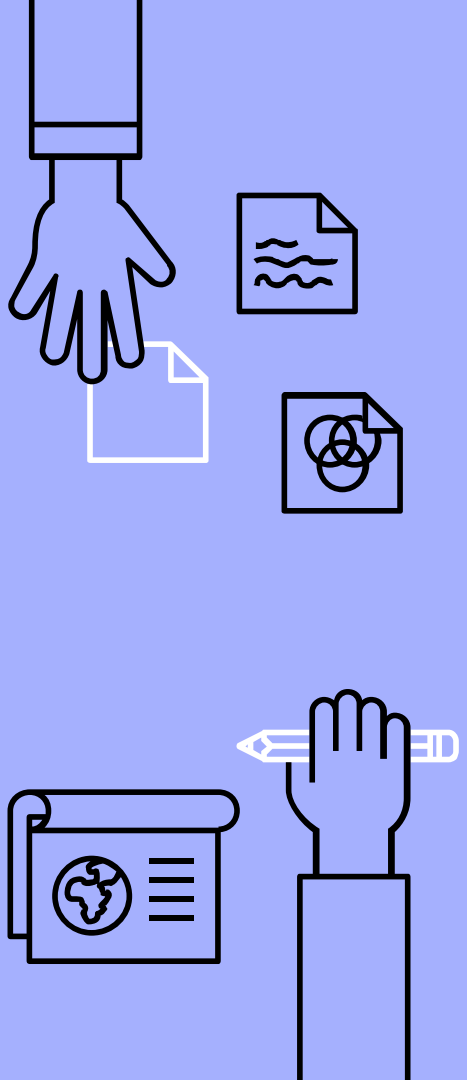


Pre-Instructions

Have a Github account.

Install git on your machine.

If you don't have either set up, you can follow along with a fellow developer or put up your hand and a mentor can come help out!

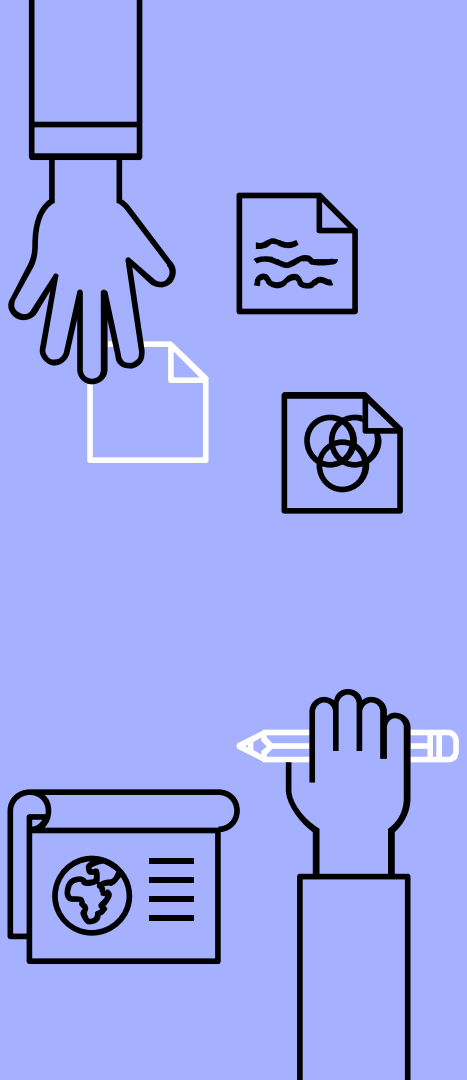


What is this workshop?

The goals of this workshop are:

- To learn the vocabulary of git and Github
- To have an understanding why and when you'd use certain git commands

Note: anything with an **I** *means intermediate info.*

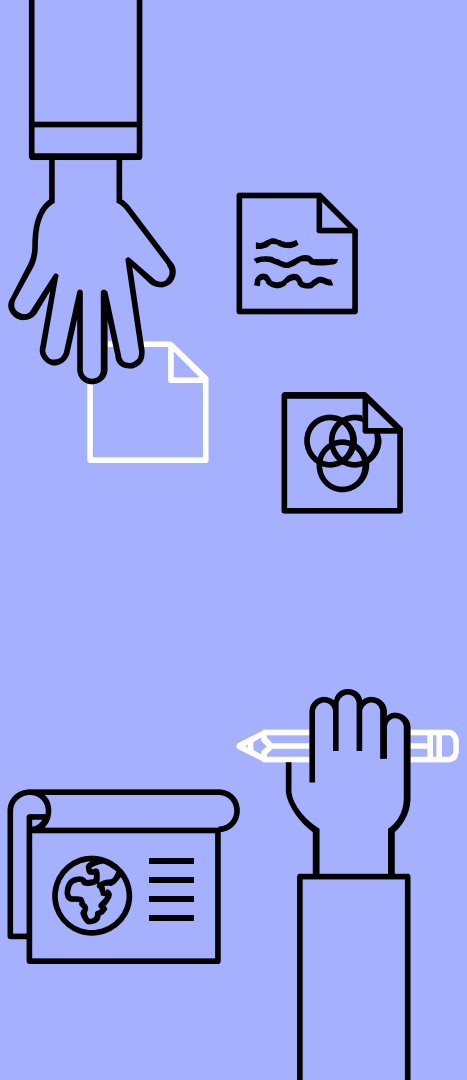


What is this workshop?

There are three main parts:

1. What are these and how do we use them?
2. Other Useful Git Commands
3. Extra Dev Terms

*Note: anything wrapped in squiggly brackets, **${}**, means you put your own input in.*



HELLO!

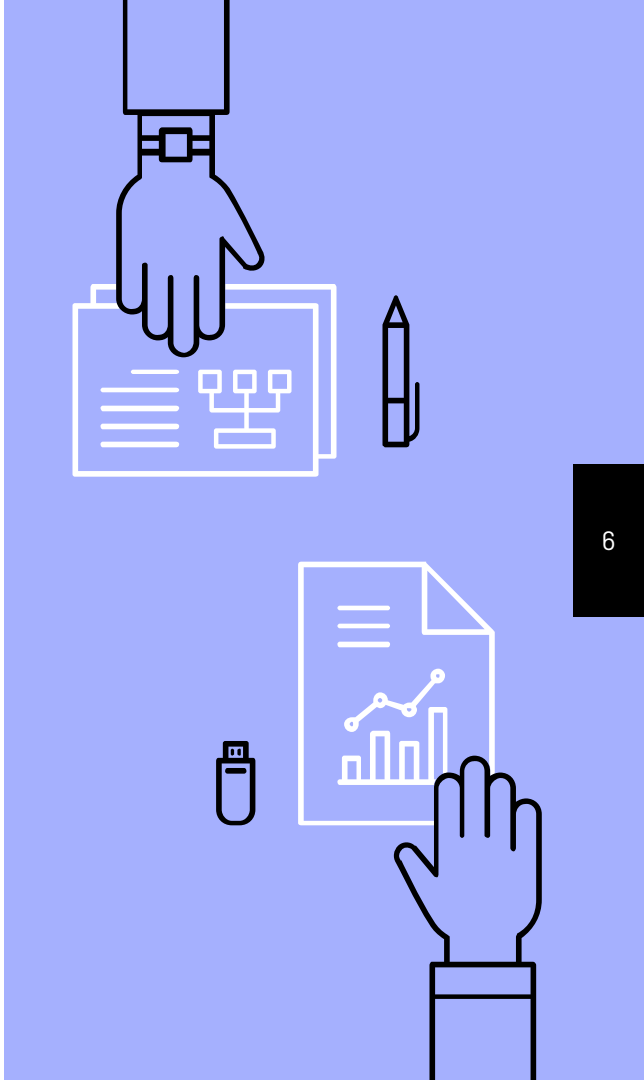
I'm Kaitlin Erb.

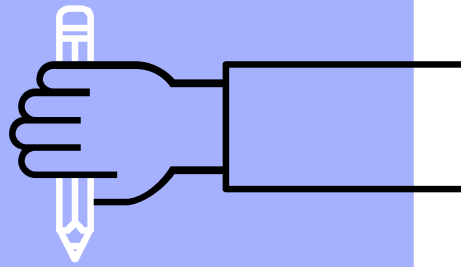
I do things on the internet, person at @uvicwebdev.

You can find me at **@k-erby**

Previous Experience

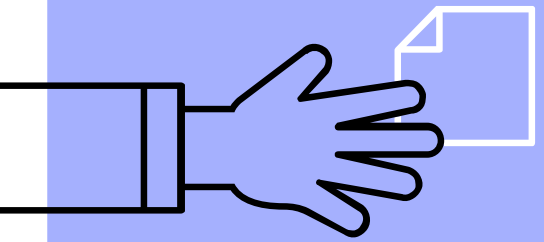
- Who has used git before?
- Who has worked on a team that used git?
- Who is comfortable in their knowledge of git and Github?





1.
What’s git?

And how do we use it?



Git and Github

What is git?

- An open source distributed version control system.
- **Open source:**
  - Made by Linus Torvalds to manage Linux kernel development.
- **Distributed:**
  - Not centralized - everyone has their own git. (Github is centralized.)
  - Github stores *remote repositories* in a server. **(remote, ie. origin/upstream)**
  - A developer's computer has a *local version* of code. **(local)**
- **Version control:**
  - A system that records changes to files and allows you to reference previous states.

Git and Github

Why version control?

- Because we're humans and we fuck up.
  - Rolling back bugs, checking out previous code
- Because development is a team sport.
  - Fixing code conflicts between developers.
  - Working on parallel features.
- Because requirements change.
  - Allowing a single source of truth.

Git and Github

What is Github?

- A **hosting service** for git repositories.
- There are other hosting services besides Github:
  - GitLab
  - BitBucket
  - SourceForge
  - Launchpad
- Github is the **most common** hosting service for code.

Github Student Developer Pack

!!!! Get this if you haven't already! !!!!

- It's basically a bunch of different tools rolled into one package!
- It's **FREE!**
- Go to the link below, or search "github student developer pack"

https://education.github.com/pack

Max Howell

mxcl

Unfollow

Creator of Homebrew. Open source all day every day.

Savannah, GA

mxcl@me.com

https://mxcl.dev

Block or report user

Organizations

Pinned

AmA

Ask mxcl anything.

★ 13

Workbench

Seamless, automatic, "dotfile" sync to iCloud.

Swift ★ 566 9

swift-sh

Easily script with third-party Swift dependencies.

Swift ★ 1.2k 35

Cake

A delicious, quality-of-life supplement for your app-development toolbox.

Swift ★ 514 7

Path.swift

Delightful, robust, cross-platform and chainable file-pathing functions.

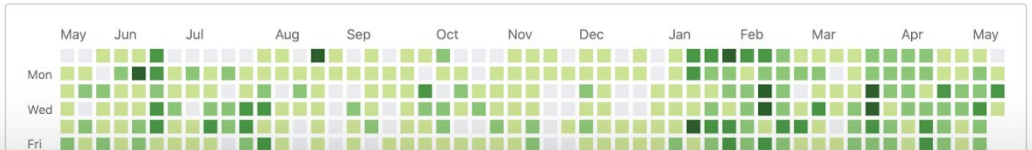
Swift ★ 655 17

LegibleError

Beating `Error.localizedDescription` at its own game.

Swift ★ 117 2

3,533 contributions in the last year



2019

2018

2017

Repositories

What is a repo?

- Something we use to organize a project or set of files (images, videos, etc).

What does a repo contain?

- code
- commits
- branches
- issues
- pull requests (PRs)
- projects

mosop / cli

Watch 2 | Star 80 | Fork 8

Code | Issues 2 | Pull requests 0 | Projects 0 | Wiki | Insights

Yet another Crystal library for building command-line interface applications.

crystal-library cli

185 commits | 4 branches | 53 releases | 4 contributors | MIT

Branch: master | New pull request | Create new file | Upload files | Find File | Clone or download

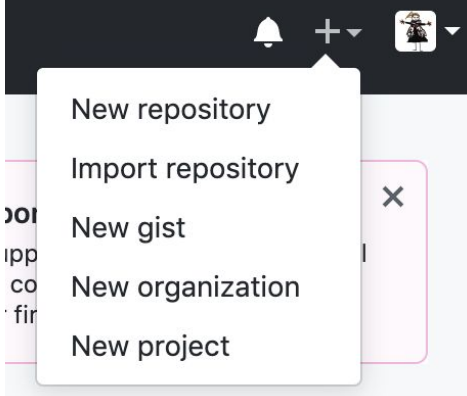
mosop conforms to Crystal 0.25 — Latest commit f3f7068 on Jun 7, 2018

.circleci	configure circleci	11 months ago
bin	configure circleci	11 months ago
spec	conforms to Crystal 0.25	11 months ago
src	0.7.0	11 months ago
.gitignore	0.3.1.2	3 years ago
.travis.yml	optarg 0.5.1	2 years ago
CHANGELOG.md	0.7.0	11 months ago
LICENSE	0.1.0	3 years ago
README.md	configure circleci	11 months ago
shard.yml	0.7.0	11 months ago

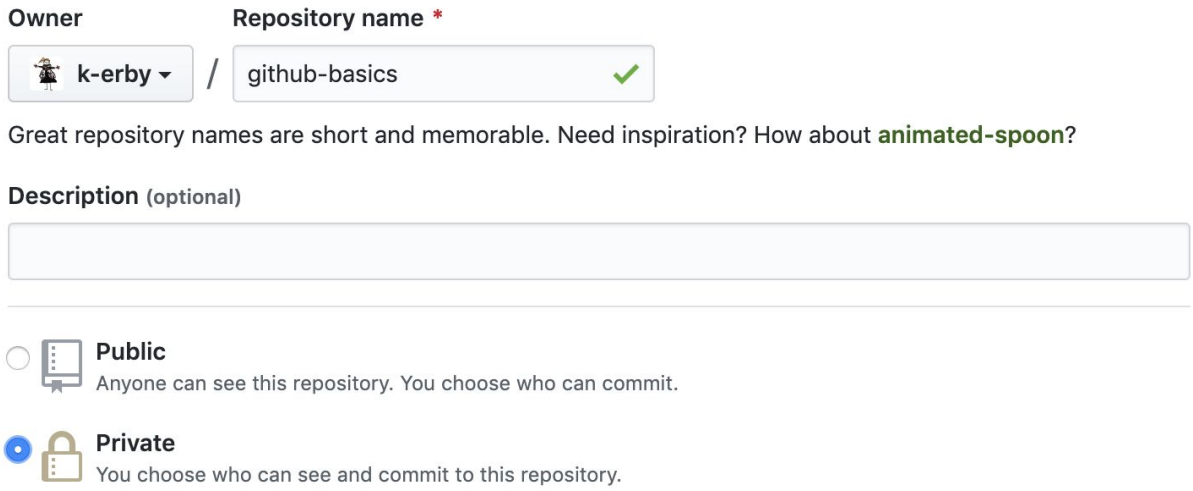
README.md

Crystal CLI

Creating a repo



1. Create a new repo in Github.
2. Go to github.com
3. In the right-hand corner, click on the “+” icon, then “new repository”
4. Give your repository a name and set it to private:



Creating a repo

After you hit "`Create Repository`" you'll get a **`${url}`**

1. Make a folder for your project.
2. Go into your folder.
3. Initialize a git repository.
4. Create a file to add to your repo.
5. Add file to staging.
6. Commit your file for staging.
7. Create a new remote, called origin, at the url.
8. Add an upstream (tracking) branch (aka, link local and remote).

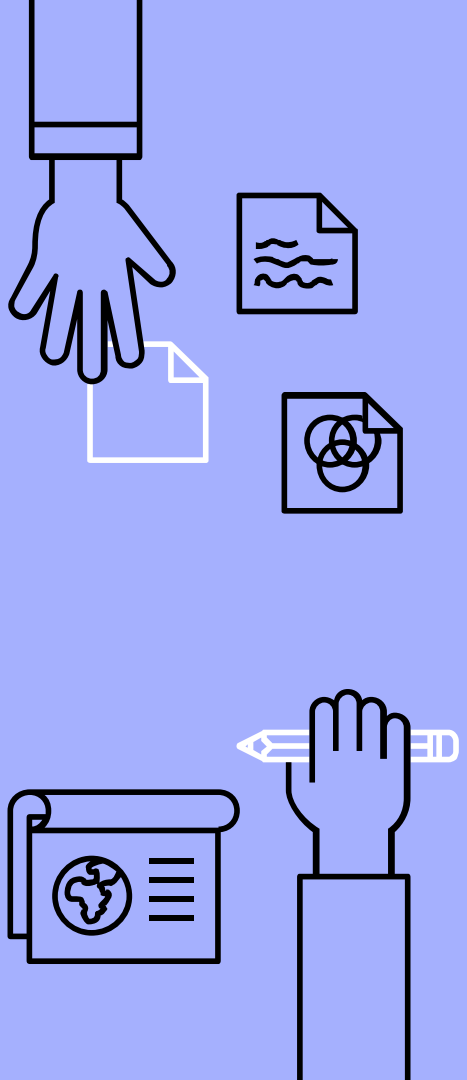
1. `mkdir github-basics`
2. `cd github-basics`
3. `git init`
4. `touch README.md`
5. `git add README.md`
6. `git commit -m "Add README"`
7. `git remote add origin ${url}`
8. `git push -u origin master`

What is (one of) the most important git commands?

```
git status
```

Why?

- It helps you know what's happening in your repo (ie. file changes, stages, commits)
- **Use this all the time**



Staging

- **Staging** is the first step to **pushing** code.
- The **staging area** is where git keeps track of all your modified files.
- **!!! Any file not staged will NOT be committed to your code.**
- You can **stage code** by using one of these commands:

```
git add ${file}
```

```
git add ${file1} ${file2} ${file3}
```

```
git add .
```

Committing

- Once changes are added to the **staging area,** you **commit** them.
- Commits have **messages** attached to them to help fellow developers quickly read what you've changed.

```
git commit -m "Change country title based on map click"
```

- Forgot to add something to your commit **before** pushing? Amend it!

```
git commit --amend --no-edit
```

Anatomy of a Commit Object

- **SHA-1** hashes of metadata and the hash of the **root tree object**.
- **Tree objects map names to SHA-1 hashes**, which are either more tree objects or files (represented by **blobs**).

```
sha1(
    commit message  => "Add README.md"
    committer       => Kaitlin Erb <kerb@gmail.com>
    commit date     => Thu May 30 11:13:49 2019 +0100
    author          => Kaitlin Erb <kerb@gmail.com>
    author date     => Thu May 30 11:13:49 2019 +0100
    tree            => 9c435a86e664be00db0d973e981425e4a3ef3f8d
    parents         => [0d973e9c4353ef3f8ddb98a86e664be001425e4a]
)
```

Pushing

- Once staged and commit, you can now **push them** into your **branch**.
- In school and personal projects you'll probably be pushing into **master** and on dev teams you'll be most likely pushing into **feature branches**.
- Pushing means that you're **taking your local code and pushing to remote**.

```
# origin is our pointer to remote
git push origin ${branch_name}

# pushes commits to the master branch
git push origin master
```

Branches

- It's nothing but a **pointer to a git commit.**
- To see what all **your local branches on your current repo** are you type:

```
git branch
```

- To see what all **your branches on your current repo** are you type:

```
git branch -a
```

Branches

Why/when to use them?

- You're working with other developers on a single codebase.
- There are multiple features you want to work on concurrently.
- You're going to do something *weird.*
- Some teams have a **deployment branch** and a **development branch** to ensure code quality. (Testing the dev branch often before merging into deploy.)

Why/when would you maybe not use them?

- "Master the master branch first." - Courtney Maricle
- If you're in SENG 265: **warning about branches.**
- When you're working on a solo personal project where it doesn't matter if the code in **master** is broken.

Branches

How do we create one?

- There are multiple ways. You can create a branch then go into it. Or do both!

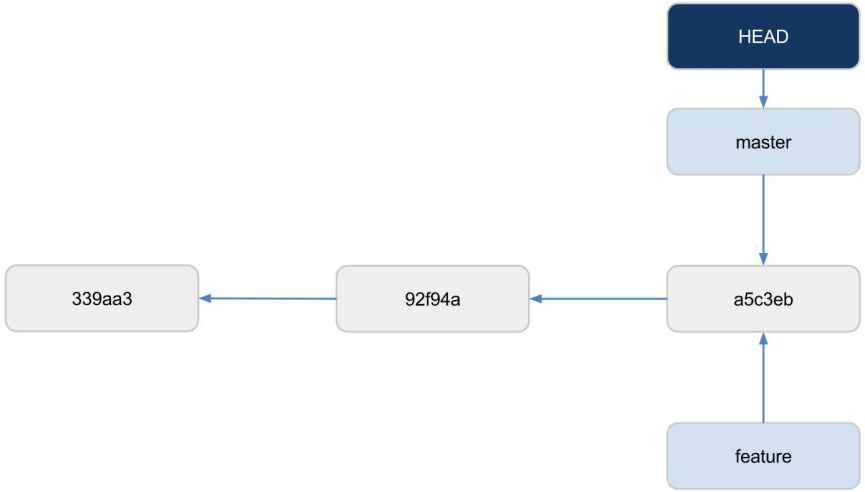
```
git branch ${branch_name}
git checkout ${branch_name}

git checkout -b ${branch_name}
```

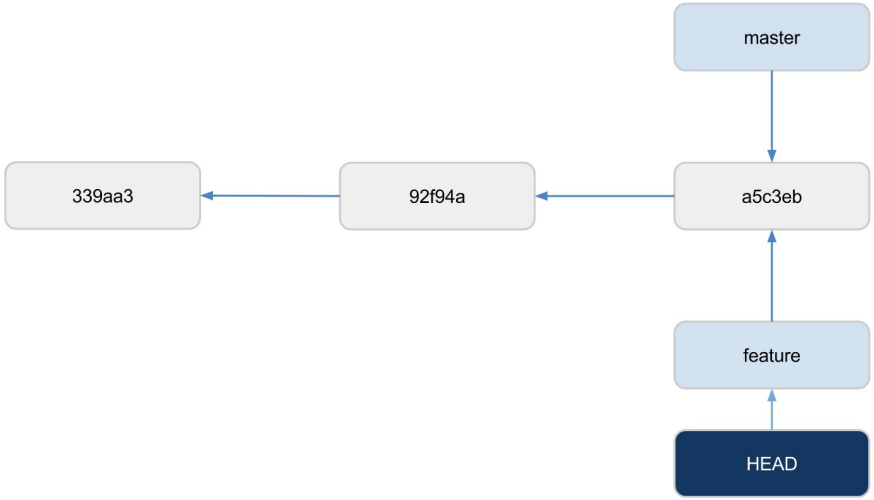
- To go between branches, you use the checkout command **without the flag**:

```
git checkout ${branch_name}
```

Branches

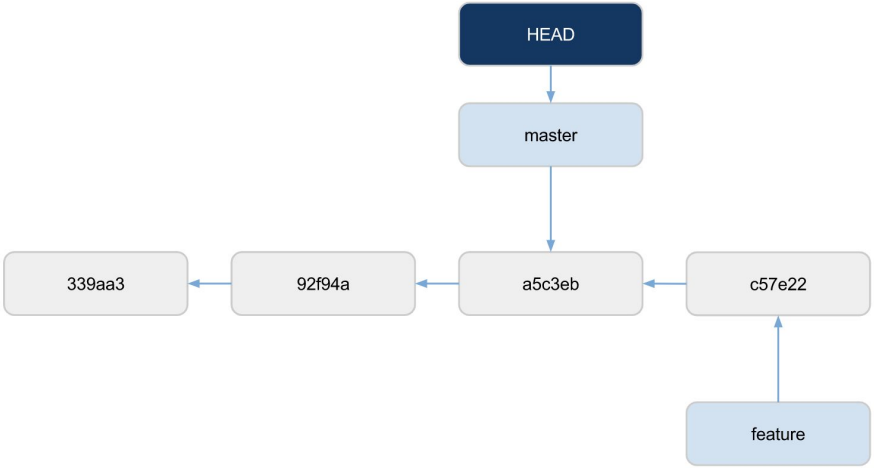


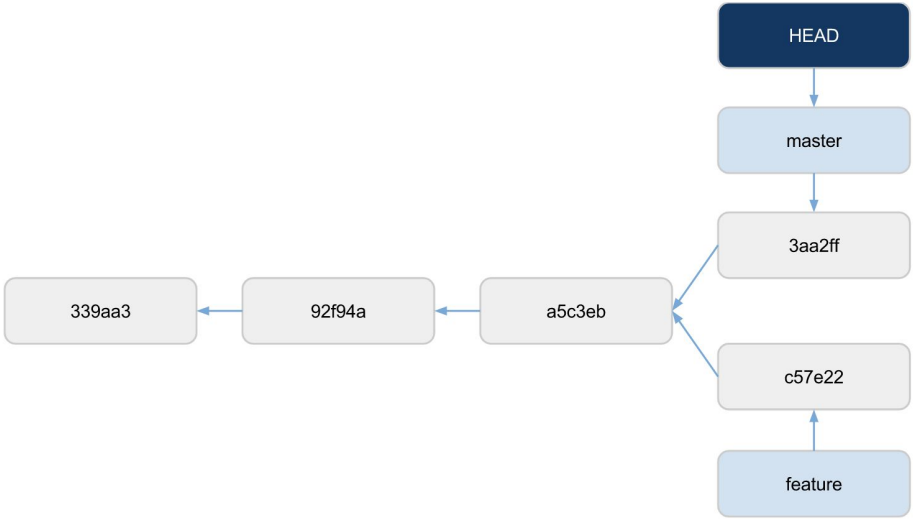
git branch feature



git checkout feature

Branches



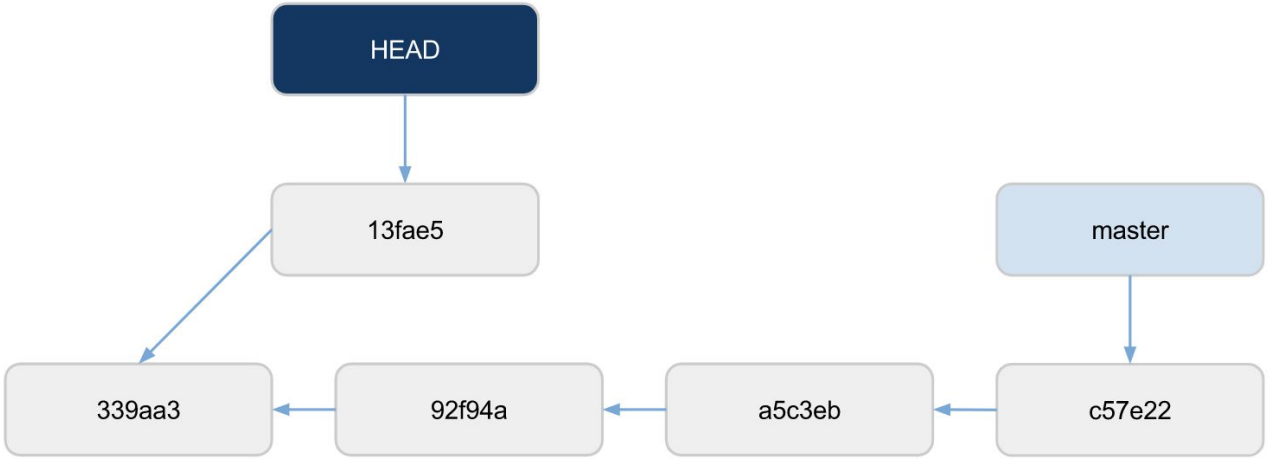



```
# add and commit to feature
git push origin feature
git checkout master
```

```
# add and commit to master
git push origin master
```

Common issue: Detached HEAD

- **Detached HEAD** is simply: HEAD pointed to an explicit commit hash, rather than HEAD pointing to a branch.
- They're not great: **memorizing hashes is hard**, **commits aren't reachable** to others, and they'll will be **garbage collected after 30 days**.



Interactive Steps

1. git checkout -b **${branch_name}**
2. **${Do something????}**
3. git add **${file_name}**
4. git commit -m "**${some_message}**"
5. git push origin **${branch_name}**

How do we add the work we did on our branch into our master branch?

Merging

- Our new branch should be **ahead of master by 1 commit.**
  - **Ahead** = we are up-to-date with master, but added more code.
  - **Behind** = your code is behind master (you might need a **rebase**)
- There are numerous ways to merge your code into master. We'll be using the way that's similar to a `git commit` message:

```
# if you're on your branch, go to master
git checkout master
# merge your branch into the branch you're currently in
git merge ${branch_name}
# push your local branch to remote
git push origin master
```

Pulling and Fetching

- To ensure **master is always be up-to-date,** you'll need to pull or fetch.
- To pull in remote changes into your local repository:

```
git pull origin master
```

- `git pull` is actually **two steps** behind the hood: fetch and merge.
- Fetching is more harmless, since it doesn't manipulate files:

```
git fetch

# the following command resets the pointers
git reset --hard origin/master
```

Rebasing

- Even when you update master, it won't update any branches you have.
- To update your branches, you'll need to **rebase** them or **pull origin master**.
- **Rebasing can be a hellish process**.

Rebasing

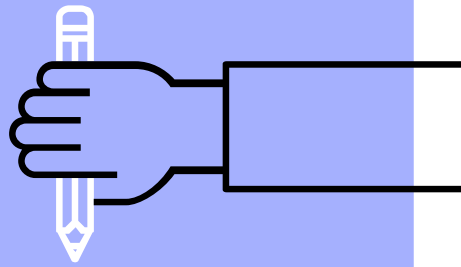
- Rebasing places your work on top of master. To ensure you get up-to-date code from master on your branch, **you may need to fix conflicts**.

```
git rebase ${branch/HEAD@1}
```

- In interactive (`-i`) mode you can do a variety of commands, too:

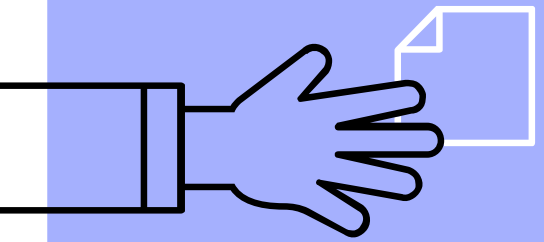
```
pick 3c041de commit 1
f    c0203cb commit 2

# Rebase 85cf1c2..c0203cb onto 85cf1c2 (2 command(s))
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
...
```



2.
Other Useful Commands

Just a hodgepodge of stuff.



Other useful commands?

```
git log
```

- This command lists all the commits to the repo thus far (exit using q).

- You can also make this command ~*fancy*~ and create a tree for the repo:

```
git log --graph --decorate --abbrev-commit
```

The above command will show all the **branches** of the repo and their fate.

Other useful commands?

```
git reflog ${option} ${reference}
```

- If you think you've messed up your pointers locally, you can run this command to **check if branches or other references changed.**

- A useful option is `show`, which is aliased to a ~*fancy*~ log command.

```
git reflog show
==
git log -g --abbrev-commit --pretty=oneline
```

Other useful commands?

```
git stash
```

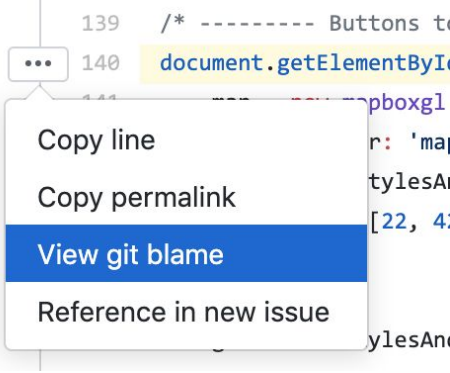
- Uncommitted local changes will be **stored away** for later use.
- **Use when:** you want to save work, but need to switch to a different branch.

- You can get back **the last saved work** by running:

```
git stash pop
```

You can also `list` your previous stashes or even create a `branch`!

Other useful commands?




```
git blame ${option} ${file}
```

- Don't know what some code is doing? Want to ask the developer who wrote it but don't know who they are? Use this command!
- You can also easily do this in the **Github UI**!

- For the command line purest, you can blame code between lines (don't forget the comma between the numbers):

```
git blame -L ${42},${45} ${file}
```

Note: there are a ton of super useful flags for this one!

Other useful commands?

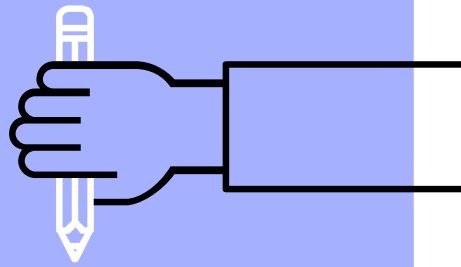
```
git cherry-pick ${option} ${file}
```

- Cherry picking allows you to pick specific commit you want to use and exclude other commits.
- It's meant to clean up your git history when it becomes muddy.

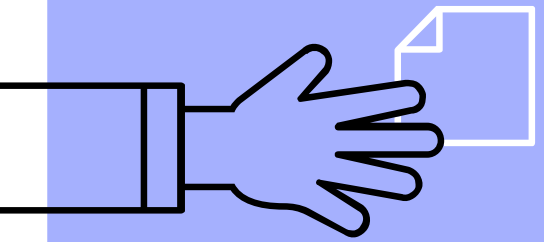
Other useful commands?

```
git diff ${option} ${option}
```

- Allows you to quickly see the differences in commits, working trees, or branches.
- Note: when you do a pull request on a branch, your branch will show a **diff** between itself and a base (most of the time **master**)



3.
Extra Dev Terms



Stuff not covered in the workshop because honestly, this was very long already.

Terminology on-the-job

- **PR:** pull request
  - When you create a branch you want to merge, you **create a PR** so your teammates can do a **code review** on it.
- **Issues:**
  - Bugs and stuff you want developers to know about/fix.
- **Milestones:**
  - Under *Issues* tab. Major aspects of the project that drive development.
- **Tickets:**
  - Small tasks that developers get assigned and can focus on.
- **Slack:**
  - Chat app most tech people use. Tell me or a mentor after the workshop if you're not in the workspace `yyjtech`. We'll get you in.

Terminology on-the-job

- **Cloning a repo:**
  - Copying **the files and code** in a repo.
- **Forking a repo:**
  - Copying **an entire repo** so it exists on your Github account.
  - This allows you to add or propose new features to the creator via a PR.
- **Hash:**
  - In the context of git, it's a unique identifier of an object.
- **HEAD:**
  - Pointer to the current checkout'd out branch or commit.
- **mosaic:**
  - That green grid on the bottom of a users profile.

The best way to learn git is on a team.

Git is great by itself, but it's full power comes out when you're working on a project with other people.

THANKS!

Any questions?

You can find me at:

@k-erby *(Github)*

@kerb *(yyjtech)*