These slides can be viewed in your browser at https://princessruthie.github.io/59X-slides/.

Are they riddled with mistake? Make a PR. Thanks!

Expectations:

Most of you have already seen git and GH in 591. Yay. I expect some of you have forgotten a thing or two. So let's just all git on the same page here.

Git is a program

- git is VCS software that runs on your computer.
  - Do you have git? Check: `which git` If you get crickets, you don't have it. Decide now when you will install it. Maybe tonight after the social?
  - to use git on a project, you must invite the software into your project (I have lots of peanut butter at my house, there is none in my office because I didn't bring it there.)
  - `cd my_super_cool_project` #if you're not already there
  - `git init` #now there's peanut butter in the office.
  - you can now track changes to your project using git.
  - if you're tempted to point out the flaws in my peanut butter metaphor, that's good. It means you're thinking critically.

Git is distributed

- because git is software and servers are computers, you can run git on servers. Don't do it for yourself, let GitHub, Gitlab, BitBucket do it.
- so much software does background synchronization that I'm gonna shout this: **git will not move your changes to/from the server unless you explicitly tell it to**. Sorry. I won't scream anymore.
- we call the server *remote* (because it's far away) and your local copy…we call that *local*.
- to get started on an existing *repository* you must *clone* it.
  - `git clone git@github.com:princessruthie/canvas-submitter.git`
  - and then can `cd canvas-submitter` and run things like `git status` and generally start your life.
- Now you have whatever was at the repo at the time of the clone. You can make changes and track them.
  - Question: are your changes on the remote?
  - Question: are changes to the remote reflected in your local?

Setup Reminders

- If you used git in the past, you can check your email and username:
  - `git config user.name`. Crickets means you have to set it up:
  - `git config --global user.name "Princess Ruthie"`
  - ditto for `user.email` I'm gonna shout again, sorry. **Your email should match your GitHub email or there will be crying and gnashing of teeth**. Actually not really, it can just be annoying and if you care about your GH heat chart thingy, etc. Maybe I have allergies and TMJ.

The Lifecycle:

- Existing files go from unmodified ---> modified ---> staged ---> committed back to unmodified and it starts all over.

  i. modified to staged: `git add path-to-file`
  ii. staged to committed: `git commit -m "blah blah blah"`
  iii. committed to unmodifed: happens when you commit

iv. unmodified to modified: happens when you modify a file

- New files go from non-existing ---> existing ---> staged ---> committed. Then they're on the existing file lifecycle.
- New/existing files can also be(come) ignored by listing them in `.gitignore`
- To move a file from modified to staged you do `git add path-to-file`. Do this until you have everything you want to stage staged.
- To move your staged files to committed, run `git commit -m "tag: Explanation of what exactly the changes are for"` Change the stuff in the double quotes.
- Many commit messages are great as one-sentence summaries. But if you need more, do `git commit` and git will open a text editor (probably vim) and let you write what you need. Remember `i` to go into insert mode and type your text. `ESC` then `:wq` to save your text.
- There's more to it. Files can from from staged back to just modified by running `git reset path-to-file`. Files can go from modified to unmodified by ~~`git checkout path-to-file`~~ This deletes all the changes to the file since last commit, use carefully. Files can go from committed back to staged, committed back to modified, etc.

The Commit model:

A commit has a pointer to a snapshot "tree" and the "tree" has pointers to all the "blobs" (think files) at that commit. Every commit but the first one has a pointer to its parent commit, the commit that came right before. Each commit also includes the username and email of the person who committed it. If I make commit 2a04d then commit f33bc, they'll both have my username and email. f33bc will point to 2a04d as a parent. Where does 2a04d point?

If you accept the commit model, you are ready for branches:

Branching

A branch is a pointer to a commit.

Branching for real

Only read this after you accept the Branching section above. Branching is a way of making independent changes to code. You want to work on the website but don't want to put your changes into production, for example. You make a branch with `git branch cool-feature` or whatever you're going to call it. What have you done? You've made a branch. What's a branch? Right, you've made a pointer to a commit.

Now get your HEAD in the game. HEAD is a pointer to a branch. You can have a ton of different branches but there is only one HEAD. Question, if you only have one branch called master, where does HEAD point? (Answer aloud to yourself.)

Again: HEAD points to a branch and a branch points to a commit. When you branch, you create a new pointer to a commit. To change to that branch, you have to checkout: `git checkout cool-feature`. Stop and say to yourself what you think is happening. Make a sentence using the word HEAD, point and branch. "When you checkout…"

Once you're on a branch, changes there are only on the branch unless you make an explicit effort to bring them over. As you make commits on a branch, `cool-feature` will continue to point to the most recent commit. HEAD will continue to just point to `cool-feature`. To bring changes FROM one branch INTO another, you can merge.

Merging

Changes on different branches know nothing of one another unless you merge. You MERGE INTO the currently checked out branch. So if you have a master branch and a `cool-feature` branch and the master branch has a security update, which one do you checkout? If you're done developing on the `cool-feature` branch and want to bring the changes into production, which branch do you checkout?

1. `git checkout branch-to-merge-into` then
2. `git merge branch-with-new-needed-features`

Often this is super easy and nothing interesting happens. If you get a merge conflict, the format will be:

Merge Conflict

```
<<<<<<< HEAD
Stuff that's modified on the current branch. That is, the one you're merging into.
=======
Stuff that's also modified on the branch you're merging from.
>>>>>>> b
```

There's a million ways to fix this but the point is to remove the lines that start with <<, ==",">> and either pick the stuff before the equals, after the equals or some careful edit of the two.
Once you edit the conflicting file, it has been modified. Think about the lifecycle. What's next? You know this.

When you checkout branch a and merge branch b (merge b into a), a has everything that b has plus the merge commit. But as work continues on a and b, the only way to get these changes over is to explicitly do so. It's not at all automatic.

Remotes

If you joined the project by cloning, then you have a remote already setup. You can check with `git remote -v`. If you see nothing, you sadly don't have a remote set up. You can go to GH and make a new one and don't add any README/license/.gitignore. Then

```
git remote add origin link-to-your-repo
git branch -M main
git push -u origin main
```

Once you have a remote set up, you can push local changes to the remote and pull changes from the remote to local. Similar to merging in that you only send/receive changes if you explicitly ask.

Your team

There are so many ways to flow your work. So many. Here's one.

1. One team member, A, begins by making a public repo on their own GitHub. Click the "initialize with a README" option.
2. Team member A then pastes the contents at this link into the readme.
   starter readme for hackathon submission
3. Team member A shares the repo link with the entire team.
4. As it is, only A can push to the repo. It's a hackathon, be wild. A can give everyone push permissions. It's under settings ---> manage access.
5. Team members A through Z all clone the repo.
6. Members make their changes. Push/pull them. Resolve conflicts. Build character and team spirit. Seriously.
7. Going forward, each member can branch from master and make their changes in peace. When an entire logical block of code is done, merge into master and push. You'll still get a few conflicts here and there but you did the previous step and

can handle it.

i. `git checkout master && git pull` # start with latest version of master
ii. `git checkout -b feat-5` This is where you'll be doing work
iii. `git push --set-upstream origin feat-5` only need to do it this way once.
iv. use the lifecycle as above and push regularly.
v. When you've finished a solid feature, pull master for latest teammate changes, merge feat-5 into master and push it up. Start on the next feature.
    a. `git checkout master`
    b. `git pull`
    c. `git merge feat-5`
    d. `git push`
    e. `git checkout -b feat-12`
    f. etc.

That's it. You now know everything ever about git and GH.