

IN CASE OF FIRE



1. git commit



2. git push



3. git out!

3. Git

Logistics

- **First day of class?** Welcome! Review Lecture 1, join the Discord, catch up on HW0/1/2 on iCollege, and email me if you have more questions.
- **Homework 3** is due by the start of next class - you'll be getting your development environment and GitHub accounts set up!
- **Finalized syllabus with office hours** was posted! I just updated the existing link on iCollege. Office hours are Tuesday/Friday 12-2pm!

Logistics

- **Homework 4 will involve working with your project group, so this is the week to set those!**
 - Technically, you COULD figure out your project group during the week you're supposed to work on homework 4.
- **Here's what I've got in mind:**
 - I'll make a channel in the Discord for group-seeking
 - If you've got a technology you really want to learn or use, you can post and try to get more people on board.
 - If you get a group of 4 (or you're the first group of 3), tag me in a post announcing your group.
 - If I don't hear from you by Thursday night, I'll make a group for you (using the requested partners from HW1)

Agenda









1. Intro to Git
2. Basic Git Workflow
3. Git Branching



Why do we care? 🤔

- Version control is used EVERYWHERE
 - And Git will be **the only** way to turn in most assignments in this class 😁

Who has seen something like this?

Name ↑	Owner
 **** FINAL PAPER 1	me
 Copy of Paper 1 Draft	me
 Final Paper 1	me
 Paper 1	me
 Paper 1 Draft	me
 Paper 1 Draft 2	me
 Paper 1 Final	me
 Paper 1 Final Draft	me

Version Control

or Source Control (for code)

...an organized way of thinking about and keeping track of changes of files.

- Which changes were made?
- Who made the changes?
- When were the changes made?
- Why were changes needed?

Examples of Version Control Implementations



Git



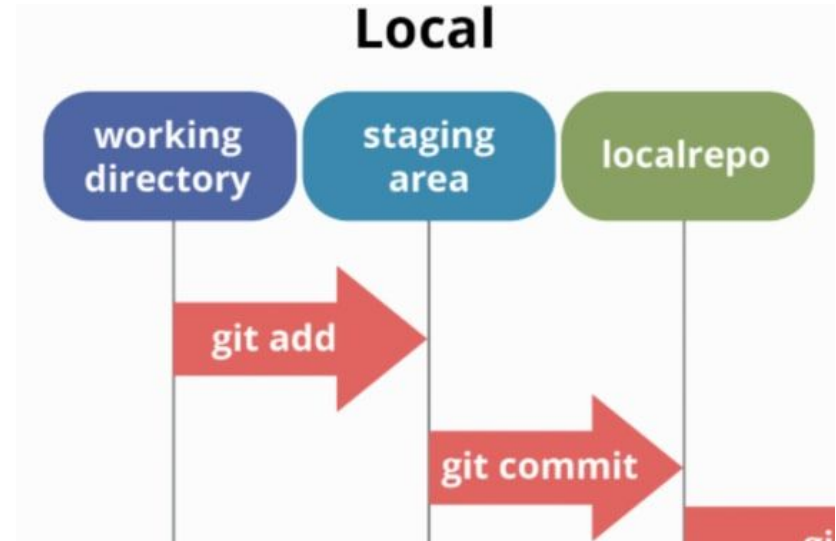
- More than 70% of developers use it
- Manage updates to a project via commands
- Is not really simple to use
 - Especially if you don't understand how it works
 - Even if you understand the commands

Terminology

- **Repository** (repo): all files/folders of a project (and its history)
 - Starts empty, slowly grows
- **Commit**: a unit of change (for 1+ files) in the repo; add, remove, edit files
- **Branch**: branch from the main line of development (i.e. to work on a specific feature without messing up the main)
- **Tracked/Untracked File**: whether your Version Control acknowledges a file is a part of the repo and is following its changes

Diagram of a Local Repository

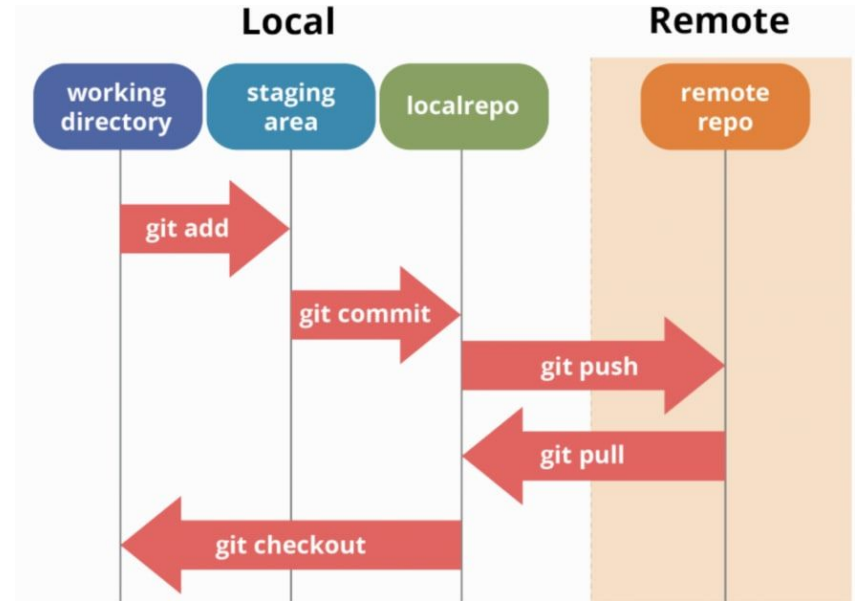
- The **working directory** contains all the files that Git knows exist (tracked files), but have not been staged.
- Files are pushed to the **staging area** when a user tells Git to stage the file (via `git add`).
- Once you **commit** the change (AKA officially record the change, it is saved to your local repo!



Local vs. Remote Repos

A **local** repository sits in your local directory on your computer. Others can't access your code and its history.

A **remote** repository sits on a separate server (usually). Think of it as backing up your version control project to the Cloud so that others can access it and work with it.



Quick note on CLI vs. GUI Tools

- (CLI = Command line)
- You can use things like GitHub Desktop and VSCode extensions to bypass having to learn a lot of these commands.
- I'm going to focus on CLI ways of doing things because

Single-Player Git

Locally - Setting up Git

1. cd into your project directory
2. Create a (local) repository: **git init**
3. Check what files and changes git is tracking/staging: **git status**
4. Prepare files for commit (stage them): **git add {filenames}** or **git add -a** (to stage all changes)
5. Officially save the changes for the files you chose as a commit to the project history: **git commit -m "Description of changes"**

Locally - Basic Git Flow (without branching)

1. Change (create, delete, edit) files.
2. Check what files and changes git is tracking/staging: **git status**
3. Prepare files for commit (stage them): **git add {filename1} {filename2}** or **git add -a** (to stage all changes)
4. Officially save the changes for the files you chose as a commit to the project history: **git commit -m “Description of changes”**
5. Repeat Steps 1-4. **Commit often and in smaller chunks.**
 - a. At Facebook Meta, you’re generally not supposed to commit anything with > 200 lines of code changes.

Where does Github fit in?

- UI platform
- Store (backup) and access remote repositories
- Easily collaborate and share code



From Local -> Remote (Github) - New Repo

1. `git add {filename1}`
2. `git commit -m "Description of changes"`
3. Connect your local to remote: `git remote add origin {url}`
 - a. You can find the url on your Github repo
4. `git branch -M main`
 - a. This just renames your current branch from "master" to "main." Not required, but most GitHub docs assume that your starting branch is called "main," so it may make your life easier.
5. `git push -u origin main`
 - a. Where origin is your remote repo, main is the branch

Quick check: is committing safe?

You've worked with Git for months and now it's pure muscle memory to do stuff. Without even thinking about it, you make some changes to the files you're working on, then run ``git commit -m "stuff"`. What happens?`

- 1) Nothing
- 2) Your changes are saved locally, but not on GitHub
- 3) Your changes are saved locally AND on GitHub

Quick check: is committing safe?

You've worked with Git for months and now it's pure muscle memory to do stuff. Without even thinking about it, you make some changes to the files you're working on, then run ``git commit -m "stuff"``. What happens?

1) Nothing

2) Your changes are saved locally, but not on GitHub

3) Your changes are saved locally AND on GitHub

You didn't run ``git add``.

Quick check: is committing safe?

You've worked with Git for months and now it's pure muscle memory to do stuff. Without even thinking about it, you make some changes to the files you're working on, then run ``git add -a; git commit -m "stuff"`. What happens?`

- 1) Nothing
- 2) Your changes are saved locally, but not on GitHub
- 3) Your changes are saved locally AND on GitHub

Quick check: is committing safe?

You've worked with Git for months and now it's pure muscle memory to do stuff. Without even thinking about it, you make some changes to the files you're working on, then run ``git add -a; git commit -m "stuff"`. What happens?`

- 1) Nothing
- 2) Your changes are saved locally, but not on GitHub
- 3) Your changes are saved locally AND on GitHub

Big idea: don't be afraid of commitment!

You're not going to break anything by running the commit command. You can always do commit history surgery to fix things to your liking.

Review: Linux Cheat Sheet

Command	Keyboard Shortcut
List current file path	pwd
List all files in current path	ls
Navigate to subdirectory	cd directory_name
Navigate to parent directory of current path	cd ..
Print "hello world" to terminal	echo "hello world"
Create new file in current path	touch file_name
Open a file in the default text editor	open file_name
Delete/remove a file	rm file_name
Delete/remove a directory	rm -r directory_name
Print first 15 lines of file	head -n 15 file_name
Print last 15 lines of file	tail -n 15 file_name
Append "foo" to file "bar.txt"	echo "foo" »bar.txt
Overwrite file "bar.txt" to say "foo"	echo "foo" >bar.txt
Stop current command	Ctrl + c
Display line numbers while printing contents "bar.txt"	cat -n bar.txt
Read "bar.txt" in interactive read-only view	less bar.txt
Quit read-only view from less	q
(Really useful!) Find previous command you ran	Ctrl + r

Multiplayer Git

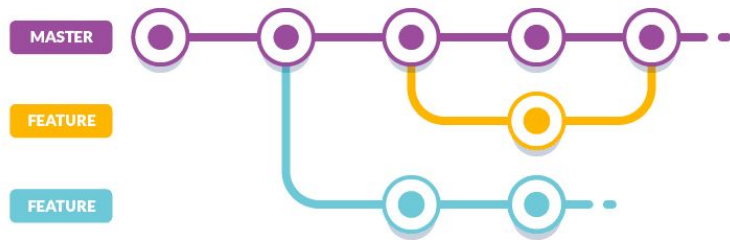
Choose your fighter!

You work for a company that builds a web app that people can use to stream music. You have thousands of people using your site every day. When you make a change to your code, do you:

- Immediately roll it out to everyone?
- Wait for someone else to approve the code for release?
- Release all changes at some fixed cadence (e.g. every week)?
- Other?

Git Branching 🌳

- Branching is a way to **separate your concerns** on source control. It can have **multiple commits**
- When you create a new repository by running `git init`, a default branch called master (main in newer versions, apparently) will be created.
- The main branch should always be **stable** and **bug-free**, and never used for feature development!



Why do we care? 🤔

Why are we doing this instead of just yolo pushing to main and dealing with main having bugs?

Why do we care? 🤔

- In industry...
 - Main is the branch that goes out to people! So if you just have random bugs all the time in the main branch, you have no way to make stable software that works for everybody.
- In our class...
 - You'll be doing a group project later. If you screw up the code for everybody in your team, people will be upset and it will be counterproductive and waste lots of time. Have a stable main!!

Key Commands

- `git checkout feature` - switch to branch “feature”
- `git branch feature` - create new branch called “feature”
 - `git checkout -b feature` - create AND switch to new branch “feature”
- `git branch -d feature` - delete branch “feature”
- `git merge feature` - merge branch “feature” into current branch

Remote Git Flow (so far...)

1. (only once) Create a local repo: `git init`
2. (only once) Connect local to remote repo: `git remote add origin {url}`
3. Change (create, delete, edit) files.
4. Check what files and changes git is tracking: `git status`
5. Prepare files for commit: `git add {filename1} {filename2}` or `git add -A` (to add all files)
6. Officially save those changes in the project history: `git commit -m "Description of changes"`
7. *Repeat Steps 1-4. **Commit often and in smaller chunks.***
8. Add your changes to your remote repo: `git push origin main`

The Right Way (Local Merging)

1. Create a new branch for a feature: `git branch {branch_name}`
2. Switch to new branch: `git checkout {branch_name}`
3. Change (create, delete, edit) files (just for that feature).
4. Check what files and changes git is tracking: `git status`
5. Prepare files for commit: `git add {filename1} {filename2}` or `git add -A`
6. Officially save those changes in the project history: `git commit -m "Description of changes"`
7. Switch back to main, get the latest remote changes: `git checkout main && git pull origin main`
8. Reconcile changes in main w/ your branch: `git merge {branch_name}`
 - a. If conflict: fix merge conflicts in files
 - b. Run `git status`, follow instructions to add and commit to fully resolve.
9. Add your changes to your remote repo: `git push origin main`

It's demo time! 

Branches

The Right Way - MERGE CONFLICT!?

1. Create a new branch for a feature: `git branch {branch_name}`
2. Switch to new branch: `git checkout {branch_name}`
3. Change (create, delete, edit) files (just for that feature).
4. Check what files and changes git is tracking: `git status`
5. Prepare files for commit: `git add {filename1} {filename2}` or `git add -A`
6. Officially save those changes in the project history: `git commit -m "Description of changes"`
7. Switch back to main, get the latest remote changes: `git checkout main && git pull origin main`
8. Reconcile changes in main w/ your branch: `git merge {branch_name}`
 - a. If conflict: fix merge conflicts in files
 - b. Run `git status`, follow instructions to add and commit to fully resolve.
9. Get the latest remote changes: `git pull origin main`
10. Add your changes to your remote repo: `git push origin main`

Merge conflicts 🤯

- Merge conflicts occur when there are collisions between the state of your local repo and the changes that you're making to the repo. Most of the time, git figures it out by itself, but when it's too complicated, you have to manually tell Git what to do!
- What changes could you make that could result in merge conflicts?
 - `git pull` after someone else pushes code to the repo.
 - `git merge <feature_branch>` that touches the same code as the branch you're currently on.

Resolving merge conflicts 🤝

- Merge conflicts look something like this. In this case, we got a merge conflict when merging the branch `better_print_statements` into `HEAD`.
- The area between `<<<<` and `===` is the changes made in `HEAD` (i.e. the current state of your repo). The area between `===` and `>>>>` is the changes made in your branch (called `better_print_statements`)
- You resolve merge conflicts by deleting the `<<<<`, `===`, and `>>>>` indicators, and figuring out which change you want to take.

```
<<<<<<< HEAD
    print("This way!")
=====
    print("That way!")
>>>>>>>>
better_print_statements
```

Resolving merge conflicts (Continued)

For example, if we want to take the changes from the branch, we would delete the following lines:

```

<<<<<<<<< HEAD
    print("This way!")
=====
    print("That way!")
>>>>>>>>>
    better_print_statements

```

Resolving merge conflicts (Continued)

On the other hand, if we want to keep the local changes and abandon the changes in the branch, we might do something

```
<<<<<<< HEAD  
    print("This way!")  
=====  
    print("That way!")  
>>>>>>>>>  
better_print_statements
```

Resolving merge conflicts (Continued)

Lastly, we can decide to take elements of both changes and combine them. In this case, we could do the following, where we take the capitalization from HEAD and the grammar from better_print_statements

```
<<<<<<< HEAD  
    print("This way!")  
=====  
    print("That way!")  
>>>>>>>>  
better_print_statements  
  
    print("Neither way!")
```

Resolving merge conflicts (Continued)

So which solution is best?

- Accept local?
- Accept remote?
- Combine them together?

Resolving merge conflicts (Continued)

Merge conflicts are completely different each time, and no merge strategy is always the right answer. The correct merge strategy depends on the situation and your goals!

It's demo time! 

Branches

From Last Time: Ok... but how do I apply?

- “Fight the horse-sized duck”
- (It is really convenient to mass apply online but that should be a LAST resort)



Until next time... 🙌

- Have a great weekend!
 - I guess I'm just going to say this every lecture?
- Next lecture we'll introduce Pull Requests
- Go on #teammate-search if you want to round up some teammates!