A Practical Approach to Git

Upsilon Pi Epsilon

Git Experience?

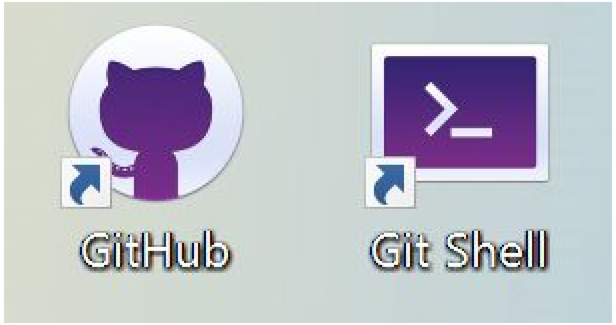
Part 1

Setting Up Git - Linux

- Package Manager

Setting Up Git - Windows

- Download Git for Windows: https://desktop.github.com/
- Open the git shell



Setting Up Git - Mac

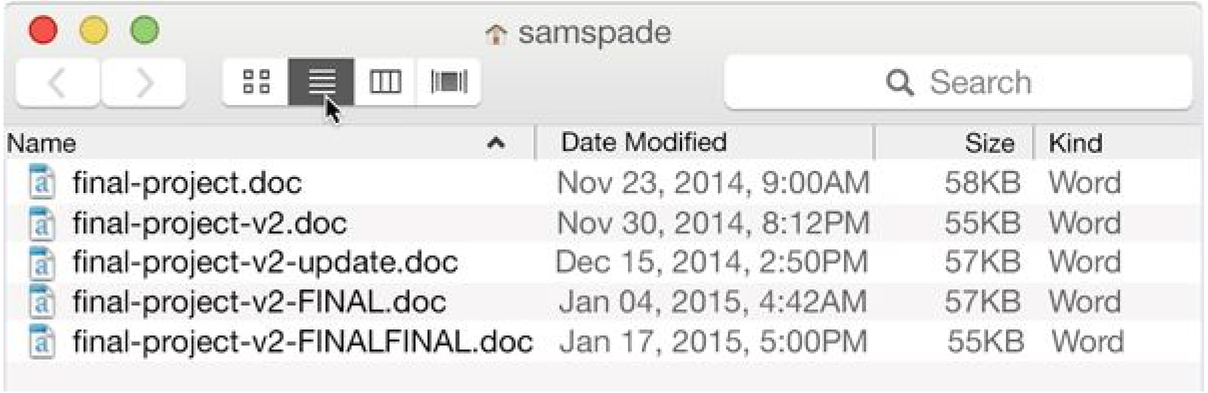
- Download Git for OS X: https://desktop.github.com/
- Use in Mac terminal

Set Globals

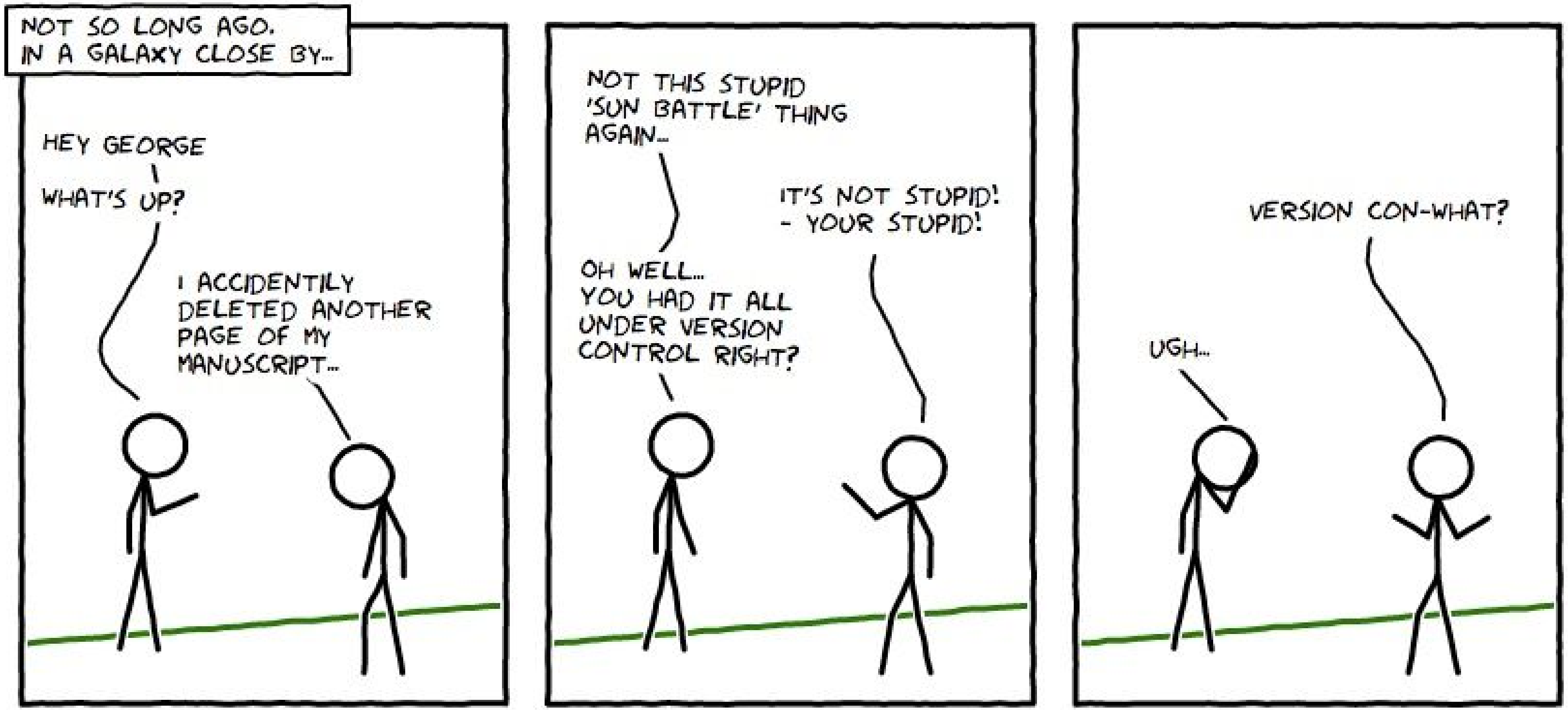
- Git Command: `git config --global user.<var> <arg>`
    - This command allows you to configure global variables in git.
- Do this:
    - `git config --global user.name "John Doe"`
    - `git config --global user.email johndoe@example.com`
    - `git config --list`

What is version control?

- Tracks files and changes
- Easily go back to previous versions
- Views a log of changes
- Coding on a team
- Make new changes while maintaining working copies
- Prevents this:



Name	Date Modified	Size	Kind
final-project.doc	Nov 23, 2014, 9:00AM	58KB	Word
final-project-v2.doc	Nov 30, 2014, 8:12PM	55KB	Word
final-project-v2-update.doc	Dec 15, 2014, 2:50PM	57KB	Word
final-project-v2-FINAL.doc	Jan 04, 2015, 4:42AM	57KB	Word
final-project-v2-FINALFINAL.doc	Jan 17, 2015, 5:00PM	55KB	Word


NOT SO LONG AGO. IN A GALAXY CLOSE BY...
HEY GEORGE
WHAT'S UP?
I ACCIDENTILY DELETED ANOTHER PAGE OF MY MANUSCRIPT...
NOT THIS STUPID 'SUN BATTLE' THING AGAIN...
OH WELL... YOU HAD IT ALL UNDER VERSION CONTROL RIGHT?
IT'S NOT STUPID! - YOUR STUPID!
UGH...
VERSION CON-WHAT?

Why Git?

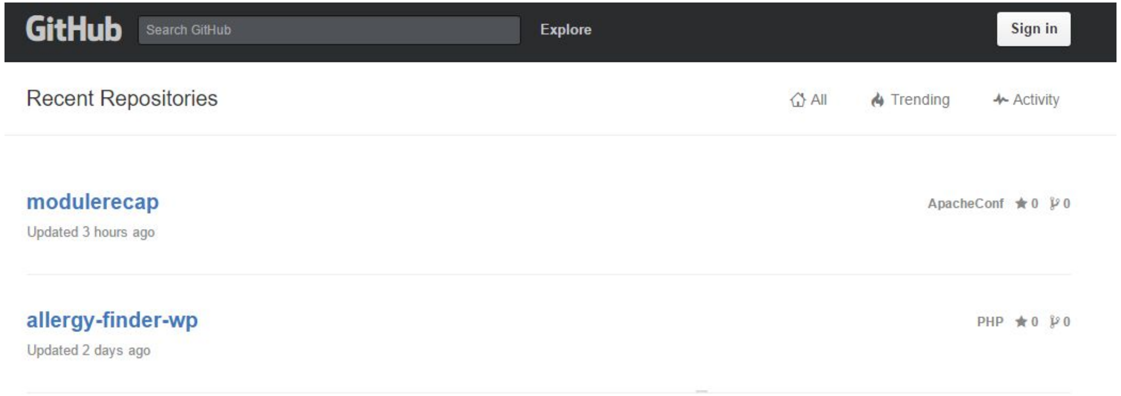
- Distributed system (local and remote copy)
- Cheap and easy branching
- One of the most popular
- Ease of use
- Open Source
- Used at Uconn
- Created by Linus Torvalds

What is a repository?

- Where the project lives
- Place where work, and history of work, is stored

UConn GitHub

- Using GitHub
- UConn provides all students with GitHub accounts
- https://github.uconn.edu/



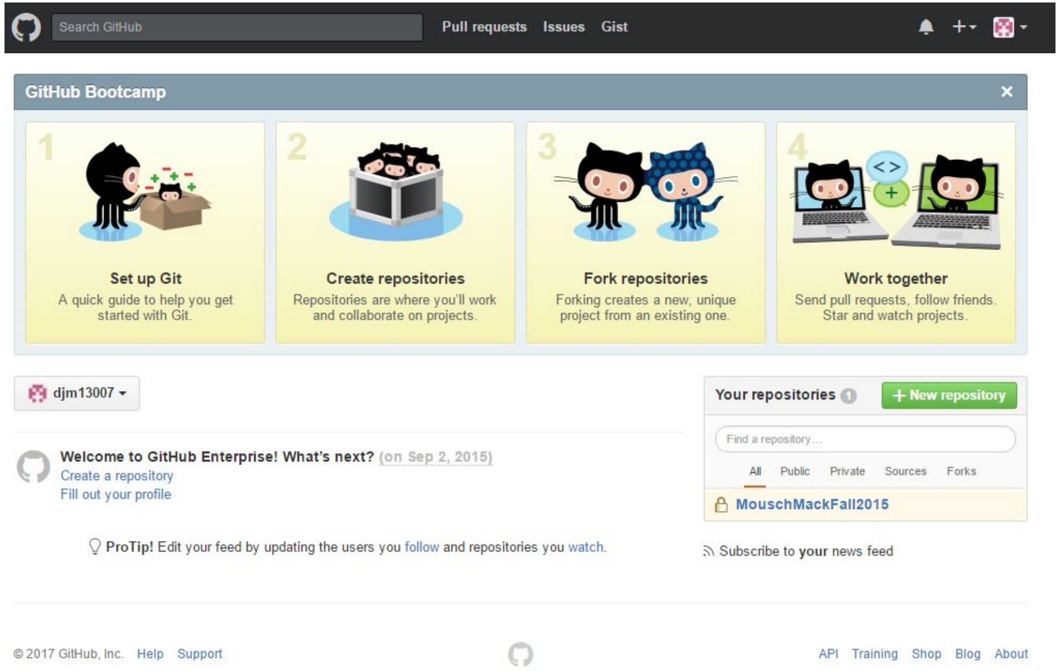
Recent Repositories

moduleconf
Updated 3 hours ago
ApacheConf ★ 0 0

allergy-finder-wp
Updated 2 days ago
PHP ★ 0 0

User Dashboard

- After logging in click the icon in the top left corner
- Shows your repositories
- Click the "New Repository" button to create a new repository.



A quick guide to help you get started with Git.

Repositories are where you'll work and collaborate on projects.

Forking creates a new, unique project from an existing one.

Send pull requests, follow friends. Star and watch projects.

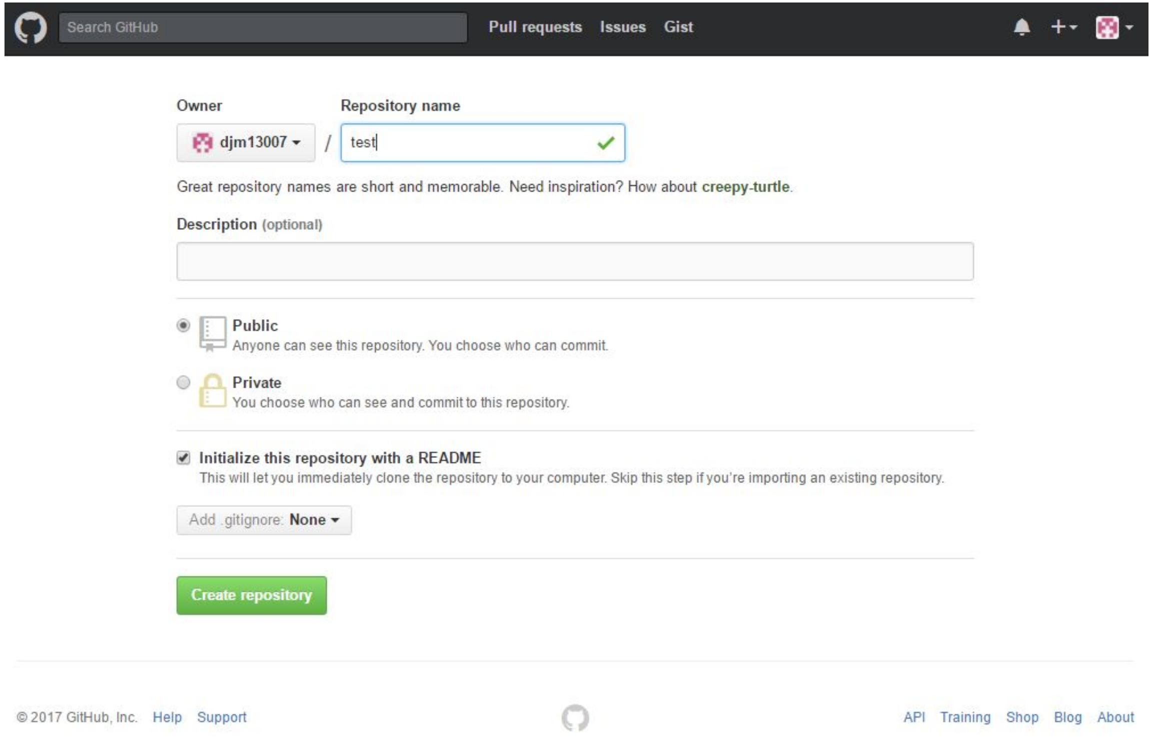
Welcome to GitHub Enterprise! What's next? (on Sep 2, 2015)

ProTip! Edit your feed by updating the users you follow and repositories you watch.

© 2017 GitHub, Inc.

Making a repository

- Set repository name and description
- Set visibility settings
- Create a README in the new repository



Owner: djm13007 / Repository name: test ✓

Great repository names are short and memorable. Need inspiration? How about **creepy-turtle**.

Description (optional)

[✓ Public] Anyone can see this repository. You choose who can commit.

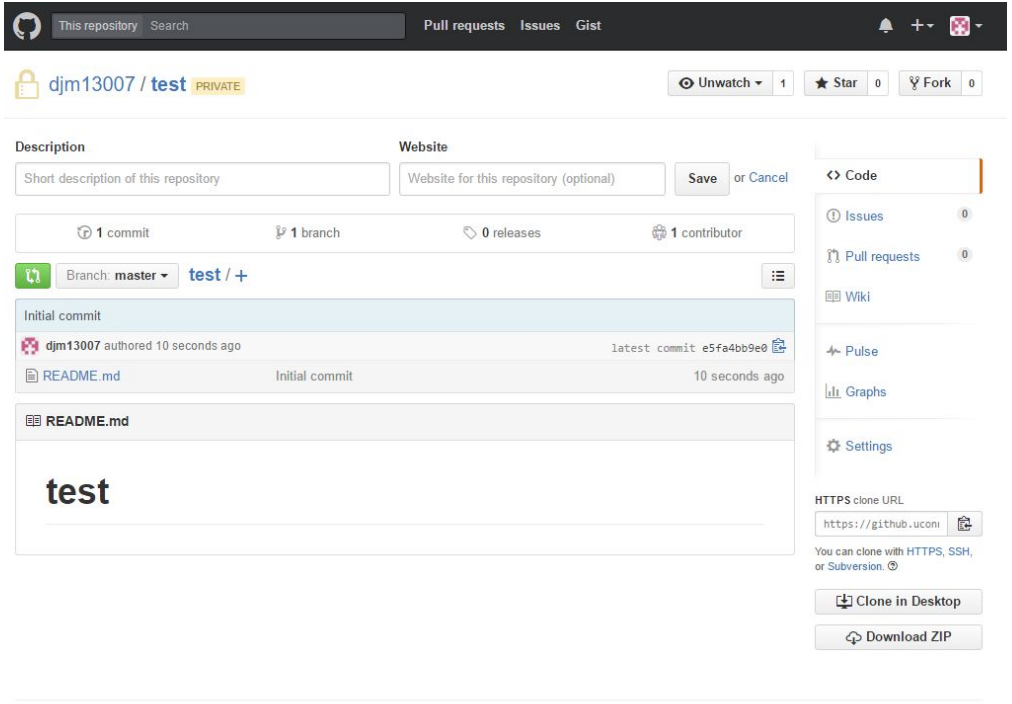
Private — You choose who can see and commit to this repository.

[✓] Initialize this repository with a README
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: **None**

Success

- The repository was created
- Contains only a README with the name of the repository
- Clone URL



Description

Short description of this repository

Website

Website for this repository (optional)

Save or Cancel

1 commit   1 branch   0 releases   1 contributor

Branch: master   test / +

Initial commit

djm13007 authored 10 seconds ago   latest commit e5fa4bb9e0

README.md   Initial commit   10 seconds ago

README.md

test

HTTPS clone URL

https://github.uconn

You can clone with HTTPS, SSH, or Subversion.

The Structure of Git

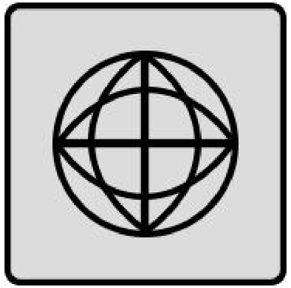
- There are four main sections to a git repository:
    a. Working Directory
    b. Staging Area (Index)
    c. Local Repository
    d. Remote Repository

The Structure of Git

- Three of these areas reside on your computer
    a. Working Directory
    b. Staging Area (Index)
    c. Local Repository
- Only one area resides outside your computer
    a. Remote Repository

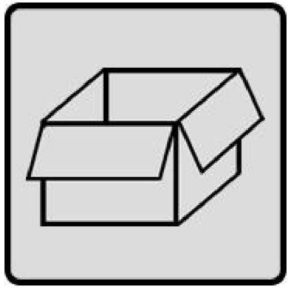
The Structure of Git: The Remote Repository

- This is the external storage of your repository.
- Here, all of the data that git needs to maintain your repository is stored.
- Everyone else can see a remote repository (if you let them).
- We just made a remote repository on GitHub.



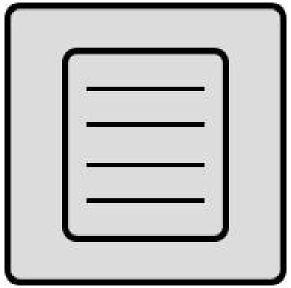
The Structure of Git: The Local Repository

- This is your local copy of the Remote Repository.
- This is where git stores all of the data it needs to maintain your repository, on your computer.
- We will be making one of these shortly.



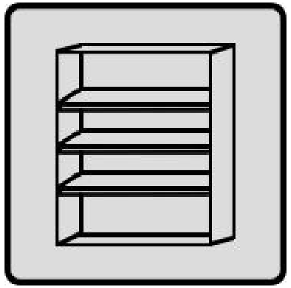
The Structure of Git: The Working Directory

- This is where the working copy of all files in your Local Repository are located.
- Editing files here does not update the files in your Local Repository.

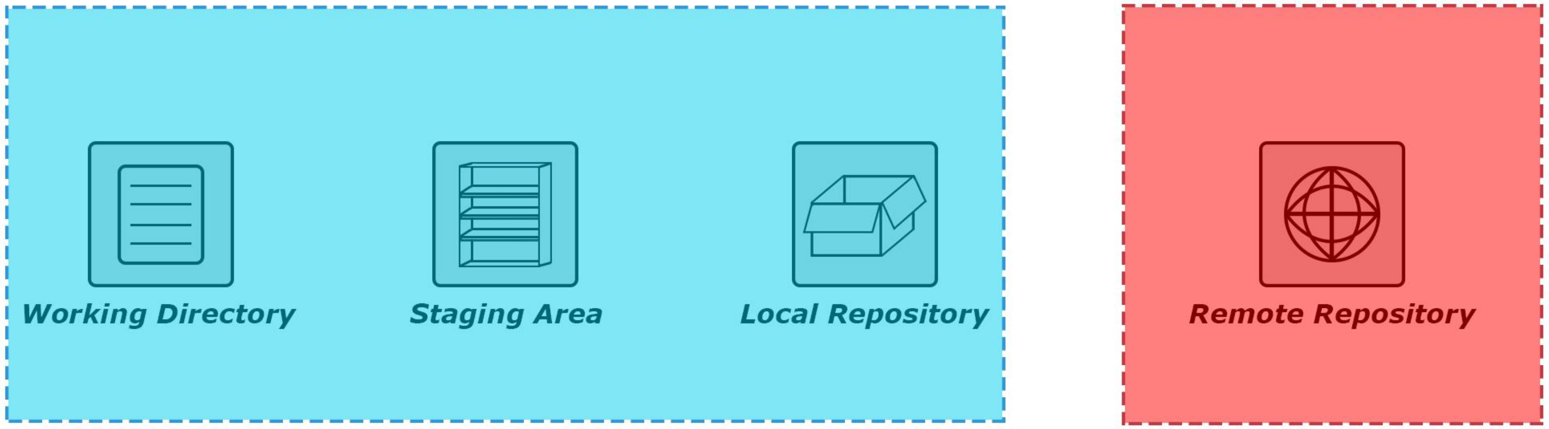


The Structure of Git: The Staging Area (Index)

- The Staging Area (Index) is an intermediate zone between the Working Directory and the Local Repository.
- To update your Local Repository, you must put files here first.



The Structure of Git: Overview



Git Clone

- Git Command: `git clone <URL>`
  - The git clone command copies the contents of the remote repository to a local git repository in a new directory on your computer
  - The URL is the URL related to your remote repository

Git the Goods (Cloning your repository)

- Do this:
  1. Navigate (in terminal) to where you want your repository located
  2. Copy URL from git repository web page
  3. In terminal: `git clone <URL>`
  4. `cd <repository name>`

Making your first change

- Do this:
    1. Open your README using your favorite text editor (Note: If your favorite text editor isn’t vim, it should be)
    2. Add your name (or some text) to the README
    3. Save the file

Checking your status

- Because git tracks the state of the files in the local repository, we can ask git to tell us the status of all these files
- Do this:
  1. `git status`

Breaking down `git status`

- `On branch master`
    - We'll get to this later (don't worry, it's easy)

Breaking down `git status`

- `Changes not staged for commit` followed by `modified: README.md`
    - These are files in the **working directory** that are not in the **staging area**
        - Tracked files that are modified but not ready for **commit**
- Recall the definitions of these terms
    - You already know the **repository** (stores your files and metadata about the files)
    - **Working directory**: This is where the working copy of all files in your Local Repository are located
        - Your `README` is being tracked by git, and git recognizes that there are changes to this file
    - **Staging area**: An intermediate zone between the Working Directory and the Local Repository
    - **Commit**: A snapshot of your code at a particular point in time

How do we tell Git to keep these changes?

1. Move files to the **staging area** (also called the **index**)
    - The **staging area** is how you can tell git you want to use the current state of the file (as it was when it was added to the staging area)
    - There can be multiple files in the staging area, but not all modified files need be in the staging area
2. Tell git to take all files in the staging area and create a new snapshot of your code base

The `add` command

- Our README is a part of the working directory, but our modifications have not been added to the index yet
  - Nothing in the index (nothing to snapshot)
- If we want to add something to the index, we use the `git add` command
- Do this:
  1. `git add README.md`
- Now, the README has been moved to the index/staging area

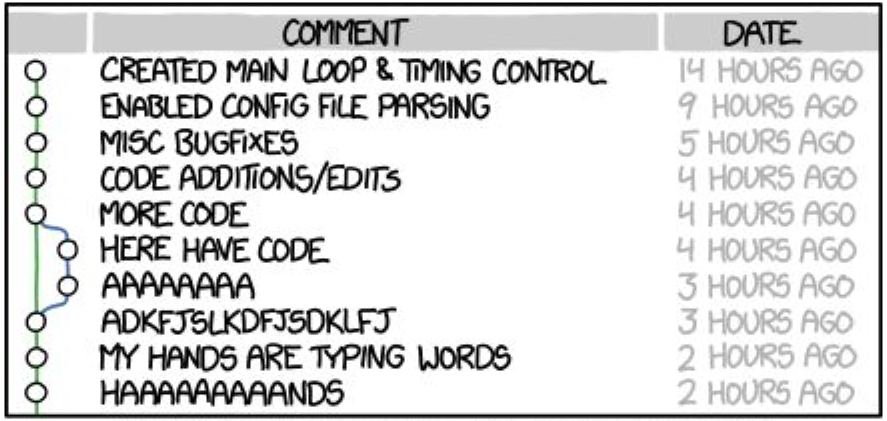
Checking your status (again)

- Do this:
    1. `git status`
- You can now see that we have changes that are going to be committed (pretty colors too!)
- Our README has been added to the index, and is now staged for commit
    1. Reminder: a **commit** is just a snapshot of a particular version of the code

Make a commit-ment

- Git Command: `git commit`
    - Create a new commit (snapshot) based on the contents of the staging area
- We have our README staged, and now we want to commit it
- There are some helpful guidelines on commit messages
    1. Format
        i. Short description (less than 50 characters)
        ii. Blank line
        iii. A longer description that describes the changes in more detail
    2. The description should be in the imperative tense (Change README; Fix bug XYZ)
    3. Consistency is key
- The commit message allows you to go back months from now and see what changes you made for a given commit

Make a commit-ment



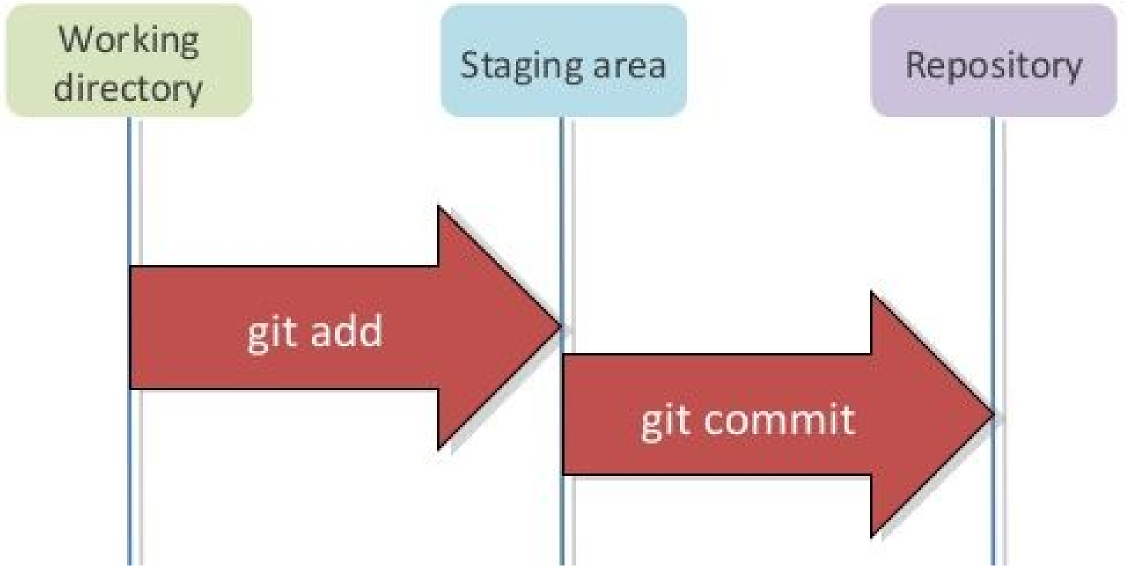
	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS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

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

Make a commit-ment

- Do this (one or the other):
  1. `git commit`
     i. Opens up a text editor and allow you to type a message
  2. `git commit -m` *“INSERT YOUR MESSAGE HERE”*
     i. Allow you to type a message on the command line (can be more than one line)

Some vocab explained (again)



C-c-c-c-changes (face the strange)

- Let’s run `git status` again!
- We’ve updated our local repository!
    - Let’s look at GitHub!

Push It To the Limit

- Git Command: `git push`
  - Send your local branch to the remote repository (we'll explain branches soon, don't worry!)
- We need to send our local changes to the remote repository
- The quick and dirty: send your code to GitHub
- Do this:
  1. `git push origin master`
     - I swear, the definitions for these things are coming soon
  2. Now check out GitHub! (refresh the page)

End of Part 1. Questions?

Part 2

The reset command

- Git Command: `git reset`
- The reset command allows us to undo changes we have made to the repository on our own computer.
- There are three important tasks we can do with the reset command:
  1. Undo the last commit we **did not** push
  2. Unstage files
  3. Undo all modifications since the last commit

The `reset` command - Setting the stage

- Do this:
    1. Create three text files: a.txt, b.txt, and c.txt
    2. `git add .`
        a. This will stage all new and changed files in the current directory
        b. Note: this is not always a good idea
    3. `git status`
    4. `git commit -m "This is my second commit."`
    5. `git status`
    6. `git push origin master`
    7. Open your README file
    8. Add the line "My third commit." to the README and save
    9. `git add README.md`
    10. `git commit -m "This is my third commit."`

The reset command - Undoing the last local commit

- Git Command: `git reset --soft HEAD~`
  - `--soft` makes sure that we only update the local repository (we don't touch the staging area or working directory)
  - `HEAD~` is the reference to the previous commit
- Do this:
  1. `git reset --soft HEAD~`
  2. `git status`
- We can see that README.md now shows up again

The reset command - Unstaging files

- Git Command: `git reset`
    - Running git reset without any other options will unstage all of the current files
    - You can also specify one or more filenames after `git reset` to unstage individual files
- Do this:
    1. Make a change to a.txt, b.txt, and c.txt
    2. `git status`
    3. `git add .`
    4. `git status`
    5. `git reset a.txt`
    6. `git reset b.txt`
    7. `git status`
    8. `git reset`
    9. `git status`

The reset command - Undo all modifications

- Git Command: `git reset --hard`
    - `--hard` will make the reset command override our Working Directory
    - **Be careful with this command!** You will be throwing away all modifications you have made in the Working Directory.
    - You cannot perform `git reset --hard` on individual files.
- Do this:
    1. `git add .`
    2. `git reset --hard`
    3. `git status`
    4. Open `a.txt` (you'll see that the change you made is gone!)

Let's checkout the working directory

- Typing `git reset --hard` every time we want to remove changes in the working directory is kind of a lot.
- It also doesn't let us undo changes to a specific file.
- Luckily, there is another command that will let us do just that

Let's checkout the working directory

- Git Command: `git checkout <filenames/paths>`
  - This command will make `<filenames/paths>` look like it does in the staging area.
    - Essentially, this will move the files from the staging area and put them in the working directory
    - Note: This means that if you have not added your changes to the staging area for a specified file(s), all changes in the working directory will be removed for those files
  - Usually, this means your files will be reverted to the states they were in during the last commit.
    - But if you have already staged the file, you must first unstage it to revert the changes.

Let's checkout the working directory

- Do this:
  1. Make a change to a.txt, b.txt, c.txt, and README.md
  2. `git add README.md`
  3. `git status`
  4. `git checkout a.txt b.txt`
  5. `git status`
  6. Make another change to a.txt and b.txt
  7. `git status`
  8. `git checkout .`
  9. `git status`
  10. `git reset README.md`
  11. `git checkout README.md`

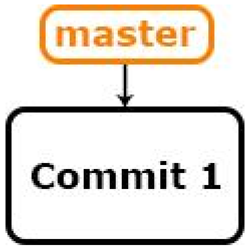
Branching - What is a branch?

- A branch is a named pointer to a commit.
- When you first made your repository, a single branch was created, which is named "master" by default.
- All the work we have done so far has been on the "master" branch.
- When you make a commit, the branch you are currently on will update itself to point to the most recent commit.

Branching - How branches work

- When you make a commit, the branch you are currently on will update itself to point to the most recent commit.

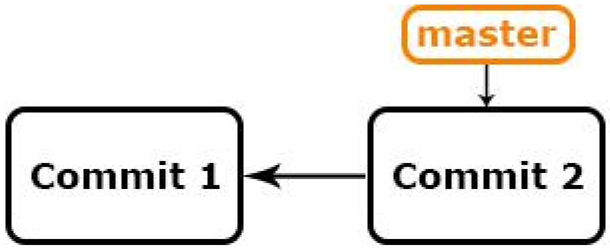
Current Branch: **master**



Branching - How branches work

- When you make a commit, the branch you are currently on will update itself to point to the most recent commit.

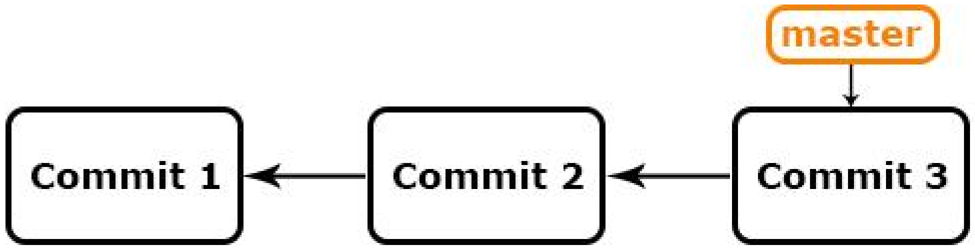
Current Branch: **master**



Branching - How branches work

- When you make a commit, the branch you are currently on will update itself to point to the most recent commit.

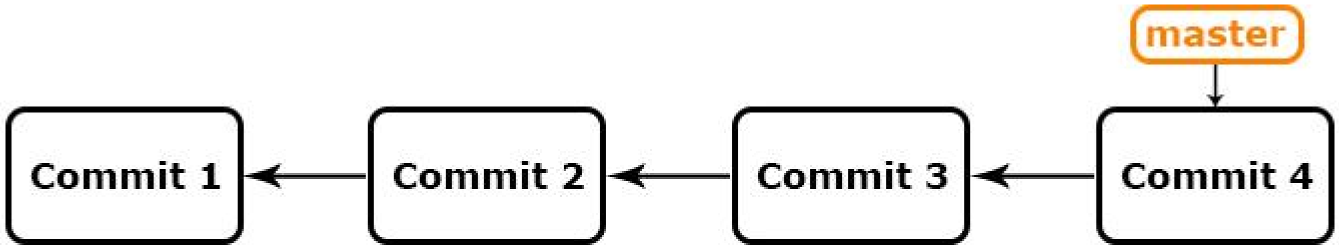
Current Branch: **master**



Branching - How branches work

- When you make a commit, the branch you are currently on will update itself to point to the most recent commit.

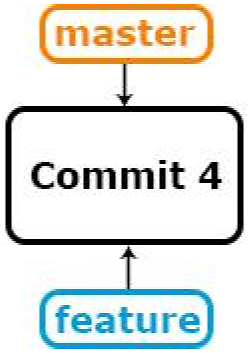
Current Branch: master



Branching - How branches work

- When you make another branch, another pointer to the current commit is made.

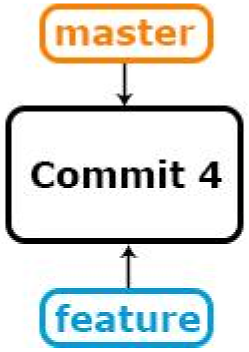
Current Branch: **master**



Branching - How branches work

- When you checkout (switch) to a different branch, that branch gets updated when you make new commits.

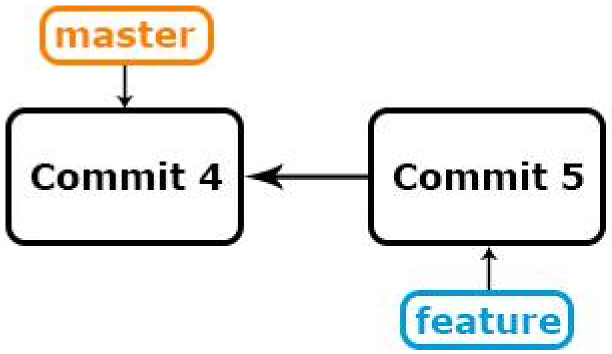
Current Branch: **feature**



Branching - How branches work

- When you checkout (switch) to a different branch, that branch gets updated when you make new commits.

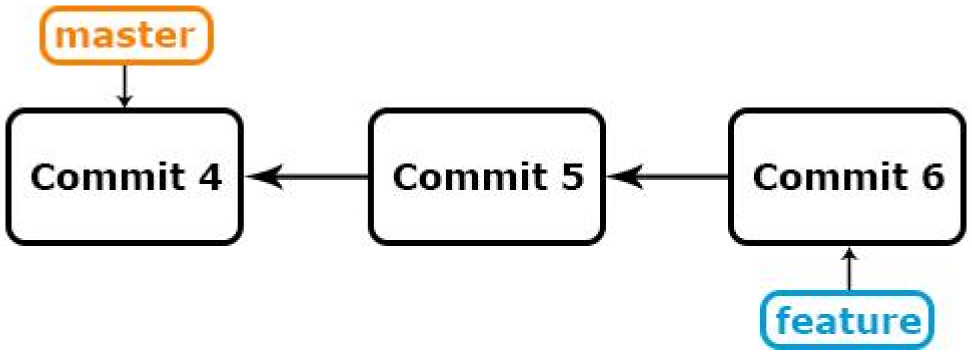
Current Branch: **feature**



Branching - How branches work

- When you checkout (switch) to a different branch, that branch gets updated when you make new commits.

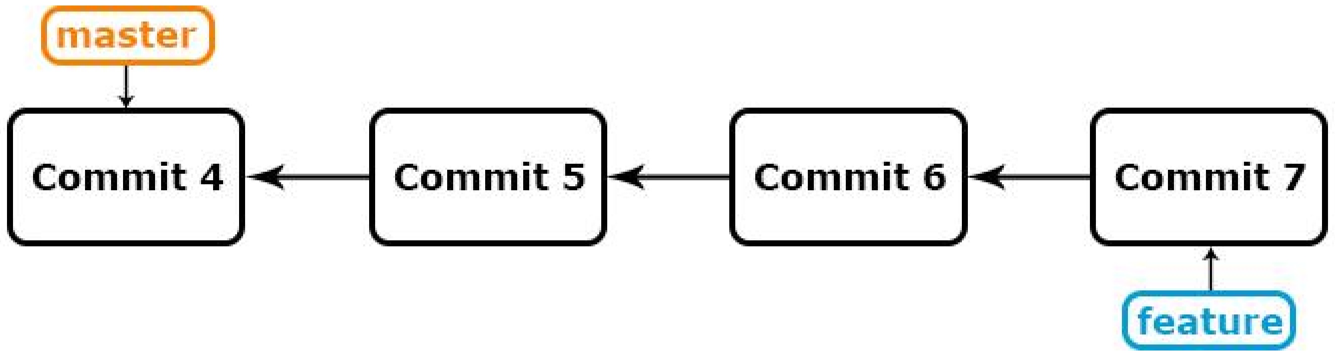
Current Branch: **feature**



Branching - How branches work

- When you checkout (switch) to a different branch, that branch gets updated when you make new commits.

Current Branch: **feature**



Branching - Why make branches?

- Branching allows you to work in a secluded and safe environment.
- It's harder to mess up the main body of the code when you're not working on the "master" branch.
- You can experiment with new ideas without the fear of making permanent changes.

Branching - Viewing our branches

- Git Command: `git branch`
  - This command lists all of the current branches
  - The current branch will have a * next to it
- Do this:
  1. `git branch`
- You should see:
  - `* master`

Branching - Making a new (local) branch

- Git Command: `git branch <branch>`
  - This command will create a new branch with name `<branch>`
  - This branch will only exist on your local repository
- Do this:
  1. `git branch feature`
  2. `git branch`
- You should see:
  - `feature`
  - `* master`
- Notice that we are still on the "master" branch

Branching - Let’s checkout that branch

- Git Command: `git checkout <branch>`
  - This command will switch the current branch to `<branch>`
  - Any unstaged changes will be carried over and applied to the branch you switch to
  - The contents of the staging area will also be carried over
  - If git cannot make these changes without an issue, it will not let you switch branches
- Do this:
  1. `git checkout feature`
  2. Create a new file, feature.txt
  3. `git add feature.txt`
  4. `git commit -m "First commit on feature branch"`
  5. `git checkout master`
- If you look for feature.txt, you will notice it is not there!
  - This is because the last commit on "master" did not contain the file feature.txt

Branching - Working remotely

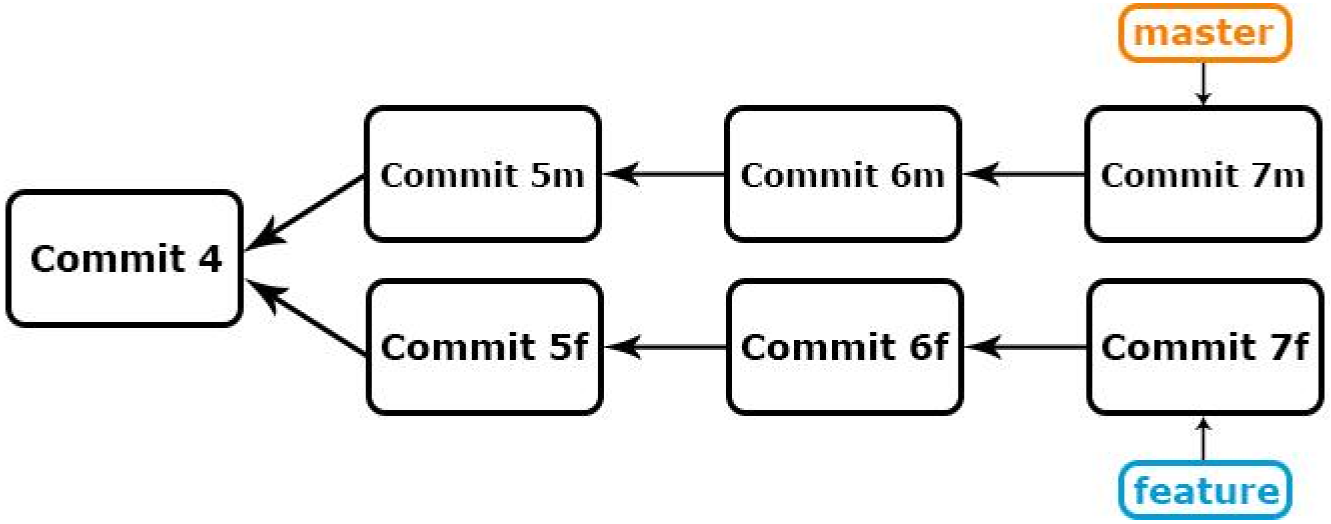
- Our new "feature" branch only exists locally.
- What if we want someone else to be able to work on our branch?
- We can create a new remote branch that anyone who has access to the repository will be able to work in.
- **Notice:** Making a new remote branch is a big structural change to a project! You should talk with your team (and your boss!) if you think your project would benefit from a new remote branch.

Branching - Making a new (remote) branch

- Git Command: `git push -u <remote> <branch>`
  - `-u` Links the current local branch to a remote branch
    - You don't need to use `-u` every time you push, only when you want to connect the current branch to a remote branch
  - `<remote>` is the name of the remote repository
    - The default remote repository name is `origin`
  - `<branch>` is the name of the remote branch we want to create (or link to)
    - This should be the same name as current local branch
- Do this:
  1. `git checkout feature`
  2. `git push -u origin feature`
  3. Look on GitHub to see that we have created another remote branch

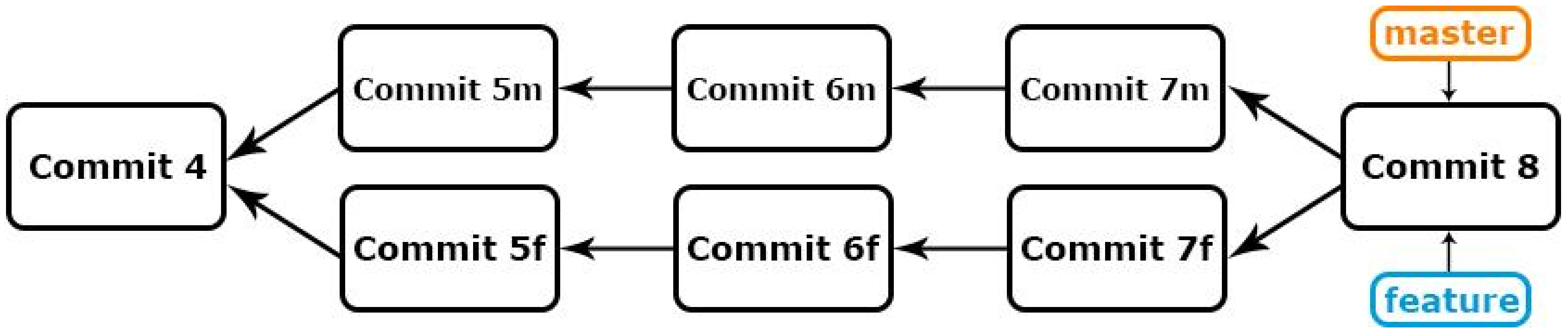
The result of branching

- When working with multiple branches, your project history can look something like this:



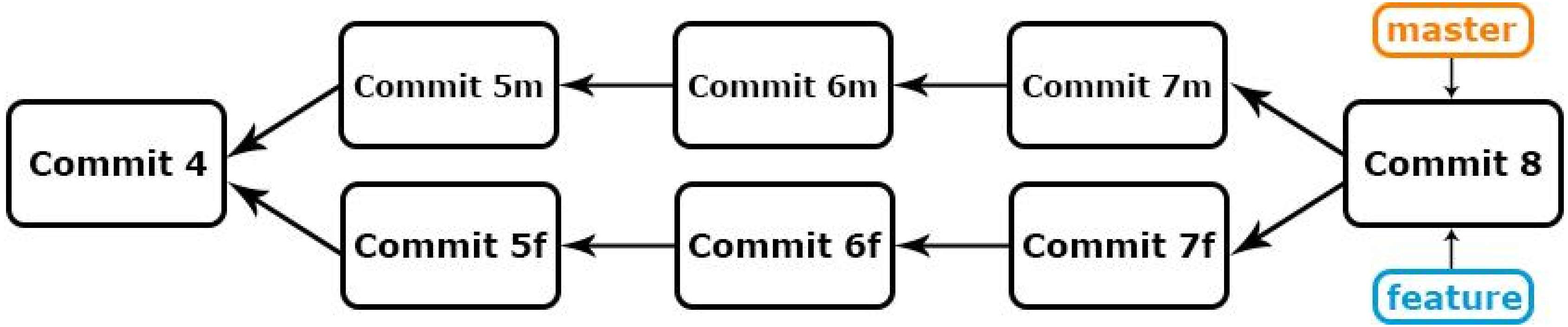
The result of branching

- Eventually, we will want to **merge** the "feature" branch into the "master" branch, and make our project history look like this:



Merging

- Git Command: `git merge <branch>`
  - Apply changes from `<branch>` to the current branch starting from the point where they differ
  - In the below picture, `master` with all commits shown once `feature` is merged

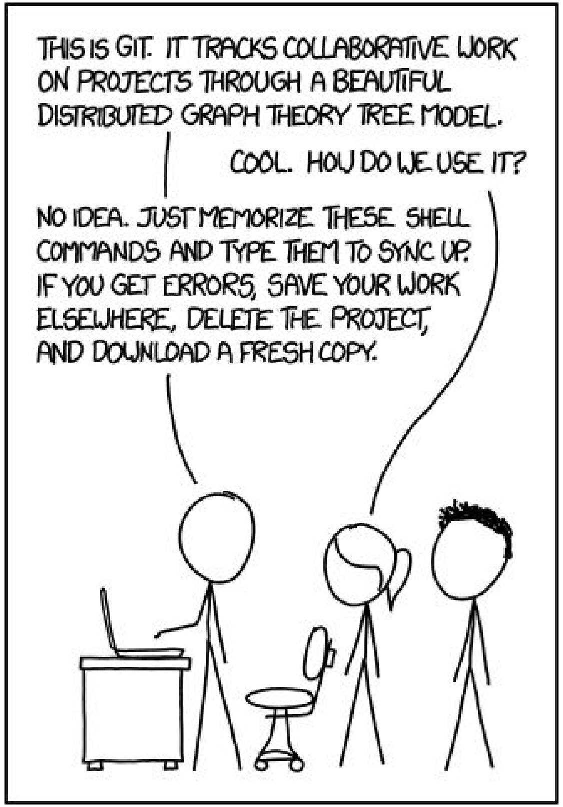


Merging

- Do this:
  1. `git checkout master`
  2. `git merge feature`
  3. `git status`
  4. `ls` (or `dir`, or look at the folder)
- We can now see that `feature.txt` is in the `master` branch
- Do this:
  1. `git push origin master`
- On GitHub, navigate to “graphs” and then “network”; `feature` and `master` now point to the same commit

Teamwork Makes the Dream Work!

- Git is best used in a team setting
  - Multiple developers working on different parts of the same code base
  - Conflicts can happen, and git can help!
  - When you and another teammate both modify the same code, **merge conflicts** can happen
- People think that merge conflicts are bad


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.

Teamwork Makes the Dream Work!

- How to resolve merge conflicts
  1. The easy way:
     1. Open affected files
     2. Find conflicts
     3. Fix conflicts
  2. The right way:
     1. `git mergetool`
- We’re going to walk through the easy way today, but we HIGHLY encourage you to use try `git mergetool` (and if we have time we’ll show you how to use it)

Conflicts of Interest

- Do this to simulate developer 1:
  1. `git checkout -b developer1`
     - Side note: `git checkout -b <branch name>` is shorthand for `git branch <branch name>; git checkout <branch name>`
  2. Open `README.md` and add some stuff where your name is
  3. Open `feature.txt` and change that file in some way
  4. `git add README.md feature.txt`
  5. `git commit -m “simulating developer 1”`

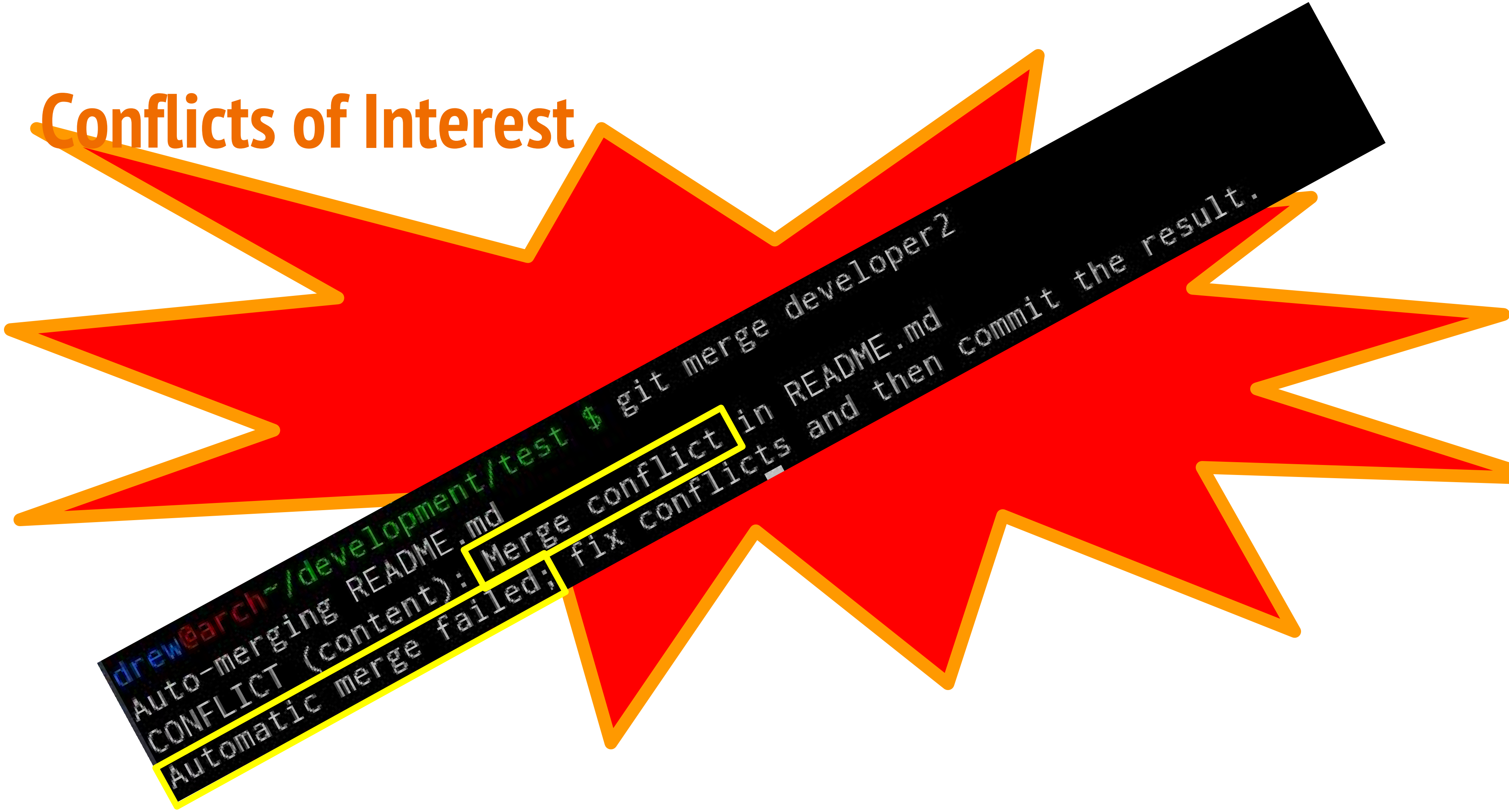
Conflicts of Interest

- Do this to simulate developer 2:
    1. `git checkout master`
    2. `git checkout -b developer2`
    3. Open `README.md` and add some stuff where your name is (but don't change it to the same thing you did before!)
    4. `git add README.md`
    5. `git commit -m “simulating developer 2”`

Conflicts of Interest

- Now let’s merge developer 1 work into `master`
- Do this:
    1. `git checkout master`
    2. `git merge developer1`
    3. Open README.md to see your changes have been merged into `master`
- Now let’s merge developer 2 work into `master`
- Do this:
    1. `git merge developer2`

Conflicts of Interest



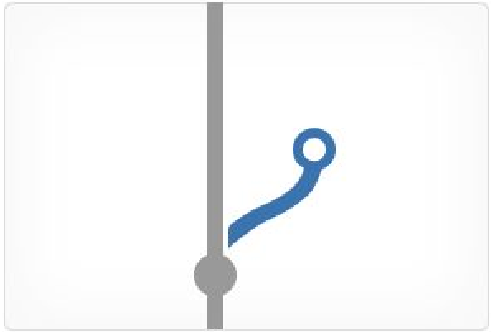
```
drew@arch~/development/test $ git merge developer2
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
```

Conflicts of Interest

- Do this:
  1. `git status`
  2. Open `README.md`
  3. Choose the changes you want (delete the ===, <<<, and >>> as well as the stuff you don't want)
     - <<< is the HEAD of the current branch
     - >>> is the changes you're trying to merge in
     - === is the division between them
  4. Save the file
  5. `git status`
  6. `git add README.md`
  7. `git commit -m “fix merge conflicts”`
  8. `git status`
  9. Open `README.md`
  10. Open `feature.txt`

Pull Requests

- Notify others that you've made changes that you want merged
- Discuss and review changes
- Make follow up commits
- Accept pull request to merge

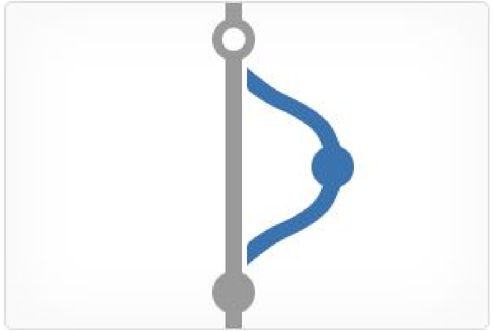


Branch

Develop features on a branch and create a pull request to get changes reviewed.

Discuss

Discuss and approve code changes related to the pull request.

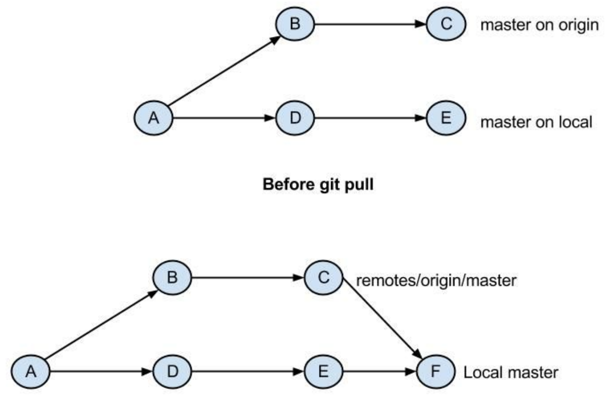


Merge

Merge the branch with the click of a button.

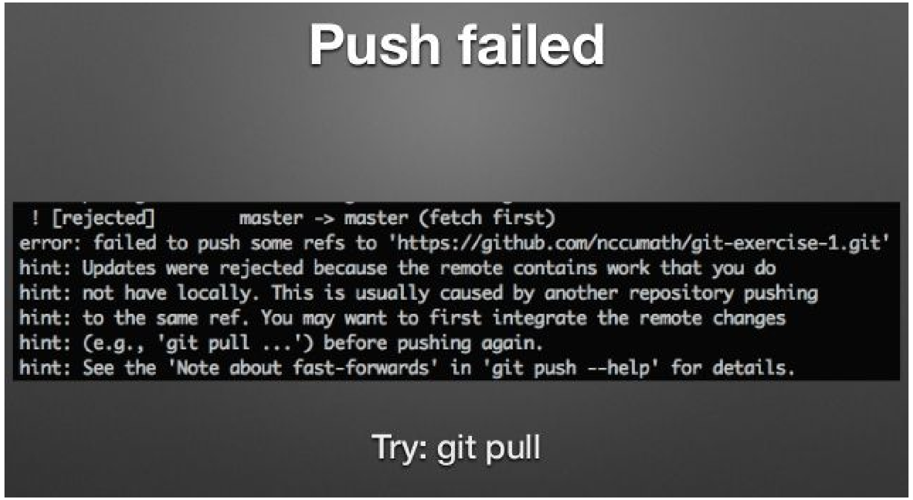
Git pull

- Git command: `git pull`
- Bring local branch up to date with remote
    - Any changes on remote branch will now exist on local branch
- Make habit of pulling before making changes



Before git pull

After git pull



```
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/nccumath/git-exercise-1.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Miscellaneous Commands and Other Information

- `git log`
- `git stash`
- `git rebase`
- `git bisect`
- `git clean`
- `.gitignore`
- Hooks (client side and server side)

Useful Sources

- Git Documentation
  - https://git-scm.com/documentation
- Useful utility for creating a gitignore
  - gitignore.io
  - You can `curl` the resulting url directly into a `.gitignore`
- Everyone references this site when talking about branching models
  - http://nvie.com/posts/a-successful-git-branching-model/
- Using `git log` to create a tree in the terminal
  - http://stackoverflow.com/questions/1064361/unable-to-show-a-git-tree-in-terminal