Git Notes:

Notes from “The Git & Github Bootcamp” on Udemy, taught by Colt Steele.

Table of Contents

Section 3 – Installation & Setup:

Installing Git: Terminal vs GUIs:

- Git is primarily a terminal tool.
    - Git was created as a command-line tool. To use it, we run various git commands in a Unix shell. This is not the most user-friendly experience, but it's at the very core of Git.
- The rise of GUIs
    - Over the last few years, companies have created graphical user interfaces for Git that allow people to use Git without having to be a command-line expert.
    - Popular Git GUIs include:
        - Github Desktop
        - SourceTree
        - Tower
        - GitKraken
        - Ungit

Configuring Your Git Name & Email:

- We did this in Git Bash.
- For name:
    - Check: git config user.name
    - Set: git config --global user.name "Travis Rillos"
- For email:
    - Check: git config user.email
    - Set: git config --global user.email xroadtraveler@gmail.com

Installing a GUI (GitKraken):

- Linked it to my existing GitHub account, made sure my name and email match both that and what I configured in Git Bash.

Section 4 – The Very Basics of Git: Adding & Committing:

What is a Git Repo:

- A Git "Repo" is a workspace which tracks and manages files within a folder.
    - Anytime we want to use Git with a project, app, etc we need to create a new Git repository. We can have as many repos on our machine as needed, all with separate histories and contents.

Our First Commands: "Git Init" and "Git Status":

- **git status** gives information on the current status of a git repository and its contents. It's very useful, but at the moment we don't actually have any repos to check the status of!
- Use **git init** to create a new git repository. Before we can do anything git-related, we must initialize a repo first!
    - This is something you do once per project. Initialize the repo in the top-level folder containing your project.

The Committing Workflow Overview:

- Do work, git add, git commit.

Staging Changes with Git Add:

- "**git add file1 file2**" adds files to staging area.
- They can then be committed.

Finally, the Git Commit Command!

- A “commit message” should summarize the changes made by a commit.
- “**git commit**” or ‘**git commit -m “my message”**’
    - The -m flag allows us to pass in an inline commit message, rather than launching a text editor (like VSCode or vim).
    - We’ll learn more about writing good commit messages later on.

The Git Log Command (and more committing):

- “**git status**” will show both untracked files (“new file”) and modified files (“modified”).
- “**git log**” retrieves a log of the commits for a given repository.
- “**git add .** ” will stage all changes at once.

Committing Exercise:

- Easy enough.

Section 5 – Commits in Detail (and Related Topics):

Keeping Your Commits Atomic:

- **Atomic Commits:** When possible, a commit should encompass a single feature, change, or fix. In other words, try to **keep each commit focused on a single thing**.
- This makes it much easier to undo or rollback changes later on. It also makes your code or project easier to review.

Commit Messages: Present or Past Tense:

- **Convention:** Git docs recommend "present-tense imperative style". "Describe your changes in imperative mood, e.g. "make xyzzy do frotz" instead of "[This patch] makes xyzzy do frotz" or "I changed Xyzzy to do frotz", as if you are giving orders to the codebase to change its behavior."

Fixing Mistakes with "Amend":

- **Amending Commits:**
- Suppose you just made a commit and then realized you forgot to include a file. Or, maybe you made a typo in the commit message that you want to correct.
- Rather than making a brand new separate commit, you can "redo" the previous commit using the **--amend** option.
- Note: Only works on the most previous commit.

Example:

- Purposely pretended to forget to "add" one of four files to staging/commit. Running "git status" would've shown the fourth file ("outline.txt") in red.
- "Realized" after committing that fourth file was red in "git status".
- Next:
- **git add outline.txt**
- **git commit –amend**
- Opens previous commit up in VS Code (or other configured editor) with the previous commit message displayed (in case that also needs modification, I guess).
- Newly included file will be shown down in commented section along with the others.

Ignoring Files w/ .gitignore:

- **Ignoring Files:**
- We can tell Git which files and directories to ignore in a given repository, using a **.gitignore** file. This is useful for files you know you NEVER want to commit, including:
    - Secrets, API keys, credentials, etc.
    - Operating System files (.DS_Store on Mac)
    - Log files
    - Dependencies & packages
- **.gitignore**:
- Create a file called ".gitignore" in the root of a repository. Inside the file, we can write patterns to tell Git which files & folders to ignore:
    - .DS_Store will ignore files named .DS_Store
    - [folderName]/ will ignore an entire directory
    - *.log will ignore any files with the .log extension
- Commonly used .gitignore options for many programming languages can be found at gitignore.io.

Section 6 – Working with Branches:

The Master Branch (or is it Main?):

- **The Master Branch:**
- In git, we are always working on a branch. The default branch name is **master**.
- It doesn't do anything special or have fancy powers. It's just like any other branch.

Viewing All Branches with Git Branch:

- **Viewing Branches:**
- Use **git branch** to view your existing branches. The default branch in every git repo is master, though you can configure this.
- Look for the ***** which indicates what branch you are currently on.

Creating & Switching Branches:

- **Creating Branches:**
- Use **git branch <branch-name>** to make a new branch based upon the current HEAD.
    - (The name shouldn't include spaces)
- This just creates the branch. It does not switch you to that branch (the HEAD stays the same).
- **Switching Branches:**
- Once you have created a new branch, use **git switch <branch-name>** to switch to it.
- Using **git switch -c <branch-name>** both creates and switches in one line.

More Practice with Branching:

- **git commit -a -m "[add commit message here]"** is a nice one-line way to add and commit at the same time.

Another Option: Git Checkout vs. Git Switch:

- **Another Way of Switching??:**
- Historically, we used git **checkout <branch-name>** to switch branches. This still works.
- The **checkout** command does a million additional things, so the decision was made to add a standalone switch command which is much simpler.
- You will see older tutorials and docs using checkout rather than switch. Both now work.
- **Creating & Switching:**
- Use **git switch** with the **-c** flag to create a new branch AND switch to it all in one go. Looks like **git switch -c <branch-name>**.
- Remember **-c** as short for "create".

Switching Branches with Unstaged Changes?:

- Use **git branch -d <branch-name>** to delete a branch.
- However, you can't delete a branch if it's checked out by someone or if it isn't fully merged.
    - If you're sure you want to delete a branch even if it's not merged, use the -D flag, which looks like: **git branch -D <branch-name>**.
- To rename, the flag is **-m** (for "move/rename").
- To rename, you have to BE on the branch.
- Use **git switch** or **git checkout** to switch to the branch you want to rename, and then
- Use **git branch -m <new name>**.

How Git Stores HEAD & Branches:

- HEAD points to the most recent hash number of a given branch.
- They are kept in .git/HEAD

Branching Exercise:

- Remember, while in a branch, one can open a file in VSCode by using **code <file-name>** and then making whatever edits are needed.
- Easy enough, just followed the directions.

Section 7 – Merging Branches:

An Introduction to Merging:

- **Merging:**
- Branching makes it super easy to work within self-contained contexts, but often we want to incorporate changes from one branch into another!
- We can do this using the **git merge** command.
- The merge command can sometimes confuse students early on. Remember these two merging concepts:
    - **We merge branches, not specific commits**
    - **We always merge to the current HEAD branch**
- **Merging Made Easy:**
- **To merge, follow these basic steps:**
    - 1) Switch to (or checkout) the branch you want to merge the changes into (the receiving branch).
    - 2) Use the **git merge** command to merge changes from a specific branch into the current branch.

Generating Merge Commits: (non "fast-forward" merges)

- Git creates a new commit (a "merge commit") that contains changes from both branches.
- This is the first commit we've seen that has two parents instead of just one.
- There can be conflicts between merges, say if the same line of code is modified in different ways in two branches.

Oh No! Merge Conflicts!:

- **Heads Up!**
- Depending on the specific changes you are trying to merge, Git may not be able to automatically merge. This results in **merge conflicts**, which you need to manually resolve.
- ""> CONFLICT (content): Merge conflict in blah.txt
- Automatic merge failed; fix conflicts and then commit the result."""

- **What the…**
- When you encounter a merge conflict, Git warns you in the console that it could not automatically merge.

- **It also changes the contents of your files to indicate the conflicts that it wants you to resolve.** <<< well that's fuckin' useful.

- **Conflict Markers:**
- The content from the branch you are trying to merge from is displayed **between the ======= and >>>>>>> symbols.**

- **Resolving Conflicts:**
- Whenever you encounter merge conflicts, follow these steps to resolve them:
- 1) Open up the file(s) with merge conflicts.
- 2) Edit the file(s) to remove the conflicts. Decide which branch's content you want to keep in each conflict. Or keep the content from both.
- 3) Remove the conflict "markers" in the document.
- 4) Add your changes and then make a commit!

Using VSCode to Resolve Conflicts:

- VSCode has a built-in interface to help resolve conflicts. Can select options such as "**Accept Current Change**" (what's in current branch), "**Accept Incoming Change**" (what's in merge branch), "**Accept Both Changes**" (adds everything from both?), and "**Compare Changes**" (visual representation of changes, I guess?).
- These choices speed things up. For example, you don't need to manually delete the conflict "markers".
- Can also resolve conflicts in a similar way on GUI programs like GitKraken, but he didn't seem to have any experience with this. It probably works very similarly.

Merging Exercise:

- This one was more difficult. Had trouble getting the third result.
- For parts 2 and 3, should've done some work on master as well as new branch. In part 2, they shouldn't conflict, and in part 3 they should. Without doing work on master, it just causes more fast-forward merges.

Section 8 – Comparing Changes with Git Diff:

Introducing the Git Diff Command:

- **Git Diff:**
- We can use the **git diff** command to view changes between commits, branches, files, our working directory, and more!
- We often use git diff alongside commands like git status and git log, to get a better picture of a repository and how it has changed over time.
- **git diff:**
- Without additional options, **git diff** lists all the changes in our working directory that are NOT staged for the next commit.
- Compares Staging Area and Working Directory.

A Guide to Reading Diffs:

- **Compared Files:**
- For each comparison, Git explains which files it is comparing. Usually this is two versions of the same file.
- Git also declares one file as "A" and the other as "B".
- **Markers:**
- File A and File B are each assigned a symbol.
    - File A gets a minus sign (-)
    - File B gets a plus sign (+)
- This helps show which files "own" which differences listed in the following lines.
- **Chunks:**
- A diff won't show the entire contents of a file, but instead only shows portions or "chunks" that were modified.
- A chunk also includes some unchanged lines before and after a change to provide some context.
- Can show multiple chunks if file was changed in multiple places.
- **Chunk Header:**
- **@@ -3,4 +3,5 @@**
- Each chunk starts with a chunk header, found between @@ and @@.
- From File A, 4 lines are extracted starting from line 3.
- From File B, 5 lines are extracted starting from line 3.
- **Changes:**
- Every line that changed between the two files is marked with either a + or – symbol.
- Lines that begin with - come from File A.
- Lines that begin with + come from File B.

Viewing Unstaged Changes:

- **git diff:** (from before)
- Without additional options, **git diff** lists all the changes in our working directory that are NOT staged for the next commit.
- Compares Staging Area and Working Directory.

Viewing Working Directory Changes:

- **git diff:**
- **git diff HEAD** lists all changes in the working tree since your last commit.
- Includes staged AND unstaged changes (so if you **add** but haven't **commit**ted yet).

Viewing Staged Changes:

- **git diff:**
- **git diff --staged** or **--cached** (same thing, just an alias) will list the changes between the staging area and our last commit.
- Only includes staged, not unstaged changes.
- "Show me what will be included in my commit if I run **git commit** right now".

Diffing Specific Files:

- **Diff-ing Specific Files:**
- We can view the changes within a specific file by providing gif diff with a filename.
- Examples:
    - **git diff HEAD [filename]**
    - **git diff --staged [filename]**

Comparing Changes Across Branches:

- **Comparing Branches:**
- **git diff branch1..branch2** will list the changes between the tops of branch1 and branch2.
- Side note: The order you list the branches will determine which file is File A and which is File B.

Comparing Changes Across Commits:

- **Comparing Commits:**
- To compare two commits, provide git diff with the commit hashes of the commits in question.
    - **git diff commit1..commit2**
- Note: Used **git log --online** to show a list of commits with their hashes to the left.
- Compared two commits from different times in the master branch.
- Can narrow it down to specific files like earlier, as well as staged files. Can also use regular git diff as well as git diff HEAD to see unstaged and staged files.

Visualizing Diffs with GUIs:

- He just showed how the same processes work in GitKraken.
- When you clicked on an unstaged file in the "Unstaged Files" window, there are two tabs in the top-middle that say "File View" and "Diff View".
- In this way, you only need to click on buttons and locations rather than using all of the previous commands.
- Similarly, two commits can be compared in the graph tree by clicking one, holding shift, and clicking another.
- GUIs make it very easy to visualize diffs.
- There's a "Hunk View" button towards the top right of the screen. You can toggle that to "In-Line View" to view the entire file instead of just chunks/hunks. Could be useful sometimes.

Git Diff Exercise:

- Involves an existing repository that needs to be downloaded (a first for this course).
- **Part 1: Setup:**
- Command for downloading:
    - **git clone https://github.com/Colt/git-diff-exercise**
    - This brings a list of commits from an existing repo, without me having to make the changes myself.
- One branch, "current", was created/switched to by default.
- Switched to another branch, "1970s". Both branches contain versions of both files.
- **Part 2: The Diffs:**
- First two were easy enough, then used that **git log --online** trick to get the hashes for the third.
- The rest of it was just following directions and referring to my notes. Easy enough.

Section 9 – The Ins and Outs of Stashing:

Why We Need Git Stash:

- If you need to switch to another branch for some reason, you can **stash** your current unstaged/uncommitted work on your previous branch for later.
- Switching to another branch can sometimes be unintentionally brought along to the branch you're switching to.
- The two possibilities he mentioned are:
    - 1) My changes come with me to the destination branch.
    - 2) Git won't let me switch if it detects potential conflicts.

Stashing Basics: Git Stash Save & Pop:

- **Stashing:**
- Git provides an easy way of stashing these uncommitted changes so that we can return to them later, without having to make unnecessary commits.
- **Git Stash:**
    - **git stash** is a super useful command that helps you save changes that you are not yet ready to commit. You can stash changes and then come back to them later.
- Running git stash will take all uncommitted changes (staged and unstaged) and stash them, reverting the changes in your working copy.
- (You can also use **git stash save** instead).
- **Stashing:**
- Use **git stash pop** to remove the most recently stashed changes in your stash and re-apply them to your working copy.

Practicing with Git Stash:

- Mostly he just shows what he already talked about in this video.
- He says that generally he only has one thing (work from one branch, that is) in his stash at a time. This makes sense if you want to keep things simple.

Git Stash Apply:

- He says mostly he just uses **git stash** and **git stash pop**.
- **Stash Apply:**
- You can use **git stash apply** to apply whatever is stashed away, without removing it from the stash. This can be useful if you want to apply stashed changes to multiple branches. (I guess it would be like using copy/paste or "replace all" in other types of documents).
- Applying changes to other branches can create conflicts, just like **merging branches**, and you are given the opportunity to choose which changes you want to keep.
- Using **git stash apply** keeps the stashed changes, whereas git stash pop will **pop** the changes out of the stash (think of the "pop" command in Python and other languages).

Working with Multiple Stashes:

- **Stashing Multiple Times:**
- You can add multiple stashes onto the stack of stashes. They will all be stashed in the order you added them.
    - I wonder if this functions like a list or a similar set in other languages, where the "pop" command just pops off and returns the end of the set.
- As he said earlier, he doesn't do this often himself.
- He's using rainbow-ordered colored backgrounds and stashing each of them in turn. If my hunch was right about how "pop" works, I think the colors will show up in reverse order when he applies them one-by-one at the end of the video.
- He only did it for three colors, but let's see what happens.
- **git stash list** allows you to see what you've stashed so far. For example:
    - stash@{0}: WIP on rainbow: 56b5b74 remove background color *(most recent)*
    - stash@{1}: WIP on rainbow: 56b5b74 remove background color
    - stash@{2}: WIP on rainbow: 56b5b74 remove background color
    - stash@{3}" WIP on goodbye: 94ec757 create index.html and app.css *(leftover from previous video)*
- I was wrong about the order. Using **git stash pop** will pop off the oldest change first, so that leftover stash from the previous video is a slight problem (probably not a real problem though).
- **Applying Specific Stashes:**
- Git assumes you want to apply the most recent stash when you run **git stash apply**, but you can also specify a particular stash like **git stash apply stash@{2}**.
    - Well now I don't know what to think about what order this is gonna happen in.
- He ended up doing a specific one, so I don't think my question is going to get answered. Not a big deal, but I was curious.
    - In the next video he ended up answering my question, and yes it was the most recent.

Dropping & Clearing the Stash:

- **Dropping Stashes:**
- To delete a particular stash, you can use **git stash drop <stash-id>**.
    - Ex.: **git stash drop stash@{2}**
- **Clearing the Stash:**
- To clear out all stashes, run **git stash clear**.
    - Ex.: **git stash clear**

Stashing Exercise:

- Easy enough, just followed the directions.
- Note: Directions had me switch to master branch BEFORE stashing the changes. Interesting.
- Also, got some weird warning but ignored it and stuff worked.

Section 10 – Undoing Changes & Time Traveling:

Checking Out Old Commits:

- **Checkout:**
    - The **git checkout** command is like a Git Swiss Army Knife. Many developers think it's overloaded, which is what lead to the addition of the **git switch** and **git restore** commands.
- We can use **checkout** to create branches, switch to new branches, restore files, and undo history!
- **Checkout:**
- We can use **git checkout commit <commit-hash>** to view a previous commit.
    - Ex.: **git checkout d8194d6**
- Remember, you can use the **git log** command to view commit hashes. We just need the first 7 digits of a commit hash.
- Don't panic when you see the following message…
- **Detached HEAD:**
- You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by switching back to a branch.
- In this case, HEAD points to a specific commit instead pointing to the branch reference like normal.

Re-Attaching Our Detached HEAD:

- **Detached HEAD:**
- Don't panic when it happens, it's not a bad thing!
- **You have a couple of options:**
    - 1. Stay in detached HEAD to examine the contents of the old commit. Poke around, view the files, etc.
    - 2. Leave and go back to wherever you were before – reattach the HEAD.
    - 3. Create a new branch and switch to it. You can now make and save changes, since the HEAD is no longer detached.
- Use **git switch master** or **git checkout master** to switch back to the master branch and reattach the HEAD that way. Or, switch to whatever branch you were on before detaching the HEAD (doesn't have to be master).
- Can also create a new branch from whichever commit the detached HEAD is pointing.

Referencing Commits Relative to HEAD:

- **Checkout:**
    - **git checkout** supports a slightly odd syntax for referencing previous commits relative to a particular commit.
- HEAD~1 refers to the commit before HEAD (parent)
- HEAD~2 refers to 2 commits before HEAD (grandparent)
    - Ex.: **git checkout HEAD~1**
    - Think of them as "HEAD minus 1" or HEAD minus 2"
    - Can do that without having to go look up the commit hash for a particular commit.
- This is not essential, but I wanted to mention it because it's quite weird looking if you've never seen it.
- Shortcut: Using "**git switch -**" will take you back to whichever branch you were just on. Kind of like using "." or ".." in various Linux/bash uses.

Discarding Changes with Git Checkout:

- **Discarding Changes:**
- Suppose you've made some changes to a file but don't want to keep them. To revert the file back to whatever it looked like when you last committed, you can use:
- **git checkout HEAD <filename>** to discard any changes in that file, reverting back to the HEAD.
- In this example, he made a bunch of "accidental" changes to his two .txt files, cat.txt and dog.txt, then ran **git checkout HEAD dog.txt** and **git checkout HEAD cat.txt** to undo all changes since the last commit.
- This is because HEAD is still pointed at the last commit.
- Can also use the shortcut **git checkout -- dog.txt** and **git checkout -- cat.txt** (note, this is a double-dash, not a single-dash) to do the same.

Un-Modifying with Git Restore:

- **Restore:**
    - **git restore** is a brand new Git command that helps with undoing operations.
- Because it is so new, most of the existing Git tutorials and books do not mention it, but it is worth knowing.
- Recall that **git checkout** does a million different things, which many git users find very confusing. **git restore** was introduced alongside **git switch** as alternatives to some of the uses for checkout.
- **Un-Modifying Files with Restore:**
- Suppose you've made some changes to a file since your last commit. You've saved the file but then realize you definitely do NOT want those changes anymore!
- To restore the file to the contents in the HEAD, use **git restore <file-name>**.
    - NOTE: The above command is not "undo-able". If you have uncommitted changes in the file, they will be lost.
- Example he used was **git restore dog.txt**, one of the files from the previous video.
- **Un-Modifying Files with Restore:**
- **git restore <file-name>** restores using HEAD as the default source, but we can change that using the --**source** option.
- For example, **git restore --source HEAD~1 home.html** will restore the contents of home.html to its state from the commit prior to HEAD. You can also use a particular commit hash as the source.

Un-Staging Changes with Git Restore:

- **Un-staging Files with Restore:**
- If you have accidentally added a file to your staging area with **git add** and you don't wish to include it in the next commit, you can use **git restore** to remove it from staging:
- Use the **--staged** option like this:
    - **git restore --staged <file-name>**

Undoing Commits with Git Reset:

- **Git Reset:**
- Suppose you've just made a couple of commits on the master branch, but you actually meant to make them on a separate branch instead. To undo those commits, you can use **git reset**.
- **"Plain" Reset: git reset <commit-hash>** will reset the repo back to a specific commit. The commits are gone.
    - Ex.: **git reset 4661ab9** (← got commit hash from **git log --oneline**)
- In example, the changes were still there. The commits were removed. This is the "plain" reset.
- Can then create new branch with those changes.
- **Reset --hard:**
- If you want to undo both the commits AND the actual changes in your files, you can use the **--hard** option.
    - Ex.: **git reset --hard <commit-hash (or HEAD~x)>**.
- For example, **git --hard HEAD~1** will delete the last commit and associated changes.

Reverting Commits with…Git Revert:

- **Git Revert:**
- Yet another similar-sounding and confusing command that has to do with undoing changes.
- **git revert** is similar to **git reset** in that they both "undo" changes, but they accomplish it in different ways.
    - **git revert <commit-hash>** (or HEAD~x)
- **git reset** actually moves the branch pointer backwards, eliminating commits.
- **git revert** instead creates a brand new commit which reverses/undoes the changes from a commit. Because it results in a new commit, you will be prompted to enter a commit message.
- This one seems best for collaborating, since you can still keep track of changes and why they were made/reverted.
- **Which One Should I Use?**
- Both **git reset** and **git revert** help us reverse changes, but there is a significant difference when it comes to collaboration (which we have yet to discuss but is coming up soon!)
- **If you want to reverse some commits that other people already have on their machines, you should use revert.**
- If you want to reverse commits that you haven't shared with others, use reset and no one will ever know!

Undoing Changes Exercise:

- Got it in one.

Section 11 – Github: The Basics:

What Does Github Do for Us?

- **What is Github?**
- Github is a hosting platform for git repositories. You can put your own Git repos on Github and access them from anywhere and share them with people around the world.
- Beyond hosting repos, Github also provides additional collaboration features that are not native to Git (but are super useful). Basically, Github helps people share and collaborate on repos.
- **Github is not your only option…**
- There are tons of competing tools that provide similar hosting and collaboration features, including GitLab, BitBucket, and Gerrit.
- With that said….
- **It's very popular!**
- Founded in 2008, Github is now **the world's largest host of source code**. In early 2020, Github reported having over 40 million users and over 190 million repositories on the platform.
- **It's Free!**
- Github offers its basic services for free! While Github does offer paid Team and Enterprise tiers, the basic Free tier allows for unlimited public and private repos, unlimited collaborators, and more!

Why You Should Use Github!

- **Collaboration:**
- If you ever plan on working on a project with at least one other person, Github will make your life easier! Whether you're building a hobby project with your friend or you're collaborating with the entire world, Github is essential.
- **Open Source Projects:**
- Today Github is THE home of open source projects on the internet. Projects ranging from React to Swift are hosted on Github.
- If you plan on contributing to open source projects, you'll need to get comfortable working with Github.
- **Exposure:**
- Your Github profile showcases your own projects and contributions to others' projects. It can act as a sort of resume that many employers will consult in the hiring process. Additionally, you can gain some clout on the platform for creating or contributing to popular projects.
- **Stay Up to Date:**
- Being active on Github is the best way to stay up to date with the projects and tools you rely on. Learn about upcoming changes and the decisions/debates behind them.

Cloning Github Repos with Git Clone:

- **Cloning:**
- So far we've created our own Git repositories from scratch, but often we want to get a **local copy of an existing repository** instead.
- To do this, we can clone a remote repository hosted on Github or similar websites. All we need is a URL that we can tell Git to clone for use.
- **git clone:**
- To clone a repo, simply run **git clone <url>**.
- Git rill retrieve all the files associated with the repository and will copy them to your local machine.
- In addition, Git initializes a new repository on your machine, giving you access to the full Git history of the cloned project.
    - Note: Make sure you are not inside of a repo when you clone!

Cloning Non-Github Repos:

- **Permissions?**
- **Anyone can clone a repository from Github**, provided the repo is public. You do not need to be an owner or collaborator to clone the repo locally to your machine. You just need the URL from Github.
- Pushing up your own changes to the Github repo…that's another story entirely! You need permission to do that!
- **We are not limited to Github Repos!**
- **git clone** is a standard git command.
- It is NOT tied specifically to Github. We can use it to clone repositories that are hosted anywhere! It just happens that most of the hosted repos are on Github these days.

Github Setup: SSH Config:

- **SSH Keys:**
- You need to be authenticated on Github to do certain operations, like pushing up code from your local machine. Your terminal will prompt you every single time for your Github email and password, unless…
- You generate and configure an SSH key! Once configured, you can connect to Github without having to supply your username/password.
- Following steps on Github's website in the SSH Key section:
    - Checked for existing SSH key with **ls -al ~/.ssh** and nothing was there, so I followed a link leading me to the next page (https://docs.github.com/en/authentication/connecting-to-

github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent) in order to generate a new one.
    - Website instructed me to use the following command:
        - **ssh-keygen -t ed25519 -C "your_email@example.com"**
    - This creates a new SSH key, using the provided email as an example. I used my personal email. Can probably generate ones for "professional" emails later if needed.
    - **Used the passphrase** I use for Oracle VirtualBox Linux root access from my other class.
    - Ended up continuing through the directions, running the ssh-agent in the background and adding my ed25519.
    - Followed directions to add SSH key to my Github account.

Creating Our First Github Repo!

- **How Do I Get My Code on Github?**
- **Option 1: Existing Repo:**
- If you already have an existing repo locally that you want to get on Github:
    - Create a new repo on Github
    - Connect your local repo (add a remote)
    - Push up your changes to Github
- **Option 2: Start from Scratch:**
- If you haven't begun work on your local repo, you can:
    - Create a brand new repo on Github
    - Clone it down to your machine
    - Do some work locally
    - Push up your changes to Github

- **Option 1 Follow-Along:**
- Create empty Github repo.
- I'll pick my Patronus exercise (section 6) for this practice.
- We got as far as creating a new empty repo in this vid, more steps in next vid.

A Crash Course on Git Remotes:

- **Remote:**
- Before we can push anything up to Github, we need to tell Git about our remote repository on Github. We need to setup a "destination" to push up to.
- In Git, we refer to these "destinations" as **remotes**. Each remote is simply a URL where a hosted repository lives.
-
-

- **Viewing Remotes:**
- To view any existing remotes for your repository, we can run **git remote** or **git remote -v** (verbose, for more info).
- This just displays a list of remotes. If you haven't added any remotes yet, you won't see anything.
- **Adding a New Remote:**
- A remote is really two things: a URL and a label. To add a new remote, we need to provide both to Git.
    - Command: **git remote add <name> <url>**
    - Ex.: **git remote add origin https://github.com/blah/repo.git**
    - "Okay Git, anytime I use the name "**origin**", I'm referring to this particular Github repo URL".
- **Origin?**
    - **Origin** is a conventional Git remote name, but it is not at all special. It's just a name for a URL.
- When we clone a Github repo, the default remote name setup for us is called "origin". You can change it. Most people leave it.
- **Other Commands:**
- They are not commonly used, but there are commands to rename and delete remotes if needed.
    - **git remote rename <old> <new>**
    - **git remote remove <name>**

Introducing Git Push:

- **Pushing:**
- Now that we have a remote set up, let's push some work up to Github! To do this, we need to use the **git push** command.
- We need to specify the remote we want to push up to AND the specific local branch we want to push up to that remote.
    - **git push <remote> <branch>**
- **Option 1:** (that's what we're following along with)
- An Example: **git push origin master** tells git to push up the master branch to our origin remote.
- Due to new convention, it's common to rename the master branch to **main**.
- Instructions for doing this, from Github:
    - **git branch -M main**, followed by:
    - **git push -u origin main**
    - But we're not doing that this time, just to keep things simple. I still played around with it a little.
- After running **git push origin master**, a popup showed up and I had to authorize/sign-in to Github. Not sure if this happens every time, one time, or if it's because I haven't been active in my Github window since yesterday. After authorizing, Git Bash uploaded everything quickly.

- Refreshing that page on Github now shows a repo.
- Pushed the other branches too.
- Jut for fun, I added “random.txt” to my master branch, then I staged and committed that before pushing that up to Github as well.

Touring a Github Repo:

- He mostly just gave a brief tour of a Github repo.

Practice with Git Push:

- Just some more examples of using **git push**.

A Closer Look at Git Push:

- He created a new empty repo on Github called “pushme”, then created/initialized a new local repo called “Pushing” and touched a file called “candy.txt”.
- Added a remote for the Github repo.
- Pushed the local master branch onto Github. He wanted to illustrate that this creates a new linked master branch on Github, when there were no branches before.
- **Push In Detail:** (pushing from one branch to another)
- While we often want to push a local branch up to a remote branch of the same name, we don’t have to.
- To push our local “pancake” branch up to a remote branch called “waffle” we could do:
    - **git push origin pancake:waffle**
    - General syntax: **git push <remote> <local-branch>:<remote-branch>**
- Not that common, but he wanted to use this to illustrate the potential links between local and remote branches.

What does “git push -u” mean?

- **The -u Option:**
- The -u option allows us to set the upstream of the branch we’re pushing. You can think of this as a link connecting our local branch to a branch on Github.
- Running **git push -u origin master** sets the upstream of the local master branch so that it tracks the master branch on the origin repo.

- **What this means:**
- Once we've set the upstream for a branch, we can use the **git push** shorthand which will push our current branch to the upstream.
- Seems that you can do this for every branch you want to, or to set an upstream with a different name than the local version (like in the previous video).

Another Github Workflow: Cloning First:

- **Option 2: Start from Scratch:**
- If you haven't begun work on your local repo, you can:
    - Create a brand new repo on Github
    - Clone it down to your machine
    - Do some work locally
    - Push up your changes to Github
- He created an empty Github repo called "chickens-demo" to start with.
- Went into an empty directory/folder he created called "Cloning", checked **git status** to make sure it wasn't already a Git repo, then ran **git clone <url>**.
- This created a folder in the Cloning folder called "chickens-demo", which he then switched into. Git also alerted us to the fact that we cloned an empty repo.
- Since it was cloned, it came with a pre-configured remote called origin.
- Did some work locally (added a chickens.txt file with two of his chickens in it).
- Ran **git push origin master**.

Main & Master: Github Default Branches:

- Note: When creating a new repo in Github, if you check the "Add a README file" box, a message appears lower down that says "This will set main as the default branch. Change the default name in your settings".
- He then cloned the repo (which he named "colors"). The cloned branch is automatically named "main".
- To rename, use **git branch -M main** (renames the branch you're currently in). There are more instructions on Github for similar situations where you need to rename a branch.
- If needed, you can go into the "Settings" tab in the Github repo and change the default branch.

Github Basics Exercise:

- Made a minor error early in the directions and had to restart. Remember, if you've made commits to a Github repo (including checking the "Add a README file" checkbox) that aren't found on your local repo, it won't let you push from local to Github.
- Other than that, followed the directions all the way through. Easy enough.

Section 12 – Fetching & Pulling:

Remote Tracking Branches: WTF Are They?

- **Remote Tracking Branches:**
- "At the time you last communicated with this remote repository, here is where x branch was pointing".
- They follow this pattern **<remote>/<branch>**.
    - **origin/master** references the state of the master branch on the remote repo named origin.
    - **upstream/logoRedesign** references the state of the logoRedesign branch on the remote named upstream (a common remote name).
- **Branches:**
- Run **git branch -r** to view the remote branches our local repository knows about.

Checking Out Remote Tracking Branches:

- I make a new commit locally, my master branch reference updates, like always.
- The remote reference stays in the same place. It never moves.
- Making new commits locally now gives a **git status** message saying "Your branch is ahead of 'origin/main' by 1 commit".
- **You can checkout these remote branch pointers:**
    - Running **git checkout origin/master** puts you in detached HEAD state so you can look at it and such.
- If you end up pushing your local changes up to the Github repo and then run **git status**, you'll now get a message saying "Your branch is up to date with 'origin/main'".

Working with Remote Branches:

- This video was a follow-along that he suggested we all do. Watching seemed simple enough though.
- **Remote Branches:**
- Once you've cloned a repository, we have all the data and Git history for the project at that moment in time. However, that does not mean it's all in my workspace!
- The Github repo has a branch called **puppies**, but when I run **git branch** I don't see it on my machine! All I see is the master branch. What's going on?
- However, using **git branch -r** will show all of the online branches, so the local repo is aware they exist.
- By default, my master branch is already tracking origin/master. He didn't connect these himself.

- **I want to work on the puppies branch locally!**
- I could **checkout origin/puppies**, but that puts me in detached HEAD.
- I want my own local branch called **puppies**, and I want it to be connected to **origin/puppies**, just like my local **master** branch is connected to **origin/master**.
- **It's super easy!**
- Run **git switch <remote-branch-name>** to create a new local branch from the remote branch of the same name.
- **git switch puppies** makes e a local puppies branch AND sets it up to track the remote branch origin/puppies.
- **Note!** The new command **git switch** makes this super easy to do! It used to be slightly more complicated using **git checkout**:
    - **git checkout --track origin/puppies**

Git Fetch: the Basics:

- For updating your local branch if the remote branch has been changed.
- According to a diagram he showed:
    - **git fetch** brings a branch from the remote repo to the local repo.
    - **git pull** will bring a branch from the remote repo to your workspace/working directory.
- **Fetching:**
- Fetching allows us to download changes from a remote repository, BUT those changes will not be automatically integrated into our working files.
- It lets you see what others have been working on, without having to merge those changes into your local repo.
- Think of it as "please go and get the latest information from Github, but don't screw up my working directory".
- **Git Fetch:**
    - The **git fetch <remote>** command fetches branches and history from a specific remote repository. It only updates remote tracking branches.
- **git fetch origin** would fetch all changes from the origin remote repository.
    - If not specified, <remote> defaults to origin.
- We can also fetch a specific branch from a remote using **git fetch <remote> <branch>**.
- For example, **git fetch origin master** would retrieve the latest information from the master branch on the origin remote repository.

Demonstrating Git Fetch:

- Mostly just showing the concepts from the previous video.
- When he made changes to the Github "movies" branch in the repo and then ran **git fetch origin**, he got a message saying "Your branch is behind 'origin/movies' by 1 commit, and can be fast-forwarded. (use "git pull" to update your local branch)".

Git Pull: the Basics:

- **Pulling:**
  - **git pull** is another command we can use to retrieve changes from a remote repository. Unlike fetch, pull actually updates our HEAD branch with whatever changes are retrieved from the remote.
- "Go and download data from Github AND immediately update my local repo with those changes".
  - **git pull = git fetch + git merge**
- **git pull:**
  - To pull, we specify the particular remote and branch we want to pull using **git pull <remote> <branch>**. Just like with git merge, it matters WHERE we run this command from. Whatever branch we run it from is where the changes will be merged into.
- **git pull origin master** would fetch the latest information from the origin's master branch and merge those changes into our current branch.

Git Pull & Merge Conflicts:

- In this video, he showed a case where a coffee.txt file with ASCII art had been added to the Github repo's "food" branch, while at the same time he added his own (different) coffee.txt file to his local food branch.
- Before pushing a change to a remote branch, it's important to pull from that remote branch to see what changes have been made.
- He ran **git pull origin food** and got a message "CONFLICT (add/add): Merge conflict in coffee.txt".
- He manually resolved the conflict (kept both ASCII images in one coffee.txt file), then added and committed.
- Got message saying "Your branch is ahead of 'origin/food' by 2 commits" and that a push would fast-forward the remote branch. He then pushed these changes to the remote branch.

A Shorter Syntax for Git Pull?

- **An even easier syntax!**
- If we run **git pull** without specifying a particular remote or branch to pull from, git assumes the following:
    - Remote will default to origin.
    - Branch will default to whatever tracking connection is configured for your current branch.
- Note: This behavior can be configured, and tracking connections can be changed manually. Most people don't mess with that stuff.

- **git fetch:**
    - Gets changes from the remote branch(es).
    - Updates the remote-tracking branches with the new changes.
    - Does not merge changes onto your current HEAD branch.
    - Safe to do at any time.
- **git pull:**
    - Gets changes from the remote branch(es).
    - Updates the current branch with the new changes, merging them in.
    - Can result in merge conflicts.
    - Not recommended if you have uncommitted changes.

Section 13 – Github Grab Bag: Odds & Ends:

Github Repo Visibility: Public vs. Private:

- **Public repos** are accessible to everyone on the internet. Anyone can see the repo on Github.
- **Private repos** are only accessible to the owner and people who have been granted access.
    - Note: At the Enterprise level a private repo can be seen by anyone in the organization.

Adding Github Collaborators:

- Settings >> Manage access >> Invite a collaborator

Github Collaboration Demo:

- Demoing what was talked about in previous video.
- Pretended to be two different people collaborating on a project.
- Pushed and pulled some commits between the two.

What are READMEs?

- **READMEs:**
- A README file is used to communicate important information about a repository including:
    - What the project does.
    - How to run the project.
    - Why it's noteworthy.
    - Who maintains the project.
- **READMEs:**
    - If you put a README in the root of your project, **Github will recognize it and automatically display it on the repo's home page.**
- **README.md:**
    - **READMEs** are Markdown files, ending with the .md extension. Markdown is a convenient syntax to generate formatted text. It's easy to puck up!

A Markdown Crash Course:

- Markdown is a tool that generates “Markup”.
- Project documentation at **Daring Fireball: Markdown**.
    - “Markdown is a text-to-HTML conversion tool for web writers. Markdown allows you to write using an easy-to-read, easy-to-write plain text format, then convert it to structurally valid XHTML (or HTML).”
- Easier than straight HTML script.
- Various tools and extensions out there to help, including an extension for VSCode.
- Interactive screen showing raw markdown on left and generated HTML on the right:
    - markdown-it.github.io
- He gave some examples of how to format different ways using Markdown, but the interactive page also comes with a lot of pre-made examples.
- Lots of functionality. I guess Reddit uses markdown (I’ve seen it when including links). I also wonder if WordPress does (or if there’s a WordPress extension).

Adding a README to a Project:

- Could just make it on Github.
- Or make it locally:
- Ran a **git pull origin main** first just to make sure he had the most up-to-date version of everything (this is good practice).
- **touch README.md**
- Added Markdown in VSCode. It seems he has the extension for that too.
    - VSCode can preview Markdown with the extension:
    - Open the “command pallet”, type “markdown” and click “Open Preview”.
- **git add README.md**
- **git commit -m “add README file”**
- **git push origin main**
- It appears in the Github repo and its contents are displayed on the main page.
- He then went over to his second “pretend” collaborator’s side of things and ran a **git pull origin main** to get the README.md file to the second collaborator’s local repo.

Creating Github Gists:

- **Github Gists** are a simple way to share code snippets and useful fragments with others. Gists are much easier to create, but offer far fewer features than a typical Github repository.
    - PasteBin: similar tool, I guess.
    - gist.github.com, but should also be able to get to there from your Github account.
- He showed someone's Gist that solved the problem of needing to run both their personal and work Github accounts from the same computer.
- There are both secret Gists and public Gists.

Introducing Github Pages:

- **Github Pages** are public webpages that are hosted and published via Github. They allow you to create a website simply by pushing your code to Github.
- Github Pages is a hosting service for statis webpages, so it does not support server-side code like Python, Ruby, or Node. Just HTML/CSS/JS.
    - Can still create things like portfolio websites, documentation websites for repositories, a JavaScript game, etc.
    - Don't use for anything high-capacity where you're expecting a lot of traffic.
- Showed some repositories/pages that he also included as links in the video.
- There are two main flavors of Github Pages:
    - **User Site:**
      You get one user site per Github account. This is where you could host a portfolio site or some form of personal website. The default URL is based on your Github username, following this pattern: **username.github.io** though you can change this!
    - **Project Sites:**
      You get unlimited project sites! Each Github repo can have a corresponding hosted website. It's as simple as telling Github which specific branch contains the web content. The default URLs follow this pattern: **username.github.io/repo-name**.

Github Pages Demo:

- First step: make a repository or pick a repository that we want to make Github Page for.
- He made an empty one called "chickens", then (in a local repo called "GHPages") he ran a **touch chickens.txt**, then **git add chickens.txt**, then **git commit -m "initial commit"**. Listed some chicken breeds in the .txt file. Then **git commit -am "add some chicken breeds"**.
- **git branch -M main**
- **git remote add origin <URL>**, then pushed the local commits to this Github repo. So far, same stuff we've been doing.
- Github pages is going to be looking for an **index.html** file in a branch of the repo. When ready to add a Github Page, you go to settings, go down almost to the bottom (a section called "GitHub Pages", and select source from a drop-down menu. The drop-down menu is a list of branches for the repo, and you should select the branch that contains **index.html**.
- In his local repo, he ran **git switch -c gh-pages** from the main branch.
- Then, **touch index.html**. He also removed the **chickens.txt** from this "gh-pages" branch, then **git add .** and **git commit -m "add index.html"**.
- Created a very basic html file with **<h1>Chickens!</h1>** as the header and some "lorem ipsum" text in a paragraph. Then, **git add index.html** and **git commit -m "add basic html"**.
- Then, **git push origin gh-pages**, adding the new branch with the index.html in it.
- He then went into settings and chose "gh-pages" from the drop-down. Since this is a very conventional name for this, it was automatically selected in settings. In this case he chose the default folder (/root), but you could also choose a "/docs" folder here.
- He then went to a site called **codepen.io** (which contains both some HTML and some CSS) to get some custom code to spruce up his "Chickens" webpage. He replaced the default text with "Chickens!", then chose "Compiled HTML" from a drop-down, then copied it and replaced the body of his local index.html file with that code. He then copied all the associated CSS and pasted that into a newly created CSS file, "**app.css**". Added a link to this CSS file in his index.html file (added **<link rel="stylesheet" href="app.css">**) to the bottom of the **<head>** portion of the HTML file.
- Then **git add .** and **git commit -m "add animated text to index"**.
- Then **git push origin gh-pages** and double-checked that the new app.css file was in the gh-pages branch in Github. Then he refreshed the page for his "chickens" website to show the changes.

Section 14 – Git Collaboration Workflows:

The Pitfalls of a Centralized Workflow:

- **Centralized Workflow:**
- AKA Everyone Works on Master/Main
- AKA the Most Basic Workflow Possible
- The simplest collaborative workflow is to have everyone work on the master branch (or main, or any other SINGLE branch).
- It's straightforward and can work for tiny teams, but it has quite a few shortcomings.
- **The Problem:**
- While it's nice and easy to only work on the master branch, this leads to some serious issues on teams!
    - Lots of time spent resolving conflicts and merging code, especially as team size scales up.
    - No one can work on anything without disturbing the main codebase. How do you try adding something radically different in? How do you experiment?
    - The only way to collaborate on a feature together with another teammate is to push incomplete code to master. Other teammates now have broken code…

Centralized Workflow Demonstration:

- Using his Colt vs Stevie Chicks Github repos again.
- Demonstrated pitfalls mentioned in previous video.
- Created **index.html** as Colt using a "**bootstrap boilerplate**". Got a lot of stuff from "getbootstrap.com". Could be useful to know in the future.

The All-Important Feature Branch Workflow:

- **Feature Branches:**
- Rather than working directly on master/main, all new development should be done on separate branches!
    - Treat master/main branch as the official project history.
    - Multiple teammates can collaborate on a single feature and share code back and forth without polluting the master/main branch.
    - Master/main branch won't contain broken code (or at least, it won't unless someone messes up).

Feature Branch Workflow Demo:

- Demonstrates concepts from previous video, similar to the "Centralized Demonstration".
- Once again, started with the bootstrap website.
- "Stevie" cloned down the basic repo, then created a new branch: **git switch -c navbar**
- Added/committed incomplete work (index.html) as "add broken navbar". Pushed branch "navbar" to Github to get feedback from teammate.
- "Colt" is working on a pricing table: did a **git pull origin main** to double-check for the most recent version/changes to main, then **git switch -c pricing-table**. Added HTML markup straight from bootstrap (went into inspection mode to copy/paste).
- "Colt" gets message from "Stevie" asking for help with navbar. "Colt" runs **git fetch origin** to check for changes, sees branch navbar. Starts by going into detached HEAD to take a look at changes. Runs **git checkout origin/navbar** to do that. Leaves detached HEAD with **git switch -**.
- Runs **git switch navbar** so Git will automatically set branch navbar to track the remote version of the branch. Finds out that the problem was that the navbar requires "bootstrap JavaScript", which wasn't initially included (makes things more interactive).
- Went and got Bootstrap JavaScript file and added it to the bottom of index.html.
- Added and committed changes, -m "fix navbar bug". Pushed branch up to Github, then went back to what he was doing (pricing-table).
- Went back to "Stevie's" machine, pulled down most recent navbar branch.

Merging Feature Branches:

- **Merging in Feature Branches:** At some point, new the work on feature branches will need to be merged into the master branch! There are a couple of options for how to do this:
    - 1. Merge at will, without any sort of discussion with teammates. JUST DO IT WHENEVER YOU WANT.
    - 2. Send an email or chat message or something to your team to discuss if the changes should be merged in.
    - **Pull Requests!**
- Once a branch has been merged into master/main on Github, it can be a good idea to delete that branch locally.

Introducing Pull Requests:

- **Pull Requests** are a feature built into products like Github and Bitbucket. **They are not native to Git itself**.
- They allow developers to alert team-members to new work that needs to be reviewed. They provide a mechanism to approve or reject the work on a given branch. They also help facilitate discussion and feedback on the specified commits.
- "I have this new stuff I want to merge into the master branch…what do you all think about it?"
- **The Workflow:**
    - 1. Do some work locally on a feature branch.
    - 2. Push up the feature branch to Github.
    - 3. Open a pull request using the feature branch just pushed up to Github.
    - 4. Wait for the PR to be approved and merged. Start a discussion on the PR. This part depends on the team structure.

Making Our First Pull Request:

- Went back to where we were during the "Feature Branch Workflow Demo" video (with "Colt" and "Stevie" working on navbar and pricing-table branches).
- As "Stevie Chicks": Clicked on the "Compare" button (next to the "Pull request" button). The comparison page gave a green message saying "Able to merge: These branches can be automatically merged", which is a nice little feature.
- From this "Compare" page, you can click on the green "Create Pull Request" button after reviewing compared changes.
- Switched over to "Colt" and went to the "Pull requests" tab to see a notification of the PR. There was also a notification in the top right of the page. Left a comment on the PR saying "Looks good, merging now", then clicked the "Merge pull request" button.
- Message appeared: "Pull request successfully merged and closed. You're all set—the navbar branch can be safely deleted." Next he hit the "Delete branch" button.
- The **index.html** on main now includes the navbar stuff (including the fix). Team members now need to pull the latest version of main if they want that information to be up-to-date locally.
- On "Colt's" machine: **git pull origin main** to update his local main branch.
- On "Stevie's" machine: **git switch main** (because Stevie was still on their local navbar branch), and then **git branch -D navbar** to delete the local navbar branch. Basic housekeeping. Then, **git pull origin main** to get Stevie's local main branch also updated.

Merging Pull Requests with Conflicts:

- A merge on Github from a PR can have conflicts. They won't always be able to be merged automatically.
- To illustrate: he switched to "Stevie's" machine and created a new branch called new-heading. Stevie then changes a few HTML things for the website heading.
- Stevie adds and commits this to the branch, then pushes the branch up to Github.
- Simultaneously, "Colt" makes some changes to index.html directly on the main branch in Github. He changed the heading "Hello World!" to "Hello there everyone!!!!!", committed the changes directly to the main branch.
- Now when he tries to make a Pull Request as Stevie in Github, there's a new message in red saying "Can't automatically merge. Don't worry, you can still create the pull request".
- After creating the PR, the pull requests tab page now contains a message saying "This branch has conflicts that must be resolved" with a clickable button labeled "Resolve conflicts".
- Theoretically anyone collaborating on this project can resolve conflicts, but to be realistic he switched back to the "Colt" profile for the next step.
- New PR request in "Colt's" workspace with the same info. Clicking on the "Resolve conflicts" button gives the option to make changes in the browser, but he wanted to show the real-world (old-fashioned) way. He clicked on "view command line instructions", which gave directions to bring everything to the local machine.
    - Note: Since he had made his own changes to the heading within the Github main branch, he realized at the "git merge main" step of the following that he needed to pull the updated main first, so that he would have the conflicting info on his local machine.
- As "Colt", he ran **git fetch origin**, which gave info about a new "new-heading" branch.
- The instructions then say to use **git checkout -b new-heading origin/new-heading**, though Colt said that **git switch new-heading** works just fine. The Github website just still says the old way.
- Then on local, **git merge main**, merging the main branch into the new branch to test for the conflicts. He resolved the conflict in VSCode and decided to keep elements of both changes.
- Added and committed changes (-m "resolve conflict") for this new-heading branch, then switched back to main, then: **git merge --no-ff new-heading** to merge the resolved "new-heading" branch back into an updated main branch. The "--no-ff" flag keeps Git from doing a fast-forward, which is better in this case for record-keeping purposes (creates a merge-commit). He then did a **git push origin main**. Github automatically senses this happen and closes the PR, giving the option to delete the branch once again.
- **Concise version:**
    - Switch to the branch in question. Merge in master/main and resolve the conflicts.
    - **git fetch origin**
    - **git switch <my-new-feature>**
    - **git merge master/main**
    - fix conflicts!
    - Switch to master/main. Merge in the feature branch (now with no conflicts). Ush changes up to Github.
    - **git switch master/main**
    - **git merge <my-new-feature>**

    - **git push origin master/main**
- The last thing to do is to switch to “Stevie” and **git pull origin main** to get the latest changes on their local machine.

Configuring Branch Protection Rules:

- In settings: under “Branches” one can add Branch Protection Rules here with the “Add rule” button.
- It gives the option of using a “Branch name pattern”, where branches following a certain naming convention can all have a rule applied to them. This is helpful in repos with thousands of branches, and which use a standardized naming convention for this purpose.
- In this example, we only have main to worry about.
- Checked “Require pull request reviews before merging”. He then clicked the “Create” button.
- Switched over to “Stevie’s” Github to demonstrate these changes in action. He showed the rest of the process within the browser rather than using local.
- After “Colt” did a review on “Stevie’s” changes and approved them (with comment), the option “Merge pull request” was shown next. So there’s approval and THEN there’s merge.

Introducing Forking:

- **Fork & Clone: Another Workflow:**
- The “fork & clone” workflow is different from anything we’ve seen so far. Instead of just one centralized Github repository, every developer has their own Github repository in addition to the “main” repo. Developers make changes and push to their own forks before making pull requests.
- It’s very commonly used on large open-source projects where there may be thousands of contributors with only a couple maintainers.
- **Forking:**
- Github (and similar tools) allow us to create personal copies of other peoples’ repositories. We call those copies a “fork” of the original.
- When we fork a repo, we’re basically asking Github “Make me my own copy of this repo please”.
- As with pull requests, forking is not a Git feature. The ability to fork is implemented by Github.
- In a person’s (public) repo, there is a “Fork” button up in the top right.

Forking Demonstration:

- For this demonstration, he chose to fork that "2048" game.
    - He used this for the "cloning" demonstration, but if one were to simply clone this repo, they'd have no way to push changes up to Github. This is where "forking" comes into play.
- After forking the repo into your own account, you now clone it down to your machine. This sets up the remote so you can push changes.

The Fork & Clone Workflow:

- **Now What?**
- Now that I've forked, I have my very own copy of the repo where I can do whatever I want!
- I can clone my fork and make changes, add features, and break things without fear of disturbing the original repository.
- If I do want to share my work, I can **make a pull request from my fork to the original repo**.
- **Basic Demo:**
- Forked a repo from original source, then cloned to local machine. THEN he added a second remote pointing to the original project repo (NOT the fork). This remote can be named anything, but you'll often see "upstream" or "original" used. This is so that, even though at this stage we're not pushing changes up, we can still pull changes down.
- Next, we can make a pull request from our fork on Github to the original project repository, if we have changes we want merged into the original repo.
- New work that is done to the original repo can be pulled down to our local machine to update it, and then pushed up to our forked repo on Github. A sort of cycle.
- **An Even Briefer Summary:**
    - 1. Fork the project.
    - 2. Clone the fork to local machine.
    - 3. Add upstream remote pointing at original project repo.
    - 4. Do some work.
    - 5. Push to origin (our forked repo).
    - 6. Open pull request to original project.

Fork & Clone Workflow Demonstration:

- **Fork & Clone:**
- This “Fork & Clone” workflow might seem complicated, but it’s extremely common for good reason!
- It allows a project maintainer to accept contributions from developers all over the world without having to add them as actual owners of the main project repository or worry about giving them all permission to push to the repo (which could be disastrous!).
- **Follow-Along:**
- As “Colt” he made a new Github repository without giving anyone a collaborator (no one can push to it). He named the repo “fork-and-clone” and left it public, added a README, and cloned it down.
- Switched over to “Stevie Chicks”, found “Colt’s” “fork-and-clone” repo, and forked it as “Stevie”. Cloned the repo to “Stevie’s” local machine.
- He ran **git remote -v** to illustrate that the remote is still only pointing to the fork on Github, not the original project. He then set up a second remote to the original project by going to the original project in Github, copying the URL, and adding the second remote with:
    - **git remote add upstream <URL>**
- Now with **git remote -v** you see two remotes.
- Switched back to “Colt” and edited the README just for a simple change and made a commit. “Stevie’s” repo doesn’t have these changes, but can still get them.
- Ran **git pull upstream main** as “Stevie” to pull changes from the original project.
- As “Stevie”, made another change to the README, then added and committed. Pushed to Github fork (origin). Message on fork’s repo page says we are now one commit ahead of Colt’s version.
- Next, opened a pull request to the original project.
- As “Colt”, reviewed pull request and confirmed to merge it in. “Colt’s” account now includes Stevie’s changes.

Section 15 – Rebasing: The Scariest Git Command?:

Why is Rebasing Scary? Is It?

- **Rebasing:**
- "When I first learned Git, I was told to avoid rebasing at all costs."
- ("It can really #%@* things up", "It's not for beginners!")
- "So I avoided the **git rebase** command for YEARS!"
- It's actually very useful, as long as you know when NOT to use it!
- There are two main ways to use the git rebase command:
    - As an alternative to merging.
    - As a cleanup tool.

Comparing Merging & Rebasing:

- **Something that rebasing addresses that merging does not:**
- Working on a collaborative project:
- Working on a feature branch, but then some new work gets committed to master. I want that new information on my feature branch as well, so I merge master into my feature branch, creating a new merge commit.
- As I'm doing more work on my feature branch, more work gets done on master. Once again, merge commit master into feature.
- The feature branch now has a bunch of merge commits. If the master branch is very active, my feature branch's history is muddied. You end up with a lot of merge commits that don't actually say what it is YOU'RE working on. Not informative.
- **Rebasing!**
    - We can instead rebase the feature branch onto the master branch. This moves the entire feature branch so that it BEGINS at the tip of the master branch. All of the work is still there, but **we have re-written history**.
- Instead of using a merge commit, rebasing rewrites history by **creating new commits** for each of the original feature branch commits.
    - **git switch feature**
    - **git rebase master**
- We can also wait until we are done with a feature and then rebase the feature branch onto the master branch.

Rebase Demo Pt 1: Setup & Merging:

- Created "website.txt" to act as a simple pretend log of what has been worked on. Added "NAVBAR ADDED!" to the .txt file, added and committed this.
- Created branch "feat" and created "feature.txt" in that. Using numbers (ONE, TWO, etc) to track changes there. This is the pretend work we're doing on feature branch.
- **git switch master**, "FOOTER ADDED!", add and commit.
- **git switch feat**, then **git merge master** to merge master into feat. This illustrates the first case presented in the previous video, the messy way.
- Continued in this way for a few more rounds to show how messy the history can look after a while. This can be even worse with a very large project.

Rebasing Demo Pt 2: Actually Rebasing:

- Picking up from the last video:
- **git switch feat**, or make sure you're already on that branch.
- **git rebase master**.
- In GitKraken, the structure has now changed and is linear (but there are still two branches, technically). Branch feat is tacked onto the end of branch master.
- He then did a new commit on the master branch, so feat doesn't have that work. The branches are now more obviously split (two branches) in GitKraken.
- Rebase again to move feat to the top of the master branch again. You end up with all the code for both, but the history gets updated.

The Golden Rule: When NOT to Rebase:

- **Why Rebase?**
- We get a much cleaner project history. No unnecessary merge commits! We end up with a linear project history.
- **WARNING!**
- Never rebase commits that have been shared with others. If you have already pushed commits up to Github…DO NOT rebase them unless you are positive no one on the team is using those commits.
- You want to rebase feature branches you have on your own machine. You don't want to rebase the master branch. (This is why we "merge/rebase" the master branch INTO our feature branch, even though it will show the feature branch moved ONTO the tip of the master branch. Or rather, we are in our feature branch and then we rebase that onto master).

Handling Conflicts & Rebasing:

- In feat branch he changed a few lines in “website.txt”, added, committed.
- Switched to master branch and changed lines in the same .txt. These two .txt files will now disagree.
- Switched to feat branch and added a line to “feature.txt”.
- On feat, **git rebase master** to get that new work, except we get a message saying there are conflicts. GitKraken shows a “partial rebase”, since it managed to rebase a few of the commits.
    - Note: The error message includes instructions for undoing a rebase if you want to at this stage: **git rebase --abort**.
- Differences in VSCode are shown the same way as regular merge conflicts, and you can manually combine/change.
- Error message includes instructions for once you’ve resolved conflicts: **git rebase --continue**. This completes the rebase.
- But first, **git add <file>** to add to staging area, then **git rebase --continue** (instead of the usual git commit) to finish rebase.
- The rebase is now complete, and GitKraken shows a linear history.

Section 16 – Cleaning Up History with Interactive Rebase:

Introducing Interactive Rebase:

- **Rewriting History:**
- Sometimes we want to rewrite, delete, rename, or even reorder commits (before sharing them). We can do this using **git rebase**!
- **Interactive Rebase:**
- Running **git rebase** with the **-i** option will enter the interactive mode, which allows us to edit commits, add files, drop commits, etc. Note that we need to specify how far back we want to rewrite commits.
- Also, notice that we are not rebasing onto another branch. Instead, we are rebasing a series of commits onto the HEAD they are currently based on.
    - For example: **git rebase -i HEAD~4**

Rewording Commits with Interactive Rebase:

- Included in this video is a link to a Github repo to practice interactive rebase on. The goal is to use interactive rebase to clean things up in a somewhat messy commit log.
- Started by cloning project down.
- Ran **git log --oneline** to see history (same history as online, except there was also a README file added).
- Created branch my-feat.
- Ran **git rebase -i HEAD~9** (because that's how many commits backward he wanted to start). This opens up a text editor.
- **What Now?**
- In our text editor, we'll see a list of commits alongside a list of commands that we can choose from. Here are a couple of the more commonly used commands:
    - **pick** – use the commit.
    - **reword** – use the commit, but edit the commit message.
    - **edit** – use commit, but stop for amending.
    - **fixup** – use commit contents but meld it into previous commit and discard the commit message.
    - **drop** – remove commit.
    - More options are listed in the interactive text editor, down below the commits.
- In this video, I guess we're focusing on "**reword**".
- By default, all the options are set to "pick". Also, note that the order of the commits is reversed compared to running git log.

- Changed "pick cbee26b I added project files" to "reword cbee26b I added project files", then simply saved and closed the window. A "COMMIT_EDITMSG" window then popped up, where he entered "add project files" and then closed out of the window again.
- Next, ran **git log --oneline**, and the commit message had been updated. All of the commit hashes from that commit onward is now different.
- More interactive rebasing will continue in next video.

Fixing Up & Squashing Commits with Interactive Rebase:

- Picking up where the last video left off:
- The plan is to combine the commits "add bootstrap" and "whoops forgot to add bootstrap js script" into one commit.
- We're going to use "**fixup**" this time.
- Going to run **git rebase -i HEAD~9** again (even though we don't necessarily need to go back a whole 9 spaces, it's just a quick way to bring them all up again).
    - "Fixup" works like "squash", except fixup discards the extra commit log message and squash does not.
- Found the commit to be changed, changed "pick" to "fixup", saved and closed the window.
- Message in terminal says "Successfully rebased", and then he ran **git log --oneline** to check/show the changes to the log.
- Repeated the same steps, combining three commits involving the navbar.

Dropping Commits with Interactive Rebase:

- In this case, we want to get rid of both a commit AND its changes. In this case, the accidental commit with the message "my cat made this commit".
- The "app.js" file in this commit has a bunch of gibberish symbols, like if a cat walked on a keyboard for example.
- He ran **git log --oneline** to see how far back to go (just two commits in this case).
- He then ran **git rebase -i HEAD~2**, and changed "pick" to "drop", then saved and closed out.
- Running **git log --oneline** now shows that the commit is gone, and checking the "app.js" file shows that the gibberish is gone.
- As an aside, he also noticed that the most recent commit ("Create the README") was the only commit starting with a capital letter. One could use interactive rebase with the "reword" function again, or (going back a long time in this course) use **git commit --amend** to change it, since it's the most recent.

Section 17 – Git Tags: Marking Important Moments in History:

The Idea Behind Git Tags:

- **Git Tags:**
- Tags are pointers that refer to particular points in Git history. We can mark a particular moment in time with a tag Tags are most often used to mark version releases in projects (v4.1.0, v4.1.1, etc.)
- Think of tags as branch references that do NOT CHANGE (unlike HEAD or "master", for example). Once a tag is created, it always refers to the same commit. It's just a label for a commit.
- **The Two Types:**
- There are two types of Git tags we can use: lightweight and annotated tags.
- **Lightweight tags** are…lightweight. They are just a name/label that points to a particular commit.
- **Annotated tags** store extra metadata including the author's name and email, the date, and a tagging message (like a commit message). Preferred in bigger projects, because that metadata is useful, yo.

A Side Note on Semantic Versioning:

- **Semantic Versioning:**
- The semantic versioning spec outlines a standardized versioning system for software releases. It provides a consistent way for developers to give meaning to their software releases (how big of a change is this release??)
- Versions consist of three numbers separated by periods (i.e., **2.4.1**).
    - 2.4.1 broken down:
    - **2**: major release
    - **4**: minor release
    - **1**: patch
- Initial Release: Typically, the first release is **1.0.0**
- Patch Release: (i.e., **1.0.1**) Patch releases normally do not contain new features or significant changes. They typically signify bug fixes and other changes that do not impact how the code is used.
- Minor Release: (i.e., **1.1.0**) Minor releases signify that new features or functionality have been added, but the project is still backwards compatible. No breaking changes. The new functionality is optional and should not force users to rewrite their own code.
- Major Release: (i.e., **2.0.0**) Major releases signify significant changes and is no longer backwards compatible. Features may be removed or changed substantially.

Viewing & Searching Tags:

- **Viewing Tags:**
- **git tag** will print a list of all the tags in the current repository.
- In this video, he's going to clone the React repo from Facebook in order to show all the tags.
- We can search for tags that match a particular pattern by using **git tag -l** and then passing in a wildcard pattern. For example, **git tag -l "*beta*"** will print a list of tags that include "beta" in their name. Another example he used: **git tag -l "v17*"**, which only brought up a few.

Comparing Tags with Git Diff:

- **Checking Out Tags:**
- To view the state of a repo at a particular tag, we can use **git checkout <tag>**. This puts us in detached HEAD!
- **Comparing Tags:**
- **git diff** can be used to compare changes between different tags, such as different patches.
- Examples: **git diff v17.0.0 v17.0.1** and **git diff v16.14.0 v17.0.0**.

Creating Lightweight Tags:

- To create a lightweight tag, use **git tag <tag-name>**. By default, Git will create the tag referring to the commit that HEAD is referencing.
- The tag names used in this video followed semantic versioning and were called 17.0.2 and 17.0.3.

Creating Annotated Tags:

- Use **git tag -a** to create a new annotated tag. Git will then open your default text editor and prompt you for additional information.
- Similar to git commit, we can also use the **-m** option to pass a message directly and forgo the opening of the text editor.
    - "**git tag -a <tag-name>**" or "**git tag -am <tag-name> "<tag message>""**
    - Note: He didn't actually say where to put the message if using "-m" flag, so the above is a guess on my part.
- To see the metadata from an annotated tag, we use **git show <tag-name>** (in the video example, **git show v17.1.0** after he made a "minor release" of a tag and commit).
    - This shows the tagger, the date, the tag message, and the commit.

Tagging Previous Commits:

- So far we've seen how to tag the commit that HEAD references. We can also tag an older commit by providing the commit hash: **git tag -a <tag-name> <commit-hash>**.
- In his example, he ran git log --oneline to choose the commit hash from a previous commit.

Replacing Tags with Force:

- **Forcing Tags:**
- Git will yell at us if we try to reuse a tag that is already referring to a commit. If we use the **-f** option, we can FORCE our tag through.
    - **git tag -f <tag-name>**
- This moves the tag to somewhere else.

Deleting Tags:

- To delete a tag, use **git tag -d <tag-name>**.

IMPORTANT: Pushing Tags:

- **Pushing Tags:**
- By default, the **git push** command doesn't transfer tags to remote servers. If you have a lot of tags that you want to push up at once, you can use the **--tags** option to the **git push** command. This will transfer all of your tags to the remote server that are not already there.
    - **git push --tags**
- You can also push a single tag.
    - **git push <remote-name> <tag-name>** (such as **git push origin v1.5.0**)

Section 18 – Git Behind the Scenes – Hashing & Objects:

Working with the Local Config File:

- Included link: Git Config Docs.
- What is in **.git**??
    - Folders: **objects**, **refs**
    - Files: **config**, **HEAD**, **index**
    - There's more, but this is the juicy stuff.
- **Config:**
- The config file is for…configuration. We've seen how to configure global settings like our name and email across all Git repos, but we can also configure things on a per-repo basis.
- The config file is kept in the hidden **.get** folder within a given repo.
    - Note: He had cd'd into the hidden .git folder at the time of running the following:
- **git config user.name** shows configured global name (← we set this at the beginning of the course).
- He used the command **git config user.name "asknfkjgjk"** to illustrate that when we set this as a name, we're setting that value within the config file.
- The command **git config --local user.name** prints nothing out, because we haven't set a local name for this file, but running **git config --local user.name** "chicken little" will set the local username for just that repo to "chicken little".
- Similarly, running **git config --local user.email "chicken@gmail.com"** will change the local email for that repo.
- He says that a good way to play around with the config file to find useful settings is to look around the internet for what other Git users are doing (they often post copies of their config files online for reasons like this).

- For the next example, he decided to play around with "color" settings.
- Manually editing the "config" file:
    - At the bottom, added:
    - [color]
        - ui = true
    - [color "branch"]
        - local = cyan bold
- He then saved and ran **git branch** again to show the color changes.
    - Added:
    - [color]
        - ui = true
    - [color "branch"]
        - local = cyan bold
        - current = yellow bold
    - [color "diff"]
        - old = magenta bold *(← default was red before)*

Inside Git: The Refs Directory:

- **Refs Folder:**
- Inside of refs, you'll find a heads directory. **refs/head** contains one file per branch in a repository. Each file is named after a branch and contains the hash of the commit at the tip of the branch.
    - For example, **refs/heads/master** contains the commit hash of the last commit on the master branch.
- Refs also contains a **refs/tags** folder which contains one file for each tag in the repo.
- He then ran **ls .git/refs** which printed out the heads, remotes, tags directories. He also noted that if you run these sorts of commands and look what's in an empty/new repo, the contents are quite different.
- He also showed many of these files and folders within VSCode to give a visual idea of how this all works. The "head" file just points to a commit hash, the most recent one on a given branch.

Inside Git: The HEAD File:

- **HEAD:**
- HEAD is just a text file that keeps track of where HEAD points.
- If it contains refs/heads/master, this means that HEAD is pointing to the master branch.
- In detached HEAD, the HEAD file contains a commit hash instead of a branch reference.

Inside Git: The Objects File:

- **Objects Folder:**
- The objects directory contains all the repo files. This is where Git stores the backups of files, the commits in a repo, and more. It's really the core of Git.
- The files are all compressed and encrypted, so they won't look like much!
    - Looking inside of the objects directory in VSCode shows a bunch of two-digit hexadecimal files inside (longer Git history = more folders in here). Each folder contains a single file, named (what looks like) commit hashes. Trying to open one of these files gives an error, saying that VSCode can't read them. If you try anyway, you get a bunch of gibberish symbols. This is the compression/encryption, as binary files.

A Crash Course on Hashing Functions:

- Included link: SHA-1 Demo.
- Hexadecimal characters, 40 characters long, Base-16. (Characters 0-9, a-f).
- **Hashing Functions:** Functions that map input data of some arbitrary size to fixed-size output values.
- **Cryptographic Hash Functions:** A subset of hash functions.
    - 1. One-way function which is infeasible to invert.
    - 2. Small change in input yields large change in output.
    - 3. Deterministic – same input yields same output.
    - 4. Unlikely to find 2 outputs with same value.
- **SHA-1:**
- Git uses a hashing function called SHA-1 (though this is set to change eventually).
- SHA-1 always generates 40-digit hexadecimal numbers.
- The commit hashes we've seen a million times are the output of SHA-1.

Git as a Key-Value Datastore:

- **Git Database:**
- Git is a **key-value data store**. We can insert any kind of content into a Git repository, and Git will hand us back a unique key we can later use to retrieve that content.
- These keys that we get back are SHA-1 checksums.
    - Remember: **4 Types of Git Objects:** commit, tree, blob, annotated tag.
- Hashes are keys and contents are values.
- Git actually uses hashes all over the place, not just as commit hashes. Hashes are actually central for how Git works internally.

Hashing with Git Hash-Object:

- **Let's Try Hash(ing):**
- **echo 'hello' | git hash-object –stdin** (← is the command he had at the top of this slide.)
- The **--stdin** option tells git hash-object to use the content from stdin rather than a file. In our example, it will hash the word 'hello'.
- The echo command simply repeats whatever we tell it to repeat to the terminal. We pipe the output of echo to **git hash-object**. (I remember "echo" and "pipe" from Linux).
    - Note: He never really uses this function aside from when he's teaching.
- **Let's Try Hash(ing):**
- The **git hash-object** command takes some data, stores it in our ".git/objects" directory, and gives us back the unique SHA-1 hash that refers to that data object.

- In the simplest form (shown on the right), Git simply takes some content and returns the unique key that WOULD be used to store our object. But it does not actually store anything.
    - What he referred to as "shown on the right": **git hash-object <file>**
- **Let's Try Hash(ing):**
    - **echo 'hello' | git hash-object --stdin -w**
- Rather than simply outputting the key that Git would store our object under, we can use the **-w** option to tell Git to actually write the object to the database.
- After running this command, check out the contents of **.git/objects**.
    - Hash output starts out as "**ce*013625***…"
    - New "**ce**" folder within the .git/objects folder, its contents are a file called "***013625***…"
    - The folder is the first **2** digits, and the file is the remaining ***38*** digits.

Retrieving Data with Git Cat-File:

- **Let's Try Hash(ing):**
    - **git cat-file -p <object-hash>** (← is the command he had at the top of the slide.)
- Now that we have data stored in our Git object database, we can try retrieving it using the command **git cat-file**. (This reminds me of the regular "cat" command in Linux).
- The **-p** option tells Git to pretty-print the contents of the object based on its type.
- For example involving files:
- Created a new file, **touch dogs.txt**, then listed two of his dogs in the .txt file.
- Ran **git hash-object dogs.txt** to get the hypothetical hash for that file. He then added **-w** to store the hash.
- Note: We haven't made any commits yet in this exercise.
- Added two more dogs to his **dogs.txt** file. When he hashed it, the hash is now very different. He stored that one too. There's now a new folder/file combo added in .git/objects.
- Using **git cat-file <first dog hash> -p** and got the first version of the .txt file and used **git cat-file <second dog hash> -p** and got out the second version of the .txt file. This is how Git stores all these different versions.
- He then showed an example where he had deleted the contents of **dogs.txt**, but then ran command **git cat-file -p <second dogs hash> > dogs.txt** to output the contents of the hash back into the dogs.txt file, and it restored its contents. This is how Git restores old files.

Deep Dive into Git Objects: Blobs:

- Everything we've stored so far before are examples of one type of Git object: blobs.
- **Blobs:**
- Git blobs (binary large object) are the object type Git uses to store the **contents of files** in a given repository. Blobs don't even include the filenames of each file or any other data. They just store the contents of the file.
- The 38-digit files stored in those 2-digit folders (the hashes) are all blobs. They specifically store the content.

Deep Dive into Git Objects: Trees:

- Blobs are just the content of the files, but what about the relationships and structures between the files?
- **Trees:**
- Trees are Git objects used to store the contents of a directory. Each tree contains pointers that can refer to blobs and to other trees. Every tree has its own hash.
- Each entry in a tree contains the SHA-1 hash of a blob or tree, as well as the mode, type, and filename.
- He included a handy little diagram around 3:01:
    - Tree:
        - Blob: (index.html (← name))
        - Blob: (main.js (← name))
        - Tree: (styles (← name))
            - Blob: (app.css (← name))
            - Blob: (nav.css (← name))
    - To the left of the diagram was a big folder containing "index.html" and "main.js" as files, as well as a folder called "styles" which contained the files "app.css" and "nav.css".
- He also mentioned that the blob doesn't store a name, the tree stores the name.
- **Viewing Trees:**
    - **git cat-file -p master^{tree}** (← is the command he had at the top of this slide)
- Remember that **git cat-file** prints out Git objects. In this example, the **master^{tree}** syntax specifies the tree object that is pointed to by the tip of our master branch.
- Back in the React repo:
- He ran the above command, which printed out a lot of stuff. For every file, it lists whether it's a tree or a blob, the hash, and the file name.
- Also, running **git cat-file -t <hash>** it will tell the object type (i.e., whether it's a blob or a tree).
- One can think of a tree as a folder.

Deep Dive into Git Objects: Commits:

- Lots of good diagrams in this video.
- **Commits:**
- Commit objects combine a tree object along with information about the context that led to the current tree. Commits store a reference to parent commit(s), the author, the committer, and of course the commit message!
- When we run **git commit**, Git creates a new commit object whose parent is the **current HEAD commit** and whose tree is the current content of the index (the staging area).
    - Think of a commit's tree as the folder containing all the contents of that commit.
    - A commit contains a tree, which can contain trees and blobs.

- At this point in the video, he made a basic initial commit of his dogs.txt file (he hadn't committed that prior to this video). He then copied the commit hash found in git log, then ran that hash through **git cat-file -t <hash>**, and we can see that we now have a commit object.
- He then ran **git cat-file -p <hash>**, and we can now see the commit's tree, and its hash.
- He then ran **git cat-file -p <tree's hash>**, and we can now see the tree's content is a blob with its own hash-name, dogs.txt.
    - Commit: (initial commit)
        - Tree: (container for blob)
            - Blob: (dogs.txt)
- He then created a new **cats.txt** file in the command line by running **echo "meow meow" > cats.txt** just to be quick.
- He then ran status, then added and committed. There are now two commits.
- **git log** now shows two commits. He copied the hash for the new commit (<second commit hash>).
- Running **git cat-file -t <second commit hash>** shows us that its object type is a commit, and running **git cat-file -p <second commit hash>** shows that (unlike the initial commit) this one has a parent commit. The parent commit's hash is the hash of the previous/initial commit.
- Running **git cat-file -p <tree hash>** shows that now we have TWO blobs/hashes, one for dogs.txt and one for cats.txt. However, since we didn't make any changes to the dogs.txt file, it's the same object and it has the same hash as it did in the previous commit.

Section 19 – The Power of Reflogs – Retrieving "Lost" Work:

Introducing Reflogs:

- **Reflogs:**
- Git keeps a record of when the tips of branches and other references were updated in the repo.
- We Can view and update these reference logs using the **git reflog** command.
- In this video, he chose to use that same React repository that he cloned for few previous sessions, because it has a long history.
- He went into the **.git** folder and ran **ls**, and in that folder there's a directory called **"logs"**.
- He also showed the **.git** folder open in VSCode and opened up the "logs" folder inside there. Inside were two items: a **"refs"** folder and a file called **"HEAD"**. This "HEAD" file is actually readable; it's not compressed or binary (though it is a little hard to read). Each entry in this file is a "move" for the HEAD reference.
- Can use this log to find old commit hashes that we may no longer see in our git log command.

The Limitation of Reflogs:

- **Limitations:**
- Git only keeps reflogs on your **local activity**. They are not shared with collaborators.
- Reflogs also expire. Git cleans out old entries after around 90 days, though this can be configured.

The Git Reflog Show Command:

- **Git Reflog:**
- The **git reflog** command accepts subcommands **show**, **expire**, **delete**, and **exists**. "Show" is the only commonly used variant, and it is the default subcommand.
- **git reflog show** will show the log of a specific reference (it defaults to HEAD).
- For example, to view the logs for the tip of the main branch we could run **git reflog show main**.

Passing Reflog References Around:

- **Reflog References:**
- The syntax to access specific git refs is **names@{qualifier}**. We can use this syntax to access specific ref pointers and can pass them to other commands including checkout, reset, and merge.
    - For example, **HEAD@{2}** is the same as saying "where HEAD was 2 moves ago".
    - Can pass things into **git reflog show HEAD** like **git reflog show HEAD@{10}**, this will give everything from the beginning of the repo to the point we've chosen (in the example case, HEAD@{58} to HEAD@{10}).
- To see something two moves ago, we can pass this syntax into the git checkout command, such as: **git checkout HEAD@{2}**. This may not even be a different commit, it could just be us switching branches.
    - This is different than **git checkout HEAD~2** which shows the parent commit of the parent commit of HEAD.
- We can also pass this syntax to other commands, such as **git diff HEAD@{0} HEAD@{5}** to see what has changed between those.

Time-Based Reflog Qualifiers:

- **Timed References:**
- Every entry in the reference logs has a timestamp associated with it. We can filter reflog entries by time/date by using time qualifiers like:
    - 1.day.ago
    - 3.minutes.ago
    - Yesterday
    - Fri, 12 Feb 2021 14:06:21 -0800
- On the right side of this slide, he included more examples like:
    - git reflog master@{one.week.ago}
    - git checkout bugfix@{2.days.ago}
    - git diff main@{0} main@{yesterday}
- Say you need to see what things looked like one week ago, because you know that things were working one week ago:
    - **git checkout master@{1.week.ago}**

Rescuing Lost Commits with Reflogs:

- **Reflogs Rescue:**
- We can sometimes use reflog entries to access commits that seem lost and are not appearing in git log.
- For this video, he created a simple git repo from scratch rather than using that complicated React repo that he'd cloned.
- **touch veggies.txt**, then he added and committed. He's planning on doing two or three commits here.
- After committing, he added the following list to **veggies.txt**—
    - Broccoli
    - Rainbow Chard
    - Kale
- —and then added and made a second commit with the message "plant winter veggies.
- He then added—
    - Cabbage
    - Arugula
- —and added/committed again with the message "add more greens".
- He then added—
    - Watermelon
    - Pumpkins
    - Squash
- —and he "got ahead of himself" and added/committed them with the message "add summer seeds".
- Then he realized he should mix the summer seeds with his earlier winter veggies, decided he wanted to get rid of the previous commit.
- He ran **git log --oneline** to see which point he wanted to go back to and chose the hash for "add more greens".
- He then ran **git reset --hard <hash>** to get back to that point. This gets us back to the earlier point and we lose the work we had done in our working directory (the summer seeds).
- At this point his hypothetical became "oh, I can still start those summer seeds, I have a greenhouse!" in which case we want those changes to be put back.
- That deleted commit no longer shows up in git log, but it will show up in git reflog.
- Running **git reflog show master** shows the commit hash for the reset commit in the second line. Its position is "master@{1}" and its hash is available for us. Running **git checkout <hash>** puts us in detached HEAD but also brings those changes back into our working directory. He then switched out of detached HEAD.
- Rather than using checkout, we can restore the old work by running either:
    - **git reset --hard master@{1}**, or
    - **git reset --hard <hash>**
- …both of which move the HEAD, and the new tip of the master branch is whatever master@{1} is, in the reflog.
- Remember, this only works for local changes, and reflogs DO expire after 90 days.

Undoing a Rebase w/ Reflog – It's a Miracle!

- For this video, he began by building off of the files and branch from the previous video, which I believe were just **veggies.txt** and master branch. His plan is to do a rebase and then show that they can be undone.
- He started by creating a new branch, **git switch -c flowers**, then in VSCode he created a new empty **flowers.txt** file to the working directory. He then did an add and commit to this new flowers branch.
- Added flowers to the new .txt file—
    - Salmon Zinnias
    - Queen Lime Zinnias
- —and then he did a **git commit -am "add some zinnias"**.
- Then he added—
    - Old Rose Dahlias
    - Dinnerplate Dahlias
- —and **git commit -am "add dahlias to flowers list"**.
- Then he added—
    - Coral Fountain Amaranth
    - Hot Biscuits Amaranth
- —and **git commit -am "add amaranth to flowers"**.
- Then he added—
    - Coral Reef Celosia
- —and **git commit -am "add celosia"**.
- At this point he said that his eventual plan was to merge this flowers branch into master, but before he does that, he's going to do an **interactive rebase** to get rid of some of these commits, or to combine them; generally just to clean things up.
- He ran **git log --oneline** before just to show all our commits and hashes, then decided that he's going to rebase back to the "add flowers file" initial commit, or **HEAD~4** (4 commits ago, or 4 commits before HEAD~0 or HEAD).
- Ran **git rebase -i HEAD~4**, which opened up a file in VSCode to do the interactive rebasing in. He then switched "pick" to "fixup" for the bottom three commits in the list in order to squash them together with the top commit and discard the commit messages.
- Running **git log --oneline** again just shows two commits in flowers now, the "add some zinnias" one and the initial "add flowers file" one.
- He decided to repeat the process by running **git rebase -i HEAD~2** and switching the "add some zinnias" one from "pick" to "fixup" to squash all the flower commits together. He also chose to use "reword" on the initial commit, which opened up a window to change the commit message to "add list of flower seeds".
- Now when we do a **git log --oneline**, we just have a single commit on the flowers branch saying "add list of flower seeds".
- At this point, what if we want to bring back some of those individual commits or commit messages???

- **Well:**

- At this point he ran **git reflow show flowers**, which still shows all of those individual commits, the commit hashes, and the commit messages.
- Let's say we want all five of those commits that are gone now to be brought back:
- We chose the commit hash for the last of the commits ("add celosia") or we could copy "flowers@{2}" in this case as well.
- We then run **git reset --hard <hash>**, and we now get a message saying "HEAD is now at <hash> add celosia".
- Now if we run **git log --oneline**, that commit and all of the previous commits leading to it are back in the log.
- We rebased to rewrite history, the commits were lost, and then we were able to "rescue" all of those commits using **reflog**.

Section 20 – Writing Custom Git Aliases:

The Global Git Config File:

- **Global Git Config:**
- Git looks for the global config file at either **~/.gitconfig** or **~/.config/git/config**. Any configuration variables that we change in the file will be applied across all Git repos.
- We can also alter configuration variables from the command line if preferred.
- Running—
    - **git config --global user.name**, or
    - **git config --global user.email**
- —will retrieve those respective values, but we can also change those values in the command line. However, this can be a little tedious.
    - Ex.: **git config --global user.name "<name>"** will change the global name in the command line.
- Instead of changing those values in the command line, we can open up the global config file and change any number of values at once.
- He ran **cat ~/.gitconfig** to show what the config file currently has in it.
- He also opened .gitconfig in VSCode to show that we can make changes here and save the file.

Writing Our First Git Alias:

- **Adding Aliases:**
- We can easily set up Git aliases to make our Git experience a bit simpler and faster.
- For example, we could define an alias "**git ci**" instead of having to type "git commit".
- Or, we could define a custom **git lg** command that prints out a custom-formatted log.
- Example on the right of the slide:
    - In .gitconfig file:
    - [alias]
        - s = status
        - l = log
- To start, we need to be in a Git repo (for this video/example, at least). I created an empty one in my "git_course" folder for this.
- In my system's "Users" directory (the default directory that Git Bash always opens on), running **ls -ltra** shows a hidden file: **.gitconfig**. From here, we can run **code .gitconfig** to open the global config file in VSCode.
    - Note: My .gitconfig file is quite sparse compared to his… But then again, he probably uses Git a lot and has added useful options to his .gitconfig file over time.

Setting Aliases from the Command Line:

- We can set aliases with the command line by using (for example:
    - **git config --global alias.showmebranches branch** (which is something silly, making the alias longer than the regular command.
        - Note: the "alias.etc" section of this corresponds to [alias] in .gitconfig, similar to how "user.name" corresponds to the [user] section of .gitconfig.
    - Under the [alias] section of .gitconfig, "showmebranches" has now been added at the bottom of the list.

Aliases with Arguments:

- Examples of arguments:
    - **git branch <branch-name>**, or
    - **git commit -m "commit message"**
- How do we set up aliases to use arguments?
- Setting up "**cm**" as an alias for **git commit -m**, git will automatically append the commit message to the end. No further setup required.
- It is very easy to define aliases that take arguments.

Exploring Existing Useful Aliases Online:

- Links included with this video:
    - https://github.com/GitAlias/gitalias
    - https://www.durdn.com/blog/2012/11/22/must-have-git-aliases-advanced-examples/
    - https://gist.github.com/mwhite/6887990
    - These are all examples of useful aliases that have been shared online.
- There are some very useful and complicated aliases out there for use.
- Note: When you see something like "**!**git config (etc)", the exclamation point tells Git that these are shell scripts.

Personal Note:

Brad mentioned that the following subjects are considered impressive to know in Git/Github by employers:

- **git aliases:** this will be covered at the end of the course anyway.
- **git submodules:** unknown. Brad says that "sub-modules are just saved as a hash of the latest referenced commit for that sub-project for instance".
- **"cherry-picking":** unknown
- **merge resolutions:** I think we already covered this.