# Git and GitHub: An Individual's Perspective

- Git and GitHub are two of the—if not *the* two—most widely used tools in the software development world. Nearly every meaningful software project in the world uses these tools or similar ones.

- At a conceptual level, these tools are straightforward, but each of them offers a somewhat dizzying array of features that can make them seem complex. There's a lot you can do with both Git and GitHub if you know how to use them well, but learning to use them well can sometimes seem daunting.

- However, because these tools are so central, it's important to understand them well and to be able to use them effectively, since you'll likely be expected to use them when you begin contributing to real-world software projects. In fact, we'll use both of these tools heavily in this course. Thus, we'll spend some time right at the outset of the course investigating how to use Git and GitHub.

- Both Git and GitHub offer features designed to make it easy to collaborate on a software project with a team of other developers. For now, though, we'll study how to use these tools from the perspective of an individual developer working on a software project without any collaborators. Later in the course, we'll come back to explore how to use Git and GitHub effectively as part of a team-based project.

- The first thing to know is that, while they're often used together, Git and GitHub are separate tools, so we'll explore each one separately. We'll start off with Git.

## What is Git?

- Git is a specific kind of tool known as a ***version control system*** (or ***VCS***). **Its job is simply to record the changes to a set of files over time** and to provide access to the previous versions of those files.

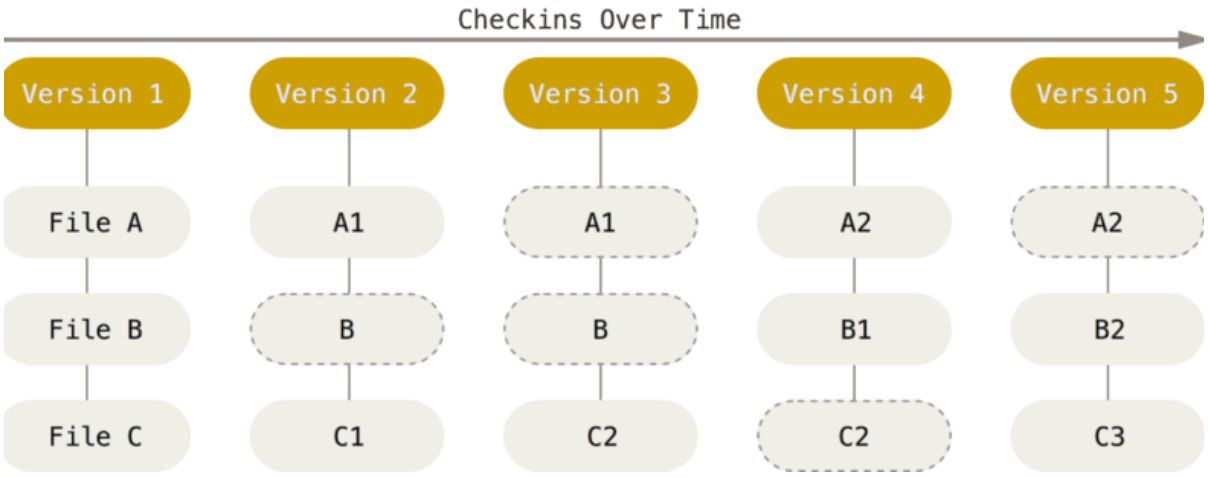
- Using a VCS to maintain the history of changes to a set of files opens up a number of possible very useful actions:
    - Revert one or more files to a previous version.
    - Compare changes over time.

    - Synchronize the version history across multiple locations/collaborators.
    - Identify when and/or by whom a particular change was made.
    - And more.

- There are many different VCSs. One of the most important differences between Git and other VCSs is the way Git stores the version history of a project. Specifically, Git represents version history data **as a series of snapshots** of the project.

- When you record a new version with Git (called "making a ***commit***" or just "committing" in Git terminology), Git essentially stores a snapshot of what all your project files look like at that moment, and, to Git, the version history is like a stream of snapshots.
    - Of course, Git is designed to store these snapshots efficiently. For example, if a file doesn't change from one version/commit to another, then a copy of that file isn't included in the snapshot, only a link to the previous (identical) version of that file.

- This stream of snapshots is stored in a kind of database called a ***Git repository*** (or just "***repo***") that lives directly within the corresponding project directory and conceptually looks something like this:

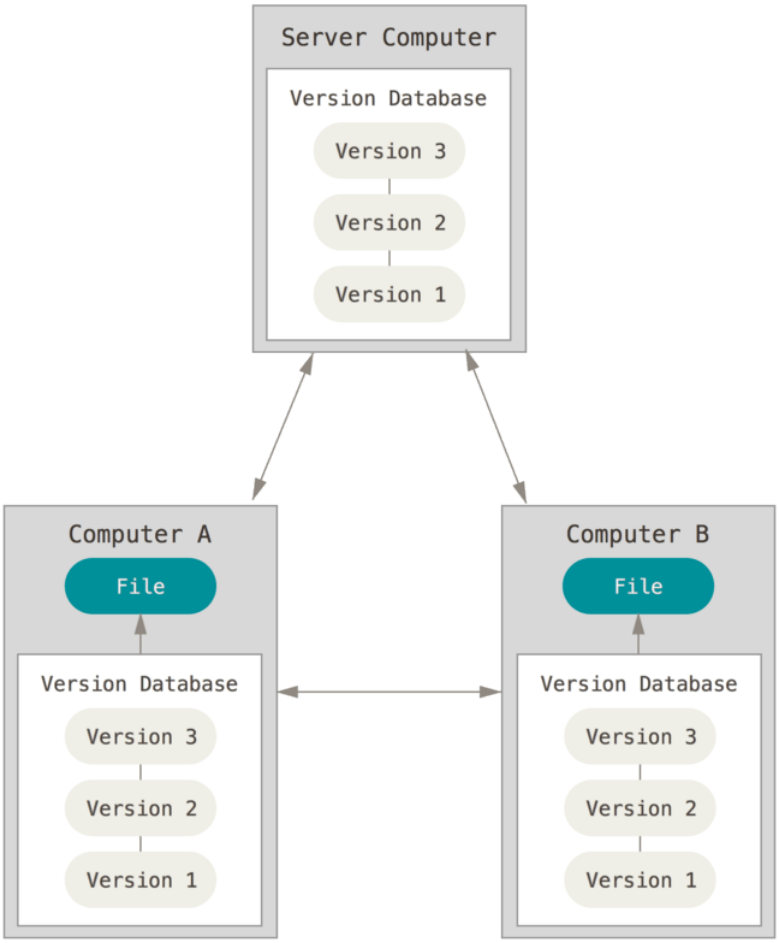



From Pro Git

- Representing the data as a stream of snapshots opens up new, powerful possibilities, such as ***branches***, which we'll explore later.

- Another key difference between Git and many other VCSs is that **Git is a distributed VCS**. This means that every location/computer where the version

history of a particular project lives fully mirrors the *complete* history of that project, i.e. every version/commit/snapshot.  This looks something like this:




From Pro Git

- This aspect of Git can be both a blessing and a curse.  One of its major benefits is that every copy ("***clone***" in Git terminology) of a Git repository acts as a complete backup of a project's version history, making the project more resilient to the failure of an individual machine.

- However, because every clone of a Git repository represents the full version history of the project (i.e. the entire stream of snapshots), special care must be taken to make sure the stream of snapshots does not wildly diverge between the multiple clones.

- For example, if developer A makes several commits within their clone of a repo while at the same time developer B makes several commits within their clone of the same repo, then those two developers have actually created **two different**

**versions of that project's history**. These two versions of the history can be reconciled (often easily), but this can be challenging if they have diverged long from their common starting point in the project's version history or if they conflict with each other.

- Thus, special workflows are often helpful when working on a Git project with other collaborators to help ensure that the work of different developers can be easily reconciled together into a coherent single product. We will discuss some of these workflows later in the course.

- One last important thing to know about Git is that **all of the version history data about a project and nearly all Git operations are local.** This means that you can still continue to work on a Git project even if you don't have an internet connection.

- Git does have remote operations that require a network connection (e.g. to synchronize different copies of a project across multiple machines), but network is not needed until you explicitly execute one of these remote operations.

- We'll see more about how to actually use Git in just a bit.

# What is GitHub?

- **At its core, GitHub is just a cloud hosting service for Git repositories**, and one common use case for GitHub is simply to serve as a location for developers to backup their code on the cloud. Under this use case, a developer would use Git commands to synchronize their code with a remote repository on GitHub as they make commits.

- GitHub is far more than just a cloud hosting service, though. Its most important features are the tools it provides that build on top of its hosted Git repositories to turn each one into a central point for collaboration with other developers.

- These tools are most typically accessed through GitHub's web client (available at https://github.com) and include bug/issue tracking, code review (through a feature known as ***pull requests***), task management, continuous integration, and more. GitHub even offers some social features, allowing developers to follow each other, "star" their favorite repos, etc.

- Indeed, because of these features, **GitHub is most commonly used to help a team of developers to collaborate on a code project.** Under this model the code project is hosted in a repository on GitHub that becomes the central focal point for collaboration on that project.

- Typically, each collaborator on a GitHub-based project has their own clone of the central GitHub repository, and as they write code and make commits to their clone, they ***push*** those commits back to the central repository. Then, other developers can ***pull*** those commits from the central GitHub repository into their own clones to incorporate each other's commits into their own copy of the code.

- Later in the course, we'll explore some of the collaboration tools GitHub offers as well as workflows for using these tools to work with other developers on a shared project hosted in a GitHub repository.

# Basic individual Git/GitHub workflow

- Now that we understand a bit about what Git and GitHub are, let's explore a basic workflow you might use as an individual developer working on a project hosted on GitHub, including specific Git operations for making commits, synchronizing those commits to GitHub, etc.

- Note that **Git is by nature a command-line program**, so we'll explore the command-line form of the various Git operations here. **I strongly encourage you to get comfortable with the command-line version of Git.** This will give you access to the full range of Git's powerful functionality. Note though, that if you're not comfortable with the terminal (though you probably should be), there are various nice GUI Git clients available, such as GitHub Desktop (which is designed to interface with repositories hosted on GitHub) and GitKraken. Many code editors, such as VS Code, also have GUI Git clients built into them. You are free to use any of these clients for this course.
    - Note that if you want to use GitKraken for this course, you'll need to be able to work with private repos on GitHub, which requires GitKraken Pro. Normally, you have to pay for GitKraken Pro, but as long as you're a student, you can get GitKraken Pro (and lots of other cool stuff) for free through the GitHub Student Developer Pack.

- One way or another, if you want to use Git, you'll have to make sure you have *some* Git client installed on your development machine. The Pro Git book

contains documentation on how to install Git on various platforms. Here are a few different notes:

  - Windows users will need to install Git for Windows, which provides a command-line version of Git (called Git BASH) as well as a simple GUI version of Git.
  - Mac users may find it easiest to install a recent version of Git using Homebrew (`brew install git`).
  - If you like to use the ENGR servers (e.g. `access.engr.oregonstate.edu`), the command-line version of Git should already be installed there.

- If you're installing or using Git for the first time on your development machine, you'll need to set some basic configuration information before you can use Git. At a minimum, you'll need to tell Git what your name and email address are. Git will use this information to assign you as the author of the commits you make on this machine. You can use Git's `config` operation to set your name and email address, e.g.:

```
git config --global user.name "Rob Hess"
git config --global user.email "hessro@oregonstate.edu"
```

- Now we're ready to start using Git. There are two basic ways we can begin with a Git-based project: we can create one from scratch, or we can start with an existing repo, e.g. from GitHub. We'll look at both of these methods here.

## Creating a Git repository from scratch

- A Git repository is always associated with a directory. This works perfectly, since a code project typically lives in its own dedicated directory. As we'll see in a minute, we can pick and choose which of the files and subdirectories in our project directory are actually put under version control with Git.

- If we want to create a Git repository from scratch, our first task is to identify the directory that will be associated with the repository. This can be an existing directory with an existing code project if we want, but here, we'll start with a new, empty directory. Let's assume we have a new, empty directory to work with called `my_git_project/`. If you're following along here, you should create this directory on your development machine using whatever mechanism you usually use to create new directories.

- Once we have our project directory, our first step will be to initialize that directory as a Git repo. We can do this by first navigating to the project directory in our terminal (e.g. `cd my_git_project/` in a Unix terminal) and then running Git's `init` operation:

  ```
  git init
  ```

- This will set up the initial "database" in which our project's version history and other information associated with the Git repo will be stored. In fact, this "database" is represented as a directory called `.git/` that lives in our project directory. Directories whose names start with a `.` are hidden by default, but we can still see the `.git/` directory. For example, if you're in a Unix terminal, you can run the command `ls -a` in your project directory, and it'll show all hidden files and directories, or in Windows Command Prompt, you can run `dir /a:hd` to see hidden directories.

- **Importantly, you should never manually modify anything in the `.git/` directory.** We'll rely on Git operations to maintain the data stored there, and manually modifying that data could corrupt the Git repository. Typically the only time we need to worry about the `.git/` directory is just to check whether it's present, e.g. if we want to confirm that a project has a Git repo associated with it.

## Adding files and making our first commit

- Now that our project directory is initialized as a Git repo, let's start to put some files under version control with Git. We'll need some files first, so let's create two files in our project directory called `cat.js` and `dog.js` (you can create these files however you want). Let's also add a little code to each of these files:

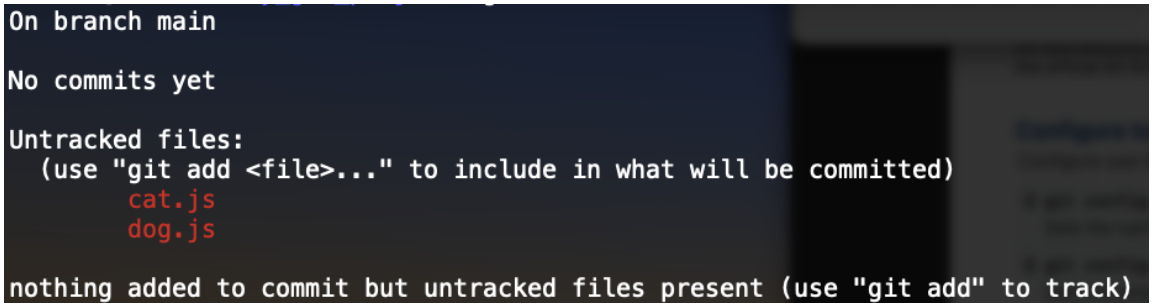
  ```
  // cat.js
  console.log("Mew!")

  // dog.js
  console.log("Woof!")
  ```

- Before going further, let's run another Git operation, the `status` operation, which will let us know the current state of our ***working directory*** (i.e. the project directory associated with our Git repo):

```
git status
```

- If we run this operation in our terminal, it will print a status message that looks something like this:

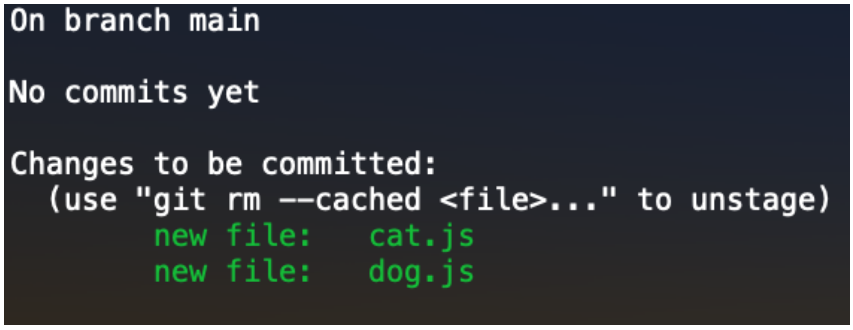



- At this point we can focus on the middle part of this status message, where it says we have “untracked files”. These are files that Git can see in our working directory but that aren’t currently under version control by Git.

- If we want Git to start tracking the files we just created, we’ll have to explicitly put them under version control. We can do this by running Git’s `add` operation (as described in the status message above):

```
git add cat.js dog.js
```

- Once we do this, we can run `git status` again, and we’ll see that the status message will have changed:




- Now, instead of saying we have untracked files, Git’s status message tells us we have “changes to be committed”. In other words, while we’ve put our two new files under version control, we haven’t yet made a commit (i.e. taken a snapshot) that includes those files. We can do this using Git’s `commit` operation:

```
git commit
```

- When you run `git commit` in your terminal Git will automatically open a text editor for you. The reason Git opens an editor is so you can type a ***commit message*** for the commit you're creating. A commit message is a brief description (usually one or two sentences) that concisely summarizes the changes being made in a commit. These messages are helpful when we are looking through a project's version history, since they help us remember the changes each commit represents. Every commit needs a commit message. For this commit, we'll type a message like this into the editor:

    *Created cat.js and dog.js.*

- Once we save that message and exit out of the editor, Git will create the commit for us. If we run `git status` now, we'll see a message that should say something like "nothing to commit, working tree clean".

- There are a couple different things to note about the way we entered the commit message. First, note that we can use `git config` to set the editor Git opens for us when we run `git commit`. For example, if we want Git to use `vim` as the default editor, we could run this command (you can replace `vim` with the command-line command for your favorite editor):

    ```
    git config --system core.editor vim
    ```

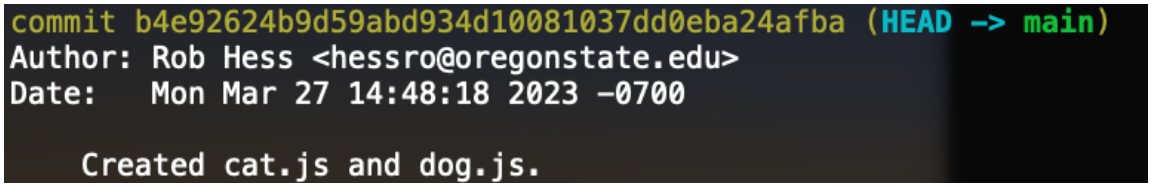
- Alternatively, if we didn't want to bother opening an editor, we could pass the `-m` option to `git commit` to specify our commit message directly on the command line. For example, we could have made a commit with the same commit message as the one we entered into the editor above by using this command (note that the commit message must be wrapped in quotes if it contains any spaces):

    ```
    git commit -m "Created cat.js and dog.js."
    ```

- Now that we've made a commit, we can look at our project's commit history using Git's `log` operation:

```
git log
```

- For now, since we've made just one commit, we'll only have one entry in the commit log, which will look like this:




- There are a couple things to note about this entry in the commit log. First, note that it contains information about the author who made the commit along with a timestamp indicating exactly when the commit was made. In addition, the commit message we wrote is included in the log to help us remember what the commit represents.

- Finally, note that the commit has a long hexadecimal value associated with it, which is known as the ***commit hash***. Here, the commit hash is:

```
b4e92624b9d59abd934d10081037dd0eba24afba
```

- There are a couple things to know about the commit hash:
  - The commit hash is a ***checksum*** computed based on the contents of the commit itself. This means that the commit hash verifies the integrity of the commit it's associated with.
  - The commit hash also serves as a unique identifier for the commit it's associated with. When we need to refer to a specific commit in certain Git operations, we will typically do so using the hash of that commit.

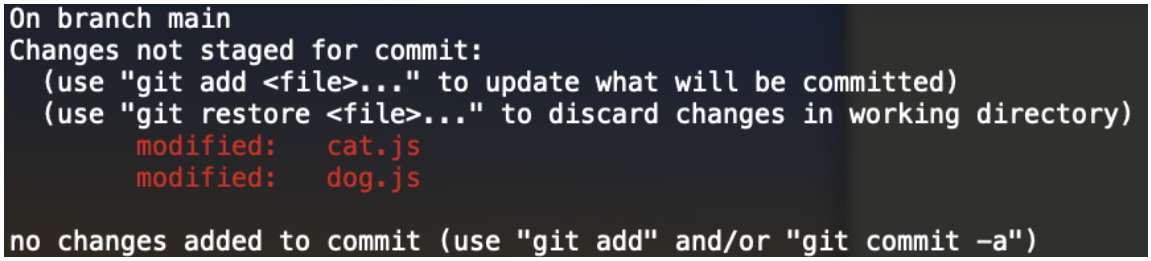
## Making changes and reviewing them using `git diff`

- Let's go ahead and make some changes to the files we created—after all, there aren't many software projects that are finished all at once. Specifically, let's slightly change what the cat says in `cat.js` (from "mew" to "meow"), and let's add some more barking in `dog.js`:

```
// cat.js
console.log("Meow!")
```

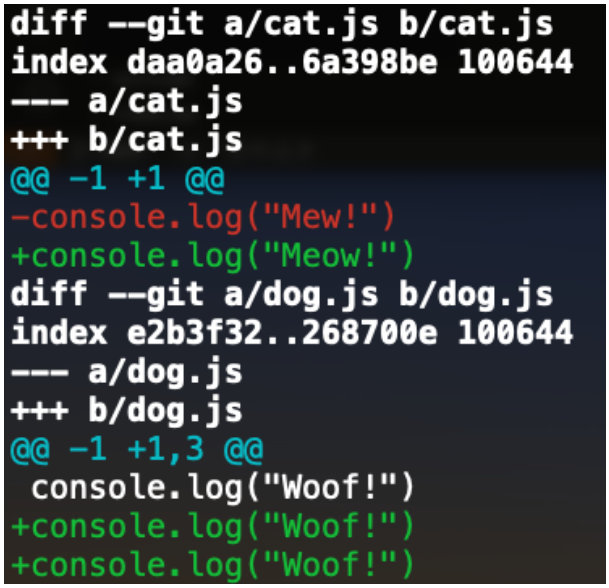
```
// dog.js
console.log("Woof!")
console.log("Woof!")
console.log("Woof!")
```

- Before we commit these changes let's run `git status` again. This is typically a good thing to do before making a commit, so we know exactly what changes we've made since the last commit. Now, our status message will look something like this:




- Now we see the status message tells us we have "changes not staged for commit". We'll talk about what this means in just a second.

- First, though, let's get a more fine grained view of the changes we've made using Git's `diff` operation. This will generate a ***diff*** displaying all the differences between the current working directory and the last commit we made. It'll look like this:




- Git produces a diff with a specific format known as unified format. The syntax here is somewhat cryptic because it's designed to be read by both humans and machines. Here's what to know about a Git diff:
    - Each file under version control that has changed since the last commit will be represented in the diff. The representation of the changes of each file is designated with a 4-line header, e.g.:

      ```
      diff --git a/dog.js b/dog.js
      index e2b3f32..268700e 100644
      --- a/dog.js
      +++ b/dog.js
      ```

    - The most important thing to recognize about this header is that it contains the name of the file it represents, here `dog.js`.

    - After the header for a file will be one or more ***change hunks*** representing the specific changes made to the file. Each hunk begins with a line indicating line numbers of the file represented by the hunk, e.g.:

      ```
      @@ -1 +1,3 @@
      ```

    - The range information here means that the hunk represents changes that start at line 1 of `dog.js` in the last commit (the last commit is symbolized with a `-` sign in the diff) and that range from line 1 to line 3 in the current

working directory (the current working directory is symbolized with a `+` sign in the diff).

- Finally, the hunk will contain a representation of the changes themselves. This representation will include three different kinds of lines (in the output above, these lines are each colored differently, but not all Git implementations will produce colored output):
  - A line that starts with a `-` represents a ***deletion***, i.e. a line that was present in the last commit but is no longer present in the working directory.
  - A line that starts with a `+` represents an ***addition***, i.e. a line that is present in the current working directory that was not present in the last commit.
  - A line that starts with neither a `-` nor a `+` is a ***contextual line***. These help provide some surrounding context for the actual changes.

- For example, the single change hunk associated with `dog.js` in the diff above represents the two lines we added since the last commit (along with a single contextual line):

```
 console.log("Woof!")
+console.log("Woof!")
+console.log("Woof!")
```

- As you can see in the single hunk associated with `cat.js` in the diff above, a modified line is represented as both a deletion and an addition:

```
-console.log("Mew!")
+console.log("Meow!")
```

## Staging and committing changes

- Before we commit our changes, it's important to understand that **a file under version control with Git always exists in one of three different states**:
  - **Modified** – A modified file is one that has been changed but is not yet committed.
  - **Staged** – A staged file is one that has been changed *and* that has been marked (as it currently exists) to be included in the next commit.

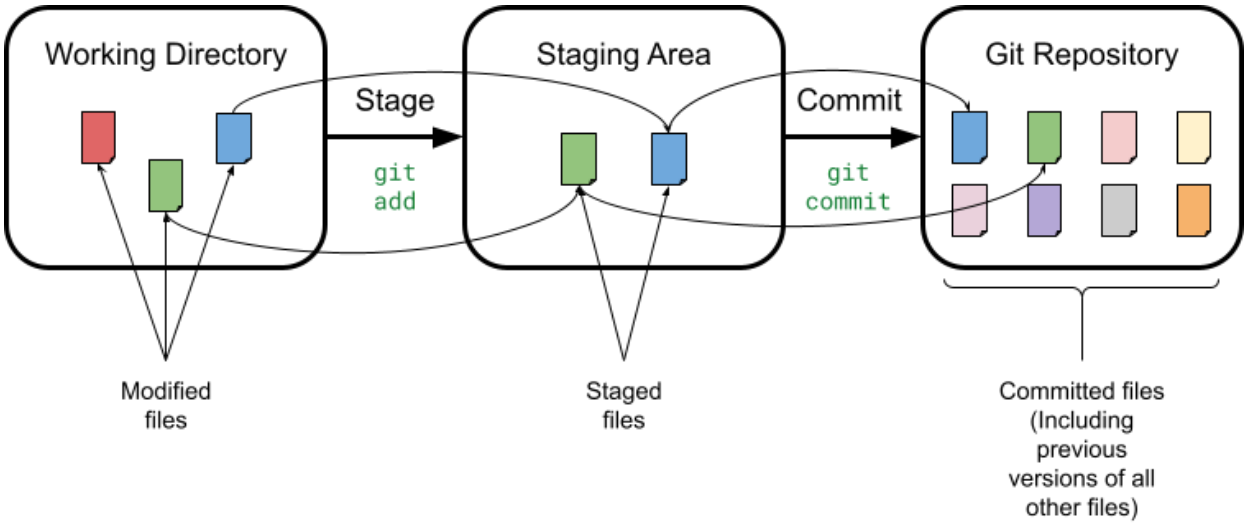
    - **Committed** – A committed file is one whose most recent changes have already been committed, i.e. one that doesn't have any changes since the last commit.

- **This means that before a file can be committed, it must be staged.** This is why the status message we saw just above said we had "changes not staged for commit." We have two files that were modified but that hadn't yet been marked as staged.

- In general, we can think of a Git project as having three separate parts: the working directory, the staging area, and the Git repository itself (i.e. the `.git/` directory).

- We already understand what the working directory and the Git repository are. **The purpose of the staging area is to allow us to pick and choose which modified files to include in the next commit.**

- In particular, as we work on a project, we may make changes to many files at once. At some point, we may decide that we are ready to commit some of those changes but not ready to commit others. This is what the staging area allows us to do. The working directory, the staging area, and the Git repository interact like this:

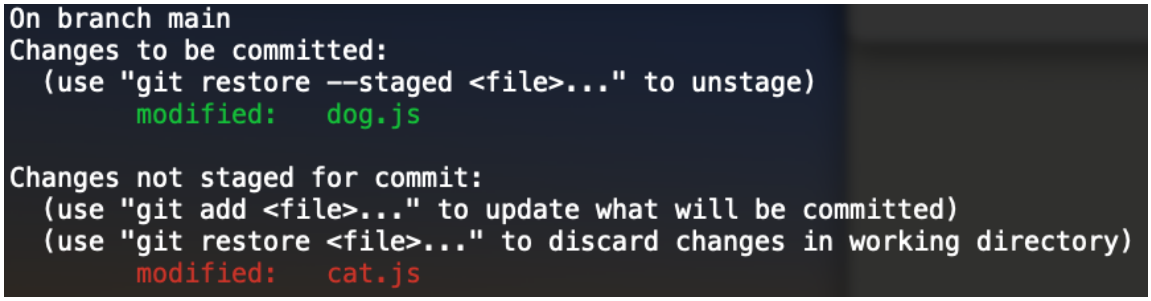



- We can stage a file using the `git add` operation. For example, if we decide we're ready to commit our changes to `dog.js` but not `cat.js`, we can stage

only `dog.js` with this command:

```
git add dog.js
```

- If we do this and then run `git status` again, the status message will reflect that only the changes to `dog.js` are staged:




- Now, we can run `git commit` to commit our changes to `dog.js`. Let's give it a commit message like this:

    *Make dog bark more.*

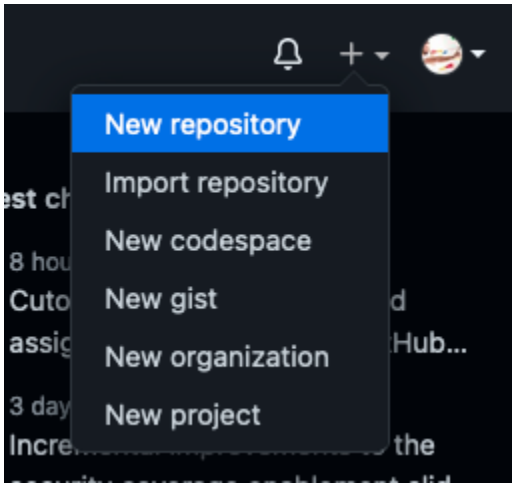
- Now, we should be able to run `git log` and see our new commit included in the commit history. Running `git status` will give a status message that includes only the unstaged changes to `cat.js`. If we wanted to commit those changes, too, we could do so using the same two-step process of staging and committing we used just now to commit the changes to `dog.js`.

## Mirroring a local Git repo on GitHub

- At the moment, the repo we've been working with lives only on our development machine. There are no remote copies of it. Let's work on changing that by mirroring our repo on GitHub.

- Start out by navigating to github.com in your web browser. If you haven't done so yet, you'll need to sign up for a GitHub account and make sure you're logged in.

- Once you're logged into GitHub, there are several ways to create a new repo there. For example, under the ➕ menu in the upper right corner of most pages on GitHub, there is an entry for creating a new GitHub repo:

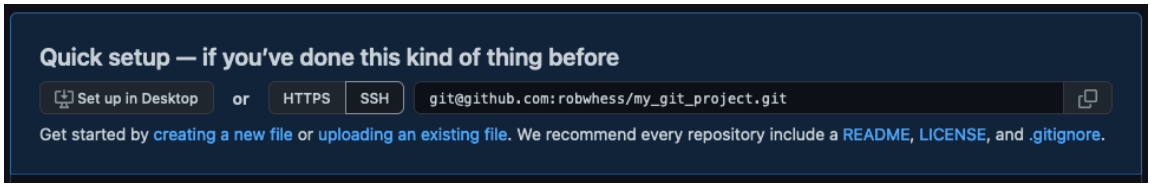


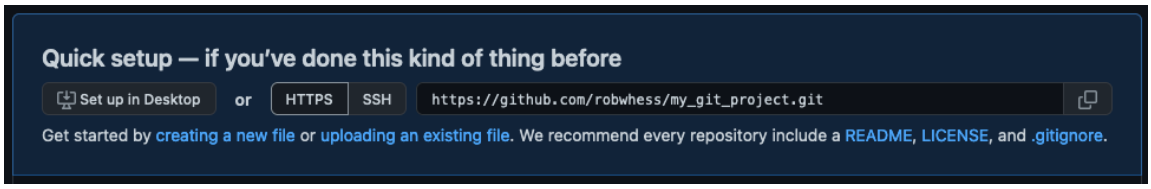

- Go ahead and create a new repo on GitHub. GitHub will ask you for some information about the repo you're creating. **Since we're starting with an existing repo (the one on our development machine), we'll have to be careful to set up our new GitHub repo correctly.** Specifically, on the "create a new repository" page on GitHub, we want to specify the following settings **so that no new files are created in the repo on GitHub**:
    - **Repository name** – Typically, the repository name should match the name of the project directory, which in this case is `my_git_project`.
    - **Description** – You can set this if you want.
    - **Public/private** – You can choose either. A public repo will be visible to anyone. A private repository will be visible only to you (until you give other developers access).
    - **Add a README file** – **NO**
    - **Add .gitignore** – **None**
    - **Choose a license** – **None**

- After you create the repository with the above settings, GitHub will navigate you to a page representing your new repo.

- We're almost ready to push the commits from our development machine to the new repo we just created on GitHub. Before we do this, though, we need to decide how our development machine will communicate with GitHub. There are two options: SSH and HTTPS.

- At the top of the page representing your new repo on GitHub, you should see a box with buttons that allow you to toggle between these two modes of communication (as you click them, you'll see the URL change):




- Either communication option will require some setup with GitHub and on your development machine. We'll briefly explore this setup below.

## Creating a personal access token to communicate with GitHub via HTTPS

- If we choose to communicate with GitHub using HTTPS, Git will prompt us to authenticate ourselves with GitHub when we execute a Git operation that communicates with GitHub.

- **Importantly, GitHub does not support simple password-based authentication.** Instead, if we want to use HTTPS communication, we'll have to generate a GitHub ***personal access token***. This is a special string that acts as an alternative to a password and has specific permissions built into it.

- To use HTTPS communication, you'll need to start out by following the instructions in the GitHub documentation for creating a personal access token. These instructions give you a couple options about what kind of personal access token to create and what permissions to associate with the token. For this course, I'd recommend the following settings:
    - **Type** – personal access token (classic)
    - **Scopes** – repo

- The personal access token you create will be a long string something like `ghp_uY8...`. Make sure you copy this string and keep it somewhere secure. **You should treat your personal access token like you would treat a password.**

- Have this token ready. We'll use it in a minute when we want to push commits from our development machine to GitHub. In some environments, you may be

prompted for your personal access token every time you want to use Git to communicate with GitHub. In other environments, Git will remember your personal access token after the first time you use it.

## Setting up SSH keys to communicate with GitHub via SSH

- In order to be able to communicate with GitHub via SSH, we'll need to create an SSH key and register it with GitHub. An SSH key is an authentication credential that will be exchanged with GitHub when we connect with it via SSH. It serves as an alternative to a username and password. In fact, even though an SSH key typically has a password associated with it, we'll usually only need to enter that password once, and thus an SSH key can be used to perform "passwordless" authentication.

- There are several steps to follow to set up an SSH key and register it with GitHub. Follow these steps on your development machine:
  - First, check to see if you already have existing SSH keys.
  - If you don't already have an existing SSH key, generate a new one and set your machine up to use it.
  - Next, register your SSH key with GitHub.
  - If you want to, test your SSH connection with GitHub.

## Setting our GitHub repo up as a remote for our local repo

- Once we're ready to communicate with GitHub using either HTTPS or SSH, we can now push the repository data from our development machine to GitHub.

- On the new repository page on GitHub, select either HTTPS or SSH, depending on which form of communication you want to use, and then copy the corresponding URL, which should look something like one of these two:
  - **HTTPS** – `https://github.com/YourGHUsername/my_git_project.git`
  - **SSH** – `git@github.com:YourGHUsername/my_git_project.git`

- In order to connect the local repository on our development machine with the one we just created on GitHub, we'll need to register the GitHub repo as a ***remote*** within our local repo. A Git remote is simply a repo that lives somewhere else that we want to keep synchronized with our local repo.

- To do this, we can use Git's `remote` operation, which allows us to view and manipulate remotes. We'll register our GitHub repo as a remote named `origin`,

indicating that it will be the default remote we want to synchronize with. You can do this by running the following command, pasting in either the HTTPS URL or the SSH URL, depending on which form of communication you want to use:

```
git remote add origin <Your_HTTPS_or_SSH_URL>
```

- You can verify that the remote was correctly set up by running this command, which prints out all the remotes associated with a local repo along with their URLs:

```
git remote -v
```

- This should print out a message that looks something like this (with your own HTTPS or SSH URL, of course):

```
origin  git@github.com:robwhess/my_git_project.git (fetch)
origin  git@github.com:robwhess/my_git_project.git (push)
```

- Next, we'll make sure our local repository is using the correct default branch name. By convention, the name `main` should be used for the default branch:

```
git branch -M main
```

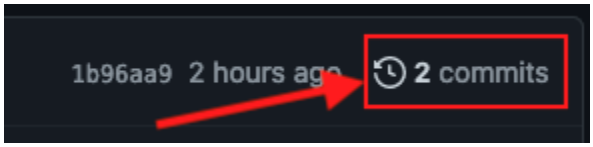
- Finally, we can use Git's `push` operation to send the commit history from our local repo to the remote repo on GitHub:

```
git push -u origin main
```

- This command specifically tells Git to send the changes from the current branch on our local repo (whose name we just set to `main`) to the branch named `main` on the remote named `origin`. The `-u` option makes a permanent connection between the local branch and the remote branch so that in the future, we'll be able to execute the `git push` command (as well as other commands, like `git pull`) without arguments.

- If the `push` operation successfully executes, the files you committed to your local repo should be visible on GitHub if you refresh the GitHub repo page in your browser.

- Take a minute to explore the repo page on GitHub to see what information is available there. For example, you should be able to view your project's entire commit history by clicking the commit history link in the GitHub repo. It looks like this:




- We'll explore other features available through this repository page on GitHub as we move through the course.

## Starting with an existing repo on GitHub

- The other main way we might begin working on a Git project is to start with an existing remote repo, e.g. one on GitHub.

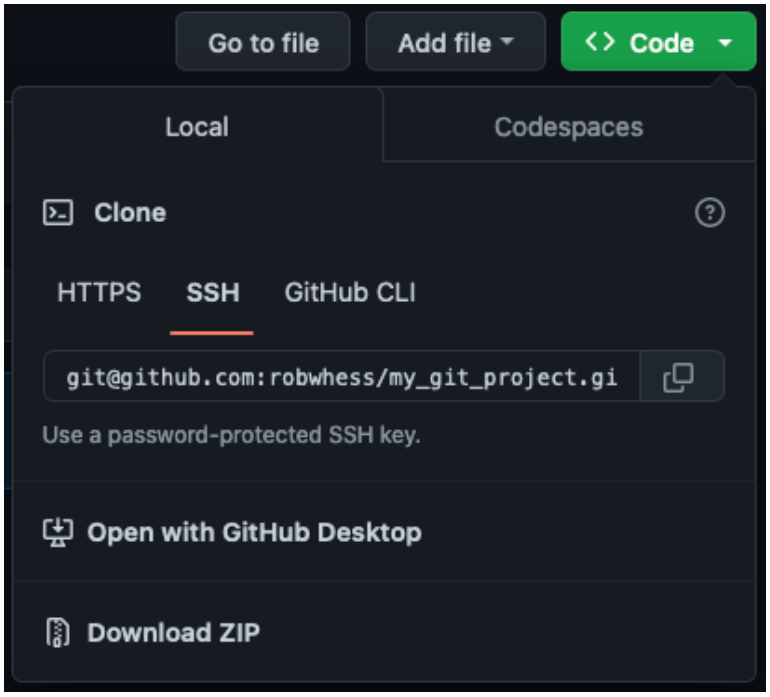
- In order to do this, we'll need to make a working copy of the remote repo on our development machine. In git terminology, this is called making a ***clone*** of the remote repo.

- To see how to do this, let's actually make a clone of the repo we created and mirrored on GitHub just above.

- If you want to do this on the same development machine where you created the original repo, you'll need to start out by navigating to a different directory on that machine. Alternatively, you can move to a different development machine for now (e.g. by SSH'ing to one of the ENGR servers).

- We'll refer to whatever location you're going to make a clone of the repo as "**location #2**". We'll refer to the original location of the repo as "**location #1**".

- On the home page for your project on GitHub, you should see a green "Code" button. If you click that button, it'll open a dropdown with options for cloning the repo (along with other options for working with the code):




- In the “clone” section of that dropdown, make sure you select the form of communication you want to use to clone the repo, either HTTPS or SSH, and copy the URL presented in the dropdown for that form of communication (it should be the same URL we used when we set up the remote above).

- At **location #2**, run this command to clone the repo, pasting in your HTTPS or SSH URL:

  ```
  git clone <Your_HTTPS_or_SSH_URL>
  ```

- When this command completes, you’ll have a clone of the repo at location #2. This clone will live in a directory whose name matches the name of the repo on GitHub (i.e. `my_git_project/`), and it will contain the project’s complete version history.

- If you list the contents of that directory and look at the files, you’ll see that they match the most recent commit we made at location #1 (remember, we didn’t commit the modification we made to `cat.js`).

- Importantly, because we created this local repo by cloning a remote repo, Git will automatically set up a connection to the remote repo, giving it the name `origin` like we did when we manually set up a remote at location #1 above.  You can see

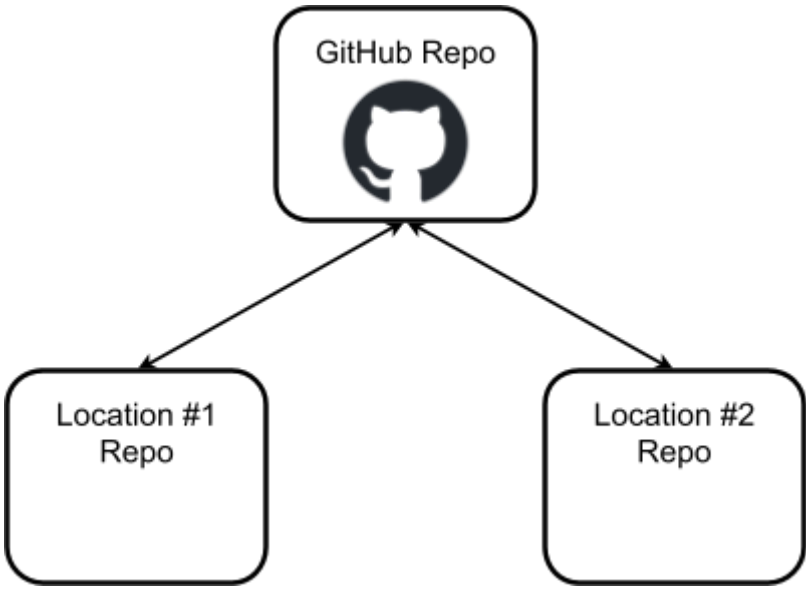
this if you navigate into the new repo and run `git remote -v`. Like we did manually above, Git will set up the branch `main` of the local repo to track the branch `main` of `origin`, so we can run `git push` and `git pull` without arguments.

- We can now work on this cloned repo at location #2 in exactly the same way we worked on the original repo at location #1. If we do continue to work in both locations, there will be a few extra considerations we'll need to make. We'll cover those next.

## Working on a Git repo in two different locations

- If we want to work on two different instances of the same Git repo (e.g. on two different machines), we'll have to take a bit of extra care to make sure our commit history stays consistent between the two repos.

- In this case, we have two different clones of a repo (in location #1 and location #2) that are both connected to the same repo on GitHub:




- Let's make a new commit at location #2 and see how to deal with that commit at location #1. Specifically, let's make and commit the following modification (adding one more "woof") **at location #2**:

```
// dog.js at location #2
console.log("Woof!")
```

```
console.log("Woof!")
console.log("Woof!")
console.log("Woof!")
```

- Once that change is committed, let's push it to the repo on GitHub by running `git push`.

- Now, let's say we want to go back to work at location #1. We know we made changes to the code at location #2 and pushed those changes to GitHub, so before we start to make changes at location #1, we need to make sure we have the most recent version of the code there by synchronizing the repo at location #1 with the one on GitHub. We can do this with Git's `pull` operation:

  ```
  # At location #1
  git pull
  ```

- This will "pull" all of the commits from GitHub into the repo at location #1 and incorporate those commits into the working directory.

- A good rule of thumb when working on two different instances of the same Git repo is this:

  ***Before you start working in a different location, pull all changes from GitHub to make sure you're working on the most up-to-date version of the code.***

- Following this rule of thumb will help to prevent conflicts from being introduced into the version history.

- Let's keep going a bit to see what might happen when we don't follow this rule of thumb.

## Dealing with merge conflicts

- Let's remain at location #1 for a minute. Recall when we stopped working at location 1, we had actually made a modification to `cat.js` we didn't yet commit. You can run `git diff` there to see it:

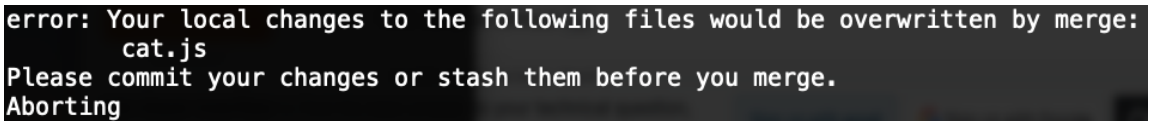

- Let's commit and push this modification:

```
# At location #1
git add cat.js
git commit
git push
```

- Now let's move back to location #2 and pretend we've forgotten to follow our rule of thumb about pulling before we start working there. **Specifically, before we pull at location #2, let's make a change to `cat.js` there**, adding a new line to make the cat purr (the first line of `cat.js` should *not* reflect the change we committed and pushed from location #1):

```
// cat.js at location #2
console.log("Mew!")
console.log("Purr...")
```

- Now, let's say we remember at this point that we forgot to pull from GitHub before starting to work again at location #2, and we go ahead and run `git pull` there. If we do that, git will report an error to us:




- As the error message tells us, we have two different options to handle this situation: commit our changes or stash them. One of these options will be fairly painless, and one will lead to some trouble. Let's briefly look at the painless option first, then we'll explore the one that will lead to trouble.

- The painless option here is to ***stash*** our changes. The term “stash” here has a specific meaning. In particular, Git has a `stash` operation we can use to record the current, modified state of the working directory and then roll the working directory back to the most recent commit:

  ```
  # At location #2
  git stash
  ```

- If we run `git stash` at location #2, and look at our code, we’ll see it now reflects the state it was in when we last committed at location #2. If we run `git status` at this point, it will report that our working directory is clean (i.e. it has no modifications). We could also see a record of our stashed changes if we wanted to:

  ```
  git stash list
  ```

- At this point, we could run `git pull` at location #2 if we wanted to, and it would successfully pull the most recent commits from GitHub. Then, we could reapply the changes we stashed and keep going from there. However, let’s not do this yet. Instead, let’s explore the more painful option we could have taken when we first tried to pull changes from GitHub.

- **Without pulling**, let’s reapply the changes we stashed at location #2:

  ```
  # At location #2
  git stash pop
  ```

- Note that the subcommand here is called `pop` because Git’s `stash` functionality stores stashed changes in a stack. Each time we run `git stash`, it pushes a new set of changes on top of the stash stack. Running `git stash pop` pops the changes from the top of the stash stack and reapplies them.

- *Now*, let’s commit those changes:

  ```
  # At location #2
  git add cat.js
  git commit
  ```

- If we go one step further now and try to push the new commit to GitHub, we'll see an error message that looks something like this:

```
To github.com:robwhess/my_git_project.git
 ! [rejected]        main -> main (non-fast-forward)
error: failed to push some refs to 'github.com:robwhess/my_git_project.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

- The error message is slightly misleading in this particular situation. The real issue here is that we have now created two different versions of the commit history, one that lives at location #2 and another that lives both on GitHub and at location #1. We can see the two different commit histories if we run `git log` at both location #1 and location #2.

- If we try to run `git pull` at location #2 as suggested by the error message above, we'll get another error message that's more accurate:

```
hint: You have divergent branches and need to specify how to reconcile them.
hint: You can do so by running one of the following commands sometime before
hint: your next pull:
hint:
hint:   git config pull.rebase false  # merge
hint:   git config pull.rebase true   # rebase
hint:   git config pull.ff only       # fast-forward only
hint:
hint: You can replace "git config" with "git config --global" to set a default
hint: preference for all repositories. You can also pass --rebase, --no-rebase,
hint: or --ff-only on the command line to override the configured default per
hint: invocation.
fatal: Need to specify how to reconcile divergent branches.
```

- Again, we have a couple different options here for how to deal with the issue. We'll try to resolve it by ***merging*** the two commit histories together:

```
# At location #2
git merge origin/main
```

- This command will specifically try to take the commit history from the `main` branch of the `origin` remote and join it together with the local `main` branch at location #2. Unfortunately, when we do this, we get another error:

```
Auto-merging cat.js
CONFLICT (content): Merge conflict in cat.js
Automatic merge failed; fix conflicts and then commit the result.
```

- The underlying problem here is that the latest commit in each of the two different commit histories here have different versions of the same line of code. Specifically, we have two different versions line 1 of `cat.js`:

  ```
  // Line 1 of cat.js at location #1 and on GitHub
  console.log("Meow!")

  // Line 1 of cat.js at location #2
  console.log("Mew!")
  ```

- This is known as a ***conflict***, and it is preventing Git from being able to automatically merge the two commit histories together (which it can normally do if there are no conflicts). This means we must manually resolve the conflict to complete the merge.

- If we actually look at the contents of `cat.js` at location #2 at this point, we will see that Git has modified the file to highlight the conflict. Specifically, at location #2, `cat.js` currently looks like this:

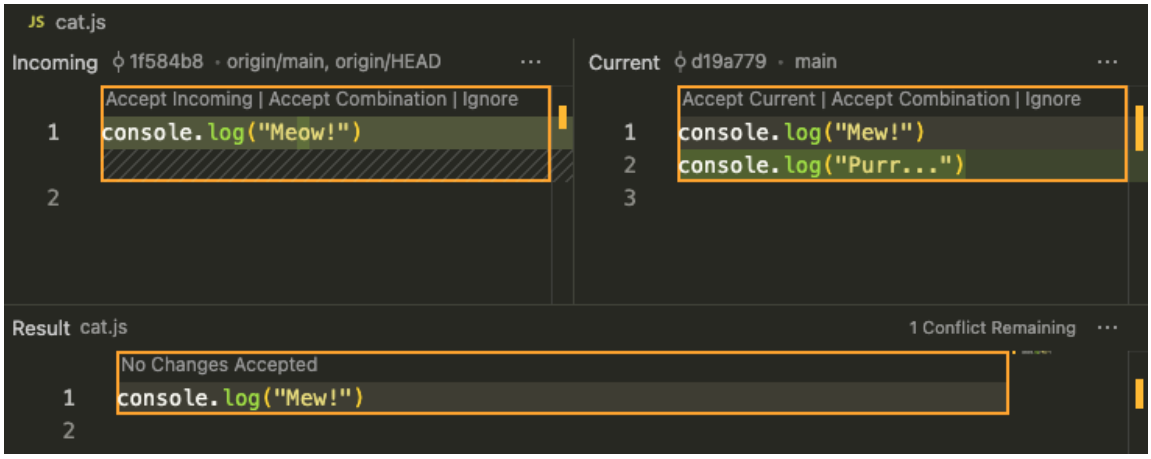
  ```
  <<<<<<< HEAD
  console.log("Mew!")
  console.log("Purr...")
  =======
  console.log("Meow!")
  >>>>>>> origin/main
  ```

- The syntax here may be somewhat cumbersome, but it's presenting us with the two different sides of the conflict. The first version (between the `<<<<<<<` and the `=======`) is the local version, and the second version (between the `=======` and the `>>>>>>>`) is the version from `origin/main`.

- We need to edit the file to indicate how we want to resolve the conflict. We can do this however we want. For example, we could delete everything except the local version of the code to resolve the conflict in favor of the local commit

history, or we could delete everything except the remote version of the code to resolve the conflict in favor of the remote commit history. Or, we could create a hybrid of the two versions of the code, with some pieces from the local version and others from the remote version.

- Many visual code editors have built-in tools to present this conflict to us in a more intuitive way. For example, in VS Code, we could use the built-in "merge editor" to view the conflict like this:




- Other visual editors have similar tools, and we can use them to click and type our way through the conflict.

- However we do it, let's resolve the merge conflict as a hybrid between the two version histories, taking line 1 of `cat.js` from the remote history and line 2 from the local history:

```
// Resolution of the conflict
console.log("Meow!")
console.log("Purr...")
```

- To complete the conflict resolution, we can mark the conflict as resolved by running `git add` and then running `git commit` to finalize the merge:

```
# At location #2
git add cat.js
git commit
```

- Here, when we run `git commit`, we will see that Git will give us the start of a commit message. We can add to this if we want.

- Now, we should be able to successfully push everything from location #2 to GitHub:

```
# At location #2
git push
```

- And, just for good measure, we can go back to location #1 to make sure we can pull the merged commit history there:

```
# At location #1
git pull
```

- Now, if we use `git log` to look at the commit histories at both locations, we should see that they are again the same. The commit history on GitHub should also be the same.

- Hopefully, this is the last time you ever have to worry about merge conflicts, but it probably won't be. In any case, if you do run into a merge conflict again, you should now have some idea about how to approach it.

- However, this should all make the importance of our rule of thumb above more clear. Let's repeat it again here for emphasis:

  ***Before you start working in a different location, pull all changes from GitHub to make sure you're working on the most up-to-date version of the code.***

# Git cheat sheets

- One of the nicest kinds of resources you can have as you're continuing to learn how to use Git and GitHub is a Git cheat sheet, which just provides a basic summary of some of the most important Git commands. You can find a couple nice Git cheat sheets at these locations:
  - GitHub's Git cheat sheet
  - Atlassian's Git cheat sheet