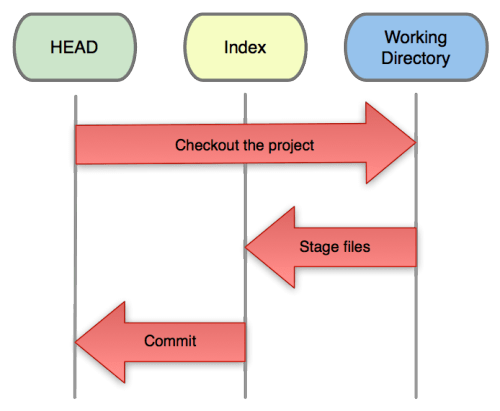
GitHub Basics

* Version Control Tools
  + Git: A method of version control
  + GitHub: A public platform for housing version-controlled repositories
  + GetHub Enterprise: A private platform for housing version-controlled repositories
* Repositories
  + Think of a repository just as you would a folder or file system.
  + Repositories, or "repos" as they are sometimes called, come in two standard varieties: local and remote.
    - Local: The copy of the data that is stored directly on your machine.
    - Remote: The copy of the data that is stored in a location other than on your machine.
  + For example, consider Dropbox and Google Docs. You may have copies of these files saved on your hard drive, but they also exist in the cloud and can be accessed from other devices.
  + A remote repository works the same way. If your computer meets an untimely end (whether lost, stolen, or fried by that cup of coffee you elbowed onto it) — your data remains safe!
  + Additionally, just like Google Docs, a remote repository makes it easier for different people to collaborate on a single project.
* What is Git?
  + Git is:
    - A distributed version control system
    - A program you run from the command line
  + Programmers use Git so they can keep a history of all changes made to their code and if necessary, switch back to older versions. Think of Git as a time machine for a programmer's project.
  + In fact, a Git repository is often structured around a specific project, containing all of the relevant files and directories needed to run that project. Git acts as a version control framework to track updates and iteration, making it easier to synchronize small changes with all the other files.
  + Git was created by Linus Torvalds, the principal developer of Linux.
* What is GitHub?
  + GitHub is:
    - A hosting service for Git repositories.
    - A web interface for exploring Git repositories.
    - A social network of programmers.
  + In other words, Github is built on top of Git, making it simpler for programmers to host and collaborate on projects. Github also offers tons of fun social features. Much like Facebook, Instragram, or Pinterest, you can follow GitHub users, star your favorite projects, and see what other people have been working on.
  + Developers use Github so they can store and share code bases, particularly when working collaboratively, either for a company or on open-source projects.
  + Developers can also simply use Github to back up and store their own work online, much like you would use Dropbox or Google Drive to store your files.
* What is GitHub Enterprise (GHE)?
  + GitHub Enterprise is:
    - A professional version of GitHub that runs a bit more securely, storing data on private servers.
  + Whereas regular GitHub (Github.com) is a public platform, kind of like a social network for programmers, Github Enterprise is a custom professional platform for companies.
  + Github Enterprise works exactly the same as Github in many ways, except content and materials are not usually public; instead, they are shared internally within a company or organization.
  + Great, so what does this mean for you? Throughout your time at General Assembly, you'll be interacting primarily with Github Enterprise, where we store many of our courses and lessons. For instance, in order to participate in our programming and data science courses, you'll need to create an account on GA's version of Github Enterprise: git.generalassemb.ly.
  + However, we also encourage students to create a separate, public Github.com account, in order to host projects, build a public portfolio, and share your work with the world!
  + Pro Tip: You can find more information on the differences between GitHub.com and GitHub Enterprise here.
* Creating a Local Repository
  + Now, let's make our directory into a Git repository.
    - First: Create and/or change directories to find the repo you want to set as a Git repository.
      * cd desktop
      * mkdir example1
      * cd example1
    - Second: Initialize this repo as a Git repository.
      * git init
    - You should see this:
      * Initialized empty Git repository in /Users/Username/Desktop/example1/.git/
    - This means it worked! Another way to tell is to use this command:
      * ls -A
      * The -A flag forces it to list all files in the directory, including hidden ones. You should see a hidden folder called .git. This is where all of the information about your repository is stored!
      * In general, you'll never need to actually see or use this file, it's just nice to know it's there, keeping a log for us. We'll teach you to control the versions this file tracks using some common git commands.
  + Your local repository consists of three “trees” that are maintained by Git
    - The **Head** points to the last command you’ve made; in other words, this is your most recent “official” save.
    - The **Index** is the staging area: this include any files you’ve told git to watch but you haven’t officially saved yet.
    - The **Working Directory** is the place where you are working with your local files. This is the current state of your directory, containing any files and edits you are in the middle of making.
    - 
  + Updating the Repository
    - Let's say we want to make an update to our repo.
    - We can use the touch command to fill our new repo with empty text files. Once we make some changes, we'll be able to see how git lets us save and view our versions.
      * touch file1.txt file2.txt file3.txt
    - What if you forget what's happening while you're working? No sweat. Use git status to see what's changed in your working directory (and remember what you were just working on).
      * git status
    - Your output should look something like this:
      * On master branch:
      * Initial commit.
      * Untracked files:
      * (Use `git add <file>...` to include what will be committed.)
      * file1.txt
      * file2.txt
      * file3.txt
      * Nothing has been added to commit, but untracked files are present (use `git add` to track).
    - "Untracked files" means you've made some local edits but git is not tracking them yet, so if anything happens, those edits will be lost!
    - Luckily, git to the rescue. As you'd expect, we'll want to indicate that git should start tracking those edits. We'll need to add them to the index before they can be committed.
  + Git Add
    - In order to tell git that we want it to keep tabs on our changes, we need to use the command git add. This specifically tells git to move our file from our working directory to the index/staging directory, so that it is "watching" our edits.
    - You'll be using this command a lot. Using git add you can either track individual file changes or tell it to log all changes at once.
    - Here's how to track a single file change:
      * git add file1.txt
    - Notice that after git add we specified exactly which file to watch. However, if we have made edits to multiple files and want to track a whole bunch of changes, we can type:
      * git add .
    - The . tells it to track all changed files in the directory. This is a great way to make sure you don't forget anything!
    - Ok. So now git is tracking our changes; they are "staged" in the index of our git file. Great! Now what?
    - The final step is to officially save these changes, which will "commit" them to the "head" and create a historical record of our edits.
  + Git Commit
    - To save our changes officially, we'll use the command git commit. This will move our file changes from the staging index to the head, so that they can be referred back to or restored at any time. This is the beauty of git!
    - Committing a change will prompt you to type out a message. This is your commit message. Here you should briefly state what changes you made. This is helpful if you want to remember what you did later or restore a specific edit. It's also one reason why git makes collaboration easy - team members can look at these commit logs to see what everyone else has done!
    - To bypass the prompt, you can type git commit -m "message". For example:
      * git commit -m "adding 3 text files"
    - The -m flag allows you to include a message right in your commit, making it a fast and efficient method for saving your work.
    - After typing git commit and providing a message, your output should look something like this:
      * [master (root-commit) b946f3f] adding 3 text files
      * 3 files changed, 0 insertions(+), 0 deletions(-)
      * create mode 100644 file1.txt
      * create mode 100644 file2.txt
      * create mode 100644 file3.txt
    - This shows that our file changes have been saved to the "head" of our repository. Our changes are officially saved!!
    - Try typing git status now, which should tell us that our working directory is up to date with the head.
  + Git Workflow
    - Ok, let's recap!
      * First navigate to the directory you want to work in. Later, we'll show you how to download content from remote servers in order to work locally. You'd want to navigate to this local directory on your terminal first.
      * Next, type git init to create a .git file in your directory. Git will passively notice any changes made to files in this directory from this point on, and refer to them as "untracked."
      * Do some work! Make any kind of change to your files.
      * If you get busy working and forget whether you've saved your changes or not, you can type git status at any time. Try it! This will tell you whether your edits are untracked, staged, or saved.
      * Next, let's save our work. Type git add . to save multiple files, or type git add filename.txt to save a single file.
      * Almost there! Your changes are staged but not saved yet. Next you need to type git commit or git commit -m "message" to officially save your changes.
    - Great work! Remember, saving files with git is a two-step process. First you have to tell git to add the changes to the file, then you have to officially "commit" them. You won't be able to save files without adding them first.
    - What's next? Normally once you've committed a change, you'll want to push your change to a remote location so that your edits can be accessed in other locations or restored if anything happens to your local machine. We'll cover that in a minute!
  + Creating a GitHub Repository
    - Ok! So as we mentioned, git is a powerful tool that works best when paired with a service like Github, which lets you save your work to a safe remote server.
    - Let's create a repository on GitHub to see how this works.
      * Go to www.github.com and create an account, if you haven't done so already. If so, just log-in.
      * Click on the New Repository button. This is a green button (and is normally near the top-right of your screen).
      * You should see a prompt to create a "New Repo." Here you can name your repository. Be creative! However, don't use any spaces between words. Try hyphens or underlines if needed.
      * You'll see some other fields. "Description" is what it sounds like - this is a little text field where you can describe the project you're working on. "Public" or "Private" indicates options for sharing your work or keeping it under wraps. The point of Github is to share, but you may want to keep some works-in-progress private while you get started. Either option is fine.
      * Before continuing, make sure to check the little box that says: "Initialize with a README." This will provide us with a sample markdown text file that we can edit in our new repo. You can ignore the other options for now.
      * Finally, click "Create Repository."
    - Nice work! Now you should see a screen showing you the contents of your new repository, including the README file.
  + Cloning a GitHub Repository
    - Ok, let's connect our online Github repo to our local machine. To create a local version of our repo, we'll need to copy and download our files. Github calls this process cloning. In essence, you're duplicating your online files locally.
    - Make sure you are in the root directory of your new Github.com repo. Click on the blue title to be sure. Now you should see a big green button on the far right that says "clone or download". What does this do? You guessed it!
    - Github gives you many different options for cloning. For now, just click on the hyperlink and copy it.
    - Next, switch back over to your terminal. Take a minute to change the directory to a location where you want to download your new repo folder; e.g. your desktop.
      * cd desktop
    - In your terminal, type git clone and paste the URL you copied from Github. For example:
      * git clone https://github.com/your-name/repo-name.git
    - Hey, look at that! Now you have a new local folder for your repo, containing any files your repo had. This is a local version of your Github repository.
    - Now navigate to your local repository and practice the same process as before - making edits, staging, and committing them. Make a change to the README.md file.
  + Pushing to a GitHub Repository
    - Now, let's say you've finished making some updates to the files on your local repository.
    - Next, we want to back these up by pushing your changes to your remote repository. In other words, we want to keep our local and remote repositories in sync.
    - First make sure you've navigated to your cloned repo directory, and made your edits in the sample file. Then - like before - you'll need to "stage" your edits and "commit" them.
      * git add .
      * git commit -m "sample commit"
    - Great, now let's synchronize and "push" those edits up to our remote Github.com repo. Use this command:
      * git push
    - Or to be more precise, you can type git push origin master. Either command should work. The more precise version is just another way of telling git exactly where you want those files to go.
    - The full command is simply saying: git, push these files up to my remote repo, which is by default named origin, and put them on the master branch, which is the default "branch" to store your files. We'll cover more on these topics later.
    - If your command works correctly, you should see a message in your terminal that looks something like:
      * Counting objects: 3, done.
      * Delta compression using up to 4 threads.
      * Compressing objects: 100% (2/2), done.
      * Writing objects: 100% (3/3), 270 bytes | 0 bytes/s, done.
      * Total 3 (delta 1), reused 1 (delta 0)
      * remote: Resolving deltas: 100% (1/1), completed with 1 local object.
      * To https://github.com/user-name/sample-repo.git
      * dd1072e..9233837 master -> master
    - That means it worked! Check for yourself by clicking over to your Github.com repo and taking a look.
  + Pulling form a GitHub Repository
    - Awesome work so far. Now, what if you really like Github's GUI and you decide to make some edits to your file remotely? In other words, your remote repo is no longer synchronized with your local copies.
    - Before you do any work locally, you'll want to **pull** down your remote changes.
    - Try it yourself. Click over to the Github GUI in your browser and make some more edits to your README file using the "pencil" icon. Make sure to scroll down and click the green "commit" button to save them.
    - Now your remote file should be different than your local file. To "pull" down your remote updates to your local file, use this command:
      * git pull
    - Hint: Yep, the more precise version can be used here too - git pull origin master.
    - If it works, you should see some sample output in your terminal that looks something like:
      * remote: Counting objects: 1, done.
      * remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 1
      * Unpacking objects: 100% (1/1), done.
      * From https://github.com/user-name/sample-repo.git
      * 9233837..ab1a00c master -> origin/master
      * Updating 9233837..ab1a00c
      * 1 file changed, 4 insertions(+), 1 deletions(-)
    - This is telling you that it successfully found changes in your remote file and has merged them with your local file. Success! Open up your file locally in order to see for yourself.
* Pulling It All Together
  + What do you think would happen if you worked locally without pulling down your remote changes first, then tried to synchronize? Would there be two conflicting versions of the file? Yes! This is called a merge conflict and they are no fun to deal with. That's why you should always keep your repos in sync!
    - To avoid that unhappy scenario, let's review our workflow from start to finish.
  + **Create Your Repos**
    - Create a repo on Github.com.
    - Click on the green "clone or download" button and copy the hyperlink.
    - In your terminal, navigate to your desktop or wherever you want your local repo copy to live.
    - Type git clone and paste the hyperlink.
    - Success! Now you have local and remote repos, linked by a git file that is ready whenever you want to store and synchronize your files.
  + **Link Your Repos**
    - Perhaps you've already created some files on your local machine and you want to make a Github.com repository for them. What would you do?
      * Create a folder on your desktop with the sample files you want to save.
      * In your terminal, navigate to that folder and type git init.
      * Next, go to Github.com and create a repo for your files.
      * Click on the green "clone/download" button and copy the hyperlink.
      * Back in your terminal, type git remote add origin and paste the hyperlink.
      * Type git status which should show you a message saying that your local and remote repos are not currently synchronized! Do you know what to do next? Note that when you add a remote to an existing local repo, when you want to push or pull, you'll need to explicitly specify the location (e.g., git push origin master).
  + **Work In Your Repos**
    - Let's say you left for a while to watch some cat videos. Time to get back to work! In your terminal, navigate back into your local repo directory.
    - Type git init to re-start git.
    - Type git status to see where you are and what's happening.
    - Before starting any work locally, it's almost always a good idea to make sure you are synchronized. Go ahead and type git pull.
    - Cool. No more procrastinating, time to get to work. Make some edits.
    - Awesome, let's make sure all this work gets tracked: git add .
    - Ready to save? git commit -m "your message goes here".
    - Push that commit on up to remote to keep your repos in synch: git push, or to be extremely precise git push origin master.
    - Confirm by clicking over to your Github.com repo. Everything look good?
  + Congratulations, you just mastered the basics of git and Github. Give yourself a high five!
* Forking A Repo
  + One interesting feature of git and Github is the ability to share repos with others. That way developers can share code and build off each other's work.
  + For example, in our data science classes, we've built our curriculum using many different repositories, or online folders, of content. Let's say you want to get a copy of those folders to download their contents for yourself.
  + Luckily, making copies of other repositories on Github is easy. We call this process forking. If you visualize a repo as a straight line of activity, a fork is a copy that splits off to go do its own thing.
  + Forking a repo in Github means copying repositories from one location to another (e.g., from someone else's Github account to yours). To practice forking a repo on Github, click on Github's icon to go to its home page, and use the "discover" tab to look up some cool public repos. Here's an example repo for a cool data science tool we use'll be using later:
    - [Jupyter Notebook](https://github.com/jupyter/notebook)
  + To "fork" and make your very own copy of this repo, click on the tiny button near the top right that says "Fork". You'll see a cute animation while Github creates your own copy for you.
  + In Github you can also "star" or "watch" repositories for easy reference.
  + Now, when you go to look at a list of your Github.com repos, you'll notice the repo you forked is there! This is now yours to edit (or break) as you see fit!
  + Once you have "forked" a repo, you can also use pull requests to send changes back to the source, or pull in edits from the original repo. We'll cover this topic more in later lessons.
* Lesson Practice
  + Ready for some hands-on practice? Great! Take some time to practice the following steps a few times before advancing to the next slide.
    - Go to your home directory and create a new folder called awesome\_repo.
    - Move into this folder and initialize it as a git repo.
    - Create a file called awesome\_file, add it to Git, and commit the file locally. (Don't forget your commit message!)
    - Create a new repository on github.com called awesome\_repo.
    - Add your locally committed awesome\_file to your new remote repo.
    - Fork this practice repo on GitHub. Then clone your new fork down onto your machine.
* Lesson Practice Solutions
  + Have you finished practicing? Next, check your work against our suggested solutions below:
    - Go to your home directory and create a new folder called awesome\_repo:
      * cd ~
      * mkdir awesome\_repo
    - Move into this folder and initialize it as a git repo:
      * cd awesome\_repo
      * git init
    - Create a file called awesome\_file, add it to Git, and commit the file locally (don't forget your commit message!):
      * touch awesome\_file
      * git add awesome\_file
      * git commit -m"add awesome\_file"
    - Create a new repository on github.com called awesome\_repo.
    - Add your locally committed awesome\_file to your remote repo. In the example below, replace "username" with your own Github account name:
      * git remote add origin https://github.com/username/awesome\_repo.git
      * git push
    - Fork this practice repo on GitHub and clone your fork onto your machine:
      * Fork the repo on Github first, then
      * git clone https://github.com/username/practice\_repo