Lecture 4: GitHub basics

GitHub is a web service built on the program *git*. This very short piece https://guides.github.com/introduction/getting-your-project-on-github/ talks a little about the basic things you can do with GitHub, and how to get started. Y Here's a long list of tutorial and guides: https://help.github.com/articles/good-resources-for-learning-git-and-github/. You're welcome to download install the "GitHub Desktop," but Eclipse includes all the functionality you'll need, and you'll probably find it's easier to work within Eclipse.

So what, and why, is git?

The website (http://git-scm.com/) says

"Git is a free and open source distributed version control system"

What do those pieces mean?

Historically, git was developed to support the process of developing Linux—quite a complicated project, with hundreds or thousands of volunteers from all over the world contributing code. How to manage all those little chunks of code?

Notice that the description of git says nothing about code! You don't have to use git with code; you can use it with any kind of project you want to keep track of. I'm going to talk in terms of code, just because that's what we're dealing with in this class—though note that by 'code' I certainly mean documentation files and other kinds of "extra" stuff you'd expect to see in a software project.

In this class (as you've hopefully read), I'll be using github to make it easy for me to distribute code for your projects; it will also serve as a kind of backup for your work. And then when the project is done, I'll be able to use github to make it (slightly) easier to grade your work. So, we won't be using *many* of the important features of github.

Git stores your project in a *repository*. This keeps track of the entire history of your project. Now, a software project may involve several kinds of activity at once: maybe some people are working on preparing a release version of the software, other people are incorporating bug fixes, and still other people are trying to implement some experimental features. So a repository can keep track of many *branches* of a project. At the beginning of a project, though, there's only one "branch," and it's named `master`. For our purposes in this class, you'll mostly only need one branch, though you might want to play with the idea of using another branch for "experimenting."

Because git is a *distributed* system, there are two kinds of ways you might interact with git. First, you might use it to make changes to your "local" repository., on your computer. But (more interestingly), you also might use it to synchronize your local repository with another repository somewhere on the Internet. That's the *distributed* part. You don't have to use this at all—you can use git just as a local record of your work. But in this class, and in most projects, the general idea is that the repository "in the cloud" is the main one, and you periodically sync up your local repository with the one "out there."

Let's talk local first.

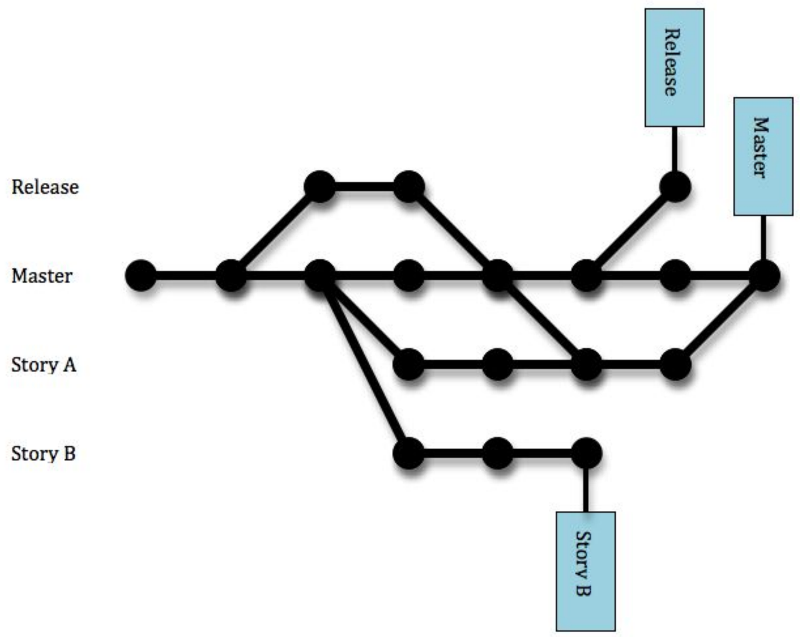
Once you've written a useful chunk of code—maybe you fixed a bug, or you added a new class definition, you'll want to "save" it in git. This will be a record of what you did (who knows, maybe this will turn out to be a bad idea and you'll want to undo it later), and also something you'll be able to share with others working on the same project (if there are any others!).

In git terminology, you want to commit work from your *working directory* (the file system) into the repository. But it's possible that you don't want to put *all* your changes into a single commit, so git lets you decide which files should go into the commit—it provides a "staging area" where you can decide exactly which files should be committed. (On the command line, this is usually controlled with the `git add` command.).

Once you've added the files you want to commit, you then use the `git commit` command. You'll be prompted to provide a brief explanation of what the 'purpose' of this commit is. Usually these explanations are written as present-tense commands, like "Fix the login bug." or "Correct the value of pi."

git then saves your files in a new *snapshot*—yes, each snapshot is a copy of all the files in your project! Git also changes the `master` branch so that it points to your newest commit—and this new commit also points to the previous version of your code. This makes it easy for you to move around in the history of your work, if you need.

As we'll see (though we probably won't use this much), git makes it pretty easy to create new branches (for that experimental feature)



git makes it fairly easy to navigate the various nodes of your local repository, but I'll leave that for you to explore as you wish.

Let's talk about working with distributed repositories

There are three primary git commands for relating two repositories: **clone**, **push**, and **pull**.

Clone is typically only used at the beginning of a project—if you decide you want to start contributing to the foo project, you'd find the address of its public repository online (maybe http://github.com/foo), and say **git clone http://github.com/foo**

Or, you can add a remote repository with git remote add — you need to provide it the name you want to use for the remote repository, plus its address. Often, remote repositories are called origin because they're the "main" repository for the project. So,

git remote add origin http://github.com/foo

Once you've cloned a remote repository, you'll want to stay up to date with it. To get the latest version, use the pull command.

git pull origin master

This, roughly, brings your repository up to date with the remote one. Note that you should make sure you've committed all your changes *before* you do this, or else the pull will not be executed. Depending on what you're pulling, you may have to do some work to merge the two versions of the code.

Finally, to put your changes up on the remote, you can say

git push origin master

Again, you should make sure that you've pulled and merged any changes from the remote repo before you do this, otherwise git will refuse your code.

For projects in this class, you'll pretty much only need to use clone and push—no-one besides you will be working in the repositories.

https://try.github.io

For today's exercise, follow these two tutorials (note that you'll need to have your github account!)
First: https://guides.github.com/activities/hello-world/

Don't remove any of your work from that tutorial; use it to complete this one:
http://eclipsesource.com/blogs/tutorials/egit-tutorial/