Git

Preliminaries

We will be using GitHub in this class for assignments. Everyone needs to create a GitHub account if you don't already have one. Do this at:

https://github.com/

After you set up your Github account you will need to inform us what your GitHub username is. Their will be instructions for how to do that in the near future.

Preliminaries

- We will be using the command line version of git.
  - If you are using Cygwin you may need to install it. (Run the installer again and search for the git program.)
  - To see if you have git installed, type the command

    ```
    which git
    ```

    If it gives you a location of git, then you have it.

Resources

- There is a man page for the top-level **git** command but there are man pages for each Git operation, too. For example, these all work:

  ```
  % man git
  % man git-clone
  % man git-checkout
  ```

- *Ry's Git Tutorial* by Ryan Hodson is available in several forms.

- A free online copy of the book *Pro Git* by Scott Chacon and Ben Straub is available at https://git-scm.com/book

- *PeepCode Git Internals* by Scott Chacon reveals some of Git's innards. https://github.com/pluralsight/git-internals-pdf/releases

Git is a version control system

- The essential job of a version control system is to maintain a history of changes to a collection of files.

Git is a version control system

- The essential job of a version control system is to maintain a history of changes to a collection of files.
- This is more than just a text list of what has changed in a project, but a sort of “time machine” that can recreate exactly what the files in your project looked like at particular points in time

Git is a version control system

- The essential job of a version control system is to maintain a history of changes to a collection of files.
- This is more than just a text list of what has changed in a project, but a sort of “time machine” that can recreate exactly what the files in your project looked like at particular points in time
- In association with github git can also add in coordinating a project between programmers and in working on a project on several machines.

Motivation

Suppose you were working on a project in a directory called **`myGreatProj`** that contained many source files, you have no tool like **git**, but you want to maintain some sort of version control. What could you do?

Motivation

Suppose you were working on a project in a directory called **`myGreatProj`** that contained many source files, you have no tool like **git**, but you want to maintain some sort of version control. What could you do?

You might make multiple copies of the directory manually:

```
    % cp -R myGreatProj myGreatProjv1
```

and after you do more work you do it again to a new name:

```
    % cp -R myGreatProj myGreatProjv2
```

What are some disadvantages of this system?

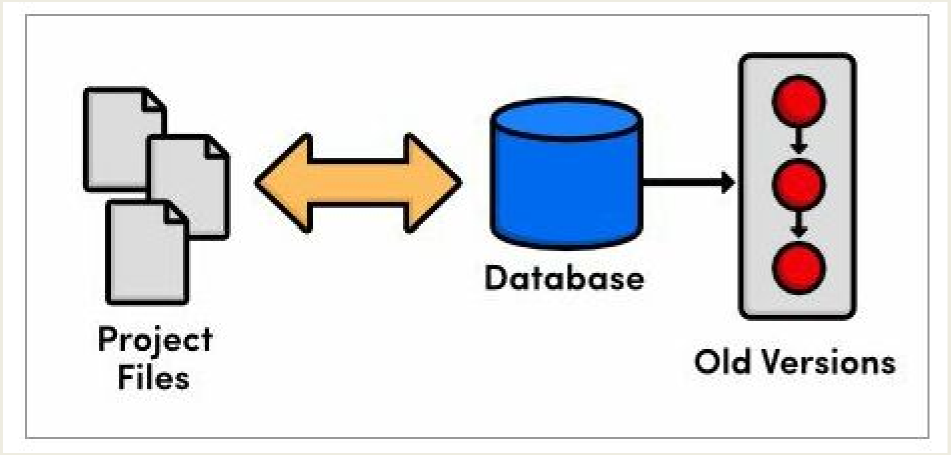
Motivation

What are some disadvantages of this system?

1. It’s clunky
2. It takes up a lot of space.
   a. You are saving copies of files that haven’t changed
   b. You are saving complete files even if only a small change was made.
   c. You may be saving files you don’t care about (e.g. test files, object files, compiled code)
3. It’s slow. (You’re copying all that extra stuff mention in 2.)
4. It’s hard to navigate. (Which version had what change?)

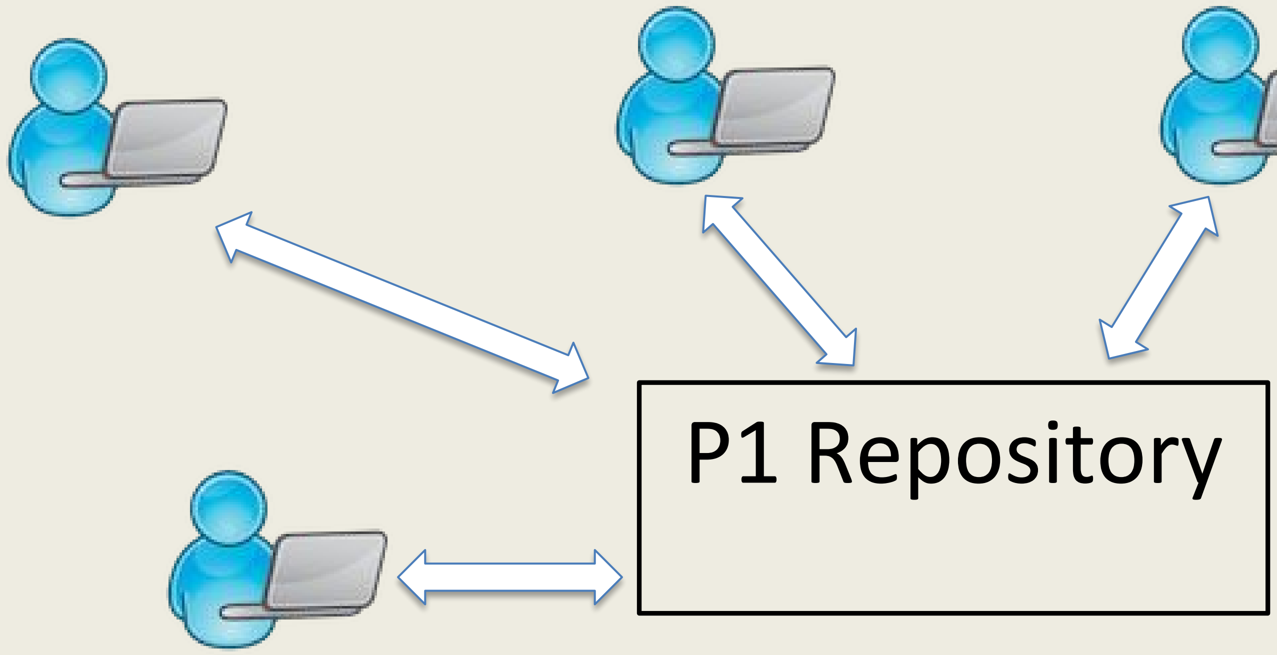
Local Version Control Systems

- A Version Control System VCS automates the process.
- It saves the changes (say in a database) and the user "checks out" a version of the project.
- The user only has one version of the project she has to deal with.
- This process took place on a programmer's local computer. There was no good way to share code amongst several developers.



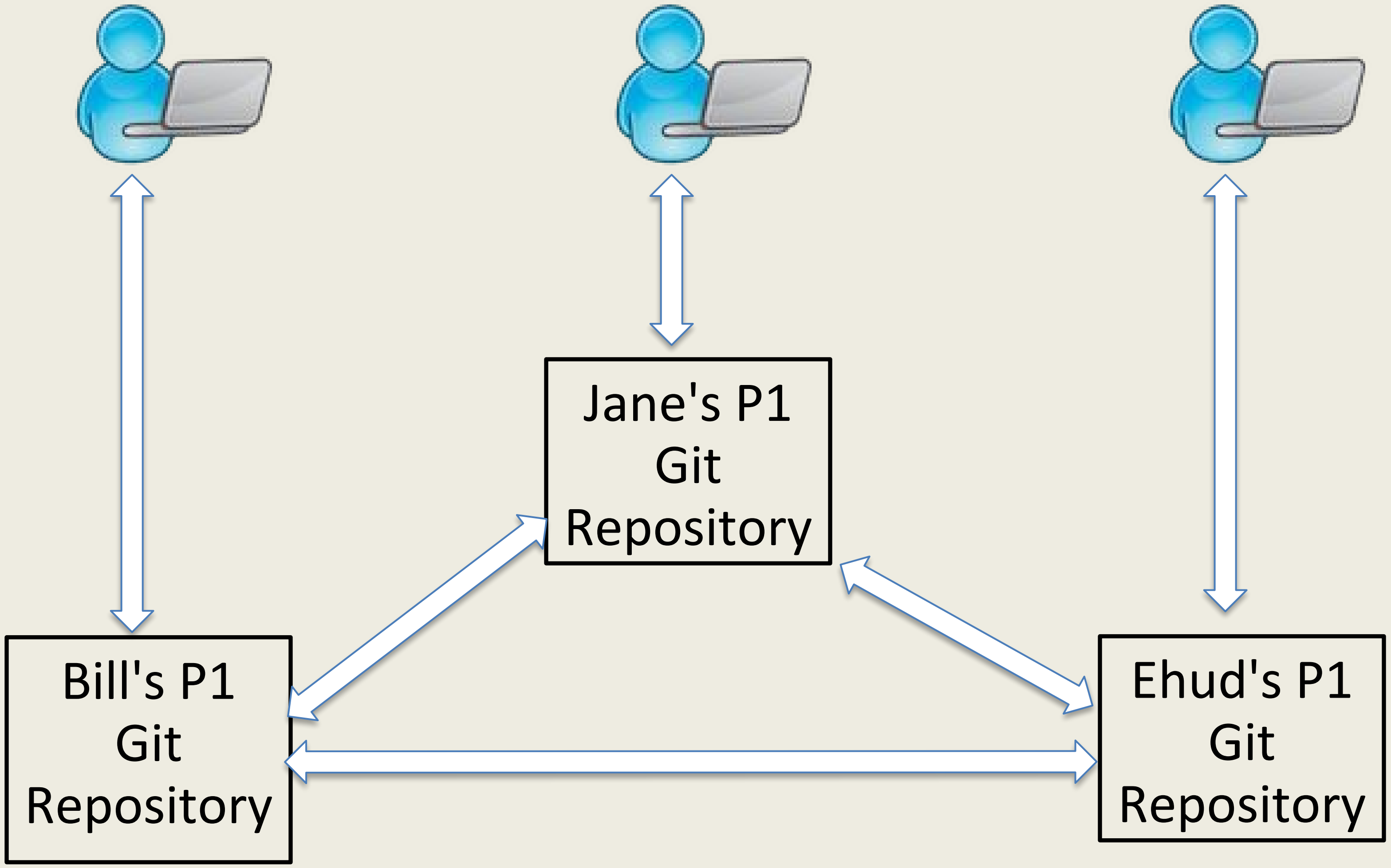
Centralized VCS

In a Centralized VCS instead of storing things locally, the project was saved on a server and multiple users could check out files to work on. CVS was an example of such a VCS.



Distributed VCS

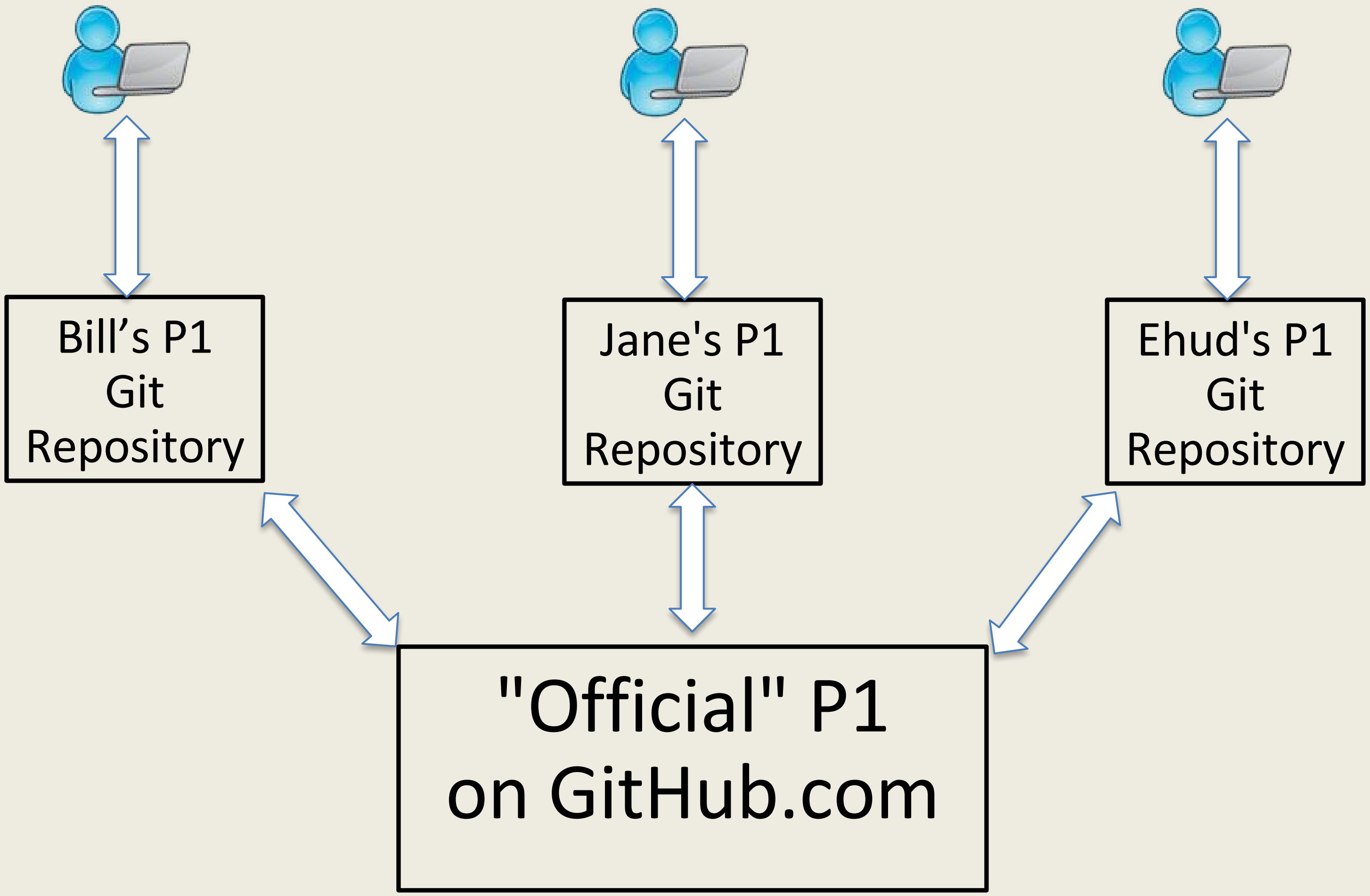
Git is a distributed version control system. Every developer on a project will have on their machine a copy of a repository for that project.



GitHub

Git provides ways to transfer groups of changes between repositories.

Git is typically used with a web-hosted service that maintains the "official" repository for a project. We'll be using GitHub.



git

- git is a distributed VCS
- developed in 2005 for use by the Linux community
- Being used to manage the Linux kernel forced several requirements on git including:
    - Reliability
    - Efficient management of large projects
    - Support for distributed development
    - Support for non-linear development

git config

Before you start using git you might want to do some configuration. First set your name and email address:

```
% git config --global user.name "Eric Anson"
% git config --global user.email eanson@email.arizona.edu
```

Obviously you should use your own name and email. The global flag tells git to save these values in your home directory and thus you use them for every git project you work on.

When you save something to the repository (**commit** a change) and you don’t give a description of the change, **git** will invoke an editor for you to enter a description. By default that editor is vim (if it’s installed). You can set which editor is used via:

```
 % git config --global core.editor <your editor choice>
```

git config

Lastly, when you interact with github is will ask you for your user id and email. To have it cache your credentials for a day you can set:

```
% git config --global credential.helper "cache --timeout 86400"
```

using git

Your files for any project will be organized inside a directory. Usually in this class that directory will be created from a repository on github. If you are working on a private project, you can use git locally:

```
% mkdir myProject
% cd myProject
% git init
```

The commands above create a directory called myProject, move into it, and then and then the **`git init`** command will change that directory into a repository.

Creating a git repository creates a hidden directory called **`.git`** which contains all the tracking information.

```
% mkdir myProject
% cd myProject
% git init
% ls -a
.   ..   .git
```

The only difference between a normal folder and a Git repository is the **`.git`** directory. You do not want to manually change anything in it.

You can add files, edit files, and delete files from this directory and these changes will not be saved by git until you tell it to. As an example, suppose you add the files **`README`** and **`prog1.py`** to your directory. Now invoke the command:

```
% git status
```

You will get a result like something shown on the following slide:

```
% git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what
will be committed)

        README
        prog1.py

nothing added to commit but untracked files
present (use "git add" to track)
```

git add

This indicates that the files are not under version control (untracked). You need to tell git to add these files to the repository if you want to track them. To do this you must first stage a snapshot. The command:

```
% git add README
```

Tells git that the next time you take a snapshot, you want to include the file README. To also include **prog1.py** you can follow with:

```
% git add prog1.py
```

Git add will also take multiple arguments so we could have just typed:

```
% git add README prog1.py
```

git commit

Note staging a snapshot does not save the state of your files, it just indicates what will be saved. To actually do the saves you use the command:

```
% git commit
```

The git commit command saves the staged changes along with a description of what those changes are. With no options, git commit opens a default editor for you to write a description of this snapshot. To avoid this step you can use the -m option to include the description with the command:

```
% git commit -m “Files README & proj1 added”
```

git so far

So we’ve seen three stages of working on our project:

1. Edit the files in your directory
2. Stage the changes to save in next commit
3. Commit the changes

```
% git status
```

Will tell you which files are changed and what is staged.

git log

To see a history of your commits you can use the command:

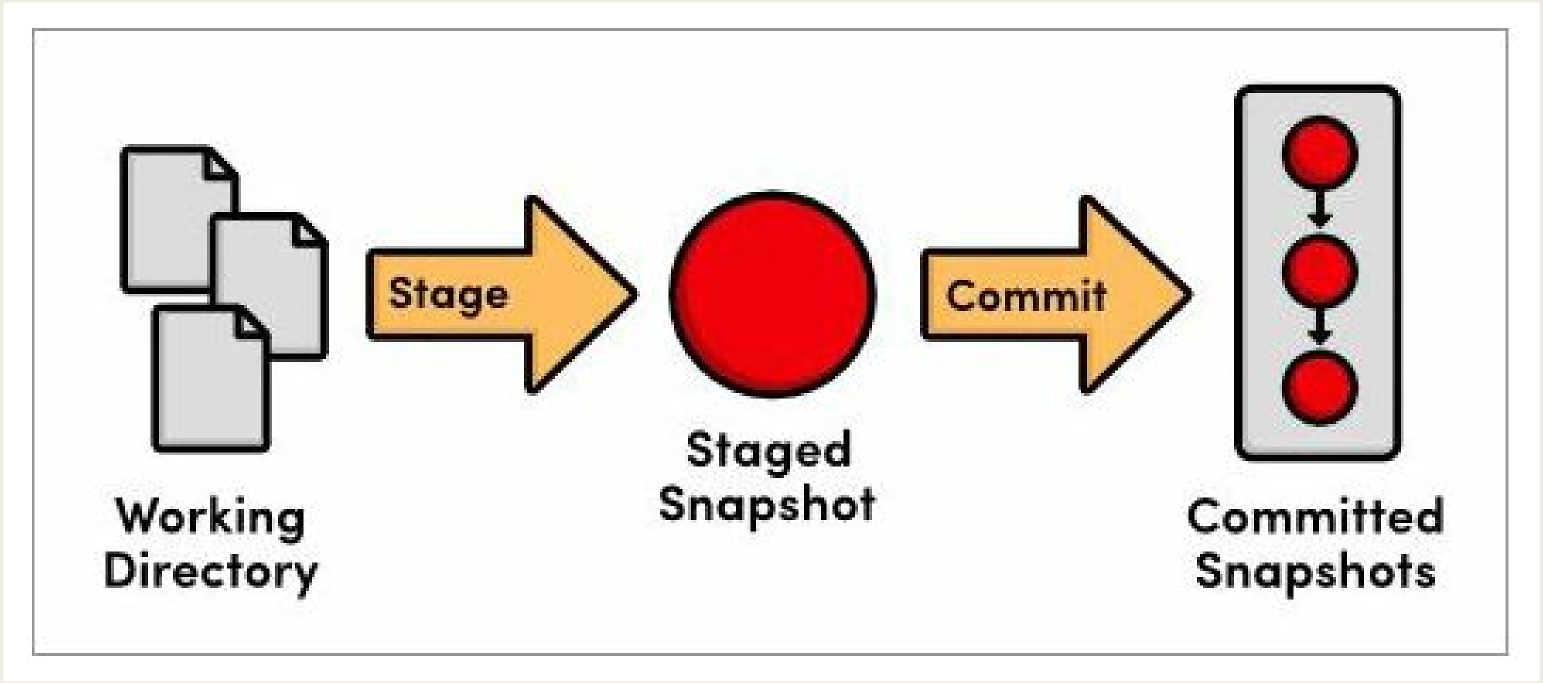
```
% git log
```

This will list out all the commits made, who made them, and the description written for each one. It is pretty wordy. Most often you will want to use the option --oneline to condense the output:

```
% git log --oneline
a893df6 (HEAD -> master) Text added to README
214ff25 Files README and prog1.py created
```

This shows only a hash identifier for the commit and the description of the change. In this case there were two commits to this repository.

So far



This diagram indicates how to think of **git**. Use **add** to stage changes, use **commit** to save snapshots. **status** tells you about the working directory and the stage. **log** tells you about the committed snapshots.

git show

You can see more information on some committed snapshot using the git show command

```
% git log --oneline
a893df6 (HEAD -> master) Text added to README
214ff25 The files README and prog1.py are
created
% git show a893df6
```

Notice you use the hash number to identify the snapshot you want information on.

git checkout

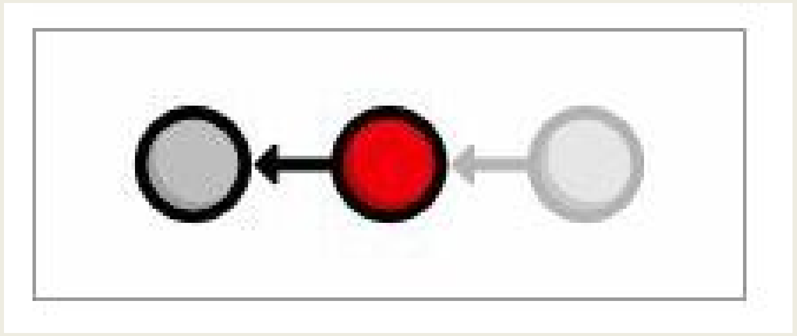
You can use the git show checkout command to view what the project looked like at a particular snapshot

```
% git log --oneline
f2e80fc (HEAD -> master) added code to prog1
a893df6 Text added to README
214ff25 The files README and prog1.py are
created
% git checkout a893df6
```

This command changes all the tracked files in your directory to be as they were at the time of the snapshot. Even the log will look as it did at the time.

git checkout

So have we lost all the changes since that snapshot? No, we have just changed our view. Look at the following diagram:



This represents our snapshot history. The red circle is the current view, and the faded circle is the latest version.

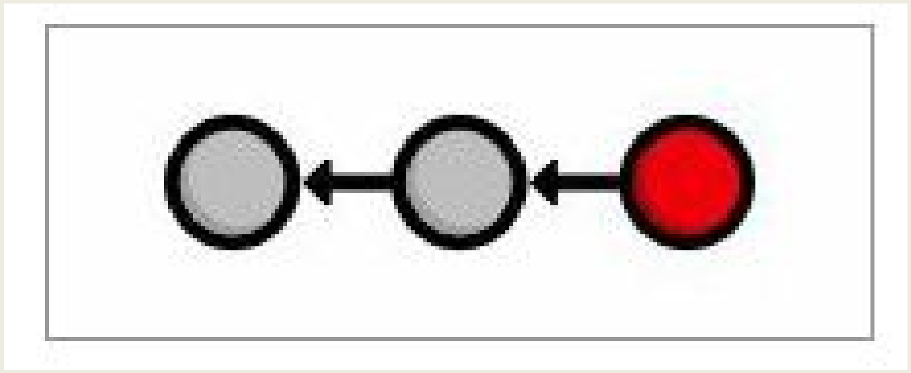
We can always restore back to the latest version.

git checkout

You can return to the latest snapshot by using the command:

```
% git checkout master
```

master is the name of the main branch of your repository. We won't talk much about branches, but you can read about them if you're interested.



undoing a committed change

You can undo a committed changed using the revert command:

```
% git log --oneline
9d4df69 Add an experiment
c40879b add some stuff
8a9e200 Added two file stubs
bae3928 First draft
% git revert 9d4df69
```

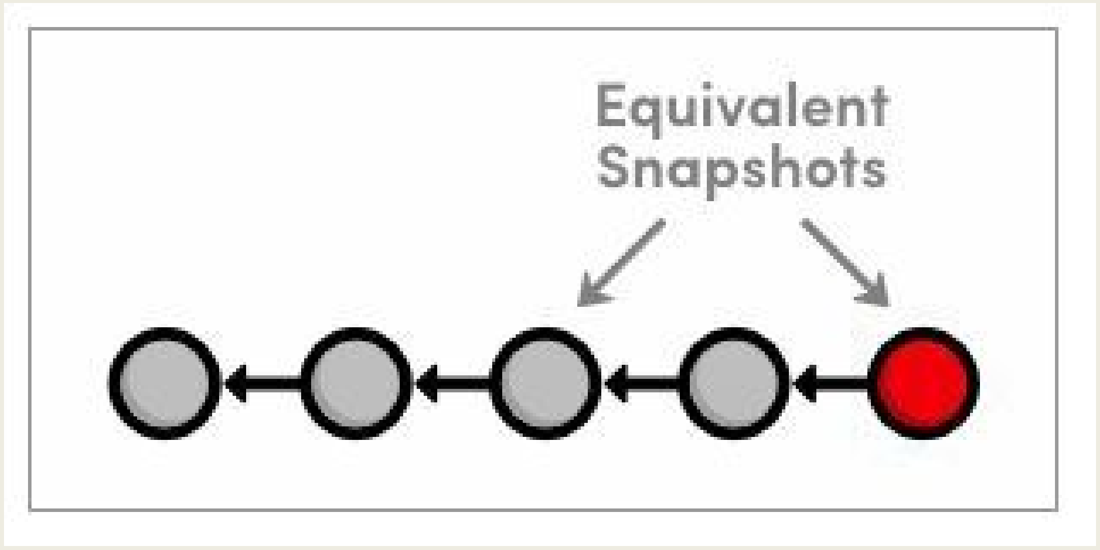
This command will bring you in an editor to record a comment and then undo the changes done in the commit identified. It doesn’t delete the history, but adds a new snapshot that has the changes undone.

undoing a committed change

Here is what the log looks like after the commands of the last slide:

```
% git log --oneline
2e136b2 Revert "Add an experiment"
9d4df69 Add an experiment
c40879b add some stuff
8a9e200 Added two file stubs
bae3928 First draft
```

A graph of the snapshots:



undoing uncommitted changes

The command:

```
% git reset --hard
```

Changes all tracked files to match the most recent commit. This change cannot be undone, so use it with care. Without the **--hard** option the **reset** command clears the staged snapshot.

The command:

```
% git clean -f
```

deletes all untracked files from the directory. This is another change that cannot be undone. The **clean** command does have options that check what will be deleted that you might want to use.

working with GitHub

Most of the time in this class you will not create a repository with the git init command, but instead copy a repository from the web. GitHub lets you create a repository using a web browser, and then copy it to your local machine using:

```
% git clone <url given by GitHub>
```

For example

```
% git clone https://github.com/csc210dev/sample
Cloning into 'sample'...
...
Unpacking objects: 100% (10/10), done.
Checking connectivity... done.
```

Aside from clone, there are only two more commands you have to learn for working with GitHub. The command:

```
% git push
```

Will push your repository up to the one stored on GitHub. This is a method of backing up your work to the cloud and also turning in your assignments. You must push your work to GitHub by the due date.

The command:

```
% git pull
```

Pulls changes from the repository on GitHub to your local machine. This can be useful if you want to work on different machines. If you do this, just be careful to always start your sessions with a pull and end them with a push to make sure you're working with the latest copy.

If you are not working on different machines, you probably won't use this command unless we use it as a way to give you feedback.

Note that unless you pay for it, the repositories you create for on GitHub are public, meaning anyone can clone them. This is not true for our assignments. When you accept an assignment for this class, GitHub will create a **private** repository for you on its server and give you it’s url. You then use that url in a clone command on your own machine.

Assignments with GitHub Classroom

An invitation link will be posted on Piazza and the class web page. Example:

https://classroom.github.com/a/b8c-a4gI

Hitting the link will take you to a page where you can "Accept this assignment". Accepting it creates a repository with a URL like

https://github.com/csc210f17/a*NUMBER-GITHUB_ID*

Do a **git clone**:

```
git clone https://github.com/csc210f17/a2-jsmith  a2
```

Use **git add** and **git commit** commands to commit your work to the assignment-specific repo on your machine and then **git push** to copy those commits into your assignment-specific repo on GitHub.

Conclusion

These slides only scratch the surface of all you can do with **git**. I encourage you to read some of the documentation, look at man pages to see different options, talk to peers about **git**, and most importantly EXPERIMENT.