GIT

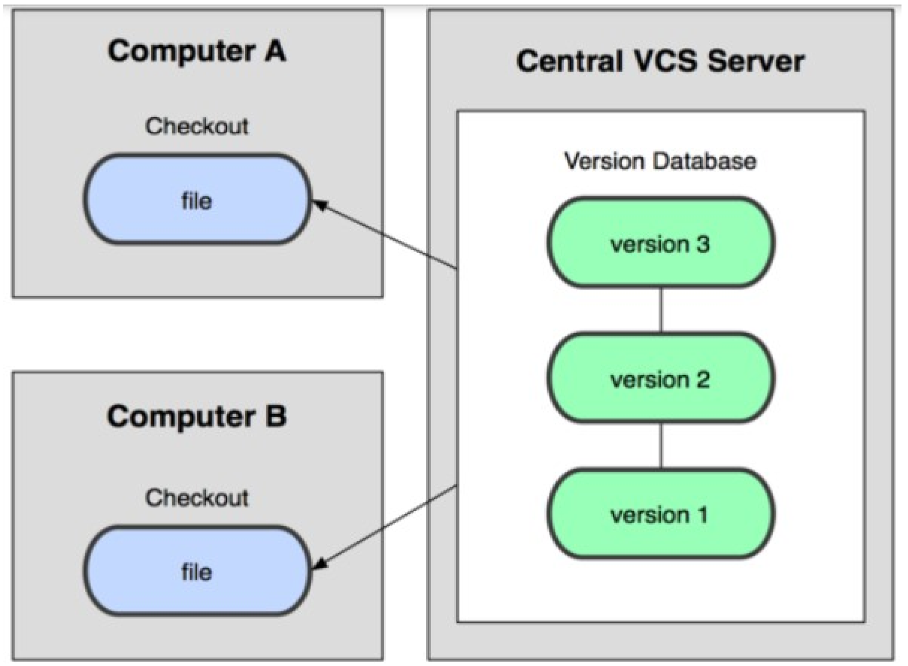
About Git

- Created by Linus Torvalds, creator of Linux, in 2005
  - Came out of Linux development community
  - Designed to do version control on Linux kernel

- Goals of Git:
  - Speed
  - Support for non-linear development (thousands of parallel branches)
  - Fully distributed
  - Able to handle large projects efficiently

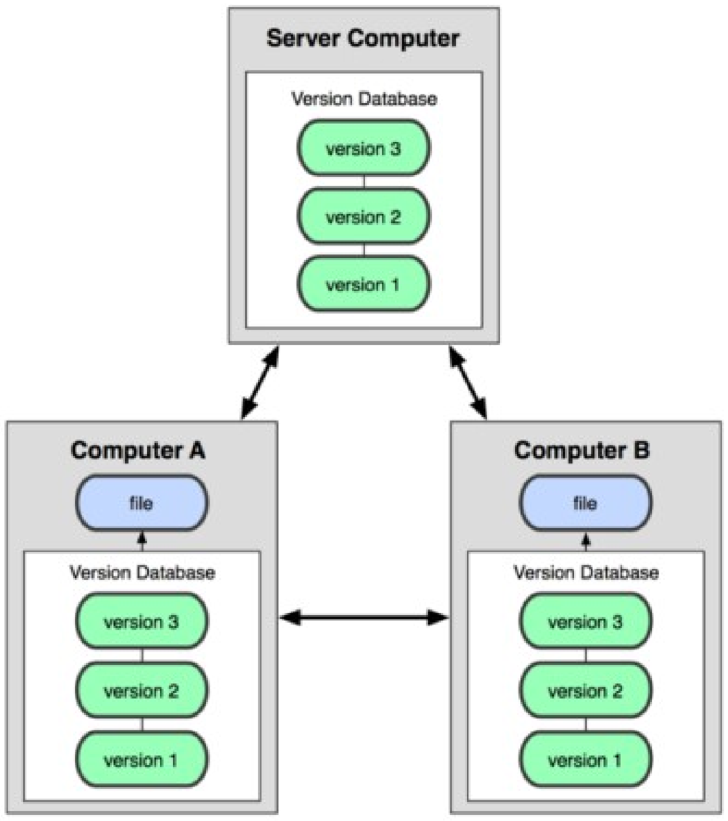
Centralized VCS

- In Subversion, CVS, Perforce, etc. A central server repository (repo) holds the "official copy" of the code
  - the server maintains the sole version history of the repo
- You make "checkouts" of it to your local copy
  - you make local modifications – your changes are not versioned
- When you're done, you "check in" back to the server
  - your checkin increments the repo's version



Distributed VCS (Git)

- In git, mercurial, etc., you don't "checkout" from a central repo
  - you "clone" it and "pull" changes from it
- Your local repo is a complete copy of everything on the remote server
  - yours is "just as good" as theirs
- Many operations are local:
  - check in/out from local repo
  - commit changes to local repo
  - local repo keeps version history
- When you're ready, you can "push" changes back to server



Advantages of Git

- Free and open source
- Fast and small
- Implicit backup
- Security
- No need of powerful hardware
- Easier branching

Terminologies

- **Local Repository:** Every VCS tool provides a private workplace as a working copy.
- **Working Directory :** The working directory is the place where files are checked out.
- **Staging Area or Index :** Whenever you do commit an operation, Git looks for the files present in the staging area. Only those files present in the staging area are considered for commit and not all the modified files.
- **Blobs :** Blob stands for **B**inary **L**arge **Ob**ject. Each version of a file is represented by blob.
- **Trees :** Tree is an object, which represents a directory.
- **Commits :** Commit holds the current state of the repository.

Terminologies

- **Branches :** Branches are used to create another line of development.
- **Tags :** Tag assigns a meaningful name with a specific version in the repository.
- **Clone :** Clone operation creates the instance of the repository.
- **Pull :** Pull operation copies the changes from a remote repository instance to a local one.
- **Push :** Push operation copies changes from a local repository instance to a remote one.
- **HEAD :** HEAD is a pointer, which always points to the latest commit in the branch.
- **Revision :** Revision represents the version of the source code.
- **URL :** URL represents the location of the Git repository.

Git Installation

- > sudo apt-get install git-core *[Installation on Ubuntu]*
- > git --version *[version check]*

- Git provides the git config tool, which allows you to set configuration variables.
- Git stores all global configurations in ***.gitconfig*** file, which is located in your home directory.
- To set these configuration values as global, add the ***--global*** option, and if you omit ***--global*** option, then your configurations are specific for the current Git repository.

Customize Git Environment

- Setting username:
  - `git config --global user.name “Alpha Beta“`
- Setting email id
  - `git config --global user.email “alpha@domain.com"`
- These information is used by Git for each commit.

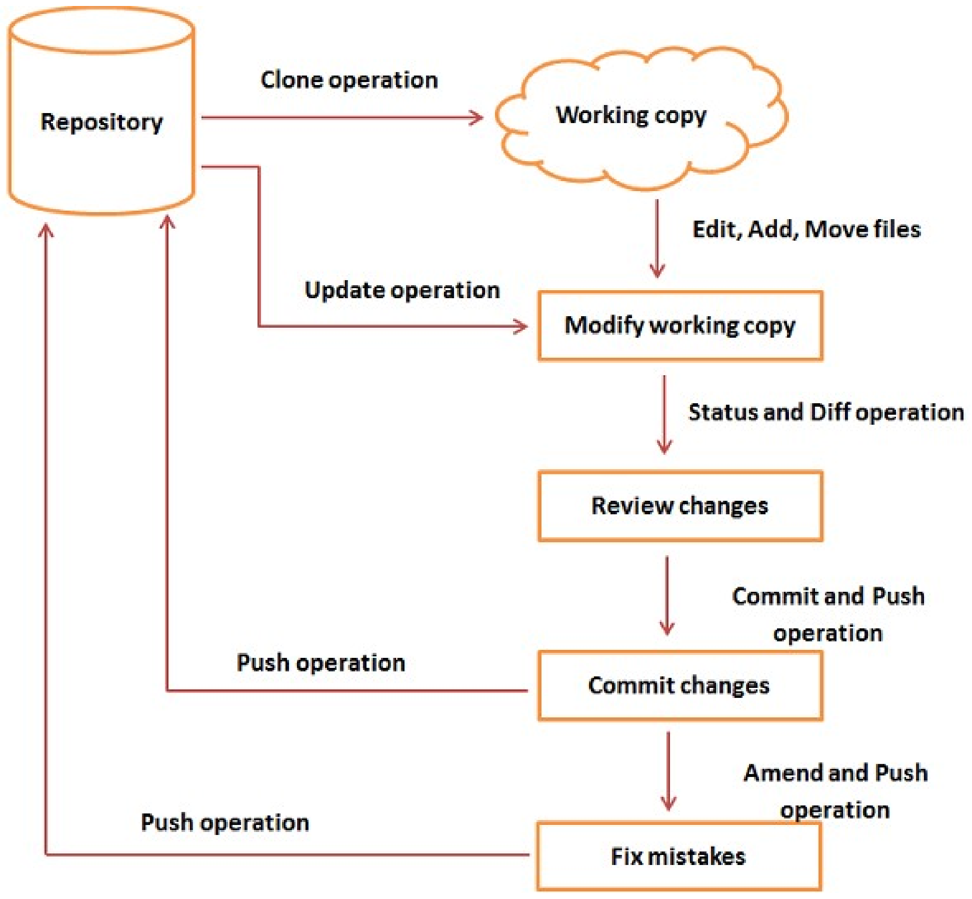
- Apart from these, you can have color highlighting for Git console, choose default editor for Git and set default merge tool.

- `git config --list` can be used to see all configurations of Git.

Git Work-Flow

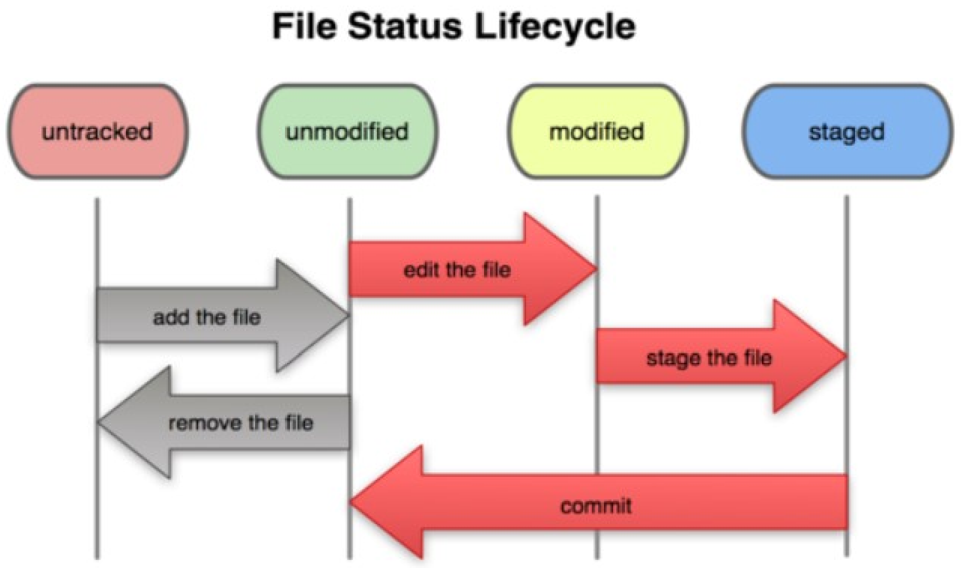
- You clone the Git repository as a working copy.
- You modify the working copy by adding/editing files.
- If necessary, you also update the working copy by taking other developer's changes.
- You review the changes before commit.
- You commit changes. If everything is fine, then you push the changes to the repository.
- After committing, if you realize something is wrong, then you correct the last commit and push the changes to the repository.

Git Work-Flow

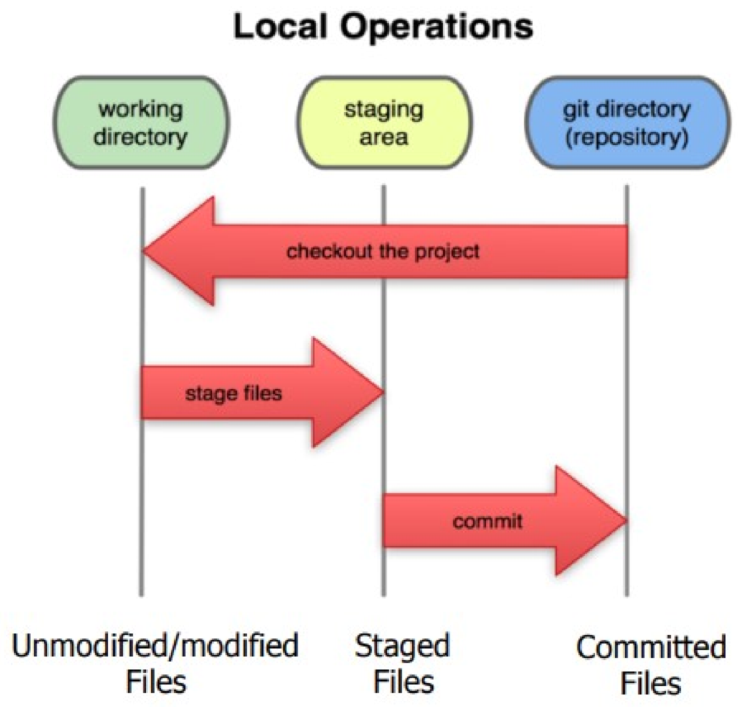


Git Work-Flow

- **Modify** files in your working directory.
- **Stage** files, adding snapshots of them to your staging area.
- **Commit**, which takes the files in the staging area and stores that snapshot permanently to your Git directory.



Local Git areas

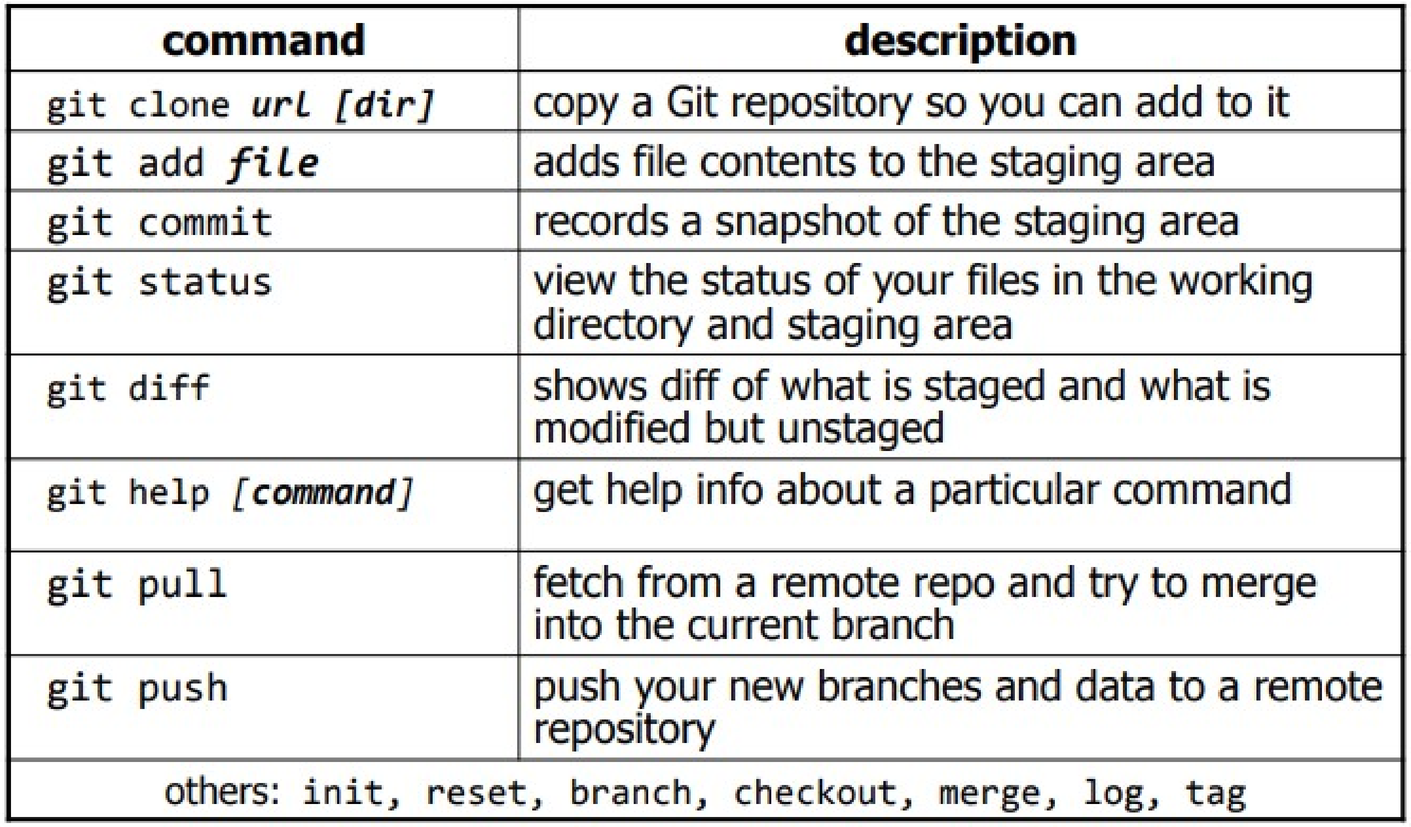


- In your local copy on git, files can be:
  - In your local repo *(committed)*
  - Checked out and modified, but not yet committed *(working copy)*
  - Or, in-between, in a "staging" area
    - Staged files are ready to be committed.
    - A commit saves a snapshot of all staged state.

Git commit checksums

- In Subversion each modification to the central repo increments the version hash of the overall repo.
  - In Git, each user has their own copy of the repo, and commits changes to their local copy of the repo before pushing to the central server.
  - So Git generates a unique SHA-1 hash (40 character string of hex digits) for every commit.
  - Refers to commits by this ID rather than a version number.
  - Often we only see the first 7 characters:
    - `1677b2d Edited first line of readme`
    - `258efa7 Added line to readme`
    - `0e52da7 Initial commit`

Git Commands



command	description
git clone *url [dir]*	copy a Git repository so you can add to it
git add *file*	adds file contents to the staging area
git commit	records a snapshot of the staging area
git status	view the status of your files in the working directory and staging area
git diff	shows diff of what is staged and what is modified but unstaged
git help *[command]*	get help info about a particular command
git pull	fetch from a remote repo and try to merge into the current branch
git push	push your new branches and data to a remote repository
others: init, reset, branch, checkout, merge, log, tag	

Create and fill a repository

1. `cd` to the project directory you want to use
2. Type in `git init`
    - This creates the repository (a directory named `.git`)
    - You seldom (if ever) need to look inside this directory
3. Type in `git add .`
    - The period at the end is part of this command!
        - Period means “this directory”
    - This adds all your current files to the repository
4. Type in **`git commit –m "Initial commit"`**
    - You can use a different commit message, if you like

Clone a repository from elsewhere

- `git clone` *`URL`*
- `git clone` *`URL mypath`*
  - These make an exact copy of the repository at the given URL
- `git clone git://github.com/`*`rest_of_path/file.git`*
  - Github is the most popular (free) public repository
- All repositories are equal
  - But you can treat some particular repository (such as one on Github) as the “master” directory
- Typically, each team member works in his/her own repository, and “merges” with other repositories as appropriate

The repository

- Your top-level **working directory** contains everything about your project
  - The working directory probably contains many subdirectories—source code, binaries, documentation, data files, etc.
  - One of these subdirectories, named `.git`, is your repository
- At any time, you can take a "snapshot" of everything (or selected things) in your project directory, and put it in your repository
  - This "snapshot" is called a commit object
  - The commit object contains (1) a set of files, (2) references to the "parents" of the commit object, and (3) a unique "SHA1" name
  - Commit objects do *not* require huge amounts of memory
- You can work as much as you like in your working directory, but the repository isn't updated until you `commit` something

init and the .git repository

- When you said `git init` in your project directory, or when you cloned an existing project, you created a repository
  - The repository is a subdirectory named .git containing various files
  - The dot indicates a “hidden” directory
  - You do *not* work directly with the contents of that directory; various git commands do that for you
  - You *do* need a basic understanding of what is in the repository

Making commits

- You do your work in your project directory, as usual
- If you create new files and/or folders, they are *not tracked* by Git unless you ask it to do so
    - `git add` *`newFile1 newFolder1 newFolder2 newFile2`*
- Committing makes a “snapshot” of everything being tracked into your repository
    - A message telling what you have done is required
    - `git commit –m “Uncrevulated the conundrum bar”`
    - `git commit`
        - This version opens an editor for you the enter the message
        - To finish, save and quit the editor
- Format of the commit message
    - One line containing the complete summary
    - If more than one line, the second line must be blank

Commits and graphs

- A commit is when you tell git that a change (or addition) you have made is ready to be included in the project
- When you commit your change to git, it creates a commit object
  - A commit object represents the complete state of the project, including all the files in the project
  - The *very first* commit object has no "parents"
  - Usually, you take some commit object, make some changes, and create a new commit object; the original commit object is the parent of the new commit object
    - Hence, most commit objects have a single parent
  - You can also merge two commit objects to form a new one
    - The new commit object has two parents
- Hence, commit objects form a **directed graph**
  - Git is all about using and manipulating this graph

Working with your own repository

- A head is a reference to a commit object
- The "current head" is called HEAD (all caps)
- Usually, you will take HEAD (the current commit object), make some changes to it, and commit the changes, creating a new current commit object
  - This results in a linear graph: A → B → C → …→ HEAD

- You can also take any previous commit object, make changes to it, and commit those changes
  - This creates a branch in the graph of commit objects
- You can merge any previous commit objects
  - This joins branches in the commit graph

Commit messages

- In git, “Commits are cheap.” Do them often.
- When you commit, you must provide a one-line message stating what you have done
  - Terrible message: “Fixed a bunch of things”
  - Better message: “Corrected the calculation of median scores”
- Commit messages can be very helpful, to yourself as well as to your team members
- You can’t say much in one line, so commit often

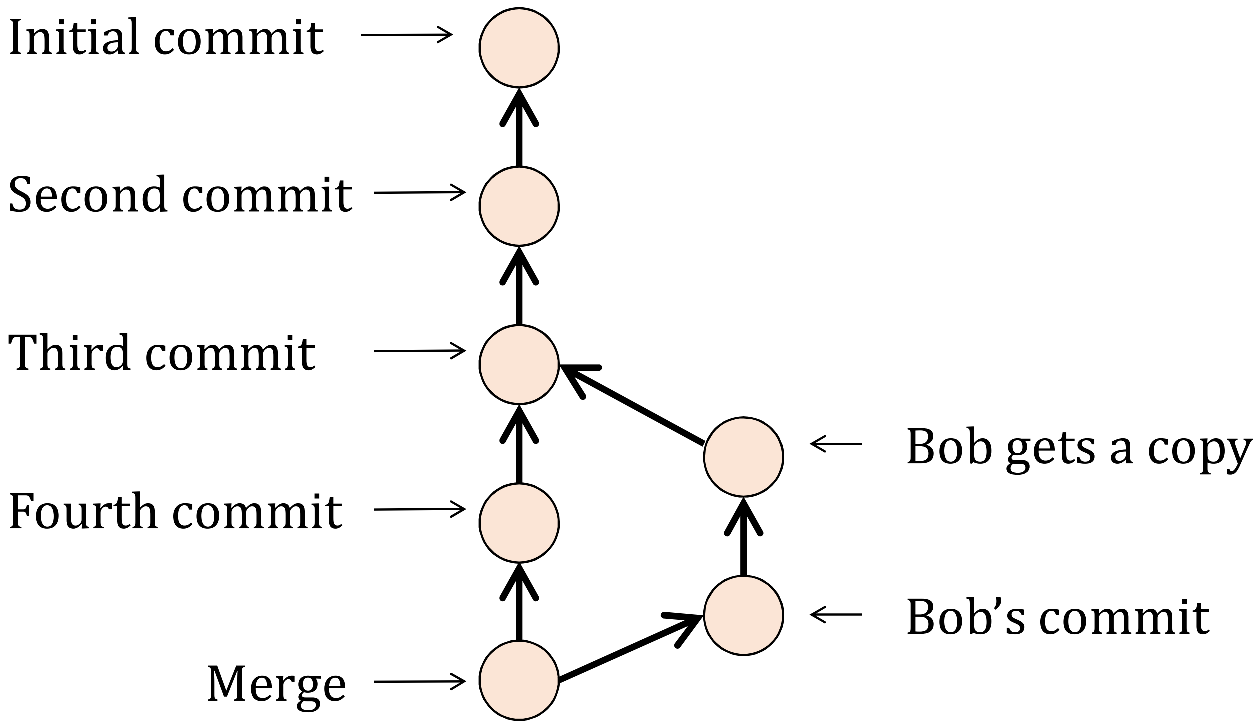
Working with others

- All repositories are equal, but it is convenient to have one central repository in the cloud
- Here’s what you normally do:
  - Download the current HEAD from the central repository
  - Make your changes
  - Commit your changes to your local repository
  - Check to make sure someone else on your team hasn’t updated the central repository since you got it
  - Upload your changes to the central repository
- If the central repository *has* changed since you got it:
  - It is *your* responsibility to **merge your two versions**
    - This is a strong incentive to commit and upload often!
  - Git can often do this for you, if there aren’t incompatible changes

Typical workflow

- `git pull` *`remote_repository`*
  - Get changes from a remote repository and merge them into your own repository
- `git status`
  - See what Git thinks is going on
  - Use this frequently!
- Work on your files (remember to `add` any new ones)
- `git commit –m` *`“What I did”`*
- `git push`

Multiple versions



Keeping it simple

- If you:
  - Make sure you are current with the central repository
  - Make some improvements to your code
  - Update the central repository before anyone else does
- Then you don’t have to worry about resolving conflicts or working with multiple branches
  - All the complexity in git comes from dealing with these
- Therefore:
  - Make sure you are up-to-date before starting to work
  - Commit and update the central repository frequently
- If you need help: https://help.github.com/