CSE 391
Lecture 9

Version control with Git

slides created by Ruth Anderson, Marty Stepp, and Brett Wortzman

images from http://git-scm.com/book/en/

http://www.cs.washington.edu/391/

Problems Working Alone

- Ever done one of the following?
  - Had code that worked, made a bunch of changes and saved it, which broke the code, and now you just want the working version back…
  - Accidentally deleted a critical file, hundreds of lines of code gone…
  - Somehow messed up the structure/contents of your code base, and want to just "undo" the crazy action you just did
  - Hard drive crash!!!! Everything's gone, the day before deadline.

- Possible options:
  - Save as (MyClass-v1.java)
    - Ugh. Just ugh. And now a single line change results in duplicating the entire file…

Problems Working in teams

- Whose computer stores the "official" copy of the project?
  - Can we store the project files in a neutral "official" location?
- Will we be able to read/write each other's changes?
  - Do we have the right file permissions?
  - Lets just email changed files back and forth! Yay!
- What happens if we both try to edit the same file?
  - Bill just overwrote a file I worked on for 6 hours!
- What happens if we make a mistake and corrupt an important file?
  - Is there a way to keep backups of our project files?
- How do I know what code each teammate is working on?

Solution: Version Control

- **version control system**: Software that tracks and manages changes to a set of files and resources.

- You use version control all the time
  - Built into word processors/spreadsheets/presentation software
    - The magical "undo" button takes you back to "the version before my last action"
  - Wikis
    - Wikis are all about version control, managing updates, and allowing rollbacks to previous versions

Software Version Control

- Many version control systems are designed and used especially for software engineering projects
  - examples: CVS, Subversion (SVN), **Git**, Monotone, BitKeeper, Perforce

- helps teams to work together on code projects
  - a shared copy of all code files that all users can access
  - keeps current versions of all files, and backups of past versions
  - can see what files others have modified and view the changes
  - manages conflicts when multiple users modify the same file

  - not particular to source code; can be used for papers, photos, etc.
    - but often works best with plain text/code files

Repositories

- **Repository (aka "repo")**: a location storing a copy of all files.
  - you don't edit files directly in the repo;
  - you edit a local **working copy** or "working tree"
  - then you **commit** your edited files into the repo

- There may be only one repository that all users share (CVS, Subversion)
- Or each user could also have their own copy of the repository (Git, Mercurial)

- Files in your working directory must be **add**ed to the repo in order to be tracked.

What to put in a Repo?

- Everything needed to create your project:
  - Source code (Examples: .java, .c, .h, .cpp)
  - Build files (Makefile, build.xml)
  - Other resources needed to build your project: icons, text etc.

- Things generally NOT put in a repo (these can be easily re-created and just take up space):
  - Object files (.o)
  - Executables (.exe)
  - Machine-specific configuration files

Repository Location

- Can create the repository anywhere
  - Can be on the same computer that you're going to work on, which might be ok for a personal project where you just want rollback protection
- But, usually you want the repository to be robust:
  - On a computer that's up and running 24/7
    - Everyone always has access to the project
  - On a computer that has a redundant file system (ie RAID)
    - No more worries about that hard disk crash wiping away your project!
- Options:
  - attu, CSE GitLab, GitHub (do NOT use GitHub for homework!!!)

Aside: So what is GitHub?

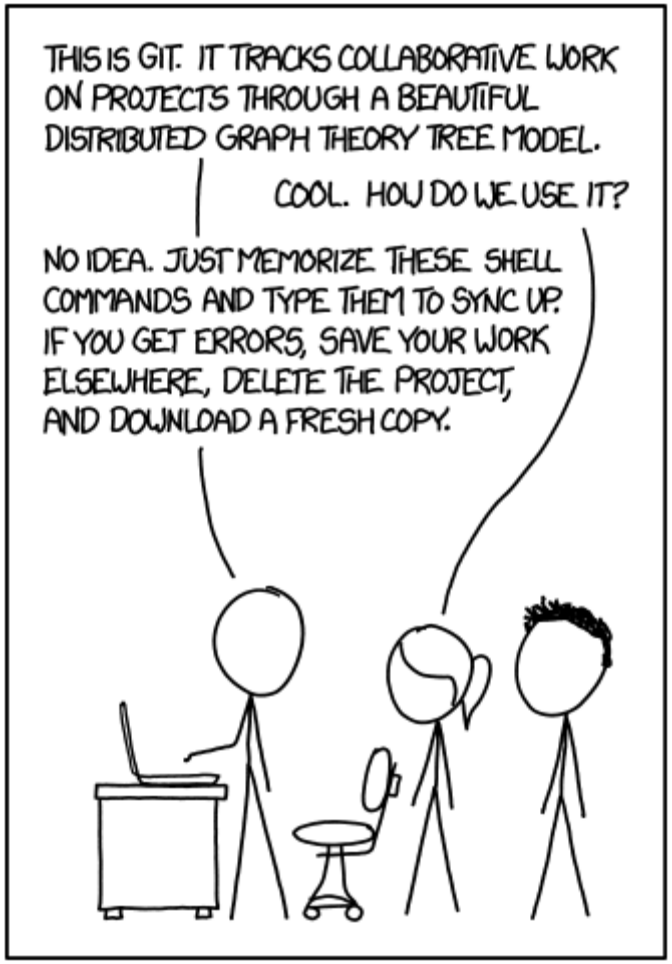
- GitHub.com is a site for online storage of Git repositories.
- Many open source projects use it, such as the Linux kernel.
- You can get free space for open source projects or you can pay for private projects.
- Do NOT use GitHub to store your homework!!

Question: Do I have to use GitHub to use Git?

Answer: No!

- you can use Git completely locally for your own purposes, or
- you can use the CSE GitLab server, or
- you could share a repo with users on the same file system (e.g. attu) as long everyone has the needed file permissions.

Git



HTTP://XKCD.COM/1597/

Git Resources

- At the command line: (where <verb> = config, add, commit, etc.)

```
$ git help <verb>
$ git <verb> --help
$ man git-<verb>
```

- Free on-line book: https://git-scm.com/book/en/v2
- Git tutorial: http://schacon.github.com/git/gittutorial.html
- Reference page for Git: http://gitref.org/index.html
- Git website: http://git-scm.com/

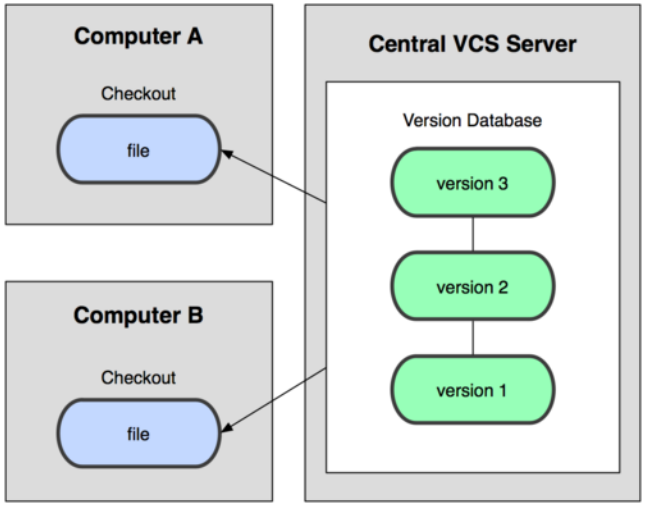
- Git for Computer Scientists: http://eagain.net/articles/git-for-computer-scientists/

History of Git

- Came out of Linux development community
- Linus Torvalds, 2005
- Initial goals:
  - Speed
  - Support for non-linear development (thousands of parallel branches)
  - Fully distributed
  - Able to handle large projects like Linux efficiently

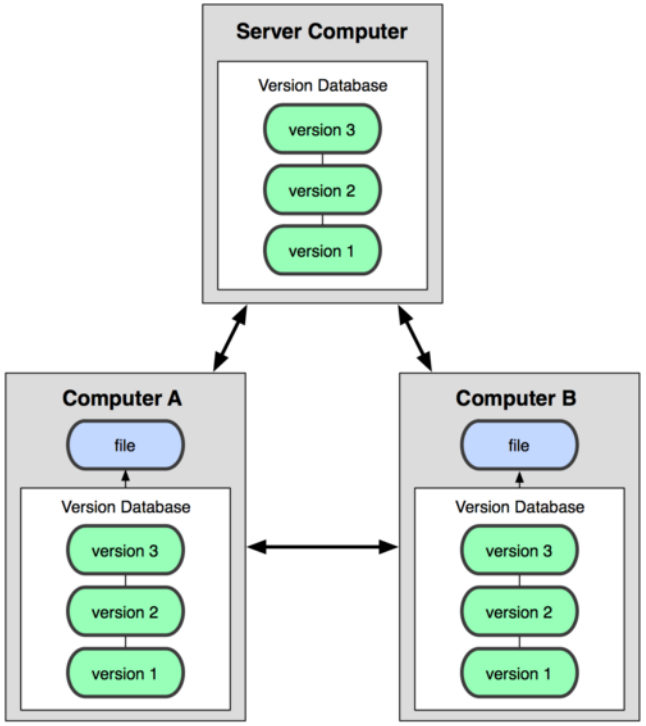
Git uses a distributed model

Centralized Model



(CVS, Subversion, Perforce)

Distributed Model



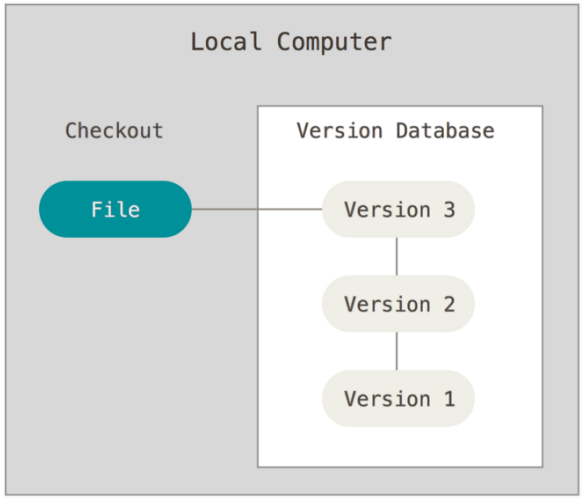
(**Git**, Mercurial)

Result: Many operations are local

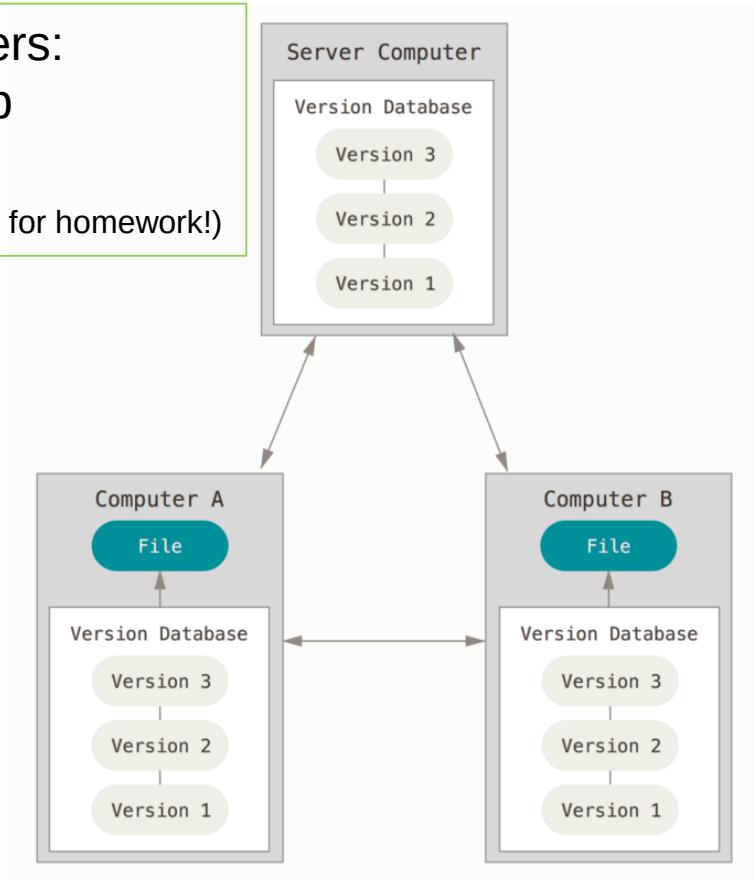
Ways to use Git

Possible servers:

- CSE GitLab
- attu
- GitHub (NOT for homework!)

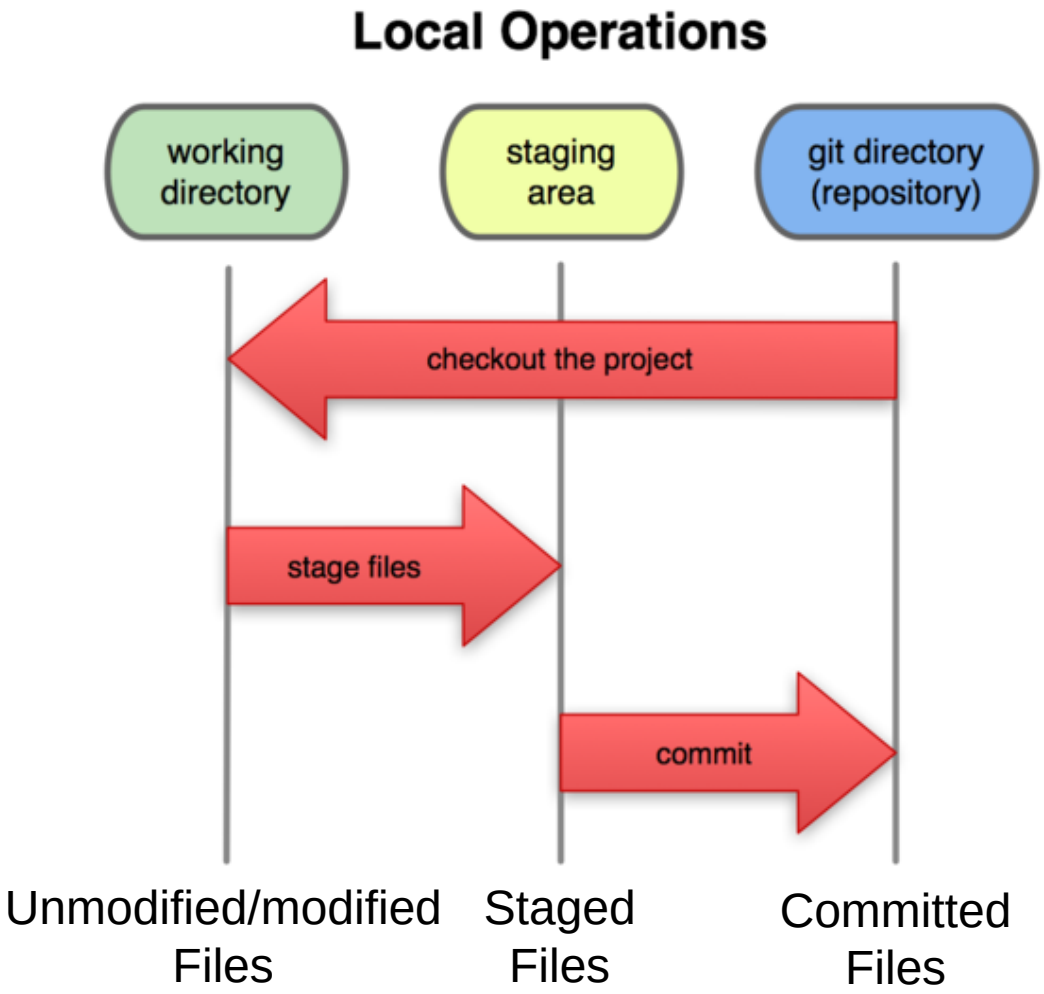


Using Git on your own computer, one user



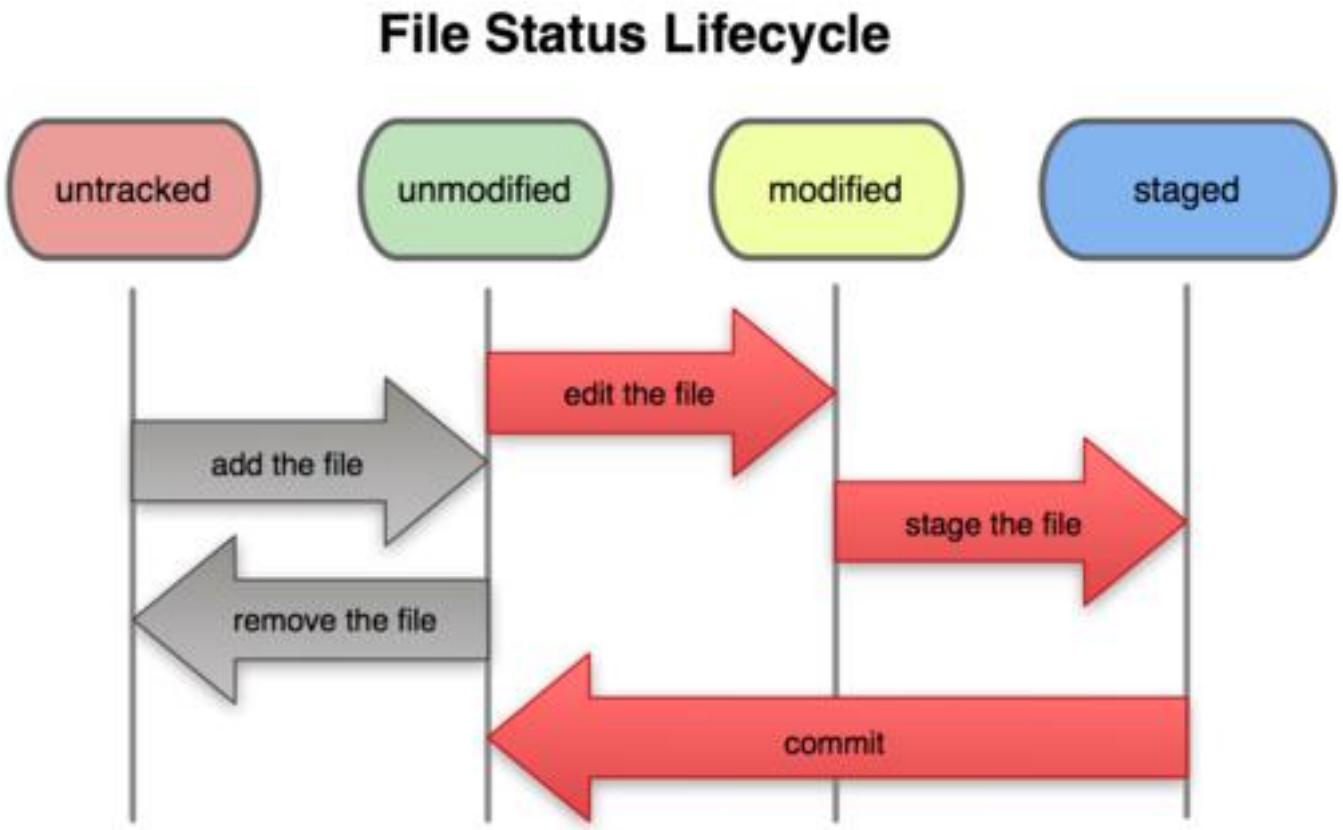
Using Git on multiple computers, multiple users or one user on multiple computers

A Local Git project has three areas



Note: working directory sometimes called the “working tree”, staging area sometimes called the “index”.

Git file lifecycle



Basic Workflow

Basic Git workflow:

1. **Modify** files in your working directory.
2. **Stage** files, adding snapshots of them to your staging area.
3. Do a **commit**, which takes the files as they are in the staging area and stores that snapshot permanently to your local Git directory (your local copy of the repo).

- Notes:
  - If a particular version of a file is in your local **git directory**, it's considered **committed**.
  - If it's modified but has been added to the **staging area**, it is **staged**.
  - If it was **changed** since it was checked out but has not been staged, it is **modified**.

CSE GitLab

Everyone in this course has been given an account on CSE GitLab

To use CSE GitLab:

1. Log onto CSE GitLab: https://gitlab.cs.washington.edu/
   - IMPORTANT: **If you have a CSENetID use that**, otherwise use your UWNetID
2. Add an ssh key: https://gitlab.cs.washington.edu/help/ssh/README.md
   - Follow the instructions in README.md which say to type:

     ```
     ssh-keygen -t rsa -C "yourUWNetID@uw.edu" -b 4096
     ```

     - Just hit return to accept the default file location and **you do not need a password** so **hit return both times** when **prompted for a password**
   - Then type: `cat ~/.ssh/id_rsa.pub`
   - Copy-paste the key to the 'SSH Keys' section under 'Profile Settings' in your user profile on CSE GitLab. Or go directly here:

     `https://gitlab.cs.washington.edu/profile/keys`

Get ready to use Git!

1. Set the name and email for Git to use when you commit:

```
$ git config --global user.name "Bugs Bunny"
$ git config --global user.email bugs@gmail.com
```

   - You can call `git config --list` to verify these are set.
   - These will be set globally for all Git projects you work with.
   - You can set variables on a project-only basis by **not** using the `--global` flag.

- The latest version of git will also prompt you that `push.default` is not set, you can make this warning go away with:

```
$ git config --global push.default simple
```

- You can also **set the editor** used for writing commit messages:

```
$ git config --global core.editor emacs
```

(it is vim by default)

vim tips: "a" add, "esc" when done adding, "wq:" to save and quit

vim editor: http://www.gentoo.org/doc/en/vi-guide.xml

vim ref card: http://tnerual.eriogerg.free.fr/vimqrc.pdf

Create a local copy of a repo

2. Two common scenarios: (only do one of these)

   a) To **clone an already existing repo** to your current directory:

   ```
   $ git clone <url> [local dir name]
   ```

   This will create a directory named *local dir name*, containing a working copy of the files from the repo, and a `.git` directory which you can ignore (used to hold the staging area and your local repo)

Example: `git clone git@gitlab.cs.washington.edu:rea/superTest.git`

   b) To **create a Git repo** in your current directory:

   ```
   $ git init
   ```

   This will create a `.git` directory in your current directory which you can ignore (used to hold the staging area and your local repo).

   Then you can commit files in your current directory into the local repo:

   ```
   $ git add file1.java
   $ git commit -m "initial project version"
   ```

Git commands

command	description
`git clone` ***`url [dir]`***	copy a git repository so you can add to it
`git add` ***`files`***	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`	

Adding & Committing files

1. The first time we ask a file to be tracked, *and* **every time before we commit a file** we must add it to the staging area:

```
$ git add README.txt hello.java
```

This takes a snapshot of these files at this point in time and adds it to the staging area.

Note: To unstage a change on a file before you have committed it:

```
$ git reset HEAD filename
```

2. To move staged changes into the local repo we commit:

```
$ git commit –m “Fixing bug #22”
```

Note: You can edit your most recent commit message (if you have not pushed your commit yet) using: `git commit --amend`

Note: These commands are just acting on **your local version of repo**.

Use Good Commit Messages

HTTP://XKCD.COM/1296/

Status and Diff

- To view the **status** of your files in the **working directory** and **staging area**:

```
$ git status          or
$ git status -s
```

(`-s` shows a short one line version)

- To see **difference** between your **working directory** and the **staging area** (This shows what is modified but unstaged):

```
$ git diff
```

- To see difference between the **staging area** and your **local copy of the repo** (This shows staged changes): (`--staged is synonymous`)

```
$ git diff --cached
```

After editing a file…

```
$emacs rea.txt
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   rea.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
$ git status -s
 M rea.txt                                   ← Note: M is in second column = “working  tree”
$ git diff                          ← Shows modifications that have not been staged.
diff --git a/rea.txt b/rea.txt
index 66b293d..90b65fd 100644
--- a/rea.txt
+++ b/rea.txt
@@ -1,2 +1,4 @@
 Here is rea's file.
+
+One new line added.
$ git diff --cached                 ← Shows nothing, no modifications have been staged yet.
$
```

Viewing logs

To see a log of all changes in your **local repo**:

- `$ git log` or
- `$ git log --oneline` (to show a shorter version)

1677b2d Edited first line of readme

258efa7 Added line to readme

0e52da7 Initial commit

- `git log -5` (to show only the 5 most recent updates, etc.)

Note: changes will be listed by commitID #, (SHA-1 hash)

Note: changes made to the remote repo before the last time you cloned/pulled from it will also be included here

Pulling and Pushing

Good practice:

1. **Add** and **Commit** your changes to your local repo
2. **Pull** from remote repo to get most recent changes (fix conflicts if necessary, then add and commit those changes to your local repo)
3. **Push** your changes to the remote repo

To fetch the most recent updates from the remote repo into your local repo, and put them into your working directory:

```
$ git pull origin master
```

To push your changes from your local repo to the remote repo:

```
$ git push origin master
```

Notes: `origin` = an alias for the URL you cloned from

`master` = the remote branch you are pulling from/pushing to,
(the local branch you are pulling to/pushing from is your current branch)

Avoiding Common Problems

From CSE 331: http://courses.cs.washington.edu/courses/cse331/18wi/tools/versioncontrol.html#git-pitfalls

- Do not edit the repository (the .git directory) manually. It wasn't designed for modifications by humans.
- Try not to make many drastic changes at once. Instead, make multiple commits, each of which has a single logical purpose. This will minimize merge conflicts. This is good coding practice in general.
- Always **`git pull`** before editing a file. It's easy to forget this. If you forget, you may end up editing an outdated version, which can cause nasty merge conflicts.
- Don't forget **`git push`** after you have made and committed changes. They are not copied to the remote repository until you do a push.

Branching

To create a branch called experimental:

- `$ git branch experimental`

To list all branches: (* shows which one you are currently on)

- `$ git branch`

To switch to the experimental branch:

- `$ git checkout experimental`

Later on, changes between the two branches differ, to merge changes from experimental into the master:

- `$ git checkout master`
- `$ git merge experimental`

Note: `git log --graph` can be useful for showing branches.

Note: These branches are in *your local repo*!

SVN vs. Git

- SVN:
  - central repository approach – the main repository is the only “true” source, only the main repository has the complete file history
  - Users check out local copies of the current version
- Git:
  - Distributed repository approach – every checkout of the repository is a full fledged repository, complete with history
  - Greater redundancy and speed
  - Branching and merging repositories is more heavily used as a result

Wrap-up

- You *will* use version control software when working on projects, both here and in industry
  - Rather foolish not to
  - Advice: just set up a repository, even for small projects, it will save you time and hassle

- HW9 (Git) has more details and walks you through creating a Git repo and adding to a shared repo.