Git for Version Control

These slides are based on slides by Ruth Anderson for cse 390a at UW.

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

About Git

- Created by Linus Torvalds, creator of Linux, in 2005
  - Cane 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)
  - Full distributed
  - Able to handle large projects efficiently

- A "git" is a cranky old man. Linus meant himself.

Git Resources

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

  $ git help <verb>

  $ git <verb> --help

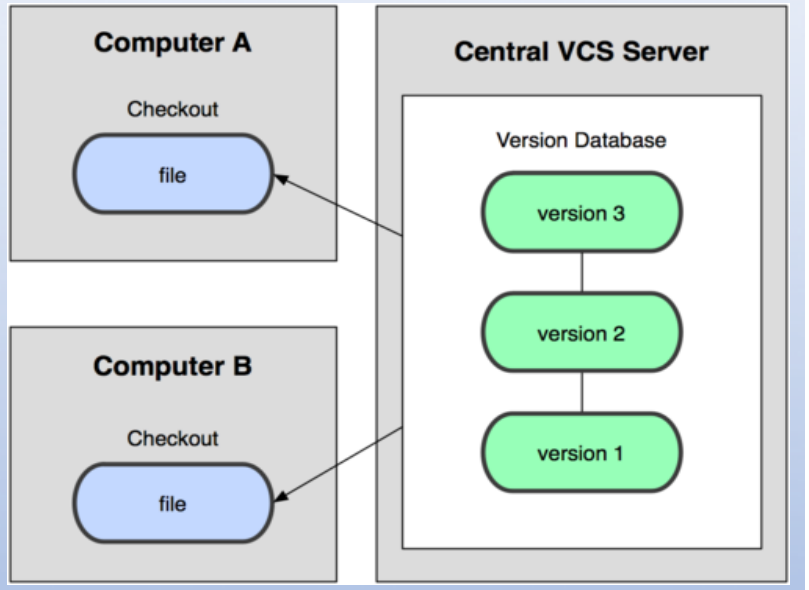
  $ man git-<verb>

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

Centralized VCS

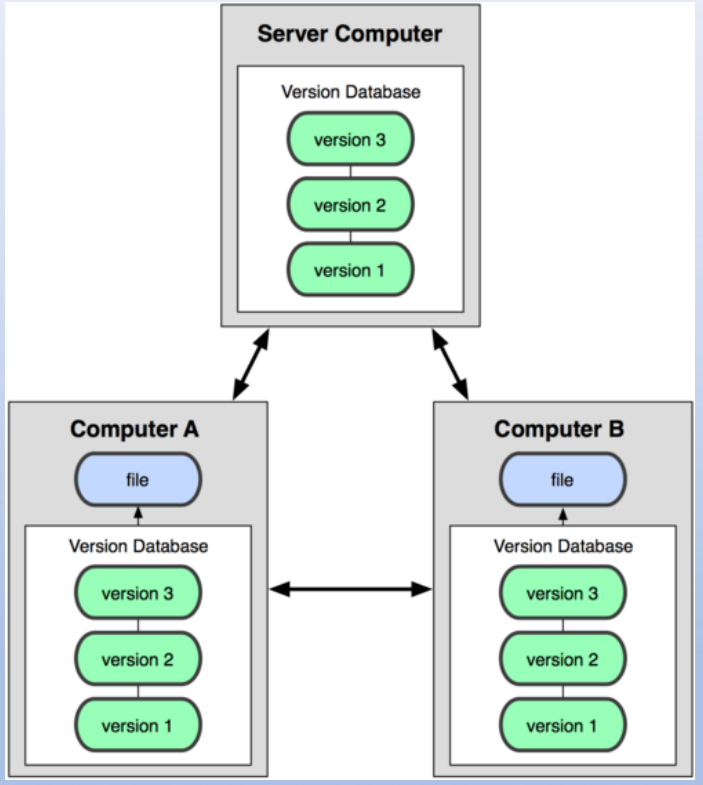
- In Subversion, CVS, Perforce, etc.

  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

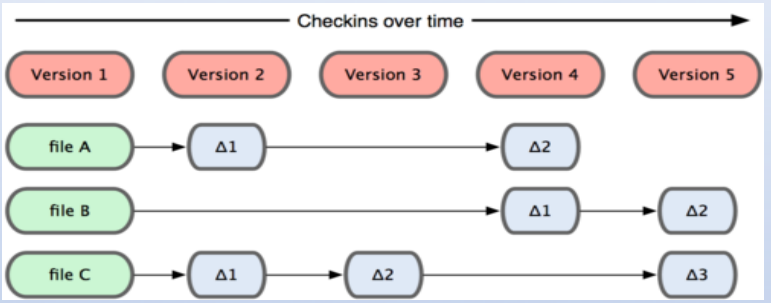


- When you're ready, you can "push" changes back to server

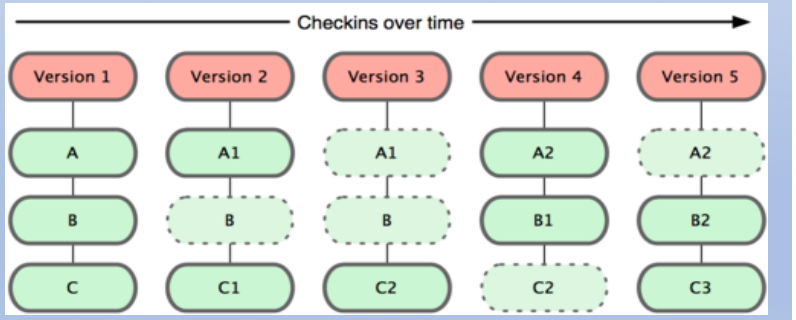
Git Snapshots

- Centralized VCS like Subversion track version data on each individual file.
- Git keeps "snapshots" of the entire state of the project.
  - Each version of the overall code has a copy of each file in it.
  - Some files change from one version to the next, some do not.
  - More redundancy, but faster.

Subversion

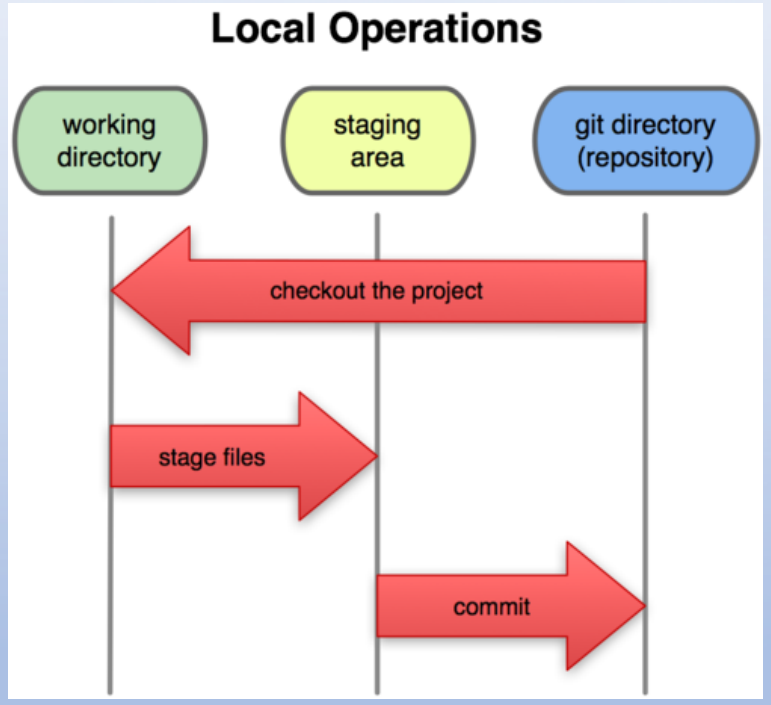


Git



A Local Git project has three areas

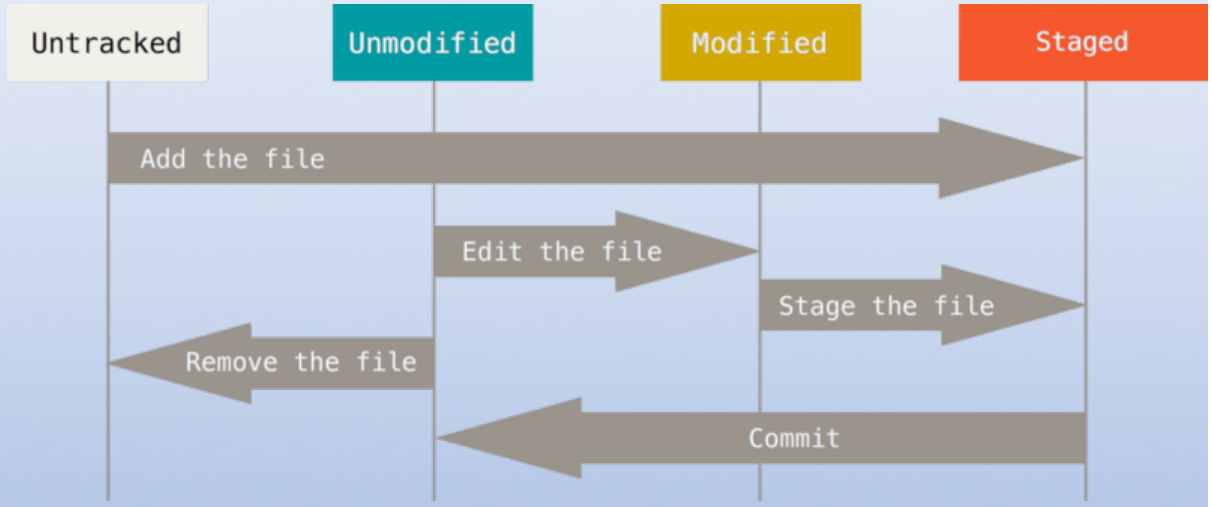
- files can be:
  - In your local repo
    - (committed)
  - Tracked and modified, but not yet staged
    - (working copy)
  - Or in-between, in a "staging" area
    - Staged files are ready to be committed.
    - A commit saves a snapshot of all staged state.



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

Basic Git Workflow

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



Basic Workflow

- If a particular version of a file is in the **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 committed but has not been staged, it is **modified**.

Git uses checksums

- In Subversion each modification to the **central** repo increments the version # of the overall repo.
- How would this numbering scheme work **when 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?
- Instead, Git generates a unique **SHA-1 hash** – 40 character string of hex digits, for every commit.
  - Refer 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
```

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 (get a github student pack – free private repos)

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

Answer: No!

- you can use Git completely locally for your own purposes, or
- you or someone else could set up a server to share files, or
- you could share a repo with users on the same file system

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 also set variables on a project-only basis by not using the **`--global`** flag.
- You can also set the editor that is used for writing commit messages:
  $ `git config --global core.editor emacs` (it is vim by default)

Create a Git 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 (used to hold the staging area and your actual repo)

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

```
$ git init
```

   This will create a `.git` directory in your current directory.

   Then you can commit files in that directory into the 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` ***`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`	

Add and Commit file

- 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
```

  - Takes a snapshot of these files at this point in time and adds it to the staging area.
  - In older VCS, "add" means "start tracking this file." In Git, "add" means "add to staging area" so it will be part of the next commit.

- To move staged changes into the repo, we commit:

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

Note: To undo changes on a file before you have committed it:

```
$ git reset HEAD -- filename   (unstages the file)
```

Note: To unmodify a modified file:

```
$ git checkout -- filename    (undoes your changes)
```

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

Viewing Changes: status

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

`$ git status` or `$ git status -s` (short version)

- To see what is modified but unstaged:

```
$ git diff
```

- To see staged changes:

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

An Example Workflow: After Editing a File...

```
[rea@attu1 superstar]$ emacs rea.txt
[rea@attu1 superstar]$ 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")
[rea@attu1 superstar]$ git status -s
 M rea.txt
[rea@attu1 superstar]$ git diff   ← shows changes that are not staged yet
diff --git a/rea.txt b/rea.txt
index e69de29..c9d8d02 100644
--- a/rea.txt
+++ b/rea.txt
@@ -0,0 +1 @@    ← 1 line added in postimage
+Added something to rea.txt   ← added line

[rea@attu1 superstar]$ git diff -cached
[rea@attu1 superstar]$
```

After adding file to staging area…

[rea@attu1 superstar]$ **git add rea.txt** ← **staging modified file**

[rea@attu1 superstar]$ **git status**

```
On branch master
Changes to be committed:
     (use "git reset HEAD <file>..." to unstage)

           modified:   rea.txt
```

[rea@attu1 superstar]$ **git diff** ← **Note: Shows nothing, no modifications that have not been staged.**

[rea@attu1 superstar]$ **git diff –cached** ← **Note: Shows staged modifications.**

```
diff --git a/rea.txt b/rea.txt
index e69de29..c9d8d02 100644
--- a/rea.txt+++ b/rea.txt
@@ -0,0 +1 @@
+Added something to rea.txt
```

Viewing logs

To see a log of all commits in your local repo:

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

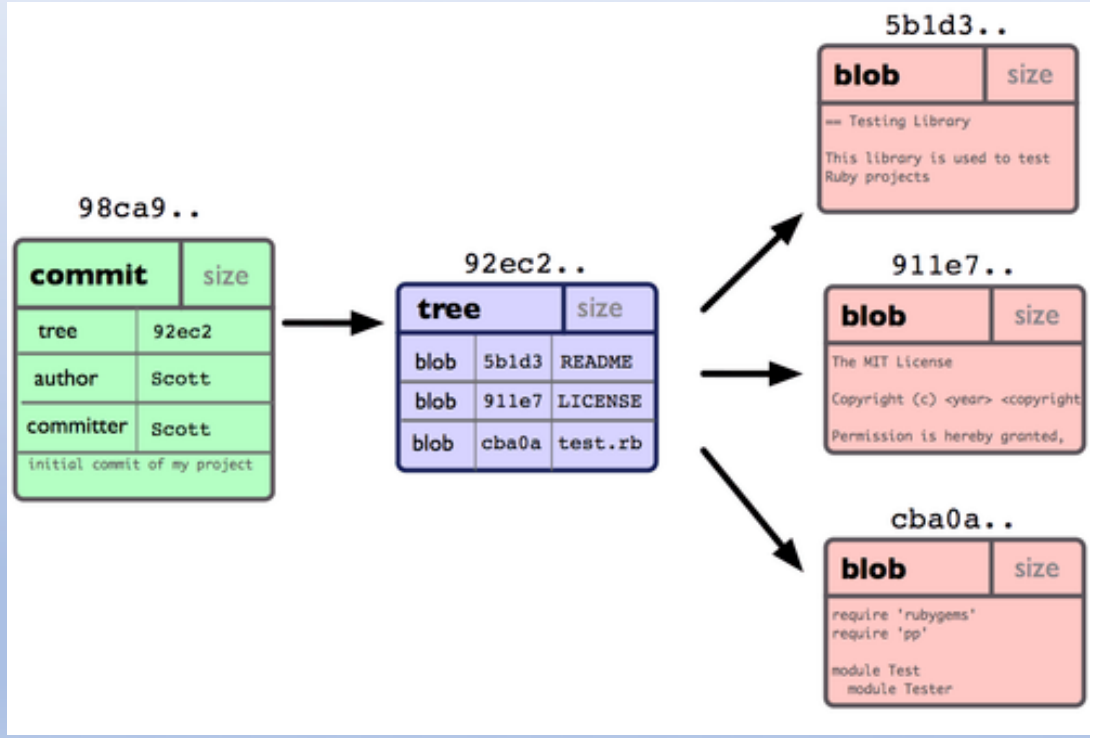
```
1677b2d Edited first line of readme
258efa7 Added line to readme
0e52da7 Initial commit
```

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

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

Aside: How Does Git Store Data?

- Suppose you have three files in your working directory
- You stage all files and commit:
  - `git add .`
  - `git commit –m "initial commit of project"`
- Git stores a commit object which points to a tree of objects associated with your files
- blob object: represents file
- tree object: list directory contents
- commit object: commit metadata, pointer to tree



Single commit repository data

Pulling and Pushing: Interaction w/ Remote Repo

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, add and commit them 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 (default name for main remote repo

`master` = main branch's default name

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*!

Merge Conflicts

- The conflicting file will contain <<< and >>> sections to indicate where Git was unable to resolve a conflict:

```
<<<<<<< HEAD:index.html
<div id="footer">todo: message here</div>
=======
<div id="footer">
  thanks for visiting our site
</div>
>>>>>>> SpecialBranch:index.html
```

- Find all such sections, and edit them to the proper state (whichever of the two versions is newer / better / more correct).

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

Questions?

To do...

- Install git on your machine
- Make a directory and cd to it
  - mkdir <myDir>
  - cd <myDir>
- Initialize this directory as a local git repo
  - git init
- Add a new file to your directory
- Look at the status of your directory in the context of Git:
  - git status
- Tell Git to track your file:
  - git add <yourFileName> or git add .
- Look at the status of your directory again
- Now do you first commit (don't forget to add a description)
  - git commit –m "initial commit"
- Look at the status of your directory again