Introduction to **Git** and **Github**

By Athol Whitten for Super Advanced R (FISH512)
University of Washington (SAFS), June 2014

NOTE: This presentation links to, borrows from, and includes images from the Git Book, Github Website, and Git Reference websites.

http://git-scm.com/about for general introduction to Git.

http://git-scm.com/book for the nitty gritty.

https://github.com/ for overview of Github.

https://octodex.github.com/ for the Octocat images.

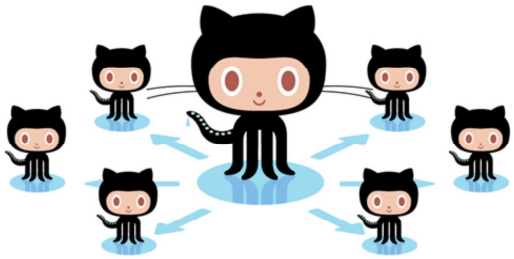
What is **Git**?

- Git is a distributed version control system
- Written by Linus Torvalds
- Designed to handle both large and small projects
- Allows tracking, branching, merging, and multiple workflows

Click for more git info

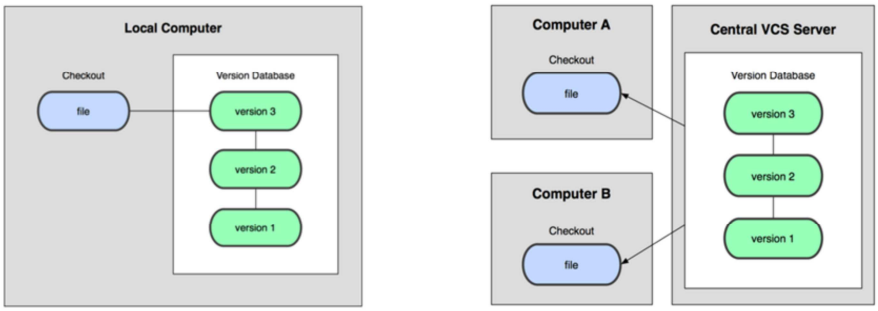
What is **Github**?

- Online host for **Git** repositories
- Allows for '**Social Coding**'
- Free for public projects
  *private repositories possible too
- Acts as server for **collaborative** projects



About **version control**

- Version control records changes to a file or set of files over time



Local System **Centralized Version Control**

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later.

A VCS allows you to: revert files back to a previous state, revert the entire project back to a previous state, review changes made over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more.

Using a VCS also means that if you screw things up or lose files, you can generally recover easily. In addition, you get all this for very little overhead.

Local Version Control Systems

Many people's version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they're clever). This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you're in and accidentally write to the wrong file or copy over files you don't mean to.

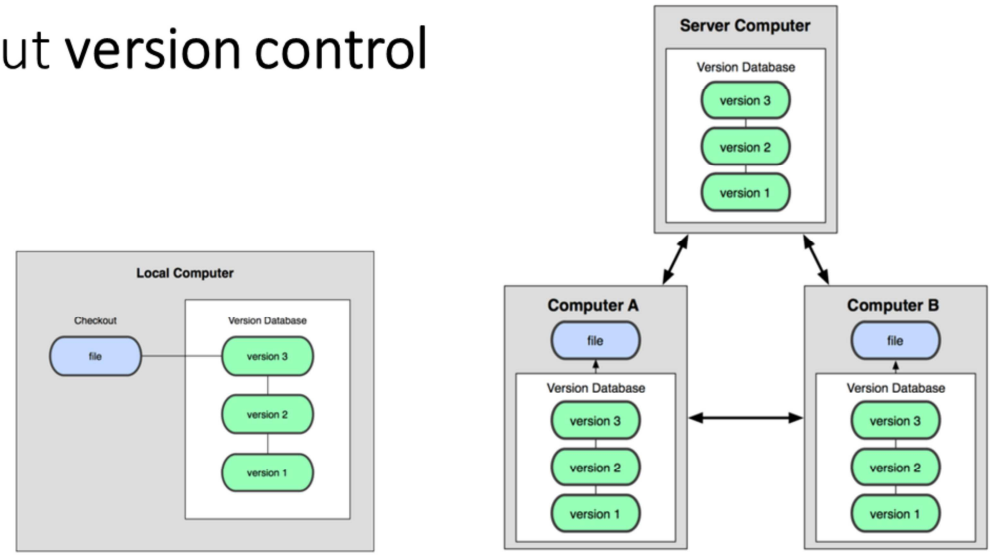
Centralized Version Control Systems

The next major issue that people encounter is that they need to collaborate with developers on other systems. To deal with this problem, Centralized Version Control Systems (CVCSs) were developed. These systems, such as CVS, Subversion, and Perforce, have a single server that contains all the versioned files, and a number of clients that check out files from that central place. For many years, this has been the standard for version control.

However, this setup also has some serious downsides. The most obvious is the single point of failure that

the centralized server represents. If that server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they're working on. If the hard disk the central database is on becomes corrupted, and proper backups haven't been kept, you lose absolutely everything—the entire history of the project except whatever single snapshots people happen to have on their local machines. Local VCS systems suffer from this same problem—whenever you have the entire history of the project in a single place, you risk losing everything.

About version control



Local System

Distributed Version Control

Distributed Version Control Systems

This is where Distributed Version Control Systems (DVCSs) step in. In a DVCS (such as Git, Mercurial, Bazaar or Darcs), clients don't just check out the latest snapshot of the files: they fully mirror the repository. Thus if any server dies, and these systems were collaborating via it, any of the client repositories can be copied back up to the server to restore it.

Furthermore, many of these systems deal pretty well with having several remote repositories they can work with, so you can collaborate with different groups of people in different ways simultaneously within the same project. This allows you to set up several types of workflows that aren't possible in centralized systems, such as hierarchical workflows.

It's **important to note** that version control systems don't *store multiple copies of your files,* rather, they store simple text based files that record the *history of changes* to your files. This saves a lot of space and makes you file organisation very much cleaner.

Why Git?

- Free and open source

- Many features
- Simple (for the basics)
- Small footprint (single directory)
- A suite of interrelated tools
- Offline version control
- Platform agnostic
- Branching based workflow

Getting **started**

- **Git** can be downloaded from http://git-scm.com/
- It's easy to install, only binaries are required
- Put Git on your PATH (or equivalent)
- Run from Command, Shell, BASH, etc.

There are many ways to interact with Git and your chosen online repository:

(1) Git and Github for Windows and/or Mac

(2) Git Preview

(3) Git via Sublime or many other plugins…

Setting up

- Test current version of git [`git --version`]
- Configure your Git [`git config`]

```
$ git config --global -l
```

- Set your name, email address, and editor:

```
$ git config --global user.name 'Your Name'
$ git config --global user.email 'you@web.com'
$ git config --global core.editor 'notepad'
```

The basics

- Initialize a repository [`git init`]
- Stage changes [`git add`]
- View changes [`git status`] or [`git diff`]
- Commit changes [`git commit`]
- Add multiple files with wildcards
  `git add '*.txt'`
- Commit with an inline message
  `git commit -m 'inline message here'`

As an example,

- Initialize a new repository (create a folder for example, and in that directory, type **git init**
- Take a look at the new *hidden folder* in your directory, it's the .git folder that tracks your changes!
- Now create some basic text or code file in that directory (use some R code for example)
- You can type **git status** to see the changes you have made

- Now you can add the new file you created, e.g. **git add file.txt**
- Try **git status** again to see that the changes have now been '*staged*' which means they are not in the repository yet, but have been earmarked as such

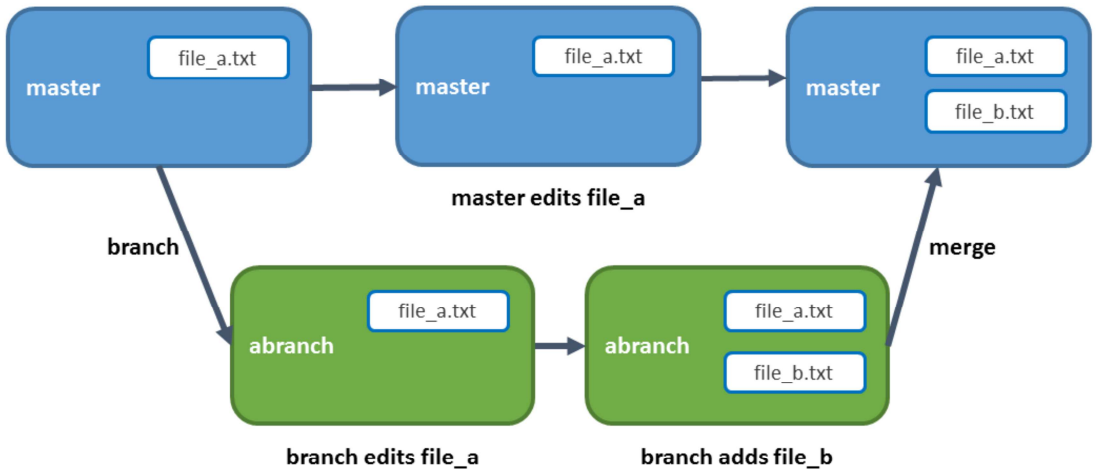
- To store the staged changes, run the command **git commit** which will commit all changes that have been staged (try **git commit –m "commit message"** as a shortcut to entering a message)
- And that's it! Your files have now been added to the repository

- You can use wildcards to add multiple files. For example, if you have many text files you want to **add** then try **git add '*.txt***

- You might also like to investigate the use of **git reset** which can help you to *unstage* a file

Further reading: http://git-scm.com/book/en/Git-Basics-Undoing-Things

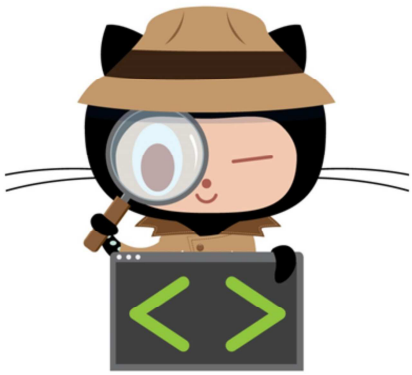
Git branching



Take a look at the .git folder in your directory: notice how there is still only one simple set of files.

git mantra: **branch early, branch often!**

Branching, merging, tracking



- View the change log [`git log`]
  - * each commit has a unique number (**sha**)
- Branch a repo [`git branch`]
- Checkout the branch [`git checkout`]
- Make changes, then `add` and `commit`
- Merge the changes [`git merge`]
- Try [`git tag`] to mark a special point

As an example,

- Branch a new repository, from the current directory, enter **git branch *newbranchname***
- Now switch to that branch by running the command **git checkout *newbranchname***
- Running the **git branch** command at any time will show you all possible branches and identify the *current branch*
- Now make some changes in that directory (note the new branch). You could change existing files, or add some new ones
- As always, you can type **git status** to see the changes you have made, note that git will tell you which branch you are on

- Run **git merge** from your master branch (the original)
- The **git merge** command will determine the best way to turn multiple *unique* changes into a single snapshot. However changes to the same piece of a file, in multiple changes, could cause a *merge conflict.* There are neat tools to figure those out, but git will provide you with simple markup in your file, so you can resolve the conflict yourself. Once resolved, simple run **git add filename** to let git know that the current file is the one you wish to keep.

- You can mark a special point in your code with a **git tag** command. This is a good way

to mark a major milestone in your project, and is like naming a branch. You can checkout the point where you set a tag by running the command **git checkout tagname**

- **Bonus:** Try out the useful **git stash** command if you come across the need to move to another branch, or fix a bug, before committing your current changes

Exercise: Add, Commit, Branch, Merge

- Try this typical workflow example:
  1. Make some changes to your files
  2. `add` changes
  3. `commit` changes
  4. `branch` repository
  5. `checkout` new branch
     - make changes to new branch files
  6. view the changes to your files: `diff`
  7. `merge` branch to master

Bonus: You can return your current branch to a previous state at anytime by running the command **git checkout [**commit number**]**

Getting started

- Create a Github account at https://github.com
  - Create a new **repository** (leave it blank)
  - It's a good idea to give your repository and project the same name
- Add a remote [git remote]
  - `$ git remote add origin https://github.com/name/project.git`
- Push a repository [git push]

  `$ git push -u origin master`

name = github account name

project = name of your project and/or repository

Example: Go to https://github.com and create an account, and then a new repository. Name it after your current locally based project, and leave it blank (no readme file).

Run the command **git remote add origin '**https://Github.com/name/project.git**'** to create a new 'remote' reference to an online repository called 'origin'

Now run the command **git push –u origin master** which will set the 'upstream' place for your push commands to the remote name origin

If you want a challenge, make the new repository with a readme.md file, and then try to push to the remote from Git. You might see a merge conflict because of the different files (depending which was created first). In that case, you can run **git pull** to pull the files down from the repo before running **git push** to push the local version to the remote. You might need to specify which remote and which branch you're referring to, so here, that would be **git pull origin master**

Below are the instructions that Github provides when you create a new ***Github*** repository. You can exercise two different options here: a) create a new ***local*** repository on your computer by the same name, and set the relevant remote or, b) push an existing repository with local files and version history stored on your computer to the new ***Github repository.***

A: Create a new repository on the command line
navigate to your main working folder, or create a new one

```
touch README.md
git init
```

B: Push an existing repository from the command line

```
navigate to existing .git repository
git remote add origin 'url'
git push -u origin master
```

```
git add README.md
git commit -m “first commit message"
git remote add origin ‘url’
git push -u origin master
```

Exercise: Local changes, push to Github

- Try this typical **Github** workflow:

1. make changes to a repository
2. add changes
3. commit changes
4. push changes
5. make new branch
6. make changes in branch
7. push new branch
8. merge changes locally
9. push merged changes

- **Hint**: Work on the README file to see changes via Github on the main page of your repository

Bonus: view your changes and workflow on the network diagram
https://github.com/name/project/network

Collaborating via **Github**

- Clone a repository [`git clone`]
- Fork a repository [`github equivalent`]
- Work as **collaborators** or **fork**
- Having made local changes, you can `push` (with permission), or make a `pull request` to the repository owner

You can mark a special point in your code with a **git tag** command. This is a good way to mark a major milestone in your project on the Github repository and it can act like a **release** for ongoing software development.

Exercise: Fork, Add, Commit, Pull Request

- Try this typical **collaborative** workflow example:

1. `fork` a repository
2. `clone` to local machine
3. make changes
4. `add` changes
5. `commit` changes
6. `push` to forked repository
7. `submit` pull request

Exercise: We can all work on a piece of code, or together submit a function to a joint repository. Try a **git clone** of the demo project, or a **fork** via github.com, and then make some local changes. Once you're done, try to **push** those changes, and see what happens…

Bonus: view the combined changes and workflow on the network diagram https://github.com/name/project/network

For a great example of a collaborative workflow see: https://github.com/smartell/iSCAM/network

More information

- http://git-scm.com/about for introduction to **Git**
- http://git-scm.com/book for the nitty gritty
- http://gitref.org for a basic how-to guide
- https://github.com/ for overview of **Github**
- https://try.github.io for an interactive trial

Also see http://rogerdudler.github.io/git-guide/ for some easy to use info, including some great tips on **git diff**

And for a different point of view see http://steveko.wordpress.com/2012/02/24/10-things-i-hate-about-git/