Lecture Agenda and Notes Draft:

Lecture Agenda and Notes Draft:
Basics of Version Control

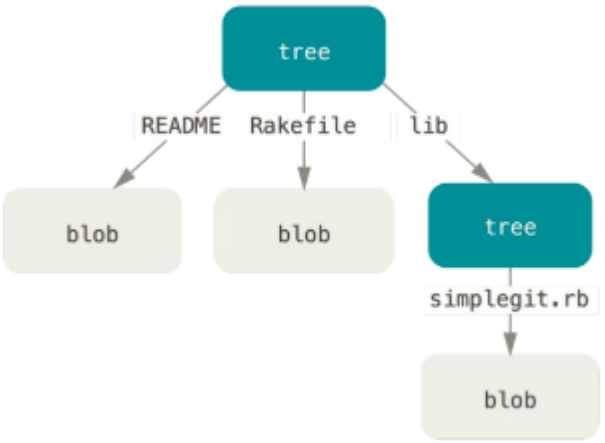
Intro: show https://youtu.be/o8NPllzkFhE?t=440 7:20 to 9:20

Version control fundamentals

1. Today we are going to talk about version control, and about Git in particular. *Ask the audience if they have tried the Git before.*
2. Version control tools help to keep track of changes applied to files and folders and in addition to that they facilitate collaboration. So, it's really useful when work with group of people
3. Version control tracks the changes of files and folders with a series of snapshots that capture the entire state of the folder and everything inside. Along with actual changes version control system keeps metadata that says
    a. "who is author of the particular change",
    b. "when the change was made",
    c. "which message author left with change".
4. Why is it useful? Even if you work on the project by yourself it's very helpful to look at the old versions of your code, that you've written, figure out why something was changed (with commit messages), implement features in parallel (in different branches), e.g. working on bug fixes and on the new features independent.
5. When you work in a team it helps to facilitate sharing the code and resolve conflicts in the files that occured when multiple people work with the same file simultaneously. Track the changes "who change that", "when it was done" of entire codebase

Git data model

1. Git models the history of a collection of files and folders within some top level directory as a series of snapshots. In git terminology:
    a. the file is called a **blob** and it is just a bunch of byters
    b. the directory is called a **tree**, it maps names to blobs or trees within it
    c. the **snapshot** is a top level tree that being tracked

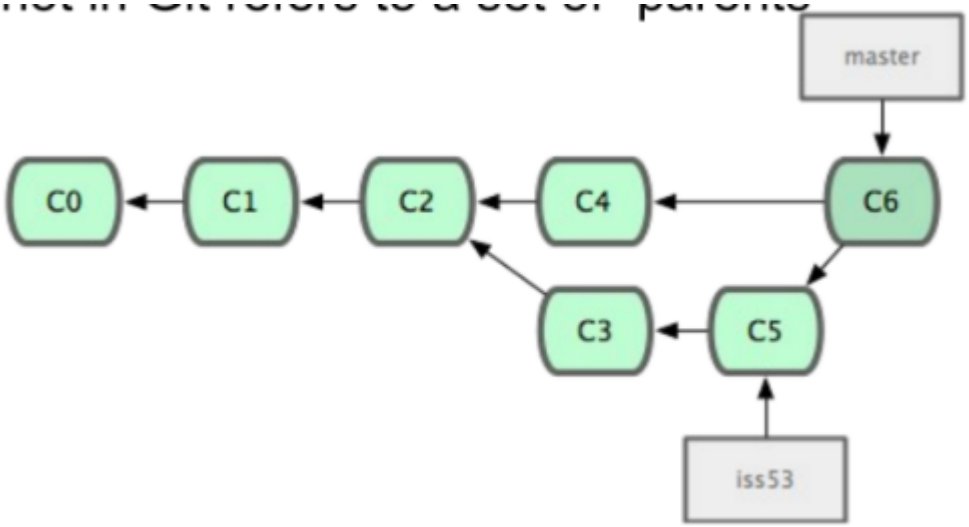


2. Examples of a top level tree (*listing/image)*

```
<root> (tree)
|
+- foo (tree)
|  |
|  + bar.txt (blob, contents = "hello world")
|
+- baz.txt (blob, contents = "git is wonderful")
```

3. How should a version control system relate snapshots? A simple model could have a linear history, a list of snapshots in time order. For many reasons Git doesn't use a simple linear model like this. In Git a history is represented by a directed acyclic graph where each snapshot refers to a set of snapshots that preceded it. Each snapshot may be descended from multiple parents.
4. Example of a history *(image)*.
    a. Each C on the slide corresponds to a snapshot (commit), and contains a complete state of the project in a particular moment.
    b. Each commit points to the parent commits
    c. After the third commit C2, the history branched and the development continued in parallel
    d. C6 has two parents. In this point of history a **merge** has happened, when two branches of development have joined together

e. ***Note***: commits in Git history are immutable, BUT mistakes can be corrected



5. Three main data types of a Git data model in pseudocode
    a. Blob is just an array of bytes (represents files)
    b. Tree a mapping from a string name to another tree or a blob (represents directory)
    c. Commit a structure with: set of parents, name of the author, commit message, snapshot (project state)

```
// a file is a bunch of bytes
type blob = array<byte>

// a directory contains named files and directories
type tree = map<string, tree | blob>

// a commit has parents, metadata, and the top-level tree
type commit = struct {
    parent: array<commit>
    author: string
    message: string
    snapshot: tree
}
```

Git Repository

1. Git stores everything in the “data store”. Each representative of any type (blob, tree, commit) may be treated as an “object” that can be stored and loaded from a “data store”. Objects are stored by its hash in a map (you see this very hashes when see “commit f70ccaeb2” in github)
2. Pseudocode of a “data store”

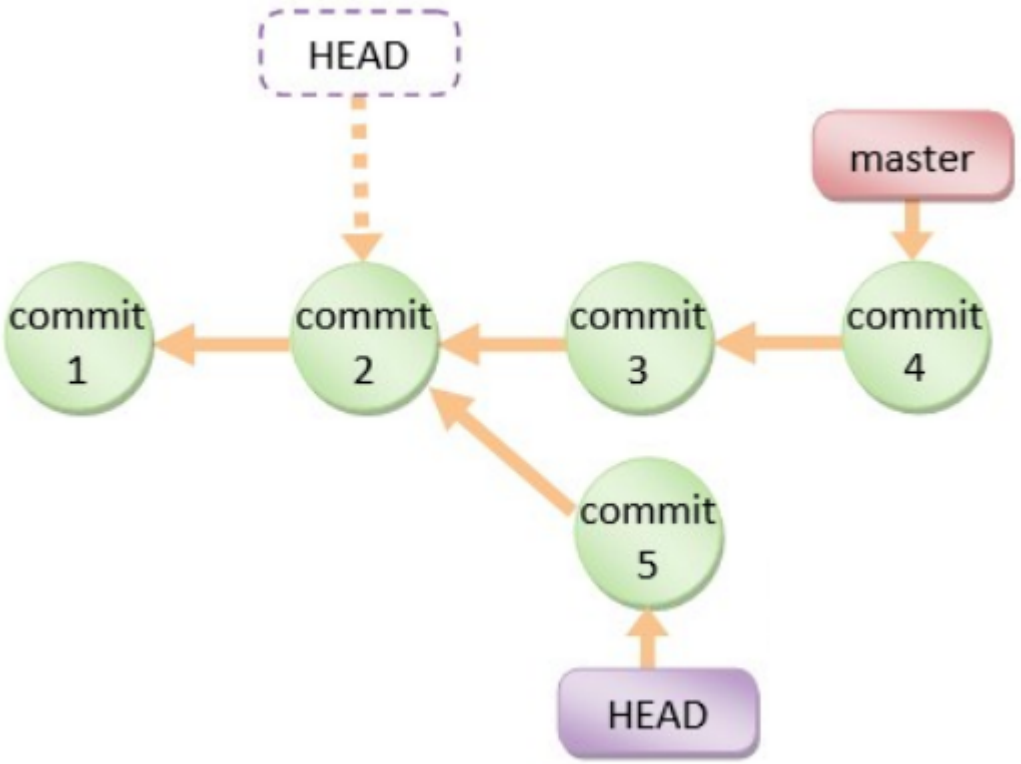
```
// a file is a bunch of bytes
type blob = array<byte>

// a directory contains named files and directories
type tree = map<string, tree | blob>

// a commit has parents, metadata, and the top-level tree
type commit = struct {
    parent: array<commit>
    author: string
    message: string
    snapshot: tree
}
```

3. To view git docs, type `man git-<command>` e.g. `man git-checkout`
4. Prepare example-01:
    a. Create a directory `example-01` and `git init` there
    b. `git checkout -b example-01`
    c. Create `./foo/bar.txt` and `./baz.txt` with some text in both
    d. `git add` and `git commit`
5. Using example-01 branch show the following
    d. `git ls-tree HEAD` --- to output tree structure with hashes (use the hash in the second step)
    e. `git cat-file -p 10e1b4d8c` --- to see what’s inside the file by hash
4. Humans are not good at remembering 40 digits hexadecimal strings, that’s why Git provides us a convenient way to refer to different objects. It gives us **references**, they are pointers to commits, they can be updated to point to a new commit.

5. For example **master** pointer points to a latest commit in a main branch of the development, **HEAD** points to a current state of the project we are working with.



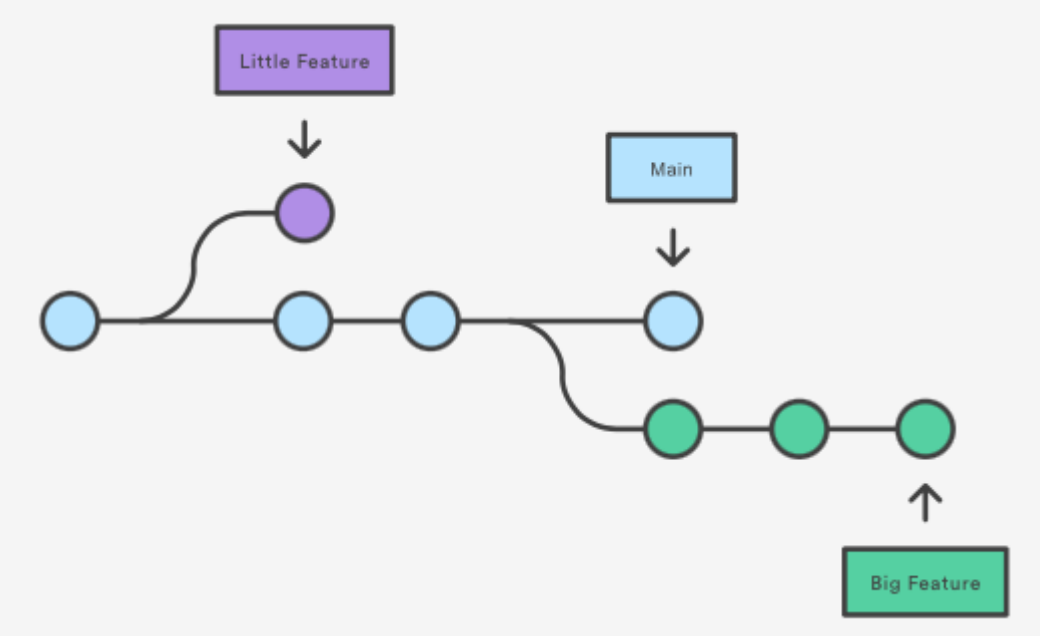
6. Finally, what is a Git repository? It is a **data store** and **references**. Always think about manipulations with a project in terms “how the storage and references should be modified” in order to make right moves

Staging area. *git add*, *git commit*, *git checkout*

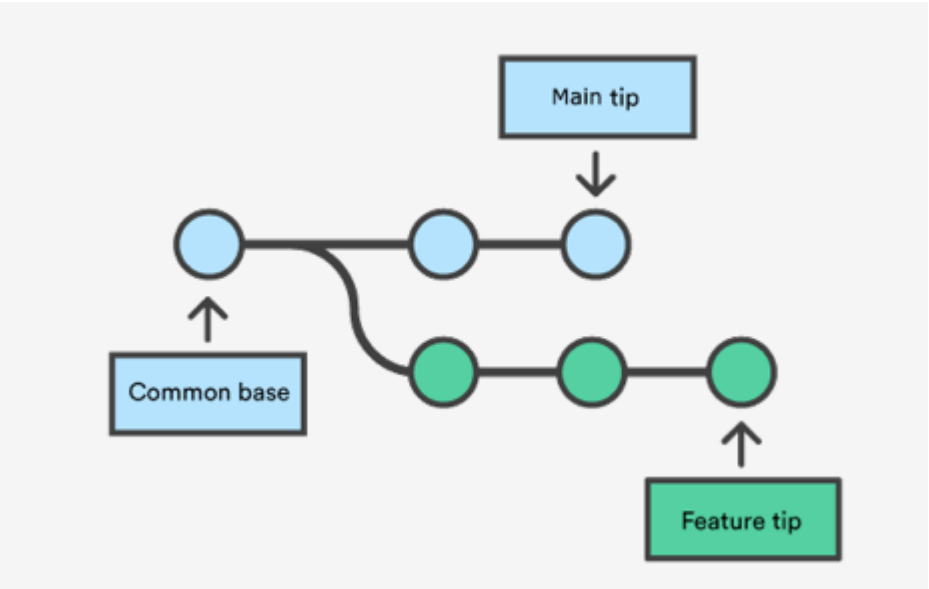
1. Changes in the project won't go to a snapshot automatically. Imagine for example you implemented a feature and some debugging statements in the one file. Obviously, it is not necessary to make a full snapshot of your code but the important changes. For this purpose the staging area exists
2. With example-01 branch show how the staging works:
    a. show the graph using *git log --all --graph --decorate*
    b. tell about graph you see, about references
    c. tell about aliases in Bash: *alias **glog**="git log --all --graph --decorate"*
3. Continue experimenting with created baz.txt and foo/bar.txt
    a. add text "hello world!" foo/bar.txt
    b. using *git diff foo/bar.txt* look at the changes you made relative to the staging area
    c. *git add foo/bar.txt*, *git commit -m "update bar"*
    d. run **glog**, pick the <commit_hash> of the previous commit
    e. use *git diff <commit_hash> foo/bar.txt* to see changes between snapshots
4. Checking out to previous snapshots
    a. use *git checkout <commit_hash>* to go to previous commit
    b. read the message git provides
    c. use **glog** to see where is the HEAD now
5. Other useful stuff
    a. *git help <command>* to see help message
    b. *git init* to create a git repo based on folder where you are
    c. *git status* tells what's going on with project

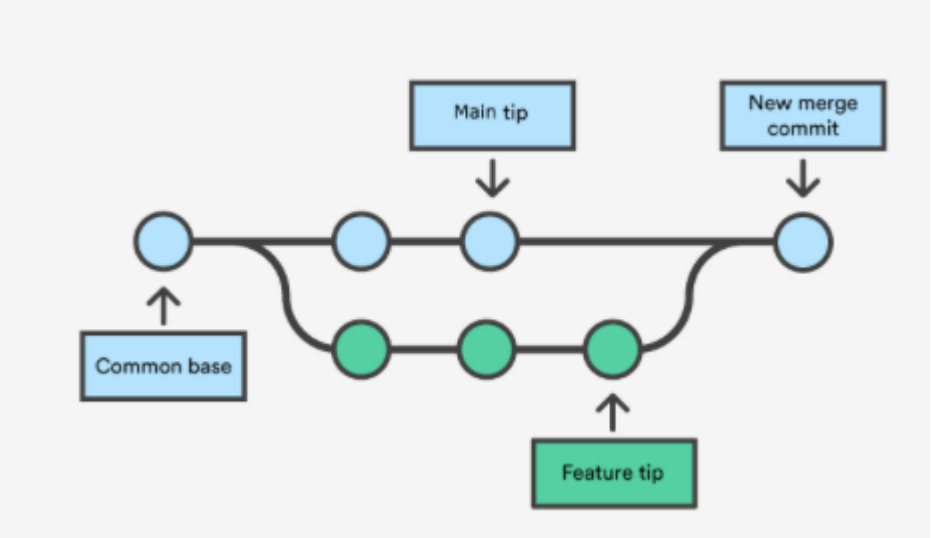
Branching, merging and rebasing

1. As we said before history of change is not linear and might branch in some point of time. One can do set of things with branches: create them, switch between them, merge them and rebase
2. Let's start with branch command



   a. In the current directory run *git branch* to see all available branches
   b. run *git branch lateral-branch* to create a new branch from here
   c. run *git checkout lateral-branch* to switch (it is possible to do latter two by running *git checkout -b lateral-branch*)
   d. add the line "git's the best" to baz.txt file, add and commit changes
   e. run ***glog*** to see a new graph, tell about references and branches
3. So now we have a separate branch lateral-branch that contains changes not present in the master. What if we update the master and then try to merge changes from both branches?

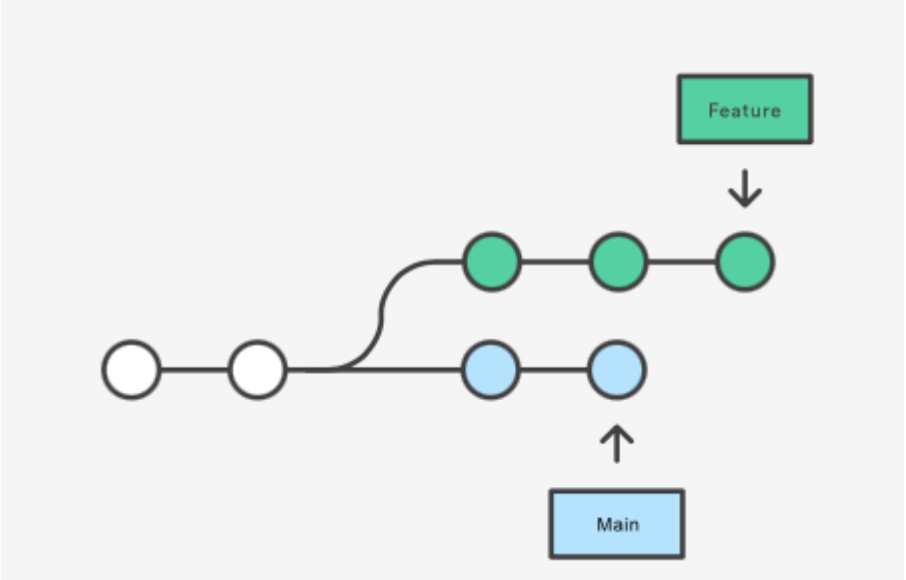


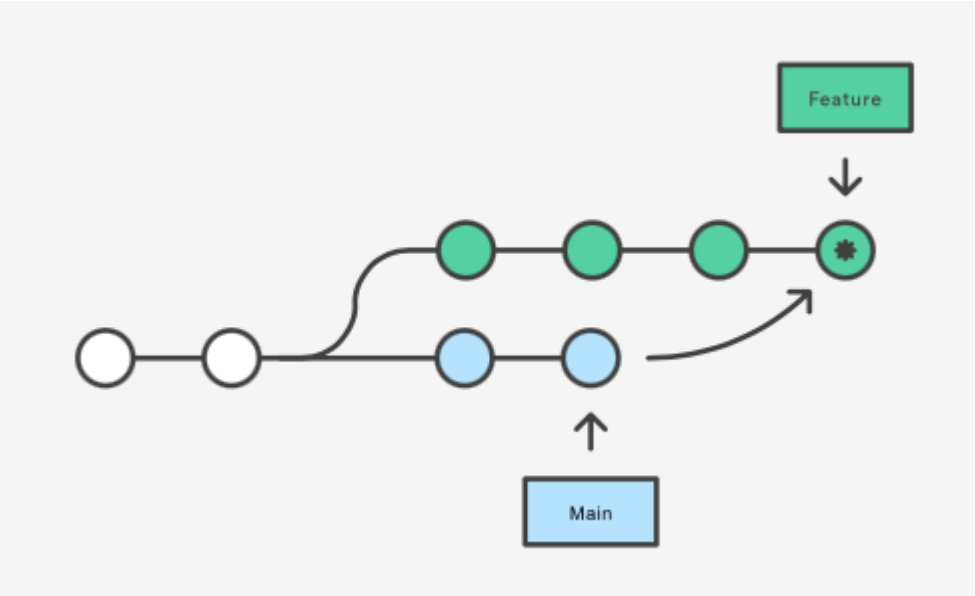


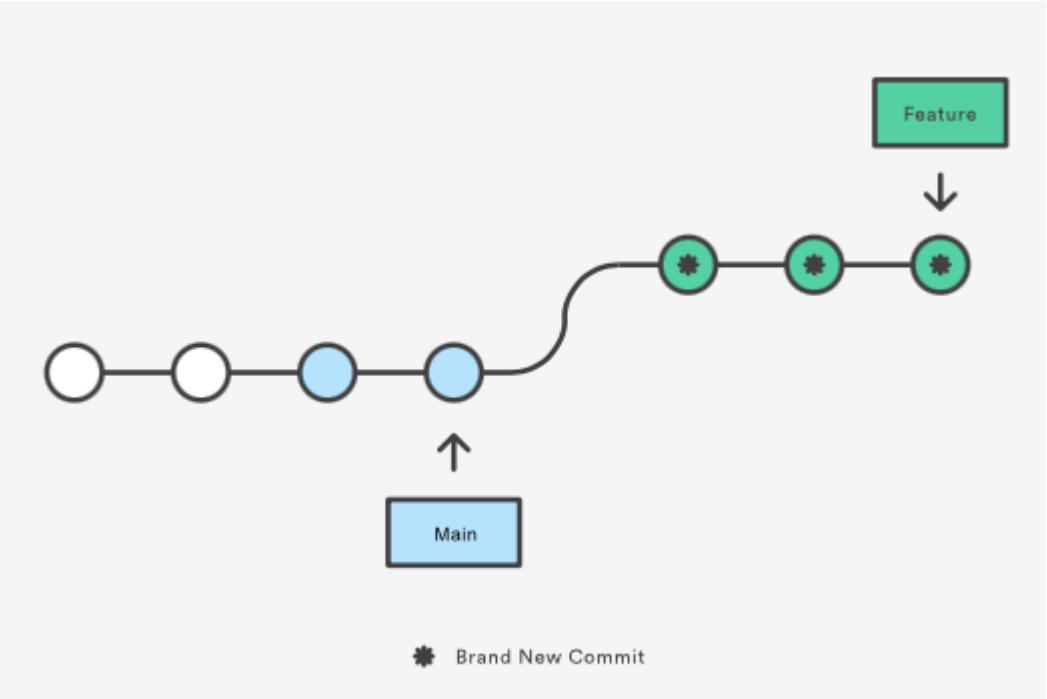
a. do checkout to the master branch
b. add the line “even with the conflicts” to the baz.txt
c. *git add, git commit*
d. search for a commit hash of the lateral-branch in the glog
e. while on master branch run *git merge <commit-hash>*
f. conflict occurred, show it with *vim baz.txt*
g. run *git merge --abort* to undo the latest merge try OR
h. or resolve conflict (hopefully not in vim but it’s easier with *git mergetool --tool=vimdiff3*) and run *git merge --continue*

Git rebase and graphical interface tools; pushing and pulling

1. One very helpful thing while avoiding merge conflicts is to use rebasing. When you introduce some changes in a branch and master has gone forward, you can change the root of your branch to a current master!







2. It is possible to do with git rebase see the tutorial on github or use of the GUI such as
    a. GitKraken
    b. SmartGit
    c. GitHub Desktop
    d. Sourcetree and others

    This tools give you a convenient way to do rebasing, committing, pushing and pulling
3. When you work with your repo locally it comes the time to publish your work to the internet. To do so you can use *git push*
4. To get most recent result from the remote you can do
    a. *git fetch* that will load the updates but keep you on the same HEAD
    b. or *git pull* that introduce all remote changes in a branch you're right now
5. One can put a .gitignore file inside a project directory. It's used to tell which directories and files must be excluded from snapshots. General rules are:
    a. `__pycache__/` --- used to exclude directory anywhere inside project
    b. `logs/`
    `!logs/important.log` --- to exclude all logs directories but keep which path contains `logs/important.log`
    c. `*.log` --- to exclude any `.log` files from tracking

Reading

- https://www.atlassian.com/git/tutorials/learn-git-with-bitbucket-cloud