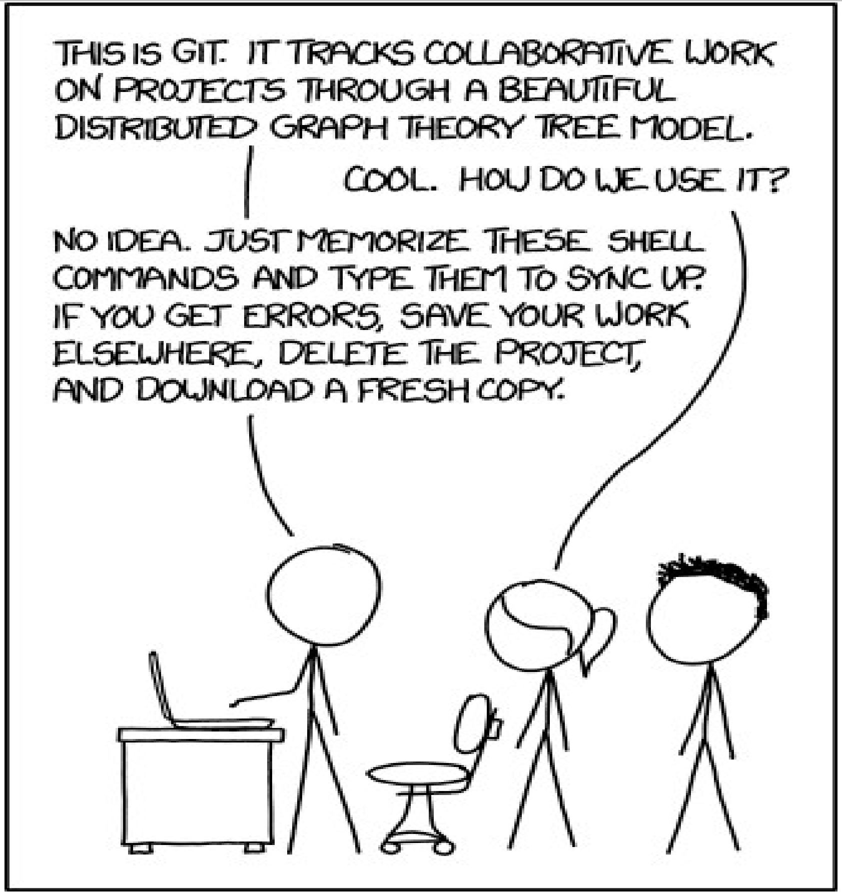
LECTURE: VERSION CONTROL (GIT)

WHAT IS VERSION CONTROL SYSTEMS

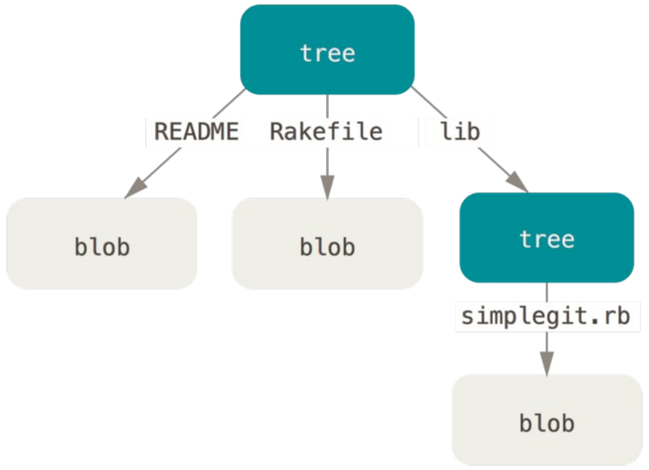

THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

Skoltech
Skolkovo Institute of Science and Technology

GIT’S DATA MODEL

SNAPSHOTS

- **In Git's data model**
- A file is called a "blob", and it's just a bunch of bytes
- A directory is called a "tree"



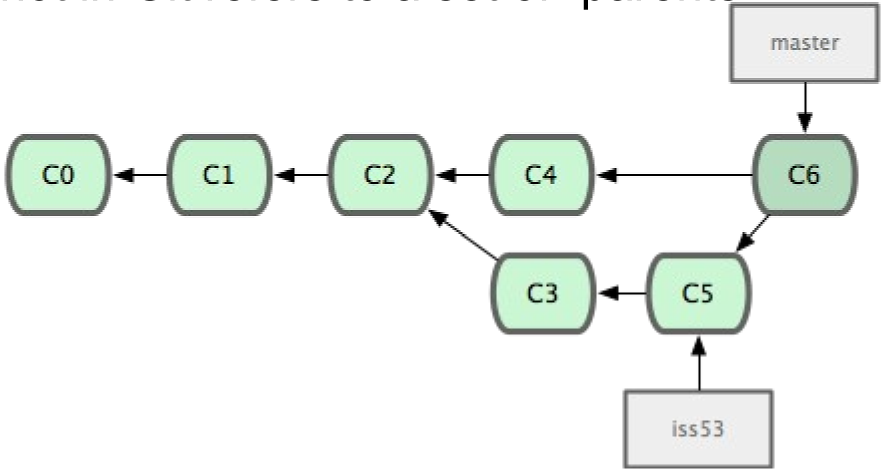
Directory's tree structure

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

The top-level tree contains two elements, a tree "foo" (that itself contains one element, a blob "bar.txt"), and a blob "baz.txt".

MODELING HISTORY: RELATING SNAPSHOTS

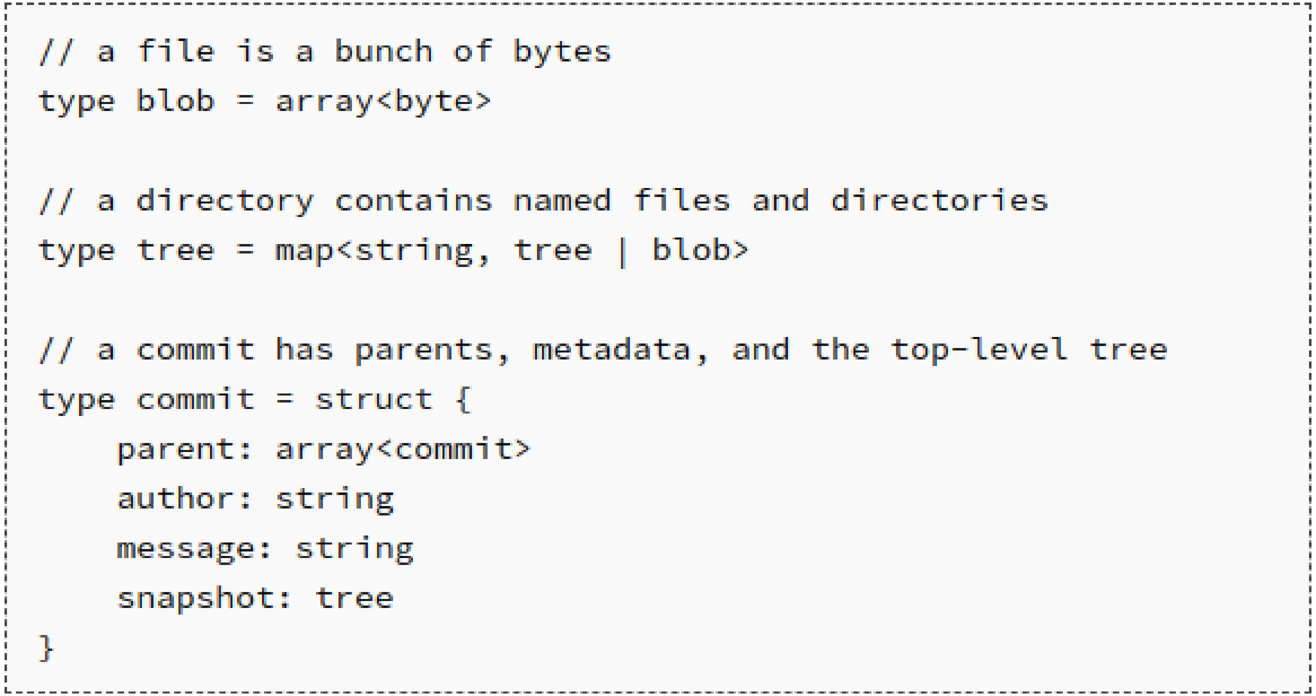
A history is a directed acyclic graph of snapshots. This means is that each snapshot in Git refers to a set of “parents”



Remember: commits in Git are immutable, BUT mistakes can be corrected!

DATA MODEL AS PSEUDOCODE

We now know that Git data model contains of three main types of objects



```
// 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
}
```

DATA MODEL AS PSEUDOCODE

An “object” is a blob, tree, or commit:

```
type object = blob | tree | commit
```

In Git data store, all objects are content-addressed by their SHA-1 hash.

```
objects = map<string, object>

def store(object):
    id = sha1(object)
    objects[id] = object

def load(id):
    return objects[id]
```

CONTENT-ADDRESSING WITH EXAMPLE

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

For example, the tree for the example directory structure above

(visualized using `git ls-tree HEAD`)

```
100644 blob 4448adbf7ecd394f42ae135bbeed9676e894af85    baz.txt
040000 tree c68d233a33c5c06e0340e4c224f0afca87c8ce87    foo
```

With `git cat-file -p 4448adbf7ecd394f42ae135bbeed9676e894af85`, we get the following:

```
git is wonderful
```

Skoltech
Skolkovo Institute of Science and Technology

REFERENCES

Git's solution to this problem is human-readable names for SHA-1 hashes, called "references"

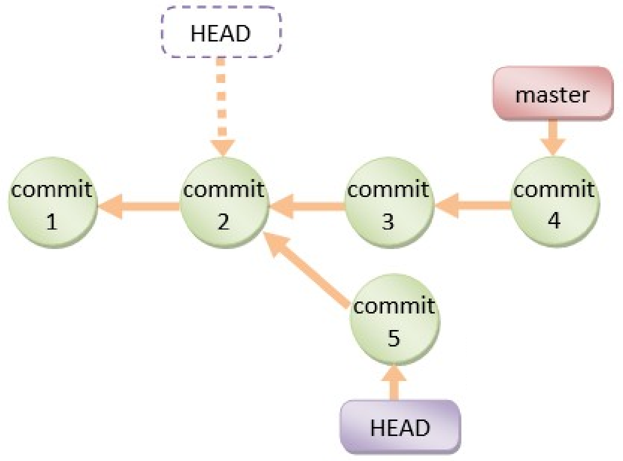
```
references = map<string, string>

def update_reference(name, id):
    references[name] = id

def read_reference(name):
    return references[name]

def load_reference(name_or_id):
    if name_or_id in references:
        return load(references[name_or_id])
    else:
        return load(name_or_id)
```

Changed HEAD and master references



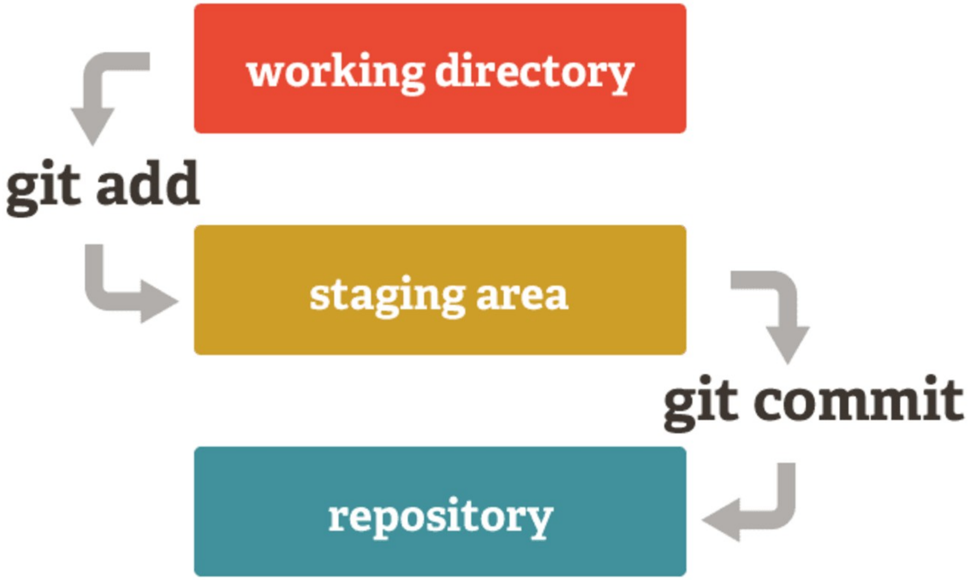
REPOSITORIES

Think about what manipulation the command is making to the underlying graph

e.g. *“discard uncommitted changes and make the ‘master’ ref point to commit 5d83f9e”*

```
git checkout master; git reset --hard 5d83f9e
```

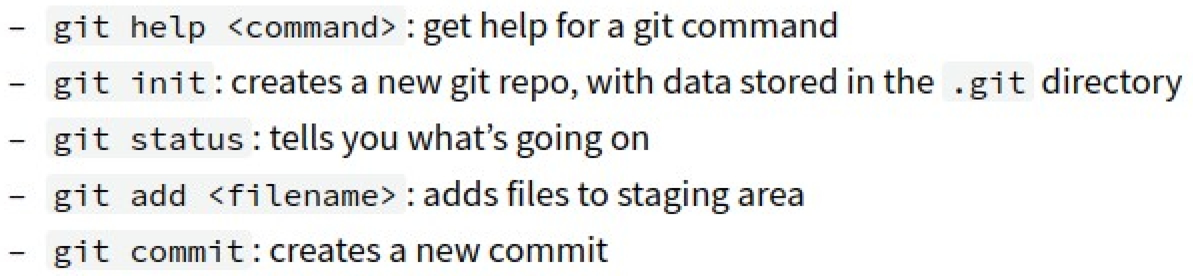
STAGING AREA



GIT COMMAND-LINE INTERFACE

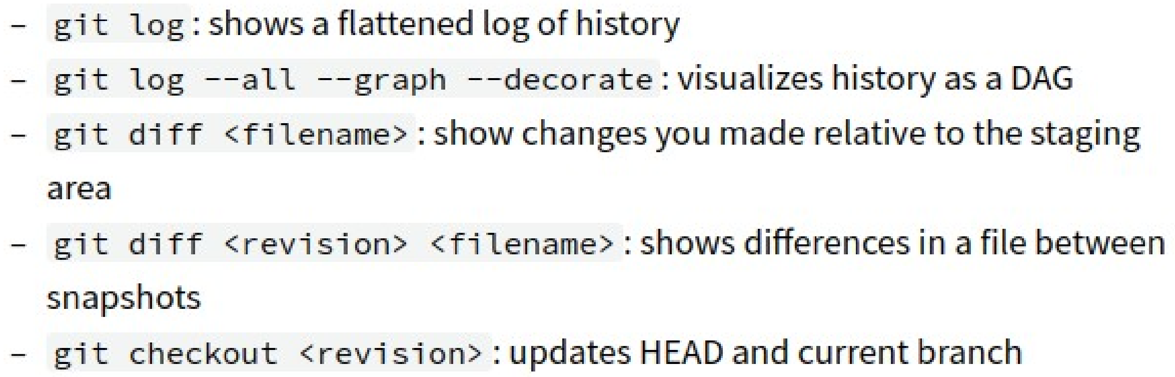
Skoltech
Skolkovo Institute of Science and Technology

BASICS



- `git help <command>`: get help for a git command
- `git init`: creates a new git repo, with data stored in the `.git` directory
- `git status`: tells you what's going on
- `git add <filename>`: adds files to staging area
- `git commit`: creates a new commit

On how to write good commit messages!



- `git log`: shows a flattened log of history
- `git log --all --graph --decorate`: visualizes history as a DAG
- `git diff <filename>`: show changes you made relative to the staging area
- `git diff <revision> <filename>`: shows differences in a file between snapshots
- `git checkout <revision>`: updates HEAD and current branch

https://git-scm.com/book/en/v2

REMOTES

- `git remote`: list remotes
- `git remote add <name> <url>`: add a remote
- `git push <remote> <local branch>:<remote branch>`: send objects to remote, and update remote reference
- `git branch --set-upstream-to=<remote>/<remote branch>`: set up correspondence between local and remote branch
- `git fetch`: retrieve objects/references from a remote
- `git pull`: same as `git fetch; git merge`
- `git clone`: download repository from remote

https://git-scm.com/book/en/v2

GIT GUIS

axosoft GitKraken

Great tool to start Git experience

SmartGit

Useful for large-ish repositories

GitHub Desktop

Available only on Mac & Windows, works good with GitHub