



DIS08 – Data Modeling

03 – Introduction to git and GitHub

Philipp Schaer, Mandy Neumann, Technische Hochschule Köln, Cologne, Germany

Version: SS2020

git != GitHub

git

- Version control system
- Created by Linus Torvalds (2005)
- Other VCS include CVS, SVN, Mercurial

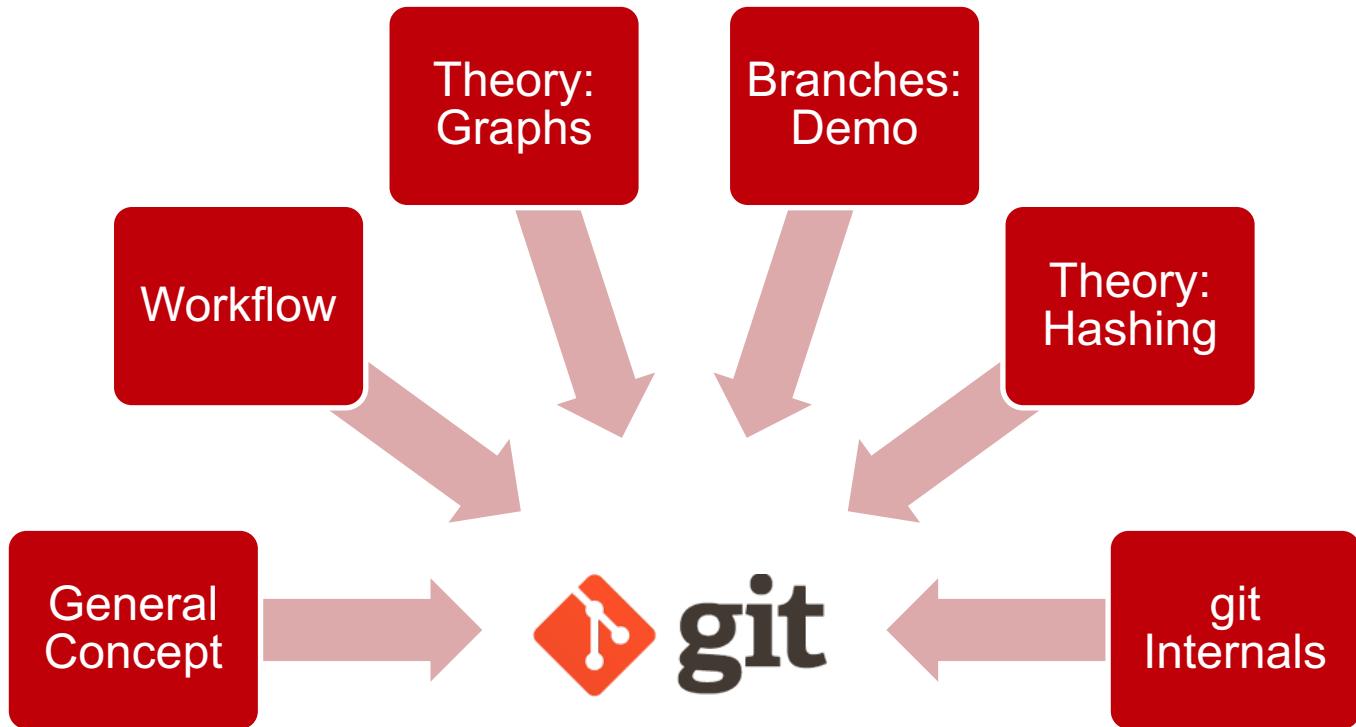


GitHub

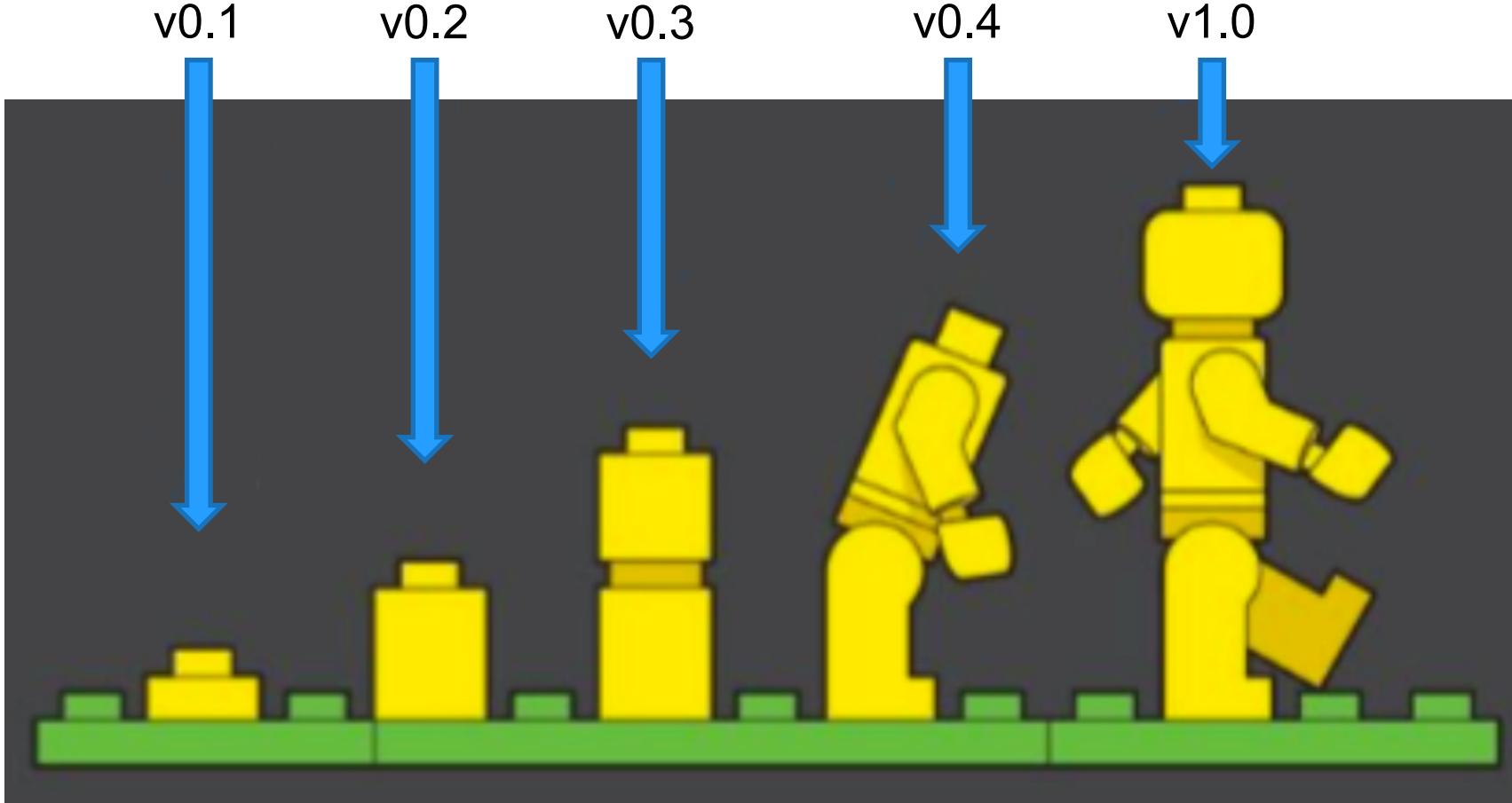
- Website for hosting projects that use git
- Launched in 2008
- Others include [BitBucket](#), [GitLab](#)

GitHub

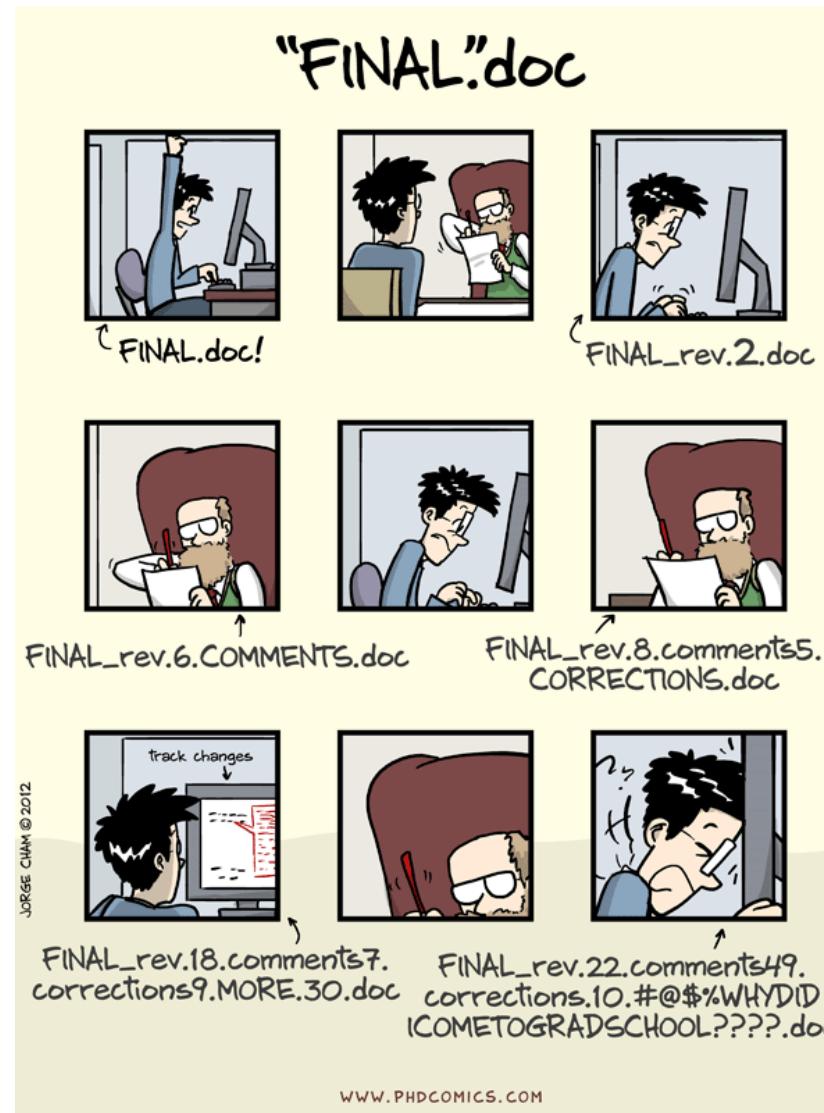
Outline



git is a *Version Control System* (VCS)



Files and Revisions – The Problem



Version Control – The Solution

Version Control Systems

- **Track progress** over time
- Save **snapshots** to your history to **retrace your steps**
- **What** changed, **who** changed it, and **why**

Order with coordination

- In a **centralized system**, you can objectively call versions a numerical progression:
version 1, version 2, version 3...
- Since John made a new version before Vanessa, his is n+1, and Vanessa is n+2.

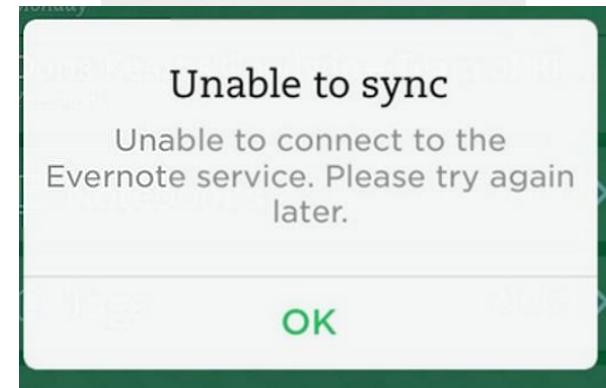
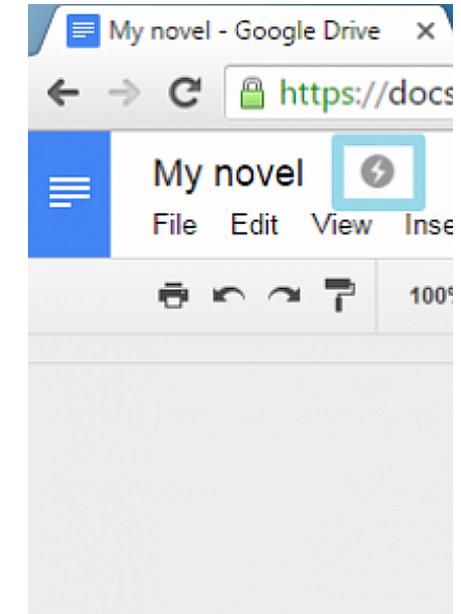
The screenshot shows a 'Version history' interface with the following data:

- Today**:
 - January 9, 11:32 AM (Current version)
Vanessa Gennarelli
- Sunday**:
 - January 7, 1:04 AM
John Britton
- Saturday**:
 - January 6, 11:47 PM
John Britton
- January 6, 10:27 PM**
John Britton

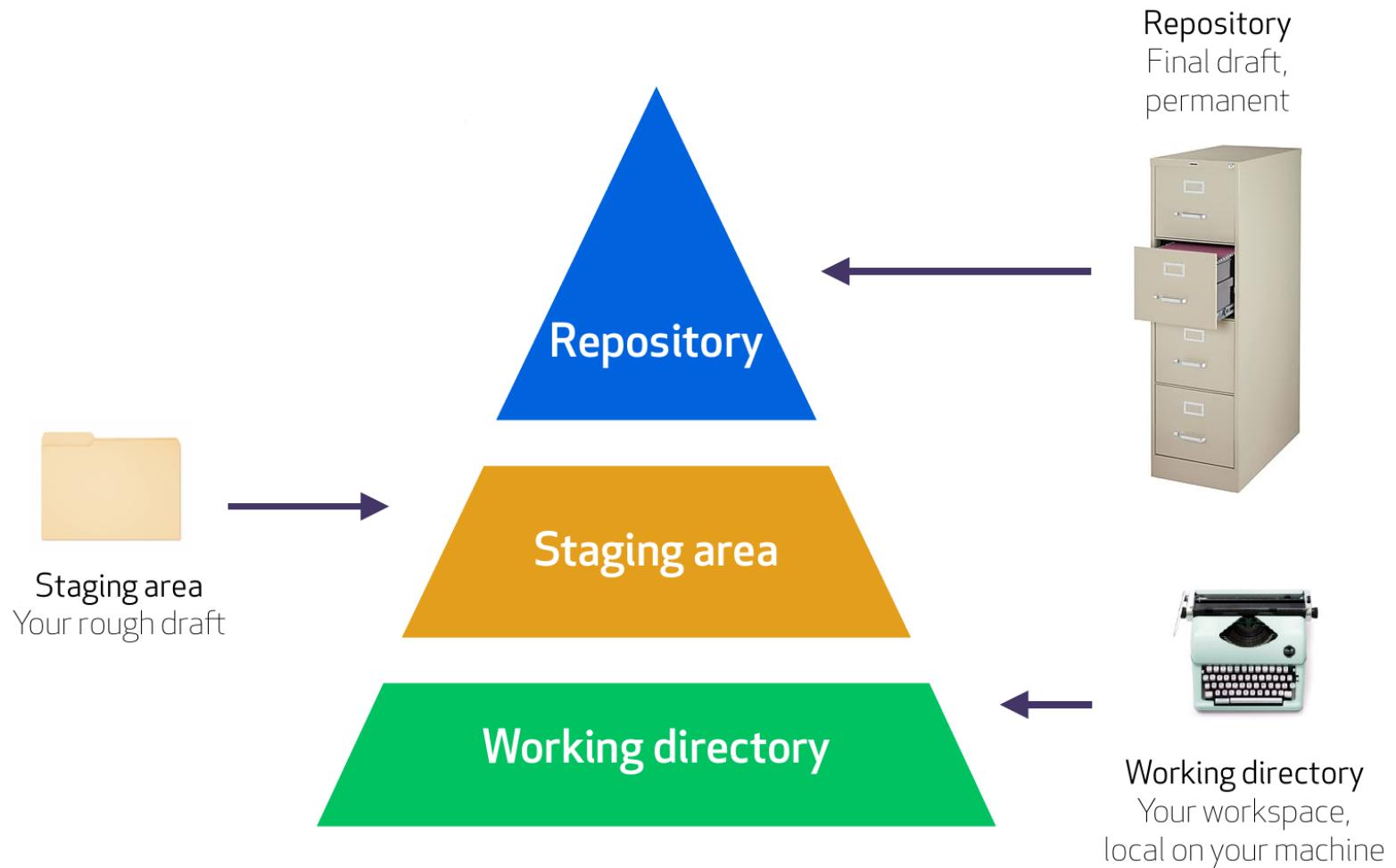
Centralized systems require coordination

In reality we are working in **parallel**

- This leads to **conflicts**
- So, we need **order without coordination**



git - General Concept



git - General Concept

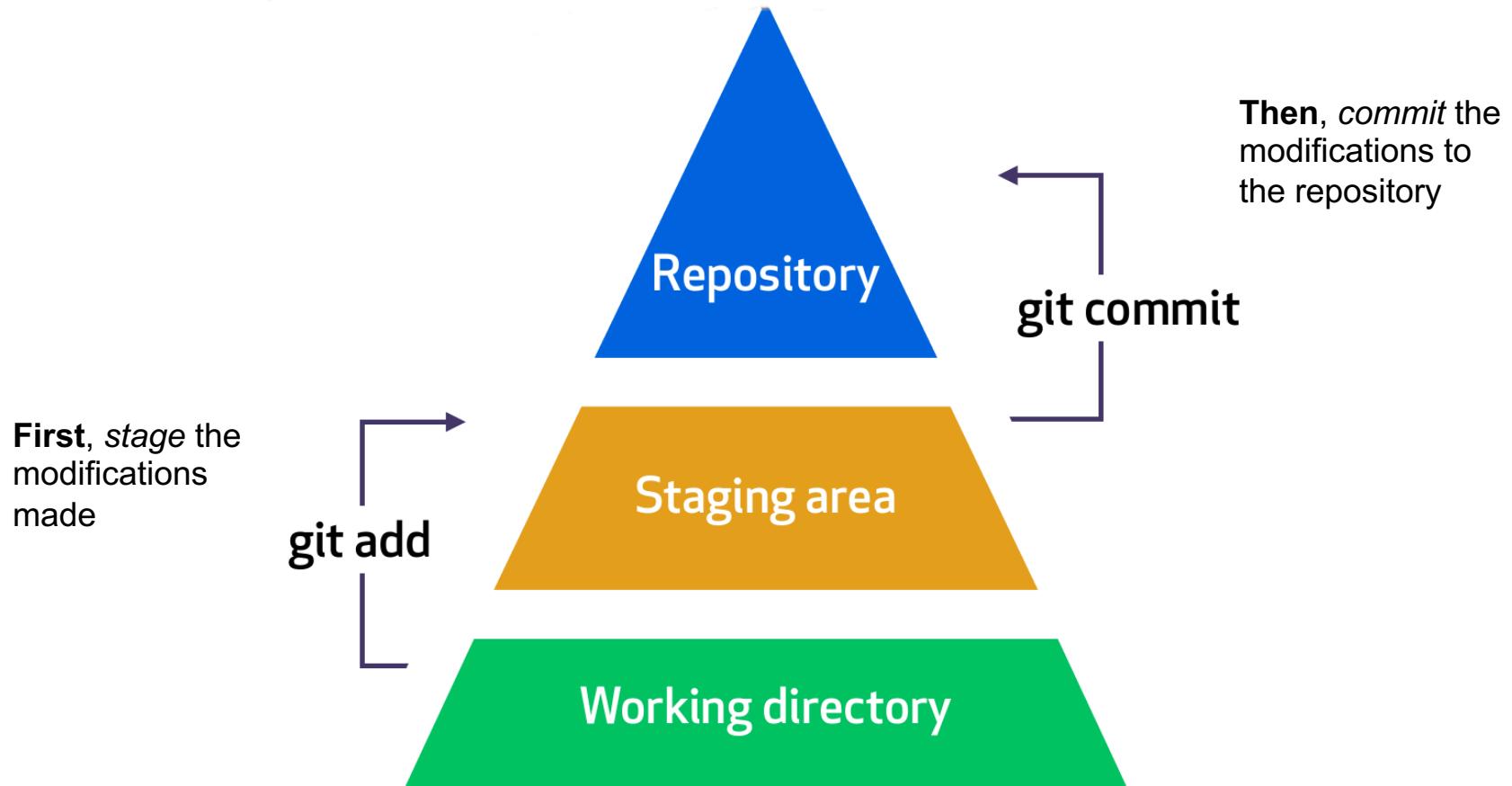
- All the internal „tracking data“ is stored inside of a **repository**.
- Logically, a repository holds all files belonging to one project.
- The repository lives in a special folder called `.git`

```
$ git init example
```

```
Initialized empty Git repository in / [...] /example/.git/
```

- The newly-created directory `example` is now the **working directory** for a project.
- You can also perform `git init` inside of an existing directory to put it under version control.

git Workflow



git add

```
$ git status
```

On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)

Ask git for the
status of this
repository

git gives
some hints
what can be
done next

```
$ echo 'Hello, World!' > hello.txt
```

```
$ git add hello.txt
```

```
$ git status
```

On branch master

No commits yet

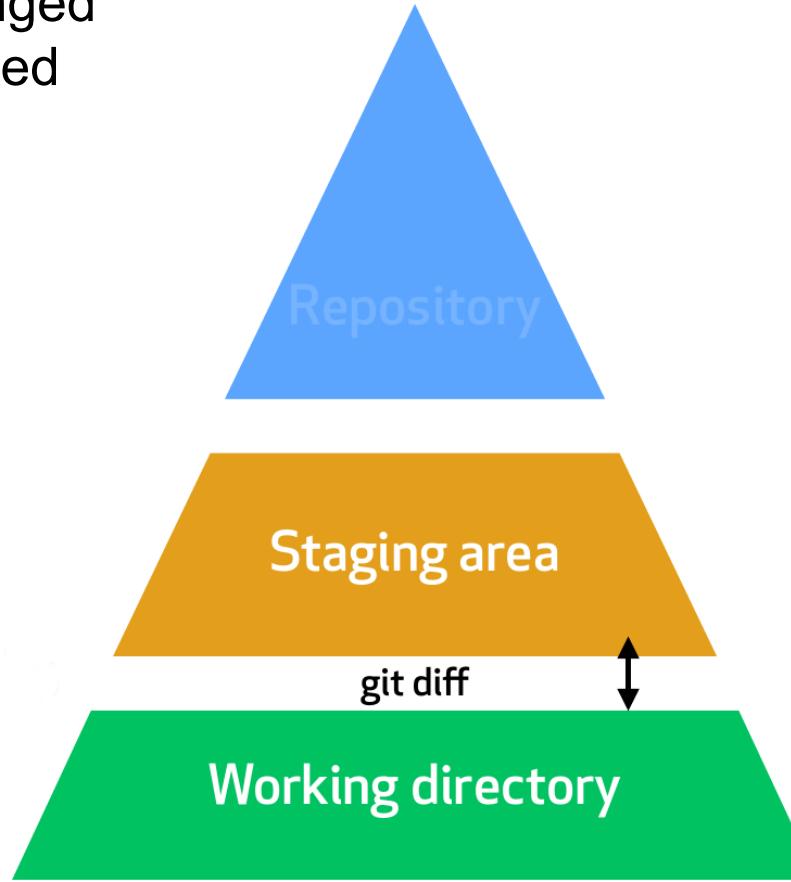
Changes to be committed:

(use "git rm --cached <file>..." to unstage)

new file: hello.txt

git diff

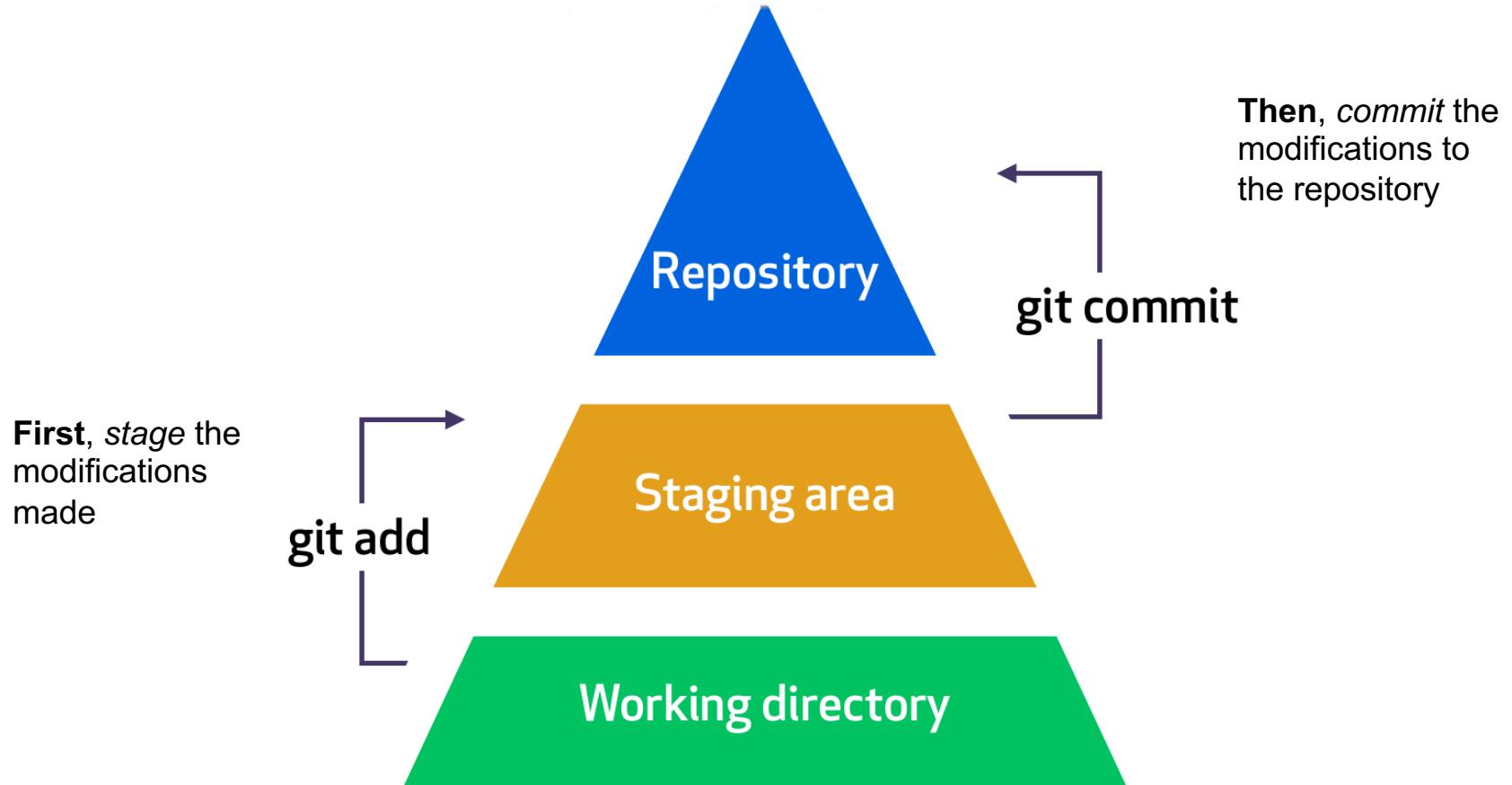
See what has changed
that is not yet tracked
with `git diff`.



Compares
staging area to
working directory.

There's no output
if they are the
same.

git Workflow



git commit

- After we have done some work, we want to create a snapshot of the project.

```
$ git status -s  
A README.md  
A hello.txt
```

Short version of status – one file per line with status code

- All modifications to the staging area since the last commit will be part of this snapshot. This includes new files, changed files and deleted files.

-m option: no external editor needed, provide message as argument

```
$ git commit or $ git commit -m 'Initial commit'
```

- The commit stores the **who**, **when** and **why** to a snapshot: author information, commit message, and current timestamp.

git commit

- Pro-Tip: Write **meaningful commit messages** to make **clear for everyone** (including future-you) **what** changes were introduced in this commit.
- The first line should be a short summary. Additional lines can be used to elaborate.
- Best practice: group commits logically, e.g. one commit for a new feature, another one for fixing typos.

git commit -m "changes"



Writing

Useless Git
Commit Messages

O RLY?

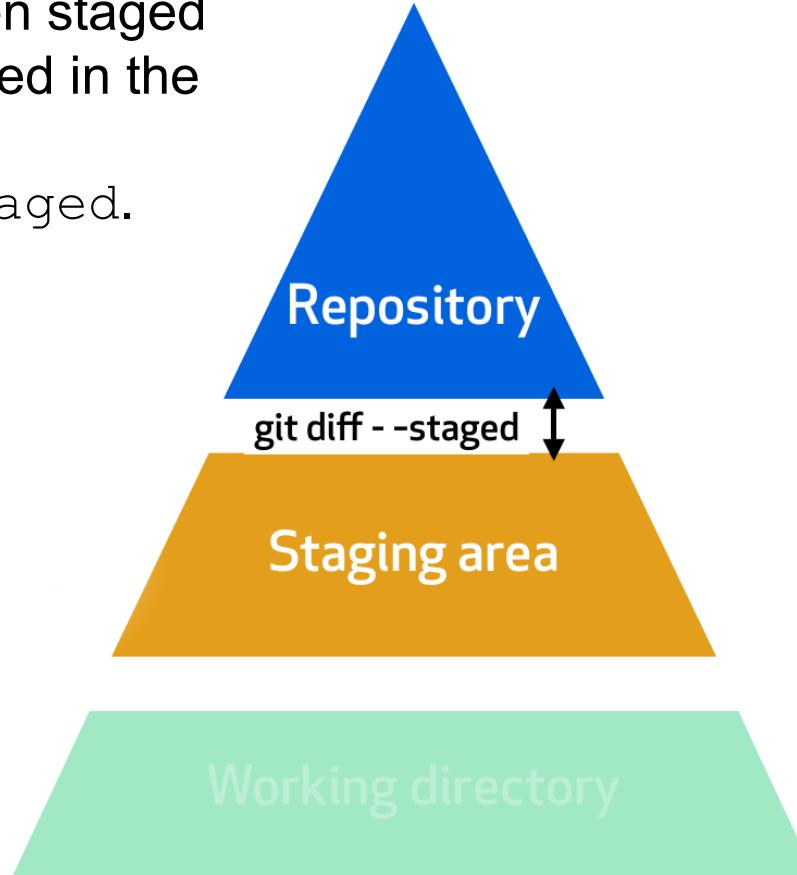
@ThePracticalDev

git diff --staged

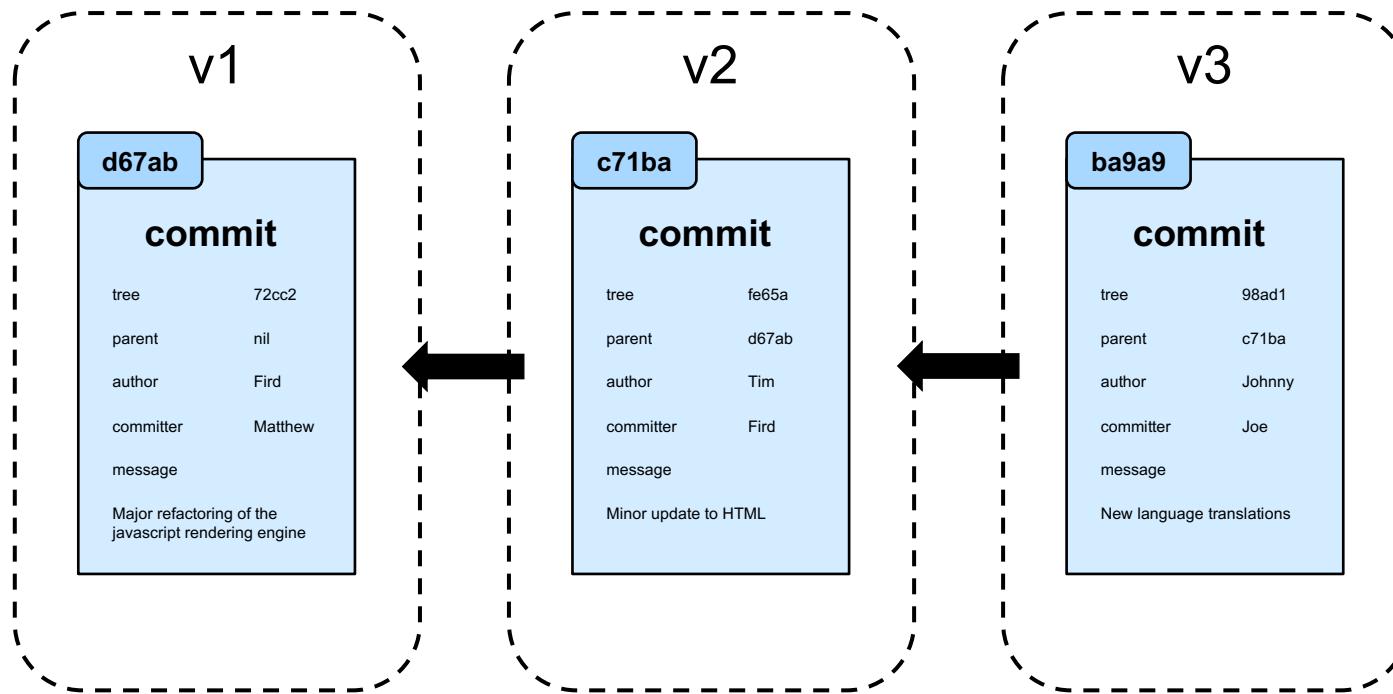
See what has been staged
that is not yet stored in the
repository with
`git diff --staged`.

Compares
staging area to
the repository.

There's no output
if they are the
same.

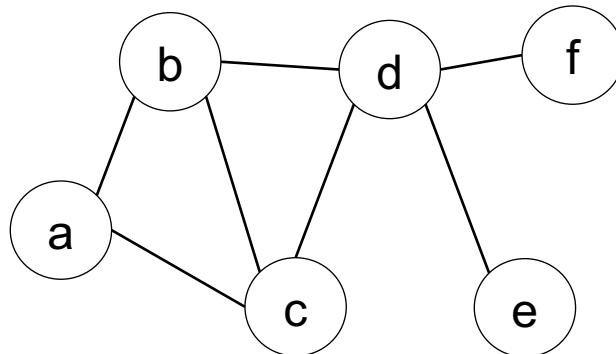


Several commits build up a history

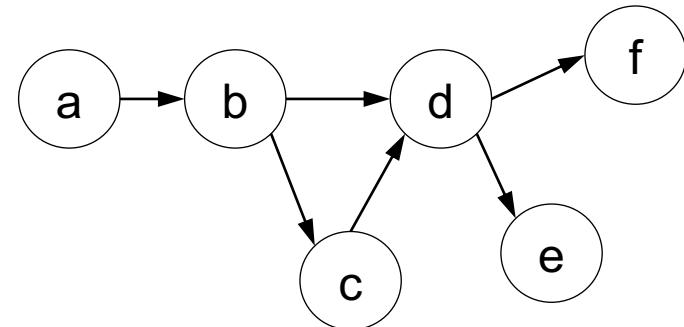


Theory: Graphs

- A graph is a collection of **nodes** and **edges** that connect pairs of nodes.
- Graphs can be *undirected* or *directed*, which has consequences for possible modes of traversal.



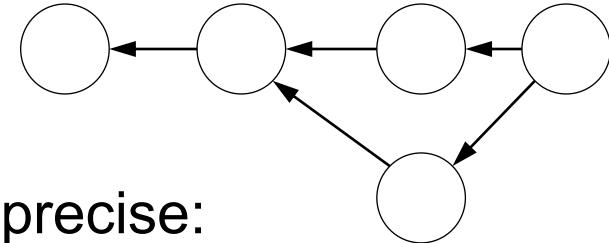
Undirected graph



Directed graph

Theory: Graphs

- A git repository is in fact a giant **graph**.
- A **directed acyclic graph** (DAG) to be precise:
 - **Directed**: edges have a direction
 - **Acyclic**: no path starting from one node will ever lead back to that node
- Each commit creates a new node in the graph. The **parent** of each new commit is then the previous commit.
- git commands are used to navigate this graph, add new nodes, get back to previous ones etc.

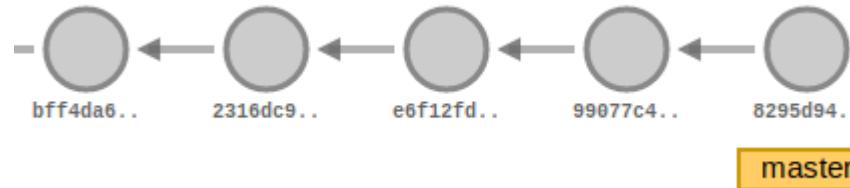


Branches and the Ominous „Master“

```
$ git status
On branch master
nothing to commit, working tree clean
```

Branches are named references to commits and are continuously updated

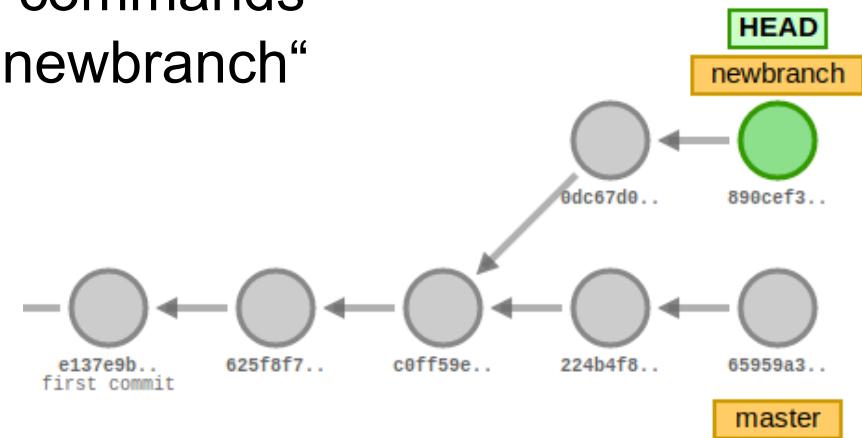
- e.g. to separate work on bugfixes or new features from the main development
- The default/main branch is always called „master“ as a convention.



Think of branches as **bookmarks or pointers** to commits.

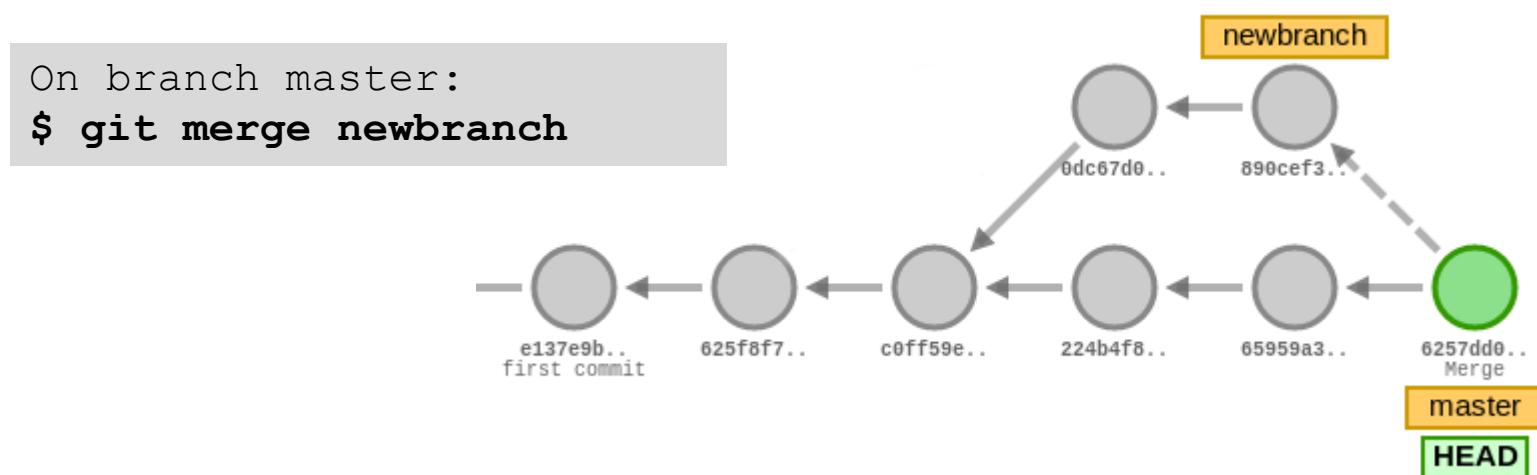
Commands related to branches

- git branch
list all existing branches
- git branch newbranch
create a new branch called „newbranch“
- git checkout newbranch
switch to the new branch
- git checkout -b newbranch
shortcut for the previous two commands -
create a new branch called „newbranch“
and immediately switch to it



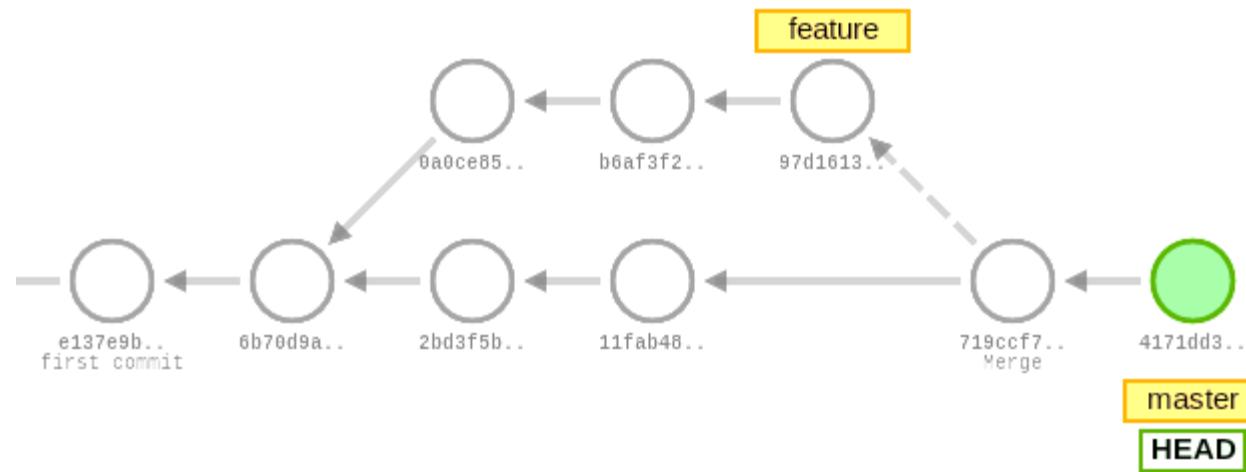
Integrating lines of history: Merging

- To integrate changes made on one branch into another, use `git merge`.
- When merging branches, a new commit (a so-called *merge commit*) is created that combines the most recent commit from each branch and the common ancestor.



- To avoid conflicts, work on two branches should not affect the same files.

Demo



Viewing History: git log

- `git log` provides an overview on your repository's history

```
[...]/example/$ git log
commit 8dffdeb3570dd63612f420fe432401498e0 (HEAD -> master)
Author: Mandy Neumann <mandy.neumann@th-koeln.de>
Date:   Mon Mar 11 11:21:44 2019 +0100
Initial commit
```

So, what are these strange strings of letters and digits?

- Useful options to `git log`:
 - `-p / --patch`: view changes introduced in each commit
 - `--oneline`: one commit per line (short info only)
 - `--stat`: show statistics on modified files
 - `--graph`: display an ASCII graph of branch/merge history

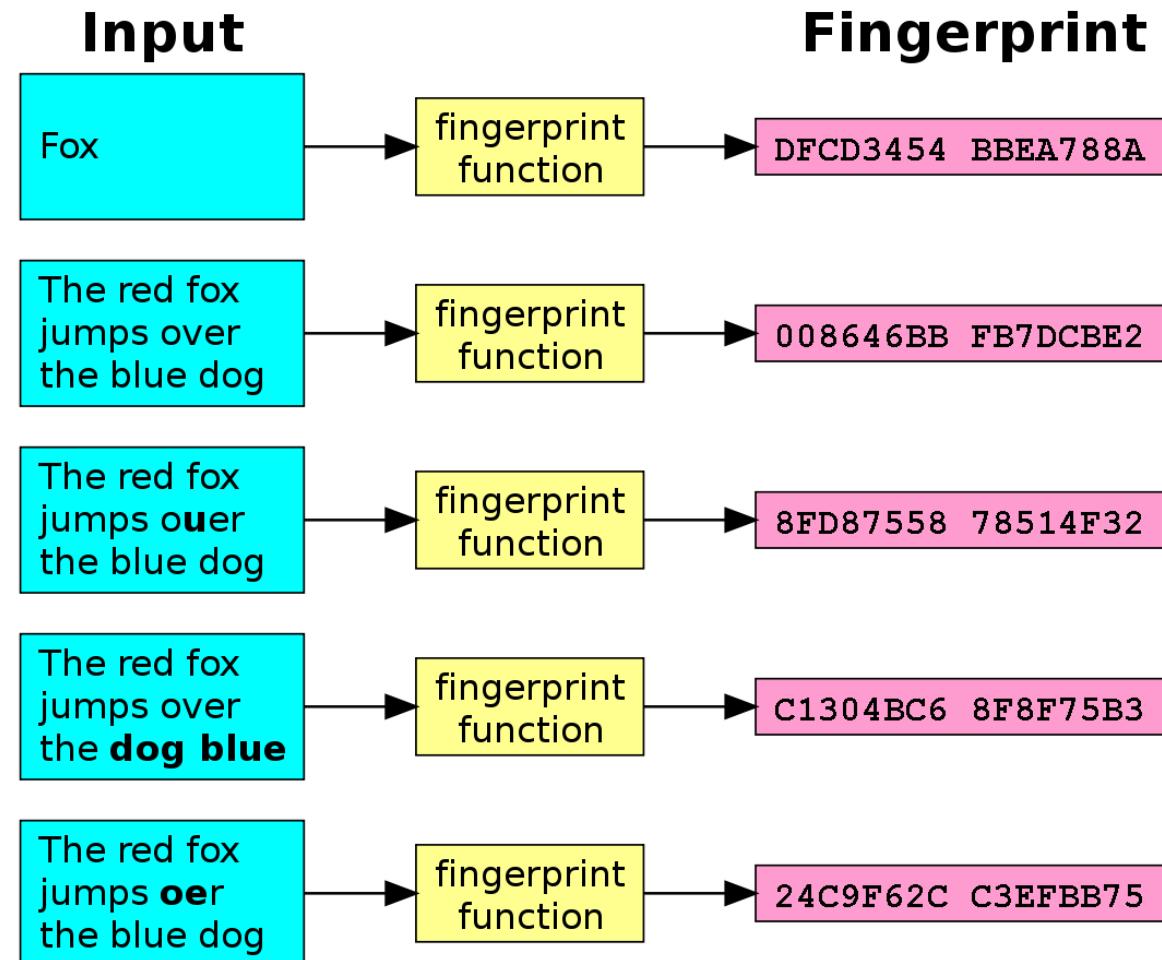
Theory: Hashes

- Every object in git is identified by a unique **hash**. You can think of a hash as a *fingerprint*.



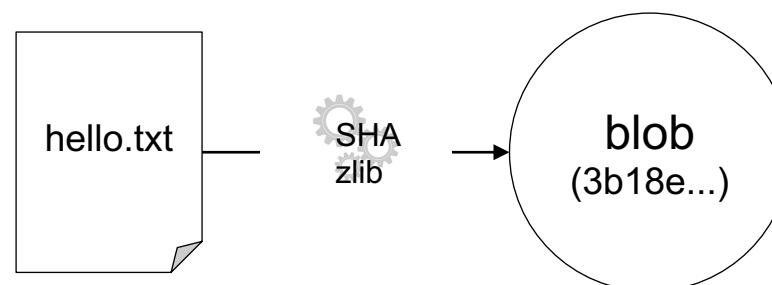
- A hashing algorithm is a **mathematical function** that condenses *variable-length* input to a bit string of *fixed size*.
- It is meant to produce **irreversible** and **unique** hashes.

Even the slightest change results in a completely new hash:



SHA – Secure Hash Algorithm

- Group of standardized **cryptographic** hash functions
 - SHA-1 – 160 bit hash
 - SHA-2 – family of hashes (224, 256, 384 or 512 bit hashes)
 - SHA-3 – same lengths as SHA-2, but completely new structure
- Git computes the *SHA-1* hash of a file with a hash value of 160 bits (which is 20 8-bit bytes or 40 4-bit hex digits) that *uniquely identifies the contents of the file*. The content itself is stored in compressed form in a blob file.



git Internals: The git File System

- git is basically a special file system. File contents are stored in objects called **blobs** (binary large objects), which are grouped together in **trees**.

A ... in a file system	is a ... in git
Directory	Tree
File	Blob

- Every single object in the file system is referenced by its hash value.

git Objects

```
$ echo 'hello world' | git hash-object --stdin
```

```
3b18e512dba79e4c8300dd08aeb37f8e728b8dad
```

40 hex digits

Compute
the object
hash from
input

```
$ git init  
$ echo 'hello world' > hello.txt  
$ git hash-object -w hello.txt
```

```
3b18e512dba79e4c8300dd08aeb37f8e728b8dad
```

```
$ tree .git/objects/
```

```
.git/objects/
```

```
|   └── 3b
```

```
    └── 18e512dba79e4c8300dd08aeb37f8e728b8dad
```

```
    └── info
```

```
    └── pack
```

```
3 directories, 1 file
```

git Objects

```
$ echo 'Hello, World!' > hello.txt
$ git hash-object -w hello.txt
8ab686eafefb1f44702738c8b0f24f2567c36da6d
$ tree .git/objects/
.git/objects/
    └── 3b
        └── 18e512dba79e4c8300dd08aeb37f8e728b8dad
    └── 8a
        └── b686eafefb1f44702738c8b0f24f2567c36da6d
    └── info
    └── pack

4 directories, 2 files

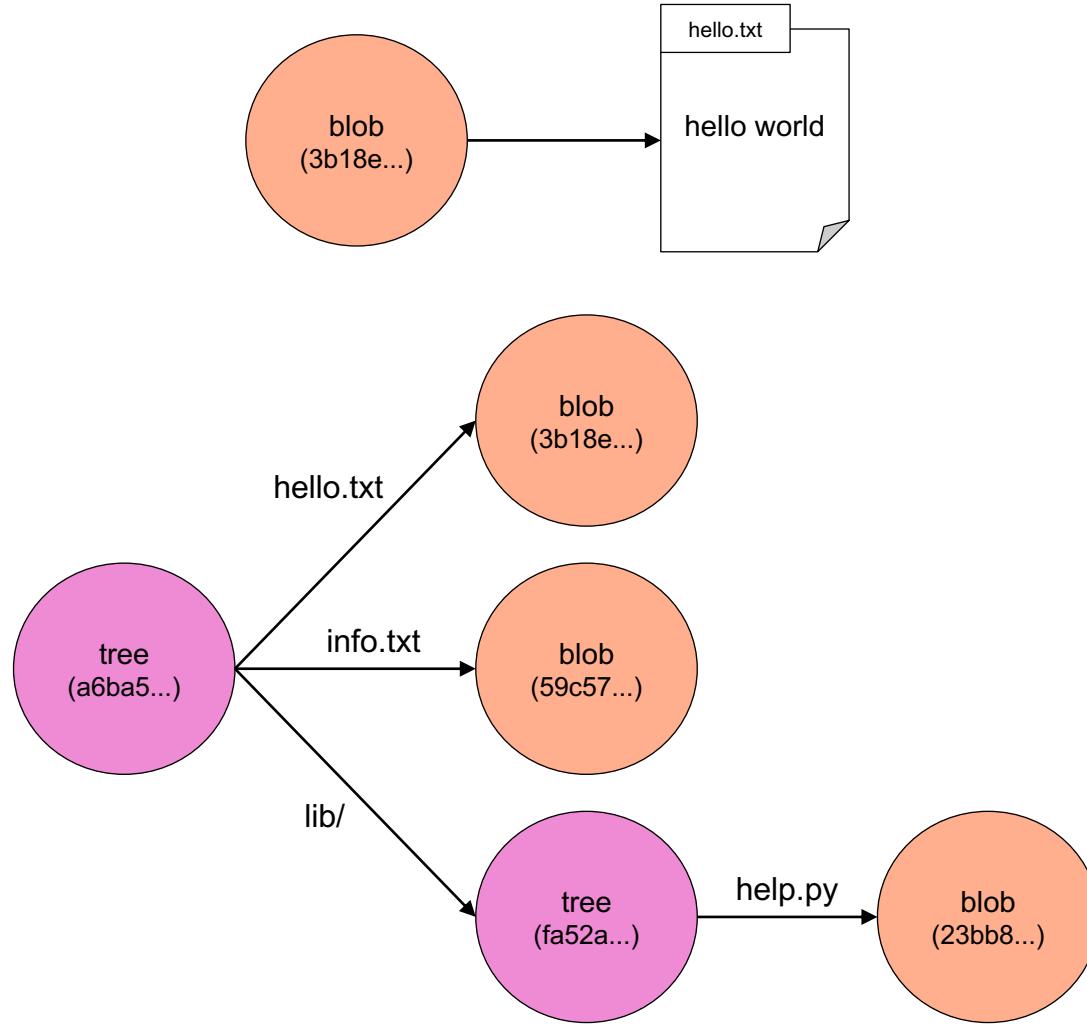
$ ls
hello.txt
$ cat hello.txt
Hello, World!
```

For each new version of a file, a new git object with different hash is created.

Notice that the old version is still there.

But there's only one actual file in the directory.

A Tree Stores Blobs and Other Trees



Some Notes

- Note that only the content of a file is hashed.

Print the
content of
this object

```
$ git cat-file -p 8ab686eafbeb1f44702738c8b0f24f2567c36da6d  
Hello, World!
```

- To also store info about the file like file name, type and mode, **tree** objects are used.
- A tree in git represents directories for blobs and more trees.

```
$ git cat-file -p 8250974359066f3218053315be483201bd7d37e2  
100644 blob 861dbb4fd2cb99053a9b4ef8450aa4f6351b7484 README.md  
100644 blob 8ab686eafbeb1f44702738c8b0f24f2567c36da6d hello.txt
```

{ } { }

mode type

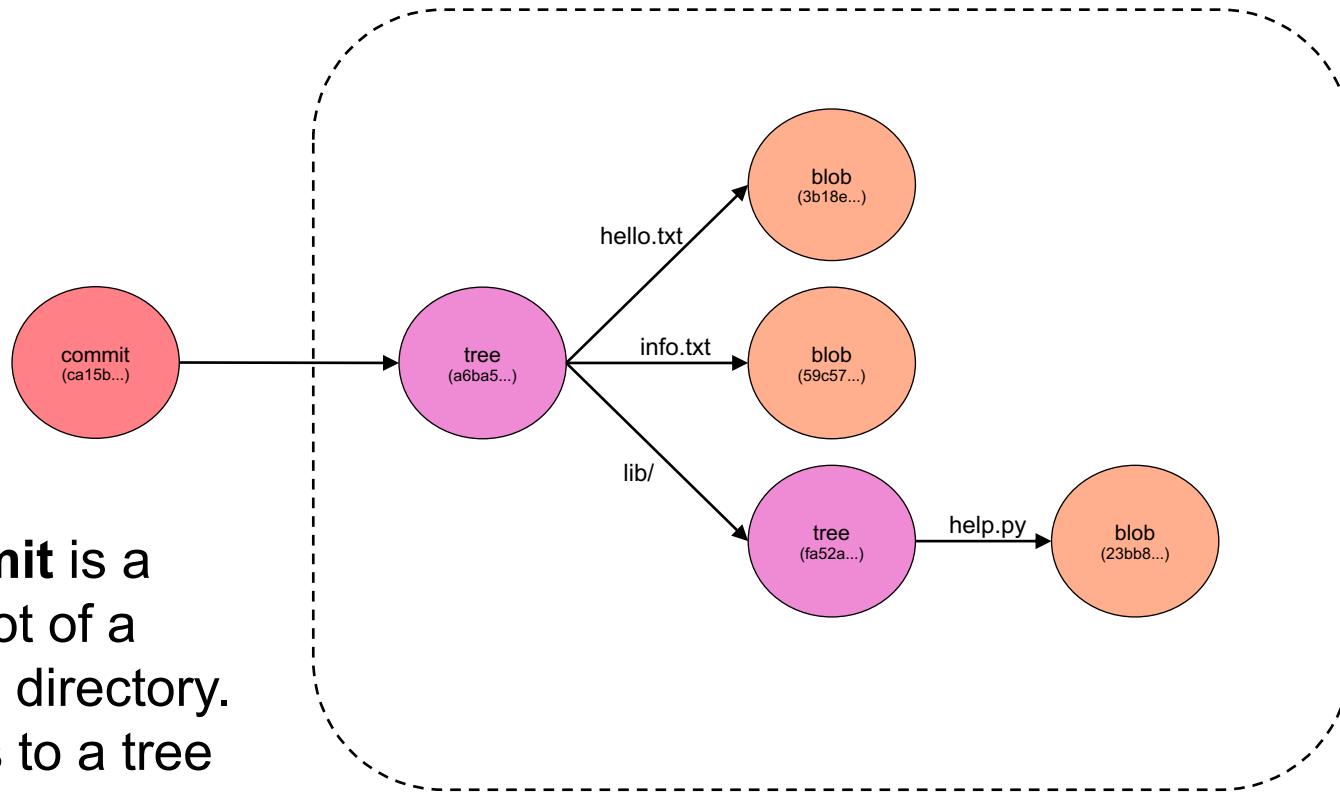
{ }

hash of the object

{ }

file name

Commits Point to Trees and Have Their Own Hashes



A **commit** is a snapshot of a working directory. It points to a tree and stores metadata on the snapshot.

Commits under the hood

```
$ git cat-file -p 8dffdeb3570dd63612f420fe43240149c15708e0  
tree 8250974359066f3218053315be483201bd7d37e2  
author Mandy Neumann <neumann@th-koeln.de> 1552299704 +0100  
committer Mandy Neumann <neumann@th-koeln.de> 1552299704 +0100
```

Initial commit

timestamp

commit message

```
$ git cat-file -p bd5c4ef8aacfc24a660a736659c647f52006af04  
tree 9d960d1b6141ad4e1cb07d80a20b5eb3289de09c  
parent 8dffdeb3570dd63612f420fe43240149c15708e0  
author Mandy Neumann <neumann@th-koeln.de> 1552307124 +0100  
committer Mandy Neumann <neumann@th-koeln.de> 1552307124 +0100
```

Extend hello.txt

the tree object referencing the two committed files

Don't worry

- You neither have to interact with git's internal file system, nor would you normally deal with commit hashes.
- In case you really need to get back to a specific commit, you can use the abbreviated hash with `git checkout`:

```
[...]/example$ git checkout bd5c4ef8
```

Note: checking out 'bd5c4ef8'.

You are in 'detached HEAD' state. [...]



If you want to create a new branch to retain commits you create, you may do so (now or later) by using `-b` with the `checkout` command again. Example:

```
git checkout -b <new-branch-name>
```

HEAD is now at bd5c4ef Extend hello.txt

Only use this
if you really
know what
you're doing!

Summary

git command	Action
git init	Initialize new repository
git add	Add file(s) to staging area
git status	Check status
git commit	Create a snapshot of staging area
git diff	Show differences
git branch	Show available branches
git branch <newBranch>	Create new branch
git checkout <newBranch>	Switch to another branch
git checkout -b <newBranch>	Create new branch and switch to it
git merge	Combine changes from several branches
git log	Show commit history
git show [<branch>]	Show the commit where HEAD (or some branch) points to

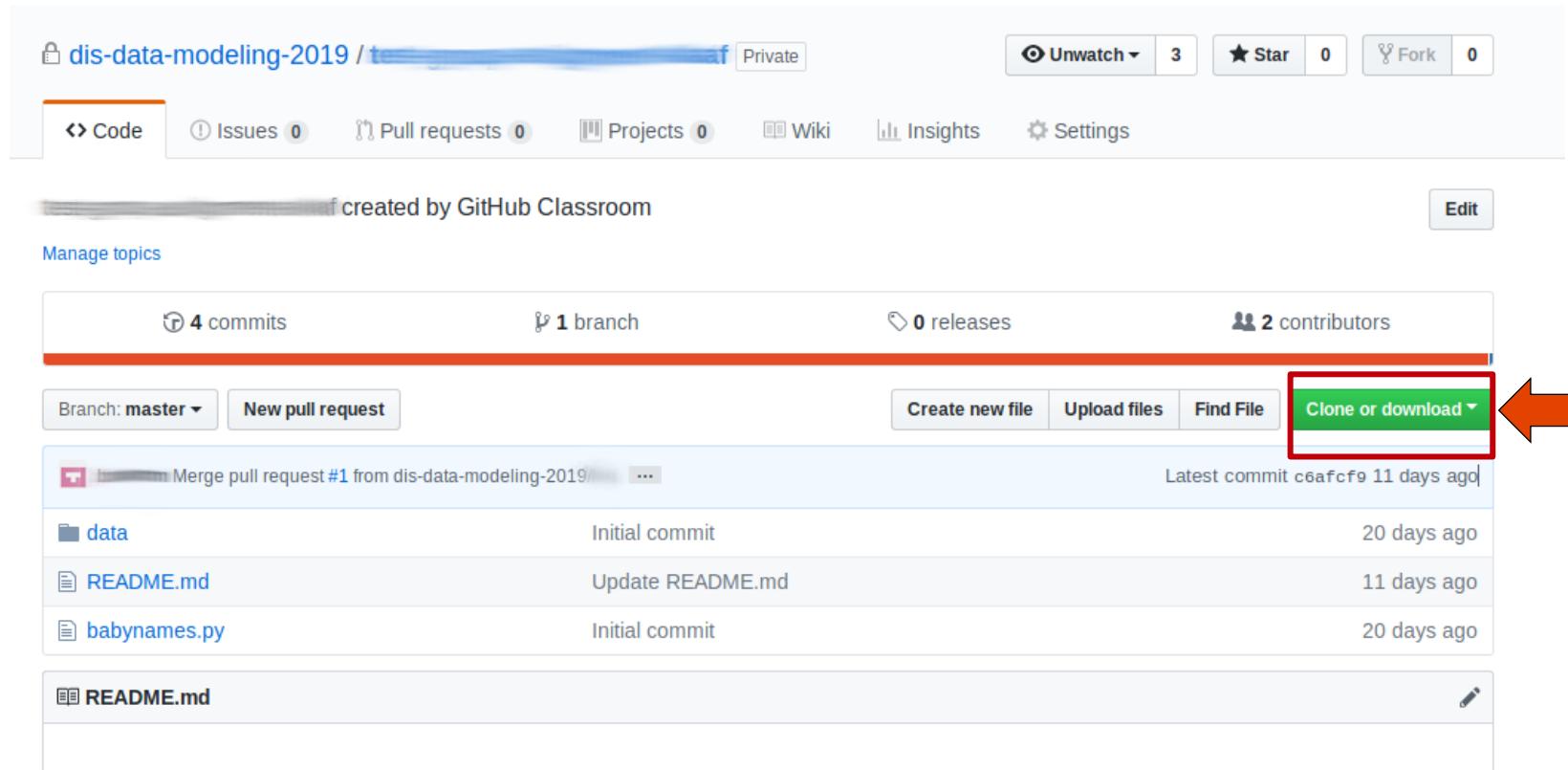


GitHub

Why GitHub?

- While git can be used completely offline, GitHub or similar services make it possible to
 - **Sync** the local repository with a remote one
→ like a cloud backup, and to be able to work on the same project from multiple machines
 - **Share and collaborate** with others
- In this class, we will also use GitHub to **manage the assignments**.

Remote Repositories



The screenshot shows a GitHub repository page for 'dis-data-modeling-2019'. The top navigation bar includes a lock icon, the repository name, a 'Private' button, and social sharing links for Unwatch (3), Star (0), and Fork (0). Below the bar are tabs for Code, Issues (0), Pull requests (0), Projects (0), Wiki, Insights, and Settings. A note indicates the repository was created by GitHub Classroom. A 'Manage topics' link is present. Key statistics are displayed: 4 commits, 1 branch, 0 releases, and 2 contributors. A red box highlights the 'Clone or download' button, which is green with white text. An orange arrow points to this button from the right side of the image. Below the stats, a dropdown shows 'Branch: master' and a 'New pull request' button. A list of commits is shown, with the latest being a merge pull request from 'dis-data-modeling-2019'. The commit history includes:

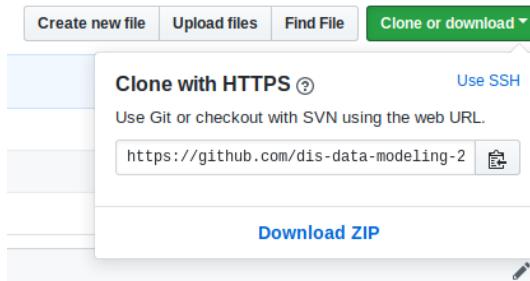
- Merge pull request #1 from dis-data-modeling-2019 (11 days ago)
- data (Initial commit, 20 days ago)
- README.md (Update README.md, 11 days ago)
- babynames.py (Initial commit, 20 days ago)

A file named 'README.md' is listed at the bottom.

- Existing repositories are **cloned** to a local machine.

Remote Repositories

- Copy the URL (use HTTPS method)...

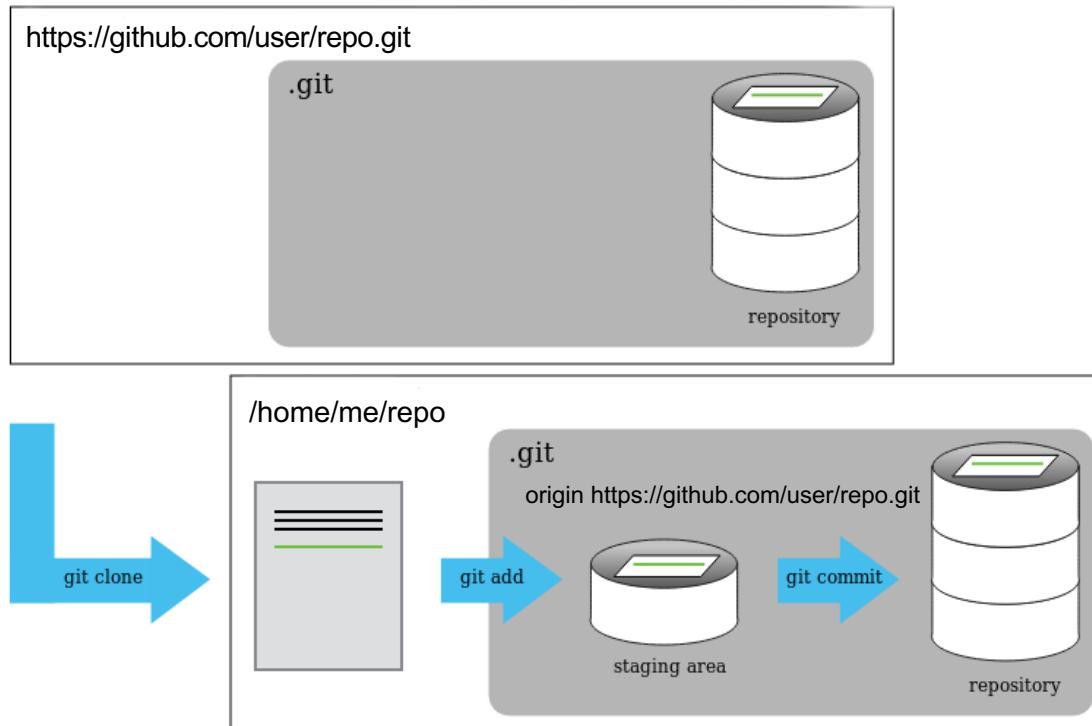


- ... and clone to your local machine:

```
$ git clone https://github.com/dis-data-modeling-2019/<repository>.git
Cloning into '<repository>'...
remote: Enumerating objects: 22, done.
remote: Counting objects: 100% (22/22), done.
remote: Compressing objects: 100% (12/12), done.
remote: Total 22 (delta 8), reused 15 (delta 8), pack-reused 0
Unpacking objects: 100% (22/22), done.
```

Remote Repositories

- Now you've got a local copy of the remote repository in a directory with the repository's name.



You can now go on and work with it – add and change files, tell git to track them (`git add`), snapshot your work (`git commit`), check the status of your working directory (`git status`) and the history of commits (`git log`).

Syncing with Remotes

- Check available „bookmarks“ to remote repositories:

```
$ git remote -v
origin  https://github.com/neumannm/example.git (fetch)
origin  https://github.com/neumannm/example.git (push)
```


name URL

- Send your recent commits to the remote repository:

```
$ git push
Username for 'https://github.com': neumannm
Password for 'https://neumannm@github.com':
Counting objects: 7, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (7/7), 605 bytes | 605.00 KiB/s, done.
Total 7 (delta 0), reused 0 (delta 0)
To https://github.com/neumannm/example.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from
'origin'.
```

Syncing with Remotes

- Update your local copy by **pulling** remote changes:

```
$ git pull
```

```
Username for 'https://github.com': neumannm
```

```
Password for 'https://neumannm@github.com':
```

```
[...]
```

```
Unpacking objects: 100% (3/3), done.
```

```
From https://github.com/neumannm/example
```

```
  bd5c4ef..5e7d3f8  master      -> origin/master
```

```
Updating bd5c4ef..5e7d3f8
```

```
Fast-forward
```

```
 README.md | 2 ++
 1 file changed, 2 insertions(+)
```



Summary

Git command	Action
git remote -v	List info on remotes
git remote add <bookmark> <url>	Add a new remote named „bookmark“ at given url
git fetch	Fetch latest changes from remote
git pull	Fetch and integrate latest changes from remote
git push	Push latest changes to remote
git push --set-upstream <bookmark> <branch> or git push -u <bookmark> <branch>	Tell git to always use the remote „bookmark“ when pushing „branch“ (only needed once)

Resources

- Official reference, free eBook, videos and more: <https://git-scm.com/doc>
- Cheat sheet by GitHub: <https://github.github.com/training-kit/downloads/github-git-cheat-sheet.pdf>
- Visual git cheat sheet: <http://ndpssoftware.com/git-cheatsheet.html>
- Visual git reference:
<https://marklodato.github.io/visual-git-guide/index-de.html>
(German)
<https://marklodato.github.io/visual-git-guide/index-en.html>
(English)
- Interactive online course to learn git (paid-only):
<https://www.codecademy.com/learn/learn-git>