VARIOUS

- To eliminate a file in a rep through cmd line (source: https://github.community/t5/How-to-use-Git-and-GitHub/How-to-delete-multiples-files-in-Github/td-p/4623)
    1. In the command-line, navigate to your local repository.
    2. Ensure you are in the default branch: **git checkout master**
    3. The rm -r command will recursively remove your folder: **git rm -r fileName**
    4. Commit the change: **git commit -m "Remove file"**
    5. Push the change to your remote repository: **git push origin master**

- Difference between Projects and Repositories in Github:

> GitHub recently introduced a new feature called Projects. This provides a visual board that is typical of many Project Management tools:
>
> A **Repository** as documented on GitHub:
>
> A repository is the most basic element of GitHub. They're easiest to imagine as a project's folder. A repository contains all of the project files (including documentation), and stores each file's revision history. Repositories can have multiple collaborators and can be either public or private.
>
> A **Project** as documented on GitHub:
>
> Project boards on GitHub help you organize and prioritize your work. You can create project boards for specific feature work, comprehensive roadmaps, or even release checklists. With project boards, you have the flexibility to create customized workflows that suit your needs.
>
> Part of the confusion is that the new feature, Projects, conflicts with the overloaded usage of the term project in the documentation above.
>
> Source: https://stackoverflow.com/questions/40509838/project-vs-repository-in-github#:~:text=GitHub%20Repositories%20are%20used%20to,resources%20which%20you%20care%20about.&text=If%20you%20create%20a%20project,multiple%20projects%20in%20a%20repository.

LEARNWEBCODE

- Projects = Repository (o repo) in GIT
- git config --global user.name “name”
  git config --global user.email “email” so that git can put the right name and email beside the commitments

- **workflow** to create a **new repo from scratch onto your hard drive**
    a. create a folder on your computer with a generic name (projects, for istance)
    b. go to that folder with the command line (git bash on windows, terminal on mac)
        - pwd to see where the command line is pointing at the moment (current working directory)
          to change directory, 2 possibilities:
            - cd folderPath
            - cd + drag and drop the folder just created on the terminal
    c. mkdir “projectsTitle” to make a directory (a folder) from the command line
    d. cd “projectsTitle” to get into the directory just created (or cd + drag&drop ...)
    e. git init to make GIT starting a new repo on this folder (it starts to follow what’s happening here)
    f. touch “fileName.ext ” to create a new file inside the folder (from the terminal)

g. git status
h. git add “fileName.ext”
    - git status to check
i. git commit -m “my first commit”

- **workflow** to create **a repo onto your hard drive downloading it from the cloud**
    a. cd projectsPath (the generic folder you created in the previous step a)
    b. git clone repoUrl

- **workflow** to **link your repo into the hard drive to a GITHUB repo**
    a. go to github, set a new repo with the exact name of the repo (folder) on your hd (do not set a readme file)
        - the premium account can be useful if you are working on projects involving passwords (for istance in a config file)
            - there’s also another service that gives private account for free: big bucket.

    b. **git remote add origin gitRepositoryAddress**
        - it’s the https address close to the name of the rep (example:)
    c. **git push -u origin master**

if the folder comes to a cloned rep that you want to upload to your own rep, change the address
    a. **git remote set-url origin newUrl** to set a different url for GIT push
        - newUrl is the web address of your github repo with the same name (you can find it in overview)
    b. **git remote -v** to see the address GIT will use to push your files
    c. **git push origin master** to push the file from your local hd to the online rep
        - origin stands for the address of our repo
        - master because we are on the master branch
        - the first time you do step c, you will probably asked to write your github username and password

BASIC KNOWLEDGE

- git is a version control system
  it is the standard for the web development industry

- git workflow:
    a. **working folder** (where adding, deleting, editing happens)
    b. **staging area** (in which you list the changes made to the file, getting the file ready to be committed)
    c. **repository** (after commit, changes are saved permanently as a different version of the file in the repository)

- **git init** to initialize a new git repository, starting a project from the terminal

- **git status** to check the status of the files in your local folder (untracked files need to be added to the staging area for git starting to tracking them)

- **git add fileName.ext** to tell git to start following the file (file -> staging area)
  git add fileName1.ext fileName2.ext to add more file simultaneously
  git add -A to add all the files have been modified
  git add . to add all the files in the working folder at once

    a. then check: git status (the file is now tracked)

- **git diff** fileName.txt to get the difference between the file already in the staging area and the new edits done in the working area.
    a. white: the file in the staging area
    b. + green: the file in the working folder (new edits)
    c. to exit from the diff mode: press q on your keyboard
        ■ git add fileName.txt to add the file with new changes to the staging area

- **git commit -m "**comments**"** to store the file in the repository area (staging area -> repository)
    a. comments need to be written in present tense
    b. comments using -m should be brief (no more than 50 chars)
    c. comments must be in quotation marks
    d. **git commit -am** "comments": add and commit the modified files which are already in the stage area

- **git log** to get the list of changes made to the file:
    a. in orange the 40 chars unique indentifier (SHA) for the commit
    b. then the author of the commit
    c. then date and time of the commit

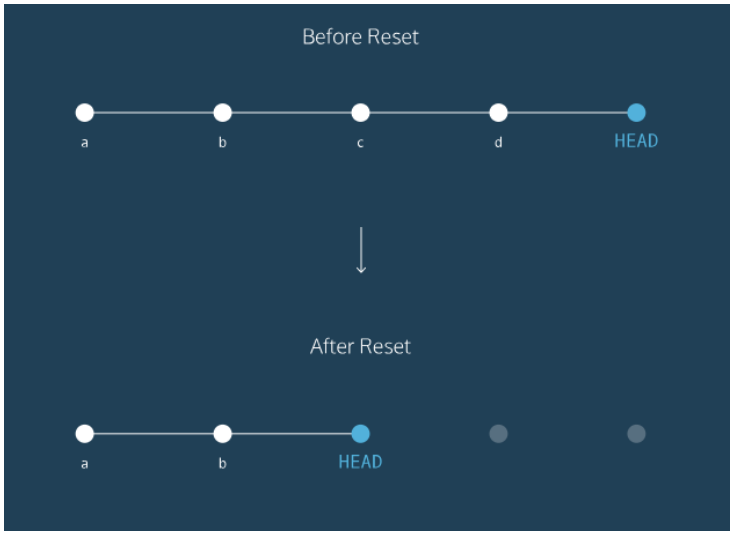
BACKTRACKING

- **git show HEAD** to see the commit you're currenlty working on (called HEAD commit).

- **git checkout HEAD** fileName.ext to get to the latest committed version of the file (HEAD commit), discarding the changes you're making to the file in the working directory [#working directory]
    a. **git checkout --** fileName.txt is a common shortcut, it does exactly the same thing
    b. **git ckeckout -- .** to apply to all the fileresets in a project (to be verified, I'm not super sure)
        ■ git diff fileName.txt to check if the changes are still there (close and reopen the file to see the changes)

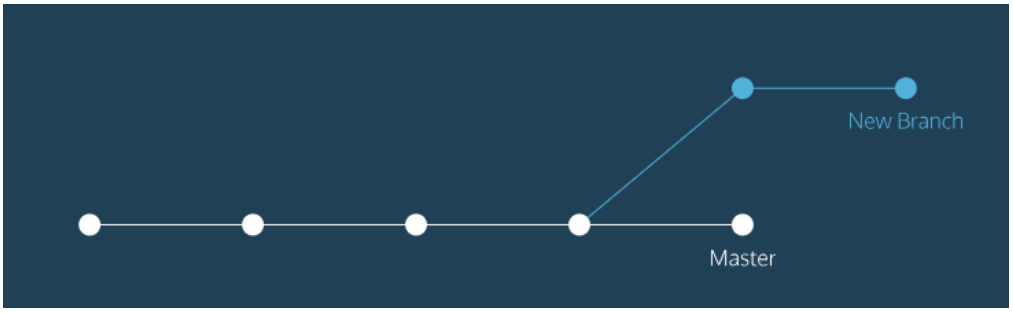
- **git reset HEAD** fileName.ext to unstage a file: it does not discard file changes from the working directory, it just removes them from the staging area referring to the last commited version of the file [#staging area]
    a. it can be useful to unstage only one of the files you're ready to commit
    b. in the output, M is short for modification

- **git reset** commitSHA to go back to a previous committed version from your current HEAD commit.
    a. commitSHA is = to the first 7 characters of the commit's SHA you want to get back to.
    b. how to:
        ■ git log to choose the version you want to go back to
        ■ git reset commitSHA
            • if you do git log again, you'll see the versions after the one you chose are no more there
        ■ to reset also the working area: git checkout HEAD fileName.ext



BRANCHING

- Git allows us to create branches to experiment with versions of a project without changing the master branch



the circles are commits, and together form the Git project's commit history
New Branch is a different version of the Git project: it contains commits from Master (until its creation) but also has commits that Master does not have (the ones after its creation)

- **git branch** to know on which branch I'm working on
  - in the output, the * (asterisk) is showing you what branch you're on (also coloured in green)

- **git branch** branchName to create a new branch
  - branchName can't contain white spaces (branch-Name or branch_Name are ok)

- **git checkout** branchName to move to another branch
  - you can always verify where you are with git branch
  - to go back to the master branch: git checkout master

- **git merge** branchName to merge the new branch into the master one
  (probably you need to switch to the master branch before - git checkout - but I'm not sure)
  - your goal is to update master with changes you made to the new one:
    the new branch is the giver branch, since it provides the changes
    master is the receiver branch, since it accepts those changes.
  - In the output: the merge is a "**fast forward**" because Git recognizes that the new branch contains the most recent commit.
    - if you changed smt in your master version after you made a new branch, and maybe you changed the exact file you edited also in the new branch, git doesn't know which changes has to keep when you merge them = **merge conflict**

- => in the output, notice the lines:
  "CONFLICT (content): Merge conflict in fileName.ext
  Automatic merge failed; fix conflicts and then commit the result."
- in the code editor, Git uses markings to indicate the HEAD (master) version of the file and the new branch version of the file (in the example, named fencing), like this:

```
<<<<<<< HEAD
master version of line
=======
fencing version of line
>>>>>>> fencing
```

- to solve the problem you manually need to delete the part you don't want to keep from the file as well as all the marking done by Git (====, >>>>).
- save the file, then stage again the file (git add) and then commit it (git commit, usually with a comment like "Resolve merge conflict")

- **git branch -d** branchName to delete a branch you don't need anymore: Git branches are meant to be temporary and to be deleted after merged with the master one.
  - if the branch was never merged into the master one, use **-D** instead of -d
  - to delete more branches at once git branch -d branchName1 branchName2
  - to verify, git branch

TEAMWORK

- a remote is a shared Git repository that allows multiple collaborators to work on the same Git project from different locations. Collaborators work on the project independently, and merge changes together when they are ready to do so.
  a. Git projects are usually managed on Github, a website that hosts Git projects for millions of users

- git clone remote_location clone_name to start working: smn created a repository in a directory shared in a network, now you want a copy of it on your computer
  a. remote_location is the web/local address of the
  b. clone_name is the name of the directory (you chose) in which Git will copy the repository.
  c. Output: cloning into "..."
     - if you commit changes in your local rep, they won't be uploaded in the remote rep.

- git remote -v to see a list of Git's project remotes
  a. get to the folder of your cloned rep with the command cd (from terminal), then git remote -v
     - behind the scenes, when you clone a remote, Git gives the remote address the name origin, so that you can refer to it more conveniently.
     - Git automatically names this remote origin, because it refers to the remote repository of origin. However, it is possible to safely change its name

- **git fetch -v** to see if the remote origin has got some changes and in case it will bring them to a remote branch, not merging them into your local copy: they are in origin/master
  a. remember to get to your local clone first (cd filePath)

- git merge origin/master to merge the changes made in the remote origin/master to your local clone.
  a. remember to get to your local clone first (cd filePath)

- The workflow for Git collaborations typically follows this order:
  a. Fetch and merge changes from the remote

b. Create a branch to work on a new project feature (branchName)
c. Develop the feature on your branch and commit your work
d. Fetch and merge from the remote again (in case new commits were made while you were working)
e. Push your branch up to the remote for review
    - Steps a and d are a safeguard against merge conflicts, which occur when two branches contain file changes that cannot be merged with the git merge command.

- git push origin branchName to make the new branch you have worked on available to the person who created the origin, so that it can be reviewed and eventually merged to the remote origin/master branch.
    a. remember to get to your local clone first (cd filePath)
    b. In the output, notice the line:
    To /home/ccuser/workspace/curriculum/science-quizzes
     * [new branch]   bio-questions -> bio-questions
    Git informs us that the branch bio-questions was pushed up to the remote.

RESOURCES

- https://git-scm.com/book/it/v1/Per-Iniziare-Basi-di-Git