`git init`

- running it in an empty folder creates .git directory in the folder

`git status`

- shows branch you are working on, commits, untracked/modified but tracked files

`git add <location of untracked file>` or `git add .`

- to track a file, we cannot commit it directly, we first need to add it to the staging area
- `git add .` will track all untracked files in current working directory
  - otherwise you can specify which untracked files you want to track by `git add <location of untracked file>`
- git add will create history but not create a restore point with a message(called a commit)

`git commit -m "<commit message>"`

- stage is now empty as we made a commit, `git status` says nothing to commit, working tree clean

`git restore --staged`

- if you committed by mistake you can undo the commit using this command, modified files will be back on the stage as if you had added them
  - you can only rollback one commit behind

`git log`

- prints the commit history with each commit having its commit id

`git reset <commit id>`

- if you want to roll back to older commit
- git log removes the commits above `<commit id>` including the `<commit id>`
  - these commits were unstaged so you need to add them to the stage before committing again so you have untracked changes

`git stash`

- when you want to reset, but could modify the files and still might want to undo the reset later
- before stashing, files should be staged using add
- status after stashing is nothing to commit, working tree clean
- allows you to try reimplementing something without having to commit incomplete work that you were implementing before
- rolls back to most recent commit

`git stash pop`

- if something was stashed it is moved to staging
- this may override the changes you have made between stashing and popping the stash

`git stash clear`

- suppose you successfully reimplemented something so you don't need the stashed incomplete work anymore

`git remote add origin https://github.com/<username>/<reponame>.git`

- binds a git repo on github to a local git folder
- origin means that `<username>` is the owner of the `<reponame>`

`git push origin master`

- uploads the local git folder/new commits to github.com

Branches

- the master branch reflects production ready code that is not work in progress
- to start working on a feature addition you should create a separate branch and merge it with master when the work is done
  - meanwhile it is possible that the same files you worked on were modified by someone else as they got their branch merged
  - lets suppose this did not happen, then your branch can be merged smoothly

1. `git branch <name of new branch>` creates the new branch
2. `git checkout <name of new branch>` like git add
3. `git merge <name of new branch>` merges the branch with master
   - commits made in the new branch will be visible from master branch

Contributing

- origin URL is the forked version which you have access to
- upstream URL is the repo you forked
- once you clone the fork locally run `git remote add upstream <upstream URL>`
- NEVER COMMIT TO ORIGIN, Create a separate branch instead as it is considered a good practice
    - once the last commit is made, we can make a pull request
        - git restricts one pull request per branch so if you had used your origin branch then for another set of features you would have to create a new fork unless the older fork is merged
            - hence we create different branches for different features so that we can have multiple pull requests about specific features
        - further commits from that branch will be reflected in the previously made pull request
        - this ensures that the maintainer knows that all commits in a single pull request can be related to a single feature
            - instead of all commits in a single pull request can be related to a multiple unrelated features

force push

- lets say if the pull request shows some commits that were made
- then I locally reset and stash some commits
- we cannot simply push it as the pull request contains commits that my local machine doesn't
- hence we need to add `-f` argument during pushing

`git fetch`

- when I was working on my branch on a file x, some changes were made upstream to files y and z

- I cannot see those changes in my fork/branch till I fetch those changes
- firstly you change to the main branch of your fork using `git checkout main`
- run `git checkout --all --prune`
- now reset main branch of origin to the main branch of origin using `git reset --hard upstream/main`
- `git push origin main`

or just use `git pull` instead

- `git pull upstream main` and `git push origin main` will do the same as above
- **WHENEVER YOU CREATE A NEW BRANCH OF ORIGIN PULL THE CHANGES FROM UPSTREAM BEFORE**

`git rebase -i <commit id>`

- merges all commits above commit id interactively
- replace `pick` with `s` stands for squash meaning those commits will be merged into one single commit with the nearest push commit above it
- then create a new message for the merged/squashed commit

Merge Conflicts

- when the maintainer has 2 or more pull requests that modify the same lines of code
- when we try to merge one of the pull request we get a merge conflict
- we need to resolve it manually by specifying code from which pull request should be merged
    - then you mark the conflict as resolved