Introduction to git

Getting started with the most universal Version Control Software

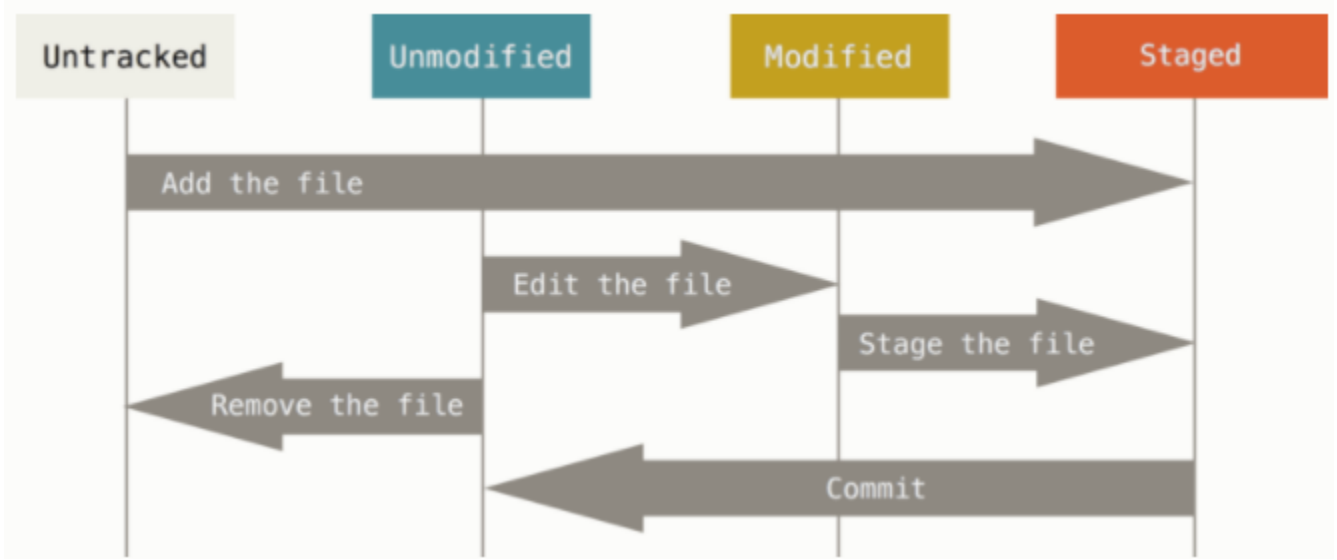
by Linus Pust

Goal of this Session

- Learn the basic git commands in a command line
- A basic understanding of the git data structure
- Being able to differentiate between commands

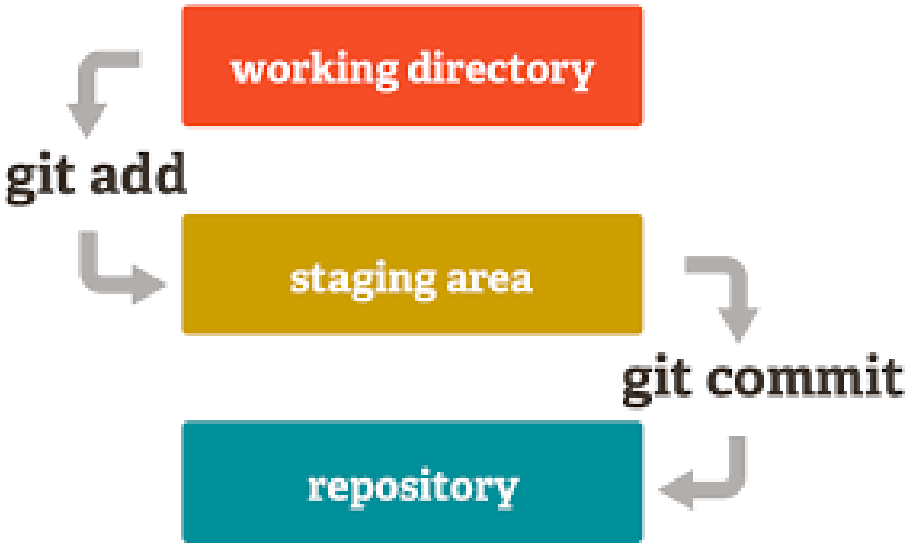
Staging- What is it?

- There are multiple stages your files can be in before you are ready to commit them
- When „staged“, they are ready to be moved to the next commit
- Stages are imaginary



The Three Trees of Git

- Working Directory
  - Represents the local directory
  - Files can be directly accessed
  - To view current state, use `git status`
- Staging Index / Staging Area
  - Tracks all changes to be stored in the next commit with `git add`
  - Allows for many versions of the same files
  - Can be found in .git/index file
- Commit History (Repository)
  - „Snapshot“ of working directory AND Staging Index at time of commit
  - To view, use the `git log` command (more infos with option –p)



git add

- git add moves the selected file(s) into the Staging Index tree
  - Can be used to track new untracked files
  - Can also move tracked/modified files into the staging area

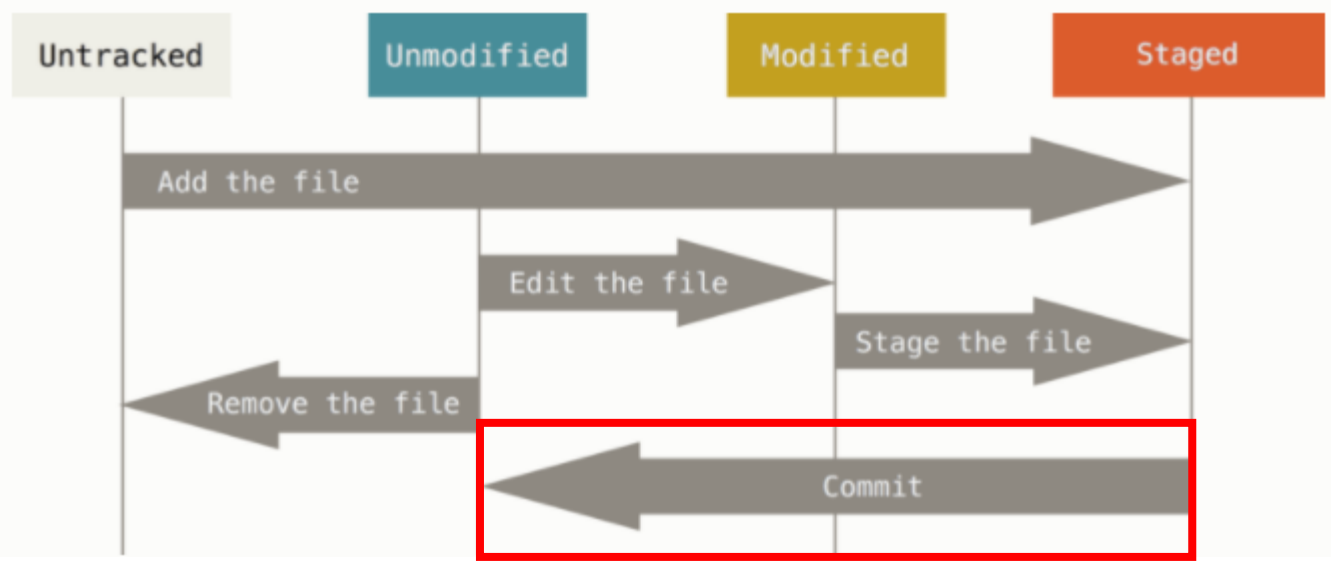
- With `git add .` all files are added to the staging area
- To select parts of a file to stage but ignore other parts, use
  - `git add --interactive <FileName>`
    - There you can select which parts you want to include in your next commit and which not to include

git fetch vs git pull

- Fetch: Safer option
  - Downloads remote but will not change local working directory
  - Will not execute git merge before you want to
  - Syntax: `git fetch <remote> <branch>` (branch is optional)
- Pull: Faster option
  - When time is the key factor, use `git pull`
  - Will immediately merge remote branch with your latest local commit
  - Syntax: `git pull <remote> `
  - Use `git pull --rebase <remote>` to use rebase instead of merging

git commit

- Creates new „snapshot“ of your current project
- Moves content of staging area to the commit history
  - Index-file moved to objects folder



amend

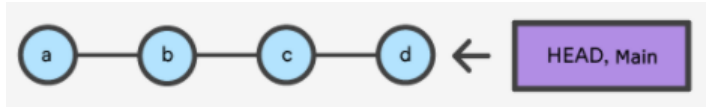
- Amending allows to retroactively **change last commit**
- Possible changes are
  - Combine current staged changes with last commit
  - Edit the commit message of your last commit
- Amending creates a new commit with new ID
- WARNING: Don‘t amend a public commit, especially after people already fetched it
  - Amending a public commit can quickly result in continuity errors
- Syntax: git commit --amend

What should you take away?

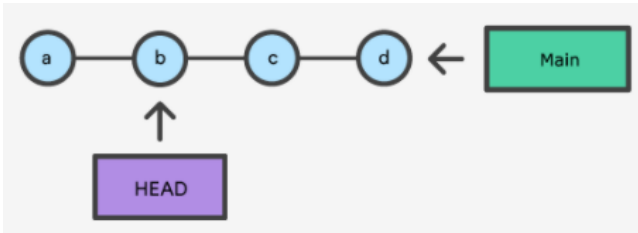
- Stages are imaginary states for files to keep them organized
  - The „three trees of git“ are the actual internal git structure
- Git add updates the index of your repository
- Use git fetch to safely update your repository
- Use git pull to quickly fetch and merge / rebase in one command
- Git commit saves current state of your project
  - Use the amending function to edit your last commit

Checkout

- Combines multiple functions
  - Roll back to certain commit
  - Switch branches
  - Delete unwanted commits
  - Restore files from index
- When checking out branch creates a detached HEAD
- Checking out file keeps HEAD in position
- Can create branch and directly check out with
  - git checkout –b <new-branch> [<existing-branch>]
- WARNING: Don‘t develop new features while in detached HEAD state!



git checkout b



detached HEAD

git switch

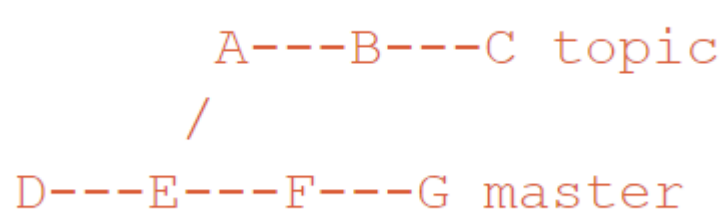
- New git command
  - It takes functions from git checkout to mitigate possible confusion
- Switches HEAD to specified branch
- Staging area and working directory are kept the same
- Equivalent to `git checkout <branch>`
- To create new branch and then switch use `git switch –c <branch>`

git restore

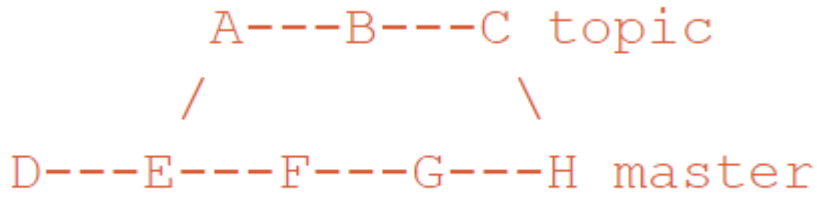
- New git command
  - It takes functions from git checkout to mitigate possible confusion
- Restores files from index into working tree
- Syntax: *git restore <filename>*
- Equivalent to: *git checkout <filename>*

git merge

- Merge two branches into a new commit on one of the branches
- Does not change any of the previous commits of either branches
- Syntax: git merge <branch-name> (switch to merging branch before)



Origin position: current branch is „master“

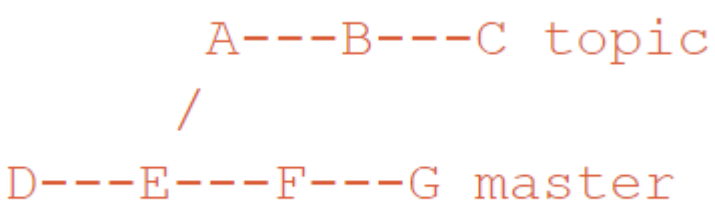


Created new commit `H` by merging „topic“ into „master“, but didn‘t change any other commit

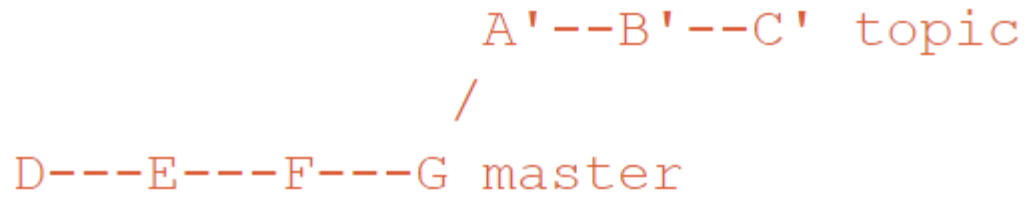
source: https://git-scm.com/docs/git-merge

git rebase

- Merge by creating a continous timeline including two branches
- Searches for last common commit and rebases from there on
- Essentially moves the branching point forward



Origin position; topic branching point is at the `E` commit of the master branch

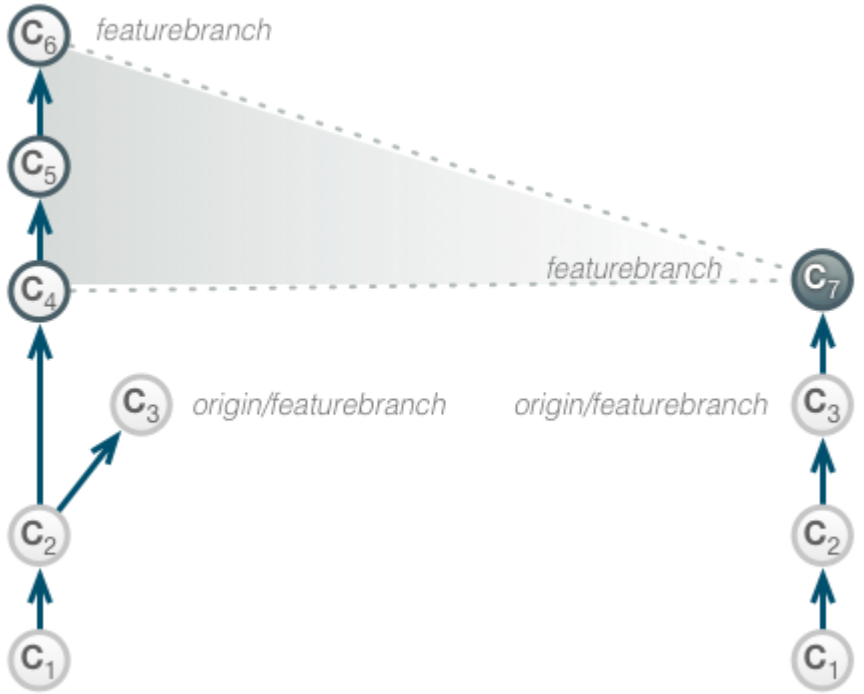


By rebasing, the merging point is now essentially at the `G`commit of the master branch

source: https://git-scm.com/docs/git-rebase

Squashing

- Squashing is an option when using `interactive rebase` or `merging`
- Squashing will transform all commits from specified point into one
- How-to squash
  - `git rebase -i <squash-from-ID>` will squash a certain amount of commits
  - `git merge --squash <branch>` will squash commits and delete source-branch

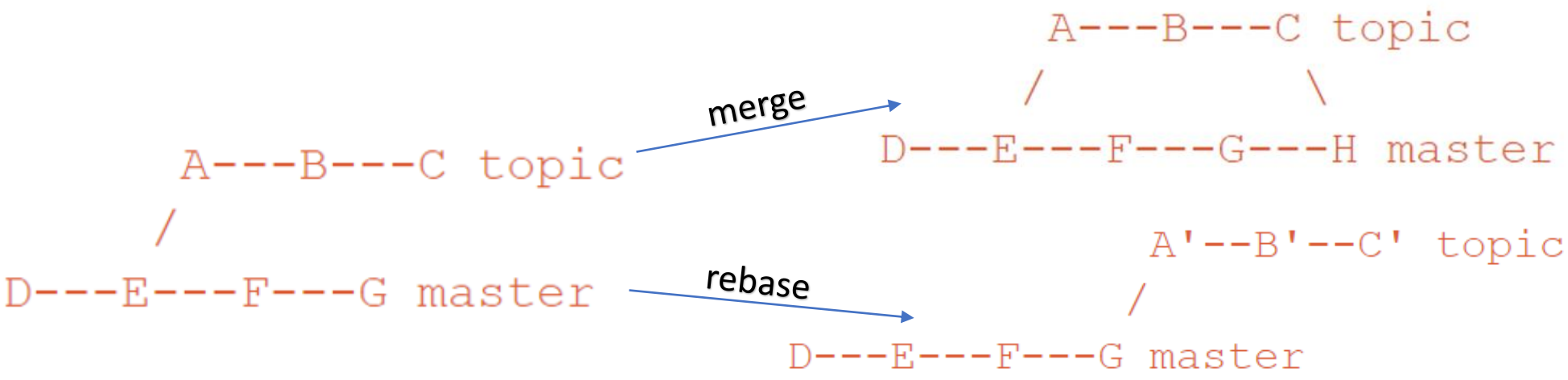


Squashing C5,C6 into C4 with interactive rebasing

source: https://www.sascha-just.com/scig/

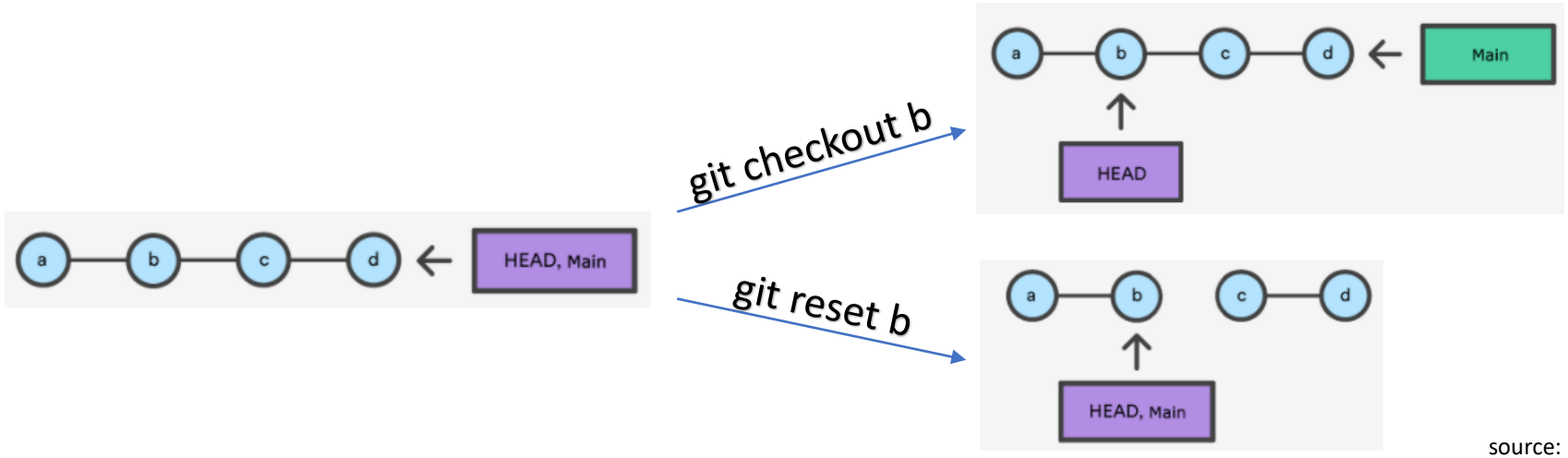
What should you take away?

- Checkout is a very versatile command
  - Use `git switch` instad of `git checkout <branch>`
  - Use `git restore` instead of `git checkout <file>`
- What is the difference between merge and rebase?
- Squashing can help keep organized



Git reset

- Similar to `git checkout <branch>` or `git switch <branch>`
- Moves HEAD and branch pointers
  - Checkout only moves HEAD
- **Resetting can lose data!**
  - Use git reflog as a way to restore them
  - Garbage collector runs (by default) every 30 days



source: https://www.atlassian.com/git/tutorials/undoing-changes/git-reset

Git reset modes

- --hard
  - will reset all pending changes, working directory and staging index of the specified commit
  - **VERY DANGEROUS**, data cannot be restored!
- --mixed (default)
  - staging index of commit are reset
  - changes stored in staging index are moved to working directory
- --soft
  - Only resets the commit tree to desired location
  - Doesn‘t reset anything from staging index or working directory

git reset example

- `git reset HEAD <filename> ` to unstage files
  - ` git reset HEAD * ` will unstage all files
- ` git reset HEAD~<x> ` will remove x commits from you local repo
- ` git reset <commit-id> `

git revert

- Does not move HEAD
- Reverses specified Commit and creates a new „reverted commit“
- All commits between reverted and HEAD stay in the commit history
- Safer and oftentimes easier than using git reset
- Without arguments it will revert HEAD-commit
- Syntax: `git revert <commit-id>`

Before revert:

a – b – c – d

After reverting b:

a – b – c – d – ~~b~~

=

a – c – d

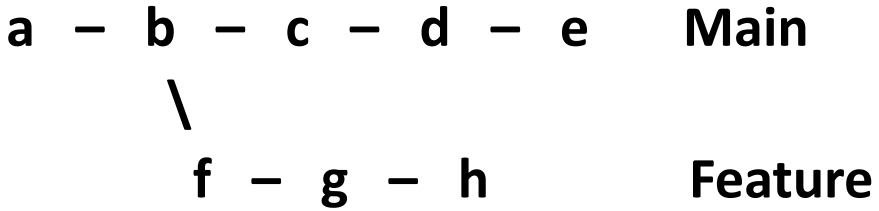
Cherry-pick

- Alternative to merging
- Take specific commit from another branch and apply it to your current branch
- Use-cases:
  - Hotfixes which are fixed in another version can be quickly applied
  - Commit used in one feature might be needed in another feature or main/master branch
- Note: Be careful to move your HEAD to the right branch

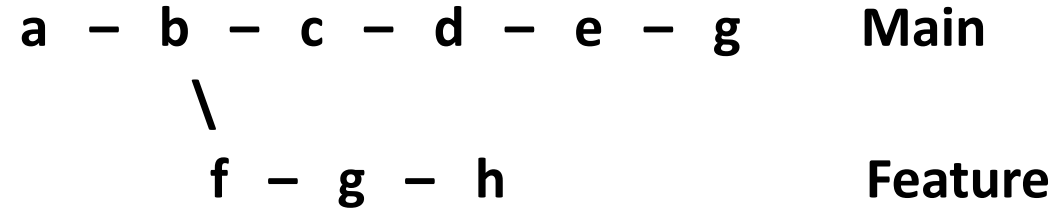
Example: `git checkout main`

`git cherry-pick g` (g is the CommitSHA)

Before execution:



After execution:



What should you take away?

- Understand `git reset`
  - What is the difference to `git checkout`?
  - When can I safely use the „hard“-mode?
  - How do I restore lost commits?
- What does git revert do?
- When and when no to use cherry-picking

End of the Session

- Did you understand all the discussed commands?
  - How to use them
  - When to use which command

- Were there some points of confusion?

- Does the internal structure of git make sense to you?

THANK YOU FOR YOUR ATTENTION!

Stashing

- When you do not want to commit the changes you just made to a branch use git stash to save it for later
- Stashes are saved in a stack
- How it works:
  - git stash save „Message“ → Create new stash with message
  - git stash list → List all available stashes
  - git stash apply stash@{ID} → Apply stash „ID“
  - git stash pop → Applies the first element of your „stash-stack“ and then deletes it
  - git stash drop stash@{ID} → Delete stash „ID“
  - git stash clear → Delete whole stack of stashes