Introduction to Git and GitHub

QLS612 2025

Check if you're ready

☑️ Can you open a bash shell?

- Open a terminal, type `echo $SHELL` and press ENTER.
- The output should be `/bin/bash`

☑️ Do you have git installed?

- In the bash terminal, `git --version` and press ENTER.
- The output should be `git version X` (where the X is the version number)
- *Don't worry if you don't have the exact same version as I do*

☑️ Do you have git configured?

- In the bash terminal, type `git config --list` and press ENTER
- You should see your name and email (and other things that aren't essential to configure)

☑️ Can you open a text editor? E.g.,

- Linux: gedit, nano
- macOS: textedit
- Windows: notepad

☑️ Can you go your GitHub account?

slide by Kendra Oudyk

Introduction

What is version control?

= keeping track of the changes you make in your files

Manual “version control”

Or

Your working directory

File_version-1

File_version-2

File_final-version

File_final-final-version

Your working directory

File_version-1

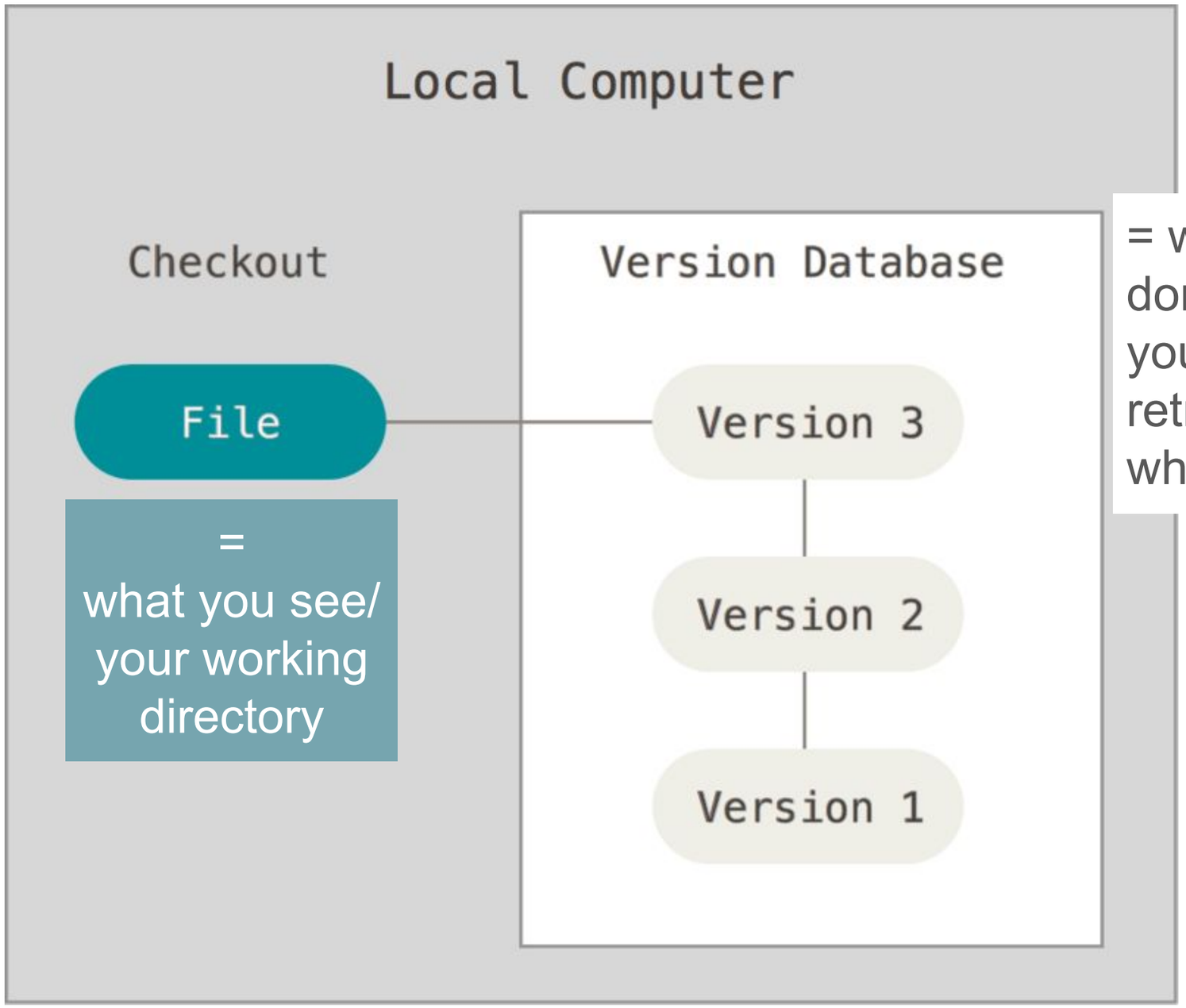
What if the code breaks?

What if the code breaks?

Version control

- keep track of what you changed in the code (and in collaborative settings what each person changed)
- incorporate changes without breaking another module
- or: if it breaks, figure out when and why it broke
- have multiple streams of work for different versions parallel to each other
- Reproducibility!

Version control by Git
or: The basic principle of Git

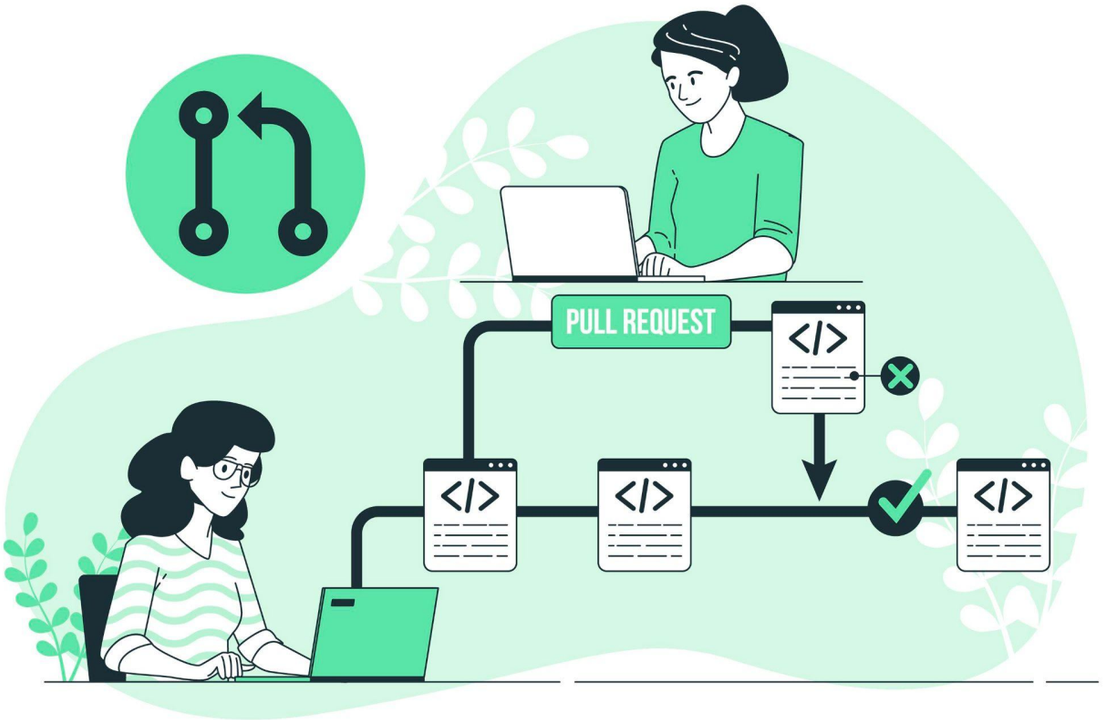


slide by Kendra Oudyk

Software development

modular code development

collaboration



How the versioning works

- git is taking a snapshot of the current stage of your project (*commit)* and saving it in the version database (*git repository*)

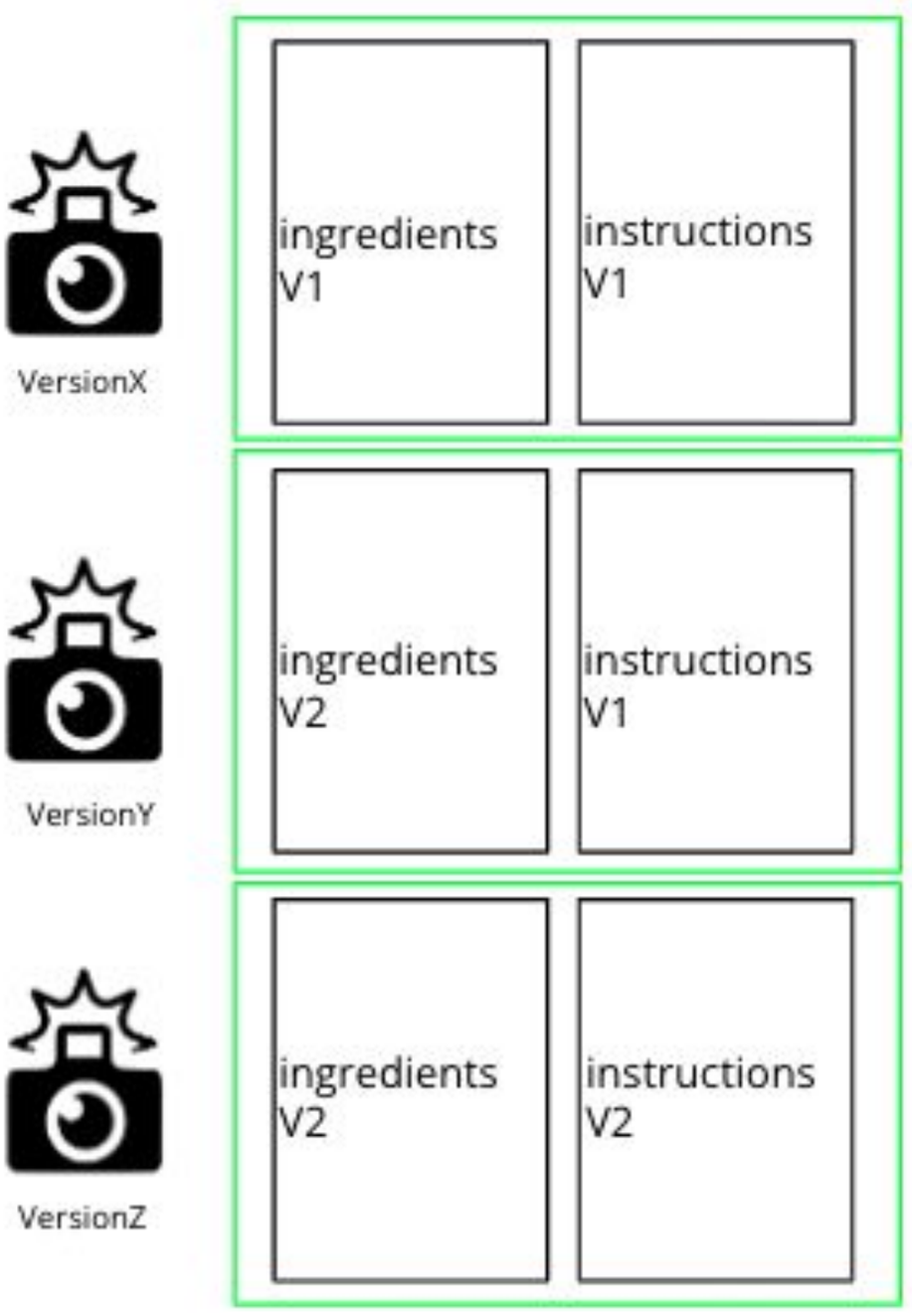
- commits are stand alone versions of the project

- for every commit, git creates a hash which looks like this: 7c35a3ce607a14953f070f0f83b5d74c2296ef93 = file content identifier for git

- all hashes can be found in the commit history and can be used to look at or retrieve an earlier version

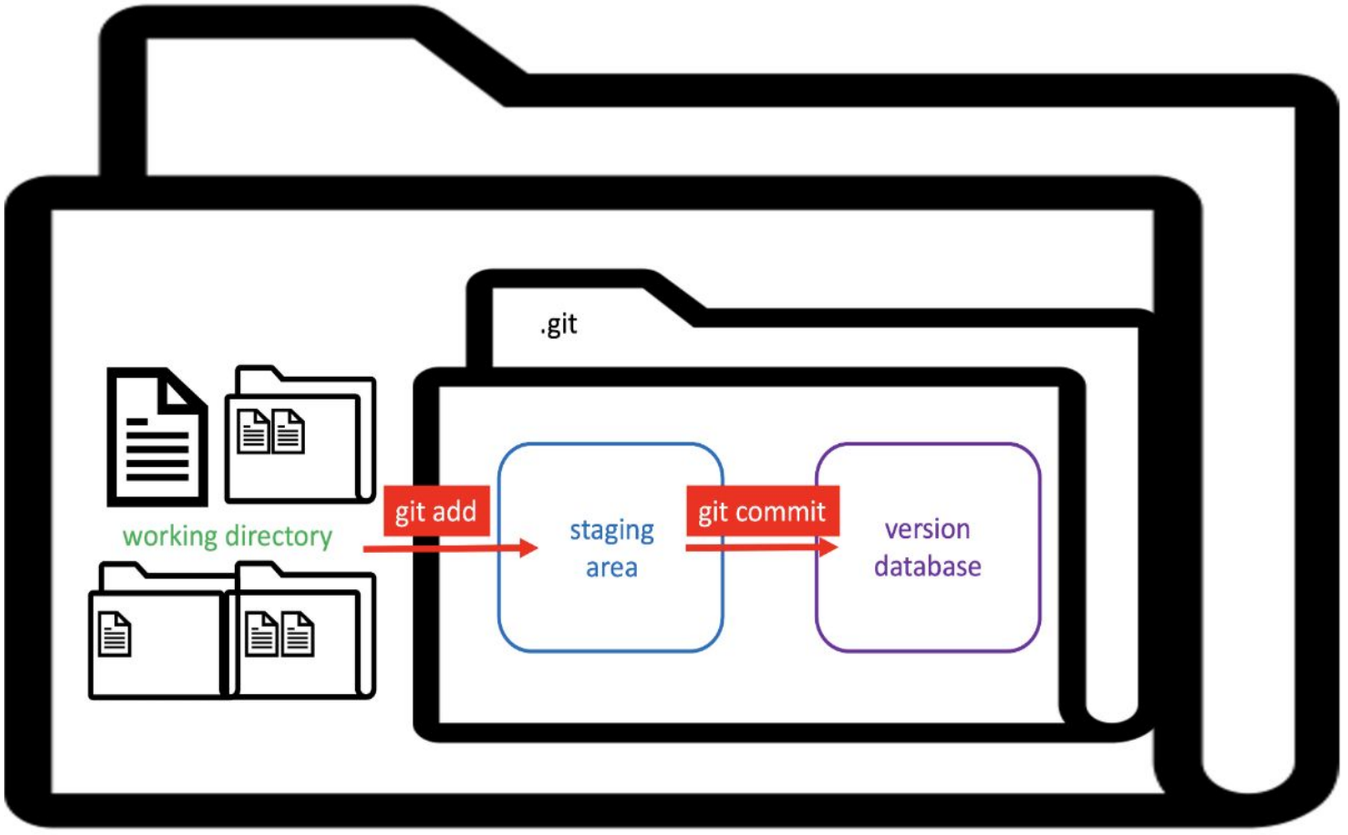
Version Database/ git repository

= hidden .git folder inside your project folder



Staging before committing

= indicating that you want a modified file in your next commit



Basic git workflow. Every change you want to commit needs to be staged first. Only staged changes will be committed in a new version.

Why staging before committing?

- Related vs. unrelated changes
- commit message

You are working on a project where you have to conduct an experiment, analyse the data, and publish a manuscript. You make a change to your stimuli-coding-file by coding a new stimuli shape. You insert this new stimuli shape in your experiment code as well. The changes happened in different files, yet the changes are logically related (make new shape - use new shape). You should commit those changes together, meaning *after* you changed both files.

Why staging before committing?

- Related vs. unrelated changes
- commit message

You are working on a project where you have to conduct an **experiment**, **analyse** the data, and publish a **manuscript**. You make a change to your stimuli-coding-file by coding a new stimuli shape. On the same day, you also make a change in our manuscript (which is awesome, btw), changing the affiliation of one of the co-authors. Those changes are not logically related. You should not put them in the same commit.

3 file states

1. **Modified**
   You made a change to the file

1. **Staged**
   You indicated that you want the modified file in your next snapshot

1. **Committed**
   You took the snapshot

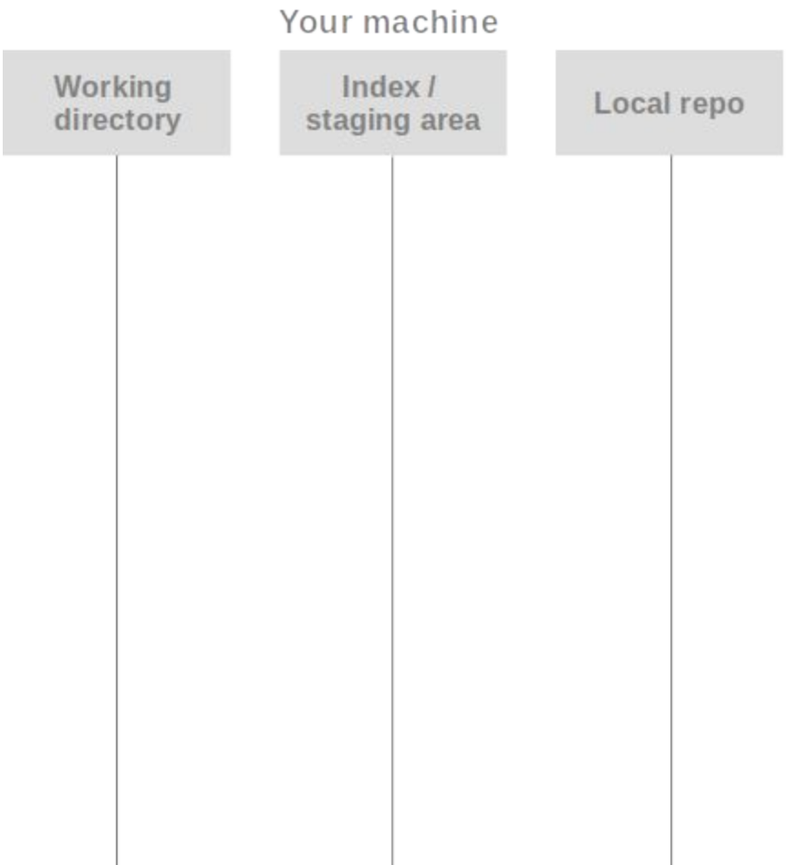
slide by Kendra Oudyk

3 parts of a Git project

1. **Working directory**
   The version of the project that you're working on

2. **Staging area / Index**
   What will be in your next snapshot

3. **Local repository (i.e., `.git/` folder)**
   Metadata and objects that make up the snapshots



slide by Kendra Oudyk

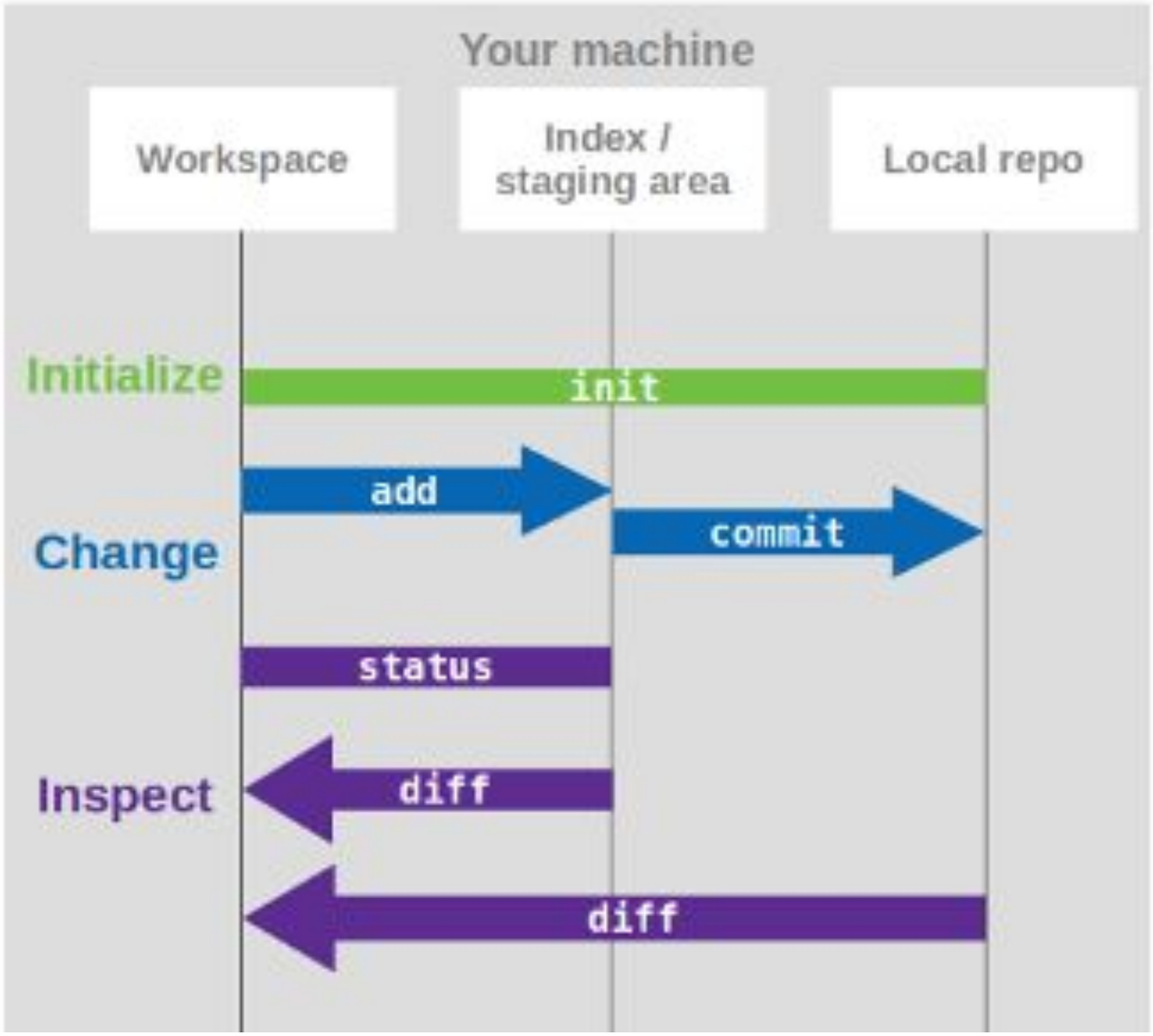
Basic git workflow

Type along!

Git vocabulary

- `git init`: to give git initial control over your files
- `git add`: adds a change from the working directory into the staging area
- `git commit`: saves a snapshot of the current version of your project in the repository
- `git status`: check the status of the files
- `git diff`: let's you inspect differences between versions (in a certain way)
- `git log`: to view the commit history

What we did so far



+ git log

Image by Peer Herholz

git status, diff, and log do not change anything about the status of your git repository

Exercise 1

1) Make a change to one of your files. Add and commit it like we just did.

2) Make another change. Don't add it.

3) Create a new file and write something in it.

4) Do a `git status`. What does it say? Do what it says needs to be done to get a clean working tree.

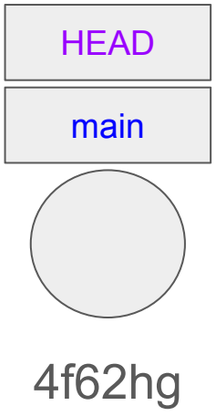
If you type g`it commit -m` instead of git commit, git expects you to write a very short commit message after the m, like `git commit -m "my commit message"`. This short commit message basically replaces the title+message form and the commit message should be short and precise.

Looking at differences

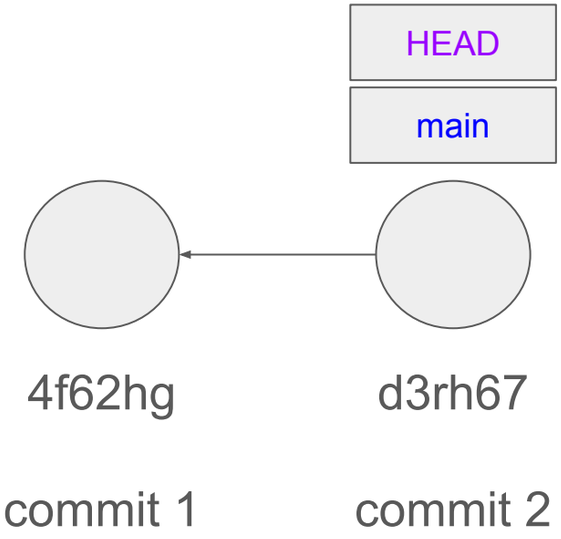
slide by Kendra Oudyk

The commit history

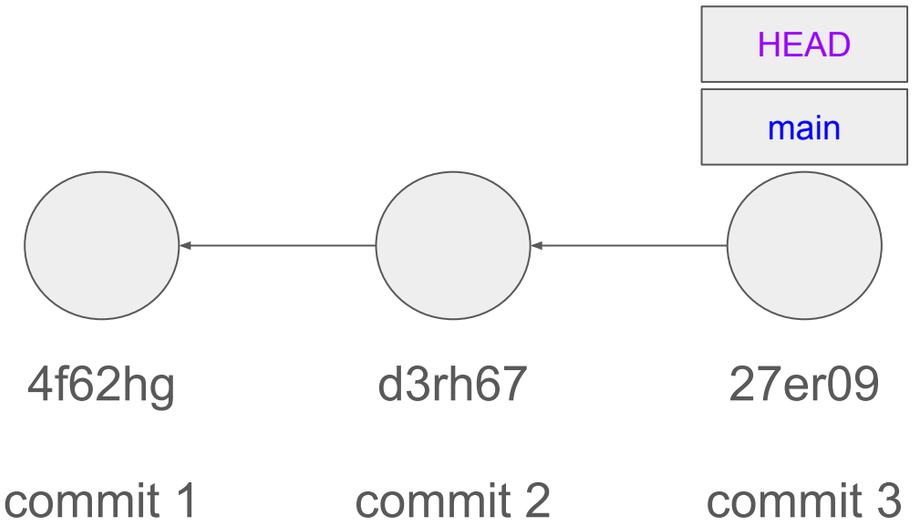
- Every commit has its own commit hash and commit message
- HEAD is a symbolic reference pointing to wherever you are in your commit history. It moves with you, like a shadow
- Every commit is “pointing” to its previous commit
- the branch tag always stays with the latest commit made on its branch



The commit history

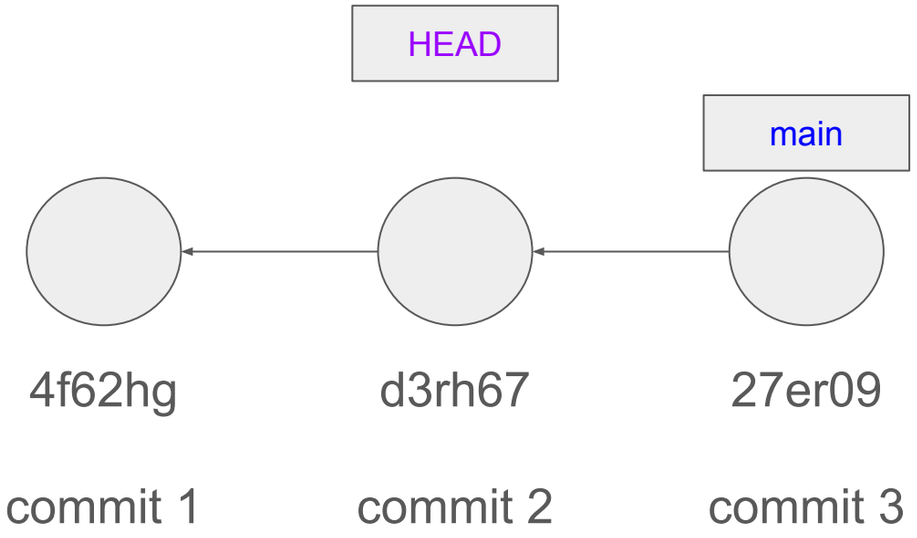


The commit history



Moving around in commit history

`git checkout`



`git checkout HASH` leads you to an earlier version of your project on your **current branch**

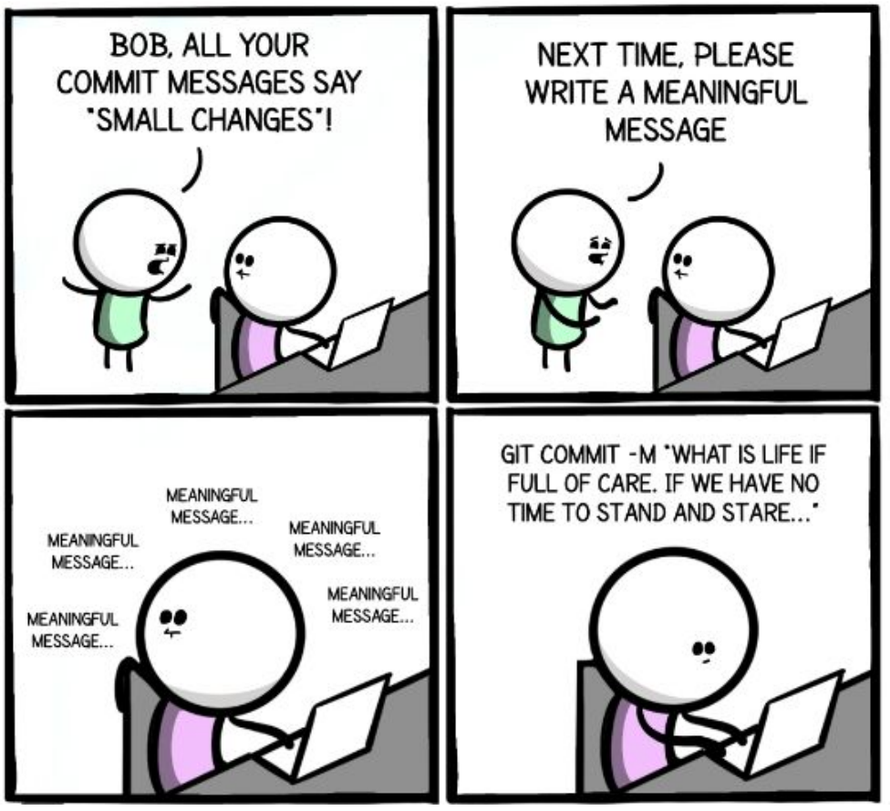
Exercise 2

Do a git diff between your second commit and your second to last commit.

By simply typing `git diff` you will be shown the difference between the current version in the working directory and the last committed version. However, we can also look at differences between two committed versions or the difference to a staged version. For this we need the assigned commit hashes, which we can find out through a `git log`

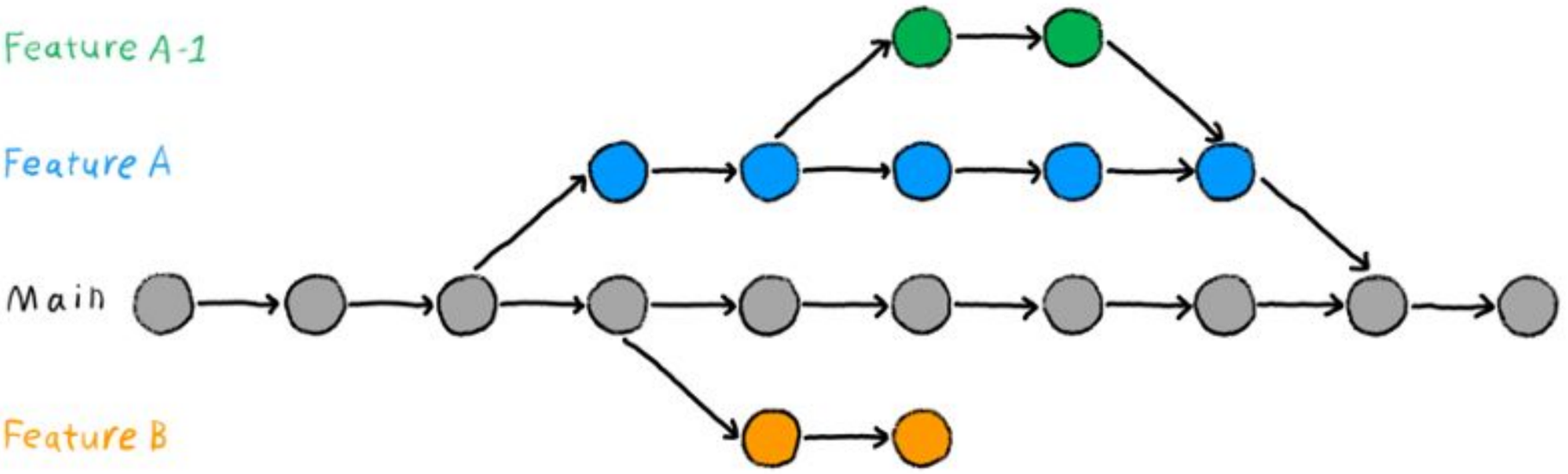
Some comments

- Avoid initializing a git repository within another git repository (“nesting”) → good project folder structure is key!
- Write useful commit messages! This is your conversation with your future self/ your collaborators
- unless you add before you commit those changes are not recorded
- `git status` is your best friend



Branching and merging

Branching and merging



- keep your main branch clean (= is what other people will clone when they want to use your code)
- try out things without messing up main branch
- try out things and merge them when they’re ready
- try out things and decide not to use them
- modular code development

More Git vocabulary

- `git branch or git checkout -b`: create a new branch

- `git checkout`: check out (or “replace”) file in working directory with another staged or committed version **AND move between branches**

- `git merge`: to merge two separate sets of version (branches) into one

Exercise 3: branching

1) Create a new branch by typing `git branch feature`. Check with `git branch` if it worked.

2) Switch to your new `feature` branch by typing `git checkout feature`. You should see *"Switched to branch feature"*. Check if you are really on the new branch by typing `git branch`

3) Make a change to your code, add and commit it.

4) Do a `git log`.

What is different about the git log output compared to the last git log we did?

Why is `HEAD` not with `main`?

Exercise 4: branching

1) Create a new branch from the `feature` branch make a commit on the new branch.

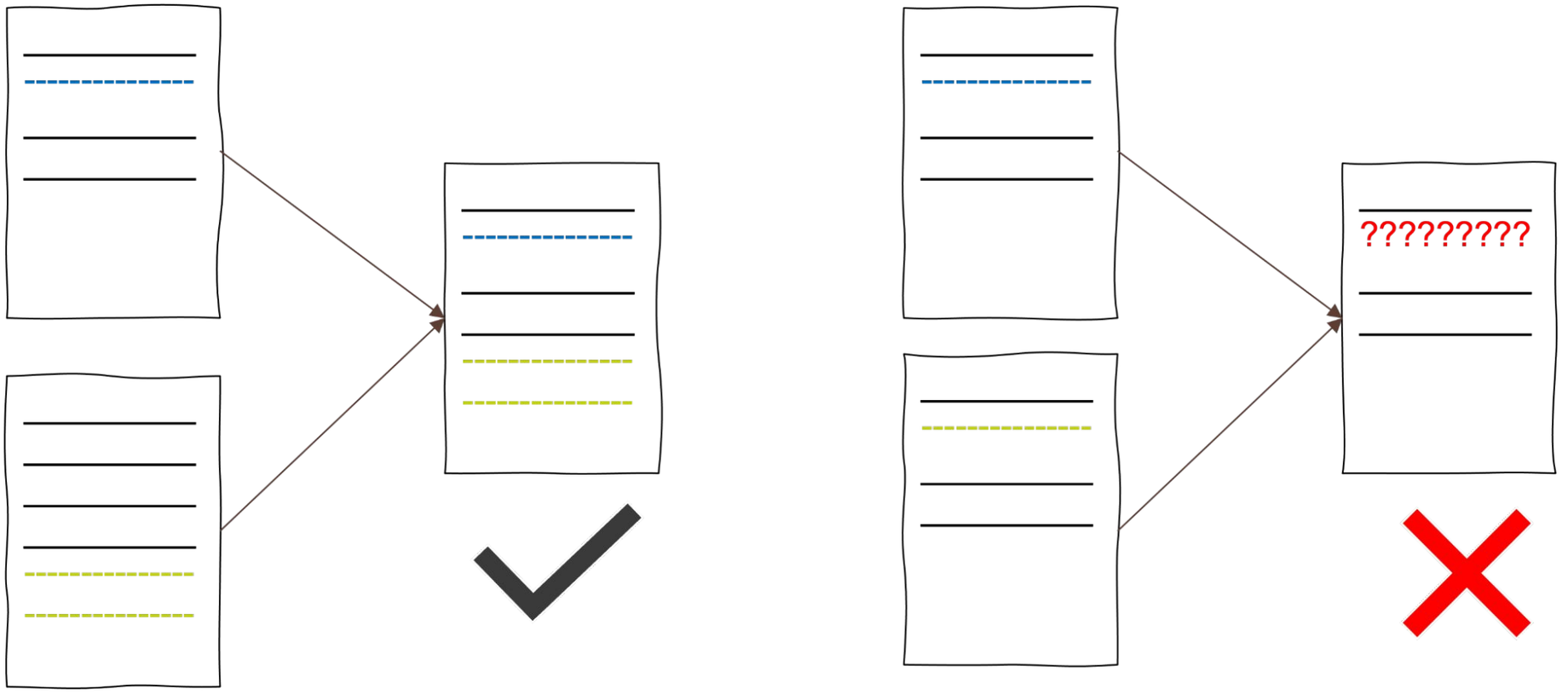
2) Switch to the `main` branch and make a commit.

3) Move to your third commit. Create a branch from this commit.

4) Switch back to the `main` branch

VSCode extension: Git Graph

Merging and merge conflicts



- A merge always refers to separate versions on **two** seperate **branches**

- Merge conflicts are normal

Exercise 5: merge conflict

1) change something on the file currently in your working directory that you know will conflict with your version on the `feature` branch

2) merge the `feature` branch into `main:`

```
git merge feature
```

3) open the file in which the merge conflict appeared. At the place where the merge conflict is you will see this:

```
<<<<<<< HEAD
this is the content of the file on main
...
=======
this is the content of the file on feature
…  >>>>>>> feature
```

You can see that git tells you where the merge conflict starts and what the merge conflict is about. Thanks to VSCode you have some buttons where you can click on to choose which of the two versions you want to keep.

4) solve the merge conflict, add and commit the file

5) do a `git merge` again

Working collaboratively on GitHub

GitHub: centralized/online/remote git repository

- GitHub ≠ Git

- GitHub = your git repository online

- GitHub has its own remote branches = siblings for your local branches

`main` vs. `remotes/origin/main`

GitHub

Everyone working on the repo on GitHub can

- always see the latest version of the project
- see the development stream of the project (aka commit history)
- retrieve the project and make changes and either commit them directly or ask you if you want to incorporate those changes and discuss on the changes (`pull request`)
- comment
- open an `issue`

More vocabulary

- `git push`: synchronize your local repository with the remote repository by pushing changes in the local repository into the remote repository

- `git pull`: synchronize your local repository with the remote repository by pulling changes from the remote repository into the local repository

- `git clone`: this gives you an exact copy of a remote repository on your local computer

- `git fetch`: fetching the latest version from the remote repository into your staging area

Exercise 5: setting up GitHub repo

1) Go to GitHub and log in

2) On the landing page, click on the green button “New”

3) give the project the name, preferably the same name as your local folder has.

4) under "Project URL" select your name.

5) select "public" as visibility level.

6) uncheck the box "Initialize with a README".

7) Follow the instructions given by github for "Push an existing git repo"

After you did all of this, refresh the page.

Exercise 6: working collaboratively

- Find a partner. Choose one of your github repos to be the one you will both work on during the rest of this session.
- Add the person who is not the owner of the repository as a project member with the role "Developer"

From now on the owner of the repository is called `owner` and the collaborator is called `developer`.

`developer` clones the repository:

    - go to the `owners` github repo by searching for their name. Click on the respective repo.
    - click on the `code` button on the upper right
    - copy the "clone via ssh"
    - type in your terminal `git clone repo-link`
- `developer` opens a file, makes a change, adds and commits it
- `developer` pushes the change to the repo by doing `git push repo-link`
- `owner` refreshes github repo to see if the change was updated
- `owner` needs to incorporate the changes from the remote repo into the local repo: `git pull repo-link`
- `owner` opens a file, makes a change, adds and commits it
- `owner` pushes the change to the repo by doing `git push`

Vocabulary explained: `git pull`

- `git pull` means you are pulling the version from the remote origin branch directly into your working directory
- = an automatic merge of remote and local branch

→ updates your git repo AND your working directory in one command.

Vocabulary explained: `git fetch`

- `git fetch` means that git is updating the remote branch in your local git repo called `origin/main` where it contains the latest *remote version* of the project. It is not updating your local branch `main`.
- that's why you have to (or can) merge: your corresponding local branch `main` in your working directory contains a different version of the project than the remote branch `origin/main`
- that's also the reason why you can do git diff
- `git fetch` is helpful if you want to inspect the changes from the remote repository first before incorporating them locally. Because maybe a merge conflict is hiding in the remote version

Summary GitHub workflow

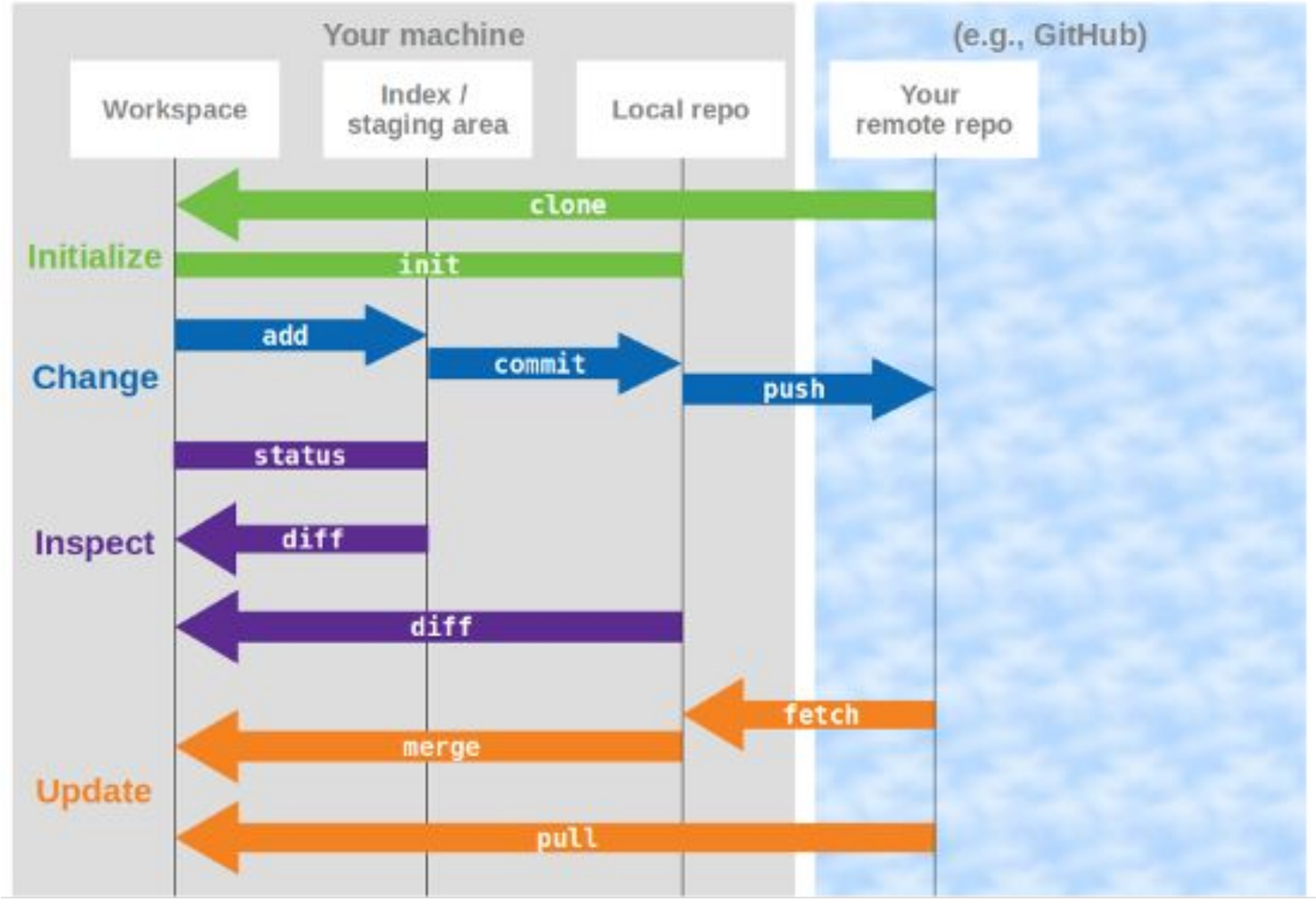


Image by Peer Herholz

Exercise 7: Merge conflict with remote repo

- work with your partner from the previous task again
- `developer` does a `git pull`
- `developer` and `owner` both make a change to the project at the same spot, e.g., modify the same line in the instructions
- `developer` does a `git push` first
- then `owner` does a `git push`
- `owner` should get a message saying:

```
rejected! error: failed to push some refs to 'repo
url'
hint: Updates were rejected because the remote
contains work that you do not have locally. This is
usually caused by another repository pushing to the
same ref. You may want to first integrate the remote
changes (e.g., 'git pull ...') before pushing again.
```

- resolve the merge conflict just how we did it with the local merge conflict

Fork

- making a remote copy of a remote repo
- contributing to the original by opening a pull request
- also: you can set a default on your github repo that everyone has to open a pull request and cannot commit to main directly
- also: needed to contribute to a repo that you don’t have write access to

Exercise 8: Forking and Pull request

- switch roles of `developer` and `owner`
- `developer` forks repo of owner
- `developer` clones their fork
- `developer` makes change and pushes it back to their fork
- `developer` opens a pull request in owner's repo
- `owner` reviews and merges pull request

Comments on working collaboratively

- make use of branches and Pull requests
- make your own fork
- make use of issues and link them to Pull requests

References and recommendations

- YouTube is full of Git/GitLab/GitHub videos for all kinds of levels and features!!! For example: Brainhack Git introduction or GitHub CI

- Git cheat sheet by github

- Atlassian tutorials and cheat sheet

- Troubleshooting: Oh shit git

- Git branching

- Git GUIs

- Full written Git and GitLab Tutorial

- really, just type anything you want to know about Git in YouTube and you'll find a tutorial for it.

Thanks to Kendra Oudyk for some of the slides! You can find here slides here