

# Let's Git together

Alessandro Sciarra

CRC-TR 211 – Software Development Center

PUNCH Young Academy – PUNCH4NFDI

23 May 2024

# Outline of the talk

- 1 A short recap from last time
- 2 Using branches
- 3 Working with remote repositories
- 4 The stashing area
- 5 Bare repositories
- 6 The remaining git commands
- 7 Conclusions

# A short recap from last time

•  $\mathcal{H}$  is a Hilbert space

•  $\mathcal{H}$  is separable

•  $\mathcal{H}$  is reflexive

•  $\mathcal{H}$  is complete

•  $\mathcal{H}$  is normed

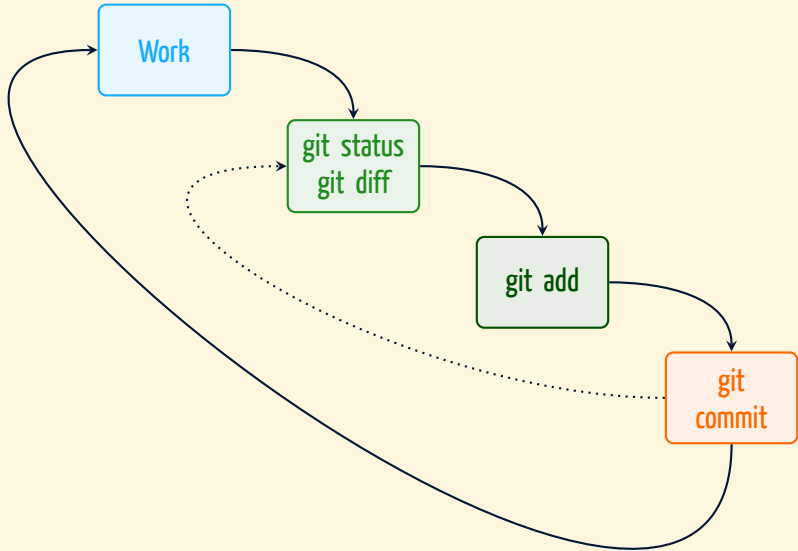
•  $\mathcal{H}$  is linear

•  $\mathcal{H}$  is inner product

•  $\mathcal{H}$  is Banach

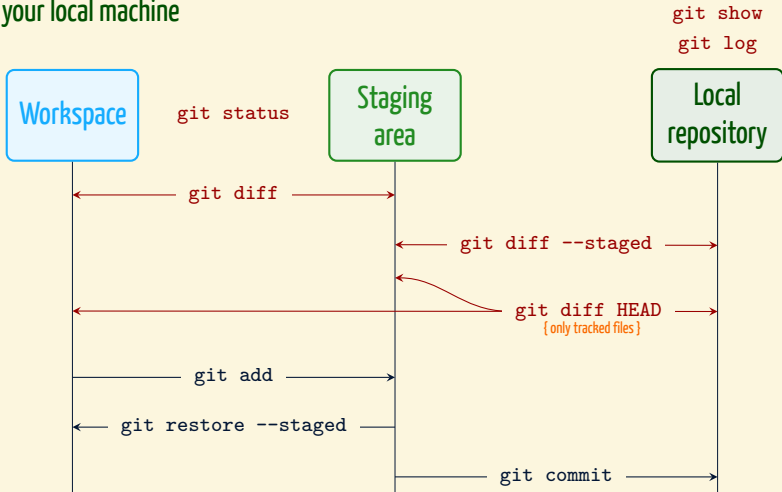
•  $\mathcal{H}$  is separable

# By now, this is how your workflow looks like



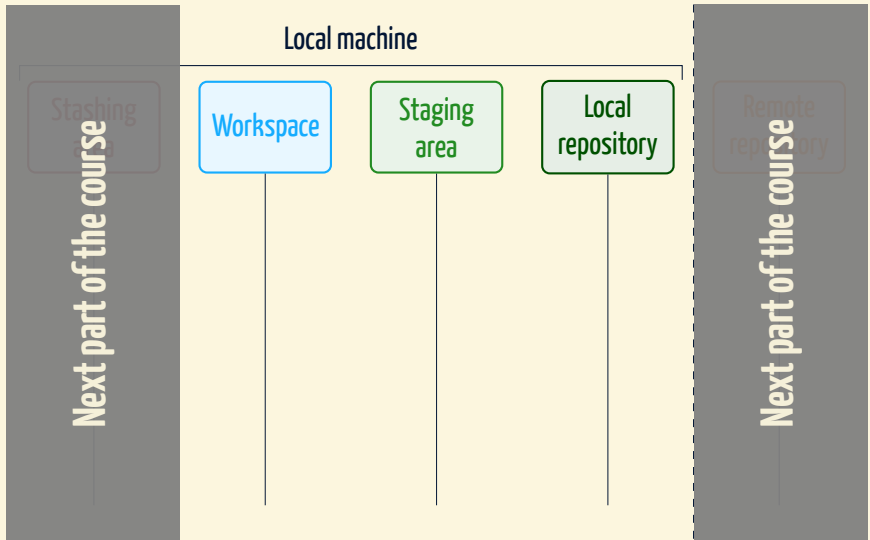
# Our mental picture, so far

On your local machine



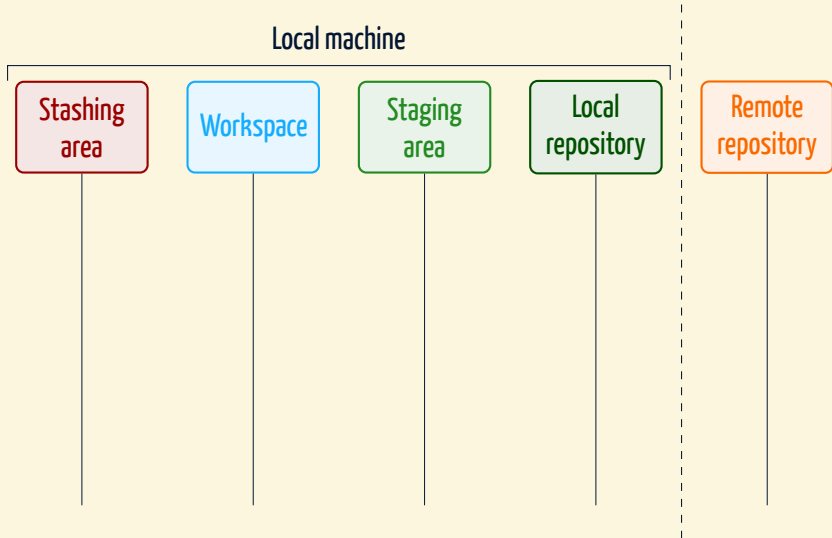
Commands marked in **dark red** do not change anything in the repository!

# The complete correct abstract mental setup



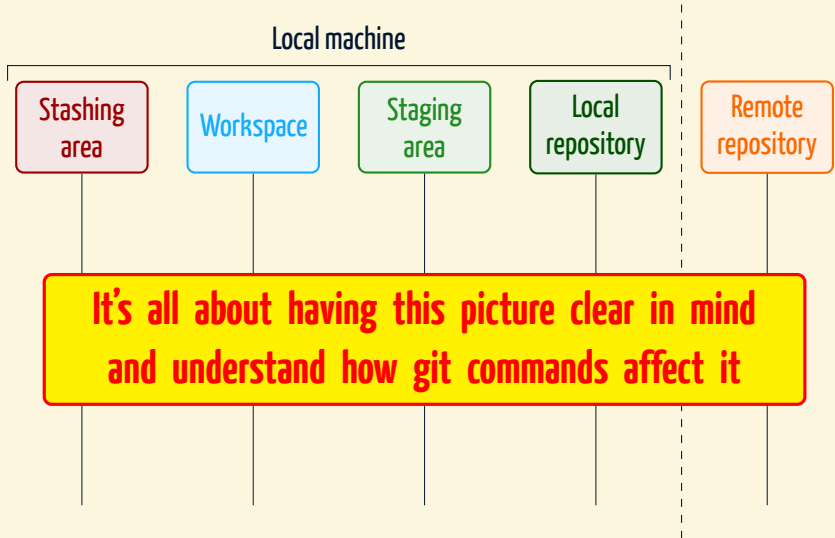
What we explored last time

# The complete correct abstract mental setup



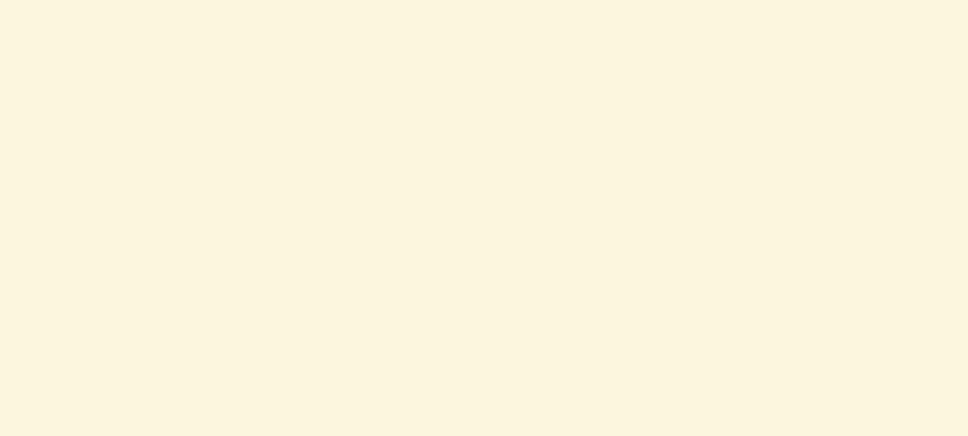
Now we'll complete the picture

# The complete correct abstract mental setup



Now we'll complete the picture






# Git is your friend!

## What happens if I ask Git something not possible?

In many case, when Git fails, it reports to you a detailed error message that is likely to make you understand what has happened. **Make sure you read it!** Sometimes you need to figure out what should be done, but you never get a silent failure.

## Git will hardly make you loose changes

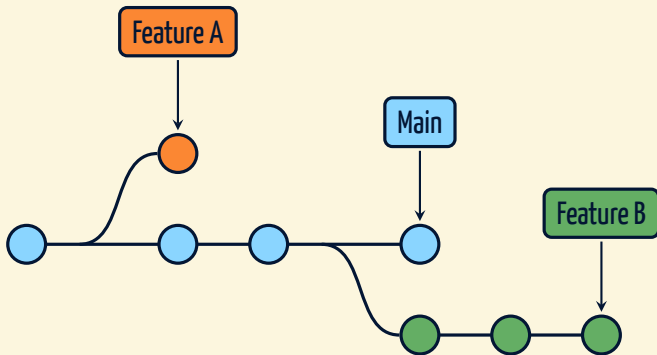
There are only few commands that potentially make you loose changes and usually these require you to force the operation with a command line option. This together with the verbose reporting system allows you to be pretty confident in using and experimenting with Git.

Even in some cases when changes seem lost, `git reflog` or `git fsck` could rescue you.  [Here](#) a nice overview.



# A key feature of Git

- Branches store **different versions of your project**
- Technically just pointers to a commit



# A key feature of Git

- Branches store **different versions of your project**
- Technically just pointers to a commit
- They enable parallel development
  - Implement new features
  - Fix bugs
  - Try out something
  - [...]
- The always existing `main` branch:
  - By default created at initialization
  - Development should be done on other branches
  - Till ~2020 it was called `master`

# Git branch

```
# List all existing local branches
$ git branch
* main
```

```
# Create a new branch
$ git branch new-branch
$ git branch
* main
  new-branch
```

```
# Delete a branch
$ git branch -d new-branch
Deleted branch new-branch (was a45b032).
$ git branch
* main
```

## Git is safe

If a modifications would be lost, Git does not allow you to delete the branch using the `-d` option. Use the `-D` option, instead, to force the deletion.

# Git switch

This will in general change your workspace!

```
# Switching to another branch
$ git branch
* main
  new-branch
$ git switch new-branch
Switched to branch 'new-branch'
$ git branch
main
* new-branch
```

## Git is safe

You may switch branches with uncommitted changes in the work-tree if and only if said switching does not require clobbering those changes.

Before v2.23 of Git (August 2019), the command `git checkout` needed to be used to switch branch.

# Git switch

This will in general change your workspace!

```
# Switching to another branch
$ git branch
* main
  new-branch
$ git switch new-branch
Switched to branch 'new-branch'
$ git branch
main
* new-branch
```

```
# Creating and switching to a new branch at once
$ git switch -c another-branch
Switched to a new branch 'another-branch'
$ git branch
* another-branch
  main
  new-branch

# Pay attention from which
# branch you create a new one!
```

Before v2.23 of Git (August 2019), the command `git checkout` needed to be used to switch branch.



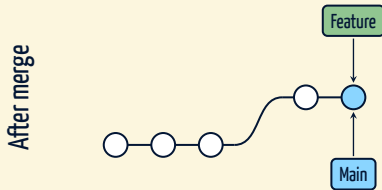
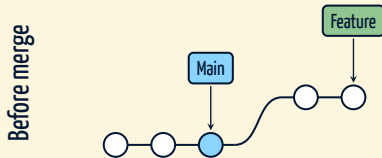
# Merging branches

- To merge means to unify the snapshots of two different branches
- This is automatically done by Git in a clever way
- When Git does not know how to merge the content of some file, it will create a conflict
- If conflicts occur, the merge will not automatically finish
- A merge can be aborted in case of conflicts with the `--abort` option { The content of the repository will be restored to the state before issuing the merge command }
- To fix conflicts, open and manually adjust files where Git failed

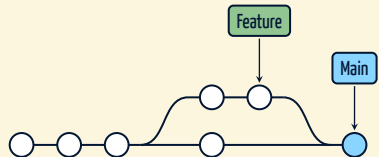
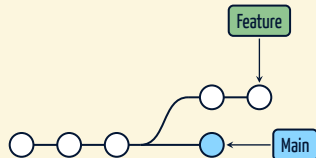
Git is safe, conflicts are not a bad thing!

# Different types of merge

## Fast-forward merge



## Three-way merge



The **three**-way nomenclature comes from the fact that Git uses three commits to generate the merge commit: the two branch tips and their common ancestor.

# Git merge: How does it work?

- 1 If possible, Git performs a fast-forward merge
- 2 Otherwise a three-way merge is done and a new commit created

Be sure to be on the correct branch!

```
git merge <source-branch>
```

It incorporates changes from the specified branch into the present branch!

```
# It is possible to force a three-way merge:  
$ git merge --no-ff <source-branch>
```

# Merge conflicts: Fixing procedure

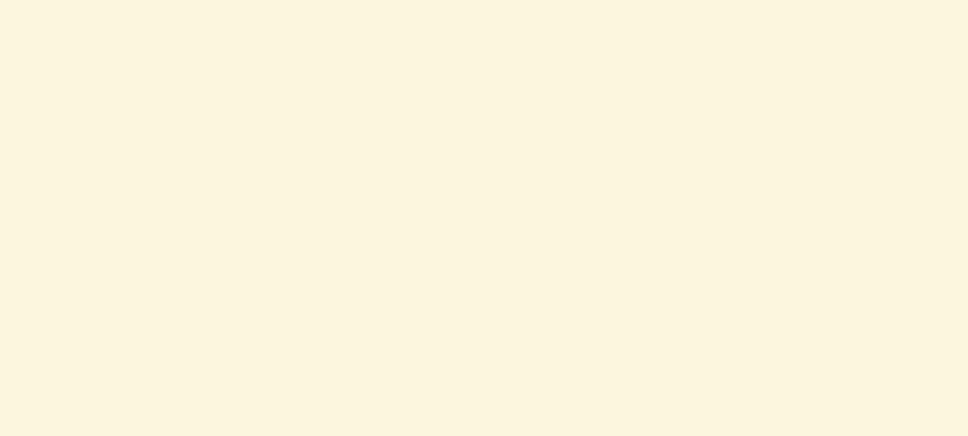
```
# A general example
$ git merge <branch_name>
Auto-merging <file>
CONFLICT (content): Merge conflict in <file>
Automatic merge failed; fix conflicts and then commit the result.
```

- 1 Run `git status` to see **unmerged paths**
- 2 Find problematic hunks in files that contain conflicts
  - ↳ Look for delimiters in the files: <<<<<<, =====, >>>>>>
- 3 Remove delimiters and adjust content
- 4 Check the project works (e.g. compile, run tests)
- 5 `git add` the files with fixed conflicts
- 6 Commit added files
  - ↳ Git propose you an auto-generated commit message





The Git prompt I use in my **Bash** terminal is a customization of [🔗this one!](#)

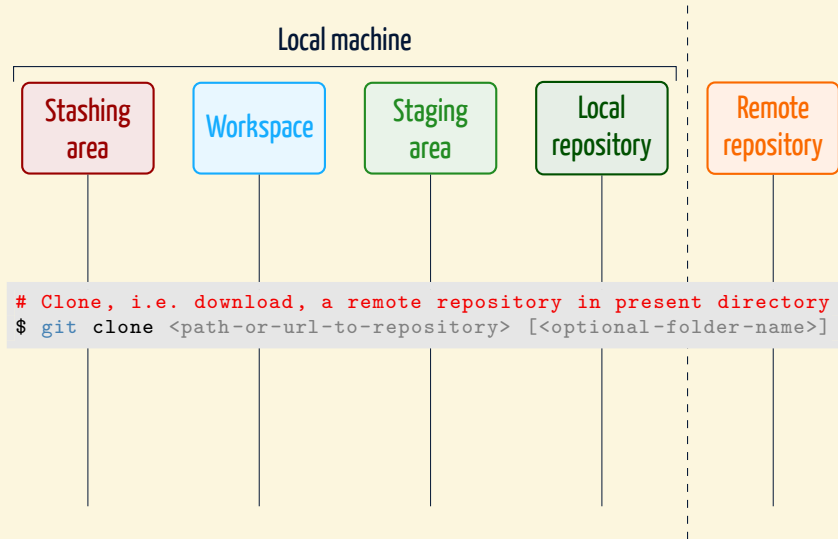


# Cloning a remote repository





# Cloning a remote repository



# Cloning a remote repository

```
# Clone the repository of this course
$ ls
$ git clone git@github.com:AxelKrypton/Git-crash-course.git
Cloning into 'Git-crash-course'...
remote: Enumerating objects: 82, done.
remote: Counting objects: 100% (66/66), done.
remote: Compressing objects: 100% (47/47), done.
remote: Total 82 (delta 26), reused 52 (delta 18), pack-reused 16
Receiving objects: 100% (82/82), 4.47 MiB | 4.49 MiB/s, done.
Resolving deltas: 100% (29/29), done.
$ ls
Git-crash-course
```

The local repository is aware of the remote one!

# Git remote

In the local repository, the remote one is (by default) referred as **origin**

```
# Check out the remote information
$ cd Git-crash-course
$ git remote
origin
$ git remote -v
origin      git@github.com:AxelKrypton/Git-crash-course.git (fetch)
origin      git@github.com:AxelKrypton/Git-crash-course.git (push)
$ git remote rename origin GitHub
$ git remote
GitHub
# A repository can have multiple remote locations
$ git remote add MyServer <url-to-new-remote>
$ git remote
GitHub
MyServer
```

# First interactions with the remote repository

```
$ git branch -r
origin/HEAD -> origin/main
origin/main
origin/experiment
$ git branch -a
* main
remotes/origin/HEAD -> origin/main
remotes/origin/main
remotes/origin/experiment
```

```
# Switch to a new branch that mirrors the state of a remote one
$ git switch experiment
Branch 'experiment' set up to track remote branch 'experiment'
from 'origin'.
Switched to a new branch 'experiment'
$ git branch -vv
* experiment      a1d62e63 [origin/experiment] last-commit-message
main             a1d62e63 [origin/main] last-commit-message
```

We'll come back to the idea of tracking in a moment!

# Fetching and pulling

When collaborating in a project, the remote repository will in general change because of other people's work

```
$ git fetch <remote-name>
```

- Information about the remote repository (e.g. branches) can be updated by fetching from a remote
- **Fetching does not change the local workspace!**

# Fetching and pulling

When collaborating in a project, the remote repository will in general change because of other people's work

```
$ git pull <remote-name> <remote-branch-name>
```

- Pulling instead is updating both the information about the remote repository and the local workspace
- Git pull is actual a shortcut to do a fetch followed by a merge with a remote branch

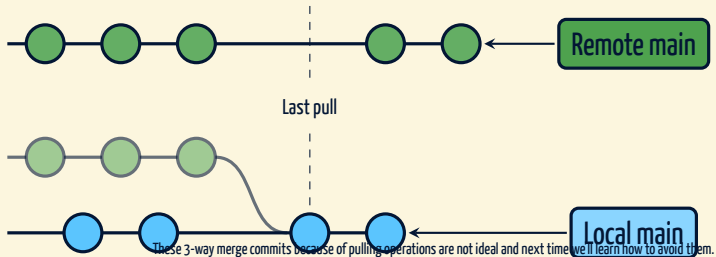
# Fetching and pulling: Examples

```
# If there is only one remote, you can omit it
$ git fetch origin
a1e8fb5..45e66a4 main -> origin/main
a1e8fb5..9e8ab1c develop -> origin/develop
* [new branch] some-feature -> origin/some-feature
# To remove locally references to remote deleted branches:
$ git fetch --prune
From github.com:AxelKrypton/Git-crash-course
- [deleted]                (none)        -> origin/experiment
```

```
# Be sure to be on the correct branch before pulling
$ git branch
* main
$ git pull origin main
From github.com:AxelKrypton/Git-crash-course
* branch                main            -> FETCH_HEAD
Already up to date.
```

# Fetching and pulling: Examples

If you created commits on the present branch, pulling it from remote will perform a merge and, if this is not a fast-forward merge, the editor to make a commit with an auto-generated message will be displayed to you. **Conflicts might occur as well.**





# Fetching and pulling: Examples

If you created commits on the present branch, pulling it from remote will perform a merge and, if this is not a fast-forward merge, the editor to make a commit with an auto-generated message will be displayed to you. **Conflicts might occur as well.**

```
$ git log --oneline
a45b032 (HEAD -> main) Some work done locally on main
6e5ea4b (origin/main, origin/HEAD) Last commit pulled
236d4af Previous history
# If I now pull and some new commit exists remotely after
# 6e5ea4b, a 3-way merge occurs and the editor will open
$ git pull origin main
[...]
```

These 3-way merge commits because of pulling operations are not ideal and next time we'll learn how to avoid them.

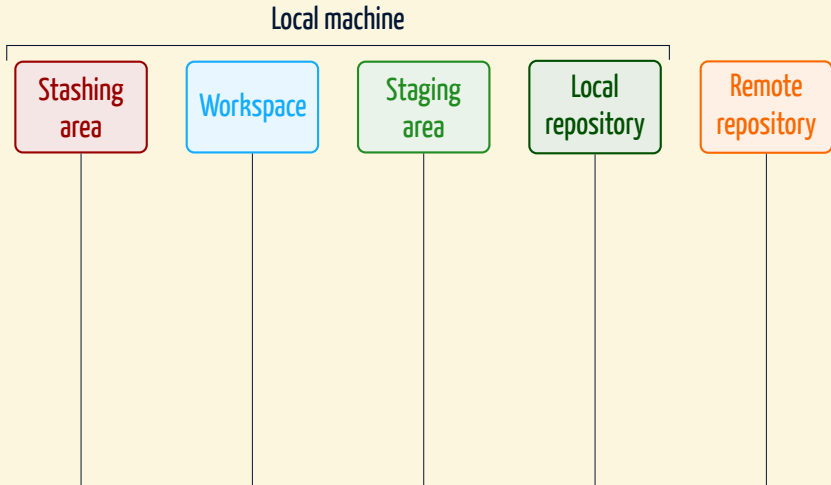
# Pushing your own work

- To push means to make the changes done in the local repository available in the remote one, i.e. to update a remote branch with a local one
- Only changes that are committed are pushed
- If the remote and the local history diverge (i.e. you forgot to pull before committing), the push operation will be rejected

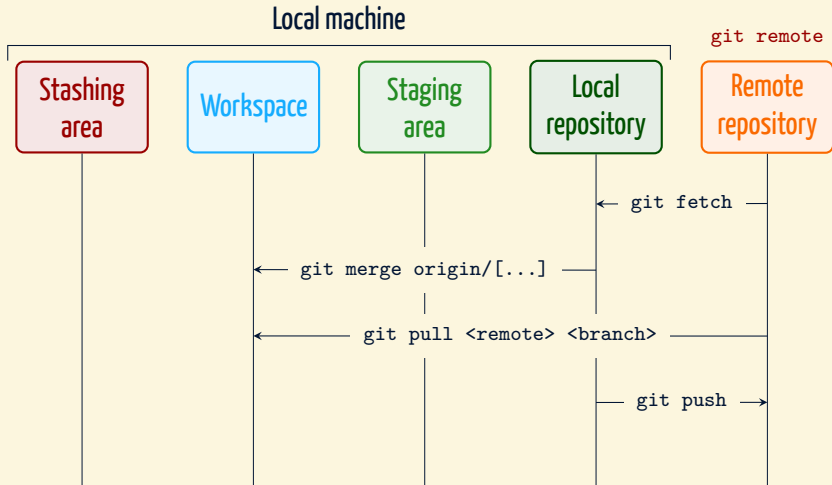
Push before saying to your collaborators you did some changes!

```
$ git push origin main  
[...]  
# To delete remote branches 'git push' is also used:  
$ git push <remote-name> --delete <branch_name>
```

# How does this fit into our mental picture?



# How does this fit into our mental picture?



Keep in mind that it is very tough (if not impossible) to undo these commands.

# Few words about tracking branches

## A convenient feature

It is possible to make a default association between a local and a remote branch which is used for pull and push operations!

```
# Show all existing branches in verbose mode
```

```
$ git branch -a -vv
```

```
exp1                6e5ea4b Commit msg
exp2                6e5ea4b Commit msg
* main              6e5ea4b [origin/main] Commit msg
remotes/origin/HEAD -> origin/main
remotes/origin/main 6e5ea4b Commit msg
remotes/origin/exp1 6e5ea4b Commit msg
```

```
# First possibility (when remote branch already exists)
```

```
$ git switch exp1
```

```
Switched to branch 'exp1'
```

```
$ git branch --set-upstream-to=origin/exp1
```

```
Branch 'exp1' set up to track remote branch 'exp1' from 'origin'.
```

# Few words about tracking branches

## A convenient feature

It is possible to make a default association between a local and a remote branch which is used for pull and push operations!

```
# Second possibility (even without remote branch, yet)
$ git switch exp2
Switched to branch 'exp2'
$ git push -u origin exp2
[...]
To github.com:AxelKrypton/Git-crash-course.git
* [new branch]          exp2 -> exp2
Branch 'exp2' set up to track remote branch 'exp2' from 'origin'.
```

```
# Check branch tracking associations
$ git branch -vv
exp1          6e5ea4b [origin/exp1] Commit msg
* exp2        6e5ea4b [origin/exp2] Commit msg
main          6e5ea4b [origin/main] Commit msg
```

# Few words about tracking branches

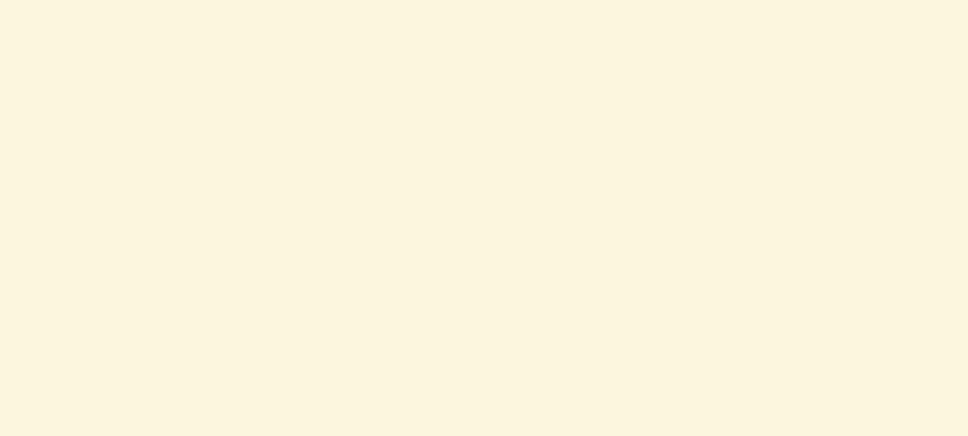
## A convenient feature

It is possible to make a default association between a local and a remote branch which is used for pull and push operations!

## Between tracking branches

It is then possible to simply use **git pull** and **git push** commands!

```
# Check branch tracking associations
$ git branch -vv
  exp1                6e5ea4b [origin/exp1] Commit msg
* exp2                6e5ea4b [origin/exp2] Commit msg
  main                6e5ea4b [origin/main] Commit msg
```





# What if I want neither to commit nor to restore?

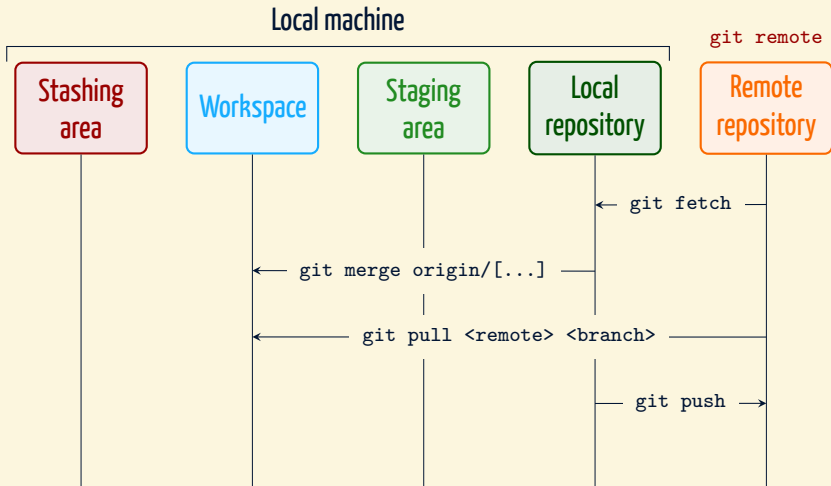
## From Git documentation

Often, when you've been working on part of your project, things are in a messy state and you want to switch branches for a bit to work on something else. The problem is, you don't want to do a commit of half-done work just so you can get back to this point later. The answer to this issue is the **git stash** command.

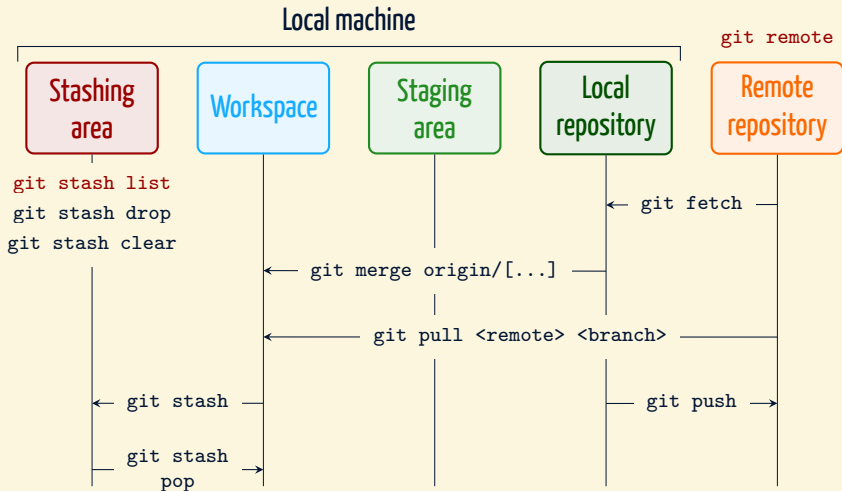
- **Tracked files and staged changes** can be moved to the **stashing area**
- The stashing area is a stack\* of unfinished changes to be used later (or discarded) even on different branches
- From 2018 on it is possible to stash only some files/changes

\* Although people still refer to the stashing area as a stack, the last-in-first-out principle is not anymore a must and item in the middle of the stack can be used.

# Let's complete our mental picture



# Let's complete our mental picture



This sketch together with that of last time will help you daily!

# A taste about stashing

```
$ git log -n1 --oneline
b3092cc (HEAD -> main, origin/main, origin/HEAD) Last commit
$ git status -s
M Part_2/Git-together.pdf # Staged for commit
M Part_2/Git-together.tex # Modified but not staged
```

```
$ git stash list
stash@{0}: WIP on exp1: a3edd5f Epic fail
# Stash present work, 'git stash' defaults to 'git stash push'
$ git stash
Saved working directory and index state WIP on main: b3092cc [...]
$ git stash list
stash@{0}: WIP on main: b3092cc Last commit
stash@{1}: WIP on exp1: a3edd5f Epic fail
```

```
# Let's get back our work (in general conflicts are possible)
$ git stash pop
[...] # <- new status of the repository
Dropped refs/stash@{0} (25d8510240cdb562be3d3a7bd22be28ffa5a29e3)
# If conflicts occur, no stash is dropped
# => use 'git stash drop' after having resolved them
```

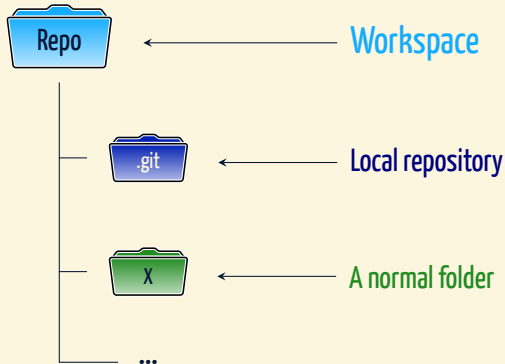
Using apply instead of pop the entry in the stash list remains untouched.

You can also apply, pop, drop, etc. any stash in the list referring it as "stash@{n}" or simply as n (from Git v2.11).



# There are two types of repositories

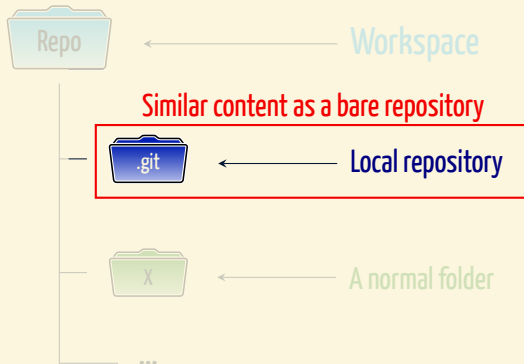
## 1 Non-bare repositories: Meant for working



# There are two types of repositories

- 1 **Non-bare repositories:** Meant for working
- 2 **Bare repositories:** Meant for sharing

↳ a repository that doesn't contain a working directory



# How to create and use bare repositories?

- By default it is not possible to push to a non-bare repository
- If you create a new repository in the browser on a platform like e.g. GitHub, you will not notice it, but a bare one will be created
- Bare repositories are meant to serve as remote
  - ↳ i.e. to pull from and push to

## 1 Initialize a bare empty repository, clone it, work and push to it

```
$ mkdir <bare-repository>.git # .git is often used as suffix
$ git init --bare <bare-repository>.git
Initialised empty Git repository <bare-repository>.git
$ git clone <bare-repository>.git <non-bare-repo>
Cloning into '<non-bare-repo>'...
warning: You appear to have cloned an empty repository.
done.
$ cd <non-bare-repo>
# Work as usual and push when needed.
```

You can put your bare repository e.g. on a server or an external HDD (to clone on different machine without network).



# How to create and use bare repositories?

## 2 Initialize a bare empty repository and push an existing one to it

```
$ mkdir <bare-repository>.git
$ git init --bare <bare-repository>.git
Initialised empty Git repository <bare-repository>.git
$ cd <existent-non-bare-repository>
$ git remote add origin /path/to/<bare-repository>.git
$ git push -u origin main # <- From local desired branch
```

You can put your bare repository e.g. on a server or an external HDD (to clone on different machine without network).

# How to create and use bare repositories?

## 2 Initialize a bare empty repository and push an existing one to it

```
$ mkdir <bare-repository>.git
$ git init --bare <bare-repository>.git
Initialised empty Git repository <bare-repository>.git
$ cd <existent-non-bare-repository>
$ git remote add origin /path/to/<bare-repository>.git
$ git push -u origin main # <- From local desired branch
```

## 3 Create a bare version of an existing non-bare one

```
$ git clone --bare <non-bare-repository> <bare-repository>.git
Cloning into bare repository '<bare-repository>.git'...
done.
$ cd <non-bare-repository>
$ git remote add origin /path/to/<bare-repository>.git
```

You can put your bare repository e.g. on a server or an external HDD (to clone on different machine without network).

# How to create and use bare repositories?

## 2 Initialize a bare empty repository and push an existing one to it

```
$ mkdir <bare-repository>.git
$ git init --bare <bare-repository>.git
Initialised empty Git repository <bare-repository>.git
$ cd <existent-non-bare-repository>
$ git remote add origin /path/to/<bare-repository>.git
$ git push -u origin main # <- From local desired branch
```

## 3 Create a bare version of an existing non-bare one

```
$ git clone --bare <non-bare-repository> <bare-repository>.git
Cloning into bare repository '<bare-repository>.git'...
done.
$ cd <non-bare-repository>
$ git remote add origin /path/to/<bare-repository>.git
```

Pick your favourite way

Probably options 2 and 3 are handier.

You can put your bare repository e.g. on a server or an external HDD (to clone on different machine without network).



# You can now explore the rest alone!

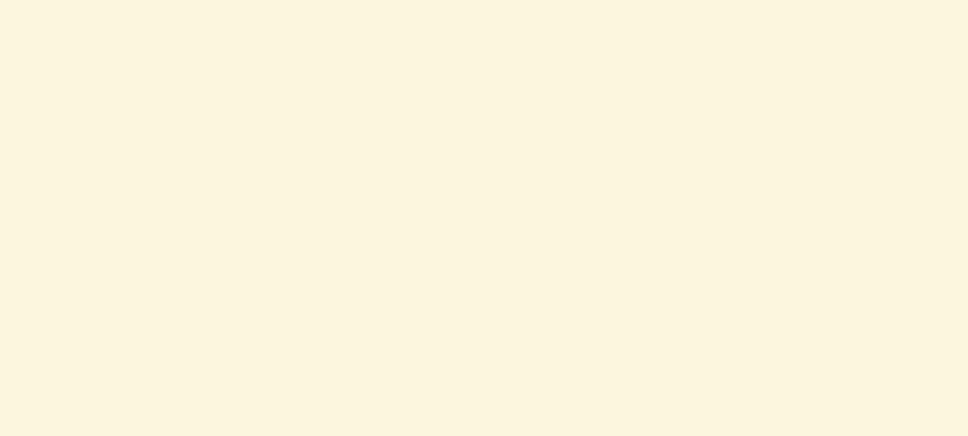
<b>git mv</b>	Move or rename a file, a directory, or a symlink
<b>git restore</b>	Restore working tree files { we mentioned it already }
<b>git rm</b>	Remove files from the working tree and from the index
<b>git bisect</b>	Use binary search to find the commit that introduced a bug
<b>git grep</b>	Print lines matching a pattern
<b>git rebase</b>	Reapply commits on top of another base tip
<b>git reset</b>	Reset current HEAD to the specified state
<b>git tag</b>	Create, list, delete or verify a tag object signed with GPG
<b>git [...]</b>	Few more technical commands

# You can now explore the rest alone!

<b>git mv</b>	Move or rename a file, a directory, or a symlink
<b>git restore</b>	Restore working tree files { we mentioned it already }
<b>git rm</b>	Remove files from the working tree and from the index
<b>git bisect</b>	Use binary search to find the commit that introduced a bug
<b>git grep</b>	Print lines matching a pattern
<b>git rebase</b>	Reapply commits on top of another base tip
<b>git reset</b>	Reset current HEAD to the specified state
<b>git tag</b>	Create, list, delete or verify a tag object signed with GPG
<b>git [...]</b>	Few more technical commands

Wait for it!

In the next and last part of the course we'll learn to rebase and tag









# That's **ALMOST** all folks

- 1 Start using Git. **Now.** Not tomorrow or next week, today!  
→ Repeat what done on these slides

# That's **ALMOST** all folks


- 1 Start using Git. **Now.** Not tomorrow or next week, today!  
→ Repeat what done on these slides
- 2 Was anything unclear? Did you get stuck at some point?  
→ Drop me  an email

# That's **ALMOST** all folks

- 1 Start using Git. **Now.** Not tomorrow or next week, today!  
→ Repeat what done on these slides
- 2 Was anything unclear? Did you get stuck at some point?  
→ Drop me  an email
- 3 You can use Git to work in a very professional way!  
→ Attend next part: «Git in real life»

# That's **ALMOST** all folks

Thank you!

- 1 Start using Git. **Now.** Not tomorrow or next week, today!  
→ Repeat what done on these slides
- 2 Was anything unclear? Did you get stuck at some point?  
→ Drop me  an email
- 3 You can use Git to work in a very professional way!  
→ Attend next part: «Git in real life»

# Live exercise

# Let's collaborate!

- 1 Clone the exercise repository from Redmine

↳ `https://<USERNAME>@git.physik.uni-bielefeld.de/git-together`

- 2 Create and switch to a new branch

- 3 Create a new file, e.g. via

```
$ echo "$(whoami): $(date)" > "$(whoami).txt"
```

and add your name to the AUTHORS file

- 4 Commit your changes to the repository and push your branch
- 5 Let your local branch track the remote one
- 6 Check your work in the browser
- 7 Be sure the main branch is up-to-date and merge your branch into it\*
- 8 Pull and push it to the remote
- 9 Check that your changes appear as expected on `master` in the browser and, if so, remove your branch locally and remotely