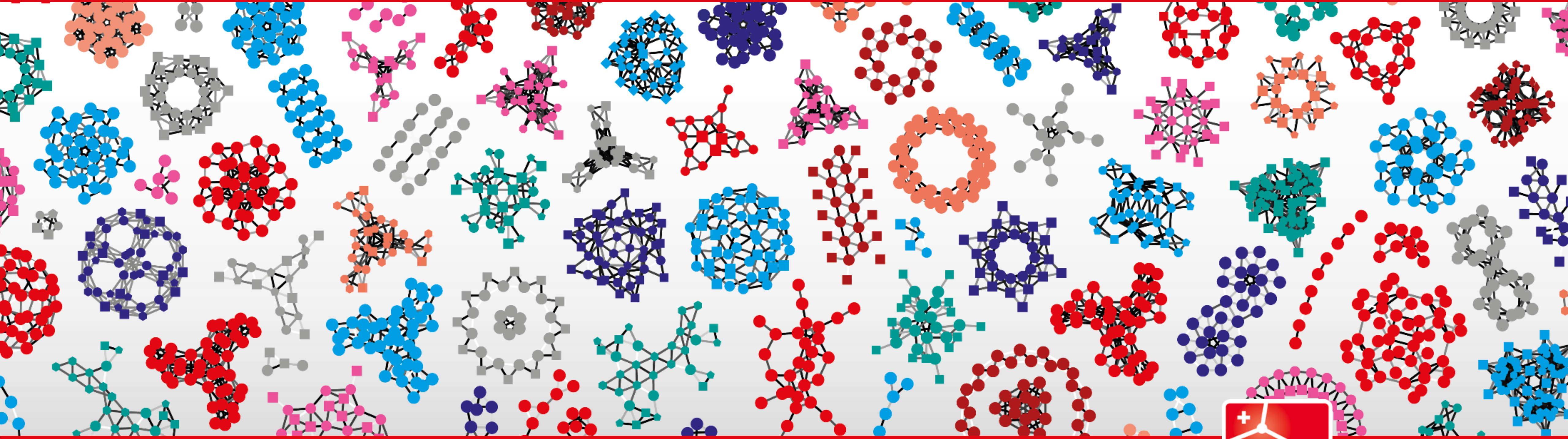


Version control with Git – advanced topics

Swiss Institute of Bioinformatics

www.sib.swiss

Robin Engler
Vassilios Ioannidis

Lausanne, 14-16.03.2022

Course outline

- **Review / Refresher:** quick review of basic commands.
- **Rewriting history**: interactive rebase, git reset and commit amending.
- **Detached HEAD** state explained.
- **The Git stash:** Git’s “cut and paste” functionality.
- **Git tags**: label important commits.

Optional Git extensions (these can be useful for specific applications).

- **Git submodules:** "symlink" Git repos.
- **Git LFS:** large file storage.

Course resources

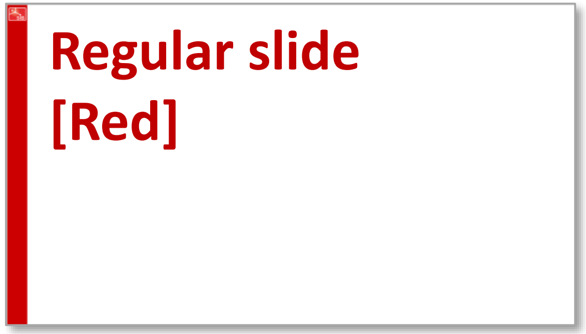
Course home page: slides, exercises, exercise solutions (available at end of day), command summary (cheat sheet), feedback.

Google doc: ask questions.

Questions: feel free to interrupt at anytime to ask questions, or use the Google doc.

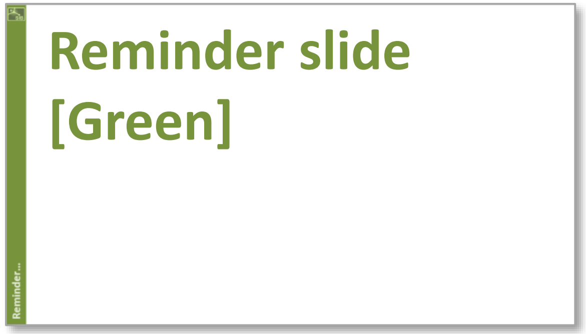
Course slides

- 3 categories of slides:



Regular slide
[Red]

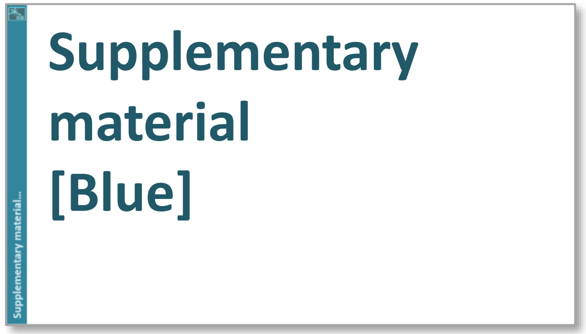
Slide covered in detail during the course.



Reminder slide
[Green]

Reminder…

Material we assume you know.
Covered quickly during the course.



Supplementary material
[Blue]

Supplementary material…

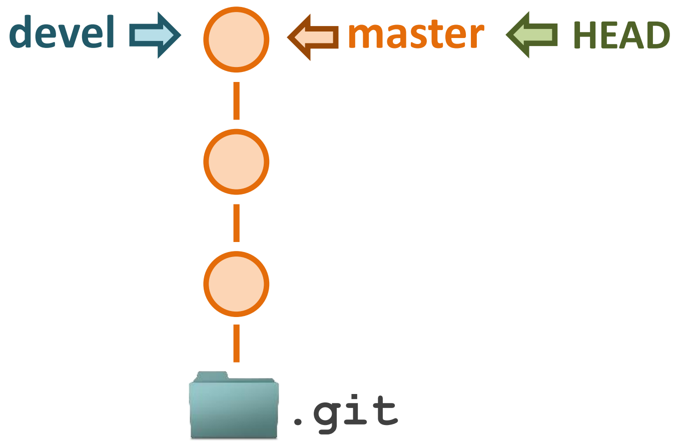
Material available for your interest, to read on your own.
Not formally covered in the course.
We are of course happy to discuss it with you if you have questions.

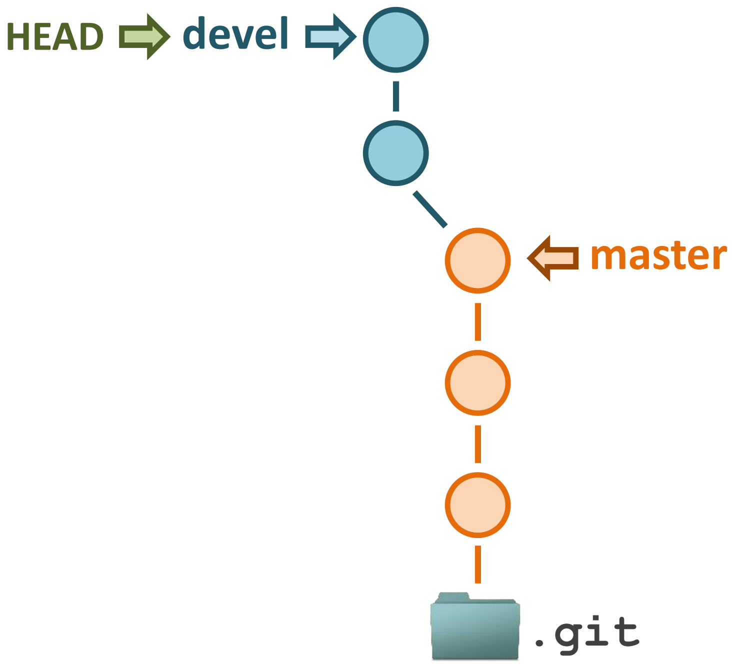
review / refresher

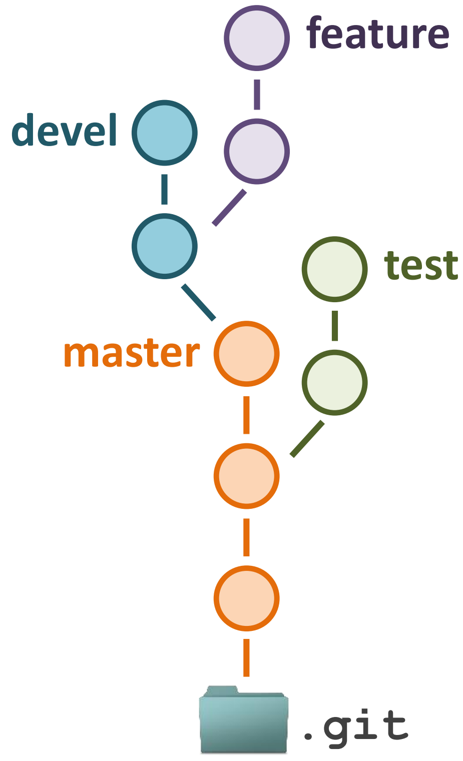
Git commands we assume you know

Reminder...

```
git init / git clone

git add <file>

git restore --staged <file> /
     git rm --cached <file>

git rm <file>

git commit –m “commit message”


git branch <branch>

git switch <branch>
git checkout <branch>

git switch -c <branch>
git checkout -b <branch>
```



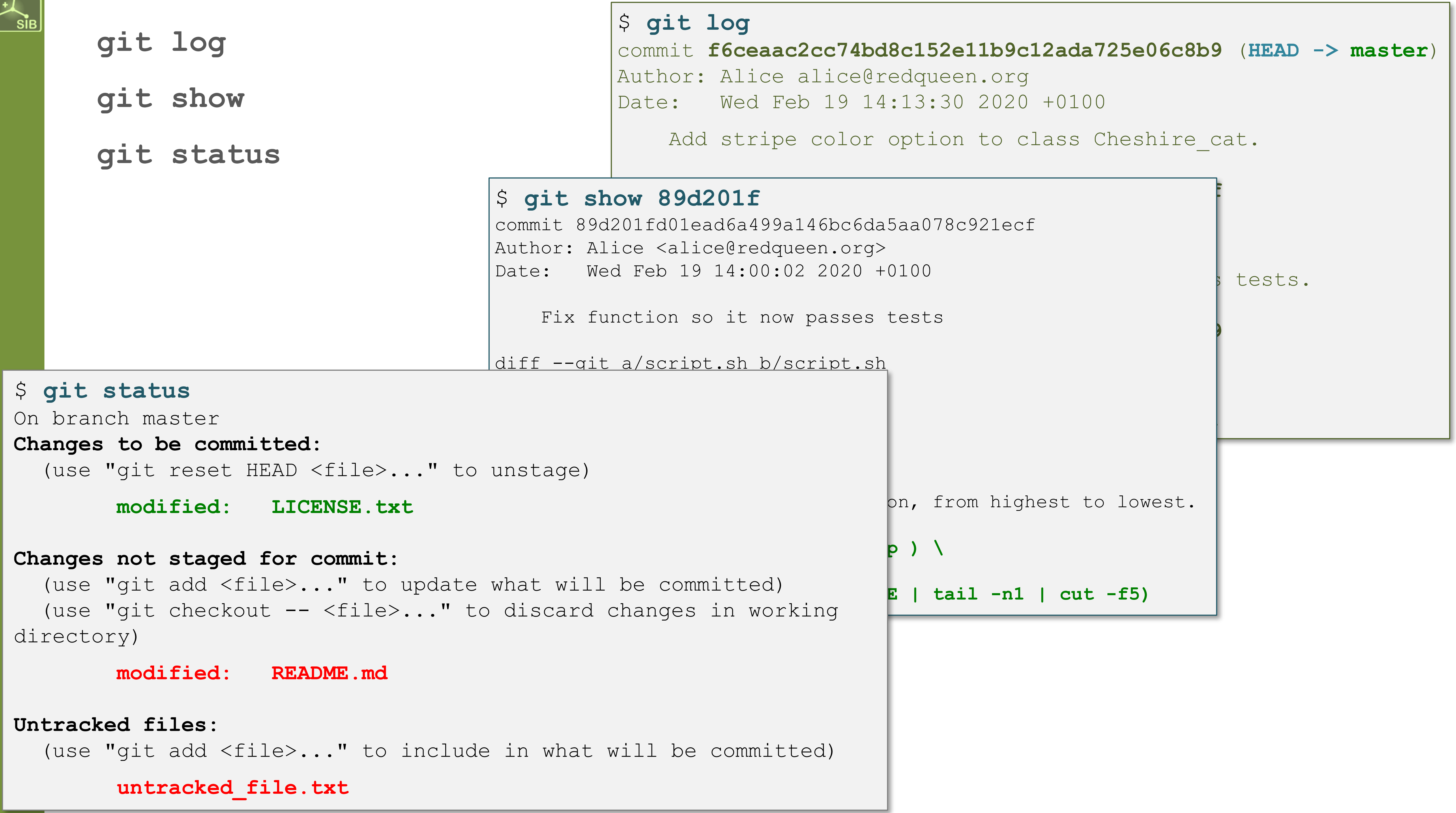




git log

git show

git status



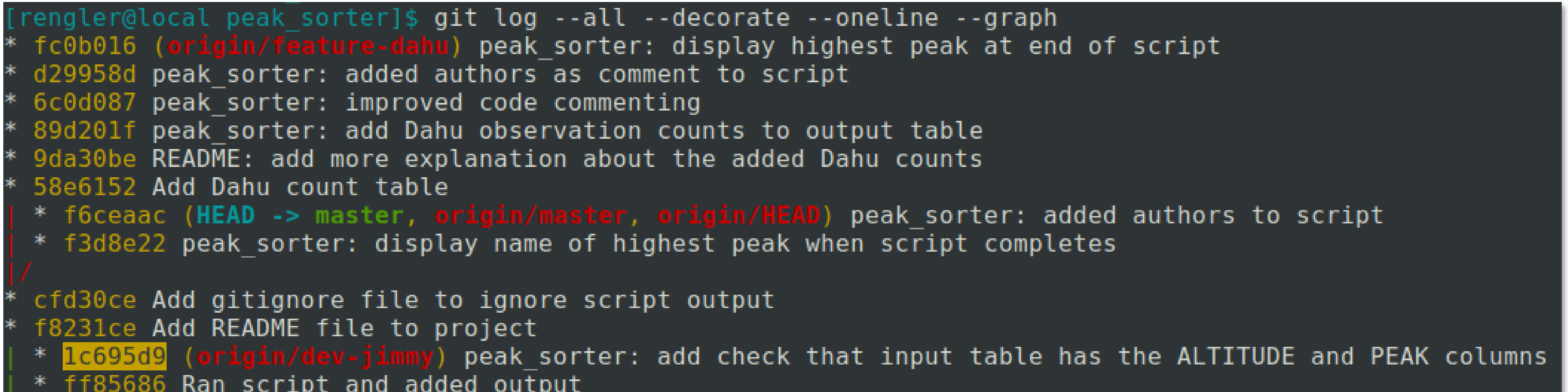
```
$ git log
commit f6ceaac2cc74bd8c152e11b9c12ada725e06c8b9 (HEAD -> master)
Author: Alice alice@redqueen.org
Date:   Wed Feb 19 14:13:30 2020 +0100

    Add stripe color option to class Cheshire_cat.

                                                       f

                                                       s tests.

                                                       9
```

```
$ git show 89d201f
commit 89d201fd01ead6a499a146bc6da5aa078c921ecf
Author: Alice <alice@redqueen.org>
Date:   Wed Feb 19 14:00:02 2020 +0100

    Fix function so it now passes tests

diff --git a/script.sh b/script.sh

                                      on, from highest to lowest.

                                      p ) \

                                      E | tail -n1 | cut -f5)
```

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   LICENSE.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)

        modified:   README.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        untracked_file.txt
```

Reminder...

git log --all --decorate --oneline --graph



```
[rengler@local peak_sorter]$ git log --all --decorate --oneline --graph
* fc0b016 (origin/feature-dahu) peak_sorter: display highest peak at end of script
* d29958d peak_sorter: added authors as comment to script
* 6c0d087 peak_sorter: improved code commenting
* 89d201f peak_sorter: add Dahu observation counts to output table
* 9da30be README: add more explanation about the added Dahu counts
* 58e6152 Add Dahu count table
| * f6ceaac (HEAD -> master, origin/master, origin/HEAD) peak_sorter: added authors to script
| * f3d8e22 peak_sorter: display name of highest peak when script completes
|/
* cfd30ce Add gitignore file to ignore script output
* f8231ce Add README file to project
| * 1c695d9 (origin/dev-jimmy) peak_sorter: add check that input table has the ALTITUDE and PEAK columns
| * ff85686 Ran script and added output
|/
* 821bcf5 peak_sorter: add +x permission
* 40d5ad5 Add input table of peaks above 4000m in the Alps
* a3e9ea6 peak_sorter: add first version of peak sorter script
```

I DON'T ALWAYS USE GIT LOG
BUT WHEN I DO, I USE "A DOG" --ALL --DECORATE --ONELINE --GRAPH
memegenerator.net

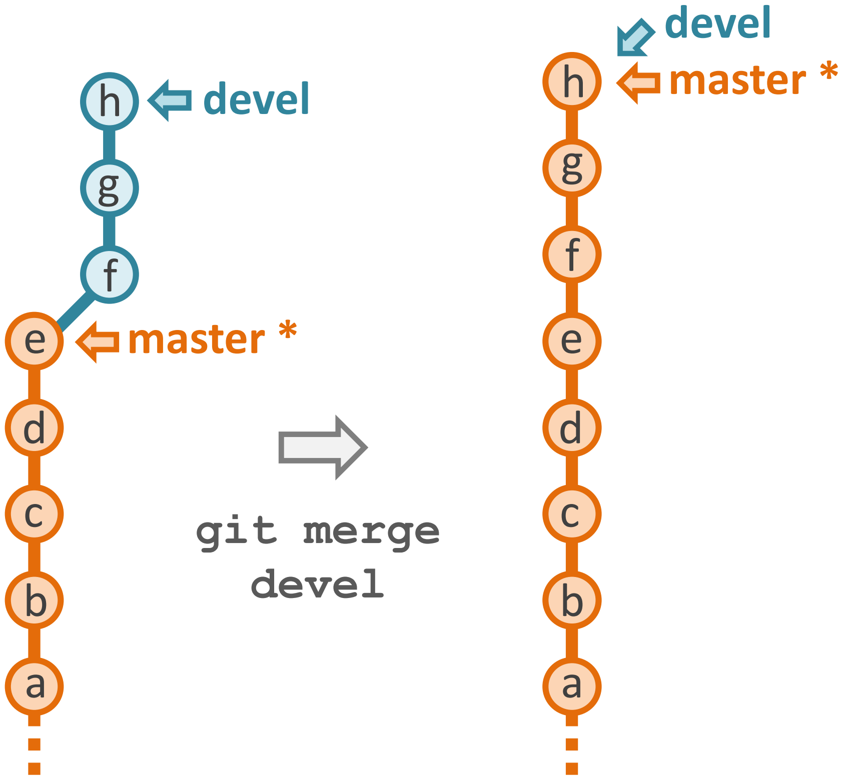
```
git config --global alias.adog "log --all --decorate
                                          --oneline --graph"
```

Branch merging

- For merge operations, the branch into which one merges must be the currently active branch (***** in the figures below).
- When the branch that is being merged (here **devel**) is rooted on the latest commit of the branch that it is being merged into (here **master**), the merge is said to be **fast-forward**.

Fast-forward merge

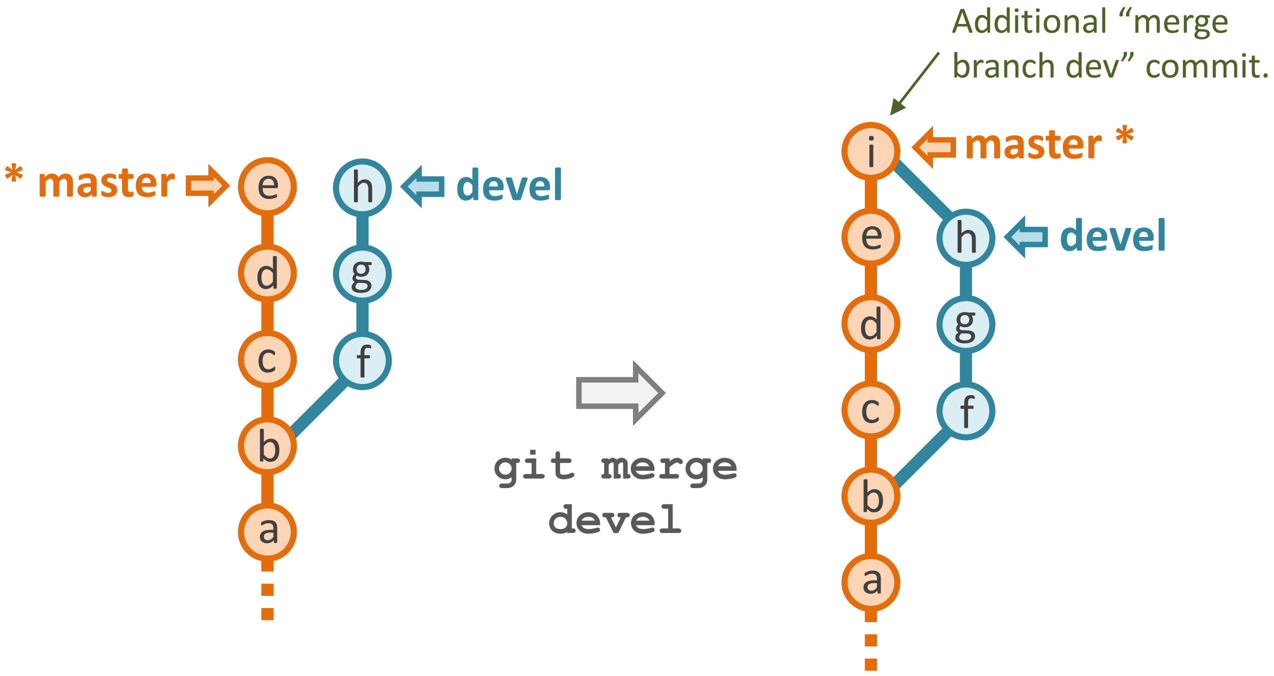
- Guaranteed to be conflict free.



```
git merge
devel
```

Non fast-forward merge

- Creates an additional “merge commit”.
- Conflicts may occur.



```
git merge
devel
```

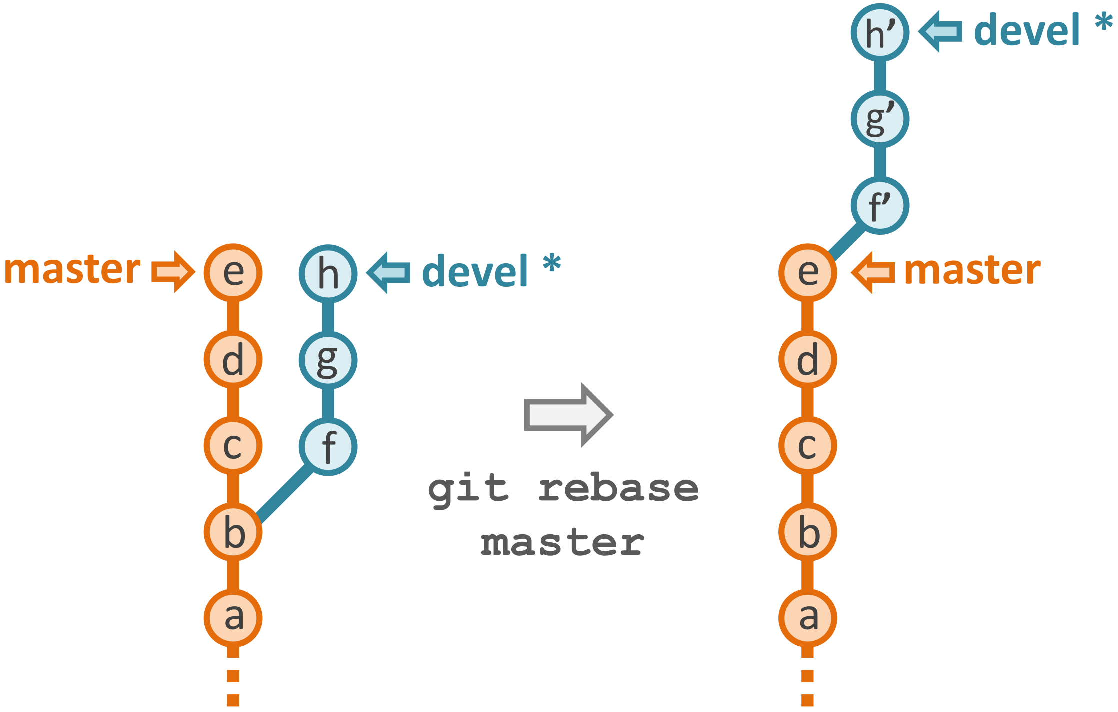
Additional “merge branch dev” commit.

Reminder…

Reminder...

Branch rebasing

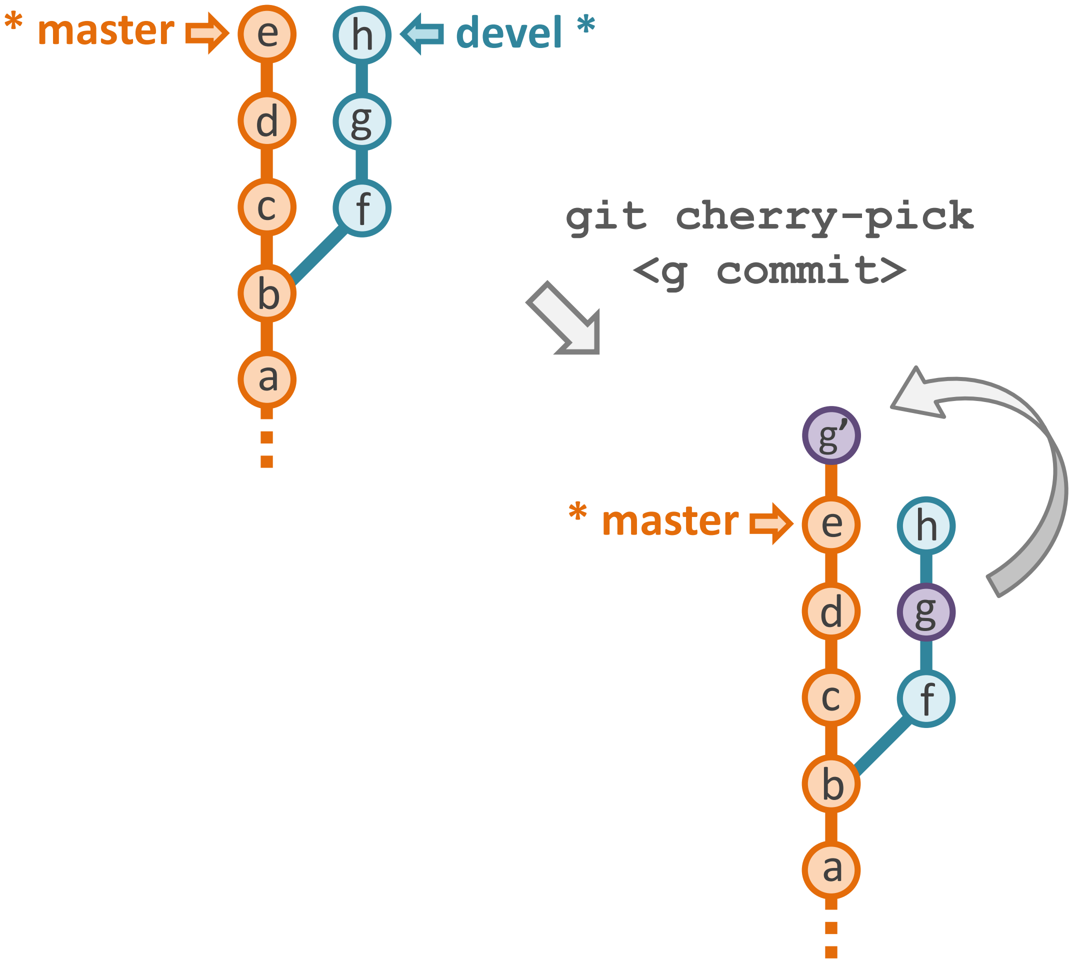
- For rebase operations, the branch being rebased must be the current branch (***** in the figures below).
- **Rebase operations re-write history**: the ID of rebased commits is modified (**'** in the figures below).
- Branches can be rebased on other branches, or on an older commit of themselves (interactive rebase).



```
git rebase
  master
```

Cherry-picking

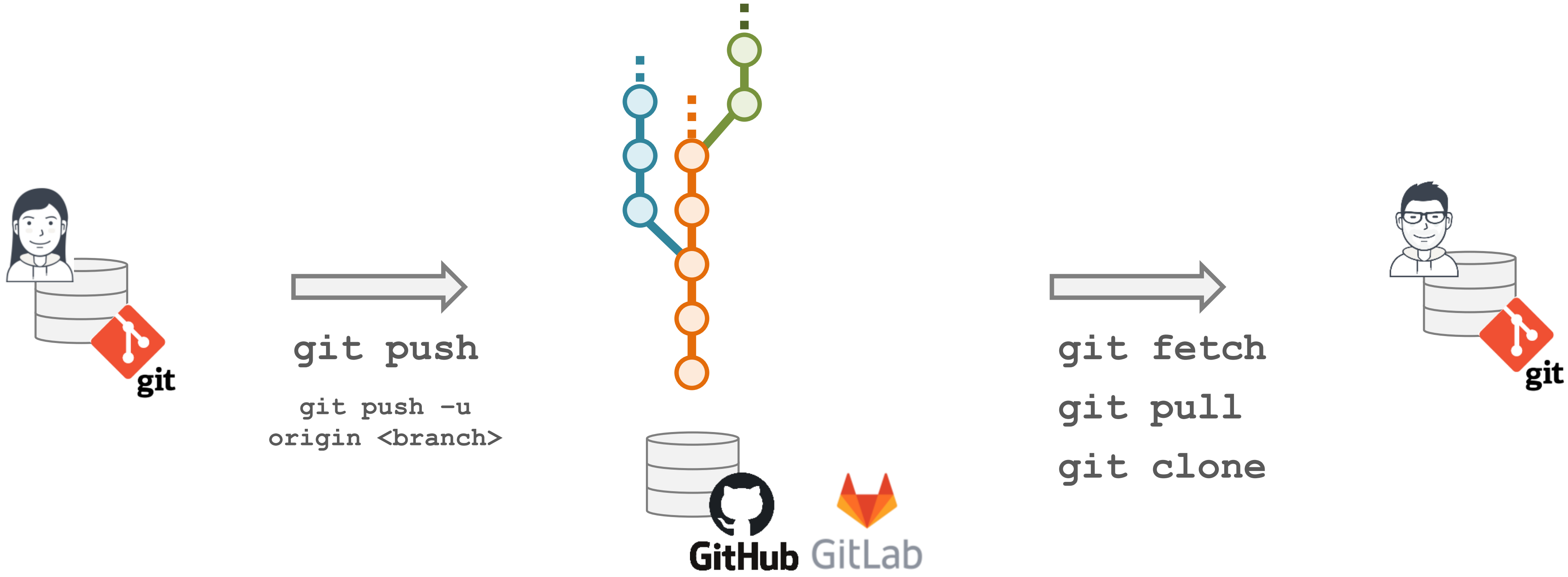
- "copy" changes introduced by a commit on another commit.



```
git cherry-pick
   <g commit>
```

Working with remotes

`git push`	Push (upload) changes on current branch to a remote.
`git push -u origin <branch>`	When pushing a newly created branch to the remote for the 1st time. “-u” is short for “--set-upstream”
`git fetch`	Retrieve (download) all changes from the remote.
`git pull`	`git fetch` + `git merge` of current branch with its remote counterpart.
`git clone`	Create a local copy of a remote repository.



```
git push
git push -u
origin <branch>
```

```
git fetch
git pull
git clone
```

Reminder…

rewriting history

power (and responsibility) at your fingertips
with **interactive rebase** and **git reset**

REMINDER

git commit --amend

Overwrite (re-write) the latest commit of a branch

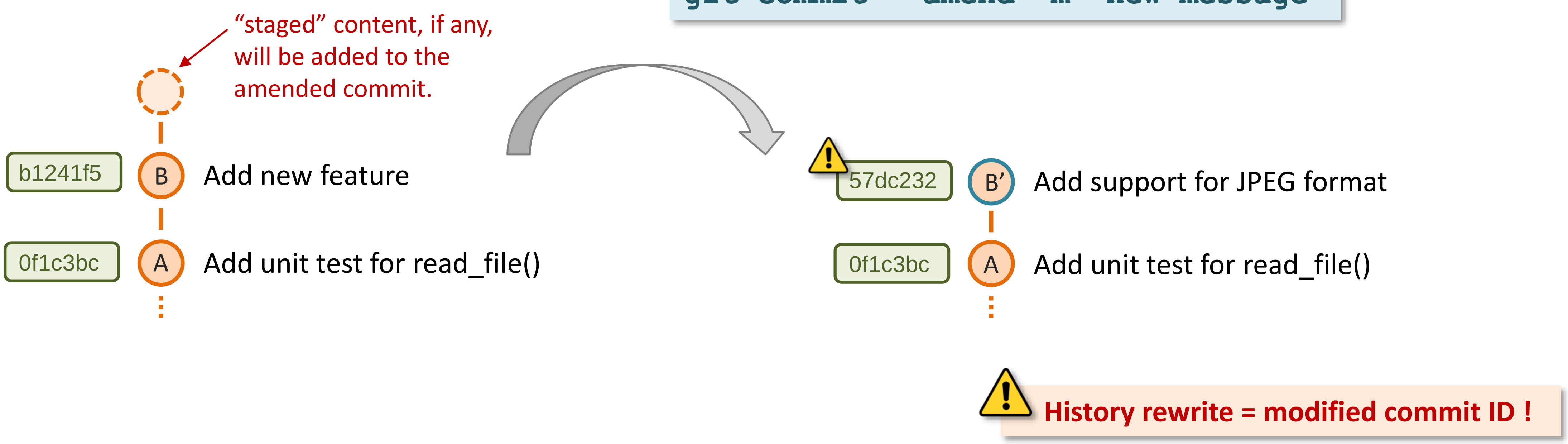
Reminder...

Amending the latest commit of a branch

Amend commit message (interactively or not)

```
git commit --amend
```

```
git commit --amend –m “new message”
```


“staged” content, if any, will be added to the amended commit.
b1241f5
B
Add new feature
0f1c3bc
A
Add unit test for read_file()
57dc232
B’
Add support for JPEG format
0f1c3bc
A
Add unit test for read_file()

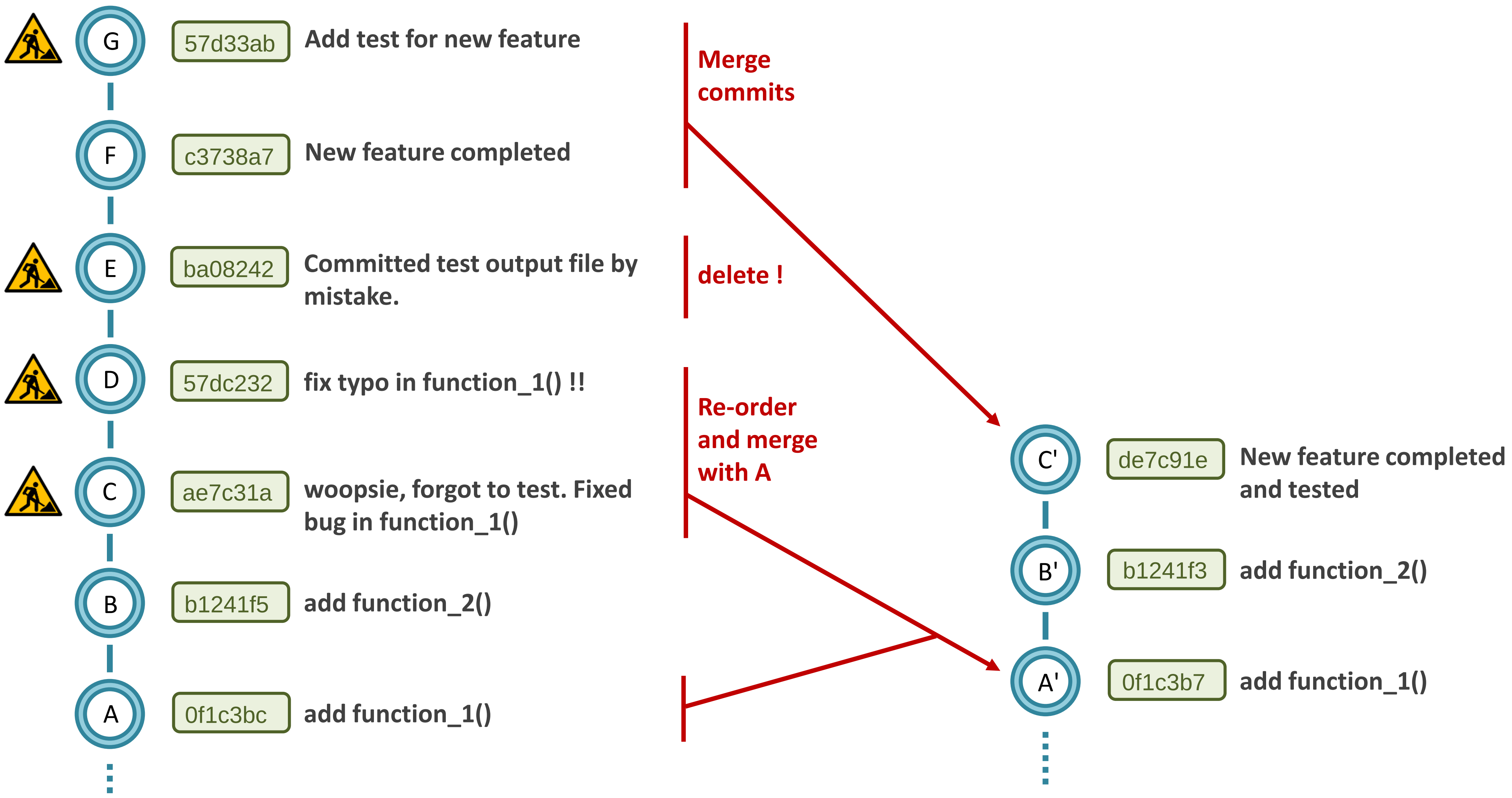
History rewrite = modified commit ID !

To amend while keeping the same commit message:
(only edit the content of the commit)

```
git commit --amend --no-edit
```

interactive rebase

Interactive rebase: re-order, squash, and delete commits



Commit history of your new feature …

G 57d33ab Add test for new feature

F c3738a7 New feature completed

E ba08242 Committed test output file by mistake.

D 57dc232 fix typo in function_1() !!

C ae7c31a woopsie, forgot to test. Fixed bug in function_1()

B b1241f5 add function_2()

A 0f1c3bc add function_1()

Merge commits

delete !

Re-order and merge with A

… and how you wish it was.

C' de7c91e New feature completed and tested

B' b1241f3 add function_2()

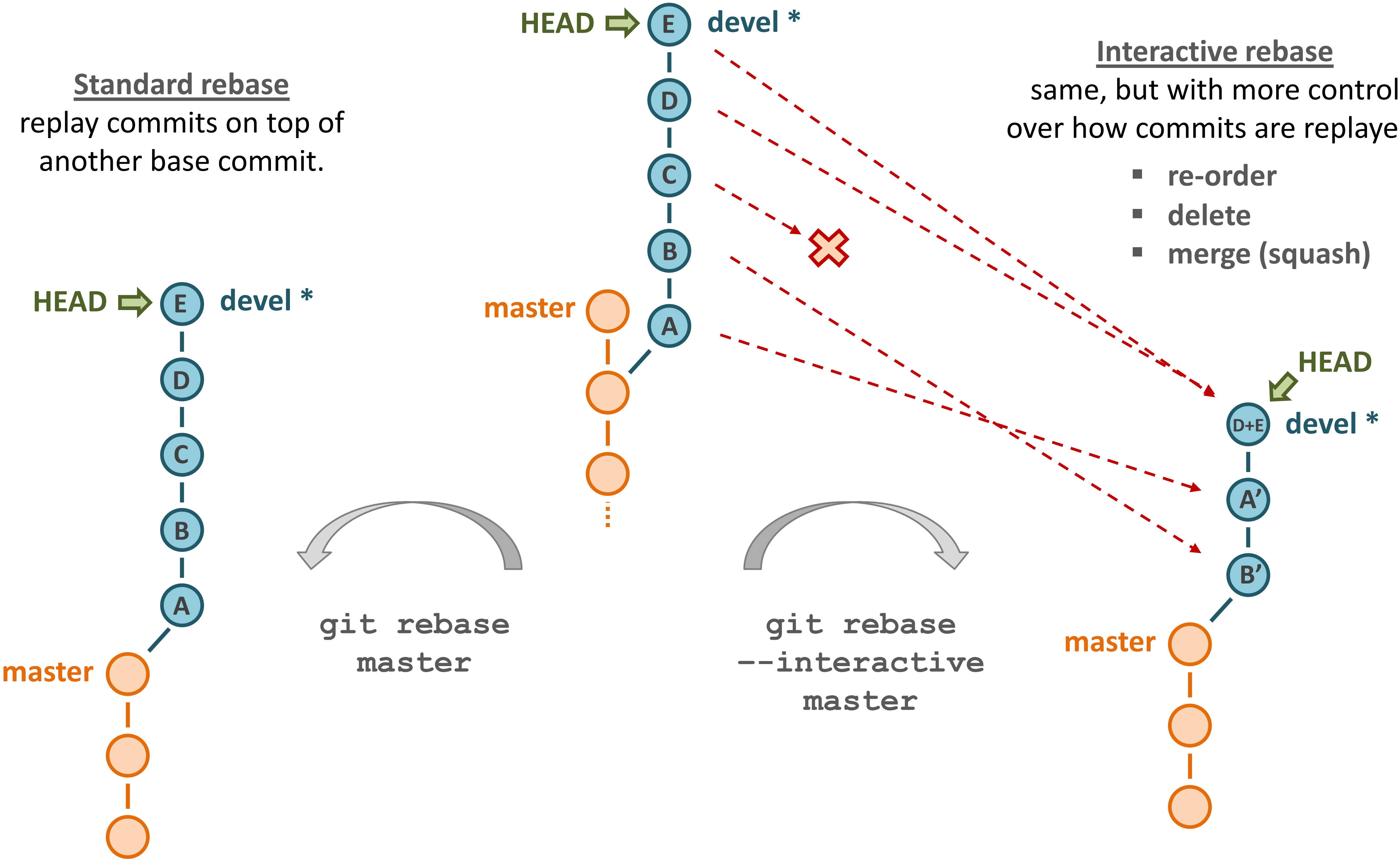
A' 0f1c3b7 add function_1()

Standard vs. interactive rebase

Standard rebase
replay commits on top of
another base commit.

Interactive rebase
same, but with more control
over how commits are replayed:

- re-order
- delete
- merge (squash)



`git rebase master`

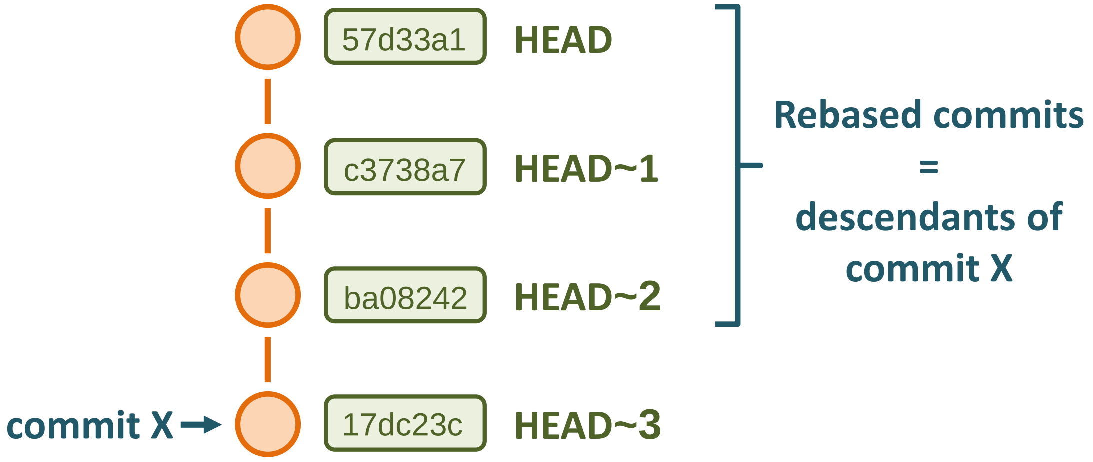
`git rebase --interactive master`

Interactive rebase: re-order, squash, and delete commits

```
git rebase --interactive/-i <commit X ref>
```

parent of first commit in the rebase

- Starting from the specified `<commit X>`, Git opens a text editor where you interactively give instructions on how to modify the history of all descendent commits of **X** by:
  - Re-ordering commits.
  - Merging one or more commits together.
  - Deleting commits.
- Then Git will rewind to `<commit X>`, and re-apply the descendant commits as instructed.



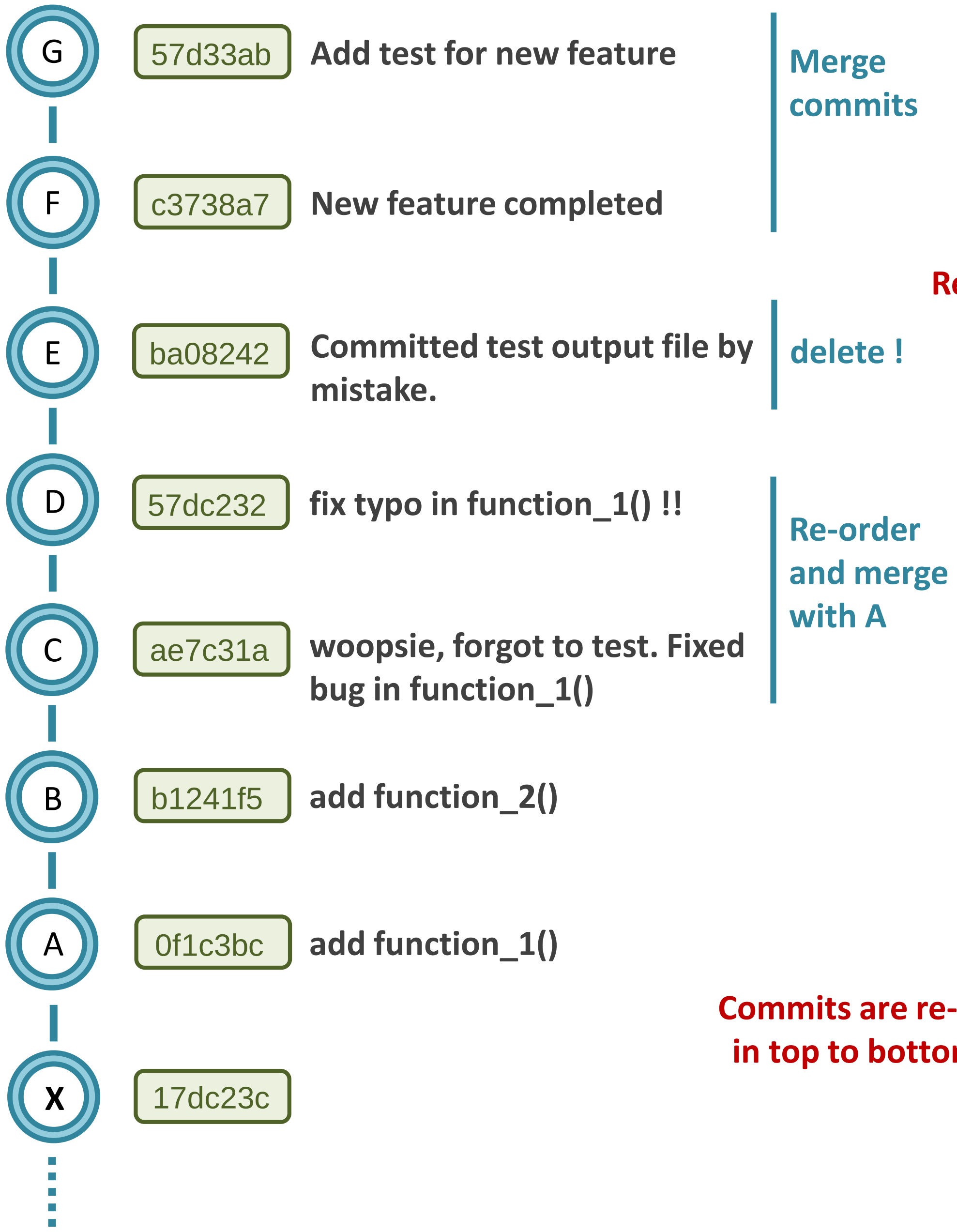
To rebase the last 3 commits (descendants of commit X), these 2 commands will yield the same result:

```
$ git rebase -i 17dc23c
$
```

Absolute reference to commit X

```
$ git rebase -i HEAD~3
$
```

Relative reference to commit X



G 57d33ab Add test for new feature

F c3738a7 New feature completed

E ba08242 Committed test output file by mistake.

D 57dc232 fix typo in function_1() !!

C ae7c31a woopsie, forgot to test. Fixed bug in function_1()

B b1241f5 add function_2()

A 0f1c3bc add function_1()

X 17dc23c

Merge commits

delete !

Re-order and merge with A

```
$ git rebase -i 17dc23c
```

or

```
$ git rebase -i HEAD~7
```

opens the following in Git's default editor (e.g. vim)

⚠ Reversed order!

```
pick 0f1c3bc add function_1()
pick b1241f5 add function_2()
pick ae7c31a woopsie, forgot to test. Fixed bug in function_1()
pick 57dc232 fix typo in function_1() !!
pick ba08242 Committed test output file by mistake.
pick c3738a7 New feature completed
pick 57d33ab Add test for new feature.

# Commands:
# p, pick <commit> = use commit
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard log message
# d, drop <commit> = remove commit
... there are more commands.
```

manual editing of file.

⚠ Commits are re-applied in top to bottom order

```
pick 0f1c3bc add function_1()
f    ae7c31a woopsie, forgot to test. Fixed bug in function_1()
f    57dc232 fix typo in function_1() !!
pick b1241f5 add function_2()
d    ba08242 Committed test output file by mistake.
pick c3738a7 New feature completed
s    57d33ab Add test for new feature
```

```
pick 0f1c3bc add function_1()
f    ae7c31a woopsie, forgot to test. Fixed bug in function_1()
f    57dc232 fix typo in function_1() !!
pick b1241f5 add function_2()
d    ba08242 Committed test output file by mistake.
pick c3738a7 New feature completed
s    57d33ab Add test for new feature
```

Save and close to start rebasing (":wq" or “:x” in vim).

For squashes, Git will open an editor so you can edit the commit message.

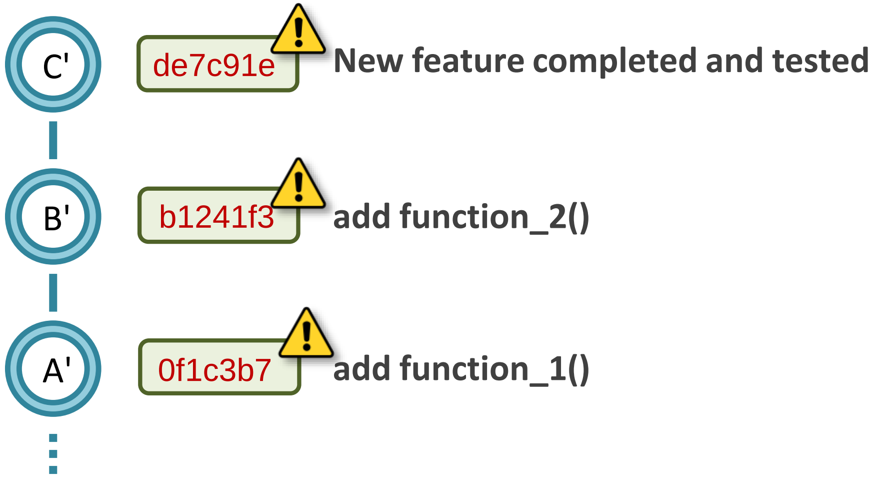
If there are any conflicts, you will need to solve them manually.

```
$ vim <file with conflict>           # manual conflict resolution
$ git add <file(s) with conflict>
$ git rebase --continue
```

Rebase completed

⚠ **Rebase re-writes history -> Commit ID values are now different !**

History after the rebase:



Supplementary material...

Example of interactive rebase file (in full):

```
pick 0f1c3bc add function_1()
f    ae7c31c woopsie, forgot to test. Fixed bug in function_1()
f    57dc233 fix typo in function_1() !!
pick b1241f5 add function_2()
d    ba08242 Committed test output file by mistake.
pick c3738a7 New feature completed and tested
f    57d33ab Well, there was still a bug and a typo… now fixed

# Rebase 17dc23c..0f1c3b2 onto 17dc23c
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

Last descendent

Parent commit (i.e. commit X)

Commits are re-applied in the order from top to bottom.

squash vs **fixup**:
Both will squash the specified commit into the previous one, the difference is how the log message is handled:

- fixup: log message the squashed commit is discarded, the message of commit into which the squash occurs is kept.
- squash: an editor opens to let you interactively enter a new log message. It is pre-filled with the messages of both commits.

You can delete a line to delete a commit (instead of changing "pick" to "d"/"drop".

To abort the rebase, delete all lines in the file (comments do not need to be deleted).

Commands: either the 1-letter shortcut or the full command name can be used.

--fixup commits

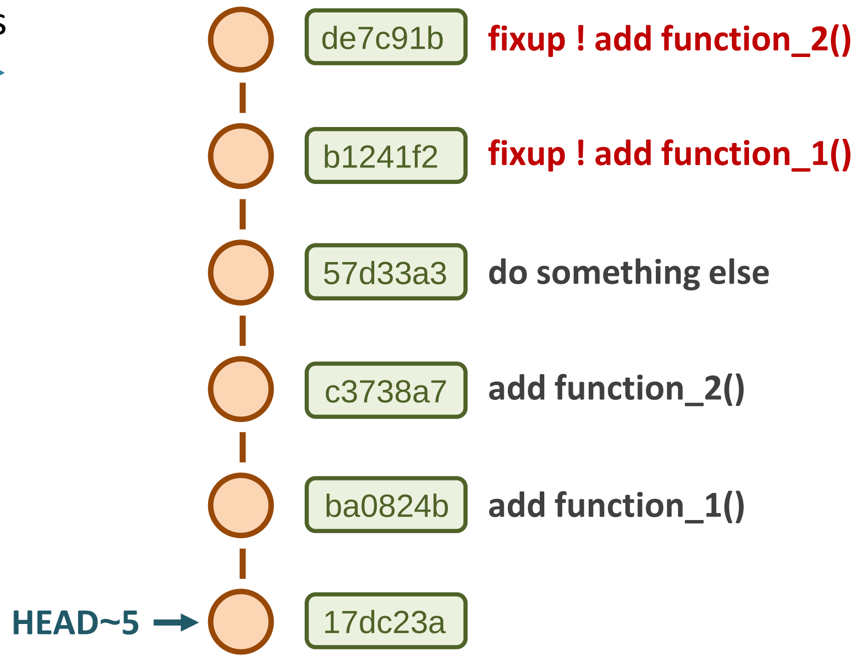
- When you realize you made a mistake in an earlier commit, you can directly tag it as a fixup with `git commit --fixup=<hash/ref of commit to be fixed>`
- Running an interactive rebase with the `--autosquash` option added, Git will automatically re-order commits for you.

```
# work on the fix for function_1(). Commit it as a --fixup.
$ git add <file that was fixed>
$ git commit --fixup=ba0824b
$
# work on fix for function_2(). Commit it as a --fixup.
$ git add <file that was fixed>
$ git commit --fixup=c3738a7
$
# Now we can rebase with the –autosquash option.
$ git rebase –i --autosquash HEAD~5
$
```

with the `--autosquash` option enabled, Git automatically places the fixup commits in at the correct position, and marks them as "fixup". No manual editing needed !

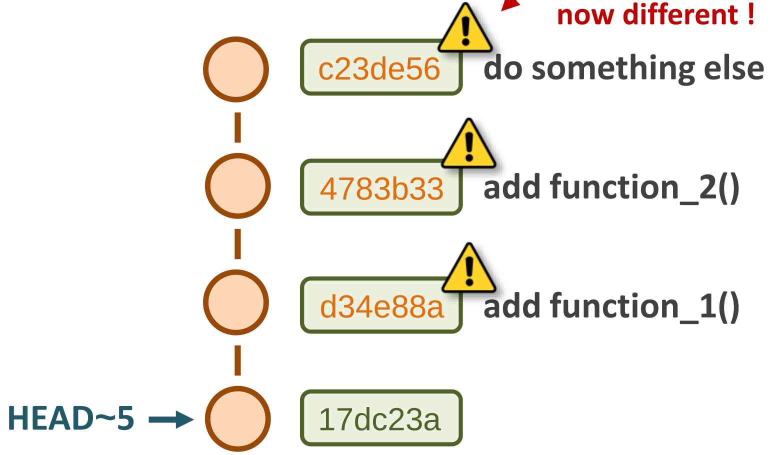
```
pick ba0824b add function_1()
fixup b1241f2 fixup ! add function_1()
pick c3738a7 add function_2()
fixup de7c91b fixup ! add function_2()
pick 57d33a3 do something else
```

History before the rebase.



de7c91b fixup ! add function_2()
b1241f2 fixup ! add function_1()
57d33a3 do something else
c3738a7 add function_2()
ba0824b add function_1()
HEAD~5
17dc23a

History after the rebase.



Commit hash are now different !

Rebasing the root commit (first commit of a repository)

- The regular `git rebase -i/--interactive` command does not allow to edit the first commit of a Git repository.

- To rebase history including the first commit, the `--root` option must be added:

```
git rebase --root --interactive <tip commit of branch>
```

⚠ With the `--root` option, you must indicate the tip of the branch to rebase, not the parent commit (there is no parent to the root commit)

Examples:

```
$ git rebase --root --interactive HEAD
$ git rebase --root -i master
```

Supplementary material...

demo: interactive rebase

exercise 1

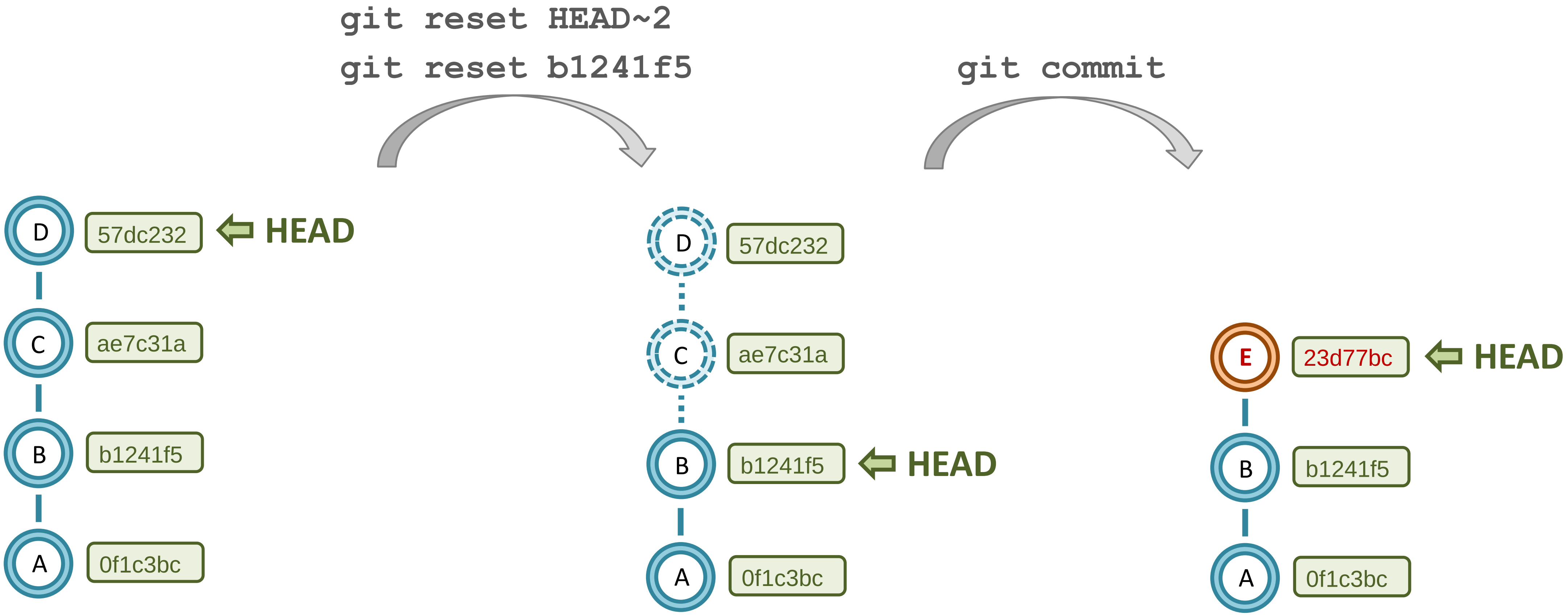
The vim cheat-sheet rebase

git reset

git reset – move the HEAD to a specific commit

- The `git reset` command moves the **HEAD** pointer to the specified commit.
- Commits between the former **HEAD** position and its new positon will be "removed" from history upon the next commit (but they will remain in the Git database for a little while).

```
git reset <commit to where HEAD should be moved>
```



```
git reset HEAD~2
git reset b1241f5
```

```
git commit
```

git reset – move the HEAD to a specific commit

- 3 options allow to specify how the index and working tree should be affected:
  - `--soft` : reset the HEAD only (keep staged content in the index).
  - `--mixed`: reset the HEAD + the index.
  - `--hard` : reset the HEAD + the index + the working tree.

⚠ **The --hard option resets (overwrites) the working tree !**

This can lead to data loss if you have uncommitted changes.

```
git reset --mixed/--soft/--hard  <commit ref>
```

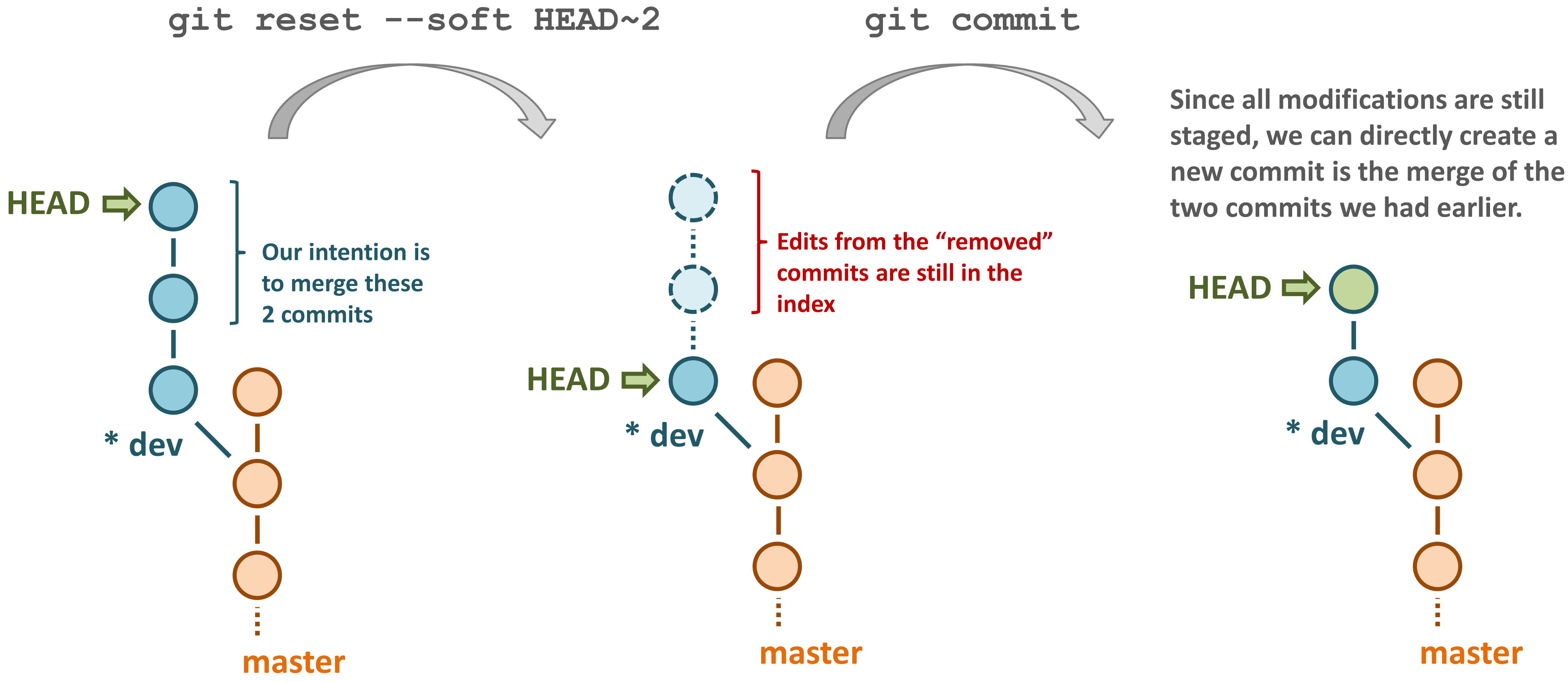
mixed is the default value (so you don't need to actually specify it)

Reset options effects: a check mark indicates elements that are reset.

	HEAD	index Staged content	work tree Files on disk
`--soft`	✓		
`--mixed`	✓	✓	
`--hard`	✓	✓	✓

git reset --soft use case: merge the last 2 commits into one

`--soft` : reset the HEAD only (keep staged content in the index).


git reset --soft HEAD~2
git commit
HEAD
Our intention is to merge these 2 commits
* dev
master
HEAD
Edits from the “removed” commits are still in the index
* dev
master
HEAD
* dev
master

Since all modifications are still staged, we can directly create a new commit is the merge of the two commits we had earlier.

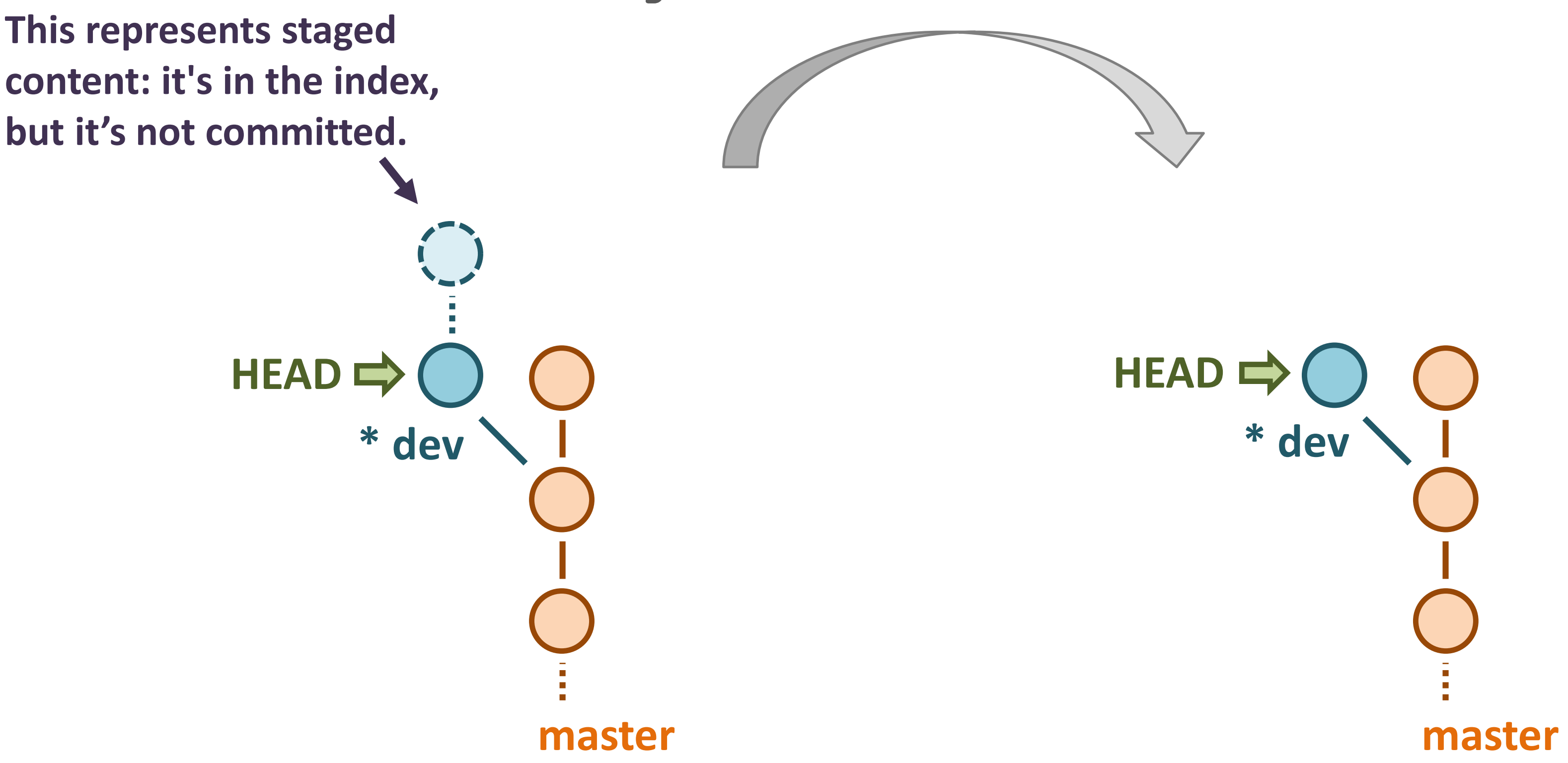
The HEAD was reset, but the modifications introduced by the “removed" commits are still in the index and the working tree.

If there are conflicts between the content of the “removed” commits, the latest version of the conflicting lines remains in the index.

git reset --mixed use case: clear the staging area from new content

- `--mixed` : reset the HEAD + the index.
- Useful to clear the index from newly staged content, e.g. when you staged something by mistake.

```
git reset --mixed HEAD
git reset HEAD
```



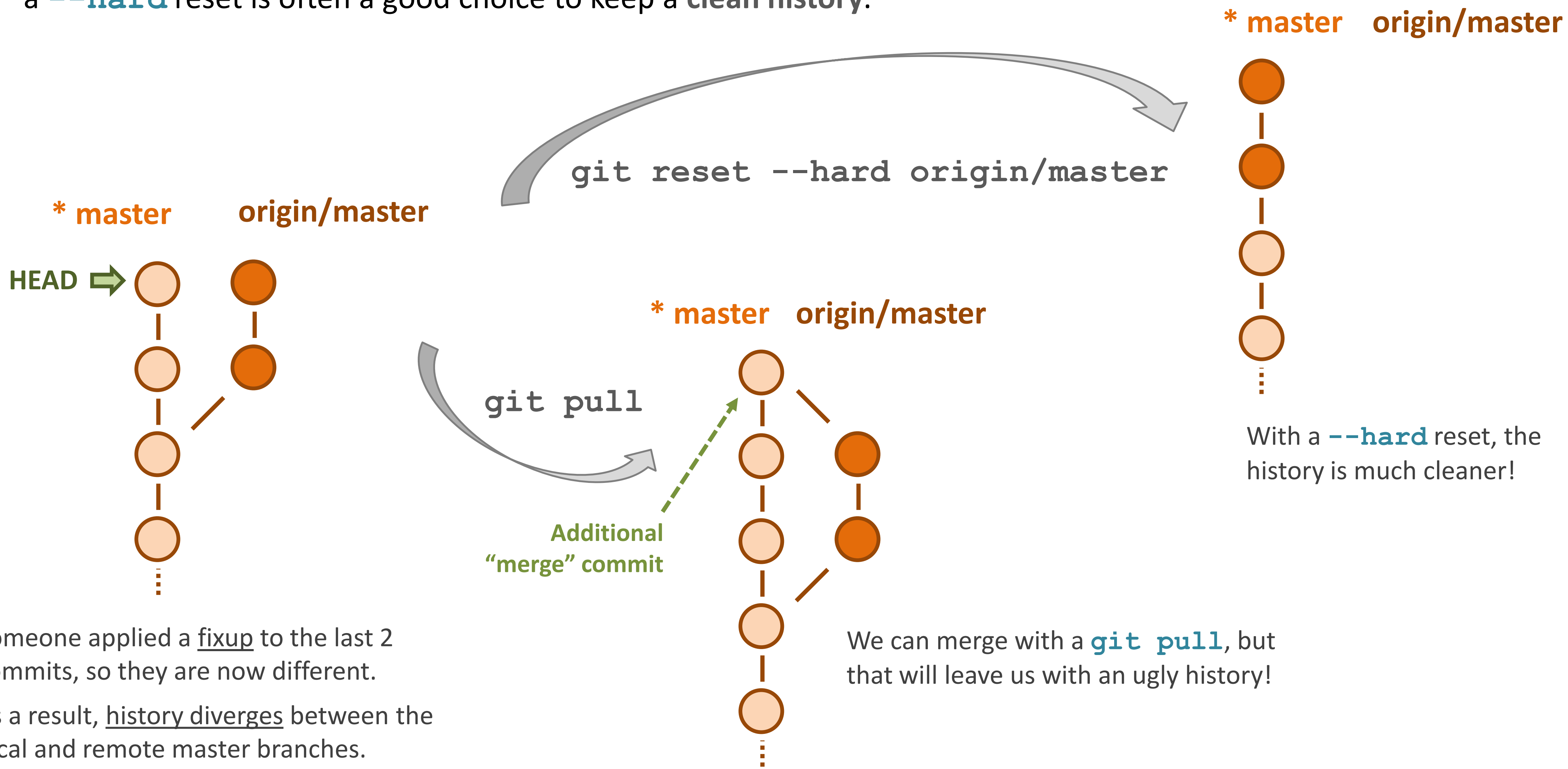
This represents staged content: it's in the index, but it’s not committed.

The newly staged content is now removed from the index, because the index was reset to its state at the **HEAD** position.

But any changes made in the working tree is still there: `--mixed` <u>does not modify the working tree</u>, so we are not losing any work.

git reset --hard use case: reset a branch to a remote

- `--hard` : reset the HEAD + the index + the working tree.
- When a remote branch had "forced updates" (i.e. someone changed its history), a `--hard` reset is often a good choice to keep a **clean history**.



```
git reset --hard origin/master
```

```
git pull
```

Someone applied a fixup to the last 2 commits, so they are now different.

As a result, history diverges between the local and remote master branches.

We can merge with a `git pull`, but that will leave us with an ugly history!

With a `--hard` reset, the history is much cleaner!

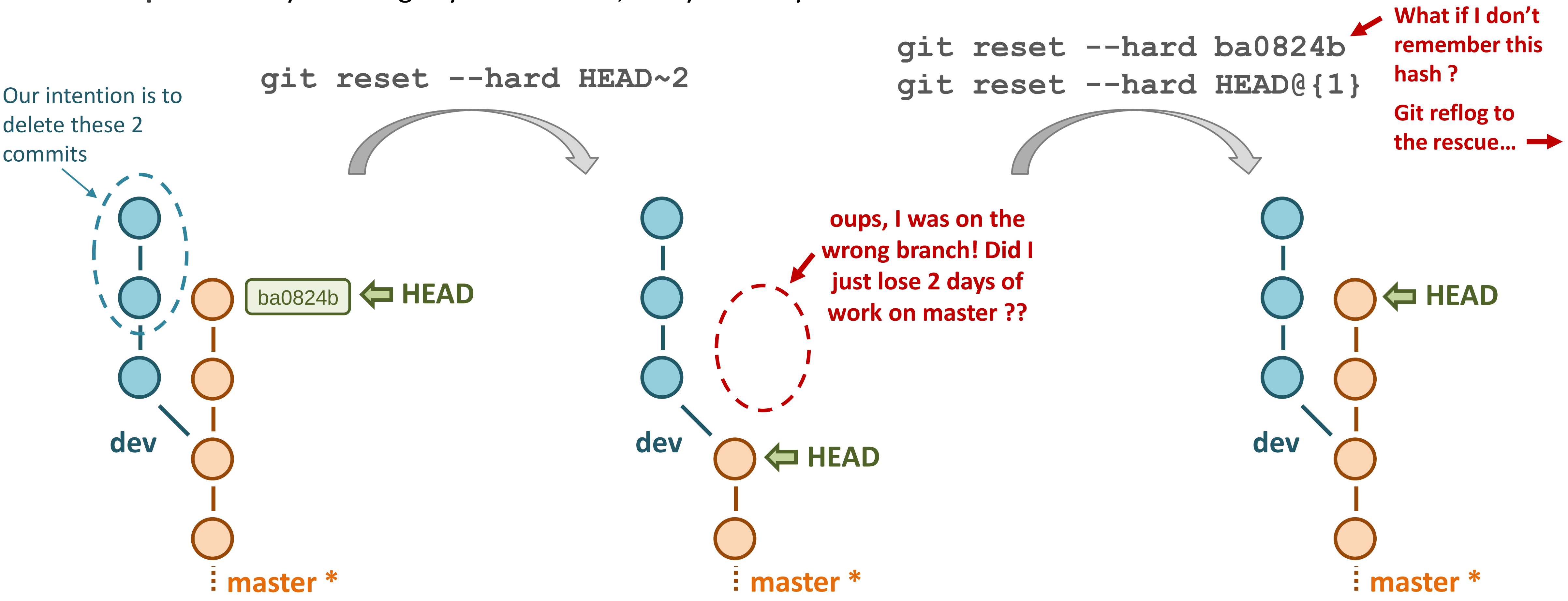
git reset --hard use case: reset a reset, a merge, a rebase (or anything, really)

- A `--hard` reset can be used to undo (almost**) any operation, and get back exactly to the previous state *

  * as long as Git did not do garbage collection on orphaned commits and deleted them (see two slides further).

⚠ If you reset `--hard` changes that have not been committed/staged/stashed, **you will lose your work**!
(untracked files are not affected)

- **Example**: When you thought you're on dev, but you really are on master…



```
git reset --hard HEAD~2
```

```
git reset --hard ba0824b
git reset --hard HEAD@{1}
```

Our intention is to delete these 2 commits

oups, I was on the wrong branch! Did I just lose 2 days of work on master ??

What if I don't remember this hash ?

Git reflog to the rescue…

The Git reflog and the HEAD@{x} relative reference

- `git reflog` shows the “reflog”: a chronological log of all operations that were performed on a repository.

```
git reflog
```

```
$ git reflog
11d4dc8 (HEAD -> master, dev) HEAD@{0}: merge dev: Fast-forward
5061456 HEAD@{1}: checkout: moving from dev to master
11d4dc8 (HEAD -> master, dev) HEAD@{2}: commit: Update README
5061456 HEAD@{3}: checkout: moving from master to dev
5061456 HEAD@{4}: commit: Add README file
0f84d17 HEAD@{5}: commit (initial): Initial commit
```

Commit IDs of commit at HEAD position

- The `HEAD@{x}` notation indicates the positon of the HEAD pointer relatively to the reflog.
- It can be used as a commit reference, e.g. `git reset --hard HEAD@{1}`.

`HEAD@{0}`	Current position of HEAD.
`HEAD@{1}`	Positon of HEAD 1 operation ago.
`HEAD@{2}`	Positon of HEAD 2 operations ago.
`...`	
`HEAD@{x}`	Positon of HEAD x operations ago.

What happens to orphaned commits ?

- When a commit (or a group of commits) are no longer part of a branch or referenced by a tag, they are said to be ***orphaned***. E.g. in the diagram on the right, after a `git reset`, two commits (dashed circles) are now orphaned.

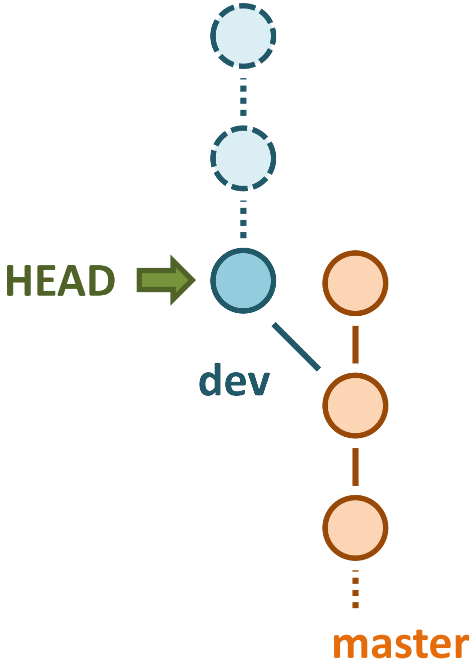
- Orphaned commits remain accessible in Git's database for a while, until they are ***garbage collected*** (i.e. deleted) by Git.

- To retrieve content from an orphaned commit, you can:
  - Display its content with `git show <orphaned commit ID>`
  - Check it out in a new branch: `git switch -c <orphaned commit ID>` or
    `git checkout -b <orphaned commit ID>`
  - Check it out in detached head mode: `git checkout <orphaned commit ID>`
  - Reset your current branch to it: `git reset --hard <orphaned commit ID>`
    Warning: this last option might itself create orphaned commits – also make sure you have a clean working tree, otherwise uncommitted changes will be lost).

- If you don't know the hash of an orphaned commit, you can find it by looking at the output of `git reflog --all`. This is a log of all operations that that were done by Git, and all commits will be referenced in there.

- If, for some reason, you want to force-delete all orphaned commits (and associated data), run the following command sequence. ***Warning:*** only do that if you understand why you're doing it.

```
git reflog expire --expire=all --all
git gc --aggressive --prune=now
```



History overwrite warning !

Commands illustrated in this section (in particular `git rebase` and `git reset`) often result in a **modification of a repo's history**.

When pushed to a remote, this can cause various levels of "inconvenience" to other people working on the same project.

- Ideally, do this type of operations ***before*** pushing to a remote.
- If you nevertheless need to push history modifications:
  - Use "force" push: `git push --force`
  - Coordinate the update with other people working on the repo, as they might need to do a `git reset --hard origin/<branch name>` on their local repo.
- (Try to) never rewrite a "production" branch shared with the outside world.
  Typically this would be the "main" or "master" branch.

exercise 2

The big reset

git checkout

The "detached HEAD" state explained

Reminder: checkout the entire state of an earlier commit

- Checking out a commit will restore both the working tree and the index to the exact state of that commit.

- It will also move the **HEAD** pointer to that commit.

```
git checkout <commit reference>
```

Example:

```
$ git checkout ba08242
$ git checkout HEAD~10
$ git checkout v2.0.5
```

- But you will enter a "detached HEAD" state....

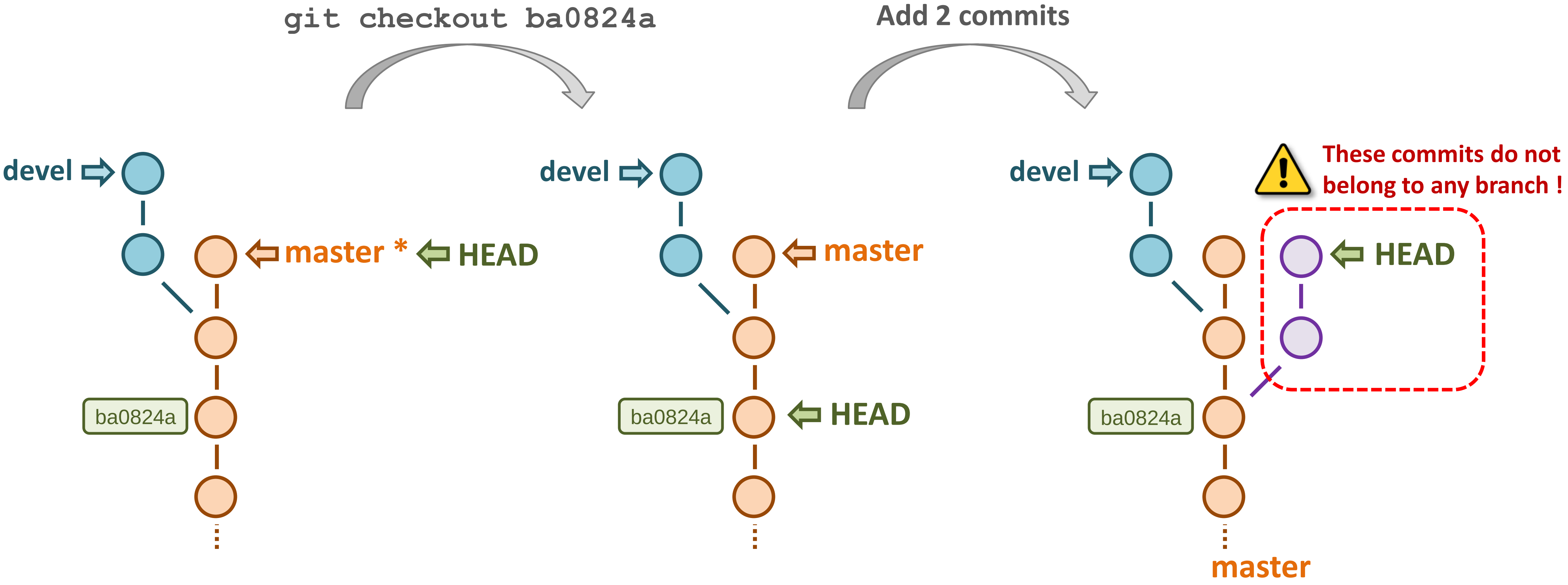
```
$ git checkout ba08242
Note: checking out 'ba08242'.

You are in 'detached HEAD' state. You can look
around, make experimental changes and commit
them, and you can discard any commits you make
in this state without impacting any branches
by performing another checkout.
```

- To get back to a “normal” state:
  `git checkout <branch>`

Reminder...

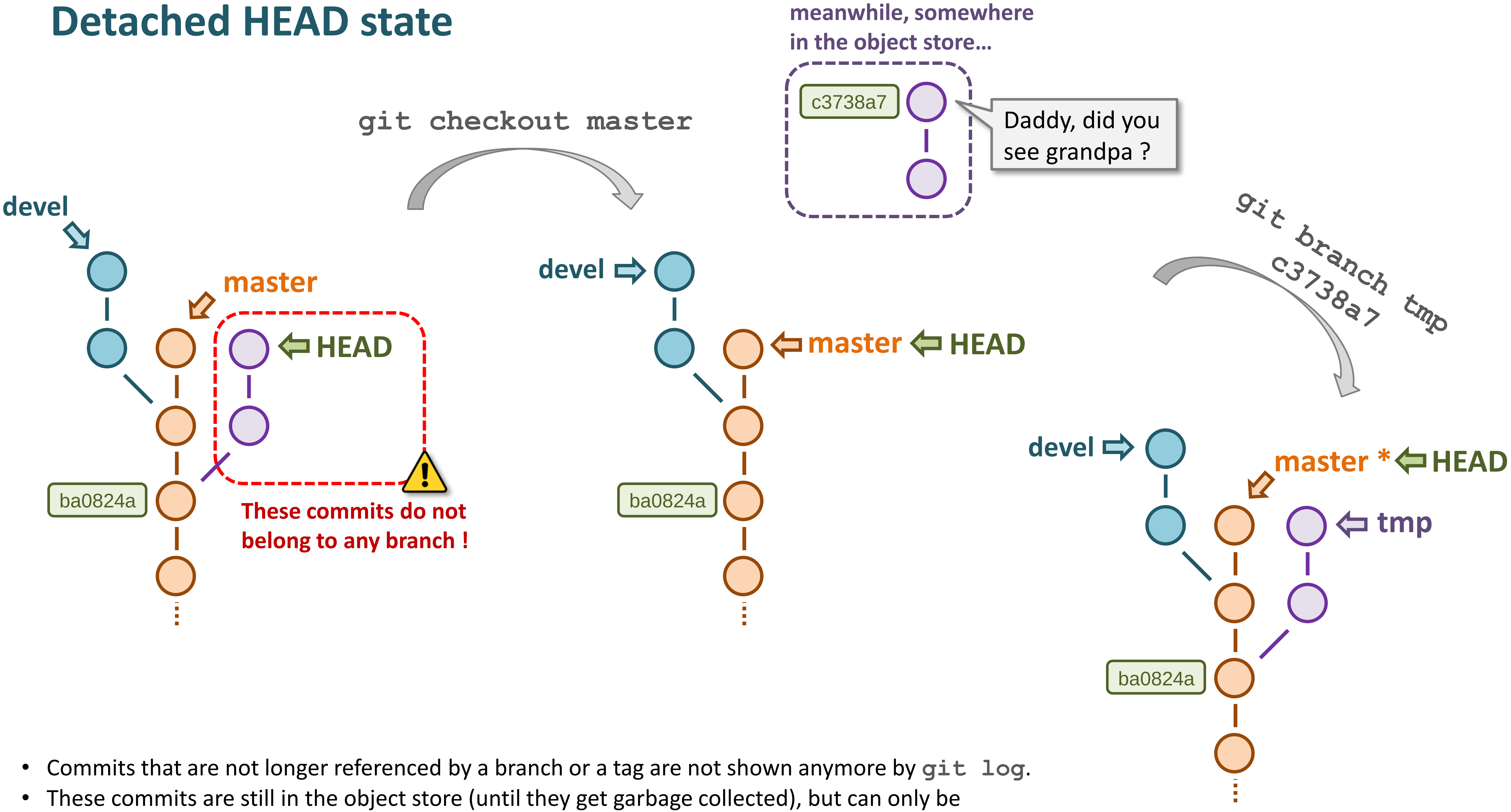
Detached HEAD: when HEAD points directly to commit instead of a branch

- After a `git checkout <commit>` command, **HEAD** points directly to a commit rather than a branch: this is known as **detached HEAD state.**



What if I go back to a "real" branch ?

Detached HEAD state



meanwhile, somewhere in the object store…

```
git checkout master
```

```
git branch tmp c3738a7
```

These commits do not belong to any branch !

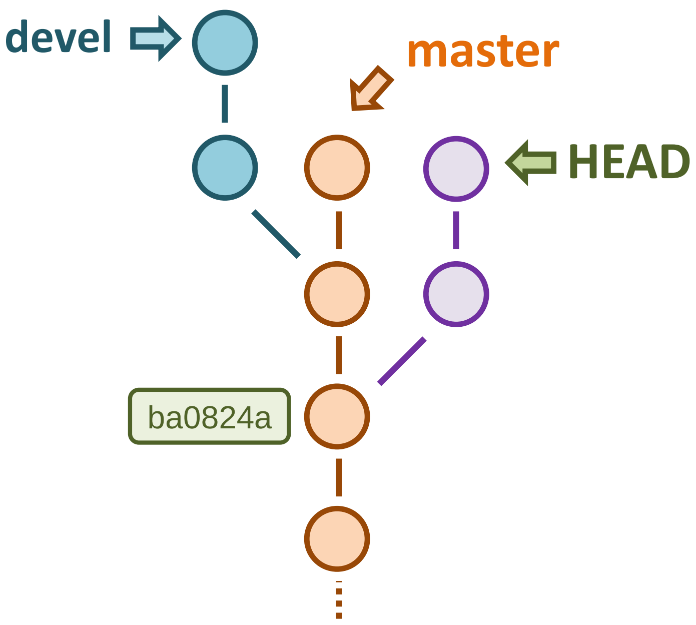
- Commits that are not longer referenced by a branch or a tag are not shown anymore by `git log`.
- These commits are still in the object store (until they get garbage collected), but can only be reached directly through their commit hash - or reflog references `HEAD@{x}`.

Creating a new branch while in detached HEAD state

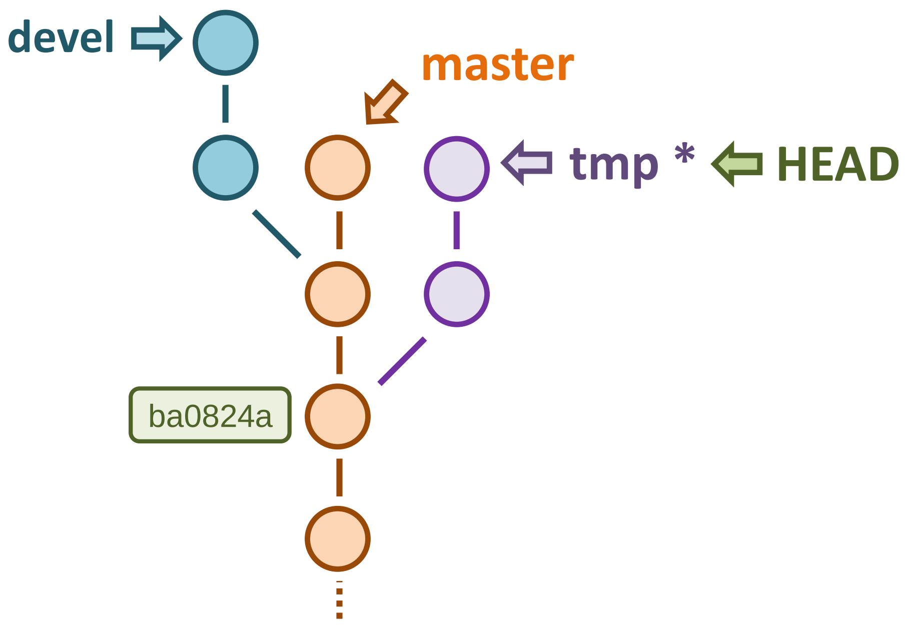
- To preserve commits created in detached HEAD state, a new branch can be created at any time while we are in “detached head” state. After the branch is created, we are no longer in detached HEAD state.

```
git switch -c/--create <branch name>
   git checkout -b <branch name>
```

In “detached head” state



On a regular branch (here “tmp”)



Note: `git switch -c` is the modern alternative to `git checkout -b` in Git versions >= 2.23

Detached HEAD

- In practice, Git will give you a lot of warnings and advice when in detached HEAD state:

```
$ git checkout e35e2a4
Note: switching to 'e35e2a4'.

You are in 'detached HEAD' state. You can look around, make
experimental changes and commit them, and you can discard
any commits you make in this state without impacting any
branches by performing another checkout.

If you want to create a new branch to retain commits you
create, you may do so (now or later) by using -c with the
switch command. Example:

  git switch -c <new-branch-name>

HEAD is now at e35e2a4 removed from git file
```

```
$ git checkout master
Warning: you are leaving 2 commits behind, not connected to
any of your branches:

  0860b65 another commit outside of branch
  0dc47b9 where will that lead us ??

If you want to keep them by creating a new branch, this may
be a good time to do so with:

 git branch <new-branch-name> 0860b65

Switched to branch 'master'
```

Git reminds you of the hash of the commit, in case you don't have it.

the **git stash**

Git’s “cut and paste” functionality

When workflow interruption strikes …

Sometimes we quickly need a clean working tree, but without losing un-committed changes already made to our files. For instance:

- Work on in a different branch (e.g. fix a bug) before finishing work on the current branch.
- Move current edits to another branch (e.g. you started to work in the wrong branch).
- Do a rebase (rebase with un-committed is not allowed).

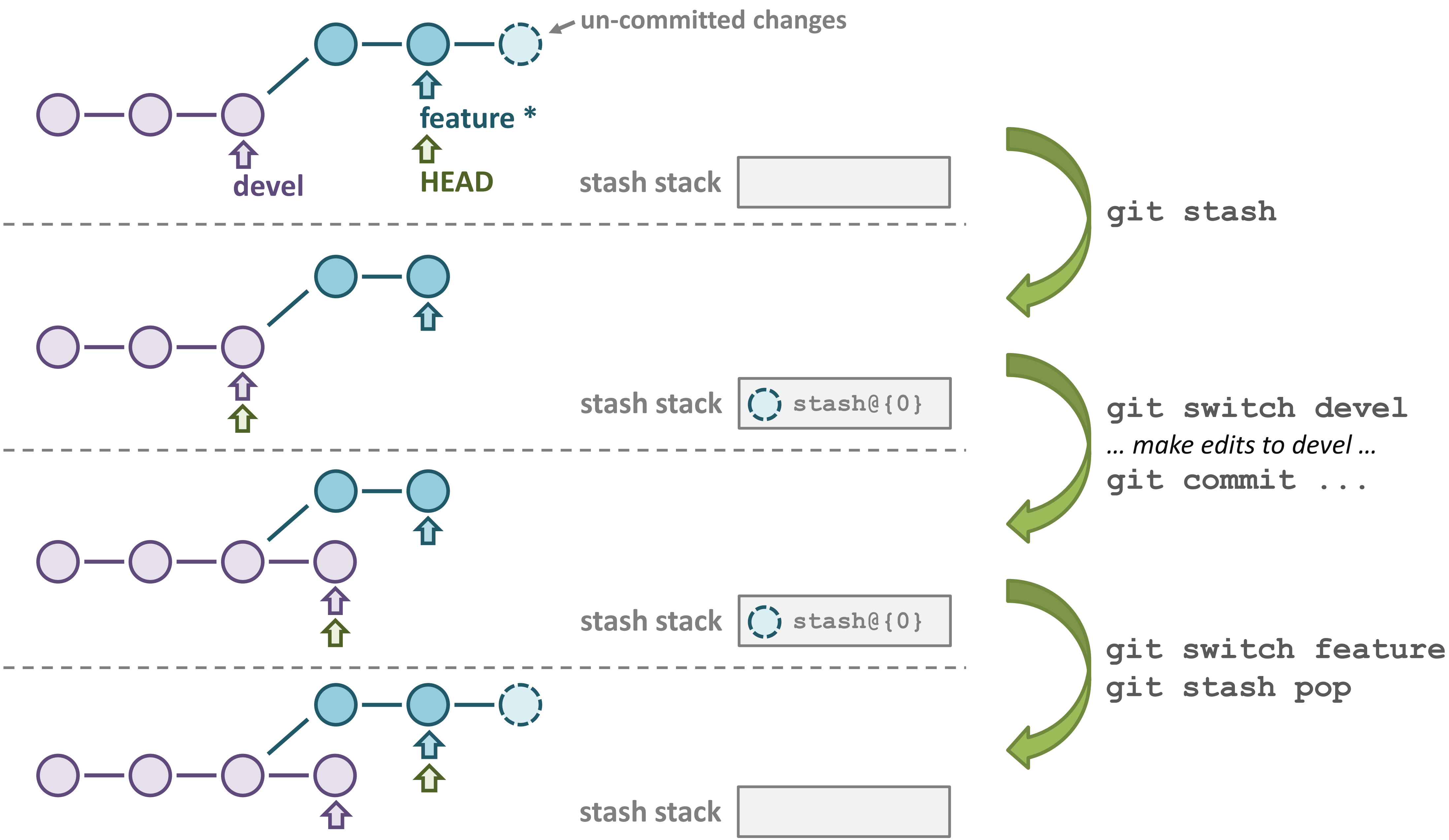
```
git stash
```

⇨ Saves un-committed changes in the working tree (both staged and un-staged) to a “temporary commit“. Then resets the working tree to the current HEAD position (i.e. the last commit in your current branch), leaving a clean working tree.

```
git stash pop
```

⇨ Restores stashed modification, and merges them into the current HEAD. (This can potentially require user input for conflict resolution)

Example: make edits on a different branch while having work in progress.



```
git stash
```

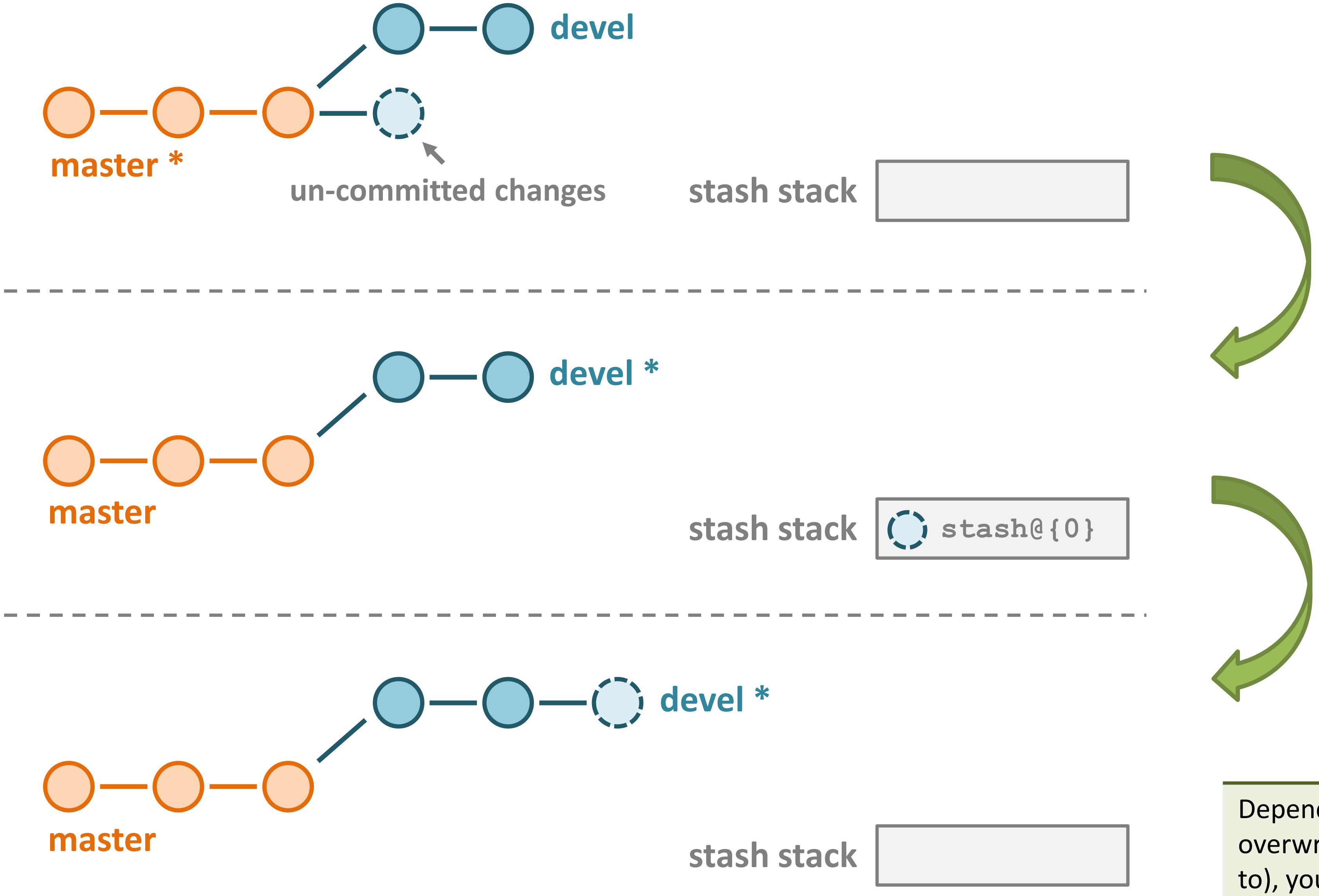
```
git switch devel
```

… make edits to devel …

```
git commit ...
```

```
git switch feature
git stash pop
```

Example: move edits to different branch (e.g. started working on the wrong branch).



```
git stash
git switch devel
```

```
git stash pop
```

Depending on your edits (if they do not overwrite a file on the branch you are switching to), you might be able to switch branches directly without having to do and stashing.

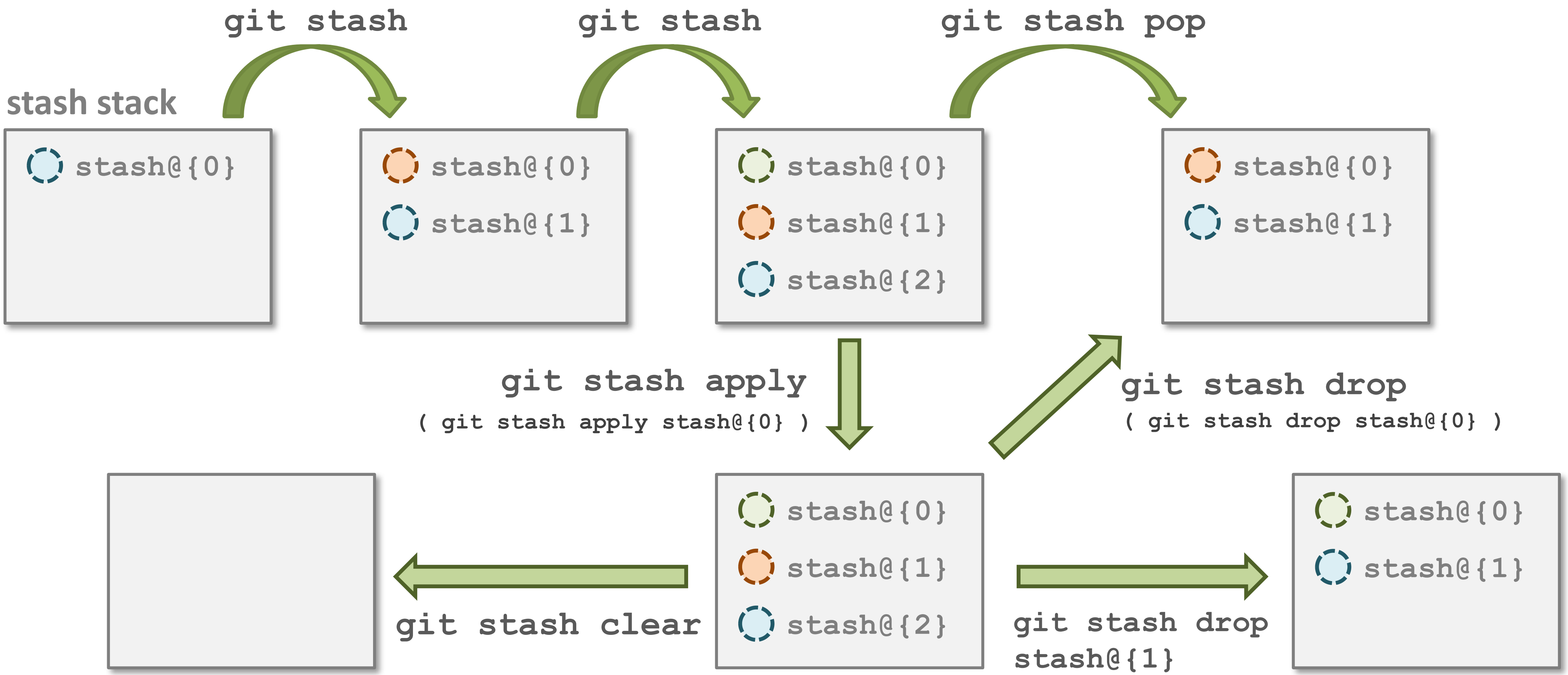
Additional info about git stash…

- More than one set of changes can be stashed (see next slides).
- Although stashed items can in principle remain in the stash for a long time, it's best to view it as a **temporary location**. Don't turn it into an alternate development branch!
- The content of the stash stays local (even if you `git push`), so **there is not backup** for it on a remote.
- Anything done with `git stash` can **also be achieved using branches** (i.e. create new temporary branch and later rebase/merge its content), it's just more convenient to do it with git stash.
- By default **untracked files are not stashed**. To stash them, the `-u/--include-untracked` option must be added.
- By default, both staged and un-staged modifications are stashed. However, **the distinction between staged and unstaged changes is lost upon applying the stash** and all modifications will be un-staged.
  Note: to not include staged changes, the `--keep-index` option can be used.
- `git stash` is actually a shortcut for `git stash save`
  (`save` is the default action for the git stash command).

Supplementary material...

Using multiple stash slots

- `git stash` can actually store multiple stashes.
- `git stash pop` is a shortcut for `git stash apply` + `git stash drop`
- specific stashes can be accessed with `stash@{x}` (where x = stash index)
- `git stash clear` deletes all stashes.



Listing the content of the stash

- List the content of the git stash: `git stash list`

Example:

```
$ git stash list
stash@{0}: WIP on master: 86eae5c Adds new file
stash@{1}: WIP on master: 86eae5c Adds new file
```

- Show the content of a specific stash item. By default, `stash@{0}` is shown. Adding the `-p` option displays the exact content (diff view) of a stash item.

```
git stash show
git stash show -p                # detailed diff view of stash item.
git stash show -p stash@{x}      # show a specific stash item.
```

Supplementary material...

git stash command summary

command	description

`git stash`
`git stash save “message”`
	stash uncommitted changes to a new stash item in the stash@{0} spot.
	An optional “message” can be added.

`git stash pop`
`git stash pop stash@{x}`
	Shortcut for apply + drop.
	By default, stash@{0} is popped. Other stashes can be popped with stash@{x} notation.

`git stash apply`
`git stash apply stash@{x}`
	Merge stashed item into current branch.

`git stash drop`
`git stash drop stash@{x}`
	Delete item from stash. By default, item stash@{0} is deleted.

`git stash list`	List content of stash.

`git stash show`
`git stash show -p`
	Show summary view of stashed item content.
	Show detailed view of stashed item content.

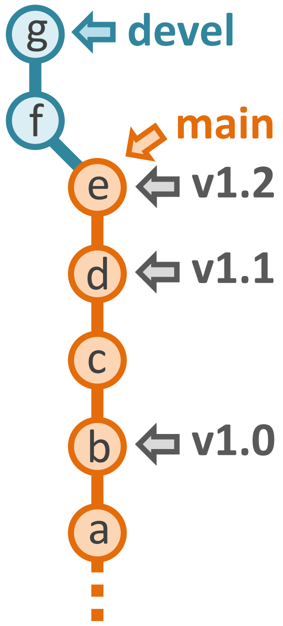
`git stash clear`	Delete all items from stash.

Supplementary material...

git tags

Label important commits

Why use tags ?



Tags are "labels" used to annotate **important commits**.

- Typical use case: tagging commits corresponding to versions. E.g. 1.0.7, v1.0, v2.1, etc.

There are 2 types of tags:

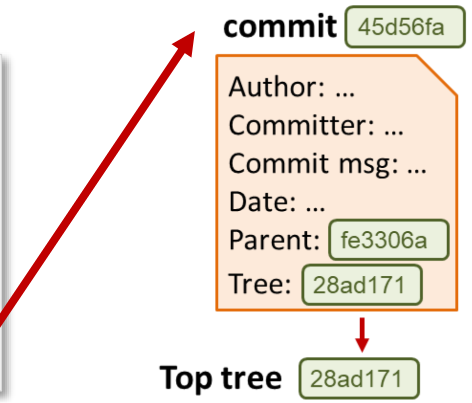
- **Lightweight tags** - pointers to a commit (like a branch).
- **Annotated tags** - pointers to commits with additional metadata:
  - Tagger (person who made the tag).
  - Date and time.
  - Message.

Example of annotated tag metadata

```
tag 12.04
Tagger: Alice Smith <alice@redqueen.org>
Date:   Tue Feb 22 20:44:27 2022 +0100

Version 12.04 LTS (Precise Pangolin)

commit 45d56fa3c75e5e6a67d067e9b8eae1679d3806e7
```

Tag message →

Commit to which the tag is pointing →



Creating tags

If no commit reference is specified, the tag is applied to the current HEAD commit.

Lightweight tag: `git tag <tag name> <commit reference>`

Annotated tag: `git tag -a -m “message” <tag name> <commit reference>`

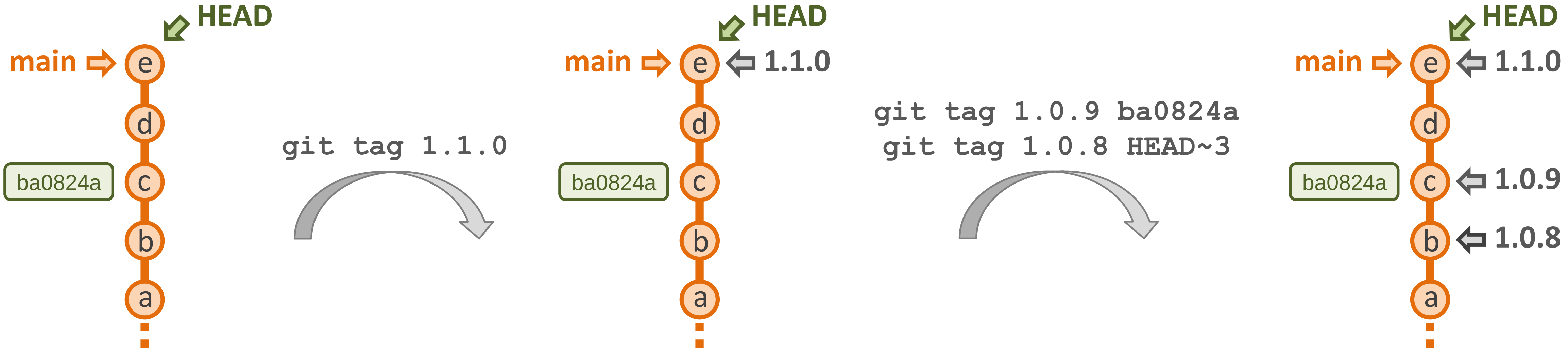
Having a message is compulsory for annotated tags (just like for commits).

Examples:

```
$ git tag 1.1.0
$ git tag 1.0.9 ba0824a
$ git tag 1.0.8 HEAD~3

$ # Create an annotated tag:
$ git tag -a -m "v20.04: Precise Pangolin" 20.04
```

illegal characters in tag and branch names

Spaces and characters such as ,~^:?*[]\ are not allowed in tag and branch names. It is recommended to stick to lowercase letters, numbers, “–”, and “.”.



```
git tag 1.1.0
```

```
git tag 1.0.9 ba0824a
git tag 1.0.8 HEAD~3
```

Listing tags

- List all tags (sorted alphabetically):

```
git tag
```

- List all tags and show their message (for annotated tags):

```
git tag -n
```

- List only tags whose name matches a specific pattern:

```
git tag -l <search pattern>
```

- Show content of a specific tag (annotation and commit content):

```
git show <tag name>
```

- The “adog” command will also show tags:

```
git log --all --decorate --oneline --graph
```

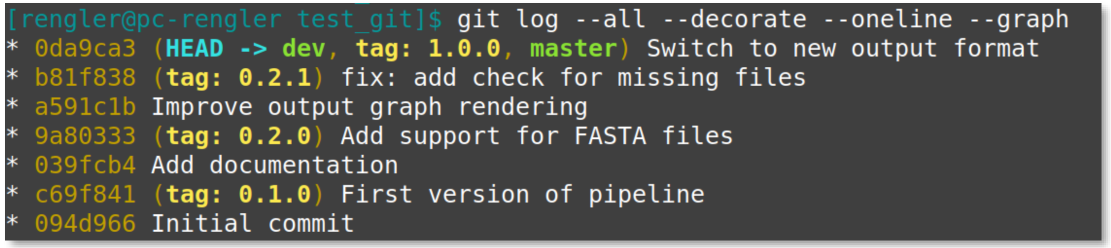
Examples:

```
$ git tag
1.8.4
1.8.5
1.8.5-rc1
2.0.5
```

```
$ git tag -n
12.04   v12.04 LTS Precise Pangolin
12.10   v12.10 Quantal Quetzal
```

```
$ git tag -l 1.8.5*
1.8.5
1.8.5-rc1
```

```
$ git show 2.0.5
tag 12.04
Tagger: Alice Smith <alice@redqueen.org>
Date:   Tue Feb 22 20:44:36 2022 +0100

v12.04 LTS Precise Pangolin

commit 1ba62733c75e5e6a67d067e9b8eae1679d3806e7
Author: Mad Hatter <clocks@wonder.org>
Date:   Tue Feb 22 20:35:09 2022 +0100

    Commit message...

diff --git a/file b/file
…
```



```
[rengler@pc-rengler test_git]$ git log --all --decorate --oneline --graph
* 0da9ca3 (HEAD -> dev, tag: 1.0.0, master) Switch to new output format
* b81f838 (tag: 0.2.1) fix: add check for missing files
* a591c1b Improve output graph rendering
* 9a80333 (tag: 0.2.0) Add support for FASTA files
* 039fcb4 Add documentation
* c69f841 (tag: 0.1.0) First version of pipeline
* 094d966 Initial commit
```

Sharing tags (push to remote)

By default `git push` doesn't upload (push) tags to remote servers.

- You can push a specific tag with: `git push <remote name> <tag name>`

  Example:

  ```
  $ git push origin v2.3
  ```

- You can push **all tags** by adding the `--tags` flag to the push command.

  Example:

  ```
  $ git push origin --tags
  ```

Deleting tags

- To delete a tag from your local repository: `git tag -d <tag name>`

<u>Example</u>:

```
$ git tag -d v3.2
$ git tag -d 12.04
```

⚠ This will not remove the tag from remotes !

- To delete a tag from a remote: `git push <remote name> --delete <tag name>`

<u>Note</u>: this is the same command as for deleting a branch from a remote.

<u>Example</u>:

```
$ git push origin --delete v3.2
```

Checking out tags (revert the working tree to a specific tag)

- Tags are references to a commit, so you can use `git checkout <tag>` to revert the working tree to its recorded state at the specified tag.

Example:

```
$ git checkout v2.0.1
$ git checkout 0.8.2
```

Reminder:

Performing such a checkout will put your repository in **detached HEAD** state:

- You can look at (or use) the “old version”, then switch back to a regular branch.
- If you plan to make changes and add commits to an older version, you can either:
  - Create a new branch rooted at your version tag.
    `git switch -c <new branch> <tag>` or `git checkout -b <new branch> <tag>`
  - Tag the (branchless) new commit your make so it doesn’t get garbage collected.

exercise 3

The backport

exercise 4

The treasure hunt

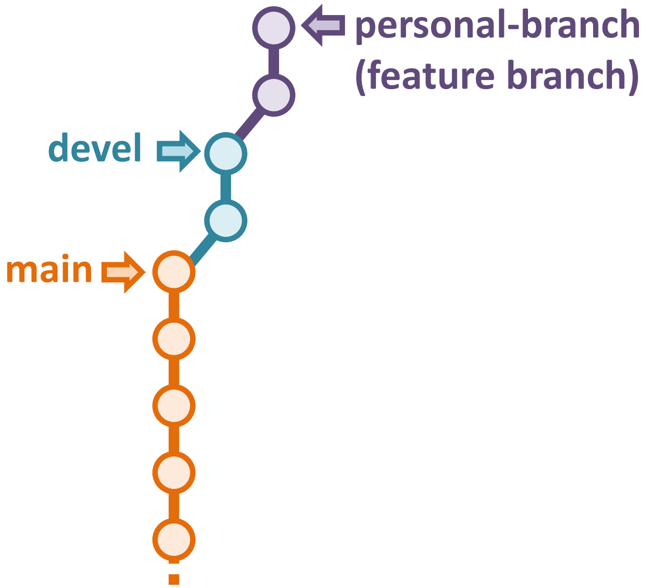
Note: this exercise can be done as exam to the course.

This exercise has helper slides

Introductory notes

While this exercise is somewhat *gameified*, it nevertheless covers many of the important operations and collaborative workflows you would encounter while doing real work:

- Each of the ***quests*** you will complete in this exercise can be seen as the equivalent of adding a ***new feature*** to a software or data analysis pipeline.
- Completing a quest, merging your work into the **main** branch and adding a **tag**, would be the equivalent of making a **new release** of your work/software.

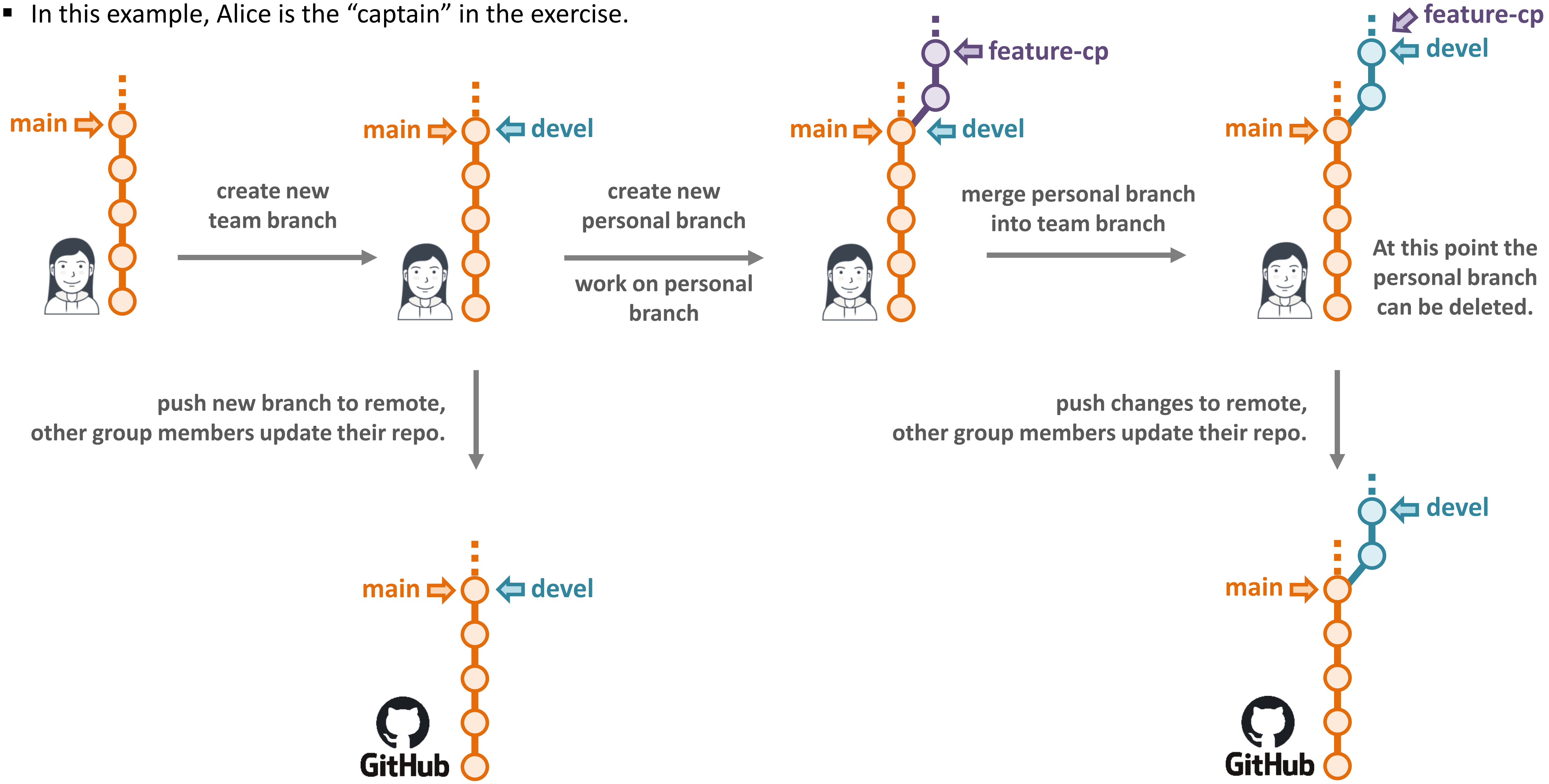


About the branches used in the exercise:

- **main** is the ***production*** branch, i.e. the branch on which only final, production ready, material is published. Do ***not*** work directly on the **main** branch.
- **devel** (for "develop") is the **pre-release** branch where the team will consolidate each "feature" (i.e. each quest of the treasure hunt) before merging it to **main** when a quest is completed.
- Short-lived **personal branches** (feature branches) will be created by each team member to add their work, before merging it into **devel**.
- As this is an exercise, and we do not have much time, the personal branches will only contain 1 (or sometimes 2) commits before they get merged into **devel**, but you can imagine that in a real application more commits would be added.

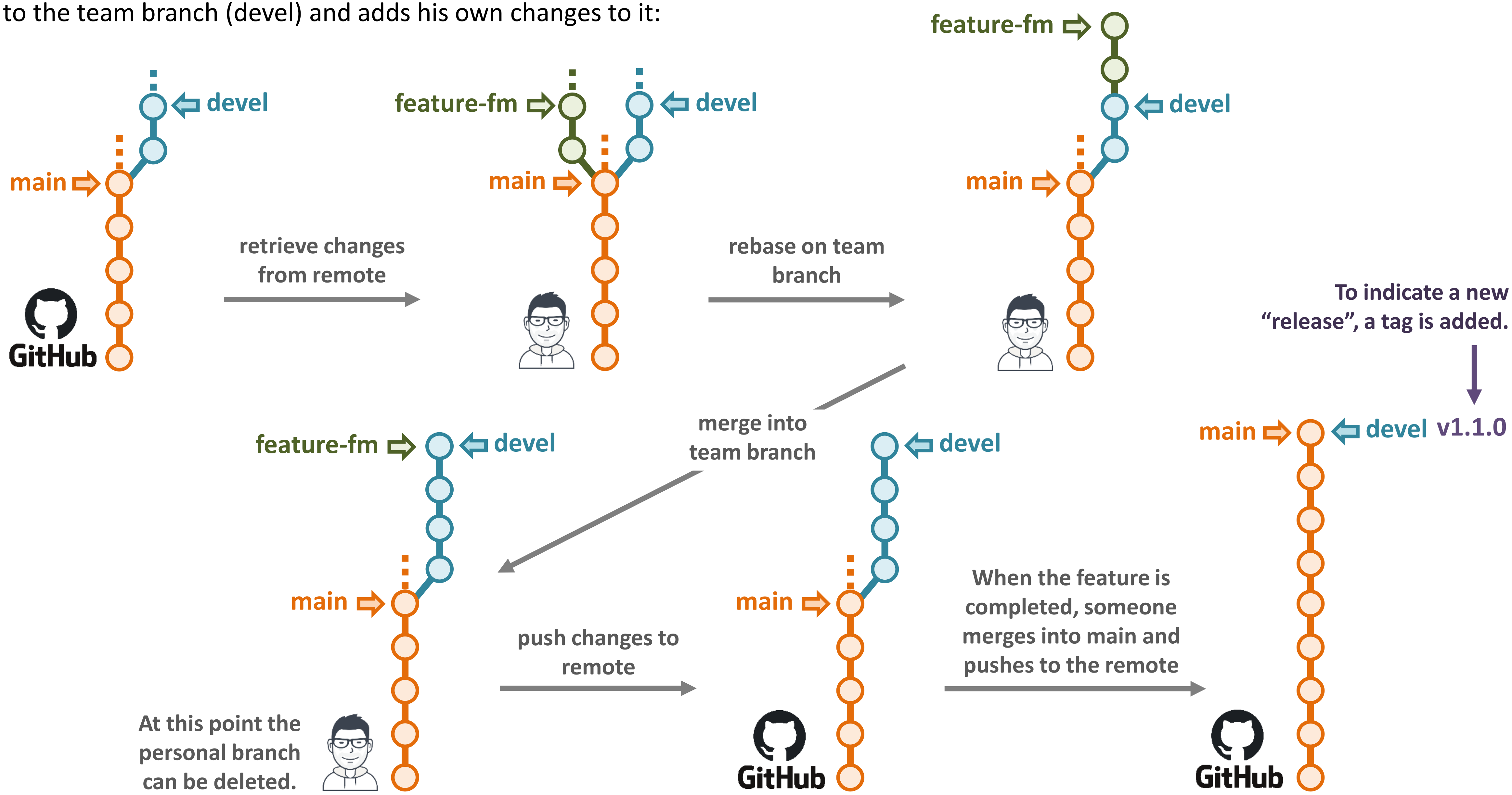
Exercise 4 help: branch – rebase – merge sequence

- One of the objectives in the exercise is to keep a clean and readable history while collaborating.
- This is a suggested procedure when working on a new “feature”.
- In this example, Alice is the “captain” in the exercise.



Exercise 4 help: branch – rebase – merge sequence

- Bob is the “first-mate” of the crew. He retrieves changes made by Alice to the team branch (devel) and adds his own changes to it:



retrieve changes from remote

rebase on team branch

merge into team branch

push changes to remote

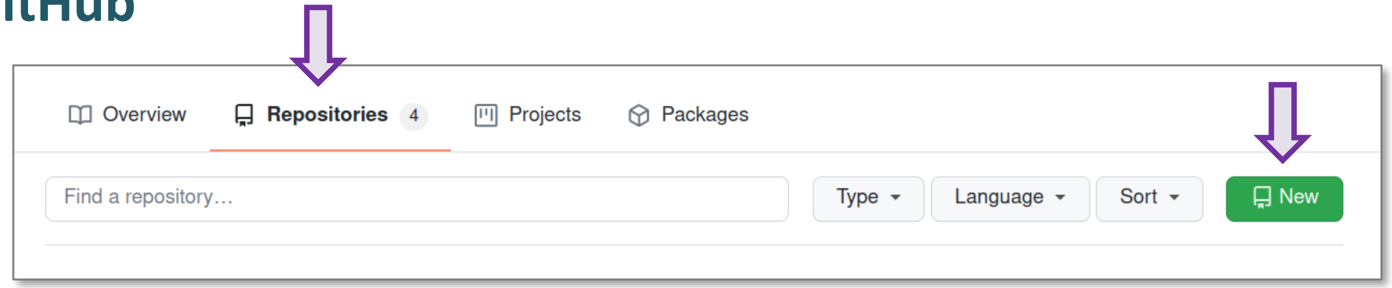
At this point the personal branch can be deleted.

When the feature is completed, someone merges into main and pushes to the remote

To indicate a new “release”, a tag is added.

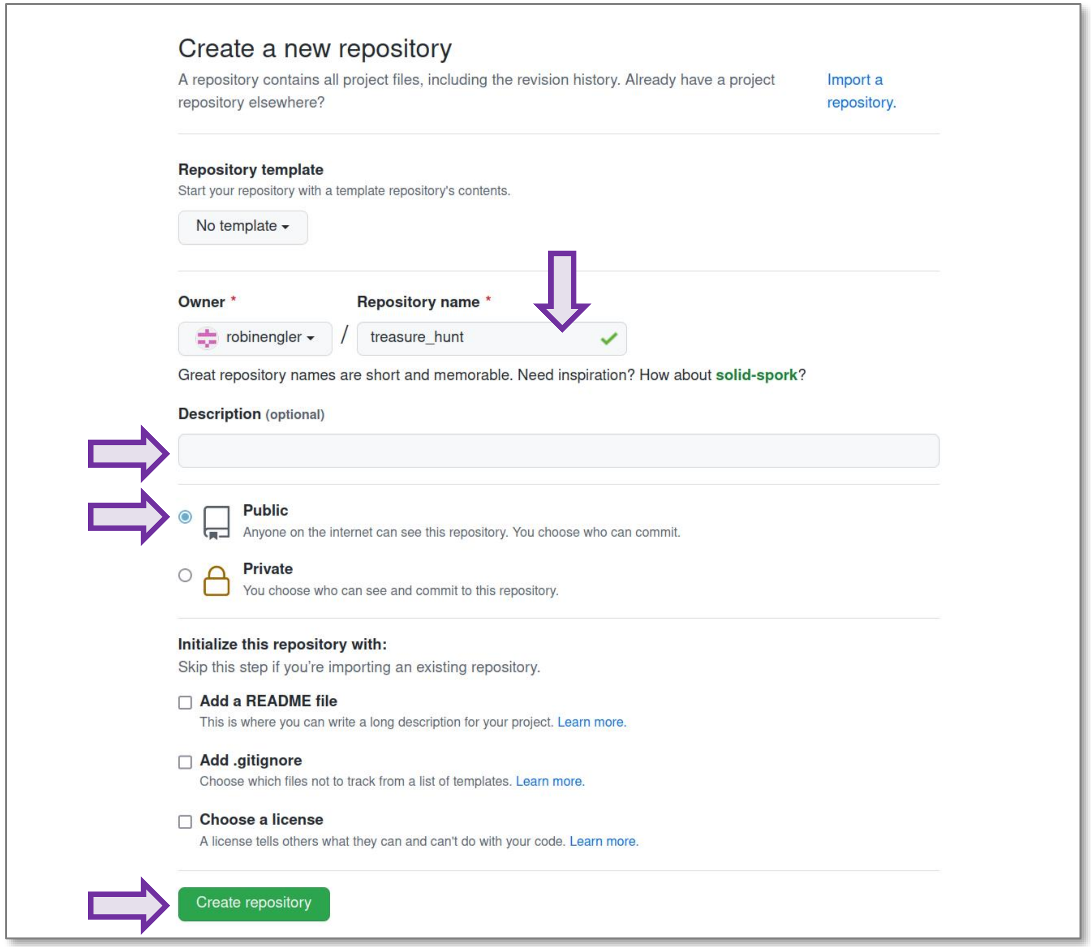
Exercise 4 help: creating a new repo on GitHub

1. In your GitHub account, go to **Repositories** and click on **New** (green button).



2. Create a new repo:

- Enter a **Repository name**.
- Add a short **Description**.
- Make the repo **Public** (default).
- Do **not** initialize the repo, as you will import data from an existing repository (leave all boxes unchecked).
- Click **Create Repository**.



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Repository template

Start your repository with a template repository's contents.

No template

Owner * robinengler / **Repository name** * treasure_hunt ✓

Great repository names are short and memorable. Need inspiration? How about **solid-spork**?

Description (optional)

● **Public**
Anyone on the internet can see this repository. You choose who can commit.

○ **Private**
You choose who can see and commit to this repository.

Initialize this repository with:
Skip this step if you're importing an existing repository.

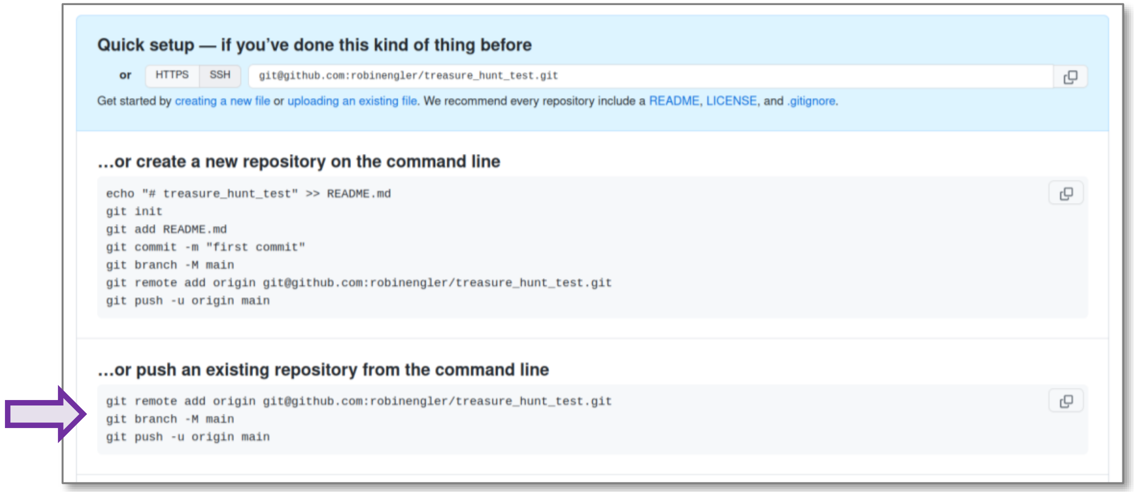
□ **Add a README file**
This is where you can write a long description for your project. Learn more.

□ **Add .gitignore**
Choose which files not to track from a list of templates. Learn more.

□ **Choose a license**
A license tells others what they can and can't do with your code. Learn more.

3. Follow instructions to **push an existing repository….**

Note: the main branch's name is already "main", so you can skip "git branch -M main".



Quick setup — if you've done this kind of thing before

or HTTPS SSH git@github.com:robinengler/treasure_hunt_test.git

Get started by creating a new file or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore.

…or create a new repository on the command line

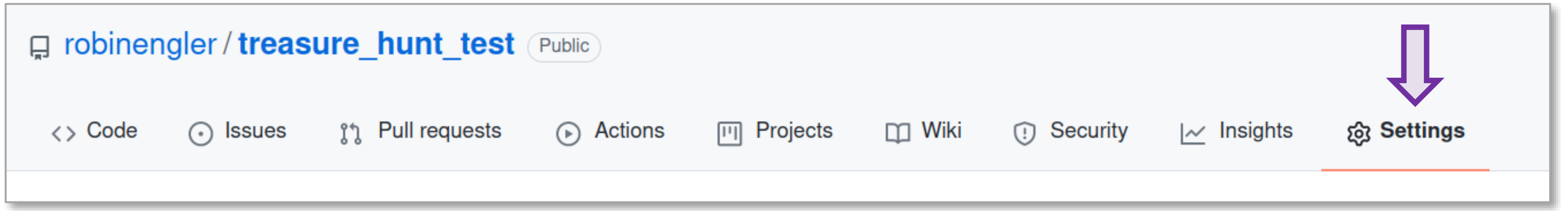
```
echo "# treasure_hunt_test" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin git@github.com:robinengler/treasure_hunt_test.git
git push -u origin main
```

…or push an existing repository from the command line

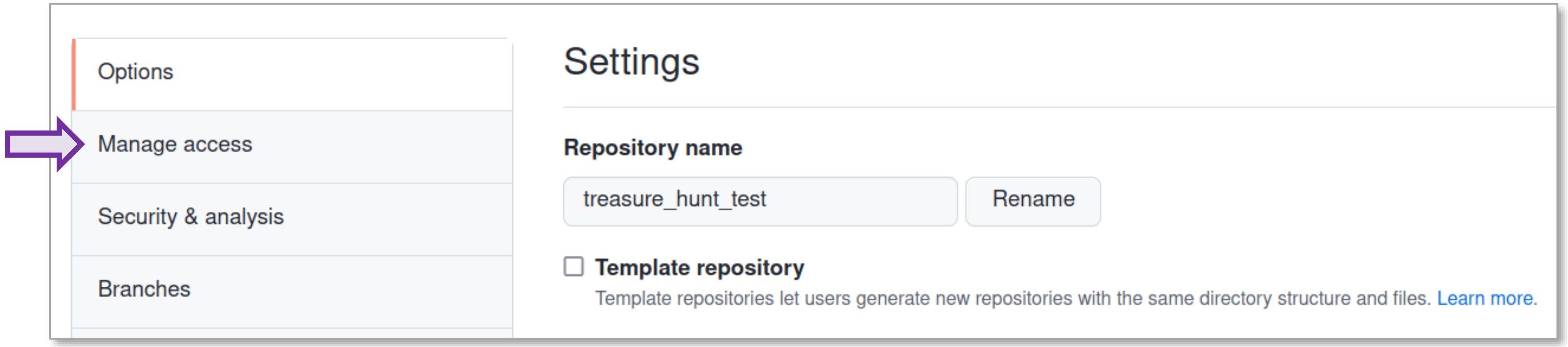
```
git remote add origin git@github.com:robinengler/treasure_hunt_test.git
git branch -M main
git push -u origin main
```

Exercise 4 help: adding members to a GitHub repo.

1. On the homepage of the repo on GitHub, select the **Settings** tab.



2. In the **Settings** tab, click on **Manage access**.



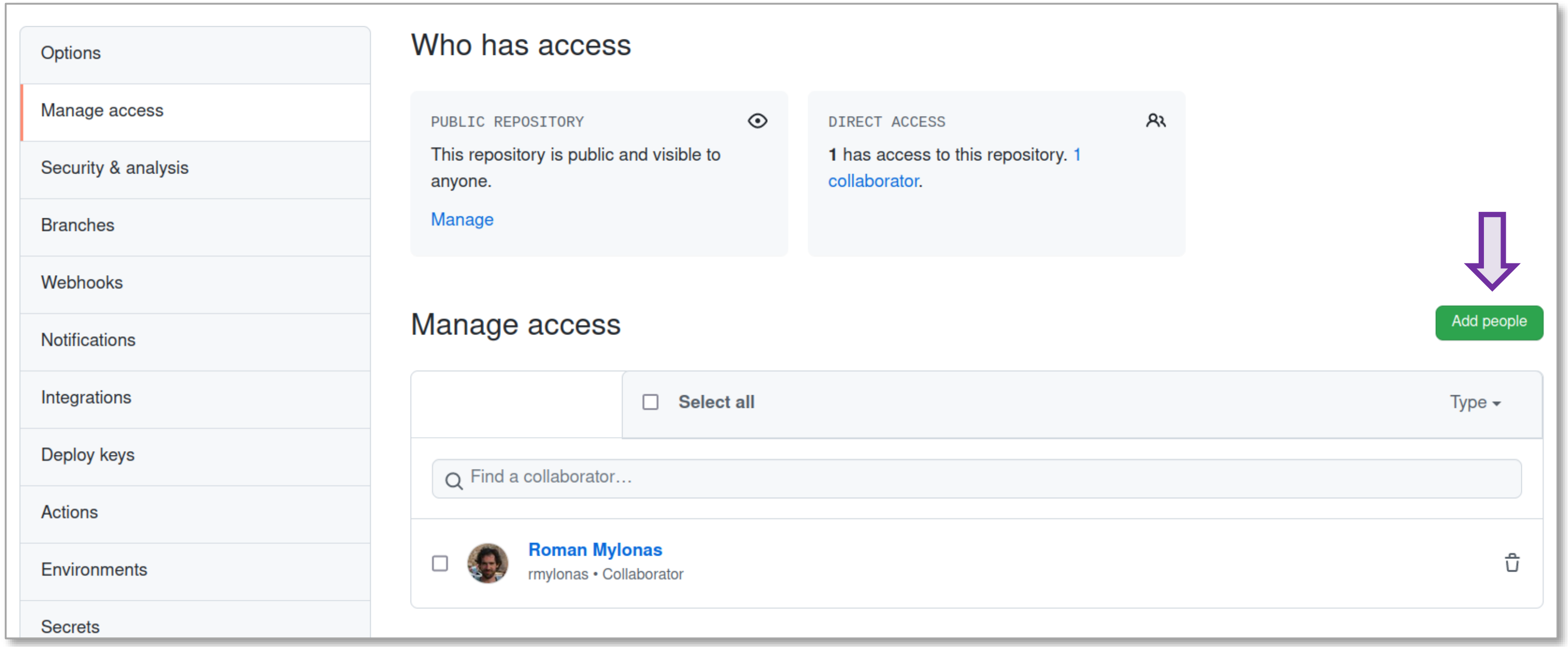
Repository name

treasure_hunt_test

□ Template repository

Template repositories let users generate new repositories with the same directory structure and files. Learn more.

3. Add your team members by clicking on **Add people** (green button) and entering their GitHub user name.



This repository is public and visible to anyone.

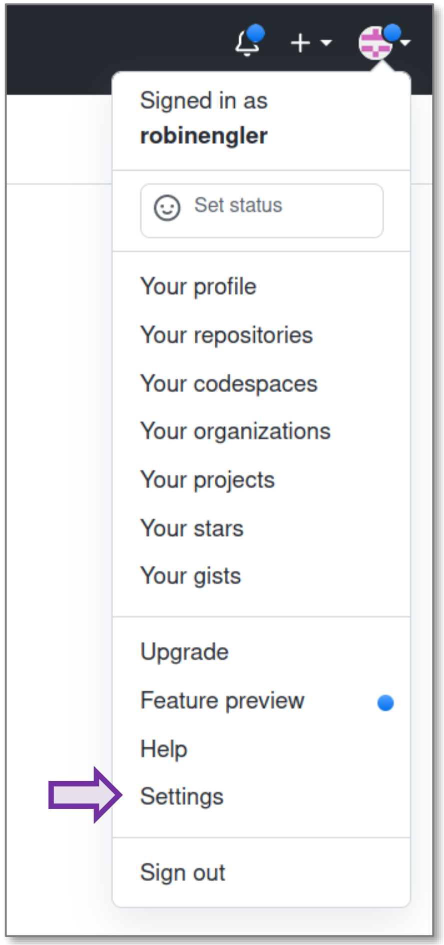
1 has access to this repository. 1 collaborator.

□ Roman Mylonas
rmylonas • Collaborator

Exercise 4 help: generating a “personal access token” on GitHub

In order to push data (commits) to GitHub, you will need a personal access token (PAT).

1. In your user profile (top right), click on Settings.



Signed in as
robinengler

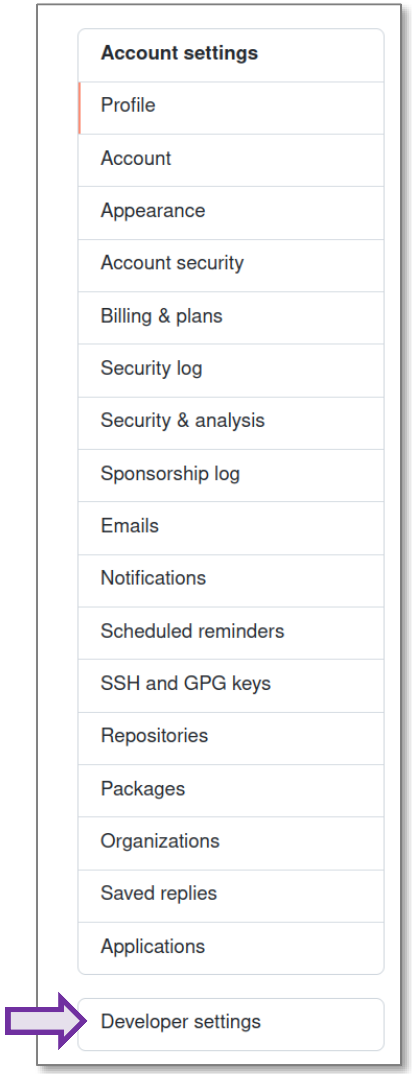
Set status

Your profile
Your repositories
Your codespaces
Your organizations
Your projects
Your stars
Your gists

Upgrade
Feature preview
Help
Settings

Sign out

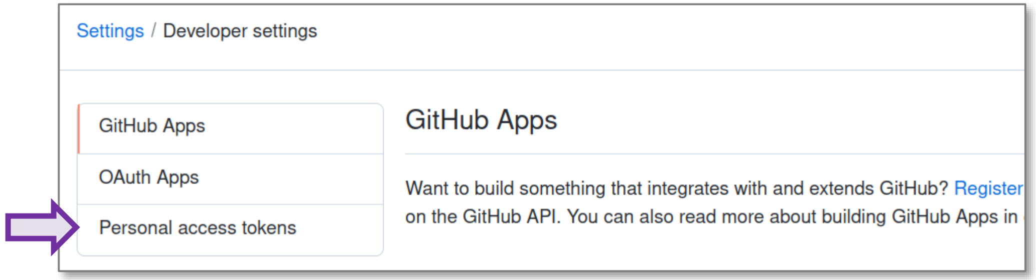
2. In your Account settings, click on Developer settings.



Account settings
Profile
Account
Appearance
Account security
Billing & plans
Security log
Security & analysis
Sponsorship log
Emails
Notifications
Scheduled reminders
SSH and GPG keys
Repositories
Packages
Organizations
Saved replies
Applications

Developer settings

3. In Developer settings, click on Personal access tokens.



Settings / Developer settings

GitHub Apps
OAuth Apps
Personal access tokens

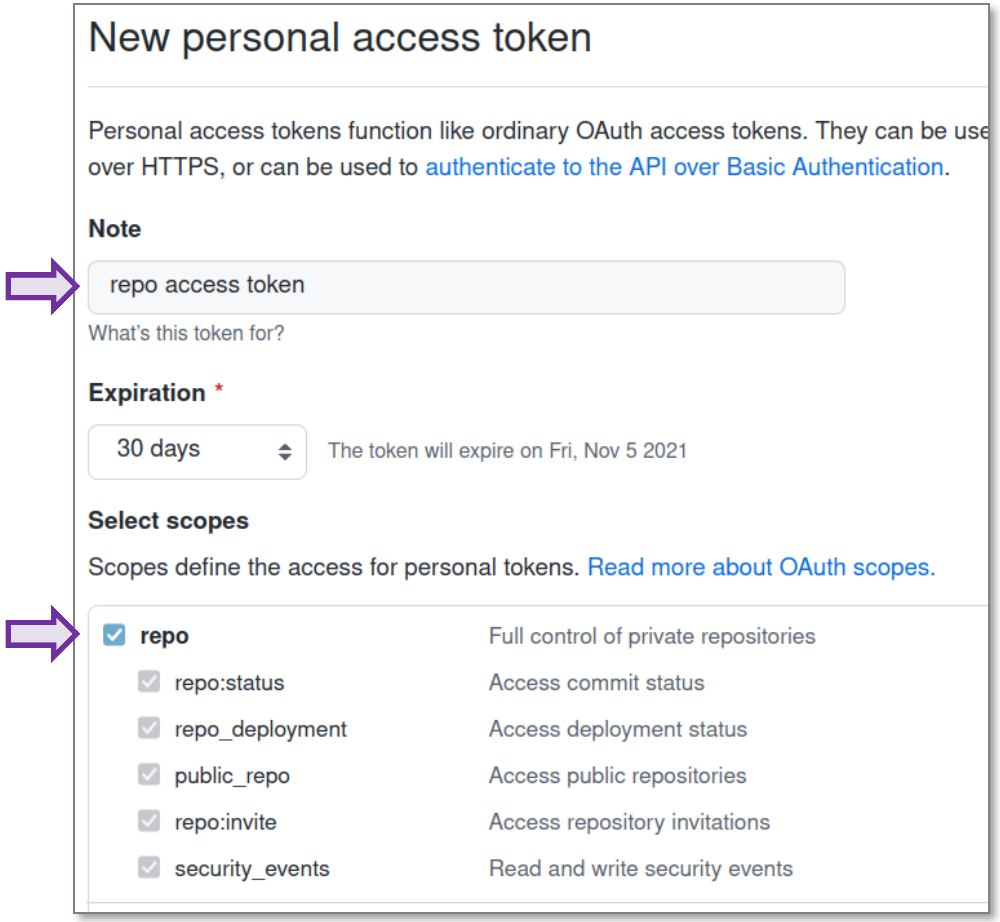
GitHub Apps

Want to build something that integrates with and extends GitHub? Register
on the GitHub API. You can also read more about building GitHub Apps in

Go to next page

Exercise 4 help: generating a “personal access token” on GitHub

4. Add a **Note** (description) to your token and select the **repo** scope checkbox. The click **Generate token**.



New personal access token

Personal access tokens function like ordinary OAuth access tokens. They can be use over HTTPS, or can be used to authenticate to the API over Basic Authentication.

Note

repo access token

What's this token for?

Expiration *

30 days ⬍ The token will expire on Fri, Nov 5 2021

Select scopes

Scopes define the access for personal tokens. Read more about OAuth scopes.

☑ **repo** Full control of private repositories

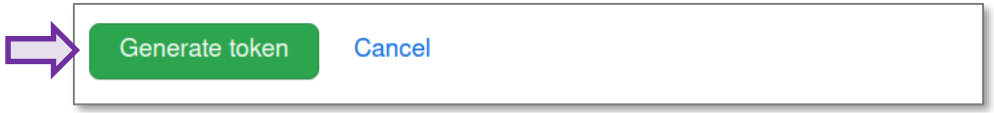
☑ repo:status Access commit status

☑ repo_deployment Access deployment status

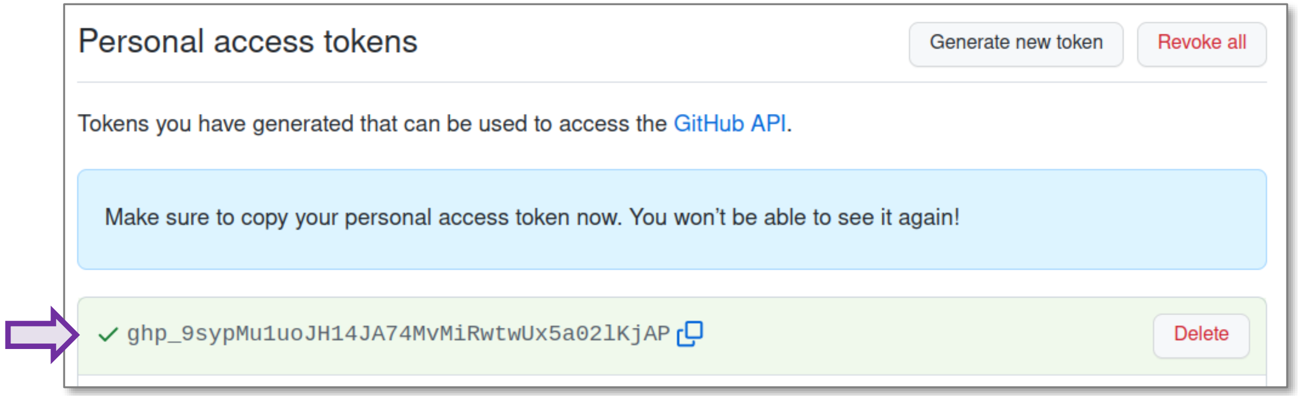
☑ public_repo Access public repositories

☑ repo:invite Access repository invitations

☑ security_events Read and write security events



5. **Copy the personal access token** to a safe locations (for now maybe in a text file, but ideally in a password manager). You will not be able to access it again later.



Personal access tokens

Tokens you have generated that can be used to access the GitHub API.

Make sure to copy your personal access token now. You won't be able to see it again!

✓ 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

6. When you will push content to GitHub for the first time in the project, you will be asked for your user name and password. Instead of the password, enter the **personal access token** you just created.

git submodules

The "symlink" of Git repositories

What are submodules ?

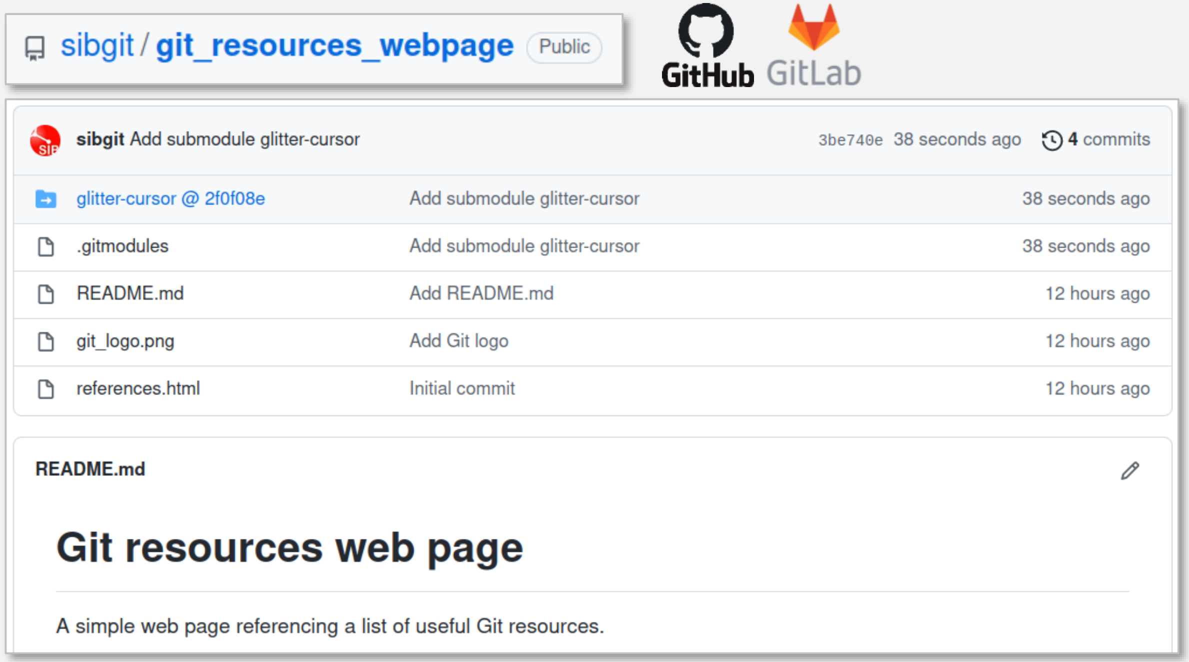
- Git submodules allow keeping a Git repository as a **subdirectory** of another Git repository **while <u>version controlling the version (latest commit) of the nested repository</u>**.
- The "super-project" and the submodule remain independent repos, and have independent remotes.

Super-project

```
git_resources_webpage
│
├── .git
├── glitter-cursor
│   ├── .git
│   ├── glitter.js
│   └── README.md
├── git_logo.png
├── README.md
└── references.html
```

Submodule

Main repository / super-project (repo containing the submodule)



sibgit / **git_resources_webpage** Public

sibgit Add submodule glitter-cursor — 3be740e 38 seconds ago — 4 commits

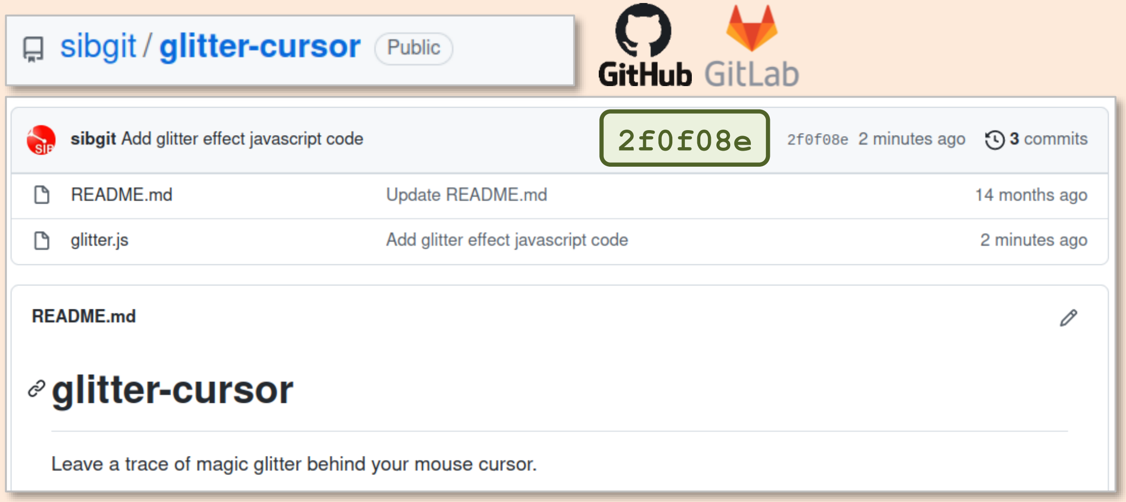
glitter-cursor @ 2f0f08e	Add submodule glitter-cursor	38 seconds ago
.gitmodules	Add submodule glitter-cursor	38 seconds ago
README.md	Add README.md	12 hours ago
git_logo.png	Add Git logo	12 hours ago
references.html	Initial commit	12 hours ago

README.md

Git resources web page

A simple web page referencing a list of useful Git resources.

Subproject (project used as submodule in the super-project)



sibgit / **glitter-cursor** Public

sibgit Add glitter effect javascript code — 2f0f08e — 2f0f08e 2 minutes ago — 3 commits

README.md	Update README.md	14 months ago
glitter.js	Add glitter effect javascript code	2 minutes ago

README.md

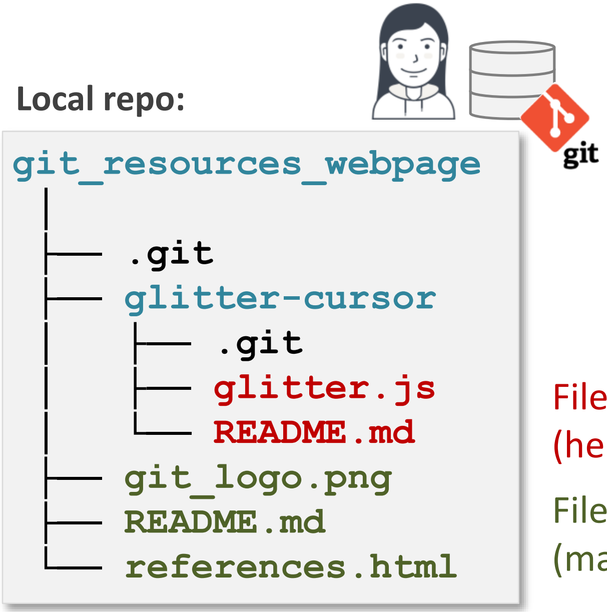
glitter-cursor

Leave a trace of magic glitter behind your mouse cursor.

What are submodules (continued)

- Git submodules are a **reference to another repository** at a **specific commit**. The super-project does not keep track of individual files inside the submodule.

Local repo:



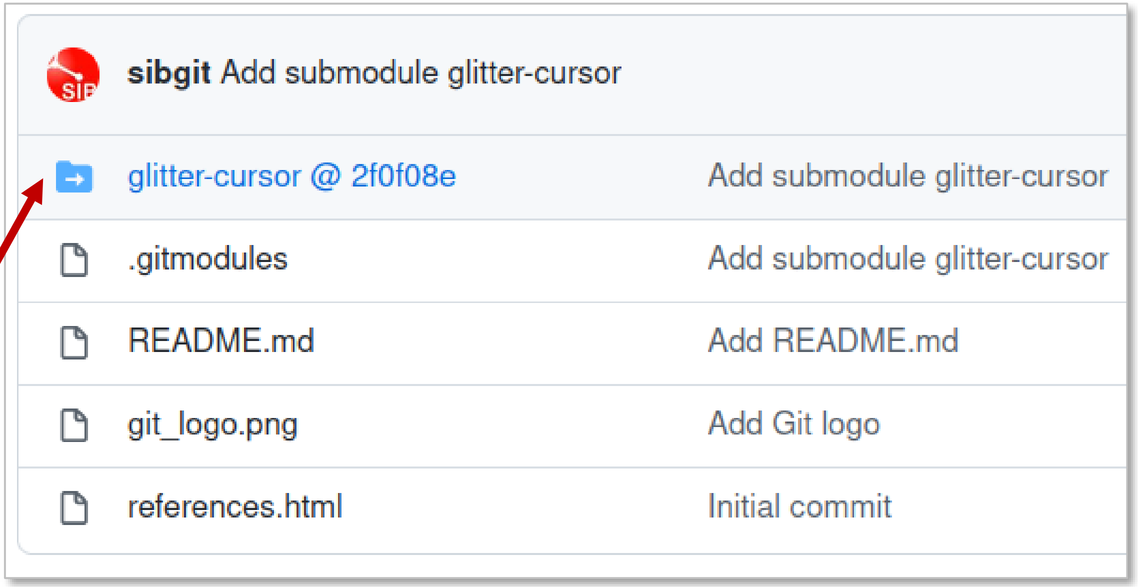
```
git_resources_webpage
│
├── .git
├── glitter-cursor
│   ├── .git
│   ├── glitter.js
│   └── README.md
├── git_logo.png
├── README.md
└── references.html
```

Files tracked by the **subproject** (here used as a submodule)

Files tracked by the **super-project** (main project)

- Because the submodule is **fixed at a specific commit** (unless explicitly changed), the maintainer of the super-project has **full control of which revision of the submodule's code they are using**.

On GitHub/GitLab, submodules are shown with the syntax:
<submodule dir name>@<commit hash>



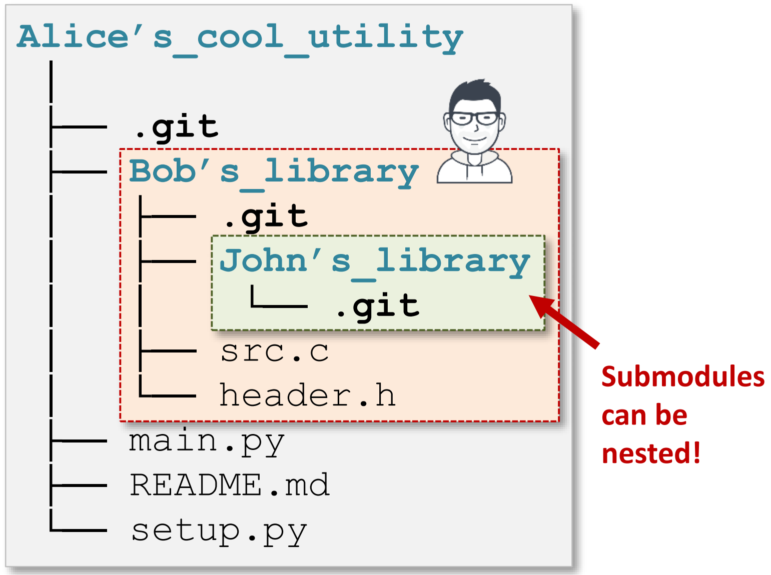
sibgit Add submodule glitter-cursor

glitter-cursor @ 2f0f08e	Add submodule glitter-cursor
.gitmodules	Add submodule glitter-cursor
README.md	Add README.md
git_logo.png	Add Git logo
references.html	Initial commit

Use cases: when to use submodules

- To include external code, i.e. code maintained by someone else (e.g. on GitHub/GitLab), into your project. With Git submodules you can easily integrate it, get updates from the upstream, and stay in control of when the external code should be updated. Can also be used to re-use one of your own repos in multiple projects.
- To make public only a part of a project. You can put the part of your code/files that you want to make public in a submodule (with public access), and keep the rest of the code in a private repository.
- Large project that uses multiple subprojects maintained independently.

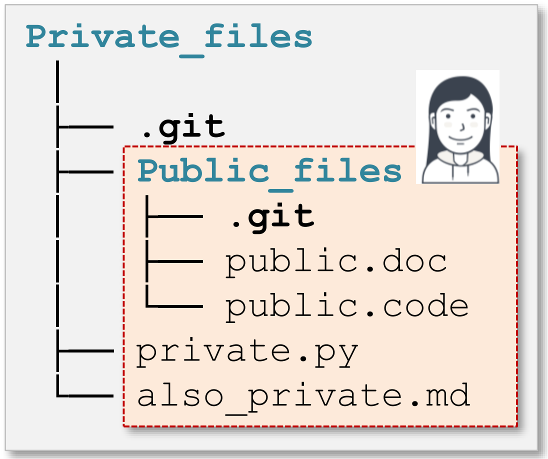
Alice uses a library maintained by Bob as a submodule



```
Alice's_cool_utility
│
├── .git
├── Bob's_library
│   ├── .git
│   ├── John's_library
│   │   └── .git
│   ├── src.c
│   └── header.h
├── main.py
├── README.md
└── setup.py
```

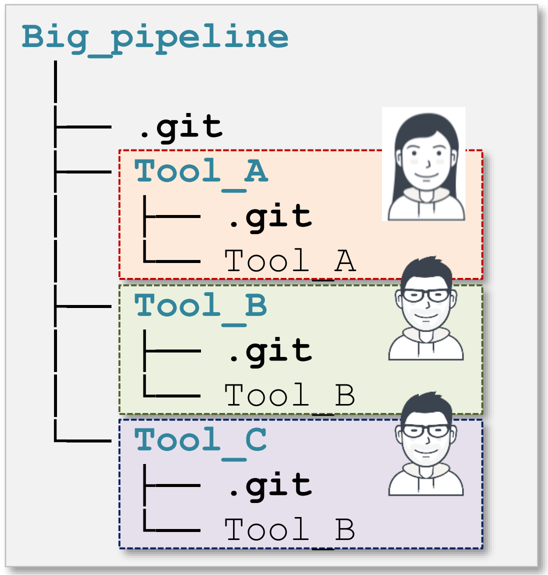
Submodules can be nested!

Alice wants to mix public and private files in a project.



```
Private_files
│
├── .git
├── Public_files
│   ├── .git
│   ├── public.doc
│   └── public.code
├── private.py
└── also_private.md
```

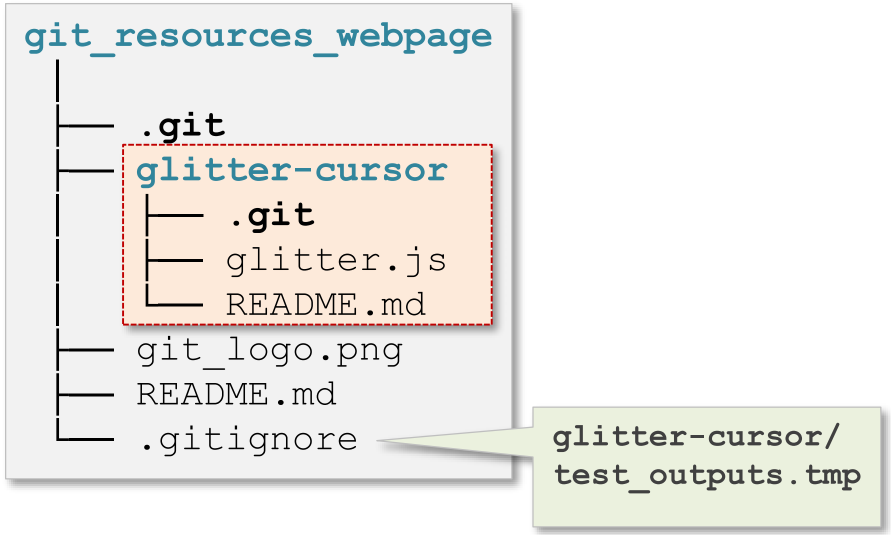
Large pipeline with multiple collaborators.



```
Big_pipeline
│
├── .git
├── Tool_A
│   ├── .git
│   └── Tool_A
├── Tool_B
│   ├── .git
│   └── Tool_B
└── Tool_C
    ├── .git
    └── Tool_B
```

When NOT to use submodules

- Don’t use submodules when not really needed, monolithic repositories are simpler to maintain.
- If you have a sub-project that you want to use in multiple projects, it might be more efficient to create a package instead. Most programming languages have a dedicated package managers/repositories (CRAN for R, npm for javascript, PyPI for Python, etc).
- If you simply want to have a nested Git repos on your local machine (but with no link between them), you can simply add the nested repo to the `.gitignore` file of the higher-level repo.



```
git_resources_webpage
│
├── .git
├── glitter-cursor
│   ├── .git
│   ├── glitter.js
│   └── README.md
├── git_logo.png
├── README.md
└── .gitignore
```

```
glitter-cursor/
test_outputs.tmp
```

If all you want is keeping a Git repo inside another one on your local computer with no link between them… you don’t need submodules – save yourself the hassle!

Adding/registering a submodule

To add/register a new submodule inside a Git repo:

```
git submodule add <URL of submodule repository>
```

This will:

- Add a new directory named after the submodule’s repo name.
- Download the content of the submodule corresponding to the latest commit (on the default branch) into that directory.
- Create a `.gitmodules` file at the root of the super-project.

.gitmodules

```
[submodule ”my-submodule"]
    path = my-submodule
    url = https://github.com/some-user/my-submodule.git
```

Local path of submodule

URL of submodule

- Initialize the submodule in the `.git/config` file.

.git/config

```
[submodule “my-submodule"]
   url = https://github.com/some-user/my-submodule.git
   active = true
```

“active = true” --> module is initialized

- If you add multiple submodules, you will have multiple entries in `.gitmodules` .
- `.gitmodules` should be version controlled, so that other people who clone the project know where the submodule projects are from (Git stages this file by default when adding a new submodule).

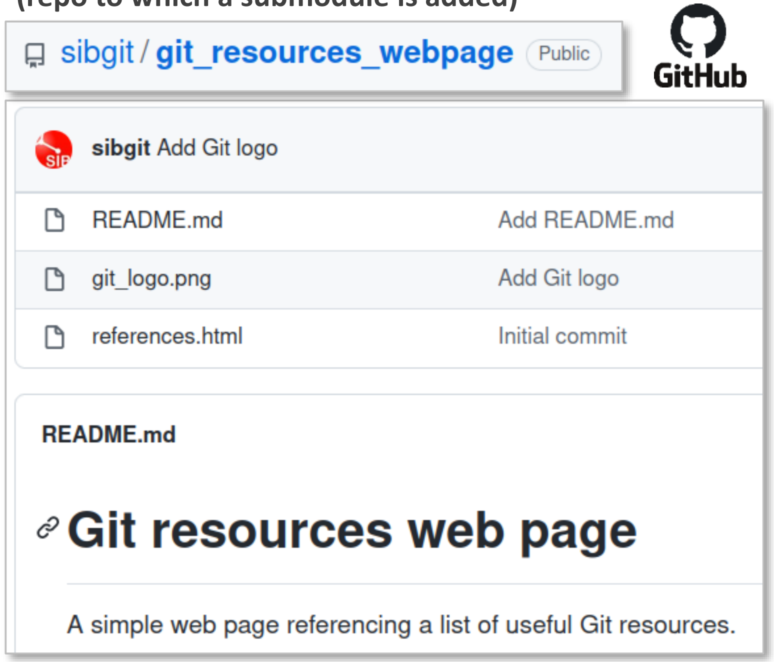
Submodule with custom name:

- Set custom name when adding submodule:
  `git submodule add <URL> <name>`
- Rename an exiting submodule:
  `git mv <submodule name> <submodule new name>`

Adding a submodule: example

Adding “glitter-cursor” as a submodule to “git_resources_webpage”

Main repository / super-project
(repo to which a submodule is added)



sibgit / **git_resources_webpage** Public

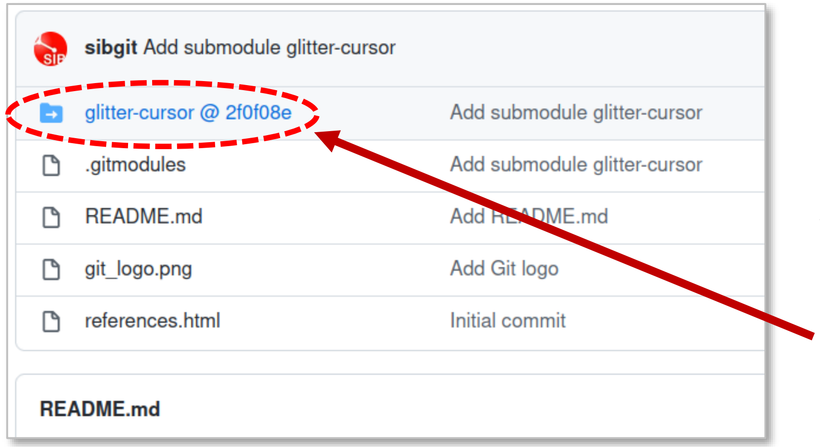
sibgit Add Git logo

README.md	Add README.md
git_logo.png	Add Git logo
references.html	Initial commit

README.md

Git resources web page

A simple web page referencing a list of useful Git resources.



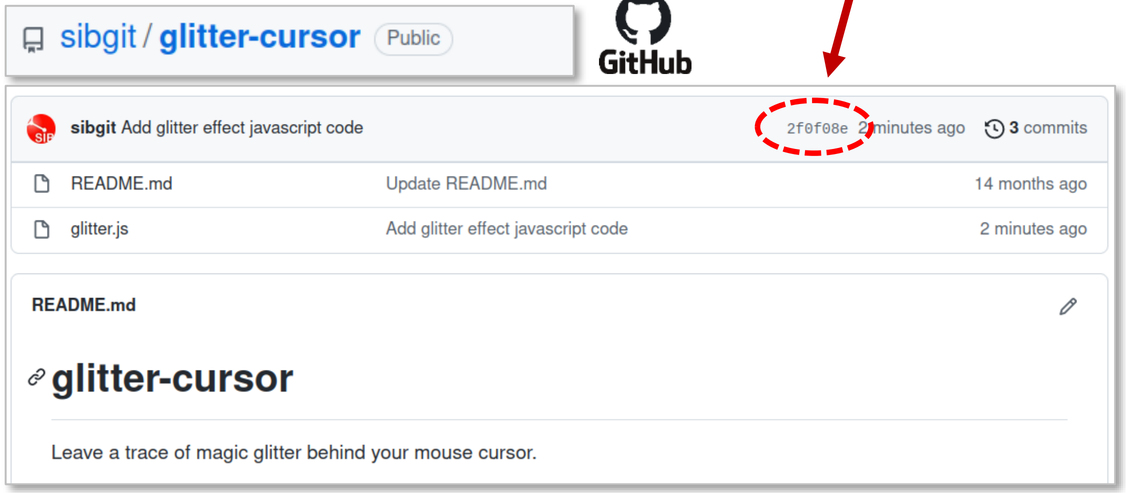
sibgit Add submodule glitter-cursor

glitter-cursor @ 2f0f08e	Add submodule glitter-cursor
.gitmodules	Add submodule glitter-cursor
README.md	Add README.md
git_logo.png	Add Git logo
references.html	Initial commit

README.md

Subproject
(used as submodule in the super-project)

Repo is currently at commit `2f0f08e`



sibgit / **glitter-cursor** Public

sibgit Add glitter effect javascript code 2f0f08e 2 minutes ago 3 commits

README.md	Update README.md	14 months ago
glitter.js	Add glitter effect javascript code	2 minutes ago

README.md

glitter-cursor

Leave a trace of magic glitter behind your mouse cursor.

```
git submodule add https://.../glitter-cursor.git
git commit -m "Add submodule glitter-cursor"
git push
```

Icon and syntax indicating a submodule, which is pointing at `2f0f08e`

When a new submodule is added, it points at the latest commit of the submodule’s online repository.

How Git keeps track of the submodule's version: some more details.

Adding "glitter-cursor" as a submodule to "git_resources_webpage"

```
$ cd git_resources_webpage
$ git submodule add https://github.com/sibgit/glitter-cursor.git
Cloning into '/home/.../git_resources_webpage/glitter-cursor'...
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 9 (delta 0), reused 3 (delta 0), pack-reused 0
Receiving objects: 100% (9/9), done.
```

`Git submodule add` does the following:

- Create a new directory named "glitter-cursor".
- Download the content of "glitter-cursor" corresponding to the latest commit (on the default branch).
- Create a `.gitmodules` file.

```
[submodule "glitter-cursor"]
    path = glitter-cursor
    url = https://github.com/sibgit/glitter-cursor.git
```

Local path of submodule

URL of submodule

- Initialize the submodule in the `.git/config` file.

```
[remote "origin"]
  url = https://github.com/sibgit/git_resources_webpage.git
  fetch = +refs/heads/*:refs/remotes/origin/*
[branch "main"]
  remote = origin
  merge = refs/heads/main
[submodule "glitter-cursor"]
  url = https://github.com/sibgit/glitter-cursor.git
  active = true
```

Section that was added

"active = true" --> module is initialized

How does Git keep track of the submodule's version ?

```
$ git status
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
          new file:   .gitmodules
          new file:   glitter-cursor
```

```
$ git diff --cached
diff --git a/.gitmodules b/.gitmodules
--- /dev/null
+++ b/.gitmodules
@@ -0,0 +1,3 @@
+[submodule "glitter-cursor"]
+       path = glitter-cursor
+       url = https://github.com/sibgit/glitter-cursor.git
diff --git a/glitter-cursor b/glitter-cursor
--- /dev/null
+++ b/glitter-cursor
@@ -0,0 +1 @@
+Subproject commit 2f0f08e991d828dd27cf399c0b88edaaa48a2bf9
```

2f0f08e

- The submodule is tracked/added as a "virtual file" to the index.
- This "virtual file" contains the **commit ID** (SHA-1 checksum) to which the submodule is pointing (and nothing else).
- Individual files in the submodule are not tracked by the super-project.

Clone a repository with submodules

```
git clone <repository>
git submodule init
git submodule update
```

or

```
git clone <repository>
git submodule update --init --recursive
```

or

```
git clone --recurse-submodules <repository>
```

Shortcut to clone, initialize and update all submodules.

This is what you will want to use in most situations.

- After cloning a repository that contains submodules, there will only be an empty directory for the submodules: their content is not automatically downloaded!
- You have to initialize* the local configuration files with: `git submodule init`
- Now the content of submodule(s) can be retrieved** with: `git submodule update`
- `--recursive` / `--recurse-submodules` means that the command also applies to nested submodules (submodules within submodules).

Notes:

- By default, the commands `git submodule init/update` apply to all submodules of a project. To apply them only to a specific submodule, the name of the submodules can be passed: e.g. `git submodule init <submodule name>`
- What does ***initialize** a submodule mean, and what exactly does `git submodule init` do?
  When Git initializes a submodule, it creates an entry for it in the `.git/config` file of the superproject repo and marks it as “active = true”.
  When working on a large project with many submodules, this makes it e.g. possible to only initialize those submodules that are really needed for your work.

`.git/config`

```
[submodule "glitter-cursor"]
    active = true
    url = https://github.com/sibgit/glitter-cursor.git
```

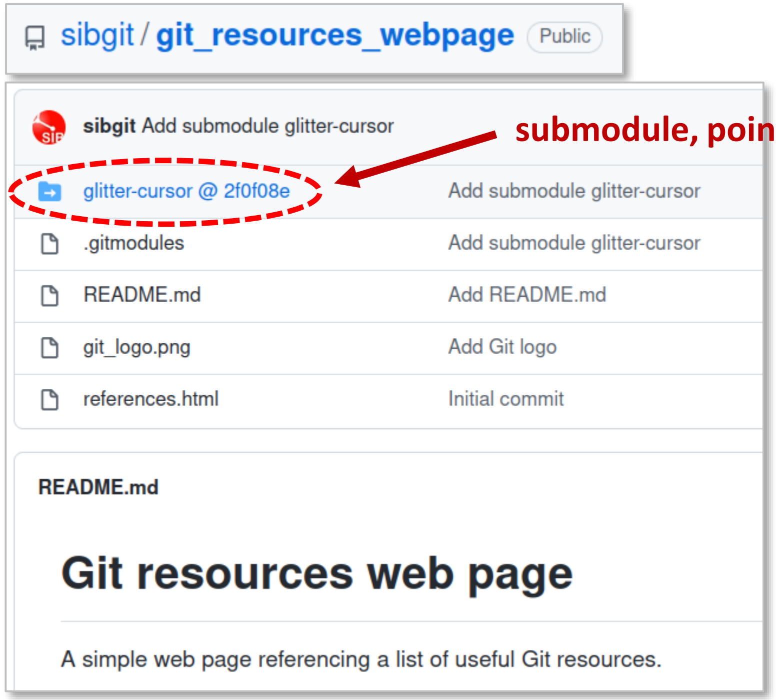
- The meaning of ****update** in `git submodule update` is to fetch updates in submodules and update the working tree of the submodules to the revision expected by the superproject. It does not mean to update the submodules to their latest version.

Clone a repository with submodules: example

Cloning “git_resources_webpage” that contains the submodule “glitter-cursor”.

GitHub GitLab

Online main repository / super-project
(repo that contains a submodule)



sibgit / **git_resources_webpage** Public

sibgit Add submodule glitter-cursor

glitter-cursor @ 2f0f08e	Add submodule glitter-cursor
.gitmodules	Add submodule glitter-cursor
README.md	Add README.md
git_logo.png	Add Git logo
references.html	Initial commit

README.md

Git resources web page

A simple web page referencing a list of useful Git resources.

submodule, pointing at `2f0f08e`

```
git clone
https://.../git_resources_webpage.git
```

Local copy of repository

git

```
git_resources_webpage
│
├── glitter-cursor
├── git_logo.png
├── README.md
└── references.html
```

Directory is empty !

```
git submodule init
```

Initializes/activates the submodule(s) in `.git/config`

```
git submodule update
–init --recursive
```

```
git submodule update
```

Downloads submodule content

```
git_resources_webpage
│
├── glitter-cursor
│   ├── glitter.js
│   └── README.md
├── git_logo.png
├── README.md
└── references.html
```

Now the files of the submodule are locally available.

Shortcut !

```
git clone --recurse-submodules
https://.../git_resources_webpage.git
```

Cloned submodules are (by default) in detached HEAD state

- After cloning a repo with submodules, the submodules are in **detached HEAD** state.
- To make it point to a branch you have to explicitly checkout (switch to) that branch.

```
$ cd glitter-cursor
$ git status
HEAD detached at 2f0f08e
$ git switch main
```

Commit the submodule is currently pointing at.

```
git_resources_webpage
│
├── .git
├── glitter-cursor
│   ├── .git
│   ├── glitter.js
│   └── README.md
├── git_logo.png
├── README.md
└── references.html
```

Supplementary material...

Update a repository with submodules (git pull on the super-project)

Similarly to `git clone`, running `git pull` in the super-project (the main project that hosts the submodule) does not automatically update the submodules’ content. You need to either:

```
git pull
git submodule update --init --recursive
```

← Downloads the submodule’s updated content

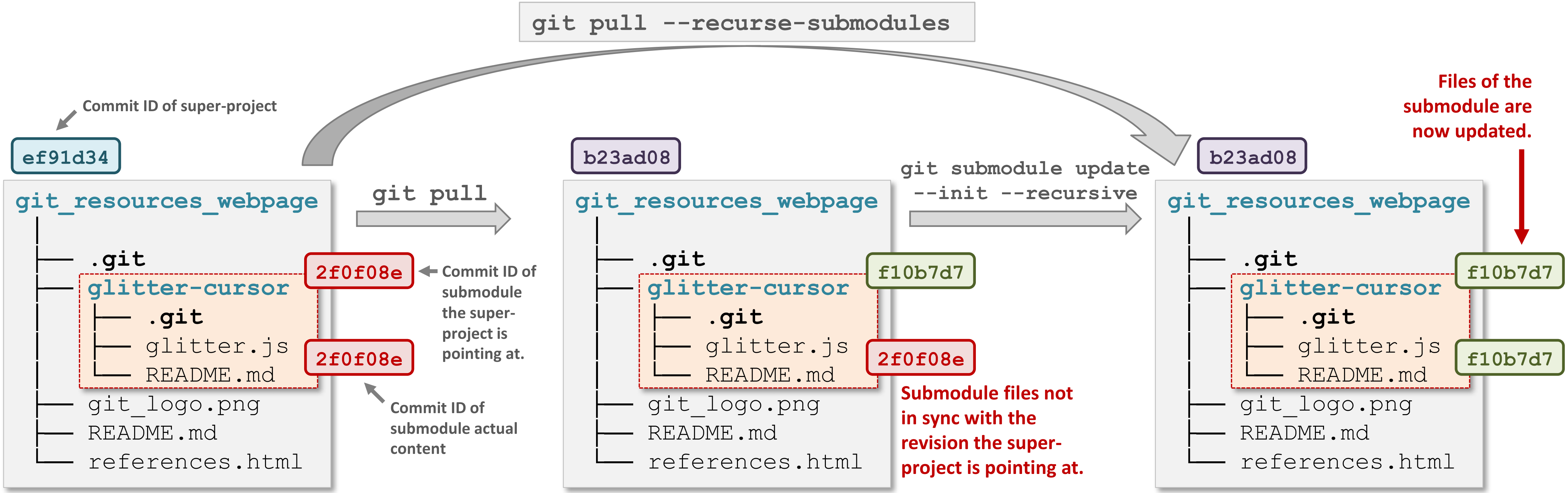
or

```
git pull --recurse-submodules
```

Shortcut !

This is what you will want to use in most situations.

Important: these are commands to run in the super project!



```
git pull --recurse-submodules
```

Commit ID of super-project

```
ef91d34
git_resources_webpage
├── .git
├── glitter-cursor      2f0f08e
│   ├── .git
│   ├── glitter.js      2f0f08e
│   └── README.md
├── git_logo.png
├── README.md
└── references.html
```

Commit ID of submodule the super-project is pointing at.

Commit ID of submodule actual content

`git pull` →

```
b23ad08
git_resources_webpage
├── .git
├── glitter-cursor      f10b7d7
│   ├── .git
│   ├── glitter.js      2f0f08e
│   └── README.md
├── git_logo.png
├── README.md
└── references.html
```

Submodule files not in sync with the revision the super-project is pointing at.

`git submodule update --init --recursive` →

```
b23ad08
git_resources_webpage
├── .git
├── glitter-cursor      f10b7d7
│   ├── .git
│   ├── glitter.js      f10b7d7
│   └── README.md
├── git_logo.png
├── README.md
└── references.html
```

Files of the submodule are now updated.

Working with submodules

- Submodules are regular Git repos. Once inside, you can run the same Git commands as you would on any repo.

Example:

```
$ cd glitter-cursor

# We are now in the submodule directory.
$ git status
$ git add ...
$ git commit ...
$ git push
```

- The super-project does not keep track of individual files in the submodule: it only keeps track of the commit to which it points.

  However, the super-project will detect when changes are made inside a submodule (but not exactly which changes).

Example: files were added/modified in the submodule.

```
$ git status   # run in the super-project's root!

Changes not staged for commit:
      modified: glitter-cursor (modified content,
                               untracked content)
```

Example: one or more new commits in submodule.

```
$ git status   # run in the super-project's root!

Changes not staged for commit:
      modified: glitter-cursor (new commits)
```

- To run the same tasks on multiple submodules, there is the handy command:

```
git submodule foreach "git command"
```

Example:

```
$ git submodule foreach "git status"
$ git submodule foreach "git log --oneline"
Entering 'glitter-cursor'
2f0f08e (HEAD -> main) Add glitter effect code
841e83a Update README.md
b0b66f8 Initial commit
```

Making changes to a submodule (modifying the content of the submodule)

Let’s assume we want to **modify the content** of a submodule, for instance:

- Update the submodule’s content to a newer version.
- Make changes to files in the submodule.
- Point the submodule at an older version.

We proceed as follows:

1. Make the desired changes in the submodule.
   If needed, pull/push the changes from/to the submodule’s remote.

Commands run in the **submodule**:

```
$ cd glitter-cursor
```

```
$ git pull
```

```
$ git add ...
$ git commit ...
$ git push
```

```
$ git checkout ...
```

2. The commit ID (hash) of the submodule has now changed, so we must update the super-project by making a new commit that will indicate the update in commit ID of the submodule.

Commands run in the **super-project**:

```
$ git status
On branch main
Changes not staged for commit:
     modified: glitter-cursor (new commits)
```

```
$ git diff
diff --git a/glitter-cursor b/glitter-cursor
--- a/glitter-cursor
+++ b/glitter-cursor
-Subproject commit 2f0f08e991d828dd27cf399c0b88edaaa48a2bf9
+Subproject commit f10d7b772342c6a9f31390af4f8a16f71c440777
```

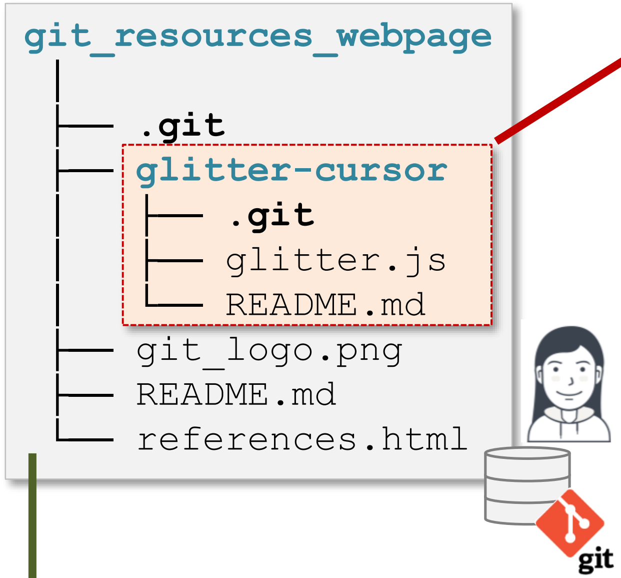
New commit to which the submodule is now pointing →

```
$ git add glitter-cursor # go back to the super-project.
$ git commit -m "Update submodule glitter-cursor‟
$ git push
```

Making a new commit in the super-project →

Making changes to a submodule

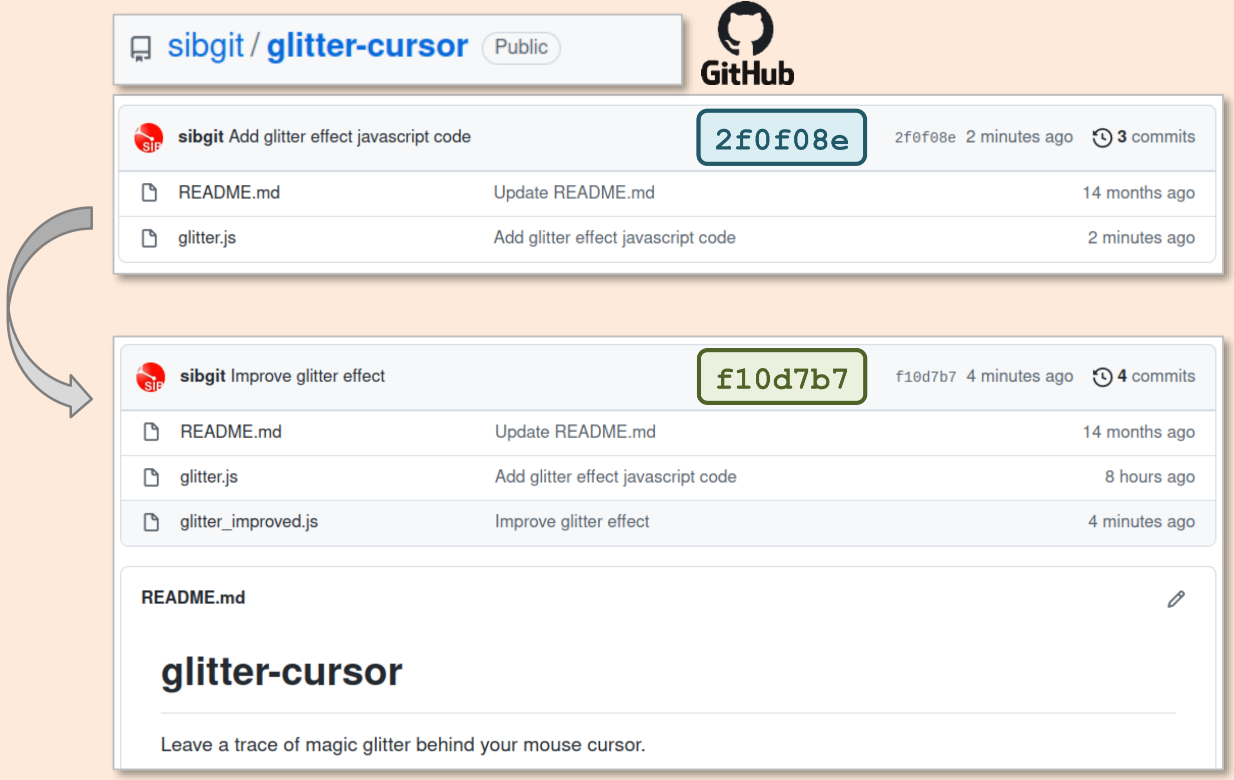
How things look on the online pages of the remotes



```
git_resources_webpage
│
├── .git
├── glitter-cursor
│   ├── .git
│   ├── glitter.js
│   └── README.md
├── git_logo.png
├── README.md
└── references.html
```

git push

Subproject
(used as submodule in the super-project)



sibgit / **glitter-cursor** Public

GitHub

sibgit Add glitter effect javascript code	2f0f08e	2f0f08e 2 minutes ago 3 commits
README.md	Update README.md	14 months ago
glitter.js	Add glitter effect javascript code	2 minutes ago

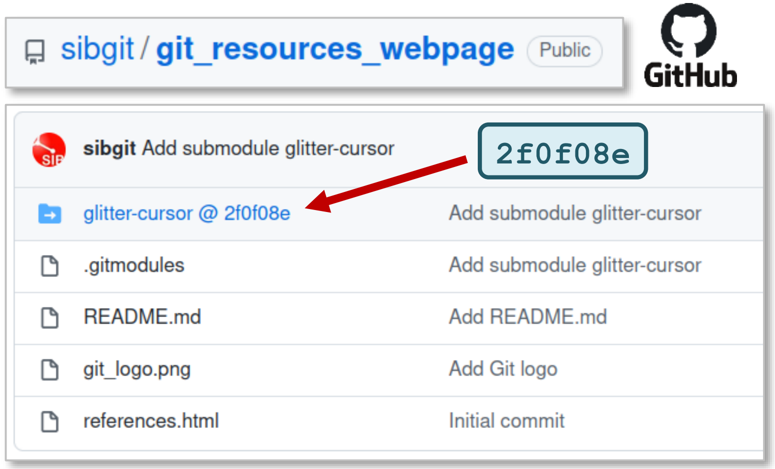
sibgit Improve glitter effect	f10d7b7	f10d7b7 4 minutes ago 4 commits
README.md	Update README.md	14 months ago
glitter.js	Add glitter effect javascript code	8 hours ago
glitter_improved.js	Improve glitter effect	4 minutes ago

README.md

glitter-cursor

Leave a trace of magic glitter behind your mouse cursor.

Main repository / super-project **(repo containing the submodule)**

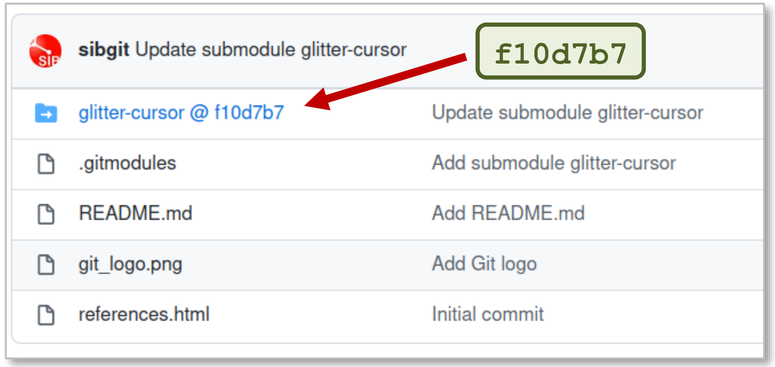


sibgit / **git_resources_webpage** Public

GitHub

sibgit Add submodule glitter-cursor	2f0f08e
glitter-cursor @ 2f0f08e	Add submodule glitter-cursor
.gitmodules	Add submodule glitter-cursor
README.md	Add README.md
git_logo.png	Add Git logo
references.html	Initial commit

git push



sibgit Update submodule glitter-cursor	f10d7b7
glitter-cursor @ f10d7b7	Update submodule glitter-cursor
.gitmodules	Add submodule glitter-cursor
README.md	Add README.md
git_logo.png	Add Git logo
references.html	Initial commit

`--recurse-submodules` option: automated submodules push

To avoid accidentally forgetting to push changes in a submodule when pushing in the super-project:

- `git push --recurse-submodules=check` : safeguard that will make your push fail is there are any "non-pushed" changes in submodules.
- `git push --recurse-submodules=on-demand` : automatically push all submodules when pushing the super-project.

- These options can also be permanently set in the Git configuration of the super-project:

```
$ git config push.recurseSubmodules check
# or
$ git config push.recurseSubmodules on-demand
```

Note: we are not using the `--global` option, so this setting only affects the current repo.

- **Important:** all these commands are run in the context (directory) of the super-project, not of the submodule!

Examples:

```
$ git push --recurse-submodules=check
The following submodule paths contain changes that
cannot be found on any remote:
    submodule-name

Please try
git push --recurse-submodules=on-demand
```

```
$ git push --recurse-submodules=on-demand
Pushing submodule 'submodule-name'
...
Pushing super-project (main project)
...
```

Supplementary material...

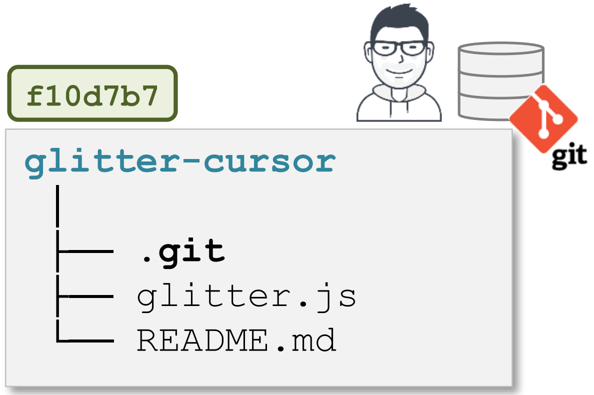
Pulling updates for a submodule

Updating a submodule to its latest commit

```
git submodule update --remote <submodule name>
```

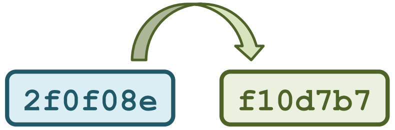
If no submodule is specified, all submodules are updated

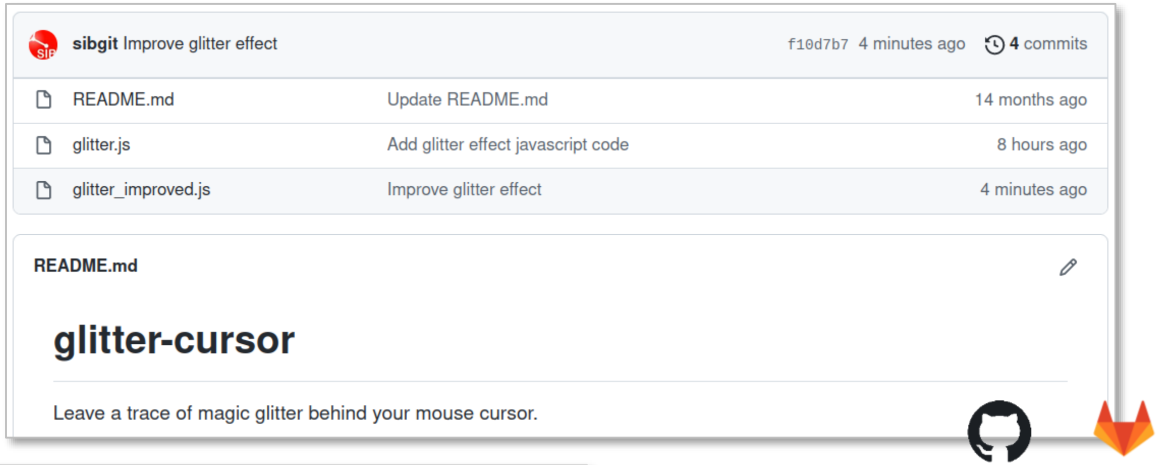
1. Bob, the maintainer of the “glitter-cursor” repo, pushes a new update.
2. Alice updates her submodule in the “git_resources_webpage” project with Bob’s new update.



```
glitter-cursor
│
├── .git
├── glitter.js
└── README.md
```

```
$ git push
```





sibgit Improve glitter effect f10d7b7 4 minutes ago 4 commits

README.md	Update README.md	14 months ago
glitter.js	Add glitter effect javascript code	8 hours ago
glitter_improved.js	Improve glitter effect	4 minutes ago

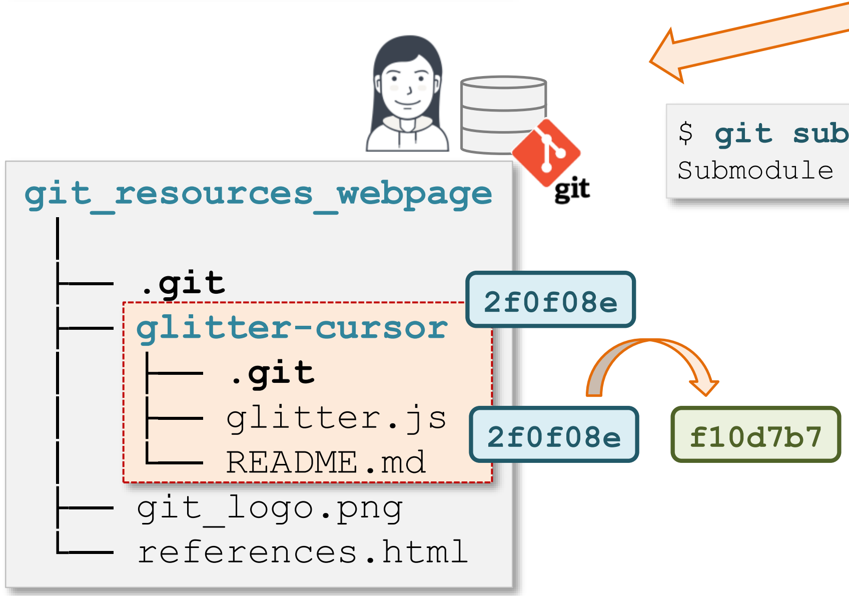
README.md

glitter-cursor

Leave a trace of magic glitter behind your mouse cursor.

GitHub GitLab

```
$ git submodule update -remote
Submodule path 'glitter-cursor': checked out f10d7b77...
```



```
git_resources_webpage
│
├── .git
├── glitter-cursor
│   ├── .git
│   ├── glitter.js
│   └── README.md
├── git_logo.png
└── references.html
```

To complete the update, Alice updates the super-project with a new commit that will make it point to the submodule commit: f10d7b7

```
$ git status
      modified: glitter-cursor (new commits)
$ git add glitter-cursor
$ git commit -m “Update submodule to latest version"
```

Pulling updates for a submodule (command details)

Updating a submodule to its latest commit

To pull the latest changes for a submodule:

```
git submodule update --remote <submodule name>
```

- If no submodule is specified, all submodules are updated.
- If the local submodule has diverged from its remote (e.g. you made some commits), `--merge`/`--rebase` must be added to the command to either merge or rebase.

```
$ git submodule update --remote --merge
```

- By default Git will try to pull the changes from the master branch. To pull from another branch, you have to specify it in `.gitmodules` by setting the parameter **branch**.

.gitmodules

```
[submodule "glitter-cursor"]
    path = glitter-cursor
    url = https://.../glitter-cursor.git
    branch = main
```

- After the content of the submodule is updated, the update in its version (commit hash) must still be committed.

```
$ git status
        modified: my-submodule (new commits)
$ git add my-submodule
$ git commit -m “Update submodule to latest version"
```

Alternatively, the pull in the submodule can also be done **manually**:

```
$ cd my-submodule
$ git switch main     # If in DETACHED HEAD state.
$ git pull
```

Supplementary material...

exercise 5

The Git reference web page gets better with submodules

git LFS

large file storage

Tracking large files can be useful…

Tracking large files together with code is an attractive proposition, e.g. in scientific applications:

- Data analysis/processing pipeline.
- Machine learning applications (training data and code in the same place).

… but Git does not work well with large files

- Git was designed for tracking code – i.e. relatively small text files.
- Adding large files to a Git repo is technically possible, however:
    - Since Git is a distributed VCS, **each local copy of a repository will contain a full copy of all versions of all tracked files**. Therefore, adding large files will quickly inflate the size of everyone's repository, resulting in higher disk space usage (on local hosts).
    - Git's internal data compression (i.e. packfiles) is **not optimized to work with binary data** (e.g. image or video files). Each change to a binary file will (more or less) add the full size of the file to the repo, taking disk space and slowing down operations such as repo cloning or update fetching.
    - Commercial **hosting platforms impose limits on the size of files** that can be pushed to hosted Git repos (GitHub: 100 MB, GitLab: no file limit but 10 GB repo limit).

The solution*: Git LFS

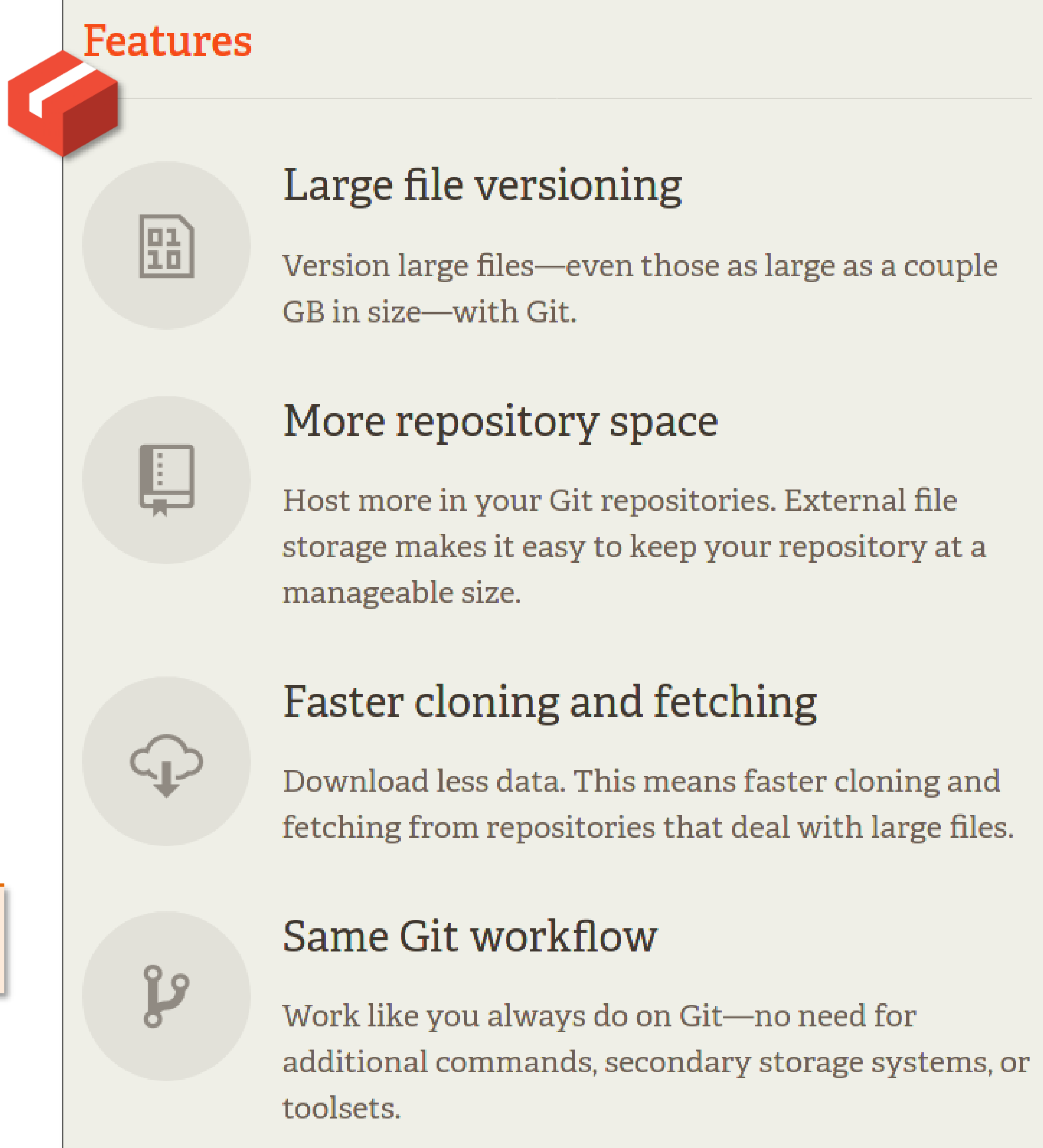
Git LFS (Large File Storage) is an extension for Git, specifically **designed to handle large files**.

Basic principle: large files are not stored in the Git database (the `.git` directory), instead:

- Only a **reference/pointer to large files** is stored in the Git database.
- The actual **files are stored in a separate repository** or "object store".

Open source project: https://git-lfs.github.com
First released in 2015.

⚠ Not all hosting services support Git LFS, and when they do, storage space is limited (additional space may be purchased).



Features

Large file versioning

Version large files—even those as large as a couple GB in size—with Git.

More repository space

Host more in your Git repositories. External file storage makes it easy to keep your repository at a manageable size.

Faster cloning and fetching

Download less data. This means faster cloning and fetching from repositories that deal with large files.

Same Git workflow

Work like you always do on Git—no need for additional commands, secondary storage systems, or toolsets.

* Alternatives to Git LFS exist, but Git LFS is the most popular.

Supplementary material...

GitHub and GitLab disk quotas, file size limit and pricing

- If your institution is running their own instance of GitLab, you can check with them if they offer LFS support (and how much space you can have their.
- Here are limits for 2 popular commercial Git hosting providers:

	GitHub.com	GitLab.com
Max file size	100 MB	No size limit
Max repo size	1 GB (recommended) 2 GB to 5 GB (max)	10 GB
LFS max file size	2 GB	No size limit (not sure)
LFS object store	1 GB storage for free 1 GB/month free bandwidth (download) 5 USD/month for each additional “pack” of 50 GB storage + 50 GB bandwidth	60 USD/year per 10GB

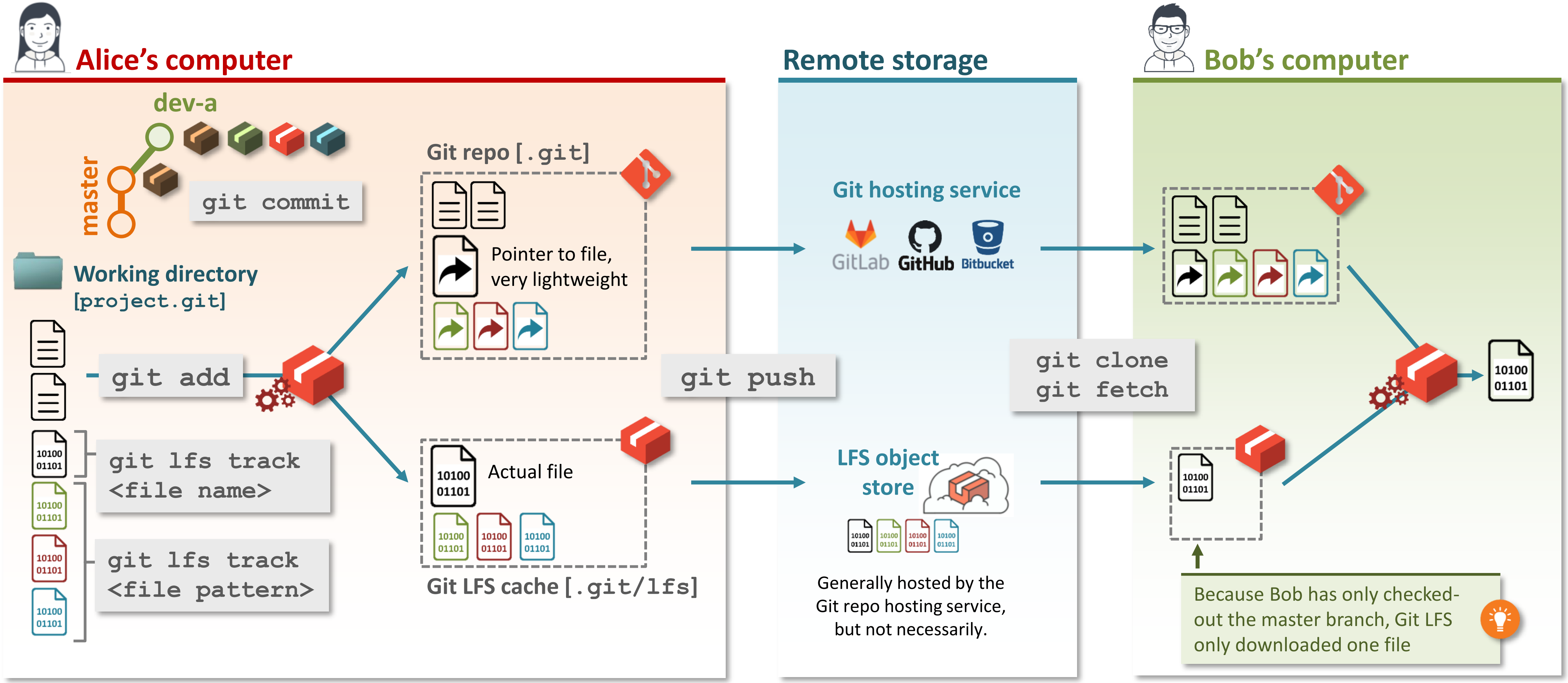
last updated on Feb 2021

You can also setup a Git LFS object store on third-party storage provider -but you need to set it up yourself and it is *not* a trivial task:

- SWITCHengines (220 CHF/TB*year) – no backup (need to organize your own).
- AWS (amazon web services).

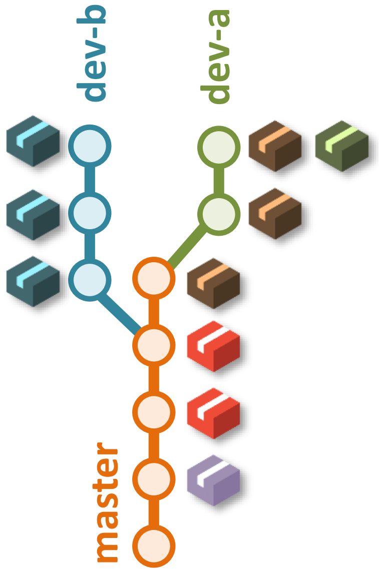
Git LFS workflow overview

- **Only a reference/pointer** to large files **is stored** in the Git database.
- The large files themselves are stored in a separate repository or "object store".
- Large files are downloaded only when needed.
- Transparent: **only 1 extra command** is needed for this workflow (`git lfs track`).



Alice's computer
dev-a
master
git commit
Git repo [.git]
Pointer to file, very lightweight
Working directory [project.git]
git add
git lfs track <file name>
git lfs track <file pattern>
10100 01101
Actual file
Git LFS cache [.git/lfs]
Remote storage
Git hosting service
GitLab
GitHub
Bitbucket
git push
LFS object store
Generally hosted by the Git repo hosting service, but not necessarily.
Bob's computer
git clone
git fetch
Because Bob has only checked-out the master branch, Git LFS only downloaded one file

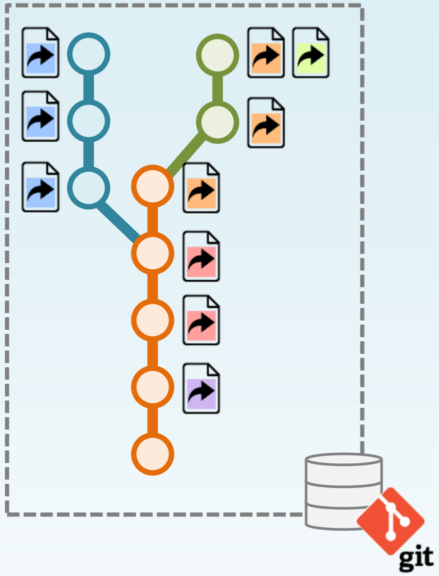
Complete Git history of project



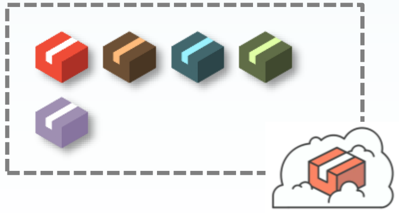
Large file. Colors represent different versions or different files.

Remote storage

Git database content



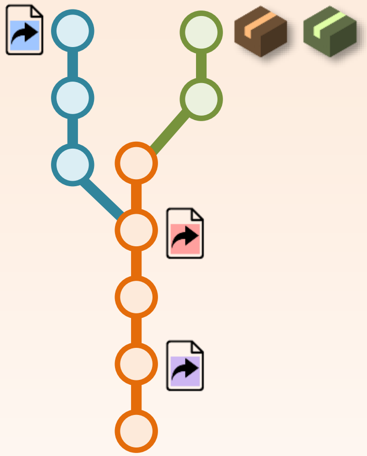
LFS object store content



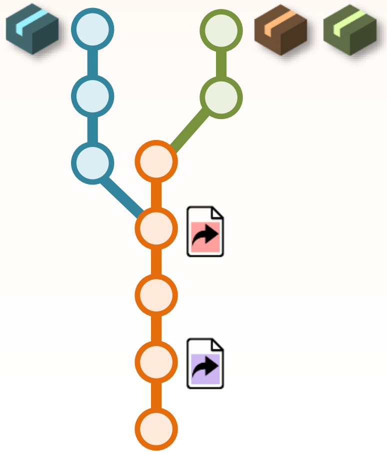
Local Git repositories

Alice's local repo

Alice just started to work on the project. She cloned the repo and created the "dev-a" branch.

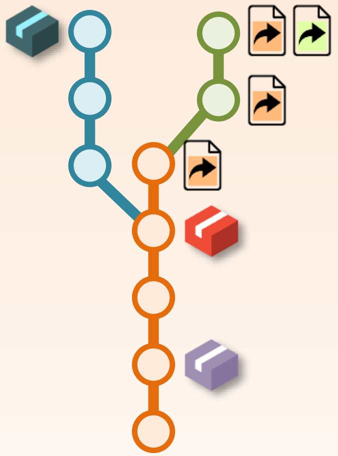


```
git checkout dev-b
```

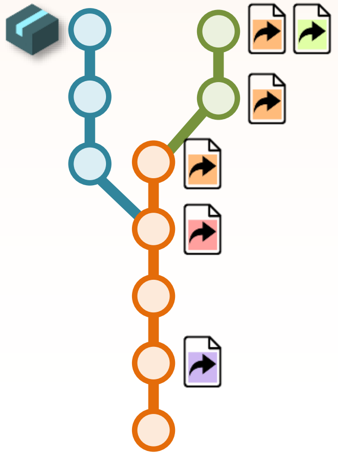


Bob's local repo

Bob contributed to the project since a while. He's currently working on "dev-b".



```
git lfs purge
```



Git LFS: initial setup

- One time setup: to be executed only once per user/machine, after Git LFS was installed.
  (this adds LFS Git filters to your global configuration file `~/.gitconfig`)

```
git lfs install
```

Git LFS: tracking files

- Adding files to Git LFS:

```
git lfs track <file name or pattern>
```

  - When using a file pattern (glob pattern), all files matching the pattern are tracked.
  - Each call to `git lfs track` creates a new entry in the `.gitattributes` file.

- Examples:

```
$ git lfs track file_1.csv
$ git lfs track file_2.csv file_3.csv
$ git lfs track “*.fasta”
$ git lfs track “*.img”
$ git lfs track “large_file_?.txt”
$ git lfs track “subdir/*.jpg”
```

Track the file named exactly “file_1.csv”

Track the files named exactly “file_2.csv” and “file_3.csv”

Track all files ending in “.fasta”

Track all files ending in “.img”

Track all files whose name are of the form “large_file_” + any single character + “.txt”

Track all files ending in “.jpg” in sub-directory “subdir”

Content of `.gitattributes`

```
file_1.csv filter=lfs diff=lfs merge=lfs -text
file_2.csv filter=lfs diff=lfs merge=lfs -text
file_3.csv filter=lfs diff=lfs merge=lfs -text
*.fasta filter=lfs diff=lfs merge=lfs -text
*.img filter=lfs diff=lfs merge=lfs -text
large_file_?.txt filter=lfs diff=lfs merge=lfs -text
subdir/*.jpg filter=lfs diff=lfs merge=lfs -text
```

It is also possible to edit directly the `.gitattributes` file instead of using the `git lfs track` command.

Do not forget “quotes” when using the `git lfs track` command with a file pattern, otherwise the pattern expands when the command is run and the matching files in your current working directory (rather than the pattern) are added to `.gitattributes`.

```
git lfs track “*.img”
```

✓

```
git lfs track *.img
```

✗

content of `.gitattributes` assuming that “file1.img” and “file2.img” are present in the working directory.

```
*.img filter=lfs diff=lfs merge=lfs -text
```

```
file_1.img filter=lfs diff=lfs merge=lfs -text
file_2.img filter=lfs diff=lfs merge=lfs -text
```

if we add a new file “file_3.img” at a later point in time…

✓ **File “file_3.img” is tracked** because it matches the *.img pattern.

✗ **File “file_3.img” is not tracked** because it matches neither file_1.img nor file_2.img.

- Recursively tracking an entire directory

```
git lfs track ″directory_path/**″
```

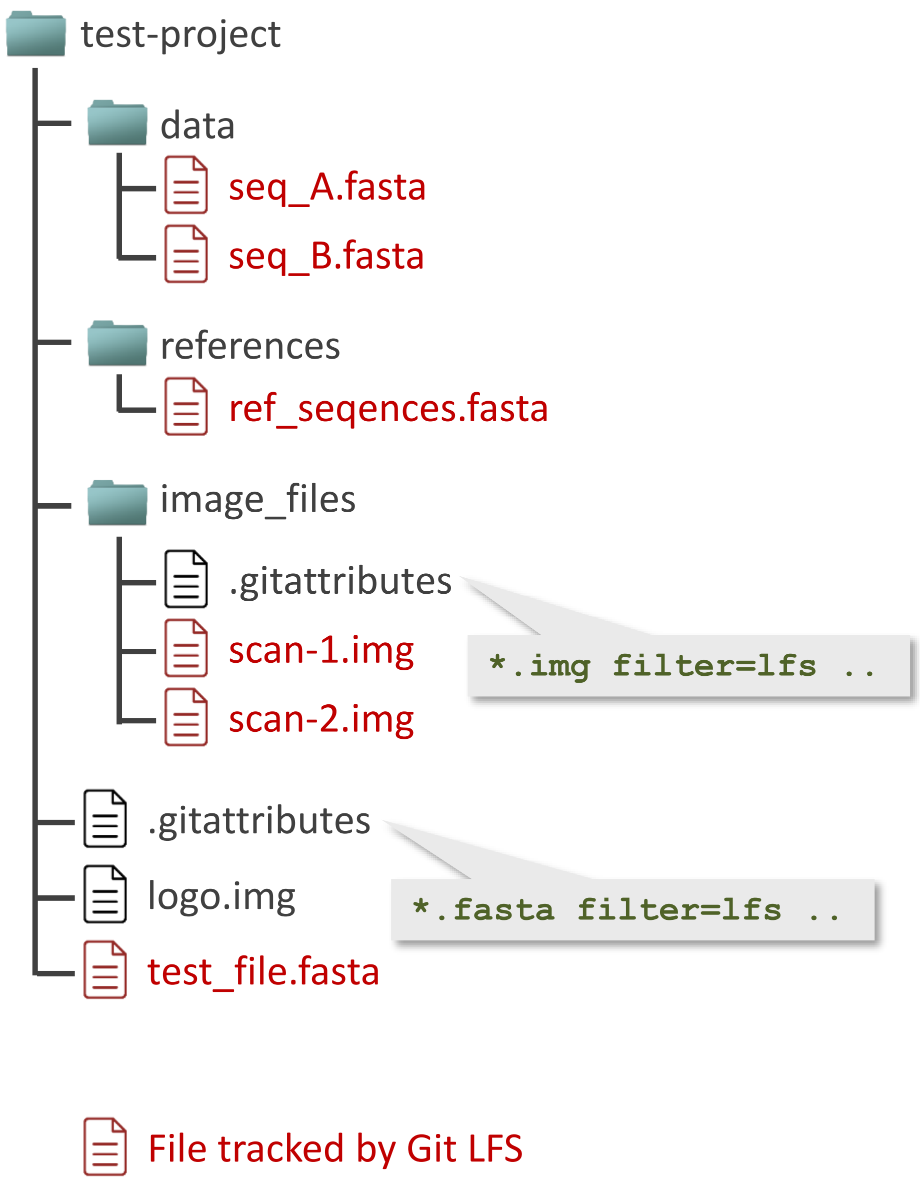
← Using /** is important.
Using / or /* will *not* work.

Content of `.gitattributes`

```
dir_to_track/** filter=lfs diff=lfs merge=lfs -text
```

Git LFS file tracking: fine-grained control

- For fine-grained control, `git lfs track <file name/pattern>` can be run in sub-directories. This places `.gitattributes` files in sub-directories (similar to how `.gitignore` files behave).
- The scope of each `.gitattributes` file is its current directory and sub-directories.
- Running `git lfs track <file name or pattern>` inside a sub-directory, creates the `.gitattributes` file inside that sub-directory



> The `.gitattributes` file(s) in your repo should be tracked - just like `.gitignore` file(s).
>
> Don't forget to commit them.

Supplementary material...

Negative pattern matching

- Unlike `.gitignore` files, `.gitattributes` files **do not** support the `!pattern` for negative matching (to tell Git LFS to not track a file).
- It is best to write `.gitattributes` files so that no negative matching is needed.
- If unavoidable, a workaround is possible by adding a line with the file/pattern that should not be tracked followed by `!filter !diff !merge` **after** the general pattern to track.

Example of `.gitattributes` file for tracking all ".jpg" files except "small_logo.jpg"

```
*.jpg filter=lfs diff=lfs merge=lfs -text
small_logo.jpg !filter !diff !merge
```

File that should not be tracked

Git LFS: untracking files

- Removing files from Git LFS:

```
git lfs untrack <file name or pattern>
```

  - Calls to `git lfs untrack` remove entries from the `.gitattributes` file.
  - The same result can be obtained by manually deleting lines from the `.gitattributes` file.

Git LFS: adding and committing files

- Nothing special to do!
- Once files are tracked by LFS, adding them to git and committing them is done as usual.

```
git add ...
```

```
git commit ...
```

```
git push ...
```

Git LFS: updating files

- Files tracked by Git LFS can be updated, staged and committed like any regular file under Git control.

```
$ git add sequence_db.fasta
$ git commit -m "updated sequence database file"
$ git push
```

The new version of the file is added to the local Git LFS cache. The pointer file is updated.

The new version of the file is pushed to the remote LFS object store.

- After commits are pushed, the remote Git LFS object store contains a copy of each version of all LFS-tracked files.

Data backup

The idea behind Git LFS is to avoid replicating large data files across local copies of a Git repository. This has implications for data-backup:

- For LFS-tracked files, local repos *cannot* be relied-upon to contain a full copy of all data.
- Therefore the remote repository has to be backed-up.

In addition, keep in mind that, depending on the data you are working with, there might be legal aspects to consider (e.g. data might have to be stored encrypted, or be stored within the country)

Using Git LFS: diff-ing files

- For LFS-tracked files, `git diff` will only show the difference between pointer files, not between actual file content (even for text files).

```
git diff HEAD~1 sequences_A.fasta
diff --git a/sequences_A.fasta b/sequences_A.fasta
index a33c8a7..01f8d67 100644
--- a/sequences_A.fasta
+++ b/sequences_A.fasta
@@ -1,3 +1,3 @@
 version https://git-lfs.github.com/spec/v1
-oid sha256:c1d5ab0faf552cdb3a365347093abc42a4e65718348e17eaad1584d650ae7aa6
-size 6010948
+oid sha256:fc51c1860c4341e175dcfc24fc2c653f75c5e8b3bae6cf80d3632788ccaf4379
+size 6011029
```

Listing files tracked by Git LFS

- List LFS-tracked files of HEAD commit (i.e. currently checked-out files).

```
git lfs ls-files
```

Example:

```
git lfs ls-files
b04f62c7a1 * large_file_1.txt
efdc76ef2a * sequences_B23.fasta
e6aa57987e * subdir/logo_image.img
```

- List files associated with any reference (commit).

```
git lfs ls-files <ref>
```

Example:

```
git lfs ls-files HEAD~1
b04f62c7a1 * large_file_1.txt
fc51c1860c - sequences_A12.fasta
efdc76ef2a * sequences_B23.fasta
e6aa57987e * subdir/logo_image.img
```

```
git lfs ls-files origin/dev
b04f62c7a1 * large_file_1.txt
e82048e6d3 - sequence_C34.fasta
e6aa57987e * subdir/logo_image.img
```

* = file is present in worktree

- = file is absent in worktree

- List all LFS-tracked files in the entire repo history.

```
git lfs ls-files --all
```

Example:

```
git lfs ls-files --all
b04f62c7a1 * large_file_1.txt
efdc76ef2a * sequences_B23.fasta
e6aa57987e * subdir/logo_image.img
e82048e6d3 - sequence_C34.fasta
fc51c1860c - sequences_A12.fasta
c1d5ab0faf - sequences_A12.fasta
```

Clearing the local Git LFS cache

- Deleting files from the Git LFS local cache [`.git/lfs/objects`] can be done using:

```
git lfs prune
```

Files that are deleted by the `prune` command are those that:

- Are not currently checked-out.
- Are not part of the latest commit of a “recent” branch or tag (“recent” defaults to 10 days and can be customized via `lfs.fetchrecentcommitsdays` and `lfs.pruneoffsetdays`).
- Are not part of a commit that was never pushed to the remote (since in this case there is not yet a copy of the file in the remote object store, and hence deleting it would amount to permanently losing the file).

- `lfs prune` command options:

```
git lfs prune --dry-run
```

- Lists the number of files that would be deleted, without actually deleting them.

```
$ git lfs prune --dry-run
prune: 6 local object(s), 4 retained, done.
prune: 2 file(s) would be pruned (12 MB), done.
```

```
git lfs prune --verify-remote
```

- Verify that files are present on the remote before deleting them.

Pulling LFS content from a remote

- Nothing special to do!
- Just use the regular Git commands and Git LFS will download content as needed.

```
git clone ...
```

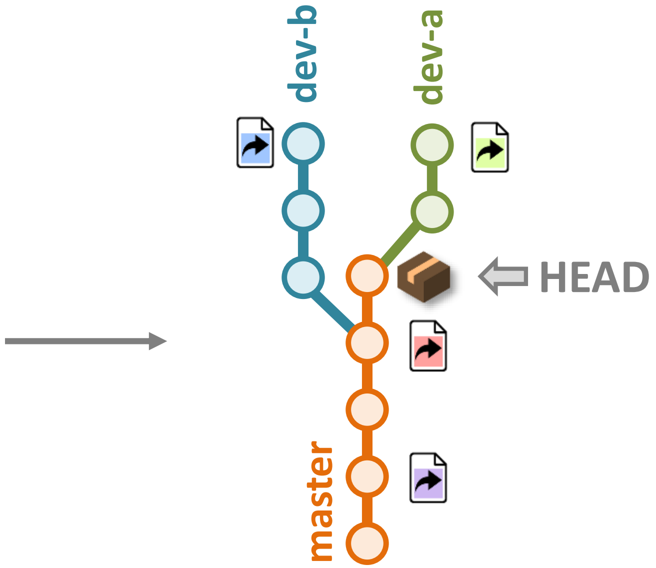
```
git fetch ...
```

```
git pull ...
```

```
git switch ...
```

- By default, only the LFS-tracked files needed for the currently checked-out branch are downloaded.

Example: if we `git clone` a new repository, only the LFS-tracked files needed for the latest commit of the “master” branch are downloaded.



Pulling additional LFS content from a remote (files from older commits or files from other branches)

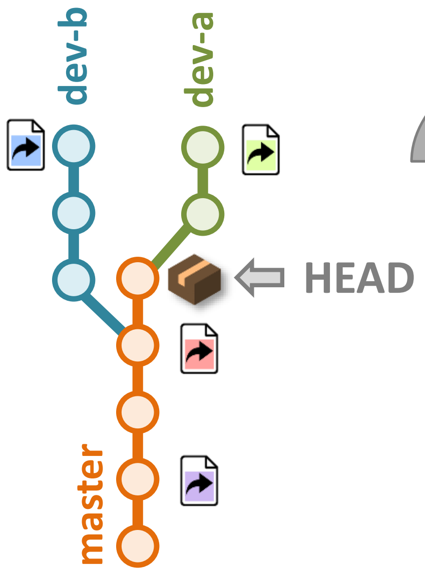
It can be useful to download LFS-tracked files to the local LFS cache, e.g. when anticipating off-line time.

```
git lfs fetch --recent
```

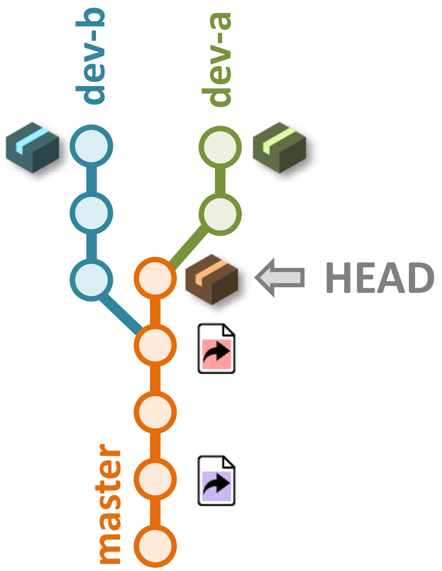
- Downloads the LFS-tracked files of *the last* commit of all branches or tags that are considered "recent".
  - By default, "recent" is defined as no more than 7 days old.
  - The definition of "recent" can be customized via the `git config lfs.fetchrecentcommitsdays <days>` configuration option (where `<days>` = number of days).

```
git lfs fetch --all
```

- Downloads all LFS-tracked files for all commits.

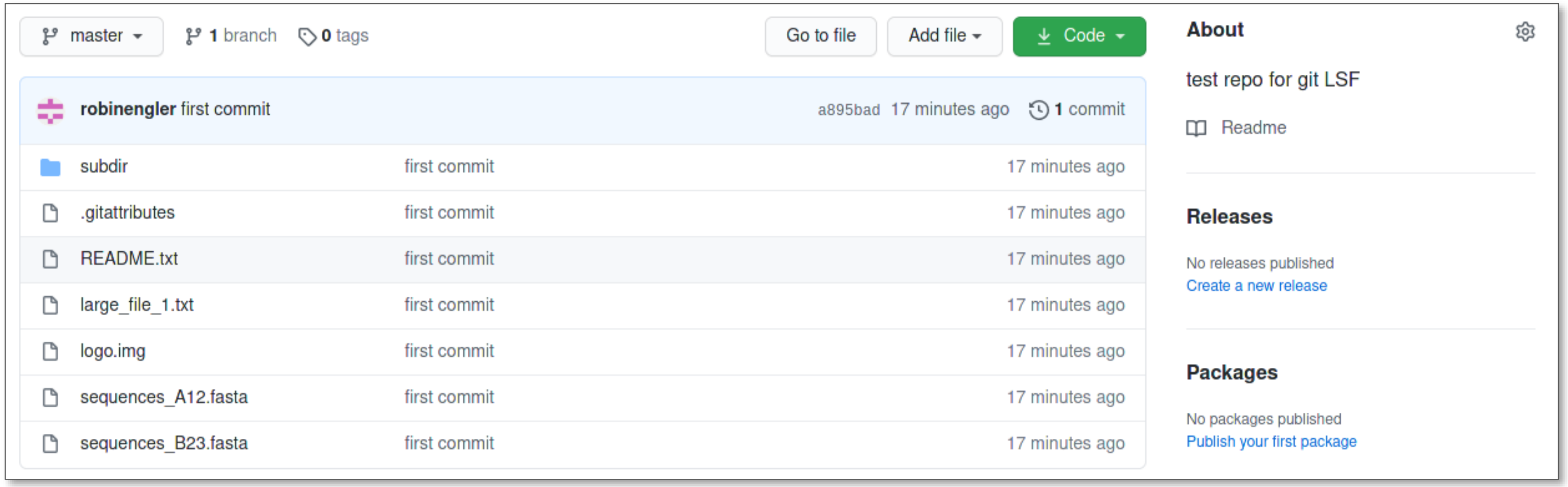


```
$ git lfs fetch --recent
fetch: Fetching reference refs/heads/master
fetch: Fetching recent branches within 7 days
fetch: Fetching reference origin/dev-a
fetch: Fetching reference origin/dev-b
```



Supplementary material...

- On Git hosting platforms like GitHub or GitLab, LFS-tracked files are listed just like regular files:



robinengler first commit a895bad 17 minutes ago 1 commit

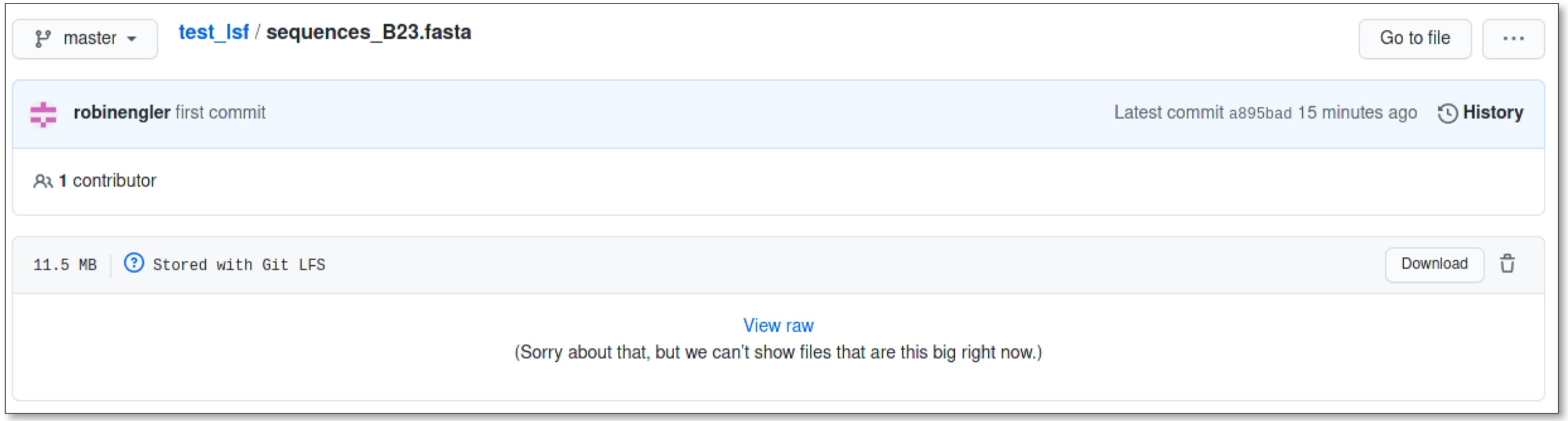
subdir	first commit	17 minutes ago
.gitattributes	first commit	17 minutes ago
README.txt	first commit	17 minutes ago
large_file_1.txt	first commit	17 minutes ago
logo.img	first commit	17 minutes ago
sequences_A12.fasta	first commit	17 minutes ago
sequences_B23.fasta	first commit	17 minutes ago

test repo for git LSF

No releases published
Create a new release

No packages published
Publish your first package

- When selecting an LFS-tracked file, the content is not shown and instead a “Stored with Git LFS” mention is listed:



test_lsf / sequences_B23.fasta

robinengler first commit Latest commit a895bad 15 minutes ago

1 contributor

11.5 MB Stored with Git LFS

(Sorry about that, but we can't show files that are this big right now.)

exercise 6 A

Tracking files already in Git

When a set of files are already part of a Git repository's history, there are two options to start tracking them with Git LFS:

1. Add the files (or file patterns) as tracked files with `git lfs track`. In this case however, the versions of the files associated with already made commits will remain in the Git database.

2. Remove the files' entire history from the Git repo, and have them tracked by Git LFS instead (over all of their history). This can be done using `git lfs migrate` command.

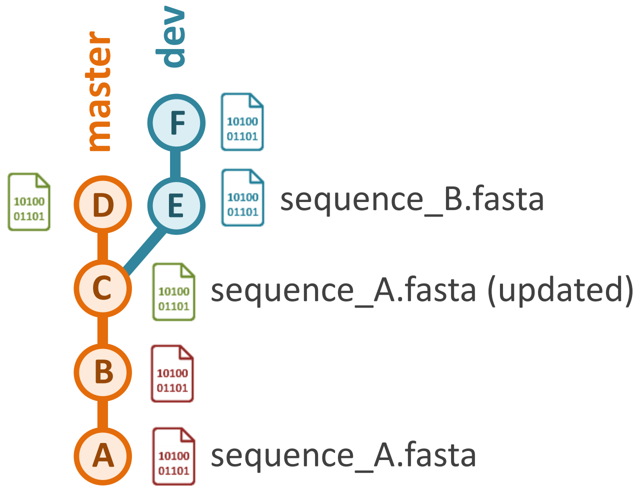
Supplementary material...

Option 1

Keep files to track history in the Git repo up to the current commit.

```
git lfs track “*.fasta”
git add *.gitattributes
git add *.fasta
git commit
```

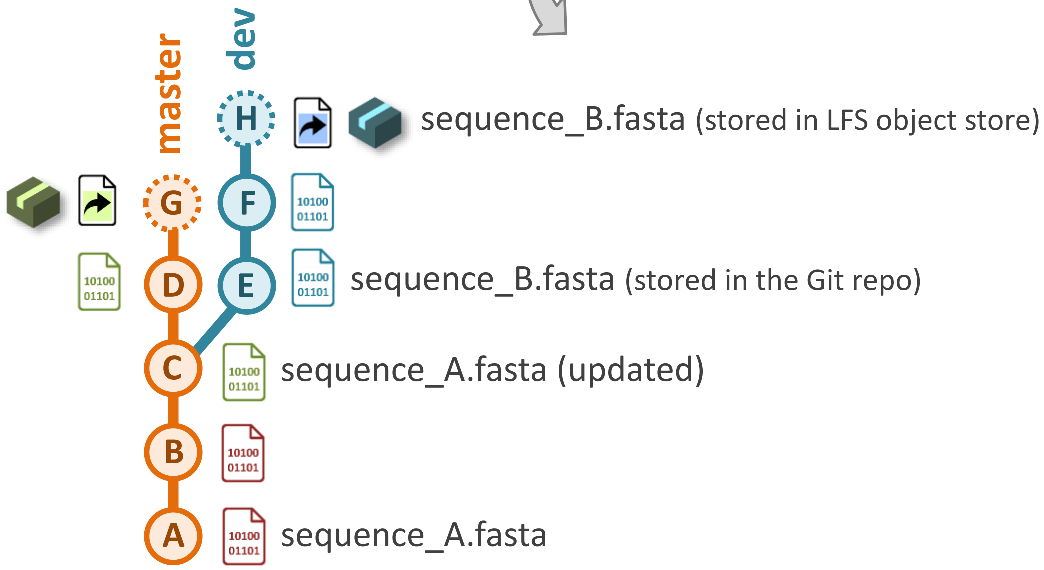
... now do the same for branch dev



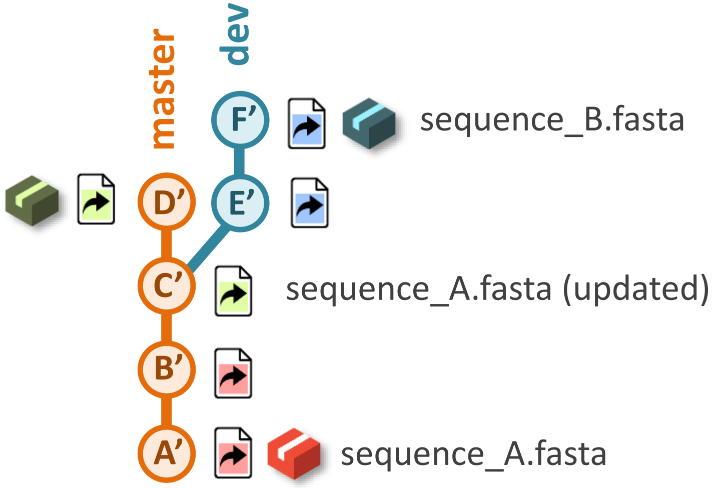
Option 2

Remove files from entire Git repo history and rewrite history with files stored in LFS.

```
git lfs migrate import \
    --include="*.fasta” \
    --everything
git lfs checkout
```



master
dev
H
sequence_B.fasta (stored in LFS object store)
G
F
D
E
sequence_B.fasta (stored in the Git repo)
C
sequence_A.fasta (updated)
B
A
sequence_A.fasta
10100 01101



+ The repo’s history remains the same.
- Git repo size possibly still too large to push to GitHub/GitLab
- Mix of files being stored in Git repo and LFS object store = not a clean solution.

+ Large files have now their entire history saved in Git LFS.
+ Size of Git database [.git/objects] truly reduced.
- History completely changed: everyone has to reset their copy of the Git repo.

The `git lfs migrate` command

```
git lfs migrate import --include=<file name or pattern> --everything
```

- List of files or file patterns to “import” into Git LFS.
- Entries in `.gitattributes` will be automatically created.
- Multiple patterns/files can be specified by separating them with a comma, e.g.: `--include="*.fasta,*.img"`

This options tells git LFS to process all (local) branches of the repository.

Example:

```
git lfs migrate import --include="*.fasta,*.img" --everything

git lfs ls-files
702c4c3a56 - logo.img
6f0a4add2f - sequences_A.fasta
```

After the migrate import command completes, LFS-tracked files in the working directory are replaced with their pointer (indicated by the “ – “).

```
git lfs checkout
git lfs ls-files
702c4c3a56 * logo.img
6f0a4add2f * sequences_A.fasta
```

The content of the files can be restored with `git lfs checkout`.

Supplementary material...

The `git lfs migrate` command

A couple of warnings…

History overwrite warning !

⚠ The `git lfs migrate import` command **rewrites the entire history** of your repository!

- Updating a remote repo with the changes requires a `git push --force`.
- Coordinate this operation with other people working on the repo.

Data loss warning !

⚠

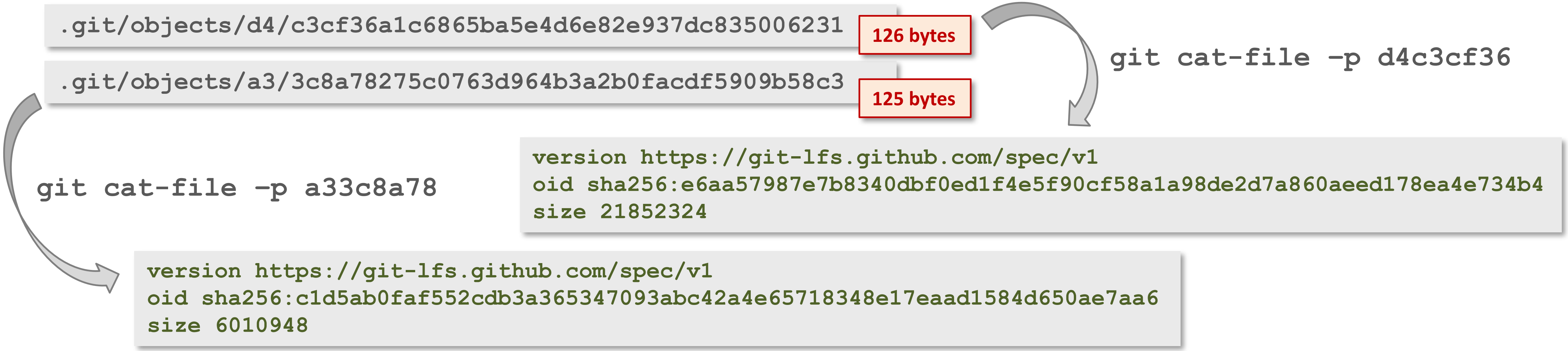
- Never run `git lfs migrate import` with a non-clean working directory. All your uncommitted changes will be lost (true story)!
- To be on the safe side, it's best to **make a full copy/backup of your Git repository before running the migrate command**. In this way, should anything go wrong, you can restore your repository from your copy.

Supplementary material…

Behind the scenes...

- Git LFS stores the tracked files in the LFS cache [`.git/lfs/objects`] rather than in the Git repo [`.git/objects`].
- A lightweight “pointer” file is saved in the git repository.

Example of “pointer” blob objects stored in the Git repo [`.git/objects`]



```
.git/objects/d4/c3cf36a1c6865ba5e4d6e82e937dc835006231
```

126 bytes

```
.git/objects/a3/3c8a78275c0763d964b3a2b0facdf5909b58c3
```

125 bytes

```
git cat-file -p d4c3cf36
```

```
version https://git-lfs.github.com/spec/v1
oid sha256:e6aa57987e7b8340dbf0ed1f4e5f90cf58a1a98de2d7a860aeed178ea4e734b4
size 21852324
```

```
git cat-file -p a33c8a78
```

```
version https://git-lfs.github.com/spec/v1
oid sha256:c1d5ab0faf552cdb3a365347093abc42a4e65718348e17eaad1584d650ae7aa6
size 6010948
```

The actual files are stored in the Git LFS cache [`.git/lfs/objects`]

```
.git/lfs/objects/e6/aa/e6aa57987e7b8340dbf0ed1f4e5f90cf58a1a98de2d7a860aeed178ea4e734b4
```

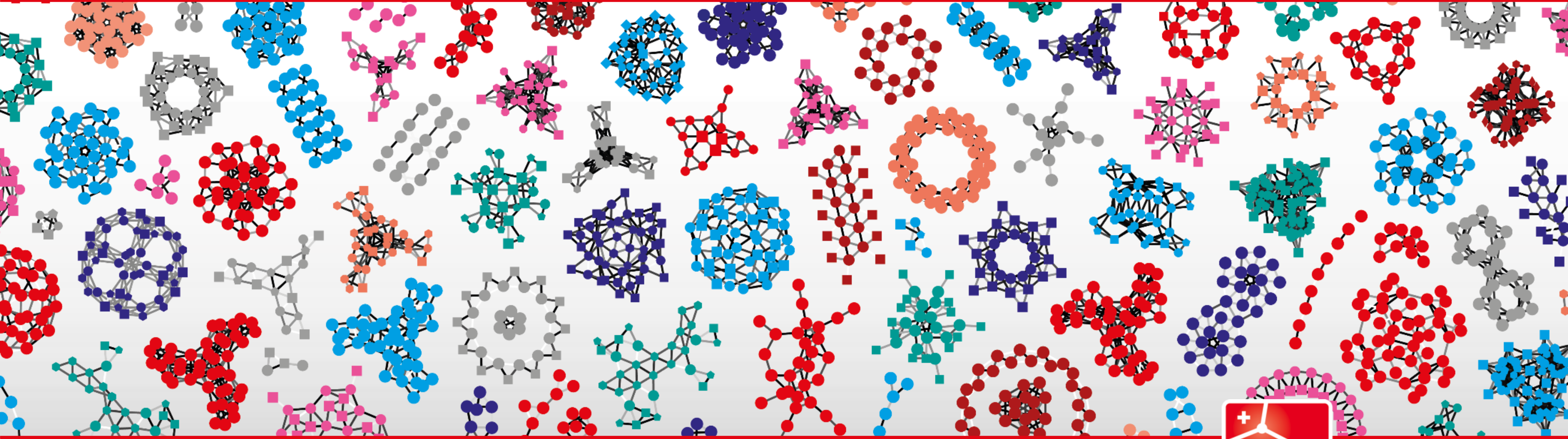
21.8 MB

```
.git/lfs/objects/c1/d5/c1d5ab0faf552cdb3a365347093abc42a4e65718348e17eaad1584d650ae7aa6
```

6 MB

Supplementary material...

exercise 6 B



Thank you for attending this course

SIB

Swiss Institute of Bioinformatics