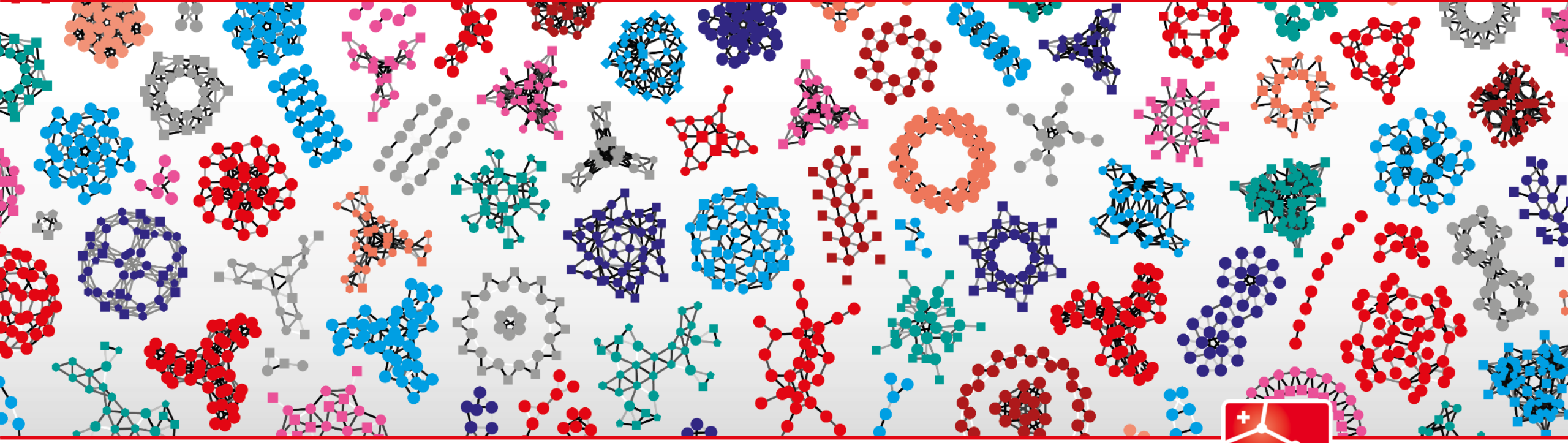


Version control with Git – advanced topics

Swiss Institute of Bioinformatics

www.sib.swiss

Robin Engler
Vassilios Ioannidis

Lausanne, 22-24 Feb 2023

Git advanced topics: 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, command summary (cheat sheet), setting-up your environment, link to feedback form, links to references.

https://gitlab.sib.swiss/rengler/git_course_public

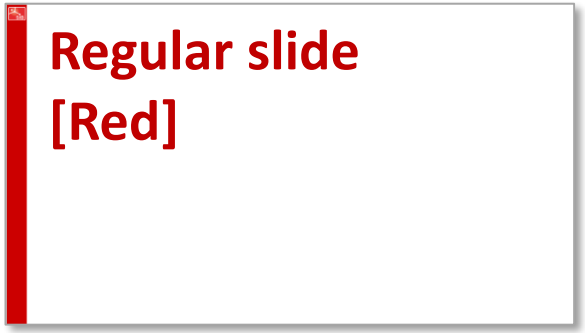
Google doc: Register for collaborative exercises (and optionally for exam), FAQ, ask questions.

https://docs.google.com/document/d/1EX72NInz-eA2d2GOa5aTB8D88GWb91Sk-sCNHwQYXqE

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

Course slides

- 3 categories of slides:

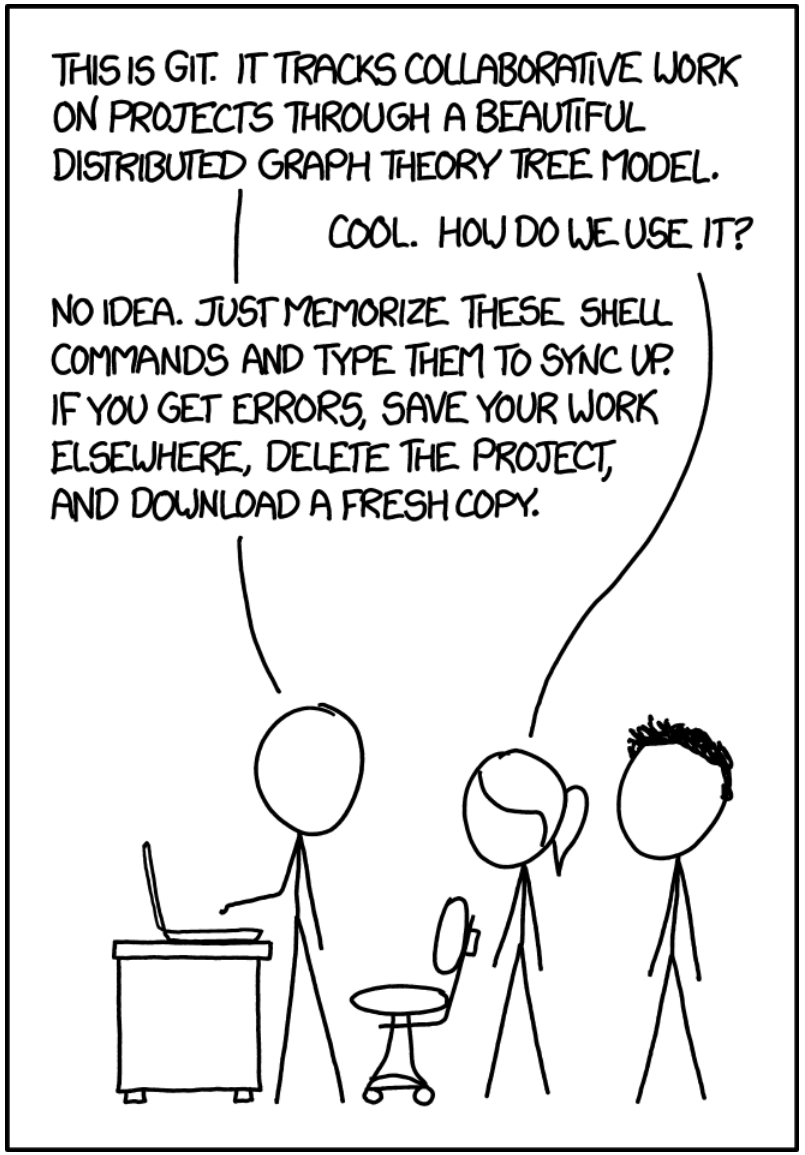


Slide covered in detail during the course.

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.

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

Learning objective



source: https://xkcd.com/1597

Command line vs. graphical interface (GUI)

- This course focuses exclusively on **Git concepts** and **command line** usage.
- Many GUI (graphical user interface) software are available for Git, often integrated with code or text editors (e.g. Rstudio, Visual Studio Code, PyCharm, …), and it will be easy for you to start using them (if you wish to) once you know the command line usage and the concepts of Git.

review / refresher

Git commands we assume you know

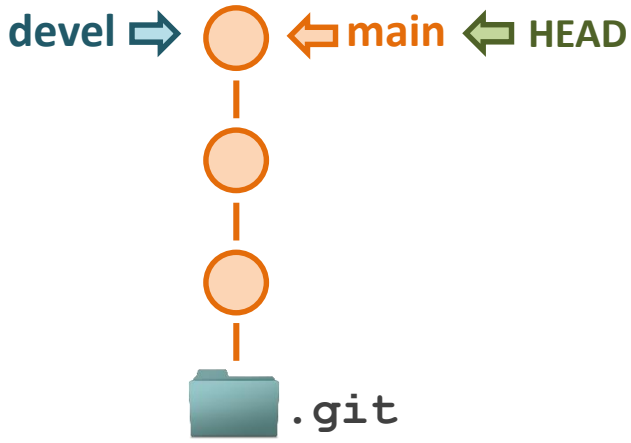
Reminder...

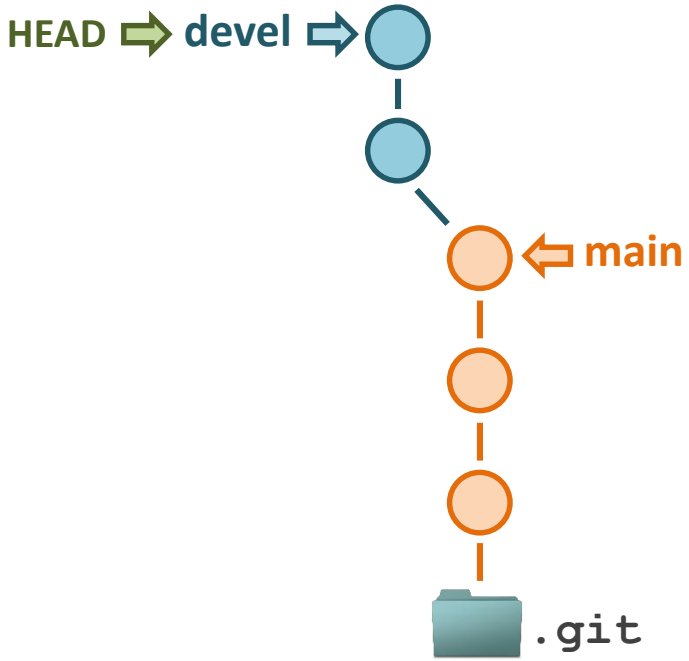
Reminder...

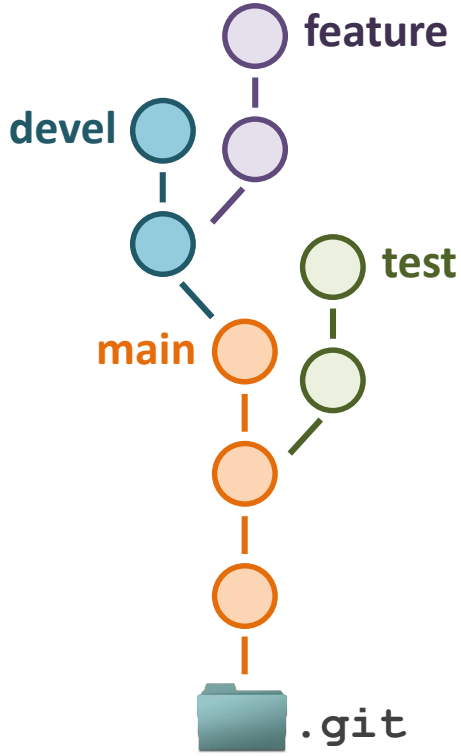
```
git init / git clone

git add <file>

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

git restore

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>
```

unstage a file from the index

Reset file in the working tree to its state present in the index.







git log

git show

git status

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

    Add stripe color option to class Cheshire_cat.
```

```
$ 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
```

```
$ git status
On branch main
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
```

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
```


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

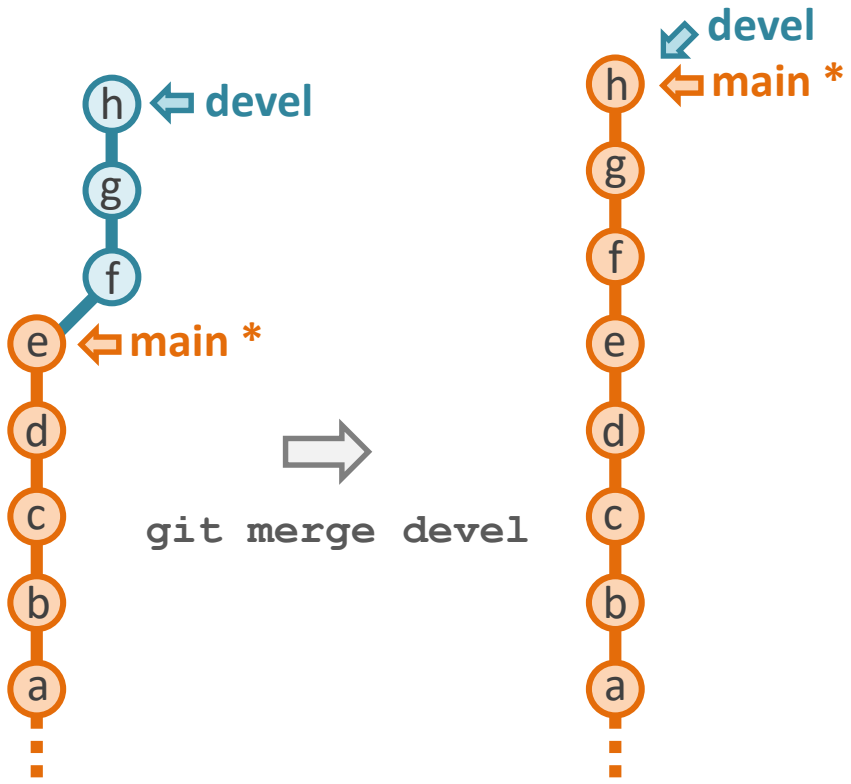
Reminder...

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 **main**), the merge is said to be **fast-forward**.

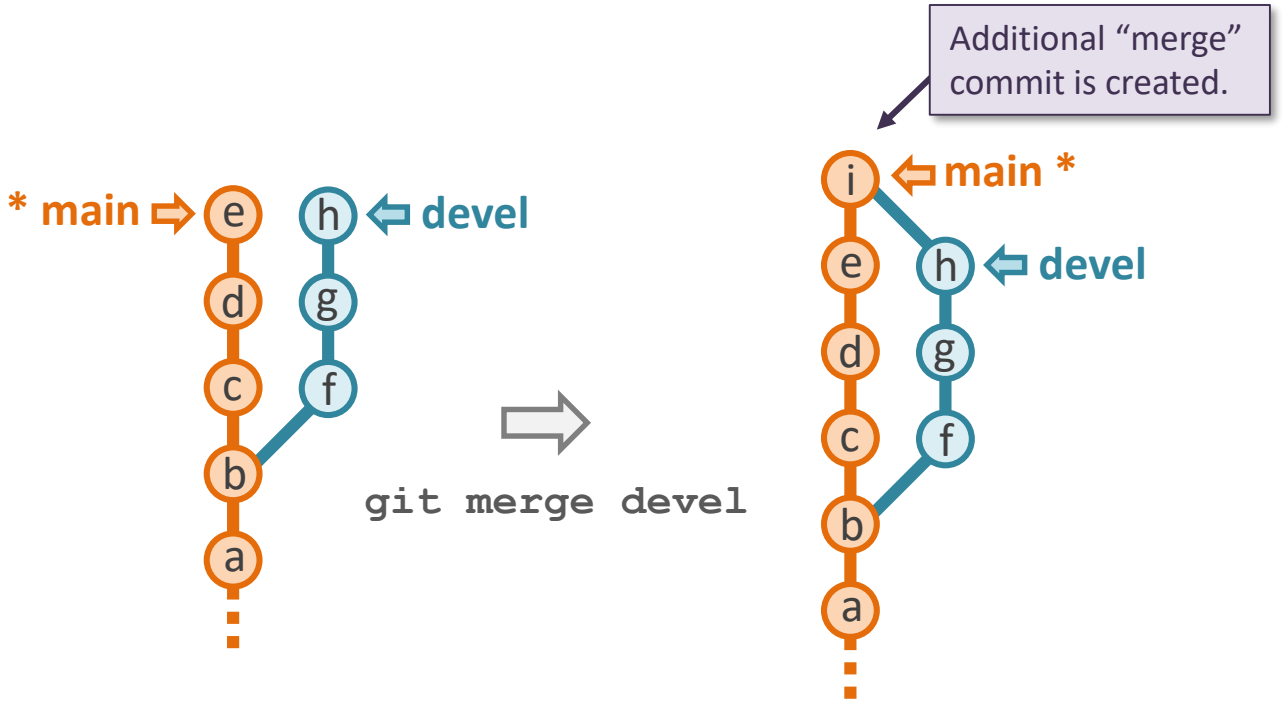
Fast-forward merge

- Guaranteed to be conflict free.



Non fast-forward merge

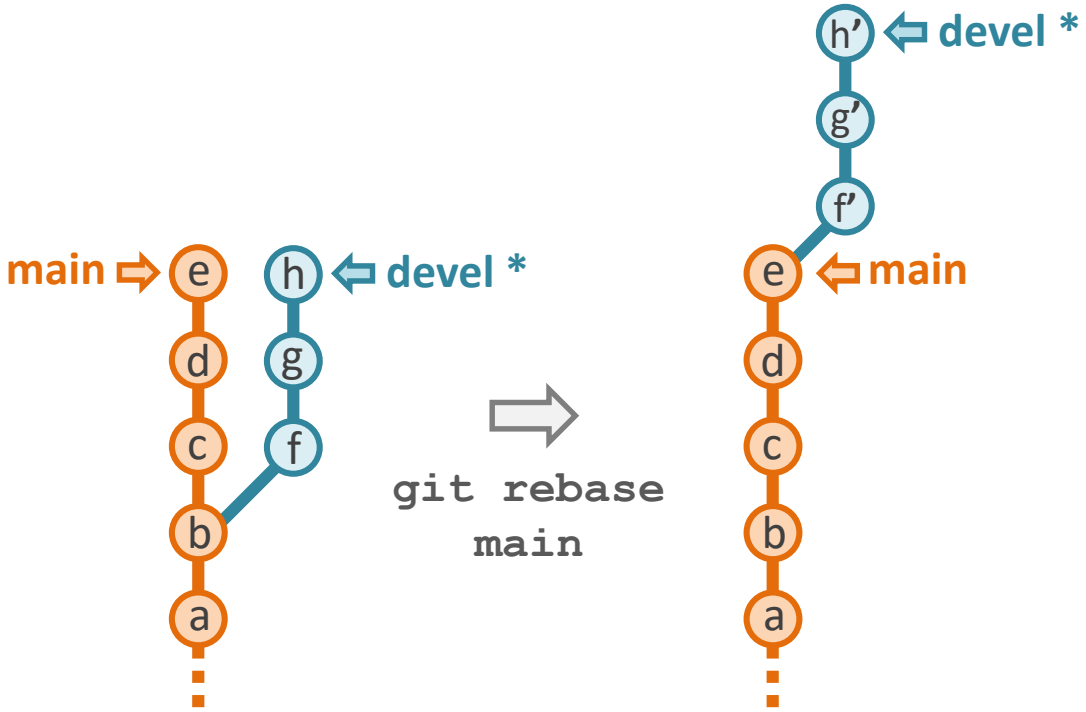
- Creates an additional “merge commit”.
- Conflicts may occur.



Reminder...

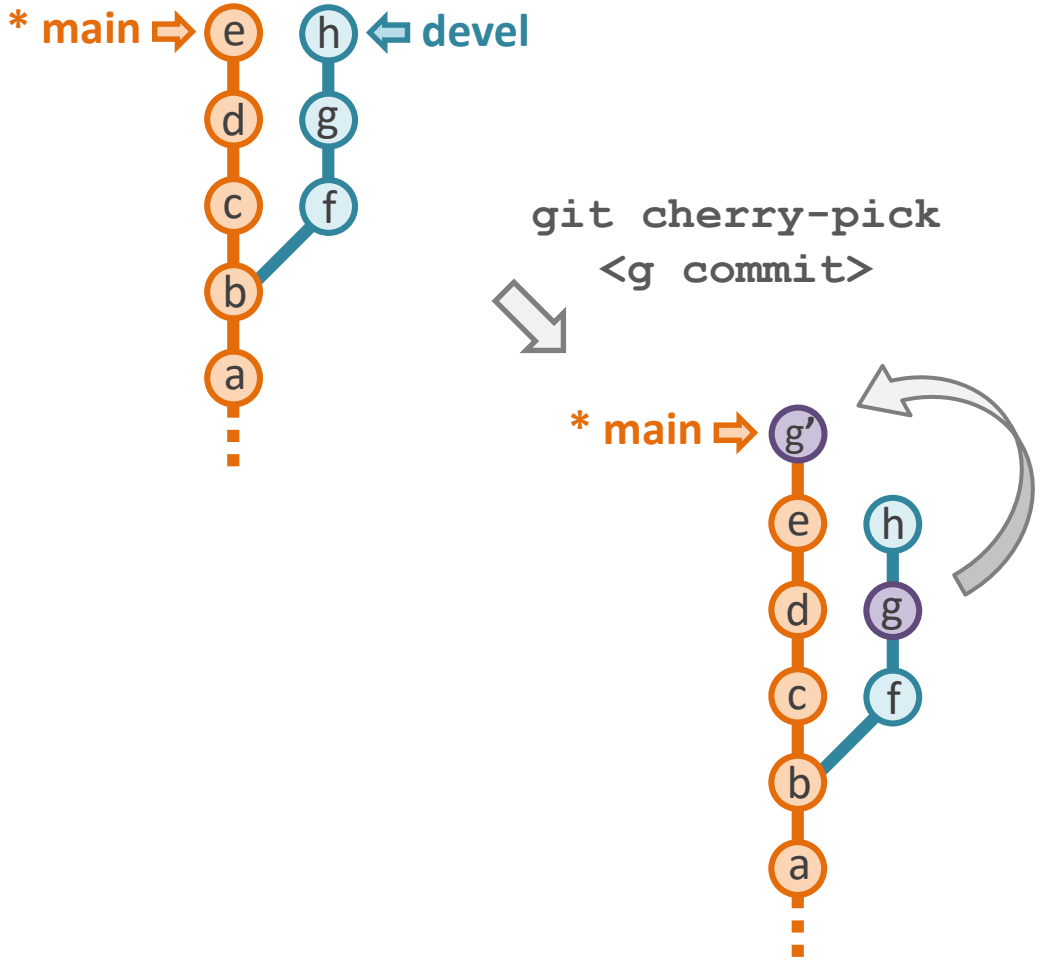
Branch rebasing

- 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).



Cherry-picking

- “copy” changes introduced by a commit on another commit.



Reminder...

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.

Reminder...

Interacting with remotes: summary

Command	What it does	Important comments
`git clone <URL>`	**Create a local copy** of an online repo.	
`git push`	**push new commits** on the current branch to the remote.	**Run on the branch that you wish to push.** (only changes on the active branch are pushed)
`git push -u origin <branch-name>`	Same as git push, but additionally sets the upstream branch to `origin/branch-name`. Only needed if branch has no upstream set.	`-u` option is only needed when pushing a branch to the remote for the very first time. It is not needed if you initially created the local branch from a remote branch.
`git fetch`	**Download all updates** from the remote to your local repo. Does not update your local branch pointer to `origin/branch-name` .	**Can be run from any branch.** Downloads all changes from the remote (even on branches for which you do not have a local version) to your local repo.
`git pull`	**Download all updates** and **merge changes** the upstream `origin/branch-name` into the active branch (i.e. update the active branch to its version on the remote).	**Run on the branch that you wish to update.** `git pull` is a shortcut for `git fetch + git merge origin/branch-name`
`git pull --no-rebase`	Fetch + 3-way merge active branch with its upstream `origin/branch-name`.	
`git pull --rebase`	Fetch + rebase active branch on its upstream `origin/branch-name`.	
`git pull --ff-only`	Fetch + fast-forward merge active branch with its upstream `origin/branch-name`. If a fast-forward merge is not possible, an error is generated.	

On recent versions of Git, the default pull behavior is to abort the pull if a branch and its upstream are diverging. On older versions, the default behavior is `git pull --no-rebase`.

Reminder...

rewriting history

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

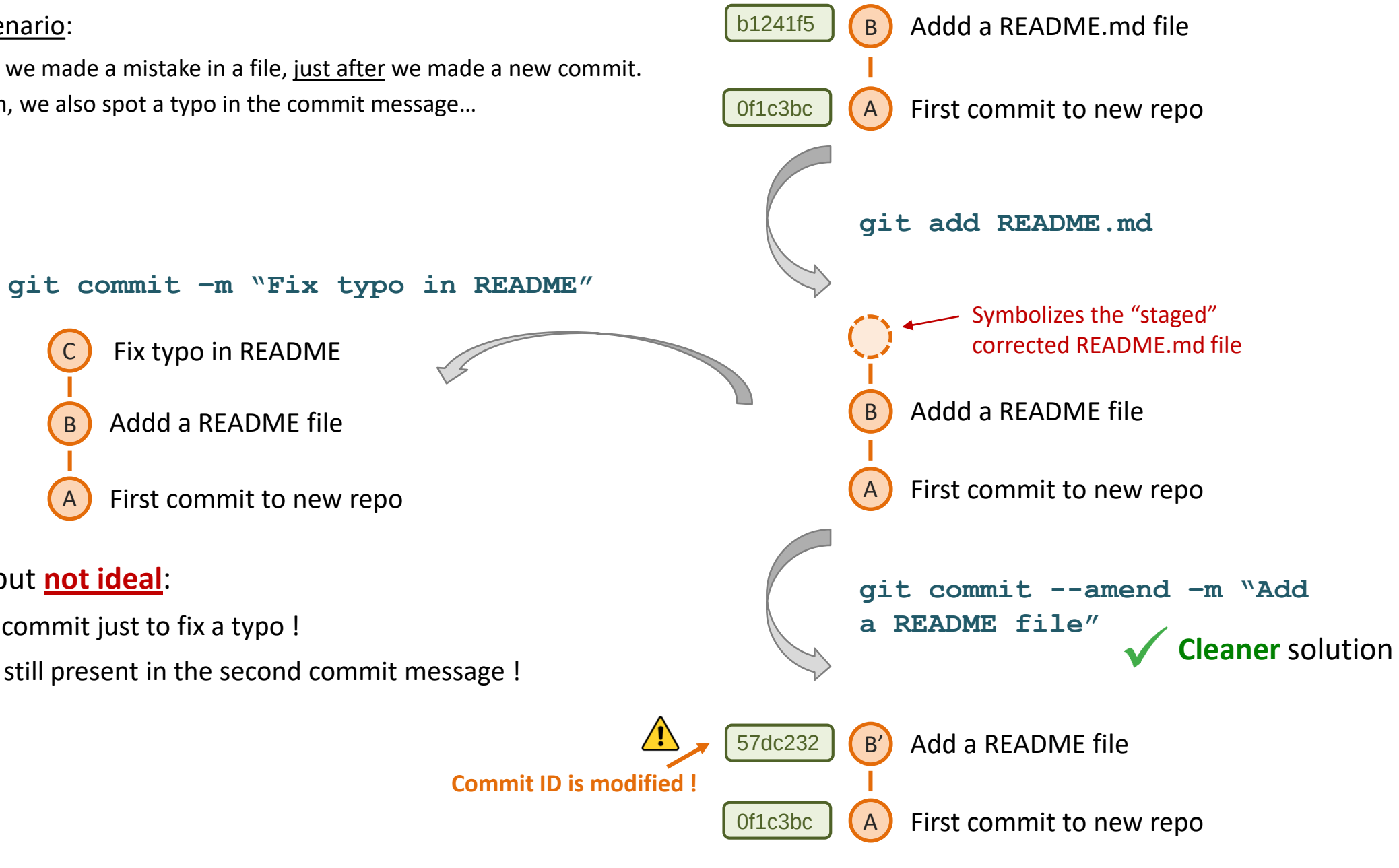
git commit --amend

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

Amending the latest commit of a branch

Use case scenario:

- we realize we made a mistake in a file, just after we made a new commit.
- In addition, we also spot a typo in the commit message…



Possible but **not ideal**:

- ✗ New commit just to fix a typo !
- ✗ Typo still present in the second commit message !

Re-writing the latest commit (amending)

To amend the latest commit of a branch:

1. Stage the changes you want to make to your commit, or, if you just want to modify the commit message, don't stage anything.

2. Run one of the `git commit --amend` commands as shown below:

   - This will open an editor where you can modify the commit message interactively.

     ```
     git commit --amend
     ```

   - This is to enter the new commit message directly in the command.

     ```
     git commit --amend -m "new message"
     ```

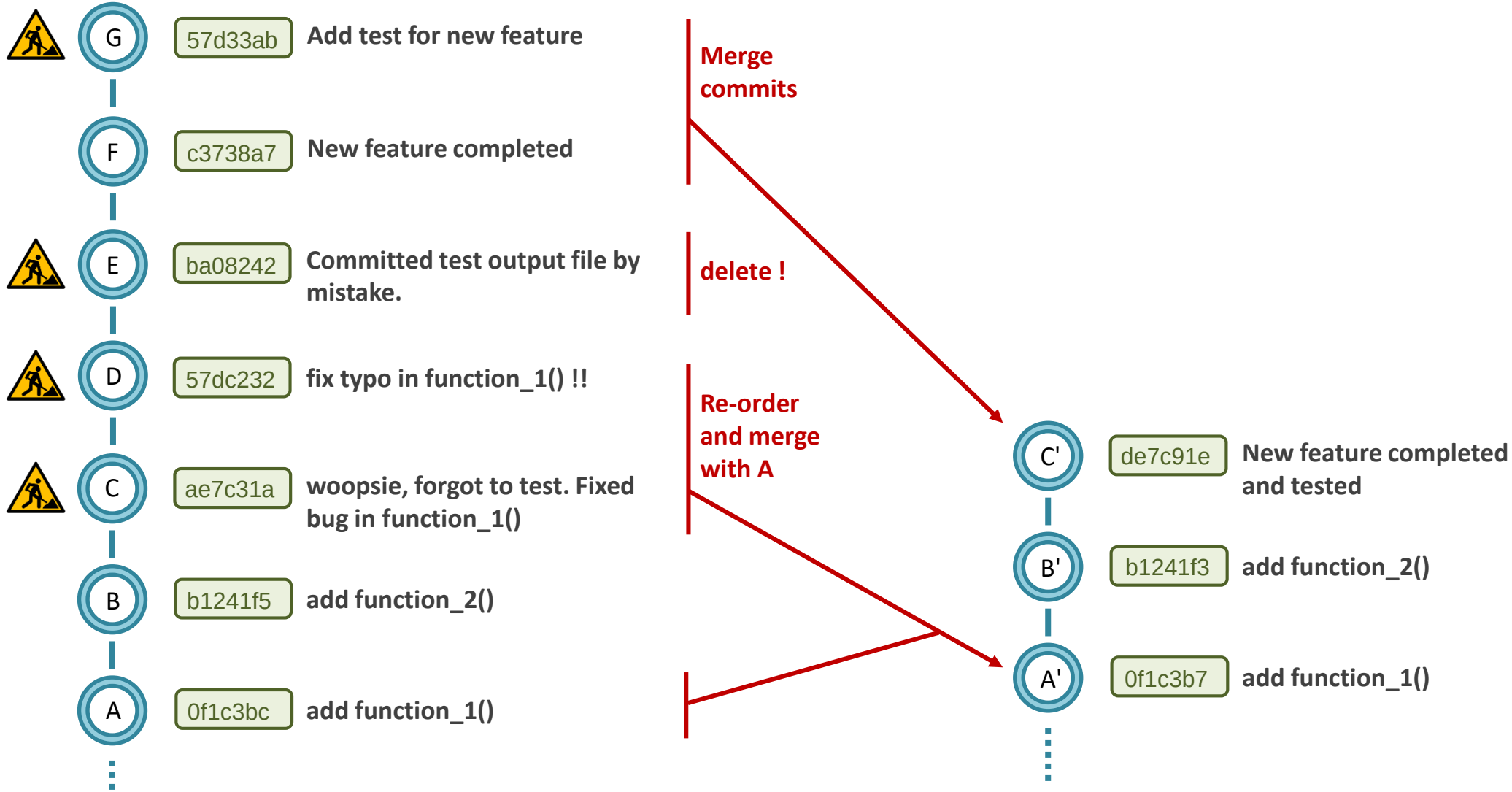
   - This is to keep the commit message unchanged (only edit the content of the commit).

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

demo: commit amending

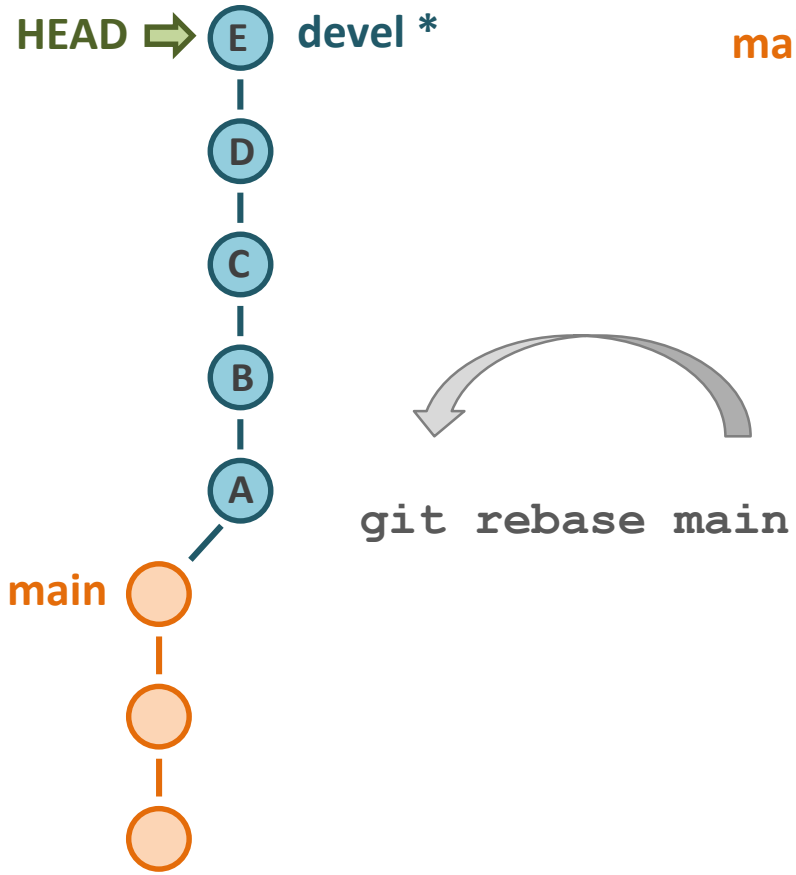
interactive rebase

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

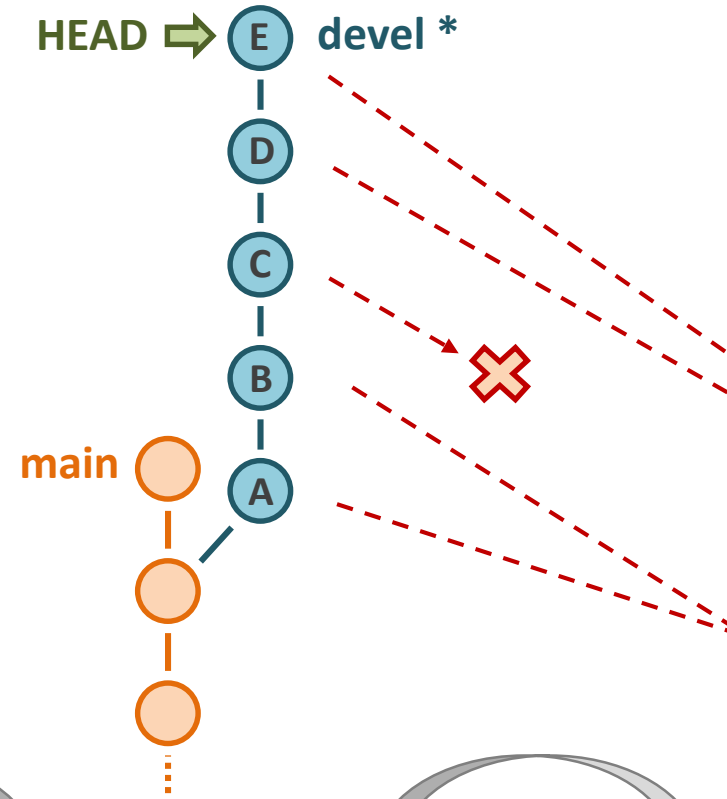


Standard vs. interactive rebase

Standard rebase
replay commits on top of another base commit.



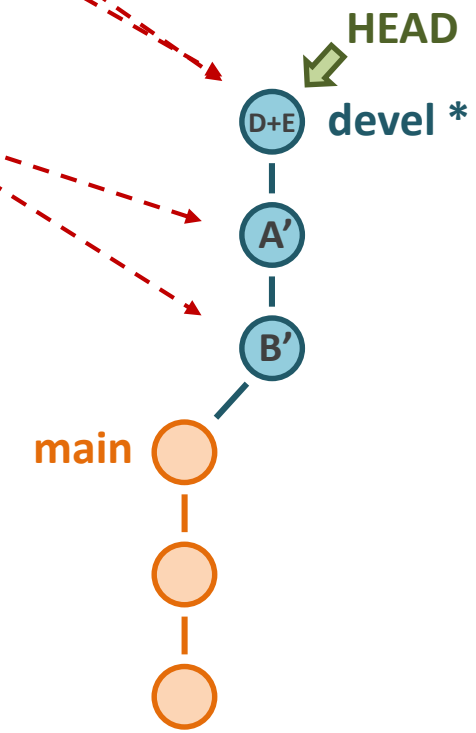
`git rebase main`



`git rebase --interactive main`

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

- re-order
- delete
- merge (squash)

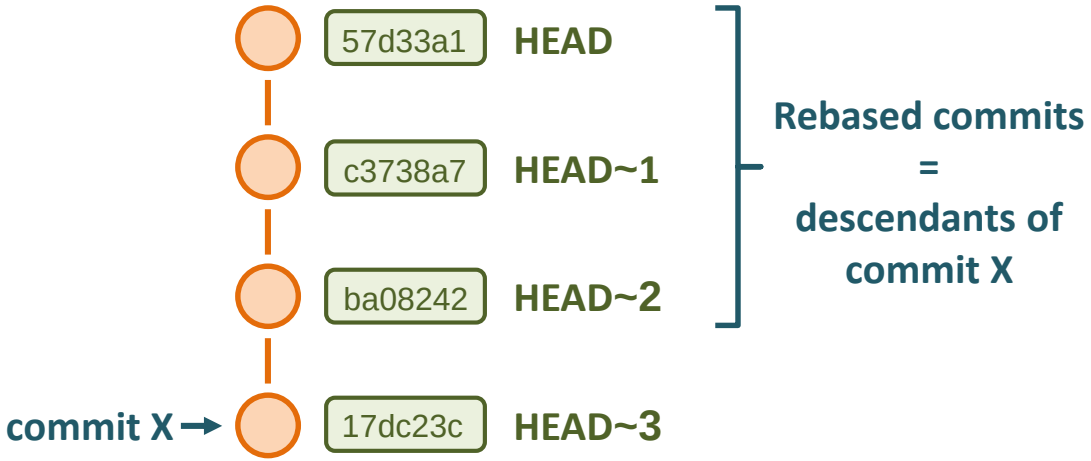


Interactive rebase: re-order, squash, and delete commits

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

parent of first commit in the rebase

- Starting from (just after) 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.
- Git then rewinds to `<commit X>`, and re-applies the descendant commits as instructed.



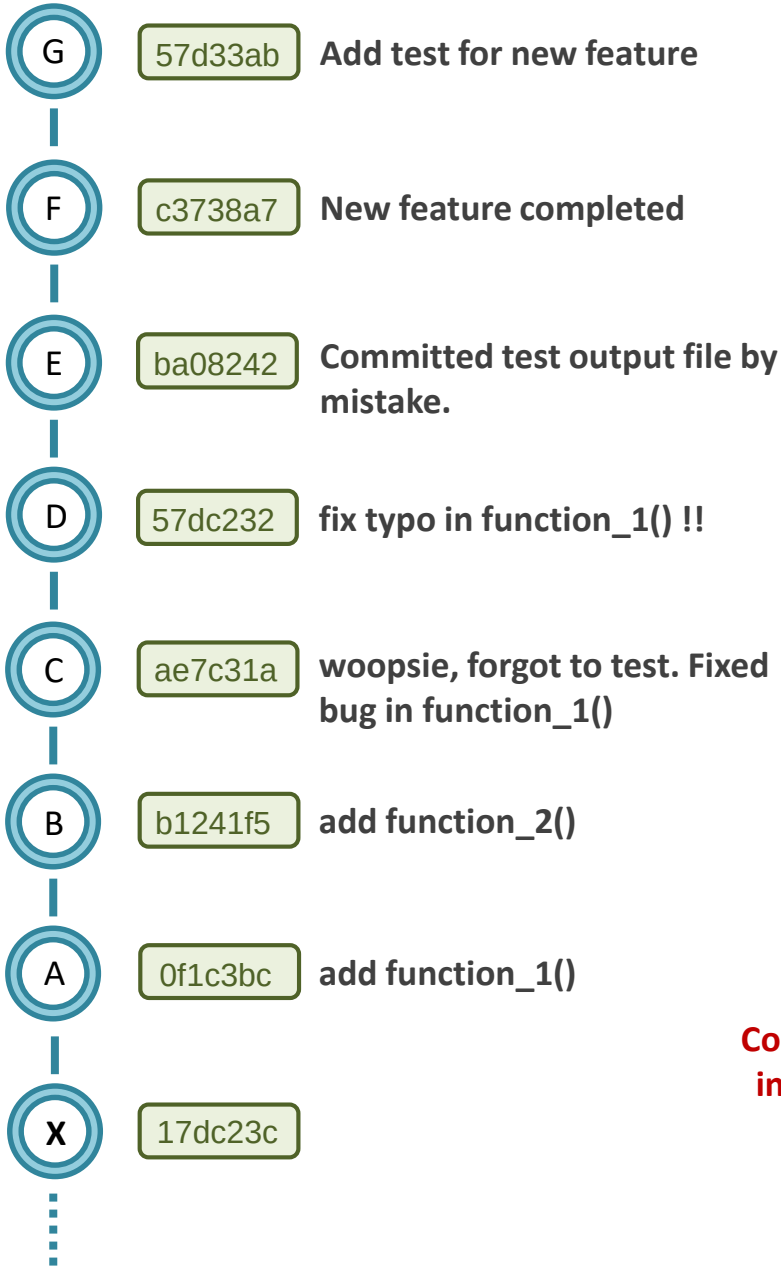
To rebase the last 3 commits (descendants of commit X), these 2 commands will yield the same result:

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

Absolute reference to commit X

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

Relative reference to commit X



Merge commits

delete !

Re-order and merge with A

Reversed order - oldest on top!

Commits are re-applied in top to bottom order

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

or

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

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

```
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.

```
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).

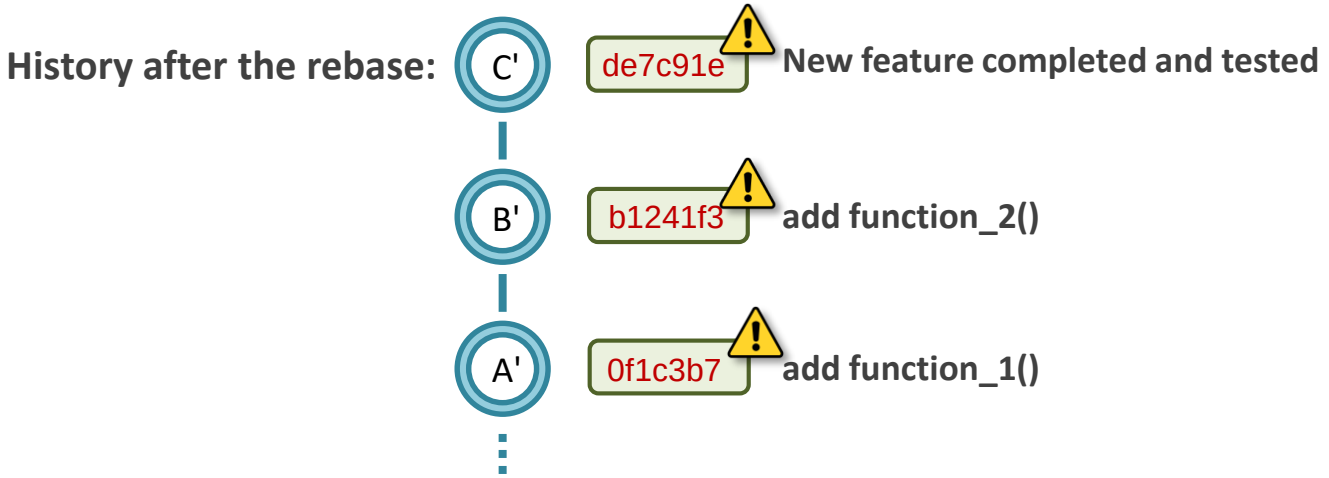
For squashes, Git will open an editor so you can edit the commit message.

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 !



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)

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

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 drop 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).

Supplementary material...

--fixup commits

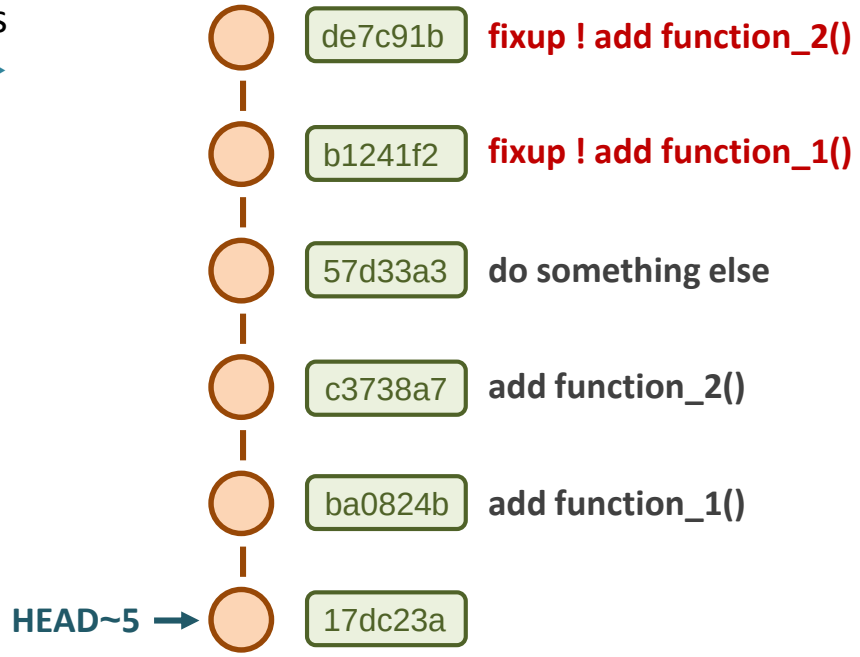
- When you realize you made a mistake in an earlier commit, you can directly tag it as a fixup with `git commit --fixup=<hash/ref of commit to be fixed>`
- Running an interactive rebase with the `--autosquash` option added, Git will automatically re-order commits for you.

```
# 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
$
```

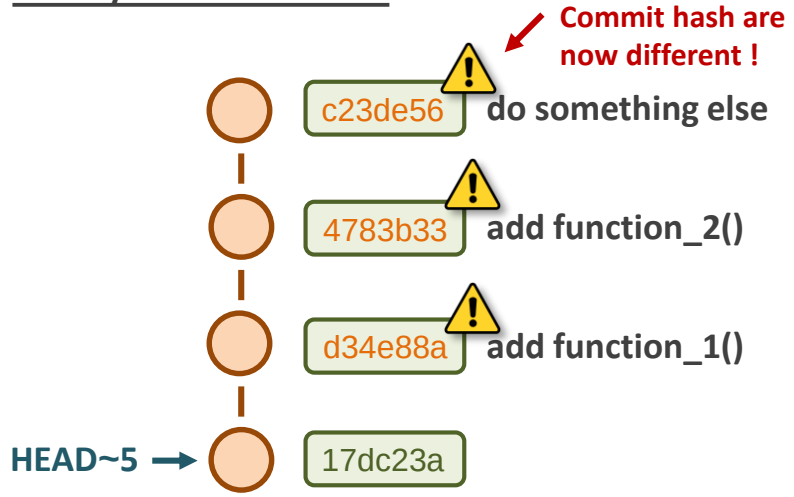
with the `--autosquash` option enabled, Git automatically places the fixup commits in at the correct position, and marks them as "fixup". No manual editing needed !

```
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.



History after the rebase.



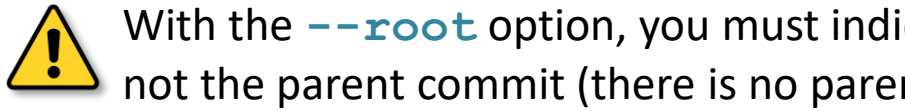
Supplementary material...

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 main
```

Supplementary material...

demo: interactive rebase

exercise 1

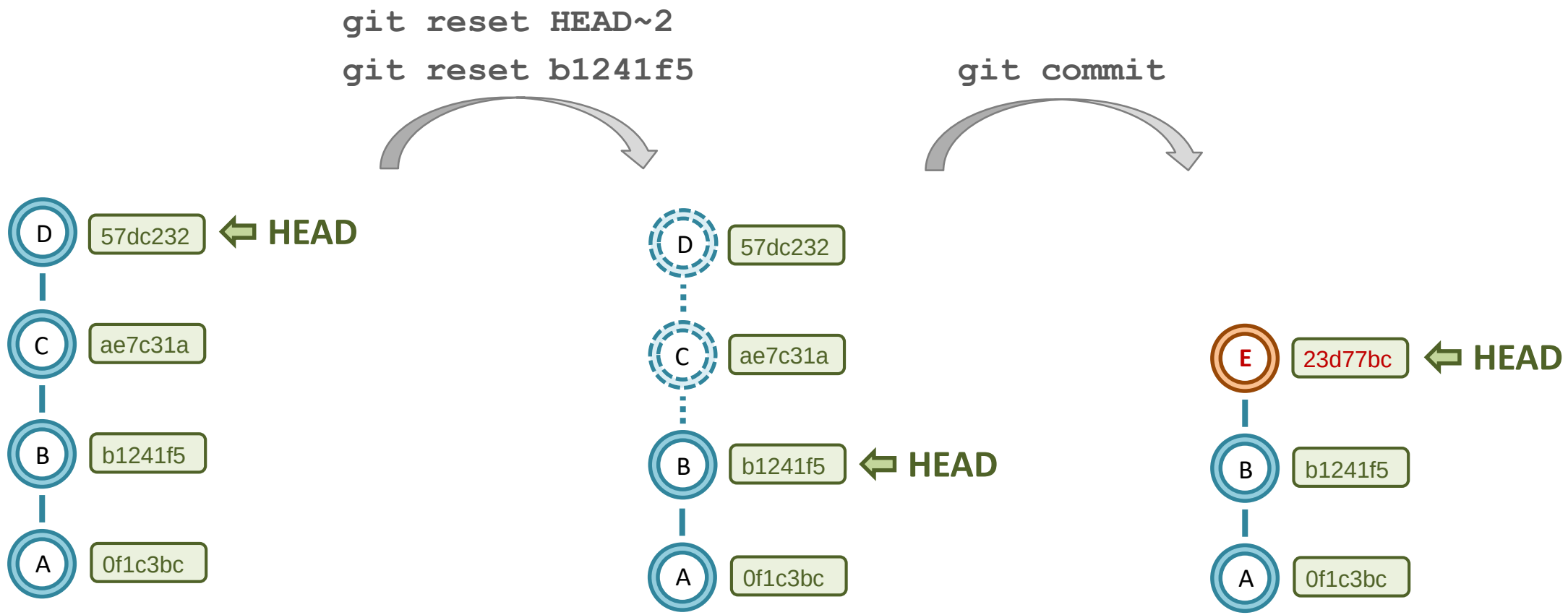
The vim cheat-sheet rebase

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 – 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.

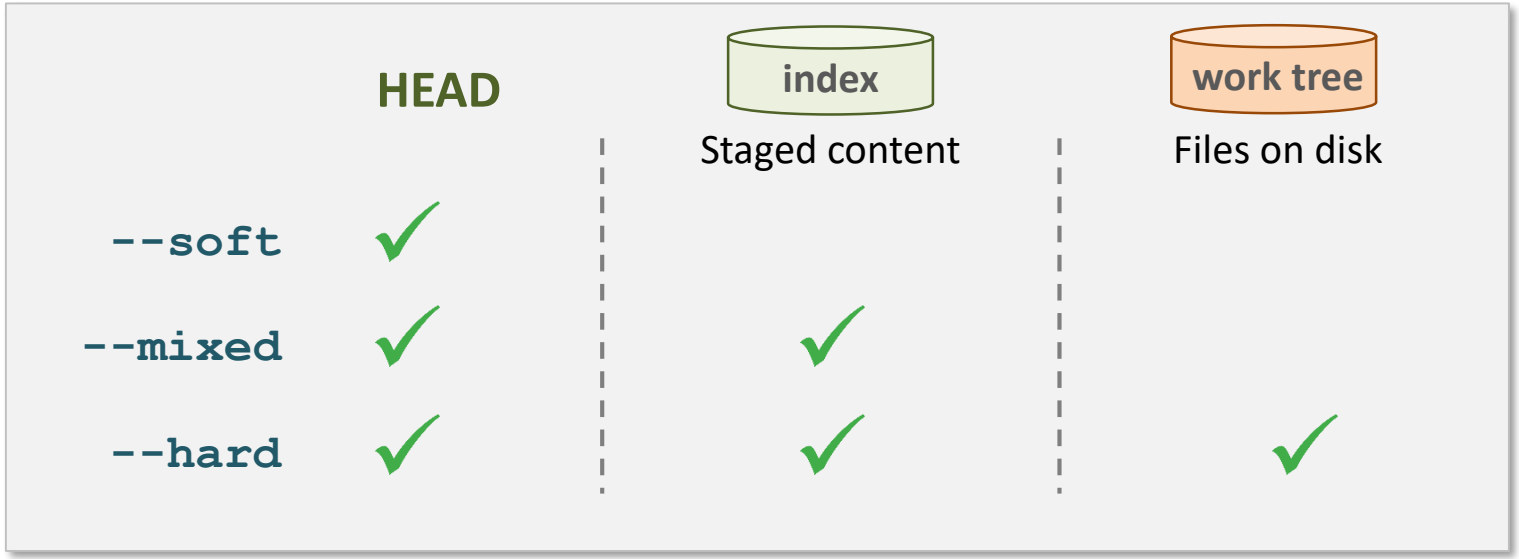
The --hard option resets (overwrites) the working tree !

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

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

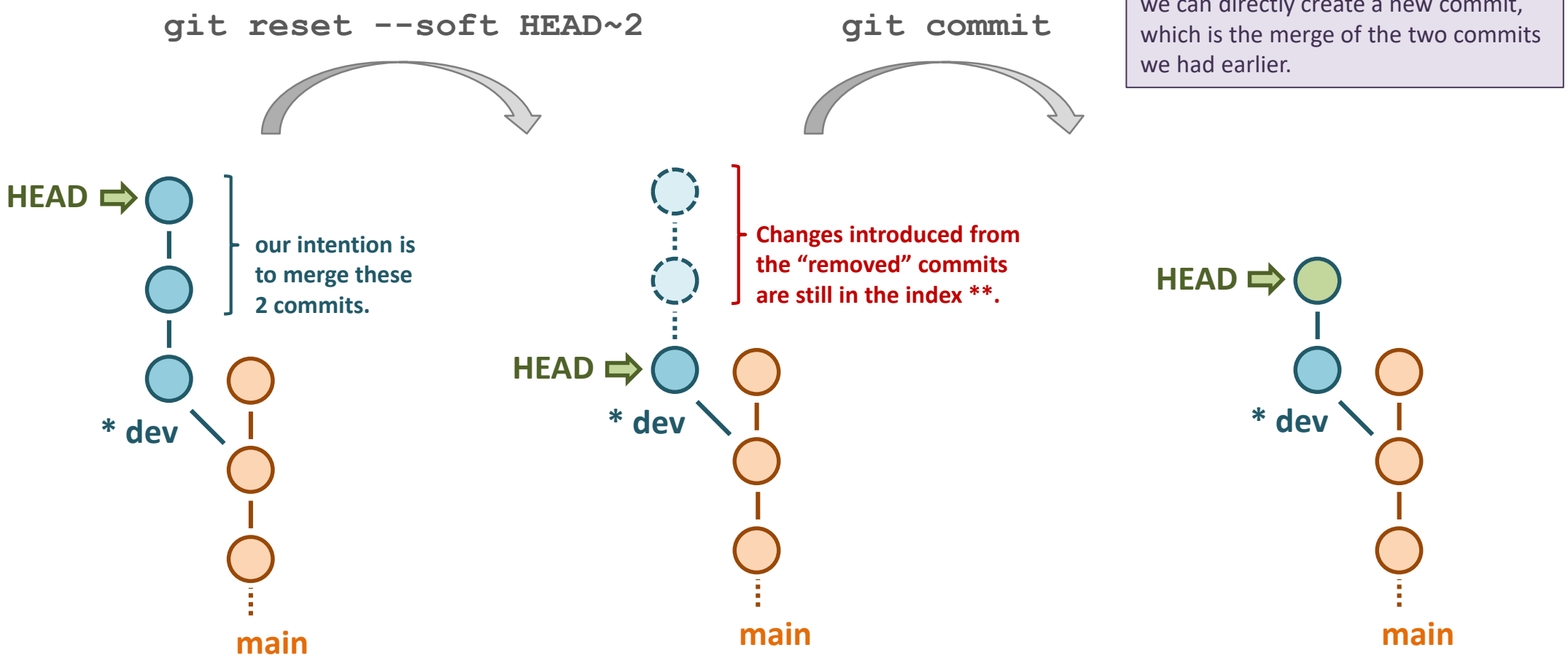
mixed is the default value (so you don't need to actually specify it)

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



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

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



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

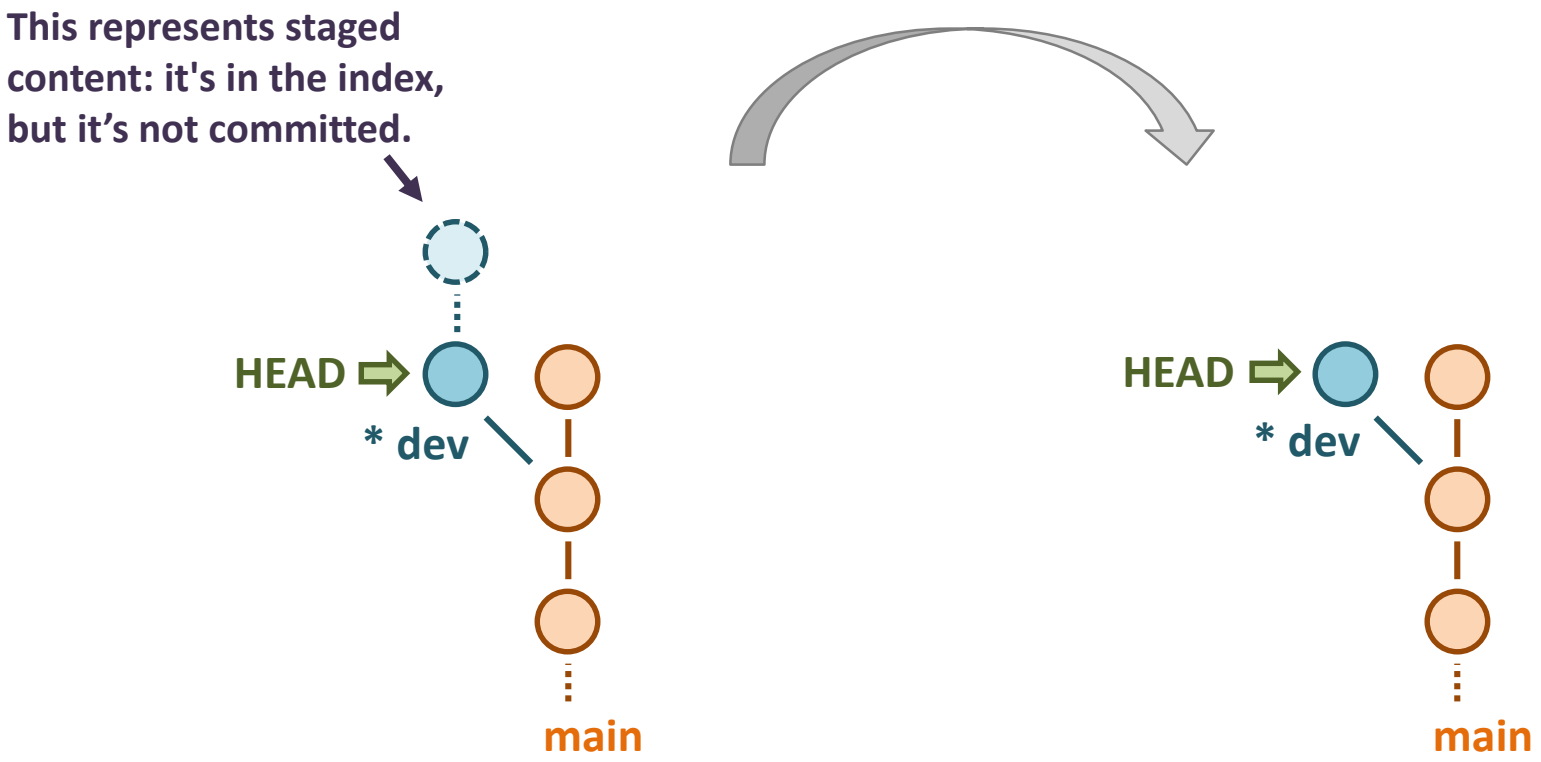
The **HEAD** was reset, but the modifications introduced by the "removed" commits are still in the index and the working tree.

** 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
```

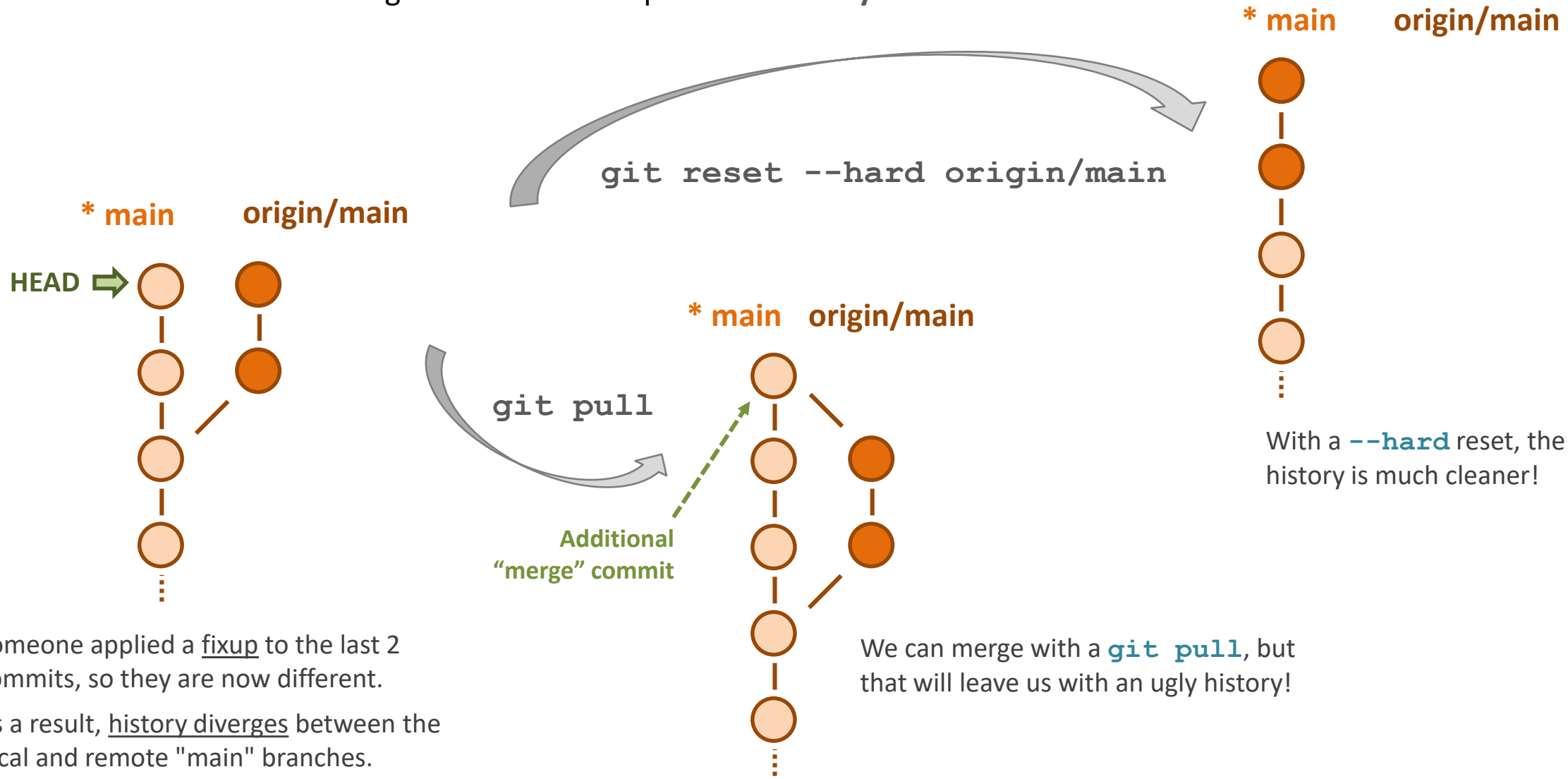


The newly staged content is now removed from the index, because the index was reset to its state at the **HEAD** position.

But any changes made in the working tree is still there: `--mixed` does not modify the working tree, 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**.



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

As a result, history diverges between the local and remote "main" 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!

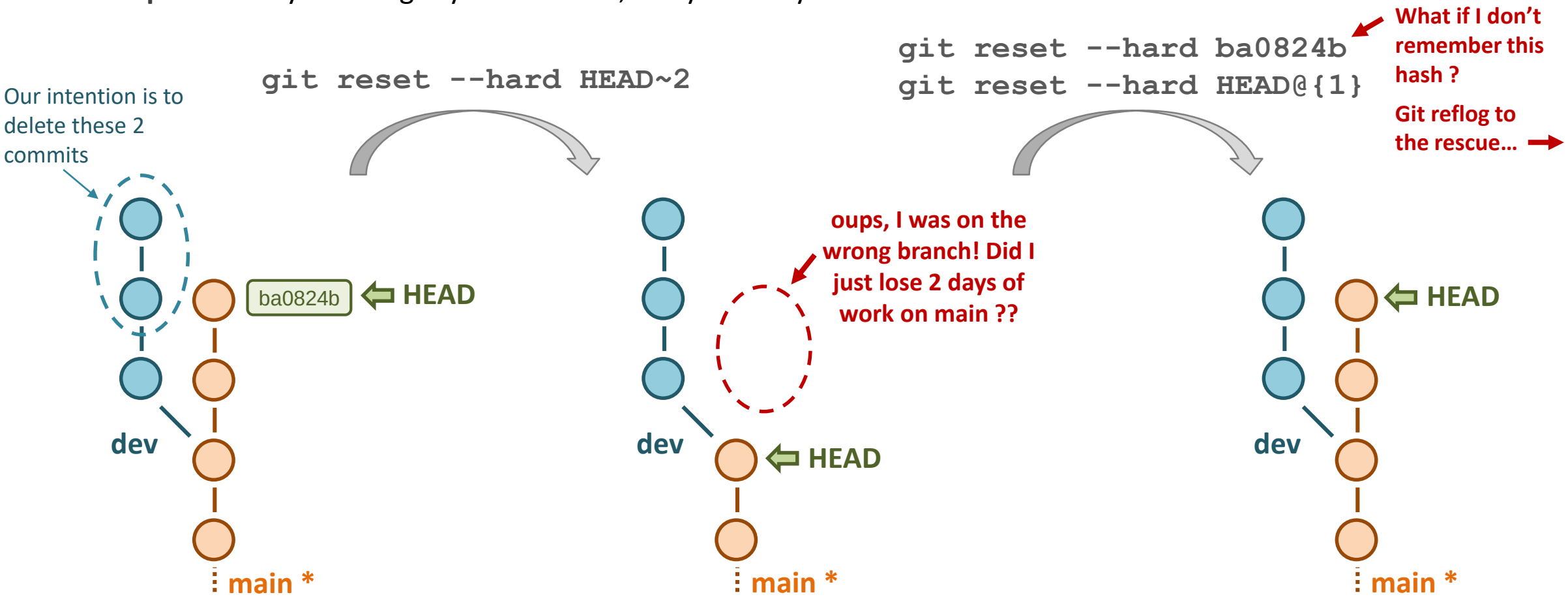
git reset --hard use case: reset a reset, a merge, a rebase (or anything, really)

- 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 main…



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

Commit IDs of commit at HEAD position

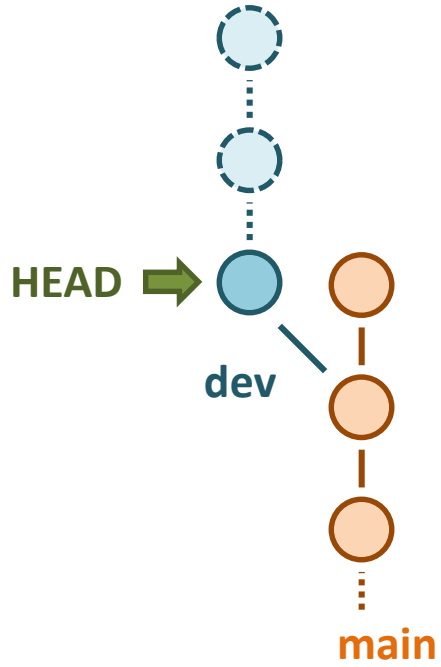
- The `HEAD@{x}` notation indicates the positon of the HEAD pointer relatively to the reflog.
- It can be used as a commit reference, e.g. `git reset --hard HEAD@{1}`.

`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.

Supplementary material...

What happens to *unreachable/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 ***unreachable/orphaned***. E.g. in the diagram on the right, after a `git reset`, two commits (dashed circles) are now unreachable/orphaned.



- Unreachable 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 unreachable/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. In addition 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 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
```

** Default “prune” time

By default, unreachable commits (and other objects are garbage collected after 14 days). This setting can be changed with the config option:

```
git config gc.pruneExpire 2.weeks.ago
git config gc.pruneExpire 3.months.ago
```

Supplementary material...

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
```

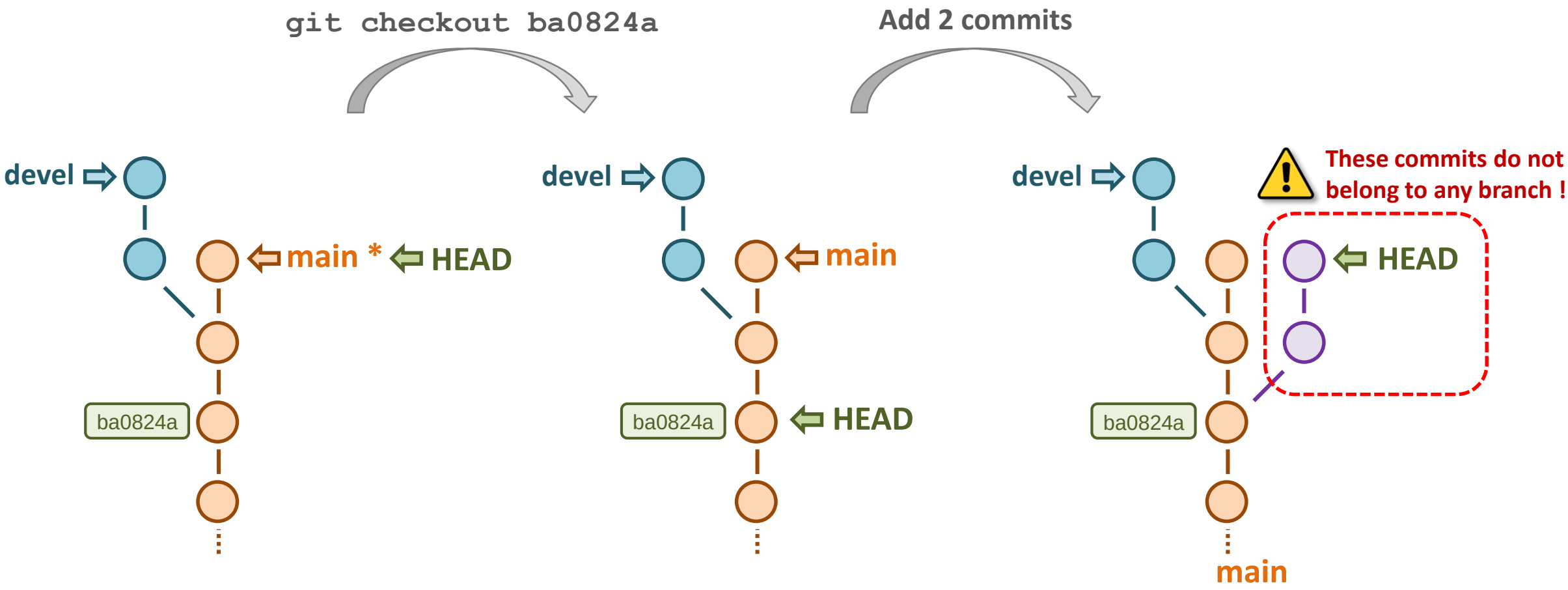
- But you will enter a "detached HEAD" state….

```
$ 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>`

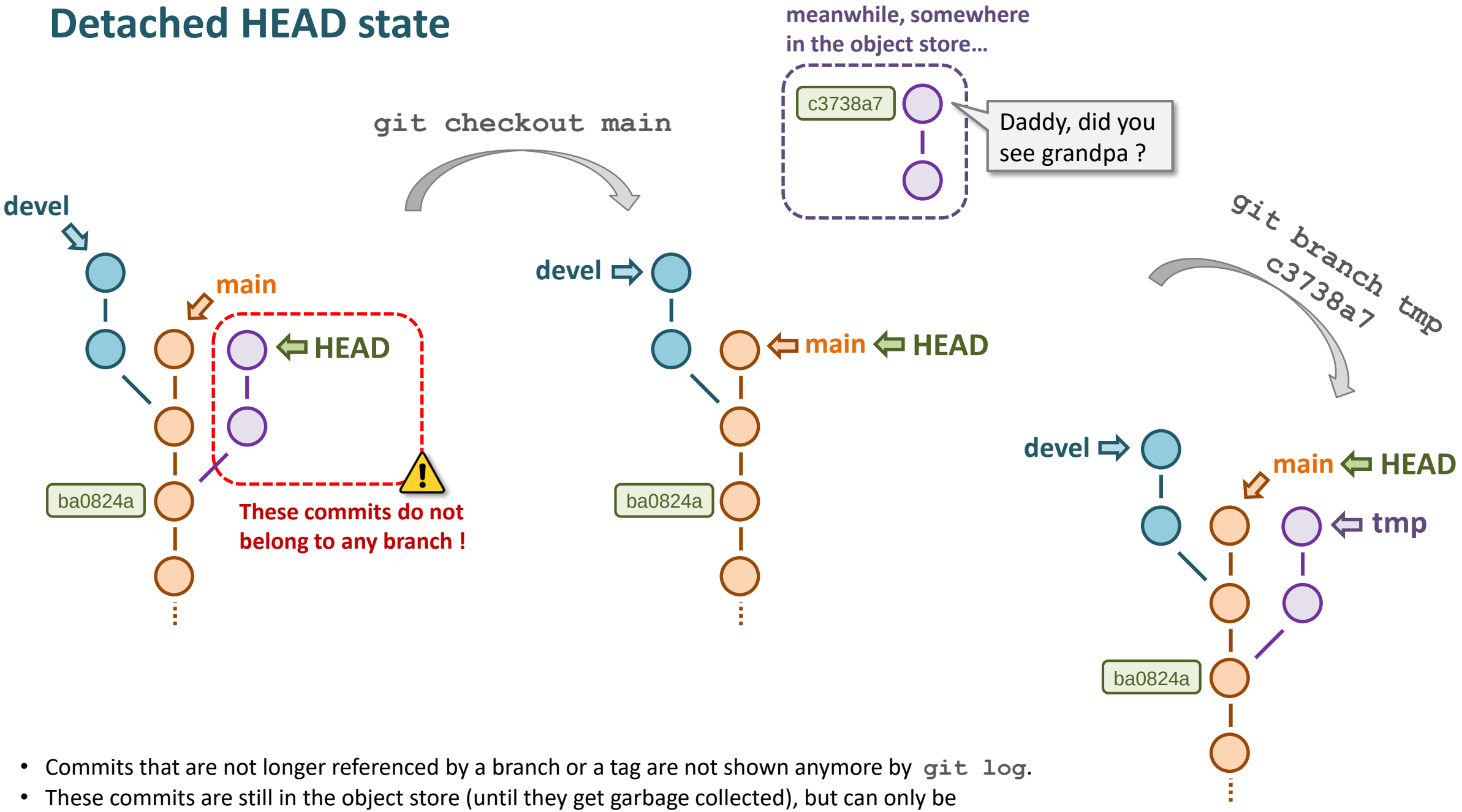
Detached HEAD: when HEAD points directly to a commit instead of a branch

- After a `git checkout <commit>` command, the **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



- 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>
```

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 main
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 'main'
```

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 …

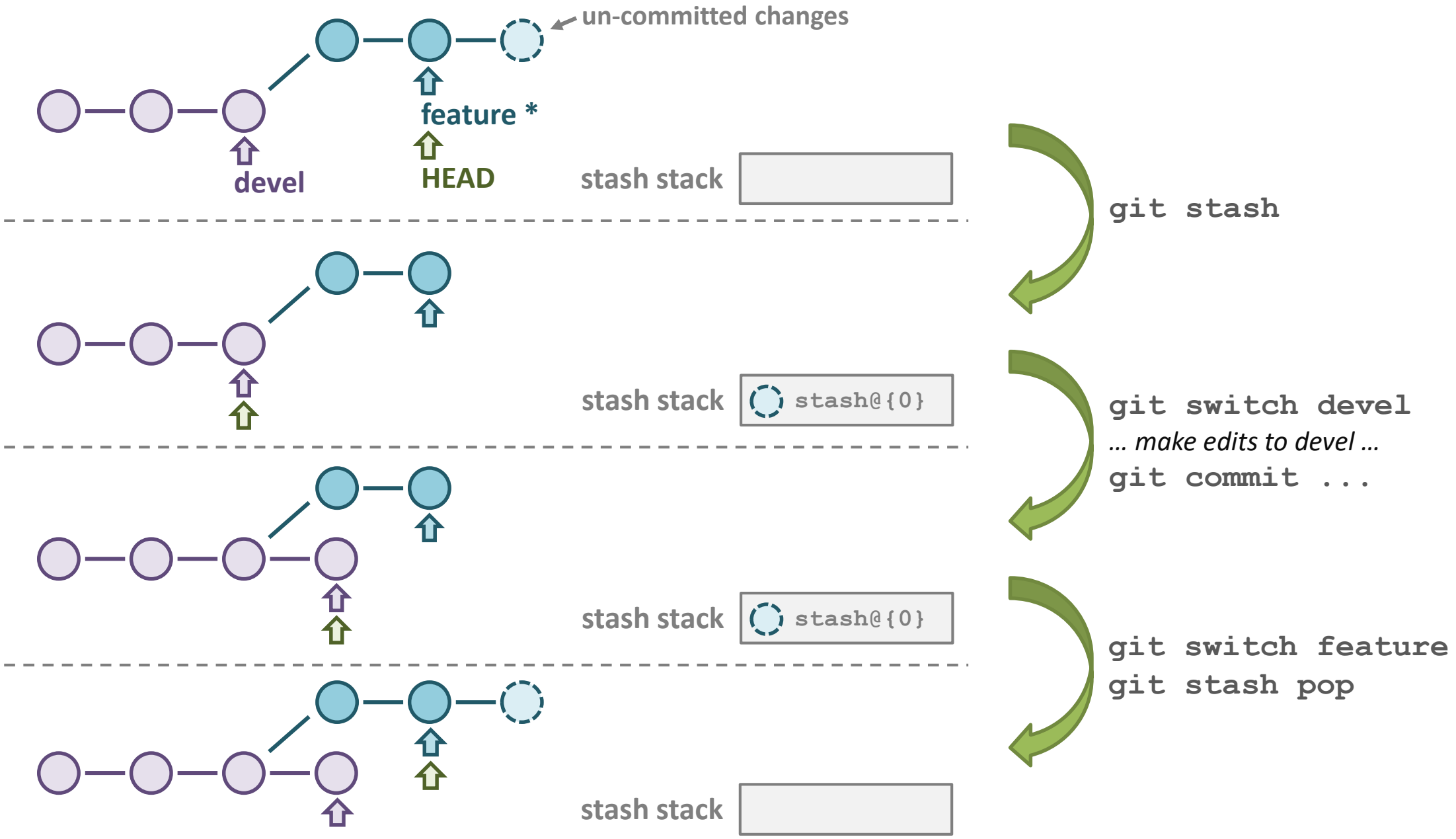
Sometimes we quickly need a clean working tree, but without losing un-committed changes already made to our files. For instance:

- 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).

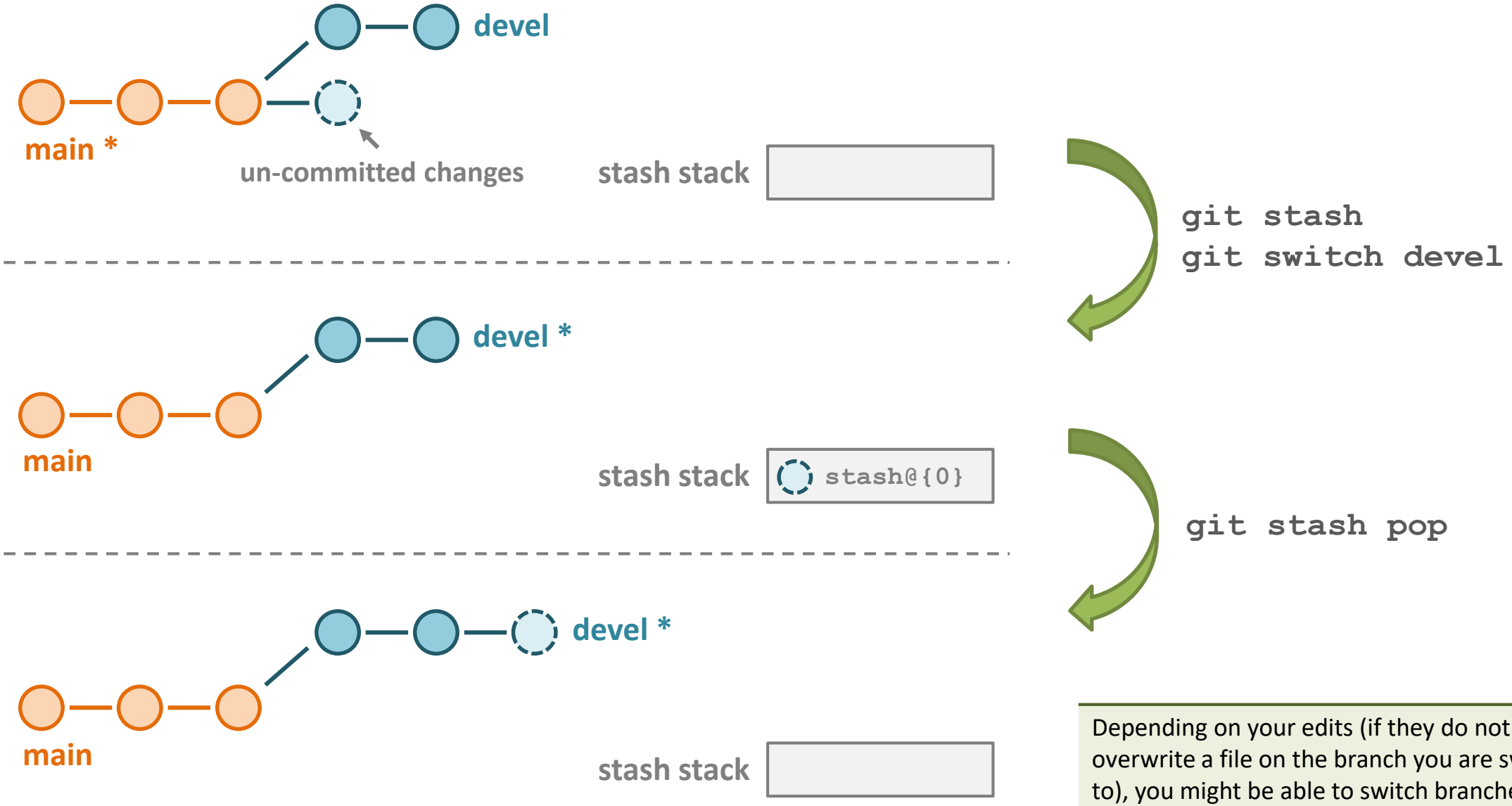
`git stash` ⇨ Saves un-committed changes in the working tree (both staged and un-staged) to a "temporary commit". Then resets the working tree to the current HEAD position (i.e. the last commit in your current branch), leaving a clean working tree.

`git stash pop` ⇨ Restores stashed modification by merging them into the current HEAD (This can potentially require manual conflict resolution).
The restored content is deleted from the stash.

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



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



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.

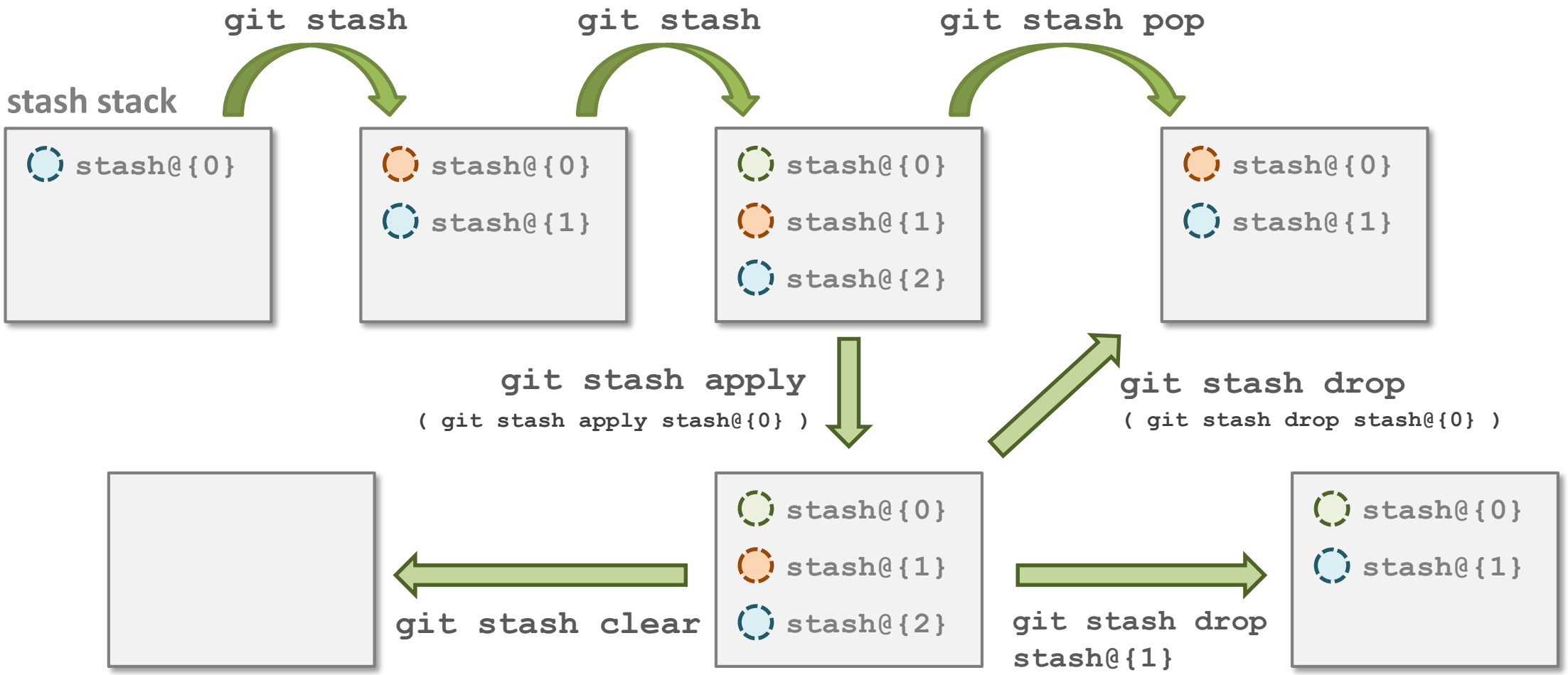
Additional info about git stash…

- 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.
- If needed, the content of the stash can be deleted with `git stash clear`
- `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.



List the content of the stash

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

Example:

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

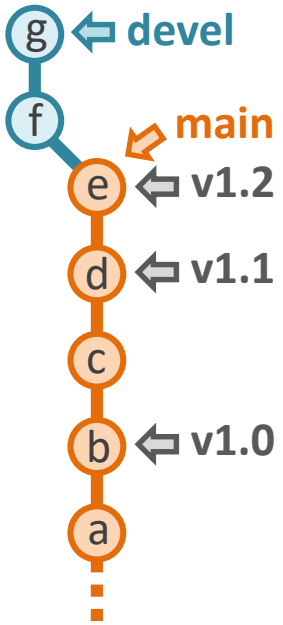
git tags

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:

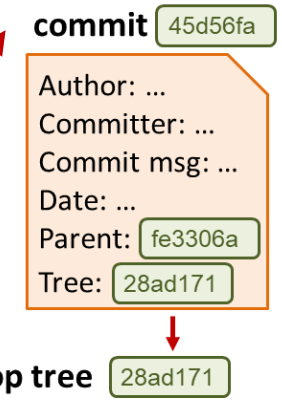
- **Lightweight tags** - pointers to a commit (like a branch).
- **Annotated tags** - pointers to commits with additional metadata:
  - Tagger (person who made the tag).
  - Date and time.
  - Message.

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>`

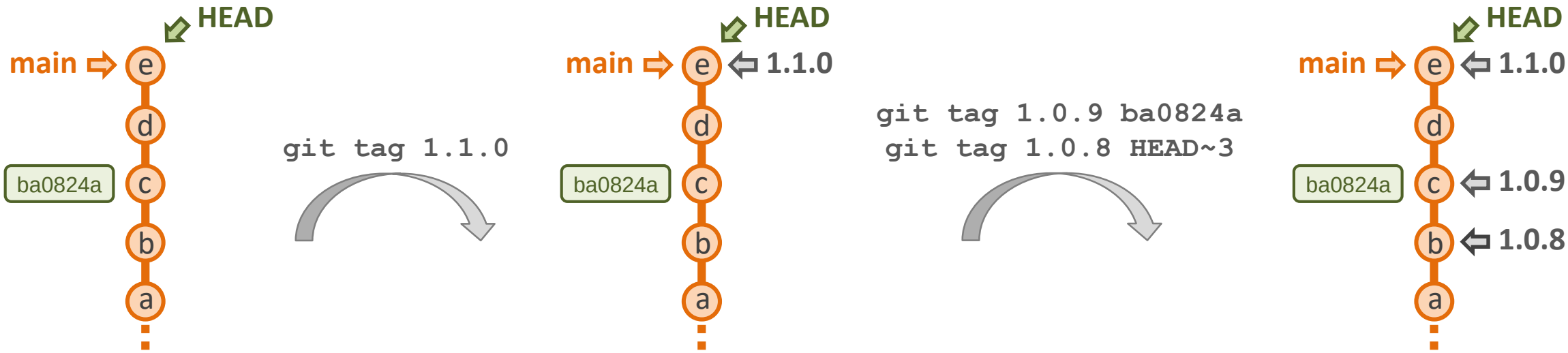
Having a message is compulsory for annotated tags (just like for commits).

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 ".".



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 tags to a 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
```

Delete tags

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

Example:

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

This will not remove the tag from remotes !

origin is the default name used for remotes. If your remote has a different name, you should replace **origin** with its name.

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

Note: this is the same command as for deleting a branch from a remote.

Example:

```
$ 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.

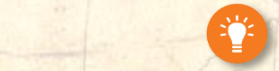
exercise 3

The backport

exercise 4

The treasure hunt

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

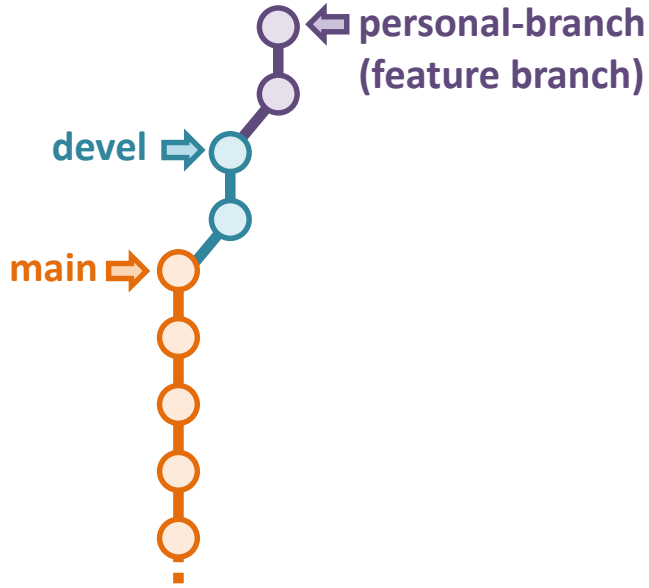


This exercise has helper slides

Introductory notes

While this exercise is somewhat *gamified*, 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.

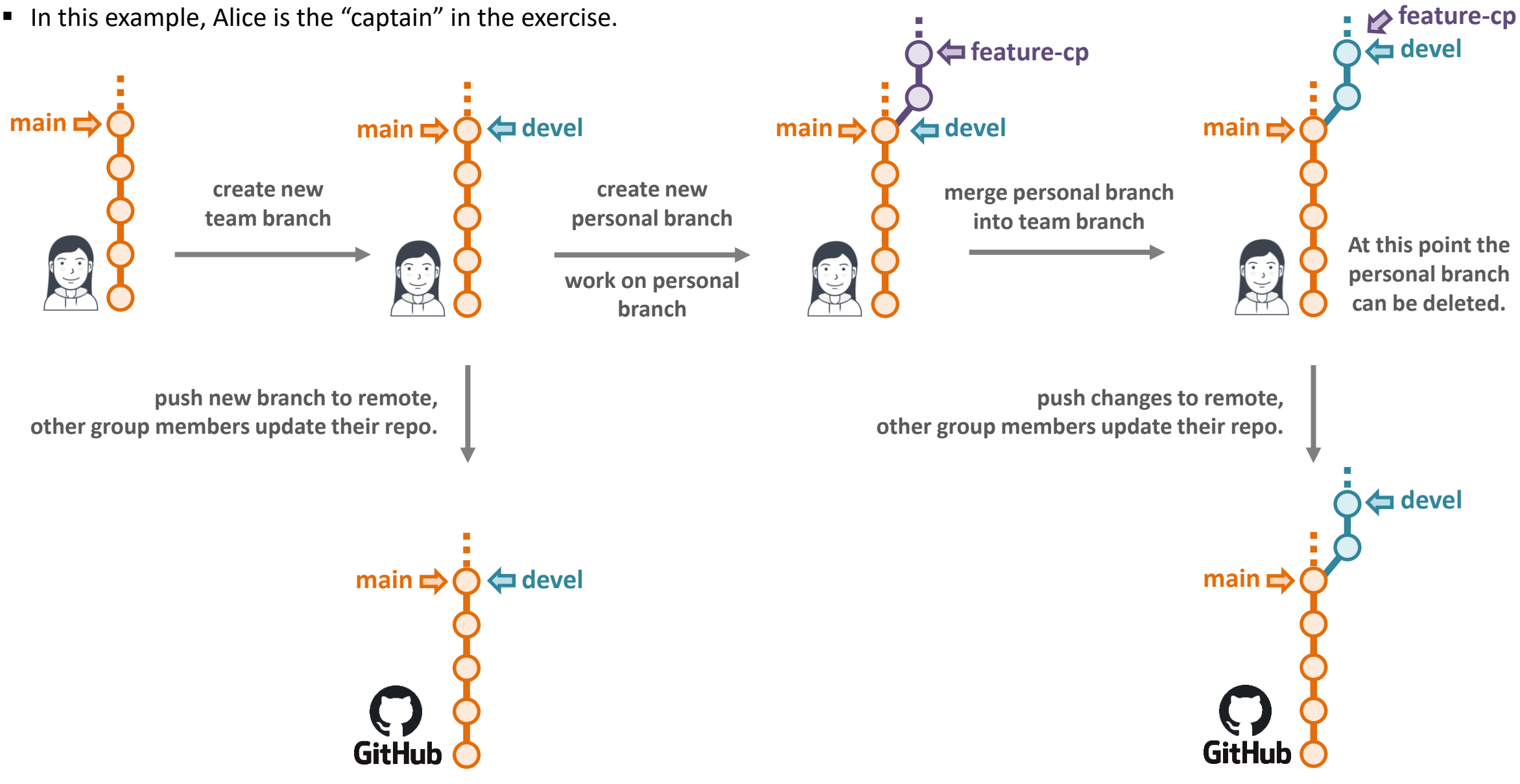


About the branches used in the exercise:

- **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/feature 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.

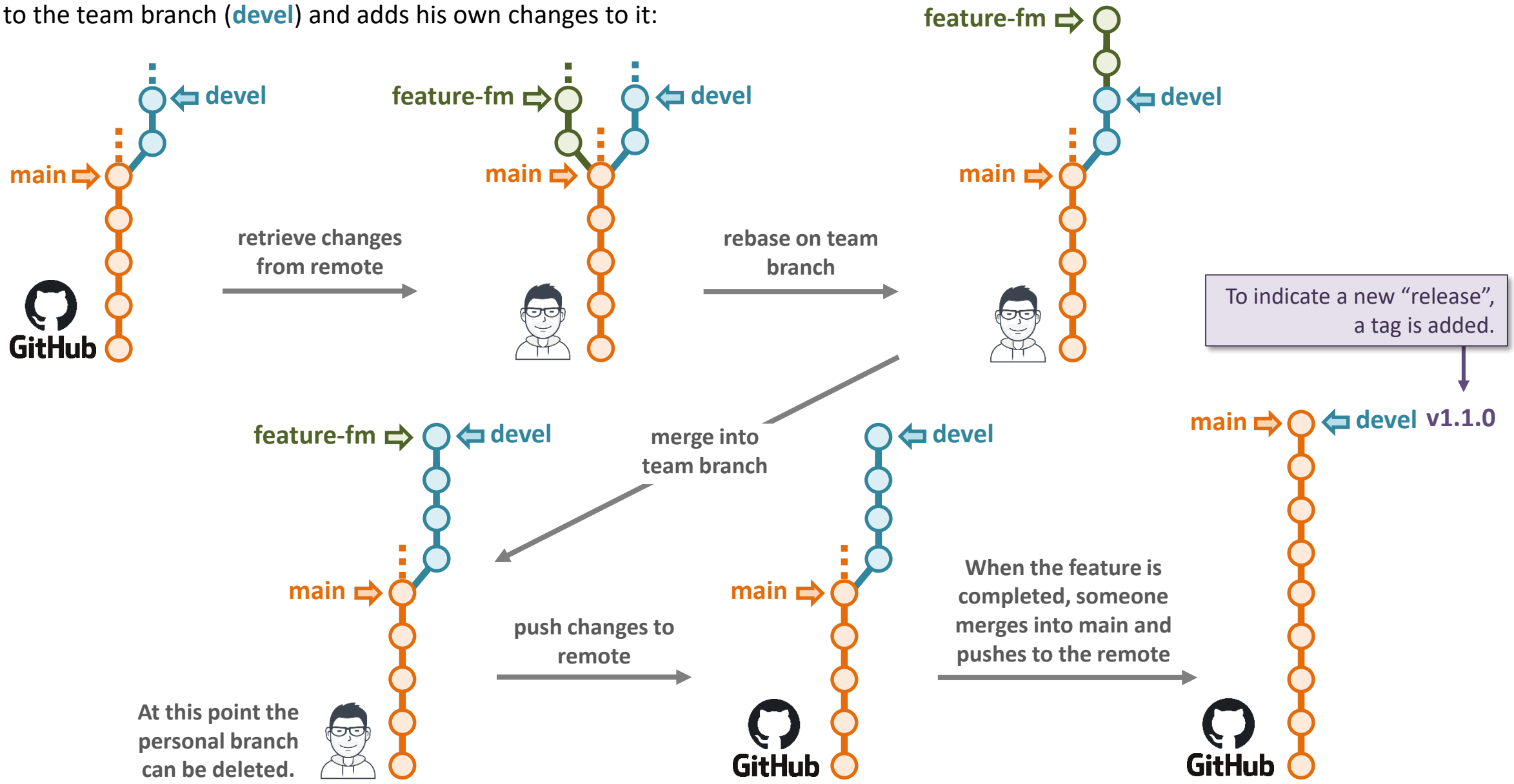
Exercise 4 help: branch – rebase – merge sequence

- 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.



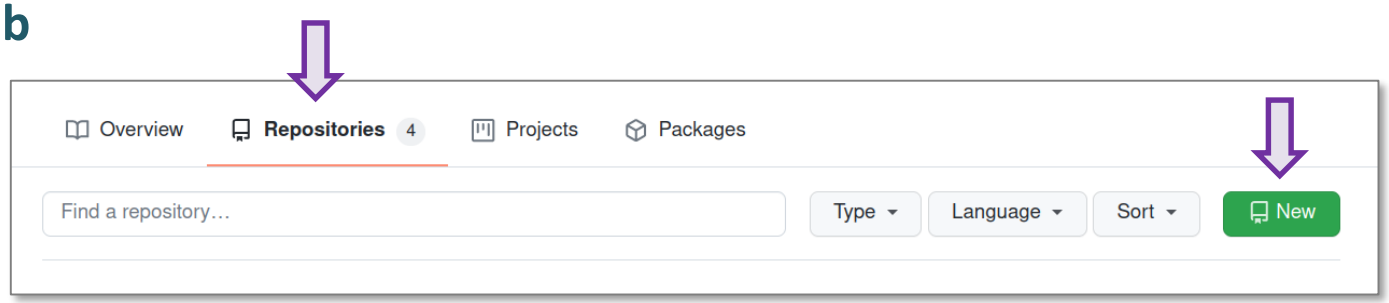
Exercise 4 help: branch – rebase – merge sequence

- 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:

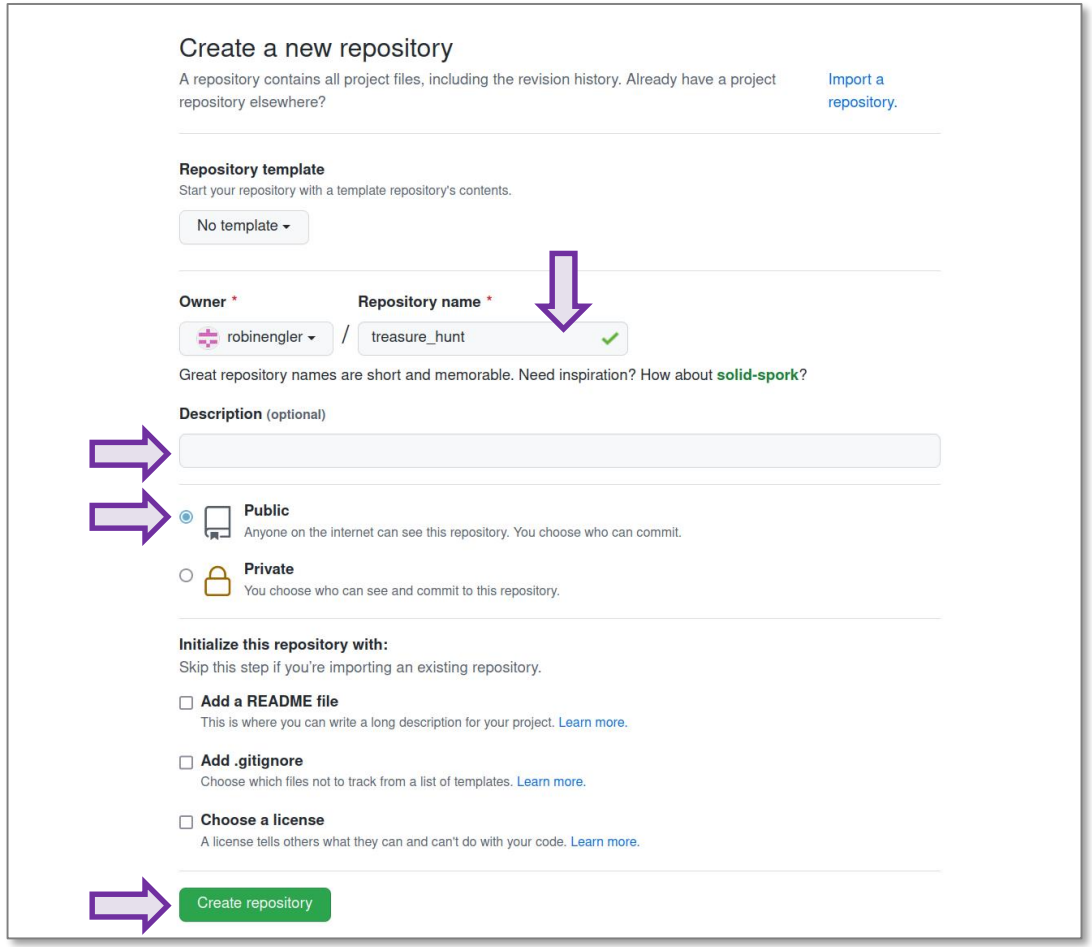


Exercise 4 help: creating a new repo on GitHub

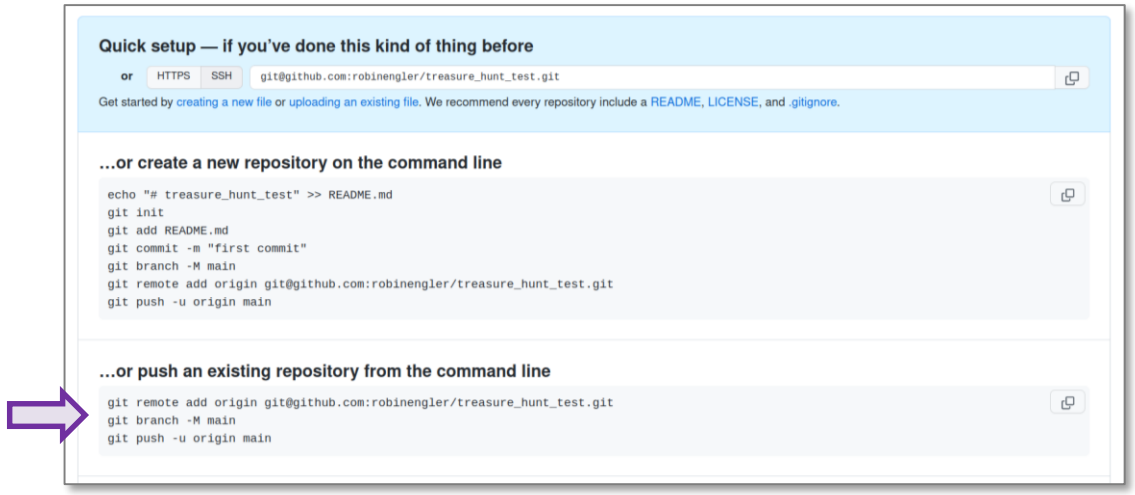
1. In your GitHub account, go to **Repositories** and click on **New** (green button).



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**.

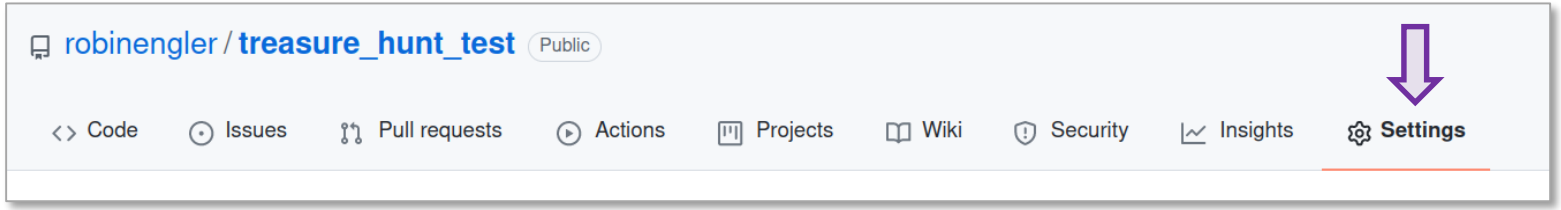


3. Follow instructions to **push an existing repository…**.
   Note: the main branch's name is already "main", so you can skip "git branch -M main".

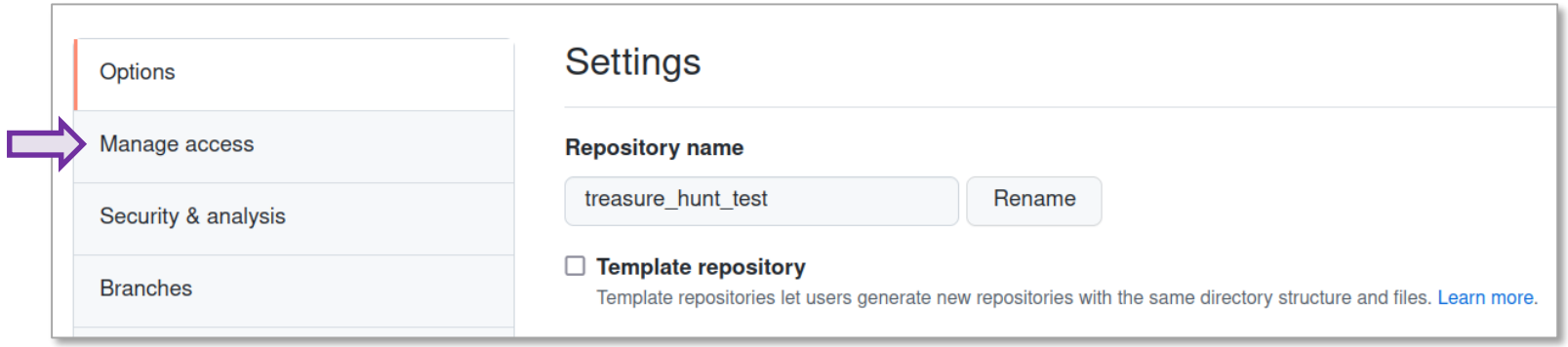


Exercise 4 help: adding members to a GitHub repo.

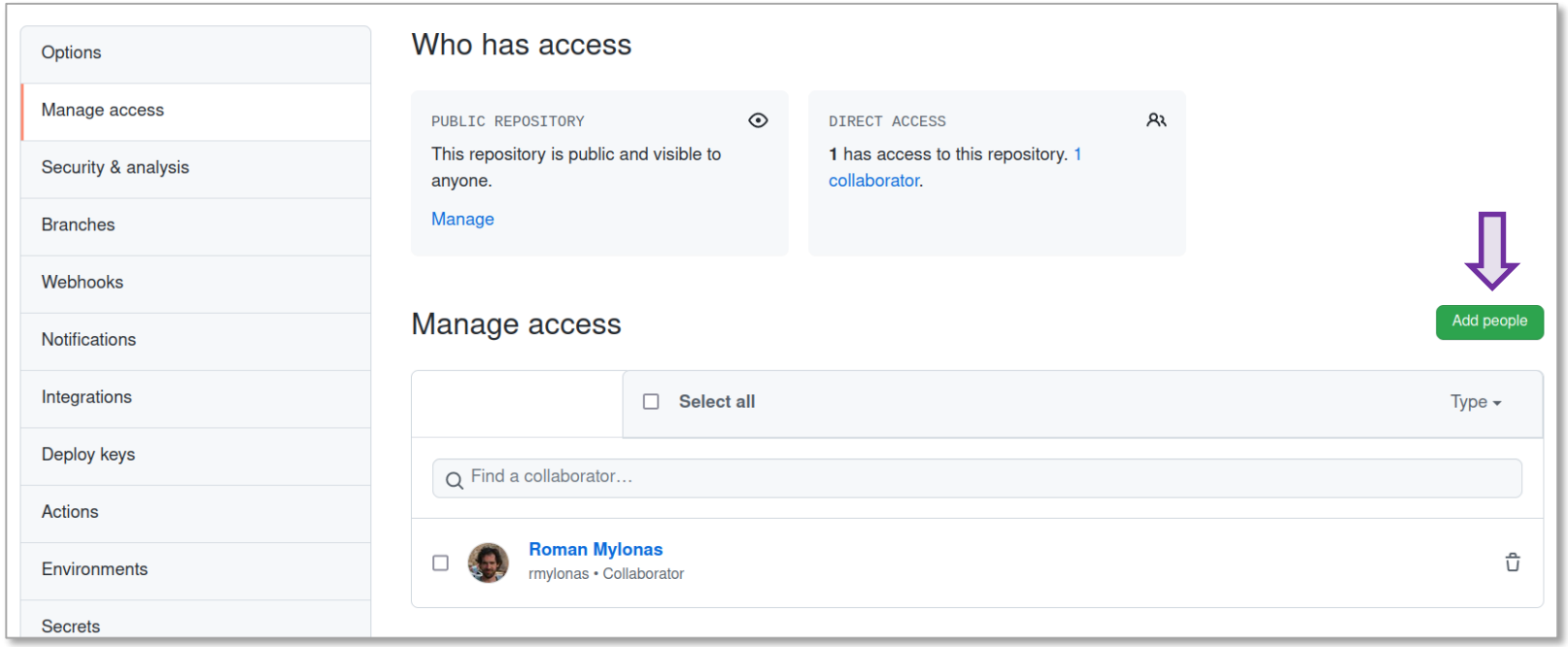
1. On the homepage of the repo on GitHub, select the **Settings** tab.



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



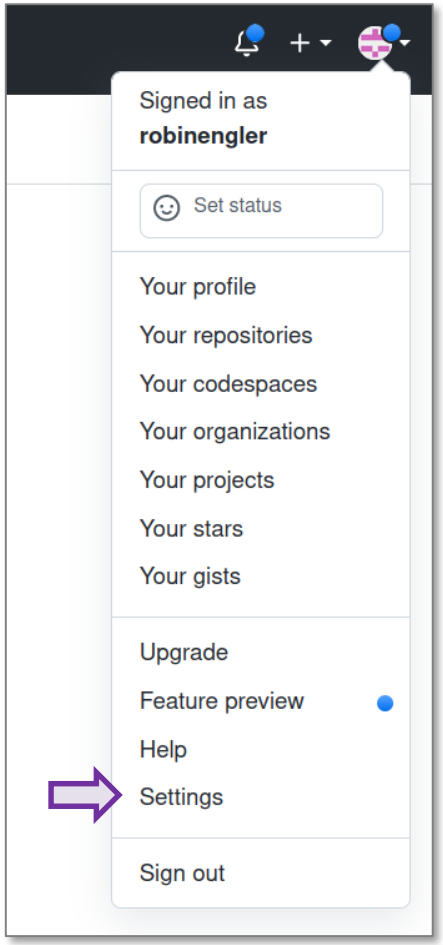
3. Add your team members by clicking on **Add people** (green button) and entering their GitHub user name.



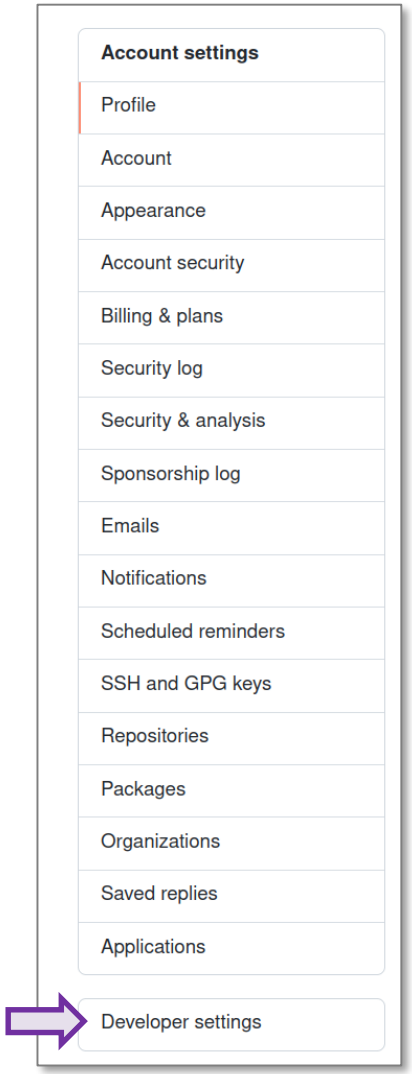
Exercise 4 help: generating a “personal access token” on GitHub

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

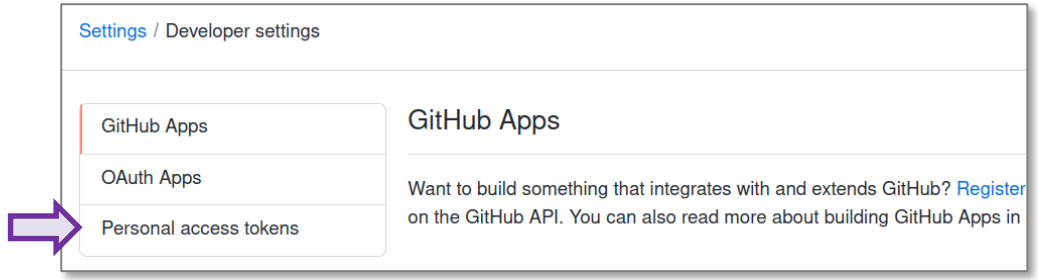
1. In your user profile (top right), click on **Settings**.



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



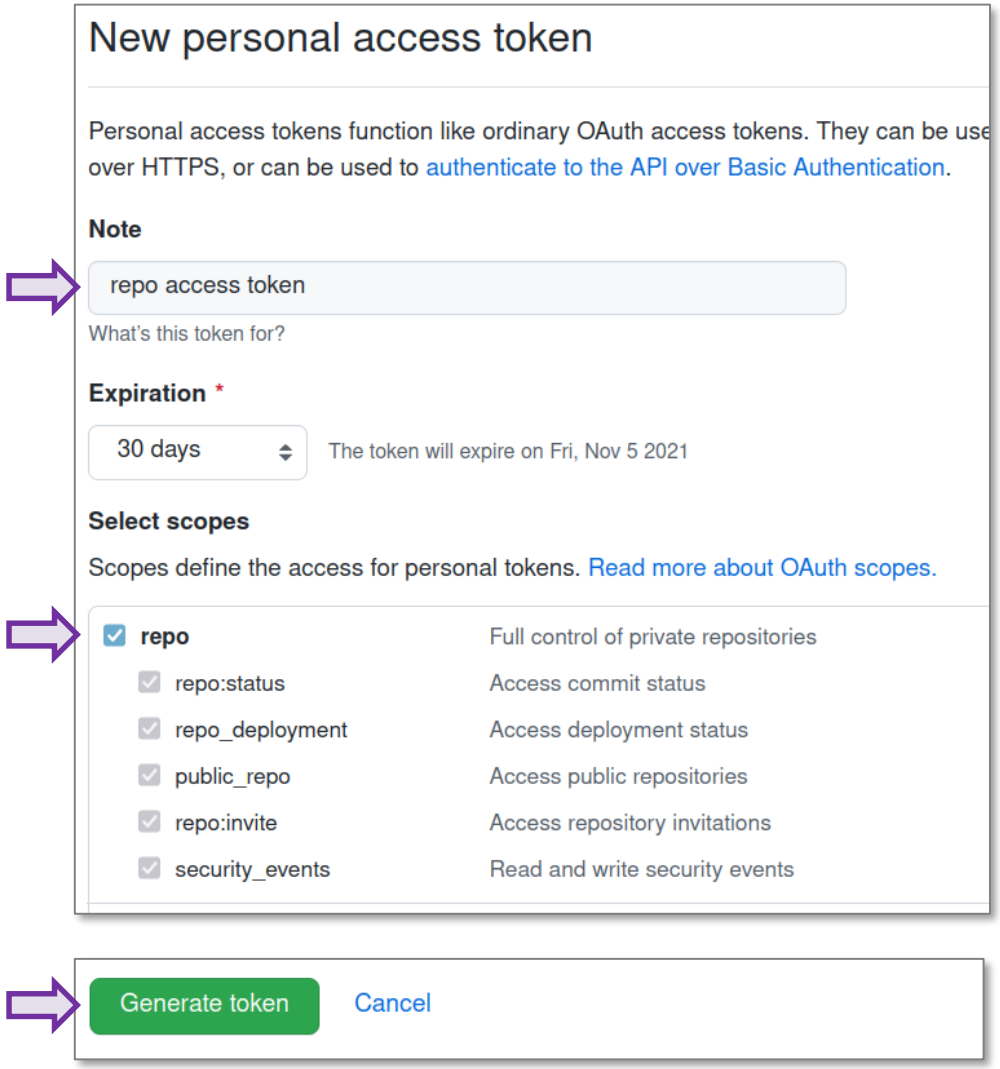
3. In **Developer settings**, click on **Personal access tokens**.



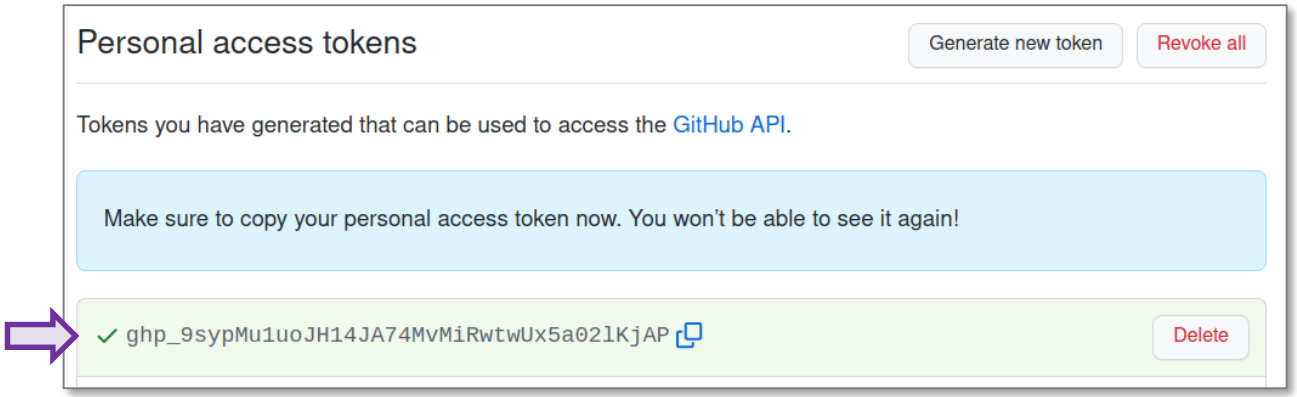
Go to next page

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**.



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.



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.

a look under **git's** hood

The Git object store

Supplementary material...

The Git object store

- The "object store" is where Git stores the data and metadata of the tracked files and commits.
- It's located in `.git/objects`

- Git stores data in 4 object types, all saved in the object store [`.git/objects`]:

Blobs: binary, compressed, file that stores the content of a file.
"blob" stands for "Binary Large OBject" (even if the object is not necessarily large)

Trees: Dictionary linking file names to blobs for a given directory.

Commits: metadata of each change introduced into the repository:
author, commit message, state of files, etc …

Tags: name (e.g. software version) that points to a specific to a commit.

Supplementary material…

Blobs (Binary Large OBjects)

- File that stores the content of a file (in a binary and compressed format).

- Does not store any metadata about the file, not even the file's name.
  - two files with the same content have the same blob/SHA-1.
  - two files with the same blob/SHA-1 have identical content. This allows fast comparison!

- Blobs are named after their content's SHA-1 hash*, and stored in the object store [`.git/objects`].

```
.git/objects/fa/263b8bb9291aaa5059dad78bb38b63f4318c62
.git/objects/4a/b7e6dbb9b1dd73a3e0292ef1d1b2909d107309
```

For performance reasons, the 2 first characters of the SHA-1 hash are used as sub-directory name (this avoids having too many files in the same directory). The remaining 38 characters are the name of the file.

- Using a hash as file name creates so-called “content addressable” storage: the content of the file defines its location. This avoids any risk of losing content when overwriting files, since any change in a file will result in a new hash, and hence a new location.

* almost: Git adds a few header bytes to the content when computing file SHA-1 values.
you can get the SHA-1 hash computed by Git with: `git hash-object -t blob <file to hash>`

Supplementary material...

Commands in shell

```
$ cd test_project
$ echo "This is just a demo
       project" > README.md
$ git init
```

```
$ git add README.txt
```

```
$ echo "Free as in
       freedom" > LICENSE.txt
$ git add LICENSE.txt
```

```
$ cp README.md README_copy.md
$ git add README_copy.md
```

Content of working tree

```
.git
README.md
```

```
.git
README.md
```

```
.git
README.md
LICENSE.txt
```

```
.git
README.md
LICENSE.txt
README_copy.md
```

Content of object store (.git/objects)

```
.git/objects/
info/
pack/
```

```
.git/objects/
info/
pack/
f5/e333dff2cf029ec213ce4bae9bc94e99381fb6
```

SHA-1 hash of "This is just a demo project"

```
.git/objects/
info/
pack/
f5/e333dff2cf029ec213ce4bae9bc94e99381fb6
b0/282337246891c91e2eb67c87f0cea0923107ac
```

SHA-1 hash of "Free as in freedom"

```
.git/objects/
info/
pack/
f5/e333dff2cf029ec213ce4bae9bc94e99381fb6
b0/282337246891c91e2eb67c87f0cea0923107ac
```

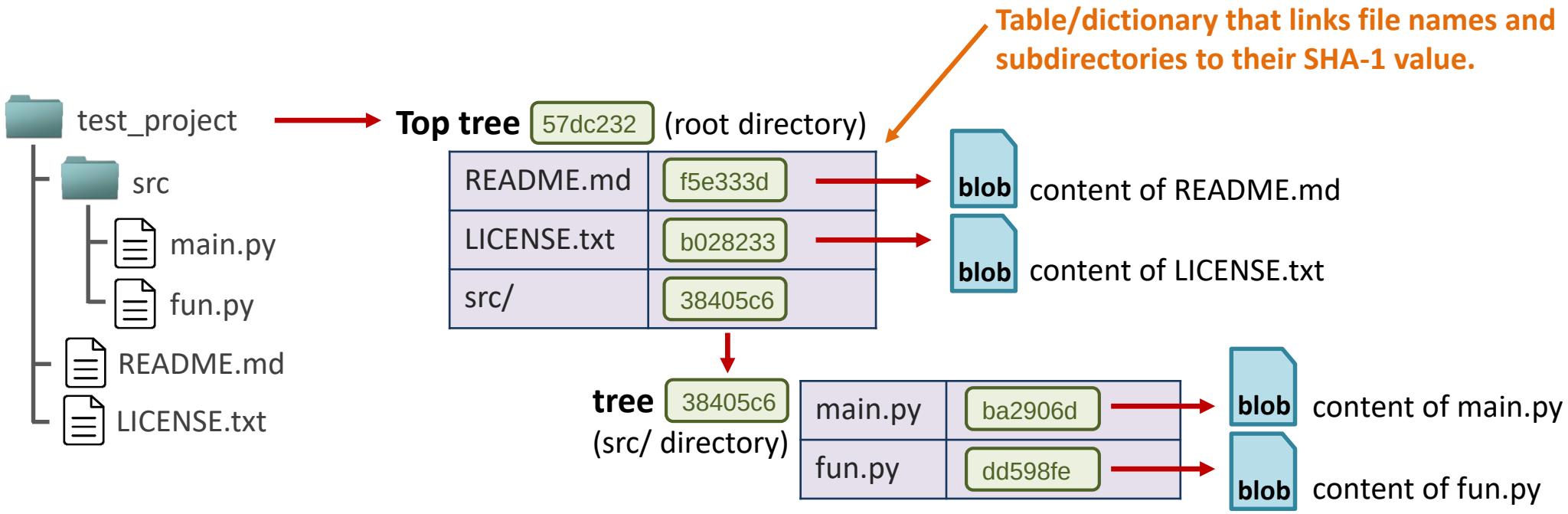
Nothing added to object store!

Because content of file1 and file3 is the same.

Supplementary material...

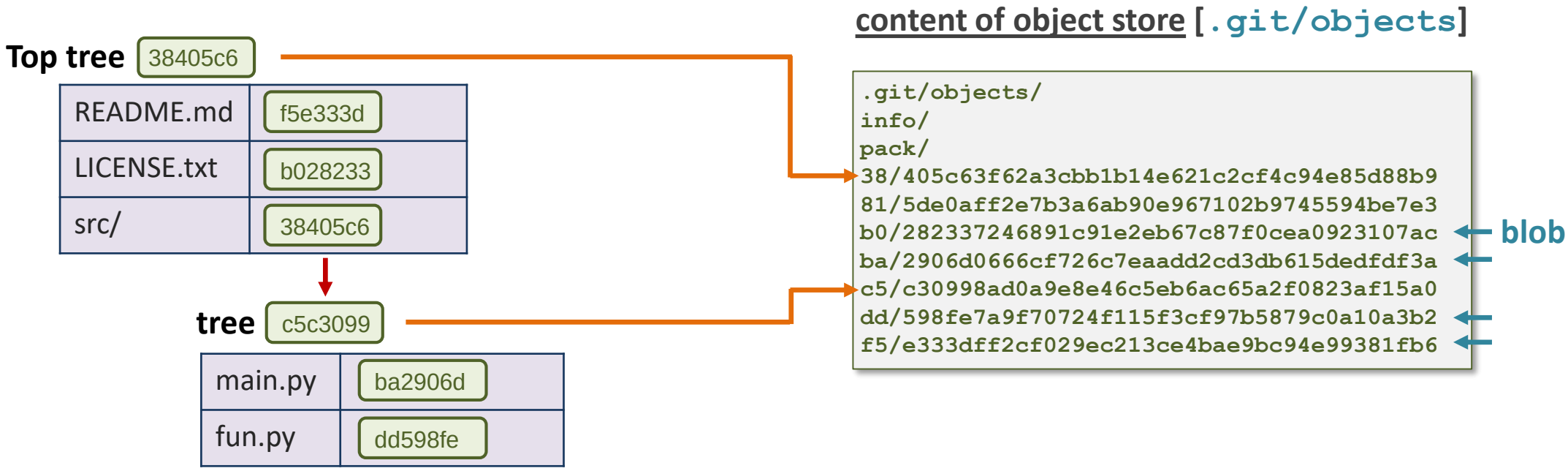
Trees

- Tree = dictionary/table linking blobs to filenames - at a given directory level.
- Sub-directories are also tree objects, referenced by their parent directory.
- **If two trees have the same hash, then their content is identical** – fast comparison as there is no need to look at individual files in the tree's sub-directory.
- The top tree (root of working tree) can be seen as a **snapshot of the entire file content** at a given time.



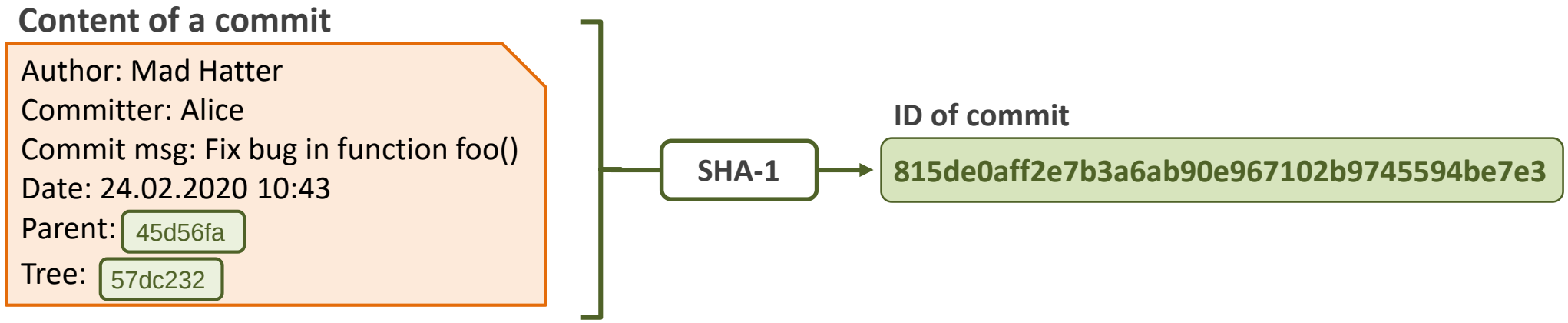
Trees

- Trees are saved in the object store, as a file named after their hash – just like blobs.

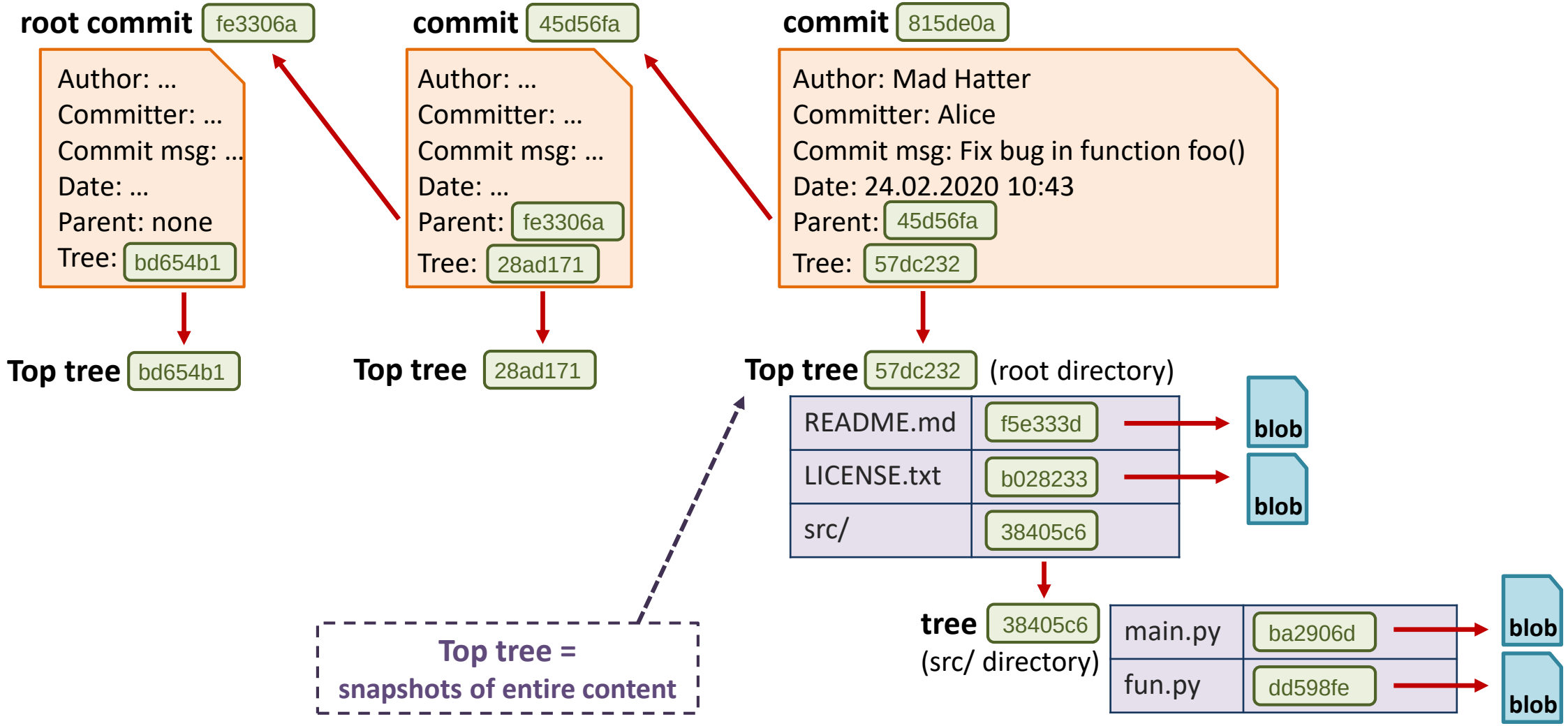


Commits

- Commit objects are lightweight:
  - they are a **collections of metadata**.
  - they **do not contain the data** itself.



- **Commits point to a Tree object** – the top tree object of the Git index content at the time the commit was made. This is how Git can retrieve the state of every file at a given commit.
- **Commits point to their direct parent** – forming a DAG (Directed Acyclic Graph) where no commit can be modified without altering all of its descendants.



Supplementary material…

Commits

- Commits are saved in the object store, as a file named after their hash – just like blobs and trees.

Content of commit

Author: Mad Hatter
Committer: Alice
Commit msg: Fix bug in function foo()
Date: 24.02.2020 10:43
Parent: 45d56fa
Tree: 57dc232

Commit saved in object store, named after its hash

content of object store [`.git/objects`]

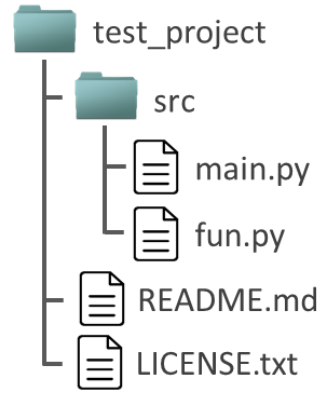
```
.git/objects/
info/
pack/
38/405c63f62a3cbb1b14e621c2cf4c94e85d88b9   ← tree
81/5de0aff2e7b3a6ab90e967102b9745594be7e3
b0/282337246891c91e2eb67c87f0cea0923107ac   ← blob
ba/2906d0666cf726c7eaadd2cd3db615dedfdf3a
c5/c30998ad0a9e8e46c5eb6ac65a2f0823af15a0
dd/598fe7a9f70724f115f3cf97b5879c0a10a3b2
f5/e333dff2cf029ec213ce4bae9bc94e99381fb6
```

`git commit` **triggers the creation of a commit object**

command in shell

```
$ git commit -m "Fix bug in function foo()"
[main (root-commit) 815de0a] Fix bug in function foo()
 4 files changed, 4 insertions(+)
 create mode 100644 LICENSE.txt
 create mode 100644 README.md
 create mode 100644 src/fun.py
 create mode 100644 src/main.py
```

- In our example, the object store has now 7 objects:
  - 4 blobs – one for each file tracked in the repo.
  - 2 trees – src/ and the root of the working dir.
  - 1 commit.



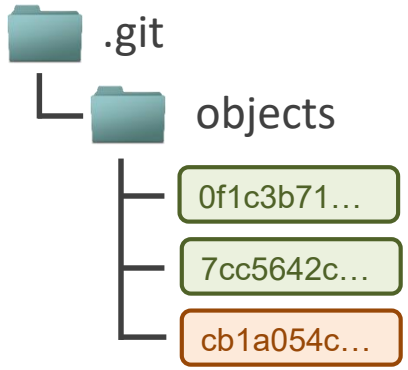
Supplementary material...

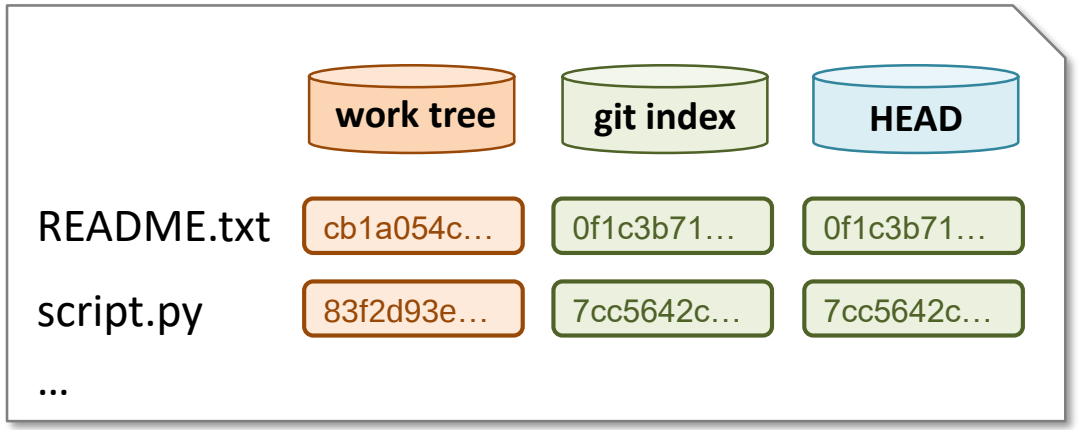
The Git index

- The Git index is a binary file located in [`.git/index`].
- The index has no copies of the data, it's only a table linking file names with blobs.

`git add README.txt`

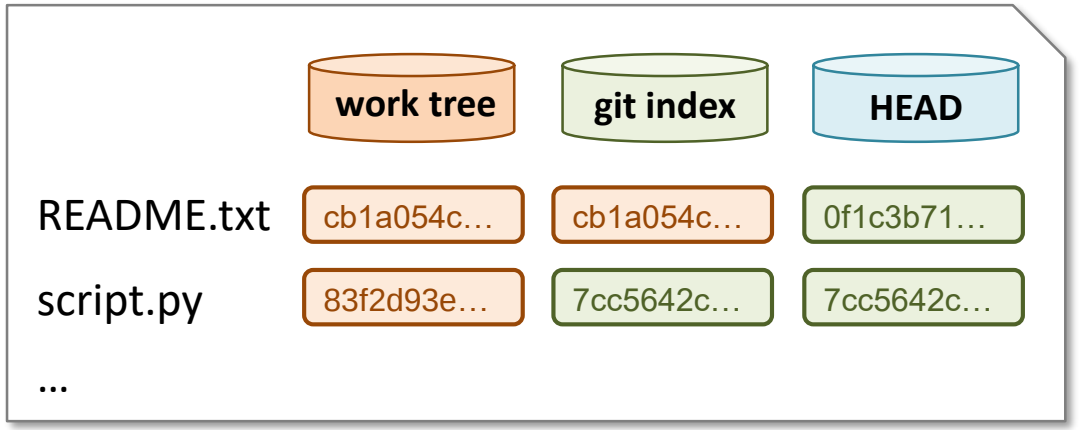
When a file is added/updated to the index, its content gets stored as blob in the object store.

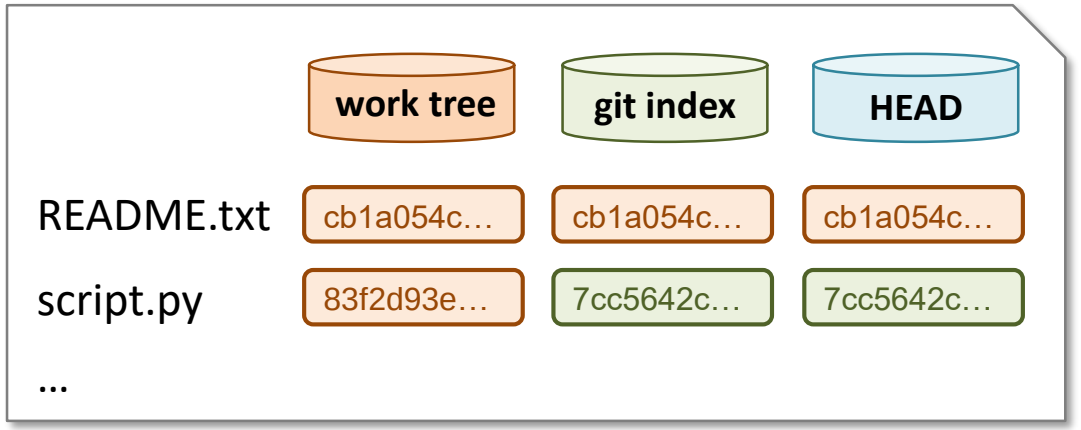




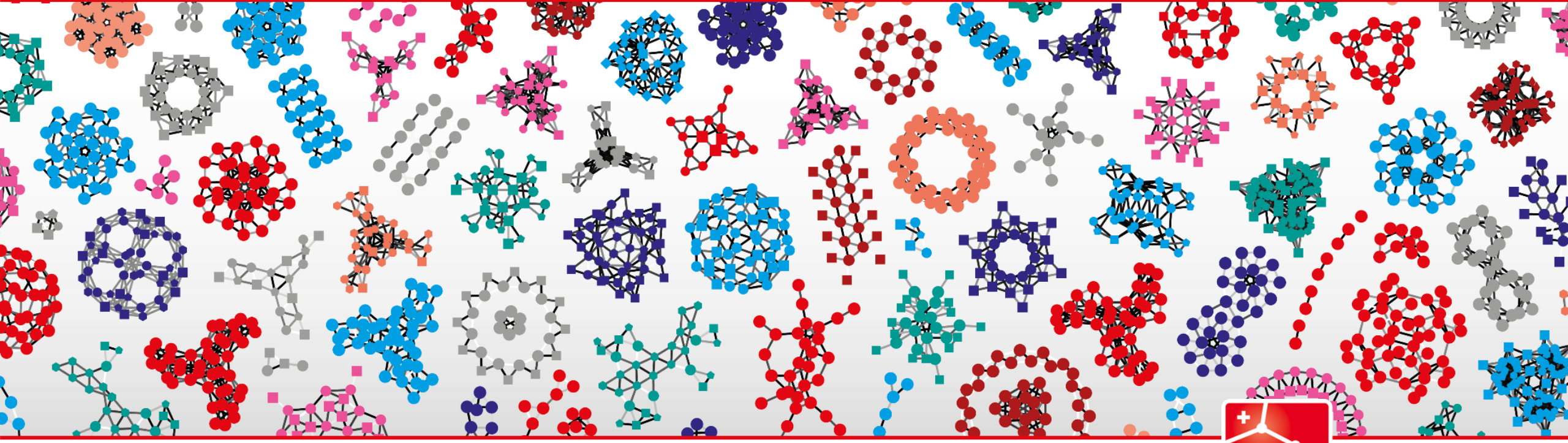
`git commit`

computes the SHA-1 hash for the top tree of the index, and uses it in computing hash of commit.





Supplementary material



Thank you for attending this course

Swiss Institute of Bioinformatics