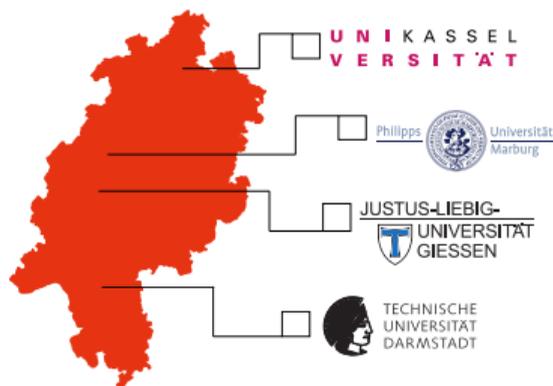


# Version control with GIT

Hessisches Kompetenzzentrum für Hochleistungsrechnen (HKHLR)

HKHLR

2024/11/05



HKHLR is funded by the Hessian Ministry of Sciences and Arts





- 1 What is version control?
- 2 Key Concepts
- 3 First Steps in GIT
- 4 GIT Workflow
- 5 Branching
- 6 Extras



- ▶ A system that keeps records of your changes
- ▶ Allows for collaborative development
- ▶ Able to see **who** made changes and **when**
- ▶ Can revert any changes back to a previous state



Src.: <https://xkcd.com/1597/>

## Why version control?

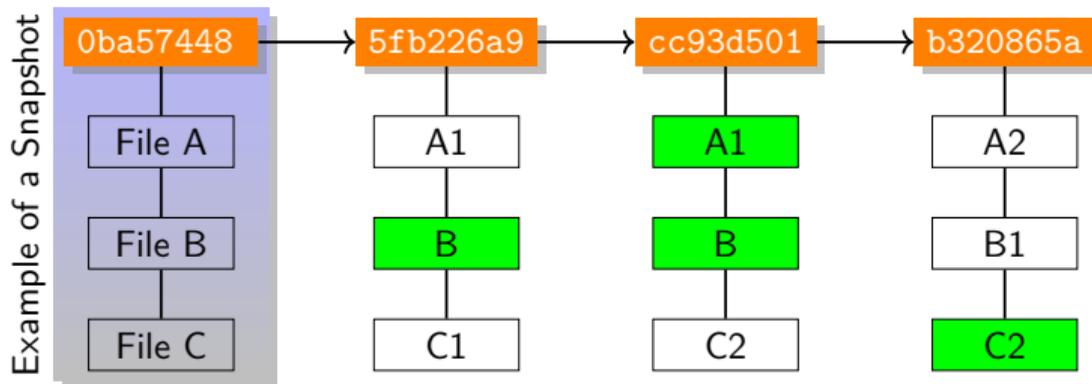
Familiar example:



Src: <http://phdcomics.com/comics/archive.php?comid=1531>

## Snapshots

- ▶ With Git we make a “picture” (snapshot) of current state of all files
- ▶ Git only stores files that actually have changed
- ▶ History of commits is treated as “stream of snapshots”



- ▶ By taking snapshots Git keeps track of our code history
- ▶ Git records what all your files look like at a given point in time
- ▶ We decide **when** to take a snapshot and of **what** files
- ▶ We have the ability to go back to visit any snapshot

- ▶ Making a “commit“ is the act of creating a snapshot
- ▶ Essentially, a project is made up of a bunch of commits
- ▶ Commits contain three pieces of information:
  - 1 Information about how the files changed from previously
  - 2 A reference to the commit that come before it (called the parent commit)
  - 3 A hash code name (Will look like: edfec504eb864dc557f3f5b9d3d301617036d15f3a)

Commits as small as possible or as big as necessary

- ▶ A collection of all the files and the history of those files
  - ▶ Consists of all your commits
  - ▶ Place where all your hard work is stored
- ▶ Only put source files into version control, never generated files.
  - ▶ A source file is any file you create, usually by typing in an editor.
  - ▶ A generated file is something that the computer creates, usually by processing a source file.
- ▶ Never put confidential information into a public repository, e.g.:
  - ▶ Passwords
  - ▶ API keys

Configure your GIT information:

### shell

```
1 $ git config --global user.name "YOUR NAME"  
2 $ git config --global user.email "your.name@domain.de"
```

Inspect Configuration:

### shell

```
1 $ git config --list
```

## shell

```
1 $ mkdir myrepo
2 $ cd myrepo
3 $ git init
```

- ▶ directory will become the working tree for the repository
- ▶ Initialized empty Git repository in `../myrepo/.git`

## shell

```
1 $ mkdir myrepo
2 $ cd myrepo
3 $ git init
```

- ▶ directory will be become the working tree for the repository
- ▶ Initialized empty Git repository in `../myrepo/.git`

- ▶ repository is created without a working tree and it is used as a remote repository that is sharing a repository among teammates

## shell

```
1 $ git init --bare
```

For shared repositories pay attention to the file permissions.  
It is recommended to prohibit changing the history.

Git defines three main stages that a file can be in:

**Committed** File is safely stored in the local repository

**Modified** The file has undergone changes but yet has to be committed to the local repository

**Staged** A modified file that is marked in its current state to go into the next commit

**untracked** File is not tracked by git

## shell

```
1 $ git status
2
3 On branch master
4
5 Initial commit
6
7 nothing to commit (create/copy files and use "git add" to track)
```

## shell

```
1 $ git status
2
3 On branch master
4
5 Initial commit
6
7 nothing to commit (create/copy files and use "git add" to track)
```

Add a file

## shell

```
1 $ echo "Hello World" > doc.md
```



## shell

```
1 $ git status
2
3 On branch master
4
5 Initial commit
6
7 Untracked files:
8   (use "git add <file>..." to include in what will be committed)
9
10  doc.md
11
12 nothing added to commit but untracked files present (use "git add" to \
   track)
```

## shell

```
1 $ git add doc.md
2
3 $ git status
4
5 On branch master
6
7 Initial commit
8
9 Changes to be committed:
10   (use "git rm --cached <file>..." to unstage)
11
12   new file:   doc.md
```

## shell

```
1 $ git add doc.md
2 $ git add *.md
```

create as small as possible, logically separated commits

short version:

## shell

```
1 $ git add -p
2 $ git add -i
```

long version:

## shell

```
1 $ git add --patch
2 $ git add --interactive
```

## shell

```
1 $ git commit -m "creating doc.md"
2
3 [master (root-commit) 25b09b9] creating doc.md
4 1 file changed, 3 insertions(+)
5 create mode 100755 doc.md
6
7 $ git status
8
9 On branch master
10 nothing to commit, working directory clean
```

A commit should contain a single, self-contained idea.

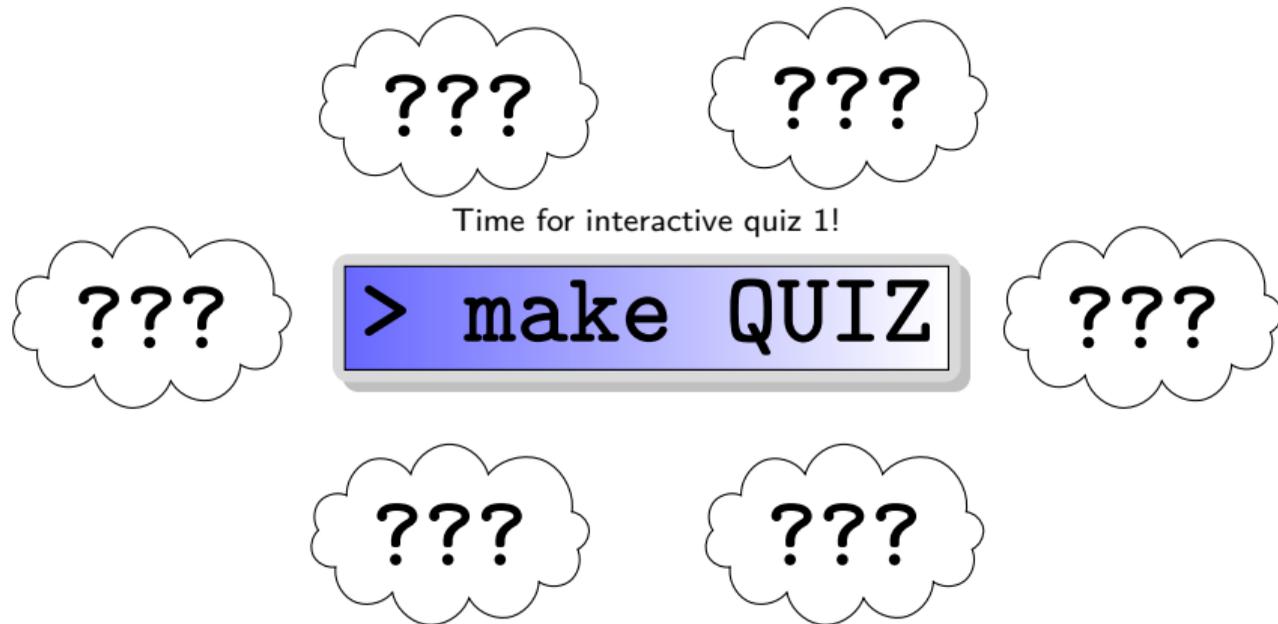
### shell

```
1 $ git commit -m "My first commit"
2
3 [master 8345967] changed
4 1 files changed, 1 insertions(+), 1 deletions (-)
```

edit last commit-message

### shell

```
1 $ git commit --amend
```



What is the first thing to do after installing git?

- A: Define the contents of working directory, staging area, and repository with `git define`.
- B: Configure mail address, name, and preferred editor with `git config`.
- C: Initialize an empty repository or clone an existing one with `git init` OR `git clone`.
- D: Choose a hashing algorithm with `git hash`.

What is the first thing to do after installing git?

- A: Define the contents of working directory, staging area, and repository with `git define`.
- B: **Configure mail address, name, and preferred editor with `git config`.**
- C: Initialize an empty repository or clone an existing one with `git init` OR `git clone`.
- D: Choose a hashing algorithm with `git hash`.

A Git Repository is essentially made up a bunch of commits. What is a commit?

- A: A commit contains the author who has changed your files.
- B: A commit is done by git automatically once in a while.
- C: Each file has its own commit.
- D: A commit is a snapshot of the current state of your project.

A Git Repository is essentially made up a bunch of commits. What is a commit?

- A: A commit contains the author who has changed your files.
- B: A commit is done by git automatically once in a while.
- C: Each file has its own commit.
- D: **A commit is a snapshot of the current state of your project.**

Where can you find further help about git?

- A: `git help <command like commit or add>`
- B: `man git`
- C: `git -help`
- D: Online e.g. <https://git-scm.com/docs>.

Where can you find further help about git?

- A: **git help <command like commit or add>**
- B: **man git**
- C: **git -help**
- D: **Online e.g. <https://git-scm.com/docs>.**

What does the command 'git add' do?

- A: It marks changes to a file to go into the next commit.
- B: It adds a line to a file.
- C: It adds a new file to the repository.
- D: It adds a file to the remote repository.

What does the command 'git add' do?

- A: **It marks changes to a file to go into the next commit.**
- B: It adds a line to a file.
- C: It adds a new file to the repository.
- D: It adds a file to the remote repository.

What are the git states a file can be in?

- A: committed (and unmodified)
- B: changed (and not staged)
- C: untracked
- D: staged

What are the git states a file can be in?

- A: **committed (and unmodified)**
- B: **changed (and not staged)**
- C: **untracked**
- D: **staged**

Why is git useful?

- A: Because it allows you to collaborate with others.
- B: Because it keeps track on who did which changes.
- C: Because it does not allow for code-breaking changes.
- D: Because it allows you to revisit any previous version.

Why is git useful?

- A: **Because it allows you to collaborate with others.**
- B: **Because it keeps track on who did which changes.**
- C: Because it does not allow for code-breaking changes.
- D: **Because it allows you to revisit any previous version.**

Which statements are true for public repositories?

- A: It is always up to date with your local copy.
- B: You can even have multiple public remotes.
- C: Never put confidential information like passwords in a public repo.
- D: It can be used to collaborate with others.

Which statements are true for public repositories?

- A: It is always up to date with your local copy.
- B: **You can even have multiple public remotes.**
- C: **Never put confidential information like passwords in a public repo.**
- D: **It can be used to collaborate with others.**

▶ Questions?

Only four major Git commands which actually talk to remote repos and do any communication:

- ▶ clone
- ▶ fetch
- ▶ pull
- ▶ push

If you have the address of a known repository and you want to create a local copy:

### shell

```
1 $ git clone git@host:/path/to/repository/testing.git mydir
2
3 Cloning into 'mydir'
4 remote: Counting objects: 3, done.
5 remote: Total 3 (delta 0), reused 0 (delta 0)
6 Receiving objects: 100% (3/3), done.
```

Note: "Remote" repo may also be a local (group) directory initialized with `git init --bare`

- ▶ Every branch that exists on the remote when you cloned the repo will have a branch in `remotes/origin`
- ▶ `git fetch` does is update all of the `remotes/origin` branches.
- ▶ `git fetch` will modify only the branches stored in `remotes/origin` and *not* any of your local branches

## Idea

Get the remote changes without chainging something locally

- ▶ `git pull` is the combination of two other commands:  
**First**, perform a `git fetch` to update the `remotes/origin` branches. **Second**, if the branch you are on is tracking a remote branch, then it does a `git merge` of the corresponding `remote/origin` branch to your branch.

## shell

```
1 $ git pull
2 remote: Counting objects: 7, done.
3 remote: Compressing objects: 100% (4/4), done.
4 remote: Total 4 (delta 2), reused 0 (delta 0)
5 Updating 361303d..f2cd831
6 Fast forward
7   doc.md | 1 +
8   1 files changed, 1 insertions (+), 0 deletions (-)
```

git push makes your new commits available on the remote server

## shell

```
1 $ git push
2   Counting objects: 5, done.
3   Writing objects: 100% (3/3), 272 bytes, done.
4   Total 3 (delta 0), reused 0 (delta 0)
5   To git@host:/path/to/repository/testing.git
6   edfec50..2fc284e  master --> master
```

## shell

```
1 $ git push [remote-name] [remote-branch-name]
```

## shell

```
1 $ git remote add origin <server>
2 $ git remote add origin git@host:/path/to/repository/testing.git
3 $ git push origin master
4
5 Counting objects: 3, done.
6 Writing objects: 100% (3/3), 231 bytes, done.
7 Total 3 (delta 0), reused 0 (delta 0)
8 To git@host:/path/to/repository/testing.git
9 [new branch] master --> master
```

## shell

```
1 $ git log
2   commit edfec504eb864dc557f3f5b9d3d301617036d15f3a
3   Author: Nikolas Luke <niko.luke@uni-kassel.de>
4   Date:   Thu Oct 18 14:00:20 2018 +0200
5
6   My First Commit
```

## search in history

## shell

```
1 $ git log --pretty=short --since=2weeks
2 $ git log --pretty=short --author="Nikolas Luke" --grep="comment"
```

## shell

```
1 $ git checkout -- <filename>
2 $ git checkout -- doc.md
```

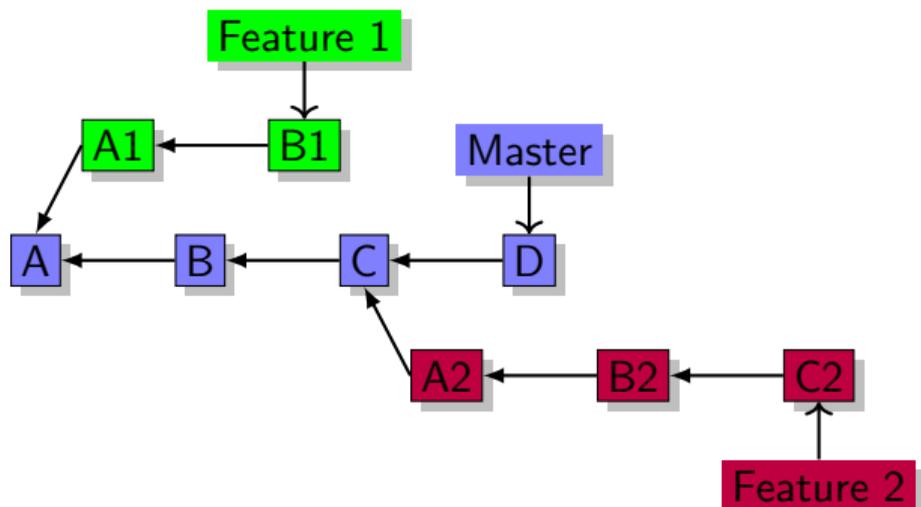
Go back to a certain snapshot:

## shell

```
1 $ git checkout <commit>
2 $ git checkout edfec504eb864dc557f3f5b9d3d301617036d15f3a
```

**Be Aware that git checkout will overwrite the current status of your work with the old state!**

- ▶ A branch represents an independent line of development
- ▶ Branches are an integral part of software development with Git
- ▶ Branches are the basis for team projects and an efficient software development process
- ▶ Standard branch in Git: `master` branch

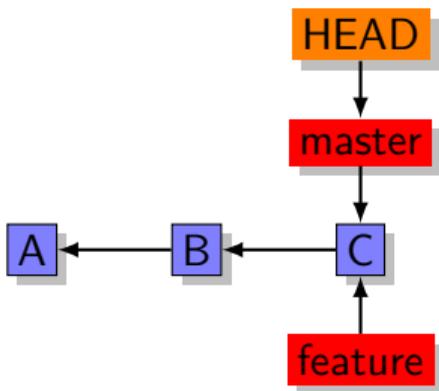


creating a new branch creates a new pointer referring to commits in a line of development.

shell

```
1 $ git branch feature
```

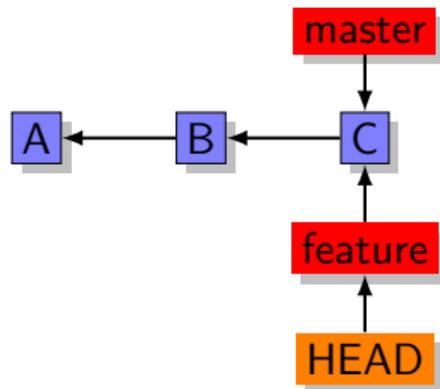
⇒ **Not yet on new branch!**



shell

```
1 $ git checkout feature
```

⇒ **Now we switched branches!**



creating a new branch creates a new pointer referring to commits in a line of development.

shell

```
1 $ git branch feature
```

shell

```
1 $ git checkout feature
```

All-in-one command:

shell

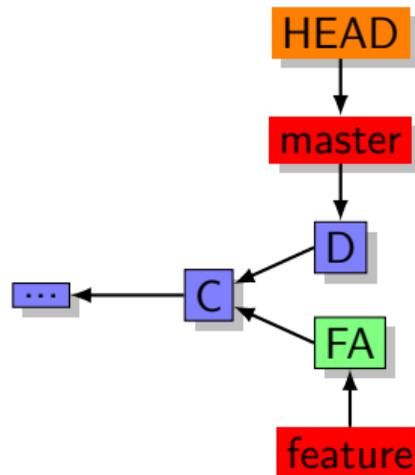
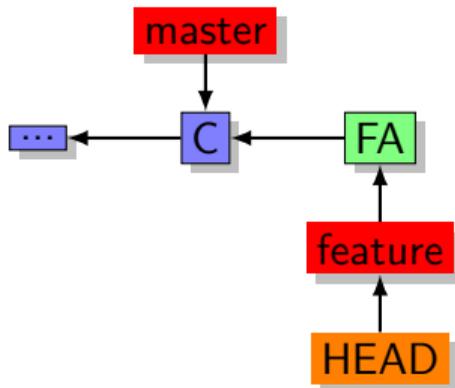
```
1 $ git checkout -b feature # execute on master branch
```

## shell

```
1 $ git checkout -b feature
2 $ vim <file> # do some work
3 $ git add <file>
4 $ git commit -m <message>
```

## shell

```
1 $ git checkout master
2 $ vim <otherfile>
3 $ git add <otherfile>
4 $ git commit -m <message>
```



Create a branch and switch to it:

### shell

```
1 $ git branch feature && git checkout feature
2 $ git checkout -b feature
```

List branches

### shell

```
1 $ git branch # local branches in your repository
2 $ git branch -r # remote branches
3 $ git branch -a # all local and remote branches
```

Rename (move) a branch: old-feature → new-feature

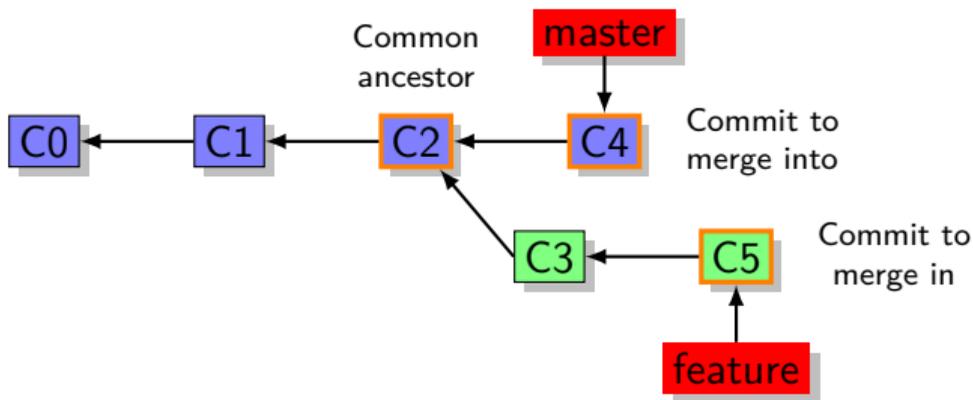
shell

```
1 $ git branch -m old-feature new-feature
```

Delete a branch:

shell

```
1 $ git branch -d new-feature
```

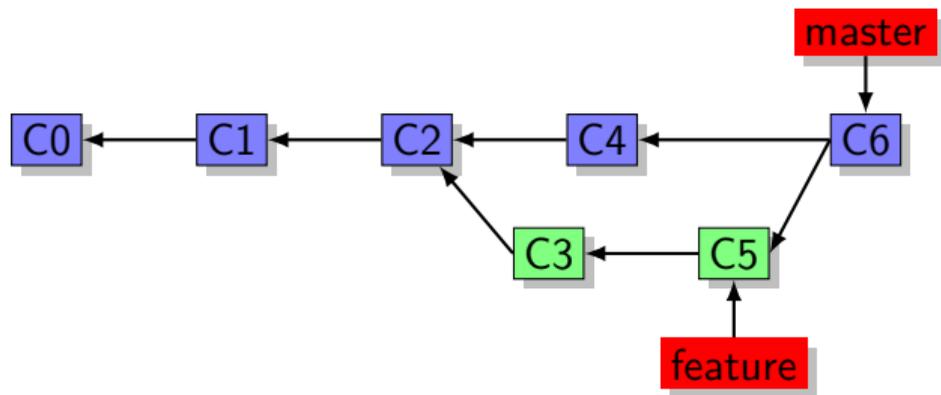


Merge feature into master:

### shell

```
1 $ git checkout master # switch to master branch
2 $ git merge feature # merge feature into master
3 $ git branch -d feature # delete feature branch (OPTIONAL!)
```

This is the situation after the merge (feature branch not yet deleted).

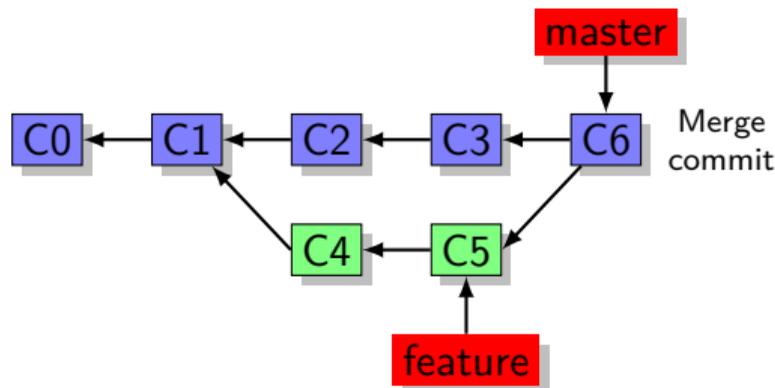
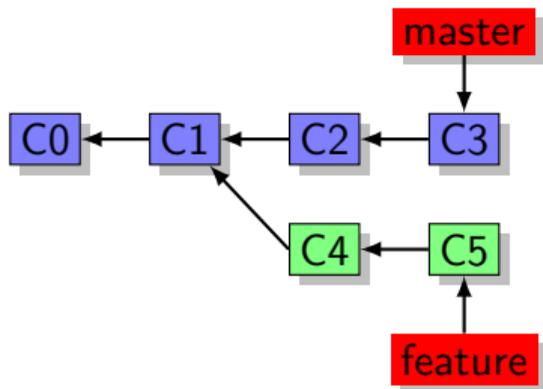


C6 is called the “merge commit”.

Branches diverged. Commits were made in *both* branches after checking out feature. Integrate branches by merging feature into master.

## shell

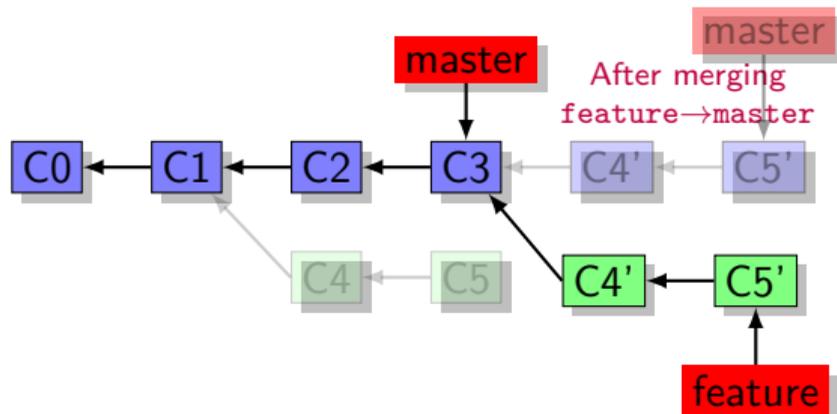
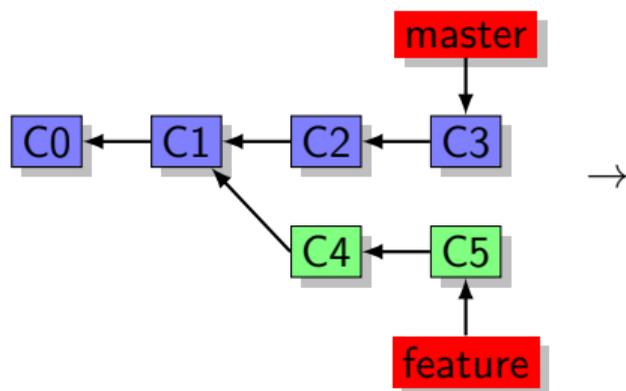
```
1 $ git checkout master
2 $ git merge feature
```



Branches diverged. Commits were made in *both* branches after checking out feature. Replay the changes made in master onto commits in feature.

## shell

```
1 $ git checkout feature
2 $ git rebase master
3 $ git checkout master
4 $ git merge feature # fast-forward merge
```



⇒ Like we had checked out after C3!

⇒ C4', C5' are new commits!

- ▶ The feature branch is moved to the tip of the master branch (i.e., all the changes in feature are effectively integrated into master); new commits are created in the feature branch
- ▶ After merging back to master the logs look linear: Seems like all the work happened in series (although it happened in parallel) since there is no merge commit (fast-forward merge)
- ▶ We change history and lose traceability; we replace existing commits with others that are similar but different

## Golden Rule of Rebasing

Do not rebase commits that exist outside your repository and people may have based work on them (meaning that you should never rebase a shared/public branch)!

The `git rebase` command can also be used in an interactive mode. It may for example be used to<sup>1</sup>

- ▶ Fix older commits (older than just the previous one)
- ▶ Squash several commits into one single commit

Example fixing the last 5 commits:

shell

```
1 $ git rebase --interactive HEAD~5
```

<sup>1</sup>A very comprehensive account of changing the commit history with rebasing can be found on <https://git-rebase.io/>

- ▶ Handling a git pull request with merge conflict
- ▶ When working with git, the relatively complex tasks are issuing a pull request & then merging with conflicts

Step 1 Verify your local repository

```
git checkout pinc
```

```
git pull origin pinc
```

Ensure that the files on local repository are in-sync with your remote git repository

Step 2 Switch to branch

```
git checkout feature-1
```

```
git pull origin feature-1
```

Switch to the branch that you want to merge

Ensure that you pull the latest files from your remote server

Step 3 Try to merge

```
git merge pinc
```

Step 4 Resolve the merge conflict If you get the message, that there is a merge conflict & it cannot automatically merge the change, you can resolve the conflict manually. Open the file & you'll need to fix this.

Step 5 Check in changes

Commit the fixes to the branch

```
git add file.py
```

```
git commit -m "some comment"
```

```
git push origin feature-1
```

Step 6 Merge the branch

## shell

```
1 $ cat .gitignore
```

e.g. L<sup>A</sup>T<sub>E</sub>X-generated files:

## shell

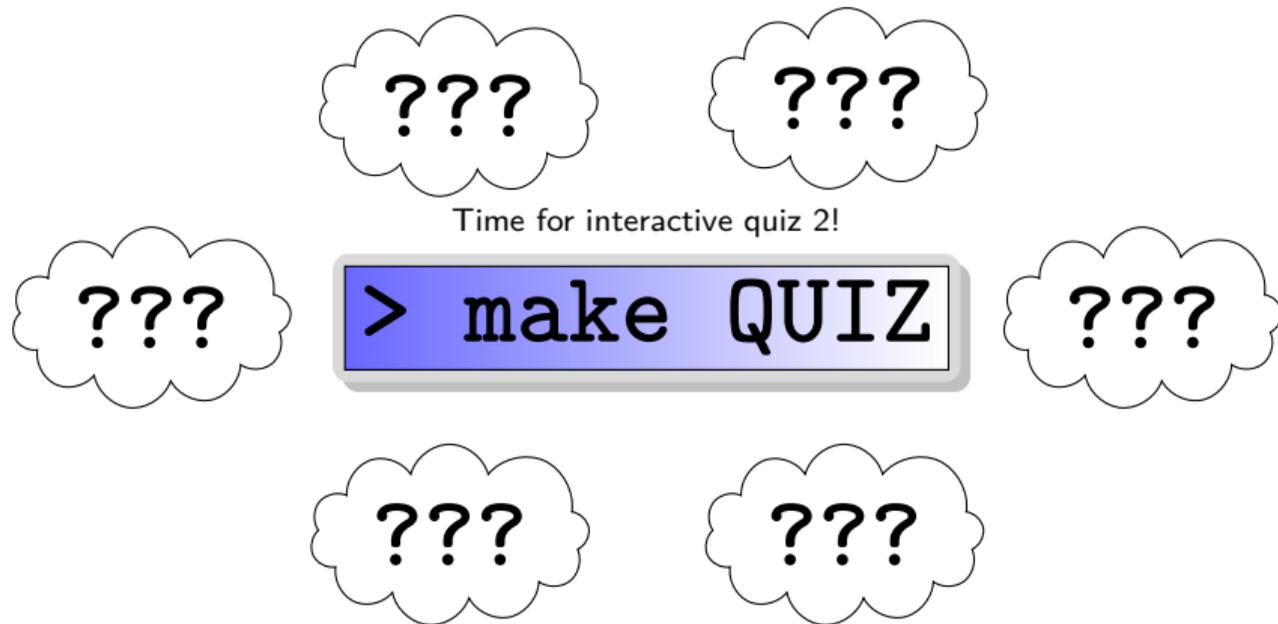
```
1 *.aux  
2 ...  
3 *.vrb
```

global .gitignore

## shell

```
1 $ git config --global core.excludesfile ~/.gitignore
```

- 1 `git status` – Make sure your current area is clean.
- 2 `git pull` – Get the latest version from the remote. This saves merging issues later.
- 3 Edit your files and make your changes.
- 4 `git status` – Find all files that are changed. Make sure to watch untracked files too!
- 5 `git add [files]` – Add the changed files to the staging area.
- 6 `git commit -m "message"` – Make your new commit.
- 7 `git push origin [branch-name]` – Push your changes up to the remote.



What is the difference between `git fetch` and `git pull`?

- A: `git pull` downloads the current state from a remote repo while `git fetch` also applies this state to your local copy.
- B: There is no difference, they are just aliases for the same command.
- C: `git fetch` downloads the current state from a remote repo while `git pull` also applies this state to your local copy.

What is the difference between `git fetch` and `git pull`?

- A: `git pull` downloads the current state from a remote repo while `git fetch` also applies this state to your local copy.
- B: There is no difference, they are just aliases for the same command.
- C: `git fetch` **downloads the current state from a remote repo while** `git pull` **also applies this state to your local copy.**

How can you recover a past state of your project?

- A: Using the `git history` command.
- B: Using the `git commit` command.
- C: Using the `git log` command.
- D: Using the `git checkout` command.

How can you recover a past state of your project?

- A: Using the `git history` command.
- B: Using the `git commit` command.
- C: Using the `git log` command.
- D: **Using the `git checkout` command.**

What is the purpose of git branches?

- A: You can track different developments (e.g. different features).
- B: Branching avoids merge conflicts.
- C: It is mandatory to use branches in git.
- D: You can use different branches to organize collaborative work.

What is the purpose of git branches?

- A: **You can track different developments (e.g. different features).**
- B: Branching avoids merge conflicts.
- C: **It is mandatory to use branches in git.**
- D: **You can use different branches to organize collaborative work.**

Which command creates a new branch (called branchname)?

- A: `git checkout -b branchname`
- B: `git branch branchname`
- C: `git branch -b branchname`
- D: `git checkout branchname`

Which command creates a new branch (called branchname)?

- A: **git checkout -b branchname**
- B: **git branch branchname**
- C: git branch -b branchname
- D: git checkout branchname

Which command can we issue to rename a branch "A" to "B"?

A: `git branch -m "A" "B"`

B: `git branch "A" "B"`

C: `git checkout -b "A" "B"`

D: `git branch -d "A" "B"`

Which command can we issue to rename a branch "A" to "B"?

- A: **git branch -m "A" "B"**
- B: git branch "A" "B"
- C: git checkout -b "A" "B"
- D: git branch -d "A" "B"

What is true for a merge commit?

- A: A recursive merge happens when branches have not diverged.
- B: A merge commit is created automatically despite a merge conflict having occurred.
- C: A merge commit always integrates changes from different branches.
- D: Only fast forward merges are possible.

What is true for a merge commit?

- A: A recursive merge happens when branches have not diverged.
- B: A merge commit is created automatically despite a merge conflict having occurred.
- C: **A merge commit always integrates changes from different branches.**
- D: Only fast forward merges are possible.

When should you avoid rebasing?

- A: When git tells you that it is not possible.
- B: When a merge without conflicts is possible.
- C: When working with multiple remotes.
- D: When someone else's work is based on yours.

When should you avoid rebasing?

- A: **When git tells you that it is not possible.**
- B: When a merge without conflicts is possible.
- C: When working with multiple remotes.
- D: **When someone else's work is based on yours.**

What is true about workflows?

- A: The best workflow combines a development branch with a release branch and feature branches.
- B: Whether you use version control systems or perform everything manually makes little difference.
- C: Each team may have their own workflow and that is a matter of culture.

What is true about workflows?

- A: The best workflow combines a development branch with a release branch and feature branches.
- B: Whether you use version control systems or perform everything manually makes little difference.
- C: **Each team may have their own workflow and that is a matter of culture.**

What is the purpose of the .gitignore file?

- A: You can include ignored files by using the -f option.
- B: It tells git which commits to ignore.
- C: It tells git which files to ignore.
- D: You can include ignored commits by using the -f option.

What is the purpose of the .gitignore file?

- A: **You can include ignored files by using the -f option.**
- B: It tells git which commits to ignore.
- C: **It tells git which files to ignore.**
- D: You can include ignored commits by using the -f option.

- ▶ General topics: <https://git-scm.com/book/en/v2/> and <https://www.atlassian.com/git/tutorials>
- ▶ Advanced merging:  
<https://git-scm.com/book/en/v2/Git-Tools-Advanced-Merging>
- ▶ Merge conflicts:  
<https://de.atlassian.com/git/tutorials/using-branches/merge-conflicts>
- ▶ Distributed workflow:  
<https://git-scm.com/book/en/v2/Distributed-Git-Distributed-Workflows>