gist / jist/ *n* [AF, it lies, fr. *gisir* to lie, ultim. fr. L *jacēre* — more at ADJACENT] (ca. 1711) **1** : the ground of a legal action **2** : the main point or part : ESSENCE <the ∼ of an argument>

[1]**git** /'git/ *n* [var. of *get*, term of abuse, fr. [2]*get*] (1929) *Brit* : a foolish or worthless person

[2]**git** *dial* var of GET

git-go var of GET-GO

git·tern /'gi-tərn/ *n* [ME *giterne*, fr. MF *guiterne*, modif. of OSp *guitarra* guitar] (14c) : a medieval guitar

git essentials

Luca Formaggia

2024

Merriam-Webster's Collegiate Dictionary, p.529

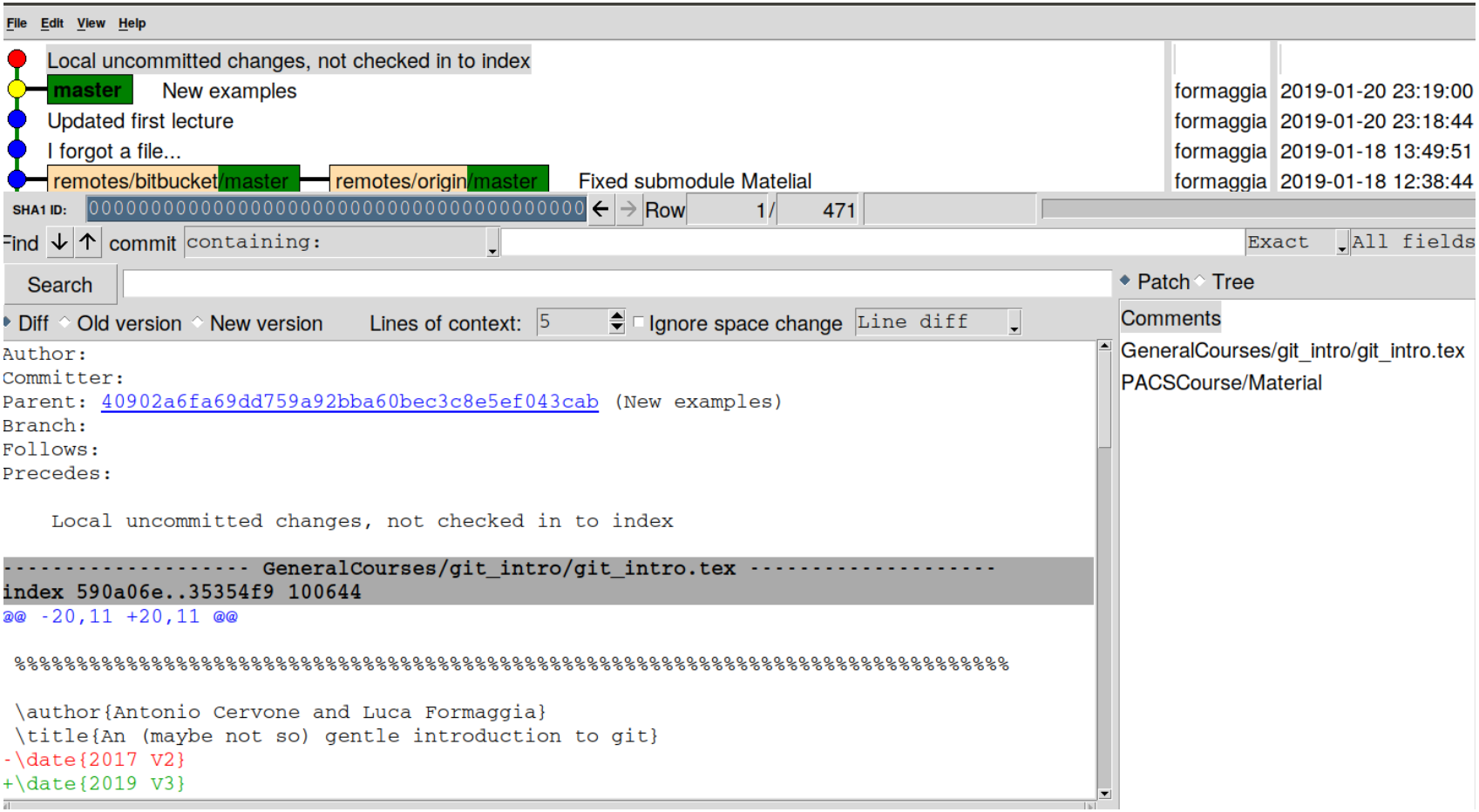
A distributed VCS: git

Git is a *distributed* version control system (VCS) with an emphasis on speed. It was initially designed and developed by Linus Torvalds for Linux kernel development. Is now used by thousands of software developers worldwide and has in fact replaced previous VCSs.

Every Git working directory is a full-fledged repository with complete history and full revision tracking capabilities, not dependent on network access or a central server.

In git (almost) every command operates just on the local repository (the one you have on your PC). You do not even need a remote repository to use git!

A snapshot of a repo (using gitk)



Free remote repositories

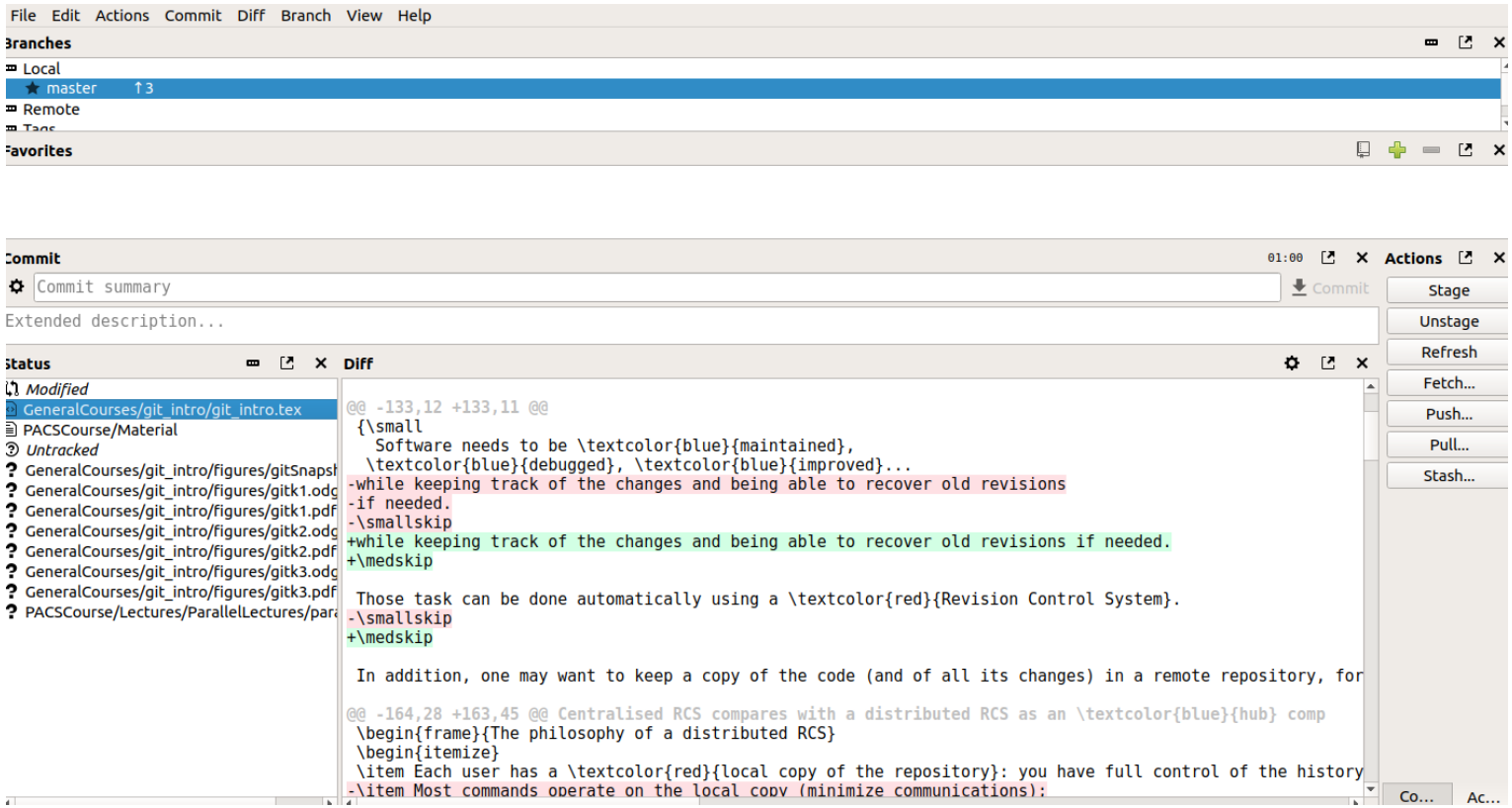
There are many websites providing a nice place for a `git` repository and the possibility of sharing it with others, such as

- GitHub
- Bitbucket
- GitLab
- ...

Usually, with an educational/professional account, you can unlock extra features, such as unlimited private repositories ....

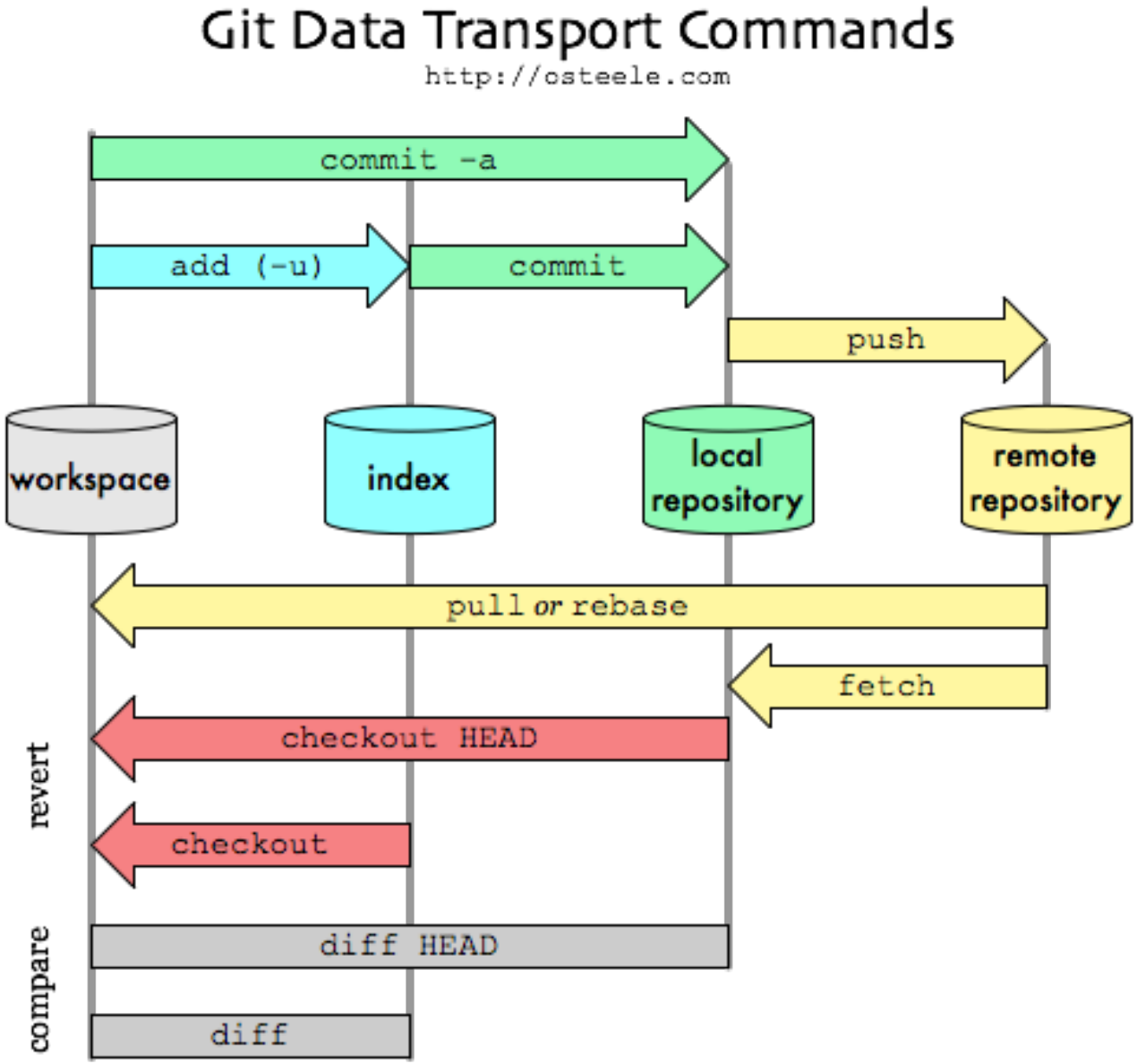
Graphical Interfaces

There are plenty of graphical interfaces for git. I sometimes use git-cola, but you may find many others on the web. A list may be found here.



Many IDEs integrate git in their workflow. And *Overleaf*, a site for shared Latex editing, may interface with GitHub.

A visual overview of a typical git workflow



A few important terms

- Workspace. The folders containing all files of your code. The root folder contains a hidden directory called `.git` with the local repo. Don't touch it if you do not know what you are doing.. Files in the workspace may be tracked by git of untracked, or ignored.
- Commit. A snapshot of the situation of the tracked files in your workspace at a given time. Commits are uniquely identified by a SHA key.
- Index. Contains the list of files ready (staged) to be committed. Typically new files or modified file. To put a file in the index you use the `add command`.
- Checkout. The operations that extract a single file or an entire commit form the repo.
- HEAD. A "pointer" to the last checked out commit.

Commands that communicate with a remote repository

You do not need a remote repository to use git, you git repository may be entirely local!. You use a remote repository either for safety (you have an extra backup) and/or to collaborate with others.

In fact, most git commands operate on your local repository.

Here you have the main three commands used by git to interact with a remote repository:

push, fetch, pull

First thing to do

If you have never used git on a computer you should first let git know who you are (so people can blame you for your wrongdoings or cherish you for your programming skill!)

```
$ git config --global user.name "Micky Mouse"
```

```
$ git config --global user.email "micky.mouse@gmail.com"
```

`git config` tells git to change some git configuration, the option `global` makes it to change parameters that applies to all git repos in your computer. Another thing you may want to change is the default editor

```
$ git config --global core.editor "gedit"
```

Need help?

Git has an integrated help. If you need help for the command `command` just do

```
$ git help command
```

Git uses a lot of terms that may confuse you at the beginning (and not only at the beginning, . . .). A useful command is

```
$ git help glossary
```

You find more on the Git book.

More help?

Other useful commands:

```
$ git help tutorial
```

A tutorial

```
$ git help tutorial-2
```

The second part of the tutorial.
Git is a very complete (and complex) version control system, don't be scared: in fact you will normally need just a few commands.

Useful references

On the web you can find plenty of resources on `git`!

- Official documentation: http://git-scm.com/docs
- Official "Pro Git" book: http://git-scm.com/book/
- Visual cheat sheet: https://ndpsoftware.com/git-cheatsheet.html
- Bitbucket tutorials: https://www.atlassian.com/git/tutorials
- GitHub tutorials: https://training.github.com/, https://guides.github.com/
- Useful Git patterns for real life: https://happygitwithr.com/workflows-intro.html
- "Oh Sh*t, Git!?": https://ohshitgit.com/ (without swears: https://dangitgit.com/), with the fantastic zine.

What should git keep track of?

You will put in the git repo only the main files (sources, headers...), *i.e.* files that are not created from other files.

You should not put in the repo object files, executables, temporary files etc. etc., *i.e.* files that are in fact the product of a process (compilation, execution etc.).

You will see that there is a way to tell git to completely ignore certain files in your working area and reduce the risk of committing them by mistakes.

Creating a new git repo

There are twoo possible ways to create a git repo

1. You create the repository locally on your computer, and then, if you want, you make a remote copy, for instance on GitHub.
2. You create the repository on a remote site like GitHub, and then you clone it on your computer.

Let's see in detail:

Create/clone a repo

Create a brand new local repository. In the directory where you want to create it type:

```
$ git init
```

Often you want to clone from an existing remote repository (on GitHub in this example).

With HTTPS protocol (simpler but more limited):

```
$ git clone https://github.com/username/mycode.git
```

With SSH protocol (you must have your public SSH key associated to your GitHub account):

```
$ git clone username@github.com:/mycode.git
```

Create a new file

Edit a new file called `file1.cpp`
Show status of the repo:

```
$ git status
```

The output is:

```
On branch master
Your branch is up to date with 'origin/master'.
Untracked files:
(use "git add <file>..." to include in what will be committed)
    file1.cpp
nothing added to commit but untracked files present
(use "git add" to track)
```

Add a new file to the git index

The file `file1.cpp` is at the moment still unknown to git (it's a new file!). It is in the so called untracked state.
We need to tell git that we want to take track of it.

```
$ git add file1.cpp
```

show again status of the repo

```
Your branch is up to date with 'origin/master'.
Changes to be committed:
(use "git restore --staged <file>..." to unstage)
new file:  file1.cpp
```

The file is ready to be committed in your (local) git repository!

The first commit

Now the file `file.cpp` is in the staging area (index), ready to be committed in the repository. May be we may want to add other files to the commit, or maybe we just want to add just that file to the repository. In the latter case we just do

```
$ git commit
```

An editor will appear asking for a commit message. The message is compulsory.
We can commit into the repository a newly created file without adding it beforehand, by using

```
$ git commit <file>
```

You can commit by giving a short message directly:

```
$ git commit -m "This is my first file"
```

The status after the commit

```
$ git commit -m "This is my first file"
```

```
[master aba508f] This is my first file
1 file changed, 5 insertions(+)
create mode 100644 file1.cpp
```

And if you do `git status`

```
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
(use "git push" to publish your local commits)
nothing to commit, working tree clean
```

What does it mean?

The first message says that in the branch master (the only one we have at the moment) we have created a commit.

A commit a snapshot of your workspace at the moment of the commit, and is identified by an hash key, a long hexadecimal number whose first significant digits are shown (`aba508f`), and are normally enough to identify the commit uniquely.
You can at any time retrieve the situation of your files (those under git control) at a given commit using its hash key.

The message also says that in that commit you have added a file (the mode indicates just the file permission), which consists of 5 lines.

To add all modified file and commit in just one command:

```
$ git commit -a -m "A nice message"
```

What to put in a full commit message

If you do not use the `-m` option an editor is launched. You can put

```
A first line with a brief description (it is the one that is
shown by some git commands)

Possibly followed by an empty line and a detailed description
(possibly wrap lines to 72 characters)

Paragraphs are separated by empty lines.

- you may create lists using the - character
```

master or main, or something else

The default branch in git used to be called master. Recently, people objected that it is not a good name since the "master branch" is not mastering anything, it is just the branch you have chosen as your main branch. So, now GitHub names as `main` the default branch (unless you choose otherwise).
You can change how git names the default branch when creating a repo:

```
$ git config -global init.defaultBranch <name>
```

and change the name of a branch

```
$ git git branch -m <name>
```

The remote

If you have a remote repository, how can git know that your local repo (more precisely the master branch of your local repo) is now ahead of the remote? Git has not communicated with GitHub since commit operates only on the local repo. Well, let's try to do

```
$ git branch -a -v
```

You get

```
* master aba508f [ahead 1] This is my first file
remotes/origin/HEAD -> origin/master
remotes/origin/master 5fe0589 Initial commit
```

What does it mean?

The local remote

`git branch -a -v` tells git to show all (`-a`) branches and be a bit verbose (`-v`). The asterisk indicates the current branch (the only one we have so far!), but we have also other stuff.

The branches that begin with `remote/` are local copies of the branch in the remote repository, made at the time of the last `pull`, or `fetch`.

`origin` is the name given by git to the remote repository (you can change it and you can have more than one remote!). It refers to our GitHub repo.

Let's at the moment forget about the `HEAD` stuff. You may see that git is saying: the local copy of the remote repo is commit `5fe0589`, committed with message `Initial commit`.

My `master` branch is at commit `aba508f` ahead of 1 commit with respect to `5fe0589`.

What should I do now?

Either continue working in your working tree, or you may decide that it is time to send you modifications to the remote repository. To do that:

```
$ git push
```

or, if you want to be more precise,

```
$ git push origin master
```

which tells git to push on the remote called `origin` the branch `master`. Since master is the current branch, and `origin` the only remote repo we have, we can use the first, simplified, version of the command.

This command communicates with the remote, *i.e.* with GitHub, so it will not work if you are not connected to internet

Your repo and the remote are now synchronized

The command `push` gives some messages that just tell that your master branch has been sent to the remote, and `git status` now give simply

```
On branch master
Your branch is up to date with 'origin/master'.
nothing to commit, working tree clean
```

and `git branch -a -v`

```
* master aba508f This is my first file
remotes/origin/HEAD -> origin/master
remotes/origin/master aba508f This is my first file
```

All set! Git has also updated the local copy of the remote to match the one actually in GitHub.

Push not possible because remote has been updated by others

It may happen that the push is blocked by git because one of your collaborators has pushed her commit to the remote. You receive a message like `error: failed to push some refs`.

You need to pull the changes from the repo. There are two strategies: merge and rebase. You can set the default behavior with

```
$ git config pull.rebase false
```

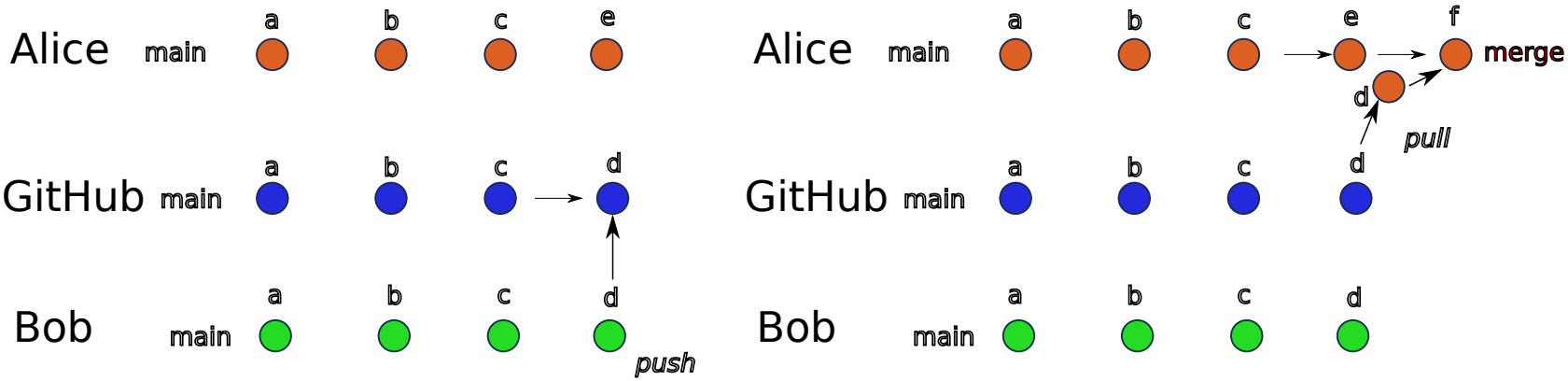
if you want merge to be the default, or

```
$ git config pull.rebase true
```

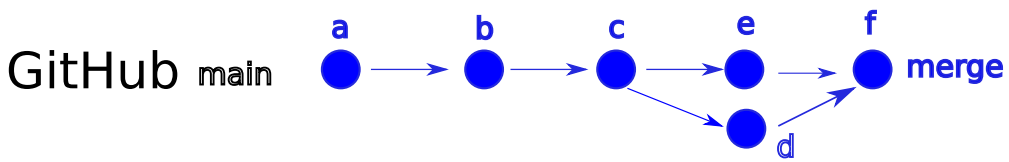
if you prefer rebasing. The difference are illustrated in the next slides.

Merge strategy

With the merge strategy the head commit in the remote, updated by Bob, is fetched and merged with the local head commit of Alice, the result is a commit with two parents: a merge commit

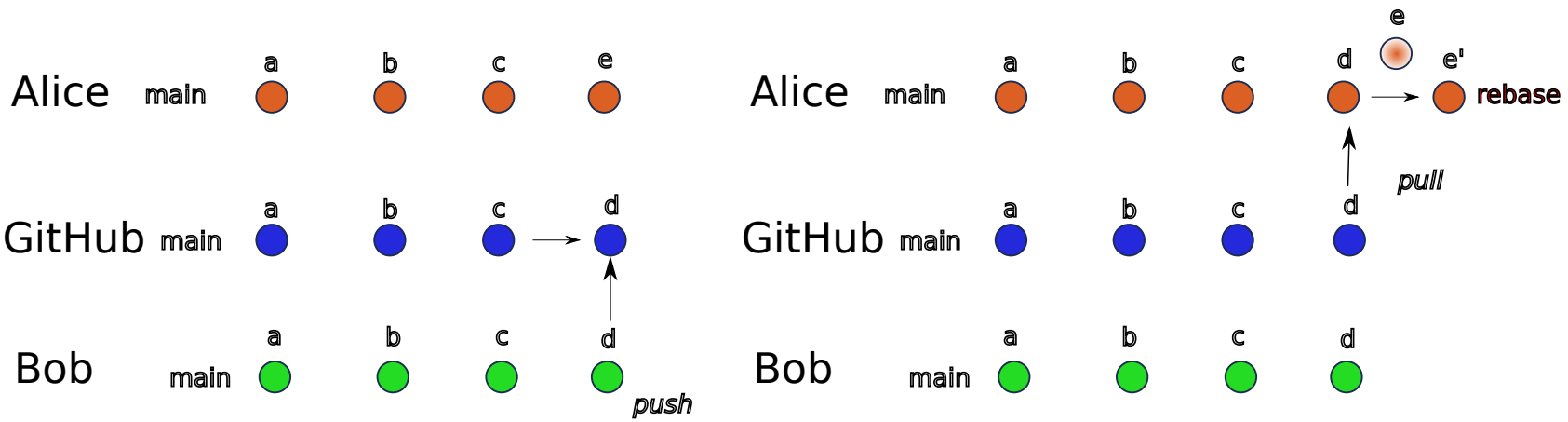


When Alice pushes to the remote the final situation on the remote is

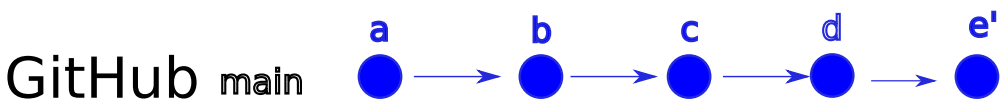


Rebase strategy

With the merge strategy the commits created by Alice are "payed back" on top of the head commit in the remote, leaving a linear history



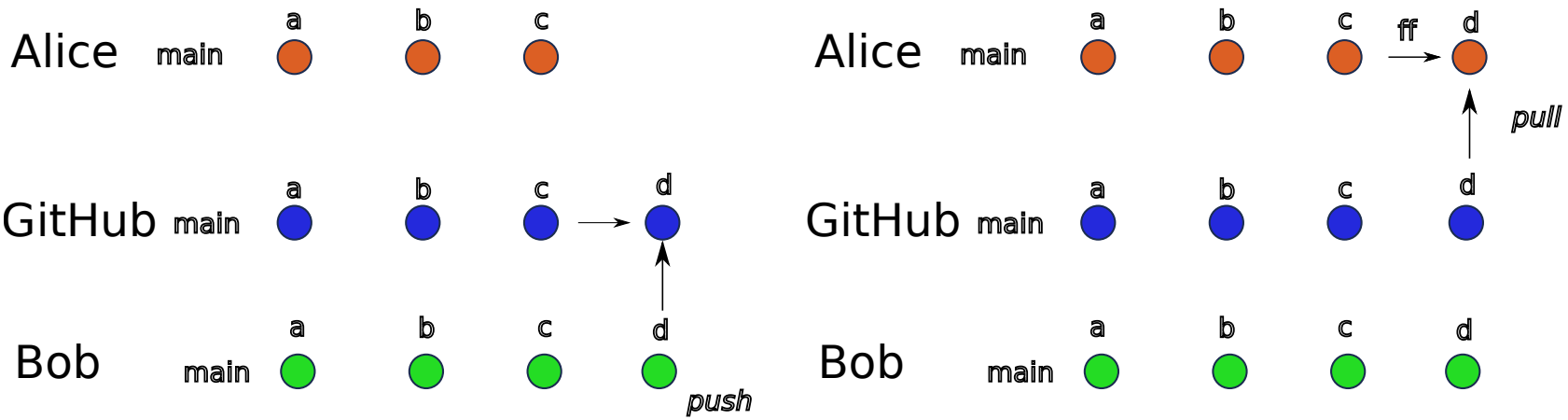
When Alice pushes to the remote the final situation on the remote is



With the rebase strategy the history of changes is altered to the benefit of a linear history

Fast forward

For completeness, we mention that if Alice had pulled the changes made by Bob without having added herself a new commit, we have a fast forward (unless we use `git pull -no-ff`



Overriding default pull strategy

You have set the default pull strategy to *merge* but you want to pull with the *rebase* strategy? Just do:

```
$ git pull --rebase=true <remote> <branch>
```

or, simply,

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

In the opposite situation:

```
$ git pull --rebase=true <remote> <branch>
```

A Note: When pulling from the remote it's better to rebase: you get a cleaner, linear, history.

Not possible to pull because of uncommitted changes

Another thing that may happen frequently is that you cannot pull from the remote repository because you have modified some files and not committed them. Git suggest what you have to do: eather commit or stash. If you are not ready to commit your modification, you use `git stash`

```
$ git stash # stash your changes
$ git pull #pull from the repo
$ git stash pop #get your changes back
```

As simple as that (unless you have conflicts... see later)

Conflicts

Sometimes conflicts may arise!

```
CONFLICT (content):  Merge conflict in file.cpp
Automatic merge failed; fix conflicts and then commit the
result.
```

It happens if Alice and Bob have modified the same lines of the same file! Git will tell you which are the conflicting files. (use `git status`) What do I do now!!?.

Since solving conflicts after a pull is identical to solving them when merging a branch, the solution is reported in the slides about branching.

Adding more files and directories

Now I create a directory `src` where I put some other files `file2.cpp` and `file3.cpp`. I do

```
$ git add src
```

Adding a directory means adding all files in the directory. Doing `git status` gives indeed:

```
On branch master
Your branch is up to date with 'origin/master'.
Changes to be committed:
(use "git restore --staged <file>..." to unstage)
new file:  src/file2.cpp
new file:  src/file3.cpp
```

As expected!

Moving file

But now I realize that also `file1.cpp` should be in `src`! The simplest thing to do is to use

```
$ git mv file1.cpp src/
```

`git status now says`

```
On branch master
Your branch is up to date with 'origin/master'.
Changes to be committed:
(use "git restore --staged <file>..." to unstage)
renamed:  file1.cpp -> src/file1.cpp
new file:  src/file2.cpp
new file:  src/file3.cpp
```

Great! I am ready for the next commit!

```
$ git commit -m "Added a few more files"
```

Modifying a file

You now work on file3.cpp to better an algorithm of correcting a bug. After the changes `git status` gives

```
Your branch is ahead of 'origin/master' by 1 commit.
(use "git push" to publish your local commits)
Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git restore <file>..." to discard changes in working directory)
modified:  src/file3.cpp
no changes added to commit (use "git add" and/or "git commit -a")
```

As you can see, besides the other information, it marks `file3.cpp` as modified. If you want to register the modifications in git you can type

```
$ git add src/file3.cpp
$ git commit -m "corrected a bug"
```

or, with a single command

```
$ git commit -a -m "corrected a bug"
```

Here, `-a` means "add all modified files before commit".

A note

You can launch git commands from any directory of the working tree! Non necessarily from the root.

The possible state of a file in the working area

To summarize, a file in the working area con be in one of these 3 states:

- untracked The file is not under git control. Maybe it is just a temporary file. Or maybe you want to put it under control using `git add <file>`.
- modified The file has been modified since the last commit in the repository. May be you want to add it to the staging area with `git add <file>`.
- staged (in the index, or staging area). Ready to be committed into the repository with a `git commit`.

Some shorthand names for commits

Shorthand	Definition
HEAD	Last commit
HEAD^	One commit ago
HEAD^^	Two commits ago
HEAD~1	One commit ago
HEAD~3	Three commits ago

These shorthands can be used in all git commands that refer to a commit. `HEAD` may be thought as a pointer to the last commit.

```
$ git log HEAD~3..HEAD
$ git checkout HEAD~2
```

The first command prints a brief log describing the last 4 commits. The second positions your working area to two commits ago.

Checking-out a commit

```
$ git checkout <treeish>
```

Where `treeish` can be:

- The hash key of a commit, or a `tag` name;
- The name of a branch: it will checkout the last commit (`HEAD`) of that branch;
- The shorthand name of a commit.

Important

If you checkout a commit that is not `HEAD` of a branch (we will deal with branches soon!) then you are put to a detached HEAD state. You can modify files but not make commits! If you want to commit changes made while in detached `HEAD` state you have to create a branch:

```
$ git checkout HEAD~1
Note:  checking out 'HEAD~1'.
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 -b with the
checkout command again.  Example:
git checkout -b <new-branch-name>
HEAD is now at f041086 Added lecture on C++ evolution
```

Fixing commits: amend reset and revert

Sometimes you want to correct your last commit, for instance to add some additional stuff, correcting the commit message, . . . , without creating a new commit

```
$ # do your additional work/add/rm
$ git commit --amend
```

Never amend a commit that has been already pushed to a remote repo

Undo staging

`git reset` sets yours HEAD to a specified commit. But it is a very versatile command

```
$ git reset <file>
```

Removes the specified file (all if `<file>` is empty) from the staging area, leaving the working directory unchanged.
It is the opposite of `git add`.

Eliminating commits but not the changes

```
$ git reset <commit>
```

Move back to <commit>, leaving the working directory alone: all changes made since <commit> will be in the working directory, not staged.

Hard Reset: bring history back!

```
$ git reset --hard <commit>
```

Move the current branch tip backward to `<commit>` and reset both the staging area and the working directory to match. You are back at the situation at commit `<commit>`. All modifications since `<commit>` are lost.

```
$ git reset --hard
```

Equivalent to `git reset --hard HEAD`. In addition to unstaging files, Git will overwrite all changes in the working directory. All uncommitted changes are obliterated, and you go back to the situation at your last commit.

Reset and checkout

Note the difference between

```
$ git reset --hard <commit>
```

and

```
$ git checkout <commit>
```

With `checkout` the commits after `<commit>` (among which `HEAD`) are still there, that's why you are put in a *detached HEAD state*. `reset --hard <commit>` eliminates the commits following `<commit>` from the git history and sets `HEAD` to `<commit>`. All changes since`<commit>` are deleted.

A warning

`git reset --hard <commit>` may change the history.

Do not use it to "delete" commits that have already been pushed to a remote repository shared with collaborators (unless you want to lose friends).

Use `revert` in this case!

Revert

`git revert` reverts commits by applying them backwards. So it does not "delete" them. It is history safe.

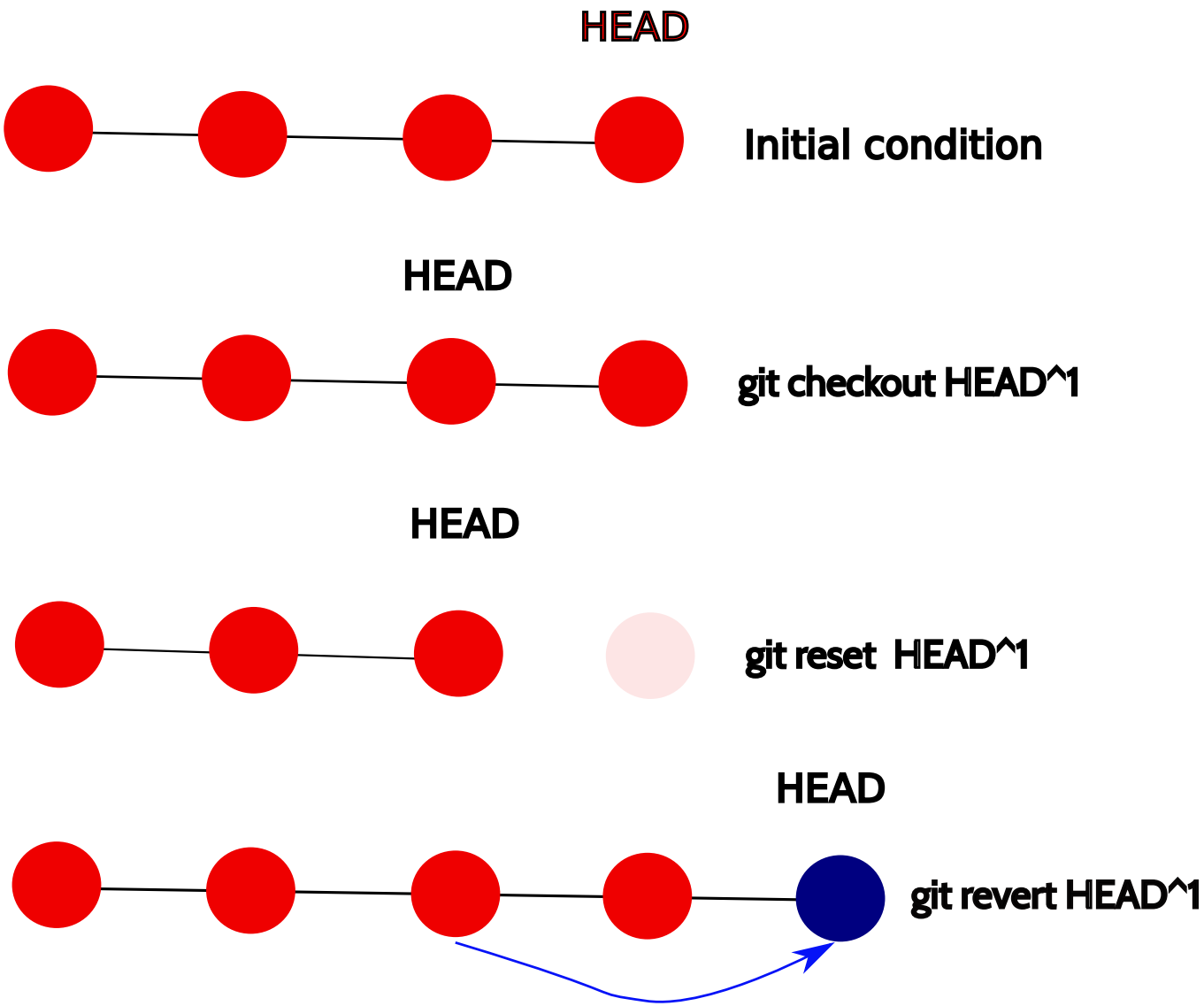
```
$ git revert HEAD~3
```

reverts the changes specified by the fourth last commit in HEAD and creates a new commit with the reverted changes.

```
$ git revert HEAD~3..HEAD
```

reverts all changes introduced in the last four commits, creating a new commit with the reverted changes.

A pictorial view



Ignoring files!

Often in your working directory you have files (temporary results, executables, libraries, ...) that you do not want to store in your repository. You may just ignore them, but if you want to avoid making mistake (and stop `git status` indicating them as untracked) you can create a `.gitignore` file with the list of files to ignore (you can use wildcards)

`.gitignore` may be at the repo root level and in a sub-folder. In the latter case, it applies to files in the folders rooted in that sub-folder.

Remember to add and commit `.gitignore`, it must be in your repo!

Cleaning up!

All commands seen in the previous slides (like almost all git commands) operate on file under git control. Sometimes you want to get rid of untracked files and have in your working area only the tracked one.

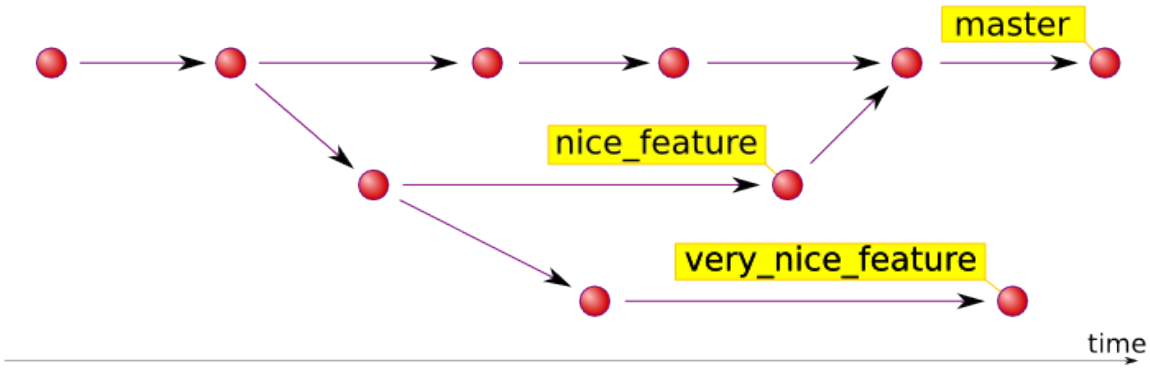
```
$ git clean [-d] [-f] [-x]
```

Option `-f` (force) is in fact compulsory. With just that option the command cleans all untracked files not listed in `.gitignore` (the files in `.gitignore` are untracked but considered as known to git.), with `-x` you get rid also of them. Directories emptied by the cleaning are kept unless you specify `-d`
There are also other options: *e.g.* -i = interactive mode

Some advice

- Do atomic commits: each commit should be related to a single "logical change" in your code: a bug fix, a new feature, ... Avoid monster commits with a lot of modified files.
- Write significant commit messages. Messages like "this is a commit" are useless! A good message helps you to remember what you have done, and others understand your work.
- Git doesn't allow commits with empty messages.

Branches



Branching is one of the most important feature of git. When you create a repo you create also the `master` branch (sometimes called `main`). But often you want to create a branch to work on a specific topic (or just to experiment something) without affecting the master branch.

Branches: Create

create new branch

```
$ git branch shiny_new_branch
```

Go to that branch.

```
$ git checkout shiny_new_branch
```

Or create and position yourself in a new branch with a single command

```
$ git checkout -b shiny_new_branch
```

Get other branches

list local branches

```
$ git branch
  b1
* b2
  master
```

list remote branches (*i.e.* branches fetched from remote, see fetching later on)

```
$ git branch -r
```

list all branches (local and remote)

```
$ git branch -a
```

Branches: Delete

delete a branch

```
$ git checkout master
$ git branch -d branch_to_delete
```

will cause an error if the branch was not merged with master!
To delete it without merging with master

```
$ git branch -D branch_to_delete
```

To delete a remote branch (see later for remotes)

```
$ git push origin :branch_to_delete
```

Branches: Merge

Often when you have finished working on a branch and you are happy of your work you want to merge the work on the `master` branch (or any other branch in fact).
To merge a branch into master:

```
$ git checkout master
$ git merge branch_name
```

You can also rebase before merging

```
$ git rebase master #rebase current branch on master
$ git checkout master
$ git merge branch_name
```

Branches: Conflicts

Sometimes conflicts may arise!

```
CONFLICT (content): Merge conflict in file.cpp
Automatic merge failed; fix conflicts and then commit the
result.
```

What do I do now!!?.

Why conflicts?

Git tries to do its best to merge the work you have done in the branch, but maybe in the meantime you or one of your collaborators have modified the same lines of the same file on `master` (or the branch to which you are merging).

In that case you have a conflict, git stops the merge and it is up to you to sort out the mess.

If you want to cancel the merge and go back to the situation just before:

```
$ git merge --abort
```

Branches: Solve conflicts I

Manual solution. In the file with the conflict you have

```
<<<<<<< HEAD
...portion in your HEAD commit
=======
...portion in the branch you are merging
>>>>>>> branch_name
```

Use your preferred editor and modify the file. Eventually you should eliminate the <<<<< and >>>> lines.

There are several graphic tools to help you fixing conflicts, such as `KDiff3` or `meld`, that can be activated using `git mergetool`. Or integrated in your IDE.

Branches: Solve conflicts II

Using a gui

```
$ git mergetool --tool=<tool>
merge tool candidates:  meld opendiff kdiff3 tkdiff xxdiff
tortoisemerge gvimdiff diffuse ecmerge p4merge araxis emerge
vimdiff
```

You may use one of the tools indicated (you must have it installed). Some of them are rather sophisticated and provide a nice graphical interface.

Branches: Solve conflicts III

After you have fixed the conflicting files, you have to stage them

```
$ git add file.cpp
```

A commit is needed when all the conflicts are solved!

```
$ git commit
```

A special branch: the stash

Checkout with hanging modifications is forbidden!

```
$ git checkout master
error:  Your local changes to the following files would be
overwritten by checkout:  ...
```

Instead of creating a branch we can stash modifications

```
$ git stash save
```

and bring them back

```
$ git stash { pop | apply }
```

remember to clear the stash if you use apply, which does not delete the stash.

```
$ git stash clear
```

Summary on commands operating on a remote

Share/retrieve a new branch with a remote called `repo_name`

```
$ git push repo_name branch_name
$ git pull repo_name branch_name
```

Access a new branch in the repo after having fetched it

```
$ git checkout --track -b branch_local_name \
    repo_name/branch_name
```

If you have just one repo it is sufficient to do

```
$ git checkout --track branch_name
```

A tracking branch is a branch that is linked to the corresponding branch in a remote repository. Indeed you may have local branches just for local modifications.

Summary on commands operating on a remote

Pull and adjourn submodules (if any)

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

Pull and do rebase instead of merging

```
$ git pull --rebase
```

Pull and do merge instead of rebase

```
$ git pull --rebase=false
```

Fetch instead of pulling (and merge/rebase later)

```
$ git fetch repo_name
$ git merge|rebase remote/repo_name/branch_name
```

Get info

```
$ git log
```

Shows commit logs.

```
$ git shortlog
```

Summarizes log output, showing commit description from each contributor

```
$ git show <object>
```

Show some properties of the `object` (blob, commit, ...)

Use a graphical interface like `gitk` or `git-cola`, ...

Other useful commands

To show the differences with the last commit (you may specify a different commit and/or some specific files)

```
$ git diff branch files
```

To remove file(s) under git control from the working directory and stage it for removal in the next commit

```
$ git rm file(s)
```

To change the name of a file under git control (stages the change for the next commit)

```
$ git mv oldname newname
```

Some advanced features: submodules

A git repository may contain another git repository (and this can be done recursively!). It is useful if you want to keep in your repository another, independent, project, so that you can use it and update it if some useful changes happen upstream.

Example: you want to add to your repo a local copy of `https://github.com/PatWie/CppNumericalSolvers.git`.
Go to the directory where I want to store the submodule

```
$ cd myrepo/numerical_solvers
```

```
$ git submodule add \
    https://github.com/PatWie/CppNumericalSolvers.git
```

Now in `myrepo/dir/CppNumericalSolvers` you have a copy of the remote repo.

Operating on submodules

You can update a submodule so that it is kept up-to-date with upstream (`--recursive` is needed only if you have more submodules inside submodules)

```
$ cd myrepo
$ git submodule update --recursive
```

Get info about submodules

```
$ git submodule status
$ git submodule summary
```

Operating on submodules

Remember that a submodule is a git repo!

```
$ cd myrepo/dir/CppNumericalSolvers $ git status
```

I get the status of the git repository associated to `CppNumericalSolvers`, not of the hosting repository!
What if I clone `myrepo` somewhere else and I want to fetch also the submodules?

```
$ git clone <myrepo> newrepo
$ cd newrepo
$ git submodule init
$ git submodule update --recursive
```

To update submodules when pulling

```
$ git pull -recurse-submodules <-rebase>
```

Frequently Asked Questions

Common issues - I

Q edited on master branch by mistake, I wanted to do in new_branch instead!

A a simple checkout will do it

```
$ git checkout -b new_branch
```

Q already committed to master branch

A create a new branch and delete the commit on master

```
$ git branch new_branch
$ git reset --hard HEAD~1
$ git checkout new_branch
```

Common issues - II

Q I want to discard my edit and get the latest version of the file!

A Simple

```
$ git checkout -- file
```

Q I want to get the file from another branch or another commit!

A It's the same command as before:

```
$ git checkout <branch|commit> -- file
```

Common issues - III

Q Get a commit from another branch

A cherry-pick it to the correct one

```
$ git checkout correct_branch
$ git cherry-pick commit_hash
```

Q cannot get the branch of the local remote

A Try with a fetch

```
$ git fetch remote_name
$ git remote show remote_name
```

Common issues - IV

Q the merge has gone bananas!

A give up the merge and try again

```
$ git merge branch_to_be_merged
$ $!&#%$&^
$ git reset --hard original_branch
```

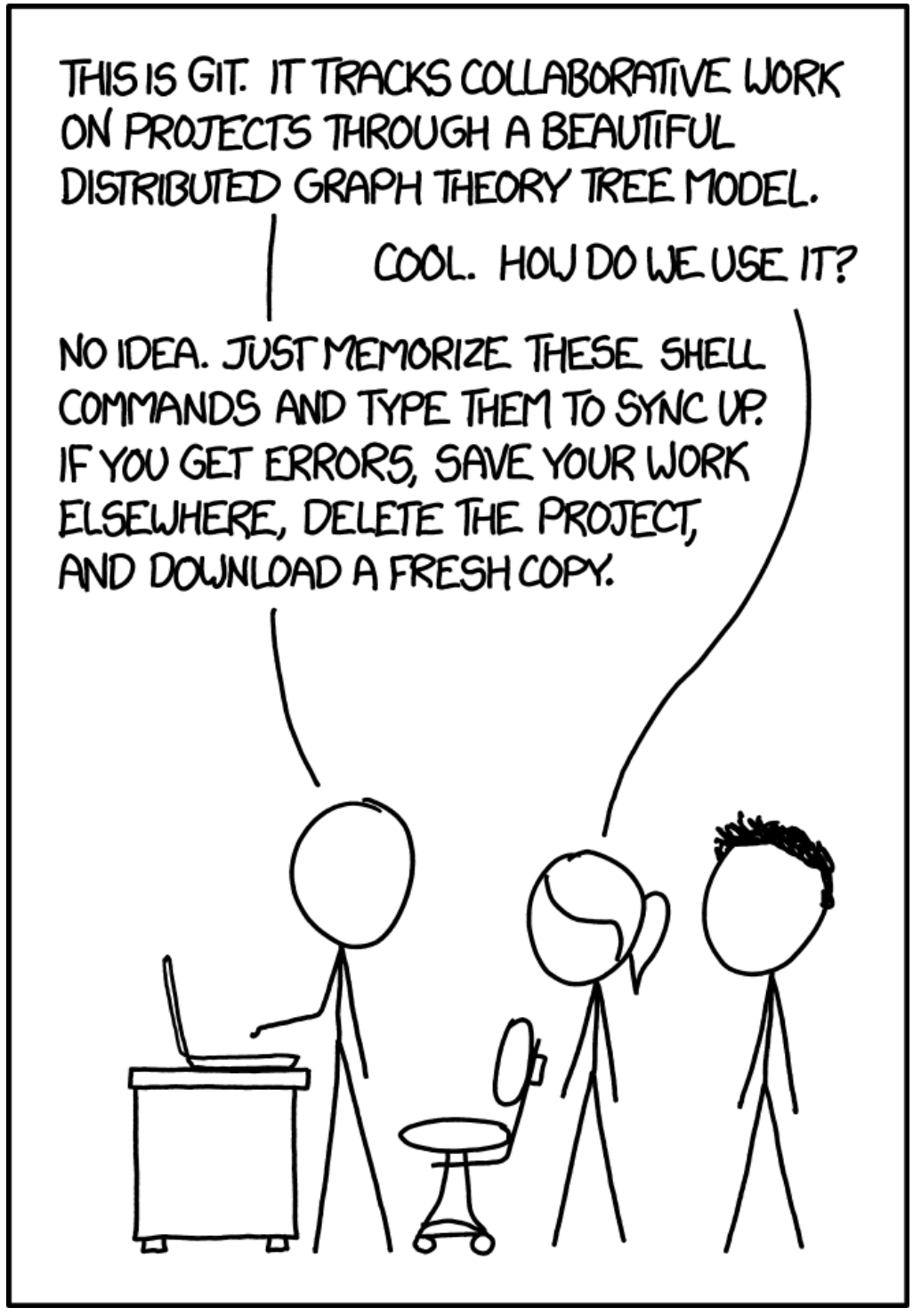
or, more simply

```
git merge --abort
```

Q I want a file from that branch!

A get it with

```
$ git checkout branch files
```


THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

And remember...