Advanced Git

DAVID PARSONS

1
Reminders

Generalities

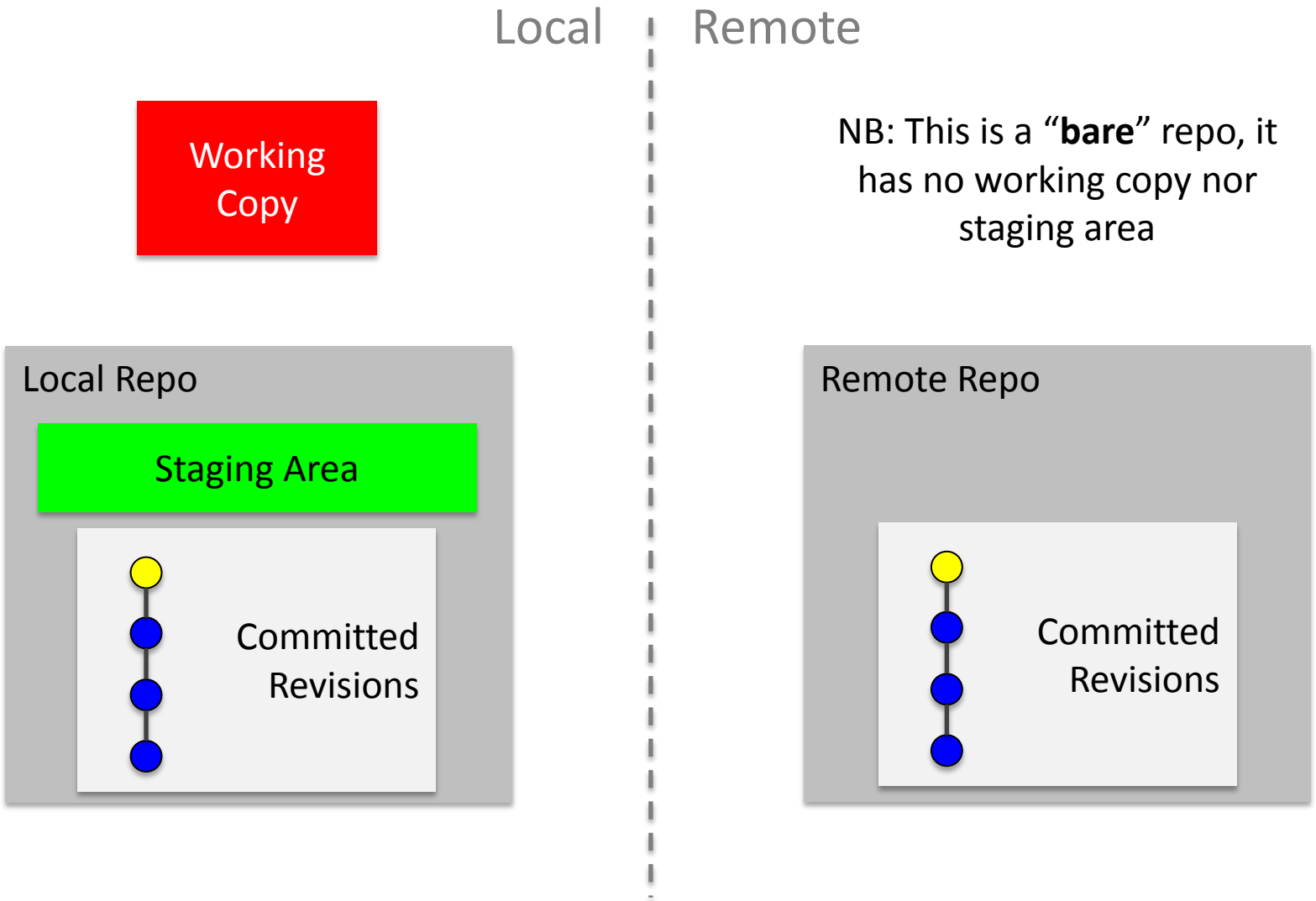
Two fundamental rules !

- Commit often
  - Keep commits small and commit together only related changes (commit = minimal independent changeset)
- Write clear and informative logs
  - A log should enable its reader to:
    1. Identify at a glance the rationale behind the commit
    2. Have detailed explanation if needed
  - Template for logs (from http://git-scm.com/book/ch5-2.html)

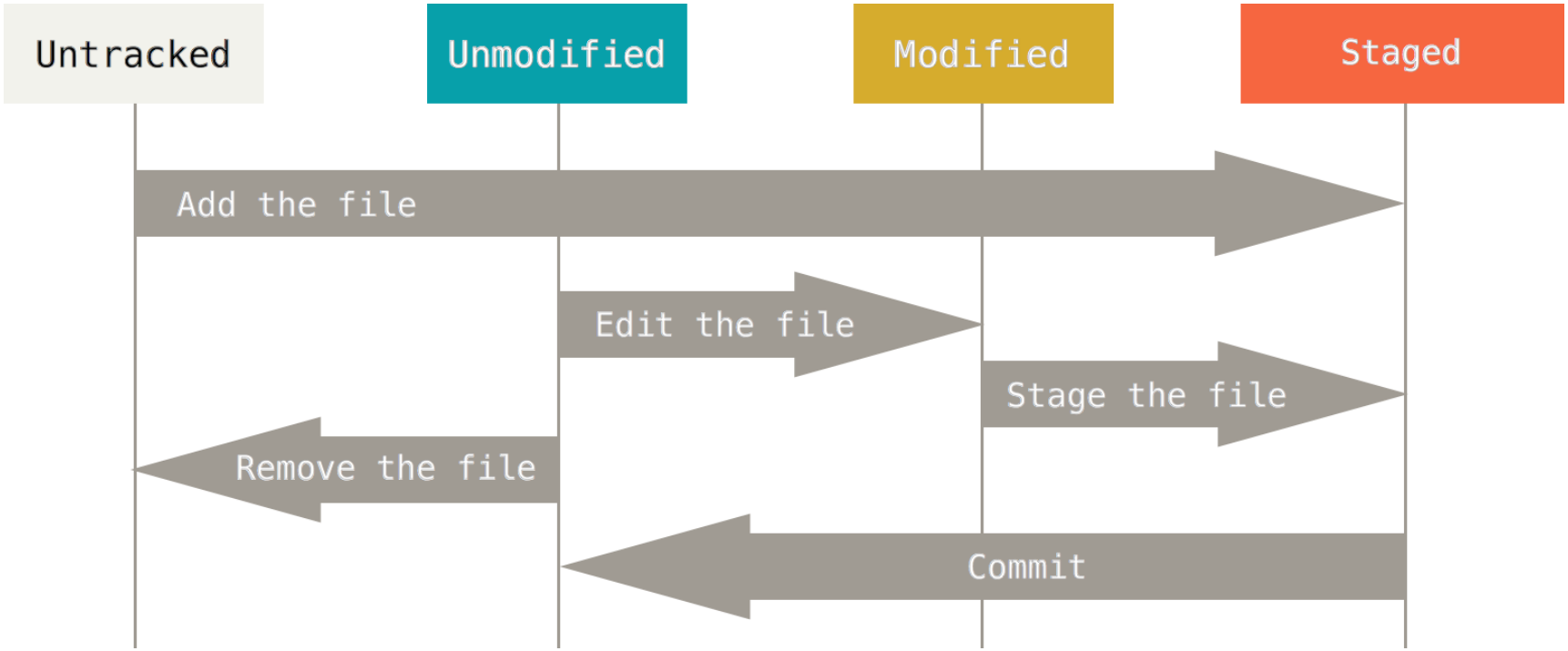
Short (50 chars or less) summary of changes

More detailed explanatory text, if necessary. Wrap it to about 72 characters or so. In some contexts, the first line is treated as the subject of an email and the rest of the text as the body. The blank line separating the summary from the body is critical (unless you omit the body entirely).

From working copy to remote repo



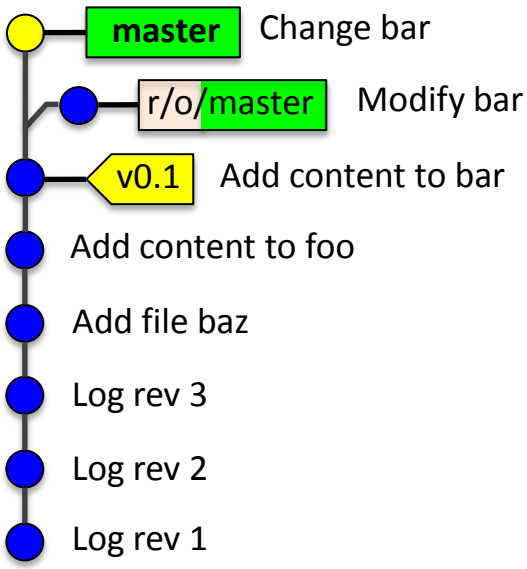
File Status Lifecycle



Managing conflicts

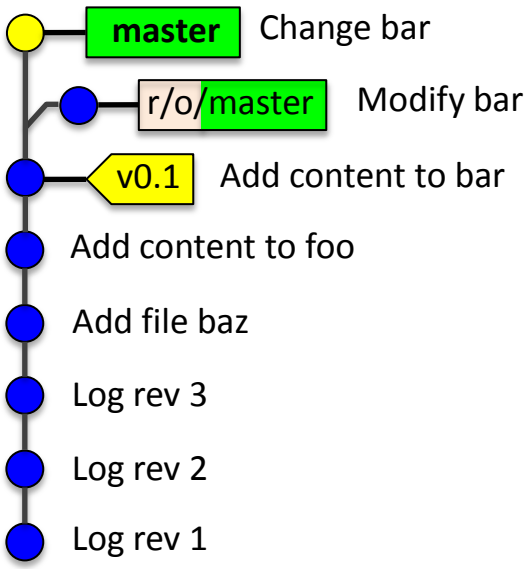
Managing conflicts

```
$ git merge
```

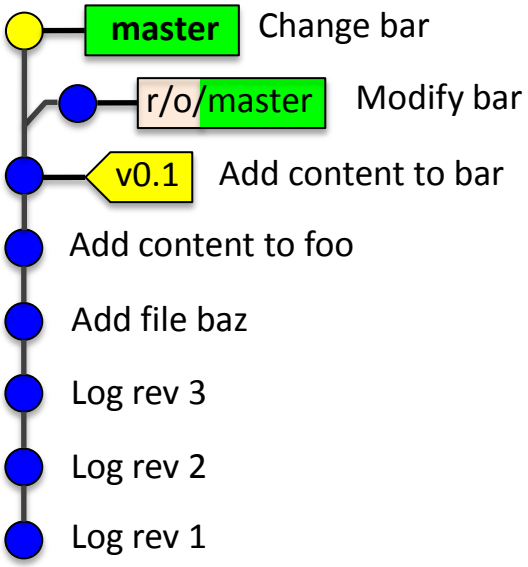


Managing conflicts

```
$ git merge
Auto-merging bar
CONFLICT (content): Merge conflict in bar
Automatic merge failed; fix conflicts and then
commit the result.
$
```



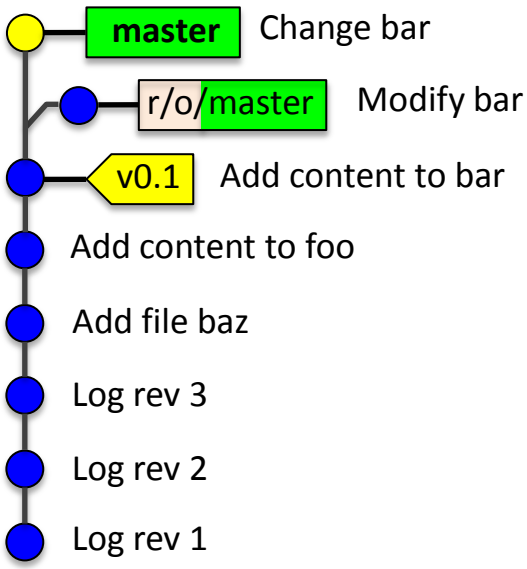
Managing conflicts



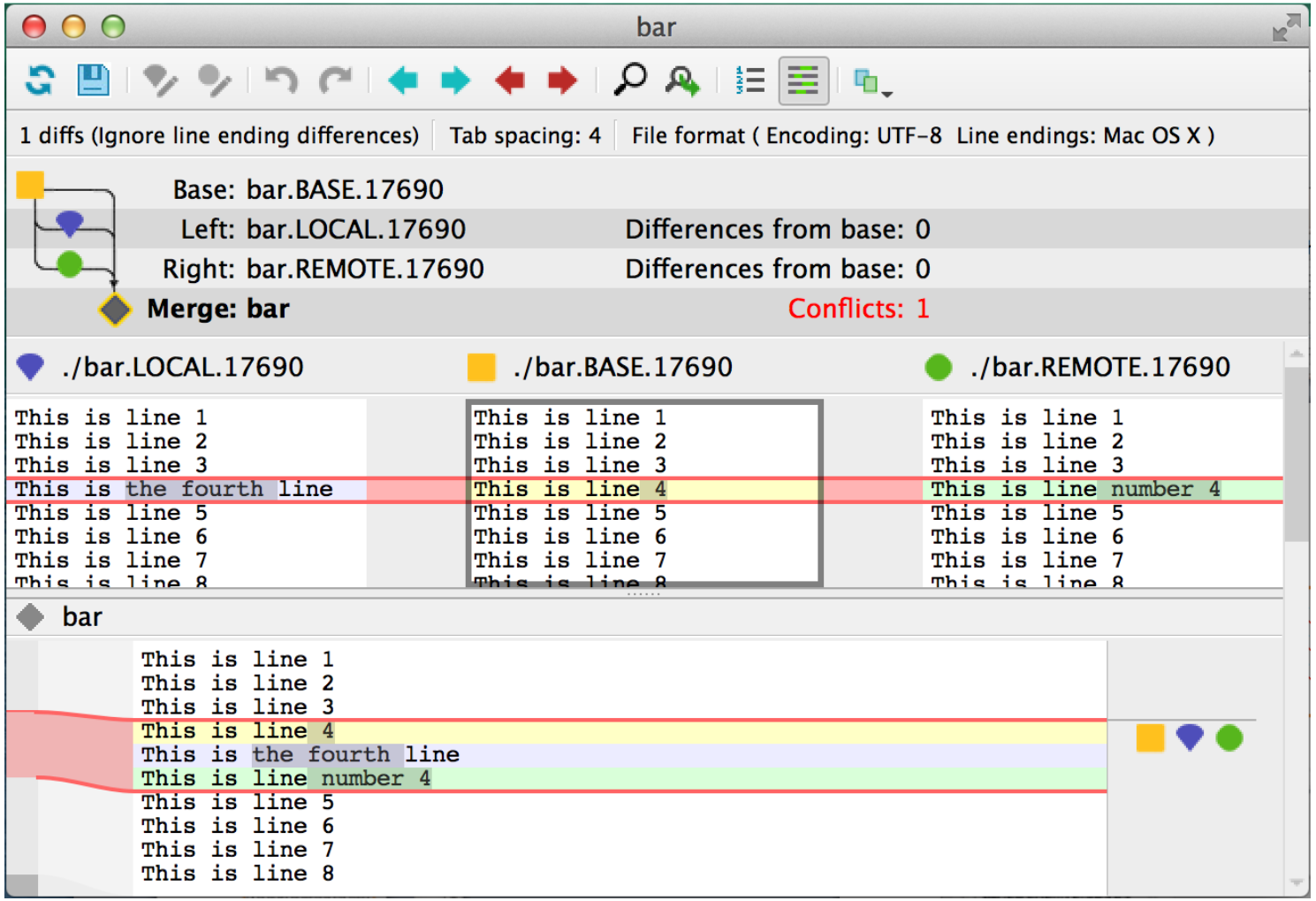

```
$ git merge
Auto-merging bar
CONFLICT (content): Merge conflict in bar
Automatic merge failed; fix conflicts and then
commit the result.
$
$ cat bar
This is line 1
This is line 2
This is line 3
<<<<<<< HEAD
This is the fourth line
=======
This is line number 4
>>>>>>> featureA
This is line 5
This is line 6
This is line 7
This is line 8
$
```

Managing conflicts

```
$ # You can edit the conflicting files directly
and then stage them and commit, or you can use
a merge tool
$
$ git mergetool
```

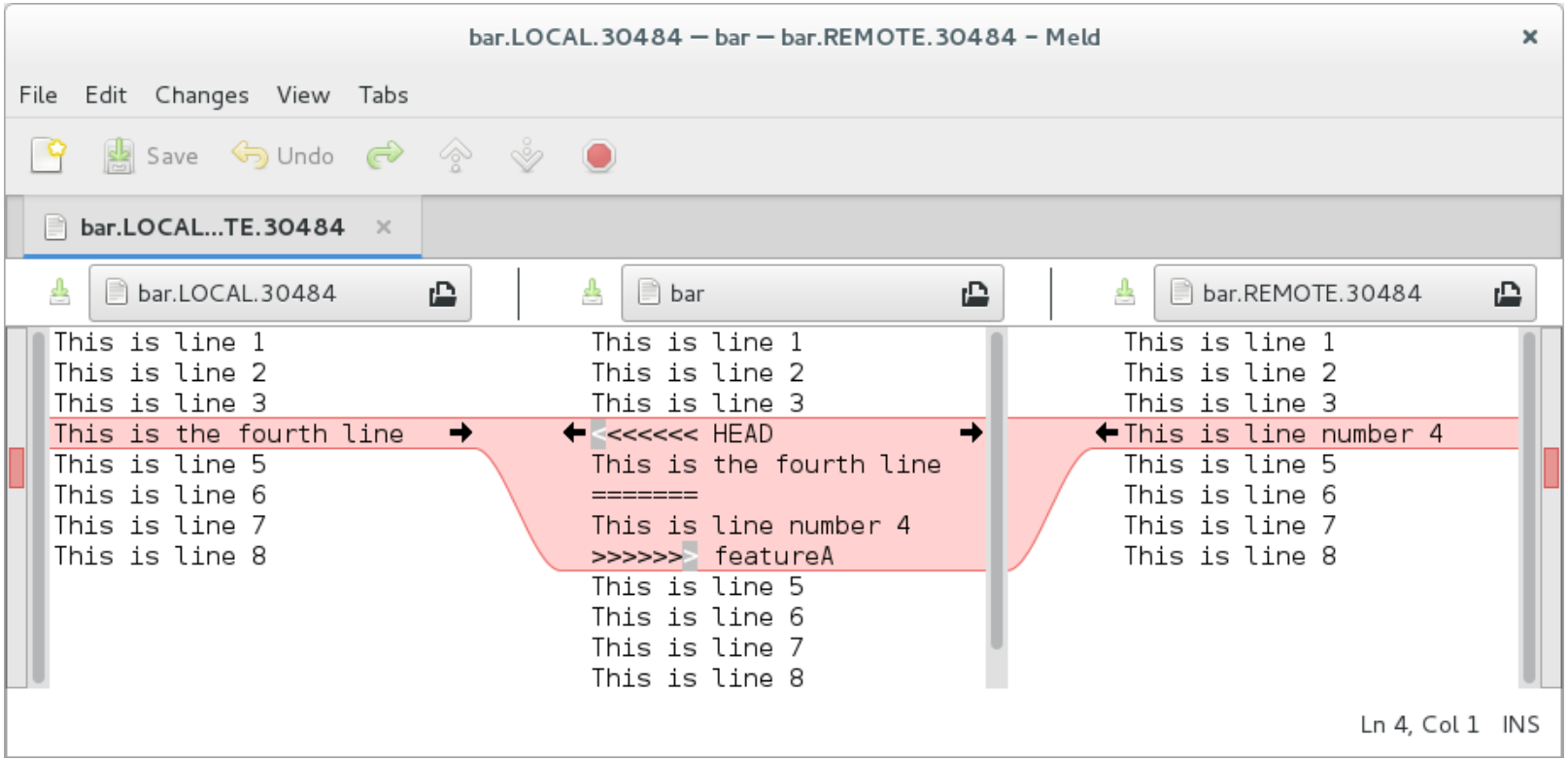


mergetool – p4merge



Configuring a mergetool

```
$ git config --global merge.tool meld
$ git config --global mergetool.meld.cmd 'meld $LOCAL $MERGED $REMOTE'
$ git config --global mergetool.meld.trustExitCode false
$
```



2

Things to know

Configuring git

```
# Configure your name and e-mail address (almost mandatory)
$ git config --global user.name "David Parsons"
$ git config --global user.email david.parsons@inria.fr
$
$ # Configure the editor git will open when needed
$ git config --global core.editor nano
$
$ # Setup a few aliases, either simple shorthands...
$ git config --global alias.co checkout
$ git config --global alias.ci commit
$ git config --global alias.s  status
$
$ # ... or including options
$ git config --global alias.lg "log --pretty=format:\"%h - %an : %s\""
$
$ # You can even create new commands
$ git config --global alias.unstage "reset HEAD”
```

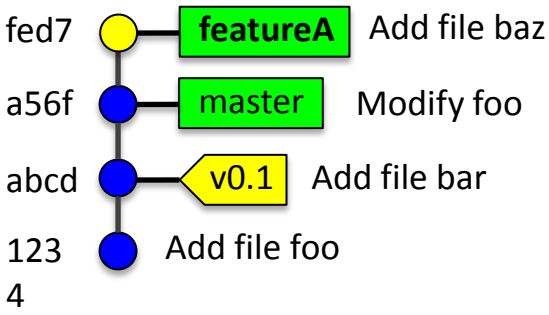
Detached HEAD ?

Detached HEAD ?

- You are in detached HEAD when you are not “on” any branch
  - Most of the time, this happens when you provide anything that is not a branch to `git checkout`. *E.g.* a tag, a SHA-1, an indirect reference (HEAD~1)
  - You can also use `--detach` when providing a branch name
- When in detached HEAD, you can commit as you like ; but be wary of the garbage collector, it could very well erase your work if you do not pay attention !
- My advice: always create a branch to commit your work to, it costs nothing to create a “tmp” branch and delete it when you do not need it any longer.

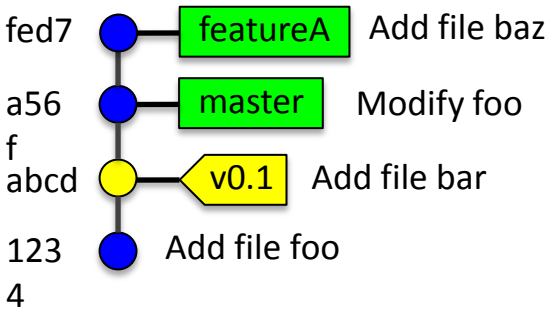
Detached HEAD ?

```
$
```



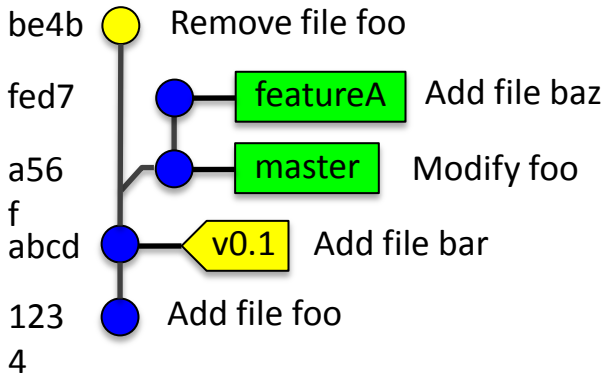
Detached HEAD ?

```
$ git checkout v0.1
Note: checking out 'v0.1'.

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 abcd1b3... Add file bar
$
```



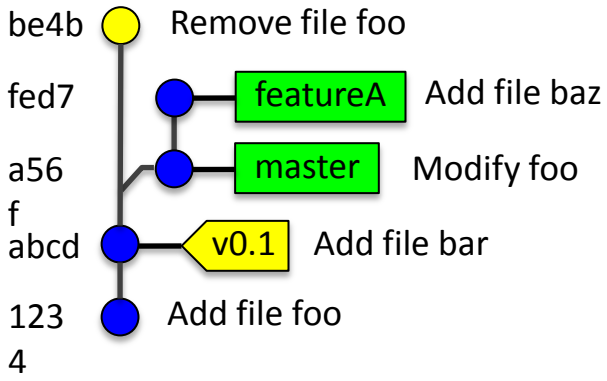
Detached HEAD ?

```
$ git rm foo
$ g ci -m ”Remove file foo”
[detached HEAD be4b243] Remove file foo
 1 file changed, 1 deletion(-)
 delete mode 100644 foo
$
```



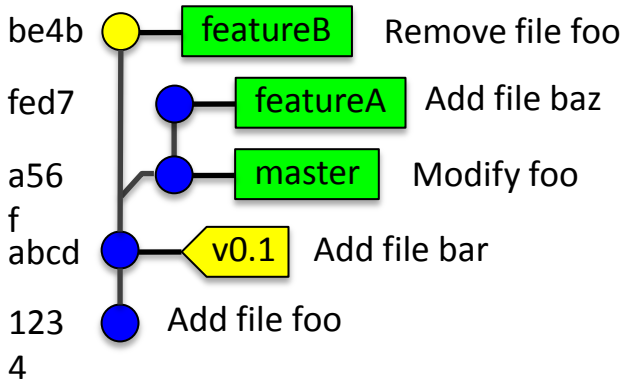
Detached HEAD ?

```
$ git rm foo
$ g ci -m ”Remove file foo”
[detached HEAD be4b243] Remove file foo
 1 file changed, 1 deletion(-)
 delete mode 100644 foo
$ g s
HEAD detached from v0.1
nothing to commit, working directory clean
$
```



Detached HEAD ?

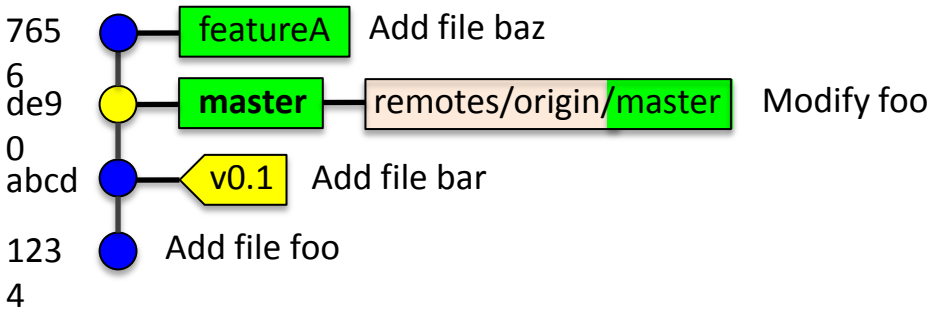
```
$ git co -b featureB
Switched to a new branch 'featureB'
$
```



Remote tracking and Upstream branches

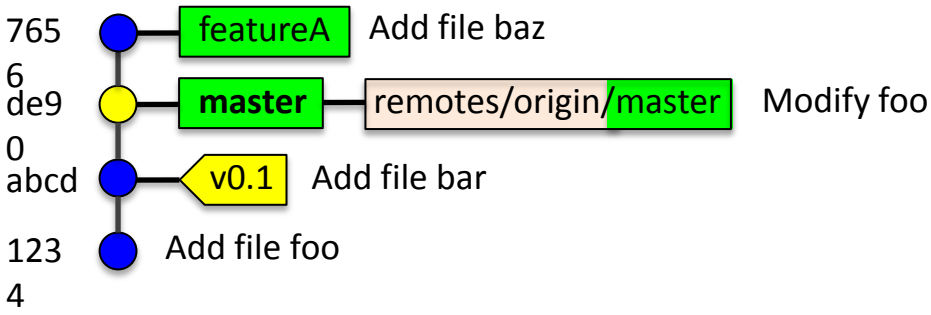
Remote tracking branches

```
$ git branch # List local branches
  featureA
* master

$ git branch -r # List remote-tracking branches
  origin/master

$ git branch -a # List both local and remote-tracking branches
  featureA
* master
  origin/master

# Question: what happens when you check out a remote tracking branch?
```



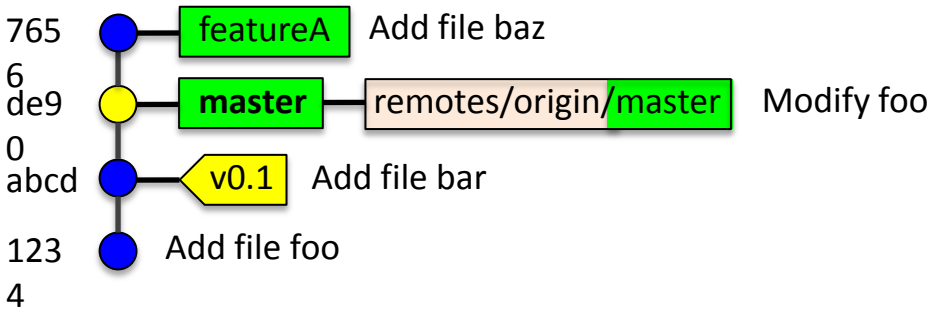
Upstream branches

```
$ git pull # What exactly does git pull do (or try to do)?
```



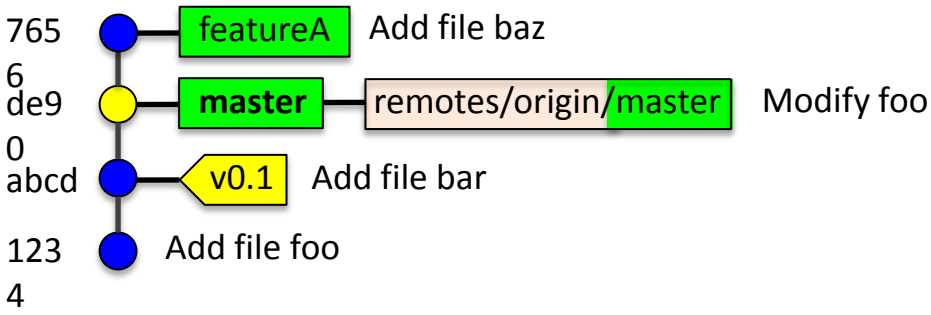
Upstream branches

```
$ git pull
There is no tracking information for the current branch.
Please specify which branch you want to merge with.
See git-pull(1) for details

    git pull <remote> <branch>

If you wish to set tracking information for this branch you can do so
with:

    git branch --set-upstream-to=origin/<branch> master

$
```



Upstream branches

```
$ git branch -vv
  featureA 7656d7c Add file baz
* master   de90ac4 Modify foo
$ git branch -u origin/master
Branch master set up to track remote branch master from origin.
$ git branch -vv
  featureA 7656d7c Add file baz
* master   de90ac4 [origin/master] Modify foo
$ git pull
Already up-to-date.
$
```



3

Git internals (dive into .git)

4

Small interesting tools

Creating and applying patches

Patches

- For some reason, you could want to send one or more changesets to a collaborator without actually pushing
- There are basically 2 ways of doing that
  - For a single changeset, whether it has been committed or not, you can use `git diff` to generate a plain patch and `git apply` to apply it
  - For one or more committed changesets, you can use `git format-patch` to generate a series of patches and `git am` to apply them (this will preserve the original committer name)

Cherry-picking

Cherry-picking

- You have spotted a commit made in an otherwise very messy branch and you would love to retrieve that commit (and only that one) in your branch ?
  - Yes, you could create and apply a patch…
  - Or you could use the feature that was created for that very purpose: cherry-picking

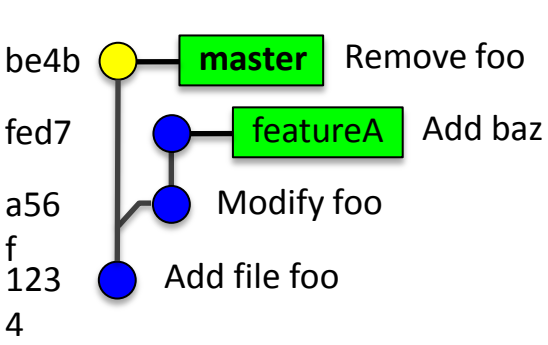
- This is only made possible by sticking to the atomic commits rule !!!

Cherry-picking

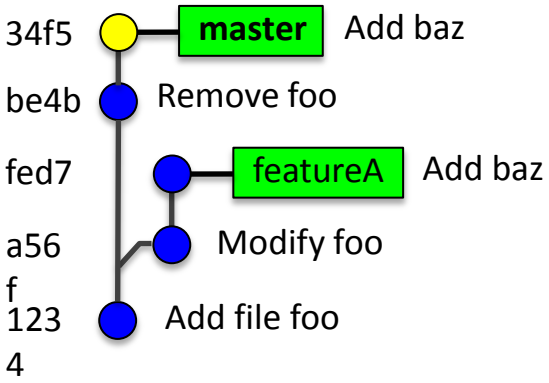
```
# Cherry-pick a fabulous commit:
$ git cherry-pick <a-fabulous-commit>
[…]

# Cherry-pick a sequence of contiguous fabulous commits:
$ git cherry-pick <parent-of-first-in-seq>..<last-in-seq>
[…]

# Get bored of typing     c h e r r y - p i c k
$ git config --global alias.cp cherry-pick
```

Cherry-picking

- Really easy and very handy
- Allows you to apply the changes introduced by one or more commits on top of HEAD




```
$ git cp fed7
```



“Partial” commits

“Partial” commits

- Commits should be **atomic**: a commit must introduce a minimal independent changeset
- Most programmers have difficulties focusing on a single task (did you just correct a typo while scrolling over the code in the midst of a bugfix ?)

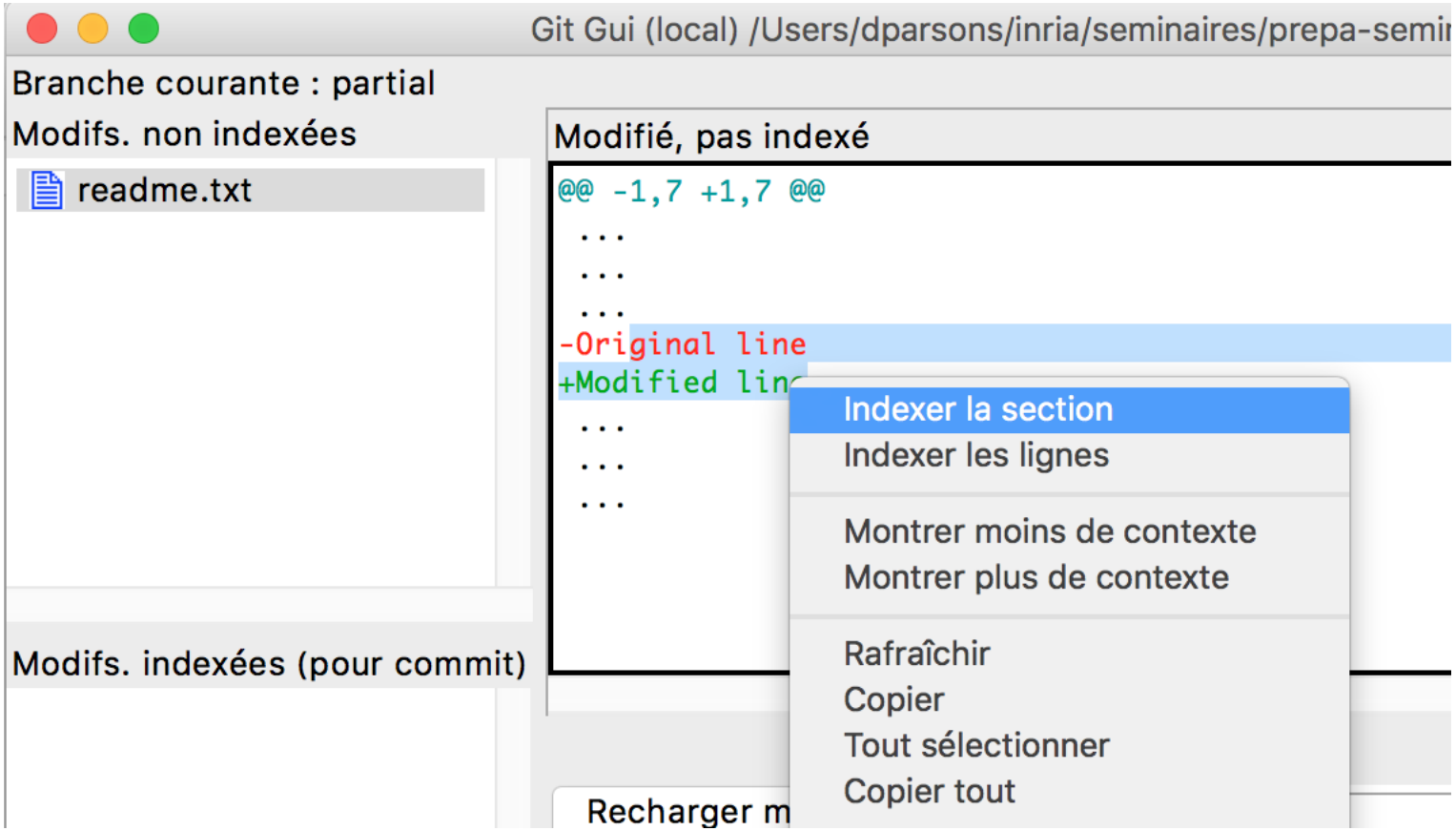
- `git add --patch` (or `-p`) is your friend !
- `git gui` is your better friend, it allows you to add parts of a file to your staging area on a per-hunk or even per-line basis

git add -p

- Works fine for simple cases (tedious otherwise)

```
$ git add -p
diff --git a/file b/file
index e72f11f..75ad869 100644
--- a/file
+++ b/file
@@ -1,7 +1,7 @@
 ...
 ...
 ...
-  Original line
+  Modified line
 ...
 ...
 ...
Stage this hunk [y,n,q,a,d,/,j,J,g,e,?]?
```

Using `git gui`



Stash

Stash

- Literally “garder sous le coude”

```
# You’re working on something and need to switch to something else
$ git co something-else
Error: Your local changes to the following files would be overwritten
by checkout:
[…]
Please, commit your changes or stash them before you can switch
branches.
Aborting
$ g stash [push –m “what you were doing”]
Saved working directory and index state On dev: what you were doing
HEAD is now at 0b0b9eb […]
$ g stash list
stash@{0}: On dev: what you were doing
$
```

Stash

```
$ git co something-else # Now you can (WD is clean)
Switched to branch 'something-else'
$
# Do whatever you needed to do on something_else

# When you want to go back to 'something',
# Checkout:
$ g co something
Switched to branch 'something'

# Recover what you have stashed:
$ g stash pop
# You're back to initial state
$
```

Stash

- Literally “garder sous le coude”
- The stash is a stack (thus the push/pop terminology) of commit objects
- Warning: there can be conflicts when applying (or pop-ing) stashed changes

  In the case of a pop, the corresponding stash will not be dropped, do not forget to do it after resolving the conflicts (`git stash drop` …)
- A (very) useful option: `--keep-index` (or `-k`) stashes only files in the “modified” state

Git bisect

`git bisect`

- Found a bug ?

  Have no idea when (or most importantly by whom) it was introduced ?

- `git bisect` is your next friend on the list, it will operate a dichotomic search in your commit tree

git bisect

```
# Initiate dichotomic search
$ g bisect start

# Mark current commit as bad (has bug)
$ g bisect bad

# Tell git about a commit that is known to not have the bug
$ g bisect good <a-known-good-commit>
Bisecting: 27 revisions left to test after this (roughly 5 step)
[…]

# Not sure about a “good” commit in the first place ?
$ g co <possibly-good-commit>
# Check manually => make check ?
# If this commit is not “good”, you’ll have to go further back in
# your history and check again.
# Don’t hesitate to go far far back in your history, the idea is to
# not drive the search yourself after all ;)
$ git bisect good
```

git bisect

```
# Once you have marked at least one good and one bad commit, the
# dichotomic search will start.
# Git will checkout commit after commit and ask you to mark them as
# good or bad (you can also skip if you're unsure)

Bisecting: 27 revisions left to test after this (roughly 5 step)
[<sha-1>] <commit-msg>
$ git bisect good|bad
Bisecting: 13 revisions left to test after this (roughly 4 step)
[<sha-1>] <commit-msg>
$ git bisect good|bad
ab23ef is the first bad commit
commit ab23ef
Author: David Parsons <david.parsons@inria.fr>
Date: …

<commit message>

List of modified files in the form:
:<old mode> <new mode> <old blob> <new blob> XY <file name>
```

git bisect

```
# Automate search:
$ g bisect start HEAD <known-good>
$ g bisect run make check
[…]
ab23ef is the first bad commit
[…]

# Building out of source ? You can use this:
cat > run_make_check.sh
! /bin/bash
cd build
make check
status=$?
cd ..
exit $status
^D
$ chmod u+x run_make_check.sh
$ g bisect run run_make_check.sh
```

Time for Practical Work !

- Retrieve exercice document :
  http://sed.inrialpes.fr/advancedgit-tuto

➢ To Do : **Section 1 + Section 2**

5
Rewriting History

Rewriting History ?

Remember you have a local repository ?

Everything you have **not** published (i.e. ***pushed***) yet is strictly **local** to your repo. It is known by you and no one else. Since then, what is stopping you from modifying it ?

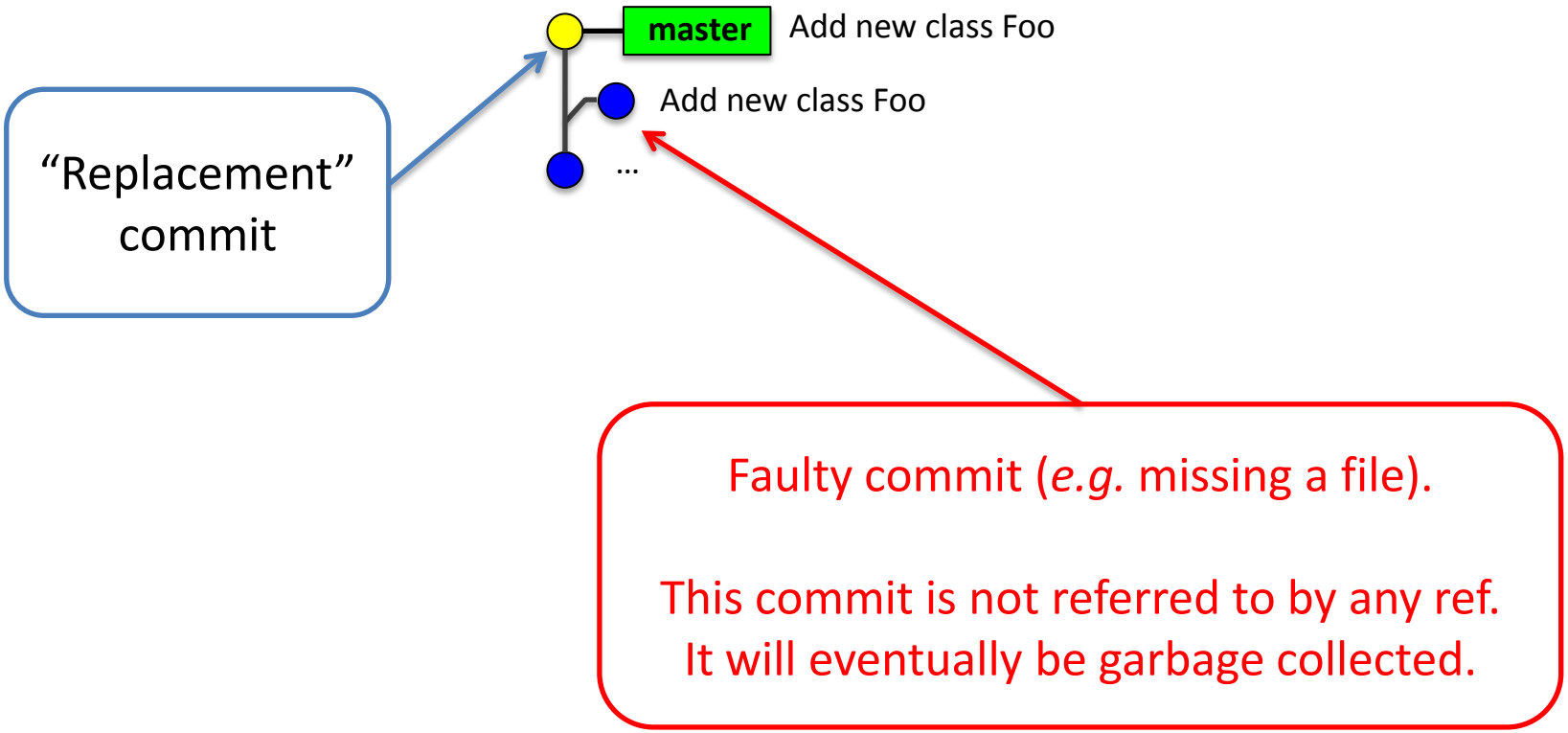
This is one of my favourite things about `git`, I can be stupid and appear not to be !

```
# You've just commited something and realize you forgot to add a file
$ git add the_forsaken_file
$ git commit --amend
# No one saw you ;)
```

WARNING: Do **not** do that if you've **pushed** the faulty commit !!!!!

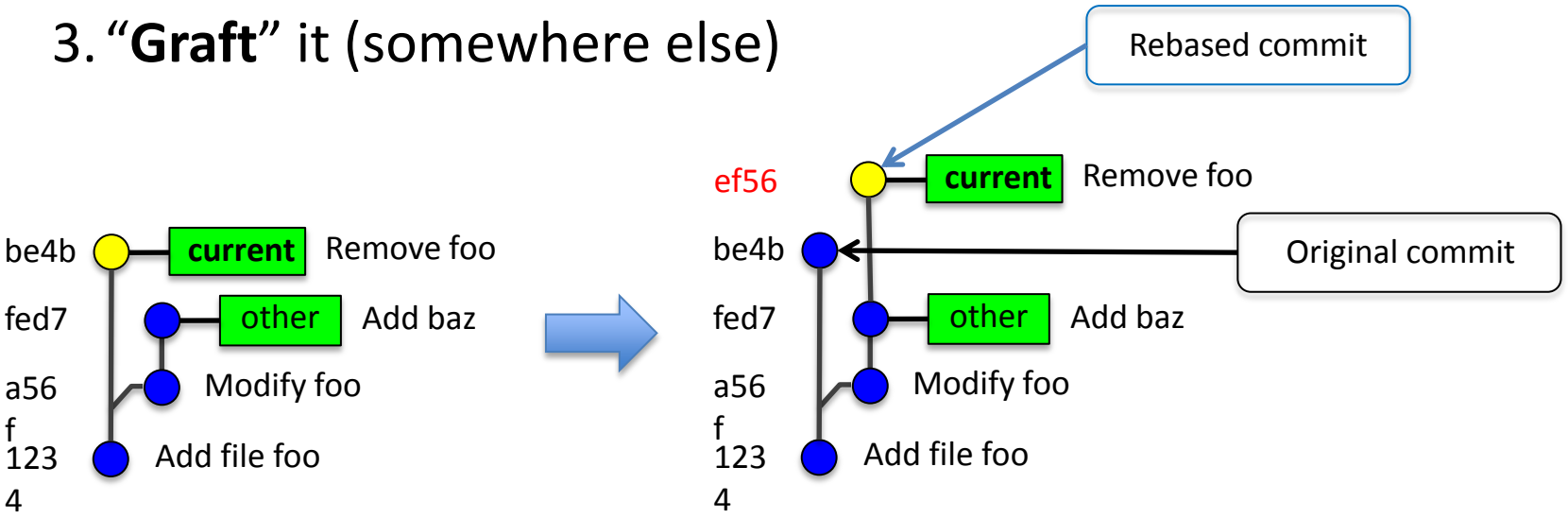
commit --amend

After using `git commit --amend`,
let's look at what our **commit graph** looks like :

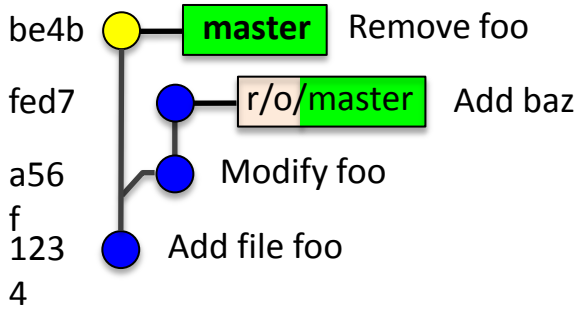


rebase

- Rebase allows you to **linearize a branching tree structure**
- Rebase literally means “set a new base” (for a branch)
- In other words, rebasing a branch consists in telling `git` to :
  1. “**Pick**” a branch
  2. “**Cut**” it (somewhere)
  3. “**Graft**” it (somewhere else)



`rebase`: default behaviour



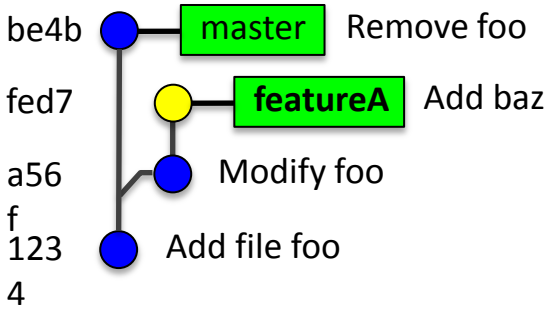
By default, `git rebase`:

- **picks** the **current branch**
- **cuts** it at its LCA (last common ancestor) with its **upstream branch** (if any)
- **graft** it onto its **upstream branch**
  - If your branch and its upstream were in sync, this means you've achieved to do nothing in a complicated way.
  - If you had a branching tree structure (*e.g.* after fetching your colleagues' work – see example above), you've replayed your changes on top of your colleagues'.

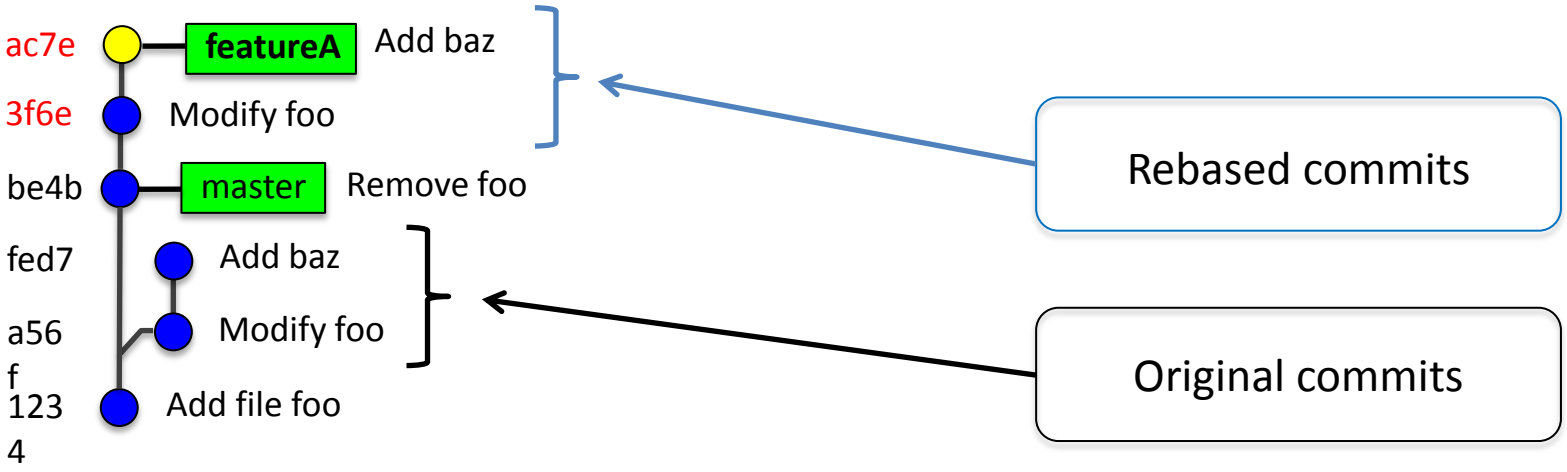
This is safe since it will only alter those commits you haven't pushed yet

`rebase:` one argument form

Branch **master** has been updated since you started to write your feature.
You would like your feature to be up-to-date (to propose it as a pull-request ?)




```
$ git rebase master # Use master for cut and graft steps
First, rewinding head to replay your work on top of it…
Applying: …
```



`rebase:` pick, cut, graft

- `git rebase` simplified synopsis:

```
$ git rebase […] [--onto <newbase>] [<upstream> [<branch>]]
```

What `git rebase` does (this is a lie…):

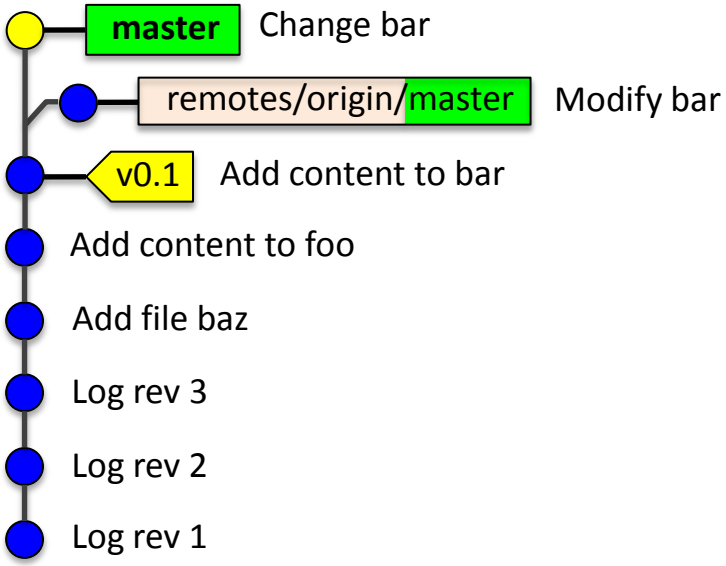
- **picks** 'branch'
- **cuts** it at the LCA (last common ancestor) of 'upstream' and 'branch'
- **grafts** it onto newbase

Actually,

- If 'branch' is specified, `git rebase` will be preceded by a `git checkout 'branch'`
- Nothing is "cut" and the LCA is all but a lie... The changes made by commits in 'branch' that are not in 'upstream' will be recorded and then replayed onto the new base. That means a cherry-picked commit will be ignored (and that is what we want !)

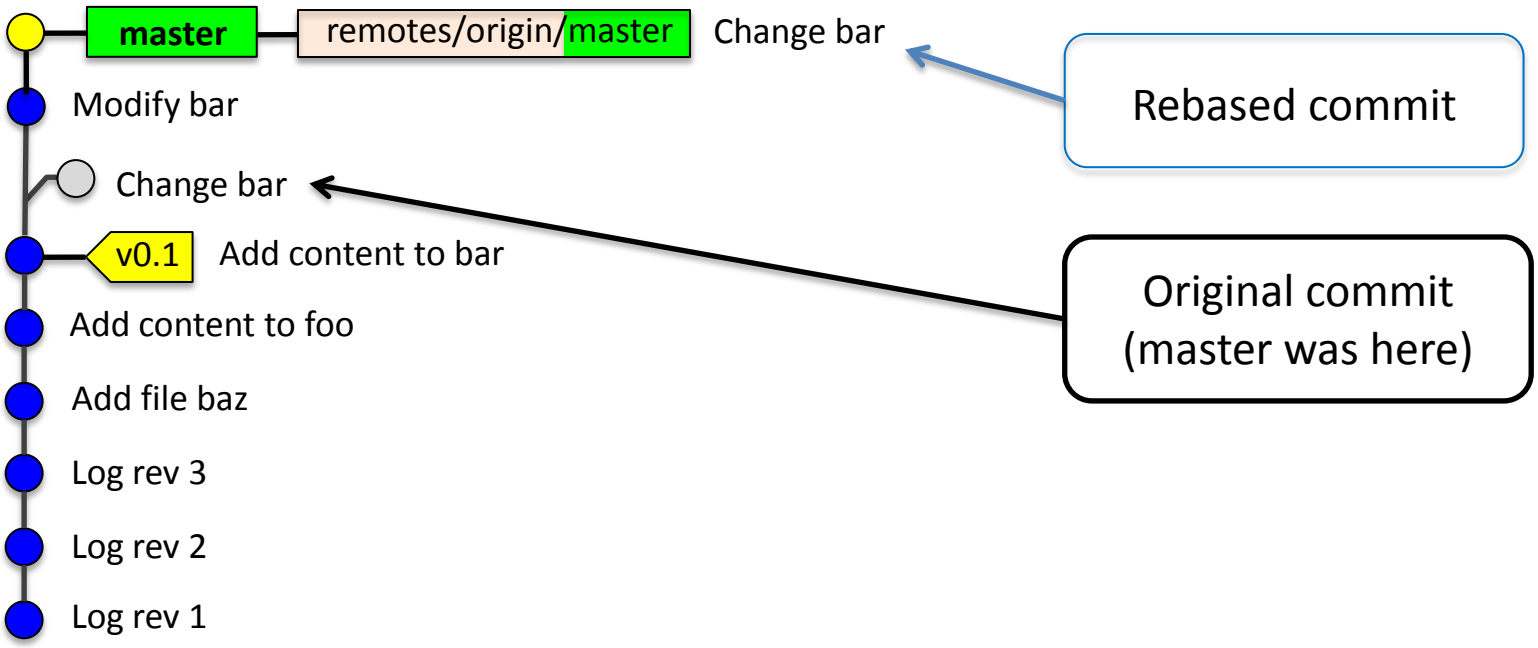
git rebase

```
$ git fetch
$ # About to run git rebase
```



git rebase

```
$ git fetch
$ git rebase
…
$
```



Rebase only **local** commits

Rebasing basically means **re-writing the history of what happened**.

- This is a very powerful feature

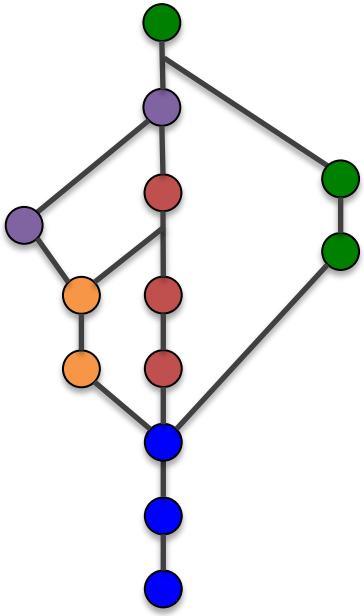
But as always, “great power comes with great responsibility” :

DO NOT REBASE COMMITS THAT EXIST OUTSIDE YOUR REPOSITORY

`merge` or **`rebase`** ?

`merge` or `rebase` ?

Which one would you prefer ?

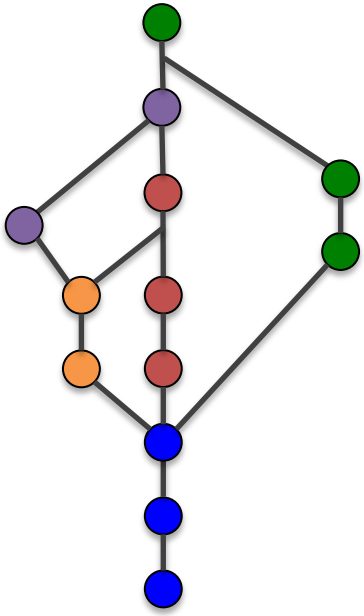


Merge

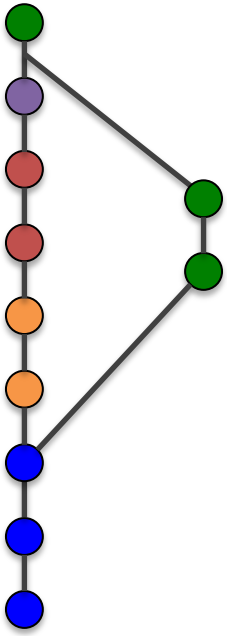
Rebase

`merge` or `rebase` ?

Or maybe this mixed one ?



Merge



Mixed

Rebase

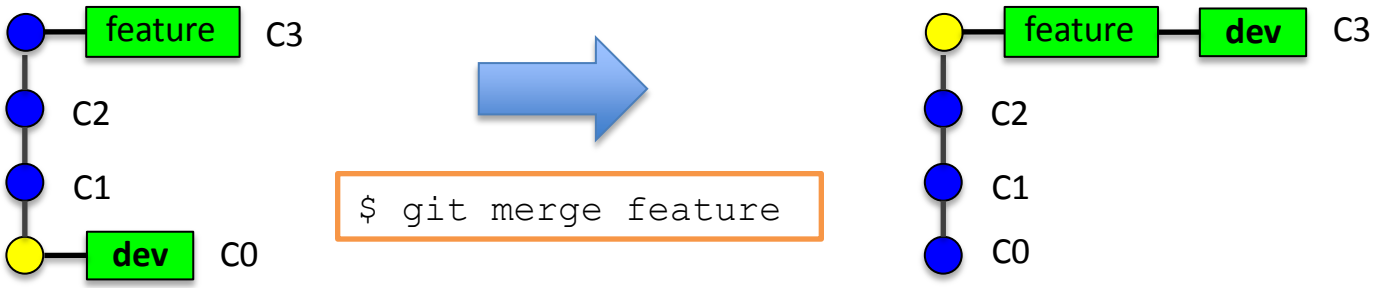
`merge` or `rebase` ?

G: So, sometimes you **do** want a “true merge”.

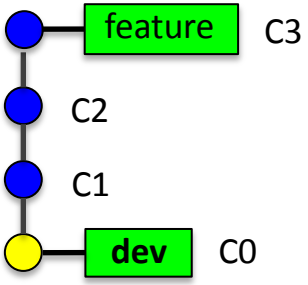
P: Yes, but that is what git merge does right ?

G: True… except when a fast-forward merge is possible.

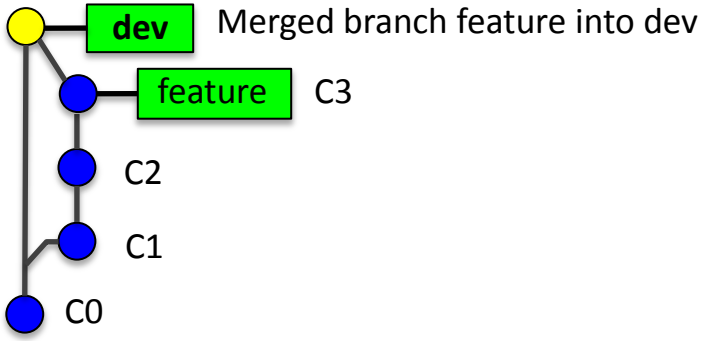
P: fast-forward merge ?



merge or rebase ?




```
$ git merge --no-ff feature
```



Prefer `fetch` to `pull`

The `pull` command is equivalent to a `fetch` followed by a `merge`.

So what you are really asking `git` to do when you pull is to ***merge*** your work with something you know nothing about (!)

To come around this problem, start by fetching what is new from the remote and have a look at it.

If what you really want is a merge, you can still do it. But this time, you will do it knowingly.

Interactive `rebase`

Interactive `rebase`

Remember what we did with `git commit --amend` ?

Now imagine you did the exact same thing but have already added a few commits on top of the faulty one.

You can not use `commit -amend` because it **only** allows to modify the commit that is **referenced by HEAD**.

This is when **interactive rebase** becomes handy.

Interactive `rebase`

Let's see what the command looks like :

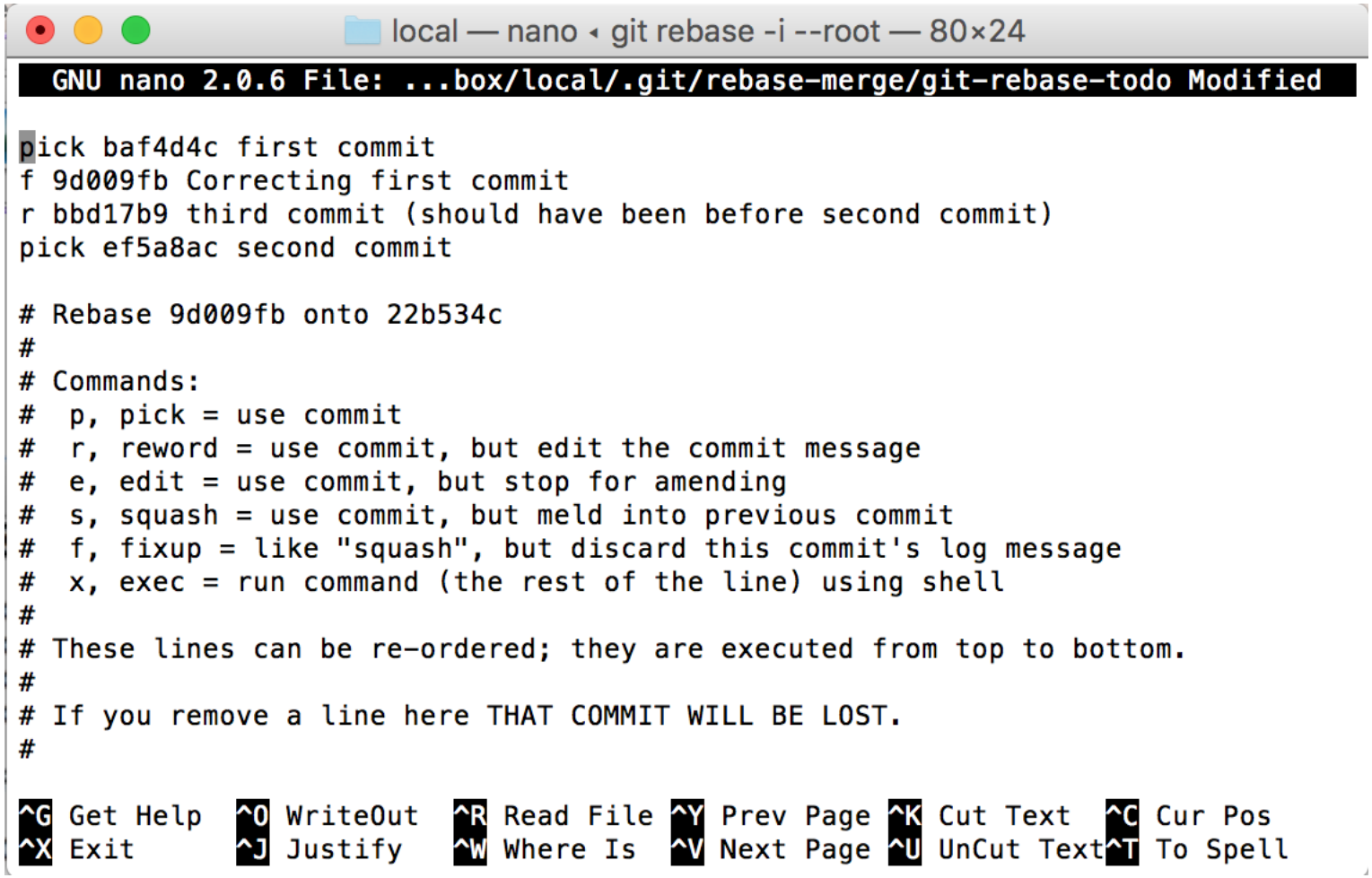
```
$ git rebase –interactive
# or
$ git rebase -i
```

What you're rebasing and onto what, follows the same rules as non-interactive rebase.

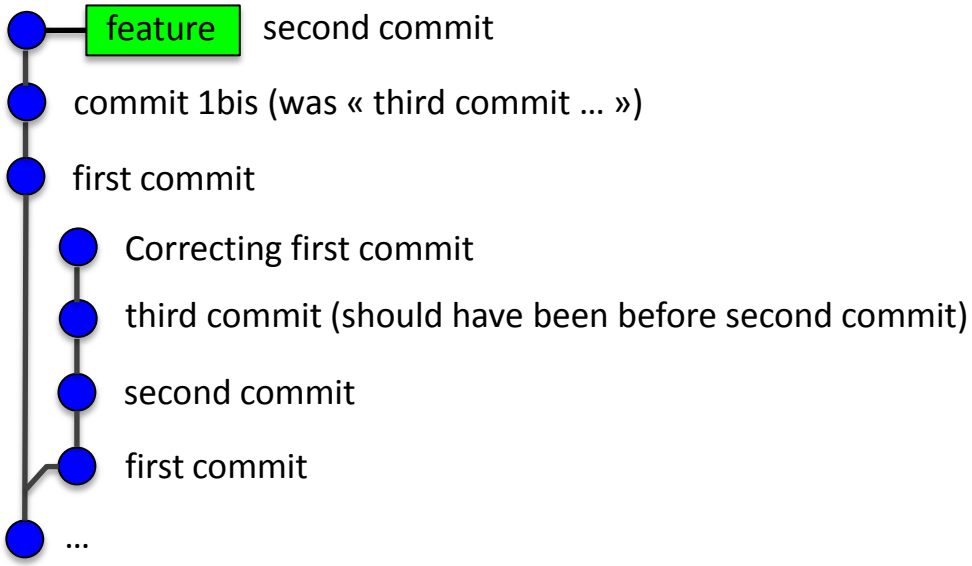
The difference is that you will get to decide what to do with each commit to be rebased instead of unconditionally "pick" them

Interactive **rebase**

Interactive `rebase`



Interactive `rebase`



6
Undoing things

Unstage

Unstaging an entire file is very easy, `git` tells you how to do it :

```
$ git status
  (use "git reset HEAD ..." to unstage)
$ git reset HEAD <file>
```

Do not want to have to remember this command ? Create an alias :

```
$ git config --global alias.unstage "reset HEAD"
$ git unstage
```

If you do not want to unstage the entire file but only some parts of it, the easiest solution is probably : `git gui`.
But you could also use : `git reset -p`

Unmodify a file

Again, `git` is kind enough to prompt you for actions you might want to do:

```
$ git status
  (use "git checkout -- <file>..." to discard changes …)
$ git co -- <file>
```

And again you can create an alias:

```
$ git config --global alias.unmod "checkout --"
$ git unmod <file>
```

If you do not want to unmodify a whole file but only some parts of it, the easiest solution is probably a ***difftool***.
But you could also use `git co -p`

Undo a commit

- If the commit has not been published yet

```
$ git rebase -i
...
```

This way, you can thoroughly **remove the faulty commit from the history**

- If the commit has been published

```
$ git revert <commit-to-undo>
```

This will **create a new commit** whose diff is the inverse of that of the commit to undo

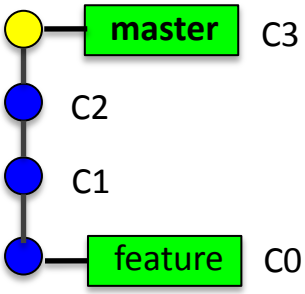
git reset

`git reset` can become very handy when things are beginning to get awry.

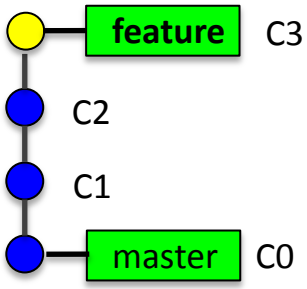
It allows you to make a branch point anywhere you want.

Let's say you have committed stuff in **master** when what you really wanted was to commit them in ***feature*** :

What you have



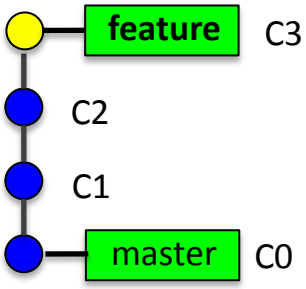
What you want



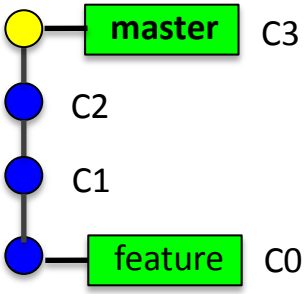
git reset

```
$ git co feature
```

What you want

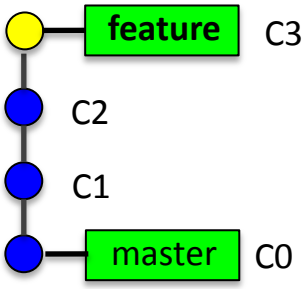


What you have



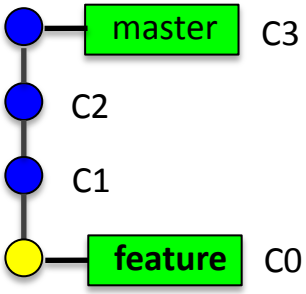
git reset

What you want



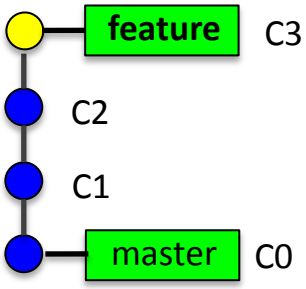

```
$ git co feature
Switched to branch 'feature'

$ # ?
```

What you have



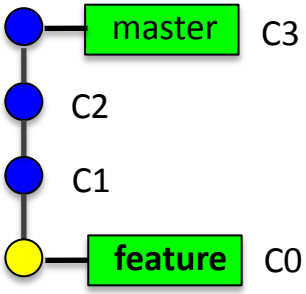
git reset

What you want



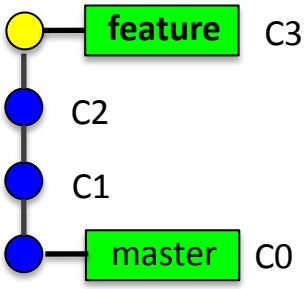

```
$ git co feature
Switched to branch 'feature'

$ git (merge | rebase | reset --hard) master
```

What you have



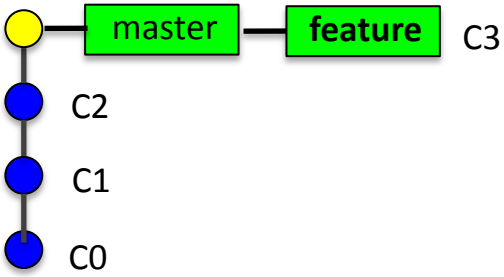
git reset

What you want



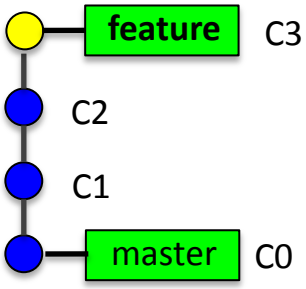

```
$ git co feature
Switched to branch 'feature'

$ git (merge | rebase | reset --hard) master
...

$
```

What you have



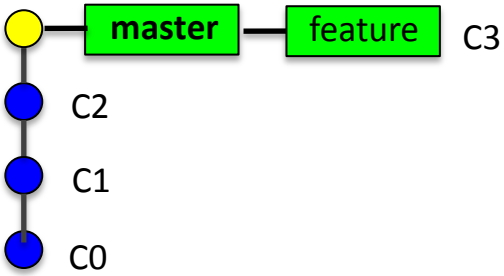
git reset

What you want



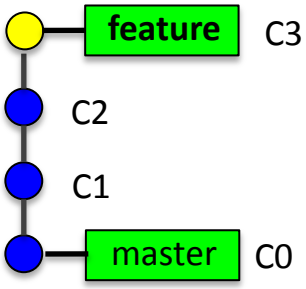

```
$ git co feature
Switched to branch 'feature'

$ git (merge | rebase | reset --hard) master
...

$ git co master
Switched to branch 'master'

$ # ?
```

What you have



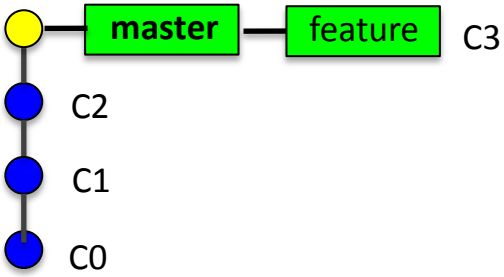
git reset

What you want



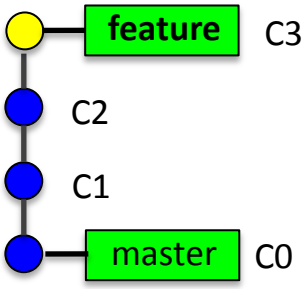

```
$ git co feature
Switched to branch 'feature'

$ git (merge | rebase | reset --hard) master
...

$ git co master
Switched to branch 'master'

$ git reset --hard bdfa # ref-to-C0
```

What you have



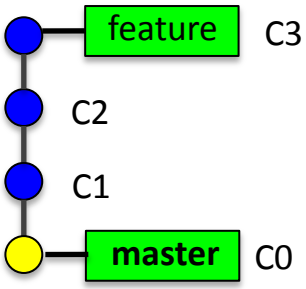
git reset

What you want



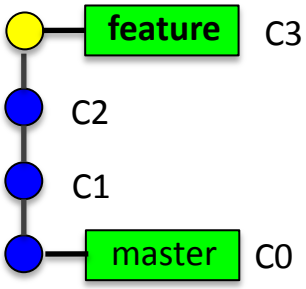

```
$ git co feature
Switched to branch 'feature'

$ git (merge | rebase | reset --hard) master
...

$ git co master
Switched to branch 'master'

$ git reset --hard bdfa # ref-to-C0
HEAD is now at bdfa5 C0

$
```

What you have



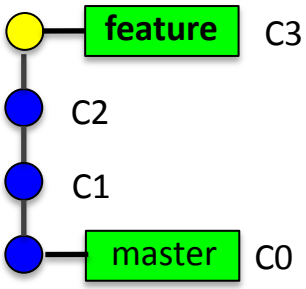
git reset

What you want



BINGO !

What you have




```
$ git co feature
Switched to branch 'feature'

$ git (merge | rebase | reset --hard) master
...

$ git co master
Switched to branch 'master'

$ git reset --hard bdfa5a # ref-to-C0
HEAD is now at bdfa5ab C0

$ git co feature
$
```

`soft`, `mixed` or `hard` ?

There are 3 main options to `git reset` :

- `soft`, `mixed` and `hard`

There are also 3 steps that can be done :

1. Move the current branch
2. Update the staging area
3. Update the working directory

Here is how to use the main options :

`--soft` does only 1.

`--mixed` does 1. and 2.

`--hard` goes all the way to 3.

Cheat sheet

	head	index	work dir	wd safe
Commit Level				
`reset --soft [commit]`	REF	NO	NO	YES
`reset [commit]`	REF	YES	NO	YES
`reset --hard [commit]`	REF	YES	YES	NO
`checkout [commit]`	HEAD	YES	YES	YES
File Level				
`reset (commit) [file]`	NO	YES	NO	YES
`checkout (commit) [file]`	NO	YES	YES	NO

More undoing …

`git-filter-branch` is a very powerful tool

- it allows you to apply “filters” on each revision of a branch

This can be useful *e.g.* when you’ve commited a file that should never have been added (binary file, confidential information, …)

```
$ git filter-branch –tree-filter 'rm -f ’ HEAD
```

An alternative to `git-filter-branch` for “cleansing bad data out of a git repo” is worth mentioning : BFG Repo-Cleaner.
See : `https://rtyley.github.io/bfgrepo-cleaner/`

Time for Practical Work !

- Document reference : http://sed.inrialpes.fr/advancedgit-tuto

➢ To Do : **Section 3**

7
Managing remotes

About remotes

- Remotes are repositories other than the local one with which you may want to synchronize.

- `origin` is the default name of the default remote

- Add a remote:
  ```
  git remote add other git://[...]
  ```

- Fetch remote branches:
  ```
  git fetch other
  ```

- Push master to remotes/other/master
  ```
  git push other master
  ```

Annoying things

- Delete a remote branch :

```
git push origin --delete featA
```

- Push a tag :

```
git push origin mytag
```

or

```
git push --tags
```

Multiple remotes, pull-requests

Many projects on GitHub/Lab allow third-party contributions by a mechanism called **pull-request**.
This mechanism can also be used internally for code review.

Multiple remotes can be very useful in the context of workflows including pull-requests but also in other cases (`git annex`, migrations, …).

8

Other interesting things

git svn

Allows to work locally with a `git` repo and sync it with an `svn` remote.
As a corollary, it allows to migrate from `svn` to git

```
# Create a file for author mapping
echo "dpa = David Parsons <david.parsons@inria.fr>" > authors

# Clone an existing svn repo
$ git svn clone <url> --authors-file=authors local-git-repo-name

$ git svn rebase
# gets new revs from svn repo and rebase your work (if any) on top of it

# "Push" your changes
$ git svn dcommit
```

Other options of `git svn` clone I have found useful :

- `--stdlayout` if the svn repo follows the trunk/branches/tags layout
- `--ignore-paths` / `--include-paths`
- `--authors-prog`

git subrepo

Alternatives :

- submodules
- subtrees

`git subrepo` allows you to work with embedded git repositories

```
# Clone an existing repo as a subrepo (in a subdirectory)
$ git subrepo clone <url> <subdir>

# Create an embedded git repo
$ git subrepo init <subdir>

# Pull from upstream
$ git subrepo pull <subdir>

# Push to upstream
$ git subrepo push <subdir>
```

git annex

Alternatives :

- `lfs`

`git annex` provides an interesting solution to version large files. It basically handles symlinks and provides tools to manage the actual files behind the links.

```
# Prepare an existing git repo for git annex
$ git annex init

$ git annex add
$ git commit
$ # => commits a symlink and stores the file in .git/annex
```

`git annex` can be used with a wide variety of types of remote storage spaces (special remotes).

Hooks

Hooks are custom scripts that can be automatically triggered when certain important actions occur.

- Client-side hooks :

  `pre-commit, prepare-commit-msg, commit-msg, post-commit, pre-rebase, post-rewrite, post-checkout, post-merge, pre-push, pre-auto-gc`

- Server-side hooks :

  `pre-receive, update, post-receive`

On platforms such as forges or GitHubs/Labs, some (many ?) predefined hooks can be set up in a matter of minutes.

Hooks can be used e.g. to send notifications, run tests prior to commits, enforce any kind of policies, …

rerere

One of the most annoying things that a git-user has to do is to resolve conflicts.

Even more annoying would be to have to resolve the same conflict several times.
Sadly, this happens. Mostly when relying heavily on rebase (?)

`rerere` helps you avoiding this situation by **re**using **re**corded **re**solutions .

```
# Configure rerere functionnality for git
git config --global rerere.enabled true
```

Thank you

Antenne INRIA Lyon la Doua

www.inria.fr

Time for Practical Work !

- Document reference : http://sed.inrialpes.fr/advancedgit-tuto

➢ To Do : **Section 4**