Tools : getting started with git

version control systems

- imagine a large project with multiple contributors. This could be code development, code maintenance, but also writing a joint paper

- errors and conflicts are unavoidable. Sometimes you want to go back to a previous (say more correct) version and discard recent changes. Or you need to go back to an old version in order to reproduce a bug that was reported in an open ticket.

- you may want to try out an idea on your own of which you are not sure it will make it to the main branch. You want to do this without disturbing the work of your partners.

- people work simultaneously on different parts of the project. How to solve conflicting overlaps?

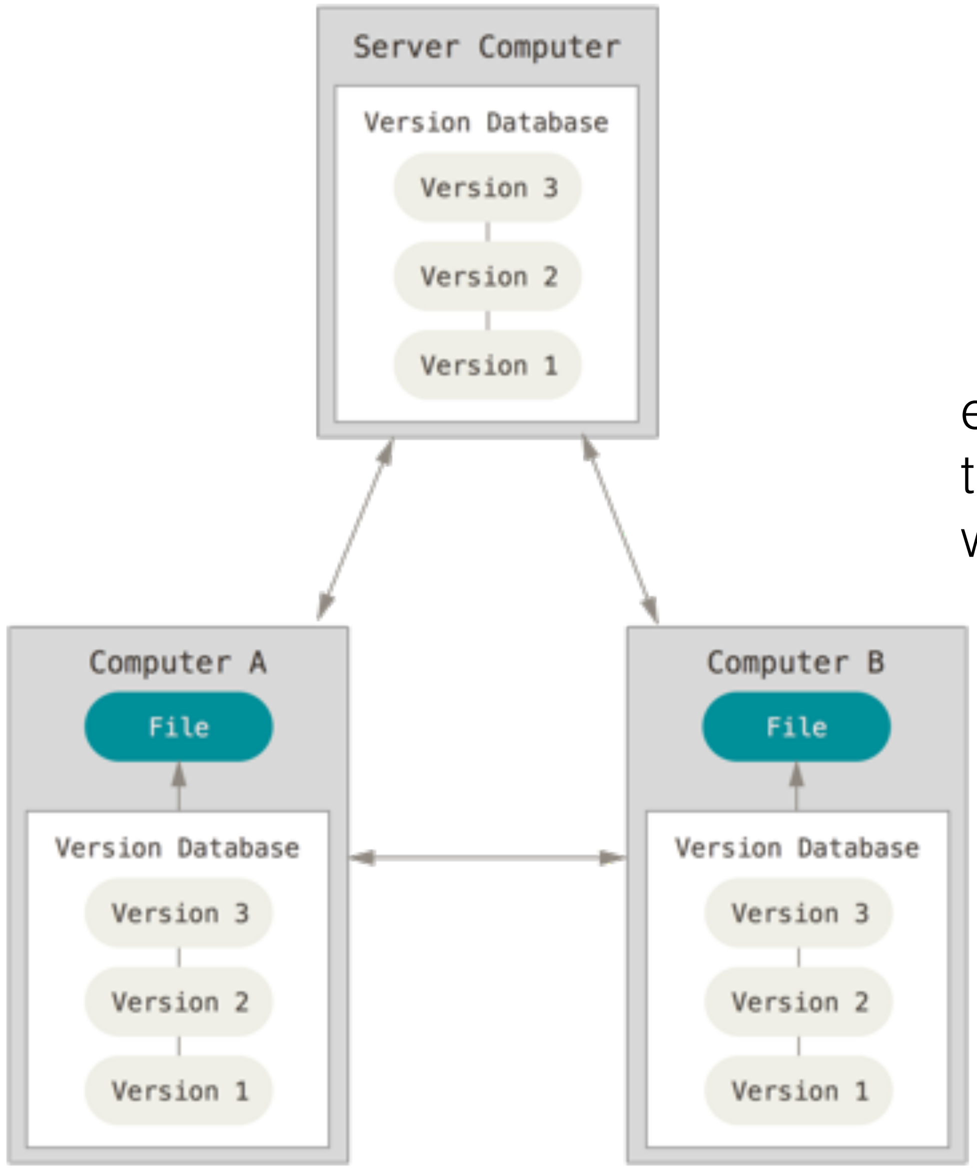
- people work simultaneously on the same parts of the project. How to solve conflicting overlaps?

Git

The answer is a version control system (VCS), such as svn or git.

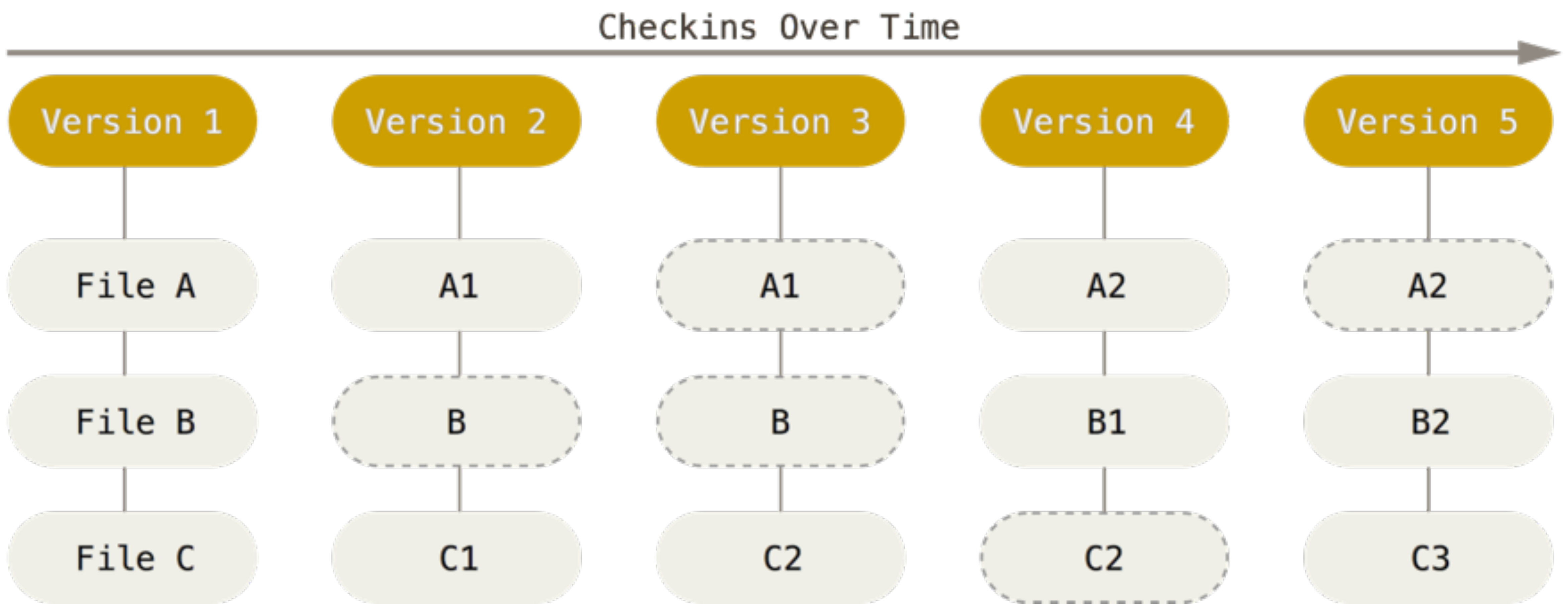
we follow the documentation on https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

git is based on a distributed version control system:



every clone has a full backup of all data. this is very different from svn or systems with a central repository only

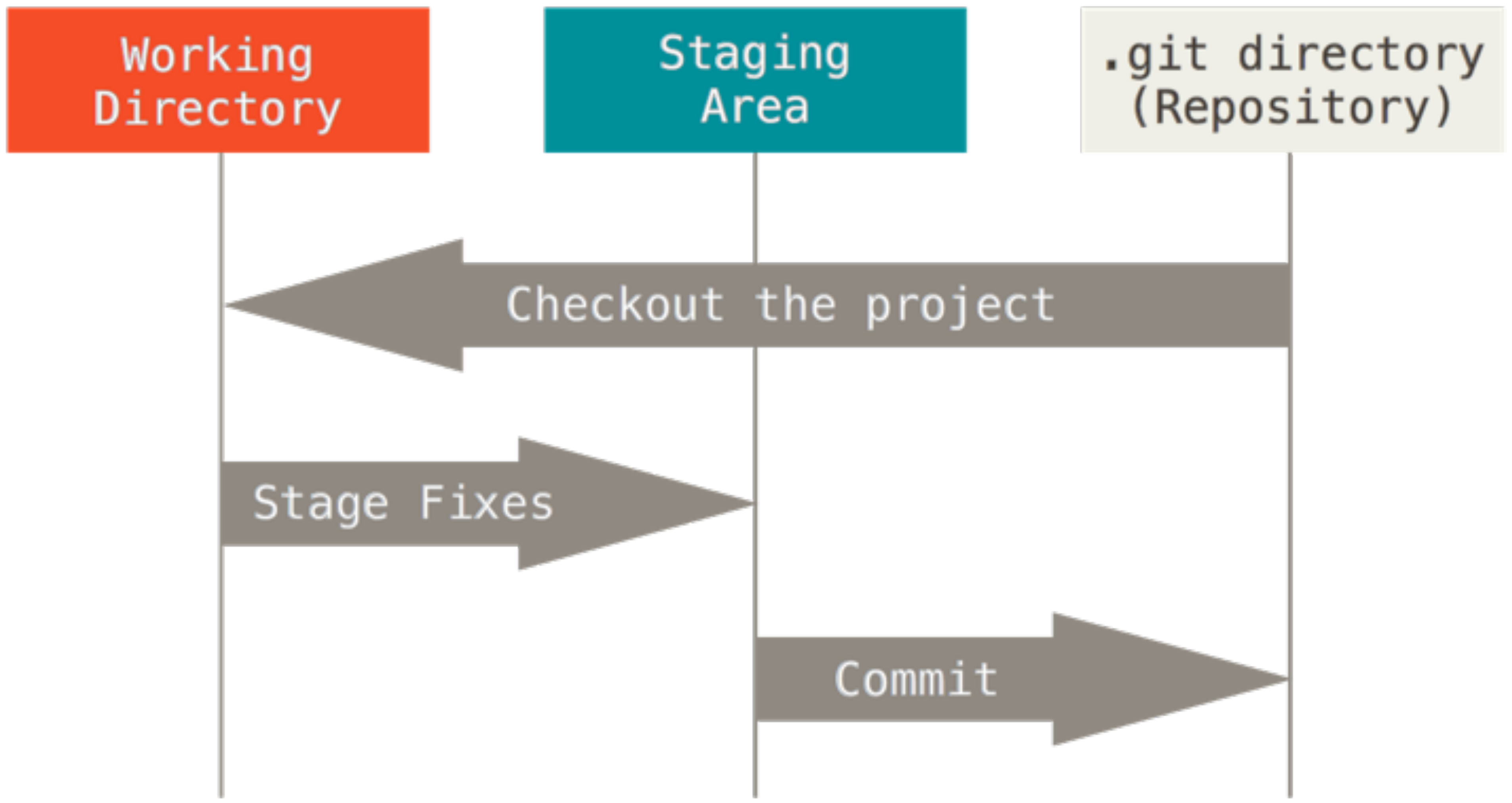
Git



Every time you commit, or save the state of your project in Git, it basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot. To be efficient, if files have not changed, Git doesn't store the file again, just a link to the previous identical file it has already stored. Git thinks about its data more like a **stream of snapshots**.

Most operations in Git only need local files and resources to operate – generally no information is needed from another computer on your network

Git



Git has three main states that your files can reside in: committed, modified, and staged. Committed means that the data is safely stored in your local database. Modified means that you have changed the file but have not committed it to your database yet. Staged means that you have marked a modified file in its current version to go into your next commit snapshot.

The basic Git workflow goes something like this:

1. You modify files in your working directory.
2. You stage the files, adding snapshots of them to your staging area.
3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

Git

First thing to do when you install git, is configure your user name and email:

```
$ git config --global user.name "Lode Pollet"
$ git config --global user.email lode.Pollet@lmu.de
```

with the option —global this will be set for all git projects. Do not use —global if you want to use another name and or email for a specific project

you can check your settings as follows:

```
$ git config --list
credential.helper=osxkeychain
user.name=Lode Pollet
user.email=Lode.Pollet@lmu.de
```

if you need help:

```
$ git help config
```

(or any other git command instead of config)

Git

- there are 2 ways to start working on a new project:

a new empty project

```
$ git init
```

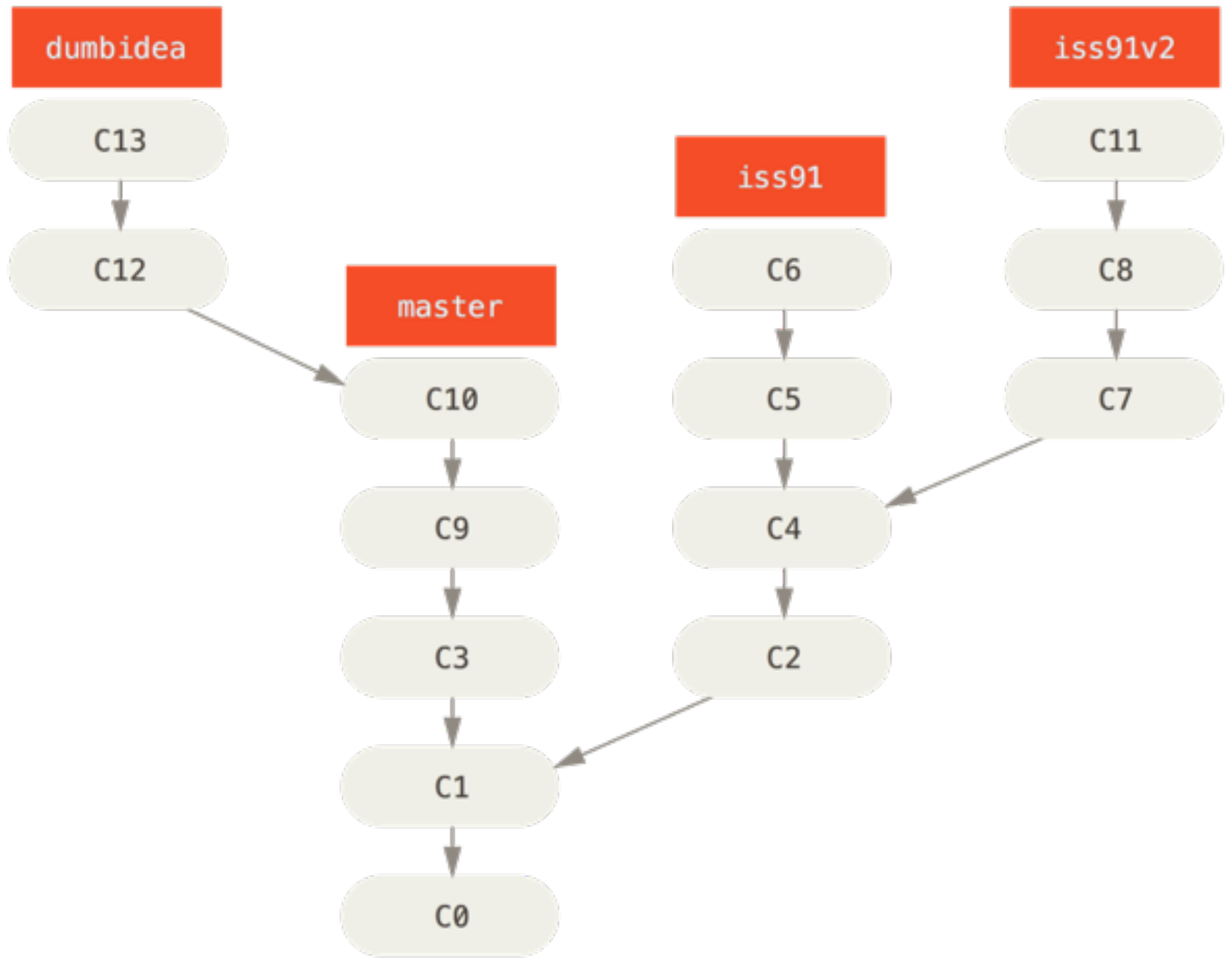
cloning from an existing repository (with login using ssh)

```
$ git clone ssh://ACCOUNTNAME@git-repository
```

or anonymously using https:

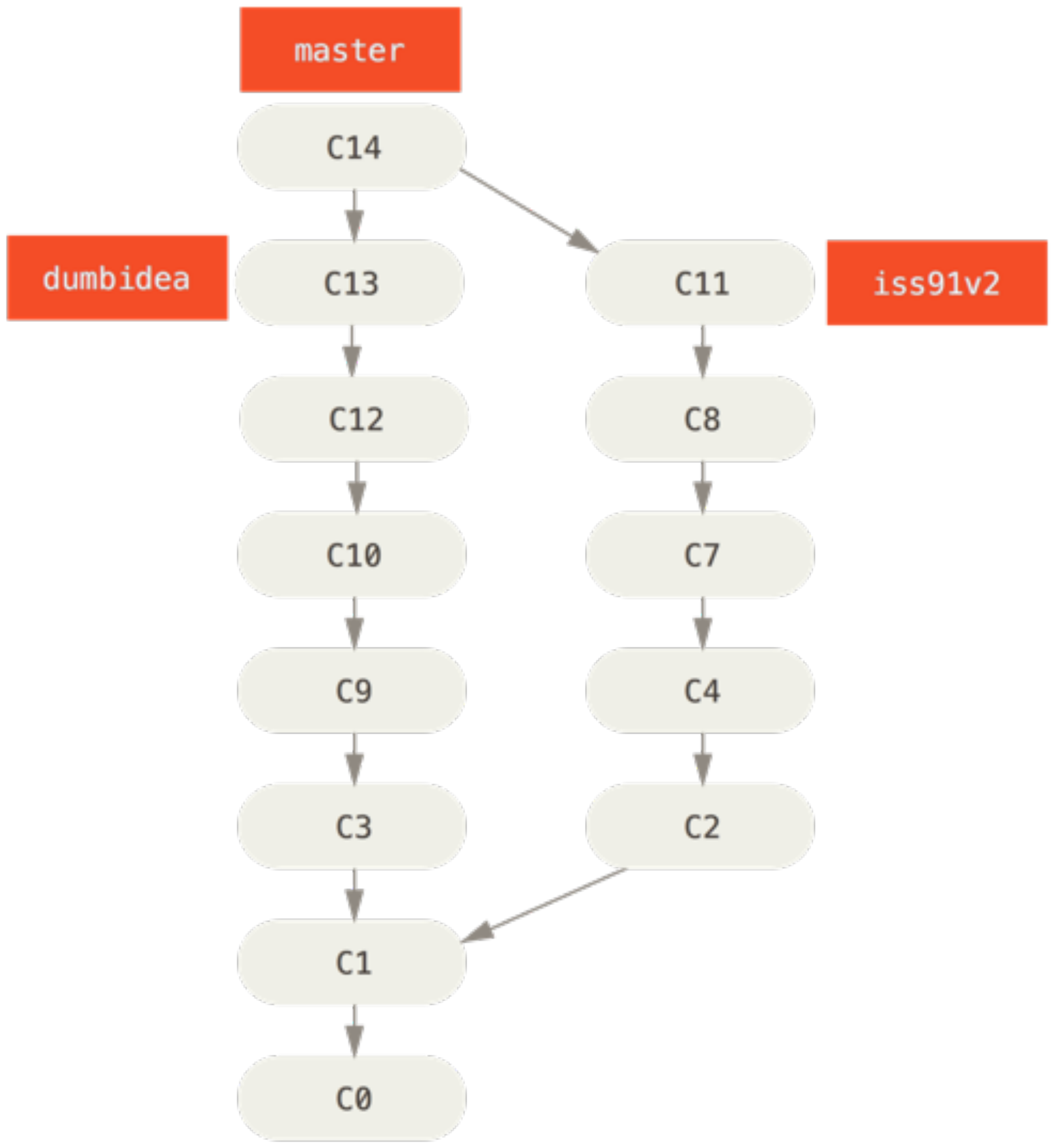
```
$ git clone https://github.com/libgit2/libgit2
```

Git branching model



say main project line is on master with some developments, then you start working on iss91 within a separate branch forked from master. In the mean time there are further developments on the master branch, and somebody else takes up a special case of iss91, namely iss91v2 whereas you continue working on iss91. A colleague has a risky idea to try out and starts working on it on a separate branch (dumbidea)

Git branching model



the dumbidea turned out to be a flash of genius and it is decided to keep it, merging it with the master branch. iss91v2 supersedes the developments in iss91 (because it is better), and the commits C5 and C6 can be discarded. Finally, iss91v2 is brought (merged) into the master line.

Git

Let us start a new git project. We *init*ialize, and create a file “hello”, which we want to be tracked (= *add*) and then *commit* it:

```
$ echo “hello” > hello.txt
```

```
$ git init
```

```
Initialized empty Git repository in /Users/Lode.Pollet/Lectures/Programming2016/Programs/Lec_git/.git/
```

```
$ git add hello.txt
$ git commit -m 'initial project version'
```

```
[master (root-commit) d3680cd] initial project version
 1 file changed, 1 insertion(+)
 create mode 100644 hello.txt
```

This is what the directory looks like. Note the hidden directory .git

```
$ ls -al
total 8
drwxr-xr-x   4 Lode.Pollet  staff  136 17 Apr 17:43 ./
drwxr-xr-x  18 Lode.Pollet  staff  612 17 Apr 16:45 ../
drwxr-xr-x  13 Lode.Pollet  staff  442 17 Apr 17:45 .git/
-rw-r--r--   1 Lode.Pollet  staff    6 17 Apr 17:43 hello.txt
```

At any time we can check the status of the files by running *git status*

```
$ git status
On branch master
nothing to commit, working directory clean
```

Git

Let us add 2 more files to be tracked in the project, again demonstrating *git status*

```
th-ea-lswtb02:Lec_git Lode.Pollet$ echo "foo" > file1.txt
th-ea-lswtb02:Lec_git Lode.Pollet$ echo "bar" > file2.txt
th-ea-lswtb02:Lec_git Lode.Pollet$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    file1.txt
    file2.txt

nothing added to commit but untracked files present (use "git add" to track)
```

git notices that there are 2 untracked files in the directory; if we add them then we get the following after running *git status*:

```
th-ea-lswtb02:Lec_git Lode.Pollet$ git add file*
th-ea-lswtb02:Lec_git Lode.Pollet$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   file1.txt
    new file:   file2.txt
```

by editing the file *.gitignore* we can specify files that we do not want to track (say, .o files, executables, …)

Git

if we modify a file that is being tracked, *git status* notices the changes:

```
$ echo "foofoo" >> file1.txt
th-ea-lswtb02:Lec_git Lode.Pollet$ more file1.txt
foo
foofoo
th-ea-lswtb02:Lec_git Lode.Pollet$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

	new file:   file1.txt
	new file:   file2.txt

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

	modified:   file1.txt
```

this is quite strange: file1.txt is both staged and modified (unstaged). It turns out that when running *git add* git stages a file exactly as it is when you run *git add*. We want however the recent changes of file1.txt to be part of the next *commit*.

Therefore, let us now *add* all files to the staged area, check again with *git status*, and then *commit* :

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

	new file:   file1.txt
	new file:   file2.txt


$ git commit -m 'second version'
[master 99a1f45] second version
 2 files changed, 3 insertions(+)
 create mode 100644 file1.txt
 create mode 100644 file2.txt
```

Git

the history can be read with *git log* :

```
$ git log
commit 99a1f452443e724691eabde1aa639eca1e3c3fc7
Author: Lode Pollet <Lode.Pollet@lmu.de>
Date:   Sun Apr 17 17:53:54 2016 +0200

    second version

commit d3680cdeebdf18591b990a8600abb97e9c3c0b70
Author: Lode Pollet <Lode.Pollet@lmu.de>
Date:   Sun Apr 17 17:44:50 2016 +0200

    initial project version
```

the directory now looks as follows:

```
$ ls -al
total 24
drwxr-xr-x   6 Lode.Pollet  staff  204 17 Apr 17:47 ./
drwxr-xr-x  18 Lode.Pollet  staff  612 17 Apr 16:45 ../
drwxr-xr-x  13 Lode.Pollet  staff  442 17 Apr 17:53 .git/
-rw-r--r--   1 Lode.Pollet  staff   11 17 Apr 17:48 file1.txt
-rw-r--r--   1 Lode.Pollet  staff    4 17 Apr 17:47 file2.txt
-rw-r--r--   1 Lode.Pollet  staff    6 17 Apr 17:43 hello.txt
```

```
$ ls -l .git/
total 40
-rw-r--r--   1 Lode.Pollet  staff   15 17 Apr 17:53 COMMIT_EDITMSG
-rw-r--r--   1 Lode.Pollet  staff   23 17 Apr 17:43 HEAD
drwxr-xr-x   2 Lode.Pollet  staff   68 17 Apr 17:43 branches/
-rw-r--r--   1 Lode.Pollet  staff  137 17 Apr 17:43 config
-rw-r--r--   1 Lode.Pollet  staff   73 17 Apr 17:43 description
drwxr-xr-x  11 Lode.Pollet  staff  374 17 Apr 17:43 hooks/
-rw-r--r--   1 Lode.Pollet  staff  281 17 Apr 17:53 index
drwxr-xr-x   3 Lode.Pollet  staff  102 17 Apr 17:43 info/
drwxr-xr-x   4 Lode.Pollet  staff  136 17 Apr 17:44 logs/
drwxr-xr-x  12 Lode.Pollet  staff  408 17 Apr 17:53 objects/
drwxr-xr-x   4 Lode.Pollet  staff  136 17 Apr 17:43 refs/
```

Git

suppose we no longer want to track file2.txt :

```
$ rm file2.txt


$ git rm file2.txt


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

    deleted:    file2.txt

th-ea-lswtb02:Lec_git Lode.Pollet$ git commit -m "removed file2.txt"
[master 6e712c7] removed file2.txt
 1 file changed, 1 deletion(-)
 delete mode 100644 file2.txt
```

let us modify hello.txt :

```
$ echo "bonjour" >> hello.txt
```

as before git status tells us about the changes:

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

    modified:   hello.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

to see what you have changed, but not yet staged, type

```
$ git diff
diff --git a/hello.txt b/hello.txt
index ce01362..65038b4 100644
--- a/hello.txt
+++ b/hello.txt
@@ -1 +1,2 @@
 hello
+bonjour
```

Git

to see what you have changed, but not yet committed, type git diff — staged

```
$ git add hello.txt

$ git diff


$ git diff --staged
diff --git a/hello.txt b/hello.txt
index ce01362..65038b4 100644
--- a/hello.txt
+++ b/hello.txt
@@ -1 +1,2 @@
 hello
+bonjour
```

now we can *commit* and check the *log*

```
$ git commit -m “French translation”

$ git log
commit 299da21105c778aab89537ec6f64be90398bec81
Author: Lode Pollet <Lode.Pollet@lmu.de>
Date:   Sun Apr 17 18:15:09 2016 +0200

    French translation

commit 6e712c7b1e2ace22faec99db2b861211a02627c9
Author: Lode Pollet <Lode.Pollet@lmu.de>
Date:   Sun Apr 17 18:03:06 2016 +0200

    removed file2.txt

commit 99a1f452443e724691eabde1aa639eca1e3c3fc7
Author: Lode Pollet <Lode.Pollet@lmu.de>
Date:   Sun Apr 17 17:53:54 2016 +0200

    second version

commit d3680cdeebdf18591b990a8600abb97e9c3c0b70
Author: Lode Pollet <Lode.Pollet@lmu.de>
Date:   Sun Apr 17 17:44:50 2016 +0200

    initial project version
```

Git

git log has many options. *-p* shows the difference introduced in each commit; *-p -1* restricts to the last commit:

```
git log -p -1
commit 299da21105c778aab89537ec6f64be90398bec81
Author: Lode Pollet <Lode.Pollet@lmu.de>
Date:   Sun Apr 17 18:15:09 2016 +0200

    French translation

diff --git a/hello.txt b/hello.txt
index ce01362..65038b4 100644
--- a/hello.txt
+++ b/hello.txt
@@ -1 +1,2 @@
 hello
+bonjour
```

if you want to see some abbreviated statistics:

```
$ git log --stat
commit 299da21105c778aab89537ec6f64be90398bec81
Author: Lode Pollet <Lode.Pollet@lmu.de>
Date:   Sun Apr 17 18:15:09 2016 +0200

    French translation

 hello.txt | 1 +
 1 file changed, 1 insertion(+)


(etc)
```

prettier looking output can be obtained as follows:

```
$ git log --pretty=oneline
299da21105c778aab89537ec6f64be90398bec81 French translation
6e712c7b1e2ace22faec99db2b861211a02627c9 removed file2.txt
99a1f452443e724691eabde1aa639eca1e3c3fc7 second version
d3680cdeebdf18591b990a8600abb97e9c3c0b70 initial project version
```

the most useful is probably git log —pretty=format:"%h - %an, %ar : %s”

```
6e712c7 - Lode Pollet, 2 weeks ago : removed file2.txt
99a1f45 - Lode Pollet, 2 weeks ago : second version
d3680cd - Lode Pollet, 2 weeks ago : initial project version
```

Git

undoing things:

suppose we committed too early and forgot to add a file:

```
$ echo “depressing” > rainyday.txt


$ git add rainyday.txt

$ git commit --amend

$ git log
commit d50ed7fabab638364e63e568dd99f5e1ee8ef5cf
Author: Lode Pollet <Lode.Pollet@lmu.de>
Date:   Sun Apr 17 18:15:09 2016 +0200

    French translation on a rainy day
```

unstaging a staged file: follow the advise of *git status*

```
$ vi README

$ more README
This is a git tutorial

$ git add README

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

    new file:   README


$ git reset HEAD README

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

    README

nothing added to commit but untracked files present (use "git add" to track)
```

Git

unmodifying a modified file: again follow the advice of git status

// README : foobar

```
th-ea-lswtb02:Lec_git Lode.Pollet$ vi README
th-ea-lswtb02:Lec_git Lode.Pollet$ vi README2
th-ea-lswtb02:Lec_git Lode.Pollet$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   README
    modified:   README2

no changes added to commit (use "git add" and/or "git commit -a")
th-ea-lswtb02:Lec_git Lode.Pollet$ git add README*
th-ea-lswtb02:Lec_git Lode.Pollet$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   README
    modified:   README2
```

as before, we run *git reset HEAD README*

```
th-ea-lswtb02:Lec_git Lode.Pollet$ git reset HEAD README
Unstaged changes after reset:
M   README
th-ea-lswtb02:Lec_git Lode.Pollet$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   README2

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

    modified:   README
```

Git

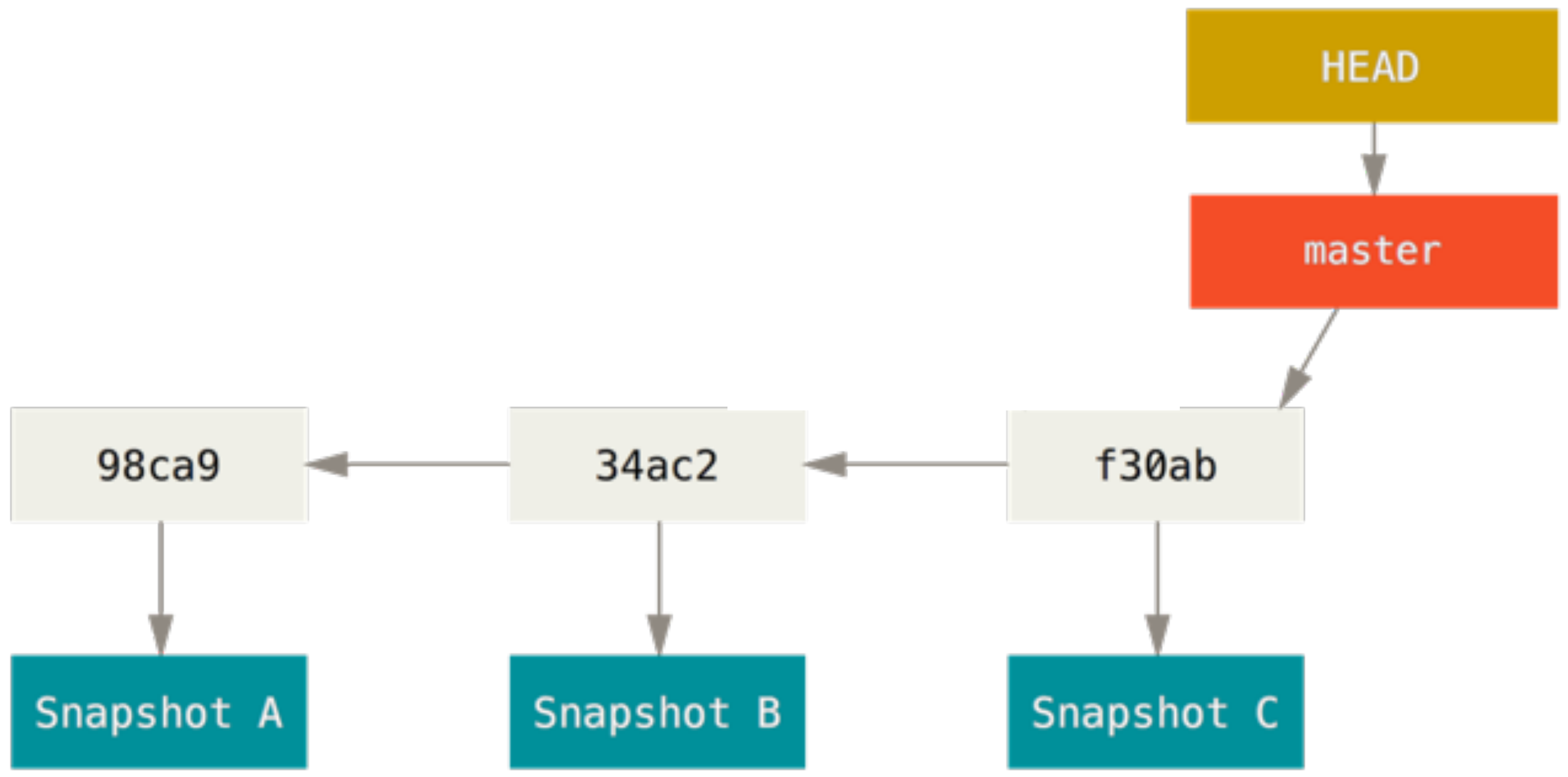
we realize we do not want to keep the changes to README:

```
th-ea-lswtb02:Lec_git Lode.Pollet$ git checkout -- README
th-ea-lswtb02:Lec_git Lode.Pollet$ more README
This is a git tutorial
th-ea-lswtb02:Lec_git Lode.Pollet$ more README2
another README file!
th-ea-lswtb02:Lec_git Lode.Pollet$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   README2

th-ea-lswtb02:Lec_git Lode.Pollet$ git commit -m "changed README2"
[master 58db01e] changed README2
 1 file changed, 1 insertion(+), 1 deletion(-)
th-ea-lswtb02:Lec_git Lode.Pollet$ git status
On branch master
nothing to commit, working directory clean
```

$\longleftarrow$ this is a dangerous command

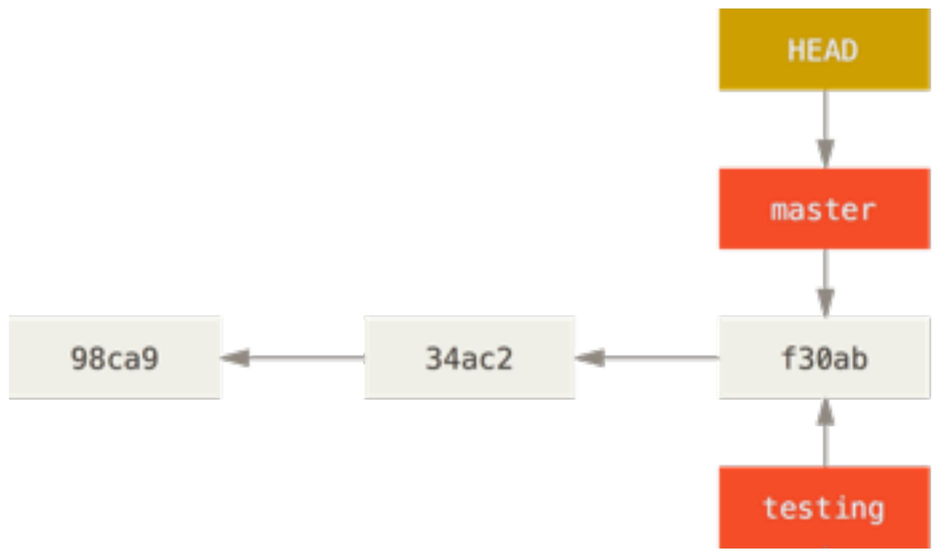
here you can lose information

Git branches



typical flow on the master branch : a pointer is stored at each commit to the previous commits; only snapshots are stored. Git knows on what branch we are by the movable pointer HEAD

Git branches



we can make a new branch by running

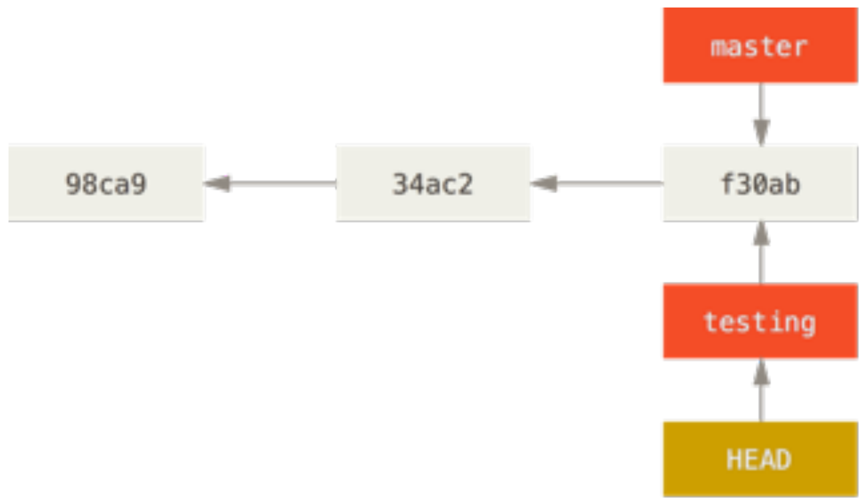
```
$ git branch testing
```

this just created a new branch; it did not change the HEAD pointer

by running

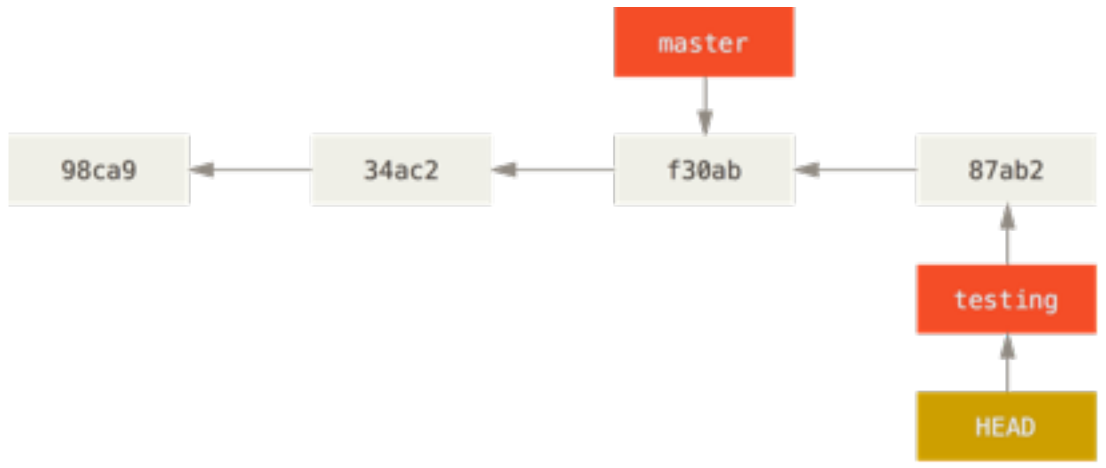
```
$ git log --oneline --decorate
```

you can see the new branch



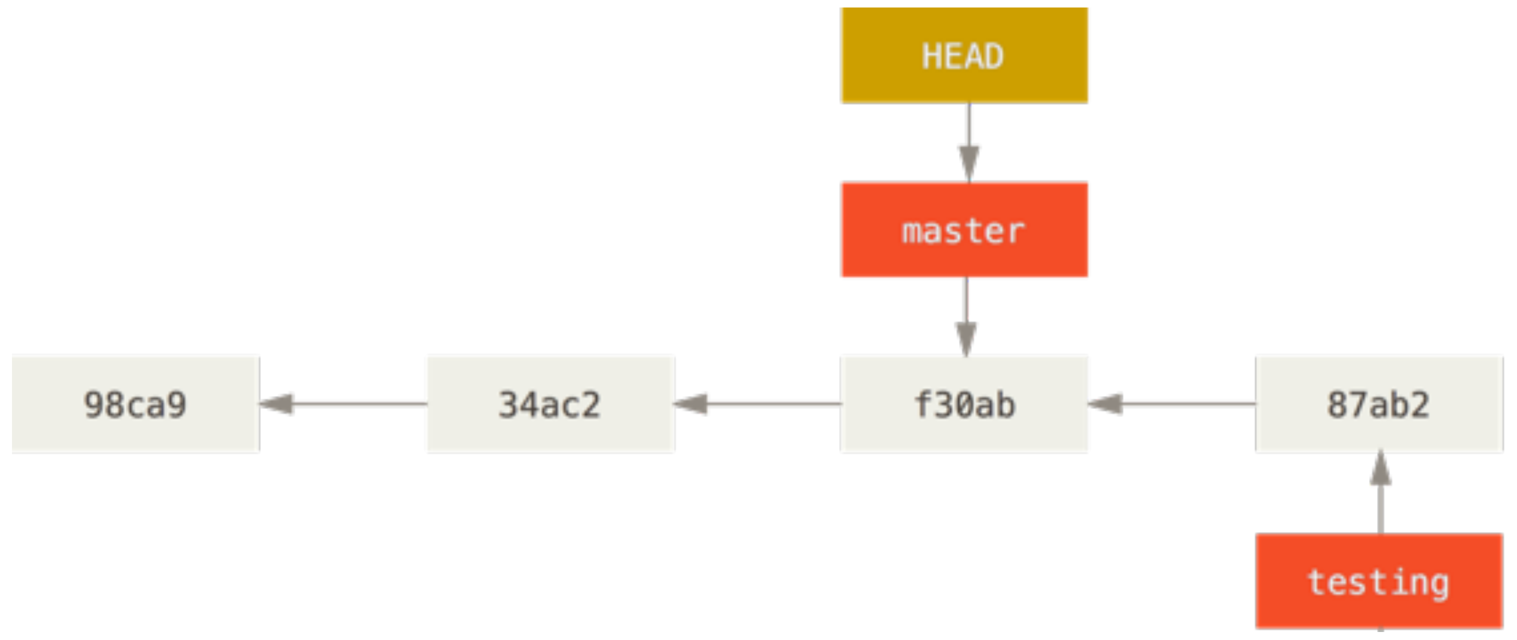
we can switch to new branch by running

```
$ git checkout testing
```



new commits update the testing branch and result in the figure shown on the left

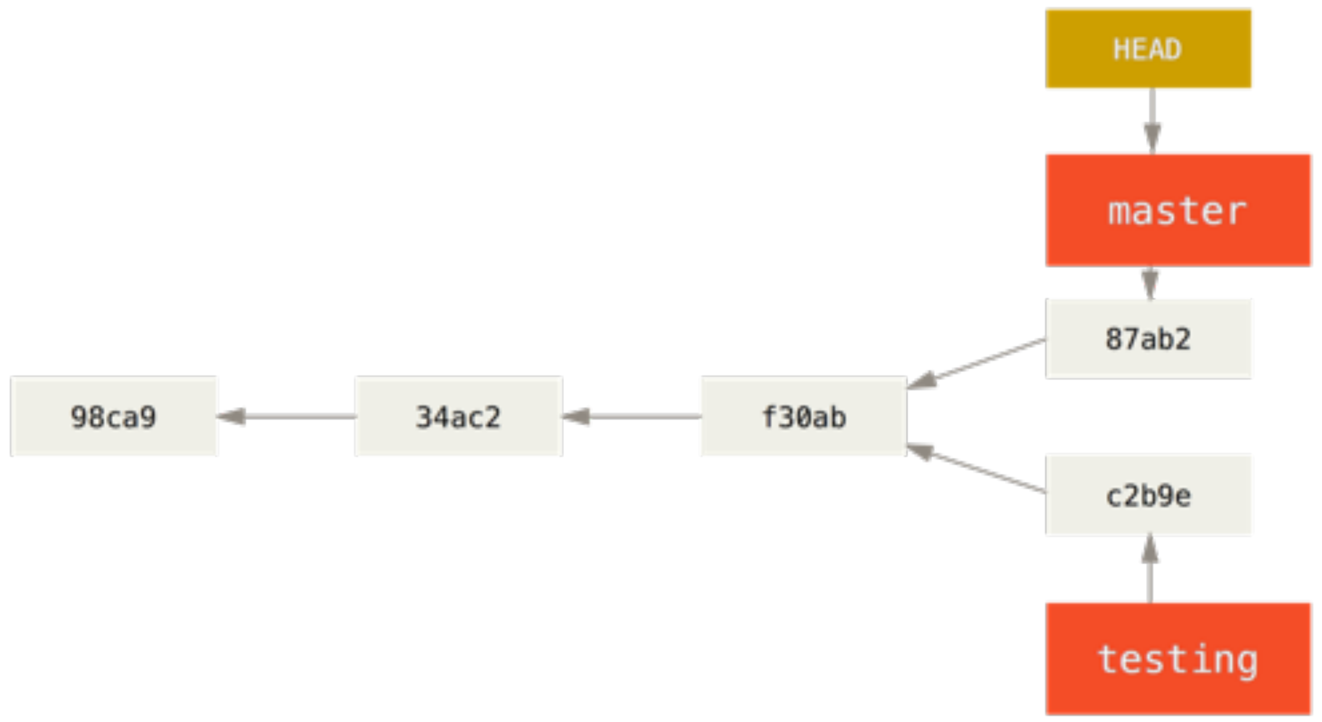
Git branches



you can go back to the master branch by

```
$ git checkout master
```

note that switching branches changes the files in your working directory!!!

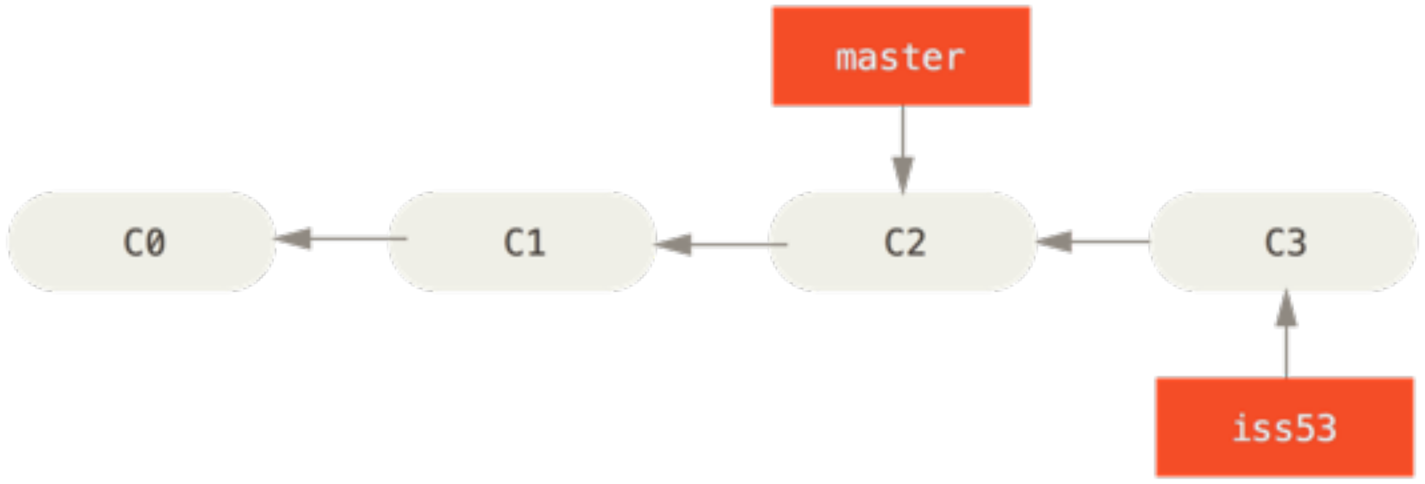


if we commit new changes on the master branch, we get a divergent history

you can see this easily with

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

branching and merging



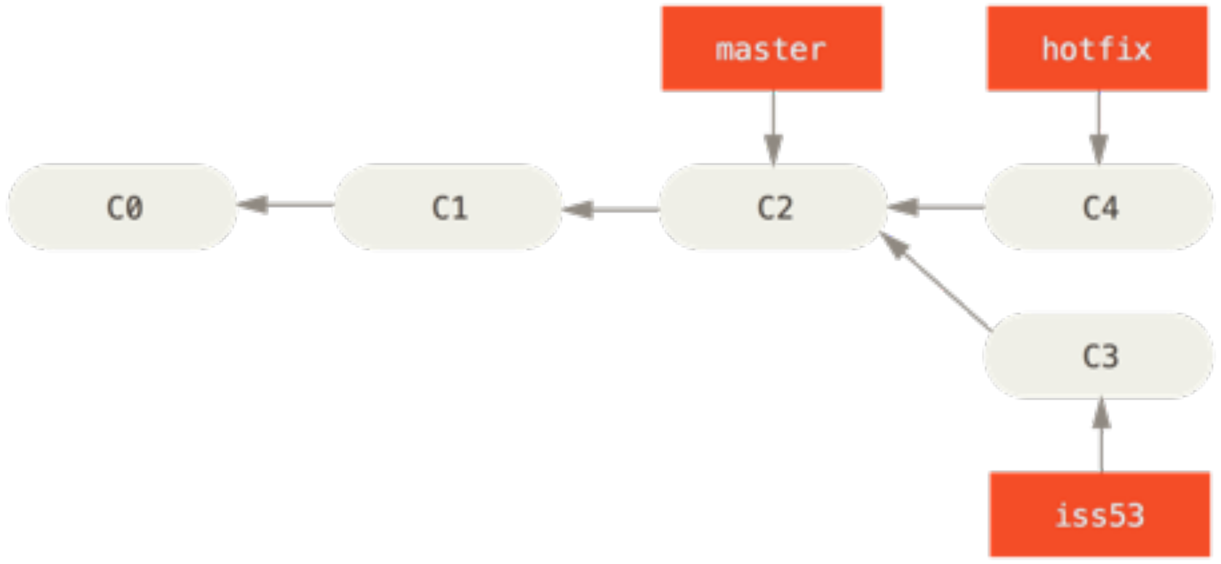
say we have a situation where you start working on a new branch iss53, which you created by

```
$ git branch iss53
$ git checkout iss53
```

this can alternatively be done in a single instruction:

```
$ git checkout -b iss53
```

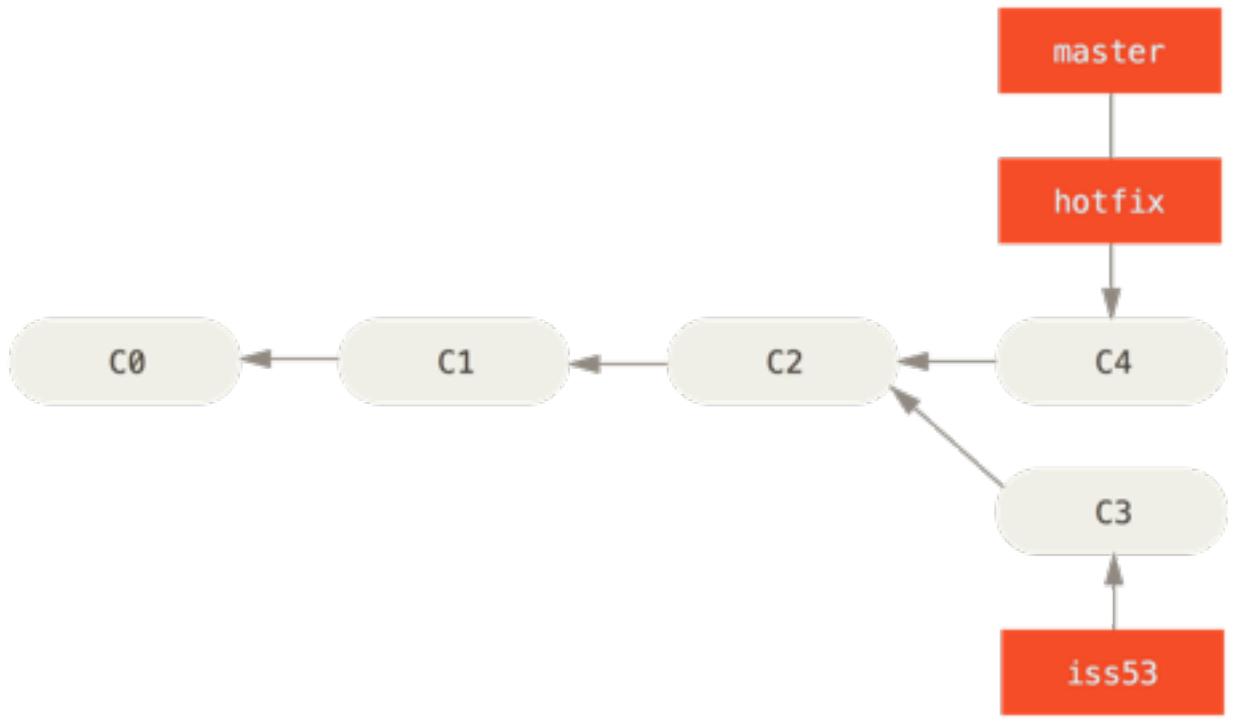
and you had a few commits in this branch.



an emergency has occurred requiring a hotfix.

```
$ git checkout master
$ git checkout -b hotfix
…
$ git commit -a -m “fixed”
```

branching and merging



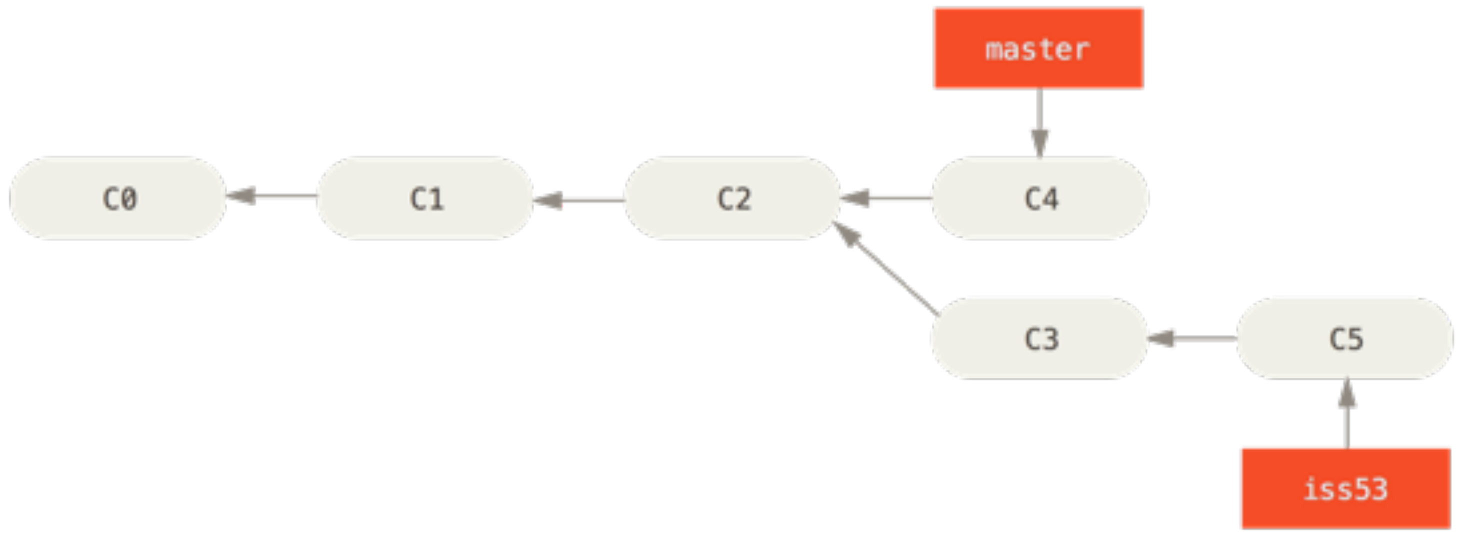
it is time to bring the master up-to-date with the hotfix:

```
$ git checkout master
$ git merge hotfix
```

this will lead to a fastforward of the master from C2 to C4.

we can now delete the hotfix branch:

```
$ git branch -d hotfix
```



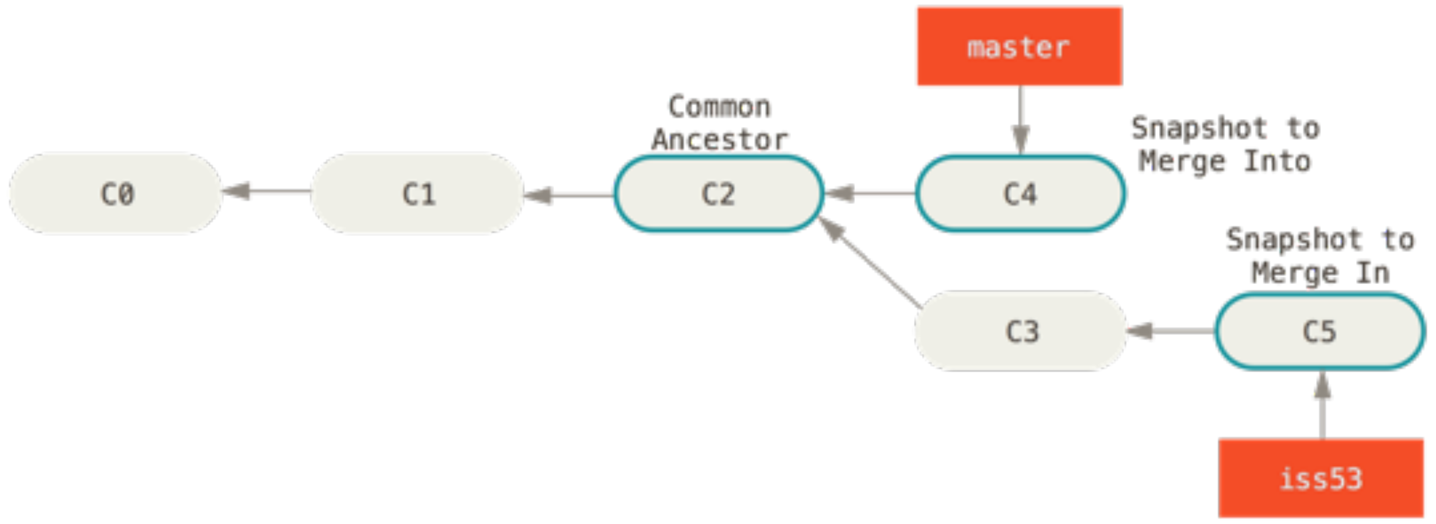
let us go back to the iss53 we were working on and commit recent work

```
$ git checkout iss53
$ git commit -a -m “recent work”
```

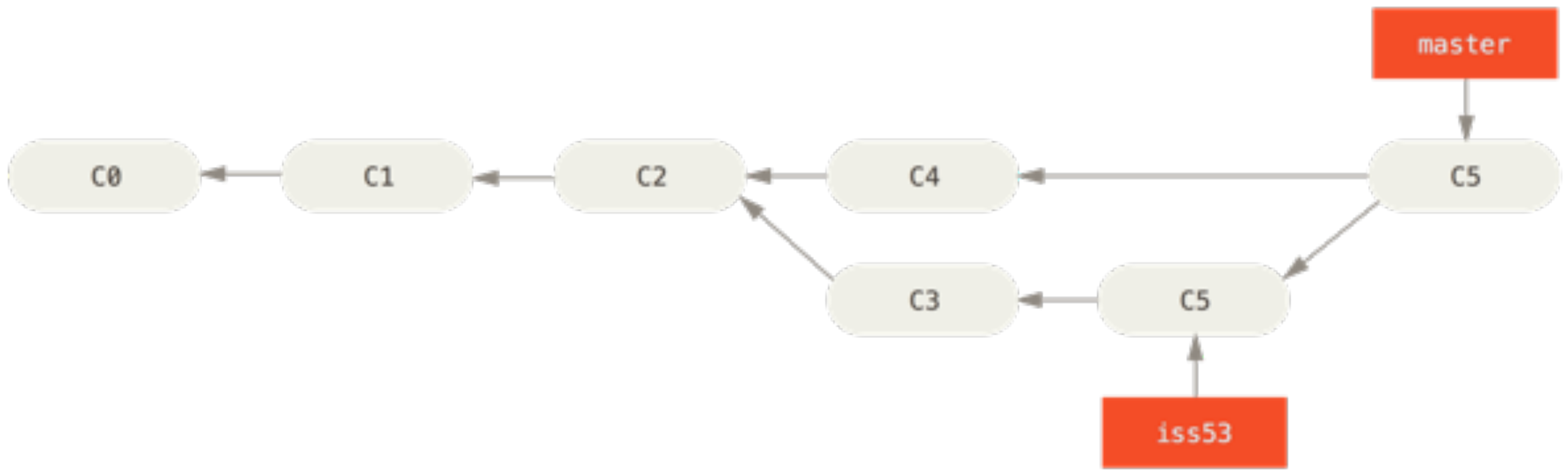
as always it is interesting to follow the history with

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

branching and merging



the work on iss53 is now completed and it is time to bring the iss53 into the main development line.
We therefore look at the last common ancestor between master and iss53



now we merge with the same commands as before

```
$ git checkout master
$ git merge iss53
```

git uses the last common ancestor and the 3 snapshots C3, C4 and C5.
Git then creates however a *new* snapshot, known as a merge-commit, instead of just moving the pointer. We can now remove the brand iss53

```
$ git branch -d iss53
```

branch management

occasionally, the commit does not go smoothly and a merge conflict needs to be resolved.
files with conflicts will remain uncommitted until the conflicts are resolved. Use *git status* to learn about remaining conflicts.

```
th-ea-lswtb02:Lec_git Lode.Pollet$ git merge testing
Auto-merging README
CONFLICT (content): Merge conflict in README
Automatic merge failed; fix conflicts and then commit the result.
```

```
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Changes to be committed:

    new file:   test.txt

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:   README
```

in this case, both branch hotfix and testing modified the same parts of the README file; the file test.txt in branch testing is new but poses no conflict.

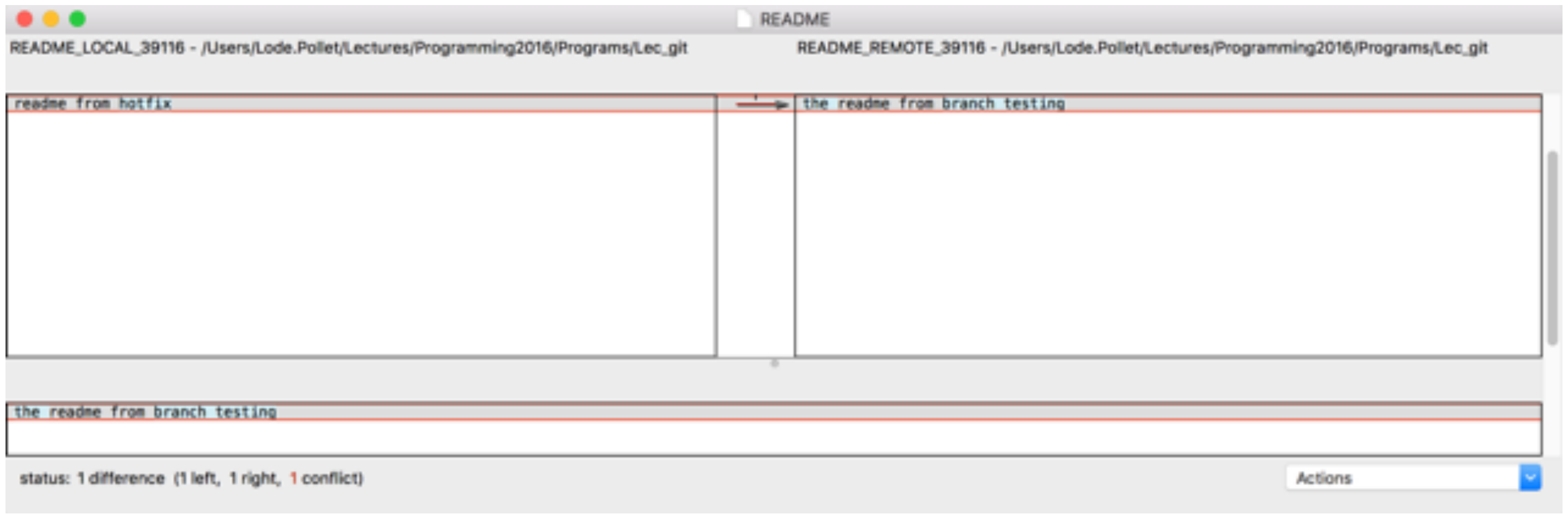
The conflicting file will look something like this:

```
th-ea-lswtb02:Lec_git Lode.Pollet$ more README
<<<<<<< HEAD
readme from hotfix
=======
the readme from branch testing
>>>>>>> testing
```

One recognizes the part on master (=HEAD) between <<<<< and ==== and the part on branch testing between ====== and >>>>>>

branch management

Resolve the conflict(s) manually or use *git mergetool* to address the conflict(s) if you install an editor for this



README

README_LOCAL_39116 - /Users/Lode.Pollet/Lectures/Programming2016/Programs/Lec_git

README_REMOTE_39116 - /Users/Lode.Pollet/Lectures/Programming2016/Programs/Lec_git

```
readme from hotfix
```

```
the readme from branch testing
```

```
the readme from branch testing
```

status: 1 difference (1 left, 1 right, 1 conflict)

The screenshot above shows *opendiff*.
under Actions we decide to take the left version. We save and exit. The directory now looks like:

```
Lode.Pollet$ ls
README        README.orig   README2       file1.txt     hello.txt     rainyday.txt  test.txt
th-ea-lswtb02:Lec_git Lode.Pollet$ more README
readme from hotfix
```

note in particular the file README.orig which is produced by opendiff. If everything is fine, we can delete this file (otherwise it will pop us as untracked in *git status*), and *commit*.

To get an overview of all branches, run *git branch* or *git branch -v* if you also want to see the last commit.

Also interesting is git branch —merged and git branch —no-merged to see all branches that are (not) merged.

we can now also delete the testing branch; *git branch -d testing*. Only merged branches can be deleted.

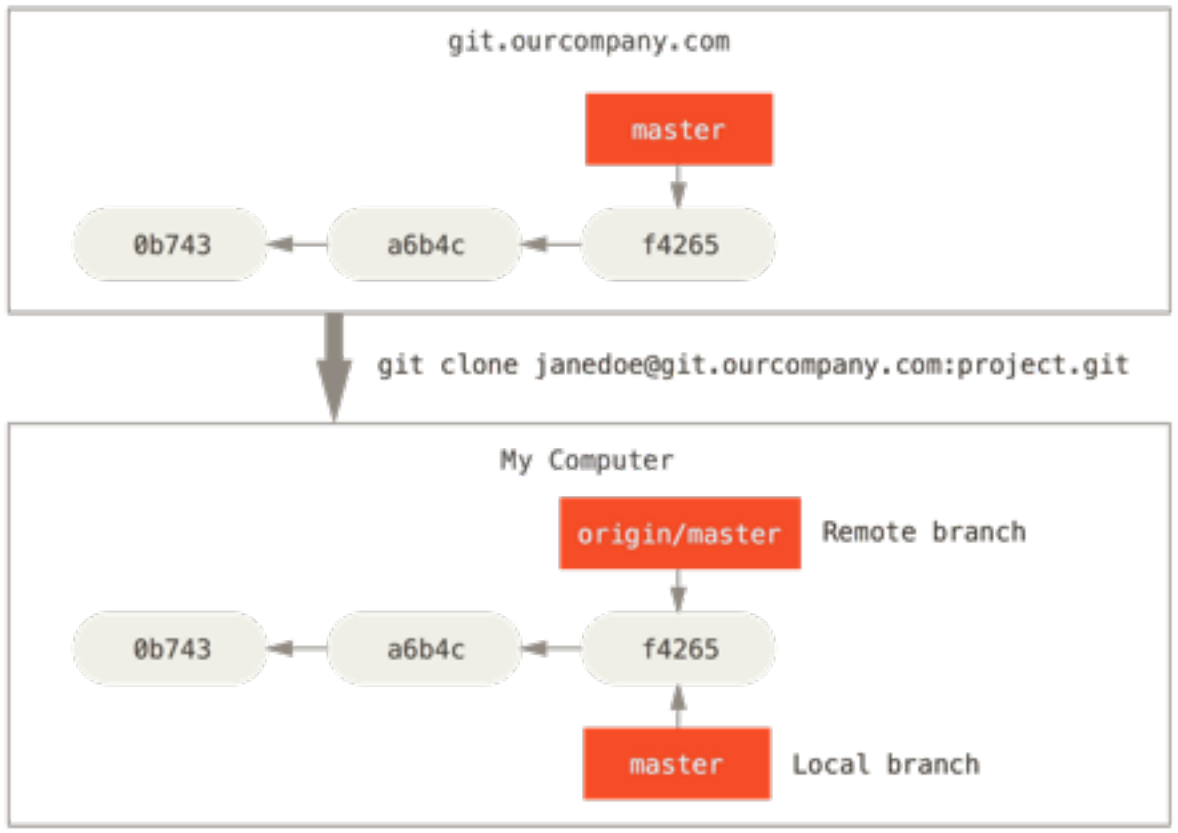
Finally, we get

```
Lec_git Lode.Pollet$ git log --oneline --graph --decorate --all
*   0384755 (HEAD -> master) Merge branch 'testing'
|\
| * add44b5 test
| * 5f0ea32 README
* | 04578ab changes in hotfix
|/
* b098393 untst
* 6df7c56 tst
```

Git : remote branches

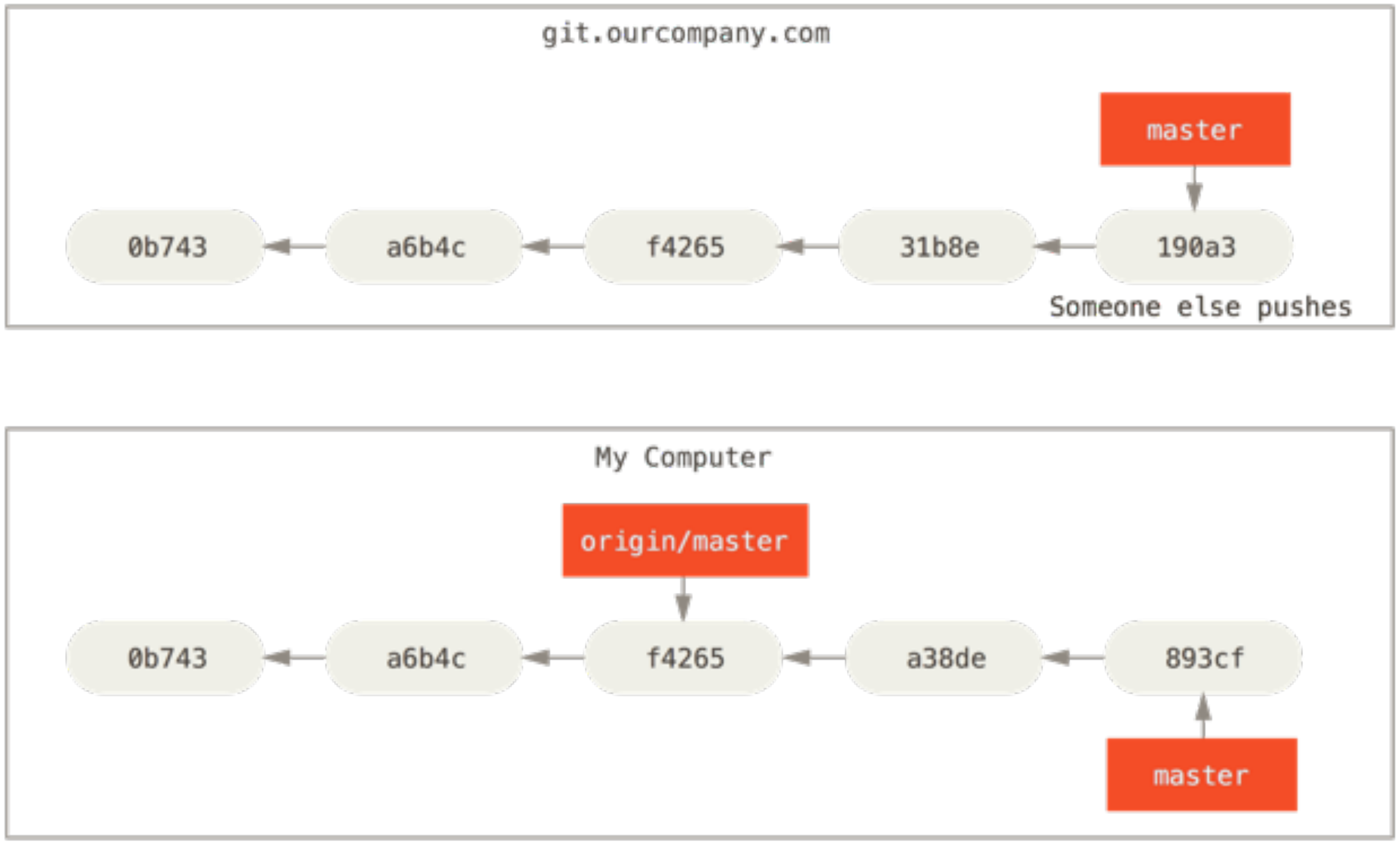
some jargon:

- master : default name for a branch when you run *git init*
- origin : default name for a remote when you run *git clone*



```
git clone janedoe@git.ourcompany.com:project.git
```

Git : remote branches

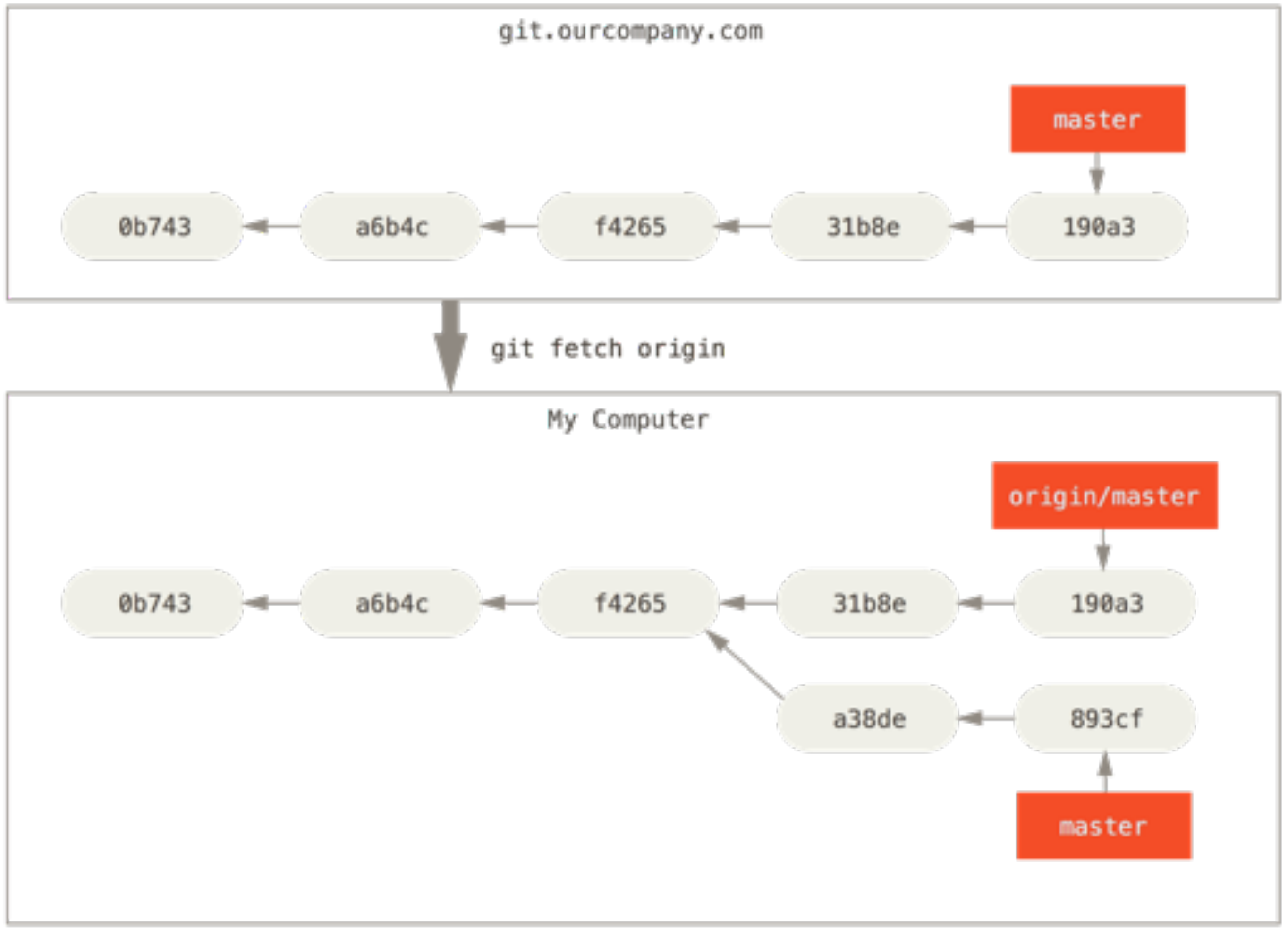


while you are working on your local branch, somebody else pushes to git.ourcompany.com and updates the master branch. Your origin/master does not move as long as you stay without contact with git.ourcomany.com

important command to get information on remote branches:

```
$ git remote show origin
$ git remote -v
```

Git : remote branches



to synchronize, run

```
$ git fetch origin
```

Fetching only downloads the data in your local repository; you have to do the changes manually

if you are ready to share your changes with the world, you have to run

```
$ git push origin master
```

Git

working with remotes

```
$ git clone https://github.com/schacon/ticgit
Cloning into 'ticgit'...
remote: Reusing existing pack: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.
Resolving deltas: 100% (772/772), done.
Checking connectivity... done.
$ cd ticgit
$ git remote
origin
```

origin is the default name Git gives to the server you cloned from

```
$ git remote -v
origin	https://github.com/schacon/ticgit (fetch)
origin	https://github.com/schacon/ticgit (push)
```

if there are multiple remotes (say several users) you can see them as follows

```
$ git remote add pb https://github.com/paulboone/ticgit
```

adding a remote repository

```
$ git remote -v
origin	https://github.com/schacon/ticgit (fetch)
origin	https://github.com/schacon/ticgit (push)
pb	https://github.com/paulboone/ticgit (fetch)
pb	https://github.com/paulboone/ticgit (push)
```

Git

Now you can use the string `pb` on the command line in lieu of the whole URL. For example, if you want to fetch all the information that Paul has but that you don't yet have in your repository, you can run `git fetch pb`:

```
$ git fetch pb
remote: Counting objects: 43, done.
remote: Total 43 (delta 21), reused 21 (delta 21), pack-reused 22
Unpacking objects: 100% (43/43), done.
From https://github.com/paulboone/ticgit
 * [new branch]      master     -> pb/master
 * [new branch]      ticgit     -> pb/ticgit
```

to get data from your remote projects, you can run:

```
$ git fetch [remote-name]
```

`git fetch origin` fetches any new work that has been pushed to that server since you cloned (or last fetched from) it. It's important to note that the `git fetch` command only downloads the data to your local repository – it doesn't automatically merge it with any of your work or modify what you're currently working on. You have to merge it manually into your work when you're ready.

`git pull` combines `git fetch` with `git merge`. Can sometimes be confusing, so maybe it is better to run them after another

When you have your project at a point that you want to share, you have to push it upstream. The command for this is simple: `git push [remote-name] [branch-name]`.

```
$ git push origin master
```

Git

```
$ git remote show origin
* remote origin
  Fetch URL: https://github.com/schacon/ticgit
  Push  URL: https://github.com/schacon/ticgit
  HEAD branch: master
  Remote branches:
    master tracked
    ticgit tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

Git summary

initializing:

```
git config —global user.name “myname”
git config —global user.email <email>
git config — list
git init
git clone <repo>
```

basic operations:

```
git add <filename>
git rm <filename>
git commit -m “msg”
git status
git log
git log —pretty=oneline;
git log -p -2
git log — stat ;
git log —pretty=format:”%h - %an, %ar : %s”
git diff
git diff — staged
git reset HEAD <file>
git checkout — <file>
```

Git summary

branches

```
git branch ; git branch -v ; git branch —merged ; git branch —no-merged
git branch <new_branch_name> ; git branch -d <old_branch_name>
git checkout <branch_name>
git checkout -b <new_branch_name>
git log —oneline —decorate —graph —all
git merge
git mergetool
git rebase
```

remote repo

```
git remote -v
git remote show origin
git remote add <name> <repo>
git push origin master
git fetch origin
git pull
```

jargon

master
origin
HEAD