Advanced Git by examples

Pocket edition

Sylvain Bouveret, Grégory Mounié, Matthieu Moy

2021

[first].[last]@imag.fr

http://systemes.pages.ensimag.fr/www-git/

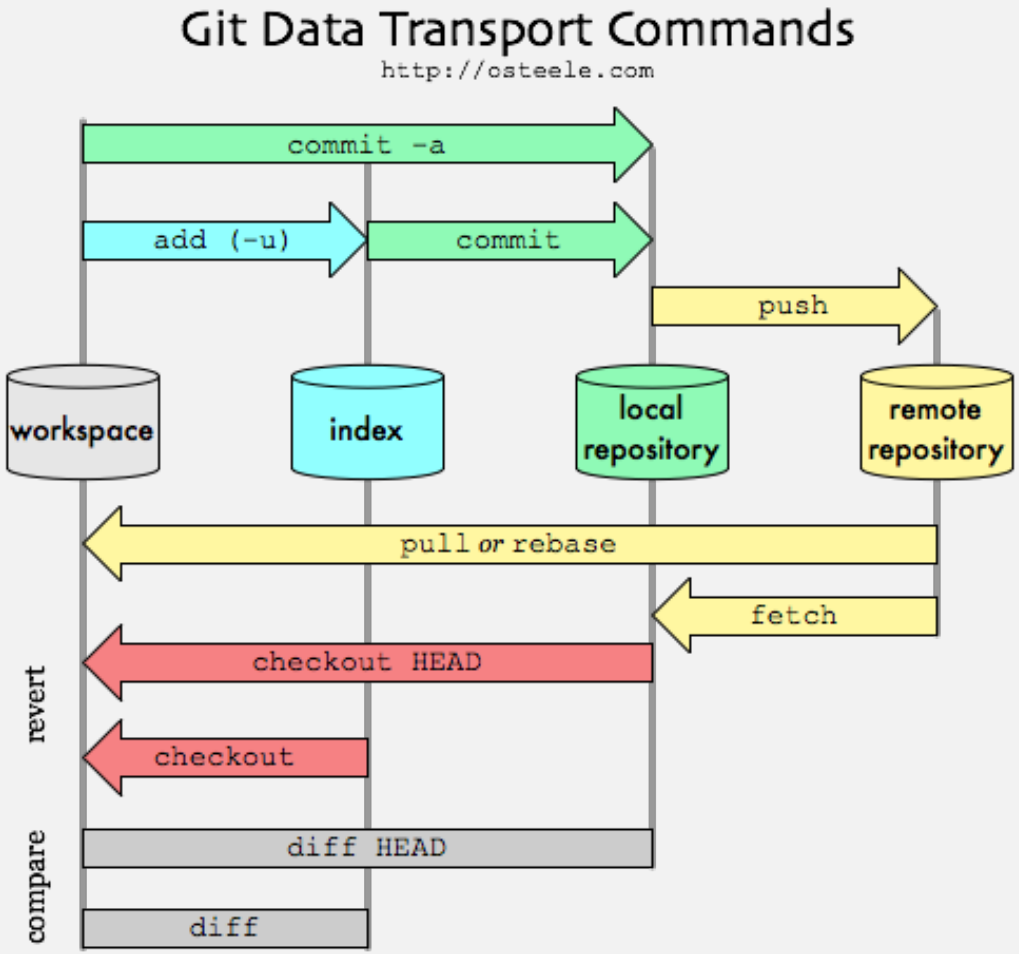
Goals of the presentation

Assumptions: basic Git (add,commit,push,pull,conflict) (small recall in the 2 next slides). Methods: a lot of small commands to do interactively.

- Few historical facts
- Git internals (+ code)
- Cleanup history (+ code)
- Global history: branching, rebasing, stashing (+ long code)
- Bonus: Configuration (+ tiny code)

Important missing point in examples

All code in this presentation are local. In real life, all "remote" stuff should be added: `git push`, `git fetch` and `git pull` of every shared branches, commits and tags; `git push` remote management and `--set-upstream`.


Git Data Transport Commands
http://osteele.com
commit -a
add (-u)
commit
push
workspace
index
local repository
remote repository
pull or rebase
fetch
checkout HEAD
checkout
revert
diff HEAD
diff
compare

`git status`, one command to rule them all

`git status` displays the current state of the working directory and the most relevant commands at the current state.

```
On branch master
Your branch is up to date with 'origin/master'.

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:   advanced1h30.tex

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

Git vs others VCS

A bit of history

1972: Birth of SCCS, Source Code Control System, included in UNIX

1982: Birth of RCS, Revision Control System

1986: Birth of CVS, Centralized Version System

2000: Birth of Subversion (SVN), a replacement for CVS with the same concepts

2001: Birth of the first distributed version control systems (GNU Arch)

2005: First version of Git

Git and the Linux Kernel

1991: Linus Torvalds starts writing Linux, using mostly tar+patch,

2002: Linux adopts BitKeeper, a proprietary decentralized version control system (available free of cost for Linux),

2002-2005: Flamewars against BitKeeper, some Free Software alternatives appear (GNU Arch, Darcs, Monotone). None are good enough technically.

Git and the Linux Kernel

1991: Linus Torvalds starts writing Linux, using mostly tar+patch,

2002: Linux adopts BitKeeper, a proprietary decentralized version control system (available free of cost for Linux),

2002-2005: Flamewars against BitKeeper, some Free Software alternatives appear (GNU Arch, Darcs, Monotone). None are good enough technically.

2005: BitKeeper's free of cost license revoked. Linux has to migrate.

2005: Unsatisfied with the alternatives, Linus decides to start his own project, Git.

Who Makes Git? (Dec. 26th 2021)

```
you@laptop$ git shortlog -s --no-merges | sort -nr | head -31
  7362  Junio C Hamano # Google (full-time on Git)
  3704  Jeff King  # GitHub (almost full-time on Git)
  1927  Johannes Schindelin  # Microsoft (full-time on Git)
  1824  Nguyễn Thái Ngọc Duy
  1290  Shawn O. Pearce  # Google
  1277  Ævar Arnfjörð Bjarmason
  1104  Linus Torvalds  # No longer very active, 1 ci in 2019
  1045  René Scharfe
   953  Michael Haggerty   # GitHub
   841  Elijah Newren
   ...
   287  Matthieu Moy # UCB/ENSL, ci when @ Ensimag (31/1930)
```

Git Adoption (Hosting)

- 2015: "There are 11.6M people collaborating right now across 29.1M repositories on GitHub" https://github.com/about/press
- Github 2017: 25M people and 75M repositories
- Github Aug 2019: 40M people and $>$ 100M repositories
- Github June 2018: Microsoft buy Github for 7.5 billion $
- Github Dec 2021 (wikipedia): 65M people
- Gitlab Dec 2021 (wikipedia): 30M people, 1M registered licence people
- How about Mercurial? Bitbucket: only Mercurial repos in 2008, drop it June 1 2020, for becoming Git only.

Numerous alternatives in two families

See Comparison of 30+ version control software in Wikipedia

Distributed: Git, Mercurial, Bazaar, Darcs, Fossil, Arch (tla) etc.

Client-server: SVN, CVS, RCS, SCCS; etc.; Propertary:
ClearCase, Perforce, etc.

Git init by examples

Code: INIT 1/2

Create a minimalist repository

Create a simple repository with two files in 2 commits, one of the file in a sub-directory.

```
you@laptop$ mkdir MyRepo.git
you@laptop$ cd MyRepo.git
you@laptop$ git init . # comments on main branch name
you@laptop$ touch myfile.txt
you@laptop$ git hash-object myfile.txt # keep the SHA1 in mind
you@laptop$ git add myfile.txt
you@laptop$ git commit -m "first commit"
[master (commit racine) aac95f0] first commit
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 myfile.txt
```

Code: INIT 2/2

Create a minimalist repository

Create a simple repository with two files in 2 commits, one of the file in a sub-directory.

```
you@laptop$ mkdir Subdir
you@laptop$ touch Subdir/mysubfile.txt
you@laptop$ git add Subdir/mysubfile.txt
you@laptop$ git commit -m "second commit"
[master eee2e0b] second commit
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 Subdir/mysubfile.txt
you@laptop$ git config --local alias.lg "log --all --graph
↪  --oneline" # Bonus
you@laptop$ git lg
* eee2e0b (HEAD -> master) second commit
* aac95f0 first commit
```

Git internals by examples

Why do I need to learn about Git's internal?

- Beauty of Git: very simple data model
  (The tool is clever, the repository format is simple&stupid)
- Understand the model, and the 150+ commands will become much simpler to understand!

Let's explore the data model of `.git` !

Data exploration 1/3

```
you@laptop$ ls -F .git
branches/  COMMIT_EDITMSG  config  description  HEAD  hooks/
↪  index  info/  logs/  objects/  refs/
you@laptop$ cat .git/HEAD
ref: refs/heads/master
you@laptop$ cat .git/refs/heads/master
eee2e0b4bf68f4dc0e0f73bc6594736253206397
you@laptop$ ls .git/objects/
aa  af  b4  e2  e6  ee  info  pack
you@laptop$ ls .git/objects/ee
e2e0b4bf68f4dc0e0f73bc6594736253206397
```

Data exploration 2/3

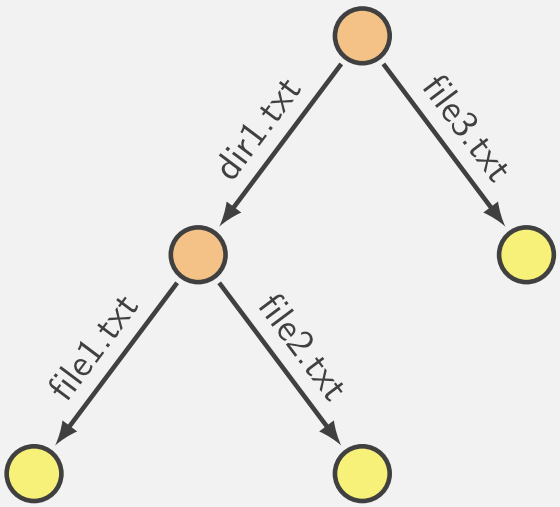
```
you@laptop$ git cat-file -p eee2e0b4bf68f4dc0e0f73bc6594736253206397 # ou eee2e
tree af6f7952b127efa16e7e60d3e39597107e3c19ce
parent aac95f06950b585f28c9027429fb806e22f55e30
author Grégory Mounié <Gregory.Mounie@imag.fr> 1578330554 +0100
committer Grégory Mounié <Gregory.Mounie@imag.fr> 1578330554 +0100

second commit
```

Data exploration 3/3

```
you@laptop$ git cat-file -p af6f7952b127efa16e7e60d3e39597107e3c19ce
040000 tree e2aa7703c36797d761b847d5cf9bf70098487ce0    Subdir
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391 myfile.txt
you@laptop$ git cat-file -p e2aa7703c36797d761b847d5cf9bf70098487ce0
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391 mysubfile.txt
you@laptop$ git cat-file -p e69de29bb2d1d6434b8b29ae775ad8c2e48c5391 # n'affiche rien
```

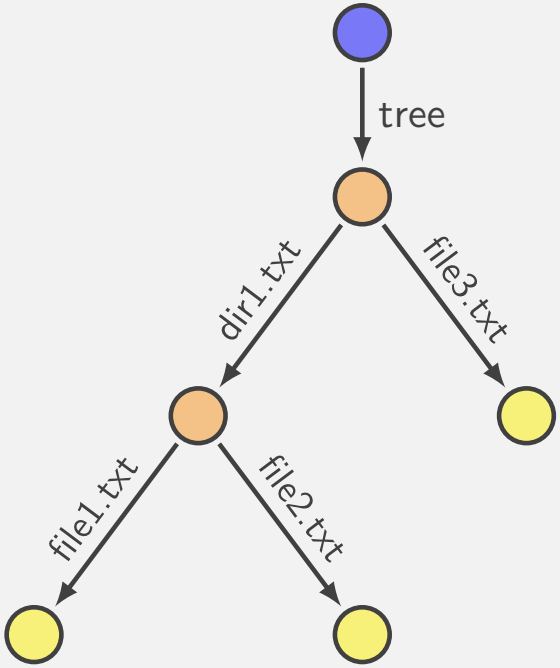
Content of a Git repository: Git objects

- **blob** Any sequence of bytes, represents file content
- **tree** Associates object to pathnames, represents a directory



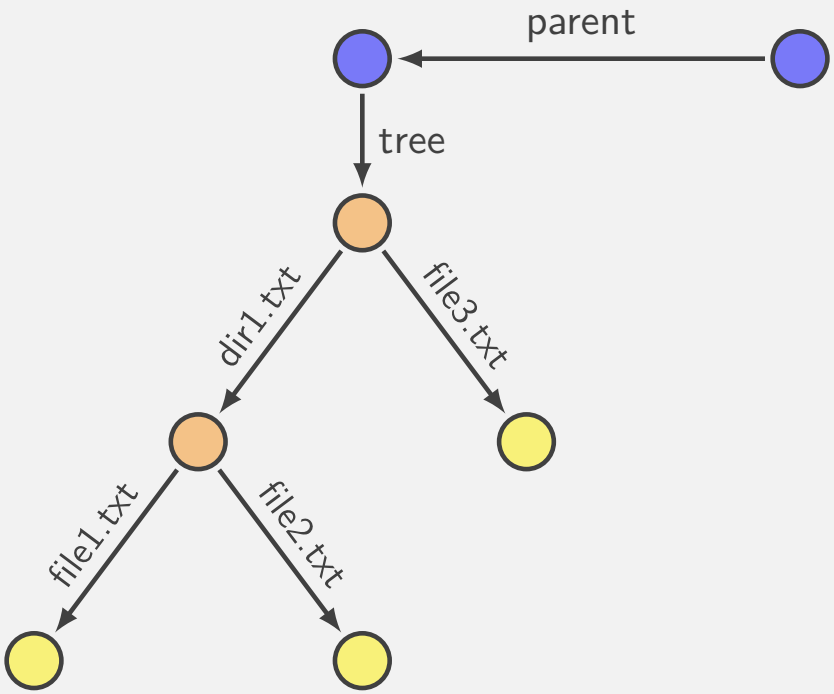
Content of a Git repository: Git objects

- **blob** Any sequence of bytes, represents file content
- **tree** Associates object to pathnames, represents a directory
- **commit** Metadata + pointer to tree + pointer to parents



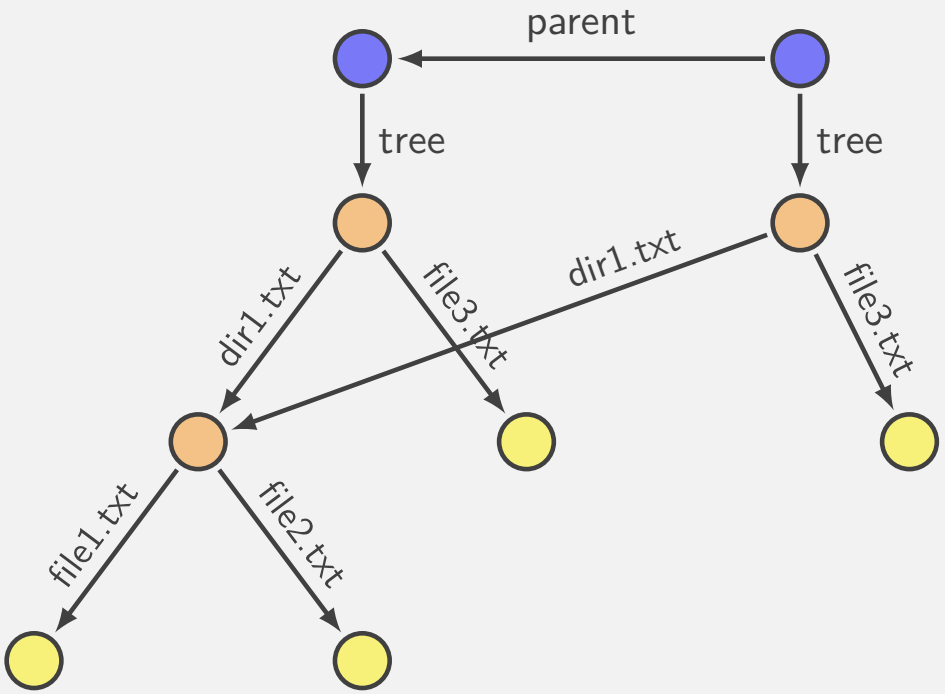
Content of a Git repository: Git objects

- **blob** Any sequence of bytes, represents file content
- **tree** Associates object to pathnames, represents a directory
- **commit** Metadata + pointer to tree + pointer to parents



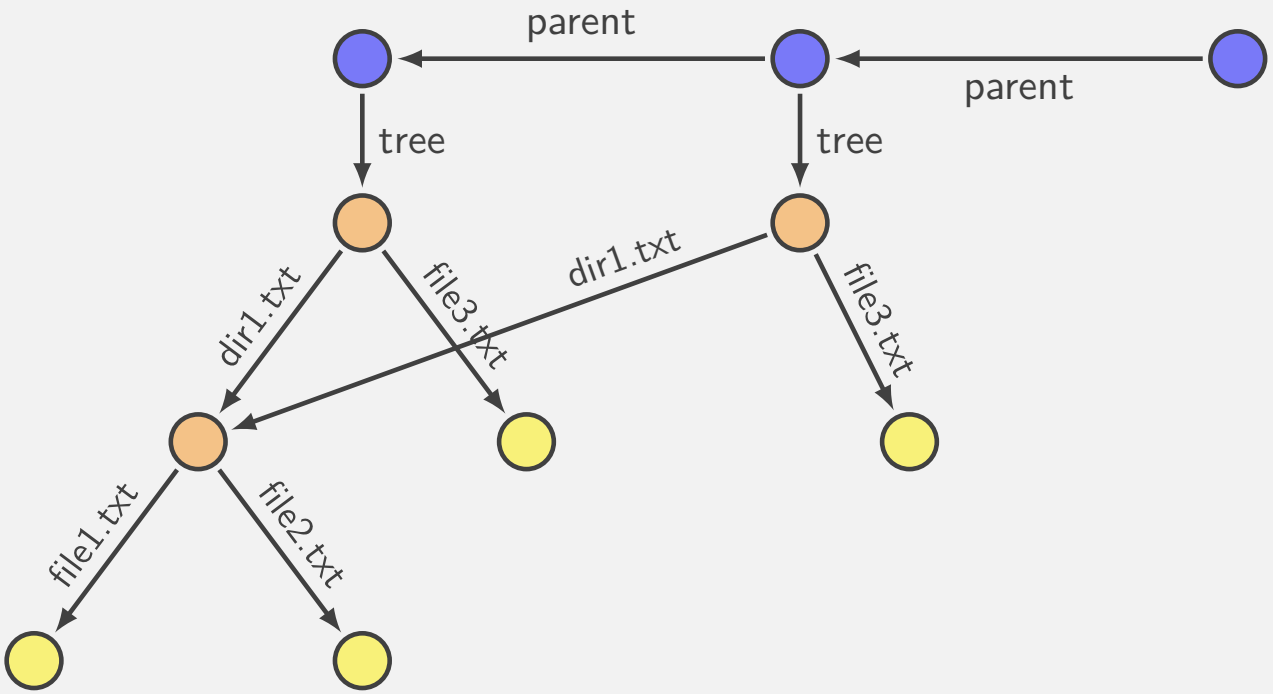
Content of a Git repository: Git objects

- **blob** Any sequence of bytes, represents file content
- **tree** Associates object to pathnames, represents a directory
- **commit** Metadata + pointer to tree + pointer to parents



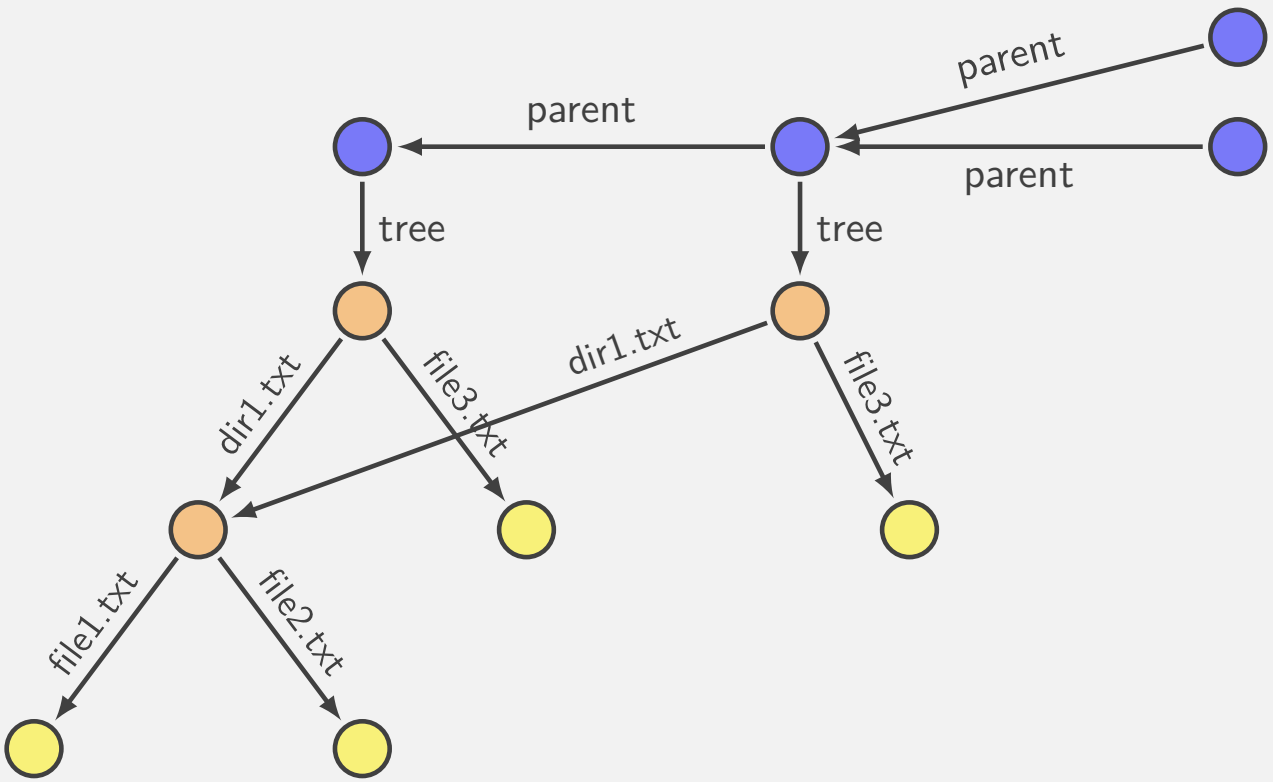
Content of a Git repository: Git objects

- **blob** Any sequence of bytes, represents file content
- **tree** Associates object to pathnames, represents a directory
- **commit** Metadata + pointer to tree + pointer to parents



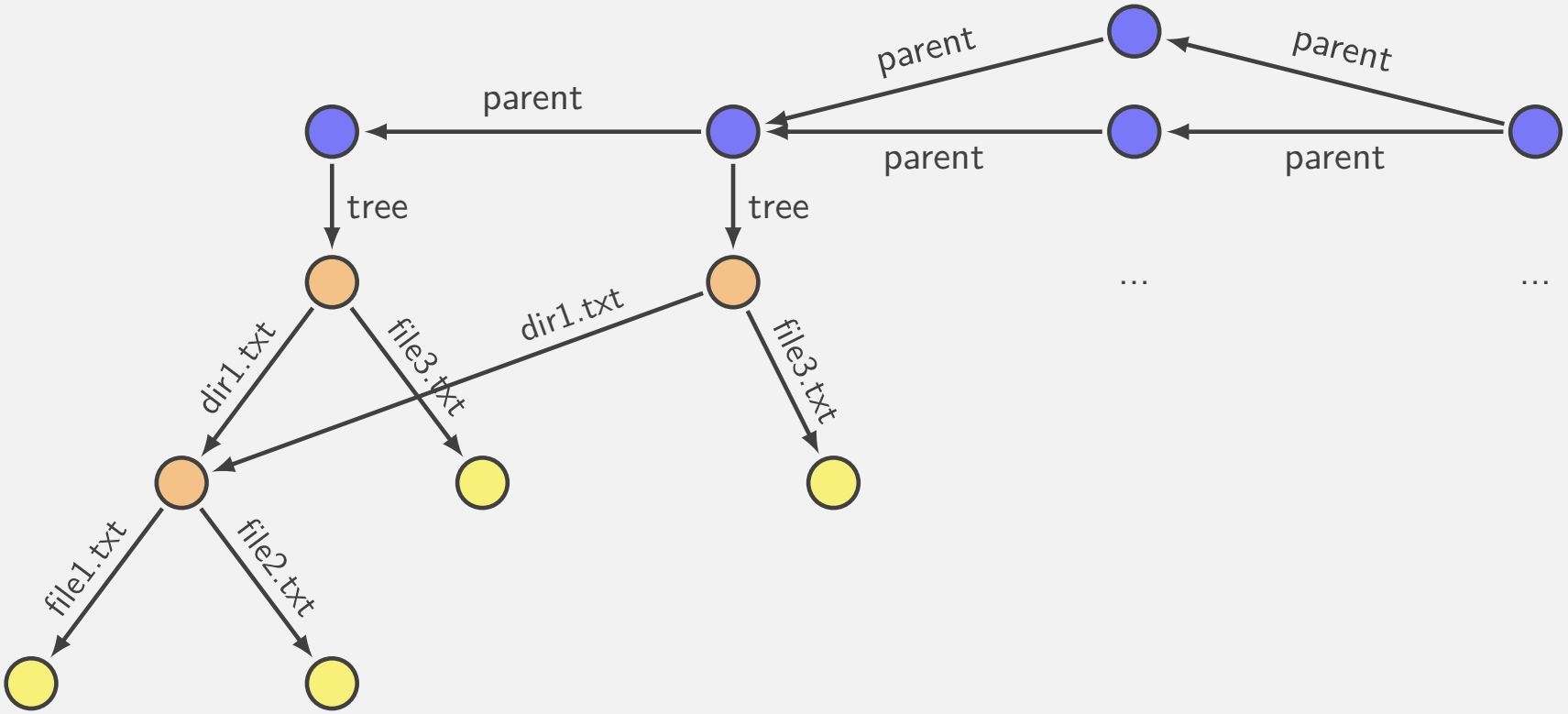
Content of a Git repository: Git objects

- **blob** Any sequence of bytes, represents file content
- **tree** Associates object to pathnames, represents a directory
- **commit** Metadata + pointer to tree + pointer to parents



Content of a Git repository: Git objects

- **blob** Any sequence of bytes, represents file content
- **tree** Associates object to pathnames, represents a directory
- **commit** Metadata + pointer to tree + pointer to parents



Git objects: On-disk format

- By default, 1 object = 1 file
- Name of the file = object unique identifier content
- Content-addressed database:
  - Identifier computed as a hash of its content
  - Content accessible from the identifier
- Consequences:
  - Objects are immutable
  - Objects with the same content have the same identity (deduplication for free)
  - No known collision in SHA1
  - Acyclic (DAG = Directed Acyclic Graph)

Branches, tags: references

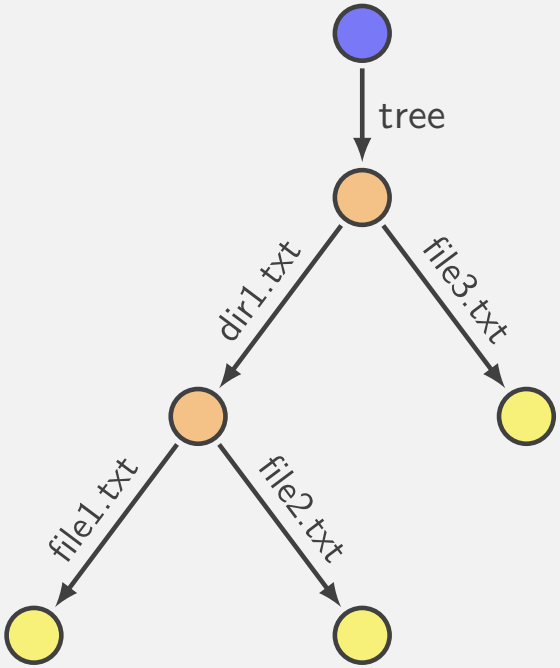
- In Java:

```java
String s; // Reference named s
s = new String("foo"); // Object pointed to by s
String s2 = s; // Two refs for the same object
```

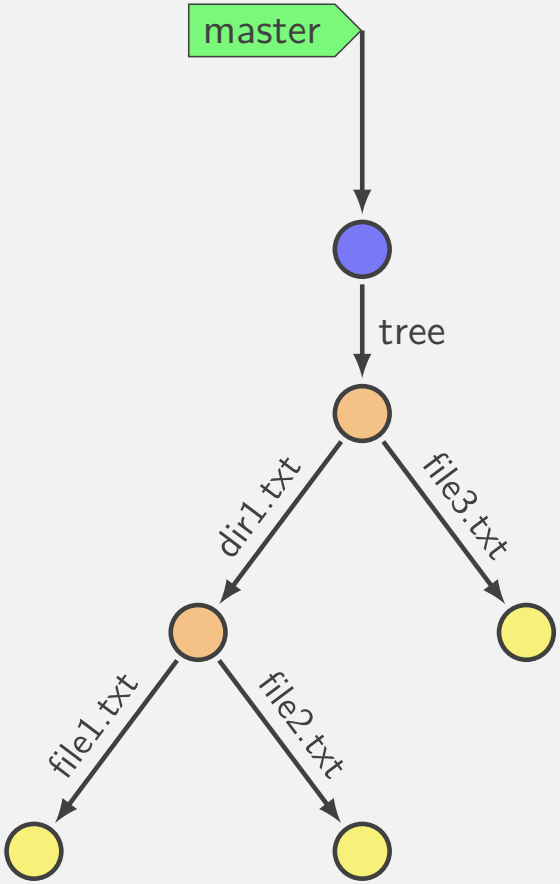
- In Git: likewise!

```
$ git log --oneline
5454e3b add README.txt
7a7fb77 Initial commit
$ cat .git/HEAD
ref: refs/heads/master
$ cat .git/refs/heads/master
5454e3b51e81d8d9b7e807f1fc21e618880c1ac9
$ git symbolic-ref HEAD
refs/heads/master
$ git rev-parse refs/heads/master
5454e3b51e81d8d9b7e807f1fc21e618880c1ac9
```

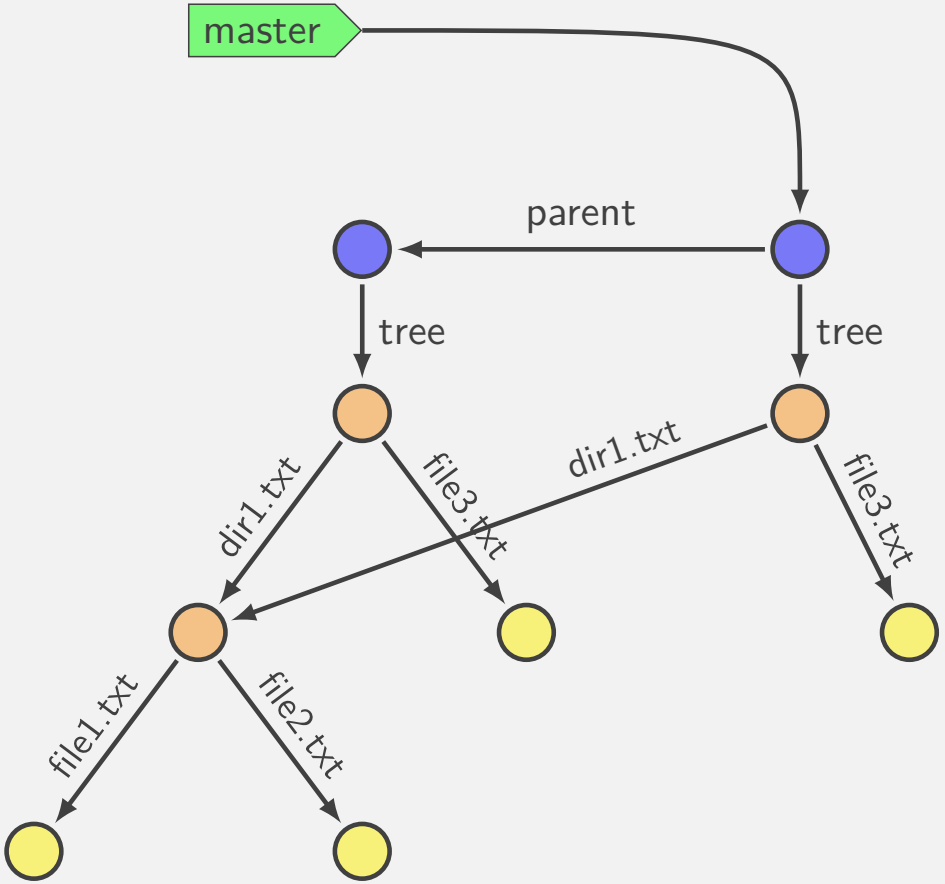
References (refs) and objects



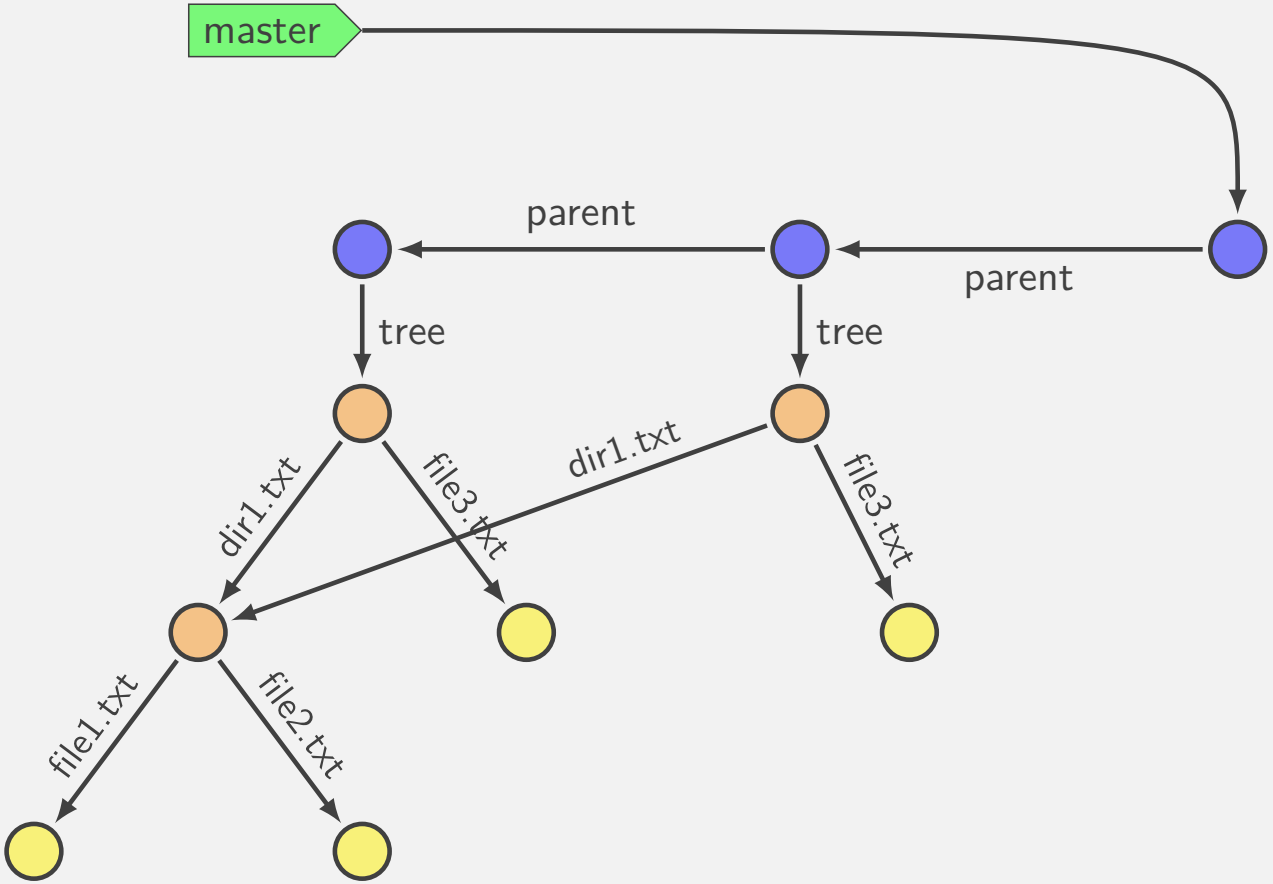
References (refs) and objects



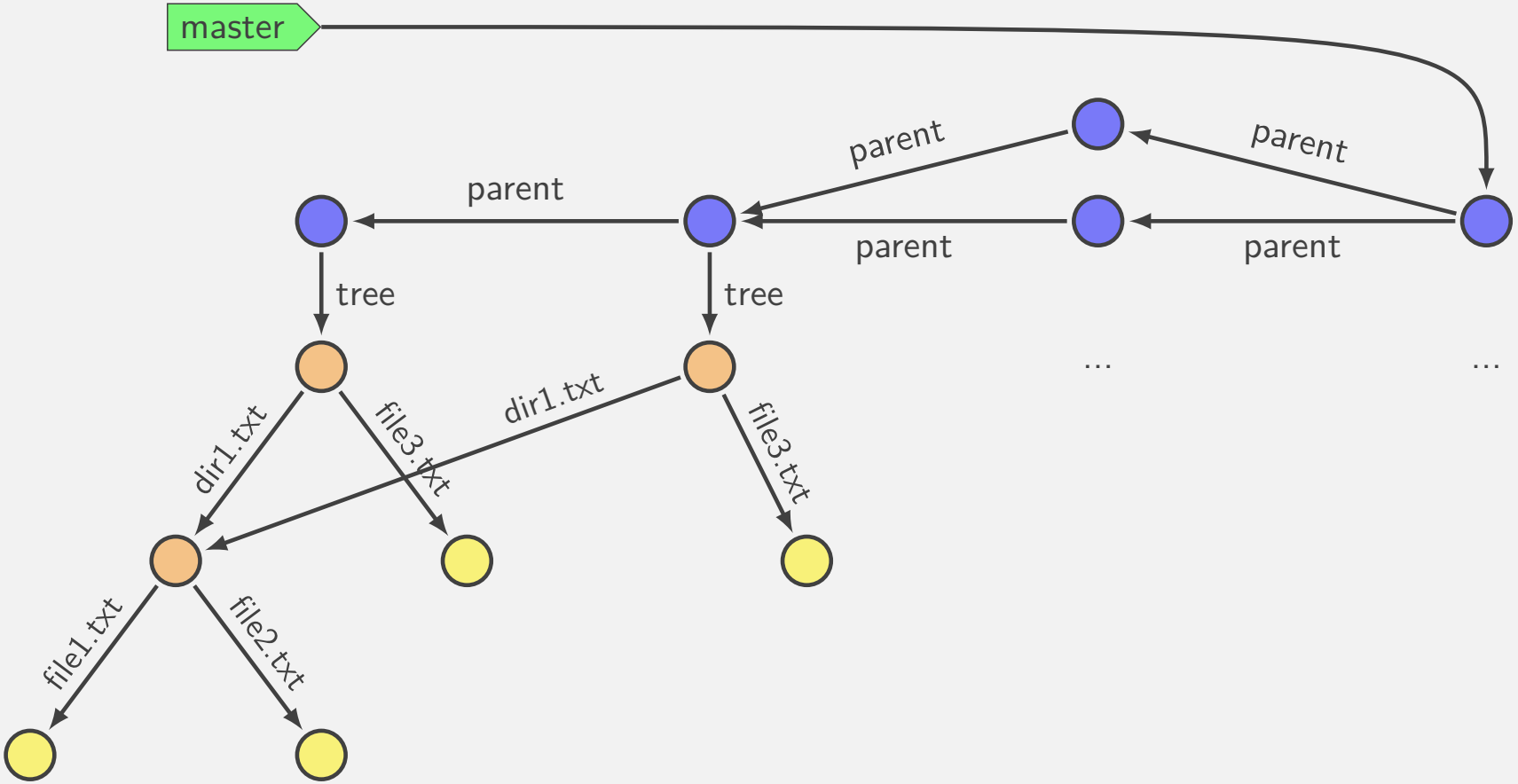
References (refs) and objects



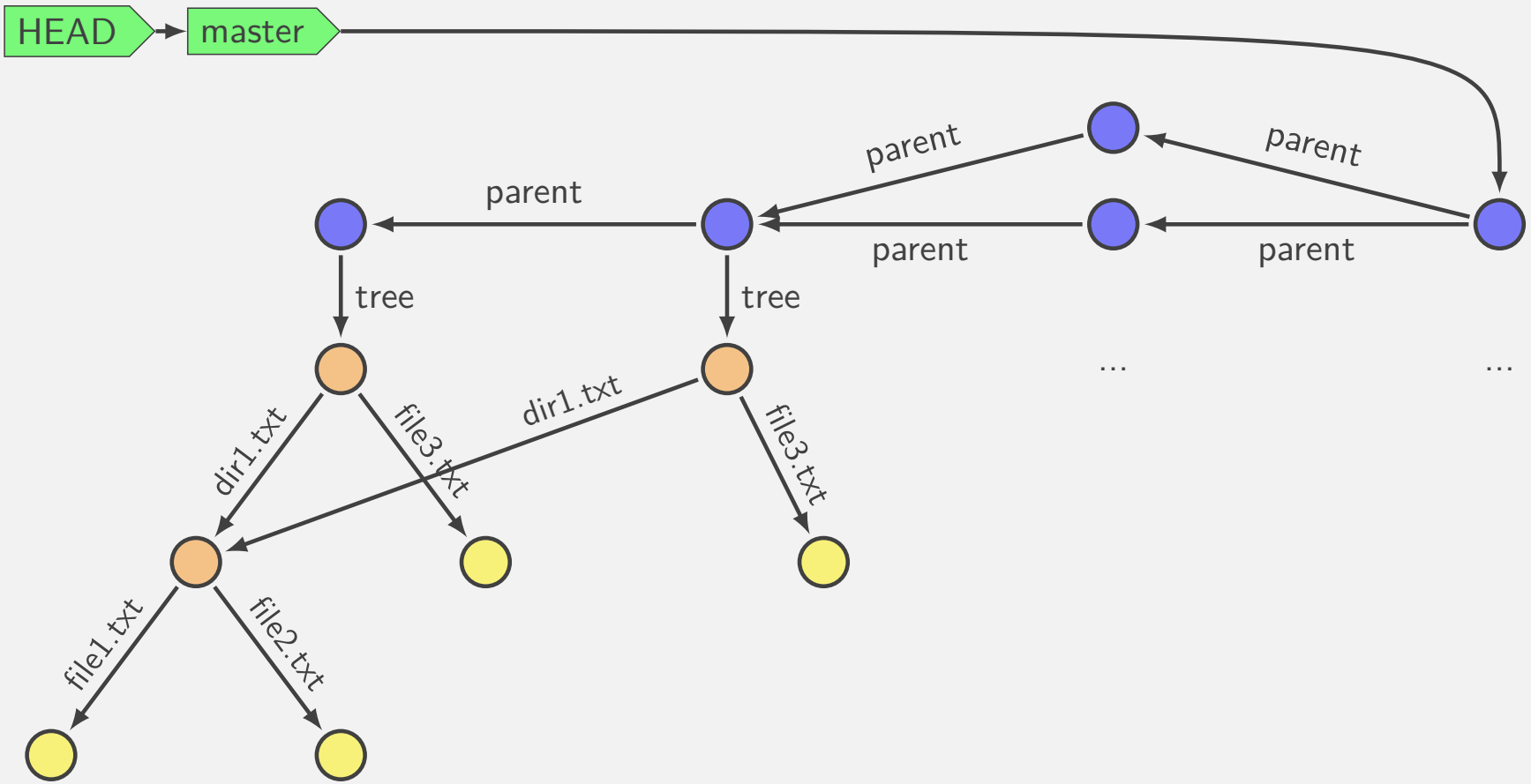
References (refs) and objects



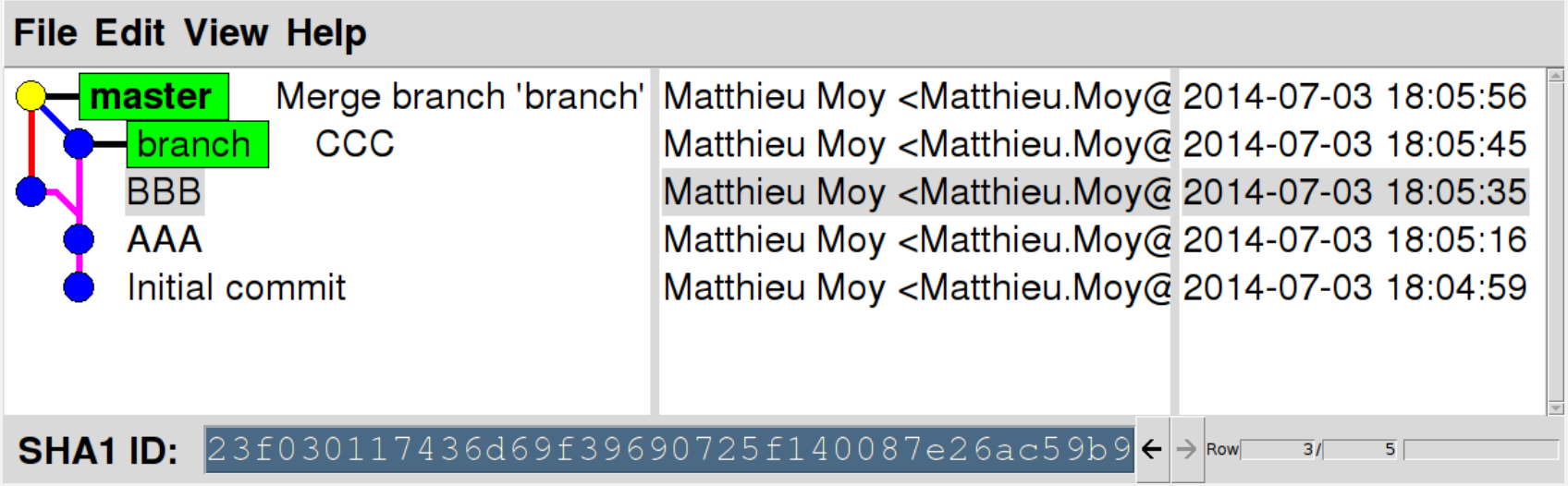
References (refs) and objects



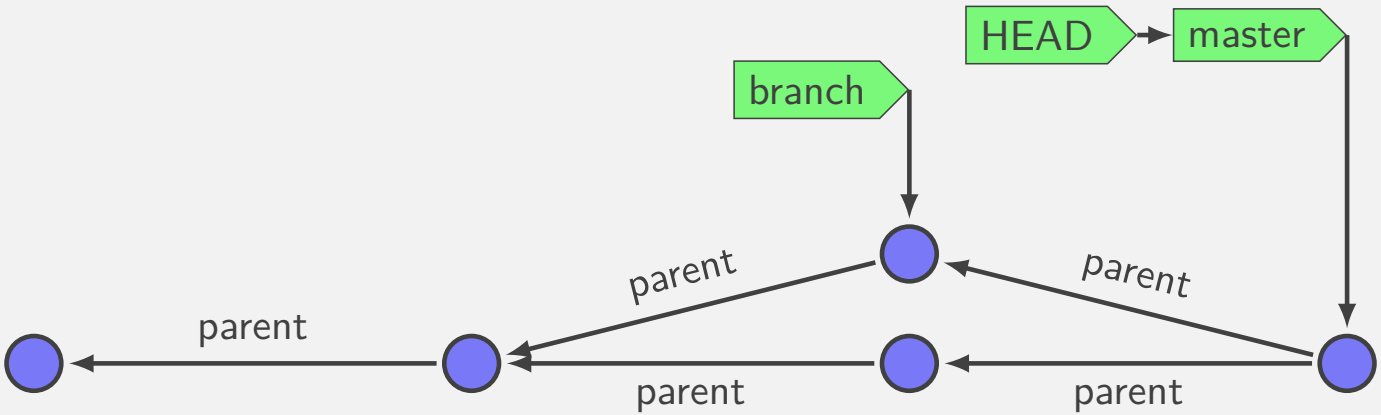
References (refs) and objects



Sounds Familiar?



≈



Branches, HEAD, tags

- A branch is a ref to a commit
- A lightweight tag is a ref (usually to a commit) (like a branch, but doesn't move)
- Annotated tags are objects containing a ref + a (signed) message
- HEAD is "where we currently are"
  - If HEAD points to a branch, the next commit will move the branch
  - If HEAD points directly to a commit (detached HEAD), the next commit creates a commit not in any branch (warning!)

Clean History: Why?


COMMENT
DATE
CREATED MAIN LOOP & TIMING CONTROL
14 HOURS AGO
ENABLED CONFIG FILE PARSING
9 HOURS AGO
MISC BUGFIXES
5 HOURS AGO
CODE ADDITIONS/EDITS
4 HOURS AGO
MORE CODE
4 HOURS AGO
HERE HAVE CODE
4 HOURS AGO
AAAAAAAA
3 HOURS AGO
ADKFJSLKDFJSDKLFJ
3 HOURS AGO
MY HANDS ARE TYPING WORDS
2 HOURS AGO
HAAAAAAAAANDS
2 HOURS AGO
AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

Git blame: Who did that?

GRENOBLE INP UGA Ensimag

`git gui blame` *file*

Bisect: Find regressions

```
$ git bisect start
$ git bisect bad
$ git bisect good v1.9.0
Bisecting: 607 revisions left to test after this (roughly 9 steps)
[8fe3ee67adcd2ee9372c7044fa311ce55eb285b4] Merge branch 'jx/i18n'
$ git bisect good
Bisecting: 299 revisions left to test after this (roughly 8 steps)
[aa4bffa23599e0c2e611be7012ecb5f596ef88b5] Merge branch 'jc/cod[...
$ git bisect good
Bisecting: 150 revisions left to test after this (roughly 7 steps)
[96b29bde9194f96cb711a00876700ea8dd9c0727] Merge branch 'sh/ena[...
$ git bisect bad
Bisecting: 72 revisions left to test after this (roughly 6 steps)
[09e13ad5b0f0689418a723289dca7b3c72d538c4] Merge branch 'as/pre[...
...
$ git bisect good
60ed26438c909fd273528e67 is the first bad commit
commit 60ed26438c909fd273528e67b399ee6ca4028e1e
```

Then what?

`git blame` and `git bisect` point you to a commit, then ...

- Dream:
  - Commit is a 50-lines long patch
  - Commit message explains the intent of the programmer
- Nightmare 1:
  - Commit mixes a large reindentation, a bugfix and a real feature
  - Message says "I reindented, fixed a bug and added a feature"
- Nightmare 2:
  - Commit is a trivial fix for the previous commit
  - Message says "Oops, previous commit was stupid"
- Nightmare 3:
  - Bisect not even applicable because most commits aren't compilable.

"History is a set of lies agreed upon", a good approach

When you write a draft of a document, and then a final version, does the final version reflect the mistakes you did in the draft?

- Popular approach with modern VCS (Git, Mercurial. . .)
- History tries to show the best logical path from one point to another
- Pros:
  - See above: blame, bisect, ...
  - Code review
  - Claim that you are a better programmer than you really are!

What is a clean history

- Each commit introduce small group of related changes ($\approx$ 100 lines changed max, no minimum!)
- Each commit is compilable and passes all tests ("bisectable history")
- "Good" commit messages

Writing good commit messages

Reminder: good comments

- Bad:

```
int i; // Declare i of type int
for (i = 0; i < 10; i++) { ... }
f(i)
```

- Possibly good:

```
int i; // We need to declare i outside the for
// loop because we'll use it after.
for (i = 0; i < 10; i++) { ... }
f(i)
```

Reminder: good comments

- Bad: What? The code already tells

```
int i; // Declare i of type int
for (i = 0; i < 10; i++) { ... }
f(i)
```

- Possibly good: Why? Usually the relevant question

```
int i; // We need to declare i outside the for
// loop because we'll use it after.
for (i = 0; i < 10; i++) { ... }
f(i)
```

Reminder: good comments

- Bad: What? The code already tells

```
int i; // Declare i of type int
for (i = 0; i < 10; i++) { ... }
f(i)
```

- Possibly good: Why? Usually the relevant question

```
int i; // We need to declare i outside the for
// loop because we'll use it after.
for (i = 0; i < 10; i++) { ... }
f(i)
```

Common rule: if your code isn't clear enough, rewrite it to make it clearer instead of adding comments.

Good commit messages

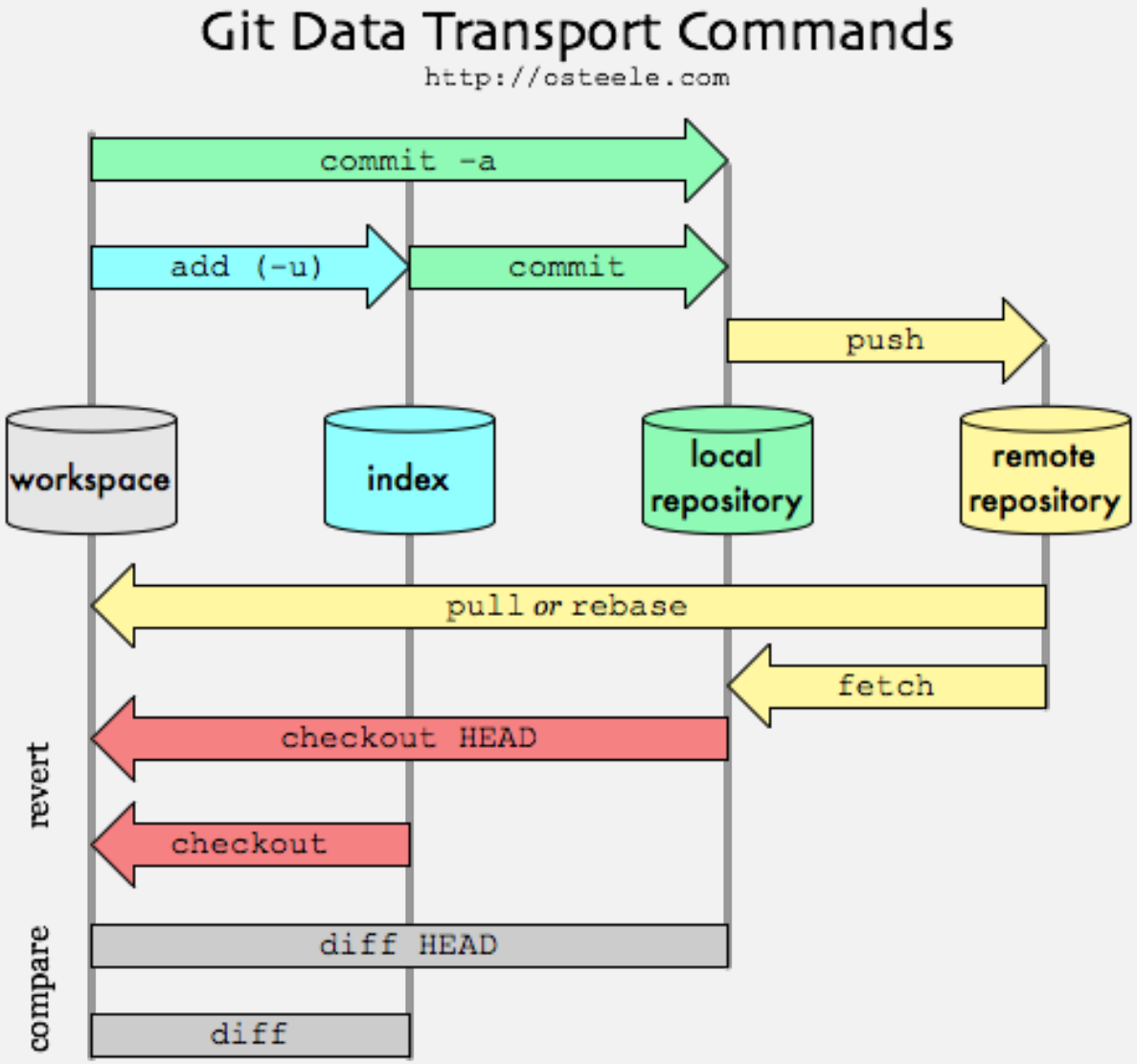
- Recommended format:

  ```
  One-line description (< 50 characters)

  Explain here WHY your change is good.
  ```

- Write your commit messages like an email: subject and body
- Imagine your commit message is an email sent to the maintainer, trying to convince him to merge your code[1]
- Don't use `git commit -m` (unlike previous exercises). Use `git gui` (or `magit-mode`, etc.) instead.

[1] Not just imagination, see `git send-email`

Partial commits, the index


Git Data Transport Commands
http://osteele.com
commit -a
add (-u)
commit
push
workspace
index
local repository
remote repository
pull or rebase
fetch
checkout HEAD
checkout
diff HEAD
diff
revert
compare

The index, or "Staging Area"

- "the index" is where the next commit is prepared
- Contains the list of files and their content
- `git commit` transforms the index into a commit
- `git commit -a` stages all changes in the worktree in the index before committing. Don't use it. You'll find it sloppy soon.

Dealing with the index

- Commit only 2 files:

```
1        git add file1.txt
2        git add file2.txt
3        git commit  # or git gui, magit ...
```

- Commit only some patch hunks:

```
1        git add -p
2        (answer yes or no for each hunk)
3        git commit  # or git gui, magit ...
```

git add -p: example

```
$ git add -p
@@ -1,7 +1,7 @@
int main()
  -       int i;
  +       int i = 0;
  printf("Hello, ");
  i++;
  Stage this hunk [y,n,q,a,d,/,K,g,e,?]? y
```

git add -p: example

```
$ git add -p
@@ -1,7 +1,7 @@
int main()
  -       int i;
  +       int i = 0;
  printf("Hello, ");
  i++;
  Stage this hunk [y,n,q,a,d,/,K,g,e,?]? y
  @@ -5,6 +5,6 @@

  -       printf("i is %s\n", i);
  +       printf("i is %d\n", i);

Stage this hunk [y,n,q,a,d,/,K,g,e,?]? n
```

git add -p: example

```
$ git add -p
@@ -1,7 +1,7 @@
int main()
  -       int i;
  +       int i = 0;
  printf("Hello, ");
  i++;
  Stage this hunk [y,n,q,a,d,/,K,g,e,?]? y
  @@ -5,6 +5,6 @@

  -       printf("i is %s\n", i);
  +       printf("i is %d\n", i);

Stage this hunk [y,n,q,a,d,/,K,g,e,?]? n
$ git commit -m "Initialize i properly"
[master c4ba68b] Initialize i properly
1 file changed, 1 insertion(+), 1 deletion(-)
```

`git add -p`: dangers

- Commits created with `git add -p` do not correspond to what you have on disk
- You probably never tested this commit ...
- Solutions:
  - `git stash -k`: stash what's not in the index
  - `git rebase --exec`: see later
  - (and code review)

Clean local history

Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

- Approach 1: merge (default with `git pull`)

Merging With Upstream

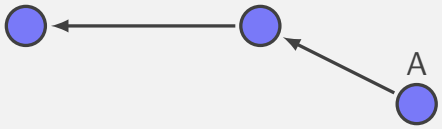
Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

- Approach 1: merge (default with `git pull`)

Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

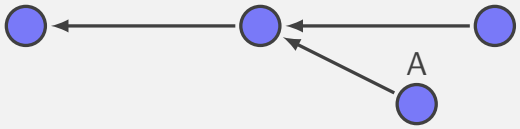
- Approach 1: merge (default with `git pull`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

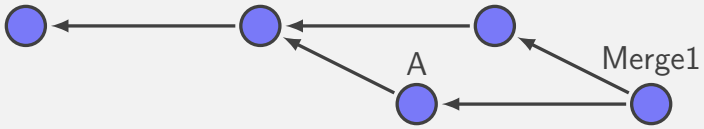
- Approach 1: merge (default with `git pull`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

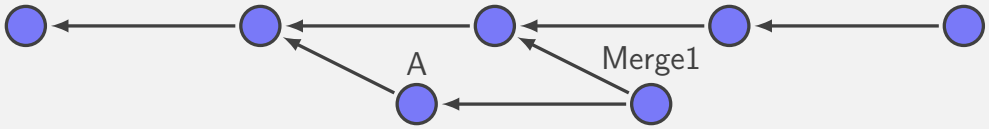
- Approach 1: merge (default with `git pull`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

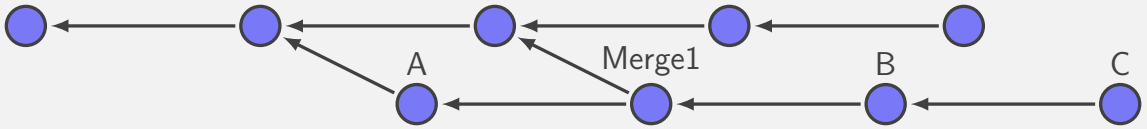
- Approach 1: merge (default with `git pull`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

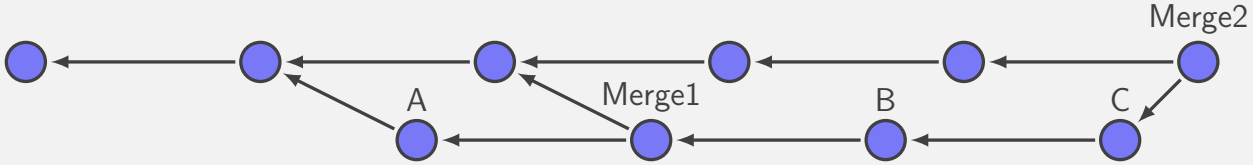
- Approach 1: merge (default with `git pull`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

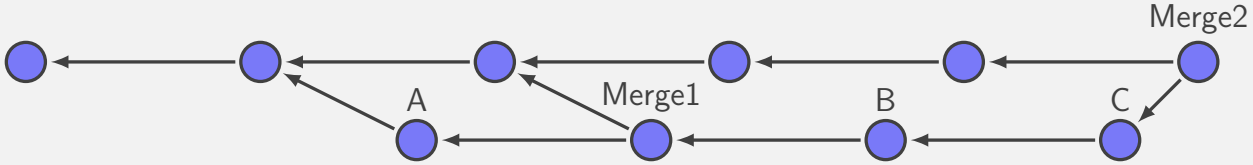
- Approach 1: merge (default with `git pull`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

- Approach 1: merge (default with `git pull`)



- Drawbacks:
  - Merge1 is not relevant, distracts reviewers (unlike Merge2).

Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

- Approach 2: no merge

Merging With Upstream

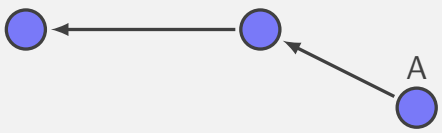
Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

- Approach 2: no merge

Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

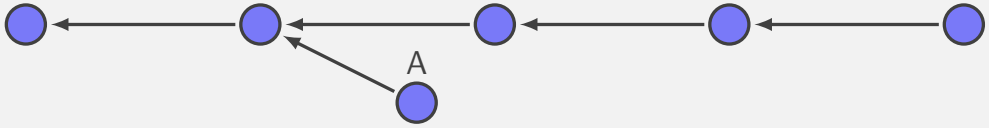
- Approach 2: no merge



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

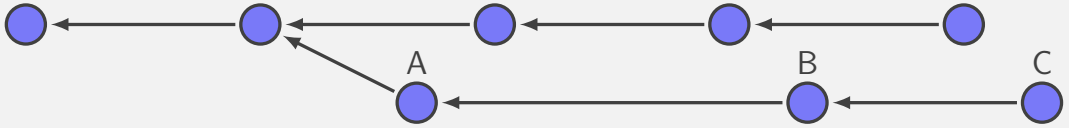
- Approach 2: no merge



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

- Approach 2: no merge



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

- Approach 2: no merge

- Drawbacks:
  - In case of conflict, they have to be resolved by the developer merging into upstream (possibly after code review)
  - Not always applicable (e.g. "I need this new upstream feature to continue working")

Merging With Upstream

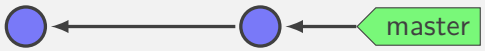
Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

- Approach 3: rebase (`git rebase` or `git pull --rebase`)

Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

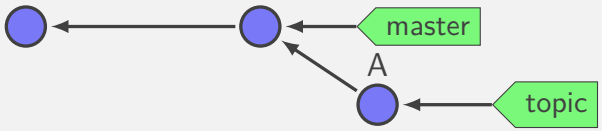
- Approach 3: rebase (`git rebase` or `git pull --rebase`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

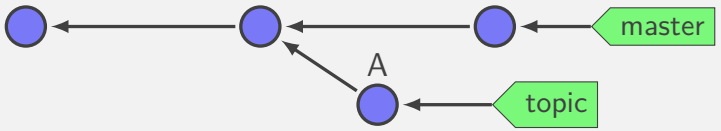
- Approach 3: rebase (`git rebase` or `git pull --rebase`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

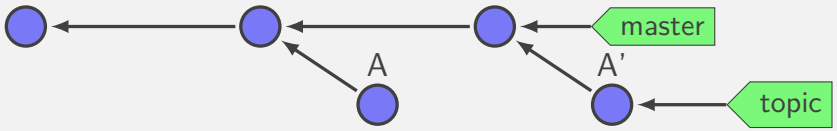
- Approach 3: rebase (`git rebase` or `git pull --rebase`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

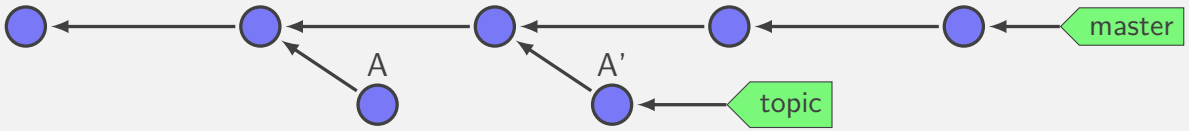
- Approach 3: rebase (`git rebase` or `git pull --rebase`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

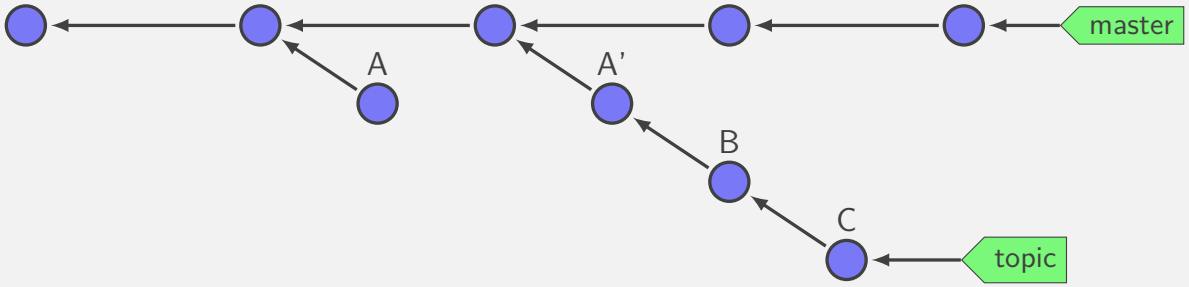
- Approach 3: rebase (`git rebase` or `git pull --rebase`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

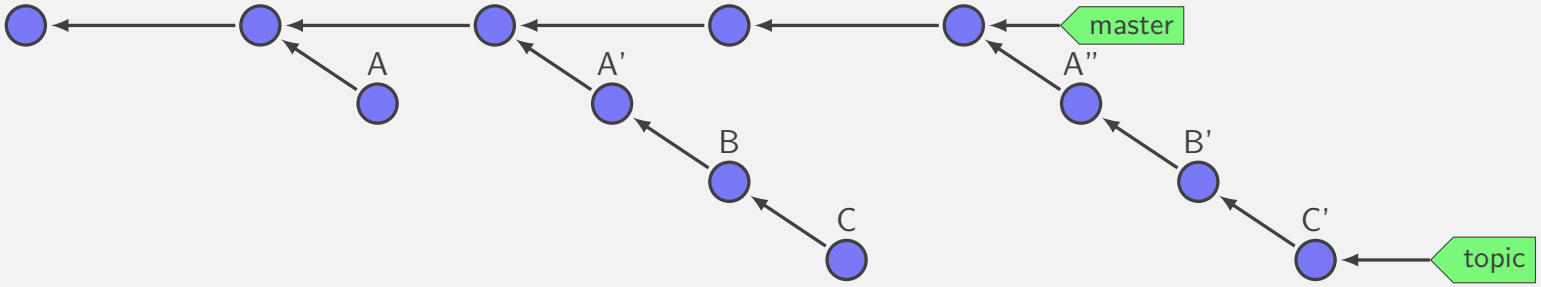
- Approach 3: rebase (`git rebase` or `git pull --rebase`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

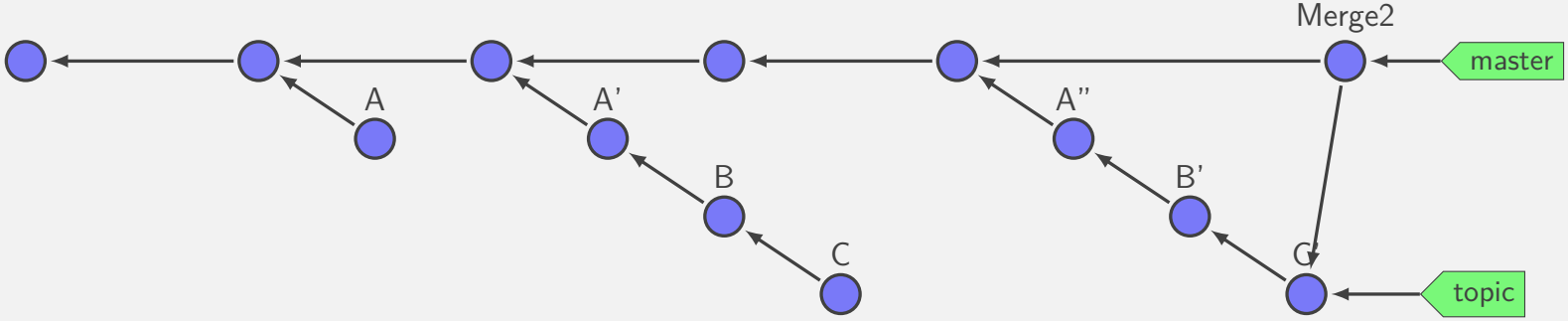
- Approach 3: rebase (`git rebase` or `git pull --rebase`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

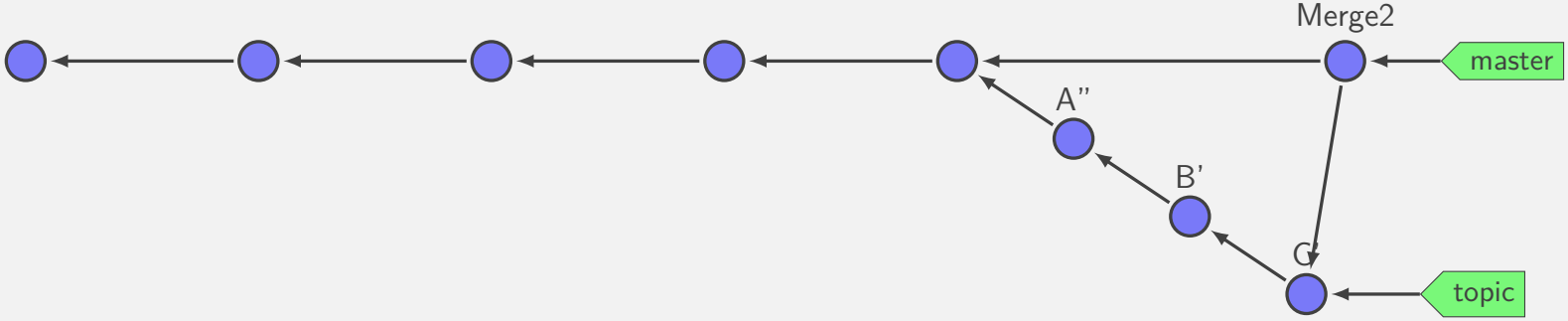
- Approach 3: rebase (`git rebase` or `git pull --rebase`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

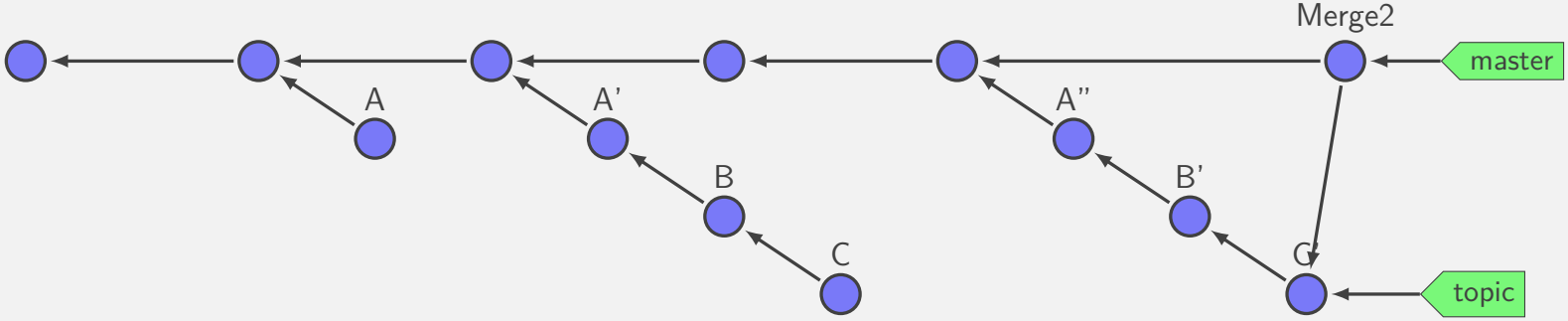
- Approach 3: rebase (`git rebase` or `git pull --rebase`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

- Approach 3: rebase (`git rebase` or `git pull --rebase`)



- Drawbacks: rewriting history implies:
  - A', A'', B', C' probably haven't been tested (never existed on disk)
  - What if someone branched from A, A', B or C?
  - Basic rule: don't rewrite published history

Rewriting history with `rebase -i`

- `git rebase`: take all your commits, and re-apply them onto upstream
- `git rebase -i`: show all your commits, and asks you what to do when applying them onto upstream.

Rewriting history with `rebase -i`

```
pick ca6ed7a Start feature A
pick e345d54 Bugfix found when implementing A
pick c03fffc Continue feature A
pick 5bdb132 Oops, previous commit was totally buggy

# Rebase 9f58864..5bdb132 onto 9f58864
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#

# Note that empty commits are commented out
```

`git rebase -i` commands (1/2)

p, pick use commit (by default)

r, reword use commit, but edit the commit message
Fix a typo in a commit message

e, edit use commit, but stop for amending
⇝ Once stopped, use `git add -p`, `git commit --amend`,...

s, squash use commit, but meld into previous commit

f, fixup like "squash", but discard this commit's log message
⇝ Very useful when polishing a set of commits (before or after review): make a bunch of short fixup patches, and squash them into the real commits. No one will know you did this mistake ;-).

`git rebase -i` commands (2/2)

x, exec run command (the rest of the line) using shell

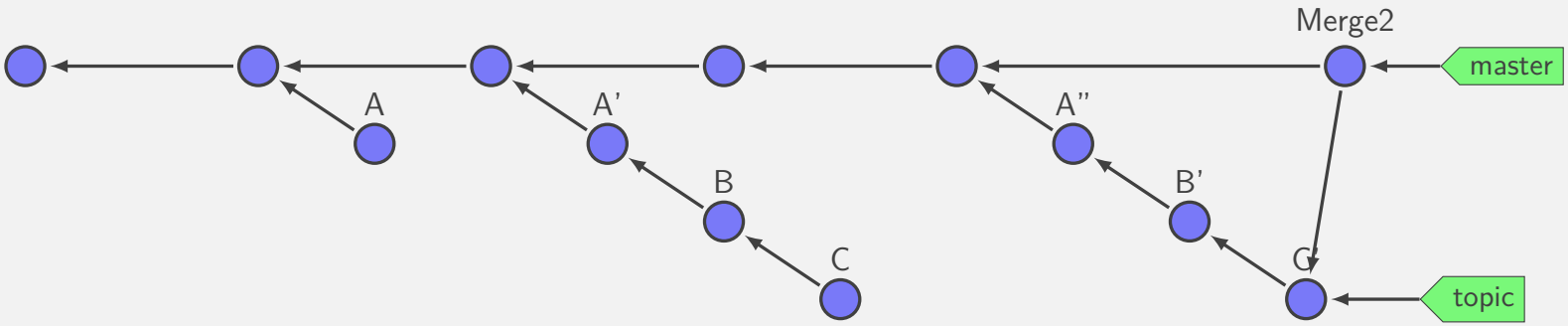
- Example: `exec make check`. Run tests for this commit, stop if test fail.
- Use `git rebase -i --exec 'make check'`[2] to run `make check` for each rebased commit.

[2] Implemented by Ensimag students!

Repairing mistakes: the reflog

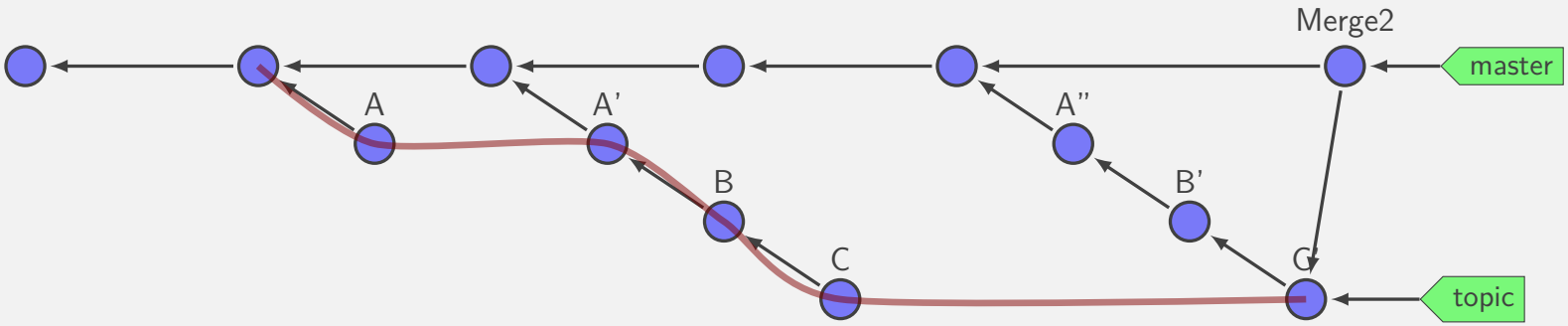
Git's reference journal: the reflog

- Remember the history of local refs.
- $\neq$ ancestry relation.



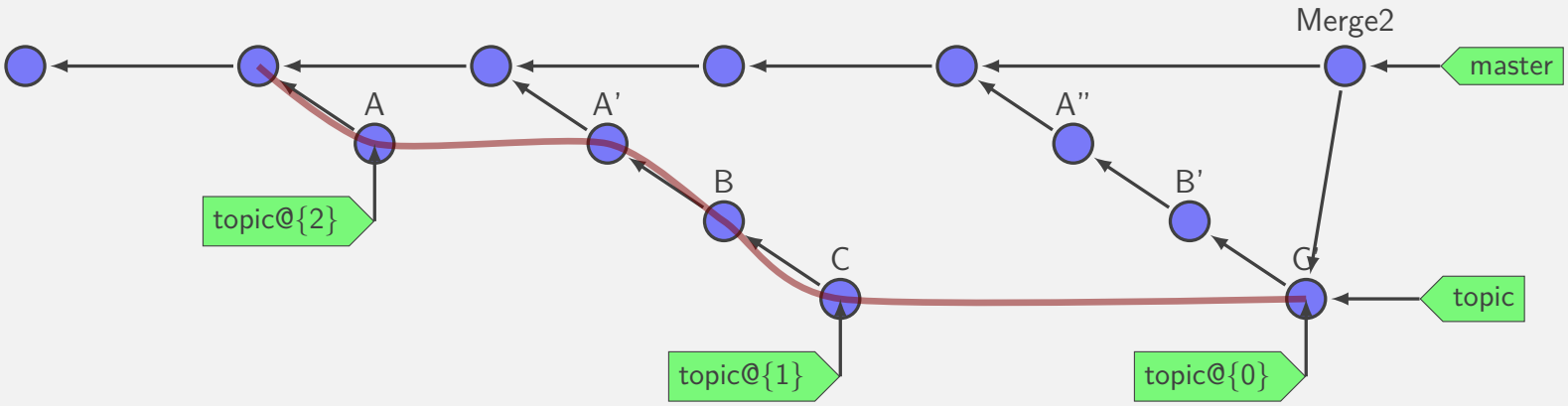
Git's reference journal: the reflog

- Remember the history of local refs.
- $\neq$ ancestry relation.



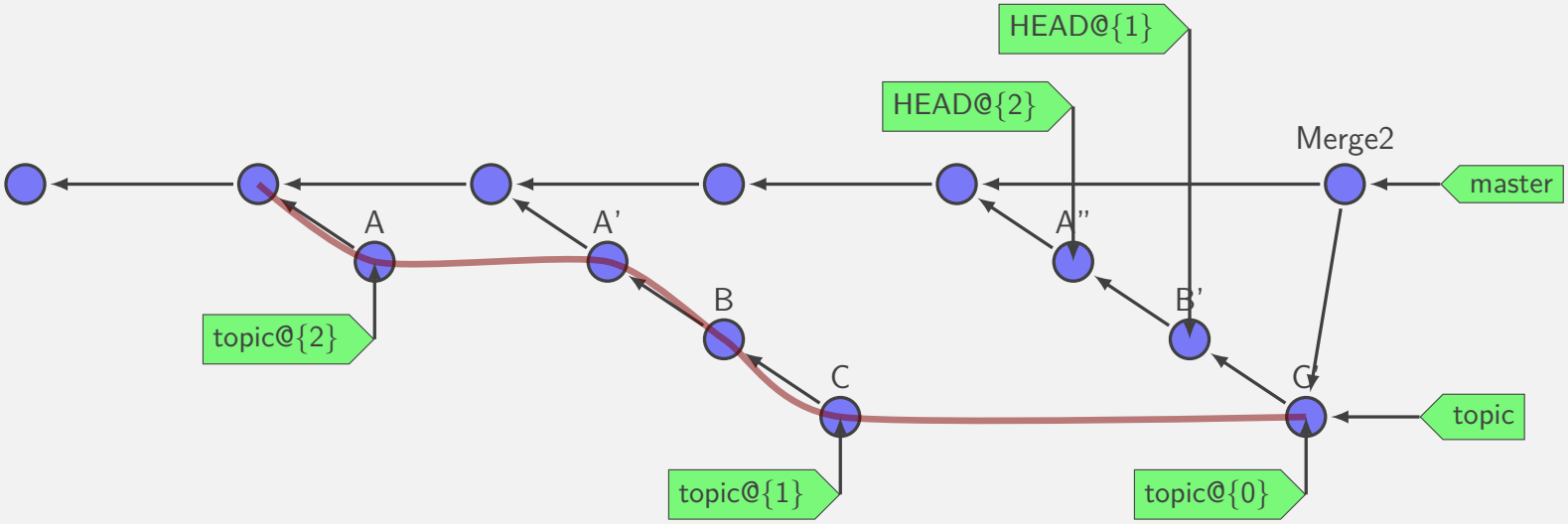
Git's reference journal: the reflog

- Remember the history of local refs.
- $\neq$ ancestry relation.



Git's reference journal: the reflog

- Remember the history of local refs.
- $\neq$ ancestry relation.



- *ref*`@{`*n*`}`: where *ref* was before the *n* last ref update.
- *ref*`~`*n*: the *n*-th generation ancestor of *ref*
- *ref*`^`: first parent of *ref*
- `git help revisions` for more

Branches and tags and workflows by examples

Branches and Tags in Practice

- Create a local branch and check it out:

  `git checkout -b` *`branch-name`*

- List local branches:

  `git branch`

- List all branches (including remote-tracking):

  `git branch -a`

- Create a tag:

  `git tag` *`tag-name`*

- Switch to a branch, a tag, or a commit:

  `git checkout` *`branch-name/tag-name/commit`*

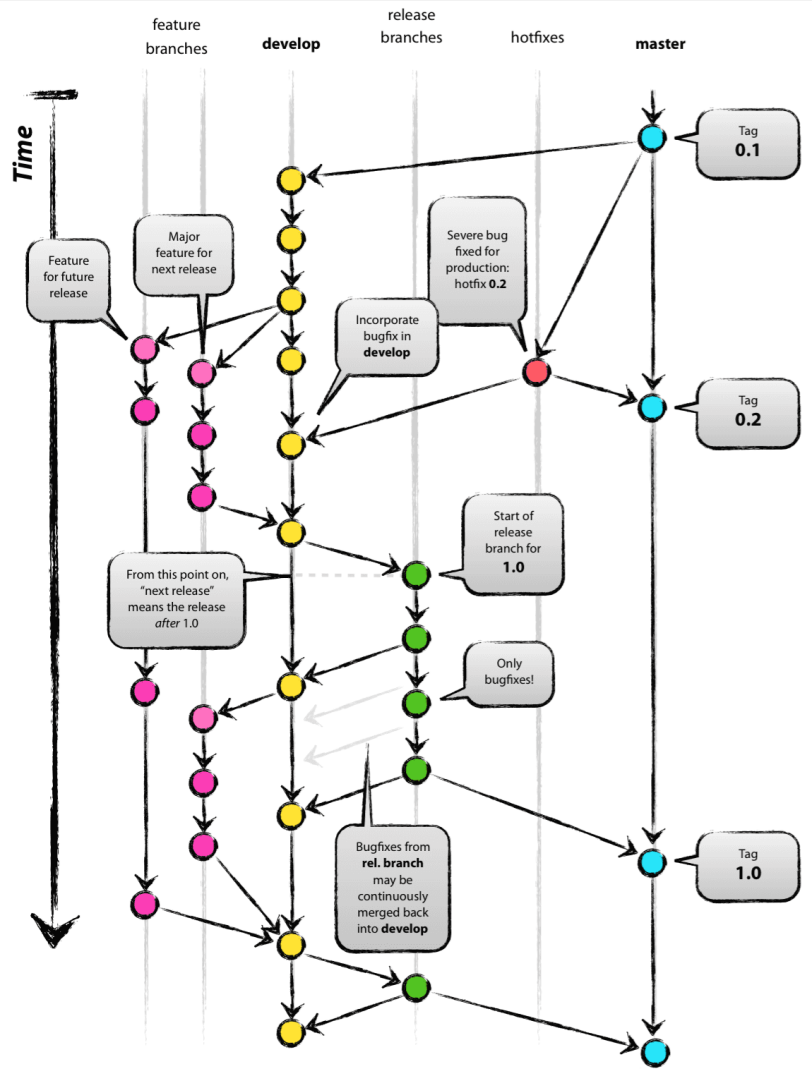
Branches

Branching exists in most VCS. The goal is to keep track separately of the various parallel works done on an ongoing software development.

This is Git killing feature Git branch management (and merge) is lightweight and efficient. Thus, it is quite easy to split the work among the developers and still let them work easily together toward the common goal.

But it is not a magic bullet: the splitting of the work should be handled with care.

Per topic branches



- *master* branch contain release versions
- *develop* branch contain ongoing development
- *release* branch to prepare new release in master
- *hotfix* branch for bug correction of release version
- *feature* branch to develop new features

[Vincent Driessen,

`http://nvie.com/posts/a-successful-git-branching-model/`]

Code: Branches 1/8 (tag, creation)

The repository will now be developed using "git-flow" model. Add a local tag, and a develop branch.

```
you@laptop$ git status
you@laptop$ git tag -a v1.0 # add message
you@laptop$ git switch -c develop # git checkout -b develop
you@laptop$ git branch
* develop
master
```

At each step, you must visualize the history with one of the two following commands:

```
you@laptop$ git lg # alias log --all --graph --oneline
you@laptop$ gitk --all
```

Code: Branches 2/8 (creation, commit, checkout)

Add two topic branches and one commit in each branch.

```
1  git checkout -b topic1
2  git status # topic1 branch at develop head
3  emacs myfile.txt # add few lines
4  git add -p myfile.txt
5  git commit -m "bad msg 3"
6  git checkout develop
7  git checkout -b topic2 # start at develop
8  emacs myfile.txt # add few lines
9  git add -p myfile.txt
10 git commit -m "bad msg 4"
11 git branch
12 gitk --all
```

Code: Branches 3/8 (merge with and without fast-forward)

Merge the topic1 branch in the develop branch. Add a commit in topic1 branch and merge it without fast-forwarding.

```
1  git checkout develop
2  git merge topic1
3  gitk --all
4  git checkout topic1
5  emacs myfile.txt # add few lines
6  git add -p
7  git commit
8  git checkout develop
9  git merge --no-ff topic1
10 gitk --all
```

Exercice: Branches 4/8 (rebase)

Rebase the *topic2* branch to get the new developments of *develop*.

```
1 git checkout topic2
2 git rebase develop # should give a conflict !
3 emacs myfile.txt
4 git add myfile.txt
5 git rebase --continue
6 gitk --all
```

Code: Branches 5/8 (stash)

Add some modifications in the *topic2*. Stash them.

```
# git checkout topic2
emacs myfile.txt # add few lines. DO NOT COMMIT
git status
git stash
git status
git stash list
```

Code: Branches 6/8 (add a hotfix)

Add a hotfix to the master v1.0 commit in a new *hotfix1* branch. Create a v1.1 commit in master merging the hotfix. Merge it in *develop* too.

```
1  git checkout v1.0
2  git checkout -b hotfix1
3  emacs myfile.txt # add few lines
4  git add -p
5  git commit
6  gitk --all
7  git checkout master
8  git merge --no-ff hotfix1
9  git tag -a v1.1
10 git checkout develop
11 git merge --no-ff hotfix1 # should conflict
12 git add myfile.txt
13 git commit
```

Code: Branches 7/8 (end the devel of topic2)

Return to the *topic2* branch. Rebase it on *develop*. Pop the stash. Add some modifications and commit them. Merge them in *develop*.

```
1  git checkout topic2
2  git rebase develop # should conflict
3  git add myfile.txt
4  git rebase --continue
5  git stash list
6  git stash pop # may conflict or not
7  emacs myfile.txt
8  git add -p
9  git commit
10 git checkout develop
11 git merge --no-ff topic2
12 gitk --all
```

Code: Branches 8/8 (Get a new master version)

Get a new version v2.0 of *master* including everything plus few modifications. Update also the *develop* branch with the small modifications.

```
# git checkout develop
git checkout -b release2
emacs myfile.txt # few modifications
git add -p
git commit
git checkout master
git merge --no-ff release2
git tag -a v2.0
git checkout develop
git merge --no--ff release2
gitk --all
```

Common alternative branching models

Flash5 All features branches start from **master** (the major difference with Driessen workflow). Everything is merged freely in **develop**. Extensive use of Continuous Integration (CI) in a **staged** branch. Public release in **master**

Github Everything is modified in **master**. Extensive use of Continuous Integration (CI).

Gitlab branching model Everything is modified in **master** (development branch). Bug fix and extensive use of Continuous Integration (CI) on a **pre-prod** branch. The **production** branch is the release branch and starting point of hot-fixes.

Master branch and visibility

The **master** branch is the default branch in many tools and settings. The Driessen's model highlights the results of the project (public releases of the code).

Branching for software development studies

When the goal is to highlight your software development history, to get good grades, we advices the use of **master** as the *develop branch.*

If you need a final result branch, you may call it ***public****.*

Push, Pull, Fetch and Remote made easy

`git push` pushes, by default, only the current branch and the related objects. You may add, e.g., a branch name or `git push --all`. To push tags, `git push --tags`.

`git fetch` downloads remote branches on **origin** and all related objects to their histories.

`git pull` is exactly `git fetch` then `git merge` or `git rebase` for the current branch.

`git remote` command family manage the source and sink of every branch.

Concluding remarks

- Git **data model is simple**: SHA-1 and zipped (mostly text) files; Git branch and tag are just a file with a single line with a file name or a SHA-1
- Every **commands are local** save push, pull and fetch. You have to push each branch and tags to publish (or delete) them in a remote repository; Remote management (remove for sake of time) will be understandable now.
- A commit message should explain **WHY** (What, When, Who are already in the commit)
- A **clean history helps a lot** large project (easier debugging and regression testing)
- Git **branches are easy** to use (eg. parallel development)

Bonus: Configuration files

Git Configuration: Which Files

- 3 places
  - System-wide ("system"): `/etc/gitconfig`
  - User-wide ("global"): `~/.gitconfig` or `~/.config/git/config`
  - Per-repository ("local", default): `$project/.git/config`
  - Per worktree (several checkout): `$project/.git/config.worktree`
- Precedence: per-repo overrides user-wide overrides system-wide.
- Not versionned by default, not propagated by `git clone`

Git Configuration: Syntax

- Simple syntax, key/value:

```
[section1]
# This is a comment
        key1 = value1 # comment as well
        key2 = value2
[section2 "subsection"]
        key3 = value3
```

- Semantics:
  - "`section1.key1` takes value `value1`"
  - "`section1.key2` takes value `value2`"
  - "`section2.subsection.key3` takes value `value3`"
- "section" and "key" are case-insensitive.

Some Useful Config Variables

- User-wide:

  user.name, user.email Who you are (used in `git commit`)

  core.editor Text editor to use for `commit`, `rebase -i`, ...

- Per-repo:

  remote.origin.url Where to fetch/push

Code: Aliases

Add an user alias for quick history display

Add "lg" alias for `log --all --graph --oneline` in the local git config

```
# Insert the definition in .git/config
# or git config alias.lg "log --all --graph --oneline"
$ cat .git/config
...
[alias]
lg = log --all --graph --oneline
...
# Use
$ git lg
*   a5da80c Merge branch 'master' into HEAD
|\
| * 048e8c1 bar
...
```

Documentation about Configuration

- `man git-config` : documents all configuration variables (> 350)
- Example:

```
user.name, user.email, author.name, author.email, committer.name,
committer.email
    The user.name and user.email variables determine what ends up in
    the author and committer field of commit objects. If you need the
    author or committer to be different, the author.name, author.email,
    committer.name or committer.email variables can be set. Also, all
    of these can be overridden by the GIT_AUTHOR_NAME,
    GIT_AUTHOR_EMAIL, GIT_COMMITTER_NAME, GIT_COMMITTER_EMAIL and EMAIL
    environment variables.
```

Bonus: (Git)Ignore files

Ignore Files: Why?

- Git needs to know which files to track (`git add`, `git rm`)
- You don't want to forget a `git add`
- ⇒ `git status` shows `Untracked files` as a reminder. Two options:
  - `git add` them
  - ask Git to ignore: add a rule to `.gitignore`
- Only impacts `git status` and `git add`.

Ignore Untracked Files: How?

- .gitignore file contain one rule per line:

```
# This is a comment

# Ignore all files ending with ~:
*~
# Ignore all files named 'core':
core
# Ignore file named foo.pdf in this directory:
/foo.pdf
# Ignore files in any auto directory:
auto/
# Ignore html file in subdir of any Doc directory:
Doc/**/*.html
```

Ignore Files: Where?

- User-wide: `~/.config/git/ignore`:
  - Example: your editor's file like `*~` or `.*.swp`
  - Don't disturb your co-workers with your personal preferences
  - Set once and for all
- Per-repo, not versionned: `.git/info/exclude`
  - Not very useful ;-)
- Tracked within the project (`git add` it): `.gitignore` in any directory, applies to this directory and subdirectories.
  - Generated files (especially binary)
  - Example: `*.o` and `*.so` for a C project
  - Share with people working on the same project

Code: .gitignore

Ignore the backup file of your favorite editors

```
emacs .gitignore
git status
git add .gitignore
git commit -m "ignore emacs backups"
```

Bonus: (Git)LFS and annex

(Very) Large Binary Files: How to ?

- Git traces the file content (hash)
- For common large files (zip, crypto, jpeg/mpeg, mp3), often tiny modifications modify all the content, thus the git delta storage is inefficient !
- ⇒ `git lfs` and `git annex` store the hash of the file, but not the file
- File storage, duplication, locking and transfer are managed by the tool: LFS, central Git repository, supported by gitlab and github; Annex: distributed model, fine grain transfer, LFS as a remote