Git training

Summary

1. Overview of GIT
2. Working locally
3. Interacting with a remote repository
4. Branching & merging
5. Third-party contribution
6. Hands-on

Overview of GIT

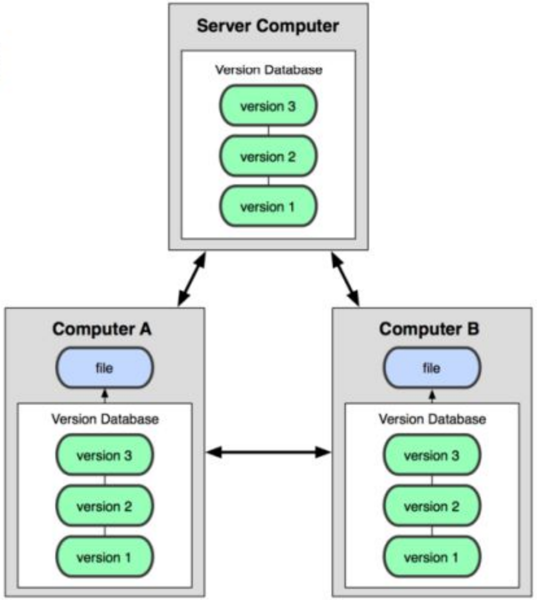
About Git

- Created by Linus Torvalds, creator of Linux, in 2005
  - Came out of Linux development community
  - Designed to do version control on Linux kernel

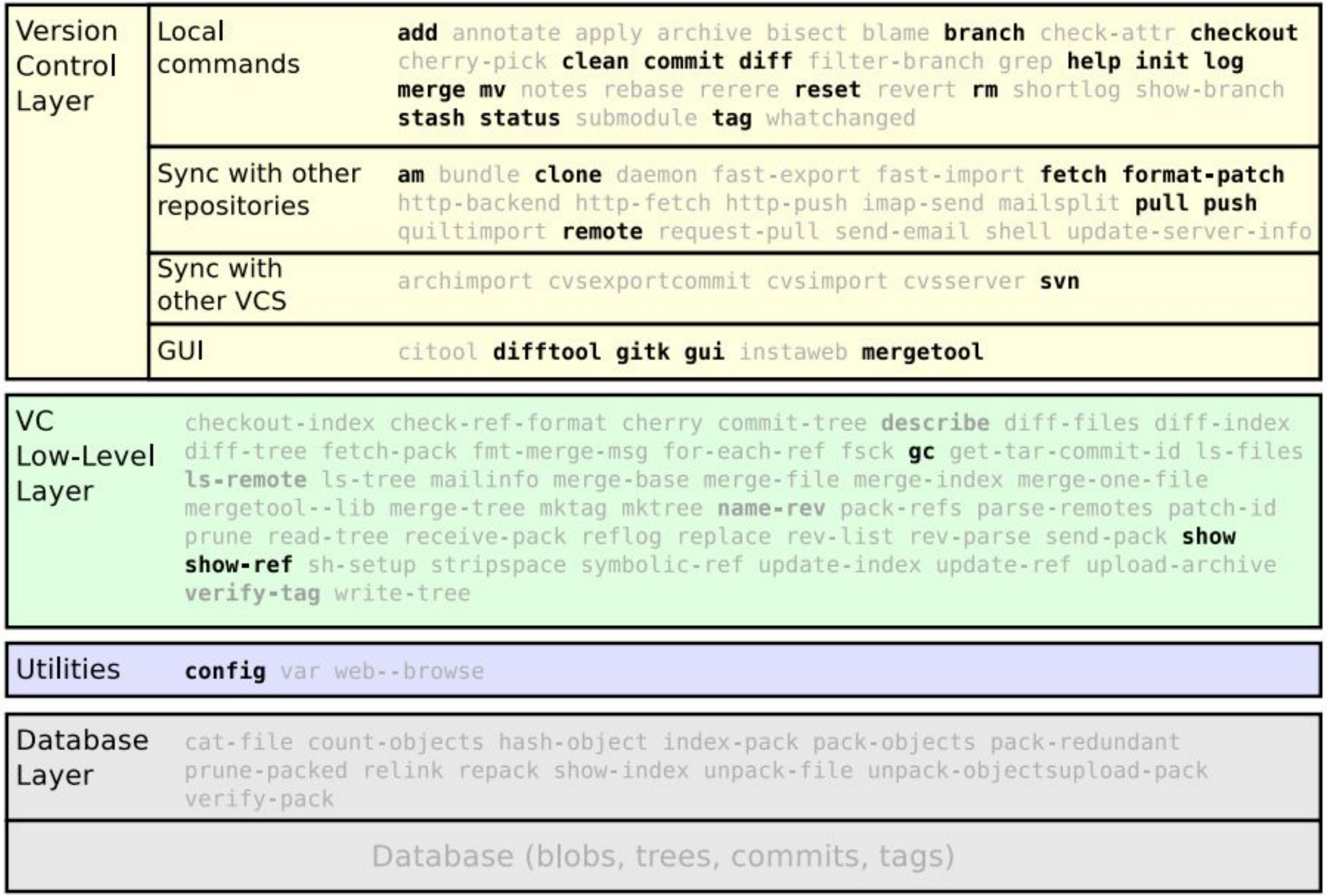
- Goals of Git:
  - Speed
  - Support for non-linear development (thousands of parallel branches)
  - Fully distributed
  - Able to handle large projects efficiently
  - (A "git" is a cranky old man. Linus meant himself.)

Distributed VCS (Git)

- In git, mercurial, etc., you don't "checkout" from a central repo
  - you "clone" it and "pull" changes from it
- Your local repo is a complete copy of everything on the remote server
  - yours is "just as good" as theirs
- Many operations are local:
  - check in/out from local repo
  - commit changes to local repo
  - local repo keeps version history
- When you're ready, you can "push" changes back to server



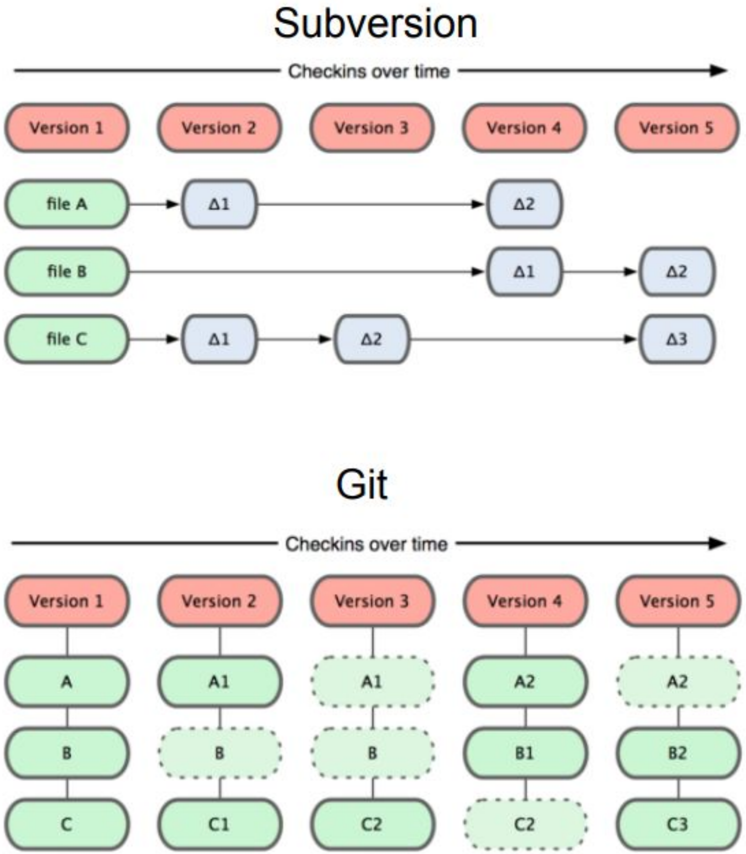
Git Commands



Version Control Layer	Local commands	**add** annotate apply archive bisect blame **branch** check-attr **checkout** cherry-pick **clean** **commit** **diff** filter-branch grep **help** **init** **log** **merge** **mv** notes rebase rerere **reset** revert **rm** shortlog show-branch **stash** **status** submodule **tag** whatchanged
	Sync with other repositories	**am** bundle **clone** daemon fast-export fast-import **fetch** **format-patch** http-backend http-fetch http-push imap-send mailsplit **pull** **push** quiltimport **remote** request-pull send-email shell update-server-info
	Sync with other VCS	archimport cvsexportcommit cvsimport cvsserver **svn**
	GUI	citool **difftool** **gitk** **gui** instaweb **mergetool**
VC Low-Level Layer		checkout-index check-ref-format cherry commit-tree **describe** diff-files diff-index diff-tree fetch-pack fmt-merge-msg for-each-ref fsck **gc** get-tar-commit-id ls-files **ls-remote** ls-tree mailinfo merge-base merge-file merge-index merge-one-file mergetool--lib merge-tree mktag mktree **name-rev** pack-refs parse-remotes patch-id prune read-tree receive-pack reflog replace rev-list rev-parse send-pack **show** **show-ref** sh-setup stripspace symbolic-ref update-index update-ref upload-archive **verify-tag** write-tree
Utilities		**config** var web--browse
Database Layer		cat-file count-objects hash-object index-pack pack-objects pack-redundant prune-packed relink repack show-index unpack-file unpack-objectsupload-pack verify-pack
Database (blobs, trees, commits, tags)		

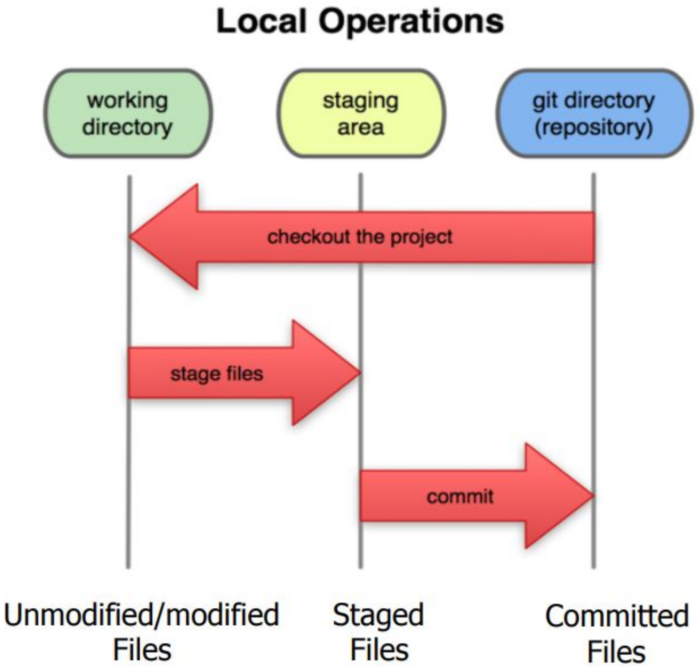
Git snapshots

- Centralized VCS like Subversion track version data on each individual file.
- Git keeps "snapshots" of the entire state of the project.
  - Each checkin version of the overall code has a copy of each file in it.
  - Some files change on a given checkin, some do not.
  - More redundancy, but faster.



Local git areas

- In your local copy on git, files can be:
  - In your local repo
    - (committed)
  - Checked out and modified, but not yet committed
    - (working copy)
  - Or, in-between, in a "staging" area
    - Staged files are ready to be committed.
    - A commit saves a snapshot of all staged state.

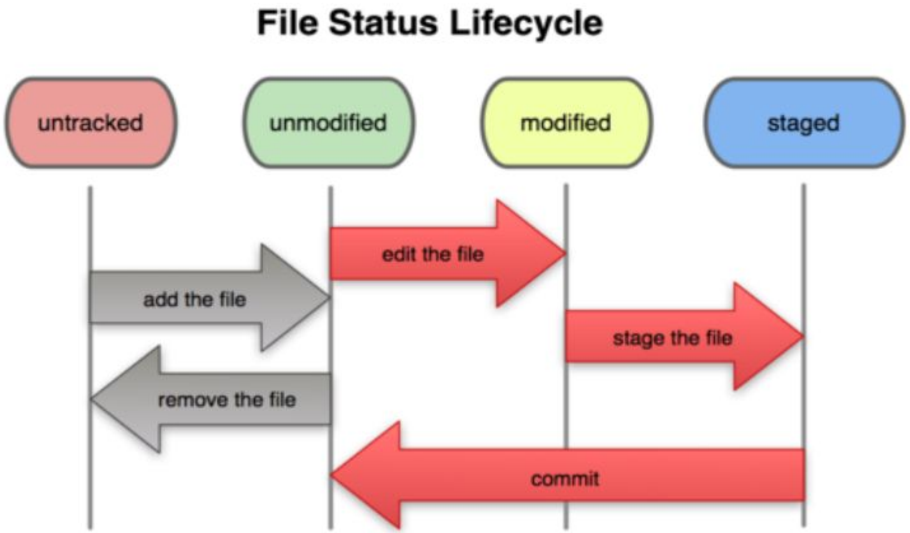


Basic Git workflow

- Modify files in your working directory.

- Stage files, adding snapshots of them to your staging area.

- Commit, which takes the files in the staging area and stores that snapshot permanently to your Git directory.



Git commit checksums

In Subversion each modification to the central repo increments the version # of the overall repo.

– In Git, each user has their own copy of the repo, and commits changes to their local copy of the repo before pushing to the central server.

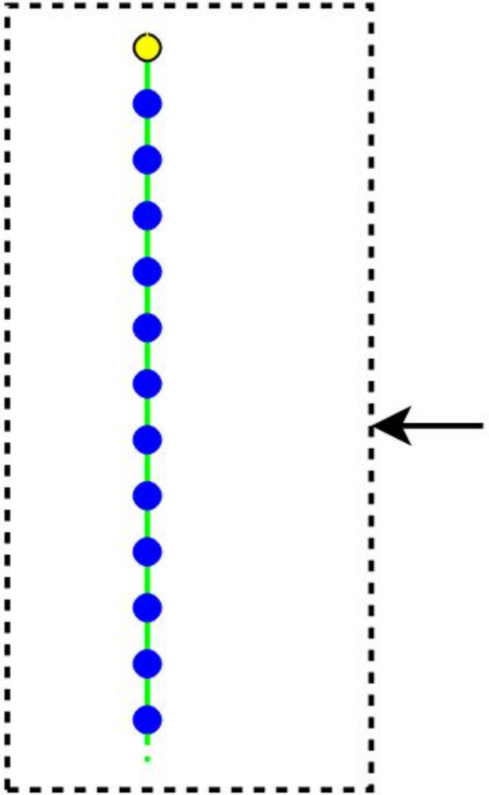
– So Git generates a unique **SHA-1** hash (40 character string of hex digits) for every commit.

– Refers to commits by this **ID** rather than a version number.

– Often we only see the first 7 characters:

- 1677b2d Edited first line of readme
- 258efa7 Added line to readme
- 0e52da7 Initial commit

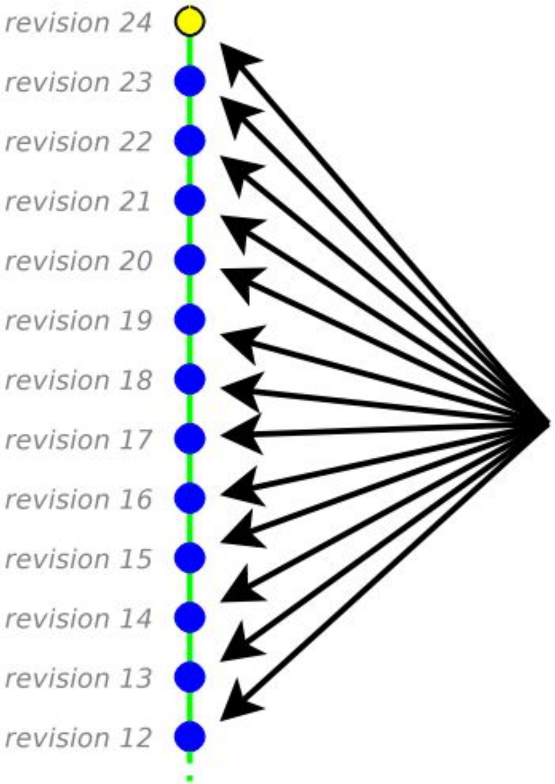
Git Terminology



The Repository

it contains the full history of your project (all revisions from the beginning)

Git Terminology

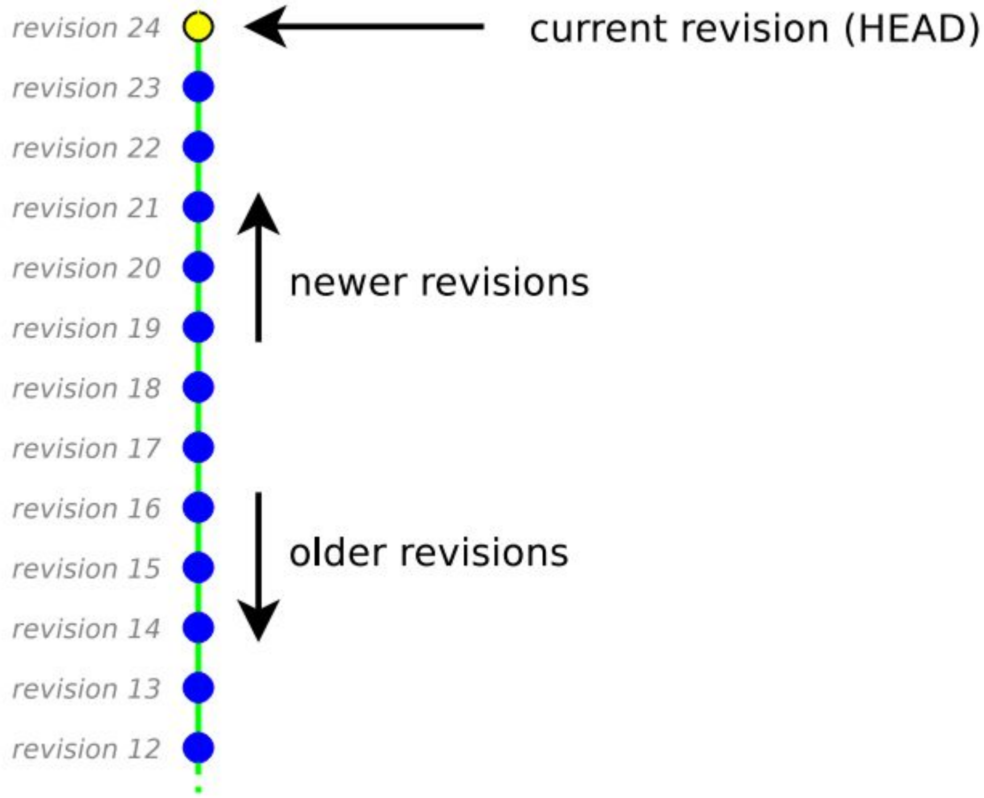


Revisions

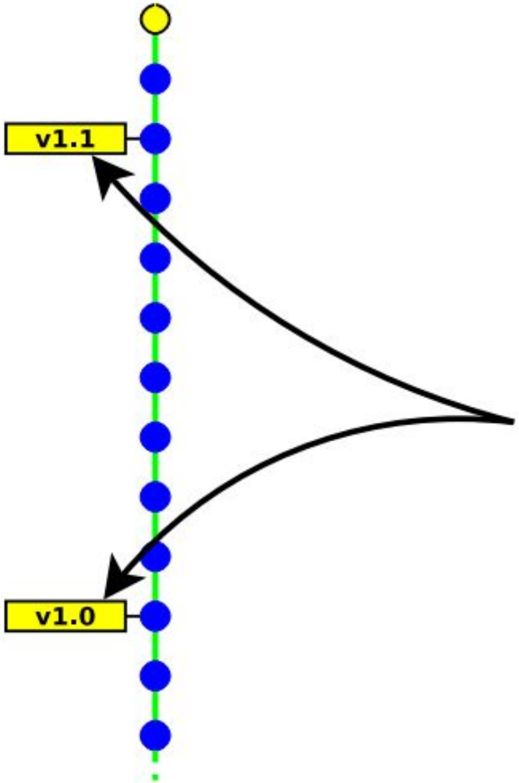
each revision:

- introduces changes from the previous revision
- has an identified author
- contains a textual message describing the changes

Git Terminology



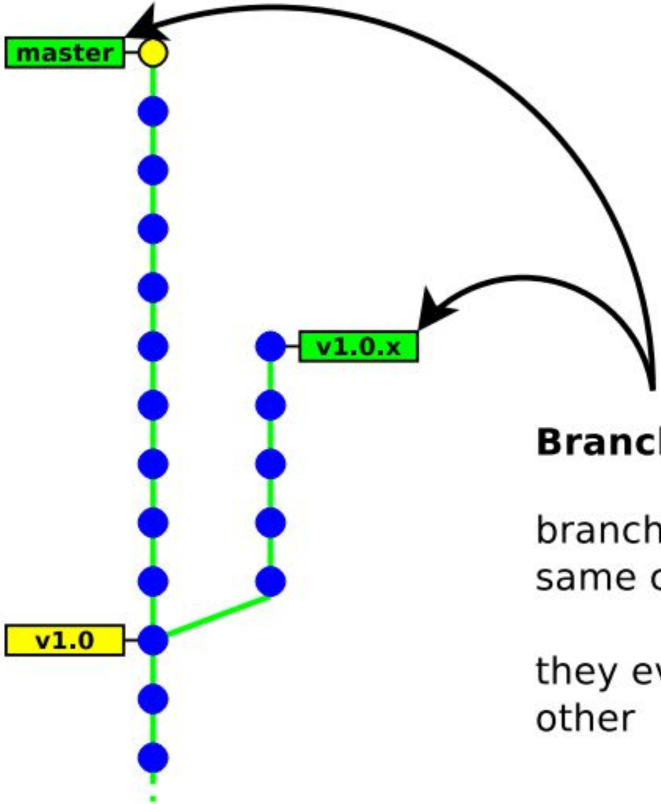
Git Terminology



Tags

a tag identifies a particular revision
(typically each release of the software)

Git Terminology

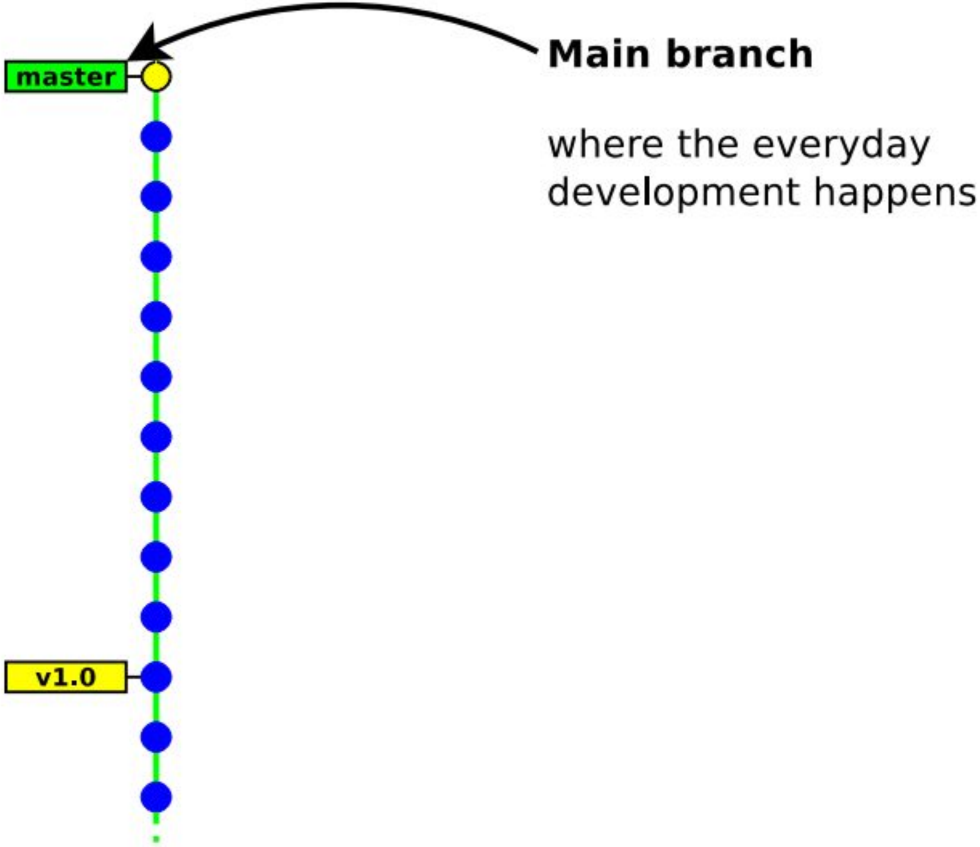


Branches

branches are different variants of the same collection of files

they evolve independently of each other

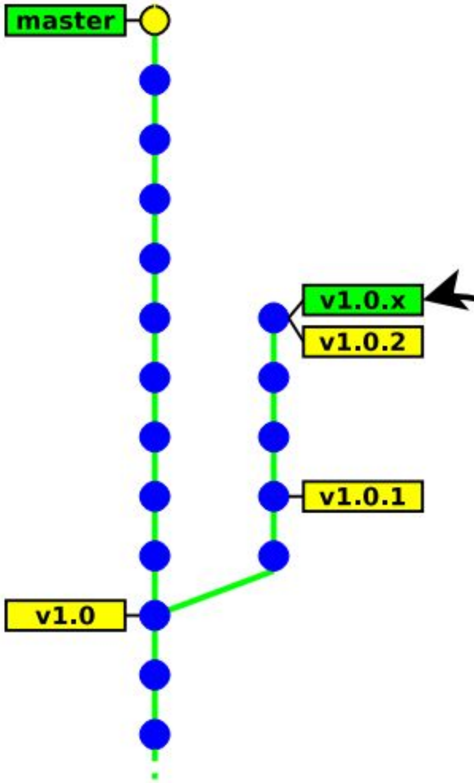
Git Terminology



Main branch

where the everyday development happens

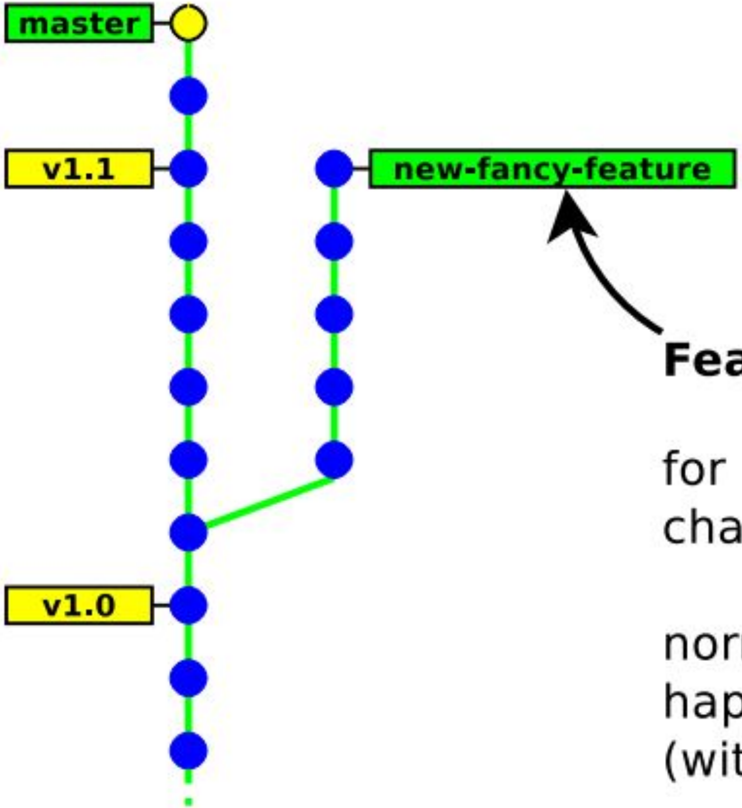
Git Terminology



Maintainance branch

to issue bug fixes for older releases of the software

Git Terminology

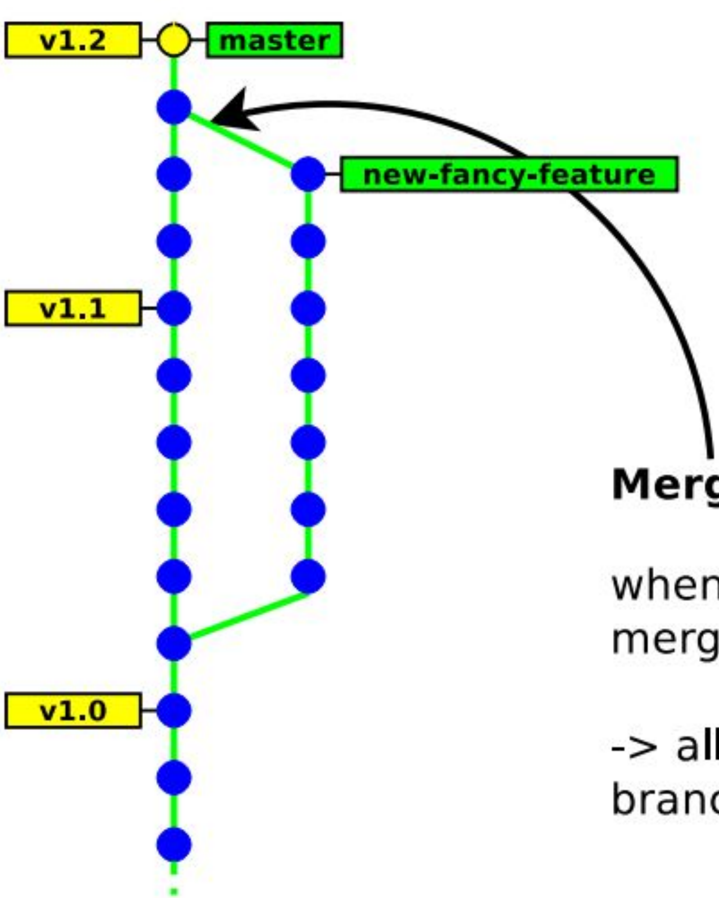


Feature branch

for a new feature requiring intrusive changes in the code

normal development continues to happen in the master branch (without disturbance)

Git Terminology

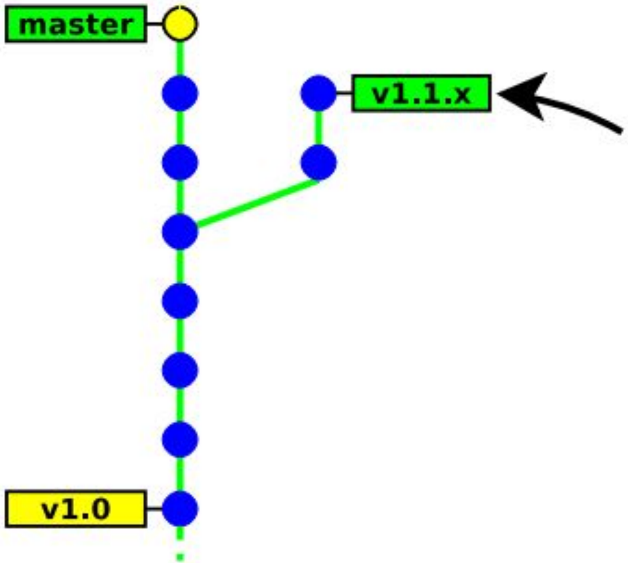


Merging

when the new feature is ready, it can merged back into the master branch

-> all changes done in the feature branch are imported

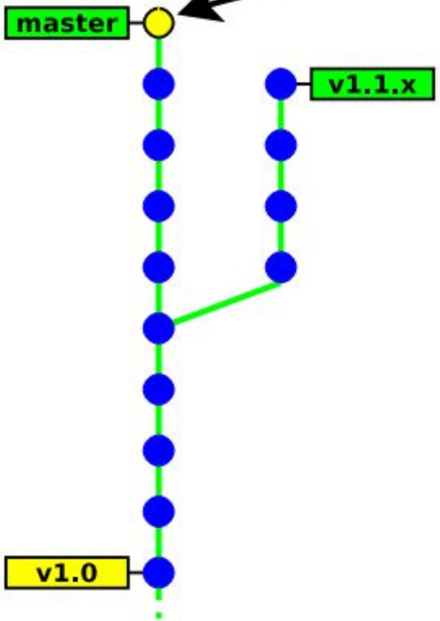
Git Terminology



Release branch

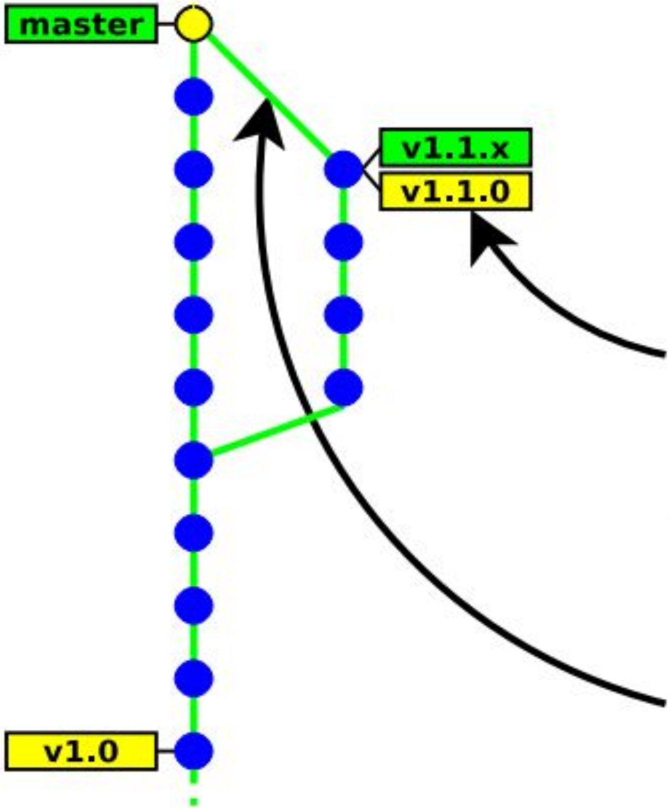
to prepare the next release
- the code is frozen
- only bug fixes are accepted

Git Terminology



meanwhile developments continue in the master branch

Git Terminology

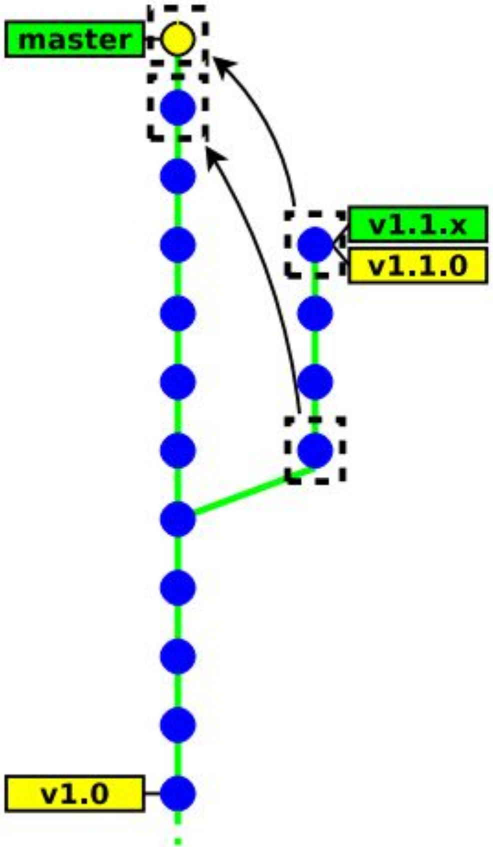


New release

when the code is ready, the new version is released

- the release branch becomes a maintainance branch
- bug fixes can be merged back into the main branch

Git Terminology



Cherry picking

it may not be desirable to merge all the commits into the other branch (e.g. a bug may need a different fix)

-> it is possible to apply each commit individually

Working locally

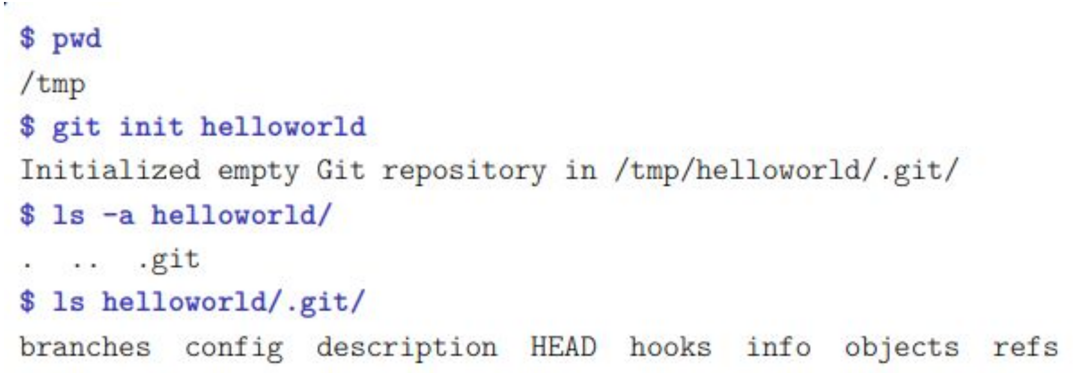
Create a new repository

```
git init myrepository
```

This command creates the directory myrepository.

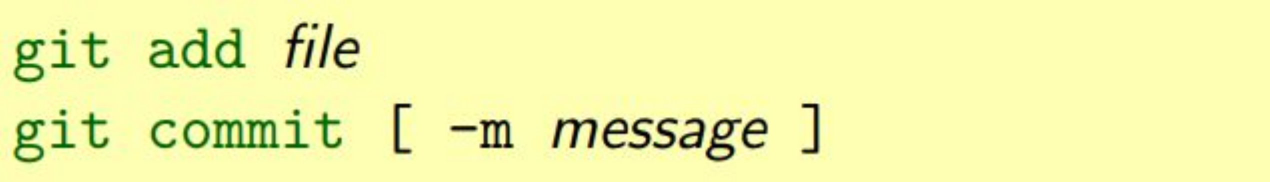
- The repository is located in myrepository/.git
- The (initially empty) working copy is located in myrepository/

Note: The /.git/ directory contains your whole history, do not delete it



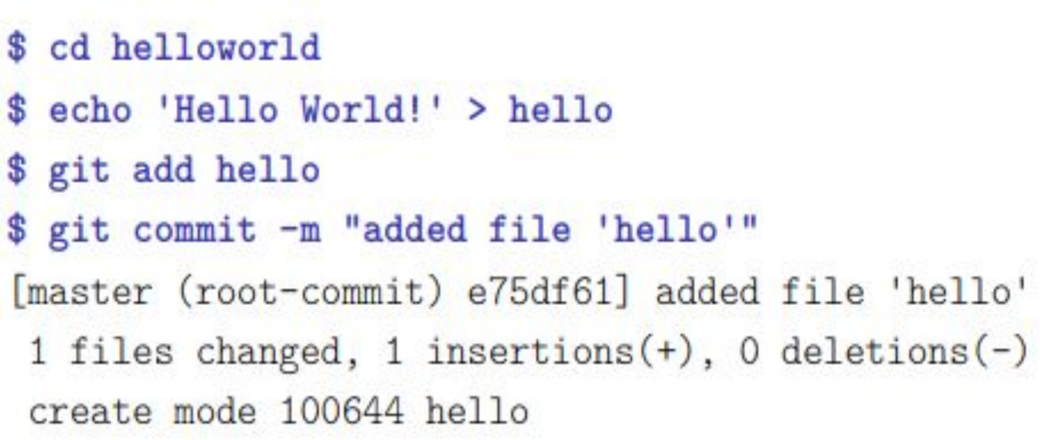
```
$ pwd
/tmp
$ git init helloworld
Initialized empty Git repository in /tmp/helloworld/.git/
$ ls -a helloworld/
.  ..  .git
$ ls helloworld/.git/
branches  config  description  HEAD  hooks  info  objects  refs
```

Commit your first files



```
git add file
git commit [ -m message ]
```

Note: “master” is the name of the default branch created by git init.



```
$ cd helloworld
$ echo 'Hello World!' > hello
$ git add hello
$ git commit -m "added file 'hello'"
[master (root-commit) e75df61] added file 'hello'
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 hello
```

The staging area (The “index”)

Usual version control systems provide two spaces:

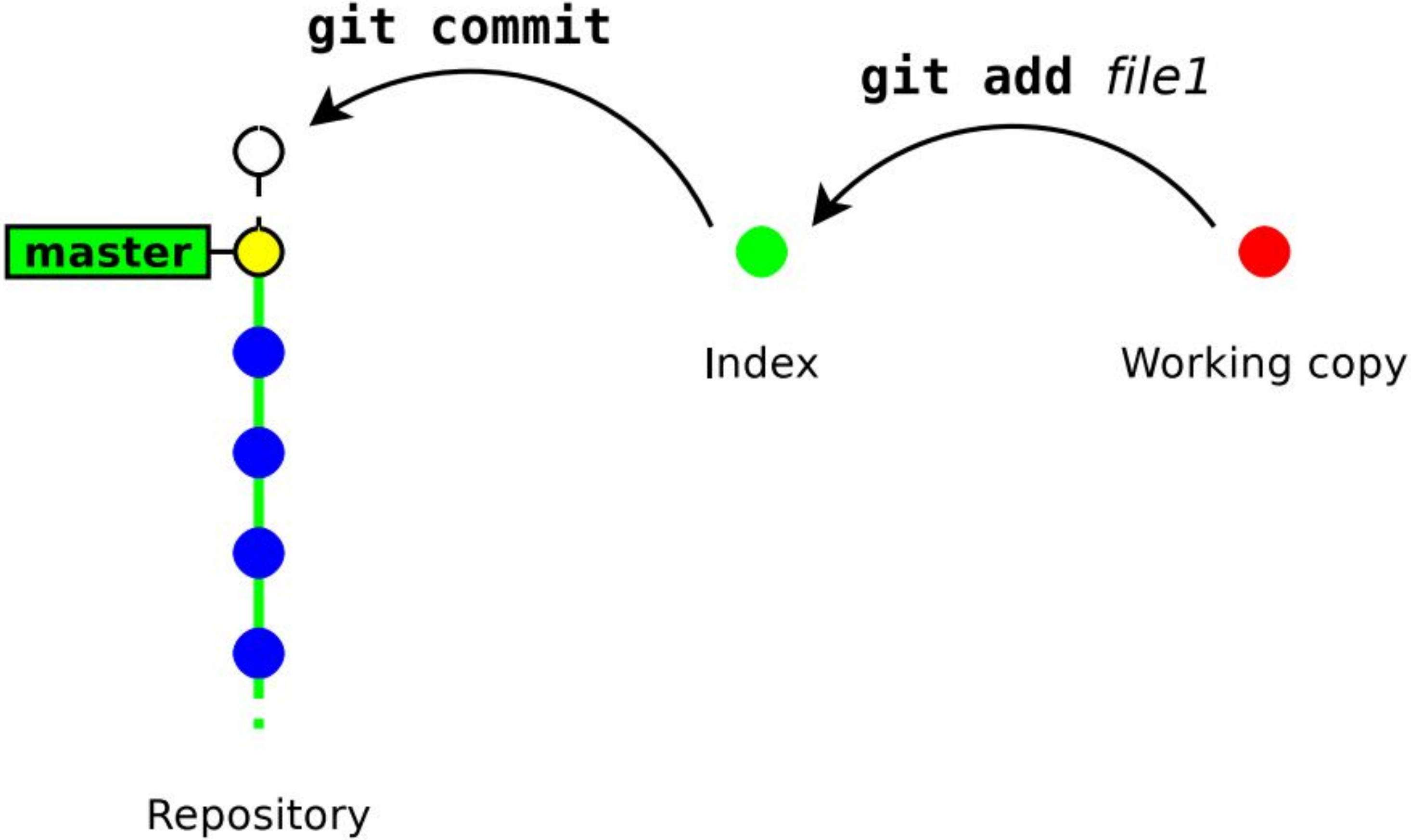
- The **repository** (the whole history of your project)
- The **working tree** (or **local copy**) (the files you are editing and that will be in the next commit)

Git introduces an intermediate space : the **staging area** (also called **index**)

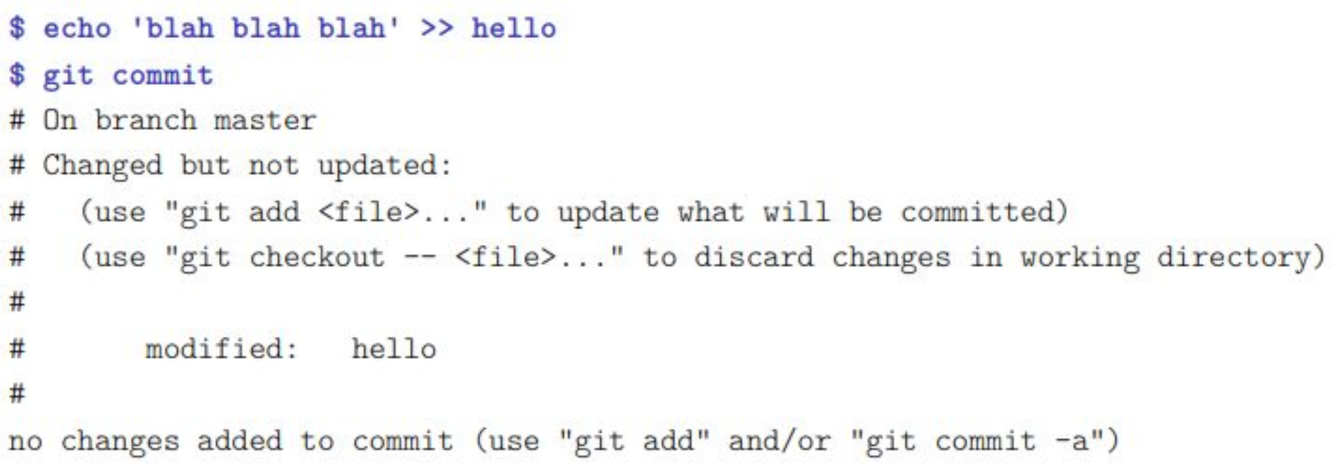
The index stores the files scheduled for the next commit:

- **git add** files → copy files into the index
- **git commit** → commits the content of the index

The staging area (aka the “index”)

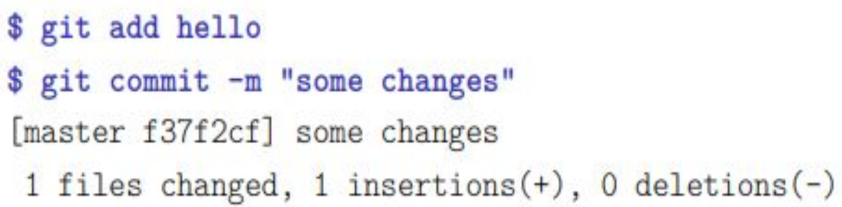


Updating a file



```
$ echo 'blah blah blah' >> hello
$ git commit
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   hello
#
no changes added to commit (use "git add" and/or "git commit -a")
```

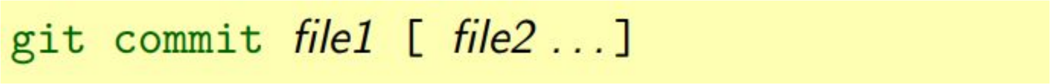
Git complains because the index is unchanged (nothing to commit) → We need to run git add to copy the file into the index



```
$ git add hello
$ git commit -m "some changes"
[master f37f2cf] some changes
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Bypassing the index (“partial commit”)

Running **git add** & **git commit** for every iteration is tedious. GIT provides a way to bypass the index.



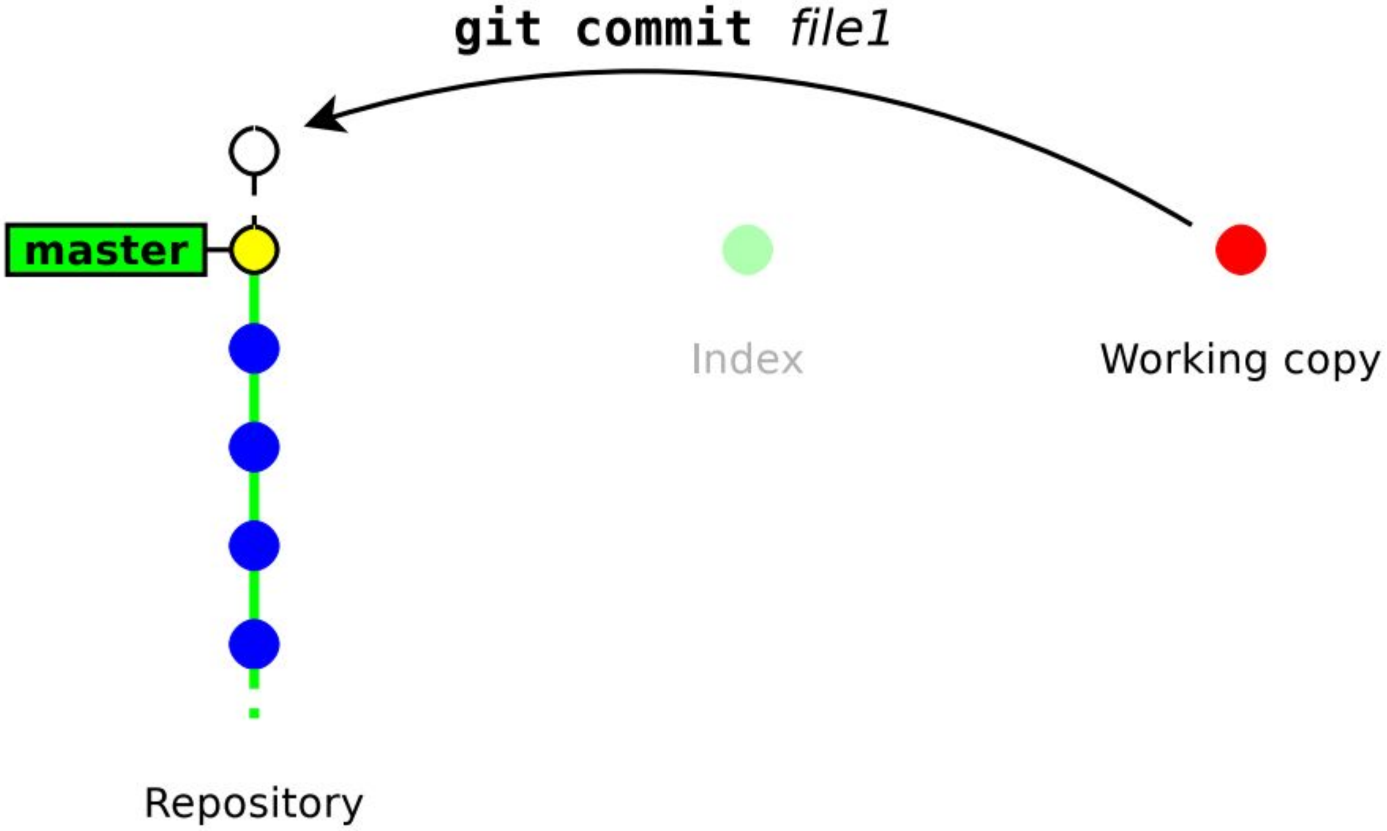
```
git commit file1 [ file2 ...]
```

This command commits files (or dirs) directly from the working tree.

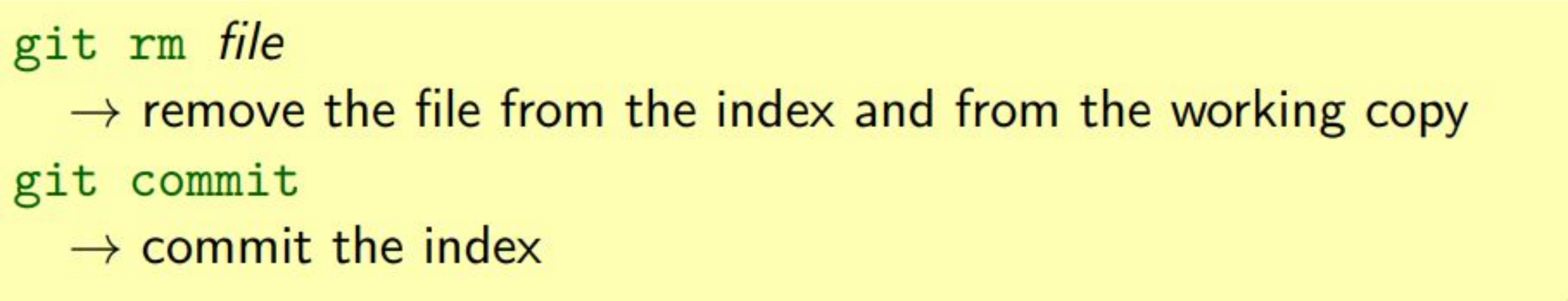
Note: when bypassing the index, GIT ignores new files:

- “**git commit** .” commits only files that were present in the last commit (updated files)
- “**git add .** && **git commit**” commits everything in the working tree (including new files)

Bypassing the index



Deleting files

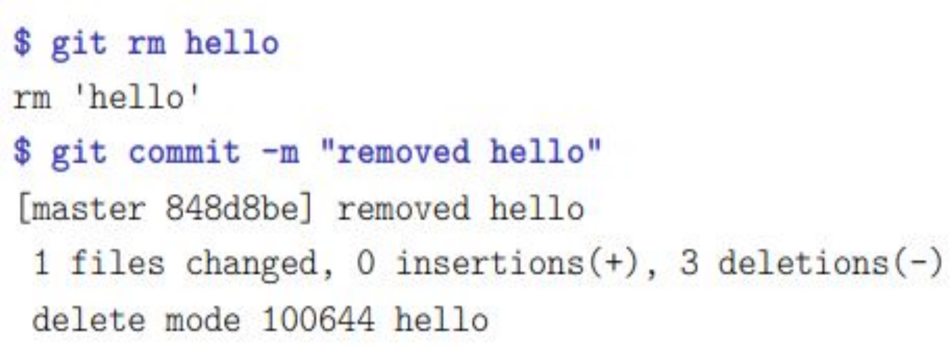


`git rm` *file*

→ remove the file from the index and from the working copy

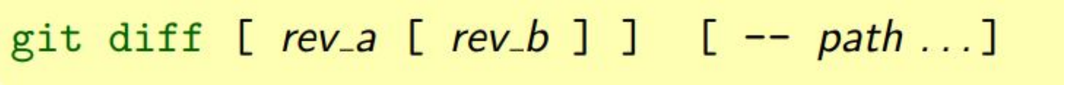
`git commit`

→ commit the index



```
$ git rm hello
rm 'hello'
$ git commit -m "removed hello"
[master 848d8be] removed hello
 1 files changed, 0 insertions(+), 3 deletions(-)
 delete mode 100644 hello
```

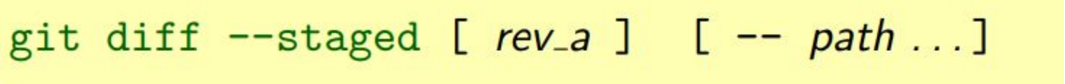
Showing differences



```
git diff [ rev_a [ rev_b ] ]  [ -- path ...]
```

→ shows the differences between two revisions **rev_a** and **rev_b** (in a format suitable for the *patch* utility)

- by default rev_a is the **index**
- by default rev_b is the **working_copy**

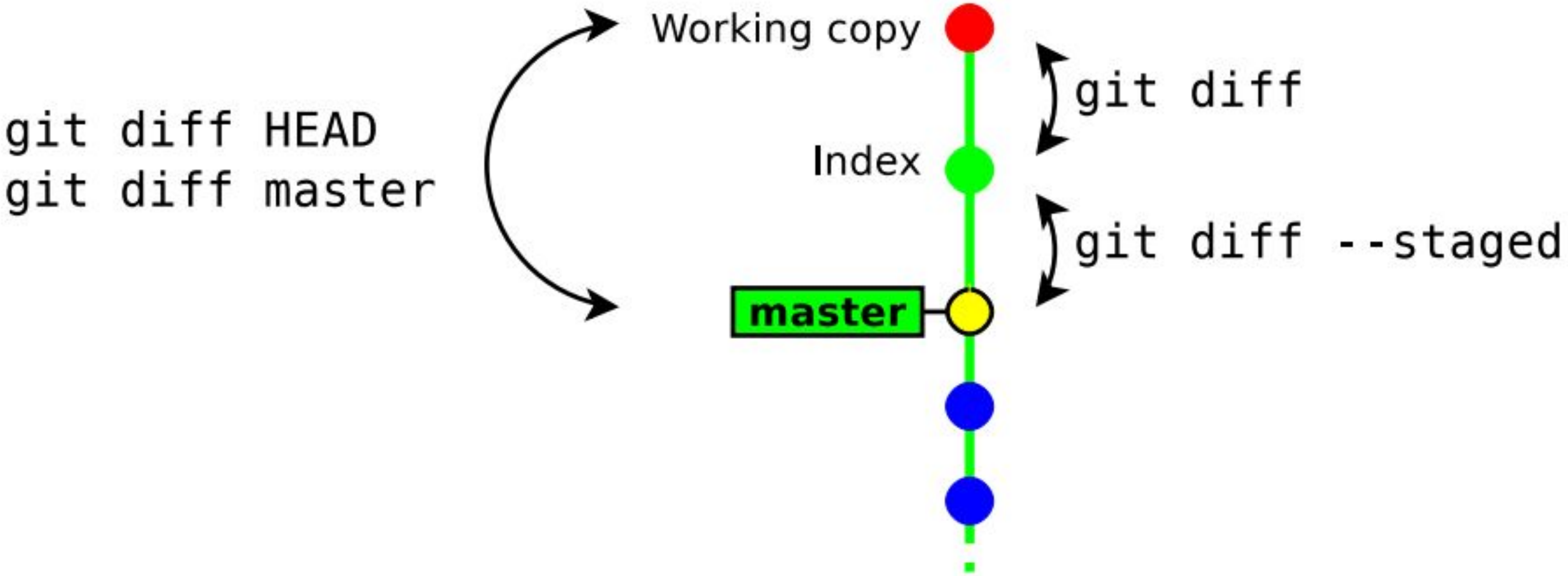


```
git diff --staged [ rev_a ]  [ -- path ...]
```

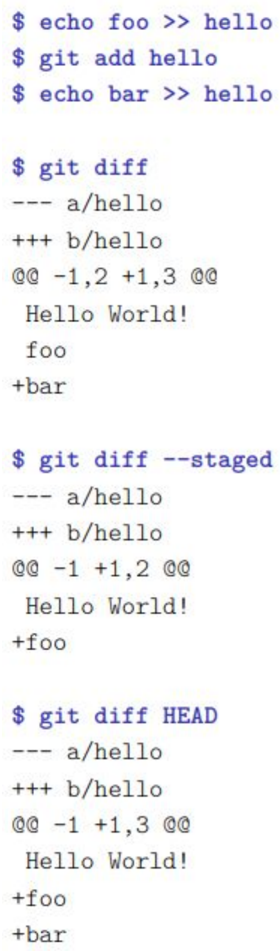
→ shows the differences between rev_a and the index

- by default **rev_a** is **HEAD** (a symbolic references pointing to the last commit)

About git diff and the index

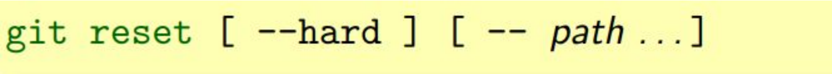


Diff example



```
$ echo foo >> hello
$ git add hello
$ echo bar >> hello

$ git diff
--- a/hello
+++ b/hello
@@ -1,2 +1,3 @@
 Hello World!
 foo
+bar

$ git diff --staged
--- a/hello
+++ b/hello
@@ -1 +1,2 @@
 Hello World!
+foo

$ git diff HEAD
--- a/hello
+++ b/hello
@@ -1 +1,3 @@
 Hello World!
+foo
+bar
```

Resetting changes

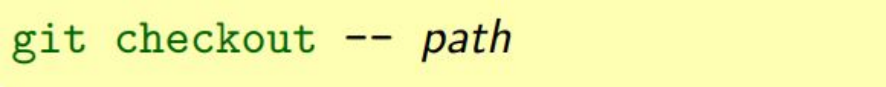


```
git reset [ --hard ] [ -- path ...]
```

git reset cancels the changes in the index (and possibly in the working copy)

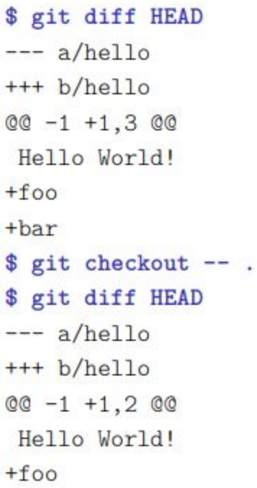
- **git reset** drops the changes staged into the index (it restores the files as they were in the last commit), but the working copy is left intact
- **git reset --hard** drops all the changes in the index and in the working copy

Resetting changes in the working copy



```
git checkout -- path
```

This command restores a file (or directory) as it appears in the index (thus it drops all unstaged changes)



```
$ git diff HEAD
--- a/hello
+++ b/hello
@@ -1 +1,3 @@
 Hello World!
+foo
+bar
$ git checkout -- .
$ git diff HEAD
--- a/hello
+++ b/hello
@@ -1 +1,2 @@
 Hello World!
+foo
```

Working remotely

How git handles remote repositories

- It is possible to work with multiple remote repositories

- Each remote repository is identified with a local alias. When working with a unique remote repository, it is usually named origin (default name used by git clone)

- Remote repositories are mirrored within the local repository

- Remote branches are mapped in a separate namespace:

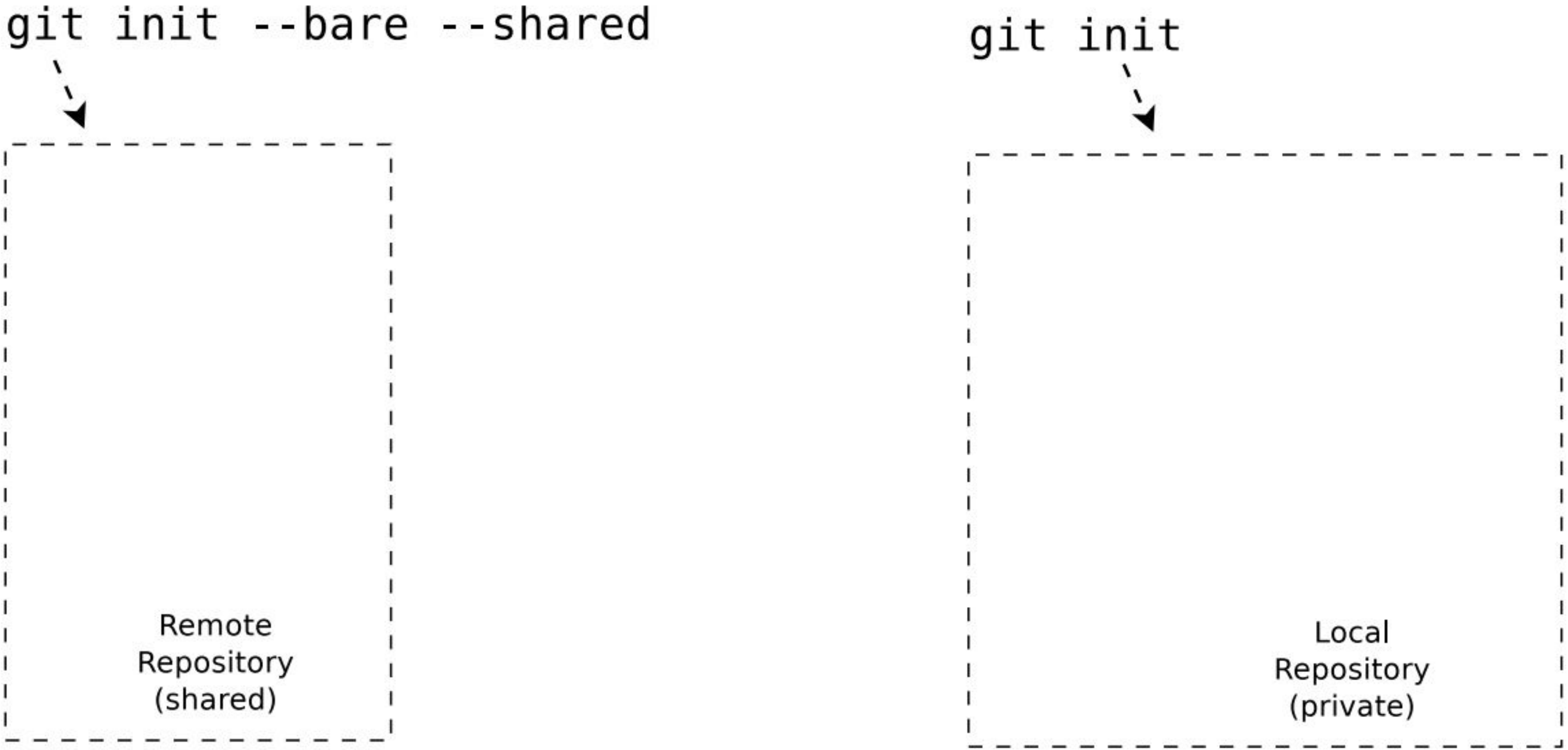
  remote/name/branch.

<u>Examples:</u>

- master refers to the local master branch

- remote/origin/master refers to the master branch of the remote repository named origin

Remote example



```
git init --bare --shared
```

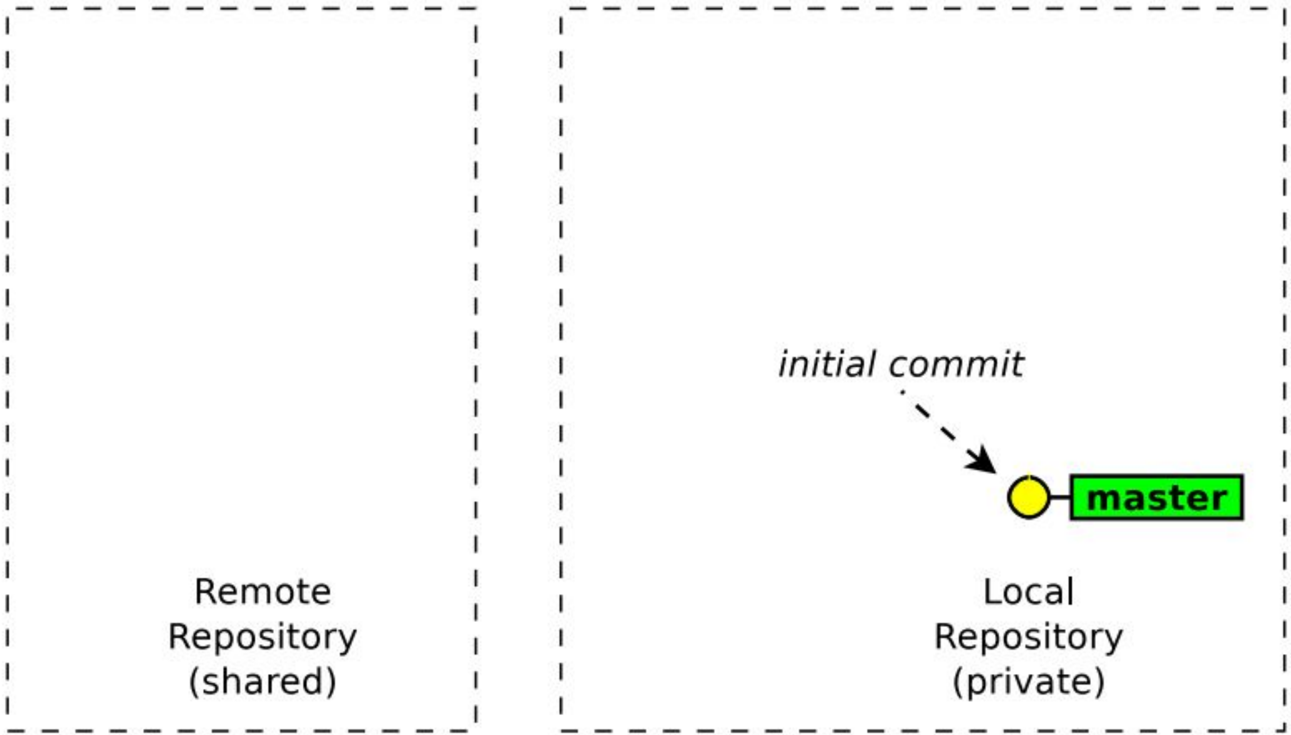
Remote
Repository
(shared)

```
git init
```

Local
Repository
(private)

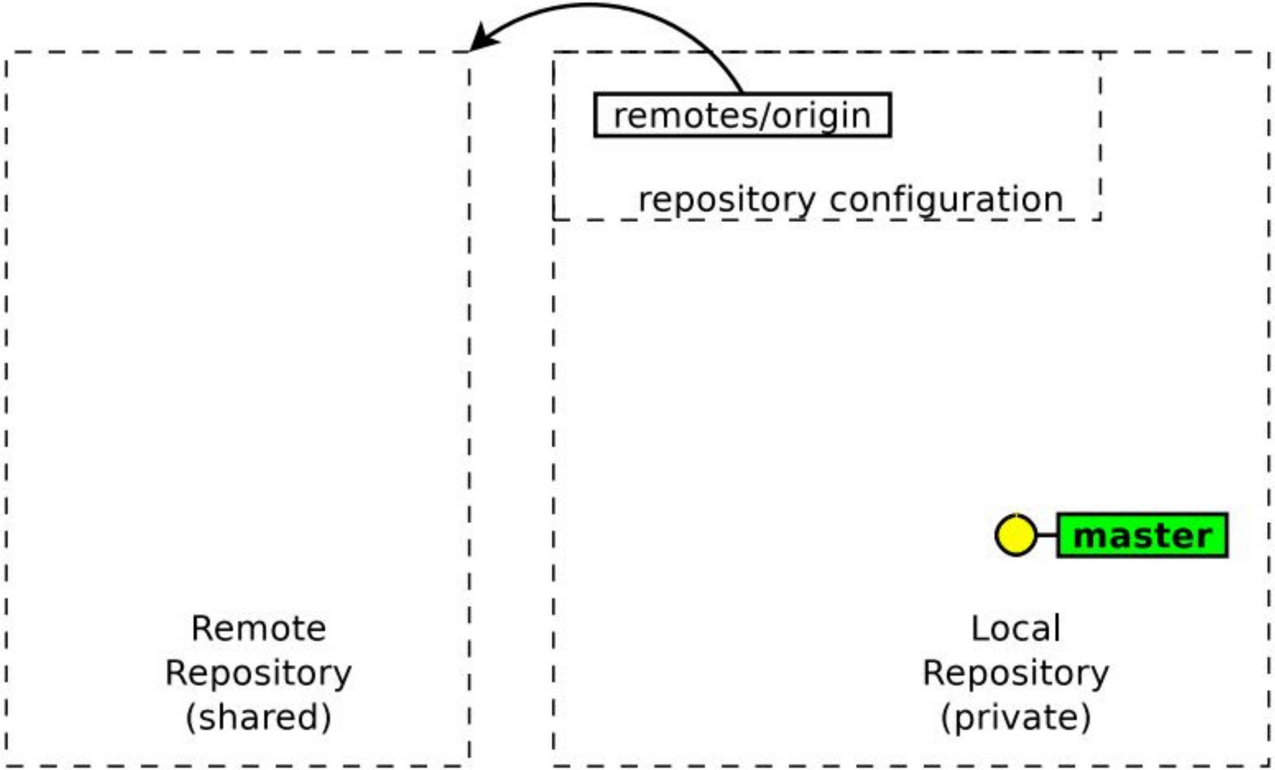
Remote example

```
git commit
```



Remote example

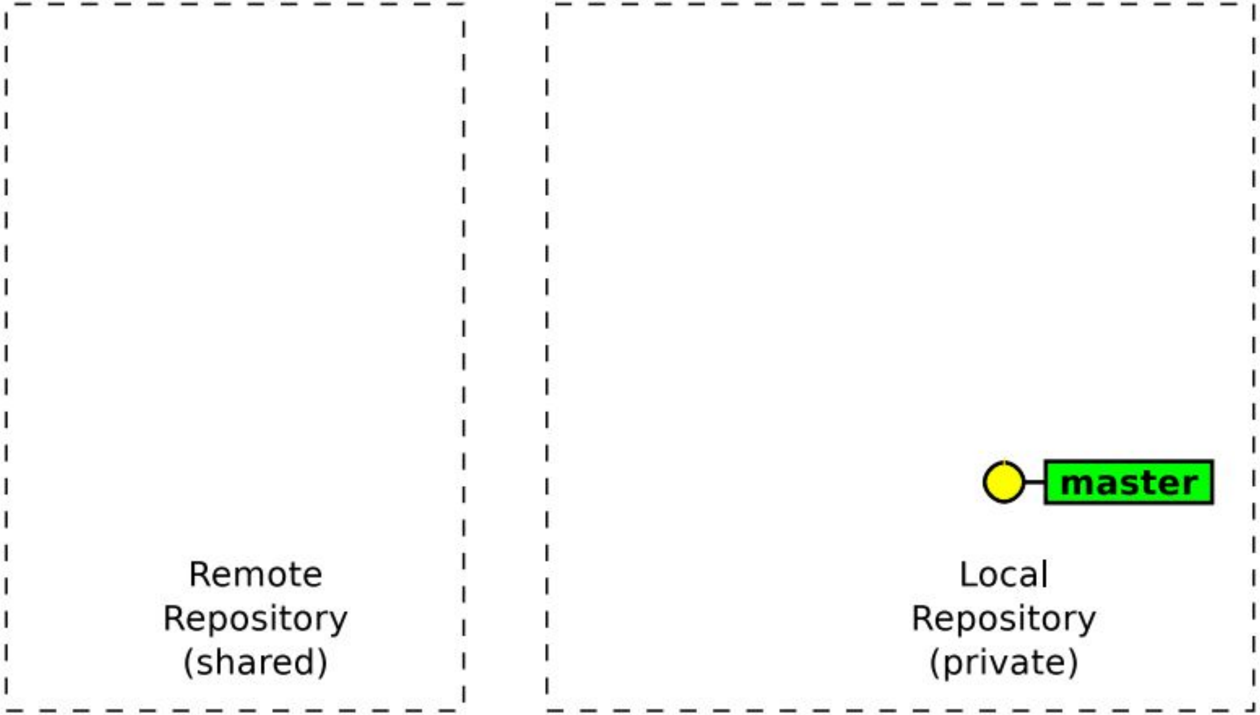
```
git remote add origin shared_url
```



Remote example

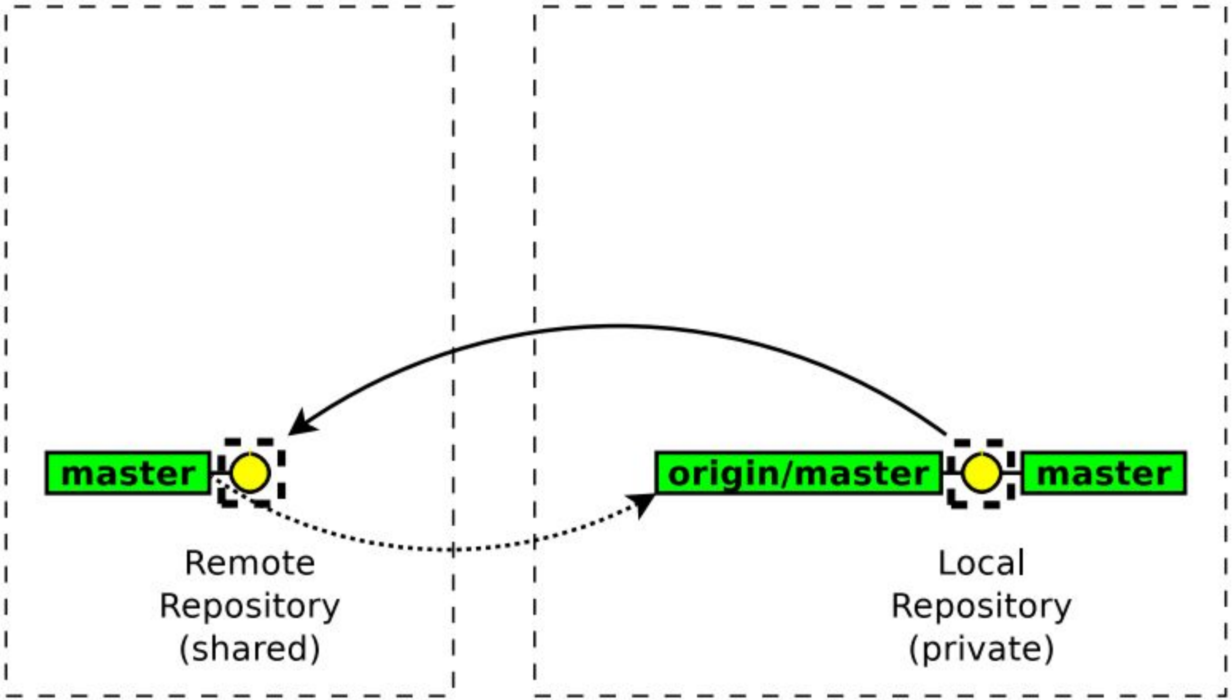
```
git push
```

-> nothing to be pushed !!



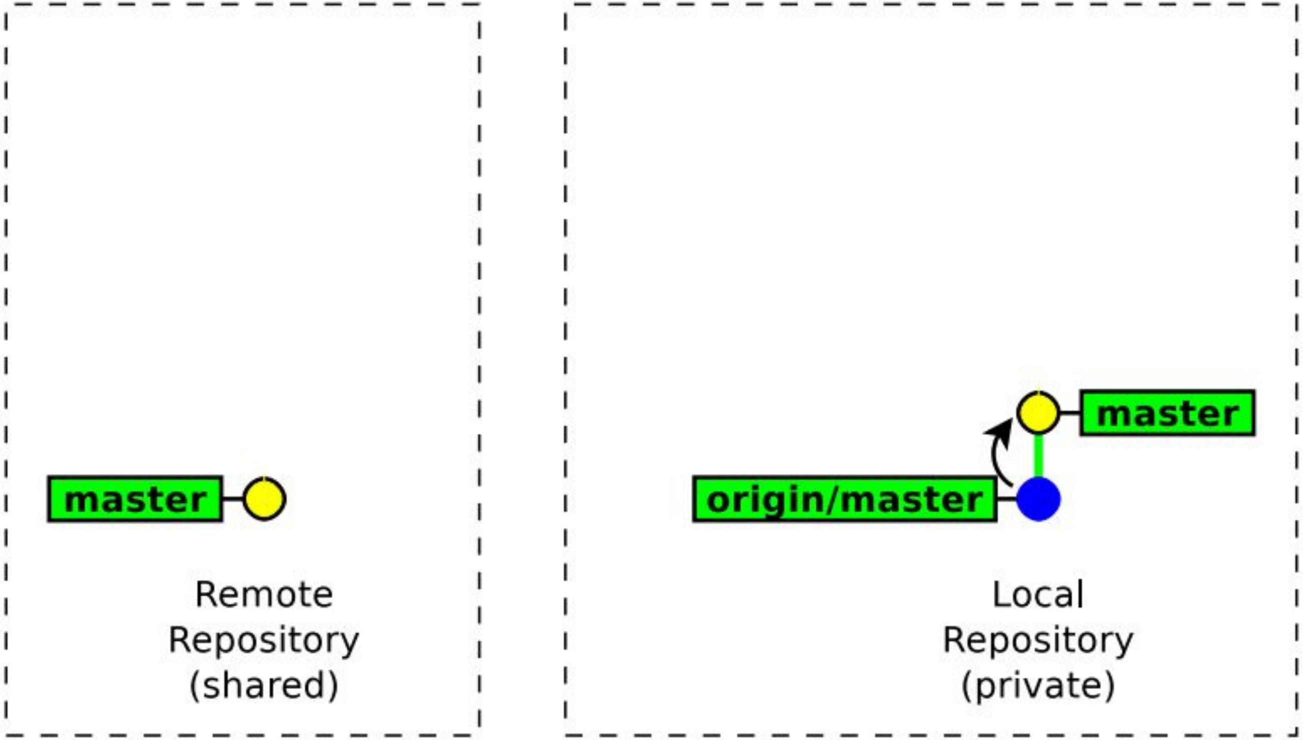
Remote example

```
git push -u origin master
```



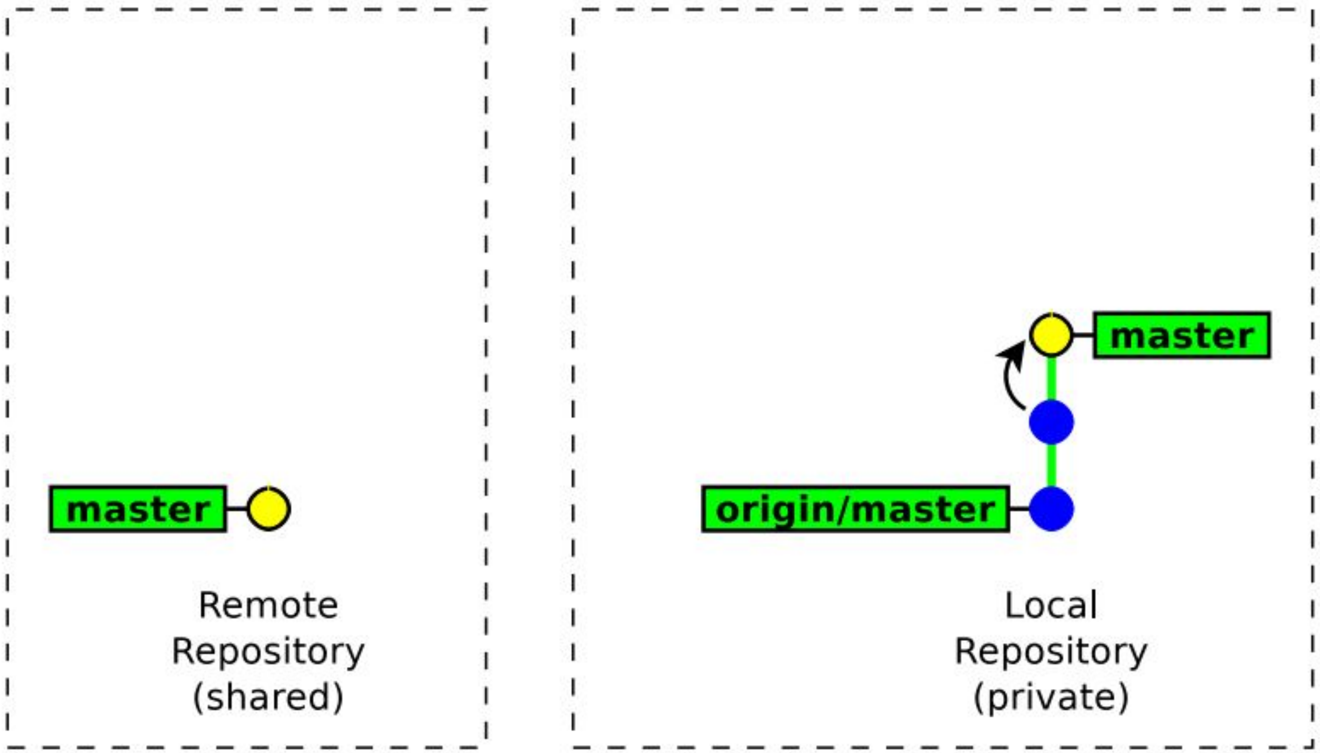
Remote example

```
git commit
```



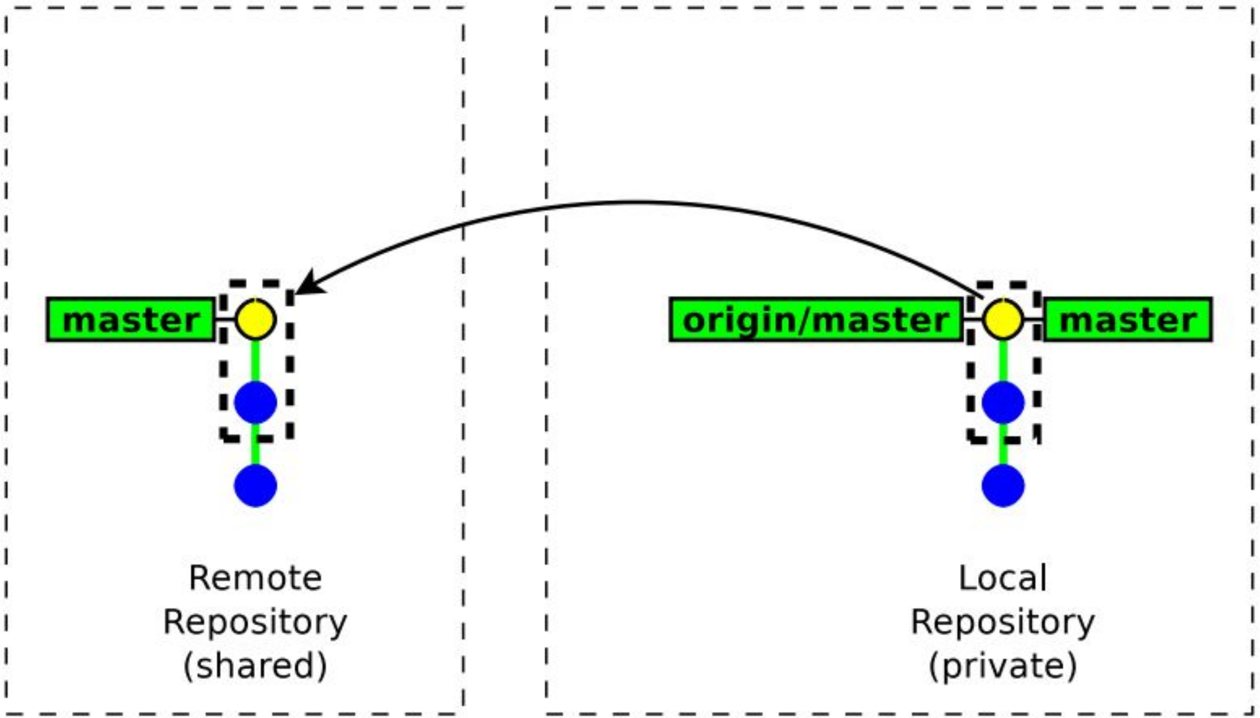
Remote example

```
git commit
```



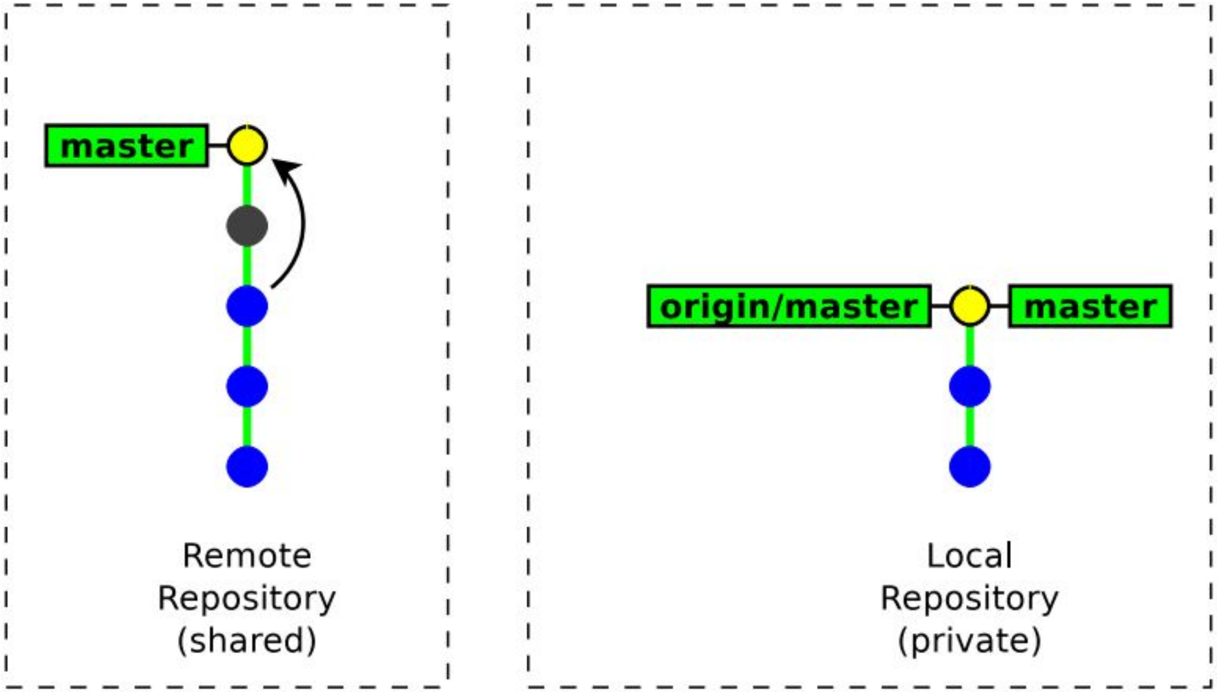
Remote example

```
git push
```



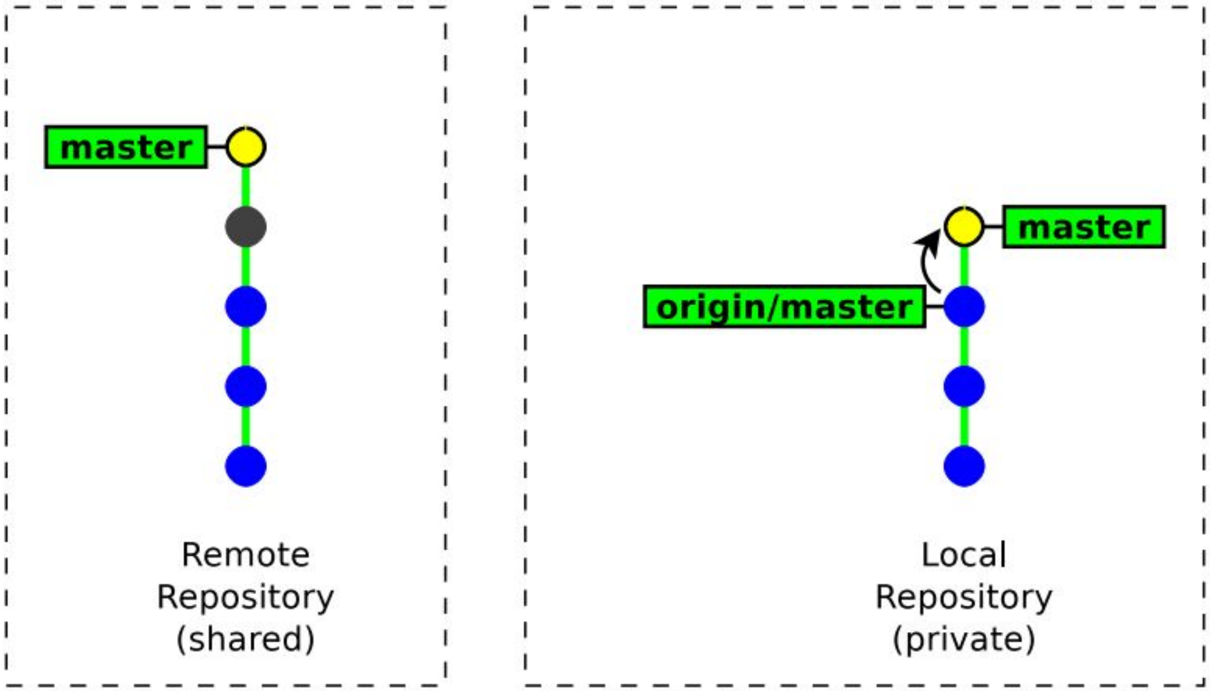
Remote example

another developer pushes his two commits



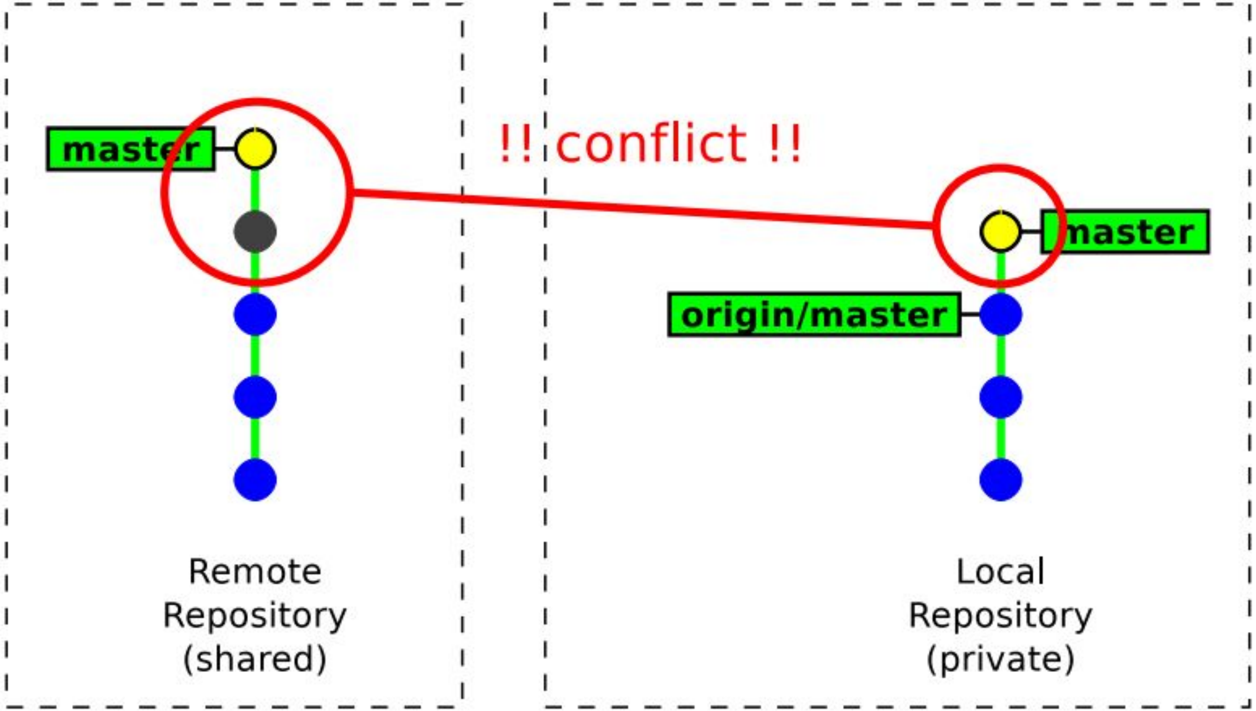
Remote example

```
git commit
```



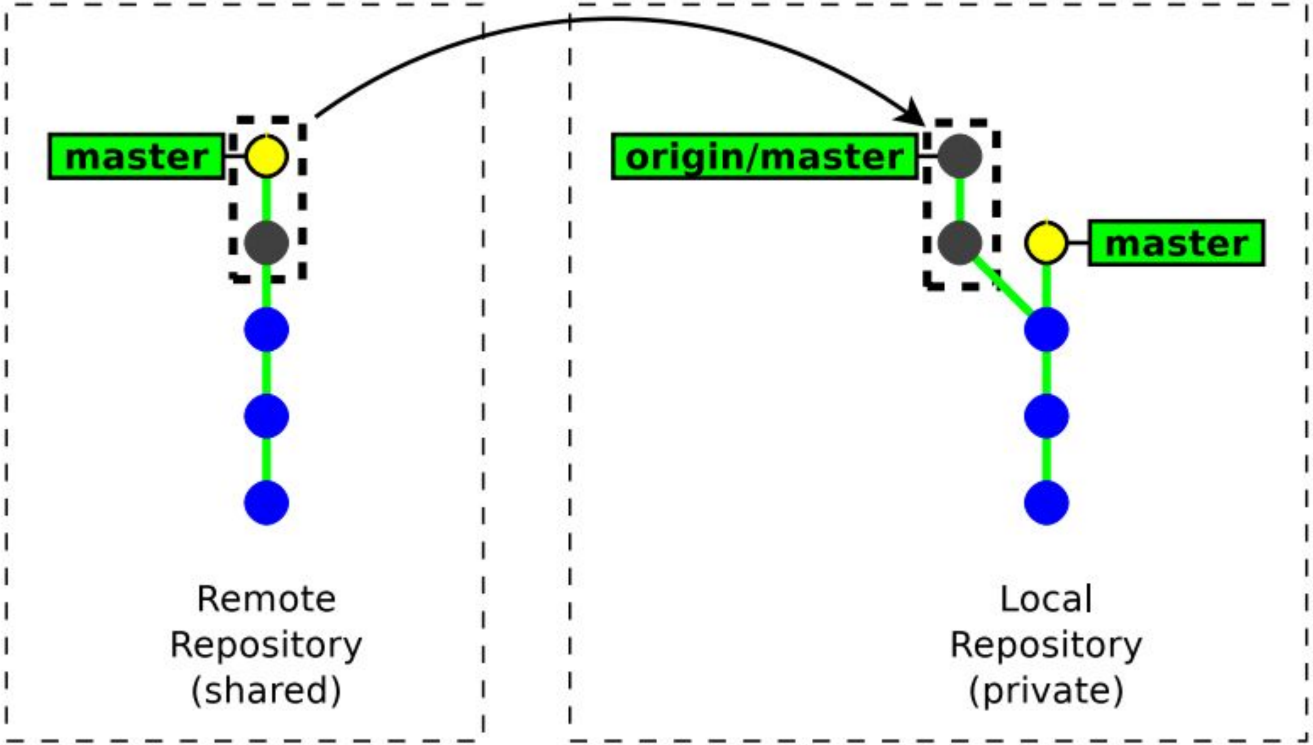
Remote example

```
git push
```



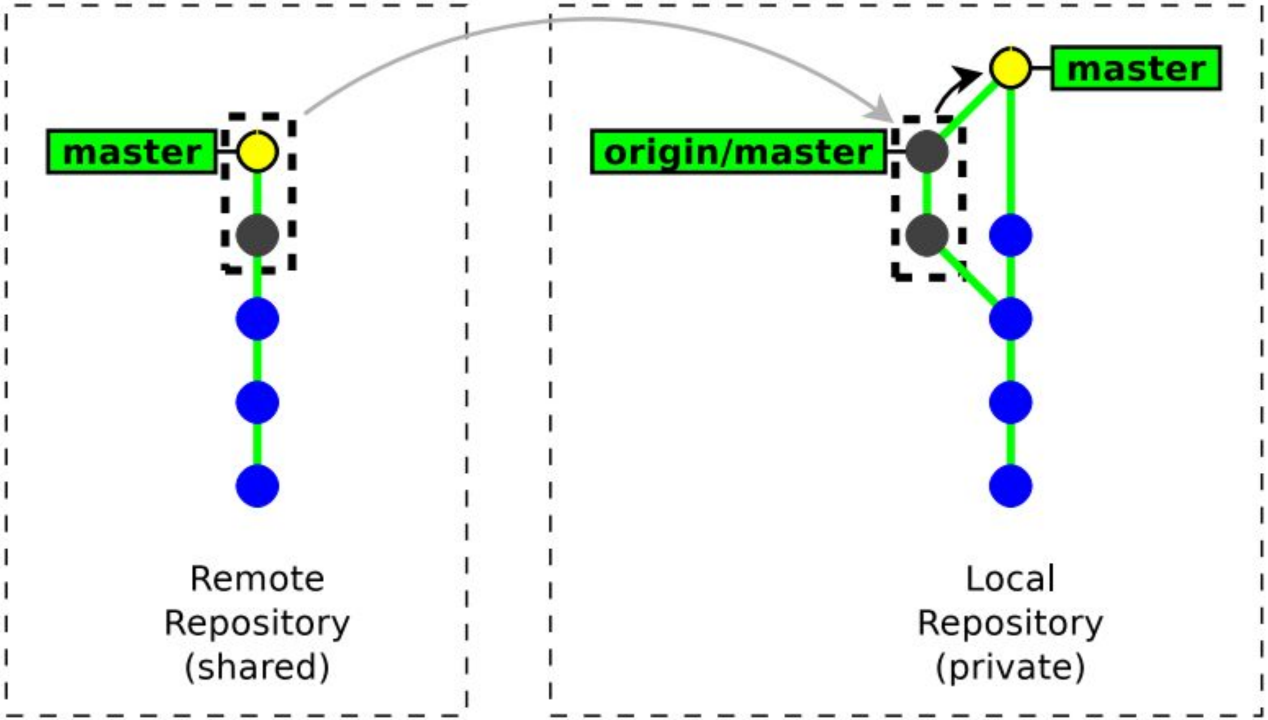
Remote example

```
git fetch
```

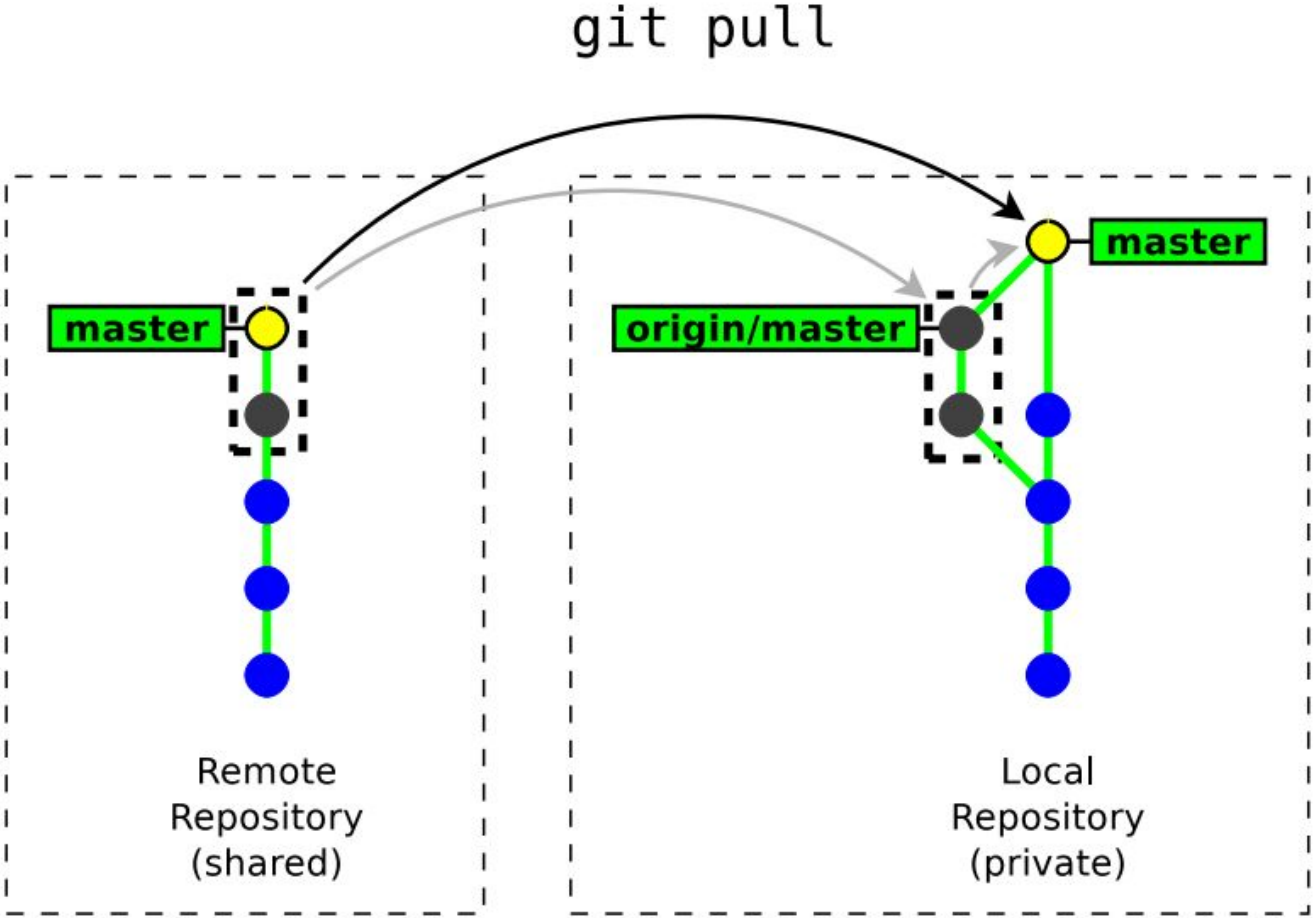


Remote example

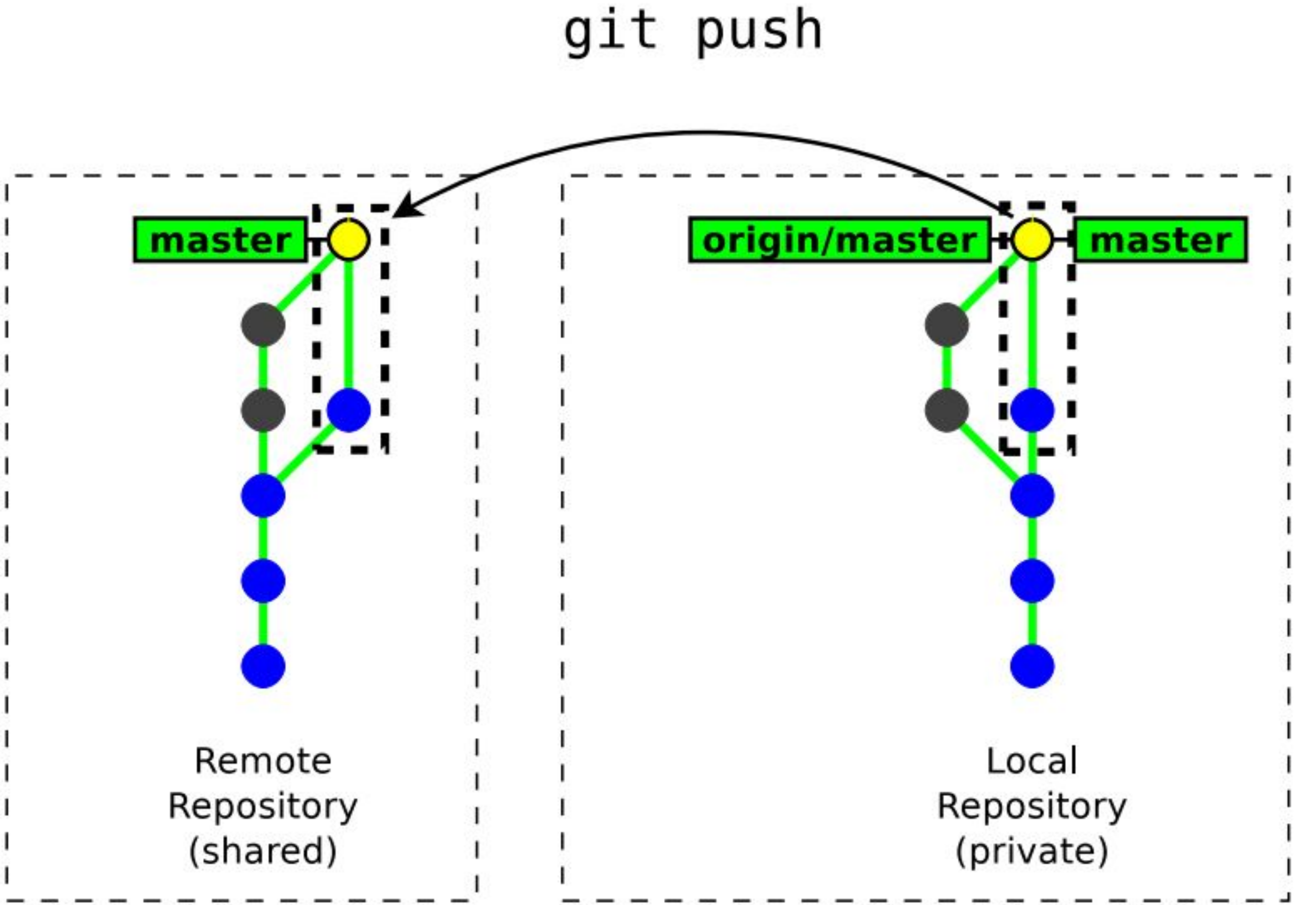
```
git merge origin/master
```



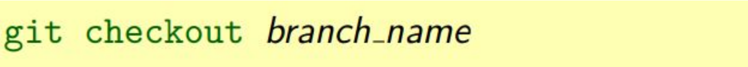
Remote example



Remote example

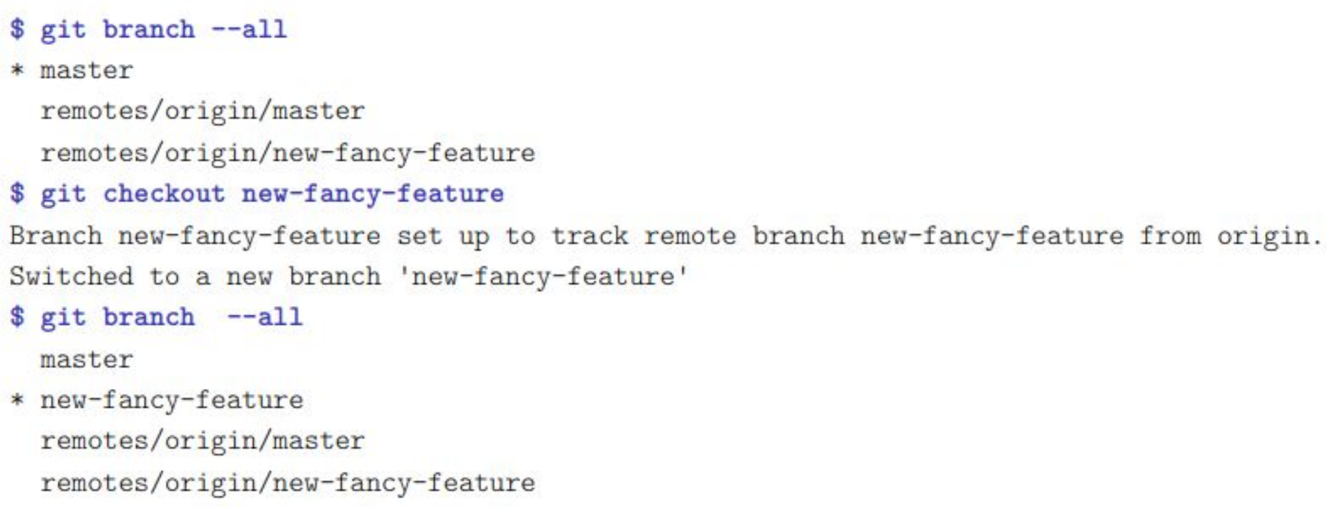


Importing a new remote branch



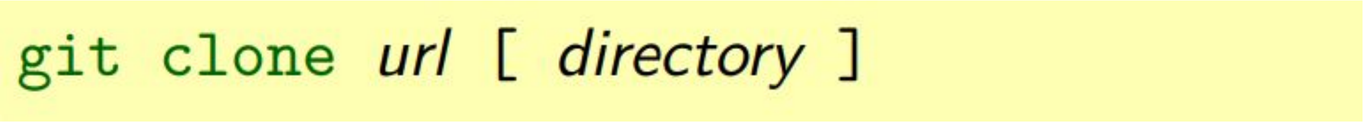
```
git checkout branch_name
```

If the branch_name does not exist locally, then GIT looks for it in the remote repositories. If it finds it, then it creates the local branch and configures it to track the remote branch.



```
$ git branch --all
* master
  remotes/origin/master
  remotes/origin/new-fancy-feature
$ git checkout new-fancy-feature
Branch new-fancy-feature set up to track remote branch new-fancy-feature from origin.
Switched to a new branch 'new-fancy-feature'
$ git branch  --all
  master
* new-fancy-feature
  remotes/origin/master
  remotes/origin/new-fancy-feature
```

Cloning a repository



```
git clone url [ directory ]
```

- **git clone** makes a local copy of a remote repository and configures it as its origin remote repository.

- **git clone** is a shortcut for the following sequence:

  1. git init directory

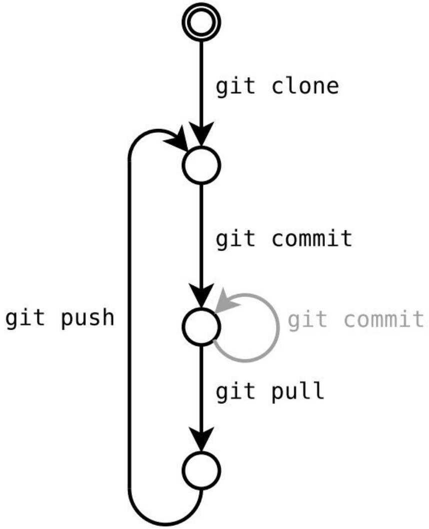
  2. cd directory

  3. git remote add origin url

  4. git fetch

  5. git checkout master

- In practice you will rarely use git init, git remote and git fetch directly, but rather use higher-level commands: git clone and git pull.



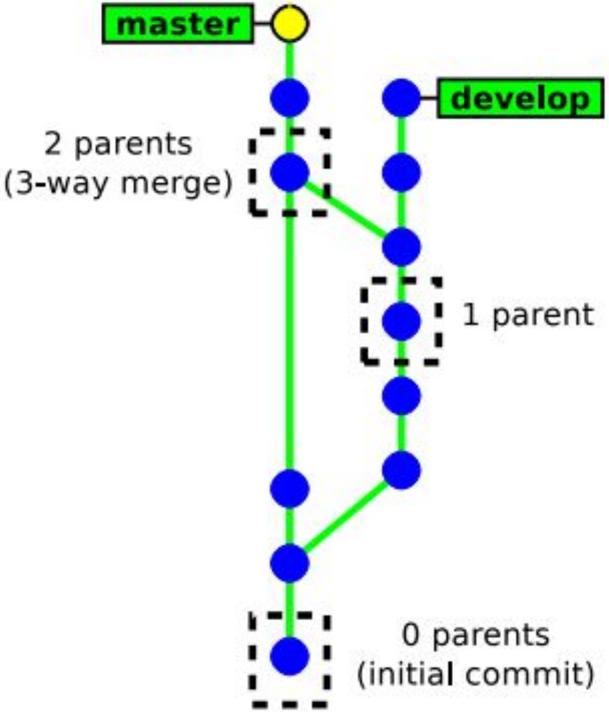
Branching & merging

GIT history

Each commit object has a list of parent commits:

- 0 parents → initial commit
- 1 parent → ordinary commit
- 2+ parents → result of a merge

→ This is a Direct Acyclic Graph



GIT history

- There is no formal “branch history”

→ a branch is just a pointer on the latest commit.

(git handles branches and tags in the same way internally)

- Commits are identified with SHA-1 hash (160 bits) computed from:

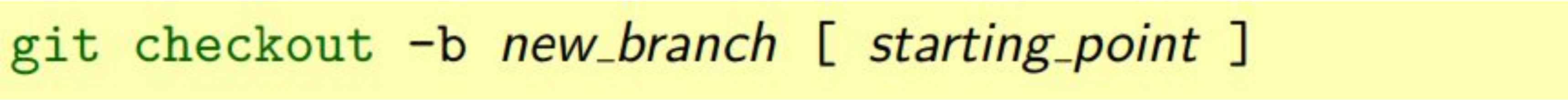
- the commited files

- the meta data (commit message, author name, . . .)

- the hashes of the parent commits

→ A commit id (hash) identifies securely and reliably its content and all the previous revisions.

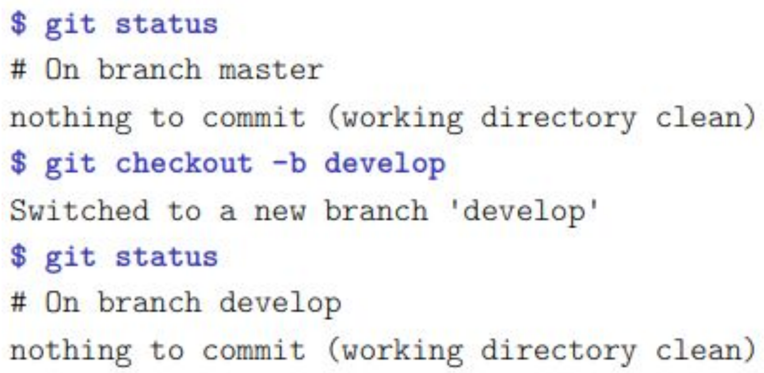
Creating a new branch



```
git checkout -b new_branch [ starting_point ]
```

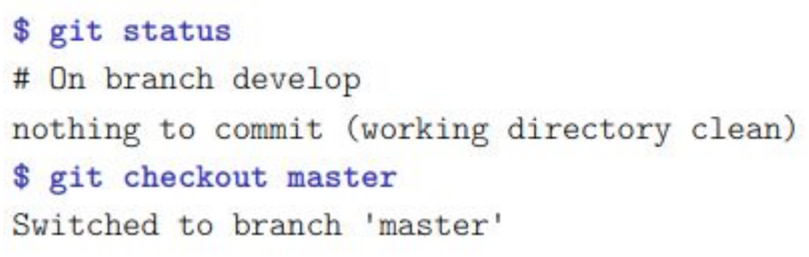
- new branch is the name of the new branch

- starting point is the starting location of the branch (possibly a commit id, a tag, a branch, . . .). If not present, git will use the current location.



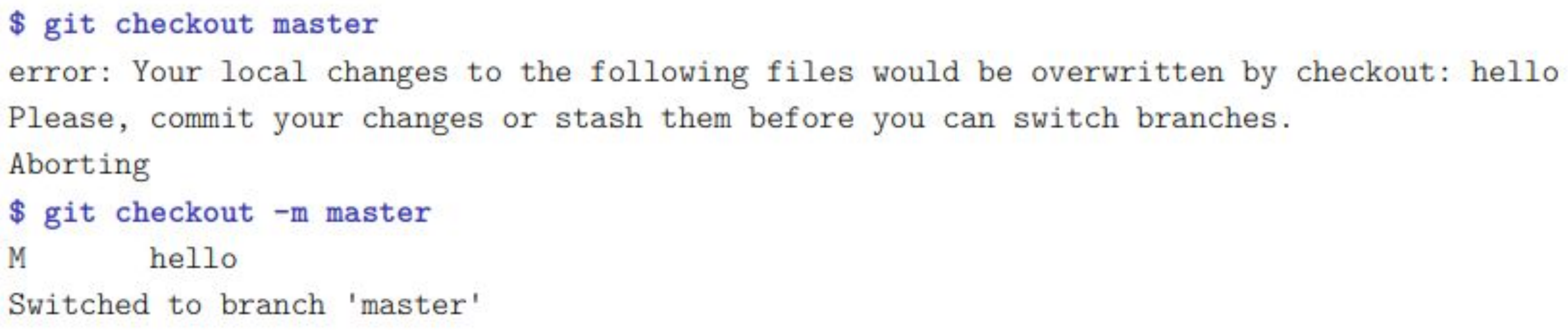
```
$ git status
# On branch master
nothing to commit (working directory clean)
$ git checkout -b develop
Switched to a new branch 'develop'
$ git status
# On branch develop
nothing to commit (working directory clean)
```

Switching between branches



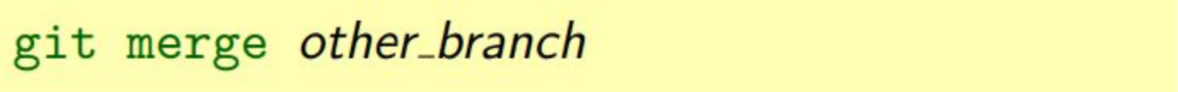
```
$ git status
# On branch develop
nothing to commit (working directory clean)
$ git checkout master
Switched to branch 'master'
```

Note: it may fail when the working copy is not clean. Add -m to request merging your local changes into the destination branch.



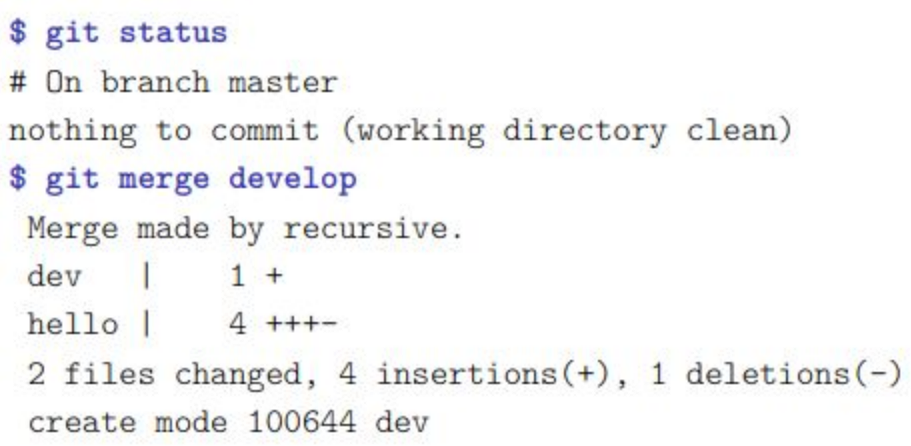
```
$ git checkout master
error: Your local changes to the following files would be overwritten by checkout: hello
Please, commit your changes or stash them before you can switch branches.
Aborting
$ git checkout -m master
M       hello
Switched to branch 'master'
```

Merging a branch



```
git merge other_branch
```

This will merge the changes in other branch into the current branch.



```
$ git status
# On branch master
nothing to commit (working directory clean)
$ git merge develop
 Merge made by recursive.
 dev   |    1 +
 hello |    4 +++-
 2 files changed, 4 insertions(+), 1 deletions(-)
 create mode 100644 dev
```

Merging a branch

Notes about merging:

- The result of **git merge** is immediately committed (unless there is a conflict)

- The new commit object has **two parents**.

  → the merge history is recorded

- **git merge** applies only the changes since the last common ancestor in the other branch.

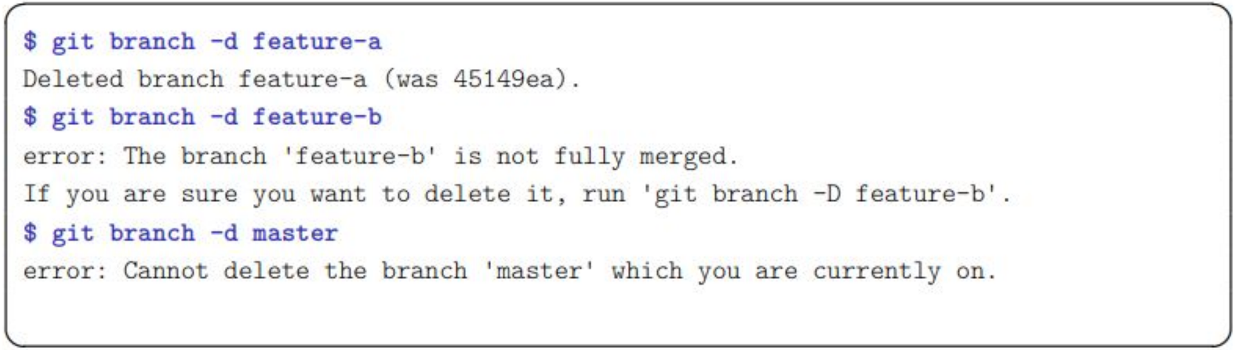
  → if the branch was already merged previously, then only the changes since the last **merge** will be merged.

Deleting branches

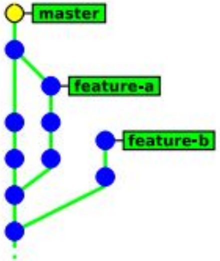
```
git branch -d branch_name
```

This command has some restrictions, it cannot delete:

- the current branch (HEAD)
- a branch that has not yet been merged into the current branch

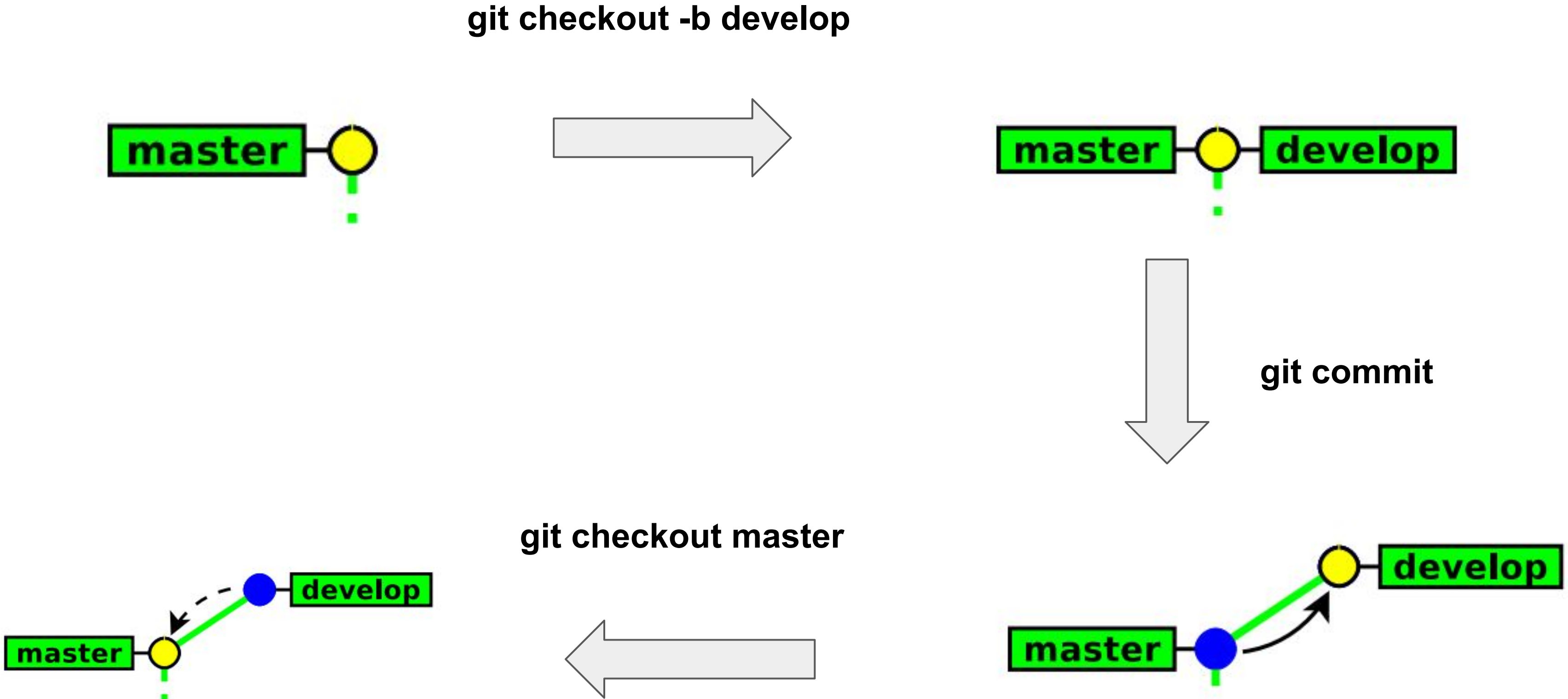


```
$ git branch -d feature-a
Deleted branch feature-a (was 45149ea).
$ git branch -d feature-b
error: The branch 'feature-b' is not fully merged.
If you are sure you want to delete it, run 'git branch -D feature-b'.
$ git branch -d master
error: Cannot delete the branch 'master' which you are currently on.
```



→ **git branch -d** is safe, unlike **git branch -D** which deletes unconditionally the branch

Branching example

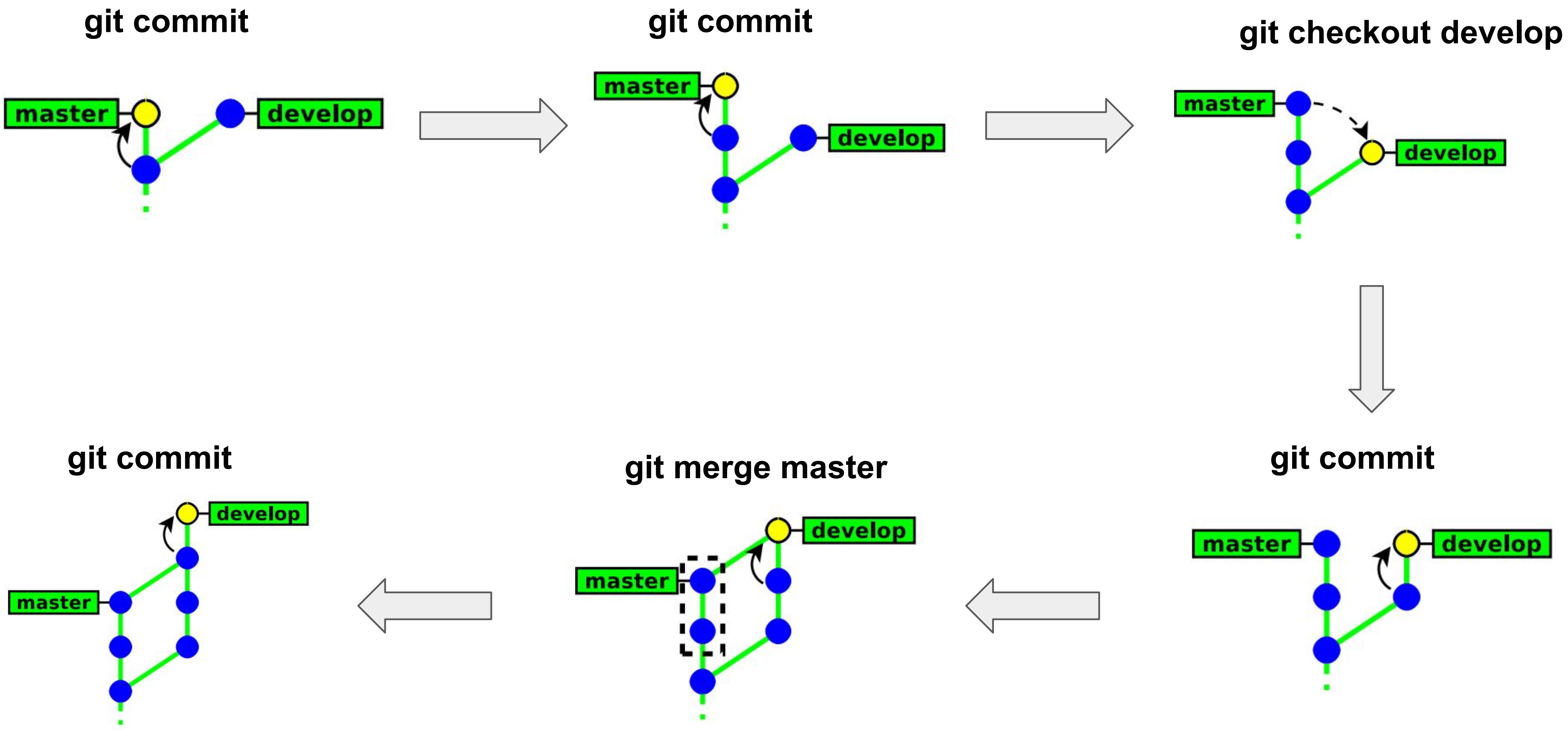


git checkout -b develop

git commit

git checkout master

Branching example



git commit

git commit

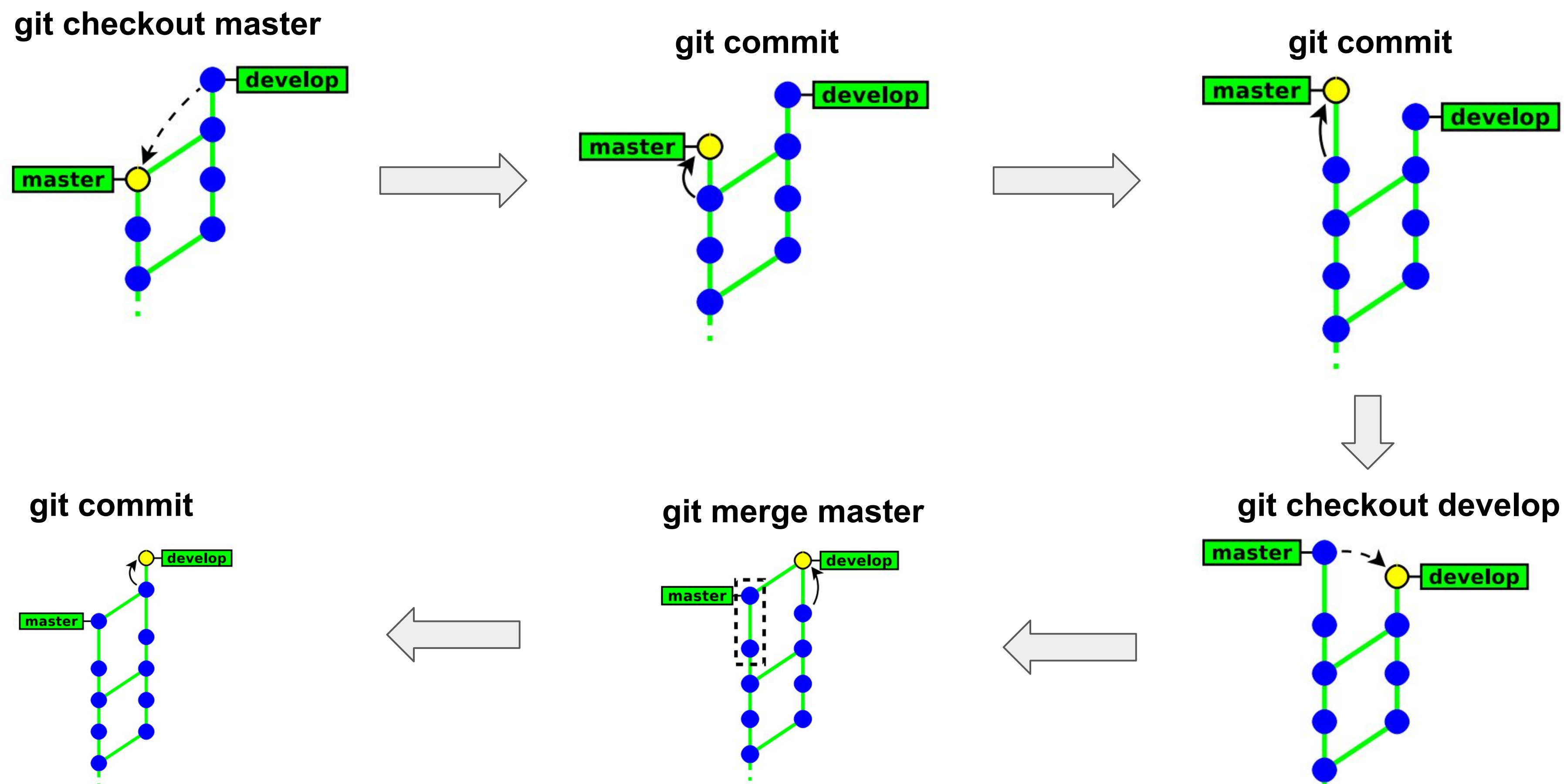
git checkout develop

git commit

git merge master

git commit

Branching example



git checkout master

git commit

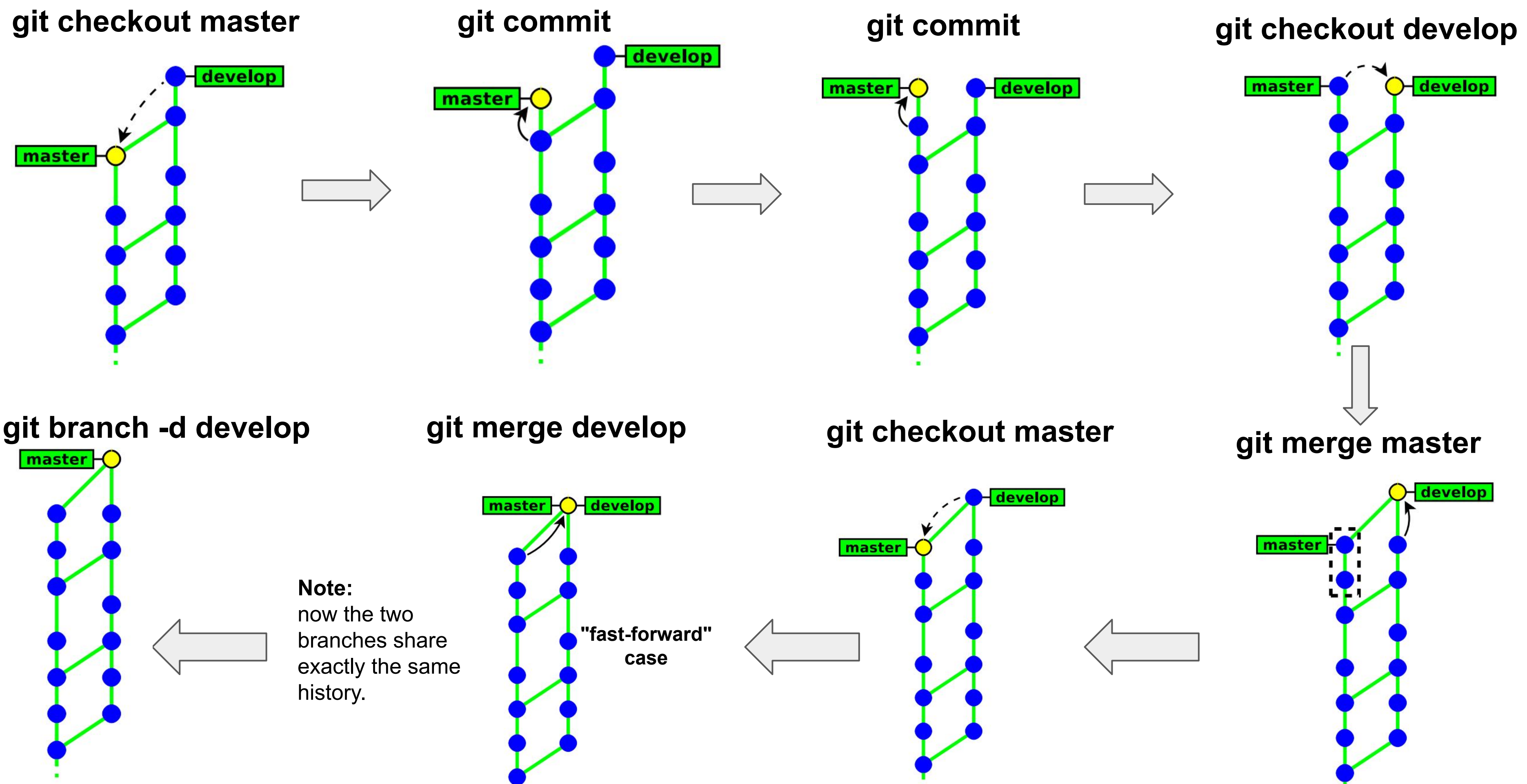
git commit

git checkout develop

git merge master

git commit

Branching example



git checkout master

git commit

git commit

git checkout develop

git merge master

git checkout master

git merge develop

"fast-forward" case

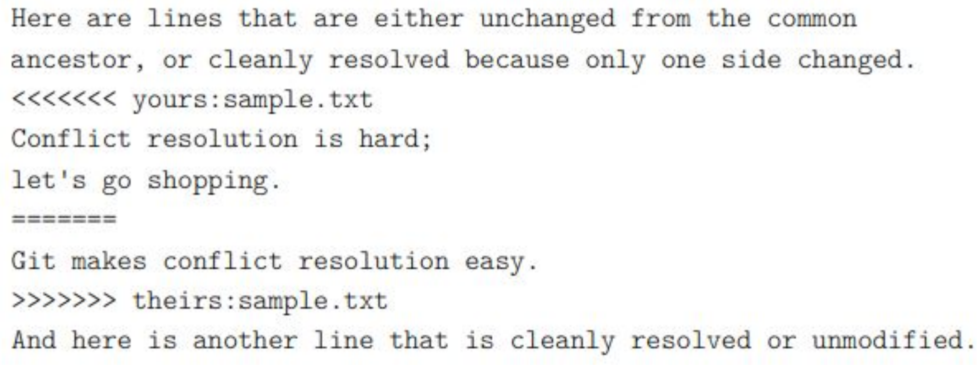
git branch -d develop

Note:
now the two branches share exactly the same history.

How Git merges files ?

If the same file was independently modified in the two branches, then Git needs to merge these two variants

- textual files are merged on a per-line basis:
  - lines changed in only one branch are automatically merged
  - if a line was modified in the two branches, then Git reports a conflict. Conflict zones are enclosed within <<<<<<< >>>>>>>



```
Here are lines that are either unchanged from the common
ancestor, or cleanly resolved because only one side changed.
<<<<<<< yours:sample.txt
Conflict resolution is hard;
let's go shopping.
=======
Git makes conflict resolution easy.
>>>>>>> theirs:sample.txt
And here is another line that is cleanly resolved or unmodified.
```

- binary files always raise a conflict and require manual merging

Merge conflicts

In case of a conflict:

- **Unmerged files** (those having conflicts) are left **in the working tree** and marked as “unmerged” (Git will refuse to commit the new revision until all the conflicts are explicitly resolved by the user).
- **The other files** (free of conflicts) and the metadata (commit message, parents commits, ...) are automatically added **into the index** (the staging area)

Resolving conflicts

There are two ways to resolve conflicts:

- either edit the files manually, then run

  git add file → to check the file into the index

  or

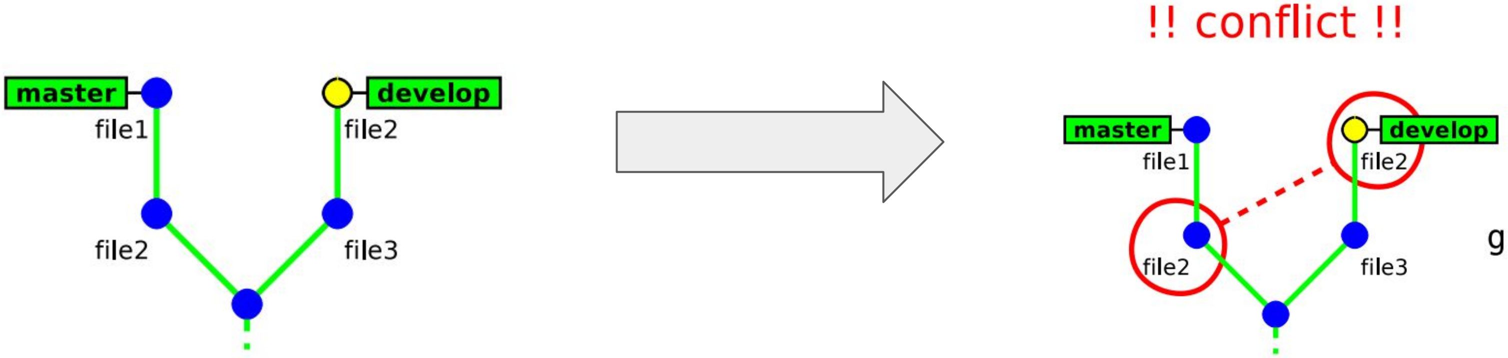
  git rm file → to delete the file

- or with a conflict resolution tool(xxdiff, kdiff3, emerge, ...)

  git mergetool [file]

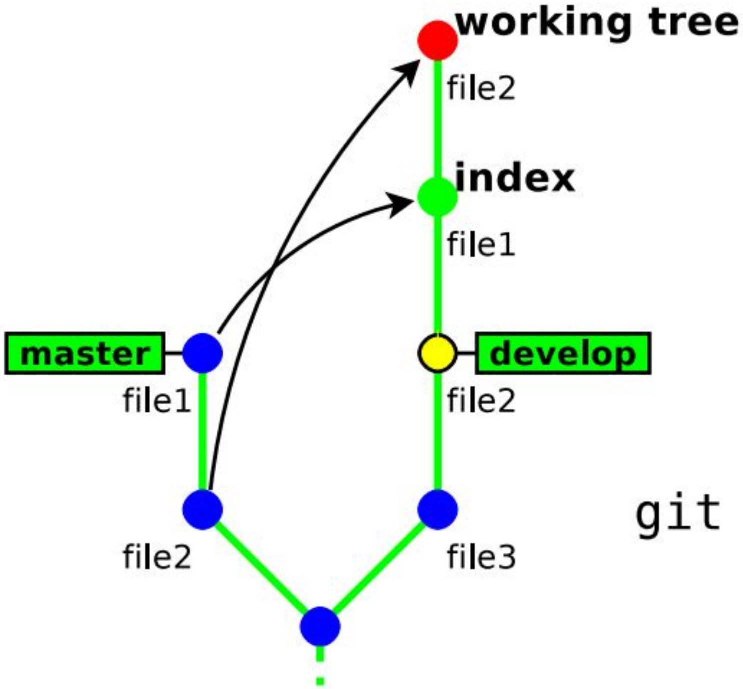
Then, once all conflicting files are checked in the index, you just need to run **git commit** to commit the merge.

Conflict example



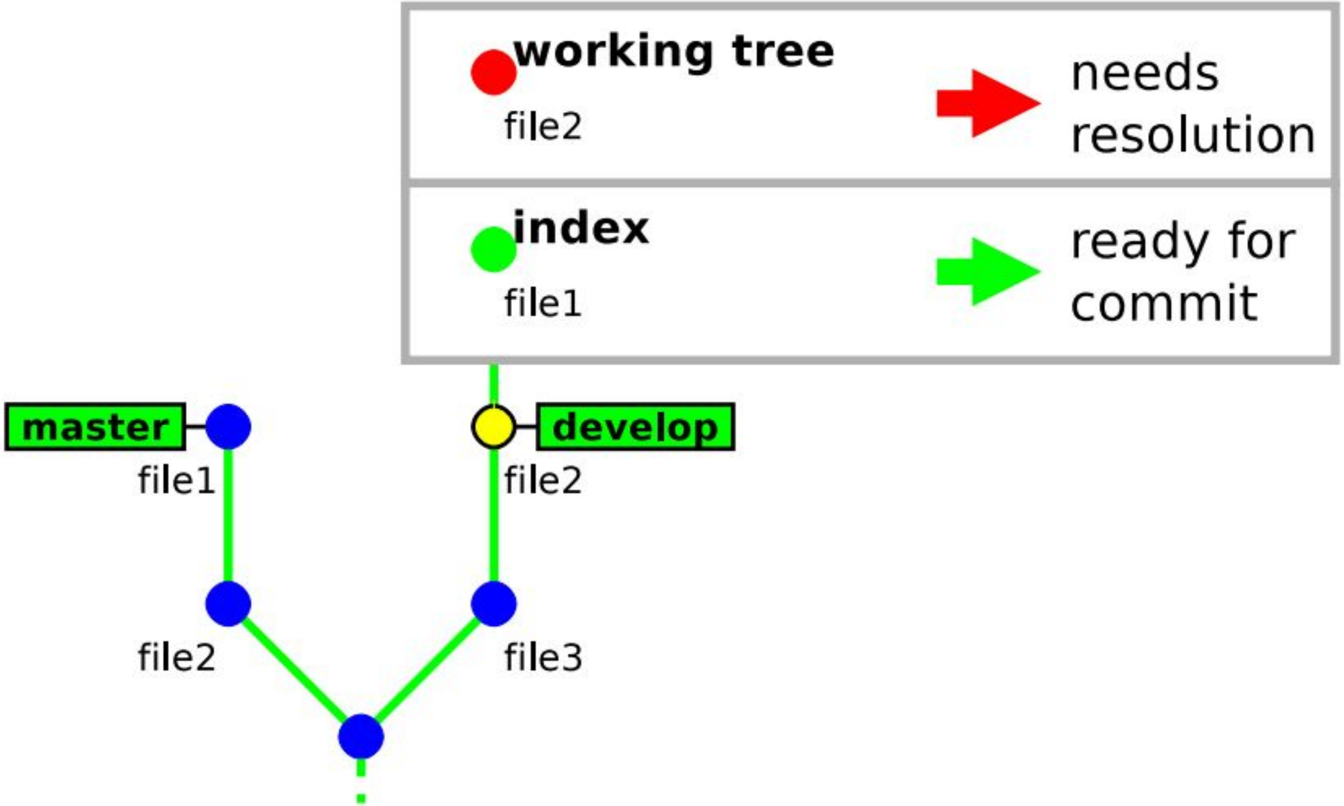
```
git merge master
```

Conflict example

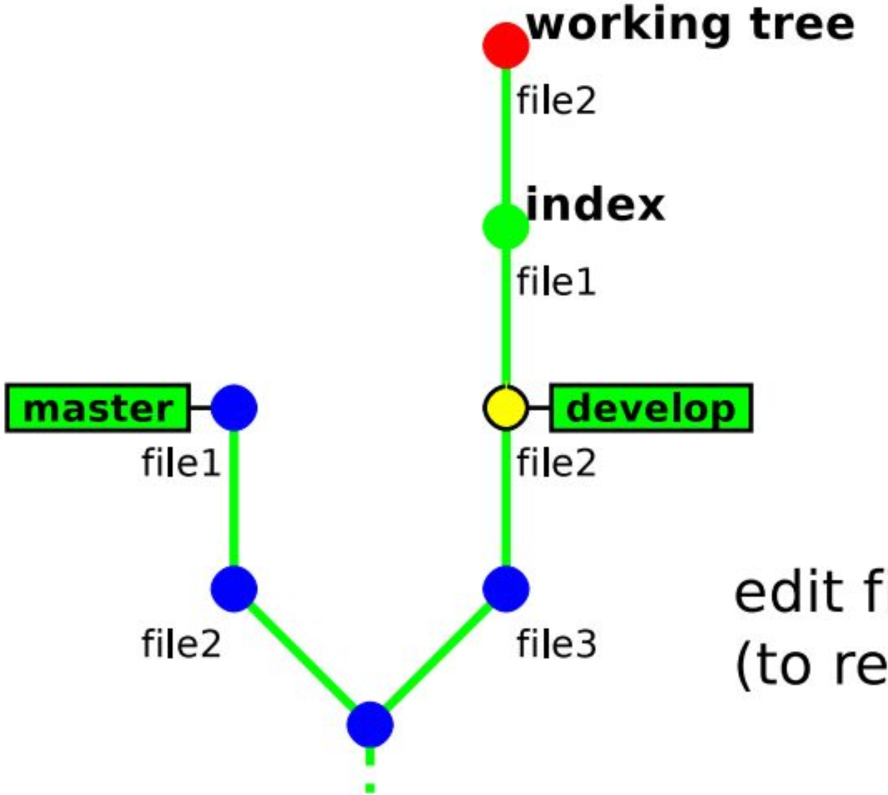


```
git merge master
```

Conflict example

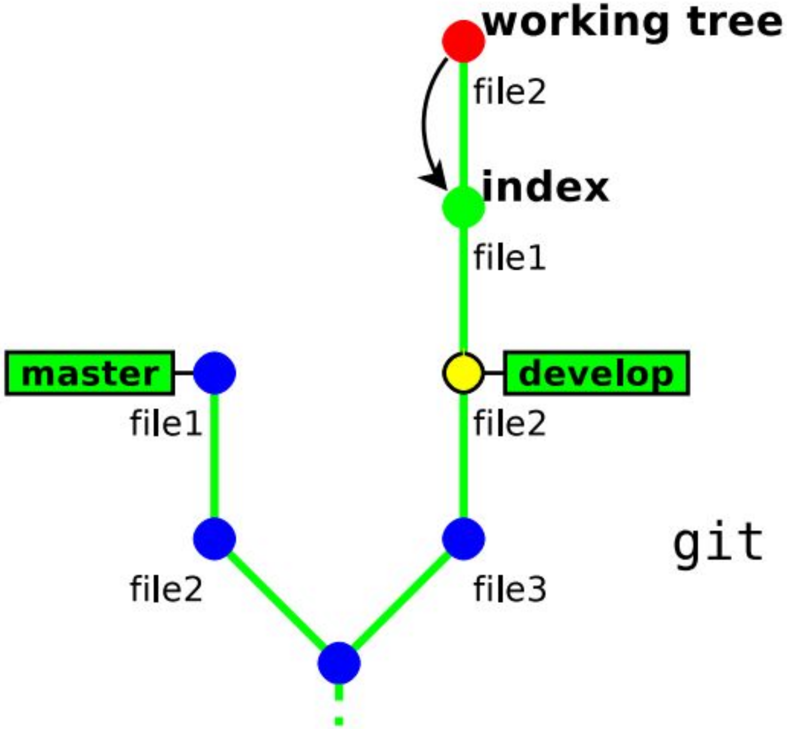


Conflict example



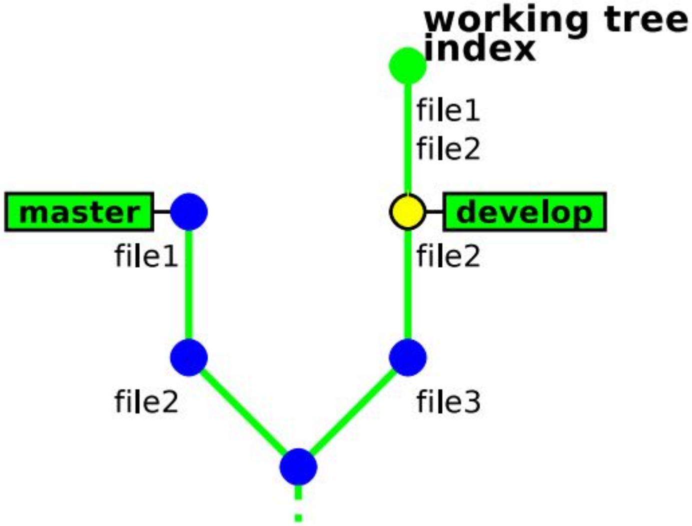
edit file2
(to resolve conflicts)

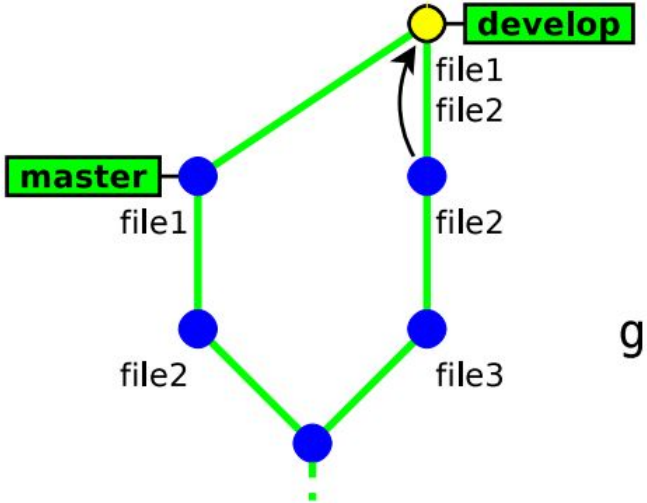
Conflict example



```
git add file2
```

Conflict example



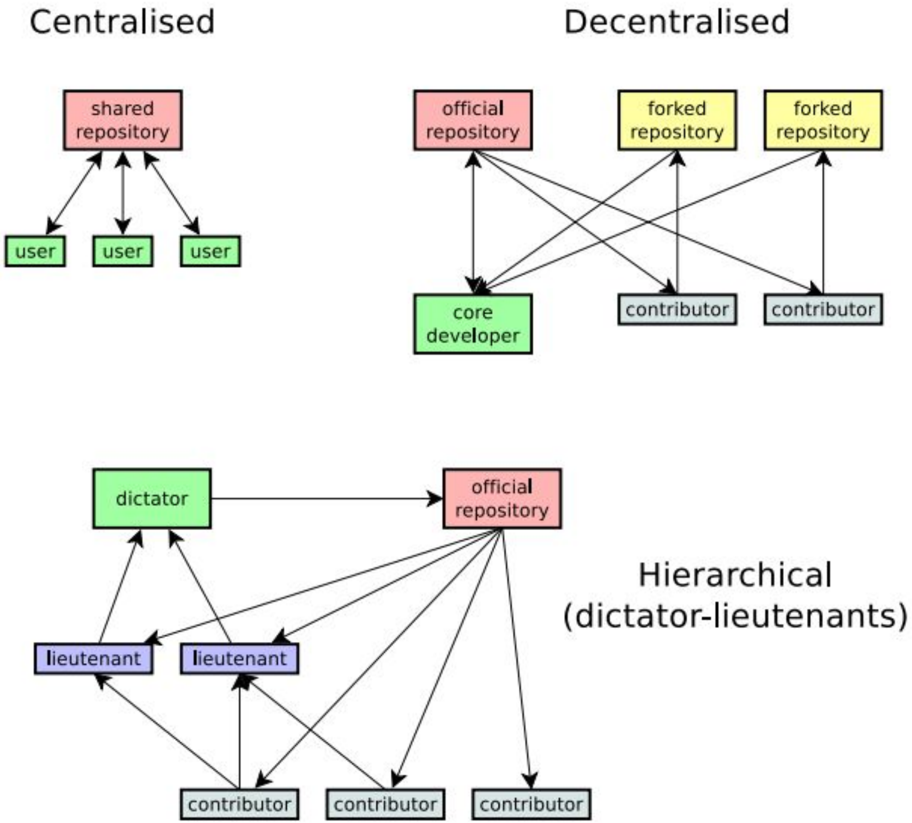


```
git commit
```

Common workflows

Common workflows

Common workflows



Centralised

Decentralised

Hierarchical (dictator-lieutenants)

Third party contributions

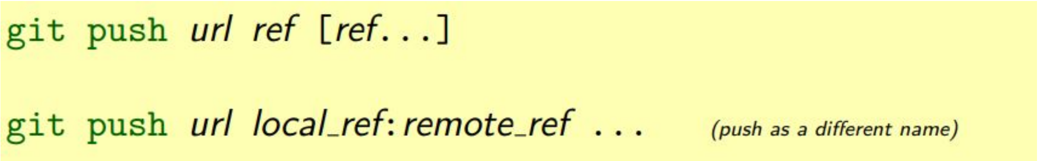
Third-party contributors (developers who are not allowed to push to the official repository) can submit their contributions by:

- Sending patches (the traditional way)
- Publishing their own (unofficial) repository and asking an official developer to merge from this repository (pull request or merge request)

Explicit pull/push

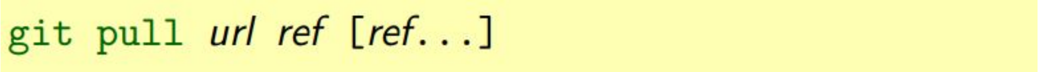
- **push/pull can work on any arbitrary repository identified by its url**

→ push the local ref (a branch or a tag) to repository url



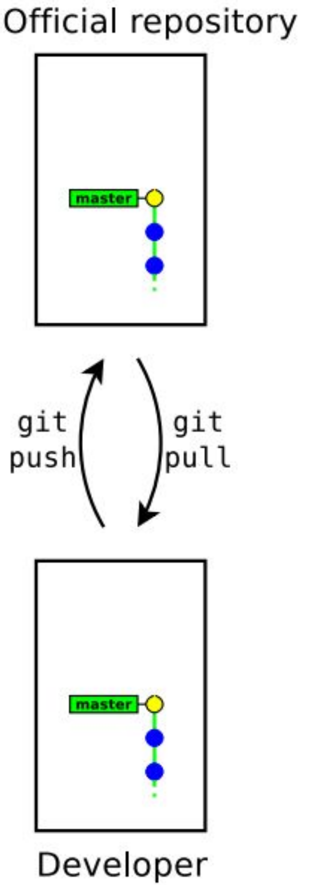
```
git push url ref [ref...]

git push url local_ref:remote_ref ...      (push as a different name)
```

→ merge the remote ref (a branch or a tag) from repository url into the current local branch



```
git pull url ref [ref...]
```

Decentralised workflow



Official repository

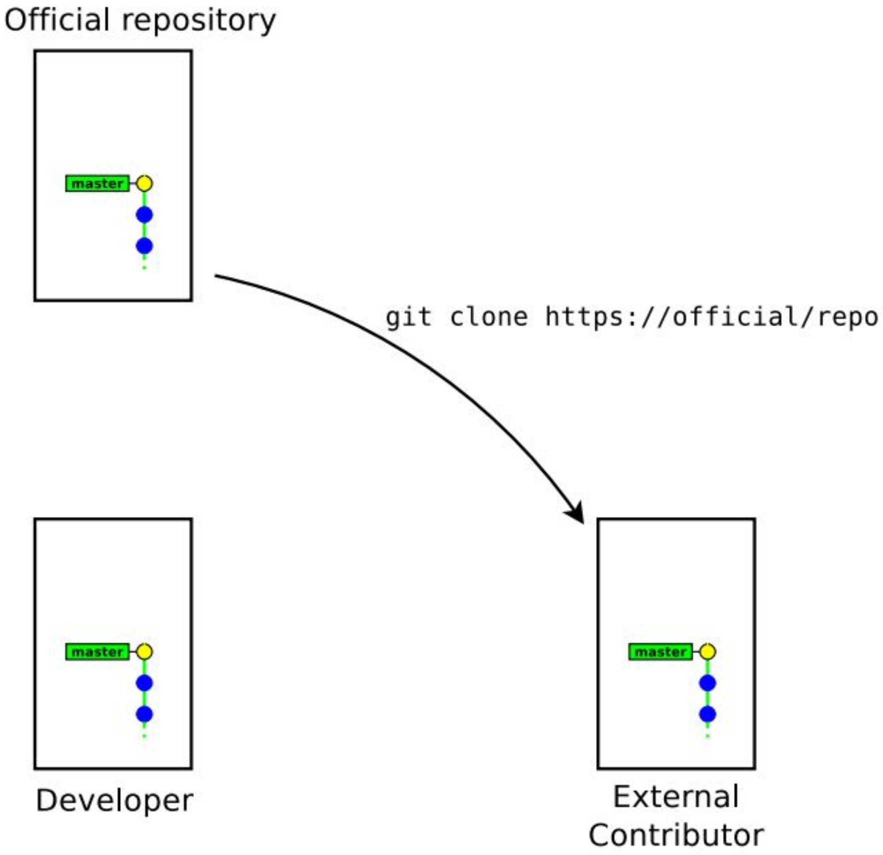
master

git push

git pull

master

Developer



Official repository

master

git clone https://official/repo

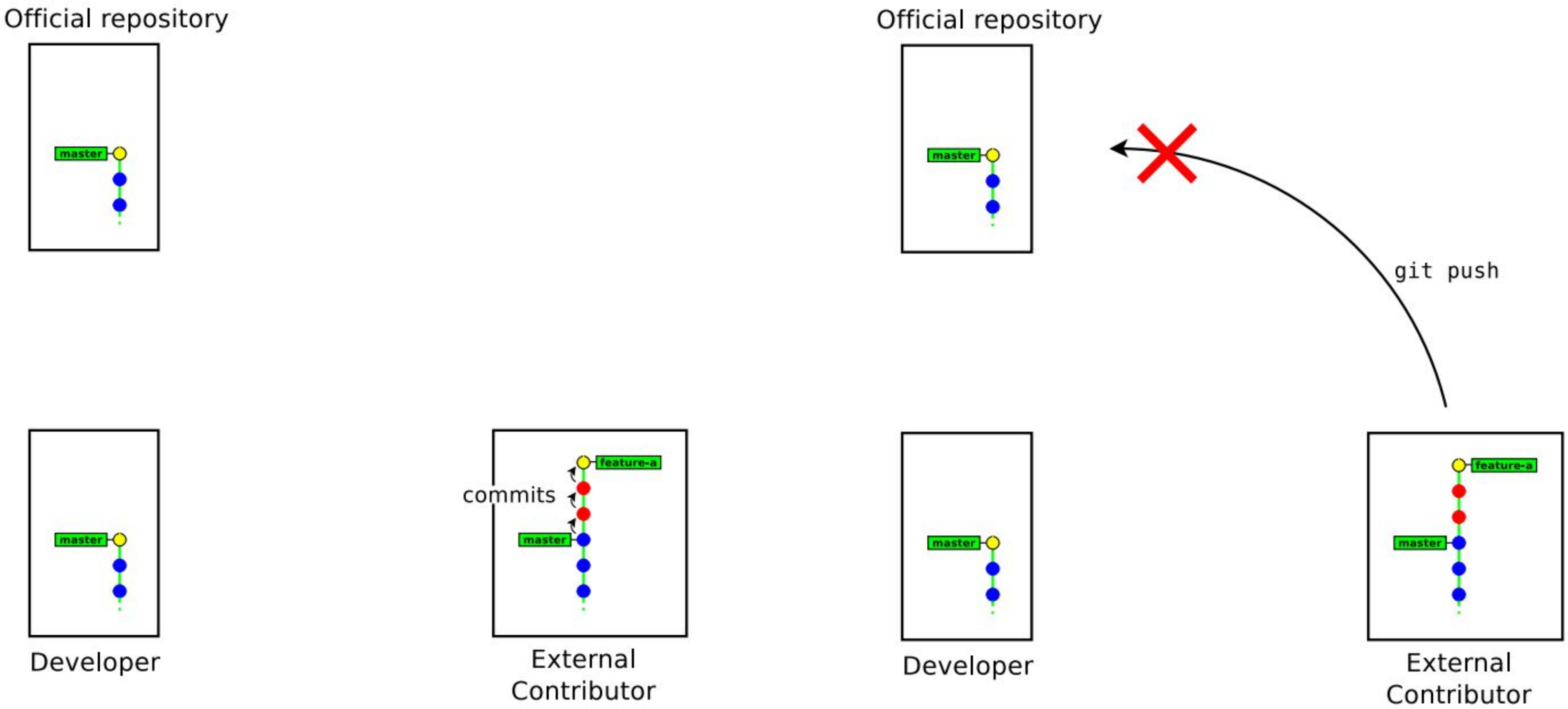
master

Developer

master

External Contributor

Decentralised workflow



Official repository

Developer

External Contributor

commits

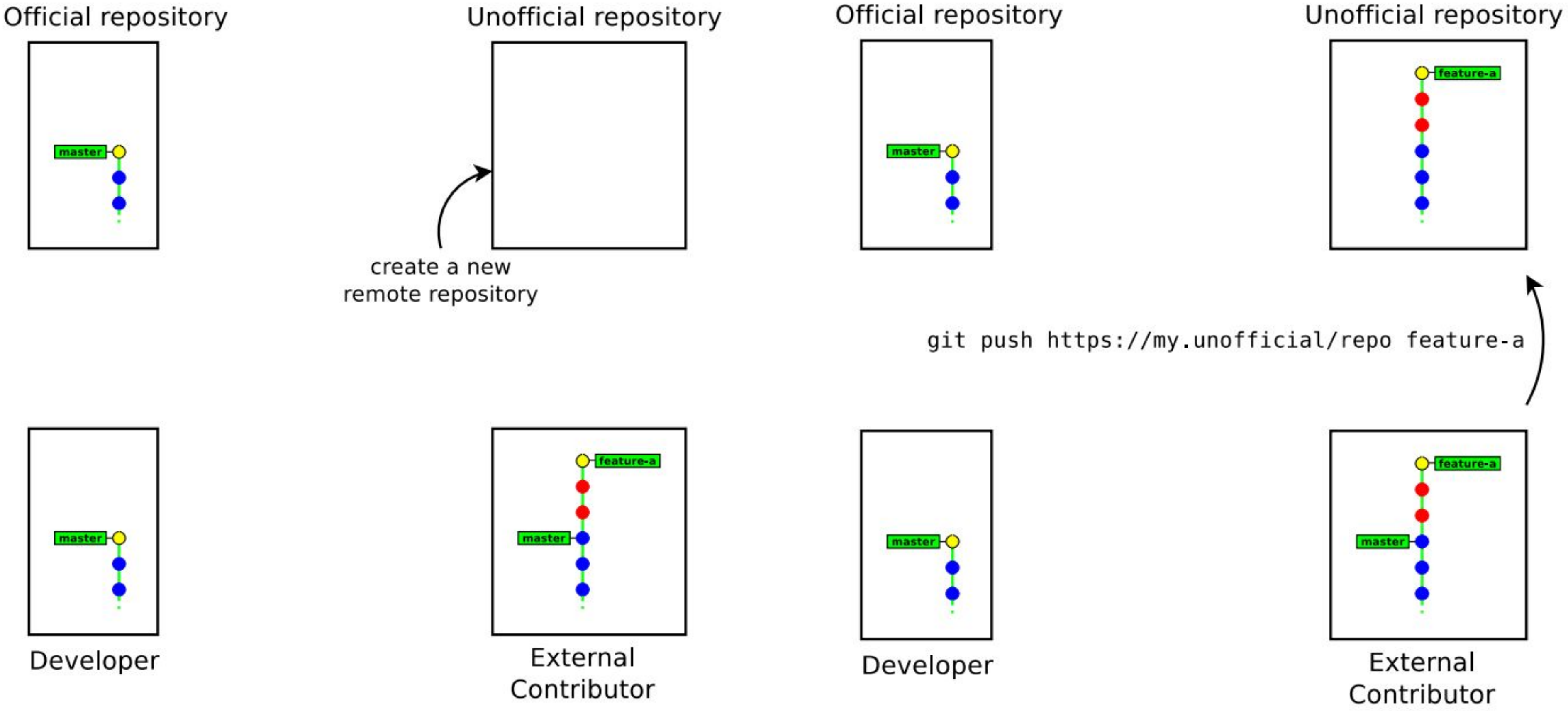
Official repository

git push

Developer

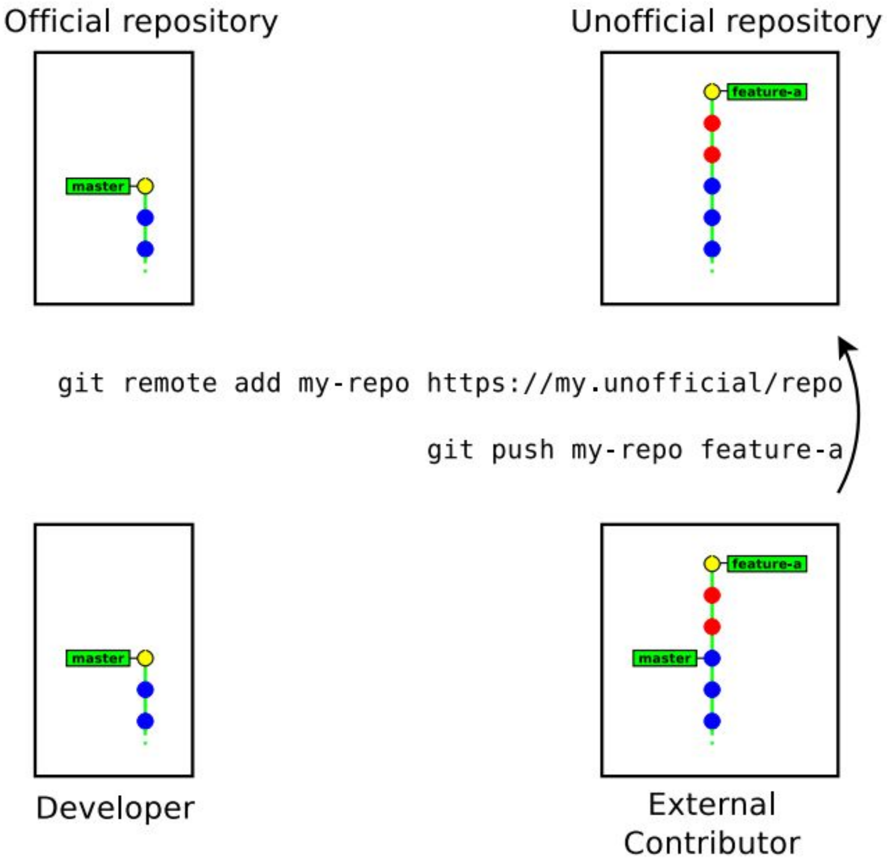
External Contributor

Decentralised workflow

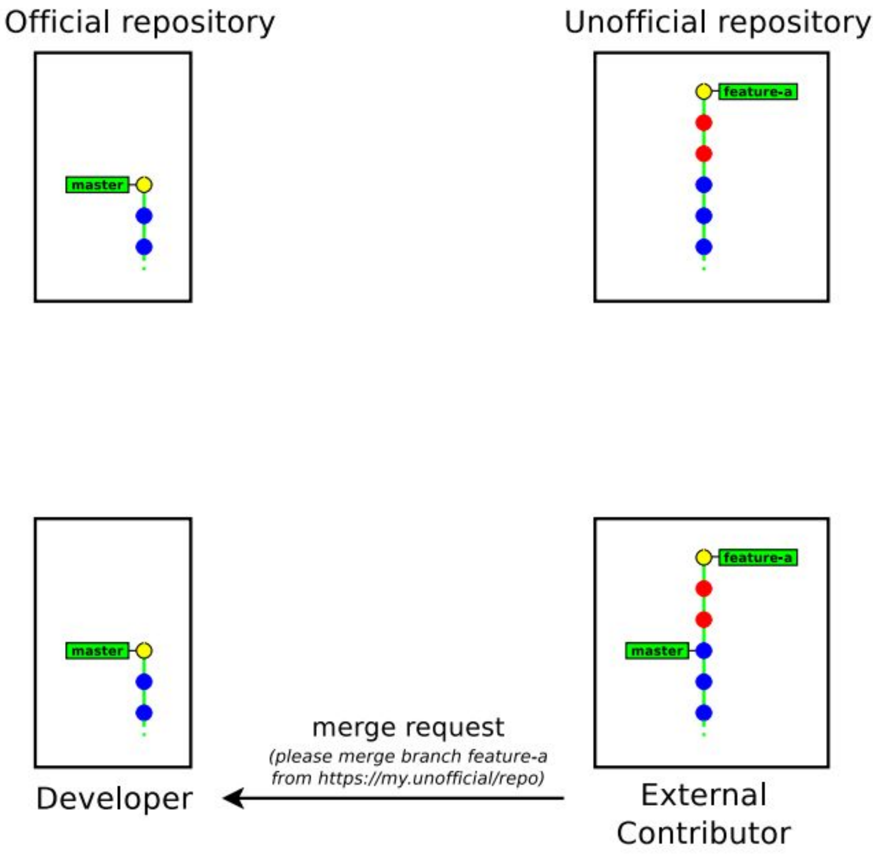


```
git push https://my.unofficial/repo feature-a
```

Decentralised workflow

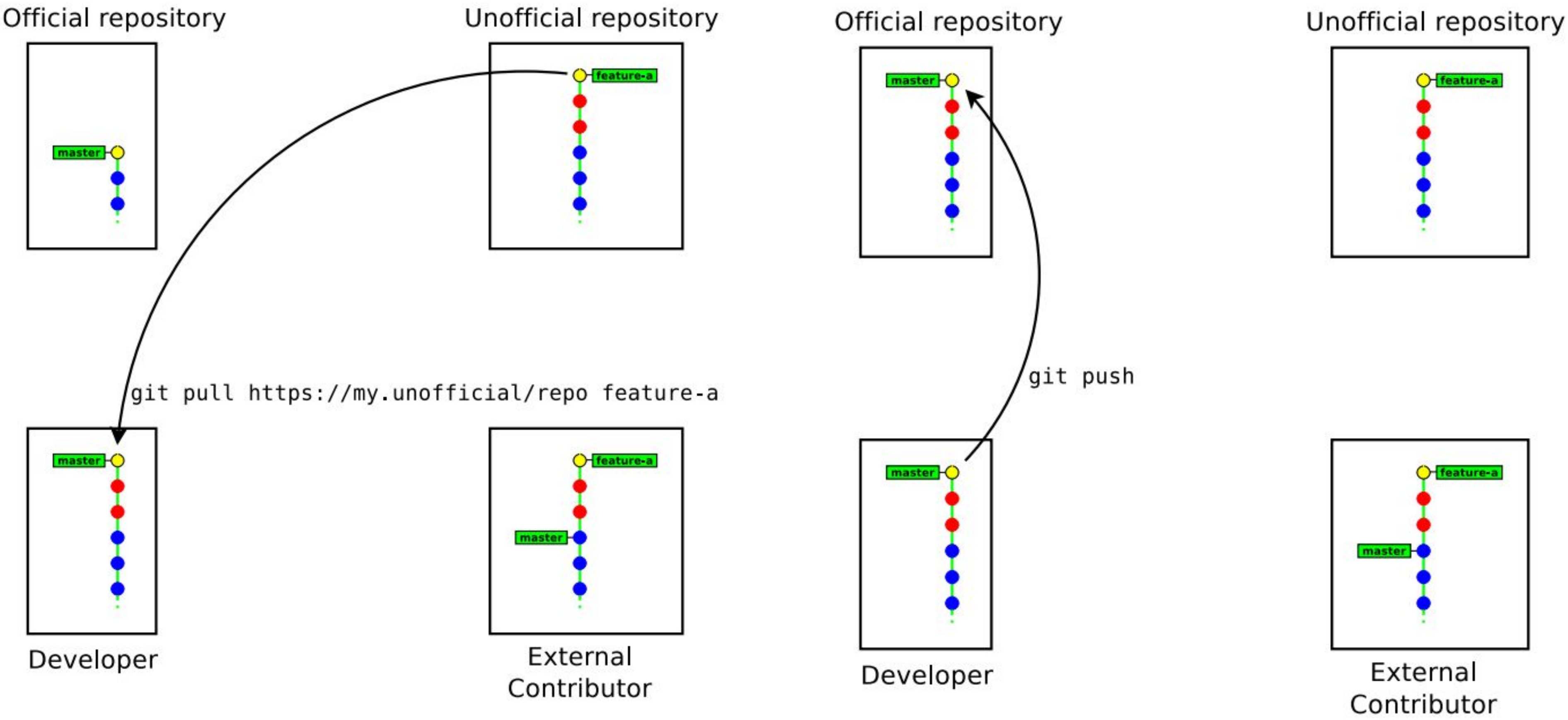


```
git remote add my-repo https://my.unofficial/repo

                       git push my-repo feature-a
```



merge request
(please merge branch feature-a from https://my.unofficial/repo)

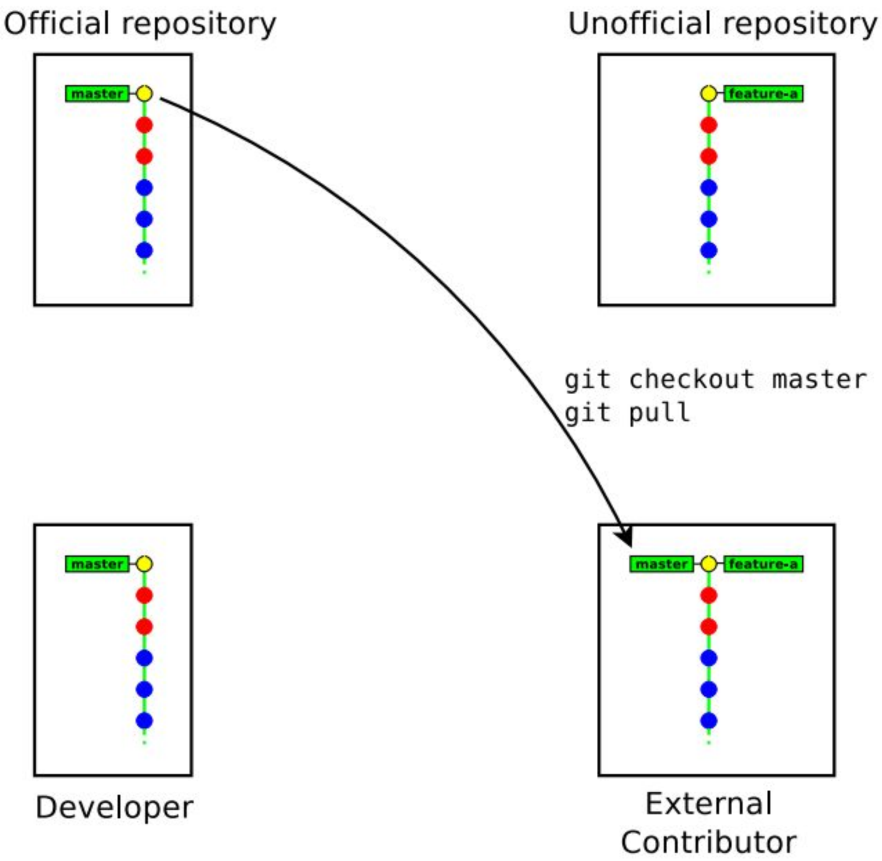
Decentralised workflow



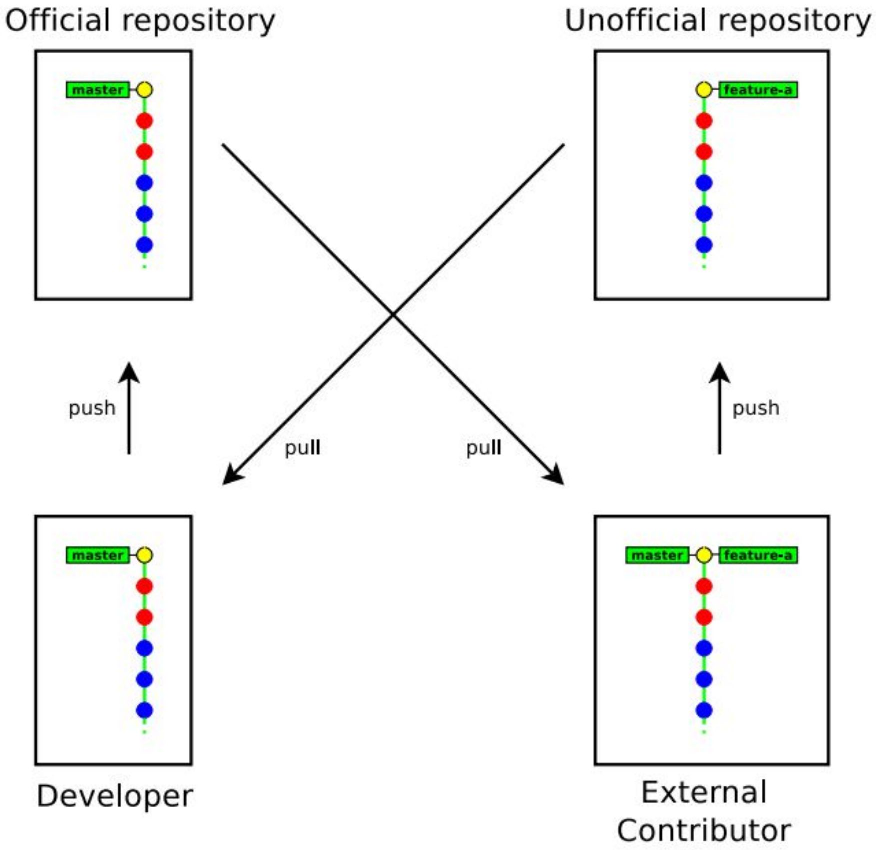
```
git pull https://my.unofficial/repo feature-a
```

```
git push
```

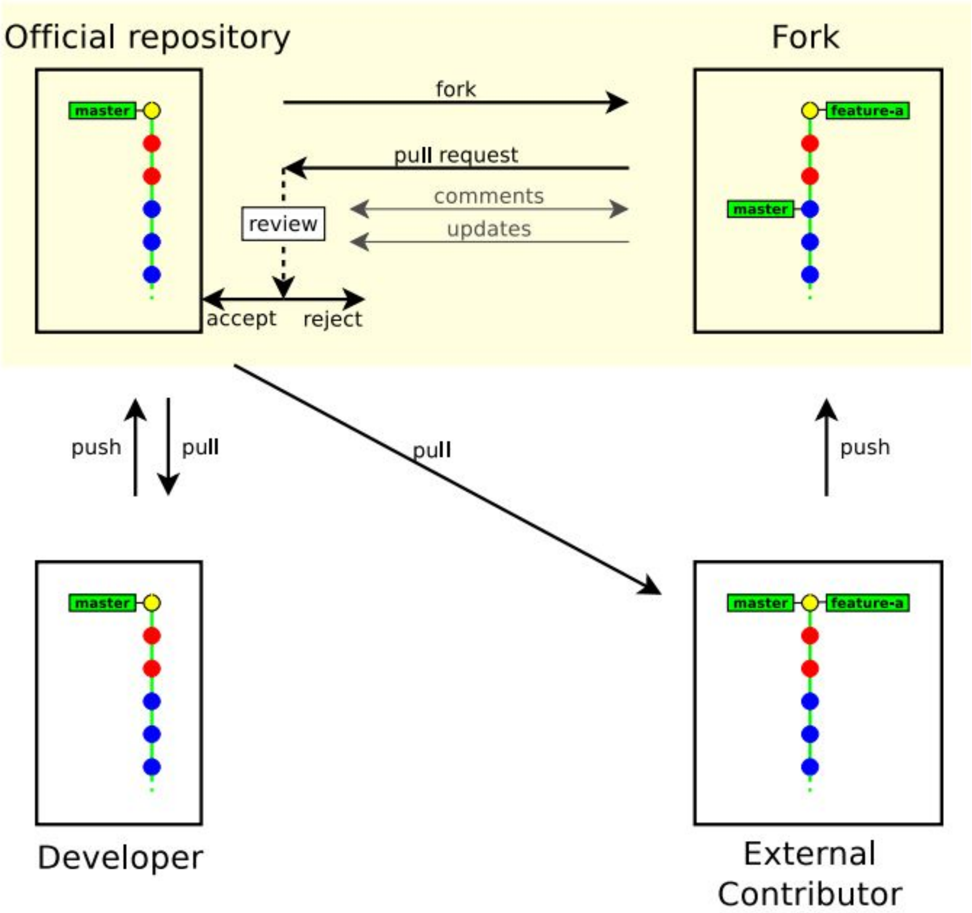
Decentralised workflow



```
git checkout master
git pull
```



GIT-centric forges



Generating patches

- **git diff**

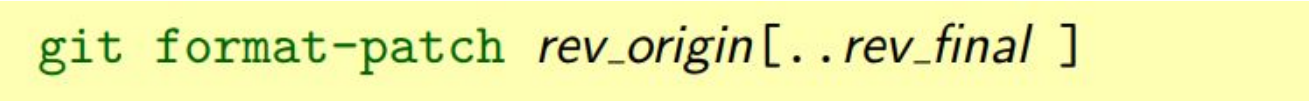
The basic (legacy) way: use git diff

- **git format-patch**

The modern way: git format-patch converts you history (commits) into a series of patches (on file per commit) and it records the metadata (author name, commit message)

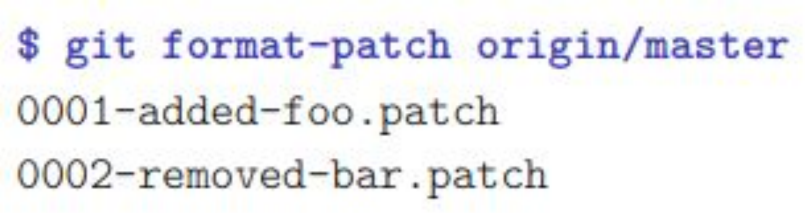
Note: git format-patch does not preserve merge history & conflicts resolution. You should only use it when your history is linear

Generating patches

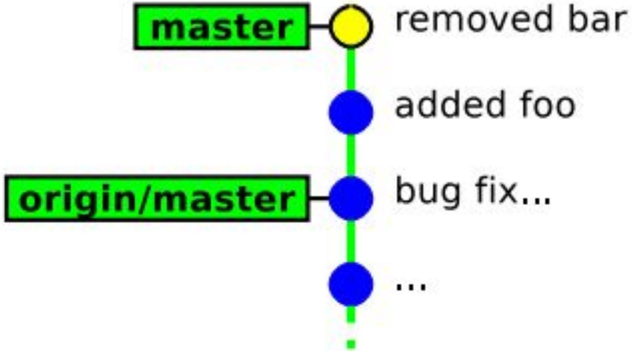


```
git format-patch rev_origin[..rev_final ]
```

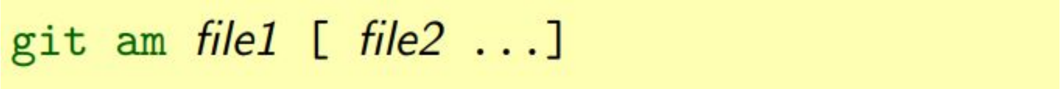
git format-patch generates patches from revision rev_origin to rev_final (or to the current version if not given)



```
$ git format-patch origin/master
0001-added-foo.patch
0002-removed-bar.patch
```



Applying patches

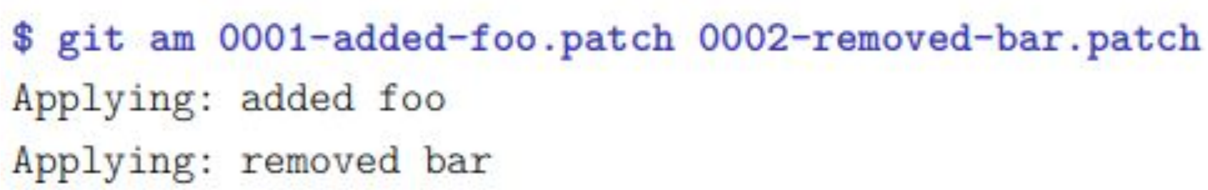


```
git am file1 [ file2 ...]
```

- **git am (am originally stands for “apply mailbox”)** applies a series of patches generated by git format-patch into the local repository (each patch produces one commit)

- **The authorship of the submitter is preserved:**

GIT distinguishes between the author and the committer of a revision (usually they refer to the same person, but not when running git am)



```
$ git am 0001-added-foo.patch 0002-removed-bar.patch
Applying: added foo
Applying: removed bar
```

Thank You!