GIT for Beginners

<https://cecs.anu.edu.au/staff/cecs-it-group/gitlab-guide>

# What is version control

Version control systems (VC)

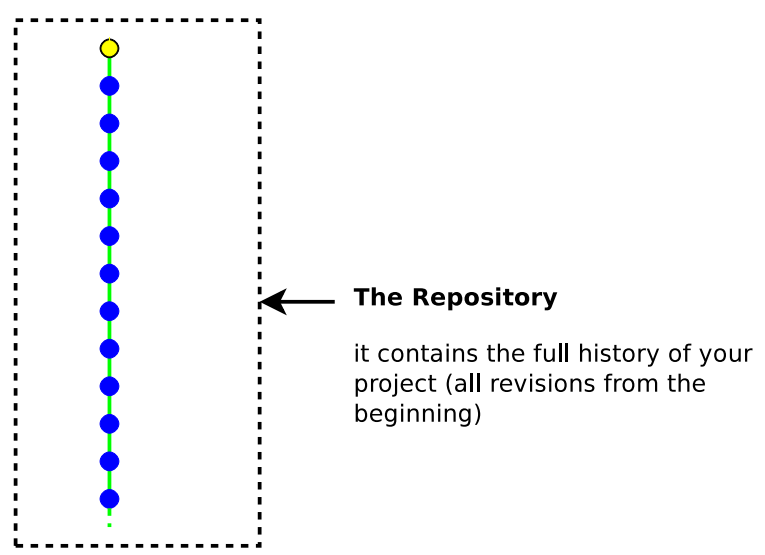
* Working with a local repository
* Synchronizing with a remote repository
* Setting up a server

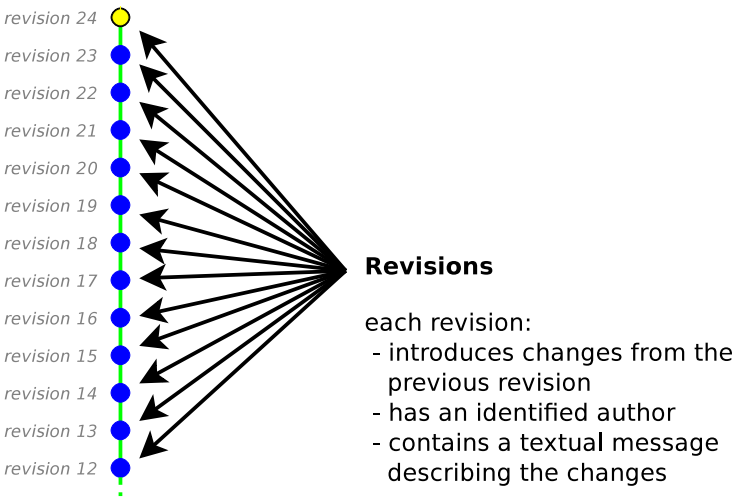
VC is the management of changes to documents

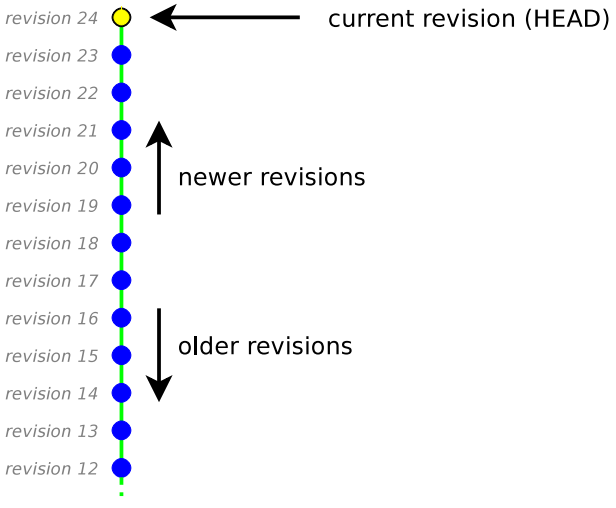
* Each change is identified by a revision number
* Each revision is associated with a timestamp and the person making the change.
* Revisions can be compared, restored or merged.

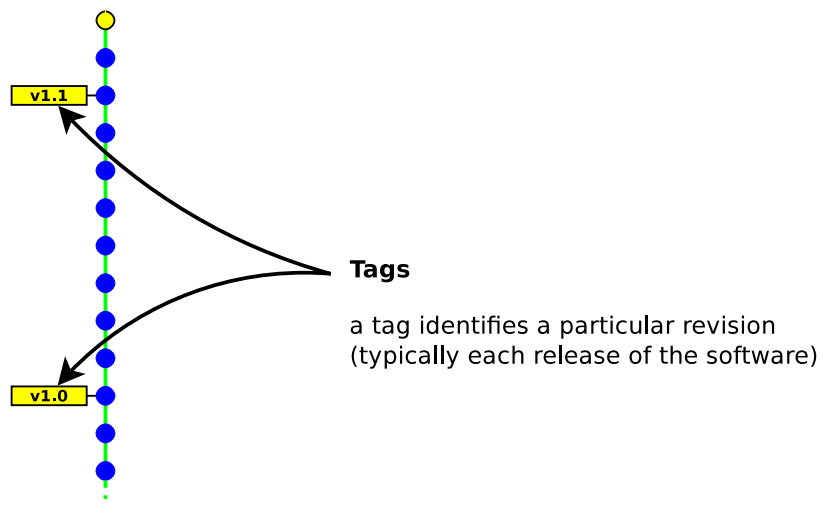
VC tools allow you to:

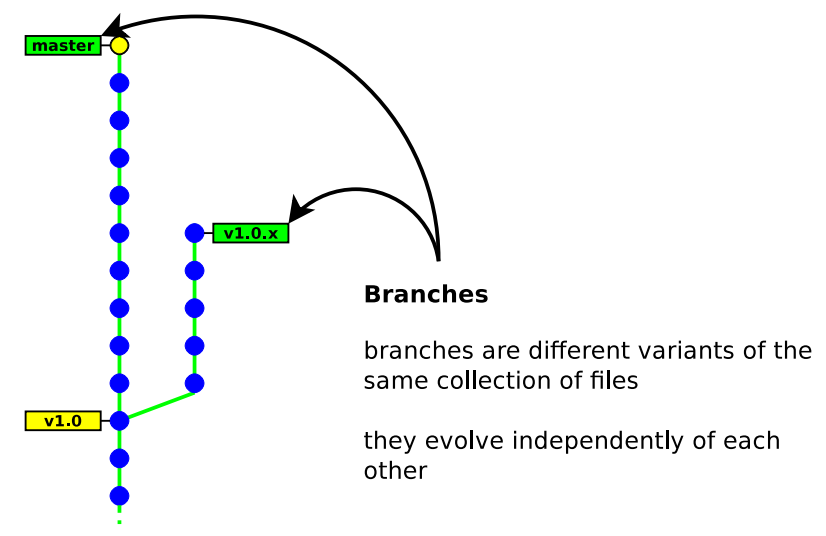
* Working with others
  + Share files with your team
  + Merge changes done by others.
  + Ensure that nothing is accidentally overwritten.
* History keeping
  + Inspect previous revisions (browsable history, when/who/what/why/in which context for the changes)
  + Revert to a previous revision.
  + Deleted content remains accessible in the history
* Branching
  + Multiple variants of the same software are called branches (main branch, maintenance branch for fixing bugs, development branch for disruptive changes, release branch for final release)
  + Handle multiple branches concurrently.
  + Merge changes from a branch into another one.
* Working with external distributors, allowing them to
  + (provide visibility) to see what’s happening in the project
  + To submit changes (patches) and allow yu to integrate these patches.
  + Fork the development of a software (to make a local copy and participate into development) and merge it back to the mainline.

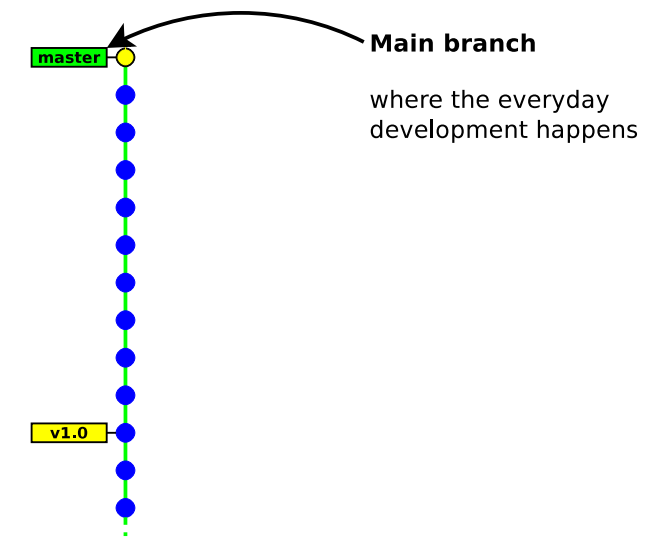


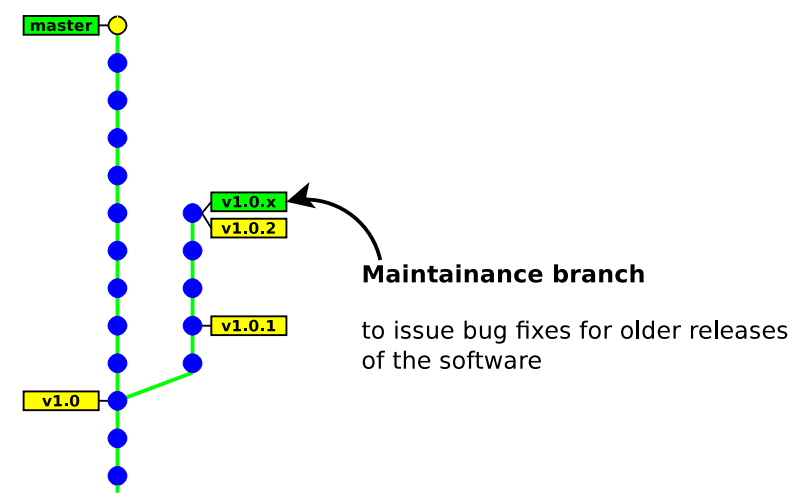


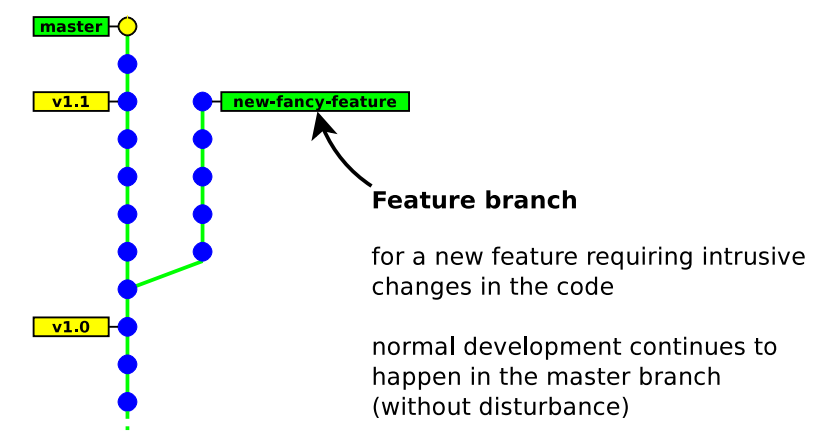


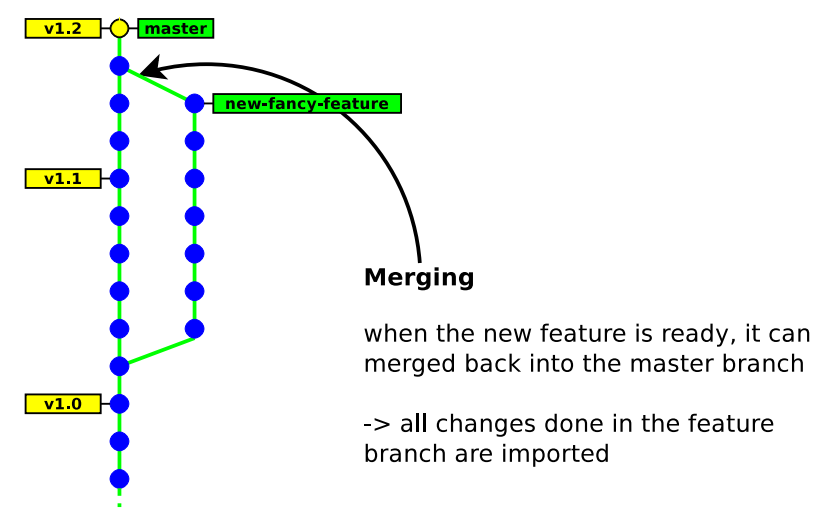












Architecture:

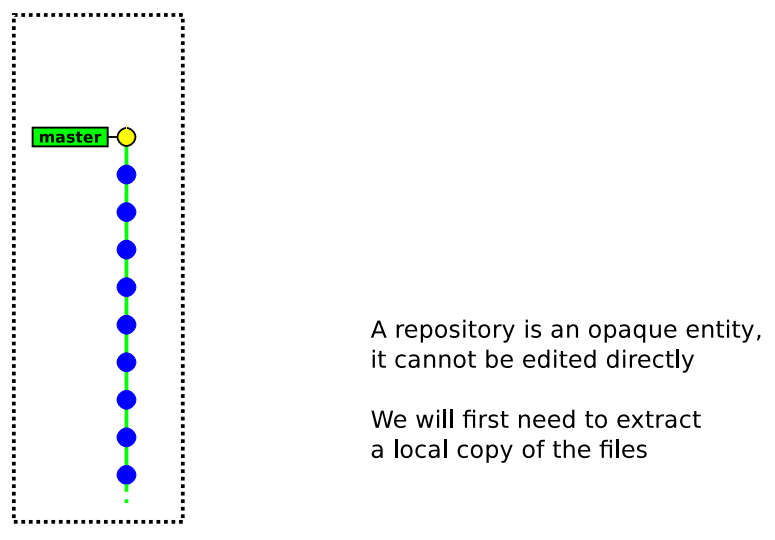
* Centralised ! everyone works on the same unique repository
* Decentralised ! everyone works on his own repository

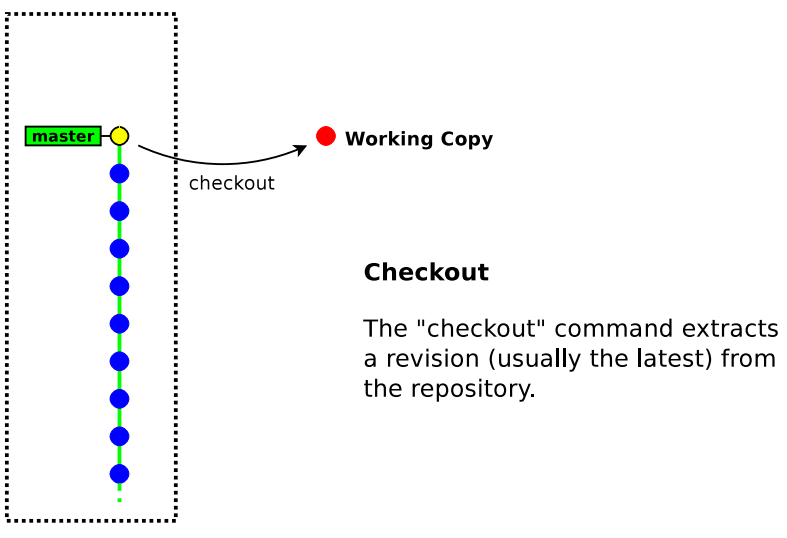
Concurrency model:

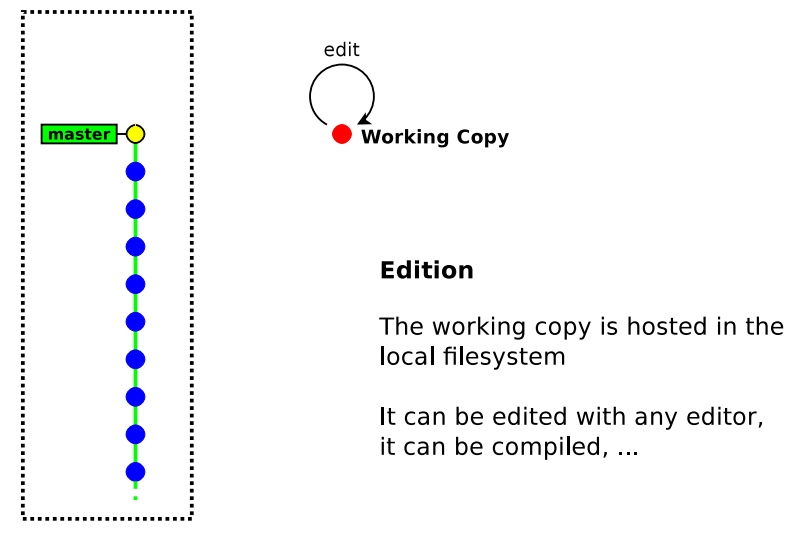
* Lock before edit (mutual exclusion)
* Merge after edit (may have conflicts)

## Creating new revisions

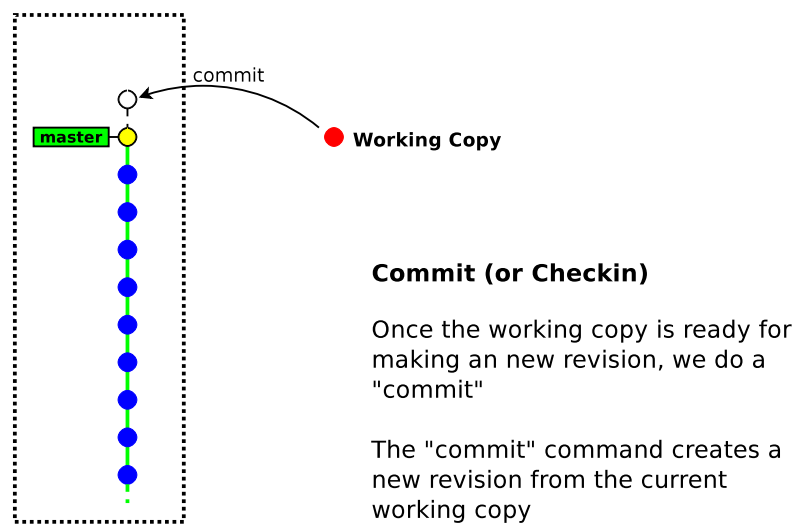
### Checkout

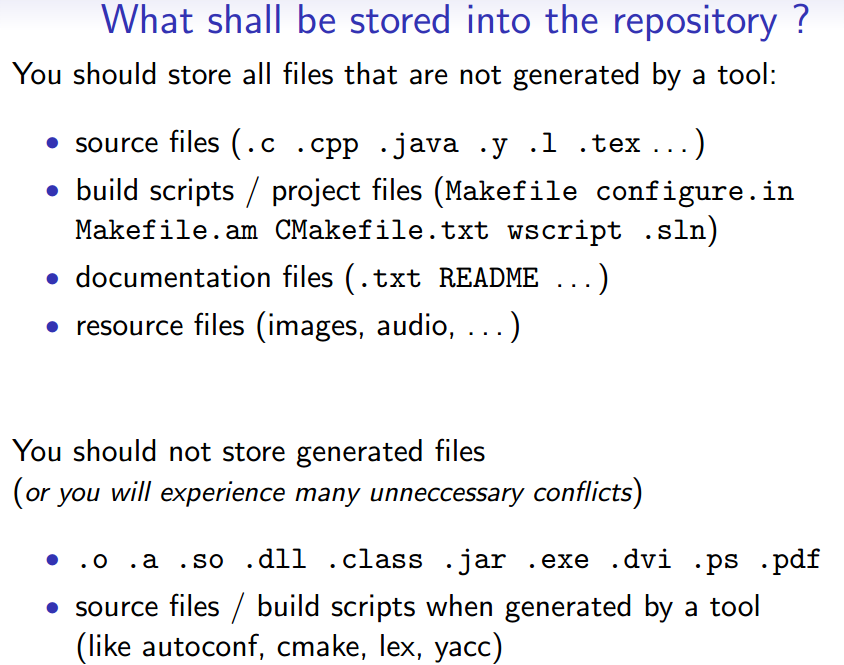


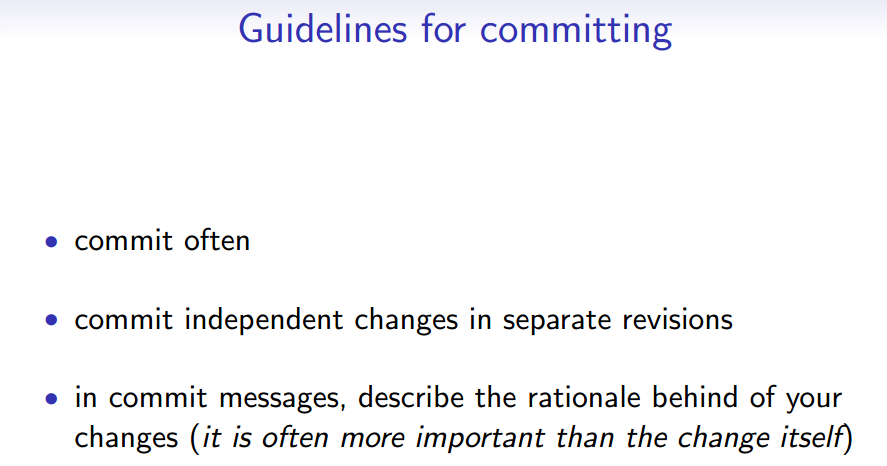




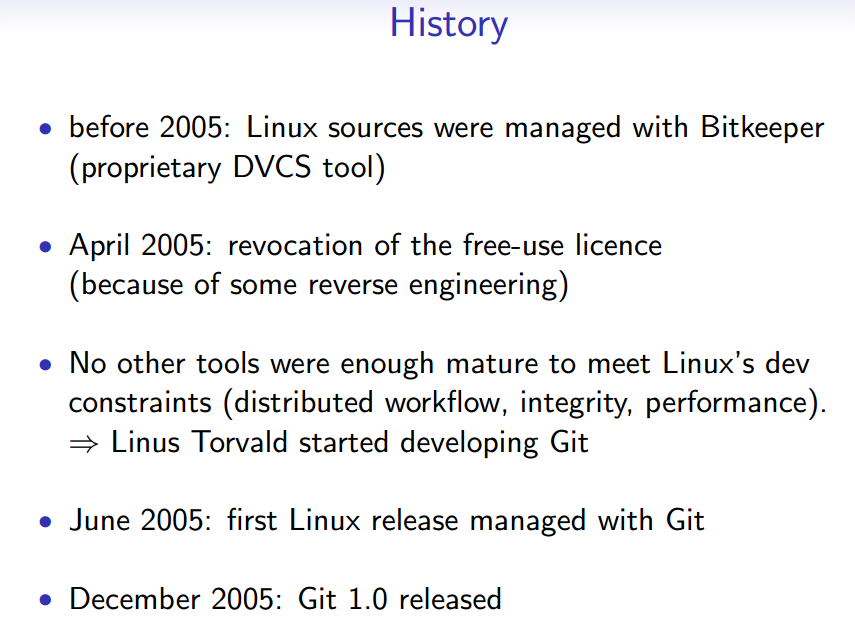
### Checkin/Commit

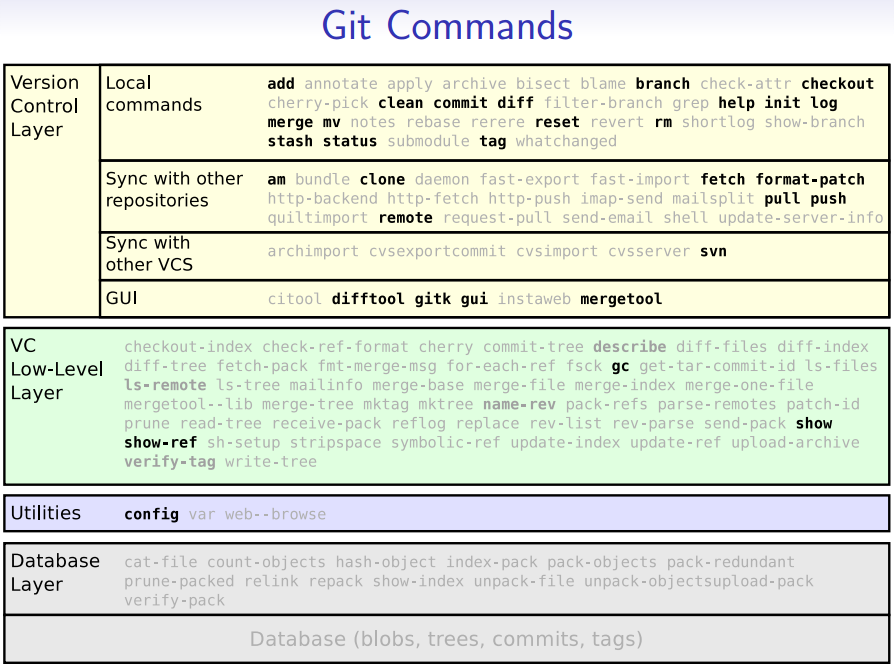




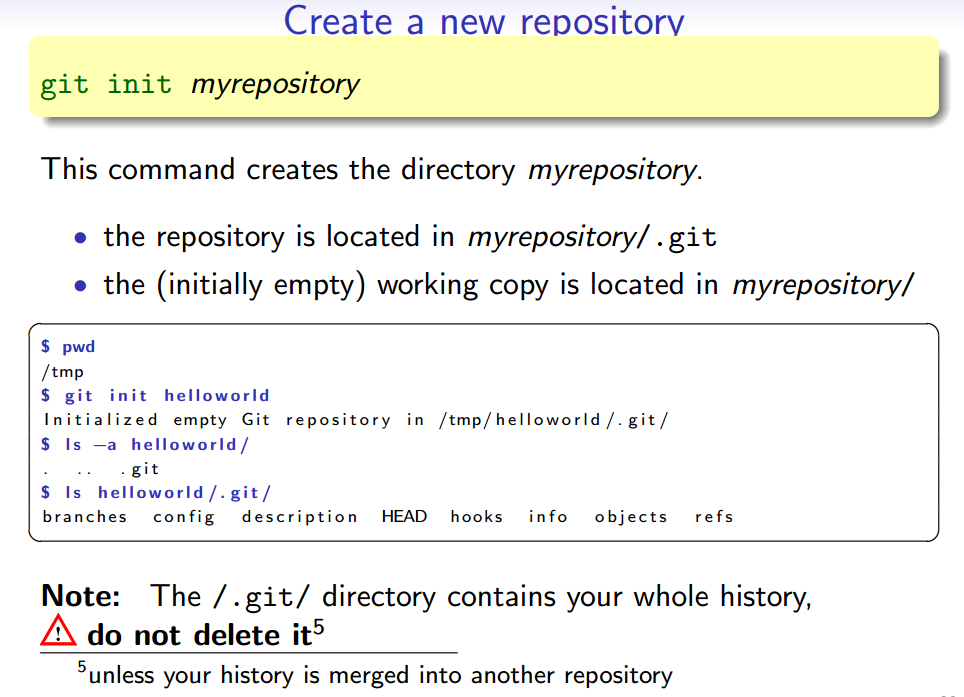


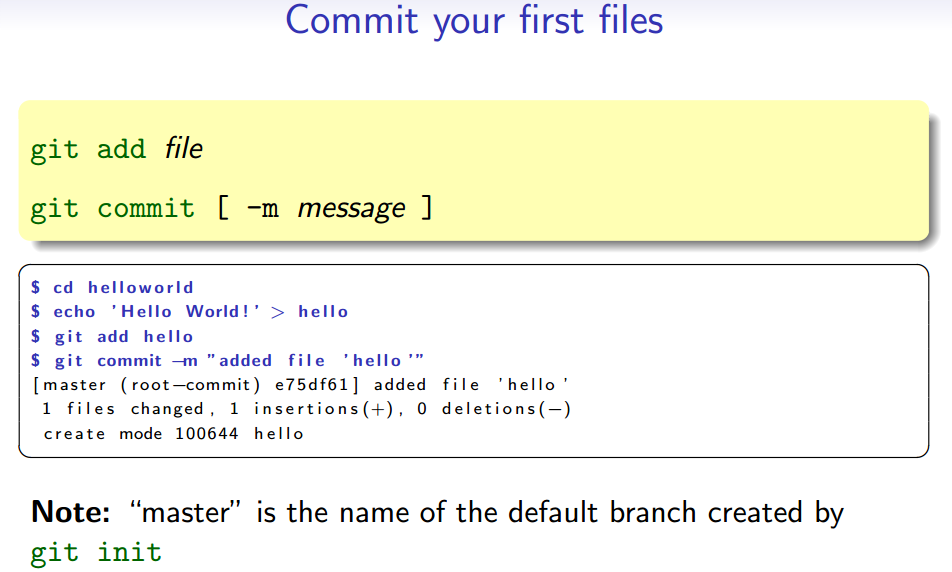
# GIT Overview

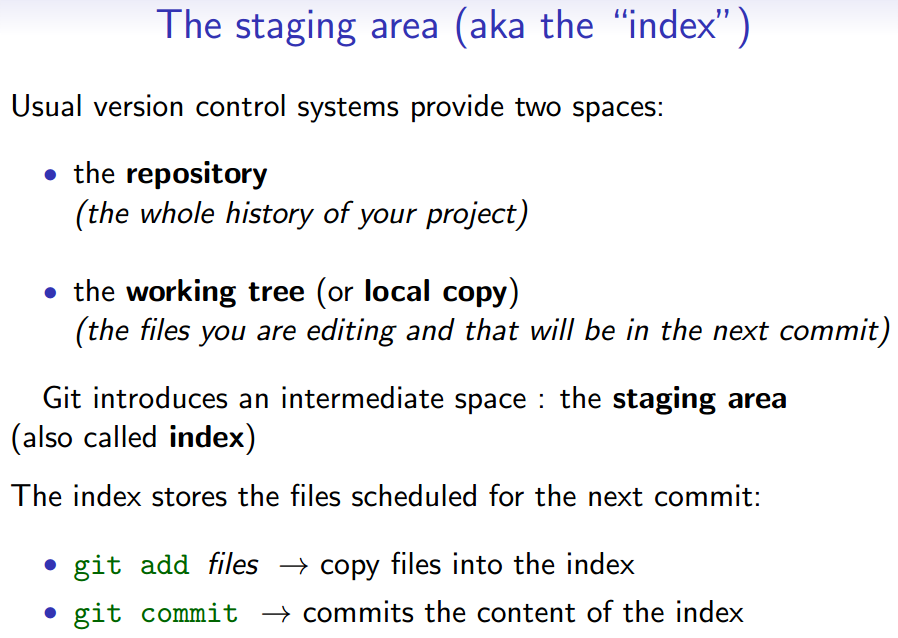


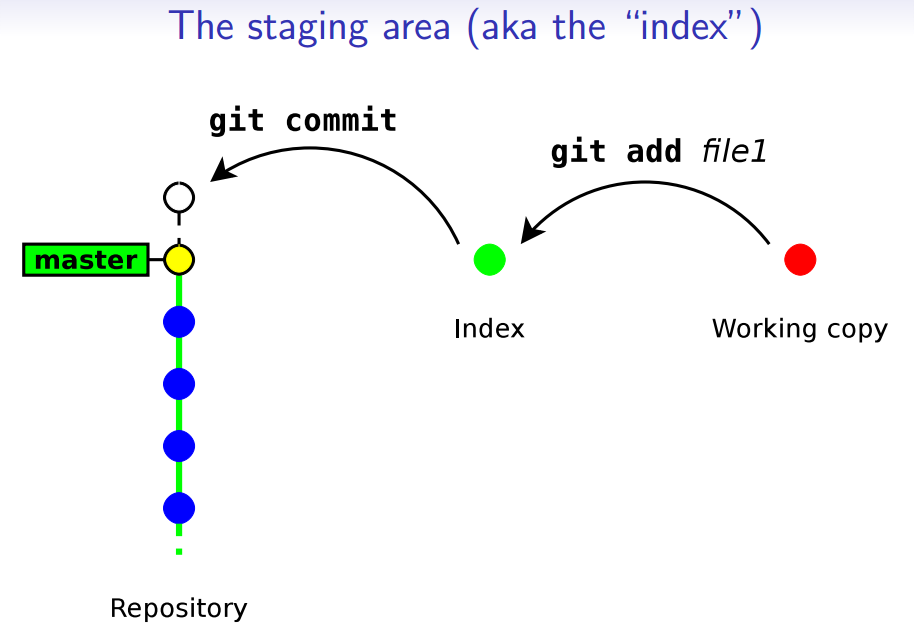


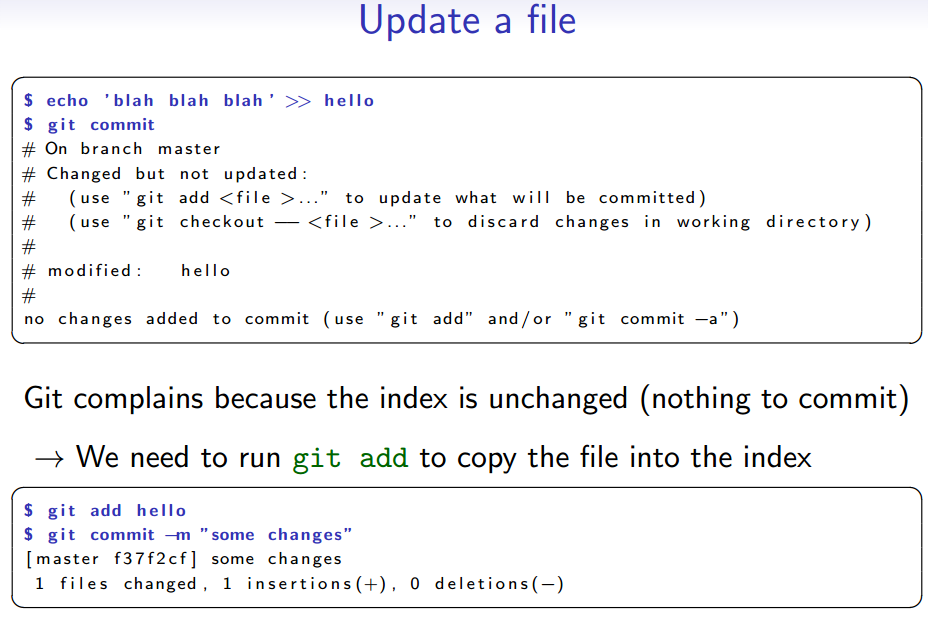
# Working locally

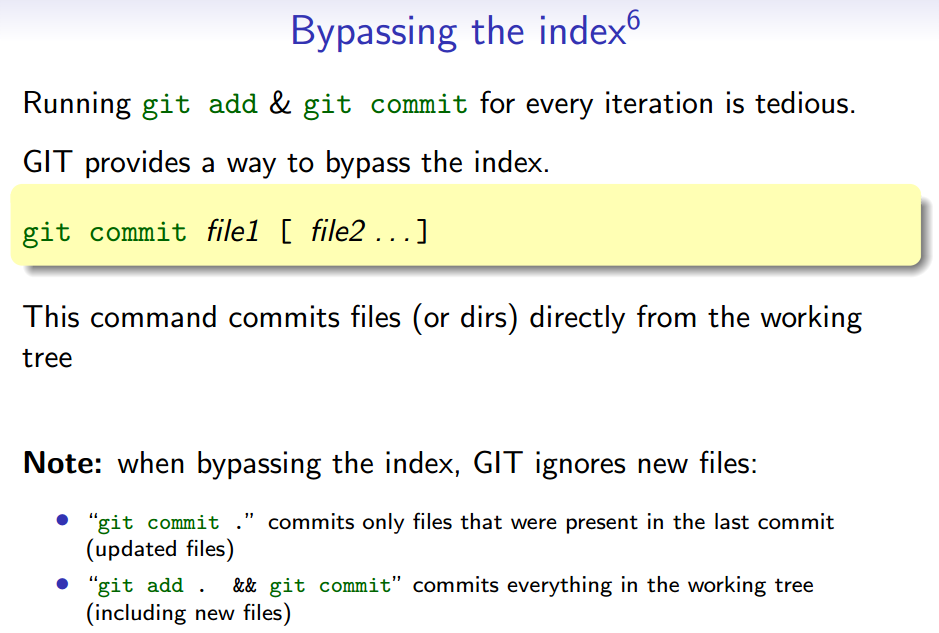


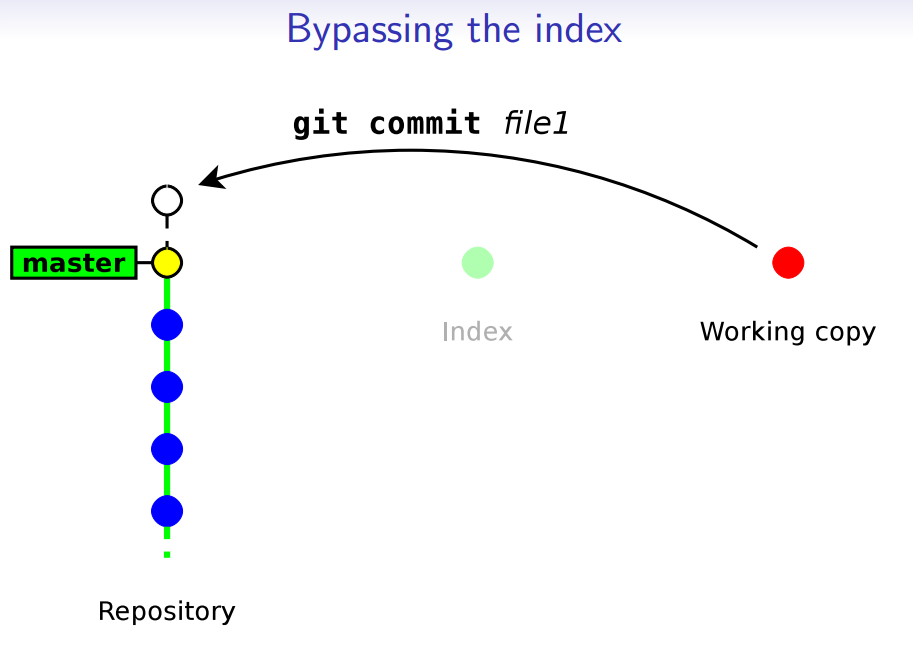


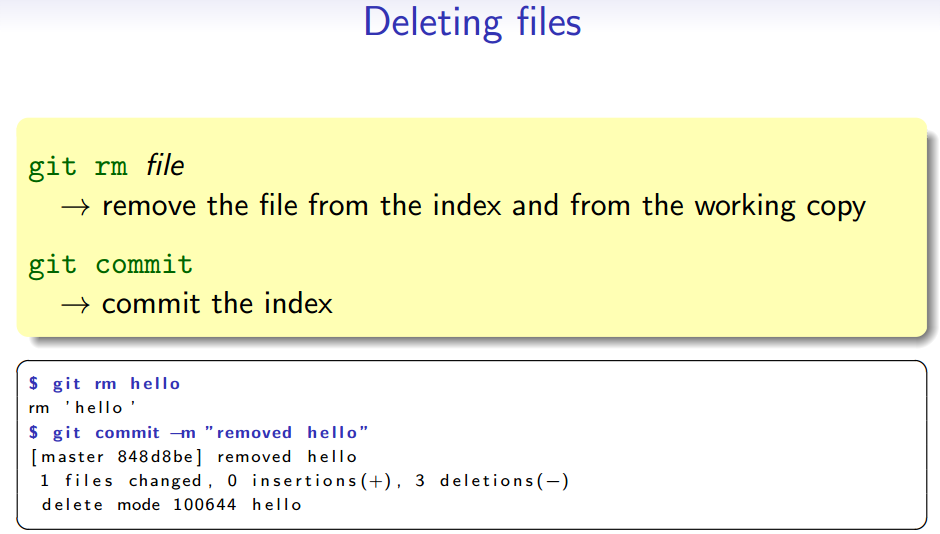


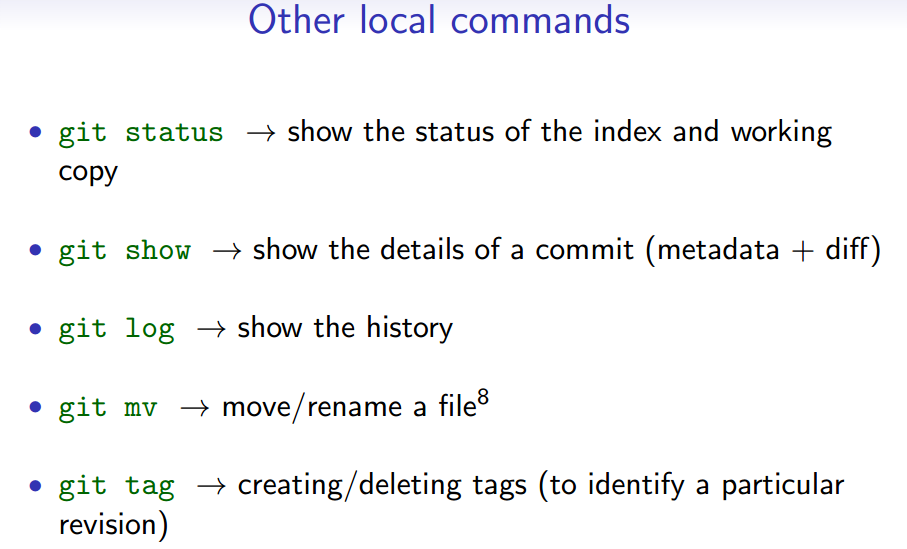




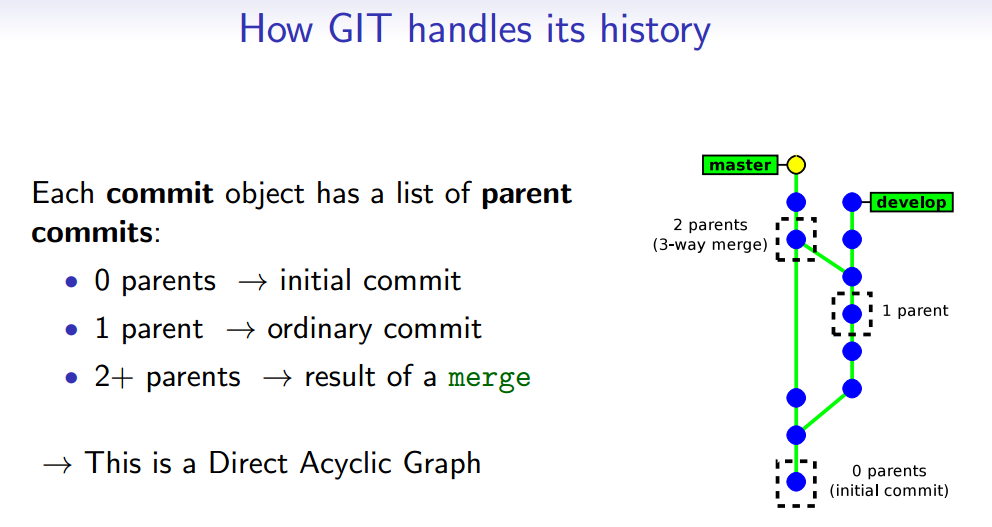


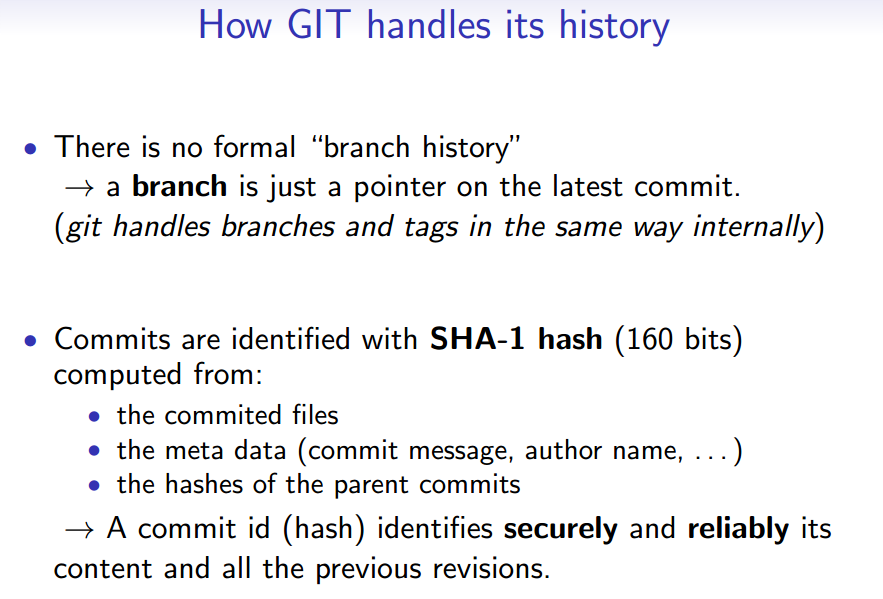


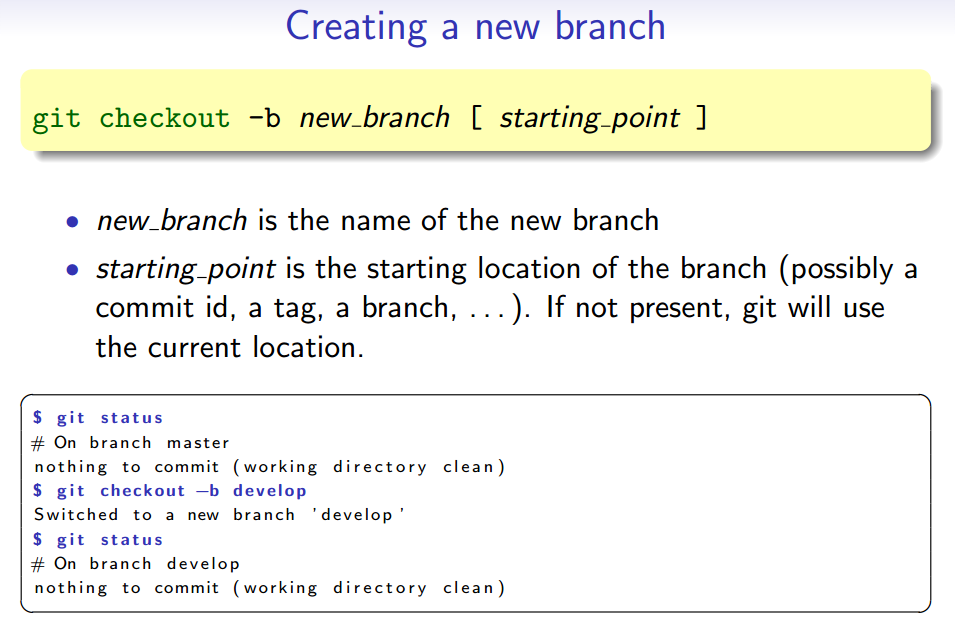


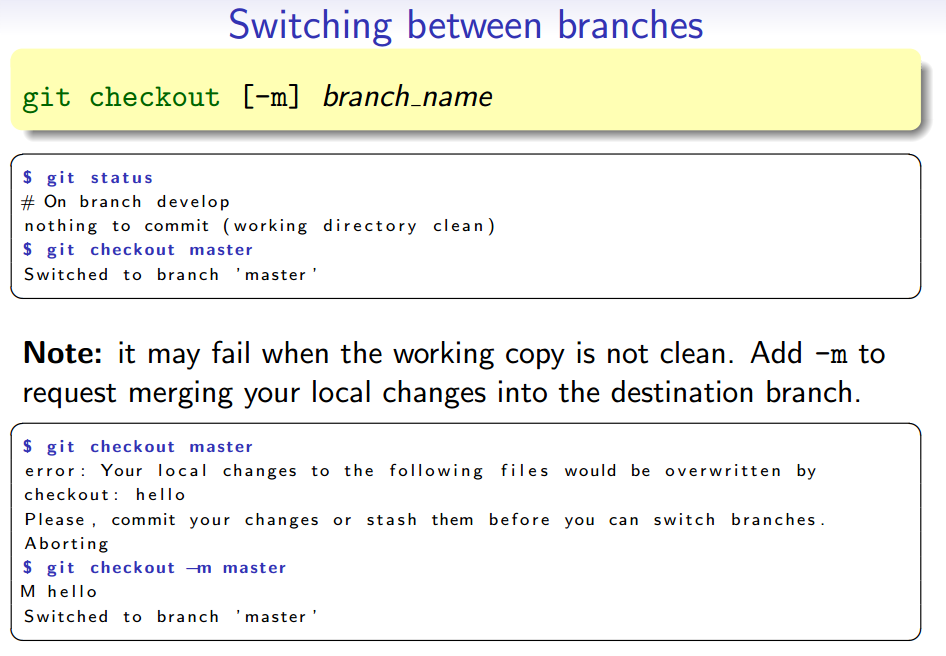


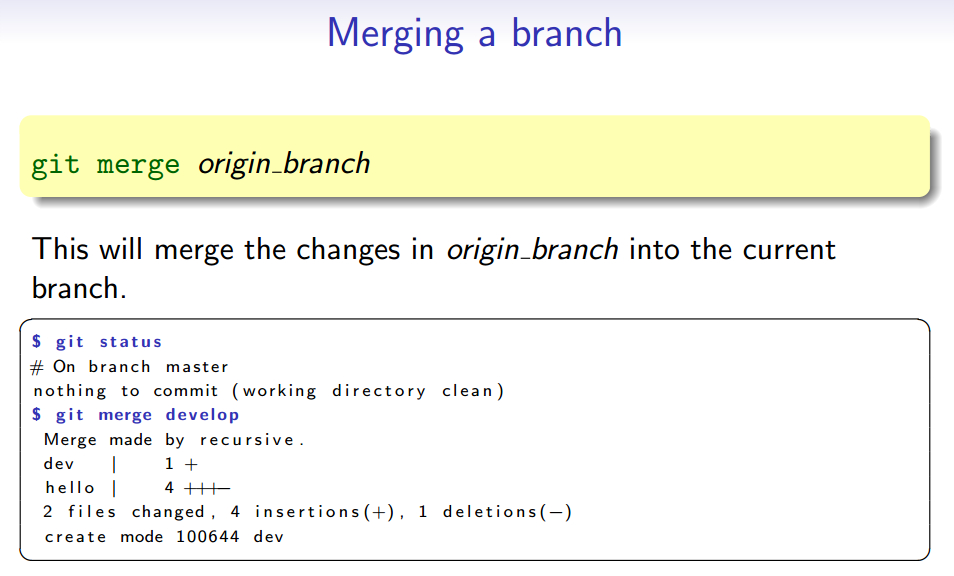
# Branching

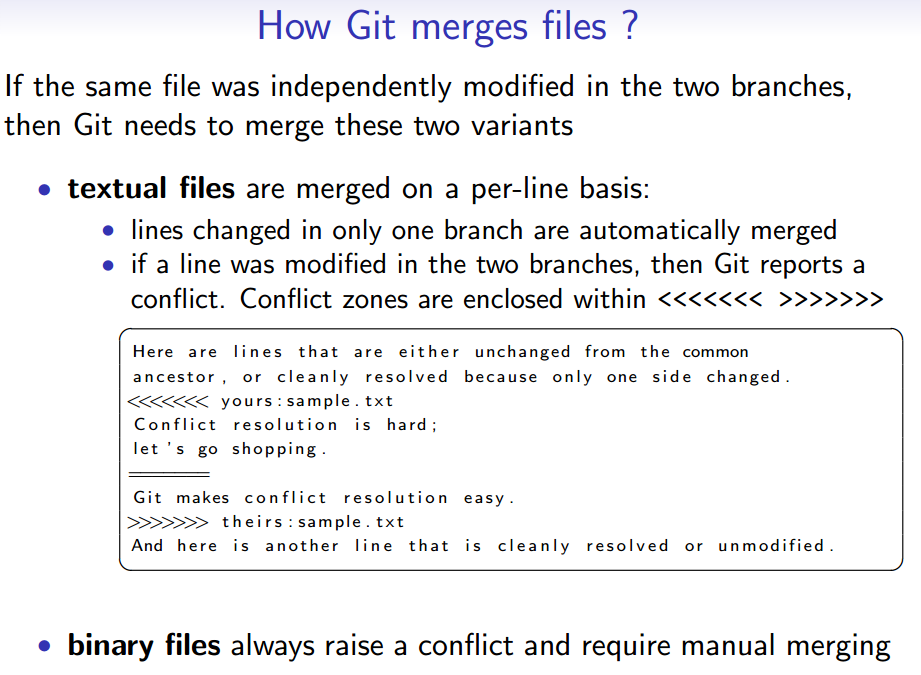


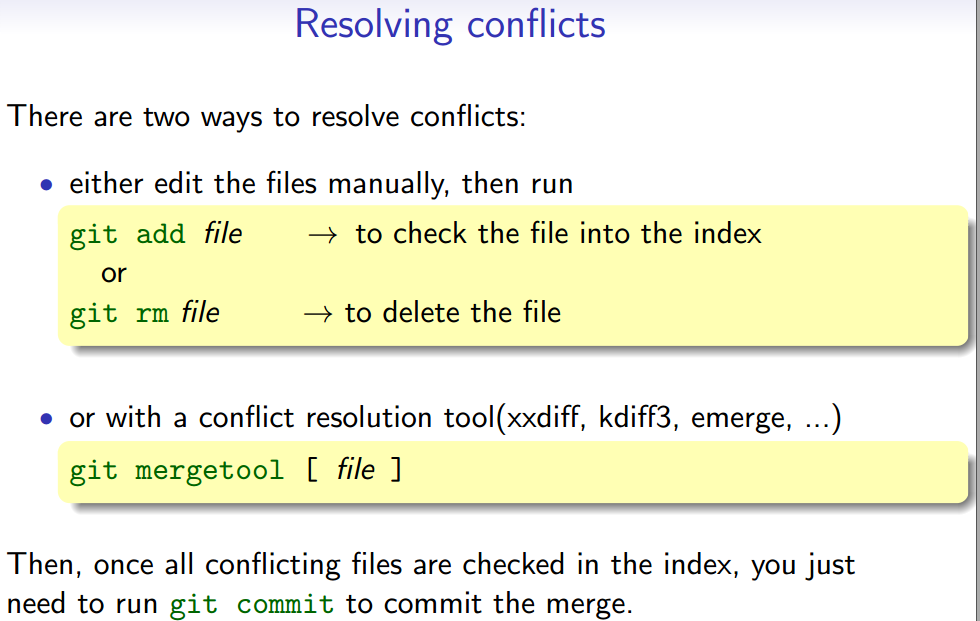






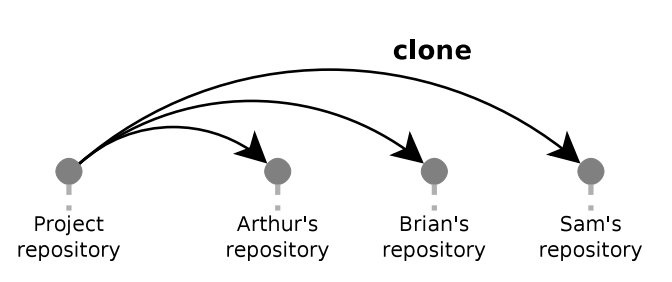




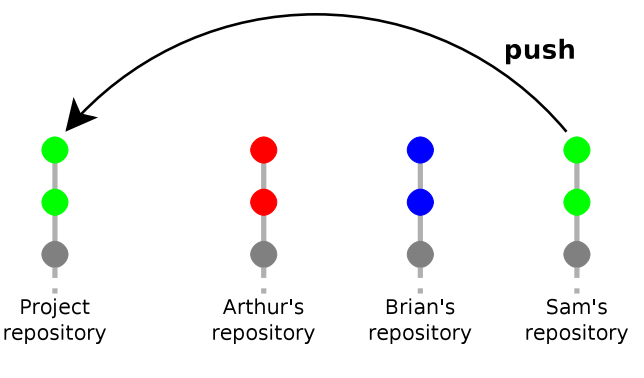
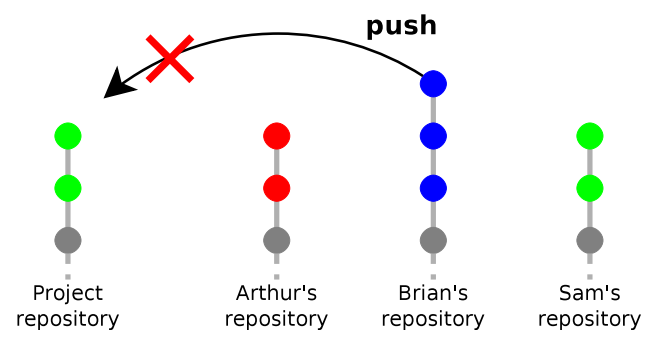


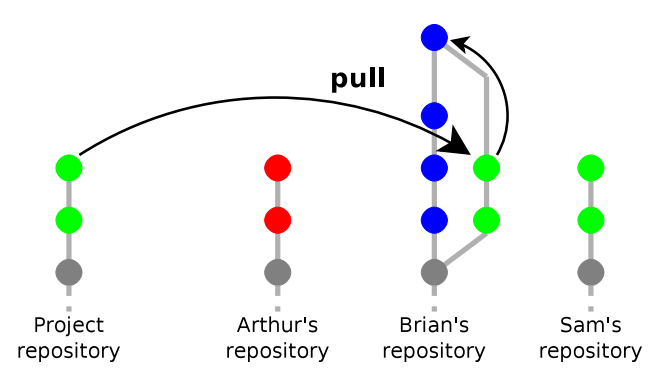
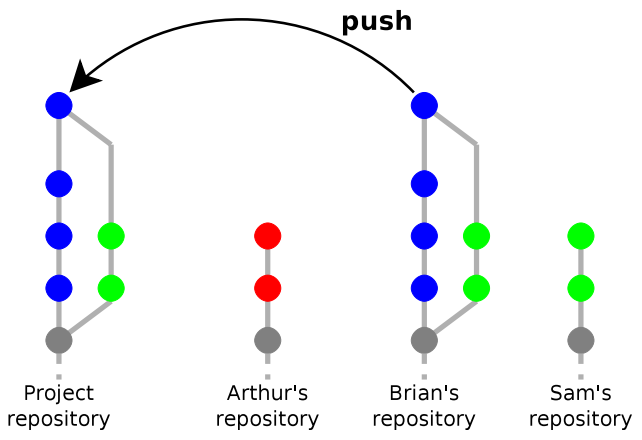
# Interacting with remote GIT server

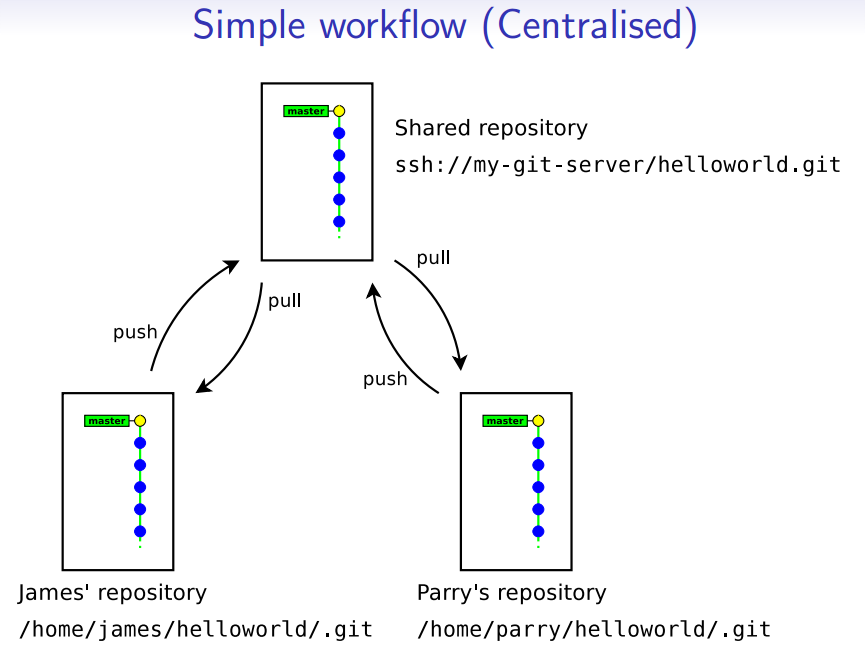
Clone

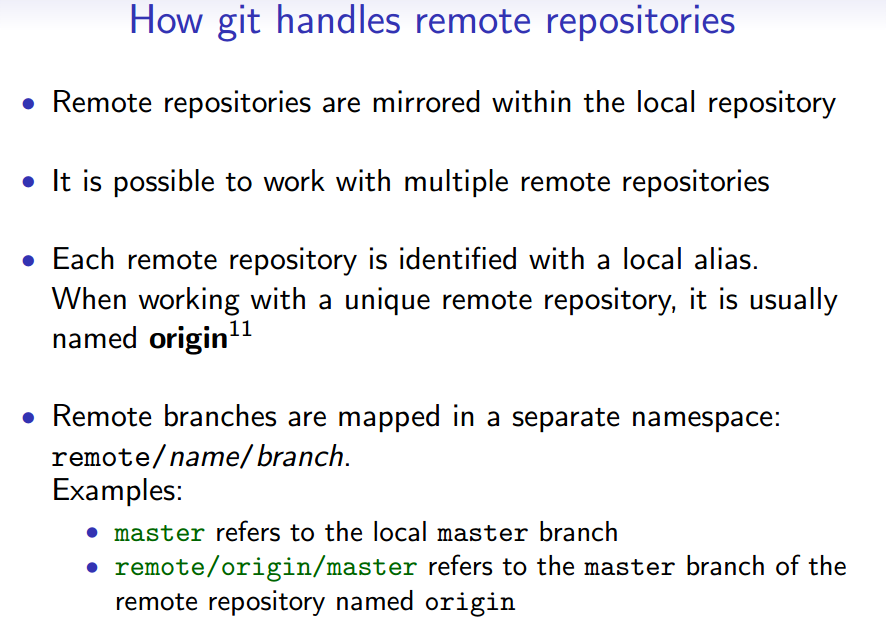


Push

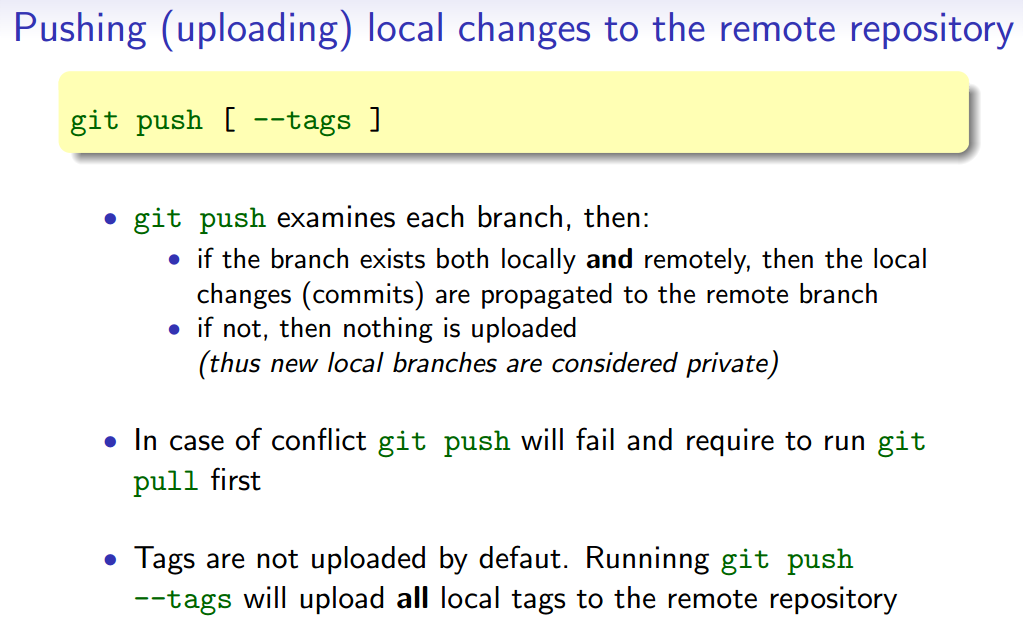
 

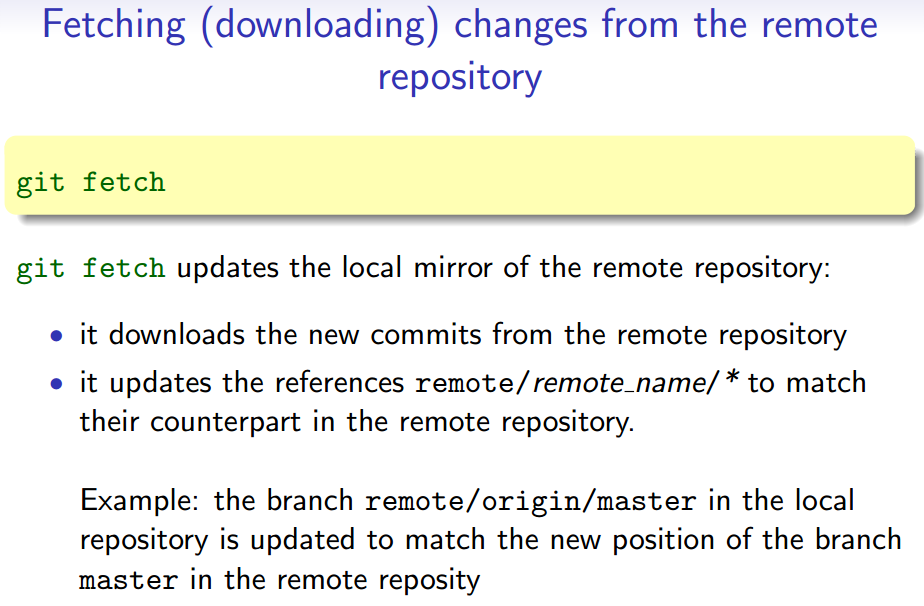
 

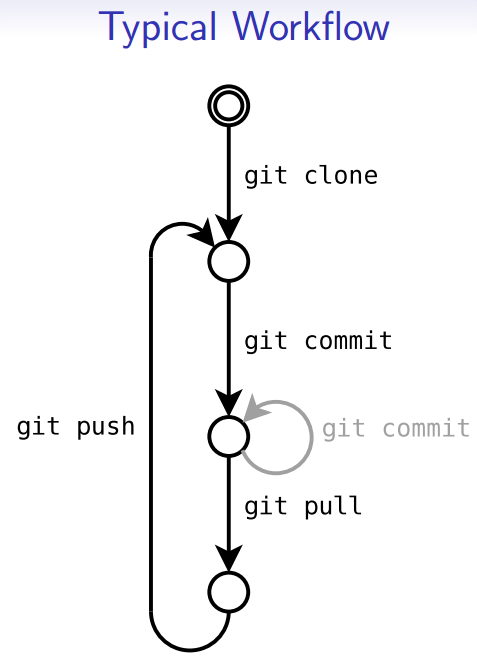
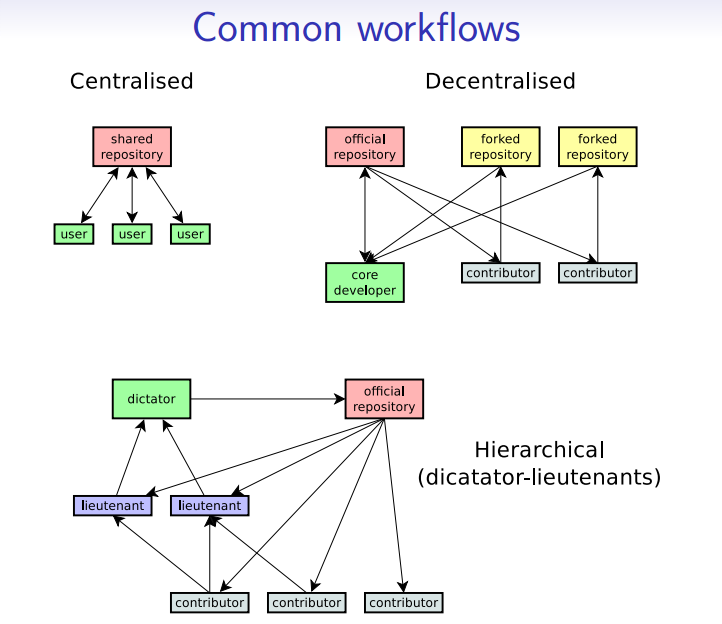


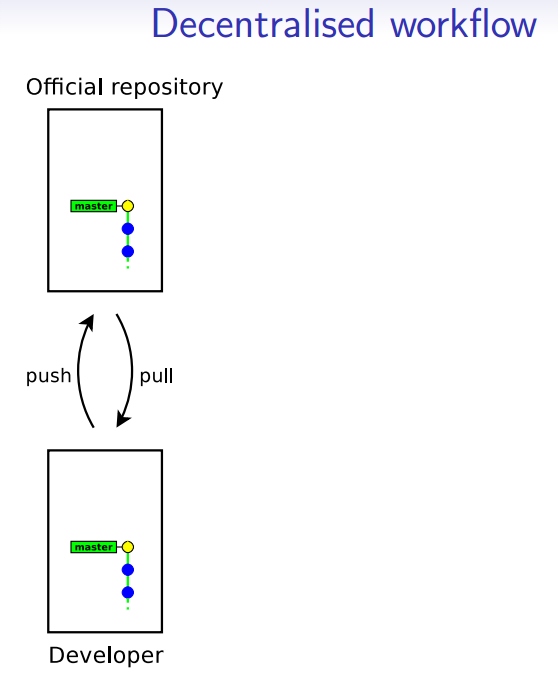
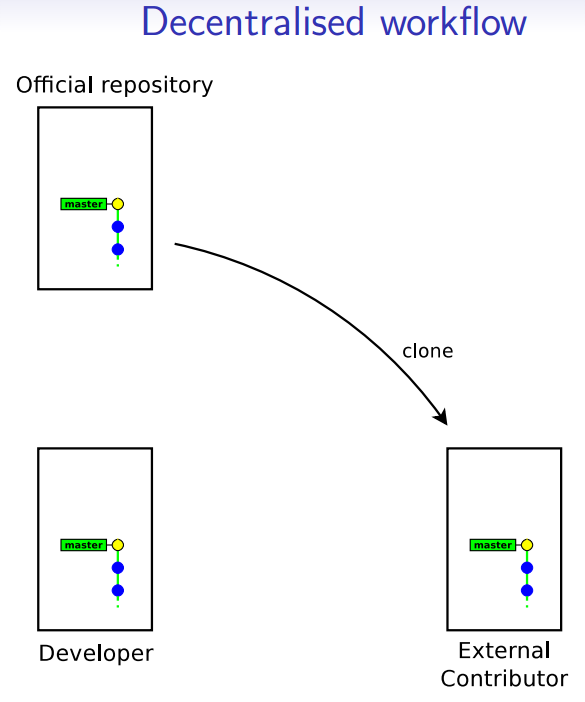




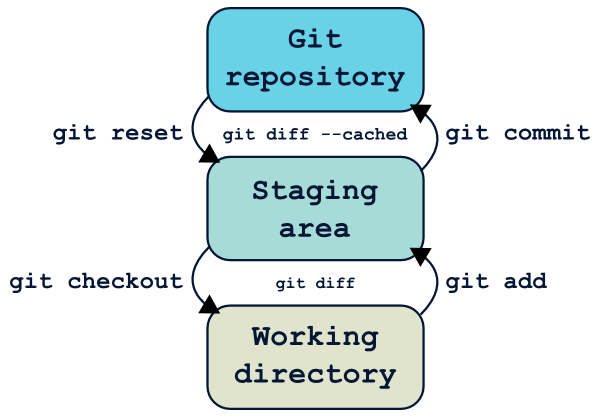


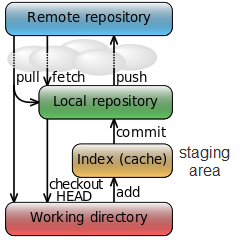


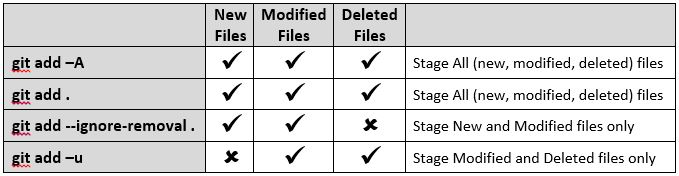
 

# Illustrations







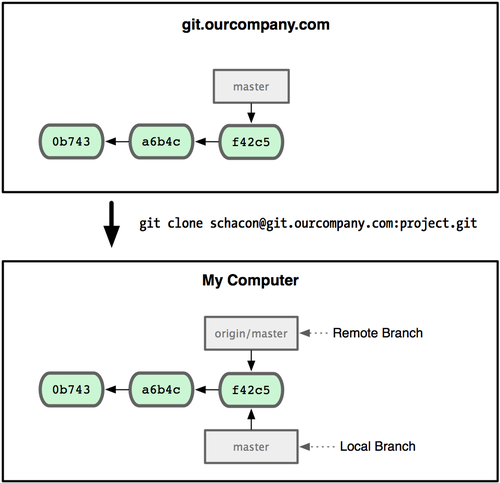
git add -A is equivalent to git add .; git add -u.

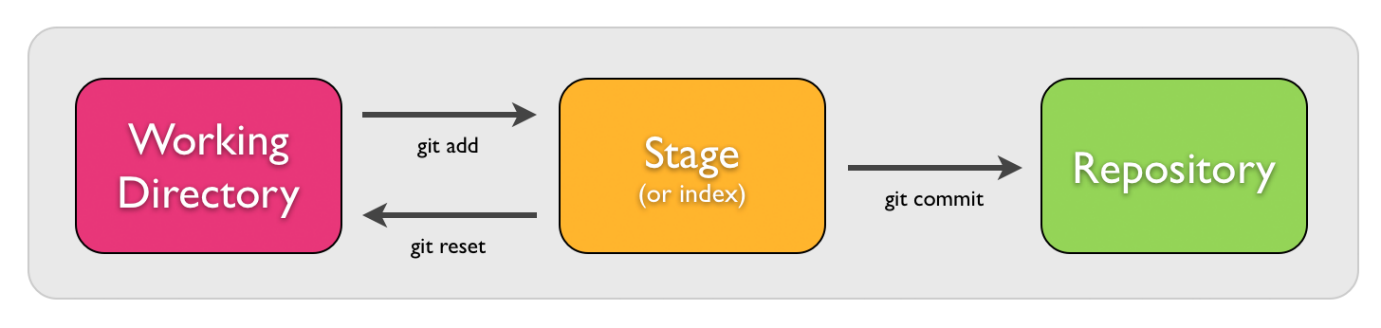
The important point about git add . is that it looks at the working tree and adds all those paths to the staged changes if they are either changed or are new and not ignored, it does not stage any 'rm' actions.

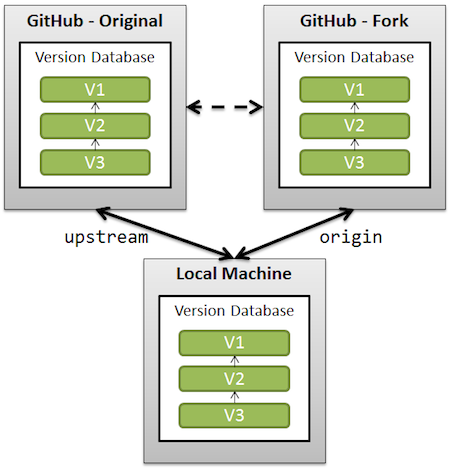
git add -u looks at all the already tracked files and stages the changes to those files if they are different or if they have been removed. It does not add any new files, it only stages changes to already tracked files.

git add -A is a handy shortcut for doing both of those.

* git add -A stages **All**
* git add . stages new and modified, **without deleted**
* git add -u stages modified and deleted, **without new**



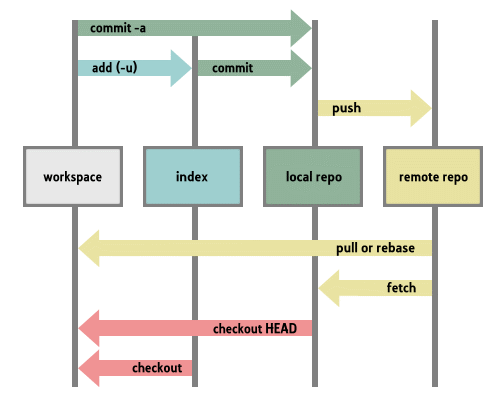


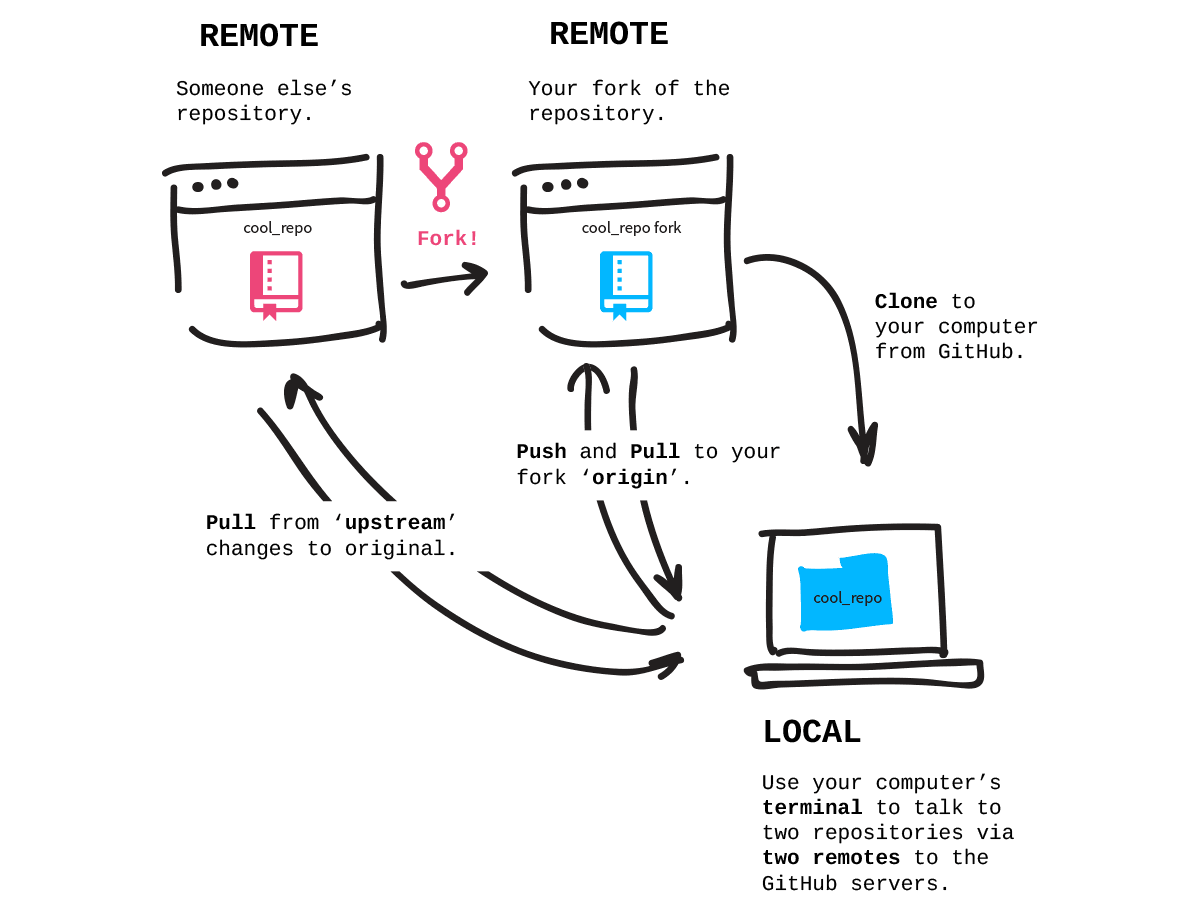


* origin is added for you when cloning your fork on GitHub.
* but you need to git remote add upstream https://github.com/user/reponame in order for your local clone to keep up-to-date with the original repo (the one you cannot contribute directly, which is why you had to make a fork on GitHub in the first place).

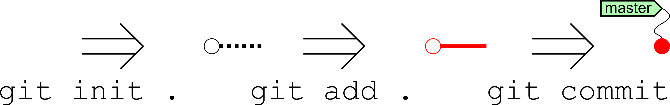
Refreshing a local Git repository with the changes from the remote repository (Pull) involves retrieving changes (Fetch) and applying them to the local data (Merge).

Once the clone is complete your repo will have a remote named “origin” that points to your fork on GitHub.  
Don’t let the name confuse you, this does not point to the original repo you forked from. To help you keep track of that repo we will add another remote named “upstream”:

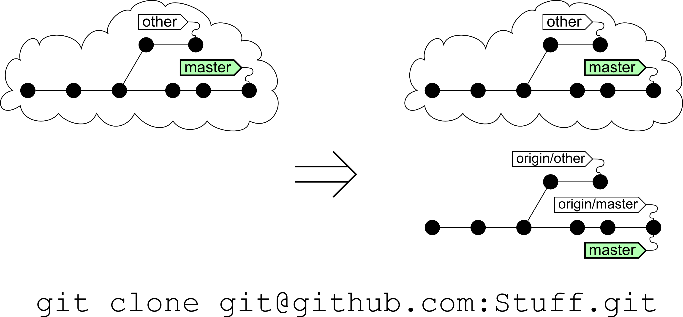




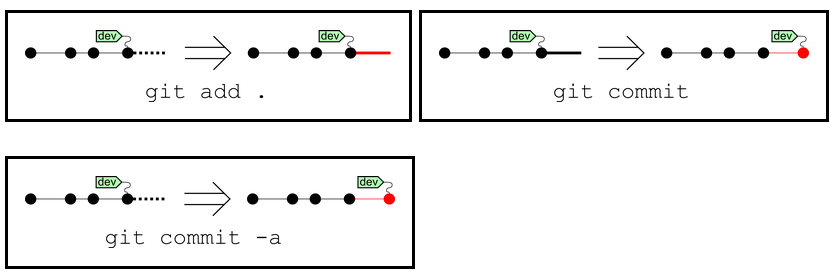
Setting up a repository (add >> into index)



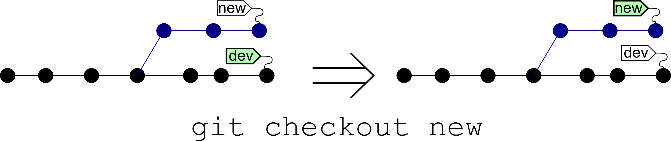
Cloning a repository



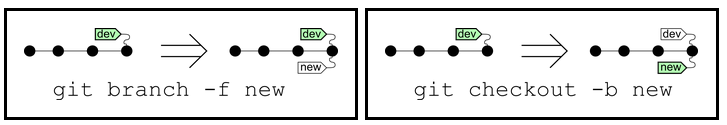
Making changes



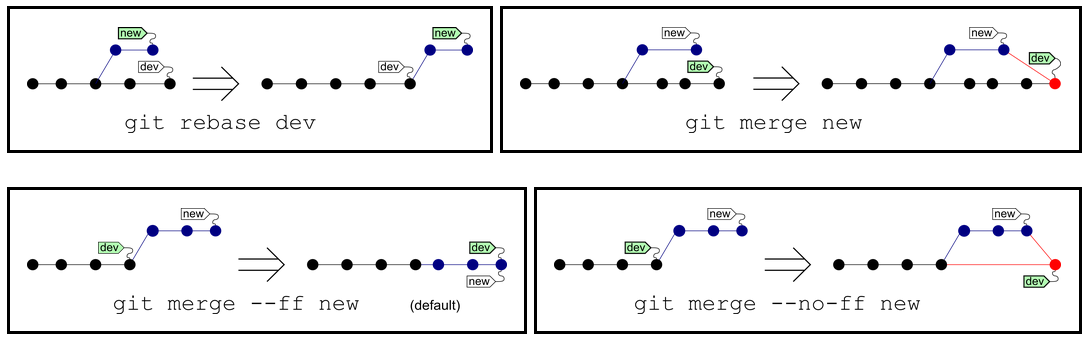
Change focus point to another branch



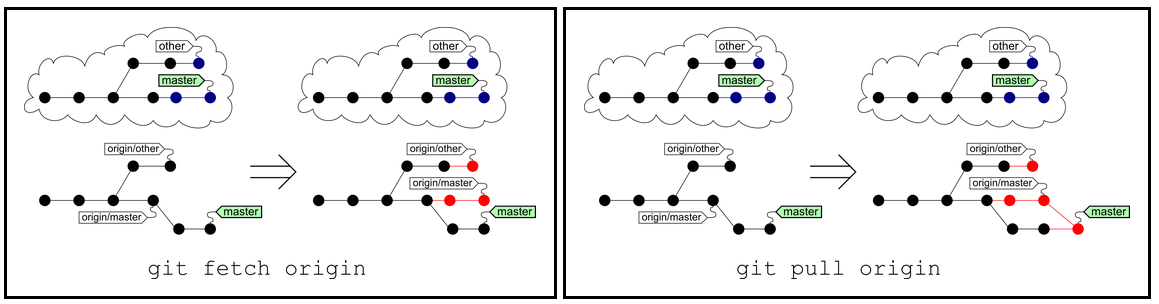
Making a new branch



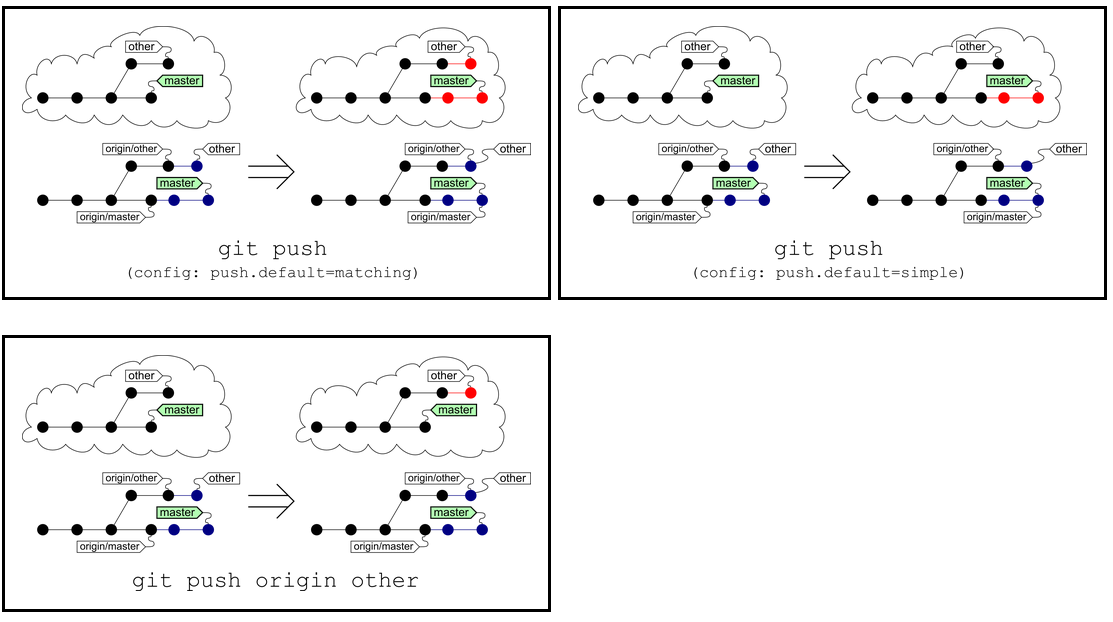
Rebase vs merge (rebase: reapply the current branch on another branch; merge = combine 2 and make a new pcommit).

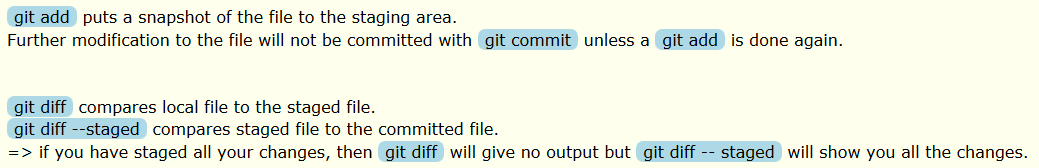
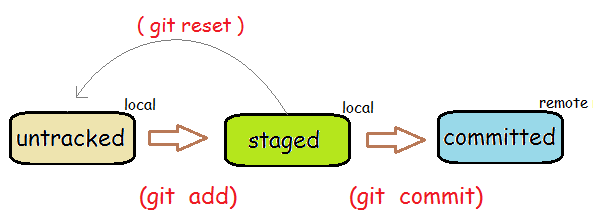


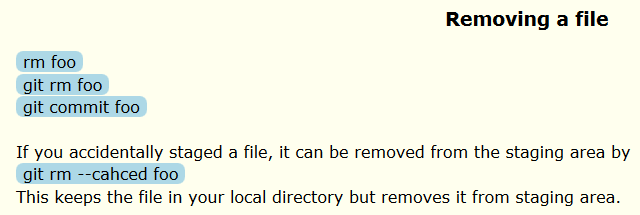
Remote repo – fetch vs pull (fetch: update origin without affecting the current working; pull: fetch and then merge)

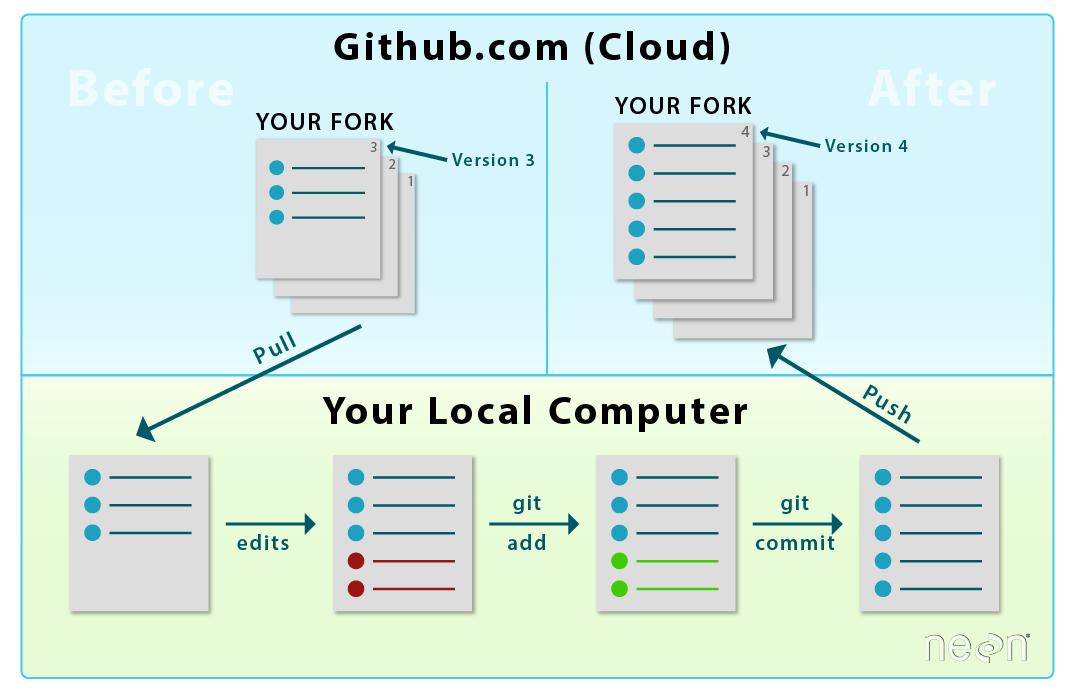


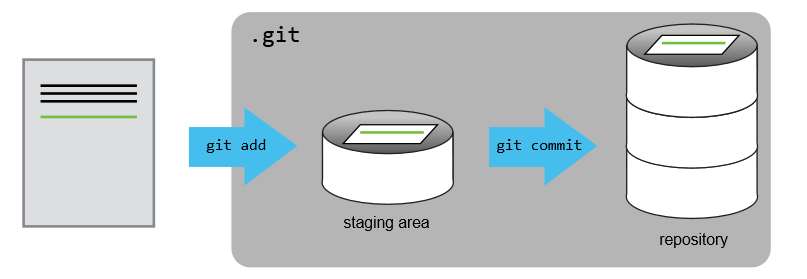
Remote repo: push





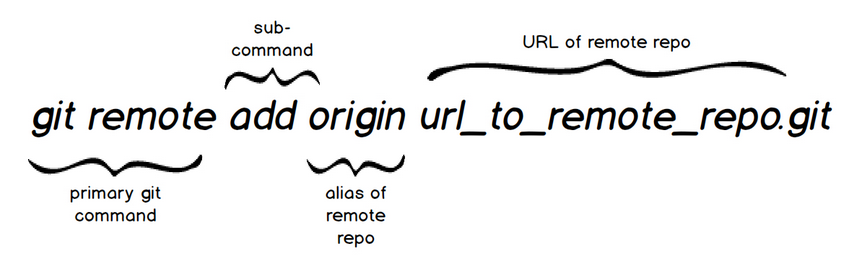


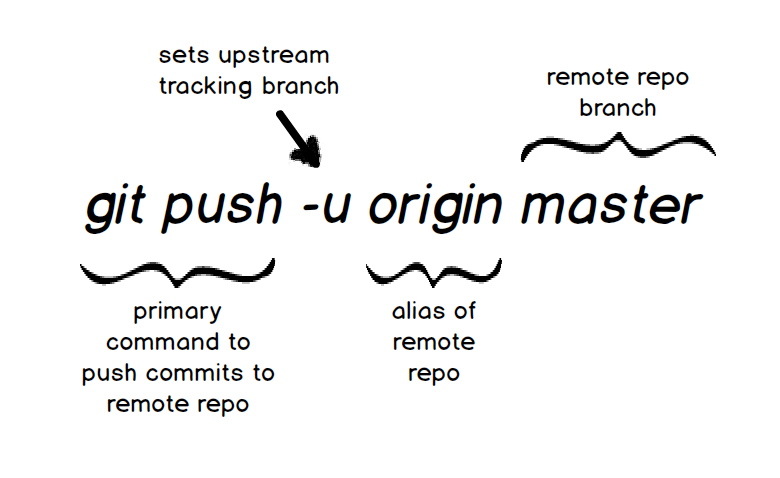




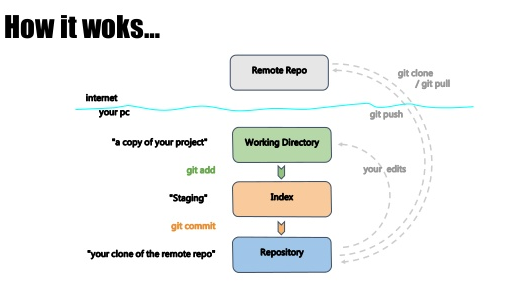
You can think of Git as taking snapshots of changes over the life of a project. git add specifies what will go in a snapshot (putting things in the staging area), and git commit then actually takes the snapshot and makes a permanent record of it (as a commit).

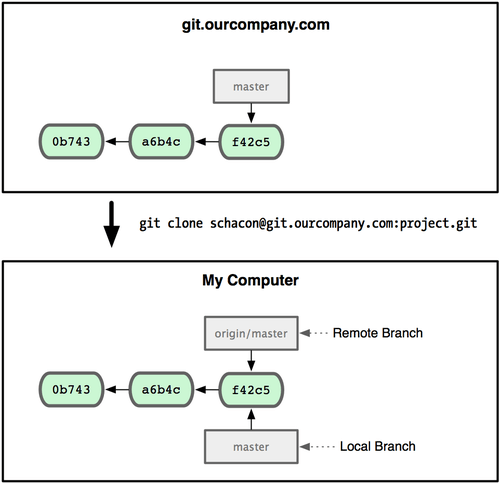
Use git push origin gh-pages. tells Git to push the files to the originating repo which in this case - is our fork on github.com which we originally cloned to our local computer. gh-pages is the repo branch that you are currently working on.

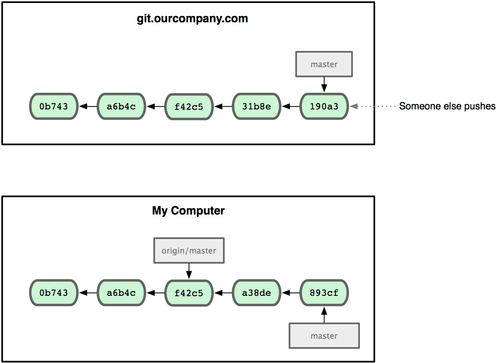


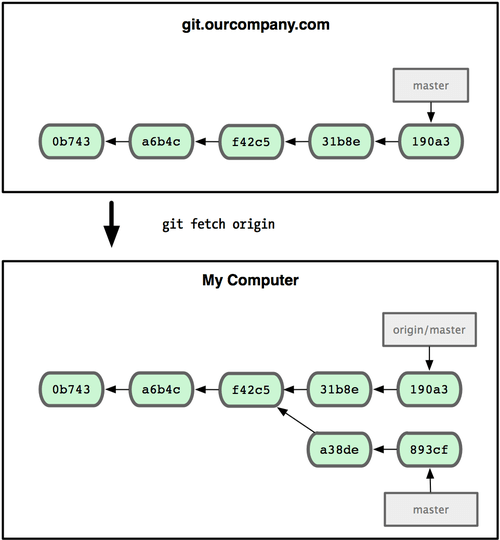


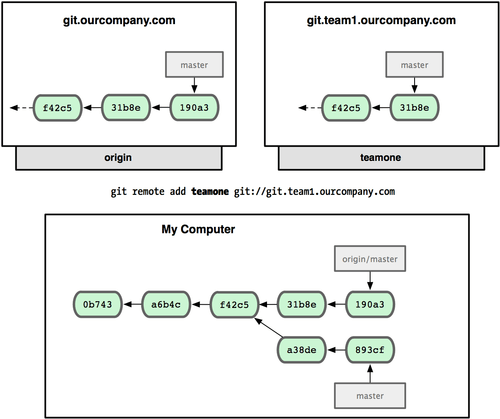
<http://www.slideshare.net/OkbaDex/git-37409341>

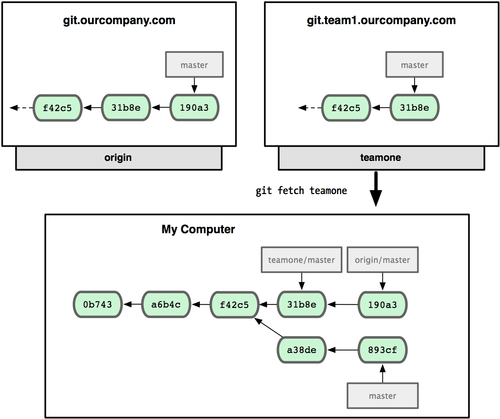












<https://git-scm.com/book/id/v1/Branching-Pada-Git-Remote-Branches>

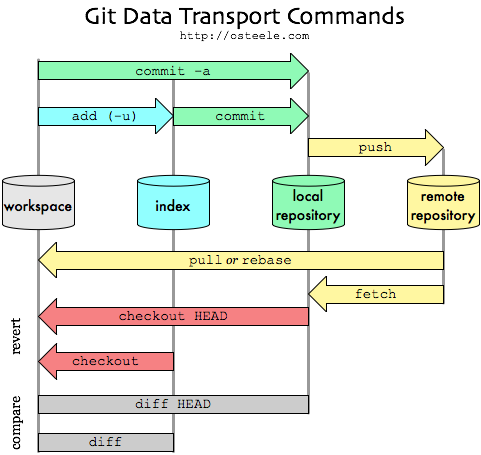
In the simplest terms, git pull does a git fetch followed by a git merge.

This operation never changes any of your own local branches under refs/heads, and is safe to do without changing your working copy.

A git pull is what you would do to bring a local branch up-to-date with its remote version, while also updating your other remote-tracking branches.

git pull will always merge into the **current branch**. So you select which branch you want to pull *from*, and it pulls it into the current branch. The *from* branch can be local or remote;

* When you use pull, Git tries to automatically do your work for you. **It is context sensitive**, so Git will merge any pulled commits into the branch you are currently working in. pull **automatically merges the commits without letting you review them first**. If you don’t closely manage your branches, you may run into frequent conflicts.
* When you fetch, Git gathers any commits from the target branch that do not exist in your current branch and **stores them in your local repository**. However, **it does not merge them with your current branch**. This is particularly useful if you need to keep your repository up to date, but are working on something that might break if you update your files. To integrate the commits into your master branch, you use merge.
* **git fetch** is the command that says "bring my local copy of the remote repository up to date."
* **git pull** says "bring the changes in the remote repository where I keep my own code."
* Normally "**git pull**" does this by doing a "**git fetch**" to bring the local copy of the remote repository up to date, and then merging the changes into your own code repository and possibly your working copy.
* The take away is to keep in mind that there are often at least **three copies** of a project on your workstation. One copy is your own repository with your own commit history. The second copy is your working copy where you are editing and building. The third copy is your local "cached" copy of a remote repository.



When you clone a repository you fetch the entire repository to you local host. This means that at that time you have an origin/master pointer to HEAD and master pointing to the same HEAD.

when you start working and do commits you advance the master pointer to HEAD + your commits. But the origin/master pointer is still pointing to what it was when you cloned.

So the difference will be:

* If you do a git fetch it will just fetch all the changes in the remote repository ([GitHub](http://en.wikipedia.org/wiki/GitHub)) and move the origin/master pointer to HEAD. Meanwhile your local branch master will keep pointing to where it has.
* If you do a git pull, it will do basically fetch (as explained previously) and merge any new changes to your master branch and move the pointer to HEAD.
* **Briefly**
* git fetch is similar to pull but doesn't merge. i.e. it fetches remote updates (refs and objects) but your local stays the same (i.e. origin/master gets updated but master stays the same) .
* git pull pulls down from a remote and instantly merges.
* **More**
* git clone clones a repo.
* git rebase saves stuff from your current branch that isn't in the upstream branch to a temporary area. Your branch is now the same as before you started your changes. So, git pull -rebase will pull down the remote changes, rewind your local branch, replay your changes over the top of your current branch one by one until you're up-to-date.
* Also, git branch -a will show you exactly what’s going on with all your branches - local and remote.

I thought I'd update this to show how you'd actually use this in practice.

1. Update your local repo from the remote (but don't merge):

git fetch

1. After downloading the updates, let's see the differences:

git diff master origin/master

1. If you're happy with those updates, then merge:

git pull

On step 2: For more on diffs between local and remotes, see: [compare local git branch with remote branch?](http://stackoverflow.com/questions/1800783/compare-local-git-branch-with-remote-branch)

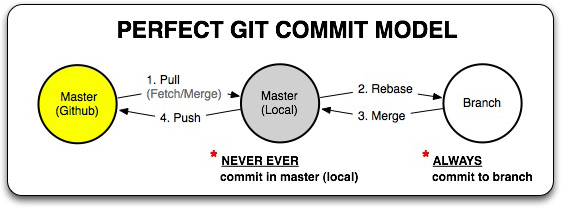
On step 3: It's probably more accurate (e.g. on a fast changing repo) to do a git rebase origin here. See @Justin Ohms comment in another answer.

enter image description here

To understand this, you first need to understand that your local git maintains not only your local repository, but it also maintains a local copy of the remote repository.

git fetch brings your local copy of the remote repository up to date. For example, if your remote repository is GitHub - you may want to fetch any changes made in the remote repository to your local copy of it the remote repository. This will allow you to perform operations such as compare or merge.

git pull on the other hand will bring down the changes in the remote repository to where you keep your own code. Typically, git pull will do a git fetch first to bring the local copy of the remote repository up to date, and then it will merge the changes into your own code repository and possibly your working copy.



<https://duzun.me/tips/git>

<http://rogerdudler.github.io/git-guide/>

# Terminologies

Repository: the whole history (multiple revisions) of the project.

Checkout: extract a revision (usually the latest) from the repository

* Switch to / move focus to a branch within the repository
* Git checkout -n >> create a new branch

Checkin / Commit: create a new revision from the current working copy.

Branch: a variant of the same software (~ working copy but is linked to the master branch and can have multiple branches!).

Init: create a repository

Add: add a file into the index (not yet commit to master)

Index: master repository >> index >> working copy.

Merge: merge changes in one branch to another.

**Remote server**

Clone: copy the entire repository into a machine.

Push: “commit” changes from local repository to project repository, making two identical repositories!

Pull: get the current copy of the project and attach to the current working branch. (git pull = git fetch + git merge)

Fetch: updates the local mirror of the remote repository.

Untracked file: These are all new files that have never been added to or tracked by Git.

Use git status anytime to view any untracked changes that have occurred, what is being tracked and what is not currently being tracked.

git remote add <remote alias> <URL> : add an alias to associate with the remote original repo (the one we fork from). This remote we can pull the updates but will not be able to push any updates!

# Process

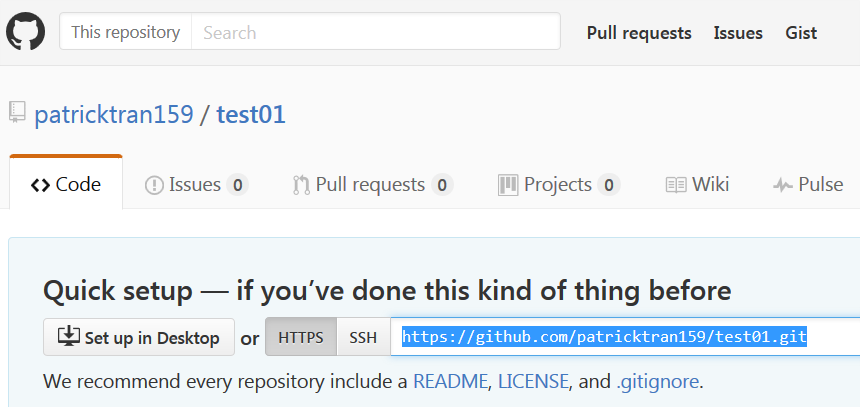
## Make a work with a local repository

1. Check git version and update if necessary
   1. git --version
   2. update git to the latest version

|  |
| --- |
| sudo apt-add-repository ppa:git-core/ppa  sudo apt-get update  sudo apt-get install git |

* 1. git --version

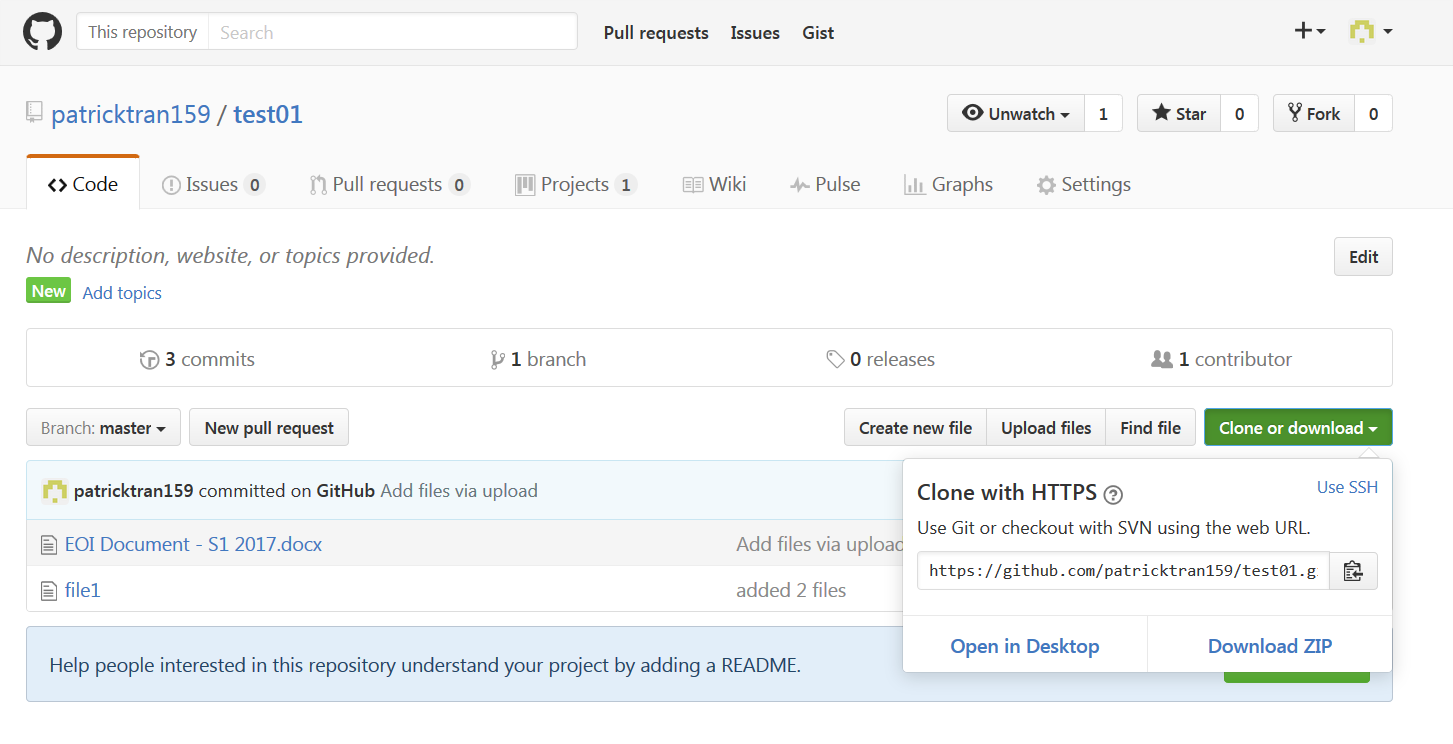
1. Set your name and email
   1. git config --global user.name “…”
   2. git config --global user.email “…”
2. Create a local repository
   1. Make a new folder: mkdir <foldername>
   2. Go inside the folder: cd <foldername>
   3. List all the items in the folder: ls / dir
   4. Make the folder a git repository
3. Edit the repository …
4. Commit changes to the repository
   1. Check git status: git status
   2. Add a file to the index to be committed: git add <Filename>
   3. Add all files changes to the index: git add .
   4. Add all files changes including removed files: git add -all .
   5. To commit the changes added: git commit -m “message here”
   6. To replace the last commit: git commit --amend
   7. Check again, make sure no untracked files: git status
5. Display all commits so far
   1. git list
6. Remote control
   1. Add your GitHub account: git config --global user.name >Username/Email>
   2. Add remote connection: git remote add <Name that you want to call remote> <URL>



* 1. To list all active remotes: git remote -v
  2. Pull in changes: git pull <remote name> <remote branch>
     1. git pull remote01 master
  3. Push changes: git push <remote name> <remote branch>

## Work with your remote repository (not someone else)

1. To clone a repository:
   1. git clone <URL>
   2. This way, the cloned repository (flder in local machine) has the following branches attached:
      1. Remotes/origin/HEAD
      2. Remotes/origin/master



1. Create a new branch
   1. List the branches: git branch
   2. Create a new branch and switch to the branch in one line: git checkout -n <branch name>
   3. Move onto a branch: git checkout <branch name>
   4. Create a new branch without moving onto it: git branch <branch name>
   5. List all branches including remote ones: git branch -a
   6. Push this new branch to the remote master. Go inside the local new branch, git push origin <local new branch>
2. Check any changes to the remote before you pull in: git fetch --dry-run
   1. Make sure you are in the branch you want to update, pull any changes from a remote branch: git pull <remoteName> <RemoteBranch>

## Update a forked repository

In your local clone of your forked repository, you can add the original GitHub repository as a "remote". ("Remotes" are like nicknames for the URLs of repositories - origin is one, for example.) Then you can fetch all the branches from that upstream repository, and rebase your work to continue working on the upstream version. In terms of commands that might look like:

# Add the remote, call it "upstream":

git remote add upstream https://github.com/whoever/whatever.git

# Fetch all the branches of that remote into remote-tracking branches,

# such as upstream/master:

git fetch upstream

# Make sure that you're on your master branch:

git checkout master

# Rewrite your master branch so that any commits of yours that

# aren't already in upstream/master are replayed on top of that

# other branch:

git rebase upstream/master

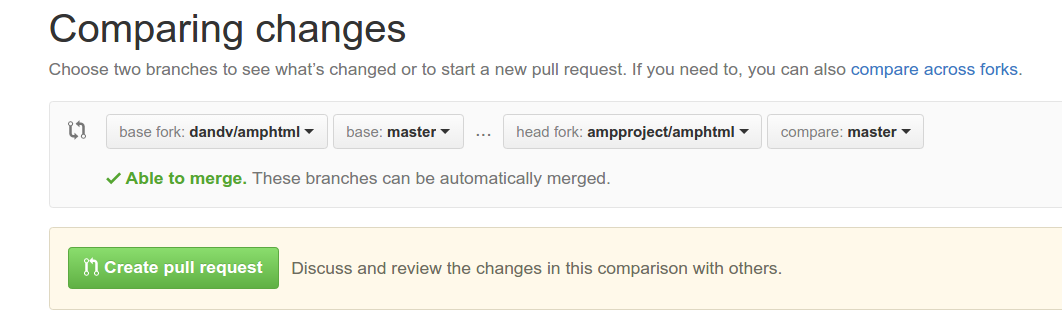
If you don't want to rewrite the history of your master branch, (for example because other people may have cloned it) then you should replace the last command with git merge upstream/master. However, for making further pull requests that are as clean as possible, it's probably better to rebase.

If you've rebased your branch onto upstream/master you may need to force the push in order to push it to your own forked repository on GitHub. You'd do that with:

git push -f origin master

You only need to use the -f the first time after you've rebased.

Starting in May 2014, it is possible to update a fork directly from GitHub. This still works as of October 2016, **BUT** it will lead to a dirty commit history.

1. Open your fork on GitHub.
2. Click on Pull Requests.
3. Click on New Pull Request. By default, GitHub will compare the original with your fork, and there shouldn't be anything to compare if you didn't make any changes.
4. Click switching the base if you see that link. Otherwise, manually set the base fork drop down to your fork, and the head fork to the upstream. Now GitHub will compare your fork with the original, and you should see all the latest changes. [](https://i.stack.imgur.com/FoNQu.png)
5. Create pull request and assign a predictable name to your pull request (e.g., Update from original).
6. Scroll down to Merge pull request, but don't click anything yet.

Now you have three options, but each will lead to a less-than-clean commit history.

1. The default will create an ugly merge commit.
2. If you click the dropdown and choose "Squash and merge", all intervening commits will be squashed into one. This is most often something you don't want.
3. If you click Rebase and merge, all commits will be made "with" you, the original PRs will link to your PR, and GitHub will display This branch is X commits ahead, Y commits behind <original fork>.

So yes, you can keep your repo updated with its upstream using the GitHub web UI, but doing so will sully your commit history. Stick to [the command line](http://stackoverflow.com/questions/7244321/how-do-i-update-a-github-forked-repository/7244456#7244456) instead - it's easy.

## To delete a branch locally and remotely

1. List all branches: git branch --all
2. Delete local branch: git branch -d dev01
3. Delete remote branch: git push origin :dev01

Executive Summary

$ git push origin --delete <branch\_name>

$ git branch -d <branch\_name>

Delete Local Branch

To delete the local branch use:

$ git branch -d branch\_name

**Note:** The -d option is an alias for --delete, which only deletes the branch if it has already been fully merged in its upstream branch. You could also use -D, which is an alias for --delete --force, which deletes the branch "irrespective of its merged status." [Source: man git-branch]

Delete Remote Branch [Updated on 1-Feb-2012]

As of [Git v1.7.0](https://github.com/gitster/git/blob/master/Documentation/RelNotes/1.7.0.txt), you can delete a **remote** branch using

$ git push origin --delete <branch\_name>

which might be easier to remember than

$ git push origin :<branch\_name>

which was added in [Git v1.5.0](https://github.com/gitster/git/blob/master/Documentation/RelNotes/1.5.0.txt) "to delete a remote branch or a tag."

* 1. Push a branch to the remote: you are in branch dev01 locally (apart from your local master) and the remote01 has only master. You need to create one branch in the remote before pushing dev01. If you don’t do this, and try: git push remote01 master, you will have the error: The current branch dev01 has no upstream branch.
     1. Use “-u“ in the git push command: git push -u origin master. (This creates (-u) another branch in your remote repo.)