# Git glossary

Repo (repository): a [Git repository](http://bitbucket.org/code-repository) is a virtual storage of your project.

Head: the pointer to the current commit of the current branch; so, there is only one head. Note that the current commit is not necessarily the latest commit like in the case Detached head state.

Detached head state: the state when the head is pointing at some commit that is not the latest commit of a branch. From this commit, if one makes a new commit then this new commit cannot be inserted to any existing branch; otherwise it will break the branch and hence make the whole history inconsistent. So if you want to retain your new commit, there is no way but creating a new branch for the new commit.

Stage: git add

Indexed = staged

Tracked files & untracked files: tracked files are files that were in the last snapshot;

Changeset vs snapshot: TFS saves changeset, i.e. only the change between two versions, while Git saves snapshot, i.e. the whole file.

Upstream branch = tracking remote branch

Tracking branch: Tracking branches are local branches that have a direct relationship to a remote branch. When the tracking branch (local) and tracked branch (remote) are paired, git keeps track of the difference between the two and can tell users this.

If you’re on a tracking branch and type git pull, Git automatically knows which server to fetch from and which branch to merge in.

When you clone a repository, it generally automatically creates a master branch that tracks origin/master. However, you can set up other tracking branches if you wish — ones that track branches on other remotes, or don’t track the master branch.

ORIGIN (default name for a remote repos) and MASTER (default name for a local branch):

* “origin” is the default name for a **remote** **REPOS** when you run git clone
* “master” is the default name for a starting **local** **BRANCH** when you run git init

Exp: the URL parameter to the "clone" command becomes the "origin" for the cloned local repository:

git clone <https://github.com/gittower/git-crash-course.git>

# Git command

## Initialization (git init, git clone)

* git init  
  To create a new repo, you'll use the git init command. git init is a one-time command you use during the initial setup of a new repo. Executing this command will create a new .git subdirectory in your current working directory. This will also create a new master branch.
* git clone

## With branch

* recall: the default name for the starting **branch** of the local repo is Master, the default name for a remote **repos** is Origin.
* list all branches on the local and remote repo:

git branch -a  
Note that if you haven’t created/modified any branch, no branch is shown even the default branch Master.

* create a new local branch  
  git branch <new\_branch\_name>

To create a new remote branch, create a local one first and then push it

* delete a local branch:

git branch -d <branch\_name>

delete a remote branch:

git push <remote\_repos> -d <branch\_name>

* rename the current local branch to another one: git branch -m “new\_name”

rename a remote branch: rename the corresponding local branch, push it and delete the old branch on the remote repos.

* switch current local branch to another one:

git checkout <branch\_name>

* create a new branch and then switch to it:

git checkout -b <branch\_name>

* Set a local branch to track a remote branch
  + General case:

git branch -u <remote\_repos>/<remote\_branch> [local\_branch]  
(If local\_branch is dropped, the current HEAD branch is used)

* + Specific case 1: when pushing a local branch to a remote, one can set up the upstream by -u flag

pushing the current Head branch to a remote with new name “remote\_branch”  
git push -u <remote\_repos> <remote\_branch>

pushing “local\_branch” to a remote\_repo with new name “remote\_branch”

git push -u <remote\_repos> <loca\_branch>:<remote\_branch>

* + Specific case 2: when switching to a local branch, one can set up a tracking branch by --track flag

git checkout --track <remote\_repo>/<remote\_branch>

## with head

* Check out a commit determined by a SHA (move the Head to a commit)

git checkout <SHA>

After this command, the state will be a “detached head”, i.e. commit after this step will belong to no branch, and hence easily lost.

## push pull fetch

* push a local branch to a remote branch

git push <remote> <local\_branch>:<remote’s branch>

* git fetch: get the information about the remote repos so that when you “git status” it shows you the difference between your local repo and the remote repo
* git pull is actually git fetch + git merge

## git merge

* merge a branch with the current local branch (and then you have 2 branches)

git merge <branch\_name>

git merge <remoteRepos/remotebranch>

* There are 2 scenarios for merging anotherBranch to currentBranch
  + Git can merge the 2 branches automatically:
    - Fast-forward: anotherBranch is just ahead of currentBranch a number of commits 🡪 git move the currentBranch’s head to match the anotherBranch’s head
    - Merge commit: GIT can automatically create a snapshot that contain the heads of both GanotherBranch and currentBranch
  + Conflict that your have to resolve manually. This happens when there are files that both anotherBranch and currenBranch update. Git temporarily pauses the merging process and let you:
    - either abort the merging by: git merge - - abort
    - or manually edit the files causing conflict and then add and commit.
      * Edit the files causing conflict on the currentBranch. The content of those files will be temporarily changed to include the comparison of the difference of the two branches.
      * git add <names\_of\_those\_files>
      * git commit
      * Note that when you use “git status” it will show you the guide.

## git add & git rm, git mv (not just rm, mv)

* git add [-A|-u] <path\_to\_files>: stage files in the <path\_to\_files>. The <path\_to\_files> can be dot . meaning current directory
  + -A = -- all: all (new, deleted, modified) files
  + -u = -- update: updated (deleted, modified) files that are being tracked by git

git add -u . : stage all the updated files that are tracked by git in the current working directory

* rm, mv (without git) works in git but the change caused by those commands are not staged
* git rm = rm and add
* git mv = mv and add
* ls <folder> lists tracked files of the current branch and also untracked files (in the <folder> of the working directory)

## With remote repos

* adding Remote Repositories  
  git remote add <remote\_alias> <remote URL>
* show remote repos configured

git remote or git remote -v

* show more info about a remote repo:

git remote show <remote repos>

## Undo

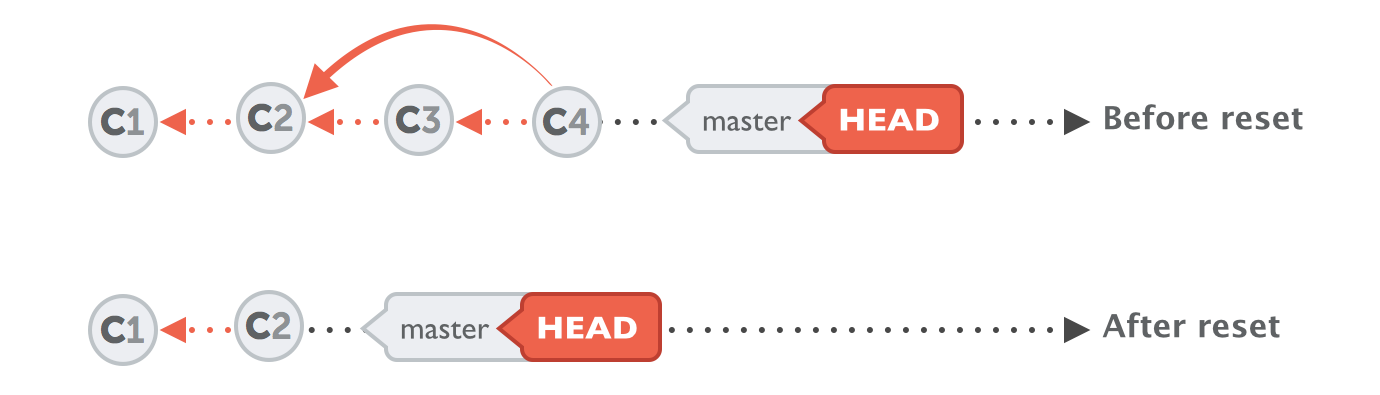
* Fixing the last commit
  + Amend only the message of the LAST commit

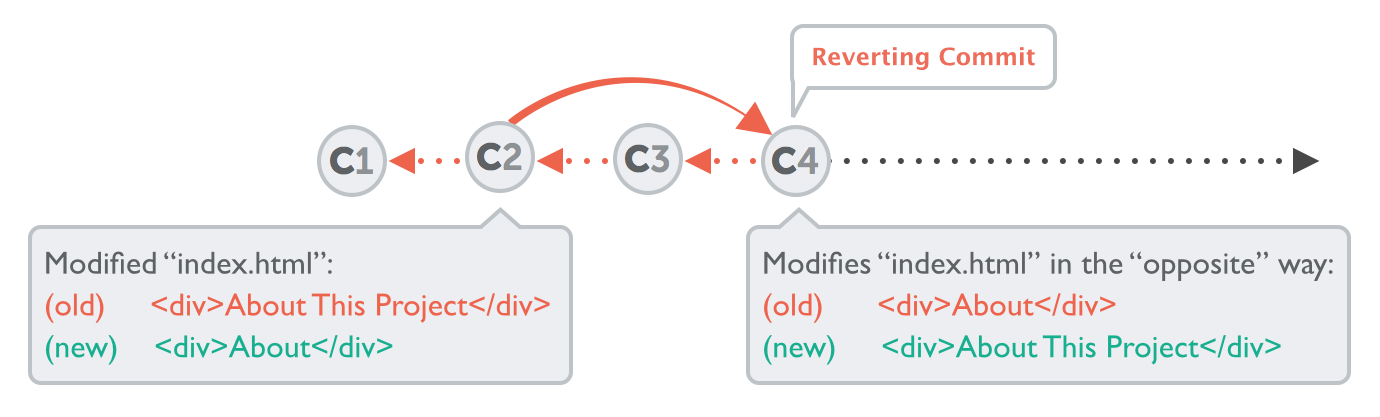
$ git commit --amend -m "This is the correct message"

* + If you want to add some more changes to that LAST commit, you can simply stage them as normal and then commit again:

$ git add some/changed/file.ext

$ git commit --amend -m "commit message"

* Undoing local changes
  + “Local” changes (a term of GIT) are changes of TRACKED files (so not necessarily staged) in the working directory
  + To discard all local changes of a specific file = To restore a specific file to its latest committed version, i.e. the current HEAD, one uses:   
    git checkout HEAD path/to/file  
    as in “git checkout <SHA>”. “HEAD” stands for the SHA of the last commit so “git checkout HEAD” is similar to “git checkout <SHA>”. One cannot drop the “path/to/file” to restore ALL the files; to do this, one uses “git reset” as below.
  + To discard all local changes of the working directory and restore to the latest committed version, i.e. the current HEAD, use git reset: git reset --hard HEAD
* Undoing committed changes
  + Resetting your current HEAD branch to an older revision (also called "rolling back" to that older revision): **$ git reset --hard <SHA>** After this command, your currently checked out branch will be at revision determined by <SHA>. The commits that came after this one are effectively undone and are no longer visible in the history of this branch. Note that calling the command with the "--hard" option will discard all local changes that you might currently have.   
    Exp: your last commit is at C4 and you want to go back to C2, git reset --hard <C2sha> will move the current HEAD to C2; C3, C4 are not deleted but not visible to you. (If you want you can use C3, C4 but this is a very advanced task) 
  + Undo a certain commit but keep the rest. Exp: you have C1, C2, C3 commits and you want to undo C2 while keep C3, then use: **git revert <sha>** This will create a new commit, called C4, which reverts the effects of a C2 commit, effectively undoing it. In this exp, C2 replaces “About this Project” by “About”, so C4, which reverts C2, will replace “About” by “About this Project”.



## Detached head mode

* To eliminate commits and go back to a specific snapshot, use git reset --hard <SHA>.
* To go back to a snapshot but not undo anything, use git checkout <SHA>. After this step, the status now is “detached head” state, if you create a new commit then this commit cannot be inserted to any existing branch so if you want to retain this new commit you have to create a new branch admitting the new commit as its head.
  + After creating a new commit, create a new branch admitting the new commit as its branch and then switch to that branch: git switch -c <newBranch>. To undo this: git switch -  
    Or just use git checkout -b <newBranch> as usual.
  + Jump to another branch, abandoning the newly created commit (if there has been): git checkout <branch>

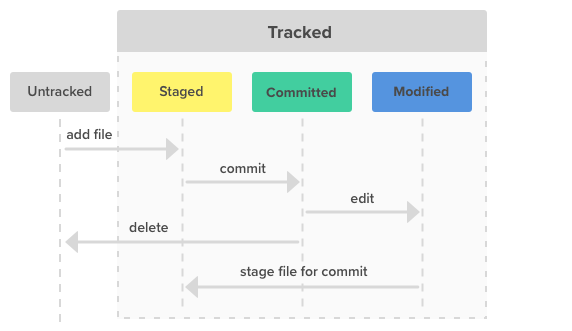
# Git concepts

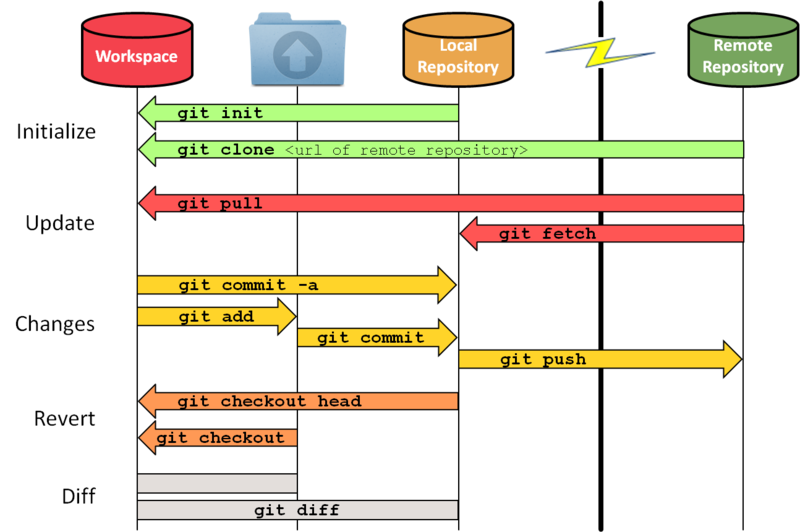
## Tracked and Untracked files

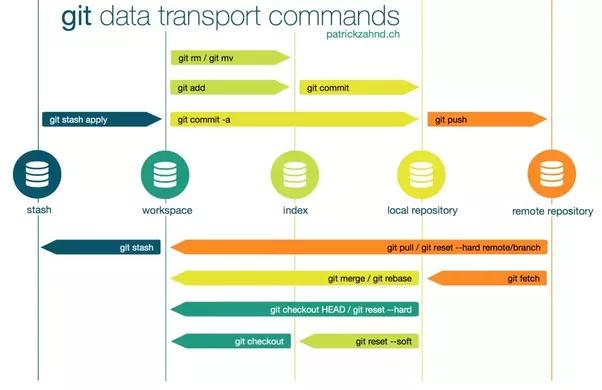
Remember that each file in your working directory can be in one of two states: **tracked** or **untracked**. Tracked files are files that were in the last snapshot; they can be unmodified (committed), modified, or staged. In short, tracked files are files that Git knows about.

Untracked files are everything else — any files in your working directory that were not in your last snapshot and are not in your staging area. When you first clone a repository, all of your files will be tracked and unmodified because Git just checked them out and you haven’t edited anything.

As you edit files, Git sees them as modified, because you’ve changed them since your last commit. As you work, you selectively stage these modified files and then commit all those staged changes, and the cycle repeats.







## What are there in a working directory (what is shown)?

* A GIT repos is more than a single directory because it contains many versions of a directory.
* What is shown in the GIT repos directory is the version that GIT’s HEAD pointing to. This is called a working tree.
* In any directory of the working tree, there are both tracked files and untracked files. Tracked files are files that appear in the current snapshot or staged (by “git add”); untracked files are the rest.

## GIT does not have a command to create directly a remote branch

Your local computer is a local repo and your GitHub account is a remote repo.

Git on your local computer does not allow you to create directly a branch on your GitHub repo.

But Either you can use GitHub to create branches on your GitHub repo OR you can create a local branch and push it the remote repo.

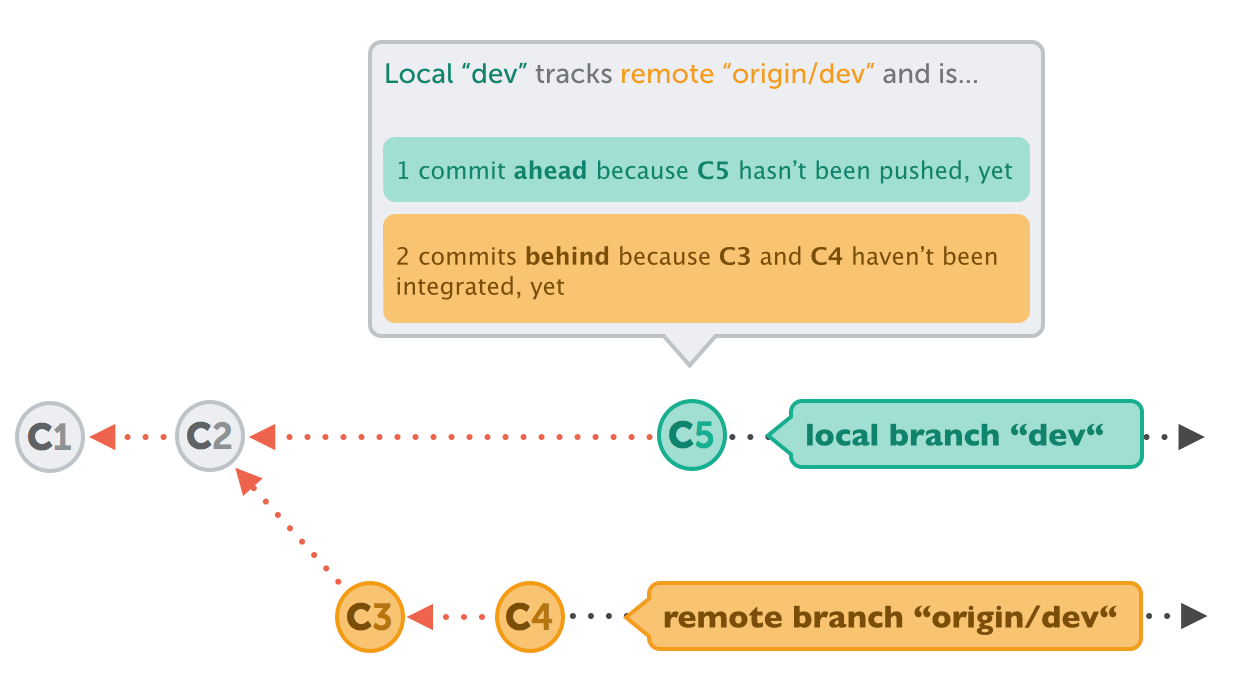
The reason for Git’s not having creating remote branch command is Git does not have .git folder of the remote repo, which is needed for creating a remote branch.

Another reason may be this job should be left for GitHub, which is the local Git of the remote repos.

## A local branch tracks a remote branch

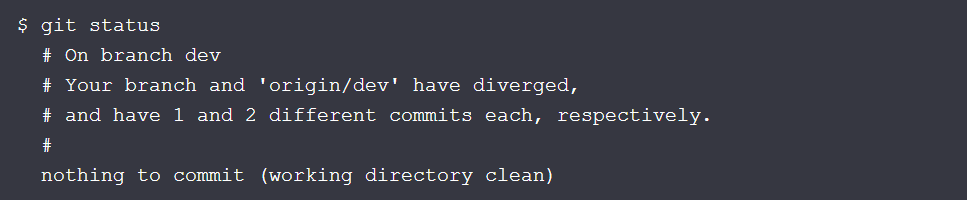
A local branch can track a remote branch so that whenever the two branches are not in sync, git can inform the local branch about this (Note this requires git fetch).

Exp: local “dev” tracks remote “origin/dev”



git status will tell how the two branches diverge. The local has 1 difference (C5) and the “origin/dev” has 2 differences (C3, C4).

Note one needs “git fetch” before “git status” shows the comparison of the two branches



## Detached Head State and impossibility to insert a new commit to an existing branch

Recall that a commit always points to the preceding commit. A commit is merely a set of new files added compared to the preceding commit. This pointing must be maintained otherwise the history is not consistent.

Imagine you go back to a commit but do not undo any commit after that commit; so you are not at the HEAD (the latest commit) of the branch. At this state, which is called detached head state, if you create a new commit then what existing branch can this new commit be inserted to? No existing branch! Because you can only insert a new commit to the HEAD of an existing branch; inserting a new commit at other position will break the branch. Otherwise, to maintain the consistency of existing branches, you have to create a new branch. So at this detached head state, if you make a new commit, the only way to retain this commit is to create a new branch and this new commit becomes the new branch’s head.

## Merging: fast-forward, merge commit, merge conflict

Consider this example: You are at C2 commit – “master” branch, and your customer finds a bug. To satisfy customer’s demand, you make an ad-hoc patch – “hotfix”, and then you work on the bug at a deeper level – “iss53” branch.



### Fast-forward merge

Merging “hotfix” with “master” 🡪 the head of master is moved to the head of hotfix (so 2 branches have heads pointing to a same commit. Note that you still have 2 branches after merging and you can delete “hotfix”)

$ git checkout master

$ git merge hotfix

Updating f42c576..3a0874c

Fast-forward

index.html | 2 ++

1 file changed, 2 insertions(+)



### Merge commit



Merging “iss53” with “master”: This cannot be done fast-forward. Git will try to create a new snapshot that contains the latest commits of the both branches. Note that “master” branch will move to the new snapshot while “iss53” remains in its original snapshot.

The newly created snapshot, C6, is called merge commit.



### Merge conflict

When git cannot automatically create a merge commit, there is a conflict that git let you manually resolve.

Consider example of merging “hotfix” with “iss53”: C3, C4 are two different versions of C2.

$ git merge iss53

Auto-merging index.html

CONFLICT (content): Merge conflict in index.html

Automatic merge failed; fix conflicts and then commit the result.

When there is a conflict, GIT pauses and let you choose: abort this merging or manually resolve the conflict and then git will finish the process.

1. Abort this merging: $ git merge –abort
2. Manually resolve the conflict
   1. To see what will be in the merged snapshot, what causes conflicts:

* $ git status
* On branch master
* You have unmerged paths.
* (fix conflicts and run "git commit")
* Changes to be committed:
* new file: readme.txt
* Unmerged paths:
* (use "git add <file>..." to mark resolution)
* both modified: index.html
* no changes added to commit (use "git add" and/or "git commit -a")

The files listed in “Changes to be committed” will be in the resulting snapshot of merging. Those files don’t cause any conflict, and git knows what to do with them.

The files listed in “Unmerged paths” need to be resolved manually.

* 1. GIT temporarily modify the files causing conflict

The content of the file causing conflict is temporarily changed; it includes the comparison of the files of the two branches. If you cat or vim it, you will be surprised:

Exp: File index.html in “master” is only

<--! The footer of webpage

<div id="footer">contact : email.support@github.com</div>

But now if you cat or vim it, the content is:

<--! The footer of webpage

<<<<<<< HEAD:index.html

<div id="footer">contact : email.support@github.com</div>

=======

<div id="footer">

please contact us at support@github.com

</div>

>>>>>>> iss53:index.html

At the line that the 2 branches differ, the two versions are included; they are separated by ==========

<<<<<< marks the file of the current branch, >>>>>>> marks the other file.

* 1. You edit the file causing conflict

In this example, you edit file index.html on the “master” branch

* 1. You add and then commit as guided (in git status) to finish merging

Recall that “git status” shows you how to finish merging

use "git add <file>..." to mark resolution

fix conflicts and run "git commit"

After editing the file, you use **git add index.html** to mark the resolution and then **git commit**

(note that the response to those commands are not as usual. Remember that git pauses and let us resolve so that git can be back to merging)

