# Most commonly used GIT commands

<https://www3.ntu.edu.sg/home/ehchua/programming/howto/Git_HowTo.html>

## Config Git

Install GIT and type GIT commands in "Git Bash" command shell (Windows) or "Terminal" (Mac/Ubuntu).

Set up username and email (used in commits):

For a specific repository: Enter the repository directory and then

// Set up your username and email (to be used in labeling your commits)

$ **git config user.name "*your-name*"**

$ **git config user.email "*your-email@youremail.com*"**

For all repository: at any directory still use git config with --global option

// Set up your username and email (to be used in labeling your commits)

$ **git config --global user.name "*your-name*"**

$ **git config --global user.email "*your-email@youremail.com*"**

The settings are kept in "<GIT\_HOME>/etc/gitconfig" (of the GIT installed directory) and "<USER\_HOME>/.gitconfig" (of the user's home directory.  
You can issue "git config --list" to list the settings:

$ **git config --list**

user.email=xxxxxx@xxxxxx.com

user.name=xxxxxx

## Set up local repo: git init or git clone

Initialize an empty local repo: enter the root folder of a project, use git init.

A hidden folder named “.git” will be created to store the files of all snapshots of the repos.

// Change directory to the project directory

$ **cd /*path-to*/hello-git**

// Initialize Git repo for this project

$ **git init**

Initialized empty Git repository in /*path-to*/hello-git/.git/

Clone a remote repo to the local machine:

// Clone the remote repo UNDER the current directory

$ **git clone <remote-url>**

// Clone the remote repo UNDER **<working-directory-name>**

$ **git clone <remote-url> <working-directory-name>**

## Stage, Status and Commit, Log

### Show the status of files (untracked, staged, unstaged) in the working directory:

Files that are not in the last snapshot are UNTRACKED; otherwise, tracked. (Due to this, if a git folder is just initialized and no commit is done yet then all files in the git folder are untracked)

For a tracked file, if its change is not staged, the change (not the file) is called UNSTAGED; otherwise, STAGED.

3 possible status of file changes: “Untracked files”, “Changes not staged for commit”, “Changes to be committed”

**$ git status**

On branch master

**Changes to be committed**:

(use "git restore --staged <file>..." to unstage)

modified: fileA.txt

**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: fileB.txt

**Untracked files**:

(use "git add <file>..." to include in what will be committed)

fileC.txt

### Stage files

Editing (Create, Modify, Rename, Delete) files produces "unstaged" file changes. Stageing files produces "Staged" status.

To stage: use “git add”.

To UNstage: use “git restore --staged" (from git 2.23.0 onward) or “git rm –cached” (older git)

“git rm”, “git mv” are combination of file command and git. It means remove/move file and stage this remove/move. In other words, it remove/move the file both in the working directory and the stage area.

// Stage content change of file

$ **git add <file>** // stage file

// Stage path change of file

$ **git rm <file>** // remove a file in both working dir and stage area

$ **git mv <old-file-name> <new-file-name>** // rename/move a file in both working dir and stage area

// The <file> in git rm, git mv must be staged

// UnStage file

$ **git restore --staged <file>** // from git 2.23.0

$ **git rm --cached <file>** // old git version

// Stage ALL files with changes

$ **git add -A** // OR, '**git add --all**'

// Stage all the files in the current directory (and all sub-directories)

$ **git add .** // the option is dot “.”

### Git does NOT care about folders (untracked, unstaged, staged)

Git doesn’t care about folders, but files. When you create a new empty folder, it’s not shown as “untracked”; afterwards, if you add files into it, the file, not the folder will be shown as “untracked”.

### Commit staged files: git commit -m “description”

// Commit the files that are staged (-m to specify the commit description)

$ **git commit -m "First commit"**

[master (root-commit) 858f3e7] first commit

2 files changed, 8 insertions(+)

create mode 100644 Hello.java

create mode 100644 README.md

// Skip staging and commit all the file changes by option -a or --all

$ **git commit -a -m "message to describe commit"**

### View the commit log: git log

You can use "git log" to list the commit data:

* “--oneline” for more concise presentation
* “--graph” for the graph of branches
* "--stat" to view the file statistics:

$ **git log**

commit 858f3e71b95271ea320d45b69f44dc55cf1ff794

Author: *username* <*email*>

Date: Thu Nov 29 13:31:32 2012 +0800

First commit

$ **git log --oneline**

$ **git log --graph**

$ **git log --stat**

commit 858f3e71b95271ea320d45b69f44dc55cf1ff794

Author: *username* <*email*>

Date: Thu Nov 29 13:31:32 2012 +0800

First commit

Hello.java | 6 ++++++

README.md | 2 ++

2 files changed, 8 insertions(+)

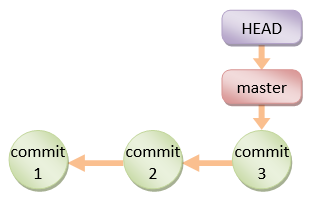
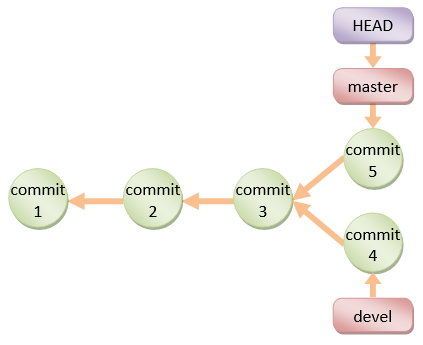
## Branch

### Pointers to branches:

A branch in Git is a lightweight movable pointer to one of the commits. For the initial commit, Git assigns the default branch name called master and sets the master branch pointer at the initial commit. As you make further commits on the master branch, the master branch pointer moves forward accordingly.

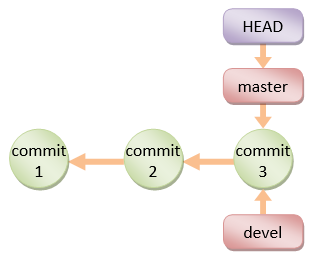
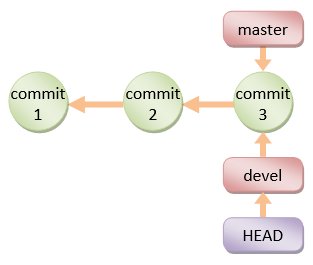
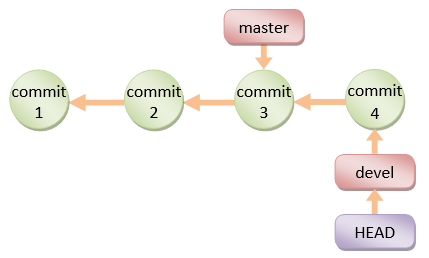
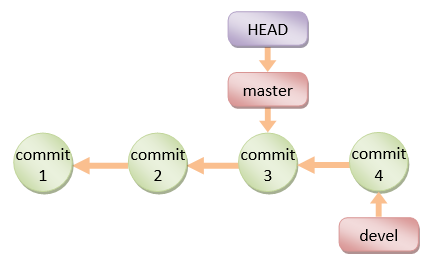
Git also uses a special pointer called HEAD to keep track of the branch that you are currently working on. The HEAD always refers to the latest commit on the current branch. Whenever you switch branch, the HEAD also switches to the latest commit on the branch switched.

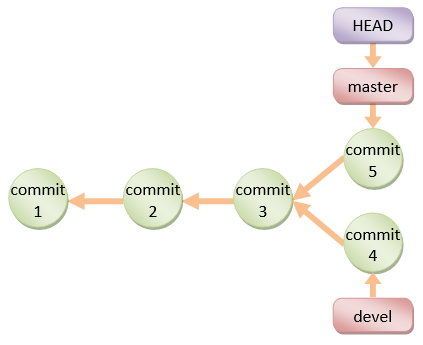
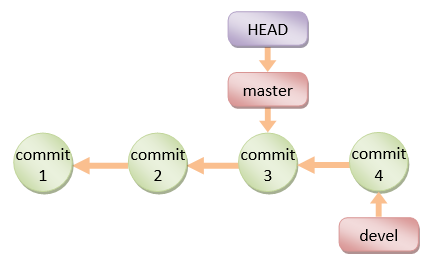
Example: How to create a status on the right from the left?

 🡺 

Step 1: Create devel branch at commit 3;

Step 2:

🡪🡪🡪



### Branch Names Convention

* “master” branch: the production branch with tags for the various releases.
* “development” (or “next” or “devel”) branch: developmental branch, to be merged into master if and when completes.
* “topics” branch: a short-live branch for a specific topics, such as introducing a feature (for the devel branch) or fixing a bug (for the master branch).

### List, create and switch

//list all branches

**git branch -a**

//If you haven’t created/modified any branch, no branch is shown even the default branch Master.

//create, delete, rename a new branch  
**git branch <new\_branch\_name>**

**git branch -d <branch\_name>**

**git branch -m <new\_name>**

// switch current branch to another one

**git checkout <branch\_name>**

**git checkout -b <branch\_name>** // create a new branch and then switch to it

**git switch <branch\_name>** // from git 2.23

**git switch -c <branch\_name>** // (from git 2.23) create a new branch and then switch to it

**git switch -** // switch back to the previous branch before one switched

## Merge/Rebase

### Merge

Merging anotherBranch to currentBranch means:

- creating a common commit, called merge commit, for the both branches

- move the head of each branch to that common commit

So after merging, you still have two branches whose heads now point to the (same) merge commit.

The merge commit is different from other commit in that it has two parents.

// To merge a branch with the current local branch:

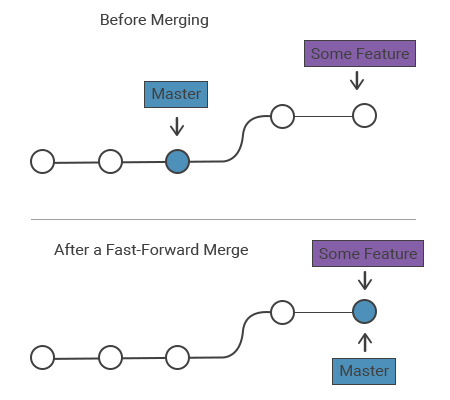
**$ git merge <branch\_name>**

**$ git merge <remoteRepos/remotebranch>**

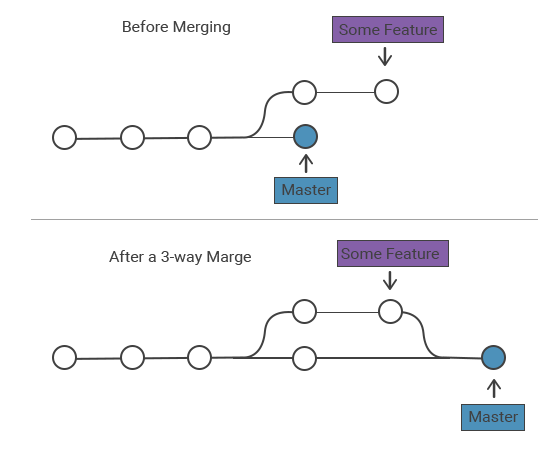
There are 2 scenarios for merging anotherBranch to currentBranch:

Git can create the merge commit automatically:

* 1. Fast-forward: When anotherBranch is just ahead of currentBranch a number of commits, Git just uses the latest commit as the merge commit and move the currentBranch’s head to match the anotherBranch’s head



* 1. 3-way merge: When currenBranch modifies some files and anotherBranch modifies **different** files then GIT can easily combine the head commits of anotherBranch and currentBranch to create the merge commit. The name “3-way” means the 3 commits.



Git cannot create the merge commit due to conflict so you have to create the merge commit manually. Conflict happens when both anotherBranch and currenBranch update a same file. In this case, Git modifies that file by adding all the changes from both anotherBranch and currentBranch to the file plus visual indicator:

<<<<<<<- Conflict marker, the conflict starts after this line.

======= - Divides your changes from the changes in the other branch.

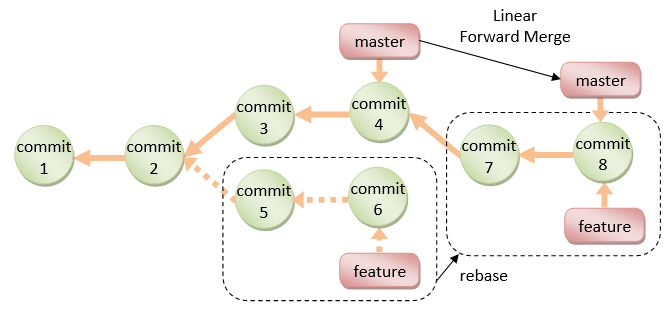
>>>>>>> - End of the conflicted lines.  
Now you have 2 choices:

* + Either abort the merging by: git merge - - abort
  + Or manually edit the files causing conflict and then commit; this commit will become the merge commit. You do this commit as usual by: git add <names\_of\_those\_files> and git commit on the currentBranch.

### Rebase

Rebasing is the process of moving or combining a sequence of commits to a new base commit. The primary purpose for rebasing is to maintain a linear project history. Rebasing is often used in a feature branching workflow.

For example, if you checkout a devel branch and work on commit-5 and commit-6, instead of doing a 3-way merge into the master branch and subsequently remove the devel branch, you can rebase the commit-5 and commit-6, on commit-4, and perform a linear forward merge to maintain all the project history. New commits (7 and 8) will be created for the rebased commit (5 and 6).



$ **git rebase <base-name>**

// <base-name> could be any kind of commit reference such as an commit-name, branch name, tag.

## Stash

## Synchronizing remote and local

### Create alias for remote repo:

Create a remote repo named “test” on GitHub/BitBucket

Associate the remote repo’s URL to a remote name in the local repo by git remote add remote-name remote-url

// Add a remote repo called "origin" via "git remote add <remote-name> <remote-url>"

// For examples,

$ **git remote add origin https://github.com/*your-username*/test.git** // for GitHub

$ **git remote add origin https://*username*@bitbucket.org/*your-username*/test.git** // for Bitbucket

You can list all the remote names and their corresponding URLs via "git remote -v", "git remote show"

// List all remote names and their corresonpding URLs

$ **git remote -v**

origin https://github.com/*your-username*/test.git (fetch)

origin https://github.com/*your-username*/test.git (push)

**$ git remote show origin**

\* remote origin

Fetch URL: https://github.com/tranhuythang/LamJava.git

Push URL: https://github.com/tranhuythang/LamJava.git

HEAD branch: master

Remote branch:

master new (next fetch will store in remotes/origin)

Local ref configured for 'git push':

master pushes to master (local out of date)

Also delete or rename a remote name

// Delete a remote name

$ **git remote rm <remote-name>**

// Rename a remote name

$ **git remote rename <old-remote-name> <new-remote-name>**

### TRACK an upstream branch

#### Upstream branch (remote branch), remote-tracking branch, tracking local branch

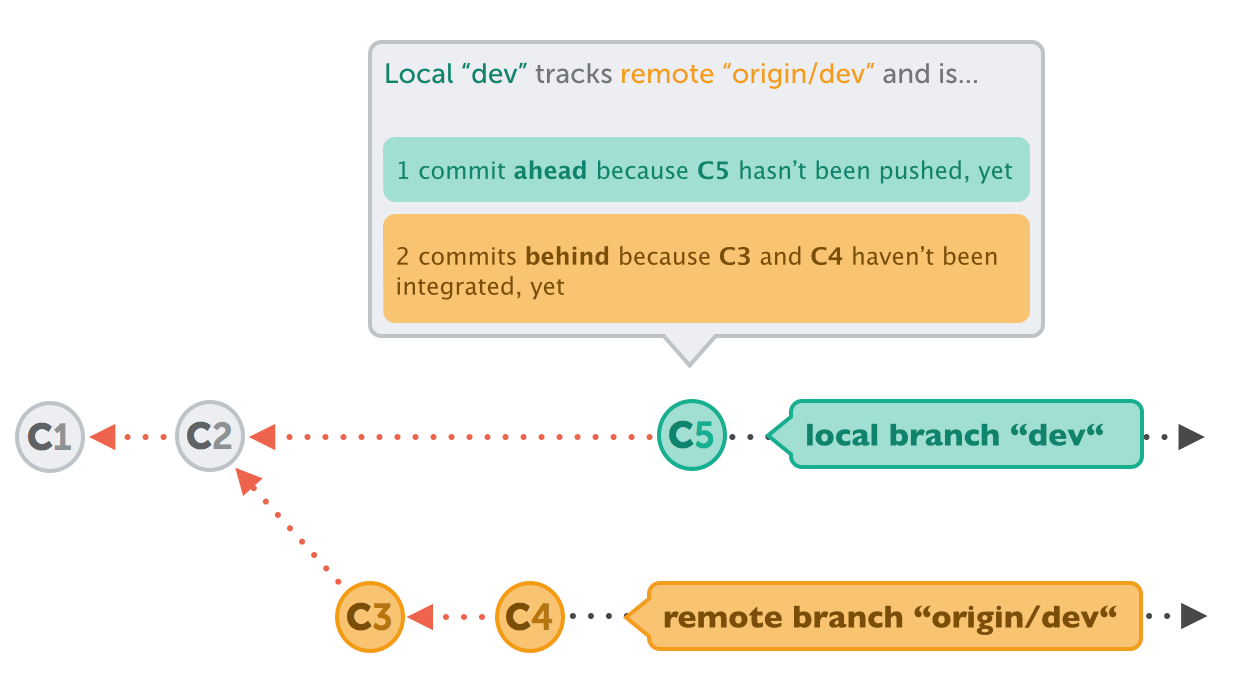
A local branch can track a remote branch so that whenever the two branches are not in sync, users can use git status (after git fetch) to know the difference between the two branches. In order to do this, git create a local copy of the remote branch, and the comparison is done actually between this local cache and the tracking branch.

The branch (on the remote repo) that is tracked is called an upstream or a remote branch.

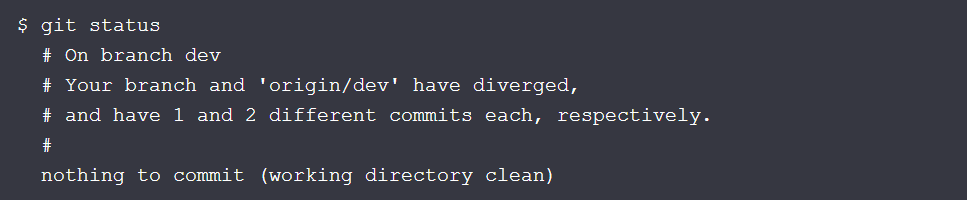
The branch (on the local repo) that tracks the upstream branch is called the tracking (local) branch.

The local copy (cache) of the remote branch is a branch (on the local repo) and it’s called the remote-tracking branch.

Exp: local “dev” tracks remote branch “dev”. A local copy of this remote branch is created as “origin/dev”.



“git fetch” command will update “origin/dev” (by downloading data from the remote repo) and then “git status” will tell how the two branches diverge. The local has 1 difference (C5) and the “origin/dev” has 2 differences (C3, C4).



#### How to track an upstream branch

A local branch tracking a remote branch is actually a local branch being associated with a remote-track branch (that is a local cache of the remote branch) so the remote-track branch should be created before or while associating.

The most natural way to create remote-track branches is to fetch them from the remote repo by:

**$ git checkout --track origin/dev**

Create a new local branch and associate it with an existing remote-tracking branch

// Set an existing local branch to track an Upstream branch with: git branch -u (“-u” means upstream)

**$ git checkout --track origin/dev**

Branch dev set up to track remote branch dev from origin.

Switched to a new branch 'dev'

Associate an existing local branch with an existing remote-tracking branch

$ **git branch -u <remote\_repos>/<remote\_branch> [local\_branch]** // “-u” means upstream

// If local\_branch is dropped, the current branch is used.

Pushing a local branch to a remote repo and set it to track the corresponding remote branch

$ **git push -u <remote-name> <branch-name>** // “-u” means upstream

#### List remote-tracking branches and tracking local branches

“git branch -r” or “git branch -a” shows remote-tracking branches

“git remote show <remote\_repo\_name> shows details about remote branches and remote-tracking branches

// Set an existing local branch to track an Upstream branch with: git branch -u (“-u” means upstream)

**$ git remote show origin**

\* remote origin

URL: https://github.com/my-org/complex-project

Fetch URL: https://github.com/my-org/complex-project

Push URL: https://github.com/my-org/complex-project

HEAD branch: master

Remote branches:

master tracked

dev-branch tracked

markdown-strip tracked

issue-43 new (next fetch will store in remotes/origin)

issue-45 new (next fetch will store in remotes/origin)

refs/remotes/origin/issue-11 stale (use 'git remote prune' to remove)

Local branches configured for 'git pull':

dev-branch merges with remote dev-branch

master merges with remote master

Local refs configured for 'git push':

dev-branch pushes to dev-branch (up to date)

markdown-strip pushes to markdown-strip (up to date)

master pushes to master (up to date)

### Fetch, Checkout, Merge a remote branch

#### Fetch

The "git fetch" command imports commits from a remote repo to the remote-tracking branches on your local repo. It updates only the remote-tracking branches, but does not update the tracking branches nor your local working directory. This gives you a chance to review changes before merging them into your working directory.

Note that git fetch will create a new remote-tracking branch if it’s not there while downloading upstream branches.

// Fetch ALL branches from the remote repo to remote-tracking branches on your local repo

$ **git fetch <remote-name>**

// Fetch a specific branch from remote repo to a corresponding remote-tracking branch in local repo

$ **git fetch <remote-name> <branch-name>**

#### List all (fetched) remote-tracking branches

List all the fetched remote-tracking branches with “-r” in “git branch”.

// List the LOCAL branches: git branch

$ **git branch**

\* master

devel

// List the REMOTE-trakcing branches: adding “-r”

$ **git branch -r**

origin/master

origin/devel

another\_remote\_repo/master

another\_remote\_repo/devel

#### Check out and merge (fetched) remote-tracking branches

You can checkout (switch to) a remote-tracking branch to inspect the files/commits. But this puts you into "Detached HEAD" state, which prevents you from updating the remote branch.

// Check out a remote branch with git checkout, but add the remote repo name before the branch name.

$ **git checkout remote\_repo/remote\_branch\_name**

Merge a fetched remote-tracking branch

// Merge a fetched remote-tracking branch to the current local branch.

$ **git merge remote\_repo/remote\_branch\_name**

### Pull and Push

“git pull” combines "git fetch" and "git merge" into one command, for convenience.

// Fetch the remote's copy of the current branch and merge it

// into the local repo immediately, i.e., update the working tree

$ **git pull <remote-name>**

// Same as

$ **git fetch <remote-name> <current-branch-name>**

$ **git merge <remote-name> <current-branch-name>**

// linearize local changes after the remote branch.

$ **git pull --rebase <remote-name>**

"git push <remote-name> <branch-name>" exports commits from local repo to remote repo.

// Push the specific branch of the local repo

$ **git push <remote-name> <branch-name>**

// Push all branches of the local repo

$ **git push <remote-name> --all**

$ **git push <remote-name> --tag**

// Push all tags. Note that "git push" does not push tags

// Push a local branch and then set it to track the corresponding remote branch

$ **git push -u <remote-name> <branch-name>**

When your local repo diverges from remote repo and you push your local repo to the remote repo then the pushing will be rejected. You will be asked to pull the remote repo first, manually merge and then push again.

## Undo: restore, checkout, reset, revert, amend

### Fixing the last commit

// Amend only the message of the LAST commit

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

// To add some file changes to that LAST commit, simply stage them and then commit --amend again

**$ git add some/changed/file.ext**

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

### Restore tracked files to an old commit

* To restore a specific tracked file (staged or unstaged) to the version of the latest commit or a commit with some hash number use git restore (git 2.23.0 or lager) or git checkout. After this command, the current content of the file is gone but all the other files remain!

// restore a tracked file to the latest commit

**$ git restore <file> // git 2.23.0 and later**

**$ git checkout <file> // old git**

// restore a tracked file to some commit in the history

**$ git restore --source=<#hash> <file> // git 2.23.0 and later**

**$ git checkout <#hash> <file> // old git**

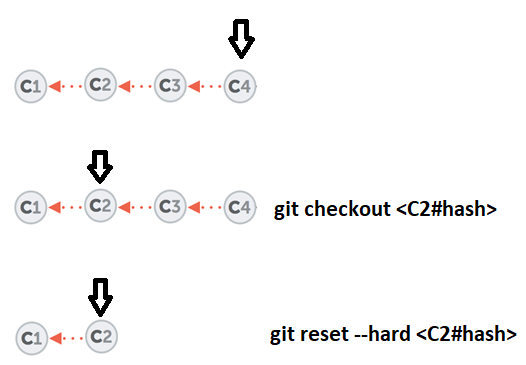
* To restore ALL tracked files to some commit, use “checkout” to keep the following commits, or use “reset --hard” to discard the following commits. Both “checkout” or “reset” will bring all files, folders in that commit to the working directory with overwriting the current ones.
  + Checking out a commit in the history will move the Head to that commit (the following commits remain). This state is called “detached head”; at this state, you can view files, but in order to change files and make a new commit, you have to create a new branch to retain that new commit.

**$ git checkout <#hash>**

* + Reset the working directory to a commit will erase all the commits following that commit.

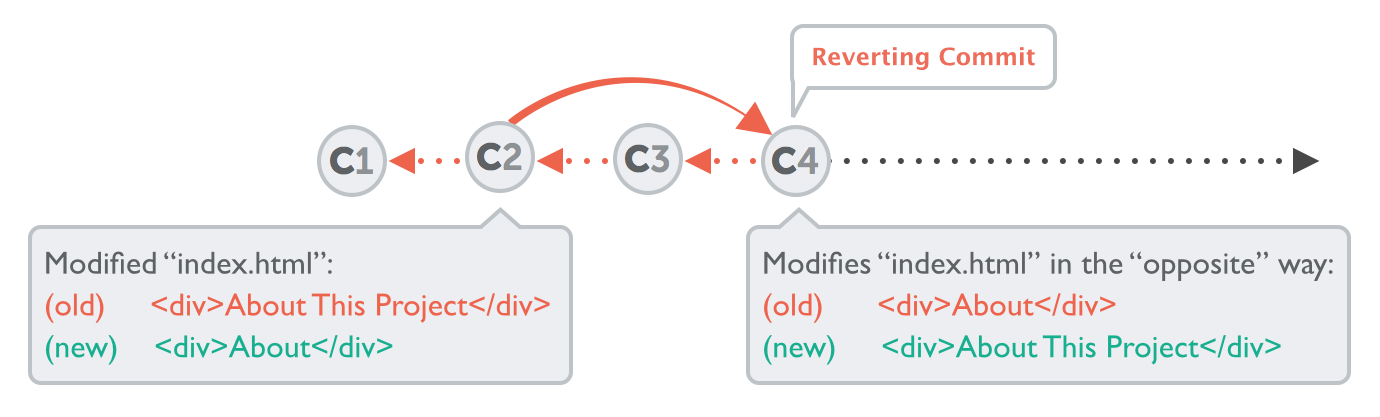
**$ git reset --hard <#hash>**

Exp: your last commit is C4 (with C1, C2, C3 preceeding) and you want to bring C2 to the working directory. “checkout” will keep C3, C4 but they are not visible to you. “reset --hard” will delete C3, C4.



### Revert changes of a commit in the current commit:

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”.



## Tagging

Tag can be used to tag a specific commit as being important, for example, to mark a particular release, like v1.1.5.

You should commit your code and push it to the remote repo as often as needed (e.g., daily), to BACKUP your code. When you code reaches a stable point, create a tag to mark the commit, which can then be used for CHECKOUT.

There are two kinds of tags: **lightweight** tag and **annotated** tag. Lightweight tag is simply a pointer to a commit. Annotated tag contains annotations (meta-data) and can be digitally signed and verified.

**Creating an Annotated Tag (git tag -a <tag-name> -m <message>)**

To create an annotated tag at the latest commit, use "git tag -a <tag-name> -m <message>", where -a option specifies annotation tag having meta-data. For example,

$ **git tag -a v1.0.0 -m "First production system"**

// List all tags

$ **git tag**

v1.0.0

// Show tag details

$ **git show v1.0.0**

// Show the commit point and working tree

To create a tag for an earlier commit, you need to find out the commit's name (first seven character hash code) (via "git log"), and issue "git tag -a <tag-name> -m <message> <commit-name>". For example,

$ **git log**

......

commit 7e7cb40a9340691e2b16a041f7185cee5f7ba92e

......

Commit 3

$ **git tag -a "v0.9.0" -m "Last pre-production release" 7e7cb40**

$ **git tag** // List all tags

v0.9.0

v1.0.0

$ **git show v0.9.0** // Show details of a tag

......

**Creating Lightweight Tags (git tag <tag-name>)**

To create a lightweight tag (without meta-data), use "git tag <tag-name>" without the -a option. The lightweight tag stores only the commit hash code.

**Signed Tags**

You can signed your tags with your private key, with -s option instead of -a.

To verify a signed tag, use -v option and provide the signer's public key.

## Compare: git diff

git diff is to compare files

// Compare files from the last commit and the unstaged files

$ git diff

$ git diff <file>

// Compare files from the last commit and the staged files

$ git diff --staged

$ git diff <file> --staged

How to read diff?

Exp1:

$ **git diff**

diff --git a/Hello.java b/Hello.java

index dc8d4cf..f4a4393 100644

--- a/Hello.java

+++ b/Hello.java

@@ -2,5 +2,6 @@

public class Hello {

public static void main(String[] args) {

System.out.println("Hello, world from GIT!");

+ System.out.println("Changes after First commit!");

}

}

The older version (as of last commit) is marked as --- and new one as +++.

Each chunk of changes is delimited by

"@@ -<old-line-number>,<number-of-lines> +<new-line-number>,<number-of-lines> @@".

Added lines are marked as + and deleted as -.

In the above output, older version (as of last commit) from line 2 for 5 lines and the modified version from line 2 for 6 lines are compared. One line (marked as +) is added.

Exp2:



## Git Host (github, bitbucket)

### Fork and Pull Request

"Fork" and "Pull Request" are features provided by GIT hosts (such as GitHub and BitBucket):

* Pushing "Fork" button to copy a project from an account (e.g., project maintainer) to your own personal account.
* Pushing "Pull Request" button to notify other developers (e.g., project maintainer or the entire project team) to review your changes. If accepted, the project maintainer can pull and apply the changes. A pull request shall provide the source's repo name, source's branch name, destination's repo name and destination's branch name.

### Issue raising and tracking

Issue tracking is integrated into

# Git glossary

Git (distributed) vs TFS (centralized): Git allows commit to local repos while TFS not

Git is a DISTRIBUTED version control system while TFS is a Centralized version control system

Both Git and TFS have local repos. But Git tracks and allows users to commit changes to local repos while TFS doesn’t have this mechanism.

To commit changes to remote repos:

GIT: commit changes to local and push local to remote

TFS: commit changes directly to the remote

Repo (repository): a [Git repository](http://bitbucket.org/code-repository) is a storage of all the **versions** of your project. Don’t think about repo as a server hosting all of your projects; rather each repo hosts a project, so a server hosts many repos.

Remote repo: the final version of the project. Local repo commit changes to the remote repo.

Working directory vs Repo: a repo stores many versions of files while working directory contains only the files you are working with.

Commit: Git store versions of project as commit, each commit points to its previous commit.

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>

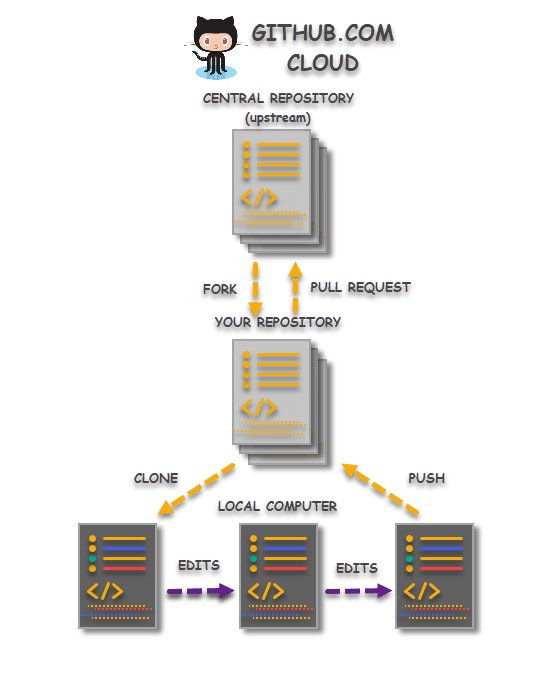
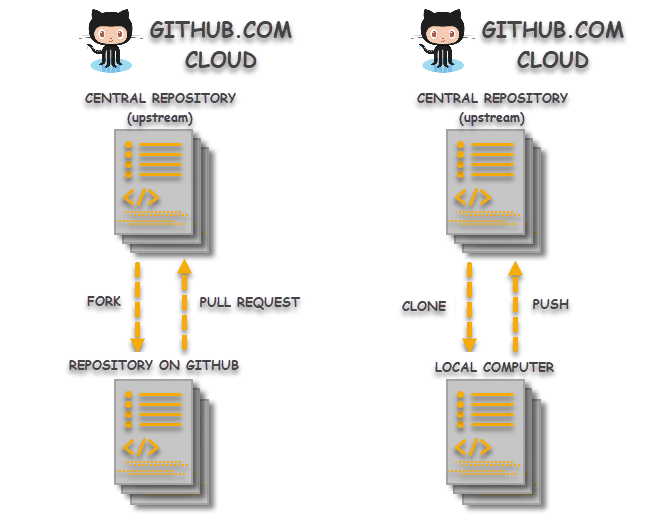
Pull request: a contributor request the owner of the project to consider accept the contributor’s change

Fork vs Clone: Both Fork and Clone means copying a repo to another place. Fork is a term of GitHub (not Git)

Github: Forking a repo (on Github) means copying that repo to another Github account. Usually the other Github account doesn’t have the right to contribute to the ogirinal repo, so the other Github account has to make a pull request when they want to update the code.

Git: Cloning a repo to a local computer. Usual the local account has the right to push the local code directly to the remote repo.

So FORK if you don’t have the right to push; otherwise, use CLONE.



# 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 remote repos

* adding Remote Repos NAME  
  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>

* 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 branch or “local\_branch” to a remote with new name “remote\_branch”  
git push -u <remote\_repos> <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 branch

Recall: the default name for the starting **branch** of the local repo is Master (but the default name for a remote **repos** is Origin).

Git doesn’t have commands for remote branch. Do things with branches on the local repo and then push it to the remote repo.

* list all branches:

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 branch  
  git branch <new\_branch\_name>
* delete a local branch:

git branch -d <branch\_name>

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

git checkout <branch\_name>

* create a new branch and then switch to it:

git checkout -b <branch\_name>

## 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

Merging anotherBranch to currentBranch means:

- creating a common commit, called merge commit, for the both branches

- move the head of each branch to that common commit

So after merging, you still have two branches whose heads now point to the (same) merge commit.

The merge commit is different from other commit in that it has two parents.

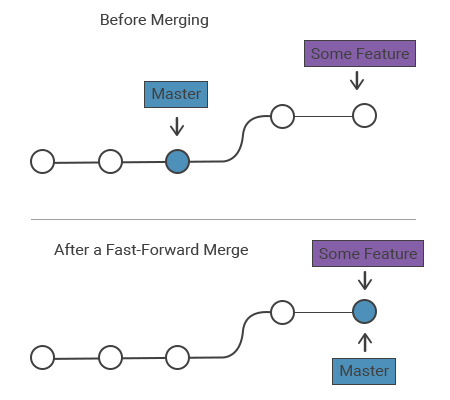
To merge a branch with the current local branch:

git merge <branch\_name>

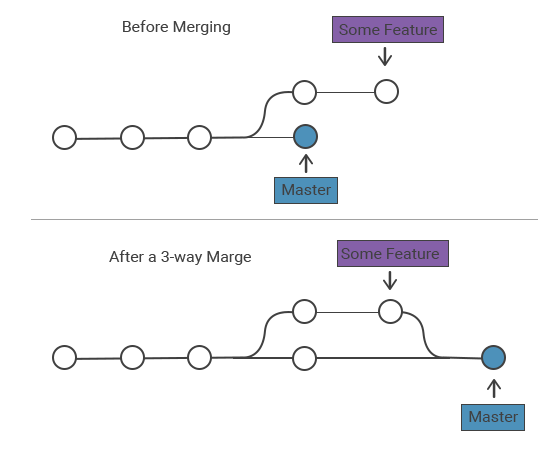
git merge <remoteRepos/remotebranch>

There are 2 scenarios for merging anotherBranch to currentBranch:

* Git can create the merge commit automatically:
  + Fast-forward: When anotherBranch is just ahead of currentBranch a number of commits, Git just uses the latest commit as the merge commit and move the currentBranch’s head to match the anotherBranch’s head



* + 3-way merge: When currenBranch modifies some files and anotherBranch modifies **different** files then GIT can easily combine the head commits of anotherBranch and currentBranch to create the merge commit. The name “3-way” means the 3 commits.



* Git cannot create the merge commit due to conflict so you have to create the merge commit manually. Conflict happens when both anotherBranch and currenBranch update a same file. In this case, Git modifies that file by adding all the changes from both anotherBranch and currentBranch to the file plus visual indicator:

<<<<<<<- Conflict marker, the conflict starts after this line.

======= - Divides your changes from the changes in the other branch.

>>>>>>> - End of the conflicted lines.  
Now you have 2 choices:

* + Either abort the merging by: git merge - - abort
  + Or manually edit the files causing conflict and then commit; this commit will become the merge commit. You do this commit as usual by: git add <names\_of\_those\_files> and git commit on the currentBranch.

## 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)

## Undo

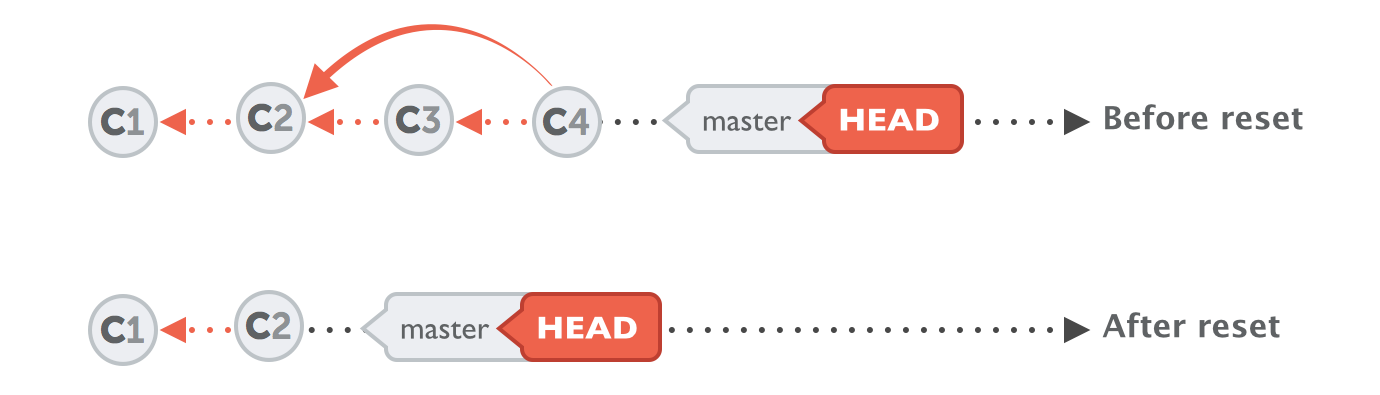
* Fixing the last commit
  + Amend only the message of the LAST commit

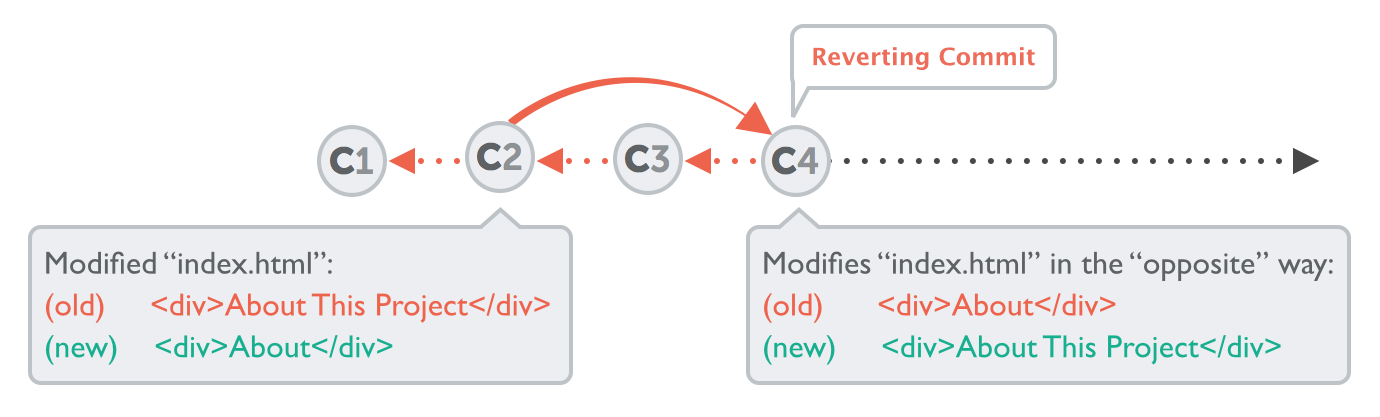
$ git commit --amend -m "This is the correct message"

* + If you want to add some more changes to that LAST commit, you can simply stage them as normal and then commit --amend 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: working directory, local repo, remote repo and stage area

## 4 basic concepts: Remote repo, local repo, working directory and stage

Version Control System: Each project is contributed by many people.

The final version of a project, which is hosted by a remote server, is called Remote repo.

Each user copies this remote repo to their own local repo. They modified and push their change to the remote repo.

**Working directory – Stage - Local repo**

You prepare files in your working directory and when they’re finalized, you commit it to your local repo.

A local repo contains all the versions of the project while the working directory contains only the files that you’re working, which is not necessarily the content of the current version (snapshot).

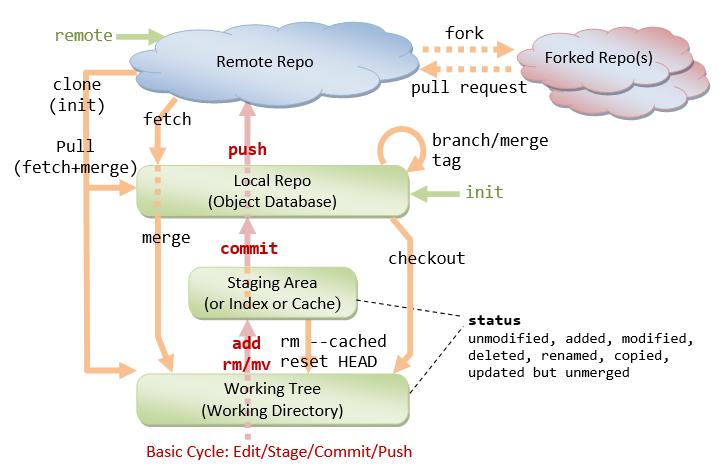
Before committing, you have to stage files. This extra step is useful in this situation: you have to modify file1, file2, file3, file4 that relate to each other. After modifying file1, file2, you want to commit file1, file2, but without file3, file4, the whole code will not run, so you stage file1, file2 and delay committing file3, file4 after finishing them.

**Local repo – Remote repo – Forked repo**

When the local repo is finalized it should be pushed to the remote repo.

On Github, a repo can be forked (copied) to another Github account. This other Github account often doesn’t have the permission to push their changes so it has to make a pull-request to the original repo.

(Local repo has the permission to push the change while forked repo doesn’t have so it has to make a pull-request).

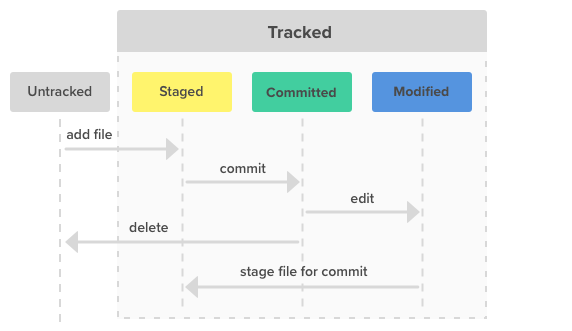
When the remote repo is on Github, it can be forked (copy) to another Github account. However, the other Github account doesn’t have the permission to push their change, so it has to make a pull-request to the owner of the project.

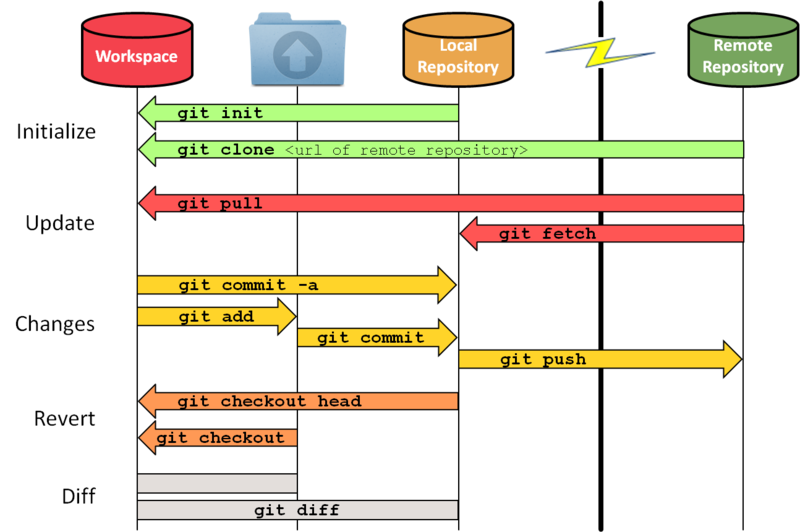
## Tracked and Untracked files

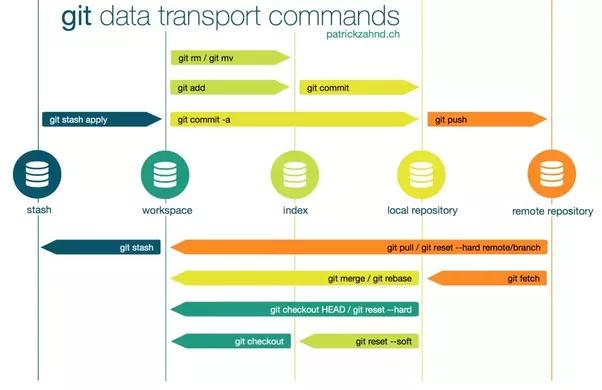
Remember that each file in your working directory can be in one of two states: **tracked** or **untracked**. Tracked files are files that were in the last snapshot; they can be unmodified (committed), modified, or staged. In short, tracked files are files that Git knows about.

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.







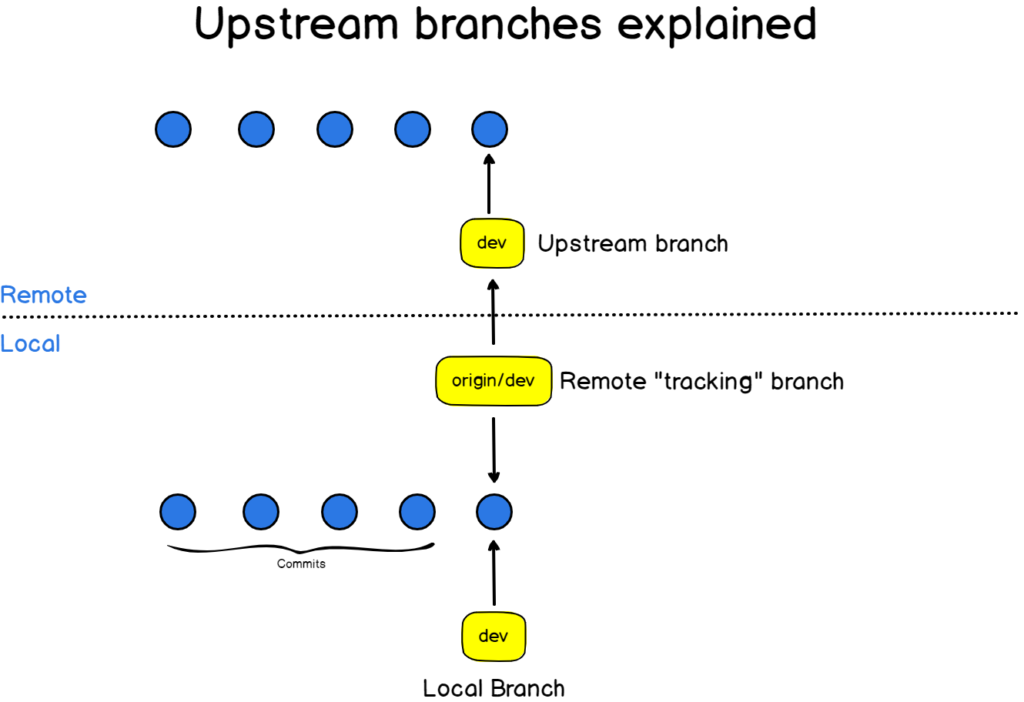
## What are there in a working directory (what is shown)?

* 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.

## Remote branch vs remote-tracking branch vs upstream and TRACKING a remote branch

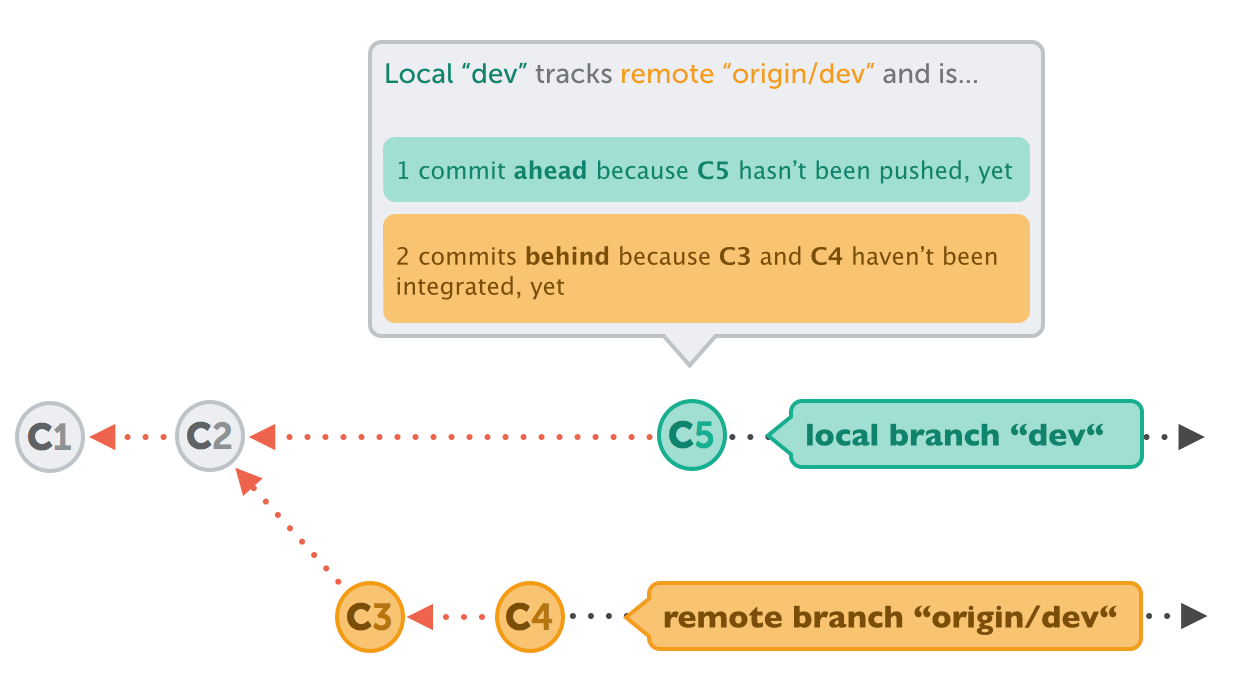
<https://www.git-tower.com/learn/git/faq/track-remote-upstream-branch/>

https://devconnected.com/how-to-set-upstream-branch-on-git/



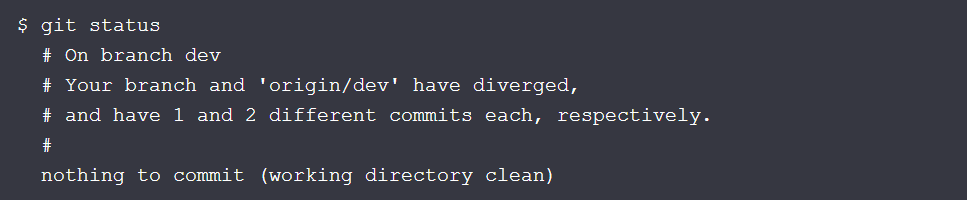
A local branch can track a remote branch so that whenever the two branches are not in sync, git can inform the local branch about this (Note this requires git fetch).

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 so you are not at the HEAD (the latest commit) of the branch; this state 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)

