GIT

Git is a very powerful & widely used Version Control System

→ records all the change to our files over time & so that we can recall specific versions when required

3 States about Git file.

→ Modified

we have a git file & took that (Extracted)

we changed few lines of code in that file. but not saved yet,

→ Staged

it's like you saved the code/document, but yet the document is not finalized or not shown to your manager to make it public.

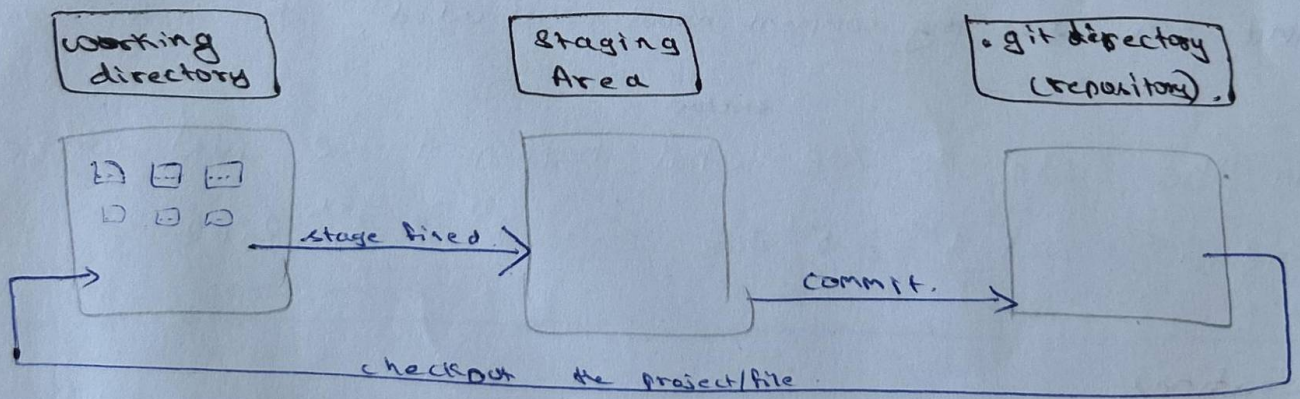
official launch

It's like telling the Git that we want to include this current version (changed document) of this modified file in our next commit.

→ Committed

It's like your saved document & clearly renamed it & uploaded to Pand or set to your manage to make it public.

Based on the things that we are doing to the file there are 3 section of our GIT project



working directory : It's like our desk where we spread all our papers & write, edit, draw --- do anything u want.

when we do "git clone" to our repository, the we get all files in our working directory

Staging area : It's like special tray on the desk, where you keep the papers that are filled & are ready to send to ur manager

It keep tracks of what changes we have done for the next commit

• git directory : It's like a storage room, where every version/document is stored safely

It is the heart of the git

Each & every file in the working directory can be → Tracked / → Untracked

every human being can be in these 3 states → Jagrath, Swapna, Sushuthi

~Tracked.

means Git knows knows about this file & have all the information about it. (↳ like:)

→ Unmodified. file haven't changes since the last time saved.

→ Modified. made the changes since the last save, but haven't told git to prepare this changes for next official save

→ Staged : specifically told Git that current version of this modified file is ready & to be included in next save (commit).

~Untracked.

means exact opposite, Git doesn't know about this. may be

→ newly created files.

→ existing files but never used it (modified/anything).

It's like new item come to ur home are in packet but u didn't sorted them & stored (or added) in particular box.

Code:

will create a Information.txt file & this data is write to that

```
echo 'This text will write into that file' > Information.txt
git add Information.txt.   → this code is where we do staging (added)
git commit -m 'give any comment about what u did'   (this is to commit)
```

status

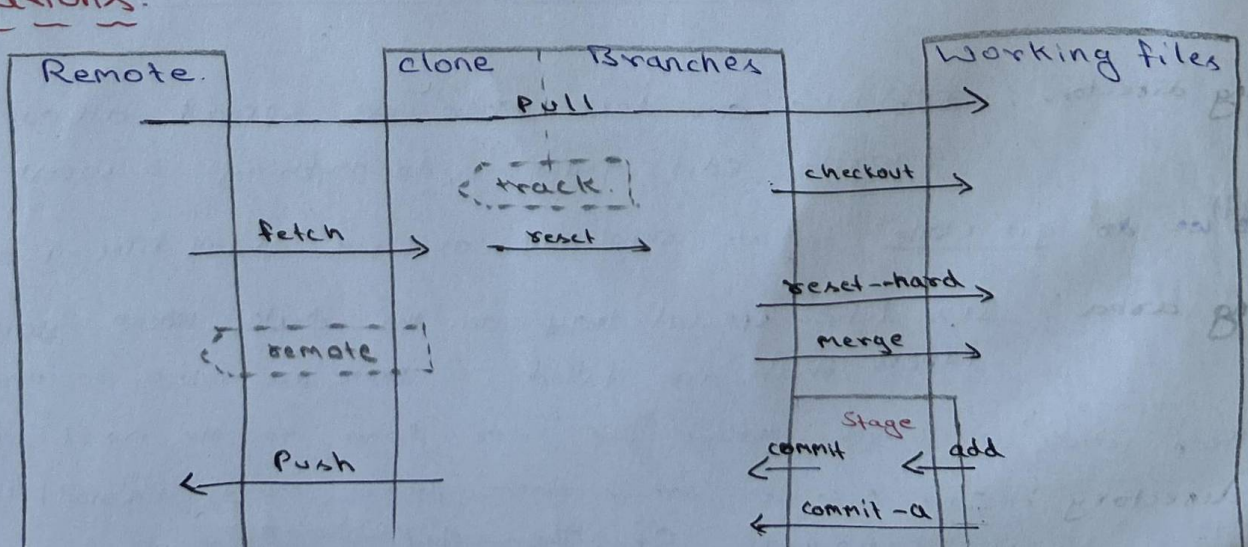
Check in the code to see what happen if we just write 1st line, (OR) just 1st & 2nd line (OR) 1st & 3rd line.

1st line → untracked

1st & 2nd line → changes to be commited

1st & 3rd line → git without staging

Git Operations.



② Git Stash

→ this allows us to temporarily "save changes without committing them"

→ Useful when we need to switch branches or pull changes, but the working directory is not clean.

It's like ~~pause~~ Pause Ur work.

→ help others.

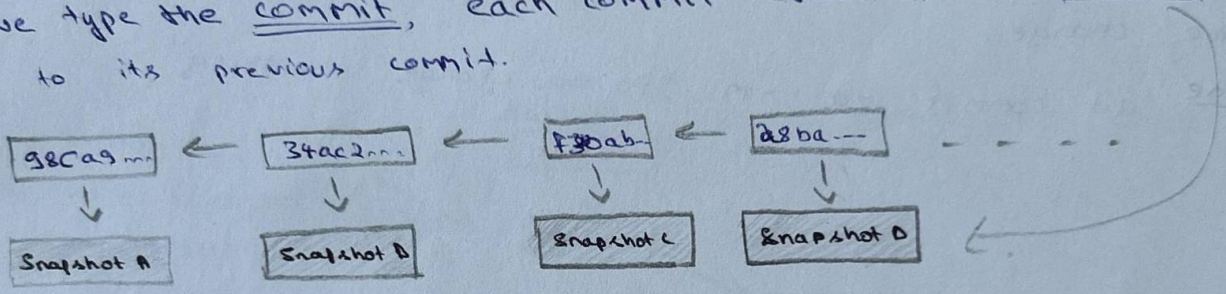
then resume back to ur work.

Basic Commands

git stash	saves our changes to a new stash.
git stash apply	applies the most recent stash to working directory
git stash pop	—//— & also removes from stash list
git stash drop	removes specific stash from list

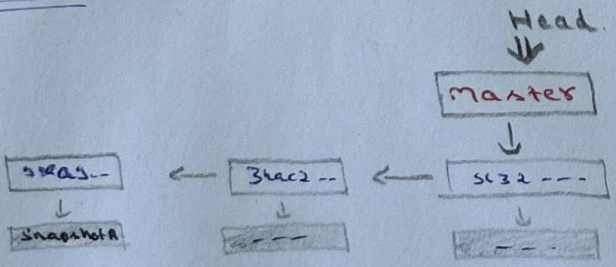
Working with branches

When we type the commit, each commit creates a snapshot and points to its previous commit.



WKT Default Branch name is : "master"

"Head" is like a pointer that shows where we are working



Branching means we create a diffrent branches & work in that instead of disturbing the [illegible]/master branch.

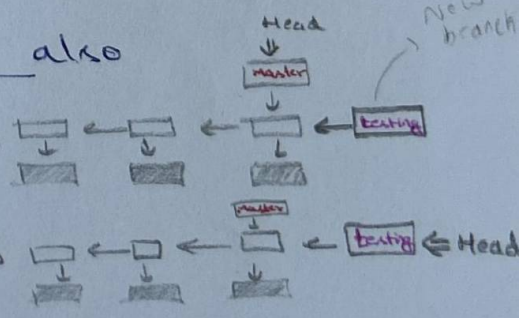
So we create a branch & change the pointer also

① ②

Code

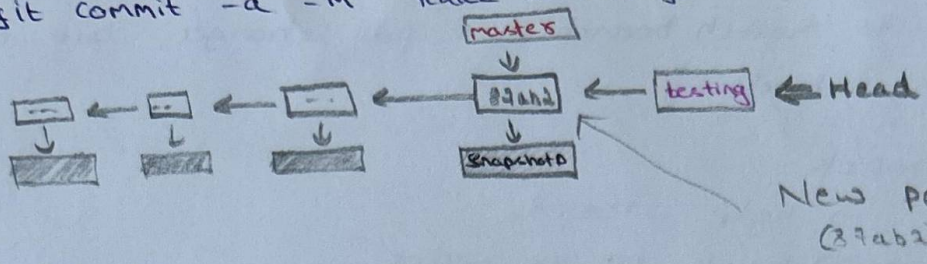
① git branch branch_name. # to create branch

② git checkout branch_name # to point the header



Suppose now we make a change & commited.

code git commit -a -m "Make a change"

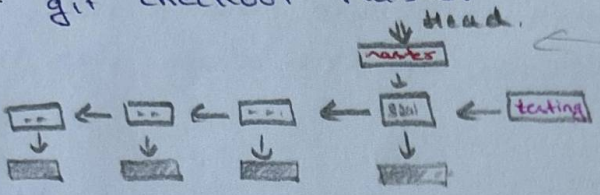


New point & snapshot is created (3?ab2)

Suppose now if we want go to master & do some changes.

I) we need to change the pointer.

code git checkout master

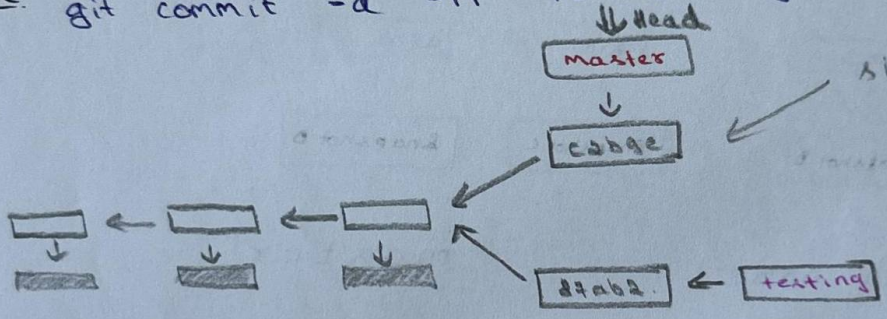


means not outing we are inning

Header is moved.

II) Do the change.

code git commit -a -m "Do the change something"

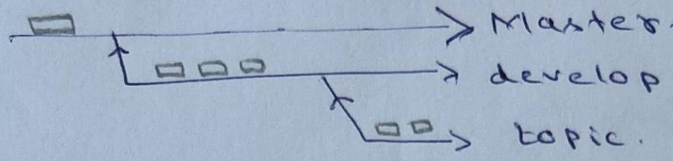


since the commit / change happened new point / screenshot is created

(3) Remote GiT Repositories

When we work on big projects, then multiple - long - running branches will be there all, some branches are always open. open & running.

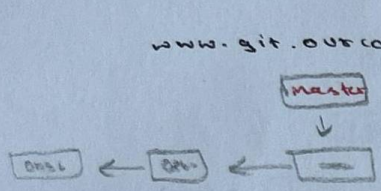
For convience we divide into different stages

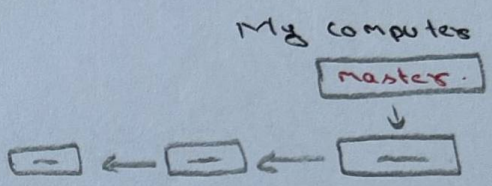


Master branch is like stable code running / to be released.

Develop branch is to work or to do test. (from develop it moves to master if code is error free & approved)

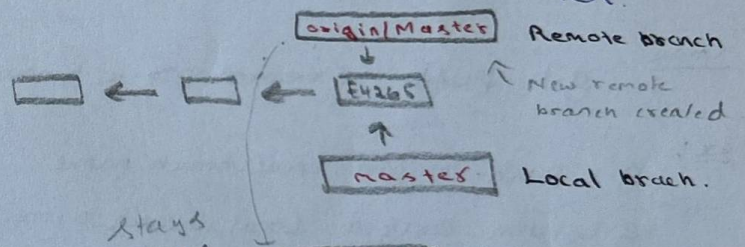
Topics branch is small branches created to add new features / to work on bugs





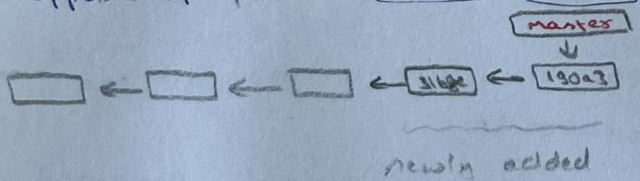
once we do the. git clone. jane@git.....git.

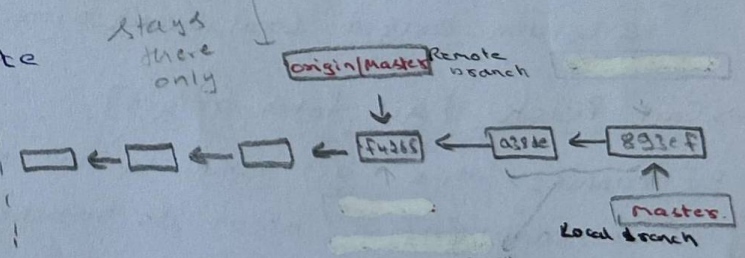
(cloning will changes the local master also)



Local vs Remote changes.

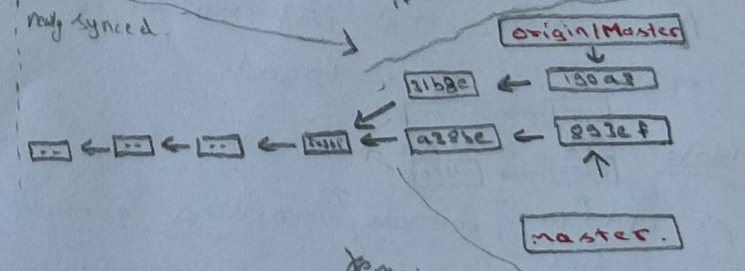
Suppose if pushes 31b8e, 190a3 to remote



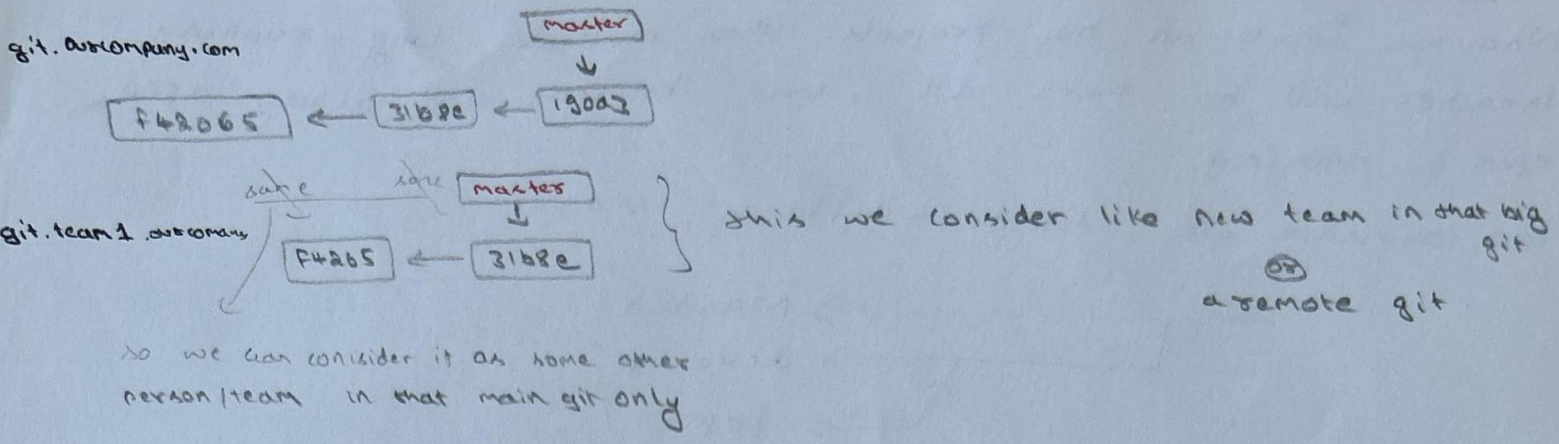


To sync (synchronize) with remote.

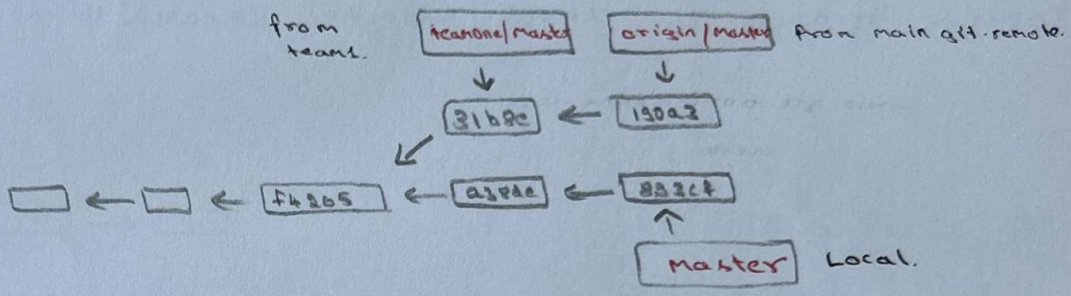
command: git fetch origin.



Suppose now if we want to syn multiple teams / remotes then.



To synch. this with our git. ① Add code ⟹ git remote add <remote> <url>
② fetch. code ⟹ git fetch <remote>



OK, Now let's see how we can push our local git to remote.

code
git push <remote> <branch>

ex:
git push origin Local_branch_name. # Pushes our branch to origin/Local_branch_name
(or)
git push origin Local_branch_name: name_on_remotebrach_name # pushes our branch to origin/name_on_remote_branch_name
in this name it stored in remote

→ Git fetch [git fetch origin].
By running this, git connects to remote repository & downloads all the latest information about its branches, tags & commits.

Suppose if someelse pushed new branch. after you type this command, then also, that information & that new branch will also automatically reflects in your local.

Note: It doesn't create a local branch for you, that you can start committing

it updates or creates "remote-tracking branch"

these are like
origin/main
origin/featurex
origin/serverfix
:
these are just read only pointers.

Overall it just gives updates but we still didn't copied them into our notebook & start editing that.

(4)

→ git checkout. [git checkout -b brach_name origin/branch_name]

Creates a new brach & immediatly switched the Head to it

Adding this at end means telling git to create new local brach_name starting from the exact same commit that origin/branch_name points to

* Advantage is "upstream tracking relationship" (simply automatically track origin/branch_name)

means now branch_name is treated as "home"

→ when we run git pull on our local branch_name, git automaticall knows to git fetch origin branch_name

→ when we run git merge (or rebase) into our local then it knows to go to where

→ when we run git push on our local branch_name, git automatically knows to push changes to origin/branch_name.

☆ git pull Shortcut to → { git fetch, git merge }

) but Problem is ends with merge conflict

Q: Merge or Rebase.

Both helps to integrate changes from one Git branch to another, achieve a similar goal, combines histories but do it in different way.

Let's consider a situation where we have. (A) (B) commits on main. and we created a new branch "feature" from B & commited. (C)

A - - B (main) → (created a new brunch) → A - - B, B - C (feature) → (switch back & make another commit) → A - - B - D (main), B - C (feature)

Merge.

Using merge, integrates the changes from one branch to anothe by. "creating a new commit" & this commit have 'two parent commits.' ☆

ex: git checkout main # head moves to main (we saw this earlier)
git merge feature (or) make sure to we are in branch where we want to merge.

now we are in D branch.

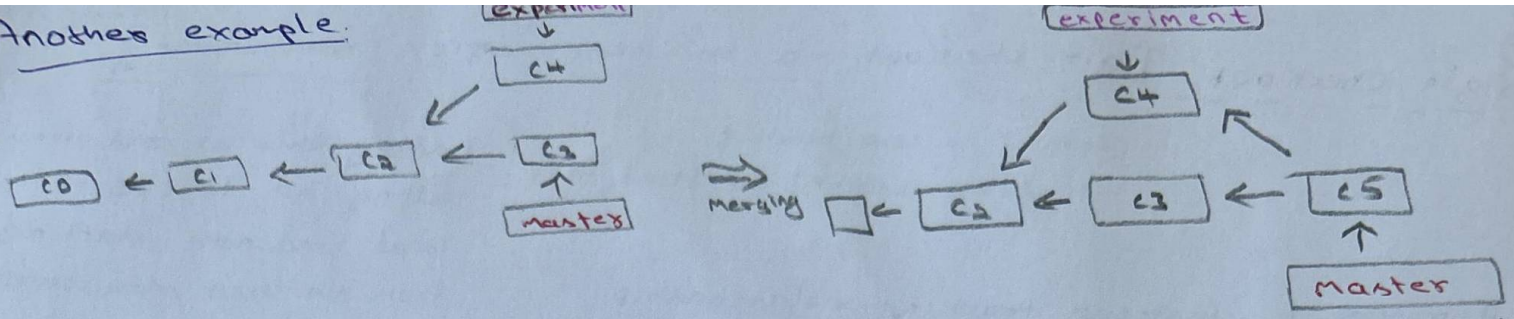
A - B - D (main), C (feature) (original) → (After merging) → A - B - D - E (main), C - E (feature) (After merged)

→ created because of merge commit

So Now (E) contains all the changes from D & all the changes from C.

(E) has 2 parents (C) & (D) origin/ commits (C & D) untouched

Another example.



Explanation :- Goes back to ancestor C2, creates new snapshot (C5) & commits

Pros :-

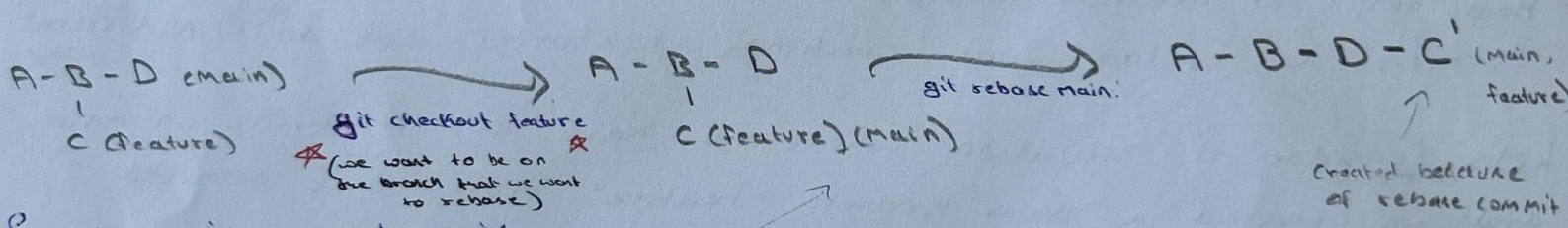
* Preserves the exact history of how branches diverged & merged (doesn't rewrites)
* If suppose merge introduces issues, single commit to revert or reset

Cons :-

* history filled with many merges doesn't making linear history, harder to follow
* Graph look like diamond, many developers find less clean

Rebase

Using rebase, integrates the changes by taking the commits from your 'feature' brach. & repaying them one by one on top of the target's branch latest comit



So, how it works is.

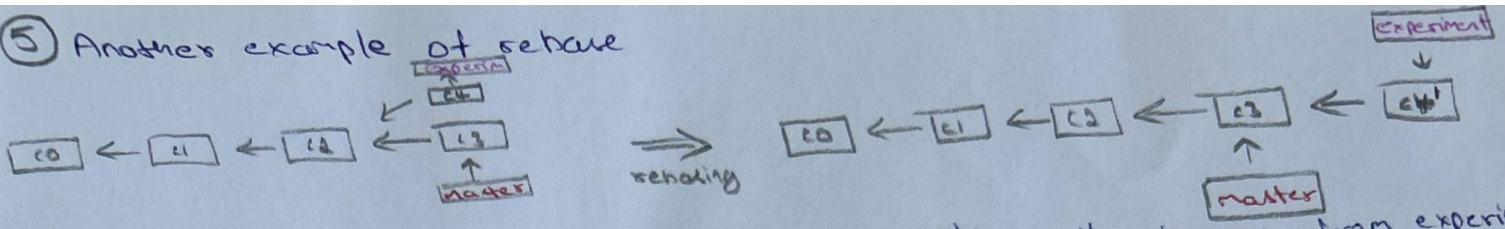
when u do git rebase main

1) Git identifies the common ancestor. for C is B.
2) temporarily "saves" our commit C.
3) rewinds our feature branch back to B.
4) Then applies commit D (from main).
5) finally it replays our commit C on the top of D, creating a new commit C'

C' has the same changes as C but has a single parent. (D) (but for merge it had both C & D) & different commit

* When we go to main (C') then do git merge feature then it will be a fast forward merge. because main is ancestor of feature.

⑤ Another example of rebase



Explanation: Goes back to ancestor C2, applies all changes from experiment to master branch, creates new snapshot C4'. Apply fast-forward merge to move pointer (master) to C4'

Pros
* clean, linear history (less history)
* avoids merge commits

Cons → Major Problem.
* Rewrites History
* Complexity with shared branches
* Reverting is very difficult.

"NEVER REBASE A BRANCH THAT OTERS HAVE ALREADY PULLED FROM

Q) When to use merge & rebase

* git merge (safely) (working in a team)
* git rebase (carefully) (when we work privately)

{ Merge when in doubt.
{ Never rebase a pushed branch. } Thumb rule in Git

Refer Slides to see on when to avoid rebase & rebase bad examples