sushilbhosaleazure@gmail.com

GIT

There are 2 types of version control system

1. Centralized → SVN, CVS, Perforce, TFS etc.
2. Distributed → Git, Mecurial, Fossil

Git is distributed version control system tool. GIT is developed by Linus Torvalds.

Why GIT is very popular ?

1. Distributed

- No single point of failure. Every developer has local repository
- Performance is more
- Without network also developer can continue his work. Workspace and remote repository need not be always connected

2. Staging Area

- Git commit is a 2 step process.
- First we have to add files to staging area and then we have to commit from that staging area.

- Advantage of staging area is we can cross check our changes before commit.
- Staging area is virtual you cannot physically see it.
- 12 GB files if stored in SVN. GIT only requires 420 Mb.
- Data is stored in encrypted format in background.
- Snapshots of our data is stored in GIT.

③ Branching and Merging

- We can create and work on multiple branches simultaneously and all these branches are isolated from each other.

④ Freeware and Open Source

⑤ Supports multiple platforms.

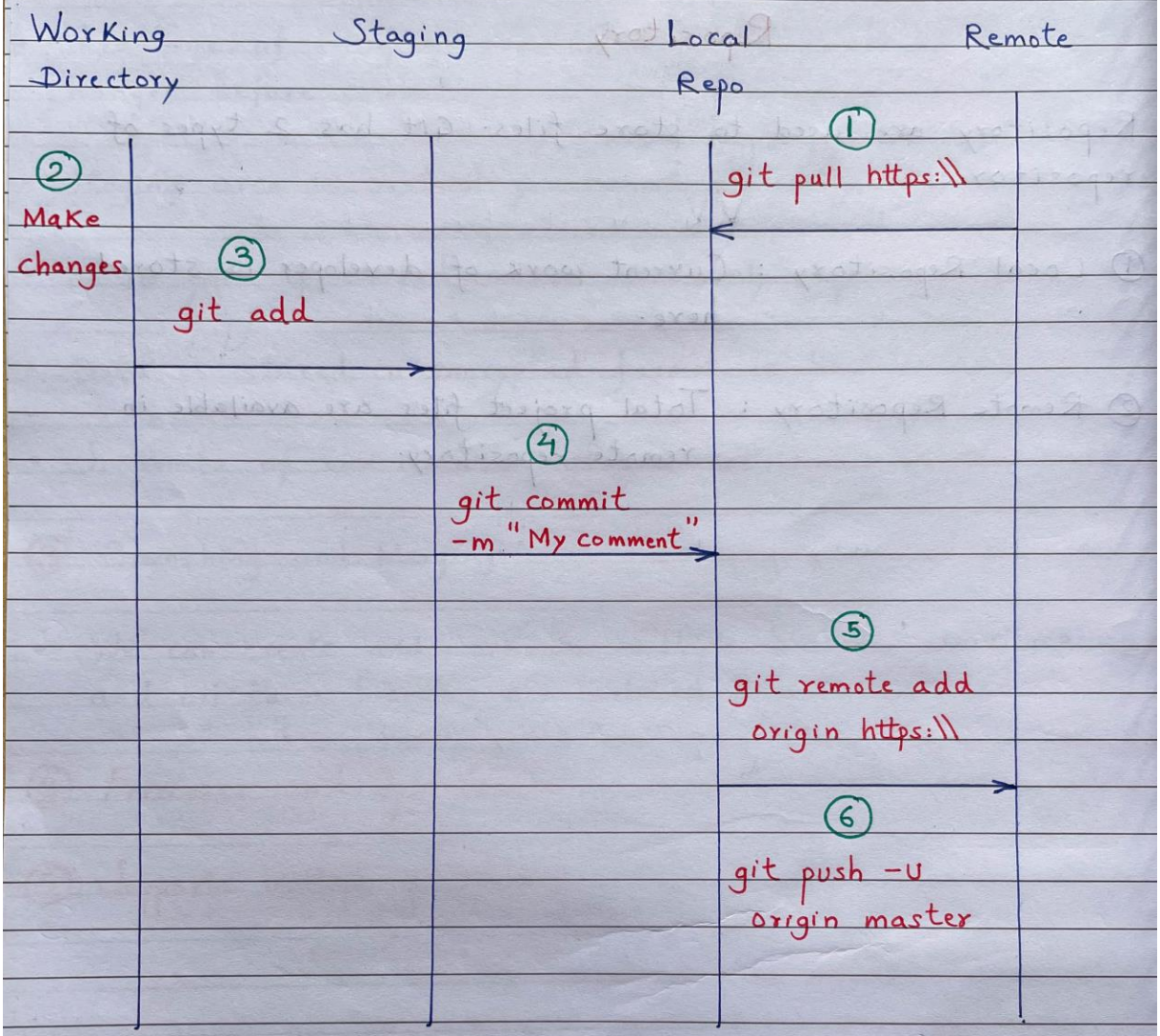
Repository

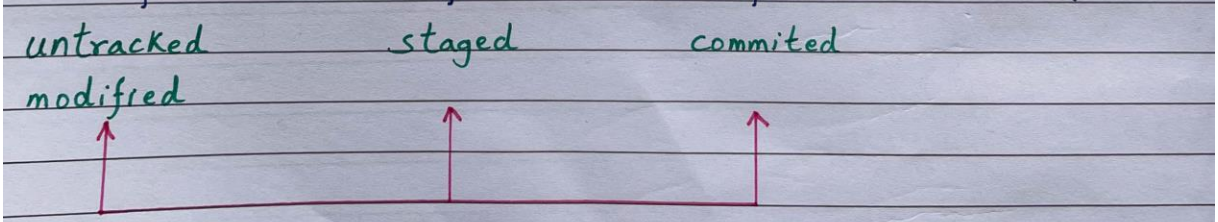
Repository are used to store files. Git has 2 types of repositories.

① Local Repository : Current work of developer is stored here.

② Remote Repository : Total project files are available in remote repository.

Date: YOUVA





State of files (Lifecycle of files) → Use git status to check state of files.

git status -s to get consise output.

Date:

git add → Sending files from working directory to staging area.

git commit → Staged changes will be moved to local repository.

git push → To move files from local repository to remote repository

git clone → To create a new local repository from the remote repository.

git pull → To get updated files from remote repository to local repository.

Date:

Lifecycle of file in GIT

Every file in git, is in one of the following 4 states

→ Untracked

→ Staged

→ In Repository / Committed

→ Modified

① Untracked : Every new file will be created in working directory. GIT does not aware of these new files. Such type of files are said to be in "untracked" state.

git status ⟶ To know status of files in all areas

② Staged : Files which are added to staging area are said to be in staged state.

```
git add a.txt

git add .

git add a.txt b.txt c.txt

git add *.txt
```

③ Committed: Any file which is committed is said to be in repository state or committed state.

```
git commit -m "My comment"
```

④ Modified: Any file which is already tracked by git, but it is modified in working directory is said to be in modified state.

Question:

① While commiting I just want to commit the particular file from stage area then in that case do we need to use filename like 'git commit filea.txt'

No, changes which are there in staging area all those changes will be commited.

② Can we check filesize in commited area?

No, it is available in encrypted form

Date:

If the file added to staging area or committed then we say it is tracked by git

git ls-files ⟶ lists all files which are tracked by git

You create a new folder /home/sushil/myproject now we want version control on myproject folder. By default git will not track the folder. So to tell git to start tracking this folder we use git init command.

cd /home/sushil/myproject

ls -a ← get hidden also
(no files here)

git init
Initialized empty Git repository in /home/sushil/myproject/.git/

ls -a
/.git ← LOCAL REPOSITORY

git init → To provide/create an empty repository to our workspace so that version control is available to our working directory

SCENARIOS

SCENARIO 1

(1) a.txt, b.txt } files already created

(2) git init → so local empty .git folder is created

(3) Will a.txt, b.txt added in local repository? NO

(4) git add → to add files to local repository

SCENARIO 2

There is a local repository and someone has already commited a.txt, b.txt

```
git status
  untracked files a.txt
                  b.txt
```

git add . or git add a.txt b.txt

```
git commit a.txt b.txt
```

↑ Common error by people. You can't use this

```
git commit -m "my comment"
```

```
git status
```

Now you do git init what will happen?

```
git init
Reinitalized existing Git repositories
```

No data is wiped out

Date: YOUVA

Shortcut : You already have a.txt, b.txt files tracked by git make changes in a.txt, b.txt and execute below commands.

```
git add a.txt b.txt
git commit -m 'commit message'
```

OR

```
git commit -a -m 'commit message'
```

NOTE: This shortcut is not applicable for newly created files. It is only applicable for files already tracked by git.

git commit -m "commit message" → We get commit id/ hash which is hexadecimal number of 40 characters

For every commit git records

→ author name & mail id.

→ timestamp

→ commit message

git commit -m 'New files added'

[master (root-commit) dcb4108] New files added.

First 7 characters of commit ID.

git log

commit dcb4108ddc.....da (HEAD → master)

symbolic reference

master represent this commit

commit ID is unique which can be used to identify commit. The 1st 7 characters are also unique

commit 44fe2785f.....88

M T W T F S S

Page No.:

Date:

YOUVA

Question: Is it possible to commit one new file and one modified file at once?

Yes, possible using commit -m 'Message' option

Question: Which hash function is used in git?

SHA-1

Question: After commit staged area becomes free?

No. Files will be available in local repository, staging area, working area

git commit -m 'New file added'

create mode 100644 filea.txt
create mode 100644 fileb.txt

100644 → file permission
. → means type of file.

100 is ASCII text data

read (r) → 4
write (w) → 2
execute (x) → 1

r + w
4 + 2 = 6

rw-	r--	r--
6	4	4
owner user	group	other permissions

Git Log

git log ⟶ Shows history of all commits in local repository

outputs.

commit id
author name , mail id
timestamp
commit message

git log file1.txt ⟶ Shows history of all commits for a file1.txt file.

git log --oneline

28bbdc3 (HEAD → master) A new file z.txt added
73e3bc5 commited a.txt
:
:
:

git log -n 2 ⟶ displays latest 2 commits

git log --grep='added' --oneline

↑ searches commit message comments

```
git log --after="2022-10-18"
git log --after="1 hour ago"
git log --after="5 minutes ago"
git log --author=Sushil
```

Can be used with --before

Git Diff

Using git diff command you can compare difference between files in

- working directory vs staging area — git diff file1.txt
- working directory vs last commit — git diff HEAD file1.txt
- staging area vs last commit — git diff --staged HEAD file1.txt
- working directory vs specified commit — git diff commitid file1.txt
- staging area vs specified commit — git diff --stage commitid file1.txt

etc...

- between two commits — git diff source_commit_id destination_commit_id file1.txt
- last commit vs last but one commit (second last) — git diff HEAD HEAD~1 file1.txt
- to compare all files — git diff source_commit_id destn_commit_id
- two branches in local repo — git diff sourcebranch destnbranch
- local repo vs remote repo — git diff localbranch remotebranch

Git RM

git rm command is used to remove files from different areas.

① Remove files from both staging area & working directory

```
git rm file1.txt
```

How to check files in working directory?

```
ls
```

How to check files in staging area or committed

```
git ls-files
```

```
git rm file1.txt
git status
git commit -m 'file1.txt removed from both working dir
                and staging area'
```

Note: git rm -r . → Removes files recursively

② Remove files from staging area but not from working directory

git rm --cached file4.txt

③ Remove file only from working directory but not from staging area.

No git command for this we have to use normal rm command

rm file4.txt

Interview Question

M T W T F S S
Page No.:
Date:
YOUVA

git checkout command.

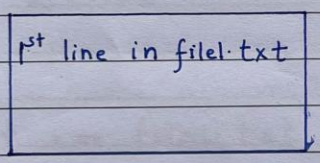
We can use git checkout command to discard unstaged changes in the tracked files of working directory.

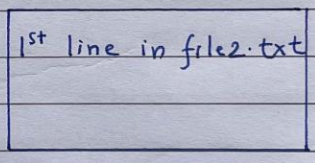
Tracked files means either they are in staging area or committed.

Unstaged changes are changes which are not added to staging area.

Scenario

STEP 1 : Create 2 files file1.txt , file2.txt in working directory





STEP 2 : Add files to staging area and local repository

```
git add .

git commit -m "Two files added"

git log
```

STEP3: Come to working directory and add new line in file1.txt

```
1st line in file1.txt
2nd line in file1.txt
```

STEP4: Add file in staging area

git add file1.txt

STEP5: Again come to working directory and add new line in file1.txt

```
1st line in file1.txt
2nd line in file1.txt
3rd line in file1.txt
```

STEP6: git checkout ⟶ outputs which files can be checked out

M file1.txt

STEP7: git checkout -- (space here) file1.txt

```
1st line in file1.txt
2nd line in file1.txt
```

Git References

For most of the commands (like git log, git diff etc) we have to use commit ids as arguments. But remembering these commit ids is very difficult.

Git provides some sample names for these commit ids. We can use these names directly instead of commit ids. These are just pointers to commit ids.

These sample names are nothing but references or refs.

Most recent commit ids ⟶ master or HEAD

Where are these references stored?

In .git/refs directory

local repository

What is master?

git status

on branch master
nothing to commit, working tree clean

1. master is name of the branch.

2. It is reference (pointer) to last commit. Hence wherever last commit id required, we can use simply master.

cd .git/refs/heads

cat master ⟶ This file contains the recent commit id.

Master will always point only to one commit

git show <commit id of master> ⟶ shows information about the commit

OR

git show master

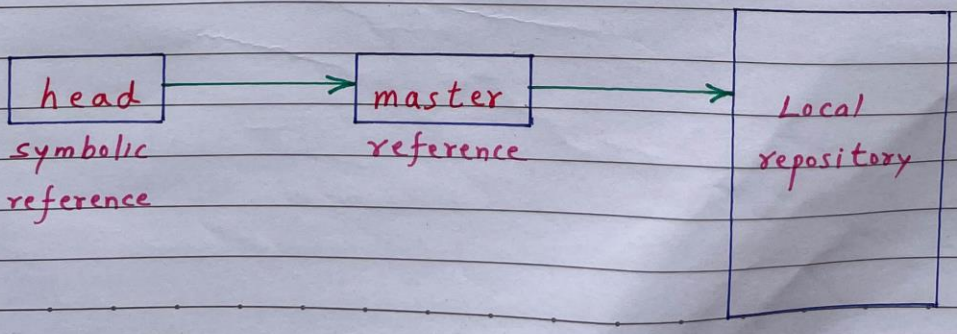
If you want to use second recent commit use below command.

git show master~1

Can we rename master?
Yes

What is HEAD?

HEAD is a reference to master also called as symbolic reference.



Bydefault HEAD always points to the master. But sometimes HEAD may not point to master such type of case is called detached HEAD

HEAD information is available in .git directory

```
cd .git
ls
:
HEAD ⟶ it is a file

cat HEAD
ref: refs/heads/master
```

What is detached HEAD?

Sometimes HEAD is not pointing to the branch name, such type of HEAD is called detached HEAD.

Interview question

M T W T F S S
Page No.:
Date:
YOUVA

git reset ← works on
① staging area
② local repository

There are 2 uses of git reset

① You added some files in staging area bymistake and now you need to remove them from staging area

e.g. git reset file1.txt

② You added some files in staging area and then committed to local repository bymistake and now you need to remove them from local repository.

Repositories always talks about commitid

git reset <mode> <commitid>

NOTE: Use git reset on local repo. only don't do it on remote repo.

Interview question.

M T W T F S S

Page No.:

Date:

YOUVA

git checkout vs git reset

git checkout can be used to discard unstaged changes in working directory	git reset can be used to discard staged changes in working directory

Interview question

M T W T F S S
Page No.:
Date:
YOUVA

git rm --cached vs git reset

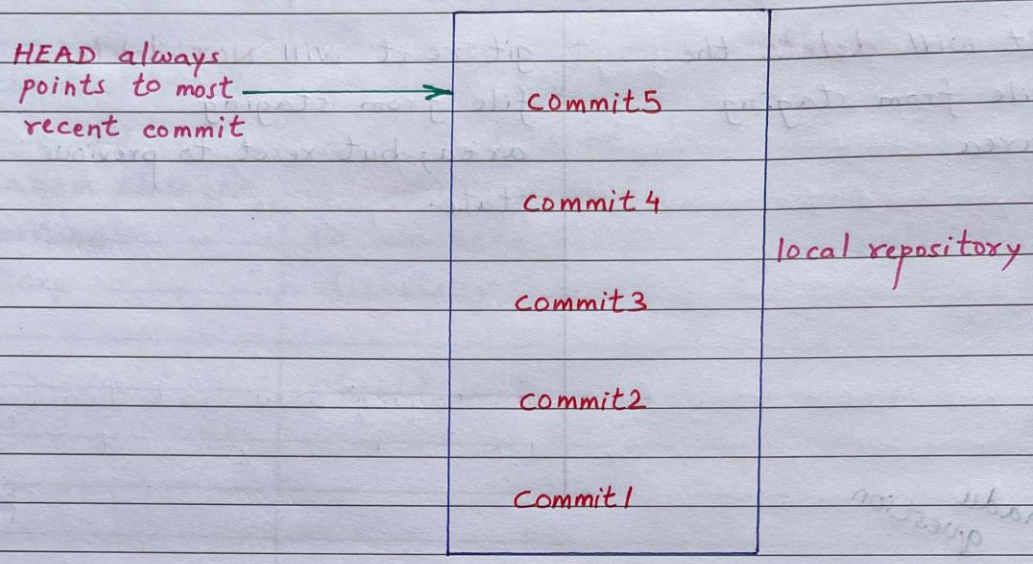
It will delete the file from staging area	git reset will NOT delete file from staging area, but reset to previous state.

Phadu question

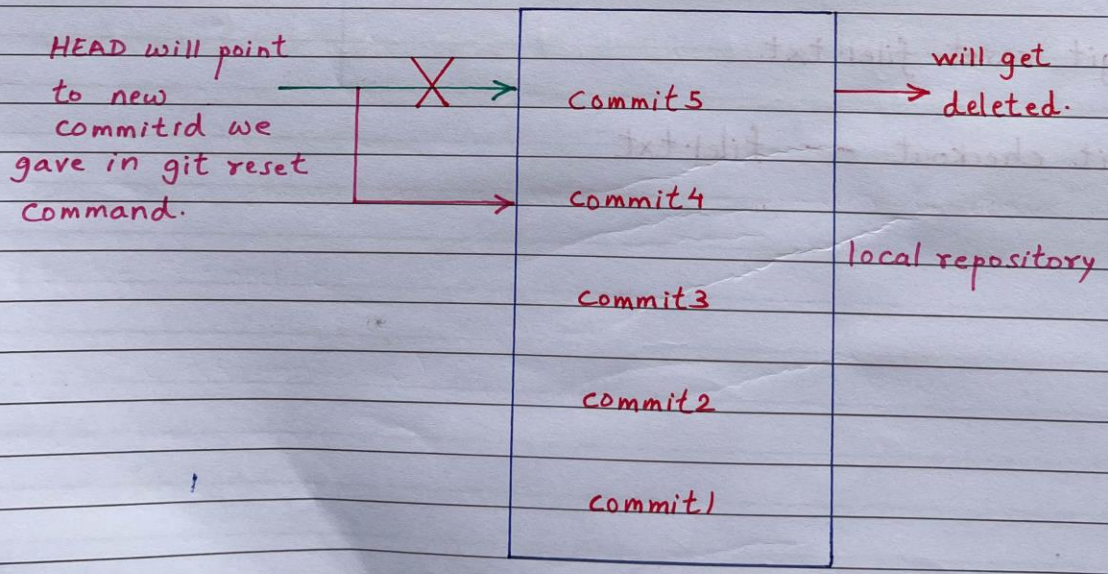
We modified the content of file1.txt and added to staging area. But we want to ignore those changes in both staging area and in working directory. Which commands to use?

```
git reset file1.txt
git checkout -- file1.txt
```

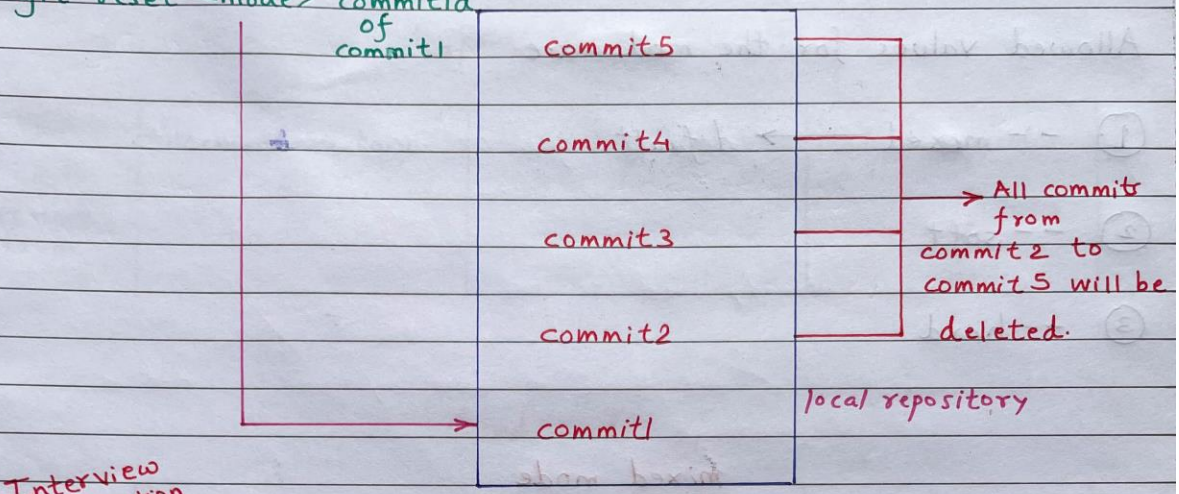
git reset on repository



Now I want it should not point to most recent commit then use command git reset <mode> commitid



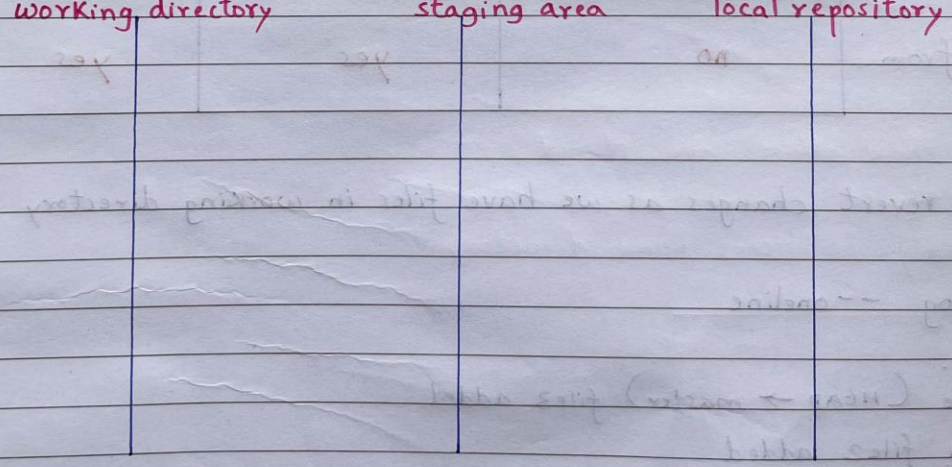
git reset <mode> commitid



Interview question

My requirement is to discard only one commit (commit3) can we do it?

NO. You cannot remove just commit3 and keep others



Now we moved HEAD to specific commitid so all above commits will get deleted but what about files that are there in staging area and working directory. mode decides to remove files from staging area, working directory

Interview question

Page No.: YOUVA
Date:

Allowed values for the mode are

1. --mixed ⟶ default
2. --soft
3. --hard

mixed mode

It is default mode

git reset --mixed commitid OR
git reset commitid.

	working directory	staging area	local repository
remove from	no	yes	yes

we can revert changes as we have files in working directory

git log --oneline

121f44e (HEAD → master) file3 added
0d0172a file2 added
257073d file1 added

To remove last commit ⟶ git reset --mixed 0d0172a
git reset 0d0172a
git reset --mixed HEAD~1
git reset HEAD~1

All are same

soft mode

	working directory	staging area	local repository
remove from	no	no	yes

hard mode

	working directory	staging area	local repository
remove from	yes	yes	yes

git

git alias command is used to give an alternate name to command.

Test alias name already used or not

git one → if it is not there it will say 'one' is not a git command

By using git config command we can define aliases

e.g. git config --global alias.one "log --oneline"

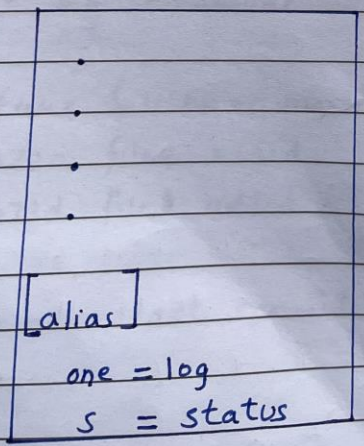
git log --oneline

OR

git one

Where these aliases are stored ?

- gitconfig in users home directory



Ignoring unwanted files

To tell Git not to track certain file we have to add that filename in .gitignore file.

.gitignore is a hidden file. Dot with filename means hidden file in linux.

.gitignore file should be in a working directory only.

cat .gitignore

```
# Ignore a.txt file ——> comment
a.txt ——> filename
```

```
# Ignore all .txt files
*.txt

# Ignore all hidden files
.*

# Ignore all files in logs directory
logs/
```

How git treat directories?

GIT always worry about files but not directories.

git never give any special importance for directories.

Go in working directory and create a folder.

```
mKdir code
```

git status ⟶ working tree clean

Now inside code folder create some txt files or .py files

git status ⟶ will output untracked files

code/ ← directory

Whenever we are adding files, them implicitly directory will also be added.

```
git add .

git status
    new file : code/a.py

git commit -m "Files added"
```

M T W T F S S

Page No.: YOUVA

Date:

Branch

A branch is a version of the repository that diverges from the main working project.

① Once we create a branch all files and commits will be inherited from parent branch to child branch. In child branches we can create new files & we can perform any changes in existing files and we can commit those changes based on our requirement.

② All branches are isolated from each other. The changes performed in one branch are not visible to other branches.

③ Once we complete our work in child branch we can merge that new branch with master branch or we can push that branch directly to remote repository.

git branch IOS ⟶ creates branch with name IOS

git checkout IOS ⟶ switch branch to IOS

Shortcut way to create branch & switch to it.

git checkout -b IOS ⟶ creates branch IOS & switches to it

```
git branche
* IOS
  master
```

Page No.: Date: YOUVA

Now, suppose we have 3 branches master, IOS, android

scenario 1

① git branch

* master
  IOS
  android

② Create new branch java

  git branch java

Now java branch will inherit all files from master branch. Along with commits.

scenario 2

① git branch
  master
* IOS
  android

② Create new branch oracle

  git branch oracle

Now oracle branch will inherit all files from IOS branch. Along with commits.

M T W T F S S

Page No.: YOUVA

Date:

In tool like SVN for branch we have to create a new directory and copy all files manually.

Is git creating a new directory
Is git copying all the files from parent to child } NO

Git only does logical duplication of files

Proof ? see beauty of git here

① Go in folder of project which already has branch & files in it you will see those files in folder.

e.g master branch has files 1.txt 2.txt 3.txt
iOS branch has files 5.txt 6.txt

② git checkout master

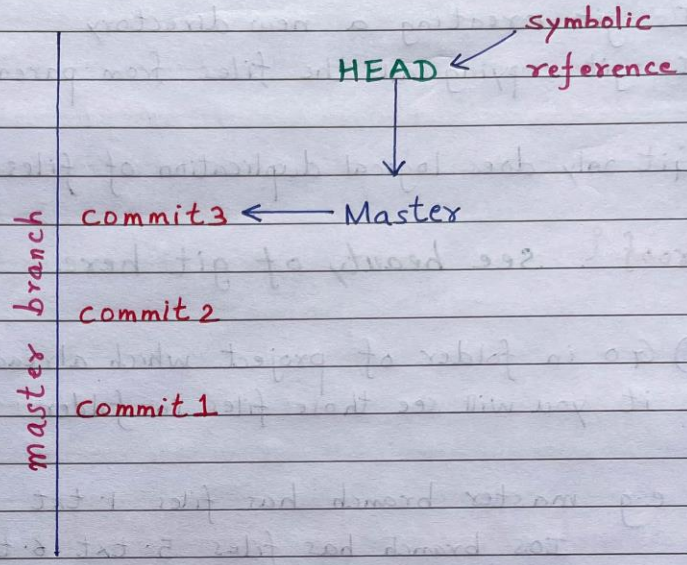
folder will show 1.txt 2.txt 3.txt

③ As soon as you change branch or switch to iOS branch folder will show 5.txt 6.txt

git checkout iOS

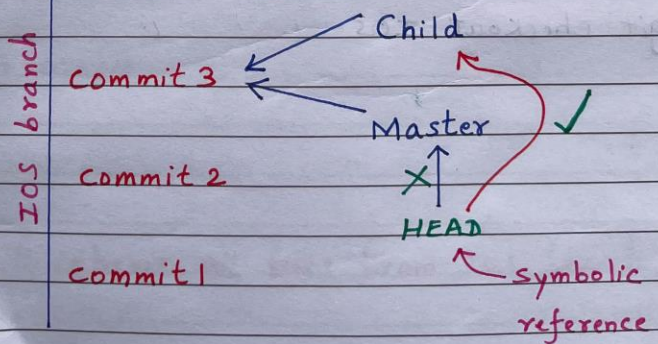
Beautiful Concept

We have a master branch with 3 commits



Master branch

Now we created a child branch IOS from master branch which will also have 3 commits. So HEAD will now point to Child branch. Reassigns HEAD to current active branch.

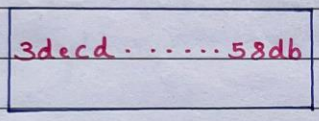


Branches location on machine

```
cd .git/refs/heads/
ls
IOS   master
```

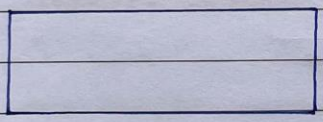
In master you will see the latest commit ID on master branch

```
cat master
```



In IOS you will see the latest commit ID on IOS branch

```
cat IOS
```



```
cd .git/[redacted]
```

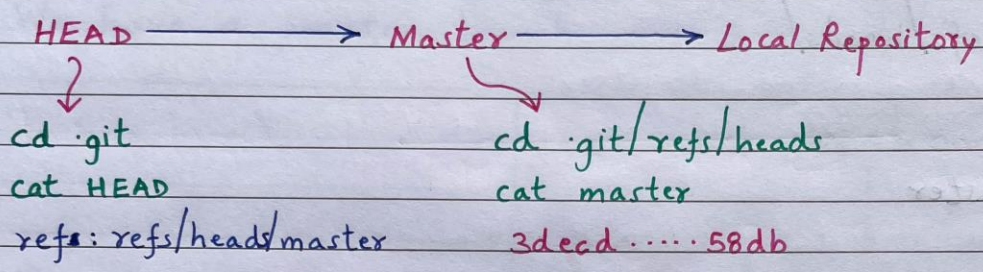
→ First come on IOS branch then open HEAD file.

```
cat HEAD
ref: refs/heads/IOS
```

Now switch to master branch → git checkout master

```
cd .git/
cat HEAD
ref: refs/heads/master
```

Switching from one branch to another internally only changes the HEAD



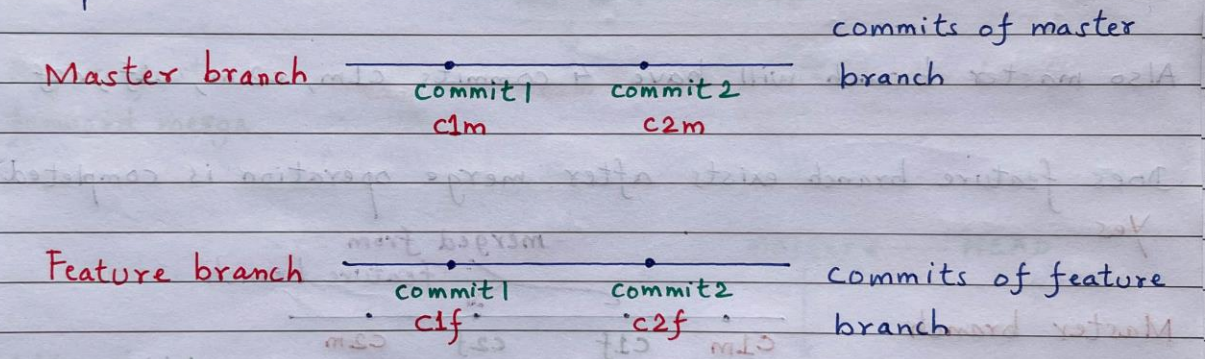
Merging of a Branch

→ we created a branch to implement new feature
→ we completed implementation of that new feature
→ we have to combine those changes from child branch with parent branch. This process is nothing but merging

From master branch we have to execute merge command

git merge childbranch

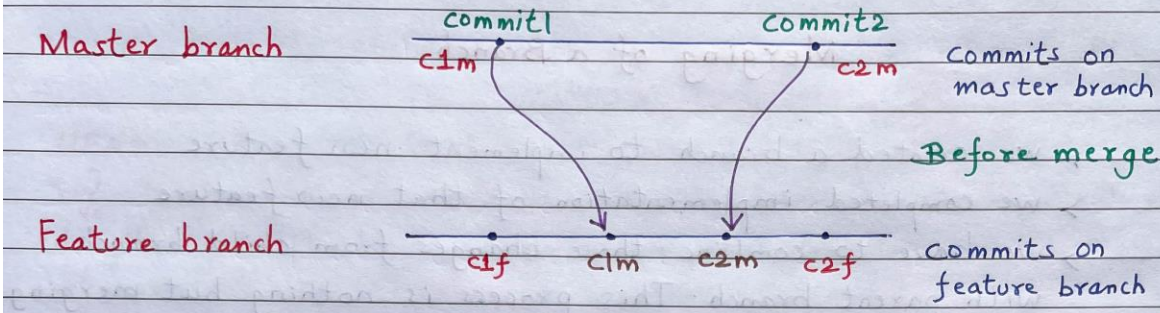
Example.



If you have a.txt, b.txt and 2 commits c1m, c2m in master branch and then you create a feature branch and then have x.txt, y.txt in feature branch with 2 commits c1f, c2f.

Since we have created feature branch from master branch files of master branch a.txt, b.txt will be inherited from master branch to feature branch.

Files in feature branch → a.txt b.txt
x.txt y.txt



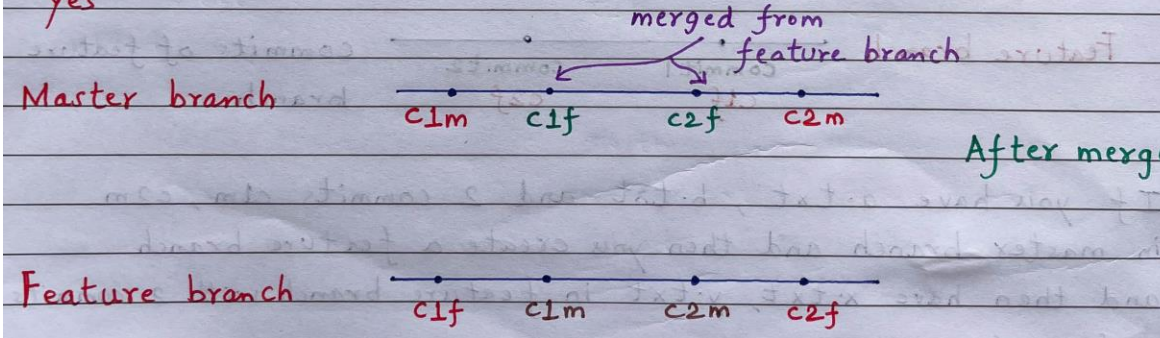
Now I want to merge feature branch to master branch. So switch to master branch and execute git merge feature

Now master branch will contain 4 files a.txt, b.txt, x.txt, y.txt

Also master branch will have 4 commits c1m, c2m, c1f, c2f

Does feature branch exists after merge operation is completed?

Yes

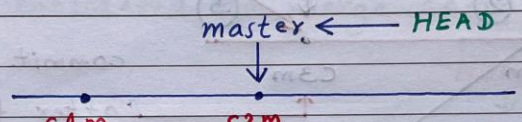


Types of Merges

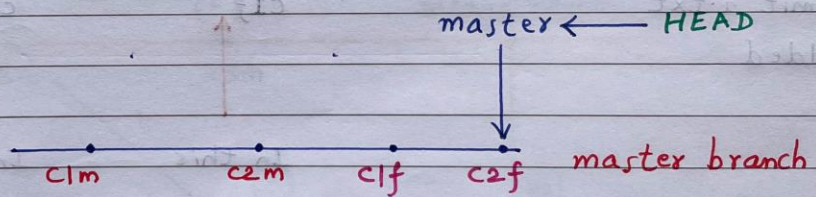
- Fast forward merge
- Three way merge

Fast forward merge : After creating child branch, if we are not doing any changes in parent branch, git will perform fast forward merge. git simply moves parent branch pointer to the last commit of child branch

Before fast forward merge



After fast forward merge



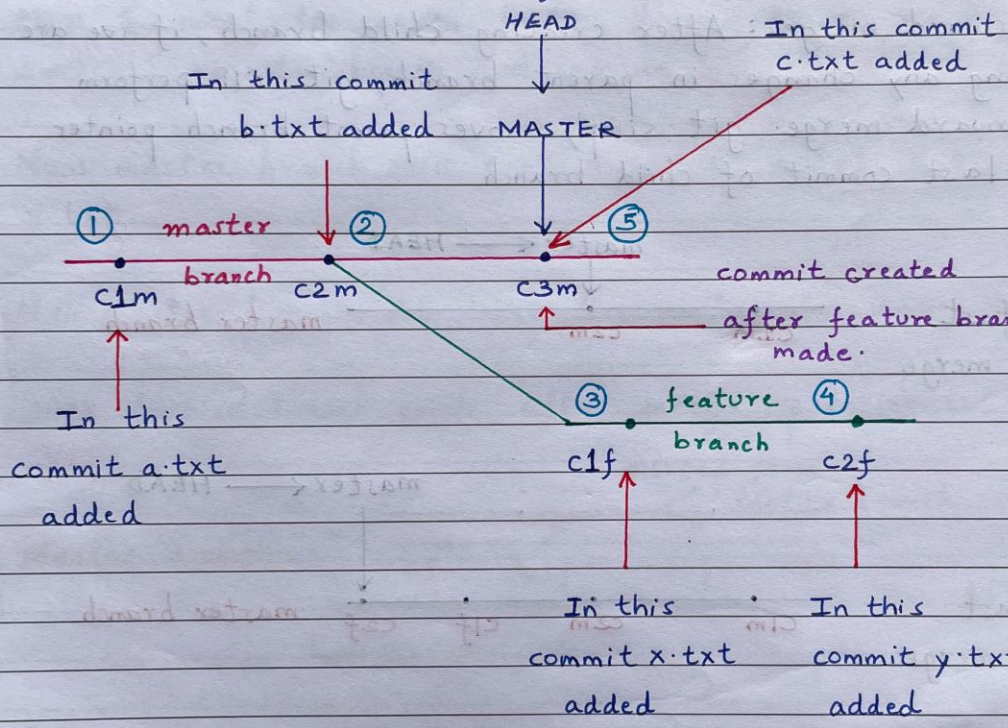
In fast forward merge there is no chance of merge conflicts.

sushilbhosaleazure@gmail.com

After creating child branch if parent branch also contains some new commits then we have to perform 3 way merge.

In 3 way merge may be we can face a scenario where we have conflicts. Conflicts we will see later.

Scenario 1 → Without conflicts

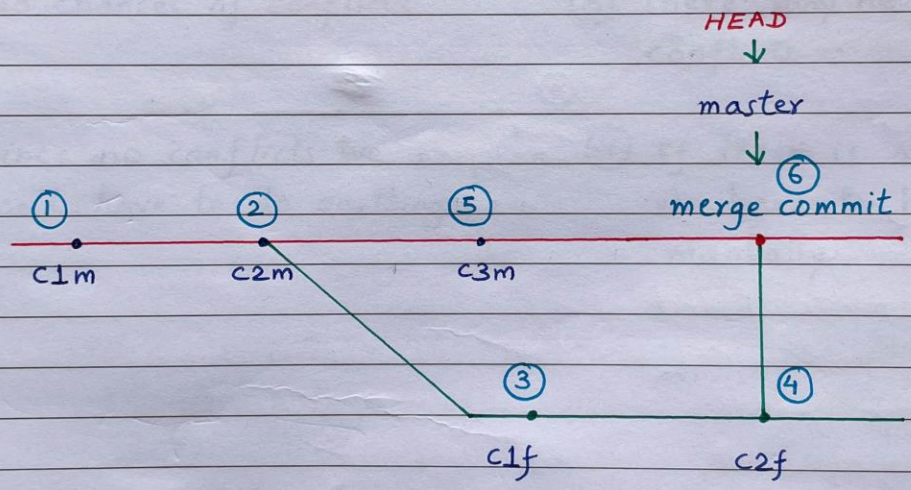


	Feature branch	Master branch
No. of commits	4	2
Commits	c1f, c2f c1m, c2m	c1m, c2m

State when feature branch is created

	Feature branch	Master branch	State after creating feature branch and then creating c3m commit on master branch
No. of commits	4	3	
Commits	c1f, c2f, c1m, c2m	c1m, c2m, c3m	

During merge operation a new commit will be created this commit is considered as merge commit



some refresher

if you execute a command git commit one vi editor will open to provide commit message same way if we execute git merge branchname then one vi editor will open and ask us to perform or add comment.

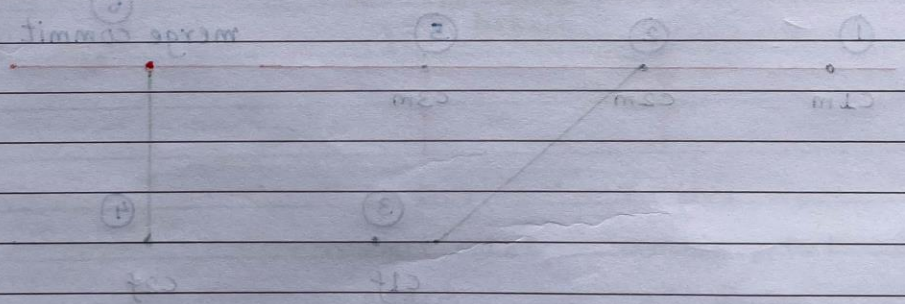
Page No.:
Date:
YOUVA

recursive strategy is nothing but a 3 way merge.

Before merge operation we had 5 commits c1m, c2m, c3m, c1f, c2f

After merge operation we will have 6 commits

git log --oneline --graph



Fast forward merge VS 3-way merge

Fast forward merge	3-way merge
① After creating child branch only child branch is modified but parent branch is not modified.	① After creating child branch both parent and child branches are updated.
② After merging NO new commit is created.	② After merging a new commit will be created which is known as merge commit.
③ No chances of conflicts	③ There may be chance of conflicts.
④ since no conflicts we don't have to do anything.	④ If there is a conflict we have to handle manually.

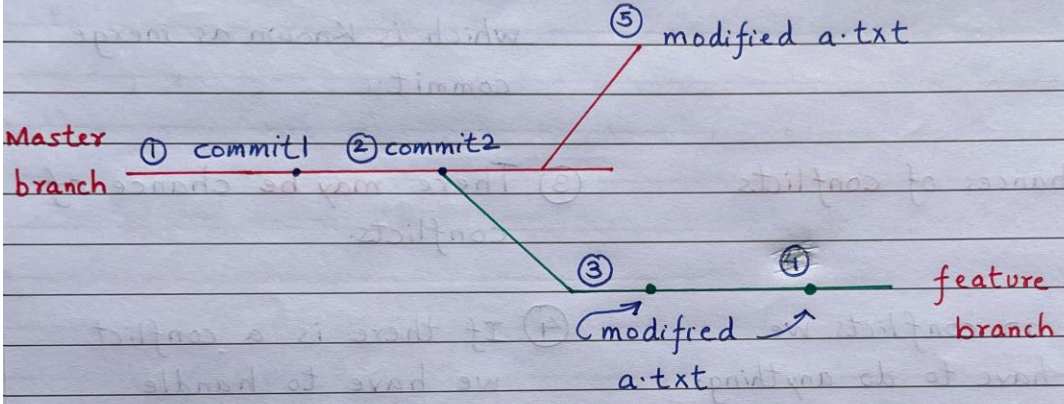
Merge Conflicts.

If same file is modified by both parent and child branches then git halts merge process and raise conflict message.

We have to edit the file manually to finalize content.

Imp

Git will markup both branches content in the file to resolve the conflict very easily.



① Go on master branch

② git merge feature
Auto-merging a.txt
CONFLICT (content): Merge conflict a.txt
Automatic merge failed; fix conflicts and then commit the results

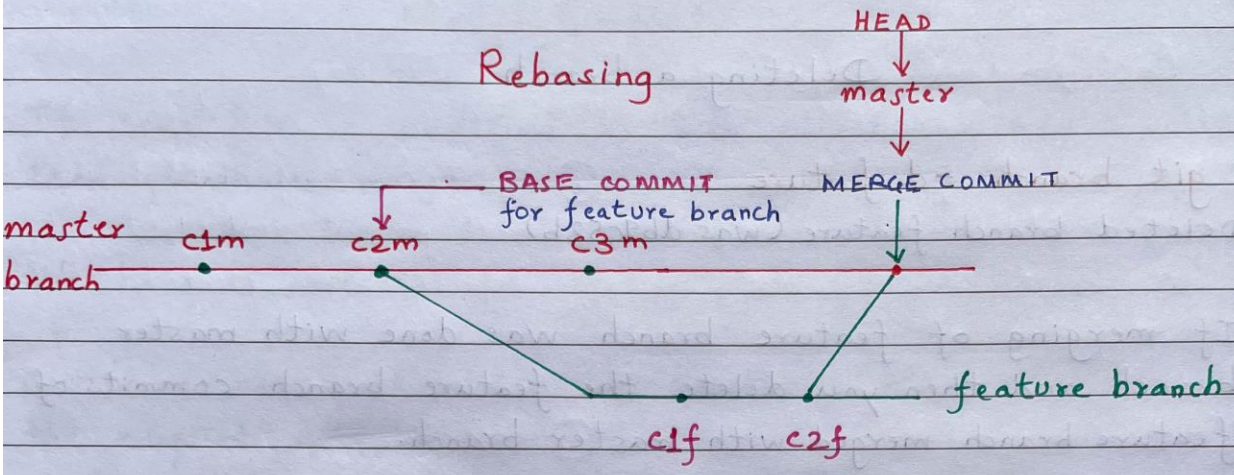
③ git status
on branch master
You have unmerged paths

Deleting a branch

```
git branch -d feature
Deleted branch feature (was db9c52b)
```

If merging of feature branch was done with master branch and then you delete the feature branch commits of feature branch merged with master branch.

Rebasing



For merge commit c3m and c2f are parent commits.

Here we have a non linear graph.

Just for merge operation new commit got created

Rebase operation

→ Linear flow instead of non-linear flow

→ New commits should not be created like merge commit

→ We should not have any conflicts

→ Rebase is alternative to merge operation

Rebase means

↓

Re + Base

↓

Re arrange Base

Merge is a single step process `git merge branchname`

Rebasing is a two step process

1. We have to rebase feature branch on top of master branch
2. We have to merge feature branch into master branch (Fast-forward merge)

STEP1: Rebase feature branch on top of master branch

```
git checkout feature

git rebase master
```

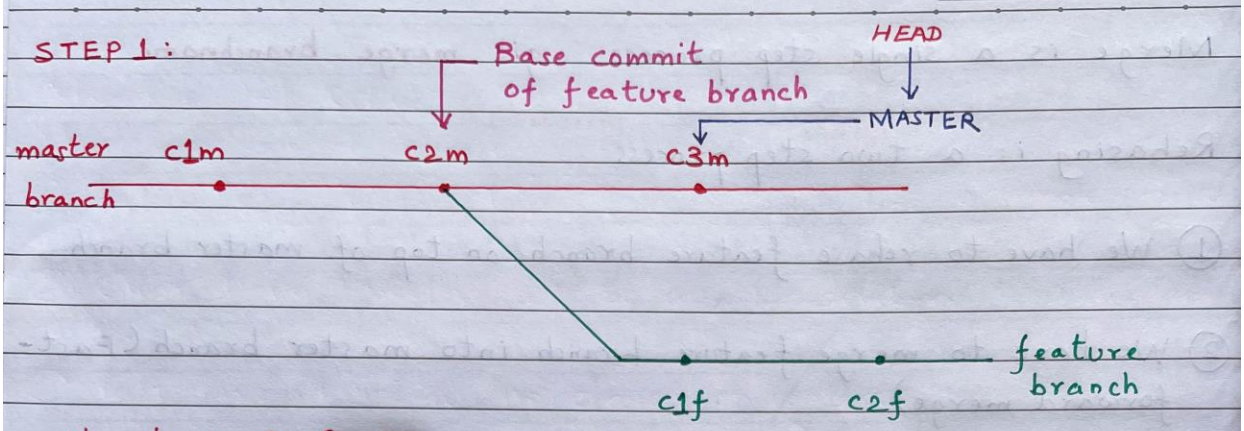
STEP2: We have to merge feature branch into master branch (fast forward merge)

```
git checkout master

git merge feature
```

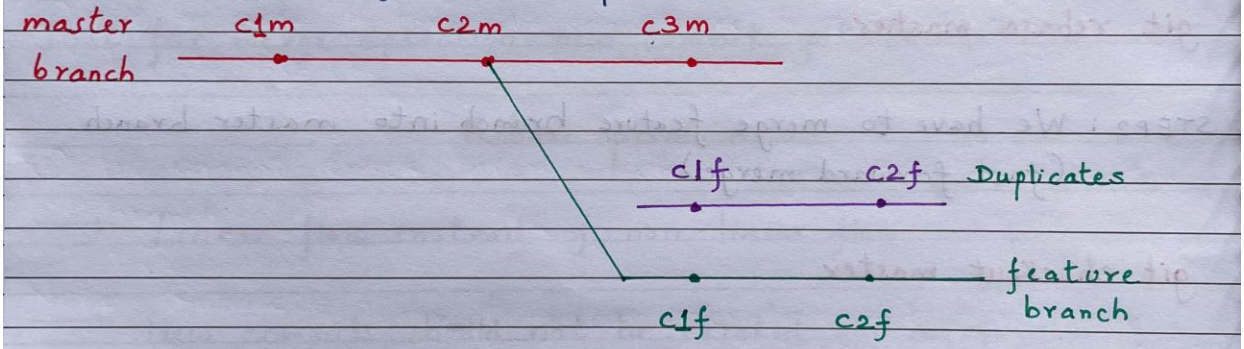
Date:

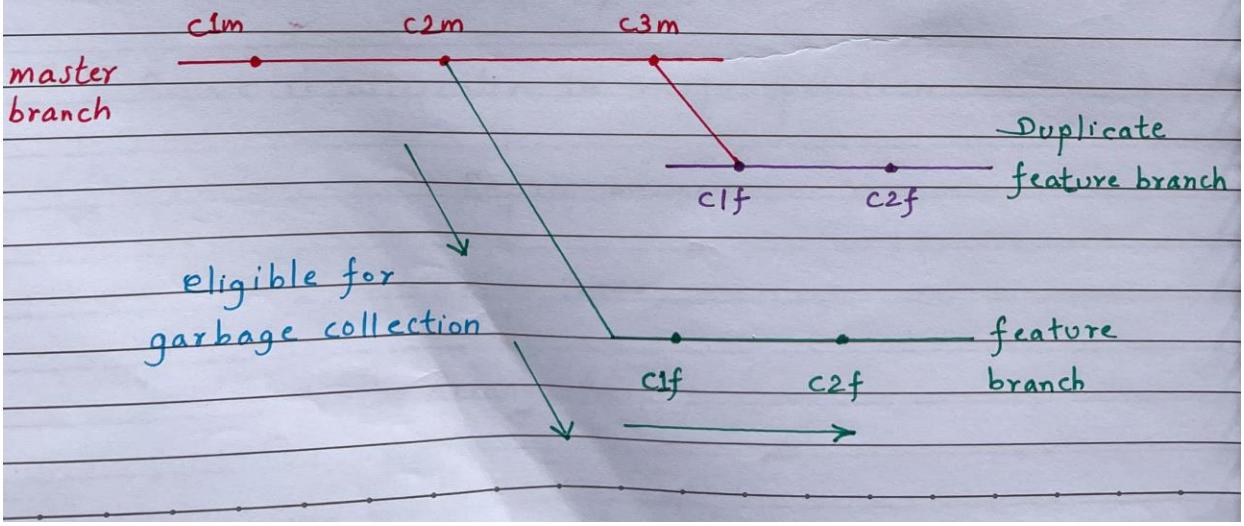
STEP 1:



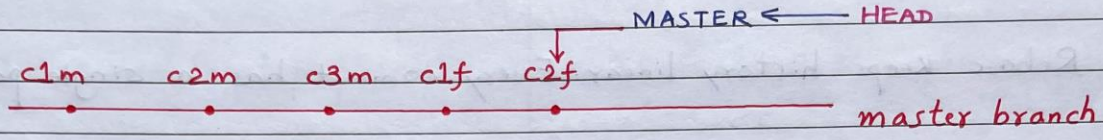
git checkout feature

As soon as we execute git rebase master command duplicate of c1f commit, c2f commit are created only thing that will differ for duplicate is commit ids rest all will be same like commit message, timestamp, author name, mail id.





So now our branch will now be linear branch.



Base commit of feature branch will be updated as last commit of parent branch.

STEP2: git checkout master

git merge feature

NOTE:
There may be a case where c3m and c1f might have differences in that case c1f will become final commit.

Advantages of Rebase

1. Rebase keeps history linear. Every commit has a single parent.
2. Clear linear workflow will be easy for developers to understand.
3. Fast forward merge, no chances of conflicts.
4. No extra commit like merge commit.

Disadvantages of Rebase

1. It rewrites history. We cannot see history of commits from feature branch.
2. We cannot track later which changes came from feature branch

NOTE: Rebase is very dangerous operation and it is never recommended to use on public repositories because it rewrites history.

Merge	Rebase
1. It is single step process. git checkout master git merge feature	1. It is a 2 step process git checkout feature git rebase master git checkout master git merge feature
2. merge preserves history of all commits.	2. Rebase clears history of feature branch.
3. commits can have more than one parent and hence history is non linear.	3. Every commit has a single parent. Hence history is linear.
4. We may have chances of conflicts	4. No chances of conflicts
5. We are aware which changes are coming from feature.	5. We are not aware which changes are coming from feature branch.
6. We can use merge on public repositories.	6. We should not use rebasing on public repositories.

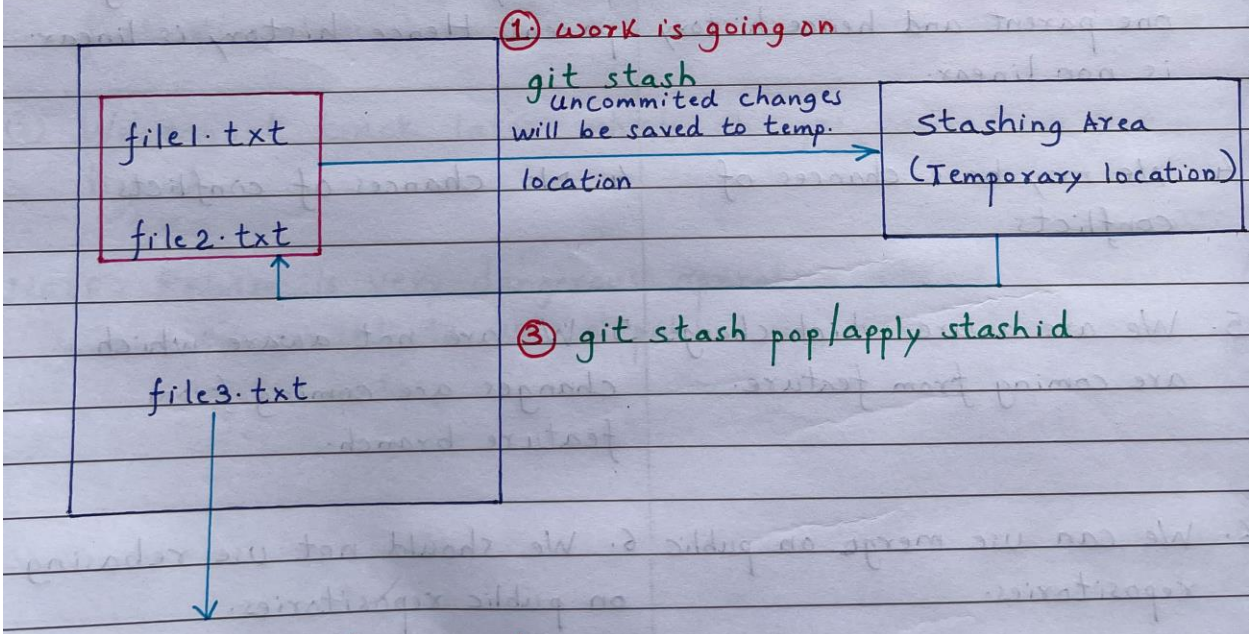
Git Stash

A bit advanced concept in GIT.

stash meaning store safely in a hidden or secret place.

What is git stash?

We have to save uncommited changes from working directory and staging area to some temporary location then we can do our urgent work and commit that urgent work. Working tree will be clean then.
We have to bring back the saved changes and then we can continue the work. (unstash) git stash pop/apply



You can do git stash only on tracked files. Git stash will not work on newly created files.

To perform git stash atleast one commit should be there.

After stashing if you need to check or list number of stashes. use command git stash list

:

stash@{0}

stash id →

git show stash@{0} will show files in stash

There are 2 ways to bring back stashed changes to working directory

① git stash pop <stashid>

② git stash apply <stashid>

git stash pop will bring stashed changes from temporary location to working directory. The corresponding entry will be deleted.

git stash apply will bring stashed changes from temporary location to working directory. The corresponding entry wont be deleted and we can use this stash in other branches.

Delete Stash

We can have multiple stashes. We can delete all stashes or a particular stash.

git stash clear ⟶ To delete all available stashes.

git stash drop ⟶ To delete a particular stash.

Remote repositories

Till now we only referred till local repository. But in projects we have to move code from local repository to remote repository. Once the code is moved to remote repository multiple people can fetch the code from remote repository and store on their own machine so that they can work on latest code for further enhancements.

Push ⟶ sending our code to remote repository

Pull ⟶ getting code from remote repository

Below are some well Known remote repository providers

Github

Gitlab

Bitbucket

Some companies can use their own repository.

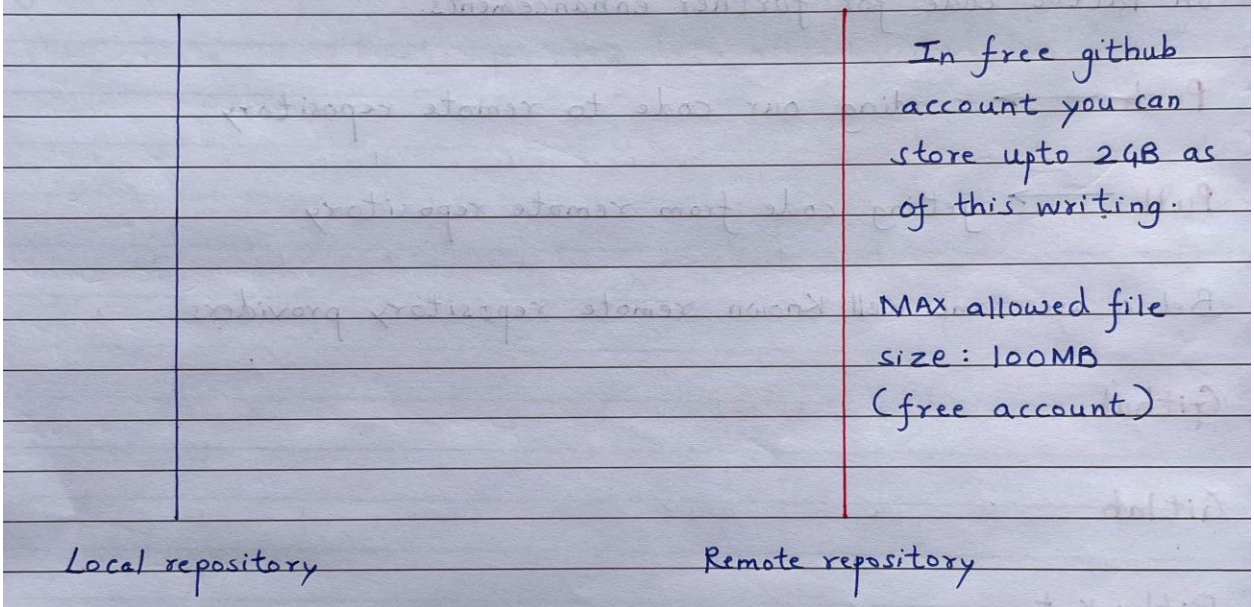
In github we can set access on repository to be public or private.

Public ⟶ Everyone on internet can view files in repository but only selected people can add changes if you have selected the proper access to people.

Private ⟶ Only people having access to read, write can access private repository.

Remote repository name is like

https://github.com/username/remoterepositoryname.git



To send the files from local repository to remote repository we have to configure the remote repository with local repository for that we use command. git remote

git remote add alias-name remote-repository-url

e.g. git remote add origin https://github.com/ ..... /...git

any name you can use but its a proper convention to use so we give origin

To see only the remote repo. aliases use command.

```
git remote
  origin
    :
    :
```

To see remote repo and urls use command

```
git remote -v
  origin   https://github.com/ ..... /...git
```

git push

git push command can be used to push our code from local repository to remote repository.

```
git push <remote_repository_name> <branch>
```

origin

Bydefault we have master branch

local, remote branch

```
git push origin master
```

→ Once you execute this command you have to enter your github credentials.

if origin master already configured use only

```
git push
```

Very Imp.

To perform push operation local repository and remote repository should be in sync.

git clone

A new member joins in team and he needs to download all files from remote repository on his machine then in that case we can use git clone command.

git clone <remote repo url>.

also, create new local repository with all files and history of remote repository. ↑ .git folder

Name of project folder on local machine will be same as remote repository name.

It is also possible to keep my own project folder name?

Yes

git clone <remote repo url> <directory_name>
↑ own project folder name.

Question: Before using git clone command, is it required to use git init command or not?

No, because git clone creates local repository

Question: In how many ways we can create local repository?

1. Empty local repository by using git init command.
2. Non-empty local repository by using git clone command.

git pull

git pull is used to get updates from remote repository into local repository

New team member ⟶ git clone

Existing team member ⟶ git pull
who did git clone before

Interview

git pull = fetch + merge

download from remote repo.

merge changes from remote repository to local repository

git pull origin master

remote repo master branch changes will be updated in local repo master branch

Before doing a pull request switch to branch where you want the files to be pulled.

git checkout master

git pull origin remote-branchname.

git fetch

git fetch command will tell us if there are new updates available in remote repository which are currently not part of local repository.

then do a git pull origin master

If output of git fetch command is empty means no updates are available.

Scenarios

Scenario 1:

Already created before executing commands

Branch	Files
main	a.txt
	b.txt

remote repository

① git clone https:\ .. ... \...git

② git branch
* main

③ git status
working tree clean

④ Make changes in a.txt

⑤ git add a.txt

⑥ git commit -m "Added line in file"

⑦ git status
working tree clean

⑧ git push origin main ⟶ pushes file from local repo main branch to remote repo main branch

Scenario2

Already created before executing commands

① git clone https:\.....\...\...git

② git branch
*main

③ git status
working tree clean

④ git checkout -b IOS

⑤ git branch
*IOS
main

⑥ git pull origin IOS
(pulls c.txt)

⑦ git status
working tree clean

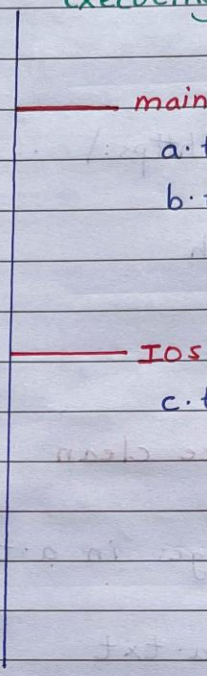
⑧ Make changes in c.txt

⑨ git status
changes not staged to commit

⑩ git add c.txt

⑪ git commit -m "Added additional lines"

⑫ git status
working tree clean.



⑬ git push origin IOS

scenario3

Already created before executing commands

① git clone https:\ .... \ ... \ ... git

② git branch
* IOS

③ git status
working tree clean

④ Make changes in a.txt

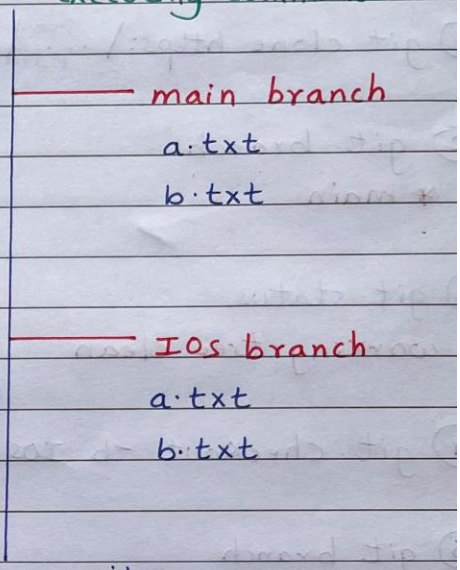
⑤ git status
changes not staged to commit

⑥ git add a.txt

⑦ git commit -m "Added line"

⑧ git status
working tree clean

⑨ git push origin IOS



scenario4

Already created before executing commands

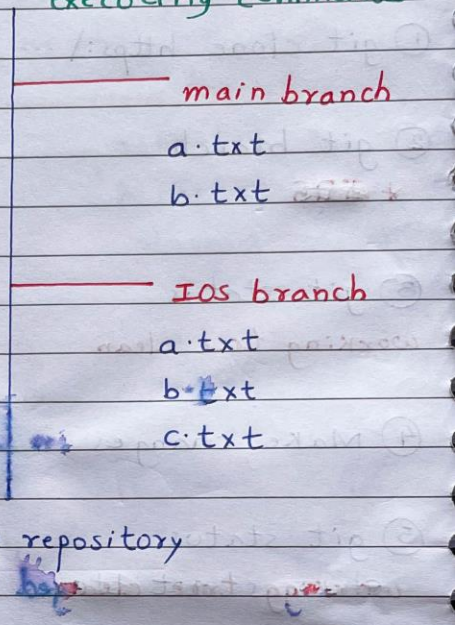
① git clone https:\...\...\...git

② git branch
* main

③ git status
working tree clean

④ git checkout -b IOS

⑤ git branch
* IOS
main



⑥ git pull origin IOS
(pulls a.txt, b.txt, c.txt)

⑦ git status
working tree clean

⑧ Make changes in c.txt

⑨ git status
changes not staged to commit

⑩ git add c.txt

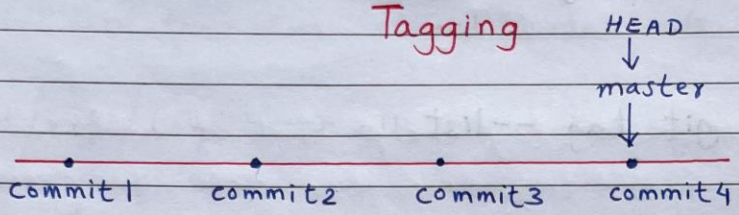
⑪ git commit -m "custom text"

⑫ git status
working tree clean

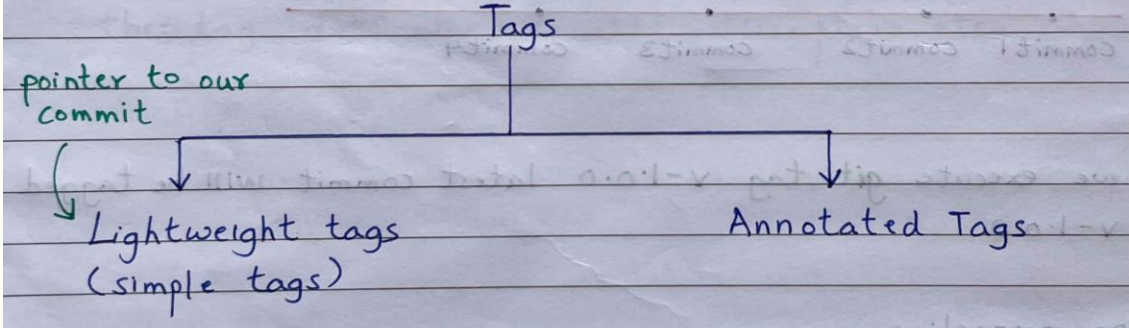
⑬ git push origin IOS

Date:

Tagging



Tag is nothing but a label or mark to particular commit in our repository. Tag is static and permanent reference to a particular commit. In general we can use tags concept for release versions.




```
git tag <tag_name>
```

Bydefault will tag to latest commit

Lightweight Tags

List available tags

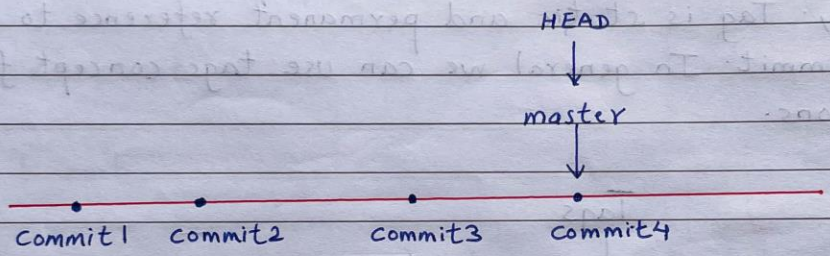
git tag -l or git tag --list

Delete tag

git tag -d <tagname>

or

git tag --delete <tagname>



After we execute git tag v-1.0.0 latest commit will be tagged with v-1.0.0

git log --oneline

2a1c1f4 (HEAD → master) tag v-1.0.0

.
.
.
.

git show 2a1c1f4
or
git show v-1.0.0

} same output

Where are these tags stored?

- .git/refs/tags ⟶ all our lightweight tags will be stored.

```
ls -l
v-1.0.0
```

```
cat v-1.0.0
```

we will see commit ID

Limitation of light weight tags.

1. It won't maintain any extra information like who created this tag, date when created, description etc.

Annotated Tag

Exactly same as lightweight tag except that it holds information like who created tag, description, date when created etc.

```
git tag -a <tagname> -m <tagmessage>
```

A tag object will be created to hold the information mentioned above.

```
git tag -a v-1.1.0 -m 'my custom message'
```

```
git show v-1.1.0
```

```
git tag -v v-1.1.0
```
→ shows object, type, tagger name

Location of annotated tag

`.git/refs/tags` and `.git/objects`

Lightweight tag → if we open the tag file we see the commit ID

Annotated tag → if we open the tag file we do not see commit ID but we see the object ~~ID~~ hash

```
git cat-file -t <hash>
```
tag

git cat-file -p <hash>

object f7ec1 ..... 25af

type commit

tag v-1.1.0

tagger name of tagger

Light weight tag	Annotated tag
① git tag <tagname>	① git tag -a <tagname> -m <tag message>
② It is text label and not implemented as object.	② It is internally implemented as object
③ Located inside .git/refs/tags folder	③ Located inside .git/refs/tags .git/objects
④ Won't store any extra information.	④ It will store extra informatı like tagger name, date of creation, message etc.

How to tag previous commit?

git log --oneline command will output commit IDs.

```
git tag -a v-1.1.0 ce4a690 -m 'My custom comment'
                   ↑
               commit ID
```

How to update tag which is currently there?

① Delete this tag and recreate the tag by pointing to correct commit

```
git tag -d v-1.0.0-beta
git tag -a v-1.0.0-beta correct_commit_id -m 'my message'
```

OR

② By using -f option or --force option to replace an existing tag without deletion.

```
git tag -a v-1.0.0-beta -f commit-id -m 'my message'
```

For same commit can we use multiple tags or not?

Yes. git tag -a Newtag b83ee22 -m 'New comment'

For multiple commits, is it possible to use same tag?

No

NOTE: Within repository tag name should be unique.

Push tag to remote repository

git tag -a v-1.0.0 1637f5f -m 'Release 1.0.0'

Add remote repository use below command.

git remote add origin https://github.com/...../....git

origin ← alias for; https ← alias for

git push origin master → By default push command won't push tags to remote repository

on github go in repository → code → releases
you won't see any releases, tags

How to push a single tag?

git push origin <tagname>

e.g. git push origin v-1.0.0

now in github repo releases you will see tag. You will see .zip, tar.gz which has files pushed in this commit

.zip ← for windows; tar.gz ← for linux

How to push all tags?

```
git push origin master --tags
```

Deleting tags

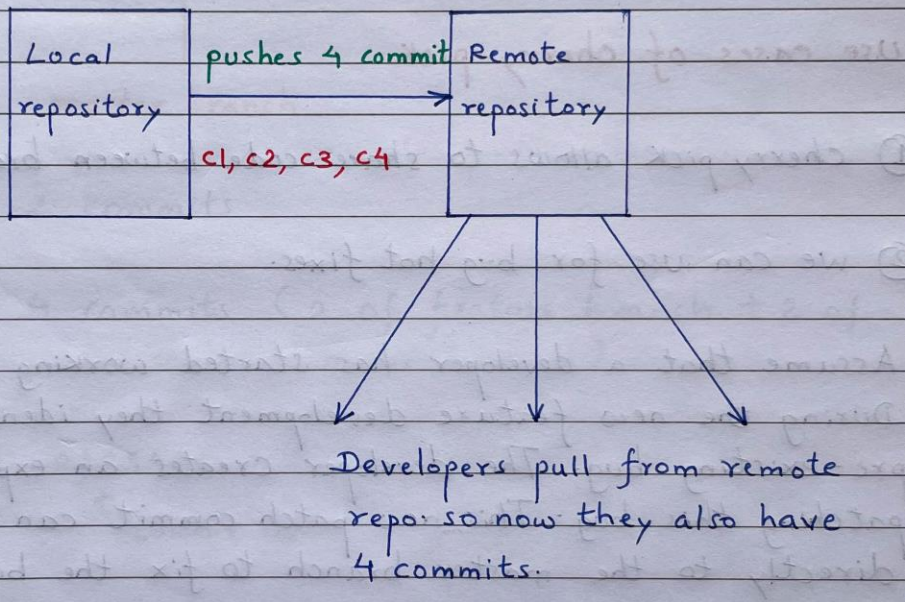
local repo : `git tag -d v-1.1.0`

remote repo : `git push origin : v-1.1.0`

Git revert

some background: git reset is destructive command and hence it is not recommended to use on public repositorres because it deletes commit history.

git revert command won't delete commit history



git revert c3 ⟶ undo changes of c3. (Means we go back to c2 changes)

so now with c2 changes a new commit will be created without deleting c3, c4 or any commit history.

c2 and c2-c (new commit will have same changes)

git cherry picking

Assume that we have two branches master, feature. If we merge feature branch to master branch, then all commits & the corresponding changes will be added to master branch.

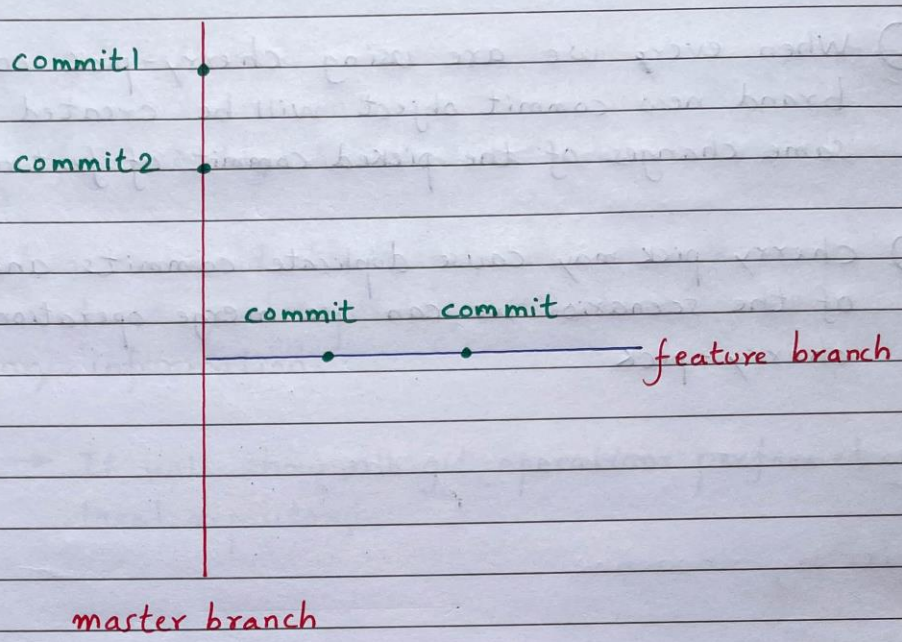
Insted of all commits, we can pick an arbitrary commit of feature branch and we can append that commit to master branch. It is possible by using cherry-pick command.

Use cases of cherry-pick

① cherry pick allows to share code between branches.

② we can use for bug hot fixes.

Assume that a developer has started working on a new feature. During the new feature development they identify a pre-existing bug. The developer creates an explicit commit patching this bug. This new patch commit can be cherry-picked directly to the master branch to fix the bug before it effects more users



master branch 2 commits

feature branch 4 commits (2 of feature branch + 2 of main branch)

git cherry-pick 2f8545e

↑ commit id

↑ Execute this from master branch

A new commit will be created in master branch

(1) When ever we are using cherry-pick command, a brand new commit object will be created but with same changes of the picked commit of feature branch.

(2) cherry pick may cause duplicate commits and hence in most of the scenarios we can use merge operation instead of cherry-pick

git reflog

reflog means reference log.

We can use reflog command to display all git operations which we performed on local repository. We won't get remote repository information.

git reflog ⟶ It will show all git operations performed on local repository.

git reflog <branch name> ⟶ It will show all git operations performed on local repository related to particular branch

By mistake if we did any unwanted destruction operation like git reset --hard still we can recover by using git reflog command.

git log --oneline

48be5d1 (HEAD → master) fourth commit
- 3rd commit
- 2nd commit
- 1st commit

git reflog

48be5d1 (HEAD → master) HEAD@{0} : fourth commit
. HEAD@{1}
. " {2}
f227c9f " {3}

git reset --hard f227c9f

git log --oneline

f227c9f (HEAD → master)

Now if you execute git reflog you will get all history of commits