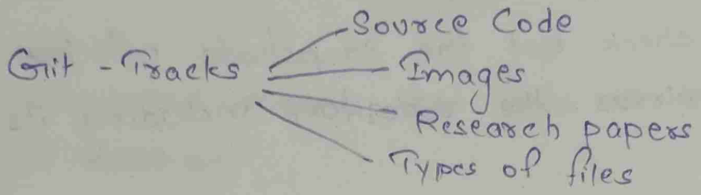
Git:

→ Git is a Version Control System also known as VCS, in websites it will show as distrubuted Version Control Conte System.

→ GIT Software is developed by the Linus Torvalds, who developed the Linux, created git for development of linux kernel for the contribution of the other kernel developers.

→ Basically VCS. is the software designed to record changes made to the file over time.

→ Git gives us ability to revert the files or the set of files you made changes.



Types of Version Control System?

1. Local Version Control System

→ This method is to copy files into another directory.
keep files - with time stamp

→ This is very simple.

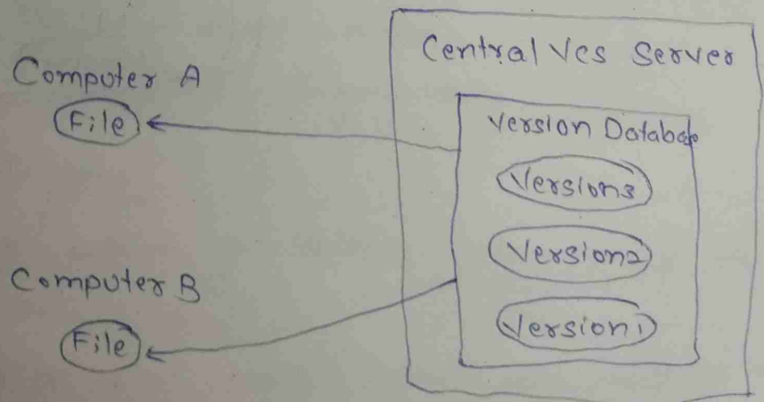
2. Centralized Version System:

Developed - inorde to collabarate with other people

→ This has Single Server - all versioned files - no:of clients can take the files from that central place.

→ For many years this is the standard for Version Control.

Ex: Subversion, Perforce



Administrator - has full control - who can do what

→ Everyone get the project update immediatly as all are linked to the one centralized server.

Centralized Server goes down - nobody can do anything

→ If harddisk get currupted then entire data get lost, same problem in the Local VCS when we keep all the data at one place.

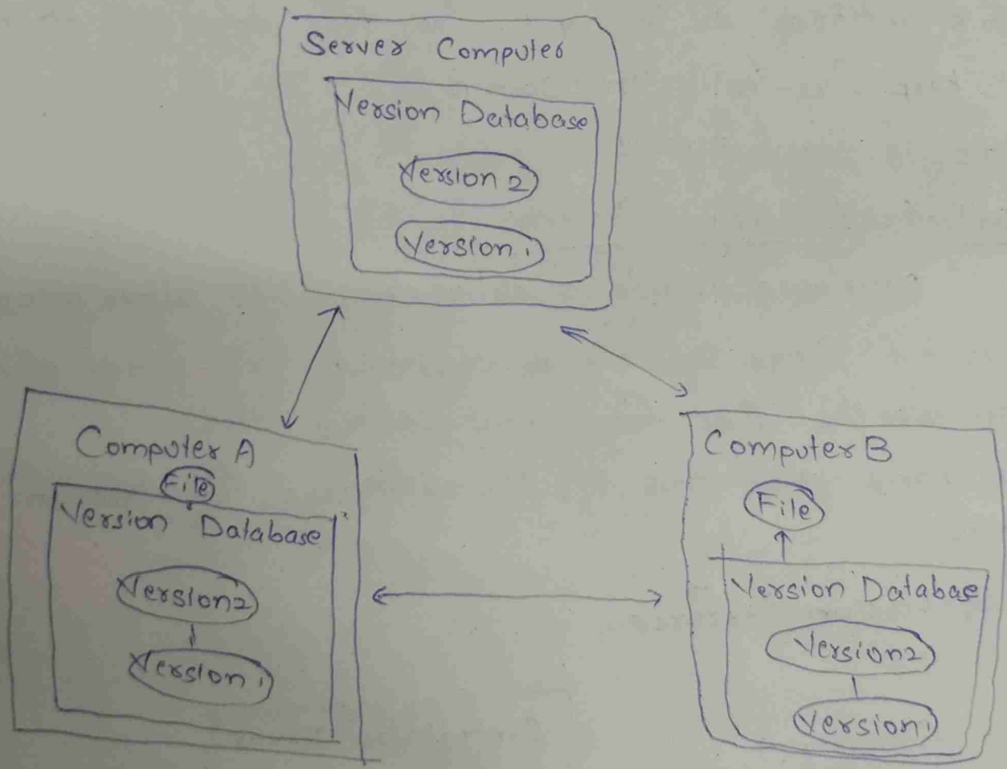
→ In DVCS he get only one snapshot at a time either version 1 or 2 or 3.

3. <u>Distributed Version Control System</u>:

→ Clients do not check out the snapshots of the files, they can also fully mirror the repository including its full history.

→ In Client system also we will maintain version so when the server is up you can fully copied back to the server.

→ Every clone is full backup of all data.



Git

Git - most operations - requires local files & local resources

→ There is no network latency (which will be there in CVCS)

Stages in Git:

1. Modified:

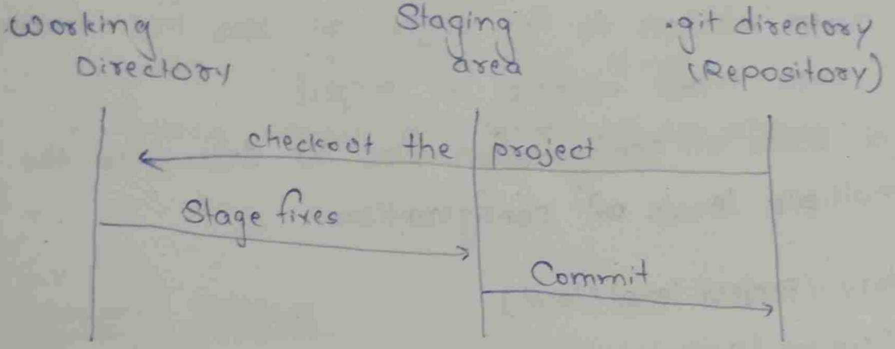
Modified means you have changed the but but not commited to the database yet.

2. Staged:

Staged means you have marked a modified file in its current version to go into your next commit Snapshot.

3. Commited:

Commited means that data is safely stored in your local database.



Working Directory:

The Working directory (tree) is a single checkout of one version of the project, these are placed on the disc which we' will use for modify.

Staging area:

Stating area is a file in git folder (directory) that store information what to go in the next commit.

Technical name - index

phase - Staging area

Repository area:

Place where git stoore - meta data & object database for our project.

This is what copied when we clone a repository from another compt.

Git Workflow:

1. You Modify the files in your working tree
2. You select only the files which to be in next commit.
3. You do a Commit, files in staging area, stored in Git repository permanently.

Different ways to use Git:

1. Using Command Line Tool
2. Using Graphical User Interface

git --version → To get version of git installed in your computer.

Git Configuration:

→ After installing git, we need to set the username and the email address.

→ When we make changes to the project, git uses this information to identify who made changes in project.

→ Git uses a series of configuration files to determine the behaviour git has multiple levels of configuration.

1. Repository / Project level (local)
2. User Account (Global level)
3. System level (Git installation)

Priority

Repository leval > User Account leval > System leval

Git Config locations:

local (Repository / Project leval)

repository /.git /config

Global (User leval)

C:/ Users /akula /.gitconfig

System (Git Installation)

C:/program Files / Git/etc/ gitconfig

→ Check complete git config

```
git config --list --show-origin
```

→ Remove specific setting for specific level of config

```
git config --global --unset user name
```

→ Remove the specific section

```
git config --global --remove-section user
```

→ To initialize the Empty git repository

```
git init
```

(add .git folder into our project)

→ To get user name and email

```
git config user.name
git config user.email
```

→ To view the hidden file in command prompt

```
ls -a
```

→ To view files and hidden files

```
ls
```

→ To view files

→ To change the email in the local level

```
git config --local user.email akulavishnuvardhanrdy@gmail.com
```

<u>In Command prompt:</u>

→ mkdir Git_Learning	(To make new folder)
→ cls	(To clear all screen on cmd)
→ dir	(To view all the files in directory)
→ echo .> index.html	(To create a new file)
→ dir /a	(To view list of normal & hidden files)
→ rmdir /s Git_Learning	(To remove the folder /s - delete files and subdirectories)
→ type temp.txt	(To see content of file)
→ echo Vishnu is learning .> temp.txt	(To add content in file)
→ del temp.txt	(To delete a file)
→ ren temp.txt index.txt	(Rename a file)
→ q	(To get out of a list in cmd)
→ i	(To go to insert mode)
→ Esc	(To go bottom)
→ :wq	(Write and quit)

→ git help (Show list of all commonly used commands)

→ git help -a (Show list of all the commands in the git)
git help --all

→ git help <Command-name> (Shows the details about the command in the browser)

→ When we initialize an empty repository automatically one branch will be created as a main branch master.

→ git status (It will show we are in which branch and other details about status of the file)

→ git add file-name (To add file into Staged area from working area)

→ git rm --cached index.html (To remove file from staged area and move to working area)

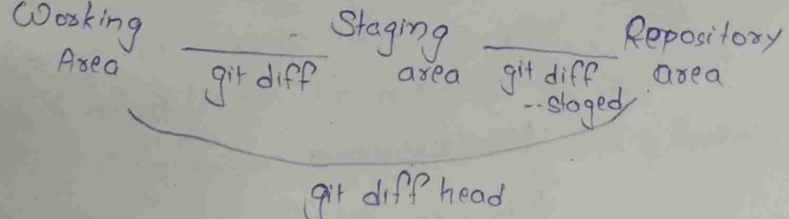
→ git commit -m "Initial_commit" (To move from Staging area to git repository)

→ git log (To get all the commit history)

Git Diff Command:

→ Diff command is used to track the difference between the changes on the file.

→ Diff command takes 2 inputs and reflect the difference between them.



How GIT Stores Data:

→ Git stores the data in form of keys and values
Values - contents of files
keys - calculate the key for value using SHA1, its nothing but hash value.

→ SHA1 is 20 byte Hexadecimal Format

→ Not only files, directories and so on commits has their own SHA1.

→ Every object in git has their own SHA1

→ To get what is present inside the hash use command

```
git cat-file <<hashcode>> -p
```

→ The details about all these is present in objects folder in .git repository.

Renaming & Restoring files in Git:

→ First if you rename the file it will say 1 file deleted one file added, but when we add these files to the staging area it will compare 2 files and realise we just renamed the file name it will show file is renamed.

→ If we use this command

```
git mv channel.txt temp.txt
```

it will automatically move to staging area insted of showing in working area.

→ To restore what changes we have made on a file in working area from last commit

```
git restore temp.txt              (restore changes in working area)
git restore --staged temp.txt     (restore from staging area
                                   to working area)
```

Git Branching:

→ Default branch in git is master

→ Where our current pointer reference is pointing is head.

→ To create a new branch

```
git checkout -b loginfeature
```

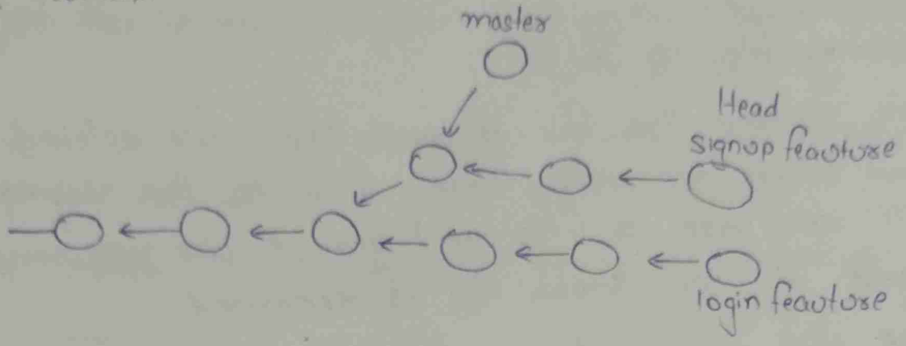
→ To move head to another branch

```
git checkout master
```

Series of commands & diagram

```
git commit
git checkout -b loginfeature
git commit
git commit
git checkout master
git commit
git checkout -b signupfeature
git commit
git checkout master
git commit
```

git checkout loginfeauture
git commit
git checkout signupfeauture
git commit



→ To add all the files from working area to staging are

git add .

→ To get the clean and neat logs in one line

git log --oneline

→ To see list of all branches

git branch

→ To see the branches present in repo go to .

.git > refs > heads

→ To create a branch without changing head position

git branch dummyBranch

→ The data which is present in the branch file is hash object id, it will be present in objects

→ Git will able to which branch it is present by head file in git

.git\HEAD (ref: refs/heads/master)

→ To rename the branch, it will change current head branch name

git branch -m new-branch

To delete a branch, the head should not be in that branch, and there should not be any merges

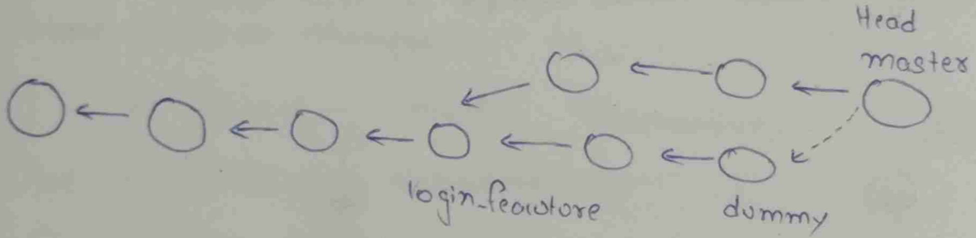
git branch -d new-branch

git branch -D new-branch (will delete branch even the branch is not fully merged)

Git Merging:

Example series of code

```
git commit
git checkout -b login-feauture
git commit
git commit
git checkout master
git merge login-feauture    (fast forward merge)
git checkout -b dummy
git commit
git commit
git checkout master
git commit
git commit
git merge dummy    (reccursive method approch)
```



→ To merge two branches, be in master branch and
git merge dummy

→ When there are changes only on one branch then no new commit will be made it is a fast forward merge.

→ When there are commits or changes in both branches a new commit will be made while merging both branches, it is reccursive method approch

Merge Conflicts:

→ When we try to merge the branches which has changed or modified the same file, then git will not be able to know which branch code should be considered so automerging get failed.

→ After that git will change the code in that file it will show the modified code by both the branches and we need to select which branch code should be considered and need to remove another code and commit the changes.

Git Rebase:

→ Rebase is an alternative to merging.

→ Rebaseing a branch updates one branch with another by applyin the commits of one branch on the top of commits of another branch.

→ Rebase is an advanced command which is used rarely.

→ Merge preserves history, Rebase does not preserve history.

Donot use Rebase when

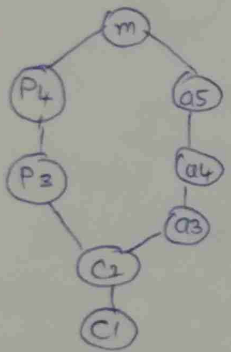
→ The branch is public when it is shared to all the developers

→ Most of the teams prefer to use merge over rebase.

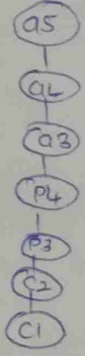
Common places when rebase is used

→ Cleaning up all the commits before sharing your branch.

→ Pulling changes from another branch without merge.



Merge
(we can know which branch made this commit)



Rebase
(We cannot know which branch made this commit, all appers to be from branch)

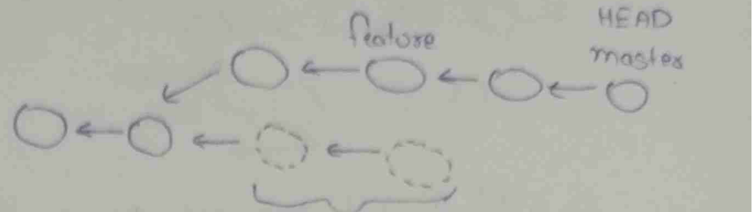
When we have a doubt - merge
when we do not have a merge
If you know what you are doing - rebase

→ To do rebase between two branches, place HEAD in master
git rebase feature

→ When we rebase master branch commits will come after the another branch commits

Example Series of code

```
git commit
git branch feature
git commit
git commit
git checkout feature
git commit
git commit
git checkout master
git rebase feature
```



Commits made by master copied next to feature after Some time this will be deleted.

→ To get the graph of the commit history use

```
git log --graph
git log --oneline --graph
```

→ When we have so many commit which we want to change to a single commit. go ~~to the commit from where You need to make~~ changes (or)

```
git rebase -i master   (Interactive)
```

This is interactive rebasing.

Modify or change the latest commit:

→ If we want to modify or change some files and we dont want to make it new commit we can modify the previous commit

→ Add the files which we want into the staging areas insted of

```
git commit -m "commit"
```

Use

```
git commit --amend
```

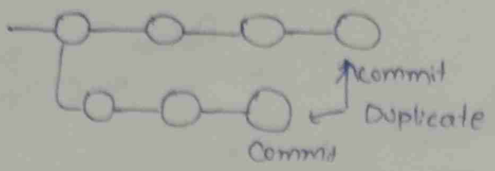
Git Cherry Pick:

→ Cherry pick is mainly used if you don't want to merge the whole branch and you want some of the commits

→ It is an advanced concept and also a powerfull command.

→ Cherry pick is a usefull tool, but always it is not a good opinion, it can cause duplicate commits.

→ Mainly cherry pick is used for the bug fixes where you want to place that bugfix commit in all version branches.



```
git cherry-pick <<hash code of commit>>
```

→ Be in the branch where we need to add the commit and type the command.

Git Head:

→ when we are working with branches we can checkout only one branch, it is Head brand.

→ Git makes a note of this branch and stores it in git>Head as the reference for the path of the branch.

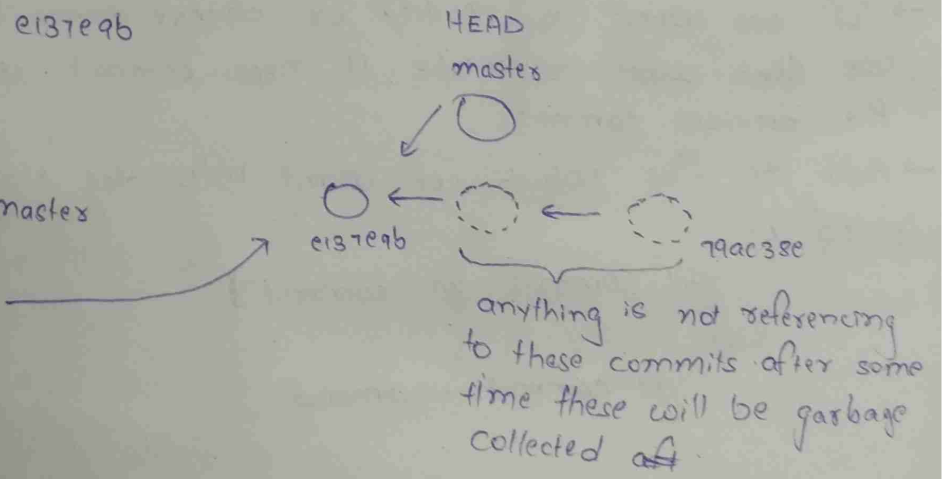
→ Head not only reference a branch it also reference the commit SHA1.

→ If Head points to a specific commit then it is called as detached head.

```
git checkout <<Commit hash code>>
```

Series of commits

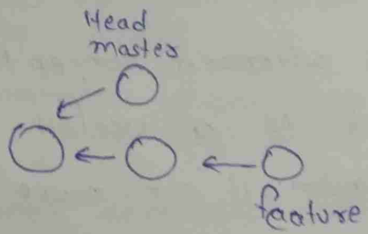
```
git checkout e137e9b
git commit
git commit
git checkout master
git commit
```



HEAD
master
e137e9b
79ac38e
anything is not referencing to these commits after some time these will be garbage collected ~~aft~~

→ To make them not garbage collected we need to remember the last commit hash.

```
git checkout 79ace38e.
git branch feature
git checkout master
```



Git reset:

→ Reset does different things in different contexts

we know that if we want to move the branch we use

1. commit
2. merge
3. rebase

→ By carefull observation these are not explicitly used for moving branch, they as a side effect of creating new branch.

→ Reset is a command that specifilly used for moving a branch. (from one commit to another)

Checkout - will move head from one commit to another

reset - will move branch from one commit to another

Reset options

--hard (moves the files to both working & staging area)
git repo ↓ (over "files")

--mixed (default, moves files only to stage area)

--soft (does not move files)

→ If we want to undo the commit we use reset command

```
git reset <<commit hash code>>
```

→ Reset can also be used to move the files from staging area to working area

```
git reset head --mixed
```

→ To remove all the files of recent changes in both staging and working area.

```
git reset head --hard
```

Git Stash:

→ Normally if we want to switch the branch we want to commit the code and switch to the new branch.

→ Git will not allow to switch the branch when we made changes on a branch, it will ask to commit or stash the changes.

→ The git stash command enables you to switch branches without comming the current branch.

Stash - store something safely in a hidden place.

If we switch branch without commiting changes.

1. Switches to the branch carrying the changes (when is no modifical of current files in that branch)
2. Git will not allow to switch the branch and asks to commit or stash the changes.

```
git stash
```

→ To see the wheather stash happened or not

```
git stash list
```

→ To get the changes we have made from the stash

`git stash pop` (It will take recently stashed and give it to the branch & remove it from stash)

→ We we write

`git stash` (It will take previous commit as the stash name)

`git stash save "modified index"`

`git stash apply` (It will apply recently stashed and not remov from the stash)

`git stash apply stash@{1}`
↓
Stash id

→ To see what changes present in the hash

`git ~~stash~~ stash show` → recent stash changes

`git stash show -p` → shows the changes also

`git stash show stash@{0} -p`

→ To delete a stash

`git stash drop` → recent one deleted

`git stash drop stash@{1}`

→ To delete entire stash

```
git stash clear
```

→ To make the stash changes in the ~~recent~~ new branch

`git stash branch new-branch` → recent stash

`git stash branch new-branch stash@{0}`

Git checkout:

1. Moves from one branch to another.
2. Creates new branch if not existed and moves the head to that branch.
3. Also not only branch it also shifts to particular commit hash.

```
git checkout <commit-hash>
```

Then you are in Detached Head

```
git checkout -
```

(This will move to the previous position or stage)

```
git checkout Head~2
```

(go to 2 commits behind the head)

→ To modify only one file up to the previous commit

```
git checkout head index.txt
        (or)
git Checkout -- index.txt
```

Git Switch: & Restore:

→ Git Switch works similarly like checkout but cannot do all the functionalities of checkout

```
git switch master
git switch -c new-branch
```

→ We cannot go to a particular commit using switch command.

→ To ~~move~~ remove a file from Staged area to working area

```
git restore --staged index.txt
```

→ To ~~move~~ remove the modified changes in working area

```
git restore index.txt
```

→ To change a file from present to a particular commit

```
git restore --source head~2 index.txt
```

to Get the changes back to normal

```
git restore index.html
```

Git Revert:

→ When we use reset commit we do not know the commits we undo, but using revert it will make a new commit by removing the commit changes.

```
git revert <<Commit hash>>
```

→ Revert command is used when our commit went to the remote.

Github:

→ Github is the hosting platform for the git repositories
→ Github allows us to share or host our git repository in the cloud.
→ We can access the code from anywhere and also share the code to the people around the world.

Git vs Github:

→ Git is version control System that runs locally on ~~my~~ machine There is no need to register an account, no internet is needed.

→ Github is a service that host repositories in the cloud makes it easier to collaborate with others, we need to signin in github

→ There are so many tools that provide similar hosting and collaboration features. (alternaties to github)

Gitlab
Bitbucket
Gerrit

Github
- basic Services free
- 2008
- World's largest host of Source code
- 60 million user
- 200 million repositories

→ The basic free tier allows for unlimited public and private repos and unlimited collaborators and more.
→ While it offers paid team and enterprise tiers

More than one person doing the project - github easier

→ If you are planning to contribute to the open source projects you have to be comfortable working with github.

Main uses of Github:

1. Collaboration
2. Open Source projects
3. Exposure
4. Stay up to date

Git Cloning:

→ Git Clone gets the repository that is not present in your machine based on the url we provide.

```
git clone <url>
```

→ Git will retrive all the files associated with the repository and will copy into the local machine, git also initializes a new repository will all the history from the cloned project.

→ Anyone can clone the repository from github, provided the repo is public.

→ Pushing up changes to the github repo, you need permissions for that.

Git Clone - Standard git command
- not tied to github we can use to clone repositories that are hosted anywhere.

Github with SSH:

we connect to the github using two methods

1. Using HTTPS
2. Using SSH

→ An SSH key is an alternative way to identify yourself that doesnt require you to enter your username and password every time.

→ Using SSH protocol, you can connect and authenticate to remote servers and services

→ With SSH keys, you can connect to GitHub without supplying Username and ~~password~~ personal access token at each visit.

→ When we are using HTTPS it will always ask for the usernam and password wheather the user is authenticated or not.

→ When you are working with a repository (Github), you'll often need to identify yourself to Github using username and password.

→ An SSH key is alternative way where we no need to enter Username & password.

<u>To implement SSH keys in the System</u>

1. We need to create the SSH key pair using

```
ssh-keygen -t ed25519 -C "akulavishnuvardhanrdy@gmail.com"
```

2. We need to add SSH key to SSH agent

To Ensure wheather ssh-agent is running or not

```
eval `ssh-agent -s'
```

3. Next add SSH key to SSH agent

```
ssh-add ~/.ssd/id_ed25519
```

4. We need to go to github profile and add SSH key pair in the settings

5. we can check we are authenticated to the github or not by using

```
Ssh -T git@github.com
```

<u>Creating a repo in Github</u>

If you already have a local repository we want to get it to github.

1. Create a new repo on github
2. Connect your local repository (add a remote)
3. Push up your changes to github.

If you donot have any local repo

1. Create new repo in github
2. Clone it in your local machine
3. Do some work locally
4. Push your changes to github

git remote add origin ..... (To make linkage with github repo)

git remote (To check local repo is linked with any github repo)

git remote -v (To see the url of repo)

Viewing remote repositories:

→ To view existing remotes for your repository

```
git remote
git remote -v
```

→ This command just displays a list of remotes. If you havent added any remotes yet, you wont see anything.

Adding remote:

```
git remote add origin <url>
```

origin is the short name for url

→ That means whenever i am using the name origin, i am refering to the particular github url like an alias name.

→ The name origin is a conventional git remote name, it is not all special.

When we Clone → Default remote name setup for us is origin → we can change it → But most of the people do not change it

→ How we have master as default branch just like origin is default, if needed we can change the name

We can also we

```
git add remote mygithuburl <url>
```

mygithuburl → refer to url

→ If we want to rename the remote

```
git remote rename <old-name> <new-name>
```

→ We can remove remote using

```
git remote remove <name>
```

→ In late 2020 in github they renamed the master branch as main

```
git push origin master
```

→ We can push only one branch at a time to the github. only that branch will be available in remote.

Git push:

→ When you try to push changes to github using

git push origin master

→ we are creating a branch in github and pushing the charges to that branch.

→ Not only to master we can also push the changes in the master or any branch to the different branch in github

git push <remote> <local-branch>:<remote-branch>

→ This type of pushing is not at all common, but we need to know we can also push the changes to the different remote branch.

git push -u origin master

-u ⇒ (Upstream) Running this command sets the upstream of the local master branch so that it tracks the master branch on the origin repo

→ Once we set up the upstream for a branch, we can use the git push shorthand which will push our current branch to upstream.

git branch -M main (Used to make master branch to rename as main branch)

→ To check the branches present in the remote

~~git remote~~
git branch -r

→ To move the commit (latest) in the remote repo

git checkout origin/main

→ To move the head to the previous position where the head is

git switch -

→ when ever we will clone a repository it will clone only the default branch

git switch branch1

→ It will check local branch (branch1) is present or not
→ If not it will check remote branches if it is present then a new branch will be created locally and point to the branch1 in the remote
→ If it is not present in remote also then it will give error.

In remote branches
1. main (default)
2. branch1
3. branch2
4. branch3

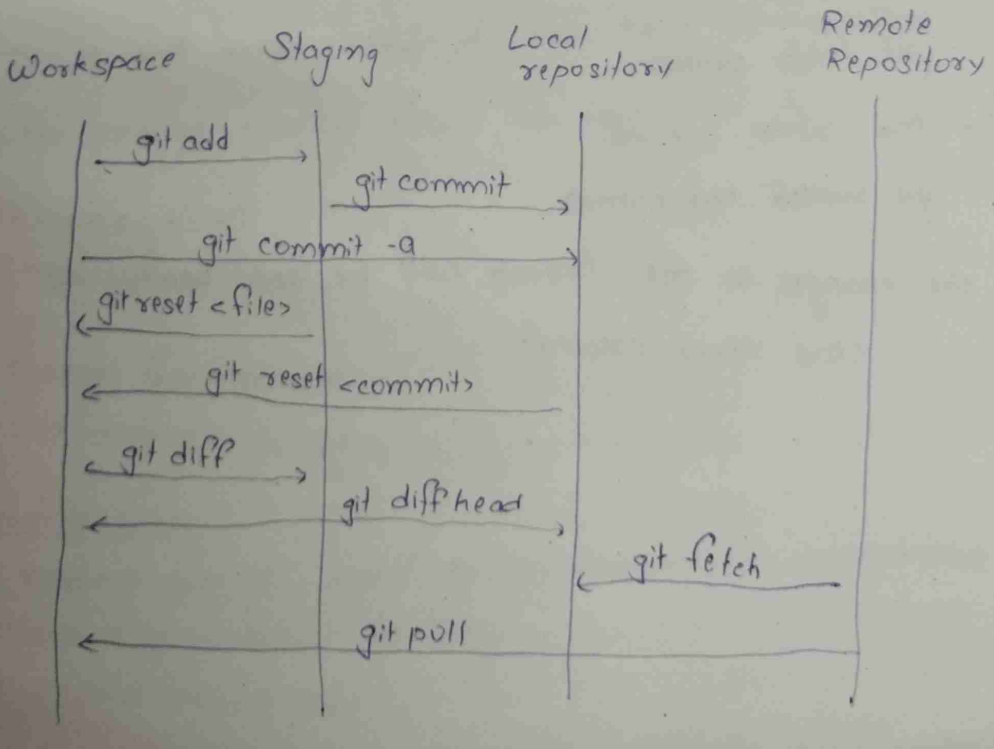
clone →

In local branch
1. main (defalut)

If we use command
git switch branch2
then branch2 will come into local and get sync with remote branch2

Git Fetch:

→ When we are working with other collaborators, one of your teamate push the changes to the master branch, but my local repo doesn't know about it.
→ git fetch and pull get those changes from the github repo to your local repo.



→ Fetching allows us to dowload changes from remote repository.

→ But those changes will not be automatically integrated to our working files.

→ It just lets you see what others have been working in, without merging those changes into your local repo.

git fetch ⇒ Please go and get the latest information from Github but dont add it into my working directory.

```
git fetch <remote>
```

→ git fetch origin will fetch all changes from the origin remote repository.

→ If <remote> is not specified it defaults to origin

→ we can also fetch a specific branch from a remote using the command

```
git fetch <remote> <branch>
```

→ By using git fetch our local git will able to get know changes about the remote repo

→ To know the changes what has made

```
git checkout origin/main
```

Edge case:

→ If we have any changes in the remote branch, which branch is not present locally

```
git fetch origin
```

→ Then after that when we do

```
git switch new-branch
```

→ Then all the changes in the branch will be get locally as the branch is now newly created.

Git pull:

→ Git pull is the command used to retrive the changes from the remote repository.

git pull ⇒ go and dowload data from github and immediately update my local repo with those changes.

git pull = git fetch + git merge

```
git pull <remote> <branch>
```

→ What really matters is where we are on. Whatever the branch i am in that is where the changes will be merged, where i am pulling down to.

```
git pull orign master
```

will pull origins master branch → merge change to corrent branch where you are

→ pull can result in merge conflicts
→ We need to resolve the conflicts just like normal merge.

Shorter command

```
git pull
```

remote - will default to origin
branch - will default whatever tracking connection is configured for the current branch.

→ It is not recomended if you have uncommited changes.

When merge conflict occur file look like this:

```
<<<<<<< HEAD
changes from Local
=====
changes from remote
>>>>>> d13e5 ... '41
```

Readme files:

→ readme file is used to communicate important information about a repository including:

:: What project does

2. How to run the project
3. why its noteworthy
4. who maintains the project

→ If you put README in the root of project, github will recognize it and automatically display it in the repo's home page.

→ README is like entry point to learn more about the project or application.

→ README's are markdown files, ending with the .md extension. Markdown is convenient syntax to generate formatted text. It's easy to pick up.

.md (Mark down):

```
#h1 Heading
## h2 heading
### -h3
#### - h4
##### -h5
###### -h6
```

we need to give a space between # and word

Horizontal lines

```
***
---
```

```
Bold   ** Bold Text **.
       __ Bold Text __

Italic  * Italic *
        _ Italic _

Strike through  ~~ Strike ~~

Block quotes   >
               >>  } representing leval
               >>>
```

Inline code `Vishnu Vardhan'

Intended code

```
    line 1 of code
    line 2 of code
```

To make block code

```
'''
  Sample text

'''
```

```
''' js
var foo = function (bar) {
  ..
}
'''
```

Github Gits:

→ Github Gits are the simple way to share the code snippets and usefull fragments to others

→ Gits are much easier to create but offers few features compared to normal git repository.

Every Gist → Git repository

→ You can see all the gists in

https://gists.github.com

Gists → Public, Secret

→ Public gists show up in discover, where people can browse new gists as they are created. They are also searchable so you can use them if you'd like other people to find and see your work.

→ Secret gists do not show up on discover and not searchable, but they are not private, if you have url you can access it.

Github gits → If you have sample snippet of code then you can create a gist

→ We cannot change secret - public or public - secret at the creation only we need to set it.

Github pages

→ Github pages are the public web pages that are hosted and published by Github

→ Github pages is a hosting service for serving static web pages.

→ It does not support server side code like PHP, Python, ruby or node.

Github pages - HTML, CSS, Js code only

→ Each github repo can have corresponding hosted website.
→ We want tell which branch to take for hosting the repo
→ The default url in github pages follow this pattern

http://username.github.com/repo-name

Pull Requests:

→ Pull Requests are the features built in to products like github & Bigbucket. they are not native of git itself
→ They provide a mechanism to approve or reject the work on a given branch also help facilitate discussion and feedback on the specific commits
→ Pull request is nothing but merging in feature branch.

Pull Requests Workflow:

1. Do some work on a feature branch
2. Push up the feature branch to github
3. Open a pull request using feature branch just pushed up to Github.
4. Wait for the PR to be approved and merged. Start a discussi on the PR. This part depends on team structure.

Collaborators & branch protection rules:

→ We can add collaborators on the particular repository. they can push the changes directly to the main branch.
→ We can apply branch protection rules to push the changes directly to the main. (like we need to make pull request inorder to merge to the main)

Git Forking:

→ When we are having large open source projects with lots of collaborators.
→ They employ this forking strategy or workflow where there might be a handfull of actual maintainers.
→ They cannot add 1000's of people as contributors or collaborators.
→ This forking workflow enables anybody to try and make a contribution for the repository.
→ There is no permission needed. You can make your own copy. You are making changes and then you make a PR

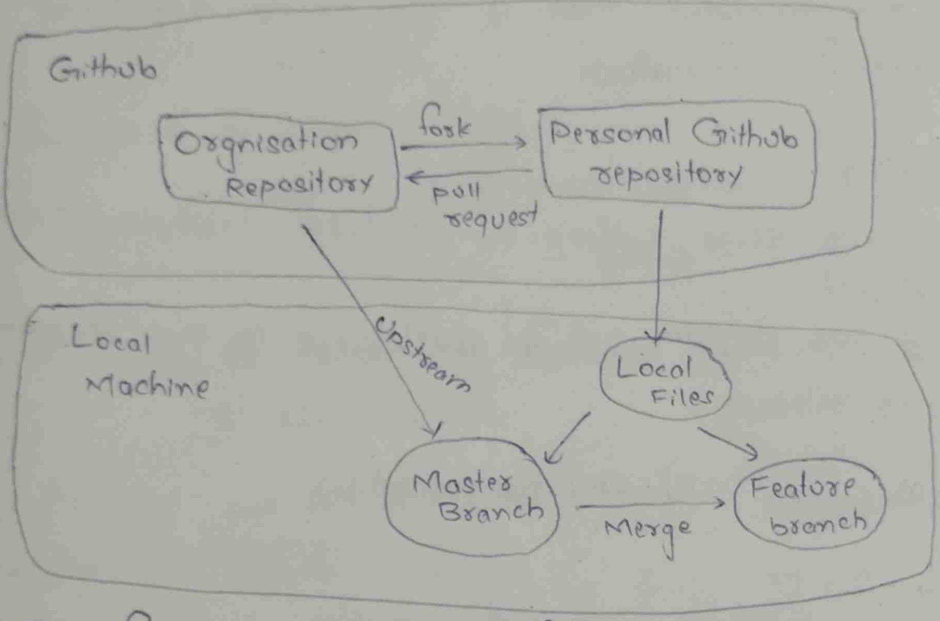
→ Anybody can make a pull request

→ Github and other similar tools allows us to create personal copies of other peoples repositories. we call it as fork of the original.

→ Fork is not a git native feature, the ability of fork is implemented by github.

→ If i want to share my work, I can make a pull request from my fork to the original repo.

→ This means that whole bunch of people can fork and can work on the project without actually having permissions to them.



→ The fork and clone workflow might seem complicated, but its extreme common for good reason.

→ To get the latest code from the original repository everytime

```
git remote add upstream <original repo Url>
```

→ To get the data from original repo

```
git pull upstream main
```

Git tags:

→ Git tags main idea is that we can tag particular commit so we can label commits by creating a tag, a reference to a moment in time.

Tags - pointers - that refer particular points in git history

→ Tags are most often used to mark version releases in projects (V4.1.0, V4.1.1 etc)

Once tag created - it always refer to same commit

Two types of tags:

1. <u>Lightweight Tags:</u>

→ They are just name /label - points - particular commit

→ Lightweight tag is much like a branch that does not change.

2. <u>Annotated Tags:</u>

→ Stores extra meta data including the authors name and email, the date and a tagging message (like a commit message)

Annotated Tags - Stored as full objects in the git database

→ It generally recommended that if you create annotated tags so you have full information.

<u>Semantic Versioning:</u>

→ Semantic versioning specs outline a standard versioning system for software releases.

→ It provides a consistent way for developers to give meaning to their software releases.

Version - 3 numbers seperated by period (.)

4 . 2 . 1

Major Minor Patch

Initial release - 1.0.0

Patch Changes - do not contain new features, signify bug fixes and (1.0.1) other changes that do not impact how code is used.

Minor Release: New features added, but projects backwards is (1.1.0) compatable, no breaking changes, new functionality is optional and should not force the user to rewrite their own code.

Major Release: It signify significant changes that is no longer (2.0.0) backward compatable. Features may be removed or changed substantially.

→ To print list of all tags in the current repository

```
git tag
```

→ We can also seach the tag name with patterns

```
git tag -l "*beta*"
```

→ To go to the tag commit (This puts us detached head)

```
git checkout <tag>
```

→ To check the difference between two tags commits

```
git diff v17.0.0 v17.0.1
```

→ To create a tag

```
git tag v1.0.0
```

→ To create an annotated tag

```
git tag -a v1.1.0
```

(It will allow us to enter a messege tagging message)

→ To view the meta data in the annotad tag (also for lightweight tag)

```
git show v1.1.0
```

→ Whenever we will push code to the repository tags are not pushed by default.

→ To push all the tags to remote repository

```
git ~~--tags~~ push origin --tags
git push origin v1.0.0
```

(To push specific Tag)

Git Reflogs:

→ The term reflogs is a short form for reference logs.
→ They are just logs that git keeps us for as a record.
→ Git keeps a record of when the tips of branches and other references were updated in the repository.
→ We can view and update these reflogs using the git reflog command.

Git reflog activity → only local activity
→ not shared with Collaborators

→ Reflogs also expire. Git cleans out old entries after around 90 days, through this can be configured

→ Git reflog accept the subcommands like
show, expire, delete and exists

→ Show is the only command used variant and it is the default subcommand.

```
git reflog show (log of specific reference) (default HEAD)
git reflog show main (view logs of tip of the main
                      branch)
```

→ We can also see reflogs for a branch

```
git reflog show head@{3} (It will show logs from the
                          3 position)
```

```
git reflog show master@{1.day.ago}
                master@{2.day.ago}
                Master@{2.week.ago}

git diff head head@{yesterday}
```

→ If your commits are

feature2
feature1
Initial commit

when you want to remove feature2 then

```
git reset <hash> --hard
```

Now you realized to want feature2 then

```
git reflog show master
```

In that feature2 hash is available go to that feature2 hash

@ - Used for reflogs
~ - Used to move to specific comits

Create Aliases:

→ To set the aliases we need to set it in the global .gitconfig file

```
[alias]
  S=Status                 git s = git status
  l = log                  git l = git log
```

→ we can also type the command

```
git cronfig --global alias.br "branch"
```