GIT
&
GITHUB

WHAT IS GIT:

↳ GIT is a VCS (Version Control System)

↳ It is the most famous version control system

What does it do?

↳ Track changes across multiple files

↳ Compare versions of a project

↳ Reverts to a previous version

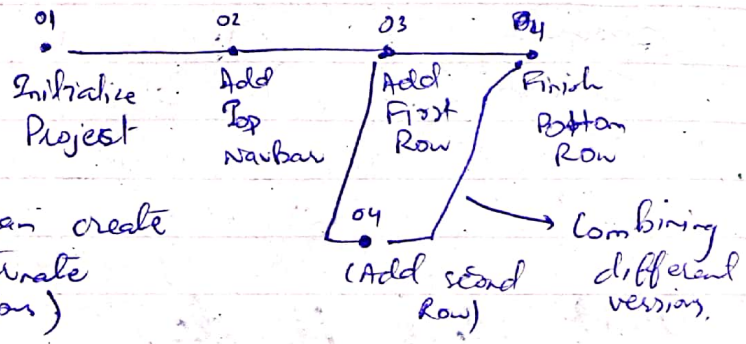
↳ Collaborate & share changes

↳ Combine changes

↳ Time travel back to old versions.

- It basically creates checkpoints as millions of collaborates with millions of lines of code every single day.

Checkpoint:



(We can even combine these versions to create a hybrid version.

Who USES GIT:

- Entrepreneurs
- Engineers.
- Governments
- Tech giants like FaceBook, Twitter, whatsapp etc.
- Tech Adjacent people.
- Writers, Essayists, Screenplay writers

GIT V/S GITHUB

GIT	GITHUB
Version control System	Puts git's work online so that it can be shared with other people
Store various versions	It connects Git Projects
	Share various versions of Git projects with others and helps in collaboration.

GIT TERMINAL CRASH COURSE

i) **ls** (ls -a : for opening hidden contents of the folder folders) currently opened.

to open a folder:-

ls <folder name / path>
w.r.t home directory

pwd

prints out your current location

cd

stands for "change directory"
this what we use to move around.

cd.. <folder directory>

cd.. will take you back one level.

start. will open the current folder in windows.
start <space>.

CREATING File & FOLDERS

touch (used to create a new file)
touch file.txt
A txt file will be created in the current folder

touch Desktop/Folder/text.txt

mkdir (used to make new folder)

mkdir git
(A folder called git will be created)

Deleting Files & Folder

rm (Deletes the file)

↳ Files deleted using rm will be gone they won't be placed in the recycle Bin.

↳ rm basically stands for remove

rm -rf

flags

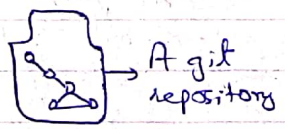
It deletes a folder completely

rm -rf Land Turtles

BASICS OF ADDING AND COMMITING

1. Git Repository

A Git "Repo" is a workspace that tracks and manages the files within a folder



A git repository

Git Commands

1. git status

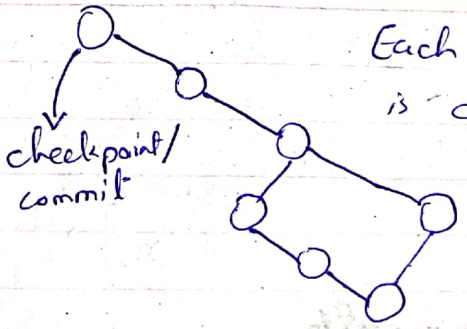
gives information on the current status of a git repository and its contents.

2. git init

Use "git init" to create a new git repository. It initializes a repository inside a folder

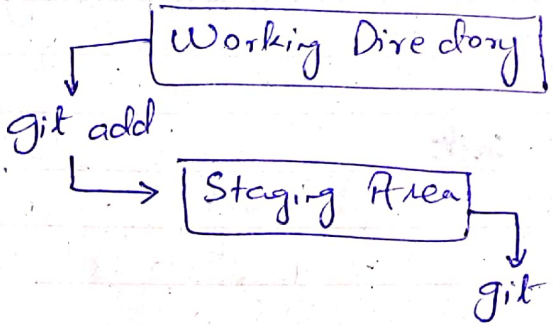
* Git also tracks all the nested subfolders within a git repository and YOU DONT want to INITIALIZE a git repository within a GIT REPOSITORY.

Committing in Git:



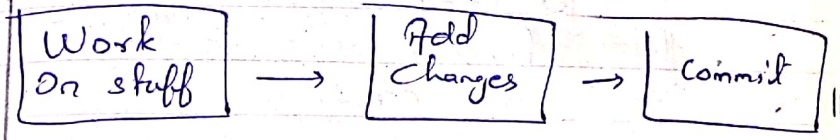
Each checkpoint is called a commit

Committing is not saving a file
You first save the change & then make a commit



git add

Basic Git Workflow:-



Make new files edit, delete files

Group specific changes together

Commit everything that was previously added

Git Command

```
git add file.txt
OR
git add file1.txt file2.txt

git add . : This will stage all the changed files at once
```

→ git commit

Commit message:

A summarised message of the changes made in that particular commit.

Command:

git commit -m "my message"

commit message

→ git log

Shows history of git commits you can also use

"git log --oneline" to see all logs in a summarized way.

COMMITS IN DETAIL

Atomic Commits:

When possible a commit should encompass a single feature change or fix, try to "keep each commit focused on a single thing."

To add a complete Directory:

git add folder/

This will add/stage all the files in the folder

TENSES USED IN COMMIT MESSAGES:

Some people recommend Present tense, some recommend Past tense use the rule which is being followed by your org.

Commiting messages using "git commit command"

01) Type "git commit"

AMMENDING COMMIT:

It only modifies and helps you change the last commit

"git commit --amend"

You can edit the last commit message as well as stage more files so that they could be added in the last commit.

→ This command only works for the last commit made.

IGNORING FILES:

We ignore some files for Git through that we ignore certain files which we don't want git to track

How To Do It?

↳ Create a .gitignore file

Write the following syntax for different scenarios in the file

→ Folder/ → complete folder
→ *.exe → any file with this extension
→ Or a single file name with ext.

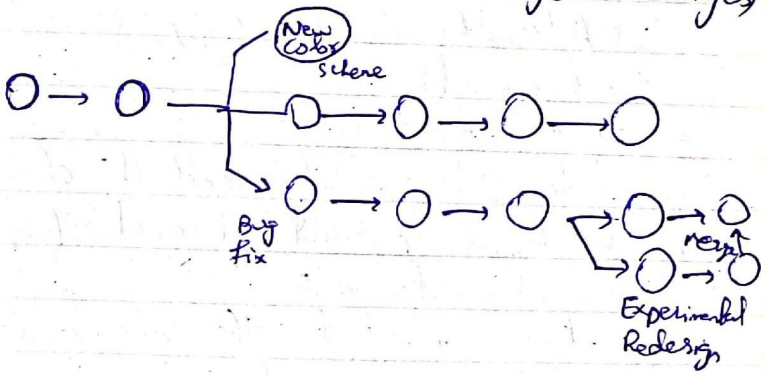
You cannot ignore an already tracked file in git.

WORKING WITH BRANCHES

BRANCHES

↳ Essential part of Git
↳ act as alternative timeline for projects
↳ Enable us to create separate contexts where we can try new things.

"If we make changes to one branch it does not affect other branches (unless we merge the changes)

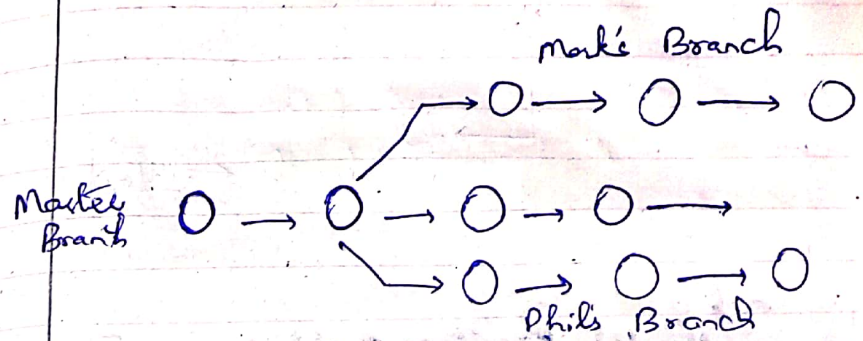


Master Branch:-

Many people designate the master branch as their "source of truth" or the "official branch" for their code-base, but that is left for you to decide.

From Git's perspective, master branch is like any other branch. It does not need to hold the master copy of your code base

"In 2020 Only github changed the default branch from master to main,
But in git the default branch is still master

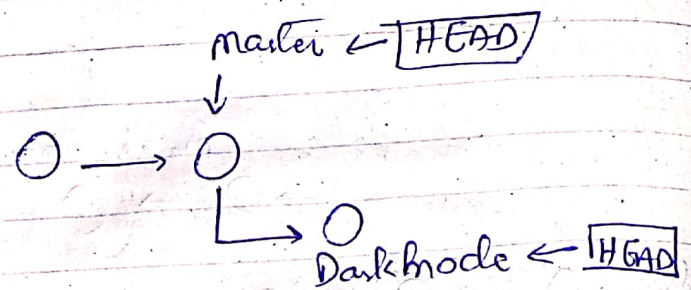


Branches are like bookmarks in a book. At any given time only one of them is active/open.

HEAD

Head is the current location that git is viewing or checking out it refers to the current Branch basically.

"Head is the reference to a Branch pointer, it can be any branch such as master, phil or any other."



↳ VIEW BRANCHES:

git Branch

It gives the list of existing branches.
The active branch has an "*" next to it.

↳ CREATING BRANCH:

git branch <branch-name>

The branch name must not have any spaces.

This just creates a branch, the head remains at the initial branch

↳ SWITCHING BRANCHES

git switch <branch-name>

for eg: git branch bugfix

creates that branch
and
git switch bugfix
moves you to that branch.

To make a new branch and switch to it directly we can use the "-c" flag like this

git branch -c Oldies
(Branch name)

↳ To see the fetched remote branches.

Git checkout <remote>/<branchname>
git checkout origin/master

To see remote branches
git branch -r

Switching Branches with unstaged changes:

(Already committed files)

If you try to change branches with uncommitted changes then it'll give you an error cuz the new work will be lost if we change branches without quiting. (This applies on a pre-committed file only).

You can either

- stash those uncommitted files
- Or commit them & then change the branches.

We don't need staging so will stick to committing for now

(new non-committed files)

If you create a new file on a branch and don't commit it and switch branches then that file is gonna come with you (uncommited) but won't cause any issue

Deleting & Renaming Branches:

deleting a branch:

you can delete a branch by

`git branch -d deleteme`

if your branch is fully merged otherwise

`git branch -D deleteme`

(-D = force delete)

this will delete the branch whether it is merged or not.

P.S You cannot delete a branch while your head is pointing on it.

Renaming a branch:

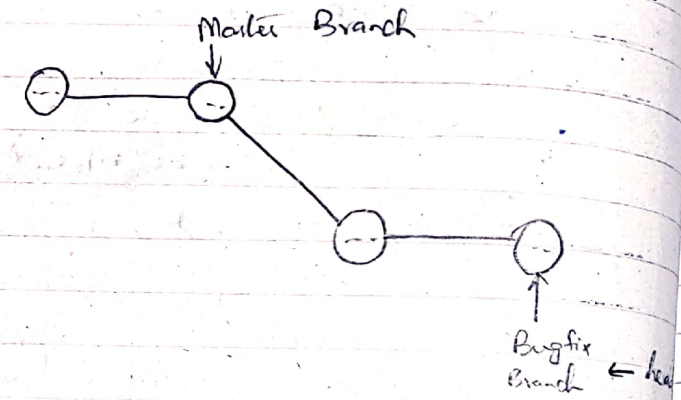
Go to the branch you want to rename then run command

`git branch -m filerenamed`

BRANCHING IN GIT:

↳ We merge branches not specific commits

↳ We always merge to the current HEAD branch

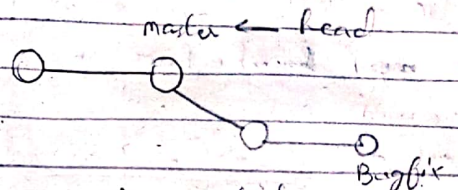


To merge a branch:

1) Switch to or checkout the branch you want to merge the changes into (the receiving branch)

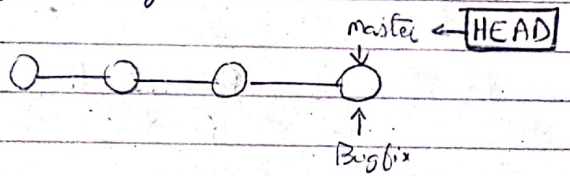
2) Use the 'git merge' command to merge changes from a specific branch into the current branch.

For Ex:- To merge "Bugfix" into "Master"



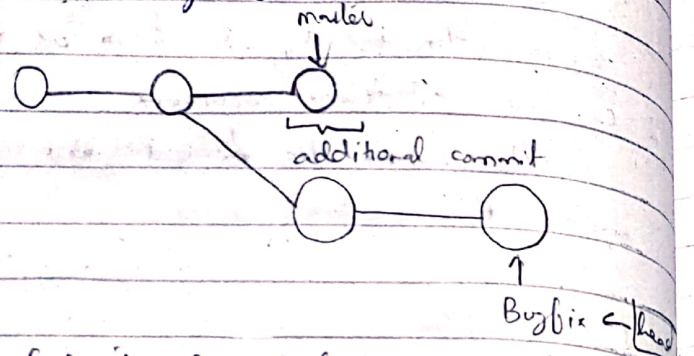
git switch master
git merge bugfix

After merge



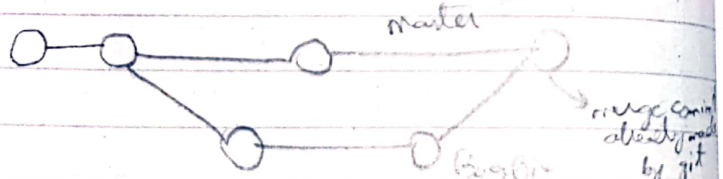
This type of merge (in which the pointer only moves forward) is called "fast forward merge".

What if master has an additional commit that other branches (which we want to merge) do not have



what if the latest comment on line 56 (let's say) has different code but the code we want to merge on the branch has different code? (A while later)

If the above issue does not arise then Git itself makes a merge comment for us



Merge conflicts:

When Git fails to automatically merge the files, it shows us the conflicts & their locations.

We manually have to resolve those conflicts then

Conflict- markers:-

```
<<<<<<< HEAD
I have 2 cats
I also have chickens
=======
I used to have ady :(
>>>>>>> bug-fix
```

Now it's upto you, if you want to keep the contents of the master branch or the bugfix branch

Here is the step-by-step guide:

01) Open up the file(s) with merge conflicts
02) Edit the file(s) to remove the conflicts. Decide which branch's content you want to keep in each conflict. Or keep the content from both.
03) Remove the conflict markers in the document.
04) Add your changes & then make a commit!

Git Diff:

We can use the git diff command to view changes between commits, branches, files, our working directory, & more!

git diff

running this command will show the change made & non-committed changes in a file

How "git diff" shows the differences?

diff files being compared

```
diff --git a/rainbow.txt b/rainbow.txt
index 72d1d5a..f2c8117 100644   } metadata (useless stuff)
--- a/rainbow.txt   } sign attributed to each file
+++ b/rainbow.txt
@@ -3,4 +3,5 @@ orange
 yellow
 green
 blue
-purple
+indigo
+violet
```

-3,4 : lines of file attributed to - sign
+3,5 : lines of file attributed to + sign

contents → text with "-" is associated with the file represented by -ve file & same is the case of +ve file

Git diff HEAD:

It shows the changes among the current working directory & the last commit i.e "HEAD"

While "git diff" shows the difference b/w the staging area and the current state.

git diff --staged / git diff --cached

This shows the difference b/w the last commit & staging area.

Diffing specific files only

To see the changes made only in a specific file we use

git diff <filename>

you can also use

git diff HEAD <filename>

You can also use multiple files

git diff <filename1> <filename2>

Comparing Branches.

git diff branch1..branch2

2 dots allowed with no space

you can also use.

git diff branch1 <space> branch2

Comparing changes b/w 2-commits

git diff <commit1 hash>..<commit2 hash>

also works with space

STASHING IN GIT:

If you don't want to "REALLY" make a commit, but want to save the work and move onto other branches

There are 2 situations which might arise:

01) The changes come with you to the second branch

02) Git will let you switch and ask you to commit those files.

The command is "git stash" it saves those changes. (All uncommitted changes) And then you can use "git stash" to remove the most recently stashed changes in your stash & re-apply them to your working copy.

git stash save is an alternate of git stash

git stash apply

applies the uncommitted changes but it doesnot remove them from the stash

WORKING WITH MULTIPLE STASHES:

You can stash multiple times using "git stash" and then use the "git stash list" command to show the stashes.

You can choose which stash to apply using:

git stash apply stash@{2}

Application of multiple stashes:

If there are more than one stashes and you use "git stash" command then the stash at the bottom of the list will be applied to the current working directory.

Dropping stashes:

You can remove particular entries in the stash using

git stash drop stash@{2}

Removing all ~~stales~~ stashes:-

git stash clear

UNDOING CHANGES & TIME ~~TRE~~ TRAVELING

How to travel back to a previous state?

git checkout 486f9a4
(1st 7-digits of the commit hash)

and you'll travel back to the state of the repository at that time. It's called the "Detached head state".

Generally head refers to a branch address but when we travel back the head actually points to a commit, hence the term 'detached head'.

What can we do in a detached head state?

To go back to a branch use
git switch <branch-name> and you'll go back to that master
or use git switch - to take you to the branch you were last on

How to make a new branch at a detached head?

Just use "git checkout <hash>" command to make a detached head & make a new branch at that head using

git switch -c branch_at_head

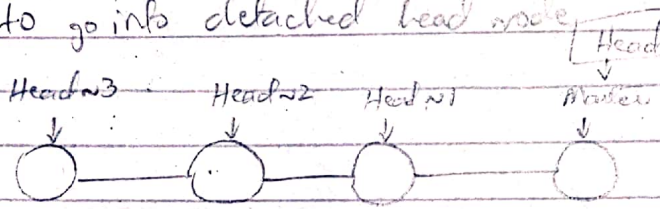
& you get a new branch at the head.

A dbb syntax

you can also use

git checkout HEAD~1 (the commit before head)

git checkout HEAD~2 (2 commits before head) to go into detached head mode



DISCARDING CHANGES:

If you've made some unnecessary changes. one option is to actually delete that else otherwise it'll show up as modified files.

OR

you can use

git checkout HEAD <filename>

and git will revert those changes back to the HEAD state of the file a shorter version. i.e.

~~git checkout HEAD <filename1> <filename2>~~

git checkout -- <filename1> <filename2>

Git Restore

It was introduced to take a bit of burde away from git checkout --<file>

git restore <filename>
its a bit easy

you can also got back to a previous version using

git restore --source <Commit hash>/<filename>
HEAD~n

KEEP IN MIND THAT ALL UNCOMMITTED CHANGES WILL BE LOST AFTER THIS

UNSTAGING FILE WITH GIT RESTORE:

Use

git restore --staged <file>

to unstage a staged file

GIT RESET:

This resets the repository back to a certain commit hash / Head1 address.
There are 2 flavors of git reset.
1) regular reset
2) hard reset.

01) Regular reset

git reset <commit hash>

This will only "un-commit" the work but the change will still be in your working directly you can edit them or do something else.

You can commit those stuff on another branch too.

02) HARD reset:

git reset --hard <commit hash>

This will roll back the repository and delete those changes from your files

Git Revert:

It works in a similar way to a git reset but it creates a diff commit where it undoes those changes.

git revert <commit-hash>

Its better to use "git revert" when you are working on a project while collaborating with others.
If the commit only resides on YOUR machine and you want to get rid of it, then you can use git reset

GITHUB

Git clone:

we use this command to download online github (or anyothe repositories) & make changes to it

```
git clone <url>
```

How Do I Get My Code On Github:

Option 01 (Existing Repo):

1) create a new repo on Github
2) Connect your local repo (add a remote)
3) Push up your changes to Github

Option 02: (Start from scratch)

If you haven't begun work on your local repo you can

- Create a brand new repo in Github
- Clone it down to your machine
- Do some work locally
- Push up your changes to Github

(OPTION 01)

Connecting a github repository to locally to a git repo:

```
git remote add <name> <url>
git remote add origin
  https://github.com/blah/repo.git
```

'origin' is the default name to the URL generally, but you can also choose any other name

Then you can check using the git remote -v option if it is configured properly or not

Renaming the remote repo

~~git remote <oldname> <new name>~~
~~git remote <~~

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

Pushing the code on github

1) Create a repo ✓
2) Connect a local repo (add a remote) ✓
3) Push up your chages to Github (~~that is to be done~~)

```
git push <remote> <branch>
```

origin in this case

generally we push

```
git push origin master
```

you dont have to be on a branch to push it

(OPTION 02)

Starting from Scratch

1) Make a github repo
2) Clone it on your laptop/Pc
3) it'll automatically configure "remote on your Pc"
4) you can push it on your github then

FETCHING AND PULLING:

Remote tracking branch:-
It's a pointer which keeps track of when you pushed pulled something from Github.
After every commit you make locally, it keeps reminding you that you are "n" commits ahead of the "origin/master" point.

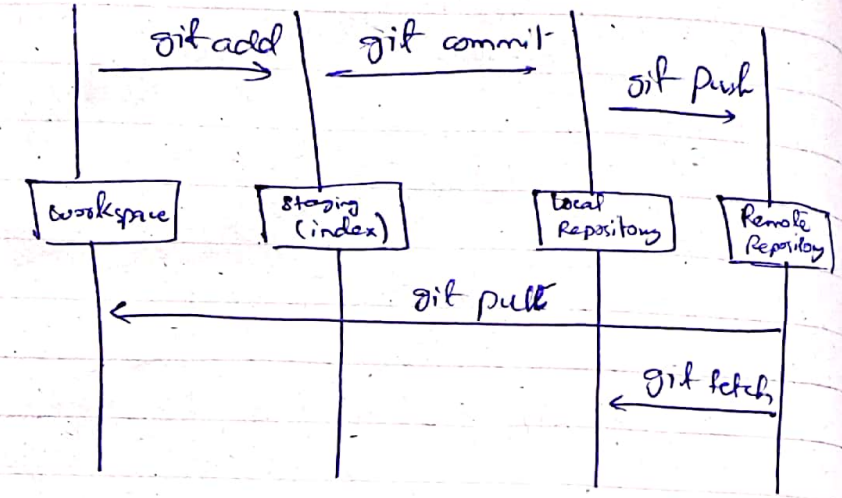
Remote Branches & working with them:

When we clone a project from github we only have the master/main branch the other branches are remotely available for eg:- if there is a remote branch called puppies, it will not be available by default when you clone a git repo. you'll have to use

git switch puppies

to actually use & switch to that branch

FETCHING:



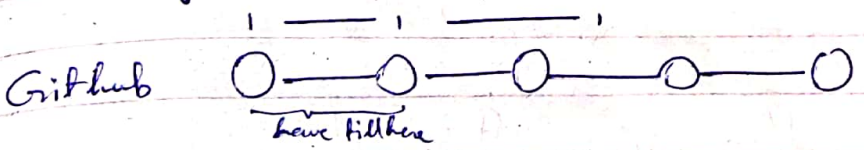
Fetching: Allows us to download changes from a remote registory But those changes are not integrated into working files

Its like saying.
"Please go & get the latest information from github, but don't screw what I am currently working on".

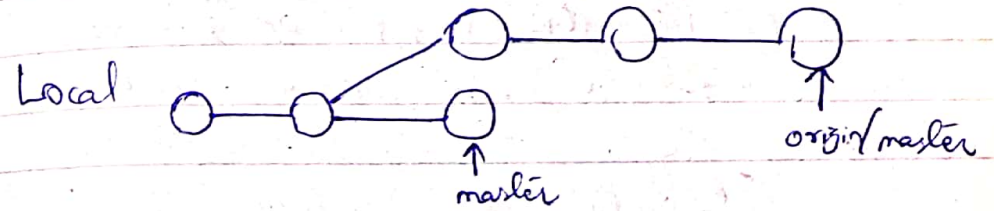
> git fetch <remote>
If not specified, <remote> defaults to origin

you can also fetch a specific branch

git fetch <remote> <branch>



you can access to those changes on your machine using the checkout origin/master



Pulling:-

git pull = git fetch + git merge
- update remote tracking branch with the latest changes from the remote repository
- Update the current branch too

How git pull works?

Whenever you run "git pull", it fetches the changes from master & merges into your current branch

Just like normal merges, pull requests also have merge conflicts.

Shorter syntax:

git pull

Its just automatically pull the origin/currentbranch from github

But this does not work with multiple remotes

SOME IMPORTANT THINGS ABOUT GITHUB:

Adding collaborators (People who can push code to the repository)

Go to Setting > Manage Access > add the github username/email

Now the person will get an email notification, he'll have to accept the invite

Now the person can push changes to the repo

README files:

A file that explains
- What the project does
- How to run the project
- why its noteworthy
- who maintains the project

If you put the README in the root directory github automatically renders it on your projects wall.

Readme is a file using markdown hence the extension .md

Git gists:

↳ Its an alternatives of pastebin
↳ You can access it by using gists.github.com

you can just share code using it, its more convenient than using a git repository

GIT REBASING:

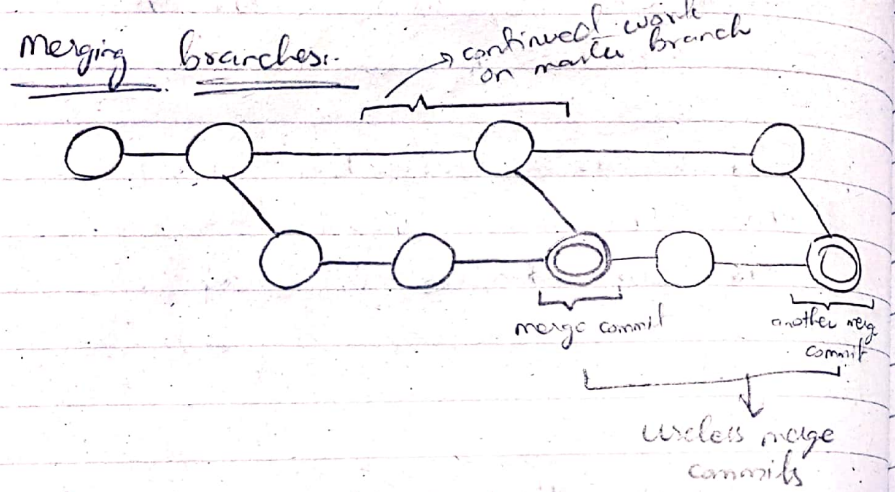
There are two main way to use the git rebase command:
- as an alternative to merging
- as a cleanup tool

Suppose you are working on a feature branch, but you have a very active main branch with lots of changes being made regularly. You have to keep merging main in your branch to keep your code base up-to-date. What'll happen as a result is that you'll end up with a lot of useless merge commits that don't really tell much about the code of your specific feature branch.
Rebasing helps us to address these issues.

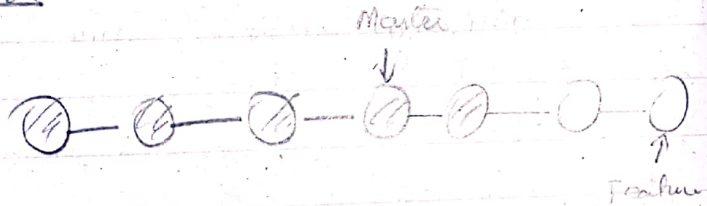
The command used is

```
git switch <feature branch>
git rebase master
```

Merging branches:-



Rebasing:-



When you rebase, new commits are generated, it rewrites history.
When you rebase, the commits are recreated & added to the tip of the master branch.

Rebasing cleans up the history & removes the useless merge commit
You can also rebase without merging

WHEN NOT TO REBASE?

Never REBASE COMMITS THAT HAVE BEEN SHARED WITH OTHERS. IF YOU have already pushed commits upto Github DO NOT Rebase them unless you are positive that no one on the team is using these commits.

USING GIT REBASE TO REWRITE HISTORY:

Rebase can be used - as an alternative to merging
- as a cleanup tool.

- DONT REWRITE HISTORY OF WORK PEOPLE ALREADY HAVE. So preferably do this work before pushing it on the main branch.

CLEANING THE GIT HISTORY:

git rebase -i HEAD~9
↳ 9 commits back from the head

↳ Pressing ENTER will open your text editor and it will allow you to pick/reword/drop commits from your selected range.

↳ The list of the commits will be in a reverse order i.e. ~~start~~ latest to oldest

Some of the most used commands are as follows:

- pick - use the commit
- reword - use the commit, but ~~edit~~ edit the commit message
- edit - use commit, but stop for amending
- fixup - use commit contents but meld it into previous commit & discard the commit message
- drop - remove commit

Reword command:

Reword will allows you to change the message of commit after you close the window. Save the message window & close that too, your message will be saved.

It will also change your commit hash.

fixing commit message "fixup" command:

there are two ways,
- squash: use commit but meld into previous commit
- fixup: like "squash" but discard this commit's log-message

pick up <commit message>
fixup <commit message>

↳ this will be merged into the previous commit message & the changes will be saved but the commit message will be deleted

"drop" a commit:

This will remove the entire commit & its changes.
Used git rebase -i HEAD~N to enter the rebase menu and then rename "pick" by "drop". The commit along with its changes will be deleted.

GIT TAGS:

Used to mark different releases of apps such as version 7.1.0, 7.1.1 etc.
A label pointing to a particular moment in time. They are kind of like a branch reference, except that the branch reference changes with time (like HEAD keeps changing), but tags keep refering to the same commit always.

There are 2 types of tags.
01) lightweight tags
02) Annotated tags

lightweight tags: They are just a name/label that points to a particular commit.

Annotated tags: stores the extra meta-data including the author's name & email, the date, and a tagging message (like a commit message).

Generally we use annotated tags more than lightweight tags.

Semantic Versioning:

2.4.1
- 2 → major release
- 4 → minor release
- 1 → patch release

Initial release will be 1.0.0
Patch releases are on far right.

1.0.1 → 1st patch

Minor Releases:-

- They specify new features & new functionality, backwards compatibility is maintained.

1.1.0 → 1st minor release

- Every minor release resets the patch number back to zero.

Major Release

- Significant changes
- No longer backwards compatible
- Features may be removed or changed substantially.

2.0.0 → minor release & patches are reset to 0 after every major release.

Viewing Tags:

Command:- git tag

will list all the tags in the current repository.

Command:- git tag -l "*beta*"
will list all tgs with wildcard "beta"

git tag -l "v17*"
will list all v17 tags

checking out tags:-

git checkout <tag>

& you will enter detached head state and you can make commits from it & make branches etc.

comparing two releases:-

you can compare two major/minor releases or patches using git diff command to see the difference btw two releases.

git diff v16.14.0 v17.0.0
- will list all the changes in these releases.

CREATING TAGS:

i) creating lightweight tag:

- make changes and add & commit them.

run command git tag v17.0.2
(tagname)

and the tag will be created where the HEAD is currently pointing.

ii) creating annotated tags:

- add and commit changes you have made.

command git tag -a <tagname>

and it will prompt you in an editor and in the window you can add additional information about tag.

- You can use the -m flag to pass the message directly and forego the opening of the text editor.

Viewing info about annotated tag

Command: git show <tagname>

This show the Tagger info, & the tag message to you

Tagging Previous Commits:

Command: git tag <tagname> <commit hash>

The above command can be used to tag previous commits.

Replacing tag with force(moving tag):

git tag <tagname> <commit hash> -f

Deleting tags:

git tag -d deletetime

Pushing tags:

When pushing a repository on github the tags are not pushed with them. You have to push these tags separately.

Commit:- git push <remote name> <tagname>

or if you want to push all tags at once

git push colt --tags

BEHIND THE SCENES IN GIT - HASHING & OBJECTS:

- Config:

- Putting setting in the local config file of a repo will change it for the repo only.
- There is also a global config file, we'll talk about it later.

- To setup thing locally in a repo, use the --local flag.

git config --local user.name "name"

- Refs Folder

- refs/head : contains one file for every branch, each file contains a commit hash.

- It also contains a tags folder containing all the tags in the repo.

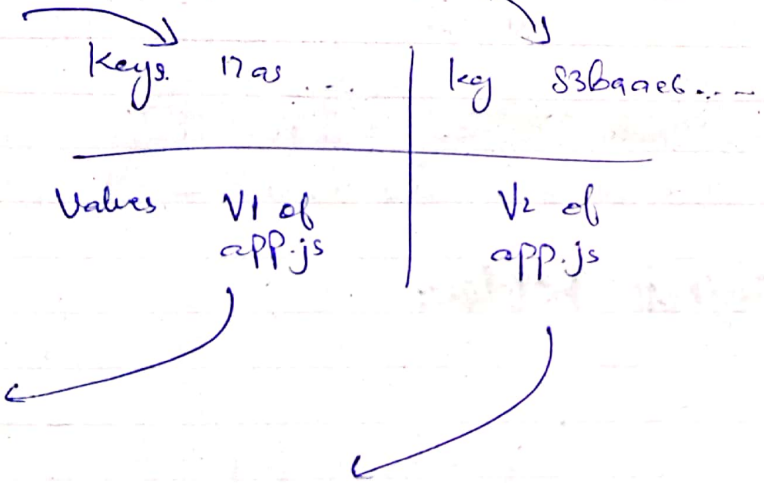
- Remote folder contains all the "remotes" setup by the user in the repo

- Objects Folder:

- Most important folder
- contains all the backups etc.
- There are 4 basic types of git objects.
  01) commit 02) tree 03) blob
  04) annotated tags

Git as A key value Data store:

Grit is a key value data store we can insert any kind of content into a Grit Repository & Grit will hand us back a unique key we can use later to retrieve that content.



Hashing with git hash-object:-

git hash-object <file>

In this form (shown above), Grit simply takes some content & returns the unique key that WOULD be used to store our object.

echo "hello" | git hash-object --stdin

will create the hash and return it, it will generate the same output every time.

If we use "-w" flag it will actually store our object in file.

[echo "hello" | git hashobject --stdin -w

will store data in .git/objects/ce/<hashnumber> in compressed form.

Reading data with the help of commit hash

git cat-file -p <object hash>

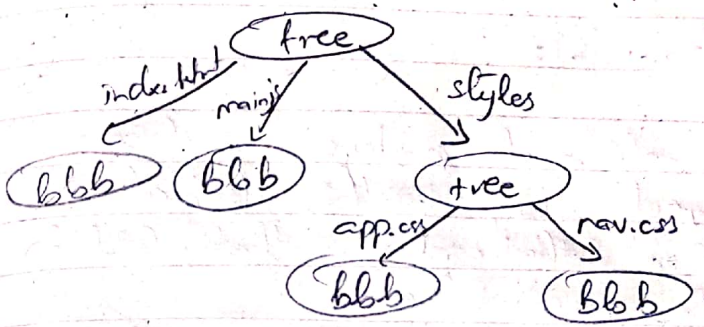
Blobs:

Git Blobs (binary large objects) are the object type Git uses to store the "contents of files" in a given repository. Blobs don't even include the filename of each file or any other data. They just store the contents of a file!

Trees:

- Trees are git objects used to store contents of a directory.
- Trees contains pointers that refer to blobs and other trees.
- Each entry in a tree contains the SHA-1 hash of a blob or tree, as well as the mode, type & filename.

tree		
blob	1f7a7	app.js
tree	98287	images
blob	bc321	README



→ Viewing Trees:

`git cat-file -p master^{tree}`

will show you the list of all the tree structures in a repository.

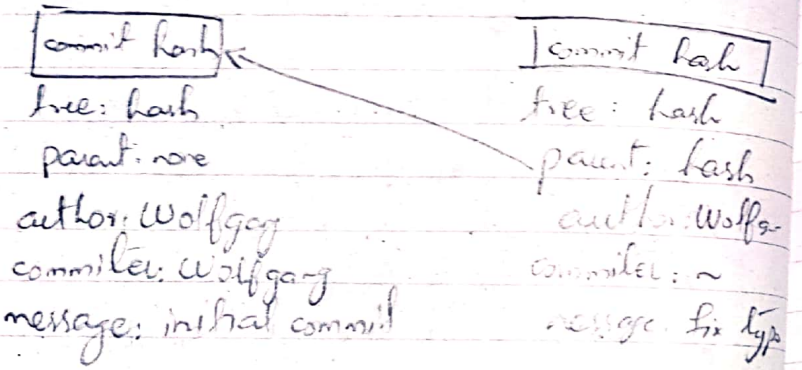
→ Find type of an object:

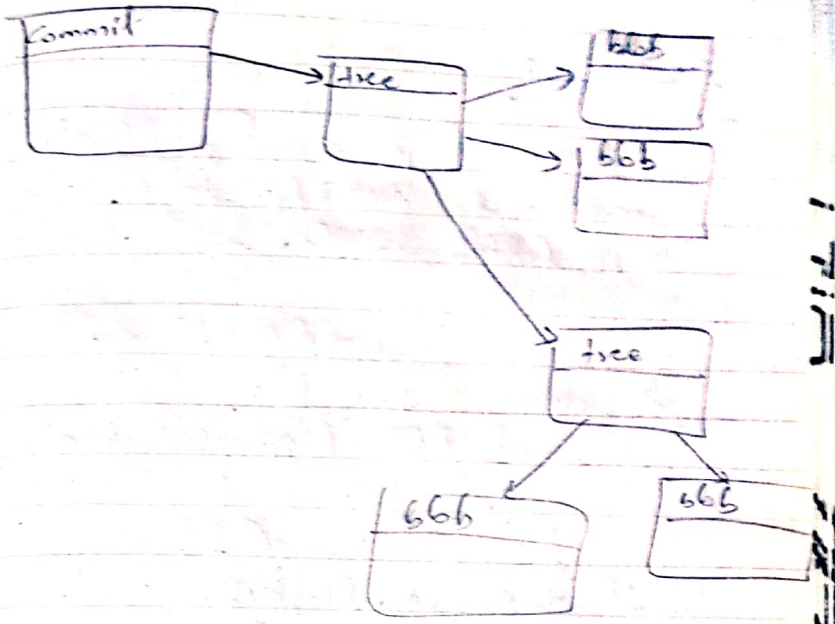
`git cat-file -t <object-hash>`

Commits:

→ Commits combine a tree object along with information about the content that led to the current tree.

→ Comits store a reference to parent commit(s), the author, the commiter, and of course the commit message.





x ← x —— x

END-OF-SECTION.

Reflogs:

Git keeps a logs of the movement of HEAD pointer in the "logs" folder in the git directory

- It keeps logs of every individual branch
- It also keeps track of all the code pushed to the remote.

Limitations of Reflogs:

- Reflogs are only local.
- They have an expiry date (around 90 days) although you can change the limit if you want to.

Command :-

git reflog show HEAD

will show a printed version of the reflog of head pointer

You don't have records of deleted, rebased, refreshed commits on git.

Reflog References:-

- We can access specific git refs is name@{qualifier}
- This syntax can be used to access specific pointers & can pass them to other ~~pointers~~ commands, including checkout, reset & merge.

Reference Example: HEAD@{2}

name qualifier.

You can use this command to hop around in git references, for eg:
git checkout Head@{2} will put your HEAD 2 steps behind. (this is not the same as detached head).

Timed References:

You can pass exact timestamps to see what exactly HEAD/ a particular branch looked like.

→ git reflog master@{one.week.ago}

→ git checkout bug-fix@{2.days.ago}

→ git diff main@{0} main@{yesterday}

Reflog Rescue:

Suppose you delete a commit using

git reset --hard <commit hash>

but you want to recover it again, you can recover it through reflogs.

git checkout <deleted commit hash>

and do 'git reset --hard master@{1}' and it will be restored and you can undo a hard reset this way, but this is only possible to do locally.

UNDOING A REBASE:

If you do interactive rebasing & "fixup" a bunch of commits you can rebase it using reflogs.

Open the git reflogs, using,

git reflogs show master@{0}

& then

git reset --hard <commit hash>

hash of deleted commit

and all of those commits will be restored.

END of section

Writing Custom Git Aliases:

Three levels of .git config files.
- ~/.gitconfig (repository file)
- ~/.config/git/config (global config file)
- system-wide config file (mostly people don't use this)

Take a look at git config file:

command:- cat ~/.gitconfig

Alias:

It is a shortcut for an existing git command to make the experience a bit faster & simpler.

For example:- we could define an alias "git ci" instead of having to type "git commit".

How to make an alias?

[alias] → starting heading in the local/global gitconfig file.

s = status

s → what you want the command to be called

status → known git command

after saving, you can run "git s", instead of "git status" and it will work just fine.

Setting aliases from command line:

git config --global alias.showmebranches branch

showmebranches → what you want it to be called

branch → original command name.

command:

git config --global alias.showmebranches branch

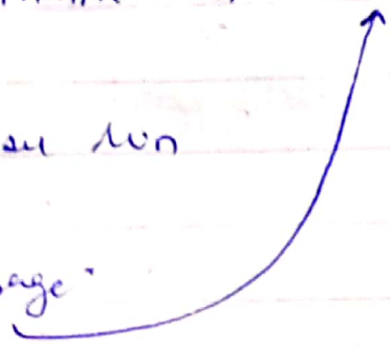
You can also set local aliases.

Creating aliases that work with arguments:

cm = commit -m

and when you run

git cm 'message'



The message will just be appended at the end.

x —— x —— x

END-OF-COMMENT