Note: Don't Solve merge conflicts in main
Solve it feature itself

Don't merge to main? ⟶ pull request after merging to feature
rebasing (own feature).

Meta - Backend development - Git & Github.

Version Control

→ Undo - time machine - Version control (Records changes)
→ Centralized / distributed
→ Conflict management

Communication - keep in track, Collaborate

→ Together - Engg & non-Engg
→ designers, managers, Engg, ...
→ prioritize which is important
→ version Control: Context switching
→ Estimate product (how long, trade)
→ Give Context to people to work.
→ Adjust, (Visual learners, think before speak people), mobile working.

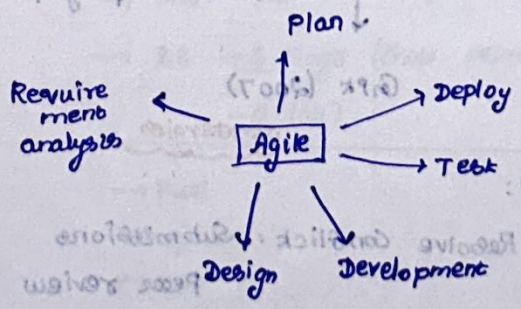
→ modern dev - work together - No messing up of each others code
→ effective work flow, automate

Version Control

1. Time machine - Record, track modification (keep track of changes)
2. Dev calls it - Source code management / Source Control
3. Access entire change history, Revert back. (Roll back)
4. Changes: Add, modify, delete (Source of truth)
5. Restoration points (Revert)

Benefits: Revision history, Identity, collaboration, automation, efficiency.
(Record of changes)
Roll back

Roll back: Confidently experiment
Identity: who made it, when? [who made], Analyze !
Collaboration: (Team) - Submit code, track changes (Peer review - inspection, Feed back)
Devops: Tools, practices - ↑ ability of service → high quality & speed.



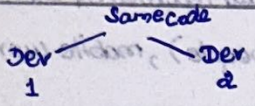
Agile methodology.

* Task list - Shedule.

Testing & Automation: make sure Integrity. - Efficiency

Meta Engineers

* Collaborate on doc, Chat
* monolithic repo - Reuse other's code. , Approve changes
* Nothing is someone else's problem.
* Challenges: merge conflicts - (Smaller code so reverntable) - gate keepers.
* Git blame: Search history (who wrote it) - collaborate with them.
* Success matters: version control.



* Same Code base: Keep track of changes

eg: 1. Subversion 2. Perforce 3. AWS code Commit 4. Mercurial 5. Git

Types → centralized - CVCS - Server ↔ Client (Pull up, pull down - from single server)
→ Distributed - DVCS - Similar - Every user - Server (pull down - gets complete history) - on local system

CVCS: Server (Central Copy of Code) - push to it - [Easy to learn, access control] - Slower

DVCS: No need to connect (pull up/down - only necessary) - good speed

---

1986 (Walter F. Tichy - Pardue university)

CVS - Concurrent version System

↓ Text file (not binary - img)
name, location, modified.
Flaw: data corruption

2000 - collabnet's Subversion (Integrity check)

↓ free hosting - linux, Google, SourceForge
centralized

Mercurial (2005)

↓ Linux Kernel
Distributed, high performance

Git (2007)

Strategies

---

1. Good workflow:

Resolve Conflict: Submissions
Peer review

2. Continuous Integration (CI) - Automate code changes
Compile, Run tests, Ensure stable.

3. Continuous delivery - merge, pack, deploy (avoid human error - while packing)

4. Continuous deployment - deploy & release frequently. Automatic Staging (validate & make live)

---

1. Aid : Full history of files (common goal) - Follow flow / upgrade (who, when, what)
2. Walk through entire project (every change - accessible)
3. Standards - more info better (transparent)

3 person - feature

1. Daily report
2. Insight
3. Which is winner ⟶ push (pull request) ⟶ peer review (approve / decline)

Developing environment (UAT / QA / Staging env)

1. Bugs ? Test (trial)
2. mimic production env - spot bugs (accurate) - cover all areas.
3. New features : Turn on / off : without affecting existing.
4. Quality Analyse teams - Run tests - production capable ?
5. Down time - Reflect Income, Reputation.
6. Vulnerability - patch, update

Command Line

1. Quick, powerful, less error - need knowledge
2. Track, Access remote servers, search, disks, install & uninstall, Access manual, unzip, Adv: automate, user access control, Stop / restart, Create aliases, Run, Download, Containerization (Self contained Software)

cd ⟶ change directory

touch ⟶ new file

mkdir ⟶ make directory

history ⟶ show command history

folder example.doc

Unix Commands

* Linux be preceded by Unix
*

⟶ man ls (manual)

⟶ ls -l Flags (show permissions) - risk

-a (all)

⟶ pwd

⟶ cp

⟶ mv

→ ls -la → all files, list view

→ less .bashrc → check file (less data shown)

.profile → env variables

vim

? → insert mode

#!/bin/bash → bash script

echo "Hello world"

executable (windows) - Bash

» Chmod 755 testShell.sh

↳ executable permission

Commands

* / → Top level directory — flag-options

* ls → list l → list format

* cd .. → To parent

* cd /etc → absolute path

Pipes

* → Cat ____ → Read file

→ WC file1.txt -w → word count

(word count)

Pipes: pass one o/p to i/p of another command

→ ls | wc -w (how many files)

→ cat file1.txt | wc -w (word count of file1.txt)

→ cat file1.txt file2.txt | wc -w [combined word count]

Redirection

1. i/p, o/p, error

↓ ↓ ↓

0 1 2 (Standard i/p, o/p, error)

Std i/p

cat > input.txt

This is me adding.....!

Ctrl+D to stop

cat < input.txt

o/p This is me

Std O/p

ls -l > output.txt

less output.txt

(view content)

Std error

2> → error

2> &1 → error / op → write to file.

eg: can't access / No such file or dir

→ ls -l /bin/urs 2> error.txt

error — write to

Grep

* Global regular expression print (Searching)

→ grep Sam names.txt
↳ list of names with Sam
case sensitive.

→ grep -i Sam names.txt
↳ case insensitive.

→ -w → exact match

List of directories

grep & ls

ls /bin | grep zip

Github / Gnik

* VCS – address challenges of Linux – Commit, update
* Speed, Reliability, opensource, syntax clear.
* Github: host (UI)

feature: pull req, automation, Access control along with GIT.

Secure, profile, opensource, doc, ticketing, ...

---

personal access token

Generate ssh keys → ssh-keygen -t ed25519 -C "your@email.com"

add using ssh (Github)

Clone repo

git clone https:// url.com

Git

* ls -la (list all) Readme.md → Info of project.

• Git → folder (hidden)

↳ Initialized by git init

As we cloned : git init (not required)

Git workflow

git : modified → staged → committed
add commit

add/edit/remove : modify
Git to track : stage (put in staged area) → add
Save point (Snapshot) : Commit → Commit
Remote Repo → Push
Fetch/checkout/merge from Remote repo

Add and Commit

git status → Any changes, Branch i'm on

Simple text file

touch sample.txt
git status → uncommitted/untracked files

Phase 1

git add sample.txt → up to date branch
staging requires

Revert changes : git restore --stage sample.txt
(unstage)
untracked status

phase 2

Staged area: prepare all files & changes to Commit (Commit ready) → local.

git commit -m "message"

Branches

git checkout -B feature/lesson → create branch
(creates & switch)

(or)

git branch → just creates branch
(branch command)

* feature/lesson
main
] branches list.

★ Branches – Isolated (need to merge back to main)

Pull request

1. peer review / validate
2. Teams: unit test, Integration test

```
# touch test2.txt
# git add test2.txt
# git commit -m "Add test2.txt"
# git push -u origin feature/lesson ⟶ push to this branch
```

↳ update from upstream

git pull (make up to date) with remote repo.

★ Compose branches ⟶ pull request ⟶ merge after approval.

Branch: manage, removes issues (Same branch)

UI

Compose & pull request

↓

approve pull request

↓

merge

→ git checkout main ⟶ go to main

→ git pull ⟶ get updates from merge.

Based on company. [feature / bug /

Remote (vs) Local

★ Remote: central / distributed.

★ Remote code: accessed using URI, only accessible to those having permission.

★ Local: machine, mobile (You)

★ clone: when pull to local device (1st time) – clone to a local folder.

pull: To get update (not 1st time)

push: change locally & update remote server. (all can see)

Adv: work offline & push ready.

```
# mkdir myrepo
# git init   (cd myrepo)
```

↳ empty repo

git remote ⟶ blank (no connection to remote repo – local)

Add remote repo

git remote add URI

git remote add origin git@github.com:_____/my-first-repo.git

git remote -v

_____ (shows remote).

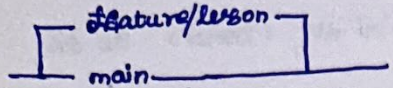
git pull

ls → blank (branches not matching)

git checkout main ⟵ create main to access remote's main.

ls → (Readme, test1.txt, test2.txt)

Push & pull



1 branch ahead

git branch

— current branch

git push origin main

Compose with main → No conflict → Automatically merges

Conflict - push fails

good practice: pull before push [~~dev~~ Programmers - may changed].

need to update - locally.

Conflicts

* merge - (automerge) - no conflict, if there - end user need to resolve (merging/rebasing) - validate & resolve.

Say: 2 dev, 2 branches - Same file - Features.JS → feature 1

New feature to existing method → feature 2

Both: pushing changes

dev 1

git pull

git checkout -b feature 1

git add feature.JS

git commit -m 'Chore: feature 1'

git pull origin main

git push -u origin feature 1

Reviewed pull request, merged to main

dev 2

git pull

git checkout -b feature 2

dev 2 - Codes

git add feature.JS

git commit -m 'feature 2'

git pull origin main

Failed

Working file - changed!

Conflict

git status → gives current branch

git log --merge → merging logs

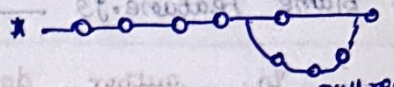
(where is the difference)

Workflow

Remote Repo ⟷ Pull (checkout) / Push (changed) ⟷ Local machine

Feature branching (Common):

* From main – work till end



Ready: merge! — pull request

Note: Before new branch – Sure: latest code

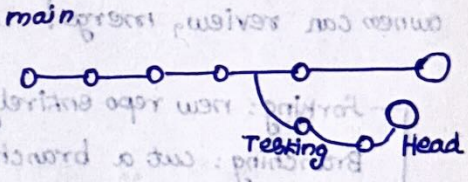
1. git pull
2. git checkout -b feature/my-new-feature
3. git add . / git add Readme.md
4. git commit -m "chore: new branch with Readme file" [local]
5. git push -u origin feature/my-new-feature.

Head

- git → track of changes

Head → Current Commit (viewing)

1. cd .git
2. cat HEAD [git: work on single branch at a time]
3. cat /refs/heads/main
4. a2cd.... [id: current Commit] ref id
5. git checkout testing
6. git branch
7. * testing



Any change – id changes after Commit.

Diff Command

1. Revisit old code – Compare changes. (what changes are)
2. git diff HEAD README.md

HEAD ↓ Now — README.md ↓ want to Compare.

Red – removed
green – added.

git log --pretty=oneline ⟶ take any 2 logs

↓ oneline o/p

git diff 8b559... a93326....

↓

Shows what happened b/w 2 versions

Compare against branches

Blame

1. Keep track of everyone – git (Blame) – who did that – when – view history

git blame feature.js ⟶ list of all changes

id author date & time whose change made (line number) content

---

git blame -L 5,15 setup.py

↳ Line 5, Line 15 (start & end)

git blame -l setup.py

↳ Full id (not reg short form)

git log -p id (particular log)

---

git blame → points whose changed ?

git log ⟶ what content changed ?

Forking

Forking: new repo entirely – Another type of workflow

entire content & history ⟶ saved (free to do anything)

owner can review, merge!

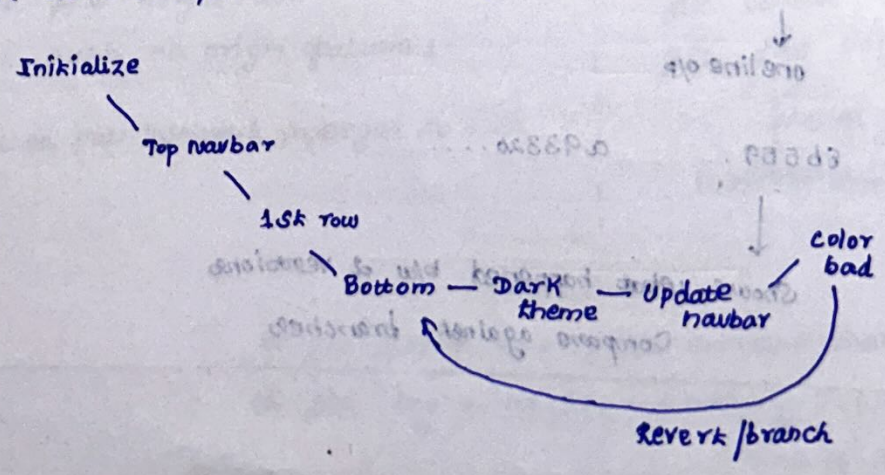
Forking: new repo entirely – clown the repo in own own Github.
Branching: cut a branch everytime
↳ credits given (whose forked!

---

green – added (+)
red – Removed (-)

Git & Github – Hands on Bootcamp

* Compare changes b/w versions, revisit earlier version, restore
* CVS, subversion, Mercurial, Git
* 90% GIT (free, opensource) – DVCS
* Git: Game (Savepoints), Revert [checkpoints]



→ Why Git : Linus Torvalds - existing ones (slow, closed-source, paid) - Create one!

→ Git : (Random 3 letter combination), Not UNIX Command, God damn Idiotic Truckload of Sh*t (when it breaks)

Global Information Tracker

Who uses GIT

* Meta, Google, Skaankups, dev, ...

* Tech adjacent roles, Governments, organizations, scientists, writers (some), keep daily diary to draft phd theses, track photoshop files - uses GIT.

*

GIT ≠ GITHUB

* GIT - locally (machine) - no reg account - offline

* GITHUB - Host repo (cloud) - need account - online - share work using GIT

GUI → Github Desktop
→ Gitkraken
→ Ungit
→ Source tree
→ Tower

pros: beginner, friendlier, visual
cons: magic (abstract), dependance (Software), No fix (Command line), Diff GUIs.

GIT Bash

* windows - Not UNIX based - GIT Bash (emulate bash experience)

*

Configure name & email

Name → shown in log

```
# git config --global user.name "____"
# git config --global user.email "____"
```

GITKraken * Free * visual - nice.

Terminal Crash Course

Navigation

start . → open file explorer (current path)

File management

touch filename → empty file create/update timestamp.

rm -rf Plants (folder)
↳ recursive force.

1. ls
2. touch
3. git add
4. git commit
5. cd..
6. git log
7. pwd

GIT

* Git repo : workspace (within a folder : manages files, track) – history own, Contents.
* git status (report status of repo : Committed / Staged / clean?)

one time : git init ⟶ # Initialize a git repo (enables git)

.GIT (hidden)

* where tracking, history done

mistake : one repo include : Subfolders (Can't be seperate repo)

Note : Don't do git init Inside repo
↳ Just delete it (happened)

---

add, Commit, Push

1. checkpoint (Commit) / Savepoints. – message + Snapshot
2. made changes (diff files) → group them ⟶ Commit

Stage area —git commit→ Repo (local) —push→ Remote

git add

---

git add file1 file2 ...
git commit -m "message"

* message – explain why this Commit
* # git status (author, email, timestamp)

git add . (all) – better be specific

Atomic Commits

* Single feature / change / fix : Small thing
* why : easy to undo / roll back – easy to review.

Commit messages

doc recomm : * present tense – Imperative mood (Fix this ...) eg

VIM ⟶ insert mode (i) → type ⟶ :wq (write quit) ⟶ enter
↑ git commit

default editor

.bashrc manual :

--wait → VS Code

git log

Pd → specially visit one. #git log

Formatting:

--pretty=oneline --abbrev-commit (or) --oneline

GUI

* Staged area → visual (no Commands)

Amending Commits

* made a Commit → forgot a file / typo in commit message

No need to brand new Commit — redo — previous one

→ git commit -m 'some content'

→ git add forgotten_file

→ git commit --amend → previous commit

Can't redo other Commits

Ignore files

*Tell git which files & directories to ignore in a given repo,

* .gitignore file

* Use: Never want to commit (secrets, API Keys, Credentials, OS files, Log files, dependencies & packages)

.gitignore (root of repo):

write patterns to tell Git - which files to ignore (wildcard)

- .DS_Store → Named .DSStore
- foldersName/ → Ignore directory
- *.log → Ignore all log files

Ref doc: for patterns.

Example:

secrets.txt

node_modules/

gitignore.io

↓

suggests Commonly Ignored ones

Branching - Signature feature of GIT

* Each Commit has unique hash

* won't impact other branches - until merged (Isolation)

Master branch

* master - default branch

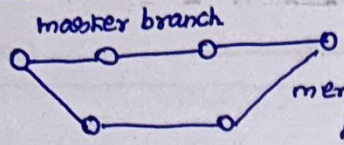
* Source of truth - main branch (official)

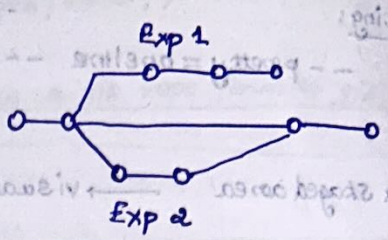
* Test in other branches - merge to master.

In 2020 – GitHub

* Renamed master to main

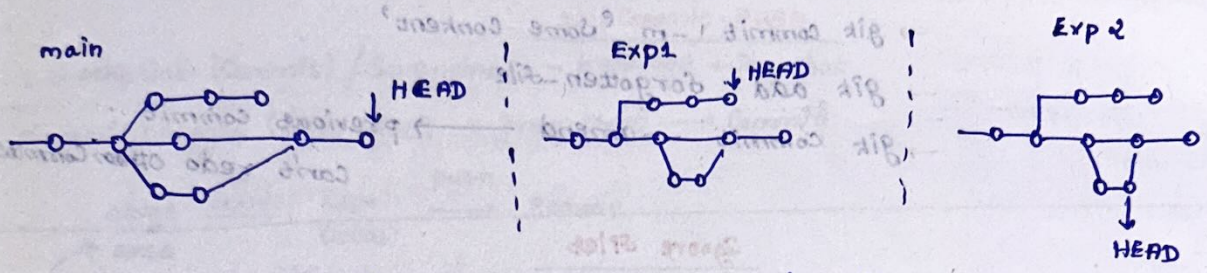
Common workflow:





HEAD → master

* What is HEAD → pointer to current branch location
* only one branch active! – current location



master → points to latest commit of master branch

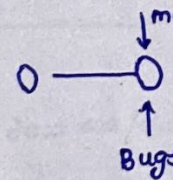
View branches

git branch → shows all branches

Create & switch branches

git branch <branch-name> → doesn't switch to the branch

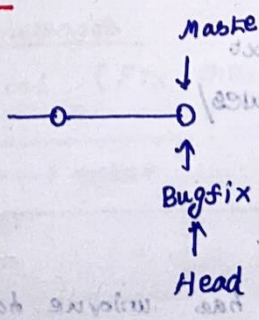
git branch bugfix



∴ Not switched yet

Switch

git switch bugfix



∴ HEAD updated.

More branching

checkout (vs) branch

1. checkout creates & jumps (switches) to that branch

git checkout <branch-name>

1) # git switch -c <branch-name>

Create & switching at once

Switching branches with unstaged changes

* Current branch — unstaged files
* Now creating another branch - what happens?

git switch some branch (Already existing)

Your local changes to the following files would be overwritten (lost)
please commit before switching.

Commit / Stash (undo) ⟶ options!

Scenario 1

playlist file ⟶ main, oldies branches
Shares this

1. In oldies — edit playlist.txt
2. git switch emptyplaylist

(not added/ Committed) ↓

error (Commit / Stash)

why: shared file b/w branches.

Scenario 2.

git switch -c chicken
touch chickens.txt

now not added / committed
try to change branches

git switch oldies

No problem occured!
Switched to oldies

why? : Advise: Commit before branching
here: chickens.txt ⟶ not shared
(unique to this branch)

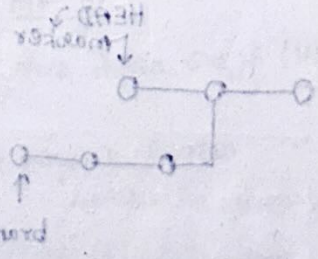
I'm in some branch: That branch also) chicken.txt there.

Reason: chicken.txt ⟶ not Committed to any branches
Say: It's still available. Any branch can take it & commit it.

chicken branch
chickens.txt
(not staged)

switch ⟶

oldies
chickens.txt
(will be here)
we can staged
Commit it!



Delete / Rename branches

*Rare!

* --delete, -d] delete branch

git branch Sample
git switch Sample
git branch -d Sample ⟶ Note: can't delete the branch one I'm viewing

Problem: Not completely merged (-d) gives warning

-D ignores that & deletes unmerged branch

-D Shortcut for --delete --force

git branch -m "hello"

-m, --move] move/rename

go to that branch (need to be there)!
git switch Sample
git branch -m Renamed_Sample
git log
git Status

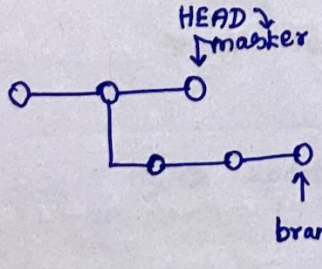
How GIT Stores HEAD & Branches

cd .git
cat HEAD
ref: refs/heads/master
git switch oldies (cd.. after)
cat .git/HEAD
ref: refs/heads/oldies

There is a file for every branch

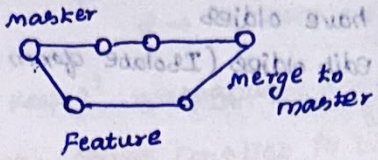
each file has - recent Commit hash

bookmark recent commit.



GIT Merge

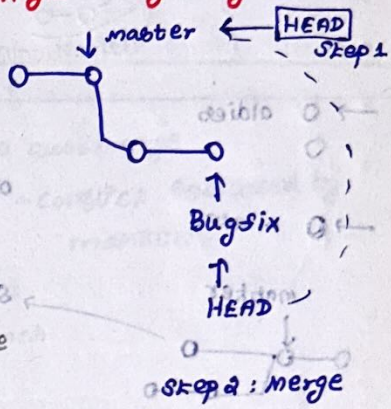
* Branching: Self Contained feature, often: Incorporate changes from one branch into another. (git merge)

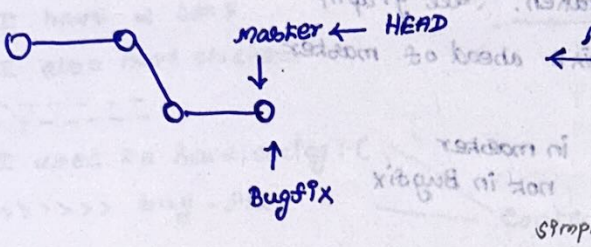


Merging:

* Merge branches — not Commits
* merge to current HEAD branch

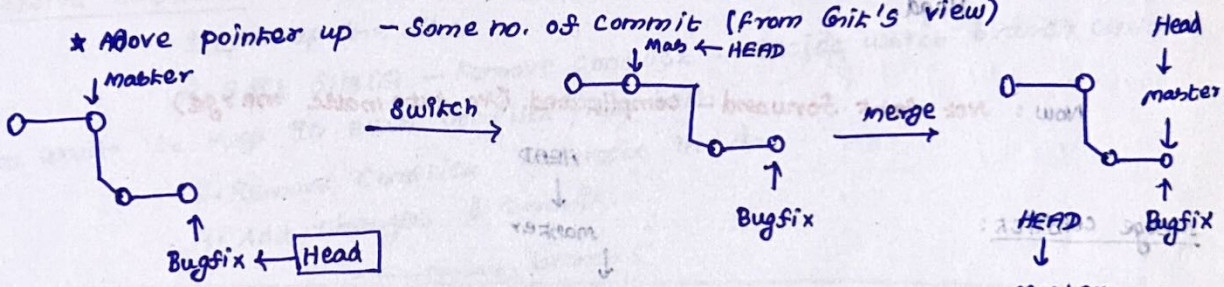
#git switch master
#git merge bugfix



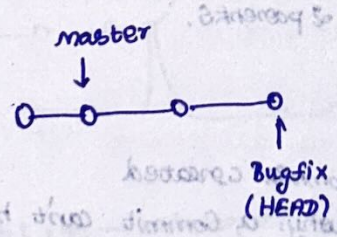


Fast Forward merge (Simple)

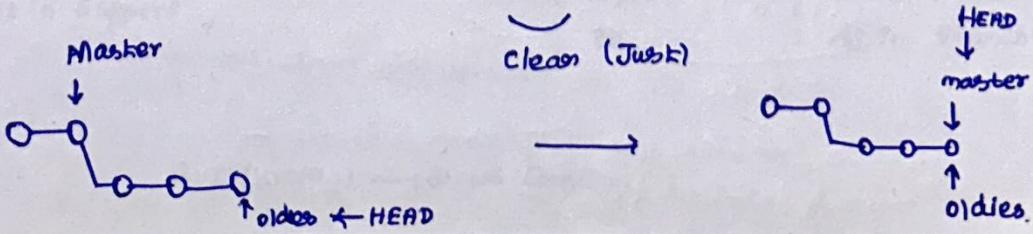
* Above pointer up — Some no. of Commit (from Git's view)



Git's view:



Just move pointers.



oldies (From bottom to top)

Add playlist header
Two ABBA Songs (master)

Add Two George Jones Songs
add Two George Harrison
Van Morrison (oldies)

master

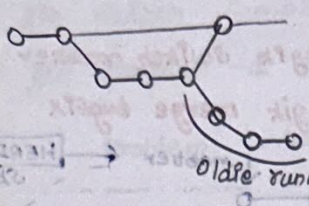
add Two ABBa Songs (HEAD → MASTER)

add playlist header

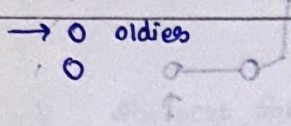
git merge oldies (after switching to master)

git log [Initial 2 commits + oldies commits]

Note: we still have oldies
we can edit oldies (Isolate from main) - run indep.
& merge when necessary

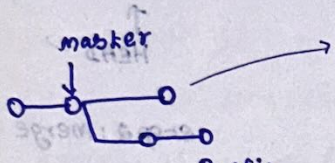


Fast forward merge!

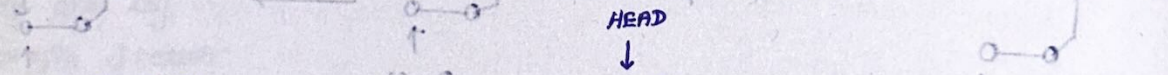


git Kraken. (See graph)

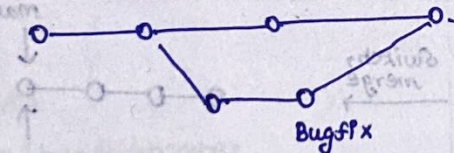
oldies - 2 commits ahead of master

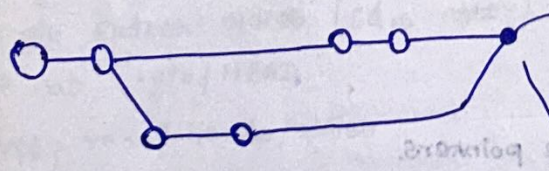


Now: Not fast forward - complicated (No Automatic merge)



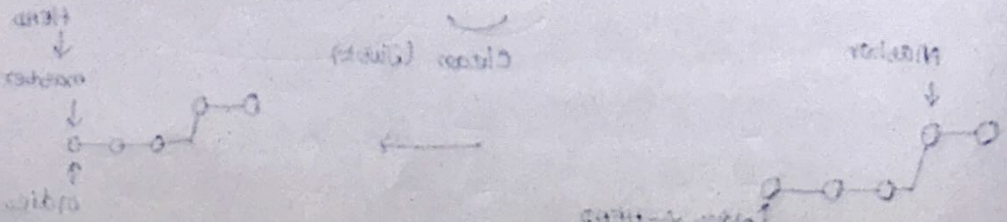
Merge conflict:





log:

2 main Commits
2 ABBA commits
2 main commits
1 merge Commit



Merge conflicts

* If no automatic merge available:

eg: 1 → edited a file | 1 → 77th line edited
2 → Deleted a file | 2 → 77th line edited

"what to keep?" → manually resolve.

Conflict (content): Merge conflict in blah.txt
Automatic merge failed, fix conflicts & then commit the result.

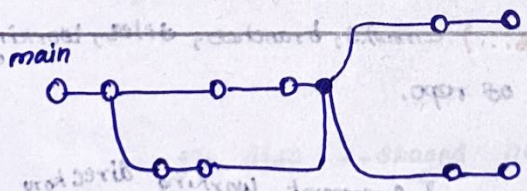
<<<<< HEAD
I have 2 cats
I also have chickens
- - - - - - - -
I used to have a dog :C
>>>>>> bug-fix

→ Conflict: No automerge
(Resolve - conflict indicated by markers)

Content from branch
=== and >>>>

Resolve Conflicts:

1. open up the files
2. Edit file(s) - Remove conflict - Decide which branch content you want to keep in each conflict / keep both.
3. Remove conflict markers in doc
4. Add changes & commit.



branch 1, 2 → Seperate (Same origin point)

* Merge: Conflict due to editing same file.

merge branch2 to branch 1.

git merge Seema → Conflict

Go to file → edit → Commit → Now merge!

VSCode to Resolve Conflict

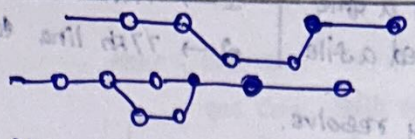
Built in Support

Current change: as in master
Incoming : As in branch

GUI (Kraken) → Shows Conflict.

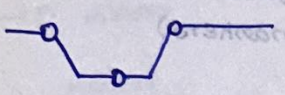
Note: Merge Conflict → once Committed after editing : Automerged (no need to run merge again)

1. Fast forward
2. No conflict
3. Conflict
   → conflict resolving then merge commit.



1. Fast forward

* Create greetings.txt
* Create branch french
* Edit greetings.txt in french branch
* Merge



2. No Conflict

* Create branch Spanish
* edit greetings.txt in Spanish
* Create farewell.txt in master
* merge

'No Conflicts'

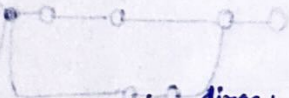
3. Conflict

1. Create Japanese branch
2. edit greetings.txt
3. master: edit greetings.txt

Resolve Conflict.

git diff

1. What changed b/w two things (2 versions...) commits, branches, files, working dir. Along with git log, status — better view of repo.



git diff → compare staging area & Current working directory

Reading diffs

red
orange
yellow
green
blue
purple

Last commit

Staged
red
orange
yellow
green
blue
Indigo
Violet

git diff

@@ -3,4 +3,5 @@

yellow
green
blue
orange — black color
- purple → Red
+ Indigo
+ violet — Green

diff --git a/rainbow.txt b/rainbow.txt → 2 file names (versions)
index 72d1d5..f2c8117 100644
hash
Internal file mode Identifiers

--- a/rainbow.txt
+++ b/rainbow.txt
changes in a: -
b: +

Show only changed lines → chunks. (areas/portions of file).

-3,4, +3,5

A B

- (File A)
+ (File B)

file : how many lines extracted.

-3,4 → File A 4 lines extracted from line 3

+3,5 → File B 5 lines extracted from line 3.

View unstaged changes

```
git diff
git diff HEAD
git diff --staged / --cached
git diff HEAD [filename]
git diff --staged [filename]
git diff branch1 ... 2
```

1. git diff → Compare staging & working directory. (uncommitted), unadded.
2. git diff HEAD → made since HEAD (staged & unstaged changes)

Once committed: HEAD changed

unadded, added to stage area not committed yet

Staged / cached changes

```
git diff --staged
git diff --cached
```

staging area & last commit

Specific files

```
git diff --staged numbers.txt (specific files)
```

Across branches

```
git diff branch1..branch2

git diff master.. odd-numbers
git diff odd-numbers.. master
git diff master add-numbers
```

.. /space

Commits

```
git diff commit1 .. commit2
```

hash codes

GUI tool:

use shift to select multiple!

HEAD~1 → parent of head

git diff HEAD HEAD~1 (or) git diff HEAD~1

(or)

use commit hash

Stash

* Convenient method – more usage.
* use: make some commits & switch to new branch
  do some work, need: go & visit master (uncommitted)
  1. changes comes to destination (unstagged) branch
  (Conflict: overwritten scenario) 2. GIT won't allow – until committed (potential conflict)
* stash for rescue: save without committing

Stashing: easy way of stashing – uncommitted changes so that – return later, commit later (no need for unnecessary commits: instantly)

git stash [remembers them – retrieve from stash when needed]

git stash pop → pop recent one!
(Reapply to same / diff branch)

Branch: bugfix

working directory	staging area	Repo
modified nav.css	mod index.js	0026739
modified nav.js	footer.js	bb43f1f

Something: master (Jump to)

1) stash & go to main/master
2) Later undo by pop later!

Stash
- mod nav.css
- mod nav.js
- mod index.js
- mod.footer.js

* switch how to other branches, work on it, commit
* look at coworkers changes
* when done revisit branch – reapply (pop) – stash.

git stash apply

1. pop – removes stash once restored.
2. apply – No removal (multiple branches) – we can use it.

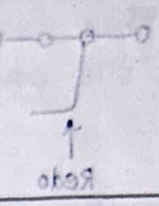
Conflict may arise – like merging

1. Manual resolution! (fix conflict)

Multiple Stash (again & again)

1. Stack (Same way!) — Last In First Out (LIFO)

`# git stash list` → view stack!



Apply : (not removed) : `# git stash apply stash@{2}`

Drop stash

* pop removes most recent one

`# git stash drop stash@{2}`

clean stash (completely)

`# git stash clean.`

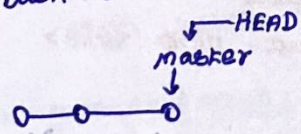
Time travelling

`# git checkout hash` [view : You're in a detatched HEAD state. Look around, make changes & commit, discard any commits : In this state - won't impact any branches by switching back to a branch.]

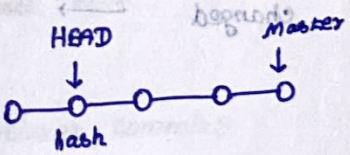
Use: Jump back to that Commit (Look around)

log - only shows commit upto that point

Look - files at that stage! (Once jump to other branch - back to business).



manually → HEAD repoint to that commit



DETATCHED HEAD

Detatched head: one detachment from current commit of current branch.

Reattach HEAD

Detatched head:

1. Examine contents of old commits, look around
2. Reattach head. — now we can do changes (as reattached)

`# git switch master`

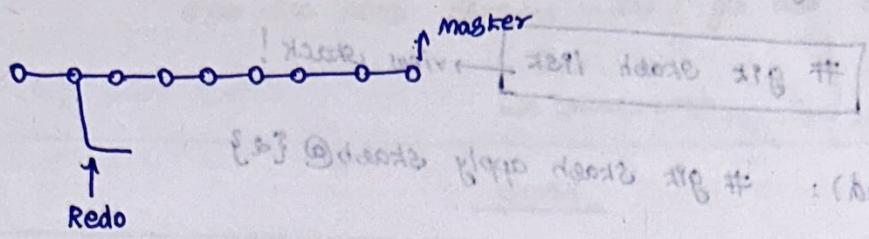
`# git checkout 4737c78`
`# git log`
`# git switch master`

can I branch in head detatched state? *

Yes!

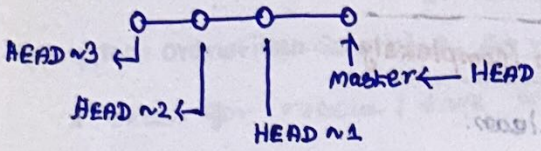
`#git checkout 4737c78`
`#git switch -c "Redo"`

As I reattached (using branching)



Referencing Commits relative to HEAD

HEAD ~1 (1 Commit before HEAD)



git checkout HEAD ~1

git switch - [back to previous branch]

Discard changes

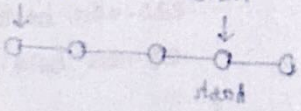
★ Revert previously committed

```
# git checkout HEAD <file>
```

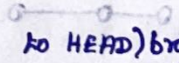
★ discard any changes in that file, Revert back to HEAD

edit:

changed ⟶ Say: I don't wanna commit



1. manually delete
2. git checkout HEAD <file>



git checkout HEAD dog.txt (undo changes to HEAD)

git checkout HEAD cat.txt

(or)

git checkout -- <file>

---

git restore

★ Reduce burden on 'checkout' Command

★ git restore <file-name> ⟶ Restore to HEAD version (recent Commit)

★ Note: uncommitted changes ⟶ Lost forever!

Note: default: HEAD

but we can restore to previous commits.

#git restore --source HEAD~1 app.js

↓

1 commit before HEAD (Restore to that point)

Note: HEAD remains at same position. But what happens is the file content restored with source version's content.

#git restore --source HEAD~2 cat.txt dog.txt

↑ 1. Unmodify with Git Restore

2. Unstage changes with Git Restore

git restore --staged <file-name> (Same as restore to HEAD in some way but only undo staged) - except content remains same (not restored)

Staging area

Cat.txt
dog.txt
Secrets.txt

unstage → Secrets.txt

#git restore --staged secrets.txt

↓

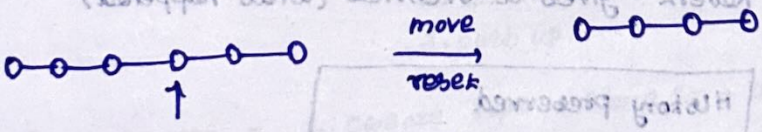
only unstages not edits content.

Note: git status reminds what to do? (unstage/stage)

Undo commits — Regular (plain) Reset / Hard Reset

#git reset <commit-hash>

* Reset master branch to a particular commit - Deletes commits



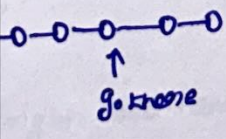
Why: Say - Realize must taken a branch 2 commits before.

Note: only done commit-level

Need to do content-level

'get rid of commits'

Keep changes - don't undo content! → If want to keep the changes commit it to a branch/master - else changes also resetted.



git reset HEAD~2

git branch temporary

git add.

#git commit -m "add dangling content"

now in main

'Content restored to HEAD~2' content.

* Note: when reset done: it gives unstaged changes - we can get to new branch then commit / commit to master!

---

git reset _____ → Removes commit but not working content

(It's in unstaged form - stage it / bring to other branch)

hard reset

git reset --hard <commit>

*Both undo commit & content → can't able to stage changes

* Changes gone!

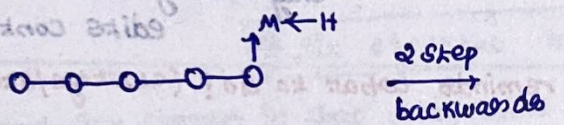
---

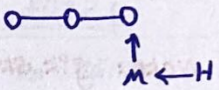
git revert

1. Similar to reset - slight difference.

Reset: 2. Branch moves pointer backwards - eliminates commits

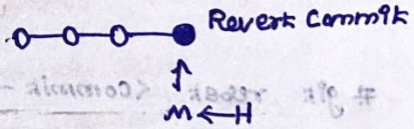
Revert: 3. Creates brand new commit - which reverses / undos the change from the commit. (Enter commit message)





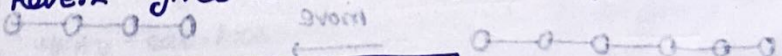
"Reset"

git revert <hash commit>



Useful: we can't just delete - others know nothing!

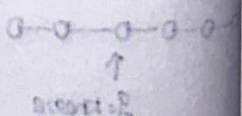
using Revert - gives a overview (what happened)



History preserved
changes reversed (content)

use: collaboration (want to reverse some commits: other people already have in their machines → should revert

haven't shared - who cares to history edited → use reset



Github – very popular (56+ million users)

* Hosting site – git repos (access remotely, collaborate)
* Additional features
* gitlab, bitbucket, gerrit
* Free – github

Why use?

1. Atleast 1 dev – hobby project (Global – Github)
2. open source projects (React to swift)
3. Exposure (other's project), hiring, gain friends.
4. up to date, debate.

cloning repos

* git clone <url> → Retrieve all files & init git repo
*

SSH registration

pushup code: need authentication (ssh key – onetime)

ls -al ~/.ssh → any previous ssh (use documentation)

open ~/.ssh/config → AddKeysToAgent (To github) → Yes
UseKeychain (passcode) → Yes
IdentityFile ~/.ssh/id_ed25519 (default algo)

or use: GUI

Repo

1. Existing repo – locally – 1. create repo on Github
   2. connect to local repo (add remote)
   3. push up.
2. From Scratch – 1. create repo – clone it – work locally – pushup

Git remote

1. destination URL (whose hosted repo lives)

git remote -v

fetch
push

Make a remote

git remote add <name> <url>

origin: std name

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

Origin: Conventional name (not special) - default name setup (we can change it)

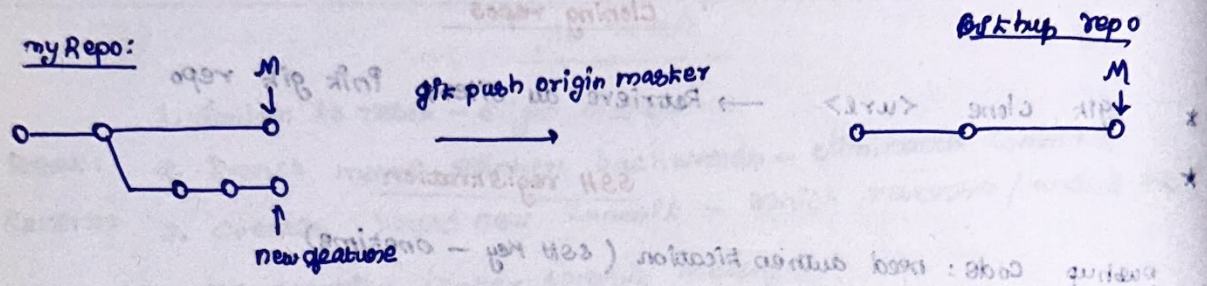
git remote rename <old> <new>

git remote remove <name>

Push

* git push <remote> <branch>
* git push origin master : Common one

Rename master to main then push (If want)



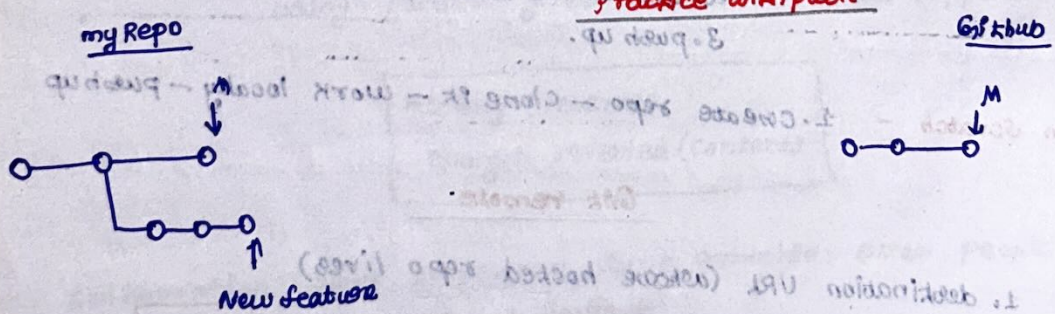
git push origin empty (branch)

Touring github repo

* Switch branches
* Show Commit with time, show messages of Commit.
* See commit log!

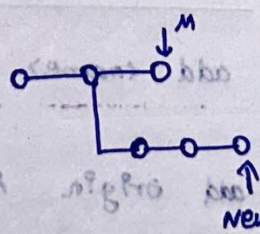
See diff → what's changed

practice with push



Now push

git push origin newfeature



More git push

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

git push origin pancake : waffle (Not common: Sometimes push master branch to main branch of github).

Cats : master

local → github

-u flag

"upstream" — Set upstream (Link b/w branch(local) to github branch)

```
# git push -u origin master
```

Set upstream - So that it tracks master branch on original repo.

-u : upstream : Recommended

Next time we can just do `git push` alone.

If we don't do -u	If we do -u
1. No connection	1. Just : git push (Connected)!
2. every time : git push origin master	

git push → fails here.

>> Set upstream (Connection) : Just do git push.

git push -u origin pancake : waffle → extremely useful (each time won't need to mention which local branch : github branch.

From Scratch - github → create repo clone, push

main (vs) master

1. default github done Readme.md → main
2. git → master.

If needed : Just rename!

Rename : git branch -M main

Go to setting → change default branch

Fetching & pulling

1. others people changing master / branches — bring that!