GIT:

**1. GIT**  git clone ---- means you are making a copy of the repository in your system.

**git fork** ---- means you are copying the repository to your Github account.

**git pull** ---- means you are fetching the last modified repository.

**git push** ---- means you are returning the repository after modifying it.

In layman's term:

git clone --- Is downloading and git pull is refreshing.

**2.** **To push the current branch and set the upstream , use**

git push –set-upstream origin <branch-name>

**3. Git Commands**

**git init** -----> To initialize a git repository

**git config --global user.name <username>** -----> To set username

**git config --global user.email <email>** -------> To set an email address

**git --list** ----> To display all configurations done on git.

**git init** ------> TO initialize a git repository

**git add .** -----> To add files to repository ( To staging)

**git add --all** ---> TO add all files to the repository

|\_\_> add files to the staging area

**git add <file\_name>** -----> add files one by one specifically using

**git add /<folder\_name>/** ----> add contents of a particular folder

**git status** ------> To see status of unstaged or uncommitted files or directories (untracked files) --> to show current state of repository

**git commit -m "message"** ----> to commit changes to the repository --- add changes to local version control system

**git diff --stages** ----> To compare local directory repository with the staging area

**git diff HEAD** ---> TO comapare working directory with the local repository

**git diff** ---------> Comapre changes of working directory with the staging area

**git diff commit\_ID commit\_ID** ----> TO comapre 2 commits

**git clone** ---- means you are making a copy of the repository in your system.

**git fork** ---- means you are copying the repository to your Github account.

**git pull** ---- means you are fetching the last modified repository.

**git push** ---- means you are returning the repository after modifying it.

In layman's term:

**git clone** --- Is downloading and git pull is refreshing.

**git ls-tree** ----> returns a tree object representation of the current repository along with the mode and the name of each item

git log ----> It shows what changes you performed on this project so far

git checkout <file\_name> -----> helps you to roll back the changes you made to the file

git branch ----> shows list of branches

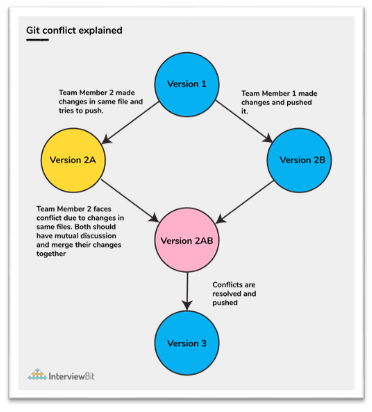
git branch -b <branch\_name> -----> to create a new branch

git checkout <branch\_name> -----> to switch between branches.

Git delete -d <branch\_name> ------> TO delete a merged branch   
git delete -D <branch-name> ----> To delete a branch that is not merged

**4. What is a conflict?**

* Git usually handles feature merges automatically but sometimes when we are working in a team environment, there might be cases of conflicts such as:  
   1. When two separate branches have changes to the same line in a file  
   2. A file is deleted in one branch but has been modified in the other.
* These conflicts have to be solved manually after discussion with the team as git will not be able to predict what and whose changes must be given precedence.  
  Steps:  
  git status -- Look for files marked as **"both modified"** — these have conflicts.  
  Open and manually resolve conflicts  
  Delete the markers and combine the code as needed.  
  git add <file-name>  
  git commit



**5. Difference b/w git pull and git fetch**

|  |  |
| --- | --- |
| **Git pull** | **Git fetch** |
| git pull retrieves the latest changes from the remote repository like git fetch, but it also automatically merges those changes into the current branch | downloads the changes from the remote repository to the local repository but does not make any changes to the current working directory |
| git pull = git fetch + git merge |  |

**6. Git reset --hard** -----> Uncommitted changes will be remove by using git reset –hard   
Ex: in a git local repo à created files file 1, file 2 then ----> then I accidentally added then to staging ----> Now if I want to remove these file I should use git reset –hard == this will remove uncommitted changes and takes you to the old state

**7. purpose of the .gitignore file?**

TO hide files from git from tracking we can update the file name in .gitignore file then git will not track those files.  
Ex: log files, temp files

**8. Git reflog** -----> to get the commit ID of the lost commits

**9. Pull request:**

- Assume I have taken a code from another account repository  
- Then I have found a bug & fixed it   
- Now I want the original code from where I got this code to get updated with this bug fix.  
- Then I will raise a PULL REQUEST to him to update his code.

**10. git cherry-pick ------> It says picking the commits**  
 - git cherry-pick <commit-ID> ---à to add the commit to the main branch.

**11. Difference b/w git merge and rebase ?**

Git merge -----> all the changes will get committed & will be at the top

Git rebase -à all the changes will be updated before your main branch commits   
 if there are 10 commits of main branch then rebase changes will be after the 10 commits.

**12. How do you revert a commit that has already been pushed and made public?**

To revert a commit that has been pushed and made public, follow these steps:

* Checkout the Branch: Switch to the branch where you want to revert the commit.

**git checkout <branch-name>**

* Find the Commit to Revert: Use 'git log' to find the commit hash of the commit you want to revert.

**git log**

* Revert the Commit: Use 'git revert' followed by the commit hash of the commit you want to revert.

**git revert <commit-hash>**

* Review Changes: Git will open your default text editor to confirm the revert message. Save and close the editor to proceed.
* Push the Revert: Finally, push the reverted commit to the remote repository.

**git push origin <branch-name>**

How do you resolve a merge conflict and what commands we use   
**What Is a Merge Conflict?**

A **merge conflict** occurs when **Git can’t automatically resolve differences** between two branches — typically when the same line of code has been changed differently in both branches.

## **🔧 Steps to Resolve a Merge Conflict**

Let’s say you're on the feature branch and trying to merge main into it:

bash

CopyEdit

git checkout feature  
git merge main

If there’s a conflict, Git will pause the merge and show:

pgsql

CopyEdit

Auto-merging file.txt  
CONFLICT (content): Merge conflict in file.txt  
Automatic merge failed; fix conflicts and then commit the result.

## **🪛 1. Identify Conflicted Files**

Use:

bash

CopyEdit

git status

It will list the files with conflicts like:

scss

CopyEdit

both modified: file.txt

## **📝 2. Open and Edit the File**

In file.txt, Git adds conflict markers like:

txt

CopyEdit

<<<<<<< HEAD  
your changes (on feature branch)  
=======  
incoming changes (from main branch)  
>>>>>>> main

Choose which lines to keep, or combine both:

txt

CopyEdit

final version of the line after resolving conflict

Then save and close the file.

## **✅ 3. Mark Conflict as Resolved**

Once you've edited the file:

bash

CopyEdit

git add file.txt

## **🧾 4. Commit the Merge**

bash

CopyEdit

git commit

Git may open a default merge commit message — save and close it to complete.

## **📌 Optional Commands**

### **To abort the merge entirely:**

bash

CopyEdit

git merge --abort

### **To view conflicts more easily:**

git diff

Or use a GUI tool like VS Code or git mergetool.

## **🔄 Summary of Commands:**

|  |  |
| --- | --- |
| **Step** | **Command** |
|  |  |
| Start merge | git merge <branch> |
| View conflicts | git status |
| Resolve conflicts manually | Edit files |
| Mark as resolved | git add <file> |
| Commit the merge | git commit |
| Abort merge (if needed) | git merge --abort |

Git cherry pick

Cherry picking is a technique that allows you to apply a specific commit from one branch onto another without merging all the changes from the source branch. This is useful when we want to add a change from another branch without adding all its commits.  
  
Example: If we are working on the master branch and we identify a single commit in a feature branch that can fix a issue on master. In this scenario we can use cherry picking to import that specific change without merging the entire feature branch.  
**-----------------------------------------------------------------------------------------------------------------------------------**

**Git revert:**  
 If you want to reverse the effects of a commit without rewriting the project history, the **git revert** command is used   
**Example**:   
A teammate accidentally pushed a faulty commit (abcd123) that introduced a critical bug, breaking the application's login feature. How you would safely undo the changes from abcd123 without rewriting Git history  
$ git revert <commit-hash>  
**-----------------------------------------------------------------------------------------------------------------------------------**

**Git reset:**  
$ git reset --soft HEAD~1  
A soft reset moves the branch pointer back to a previous commit without altering the working directory or staging area. This approach lets you review and re-commit the changes if needed.   
**Example:**   
**You’ve just committed changes to your Git repository with the message feat: add user profile page, but you realize you forgot to include a critical file (user-avatar.css). How would you undo the most recent commit (abcd123) while keeping your changes staged so you can amend the missing file?**

**-----------------------------------------------------------------------------------------------------------------------------------**  
$ git reset --hard HEAD~1   
a hard reset moves the branch pointer and discards all changes in the working directory related to the commit.   
  
**-----------------------------------------------------------------------------------------------------------------------------------**

|  |  |  |
| --- | --- | --- |
| Command | Description | Example |
| git revert <commit> | Creates a new commit that reverses changes while preserving commit history. | **$ git revert 8ad5d** |
| git reset --soft HEAD~1 | Moves the branch pointer back one commit, keeping changes in the staging area for review. | **$ git reset --soft HEAD~1** |
| git reset --hard HEAD~1 | Moves the branch pointer back one commit and discards all changes in the working directory. | **$ git reset --hard HEAD~1** |

Adding a Remote Repository to Your Local Project  
git remote add origin https://.../[name].git   
  
Verify the remote connections:  
git remote –v  
**-----------------------------------------------------------------------------------------------------------------------------------**  
git stash  
Lets say we have 1 ,2,3 changes currently I am working on the 3rd change, but some how one of my collegue notices a typo or a misktake on the first change and it needs to be immediately fixed.  
In this case we can stash our work InProgress chages in a stack-like structure  
using   
 git stash pop   
 git stash list -----> To view all the cahnges in your stash   
 git stash show <stash\_identifier> ----> to inspect a specific stash   
 git stash pop <stash\_identifier> ---> to restore a specific stash.  
  
Git reflog ----> shows a detailed history that includes even those actions which don't create new commits.  
  
  
Note

While **git log** shows the history of commits, **git reflog** provides an exhaustive record of all repository state changes, including those that do not result in new commits. This makes it an essential tool for troubleshooting and recovering lost work.  
  
git log:  
The git log command displays your project's commit history directly in the terminal. It provides essential details for each commit, including the commit hash, author name, commit date, and commit message.   
**-----------------------------------------------------------------------------------------------------------------------------------**Repository Types

* **Local Repository**:  
  Stored on your machine, it gives you direct control over your project files and facilitates rapid development. This repository is where you perform your day-to-day work.
* **Remote Repository**:  
  Hosted on a centralized server, it serves as a backup and a shared workspace for the team. Teammates can clone the remote repository, work on their own local copies, and then push their changes back. Additionally, pulling updates from the remote repository ensures all local copies remain synchronized.

**-----------------------------------------------------------------------------------------------------------------------------------**  
Git HEAD:  
In Git, HEAD -- It always points to the latest commit on the branch you are working on. Changing branches moves the HEAD pointer to the tip of the target branch

### **Rename the current branch:** git branch -m new-branch-name

### **Rename a different (not currently checked out) branch:**

git branch -m old-branch-name new-branch-name

### **If you've already pushed the branch to remote and want to rename it there too:**

1. **Rename the local branch:**

git branch -m old-branch-name new-branch-name

1. **Delete the old branch on remote:**

git push origin --delete old-branch-name

1. **Push the new branch and set upstream:**  
   git push origin new-branch-name  
   git push --set-upstream origin new-branch-name

**-----------------------------------------------------------------------------------------------------------------------------------**

**The CI build getting slow due to deep history how do you create a light weight git clone to improve the CI build**

To speed up your CI builds by avoiding a deep Git history, you can do a **shallow clone** — which limits the history and greatly reduces clone time and size.

Use --depth to create a shallow clone:   
 git clone --depth 1 <repo-url>

Clone specific branch only:   
 git clone --depth 1 --branch <branch-name> <repo-url>

**-----------------------------------------------------------------------------------------------------------------------------------**

**One of the team mate did some force push and deleted critical commits.So how can we recover lost history in git**  
  
When someone does a **force push** (git push --force) and deletes commits from a shared branch, recovering them depends on **whether those commits still exist in someone’s local clone or reflog**.

Here are step-by-step methods to **recover lost Git history after a force push**:

*Check your local reflog (if you had the commits before):*

$ git reflog

*This shows all recent branch positions (including before the force push). Example output:*  
  
Example output:

abc1234 HEAD@{0}: checkout: moving from feature to main  
def5678 HEAD@{1}: commit: Fixed payment bug

*You can then* ***reset to a previous commit*** *using:*  
$ git reset --hard <commit-hash>

*Or create a new branch from it:*  
$ git checkout -b recovery-branch <commit-hash>

**-----------------------------------------------------------------------------------------------------------------------------------**

Q) You made a commit that introduced a bug. You want to keep your changes but fix them before pushing. What Git command will you use?  
Q) How do you reset a commit to a previous commit without losing changes in the working directory?

**git reset --soft HEAD~1** (Undo the commit but keep changes staged)  
git reset --mixed HEAD~1 (Undo the commit and unstage the changes)  
  
**-----------------------------------------------------------------------------------------------------------------------------------**

A teammate pushed a bad commit to main. How do you revert that commit without changing history?  
git revert <commit-hash>  
Creates a new commit that undoes the changes, preserving history.  
**-----------------------------------------------------------------------------------------------------------------------------------**

### **You Did a Wrong git push --force**

**Question:** You accidentally force pushed and deleted someone’s commits. How can you recover?

**Answer:**

* Check your git reflog
* Find the lost commit hash
* Create a new branch from it:

git checkout -b recovery-branch <commit-hash>  
**-----------------------------------------------------------------------------------------------------------------------------------**

### **Debug Why CI is Fetching Full History** **Question: The CI pipeline is slow due to a deep clone. How to do a shallow clone?**

**Answer:**

git clone --depth 1 <repo-url>  
**-----------------------------------------------------------------------------------------------------------------------------------**  
**Scenario: Clean Local Changes**  
**Question: You want to remove all untracked and modified files (like git clean -df but also reset modified files). What do you use?**

**Answer:**

git reset --hard  
git clean –fd  
**-----------------------------------------------------------------------------------------------------------------------------------**

### **Scenario: Who Changed a Line in a File?**

**Question:** You want to find out who made a specific change in a file.

**Answer:**

git blame <file-name>  
**-----------------------------------------------------------------------------------------------------------------------------------**

### **Scenario: You Want to Work on a Feature Without Affecting Main**

**Question:** How do you start a new isolated branch?

git checkout -b feature-branch  
**-----------------------------------------------------------------------------------------------------------------------------------**

### **Scenario: Y**ou Made Changes in main but Meant to Be on a New Branch

Question: How do you move those commits to a new branch?

Answer:

git branch new-branch  
git reset --hard origin/main  
git checkout new-branch  
  
(or)  
git checkout -b new-branch *# Create and switch in one step*  
git reset --hard origin/main *# Only if you need to clean up main*

**-----------------------------------------------------------------------------------------------------------------------------------**

### **Scenario: You Want to Copy a File From Another Branch**

**Question: Without switching branches, you want to get a file from dev.**

**Answer:**

git checkout dev -- path/to/file

**-----------------------------------------------------------------------------------------------------------------------------------**  
**Scenario: Merge Conflict During Rebase**

**Q) : You get a conflict while rebasing. What do you do?**  
Answer:  
# Resolve conflicts in files manually  
git add <resolved-file>  
git rebase --continue

**Q) How do you fetch changes from multiple remotes in Git?**

To fetch changes from all remotes:  
git fetch –all  
  
**Q) How do you configure a Git repository to use a specific user for a particular project?**

To set a specific user for a project:

git config user.name "Your Name"

git config user.email "your.email@example.com"

**git clean -fd:**  
 Removes untracked files and directories from the working directory but does not affect the commit history or staged changes.