A. Getting Started with Git:

---

1. `git init`

- **What it does**: Starts a new Git project in the folder you are in.
- **Why to use it**: When you want to start tracking changes in your project.
- **Example**:
  - Run `git init` in a folder to set it up for Git. This creates a hidden folder `.git` where Git will track changes for your files.
  - After running this, your project is "under Git's control."

---

2. `git clone <repository_url>`

- **What it does**: Makes a local copy of a project stored on a remote Git repository (like GitHub or GitLab).
- **Why to use it**: When you want to work on a project someone else has shared or start working on your code from another computer.
- **Example**:
  - `git clone https://github.com/user/repo.git`
    This command downloads the project from GitHub, including all its history and branches, to your computer. You can now work on the code locally.

---

3. `git config`

- **What it does**: Sets up your personal preferences for Git.
- **Why to use it**: To tell Git who you are (name and email) and set up other options like text editors. This information is attached to your commits.
- **Examples**:
  - `git config --global user.name "Your Name"`
    Sets your name, so everyone knows who made the changes.
  - `git config --global user.email "you@example.com"`
    Sets your email, which will also show in the commit history.
  - `git config --list`
    Shows all the settings you've configured so far.

B. Checking Repository Status:

4.`git status`

- **What it does**: Checks the current state of your project.
- **Why to use it**: To see which files are new, modified, or staged (ready to be committed).
- **Example**:
  - Run `git status` to see:
    - Files you just added but haven't told Git to track yet (untracked files).
    - Files you've changed but haven't saved in a commit yet (modified files).
    - Files that are staged and ready to be included in the next commit.

Think of it like checking a to-do list for your files.

5.`git diff`

- **What it does**: Shows the exact changes you made to files.
- **Why to use it**: To double-check what has been changed before staging or committing.
- **Examples**:
  - `git diff`:
    - Shows changes in files that are **not yet staged** (still being worked on).
  - `git diff --staged`:
    - Shows changes in files that have been staged (prepared for the next commit).

This helps you review your work before saving it permanently.

C. Adding and Committing Changes:

6.`git add`

- **What it does**: Prepares your changes to be saved in the next commit.
- **Why to use it**: To tell Git which files or changes you want to save.
- **Examples**:
  - `git add file.txt`:

        - Adds a specific file (e.g., `file.txt`) to the staging area.
    - `git add .`:
        - Adds **all changes** in the current folder to the staging area.

Think of it like packing items into a box (the staging area) before shipping them off (committing).

---

7. `git commit`

- **What it does**: Saves the changes you staged (using `git add`) into Git's history.
- **Why to use it**: To create a "checkpoint" in your project where you can go back to later if needed.
- **Examples**:
    - `git commit -m "Your message"`:
        - Saves your changes with a short description (message) explaining what you did.
    - `git commit --amend`:
        - Lets you modify the last commit, like editing its message or adding more changes to it.

Think of committing as clicking "Save" on your project, with the added benefit of leaving a note about what you saved.

D. Branching and Merging:

8. `git branch`

- **What it does**: Helps you create, list, or delete branches in your project.
- **Why to use it**: To work on different features or fixes separately without affecting the main code.
- **Examples**:
    - `git branch`:
        - Lists all the branches in your project and highlights the one you're currently on.
    - `git branch new-feature`:
        - Creates a new branch called `new-feature` to start working on something new.
    - `git branch -d old-branch`:

        - Deletes a branch you no longer need, like an old or completed feature branch.

Branches let you keep your work organized and safe while trying new ideas.

---

9. `git checkout`

- **What it does**: Switches between branches or restores specific files to a previous state.
- **Why to use it**: To move to a different branch or undo changes to a file.
- **Examples**:
    - `git checkout main`:
        - Switches to the `main` branch.
    - `git checkout -b new-feature`:
        - Creates a new branch called `new-feature` and switches to it in one step.

Switching branches allows you to jump between different parts of your work easily.

---

10. `git merge`

- **What it does**: Combines the changes from one branch into another.
- **Why to use it**: To bring the work you did on a separate branch (e.g., a new feature) back into the main code.
- **Example**:
    - `git merge feature-branch`:
        - Combines the changes from `feature-branch` into the branch you are currently on.

Merging is like blending two sets of changes into one, so everyone can use the updated code.

---

11. `git rebase`

- **What it does**: Moves your branch's changes on top of the latest version of another branch.
- **Why to use it**: To keep your branch updated with the latest changes from the main branch without creating extra merge commits.
- **Example**:
    - `git rebase main`:

            - Updates your current branch by replaying your changes on top of the latest `main` branch.

Rebasing makes your project history cleaner and easier to follow.

Difference Between Merge And Rebase:

Git Merge

- What it does: Combines changes from one branch into another.
- How it works:
    - It keeps all the commits from both branches exactly as they are.
    - Creates a merge commit to show when the two branches were combined.
- Why to use it:
    - When you want to preserve the history of both branches.
    - Useful in shared projects where multiple people are working on the same codebase.

Example:
If you merge a `feature-branch` into `main`, the history will look like a "fork and join," showing that both branches existed separately before coming together.

Git Rebase

- What it does: Moves your branch's commits on top of another branch, as if they were made after the latest changes on that branch.
- How it works:
    - It rewrites the commit history of your branch to make it look like you started your work from the latest version of the other branch.
    - No merge commit is created.
- Why to use it:
    - To make the commit history cleaner and easier to read.
    - Ideal for personal branches or when working alone.

Example:
If you rebase a `feature-branch` onto `main`, the history will look as if all the changes in `main` happened first, and then your `feature-branch` changes were added on top.

Key Difference

1. Merge preserves the commit history and shows when branches were combined.
2. Rebase rewrites the commit history to make it look linear, as if all changes happened sequentially.

When to Use Each

- Use merge when:
    - Working in a team and want to preserve the full history of changes.
    - You want a clear record of branch merging.
- Use rebase when:
    - You want a clean, linear commit history (e.g., for personal branches).
    - There are no conflicts with other team members' workflows.

In Short:

- Merge = Keep all history and combine changes.
- Rebase = Rewrite history to look cleaner, like all changes happened one after another.

E. Remote Repositories:

12.`git remote`

- ***What it does****: Connects your local project to a remote repository (like GitHub or GitLab).*
- ***Why to use it****: To push and pull changes between your local machine and the remote repository.*
- ***Examples****:*
    - `git remote add origin <url>`*:*
        - *Adds a remote repository with the name "origin."*
        - *Think of it as linking your local code to the remote server.*
    - `git remote -v`*:*
        - *Lists all the remote repositories linked to your project and shows their URLs.*
    - `git remote remove origin`*:*
        - *Removes the link to a remote repository named "origin."*

---

13. `git fetch`

- ***What it does***: *Downloads updates from the remote repository without applying them to your current branch.*
- ***Why to use it***: *To check for new changes in the remote repository without changing your local files yet.*
- ***Example***:
    - `git fetch origin`:
        - *Downloads the latest changes from the remote repository named "origin."*

It's like grabbing updates from the cloud but not unpacking them yet.

---

14. `git pull`

- ***What it does***: *Fetches updates from the remote repository and* ***applies them*** *to your current branch.*
- ***Why to use it***: *To keep your local branch up-to-date with the remote branch.*
- ***Example***:
    - `git pull origin main`:
        - *Fetches updates from the* `main` *branch on the remote repository named "origin" and merges them into your current branch.*

It's like downloading and automatically merging updates into your local files in one step.

---

15. `git push`

- ***What it does***: *Sends your local changes to the remote repository.*
- ***Why to use it***: *To upload your work so others can see or collaborate with it.*
- ***Examples***:
    - `git push origin main`:
        - *Pushes the changes in your local* `main` *branch to the* `main` *branch on the remote repository named "origin."*
    - `git push --set-upstream origin branch-name`:
        - *Links your local branch (*`branch-name`*) to a branch with the same name on the remote repository. This makes future pushes easier.*

Think of pushing as uploading your work to the cloud for safekeeping or sharing.

Key Differences:

1. *`git fetch`: Only downloads updates without applying them.*
2. *`git pull`: Downloads and applies updates in one step.*
3. *`git push`: Sends your local changes to the remote repository.*

These commands are essential for syncing work between your local code and the remote repository.

12.

F. Undoing Changes:

16.`git restore`

- **What it does**: Reverts changes made to files, either in your working directory or staging area.
- **Why to use it**: To undo changes you've made to files without affecting the entire commit history.
- **Examples**:
  - `git restore file.txt`:
    - Reverts changes made to `file.txt` in your working directory. It will go back to the last committed version of the file.
  - `git restore --staged file.txt`:
    - Unstages `file.txt`, which means it removes the file from the staging area (preparing to be committed), but keeps your changes in the working directory.

Think of **`git restore`** as a way to undo specific changes without affecting previous commits or other files.

17.`git reset`

- **What it does**: Moves your branch back to a specific commit, which can undo commits or changes.
- **Why to use it**: To undo commits or changes and move the current branch to a previous state.
- **Examples**:
  - `git reset HEAD~1`:

        - Moves the branch one commit back (essentially "uncommitting" the most recent commit). Your changes are still in your working directory but not staged for commit anymore.
    - `git reset --hard HEAD~1`:
        - Completely removes the last commit and any changes in your working directory. This means the code will go back to the state it was in one commit ago, **losing any uncommitted changes**.

Think of **`git reset`** as "rewinding" your project to a previous state. The difference is that **`--hard`** completely discards changes, while without it, your changes are kept but unstaged.

---

18. `git revert`

- **What it does**: Creates a new commit that undoes the changes of a specific commit.
- **Why to use it**: To **undo** a commit but still keep the project history intact. It's useful when you want to keep track of what you've undone.
- **Example**:
    - `git revert <commit_hash>`:
        - Reverts the changes introduced by the commit with the given `commit_hash` and creates a new commit to undo those changes. This preserves history while rolling back specific changes.

Think of **`git revert`** as a way to safely undo changes without messing up your project's history. It's like saying, "Here's a new commit that takes back what was done before."

Summary of Differences:

- **`git restore`**: Undo changes in files (can revert changes or unstage files).
- **`git reset`**: Move the branch to a previous commit (can keep or discard changes in the working directory).
- **`git revert`**: Undo a commit with a new commit, preserving history.

These commands are helpful for fixing mistakes, whether it's rolling back changes in files, undoing commits, or creating a safe undo of previous changes.

G. Stashing Changes:

19. `git stash`

- **What it does**: Temporarily saves changes that you've made but don't want to commit yet.
- **Why to use it**: When you need to switch branches or do something else, but don't want to commit your current work. It allows you to "pause" your progress and come back to it later.
- **Examples**:
  - `git stash`:
    - Saves all your uncommitted changes (both staged and unstaged) and resets your working directory to the last commit.
    - This is like putting your work in a temporary storage space, so you can return to it later.
  - `git stash pop`:
    - Applies the changes you stashed earlier and removes them from the stash list. It's like saying, "Bring back the changes I saved earlier and get rid of the saved copy."
  - `git stash list`:
    - Lists all the stashes you've made. You can use this command to see what work you've temporarily saved.

Why Use `git stash`?

- **When you need to switch tasks**: You're working on something but suddenly need to switch to another branch or task. You can stash your current changes and restore them later without committing incomplete work.
- **To clean up your working directory**: Stashing lets you clear your working directory temporarily, so you can pull or merge changes from another branch, then get back to your work later.

Summary:

- `git stash`: Save uncommitted changes temporarily.
- `git stash pop`: Reapply stashed changes and remove them from the stash list.
- `git stash list`: View all saved stashes.

These commands are like a "pause" button for your work. Use them when you need to temporarily set aside your current progress.

H. Viewing Commit History:

20.git log

- **What it does**: Shows the history of commits in your repository.
- **Why to use it**: To view the changes that have been made over time and track the history of your project.
- **Examples**:
  - git log:
    - Displays the full history of commits in the repository, showing detailed information like commit hashes, authors, dates, and commit messages.
    - This is useful when you want to see the full commit history.
  - git log --oneline:
    - Shows a simplified, one-line summary of each commit. Only the commit hash and the commit message are displayed, which makes it easier to get a quick overview of your commit history.
    - This is useful when you want to see the commits in a compact format without too much detail.

21.git show

- **What it does**: Displays detailed information about a specific commit.
- **Why to use it**: To get more in-depth information about a commit, such as the changes made, the commit message, and the commit's author.
- **Example**:
  - git show <commit_hash>:

        - Shows the details of the commit identified by <commit_hash>. This will include the commit message, the changes made (diffs), and other details about that commit.
        - This is useful when you want to inspect a specific commit in more detail.

Summary:

- **git log**: View the full commit history (or a summarized version with --oneline).
- **git show**: View detailed information about a specific commit.

These commands help you track and inspect the history of your project, allowing you to see what changes were made and by whom.

I. Tagging Versions:

22.git tag

- **What it does**: Creates tags to mark specific commits in the repository. Tags are like labels you can apply to certain points in your project's history (e.g., for releases or milestones).
- **Why to use it**: Tags help you keep track of important points, like version releases.
- **Examples**:
    - git tag v1.0:
        - Creates a lightweight tag named v1.0. It's a simple label attached to the current commit, with no extra metadata. It's often used for marking a version or release point in your project.
    - git tag -a v1.0 -m "Version 1.0":
        - Creates an **annotated** tag named v1.0. Annotated tags are more detailed—they include a message (in this

case, "Version 1.0") and store information like the tagger's name, email, and the date. Annotated tags are often used for marking official releases.

23.git push --tags

- **What it does**: Pushes all local tags to the remote repository.
- **Why to use it**: After creating tags locally, you can use this command to upload those tags to the remote repository so others can access them.
- **Example**:
    - git push --tags:
        - Pushes all your tags to the remote repository (e.g., GitHub). This ensures that any important version or milestone labels are shared with others.

Summary:

- **git tag**: Create tags to mark important points in your project history (e.g., versions or releases).
    - git tag v1.0: Lightweight tag for a version.
    - git tag -a v1.0 -m "Version 1.0": Annotated tag with extra details like a message.
- **git push --tags**: Uploads all your local tags to the remote repository.

Tags help you label important points in your project's timeline, such as release versions, making it easier to go back to or refer to those points later.

J. Advanced Commands:

24.git cherry-pick <commit_hash>

- **What it does**: Applies changes from a specific commit to the current branch.
- **Why to use it**: When you want to bring over a specific change (commit) from another branch without merging the entire branch.
- **Example**:
  - git cherry-pick <commit_hash>:
    - Picks a commit identified by <commit_hash> from another branch and applies it to your current branch. This is useful when you want to include a particular change, such as a bug fix, without merging all the changes from the other branch.

25.git archive

- **What it does**: Creates a zip or tar archive of your repository.
- **Why to use it**: To package your repository's contents (or part of it) into a compressed file, which you can share or store.
- **Example**:
  - git archive --output=archive.zip HEAD:
    - Creates a zip file (archive.zip) of the current state of the repository (i.e., the latest commit on the current branch, referenced by HEAD). You can use this to create an archive of your repository that can be shared or saved for later use.

26.git blame <file>

- **What it does**: Shows who made changes to each line of a file.
- **Why to use it**: When you want to track who made a specific change in a file and when. This can be useful for understanding the history of a file or finding out who to ask about a particular line of code.

- **Example**:
    - git blame <file>:
        - Displays the commit hash, author, and date for each line in the specified file, helping you see who changed what and when.

27.git bisect

- **What it does**: Finds a commit that introduced a bug using binary search.
- **Why to use it**: If you know the bug appeared after a certain point in time but don't know exactly where, git bisect helps you quickly find the commit that introduced the problem by checking out each commit in a binary search method.
- **Example**:
    - git bisect start:
        - Begins the binary search process. You'll mark commits as "bad" (where the bug is present) and "good" (where the bug is absent) to help Git narrow down the problematic commit. Git will keep halving the search space until it finds the exact commit that caused the issue.

28.git reflog

- **What it does**: Displays a history of all actions performed on the repository.
- **Why to use it**: When you need to track the history of your actions (like commits, checkouts, rebases), including actions that aren't part of the regular commit history (e.g., resets, branch changes). This is helpful if you need to recover lost commits or see what you've done recently.
- **Example**:
    - git reflog:
        - Shows a log of all recent actions, such as checkouts, commits, rebases, etc. This helps you track everything

you've done in the repository, even if the changes are no longer part of the current branch history.

29.git clean

- **What it does**: Removes untracked files and directories from the working directory.
- **Why to use it**: If you have files that aren't being tracked by Git (e.g., temporary files or build artifacts), git clean helps you remove them to keep your working directory clean.
- **Example**:
  - git clean -fd:
    - Removes all untracked files and directories (-f forces the removal, and -d includes directories). This is useful for cleaning up files that are not being tracked or ignored by Git, such as compiled files or temporary files created during development.

Summary:

- **git cherry-pick <commit_hash>**: Applies a specific commit's changes to your current branch.
- **git archive**: Creates a zip or tar archive of your repository.
- **git blame <file>**: Shows who changed each line in a file.
- **git bisect**: Uses binary search to find the commit that introduced a bug.
- **git reflog**: Displays the history of actions performed on the repository.
- **git clean**: Removes untracked files and directories from your working directory.

These commands are helpful for specific tasks like tracking changes, cleaning up, finding bugs, or creating archives.

K. Git Utilities:

30.git gc

- **What it does**: Optimizes the repository by cleaning unnecessary files.
- **Why to use it**: Over time, Git repositories can accumulate unnecessary files, like old logs or temporary data, which can slow down performance. Running git gc (short for "garbage collection") helps clean up these files and optimize the repository, making it run more efficiently.
- **Example**:
  - git gc:
    - Cleans up unnecessary files and optimizes the repository by compressing file history and removing unreachable objects.

31.git fsck

- **What it does**: Checks the integrity of the repository.
- **Why to use it**: This command verifies the integrity of the repository's data, ensuring that everything is in order. It can help detect issues like corrupted files or missing objects.
- **Example**:
  - git fsck:
    - Scans the repository for any errors or corrupt objects, helping you ensure the integrity of your Git data. If anything is wrong with the repository, this command will report it.

32.git config --global alias.<name> '<command>'

- **What it does**: Creates a shortcut for a Git command.

- **Why to use it**: If you often use a long or complex Git command, you can create a shortcut (alias) for it. This makes using Git faster and easier, as you can just type the alias instead of the full command.
- **Example**:
    - git config --global alias.co checkout:
        - Creates a shortcut for the git checkout command, so instead of typing git checkout, you can now just type git co.

Summary:

- **git gc**: Cleans up unnecessary files in your repository and optimizes its performance.
- **git fsck**: Checks the integrity of your Git repository to ensure everything is correct.
- **git config --global alias.<name> '<command>'**: Creates a custom shortcut for a Git command, making it faster and easier to use.

These commands are useful for maintaining your repository's performance, checking for errors, and making Git commands easier to type.