Git:

Git Basics Workflow - Project Setup and File Operations

1. **Open Git Bash:**
    - Launch Git Bash on your computer.
2. **Navigate to Desktop:**
    - Use `cd` (change directory) to navigate to your Desktop.

    ```
    cd Desktop
    ```

3. **Create a Project Directory:**
    - Create a new directory for your project.

    ```
    mkdir Projecttest
    ```

4. **Create a New File:**
    - Create a new text file called `names.txt`.

    ```
    touch names.txt
    ```

5. **Check Git Status:**
    - See the current status of your repository. It will show untracked files or changes.

    ```
    git status
    ```

6. **Stage the New File:**
    - Add `names.txt` to the staging area, ready for committing.

    ```
    git add names.txt
    ```

7. **Check Git Status Again:**
    - Verify that `names.txt` is now staged.

```
git status
```

8. **Commit the Changes:**
    - Commit the staged file with a descriptive message.

```
git commit -m "names.txt file is added"
```

9. **Check Git Status After Commit:**
    - Verify that the working directory is clean after the commit.

```
git status
```

File Modification and Tracking Changes

1. **Edit the File:**
    - Open the `names.txt` file using `vim` or any text editor.

```
vim names.txt
```

2. **View File Contents:**
    - Check the contents of the file to confirm your changes.

```
cat names.txt
```

3. **Commit the Modified File:**
    - After editing, stage and commit the changes with a message indicating the modification.

```
git add names.txt
git commit -m "names.txt file is modified"
```

4. **Check Git Log:**
    - View the commit history to see all changes made in the repository.

```
git log
```

Stashing Changes

1. **Stash Changes:**
    - If you want to temporarily save changes (without committing them), you can use `git stash`.

```
git stash
```

2. **Check the File After Stash:**
    - After stashing, the changes are removed, so checking the file will show no changes.

```
cat names.txt
```

Connecting to a Remote Repository

1. **Add Remote Repository:**
    - Add a remote repository (e.g., GitHub) to push changes later.

```
git remote add origin https://github.com/kunalkushwaha/CommunityClassroom-Git.git
```

2. **Verify Remote:**
   - Verify that the remote repository is correctly set.

   ```
   git remote -v
   ```

3. **Push Changes (Optional):**
   - To push your local changes to the remote repository, use the following:

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

Additional Git Commands for Workflow

- **Clone a Repository:**
  If you want to create a local copy of a remote repository:

  ```
  git clone <repository-url>
  ```

- **Pull Latest Changes:**
  To fetch and merge changes from the remote repository:

  ```
  git pull origin master
  ```

- **Create and Switch Branches:**
  - Create a new branch and switch to it:

    ```
    git checkout -b new-branch-name
    ```

- **View Branches:**
  - To list all branches in the repository:

    ```
    git branch
    ```

- **Switch Between Branches:**
  - To switch to an existing branch:

    ```
    git checkout branch-name
    ```

- **Merge Branches:**
  - To merge a branch into your current branch:

    ```
    git merge branch-name
    ```

- **View File Changes:**
  - To see changes made in a file (before staging or committing):

    ```
    git diff names.txt
    ```

Best Practices:

- **Use Descriptive Commit Messages:** Always write meaningful commit messages that describe what has been change
- **Commit Often:** Make small, logical commits rather than one large commit with many changes.

- **Push Changes Regularly:** Regularly push your changes to the remote repository to avoid losing work.
- **Stash Unfinished Work:** If you need to switch tasks or branches but aren't ready to commit your changes, use `git stash` to save your work temporarily.

Rebase vs. Merge

- `git merge`: Combines changes from one branch into another. It keeps the history of both branches intact.
- `git rebase`: Re-applies commits from one branch onto another, effectively rewriting the commit history in a linear fashion. This is useful for making your feature branch up to date with the master branch without creating merge commits.
- **Command Example:**

```
git checkout feature-branch
git rebase master
```

2. Interactive Rebase

- This is a powerful feature for editing your commit history. You can squash commits, reorder them, edit commit messages, or even delete commits.
- **Command Example:**
  where
  `n` is the number of commits to go back. For example, `HEAD~3` will interactively rebase the last 3 commits.

```
git rebase -i HEAD~n
```

3. Git Stash

- Stash is useful for saving your uncommitted changes temporarily when you need to switch branches or work on something else.
- **Command Example:**

```
git stash save "message"
git stash apply  # To apply the last stash
git stash list   # List all stashes
git stash pop    # Apply and remove the most recent stash
```

4. Git Cherry-Pick

- You can apply individual commits from one branch to another. This is useful when you want to selectively bring in certain changes without merging the whole branch.
- **Command Example:**

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

5. Git Reflog

- The reflog keeps track of the movements of `HEAD`, allowing you to see the history of branch checkouts, merges, rebases, etc. This is useful for recovering lost commits or fixing mistakes in Git.
- **Command Example:**

```
git reflog
git checkout <commit-hash>   # Checkout a previous commit
```

6. Git Submodules

- A submodule is like a repository inside another repository. This is useful for managing dependencies or breaking down large projects into smaller modules.
- **Command Example:**

```
git submodule add <repository-url> <path>
git submodule update --init  # Initialize and update the submodule
```

7. Git Bisect

- This is a debugging tool that helps find which commit introduced a bug by performing a binary search.
- **Command Example:**
  Git will then start checking out commits in the middle to help you locate the buggy commit.

```
git bisect start
git bisect bad               # Mark the current commit as bad
git bisect good <commit-hash>  # Mark an older commit as good
```

8. Git Blame

- Git blame shows line-by-line commit history for a file, including who made the changes and when. This is very useful for tracing bugs and understanding the evolution of a file.
- **Command Example:**

```
git blame <file-path>
```

9. Git Log Customization

- You can customize how logs are displayed to get more insightful information.
- **Command Example:**

```
git log --oneline --graph --decorate --all   # Simple graph view
git log --author="Author Name"              # Show commits by a specific author
git log --since="2 weeks ago"               # Commits from the last 2 weeks
```

10. Git Show

- Displays information about a specific commit, like changes or metadata (like who committed, date, etc.).
- **Command Example:**

```
git show <commit-hash>
```

11. Git Reset

- Resets your branch to a previous commit and can be used to unstage or undo changes.
- **Command Example:**
    - **Soft Reset** (keep changes in working directory and staging area):

      ```
      git reset --soft <commit-hash>
      ```

    - **Mixed Reset** (keep changes in working directory but unstage them):

      ```
      git reset --mixed <commit-hash>
      ```

    - **Hard Reset** (discard all changes):

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

12. Git Filter-Branch (or `git filter-repo`)

- Allows you to rewrite history in a repository, like removing a file from all commits. It's powerful but dangerous, so use with caution.
- **Command Example:**

  ```
  git filter-branch --tree-filter 'rm -f <file-path>' HEAD
  ```

- **Note:** `git filter-repo` is preferred for larger repositories due to better performance.

13. Git Diff and Git Diff Tool

- Show the differences between various states of your repository, such as between commits, branches, or the working directory and the index.
- **Command Example:**

```
git diff HEAD          # Difference between the working directory and the last commit
git diff <commit1> <commit2>  # Difference between two commits
git difftool           # Use an external tool for diff (e.g., `meld`, `vimdiff`)
```

14. Git Merge Strategies

- Sometimes Git can't automatically merge changes, and you may need to manually specify a strategy.
- **Command Example:**

```
git merge -s ours <branch>   # Keep your branch's changes, discard the other branch's changes
git merge -s theirs <branch> # Keep the other branch's changes, discard your branch's changes
```

15. Git Clean

- This command is useful for removing untracked files or directories from your working directory.
- **Command Example:**

```
git clean -f   # Removes untracked files
git clean -fd  # Removes untracked files and directories
git clean -fx  # Also removes ignored files
```

16. Git Remote Prune

- Used to remove references to remote branches that no longer exist.
- **Command Example:**

```
git remote prune origin
```

17. Git Config

- Set global, local, or system configuration values, such as user name, email, editor, etc.
- **Command Example:**

  ```
  git config --global user.name "Your Name"
  git config --global user.email "your.email@example.com"
  ```

18. Using `git reset` (for local changes)

If you want to remove a commit and reset your branch to a previous state (this is a **local-only change**), you can use `git reset`. There are two main types of resets you can do:

a) Soft Reset

This will keep the changes in your working directory, but remove the commit itself from the history. Useful if you want to redo the commit.

```
git reset --soft <commit-hash>
```

b) Hard Reset

This will **remove** the commit and **discard all changes** in the working directory.

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

For example, to remove the last commit:

```
git reset --hard HEAD~1
```

This will remove the latest commit and all changes associated with it.

19. Using `git rebase` (for removing or modifying commits)

If you want to **edit** or **remove specific commits** in the history, you can use `git rebase -i` (interactive rebase).

To remove a commit from history, follow these steps:

1. **Start an interactive rebase** to the commit you want to modify:

   ```
   git rebase -i <commit-hash>^
   ```

   The `^` means you want to rebase from the commit before the one you want to modify.

2. **In the editor**, you will see a list of commits. To remove a commit, simply **delete the line** corresponding to that commit. You can also replace `pick` with `edit` if you want to modify the commit instead of deleting it.

3. **Save and close the editor**. Git will rebase the history, removing the commit.

4. **Force push** the changes to the remote repository (if applicable):

   ```
   git push --force
   ```

20. Using `git reflog` (for recovering from mistakes)

If you've made a mistake and need to recover a commit that was previously removed, you can use `git reflog` to find the commit reference and reset your branch back to it.

```
git reflog
```

Find the commit hash you want to recover and use `git reset` to return to that state:

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

21. Removing a Commit from Remote (Force Push)

After using `git reset` or `git rebase` to modify your history, if you need to reflect this on the remote repository, you'll have to force-push your changes:

```
git push --force
```

However, be cautious with force-pushing as it can overwrite history in a way that may affect other collaborators.

22. Removing the Commit from Remote (Git Revert)

If you want to keep history intact but undo the changes from a specific commit, you can use `git revert`:

```
git revert <commit-hash>
```

This creates a new commit that undoes the changes introduced by the specified commit. It is **safe for shared repositories** because it doesn't alter the commit history, unlike `git reset` or `git rebase`.