**Git Notes**

For questions -

## https://www.simplilearn.com/tutorials/git-tutorial/git-interview-questions

Difference between git merge and git rebase? Which method do you prefer?

The main difference is in how the branch history is presented.   
**`git merge`** preserves the history of a both branches by creating a new merge commit.

**`git rebase`** moves or rewrites your changes on the top of another branch, resulting in a linear commit history.  
  
**Best Practices : -**

* **Always pull latest changes** before merging
* **Keep branches short-lived** to minimize conflicts
* **Test before merging** to main branches
* **Use merge tools** (like VS Code's built-in tool) for complex conflicts
* **Consider rebasing** for cleaner history (but not on shared branches)

## Difference between `git-fetch` vs. `git pull`?

**`git-fetch`** downloads updates from a remote repository to your local repository without integrating them into your working files allows you to see what others have committed before merge..

**`git pull`** is a combination of git-fetch followed by git merge which automatically merge changes into your active working directory. keep your branch in-sync with remote branch.

## Difference between git log and git reflog ?

**git log** - It shows the commit history of a repository i.e the commits that are a part of the current branches timeline and are reachable from the branch head ( or any other branch). It's the primary tool for tracking the formal project history.  
  
**git reflog** - Tracks all changes to references ( like head ) in the local repository including changes that aren't reflected in the commit history ( eg checkout, reset or rebase ) It records any movement of head or branch pointers, weather the change is a part of the formal commit history or not.

## How do you revert a commit that has already been pushed and made public?

To undo the modifications introduced by a previous commit while ensuring safety for public commits, employ `git revert <commit\_hash>`, which generates a fresh commit that reverses the earlier changes. On the other hand, `git reset` (**Don’t use git reset on public branches** ) allows reverting to an earlier state; This changes history and can break things for others who’ve already pulled the original commits. Only use reset on private/local branches.

## How to Revert a Commit

### 1. Identify the commit to revert

First, find the commit hash you want to revert:

* git log –oneline

Example output:

a1b2c3d (HEAD -> main) Add new feature

e4f5g6h Update documentation

i7j8k9l Fix login bug

### 2. Create a revert commit

Use git revert with the commit hash:

* git revert a1b2c3d

This will create a new commit that undoes the changes from the specified commit.

### 3. Push the revert commit

> git push origin main

Commit x9y8z7w  contains the reversal of the changes in a1b2c3d .  
  
Now your history will show:

1. x9y8z7w - Revert "Accidentally deleted an important file" (new revert commit)
2. a1b2c3d - Accidentally deleted an important file
3. e4f5g6h - Previous working commit

## What Is Git ?

Git is a widely adopted distributed version control system (VCS) designed to track modifications in source code throughout the software development process and to coordinate work amoung multiple developers.

## Why Is Git Used?

* **Version Control:** Tracks changes to files over time, allowing you to recall specific versions later.
* **Backup and Restore:** Changes in Git are committed to the local repository, creating a complete project history. This history provides a backup of every change, and it's possible to revert to any previous project state.
* **Collaboration:** Multiple developers can work on the same project simultaneously.
* **Distributed :** Everyone has a full copy of the project history on their machine**.**
* **Branching and Experimentation:** Enables parallel development workflows with easy branch creation and merging..
* **Track Changes:** Git allows users to see precisely what changes have been made, by whom, and when. This is valuable for understanding a project's evolution and auditing changes.
* **Continuous Integration and Delivery (CI/CD):** Git integrates seamlessly with numerous Continuous Integration/Continuous Deployment (CI/CD) solutions, streamlining software testing and deployment. This enhances the efficiency of the development workflow and minimizes the likelihood of mistakes.

## What’s the difference between Git and GitHub?

Key Differences :

| **Feature** | **Git** | **GitHub** |
| --- | --- | --- |
| **Type** | Version control software | Hosting service for Git repos |
| **Location** | Local on your machine | Cloud-based |
| **Interface** | Command-line | Web interface + GUI tools |
| **Collaboration** | Basic (via remotes) | Advanced features |
| **Requires Internet** | No (works offline) | Yes (for most features) |
| **Pricing** | Free and open-source | Free tier + paid plans |

## What is branching in Git?

A **branch** in Git is like a **parallel version** of your project. It lets you make changes, try new features, or fix bugs **without affecting** the main project (usually called the main or master branch).

git branch branch-name # Creates branch

git checkout branch-name # Switches to branch

## Types of key branching concepts in git?

Key Branching Concepts

## Main Branch (traditionally called "master")

* **The primary/default branch.**
* Contains production-ready code.
* Often protected from direct commits.

## Feature Branches

* **Temporary branches for developing new features.**
* Naming convention: feature/feature-name.
* Example: feature/user-authentication.

## Release Branches

* For preparing **new production releases.**
* Naming convention: release/version-number.
* Example: release/v2.1.0

## Hotfix Branches

* **For urgent fixes to production code.**
* Naming convention: hotfix/issue-description.
* Example: hotfix/login-bug.

## Branching Commands

Create a new branch

git branch branch-name # Creates branch

git checkout branch-name # Switches to branch

git checkout -b branch-name # Creates and switches

List branches

git branch # Local branches

git branch -a # All branches (including remote)

Switch between branches

git checkout branch-name

## What is a merge in Git?

Merging is a Git operation **that integrates changes from one branch into another.** It combines two branches by creating a new merge commit preserving the history of both branches.

Use merge when:

* Working on shared branches
* Preserving exact commit history is important
* You want to clearly see where branches diverged

## How to Perform a Merge :

1. Checkout the target branch (where you want to merge into):

git checkout main

1. Pull the latest changes:

git pull origin main

1. Merge the source branch:

git merge feature-branch

1. Resolve any conflicts (if they occur)
2. Push the merged changes:

git push origin main

## What is `git rebase`?

`git rebase` moves or rewrites your changes on the top of another branch, resulting in a linear commit history.

Golden Rules of Rebasing

* **Never rebase public branches** (branches others are working on)
* **Only rebase local branches** you haven't shared yet
* **Use pull with rebase** to keep local branch updated: git pull --rebase origin main

## Example Workflow : -

1. Start working on a feature:

git checkout -b feature/login

1. Make several commits (A, B, C)
2. Meanwhile, main branch gets updates (D, E)
3. Rebase your feature onto main:

git fetch origin

git rebase origin/main

1. Resolve any conflicts if they occur
2. Force push (since history changed):

git push origin feature/login --force-with-lease

## What is a conflict in Git?

A conflict in Git occurs when git cannot automatically resolve changes from different branches.A conflict in Git occurs when two branches have made edits to the same line in a file or when one branch deletes a file while the other branch modifies it. Git cannot automatically resolve these changes; the developer must manually resolve the conflicts.

1. Git will mark conflicted files
2. Open files and look for conflict markers:

<<<<<<< HEAD

Current branch content

=======

Incoming branch content

>>>>>>> feature-branch

1. Edit to keep the desired changes
2. Mark as resolved:

git add conflicted-file.txt

git commit

## Best Practices

1. **Always pull latest changes** before merging.
2. **Keep branches short-lived** to minimize conflicts..
3. **Test before merging** to main branches.
4. **Use merge tools** (like VS Code's built-in tool) for complex conflicts.
5. **Consider rebasing** for cleaner history (but not on shared branches)

## How do you resolve a merge conflict?

To resolve a merge conflict, edit the files to fix the conflicting changes. Then, use `git add` to stage the resolved files and `git commit` to commit the resolved merge.

## What’s the difference b/w git merge & git rebase? Which method do you prefer?

The main difference is in how the branch history is presented.   
**`git merge`** preserves the history of a both branches by creating a new merge commit.

**`git rebase`** moves or rewrites your changes on the top of another branch, resulting in a linear commit history.

## What is `git-fetch` vs. `git pull`?

**`git-fetch`** downloads updates from a remote repository to your local repository without integrating them into your working files allows you to see what others have committed before merge..

**`git pull`** is a combination of git-fetch followed by git merge which automatically merge changes into your active working directory. keep your branch in-sync with remote branch.

## How do you revert a commit that has already been pushed and made public?

To undo the modifications introduced by a previous commit while ensuring safety for public commits, employ `git revert <commit\_hash>`, which generates a fresh commit that reverses the earlier changes. On the other hand, `git reset` allows reverting to an earlier state; however, exercise caution when applying it to public commits, as it alters the commit history.

## What is `git merge --squash`?

The `git merge --squash` command is a merge starategy that combines all commits from a feature branch into a **single new commit** when merging into the target branch, resulting in a tidier project history.  
   
**When to Use Squash Merge**

* When a feature branch has **many small, incremental commits** you want to consolidate
* When you want to **keep main branch history clean**
* For **bugfix branches** where individual commit messages aren't important
* When you want to **rewrite commit messages** for clarity

## Example Workflow

Starting Point

main: A---B---C

\

feature: D---E---F

### 1. Checkout the target branch (main)

git checkout main

git pull origin main # Ensure you're up-to-date

### 2. Perform the squash merge

git merge --squash feature

### 3. Commit the squashed changes

git commit -m "Implement user profile feature"

Result

main: A---B---C---G

\ /

feature: D---E---F

Where G is a **new single commit** containing all changes from D, E, and F.

## What is `git stash`?

The `git stash` command temporarily stores your working directory's modifications, allowing you to switch between branches without discarding your current progress.

## What command do you use to return to a previous commit?

To return to a previous commit on a private, local repository, you’ll first want to run git log to pull up the branch’s history. Then, after identifying the version’s hash you want to go to, use the git reset command to change the repository to the commit.

For a public, remote repository, the git revert command is a safer option. It’ll create a new commit with the previous edits rather than delete commits from the history.

## How do you change the last commit?

Use `git commit --amend` to modify the most recent commit. This can change the commit's message or include new changes.

## How do you delete a branch locally and remotely?

Use **`git branch -d <branch\_name>`** to delete a local branch. If the branch is not fully merged, you may need to use `-D` instead. To delete a remote branch, use **`git push <remote\_name> --delete <branch\_name>`.**

## What is `git checkout`?

**`git checkout`** allows navigating between different branches or reverting working tree files to a previous state. However, in the latest versions of Git, it is advised to use `git switch` for changing branches and `git restore` to revert files, each designated for these specific functions.

## How do you remove a file from Git but not delete it from your file system?

Use **`git rm --cached <file\_name>`** to remove a file from Git without deleting it from your filesystem.

## How do you rename a Git branch?

To rename the current branch, use **`git branch -m <new\_name>`**. To rename a different branch, use `**git branch -m <old\_name> <new\_name>`.**

## How do you rename a remote branch?

Rename the local branch using **`git branch -m <new\_name>`**, push it to the remote, and then delete the old remote branch.

## What does `git reset` do?

`git reset` is used to undo changes. It has three main modes: `--soft`, `--mixed`, and `--hard`.

## What is the HEAD in Git?

HEAD is a reference to the last commit in the currently checked-out branch.

## What is the difference between `git stash pop` and `git stash apply`?

**`git stash pop`** - It applies stashed changes and removes them from the stash list.

`git stash apply` applies stashed changes but keeps them in the stash list for later use..

## What is a fast-forward merge in Git?

A fast-forward merge happens when the target branch's tip is behind the merged branch's tip, allowing the target branch to "catch up" by just moving forward to the merged branch's tip.

## Explain the Git branching strategy you use.

A common strategy is the Git Flow, which involves having a master branch, develop branch, feature branches, release branches, and hotfix branches, each serving a different purpose in the development cycle.

## How do you revert a Git repository to a previous commit?

Use `git reset --hard <commit-hash>` to revert to a specific commit, discarding all changes since that commit.

## What is a detached HEAD in Git?

A detached HEAD occurs when you check out a commit, branch, or tag that is not the latest commit of a branch.

## How do you fix a detached HEAD?

Create a new branch from the detached HEAD state with `git branch <new-branch>` and check it out with `git checkout <new-branch>` to move back to a non-detached state.

## How do you combine multiple commits into one without merging?

## Interactive Rebase to Squash Commits

1. Check your commit history

First, see how many commits you want to squash:

git log --oneline

# Example output:

# abc1234 (HEAD -> feature-branch) Add final tweaks

# def5678 Update documentation

# ghi9012 Implement core functionality

# jkl3456 Initial setup

### 2. Start interactive rebase

Decide how many commits to include (let's squash the last 3 commits):

git rebase -i HEAD~3

Alternatively, to rebase from a specific commit:

git rebase -i jkl3456 # The commit BEFORE the ones you want to squash

### 3. The interactive editor will open

You'll see something like this in your text editor:

pick ghi9012 Implement core functionality

pick def5678 Update documentation

pick abc1234 Add final tweaks

### 4. Mark commits to squash

Change the text to (using 'squash' or just 's'):

pick ghi9012 Implement core functionality

squash def5678 Update documentation

squash abc1234 Add final tweaks

### 5. Save and close the editor

(Git will now combine these commits)

### 6. Edit the final commit message

A new editor will open showing all commit messages. Edit this to create one clean message:

Implement complete feature with documentation

# This combines:

# Implement core functionality

# Update documentation

# Add final tweaks

7. Verify the result

Check your new commit history:

git log --oneline

# Now shows:

# 789abcd (HEAD -> feature-branch) Implement complete feature with documentation

# jkl3456 Initial setup

### 8. Force push if already pushed to remote

git push origin feature-branch --force-with-lease

## Explain the difference between `git pull` and `git fetch` followed by `git merge`.

`git pull` is a combination of git-fetch followed by git merge which automatically merge changes into your active working directory. keep your branch in-sync with remote branch.

`git-fetch` downloads updates from a remote repository to your local repository without integrating them into your working files allows you to see what others have committed before merge..

`git merge` preserves the history of a both branches by creating a new merge commit. Tt allows you to review changes before merging.

`git rebase` moves or rewrites your changes on the top of another branch, resulting in a linear commit history.  
  
What is the significance of `git push --force`?

`git push --force` is used to overwrite the remote history with your local history. It should be used with caution as it can overwrite changes in the remote repository.

## What is the difference between `HEAD`, `working tree` and `index` in Git?

`HEAD` refers to the last commit on the current branch, `working tree` is the set of files in your directory, and `index` (or staging area) is a staging area for commits.

## How do you revert changes made to the working directory?

Use `git checkout -- <file-name>` to discard changes in the working directory.

## What is the purpose of `git log --graph`?

`git log --graph` displays the commit history in a graphical representation.

## How do you copy a commit from one branch to another?

Use `git cherry-pick <commit-hash>` to apply the changes from a commit on another branch to the current branch.

## What methods do you use to clean up your feature branches?

For this question, you can mention these three commands.

git merge --squash is a command that can merge multiple commits of a branch.

git commit --fixup marks the commit as a fix of the previous commit.

git rebase -i --autosquash is a rebase type of squash for merging multiple commits.

## Have you ever encountered a merge conflict? How did you go about resolving it?

Merge conflicts happen when there are competing changes within the commits, and Git is unable to automatically determine which changes to use. Merge conflicts can be resolved by editing the file that has the conflict. And once the edits get made, add and commit the file.

## Git Branching Workflow Example

This demonstrates a complete feature development workflow using branches, from creation through merging.

**1. Start from the main branch**

# Ensure you're on main and have latest changes

git checkout main

git pull origin main

### 2. Create a new feature branch

# Create and switch to new branch

git checkout -b feature/user-profile

### 3. Develop your feature (make multiple commits)

# Make code changes, then:

git add .

git commit -m "Add profile page layout"

# More changes...

git add .

git commit -m "Implement profile picture upload"

# Even more changes...

git add .

git commit -m "Add user bio section"

### 4. Sync with main branch updates

# Fetch latest changes from remote

git fetch origin

# Rebase your feature branch on top of main

git rebase origin/main

# Resolve any conflicts if they occur

# (Fix conflicts, then:)

git add .

git rebase –continue

### 5. Push your branch to remote

# First push requires setting upstream

git push -u origin feature/user-profile

# Subsequent pushes can just use:

git push

### 6. Create a Pull Request (on GitHub/GitLab)

* **Go to your repository on GitHub**
* **Click "New Pull Request"**
* **Select main as base and feature/user-profile as compare**
* **Add description and request reviews**

### 7. Address review feedback

# Make additional changes if requested

git add .

git commit -m "Address review comments"

# Push updates to same branch

git push

### 8. Merge the approved PR

* **On GitHub, click "Merge pull request"**
* **Choose merge method (typically "Squash and merge" or "Rebase and merge")**
* **Confirm merge**

### 9. Clean up local branches

# Switch back to main

git checkout main

# Pull the merged changes

git pull origin main

# Delete the local feature branch

git branch -d feature/user-profile

# Optionally delete remote branch

git push origin --delete feature/user-profile

## Visual Workflow Diagram

main: A---B---C---------------M

\ \ /

feature: D---E---F---G (PR Merged)

**Where:**

* **A,B,C: Commits on main (possibly by other developers)**
* **D,E,F,G: Your feature commits**
* **M: Merge commit (if using regular merge)**

## Key Best Practices

1. **Keep branches focused - One feature/bugfix per branch**
2. **Make small, frequent commits - Easier to review and revert**
3. **Rebase regularly - Minimizes merge conflicts**
4. **Use descriptive names - feature/xxx, fix/yyy, chore/zzz**
5. **Delete merged branches - Prevents branch pollution**

**This workflow enables clean collaboration while maintaining a stable main branch. The rebase step (step 4) is optional but recommended to keep history linear.**