Advanced Git Commands and Workflows

This document covers advanced Git concepts and workflows, such as branching strategies, rebasing, stashing, cherry-picking, and conflict resolution. These are essential for working efficiently in collaborative environments.

1. Working with Branches

Command: git branch
Command: git checkout
Command: git switch
Example:
```
git checkout -b feature/login
```
Explanation:
Use branches to isolate your work. You can create, switch, and delete branches as needed.

2. Rebasing

Command: git rebase
Example:
```
git rebase main
```
Explanation:
Moves your commits on top of another branch’s latest commits, creating a linear history.

■■ Use rebasing for clean commit history — but avoid rebasing shared branches.

3. Squashing Commits

Command: git rebase -i HEAD~n
Example:
```
git rebase -i HEAD~3
```
Explanation:
Combines multiple commits into one, keeping history cleaner. Choose 'squash' or 'fixup' in the rebase editor.

4. Cherry-Picking Commits

Command: git cherry-pick
Example:
```
git cherry-pick d28d178
```
Explanation:

Apply a specific commit from one branch onto another without merging the whole branch.

5. Stashing Changes

Command: git stash
Command: git stash pop
Example:

```
git stash save "WIP on login feature"
git stash pop
```

Explanation:
Temporarily saves your uncommitted changes. Useful when you need to switch branches without committing.

6. Viewing Differences

Command: git diff
Example:

```
git diff HEAD~1 HEAD
```

Explanation:
Shows differences between commits, branches, or the working directory.

7. Amending Commits

Command: git commit --amend
Explanation:
Modifies the last commit, letting you add forgotten changes or fix the commit message.

8. Reset Types

Command: git reset --soft HEAD~1
Command: git reset --mixed HEAD~1
Command: git reset --hard HEAD~1
Explanation:

- soft → keeps changes staged
- mixed → keeps changes but unstages them
- hard → discards changes completely

9. Restoring Files

Command: git restore
Example:

git restore README.md
Explanation:
Discards local modifications and restores the file from the last commit.

10. Resolving Merge Conflicts

Scenario: During merge, you’ll see conflict markers (<<<<<<<, =======, >>>>>>>)
Steps:
1. Edit the file and choose the correct content.
2. git add
3. git commit
Explanation:
This resolves the merge conflict and completes the merge process.

11. Using Tags

Command: git tag
Command: git push origin
Example:
git tag v1.0
git push origin v1.0
Explanation:
Tags mark specific commits, often used for release versions.

12. Undoing a Merge Before Committing

Command: git merge --abort
Explanation:
Cancels a merge process that resulted in conflicts, restoring the branch to its previous state.

13. Cleaning Untracked Files

Command: git clean -fd
Explanation:
Deletes all untracked files and directories. Be cautious — this is irreversible.

14. Inspecting History and Branches

Command: git log --graph --oneline --decorate --all
Explanation:
Displays a visual representation of commits and branches in your repository.

15. Working with Remotes

Command: git remote -v
Command: git remote show origin
Command: git fetch --all
Explanation:
Displays and fetches updates from all remotes and their branches.

16. Reflog for Recovery

Command: git reflog
Explanation:
Shows the history of all branch changes and HEAD movements. Useful for recovering lost commits.

17. Interactive Add (Partial Commits)

Command: git add -p
Explanation:
Lets you stage changes interactively — choose specific hunks to commit.

18. Git Hooks

Location: .git/hooks/
Explanation:
Automate tasks like running tests or formatting code before commits. Example: pre-commit, post-merge hooks.

19. Branching Strategies

Examples:
• Feature Branch Workflow
• Gitflow Workflow
• Trunk-Based Development
Explanation:
Defines how teams organize branches for stable and efficient collaboration.

20. Submodules

Command: git submodule add
Command: git submodule update --init
Explanation:
Allows you to include other repositories within your project — ideal for shared libraries.

Mastering these advanced Git commands and workflows allows you to collaborate smoothly, maintain cleaner project history, and recover easily from mistakes.