Git Commands — Corrected & Expanded Notes

This document organizes common Git commands from basic to advanced, with corrected explanations and practical examples. It covers initialization, branching, remotes, pushing/pulling, rebasing, resets, stashing, and collaboration workflows. Examples use 'origin' as the default remote alias and assume modern Git (>=2.23) where 'git switch' and 'git restore' are available.

1. Repository Initialization & Cloning

`git init`
Create a new Git repository in the current directory. This creates the .git directory storing all metadata. No branch pointer exists until you make the first commit; historically the initial branch was named 'master', but new Git installs often use 'main'. You can create a branch before a commit with 'git checkout -b <name>' or 'git switch -c <name>'.

`git clone <URL> [local-dir]`
Clone a remote repository into 'local-dir' (or the repo name if omitted). The clone configures a remote named 'origin' pointing to the URL. The local HEAD will point at the default branch of the remote (commonly 'main').

2. Configuration & Identity

`git config --global user.name "Name"`
Set global commit author name used in commits. Use local config (without --global) to set repo-specific identity.

`git config --global user.email "email"`
Set global commit author email.

`git config --global --list`
List global configuration values (username, email, aliases, etc.).

`git config --global alias.lg "log --oneline --graph --decorate --all"`
Create an alias 'git lg' for a compact graph log. Remove with 'git config --global --unset alias.lg'.

3. Working areas: working tree, index (staging), commit history

`Working flow`
Edit files in working directory → 'git add' to stage → 'git commit' to record in local history → 'git push' to send commits to remote. Staging area (index) is the snapshot that will be committed.

`git status`
Show current branch, staged/untracked/modified files and suggested next steps. Use 'git status --short' for compact symbols (e.g., 'M' modified, 'A' added, '??' untracked).

4. Staging & Restoring files (modern commands)

`git add <file>`
Stage changes to the index for the next commit.

`git add -u`
Stage updates and deletions of tracked files (does not add new untracked files).

`git add --ignore-removal <paths>`
Stage added or modified files but do not stage deletions (useful when you don't want to stage file removals).

`git restore <file>`
Restore a tracked file in working directory to its last committed state (discard local modifications). For untracked files this will error.

`git restore --staged <file>`
Unstage a file: move it from index back to working directory changes (equivalent to 'git reset HEAD <file>').

`git mv <old> <new>`
Rename or move a tracked file and stage the rename. Using OS 'mv' may show as delete+add in history; 'git mv' records rename intent.

5. Committing

`git commit -m "message"`
Create a commit from staged changes with the message. Commits record author, timestamp and parent reference(s).

`git commit --amend --no-edit`
Modify the last commit to include current staged changes without changing the message. This rewrites the previous commit (creates a new SHA) — do not amend commits already pushed/shared without coordination.

`git commit --amend -m "new msg"`
Amend the last commit and replace its message. Remember to force-push if you already pushed the original commit.

6. Branching & switching (recommended modern usage)

`git branch`
List local branches. Use 'git branch -a' to include remote-tracking refs.

`git switch <branch>`
Switch to an existing branch (preferred over 'git checkout' for clarity).

`git switch -c <new-branch>`
Create and switch to a new branch (equivalent to 'git checkout -b').

`git switch -`
Switch back to previous branch (convenient toggle).

`git checkout <commit>`
Move to a specific commit (detached HEAD). You can inspect or create commits but they won't belong to a named branch unless you create one and attach them.

`git branch -d <branch> / git branch -D <branch>`
Delete a local branch: '-d' refuses if branch contains commits not merged to the current branch (safe), '-D' forces deletion.

7. Renaming branches & default branch conventions

`git branch -m <old> <new> / git branch -M <old> <new>`
'-m' renames a branch; '-M' forces rename even if the target exists. You can also rename current branch with 'git branch -m <new>'. Use '-M' to align legacy 'master' → 'main'.

8. Upstream (tracking) configuration

`git push -u origin <branch>`

Push local branch to origin and set its upstream (tracking) reference. After this, 'git push' and 'git pull' will use that upstream by default for the branch.

`git branch --set-upstream-to=origin/<remote-branch> [local-branch]`
Set or change the upstream (tracking) branch for local branch without pushing. Equivalent shorthand: 'git branch -u origin/<remote-branch>'.

`git branch --unset-upstream`
Remove upstream tracking from the current branch (useful to detach default push/pull behavior).

9. Remotes: add, remove, inspect, change URLs

`git remote -v`
List remote names and their fetch/push URLs (e.g., 'origin').

`git remote add <name> <URL>`
Add a new remote alias (e.g., 'upstream' for the original project when you fork).

`git remote remove <name>`
Remove a remote alias from your config.

`git remote rename <old> <new>`
Rename a remote alias.

`git remote set-url <name> <new_url>`
Change the stored URL for a remote (switch HTTPS↔SSH or change host).

`git remote set-url --add <name> <url>`
Add additional push URL(s) so pushes go to multiple endpoints. Use with care.

10. Fetch, pull, and inspect before merging

`git fetch [remote]`
Download commits and refs from remote to local remote-tracking branches (does not modify working tree).

`git fetch --all --prune`
Fetch all remotes and prune removed remote branches from local remote-tracking refs.

`Recommended: inspect before integrating`
Good practice: 'git fetch origin' → 'git log HEAD..origin/<branch>' or 'git diff HEAD origin/<branch>' to review incoming changes before 'git merge' or 'git pull'.

11. Pushing: mapping refs and deletion via colon syntax

`git push origin <local-branch>:<remote-branch>`
Push local branch to a specific remote branch name (creates remote branch if absent).

`git push -u origin <branch>`
Push and set upstream so future 'git push' targets origin/<branch> by default.

`git push origin HEAD`
Push the current branch to the remote branch with the same name (HEAD is the current ref).

`git push origin HEAD:master`
Push current branch HEAD to update the remote 'master' branch (maps local HEAD to remote master).

`git push origin :<branch>`
Delete the remote branch '<branch>' by pushing an empty (missing) local ref to that remote ref. This is the standard way to remove a remote branch.

`Note on 'git push origin :'`

A command with only a colon and no branch names (e.g., 'git push origin :') is typically a no-op or rejected by Git because it supplies an empty refspec; the meaningful use is 'git push origin :branch' to delete 'branch'. Do not rely on an ambiguous bare colon.

12. Merging: strategies & options

`git merge <branch>`
Merge another branch into current. If current branch has no divergent commits, Git will fast-forward (no merge commit). If diverged, Git creates a merge commit that records both parents.

`git merge --no-ff <branch>`
Always create a merge commit even if fast-forwarding is possible (preserves topic branch in history).

`git merge --ff-only <branch>`
Only allow fast-forward merges; abort if a merge commit would be required. Useful for linear histories.

`git merge --squash <branch>`
Apply changes from the other branch into the index as a single set of changes (no commit). You must create a commit manually afterwards. Good for producing a single commit for a feature.

13. Rebasing & interactive rebase (history rewriting)

`git rebase <branch>`
Reapply local commits of current branch on top of <branch>. This rewrites commit history (new SHAs). Avoid rebasing branches others share/pull from.

`git rebase -i <base-commit>`
Interactive rebase: edit the list of commits, reorder, squash ('s' or 'squash'), reword or drop commits. Common pattern to clean up a feature branch before pushing.

`git rebase --continue / --abort`
Continue after resolving conflicts or abort rebase to return to the pre-rebase state.

`git rebase --onto <newbase> <upstream> <branch>`
Advanced: transplant a sequence of commits that are after <upstream> on <branch> onto <newbase>. Useful for selective history surgery.

14. Squashing & PR hygiene

`git rebase -i <previous_stable_commit>`
Squash multiple commits into one for a cleaner PR. After rewriting history, you must force-push (prefer --force-with-lease) to update the remote branch associated with your PR. Use distinct branches per PR to avoid mixing changes.

15. Reset, revert & cleaning working tree

`git reset --soft <commit>`
Move HEAD to <commit> but leave index and working tree as-is (staged). Useful for regrouping commits into a single commit.

`git reset [--mixed] <commit> (default)`
Move HEAD to <commit> and reset index to that commit, leaving working tree files as unstaged changes. This is default 'git reset <commit>'.

`git reset --hard <commit>`
Reset HEAD, index and working tree to <commit>; discards any local changes — irreversible without backups.

`git revert <commit>`
Create a new commit that inverses the effect of <commit>. Safe for public branches because it preserves history.

`git clean -f [-d]`
Remove untracked files (and with -d, untracked directories). Use with caution; you may preview with '-n' (dry run).

16. Stash (temporary shelving)

`git stash`
Save local changes (staged+unstaged) on a stack and clean working directory. Stash entries are stored as stash@{0}, stash@{1}, ...

`git stash -u`
Include untracked files in the stash. Useful for fully cleaning the working directory.

`git stash list`
List stashes with messages.

`git stash apply [stash@{N}] / git stash pop [stash@{N}]`
'apply' re-applies stash and keeps it in the list; 'pop' re-applies and removes it from the stash list.

`git stash branch <branch> [stash@{N}]`
Create a new branch from the commit where stash was made and apply the stash there; convenient to resume stashed work into a branch.

`git stash drop [stash@{N}] / git stash clear`
Drop a specific stash or clear all stashes.

17. Cherry-pick & selective commits

`git cherry-pick <commit>`
Apply the changes from <commit> onto current branch as a new commit. Useful to port bugfixes between branches.

`git cherry-pick <A>^..<B>`
Cherry-pick a sequence/range of commits (commits after A up to and including B). Conflicts may arise and must be resolved.

`git cherry-pick --no-commit <commit>`
Apply the commit's changes to index without making a new commit (lets you modify before committing).

18. Diffs & inspection

`git diff`
Show unstaged changes (working tree vs index).

`git diff --staged`
Show staged changes (index vs HEAD).

`git diff <commit1> <commit2>`
Compare two commits.

`git diff <localBranch> <remote>/<branch>`
Compare local branch to remote-tracking branch (run 'git fetch' first).

`git diff -- <path>`
Use '--' to separate options from file paths when ambiguous.

19. Logs & ranges

`git log --oneline --graph --decorate --all`
Compact visual history. Useful to inspect branching/merges.

`git log <SHA_LATER>..<SHA_OLDER>`
Show commits in the range between two SHAs.

`git log HEAD..origin/<branch>`
Show commits present on remote branch but not in local HEAD (after 'git fetch').

20. Forking & multi-remote workflows

`git remote add upstream <OG_URL>`
Add original project remote as 'upstream' when working on fork.

`Sync fork (simple)`
git pull upstream main -> git push origin main (merge upstream changes into your fork and push).

`Sync fork (reset method)`
git fetch --all --prune -> git reset --hard upstream/main -> git push origin main (force-update your fork to match upstream). Use cautiously.

21. Safety & best practices

`Never rebase published/shared branches`
Rebasing rewrites commit SHAs and can break others' history; prefer merging on shared branches.

`Prefer --force-with-lease over --force`
'--force-with-lease' refuses to overwrite if remote progressed unexpectedly, reducing risk of clobbering others' work.

`Use separate branches per PR`
Keep PRs focused and avoid accidental inclusion of unrelated commits.

22. Quick reference examples

`git push origin HEAD`
Push current branch to a remote branch with the same name.

`git push origin <local>:<remote>`
Push a local ref to an explicit remote ref name.

`git push origin :<remote>`
Delete remote branch <remote>.

`git reset --hard origin/<branch>`
Reset local branch to match remote branch exactly (dangerous: discards local changes).

End of notes — generated corrected PDF. If you want: (A) a one-page cheat-sheet, (B) code-block formatting for every command, or (C) split the file into beginner vs advanced sections, tell me which and I'll regenerate.