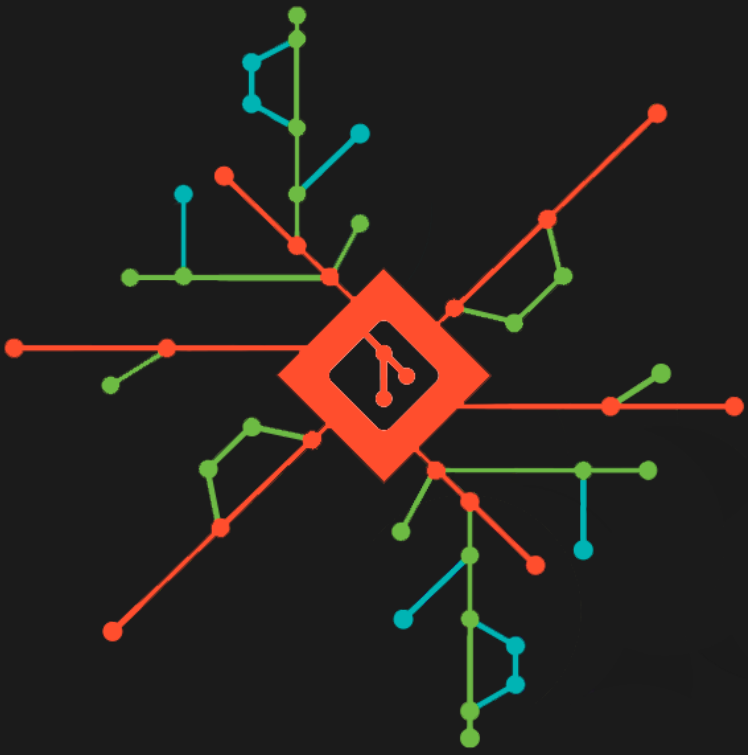
Gitmour

When the Code Gits Uncomfortably Sensitive



By R-m1n

Contents

Source

- codewithmosh

All of the contents of this handbook are entirely based on the teachings of everyone's favorite instructor **Mosh Hamedani**. Please check out his website https://codewithmosh.com/ and enjoy his wonderful courses on the topics of Software Engineering and Computer Science.

Introduction

What is Git:

Git is the most popular **Version Control System** in the world.

What is a Version Control System:

A Version Control System records the changes made to the code overtime in a special database called repository, with attributable changes, and if the result is not as expected, the project can easily be reverted back to an earlier state. There are two kinds of version control systems, **Centralized** and **Distributed**.

Centralized: Subversion, Team Foundation Server

Distributed: Git, Mercurial

Git Pros:

- Free
- Open Source
- Super Fast
- Scalable
- Cheap Branching/Merging

Using Git:

- CLI
- Code Editors and IDEs
- GUI

Why CLI:

- GUI tools have limitations.
- GUI tools are not always available.

Installation and Configuration

Git can be downloaded from https://git-scm.com/downloads.

Configuration Command:

```
git config --LEVEL SETTING VALUE
```

Setting Name:

```
git config --LEVEL user.name "FirstNAME LastNAME"
```

Setting Email:

```
git config --LEVEL user.email EMAIL
```

Setting Default Editor: (e.g. VS Code)

```
git config --LEVEL core.editor "code --wait"
```

Setting End-of-Line:

```
git config --LEVEL core.autocrlf tree       #Windows
git config --LEVEL core.autocrlf input      #Linux/Mac
```

Opening Config File in the Default Editor:

```
git config --LEVEL -e
```

Help:

```
git config --help  # Full Help
git config --h     # Summary
```

Initializing Git Repository in a Directory:

```
git init
```

Levels of Configuration Settings:

```
--system: All users
--global: All repositories of the current user
--local : The current repositories
```

WorkFlow

Staging Files:

Status of the Working Directory and the Staging Area:

```
git status
```

Short Status:

```
git status -s
```

output:

```
|C1|C2| FileNAME
# C1: Staging area | C2: Working directory
```

Adding Files to the Staging Area:

```
git add FileNAME
```

e.g.

```
git add file1.txt
git add file1.txt file2.txt
git add *.txt
git add .
```

Contents of the Staging Area:

```
git ls-files
```

Committing Stages:

One-Line Description Commit:

```
git commit -m "MESSAGE"
```

Multi-Line Description Commit:

```
git commit
# then we'll proceed to write the short and long
  description in the COMMIT_EDITMSG file.
```

Commit Best Practices:

- Commits should not be too small or too big.
- Commit often.
- Each commit should represent a logically separate chain set.
- Messages should be as meaningful as possible.
- Messages should be in present tense.

Skipping Staging Area:

```
git commit -a -m "MESSAGE"
# for committing all the tracked modified files.
```

Removing Files from both Working Directory and Staging Area:

```
git rm FileNAME
```

Removing Files only from Staging Area:

```
git rm --cached FileNAME
```

Renaming or Moving Files:

```
git mv OldFileNAME NewFileNAME
```

Ignoring Files:

To make Git ignore files/directories we don't want to commit, we should make a `.gitignore` file at the root of our project and then add the path of the files/directories that we want to ignore to this file.

`.gitignore` templates for various programming languages:

https://github.com/github/gitignore

Viewing Staged and Unstaged Changes:

Unstaged Changes:

```
git diff
# to compare files in the working directory to the
  files in the staging area.
```

Staged Changes:

```
git diff --staged
# to compare files in the staging area to the
  committed files in the repository.
```

Configuring a Visual DiffTool:

```
git config --global diff.tool vscode
git config --global difftool.vscode.cmd ->
"code --wait --diff $LOCAL $REMOTE"
```

Using a Visual DiffTool:

```
git difftool
git difftool --staged
```

Viewing History:

Full Log of Commits:

```
git log
```

One-Line Log of Commits:

```
git log --oneline
```

Reversing the Order of the Log of Commits:

```
git log --reverse
```

Viewing a Commit:

```
git show CommitID
git show HEAD~STEP
git show ObjectID
```

Contents of a Commited File:

```
git show HEAD~STEP:FileNAME
```

Git Objects in a Commit:

```
git ls-tree HEAD~STEP
```

Git Objects:

- Commits
- Blobs (Files)
- Trees (Directories)
- Tags

Restoring Files:

Unstaging Files:

```
git restore --staged .
git restore --staged FileNAME
```

Discard Local Changes:

```
git restore .
git restore FileNAME
```

note: restore command takes the copy of the file in the next environment and replaces it in the current environment.

Removing all the Untracked Files:

```
git clean
```

Restoring a File to an Earlier Version:

```
git restore --source=HEAD~STEP FileNAME
```

Browsing History

Viewing History:

```
git log

git log --oneline

git log --stat
# all the files that have been changed in each
  commit.

git log --patch
# each change that has been made to each file.
```

Filtering the History:

```
git log -NUMBER
# the last NUMBER commits.

git log --author="NAME"
# to filter by author name.

git log --before="DATE"
git log --after="DATE"
# to filter by date. (Dates could also be relative)

git log --grep="QUERY"
# to search in commit messages. (Case Sensitive)

git log -S"CONTENT"
# to search for all the commits that added or removed
  CONTENT.

git log StartID..TargetID
# to get a range of commits.

git log FileNAME
# all the commits that touched FILENAME.
```

note: If an error is encountered because git deems our file's name to be ambiguous, we should add double hyphens between the options and the file's name.

e.g.

```
git log --oneline -- test.txt
```

Formatting the Log Output:

```
git log --pretty=format:"FORMAT"
```

e.g.

```
"%an commited %H"
"%an commited %h"
"%an commited %h on %cd"
"%Cgreen%an%Creset committed %h on %cd"
```

Creating Alias:

```
git config --global alias.COMMAND "GitCOMMAND"
```

Viewing a Commit:

```
git show CommitID

git show HEAD~STEP

git show ObjectID

git show HEAD~STEP:FileNAME
```

options:

```
--name-only
# to show the name of the altered files in
  a commit.

--name-status
# to also get the status of the altered files
  in a commit.
```

Viewing Changes Across Commits:

```
git diff HEAD~STEP HEAD

git diff HEAD~STEP FileNAME


options:
      --name-only
      --name-status
```

Checking Out a Commit:

To make our working directory look exactly like a specific commit we should use the checkout command.

```
git checkout CommitID
```

note: The checkout command changes the HEAD pointer which changes the state to 'detached HEAD', any commits made in this state will become unreachable.

```
git checkout master
# to go back to the top of the master branch.
```

Finding Bugs using Bisect:

```
git bisect start

git bisect good CommitID

git bisect bad CommitID

git bisect reset
```

List of Contributors:

```
git shortlog

options:
      -c, --commiter
      # to group by author.

      -n, --numbered
      # sort by the number of commits per author.

      -s, --summary
      # suppress commit description.

      -e, --email
      # show the email address of each author.
```

Restoring a Deleted File:

```
git checkout CommitID FileNAME
# to restore a file from a commit that contained the
  file.
```

Blaming:

```
git blame FileNAME
# to find the information related to each line of
  a file including author's name, author's email, ... .

options:
      -e
      # to get the author's email.

      -L START,END
      # sort by the number of commits per author.
```

Tagging:

```
git tag TAG CommitID
```

```
git tag
# to get all the tags that have been created.
```

Annotated Tags:

```
git tag -a TAG -m “MESSAGE” CommitID
# to make a tag with a message.
```

```
git tag -n
# to get all the tags with their messages.
```

Deleting a Tag:

```
git tag -d TAG
```

Branching

What is Branching:

Branching allows us to diverge from the main line of work, and to work on something else in isolation.

Working with Branches:

Creating a Branch:

```
git branch BranchNAME
git branch
# the list of branches
```

note: The current branch is denoted by * before the name of the branch, also the "git status" shows the current branch.

Switching to a Branch:

```
git switch BranchNAME
# to move the HEAD pointer from to the target branch.
git switch -C BranchNAME
# to create and switch to a branch.
```

Renaming a Branch:

```
git branch -m OldNAME NewNAME
```

Deleting a Branch:

```
git branch -d BranchNAME
git branch -D BranchNAME
# force deletion
```

Comparing Branches:

```
git log BranchNAME..BranchNAME
# all the commits that are in a branch that are not
  in another branch.
git diff BranchNAME..BranchNAME
# all the changes that are in a branch that are not
  in another branch.
```

Stashing:

Stashing is temporarily saving changes in our working directory, so we can work on something else, and then come back and re-apply them later on. Stashing is useful if we need to quickly switch context and work on something else, but we're mid-way through a code change and aren't ready to commit.

```
git stash push -m ”MASSAGE”
git stash list
# list of stashed items.
git stash show StashNUMBER
# the changes between stashed items and our working
  directory.
git stash apply StashNUMBER
# applying stashed changes to the working directory.
git stash drop StashNUMBER
# deleting stashed items by their number.
```

Merging:

Merging is bringing changes from one branch to another. There are two types of merges in git, **Fast-Forward Merge** and **3-Way Merge**.

Fast-Forward Merge: When the branches have not diverged.

3-Way Merge: When the branches have diverged.

note: Using the --graph option of the "git log" command while inspecting branches could be useful as it would give a better representation of the branches and how they diverge.

```
git merge BranchNAME
# to merge a branch with the current branch.

git merge --no-ff BranchNAME
# to merge a branch with the current branch
  without using fast-forward merging.
```

note: Depending on the project it could be better not to use the fast-forward merge as it would make reverting commits harder and more complex, also we wouldn't have a true history.

Disabling Fast-Forward Merging:

```
git config --LEVEL ff no
```

Merged and Unmerged Branches:

```
git branch --merged
# list of branches that have been merged with the
  current branch.

git branch --no-merged
# list of branches that have not been merged with the
  current branch.
```

Merge Conflicts:

Conflicts could happen in a couple of scenarios:

1. When the same part of a file has been changed in different ways in two branches.
2. When a file is changed in a branch but is deleted in another branch.
3. When the same file is added twice in two different branches but the content of the file is different.

Merge conflicts could be resolved manually or by using visual tools.

Aborting a Merge:

```
git merge --abort
```

Undoing a Merge:

One way for undoing a merge is by removing the last commit which is only recommended if we're in our local repository.

```
git reset --ResetOPTION HEAD~STEP
# resetting the HEAD pointer to another commit.
```

`reset options:`

`--soft:`

To change the HEAD/master pointer without affecting the staging area and the working directory.

`--mixed:`

To change the HEAD/master pointer, and put the snapshot of the target commit in the staging area, the working directory will be unchanged.

`--hard:`

To change the HEAD/master pointer, and put the snapshot of the target commit in both the staging area and the working directory.

Another way for undoing a merge is by reverting the last commit. Which is the way to go, in a shared repository.

```
git revert -m ParentNUMBER HEAD
```

Squash Merging:

When encountered with small short-lived branches with bad history, we can use squash merging technique to get rid of these branches and continue on a clean linear history, like bug fixes or small easily implementable features.

```
git merge --squash BranchNAME
# to combine changes across files in the target
  branch to the files in the current branch, note
  that the changes would be in the staging and they
  won't be committed to the repository.
```

note: When doing a squash merge, it's important to remove target branches afterwards as they would cause confusion later on.

Rebasing:

Using the rebase technique we can change the base of the current branch to the latest commit on the target branch or to a commit.

In some scenarios could be useful for having a linear and a cleaner history, this technique is only recommended on our local repository, as rebasing is rewriting history.

```
git rebase BranchNAME
git rebase CommitID
```

Merge Conflicts in Rebasing:

```
git rebase --continue
# to continue rebasing after resolving the conflict.
git rebase --skip
# to skip the commit and move on to the next one.
git rebase --abort
# to abort the rebasing operation.
```

Cherry Picking:

To take a particular commit and apply it on top of the current branch.

```
git cherry-pick CommitID
```

To cherry pick a file from another branch into the current branch.

```
git restore --source=BranchNAME -- FileNAME
git restore --source=CommitID -- FileNAME
```

Collaboration

Cloning:

```
git clone URL
git clone URL LocalRepoNAME
```

note: Branches and pointers that are prefixed with origin/ are from the remote repository and cannot be switched to, or altered.

Getting the List of Remote Branches:

```
git remote

options:
      -v, --verbose
      # to get more details about the remote repo.
```

Fetching:

```
git fetch
# to download the changes made to to the remote
  repository.
git fetch RemoteNAME
git fetch RemoteNAME BranchNAME
```

Divergence of the Local and the Remote Tracking Branches:

```
git branch -vv
```

note: To move our local tracking branch to the remote tracking branch we should merge the two branches.

Remote Tracking Branches:

```
git branch -r
```

Pulling:

Pull is basically combining the fetch and the merge command.

```
git pull
git pull RemoteNAME
git pull RemoteNAME BranchNAME

options:
      --rebase
      # to do rebasing instead of a 3-way merge.
```

Pushing:

Applying changes on our local repository to the remote repository.

```
git push
git push RemoteNAME
git push RemoteNAME BranchNAME
```

Storing Credentials:

```
git config --global credential.helper StoreOPTION

safe options:
      Windows:    Windows Credentials Store
      Mac:        Keychain
      Linux:      libsecret, genome-keyring

unsafe options:
      cache # temporarily (default: 15min)
      store # permanently
```

Sharing Tags:

By default "`git push`" command doesn't transfer tags to the remote repository, tags should be explicitly pushed.

```
git push RemoteNAME BranchNAME TAG
```

Deleting a Tag from the Remote Repository:

```
git push RemoteNAME BranchNAME --delete TAG
```

Sharing Branches:

Similar to tags, branches are not pushed to the remote repository, they have to be explicitly pushed.

```
git push --set-upstream RemoteNAME BranchNAME
```

note: Upstream of a branch is only needed to be set the first time that we're pushing the branch.

Deleting a Branch in the Remote Repository:

```
git push -d RemoteNAME BranchNAME
```

Setting up a Branch to Track a Remote Branch:

```
git switch -C LocalBRANCH RemoteBRANCH
```

Removing a Remote Tracking Branch that is no Longer in the Remote Repository:

```
git remote prune RemoteNAME
```

Adding Remote Tracking Branch of a Base Repository:

```
git remote add RemoteNAME URL
```

Renaming a Remote Tracking Branch:c

```
git remote rename OldNAME NewNAME
```

Removing a Remote Tracking Branch:

```
git remote rm RemoteNAME
```

Rewriting History

The history of a repository is used to see what, why and when was charged. And if our history doesn't match a certain criteria we should rewrite it to get a clean and concise history.

Bad History:

- Poor and meaningless commit messages.
- Large commits with unrelated changes.
- Small and scattered commits.

Tips for a Cleaner History:

- Squash small, related commits.
- Split large commits, containing unrelated changes.
- Reword commit messages.
- Drop unwanted commits.
- Modify commits.

Golden Rule of Rewriting History:

Don't Rewrite Public History

Undoing Commits:

Resetting:

```
git reset --ResetOPTION HEAD~STEP
```

```
reset options:
      --soft       # Removes the commit
      --mixed      # Unstage files
      --hard       # Discards local changes
```

Reverting Commits:

```
git revert HEAD~STEP
git revert HEAD~START..HEAD~TARGET
```

```
note: HEAD~START is not counted in the reverting process.
      options:
            --no-commit
            # with this option git will add the required
              changes in the staging area, so for every
              commit that is reverted, git figures out the
              changes that have to be undone, then it would
              apply those changes in the staging area.
              (This is a cleaner way of reverting changes
              in a public repository.)
            --abort
            # to abort the revering process.
            --continue
            # to apply changes.
```

Amending the Last Commits:

```
git commit --amend
```

Interactive Rebasing:

```
git rebase -i CommitID
```

note: for targeting a commit via interactive rebasing we should pass the commit ID of its parent.

Amending an Earlier Commit:

For amending a commit we should change the command before its ID in the `git-rebase-todo` to "edit".

After making the changes:

```
git commit --amend
```

Once satisfied with the changes we should continue with rebsing operation:

```
git rebase --continue
```

Or to abort the rebasing operation:

```
git rebase --abort
```

note: the changes that we make to an earlier commit carries on through the rest of the repository.

Dropping a Commit:

For dropping a commit we should change the command before its ID in the `git-rebase-todo` to "`drop`". or we could just delete the line associated with that commit in the `git-rebase-todo.`

After resolving possible conflicts caused by dropping a commit manually, or by using "`git mergetool`":

```
git rebase --continue
```

Rewording Commit Messages:

For rewording a commit, we should change the command before its ID in the `git-rebase-todo` to "`reword`".

Reordering Commits:

We can change the order of commits by moving a commit to the desired line in the `git-rebase-todo`.

```
note: commits in the git-rebase-todo are in ascending order.
```

Squashing Commits:

"`squash`" command before a commit in `git-rebase-todo` will squash the commit with the previous commit.

Reference Log of Operations:

"`squash`" command before a commit in `git-rebase-todo` will squash the commit with the previous commit.

```
git reflog
```

Squashing Commits using Fixup Command:

By using the "`fixup`" command for the squashing operation, git will take the message of the first commit before the commit with the fixup command and apply it to the squashed commit.

Splitting a Commit:

1. `edit -> git-rebase-todo`
2. `git restore HEAD^`
3. `making desired changes`
4. `git rebase --continue`

`note: ^ after a commit ID means the parent of that commit.`

Cheat Sheet

Initializing a Repository

```
git init
```

Staging Files

```
git add FileNAME
# stages a single file.

git add FileNAME FileNAME
# stages multiple files.

git add *.FORMAT
# stages with a pattern.

git add .
# stages the current directory and all its content.
```

Viewing the Status

```
git status
# full status.

git status -s
# short status.
```

Committing the Staged Files

```
git commit -m “MESSAGE”
# commits with a one-line message.

git commit
# opens the default editor to type a long message.
```

Skipping the Staging Area

```
git commit -am "MESSAGE"
```

Removing Files

```
git rm FileNAME
# removes from working directory and staging area.

git rm --cached FileNAME
# removes from staging area only.
```

Renaming or Moving Files

```
git mv OldFileNAME NewFileNAME
```

Viewing the Staged/Unstaged Changes

```
git diff
# shows unstaged changes.

git diff --staged
# shows staged changes.

git diff --cached
# same as the above.
```

Viewing the History

```
git log
# full history.

git log --oneline
# summary.

git log --reverse
# lists the commits from the oldest to the newest.
```

Viewing a Commit

```
git show CommitID
# shows the given commit.

git show HEAD
# shows the last commit.

git show HEAD~STEP
# two steps before the last commit.

git show HEAD:FileNAME
# shows the version of the file stored in the last
  commit.
```

Unstaging Files

```
git restore --staged FileNAME
# copies the last version of file.js from repo to
  index.
```

Discarding Local Changes

```
git restore FileNAME
# copies the file from index to working directory.

git restore FileNAME FileNAME
# restores multiple files in working directory.

git restore
# discards all local changes.

git clean -fd
# removes all untracked files.
```

Restoring an Earlier Version of a File

```
git restore --source=HEAD~STEP FileNAME
```

Viewing the History

```
git log --stat
# shows the list of modified files.
git log --patch
# shows the actual changes (patches).
```

Filtering the History

```
git log -NUMBER
# shows the last 3 entries.
git log --author="NAME"
git log --before="DATE"
git log --after="DATE"
git log --grep="QUERY"
# commits with "QUERY" in their message.
git log -S"QUERY"
# commits with "QUERY" in their patches.
git log StartID..TargetID
# range of commits.
git log FileNAME
# commits that touched the file.
```

Formatting the Log Output

```
git log --pretty=format:"FORMAT"
```

Creating an Alias

```
git config --global alias.COMMAND "GitCOMMAND"
```

Viewing a Commit

```
git show HEAD~STEP

git show HEAD~STEP:FileNAME
# shows the version of the file stored in this
  commit.
```

Comparing Commits

```
git diff HEAD~STEP HEAD
# shows the changes between two commits.

git diff HEAD~STEP HEAD FileNAME
# changes to the file only.

git checkout CommitID
# checks out the given commit.

git checkout master
# checks out the master branch.
```

Finding a Bad Commit

```
git bisect start

git bisect bad
# marks the current commit as a bad commit.

git bisect good CommitID
# marks the given commit as a good commit.

git bisect reset
# terminates the bisect session.
```

List of Contributors

```
git shortlog
```

Viewing the History of a File

```
git log FileNAME
# shows the commits that touched file.txt.

git log --stat FileNAME
# shows statistics (the number of changes) for
  the file.

git log --patch FileNAME
# shows the patches (changes) applied to the file.
```

Finding the Author of Lines

```
git blame FileNAME
# shows the author of each line in the file.
```

Tagging

```
git tag TAG
# tags the last commit.

git tag TAG CommitID
# tags an earlier commit.

git tag
# lists all the tags.

git tag -d TAG
# deletes the given tag.
```

Managing Branches

```
git branch BranchNAME
# creates a new branch.

git checkout BranchNAME
# switches to the branch.
```

```
git switch BranchNAME
# same as the above.

git switch -C BranchNAME
# creates and switches.

git branch -d BranchNAME
# deletes the branch.
```

Comparing Branches

```
git log master..BranchNAME
# lists the commits in the branch that are not in
  master.

git diff master..BranchNAME
# shows the summary of changes.
```

Stashing

```
git stash push -m "MESSAGE"
# creates a new stash.

git stash list
# lists all the stashes.

git stash show stash@{StashNUMBER}
# shows the given stash.

git stash show StashNUMBER
# shortcut for stash@{StashNUMBER}.

git stash apply StashNUMBER
# applies the given stash to the working directory.

git stash drop StashNUMBER
# deletes the given stash.

git stash clear
# deletes all the stashes.
```

Merging

```
git merge BranchNAME
# merges the branch into the current branch.

git merge --no-ff BranchNAME
# creates a merge commit even if FF is possible.

git merge --squash BranchNAME
# performs a squash merge.

git merge --abort
# aborts the merge.
```

Viewing the Merged Branches

```
git branch --merged
# shows the merged branches.

git branch --no-merged
# shows the unmerged branches.
```

Rebasing

```
git rebase master
# changes the base of the current branch.
```

Cherry Picking

```
git cherry-pick CommitID
# applies the given commit on the current branch.
```

Cloning a Repository

```
git clone URL
```

Syncing with Remotes

```
git fetch origin master
# fetches master from origin.

git fetch origin
# fetches all objects from origin.

git fetch
# shortcut for “git fetch origin”.

git pull
# fetch + merge.

git push origin master
# pushes master to origin.

git push
# shortcut for “git push origin master”.
```

Sharing Tags

```
git push origin TAG
# pushes tag to origin.

git push origin --delete TAG
```

Sharing Branches

```
git branch -r
# shows remote tracking branches.

git branch -vv
# shows local & remote tracking branches.

git push -u origin BranchNAME
# pushes the branch to origin.

git push -d origin BranchNAME
# removes the branch from origin.
```

Managing Remotes

```
git remote
# shows remote repos.

git remote add upstream URL
# adds a new remote called upstream.

git remote rm upstream
# remotes upstream.
```

Undoing Commits

```
git reset --soft HEAD^
# removes the last commit, keeps changed staged.

git reset --mixed HEAD^
# unstages the changes as well.

git reset --hard HEAD^
# discards local changes.
```

Reverting Commits

```
git revert CommitID
# reverts the given commit.

git revert HEAD~STEP..
# reverts the last three commits.

git revert --no-commit HEAD~STEP..
```

Recovering Lost Commits

```
git reflog
# Shows the history of HEAD.

git reflog show BranchNAME
# Shows the history of branch pointer.
```

Amending the Last Commit

```
git commit --amend
```

Interactive Rebasing

```
git rebase -i HEAD~STEP
```