Git Cheat Sheet

The essential Git commands every developer must know

This cheat sheet covers all of the Git commands I've covered in my Ultimate Git Mastery course.

- ✓ Creating snapshots
- ✓ Browsing history
- ✓ Branching & merging
- ✓ Collaboration using Git & GitHub
- ✓ Rewriting history

Hi! My name is Mosh Hamedani. I'm a software engineer with two decades of experience. I've taught millions of people how to code or how to become a professional software engineer through my YouTube channel and online coding school. It's my mission to make software engineering simple and accessible to everyone.

Check out the links below to master the coding skills you need:

https://codewithmosh.com

https://youtube.com/user/programmingwithmosh

https://twitter.com/moshhamedani

https://facebook.com/programmingwithmosh/

Want to master Git?

Stop wasting your time memorizing Git commands or browsing disconnected tutorials. If you don't know how Git works, you won't get far.

My **Ultimate Git Mastery** course teaches you everything you need to know to use Git like a pro.

- ✓ Learn & understand Git inside out
- ✓ Master the command line
- ✓ Version your code and confidently recover from mistakes
- ✓ Collaborate effectively with others using Git and GitHub
- ✓ Boost your career opportunities

Click below to enroll today:

https://codewithmosh.com/p/the-ultimate-git-course/

Table of Content

Creating Snapshots

Initializing a repository

```
git init                    rm -rf .git    , removes repository
```

Staging files

```
git add file1.js            # Stages a single file
git add file1.js file2.js   # Stages multiple files
git add *.js                # Stages with a pattern
git add .                   # Stages the current directory and all its content
```

Viewing the status

git ls-files, viewing files in stage area

```
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 file1.js             # Removes from working directory and staging area
git rm --cached file1.js    # Removes from staging area only
```

rm file1.js, removes file from working directory only

Renaming or moving files

```
git mv file1.js file1.txt
```

this renaming or moving is reflected in both working directory and staging area

Viewing the staged/unstaged changes

```
git diff                  # Shows unstaged changes
git diff --staged         # Shows staged changes visually in terminal window,
git diff --cached         usually use visual tools for this
                          # Same as the above
```

Viewing the history

```
git log
git log --oneline         # Full history
git log --reverse         # Summary
                          # Lists the commits from the oldest to the newest
```

git log --all, shows all commits across all branches, and commits ahead of HEAD on current branch as well

Viewing a commit

```
git show 921a2ff
git show HEAD             # Shows the given commit
git show HEAD~2           # Shows the last commit
git show HEAD:file.js     # Two steps before the last commit
                          # Shows the version of file.js stored in the last commit
```

git ls-tree HEAD~2, shows all files within the last commit

Unstaging files (undoing git add)

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

Discarding local changes

```
git restore file.js         # Copies file.js from index to working directory
git restore file1.js file2.js   # Restores multiple files in working directory
git restore .               # Discards all local changes (except untracked files)
git clean -fd               # Removes all untracked files  from working directory
```

Restoring an earlier version of a file

```
git restore --source=HEAD~2 file.js

git checkout earlierCommitName fileNameYouWishToRestoreIfDeleted
```

Browsing History

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 -3                  # Shows the last 3 entries
git log --author="Mosh"
git log --before="2020-08-17"
git log --after="one week ago"
git log --grep="GUI"        # Commits with "GUI" in their message
git log -S"GUI"             # Commits with "GUI" in their patches
git log hash1..hash2        # Range of commits
git log file.txt            # shows all commits that involve this file
```

git log -- file.txt, if filename is ambiguous and git complains about file name

Formatting the log output

```
git log --pretty=format:"%an committed %H"
```

*there are a lot of formatting options, look up in documentation

Creating an alias

```
git config --global alias.lg "log --oneline" , #to run the alias we would run...git lg
```

Viewing a commit

git show HEAD~2 , go to HEAD which is last commit and go two commits back, shows all changes made

```
git show HEAD~2:file1.txt   # Shows the version of file stored in this commit
```

git show HEAD~2 --name-only, shows only name of files that were modified for this commit, but if you want it to show if they were added or deleted or modified use git show HEAD~2 --name-status

Comparing commits

```
git diff HEAD~2 HEAD        # Shows the changes between two commits
```

```
git diff HEAD~2 HEAD file.txt # Changes to file.txt only
```

, you can replace file.txt with specific options such as --name-only, --name-statsus, etc

Checking out a commit

```
git checkout dad47ed         # Checks out the given commit (this will restore working
git checkout master          directory to a snapshot of given commit)(points HEAD to a given
                             commit)
                              # Checks out the master branch , realigns HEAD with last
                             commit of master branch
```

Finding a bad commit

```
git bisect start
git bisect bad               # Marks the current commit as a bad commit
git bisect good ca49180      # Marks the given commit as a good commit
git bisect reset             # Terminates the bisect session
```

Finding contributors

```
git shortlog
```

Viewing the history of a file

```
git log file.txt             # Shows the commits that touched file.txt
git log --stat file.txt      # Shows statistics (the number of changes) for file.txt
git log --patch file.txt     # Shows the patches (changes) applied to file.txt
```

Finding the author of lines

```
git blame file.txt           # Shows the author of each line in file.txt
```

Tagging(labels for commits)

```
git tag v1.0                 # Tags the last commit as v1.0
git tag v1.0 5e7a828         # Tags an earlier commit
git tag                      # Lists all the tags
git tag -d v1.0              # Deletes the given tag
```

Branching & Merging

git branch -m oldBranchName newBranchName, renames an existing branch

git branch, lists all branches

Managing branches

```
git branch bugfix          # Creates a new branch called bugfix
git checkout bugfix        # Switches to the bugfix branch
git switch bugfix          # Same as the above, but use this approach
git switch -C bugfix       # Creates and switches
git branch -d bugfix       # Deletes the bugfix branch, -D option force deletes
```

git log --all, shows all commits across all branches, and commits ahead of HEAD on current branch as well

Comparing branches

```
git log master..bugfix     # Lists the commits in the bugfix branch not in master
git diff master..bugfix    # Shows the summary of changes, shorthand, you can omit the
                           name of the branch your in, ex...git diff bugfix
```

git switch -C newBranchName, creates branch and switches to that branch, you can also suffix with remote tracking branch reference

Stashing

```
git stash push -m “New tax rules”   # Creates a new stash, to include untracked files use -am
    git stash list                  # Lists all the stashes
    git stash show stash@{1}        # Shows the given stash
    git stash show 1                # shortcut for stash@{1}
    git stash apply 1               # Applies the given stash to the working dir
    git stash drop 1                # Deletes the given stash
    git stash clear                 # Deletes all the stashes
```

git log --all --graph, graph option gives us a better representation of branches and how they diverge

Merging

(merge the commit that is ahead into the previous commit)

```
git merge bugfix          # Merges the bugfix  branch into the current branch
git merge --no-ff bugfix  # Creates a merge commit even if FF is possible
git merge --squash bugfix # Prepares a squash merge, then follow up with commit ,
remember to delete feature branch so there isn't confusion about unmerged branch in future
git merge --abort         # Aborts the merge and reverts back to state b4 merge
```

git config --global ff no, disables FF merges for your repositories
git config ff no, disables FF merges for this repository

**SQUASH MERGING, use with small shortlived branches with bad history, such as bug fixes, or very small features, gets rid of reference to feature branch so end product is linear branch

Viewing the merged branches

```
git branch --merged        # Shows the merged branches
git branch --no-merged     # Shows the unmerged branches
```

Rebasing # Changes the base of the current branch to last commit of master

```
git rebase master
```

used to simplify merge commits into linear merge by changing base to last commit of MASTER branch
rebasing rewrites history, so only use when commits are local in your repository

Cherry picking , bringing a particular commit from feature branch and applying this commit after HEAD on MASTER

```
git cherry-pick dad47ed    # Applies the given commit on the current branch HEAD
```

UNDOING A FAULTY MERGE
git reset --hard HEAD~1, changes working directory, staging area, and HEAD pointer (current snapshot) all to same state as last commit before HEAD, this completely undoes the last commit, use this if it's local and there's no shared history on the commit with collaborators

git revert -m 1 HEAD, to undo the last commit that merged two branches, use this if there is shared history of the commit

Collaboration

Cloning a repository

git clone url	by default clones project with same name, however you can change name by adding name after url

clones the master branch, and other branch references so you still need to create that branch in local machine and set to track the remote branch reference

Syncing with remotes

git fetch origin master	# Fetches master branch from origin (specify branch)
git fetch origin	# Gets all commits from that repository(and objects)
git fetch	# Shortcut for “git fetch origin”
git pull	# Fetch + merge
git push origin master	# Pushes master branch to origin
git push	# Shortcut for “git push origin master”

You can't push your master local branch if your master and ORIGIN/MASTER diverge(commits were added to origin/ MASTER so your master HEAD isn't same as ORIGIN/ MASTER HEAD

git pull --rebase, rebases our local changes on top of changes made by others to remote repository

DONT USE git push --force OPTION IF YOU DON'T HAVE TO

Sharing tags

git push origin v1.0	# Pushes tag v1.0 to origin
git push origin —delete v1.0	

TAGS go hand in hand with RELEASE NOTES, a high level feature of GitHub to package software, source code, release notes in group with name of TAG

Sharing branches

git branch -r	# Shows all remote tracking branches
git branch -vv	# Shows local & remote tracking branches
git push -u origin bugfix	#Pushes bugfix branch(local) to origin
git push -d origin bugfix	# Removes bugfix branch from origin

-vv, shows whether the branch is linked to remote tracking branch (pointer that points to remote branch), also shows if the branches on same commit

ORIGIN, remote tracking branch(points to remote tracking branch via url)

remote repositories, or repositories not on local machine, or more accurately not currently in working directory

Managing remotes

git remote	# Shows list of remote repos,
git remote add upstream url	# Adds a new remote called upstream
git remote rm upstream	# Remotes upstream

-v, shows the remote repositories url location

git remote set-url origin newUrl, sets origin url to different location of newUrl

git remote prune origin, to get rid of tracking branch references that are no longer on remote repository

git remote rename upstream base, renames upstream remote tracking branch to base

HEAD^ can be HEAD~#, or HEAD@{#} or commit#

Rewriting History

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
```

HARD- discards local changes
Mixed- unstages files
soft- removes the commit only

hard option updates current snapshot, and puts snapshot in working directory, and staging area
mixed option updates current snapshot, and puts snapshot in staging area,
soft option updates current snapshot only

Reverting commits

```
git revert 72856ea           # Reverts the given commit
✶ git revert HEAD~3.. HEAD   # Reverts the last three commits
✶ git revert --no-commit HEAD~3..
```

revert, return to a previous state

followed by....
git revert --continue, if we are happy with previous revert command

✶ have same effect, however latter command only creates one commit to revert 3, and the first makes three commits

Recovering lost commits

```
git reflog                   # Shows the history of HEAD
git reflog show bugfix       # Shows the history of bugfix branch pointer
```

Amending the last commit

```
git commit --amend
```

remember that commits are immutable, so github actually creates new commit with your changes under the hood

Interactive rebasing

used to ammend an earlier commit,
used to drop a commit

```
git rebase -i HEAD~5
```

STORING CREDENTIALS
git config --global credential.helper cache, stores credentials 15 minutes in memory
IF YOU WANT TO STORE LONGER....
FOR WINDOWS, WINDOWS CREDENTIAL STORE

GIT is the most popular version control system in the world

HOW TO USE: the command line, Code Editors & IDEs (GitLens is popular for enabling Git into VSCode), GUISs(GitKraken) or (SourceTree for WIndows and Mac)

WHY COMMAND LINE IS BEST TO LEARN FIRST: GUI tools have limitations, GUI tools are not always available so knowing command line is essential so your not stuck

SETTING LEVELS
system, all users,
global, all repositories of the current user
local, the current repository

COMMANDS I RAN TO SET INITIAL CONFIG SETTINGS
```
dominick@DESKTOP-A3P2PAH MINGW64 /
$ git config --global user.name "Jose Cortez"


dominick@DESKTOP-A3P2PAH MINGW64 /
$ git config --global user.email jdcortez2268@gmail.com

dominick@DESKTOP-A3P2PAH MINGW64 /
$ git config --global core.editor "code --wait"

dominick@DESKTOP-A3P2PAH MINGW64 /
$ git config --global core.autocrlf true

dominick@DESKTOP-A3P2PAH MINGW64 ~
$ git config --global diff.tool vscode

dominick@DESKTOP-A3P2PAH MINGW64 ~
$ git config --global difftool.vscode.cmd "code --wait --diff $LOCAL $REMOTE"

dominick@DESKTOP-A3P2PAH MINGW64 ~
$ git config --global -e
```

WHERE TO GET HELP
github documentation
or short synopsis on command line, git config -h
Index is old name for staging area in documentation

CLONING, taking initial copy of central repository to local machine. When you make a clone, you can edit the files in your preferred editor and use Git to keep track of your changes without having to be online. The repository you cloned is still connected to the remote version so that you can push your local changes to the remote to keep them synced when you're online.

PUSH, sends local commits to remote repository

FETCH, gets changes from remote repository to local branch without committing changes, under hood is just getting new commits and moving HEAD/ORIGIN to most recent commit, and then you still have to merge

PULL, fetching any changes made and merges them into your local machine

-FORK, gets complete copy of the repository and places it in your account. You have full control over this repository

-PULL REQUEST, proposed changes to a repository submitted by a user and accepted or rejected by a repository's collaborators. Like issues, pull requests each have their own discussion forum.

-HEAD pointer points to a commit of the branch you are working with, by default is the last commit, however you can move HEAD pointer around

-STASHING CHANGES, occurs when you have uncommited changes to current branch but your trying to switch branches, then instead of committing changes you can STASH changes, or store them in a safe place for later

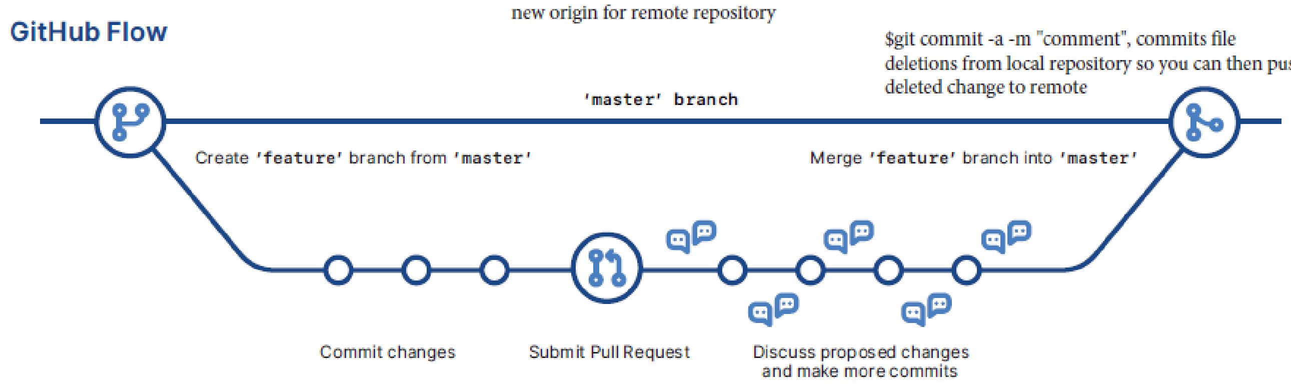
-ISSUE TRACKING, track all kinds of issues or any ideas we want to discuss on team
-you can link with pull requests so that a successful merge pull request will close issue
-you can assign issue to a MILESTONE, used to document progress, groups issues together and you can see how many have been completed/ not completed
-you can customize labels to help with issue tracking

GIT WORKFLOW

WORKING DIRECTORY, STAGING AREA(INDEX), LAST SNAPSHOT(where **HEAD** pointer points to by default)

$git remote set-url origin URL, sets new origin for remote repository

$git commit -a -m "comment", commits file deletions from local repository so you can then push deleted change to remote


GitHub Flow
'master' branch
Create 'feature' branch from 'master'
Merge 'feature' branch into 'master'
Commit changes
Submit Pull Request
Discuss proposed changes and make more commits

whenever making a commit, there is staging area, when we commit we copy that snapshot to HEAD and the staging area is still full with what you committed

you can delete file from repository, by deleting from your local and use add command to stage changes, git add fileName

WHEN TO COMMIT? whenever app reaches a different state

GitIgnore File, tells machine which files to ignore adding to staging area
github/gitignore repository has templates on typical gitignore filles based on the project

Github Objects: trees(directories), blobs(files), tags, commit

BRANCHES, you can think of branches as seperate isolated workspaces where you can work on app without causing bugs in main application on main branch

-whenever you create a new branch, Git creates new pointer with that branch Name and it points to your HEAD OF MASTER BRANCH

2 TYPES OF MERGES, fast-forward and 3-way.

FAST-FORWARD MERGE, when there is direct linear path from Master to new Branch (No additional commits to Master have been made), simply moves pointer of Master branch forward to other branches's pointer

3-WAY MERGE, when there is not direct linear path from Master to new Branch because additional commits on master have been made and branches diverge. Combines the last shared commit, last commit of Master, and last commit of newBranch to form **MERGE COMMIT**

MERGE CONFLICTS,

-CHANGE1,CHANGE2, when same code is modified differently on diff branches
-CHANGE,DELETE
-ADD1,ADD2, when same fileName is added to diff branches with different code

SOME POPULAR TOOLS: Kdiff, P4Merge, **WinMerge**(Windows Only)

INTEGRATION-MANAGER PROJECT, used for open source projects when contributors don't have write access to central repository
Whenever contributer wants to add changes,
-FORK, gets complete copy of the repository and places it in your account. You have full control over this repository
-clone the forked repository, make commits, and **push** to forked repository
-make **PULL REQUEST** to MAINTAINER
MAINTAINER pulls forked repository changes and the pushes to central repository

-to keep forked repository up to date with central repository, pull from BASE/UPSTREAM repository and push to ORIGIN (your forked repository)

HOW TO GIVE PUSH ACCESS
SETTINGS, MANAGE ACCESS, INVITE A COLLABORATOR, now they will have push access to this repository

REWRITING HISTORY
GOOD HISTORY SHOWS, why what and when, clean readable history
BAD HISTORY, poor commit messages, large commits, or too small commits
TO CREATE GOOD HISTORY...
-combine small, related commits

-split large commits
-reword commit messages
-drop unwanted commits
-modify commits
GOLDEN RULE: DON'T REWRITE PUBLIC HISTORY!,