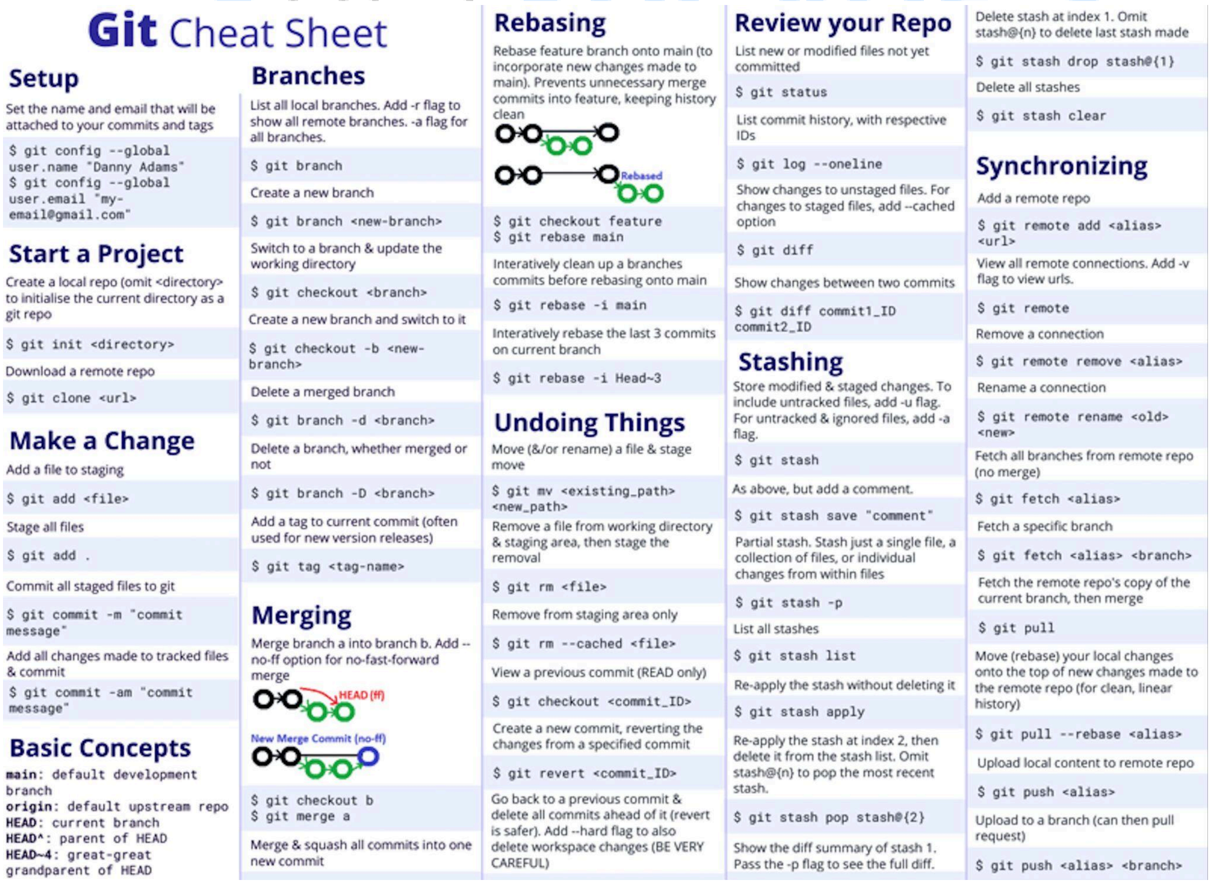
CONTENTS

[Like and Subscribe]



Git Cheat Sheet

Setup

Set the name and email that will be attached to your commits and tags

```
$ git config --global
user.name "Danny Adams"
$ git config --global
user.email "my-
email@gmail.com"
```

Start a Project

Create a local repo (omit <directory> to initialise the current directory as a git repo

```
$ git init <directory>
```

Download a remote repo

```
$ git clone <url>
```

Make a Change

Add a file to staging

```
$ git add <file>
```

Stage all files

```
$ git add .
```

Commit all staged files to git

```
$ git commit -m "commit
message"
```

Add all changes made to tracked files & commit

```
$ git commit -am "commit
message"
```

Basic Concepts

```
main: default development
branch
origin: default upstream repo
HEAD: current branch
HEAD^: parent of HEAD
HEAD~4: great-great
grandparent of HEAD
```

Branches

List all local branches. Add -r flag to show all remote branches. -a flag for all branches.

```
$ git branch
```

Create a new branch

```
$ git branch <new-branch>
```

Switch to a branch & update the working directory

```
$ git checkout <branch>
```

Create a new branch and switch to it

```
$ git checkout -b <new-
branch>
```

Delete a merged branch

```
$ git branch -d <branch>
```

Delete a branch, whether merged or not

```
$ git branch -D <branch>
```

Add a tag to current commit (often used for new version releases)

```
$ git tag <tag-name>
```

Merging

Merge branch a into branch b. Add --no-ff option for no-fast-forward merge

HEAD (ff)

New Merge Commit (no-ff)

```
$ git checkout b
$ git merge a
```

Merge & squash all commits into one new commit

Rebasing

Rebase feature branch onto main (to incorporate new changes made to main). Prevents unnecessary merge commits into feature, keeping history clean

Rebased

```
$ git checkout feature
$ git rebase main
```

Interatively clean up a branches commits before rebasing onto main

```
$ git rebase -i main
```

Interatively rebase the last 3 commits on current branch

```
$ git rebase -i Head~3
```

Undoing Things

Move (&/or rename) a file & stage move

```
$ git mv <existing_path>
<new_path>
```

Remove a file from working directory & staging area, then stage the removal

```
$ git rm <file>
```

Remove from staging area only

```
$ git rm --cached <file>
```

View a previous commit (READ only)

```
$ git checkout <commit_ID>
```

Create a new commit, reverting the changes from a specified commit

```
$ git revert <commit_ID>
```

Go back to a previous commit & delete all commits ahead of it (revert is safer). Add --hard flag to also delete workspace changes (BE VERY CAREFUL)

Review your Repo

List new or modified files not yet committed

```
$ git status
```

List commit history, with respective IDs

```
$ git log --oneline
```

Show changes to unstaged files. For changes to staged files, add --cached option

```
$ git diff
```

Show changes between two commits

```
$ git diff commit1_ID
commit2_ID
```

Stashing

Store modified & staged changes. To include untracked files, add -u flag. For untracked & ignored files, add -a flag.

```
$ git stash
```

As above, but add a comment.

```
$ git stash save "comment"
```

Partial stash. Stash just a single file, a collection of files, or individual changes from within files

```
$ git stash -p
```

List all stashes

```
$ git stash list
```

Re-apply the stash without deleting it

```
$ git stash apply
```

Re-apply the stash at index 2, then delete it from the stash list. Omit stash@{n} to pop the most recent stash.

```
$ git stash pop stash@{2}
```

Show the diff summary of stash 1. Pass the -p flag to see the full diff.

Delete stash at index 1. Omit stash@{n} to delete last stash made

```
$ git stash drop stash@{1}
```

Delete all stashes

```
$ git stash clear
```

Synchronizing

Add a remote repo

```
$ git remote add <alias>
<url>
```

View all remote connections. Add -v flag to view urls.

```
$ git remote
```

Remove a connection

```
$ git remote remove <alias>
```

Rename a connection

```
$ git remote rename <old>
<new>
```

Fetch all branches from remote repo (no merge)

```
$ git fetch <alias>
```

Fetch a specific branch

```
$ git fetch <alias> <branch>
```

Fetch the remote repo's copy of the current branch, then merge

```
$ git pull
```

Move (rebase) your local changes onto the top of new changes made to the remote repo (for clean, linear history)

```
$ git pull --rebase <alias>
```

Upload local content to remote repo

```
$ git push <alias>
```

Upload to a branch (can then pull request)

```
$ git push <alias> <branch>
```

6 Painful Git Interview Questions and Answers You need to know

Q1: What is Git fork? What is the difference between fork, branch and clone?

A fork is a remote, server-side copy of a repository, distinct from the original. A fork isn't a Git concept really, it's more a political/social idea.
A clone is not a fork; a clone is a local copy of some remote repository. When you clone, you are actually copying the entire source repository, including all the history and branches.
A branch is a mechanism to handle the changes within a single repository in order to eventually merge them with the rest of code. A branch is something that is within a repository. Conceptually, it represents a thread of development.

Q2: What's the difference between a "pull request" and a "branch"?

A branch is just a separate version of the code.
A pull request is when someone take the repository, makes their own branch, does some changes, then tries to merge that branch in (put their changes in the other person's code repository).

Q3: How to revert previous commit in git?

```
git reset –hard HEAD~1
git reset HEAD~1
git reset –soft HEAD~1
```

Q4: What is "git cherry-pick"?

The command git cherry-pick is typically used to introduce particular commits from one branch within a repository onto a different branch. A common use is to forward- or back-port commits from a maintenance branch to a development branch.

```
git cherry-pick <commit-hash>
```

Q5: Explain the advantages of Forking Workflow

The Forking Workflow is fundamentally different than other popular Git workflows. Instead of using a single server-side repository to act as the “central” codebase, it gives every developer their own server-side repository. The Forking Workflow is most often seen in public open source projects.

Q6: Could you explain the Gitflow workflow?

Master - is always ready to be released on LIVE, with everything fully tested and approved (production-ready).
Hotfix - Maintenance or “hotfix” branches are used to quickly patch production releases. Hotfix branches are a lot like release branches and feature branches except they're based on master instead of develop.
Develop - is the branch to which all feature branches are merged and where all tests are performed. Only when everything’s been thoroughly checked and fixed it can be merged to the master.
Feature - Each new feature should reside in its own branch, which can be pushed to the develop branch as their parent one.

git init: Initializes a new Git repository in the current directory.

git clone <url>: Clones an existing Git repository from a URL.

git add <file>: Adds a file to the staging area.

git commit -m "<message>": Commits changes to the repository with a message.

git status: Shows the current state of the working directory and staging area.

git diff: Shows the differences between the working directory and the staging area, or between the staging area and the last commit.

git log: Shows the commit history of the repository.

2. Branching and Merging

git branch: Lists all local branches.

git branch <branch_name>: Creates a new branch.

git checkout <branch_name>: Switches to a different branch.

git checkout -b <branch_name>: Creates a new branch and switches to it.

git merge <branch_name>: Merges the specified branch into the current branch.

3. Remote Repositories

git remote add origin <url>: Adds a remote repository named "origin".

git push origin <branch_name>: Pushes the current branch to the remote repository.

git pull origin <branch_name>: Pulls changes from the remote repository.

4. Other Useful Commands

git reset --hard HEAD^: Reverts the last commit.

git stash: Temporarily saves changes to the working directory.

git stash pop: Restores the most recently stashed changes.

git tag <tag_name>: Creates a tag for a specific commit.

Tips for Using Git

Write clear commit messages. This helps you and others understand the changes you've made.

Use branches for new features or bug fixes. This allows you to work on multiple things at once without affecting the main codebase.
Commit often. This makes it easier to revert changes if necessary.
Use a version control system like GitHub or GitLab to collaborate with others.