A version control system is a piece of software that helps the developers on a software team work together and also archives a complete history of their work.

There are three basic goals of a version control system (VCS):

1. We want people to be able to work simultaneously, not serially.
2. When people are working at the same time, we want their changes to not conflict with each other.
3. We want to archive every version of everything that has ever existed — ever.

A filesystem is two-dimensional: Its space is defined by directories and files. In contrast, a repository is threedimensional: It exists in a continuum defined by directories, files, and time. A version control repository contains every version of your source code that has ever existed.

A consequence of this idea is that nothing is ever really destroyed. Every time you make some kind of change to your repository, even if that change is to delete something, the repository gets larger because the history is longer.

Each change adds to the history of the repository. We never subtract anything from that history.

The essential difference between a Centralized Version Control System (CVCS) and a DVCS is the notion of a *repository instance*.

In a CVCS, the repository exists in one place on a central server. Every piece of software that is used to access the repository includes a network client.

Most operations interact with a local repository instance, not a network server. The only time networking code gets involved is when the repository instances are being synchronized. Every developer has his own private repository instance.

In practice, virtually all DVCS teams have a central server. With a CVCS, a central server happens because it is inherent in the centralized model. With a DVCS, a central server happens because of the team’s decision to have one.

**Git Cheat Sheet**

git diff Show unstaged changes between your index and working directory.

**git config --global –edit** Open the global configuration file in a text editor for manual editing.

git diff HEAD Show difference between working directory and last commit.

git diff –cached Show difference between staged changes and last commit

**git log --grep=”<pattern>”** Search for commits with a commit message that matches <pattern>.

|  |  |
| --- | --- |
| Create | |
| $ git init | Fom existing data |
| $ git clone | From existing repository |

|  |  |  |
| --- | --- | --- |
| Show | | |
| $ git status | | Files changed in working directory |
| diff | $ git diff | Changes made to tracked files |
| $ git diff --staged | Shows file differences staging and the last revision |
| $ git diff C1 C2 | What changed between C1 and C2 |
| log | $ git log | History of changes |
| $ git log –p File | History of changes for file with diffs |
| $ git log –S “Foo” | Show commits that make add or remove a certain string |
| $ git log -- grep=’day of week’ | Search commits that contain a log message: |
| $ git blame File | | Who changed what and when in a file |
| $ git show C1 | | A commit identified by ID |
| $ git show C1:File | | A specific file from a specific commit |
| $ git branch | | All local branches |
|  | |  |
|  | |  |

$ git log

Lists version history for the current branch

$ git log --follow [file]

Lists version history for a file, including renames

$ git diff [first-branch]...[second-branch]

Shows content differences between two branches

$ git show [commit]

Outputs metadata and content changes of the specified commit

Revert



$ git reset [file]

Unstages the file but preserve its contents

$ git reset [commit]

Undoes all commits after [commit], preserving changes locally

$ git reset --hard [commit]

Discards all history and changes back to the specified commit

Undo last commit:

git reset --hard HEAD~1

Squash last n commits into one commit:

git rebase -i HEAD~5

ou

git reset --soft HEAD~5  
git add .  
git commit -m "Update"  
git push -f origin master

Revert the previous commit:

git revert HEAD  
git commit

Revert the changes from previous 3 commits without making commit:

git revert --no-commit HEAD~3..

Checkout a tag:

git checkout tagname

git checkout -b newbranchname tagname

Checkout a branch:

git checkout destination\_branch

Use -m if there is merge conflict:

git checkout -m master // from feature branch to master

Checkout a commit:

git checkout commit\_hash

git checkout -b newbranchname HEAD~4

git checkout -b newbranchname commit\_hash

git checkout commit\_hash file

Checkout a file:

git checkout c5f567 -- Relative/Path/To/File

Get their changes during git rebase:

git checkout --ours foo/bar.java  
git add foo/bar.java

Get their changes during git merge:

git pull -X theirs

git checkout --theirs path/to/the/conflicted\_file.php

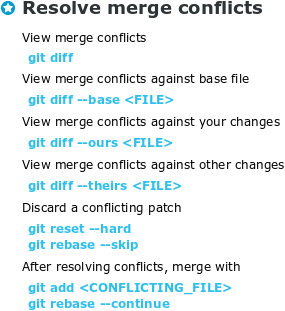
git checkout --theirs .  
git add .

git checkout branchA  
git merge -X theirs branchB

Merge commits from master into feature branch:

git checkout feature1  
git merge --no-ff master

Resolve merge conflicts



$ git rm [file]

Deletes the file from the working directory and stages the deletion

$ git rm --cached [file]

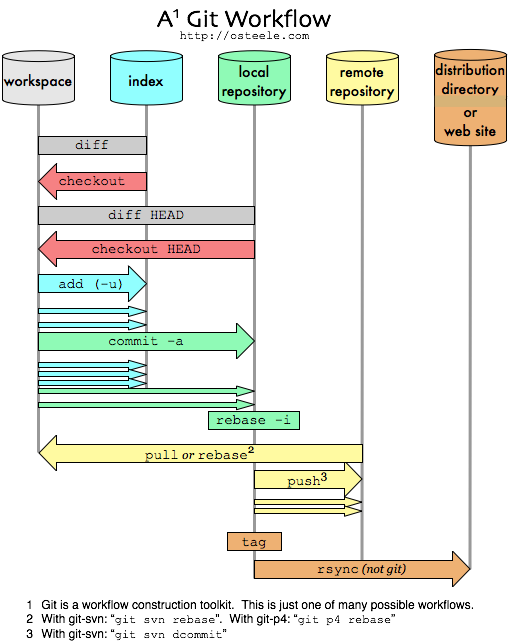
Removes the file from version control but preserves the file locally

$ git mv [file-original] [file-renamed]

Changes the file name and prepares it for commit



“git diff” tells me what I’ve changed since the last checkpoint; “git diff head” shows what’s changed since the last commit. “git checkout .” reverts to the last checkpoint; “git checkout head .” reverts to the last commit. And “git stash” and “git checkout -m -b” operate on the changes since the last commit, which is what I want.



**Setup**

**Show current configuration:**

$ git config --list

**Show repository configuration:**

$ git config --local --list

**Show global configuration:**

$ git config --global --list

**Show system configuration:**

$ git config --system --list

**Set a name that is identifiable for credit when review version history:**

$ git config --global user.name “[firstname lastname]”

**Set an email address that will be associated with each history marker:**

$ git config --global user.email “[valid-email]”

##### Set global editor for commit

$ git config --global core.editor vi

## Configuration Files

##### Repository specific configuration file [–local]:

<repo>/.git/config

##### User-specific configuration file [–global]:

~/.gitconfig

##### System-wide configuration file [–system]:

/etc/gitconfig

## Create

##### Clone an existing repository:

There are two ways:

Via HTTP

$ git clone http://domain.com/user/repo.git

##### Create a new local repository in the current directory:

$ git init

##### Create a new local repository in a specific directory:

$ git init <directory>

## Local Changes

##### Changes in working directory:

$ git status

##### Changes to tracked files:

$ git diff

##### See changes/difference of a specific file:

$ git diff <file>

##### Add all current changes to the next commit:

$ git add .

##### Add some changes in <file> to the next commit:

$ git add -p <file>

##### Commit all local changes in tracked files:

$ git commit -a

##### Commit previously staged changes:

$ git commit

##### Commit with message:

$ git commit -m 'message here'

##### Commit skipping the staging area and adding message:

$ git commit -am 'message here'

##### Change last commit:

Don’t amend published commits!

$ git commit -a --amend

##### Amend with last commit but use the previous commit log message

Don’t amend published commits!

$ git commit --amend --no-edit

##### Move uncommitted changes from current branch to some other branch:

$ git stash

$ git checkout branch2

$ git stash pop

##### Restore stashed changes back to current branch:

$ git stash apply

#### Restore particular stash back to current branch:

* {stash\_number} can be obtained from git stash list

$ git stash apply stash@{stash\_number}

##### Remove the last set of stashed changes:

$ git stash drop

## Search

##### A text search on all files in the directory:

$ git grep "Hello"

##### In any version of a text search:

$ git grep "Hello" v2.5

## Commit History

##### Show all commits, starting with newest (it’ll show the hash, author information, date of commit and title of the commit):

$ git log

##### Show all the commits(it’ll show just the commit hash and the commit message):

$ git log --oneline

##### Show all commits of a specific user:

$ git log --author="username"

##### Show changes over time for a specific file:

$ git log -p <file>

##### Display commits that are present only in remote/branch in right side

$ git log --oneline <origin/master>..<remote/master> --left-right

##### Who changed, what and when in <file>:

$ git blame <file>

##### Show Reference log:

$ git reflog show

##### Delete Reference log:

$ git reflog delete

## Branches & Tags

##### List all local branches:

$ git branch

#### List local/remote branches

$ git branch -a

##### List all remote branches:

$ git branch -r

##### Switch HEAD branch:

$ git checkout <branch>

##### Checkout single file from different branch

$ git checkout <branch> -- <filename>

##### Create and switch new branch:

$ git checkout -b <branch>

##### Create a new branch from an exiting branch and switch to new branch:

$ git checkout -b <new\_branch> <existing\_branch>

#### Checkout and create a new branch from existing commit

$ git checkout <commit-hash> -b <new\_branch\_name>

##### Create a new branch based on your current HEAD:

$ git branch <new-branch>

##### Create a new tracking branch based on a remote branch:

$ git branch --track <new-branch> <remote-branch>

##### Delete a local branch:

$ git branch -d <branch>

##### Rename current branch to new branch name

$ git branch -m <new\_branch\_name>

##### Force delete a local branch:

You will lose unmerged changes!

$ git branch -D <branch>

##### Mark the current commit with a tag:

$ git tag <tag-name>

##### Mark the current commit with a tag that includes a message:

$ git tag -a <tag-name>

## Update & Publish

##### List all current configured remotes:

$ git remote -v

##### Show information about a remote:

$ git remote show <remote>

##### Add new remote repository, named <remote>:

$ git remote add <remote> <url>

##### Download all changes from <remote>, but don’t integrate into HEAD:

$ git fetch <remote>

##### Download changes and directly merge/integrate into HEAD:

$ git remote pull <remote> <url>

##### Get all changes from HEAD to local repository:

$ git pull origin master

##### Get all changes from HEAD to local repository without a merge:

$ git pull --rebase <remote> <branch>

##### Publish local changes on a remote:

$ git push remote <remote> <branch>

##### Delete a branch on the remote:

$ git push <remote> :<branch> (since Git v1.5.0)

OR

$ git push <remote> --delete <branch> (since Git v1.7.0)

##### Publish your tags:

$ git push --tags

## Undo

##### Discard all local changes in your working directory:

$ git reset --hard HEAD

##### Get all the files out of the staging area(i.e. undo the last git add):

$ git reset HEAD

##### Discard local changes in a specific file:

$ git checkout HEAD <file>

##### Revert a commit (by producing a new commit with contrary changes):

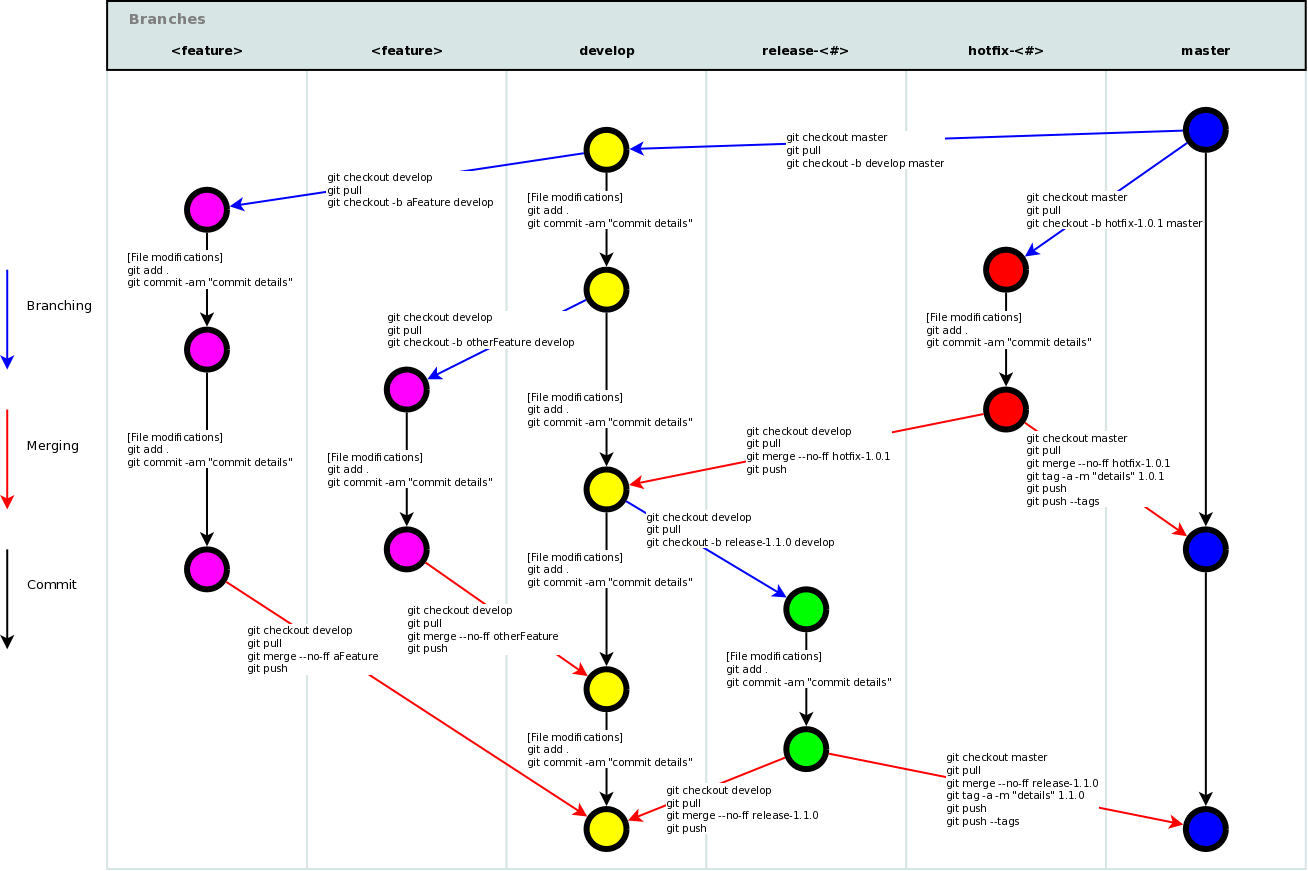
$ git revert <commit>

##### Reset your HEAD pointer to a previous commit and discard all changes since then:

$ git reset --hard <commit>

##### Reset your HEAD pointer to a previous commit and preserve uncommitted local changes:

$ git reset --keep <commit>



# Useful git commands for everyday use!

|  |  |
| --- | --- |
| $ git diff | Show changes between HEAD and working directory: |
| git log -S 'LoginViewController' | Show commits that make add or remove a certain string: |
| git log — all — grep=’day of week’ | Search commits that contain a log message: |
| git tag | List all tags: |
| git tag -a 1.4 -m "my version 1.4" | Tag a commit: |
| git push --delete origin tagname | Delete remote tags: |
| git push origin tagname | Push tag to remote: |
| git tag new old git tag -d old git push origin :refs/tags/old git push --tags | Rename tag: |
| git push origin :refs/tags/<tagname> git tag -fa tagname git push origin master --tags | Move tag from one commit to another commit: |
| git remote prune origin | Remove stale remote tracking branches: |
| git branch -m old new | Rename other branch: |
| git branch -m new | Rename current branch: |
| git branch -m old new # Rename branch locally  git push origin :old # Delete the old branch  git push --set-upstream origin new # Push the new branch, set local branch to track the new remote | Rename remote branch |
| git branch -D the\_local\_branch  git push origin :the\_remote\_branch | Delete a branch: |
| git reset --hard HEAD~1 | Undo last commit |
| git reset --soft HEAD~5 git add . git commit -m "Update" git push -f origin master | Squash last n commits into one commit: |
| git revert HEAD | Revert the previous commit: |
| git revert --no-commit HEAD~3.. | Revert the changes from previous 3 commits without making commit: |
| git commit --amend  git commit --amend --no-edit  git commit --amend -m "New commit message" | Amend previous commit: |
| git checkout destination\_branch | Checkout a branch: |
| git checkout -m master // from feature branch to master | Use -m if there is merge conflict: |
| git checkout commit\_hash  git checkout -b newbranchname HEAD~4  git checkout -b newbranchname commit\_hash | Checkout a commit: |
| git checkout c5f567 -- Relative/Path/To/File | Checkout a file: |
| git reset file | Remove file from index: |
| git reset | Reset the index to match the most recent commit: |
| git reset --hard | Reset the index and the working directory to match the most recent commit: |
| git checkout --ours foo/bar.java git add foo/bar.java | Get their changes during git rebase: |
| git pull -X theirs  git checkout --theirs path/to/conflictedfile.php  git checkout --theirs . git add .  git checkout branchA git merge -X theirs branchB | Get their changes during git merge: |
|  |  |
|  |  |
|  |  |

# Git Tutorial – Git Fu With The Command Line

There are many different GUIs for Git these days: [GitHub for Mac](https://mac.github.com), [Tower](http://www.git-tower.com), [SourceTree](http://www.sourcetreeapp.com), Xcode and more. While these options are fine for most situations, many experienced developers would tell you that the [Command Line Interface](http://en.wikipedia.org/wiki/Command-line_interface) (CLI) is the best option and should be the primary tool you use.

While “best” is an imprecise, subjective term, the Git CLI is certainly the most powerful and versatile way to use Git, as the GUI options usually just run the CLI commands under the hood. Anything a Git GUI can do, the command line can do — probably better.

This Git tutorial introduces you to a few ways to use the command line more effectively with Git.

In particular, you’ll learn how to make the command line a friendlier place by adding autocompletion, and how to make the command line prompt provide you with help and Git guidance.

Maybe you’d like a list all of the accepted pull-requests since June? Just enter the following:

git log --grep="pull request" --since=21june

# Flight rules for Git

### I set the wrong remote repository

If you set the wrong repository as the origin of an existing local repository, change the url of your origin by running:

$ git remote set-url origin [url of the actual repo]

You can show the latest commit on your current HEAD with:

(master)$ git show

f you want to see a file at a specific commit, you can also do this (where <commitid> is the commit you're interested in):

$ git show <commitid>:filename

### I wrote the wrong thing in a commit message

If you wrote the wrong thing and the commit has not yet been pushed, you can do the following to change the commit message without changing the changes in the commit:

$ git commit --amend --only

This will open your default text editor, where you can edit the message. On the other hand, you can do this all in one command:

$ git commit --amend --only -m 'xxxxxxx'

### I committed with the wrong name and email configured

If it's a single commit, amend it

$ git commit --amend --no-edit --author "New Authorname <authoremail@mydomain.com>"

An alternative is to correctly configure your author settings in git config --global author.(name|email) and then use

$ git commit --amend --reset-author --no-edit

### I want to remove a file from the previous commit

In order to remove changes for a file from the previous commit, do the following:

$ git checkout HEAD^ myfile

$ git add myfile

$ git commit --amend --no-edit

In case the file was newly added to the commit and you want to remove it (from Git alone), do:

$ git rm --cached myfile

$ git commit --amend --no-edit

The --no-edit option is used to keep the existing commit message.

### I want to delete or remove my last commit

If you need to delete pushed commits, you can use the following. However, it will irreversibly change your history, and mess up the history of anyone else who had already pulled from the repository. In short, if you're not sure, you should never do this, ever.

$ git reset HEAD^ --hard

$ git push --force-with-lease [remote] [branch]

If you haven't pushed, to reset Git to the state it was in before you made your last commit (while keeping your staged changes):

(my-branch\*)$ git reset --soft HEAD@{1}

This only works if you haven't pushed. If you have pushed, the only truly safe thing to do is git revert SHAofBadCommit. That will create a new commit that undoes all the previous commit's changes. Or, if the branch you pushed to is rebase-safe (ie. other devs aren't expected to pull from it), you can just use git push --force-with-lease.

### Delete/remove arbitrary commit

The same warning applies as above. Never do this if possible.

$ git rebase --onto SHA1\_OF\_BAD\_COMMIT^ SHA1\_OF\_BAD\_COMMIT

$ git push --force-with-lease [remote] [branch]

Or do an [interactive rebase](https://github.com/k88hudson/git-flight-rules/blob/master/README.md#interactive-rebase) and remove the line(s) corresponding to commit(s) you want to see removed.

Git documentation

A range of commits could also be removed with rebase. If we have the following situation:

E---F---G---H---I---J topicA

then the command

git rebase --onto topicA~5 topicA~3 topicA

would result in the removal of commits F and G:

E---H'---I'---J' topicA

### I accidentally did a hard reset, and I want my changes back

If you accidentally do git reset --hard, you can normally still get your commit back, as git keeps a log of everything for a few days.

Note: This is only valid if your work is backed up, i.e., either committed or stashed. git reset --hard will remove uncommitted modifications, so use it with caution. (A safer option is git reset --keep.)

(master)$ git reflog

You'll see a list of your past commits, and a commit for the reset. Choose the SHA of the commit you want to return to, and reset again:

(master)$ git reset --hard SHA1234

### I accidentally committed and pushed a merge

If you accidentally merged a feature branch to the main development branch before it was ready to be merged, you can still undo the merge. But there's a catch: A merge commit has more than one parent (usually two).

The command to use

(feature-branch)$ git revert -m 1 <commit>

where the -m 1 option says to select parent number 1 (the branch into which the merge was made) as the parent to revert to.

Note: the parent number is not a commit identifier. Rather, a merge commit has a line Merge: 8e2ce2d 86ac2e7. The parent number is the 1-based index of the desired parent on this line, the first identifier is number 1, the second is number 2, and so on.

### I accidentally committed and pushed files containing sensitive data

Amending the pushed commit is not enough, since anyone could have pulled the original commit containing your sensitive data in the meantime

If you edit the file and remove the sensitive data, then run

(feature-branch)$ git add edited\_file

(feature-branch)$ git commit --amend --no-edit

(feature-branch)$ git push --force-with-lease origin [branch]

If you want to remove an entire file (but keep it locally), then run

(feature-branch)$ git rm --cached sensitive\_file

echo sensitive\_file >> .gitignore

(feature-branch)$ git add .gitignore

(feature-branch)$ git commit --amend --no-edit

(feature-branch)$ git push --force-with-lease origin [branch]

f you want to completely remove an entire file (and not keep it locally), then run

(feature-branch)$ git rm sensitive\_file

(feature-branch)$ git commit --amend --no-edit

(feature-branch)$ git push --force-with-lease origin [branch]

If you have made other commits in the meantime (i.e. the sensitive data is in a commit before the previous commit), you will have to rebase.

### I need to change the content of a commit which is not my last

Consider a situation where you want to change the third last commit you made.

(your-branch)$ git rebase -i HEAD~4

gets you into interactive rebase mode, which allows you to edit any of your last three commits. A text editor pops up

This tells rebase that you want to edit your third last commit and keep the other two unaltered. Then you'll save (and close) the editor. Git will then start to rebase. It stops on the commit you want to alter, giving you the chance to edit that commit.

Now you can apply the changes which you missed applying when you initially commited that commit. You do so by editing and staging them. Afterwards you'll run

(your-branch)$ git commit --amend

which tells Git to recreate the commit, but to leave the commit message unedited. Having done that, the hard part is solved.

(your-branch)$ git rebase --continue

will do the rest of the work for you.

## Staging

### I need to add staged changes to the previous commit

(my-branch\*)$ git commit --amend

If you already know you don't want to change the commit message, you can tell git to reuse the commit message:

(my-branch\*)$ git commit --amend -C HEAD

### I want to stage part of a new file, but not the whole file

Normally, if you want to stage part of a file, you run this:

$ git add --patch filename.x

Then, you will need to use the e option to manually choose which lines to add. Running git diff --cached or git diff --staged will show you which lines you have staged compared to which are still saved locally.

### I want to add changes in one file to two different commits

git add will add the entire file to a commit. git add -p will allow to interactively select which changes you want to add.

### I staged too many edits, and I want to break them out into a separate commit

git reset -p will open a patch mode reset dialog. This is similar to git add -p, except that selecting "yes" will unstage the change, removing it from the upcoming commit.

### I want to stage my unstaged edits, and unstage my staged edits

This is tricky. The best I figure is that you should stash your unstaged edits. Then, reset. After that, pop your stashed edits back, and add them.

$ git stash -k

$ git reset --hard

$ git stash pop

$ git add -A

## Unstaged Edits

### I want to move my unstaged edits to a new branch

$ git checkout -b my-branch

### I want to move my unstaged edits to a different, existing branch

$ git stash

$ git checkout my-branch

$ git stash pop

### I want to discard my local uncommitted changes (staged and unstaged)

If you want to discard all your local staged and unstaged changes, you can do this:

(my-branch)$ git reset --hard

# or

(master)$ git checkout -f

This will unstage all files you might have staged with git add:

$ git reset

This will revert all local uncommitted changes (should be executed in repo root):

$ git checkout .

You can also revert uncommitted changes to a particular file or directory:

$ git checkout [some\_dir|file.txt]

Yet another way to revert all uncommitted changes (longer to type, but works from any subdirectory):

$ git reset --hard HEAD

This will remove all local untracked files, so only files tracked by Git remain:

$ git clean -fd

-x will also remove all ignored files.

### I want to discard specific unstaged changes

When you want to get rid of some, but not all changes in your working copy.

Checkout undesired changes, keep good changes.

$ git checkout -p

# Answer y to all of the snippets you want to drop

Another strategy involves using stash. Stash all the good changes, reset working copy, and reapply good changes.

$ git stash -p

# Select all of the snippets you want to save

$ git reset --hard

$ git stash pop

Alternatively, stash your undesired changes, and then drop stash.

$ git stash -p

# Select all of the snippets you don't want to save

$ git stash drop

### I want to discard specific unstaged files

When you want to get rid of one specific file in your working copy.

$ git checkout myFile

Alternatively, to discard multiple files in your working copy, list them all.

$ git checkout myFirstFile mySecondFile

### I want to discard only my unstaged local changes

When you want to get rid of all of your unstaged local uncommitted changes

$ git checkout .

### I want to discard all of my untracked files

When you want to get rid of all of your untracked files

$ git clean -f

### I want to unstage a specific staged file

Sometimes we have one or more files that accidentally ended up being staged, and these files have not been committed before. To unstage them:

$ git reset -- <filename>

This results in unstaging the file and make it look like it's untracked.

## Branches

### Create a branch from a commit

$ git checkout -b <branch> <SHA1\_OF\_COMMIT>

### I want to discard local commits so my branch is the same as one on the server

Confirm that you haven't pushed your changes to the server.

git status should show how many commits you are ahead of origin:

(my-branch)$ git status

# On branch my-branch

# Your branch is ahead of 'origin/my-branch' by 2 commits.

# (use "git push" to publish your local commits)

#

One way of resetting to match origin (to have the same as what is on the remote) is to do this:

(master)$ git reset --hard origin/my-branch

### I committed to master instead of a new branch

Create the new branch while remaining on master:

(master)$ git branch my-branch

Reset the branch master to the previous commit:

(master)$ git reset --hard HEAD^

Checkout the new branch to continue working:

(master)$ git checkout my-branch

### I want to keep the whole file from another ref-ish

You can solve it bringing the contents to your branch:

(develop)$ git checkout solution -- file1.txt

### I want to delete local branches that were deleted upstream

$ git fetch -p upstream

where, upstream is the remote you want to fetch from.

### I want to delete a branch

To delete a remote branch:

(master)$ git push origin --delete my-branch

You can also do:

(master)$ git push origin :my-branch

To delete a local branch:

(master)$ git branch -d my-branch

To delete a local branch that has not been merged to the current branch or an upstream:

(master)$ git branch -D my-branch

### I want to delete multiple branches

Say you want to delete all branches that start with fix/:

(master)$ git branch | grep 'fix/' | xargs git branch -d

### I want to rename a branch

To rename the current (local) branch:

(master)$ git branch -m new-name

To rename a different (local) branch:

(master)$ git branch -m old-name new-name

### I want to checkout to a remote branch that someone else is working on

First, fetch all branches from remote:

(master)$ git fetch --all

Say you want to checkout to daves from the remote.

(master)$ git checkout --track origin/daves

Branch daves set up to track remote branch daves from origin.

Switched to a new branch 'daves'

(--track is shorthand for git checkout -b [branch] [remotename]/[branch])

This will give you a local copy of the branch daves, and any update that has been pushed will also show up remotely.

### I want to create a new remote branch from current local one

$ git push <remote> HEAD

If you would also like to set that remote branch as upstream for the current one, use the following instead:

$ git push -u <remote> HEAD

With the upstream mode and the simple (default in Git 2.0) mode of the push.default config, the following command will push the current branch with regards to the remote branch that has been registered previously with -u:

$ git push

### I want to set a remote branch as the upstream for a local branch

You can set a remote branch as the upstream for the current local branch using:

$ git branch --set-upstream-to [remotename]/[branch]

# or, using the shorthand:

$ git branch -u [remotename]/[branch]

To set the upstream remote branch for another local branch:

$ git branch -u [remotename]/[branch] [local-branch]

## Rebasing and Merging

### I want to undo rebase/merge

You may have merged or rebased your current branch with a wrong branch, or you can't figure it out or finish the rebase/merge process. Git saves the original HEAD pointer in a variable called ORIG\_HEAD before doing dangerous operations, so it is simple to recover your branch at the state before the rebase/merge.

(my-branch)$ git reset --hard ORIG\_HEAD

#### Safe merging strategy

--no-commit performs the merge but pretends the merge failed and does not autocommit, giving the user a chance to inspect and further tweak the merge result before committing. no-ff maintains evidence that a feature branch once existed, keeping project history consistent.

(master)$ git merge --no-ff --no-commit my-branch

#### I need to merge a branch into a single commit

(master)$ git merge --squash my-branch

## Stash

### Stash all edits

To stash all the edits in your working directory

$ git stash

If you also want to stash untracked files, use -u option.

$ git stash -u

### Stash with message

$ git stash save <message>

### Apply a specific stash from list

First check your list of stashes with message using

$ git stash list

Then apply a specific stash from the list using

$ git stash apply "stash@{n}"

Here, 'n' indicates the position of the stash in the stack. The topmost stash will be position 0.

## Finding

### I want to find a string in any commit

To find a certain string which was introduced in any commit, you can use the following structure:

$ git log -S "string to find"

Commons parameters:

* --source means to show the ref name given on the command line by which each commit was reached.
* --all means to start from every branch.
* --reverse prints in reverse order, it means that will show the first commit that made the change.

### I want to find by author/committer

To find all commits by author/committer you can use:

$ git log --author=<name or email>

$ git log --committer=<name or email>

Keep in mind that author and committer are not the same. The --author is the person who originally wrote the code; on the other hand, the --committer, is the person who committed the code on behalf of the original author.

### I want to list commits containing specific files

To find all commits containing a specific file you can use:

$ git log -- <path to file>

You would usually specify an exact path, but you may also use wild cards in the path and file name:

$ git log -- \*\*/\*.js

While using wildcards, it's useful to inform --name-status to see the list of committed files:

$ git log --name-status -- \*\*/\*.js

### I want to view the commit history for a specific function

To trace the evolution of a single function you can use:

$ git log -L :FunctionName:FilePath

Note that you can combine this with further git log options, like [revision ranges](https://git-scm.com/docs/gitrevisions) and [commit limits](https://git-scm.com/docs/git-log/#_commit_limiting).

### Find a tag where a commit is referenced

To find all tags containing a specific commit:

$ git tag --contains <commitid>

## Submodules

### Clone all submodules

$ git clone --recursive git://github.com/foo/bar.git

If already cloned:

$ git submodule update --init --recursive

### Remove a submodule

Creating a submodule is pretty straight-forward, but deleting them less so. The commands you need are:

$ git submodule deinit submodulename

$ git rm submodulename

$ git rm --cached submodulename

$ rm -rf .git/modules/submodulename

## Miscellaneous Objects

### Restore a deleted file

First find the commit when the file last existed:

$ git rev-list -n 1 HEAD -- filename

Then checkout that file:

git checkout deletingcommitid^ -- filename

### Delete tag

$ git tag -d <tag\_name>

$ git push <remote> :refs/tags/<tag\_name>

### Push a branch and a tag that have the same name

If there is a tag on a remote repository that has the same name as a branch you will get the following error when trying to push that branch with a standard $ git push <remote> <branch> command.

$ git push origin <branch>

error: dst refspec same matches more than one.

error: failed to push some refs to '<git server>'

Fix this by specifying you want to push the head reference.

$ git push origin refs/heads/<branch-name>

If you want to push a tag to a remote repository that has the same name as a branch, you can use a similar command.

$ git push origin refs/tags/<tag-name>

## Tracking Files

### I want to change a file name's capitalization, without changing the contents of the file

(master)$ git mv --force myfile MyFile

### I want to overwrite local files when doing a git pull

(master)$ git fetch --all

(master)$ git reset --hard origin/master

### I want to remove a file from Git but keep the file

(master)$ git rm --cached log.txt

### I want to revert a file to a specific revision

Assuming the hash of the commit you want is c5f567:

(master)$ git checkout c5f567 -- file1/to/restore file2/to/restore

If you want to revert to changes made just 1 commit before c5f567, pass the commit hash as c5f567~1:

(master)$ git checkout c5f567~1 -- file1/to/restore file2/to/restore

### I want to list changes of a specific file between commits or branches

Assuming you want to compare last commit with file from commit c5f567:

$ git diff HEAD:path\_to\_file/file c5f567:path\_to\_file/file

Same goes for branches:

$ git diff master:path\_to\_file/file staging:path\_to\_file/file

### I want Git to ignore changes to a specific file

This works great for config templates or other files that require locally adding credentials that shouldn't be committed.

$ git update-index --assume-unchanged file-to-ignore

Note that this does not remove the file from source control - it is only ignored locally. To undo this and tell Git to notice changes again, this clears the ignore flag:

$ git update-index --no-assume-unchanged file-to-stop-ignoring

### I want to add an empty directory to my repository

You can’t! Git doesn’t support this, but there’s a hack.

Another common convention is to make an empty file in the folder, titled .gitkeep.

$ mkdir mydir

$ touch mydir/.gitkeep

You can also name the file as just .keep , in which case the second line above would be touch mydir/.keep

## Configuration

### I want to set a global user

To configure user information used across all local repositories, and to set a name that is identifiable for credit when review version history:

$ git config --global user.name “[firstname lastname]”

To set an email address that will be associated with each history marker:

git config --global user.email “[valid-email]”

### Remove All Deleted Files from the Working Tree

When you delete a lot of files using /bin/rm you can use the following command to remove them from the working tree and from the index, eliminating the need to remove each one individually:

$ git rm $(git ls-files -d)

### Previous Branch

To move to the previous branch in Git:

$ git checkout -

# Switched to branch 'master'

$ git checkout -

# Switched to branch 'next'

$ git checkout -

# Switched to branch 'master

### Empty Commits

Commits can be pushed with no code changes by adding --allow-empty:

$ git commit -m "Big-ass commit" --allow-empty

### Styled Git Status

$ git status –sb

### Git Query

A Git query allows you to search all your previous commit messages and find the most recent one matching the query.

$ git show :/query

where query (case-sensitive) is the term you want to search, this then finds the last one and gives details on the lines that were changed.

$ git show :/typo

### Git Grep

Git Grep will return a list of lines matching a pattern.

Running:

$ git grep aliases

will show all the files containing the string aliases.

You can also use multiple flags for more advanced search. For example:

* -e The next parameter is the pattern (e.g., regex)
* --and, --or and --not Combine multiple patterns.

Use it like this:

$ git grep -e pattern --and -e anotherpattern

### Merged Branches

Running:

$ git branch --merged

will give you a list of all branches that have been merged into your current branch.

Conversely:

$ git branch --no-merged

will give you a list of branches that have not been merged into your current branch.

## Search change by content

git log -S'<a term in the source>'

## Sync with remote, overwrite local changes

git fetch origin && git reset --hard origin/master && git clean -f -d

## Unstaged changes since last commit

git diff

## Changes staged for commit

git diff --cached

**Alternatives:**

git diff --staged

## Show both staged and unstaged changes

git diff HEAD

## List all branches that are already merged into master

git branch --merged master

## Quickly switch to the previous branch

git checkout -

**Alternatives:**

git checkout @{-1}

## List all branches and their upstreams, as well as last commit on branch

git branch -vv

## Track upstream branch

git branch -u origin/mybranch

## Delete local branch

git branch -d <local\_branchname>

## Delete remote branch

git push origin --delete <remote\_branchname>

**Alternatives:**

git push origin :<remote\_branchname>

## Delete local tag

git tag -d <tag-name>

## Delete remote tag

git push origin :refs/tags/<tag-name>

## Undo local changes with the last content in head

git checkout -- <file\_name>

## Revert: Undo a commit by creating a new commit

git revert <commit-ish>

## Reset: Discard commits, advised for private branch

git reset <commit-ish>

## Reword the previous commit message

git commit -v --amend

## See commit history for just the current branch

git cherry -v master

## Amend author.

git commit --amend --author='Author Name <email@address.com>'

## Changing a remote's URL

git remote set-url origin <URL>

## Get list of all remote references

git remote

**Alternatives:**

git remote show

## Get list of all local and remote branches

git branch -a

## Get only remote branches

git branch -r

## What changed since two weeks?

git log --no-merges --raw --since='2 weeks ago'

## See all commits made since forking from master

git log --no-merges --stat --reverse master..

## Saving current state of tracked files without commiting

git stash

**Alternatives:**

git stash save

## Saving current state of unstaged changes to tracked files

git stash -k

**Alternatives:**

git stash --keep-index

git stash save --keep-index

## Saving current state including untracked files

git stash -u

**Alternatives:**

git stash save -u

git stash save --include-untracked

## Saving current state with message

git stash save <message>

## Saving current state of all files (ignored, untracked, and tracked)

git stash -a

**Alternatives:**

git stash --all

git stash save --all

## Show list of all saved stashes

git stash list

## Apply any stash without deleting from the stashed list

git stash apply <stash@{n}>

## Apply last stashed state and delete it from stashed list

git stash pop

**Alternatives:**

git stash apply stash@{0} && git stash drop stash@{0}

## Delete all stored stashes

git stash clear

**Alternatives:**

git stash drop <stash@{n}>

## Grab a single file from a stash

git checkout <stash@{n}> -- <file\_path>

**Alternatives:**

git checkout stash@{0} -- <file\_path>

## Show all tracked files

git ls-files -t

## Show all untracked files

git ls-files --others

## Show all ignored files

git ls-files --others -i --exclude-standard

## Untrack files without deleting

git rm --cached <file\_path>

**Alternatives:**

git rm --cached -r <directory\_path>

## Before deleting untracked files/directory, do a dry run to get the list of these files/directories

git clean -n

## Forcefully remove untracked files

git clean -f

## Forcefully remove untracked directory

git clean -f -d

## Update all the submodules

git submodule foreach git pull

**Alternatives:**

git submodule update --init --recursive

git submodule update --remote

## Show all commits in the current branch yet to be merged to master

git cherry -v master

**Alternatives:**

git cherry -v master <branch-to-be-merged>

## Rename a branch

git branch -m <new-branch-name>

**Alternatives:**

git branch -m [<old-branch-name>] <new-branch-name>

## Archive the master branch

git archive master --format=zip --output=master.zip

## Modify previous commit without modifying the commit message

git add --all && git commit --amend --no-edit

## Prunes references to remote branches that have been deleted in the remote.

git fetch -p

**Alternatives:**

git remote prune origin

## Retrieve the commit hash of the initial revision.

git rev-list --reverse HEAD | head -1

**Alternatives:**

git rev-list --max-parents=0 HEAD

git log --pretty=oneline | tail -1 | cut -c 1-40

git log --pretty=oneline --reverse | head -1 | cut -c 1-40

## Deploying git tracked subfolder to gh-pages

git subtree push --prefix subfolder\_name origin gh-pages

## Adding a project to repo using subtree

git subtree add --prefix=<directory\_name>/<project\_name> --squash git@github.com:<username>/<project\_name>.git master

## Get latest changes in your repo for a linked project using subtree

git subtree pull --prefix=<directory\_name>/<project\_name> --squash git@github.com:<username>/<project\_name>.git master

## Show the most recent tag on the current branch.

git describe --tags --abbrev=0

## Show inline word diff.

git diff --word-diff

## Show changes using common diff tools.

git difftool [-t <tool>] <commit1> <commit2> <path>

## Don’t consider changes for tracked file.

git update-index --assume-unchanged <file\_name>

## Undo assume-unchanged.

git update-index --no-assume-unchanged <file\_name>

## Clean the files from .gitignore.

git clean -X -f

## Restore deleted file.

git checkout <deleting\_commit>^ -- <file\_path>

## Restore file to a specific commit-hash

git checkout <commit-ish> -- <file\_path>

## Always rebase instead of merge on pull.

git config --global pull.rebase true

## Dry run. (any command that supports dry-run flag should do.)

git clean -fd --dry-run

## Marks your commit as a fix of a previous commit.

git commit --fixup <SHA-1>

## Squash fixup commits normal commits.

git rebase -i --autosquash

## Skip staging area during commit.

git commit --only <file\_path>

## Interactive staging.

git add -i

## List ignored files.

git check-ignore \*

## Commits in Branch1 that are not in Branch2

git log Branch1 ^Branch2

## List n last commits

git log -<n>

**Alternatives:**

git log -n <n>

## Open all conflicted files in an editor.

git diff --name-only | uniq | xargs $EDITOR

## Count unpacked number of objects and their disk consumption.

git count-objects --human-readable

## Prune all unreachable objects from the object database.

git gc --prune=now --aggressive

## Checkout a new branch without any history

git checkout --orphan <branch\_name>

## Extract file from another branch.

git show <branch\_name>:<file\_name>

## List only the root and merge commits.

git log --first-parent

## Change previous two commits with an interactive rebase.

git rebase --interactive HEAD~2

## List all branch is WIP

git checkout master && git branch --no-merged

## List commits and changes to a specific file (even through renaming)

git log --follow -p -- <file\_path>

## Clone a single branch

git clone -b <branch-name> --single-branch https://github.com/user/repo.git

## Create and switch new branch

git checkout -b <branch-name>

**Alternatives:**

git branch <branch-name> && git checkout <branch-name>

## Show all local branches ordered by recent commits

git for-each-ref --sort=-committerdate --format='%(refname:short)' refs/heads/

## Find lines matching the pattern (regex or string) in tracked files

git grep --heading --line-number 'foo bar'

## Clone a shallow copy of a repository

git clone https://github.com/user/repo.git --depth 1

## Search Commit log across all branches for given text

git log --all --grep='<given-text>'

## Get first commit in a branch (from master)

git log --oneline master..<branch-name> | tail -1

**Alternatives:**

git log --reverse master..<branch-name> | head -6

## Unstaging Staged file

git reset HEAD <file-name>

## Force push to Remote Repository

git push -f <remote-name> <branch-name>

## Adding Remote name

git remote add <remote-nickname> <remote-url>

## Show the author, time and last revision made to each line of a given file

git blame <file-name>

## Group commits by authors and title

git shortlog

## Forced push but still ensure you don't overwrite other's work

git push --force-with-lease <remote-name> <branch-name>

## Revert: Reverting an entire merge

git revert -m 1 <commit-ish>

## Number of commits in a branch

git rev-list --count <branch-name>

## Specific fetch reference

git fetch origin master:refs/remotes/origin/mymaster

## Find common ancestor of two branches

git merge-base <branch-name> <other-branch-name>

## List unpushed git commits

git log --branches --not --remotes

**Alternatives:**

git log @{u}..

git cherry -v

## logs between date range

git log --since='FEB 1 2017' --until='FEB 14 2017'

## Exclude author from logs

git log --perl-regexp --author='^((?!excluded-author-regex).\*)

## List references in a remote repository

git ls-remote git://git.kernel.org/pub/scm/git/git.git

## List all git aliases

git config -l | grep alias | sed 's/^alias\.//g'

**Alternatives:**

git config -l | grep alias | cut -d '.' -f 2

## Show git status short

git status --short --branch

## Checkout a commit prior to a day ago

git checkout master@{yesterday}

## Push a new local branch to remote repository and track

git push -u origin <branch\_name>

## Change a branch base

git rebase --onto <new\_base> <old\_base>

# Little Things I Like to Do with Git

$ git shortlog -sn --all --no-merges

The shortlog gives us summaries of git log; the -s flag will suppress commit description and provide a commit count summary only, and the -n flag will sort output according to the number of commits per author instead of author alphabetic order. The --all flag logs all branches, and --no-merges ensures that merge commits aren’t being counted.

The above shows all commits for the lifetime of the project, but if you want to see how much people have done during a specified timeframe, you can use the --since and --until flags:

$ git shortlog -sn --since='10 weeks' --until='2 weeks'

When you diff or show an object with a lot of whitespace changes, we’re left looking at a lot of visual noise which can make it hard to see anything more important.

Thankfully, removing this noise is pretty trivial by using the -w flag which can be used alongside git diff and git show

When editing prose, as opposed to code, it can often be much more useful to see changed words rather than whole changed lines; this is particularly helpful when writing markdown, like I am right now.

Thankfully, we can show only changes words by using the --word-diff flag:

$ git diff --word-diff

It’s not uncommon for me to jump between lots of different branches on any given project, and keeping track of them can be pretty tricky. We can get Git to help us work this out:

$ git for-each-ref --count=10 --sort=-committerdate refs/heads/ --format="%(refname:short)"

This will show us the last 10 (--count=10) branches that we worked on, sorted by the time that we were last working there. It only shows us local branches (refs/heads/) and in a much nicer --format.

This is a bit verbose to commit to memory, so I have it aliased to $ git recent.

Writing a CHANGELOG can be a little tedious; we have to look over all of the work we’ve done since our last release and then pull out all of the useful bits. Thankfully we can use Git to give us a head start:

$ git log --oneline --no-merges <last tag>..HEAD

**N.B.** HEAD is optional here—if you omit it (i.e. ... --no-merges <last tag>..) then HEAD is implied. That saves you a couple of keystrokes.

This will create a simplified log showing all commits (excluding merge commits) that took place between your last release and HEAD.

## Check Which Changes You’re About to Pull

If you haven’t worked on a project for a little while, you might want to check what’s happened upstream before you pull all of those changes down into your local branch.

$ git log --oneline --no-merges HEAD..<remote>/<branch>

**N.B.** Again, HEAD is optional here, and omitting it will leave it implied.

For example, let’s see what someone has been doing in a particular feature branch whilst you were on holiday:

$ git checkout feature/fonts

$ git fetch

$ git log --oneline --no-merges ..origin/feature/fonts

## Review What You’re About to Push

Hopefully you commit and push often, but if—for whatever reason—you find yourself with a large amount of local commits that are yet to be pushed, it’s probably wise to quickly review what they all are.

In order to do this, we’re effectively just inverting the previous command:

$ git log --oneline --no-merges <remote>/<branch>..HEAD

For example:

$ git fetch

$ git log --oneline --no-merges origin/feature/fonts..HEAD

**N.B.** Again, HEAD is optional here, and omitting it will leave it implied.

This logs the commits that <remote>/<branch> needs before it resembles HEAD.

# 19 Git Tips For Everyday Use

## 1. Parameters for better logging

Chances are, by now you’ve used git log. It supports a number of command line parameters, which are very powerful, especially when used in combination. Here are the ones that I use the most:

* --author=“Alex Kras" – Only show commits made by a certain author
* --name-only – Only show names of files that changed
* --oneline – Show commit data compressed to one line
* --graph – Show dependency tree for all commits
* --reverse – Show commits in reverse order (Oldest commit first)
* --after – Show all commits that happened after certain date
* --before – Show all commits that happened before certain data

## 2. Log actual changes in a file

**Sample Command** git log -p filename

git log -p or git log -p filename lets you view not only the commit message, author, and date, but actual changes that took place in each commi

## 3. Only Log changes for some specific lines in a file

git blame is a great tool, but sometimes it does not provide enough information.

An alternative is provided by git log with a -L flag. This flag allows you to specify particular lines in a file that you are interested in. Then Git would only log changes relevant to those lines. It’s kind of like git log -p with focus.

git log -L 1,1:some-file.txt

## 4. Log changes not yet merged to the parent branch

**Sample:** git log --no-merges master..

If you ever worked on a long-lived branches, with multiple people working on it, chances are you’ve experienced numerous merges of the parent branch(i.e. master) into your feature branch. This makes it hard to see the changes that took place on the master branch vs. the changes that have been committed on the feature branch and which have yet to be merged.

git log --no-merges master.. will solve the issue. Note the --no-merges flag indicate to only show changes that have not been merged yet to ANY branch, and the **master..** option, indicates to only show changes that have not been merged to master branch. (You must include the **..** after master).

You can also do git show --no-merges master.. or git log -p --no-merges master.. (output is identical) to see actual file changes that are have yet to be merged.

## 5. Extract a file from another branch

**Sample:** git show some-branch:some-file.js

Sometimes it is nice to take a pick at an entire file on a different branch, without switching to this branch.

You can do so via git show some-branch-name:some-file-name.js, which would show the file in your terminal.

You can also redirect the output to a temporary file, so you can perhaps open it up in a side by side view in your editor of choice.

git show some-branch-name:some-file-name.js > deleteme.js

Note: If all you want to see is a diff between two files, you can simple run:  
git diff some-branch some-filename.js

## 7. Remember the branch structure after a local merge

**Sample:** git merge --no-ff

## 8. Fix your previous commit, instead of making a new commit

**Sample** git commit --amend

## 9. Three stages in git, and how to move between them

**Sample** git reset --hard HEAD and git status -s

If you simply want to forget some local changes for some files, but at the same time want to keep changes made in other files, it is much easier to check out committed versions of the files that you want to forget, via:  
git checkout forget-my-changes.js

It’s like running git reset --hard but only on some of the files.

As mentioned before you can also check out a different version of a file from another branch or commit.

git checkout some-branch-name file-name.js and  
git checkout {{some-commit-hash}} file-name.js

You’ll notice that the checked out files will be in a “staged for commit” stage. To move them back to “un-staged for commit” stage, you would have to do a git reset HEAD file-name.js. You can run git checkout file-name.js again, to return the file to it’s original state.

## 10. Revert a commit, softly

**Sample** git revert -n

This one is handy if you want to undo a previous commit or two, look at the changes, and see which ones might have caused a problem.

Regular git revert will automatically re-commit reverted files, prompting you to write a new commit message. The -n flag tells git to take it easy on committing for now, since all we want to do is look.

## 11. See diff-erence for the entire project (not just one file at a time) in a 3rd party diff tool

Simply run the following commands, making sure to replace meld with your favorite diff tools of choice:

git config --global diff.tool meld

git config --global merge.tool meld

$ git config --global difftool.meld.path "/C/Program Files (x86)/Meld/meld/meld.exe"

If you invoke git difftool with a -d flag, it will try to diff the entire folder.

## 12. Ignore the white space

**Sample** git diff -w or git blame -w

Have you ever re-indented or re-formatted a file, only to realize that now git blame shows that you are responsible for everything in that file?

Turns out, git is smart enough to know the difference. You can invoke a lot of the commands (i.e. git diff, git blame) with a -w flag, and git will ignore the white space changes.

## 13. Only “add” some changes from a file

**Sample** git add -p

it allows you to interactive select exactly what you want to be committed. That way you can logically organize your commits in an easy to read manner

## 14. Discover and zap those old branches

**Sample** git branch -a

## 15. Stash only some files

**Sample** git stash —keep-index or git stash -p

One limitation of regular git stash is that it will stash all of the files at once. And sometimes it is handy to only stash some of the file, and keep the rest in your working tree.

Remember the magic -pcommand? Well it’s really handy with git stash as well. As you may have probably guessed by now, it will ask you to see which chunks of changes you want to be stashed.

Another handy trick, for stashing only some of the files, is to:

1. add the files that you DO NOT want to get stashed (i.e. git add file1.js, file2.js)
2. Call git stash --keep-index. It will only stash files that have not been added.
3. Call git reset to un-stage the added files and continue your work.

## 18. Create aliases for your most frequently used commands

**TLDR; Use git or bash aliases for most commonly used long git commands**

Best way to use Git is via command line, and the best way to learn the command line is by doing everything the hard way first (typing everything out).

After a while, however, it might be a good idea to track down your most used commands, and create an easier aliases for them.

# Git Cheat Sheet

Visit [my blog](http://www.alexkras.com/) or [connect with me on Twitter](https://twitter.com/akras14)

## Commands

### Getting Started

git init

or

git clone url

### Configuration

git config --global color.ui true

git config --global push.default current

git config --global core.editor vim

git config --global user.name "John Doe"

git config --global user.email foo@citrix.com

git config --global diff.tool meld

### Working with Local Branch

#### Branching

# See the list of all local branches

git branch

# Switch to existing local branch

git checkout branchname

# Checkout current branch into a new branch, named new-branch-name

git checkout -b new-branch-name

# Merge branch-name into the current branch

git merge branchname

# Merge branch without fast forwarding. This is what pull requests do.

# It helps to preserve history of the changes as relavant to that branch

# It's an advanced feature, but try it out with GUI to see the difference

# between the regular merge and merge --no-ff

git merge --no-ff branchname

# Soft branch delete, will complain if the branch is not merged

git branch -d branchname

# Hard branch delete, will not complain about nothing. Like rm -rf in bash

git branch -D branchname

#### Updating Current Branch

**Standard Flow**

# See all commits

git log

# Pretty commit view, you can customize it as much as you want.

# Just google it :)

git log --pretty=format:"%h %s" --graph

# See what you worked on in the past week

git log --author='Alex' --after={1.week.ago} --pretty=oneline --abbrev-commit

# See only changes made on this branch (assuming it was branched form master branch)

git log --no-merges master..

# See status of your current git branch.

# Often will have advice on command that you need to run

git status

# Short view of status. Helpful for seeing things at a glance

git status -s

# Add modified file to be commited(aka stage the file)

git add filename

# Add all modified files to be commited(aka stage all files)

git add .

# Add only text files, etc.

git add '\*.txt'

# Tell git not to track file anymore

git rm filename

# Record changes to git. Default editor will open for a commit message.

# (Visible via git log)

# Once files are commited, they are history.

git commit

# A short hand for commiting files and writing a commit message via one command

git commit -m 'Some commit message'

# Changing the history :) If you want to change your previous commit,

# you can, if you haven't pushed it yet to a remote repo

# Simply make new changes, add them via git add, and run the following command.

# Past commit will be ammended.

git commit --amend

**Advanced**

# Unstage pending changes, the changes will still remain on file system

git reset

# Unstage pending changes, and reset files to pre-commit state. If

git reset --hard HEAD

# Go back to some time in history, on the current branch

git reset tag

git reset <commit-hash>

# Save current changes, without having to commit them to repo

git stash

# And later return those changes

git stash pop

# Return file to it's previous version, if it hasn’t been staged yet.

# Otherwise use git reset filename or git reset --hard filename

git checkout filename

**Comparing changes**

# See current changes, that have not been staged yet.

# Good thing to check before running git add

git diff

# See current changes, that have not been commited yet (including staged changes)

git diff HEAD

# Compare current branch to some other branch

git diff branch-name

# Same as diff, but opens changes via difftool that you have configured

# -d tells it to open it in a directory mode, instead of having to open

# each file one at a time.

git difftool -d

# See only changes made in the current branch (compared to master branch)

# Helpful when working on a stand alone branch for a while

git difftool -d master..

# See only the file names that has changed in current branch

git diff --no-commit-id --name-only --no-merges origin/master...

# Similar to above, but see statistics on what files have changed and how

git diff --stat #Your diff condition

### Working with Remote Branch

# See list of remote repos available. If you did git clone,

# you'll have at least one named "origin"

git remote

# Detailed view of remote repos, with their git urls

git remote -v

# Add a new remote. I.e. origin if it is not set

git remote add origin <https://some-git-remote-url>

# Push current branch to remote branch (usually with the same name)

# called upstream branch

git push

# If a remote branch is not set up as an upstream, you can make it so

# The -u tells Git to remember the parameters

git push -u origin master

# Otherwise you can manually specify remote and branch to use every time

git push origin branchname

# Just like pushing, you can get the latest updates from remote.

# By defaul Git will try to pull from "origin" and upstream branch

git pull

# Or you can tell git to pull a specific branch

git pull origin branchname

# Git pull, is actually a short hand for two command.

# Telling git to first fetch changes from a remote branch

# And then to merge them into current branch

git fetch && git merge origin/remote-branch-name

# If you want to update history of remote branches, you can fetch and purge

git fetch -p

# To see the list of remote branches

# -a stands for all

git branch -a