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

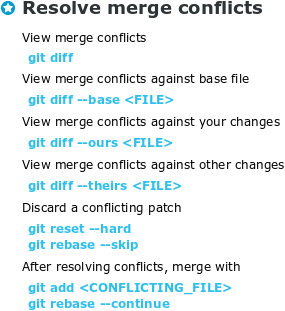
git checkout --theirs .  
git add .

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]

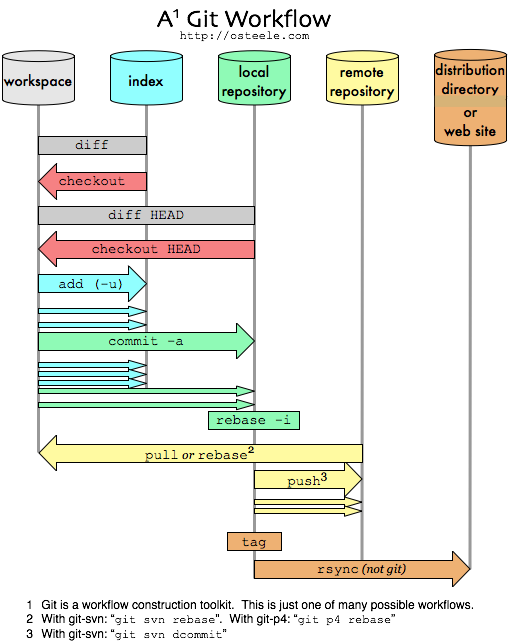
Removes the file from version control but preserves the file locally

$ 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):

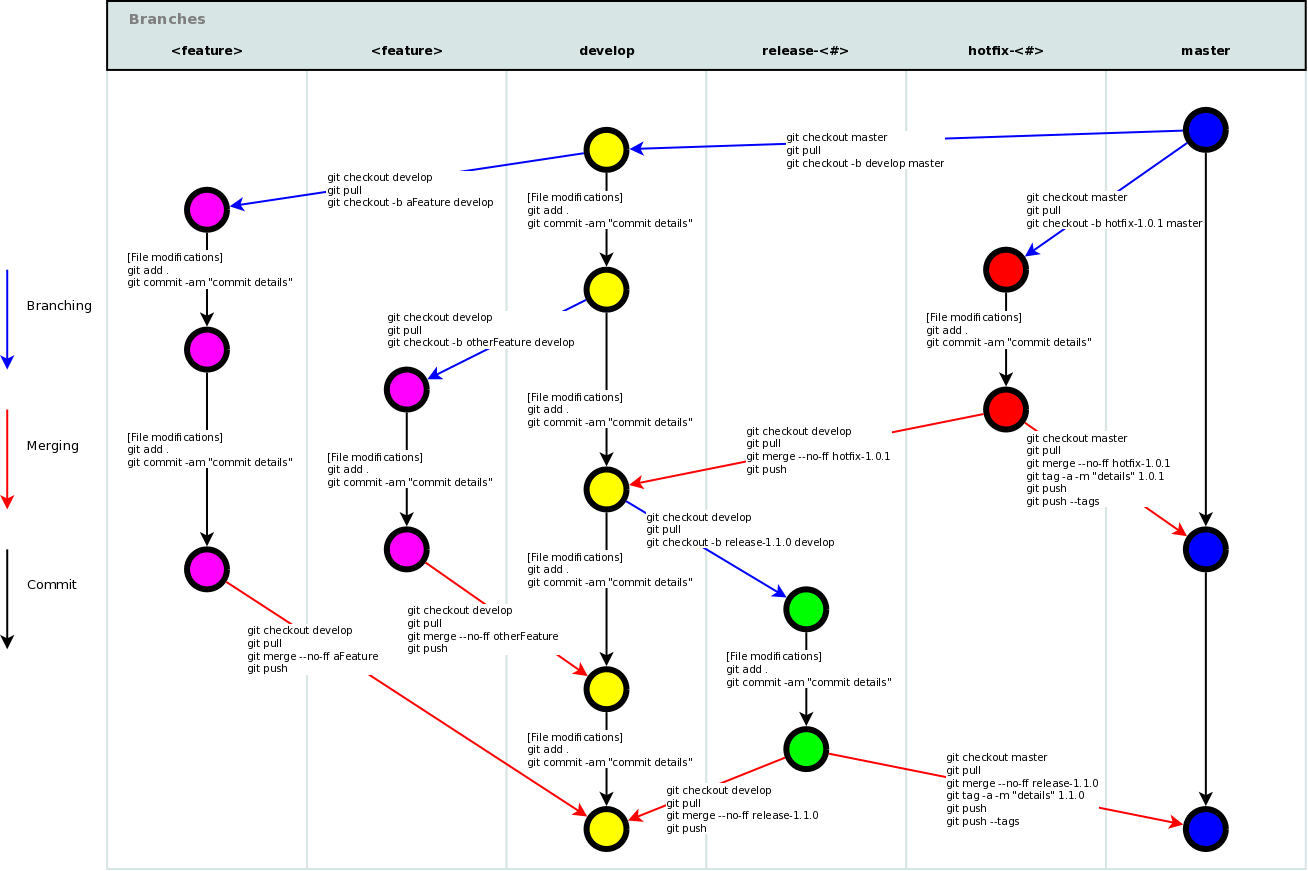
$ git revert <commit>

##### 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]

https://github.com/k88hudson/git-flight-rules/blob/master/README.md#i-tried-to-push-my-amended-commit-to-a-remote-but-i-got-an-error-message