Git Cheat Sheet

by schmidt7 via cheatography.com/30462/cs/9075/

Getting started

`git clone ssh://domain.org/project.git``	Get a copy of an extern project.
`git init myProject`	Creates a local new project.
`git config --global user.name "Max Mustermann"`	Sets your displayed name (obligatory).
`git config --global user.mail "max@mustermann.de"`	Sets your email address (obligatory).
`git config --global --list`	Verify global settings.

Global configuration is saved in the file `.gitconfig` file in your home directory. Local configuration (`git config --local`) in each repository in the file `.git/config`

Exploring the repository

`git status`	Important command to see the current state of the local repository.
`git log`	Shows the history of the project.
`git log --graph --oneline`	Useful option for a larger overview.
`git branch`	Shows the current branch and all local branches that are available. (`-a` for all available branches).
`git blame $file`	see who edited the file and when.
`gitk`	graphical tool to explore the history.
`git gui`	graphical tool for git.

Basic commands

`git add $file`	Adds *$file* to the index/staging area
`git commit`	Saves all changes that are added to the index/staging area in a commit/snapshot in your local repository.

Basic commands (cont)

`git reset $file`	Removes *$file* from the index/staging area. Useful when the file was added accidentally.
`git checkout $file`	Restores *$file* to the state of the latest commit. All local changes for this file will be deleted.
`git fetch`	Gets the most recent version from the remote repository. Changes won't be applied to the working directory. Might be useful when a conflict with the local version is expected.
`git pull`	Like **git fetch**, but changes are immediately applied to the working directory.
`git mv $name1 $name2`	To move a file within Git. Should be used to preserve the file-history
`git rm $filename`	Removes the file

Things that might be useful

How to change the editor for commit messages?

> `git config --global core.editor=editorname`

How to learn more about a certain command?

> `git command --help`

How to specify a range of commits?

> Via two dots. For example: `git log 9e45..e312`
> The SHA1-ID can be found in the history.

What does HEAD^2 mean?

> Two commits(snapshots) behind HEAD, the current snapshot-pointer.
> Note that ^ and ~ are aequivalent and you can combine all notations.
> `HEAD^1` is equivalent to `HEAD~`
> `HEAD^^~2` is equivalent to `HEAD~3^`

I only made small changes in each line. How can see them directly?

> Use `git diff $filename --word-diff=color`

I want to introduce the same commit that was made in another branch in my branch

> Use `git cherry-pick $ID`

Sponsored by **CrosswordCheats.com**
Learn to solve cryptic crosswords!
http://crosswordcheats.com

Things that might be useful (cont)

I know that there is a bug introduced between two far away versions. Can git help me to find the commit that introduced the bug?

> Yes, read the documentation for `git bisect`

$IDs are hard to remember. What can I do?

> If you often have to refer to a certain commit/snapshot of your project you can use `git tag -a $tagname $ID` to give it a name that is easier to remember. This is especially useful to remember the commit of a published version (eg. tagname *v.0.8*)

How to prevent Git from committing certain files?

> If the rule should apply to every developer create a file `.gitignore` in the repository and write in each line a pattern that should be excluded. Commit the file.
> Example: *.tmp.
> To exclude it locally, change the file `.git/info/exclude` in your repository

These 40 character *$IDs* in the log are too long.

> In order to refer to a ID, you only have to provide as many characters as necessary to be unique in your repository. But at least 4 characters.

Resolving Conflicts

1. Run `git status` to see the files where the conflict occurs.
2. Open the file that was changed in both branches with your editor.
3. Look for passages that look like

```
<<<<<<< HEAD
command1
command2
=======
command3
>>>>>>> higherbranch
```

4. Decide how the file should look like. In this example you might want to have all 3 commands in the merged result or only one of them.
5. Edit the file in a way that the result makes sense. Delete all lines that were introduced by Git.
6. Save it. Add it to the index via `git add $filename` and run **git status** again.
7. Edit all files with conflicts until it says all conflicts are resolved. You can finish it via `git commit`

There are also tools available that help you in comparing the different versions. Use `git mergetool --tool-help` to get a list of all available tools. Run `git mergetool --tool=$toolname` to use it for resolving your conflict.

Basic workflow

1. Check if you're in the right branch to work in with `git branch`.
2. Get the most recent changes from the remote repository with `git pull`.
3. Make your changes.
4. Before adding the files with `git add $filename` you might want to check what changes you made in each file via `git diff $filename`.
5. Use `git status` to verify if all changed files you want to commit are added to the index.
6. If you want to check again what changes exactly you're going to submit, use `git diff --cached` to see them all.
7. Everything is fine? Use `git pull` a last time for the case that someone submitted faster than you.
8. Finally use `git commit` to open an editor to provide a helpful commit message that describes your changes.
9. Publish your changes that everybody can see them with `git push`

Working with branches

Command	Description
`git branch $branchname`	Creates a new branch with the name *$branchname* from the current position.
`git checkout $branchname`	Change to branch *$branchname*
`git checkout -b $branchname`	Creates branch *$branchname* and switches directly to it
`git merge $otherbranch`	Merges *$otherbranch* to the current positon
`git merge --abort`	Aborts a merge in case of a conflict
`git stash`	Saves your current state
`git stash apply`	Restores the stashed state.
`git stash --help`	More information incl. options *list*, *drop*, *pop*

`git stash` is useful when you want to change your branch (for example to fix something important) but don't want to commit your current work already.
When you return use `git stash --apply` to continue your work

Sponsored by **CrosswordCheats.com**
Learn to solve cryptic crosswords!
http://crosswordcheats.com

Working with branches step-by-step

1. List all available branches with `git branch -a`
2. If you're not in the branch from where you want to start your development, switch to it via `git checkout $branchname`
3. You want to work on a issue or a feature concerning this branch? Create your sub-branch via `git checkout -b $yourbranchname`. Please follow naming conventions for *$yourbranchname* if your team agreed on any.
4. Do your changes, commit.
5. Optionally, if your separate development takes a long time, consider to merge changes from the higher branch from time to time with `git merge $higherbranch` in order to reduce the number of possible conflicts at the end. Note that you have to use `git pull` inside of *$higherbranch*` before you get the most recent changes.
6. At the end of your development merge the changes that were made in the meantime from the higher-branch as described in step 5.
7. In the case of merge conflicts, solve them. Give attention to the messages in `git status`
8. Check again if everything was committed. Test your feature.
9. If you're confident that everything is correct, go to the next higher branch (where you want to have your development included) via `git checkout $higherbranch` and merge your feature with `git merge $yourbranchname`.
10. There shouldn't be any conflicts in the last step. If there are, it means that someone published changes in the meantime. Use `git merge --abort`, switch back to your branch and repeat the steps beginning from 6.

Note: It is not the only way how to work with branches. It is also not the most elegant way but should be relatively safe with regard to possible errors.

Undo and rewriting history

How to add an additional file to the last commit or re-edit the commit message?

> If you haven't pushed the commit yet do the following: Add the file you forgot to the index and use `git commit --amend`. In the case you only want to change the message, just type the command without adding a file to the index.
> Alternative: Use `git reset --soft HEAD^`. It sets your local repository back by one commit while leaving the working directory unchanged. You can start from there to add again files and to commit again.

Undo and rewriting history (cont)

How to delete my last commit (locally)?

> In the case you haven't pushed it yet, you can use: `git reset --hard HEAD^` to reset your local repository and your working directory to the commit before.
> For the case you already published it, use the command `git revert HEAD` to create an anti-commit for the last commit. `git revert` works also with older commits. In this case provide the *$ID* instead of HEAD.

I haven't pushed the last 4 commits yet and I want to change them into two bigger ones and to adjust the messages. How can I do it?

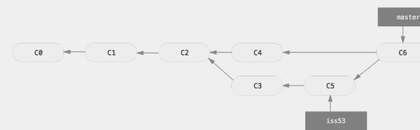
> Use `git rebase --interactive HEAD^4`. Change the letters in front of the presented commits in the way you want to change them.

I lost something.

> Use `git reflog`. You might be able to restore it. (Checkout to an earlier reflog entry or cherry-pick it again)

Warning: Only change commit messages or change history in the case that they are **NOT published** yet (via git push) or you're completely sure that nobody based his work on it.

Merged subbranch



From the (online-)book ProGit (https://git-scm.com/doc). Example of a merged issue branch.

Sponsored by **CrosswordCheats.com**
Learn to solve cryptic crosswords!
http://crosswordcheats.com