Table of Contents

Git Cheat Sheet

1. Getting Started and Configuration (Setup)

git config –global user.name “Your Name Surname”	Defines the user name for Git. If you don't use the –global flag, it only changes the user name in the relevant repo.
git config –global user.email “mail@youraddress.com”	Defines your email for Git. If you don't use the –global flag, it only changes the user name in the relevant repo.
git config –global init.defaultBranch main	Sets the default main branch as main.
git config –global merge.tool meld	Defines a default tool (e.g., Meld) to resolve conflicts.
git config –list	Lists all your currently available Git settings.
git init	Converts the relevant folder into a repository tracked by Git.
git clone <url>	Downloads an existing project from a remote server (GitHub) to your computer.

2. Basic Workflow (Saving and Reviewing Changes)

git status	Shows the current status of the project. (The command you will use the most).
git diff	Shows all changes (line by line) that have not been added to the staging area.
git diff <file_name>	Shows the changes only in that file.
git add <file1> <file2>	Adds the specified files to the staging area.
git add .	Adds all changes in the folder to the staging area.
git add -p <file_name>	Allows you to add changes by reviewing them piece by piece (interactive).
git commit	If you write only this command, the default terminal editor opens for you to write a detailed message.
git commit -m “Message”	Saves the changes to the history with a short message.
git commit -a	Automatically adds all changed files (except newly created ones) and opens the commit screen.
git commit -am “Message”	Both automatically adds all changed files and saves with a message (a combination of add . and commit -m).
git mv <old_name> <new_name>	Renames or moves the file (does this by notifying Git; if you change the file name yourself, Git thinks the relevant file is deleted and a new file is created).

3. Working with Remote Repositories (GitHub)

git remote add origin <url>	Connects your local repository to a repository on GitHub.
git push -u origin main	Sends your codes to GitHub (-u is used for the first time).

git pull	Downloads the up-to-date codes from GitHub and merges them with your existing code.
git pull –rebase <remote> <branch>	Gets updates with rebase to avoid creating unnecessary merge commits while pulling (E.g., `git pull –rebase origin main`).
git fetch	Downloads the changes in the remote repository to the computer but does not merge them.

4. Advanced Branching

git branch	Lists local branches.
git branch -a	Lists all branches, both local and remote.
git branch -r	Lists only the branches in the remote repository.
git branch –merged	Lists branches that have been successfully merged into the active branch.
git branch <new_branch>	Creates a new branch but does not switch to that branch.
git branch -m <new_name>	Changes the name of the branch you are on.
git branch -d <branch_name>	Safely deletes the branch whose work is finished and has been merged.
git branch -D <branch_name>	Forcefully deletes the branch even if it is not merged.
git branch –track <new_branch> <remote_branch>	Creates a local branch tracking a branch in the remote repository.
git checkout <branch_name>	Switches to another branch.
git checkout -	Quickly returns to the previous branch you were on (like the “previous channel” button on a TV remote).
git checkout -b <branch_name>	Both creates a new branch and instantly switches to that branch.
git checkout -b <new_branch> <existing_branch>	Creates and switches to a new branch derived from an existing branch.
git checkout <commit-hash> -b <new_branch>	Starts a new branch from the point of an old commit.
git checkout <branch_name> – <file_name>	Copies a specific file from another branch to the branch you are on.
git cherry-pick <commit_hash>	Takes a single commit (and only that one) from another branch and applies it to the branch you are on.

5. Merging & Conflict Resolution

git merge <branch_name>	Merges the specified branch into the branch you are on.
git mergetool	If a conflict occurs during merging, it allows you to resolve it by opening the visual tool you configured (e.g., Meld).
git rebase <branch_name>	Moves the starting point of your branch to the most up-to-date state of the specified branch. Provides a linear history.
git rebase –abort	If too many conflicts occur during rebase or things go wrong, it cancels the process completely and returns to the beginning.

- Continuing Rebase after resolving a Conflict:
- Fix the conflicting files.
    - `git add <resolved_file>` (or `git rm <file>` if not needed)

- `git rebase –continue` (This is repeated until the process is finished).

6. Editing History (Amend) and Squashing

A) Modifying the Last Commit (Amend)

> You wrote the wrong message or forgot to add a file. (It is safe if you haven't pushed to the remote repository yet).

`git commit –amend`	Includes the files in the staging area into the last commit and opens the editor for you to change the message.
`git commit -a –amend`	Automatically adds all changed files and updates the last commit.
`git commit –amend –no-edit`	Includes only the files you forgot into the last commit without changing the message.
`git commit –amend –date="date"`	Changes the date of the last commit.
`GIT_COMMITTER_DATE="date" git commit –amend`	Sets both the commit and committer date to a past time. (E.g.: `git commit –date="date –date='1 day ago'" -am "Message"`)

B) Squashing (Interactive Rebase / Squash)

> Combining a large number of small commits into a single, meaningful commit.

1. `git rebase -i <one_before_the_oldest_commit_you_want_to_merge>`
2. In the editor that appears:

```
Previous State:
pick 1a2b3c4 Initial structure
pick 5d6e7f8 Minor fix
pick 9g0h1i2 Color adjustment

Next State (We are merging the second and third into the first):
pick 1a2b3c4 Initial structure
squash 5d6e7f8 Minor fix
squash 9g0h1i2 Color adjustment
```

1. Save and exit.

A new message screen will appear, write a single message for the merged commit.

7. Review, Search, and Logs

`git log`	Lists the detailed commit history.
`git log –oneline`	Shows each commit on a single line, cleaner.
`git log –author="name"`	Shows only the commits made by a specific person.

git log -p <file_name>	Shows how a specific file has changed in the past (as a line-by-line diff).
git log –oneline <origin/master>..<remote/master> –left-right	Lists the differences between two different branches (who is ahead of whom) with arrows.
git log -S 'word'	Searches for commits where a specific word was added/deleted.
git log -S 'word' –pickaxe-regex	Performs the above search with Regex (Regular Expression).
git grep “SearchTerm”	Finds the text “SearchTerm” in files throughout the entire project.
git grep “SearchTerm” v2.5	Searches for this word in a specific version or branch.
git blame <file_name>	Shows who last changed each line of a file and when. (It is legendary for finding who broke the code).
git reflog show	Keeps a log of every action you make in Git (including deleted branches, reset commits). You can find everything you lost here.
git reflog delete	Deletes the reflog history.

8. Tagging - Versioning

> It is used to mark important moments (for example, the v1.0.0 release).

git tag	Lists all existing tags.
git tag -n	Lists tags along with their messages next to them.
git tag <tag_name>	Creates a simple, lightweight tag (E.g.: `git tag v1.0`).
git tag -a <tag_name>	Creates an annotated tag where extra information can be added (Recommended).
git tag <tag_name> -am 'Message	Directly creates an annotated tag with a message.

9. Stash Management

> You are working but your work is not finished. It is the method of saving unfinished codes when you need to switch to another branch.

git stash	Puts current changes into a temporary “drawer” (stash) and cleans the working area.
git stash list	Lists all records in the stash (e.g., `stash@{0}`, `stash@{1}`).
git stash pop	Brings back the most recently stashed codes and deletes them from the stash.
git stash apply	Brings the most recent codes in the stash to the workspace but does not delete them from the stash.
git stash apply stash@{number}	Brings back the stash at a specific sequence.
git stash drop	Throws the last stash completely into the trash.

10. Undo and Emergencies (Undo / Reset)

git checkout HEAD <file_name>	Cancels all uncommitted changes in a specific file and reverts the file to its state in the last commit.
git reset HEAD	Removes the files you took into the staging area with git add back out of the staging area (codes are not deleted).
git reset <commit_hash>	Reverts the project to the specified old commit but does not delete the codes (Leaves the changes in the working area).
git reset –keep <commit_hash>	While returning to a past point, it tries to preserve the local (uncommitted) changes you made in the working directory at that moment.
git reset –hard <commit_hash>	DANGEROUS! Reverts the project to that commit and completely deletes all subsequent codes.
git reset –hard <remote_repo/branch_name>	(E.g. `git reset –hard upstream/master`) Completely deletes your own local codes and makes it exactly the same as the branch in the remote repository.
git revert <commit_hash>	Adds a new commit that does the exact opposite of what a faulty commit did. (It is the safest way to undo errors in public repos because it does not change the history).

11. Useful Tips: Clearing Cache After .gitignore

Let's say you previously added a file to Git, then you wrote it into .gitignore but Git still continues to track that file.

Run the following in order to clear the cache:

```
git rm -r --cached .
git add .
git commit -m "Git tracking cache (.gitignore settings) cleared"
```

Taken from UCH Wiki.