



# Git and Github

# Datasheet



## INDEX

### 1. Introduction

1. What is Git
2. Why Git
3. Features of Git
4. What is Github

### 2. Configuring Git for the first time

### 3. General Git Features

1. Initializing Git
2. Staging files
3. Making a commit
4. Status of files and log

## **4. Git Help**

## **5. Git Branching**

1. Making a new Git Branch
2. Checking all available Branches
3. Switching to other Branches
4. Making a new branch and directly switching to it
5. Deleting a Branch
6. Merging two Branches

## **6. Working with Github**

1. Push local repo to GitHub
2. Pull local repo from GitHub
3. Pull branch from GitHub
4. Push branch to GitHub

## **7. Git Undo**

1. Git Revert
2. Git Reset
3. Git Amend

## What is Git :-



- Git is a version control system.
- Git helps you keep track of code changes.
- Git is used to collaborate on code.
- Git and GitHub are different things.

## Why Git :-

- Over 70% of developers use Git!
- Developers can work together from anywhere in the world.
- Developers can see the full history of the project.
- Developers can revert to earlier versions of a project.

## Features of Git :-

- When a file is changed, added or deleted, it is considered modified
- You select the modified files you want to Stage
- The Staged files are Committed, which prompts Git to store a permanent snapshot of the files
- Git allows you to see the full history of every commit.
- You can revert back to any previous commit.
- Git does not store a separate copy of every file in every commit, but keeps track of changes made in each commit!

---

## What is GitHub :-



- Git is not the same as GitHub.
- GitHub makes tools that use Git.
- GitHub is the largest host of source code in the world, and has been owned by Microsoft since 2018.

# Configuring git for the first time :-



```
$ git config --global user.name "<Enter your username here>"
```

```
$ git config --global user.email "<Enter your email here>"
```

## General Git Features :-

### Initializing Git :-

```
$ git init
```

Git now knows that it should watch the folder you initiated it on. Git creates a hidden folder to keep track of changes.

### Staging files/Adding files to Git repo :-

Staged files are files that are ready to be committed to the repository you are working on.

When you first add files to an empty repository, they are all untracked. To get Git to track them, you need to stage them, or add them to the staging environment.

```
$ git add <filename with extension>
```

### Staging all files in a folder :-

```
$ git add --all
```

OR

```
$ git add -A
```



## Making a Commit :-

Adding commits keep track of our progress and changes as we work. Git considers each commit change point or "save point". It is a point in the project you can go back to if you find a bug, or want to make a change.

When we commit, we should always include a message.

```
$ git commit -m "<Enter your message here>"
```

## Git Commit without Stage :-

Sometimes, when you make small changes, using the staging environment seems like a waste of time. It is possible to commit changes directly, skipping the staging environment.

```
$ git commit -a -m "<Enter your message here>"
```

## Status of files and log :-

```
$ git status
```

## File status in a more compact way :-

```
$ git status --short
```

## Log of a file :-

Log is used to view the history of commits for a repo.

```
$ git log
```

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

# Git Help :-



If you are having trouble remembering commands or options for commands, you can use Git help.

See all the available options for the specific command -

```
$ git <command> -help
```

See all possible commands -

```
$ git help --all
```

If you find yourself stuck in the list view, SHIFT + G to jump the end of the list, then q to exit the view.

## Git Branching :-

In Git, a branch is a new/separate version of the main repository. Branches allow you to work on different parts of a project without impacting the main branch. When the work is complete, a branch can be merged with the main project.

We can even switch between branches and work on different projects without them interfering with each other.

## Making a new Git Branch :-

```
$ git branch <name of branch>
```

## Checking all available Branches :-

```
$ git branch
```



## Switching to other Branches :-

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

## Making a new branch and directly switching to it :-

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

## Deleting a Branch :-

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

## Merging two Branches :-

It's preferred to change/switch to master branch before any branch needs to be merged with it.

```
$ git merge <branch name>
```

This will merge the specified branch with our master branch.

# Working with Github :-




Create a github account to create your remote repositories. Now, create a new repo where we will be uploading our files from local repo.

## Create a new repository


A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner \*

Repository name \*


 durgeshm01722 ▾

/


example 

Great repository names are short and memorable. Need inspiration? How about [fluffy-adventure?](#)

Description (optional)

☒  **Public**

Anyone on the internet can see this repository. You choose who can commit.

☐  **Private**

You choose who can see and commit to this repository.

**Initialize this repository with:**  
Skip this step if you're importing an existing repository.

☐ **Add a README file**

This is where you can write a long description for your project. [Learn more.](#)

☐ **Add .gitignore**

Choose which files not to track from a list of templates. [Learn more.](#)

☐ **Choose a license**

A license tells others what they can and can't do with your code. [Learn more.](#)

Create repository

**Note** - Local repository (repo.) means the repo. which is on our system whereas, remote repo. means the one which is on other remote system/server, for eg. - GitHub, GitLab, Bitbucket, etc.



## Push local repo to GitHub :-



Copy the url or the link of the repo that we just created. As an example, it should look like this - <https://github.com/durgeshm01722/example.git>

Paste the copied url in the below git command.

```
$ git remote add origin <paste copied URL here>
```

'**git remote add origin <URL>**' specifies that we are adding a remote repository, with the specified URL, as an origin to our local Git repo.

Finally, pushing our master branch to the origin URL (remote repo) and set it as the default remote branch.

```
$ git push --set-upstream origin master
```

Go back into GitHub and see that the repository has been updated.

## Pushing local repo to github after doing the above process at least once :-

First commit all the changes. Then push all the changes to our remote origin i.e. remote repo on github.

```
$ git push origin
```

## Pull local repo from GitHub :-

Git pull is used to pull all changes from a remote repository into the branch we are working on. It is a combination of fetch and merge. Use it to update your local Git.

```
$ git pull origin
```

## Pull branch from GitHub :-



First, check which branches we have and where are we working at the moment by 'git branch' command. Since we do not have the new branch on our local Git which is to be pulled from the Github. So, to see all local and remote branches, use -

```
$ git branch -a
```

## For viewing only remote branches :-

```
$ git branch -r
```

Now, the new branch is seen in the console but it is not available on our local repo. So, let's check it out using 'git checkout <branch name>'. Now run 'git pull' to pull that branch on our local repo. We can now check the available branches using 'git branch'.

## Push branch to GitHub :-

First, let's create a new local branch which we will be pushing to Github. Enter the command as 'git checkout -b <branch name>'. You can check the status of the files in this current branch using 'git status'. Commit all the uncommitted changes for all the files in this branch using 'git commit -a -m "<Message>"'. Now push this branch from our local repo to Github using 'git push origin <branch name>'.

## Git clone from GitHub :-

We can clone a forked repo from Github on our local repo. A clone is a full copy of a repository, including all logging and versions of files. Move back to the original repository, and click the green "Code" button to get the URL to clone. Copy the URL.



Now in the git bash, enter the following command to clone the copied repo onto your local machine -

```
$ git clone <copied URL>
```

To specify a specific folder to clone to, add the name of the folder after the repository URL, like this -

```
$ git clone <copied URL> <folder name>
```

# Git Undo :-



## Git Revert :-

'**revert**' is the command we use when we want to take a previous commit and add it as a new commit, keeping the log intact. First thing, we need to find the point we want to return to. To do that, we need to go through the log. To avoid the very long log list, use the **--oneline** option which gives just one line per commit showing –

- i. The first seven characters of the commit hash
- ii. The commit message

## Git Revert HEAD :-

We revert the latest commit using '**git revert HEAD**' (revert the latest change, and then commit). By adding the option **--no-edit**, we can skip the commit message editor (getting the default revert message).

```
$ git revert HEAD --no-edit
```

## Git Revert to any commit :-

To revert to earlier commits, use '**git revert HEAD~x**' (x being a number. 1 going back one more, 2 going back two more, etc.)

## Git Reset :-

'**reset**' is the command we use when we want to move the repository back to a previous commit, discarding any changes made after that commit. First, get the seven characters of the commit hash from the log for the commit that you want to go back for. Then we reset our repository back to that specific commit using '**git reset commithash**' (commithash being the first 7 characters of the commit hash we found in the log).



```
$ git reset <commithash>
```

## Git Undo Reset :-

Even though the commits are no longer showing up in the log, it is not removed from Git. If we know the commit hash, we can reset to it using '`git reset <commithash>`'.

## Git Amend :-

`commit --amend` is used to modify the most recent commit. It combines changes in the staging environment with the latest commit, and creates a new commit. This new commit replaces the latest commit entirely.

One of the simplest things you can do with `--amend` is to change a commit message.

```
$ git commit --amend -m "<Commit Message>"
```

Using this, the previous commit is replaced with our amended one.

## Git Amend Files :-

Adding files with `--amend` works the same way as above. Just add them to the staging environment before committing.



**Thanks for Reading!!!**

**Follow for more @Lovee Kumar**