What is Git?

- was created by **Linus Torvalds** in 2005
- maintained by Junio Hamano since then
- is a popular **version control system** (VCS)
  - version control systems are a category of software tools that helps in recording changes made to files by keeping a track of modifications done in the code.

Working with Git

- Initialize `git` on a folder, making it a repository
- `git` now creates a **hidden folder** to keep track of changes in that folder
- 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?

- GitHub is the largest host of source code in the world and is owned by *Microsoft* since 2018
- alternatives :
  - bitbucket
  - gitlab
  - codeberg
  - etc...

Git Install

- https://www.git-scm.com

Configure Git

```
git config --global user.name "Alok Shandilya"
git config --global user.email "xxxx...@..xxx"
```

- here, `""` are optional for e-mail (because of no space in b/w)
- ***NOTE***
  - to set username and e-mail for just the current repo, remove `--global` option

Initialize Git

```
git init
```

- `Initialized empty Git repository in /home/aloks/test/.git/`

Git Adding New Files

- Files in your `git` repository folder can be in one of 2 states:
  - Tracked - files that `git` knows about and are added to the repository
  - Untracked - files that are in your working directory, but not added to the repository

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 Staging Environment

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

```
git add file1.txt file2.cpp file3.js
```

- adds file1.txt, file2.cpp, file3.js to staging area

```
git add .
```

- stages new files and modifications, without deletions (on the current directory and its subdirectories).

```
git add -A
```

- stages all changes

```
git add -u
```

- stages modifications and deletions, without new files

```
git status
```

- changes in working directory

```
git status --short
git status -s
```

- changes in working directory (in more compact way)
  - Short status flags are:
    - `??` - Untracked files
    - `A` - Files added to stage
    - `M` - Modified files
    - `D` - Deleted files

Git Commit

```
git commit
```

- will open default editor for writing commit message, description (optional)
- `git config --global core.editor nvim`
- `git config --global core.editor emacs`

```
git commit -m <commit-message>
```

```
git commit -a -m <commit-message>
```

- commit the updated files directly, skipping the staging environment
  - not recommended
  - skipping the stage step can sometimes make you include unwanted changes

Git Commit Log

```
git log
```

```
git log --oneline
```

- compact (every commit in single line)

Git Help

```
git <command> -h
git <command> -help
```

- available options for the specific command
- `--help` opens the relevant `git` man page

```
git help --all
```

- all possible commands

Git Branch

```
git branch <branch-name>
```

- creates a new branch

```
git branch
```

- lists the branches

```
git checkout <branch-name>
git switch <branch-name>
```

- switch to `<branch-name>`

```
git checkout -b <branch-name>
git switch -c <branch-name>
```

- **creates** (if not pre-exists) and **switches** to `<branch-name>`

Git Merge

```
git merge <branch-name>
```

- merges `<branch-name>` to current branch
- if the `<branch-name>` came directly from `main`, and no changes are made to `main` while working, this is a continuation of `main`. it is seen as ***"Fast-forward"***, just pointing both `main` and `<branch-name>` to the same commit.

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

- deletes `<branch-name>`
  - must be fully merged in its upstream branch, or in `HEAD`
  - can't delete current branch

```
git branch -D <branch-name>
```

- `-D` is shortcut for `--delete --force`
- deletes `<branch-name>` even if not merged

Merge Conflict

```
git merge <branch-name>
```

- merges `<branch-name>` to current branch
- if not able to merge fully and `conflict` occurs, solve it in the respective files. by seeing the differences between the versions and edit it as wanted
- You can work through the conflict with a number of tools:
  - use a mergetool. `git mergetool` to launch a graphical `mergetool` which will work you through the merge.
  - look at the diffs. `git diff` will show a three-way diff, highlighting changes from both the `HEAD` and `MERGE_HEAD` versions.
  - look at the diffs from each branch. `git log --merge -p <path>` will show diffs first for the `HEAD` version and then the `MERGE_HEAD` version.