**Before class - self note:**

- **NOTE: Remove pre-commands time and start straight from "What is git?". Follow the doc.**
- No diagram for staging index.
- Till gitk and project on day 1.
- Cover merge conflict later only after they start working on projects.
- Ideally day 2 will be project working and merge conflict. And day 2 of git should be taken after DBMS.

## How to work in a team? [remove]

How do you plan to work in a team? Different files? What if you guys need to work on same files as well? And what if there is not much communication between the team.

How will you deal with it?

1. No person to person sharing. **We need a centralised place to store the project** so that everyone can fetch from there. It ensures it has the latest code.
2. It would be great if we could **see the changes made by people in the team**, because if some of our code is missing or something is added, we would know how.
3. We need to follow a basic discipline on our part, so **we do not overwrite anyone's code**.
4. We need to store **our code in form of versions** which we can refer back if needed.

GIT is a software which will help us achieve all of this.

**What is GIT?**

Git is a Version Control System. A version control system stores and maintains different versions of our code, so that we can **go back to any version if needed**.

Draw the linear commits tree diagram.

**Versions are maintained in form of commits.**

[contd...]

**Commits:**

**Commits are saved version of your work**.

These commits let us **go back to any previous version of our code**, if needed. When we go back to any previous version of our code, every file and folder will be exactly same as, as they were when the commit was being made, just like **any saved game**.

With these commits, **we can also see the changes made in that commit with respect to the previous commit.** This helps us track the progress by everyone in the team and also maintain history of our code. When we work on big projects, **this code history becomes very important**, much more than the code. After 1-2 years, we may come across some code and wonder why it has been added, these commits will help us find out the reason.

Infact, GIT does not maintain different copies of our code as versions, and stores commits in form of **changes only with respect to the previous commit**. It generates the required version of the code using these commits (or change information).

**Repositories:**

A GIT repository is a place where commits are stored.

Draw the star diagram - centralised repo and local repos.

In GIT we maintain a centralized repository to which everyone working on the project will be connected. All of us, add our code and fetch latest code from this centralised repository - no one to one communication.

**Everyone adds our code to the centralised repository in form of commits.**

**Distributed Version Control System:**

GIT is distributed VCS.

In a non-distributed VCS, in our local we just have a copy of our code on which we work.

But in a distributed VCS, in our local, we not only have a copy of our code, but a complete local version of the repository in itself, maintaining all the commit history of the project - known as **local repository**. This local repository is almost exactly same as centralised repository.

DVCS helps us in 2 ways:

- Even if centralised repository gets deleted, we can easily replicate it using any of the local repositories.
- All of us can work on our local repositories in our own way, without affecting the central repository.

**So, in DVCS commits are first made to the local repo and then moved to the centralised repo.**

**Installation and configuration:**

Install GIT: `$ sudo apt-get install git`

Setting GIT configuration:

System level: `git config --system`

User level: `git config --global`

Repository level: `git config --local`

```
$ git config --global user.name "Your Name"
$ git config --global user.email "your.email@example.org"
$ git config --global color.ui auto
$ git config --global push.default simple
```

Check ~/.gitconfig

**Git Init - Centralised repository:**

We will now replicate a scenario where a couple of team members are working on a project using GIT. We create both the local repositories along with the central repository on our local system.

```
$ cd ~
$ mkdir repo.git
$ cd repo.git
$ git init --bare
# Explain --bare later
$ ls -la
```

This is a centralised repository, currently present in your system. But in reality, it will be present remotely somewhere over the internet.

## Git Clone - Local repository:

Now, let's clone our first local repository (your) from the centralised repository.

```
$ cd ~
$ git clone repo.git repo01
$ cd repo01
$ ls -la
$ cd .git
```

The **.git** folder is the place where git stores all the commits, that makes repo01 as git repo.

We create files and folders related to project (our code) outside the .git folder which is known as the **working tree** of the repository.

But, centralised repository's folder structure was different. It didn't contain the working tree and had .git folder's content directly inside repo.git.

**Standard repositories v/s Bare repositories:**

The difference is because of the --bare keyword, which could be added to both clone and init.

Without --bare keyword, a **standard repository** is created. A standard repository contains the working tree. The local repositories are always standard repositories with working trees, where we work / code and make our commits, which will then be stored in the .git folder.

The --bare keyword creates a **bare repository**. A bare repository does not contain the working tree. The repo.git was a bare repository without a working tree. The centralised repositories are mostly bare, as the purpose of it is only **to sync commits between repositories**. No one actually works on it, so no one will ever make a commit on it - centralised repositories do not belong to anyone. So it doesn't need a working tree.

**NOTE: A bare repository doesn't contain the working tree / code, but it can generate the latest code from the commits it stores.**

## Creating random files:

**We create some random files which we would want to commit. These files will be your project related files, like html, php, css, js.**

```
$ cd ~/repo01
$ mkdir datafiles
$ touch datafiles/data.txt
$ echo "abc" > test01
$ cat test01
$ echo "bar" > test02
$ echo "foo" > test03
$ git status
# It shows the changes being made
```

Git status: It shows the status of the working tree.

It is a safe command. Use it as many times as you want to know exactly what is happening.

**Staging Area / Index:**

**When we commit, only those changes added to Staging Index gets saved.**

Before making a commit, we must tell GIT what are the changes we want to store in the commit. We do so by adding the changes to the staging area/index. Staging area/index is an intermediate stage for the changes before the commit.

```
$ git add .
# To add changes to staging area
# Dot denotes all the changes in the current directory
# We can provide specific file names
$ git status
# All the changes become green in color
# GIT is now aware of these changes
# If we commit now, all these changes will stored in the commit
```

Git add: To add changes to staging area.

## Staging Area / Index (contd.):

If we make any further change, we must add the change to the staging area again.

```
$ echo "foo2" >> test03
# The redirect operator to append at the end of the file
$ cat test03
# Shows the content of the file
$ git status
# test03 is showing both in red and green
# The first change is added to the staging area - hence green
# The second change is not added to the staging area - hence red

DON'T EXECUTE THE BELOW COMMANDS:
$ #git add .
$ #git status
# Every change is now in green
# Ready for commit
```

## Git Commit:

```
$ git commit -m "First Commit"
# Saves the changes in form a commit
$ git status
$ git log
# Shows all the commit history
```

**DON'T EXECUTE THE BELOW COMMANDS:**

Now, let's delete a file and make another commit:

```
$ #rm test03
$ #git status
$ #git add .
$ #git commit -m "test03 deleted"
$ #git status
$ #git log
```

## Git Push:

Check if commits are added to the central repo:

```
$ cd ~/repo.git
$ git log
```

Commits are not showing in central repo. We have to push our commits from local repo:

```
$ cd ~/repo01
$ git push
```

Now, check if commits are added to the central repo:

```
$ cd ~/repo.git
$ git log
# Commits are added
$ ls -la
# Confirm no working tree
$ git status

# Error: As it doesn't contain a working tree
```

## Git Remote:

How do GIT know where to add the commit? There is a concept of remote, which stores information that where should the repository push and pull form. When we clone a repository, the remote is automatically set to the source repository.

```
$ cd ~/repo01
$ git remote -v
# Check remote
# It pushes or pulls from this repository
# As we cloned from here, remote is automatically set
```

## Git Clone - 2nd repository:

Now, let's clone another local repository on your system - representing one of your friends:

```
$ cd ~
$ git clone repo.git repo02
# Without --bare, it creates a standard repo
$ cd repo02
$ git log
# It has all the existing commits
$ ls -la
# Working tree and .git folder present - standard repository
# Note it generates all files from the commits, even from bare repository. Explain how.
```

There is a concept of **HEAD pointer**, which by default points to the last commit of the branch.

We can move the HEAD pointer to any of the previous commits. Moving the HEAD pointer changes the content of the working tree as per the content as it was during that commit.

## Git Commit - 2nd repository:

Now, let's make some changes in repo02, add a commit and push:

```
$ cd ~/repo02
$ echo "Hey hey" > test01
$ echo "Bye bye" > test02
$ git status
$ git add .
$ git status
$ git commit -m "Some changes"
$ git status
$ git log
$ git push
# push the commit to the centralised repository.
$ git remote -v
# Check remote
```

## Git Pull:

Now, let's fetch the new commits from the centralised repository to local repository (repo01).

```
$ cd ~/repo01
$ git pull
# clone only once, use push and pull to sync after that
$ git log
$ cat test01
$ cat test02
$ cat test03
```

**Git Pull:**

Let's make one more commit in repo01.

```
$ cd ~/repo01
$ echo "Hello World" >> test01
$ cat test01
$ rm test02
$ git add .
$ git commit -m "A change"
$ git log
$ git push
```

Now let's fetch the new commits from the centralised repository to repo02.

```
$ cd ~/repo02
$ git pull
$ git log
```

**Gitk:**

Install gitk: sudo apt-get install gitk

Use gitk, instead of "git log"

Gitk is a more detailed visual representation of "git log"

Gitk also shows the **change information** in each commit.

**Discuss team projects next. Come to conflicts later, only after they have made a couple of commits individually.**

**Moving HEAD pointer:**

We can move the HEAD pointer to any of the existing commits. The working tree will change accordingly.

```
$ cd ~/repo02
$ git checkout <hash>
```

<hash> can be any of the following:

- Commit ID
- HEAD~2
- Branch name

Open gitk and fetch the commit id of the 2nd commit.

```
$ git checkout <2nd commit id>
$ git checkout HEAD~1
$ git checkout master
```

## Team Project (Mini Social Media website):

- Share the mobile view version of the images with the students.
- Define projects
  - The website (home.php) shows all the statuses by other people on the website in chronological order.
  - There should be a register and login option (button) at the top of the home page.
  - After the user logs in, when he comes to the home page, he can put up his own status at the top as well.
  - After the user log in, he can go to his dashboard where he can add his other personal information (edit profile), and see all his statuses at once (my profile).
- Create teams of 3 students in each team.
  - Person 1: Homepage
  - Person 2: Login + Registration
  - Person 3: Edit Profile + My Profile
- Create a folder named **social_media** inside /var/www/html for this project.

**Github for team project (execute):**

- Everyone create their github account.
- One person in each team will create a repository [ungineering_social_media] for their project on github.
- Add everyone in the team as the project collaborator.
- Everyone will clone a local copy of this centralised repo in your home folder. This is the local standard repository. We won't work in this folder directly. This folder is only for GIT related commands.
- Create a folder named social_media inside /var/www/html for this project. We will work in this folder.

**Assignments:**

- Start creating web pages.
- Start using GIT from now on. No more file sharing personally. Make as many commits as possible.