Git Basics by Suteerth Subramaniam

- Central Repository
- Git Basics Assignment Repository

What is git?

- System to record changes to the files over time.
- Recall specific versions of those files at any given time.
- Makes it easy for many people to collaborate on a project and allows them to have their own versions of the project on their computer.

Why use git?

- Store revisions of a project in one directory with revision histories.
- Rewind to any version at any given time.
- Work on features without messing up the main codebase which is also referred to as *branching*.
- Easily collaborate with other programmers.
- ***Easily sync up to github*** where we can host the projects.
  - Share code with other developers.
  - Devs can download the projects and work on them on their own machines.
  - They can re-upload the edits and merge them with the main codebase.

Some CLI Stuff

How to check whether github is installed and added to the `$PATH`

```
git --version
```

How to let git know who's making the changes? (Adding the username in a global setting)

```
git config --global user.name username
git config --global user.email email
```

How to check what's there for us?

```
git config user.name
git config user.email
```

Repositories

- These are at the heart of git and are containers for any project you wish to track with git.

- Can have multiple repositories for many different projects on your computers.
- The `.git` file is what's responsible for letting git know that the folder is a repository located at the root level.
  - Git will then track changes to all files inside the folder and the sub-folders as well.

Commits

- These are like save points in a video game.
- These allow you to rollback the code to a particular commit.
- There are 3 different stages:
  - **Modified**
    - Changed files
    - Not committed
  - **Staging**
    - Add any changed files to staging that you want to commit
  - **Committed**
    - Any files in the staging area are added to the commit when we make one

Some Commands

1. How to create a repository?

   ```
   git init
   ```

   - This initializes a repository in the folder over the master branch.
2. How to check files in the staging area?

   ```
   git status
   ```

3. You can add files to the staging area like so:

   ```
   git add <filename>,<filename>...
   ```

4. How to remove the files from the staging area?

   ```
   git rm --cached <filename>
   ```

5. Committing Files

   ```
   git commit -m "Some descriptive commit"
   ```

6. Checking commit history (either full details or just the short id with the commit messages)

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

Undoing Stuff

There are some commits to the master branch and now we have to rewind stuff. This can be done in three different ways and are listed in the order of danger:

- **Checkout Commit** (Very safe and doesn't edit anything or ruin your commit history)
  - *Just see what the code was like at that point in time. Any changes made while checking out the commit will not be saved. It's read only and you cannot alter the commit history.*
- **Revert Commit** (Still pretty safe)
  - *Undo a particular commit. Reverts the changes from the commit in question, almost like deleting it out of the commit change.*
- **Reset Commit** (Unsafe, you have to be sure if you want to use this as it can ruin your repository)
  - *Takes you back in time and deletes all commits ahead of the commit in question and brings everything to the target commit's state.*

1. Using `checkout`

   ```
   git checkout <id>
   ```

   - You're basically seeing the code in the state it was in at that commit.
   - Now to go back into normal mode

   ```
   git checkout master
   ```

2. Undo one particular commit

   ```
   git revert <id>
   ```

   - Now what this does is create a new commit which reverts all the changes we made in a previous commit.

3. Danger Danger Danger

   - Just take the code from a target commit and re-do everything from there onwards.

     ```
     git reset <id of commit we want to reset to>
     ```

   - When you do this, the changes you made after the commit still hang around (just in case). You can still add the files and undo this by re-committing.

   - Now if you don't want to do this

     ```
     git reset <id> --hard
     ```

Branches

- Massive part of git and one of the more important features of git as well.
- Generally, the master branch represents the stable version of your code. The code which is released or published.
- Thus, it is better to not try out stuff on the master branch.
- Create an isolated environment to try out that feature and then later on merge it with the master branch.
- Branching can be used to assign teams working on different features to work in their own isolated environments.

Commands related to branches in git

- Creating a branch

  ```
  git branch feature-one
  ```

- List out all branches

  ```
  git branch -a
  ```

- Change branch

  ```
  git checkout feature-one
  ```

- Delete branch

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

  - A lower case `-d` will only work if it is merged
- Shortcuts
  - Switch to a branch after creating it

    ```
    git checkout -b new-branch
    ```

Merging

- How to merge code from one feature to master?

  ```
  git merge feature-one
  ```

- This merges in a `fast-forward` manner since there were no changes to the master branch before merging.
- Now suppose the master branch changes a bit before we have to merge some other branch. In this case, the merging is done using a recursive strategy.

Conflicts

- Say the code is working perfectly and a new feature has to be added.

- Now let's say some people decided to make changes to the master branch directly.

IDFC Neev Assignment - Git Basics

SSH Setup

- Check out version

  ```
  git --version
  ```

- Setup default branch as `main`

  ```
  git config --global init.defaultBranch main
  ```

- Setup SSH
  - Create keys using ed25519 algorithm.

    ```
    ssh-keygen -t ed25519 -C <email>
    ```

  - Start agent

    ```
    eval "$(ssh-agent -s)"
    ```

  - Check for the config file and create if necessary, do the configuration.

    ```
    open ~/.ssh/config
    touch ~/.ssh/config
    ```

  - Add pvt. SSH key to the `ssh-agent`

    ```
    ssh-add ~/.ssh/id_ed25519
    ```

  - Copy the public key of the key generated for SSH to clipboard.

    ```
    pbcopy < ~/.ssh/id_ed25519.pub
    ```

  - Connect SSH

    ```
    ssh -T git@github.com
    ```

Repo Work

- After creating a repository on github and setting up SSH, the next step is to clone it on a local machine. Repository can be accessed from Root_of_Repository

  ```
  git clone git@github.com:iamsuteerth/idfc-neev-pt.git
  ```

- Working in the repository and doing basic commands.

  ```
  cd idfc-neev-pt
  git remote -v
  ```

```
mkdir odin-stuff
cd odin-stuff
touch hello_world.txt
git add hello_world.txt
git status
git commit -m "Added hello_world.txt"
git status
git log
```

- Some more commands related to file modification. (Assuming you are in the odin directory)

```
code ../README.md
code ./hello_world.txt
cd ..
git add .
git commit -m "Modified README, added documentation.md and modified
hw.txt"
git status
git log --oneline
git push origin main
```

Some Additional Information

- The basic Git syntax is `program | action | destination`
- `git add .` is read as `git | add | .`, where the period represents everything in the current directory.
- `git commit -m "message"` is read as `git | commit -m | "message"`
- Some best practices:
  - Make atomic commits which means that your commits should be for one feature at a time.
  - Leave useful and insightful commit messages and avoid things like `fixed something broken`
  - Since we are normal people, we use VSCode instead of Vim for editing commit messages. It can be done by:

    ```
    git config --global core.editor "code --wait"
    ```