

## Setting Up a New Repository on GitHub:

### 1. Create a GitHub account (if you haven't already):

- Go to [GitHub](#) and sign up for an account.

### 2. Login to GitHub:

- Once you've registered, sign in to your account.

### 3. Create a new repository:

- Click on the '+' icon on the top right corner next to your profile picture.
- Choose "New repository" from the dropdown.
- Fill out the repository details:
  - **Repository name:** Choose a name that reflects the purpose of your project.
  - **Description** (optional): Provide a brief description of your project.
  - **Visibility:** Choose either "Public" (anyone can see it) or "Private" (only you and collaborators you invite can see it).
- Initialize the repository with:
  - A README (recommended as it provides an introduction to your project).
  - A `.gitignore` file (if you know you'll need one).
  - A license (optional but recommended for open source projects).
- Click "Create repository".

### 4. Clone the repository locally (optional):

- Once the repository is created, you can clone it to your local machine to work on it.
- Click the "Code" button on your repository's main page and copy the URL shown there.
- On your local machine, open a terminal or command prompt, navigate to the directory where you want to clone your repository, and run:

```
bash Copy code  
  
git clone [URL_YOU_COPIED]
```

For example:

```
bash Copy code  
  
git clone https://github.com/your-username/your-repository-name.git
```

### 5. Add content to your repository:

- Once you've cloned the repository locally, you can start adding content, such as code, documentation, images, etc.
- Navigate to your repository's directory in your terminal or command prompt.


```
bash Copy code  
  
cd your-repository-name
```

- Create/edit files as needed.

## 6. Commit and push your changes:

- After adding or editing files, stage the changes:


csharp

 Copy code

```
git add .
```

- Commit the staged changes:


sql

 Copy code

```
git commit -m "Your commit message here"
```

- Push the committed changes to the remote repository on GitHub:

css

 Copy code

```
git push origin main
```

Note: Use `main` or `master` based on the default branch name of your repository.

## 7. Repeat steps 5 and 6 as you continue to make changes and updates to your project.

Remember, GitHub provides a visual interface for many git actions. This means you can create new files, edit files, and manage some repository settings directly on the GitHub website. However, for more extensive development work, cloning the repository and working locally (then pushing changes) is often more efficient.

## 2. Here's a list of frequently used Git commands along with brief explanations:

### 1. `git init`

- **Use:** Initializes a new Git repository and starts tracking an existing directory.
- **Why:** It adds a hidden subfolder within the existing directory that houses the internal data structure required for version control.

### 2. `git clone [url]`

- **Use:** Clones (or copies) a repository from an existing URL.
- **Why:** Provides a way to download the content and history of a remote repository to your local machine.

### 3. `git status`

- **Use:** Shows the status of changes in the current directory.
- **Why:** It's crucial to see what files have been modified and/or staged before committing them.

### 4. `git add [file-name]` or `git add .`

- **Use:** Adds changes in the specified file or directory to the staging area. The `.` adds all changes.
- **Why:** It stages the changes you want to include in the next commit.

### 5. `git commit -m "[commit message]"`

- **Use:** Commits the staged changes along with a short message describing the changes.
- **Why:** Saves your staged changes along with a descriptive message to the repository history.

### 6. `git push [remote] [branch-name]`

- **Use:** Pushes committed changes to a remote repository.
- **Why:** Updates the remote repository with the committed changes.

### 7. `git pull [remote] [branch-name]`

- **Use:** Fetches and merges changes from a remote repository into the current branch.
- **Why:** Ensures that you have the latest updates from the remote repository.

### 8. `git branch`

- **Use:** Lists all local branches in the current repository.
- **Why:** Helps manage and view available branches.

### 9. `git checkout [branch-name]`

- **Use:** Switches to the specified branch.
- **Why:** Allows you to navigate between different branches.

### 10. `git merge [branch-name]`

- **Use:** Merges the specified branch's history into the current branch.
- **Why:** Helps integrate changes from one branch into another.

#### 1. `git log`

- **Use:** Shows the commit logs.
- **Why:** Provides a detailed history of the repository.

#### 1. `git remote -v`

- **Use:** Shows a list of remote repositories connected to your local repository.
- **Why:** Helps in managing and tracking different remote repositories.

#### 1. `git fetch [remote]`

- **Use:** Fetches changes from a remote repository but doesn't merge them.
- **Why:** Allows you to see changes in the remote repository before integrating them into your local branch.

#### 1. `git diff`

- **Use:** Shows differences between the working directory and the index (staged changes).
- **Why:** Allows you to see what changes have been made before you decide to stage or commit them.

#### 1. `git reset`

- **Use:** Resets the staging area to match the most recent commit but does not discard changes in the working directory.
- **Why:** Useful when you want to unstage changes.

Understanding these commands and their uses is crucial when working with Git. As you become more familiar with Git, you'll find that there are many more commands and options, but these are some of the most frequently used commands that form the core of many Git operations.

# How to do group project in GitHub?

Using GitHub for a group project is a great way to collaborate on code, documents, or any files in a version-controlled environment. Here's a step-by-step guide on setting it up for group projects and adding your teachers (q\_tong@uncg.edu) as collaborators:

## 1. Setting Up the Repository:

### 1. Create the Repository:

- Only one group member needs to create the repository. They will be the initial "owner" of the repository.
- After logging into GitHub, click on the '+' icon at the top right, then "New repository". Choose a suitable name and description for your group project.

### 2. Set Repository Visibility:

- If the project is sensitive or for grading purposes, you might want to set it as "Private". Only collaborators can access private repositories.

### 3. Initialize the Repository:

- It's a good idea to initialize the repo with a README to provide an introduction or overview of the project.

## 2. Add Collaborators (Group Members and Teachers):

### 1. Access Settings:

- Go to your repository page.
- Click on the "Settings" tab.

### 2. Add Collaborators:

- In the left sidebar, click on "Manage access".
- Click on "Invite teams or people".
- Search for your group members' and teacher's GitHub usernames or associated email addresses and add them.

### 3. Assign Roles:

- By default, collaborators have "write" access, which allows them to push to the repository, create branches, etc.
- If you want to restrict what collaborators can do, consider using more granular permission levels or protected branches.

### 3. Collaborate on the Project:

#### 1. Clone the Repository:

- Every group member should clone the repository to their local machine:

```
bash Copy code  
  
git clone https://github.com/owner-username/repository-name.git
```

#### 2. Create Branches:

- To avoid conflicts and make collaboration smoother, consider using branches. Each member or feature could have its own branch:

```
css Copy code  
  
git checkout -b feature-name
```

#### 3. Push Changes:

- After making changes locally, commit and push them:

```
sql Copy code  
  
git add .  
git commit -m "Description of changes"  
git push origin branch-name
```

#### 4. Pull Requests:

- When you're ready to merge your branch into the main branch (often called "main" or "master"), create a pull request on GitHub. This allows team members to review changes before they're merged.
- Group members can comment on the pull request, suggesting changes or approving them.

#### 5. Syncing and Updating:

- Regularly pull changes from the main repository to ensure you're working with the latest version:

```
css Copy code  
  
git pull origin main
```

## 4. Communicate and Manage:

### 1. **Use Issues:**

- GitHub Issues are a way to track tasks, enhancements, and bugs. They're a great tool for team communication and project management.
- You can assign issues to specific group members, label them, and reference them in commit messages.

### 2. **Project Boards:**

- GitHub has a feature called "Projects" that allows you to create Kanban-like boards to manage tasks. It integrates with Issues, so you can drag and drop issues into columns (like "To Do", "In Progress", and "Done").

### 3. **Notifications:**

- Stay on top of what's happening in your project. You'll get notifications for the activities in your repo, like when someone comments on a pull request or issue.

### 4. **Document Everything:**

- Use the README and Wiki features of GitHub to document your project, decisions, and processes.

Remember, effective communication is key in group projects. Make sure to frequently communicate with your team members, use descriptive commit messages, and always pull the latest changes before starting new work.

# Brief intro of Command Prompt and Terminal

Both the Windows Command Prompt (often just referred to as "Command Prompt") and the macOS Terminal are command-line interfaces (CLIs) that allow users to interact with the system using textual commands. Here's a brief introduction to using each:

## Windows Command Prompt:

### 1. Accessing the Command Prompt:

- Click on the Windows Start menu and type "cmd" or "Command Prompt" into the search. Click on the Command Prompt app to open it.

### 2. Basic Commands:

- `dir`: Lists the contents of the current directory.
- `cd [directory name]`: Changes the directory. E.g., `cd Documents` changes to the Documents directory.
- `cd..`: Moves up one directory level.
- `cls`: Clears the screen.
- `copy [source] [destination]`: Copies files from source to destination.
- `move [source] [destination]`: Moves files or directories.
- `del [file]`: Deletes a file.
- `exit`: Closes the Command Prompt.

### 3. Paths:

- In Windows, directories (or folders) are separated using backslashes (`\`). For example: `C:\Users\Username\Documents`.

### 4. Tips:

- You can use the `TAB` key to auto-complete file and directory names.
- Use the `help [command]` command for more information on a specific command. For example: `help copy`.



## macOS Terminal:

### 1. Accessing the Terminal:

- You can find the Terminal app in the `Applications` folder under `Utilities`.  
Alternatively, use Spotlight (`Cmd + Space`) and type "Terminal" and press `Enter`.

### 2. Basic Commands:

- `ls`: Lists the contents of the current directory.
- `cd [directory name]`: Changes the directory. E.g., `cd Documents` switches to the Documents directory.
- `cd ..`: Moves up one directory level.
- `clear`: Clears the screen.
- `cp [source] [destination]`: Copies files from source to destination.
- `mv [source] [destination]`: Moves files or directories.
- `rm [file]`: Deletes a file.
- `exit`: Closes the Terminal.

### 3. Paths:

- In macOS, directories are separated using forward slashes (`/`). For example:  
`/Users/Username/Documents`.

### 4. Tips:

- Similar to the Windows Command Prompt, you can use the `TAB` key for auto-completion.
- The `man [command]` command provides a manual or help for a specific command. For instance: `man cp`.

For both Command Prompt and Terminal, remember to exercise caution, especially when using commands that can modify or delete files. Always ensure you know what a command will do before executing it.