

Using **Git & GitHub** in T&L and Assessment for Programming





Why Git / GitHub

It's content:

- It is widely used at industry level – students should learn it
- Students should learn how to code as a group – for real.

It's also methodology:

- You can ascertain every line of code is contributed by which student.
- You can see WHEN each line of code is entered.
- Through commit messages you can see the intent of each change.
- You can compare contribution levels.

Git in the Industry

2021 - "Other tools" important to developers

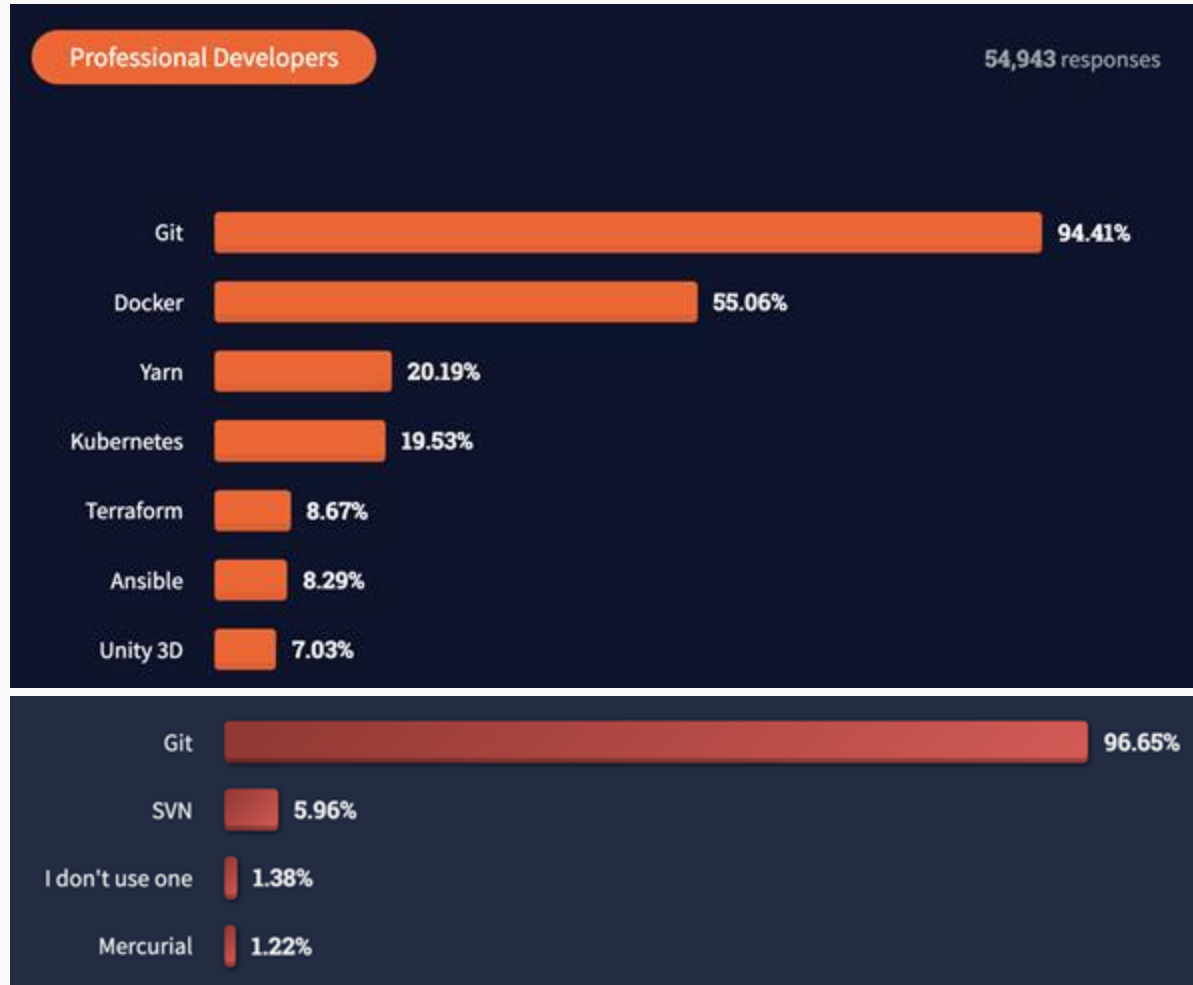
2022 - Comparing Git vs other version control.

2023 / 2024 - No longer a meaningful statistic to track. Usage of Git is a foregone conclusion

"If they're not using Git, don't work there."

Source:

<https://survey.stackoverflow.co/>



Git and GitHub



Using the Command Line

Windows:

Whenever you see "run this command" or "run it" means you should:

1. Open the program "**Command Prompt**" if you don't already have it open.
2. Type (or paste) the command, and press the **Enter** key.
3. E.g. Run the command **cls**

```
C:\Users>cls
```

Mac:

Whenever you see "run this command" or "run it" means you should:

1. Open the program "**Terminal**" if you don't already have it open.
2. Type (or paste) the command given, and press the **Enter** key.
3. E.g. Run the command **clear**

```
clear % clear
```

Command Line – Navigating Folders

Windows:

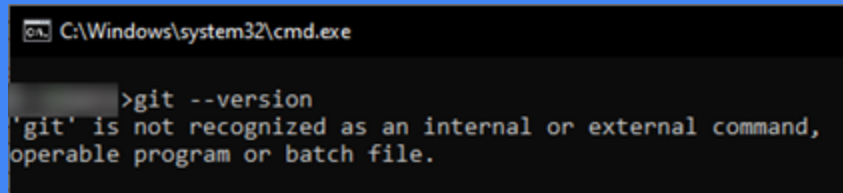
1. Your prompt always starts in your user folder. It shows `C:\Users\AccountName>`
2. Try running these commands:
`dir` – see what's in the current folder
`cd foldername` – go into foldername
`cd ..` – go up/out of current folder
`cls` – clear screen
3. You need at least these commands to 'move around' your computer and be at the right folder in your computer to start your repository.
4. You can type a file/folder name halfway and press the "Tab" key to autofill it.

Mac:

1. Your prompt always starts in your user folder. It (usually) shows `username@Macname ~ %`
2. Try running these commands:
`ls` – see what's in the current folder
`cd foldername` – go into foldername
`cd ..` – go up/out of current folder
`clear` – clear screen
3. You need at least these commands to 'move around' your computer and be at the right folder in your computer to start your repository.
4. You can type a file/folder name halfway and press the "Tab" key to autofill it (case sensitive!)

1. Installing Git

Check for/Upgrade Git



```
C:\Windows\system32\cmd.exe

>git --version
'git' is not recognized as an internal or external command,
operable program or batch file.
```

Windows

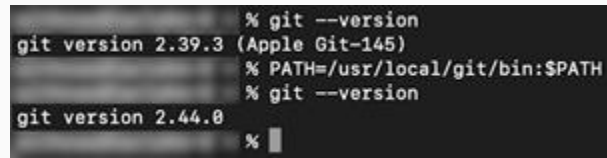
1. Before you attempt to install, check if you already have git!
2. Run the command `git --version`
3. If you don't have git, proceed to next slide.
4. If you see a version for git, update it instead (see Windows section):

<https://stackoverflow.com/questions/13790592/how-to-upgrade-git-on-windows-to-the-latest-version>

Mac

1. Before you attempt to install, check if you already have git!
2. Run the command `git --version`
3. If you don't have git, proceed to next slide.
4. If you see a version for git, and you installed it before using homebrew, run "brew upgrade git"

Note: Git does not come with Mac. However, if you see "Apple Git" you may have installed something else that comes with it. Install git (next slide). then come back and do the PATH command to switch to the latest version.



```
% git --version
git version 2.39.3 (Apple Git-145)
% PATH=/usr/local/git/bin:$PATH
% git --version
git version 2.44.0
%
```

Installing Git

Windows

1. Go to <https://git-scm.com/downloads>
2. Select Windows
3. Grab the Standalone Installer (probably 64-bit)
Note: If you're not sure what -bit your system is on, go to your PC's **Settings > Systems > About** and look for "System type"
4. There are many options screens. Make sure you set **"Choosing the default editor used by Git"** to use **Visual Studio Code**. (Step 4 or 5 in Next). Press "Next" and keep the rest of the defaults.

Mac

1. First we install homebrew. Run this command:

```
/bin/bash -c "$(curl -fsSL  
https://raw.githubusercontent.com/  
Homebrew/install/HEAD/install.sh)"
```

1. Then run `brew install git`
2. Then run `brew install --cask git-credential-manager`

Check the Install

Windows

1. Close and reopen your Command Prompt
2. Run `git --version`
3. You should see the latest version of git displayed.
4. If you don't, you either failed to install git, or your command line doesn't know where to find it.

Mac

1. Run the command `git --version`
2. You should see the latest version of git displayed.
3. If you don't, you either failed to install git, or your command line doesn't know where to find it.

Setup your identity

1. Run `git config --global user.name "Your Name"`
2. Run `git config --global user.email your@email.com`
3. Run `git config --global core.editor "code --wait"`
([Make sure you set up your Visual Studio Code to open when you run the command code.](#))

Windows should have this by default - you can test it by running the command `code`.)

- Use the same email that you will use to sign up for GitHub later.
- The `--global` setting makes sure that this works for all repositories you creating using your current Windows/macOS user account, so you only have to set this up once for all your projects.
- To see if your settings went in correctly, run `git config --list`

2. Using Git (Solo Mode)

Let's Start a Project

1. What's a project in coding? It's a bunch of text files containing your source code.
2. For this exercise, we're going to write a 'short line by line story' instead of actual coding so we can focus on git.
3. Create a New Folder somewhere on your computer. Git tracks a whole folder at a time, so you **MUST** start an empty new folder. I suggest:
`\Users\YourAccount\Projects\GitExercise\` (same for Mac/Win)
4. Make this folder using Windows Explorer / Finder (Mac) so that it's easier for you.
5. Inside this folder, create a new text file and name it `story.txt`.
Windows - you can just right-click empty folder and select New, Text Document.
Mac - use Visual Studio Code to make the file in the correct folder.
6. Then, go to the command prompt and use the `dir` or `ls` and `cd` commands to go to the folder you created earlier in your command line ([refer to Slide 4](#)). You should see **ONLY** `story.txt` in both your GUI and in your command line. Try running `dir` or `ls` in the command line to confirm that you're in the correct folder. If you see ALL your files, you're in the wrong location.

Filling story.txt

1. Fill `story.txt` with this story – use whatever text editor:

Adam wakes up in the morning.

He brushes his teeth.

He has breakfast.

Then he . . .

1. Save the text file.

Starting a Git Repository – `git init`

1. We now have a project folder with one text file inside.
2. Tell git to start tracking this project. Make sure your command prompt is IN the same folder as `story.txt`. (You did this two slides ago.)
3. Run the command `git init`

Behind the scenes – what did `git init` do?

Windows

1. Run `dir /a`
2. Run `dir`
3. Compare the two and you should see that the first one shows an extra hidden `.git` directory inside your folder.
 - Don't mess with this folder. Git automatically stores all the messy information needed to keep track of everything you do to this project. Whenever you run a git command, git is working with information in this folder.

Mac

1. Run `ls -a`
2. Run `ls`
3. Compare the two and you should see that the first one shows an extra hidden `.git` directory inside your folder.
 - Don't mess with this folder. Git automatically stores all the messy information needed to keep track of everything you do to this project. Whenever you run a git command, git is working with information in this folder.

Seeing your repo's status – `git status`

- Run `git status`
- You should see `story.txt` in red. It's NOT TRACKED and NOT STAGED.
- This is a useful command to see if you have any files you haven't tracked properly. (Don't do anything yet, just for your info/learning.)
- Use this command often.

Adding files – `git add`

1. So far we have a project folder with one text file inside, and we started a git repo that haven't started tracking anything.
2. Run `git add .`

Note: The `add` command usually works with a specific file name (e.g. `git add somefile.cpp`). When we're lazy, we tell it to add 'the whole current directory' which is what the `"."` or period means.

Seeing your repo's status – `git status`

- Run `git status`
- You should see `story.txt` in green. It's TRACKED and STAGED but NOT COMMITTED.
- Some terms:
 - Track** - The files with changes which Git is paying attention to. (You can tell git to never track certain files by [adding them to a .gitignore](#) file)
 - Stage** - The tracked files with changes which Git will commit when you run the commit command.
 - Commit** - "Saves" the all the tracked files into the repository. You can return your project to any commit you have made previously if you want to (like Google Docs version histories)

Commit your files – `git commit`

1. Run `git commit -m "Initialise repository."`
2. This is your first commit.
3. Run `git status` again and you'll see that your repository is clean - i.e. no new changes.

Note: **After the first commit, you should write your own commit messages** **after -m** explaining what you added/changed in this version. The double quotes is a must! A good habit is to prefix your commit messages: `fix: broken link in page 3` `fixed` or `feat: added button for clear all` or `chore: removed unused files` (feat stands for adding a new feature).

Another commit

1. Let's keep going. Add a new line at the bottom of `story.txt` with the line `He goes to class.` (Ignore the line with `'...'` first. **REMEMBER TO SAVE THE TEXT FILE.**)
2. Run `git status` again. Git knows there's changes. So we now need to `add` before we `commit` these changes. You should know how to do this based on what you've learned in the previous few slides.

See what changed – `git log` / `git show`

1. Run `git show`. This is the details of your latest commit. You can see which lines changed.
2. Run `git log`. This lists out a bunch of recent commits. (If there is more than one line, you can scroll up and down with the arrow keys and have to press 'q' to quit the view)
3. We seldom use these commands since viewing these things in a GUI like GitHub is usually prettier. But you may want to view something quickly once in awhile.

Renaming a branch – `git branch`

1. You may not realise, but you're right now working in a default branch called `master`. We're going to rename this to `main` based on modern conventions.
2. Run `git branch`. You can see the existing branch name.
3. Run `git branch -m main`
4. Run `git branch` again. You can see the result of the rename.

Branching – `git checkout` / `git branch`

1. We want to tell the story of what happens at class now. But just in case, we're going to branch out and make our changes in a separate branch from what we have committed so far.
2. Run `git checkout -b at-the-class` (Use `-b` only when making a new branch.)
3. Run `git branch`. You can see that there are two 'branches', one of which was `main` – your original branch.
4. In `story.txt`, add new lines **at the bottom** for things that happened at class. **SAVE** the file.
5. Commit these changes to your new branch. Do the two steps from earlier: `add` and `commit`
6. Check with `git status` to make sure that you're on branch `at-the-class` with a clean directory.
7. At this point, if you're using Notepad or TextEdit to edit your text file close it and open it in Visual Studio Code instead. (A better program shows you the contents of the file as it is changed, which we will demonstrate shortly.)

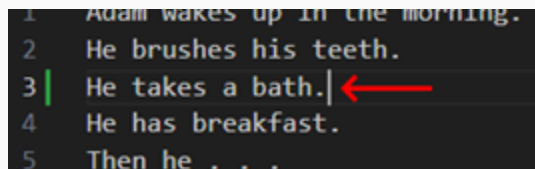
Branch Check



- Right now you have two branches.
- **at-the-class** have a few more lines at the bottom than **main**

Branching – `git checkout` (cont...)

1. Now, let's go back to the main branch – we forgot to add something earlier.
2. Run `git checkout main`
3. Notice what happens to your `story.txt` file that's open in Visual Studio Code. Your new lines at the bottom of the file have disappeared – they don't exist in the main branch.
4. Let's edit our story and add this new line. Then **SAVE** the file. You do bath in the morning, right?



```
1 Adam wakes up in the morning.  
2 He brushes his teeth.  
3 | He takes a bath. |  
4 He has breakfast.  
5 Then he . . .
```

A screenshot of a code editor with a dark background. It shows five lines of text. Line 3 is highlighted with a green cursor at the end. A red arrow points to the end of line 3, indicating where a new line should be added.

5. Add and commit your changes. These are committed to main since you're at this branch currently.

Branch Check



- Right now you have two branches.
- **at-the-class** have a few more lines at the bottom.
- **main** have an additional line at line 3 about bathing.


Discussion – Why Work in Branches?

- **You should always start a new branch when working on a new feature. Even when fixing a bug.**
But WHY?
- Situation 1 – "Ok I give up, I can't get this feature to work. Let's just build something else." It lets you **'undo' all your changes** with one magical command.
- Situation 2 – "Bro! The code you committed that we all pulled already – got error lah! Now it can't run." It lets you **'pause' your new feature, and go back to main to fix a bug for others**. Your fixes can be sent out to everyone else AND merged into your current feature branch.
- There are many other situations where it's good to isolate your code and keep bookmarks of where you are in your development.

Merging committed changes – `git merge`

1. We now need to do what we did in the `main` branch, back in our `at-the-class` branch. Go back to the `at-the-class` branch.
2. Run `git checkout at-the-class`
3. Run `git merge main`. This should open a text file in Visual Studio Code.

You can just close this text file.
(It gives an automatic commit message).



```
Users > althras > Projects > GitExercise > .git > MERGE_MSG
1 | Merge branch 'main' into at-the-class
2 | # Please enter a commit message to explain why this merge is necessary,
3 | # especially if it merges an updated upstream into a topic branch.
4 | #
5 | # Lines starting with '#' will be ignored, and an empty message aborts
6 | # the commit.
7 |
```

Branch Check



- Right now you have two branches.
- **at-the-class** have a few more lines at the bottom.
- **main**'s additional line about bathing have been merged into the branch.

Merging with conflicts – `git merge`

1. Now let's try something different. Edit `story.txt` and change `Then he . . .` to `Then he checked Instagram`. Save file, `git add` and commit this to the current branch (at-the-class).
2. Go back to the `main` branch, and edit the same line, but make it say `Then he watched YouTube` instead. (Don't forget to save, add and commit!)
3. Just like we merged changes in `main` into a branch, we can merge a branch into `main` as well.
4. Now that you're in the `main` branch, run `git merge at-the-class`.
5. You will see a merge conflict. Previously this did not occur because git can tell when a new line is added where there wasn't any. Now, git can't tell whether to keep Instagram or YouTube.
6. Git will open the file with the conflict (in this case it's `story.txt`), mark the places with the conflict, and ask you to edit the file until the file is what you want, before committing it. (See next slide)

Merging with conflicts – `git merge`

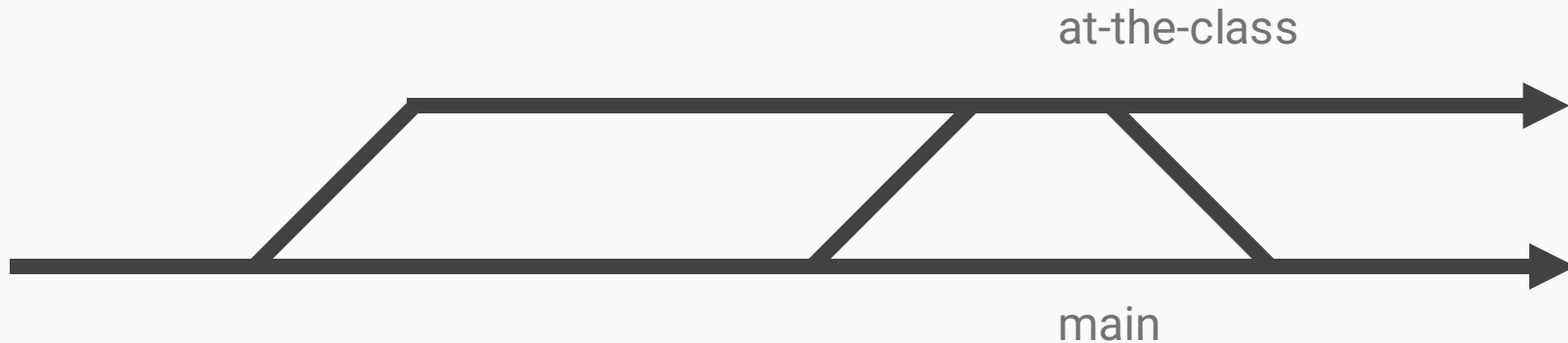
```
≡ story.txt
1 Adam wakes up in the morning.
2 He brushes his teeth.
3 He takes a bath.
4 He has breakfast.
5 Accept Current Change | Accept Incoming Change | Accept Merge
6 <<<<<< HEAD (Current Change)
7 Then he watched YouTube.
8 He goes to class.
9 =====
10 Then he checked Instagram.
11 He goes to class.
12 The lecturer was very boring.
13 He was confused with git.
14 >>>>>> at-the-class (Incoming Change)
```

Edit it to:

```
≡ story.txt
1 Adam wakes up in the morning.
2 He brushes his teeth.
3 He takes a bath.
4 He has breakfast.
5 Then he checked Instagram.
6 He goes to class.
7 The lecturer was very boring.
8 He was confused with git.
9
```

- Notice how Visual Studio Code understands git (if you open by folder) and highlights the <<< HEAD line in green and >>>> at-the-class line in blue, with some helpful shortcuts.
- Just clear the <<<, >>> and === lines and edit the text file so you get the 'correct' version.
- I'm going to go with Instagram. You can choose to keep YouTube if you want. **Save. Git add, then commit.**

Branch Check



- Right now you have two branches. Since you've merged everything from **at-the-class** back into **main**, the two branches should be more or less the same. (If you chose YouTube instead of Instagram, the next time you merge **main** into **at-the-class**, the conflict will auto-resolve since git now knows how.)

Delete branch – `git branch`

1. Now we no longer need the `at-the-class` branch.
2. Run `git branch -d at-the-class` to delete it.

Note: You cannot delete branches with unmerged changes accidentally with this command. You have to use `-D` instead, but be careful – make sure there's nothing left in there you might want to keep.

3. Using GitHub (Team Mode)

All Members – Setup GitHub

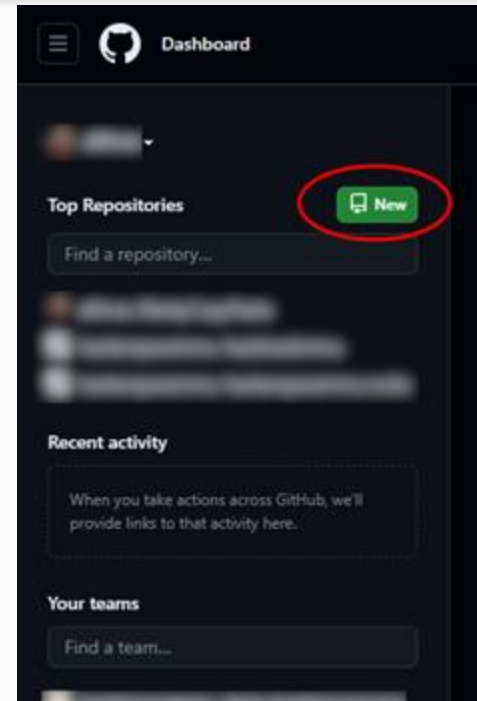
1. Go to <https://github.com/>
2. Hit Sign Up and fill in your details.
3. Verify using the verification code sent to your email.
4. Select "Just me" and "Student"
5. Check all the tools.
6. Select the "Free" option option and proceed.
(Student stuff can be applied later.)

GitHub hosts the repo that all your team members will contribute to. You will 'push' to this repo.



Team Leader – Create the remote repository

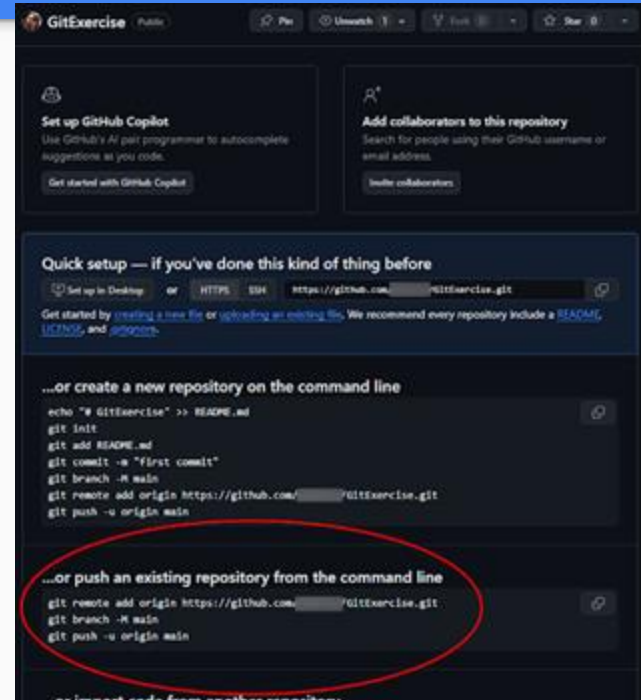
1. The repository on your computer is known as the 'local' repository. Now we need to make a 'remote' repository.
2. You **only need one** for your whole team, so choose a team leader to do this step. (All of you still need a GitHub account though!)
3. Click the "New" button. All you have to do on the next page is to give it a name. For this exercise, name it GitExercise-GroupName.
DO NOT add a Readme File.



Team Leader – Push Local to Remote Repo

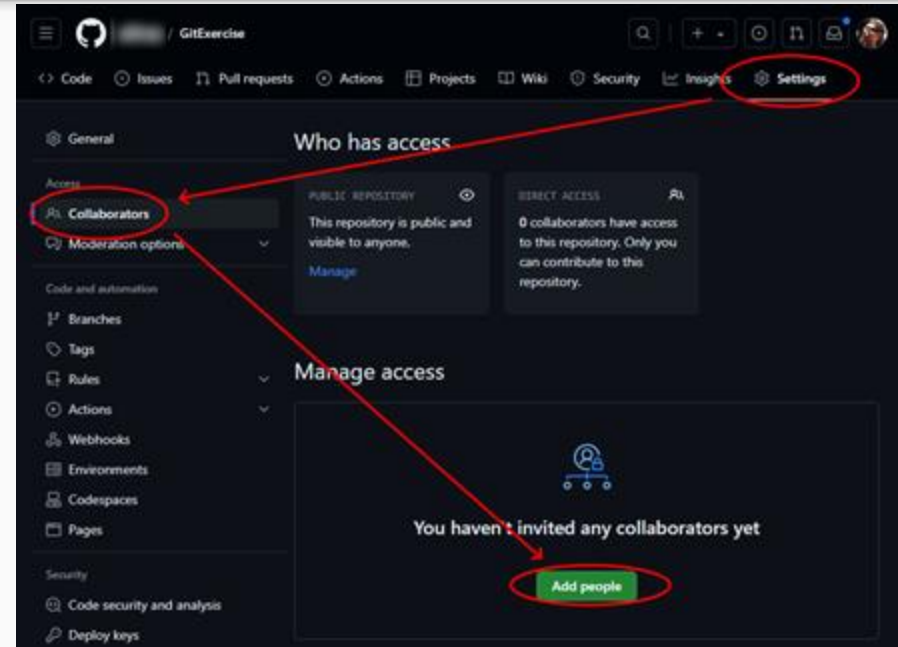
1. **Still team leader only.** On the next screen, you will get some instructions.
2. Go back to your command line, and run the `git remote add origin urlfromgithub` command given to you.
3. Skip the second command. (We already did it earlier coz we're cool.)
4. Then run `git push -u origin main`
5. You will be asked to sign into GitHub. Choose sign in with browser and authorise.

Notes: use `git remote -v` to see the remotes you set in case you put the wrong URL. Use `git remote set-url origin urlfromgithub` if you accidentally added the wrong URL the first time.



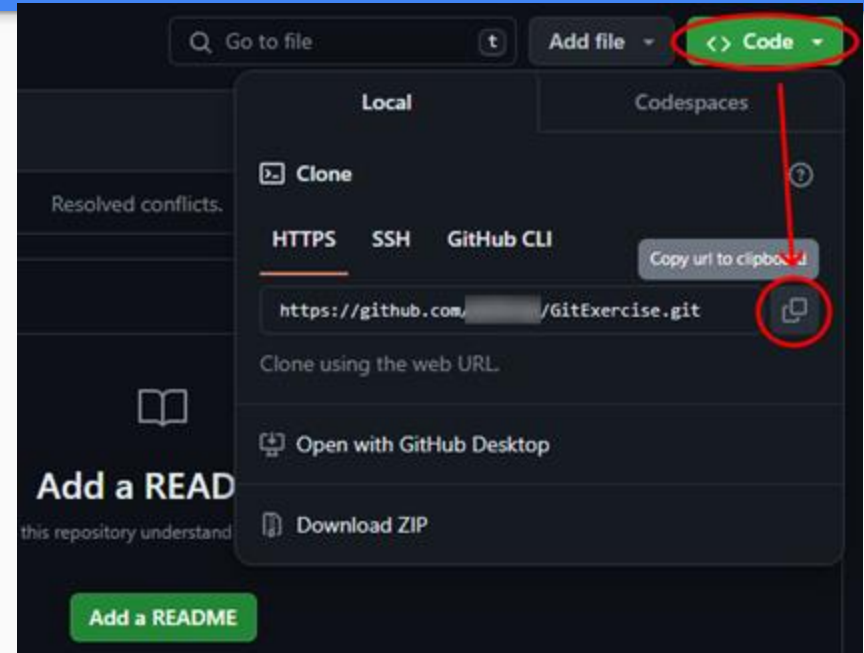
Team Leader – Add Your Members

1. **Team leader only.** In GitHub, add your team members! (See right screenshot)




Team Members - Clone the Repository

1. **Team members only.** Accept the invitation to the repository (check your email).
2. In the repository, copy the repository URL.
3. Then in your Projects folder (not GitExercise folder - this command will create a new folder for your project) run the command `git clone urlfromgithub` replacing urlfromgithub with the URL you copied.
4. After you clone, use `cd` command to enter the cloned project folder.
5. After this step, all team members will have local repos pointed at the same remote repo as 'origin'.
6. You're ready to branching, adding, committing and pushing! Try creating your own branch, making changes, committing and pushing your branch to GitHub (see next slide).



Working as a Team

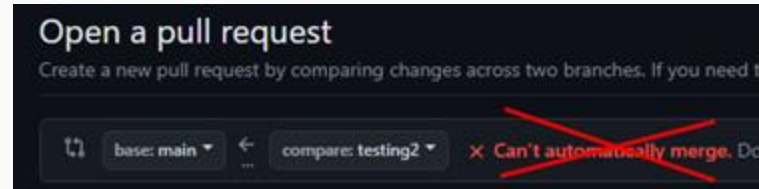
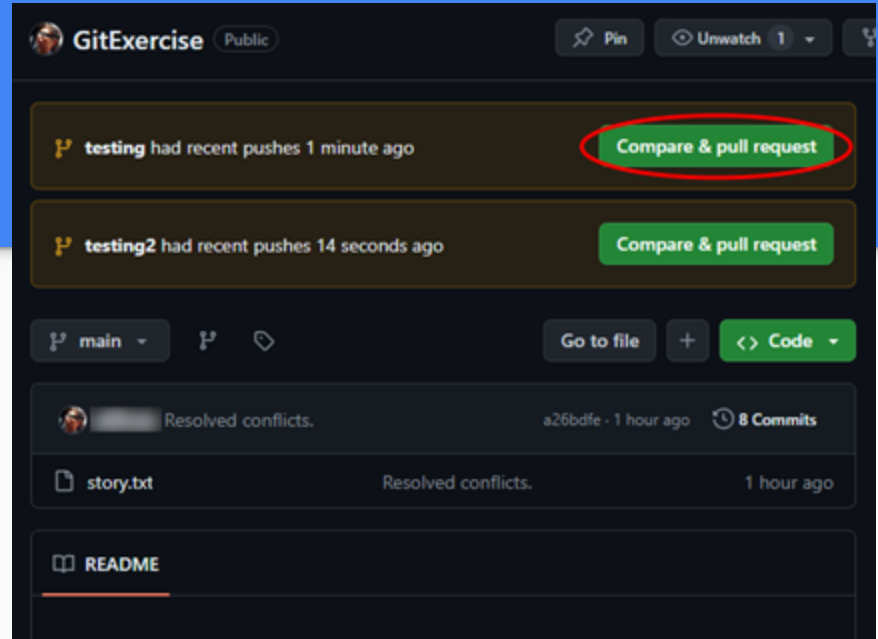
1. Use branches when you're working. If not you will overwrite each other in the `main` branch.
2. You can push your local branches to the repository. Make sure you're IN your branch, then use `git push -u origin branchname`
3. When you want to merge, never merge your branch to your own local `main` branch (this will clash with the origin later). Use pull requests (next slide).
4. To get the branches your friends pushed to GitHub locally in your machine:
`git fetch origin branchname`
`git checkout branchname`

1. To make sure your codebase is up to date with everyone else's, regularly merge changes in the remote main into your local repository:
2. `git checkout main`
3. `git pull origin main`  **new!**
If main has no updates, then skip next two steps
4. `git checkout branchname`
5. `git merge main`

Pull Requests

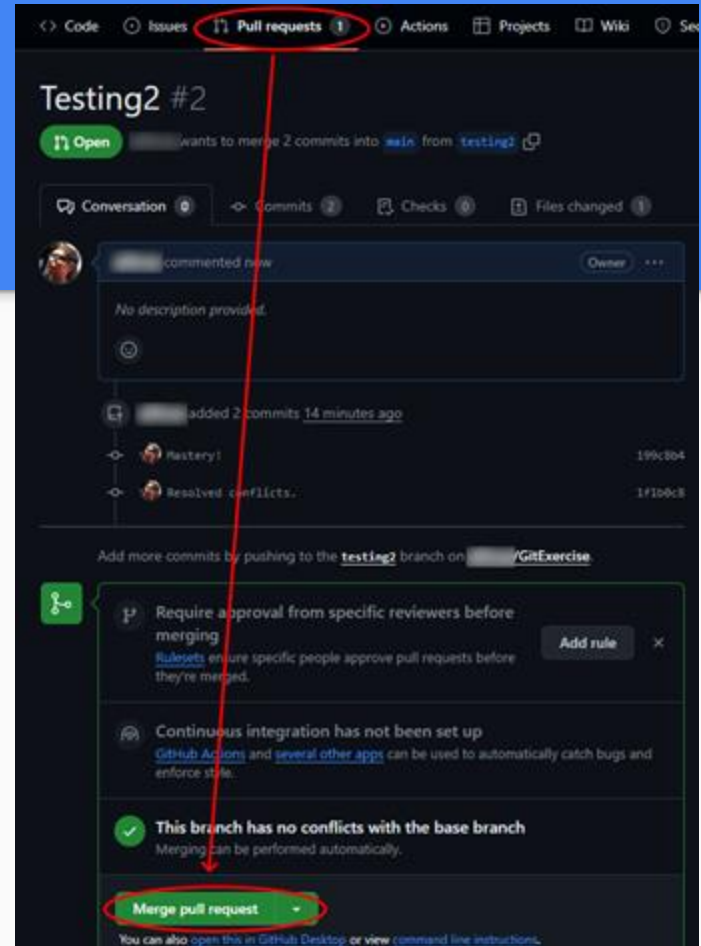
1. Once you have pushed some branches onto your origin, go to your GitHub repository page. You will see a button to make pull requests. A pull request is a feature of GitHub, not git. (It's actually a request to merge.)
2. Click the button on GitHub. Set a title and describe why you're requesting the merge, then click "Create pull request."

Note: A good practice is to never create pull-requests that can't automatically merge. Checkout to main, pull it from origin (GitHub), merge it into your branch and resolve any conflicts, push back your branch back to GitHub, then only create the pull-request.

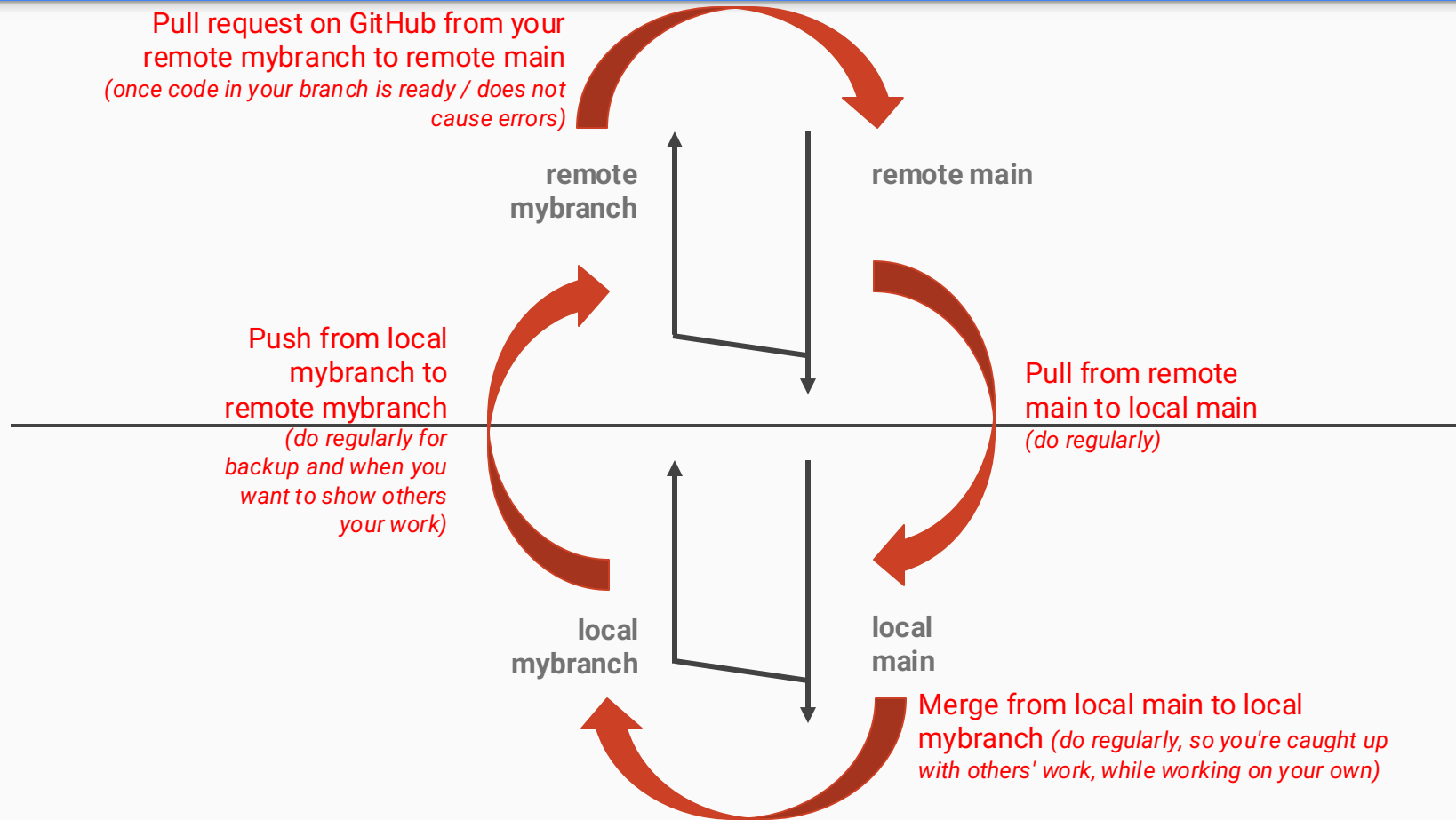


Resolving Pull Requests

1. A good practice is to never approve your own pull requests - have another team member review it to make sure the code works without errors before it is merged into main. Once it's in main, everyone else gets it in their local repo.
2. Remember that the pull request screen have social features (comments) you can use to discuss the pull request.



Merge/pull/push cycle – one-way street:



REMEMBER:
Commit often.
Merge often.

Now Go Finish the Story

- For the rest of the class, go write the story! Figure out who's supposed to write which parts, and practice branching, committing, pushing and pull-requesting your lines.
- Write in short lines to simulate coding.
- Use indentation (to simulate coding).
- Leave some empty space here and there to space the story out.

Remember - MERGE through pull requests on GitHub often. Do not wait weeks before attempting to merge - your codebases will become very different from each other, especially early on when your code is very short. Merge as often as you can.

4. Quick Reference/Cheat Sheet

Quick References (Local Only)

Starting (and going into) a new branch:

- `git checkout -b newbranchname`

Going into an existing branch:

- `git checkout branchname`

Committing code changes:

- `git add .`
- `git commit -m "Your commit description."`

Merging another branch into current branch:

- `git merge otherbranchname`

Deleting a branch once you're done with it:

- `git branch -d branchname`

Others:

- Seeing status `git status`
- See all branches `git branch`
- See last commit `git show`
- See commits `git log` (to quit press `q`)

Quick References (Remote)

Pulling from GitHub (origin)

- `git pull`

Daily routine to update branch with origin main

- `git checkout main`
- `git pull`
- `git checkout branchname`
- `git merge main`

Pushing change to others from inside a branch

- Make sure you're in the branch
- `git push -u origin branchname`

Pulling one branch from origin

- `git fetch origin branchname`
- `git checkout branchname`

Making a local copy of a GitHub repo

- `git clone repository-url`
(Command creates folder. Get URL from GitHub page)

Quick References (Additional)

Pulling everything from origin ([it's complicated](#))
– don't do this for big public open source projects!

- `git fetch --all` (get info of all branches)
- `git pull --all` (pull all the branches)
- `git checkout branchname`

Note: There is no rule that your local branch name must have the same name as a remote branch. You can pull a remote branch and track it using a local branch with a different name:
`git checkout -b local_branch_name
origin/remote_branch_name`

Get rid of changes you haven't staged (never did git add) that you don't want to commit

- `git restore .`

Note: Many special things you want to do, you may want to Google clearly before running the command to see if that is what you want. Some command versions delete unstaged files. Some command versions only change tracked files.

Restoring back to a previous commit. The answer [is complicated](#). Read carefully.

- `git reset --hard xxxxxx` or
- `git revert xxxxxx`

Git through Faster Command Line

Windows

1. Create a file with the name `gs.bat` in your project directory.
2. Inside the file put the lines:
`@ECHO OFF`
`git status`
3. Try running `gs` inside your project directory.
4. You should move this `gs.bat` file [somewhere else in your PC](#) as it doesn't belong to your project.
5. Create a new batch file for every command with the filename being your shortcut, and the command inside being the command u want to run. [More details here](#).

Mac

1. Open Terminal
2. Run `code .zshrc`
(You should get a file called `.zshrc` open in Visual Studio Code)
3. Add `alias gs="git status"` in one line
4. Save the file
5. Close and re-open Terminal
6. Go to your git folder in Terminal, and type `gs`. See what happens.
7. [More reference for other shortcuts you can make.](#)

Git through GUI

- You can use Git from within Visual Studio Code itself. Even without an extension, git recognises line additions, deletions and changes when it sees a Git repository.
- Install an extension for more commands. GitLens extension is very popular.
- Press Ctrl-Shift-P (or Cmd-Shift-P), type "Install Extensions", then search for GitLens. Learn to use it yourself.
- You can install a GUI execute Git commands.
- SourceTree's app is free and very powerful/beautiful.
<https://www.sourcetreeapp.com/>
- GUIs sometimes cannot be as specific as command lines however, so if you need to do something specific, sometimes you can Google up specific commands to execute. (Like `git rebase` for example - what does that do?)