

Week 2 Studio 2 – Version Control [Graded]

This lab report is **graded**:

- There are questions scattered in this handout that you are supposed to answer in your lab report. Each question will be clearly labelled.
- Submit a softcopy "Student_ID.docx / .pdf" into your studio group's IVLE workbin ("IVLE Workbin" → "Student Submission" → "Week 2 – Studio 2" → "Studio Group "). The workbin will close 15 minutes after the studio ending time.

Core Objectives:

- C1. Understand and using version control for personal use
- C2. The Git Client variants
- C3. Remote Repository and GitHub
- C4. Using Git for team development



Preparation (Before the studio):

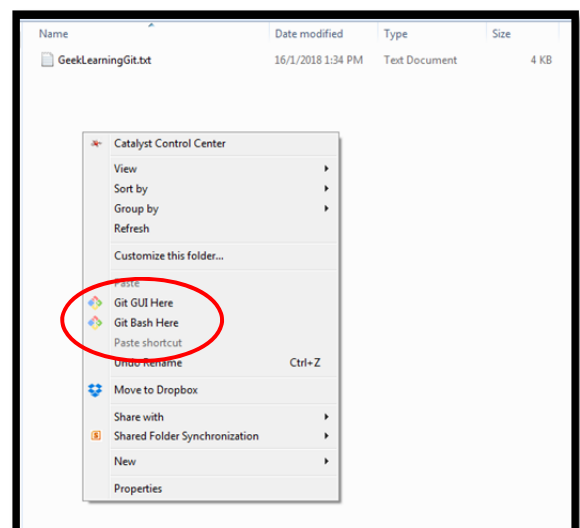
- Download and install Git on your laptop. The Windows and Mac OS versions can be found on IVLE workbin → Studio Materials → Week 02 – Studio 2. If you encounter any issue, check out the official website at <https://git-scm.com/>
- Download the "**Solo Folder.zip**" and "**Team Folder.zip**" from IVLE. Keep them in a convenient location.
- If you do not have a GitHub account, heads over to <https://github.com/> to create a new account.
- **Bring your laptop to the studio.**

Studio Setup:

- [C1-C3] Work on your own.
- [C4] Work as 2-member team (can work across section if needed). **At most 1 group with 3 members** for studio group with odd number of students.

C0. Verify Git installation

Take a moment to check the **Git** installation on your laptop. In windows explorer (Windows), right click in a folder, the pop up menu should include options "**Git GUI Here**" and "**Git Bash Here**":



What is Version Control?

Version control is a **software** that keeps track of changes we made to files, i.e. the different **versions** of a file. Common functionalities include ability to revert to any older version, highlight differences between versions, merging of versions etc.

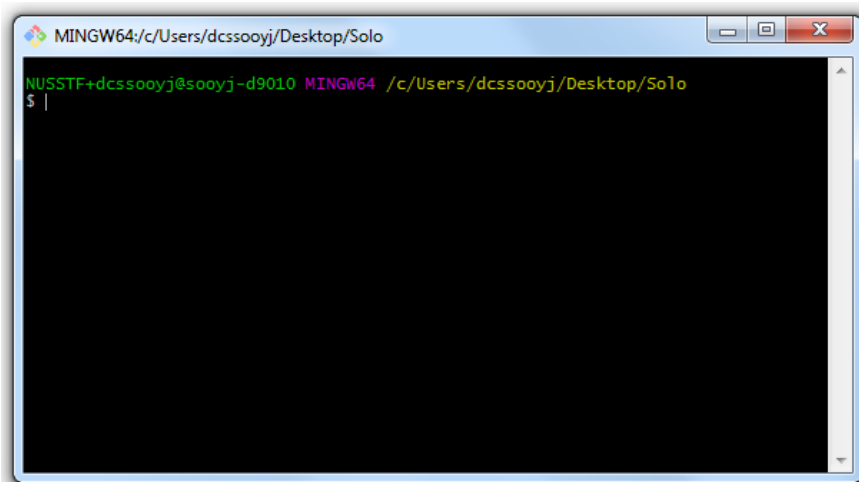
What is Git?

Git is a open-source and free implementation of version control software. It is currently the most widely used version control software in the market.

C1. Understand and using version control for personal use

Unzip the "**SoloFolder.zip**", rename it to "**Solo/**" and then use your favourite editor to take a look at "**GeekLearningGit.txt**".

1. Right click in the "**Solo**" folder and choose "Git Bash Here". You should see a terminal as follows:



This is actually a minimalistic implementation of **bash** (a popular shell interpreter for linux/unix)! It supports a simple set of unix commands like "**cp**", "**rm**" (even "**vim**"). We are using it mainly to issue **Git Commands**.

2. Enter **git status** into the bash windows. You should receive an error:

```
$ git status
fatal: Not a git repository (or any of the parent directories): .git
```

Learning Point:

Git needs to setup its own storage (known as **repository**) before it can start to keep track of the files. The repository should be placed at the highest (topmost) folder which contains the files and subfolders you want to keep track of.

3. Enter `git init`. This initializes the repository at this location. In the file browser, you can see that a new "hidden" folder `.git/` has been created. This folder contains information used by Git to perform the tracking and versioning.

4. Use `git status` and you should see a more interesting message now:

```
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        GeekLearningGit.txt

nothing added to commit but untracked files present (use "git add" to track)
```

Git warns you that there is a file ("**GeekLearningGit.txt**") that is not currently being tracked. Note that we need to instruct Git to start or stop tracking file/files as tracking are not automatically enabled. Once a file is "registered" for tracking, Git will faithfully monitor it from that point onwards.

5. Use `git add GeekLearningGit.txt` to start tracking the text file.

6. Use `git status` to find out what's happening:

```
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   GeekLearningGit.txt
```

7. We are now ready to record a version of this file by **committing** it. Once committed, a snapshot of the current state of all files will be captured in the repository. Enter `git commit -m "First verion!"`. The quoted text is a **commit message**, usually used to summarize the major changes of this particular version. (Just like all good practices, this is destined to be ignored by most ;-)).

You will see quite a bit of message due to the commit, don't panic (and ignore them!). Just ensure the commit is properly performed by checking `git status` which should reports "**nothing to commit, ...**"

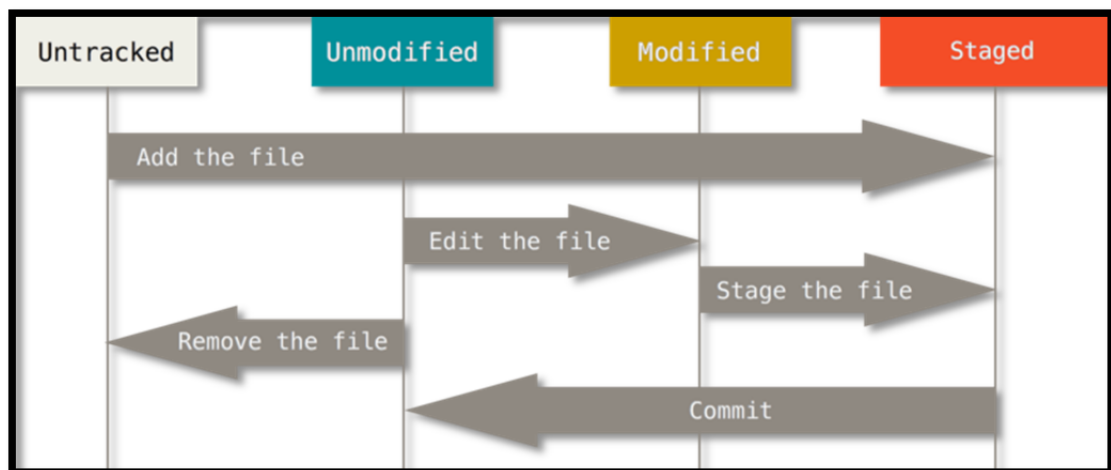
8. Let's modify the file "**GeekLearningGit.txt**". Use any text editor and perform the following changes:
 - a. Change the first line to "**Version 2.0**"
 - b. Remove all text below the "*****" line.

- c. Add a sentence below the "*****" line that reads "I don't want to know about loren ipsum!".
 - d. Save the file (in the text editor).
9. Run `git status`, Git should indicate that the changes have been detected. To find out more about the changes, you can use `git diff`.

Lab Report Question 1:

Take a screenshot of the `git diff` result. Explain the notation used by `git diff` in reporting changes. You only need to explain the notation involved in the first change (the one above "Version....").

10. It is probably natural to assume that we can now commit the new changes. But Git has a surprise for you. Try to commit using the commit message "V2.0: Much shorter now".
11. Apparently, Git follows the following workflow:



So, we need to **stage** the changes before it can be committed. Staging can be done in at least two ways:

- a. Use `git add` to stage a modified file. OR
- b. Use `git commit -a -m ...` the "-a" flag will stage and commit all files that were previously committed.

Lab Report Question 2:

Make an "educated guess" on why Git insists on you staging changes for the next commit. Since Git can easily detect the changes, why couldn't it just commit all changes? (hint: think in terms of programming source code).

12. Use one of the given methods and perform the commit with message "V2.0: Much shorter now". Use `git log` to show the current versions recorded.

Lab Report Question 3:

Take a screenshot of the `git log` result. You'll need it later.

13. With 2 versions in our repository, we can now revert to a stored version easily. Let us first try **reverting to the most recent version**, i.e. discarding all new changes.
- Edit the "GeekLearningGit.txt" by changing the first line to "Version 2.5".
 - Remove all other lines.
 - Save the file.

Suppose you have a change of heart at this point and want to get back the content in V2.0, you use `git checkout -- GeekLearningGit.txt`. The `checkout` command retrieves file(s) from a particular version. By default, checkout uses the most recent commit, i.e. version 2.0 in our running example.

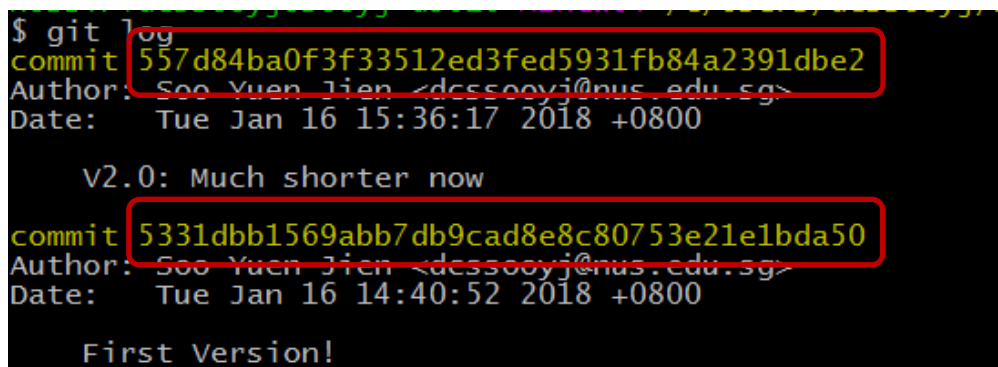
Verify the content of "GeekLearningGit.txt" has been reverted back to version 2.0

14. The checkout command can be used to check out any files from any version. So, let's try to get back our original i.e. version 1.0.

The command syntax is:

```
git checkout <Commit Hashcode> -- <filenames>
```

The <commit hashcode> refers to the long "random" characters associated with each commit. In other words, each commit is uniquely identified by the associated hashcode. Refer to the screenshot you took for `git log` to find out the hashcode associated with each commit. For example:



```
$ git log
commit 557d84ba0f3f33512ed3fed5931fb84a2391dbe2
Author: Soo Yuen Jien <dcssooyj@nus.edu.sg>
Date: Tue Jan 16 15:36:17 2018 +0800

    v2.0: Much shorter now

commit 5331dbb1569abb7db9cad8e8c80753e21e1bda50
Author: Soo Yuen Jien <dcssooyj@nus.edu.sg>
Date: Tue Jan 16 14:40:52 2018 +0800

    First Version!
```

Before you complain, please note that you only need to use the **first seven characters** in the hash code, e.g. "5331dbb" for the "First Version!" commit.

- a. Figure out the right command and retrieve the version 1.0 of the text file.
- b. Verify the content is correct.

Lab Report Question 3:

Give the command you used for (a).

Learning Point:

Checking out earlier version of files is essentially interpreted as "new change" to file. For example, `git status` will report "file changed" after V1.0 is checked out. So, further commit (e.g. the 3rd commit onwards) will record this V1.0 content.

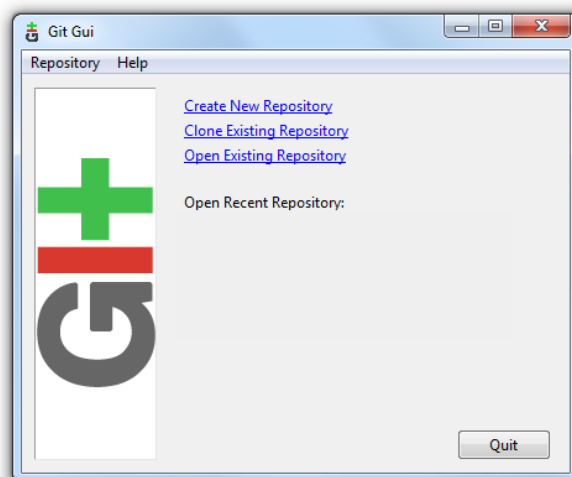
So, checking out older version of a file should not be understood as "undo-ing" a commit.

C2. The Git Client Variants

We hope you are having fun remembering all the git commands so far ;-). If you take a look at the git command reference, it is hard not to be daunted (see reference [2]). Just like many other popular open source software, Git has a large number of variants. Many of these variants focus on providing a more user friendly GUI based experience.

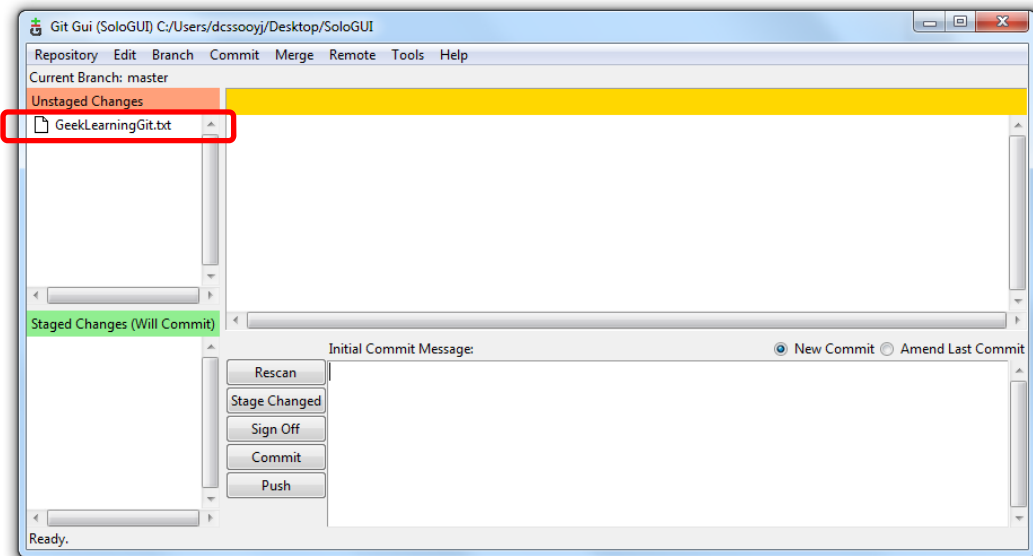
Let's explore the GUI variant came with the Git software for this **core objective**. We will retrace some of the steps used in **C1** so that you can easily relate the experience.

1. Unzip the "**SoloFolder.zip**" again and rename the folder "**SoloGUI**".
2. Right click in the "**SoloGUI**" folder and choose "Git GUI Here".
You should see this interface →



Click "**Create New Repository**" and select the "**SoloGUI**" folder. This initializes the repository (i.e. the same as `git init`).

3. The status of the files are shown automatically in the interface:



You can see clearly that "GeekLearningGit.txt" is listed under "**Unstaged Changes**". Clicking on the *file icon* will move it to the "**Staged Changes (Will Commit)**" category.

4. Enter "First Version!" in the "Initial Commit Message" text box, then click "Commit". This will perform the same functionality as **C1 – Step 7**.

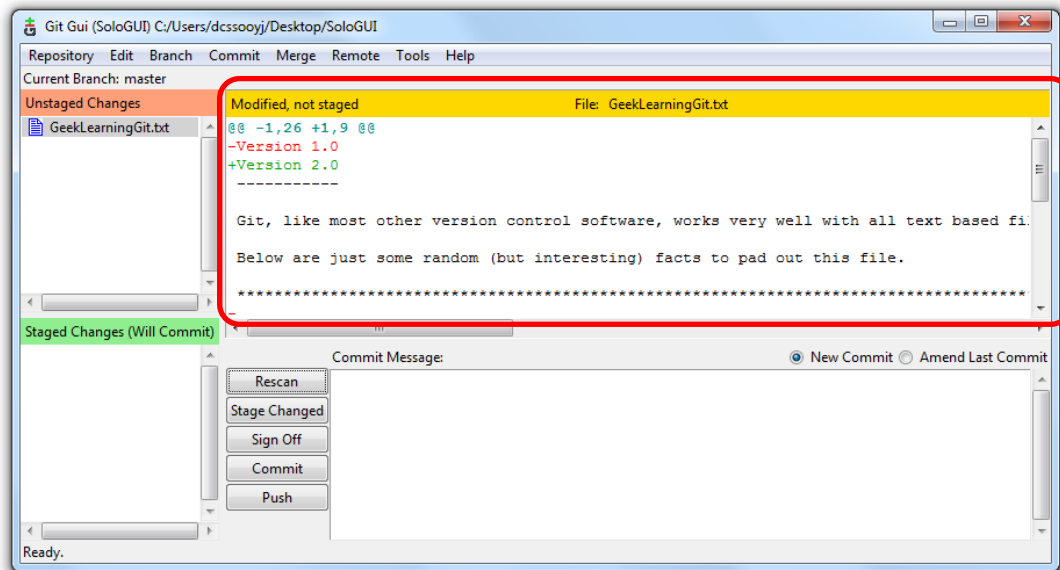
Learning Point:

Note that underneath the GUI, the same set of Git commands is used. So, you can switch from the GUI to the command line version (GUI Bash) and vice versa at any time.

There are times where the command line version is needed, e.g. when you need to perform more complicated operations that are not available in the GUI. So, don't flip table because we "forced" you to use command line in **C1** 😊.

5. Now, perform the same set of changes:
- Change the first line to "Version 2.0"
 - Remove all text below the "*****" line.
 - Add a sentence below the "*****" line that reads "I don't want to know about lorem ipsum!".
 - Save the file

Restart the Git GUI if you've closed it or click the "rescan" button to update the tracking status. You should see that "GeekLearningGit.txt" is now listed below "Unstaged Changes" again. Furthermore, the changes can be seen immediately in the panel on the right.



Go ahead to:

- a. Stage the change.
 - b. Commit the change with a message "Version 2.0: Much shorter!"
6. You can find out the versioning history (the equivalent of "git log") by clicking on the menu bar: Repository → visualize master's history.

Lab Report Question 4:

Take a snapshot of the visualized history.

7. Let us now try a couple of things that are more intuitive to do under a GUI. Follow the steps below:
- a. Add a new text file "**GitIsNotHard.txt**" under the "SoloGUI" folder. The content is not important.
 - b. Add a **subfolder** "**GitJoke**", then add two files "**funny.txt**" and "**notFunny.txt**" in the subfolder. Again, you can place some random contents in the files.

Figure out how to:

- Stage the files "**GitIsNotHard.txt**" and "**funny.txt**", **but NOT** the "**notFunny.txt**"
- Take a screenshot then commit with a message "Again, with more files!"

Lab Report Question 5:

Give the screenshot just before the commit.

Lab Report Question 6:

Although it is "obvious" in this case that we don't want to archive notFunny.txt, can you give 2 examples of file that you do want to track for a typical programming project? Briefly give the reason of excluding the file.

***** [For your own exploration] *****

8. From the above, you should see the needs to ignore certain type of files for Git. It is cumbersome to do it manually, especially if you have large number of subfolders / files to sift through. Git provides a simple solution: you can place a file **.gitignore** in the folder to specify file(s) to be automatically ignored by Git.
9. Before step 6, we have seen Git tracking a single file and capable of reverting to earlier version easily. However, this is just scratching the surface of Git capability. After step 6, you can see that multiple files (and the associated folder structures) can be kept as well. So, by reverting to an earlier commit, you get back not only the file content, but also the files and folders structure.

For example, if you revert to version 1.0 now, the "**GitIsNotHard.txt**" and "**GitJoke/funny.txt**" will disappear as they were not present at the time of commit.

10. Git GUI is functional, but not very attractive. It is really just a GUI wrapper for the Git commands. There are many more attractive (and more powerful) variants out there. Below are a few popular choices for your own exploration:
 - Source Tree (<https://www.sourcetreeapp.com/>)
 - Git Kraken (<https://www.gitkraken.com/>)
 - Many more (GIYF!)

C3. Remote Repository and GitHub

As mentioned, the git repository is simply kept as a hidden folder (`.git/` by default). If you have access to this folder → you can perform all the necessary Git operations. Now consider this scenario: What if you need to work on a project across a number of devices, e.g. the desktop PC at home, your laptop when you are on the bus, then the lab machine when you are in school?

If you answered "put the Git repository in Dropbox / Google Drive / Other network accessible locations", that's a very close answer. Git can operate **across network**, i.e. as long as the repository is accessible through network protocols (http, ssh etc), you can directly operate on the repositories with simple git commands.

Terminology:

A locally stored repository (what we have been using so far) is known as **local repository**. A repository stored on network locations is known as **remote repository**. Also, the term "repository" is commonly abbreviated to "**repo**".

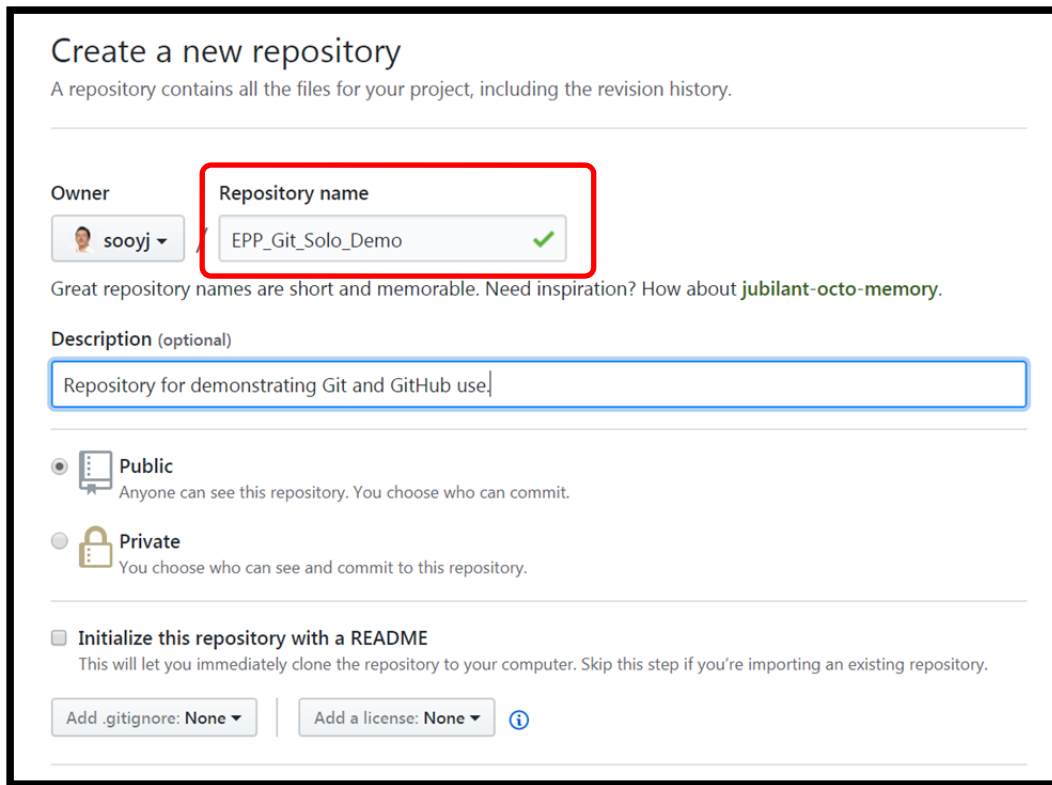
Learning Point:

This means that you can setup your own remote repository (e.g. on a server somewhere, on sunfire etc).

As you can imagine, there are plenty of demands for Git repositories hosting service on the internet. **GitHub** (<https://github.com/>) and **BitBucket** (<https://bitbucket.org/>) are two of the well known examples, where GitHub is the de facto leader in the field. You can find many **many** public code repositories on GitHub for you to contribute to / learn from. It is also a great showcase of your project work to the whole world. We have reached a point where reputable software companies prefer reading through your GitHub page to assess your strength as a developer rather than relying solely on your CV.

In this section, we are going to utilize GitHub for hosting our remote repositories. You are encouraged to continue strengthening your GitHub page after this studio, e.g. showcasing your cool pet projects, contributing to world famous open-source projects etc. The **software engineering** course, e.g. **CS2103/CS2040** also uses GitHub extensively.

1. Unzip the "**SoloFolder.zip**" yet again, and name the folder "**SoloGitHub**".
2. Log into your GitHub account. Click "New repository" (on the right side of the page) and create a sample repository. See the screenshot on next page.



Create a new repository

A repository contains all the files for your project, including the revision history.

Owner: sooyj

Repository name: EPP_Git_Solo_Demo ✓

Great repository names are short and memorable. Need inspiration? How about [jubilant-octo-memory](#).

Description (optional): Repository for demonstrating Git and GitHub use

☒ Public
Anyone can see this repository. You choose who can commit.

☐ Private
You choose who can see and commit to this repository.

☒ Initialize this repository with a README
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None | Add a license: None ⓘ

Click "Create Repository" to create your first remote repository on GitHub. The result page contains very useful summary of how you can proceed from here. We are going to use the second option " *...or push an existing repository from the command line*" in step 4 below.

Caution:

Repositories for a normal user account on GitHub are by default **public**, i.e. accessible to anyone. So, refrain from posting on-going school work (e.g. programming assignments, labs, projects) on public repositories. You can consider website that provides free **private repository**, e.g. **BitBucket** for such works.

3. Start "Git Bash Here" in the "SoloGitHub" folder, then:
 - a. Perform a git init.
 - b. Stage the "GeekLearningGit.txt" file
 - c. Commit with the message "First Version"

Up to this point, you are still working on the local repo (i.e. no difference from **C1**).

4. We will now "push" the commits in the local repo to our remote repo on GitHub:

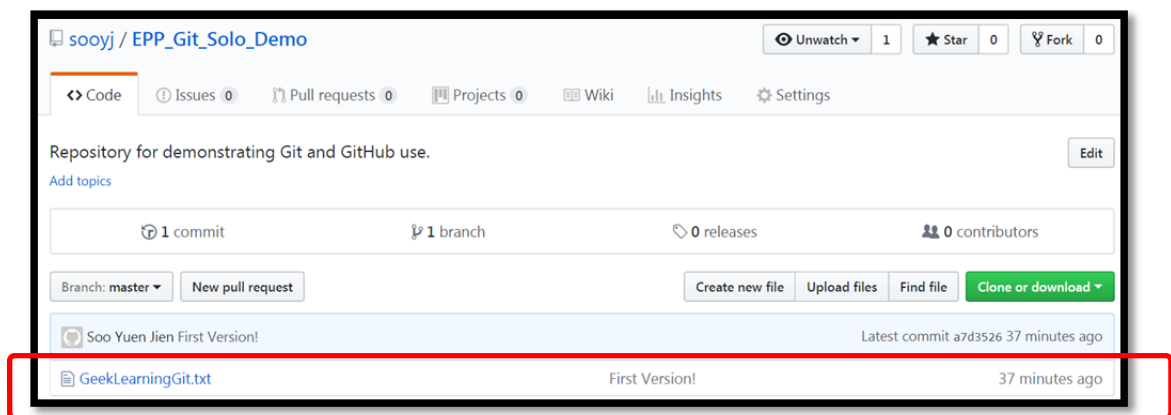
a. `git remote add origin <URL of remote Git Repo>`

Note: remote refers to remote repository, origin is the default name we use for remote repository and the last part is simply the URL for the remote repository.

b. `git push -u origin master`

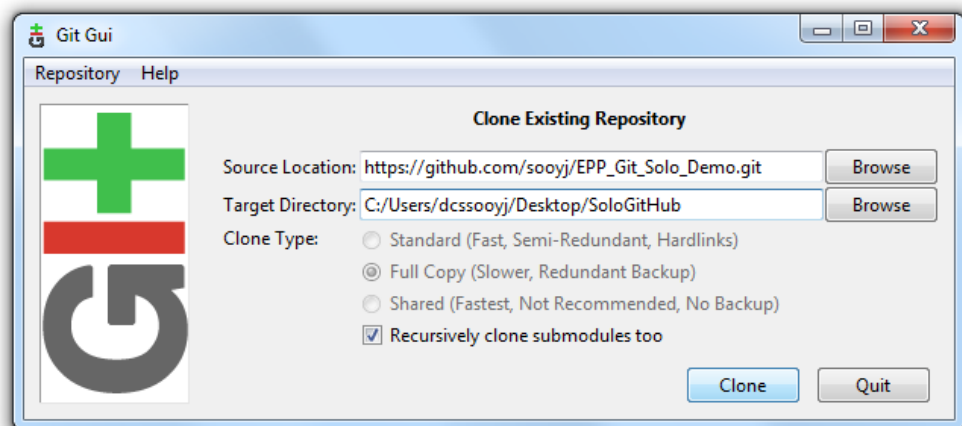
`push` means uploading the local repository to the remote repository.

5. Check your GitHub page, you should see the commit and the file "GeekLearningGit.txt":

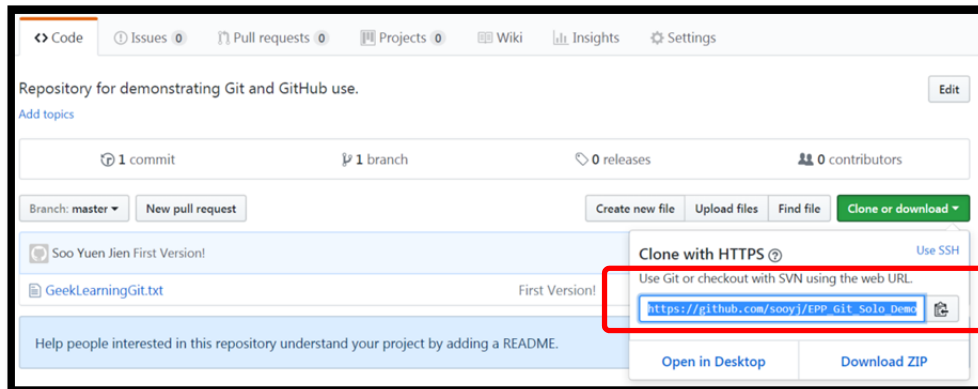


6. To show you the usefulness of a remote repo, let's do something drastic (read the following steps carefully!):

- a. **Delete** the folder "SoloGitHub/".
- b. Right click on your desktop (or wherever the "SoloGitHub/" used to reside) and start "Git GUI here", select **Clone Existing Repository**:



The **Source Location** refers to the URL of the remote Git repo. You can get this from earlier command back in step 4a, or you can get the link from GitHub page:



Note also that the **Target Directory** should not be an existing folder as Git will create the folder structure for you.

Learning Point:

This is also the general way for you to download a public repository on GitHub or similar places.

- c. Click "Clone" once you have filled in the fields properly. After a little bit of processing time, you should see the folder with the file "**GeekLearningGit.txt**" inside!

Learning Point:

Step b and c simulate the scenario where you want to work on existing repo on a new device. By "cloning" the repo, you will have the latest version of the repo in your new location.

7. Let's quickly redo the version 2.0 change as before:
 - a. Change the first line to "Version 2.0"
 - b. Remove all text below the "*****" line.
 - c. Add a sentence below the "*****" line that reads "I don't want to know about loren ipsum!".
 - d. Save the file

Use Git Bash / Git GUI to commit the changes with message "Version 2.0: Much shorter now!".

8. Now, push the local version to the remote:
 - a. Use `git push -u origin master` in Git Bash OR
 - b. Click the "Push" button in Git GUI using all the default options.

Note:

You may get prompted to key in your id and password. **[Exploration]** Find out how you can setup your "profile" so that Git automatically use the same set of id + password for you without prompting.

9. Check your remote repo on GitHub, it should now be updated with version 2.0.
10. One last point, since you can now have multiple local repo across a number of devices, it is possible for some of the local repos to be outdated w.r.t. to the remote repo (e.g. you have pushed version 2.0 from your laptop, but it is still version 1.0 on your home's desktop PC). So, it is always a good practice to "update" the local repo before you resume working. "update" is done via `git fetch` followed by a `git merge` OR `git pull`. We will learn about how to perform the fetch+merge / pull in the new core objective.

Learning Point:

Git workflow involves **two repositories**, one local (on the device you are currently using) and one remote (on a network location, e.g. on GitHub).

A typical workflow looks like the following:

- a. Fetch the latest version from remote.
- b. Merge with the local version (if needed).
- c. Work on the file(s)
- d. Local commit
- e. [c, d] may be repeated many times
- f. Push the local version to remote at the end of a work session (you had enough).

Lab Report Question 7:

Sometimes, we hear statement like "Version control is super simple! You can setup GitHub on your hard disk and use GitHub commands to keep track of everything!". Is there anything wrong with this statement? Why / Why Not?

C4. Using Git for team development

We will now learn a simple way for multiple team members to collaborate on the same code base. The following part is written for **two** imaginary persons, **Uno** and **Duo**. If somehow there is an odd number of students in your studio group, you can have another person observing the steps of **Duo** for a team of 3 members.

Before we start:

1. Find a partner, and decide **who is playing the role of Uno and Duo respectively**. Write / record it down, so that you won't be confused later. **Uno** is the "team leader" and **Duo** is the "team member" in this setup.
2. To highlight some of the interesting scenarios, the steps below are **timing sensitive**, i.e. steps must be carried out in a certain order to achieve the desired effect. We will try to use a table format to show actions for **Uno** and **Duo** on the left and right columns respectively. A single row with merged columns means both persons need to carry out the same action.
3. When you reach a step that reads "**Stop and Check**", make sure you take a moment to observe all the required steps before moving on. Please take your time and don't rush as many steps are hard to reverse / undo.

Setup:

For Uno	For Duo
Unzip the " TeamFolder.zip " on your desktop.	Observe what Uno is doing. Complain loudly as an encouragement if he/she is too slow.
Check that there are 3 files inside.	
Follow [C3 – Step 1 to 4] to: <ol style="list-style-type: none"> a. Create a new remote repo on GitHub "EPP_Git_Team_Demo" b. Commit all 3 files(message "V1.0") c. Push to remote repo on GitHub 	
Stop and Check: Verify the GitHub page of Uno . Check whether 3 files "Code A.txt", "Code B.txt" and "Code C.txt" are all present.	

For Uno	For Duo
On the GitHub, click on the "Setting" Tab for the " EPP_Git_Team_Demo", then click "Collaborators" on the left panel.	Observe closely and learn how it is done. You may need to do it in future for your own team.
Add the GitHub id /email of Duo then click on " Add Collaborator "	Double checks whether your id is correct.
	Check your email (the one used to register on GitHub). You should've received an invitation email. Click on the provided link and accept the invitation .
After Duo accepted the invitation, refresh the setting page and you should see the status no longer reads " awaiting confirmation ".	You should now have access to the repo. It should also be listed under your repo list with the format "<id of Uno> / EPP_Git_Team_Demo"
Now it's your turn to compliment Duo .	Clone the repo to your laptop using the idea learned in C3 – Step 6.
<u>Stop and Check:</u> Both Uno and Duo should now have the folder and the same set of files. Uno's was created from the zip archive while Duo's was created by the Git Clone operation.	

Scenario 1. Uno and Duo changing different files

For Uno	For Duo
Edit the file "Code A.txt". Make two set of changes: <ol style="list-style-type: none"> Add a line "Version 1.1 Uno" at the top. Change a few lines to read "X. Code Uno", e.g. "3. Code Uno", "5. Code Uno" etc. 	Edit the file "Code B.txt". Make two set of changes: <ol style="list-style-type: none"> Add a line "Version 1.1.1 Duo" at the top. Change a few lines to read "X. Code Duo", e.g. "2. Code Duo", "5. Code Duo" etc.
Save the file and <u>commit locally</u> with message " scenario 1 – Uno ".	Save the file and <u>commit locally</u> with message " scenario 1 – Duo ".
Push the commit to remote repo. Verify you see the changes on the GitHub page.	Do NOT push to remote repo yet.

Stop and Check:

Since **Uno**'s changes were based on the original V1.0, the push will succeed without issue. Before you proceed, understand that **Duo**'s case is now "complicated" as his/her changes are now based on an outdated version (the most current version is the one just pushed by **Uno**). Continue on to see how **Git** helps to alert these kind of issues.

For Uno	For Duo
	Perform a push to remote repo now.
Read the error message on Duo's laptop. As mentioned, Duo's push has problem as there were difference with the most recent version on remote repo. Follow Duo's next few steps to learn how to resolve these issues.	
	<p>We need to first fetch the current version from remote repo. Note that fetching does not affect the changes you have made.</p> <p>a. "Menu bar→Remote→Fetch From → Origin" in Git GUI OR</p> <p>b. Use <code>git fetch origin</code> in Git Bash</p>
	<p>Perform a merge where Git attempts to merge the fetched version with your changed version.</p> <p>a. "Menu bar→Merge→Local Merge" in Git GUI OR</p> <p>b. Use <code>"git merge origin/master"</code> in Git Bash</p> <p>Use the message "Merge with V1.1 Uno" if prompted.</p>
	Retry the push to remote repo now. It should now succeed.
<u>Stop and Check:</u> Take a look at the GitHub page for this repo. You should find it fairly interesting as " Code A.txt " is updated by " Scenario 1 – Uno " commit, and " Code B.txt " is by " Scenario 1 – Duo ".	

For Uno	For Duo
<p>Now, it is your turn to be outdated. In real life, you probably won't know you are on an outdated version until you perform a push to remote.</p> <p>In this case, however, we are going to skip the "push-error-update" cycle and immediately go into the "update" part.</p> <p>We are going to use this chance to show you an alternative to "Fetch+Merge" just performed by Duo.</p> <p>The alternative is known as <code>git pull</code>, which is essentially a shortcut for performing both fetch and merge in one action.</p> <ul style="list-style-type: none"> • Use "<code>git pull origin</code>" in Git Bash • There is no equivalent in Git GUI ☺ (i.e. you have to do fetch+merge). 	<p>Watch happily how Uno struggles with the same issue you had just a while back.</p>
<p>Stop and Check: Both Uno and Duo should check that they have the updated "Code A.txt" and "Code B.txt" in their folder at this point.</p>	

Lab Report Question 8:

- Give the URL for your remote repo page. We will take a look at your repo later.
- Take a screenshot showing the remote repo content.

Learning Point:

Scenario 1 demonstrated a simple way to collaborate between team members. By assigning different source code files to each member, this guaranteed zero conflict in the merging stage (i.e. the merging can be done with minimum effort). However, it is also rather restrictive. We will see a more challenging scenario next.

Scenario 2. Uno and Duo changed the same file with different updates

For Uno	For Duo
Edit the file " Code C.txt ". Make two set of changes: <ol style="list-style-type: none"> Add a line "Version 2.1 Uno" at the top. Change these specific lines to read "3. Code Uno", "5. Code Uno". 	Edit the file " Code C.txt ". Make two set of changes: <ol style="list-style-type: none"> Add a line "Version 2.1.1 Duo" at the top. Change these specific lines to read "1. Code Duo", "5. Code Duo".
Save the file and commit locally with message " scenario 2 – Uno ".	Save the file and commit locally with message " scenario 2 – Duo ".
Do NOT push to remote repo yet.	Push the commit to remote repo. Ensure you can see the changes on the GitHub page.
Stop and Check: We reverse the ordering between Uno and Duo here to spice things up. Uno is now "in trouble" as his/her changes are based on outdated version. Worse still, some of the changes actually conflict with those of Duo 's.	

For Uno	For Duo
Perform a push to remote repo now. You should see the same error message Duo received in previous scenario.	Watch (gleefully) over Uno as he/she struggles. Seriously though, this is one of the more important topic in using Git known as Merge Conflict Resolution .
Let's take it slowly and perform the "fetch and merge" separately for clarity.	
<ol style="list-style-type: none"> Use "Menu bar→Remote→Fetch From → Origin" in Git GUI OR Use "git fetch origin" in Git Bash 	
Perform a merge where git attempts to merge the fetched version with your changed version. <ol style="list-style-type: none"> Use "Menu bar→Merge→Local Merge" in Git GUI OR Use "git merge origin/master" in Git Bash 	
The merge will fail as there are conflicts	

(incompatible changes). We now need to resolve them.	
<p>We will describe the "harder way" first:</p> <ul style="list-style-type: none"> • Use editor to open the file with conflicts, in this case, "Code C.txt". • Each conflict is highlighted with markers like "<<<<<" and ">>>>>" with a "=====" in between to show "our" change on top, and "their" change in the lower part. • There should be two conflicts: On the "Version XXXX" line and "5. Code XXXX". <p>Let's resolve them as follows:</p> <ol style="list-style-type: none"> Change to "Version 2.2 Conflict is bad!" Change to "5. Code Duo" (let them win!) Save the file <p>Remember to remove the additional markers "<<<<<", ">>>>>" and "=====</p>	
<p>Learning Point: Imagine the changes are source code instead of simple text. So, what you are doing above is to decide the better version of the code and/or rewrite the conflict portion entirely. Merge conflict resolution takes time, effort and skill 😊.</p>	
<p>Essentially, we have a new version now without any conflict. So:</p> <ol style="list-style-type: none"> Stage the changes Commit locally with message "Scenario 2 – Conflict ended by Uno" Push to remote. 	
<p><u>Stop and Check:</u></p> <p>Verify the changes on the GitHub page.</p>	

Lab Report Question 9:

On the remote repo, open up the file "Code C.txt" and take a screenshot.

Question Scenario:

Lab Report Question 10:

This question is posed as a scenario. You are supposed to achieve the final result using what you have learned above. The end result can of course be achieved (too) easily if you "synchronize" the steps with your partner, i.e. "We are telepathically linked, so no conflicts whatsoever.

Hence, we will also look at whether you exercise some of the ideas learned in this studio for accomplishing the final goal.

We will check your GitHub repo after the studio. Please don't change / remove the remote repo until you've received our feedback.

For Uno	For Duo
Change all Prime Number lines (i.e. lines 2, 3, 5, 7). to read " X. Code Uno Prime " in " Code A.txt ".	Change all Fibonacci Number lines (i.e. lines 1, 2, 3, 5, 8). to read " X. Code Duo Fibonacci " in " Code A.txt ".
Leave other changes from earlier scenario intact.	Leave other changes from earlier scenario intact.
<p>Make sure the changes are eventually reflected on the remote repo. Whenever there is a conflict, Uno "wins" on all odd number lines, Duo "wins" on all even number lines. (Sorry for this ultra-realistic scenario 😊)</p> <p>The remote repo should shows the file in the final state when you are done.</p>	

Final Learning Point:

In C4, what we used is known as **centralized workflow**, in which all team members pushes to a central remote repo. There are other alternative workflows, e.g. branch-pull request – merge etc for more complicated code base / scenarios. You can take a look at reference [2] for more.

Reference / Resources

1. A great site for self-learning Git (<https://try.github.io/>)
2. Good resource, we used it for the git centralized workflow exercises
<https://www.atlassian.com/git/tutorials/comparing-workflows>
3. Centralized Workflow Explanation
<https://gist.github.com/datagrok/d1650d85496cd509d42b8656d30410cf>
4. Git command reference (<https://git-scm.com/docs>)