Week 3 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 "W3S2_*Student_ID*.docx / .pdf" into your studio group's workbin ("Luminus Files" → "Student Submission" → "Week 3 – 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. Remote Repository and GitHub
C3. Using Git for team development

Preparation (Before the studio):

- Download and install **`Git`** on your laptop. Use standard (default) options during installation for simplicity. The Windows and Mac OS versions can be found on Luminus Files → Studio Materials → Week 03 – 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 Luminus. 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.**

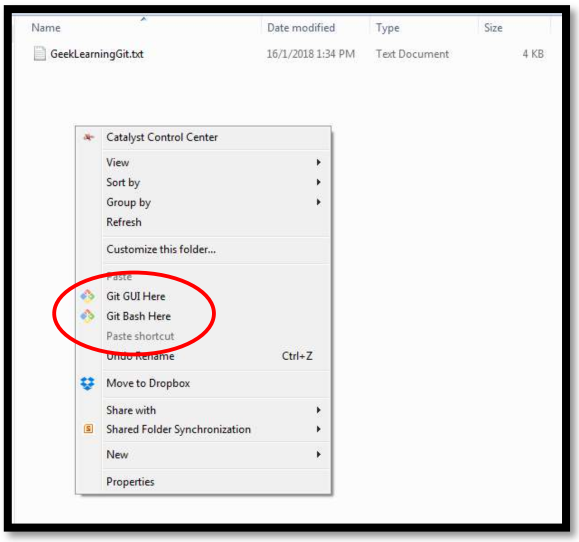
Studio Setup:

- **[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***" (as seen on the screen shot ➔).

[Mac User: Need to start a "terminal" and enter "git gui" to check.]



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.

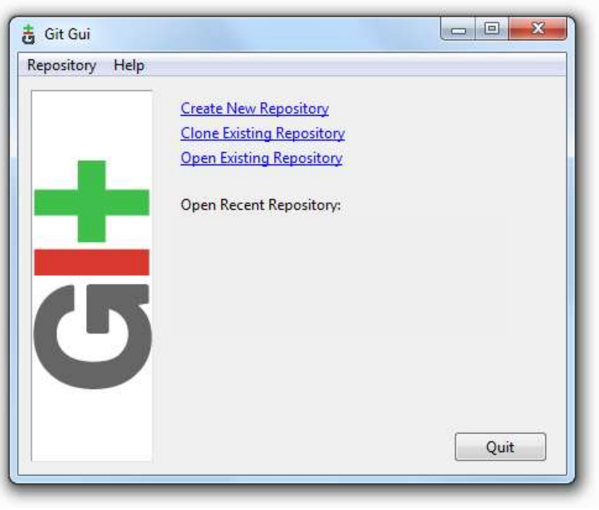
What is Git?
Git is a open-source and free implementation of version control software. It is currently the most widely used version control software in the market.

C1. Understand and using version control for personal use

"Git" is a very powerful (but also very complicated) version control system. To ease your learning, we will start with the standard **GUI variant** that comes with the installation.

1. Unzip the "`SoloFolder.zip`" again and rename the folder "`SoloGUI`".

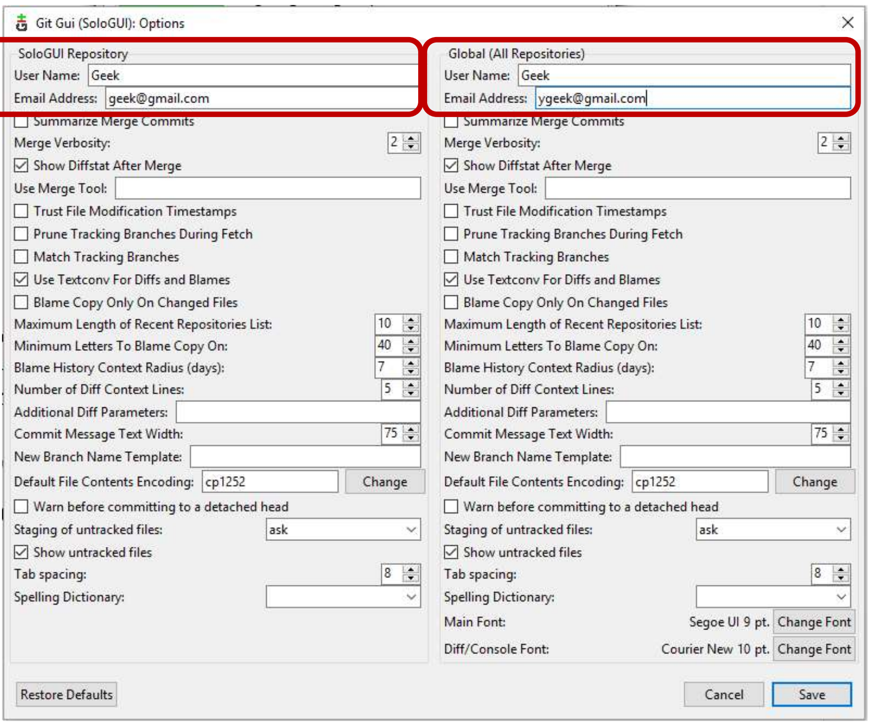
2. Right click in the "`SoloGUI`" folder and choose "Git GUI Here". You should see this interface ➔



Click "**Create New Repository**", then browse and select the "**SoloGUI**" folder. This initializes the Git repository. Open a file browser to check the in the folder, you can see that a new "hidden" folder `.git/` has been created under the "**SoloGUI**". This folder contains information used by Git to perform the tracking and versioning.

Learning Point:
Git needs to setup its own storage (known as **repository**) to keep track of the files under a working folder. The repository should be placed at the highest (topmost) folder which contains the files and subfolders you want to keep track of.

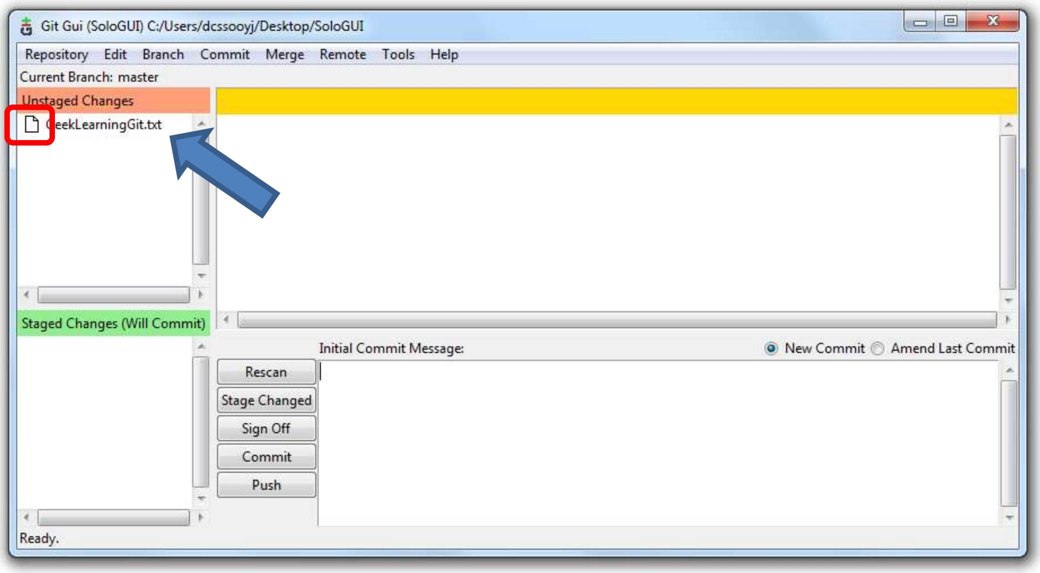
3. Git users are identified with a "name" and more importantly "email" address. Go ahead and click on "**Menu→Edit→Options….**". You should see the following dialog box:


Git Gui (SoloGUI): Options

SoloGUI Repository		Global (All Repositories)	
User Name:	Geek	User Name:	Geek
Email Address:	geek@gmail.com	Email Address:	ygeek@gmail.com
Summarize Merge Commits	☐	Summarize Merge Commits	☐
Merge Verbosity:	2	Merge Verbosity:	2
Show Diffstat After Merge	✓	Show Diffstat After Merge	✓
Use Merge Tool:		Use Merge Tool:	
Trust File Modification Timestamps	☐	Trust File Modification Timestamps	☐
Prune Tracking Branches During Fetch	☐	Prune Tracking Branches During Fetch	☐
Match Tracking Branches	☐	Match Tracking Branches	☐
Use Textconv For Diffs and Blames	✓	Use Textconv For Diffs and Blames	✓
Blame Copy Only On Changed Files	☐	Blame Copy Only On Changed Files	☐
Maximum Length of Recent Repositories List:	10	Maximum Length of Recent Repositories List:	10
Minimum Letters To Blame Copy On:	40	Minimum Letters To Blame Copy On:	40
Blame History Context Radius (days):	7	Blame History Context Radius (days):	7
Number of Diff Context Lines:	5	Number of Diff Context Lines:	5
Additional Diff Parameters:		Additional Diff Parameters:	
Commit Message Text Width:	75	Commit Message Text Width:	75
New Branch Name Template:		New Branch Name Template:	
Default File Contents Encoding:	cp1252	Default File Contents Encoding:	cp1252
Warn before committing to a detached head	☐	Warn before committing to a detached head	☐
Staging of untracked files:	ask	Staging of untracked files:	ask
Show untracked files	✓	Show untracked files	✓
Tab spacing:	8	Tab spacing:	8
Spelling Dictionary:		Spelling Dictionary:	
		Main Font:	Segoe UI 9 pt.
		Diff/Console Font:	Courier New 10 pt.

Provide the "User Name" and "Email Address" either on the **left (*SoloGUI repository*)** for this repo only or on the **right (Global)** for all repos on this machine. **Save** the changes to return to the main interface.

4. In the main interface, the status of the files being tracked is shown automatically:


Git Gui (SoloGUI) C:/Users/dcssooyj/Desktop/SoloGUI

Current Branch: master

Unstaged Changes

GeekLearningGit.txt

Staged Changes (Will Commit)

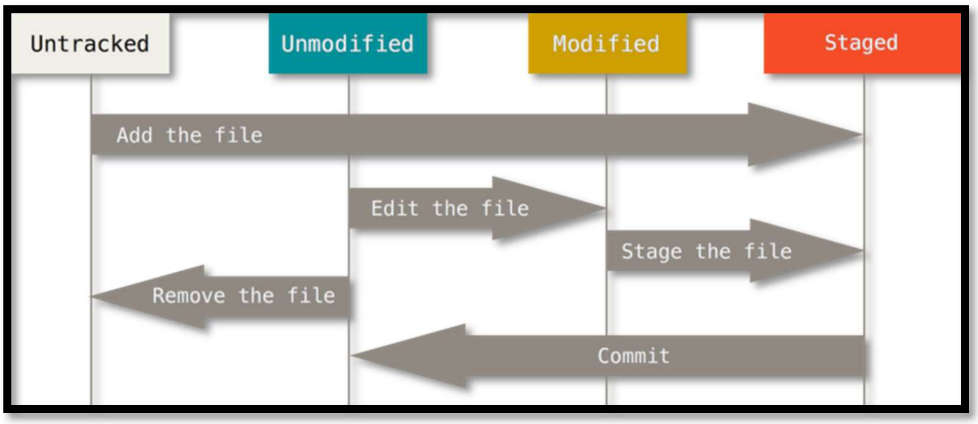
Initial Commit Message:

[✓ New Commit] Amend Last Commit

Ready.

You can see clearly that the file "**GeekLearningGit.txt**" is listed under "**Unstaged Changes".**

Learning Point:
Git uses the following workflow for all files under the folder it is monitoring:

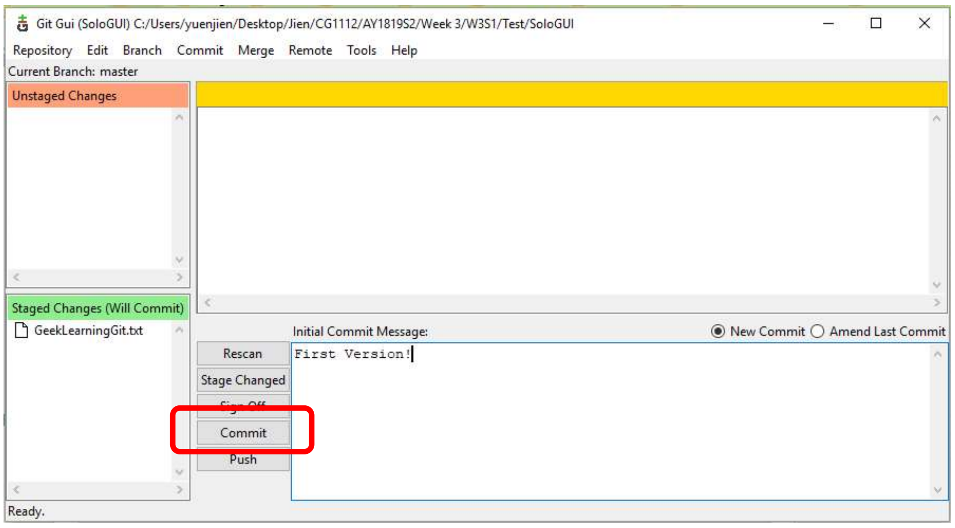


So, a file can be:
- "**Untracked**": Git does not monitor the file, i.e. the file is ignored.
- "**Tracked**": Git monitor the file, with the following 3 additional categories:
  - "**Unmodified**": The file is the same since last commit (explained later)
  - "**Modified**": The file has been modified, but not yet ready for commit.
  - "**Staged**": The file has been modified, and the new version is ready for commit.

Under the GUI version, Git automatically tracks all files. As we do not have a prior record of the file "**GeekLearningGit.txt**", the file is classified as "Tracked and Modified" but not "Staged" yet.

Clicking on the ***file icon*** (Circled in **red** above) will move it to the "**Staged Changes (Will Commit)"** category.

5. 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 "*First Version!*" in the "*Initial Commit Message*" text box, then click "Commit":


Git Gui (SoloGUI) C:/Users/yuenjien/Desktop/Jien/CG1112/AY1819S2/Week 3/W3S1/Test/SoloGUI
Repository Edit Branch Commit Merge Remote Tools Help
Current Branch: master
Unstaged Changes
Staged Changes (Will Commit)
GeekLearningGit.txt
Initial Commit Message:
New Commit
Amend Last Commit
Rescan
Stage Changed
Sign Off
Commit
Push
Ready.

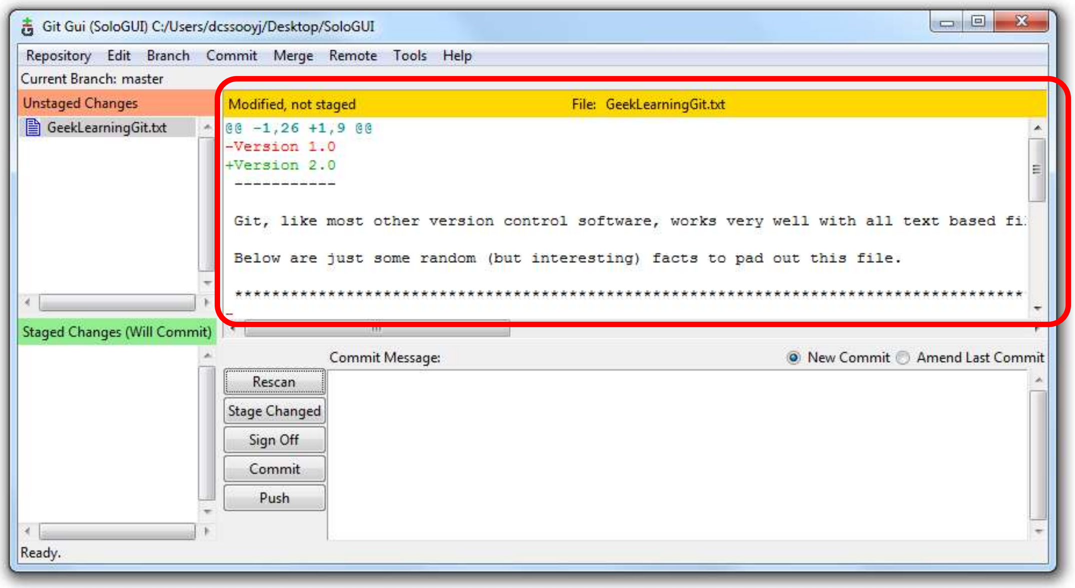
Initial Commit Message: [✓ New Commit] Amend Last Commit

```
First Version!
```

A "Commit Message" is commonly used to summarize the major changes to the files since last version. When you have many versions, the commit message can help to identify a specific version easily. (Just like all good practices, this advice is destined to be ignored by most ;-)).

6. Let's modify the file "**`GeekLearningGit.txt`**". Use a **text editor** (e.g. notepad) and perform the following changes to the file:
   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.

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.



```
@@ -1,26 +1,9 @@
-Version 1.0
+Version 2.0
 -----------

 Git, like most other version control software, works very well with all text based fi

 Below are just some random (but interesting) facts to pad out this file.

 ******************************************************************************
```

Lab Report Question 1:

You can see on the right panel that Git uses some kind of notations to summarize the changes to a file. Explain the notation used by the two lines involving the change on the sentence "*Version……*" (red and green respectively in the screenshot above. [You can google "**`git diff notation`**" to learn more.]

7. Go ahead to:
   a. Stage the change.
   b. Commit the change with a message "Version 2.0: Much shorter!"

   Now, it's a good time to figure the workflow of Git. Answer the following:

   Lab Report Question 2:

   Why does Git insists on you manually staging changes for the next commit? Since Git can easily detect the changes, why couldn't it just commit all changes?

   Hint: You should answer in term of typical software development (i.e. coding) setting, i.e. if you are writing code, why don't you want all changes committed automatically?

8. You can find out the versioning history by clicking on the menu bar: "Repository" → "visualize master's history".

   Lab Report Question 3:

   Take a snapshot of the visualized history and paste into your report.

9. Let us now do a few more "advanced" changes to the folder. Follow the steps below:
   a. Add a new text file "`GitIsNotHard.txt`" under the "SoloGUI" folder. The file content does not matter.

   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`"
   - Prepare to commit the commit message "*`Again, with more files!`*"
   - **<u>Take a screenshot</u>** then commit

   Lab Report Question 4:

   Give the screenshot **just before** the commit.

   [Note: Question 5 next page]

Lab Report Question 5:

We chose NOT to archive **notFunny.txt** for demonstrating purpose, can you give **1 type** of files that you do **not** want to track for a typical programming project? Briefly give the reason of excluding the kind of files.

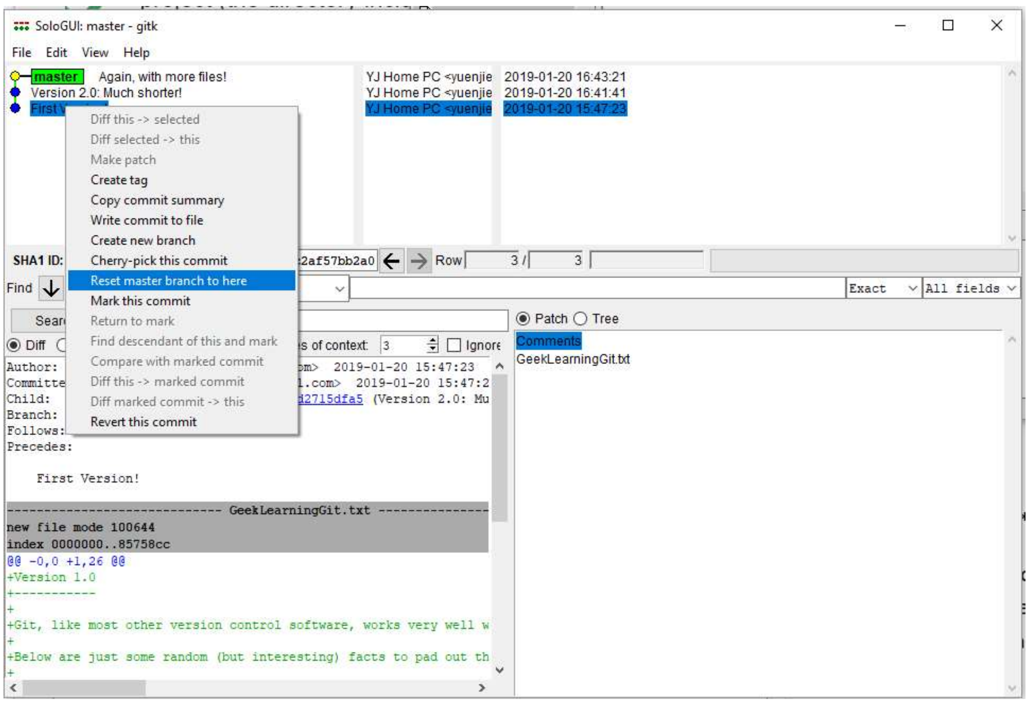
10. A committed Git snapshot essentially act as an "insurance plan" and allow us to change the files freely without worry. Should things go wrong, we have several ways to "go back in time" (i.e. revert to earlier version) in Git. The next few steps explore two options.

11. Edit the file "`GeekLearningGit.txt`", **remove all content** and save in the text editor (i.e. make it an empty file). **Do NOT commit.**
    Click "Commit→ Revert Changes" (or Ctrl-J). Confirm the operation then check the content of "`GeekLearningGit.txt` ".

Lab Report Question 6:
 a. Which "version" of the file content do you see in that file?
 b. What do you think "*`revert changes`*" does?

12. Suppose your project team leader is **super unhappy** that you included frivolous content like the "**`GitJoke`**/" folder into a supposedly serious project. Perform the following steps [Be Careful!]:

    a. Select "Visualize master's history" from the menu bar "Repository".
    b. Right click on "***First Version***" on the top right panel and select the option "Reset master branch to here". Select "**Hard Reset**" when prompted.



```
master  Again, with more files!        YJ Home PC <yuenjie   2019-01-20 16:43:21
Version 2.0: Much shorter!             YJ Home PC <yuenjie   2019-01-20 16:41:41
First V                                YJ Home PC <yuenjie   2019-01-20 15:47:23
```

```
Author:                                 om>  2019-01-20 15:47:23
Committe                                l.com>  2019-01-20 15:47:2
Child:                                  d2715dfa5 (Version 2.0: Mu
Branch:
Follows:
Precedes:

    First Version!

----------------------------- GeekLearningGit.txt --------------
new file mode 100644
index 0000000..85758cc
@@ -0,0 +1,26 @@
+Version 1.0
+-----------
+
+Git, like most other version control software, works very well w
+
+Below are just some random (but interesting) facts to pad out th
+
```

Comments
GeekLearningGit.txt

Lab Report Question 7:

a. Give the status of the following files. Status can be "exists (which version?)", "removed".
   i. GeekLearningGit.txt
   ii. GitIsNotHard.txt
   iii. funny.txt
   iv. notFunny.txt
b. Briefly describe the effect of the **hard reset**.

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

1. We chose the GUI version to tone down the learning curve for you. Although it is more "intuitive", there are many functionalities not exposed by the interface. You can take a look at the attached appendix (which is essentially a similar walkthrough for C1 but using "`git`" command directly to learn more.

2. From the experiments, 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.

3. Git allows revert changes to **any earlier version** instead of just the most recent. Unfortunately, there is no direct way to do it in GUI version. Git command allows you to revert to any version in the history!

4. 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….

   Many IDE (Integrated Development Environment) provides Git integration too, e.g. Visual Studio Code, IntelliJ, etc.

C2. 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 (*https*, *ssh* etc), you can operate directly 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".**

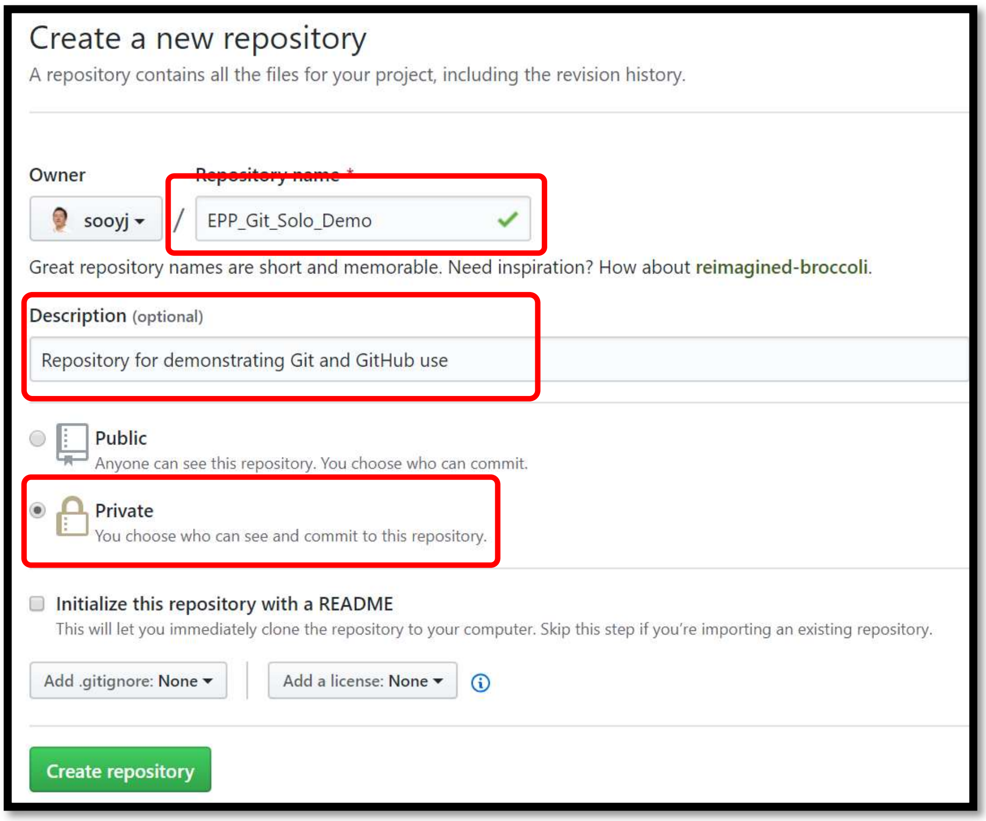
Learning Point:
This means that you can setup your own remote repository (e.g. on a server somewhere, on sunfire etc).

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 well known examples, where GitHub is currently the de facto leader in the field. You can find many **many** public code repositories on GitHub for you to contribute to and / or learn from. Your GitHub portfolio can also act as a great showcase of your project work to the whole world. *Nowadays, 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. CS2113** also uses GitHub extensively.

1. Unzip the "**`SoloFolder.zip`**" yet again, and name the folder "**`SoloGitHub`**".

2. Log into your GitHub account. Navigate to the "repositories" page (e.g. under your profile on the top right, click "your repositories". Click "New" (on the right side of the page) to 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 reimagined-broccoli.

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

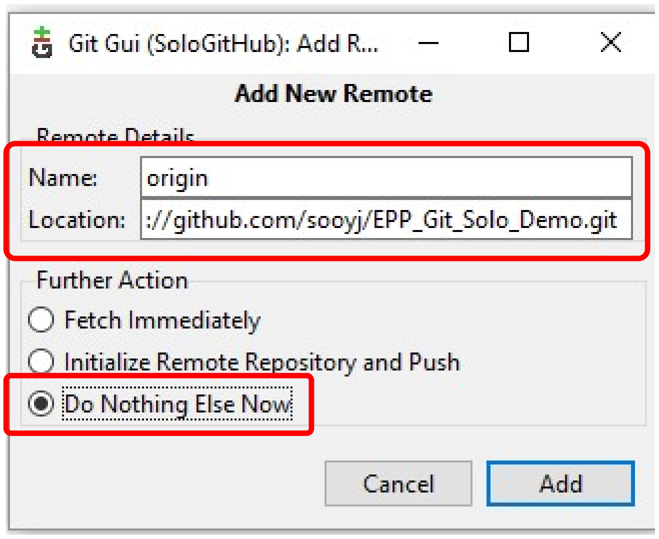
Click "Create Repository" to create your first remote repository on GitHub. The result page contains very useful summary of how you can proceed further. **Take note of the repository url in the first section, copy the url for later use.**

Caution:
Repositories for a normal user account on GitHub are by default **public**, i.e. accessible to anyone. So, <u>refrain from posting on-going school work</u> (e.g. programming assignments, labs, projects) on public repositories. You should take advantage of the fact that GitHub now provides **private** repositories hosting **for free**.

3. Start "Git GUI" and select the "`SoloGitHub`" folder, then:
   a. Stage the "`GeekLearningGit.txt`" file
   b. 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. Let us first add the remote repo on the GitHub. Click "**`Remote`**" on the menu bar and select "Add…". Use the following setting:



Git Gui (SoloGitHub): Add R...

Add New Remote

Remote Details

Name: origin

Location: ://github.com/sooyj/EPP_Git_Solo_Demo.git

Further Action

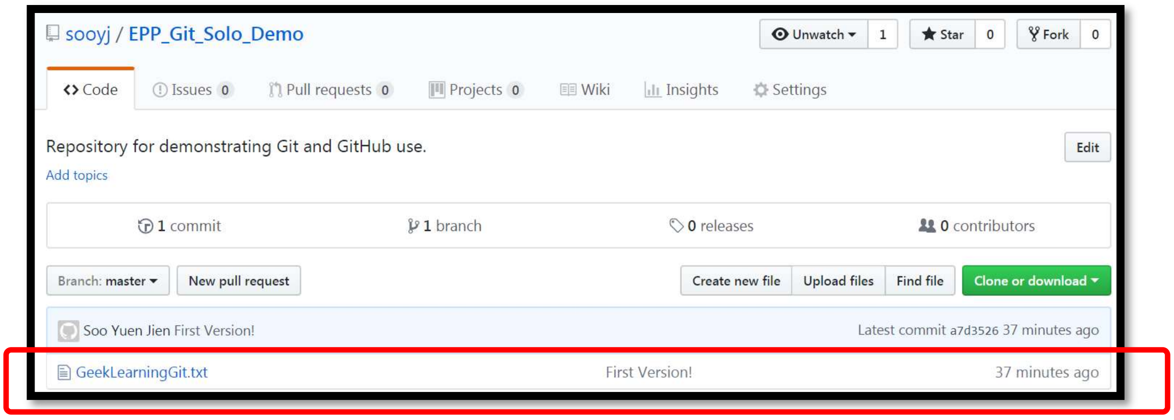
○ Fetch Immediately

○ Initialize Remote Repository and Push

● Do Nothing Else Now

Copy and paste your GitHub repo url into the "Location:" textbox. Select "Do Nothing Else Now" radio button then Click "Add". You should return to the main interface if the remote repo is added properly.

5. In the main interface, click "Push". Accept the default setting for the "Push" dialog. This will "push" the latest local commit to the remote repo. Check your GitHub page, you should see the commit and the file "`GeekLearningGit.txt`":



sooyj / EPP_Git_Solo_Demo

Repository for demonstrating Git and GitHub use.

1 commit | 1 branch | 0 releases | 0 contributors

Soo Yuen Jien First Version! — Latest commit a7d3526 37 minutes ago

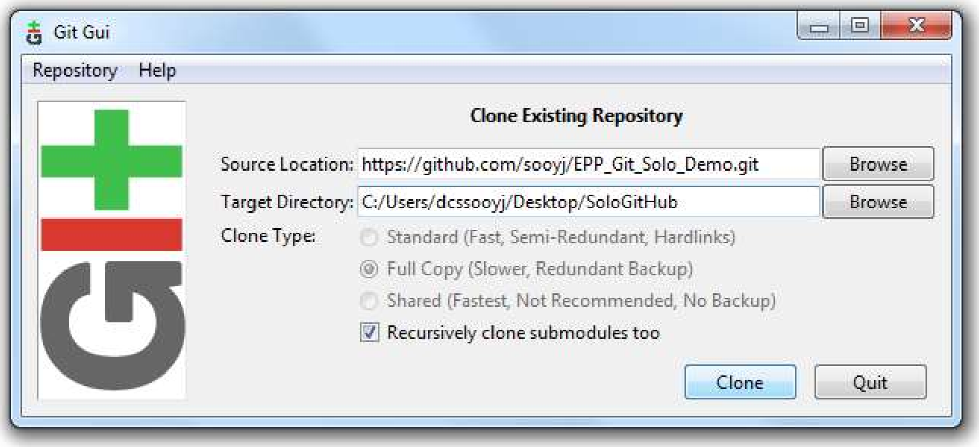
GeekLearningGit.txt — First Version! — 37 minutes ago

GitHub Credential:
You may get a prompt asking for your GitHub credential for the first use. Just give your GitHub id and password accordingly. The credential should be saved automatically by Git GUI and you wont need to key in again in future.

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/`" on your laptop.

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



Clone Existing Repository

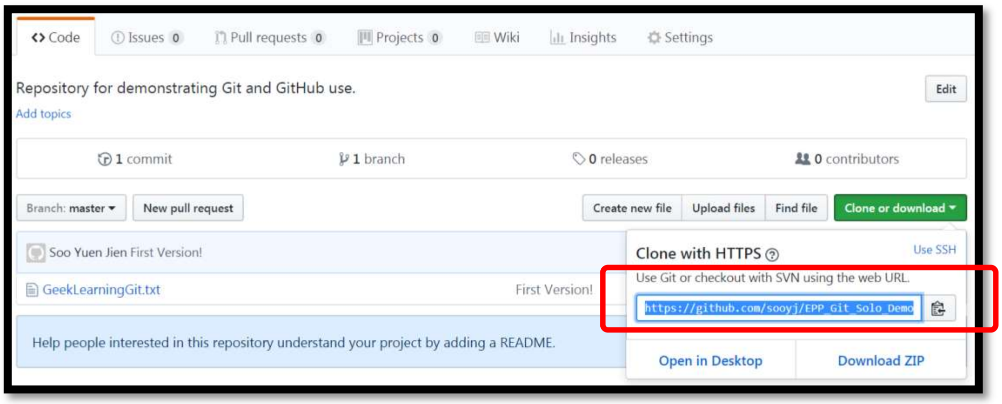
Source Location: https://github.com/sooyj/EPP_Git_Solo_Demo.git

Target Directory: C:/Users/dcssooyj/Desktop/SoloGitHub

Clone Type: ( ) Standard (Fast, Semi-Redundant, Hardlinks) [✓ Full Copy (Slower, Redundant Backup)] ( ) Shared (Fastest, Not Recommended, No Backup)

[✓] Recursively clone submodules too

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:



Repository for demonstrating Git and GitHub use.

Clone with HTTPS

Use Git or checkout with SVN using the web URL.

https://github.com/sooyj/EPP_Git_Solo_Demo

Help people interested in this repository understand your project by adding a README.

Note: 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 "`SoloGitHub`" folder created 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 "GeekLearningGit.txt" 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 in the text editor.

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

8. Now, click the "Push" button in Git GUI using all the default options push the local version to the remote. Check your remote repo on GitHub, it should now be updated with version 2.0.

9. 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 start working (i.e. this should be your working ritual: "Update, work, commit, push"). Updating repo 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 commit to remote at the end of a work session.

Lab Report Question 8:

Sometimes, we hear statements like "You can setup *GitHub* on your hard disk and use *GitHub* commands to keep track of the files!". What's wrong with this statement?

C3. 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 wont 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 specific 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 "**<u>Stop and Check</u>**", 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. Check that there are 3 files inside.	Observe what **Uno** is doing. Complain loudly as a form of encouragement if he/she is too slow.
Follow [**C2 – Step 1 to 4**] to: 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	
<u>Stop and Check:</u> 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/email id 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 receive an invitation email from **Uno**. 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 "encourage" **Duo**.	Clone the repo to your laptop using the idea learned in C2 – Step 6.
Stop and Check: 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: a. Add a line "`Version 1.1` Uno" at the top. b. 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: a. 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.**
<u>Stop and Check</u>: 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** alerts you of these issues.	

For Uno	For Duo
Observe what Duo is doing.	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.	
	We need to first **fetch** the latest version from remote repo. Note that fetching **does not affect the changes you have made.** **Git GUI:** "Menu bar→Remote→Fetch From → Origin"

	Perform a **merge** where Git attempts to merge the fetched version with your changed version. **Git GUI:** "Menu bar→Merge→Local Merge" Use the message "**Merge with V1.1 Uno**" if prompted.
	Retry the push to remote repo now. It should succeed now.

Stop and Check:
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
Now, it is your turn to be outdated. In real life, you wont know you are working on an outdated version until you perform a push to remote. In this case, however, we are going to skip the "push-error-update" cycle and immediately go into the "update" part. Perform the steps to "fetch" and "merge", then push your version to the remote repo.	Watch happily how **Uno** struggles with the same issue you had just a while back.

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.

Lab Report Question 9:

Take a snapshot of your repo content and paste into report.

Learning Point:

Scenario 1 demonstrated a simple way to collaborate between team members. By assigning different source code file 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 "**<u>Code C.txt</u>**". Make two set of changes: a. Add a line "**Version 2.1 Uno**" at the top. b. Change these **<u>specific</u>** lines to read "**3. Code Uno**", "**5. Code Uno**".	Edit the file "**<u>Code C.txt</u>** ". Make two set of changes: a. Add a line "**Version 2.1.1 Duo**" at the top. b. Change these **<u>specific</u>** lines to read "**1. Code Duo**", "**5. Code Duo**".
Save the file and commit **<u>locally</u>** with message "**scenario 2 – Uno**".	Save the file and commit **<u>locally</u>** 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.
<u>Stop and Check</u>: 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 advance Git usage known as **Merge Conflict Resolution.**
Let's take it slowly and perform the "fetch and merge" separately for clarity. **Git GUI:** "Menu bar→Remote→Fetch From → Origin"	
Perform a **merge** where git attempts to merge the fetched version with your changed version. **Git GUI:** "Menu bar→Merge→Local Merge" The merge will **fail** as there are **conflicts** (incompatible changes). We now need to resolve them.	

We will describe the "harder way" first: • 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". Let's resolve them as follows: a. Change to "Version 2.2 Conflict is bad!" b. Change to "5. Code Duo" (let them win!) c. Save the file Remember to remove the additional markers "**<<<<**", "**>>>>>**" and "**=====**"	

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 ☺.

Essentially, we have a new version now without any conflict. So: a. Stage the changes b. Commit locally with message "Scenario 2 – Conflict ended by Uno" c. Push to remote.	

Stop and Check:

Verify the changes on the GitHub page.

Lab Report Question 10:

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

[Optional] Question Scenario:

[Optional] Question 11:

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, try to exercise some of the ideas learned in this studio for accomplishing the final goal.

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**". Leave other changes from earlier scenario intact.	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.
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 ☺) The remote repo should show the file in the final state when you are done.	

Final Learning Point:

In C3, 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)

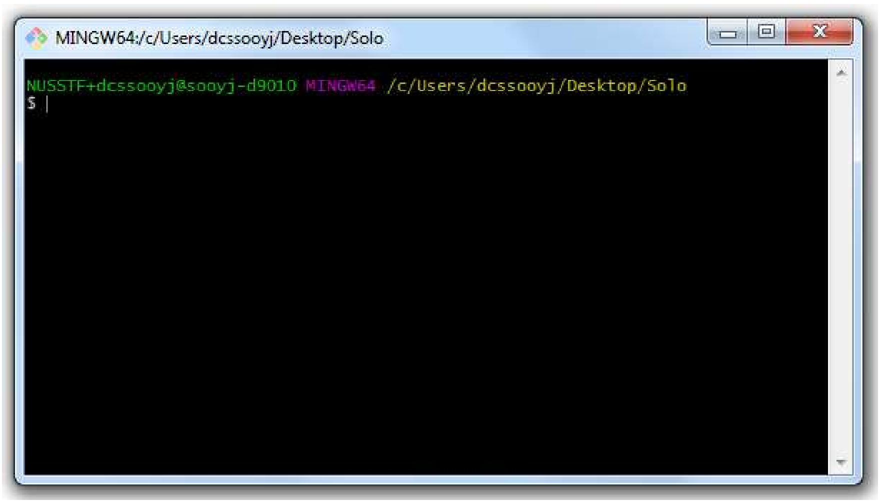
Appendix. Learning the "git" command

Below is a similar walkthrough for C1, but using the "git" command directly instead of GUI. You can use it to start learning the "interesting" details for the powerful "git" command. [Side note: This was actually part of the studio in the first run of EPP2 and caused the studio to overrun badly…… ☺]

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".

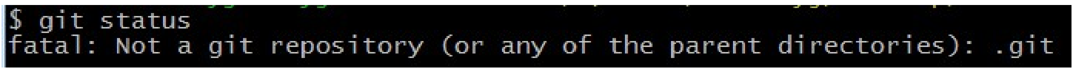
1. Right click in the "**Solo**" folder and choose "Git Bash Here". You should see a terminal as follows:



```
NUSSTF+dcssooyj@sooyj-d9010 MINGW64 /c/Users/dcssooyj/Desktop/Solo
$ |
```

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**.

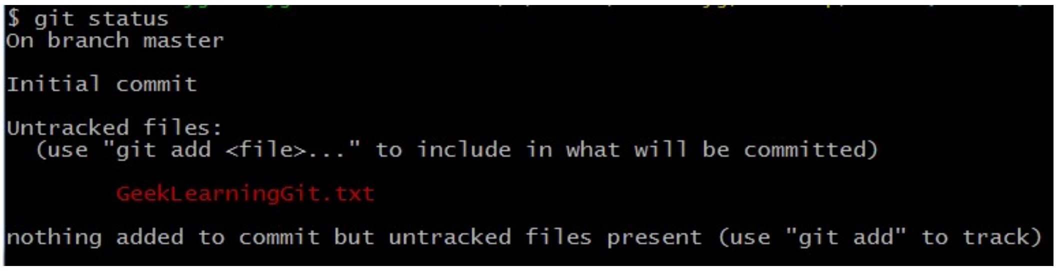
2. Enter `git status` into the bash windows. You should receive an error:



```
$ 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:

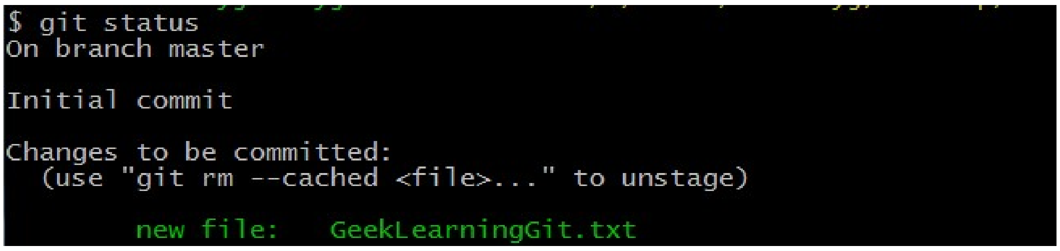


```
$ git status
On branch master

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. [Note that if this is the first time you are using Git on a particular machine, then you need to setup **user name** and **email** information. These information are kept with each commit for ease of reference. Setup the information as follows:

```
$ git config [--global] user.name "<Your Name>"
$ git config [--global] user.email <Your Email>
```

You can include the "**`--global`**" flag if you want to use the same information for all Git repositories on a machine.

8. Enter `git commit -m "First version!"`. 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. Just ensure the commit is properly performed by checking `git status` which should reports "**nothing to commit, …**"

9. 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).

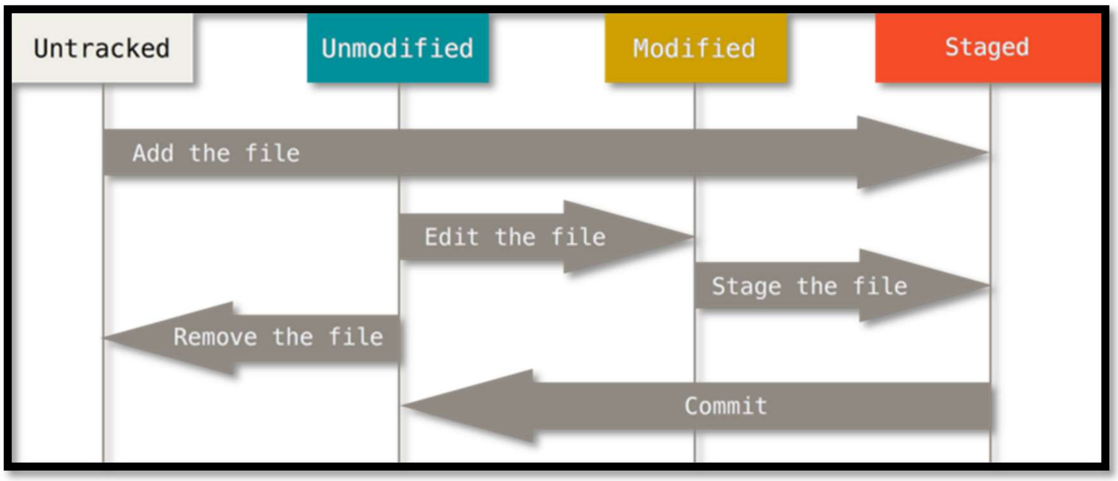
10. Run `git status`, Git should indicates that the changes have been detected. To find out more about the changes, you can use `git diff`.

Self-check 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 about "Version….").

11. It is probably naturally 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".

12. Apparently, Git follow 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.

Self-Check 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 term of programming source code).

13. 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.

Self-Check Question 3:

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

14. 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.
    a. Edit the "GeekLearningGit.txt" by changing the first line to "Version 2.5".
    b. Remove all other lines.
    c. 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.

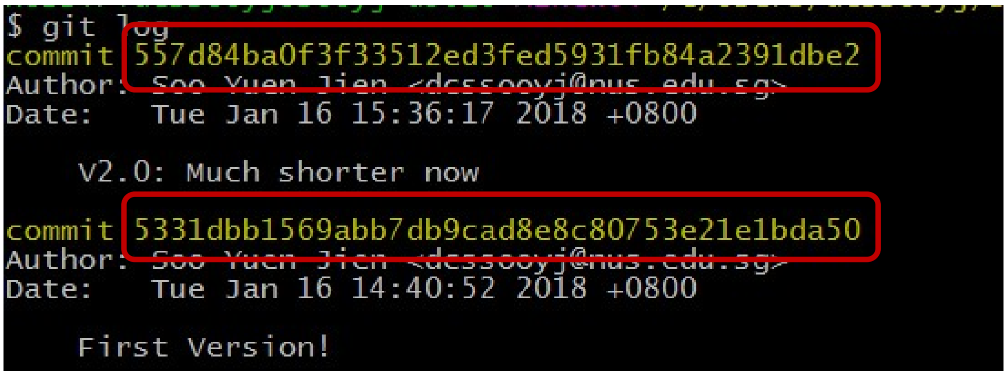
    Verify the content of "GeekLearningGit.txt" has been reverted back to version 2.0

15. 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.

Self-Check Question 4:

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.