CS2043 Assignment 3 — Getting Started with Git

Due: MON JAN-31 2022 by 4pm

PRELIMINARIES

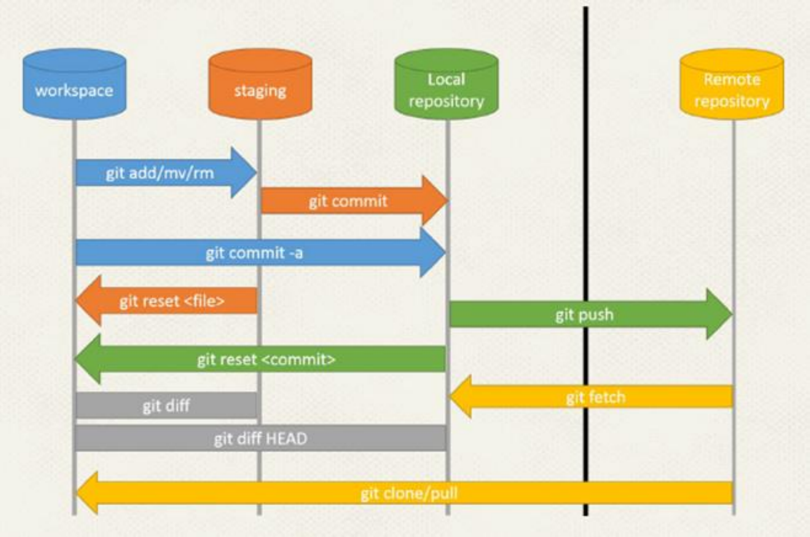
Step 0.1: A Quick Git Overview (read this to get an overview of what we will be covering)

What is git? Git is a version control system. It keeps a history of all changes to files in any folder you tell it to track, as long as you 'commit' the changes. It is used a lot for software development efforts, but it works with most types of files (not just code).

What are the major benefits that git provides? There are two:

- Local Version Control - Git allows you to keep a history of your work so you don't lose ideas buried in old versions. It also allows you to create branches of work to try different things before you decide on a direction.
- Remote Sharing - Git allows you to share work with others and integrates changes to the work submitted by multiple workers. Git doesn't provide the remote share space, you have to create one, or use one that is available in the cloud. Github, and BitBucket are examples of these.

What does a work flow look like with git? This is a snapshot of the various working areas of git. As you move through this lab, you will begin to understand how the various areas work together to help you maintain a history of your work, and share it with others



Common Git Commands:

```
>git init
>git status
>git log --oneline
>git add <filename>
>git commit -m
>git diff
>git restore <filename>
>git reset <hash>
>git revert <hash>
>git branch <branchName>
>git checkout <branchName>
>git clone <remote URL>
>git pull
>git push
```

```
NOTE 1: -x indicates a flag setting using a short form (e.g git commit -m means 'commit with a message')
Note 2: --xxxxx indicates a flag setting using full form (e.g git commit --message)
Note 3: -X often indicates the same flag setting as x, with a forced condition
```

What can I go to get more information about git?

- Everything you need to know about git is available at: https://git-scm.com/
  - Wait until the next step before you download and install anything
  - Download the Pro Git book - it's a great guide
  - Once you understand git, the Reference Manual will be useful
  - The External Links section has additional material which can be useful
- Another very useful reference is available at: https://www.git-tower.com/learn/git/ebook
- Another very useful reference is available at: https://www.atlassian.com/git

Step 0.2: Setting up Git

Git is already installed on the lab computers, but if you want to install it at home, you can download it from the following link:

Source Site: https://git-scm.com/downloads;

Installation Instructions: https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

Other Useful Installation instructions https://www.atlassian.com/git/tutorials/install-git

Additional Installation Recommendations (configuration options available during install for Windows):

1. When selecting components, be sure to include 'Windows Explorer Integration' (both components), and 'Add a Git Bash Profile to Windows Terminal'
2. When prompted, you may want to choose notepad or some other text editor rather than VIM as your default editor
3. When prompted, choose 'main' rather than 'master' as your default branch name
4. When prompted, deselect 'Enable file system caching'

Once Git installs, it will be available on your computer for use in a terminal (ex Windows uers can use DOS Command Line Window, or Powershell – which is my preferred terminal). There are GUI-based tools available for Git, and most IDEs have Git Integration, but **the easiest (and most informative) way to learn git is through the terminal – that is what I recommend**.

NOTE ABOUT POWERSHELL: If you are going to use Powershell, use version 5 or higher (usually ships with Windows 10). You can also use Powershell ISE. Another useful application now available for window is the 'Windows Terminal'. This is simply a window that allows you to open multiple terminals and types (a powershell, a gitbash and a command line). It is also easy to configure some basic visual preferences. You can download it free from the Microsoft Store (search for 'Terminal' and choose 'Windows Terminal')

Make sure your installation works by opening a terminal and running

```
PS C:\Users\user>  git --version
```

You should see something like:

```
PS C:\Users\user>  git –version
git version 2.33.0.windows.2
```

Step 0.3: Getting used to the Terminal

We will be working through the lab in a terminal. The examples provided assume you are using a powershell, but any bash shell will work in a very similar way. Depending on what type of terminal you are in, all or some of these commands will work. Familiarize yourself with them so you know what they do:

Useful terminal commands: `ls, dir, cd, clear`

Reminder: to move to a directory use `cd.. or cd "<path>"` (you can copy and paste pathnames)

Step 0.4: Setting up GitHub

The rest of this lab is split into 3 parts:

1. Working Locally
2. Working with Remotes (using GitHub)
3. Working with GitHub utilities

To speed up the process, create a GitHub account before you come to lab
https://github.com/

If you use your student email address, you can access extra features. you don't need those features for the lab, but as long as you use your student email address, you can access them at a later time if you need them.

NOTE: We Won't use our Project Repositories for this Assignment

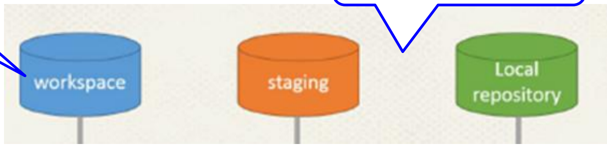
PART 1: WORKING LOCALLY

The basic workflow for working locally (on your local computer) to track changes you make to a workspace is as follows:

1. Create a folder where you will store everything you want to track.
2. Initialize the folder so that git knows to track stuff in it.
3. Work within that folder – it's your workspace. When you have something ready to track, `add` it to the staging area.
4. Once you have a unit of work 'staged', `commit` it to the local repository.
5. Continue to work 'adding' and 'committing' as necessary.
6. If you want to try out something new, `commit` your work and start a new `branch`
7. If you need to go back to an older version, use `restore`, `reset` or `revert` (depends on the situation…).

This is a physical folder on your computer

These are virtual 'places' git creates and tracks



Note that the 'staging area' and the 'local repository' aren't actually locations on your computer. They are virtual spaces that git represents in files it stores in the hidden **.git folder** in your 'workspace' – that is physical. To see the .git folder, you may have to turn on the 'show hidden files' flag in Windows (but there isn't much to see in that folder that is understandable…its git's internal way of tracking a tree structure of your work).

Step 1.1: Initializing a Workspace

This is really easy. Use Windows File Explore to create a folder called `MyProject` that you will work in. When you initialize the folder, it will become your workspace, and git will know to start tracking it. To initialize the folder, navigate to the folder in your terminal, and use:

'`PS-dmac>>`' is just my terminal prompt

Make sure you are in the right folder

```
PS-dmac>>cd MyProject                                 (changes the directory)
PS-dmac>>git init                                     (initializes MyProject)
initialized empty Git repository in C:/Users/user/MyProject/.git/
```

You will know you were successful with the initialization based on the message that is displayed, and if you see the `.git` folder in your workspace (keep Windows File Explorer open, and look inside `MyProject`). Don't forget – you may have to turn on the 'show hidden files' flag in Windows to see the .git folder.

Check to see what git is tracking using:

```
PS-dmac>>git status
...
On branch main
No commits yet
nothing to commit (create/copy files and use "git add" to track)
PS-dmac>>
```

This tells you two things:

- A default branch called `main` was created for you to work in
- You haven't added anything to the staging area, so you have nothing to commit.

In fact, you haven't got anything at all in the folder, so let's change that and create our first commit.

Step 1.2: Making your First Commit

Add a file called `names.txt` to your `MyProject` folder. **Do this in Windows File Explorer**, just like you normally would. Then go to the terminal and check to make sure its listed in your `MyProject` directory (use `ls` or `dir`). If it isn't you are probably not in the right directory (use `cd "<path>/MyProject"` to get there). If it is, do a `git status` to see what git thinks:

Untracked files show up in red

```
PS-dmac>>git status
On branch main
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        names.txt
nothing added to commit but untracked files present (use "git add" to track)>
```

This tells you that git sees the file in your Workspace, but you haven't 'staged' it yet, so its not being tracked. So, you have nothing in your staging area to 'commit' yet. Let's add it to the staging area, and then commit it:

Staging `names.txt`:

The `add` command stages files

Tracked files show up in green

```
PS-dmac>>git add names.txt
PS-dmac>>git status
On branch main
No commits yet

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

Now its tracking `names.txt`. It sees the new file as something ready to be committed. That means, the next time you `commit`, `names.txt` will be added to the local repository:

Notice git tells you how many things have been added and deleted from inside the file!

```
PS-dmac>>git commit -m "added names.txt"
[main (root-commit) 51e35e5] added names.txt
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 names.txt

PS-dmac>>git status
On branch main
nothing to commit, working tree clean
```

NOTE: when you include `–m` (or `--message`), that is a flag to the `commit` command that tells git that you are also including a message that describes the commit. **YOU SHOULD ALWAYS INCLUDE A MESSAGE** (in fact, when you don't, git will open up a default editor and ask you to provide one…write it, close the editor, and git will resume the commit process). **Craft your messages to be expressive – you will be glad you do when you start using them!**

Commit messages should describe the semantics of the unit of work you committed (you can 'stage' more than 1 file, and when you commit groups of files, write a message that describes what that group of files is or does).

Now that you have submitted a `commit`, git has stored a snapshot of what is in that commit in your local repository so that you can get it back if you want it (remember – the local repository is virtual…its really just a tree structure that git creates inside the .git folder that only git understands). You can track what you `commit` using `git log`:

```
PS-dmac>>git log --oneline

51e35e5 (HEAD -> main) added names.txt

PS-dmac>>
```

Here's your commit message!

We will explain this later!

This is a unique hash key. Git sets one for every commit

NOTE: `--oneline` provides a shortened version of the log so that each entry can be displayed on a single line

NOTE: The hash key is actually much longer (40 digits generated by the SHA1 cryptographic hash function)…these are only the first 7 digits (if you want to see the full code, use git log without the `--oneline` flag).

Let's add something to the file and commit again. Open the names.txt file in a text editor and add any two names to it – don't forget to save it. Check the status to see if git recognizes your changes:

```
PS-dmac>>git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   names.txt

no changes added to commit (use "git add" and/or "git commit -a")
PS-dmac>>tree clean
```

Red again… untracked changes!

Git recognized the changes! It tells you that you no longer have anything staged (because you committed everything that you had previously put in the staging area), and it also notes that there are unstaged changes to `names.txt` (it has been modified since the last time it was staged).

So now we stage it again and commit it. This time, lets use a <u>shortcut to combine these actions into one command</u>:

```
PS-dmac>>git commit -a -m "added two names"      (staging and committing in one step)
[main 27dd083] added names (total 2)
 1 file changed, 2 insertions(+)

PS-dmac>>git log --oneline
27dd083 (HEAD -> main) added names (total 2)
51e35e5 added names.txt
```

Now we see both commits

Notice the unique hash key

NOTE: we know what `–m` means, but note the use of `–a`. This tells git to `add` the file to the staging area and then `commit` (its such a common thing to do, git provides a shorthand notation)

So now we have two versions of `name.txt` stored in our local repository. Before we see what we can do with them, lets look at a conceptual model of how git tracks these versions.

You can think of git commits as pointers. They point to the most recent changes in your folder (a snapshot of the content you staged), but they also point to the previous commit. This way, each commit is attached to the new stuff just committed, plus all the old stuff that came before it:

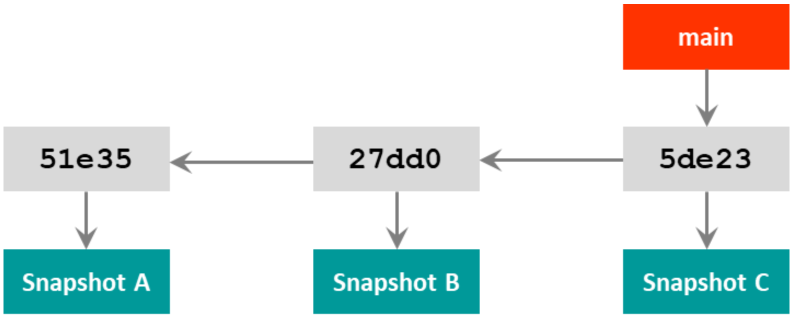


Figure 1: a conceptual view of Git: commit `27dd0` points to a snapshot of stuff that was staged when the commit was made, and to the previous commit, `51e35`.

We also see a pointer called 'main' in diagram 1. This represents the main branch that git created for us when we initialized our workspace. A git branch is also just a pointer to a commit. We will see how to create more branches later on, but for now, we should simply note that each time we make a commit, git moves the branch pointer to that that commit. In doing so, the 'main' branch actually points to a chain of commits (the current commit, and all the ones that came before it) – You can see why we call it a branch!

Let's add one more commit to our repository, so we have some stuff to move around in the next step:

- First - Leave 1 space, and add 2 more names to the names.txt file in our MyProject workspace.
- Then – add a README.md file to the workspace (md files are markdown files…we will learn about what those are later)

Create a txt file and then change the extensions to md

When you are finished making these changes, stage and commit them:

```
PS-dmac>>git status                                         (before staging)
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   names.txt

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

no changes added to commit (use "git add" and/or "git commit -a")

PS-dmac>>git add .                                    (staging – note the .)
PS-dmac>>git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   README.md
        modified:   names.txt
```

Notice that we staged more than 1 file at once using the '.' Notation (see more about this below)

continued…

```
PS-dmac>>git commit -m "updated names (4 names total) and added readme"
[main 5de2388] updated names (4 names total) and added readme
 2 files changed, 5 insertions(+)
 create mode 100644 README.md

PS-dmac>>git status
On branch main
nothing to commit, working tree clean
```

NOTE: You can specify a specific file to stage when using `add`, or you can specify to stage all of the files and modifications to files in the current directory (and its children) using the '`.`' notation. This won't include deletions…you would have to use `git rm <filename>` specifically for those. Alternatively, you can use `git add -A` to stage all changes (new files, modified files, and deletions). You also have the option to use `git add -u` to stage modifications and deletions, but not new files. Here is a quick summary:

	New Files	Modifications	Deletions
`git add -A`	YES	YES	YES
`git add .`	YES	YES	NO (use `git rm <fileneame>`)
`git add -u`	No	YES	YES

Step 1.3: Simple ways to Roll back and Undo things

Keeping a history of your work wouldn't be useful unless you could go back to older versions, to look at them, to use them as a starting point for something new, or to stop what you are doing and start again. All of these things are possible with git.

First lets look at some simple ways to undo:

- To restore uncommitted stuff in your workspace, use `git restore <filename>`
- To reset committed stuff to an older version, use `git reset <hash> --soft/mixed/hard`
- To undo changes made in a particular commit, use `git revert <hash>`

IMPORTANT NOTE: Be careful when you are rolling back. You don't want to lose stuff that you might still need. In general, `revert` is safer than `reset` because it maintains your history chain, and `reset --soft` is safer than `reset --hard` because it maintains your current workspace (it doesn't rollback your workspace).

Example using restore: Let's say you accidentally deleted the last two names in names.txt. You can get them back before you commit by restoring the file:

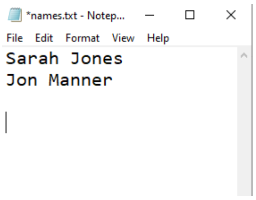
```
PS-dmac>>git restore names.txt
```

This works for deleted files too!

Use `git restore --p <filename>` to pick lines in a file to restore
Use `git restore .` to restore everything in the working directory

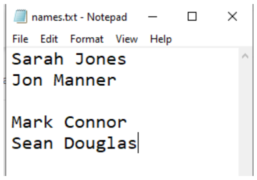
Note: You can also use
`git restore --source <hash> <filename>` to bring back an old version

Before Restore



```
Sarah Jones
Jon Manner

```

After Restore



```
Sarah Jones
Jon Manner

Mark Connor
Sean Douglas
```

Example using reset: Let's say you thought you needed `abadfile.txt` and `anotherbadfile.txt` so you created them, added them to staging and committed them. Then you decided you weren't sure if you needed them and wanted to reverse the commit. You can do this using:

```
PS-dmac>git log --oneline                                  (before resetting)
03ee5de (HEAD -> main) added some bad files
5de2388 updated names (4 total) and added readme
27dd083 added names (total 2)
51e35e5 added names.txt

PS-dmac>>git reset --soft 5de2388
PS-dmac>>git log -oneline                                   (after resetting)
5de2388 (HEAD -> main) updated names (4 total) and added readme
27dd083 (added names (total 2)
51e35e5 added names.txt
```

Bad Files committed

NO Bad Files committed

Reset can be:
`--soft`: resets to the hash commit
`--mixed`: resets to the hash commit and its staging area
`--hard`: resets to the hash commit, its staging area and its workspace

IMPORTANT NOTE: Be careful when using `--hard`: you will lose what's in your workspace because you are resetting it to what was in your workspace at the time of the commit you are resetting to.

Try it, to get rid of the bad files!

Let's take a look at what is happening when we reset (and why it can be dangerous). When we reset, we move the main branch pointer back to the commit we specify, and unlink everything that came after it. That stuff that came after it is not attached to any branch now, and will be deleted when git automatically cleans up. Thus, every commit in the branch that comes after the one that we reset to is lost forever.

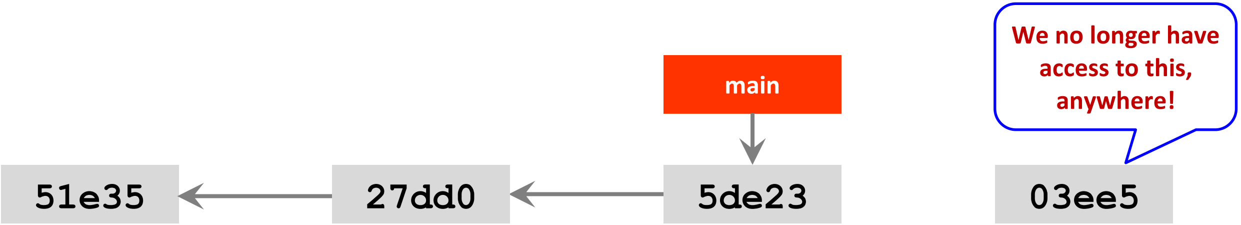


Figure 2: Git Reset: We `git reset 5de2388 --hard` to start again from that commit. Everything that we did after that commit is gone forever. To keep the workspace but lose the commits, use `git reset 5de2388 --soft`

`Revert` works differently from `reset` and is therefore safer. `Revert` maintains your history. `Revert` actually commits a new set of changes that undoes a commit in your history. The original commit remains intact, but somewhere later in the branch, another commit reverses those changes.

Example using revert: Let's add another few commits and then undo one:

```
PS-dmac>git log --oneline                            (after 2 more commits)
4fede1a (HEAD -> main) Added a campuses.txt file
9d8d795 Added a project description to README
5de2388 updated names (4 total) and added readme
27dd083 added names (total 2)
51e35e5 added names.txt
                                                           Continued...
```

New commits

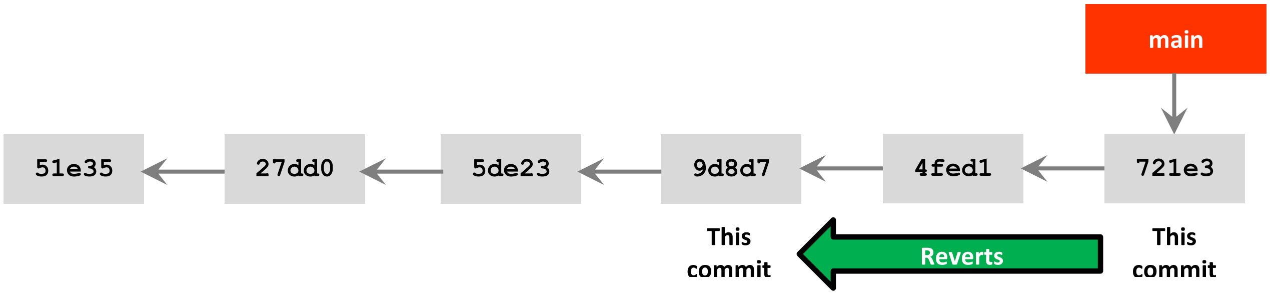


Figure 3: Git Revert: We `git revert 9d8d795`. That commit still exists in our history, but a new commit undoes what it did in our work (so we no longer have the project description in README which was committed in `9d8d7`).

```
PS-dmac>>git revert 9d8d795
[master 721e234] Revert "Added a project description to README"
 1 file changed, 1 deletion(-)

PS-dmac>>git log --oneline
721e234 (HEAD -> main) Revert "Added a project description to README"
4fede1a Added a campuses.txt file
9d8d795 Added a project description to README
5de2388 updated names (4 total) and added readme
27dd083 added names (total 2)
51e35e5 added names.txt
```

Note that git automatically opens your editor and provides a default message for you to adjust if you want to

For complete clarity, lets look at the full log entry for the revert commit:

```
PS-dmac>>git log
commit 721e23434a7963df7948d0eee7b75722a2d9458f (HEAD -> main)
Author: dmac <dmac@unb.ca>
Date:   Thu Sep 30 15:59:39 2021 -0300

    Revert "Added a project description to README"

    This reverts commit 9d8d79585fb7d760bf9c49c6f5bd915982245407. 9d8d795 Added
    a project description to README
```

One last note about undoing. If you make a mistake in the last commit you made, you can change it using `git commit –amend`. This is really useful if you want to change your commit message.

Example using commit –amend: Let’s add a commit and then change its message:

```
PS-dmac>>git commit -a -m "Changed description"
PS-dmac>>git log --oneline
ebb7342 (HEAD -> master) Changing the project description
...
PS-dmac>>git commit --amend -m "Changing the description in README"
PS-dmac>>git log --oneline
506c673 (HEAD -> master) Changing the description in README
...
```

Commit with vague message

Amending the message

So far we have explored simple scenarios where we wanted to roll back. We consider these scenarios simple, because the things we were undoing were independent of any commits that came after. When this is not the case, undoing is much more challenging and to understand that, you need to learn about branching and merging.

Step 1.5: Branching and Merging

One of the most useful things we can do with git is point to any commit in our history and start a new branch. This is really useful when developing software because you can keep a main branch, and diverge from it to figure out how to code a new feature (or a hot fix), and once you have it figured out, you can merge back to your main branch. When you do this, you don't have to worry about messing up code that already works.

Let's start by simply looking at an old commit using `checkout <hash>`. **For this part, we will assume we reset to `5de2388`, so we are back to a shorter log (you should do that).** What if I wanted to see what the names list looked like before my last update? I could checkout an old commit and look at it:

```
PS-dmac>>git log --oneline
5de2388 updated names (4 total) and added readme
27dd083 added names (total 2)
51e35e5 added names.txt

PS-dmac>>git checkout 27dd083
Note: switching to '27dd083'.
You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:
  git switch -c <new-branch-name>
Or undo this operation with:
  git switch -
HEAD is now at 27dd083 added names (total 2)
```

This one looks useful

Unless you use `checkout` in tandem with creating a new branch, you can't do much with what you are looking at. That is what this message is telling you. We will learn more about that later.

Open `names.txt` and take a look at its contents – those last two names you added with commit `5de2388`, are gone. You are looking at the version that existed when you made commit `27dd083`!

IMPORTANT NOTE: Because `checkout` changes what's in the workspace, you shouldn't checkout without committing what's in your workspace before you do – otherwise when you go back to where you were, everything that wasn't committed will be lost.

You can also use `git stash` instead of committing, but we don't have time to cover stashing in this lab.

Git usually prevents you from checking out before you commit (it aborts the checkout when you have uncommitted changes) but you should be mindful of it just in case

When you are done looking around, you can return to where you left off, simply by checking out that branch:

```
PS-dmac>>git checkout main
Previous HEAD position was 27dd083 added names (total 2)
Switched to branch 'main'
```

Look in your workspace. Is everything back to where you left it (4 names and a README)?

What's really going on when we checkout? For that, we need to understand what HEAD is. Remember this?

```
PS-dmac>>git log --oneline
5de2388 (HEAD -> main) updated names (4 total) and added readme
27dd083 added names (total 2)
51e35e5 added names.txt
```

You guessed it – HEAD is really just a pointer. It's the pointer that tells git what branch to look at. Our log above tells us that HEAD is pointing to main (ie its 'attached to main').

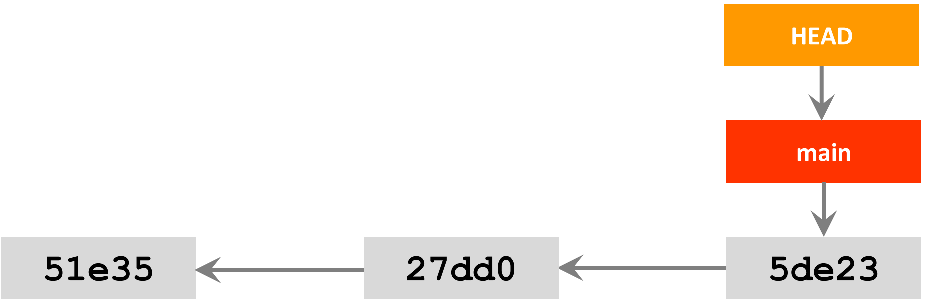


Figure 4: HEAD is a pointer that tells git what branch you are currently on.

When we checkout an old commit, git moves HEAD to that commit. But if we don't start a new branch at that commit, then HEAD is detached:

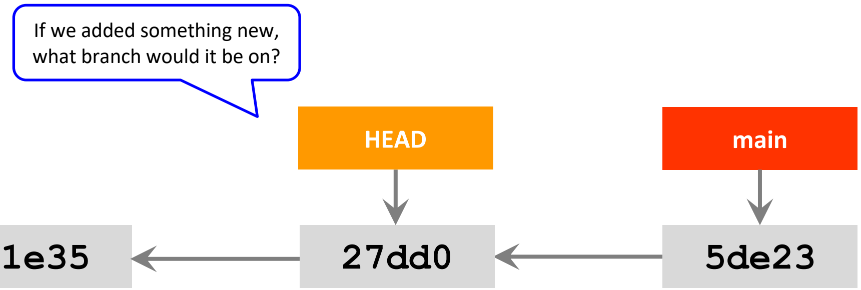


Figure 5: HEAD is pointing to an old commit, but is detached from any branch

HEAD isn't pointing to a branch!

```
PS-dmac>>git checkout 27dd083
Note: switching to '27dd083'.

You are in 'detached HEAD' state. ...
HEAD is now at 27dd083 added names (total 2)

PS-dmac>>git log --oneline
27dd083 (HEAD) added names (total 2)
51e35e5 added names.txt
```

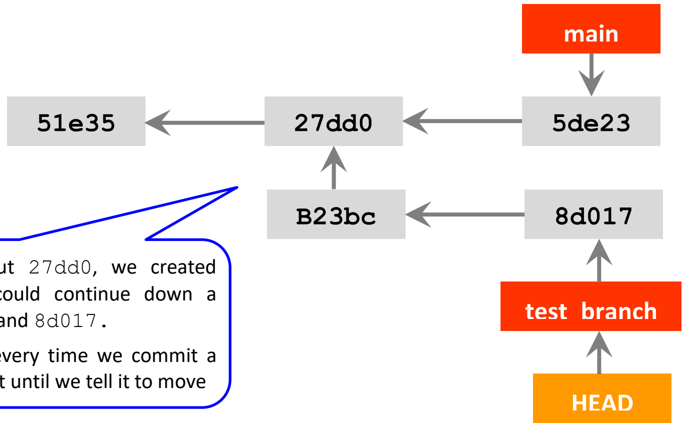
That's why we can look around when we checkout an old commit, but we can't do anything, unless we start a new branch at that commit so that the HEAD has a branch to point to. Fortunately, it easy to do this (either using branch command or checkout command…we will use checkout here, and explore branch in another step):

```
PS-dmac>>git checkout 27dd083
Note: switching to '27dd083'.

You are in 'detached HEAD' state. ...
HEAD is now at 27dd083 added names (total 2)

PS-dmac>>git checkout -b test_branch
PS-dmac>>git log --oneline
27dd083 (HEAD -> test_branch) added names (total 2)
51e35e5 added names.txt
```

This is from our previous Step

We can create a new branch and check it out at the same time using the -b flag with checkout

Now we can continue along making changes on test_branch



Here, when we checked out 27dd0, we created test_branch so that we could continue down a different path adding b23bc and 8d017.

The branch moves forward every time we commit a change, and HEAD stays with it until we tell it to move

Now we have access to 5de23 through the main branch, and b23bc and 8d017 through the test_branch...
We have two different versions of our work!

Figure 6: HEAD is pointing to a new branch that started from an old commit

Let’s assume commit b23bc added 2 more names to names.txt and commit 8d017 added a new file called cohorts.txt. Go ahead and make these changes and commit them – Be sure you choose to add names that you haven’t used yet, and make two separate commits (the goals of the changes are semantically independent).

```
PS-dmac>>git checkout test_branch
Switched to branch 'test_branch'

PS-dmac>>git log --oneline
27dd083 (HEAD -> test_branch) added names (total 2)
51e35e5 added names.txt

PS-dmac>>git commit -a -m "added 2 more names (4 in total)"
[test_branch b23bcc4] added 2 more names (4 in total)
 1 file changed, 5 insertions(+), 1 deletion(-)

PS-dmac>>git add cohorts.txt
PS-dmac>>git commit -m "Added cohorts.txt"
[test_branch 8d01719] Added cohorts.txt
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 cohorts.txt

PS-dmac>>git log --oneline
8d01719 (HEAD -> test_branch) Added cohorts.txt
b23bcc4 added 2 more names (4 in total)
27dd083 added names (total 2)
51e35e5 added names.txt
```

Remember – we are on test_branch

This is the test_branch log

So now let’s say you have decided you like what you tried in test-branch and you want to include those changes in your main branch. You can use merge to do this, but its complicated:

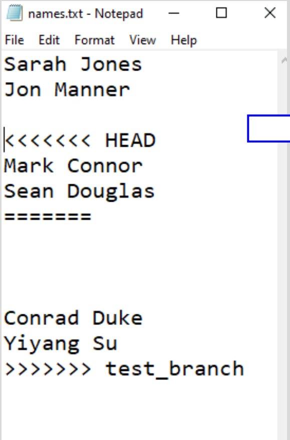
Start in the branch you want to merge to

```
PS-dmac>>git checkout main

PS-dmac>>git merge test_branch
Auto-merging names.txt
CONFLICT (content): Merge conflict in names.txt
Automatic merge failed; fix conflicts and then commit the result.
```

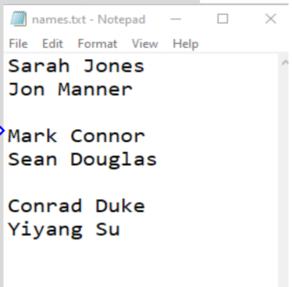
Even simple changes can result in conflicts…if the lines aren’t identical in the file, there will be conflicts.

Git will mark-up your file to show you what needs to be resolved.

Make the changes and and save the file



```
Sarah Jones
Jon Manner

<<<<<<< HEAD
Mark Connor
Sean Douglas
=======



Conrad Duke
Yiyang Su
>>>>>>> test_branch
```



```
Sarah Jones
Jon Manner

Mark Connor
Sean Douglas

Conrad Duke
Yiyang Su
```

This can look however you want the merge to look

`git status` provides pretty good instructions for what to do

```
PS-dmac>>git status
On branch main
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Changes to be committed:
        new file:   cohorts.txt

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   names.txt

PS-dmac>>git add names.txt
PS-dmac>>git status
On branch main
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:
        new file:   cohorts.txt
        modified:   names.txt

PS-dmac>>git commit
[main 61f4024] Merge branch 'test_branch'
PS-dmac>>git status
On branch main
nothing to commit, working tree clean.txt
```

Note also that git tells you how to get out of this merge

Git will provide the merge message for you

we won’t be experts in merging after this one example…but the more you do it, the better you will get at it. Also, this simple example doesn’t really express the utility of merging, but imagine merging two branches with lots of files, mostly idendependent, with just a few conflicts like the one we just reviewed. Its nice to have a system that automates the conflict search.

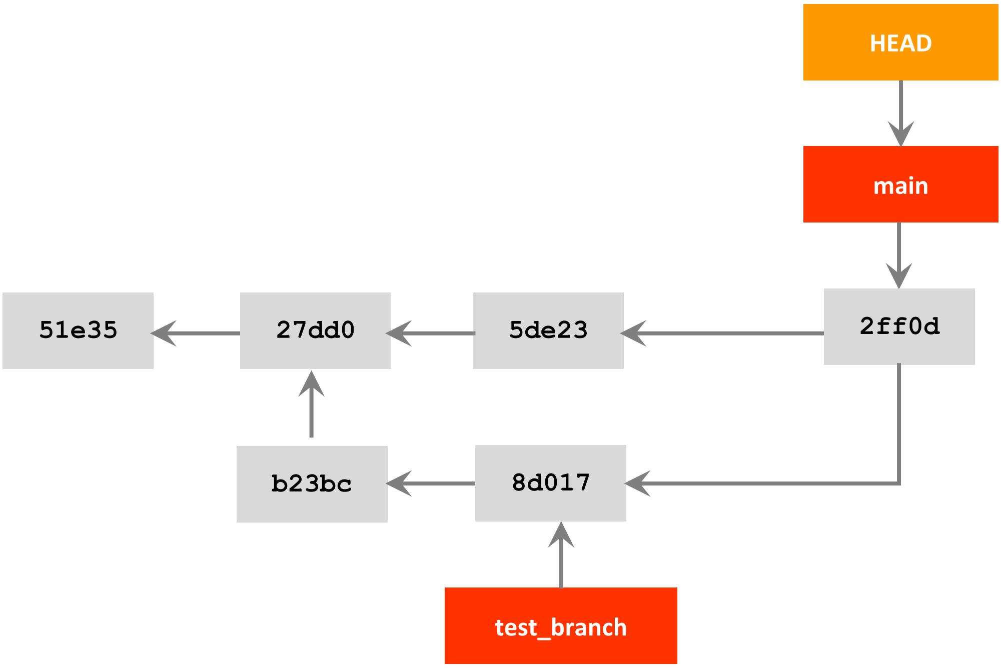


Figure 6: Two Branches merged, with `HEAD` pointing at `main` (if we want, we can delete `test_branch` now)

Another thing to note about merging, if you start another branch from HEAD, when you merge, git will simply do a fast-forward merge. That is, it will simply move HEAD forward. For example, if we create a new branch called testff_branch and use it to delete one of the names in names.txt, this is a simple merge with not conflicts that can continue along the main branch.

```
PS-dmac>>git checkout -b testff_branch
Switched to a new branch 'testff_branch'

PS-dmac>>git commit -a -m "removed a name"
[testff_branch 83154dd] removed a name
 1 file changed, 1 insertion(+), 1 deletion(-)

PS-dmac>>git checkout main
PS-dmac>>git merge testff_branch
Updating 2ff0d7e.. 83154dd
Fast-forward
 names.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)d names.txt

PS-dmac>>git log --oneline
83154dd (HEAD -> master, testff_branch) removed a name
2ff0d7e Merge branch 'test_branch'
138153f (test_branch) adjusted spacing in names.txt
8d01719 Added cohorts.txt
b23bcc4 added 2 more names (4 in total)
5de2388 updated names (4 total) and added readme
27dd083 added names (total 2)
51e35e5 added names.txt
```

This is the simple fast-forward merge

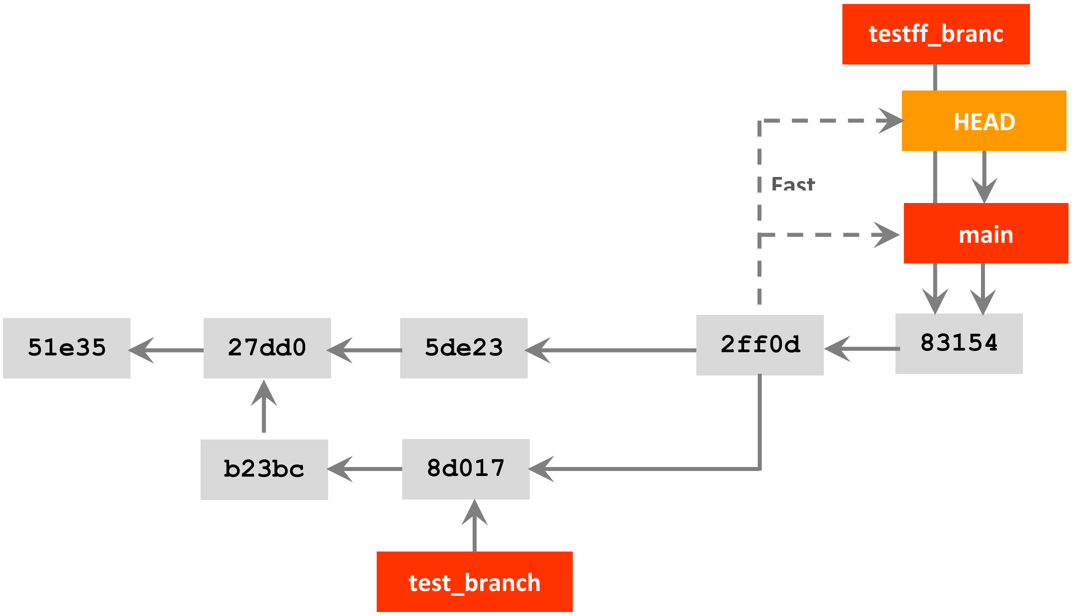


Figure 7: To merge testff_branch, simply fast-forward main and HEAD.

DELIVERABLE A

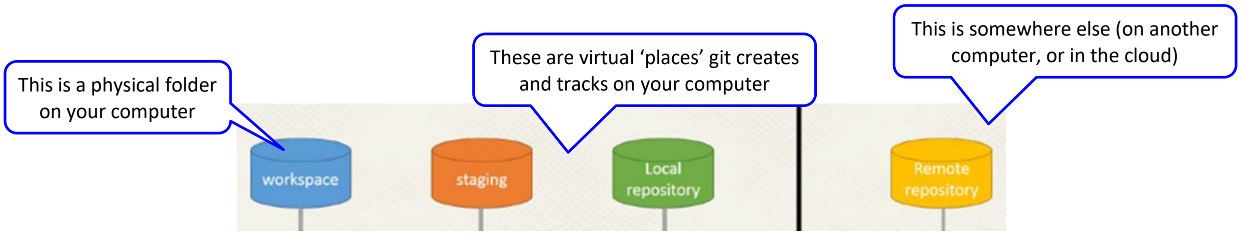
That ends part 1 of the lab. Don't worry, Parts 2 and 3 are much shorter. Here is a list of questions to answer for deliverable A. Be specific in your answers to questions about commands (include relavent flags and use <> to denote other arguments):

1. When you make a commit in git, which area (Workspace, Staging, or Local Repository) is the snapshot taken from, and which area is the commit stored?
2. What command would you use to make a commit and include a message
3. What command would you use to add if you knew you had lots to stage, but only new files and modifications?
4. What command would you use to fix a commit message that you just submitted?
5. What's the difference between revert and reset?
6. Provide an example of a scenario that can lead to a detached HEAD

PART 2: WORKING WITH REMOTES

The basic workflow for working *locally* doesn't include collaborating with others. To do that, you need a way to share your repository. Git is designed to be a distributed version control system, which means everyone who works on the project has a *local* copy of the repository. But to keep that copy updated, everyone pushes their changes to a common *remote* repository and pulls from that to update their *local* copy. With an updated copy, you can track changes you make *locally*, and when you have something complete, push your version back up to the common *remote* repository. Here are the basic steps that allow you to add a *remote* repository to your workflow:

1. Initialize a *local* repository and `commit` stuff to it (like we did in Part A)
2. When you are ready to start sharing, create an empty *remote* repository somewhere like github
3. `Add` the *remote* repository to your *local* one
4. Create *remote* `branches` and `push` your *local* `branches` up to them
5. If others `clone` your remote repository to start to contribute to the project, `pull` branches down from the *remote* to `merge` their changes to your *local* repository
6. Continue to work, `committing`, `pushing`, and `pulling` as necessary



This is a physical folder on your computer

These are virtual 'places' git creates and tracks on your computer

This is somewhere else (on another computer, or in the cloud)

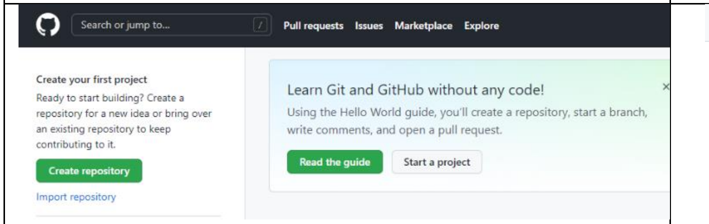
It may be useful to note that there is nothing special about a central remote repository. Since every local version is a copy of it, the only thing it offers is a common share space. However, some cloud hosting services (like github) build features into their services beyond allowing you to share you repository. We will explore some of these services in PART 3 of this lab. The focus of this part of the lab will be working with remote repositories.

Step 2.1: Creating a remote repository in GitHub to share our local repository

To start, we are going to create an empty repository in GitHub, and connect it to the local repository we worked on in part 1. Log into GitHub and create a repository:

- Call it <u>MyProject</u>, make it <u>Private</u>, and <u>Don't include any initialization files</u>
- STOP before you follow any of the instructions GitHub provides…we will do that in the steps that follow (which will align with the suggestions GitHub provides)

What you will see when you first log in:



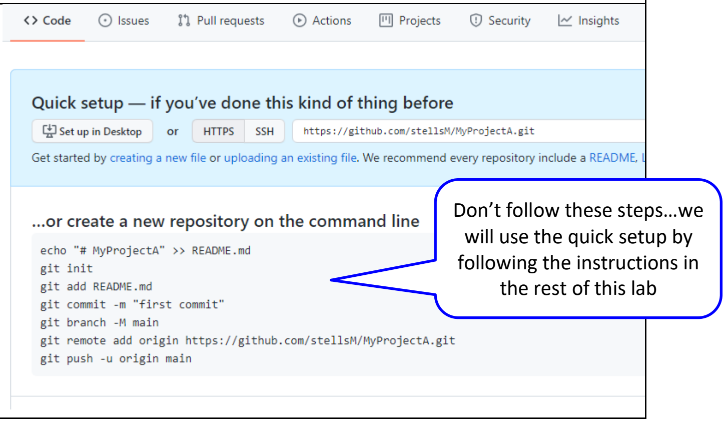
Create your first project

Ready to start building? Create a repository for a new idea or bring over an existing repository to keep contributing to it.

Learn Git and GitHub without any code!

Using the Hello World guide, you'll create a repository, start a branch, write comments, and open a pull request.

What you will see if you successfully create an empty repo



Quick setup — if you've done this kind of thing before

https://github.com/stellsM/MyProjectA.git

Get started by creating a new file or uploading an existing file. We recommend every repository include a README, L

...or create a new repository on the command line

```
echo "# MyProjectA" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/stellsM/MyProjectA.git
git push -u origin main
```

Don't follow these steps…we will use the quick setup by following the instructions in the rest of this lab

Once you have your remote repository ready, return to the terminal. Before we add the remote to your local repository, we will clean it up a bit. Since we already have everything merged, lets delete our test branches:

```
PS-dmac>>git checkout main
PS-dmac>>git branch -d test_branch
PS-dmac>>git branch -D testff_branch
```

You may have to use the -D instead of -d if git doesn't recognize the fast-forward as fully merged.

```
PS-dmac>>git branch -a
* main
```

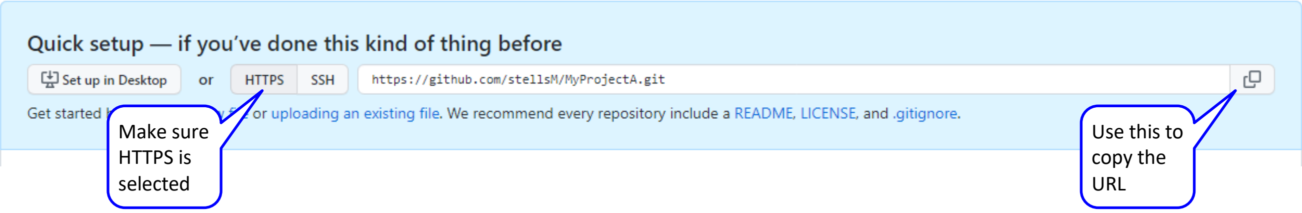
`git branch --all` (or `-a`) lists your branches, and puts a * on the one you have

you can also use: `git branch <branchname>` to create branches

Just a reminder of what we've done

```
PS-dmac>>git log --oneline
506c673 (HEAD -> main) Changing the description in README
83154dd removed a name from names.txt
2ff0d7e Merge branch 'test_branch'
8d01719 Added cohorts.txt
b23bcc4 added 2 more names (4 in total)
5de2388 updated names (4 total) and added readme
27dd083 added names (total 2)
51e35e5 added names.txt
```

Now to add the remote repository. First go back to your GitHub page to copy your remote URL (we will use https)



Quick setup — if you've done this kind of thing before

https://github.com/stellsM/MyProjectA.git

Get started … or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore.

Make sure HTTPS is selected

Use this to copy the URL

Then use `git remote` to `add` it:

```
PS-dmac>>git remote add origin https://github.com/stellsM/MyProject.git
```

We can specify an alias for the long URL. In this instance I used `origin`, which is the alias that git defaults to when it creates one

Now, whenever we need to refer to our remote repository, we can simply write `origin`

Now we have our remote repository connected to our local repository and we can refer to it by the alias `origin`. So, we can `push` our changes up to the remote repository using:

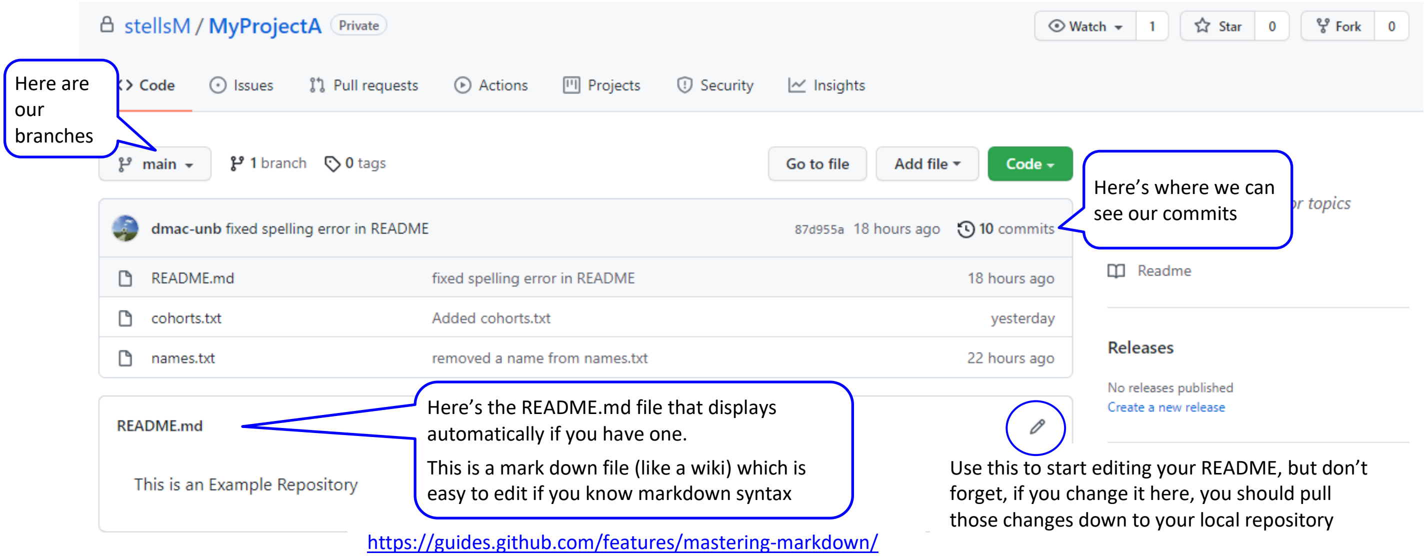
```
PS-dmac>>git push origin main
...
To https://github.com/stellsM/MyProject.git
 * [new branch]      main -> main
```

Now we have 2 versions of the `main` branch – the *local* `main` we are used to working on, and the *remote* `main`. To see these, we can use the branch command:

```
PS-dmac>>git branch --all
* main
  remotes/origin/main
```

Remote branches are red

We can't actually make changes directly to the remote branch from our terminal. We can only look at it (by checking it out) or update it by `pushing` to or `pulling` from the remote repository (more on `pulling` later). Let's check to see if our `push` made it into our remote repository. Refresh your view of your repository in GitHub. You should see something like this:



dmac-unb fixed spelling error in README 87d955a 18 hours ago 10 commits

README.md	fixed spelling error in README	18 hours ago
cohorts.txt	Added cohorts.txt	yesterday
names.txt	removed a name from names.txt	22 hours ago

README.md

This is an Example Repository

Here are our branches

Here's where we can see our commits

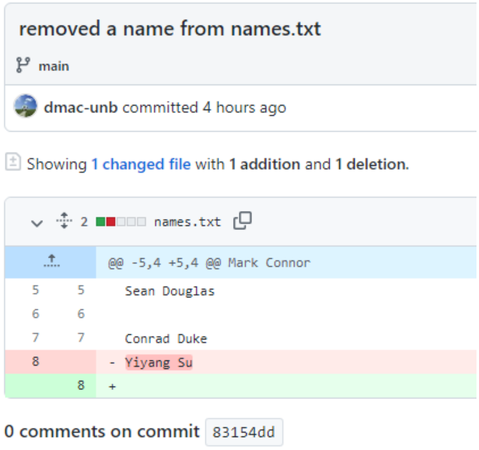
Here's the README.md file that displays automatically if you have one.

This is a mark down file (like a wiki) which is easy to edit if you know markdown syntax

https://guides.github.com/features/mastering-markdown/

Use this to start editing your README, but don't forget, if you change it here, you should pull those changes down to your local repository

Here is an example of what you see when you dig into your commits:



removed a name from names.txt

main

dmac-unb committed 4 hours ago

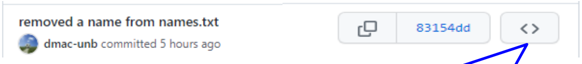
Showing 1 changed file with 1 addition and 1 deletion.

```
2 names.txt
@@ -5,4 +5,4 @@ Mark Connor
5    5     Sean Douglas
6    6
7    7     Conrad Duke
8        - Yiyang Su
     8   +
```

0 comments on commit 83154dd

Remember this one – in one of the test branches we removed a name and then merged that change. It ended up as commit `83154dd`.

One of the excellent services GitHub provides is a good visual depiction of what happened in each commit. The Red line 8 is the name that we deleted. The green line 8 is the line space that we added



If you click here in the list entry for this commit, you can see the entire repository at the time of the commit

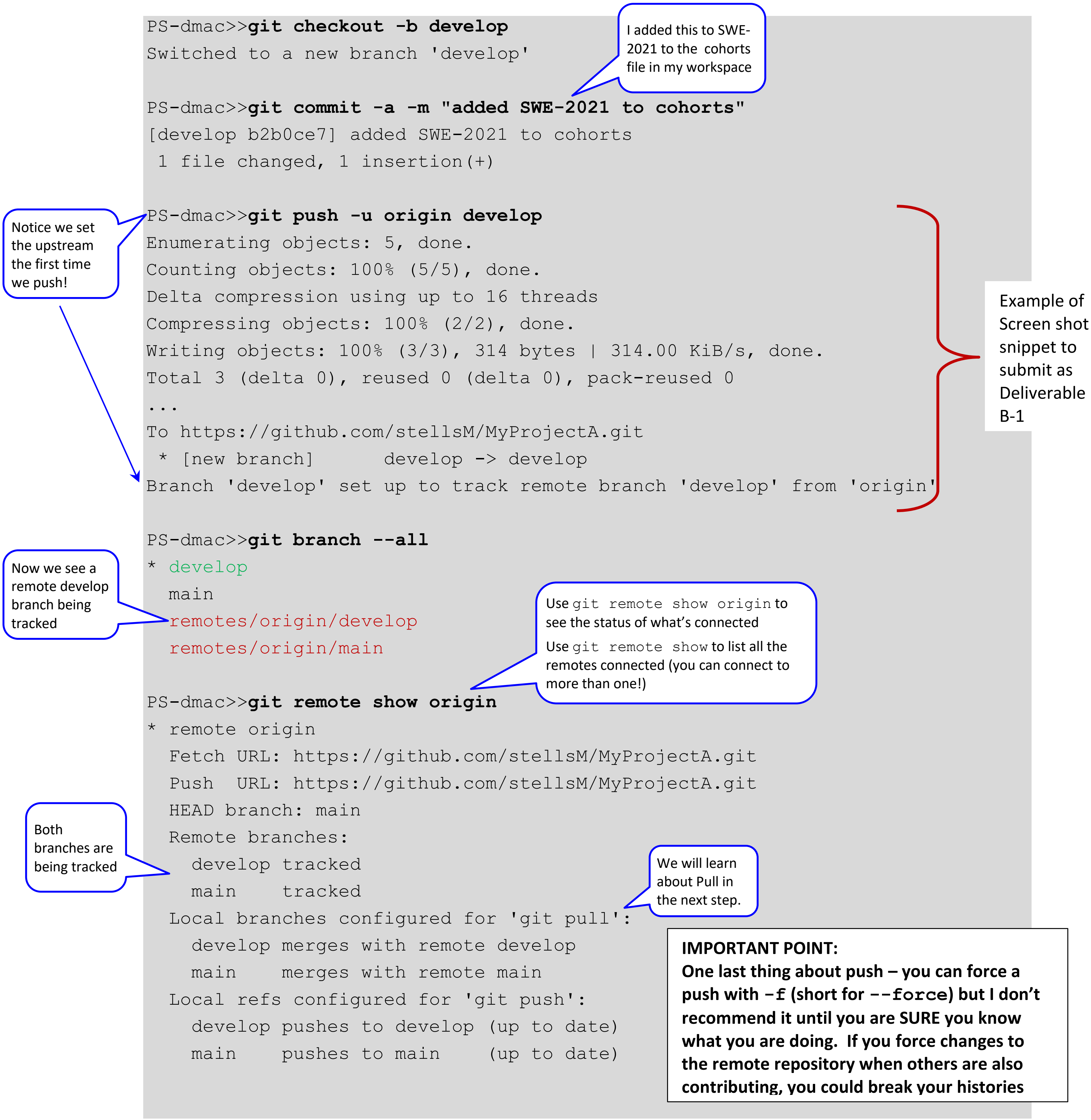
Lets get back to the terminal. There is one more thing that we should do to make sure we continue to `push` properly. We can continue to `push` by manually specifying which remote and which branch, but if we specifically specify what our local branch should be tracking in the remote, we can shorten this process. To specify this we need to set the `--set-upstream` flag (`-u` for short) when we `push`:

```
PS-dmac>>git push -u origin main                    (-u is short for --set-upstream)
Everything up-to-date
Branch 'main' set up to track remote branch 'main' from 'origin'.
```

The local main branch is now a 'tracking branch'

NOTE: Since there is nothing new to `push`, git tells us everything is up-to-date; but git also tells us that we have now set up our local `main` branch to track the `main` branch from the remote. Since we have 'connected' these two branches, now when we want to `push main`, we can just use `git push`.

Right now we only have 1 local branch in our repository (because we deleted all of the others). Let's add a new one and complete the process again to get it into our remote repository and tracking a remote branch. The entire process is depicted below:



```
PS-dmac>>git checkout -b develop
Switched to a new branch 'develop'

PS-dmac>>git commit -a -m "added SWE-2021 to cohorts"
[develop b2b0ce7] added SWE-2021 to cohorts
 1 file changed, 1 insertion(+)

PS-dmac>>git push -u origin develop
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 16 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 314 bytes | 314.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
...
To https://github.com/stellsM/MyProjectA.git
 * [new branch]      develop -> develop
Branch 'develop' set up to track remote branch 'develop' from 'origin'

PS-dmac>>git branch --all
* develop
  main
  remotes/origin/develop
  remotes/origin/main

PS-dmac>>git remote show origin
* remote origin
  Fetch URL: https://github.com/stellsM/MyProjectA.git
  Push  URL: https://github.com/stellsM/MyProjectA.git
  HEAD branch: main
  Remote branches:
    develop tracked
    main    tracked
  Local branches configured for 'git pull':
    develop merges with remote develop
    main    merges with remote main
  Local refs configured for 'git push':
    develop pushes to develop (up to date)
    main    pushes to main    (up to date)
```

I added this to SWE-2021 to the cohorts file in my workspace

Notice we set the upstream the first time we push!

Example of Screen shot snippet to submit as Deliverable B-1

Now we see a remote develop branch being tracked

Use `git remote show origin` to see the status of what's connected

Use `git remote show` to list all the remotes connected (you can connect to more than one!)

Both branches are being tracked

We will learn about Pull in the next step.

IMPORTANT POINT:
One last thing about push – you can force a push with `-f` (short for `--force`) but I don't recommend it until you are SURE you know what you are doing. If you force changes to the remote repository when others are also contributing, you could break your histories

Be sure to go back to github to check for the develop branch!

DELIVERABLE B-1

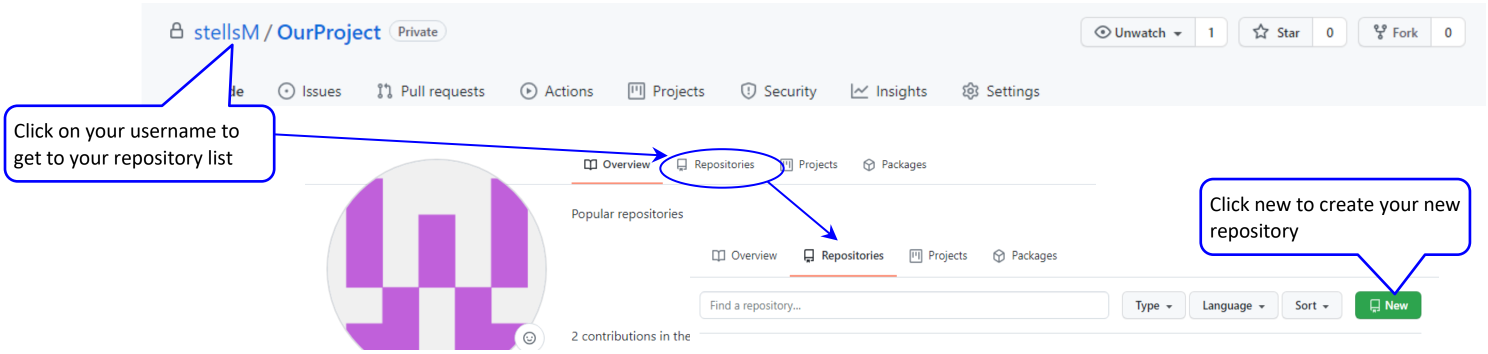
To submit Deliverable B-1: Take a Screen Shot of your push command and the results as delineated in the figure above. Make sure your repository name is included in the snippet.

Step 2.2: Creating a remote repository in GitHub for others to clone.

The main reason to push a repository to GitHub is so that you can share it with others who might want to contribute. In fact, in GitHub, if you set your repository to public, you have created an opensource project that anyone can contribute to. We won't go that far, but we do want to investigate how to share a repository with specific people so that they all can contribute along with you.

For this step **we will work in pairs** and start an entirely new repository. And this time, we will start with the remote repository!

- Find a partner
- **One of you** create a repository in Github – call it OurProject, make it private, and include a README.md



Click on your username to get to your repository list

Click new to create your new repository

If you add a file to your repository when you create it (as we just did by including the README.md), you have already made your first commit. Because of this, you don't get the instructions screen that GitHub provides for repositories with no commits and you go straight to a repository with 1 commit.

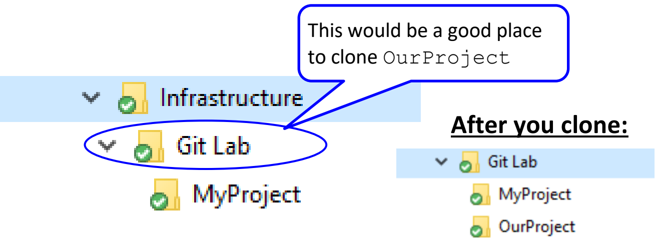
Next you need to set it up so that your partner has access too (remember its private, so you need to grant access for others to see it):

- Got to **Settings>ManageAccess>Add people** (you can search your partner's user-name to grant them access).
- Your partner needs to **accept the invite sent to their email address**. When they do, their status will switch from 'awaiting response' to 'collaborator'

Once you both have access to the repository, you are ready to create your local versions. You can do this on separate computers so you both have a local repository. It is really easy to create a local repository from a remote – you simply clone it (follow the advise below to do that).

In the terminal, navigate to the folder where you want the Workspace for your project to be (perhaps in the same folder where you put MyProject):

NOTE: Don't create the workspace…`git clone` will do that for you!



This would be a good place to clone `OurProject`

After you clone:

```
PS-dmac>>git clone https://github.com/stellsM/OurProject.git
Cloning into 'OurProject'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
```

Get this from the green Code button for your remote repository (in GitHub)

Using File Explorer, look in `OurProject` – it's a git Workspace now, so it should have a hidden `.git` folder. It should also have whatever was in your remote repository when you cloned it, which was a `main` branch with a `README.md` file.

Once you are sure you have cloned the repository, use the terminal to check what branches and remotes you have connected (don't forget to navigate to the Workspace…you are probably one folder up):

```
PS-dmac>>cd OurProject

PS-dmac>>git branch --all
* main
  remotes/origin/HEAD -> origin/main
  remotes/origin/main

PS-dmac>>git remote show origin
* remote origin
  Fetch URL: https://github.com/stellsM/OurProject.git
  Push  URL: https://github.com/stellsM/OurProject.git
  HEAD branch: main
  Remote branch:
    main tracked
  Local branch configured for 'git pull':
    main merges with remote main
  Local ref configured for 'git push':
    main pushes to main (up to date)
```

When you cloned, git automatically created a local main branch

Git also sets up all the tracking connections

NOTE: when you clone your repository, git automatically sets up most of the automated tracking for you!

Take the following steps in order (work together) to make sure everything is working:

- First – let Partner 1 add authors (your user names) to the README. Do this locally, and then push the change to the repository
- Second – let Partner 2 update their local repository by Pulling those changes from the remote:

```
PS-dmac>>git pull
...
From https://git                    Project
   0eca05f..5178                    origin/main
Updating 0eca05f..517826b
Fast-forward
 README.md | 3 ++-
 1 file changed, 2 insertions(+), 1 deletion(-)

PS-dmac>>git log --oneline
517826b (HEAD -> main, origin/main, origin/HEAD) added authors to README
0eca05f Initial commit
```

Git already set the upstream branch for this, so you don't need to specify origin or branch

Notice you have this commit now, even though you didn't create it

Check your local files to make sure README.md was actually changed. If it was, you have successfully updated your local repository.

Pushing and pulling works great when you are sitting side-by-side and coordinating who pulls what, when. But that isn't realistic, so its likely that by the time you have something to push to a remote, someone else may have already pushed, altering the remote's history. When this is the case, you need to pull what they've submitted down to your local branch so you can merge it with your changes before you push them. Let's try this in a simple scenario:

Take the following steps in order (work together) to make sure everything is working:

- First – let Partner 1 add a section to the README called 'Working Locally'. Do this locally, and then push the change to the repository. (Note –in markdown, sections start with a pound-sign: ## Working Locally).
- Second – let Partner 2 add a section to the README called 'Working with Remotes'. Do this locally, and then push the change to the repository. (Note –in markdown, sections start with a pound-sign: ## Working with Remotes).

For one of you this will go smoothly, for the other, there will be issues. If you try to push to a remote that has updates you don't have, there will be merging issues and git will access you to resolve them first. Here's the process:

```
PS-dmac>>git commit -a -m "added a Working with Remotes section to README"
[main e74aebf] added a Working with Remotes section to README
 1 file changed, 2 insertions(+)
```

I updated the README.md file with this addition

```
PS-dmac>>git push
To https://github.com/stellsM/OurProject.git
 ! [rejected]        main -> main (fetch first)
error: failed to push some refs to 'https://github.com/stellsM/OurProject.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Git explains the problem to you!

I pull the changes which tries to make the merge with my local branch

```
PS-dmac>>git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 700 bytes | 50.00 KiB/s, done.
From https://github.com/stellsM/OurProject
   517826b..03cd1dc  main       -> origin/main
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
```

But I have a merge conflict (at least we were expecting this)

I open README.md, see where git shows me the conflicts, and fix them so I am ready to add and commit

```
PS-dmac>>git add .
PS-dmac>>git commit
[main c2c5e25] Merge branch 'main' of https://github.com/stellsM/OurProject
```

Remember – in this instance, git will create my commit message for me, and open the editor for me to edit if I want.

Now I am ready to push again

```
PS-dmac>>git push
Enumerating objects: 10, done.
Counting objects: 100% (10/10), done.
Delta compression using up to 16 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (6/6), 608 bytes | 608.00 KiB/s, done.
Total 6 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), done.
To https://github.com/stellsM/OurProject.git
   03cd1dc..c2c5e25  main -> main
```

NOTE: Its always a good idea to pull before you push, just to make sure you are working with the most recent version of the branch you are pushing to.

One last point about working with remotes. The `pull` command is really a combination of two commands: `fetch` and `merge`. Using them in a 2 step process may make `pushing` to an updated remote easier to follow. Here the process again, this time using `fetch` and `merge`, rather than `pull`.

```
PS-dmac>>git commit -a -m "added content to Working with Remotes"
[main c4998e9] added content to Working with Remotes
 1 file changed, 4 insertions(+)

PS-dmac>>git push
To https://github.com/stellsM/OurProject.git
 ! [rejected]        main -> main (fetch first)
...
```

This works because I am just updating remote/main

```
PS-dmac>>git fetch
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 716 bytes | 47.00 KiB/s, done.
From https://github.com/stellsM/OurProject
   c2c5e25..9748e68  main       -> origin/main

PS-dmac>>git status
On branch main
Your branch is behind 'origin/main' by 1 commit, and can be fast-forwarded.
  (use "git pull" to update your local branch)
nothing to commit, working tree cleanPS-dmac
```

I can see from status that I am a commit behind on my local branch – I need to merge

```
PS-dmac>>git merge origin/main
Updating c2c5e25..9748e68
Fast-forward
 README.md | 4 +++-
 1 file changed, 3 insertions(+), 1 deletion(-)
```

Whoohoo, a straightforward merge!

Now I can push

```
PS-dmac>>git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 16 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 386 bytes | 386.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/stellsM/OurProject.git
   9748e68..c4998e9  main -> main
```

Example of Screen shot snippet to submit as Deliverable B-2

DELIVERABLE B-2

Add content to README.md file which lists some of the important concepts you have learned about git. You already have two sections started. You complete 1 section and let your partner complete the other. You may want to do this in a few commits – be sure to make at least 1 commit each from your local repository (ie push the change). We can see whose committing, so make sure you stick to your sections. You should have at least 5 points/section (that's all you need but more are welcomed, and only 1 *needs* to be pushed from a local repository…the rest can be edited within GitHub).

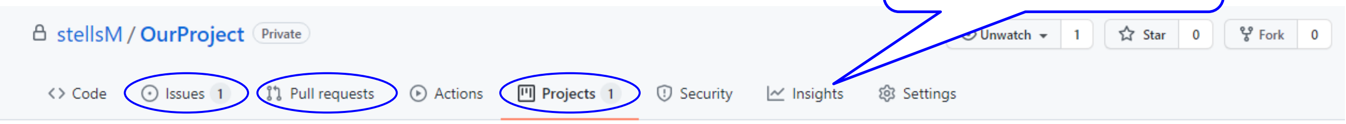
To submit Deliverable B-2: Add `allieGriffinn`, to your remote repository, and submit to D2L a screen shot of your terminal commands for 1 commit that you made locally and pushed to the remote (be sure that screen shot tells us the name of your repository…submit this individually)

PART 3: GitHub – added Features

GitHub has a lot of extra features which are mostly in place to help a software development effort move forward collaboratively, but can be useful for other types of projects to. Obviously, at its core, it's a version control system. But is also has other useful features. In this part of the lab, we will look at one of them:

1. Tracking Tasks as Issues (using Projects)

This *may* activate when you get your extra student features

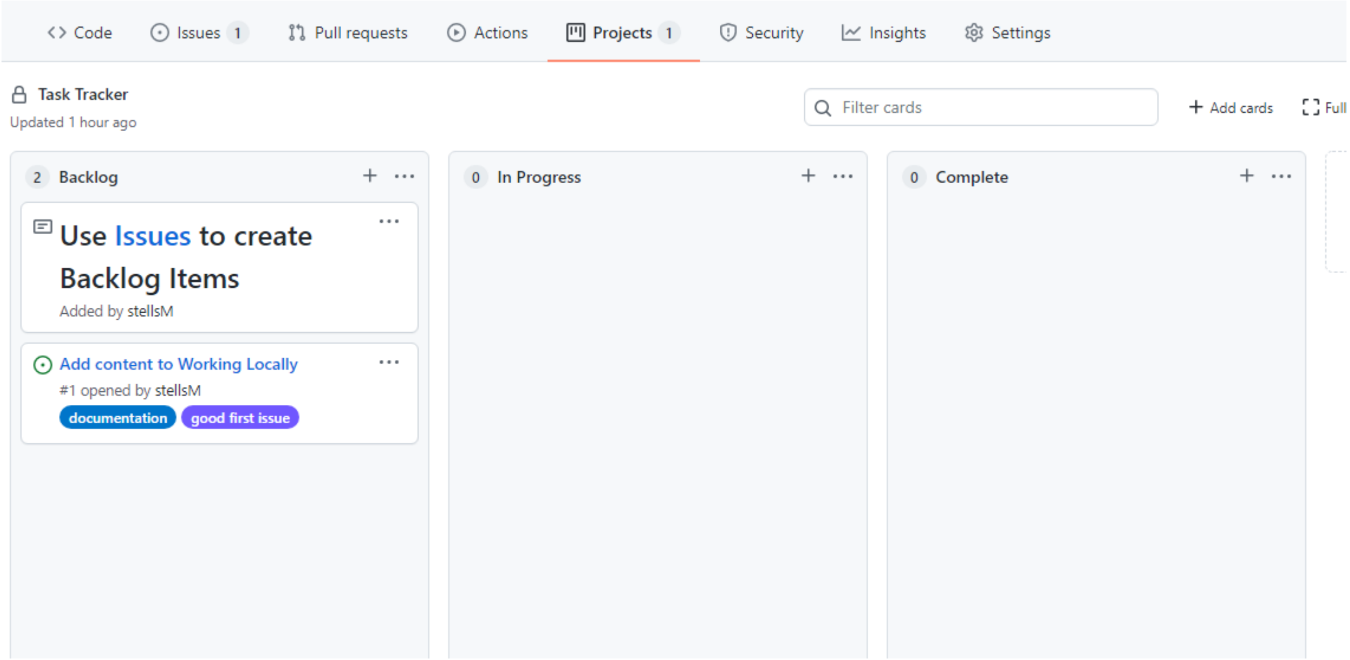


Another really useful part of GitHub, is you can add a wiki to your repository. Once you learn markdown, it's a really convenient way to share documentation among collaborators (and maintain a log about what you learn while navigating around git and GitHub when you are first starting out). We won't look at it because its an extra feature that comes once you confirm you're a student, but if you are interested, once you get your extra features, go to your Repository Settings, scroll down through the Features, and turn the Wiki on. (Its also available if you make the repository public, but then everyone can see it)

For this step, work with your partner, and use your common repository

Step 3.1: Issue Tracking with Project

A nifty way to track tasks on a project is to exploit the issue tracker and Project services in GitHub.



Use Issues to create Backlog Items
Added by stellsM

Add content to Working Locally
#1 opened by stellsM

Kanban boards are visual displays of tasks on cards, with columns that express where along the process (from Created to Completed) a task is. You can create a Kanban board with some simple automation to help you track tasks in GitHub. The figure above shows a Kanban board with 3 stages:

- **Backlog:** Where tasks are put when their identified (it's a backlog of things that have to get done)
- **In Progress:** Where tasks are put once they are started
- **Complete:** Where tasks are put one the are complete.

Create a Kanban board these 3 lists. Add a card that look like the first card in the Backlog list above (you will have to figure out some markdown to do that).

You could continue to add cards this way, but we are going to take another approach. First, click the ellipsis menu for the Backlog List to Manage Automation. Make the list a To Do list, and set Move Issues Here when they are Newly Added and when they are Reopened. Ignore the Pull Requests for now.

You won't need any automation on the In Progress list, but you will on the Complete list. Set that to a Done list, and set Move Issues Here when they are Closed.

Now, navigate to the issues page and create two new issues – one for 'Adding Content to Working Locally', and another for 'Adding Content to Working Remotely'. Put labels on these issues that are appropriate (e.g documentation), and make sure you add the Project to each issue. When you do this, these issues should show up in your backlog – check.

When you figure out whose adding content to which section, add assignees to the tasks. When you start your task, manually move it off the backlog list, onto the In-progress list. When you are done your task, close it through the issues page, and see if it moves to Done on the Projects page.

DELIVERABLE B-3

Your board will be reviewed along with the other part of deliverable B-2.

Summary of Submission Components:

☑ Text file that answers 6 questions in Part A

☑ Screen shot depicting push to your individual repository

☑ Screen shot depicting push to your shared repository

☑ TA Invitation to your shared repository which has (only 1 partner sends the invite):

- specified readMe content from both partners
- a GitHub Project set up