Git Lab 3
Undoing changes

1 Commands covered

Undoing staged and unstaged changes	Deleting tracked files

`git restore`
`git restore --staged`
 	
GitTower git restore command
 	
GitConnected git restore command
	`git clean --force`
	`git clean --force -X`
	
	Atlassian git clean command
	
	ServerSide git clean command

`git checkout --`
`git checkout HEAD`
 	
Atlassian git checkout command
 	
GitTower git checkout command	

`git reset --mixed`
`git reset --hard`
 	
Atlassian git reset command	

Reverting to older commit content	Updating commits
`git reset --hard commit-ref` StackOverflow reset command	`git commit --amend -m` Atlassian git commit command
`git revert HEAD` Atlassian git revert command CloudBees git revert command	

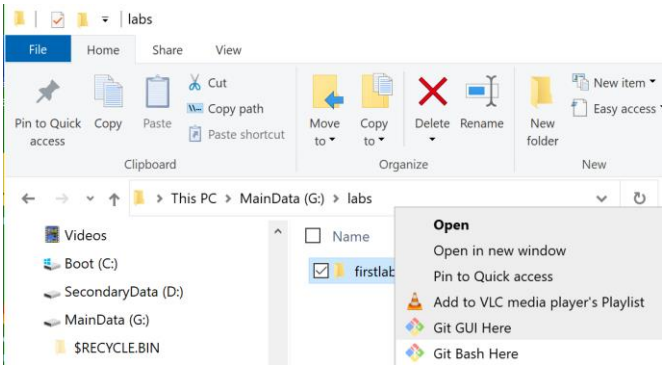
Getting reference logs	Saving and retrieving from the stash
`git reflog master` Atlassian git reflog command W3Docs git reflog command	`git stash` Atlassian git stash command OpenSource git stash command

2 Lab setup

Make sure you have a suitable text editor installed for your OS.

We will continue with the Git project from the previous Lab 1 in the working directory `labs\firstlab`. If are using a directory with a different name, substitute that name into all the commands that use this directory name in this lab session.

If you are working with Linux / MacOS, open a terminal shell to type in your Git commands.
If you are working with Windows, we will be typing in the commands via Git Bash. Right click on the directory and select Git Bash Here from the context menu to open the Bash shell.



The Git commands to type are listed after the $ prompt and their output (or relevant parts of their output) will be shown as well.

3 Undoing unstaged changes

Make the following additions to these files and save:

```
4: honda
5: subaru
```

cars.txt

```
4: dog
5: mouse
```

animals.txt

Check the status of the working directory to confirm the unstaged changes:

```
$ git status
On branch master
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:   animals.txt
        modified:   cars.txt
```

The simplest way to undo an unstaged change is to simply undo the changes in the affected files and save them (using the undo functionality in the text editor). Verify this for yourself.

If you wish to use Git instead to unstage changes, the two main ways to do this via `git restore` and `git checkout`.

The `git restore` removes any unstaged changes to the specified file, and restores the file to the state of the last staged changes. If there are no staged changes, it restores the file to its state in the most recent commit (pointed to by HEAD). There must be some unstaged changes for the operation to be successful.

```
$ git restore *.txt
$ git status
On branch master
nothing to commit, working tree clean
```

If you check the affected files now in your editor, you will see that the changes you introduced just now have being removed. This removal is permanent and cannot be retrieved.

We are going to repeat the changes that we made previously.

```
4: honda
5: subaru
```

cars.txt

```
4: dog
5: mouse
```

animals.txt

and check the status again with `git status` to confirm the changes.

The `git checkout` command is normally used to switch HEAD to point to a new branch. This is typically done to begin a new line of development from the main code base, as we will see in an upcoming lesson. However, `git checkout` can also be used to remove any unstaged changes as shown below.

Type:

```
$ git checkout -- *.txt
$ git status
On branch master
nothing to commit, working tree clean
```

Once again, you will see that the changes you introduced just now have being removed. Again, this removal is permanent and cannot be retrieved.

4 Changing staged changes to become unstaged

If we have some staged changes that we wish to unstage, we can use either the `git reset` or `git restore` commands.

Repeat the changes to the files as shown at the start of previous topic and save.

```
4: honda
5: subaru
```

cars.txt

```
4: dog
5: mouse
```

animals.txt

Then stage them and check their status:

```
$ git add --all
$ git status

On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   animals.txt
        modified:   cars.txt
```

The `git reset` command has 3 invocation modes which produces different results:

- `--hard`: Staged and unstaged modifications are permanently undone. The working directory is restored to the state of the most recent commit (pointed to by HEAD).

- `--mixed`: Staged modifications become unstaged, but are still retained. Working directory state remains unchanged. If no invocation mode is specified, then this is the default.

- `--soft`: No changes to either staged or unstaged modifications. Working directory state remains unchanged. This argument typically only makes sense when you are resetting to some commit earlier in the commit history than HEAD.

Now run this command:

```
$ git reset
Unstaged changes after reset:
M       animals.txt
M       cars.txt
```

Check the status to verify that the staged changes have now become unstaged again.

```
$ git status
On branch master
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:   animals.txt
        modified:   cars.txt
```

Lets stage the changes again:

```
$ git add --all
```

Another way to unstage the changes is to use the `git restore` command that we used in an earlier lab but with a new option:

```
$ git restore --staged *.txt
$ git status
On branch master
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:   animals.txt
        modified:   cars.txt
```

Once the changes are unstaged, they can be removed permanently with any of the commands we have seen in the previous section.

5 Undoing both staged and unstaged changes

If you have both staged and unstaged changes which you want to undo permanently, then you can use either the `git reset` or `git checkout` command.

Lets stage the changes again:

```
$ git add --all
```

Make the following additions to the 3rd file:

```
6: admin officer
7: PR rep
```

humans.txt

Check the status:

```
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   animals.txt
        modified:   cars.txt

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:   humans.txt
```

We now have both staged and unstaged changes. We can now use `git reset` with a new option to remove both the staged and unstaged changes.

```
$ git reset --hard
HEAD is now at 1764e2d Put in some awesome new animals !

$ git status
On branch master
nothing to commit, working tree clean
```

Notice now that both staged and unstaged changes have been removed. All 3 affected files have been restored back to their original content prior to the changes.

Now repeat the changes again:

```
4: honda
5: Subaru
```

cars.txt

```
4: dog
5: mouse
```

animals.txt

and stage them with `git add --all`. Then repeat the changes to humans.txt as well

```
6: admin officer
7: PR rep
```

humans.txt

and verify again with `git status`.

We saw earlier `git checkout` is used to switch HEAD to point back to a new branch. When we specify the branch as HEAD, this effectively removes any existing staged and unstaged changes:

Run this command and check the status again:

```
$ git checkout HEAD *.txt
Updated 3 paths from 69d955a

$ git status
On branch master
nothing to commit, working tree clean
```

Again, notice now that both staged and unstaged changes have been removed. All 3 affected files have been restored back to their original content prior to the changes.

6 Reverting to the contents of a previous commit

A very common activity in most development workflows is to revert the code base to an older version in a previous commit. This may occur when the changes introduced by the most recent series of commits are implementing a new feature that is no longer required due to a sudden change in client requirements.

Generally speaking, there are 3 options to revert the working directory state to the contents of an earlier commit. Each of them differ from each other and are used for specific situations:

a) git reset --hard
b) git revert
c) git checkout - to be covered in an upcoming lab

We will simulate a situation similar to that described above here:

Make the following additions and save:

```
4: chevrolet
```

cars.txt

Stage and commit it with:

```
$ git commit -am "A meaningless change"
```

Make the following additions and save:

```
4: snake
```

animals.txt

Stage and commit it with:

```
$ git commit -am "Yet one more meaningless change"
```

Check that these are committed properly by inspecting the commit history:

```
$ git log --oneline -n 3
commit a925e9e (HEAD -> master) Yet one more meaningless change
commit 4df8a11 A meaningless change
commit 1764e2d Put in some awesome new animals !
```

Make the further following additions and save:

```
5: donkey
```

animals.txt

and stage them with:

```
$ git add --all
```

Make the following additions and save:

```
6: driver
```

humans.txt

Check that you now have both staged and unstaged changes with:

```
$ git status
```

Let imagine for a moment that we are working with a real code base rather than text files, and that we now wish to revert the code base back to the commit with the message `Put in some awesome new animals !` and test it. We can do this using `git reset --hard` and identifying the hash of the commit and type this command:

```
$ git reset --hard commit-ref
HEAD is now at 1764e2d Put in some awesome new animals !
```

(Note that you could also have used the relative reference operator in the `git reset --hard` command, for e.g. `git reset --hard master~2`. The relative reference operator can be used anywhere a *commit-ref* is required)

Now check the commit history again:

```
$ git log --oneline -n 2
commit 1764e2d9225022cefda8f6fafddab3fb38f2ee65 (HEAD -> master)
Author: Peter Parker <spiderman@gmail.com>
Date:   Thu Jul 8 16:55:36 2021 +0800

    Put in some awesome new animals !

commit 11606b706570f1496c2234a5115c9d15a616b9eb
Author: Peter Parker <spiderman@gmail.com>
Date:   Thu Jul 8 08:57:26 2021 +0800

    Another set of changes
```

Notice that:

- both the HEAD and Master pointers are now pointing to a previous commit that was in the commit history.
- the contents of your working directory have reverted back to the state of the snapshot at this particular commit
- the staged and unstaged changes that you had prior to executing this command are completely lost

Since commit objects are only able to point backwards to their parent commits, we are no longer able to access the two new commits that we just created earlier. These are known as orphaned (or dangling) commits, and will eventually be removed by Git's internal garbage collector. This is similar to how unreferenced objects are removed via dynamic memory management in languages like Java or C#.

Because the `git reset --hard` command results in the loss of all staged and unstaged changes, as well as making all child commits of the reverted commit orphaned commits, it should be used with extreme caution. There are ways to minimize or undo the effects of a `git reset --hard` command such as using the `stash` and `reflog` command as we will see later. However, if you don't need or want to risk the extreme changes caused the `git reset --hard` command, you should consider using `git revert` or `git checkout` as we will see in subsequent labs.

7 Accessing orphaned / dangling commits

We saw that in the previous lab that git `reset --hard` may potentially result in one or more orphaned commits. This orphaned commits still existing in the Git repository (.git folder), although they are no longer accessible via commands such as `git log`. However, if we are able to obtain their hashes, we can still access them via any command that accept commit references.

One way to obtain the hashes of dangling commits is through the `git reflog` command (short for reference logs). This produces a listing of commits that a particular commit pointer (by default this will be HEAD, but it can also be a branch, tag, etc) has referenced over its entire lifetime. Since HEAD always tracks the current branch whenever a new commit is added, examining the log of all commits that HEAD has ever pointed to will help us locate any orphaned commits that we cannot access via `git log`.

```
$ git reflog

d0cf38a (HEAD -> master) HEAD@{0}: reset: moving to d0cf38
19502b7 HEAD@{1}: commit: Yet one more meaningless change
45a3420 HEAD@{2}: commit: A meaningless change
….
….
ffc240f HEAD@{11}: commit: Added the useless file
2102bee HEAD@{12}: commit (initial): Yay ! My first commit
```

The reflog listing for a given commit pointer starts from `@{0}` (which is the most recent commit that the pointer is positioned at) and proceeds all the way to `@{x}` (which is the commit that the pointer was pointing at when it was initially created).

We can also check the reflog listing for the single `master` branch:

```
$ git reflog master

d0cf38a (HEAD -> master) master@{0}: reset: moving to d0cf38
19502b7 master@{1}: commit: Yet one more meaningless change
45a3420 master@{2}: commit: A meaningless change
….
….
2102bee master@{10}: commit (initial): Yay ! My first commit
```

The listing is nearly the same with that for HEAD with the main difference being that the HEAD listing have a few additional entries (indicated with a `reset` instead of a `commit`) that correspond to the instances when we used `git reset` in an earlier lab to discard staged and unstaged changes.

Let's assume at this point we wish to undo the previous `git reset --hard` command by moving both HEAD and master to point back to the latest commit in the commit history of the current sequence of commits (i.e. the commit with message "Yet one more meaningless change"). Since we can obtain the hash of this commit from the reflog listing, we can just simply move HEAD and master back to this latest commit by just issuing another `git reset --hard` with the hash of this commit.

```
$ git reset --hard commit-ref

HEAD is now at 19502b7 Yet one more meaningless change
```

Verify that the file contents in the working directory have now changed to the state of the snapshot in that commit and check that the previously orphaned commits are now accessible again with:

```
$ git log --oneline
```

You can experiment moving both the HEAD and master around to point to different commits using `git reset --hard`, as long as you have the hash for those commits either through the conventional `git log` or through `git reflog`. For e.g. you could move it all the way back to the root commit, and then move it back again to the most recent commit.

Notice that `git reflog` will show that Git continues to log all movements of both HEAD and master as a result of executing `git reset --hard` at any point.

Keep in mind though that `git reset --hard` causes all staged and unstaged changes to be permanently lost, so even if you return back to the previous commit that you moved away from through another `git reset --hard`, you will not be able to recover any of those changes.

The reflog for the HEAD pointer can be found in `.git` folder at logs\HEAD while the reflog for any other branches (at this point there is only one: master) can be found at `logs\refs\heads\XXX`, where `XXX` is the name of the branch.

8 Saving staged and unstaged changes in the stash

We have seen that `git reset --hard` causes all staged and unstaged changes to be permanently discarded. This means we need to persist all these changes into a new commit if we want to retain them before executing the command. However, a key idea of a commit is that it should contain complete, meaningful work, and we may not be anywhere close to evolving the existing work in our staged and unstaged changes to that state. So, we need a temporary place to store these staged and unstaged changes so that we can retrieve them at a later point of time. Git provides such a temporary place through the stash.

Move HEAD and master back to the latest commit (if you have not done so already) with:

```
$ git reset --hard commit-ref
```

Create this new file and add some content into it:

```
India
```

countries.txt

Stage it with:

```
$ git add --all
```

Create another new file and add some content into it:

```
Pacific
```

oceans.txt

Make some additions to animals.txt and save:

```
5: donkey
6: horse
```

animals.txt

Verify now that you have both saved and unsaved changes, as well as an untracked file with:

```
$ git status
```

Now save both these changes in the stash with:

```
$ git stash

Saved working directory and index state WIP on master: 19502b7 Yet
one more meaningless change
```

Check again on the status with:

```
$ git status
```

Notice now that all your staged and unstaged changes are gone, as they have now been saved as the first entry into the stash. The untracked file is still present.

You can check the entries in the stash with:

```
$ git stash list

stash@{0}: WIP on master: 19502b7 Yet one more meaningless change
```

Currently there is only a single entry. You can execute `git stash` at different branches / commits to continue adding entries to the stash, with each entry being labelled with a incrementing number, for e.g. `stash@{1}, stash@{2}, etc.`

To get a brief summary of the staged and unstaged changes in the topmost entry in the stash, we can use:

```
$ git stash show
```

To get a diff between the contents of the commit referenced by HEAD (the current branch) and the topmost entry in the stash, we can use:

```
$ git stash show -p
```

If there is more than one entry in the stash, you have the option of performing a diff using the changes saved in that particular entry with:

`git stash show -p stash@{x}` where x is 0, 1, 2 …..

If you check on the commit history now with:

```
$ git log --oneline --all --graph
```

you will notice that the stash entry actually appears in the commit history with a hash of its own. In that sense, you can think of a stash entry as a special kind of commit which represents temporary changes as opposed to normal commits which represent permanent changes to the project content.

Move HEAD and master back to the root commit with:

```
$ git reset --hard commit-ref
```

Check again on the status with:

```
$ git status
```

Notice the untracked file is still present. The reverting of the working directory content to the state of this commit does not remove any untracked files.

Now return HEAD and master back to the most recent commit with:

```
$ git reset --hard commit-ref
```

As usual, use `git reflog` to obtain the hash of this commit if necessary.

We can now remove the most recently saved entry (`stash@{0}`) from the LIFO list and reapply it to the working directory.

```
$ git stash pop
```

Verify now that your staged and unstaged changes have been restored.

Check the contents of the stash again with:

```
$ git stash list
```

Notice that it is now empty as the single entry has been removed.

Now save these changes in the stash again:

```
$ git stash
```

Move HEAD and master back to the root commit with:

```
$ git reset --hard commit-ref
```

Check again on the status with:

```
$ git status
```

Although we will normally apply the latest stash entry to the same commit that the entry was saved from, we have the option of applying the latest stash entry to any arbitrary commit. However, this can result in conflict with the existing content in the working directory. In that case, Git will auto merge the content and leave messages in the files with conflicting content to indicate the occurrence of this conflict. We will examine merging and resolving merge conflicts in more detail in lab session on the `git merge` command.

Let's now apply the stash that we saved from a previous commit (the most recent commit) to the current commit that we are on (the root commit).

```
$ git stash pop

Auto-merging animals.txt
CONFLICT (content): Merge conflict in animals.txt
The stash entry is kept in case you need it again.
```

Now let's check the status again with:

```
$ git status
```

Notice now that for the case of `animals.txt` the message is slight different:

```
Unmerged paths:
  (use "git restore --staged <file>..." to unstage)
  (use "git add <file>..." to mark resolution)
        both modified:   animals.txt
```

This indicates a content conflict when Git automerged the content from the stash entry and the existing content in `animals.txt`. Specific messages are placed in this file to indicate this situation, which you can verify in the text editor. You then have the option of editing the content to reach the correct desired state.

Now if you perform:

```
$ git stash list
```

Notice that the last entry is still on the stash, because Git automatically retains it there in the event of a conflict like this to allow you a chance to reapply it again if necessary. We can also always explicitly keep the latest entry on a stash (instead of removing it) by using `git stash apply` instead of `git stash pop`

Let's return to the most recent commit with:

```
$ git reset --hard commit-ref
```

And then return again to the root commit with:

```
$ git reset --hard commit-ref
```

Notice now that the previous staged and unstaged changes introduced with `git stash pop` are now gone, as expected.

9 Undoing changes in a commit through a new commit

We saw earlier that there are 3 options to revert the working directory state to the contents of an earlier commit. Each of them differ from each other and are used for specific situations:

a) git reset --hard
b) git revert
c) git checkout - to be covered in an upcoming lab

We have already explored the `git reset --hard` command in detail in a previous lab. We discussed briefly some of the problems related to the use of `git reset --hard` command, one of which was that it resulted in orphaned commits. These commits may no longer accessible in the commit history of the current branch. Although we can still view these commits and their hashes via

`git reflog`, the changing of the commit history of the current branch has serious repercussions if the current branch is a public one in a remote repo that is used in collaborative development.

As an alternative to `git reset --hard` to correct bugs that arise in the code base, we can use `git revert` instead. This command reverts the contents of the working directory to the state of a specific commit but achieves this by creating a new commit by introducing changes that reverts the changes introduced by the child commit of the targeted commit. This has the very important advantage over `git reset --hard` in that it DOES NOT change the commit history of the current branch.

Let start off by moving to the commit with the message `Put in some awesome new animals !`

```
$ git reset --hard commit-ref
```

Make these additions and save:

```
4: boeing
5: airbus
```

cars.txt

Add these changes as a new commit with:

```
$ git commit -am "Mistake ! Added planes instead of cars"
```

Check the commit history with:

```
$ git log --oneline

0e748bd (HEAD -> master) Mistake ! Added planes instead of cars
d0cf38a Put in some awesome new animals !
ea4da51 Another set of changes
….
….
```

We will now perform a revert operation with:

```
$ git revert HEAD
```

This opens up the file COMMIT_EDITMSG in the text editor where we can enter a custom message for the revert commit or accept the default provided by Git. We can just accept the line

```
Revert "Mistake ! Added planes instead of cars"
```

and save and close the editor. We can check that the new revert commit has been created:

```
$ git log --oneline

3611d10 (HEAD -> master) Revert "Mistake ! Added planes instead of
cars"
0e748bd Mistake ! Added planes instead of cars
d0cf38a Put in some awesome new animals !
ea4da51 Another set of changes
```

….
….

Check as well in the editor that the previous incorrect additions to `cars.txt` have now been removed in the latest commit.

To summarize the advantages / disadvantages of using `git reset --hard` vs `git revert` to correct a bug in a commit:

With `git reset --hard`, we can return to a commit in the commit history of the affected branch that is completely bug-free and continue adding new commits from that point onwards. This however means the more recent commits after this bug-free commit will become orphaned commits resulting in a rewriting of the commit history. If the current branch is available on a public repo and was downloaded by other devs BEFORE the `git reset --hard` was executed by another dev, this would mean that these other devs are unknowingly working on a branch containing orphaned commits. All their future development work is likely to build on these orphaned commits which would no longer be present in the latest version of that branch: meaning a lot of additional confusion in trying to align their latest work with the new commits in the latest version of the branch. On the other hand, `git reset --hard` can be used to correct a bug in a private local branch that has not yet been made public to anyone else yet.

With `git revert`, there is no changing of commit history of the affected branch. We simply add in a new commit that undoes changes from previous commits that was responsible for the bug in order to obtain a bug-free state. We can then continue building from that new commit onwards. If this branch is available in a public repo and was downloaded by other devs BEFORE the `git revert` was executed, they can easily incorporate this new commit to correct any issues in their future development work with minimal confusion. However, because the commit history is not changed, there may be many older commits in the commit history are affected by the bug which are not useful and are just adding to the length of the commit history. Therefore, use `git revert` to undo changes on a branch that has already been made available on a public repo and downloaded by other devs: otherwise `git reset --hard` is a more efficient option.

10 Adding additional content to the latest commit

Make the following additions:

```
4: chevrolet
```

cars.txt

Stage and commit it with:

```
$ git commit -am "Added new French car"
```

Verify it is added with:

```
$ git log --oneline -n 3
```

Let's assume that we realized that we forgot to add some additional content to cars.txt for this commit after it was created. To resolve this issue, we could:

- Use the `git reset --amend` option to add additional changes to most recent commit without the need to create a new commit or exclude this existing commit. This includes the ability to change the message of the most recent commit as well

- Revert back to the 2nd most recent commit using `git reset --mixed` option which retains the differences between the two most recent commits.

Let's demonstrate the first option. Make the following additions and save.

```
5: peugeot
```

cars.txt

Stage it with:

```
$ git add cars.txt
```

Add this to the latest commit with a new message:

```
$ git commit --amend -m "Added 2 new French cars"
[master 9220da7] My new cool cars
 Date: Fri Jul 9 08:08:08 2021 +0800
 1 file changed, 2 insertions(+)
```

Check the commit history again:

```
$ git log --oneline -n 3
```

Notice that we have not added any new commits, but the commit that we are currently on (pointed to by HEAD and master) has the new message and additional content that we just specified.

Let's demonstrate the second option we discussed earlier.

Make the following additions:

```
4: dolphin
```

animals.txt

Stage and commit it with:

```
$ git commit -am "Added new sea creature"
```

Verify that the new commit is created with:

```
$ git log --oneline -n 3
```

As an alternative to using `git commit --amend`, we will use `git reset` with a new option to revert back to the 2nd most recent commit but at the same time retain the differences between this commit and the most recent commit as unstaged changes. Then we can simply add in the new additions and then perform another commit.

```
$ git reset --mixed HEAD~1
```

```
Unstaged changes after reset:
M       animals.txt
```

Notice there we are using the relative reference operator, but we could have used the hash of the 2nd most recent commit as well.

Verify that you have now moved to the 2nd most recent commit:

```
$ git log --oneline -n 3
```

Notice however that the latest changes you made in animals.txt (4: dolphin) is still present in the editor. Git has taken the changes for the "`Added new sea creature`" commit and left them as unstaged changes at this point. If you had used `git reset` with the `--hard` option instead, this would not have happened: the working directory would then be in the exact state as the `Added 2 new French cars` commit

To verify this, use git diff to perform a difference comparison between all unstaged changes in the working directory and their previous contents at the most recent commit pointed to by HEAD.

```
$ git diff

diff --git a/animals.txt b/animals.txt
index 37ef319..5a1bf0b 100644
--- a/animals.txt
+++ b/animals.txt
@@ -1,3 +1,4 @@
 1: cat
 2: jaguar
 3: elephant
+4: dolphin
```

Having the changes that result in the latest commit "`Added new sea creature`" retained as unstaged changes at this point is very useful, otherwise we would have to additionally figure out these changes ourselves (for e.g. by doing a git diff between the most recent commit and 2nd most recent commit) so that we can apply it again with our extra addition.

With these changes available already, all we need to do is just add in the extra content that we wanted to include in animals.txt

```
 5: whale
```

animals.txt

This is now added to the unstaged changes, and we can again verify with:

```
$ git diff

diff --git a/animals.txt b/animals.txt
index 37ef319..1b2dc18 100644
--- a/animals.txt
+++ b/animals.txt
@@ -1,3 +1,5 @@
 1: cat
 2: jaguar
```

```
 3: elephant
+4: dolphin
+5: whale
```

Finally, we create a new commit with:

```
$ git commit -am "Added 2 new sea creatures"
```

Verify that this new commit has been added successfully:

```
$ git log --oneline -n 3
```

There is one drawback with using either `git commit --amend` or `git reset --mixed` as we have just demonstrated for the purposes of adding additional content to the latest commit. We saw in the previous lab that once we have pushed the contents of a local branch to a public remote repo, we should no longer change the commit history of that branch locally. Otherwise, we may cause problems and confusion for other devs who have downloaded that branch to work on it locally BEFORE we started making are working off the same public remote repo. We will be examining collaborative development from a shared remote repo in an upcoming lab.

As we see, `git commit --amend` or `git reset --mixed` will actually change the commit history by changing the contents of the most recent commit. Thus, just like the case of `git reset --hard`, it should be used only for changes to a private local branch that has not yet been uploaded to a public remote repo.

If the local branch has already been made public, then the safest way to add new content is just to simply add in a new commit with the new extra content. This will not alter the existing commit history.

11 Deleting untracked files

Remember that Git does not track changes to untracked files. At some point in the development workflow, you may have created a lot of new files that you did not add to the staging area with git add. These files will be untracked. It might become problematic to delete these untracked files directly from the file manager utility of the OS since we may not be able to distinguish them from the tracked files. To solve this problem, we can use `git clean`.

You should have an untracked file oceans.txt in the working directory which you created some time ago. Verify this with:

```
$ git status
```

Let assume you have a situation where there are many untracked files scattered across multiple subdirectories in the project folder. You can remove all these untracked files at once with:

```
$ git clean --force .

Removing oceans.txt
```

Notice the file `program.obj` still remains. This is because this file is considered an ignored file under the file filtering patterns we have specified in .gitignore. The concept here is that ignored files

are a necessary part of the project, but which we do not wish to enforce version tracking on for specific reasons. Therefore, `git clean` will not remove this type of files.

If were to attempt to explicitly remove `program.obj` with:

```
$ git clean --force program.obj
```

You will notice that the file still remains.

Git does however provide an option to remove only the files specified in .gitignore. Typically this would be useful if you specify build artifacts (such as binary executables) in .gitignore and you would like to remove all of them in a go.

Before doing that, lets remove the `.gitignore` entry from the `.gitignore` file (to prevent this file from deleted as well). Open the file and change its contents to:

```
*.obj
```

.gitignore

Now type:

```
$ git clean --force -X
Removing program.obj
```

This time around, program.obj is successfully removed.