Git Essentials
Lab 3
Working with branches and tags

1 Commands covered

Getting info on branches	Creating and switching between branches

`git branch`
`git branch -vv`
`git branch --merged | --no-merged`
 	
Atlassian branch tutorial
	`git branch` *`branch-name`*
	`git branch` *`branch-name hash-id`*
	
	W3Schools git branch command

`git log --all` `git log --graph`	`git checkout` *`branch-name`* `git checkout -b` *`branch-name`* `git checkout` *`commit-ref`* Atlassian git checkout GitTower git checkout command

	`git switch -c branch-name`
	`git switch branch-name`
	
	FreeCodeCamp git switch command
	
	BlueCast git switch command

Merging branches	**Renaming and deleting branches**

`git merge branch-name`
`git merge --abort`
 	
Atlassian git merge command
 	
W3Schools git merge command
	`git branch --move old-branch-name new-branch-name`
	`git branch -d | -D branch-name`

Getting info on authors	**Creating and accessing tags**

`git shortlog`
`git blame file-specifier`
`git blame L x,y file-specifier`
`git log -S term-to-search-for`
 	
Shortlog tutorial
 	
Atlassian git blame command
 	
FreeCodeCamp git blame command
	`git tag --annotate tagname -m message`
	`git tag --annotate tagname -m message commit-ref`
	`git describe`
	
	DevConnected creating tags
	
	Atlassian git tag command
	
	Finding git tags
	
	git describe command

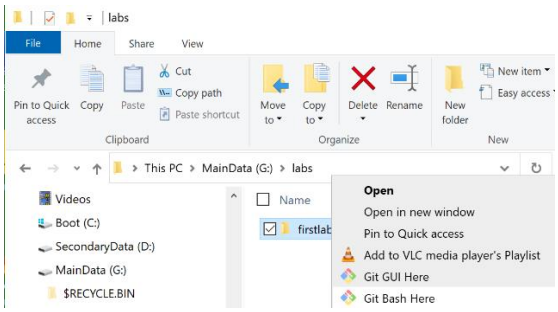
2 Lab setup

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

We will start with a new Git project which we will initialize in the working directory `labs\secondlab`. 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 a Git Bash shell.

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 Populating content in new repo

We will initialize and place several initial commits into the new repository to prepare for the upcoming lab sessions.

Create 3 files named as below and populate them with a single line each as shown using a text editor. Make sure you include a new line after the end of the single line

```
1: developer
```

humans.txt

```
1: cat
```

animals.txt

```
1: honda
```

cars.txt

Initialize a local repo in the current directory with:

```
$ git init
```

Stage these new files with:

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

Create our first commit with:

```
$ git commit -m "Initialized repo with single line in all files"
```

Check that the commit has been added properly:

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

Add in the following lines to the existing files and save. Make sure you include a new line after the end of the single line

```
2: project manager
```
humans.txt

```
2: dog
```
animals.txt

```
2: mercedes
```
cars.txt

Create the second commit with:

```
$ git commit -am "Added 2nd line to all files"
```

Check that the commit has been added properly:

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

Add in the following lines to the existing files and save. Make sure you include a new line after the end of the single line

```
3: CEO
```
humans.txt

```
3: mouse
```
animals.txt

```
3: bentley
```
cars.txt

Create the third commit with:

```
$ git commit -am "Added 3rd line to all files"
```

Check that the commit has been added properly:

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

4 Creating new branches

Branches are a very important feature of version control in Git. They support code development in parallel by ensuring that work streams of devs who are collaborating on the same code base are kept isolated and independent of each other.

Before we create a new branch, let's check that HEAD and master are both pointing to the most recent commit: `Added 3rd line to all files`

```
$ git log -n 1
```

Before we create a new branch, we will switch to a different user name and email to simulate the environment where we have different users or devs working on developing different branches at the same time.

```
$ git config --global user.name "Bruce Banner"
$ git config --global user.email "hulk@gmail.com"
```

We will now create a new branch by using the `git branch` command with the new branch name.

```
$ git branch new-animals
```

This creates a new branch that points to the same commit that the current branch (`master`) is pointing to. We can verify its creation with:

```
$ git log -n 1
```

We should now see a new branch `new-animals` pointing to the same commit as `master`.

To a get a list of local branches available in the repo:

```
$ git branch

* master
  new-animals
```

The `master` branch is highlighted in green and has an asterisk preceding it: this indicates that `master` is the current branch. This means that any new commits that are added to the latest commit will have `master` and HEAD advanced to point to it.

For additional information about the branches:

```
$ git branch -vv

* master      02e0c85 Added 3rd line to all files
  new-animals 02e0c85 Added 3rd line to all files
```

This gives us information about the commits (the commit hash and commit messages) that the various branches are pointing at. At the moment, both `master` and `new-animals` are pointing to the same commit.

Lets make `new-animals` the current branch by switching the HEAD pointer to point at it with:

```
$ git checkout new-animals

Switched to branch 'new-animals'


$ git branch

  master
```

```
* new-animals
```

The asterisk is now in front of `new-animals` indicating that it is the current branch. Notice as well that the Git Bash prompt has changed to indicate that `new-animals` is the current branch

Make the following additions and save:

```
4: tiger
```

animals.txt

Stage and commit with:

```
$ git commit -am "Added a new tiger"

[new-animals 0828937] Added a new tiger
 1 file changed, 1 insertion(+)
```

Check the commit history with:

```
$ git log -n 3

commit af3705a2b564786e3ce681a08c9175341bf9f5bc (HEAD -> new-animals)
Author: Bruce Banner <hulk@gmail.com>
Date:   Sat Mar 5 14:20:10 2022 +0800

    Added a new tiger

commit 02e0c853f552eca85297d8c189ec686c25b16a33 (master)
Author: Peter Parker <spiderman@gmail.com>
Date:   Sat Mar 5 12:37:45 2022 +0800

    Added 3rd line to all files

commit ea8b10ba42446a2a74f6493cc97310651d533717
Author: Peter Parker <spiderman@gmail.com>
Date:   Sat Mar 5 12:36:48 2022 +0800

    Added 2nd line to all files
```

Note several things:

- Adding the new commit moves the current branch `new-animals` and HEAD to point at it.
- The new commit has a new author and email, which is what we had set earlier
- `git log` shows the commit history starting from the commit pointed to by HEAD.
- We can also see the other branch (`master`) in the commit history because it is pointing to an ancestor commit of the current branch `new-animals`

Make the following additions and save:

```
5: rhino
```

animals.txt

Stage and commit with:

```
$ git commit -am "Added a new rhino"
```

Check the commit history with:

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

Once again, note that adding the new commit moves the current branch `new-animals` and HEAD to point at it.

5 Switching between branches

Now we can switch back to the `master` branch with:

```
$ git checkout master
```

and again verify that it is the active branch with:

```
$ git branch -vv
* master      884263b Revert "My new cool cars"
  new-animals b011570 Added a new rhino
```

When we perform a switch to a different branch, the HEAD pointer is moved to point to this branch and the working directory content is changed to the state of the commit that the branch is pointing at.

Verify that the content of animals.txt has reverted back to its original form before the most recent 2 commits.

With `git checkout`, only the HEAD moves while the branch that HEAD was pointing to remains in position. As we will see later, the commit that the branch is pointing to and all its subsequent ancestor commits in the commit history are still accessible.

Lets check the commit history again with:

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

Notice that the commit history doesn't show any of the 2 new commits we created earlier in the `new-animals` branch. This is because the commit history produced by `git log` always starts from the commit that the branch that HEAD is pointing to (which in this case is `master`) and iterates backwards through the sequence of ancestor commits to the root commit. Since the 2 new commits we created earlier in the `new-animals` branch are child commits (instead of ancestor commits) of master, they do not appear.

However, we can get a list of all accessible commits available in the repo with:

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

The way that `--all` option works is that Git looks for all branches in the repository, and performs a `git log` on all branches that it can find, rather than on the branch that the HEAD pointer is pointing at.

Lets switch back again to the `new-animals` branch using a different but equivalent Git command `switch`.

```
$ git switch new-animals
```

and again verify that it is the active branch with:

```
$ git branch -vv
```

One thing to note about switching branches: If you attempt to switch branches while the working directory is not clean (i.e. if there are still unstaged changes or staged changes have not yet been committed), Git will give a warning and abort the operation. This is because these changes will be permanently lost once the switch happens.

Make the following additions and save:

```
6: baboon
```

animals.txt

Now attempt to switch back to the `master` branch:

```
$ git switch master
```

```
error: Your local changes to the following files would be overwritten
by checkout:
        animals.txt
Please commit your changes or stash them before you switch branches.
Aborting
```

This means we need to persist all these changes into a new commit if we want to retain them before performing the switch. 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. At this point, we can save the unstaged changes into the stash before proceeding.

```
$ git stash
```

Then we can now switch to master with:

```
$ git switch master
```

Now let's switch back to `new-animals` again

```
$ git switch new-animals
```

Check the contents of animals.txt and verify that you have the 5 lines of text corresponding to the commit at this branch. Now we can retrieve the contents of the stash to get back our unstaged changes with:

```
$ git stash pop
```

Verify the 6th line has been added back in the editor. Also check that the unstaged changes are present with:

```
$ git status
```

We can place these unstaged changes into a commit in the usual way, but for now lets discard this unstaged changes with:

```
$ git restore animals.txt
```

And again verify that the working tree is now clean with:

```
$ git status
```

Let's switch back to the `master` branch again with:

```
$ git switch master
```

We are going to make another new branch and populate it with new commits, but before we do that, lets switch over to another user identity and email.

```
$ git config --global user.name "Clark Kent"
$ git config --global user.email "superman@gmail.com"
```

Let's make another new branch `new-cars`. There is a shortcut to create a new branch and switch to it in a single command:

```
$ git checkout -b new-cars
```

Once again, verify that this new branch is the active branch with:

```
$ git branch -vv
```

Make the following additions and save:

```
4: volkswagen
```

cars.txt

Stage and commit with:

```
$ git commit -am "Added a new volkswagen"
```

Make some more additions and save:

```
5: volvo
```

cars.txt

Stage and commit with:

```
$ git commit -am "Added a new volvo"
[new-cars f50a995] Added a new volvo
 1 file changed, 2 insertions(+)
```

Let's get a summary of the commit history for this new branch with:

```
$ git log --oneline -n 4
```

Notice again that we can see both the `new-cars` and `master` branch but not the `new-animals` branch because this branch is not accessible from the commit history commencing from HEAD.

We say that `master` is a direct ancestor of both `new-cars` and `new-animals` because all the commits in its commit history are also present in these two branches. We say that it has a non-divergent history with either `new-cars` or `new-animals`

On the other hand, we say there is a divergent history between `new-cars` and `new-animals` because neither of these branches are direct ancestors of each other (i.e. there are some commits in the commit history of `new-cars` which are not present in `new-animals` and vice versa).

To get a graphical illustration of the divergent history of these two branches, type:

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

The symbol | highlighted in red shows the two divergent branches.

If we use the longer version of the `git log` command, we should be able to see the different usernames and emails for the corresponding commits:

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

Practice switching between these 3 branches (`master`, `new-animals` and `new-cars`) using either `git switch` or `git checkout` and verify that the contents of the working directory change correspondingly as well, for e.g.

```
$ git checkout master

$ git switch new-animals

$ git switch new-cars
```

6 Working with tags

Tags are references that point to specific commits in order to mark that commit as special in some way: for e.g. representing a code base that is stable at a specific version number. While a tag refers to a commit just like a branch, it does not change. In other words, when a new commit is added from a current commit that has a tag, the tag will not move to point to the new commit unlike a branch.

There are two kinds of tags that can be created:

- A lightweight tag (which just has a name)
- An annotated tag, which includes additional metadata. Annotated tags are represented via tag objects (Git object model)

We will use annotated tags most of the time, because they provide the ability to store metadata which is the primary reason we use tags in the first instance.

Lets switch over to the `new-cars` branch with:

```
$ git switch new-cars
```

We can add an annotated tag to this commit with tag name of `v2.0` and specific message:

```
$ git tag --annotate v2.0 -m "Codebase with complete listing of cars"
```

The tag is added at the commit that HEAD points to. We can view the tag in the commit history listing with:

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

We can also add tags by specifying the hash of the commit or the branch that points to the commit that we wish to tag. For e.g. we can tag the commit that the `new-animals` branch current points to with without needing to switch over to it first:

```
$ git tag --annotate v3.0 -m "Codebase with complete listing of animals" new-animals
```

We can view both newly added tags by checking the complete commit history listing for all branches with:

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

Let's add one more tag to the commit with the message: `Added 2nd line to all files`. This is the next commit after the root commit. Note down its hash from the previous commit history listing.

```
$ git tag --annotate v1.0 -m "Midway through master" commit-ref
```

Check for the newly added tag again with:

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

To get a listing of an existing tags in the repo:

```
$ git tag
```

A tag can be used whenever a reference to a commit reference is required. This provides us another alternative to reference a commit other than a branch, commit hash or using the ~ relative reference operator

For e.g. to get details about a specific commit with the tag v3.0

```
$ git show v3.0
```

Notice that we get metadata on the tag such as the tagger name and email, as well as the tag message and date.

To get a listing of all commits within a given range, for e.g.

```
$ git log --oneline v1.0..v3.0
```

We can traverse backwards from the current commit pointed to by HEAD in order to find the first tag in the commit history using `git describe`. Lets switch back to master first with:

```
$ git switch master
```

Now if we run the command:

```
$ git describe

v1.0-1-g02e0c85
```

The information that appears is in the form of *`tagname-nocommits-hashofcurrentcommit`*.

In this case, `v1.0` is the first tag encountered traversing backwards in the commit history from the current commit, this tag is on a commit that is exactly one commit behind the current commit and the hash of the current commit is `g02e0c85`

Lets switch back to the commit with tag v3.0, which is currently pointed to by the branch `new-animals`.

```
$ git switch new-animals
```

Now if we perform:

```
$ git describe

v3.0
```

We don't get any additional information as we did with the previous `git describe` because the commit that we are currently on is already tagged with v3.0.

We can remove existing commits with `git tag --delete`.
Lets remove `v3.0` with:

```
$ git tag --delete v3.0
```

Verify that the tag is now gone with:

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

Now if we perform:

```
$ git describe
```

```
v1.0-3-g6ebda51
```

We can now see that the first tag encountered in traversing back from the current commit is `v1.0` and this tag is located 3 commits behind the current commit.

Lets switch over to `new-cars` branch, which is pointing to a commit tagged with v2.0

```
$ git switch new-cars
```

Make the following additions and save:

```
6: tesla
```

cars.txt

Stage and commit with:

```
$ git commit -am "Added a new tesla"
```

Check the commit history again with:

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

Notice that while HEAD and `new-cars` is advanced to point to this latest commit as expected, the tag v2.0 still remains fixed at the original commit it was placed at. Tags are stationary and do not move around like branches.

7 Untracked and orphaned commits

So far, we have noticed that whenever we switch branches, the HEAD pointer will be moved to point to the that branch and that branch becomes the active branch. The contents of the working directory will then be restored to the state of the commit pointed to by that branch.

However, we may sometimes wish to restore the contents of the working directory to the state of a commit that is NOT pointed to by any branch. This would typically be an arbitrary commit in the earlier commit history of any of the existing branches (in this case, `master`, `new-animals` and `new-cars`).

A possible use case for this might be when we discover a bug in the current code base of the branch we are currently working on. If the bug was not the result of the most recent changes in the latest commit, we might be interested to find the commit where the bug first made its appearance.

To accomplish this, we may need to examine the contents of suspect commits that appear earlier in the commit history. On way is of course to perform a diff between the current commit and the suspect commit, as we had done earlier. Another alternative is to simply restore the state of that commit to the contents of the working directory. In that case, we just simply need to move HEAD to point to the commit in question.

In this situation, we can use `git checkout` to move the HEAD pointer, but this time we will specify a reference to the commit that we want to move HEAD to point to (either using the commit hash, the

relative reference operator or a tag for that commit). When HEAD is moved so that it points directly to a commit rather than to a branch, it is termed to be in a DETACHED STATE.

We will move the HEAD pointer to the second last commit (currently tagged with v1.0) with:

```
$ git checkout commit-ref

Note: switching to 'commit-ref'.

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.
…..
…..
……
```

Note that since this commit is tagged, we could have used the tag in place of the *commit-ref*.

Firstly, notice again that the contents of the 3 files have reverted to the commit contents. Also notice that the hash (or the tag, it if it has one) of the commit that HEAD is pointing to is now shown in the prompt (in place of the branch name that is the active branch)

As usual, the displayed commit history will start from this commit:

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

At this point of time, you can run any suitable tests on the code base and then return to a previous valid branch. On the other hand, you may also wish to start a new line of development by creating new commits at this point of time

Lets examine in detail what happens when new commits are created when HEAD is in a detached state.

Make the following additions and save:

```
3: bear
```

animals.txt

Stage and commit with:

```
$ git commit -am "My first untracked commit"

[detached HEAD fa5106d] My first untracked commit
 1 file changed, 2 insertions(+)
```

Notice now that the prompt has changed to the hash of the newly added commit which indicates that HEAD has being moved to point to it in the usual way.

Make a further addition and save:

```
4: camel
```

animals.txt

Stage and commit with:

```
$ git commit -am "My second untracked commit"

[detached HEAD 5b3a453] My second untracked commit
 1 file changed, 1 insertion(+)
```

Notice again that the prompt has changed to the hash of the newly added commit which indicates that HEAD has being moved to point to it in the usual way.

Verify the commit history for these 2 commits:

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

Now switch back to any valid branch, for e.g. `new-animals`

```
$ git switch new-animals

Warning: you are leaving 2 commits behind, not connected to
any of your branches:

  5b3a453 My second untracked commit
  fa5106d My first untracked commit

If you want to keep them by creating a new branch, this may be a good
time to do so with:
…..
….
```

This warning is given warning is given about the latest commits not being accessible. Since there is no branch available to reference either of these 2 newly created commits, they are orphaned commits.

Copy down the hash of the commit with the message `My second untracked commit` so that you can use it later on

Now get a listing of all commits accessible with:

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

Notice that the two new commits that you added in are not listed because they are not accessible from any of the existing branches (`master`, `new-cars` or `new-animals`).

IMPORTANT: Even though these commits are not immediately accessible in the commit history listing, it does not mean that they have been deleted from the Git repository (`.git` folder). As long as you have the hash of any of these two orphaned commits you will still be able to access them directly via a `checkout` command. Git specifically makes it difficult for you to accidentally remove a commit permanently from the repository. Orphaned commits which are not referenced by anything will eventually be removed by Git's garbage collection in time - we can configure a default period for this (the default is between 30 - 90 days depending on the system).

Next, we can make a new branch that points to last in the sequence of untracked commits (the commit with the message: `My second untracked commit)`. This will allow us to access all the previous commits in that sequence. We can do this using the hash of that commit that we copied down previously:

```
$ git branch wierdbranch commit-ref-of-last-untracked-commit
```

We then switch to this branch and check its commit history:

```
$ git switch wierdbranch

$ git log --oneline
```

Lets again check the list of all commits as well as the relationships between the branches in the DAG:

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

This time we are able to see the two commits that we added because we created a new branch to keep track of them.

8 Integrating branches with a fast-forward merge

The commit history of different branches typically represent different independent lines of development from an initial common code base. At some point, we will want to integrate these different branches into a single branch again.

In general, there are two strategies for integrating two branches:

- merging (the most common approach encountered in general workflows)
- rebasing (more complex and used for specific scenarios)

There basically two types of merges possible:

- Fast forward merge (when there is no divergent history between the two branches being merged). This simply moves forward the branch that is backward to point at the same commit as the branch being merged from – no additional commit is created.

- 3-way merge (when there is a divergent history). A 3-way merge will result in an additional special commit (the merge commit) that holds the result of the merge. The merge commit will have two parent commits (unlike other normal commits that only have one). It may also cause a merge conflict that needs to be resolved manually.

When we merge branch Y into branch X, we are essentially taking the latest commits from branch Y and integrating it into branch X. This will result in some change in branch X, which depends on whether we are doing a fast forward merge or 3 way merge. Branch Y remains unchanged.

To perform a merge requires a 2-step process:

a) Identify the branch you want to merge into and switch into it to make it the current branch
b) Run `git merge` and specify the branch you want to merge from

In a typical development workflow process, we will usually merge from feature or topic branches into the `master` or `main` branch.

Let's switch over to `master`:

```
$ git checkout master
```

Let's merge from `new-animals` with:

```
$ git merge new-animals

Updating 884263b..b011570
Fast-forward
 animals.txt | 3 +++
 1 file changed, 3 insertions(+)
```

Notice that the type of merge is indicated (fast-forward). Lets check the commit history with:

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

Notice now that both `master` and HEAD pointers have been advanced to point to the same commit as `new-animals`. There have been no new commits created.

Because merging is such a common activity in a development workflow, we will often want to check which branches have been / not been merged into the current branch. We can do this with:

```
$ git branch --merged
* master
  new-animals
```

Here, we see that the current branch is `master` and the `new-animals` branch has been merged into it.

```
$ git branch --no-merged

new-cars
wierdbranch
```

Here, we see that the current branch is `master` and the new-cars and `wierdbranch` branch has not yet been merged into it.

9 Dropping a commit and reverting to a previous one

We will start off by switching over to the `new-cars` branch

```
$ git switch new-cars
```

Check the commit history:

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

```
f93133c (HEAD -> new-cars) Added a new tesla
f9d02bf (tag: v2.0) Added a new volvo
8b5fb93 Added a new Volkswagen
….
….
….
```

Lets assume that we now wish to drop the most recent commit in `new-cars` branch (`Added a new tesla`).

There are two typical use cases for dropping one (or more) commits in the most recent sequence of commits of a branch:

a) The latest sequence of commits we have created in the branch to implement some new functionality is no longer required due to sudden changes in the client requirements. In that case, we would probably want to revert the contents of our working directory directly back to an earlier commit that did not have all that unrequired functionality.

b) The latest sequence of commits (one or more commit) in the branch introduces bugs into the code base. In that case, we have the option of creating another new commit that corrects all these bugs OR revert back to an earlier commit that did not have the bug.

We cannot directly delete a commit in Git (because as mentioned earlier, Git explicitly makes it difficult for us to do this). Instead we can make this latest commit (`Added a new tesla`) an orphaned commit, and therefore inaccessible. To do this, we move BOTH the HEAD and `new-cars` branch to point back to the 2nd most recent commit (`Added a new volvo`) with:

```
$ git reset --hard v2.0
```

And verify the commit history again:

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

Notice now that the previously most recent commit in the `new-cars` branch (`Added a new tesla`) is no longer visible because this is now an orphaned commit. However, if we had saved the hash of this commit, we are still able to access it as demonstrated before in a previous lab.

The key point to note here is that we cannot directly delete a commit (i.e. remove it permanently from the Git repository - the `.git` folder). We can however indirectly delete it by making it an orphaned commit, and if we do not create any branch to reference it (just like we did in a previous lab) after a certain period of time, it will be permanently removed by Git's GC. This default period is typically 30 - 90 days and is configurable.

10 Integrating branches with a 3-way merge

Next, let's make master the current branch:

```
$ git switch master
```

Now we are ready to attempt to merge `new-cars` into `master`. Notice that master now has a divergent history from new-cars. More specifically master has two commits (Added a new rhino, new tiger) which new-cars does not have, and new-cars has two commits (Added a new volkswagen, volvo) which master does not have.

Then we attempt a merge with:

```
$ git merge new-cars

hint: Waiting for your editor to close the file...
```

This opens up your default editor to provide a message for the new merge commit that will be created. This has the default message of:

```
Merge branch 'new-cars'
```

You can accept that by closing the editor.

```
Merge made by the 'ort' strategy.
 cars.txt | 4 +++
 1 file changed, 4 insertions(+)
```

Notice that the specific merge strategy `'ort'` is specified (this replaces the older default of `recursive)`. This is the default strategy if none is explicitly specified. The other types of strategies that can be employed include:

- Ours
- Octopus
- Resolve
- Subtree

When performing a merge between two branches, the first thing that Git will do is to attempt to find a common base commit in their respective commit histories. Merge strategies refers to the different ways in which Git can employ to locate this base commit.

Notice that the contents of `cars.txt` in the working directory for this new merge commit matches that of `new-cars` while the contents of animals.txt matches that of `new-animals`.

Lets check the commit history with:

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

Notice the new commit (the merge commit) for `master`. This merge commit has 2 parent commits.

Lets check the commit history for the 2 other branches:

```
$ git log new-cars --oneline --graph -n 3

$ git log new-animals --oneline --graph -n 3
```

Notice that neither of them has changed. Remember that merging branch A into branch B does not change branch A. We have consecutively merged `new-animals` and then `new-cars` into

`master` - so only the `master` branch changes (either moving forward in the case of a fast forward merge, or pointing to a newly created merge commit in the case of a 3 way merge).

Let's verify that both these two branches have being merged into `master`

```
$ git branch --merged

* master
  new-animals
  new-cars

$ git branch --no-merged

wierdbranch
```

The only branch left that has yet to be merged into `master` is `wierdbranch`

11 Resolving a merge conflict

Lets create another new branch `new-humans` and switch to it:

```
$ git switch -c new-humans
```

Make the following additions and save:

```
4: infra staff
5: office cleaner
```

humans.txt

Stage and commit with:

```
$ git commit -am "Added some folks from new-humans"
```

Verify the commit history:

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

Lets switch back to `master`:

```
$ git switch master
```

Make the following additions and save:

```
4: accountant
5: receptionist
```

humans.txt

Stage and commit with:

```
$ git commit -am "Added some people from master"
```

Verify the commit history:

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

Now let's attempt to merge `new-humans` into `master` and then check the status:

```
$ git merge new-humans

Auto-merging humans.txt
CONFLICT (content): Merge conflict in humans.txt
Automatic merge failed; fix conflicts and then commit the result.

User@computer MINGW64 /g/labs/firstlab (master|MERGING)

….
```

We can check the status now as well with:

```
$ git status

….
…
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

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

Some key things to note:

- A merge conflict resulting in a failure to automatically merge is flagged. This indicates that we will need to complete the merge ourselves by manually resolving the merge conflict
- The prompt changes to `(master|MERGING)` which indicates that we are currently halfway through an incomplete merge attempt
- The status indicates `unmerged paths` and identifies the files involved (`humans.txt`) where both the branches that are to be integrated have conflicting content.

Open humans.txt. You should see a section of the file where Git has automatically flagged the conflict:

```
<<<<<<< HEAD
4: accountant
5: receptionist
=======
4: infra staff
5: office cleaner
>>>>>>> new-humans
```

The `<<<<<<<` indicates that the content below is from the current branch that is being merged into.
The `>>>>>>>` indicates that the content below is from the branch that is being merged from

The ======= is a divider between the conflicting content

At this point, we can choose to abort the merge attempt or proceed to manually edit the affected file.

Let's demonstrate an abort first:

```
$ git merge --abort
```

Now the contents of `humans.txt` reverts back to what it was prior to the merge attempt. Let's repeat the merge attempt:

```
$ git merge new-humans
```

Open `humans.txt` and change the conflicting portion and save:

```
4: accountant
5: office cleaner
```

humans.txt

Note that how you manually resolve the conflict is entirely up to you (or more specifically to the developers responsible for the branches being merged). You could elect the keep the portion from one branch and drop the other portion, or you could choose to combine some of the changes from both branches (as we have done here).

Next commit the changes for this merge commit in the usual way, with an appropriate commit message:

```
$ git commit -am "Merged the people and the folks successfully !"
```

Notice now that the prompt no longer has the additional term MERGING at the end of it, indicating that the merge operation is now complete.

Check the commit history:

```
$ git log --oneline -n 4 --graph
```

Once again, we see that the latest merge commit has two parent commits from the two branches being merged.

Once a merge has been completed, we can also undo it if we wish using `git reset --hard` as we have seen in the previous lab:

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

If you check the contents of humans.txt, we will be able to see that it has now reverted back to its content in the last commit in the `master` branch prior to the merge operation.

12 Renaming and deleting branches

We can rename branches in a reasonably straight forward way. Lets rename the branch that we just created in the previous section:

```
$ git branch --move wierdbranch normalbranch
```

We check the names of all the branches to verify that the rename was successful.

```
$ git branch -vv
```

We can also delete a branch if we wish to. Remember that a branch is only a pointer to a commit, so deleting a branch simply removes a reference to that commit -> it does not remove the commit or its associated snapshot or metadata.

Let's switch back to master and repeat the merge operation that we undid earlier:

```
$ git switch master

$ git merge new-humans
```

Open humans.txt and change the conflicting portion and save:

```
4: accountant
5: office cleaner
```

humans.txt

Next commit the changes for this merge commit:

```
$ git commit -am "Merged the people and the folks successfully !"
```

Now we can delete this branch with:

```
$ git branch -d new-humans
```

The general development workflow will usually involve deleting a particular branch once its contents have been fully integrated into the master/main branch representing the main code base. This is to prevent the proliferation of branches in a complex development workflow.

Note that if we attempt to delete a branch whose contents have not being fully merged into another branch, we will obtain a warning. For e.g.

```
$ git branch -d normalbranch
```

This is to ensure that the commits involved in the commit history of that branch do not become orphaned commits once that branch is deleted.
We can still proceed with the delete operation nonetheless with a different option:

```
$ git branch -D normalbranch
```

As we have already seen previously, without a branch pointer to either one of the two untracked commits we created earlier, they are no longer accessible.

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

We can of course again use `git reflog` to gain access to the hashes of these commits if we wish.

Finally, lets check the list of branches that are available again:

```
$ git branch -vv
```

We can see that `new-humans` and `normalbranch` are no longer there.

13 Getting info on authors and their changes

So far we have seen how we can use `git log` with a variety of options to obtain the commit history of various branches in the repository. The `git shortlog` command is a special version of git log intended for creating release announcements. It groups each commit by author and displays the first line of each commit message. This is an easy way to figure out which developer has worked on which commit - which may not be so clear in a normal commit history.

Let's try this out:

```
$ git shortlog
```

Here we can see the list of commit messages grouped by author. If you recall, we had switched between 3 different usernames using the `git config` command in earlier labs while working on this particular repository.

The `git blame` command is primarily used to get information related to author metadata attached to specific lines in a file. This is used to examine specific points of a file's history and get context as to who was the last author that modified the line. This is useful in exploring the history of evolution of specific code segments in a file.

Let's try this out on the 3 files in the project folder:

```
$ git blame humans.txt

$ git blame animals.txt

$ git blame cars.txt
```

Here we can see information about the author, date and commit in which each line in that file was changed in.
If the file is really large, this might be overwhelming to see all the lines in the file. In that case, we can choose to restrict to a certain number of lines for e.g.

```
$ git blame -L 2,4 humans.txt
```

Sometimes you may wish to figure out which author(s) introduced a particular line into a file. If that line is introduced into multiple files, trying to locate all of them using git blame might be problematic if there are a large number of files. Instead, we can use `git log` with a new option:

```
$ git log -S "rhino"
```

```
$ git log -S "office cleaner"
```