Git Lab 7
Working with rebase

1 Commands covered

Rebasing
`git rebase --interactive` *`commithash`* Atlassian rebase tutorial Atlassian merging vs rebasing

2 Lab setup

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

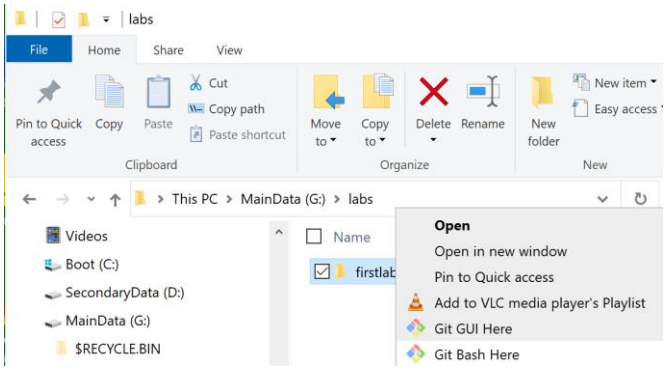
We will be using the following folders from the lab material that you have been provided for the workshop.

```
rebase-interactive-lab-original
rebase-interactive-lab-vxxxx
rebase-divergent-lab-original
rebase-divergent-lab-vxxx
rebase-divergent-conflict-lab-original
rebase-divergent-conflict-lab-vxxx
```

These folders are exact replicas of the original version; they each contain a working directory with a Git repo and some sample content and existing commit history. We will be using each of them in the next labs so we can see the results of using the different possible variations of the rebase command.

Copy them into the top-level `labs` directory that we have been using for our labs so far.
NOTE: Please copy them as just instructed, instead of using the Git Bash shell in these folders directly, so that you make another copy when you wish to repeat the exercises.

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 Basics of Rebase

The fundamental idea of rebase is to reapply or replay a sequence of commits from the current branch onto a different branch (the branch that is being rebased onto). The replayed commits will be different from the original commits on the current branch, depending on the action that is applied to them during the replay process.

There are essentially two types of rebase that can occur when branch B is being rebased onto branch A.

a) When there is no divergent history between both branches and branch B is ahead of branch A (branch A is a direct ancestor commit of branch B's commit history).

b) When there is a divergent history between both branches.

If you recall, we can integrate these branches together using a fast forward merge (for non-divergent branches) and 3 way merge for divergent branches. Rebase is an alternative approach to integrate content from one branch (source branch) into another (destination branch) in a way that results in a cleaner or more optimal commit history.

4 Interactive rebase (squash)

Interactive rebasing is primarily used to clean up or produce a neater version of the commit history of a feature branch before merging it into the main / master branch, or before pushing it to its remote counterpart. This usually involves squashing insignificant commits, deleting obsolete ones, editing commit messages and content, etc. As a result, the overall project commit history (particularly the master branch) is clearer and easier to inspect.

Here we perform the most common operation associated with rebase, which is squashing commits.

Open a Git Bash shell in `rebase-interactive-lab-v1`
This is a simple working directory with a Git repo with some existing content and commit history.
Open the files humans.txt and animals.txt in this folder with Notepad++

Let's switch over to the `new-animals` branch (if it is not already the default branch with):

```
$ git checkout new-animals
```

Check the current commit history of all branches with:

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

You should see that there are two branches: `master` and a feature branch `new-animals`. The feature branch is exactly 4 commits ahead of the master branch.

We can check for the changes between each successive commit for these 4 most recent commits which are specific to the feature branch `new-animals`

```
$ git log --patch
```

We can see that each commit successively adds a new line to the end of animals.txt. The commits in the master branch are related to the humans.txt file.

Assume at this point of time that development in this feature branch is complete, and we are ready to integrate its contents back into the master branch using a merge operation, which would result in a fast forward merge.

Before performing this integration, we may wish to first change the sequence of commits on the `new-animals` feature branch. For this case, we want to squash (combine) all the feature-specific commits of this branch into one single commit.

The general format for an interactive rebase is:

```
git rebase --interactive commithash
```

This specifies that the sequence of commits to be rebased will commence from the child commit of *commithash* all the way up to the current branch

Let's go ahead and do that now:

```
$ git rebase --interactive master
```

This specifies the sequences of commits to be rebased as starting from the child commit of `master` (`First entry in animals.txt`) all the way up to the current branch (`new-animals`).

The interactive mode opens the `git-rebase-todo` file in the Git repository (`.git`) in the default text editor for editing. The file provides a listing of all commits that are about to be replayed at the top and the interactive commands that will be applied to them. By default, this will be `pick` - which means to leave the commit exactly as it is without any change. We can then choose the specific commands (i.e. `reword, edit, squash, fixup, etc`) to be applied to each of these commits respectively and then save and close the editor.

This opens the default editor to allow you to specify actions to perform on the 4 selected commits. Make the following changes to `git-rebase-todo` and save and close the editor.

```
p 93d778a First entry in animals.txt in branch
s d867636 Second animal added
s 5448931 Third animal added
s 3a367a3 Fourth animal added
```

This basically keeps the first commit (`First entry`) and combines (squashes) the rest of the other 3 successive commits into it, resulting in one final commit.

The default editor opens up to prompt you to specify a new commit message for the final single commit. By default, it will show you all the commit messages for all 4 commits that are squashed into this final commit. If you save and close the editor now, the commit message for the final single commit will contain all commits messages from the original 4 commits.

You can do that if you wish, but here we will instead just use a new commit message as shown below. Make sure you delete the rest of the other commit messages as well.

```
# This is a combination of 4 commits.
# This is the 1st commit message:

Combined all 4 commits here
```

Save and close the editor. You should see some messages in the shell prompt indicating that the rebase operation has completed successfully:

```
Date: Fri Jun 10 09:23:26 2022 +0800
 1 file changed, 4 insertions(+)
```

```
 create mode 100644 animals.txt
Successfully rebased and updated refs/heads/new-animals.
```

The shell prompt indicates that the rebase has completed immediately after this. If you now check the commit history again with:

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

You will now see all 4 commits squashed into a single commit with the new message that you specified earlier, with `new-animals` at this single commit. If you check animals.txt, it still contains all the 4 lines from the last commit.

At this point, we can perform the merge of `new-animals` into `master` in the way that we have done in a previous lab:

```
$ git checkout master
```

```
$ git merge new-animals
```

At this point, if we check the commit history again with:

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

We can see that master is now aligned with `new-animals` at the latest commit, and more importantly, the commit history of master is now concise: containing only the final commit from the feature branch that has the source code for the fully implemented feature.

At this point of time, since the content of `new-animals` (the single commit) has been merged successfully into master, we can proceed to delete it if we wish:

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

And we can check the commit history again with:

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

The squashing of all the feature-specific commits in a feature branch before merging it into the master branch is such a common operation that there is short cut to achieve the operations above.

Open a new Git Bash shell in `rebase-interactive-lab-v2`
This has exactly the content and commit history of the previous repo `rebase-interactive-lab-v1`

Check the current commit history of all branches with:

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

You should see that there are two branches: `master` and a feature branch `new-animals`. The feature branch is exactly 4 commits ahead of the master branch.

Now without performing an explicit rebase as we did previously, we can perform a merge immediately with the `--squash` option to accomplish the same result:

```
$ git checkout master

$ git merge --squash new-animals
```

This operation basically squashes all the feature specific commits from `new-animals` into an staged change in the affected file animals.txt. It does not yet create a new commit for these staged changes: you will have to do this explicitly yourself.

Verify this with a:

```
$ git status
```

And check on the staged changes:

```
$ git diff --staged
```

Let's go ahead and commit these staged changes in the usual manner:

```
$ git commit -m "Combined all 4 commits here"
```

Now if we check the current commit history of all branches with:

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

We can see that master is now pointing to the latest commit that contains all the squashed commit content from `new-animals`.
However, in this approach, all the feature specific commits of `new-animals` are still present. In other words, we can get the goal we are trying to achieve (a clean commit history for the master branch) but at the same time, not removing any of the commits in the original `new-animals` feature branch (which would have happened if we had done an explicit `git rebase --interactive`). This way, we can still return to inspect the contents of these commits if need be in the future.

At this point, we can safely remove the new-animals branch, just as we did in the previous example:

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

And we can check the commit history again with:

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

Now the commit history looks exactly the same as the first example where we applied an explicit rebase.

Squashing all the commits in the feature branch into a single final commit before merging into master is a very common operation. However, we also have the option to squash some of the smaller commits (rather than all of them) into multiple larger commits on the feature branch.

Open a new Git Bash shell in `rebase-interactive-lab-v3`

This has exactly the content and commit history of the previous repo `rebase-interactive-lab-v1`

Lets again repeat the rebase operation with all feature-specific commits with:

```
$ git checkout new-animals
```

```
$ git rebase --interactive master
```

Make the following changes to `git-rebase-todo` and save and close the editor.

```
p 93d778a First entry in animals.txt in branch
s d867636 Second animal added
p 5448931 Third animal added
s 3a367a3 Fourth animal added
```

This basically squashes the 2nd commit into the 1st and squashes the 4th commit into the 3rd, leaving 2 new commits in the rebased branch.

Next, the default editor next opens up to prompt you to specify a new message for first new commit (combining commit #1 and #2). As in the previous case, this by default shows the existing commit messages for these 2 commits. We can accept both of these messages by closing the editor if we wish, but here we will specify a new message, making sure to delete any other commit messages:

```
# This is a combination of 2 commits.
# This is the 1st commit message:

Combination of original commit #1 and #2
```

Next, the default editor next opens up to prompt you to specify a new message for second new commit (combining commit #3 and #4). As in the previous case, this by default shows the existing commit messages for these 2 commits. We can accept both of these messages by closing the editor if we wish, but here we will specify a new message, making sure to delete any other commit messages:

```
# This is a combination of 2 commits.
# This is the 1st commit message:

Combination of original commit #3 and #4
```

Finally after saving and closing, the shell prompt again indicates that the rebase has completed. If you now check the commit history again with:

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

You will now see the 2 new commits with the new messages specified earlier. We can then perform a consecutive diff to verify the changes to each of these 2 new commits:

```
$ git log --patch
```

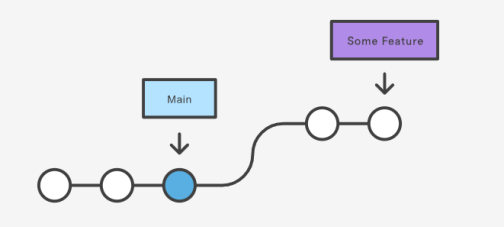
You will see the most recent commit adds in the last 2 lines (3 and 4), which is the result of squashing the contents of the 3rd and 4th original commit together, while the 2nd most recent commit adds in the first 2 lines (1 and 2), which is the result of squashing the contents of the 1st and 2nd original commit together

At this point, you can choose to merge the rebased `new-animals` into master again as we have already demonstrated previously in the 2 other examples.

To conclude: we can squash all the commits specific to a feature branch into a one single final commit, or we can summarize the feature-specific commit history by squashing small commits into several larger commits. For e.g. if you have a feature branch with 20 commits in it, you could opt to squash each 5 consecutive commits into 1 single commit, resulting in a rebased branch with 4 commits. This is a compromise between squashing everything into commit or leaving the feature branch with all 20 commits.

5 Interactive rebase (reword, edit)

Other than squashing commits, interactive rebasing also allows you to perform other operations to clean up or produce a neater version of the commit history of a feature branch. This include deleting obsolete or unwanted commits, editing commit messages, editing commit content and so on. Again, the motivation is to create an overall commit history for the master branch that is cleaner and easier to inspect.



In the illustration above, we have a master and feature branch as this is the most typical scenario for performing an interactive rebase (as we demonstrated earlier), but we can perform an interactive rebase on a single branch by simply specifying an ancestor commit of that branch in the interactive rebase command. The ancestor commit would then simply be the equivalent of the main / master branch in the diagram above.

Open a Git Bash shell in `rebase-interactive-lab-v4`
This has exactly the content and commit history of the previous repo `rebase-interactive-lab-v1`
Open the file animals.txt in this folder with Notepad++

Check the current commit history of all branches with:

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

You should see that there are two branches: `master` and a feature branch `new-animals`. The feature branch is exactly 4 commits ahead of the master branch.

Interactive rebasing is used to edit, delete or squash any of the commits in the commit history of a branch by replaying the sequence of commits and allowing us to specify the actions to be performed on each replayed commit. We saw previously that the general format is:

`git rebase --interactive` *`commithash`*

This specifies that the sequence of commits to be changed commences from the child commit of *`commithash`*. For e.g., `git rebase --interactive` *`branchA`*`~3` will replay the 3 most recent commits, up to and including the commit pointed to by *`branchA`*

Let's go ahead and do that now:

```
$ git rebase --interactive new-animals~3
```

Notice now that we are NOT specifying all the feature specific commits in `new-animals` for the rebase operation, only the most recent 3 (out of the available 4 feature specific commits).

Make the following changes to `git-rebase-todo` and save and close the editor.

```
p d867636 Second animal added
r 5448931 Third animal added
e 3a367a3 Fourth animal added
```

Git will then commence to apply these interactive commands sequentially to these commits in sequence, opening up the default text editor for editing when appropriate (for e.g. when commit messages need to be reworded) or returning to the prompt when the operation pauses at a `edit` command.

Here, we have chosen to accept the `Second animal` commit as it is, reword the commit message for the `Third animal` commit and edit the contents of the `Fourth animal` commit.

After the editor is closed, it will reopen again at the `Third animal` commit to allow us to edit the commit message. Enter this new commit message:

```
Third cool animal added
```

save and close your editor.

If you return to the Git Bash shell, you will see some messages regarding the operations performed so far and where the rebase operation is currently stopped at:

```
[detached HEAD 0d8e3e2] Third cool animal added
 Date: Fri Jun 10 09:25:41 2022 +0800
 1 file changed, 1 insertion(+)
Stopped at 3a367a3...  Fourth animal added
```

The shell prompt will also indicate that you are in middle of a rebase operation:

```
labs/rebase-interactive-lab-v4 (new-animals|REBASE 3/3)
```

If the prompt is accessed in the middle of a Git rebase operation, it will indicate the status of the rebase with:

```
(currentbranch|REBASE x/y)
```

where y represents the total number of commits to be replayed and x represents the current commit in y where the rebase operation has paused at.

If you check on the commit history of all branches with:

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

* a8c65eb (HEAD) Fourth animal added
* 0d8e3e2 Third cool animal added
| * 3a367a3 (new-animals) Fourth animal added
| * 5448931 Third animal added
|/
* d867636 Second animal added
```

You will see two divergent branches starting from the Second animal commit.
The first divergent branch is the original `new-animals` branch with its original commit history.
The second divergent branch represents the replaying of the sequence of commits from the original `new-animals` branch with the various operations that we specified earlier (pick, reword, edit). The HEAD pointer will be in detached state and will be pointing to each new replayed commit in this second divergent branch until the point when the rebase operation is complete. When that happens, `new-animals` will be moved to point to the same commit as HEAD with a commit history of new commits.

At this point of time, the rebase operation is paused as the 3rd commit to be replayed (in this case, `Fourth animal added` commit), and waiting for us to make changes to the working directory content and save that as the new commit.

Make the following amendment to the last line and save:

```
4: donkey and horse
```

animals.txt

Now check on the status with:

```
$ git status

interactive rebase in progress; onto 93d778a
Last commands done (3 commands done):
   reword 5448931 Third animal added
   edit 3a367a3 Fourth animal added
  (see more in file .git/rebase-merge/done)
No commands remaining.
```

The first line indicates that this rebase operation is currently being performed on a sequence of commits starting from the child commit of the commit with the hash of `93d778a`.

We can see the last 2 commands being performed (reword, edit) as well as the number of commands remaining (in this case, none). Although we did specify 3 command originally (pick, reword, edit) - the p command basically means to accept the original commit as it is, so we only really need to perform two commands (reword, edit)

Lets now stage and commit the new changes we made to the previous `Fourth animal added` commit and also provide a new commit message with:

```
$ git add *.txt

$ git commit --amend -m "2 new riding animals"
```

Now we can check on the status again with:

```
$ git status

….
….
nothing to commit, working tree clean
```

Finally, we continue the rebase operation with

```
$ git rebase --continue

Successfully rebased and updated refs/heads/new-animals.
```

At this point, the shell prompt should move out from the REBASE state to just indicate the current / active branch as we have finally complete the rebase operation as all specified commands have been executed on the selected commits.

If we now check the commit history again with:

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

We can now see that there are again 2 branches (`master` and `new-animals`), but now `new-animals` has the new replayed commits in its commit history sequence.

To summarize:

a) Interactive rebasing recreates / rewrites the existing commit history of a branch by applying specific actions (delete, squash, edit, etc) to the specified commits (not all) of the original history. The primary purpose of doing this is to clean up or produce a neater version of the commit history of a feature branch before merging it into the main / master branch

b) The HEAD pointer tracks the creation of the new replayed commits while the rebase operation is in progress.

c) For a pick (p) operation, we will accept the commit as it is without performing any operation on it. This is the default action for all specified commits.

d) For a reword (r) operation, the default editor will open to allow us to specify a new commit message.

e) For an edit (e) operation, the rebase operation will pause to allow us to make changes to the state of the specified commit, and then add these modified changes with `git commit --amend`.

f) The `git status` and the shell prompt will indicate the progress through the rebase operation.

g) When the rebase operation completes, the original branch will have a new commit history with the replayed commits.

6 Merge conflicts in interactive rebase (drop, edit)

When certain operations are performed on specific commits in the group of commits that are involved in the rebase operation, there is a possibility of a merge conflict. This is due to the way that Git performs the replay of the commits involved in the rebase operation.

Open a Git Bash shell in `rebase-interactive-lab-v5`
This has exactly the content and commit history of the previous repo `rebase-interactive-lab-v1`
Open the file animals.txt in this folder with Notepad++

Check the current commit history of all branches with:

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

You should see that there are two branches: `master` and a feature branch `new-animals`. The feature branch is exactly 4 commits ahead of the master branch.

We will go ahead and perform a rebase on all feature-specific commits with:

```
$ git rebase --interactive master
```

Make the following changes to `git-rebase-todo` and save and close the editor.

```
p 93d778a First entry in animals.txt in branch
d d867636 Second animal added
p 5448931 Third animal added
p 3a367a3 Fourth animal added
```

We designate the 2nd commit for deletion.

Upon return to the Git bash shell prompt, we get messages regarding a merge conflict:

```
Could not apply 5448931... Third animal added
Auto-merging animals.txt
CONFLICT (content): Merge conflict in animals.txt
```

If we check the status with:

```
$ git status
```

You will see information about the merge conflict and the place it occurs in (animals.txt). The conflict occurs as a result of trying to reconcile the dropping of the 2nd commit and the replaying of the 3rd commit which contains content from the 2nd commit (`2: dog`)
Opening up animals.txt shows this conflict clearly:

```
1: cat
<<<<<<< HEAD
=======
2: dog
3: mouse
>>>>>>> 5448931... Third animal added
```

Change the content of the file to the following (indicating that you want to drop the content associated with the 2nd commit: 2: dog) and close and save.

```
1: cat
3: mouse
```

We will stage and commit this new commit with:

```
$ git commit -am "Continuing without dog"
```

Next, we continue the rebase with:

```
$ git rebase --continue
```

The rebase operation completes without any further interruptions.

If we now recheck the commit history of the repo with:

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

You should see now see there are only 2 feature specific commits left as we dropped the 2nd commit, with a new commit message for the 3rd commit where the conflict occurred at. The final commit adds an additional line (4: dogs) but with the 2nd line corresponding to the 2nd commit missing: as we would intuitively expect. You can check the contents of animals.txt to confirm this.

Also, we can check the changes introduced in the new sequence of commits at each stage with:

```
$ git log --patch
```

To verify that the changes are in accordance to the actions that we have just performed.

In this example, there was no further merge conflicts after the resolving of the conflict at the 3rd replayed commit. However, depending on how the conflict was resolved, there could be another further conflict with the content at the next commit to be replayed and this needs to be resolved in the same way until the entire rebase operation completes.

We will now repeat this process again but this time by making an edit to the contents of a commit in the middle of the sequence of commits to be rebased.

Open a Git Bash shell in `rebase-interactive-lab-v6`
This has exactly the content and commit history of the previous repo `rebase-interactive-lab-v1`
Open the file animals.txt in this folder with Notepad++

Check the current commit history of all branches with:

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

You should see that there are two branches: `master` and a feature branch `new-animals`. The feature branch is exactly 4 commits ahead of the master branch.

We will go ahead and perform a rebase on all feature-specific commits with:

```
$ git rebase --interactive master
```

Make the following changes to `git-rebase-todo` and save and close the editor.

```
p 93d778a First entry in animals.txt in branch
e d867636 Second animal added
p 5448931 Third animal added
p 3a367a3 Fourth animal added
```

We designate the 2nd commit for editing.

If you return to the Git Bash shell, you will see some messages regarding the operations performed so far and where the rebase operation is currently stopped at:

```
Stopped at d867636...  Second animal added
```

The shell prompt will also indicate that you are in middle of a rebase operation:

```
labs/rebase-interactive-lab-v6 (new-animals|REBASE 2/4)
```

Make the following changes to the 2nd commit by adding the following lines to animals.txt

```
2.1: labrador
2.2: retriever
2.3: bulldog
```

Then we can go ahead and stage and commit it with a new message:

```
$ git add *.txt

$ git commit --amend -m "List of 3 nice dogs"
```

Then we can go ahead and continue the rebase operation with:

```
$ git rebase --continue
```

We are now provided with messages regarding a merge conflict at the prompt:

```
Could not apply 5448931... Third animal added
Auto-merging animals.txt
CONFLICT (content): Merge conflict in animals.txt
```

When we study animals.txt, we can see the source of the conflict is again the changes from the previous modified commit (pointed to now by HEAD) and the third commit.

```
1: cat
<<<<<<< HEAD
2: dogs
2.1: labrador
2.2: retriever
2.3: bulldog
=======
2: dog
3: mouse
>>>>>>> 5448931... Third animal added
```

Change the content of animals.txt to the following (where we simply add on the content from the 3rd commit to the previous modified changes in the 2nd commit):

```
1: cat
2: dogs
2.1: labrador
2.2: retriever
2.3: bulldog
3: mouse
```

We will stage and commit this new commit with:

```
$ git commit -am "Third animal added to the list of dogs"
```

Next, we continue the rebase with:

```
$ git rebase --continue
```

The rebase operation completes without any further interruptions.

If we now recheck the commit history of the repo with:

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

We see the modified commit messages for the 2nd and 3rd commit, and the final and most recent commit contains the last line (`4: donkey`) as well as the additional content we introduced in the 2nd commit.

Also, we can check the changes introduced in the new sequence of commits at each stage with:

```
$ git log --patch
```

To verify that the changes are in accordance to the actions that we have just performed.

In this example, there was no further merge conflicts after the resolving of the conflict at the 3rd replayed commit. However, depending on how the conflict was resolved, there could be another further conflict with the content at the next commit to be replayed and this needs to be resolved in the same way until the entire rebase operation completes.

7 Rebasing divergent branches

For non-divergent branches, we can perform the rebase operation on the branch first in an interactive manner before then merging it into another branch. For divergent branches, the rebase is done as part of integrating the two divergent branches involved and provides an alternative to a 3-way merge.

Before we look at rebasing divergent branches, let's revise the process of a 3 way merge between two divergent branches using an example from a previous lab:

Open a Git Bash shell in `rebase-divergent-lab-v1`
This is a simple working directory with a Git repo with some existing content and commit history.
Open the files cars.txt and animals.txt in this folder with Notepad++

Lets check the commit history for all branches with:

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

We can see two divergent branches: `master` and `new-cars`

Let's switch over to the `master` branch (if it is not already the default branch) with:

```
$ git switch master
```

Check the contents of cars.txt and animals.txt at this commit with Notepad++

Next, switch to the `new-cars` branch

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

Check the contents of cars.txt and animals.txt at this commit with Notepad++

You can see that `new-cars` is ahead of `master` with some additional content in cars.txt while master is similarly ahead of `new-cars` with some content in animals.txt. These makes both these branches divergent branches.

We will now merge `new-cars` into `master`

First, lets make `master` the current branch:

```
$ git switch master
```

Then we merge `new-cars` into `master`.

```
$ 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' into master
```

You can accept that by closing the editor.

With the completion of this merge operation, let's check the commit history with:

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

Notice the new commit (the merge commit) for `master`. This merge commit has 2 parent commits. The contents of `cars.txt` in the working directory for this new merge commit matches that of `new-cars`.

We can use rebase as an alternative to the standard 3 way merge that we have just performed. Rebasing in this situation involves replaying a sequence of existing commits from a given branch (usually the current branch) onto the tip of another divergent branch (the new base). These sequence of commits will typically commence from the common ancestor commit for both divergent branches but can start from any commit in the commit history of the given branch. Completion of the rebase results in a new replayed sequence of commits commencing from the new base.

Open a Git Bash shell in `rebase-divergent-lab-v2`
This is a simple working directory with a Git repo that is identical to `rebase-divergent-lab-v1`
Open the files cars.txt and animals.txt in this folder with Notepad++

Let's check the commit history for all branches with:

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

We will now rebase `new-cars` onto `master`

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

```
$ git rebase master
```

If you now recheck the commit history with:

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

You can see that all the feature-specific commits of `new-cars` have been added on (replayed) after the master branch.

If you now check the contents of the cars.txt and animals.txt in the current branch (`new-cars`) , you will see it contains all the latest content from master as well (in animals.txt)

Now if we check the contents of the working directory at the first replayed commit (`Added a new volkswagen`) with:

```
$ git checkout new-cars~1
```

You will see it has the complete content of animals.txt from the master branch. In other words, when this commit from the original `new-cars` branch was replayed to the end of master, all the latest content from master was combined / merged into it.

Lets return back to the master branch with:

```
$ git checkout master
```

Notice now that `master` and `new-cars` are non-divergent branches. This means we can now integrate the content of `new-cars` into `master` with a simple fast forward merge.

```
$ git merge new-cars
```

Now if we check the commit history again with:

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

You can see that both branches are now pointing to the latest commit.

If we compare this final state with the final state of the previous lab (where we immediately did a straight forward merge), we can see that there are 2 key differences / advantages:

a) There is no extra merge commit in the approach using rebase. This simplifies the commit history timeline, particularly in a large application with many feature branches, there will be a merge commit created every time we merge a complete branch into the main / master branch.

b) The project history now only consists of the commit history of the main / master branch: there are no additional feature branch commit histories to consider.

At this point, we have performed a rebase without any modifications to the commits that were rebased. However, we can also perform an interactive rebase here to specify additional operations to perform on the replayed commits, just as we did in earlier labs.

Open a Git Bash shell in `rebase-divergent-lab-v3`
This is a simple working directory with a Git repo that is identical to `rebase-divergent-lab-v1`

Open the files cars.txt and animals.txt in this folder with Notepad++

Let's check the commit history for all branches with:

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

We will now rebase `new-cars` interactively onto `master`

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

```
$ git rebase --interactive master
```

Make the following changes to `git-rebase-todo` and save and close the editor.

```
p be22439 Added a new volkswagen
s ffcc540 Added a new volvo
```

This simply squashes the 2nd commit into the 1st commit.

The default editor again opens up to prompt us for a commit message for the single final commit. We will use a new message:

```
# This is a combination of 2 commits.
# This is the 1st commit message:

Added two new cars
```

Now if we check commit history for all branches with:

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

Notice that the two original feature-specific commits from `new-cars` are now squashed into a single final commit, this is then rebased onto the end of `master` as expected.

In this simple demo, there is no merge conflict to be resolved, but as we have already seen in previous labs, merge conflicts can occur depending on the specific operations to be performed on the commits to be rebased and they will then need to be resolved in the same way that we have already described.

8 Merge conflicts in rebasing divergent branches

In the previous lab, the merging of the commit content from the rebased branch and the branch being rebased into did not result in a conflict. If it did, we would need to resolve the conflict manually as we have seen in previous cases.

Open a Git Bash shell in `rebase-divergent-conflict-lab-v1`
This is a simple working directory with a Git repo with some existing content and commit history.

Open the files cars.txt and animals.txt in this folder with Notepad++

Lets check the commit history for all branches with:

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

We can see two divergent branches: `master` and `new-stuff`

Let's switch over to the `master` branch (if it is not already the default branch) with:

```
$ git switch master
```

Check the contents of cars.txt and animals.txt at this commit with Notepad++

Next, switch to the `new-stuff` branch

```
$ git switch new-stuff
```

Check the contents of cars.txt and animals.txt at this commit with Notepad++

You can see that both branches have differing content in both cars.txt and animals.txt at the same lines that were introduced in their earlier commits. These makes both these branches divergent branches.

We will now rebase `new-stuff` onto `master`

```
$ git checkout new-stuff
```

```
$ git rebase master
```

As expected, we now have messages indicating a merge conflict that occurs as a result of the rebase operation and the shell prompt indicates that we are in the middle of rebase operation. Let's go ahead and attempt to resolve all these conflicts.

Let's verify where the conflict has occurred in for the current replayed commit:

```
$ git status
```

Opening animals.txt identifies the source of the conflict:

```
1: cat
2: dog
3: mouse
<<<<<<< HEAD
4: donkey
5: horse
=======
4: cheetah
5: lion
>>>>>>> 575514f... Added some cool animals
```

We will resolve the conflict by keeping the conflicting content from both commits:

```
1: cat
2: dog
3: mouse
4: donkey
5: horse
6: cheetah
7: lion
```

We will stage and commit this new commit with:

```
$ git commit -am "All animals are here"
```

Next, we continue the rebase with:

```
$ git rebase --continue
```

This time the conflict occurs in cars.txt, and we can check for its source there:

```
1: honda
2: mercedes
3: bentley
<<<<<<< HEAD
4: perodua
5: kancil
=======
4: porsche
5: ferrari
>>>>>>> 70d84b4... Added some cool cars
```

Again, we will resolve the conflict by keeping the conflicting content from both commits:

```
1: honda
2: mercedes
3: bentley
4: perodua
5: kancil
6: porsche
7: ferrari
```

We will stage and commit this new commit with:

```
$ git commit -am "All cars are here"
```

Next, we continue the rebase with:

```
$ git rebase --continue
```

This time the rebase completes successfully with us having manually resolve the conflicts in the two replayed commits in the rebase operation.

We will again check the commit history for all branches with:

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

As expected, we have the two new commits for `new-stuff` rebased on after the `master` branch. The working directory currently contains the latest combined content for both animals.txt and cars.txt that we provided in our earlier manual conflict resolution.

Finally, as in the previous case of rebasing divergent branches, we can now align both the `new-stuff` and `master` branch with:

```
$ git checkout master
```

```
$ git merge new-stuff
```

Now if we check the commit history again with:

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

You can see that both branches are now pointing to the latest commit.

9 Setting up for SourceTree labs

We will basically perform all the previous operations that we performed in the Git Bash shell, but this time using the SourceTree Git GUI client. To do setup for this, first delete all the existing lab folders we have already worked in with the v prefix.

Next copy new folders from the `-original` folders to recreate replicas of the working directory and Git repo contained within:

From `rebase-interactive-lab-original,` create 6 new copies

`rebase-interactive-lab-v1`
until
`rebase-interactive-lab-v6`

From `rebase-divergent-lab-original,` create 3 new copies

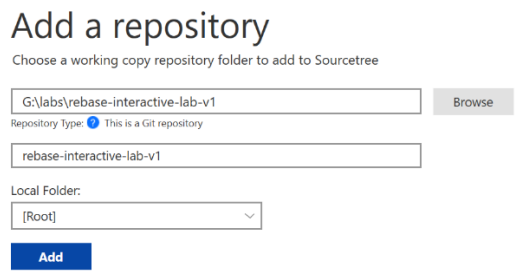
`rebase-divergent-lab-v1`
until
`rebase-divergent-lab-v3`

From `rebase-divergent-conflict-lab-original,` create 1 new copy

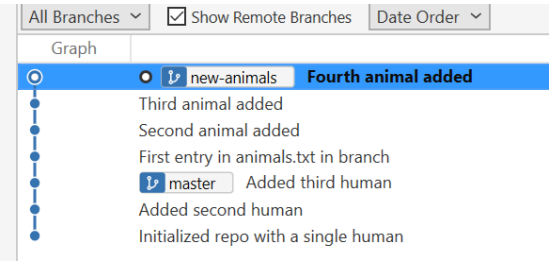
`rebase-divergent-conflict-lab-v1`

10 Interactive rebase for non-divergent branches in SourceTree

In SourceTree, click on the + key to open a new tab, and click on Add. Select the `rebase-interactive-lab-v1` project folder and click Add.

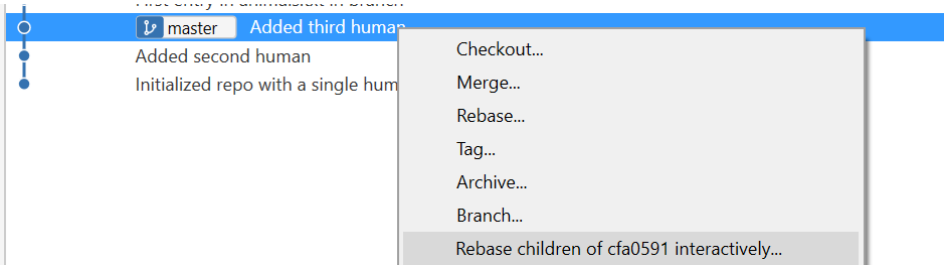


You should see the commit history indicating that `new-animals` is 4 commits ahead of master.

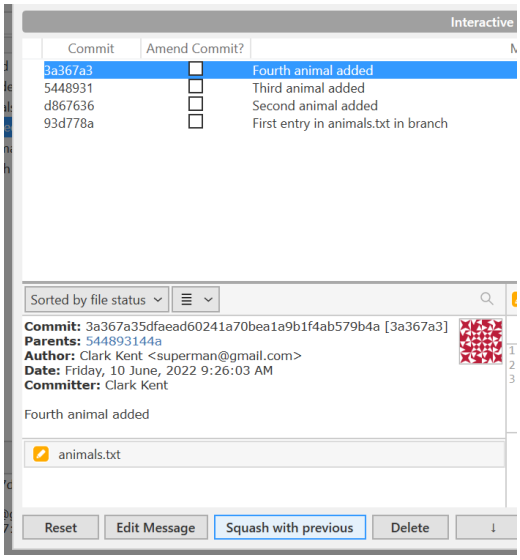


Ensure that `new-animals` is the current / active branch (double click on its commit to be sure).

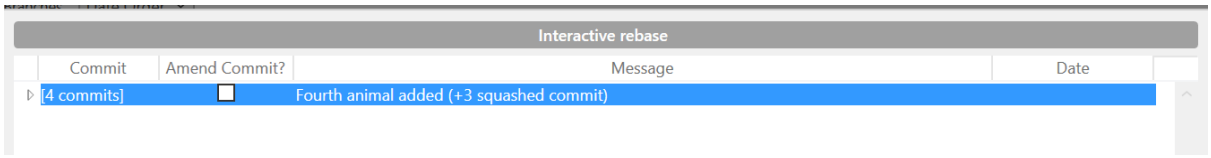
Right click on the master branch and select the option shown below from the context menu. It specifies a rebase operation involving the child commit of master all the way up to the commit of the current branch.



The interactive dialog box appears showing all 4 commits that are to be rebased and allows you to specify the actions to be performed on each commits.
Select the most recent commit (Fourth animal added) and select the button Squash with previous.

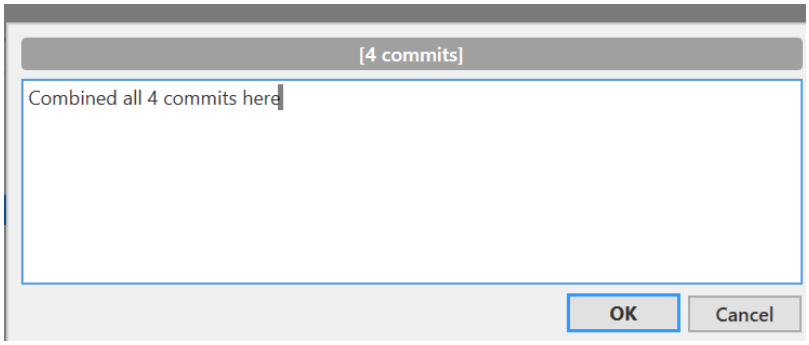


Repeat this until all commits are aggregated (squashed) into that one single commit.

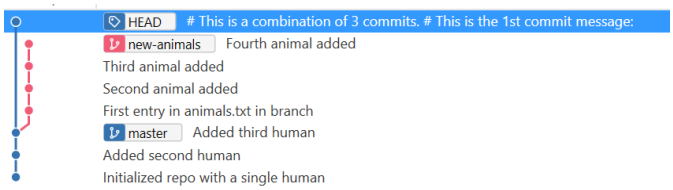


At any point of time, if you make an error or are unhappy with the operations you have specified so far, you can click the Reset button to undo all these operations and begin from scratch.

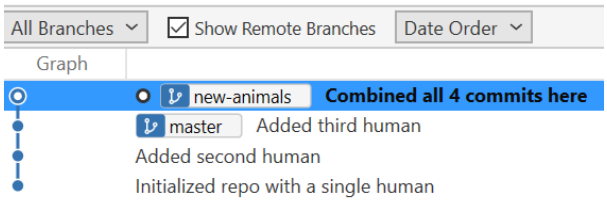
Then click on Edit Message to enter a new message for this final squash commit:
`Combined all 4 commits here`
And click ok.



At this point, all 4 commits are combined into a single commit pointed to by HEAD (as can be seen in the repo commit history), but the rebase operation is not yet complete.



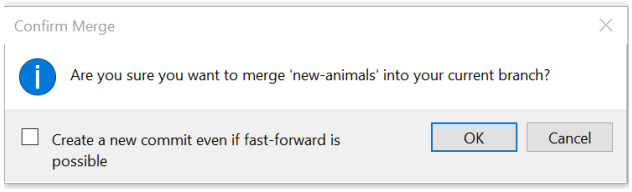
To complete it, double click on either the `new-animals` or `master` branch, after which the commit history should look like the following:



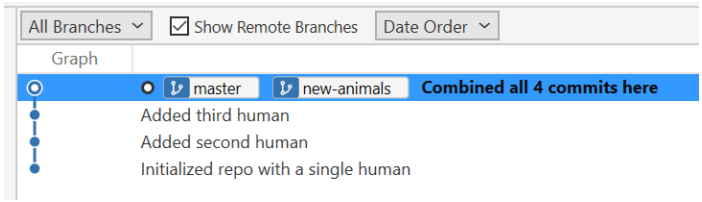
All 4 commits are squashed into a single commit with the new message that you specified earlier, with `new-animals` at this single commit. If you check animals.txt, it still contains all the 4 lines from the last commit.

At this point, we can perform the merge of `new-animals` into `master` :

Select `master` by double clicking on it (this makes it the current branch). Right click on `new-animals` and select merge from the context menu. You are prompted on whether you wish to merge `new-animals` into your current branch, to which you can click OK.



Now both `master` and `new-animals` are pointing to the latest single commit. We have a clean project history now.

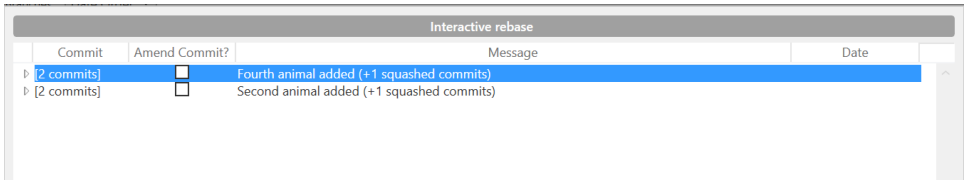


SourceTree does not provide the option to perform a shortcut `git merge --squash` option as we did in a previous lab in order to accomplish the result above.

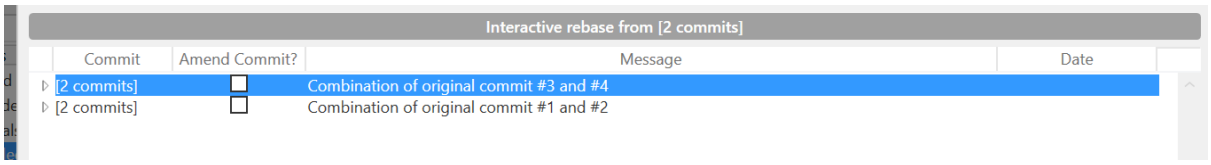
Let's repeat this again, but this time by squashing some of the smaller commits (rather than all of them) into multiple larger commits on the feature branch

In SourceTree, click on the + key to open a new tab, and click on Add. Select the `rebase-interactive-lab-v3` project folder and click Add.

Repeat the process that we went through earlier and in the interactive rebase dialog box, select the Second animal added commit and then select Squash with Previous. Repeat this with the Fourth animal added commit

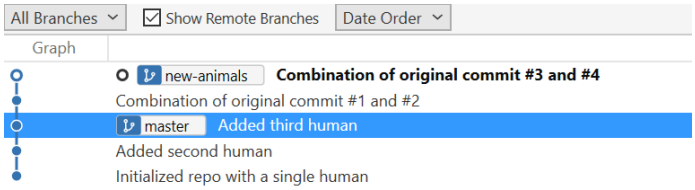


For the squashed commit: `Second animal added` give it a new commit message of: `Combination of original commit #1 and #2`
For the squashed commit: `Fourth animal added` give it a new commit message of: `Combination of original commit #3 and #4`



Finally, click Ok.

We should now see the rebase operation completing successfully with 2 new squashed commits for `new-animals`.

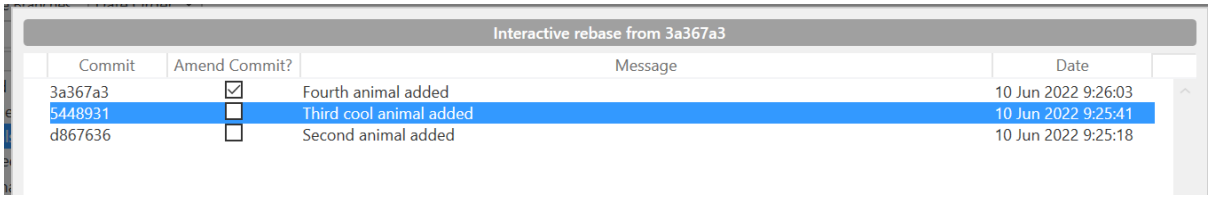


We will now repeat the rebase operation but this time on a subset of the feature specific commits, and specifying different operations (other than squash on them).

In SourceTree, click on the + key to open a new tab, and click on Add. Select the `rebase-interactive-lab-v4` project folder and click Add.

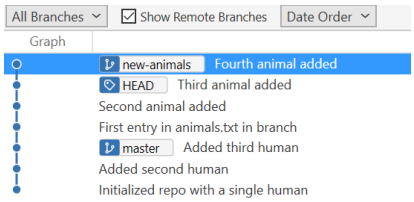
This time right click on the commit `First entry in animals.txt` and select the Rebase children of …. option from the context menu.

We will rename the `Third animal added` commit to: `Third cool animal added`
We will check the checkbox for amending the `Fourth animal added` commit
Then click Ok.

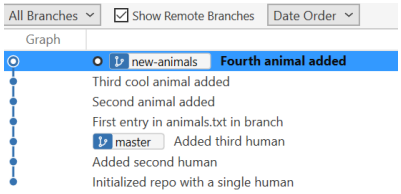


At this point, just as in the previous lab in the Git Bash Shell, HEAD pointer will be in detached state and will be pointing to each new replayed commit until the rebase operation is complete. At this point

of time, the rebase operation is paused at the 2nd commit to be replayed (in this case, `Third animal added`).



To continue the rebase operation, select Actions from the main menu and Continue Rebase. After this HEAD will now move to point to the last commit (which is also pointed to by `new-animals`):



Lets make a modification to this commit. Double click on `animals.txt` to open it up in the default system editor for text files (or open it using an editor/IDE of your choice directly in the folder) and then make the following changes:

Make the following amendment to the last line and save:

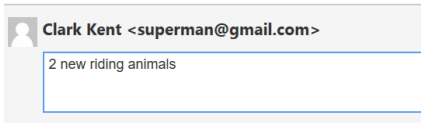
```
4: donkey and horse
```

animals.txt

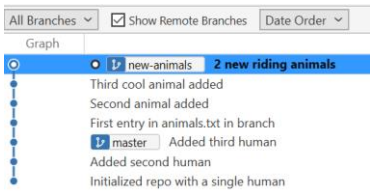
The commit history should refresh to indicate that there are uncommited changes in the detached HEAD. Go ahead and stage it in the usual manner.

We are not going to add these changes as a new commit, but rather to amend the current commit (as we per our original intent in the rebase operation). From the commit options drop down list in the commit box, select Amend Latest Commit. Click No to the dialog box that prompts whether you want to replace the commit text with the message from the previous commit.

In the commit message box, enter the following message: `2 new riding animals` and then click Commit



We should now see the new commit history that includes the 2 rebased commits with the specified actions on them performed correctly:

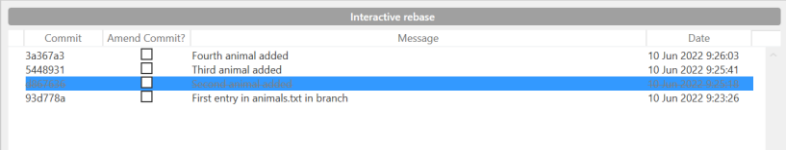


11 Merge conflicts in interactive rebase in SourceTree

In SourceTree, click on the + key to open a new tab, and click on Add. Select the `rebase-interactive-lab-v5` project folder and click Add.

Right click on `master` and select the Rebase children of …. option from the context menu.

Select the `Second animal added` commit for deletion



After clicking Ok, an error dialog box indicating a merge conflict will appear. Click Close to close it.

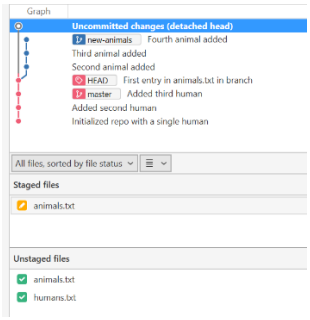
A Merge Conflicts dialog box will appear and you can click Close here as well.
Select the commit with the detached HEAD and double click on animals.txt to open it (or open it directly from the folder using your IDE / editor). You should able to see the conflict:

```
1: cat
<<<<<<< HEAD
=======
2: dog
3: mouse
>>>>>>> 5448931... Third animal added
```

Change the content to this and save:

```
1: cat
3: mouse
```

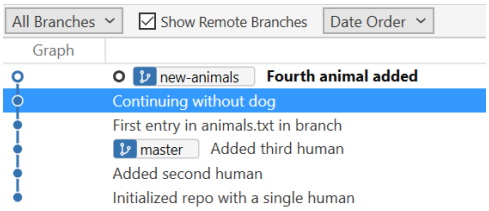
Next select the uncommitted changes entry in the Commit History view listing, then stage the animals.txt file in the usual way.



Then in the commit box, enter this commit message: `Continuing without dog` and click Commit

We will still be able to see the HEAD pointer, which means that the rebase operation is not yet complete and is in progress. To continue the rebase operation, select Actions from the main menu and Continue Rebase.

The final new commit history for `new-animals` will show that the selected commit has been dropped and the third commit has a new message.

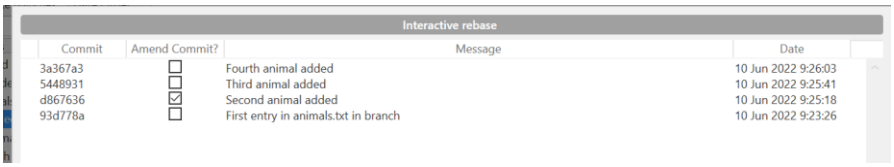


We will now repeat this process again but this time by making an edit to the contents of a commit in the middle of the sequence of commits to be rebased.

In SourceTree, click on the + key to open a new tab, and click on Add. Select the `rebase-interactive-lab-v6` project folder and click Add.

Right click on `master` and select the Rebase children of …. option from the context menu.

Select the `Second animal added commit` as a commit to be amended and then click Ok.



The rebase operation now pauses with HEAD at this commit. Select this commit and double click on animals.txt to open it (or open it directly from the folder using your IDE / editor).

Make the following changes to the 2nd commit by adding the following lines to animals.txt

```
2.1: labrador
2.2: retriever
2.3: bulldog
```

Next select the uncommitted changes entry in the Commit History view listing, then stage the animals.txt file in the usual way.

Then in the commit box, select Amend Latest Commit from the commit options box and click Yes in the dialog box that appears. Replace the commit message with: `List of 3 nice dogs` and click Commit. In the commit history listing, HEAD will now advance to point to this new commit.

To continue the rebase operation, select Actions from the main menu and Continue Rebase.

The Merge Conflicts dialog box appears and click close.

Select the commit with the detached HEAD and double click on animals.txt to open it (or open it directly from the folder using your IDE / editor). You should able to see the conflict:

```
1: cat
2: dog
<<<<<<< HEAD
2.1: labrador
2.2: retriever
2.3: bulldog
=======
3: mouse
>>>>>>> 5448931... Third animal added
```

Change the content to this and save:

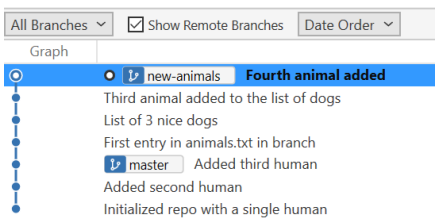
```
1: cat
2: dogs
2.1: labrador
2.2: retriever
2.3: bulldog
3: mouse
```

Next select the uncommitted changes entry in the Commit History view listing, then stage the animals.txt file in the usual way.

Then in the commit box, enter this commit message: `Third animal added to the list of dogs` and click Commit. In the commit history listing, HEAD will now advance to point to this new commit.

To continue the rebase operation, select Actions from the main menu and Continue Rebase.

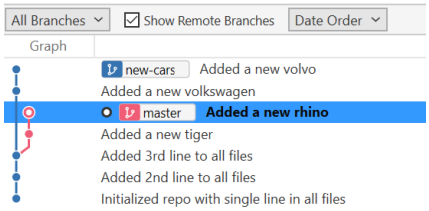
The rebase operation should now complete successfully. In the commit history view listing, we should see the modified commit messages for the 2nd and 3rd commit, and the final and most recent commit contains the last line (`4: donkey`) as well as the additional content we introduced in the 2nd commit.



12 Rebasing divergent branches in SourceTree

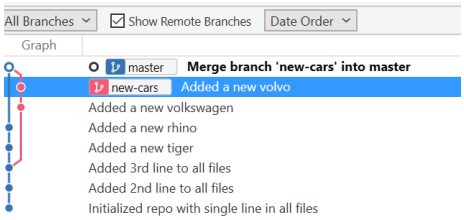
In SourceTree, click on the + key to open a new tab, and click on Add. Select the `rebase-divergent-lab-v1` project folder and click Add.

You should be able to see the divergent history of the two branches in the commit history view listing.



We will perform a simple 3 way merge between the two divergent branches here.
Make sure that master is the current branch (if not double click on it to make it so).

Right click on `new-cars` and select Merge from the drop down menu, and click Ok in the dialog box that appears.
The merge should complete successfully with a new merge commit with a default merge commit message.

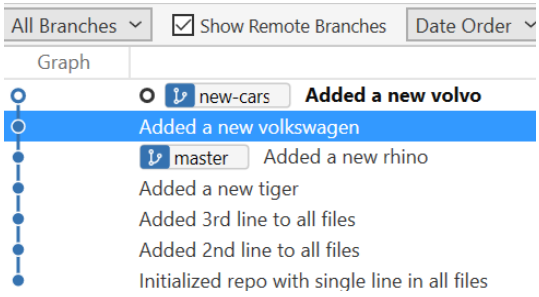


We will now perform a rebase as an alternative to the standard 3 way merge that we have just performed

In SourceTree, click on the + key to open a new tab, and click on Add. Select the `rebase-divergent-lab-v2` project folder and click Add.
The divergent history of the two branches in the commit history view listing is identical as in the start of the previous lab.

Make `new-cars` the active branch by double clicking on it.
Right click on master and select Rebase. Click Ok in the dialog box that appears.

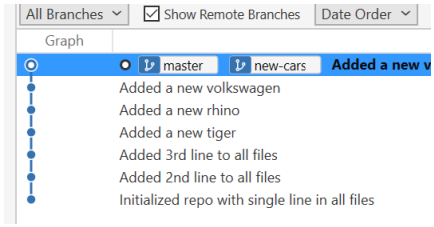
The commit history view listing should now show that all the feature-specific commits of `new-cars` have been added on (replayed) after the `master` branch. If you now check the contents of the cars.txt and animals.txt in the current branch (`new-cars`) , you will see it contains all the latest content from master as well (in animals.txt)



Double click on master to make it the current branch.

Right click on `new-cars` and select Merge. Click on Ok in the dialog box that appears.

The commit history view listing should now show both branches pointing at the latest commit in a linear commit timeline without any more divergent branches.



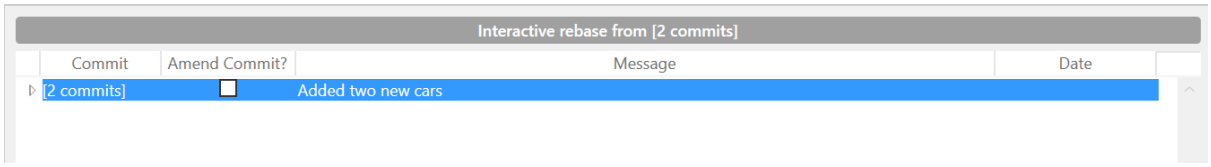
At this point, we have performed a rebase without any modifications to the commits that were rebased. However, we can also perform an interactive rebase here to specify additional operations to perform on the replayed commits, just as we did in an earlier lab.

In SourceTree, click on the + key to open a new tab, and click on Add. Select the `rebase-divergent-lab-v3` project folder and click Add.
The divergent history of the two branches in the commit history view listing is identical as in the start of the previous lab.

From the Git Bash shell, we can combine the interactive rebasing of a branch on its own followed by a rebase into a divergent branch in a single command. However, there is no way to do this in SourceTree. We will first have to perform the standard interactive rebasing and then explicitly follow it with a rebase into a divergent branch.

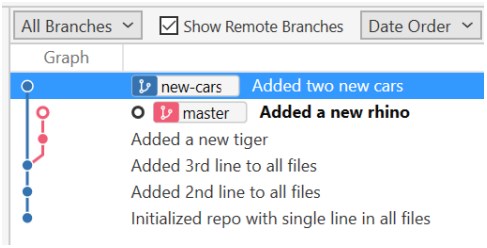
The first thing we will do is to rebase the `new-cars` branch by squashing its two feature specific commits into a single commit.
If `new-cars` is not the current branch, make it the current branch by double clicking on it.
Right click on the commit `Added 3rd line to all files` (the common base commit for the 2 divergent branches) and select the Rebase children of …. option from the context menu.

In the interactive rebase dialog box, specify the merging of the two commits, and give the final commit the message of: `Added two new cars` and click Ok.



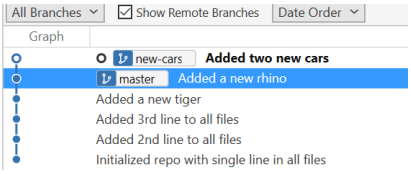
The rebase operation will stop with HEAD at the `Added a new Volkswagen commit`.

To complete the rebase operation, double click on any of the two branches. We should now see the new single commit in the rebased `new-cars` branch.



Next, make `new-cars` the active branch by double clicking on it.
Right click on master and select Rebase. Click Ok in the dialog box that appears.

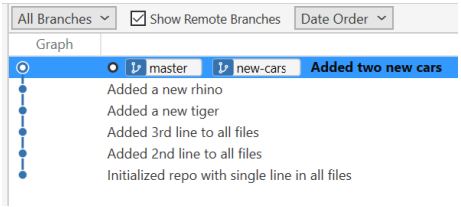
The commit history view listing should now show that all the feature-specific commits of `new-cars` have been added on (replayed) after the `master` branch.



Double click on master to make it the current branch.

Right click on `new-cars` and select Merge. Click on Ok in the dialog box that appears.

The commit history view listing should now show both branches pointing at the latest commit in a linear commit timeline without any more divergent branches.



13 Merge conflicts in rebasing divergent branches in SourceTree

In the previous lab, the merging of the commit content from the rebased branch and the branch being rebased into did not result in a conflict. If it did, we would need to resolve the conflict manually as we have seen in previous cases.

In SourceTree, click on the + key to open a new tab, and click on Add. Select the `rebase-divergent-conflict-lab-v1` project folder and click Add.

We can see two divergent branches: `master` and `new-stuff`

You can see that the commits specific to these two divergent branches contain conflicting content.

We will now rebase `new-stuff` onto `master`

Next, make `new-stuff` the active branch by double clicking on it.
Right click on master and select Rebase. Click Ok in the dialog box that appears.

A merge conflict dialog box appears. Click Close.

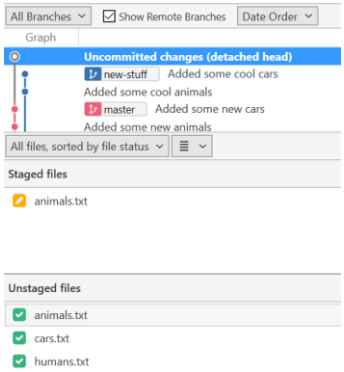
Select the commit with the detached HEAD and double click on animals.txt to open it (or open it directly from the folder using your IDE / editor). You should able to see the conflict:

```
1: cat
2: dog
3: mouse
<<<<<<< HEAD
4: donkey
5: horse
=======
4: cheetah
5: lion
>>>>>>> 575514f... Added some cool animals
```

We will resolve the conflict by keeping the conflicting content from both commits:

```
1: cat
2: dog
3: mouse
4: donkey
5: horse
6: cheetah
7: lion
```

Next select the uncommitted changes entry in the Commit History view listing, then stage the animals.txt file in the usual way.



Then in the commit box, enter this commit message: `All animals are here` and click Commit

We will still be able to see the HEAD pointer, which means that the rebase operation is not yet complete and is in progress. To continue the rebase operation, select Actions from the main menu and Continue Rebase.

A merge conflict dialog box appears. Click Close.

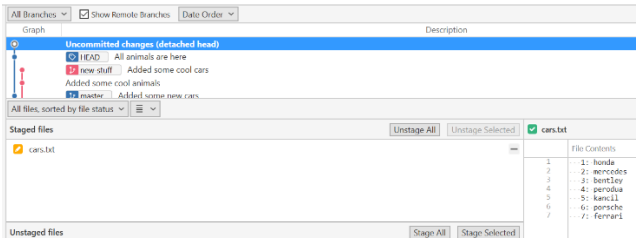
Select the commit with the detached HEAD and double click on cars.txt to open it (or open it directly from the folder using your IDE / editor). You should able to see the conflict:

```
1: honda
2: mercedes
3: bentley
<<<<<<< HEAD
4: perodua
5: kancil
=======
4: porsche
5: ferrari
>>>>>>> 70d84b4... Added some cool cars
```

We will resolve the conflict by keeping the conflicting content from both commits:

```
1: honda
2: mercedes
3: bentley
4: perodua
5: kancil
6: porsche
7: ferrari
```

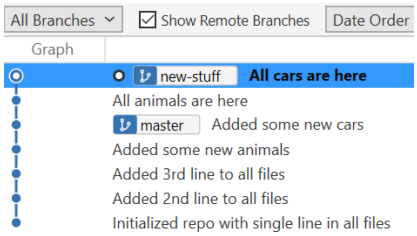
Next select the uncommitted changes entry in the Commit History view listing, then stage the cars.txt file in the usual way.



Then in the commit box, enter this commit message: `All cars are here` and click Commit

We will still be able to see the HEAD pointer, which means that the rebase operation is not yet complete and is in progress. To continue the rebase operation, select Actions from the main menu and Continue Rebase.

This time the rebase completes successfully with us having to manually resolve the conflicts in the two replayed commits in the rebase operation. The two merge conflict commits can be seen in the commit timeline:



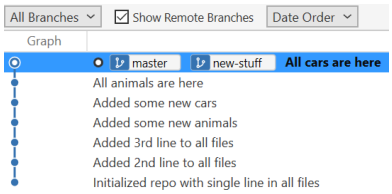
The working directory currently contains the latest combined content for both animals.txt and cars.txt that we provided in our earlier manual conflict resolution.

Finally, as in the previous case of rebasing divergent branches, we can now align both the `new-stuff` and `master` branch.

Double click on master to make it the current branch.

Right click on `new-stuff` and select Merge. Click on Ok in the dialog box that appears.

The commit history view listing should now show both branches pointing at the latest commit in a linear commit timeline without any more divergent branches.



14 Summary

The advantages of using a rebase to integrate the contents of two branches (as opposed to merge) is as follows:

a) There is no extra merge commit in the approach using rebase. This simplifies the commit history timeline, particularly in a large application with many feature branches, there will be a merge commit created every time we merge a complete branch into the main / master branch.

b) The project history now only consists of the commit history of the main / master branch: there are no additional feature branch commit histories to consider.

The most important limitation with rebasing is that you should never use it on branches that have been pushed / published to remote repo that is shared with other developers in a team. In particular, you should **NEVER** rebase the `master` branch onto any other branch, since this branch is by default shared by all team members. This is because rebasing changes the commit history of the branch involved. Consider a sequence of events as follows:

a) You have a branch in your local repo that is not yet in the remote repo. You push this local branch to the remote repo, creating an upstream counterpart

b) One or more team members pull down this new upstream branch into their local repos

c) After this happens, you rebase the same branch in your local repo (thus changing its commit history) then you push your latest changes to update the upstream branch.

d) While this is happening, other team members might be performing development work that is based on the outdated upstream branch.

e) When they finally pull the latest changes to the upstream branch to their local repo, unexpected conflicts will occur causing a lot of problems and confusion.

The warning about rebasing also applies to **ALL** Git commands / operations that can change the commit history of a branch (for e.g. `git reset --hard`, `git commit --amend`).

However, if a branch is local and has not yet been published to a remote repo (where it can be potentially downloaded by other team members), then any kind of operation can be done to it.