InfPALS

Branching,

Merging & Rebasing

OVERVIEW

Welcome to **Task 2** (*deep creepy voice*). You passed the entry requirements for this diamond league of **git**iness and are ready for what is next, **branching & re-basing**, two absurdly awesome concepts you really want to know by heart!

Let's recap what we have done so far in this session. You managed to log into your (new) GitHub account and got familiar with the dashboard there and all those fancy buttons. Then you moved on to initialize a Git folder locally, to link it to your newly created GitHub repository (you learned that cool kids say **repo**), and play around with some committing, pushing, and pulling.

Now, **level-up**, we will make cooperation easier and bring a lot of safety and structure to your git repo.

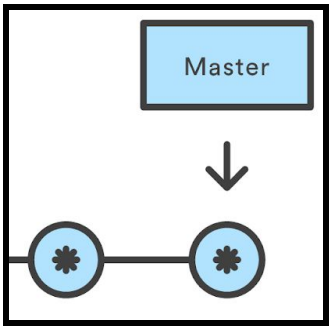
A short explanation

Branching is the action of creating a new branch. At the end of every **push command** (when sending your code to your cloud repo), git prints some output similar to this:

```
git push
Writing objects: 100% (10/10), 1.76 KiB | 0 bytes/s, done.
Total 10 (delta 3), reused 0 (delta 0)
remote: Resolving deltas: 100% (3/3), completed with 3 local objects.
To https://github.com/s0000000/SomeGITrepo
4969ac9..5b76326  master -> master
```

See that highlighted bit? That means that you pushed from branch **master** onto branch **master**.

You may wonder how that looks like logically and it actually is pretty simple:



The blue node on the left is your old master branch. It represents the commit (and push in case of GitHub or any online storage) that was active just before you submitted your new one. Now, by **adding**, **committing**, and **pushing**, Git created a new hash for your new commit and moves your whole project onto the newly created node. Everything you did before now is represented by this new node and if you want to come back to it, you have to go through a tedious process where you have to find out the old hash and then restore that node. And that can not only be really annoying, but also very time consuming. Furthermore, it could be that you want test a new feature or implement something without wanting to change the master branch for now.

Well, here comes branching

Some git Objects (Optional)

*When you stage a file or files, Git creates a container (`git add .`) called a **blob** (binary large object), which stores information about the changes made.*

Whenever you now commit, Git automatically creates a commit object for you which contains

- *a pointer to the most recent blob (snapshot of the content you staged)*
- *the author and message metadata*
- *zero or more pointers to the commit or commits that were the direct parents of this commit*

*What about this random knowledge is now useful? It's the pointer to the blob because that is all that we deal with when **branching** and **merging**. In order to visualize that pointer, you can think of it as the little arrow in the illustration above. So, let's dive into it!*

Branching

First, let's check where we are currently at with `git status`. Git should now tell you that you are on branch master (and that you have nothing to commit, your working directory is clean).

Now, to create a new branch, let's run the command `git checkout -b test`. The core command is **checkout**. It allows you to switch between branches. With the flag **-b** you tell Git that you not only want to checkout the branch **test** but als that it should first be created.

The **git checkout -b** command is the shortcut. The vanilla version is:

```
git branch test
git checkout test
```

To list the branches on your computer, use `git branch.`To list all the branches in a repository, use `git branch -a.` Both will also tell you which branch you are currently on.

To push a specific branch, use `git push origin <branch name>`. This can be done from any branch.

HEAD

Now, while we are still not too deep into this topic, let's discuss the **head** pointer. It is basically the one pointer that represents where you are within your repo.

Side note:

Creating a new branch is pretty schwifty in Git (rather than any other Version Control), as we just have to generate a 40 character long hash code to identify the new branch.

Head keeps track of which branch and commit you are currently working on (and where to point the blobs to when you stage and commit new files). Whenever you use the checkout command, you basically just switch your head from one branch to the other.

Let's try it:

Check your current head with the command `git show-ref --head`. This will give you a listing with all the branches that are currently alive, their respective hash codes, and also (the first entry) the current **Head** pointer. If you compare the hash codes of the Head and your test branch, you'll find that they are equal!

To make sure that this is not black magic, try to switch to the **master branch** and run the command above again. What can you observe?

Remember that you can use **git log** to see an overview of your commit and branch structure, which should come in handy right now.

Changes

While you are in your master branch, why don't you just create a new file? As a shortcut you could use the command `touch newFile1.txt`. Now, stage and commit.

What happens if you now switch back to your **test branch**? Run the **ls** command to see the files that are in your repo.

If you make a commit in one branch, this commit won't be seen in other branches. This also works with the master branch. That means that you can change files and code worry-free, with the peace of mind that everything that worked before, will continue working.

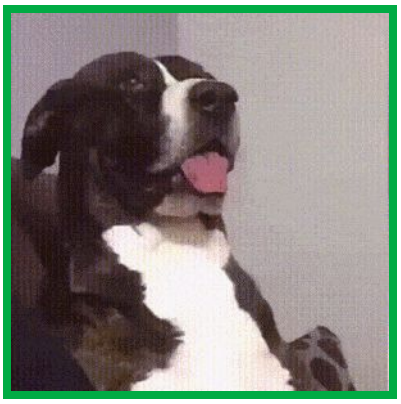
Multiple Branching

Let's assume you did some work in **test**, committed everything several times, and you are about to send it back – merge again with your **master branch** – but you really wanted to change that one thing and test it, but without destroying your code. What can you do? **Right: branch again!** And you can do that as many times as you want!

Exercise #1

Do the following:

1. Create a new folder within your Git repo and call it "MyNewChanges"
2. Create a new text file **first.sh** within it
3. Stage that file and commit
4. Create a new branch **dev1**
5. Add the following code to **first.sh**: `echo "kittens" > first.sh`
6. Now stage that, commit, and branch again to **dev2**
7. Change the code to: `echo "kittens are super cute" > first.sh`
8. Now, check all of your current branches
9. And switch back to **dev1**
10. Oh no! We forgot to commit it! Switch back to **dev2** and
11. Check whether everything is still there in your newly edited **first.sh**!
12. If everything seems fine, stage & commit in **dev2**
13. Now, checkout to **dev1**
14. **Ultimately**, run the command `git merge dev2`



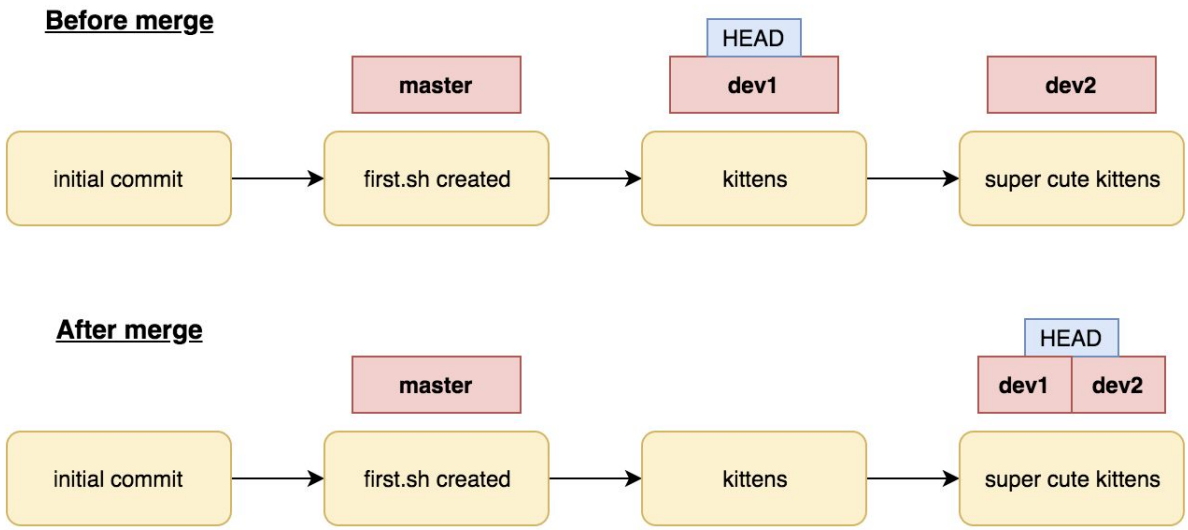
You did some magic right there!

Merging

Whoa, the file **first.sh** had different contents in the two branches that you've created and yet it merged without conflict? What kind of sorcery is this?! Well, this is absolutely normal because there wasn't actually any conflict there to begin with.

When you branched into **dev1**, changed the file and committed your changes, this commit was a direct consequence of your original **master** branch. There was no conflict between the two branches – **dev1** was simply ahead of **master**. The same thing happened when you branched into **dev2**. All the commits made in that branch came directly after the commits made in **dev1**, hence when you wanted to merge the two branches, no conflict arose and the older branch was **fast-forwarded**. This fancy word means nothing more than that the the pointer to the older branch was moved next to the pointer of the newer branch.

Hopefully this diagram will make it even clearer.



You may have noticed that since now two branches are pointing at the same commit, one of them is redundant, so we might as well delete it (don't worry, nothing will happen to the commit!).
Run `git branch -d dev2`
Now check **git log** again to see what changed.

Exercise #2

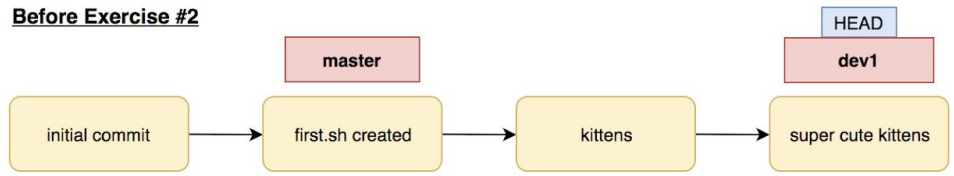
Now that you've tried the simplest kind of a merge, let's have a look at a slightly more complex example. This time, you want your branches to diverge, so that you won't be able to just fast-forward one of them.

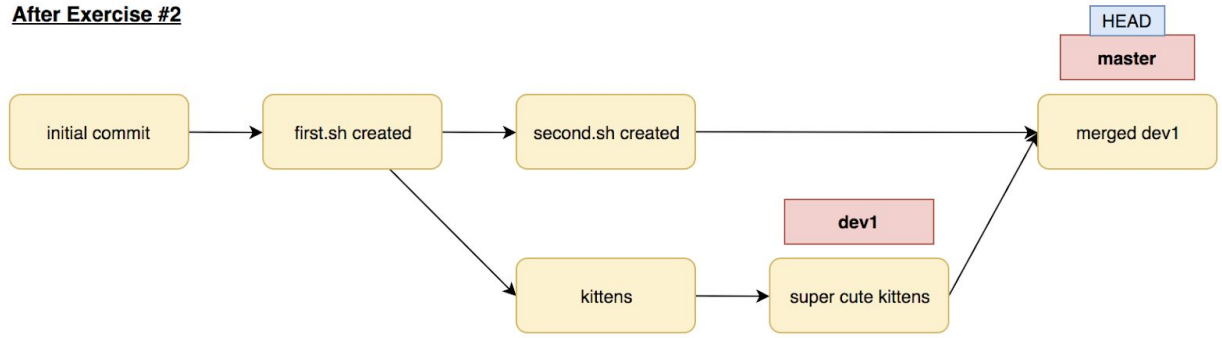
1. First, checkout back into your **master** branch
2. Create a new file called **second.sh** using **touch** or otherwise
3. Add some contents for this file, like: `echo "puppies are pretty cool" > second.sh`
4. Stage and commit your changes and have a look at `git log`
5. Now **pray** a little and then run `git merge dev1`

6. If all went well, just change the message or leave it as it is and **try to** quit `vi`

Well that wasn't so bad now, was it? If you have a look at **git log** again, you should see that your **HEAD** and **master** are at a new commit, which is pointing both to last commit of **master** and to the last commit of **dev1**.

To illustrate this, voilà: another diagram!





Now that you know how to merge without conflict, let's try to break some things. So that we don't have to create new branches again, let's just keep **dev1** for now and reset **master** into an older commit:

Exercise #3

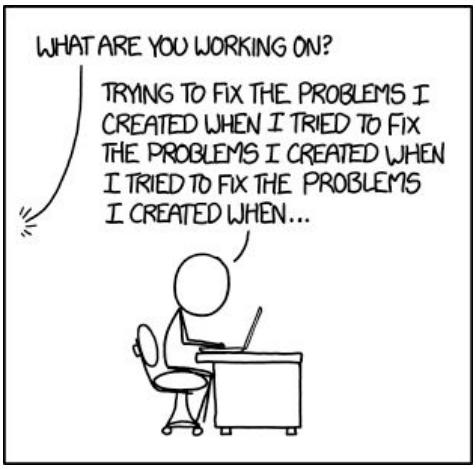
1. Reset your **master** branch into the commit where you added **second.sh** using `git reset fffffff` (where `fffffff` is the hash of the commit you want to revert back to)

   Remember: you can get the hash codes of previous commits using `git log`

2. To get rid of the unstaged changes, run `git reset --hard`
   (**caution!** git reset deletes any unstaged changes and removes newer commits!)
3. Now change the contents of **first.sh** to something like `echo "all animals are awesome" > first.sh`
4. Stage and commit these changes and check **git log** to see where you are
5. And now the fun part: run `git merge dev1`

Before moving on to how to solve this, make sure you see something similar to:
`Automatic merge failed; fix conflicts and then commit the result.`

Merge Conflicts

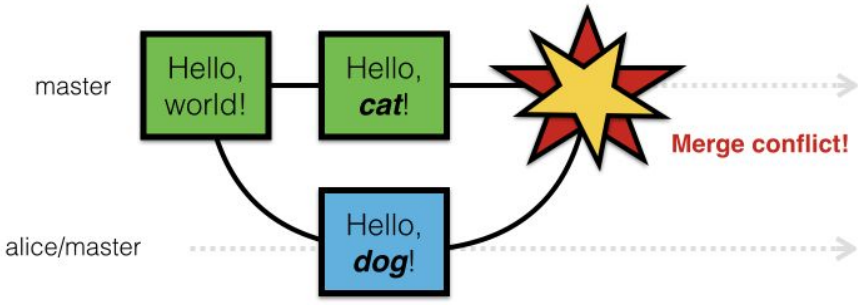


You did it! You really **broke it**. You ask what now? No worries, we got [1] you covered! And don't worry too much, you just experienced one of the many many many (... $many_n \mid n \in \mathbb{Z}$) merge conflicts you will have in your life. So better get **started now**.

As we have seen, you have a pretty unpleasant line in your terminal window, telling you that there's a merge conflict.

This means nothing else than that **fast-forwarding** worked and Git does not really know what it should do, or better, what you would like it to do.

Conceptually you could think of something similar to this:



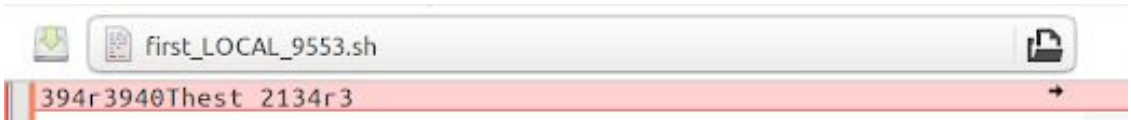
So it means nothing else than that you (or someone on your team if it's online) worked on the same file situated on the parent branch (or the branch you want to merge your current one with). This is no longer tragic as we can use something like a **mergetool**. There is actually a vast variety, but the one which opens when we will deal with some merge conflicts on DICE (**meld**) will be more than enough. Reddit users and similar seem to really like **vimdiff2** but it's more of a personal choice. To see which tools are installed (if you're really interested) run the following command: `git mergetool --tool-help`

Solving the merging conflict - Exercise #4

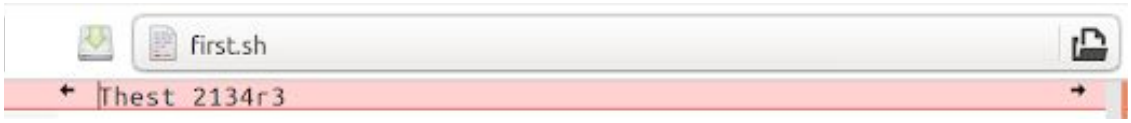
[1] https://www.explainxkcd.com/wiki/images/1/1c/fixing_problems.png

1. First check with the **status** command which files experienced the conflict
2. Now by simply running the git **mergetool** (like the one above but just without the flags at the end) you will find yourself with a newly opened window and the two files.
3. The mergetool shows you which file differs from the one you are about to merge on. This should be displayed with three different sub windows, one with the first.sh from master, (on the left) one with the version from dev1 (on the right), and another one (the middle one) which represents the updated file. Like here:

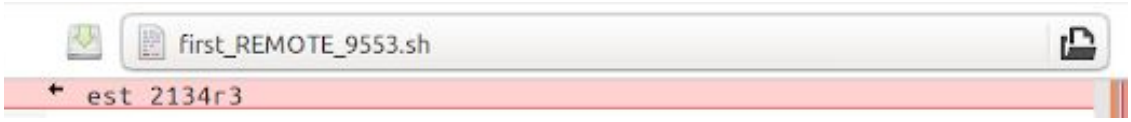
   LEFT

   MIDDLE

   RIGHT

   (don't worry about the content of the file, it differs from yours).

4. Notice that we have also three different file names. **Local, Blank, Remote**. Guess what? Local represents the file that is on the branch you are currently merging into. Blank is the file where we resolve the merge conflict. And Remote is the file from the branch we are merging into our current one. Pretty intuitive, right?
5. Now by simply clicking on the direction arrows next to the errors, choose which changes you want to save and when you are done, click on the big save button on the top left. If you need to redo some things you did, feel free to play around with the arrows on the top and make yourself a bit **confident with the environment**.
6. When you are done, save and close the window. If Git feels like you didn't do a great job (saying, you didn't resolve everything), it asks you whether the merge was successful or not. In this small example it shouldn't have asked that.
7. Ultimately, if you now use the **ls** command in your folder, it could be that it is full of **BACKUP, BASE, LOCAL, REMOTE,**

BACKBASELOCALREMOTEMETADATAwhateverFILE, which you definitely want to get rid of, in order to keep a clean repo.

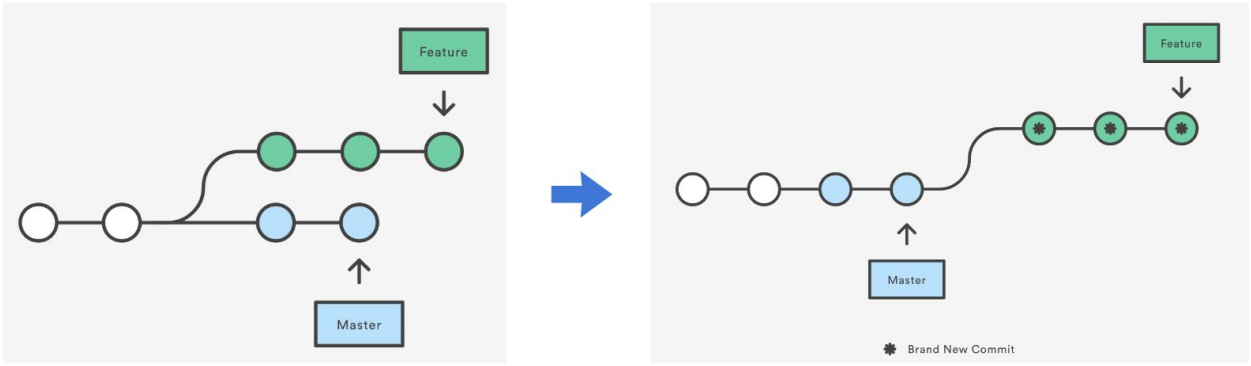
8. Use the following command to see what Git **would remove**:
   `git clean -n`
9. And use this one to **actually remove the meta files:**
   `git clean -f`
10. Ultimately, check your files and you will see that the harmony, the jing & jang of your repo, is again in balance and everyone, even Git is happy. Just remember to add everything and commit it.

This is all the magic.

But there is more (black) magic! - Rebasing

Merging is cool, can make a lot of cool stuff, and can ultimately make your life so much simpler if you use it wisely. However, a lot of companies and developer teams left merging behind as it was just way too messy and switched to **rebasing**.

What sounds like a wild Deorro remix with heavy bass can be visually described as this:



Instead of merging one branch onto the other, we rebase the HEAD to our current modification.

To explain it in words (and this might need a few reads to fully wrap your head around it): you pull the new changes from **master** into your **feature** branch, and then apply all of your **feature** changes on top of it. This means that your **feature** branch will now be a descendant of **master**, which is what is visualised on the last diagram.

So... why rebase?

The major benefit of rebasing is that you get a much cleaner project history. First, it eliminates the unnecessary merge commits required by **git merge**. Second, as you can see in the above

diagram, rebasing also results in a perfectly linear project history—you can follow the tip of **feature** all the way to the beginning of the project without any forks.[2]

Also, in a multi-user environment, where there might be some conflicts to resolve, the burden of making the code compatible lies on you, the maintainer/developer of the **feature** branch, not on the person who pushed into master.

However, the **drawbacks** of it are that you can mess up quite a bit if you don't follow some rules and good practice advises. Therefore we won't go into further detail, and we'll just leave you with "you should definitely check out rebasing, it's pretty darn cool". You should really follow-up on this one and the best resource we found is (as cited) atlassian from BitBucket:

Further Reading

Rebasing: https://www.atlassian.com/git/tutorials/merging-vs-rebasing

Merging: https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

[2] Citation from https://www.atlassian.com/git/tutorials/merging-vs-rebasing