Week 2 - Git

Initialize git repository

```
mkdir project1
cd project1
git init   // created an empty git repository in a subfolder called `.git`.
```

CMLs

1. check status --> `git status`
2. add untracked files --> `git add <file>`
   untracked files - this is a new file that git does not know about yet
3. commit the file with message --> `git commit -m "..."`
4. check records --> `git log`
5. adding ignore files, firstly create `.gitignore` in the directory, then add files into it, then check status and follow the instructions
6. check old version --> `git checkout <...>`
7. return to the latest version --> `git checkout main`
8. make a new branch --> `git checkout -b NAME` & `git push --set-upstream origin NAME`

- `git revert <...>`
  It does not modify the existing commit history but **adds a new commit on top**
- `git reset <...>`
  affects the commit history. In branches where commits are reset, the commit history is rewritten (delete)
  (as if the commits which you've reset had never happened.)
  `--soft / --mixed / --hard`

Git Merge

- Git will merge the specified branch into the **current** active branch
  1. `git checkout <main>`
  2. `git merge NAME` --> with the name of your feature branch

Git Rebase

`git rebase` is a powerful Git command used to integrate changes from one branch into another.

- commonly used to update a **feature** branch with the latest changes from the **main** branch
- cleaning up commit history before merging a feature branch into the main branch

The fundamental idea behind rebasing is **to take the commits from one branch and reapply them on top of another branch**

How Does Git Rebase Work?

When you rebase a branch (let's call it `feature`) onto another branch (let's call it `main`), Git will:

1. Find the common ancestor commit between the `feature` branch and the `main` branch.
2. Temporarily **remove** the commits on the `feature` branch that occurred after this common ancestor.
3. **Apply** the commits from the `main` branch (that weren't in the `feature` branch) onto the `feature` branch.
4. **Reapply** each of the `feature` branch's commits **on top of** the `main` branch's commits.

```
# switch to the branche that want to update
git checkout <branch needed to be rebase>

# does not alter the content of the base-branch
git rebase <base-branch>
```

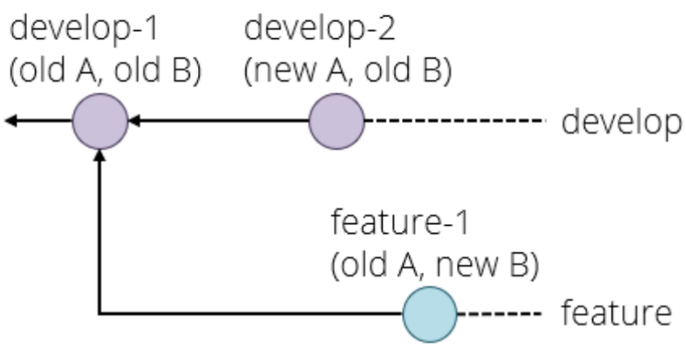
e.g. If branch B created based on branch A
initially, branch A --> File A
after branch, branch B --> File A

now, branch A got --> File A and File B, branch B got --> File A and File C
if, `git checkout branch B`, `git rebase branch A`
then, it won't alter the content of branch A (File A and File B)
but branch B will update (File A, File B, File C)

use the *squash* feature when using *git rebase*

allow combining multiple commits into a single commit. This process is particularly useful for cleaning up a commit history before merging a feature branch into a main branch.

> You are now in the situation that `develop-2` has (new A, old B) and your `feature-1` has (old A, new B). Neither of these is what you want, you presumably want (new A, new B). We have met this situation before, but without branches. Graphically:



- `git rebase <develop>`
- `git push --force origin <feature>` Only force push on private branches

Question review

QUESTION 1

You call fork() and see a return value of 404. Which of the following is correct?

- (●) Fork succeeded, you are in the parent process, and the pid of the child process is 404.
- ( ) Fork succeeded, you are in the child process, and the pid of the parent was 404.
- ( ) Fork succeeded, and the new process belongs to the user with a user id of 404.
- ( ) The call failed because the process you were trying to fork was not found.
- ( ) You need to check the errno variable to see if there was an error during the fork call. If there was no error, then you are in the parent process, and the pid of the child is 404.
- ( ) You need to check the errno variable to see if there was an error during the fork call. If there was no error, then you are in the child process, and the pid of the parent was 404.

---

in Unix-like operating systems, when you call `fork()` to create a new process, **the fork call returns twice**: once in the parent process and once in the newly created child process. The return value is different in each process:

- In the **parent** process, `fork()` returns the Process ID (PID) of the newly created child process
- In the **child** process, `fork()` returns 0

Therefore, if `fork()` returns a non-negative value that is not 0, the call succeeded, and you are currently in the parent process. The value returned is the PID of the child process, which in this case is given as 404.