This document comprises branching and merging steps covered in class on 9/1, but also has some additional steps to resolve conflicts between 2 branches that could not be covered in class.

IMPORTANCE

Branching means making a copy of the current project so that we can work on that copy independently from the other copies, be it other branches or the main branch. Then we can decide whether we want to keep both branches or merge them at some point. This is particularly useful because if you think about how we generally develop software, we work with artifacts. We might need to create a separate copy of your work space to do some experiments, for example. You want to change something in the code; you are not sure it will work out, and you do not want to touch your main copy (main branch). So that is the perfect application for branching. If you are happy with the changes, you will merge that branch with the original one; If you are not happy with the changes, you will delete that branch.

GIT STEPS

Git branch: to see which branches are present. (Until this point of the demonstration, we only had one main branch)

Git branch newBranch : to create a new branch

Git branch: We have two branches now, with the current branch (main) as star marked

Git checkout newBranch: To switch to *newBranch*

Git checkout -b testing: To create a new branch and switch to it

Create a new file called *testfile* in *testing* branch, and push it to the remote repository

Echo this is a testfile > testFile

Git add testFile – staged state

Git commit -m "test file added" – committed state

Move to the new branch and merge *testing* branch with *main* branch since we are happy with the changes made in the *testing* branch

Git checkout main

Git merge testing: merge testing branch with main

Let us delete the testing branch because it is no longer of any use

Git branch -d testing

PORTION NOT COVERED IN CLASS

So, something that might happen when you merge a branch is that you might have conflicts, such as changing the same file in two different branches. Let's see an example of that.

Move to the *main* branch and change *newfile* there

Git branch : It shows we have two branches, *main* and *newBranch*

Notepad newfile : Change newfile

Git commit -a -m "new file changed in main branch"

Move to the *newBranch* branch and change *newfile* there

Git checkout newBranch : Change *newfile* again

Git commit -a -m "new file changed in newBranch"

Now, newfile is modified independently in *newBranch* and *main* branch

Move to the *main* branch and merge newBranch

Git checkout main

Git merge newBranch

Conflict message displayed since both branches have independent copies of *newfile*

How to Resolve:

Open *newfile*

You will see annotations showing the different versions in both branches. You can edit that file, decide which version to keep and which to delete, delete the annotations and save the file.

Git commit -a -m "merged version of newfile" – Git already has merged the branches.

Git branch -d newBranch: We have now resolved the conflict and can delete the newBranch