

Exercise: Git Rebase and Merge

Due: October 26, 2025

Overview

Recently in class we talked about Git Rebase & Git Merge, and I wanted to provide you with a ‘safe’ way to experience these concepts without subjecting your team projects to this behaviour¹.

The commands below assume that you are using a Windows-based machine with **git** installed, and also assume that you have VSCode extensions like GitLens (or GitGraph) installed.

Instructions

When going through the prescriptive steps listed below, pay close attention to what you are being asked to do.

When asked to take screenshots of your work, place the captured image in a Word (or Google Doc) file, make sure to label the image (i.e., screenshot #1), then proceed with the subsequent step.

Once you have completed the exercise, you will submit a PDF copy of the document that contains all screenshots labelled, and in order.

Project Setup

Inside an empty folder of your choice, run the following (Windows) commands.

```
git init
```

Initialized empty Git repository in C:/git-merge/.git/

These commands initialized a git repository and created the file `feature.txt` that will simulate a feature in a real project.

Now, complete a sequence of separate tasks for our ‘feature’ and add a few commits to our repo. Each command below should be completed individually, and you should double-check the state of your file after each step.

¹The instructions for this exercise were inspired by a blog post from Mason Hu: <https://medium.com/@xiaominghu19922/git-merge-vs-scary-git-rebase-5eceed4badc>

```
git add . && git commit -m "feat: initial commit"
```

```
[main (root-commit) 28abe14] feat: initial commit  
1 file changed, 0 insertions(+), 0 deletions(-)  
create mode 100644 feature.txt
```

```
echo "complete task 1" >> feature.txt
```

Verify that feature.txt contains "complete task 1"

```
git add . && git commit -m "feat: complete task 1"
```

```
[main 0cc4bea] feat: complete task 1  
1 file changed, 1 insertion(+)
```

```
echo "complete task 2" >> feature.txt
```

Verify that the last line in feature.txt is "complete task 2"

```
git add . && git commit -m "feat: complete task 2"
```

```
[main abc78f9] feat: complete task 2  
1 file changed, 1 insertion(+)
```

Now your repo has three commits, and running `git log --oneline` should result in the following output, and **note** that your commit hashes will not be the same as mine.

Also, note that the most recent commit appears at the top of this **stack**, and the oldest commit appears at the bottom.

```
abc78f9 (HEAD -> main) feat: complete task 2  
0cc4bea feat: complete task 1  
3e8c86c feat: initial commit
```

Capture your work.

Take a screenshot of your terminal at this step, and label the image: Screenshot #1:
Terminal after initial commits

Git Merge

Consider a scenario where you are working on a project with other team members and you are responsible for implementing a new feature2. You decide to branch off from the `main` branch and start working on your tasks.

Commits made to feature2

```
git checkout -b feature2
```

Switched to a new branch 'feature2'

```
type nul > feature2.txt
```

The file feature2.txt is added to your working folder

```
git add . && git commit -m "feat: start work on feature2"
echo "complete feature2 task 1" >> feature2.txt
```

Verify that feature2.txt contains "complete feature2 task 1"

```
git add . && git commit -m "feat: complete feature2 task 1"
echo "complete feature2 task 2" >> feature2.txt
```

Verify that the last line in feature2.txt is "complete feature2 task 2"

```
git add . && git commit -m "feat: complete feature2 task 2"
```

Run `git log --oneline` to verify the work that you just completed on the branch.

```
417888f (HEAD -> feature2) feat: complete feature2 task 2
3589b68 feat: complete feature2 task 1
a88afd7 feat: start work on feature2
abc78f9 (main) feat: complete task 2
0cc4bea feat: complete task 1
3e8c86c feat: initial commit
```

Capture your work.

Take a screenshot of your terminal at this step, and label the image: Screenshot #2: Terminal after commits to feature2 branch

Commits made to main

Imagine that simultaneously, a teammate was working on a bug fix that resulted in some commits to `main`. These next few steps will simulate their behaviour that was taking place concurrently with the steps you just completed.

```
git switch main
```

Switched to branch 'main'

```
git checkout -b fix-feature
```

Switched to a new branch 'fix-feature'

```
echo "fix feature bug 1" >> feature.txt
git add . && git commit -m "fix: fix feature bug 1"
```

```
[fix-feature 9cd0dce] fix: fix feature bug 1
1 file changed, 1 insertion(+)
```

Verify that "fix feature bug 1" was added to the end of feature.txt

```
echo "fix feature bug 2" >> feature.txt
git add . && git commit -m "fix: fix feature bug 2"
```

```
[fix-feature 852273b] fix: fix feature bug 2
1 file changed, 1 insertion(+)
```

Verify that "fix feature bug 2" was added to the end of feature.txt

```
git switch main
```

Switched to branch 'main'

```
git merge fix-feature
```

```
Updating abc78f9..852273b
Fast-forward
 feature.txt | 2 ++
1 file changed, 2 insertions(+)
```

With this, your team member's branch `fix-feature` got fast-forward merged into the main branch before your `feature2` is complete. Which results in the following commit history for your project.

Run `git log --oneline` to verify the work that you just completed on the branch.

```
852273b (HEAD -> main, fix-feature) fix: fix feature bug 2
9cd0dce fix: fix feature bug 1
abc78f9 feat: complete task 2
0cc4bea feat: complete task 1
3e8c86c feat: initial commit
```

Capture your work.

Take a screenshot of your terminal at this step and label the screenshot as: Screenshot #4: Terminal parallel commits to main.

1. In VSCode, open the working folder;
2. Next, open GitLens (or GitGraph) to see its visualization of the changes that you have made to `feature.txt`;
3. Take a picture of this graph.
4. Label this picture: Screenshot #5: Graph of parallel commits to main:

Merge main into feature2

At this point, `main` branch is now two commits ahead of your `feature2` branch.

Since you need the bug fix to continue working on your feature, you decide to **merge** `main` **into** `feature2` before completing work on your feature.

```
git switch feature2
```

Switched to branch 'feature2'

```
git merge main
```

Merge made by the 'ort' strategy.

feature.txt | 2 ++

1 file changed, 2 insertions(+)

```
echo "complete feature2 task 3" >> feature2.txt
```

```
git add . && git commit -m "feat: complete feature2 task 3"
```

```
[feature2 99c5752] feat: complete feature2 task 3
```

1 file changed, 1 insertion(+)

Run `git log --oneline` to verify the work that you just completed on the branch.

```
99c5752 (HEAD -> feature2) feat: complete feature2 task 3
bb8dbcc Merge branch 'main' into feature2
852273b (main, fix-feature) fix: fix feature bug 2
9cd0dce fix: fix feature bug 1
417888f feat: complete feature2 task 2
3589b68 feat: complete feature2 task 1
a88afd7 feat: start work on feature2
abc78f9 feat: complete task 2
0cc4bea feat: complete task 1
3e8c86c feat: initial commit
```

Capture your work

Take a screenshot of your terminal at this step and label the screenshot as: Screenshot #6: Terminal merge main into feature2.

1. In VSCode, open the working folder;
2. Next, open GitLens (or GitGraph) to see its visualization of the changes that you have made to `feature.txt`;
3. Take a picture of this graph.
4. Label this picture: Screenshot #7: Graph of merge main into feature2

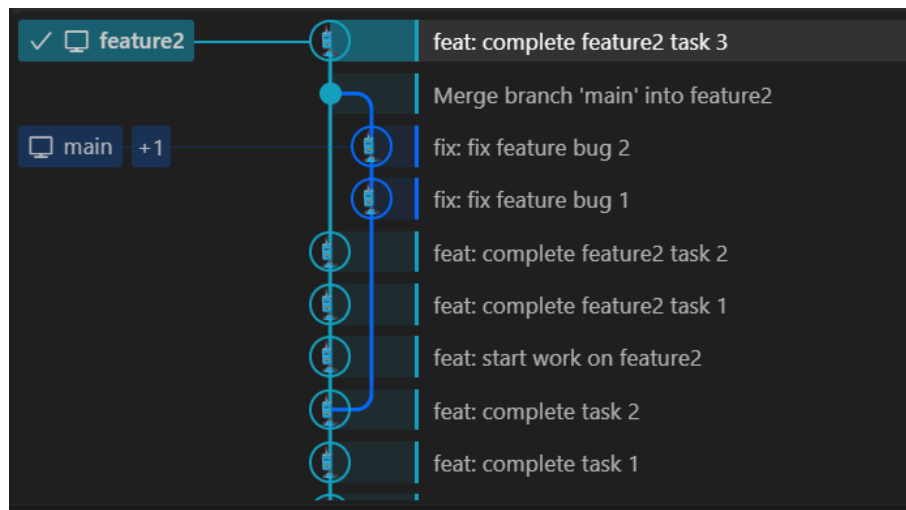


Figure 1: Git Lens: feature2 commit history after code completion and BEFORE merging into main

Notice that an extra commit was created on your branch for merging `main` into `feature2`.

This is how git merge works, it only changes the target branch (i.e., `feature2`), and creates a commit, preserving the history of the source branch (i.e., `main`).

Now that you are done with the feature, it is time to merge to `main` and deploy.

Merge feature2 into main

```
git switch main
```

Switched to branch 'main'

```
git merge feature2
```

```
Updating 852273b..99c5752
Fast-forward
 feature2.txt | 3 +++
 1 file changed, 3 insertions(+)
 create mode 100644 feature2.txt
```

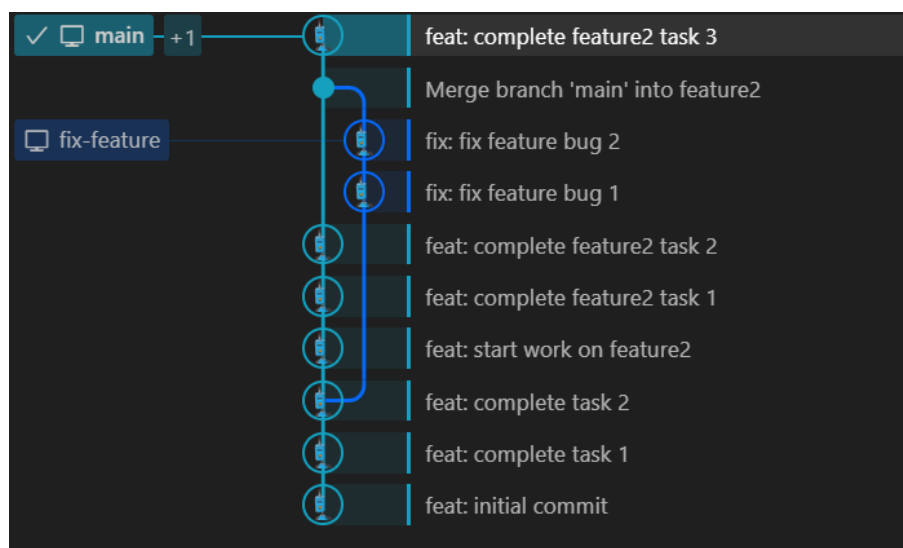


Figure 2: Git Lens: Commit history of main AFTER feature2 merge into main

Capture your work

1. In VSCode, open the working folder;
2. Next, open GitLens (or GitGraph) to see its visualization of the changes that you have made to `feature.txt`;
3. Take a picture of this graph.
4. Label this picture: Screenshot #8: Graph of merge feature2 into main

Looking at the commit history, notice that after running `merge`, all the commits from `main` branch are now connected into the branch `feature2` with a merge commit message `Merge branch 'main' into feature2` which appears after your commits in `feature2`.

Based on the steps that you've just completed, although the workflow for 'git merge' is straightforward to understand and execute, the resulting `main` commit history can get 'messy' due to the full preservation of commit histories from **all** incoming branches.

This only gets messier with an increasing team size, and if someone needs to look into your project's commit history it could be confusing.

Git Rebase

To simplify things, I suggest working from a 'new' working folder that was not previously used with git.

I will go through similar steps as earlier to demonstrate git rebase. I am not going to explain the full scenario again, however I will drop the commands to run from the new working folder for reference.

Initial commits to branch: main

```
#initialize git
git init
type nul > feature.txt

# initial feature release on main
git add . && git commit -m "feat: initial commit"
echo "complete task 1" >> feature.txt
git add . && git commit -m "feat: complete task 1"
echo "complete task 2" >> feature.txt
git add . && git commit -m "feat: complete task 2"
```

Commits to branch: feature2

```
# initial progress on feature2 branch
git checkout -b feature2
type nul > feature2.txt
git add . && git commit -m "feat: start work on feature2"
echo "complete feature2 task 1" >> feature2.txt
git add . && git commit -m "feat: complete feature2 task 1"
echo "complete feature2 task 2" >> feature2.txt
git add . && git commit -m "feat: complete feature2 task 2"
```

Run the command below, and compare the terminal output to what you saw earlier to ensure that you are on the right path.

```
git log --oneline
```

Commits to branch: main after fix-feature development

```
# Your team member fixing a bug
git switch main
git checkout -b fix-feature
echo "fix feature bug 1" >> feature.txt
git add . && git commit -m "fix: fix feature bug 1"
```










Graph	Description	Commit
	 feature2 feat: complete feature2 task 2	c47ab48f
	feat: complete feature2 task 1	1c8d314d
	feat: start work on feature2	14b30a27
	 main feat: complete task 2	59ba52dd
	feat: complete task 1	847589f2
	feat: initial commit	28abe141

Figure 3: Git Graph visualization of feature2 commit history

```
echo "fix feature bug 2" >> feature.txt
git add . && git commit -m "fix: fix feature bug 2"
git switch main
git merge fix-feature
```

Run the command below, and compare the terminal output to what you saw earlier to ensure that you are on the right path.

```
git log --oneline
```












Graph	Description	Commit
	 main  fix-feature fix: fix feature bug 2	46d95da3
	fix: fix feature bug 1	2c0b879a
	 feature2 feat: complete feature2 task 2	c47ab48f
	feat: complete feature2 task 1	1c8d314d
	feat: start work on feature2	14b30a27
	feat: complete task 2	59ba52dd
	feat: complete task 1	847589f2
	feat: initial commit	28abe141

Figure 4: Git Graph visualization of main commit history

At this point, the **main** branch is two commits ahead of the **feature2** branch. And, again, you want to bring the bug fix from **main** into your branch.

This is where git gives you **two** methods to accomplish this goal. Since you have already seen 'git merge', so let's use 'git rebase' to accomplish the goal.

Rebase main into feature2

While on your feature2 branch, run the following command.

```
git switch feature2
git rebase main
```

Successfully rebased and updated refs/heads/feature2.

Notice that after **rebasing**, the commits from the feature2 branch are now connected to the main branch's latest commit in a linear order. This is what rebasing does, it recreates the base for the local branch using another branch as the base reference.

In this case, feature2 = 'local branch' and main = 'reference branch'. The history of the branch feature2 has been rewritten!

Run the command below, and compare the terminal output to what you saw earlier to ensure that you are on the right path.

```
git log --oneline
```

You should see something like this in the terminal after running the command above.

```
43a066e (HEAD -> feature2) feat: complete feature2 task 2
807e0b3 feat: complete feature2 task 1
bb4a2ab feat: start work on feature2
46d95da (main, fix-feature) fix: fix feature bug 2
2c0b879 fix: fix feature bug 1
59ba52d feat: complete task 2
847589f feat: complete task 1
28abe14 feat: initial commit
```

Capture your work

1. In VSCode, open the working folder;
2. Next, open GitLens (or GitGraph) to see its visualization of the changes that you have made to feature.txt;
3. Take a picture of this graph.
4. Label this picture: Screenshot #9: Graph of rebase main into feature2

Now, compare the commit history for feature2 branch before and after rebasing in either the terminal or VSCode. If you look in VSCode, you should see something like the following.





Graph	Description	Commit
	 feature2 feat: complete feature2 task 2	43a066e7
	feat: complete feature2 task 1	807e0b3c
	feat: start work on feature2	bb4a2ab7
	 fix-feature  main fix: fix feature bug 2	46d95da3
	fix: fix feature bug 1	2c0b879a
	feat: complete task 2	59ba52dd
	feat: complete task 1	847589f2
	feat: initial commit	28abe141

Figure 5: Git Graph: feature2 commit history AFTER rebase





Graph	Description	Commit
	 main  fix-feature fix: fix feature bug 2	46d95da3
	fix: fix feature bug 1	2c0b879a
	 feature2 feat: complete feature2 task 2	c47ab48f
	feat: complete feature2 task 1	1c8d314d
	feat: start work on feature2	14b30a27
	feat: complete task 2	59ba52dd
	feat: complete task 1	847589f2
	feat: initial commit	28abe141

Figure 6: Git Graph: feature2 commit history BEFORE rebase

You should see that all the commits on `feature2` branch before rebasing have been completely replaced by new commits after rebasing....look at the images and commit history carefully. This is an important observation so please remember this!

Since the commits on a branch are rewritten when using `git rebase`, this may cause desynchronization between the `git` branches of collaborators if you have previously shared **your** branch with them. This is what makes `git rebase` scary, but it is not that bad if you just adhere to the following mental model:

Never rebase on a public branch that many developers are working on at the same time.

The benefit is obvious if we look back at the commit history...it is linear, which means that your project maintains a clean historical view of contributions.

Submission

Submit a PDF version of your document with naming convention:

RebaseAndMerge-[firstname]-[lastname].pdf to D2L by the deadline.