Lecture Notes: Merging

3.1 Introduction to Merging

Merging is the process of combining two or more commit histories into a single unified history. In simple words, you can say that a merge operation integrates independent lines of development together into a single line. Merging is generally a last-step process in any development activity. When the features have been tested and are ready to be deployed, merging takes place to incorporate the changes in the main development line.

Whenever merging is performed, only the target branch is changed. There is no change made to the history of the branch being merged. Nevertheless, merging plays a very important role in preserving the complete history of the commits made in a chronological order.

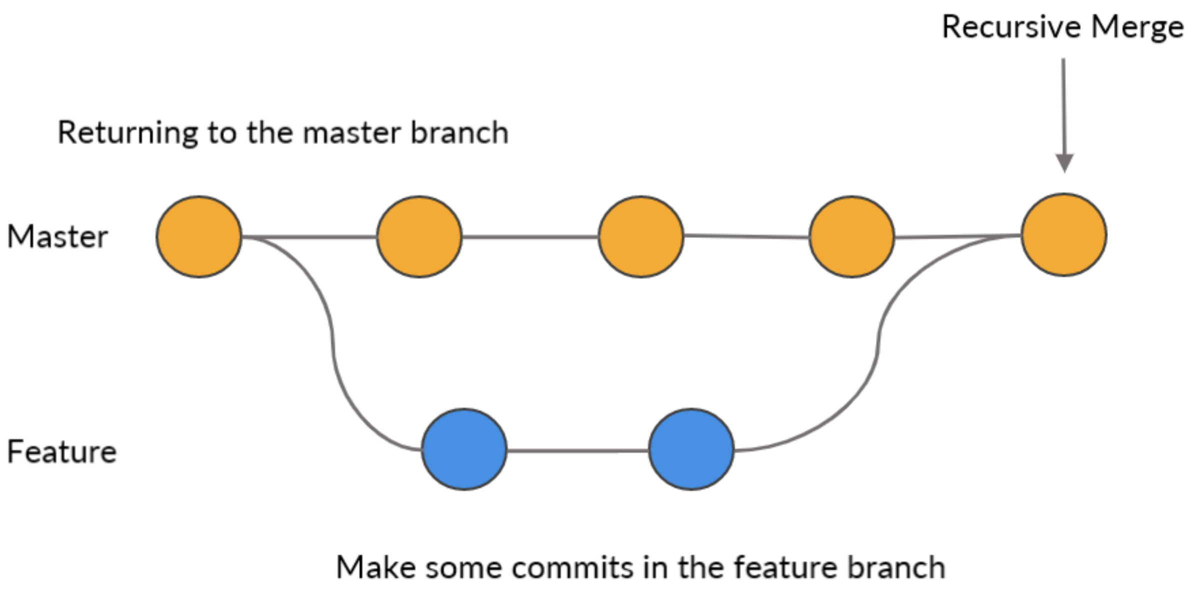
3.2 Different Merging Strategies

Previously, you learnt that merging is the process of combining two or more branches into a single branch. Nevertheless, there are different merging strategies available, letting you choose how you want to perform the merge operation. Now, let's take a look at what happens internally whenever you run the git merge command. Git basically takes the two commits present at the tip of the branches that you want to merge and then tries to find the common ancestor, i.e., the base commit for them. This method by which Git finds this common base is known as merging strategy.

However, you need to understand that if a merge strategy is not specified by the developer, then Git implicitly chooses the best strategy for the available branches. Nevertheless, you can specify your branching strategy by using the -s option along with the name of the strategy that you want to use. -s basically stands for strategy. So, you will now learn about the different merging strategies that are available to you to apply. Here are the strategies

- **Recursive:** It is the default strategy used by Git whenever you pull or merge one branch only. This merge strategy is also known as three-way merge strategy. In this merging strategy, Git recurses over both the branches, and it creates a new merge commit after finding the common base.
  The general syntax to perform recursive merge is this:
  git merge -s recursive <branch_name>.

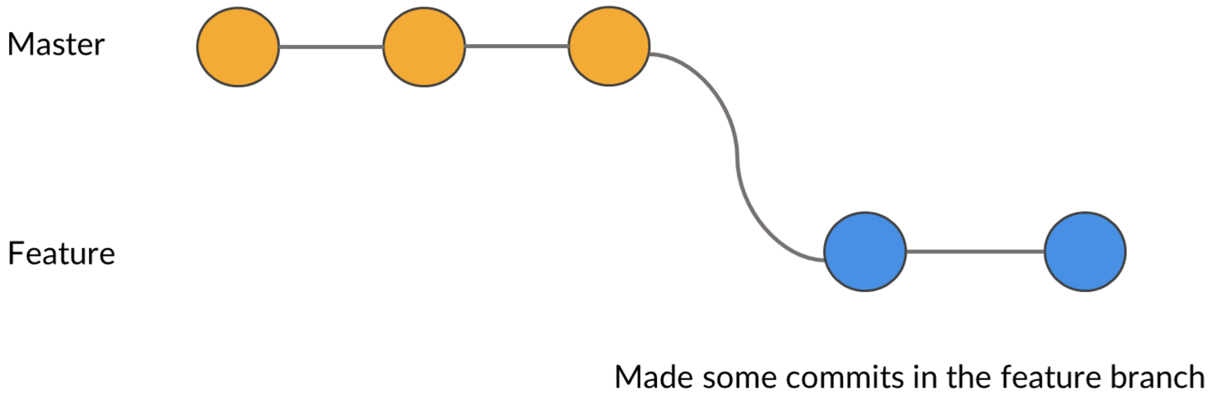
  This diagram shows the working of recursive merge. First, you make a commit on the master branch and then switch to the feature branch and make two commits. After that, you return to the master branch and make some more commits. Now, when you merge the feature branch with the master branch, a recursive merge takes place.


Recursive Merge
Returning to the master branch
Master
Feature
Make some commits in the feature branch

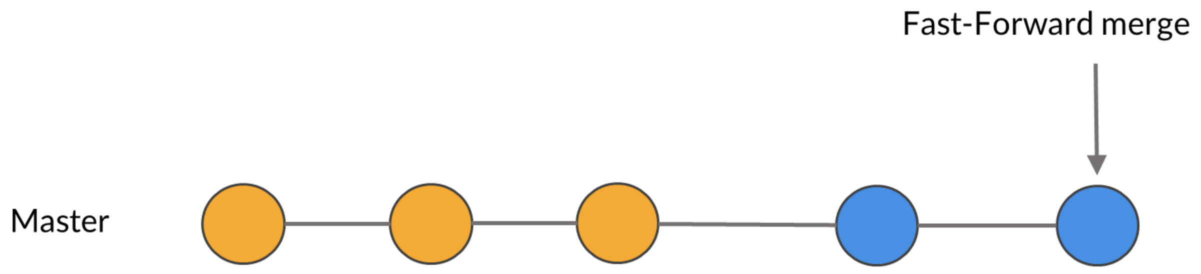
- **Fast-forward merge:** A fast-forward merge is employed whenever you have a linear commit history from the current branch head to the target branch. In a fast-forward merge, only the head is updated and no new merge commit is created. However, there are often scenarios in which we always want to have merge commits in order to have a better understanding of the commit history. In such cases, you can use this syntax to ensure that a merge commit is created:

  git merge --no-ff <branch_name>.

  This diagram shows the working of a fast-forward merge. First, you made some commits on the master branch and then switched to the feature branch and made new commits.

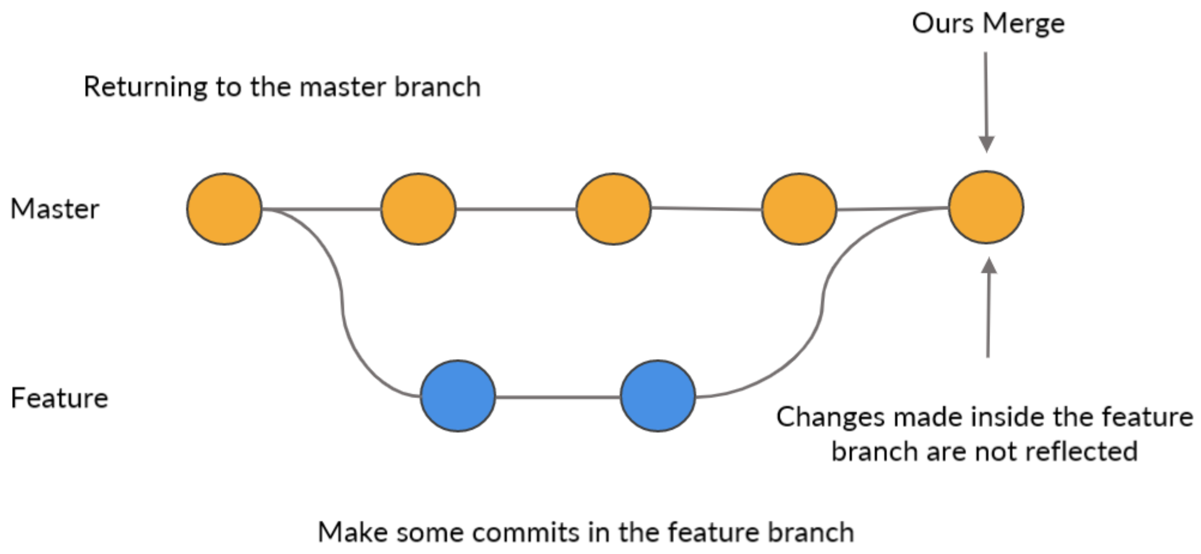


After that, you return to the master branch and find that no new commits had been created since you switched to the master branch. Now, when you perform the merge operation here, then Git, instead of creating a new merge commit, simply moves the head pointer to the topmost commit of the feature branch, thus resulting in a fast-forward merge



- **Ours:** This merging strategy is generally used when you want to ignore the changes in all other branches being merged and consider the changes in the current branch only. This strategy is generally used to leave out the progress made in the side branches. It can resolve any number of heads. The general syntax to apply an ours merge is this:
  git merge -s recursive <branch_1, branch_2...branch_N>.

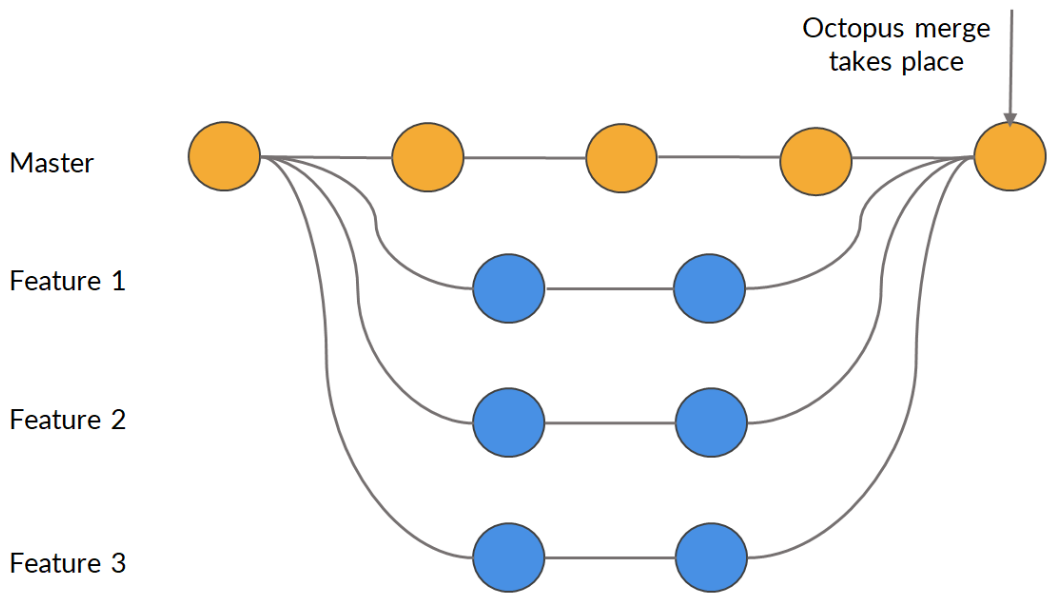
  This diagram shows the working of the **ours** merging strategy. When you have made some commits in the feature branch and need to merge back this branch but want to leave the changes made on it, you can use the **ours** merge strategy.


Ours Merge
Returning to the master branch
Master
Feature
Changes made inside the feature branch are not reflected
Make some commits in the feature branch

- **Octopus:** This is the default merging strategy used when you try to merge more than one branch to the current branch. It is generally useful when you need to merge three or four branches at once. It is advantageous as it does not pollute your commit history, since it integrates all the branches at once. However, it is normally avoided when the number of branches increases, as it becomes difficult to solve merge conflicts that may occur. The general syntax to perform an octopus merge is this:

  git merge -s octopus <branch_1, branch_2...branch_N>.

  This diagram shows the working of octopus branches. You have three branches to merge to the master branch. Now, if you merge the branches one by one, then you will have three merge commits. However, if you use an octopus merge, then you will have a single commit.



2.3 Rebasing and Cherry-Picking

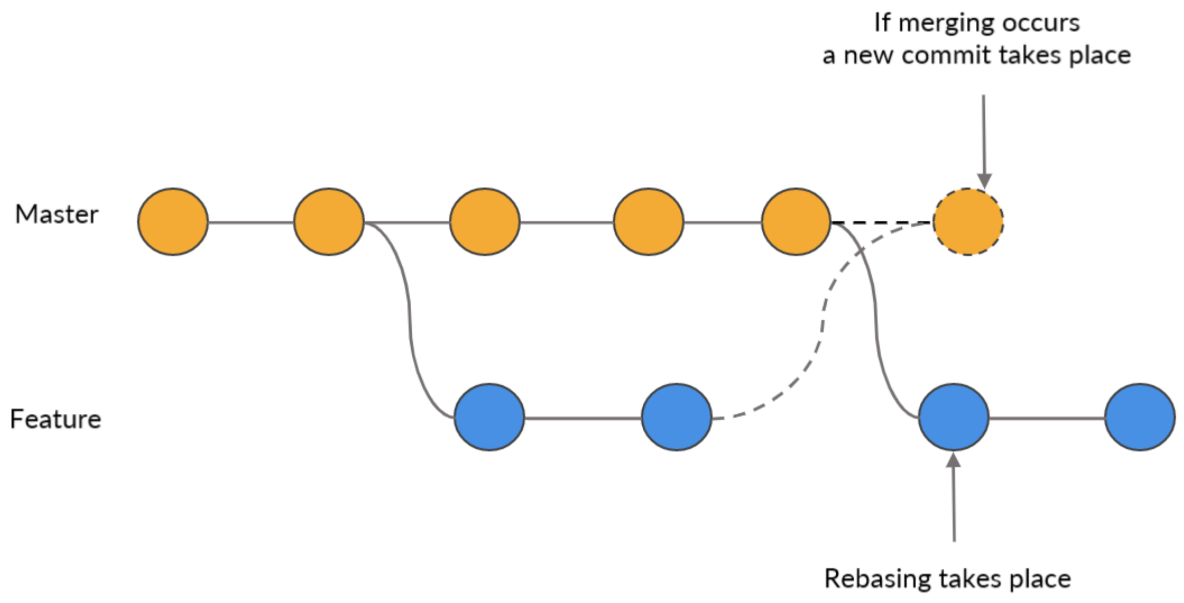
Rebasing

Rebasing is another way to integrate changes from one branch to another. Or you can say that it is an alternative to merging. We often find ourselves in situations wherein we want to have a clean history or do not want to pollute our commit history by creating unnecessary merge commits. In such cases, a rebase operation can be performed. Now, you will learn about git rebase through these points:

- Rebase is technically another method to merge the work in two branches, although it is different from a merge.

- Rebase is the process of moving or combining a sequence of commits to a new base commit.
- Rebase is done to maintain a linear history of commits in a branch and get rid of branches.
- Rebase compares two branches and places the commits in the branch being rebased from another branch in front of the previous commit.

Now, you will see an example to understand git rebase. Suppose you are working on the master branch and created a new branch feature after making some commits. Now, while working on the feature branch, you made some commits, although it may be possible for your master branch to have made progress at the same point in time. Now, if you perform a merge operation, then a new merge commit will be created, with the commits arranged on the basis of the timestamps of their creation. But what if you want to move all of your commits from the feature branch over to the master branch? Well, you can use git rebase here. The diagram below shows the working of **git rebase**.

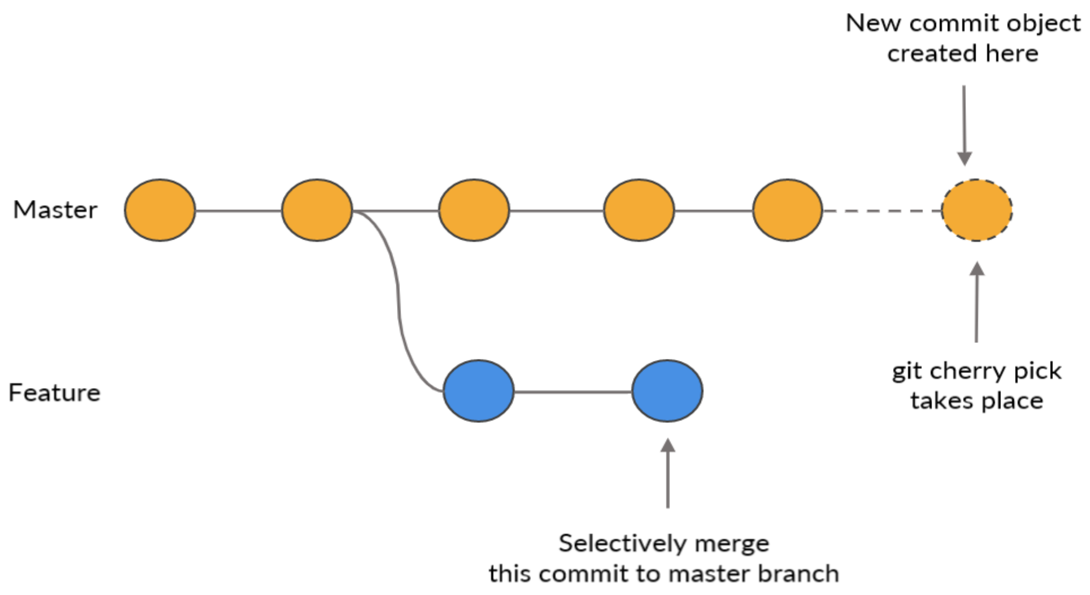


Cherry-Picking

Often, we encounter scenarios wherein we are working on a particular branch and want to bring over some changes from another branch. However, we do not want to bring over all the commits from specific branches. git rebase and merge do not work in such cases. For example, suppose you are working on the development branch and some bug occurred in the production branch. After fixing the bug, the development branch needs to be updated with the update. You would, thus, need only those commits that were involved in the bug fixing, and not all the commits. In such cases, you can use git cherry-pick. Let us try to understand git cherry pick through these points:

- Both merging and rebasing merge all the commits to the master branch.
- Nevertheless, **git cherry-pick** is used when the need arises to selectively pick some commits and then merge to the master.
- Cherry-picking creates a completely new commit object with its own SHA identifier.
- It should be used rarely as it creates duplicate commits.

Now, you will see an example of a cherry-pick operation to understand how it works. This diagram shows the working of the git cherry-pick command.


New commit object created here
Master
Feature
git cherry pick takes place
Selectively merge this commit to master branch

2.4 git fetch and git pull

Whenever a team starts working on a project, the first step that the individual team members perform is git clone. git clone is a one-time activity that creates a copy of the remote repository on their local systems. Now, they make changes in their local repository. And whenever they think that a feature is ready to be merged, they push the changes and raise a pull request. However, other developers will not see these changes in their repository until they update it. One option that they can use to update their repository could be **git clone**. Nevertheless, there is a drawback that is associated with git clone. Each time it is executed, it will create a redundant copy of the remote repository, which you do not want. Hence, the **git pull** and **git fetch** commands are used in these cases. You will now learn how these commands work in detail.

git pull

The **git pull** command is used to pull the changes made in the remote repository to your local repository. The general syntax of the **git pull** command is this:
git pull <options> <repository_url>.
However, **git pull** is considered an unsafe command as it directly fetches all the changes and applies them. This may result in conflicts.

git fetch

The **git fetch** command is also used to pull changes from the remote repository to your local repository. However, the difference here is that instead of merging the changes directly, it places them in a separate branch. The developer can, thus, check the code and can merge it if they think it will not break the flow. Therefore, it is safe to use the **git fetch** command. The general syntax to use the git fetch command is this:
git fetch <options> <repository_url>.

In simple words, it can be said that applying **git pull** is the same as applying **git fetch** followed by the **git merge** command.

2.5 Merge Conflict

You learnt that merging is the process of integrating independent lines of development into a single branch. In most cases, Git is intelligent enough to automatically find out how these branches can be integrated. Nevertheless, in real-time scenarios, we deal with a lot of branches while working in an organisation. Therefore, merge conflict is one of the most common problems that arises in Git. Let's take an example: Suppose two developers are working on the same piece of code in different branches. When they try to merge their code, there are several possibilities that can occur:

- Both the developers may have modified the same line of the file.
- A line that might have been modified by one developer may have been deleted simultaneously by other developers.

Usually, in such situations, Git is not able to decide which changes should be kept and which should be discarded. Thus, Git generates a merge conflict here, and it is the developer's responsibility to resolve these commits before working on anything else.

Steps to Resolve Merge Conflicts

Once a merge conflict occurs, Git generates a descriptive message regarding the conflict. This message can be viewed with the git status command. You need to perform these steps to avoid a merge conflict:

1. Identify the file that is resulting in a merge conflict
2. Open the file using your preferred text editor
3. Remove the conflict dividers and decide what to keep and remove
4. Stage the changes for a commit
5. Commit the changes. The conflict has now been resolved.

Note: In case you want to go back to the previous state before a merge conflict, you need to use the **git merge --abort** command to abort the conflict.

Practices to Avoid Merge Conflicts

Here are some good practices to avoid merge conflicts:

1. Always try to create a new file instead of working on the same file
2. Push and pull changes frequently to keep your repository up to date
3. Always merge your branch as soon as its task is finished