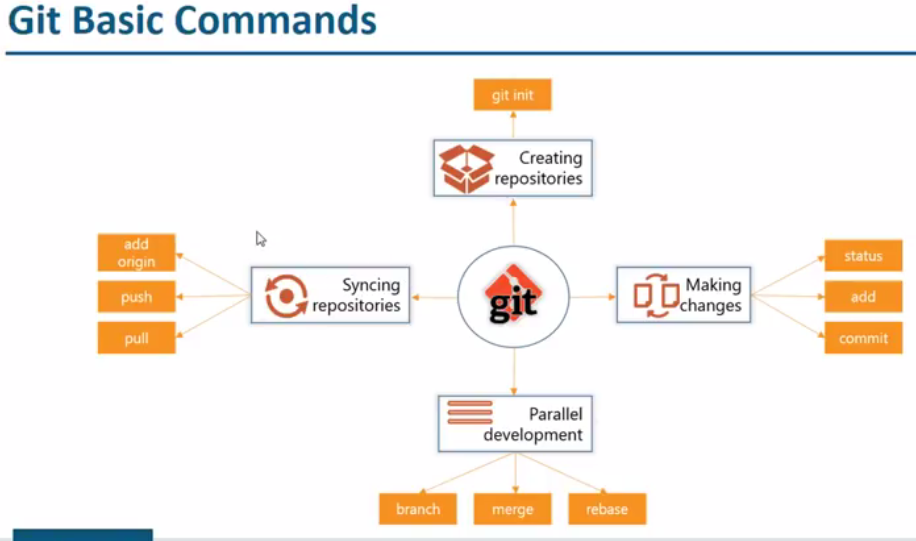
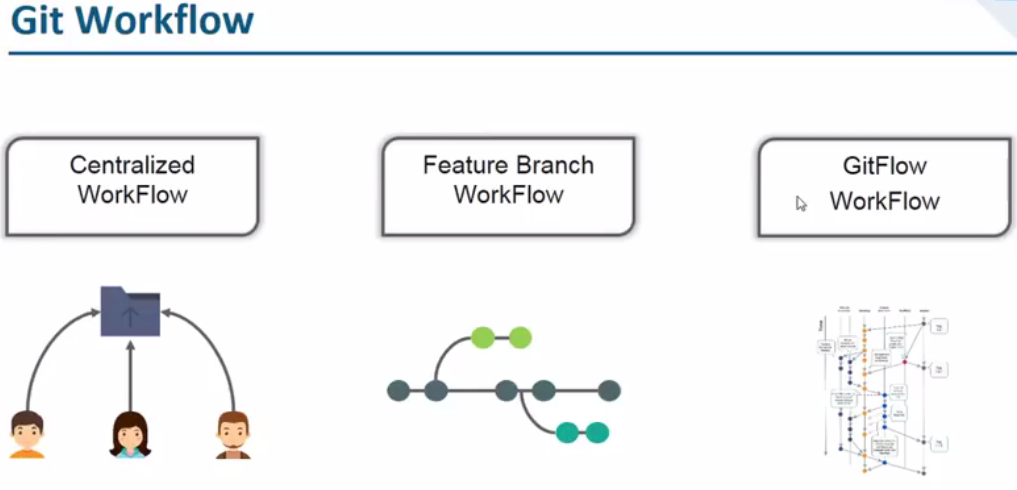
**Git Branching:**

1. local branches

2. remote tracking branches

* Remote tracking branches are connected to master branch, but local branches are not





* Centralised workflow is where all the developers connect and do their work
* Add feature branch workflow means we create different branches for each feature

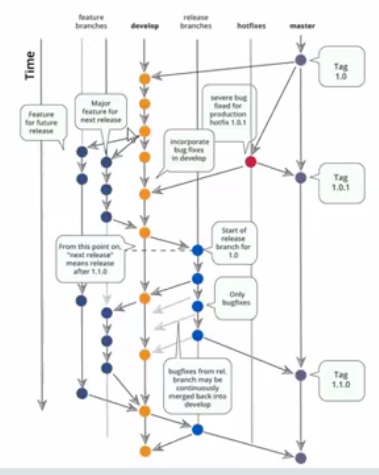
**Centralised workflow:**

* In centralised workflow, the default repository is called master where all the developers connect and commits their code

**Feature branch workflow:**

* Here multiple developers can work on each feature with branches
* For example, developer A created a branch called feature1 and added a file to that, committed and pushed to the central repository
* If we want to pull that branch, we can make a pull request in GitHub

**Git workflow:**



* Instead of having single master branch, we can have two branches. One is for official release and another is for integration
* Many companies use git workflow. All we have to do is, we need to create feature branches, development branches and follow git workflow

**Branching commands:**

**$ git branch (to check the branches)**

**Branch details not getting up without even checking into the branches for at least once with the above command. As the branches won’t getting displayed in local repo without checkout**

**Once we create the branch, we can push it with git push origin <branch name>. it will automatically create the branch in remote as well**

**$ git branch –a (list of branches [local & remote])**

**$ git branch firstbranch (to create a new branch)**

* now this contains all the files from master branch as because we created it from master

**$ git checkout firstbranch (to change to new branch)**

* made some changed in firstbranch

**$ git push --all origin (push all branches to remote repo)**

**$ git branch -r**

**Or**

**$ git branch –remote (to check the remote branches)**

**$ git branch -d [branch name] (to delete a branch)**

**$ git push origin --delete [branchName] (to delete a remote branch)**

* if we have made changes in branches and want to add all those to one branch, then we need to merge it
* when even we are merging, we should be in destination branch

**$ Git branch -d <branchname>**

* To delete the branch

**$ Git branch -D <branchname>**

* We can also use this command to delete the branch. But with this, it deletes unmerged data as well. But if we use -d it will show if any unmerged data is missed. So, better use -d

**$ Git checkout -b <branchname>**

* With this command. New branch will be created and will be checked in to that

**$ git checkout -b [branch name] origin/[branch name]**

* create a remote branch and checkout to it

**$ git checkout –**

* Switch to the last checked out branch

**$ git merge [source branch] [target branch]**

* Merge branch into a target branch

**The changes in local repo will be overwritten by remote repo if we pull that**

**If there are any new files in local repo will not get modified**

**But pushing the changes won’t get overwritten in remote repo**

**If we tried to merge two branches from same user, then there won’t get any conflict**

**Renaming branch name:**

If you are on the branch you want to rename:

* **git branch -m new-name**

If you are on a different branch:

* **git branch -m old-name new-name**

Delete the old-name remote branch and push the new-name local branch.

* **git push origin :old-name new-name**

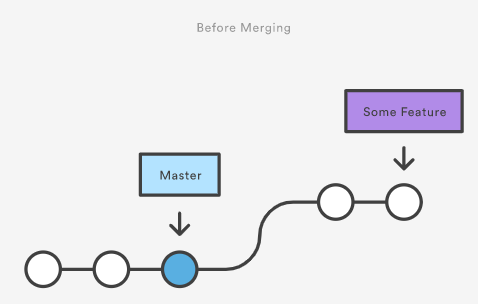
**Git merging:**

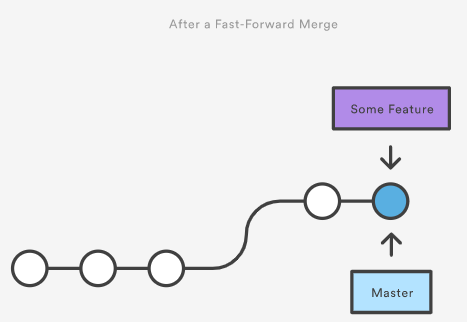
There are different ways of merge as below

1. Fast forward merge
2. Recursive/ 3-way merge

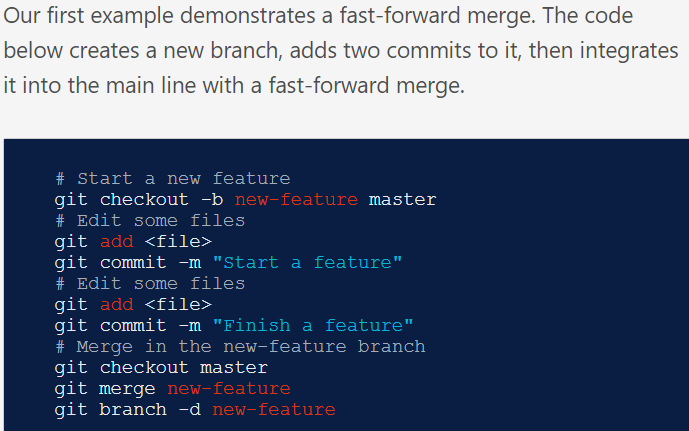
**Fast forward merge:**

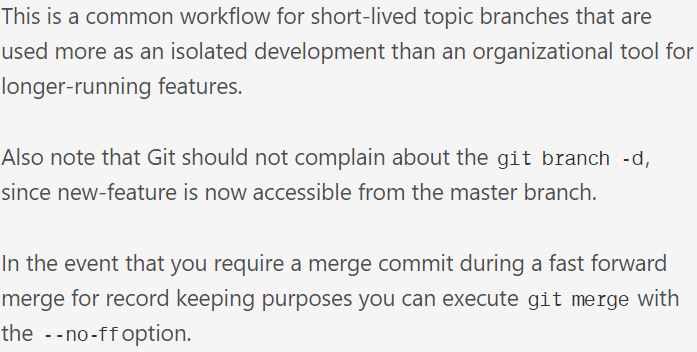
* Git has to do to integrate the histories is move (i.e., “fast forward”) the current branch tip up to the target branch tip. This effectively combines the histories, since all of the commits reachable from the target branch are now available through the current one. For example, a fast forward merge of some-feature into master would look something like the following:
* It takes the tip of the commit ID of current and target branch and merges the remaining commits on top of that.

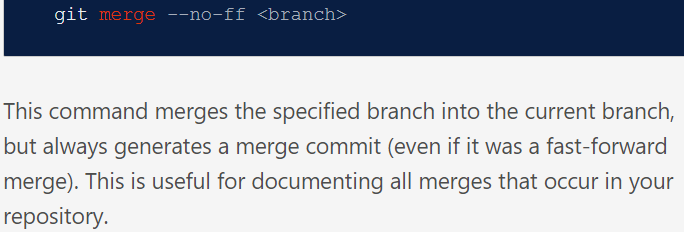




**Example:**

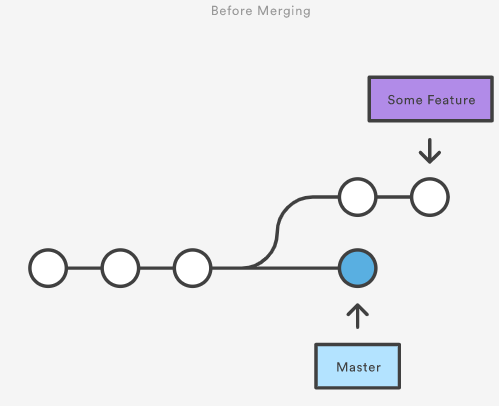


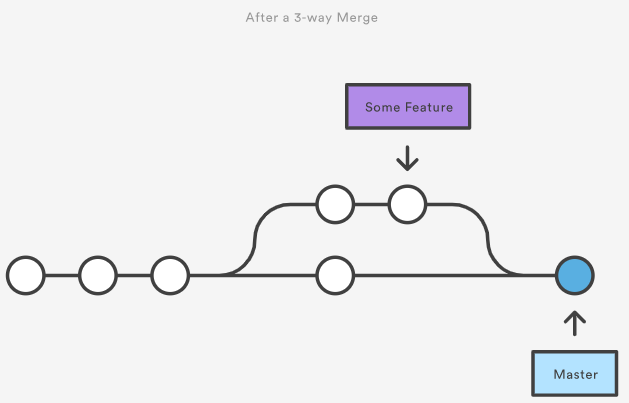




**3-way merge:**

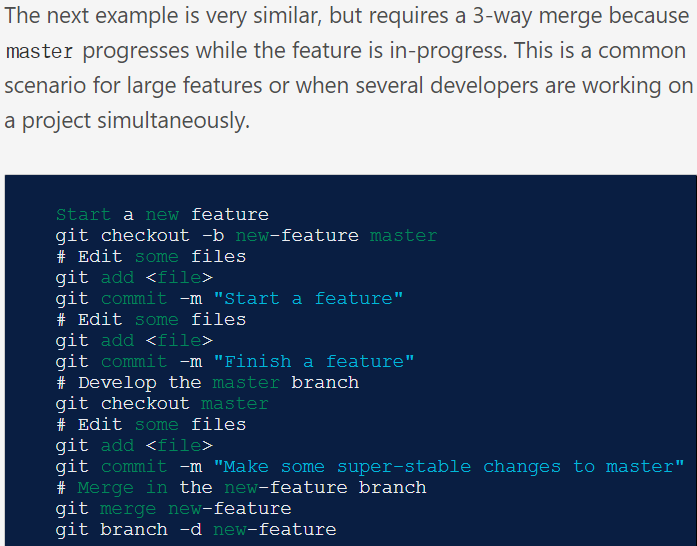
* However, a fast-forward merge is not possible if the branches have diverged. When there is not a linear path to the target branch, Git has no choice but to combine them via a 3-way merge. 3-way merges use a dedicated commit to tie together the two histories. The nomenclature comes from the fact that Git uses three commits to generate the merge commit: the two branch tips and their common ancestor.

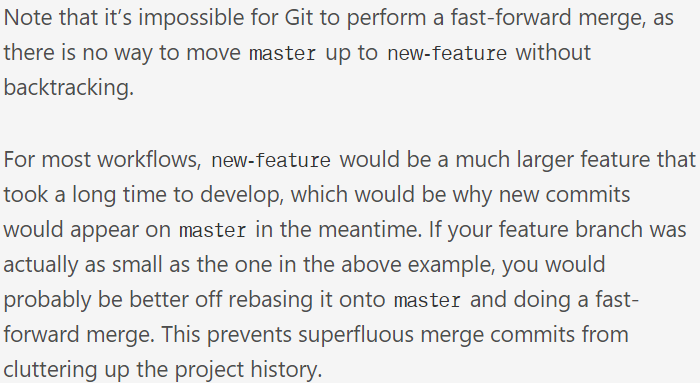




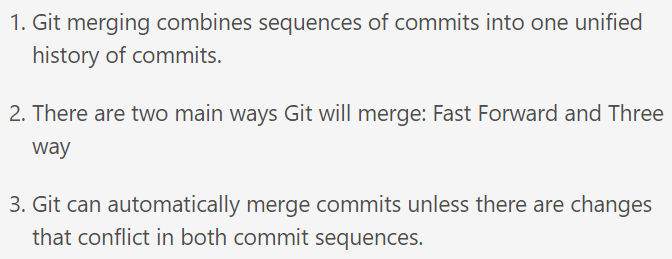
* While you can use either of these merge strategies, many developers like to use fast-forward merges (facilitated through rebasing) for small features or bug fixes, while reserving 3-way merges for the integration of longer-running features. In the latter case, the resulting merge commit serves as a symbolic joining of the two branches.

**Example:**





**Summary:**



**Merge commands:**

* **git merge firstbranch**

go back to firstbranch and made changes to files

* **--no-commit**

With --no-commit perform the merge and stop just before creating a merge commit, to give the user a chance to inspect and further tweak the merge result before committing.

Note that fast-forward updates do not create a merge commit and therefore there is no way to stop those merges with --no-commit. Thus, if you want to ensure your branch is not changed or updated by the merge command, use --no-ff with --no-commit

* **-v**
* **--verbose**

Be verbose.

* **--continue**

After a git merge stops due to conflicts you can conclude the merge by running git merge --continue (see "HOW TO RESOLVE CONFLICTS" section below).

* **--abort**

Abort the current conflict resolution process and try to reconstruct the pre-merge state. If an auto stash entry is present, apply it to the work tree.

If there were uncommitted work tree changes present when the merge started, git merge --abort will in some cases be unable to reconstruct these changes. It is therefore recommended to always commit or stash your changes before running git merge.

* **--no-off**
* **git merge –no-off <branch>**

This command merges the specified branch into the current branch, but always generates a merge commit (even if it was a fast-forward merge). This is useful for documenting all merges that occur in your repository.

* **git merge <tag>**

if we want to merge the code till specific tag

* **git merge <commit ID>**

to merge the code till specific commit ID

* **git merge <commit> <branch>**

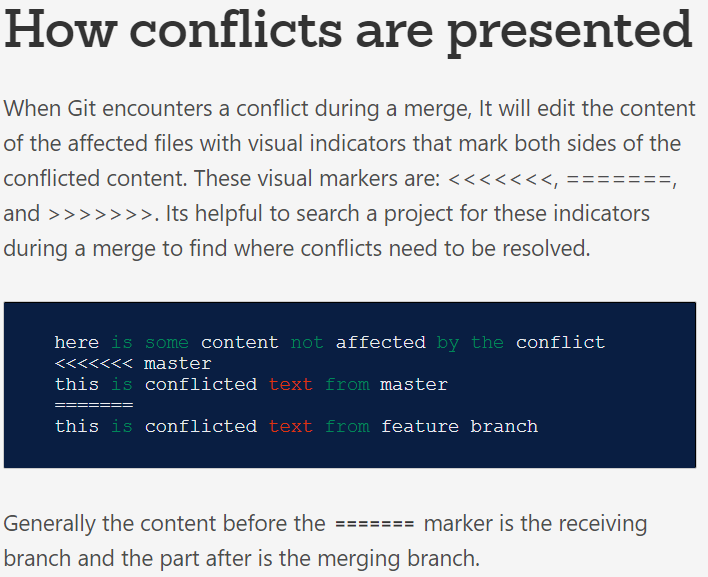
to merge the branch until specific commit ID

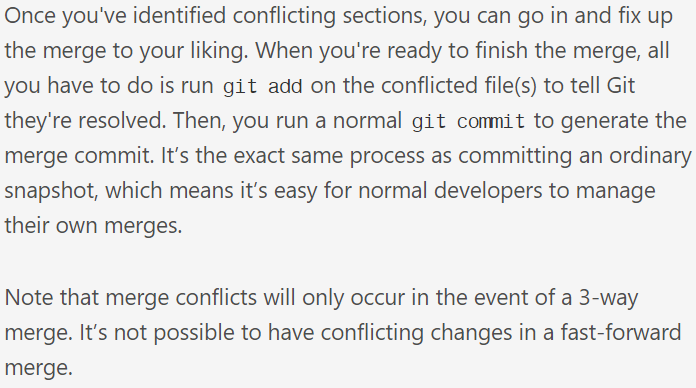
**Resolving conflicts:**





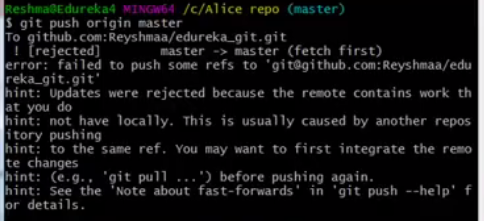
* Above is the error we get when git push conflict occurs





* If there are developers A and B using same central repository. For example, developer A made a change and pushed it to central and developer B tried to change the same file, then there will be a merge conflict as because developer B hasn’t pulled the code which contains the modified data of developer A

**Below is the error we get when conflict occurs:**



**If the users are same in both branches, then there won’t be any conflict. Otherwise we get conflicts if there is any mismatch**