Git: merge, rebase, revert

Agenda

1. Merging changes
    - Merge
    - Rebase
2. Reverting changes
3. Managing conflicts

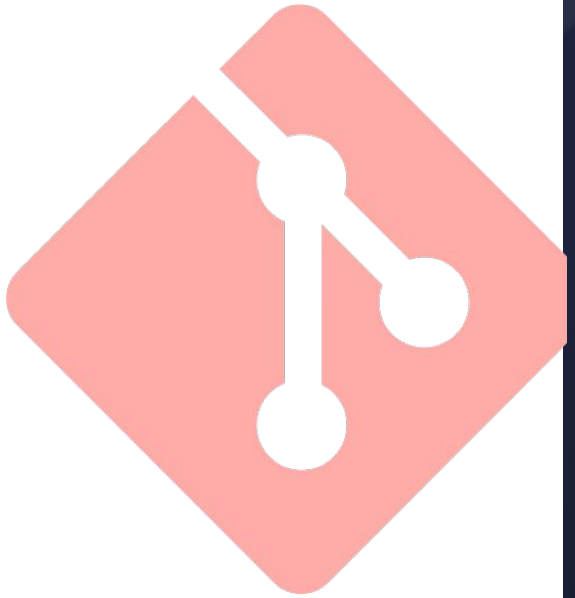
Git

Git is a distributed version control system to handle all kind of projects.

It is so popular that had become the *de facto* industry standard system to version control software projects.

Version control means that it can handle the history of changes that happen to a project, keeping track of all the details about what, when and who performed them. Also, it allows to branch the history and to manage the merging of conflicting changes.

Distributed means that the repository is stored not only in a central place, but a copy of the code exists in all the developer's computers.



What is version control

Merging changes

Merging changes

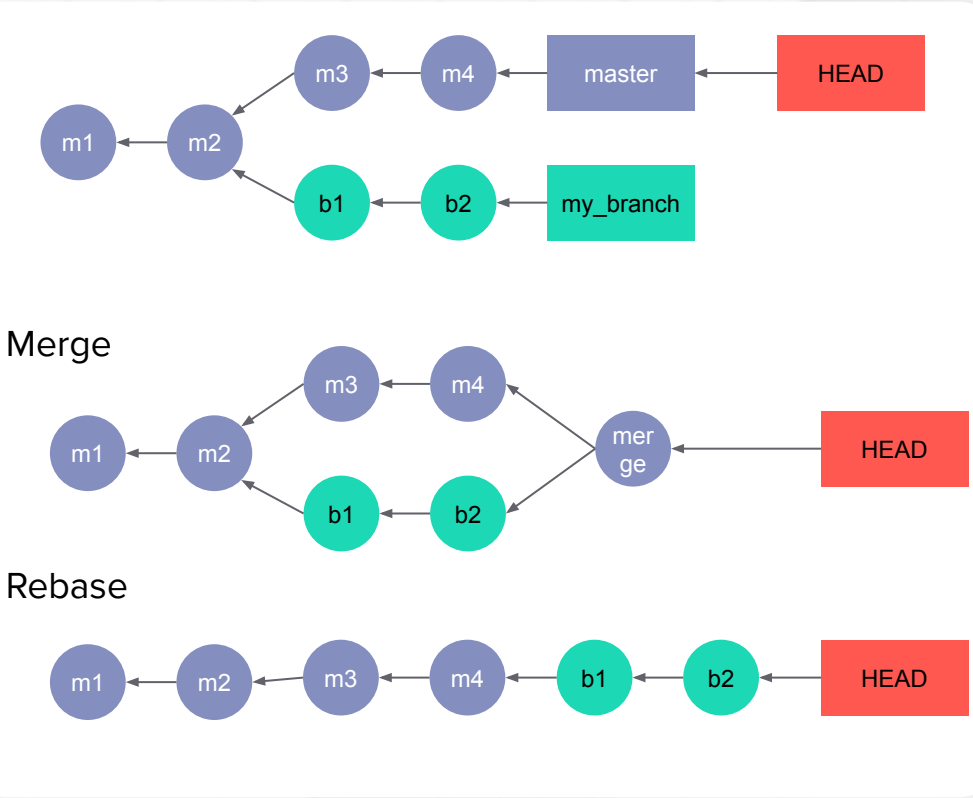
At some point you will need to merge your changes into the trunk or some other branch.

There are two ways:

- **Merge**: join the two branches adding an special merge commit.
- **Rebase**: moves one branch on top of another.

If the two branches contain contradictory changes a **conflict** will arise.

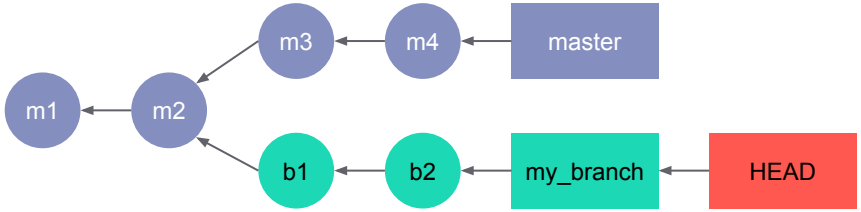
Merging branches



Merge

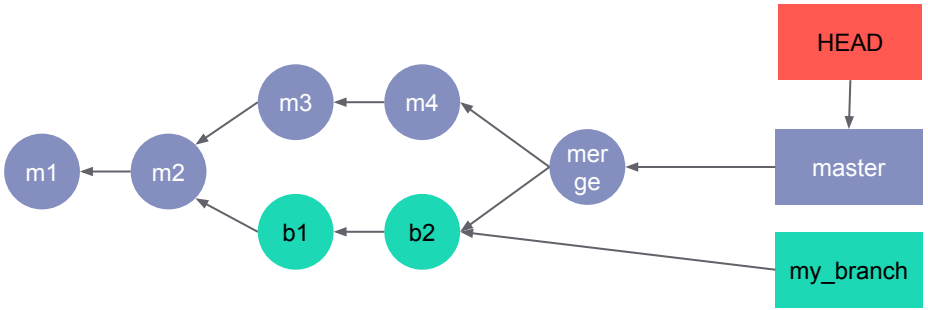
Creates a new commit that contains the changes from the branches you are merging since the commit they diverged.

The new commit will become the tip of the receiver branch.




```
git checkout master // Choose dest. branch
```

```
git merge my_branch
```

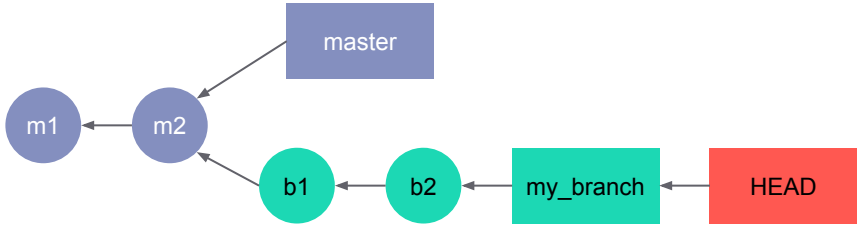


Git Merge

Merge fast-forward

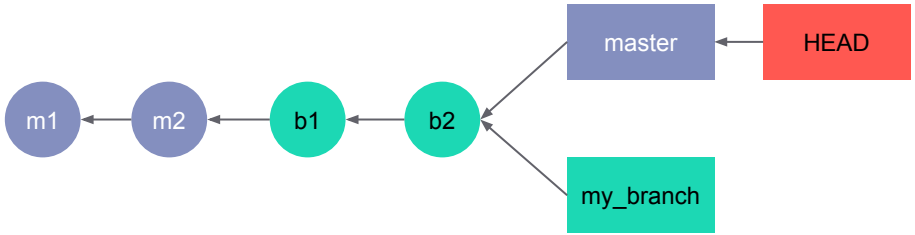
Sometimes, it is possible to do a fast forward merge, so the special commit is not needed.

This happens when there are no new commits in the receipt branch (branches has not diverged).




```
git checkout master // Choose dest. branch

git merge my_branch
```



Fast-forward

Rebase

Moves a branch to the tip of the other.

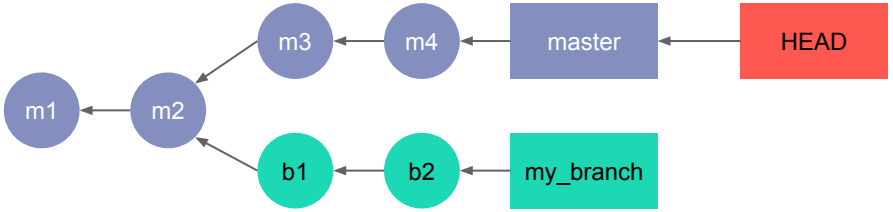
Commits in git are immutable objects, so git creates new commits with the same contents but pointing to the new parents.

This way we bring the changes in the base branch to our current one, paving the way for a fast-forward merge. On the other hand, conflicts will be managed in our branch (more on this later).

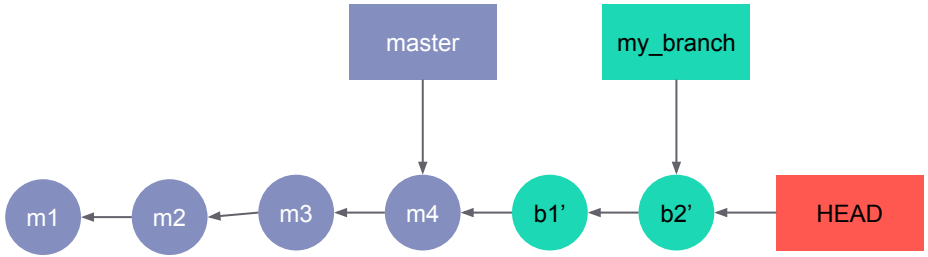
In order to share your changes you will need to do:

```
git push --force
```

Git Rebase




```
git checkout my_branch // Choose dest. branch
git rebase master // Bring changes from master
```



Reverting changes

Modify the last commit

You can add new changes to your last commit.

Git will create a new commit with the contents of the last one and the new changes.

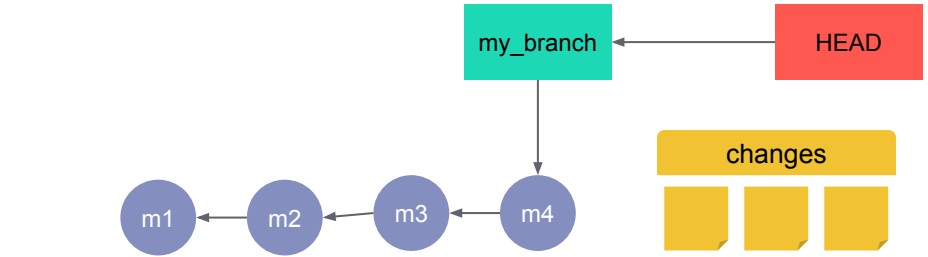
In order to share your changes you will need to do:

```
git push --force
```

To change the commit message:

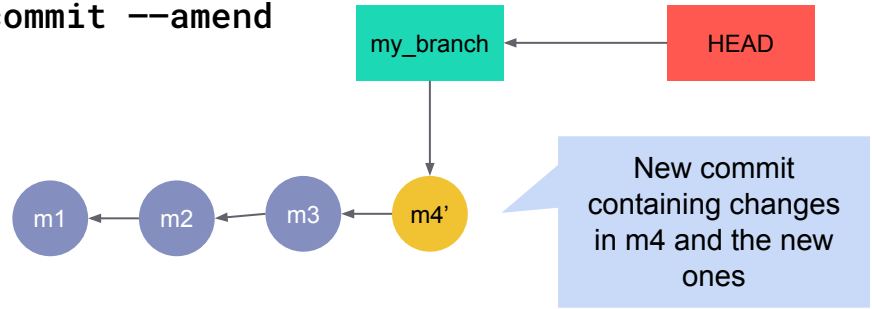
```
git commit --amend -m "New message"
```

Git Amend




```
git add . // stage changes
```

```
git commit --amend
```



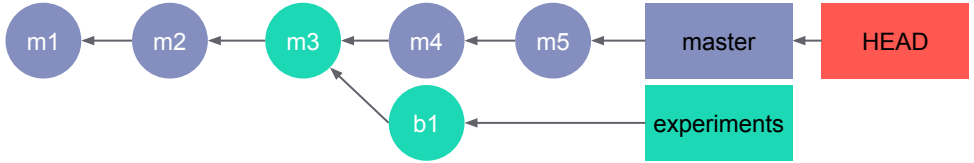
Know the story of the project

You can review the history of the project with git log.

This will allow you to better understand the history of the project and identify each commit. For example, to checkout one specific point in the history and work from there.

Checking out to a specific commit puts the HEAD in detached state (not pointing to a branch).

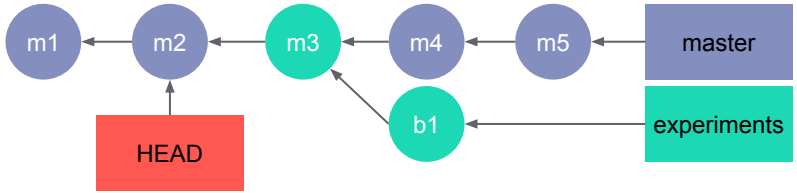
Git Log




```
git log --oneline

    881e258 (HEAD -> master) Arrange code
    ae127fb Convoluted way of thinking
    e7c2d64 (experiments) Leap year without all the objects
    6a05f0e Leap Year kata with extra objects
    9d94bb7 (origin/master, origin/HEAD) initial commit
```

```
git checkout 6a05f0e
```

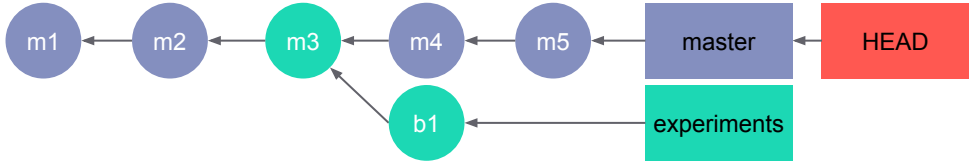


Resetting to a point in the history

Reset moves not only the HEAD, but it also moves the branch pointer.

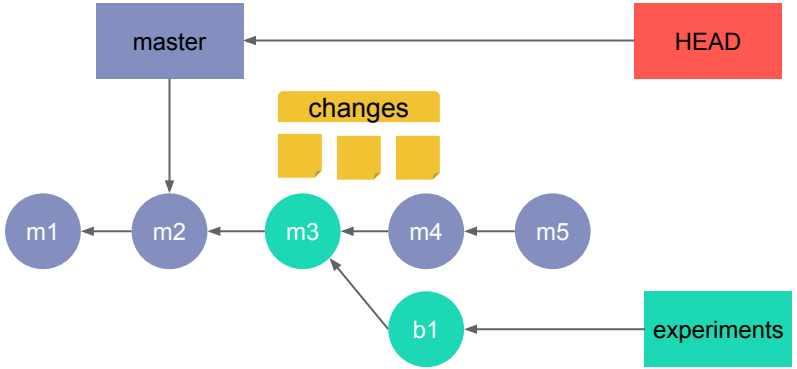
The changes in the commits after the reset destination can be preserved or not depending on options passed to the command.

The history in other branches is not affected.




```
git checkout master // working branch
```

```
git reset 6a05f0e (m2)
```



[Git Reset]

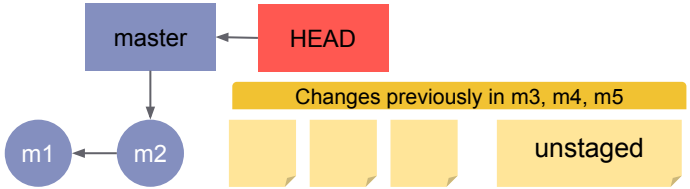
Resetting to a point in the history

What happens with changes? Depends on the following options:

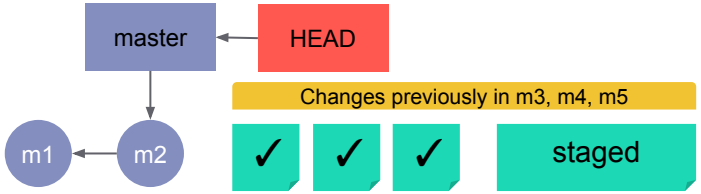
––mixed (default): changes are preserved but not staged for commit
––soft: changes are preserved and staged for commit
––hard: changes are lost

Git Reset

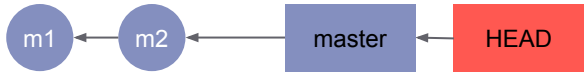
```
git reset 6a05f0e (m2)
```




```
git reset --soft 6a05f0e (m2)
```




```
git reset --hard 6a05f0e (m2)
```



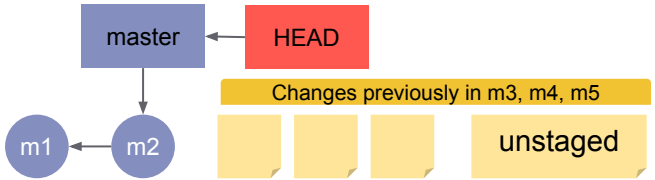
Reset but keeping all changes to work on or commit them with a different organization

––mixed (default): changes are preserved but not staged for commit

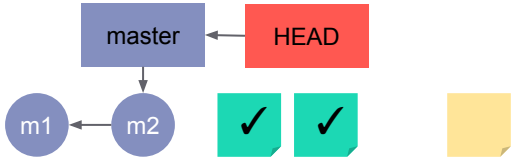
This is the default behavior. Changes are unstaged, so you can perform more changes or commit them.

Git Reset

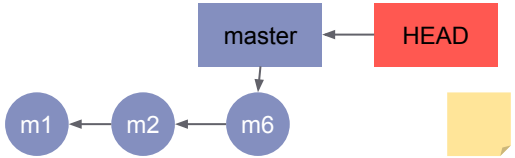
```
git reset 6a05f0e (m2)
```




```
git add file1 file2
```




```
git commit -m “Add some of the previous changes”
```



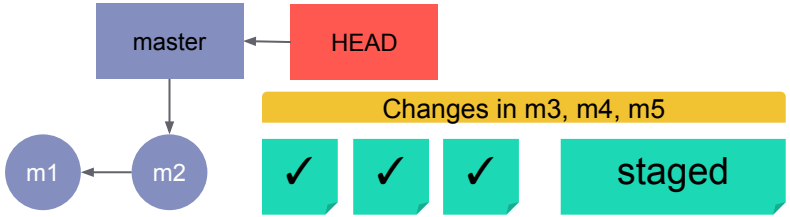
Reset but you want to keep the changes in a new single commit

git reset <commit> ––soft: changes are preserved and staged for commit

Changes contained in the last commits will be marked as staged, so you could commit them again immediately. This could help to tidy up the history.

Git Reset

```
git reset ––soft 6a05f0e (m2)
```



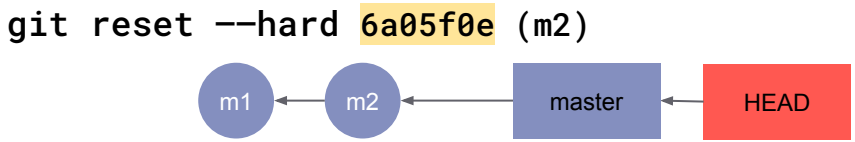

```
git commit -m “All changes joined”
```

Reset and forget changes after reset commit

git reset <commit> ––hard: changes are lost

Use this when you want to remove the last commits and forget about the changes they contain.

Git Reset



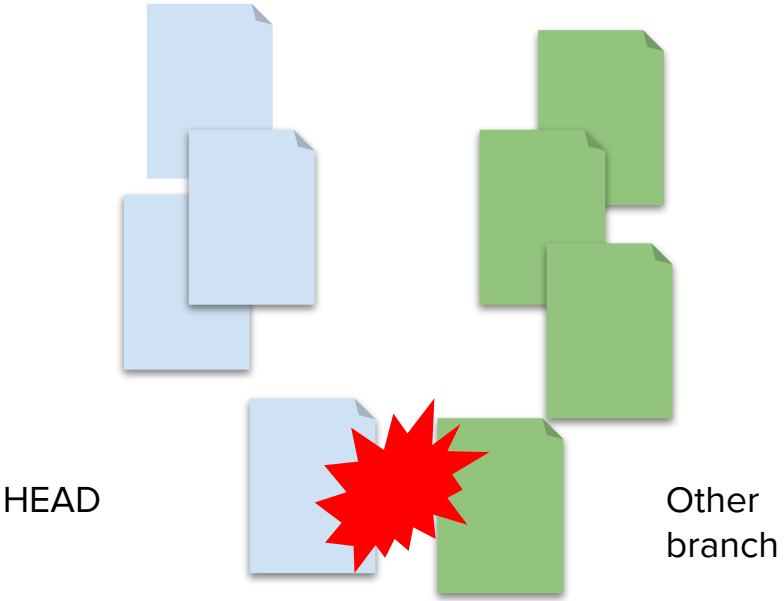
Managing conflicts

Conflicts

When merging branches, changes can be contradictory.

If two branches have changes in the same files, chances are that git cannot merge them automatically.

When this happens, it will ask for help.



Conflicts

You have the last word in conflicts.

In fact, git can manage a lot of changes automatically. But, when the changes happen in the same file, specially in the same lines, it cannot determine which of them are the good ones.

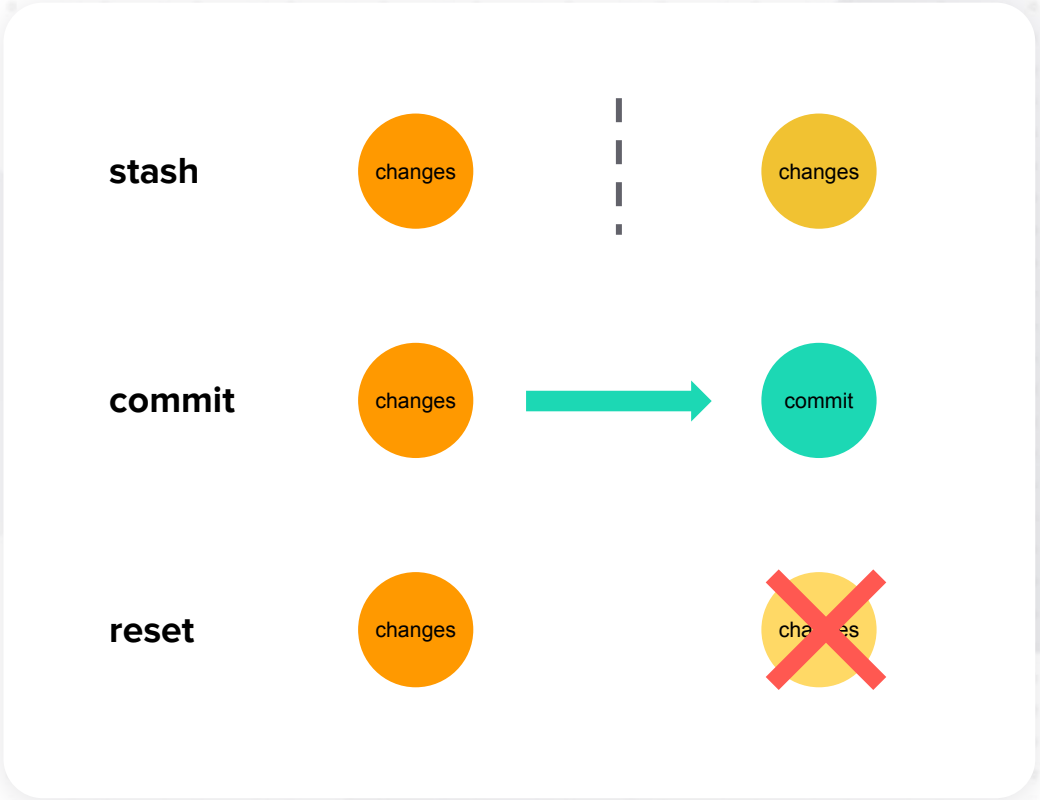
Conflicts before Merge

First conflicts happen when you have pending changes.

Git cannot merge branches if you have changes hanging around. Basically because the merge could overwrite the untracked changes.

So, first of all, you have to prepare for merging:

- **Stash** them to apply later.
- **Commit** to include them in the merge
- **Reset** to remove them.



Prepare to Merge by stashing changes

You can remove the changes for a time, merge and apply the changes after.

The stash list is an space in which you can store sets of changes for some time. This allows you to clean the working directory, but preserve those changes for future use.

Git stash

```
// after making some changes…

// move the changes to the stash list

git stash push --include-untracked

// now, you can perform the merge

git merge other-branch

// re-apply the changes

git stash pop

// if there are conflicts you need to manage them now
```

Prepare to Merge by committing changes

If you think that your current changes are OK you could simply perform a commit.

Once all changes are confirmed with a commit, git can manage most of the conflicts.

```
// after making some changes…

// commit the pending changes

git add .

git commit -m “Pending changes”

// now, you can perform the merge

git merge other-branch

// if there are conflicts you need to manage them now
```

Prepare to Merge by resetting changes

Maybe you have exploratory changes that you simply don't want to use anymore.

Reset the changes, so you are in clean state again.

```
// after making some changes…

// reset the pending changes

git reset

// now, you can perform the merge

git merge other-branch

// if there are conflicts you need to manage them now
```

Git reset

When merge finds conflicts

When you perform the merge, git can find conflicts. It will stop the process and will ask you for action.

Git expects you to review the files, making the needed changes and committing them.
But if you are not ready to do that for whatever reason, you can abort the merge and return to the previous state.

```
git merge the_other_branch

Auto-merging some-file.txt

CONFLICT (content): Merge conflict in
some-file.txt

Automatic merge failed; fix conflicts
and then commit the result.

// Abort merge if you are not ready to
manage this

git merge --abort
```

How a conflict looks like?

Git will show you the differences between the versions of the files.

Git performs a diff comparison between files in the two branches merged. If it cannot reconcile them, it will mark the differences sou you can decide which of them should prevail.

```
Content before the conflict.

<<<<<<< HEAD

this is the content of the file you
have in your working directory

=======

Different content that is in the other
branch that you want to merge

>>>>>>> the_other_branch

Content after the conflict.
```

Resolution of a conflict

Basically, you have to choose which is the right version.

The HEAD section contains the changes that are in your branch (the current HEAD). The other section comes from the branch you want to merge.

Simply delete all what you don’t want, add the changes and commit.

Take into account that several conflicts can happen in the same file.

```
Content before the conflict.

Different content that is in the other
branch that you want to merge

Content after the conflict.


// Add the changes and commit

git add conflicted-file.txt

git commit -m “Conflict fixed”
```

Advice to prevent conflicts

- Merge or rebase frequently your branch with the base or trunk to have the latest changes and take them into account.
- Publish changes frequently and in small batches.
- Small commits are way easier to manage in the case of conflicts.
- The smaller the commits, the less probable is that they can generate a conflict and the easier to manage them.

```
Content before the conflict.

Different content that is in the other branch that you want to merge

Content after the conflict.


// Add the changes and commit

git add conflicted-file.txt

git commit -m “Conflict fixed”
```