Contents

Git introduction:

- Git is the version control system. Version control is software that tracks and manages changes to files over time.

Version control systems generally allow users to revisit earlier versions of the files, compare changes between versions, undo changes, and a whole lot more.

- Git is just one of the many version control systems. Other well-known ones include Subersion, CVS and Mercurial.

- Difference between Git and Github:

+ Git

=> Git is the version control software that runs locally on your machine. You don't need to register for an account.

You don't need the internet to use it. You can use Git without ever touching Github

+ Github

=> Github is a service that hosts Git repositories in the cloud and makes it easier to collaborate with other people.

You do need to sign up for an account to use Github. Its an online place to share work that is done using Git.

- Popular Git GUI 's include:

+ Github Desktop

+ SourceTree

+ Tower

+ GitKraken

+ Ungit

- Configuring Git:

You do not need to register for an account or anything, but you will need to provide:

+ Your name:

git config --global user.name "Your name"

+ Your email:

git config --global user.email Youremail@abc.com

If you are using a GUI, it should prompt you for your name and email the first time you open the app.

- Repository

A Git "Repo" is a workspace which tracks and manages files within a folder

1/ git status

command syntax: git status

gives information on the current status of a git repository and its contents

2/ git init

command syntax: git init

Use git init to create a new git repository

Before we can do anything git-related, we must initialize a repo first

3/ The basic git workflow

Working Directory --- (git add) → Staging Area --- (git commit) → Repository

Work On Stuff (make new files, edit files, delete files, etc)

-> Add Changes (Group specific changes together in preparation of committing)

--> Commit (Commit everything that was previously added)

3a/ git add

command syntax: git add file1 file2

command syntax: git add . (stage all changes at once)

We use the git add command to stage changes to be committed.

It's a way of telling Git, "please include this change in our next commit"

Use git add to add specific files to the staging area. Separate files with spaces to add multiple at once.

use git add . to stage all changes at once

3b/ git commit

command syntax: git commit -m "my message"

Running git commit will commit all staged changes. It also opens up a text editor and prompts you for a commit message

We use the git commit command to actually commit changes from the staging area

When making a commit, we need to provide a commit message that summarizes the changes and work snapshotted in the commit

3c/ Working Directory -> git add -> Staging Area -> git commit -> Repository

4/ Atomic Commits

When possible, a commit should encompass a single feature, change or fix. In other words, try to keep each commit focused on a single thing

This makes it much easier to undo or rollback changes later on. It also makes your code or project easier to review

5/ Amending Commits

Suppose you just made a commit and then realized that you forgot to include a file! Or may be, you made a typo in the commit mesage that you want to correct.

Rather than making a brand new separate commit, you can "redo" the PREVIOUS commit using the --amend option.

(JUST ONE COMMIT AGO - PREVIOUS COMMIT)

Command syntax: git commit -m 'some commit'

git add forgotten_file

git commit --amend

6/ Ignoring files

We can tell Git which files and directories to ignore in a given repository, using a .gitignore file.

This is useful for files you know you never want to commit, including:

+ Secrets, API Keys, Credentials, etc...

+ Operating System Files

+ Log files

+ Dependencies and packages

.gitignore

Create a file called .gitignore in the root of a repository. Inside the file, we can write patterns to tell Git which files and folders to ignore:

+ .DS_Store will ignore files named .DS_Store

+ folderName/ will ignore an entire directory

+ *.log will ignore any files with the .log extension

NOTE: gitignore only ignores files that are not part of the repository yet. If you already git added some files, their changes will still be tracked. To remove those files from your repository (but not from your file system) use git rm --cached on them.

[*https://stackoverflow.com/questions/45400361/why-is-gitignore-not-ignoring-my-files*](https://stackoverflow.com/questions/45400361/why-is-gitignore-not-ignoring-my-files)

With git rm --cached you stage a file for removal, but you don't remove it from the working dir. The file will then be shown as untracked.

Command: git rm –cached fileNameToRemoveFromGitRepository

7/ Branches

Think of branches as alternative timelines for a project

They enable us to create separate contexts where we can try new things, or even work on multiple ideas in parallel.

If we make changes on one branch, they do not impact the other branches (unless we merge the changes)

7a/ The Master branch

In git, we are always working on a branch. The default branch name is master

It doesn't do anything special or have fancy powers. Its just like any other branch

7b/ HEAD

HEAD is simply a pointer that refers to the current "location" in your repository. It points to a particular branch reference.

So far, HEAD always point to the latest commit you made on the master branch, but soon we'll see that we can move around and HEAD will change!

7c/ Viewing Branches

Use git branch to view your existing branches. The default branch in every git repo is master, though you can configure this.

Look for the * which indicates the branch you are currently on.

Command Syntax: git branch

7d/ Creating Branches

Use git branch <branch-name> to make a new branch based upon the current HEAD.

This just creates the branch. It does not switch you to that branch (the HEAD stays the same)

Command Syntax: git branch <branch-name>

7e/ Switching Branches

Once you have created a new branch, use git switch <branch-name> to switch to it

7f/ Another way of switching??

Historically, we used git checkout <branch-name> to switch branches. This still works.

The checkout command does a million additional things, so the decision was made to add a standalone switch command which is much simplier.

You will see older tutorials and docs using checkout rather than switch. Both now work.

Command syntax: git checkout <branch-name>

7g/ Creating and switching

Use git switch with the -c flag to create a new branch AND switch to it all in one go.

Remember -c as short for "create"

Command Syntax: git switch -c <branch-name>

7h/ Switching Branches with unstaged Changes?

When you switch branch, sometimes if you have unstaged changes they will come with you (usually when you create new file that not exist in other branches). Other time, when it is conflict, git will yell at you for you to know that.

7i/ Delete Branches

Go to anywhere else outside from the branch you want to delete.

Command Syntax: git branch -d <branch-name>

Force Delete Branches

Command Syntax: git branch -D <branch-name>

7j/ Renaming Branches

First change to the branch that we want to rename.

Command Syntax: git branch -m NEW_NAME

8/ Merging

Branching makes it super easy to work within self-contained contexts, but often we want to incorporate changes from one branch into another!

We can do this using the git merge command

The merge command can sometimes confuse students early on. Remember these two merging concepts:

+ We merge branches, not specific commits

+ We always merge to the current HEAD branch

8a/ To merge, follow these basic steps:

+ Switch or checkout the branch you want to merge the changes into (the receiving branch)

+ Use the git merge command to merge changes from a specific branch into the current branch

Command syntax: To merge thưe bugfix branch into master:

+ git switch master

+ git merge bugfix

Depending on the specific changes you are trying to merge, Git may not be able to automatically merge.

This results in merge conflicts, which you need to manually resolve.

When you encouter a merge conflict, Git warns you in the console that it could not automatically merge.

It also changes the contents of your files to indicate the conflicts that it wants you to resolve.

Conflict Markers:

The content from your currenty HEAD (the branch you are trying to merge content into) is displayed between the <<<<HEAD and =====

The content from the branch you are trying to merge from is displayed betwwen the ====== and >>>> symbols.

==> **Resolving Conflicts:**

Whenever you encouter merge conflicts, follow these steps to resolve them:

+ Open up the file(s) with merge conflicts

+ Edit the file(s) to remove the conflicts. Decide which branch's content you want to keep in each conflict. Or keep the content from both.

+ Remove the conflict "markers" in the document

+ Add your changes and then make a commit.

9/ Git Diff

9a/ git diff

We can use git diff command to view changes between commits, branches, files, our working directory and more!

We often use git dif alongside commands like git status and git log, to get a better picture of a repository and how it has changed over time.

Command Syntax: git diff

(Compares staging area and working directory)

Without additional options, git diff lists all the changes in our working directory that are NOT staged for the next commit.

9b/ git diff HEAD

git diff HEAD lists all changes in the working tree since your last commit.

Command Syntax: git diff HEAD

9c/ git diff --staged

git diff --cached

git diff --stage or --cached will list the changes between the staging area and our last commit.

"Show me what will be included in my commit if i run git commit right now"

Command Syntax:

+ git diff --staged

+ git diff --cached

9d/ Diff-ing Specific Files

We can view the changes within a specific file by providing git diff with a filename

Command Syntax:

+ git diff HEAD [filename]

+ git diff --staged [filename]

9e/ Comparing Branches

git diff branch1..branch2 will list the changes between the tips of branch1 and branch2

Command Syntax: git diff branch1..branch2

9f/ Comparing commits

To compare two commits, provide git diff with the commit hashes of the commits in question.

Command Syntax: git diff commit1..commit2

10/ Stashing

When I have some stuffs of work (not add to stage area) in a branch, and I want to switch back immediately to destination branch.

So there are 2 scenarios:

+ My changes come with me to the destination branch.

+ Git won’t let me switch if it detects potential conflicts. => Use git stash

Git provides an easy way of stashing these uncommitted changes so that we can return to them later, without having to make unnecessary commits.

10a/ git stash save

git stash is super useful command that helps you save changes that you are not yet ready to commit. You can stash changes and then come back to them later.

Running git stash will take all uncommitted changes (**staged and unstaged**) and stash them, reverting the changes in your working copy.

Command syntax: git stash

You can also use **git stash save** instead

Now that I have stashed my changes, I can switch branches, create new commits, etc,.

I head over to master and take a loot at any other changes.

When I'm done, I can re-apply the changes that I stashed away at any point. (git stash pop)

10b/ git stash pop

Use git stash pop to remove the most recently stashed changes in your stash and re-apply them to your working copy.

Command Syntax: git stash pop

10c/ git stash apply

You can use git stash apply to apply whatever is stashed away, without removing it from the stash.

This can be useful if you want to apply stashed changes to multiple branches.

Command Syntax: git stash apply

10d/ Stashing Multiple times

You can add multiple stashes onto the stack of stashes. They will all be stashed in the order you added them.

Command Syntax:

+ git stash

    do some other stuff

+ git stash

    do some other stuff

+ git stash

10e/ View stashes

run git stash list to view all stashes

Command Syntax: git stash list

10f/ Applying specific stashes

git assumes you want to apply the most recent stash when you run git stash apply, but you can also specify a particular stash like git stash apply stash@{2}

Command Syntax: git stash apply stash@{2}

10j/ Dropping stashes

To delete a particular stash, you can use git stash drop <stash-id>

Command Syntax: git stash drop stash@{2}

10h/ Clearing the stash

To clear out all stashes, run git stash clear

Command Syntax: git stash clear

11/ Undoing Changes and Time Traveling

11a/ Checkout

Many developers think it is overloaded, which is what lead to the addition of the git switch and git store command.

We can use checkout to create branches, switch to new branches, restore files, and undo history!

We can use **git checkout <commit-hash>** to view a previous commit

Remember, you can use the git log command to view commit hashes. We just need the first 7 digits of a commit hash.

Command Syntax: **git checkout d8194d6**

***

Detached HEAD

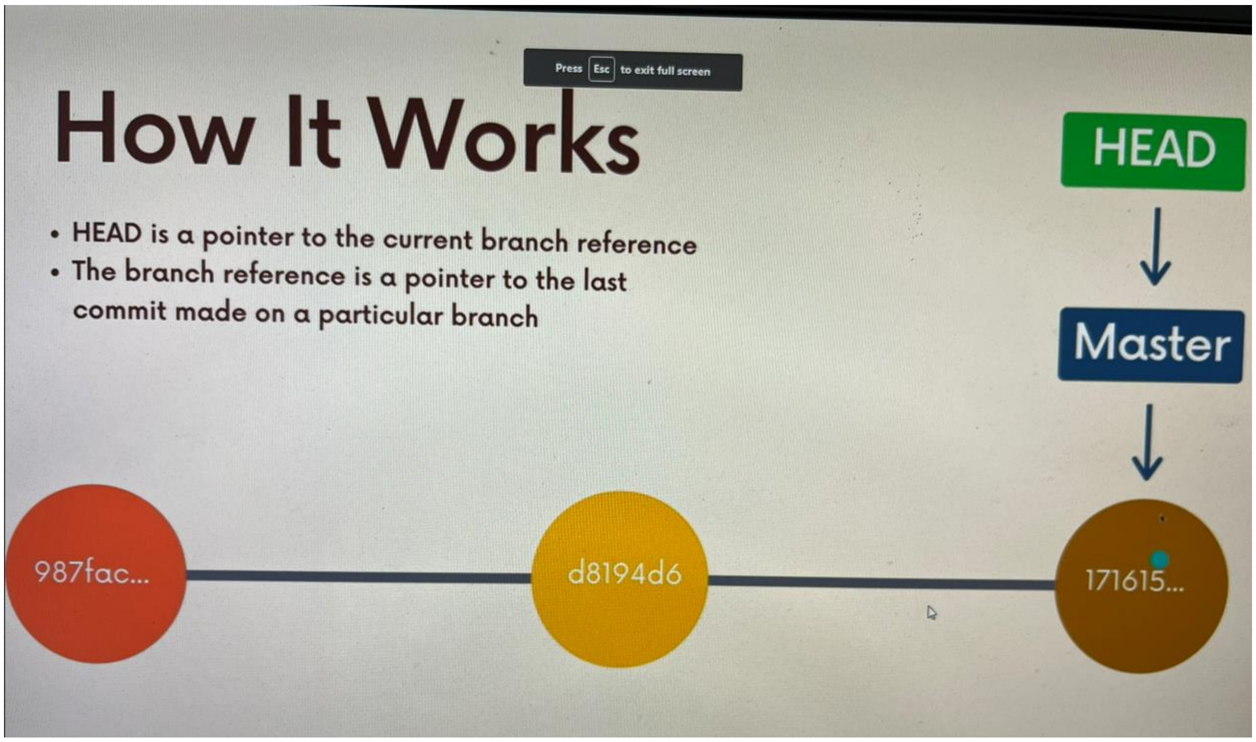
You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by switching back to a branch.

You have a couple options:

a/ Stay in detached HEAD to examine the contants of the old commit. Poke around, view the files, etc.

b/ Leave and go back to wherever you were before - reattach the HEAD.

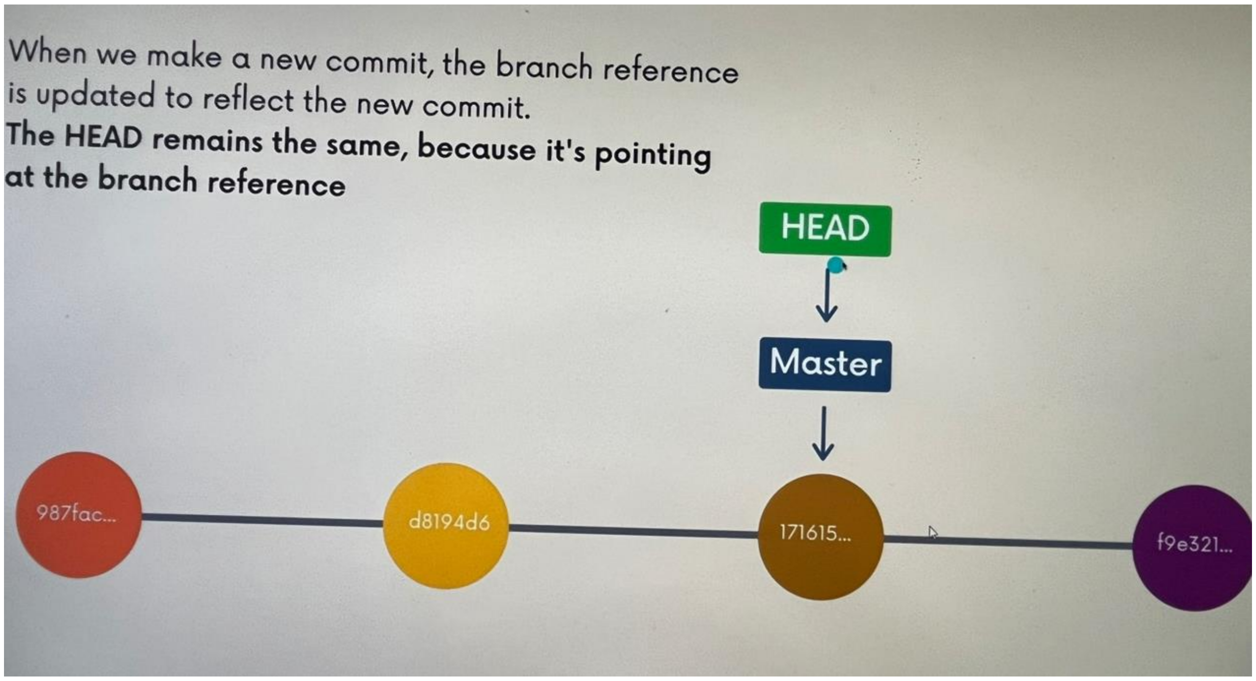
c/ Create a new branch and switch to it. You can now make and save changes, since HEAD is no longer detached.



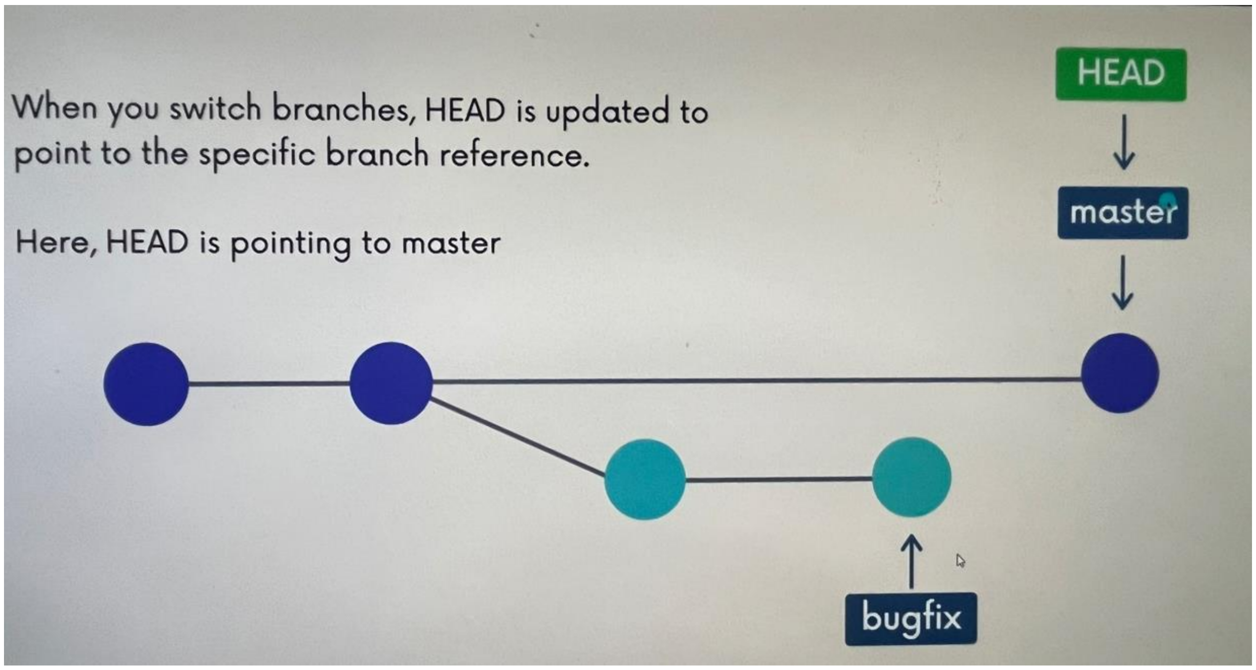
Press Esc to exit full screen

How It Works

- HEAD is a pointer to the current branch reference
- The branch reference is a pointer to the last commit made on a particular branch

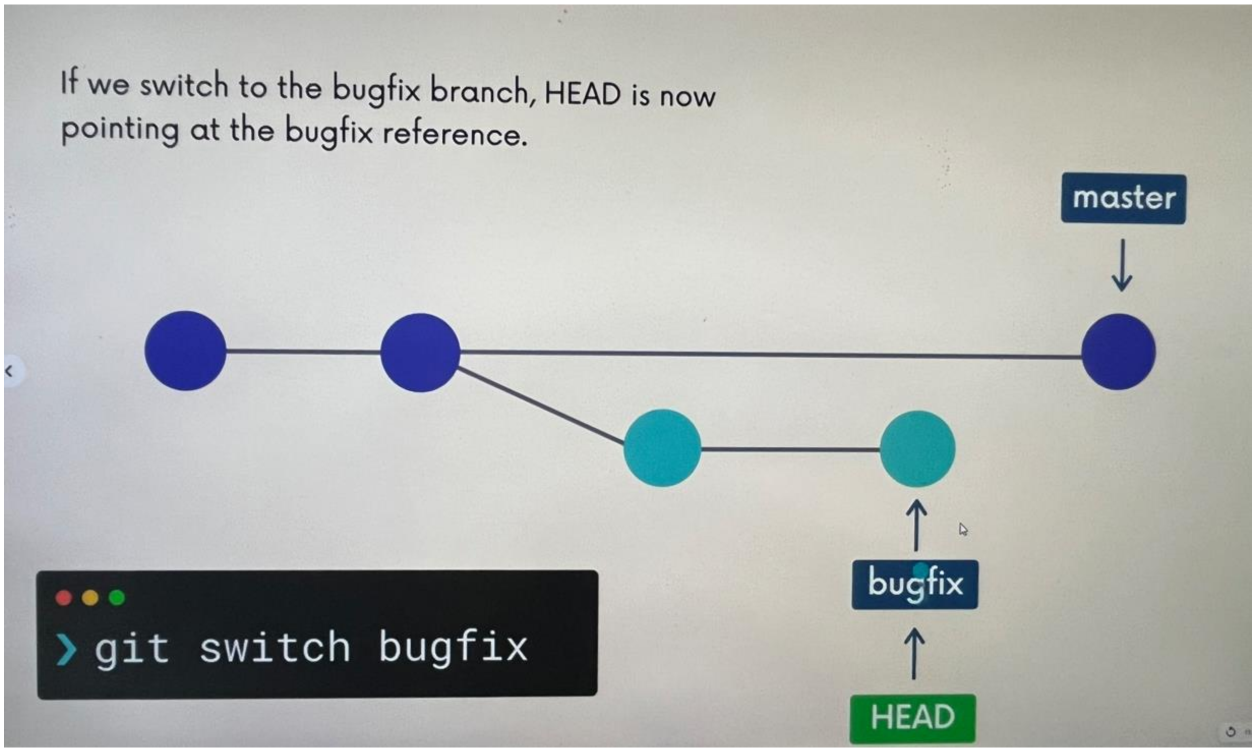


When we make a new commit, the branch reference is updated to reflect the new commit.
The HEAD remains the same, because it's pointing at the branch reference



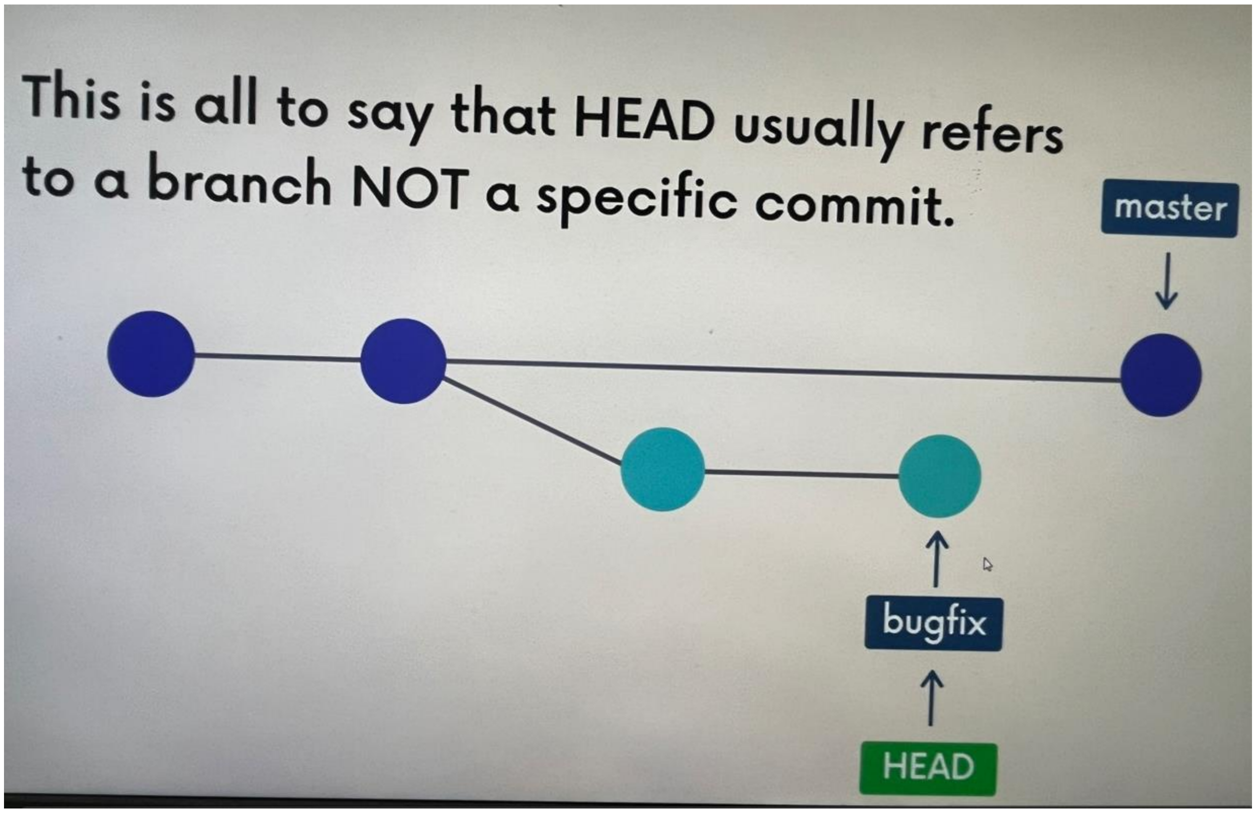
When you switch branches, HEAD is updated to point to the specific branch reference.

Here, HEAD is pointing to master

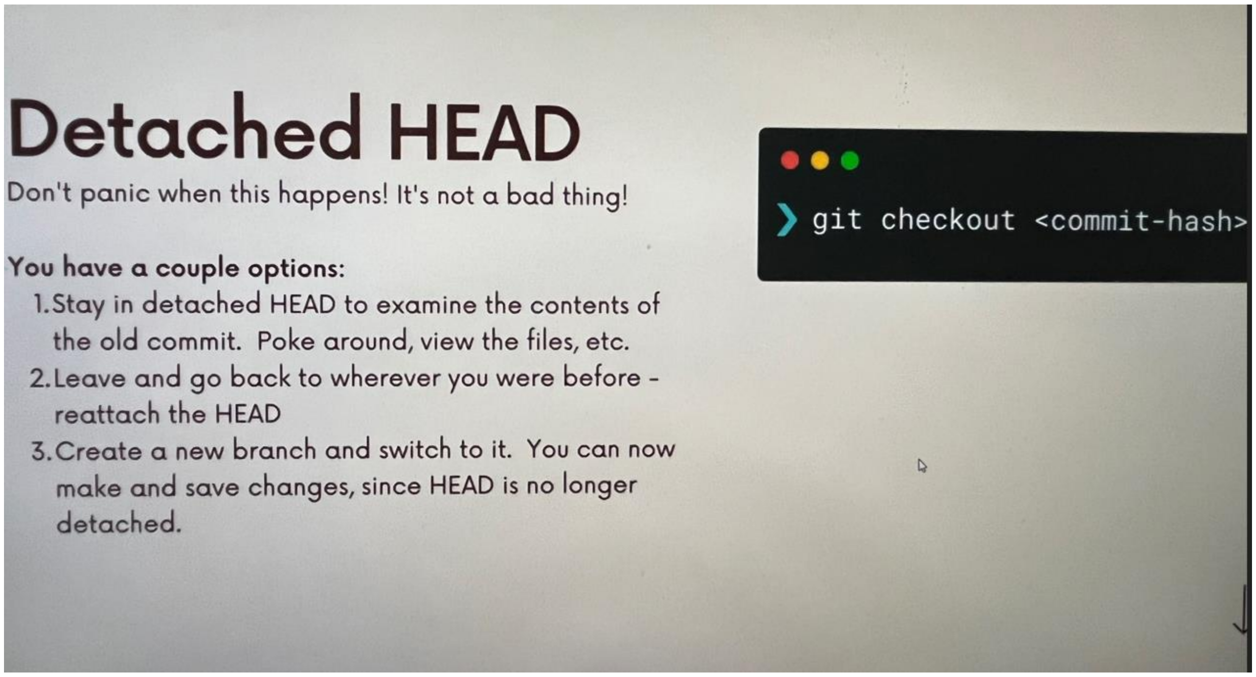


If we switch to the bugfix branch, HEAD is now pointing at the bugfix reference.

```
❯ git switch bugfix
```



This is all to say that HEAD usually refers to a branch NOT a specific commit.



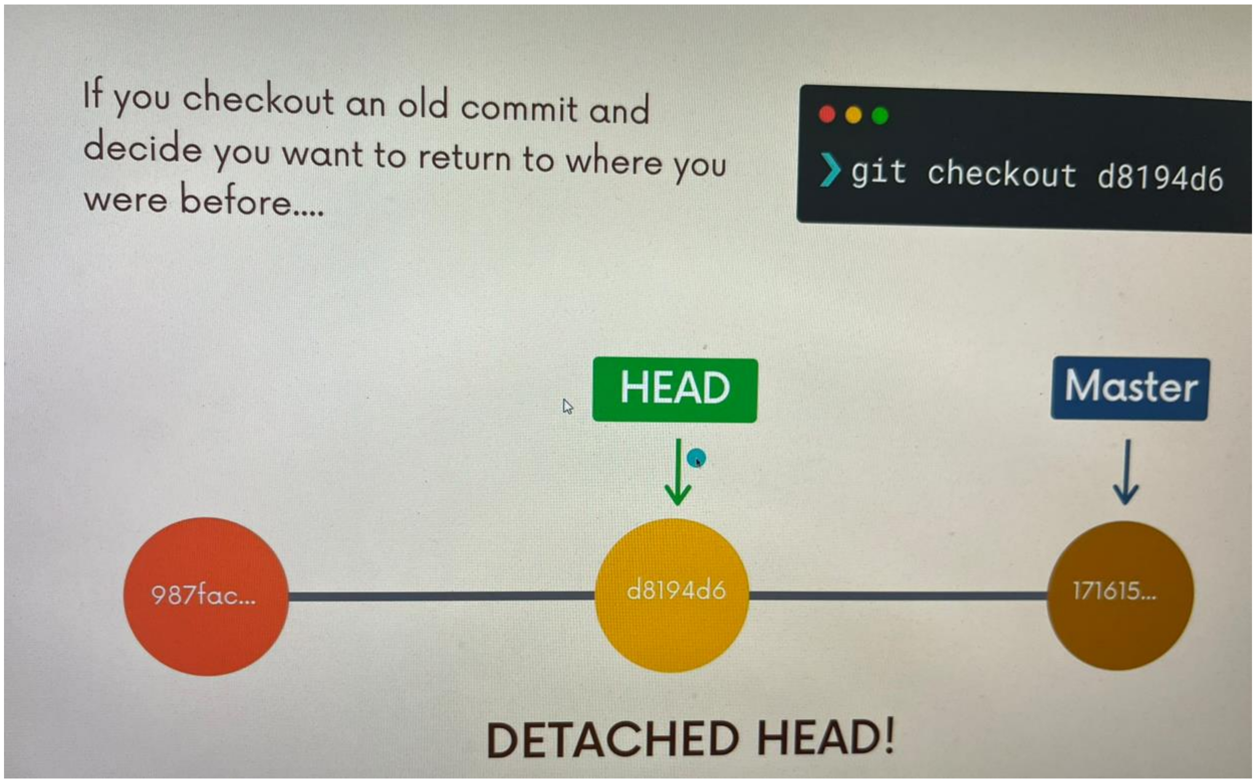
Detached HEAD

Don't panic when this happens! It's not a bad thing!

You have a couple options:

1. Stay in detached HEAD to examine the contents of the old commit. Poke around, view the files, etc.
2. Leave and go back to wherever you were before - reattach the HEAD
3. Create a new branch and switch to it. You can now make and save changes, since HEAD is no longer detached.

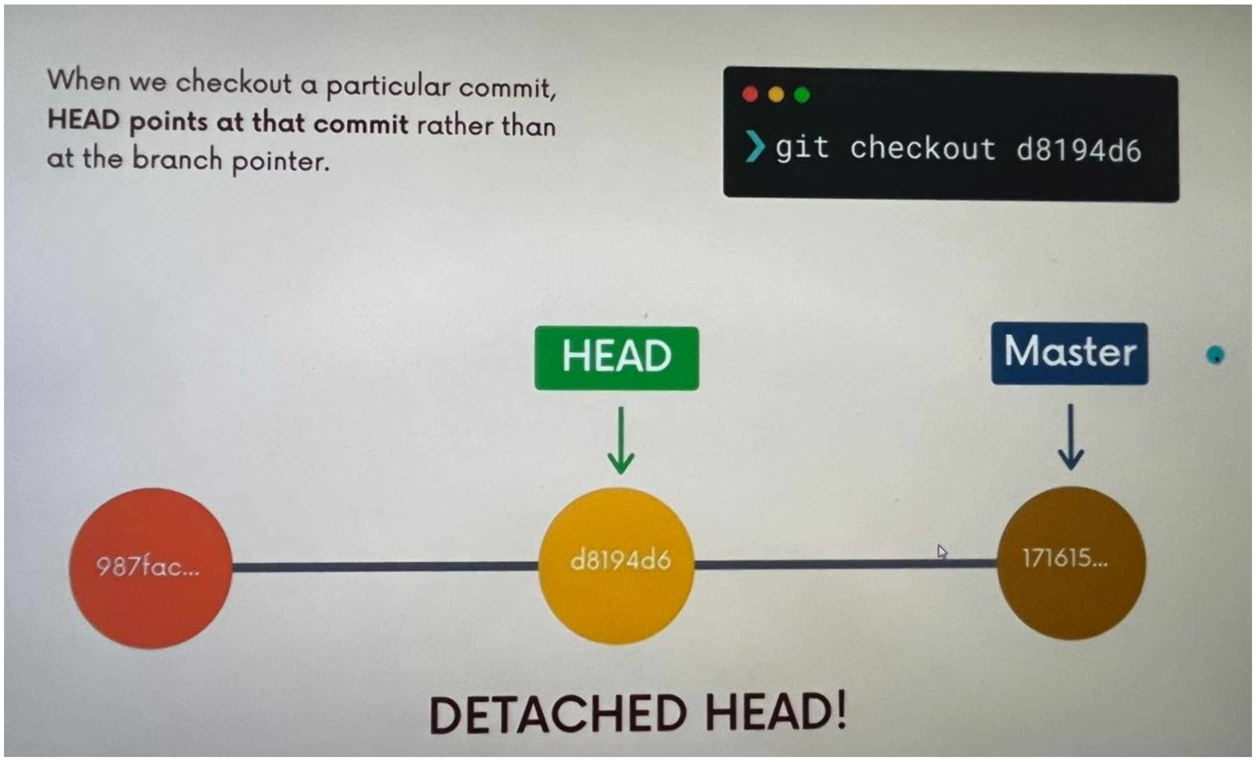
```
❯ git checkout <commit-hash>
```



If you checkout an old commit and decide you want to return to where you were before....

```
❯ git checkout d8194d6
```

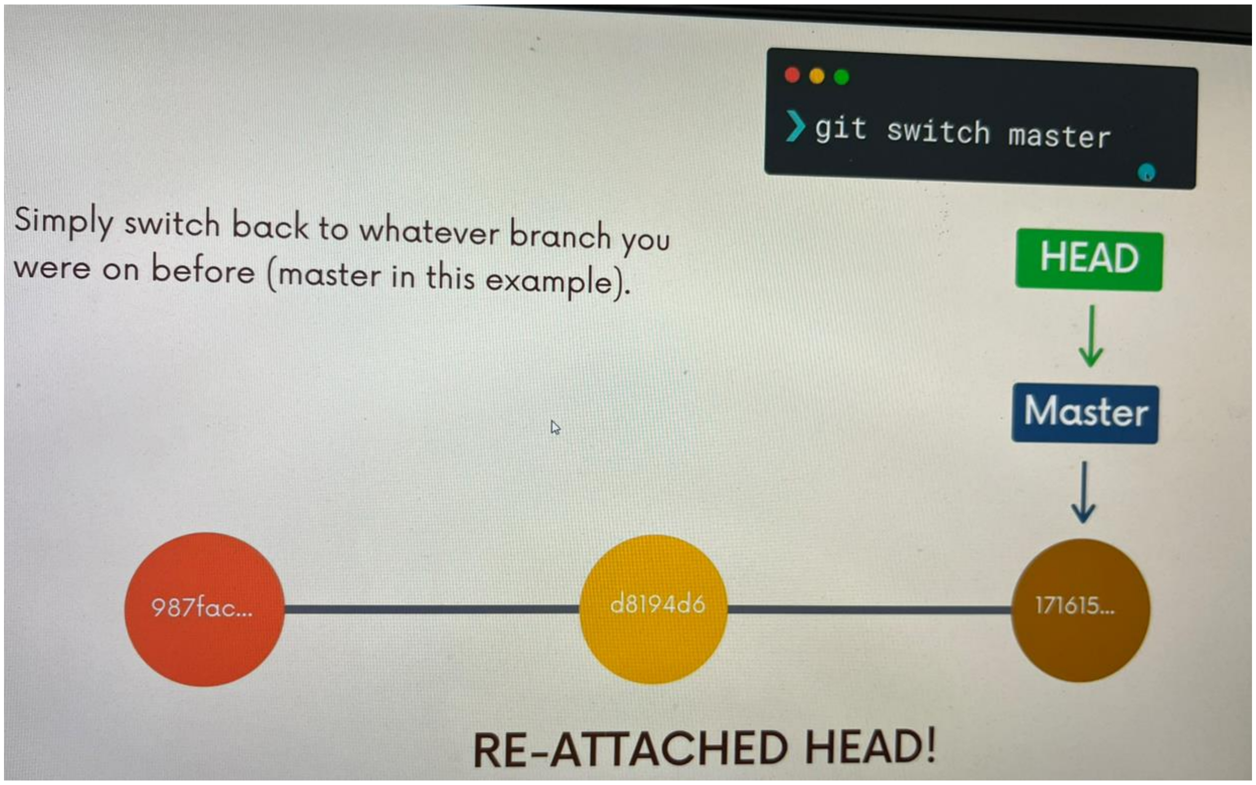
DETACHED HEAD!



When we checkout a particular commit, **HEAD points at that commit** rather than at the branch pointer.

```
❯ git checkout d8194d6
```

DETACHED HEAD!



Simply switch back to whatever branch you were on before (master in this example).

```
> git switch master
```

***

11b/ Checkout

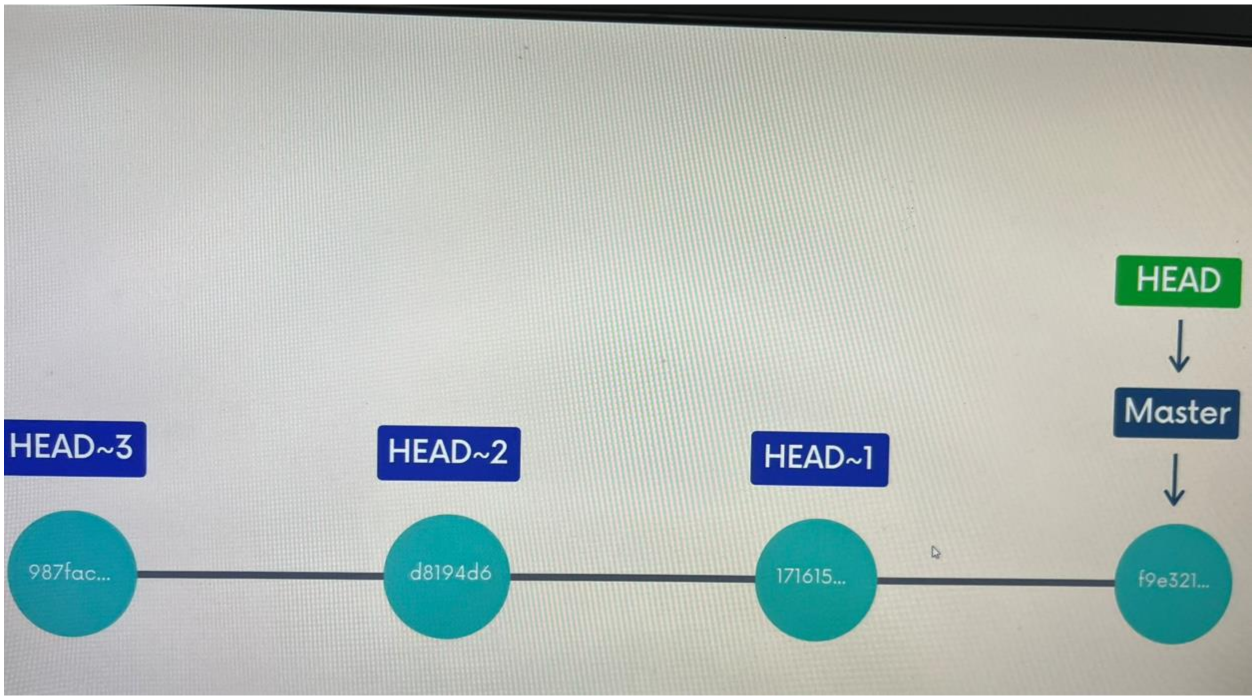
git checkout supports a slightly odd syntax for referencing previous commits relative to a particular commit.

HEAD~1 refers to the commit before HEAD (parent)

HEAD~2 refers to 2 commits before HEAD (grandparent)

This is not essential, but i want to mention it because it's quite weird looking if you've never seen it

Command Syntax: git checkout HEAD~1



11c/ Discarding changes

Suppose you've made some changes to a file but do not want to keep them. To revert the file back to whatever it looked like when you last committed, you can use:

git checkout HEAD <filename> to discard any changes in that file, reverting back to the HEAD.

Command Syntax: git checkout HEAD <file>

11d/ Another Option

Here's another shorter option to revert a file..

Rather than typing HEAD, you can substitute -- followed by the file(s) you want to restore

Command syntax: git checkout -- <file>

11e/ Restore

git restore is a brand new Git command that helps with undoing operations.

Because it is so new, most of the existing Git tutorials and books do not mention it, but it is worth knowing!

Recall that git checkout does a million different things, which many git users find very confusing.

git restore was introduced alongside git switch as alternatives to some of the uses for checkout.

11e1/ Unmodifying files with restore

Suppose you've made some changes to a file since your last commit. You've saved the file but then realize you definitely do NOT want those changes anymore!

To restore the file to the contents in the HEAD, use git restore <file-name>

Command Syntax: git restore <file-name>

NOTE: The above command is not "undoable" if you have uncommited changes in the file, they will be lost!

11e2/ Unmodifying Files with Restore

git restore <file-name> restores using HEAD as the default source, but we can change that using the --source option.

For example, git restore --source HEAD~1 home.html will restore the contents of home.html to its state from the commit prior to HEAD. You can also use a particular commit hash as the source

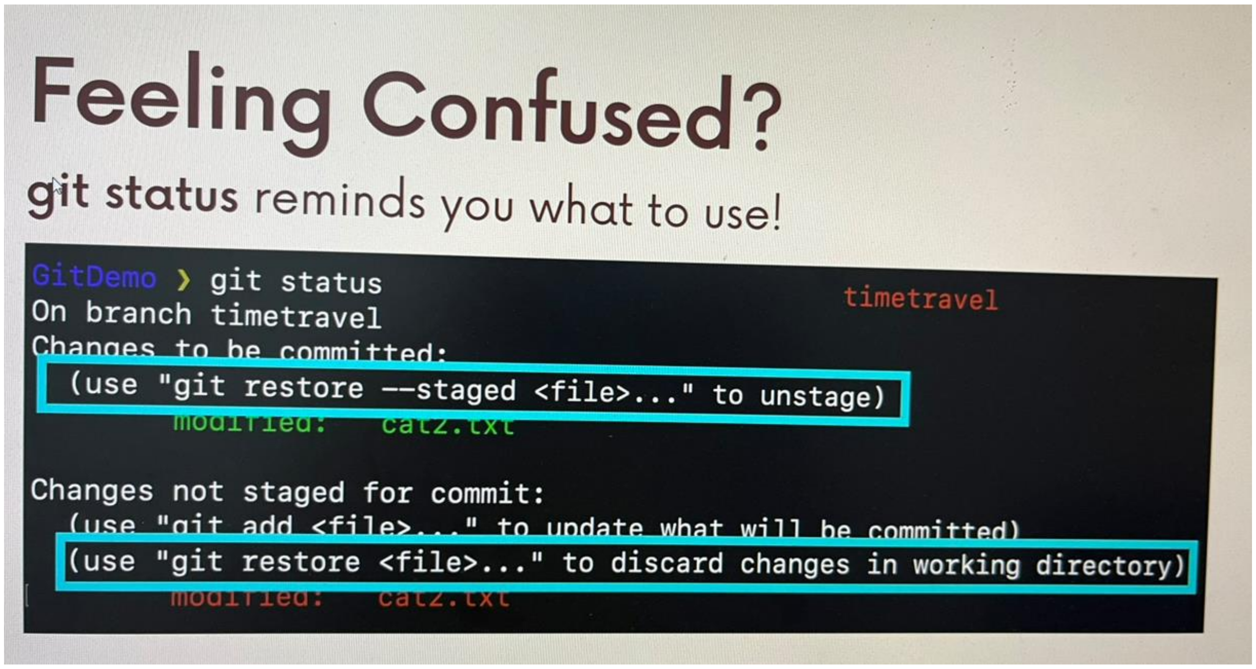
Command Syntax: git restore --source HEAD~1 app.js

11e3/ Unstaging Files with Restore

If you have accidentally added a file to your staging area with git add and you do not wish to include it in the next commit, you can use git restore to remove it from staging.

Use the --staged option like this: git restore --staged app.js

Command Syntax: git restore --staged <file-name>



Feeling Confused?

git status reminds you what to use!

```
GitDemo ❯ git status                                    timetravel
On branch timetravel
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   cat2.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   cat2.txt
```

12/ Undoing commits with git reset

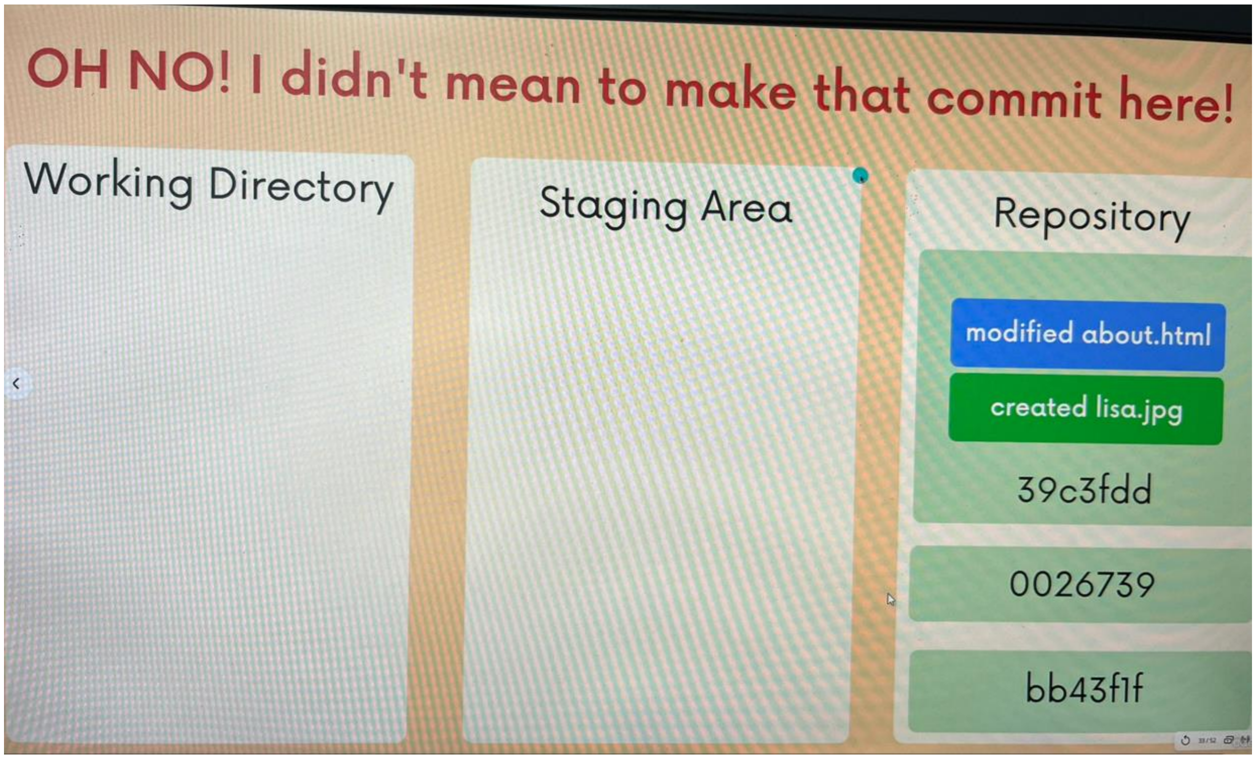
12a/ Git Reset

Suppose you've just made a couple of commits on the master branch, but you actually meant to make them on a separate branch instead.

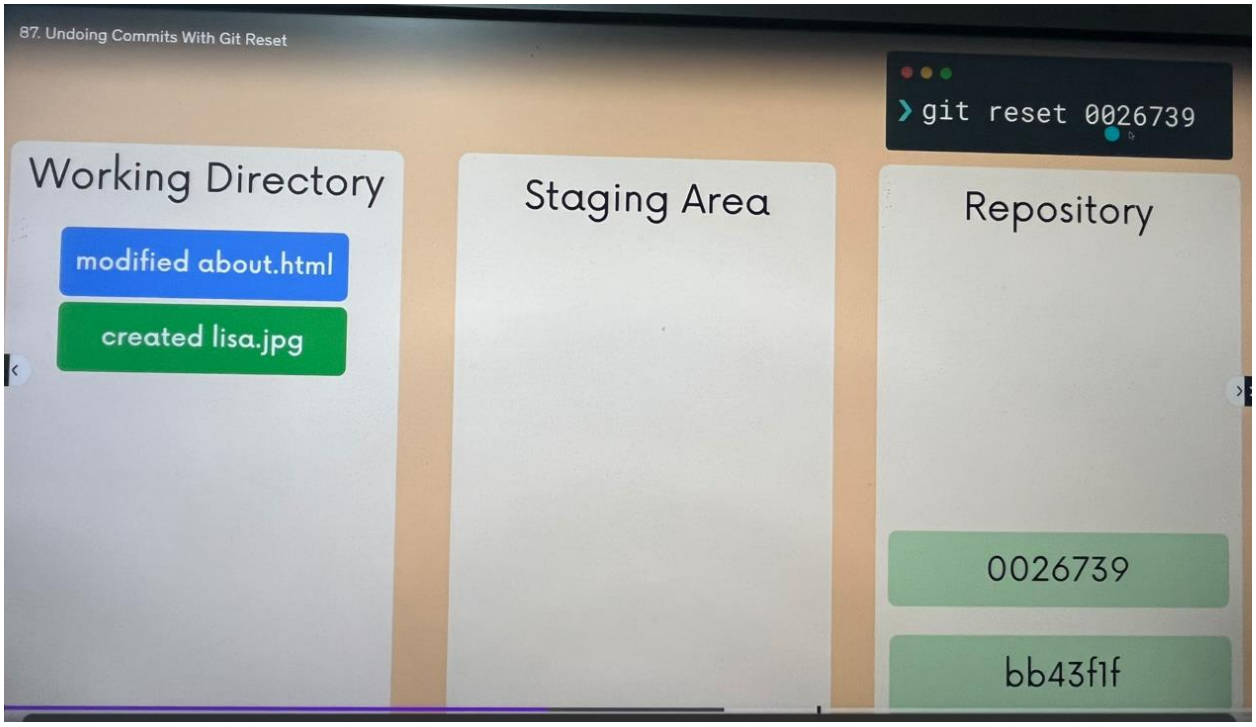
To undo those commits, you can use git reset.

git reset <commit-hash> will reset the repo back to a specific commit. The commits are gone.

Command syntax: git reset <commit-hash>



OH NO! I didn't mean to make that commit here!



87. Undoing Commits With Git Reset

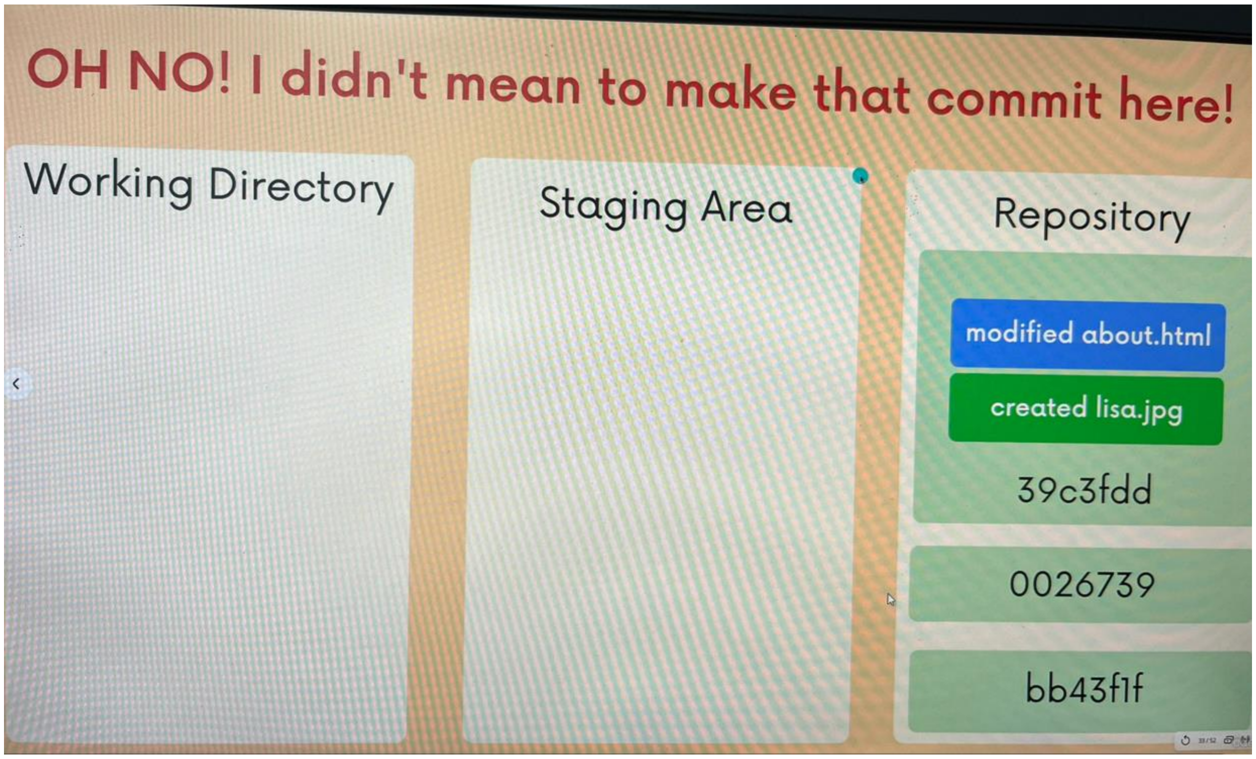
```
git reset 0026739
```

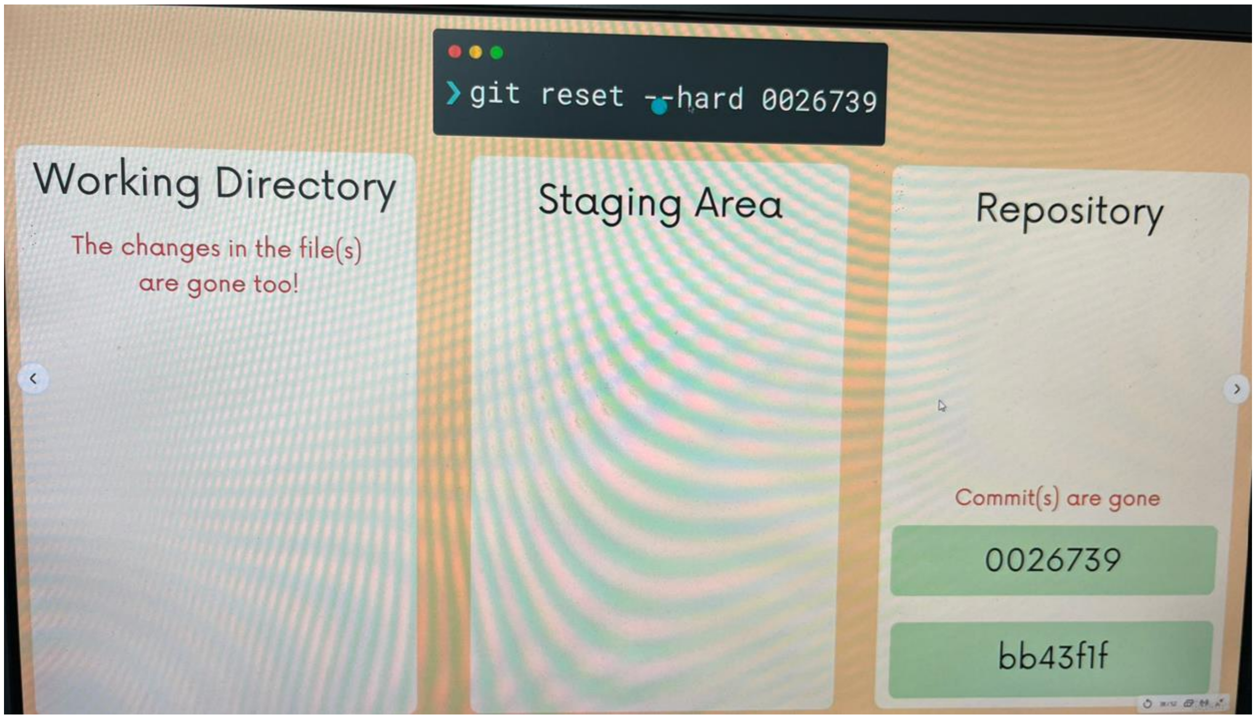
12b/ Reset --hard

If you want to undo both the commits AND the actual changes in your files, you can use the --hard option.

for example, git reset --hard HEAD~1 will delete the last commit and associated changes.

Command syntax: git reset --hard <commit-hash>





```
❯ git reset --hard 0026739
```

13/ Reverting Commits with Git Revert

13a/ Git Revert

git revert is similar to git reset in that they both "undo" changes, but they accomplish it in different ways.

git reset actually moves the branch pointer backwards, eliminating commits.

git revert instead creates a brand new commit which reverses/undo the changes from a commit. Because it results in a new commit, you will be prompted to enter a commit message.

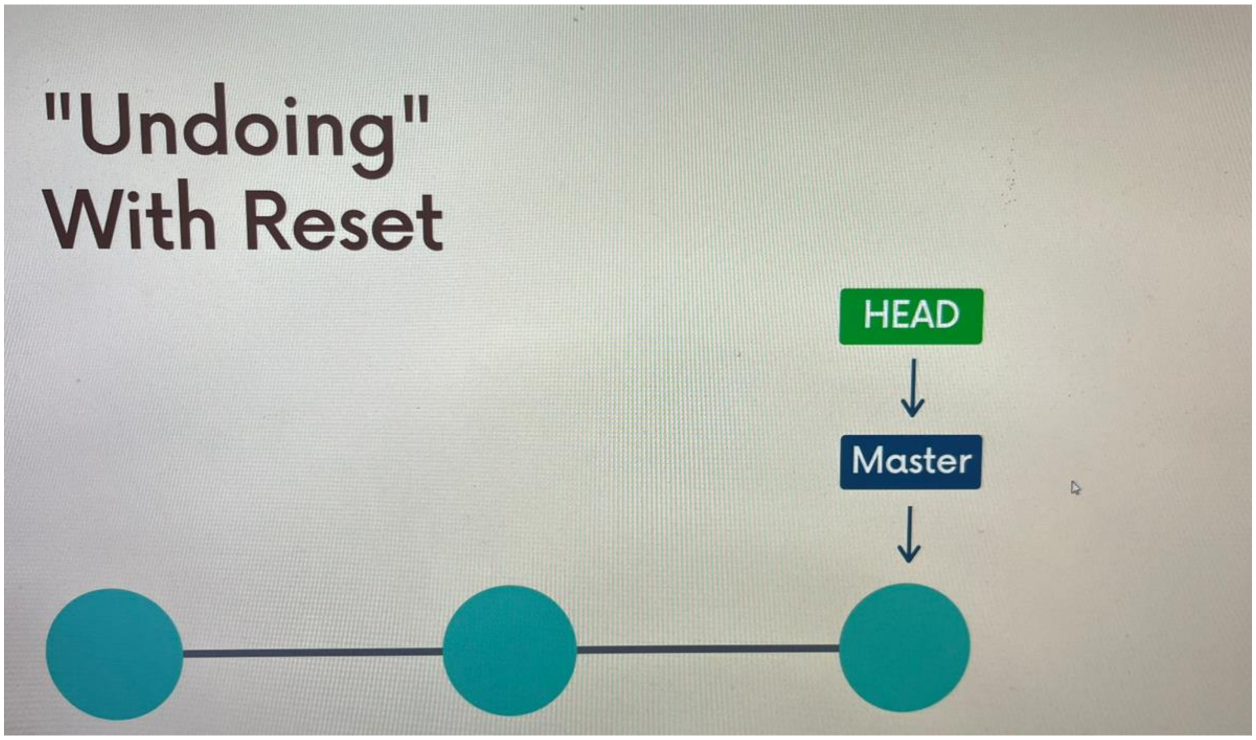
Command syntax: git revert <commit-hash>

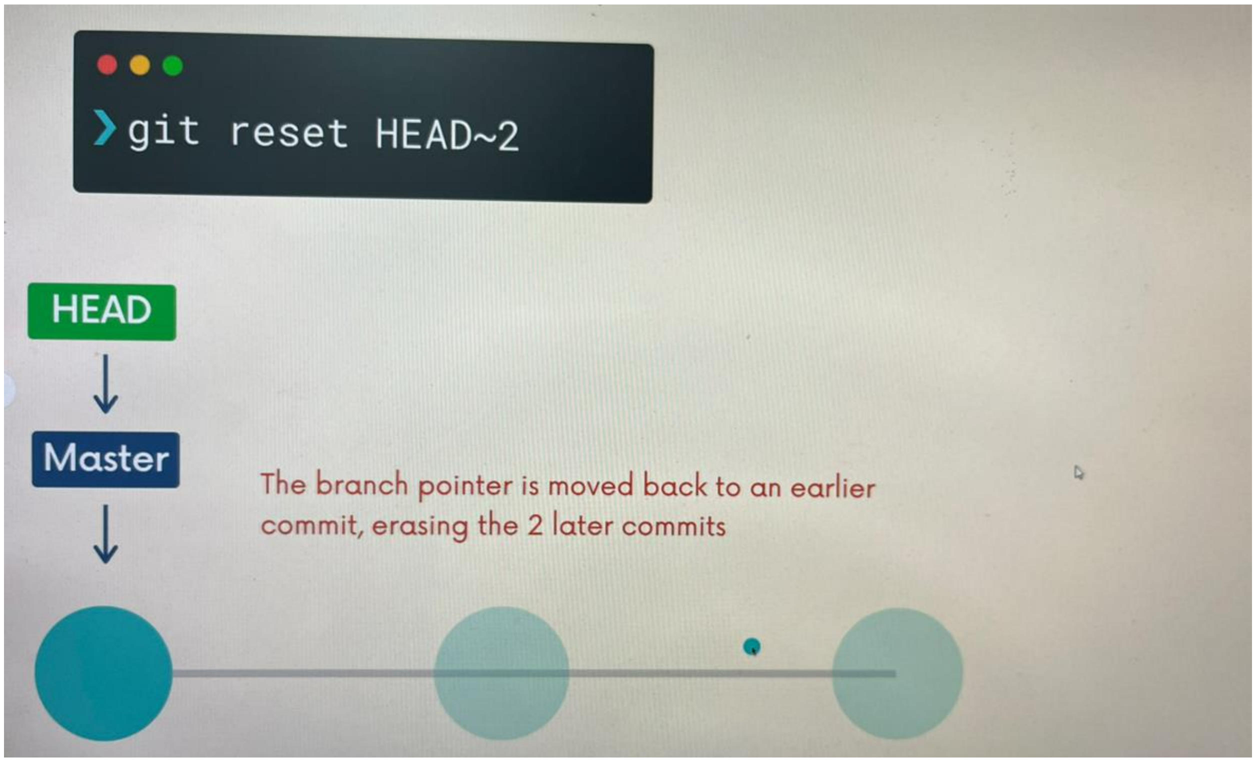
Note: Which one should i use?

Both git reset and git revert help us reverse changes, but there is a significant difference when it comes to collaboration.

+ If you want to reverse some commits that other people already have on their machine, you should you revert.

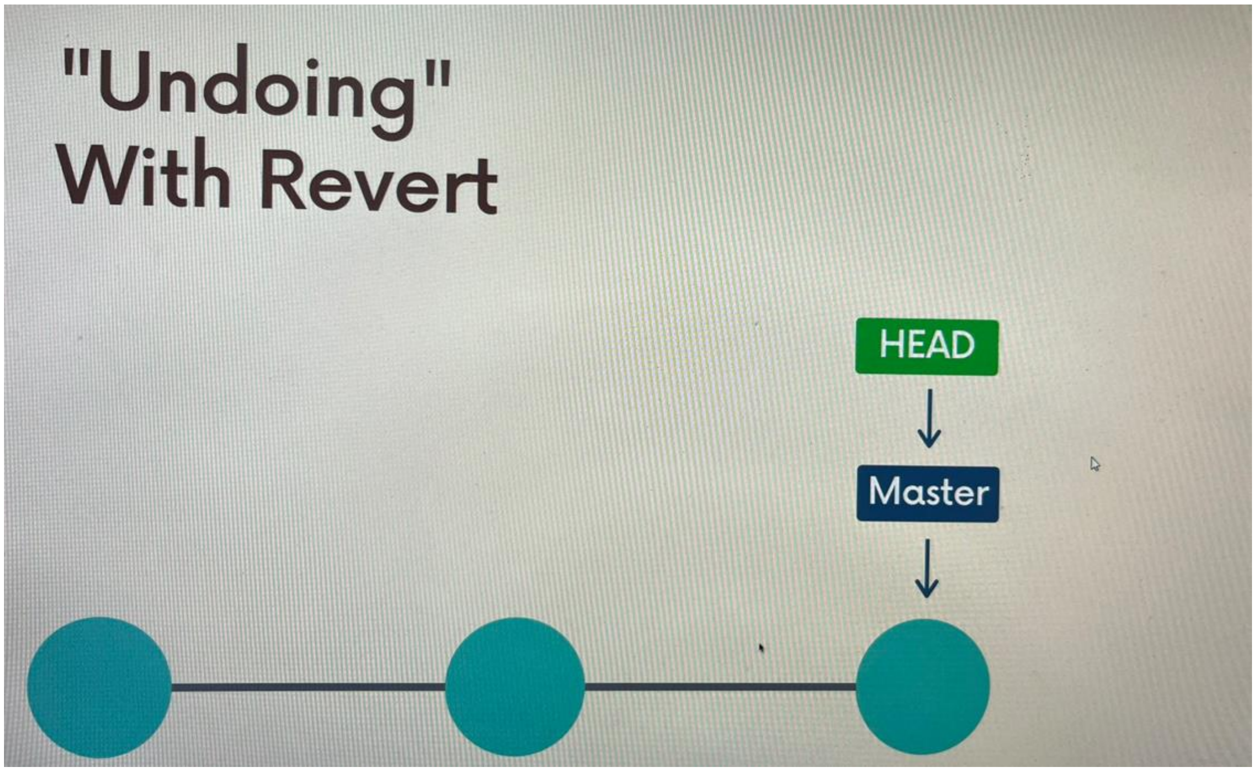
+ If you want to reverse commits that you haven't shared with others, use reset and no one will ever know!



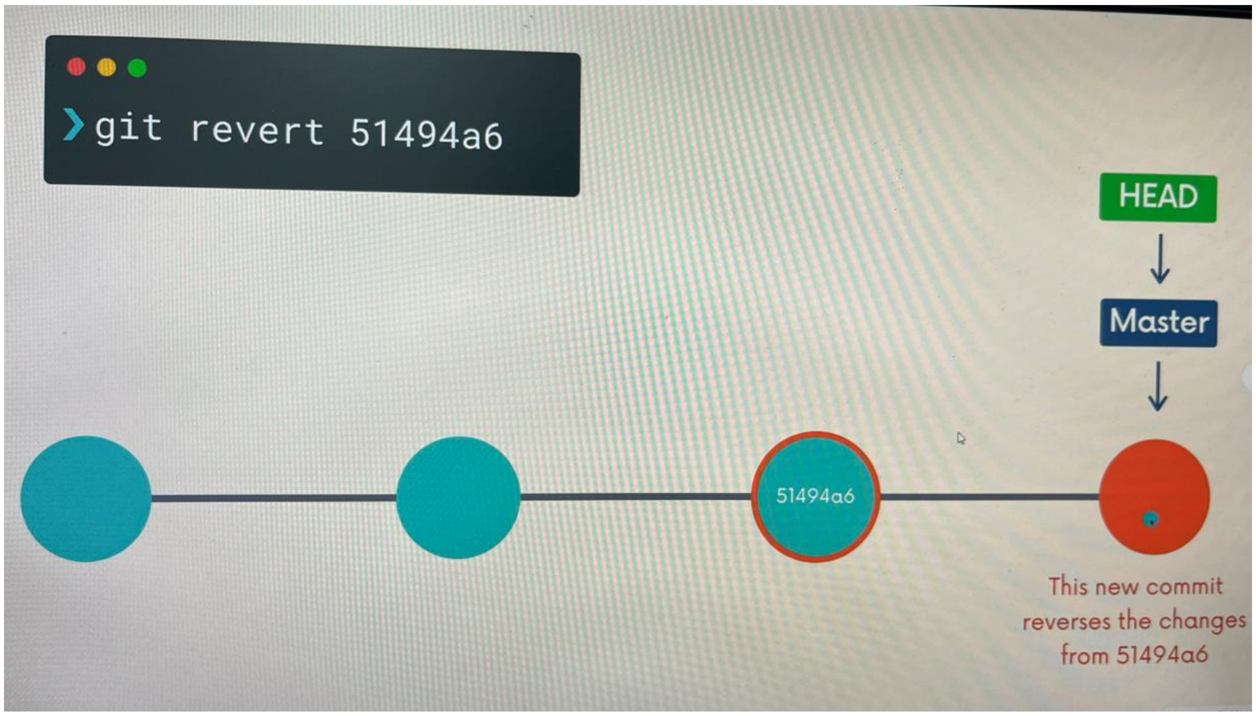


```
❯git reset HEAD~2
```

The branch pointer is moved back to an earlier commit, erasing the 2 later commits



"Undoing" With Revert



```
❯git revert 51494a6
```

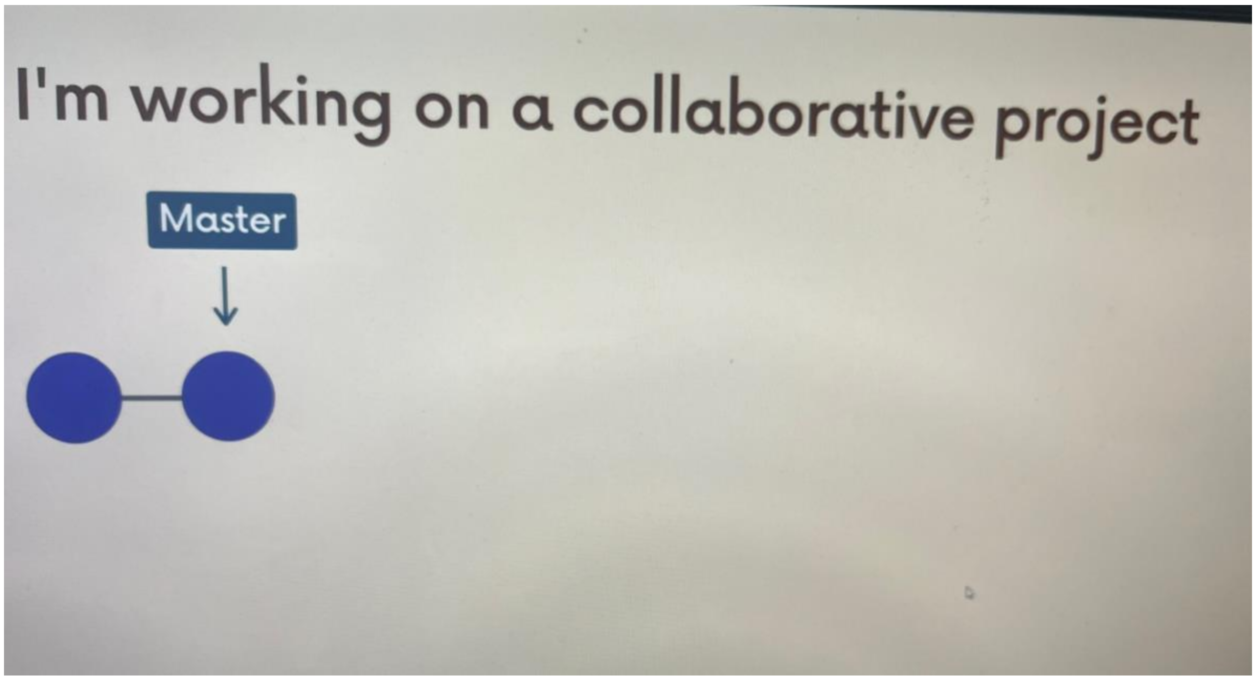
14/ Rebase

14a/ Rebasing

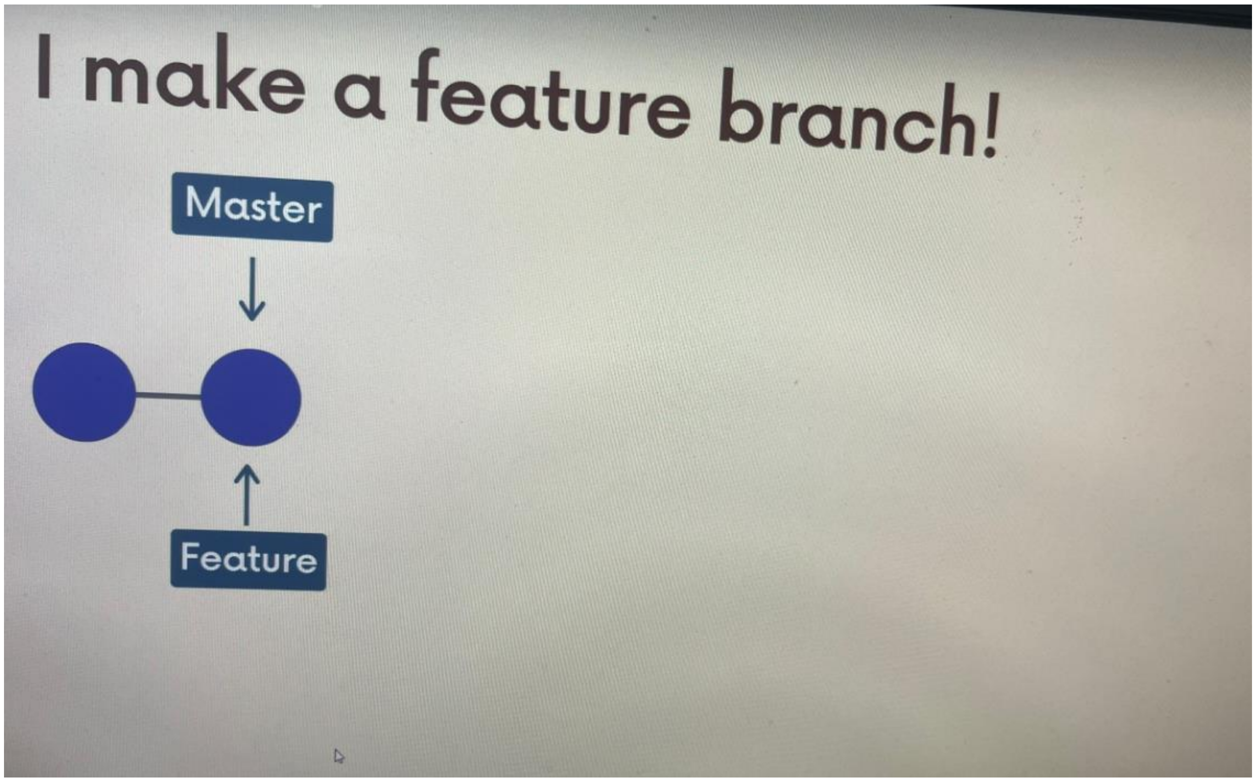
There are two main ways to use the git rebase command:

+ as alternative to merging

+ as a cleanup tool

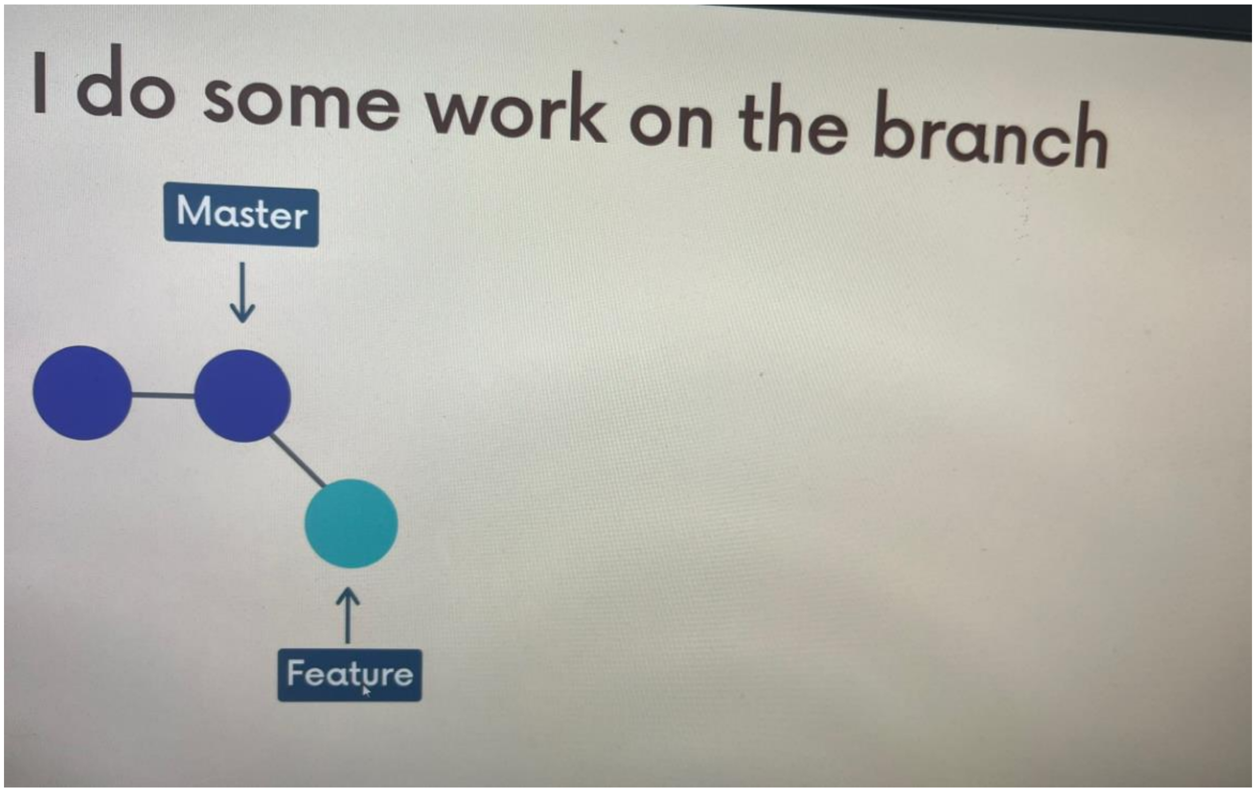


I'm working on a collaborative project

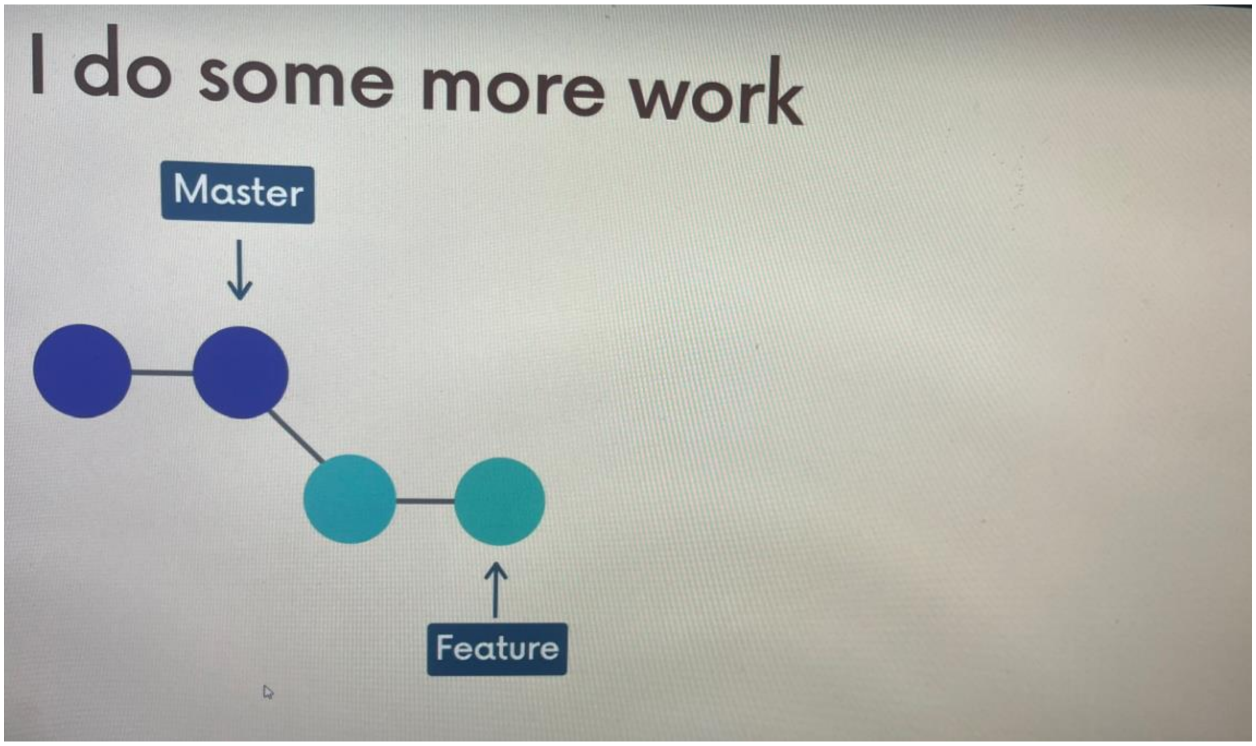


I make a feature branch!

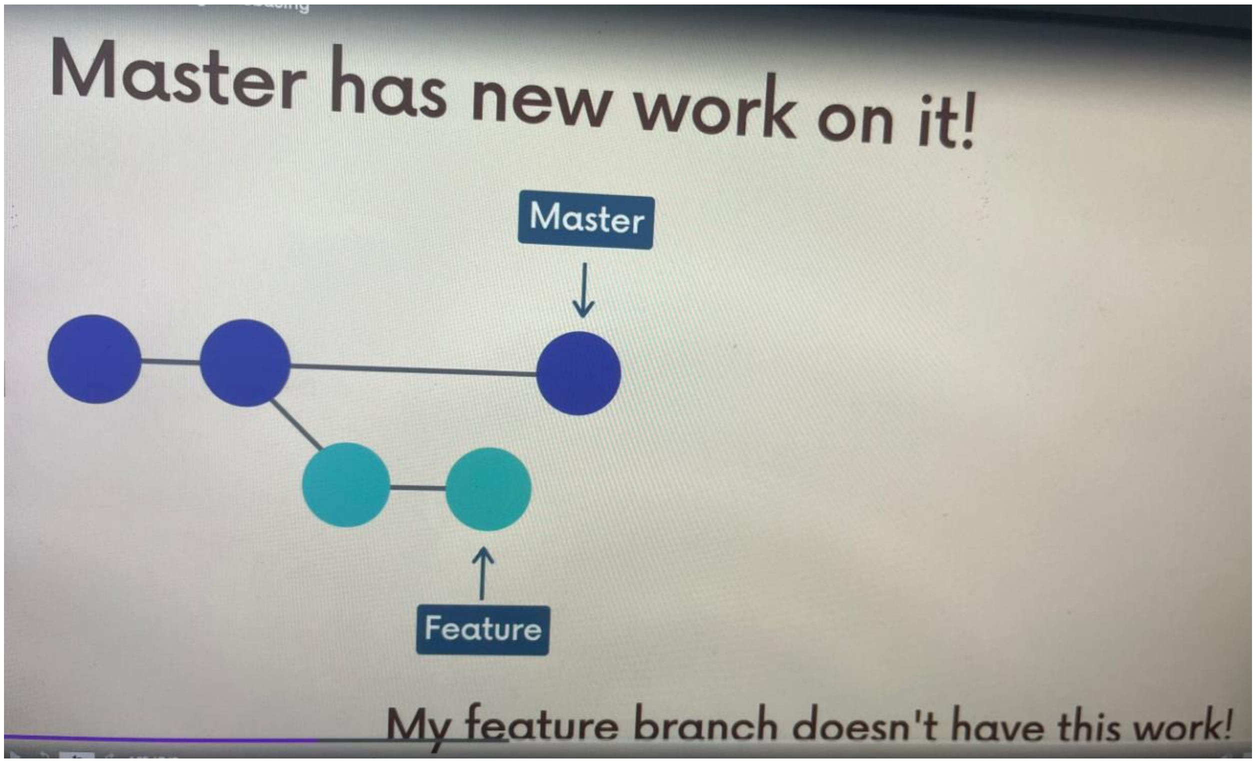
Master
Feature



I do some work on the branch

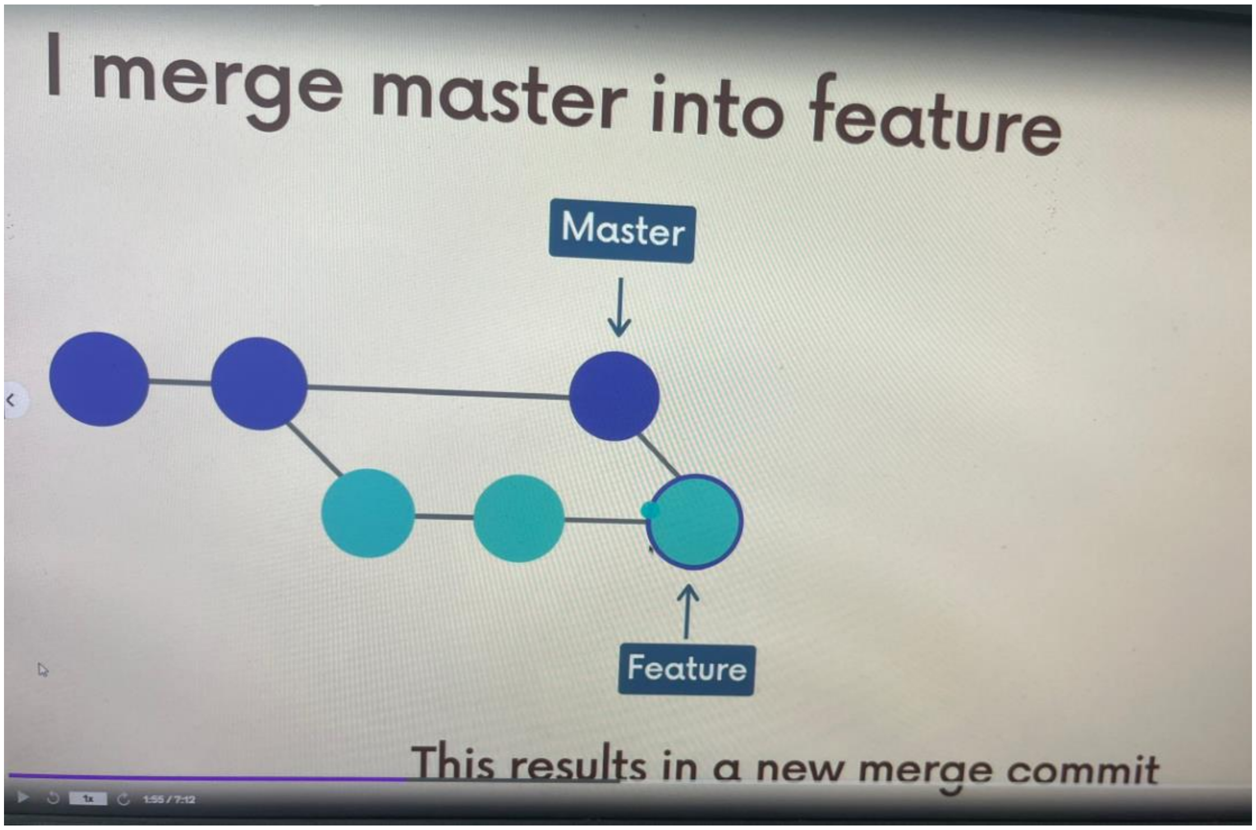


I do some more work



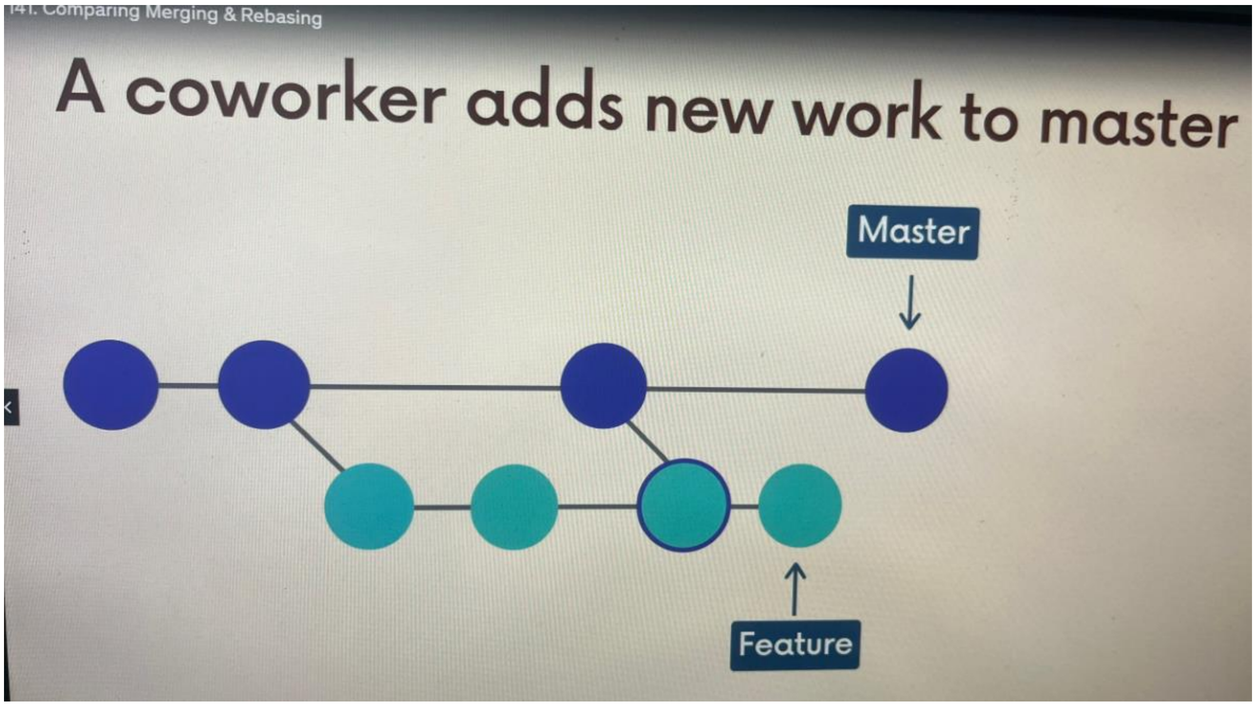
Master has new work on it!

My feature branch doesn't have this work!

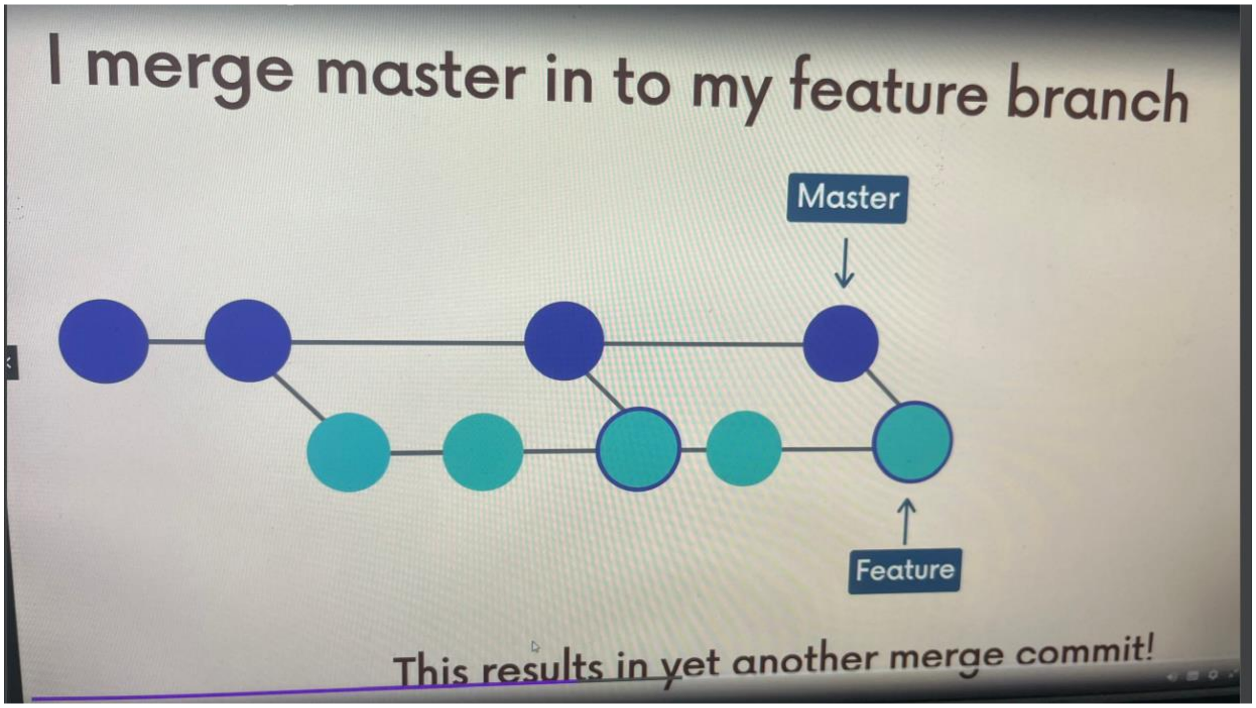


I merge master into feature

This results in a new merge commit

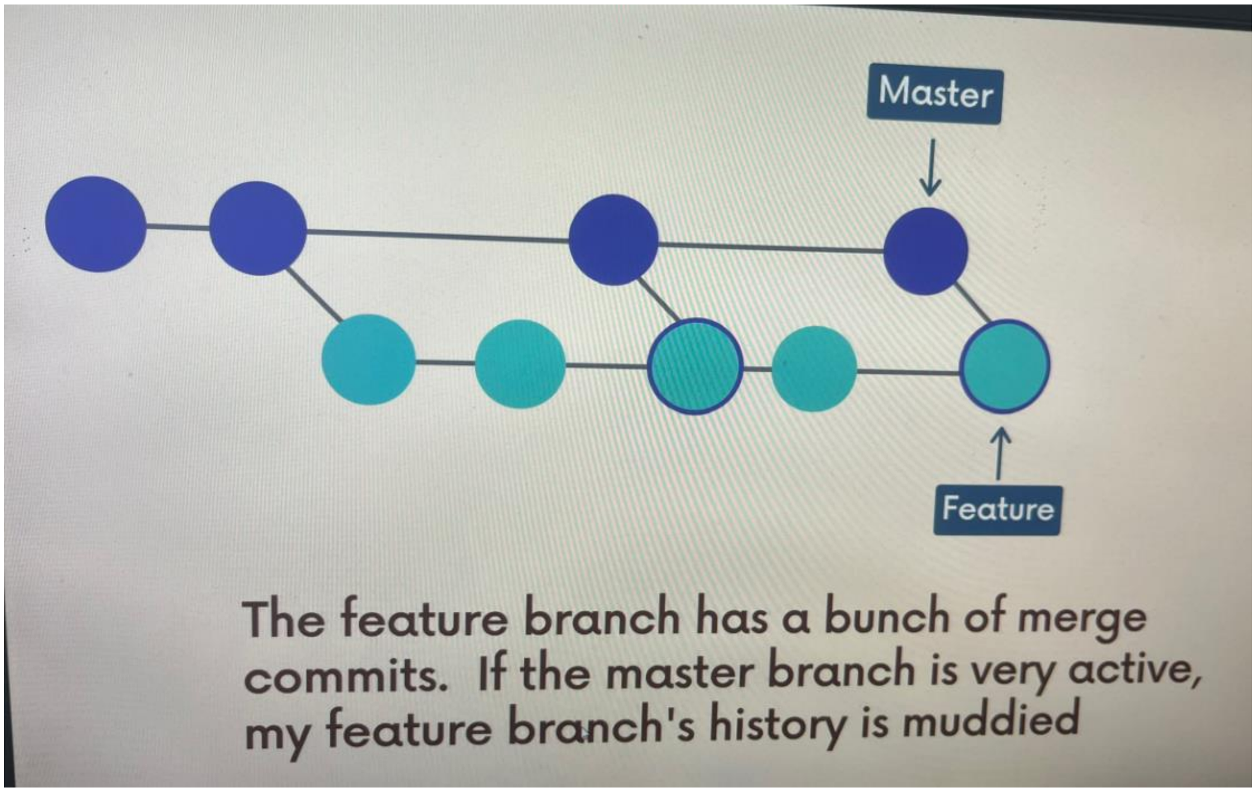


A coworker adds new work to master

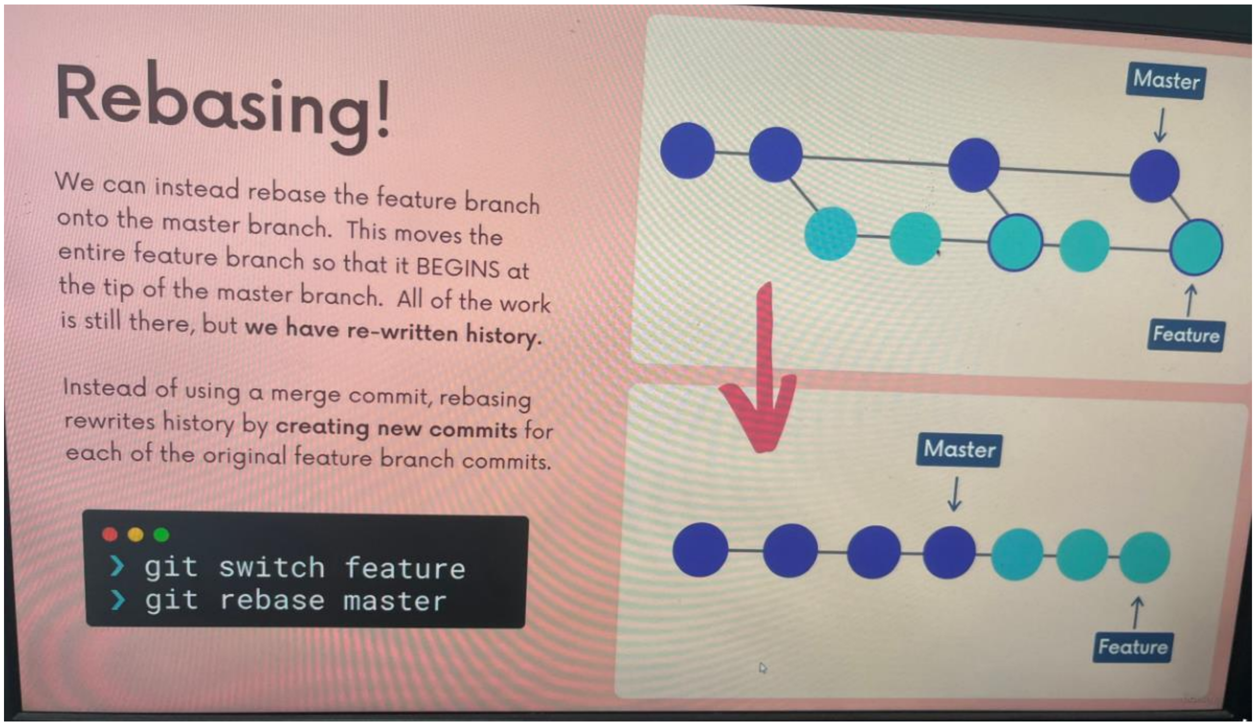


I merge master in to my feature branch

This results in yet another merge commit!



The feature branch has a bunch of merge commits. If the master branch is very active, my feature branch's history is muddied

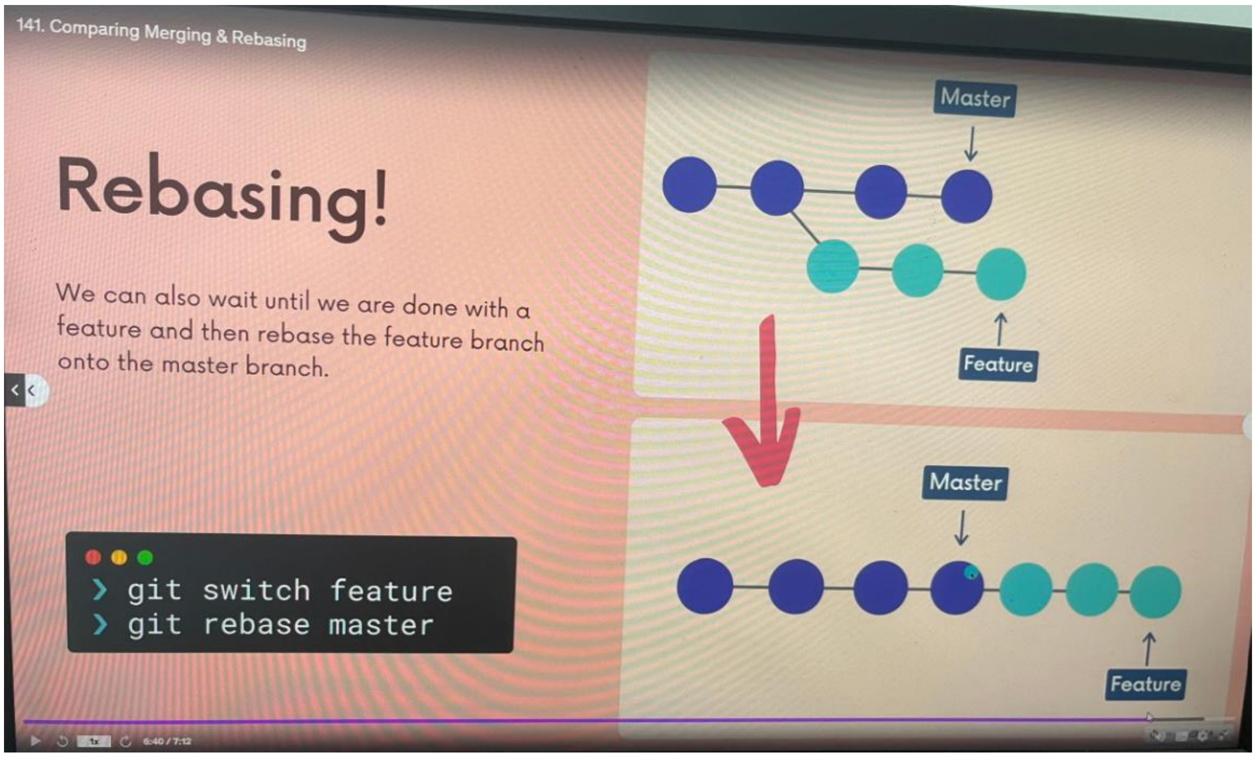


Rebasing!

We can instead rebase the feature branch onto the master branch. This moves the entire feature branch so that it BEGINS at the tip of the master branch. All of the work is still there, but **we have re-written history.**

Instead of using a merge commit, rebasing rewrites history by **creating new commits** for each of the original feature branch commits.

```
❯ git switch feature
❯ git rebase master
```


141. Comparing Merging & Rebasing

Rebasing!

We can also wait until we are done with a feature and then rebase the feature branch onto the master branch.

```
> git switch feature
> git rebase master
```

14b/ Why rebase?

We get a much cleaner project history. No unnecessary merge commits!. We end up with a linear project history.

WARNING!

Never rebase commits that have been shared with others. If you have already pushed commits up to Github... DO NOT rebase them unless you are positive no one on the team is using those commits.

Seriously!

You do not want to rewrite any git history that other people already have. It's a pain to reconcile the alternate histories.

14c/ Rewriting History

Sometimes we want to rewrite, delete, rename, or even reorder commits (before sharing them). We can do this using **git rebase**!

Interactive Rebase

Running git rebase with the -i option will enter the interactive mode, which allows us to edit commits, add files, drop commits, etc.. Note that we need to specify how far back we want to rewrite commits..

Also, notice that we are not rebasing onto another branch. Instead, we are rebasing a series of commits onto the HEAD the currently are based on.

Example command syntax: git rebase -i HEAD~4

15/ Git Tags: Marking Important Moments in History

15a/ Git Tags

Tags are pointers that refer to particular points in Git history. We can mark a particular moment in time with a tag. Tags are most often used to mark version release in projects (v4.1.0, v4.1.1, etc.)

Think of tags as branch references that do NOT CHANGE. Once a tag is created, it always refers to the same commit. It's just a label for a commit.

The Two Types:

There are two types of Git tags we can use: lightweight and annotated tags.

Lightweight tags are …lightweight. They are just a name/label that points to a particular commit.

Annotated tags store extra meta data including the author's name and email, the date and a tagging message (like a commit message)

15b/ Viewing Tags

git tag will print a list of all the tags in the current repository

Command syntax: git tag

We can search for tags that match a particular pattern by using **git tag -l** and then passing in a wildcard pattern. For example, **git tag -l "*beta*"** will print a list of tags that include "beta" in their name.

Command syntax: git tab -l "*beta*"

15c/ Checking Out Tags

To view the state of a repo at a particular tag, we can use **git checkout <tag>.** This puts us in detached HEAD!

Command Syntax: git checkout <tag>

15d/ Creating lightweight tags

To create a lightweight tag, use **git tag <tagname>.** By default, Git will create the tag referring to the commit that HEAD is referencing.

Command syntax: git tag <tag name>

15e/ Annotated Tags

Use **git tag -a** to create a new annotated tag. Git will then open your default text editor and prompt you for additional information.

Similar to git commit, we can also use the **-m** option to pass a message directly and forgo the opening of the text editor.

15f/ Tagging previous commits

So far, we 've seen how to tag the commit that HEAD references. We can also tag an older commit by providing the commit hash: **git tag -a <tagname> <commit-hash>**

Command Syntax: git tag <tagname> <commit-hash>

15g/ Forcing Tags

Git will yell at us if we try to reuse a tag that is already referring to a commit. If we use the -f option, we can FORCE our tag through

Command syntax: git tag -f <tagname>

15h/ Deleting Tags

To delete a tag , use **git tag -d <tagname>**

Command Syntax: git tag -d <tagname>

15i/ Pushing Tags

By default, the **git push** command doesn’t transfer tags to remote servers. If you have a lot of tags that you want to push up at once, you can use the –tags option to the **git push** command. This will transfer all of your tags to the remote server that are not already there.

Command syntax: git push --tags