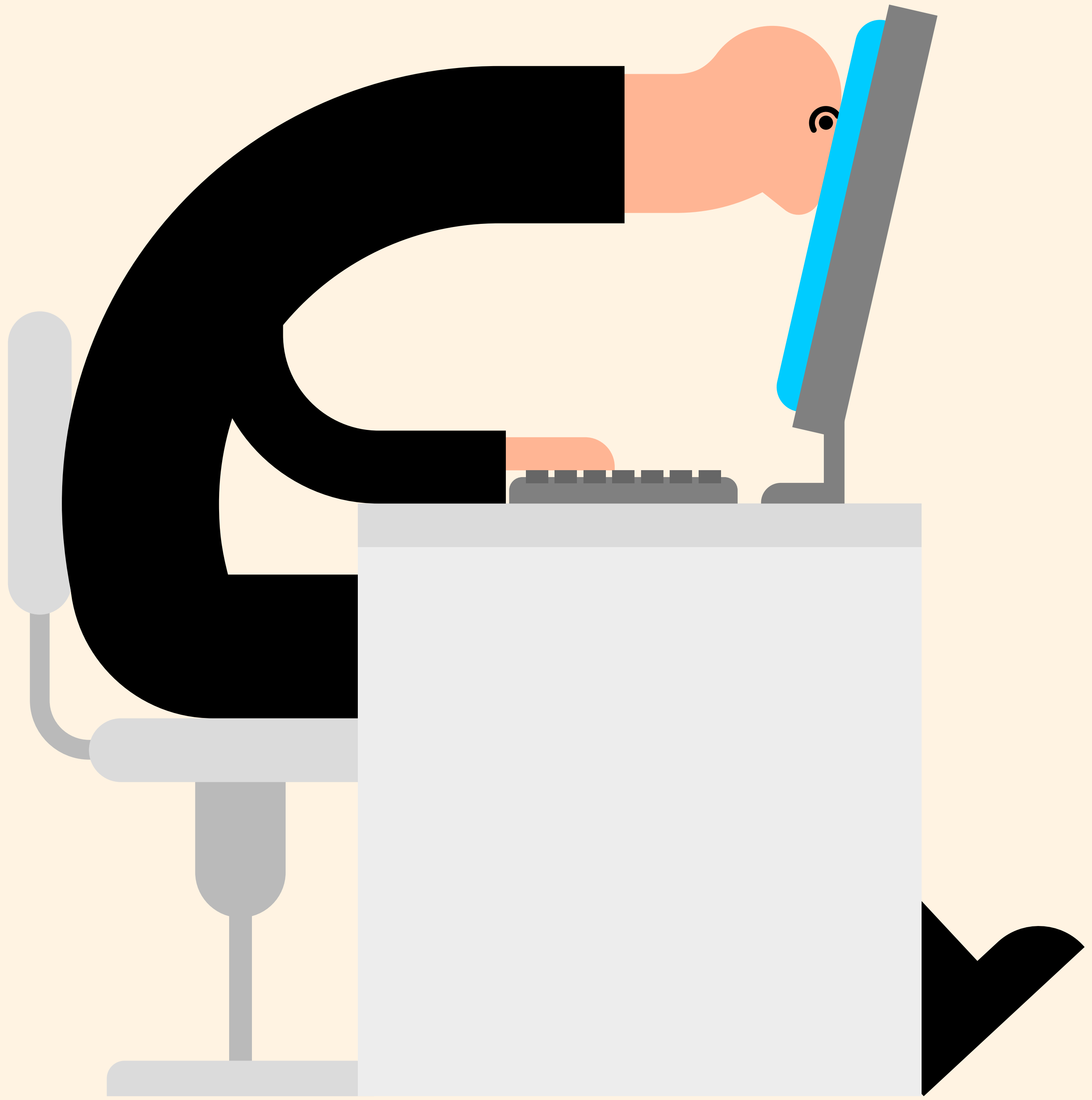
# Undoing Stuff & Time Traveling

## Checkout

The **git checkout** command is like a Git Swiss Army knife. Many developers think it is overloaded,  which is what lead to the addition of the **git switch** and **git restore** commands

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

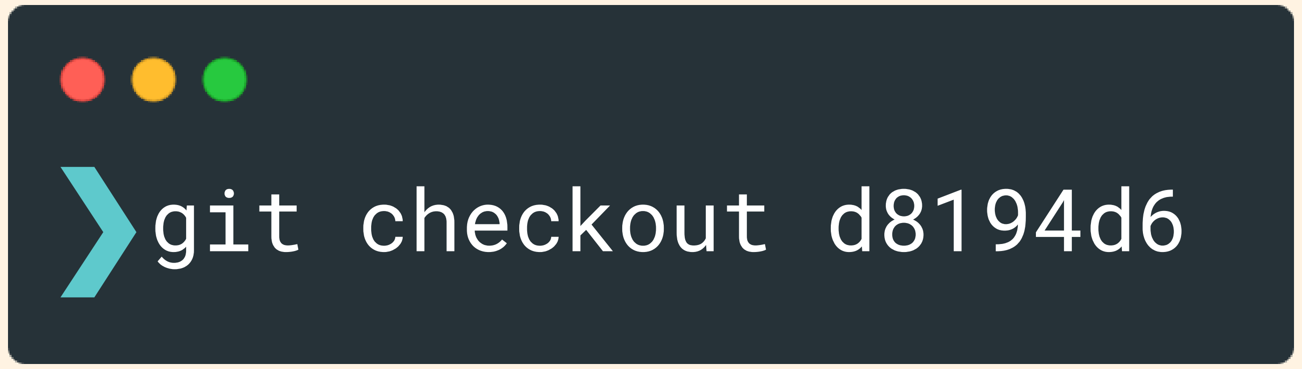


## Checkout

We can use **git checkout commit <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.

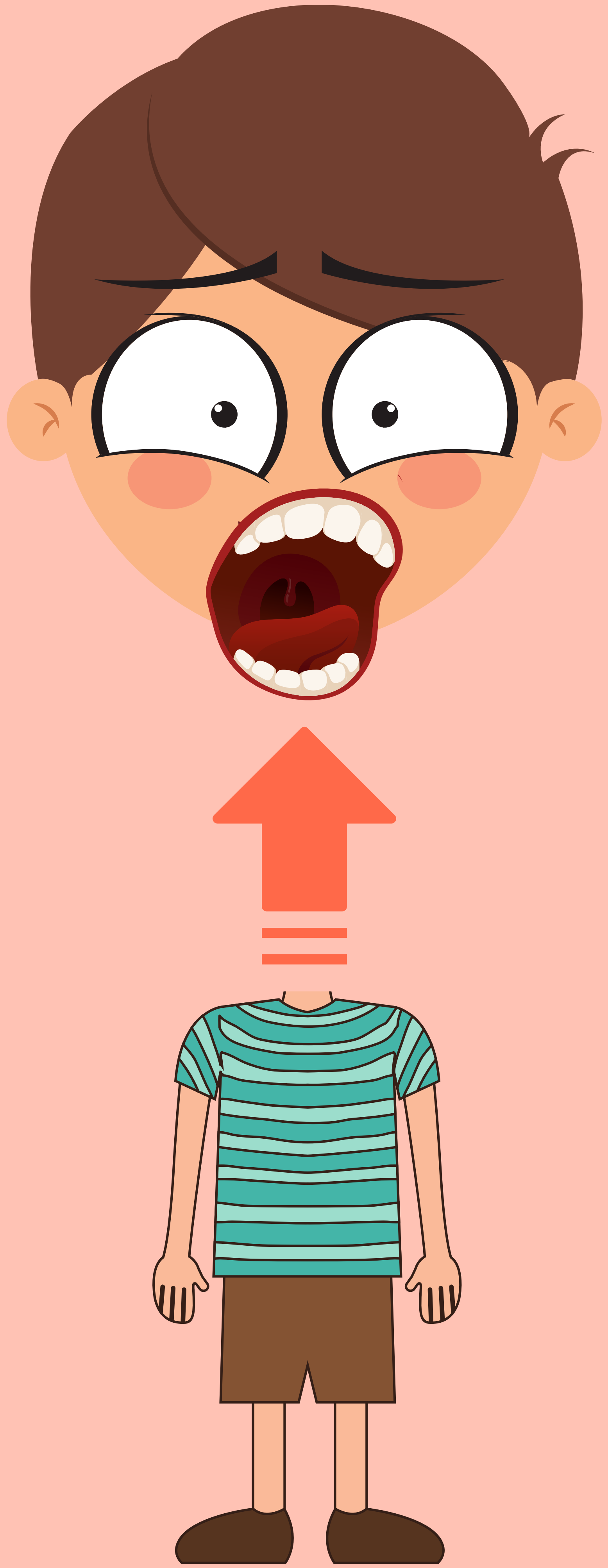
Don't panic when you see the following message...



```
❯git checkout d8194d6
```

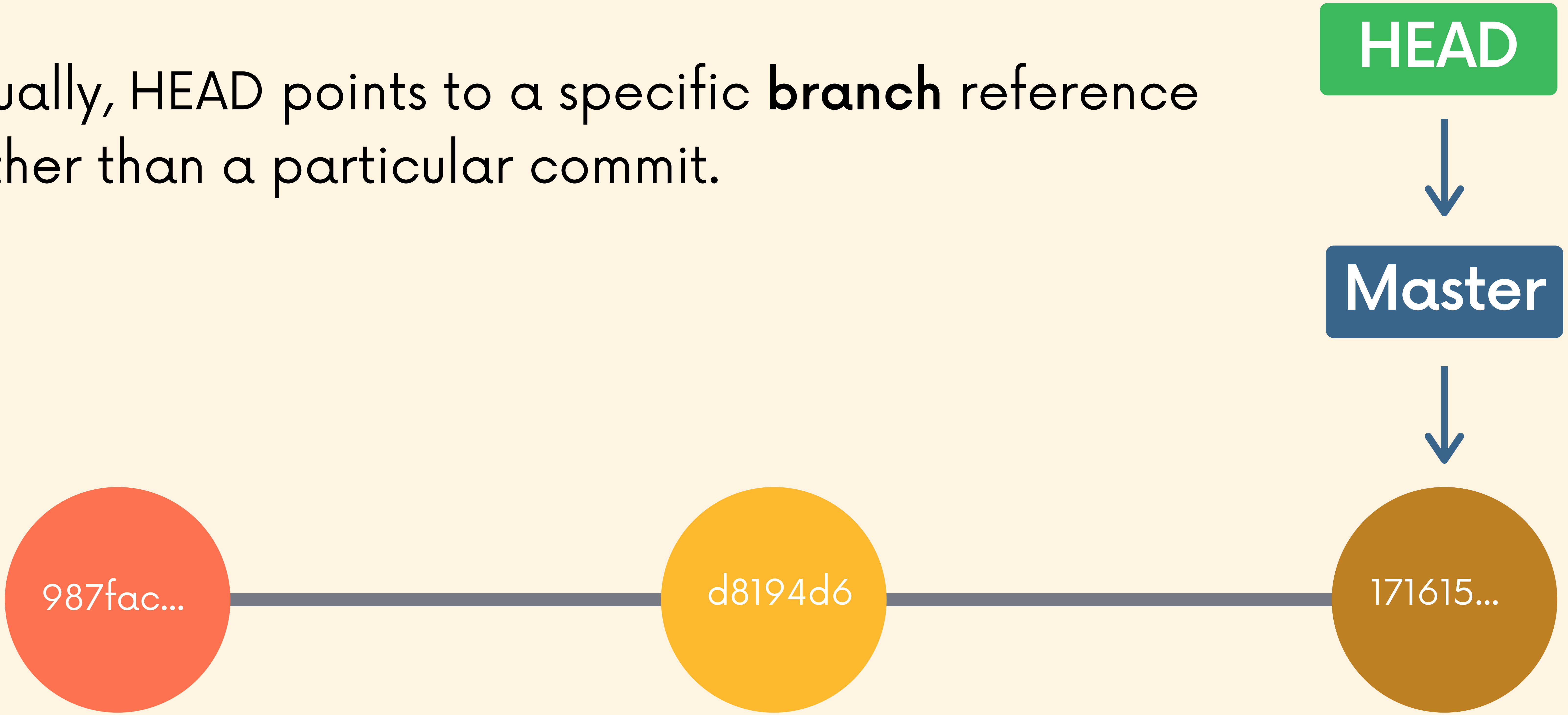
# Detached HEAD??

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.



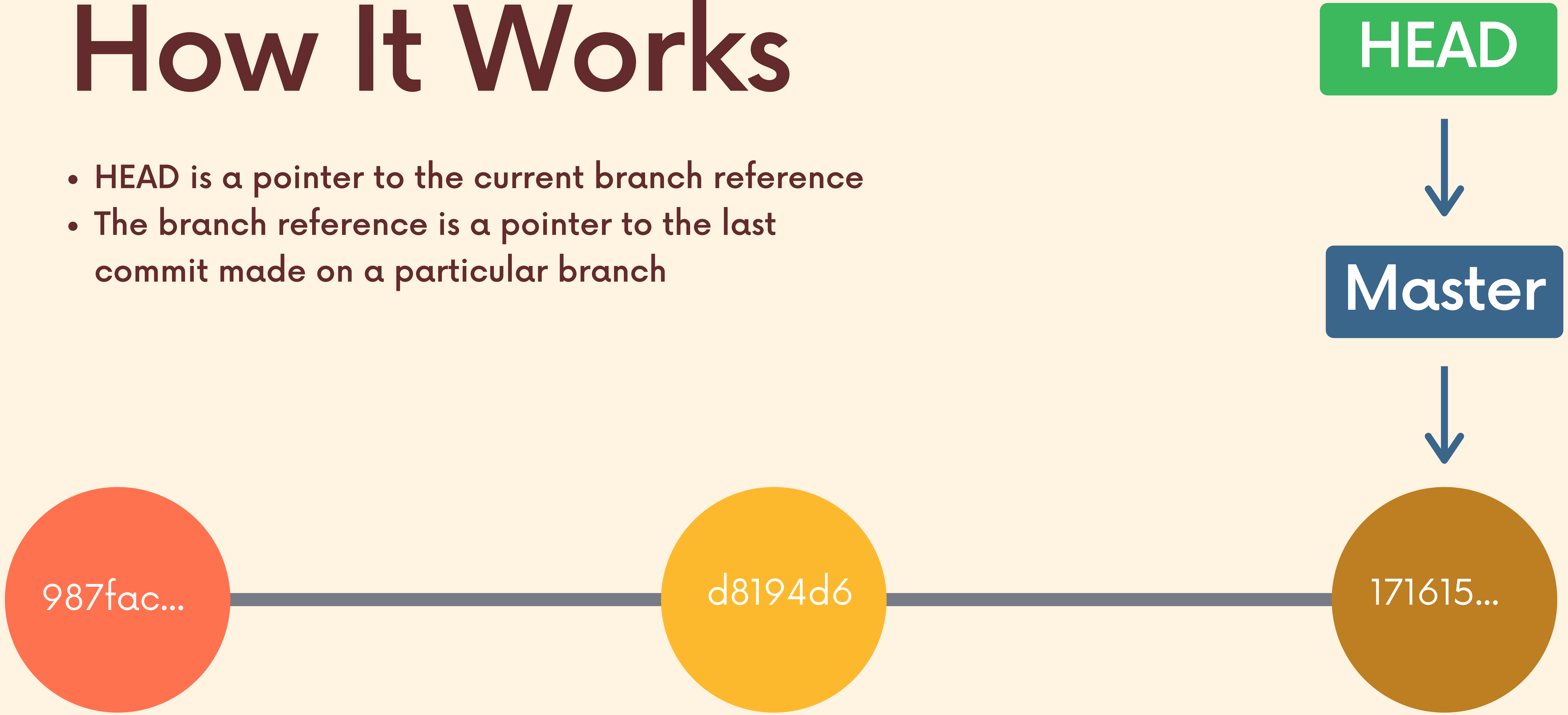
## What On Earth Is Going On??

Usually, HEAD points to a specific **branch** reference rather than a particular commit.

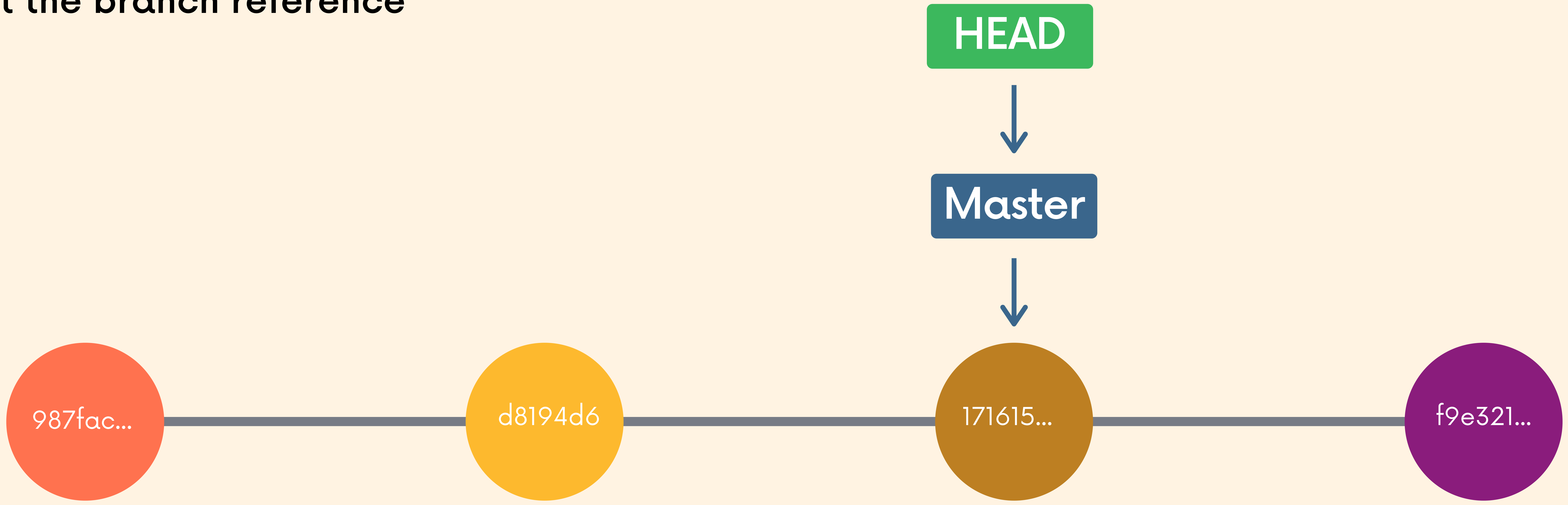


## How It Works

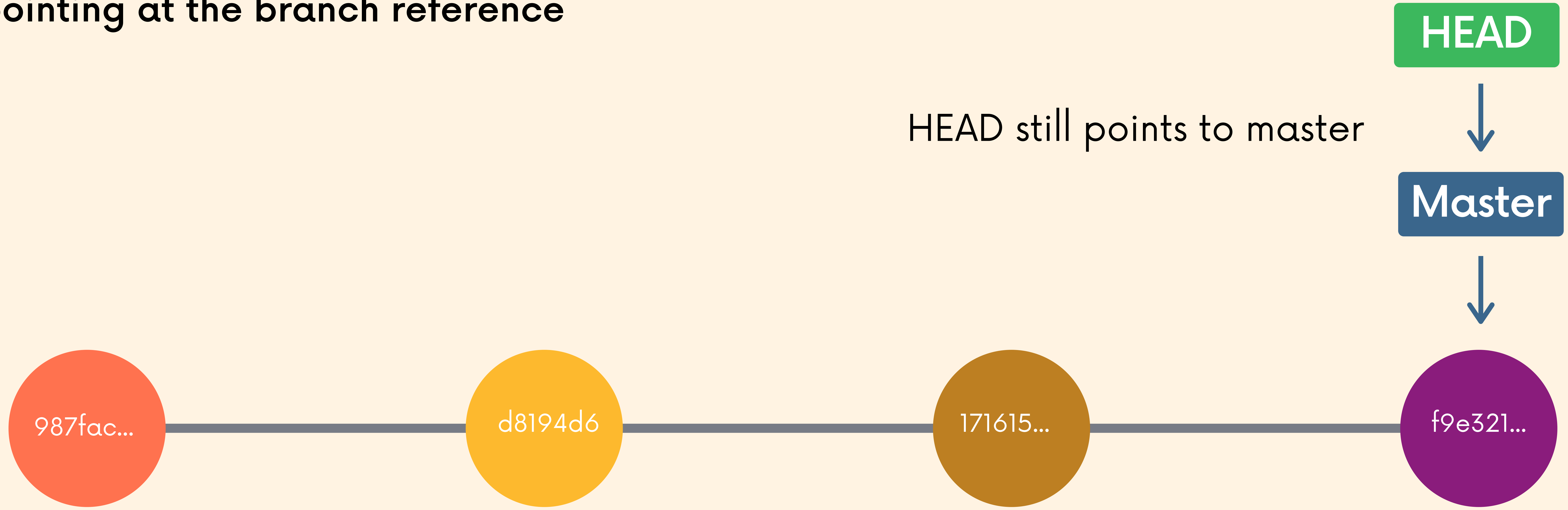
- HEAD is a pointer to the current branch reference
- The branch reference is a pointer to the last commit made on a particular branch



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**

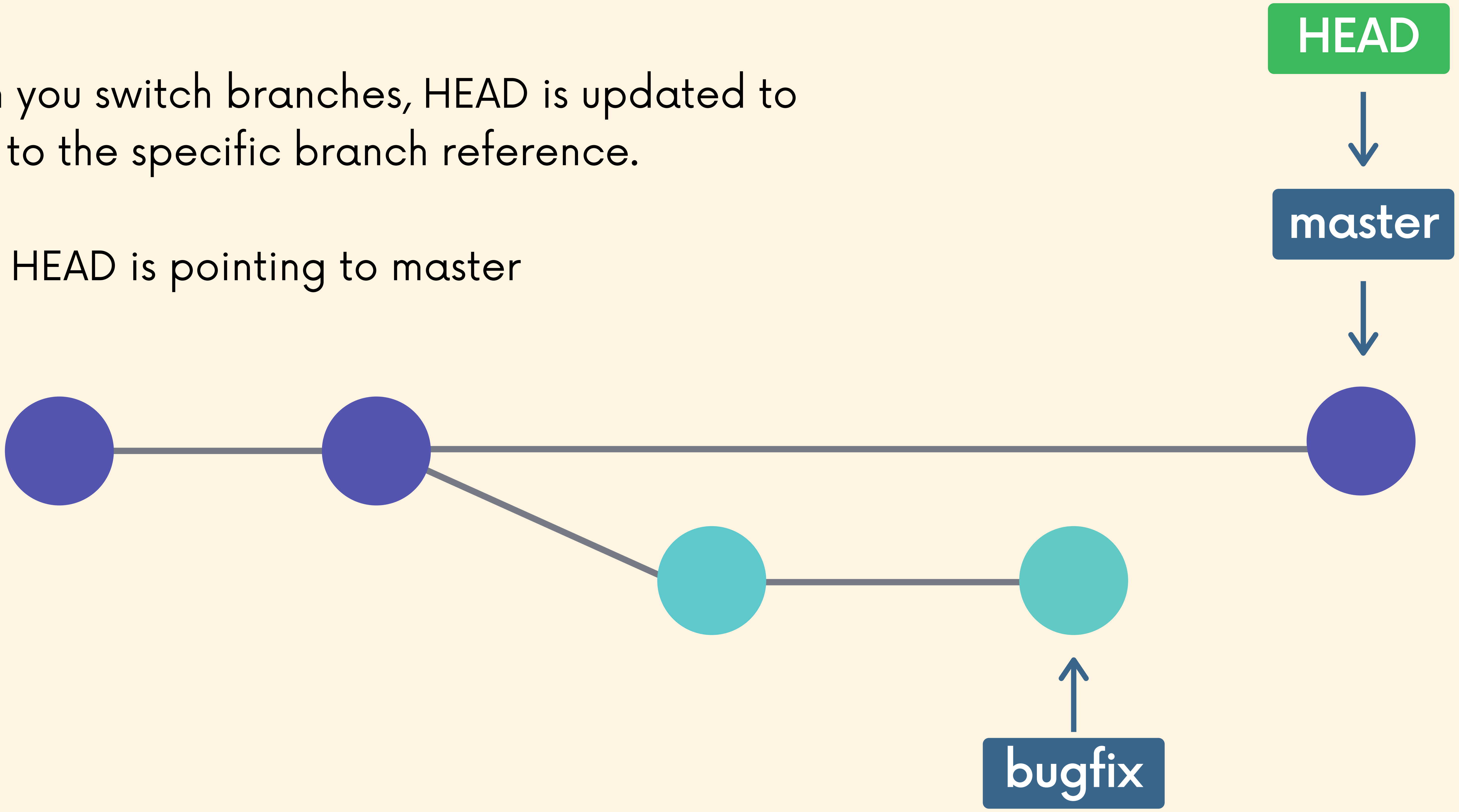


When we make a new commit, the branch pointer is updated to reflect the new commit.
**The HEAD remains the same, because it's pointing at the branch reference**

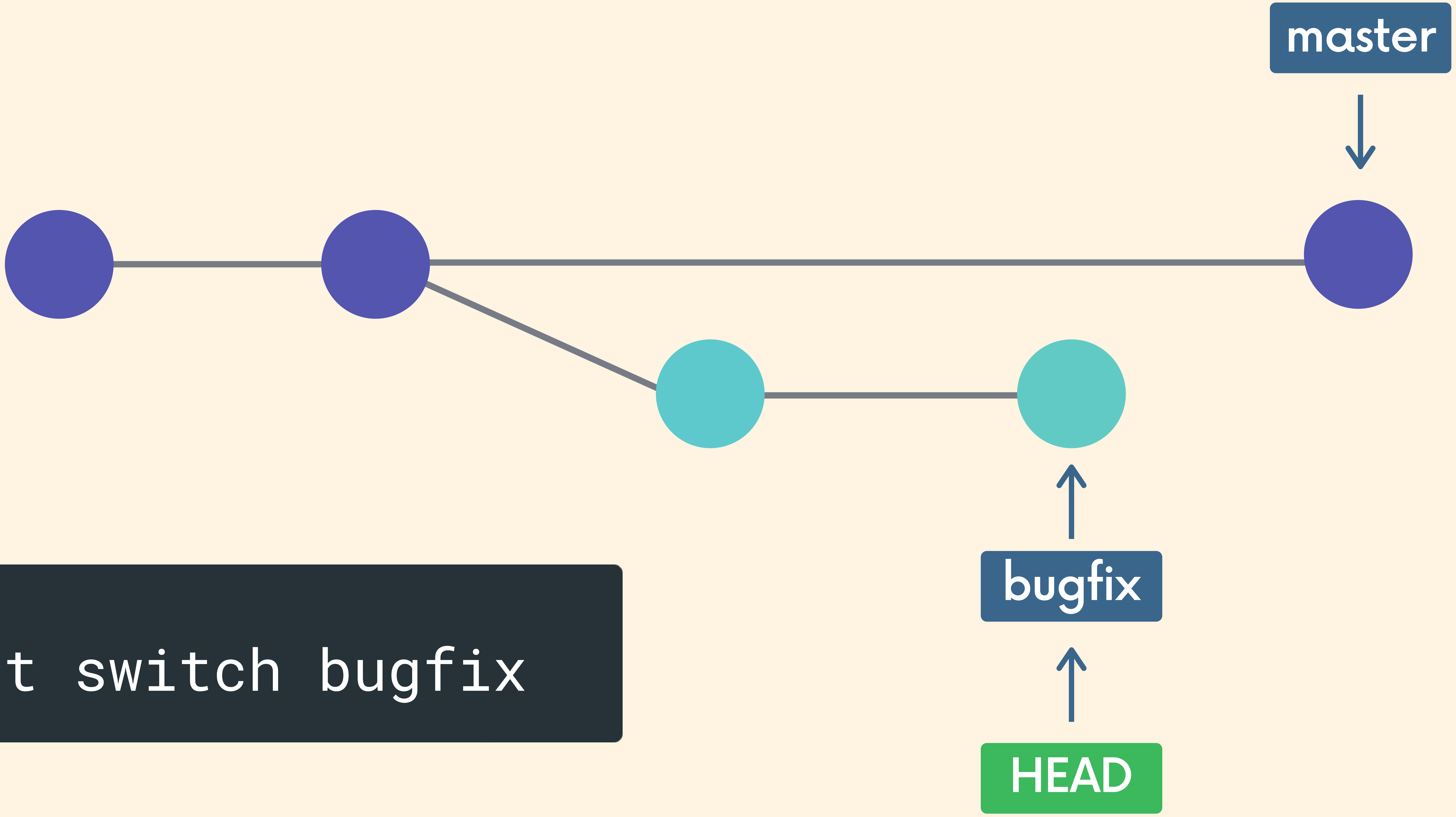


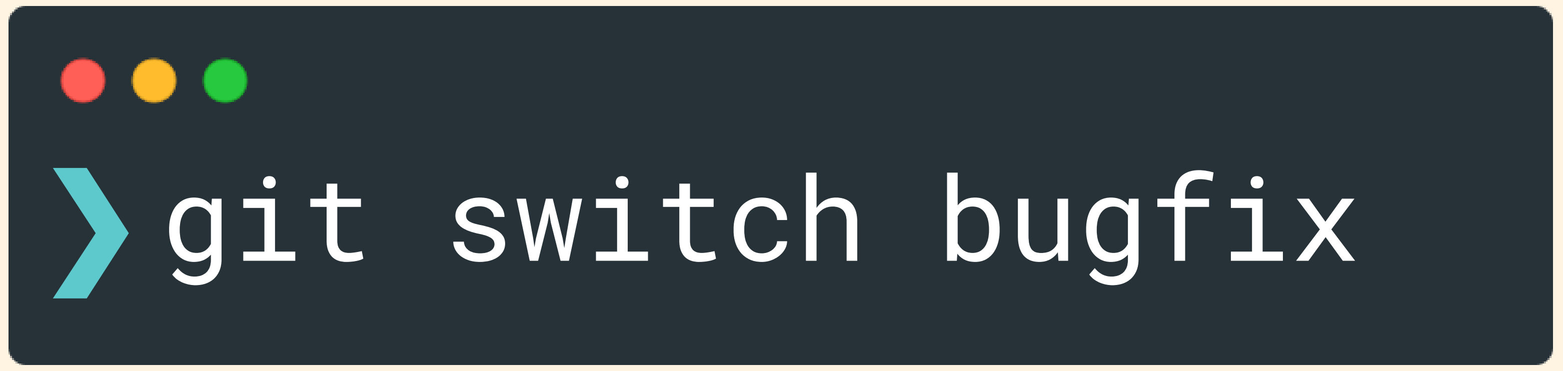
When you switch branches, HEAD is updated to point to the specific branch reference.

Here, HEAD is pointing to master



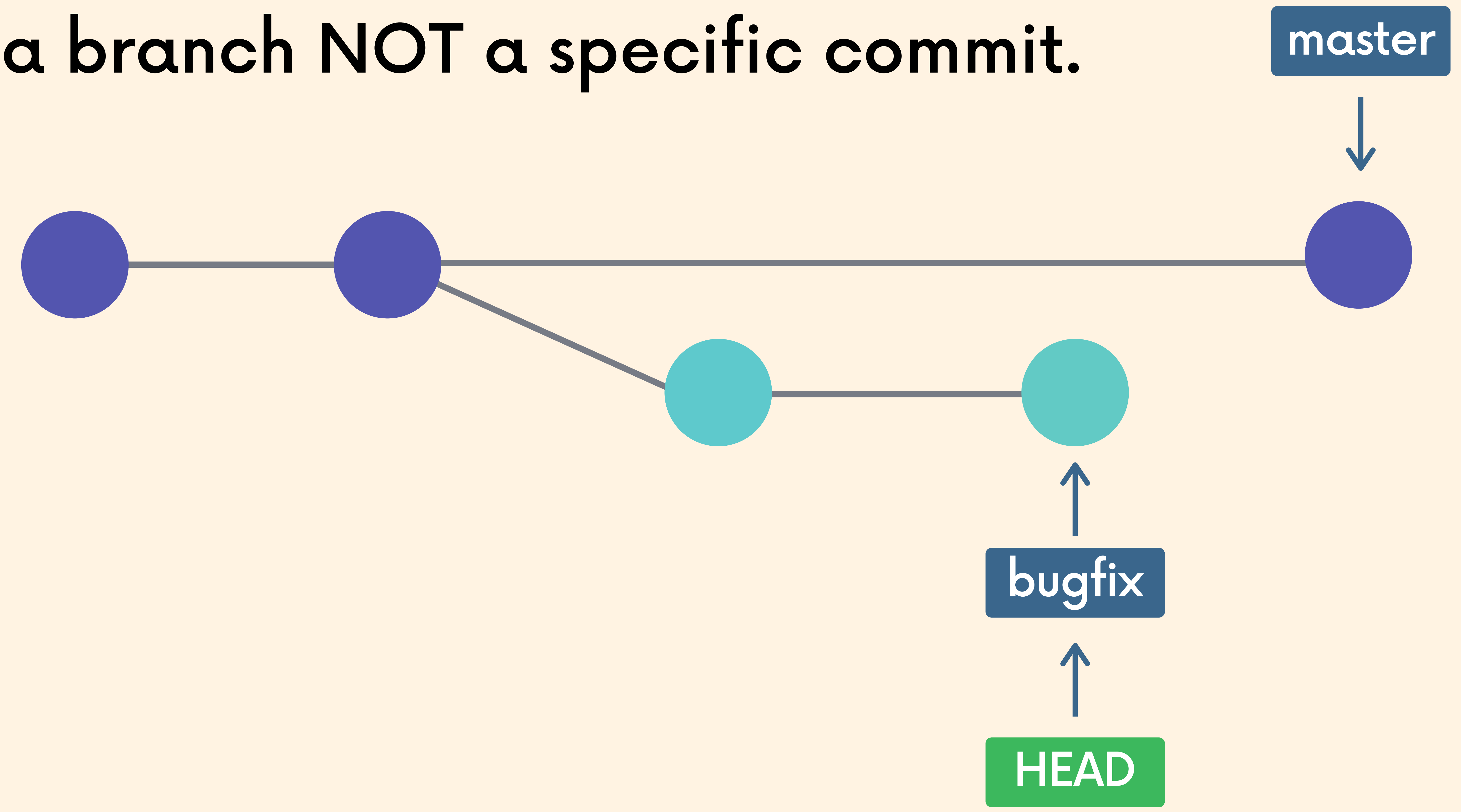
If we switch to the bugfix branch, HEAD is now pointing at the bugfix reference.



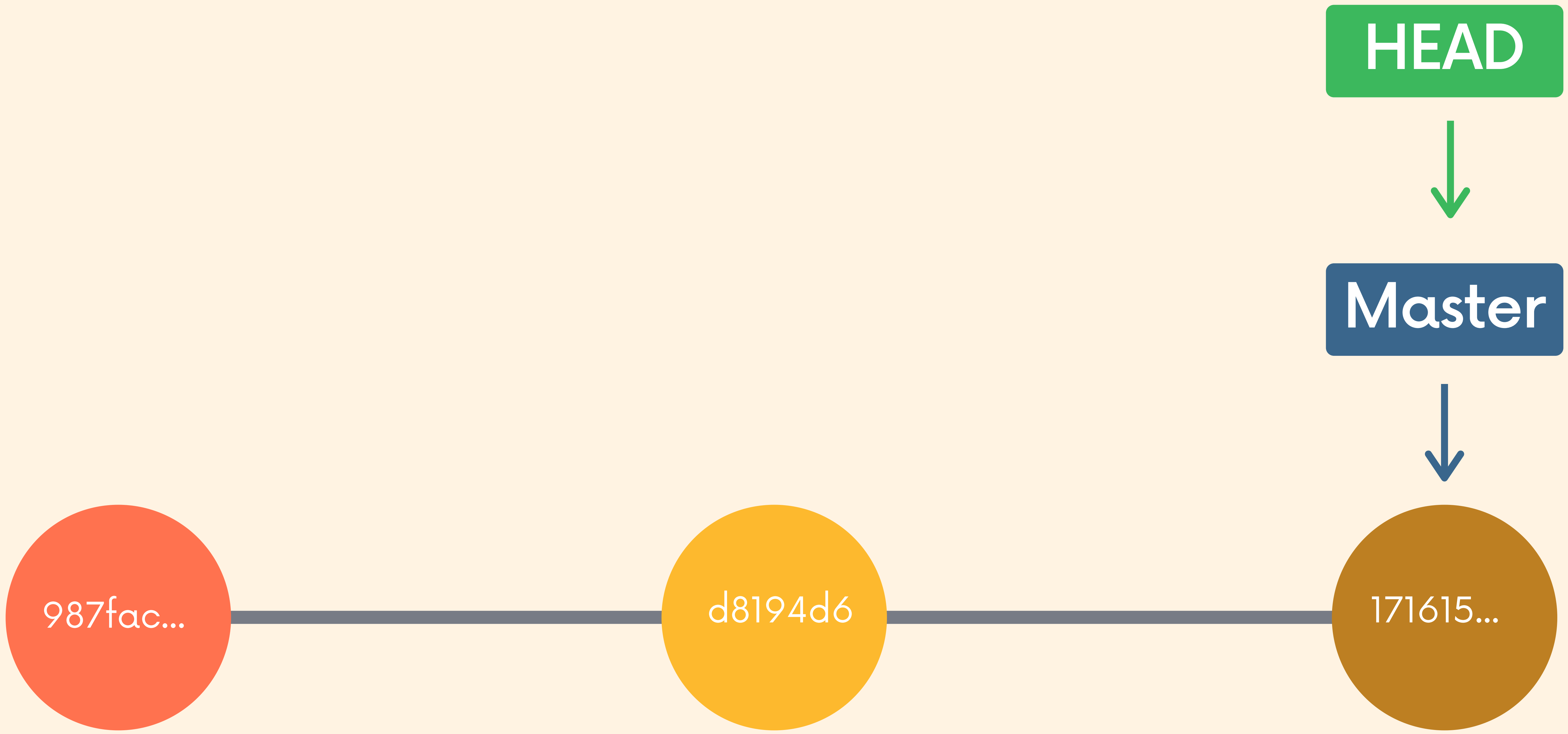


```
❯ git switch bugfix
```

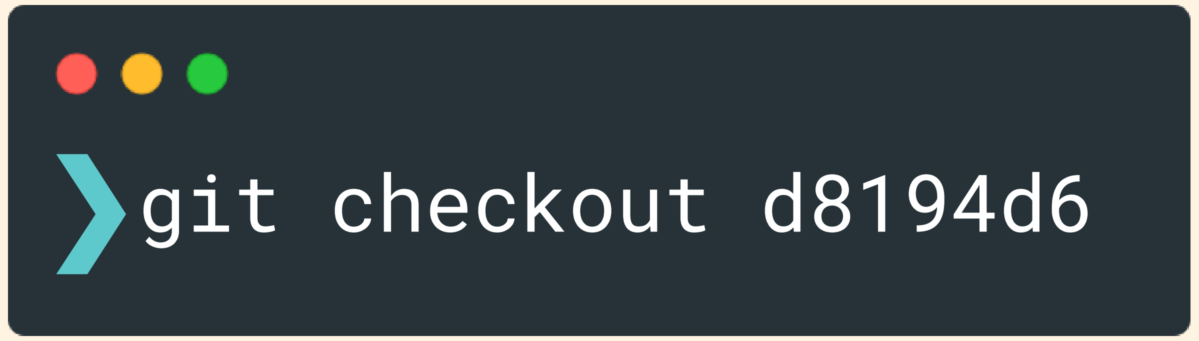
## This is all to say that HEAD usually refers to a branch NOT a specific commit.



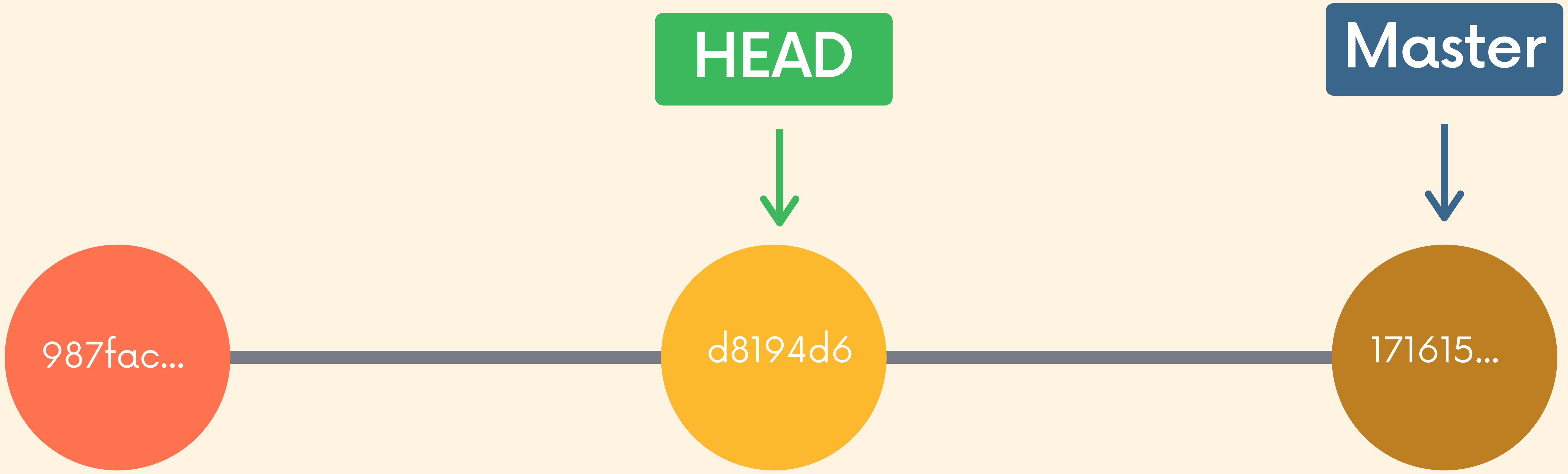
# Back to this Detached HEAD thing


HEAD
Master
987fac...
d8194d6
171615...

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



```
git checkout d8194d6
```



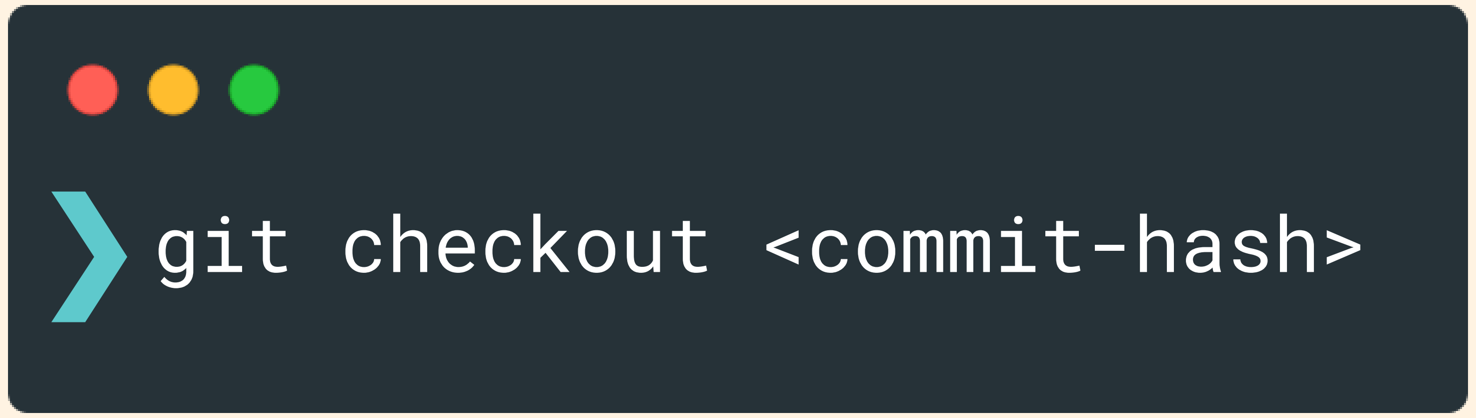
DETACHED HEAD!

## Detached HEAD

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

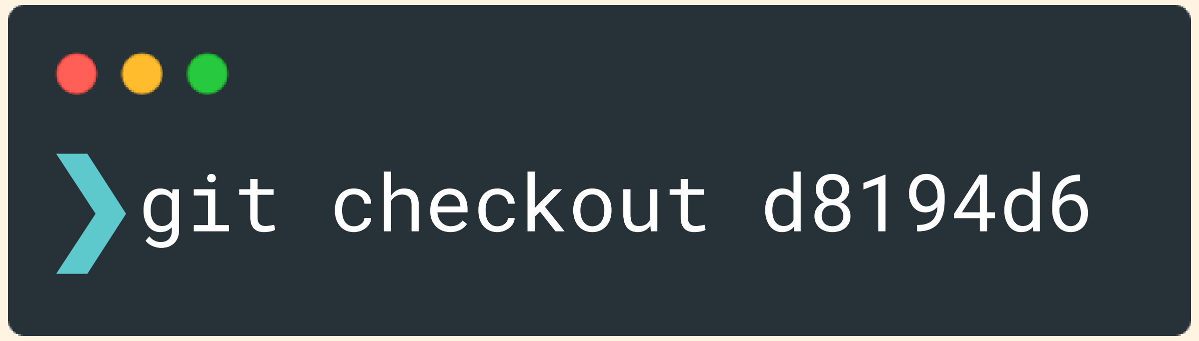
**You have a couple options:**

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.

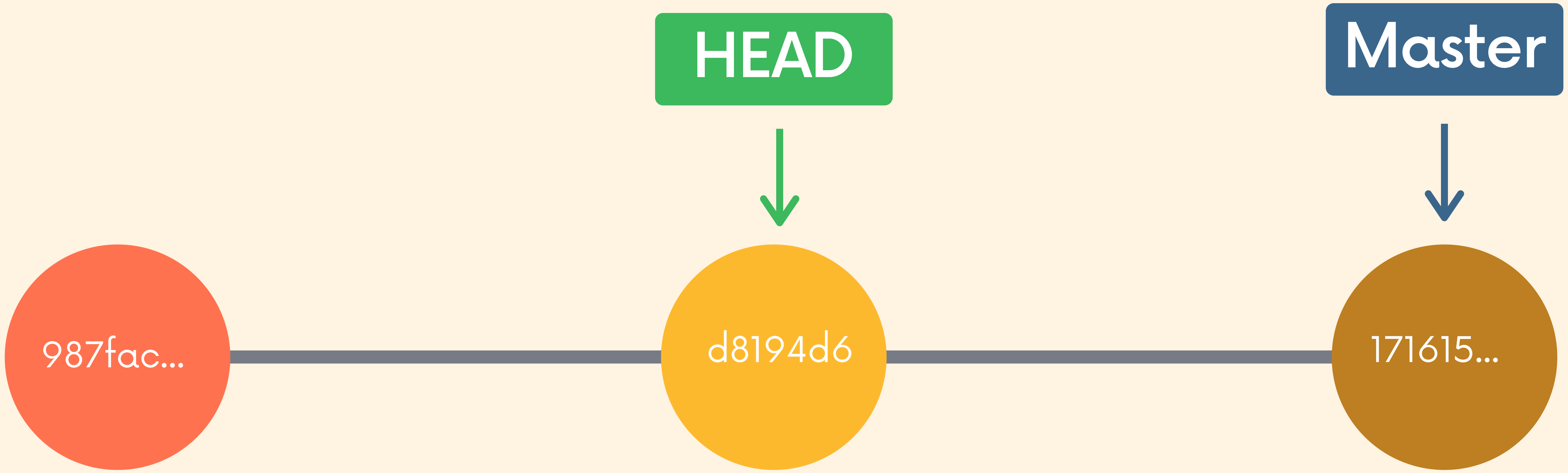


```
git checkout <commit-hash>
```

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



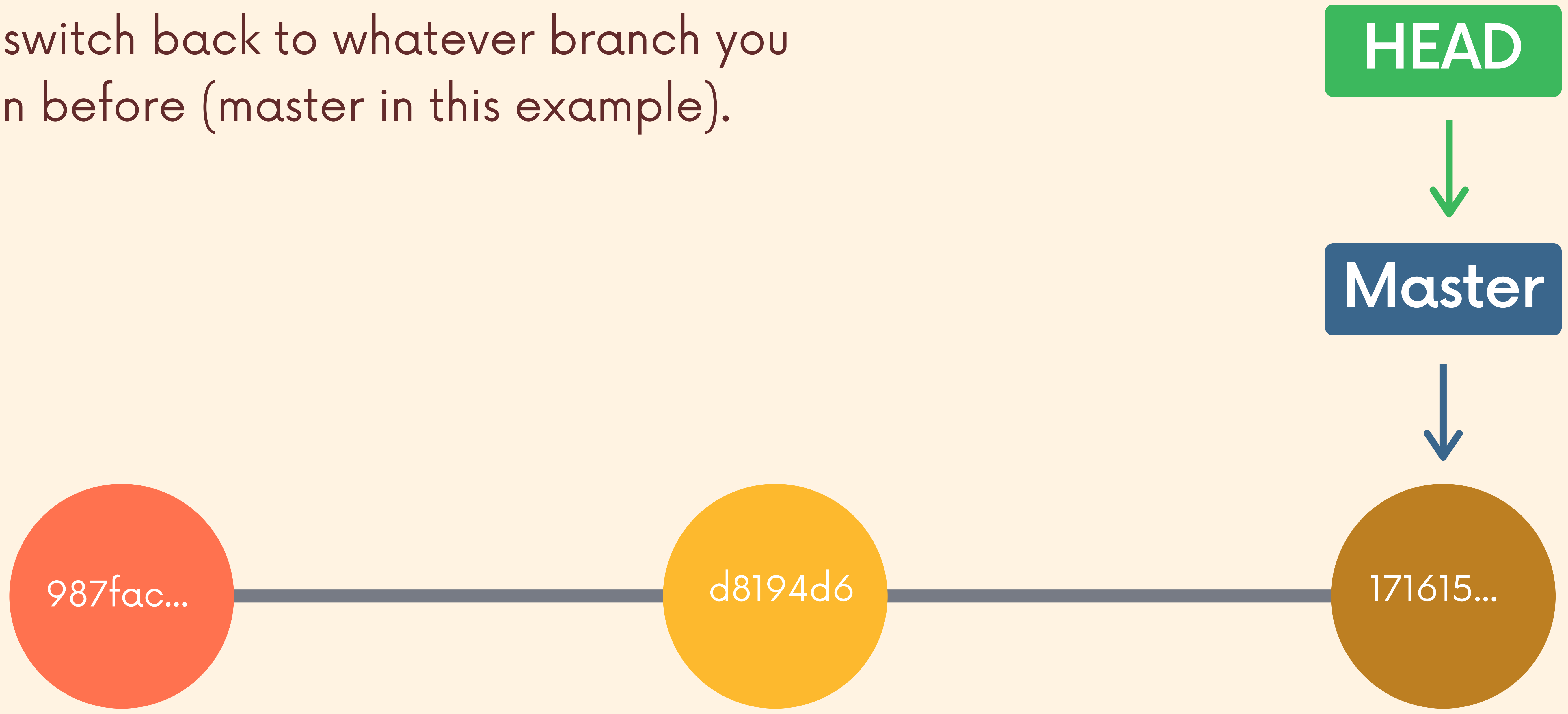
```
❯git checkout d8194d6
```



DETACHED HEAD!

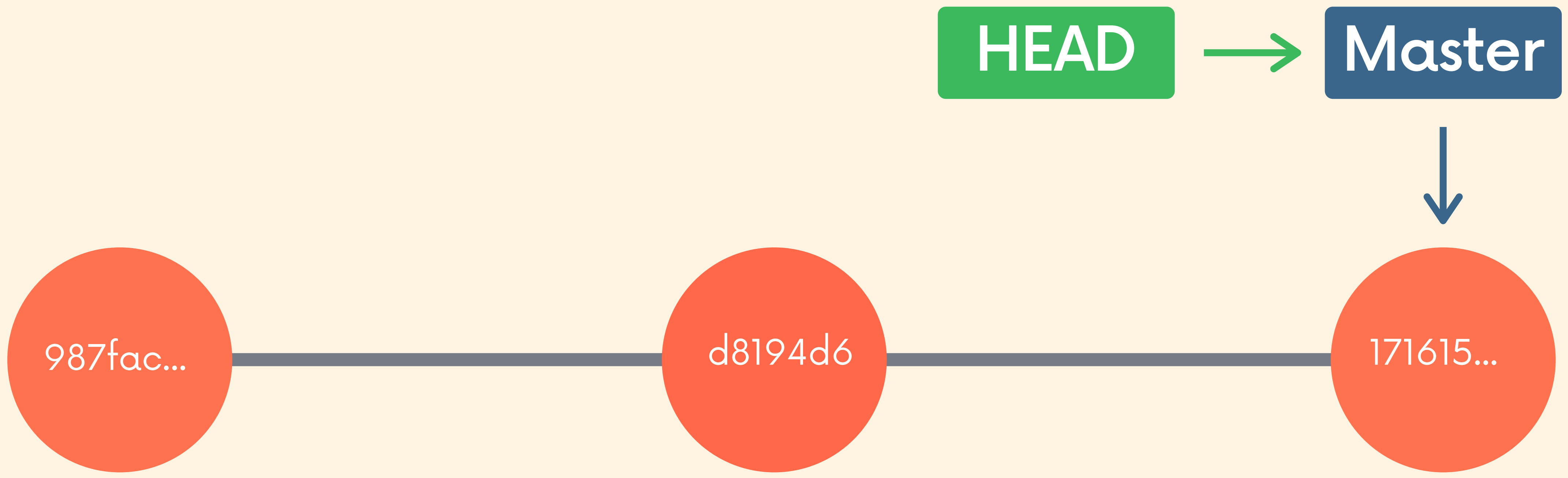
```
git switch master
```

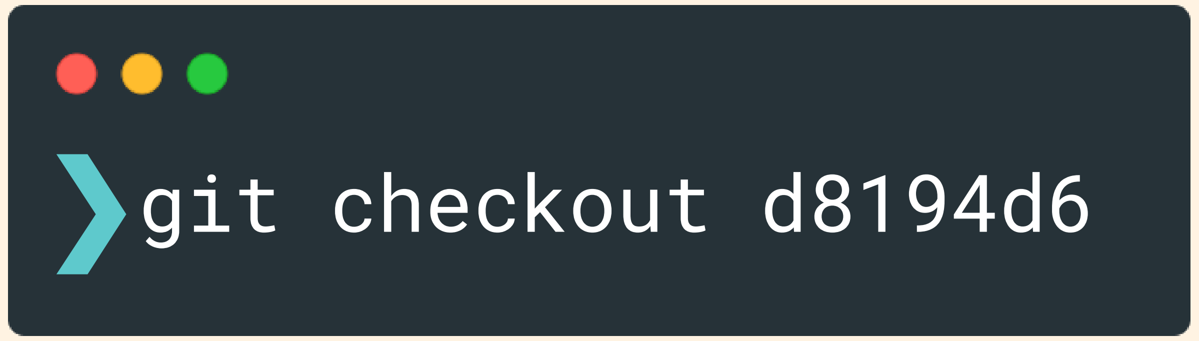
Simply switch back to whatever branch you were on before (master in this example).



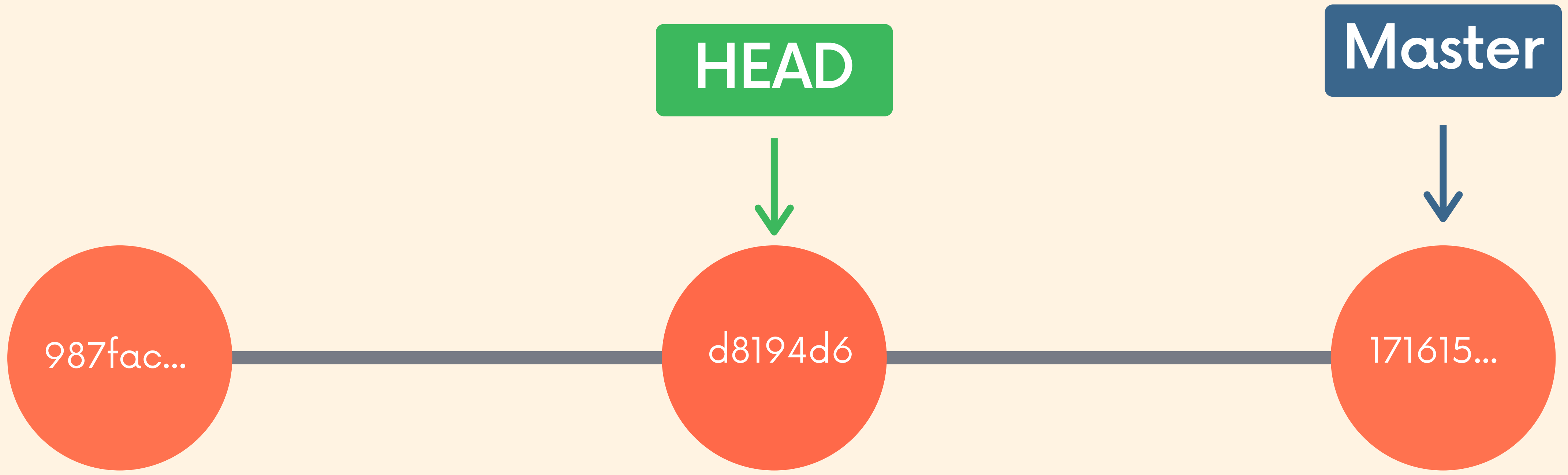
RE-ATTACHED HEAD!

Suppose you want to go back to an old commit and make some new changes





```
❯git checkout d8194d6
```

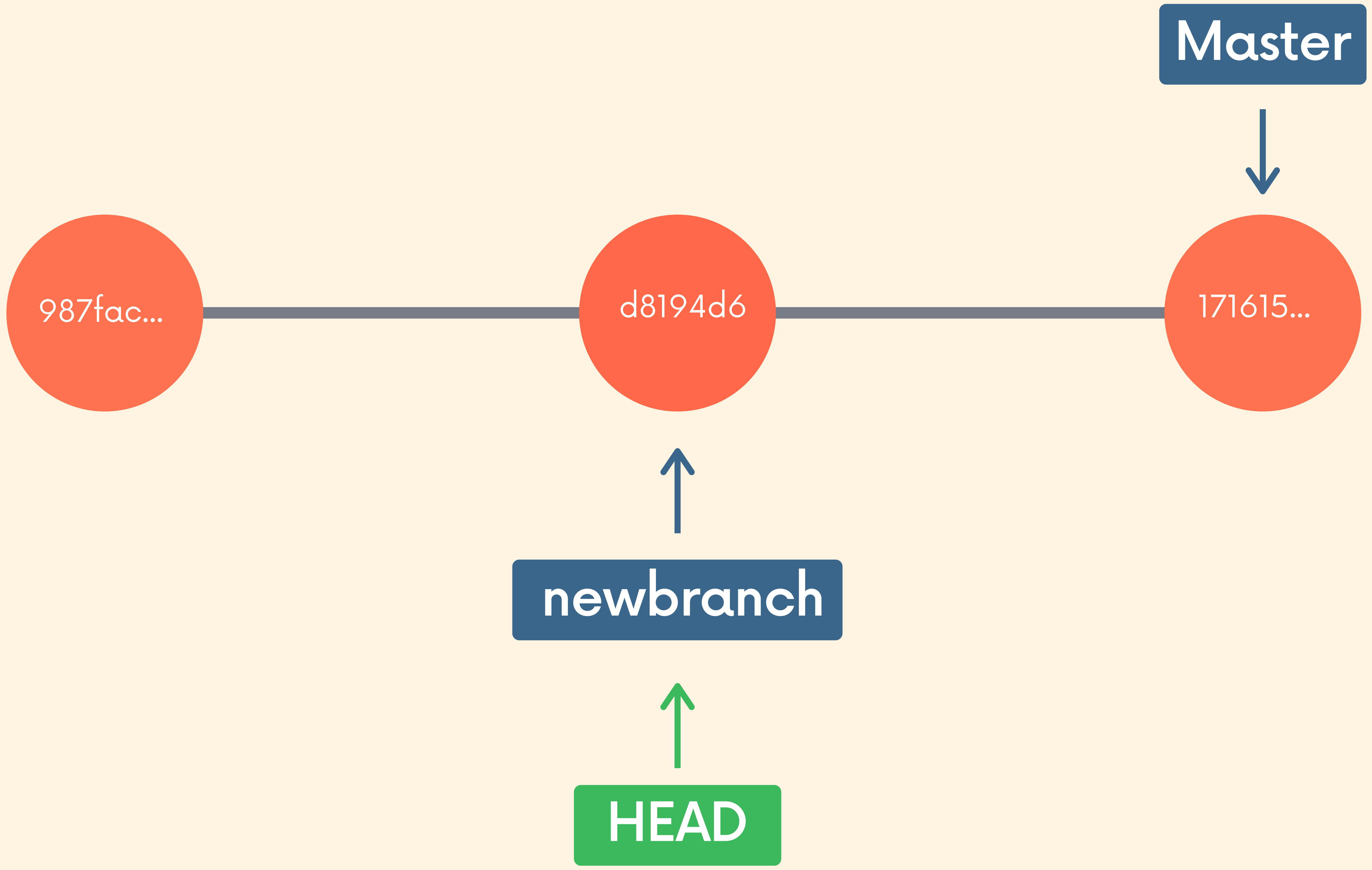


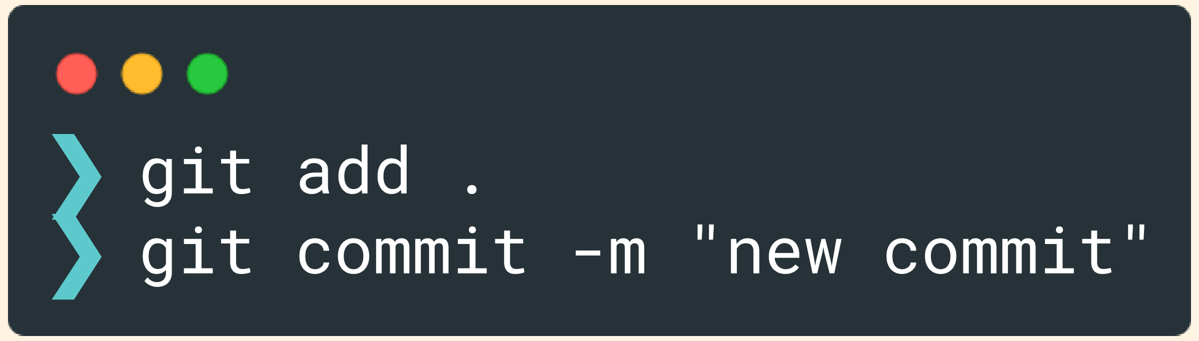
Checkout the old commit.
Now in detached HEAD state.

```
git switch -c newbranch
```

While in detached HEAD,
Make a new branch and switch to it.

Head is now back to pointing at a branch reference!

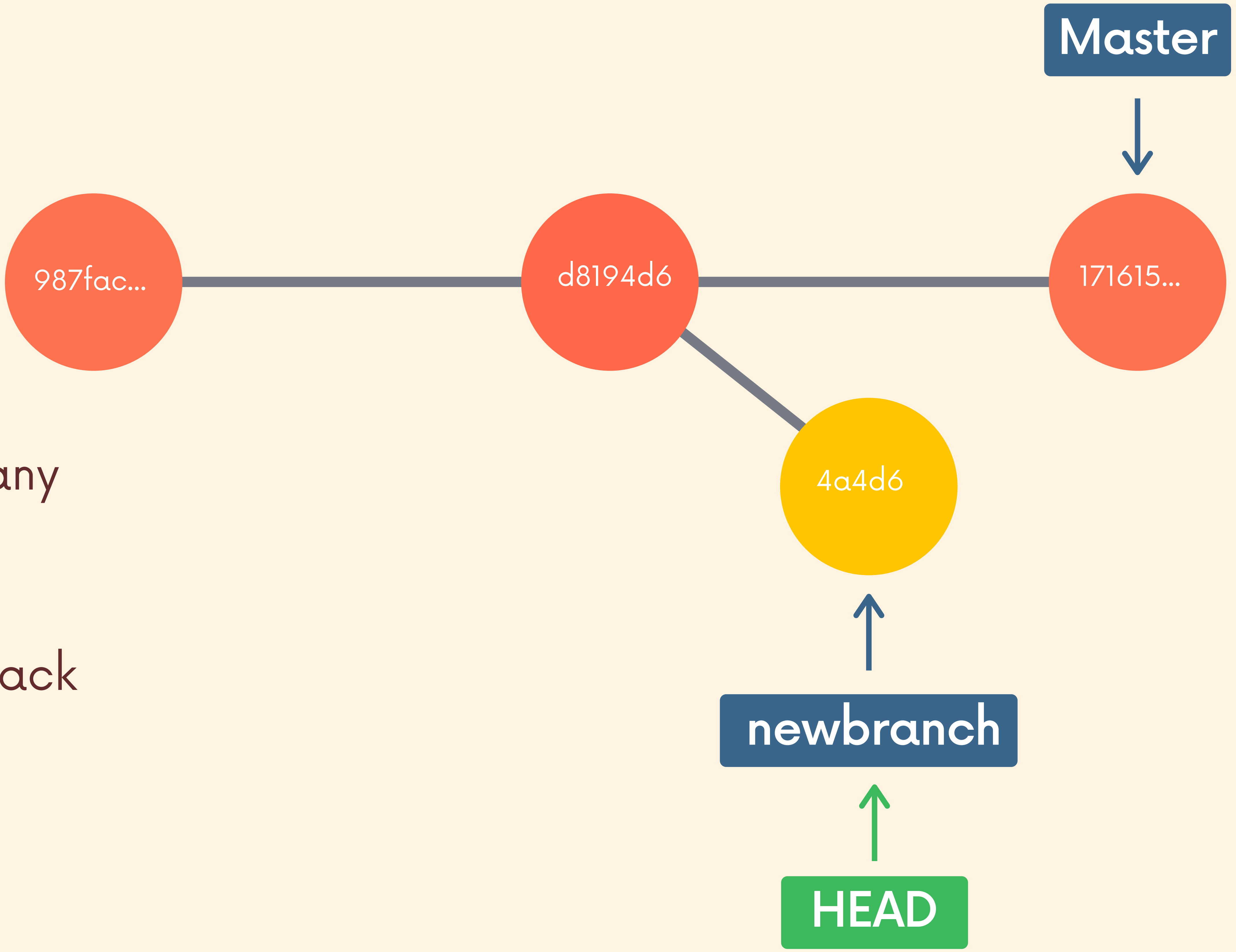




```
git add .
git commit -m "new commit"
```

Now on the new branch, make as many new commits as you want!

It's like you time traveled! We went back to an old commit and made a new branch based on it.

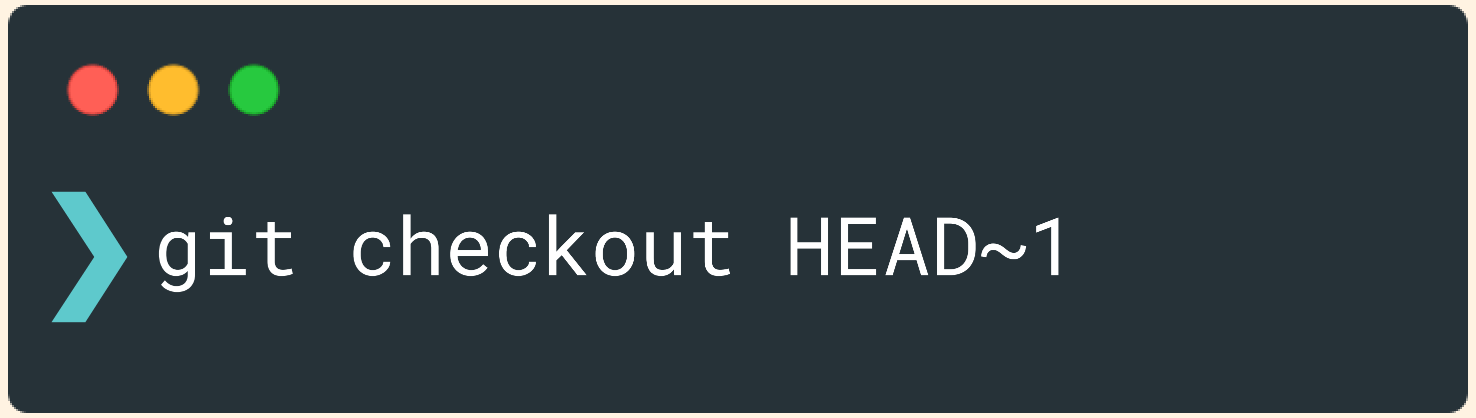


# Checkout

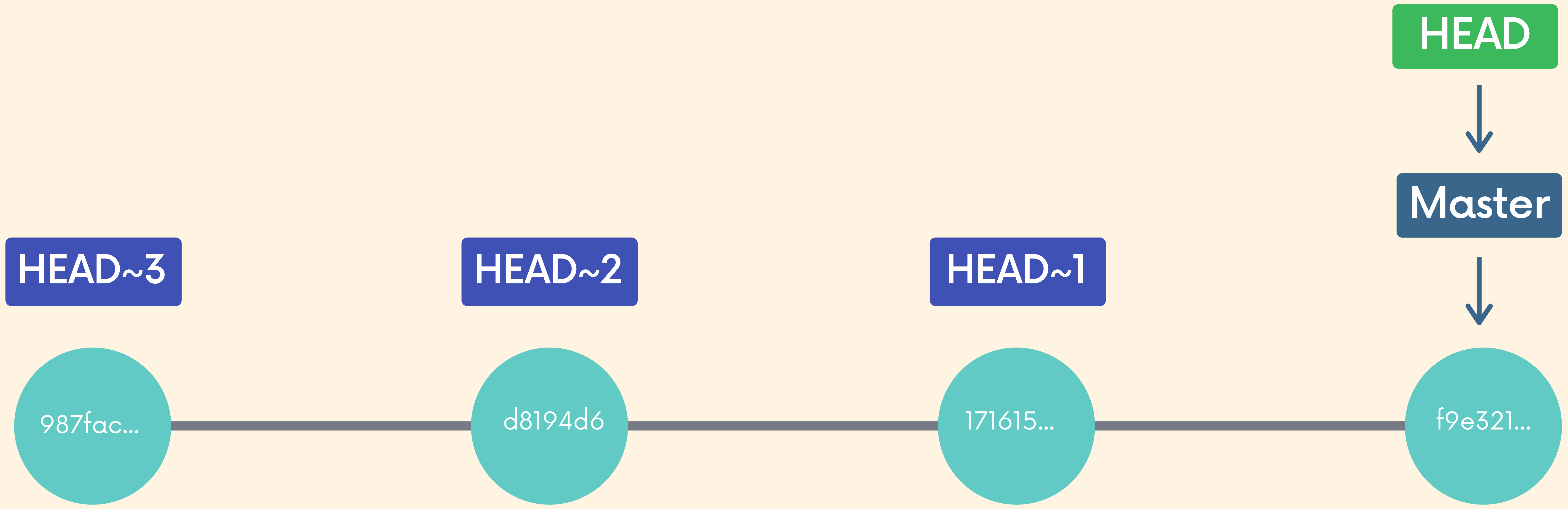
**git checkout** supports a slightly odd syntax for referencing previous commits relative to a particular commit.

HEAD~1 refers to the commit before HEAD (parent)
HEAD~2 refers to 2 commits before HEAD (grandparent)

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



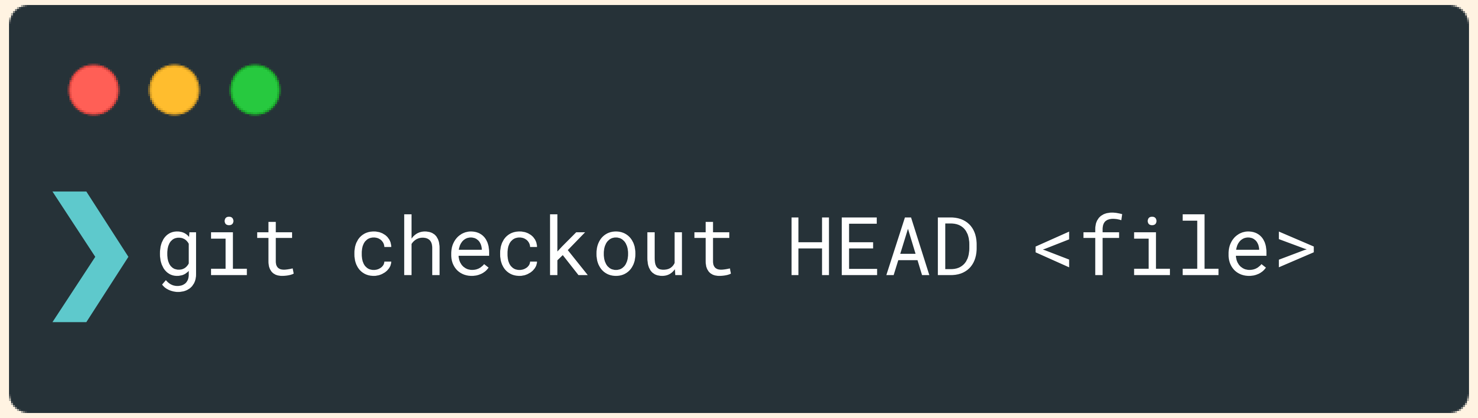
```
git checkout HEAD~1
```


HEAD
Master
HEAD~3
HEAD~2
HEAD~1
987fac...
d8194d6
171615...
f9e321...

## Discarding Changes

Suppose you've made some changes to a file but don't 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.

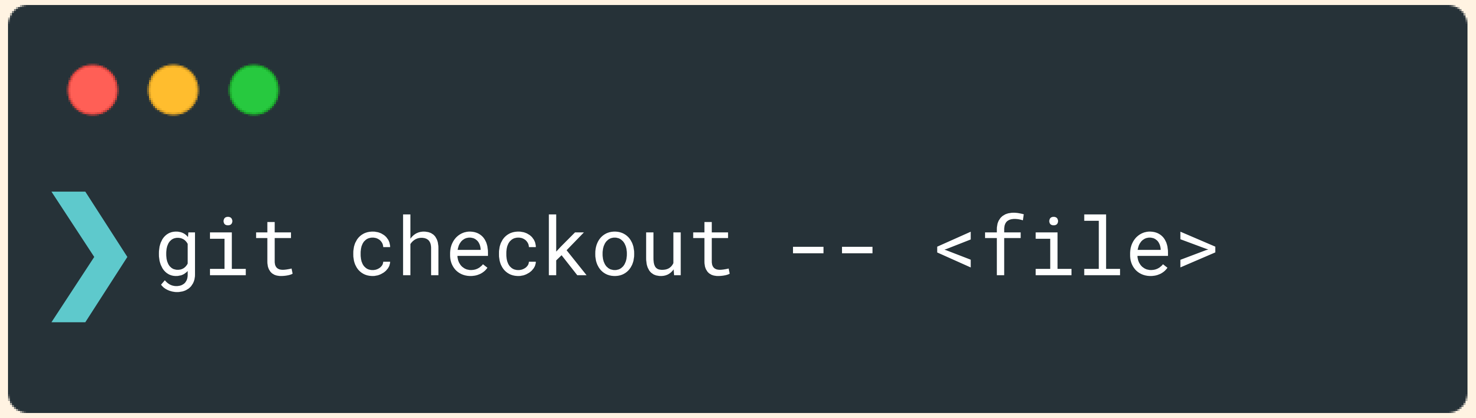


```
❯ git checkout HEAD <file>
```

# 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.



```
git checkout -- <file>
```

## Restore

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

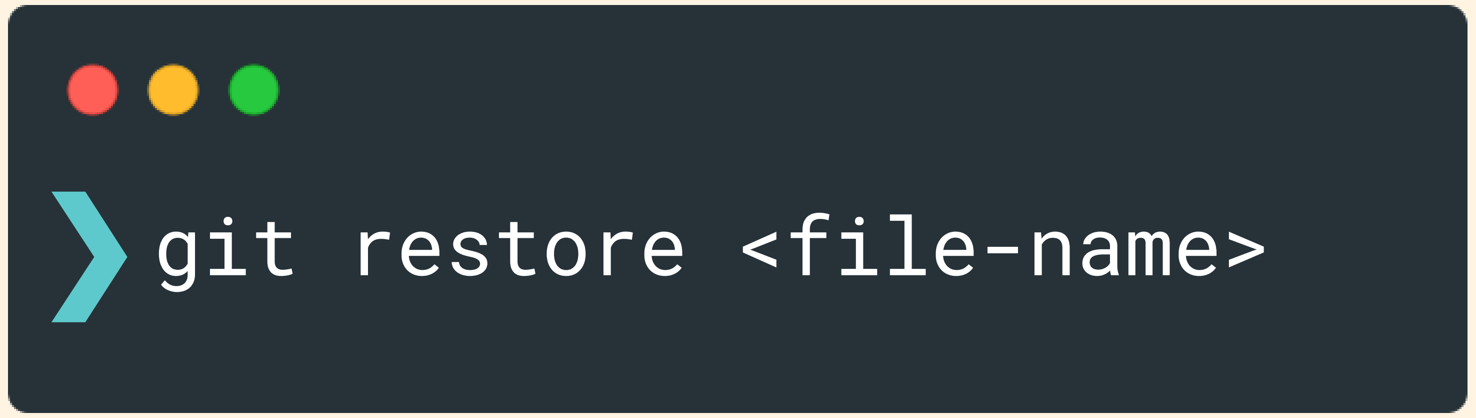
Because it is so new, most of the existing Git tutorials and books do not mention it, but it is worth knowing!

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.**

## 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>**



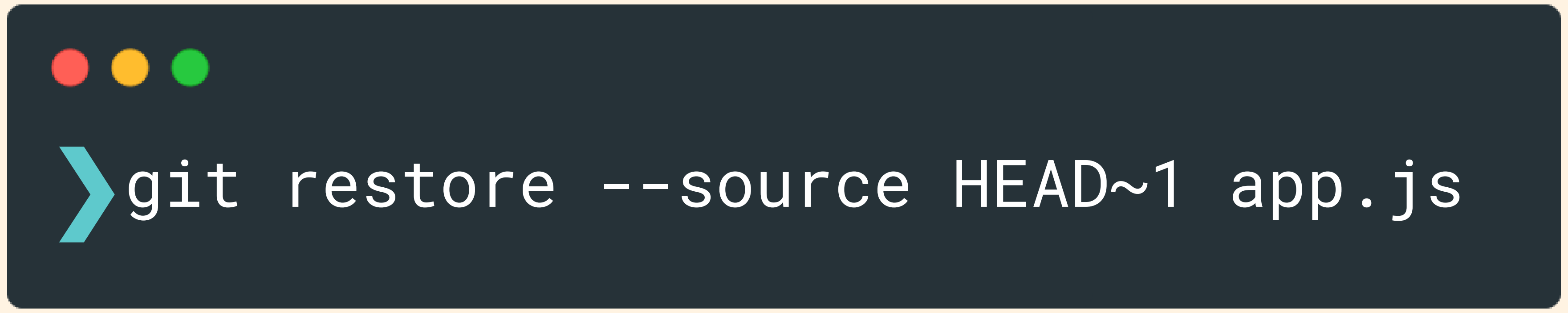
```
❯ git restore <file-name>
```

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

## 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.



```
❯git restore --source HEAD~1 app.js
```

## Unstaging Files with Restore

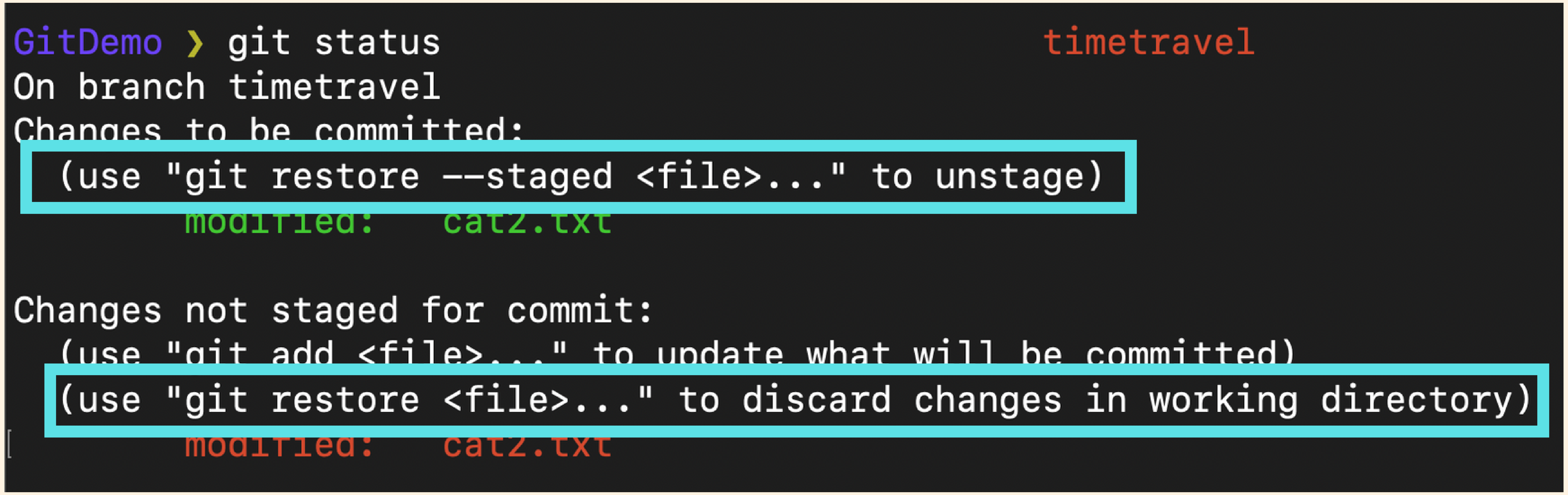
If you have accidentally added a file to your staging area with **git add** and you don't wish to include it in the next commit, you can use **git restore** to remove it from staging.

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

```
❯ git restore --staged <file-name>
```

# Feeling Confused?

**git status** reminds you what to use!



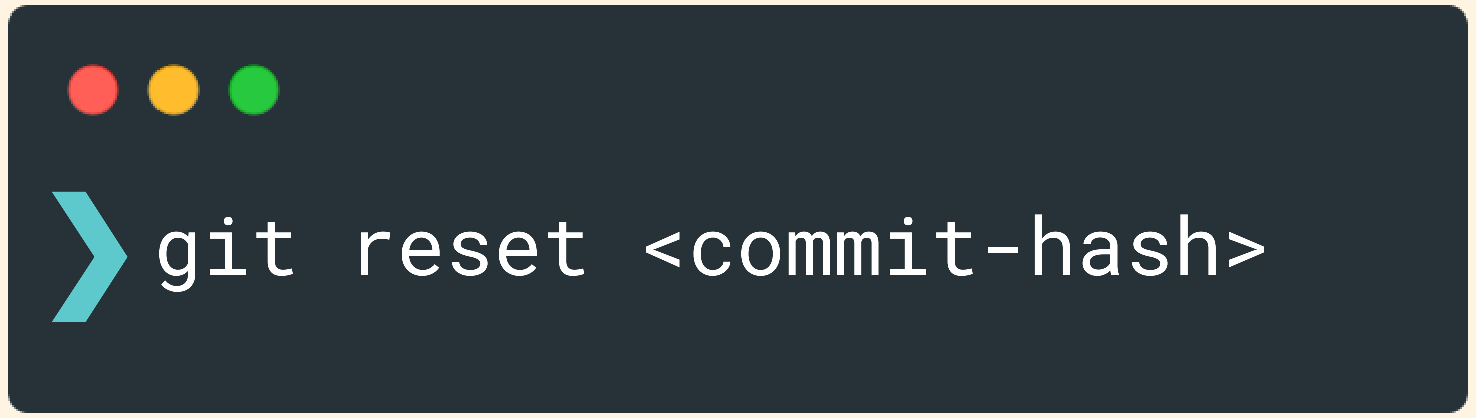
```
GitDemo ❯ git status                                          timetravel
On branch timetravel
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   cat2.txt

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
```

## Git Reset

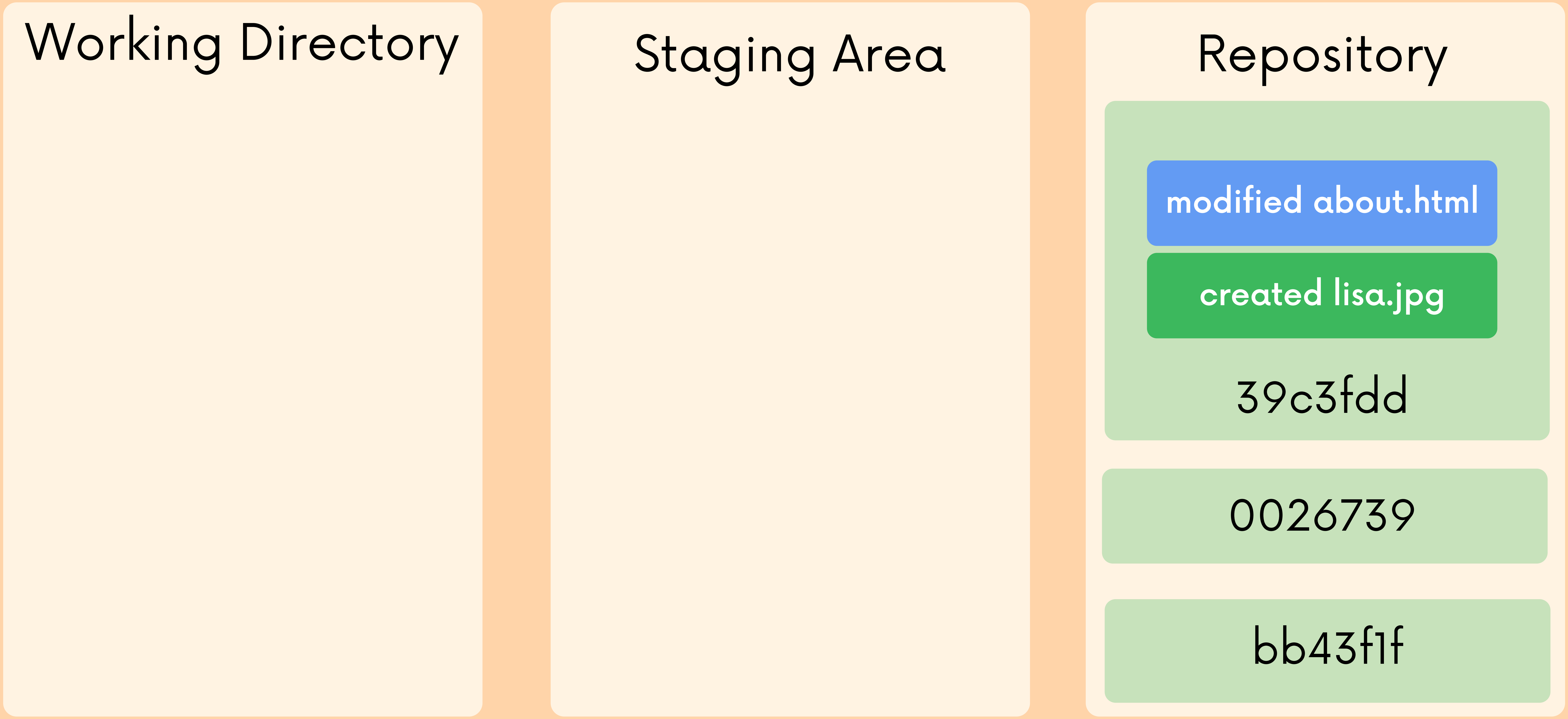
Suppose you've just made a couple of commits on the master branch, but you actually meant to make them on a separate branch instead. To undo those commits, you can use **git reset.**

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



```
❯ git reset <commit-hash>
```

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



```
❯ git reset 0026739
```

## Working Directory

modified about.html

created lisa.jpg

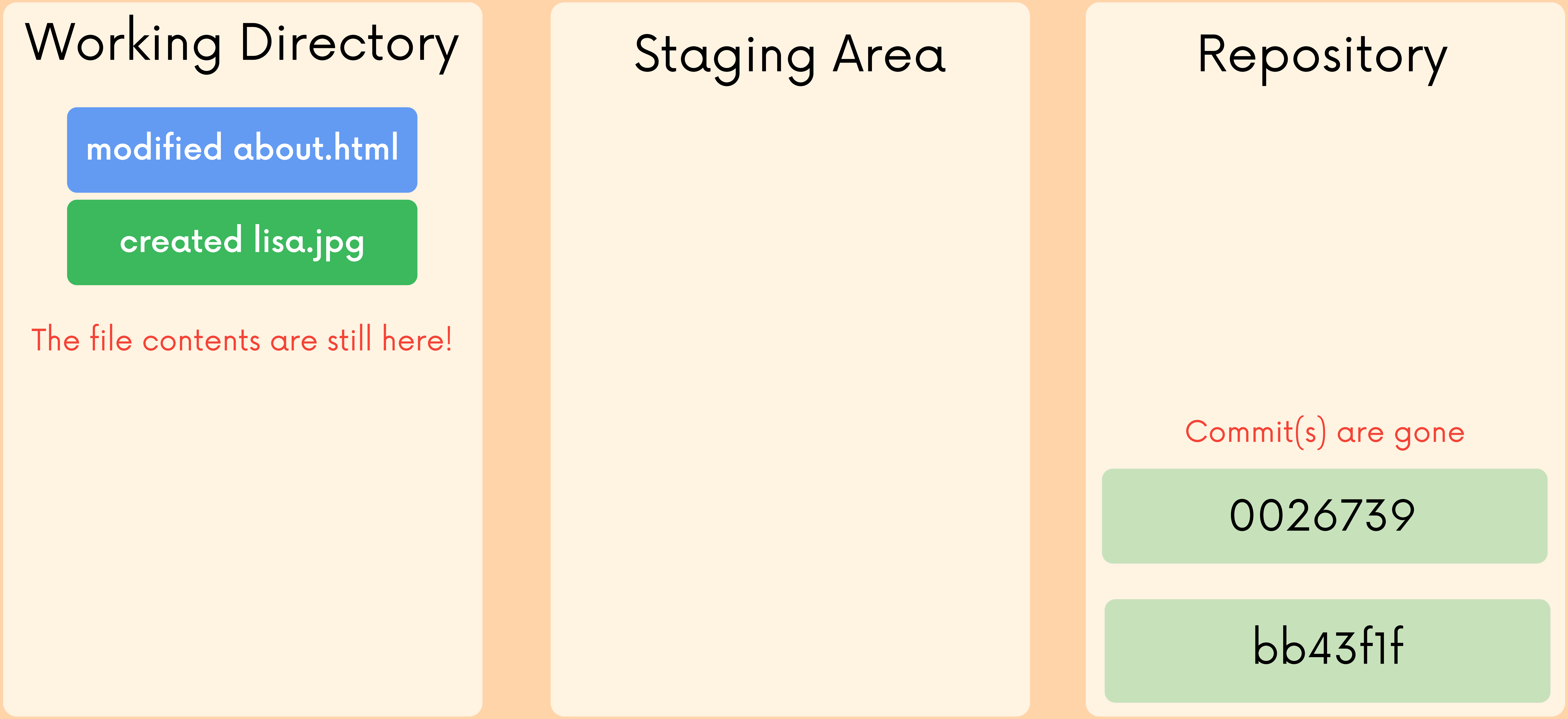
## Staging Area

## Repository

0026739

bb43f1f

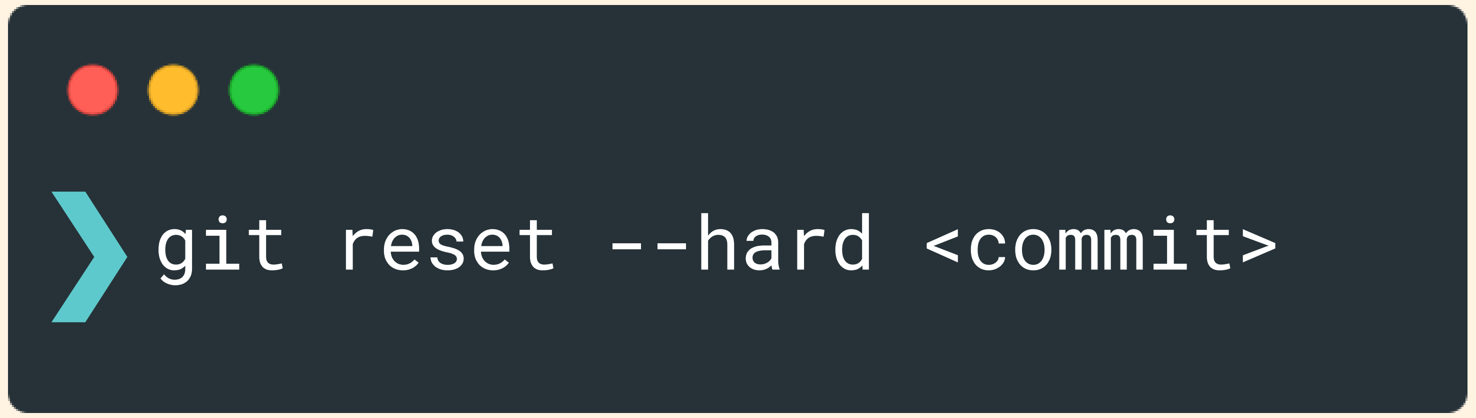
```
❯ git reset 0026739
```



## Reset --hard

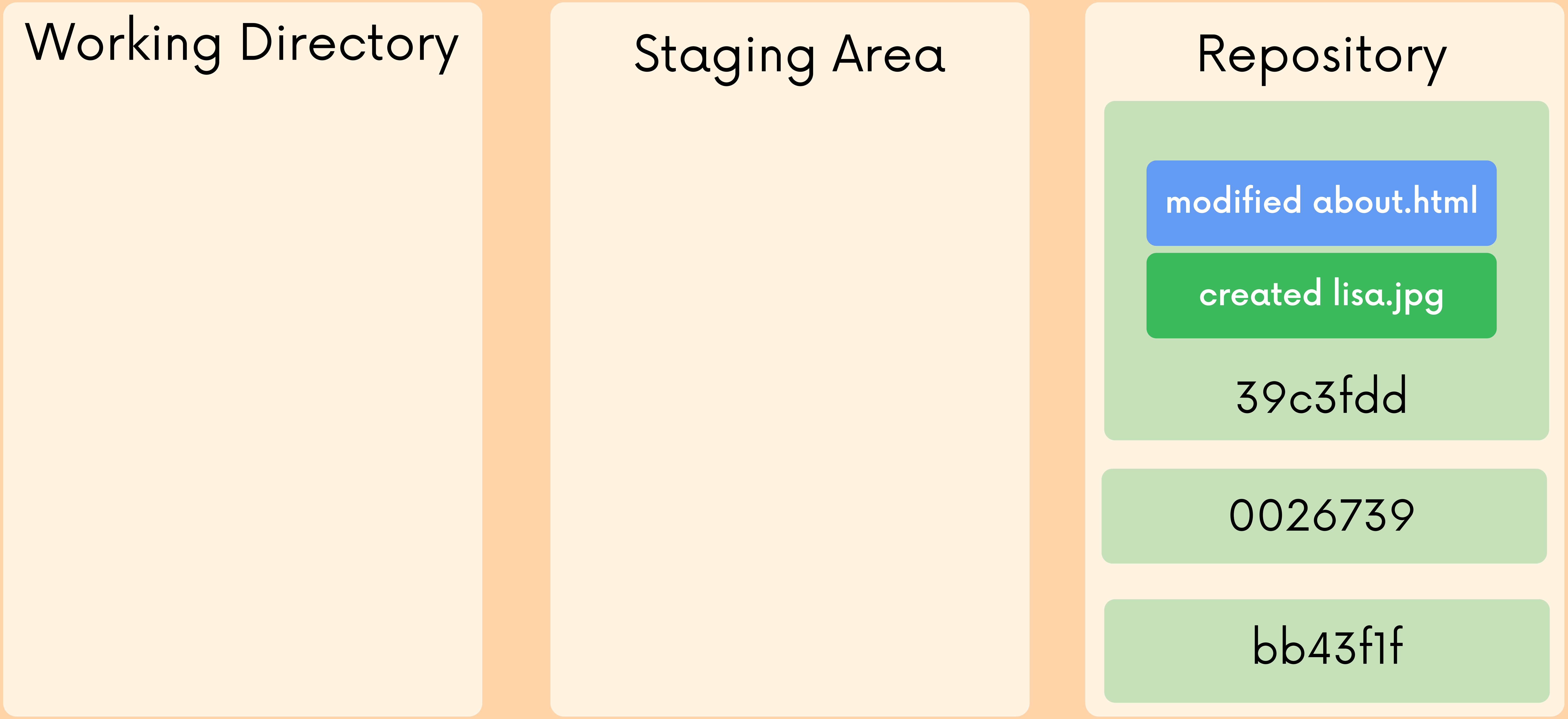
If you want to undo both the commits AND the actual changes in your files, you can use the **--hard** option.

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



```
git reset --hard <commit>
```

## OH NO! I don't want that commit OR the changes



```
❯ git reset --hard 0026739
```

## Working Directory

The changes in the file(s) are gone too!

## Staging Area

## Repository

Commit(s) are gone

0026739

bb43f1f

## git revert

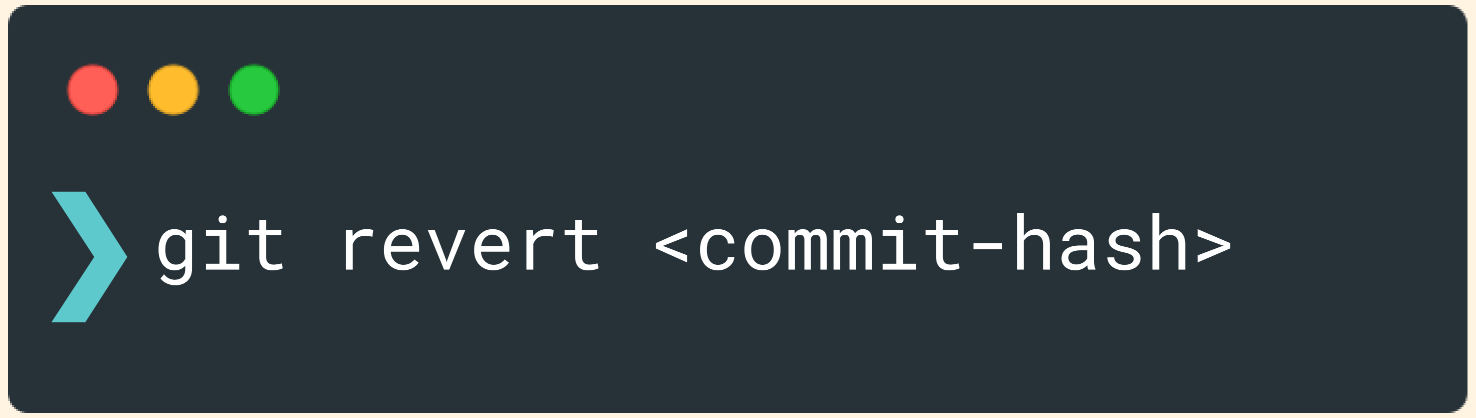
Yet another similar sounding and confusing command that has to do with undoing changes.

## Git Revert

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

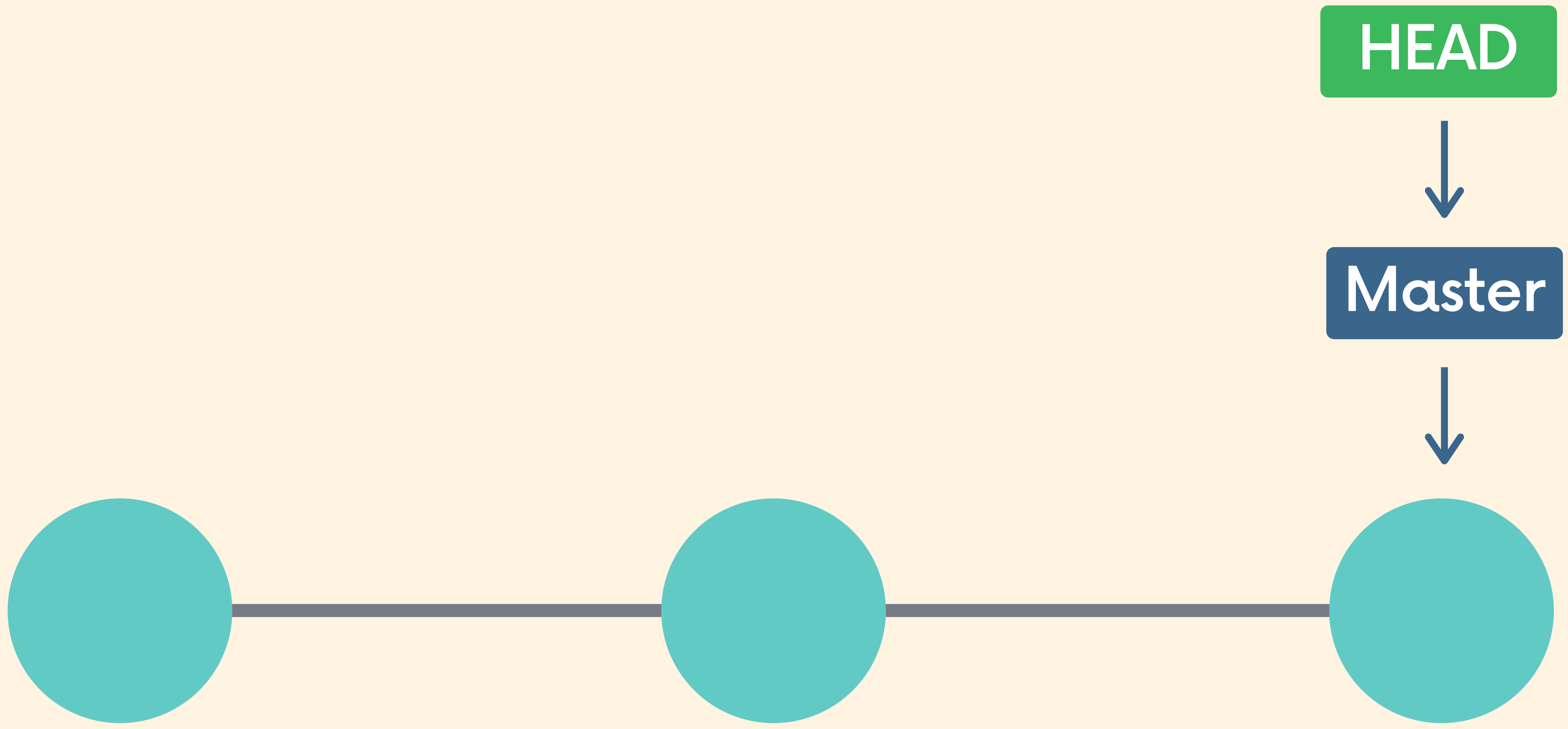
**git reset** actually moves the branch pointer backwards, eliminating commits.

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

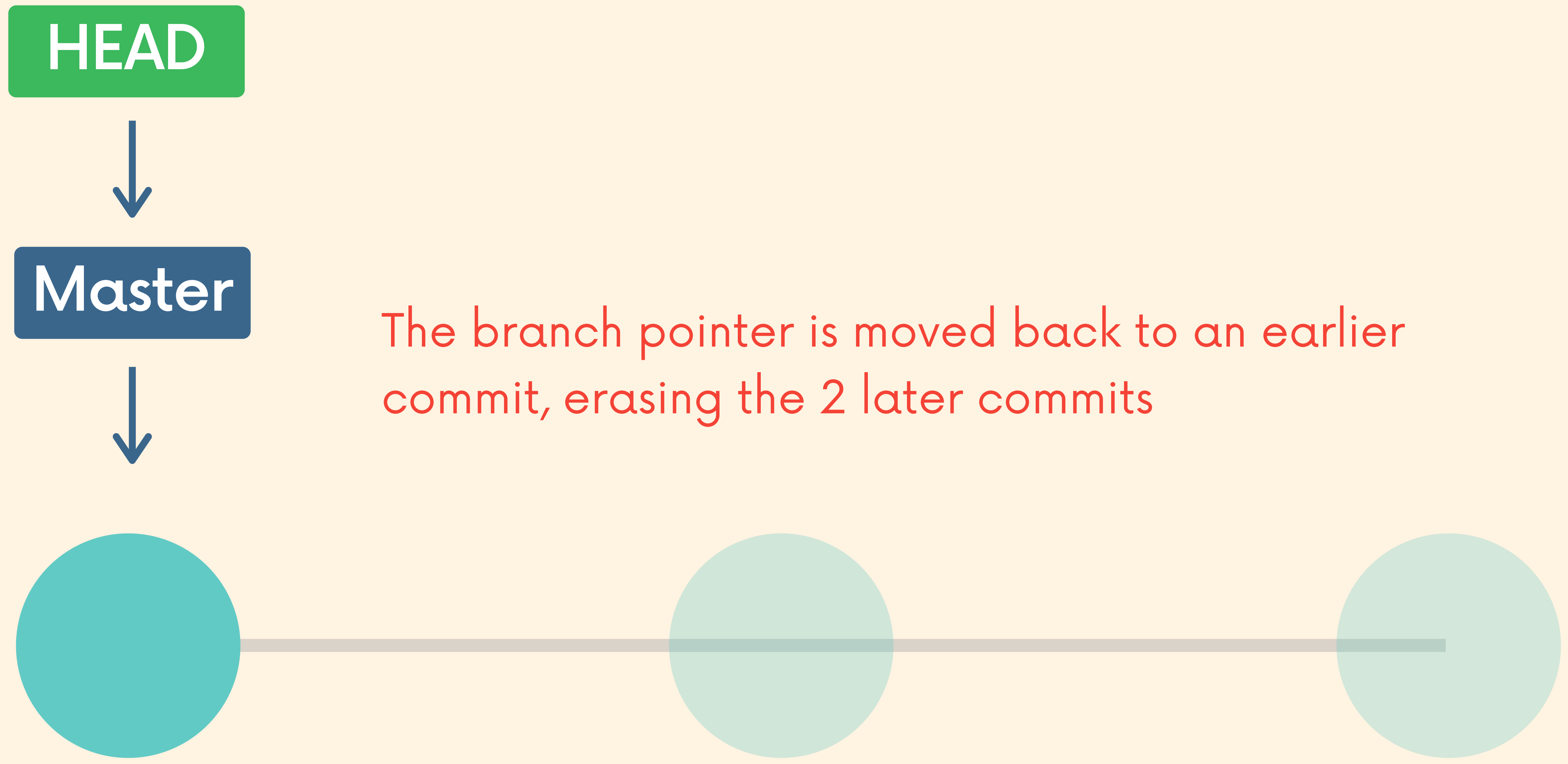


```
git revert <commit-hash>
```

## "Undoing" With Reset

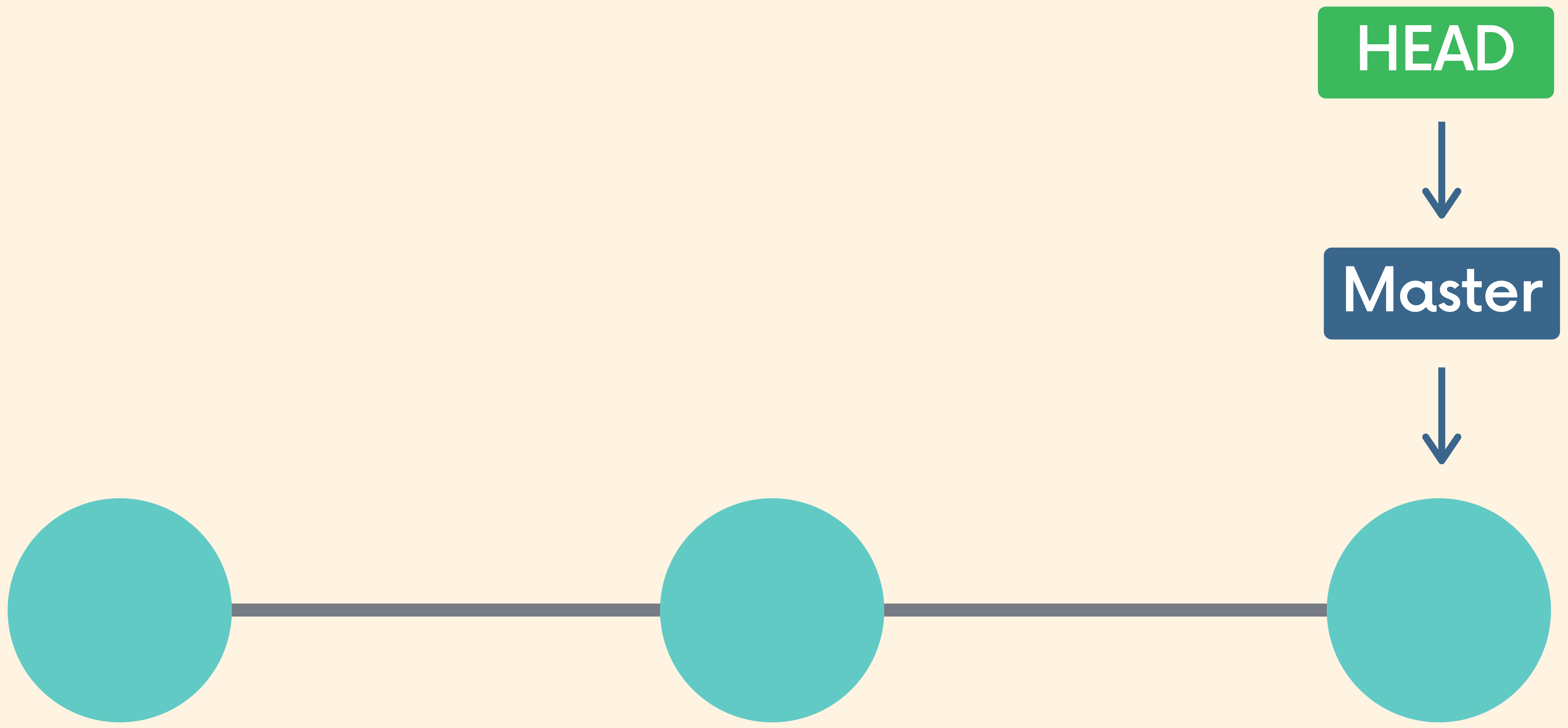


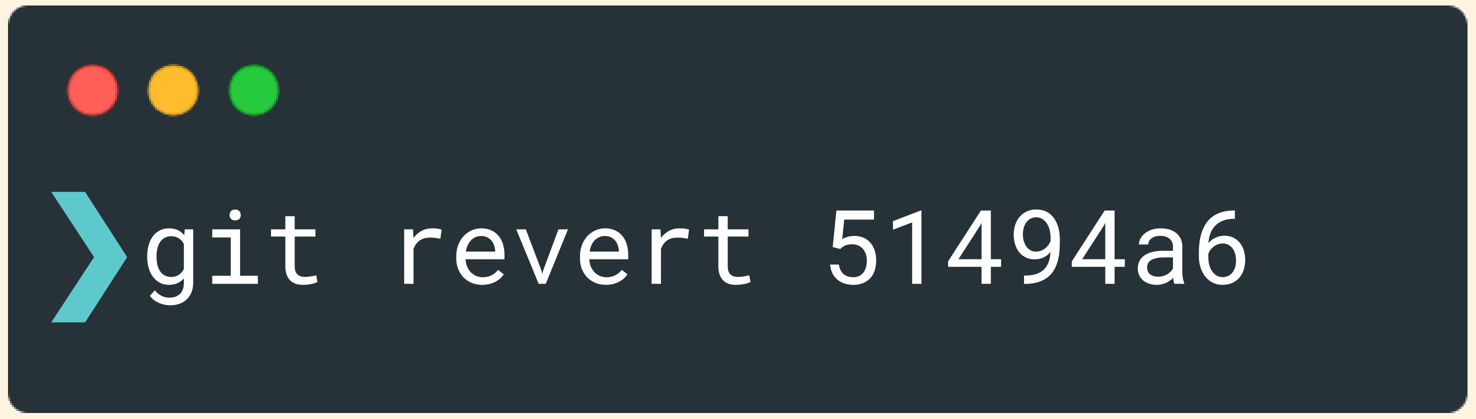
```
❯git reset HEAD~2
```



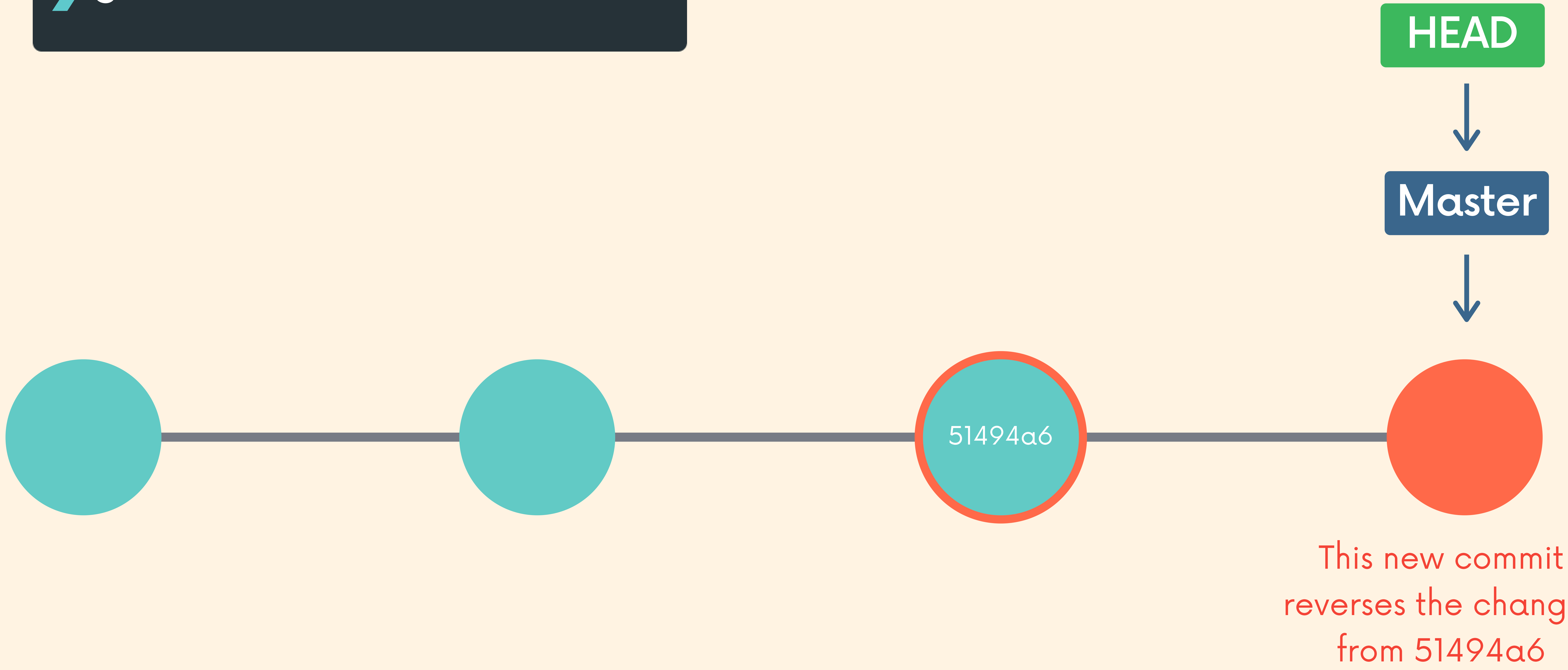
The branch pointer is moved back to an earlier commit, erasing the 2 later commits

## "Undoing" With Revert





```
❯git revert 51494a6
```

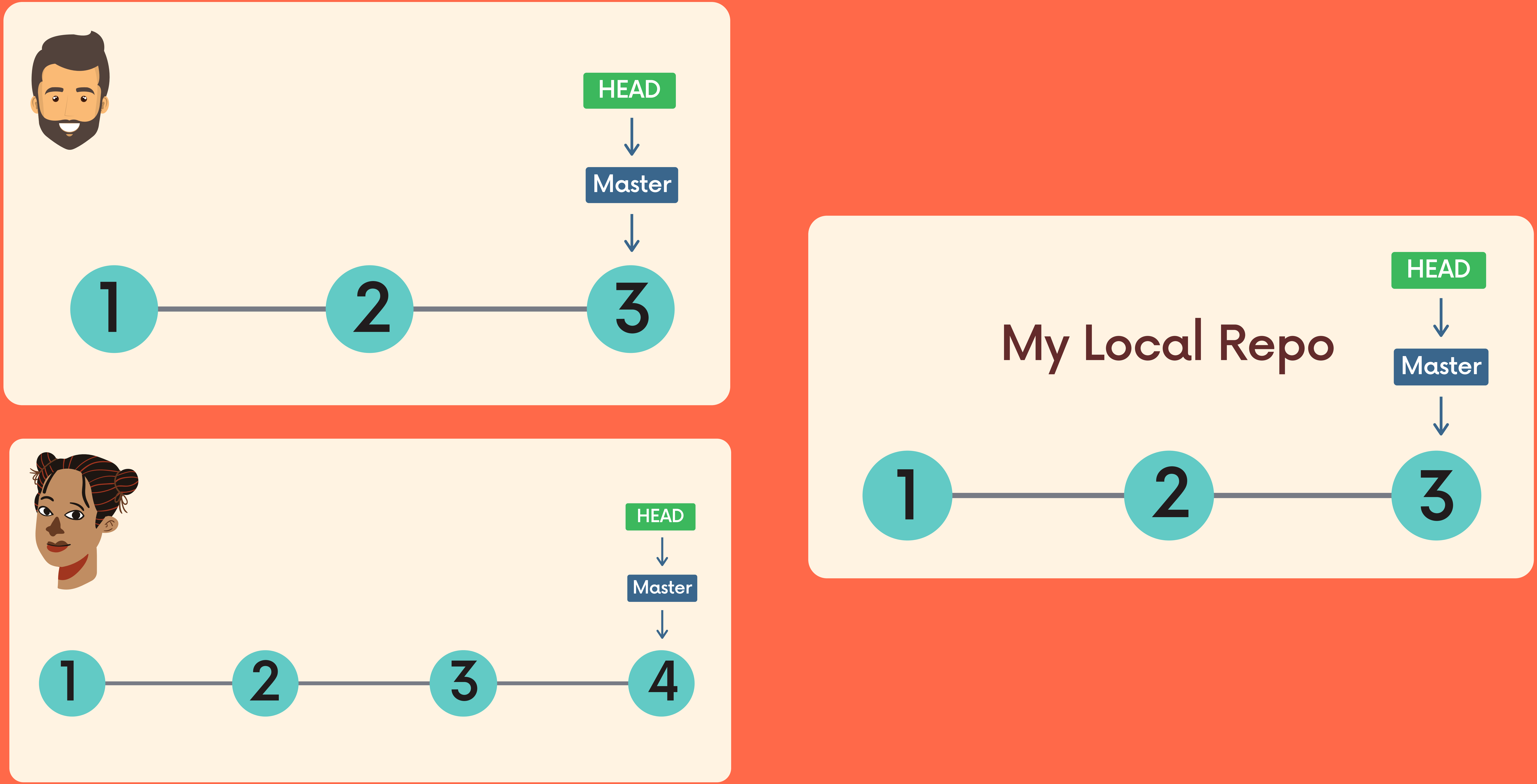


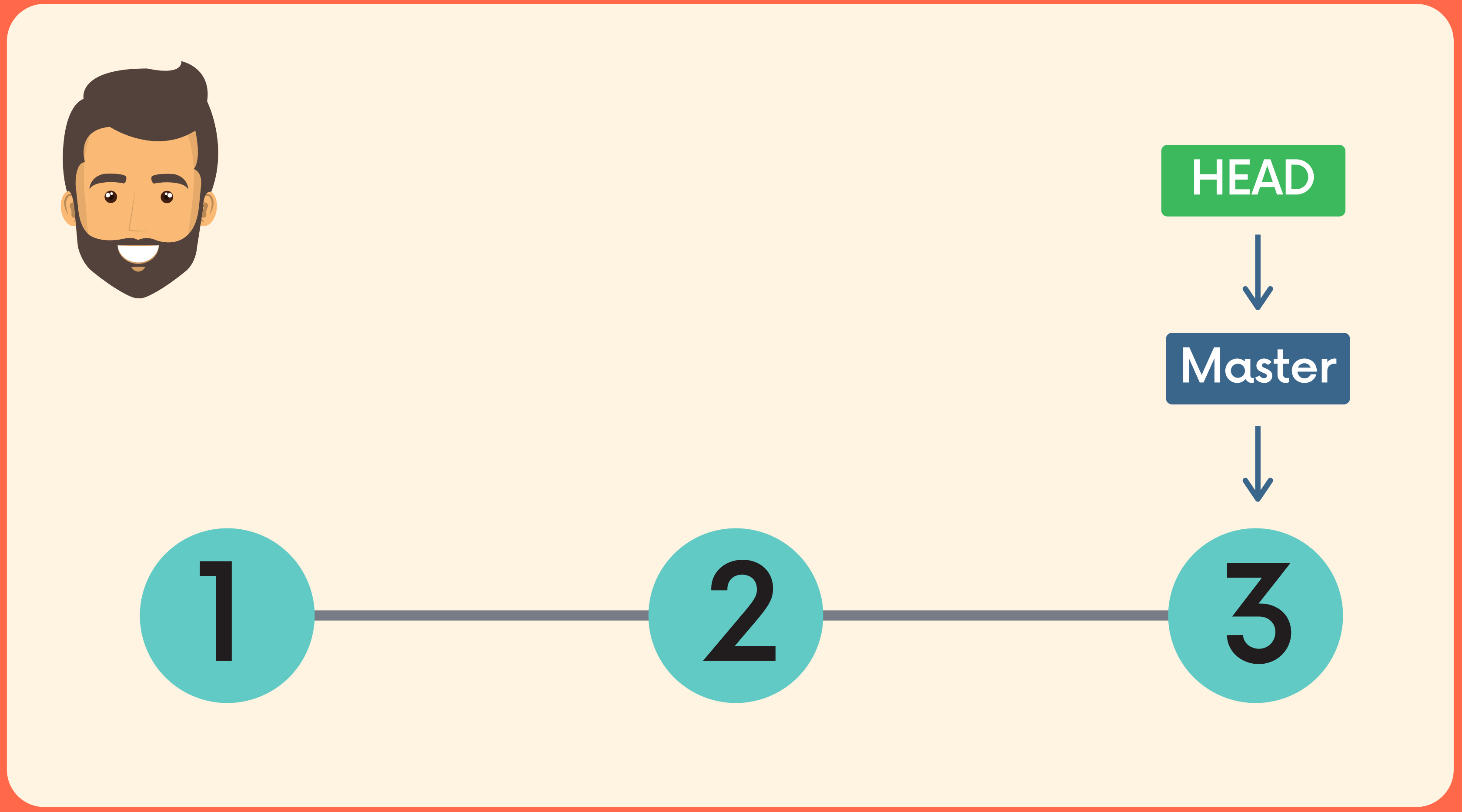
# Which One Should I Use?

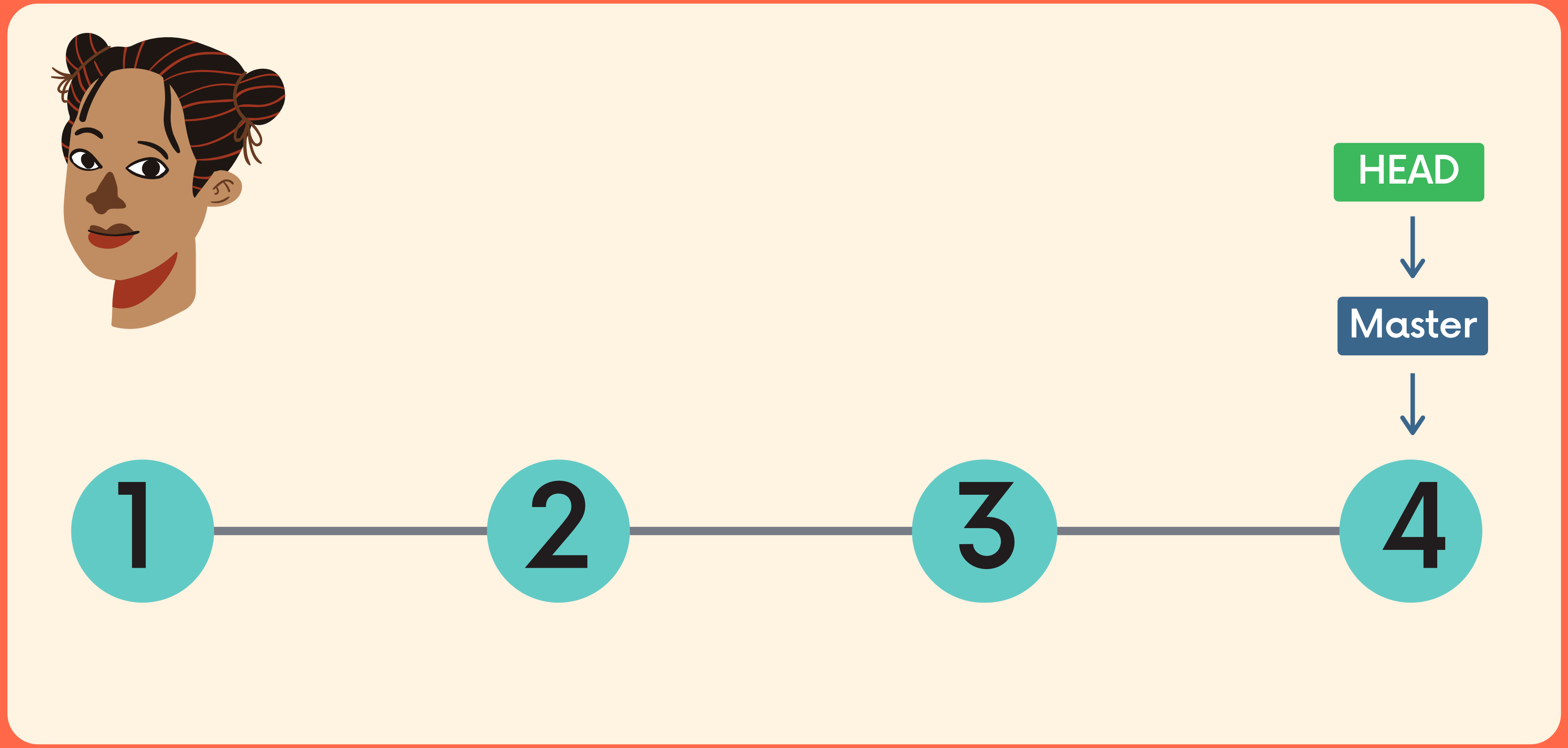
Both **git reset** and **git revert** help us reverse changes, but there is a significant difference when it comes to collaboration (which we have yet to discuss but is coming up soon!)

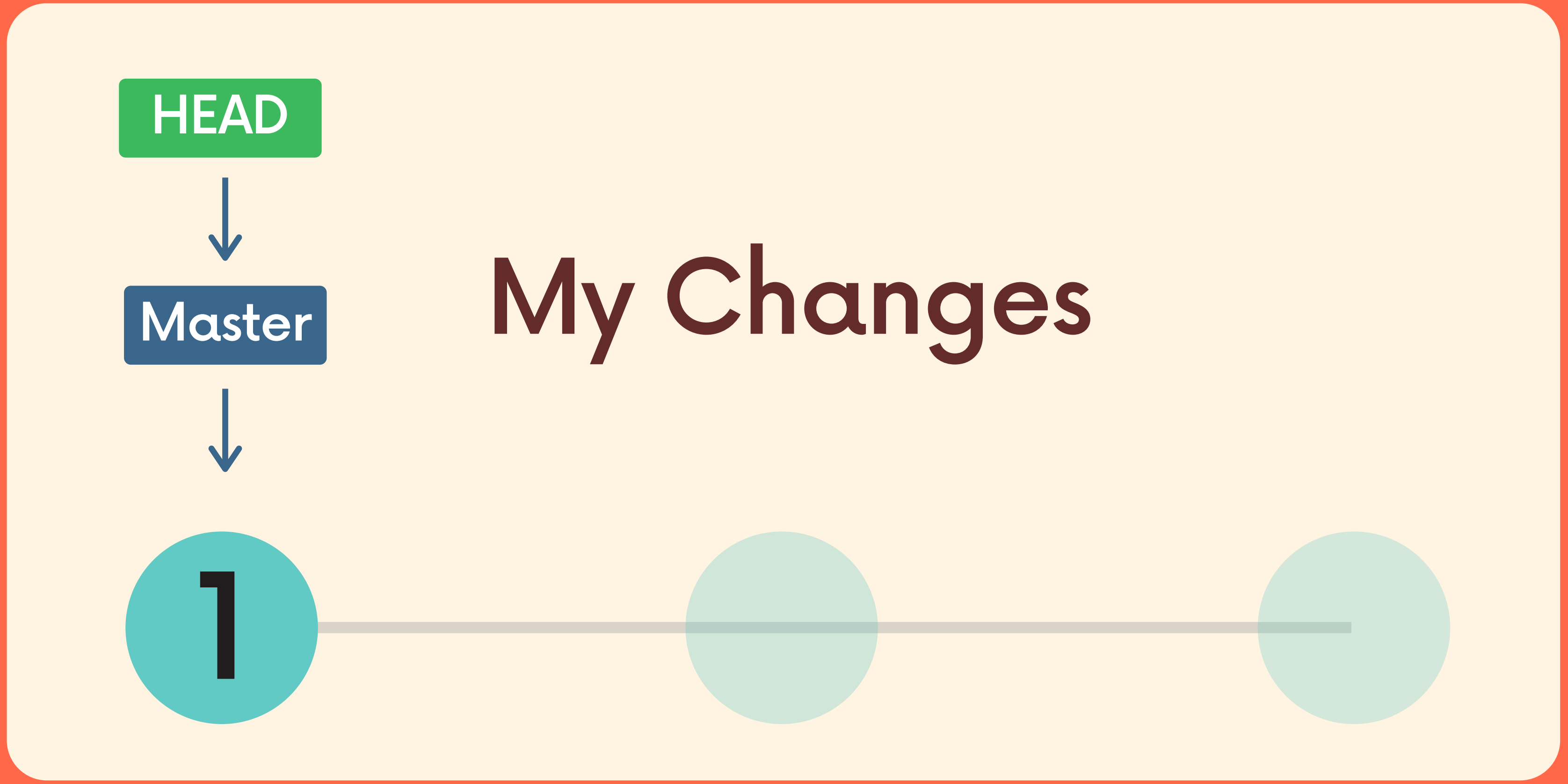
**If you want to reverse some commits that other people already have on their machines, you should use revert.**

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

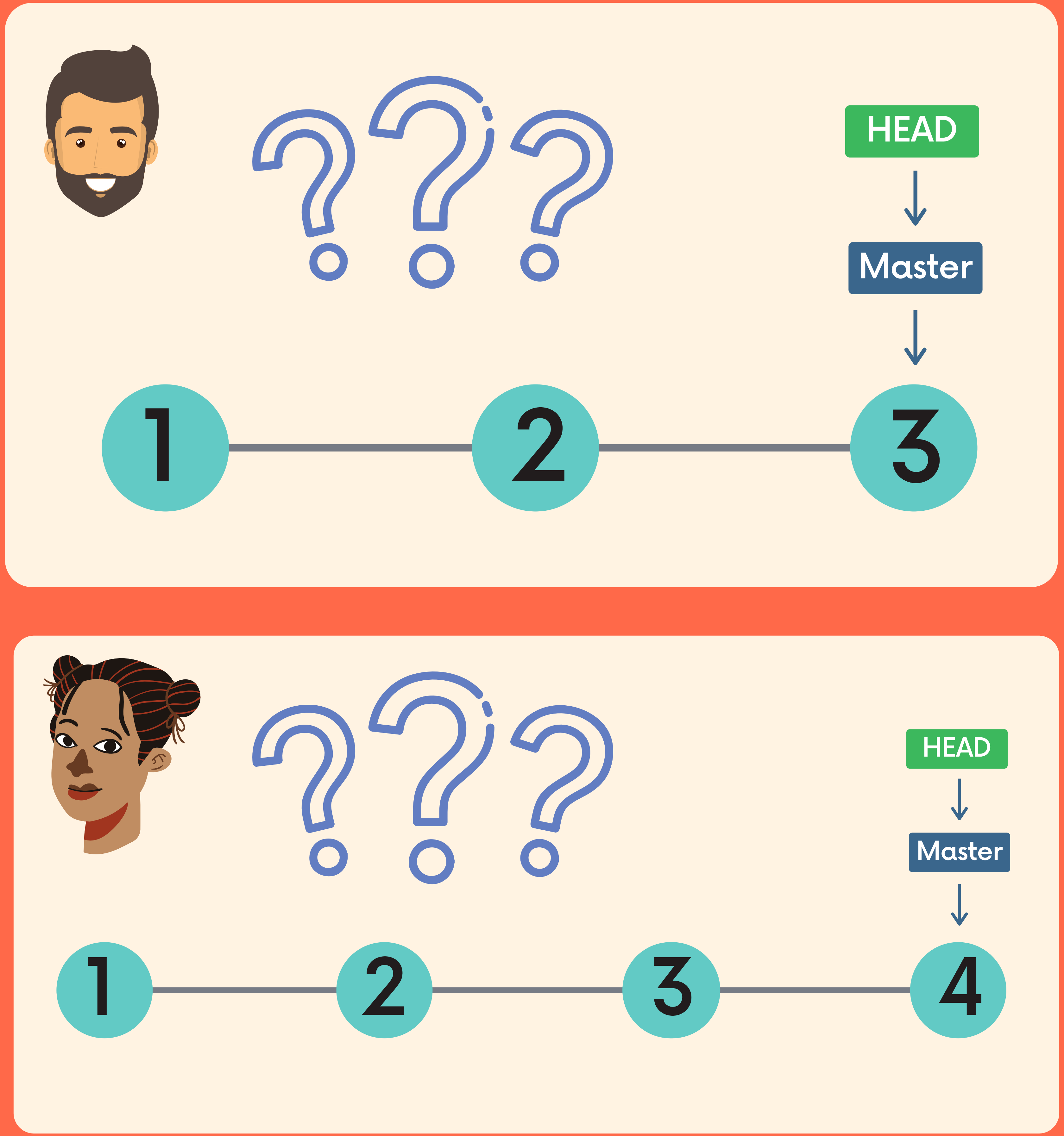

HEAD
Master
1
2
3
HEAD
Master
1
2
3
4
My Local Repo
HEAD
Master
1
2
3

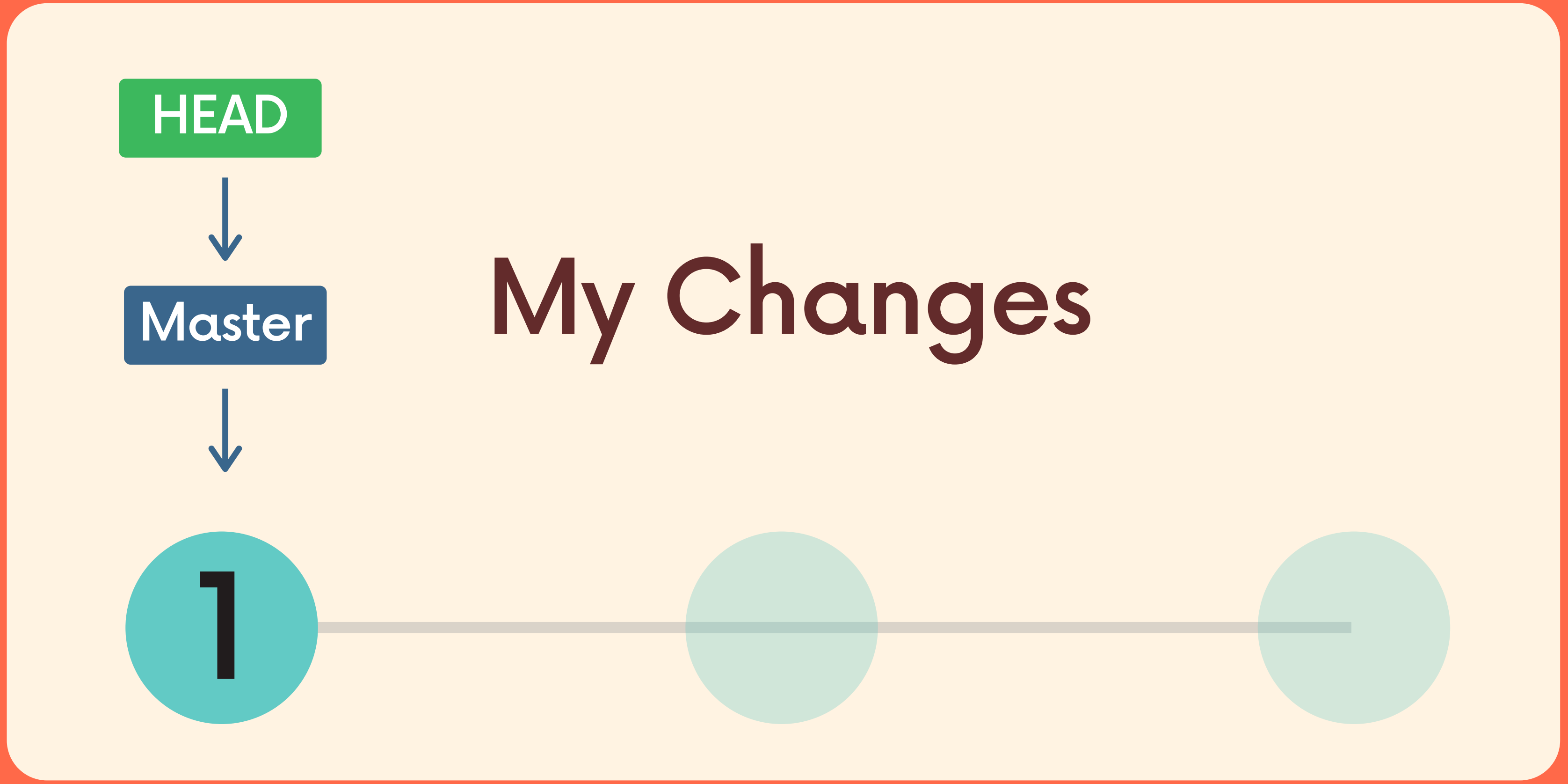




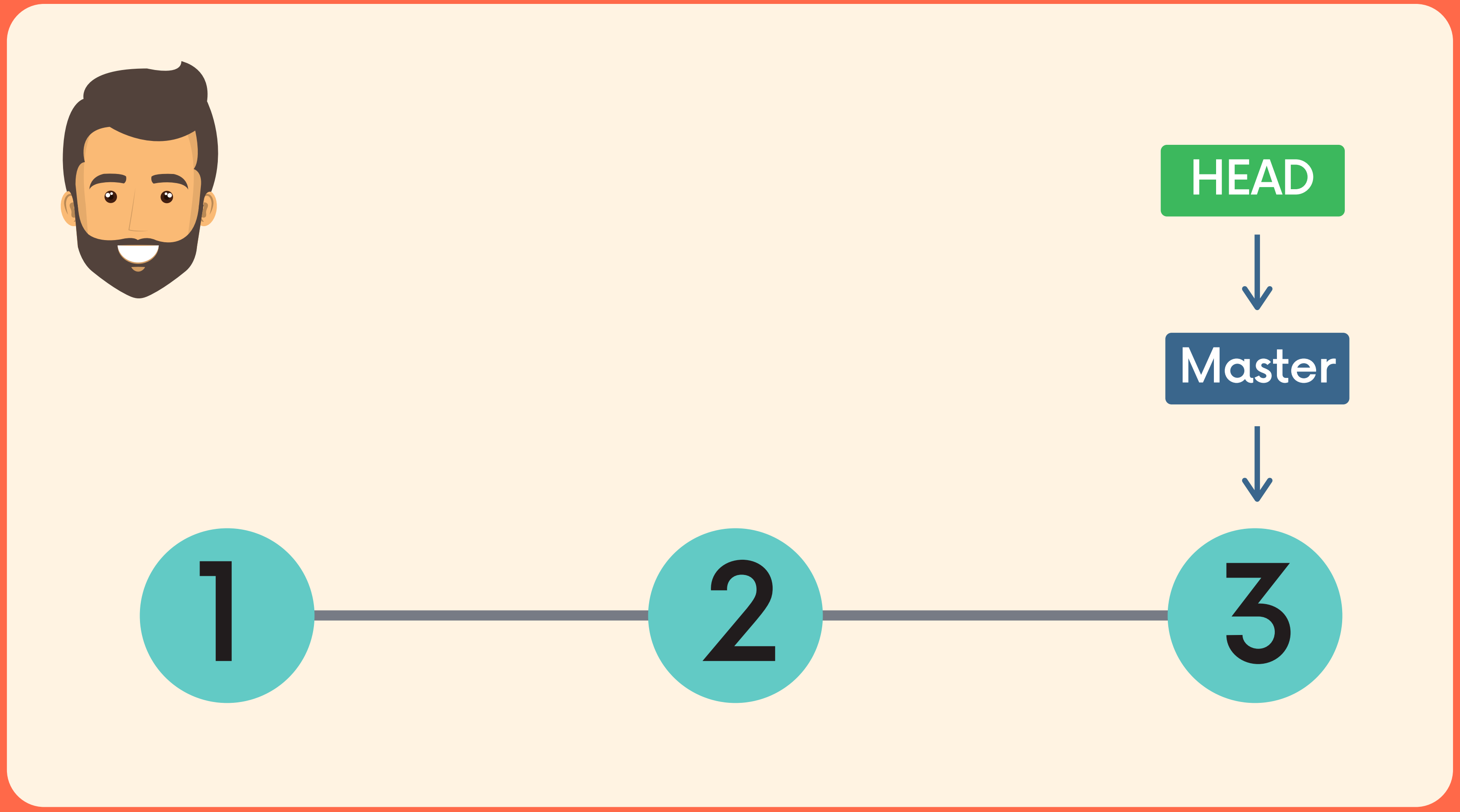


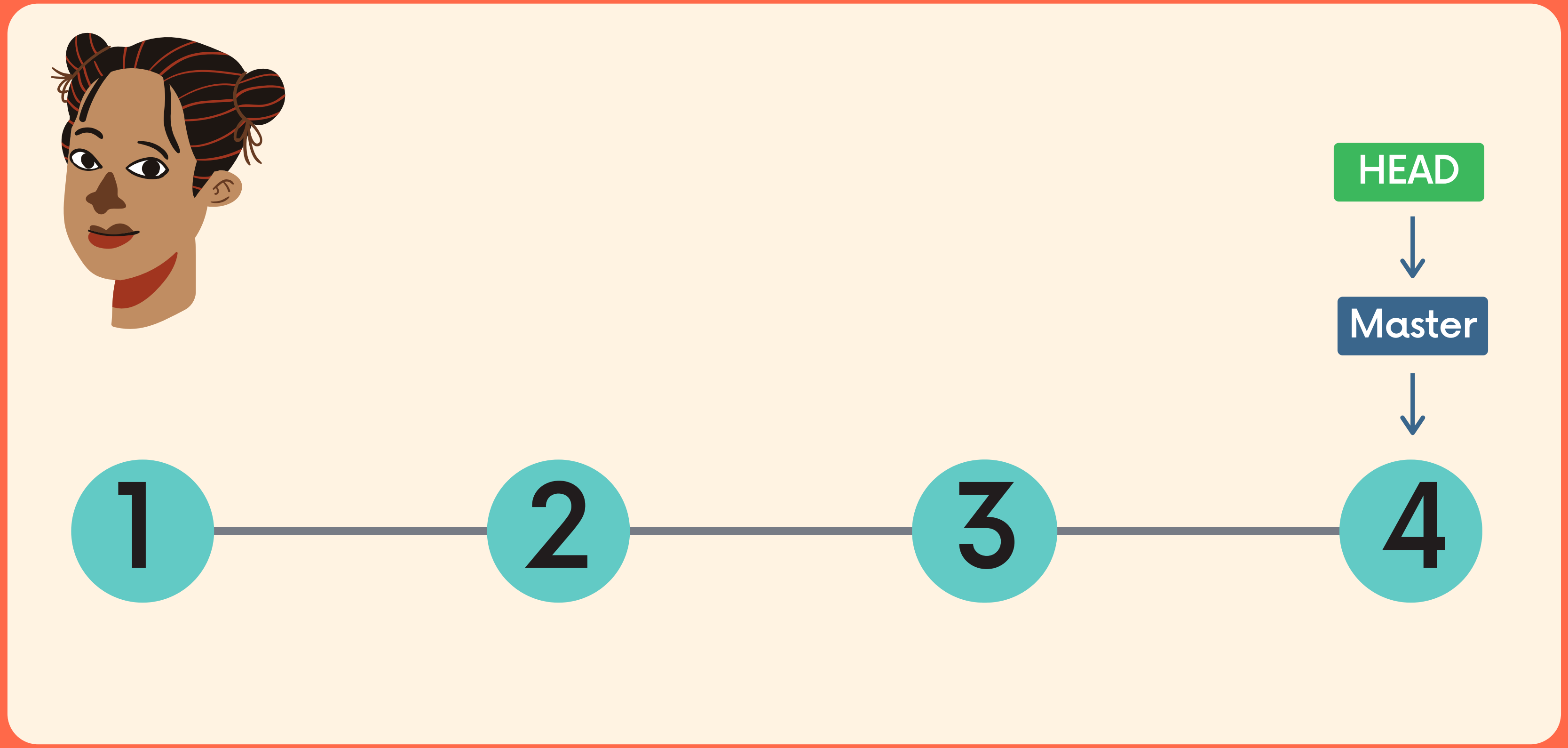
I use git reset to remove commits that I already shared with my team!

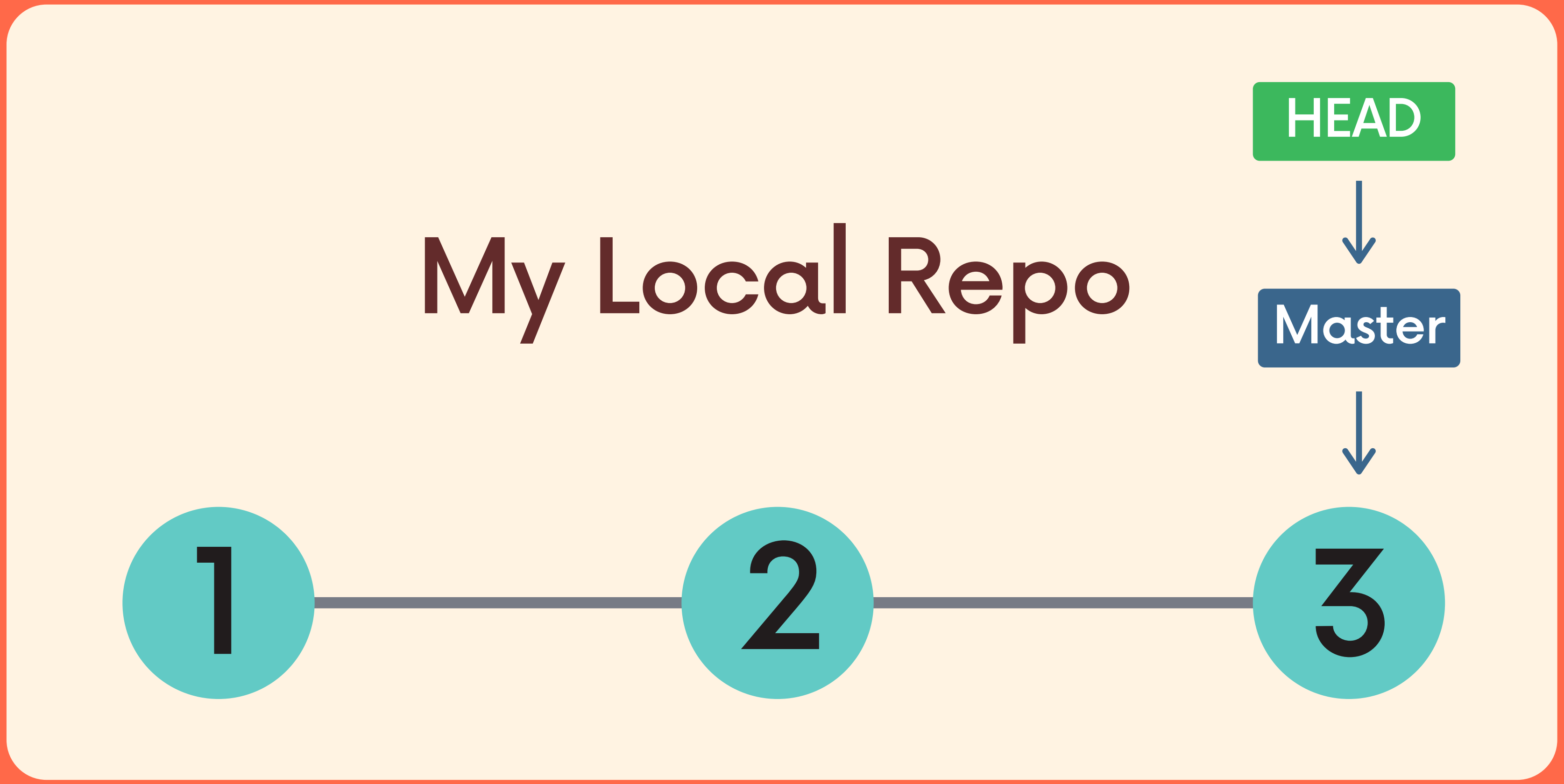




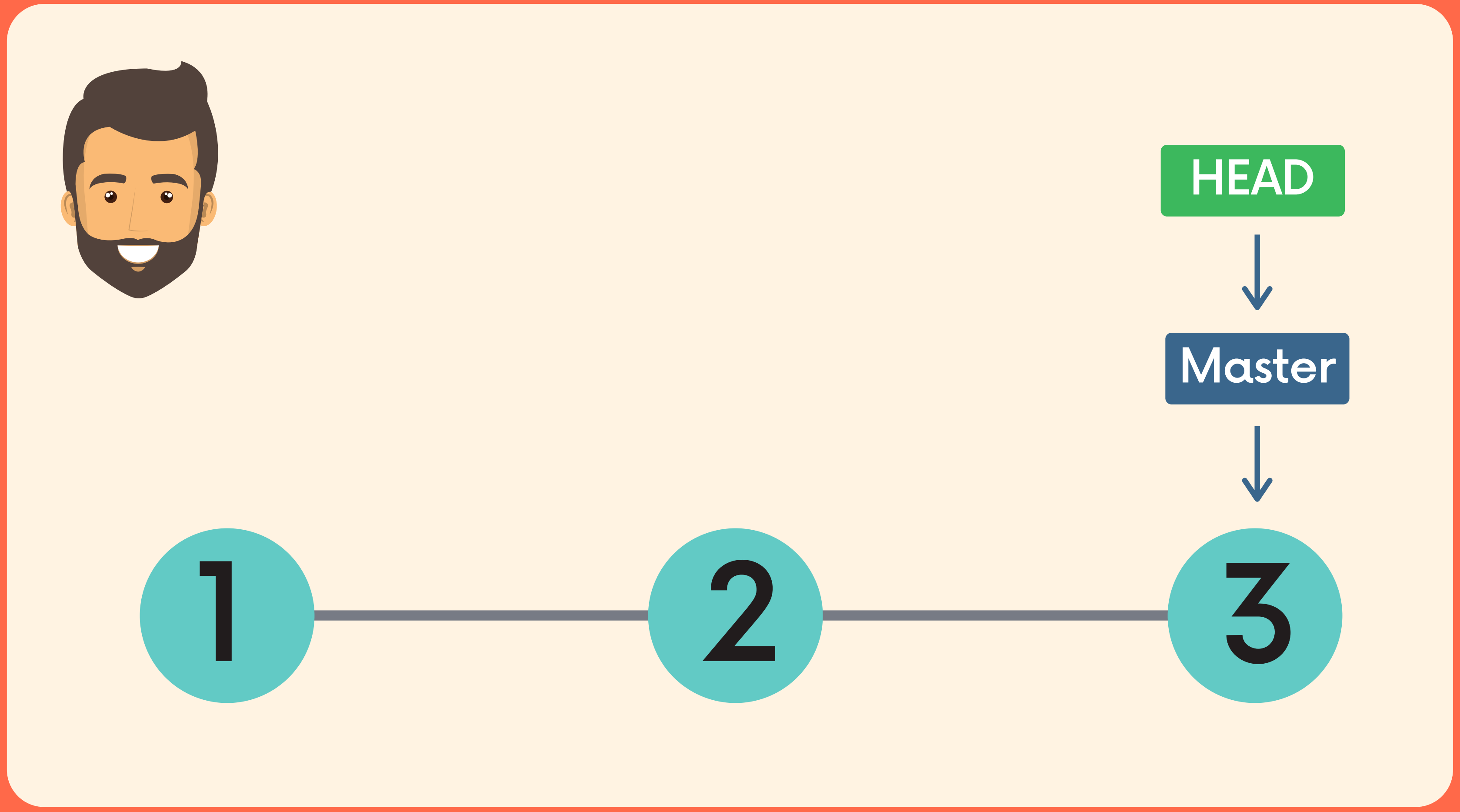
This makes their lives harder. I altered history that they already have. BAD!

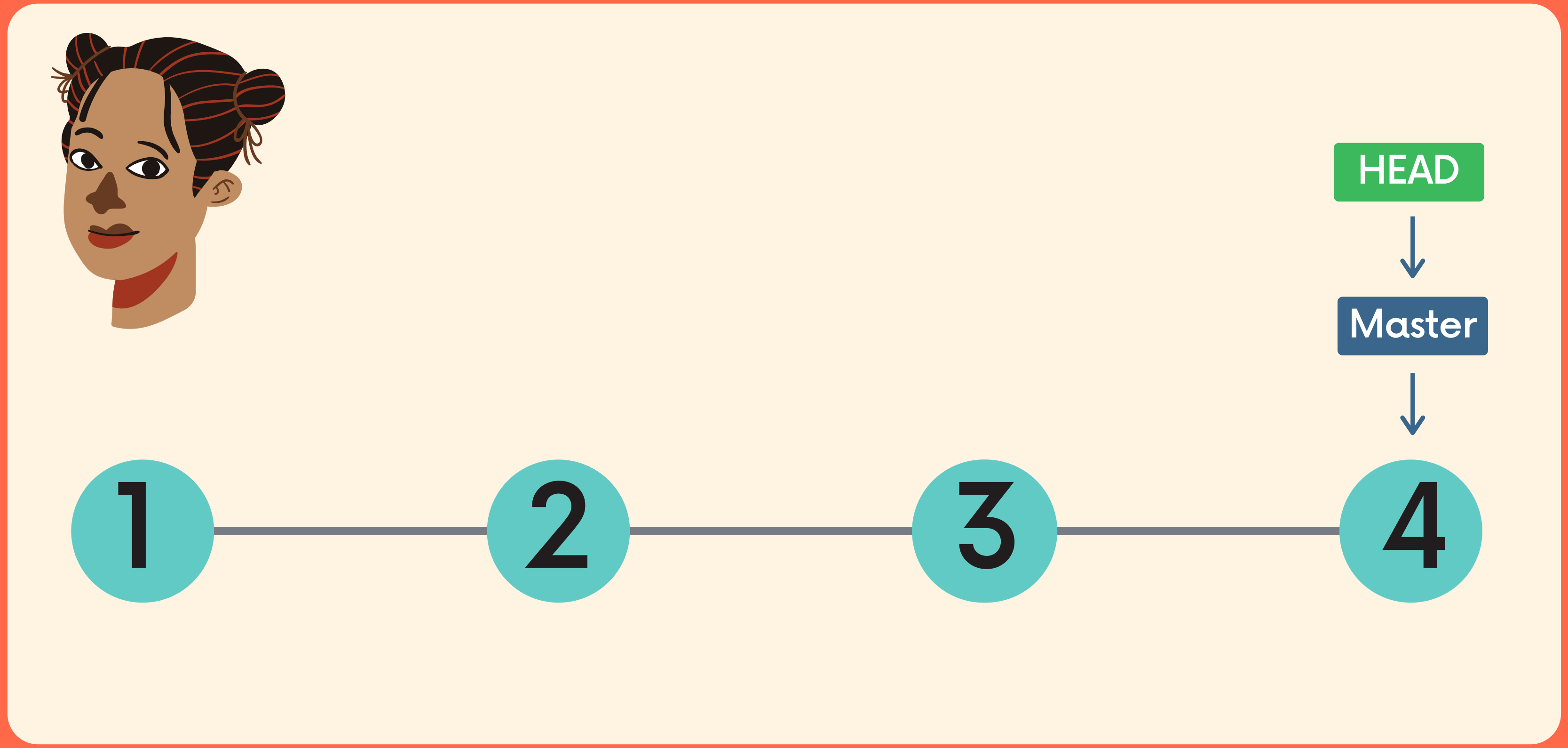


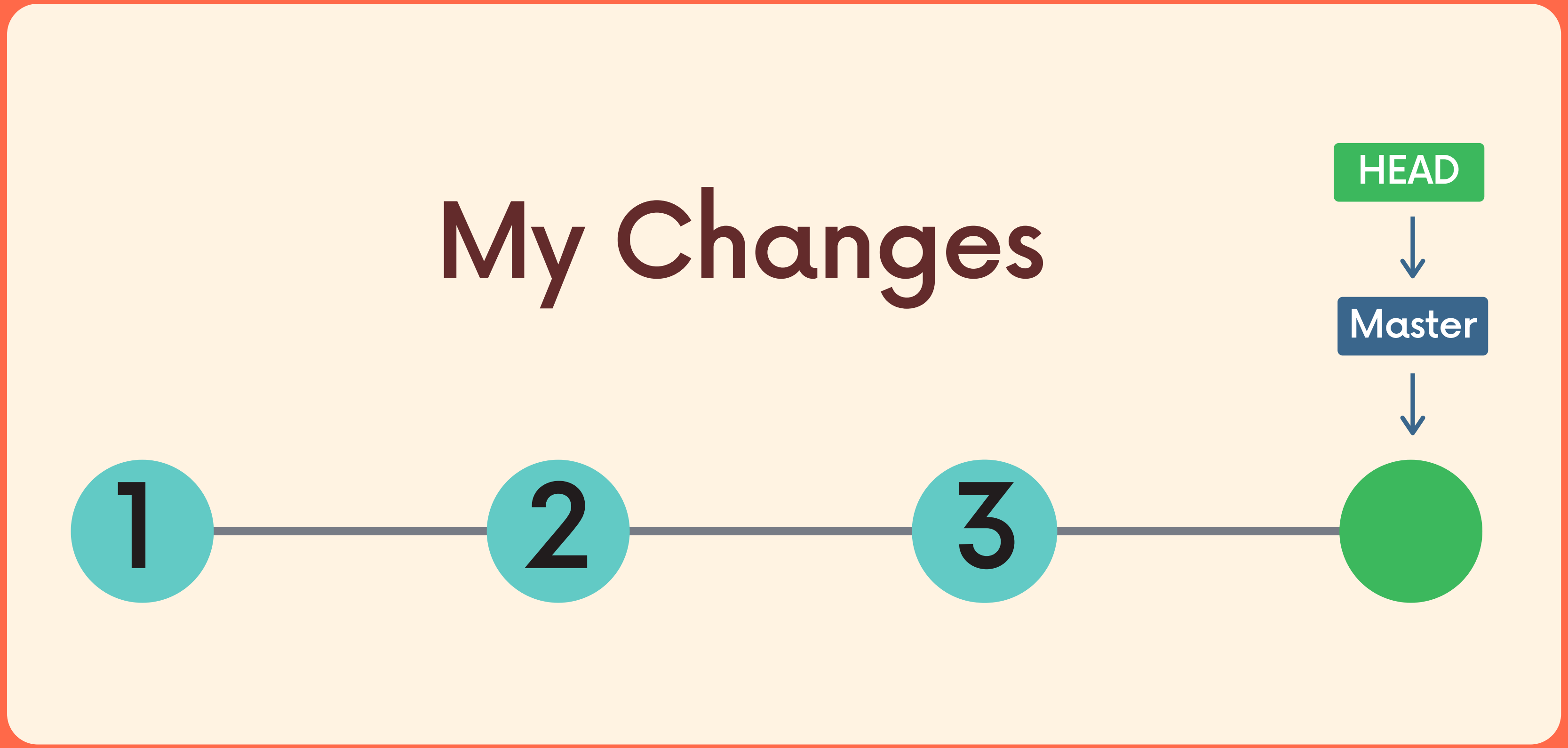




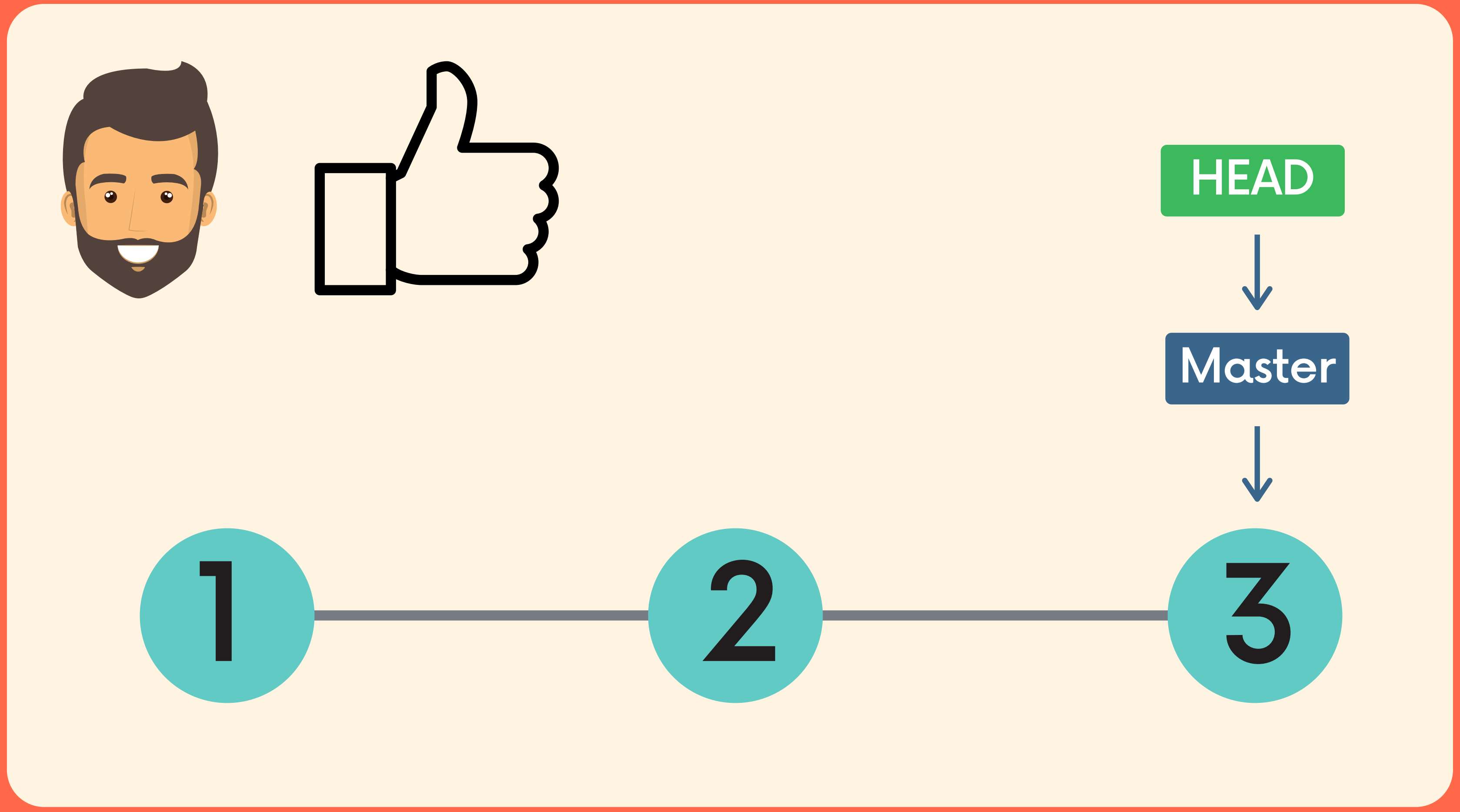
Let's try again...

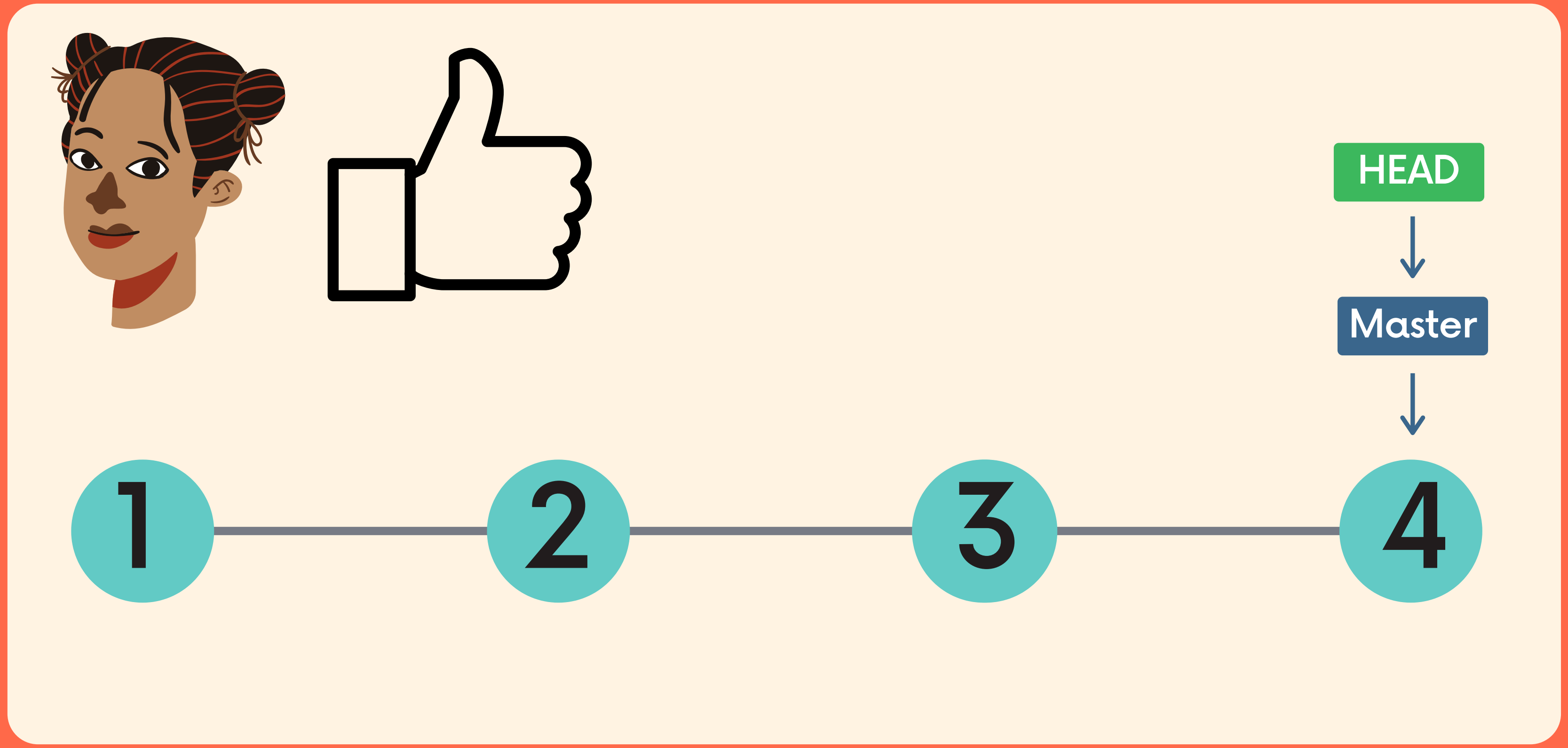


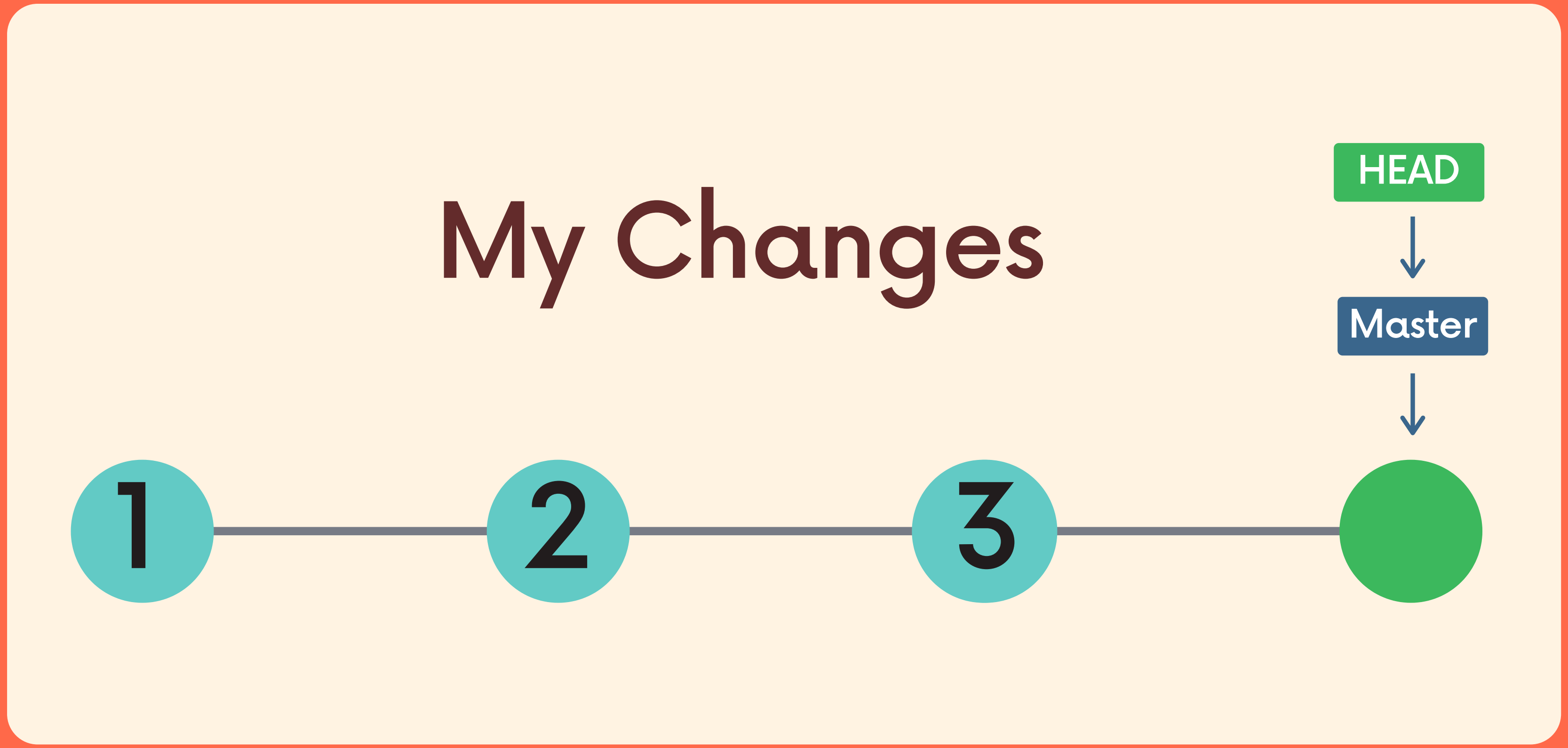




I use **git revert** to reverse the same commits as before, by ADDING a new commit to the chain







My team can merge in the new "undo" commit without issue.
I didn't alter history.