**Introduction:**

We will look at various ways of browsing history of our project.

🡪 Search for commits (*by author, date, message, content etc.*)

🡪 View a commit (*to see what files have been changed and how*)

🡪 Restore your project to an earlier point in time (*to the snapshot stored in one of our commits*).

🡪 Compare commits (*to see what files have been changed and how*)

🡪 View a history of file (*to see how it has evolved from day one and who has written each line of code*)

🡪 Find a bad commit that introduced a bug (*using a tool*).

**Getting a repository**:

Download the *venus.zip* sample repository.

**Viewing the history**:

In the previous section we learned about ***git log*** command. In this section we will learn it in more detail.

$ git log



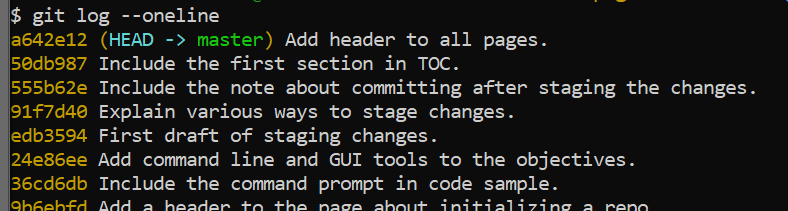
Here we can see all the commits in this repository. For each commit there is a unique identifier or *hash*. As well as *author name*, *email*, *date* and *time* it was created and a one liner *message* or *subject* of the commit.

By default they are ordered from newest to oldest.

Some useful arguments in this command are:

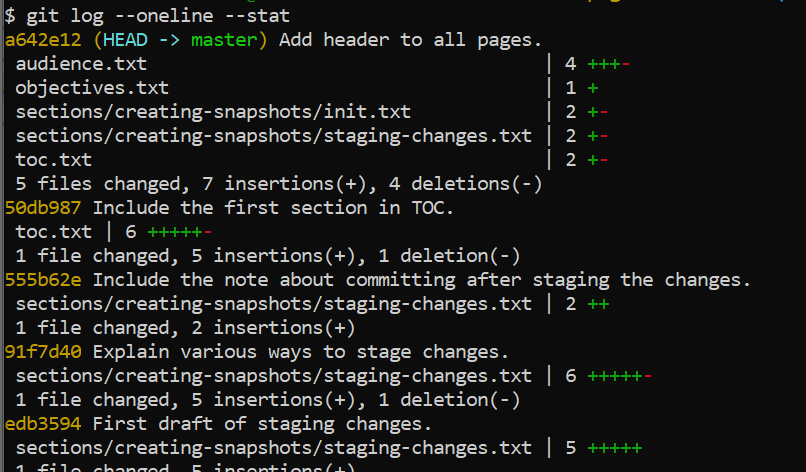
1. To get the *summary of all commits*.

$ git log --oneline



1. To *see all the files that have been changed in each commit*.

$ git log --oneline --stat



As you can see in our last commit, 5 files have been changed.

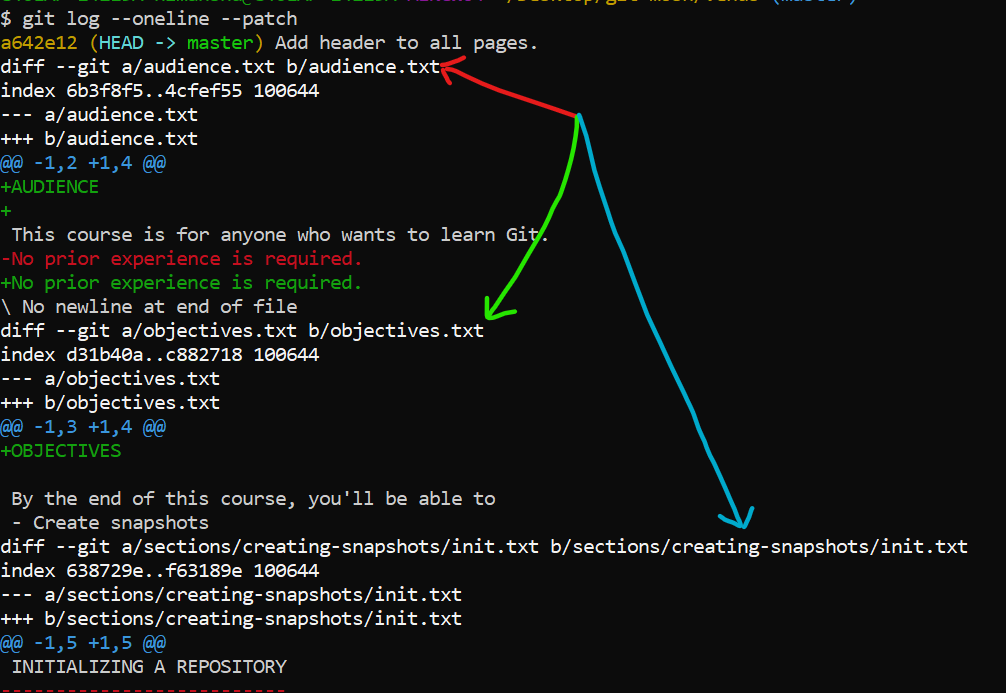
Also notice that in front of each file name, we can see the number of changes. For example in *audience.txt* file we have 4 changes (*3 insertions and 1 deletion*).

Finally we see the *summary* below each commit, for example

*5 files changed, 7 insertions (+), 4 deletions (-)*

1. To *see actual changes done in a commit*,

$ git log --oneline --patch



In each commit we can see *diffs* of each file so we can view changes done in each file.

Note: *patch* and *stat* options can be directly applied *without using oneline* option.

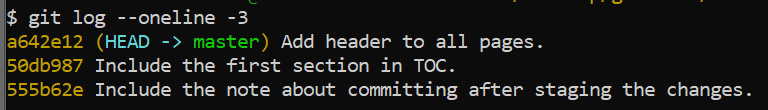
**Filtering the history**:

In this repository, we have just over 10 commits. But in real repository we can have hundreds or even thousands depending on age of repository.

Now we do not need to see all the commits so there are a few ways to filter the history.

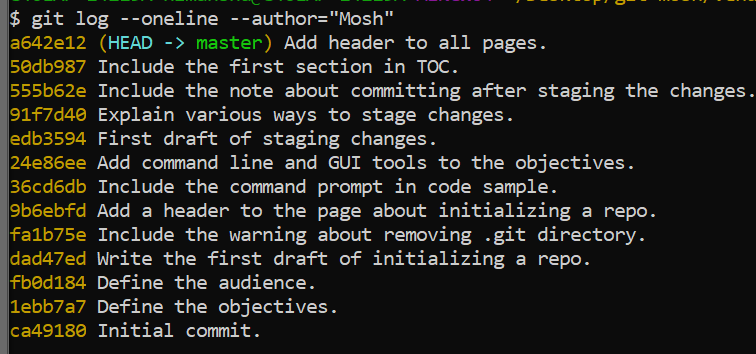
1. To see last 3 commits:

$ git log --oneline -3



1. Filter commits by author:

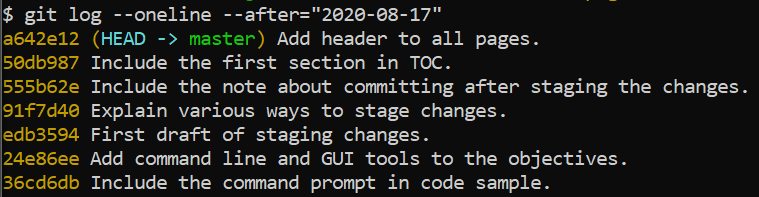
$ git log --oneline --author="Mosh"



1. Filter commits by date:

$ git log --oneline --before="2020-08-17"

$ git log --oneline --after="2020-08-17"



1. Filter commits by relative date (like yesterday, one week ago, one month ago):

$ git log --oneline --after="*yesterday*"

$ git log --oneline --after="*one week ago*"

1. Filter a commit by commit message or submit:

For example, show all the commits with word GUI in their subject.

$ git log --oneline --grep="GUI"

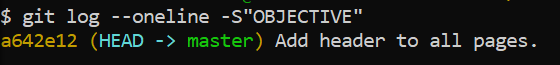


Note: *grep* command is case sensitive, so *gui* instead of *GUI* will not work.

1. Filter commits by file content change:

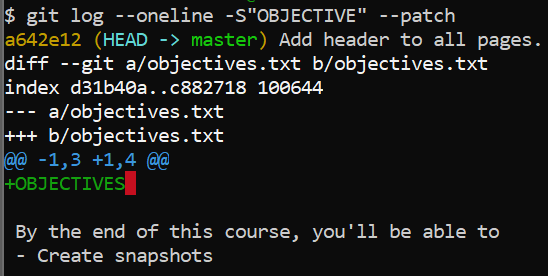
For example, all the commits that have added or removed a *HelloWorld* () function declaration or all the commits which added the word “OBJECTIVES” in it.

$ git log --oneline -S"OBJECTIVE"



And to see exact changed content, post fix with *patch*.

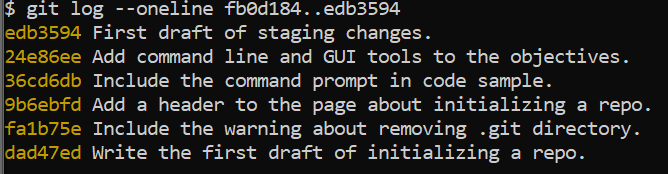
$ git log --oneline -S"OBJECTIVE" --patch



1. Filter history by range of commits:

For example all the commits between fb0d184 to edb3594.

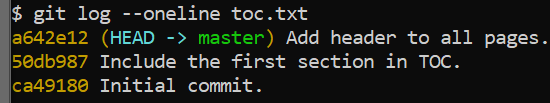
$ git log --oneline fb0d184**..**edb3594



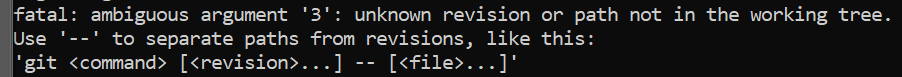
1. Filter the commits that touched / modified a particular file or bunch of files.

For example, find commits that have touched toc.txt file.

$ git log --oneline toc.txt



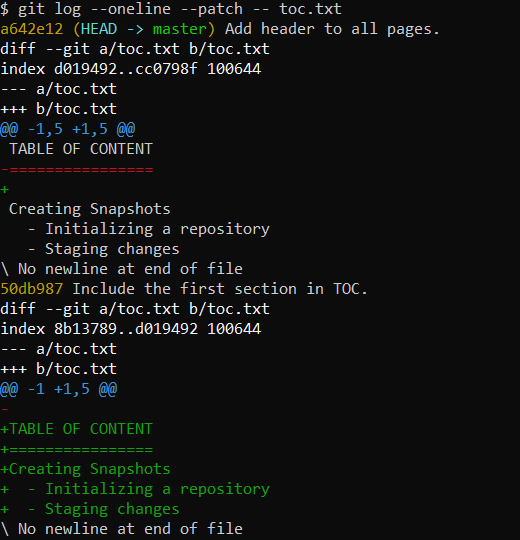
Note: Sometimes depending on the name of our file, git might complain that this name is ambiguous.



In these cases, we need to separate the file name from other options like --oneline.

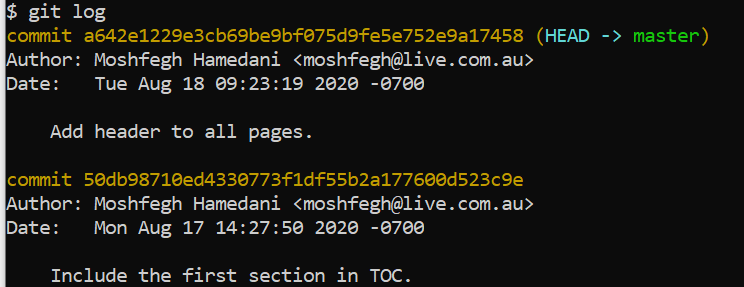
If need to actually see the changes done in the file by each commit, use patch command (*make sure patch comes before the file name*).

$ git log --oneline --patch -- toc.txt(*patch is before*)



**Formatting the Log output**:

This is the default output of log command,

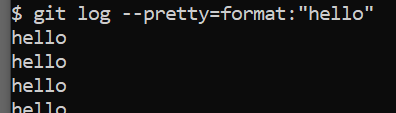


We can easily customize this output,

$ git log --pretty=format:""

Inside double quotes we can provide a *format string* which can be *a combination of plain text and some placeholders that will get replaced by GIT to display pieces of information about each commit*.

For example, if I give “*hello*” as format string,

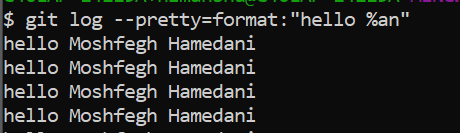


For each commit, I see the word *hello*.

Which means we can also display some information dynamically

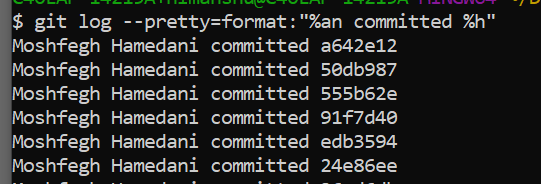
1. Like displaying author name using %an.

$ git log --pretty=format:"hello %an"



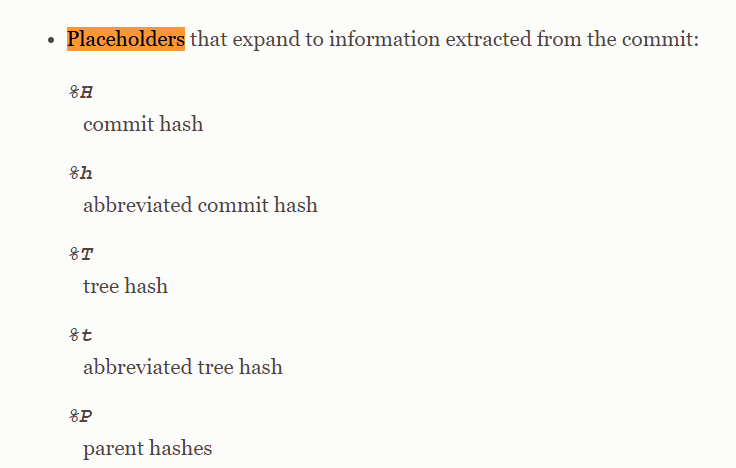
1. Giving commit ID as well with %h or %H,

$ git log --pretty=format:"%an committed %h"



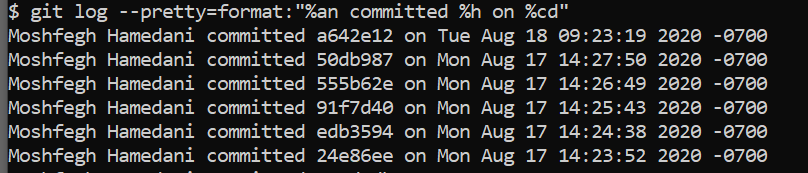
Note: To view various placeholder options with git log,

<https://git-scm.com/docs/git-log>



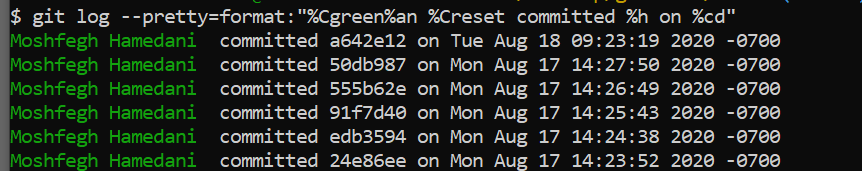
1. With commit date using %cd,

$ git log --pretty=format:"%an committed %h on %cd"

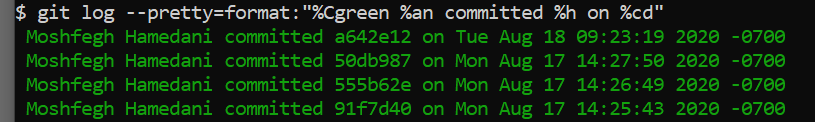


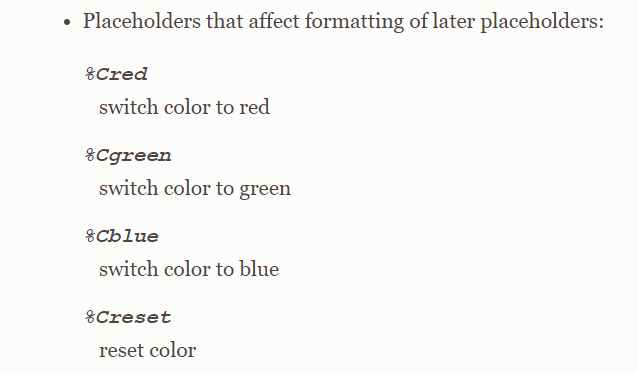
1. Change the color of a specific placeholder using %C

$ git log --pretty=format:"%Cgreen%an %Creset committed %h on %cd"



Note: Here we used %Creset after %an to change the color of rest of text to normal otherwise everything will be green.





**Creating Aliases**:

In git we can easily create aliases for the commands that we use frequently, so we do not have to type them in their long form.

$ git config --global

We use git config --global so that this configuration applies to all our repositories.

Navigate to *alias* section inside config and define a new property or the *custom command name* (*like lg here*),

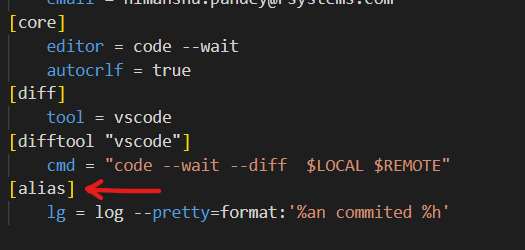
$ git config --global alias.lg

Now fill this property with the actual command

$ git config --global alias.lg "log --pretty=format:'%an commited %h'"

To view this global config file,

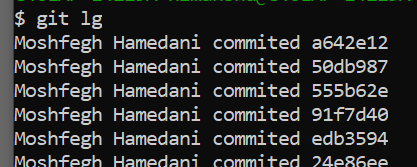
$ git config --global -e



So our custom command *lg* is saved inside alias section

We can test it,

$ git lg



Another example,

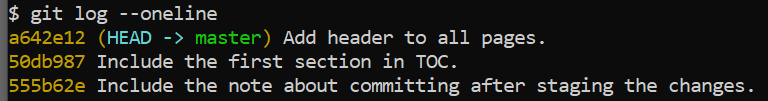
Create another alias called unstage and bind it will git restore command.

$ git config --global alias.unstage "restore --staged ."

With this we can restore all the files from our staging area,

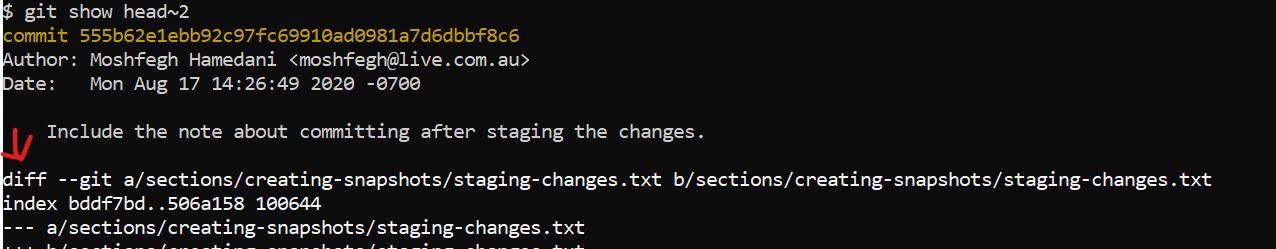
$ git unstage

**Viewing a Commit**:



1. View 2nd commit from the head

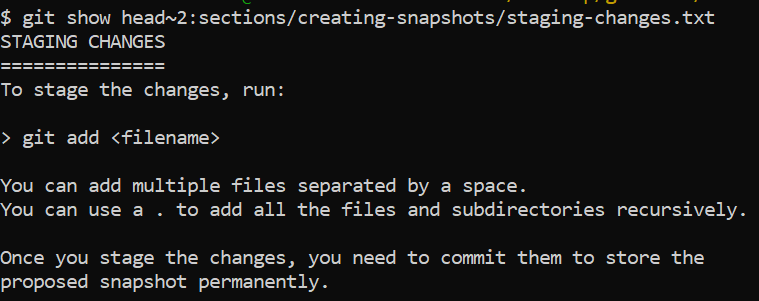
$ git show head~2



We can see the diff of each file that has changed in this commit.

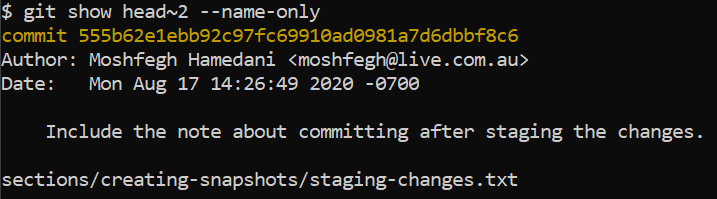
1. If we do not want to see changes but the final version of the file in this commit.

$ git show head~2:sections/creating-snapshots/staging-changes.txt (*same command as before just added complete path of the file we want to view after colon*).



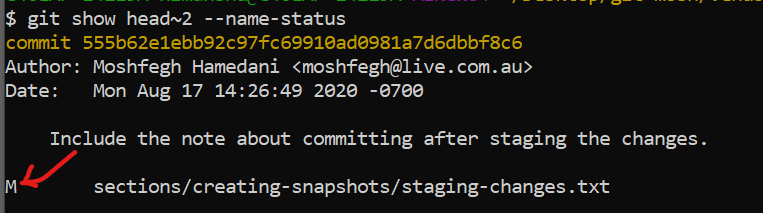
1. To see files that have been modified by a specific commit

$ git show head~2 --name-only (*We use* --name-only *argument*)



1. To see which files have been added, modified or deleted in a commit.

$ git show head~2 --name-status (use --name-status option)



Here M means this file has been modified.

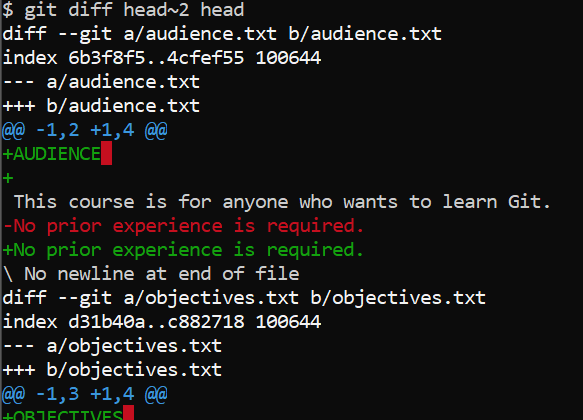
**Viewing the changes across commits**:

Suppose we want to see what has been changed from last 2 or 3 commits, to do that we use *diff* command.

(*In previous section we learned to use diff command for viewing staged and unstaged changes*)

1. Now we can use the same command to see the differences between two commits.

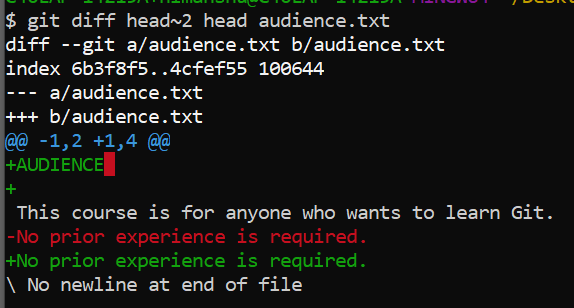
$ git diff head~2 head (*diff between head~2 and head*)



Now we can see diff of every file across these two commits.

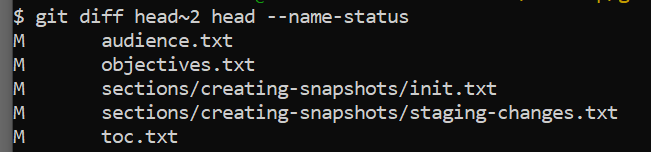
1. To see the changes of a particular file across two commits

$ git diff head~2 head audience.txt



Note: Similar to show command we have options for --name*-status* and --name*-only* as well.

$ git diff head~2 head --name-status



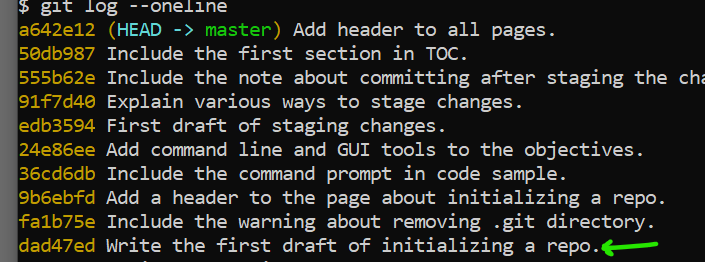
**Checking out a Commit**:

So we talked about viewing individual files in the commit. Now sometimes we want to see complete snapshot of our project at a point in time.

In these situations we can check out a given commit and this will restore our working directory to the snapshots stored in that commit.

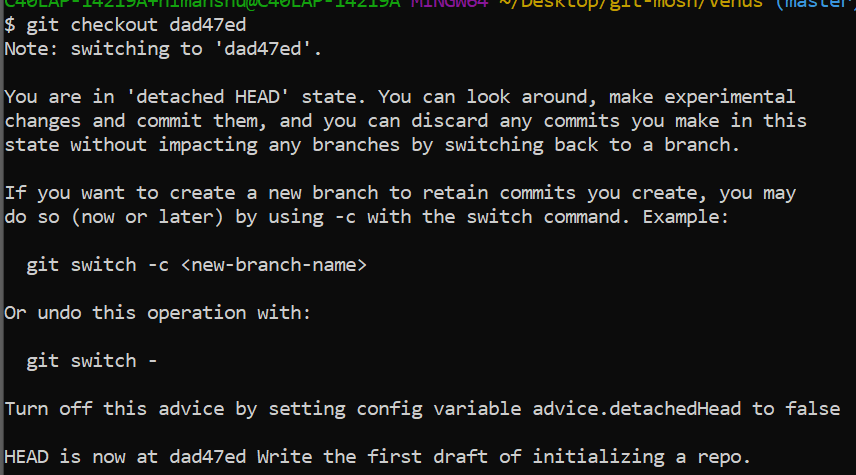
So our working directory will look exactly like in earlier point of time.

Let us say we want to restore our working directory to dad47ed.

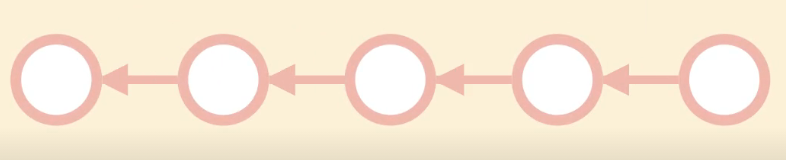


$ git checkout dad47ed

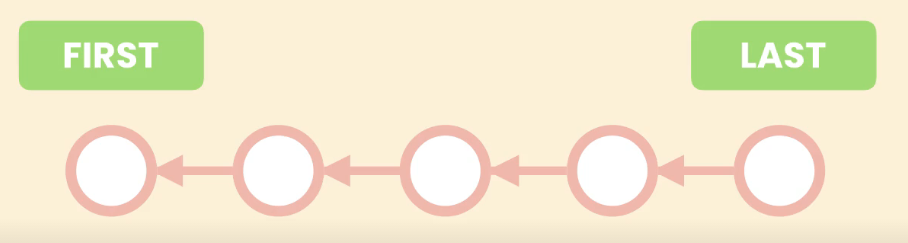
We get this message after entering this git command.



You are in ‘*detached Head*’ state.



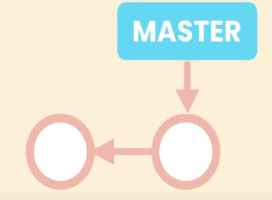
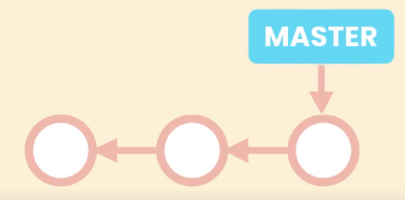
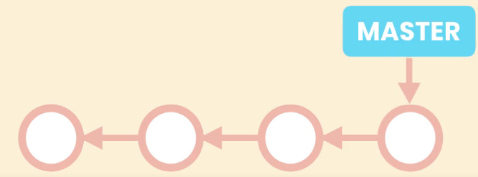
Here are all the commits we have created so far.



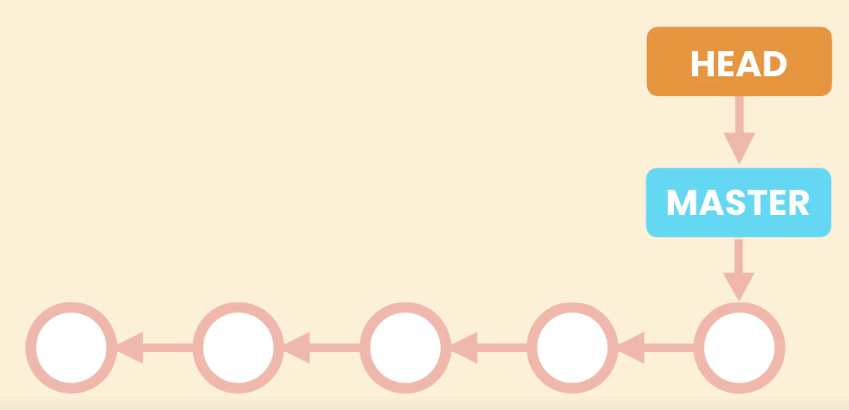
First commit is on the left and last commit is on the right. As you can see each commit points to previous commit. This is how GIT maintains history.

All these commits we have created so far are part of what we call ‘*Master branch*’, the main line of work. In GIT and many other version control systems *we can have multiple branches to work on different stories in isolation*.

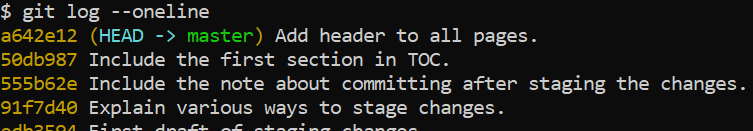
The way we represent branches is by using a pointer. So master is pointing to the last commit we have created so far.

 --> -->

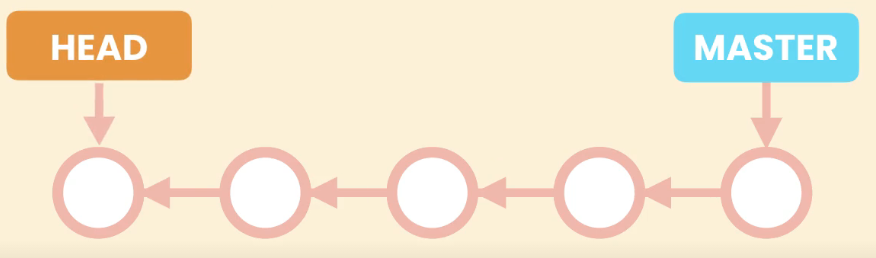
As we create new commits master moves forward to point to the last commit. Since we can have multiple branches GIT needs to know what the branch we are currently working on is.



To do this it uses another special pointer called ‘*Head*’. So *head points to current branch we are working on*, in this case it is master.



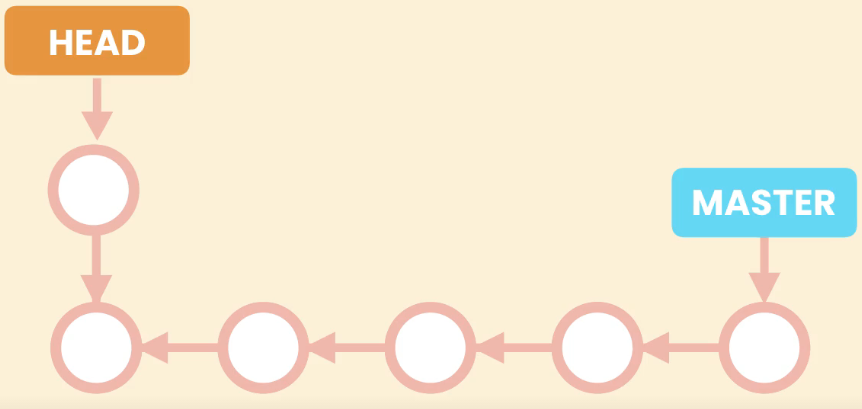
(HEAD - > master) is in front of last commit. So as we create new commits. These two pointers *head* and *master* also move forward.



*Detached head state*

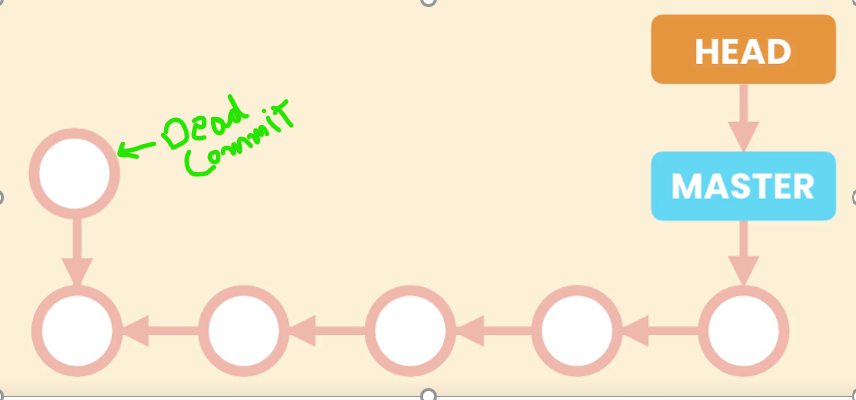
Here is the thing, *when we checkout a particular commit. The head pointer will move to that commit*. This is what we call a detached head, which *means it is not attached to a branch anymore. Instead it is pointing to a specific commit*.

Note: In case of detached head, we should not make new commits. We should only view our code. The reason is if create a new commit, that commit will be added over here.

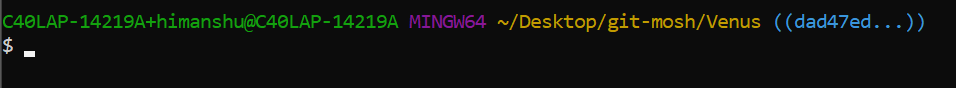


Since at some point we have to attach head pointer to a branch. There is a problem here. This commit we just created here is not reachable by any other commit or pointers, so it is like a *dead* commit.

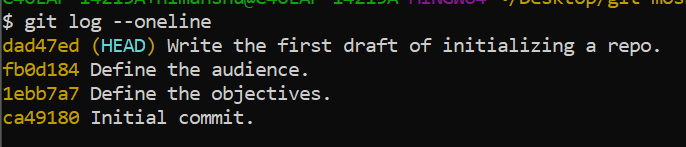
GIT checks commits like these periodically and removes them to save space. So we will lose that commit along with all the changes.



I am not seeing master in my GIT bash but instead this,

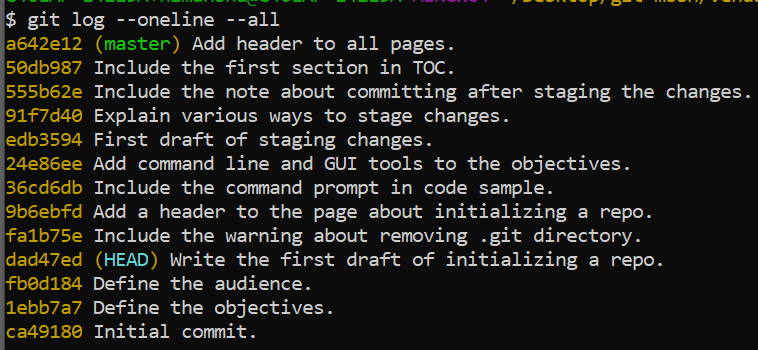


And git log does not return all the commits we made earlier,



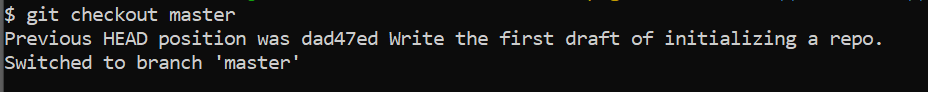
To view all those commits as well,

$ git log --oneline --all



To attach the head pointer to master branch again,

$ git checkout master



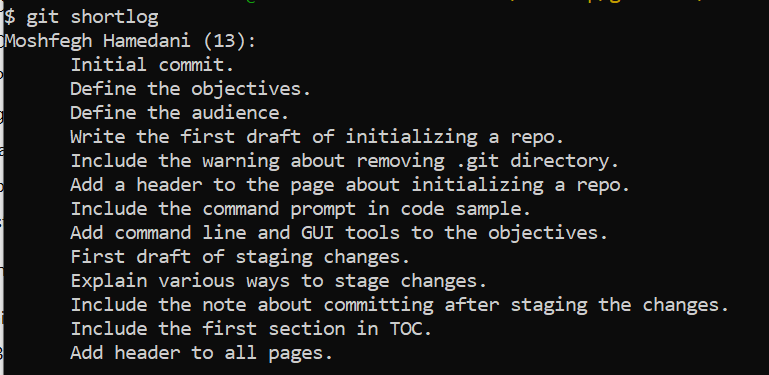
**Finding Bugs using Bisect**:

**Finding contributors using Shortlog**:

Sometimes we need to find all the people who have contributed to our project.

We have a command for that called *shortlog*.

$ git shortlog



Here we can see name of contributor and number of commits they have created and a summary of commit messages.

To see available options with this command use – h.

