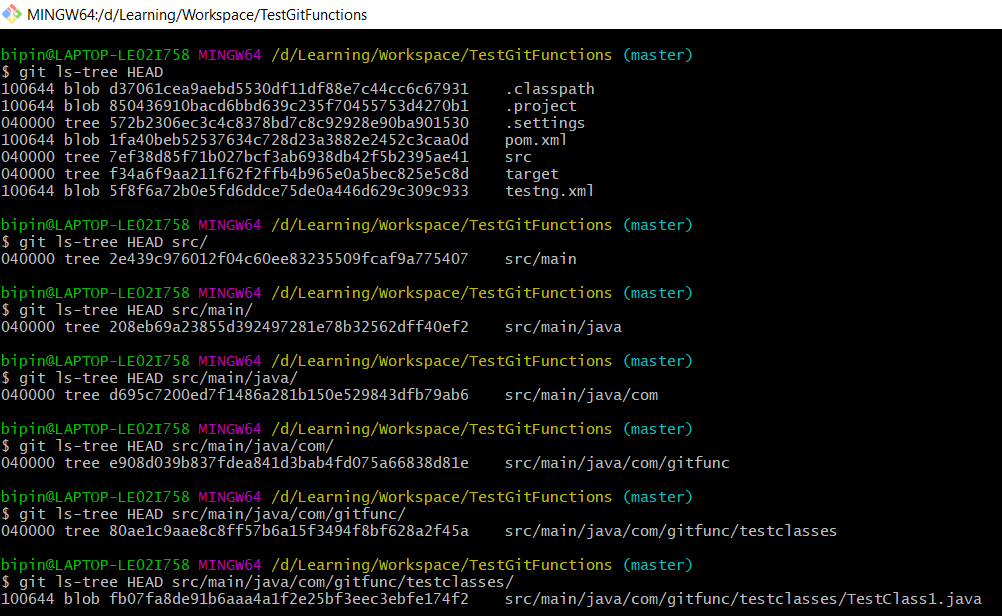
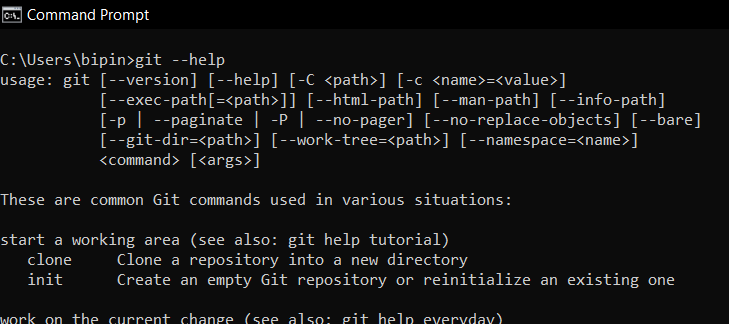
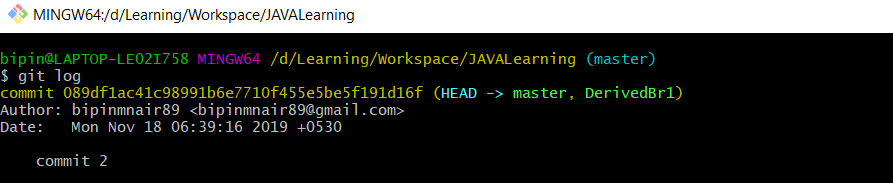
Commands covered in this document with examples

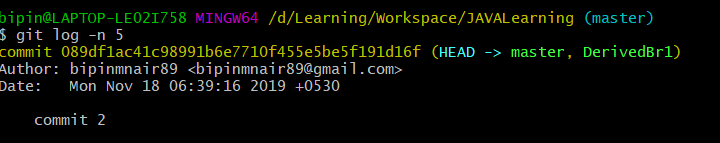
1. **Git version**
2. **Git help**
3. **Git log**
4. **Git init**
5. **Git status**
6. **Git add**
7. **Git commit**
8. **Git push**
9. **Git diff** –> *git diff, git diff--staged, git diff--cached, git diff to compare between commits*
10. **Git restore**
11. Deleting a class from working directory and moving the changes to remote repository branch
12. **Git show** –> *to view the details of a specific commit*
13. **Git checkout -- \*** - *this command will revert all the changes in the working directory\*
14. **Git commit --*amend –m “<msg>”*** *– this command is used to amend the latest commit*
15. **Git revert** *-> This command is used to revert a commit*
16. **Git clean** *-> To remove untracked files*
17. **Finding Ancestry** commits
18. **Branching in GIT**
19. **Git reset**
20. **Merging in GIT**
21. **Stashing in GIT**
22. **Remote in GIT**

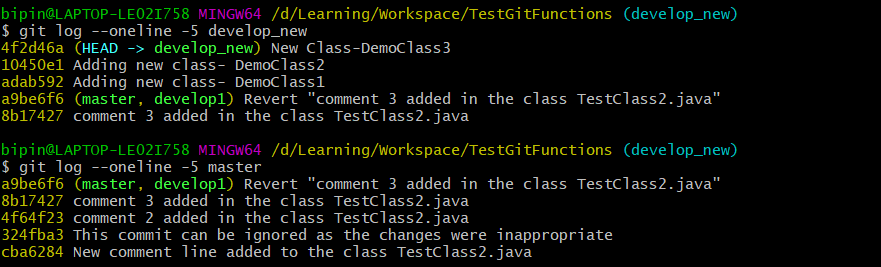
*Pending 🡪 .gitignore folder creation and functionality*

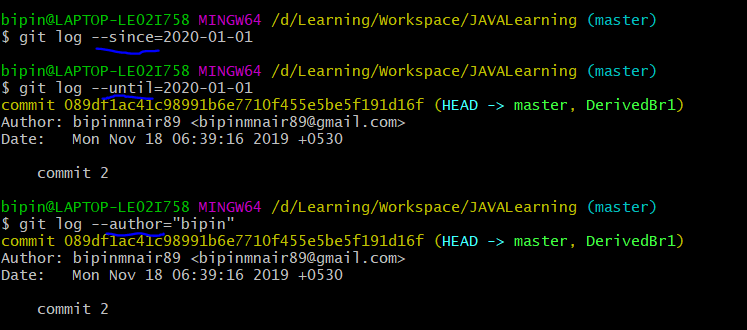
*Tree-ish 🡪 Each commit is considered as tree-ish*

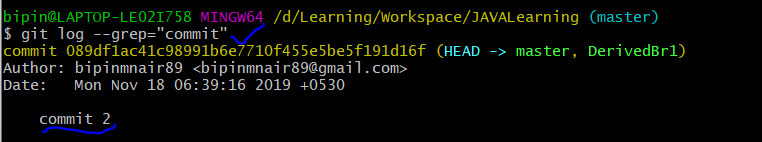
***Git ls-tree HEAD*** *-> command used to get the directory structure of each commits*

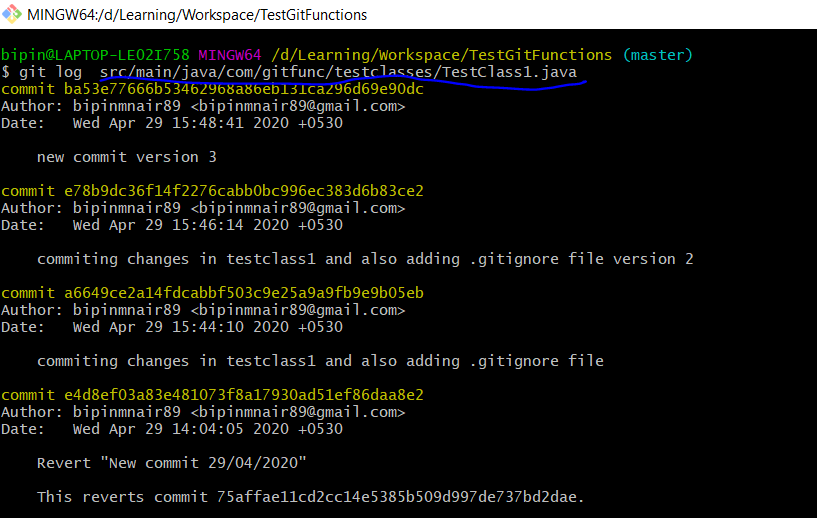
1. **Git version**
2. **Git help**
3. **Git log** – to see the details and history of the commits that is performed

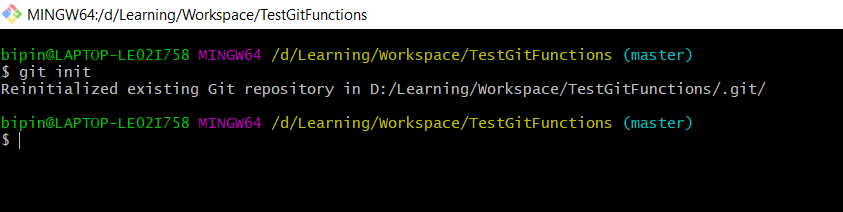
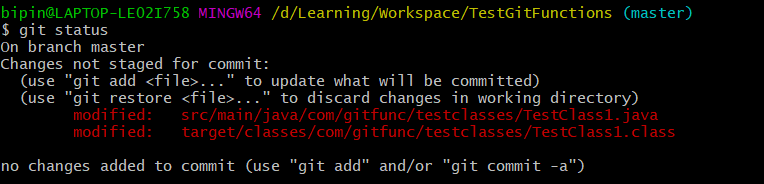
To see the last 5 commits

To see the logs for a specific branch in oneline

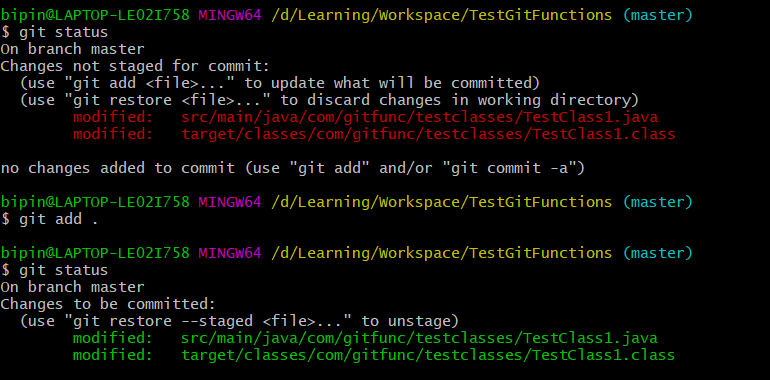
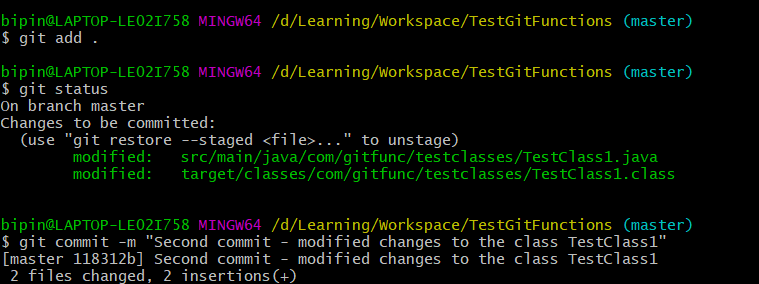
Search logs using since, until and author

Search any text in the comments section of commits in logs using grep (grep means globally searching for regular expression) 

We can also use git log command to find the commits which are related to a particular folder or class and hence narrow the search further

1. **Git init** – used to initialise a project structure as a git repository
2. **Git status** – to check the status of the files in the working copy against the local git repo

Here modified means that changes has been made to an existing file

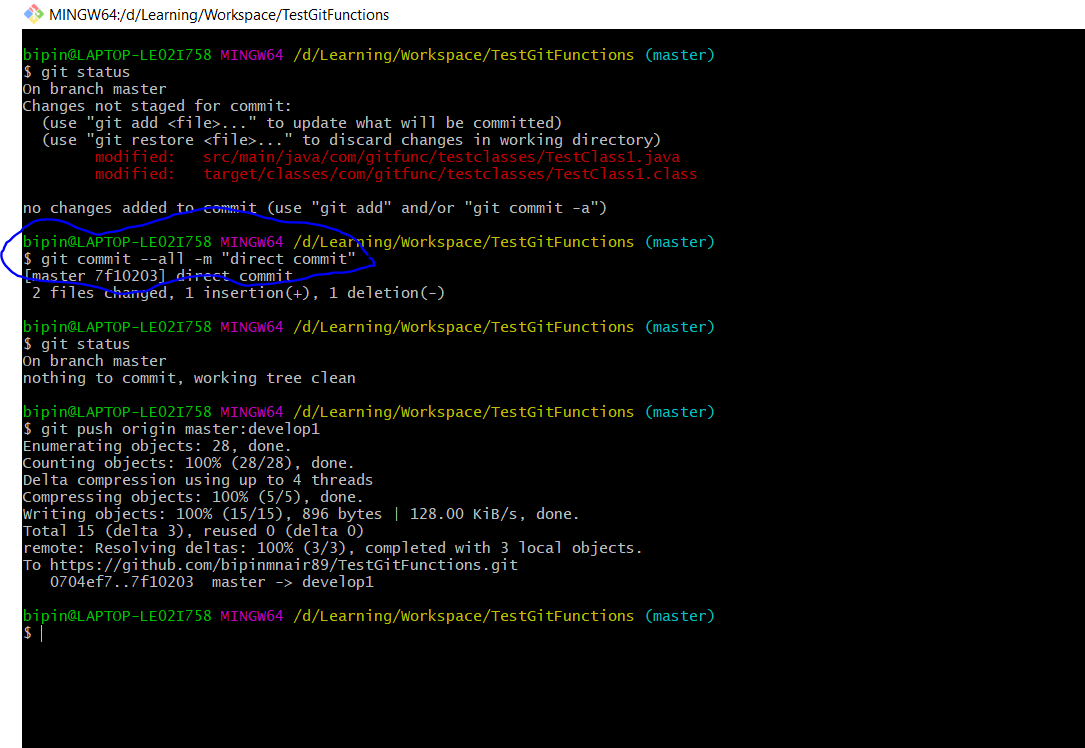
1. **Git add** – command is used to add the changes to staging area
2. **Git commit** – to commit the changes to the local git repository

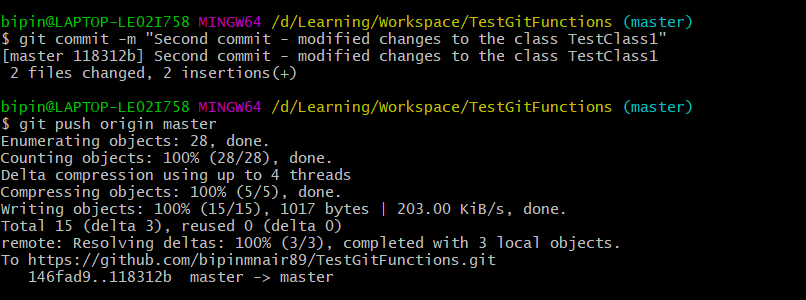
To directly commit without adding it to staging we can use either of the below commands

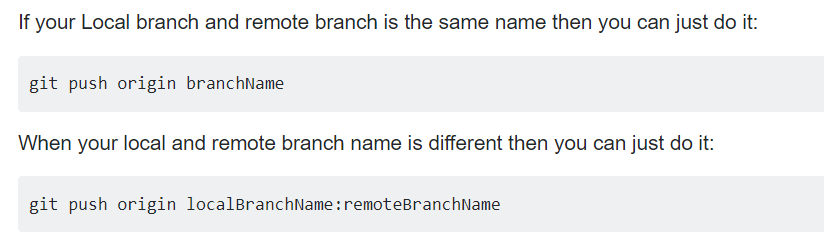
**Git commit --all –m”Direct commit**” or

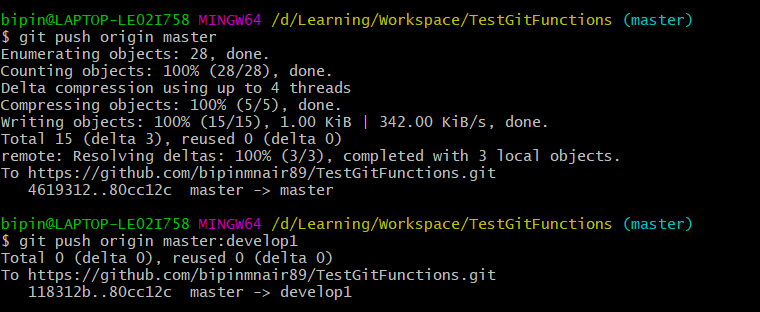
**Git commit –a –m”Direct commit”**

Disadvantage of this method is that the untracked files won’t get committed.(ie only the modified changes would get committed and the new ones wont as they need to be added to staging)

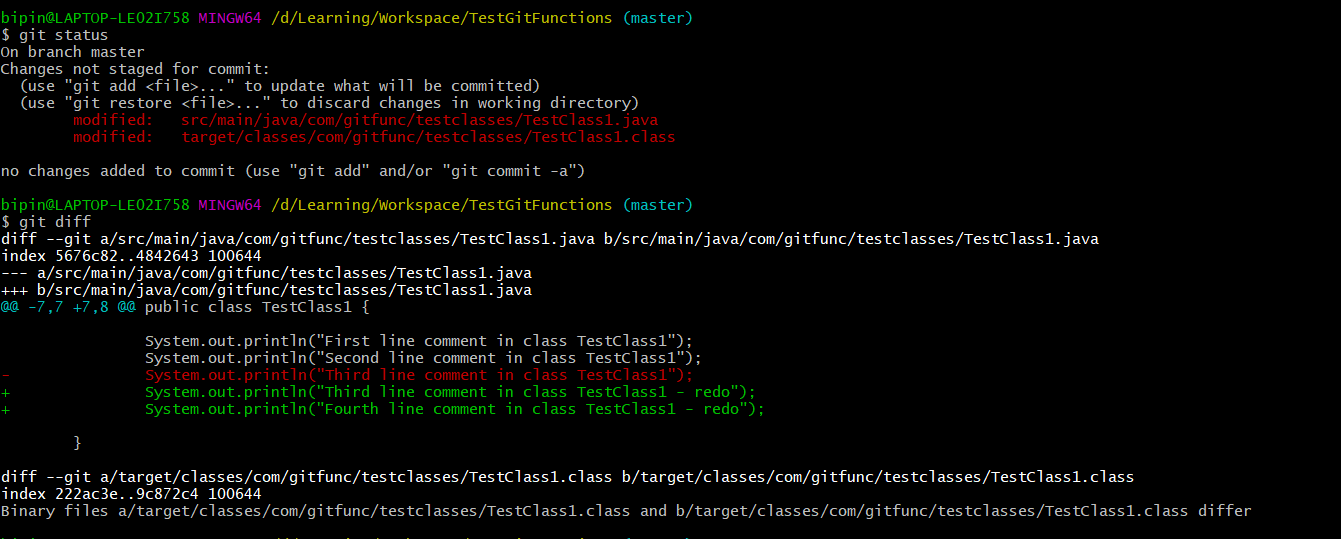


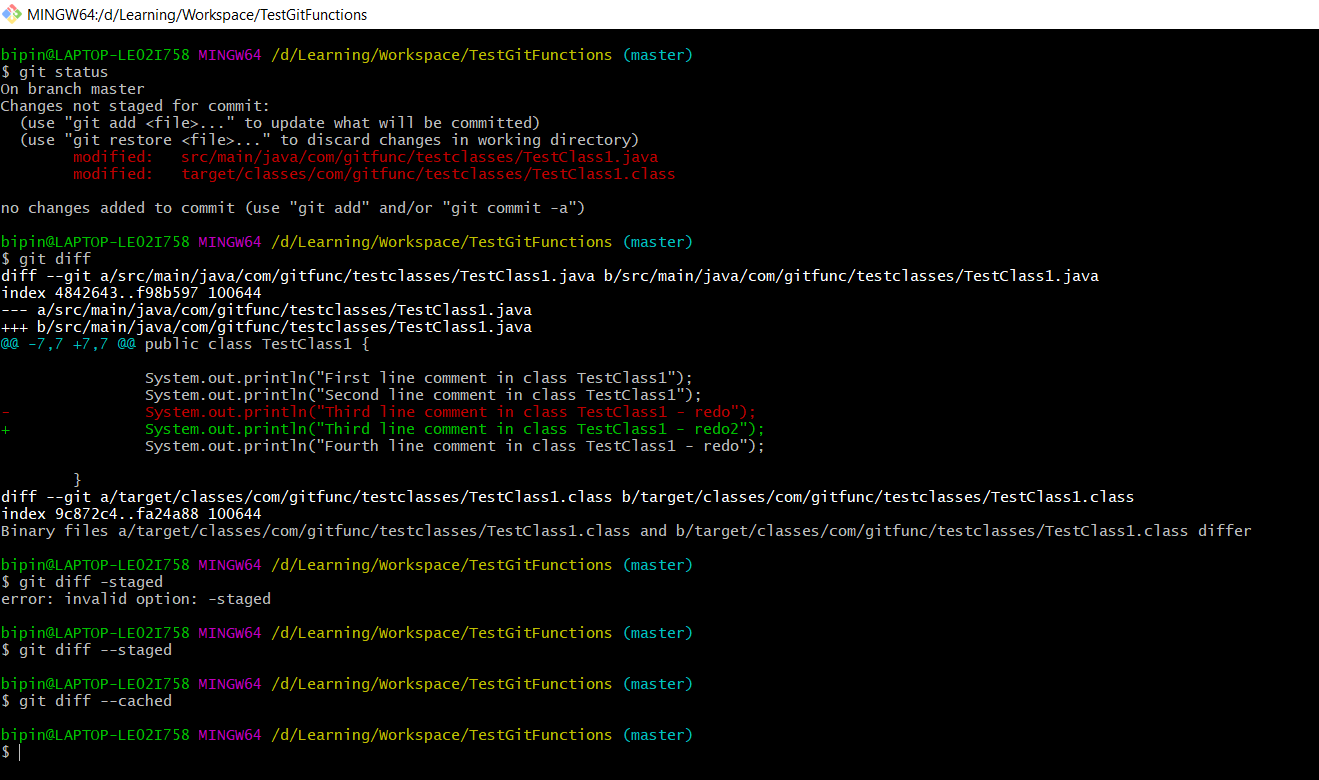
1. **Git push** – to push the changes to the origin which is the remote github repository

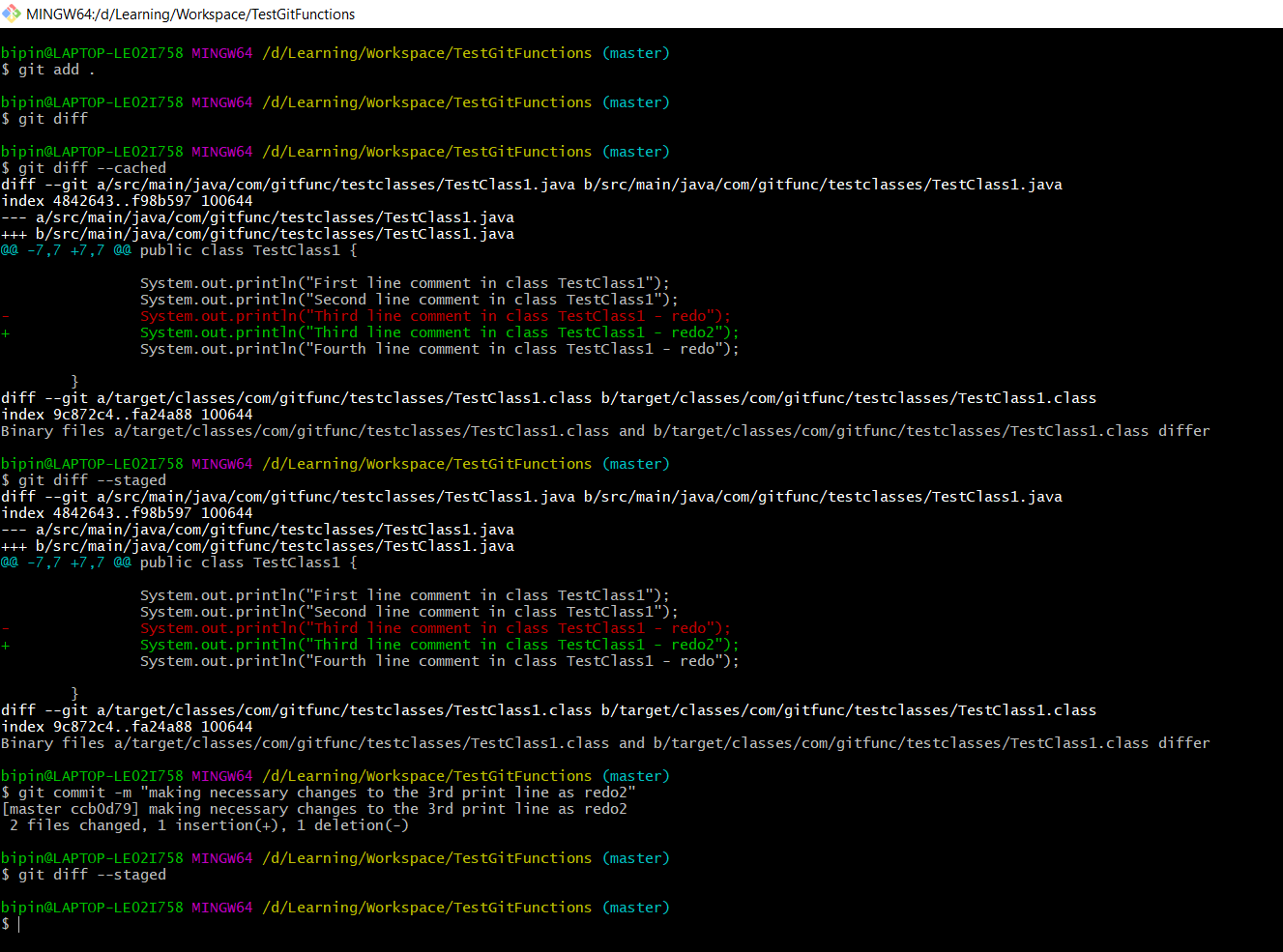




1. **Git diff** – command is used to find the difference that has been made to a modified file between the working directory and the staging tree.

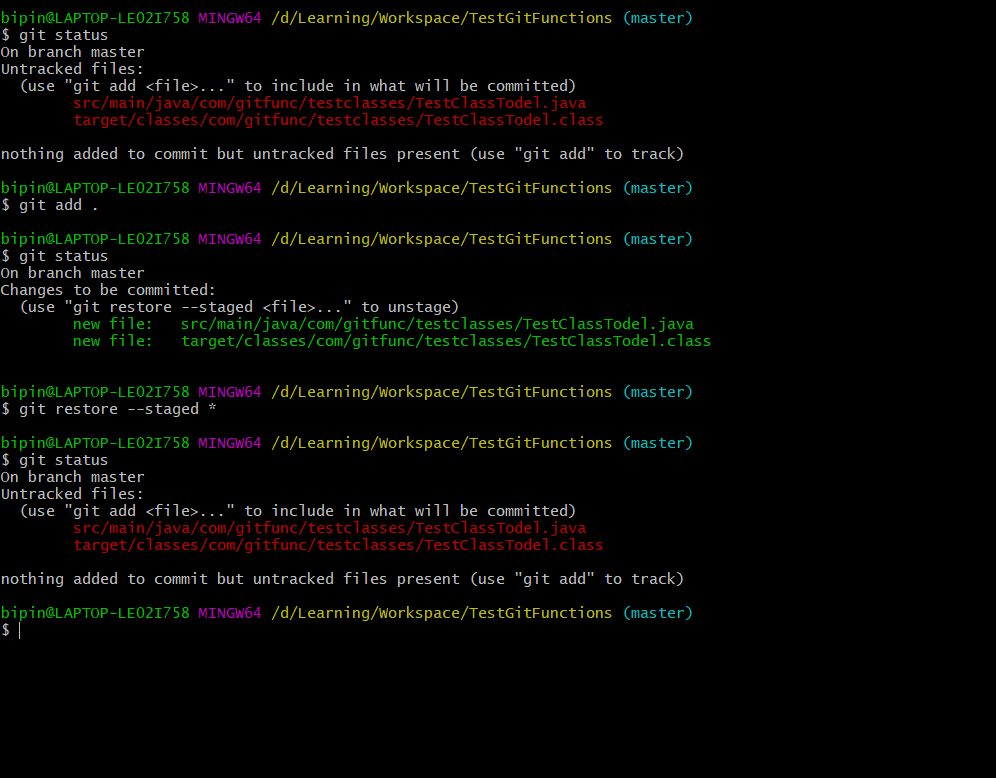
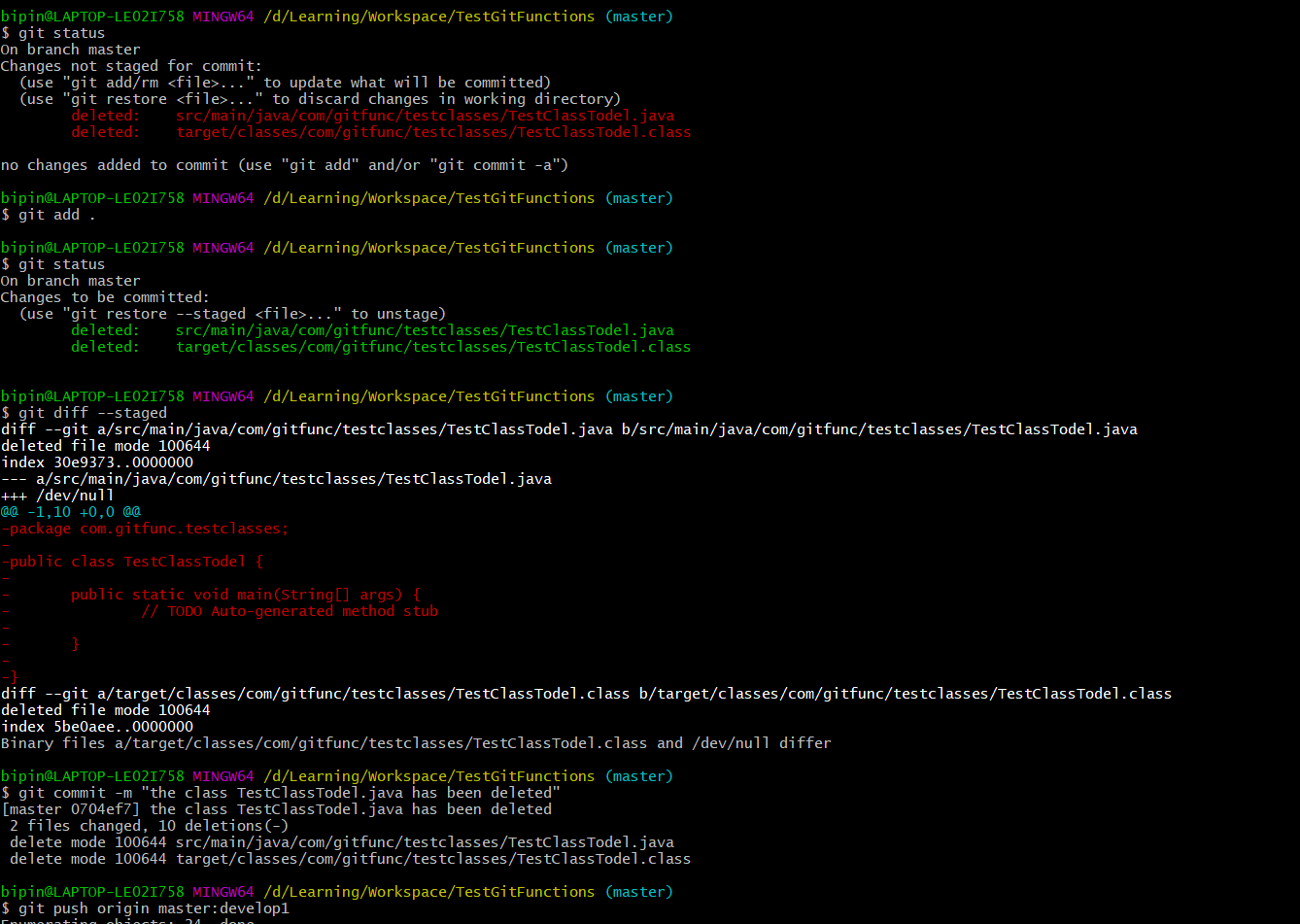
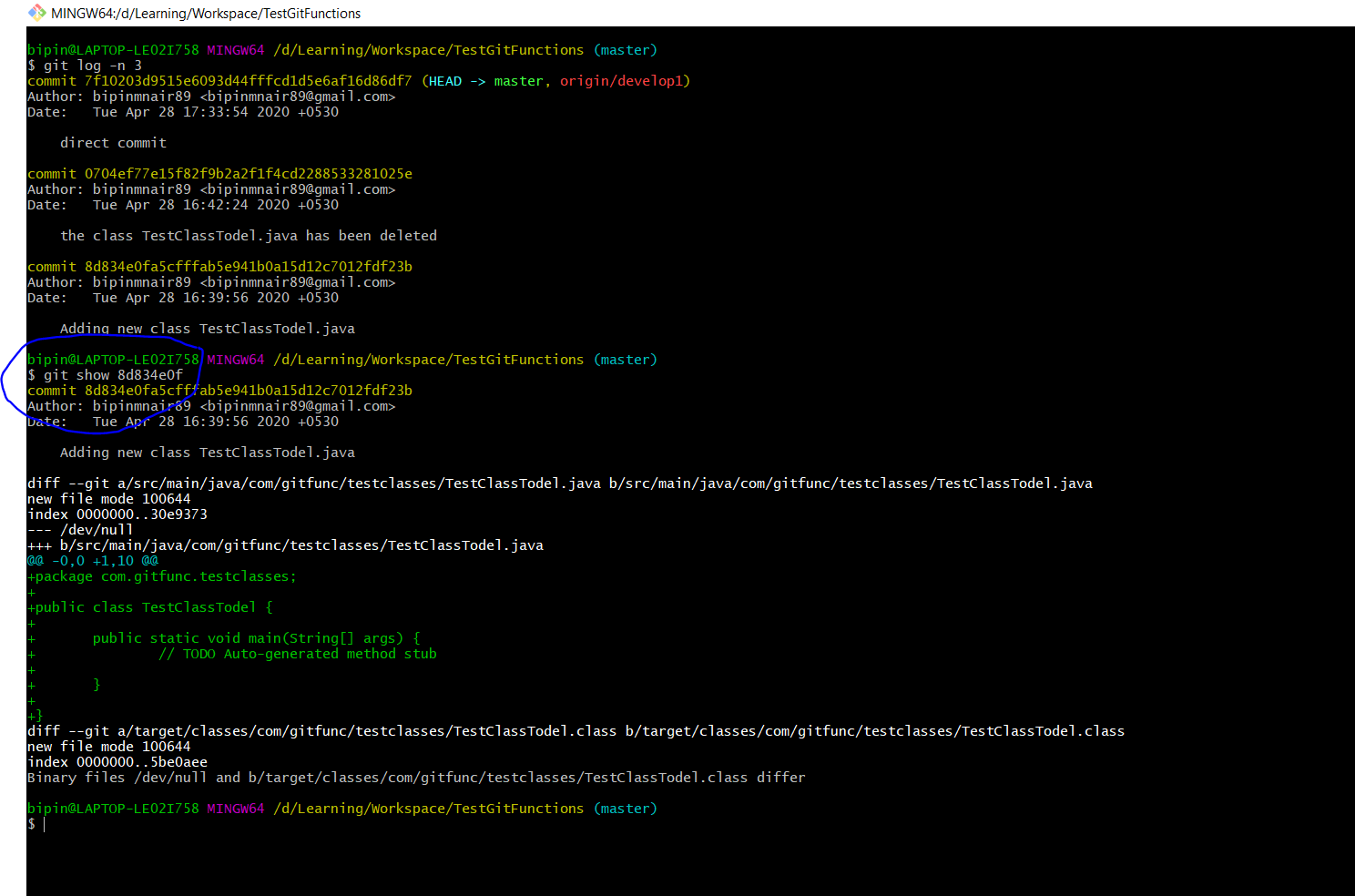
Here in the below screenshot. The modified line is marked in red and its change is shown below in green and under it shows the fourth commit line which was newly added

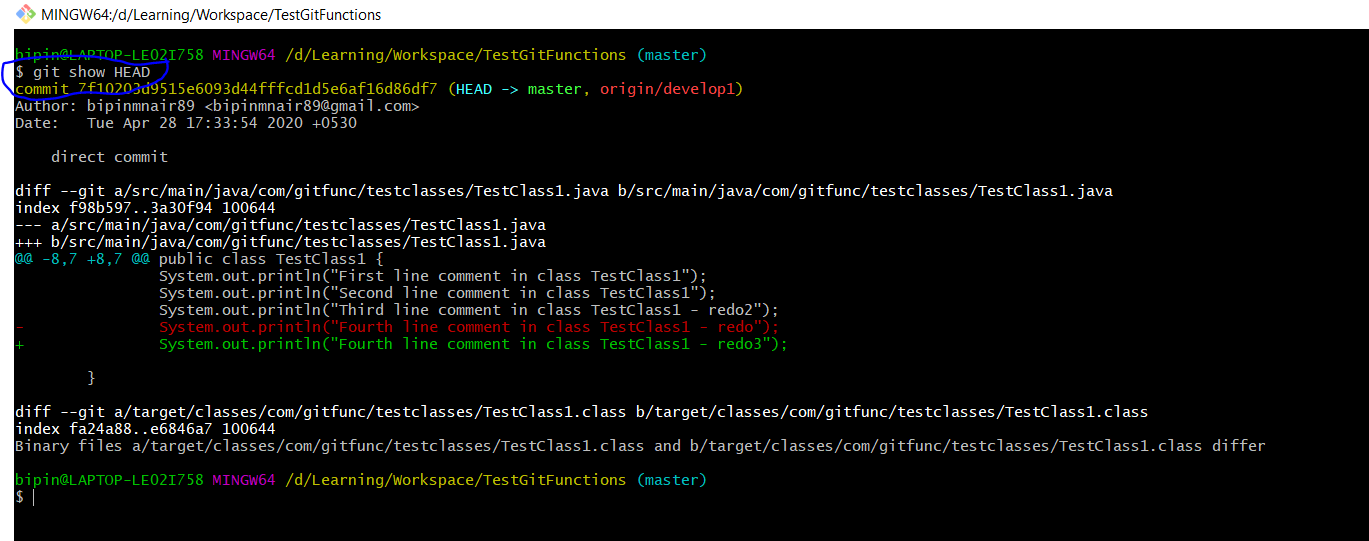
**Git diff--staged** or **Git diff --cached** -> this command would show the difference that has been made to a modified file between the staging tree and the local repository

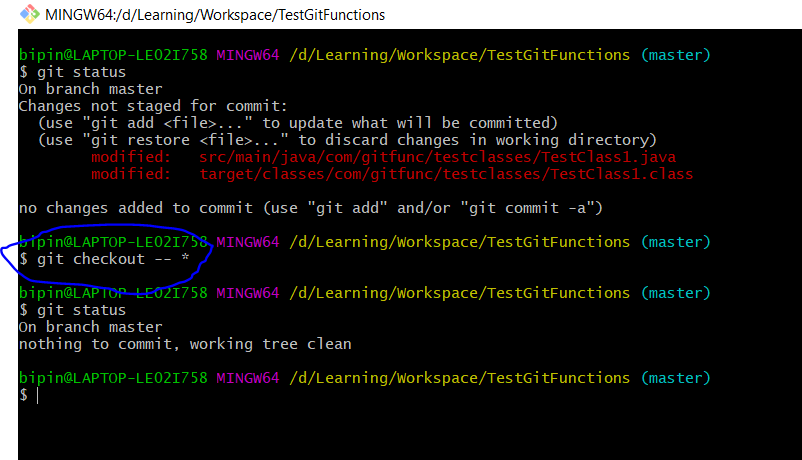


**git diff ccb0d79d..HEAD --color-words**

-> compare between commits ,here **ccb0d79d** is the SHA no of one commit and HEAD is the latest commit. **–color-words** is given to highlight the specific differences in separate colors

1. **Git restore** -> **Git restore --staged <filename>** This command will help to move the items from staging tree back to the working directory
2. **Deleting a class from working directory and moving the changes to remote repository branch** 
3. **Git show** – command is used to view the details of a specific commit. 

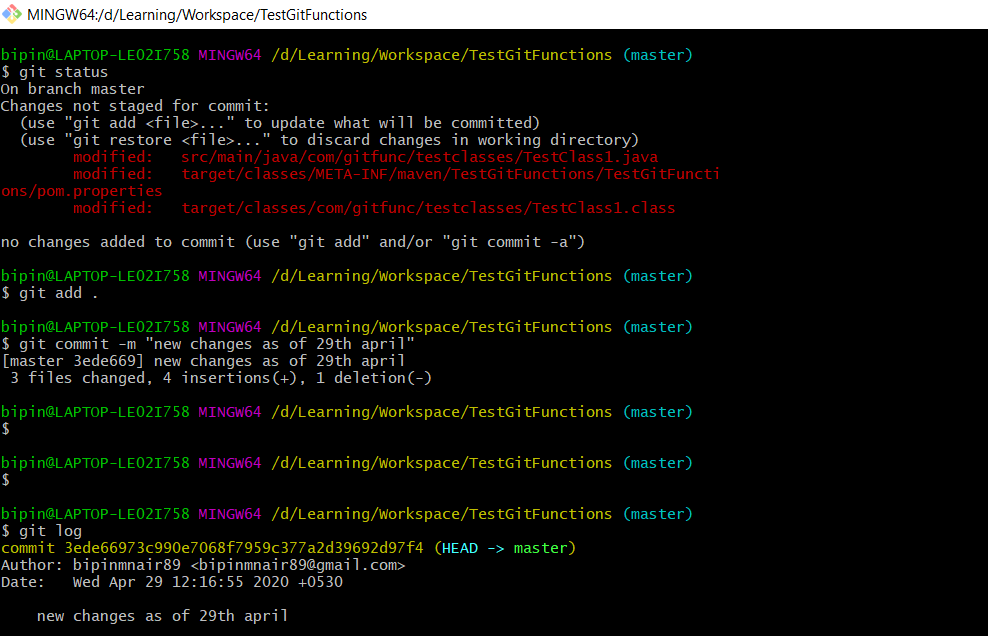
**Git show HEAD** – this command will show the latest commit, instead of giving the SHA number we can also give HEAD to view the latest commit as HEAD points to the latest commit 

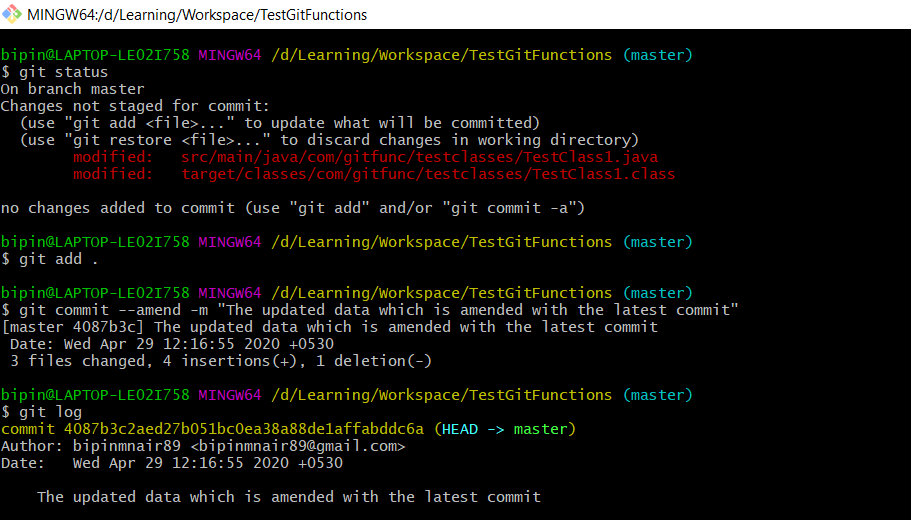
1. **Git checkout -- \*** -> this command will revert all the changes that has been made in the working directory 
2. **Git commit --amend –m “new commit for latest class”** -> For each commit there is a SHA value which is generated(can be seen by giving git log or git show) and this SHA value is usually the combination of the username,email,msg added and the previous SHA value

Hence it is not possible to amend other commits except the latest one where the HEAD value is pointing.

This amend command will literally move the commit back to staging and then bundle with the new change.

Here a commit is made

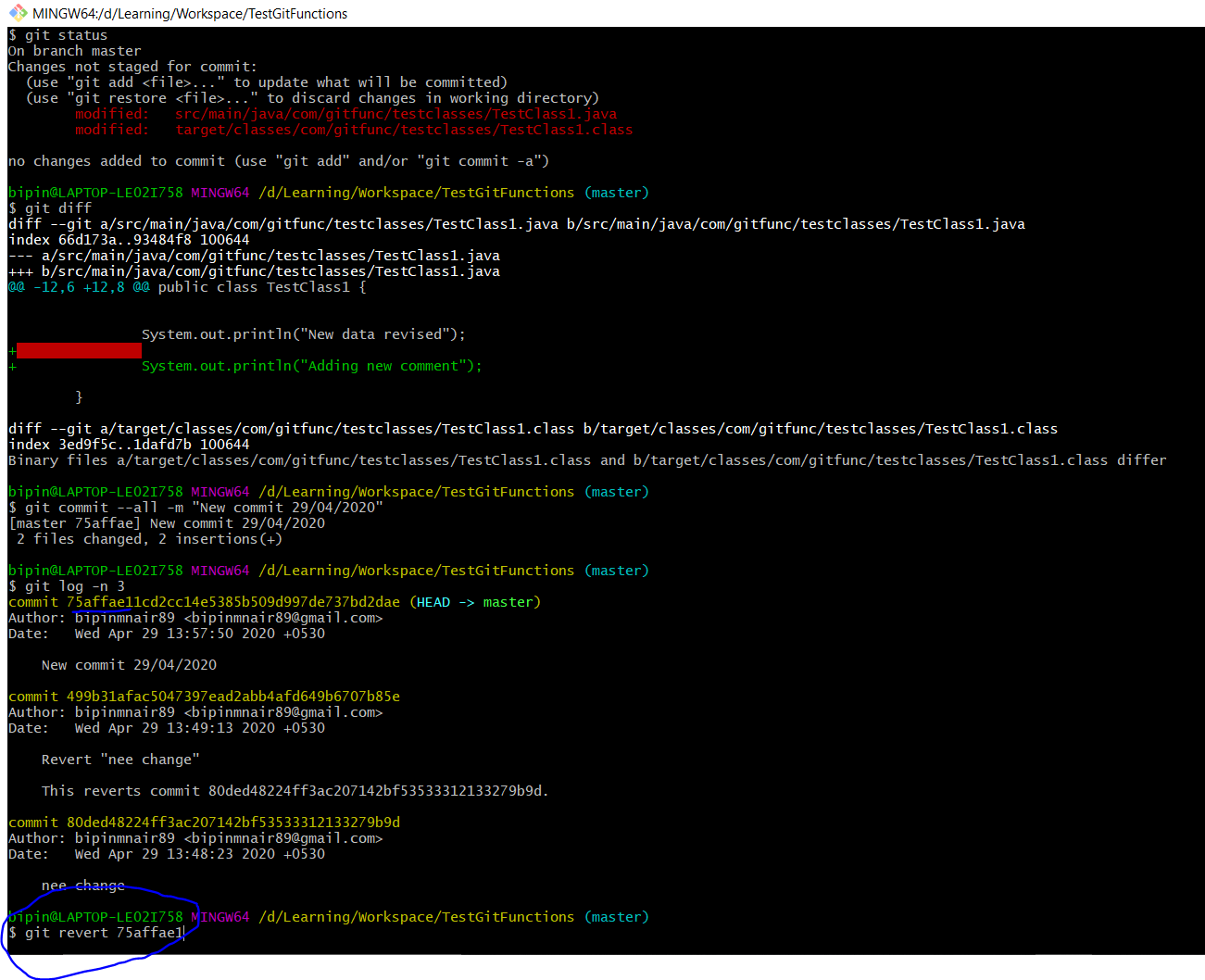


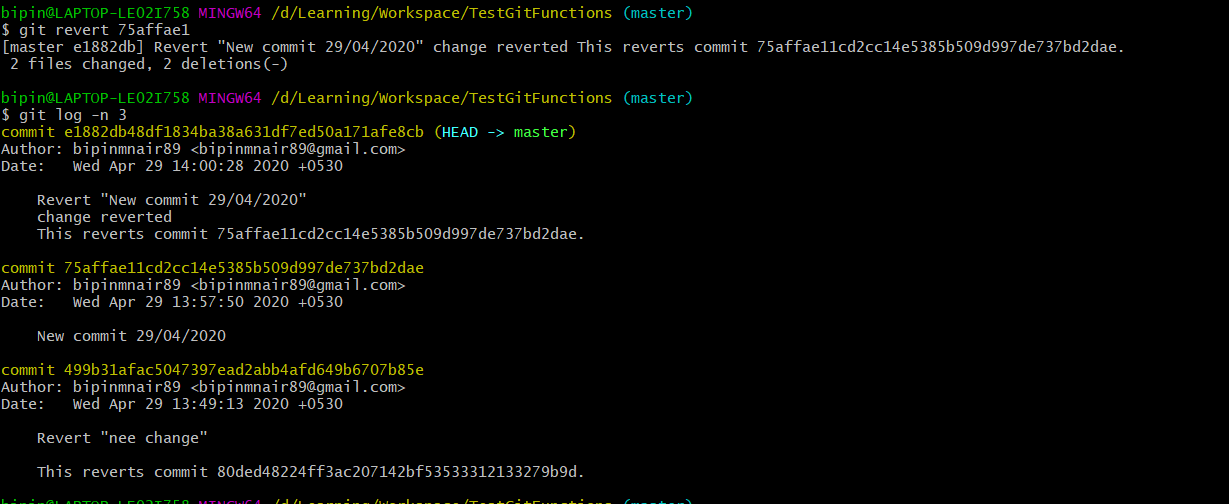
Now we are making one more change but instead of adding it as a new commit we would use the amend command to bundle it with the latest commit to which the HEAD pointer is pointing. 

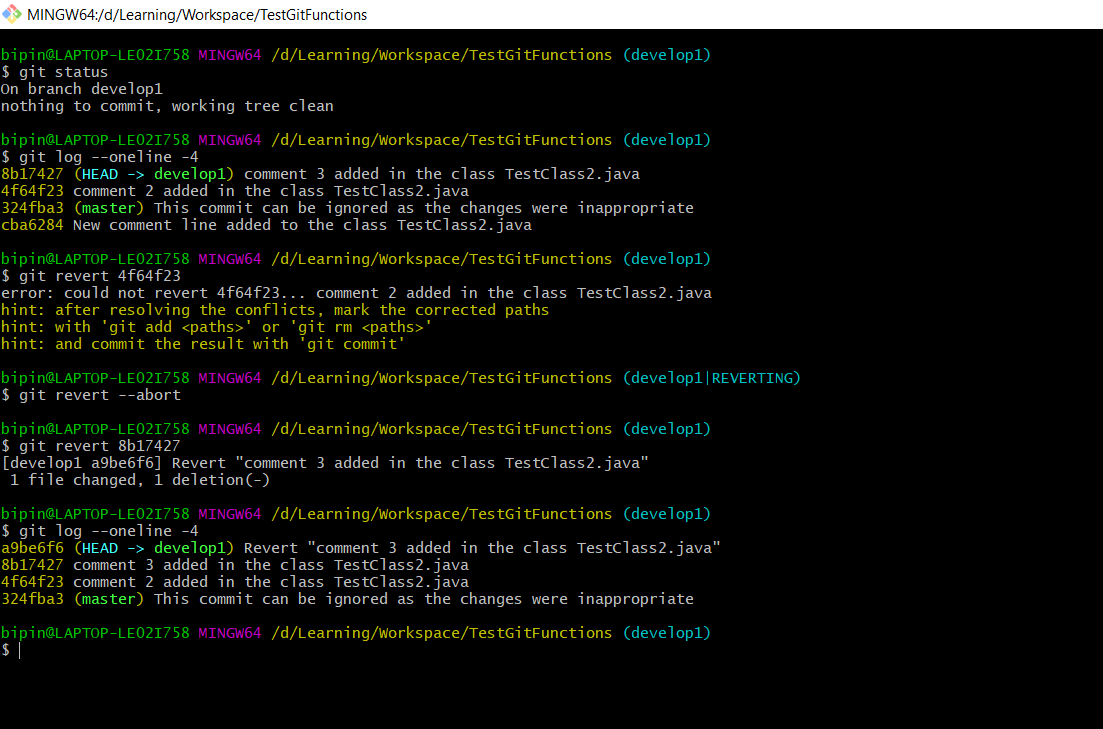
1. **Git revert <SHA value>** -> This command will revert the commit and undo all the changes till the working directory.

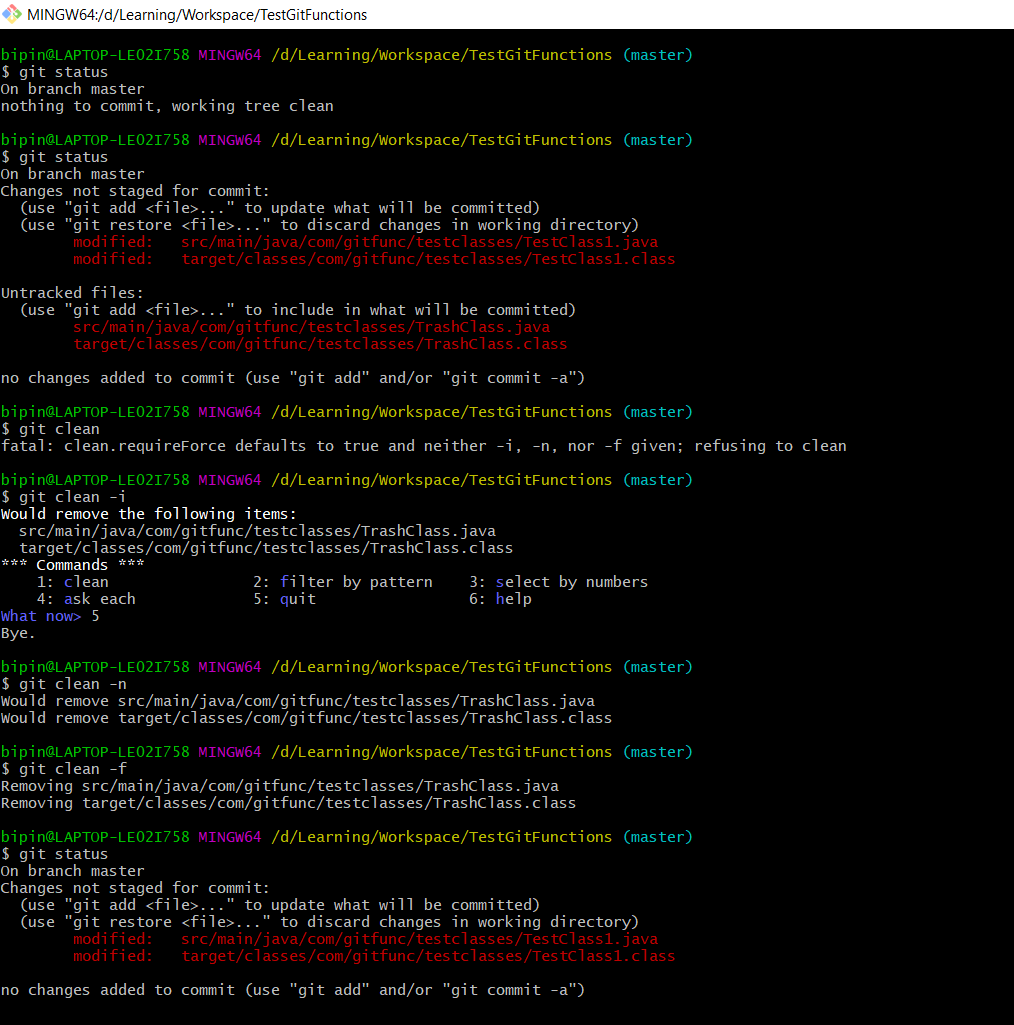
Git revert command will create a new commit wherin it states that the earlier commit has been reverted

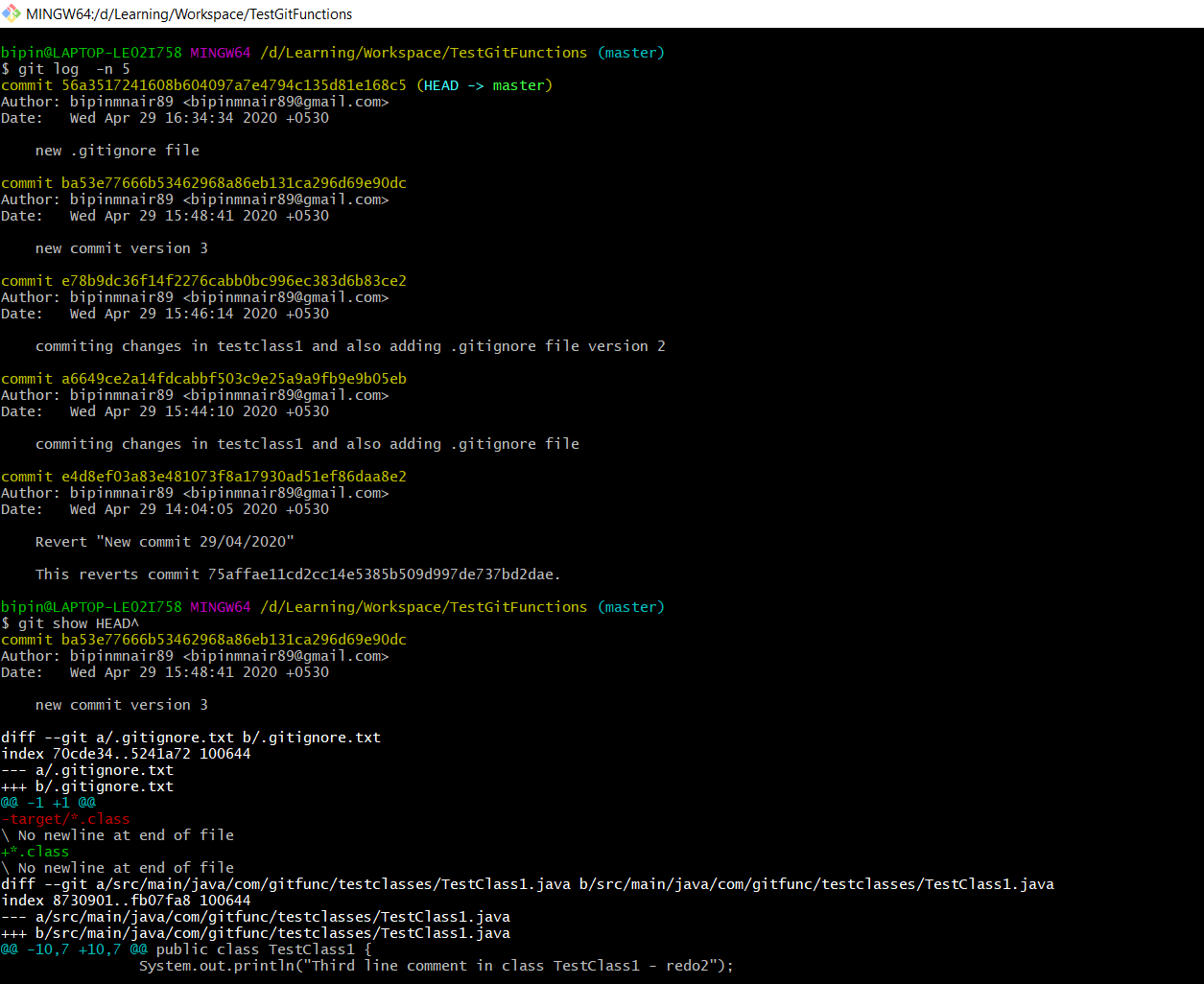
But the Git revert command will not remove the earlier commit from the logs and instead it will revert the changes till the working directory and also create a new commit stating that earlier commit has been reverted

In the below screenshot we can see the change that has been made and commited and later using the SHA value change is reverted

Below we can see that the change has been reverted and the logs have been updated

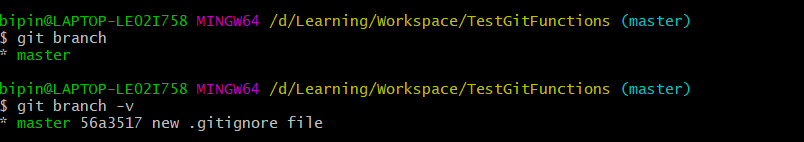
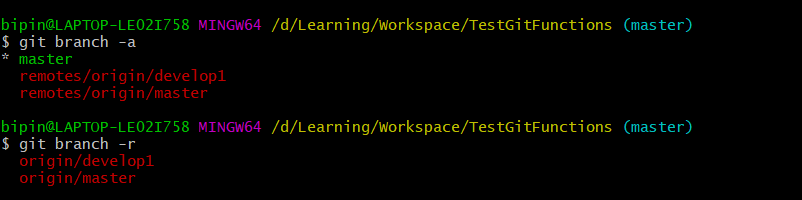
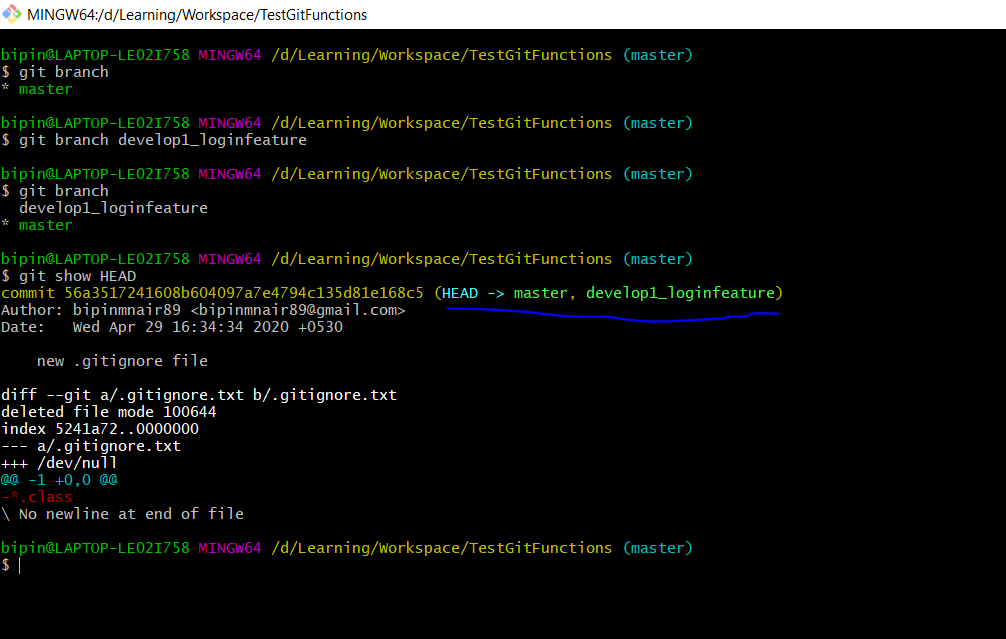
The **Git revert –abort** command can be used to abort a specific revert

1. **Git Clean** -> This command is used to remove untracked files. This only works with untracked files and not modified files. Only if we give Git clean –f , then the files would get removed as this command is quite risky hence the extra caution
2. **Finding Ancestory commits** -> This method can be easily used to find the parent , grandparent or greatgrandparent commit. Example – To find the parent commit - ^ or HEAD~1 and to find the grandparent commit - ^^ or HEAD~2

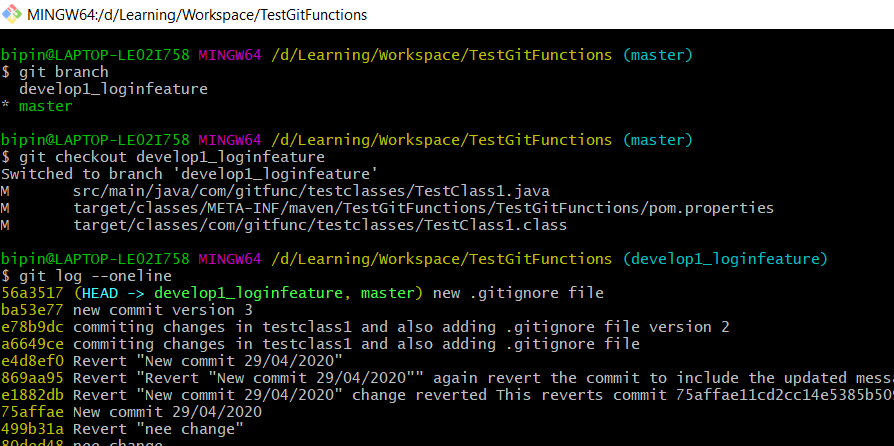
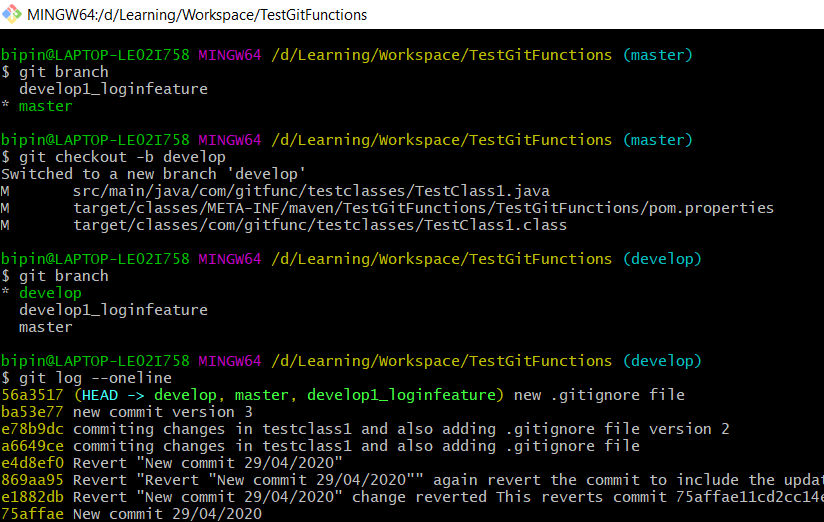
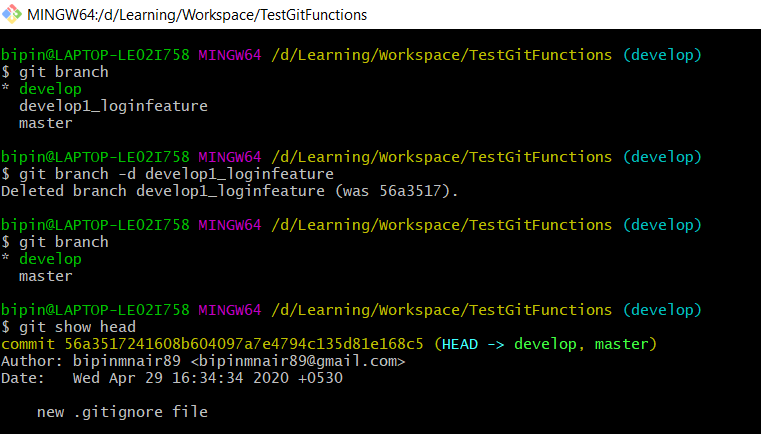


1. **Branching in GIT** -> Branching helps users to work on a specific functionality without impacting the whole project changes. For example. The development is going on branchdev1 and you have to start working on a new functionality. Instead of working on the branchdev1 itself, branching helps you to checkout to a new branch branchdev2 where you can work on the new functionality and once the functionality is completed you can merge it back with the branchdev1.

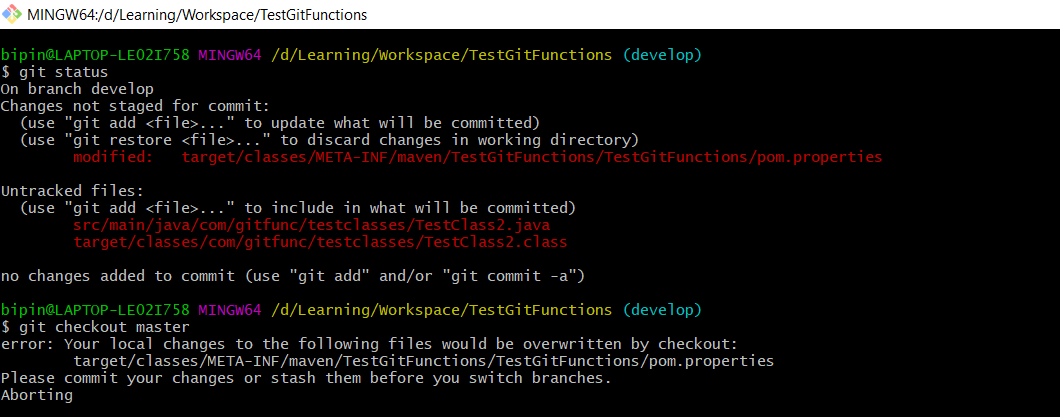
While switching between multiple branches it is important to note that the Working directory will remain only one. When the user switches to the branchdev2 then working directory would show the changes of branchdev2 and once the user switches back to branchdev1 then the working directory would not show the new functionality changes which the user is working on branch2.

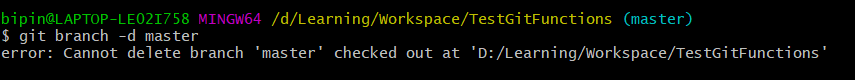
1. **Git branch or Git branch –v** -> these commands will list the branches in the **local** repository
2. **Git branch –r** -> this command will list the tracking branches in the remote repository
3. **Git branch –a** -> this command will list the branches in the local and tracking remote repositories
4. **Git branch <branchname>** -> this command is used to create a new local branch

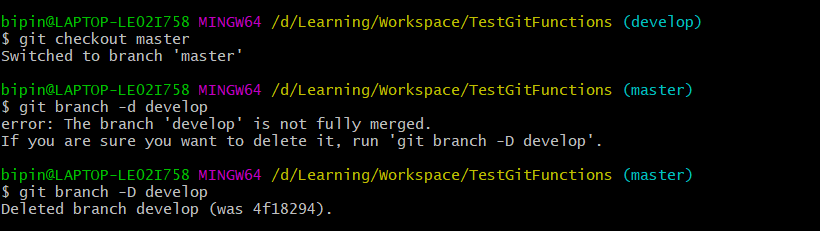
The HEAD pointer points to the master and the develop1\_loginfeature branch, this means that the new branch has all the changes which were present in the master branch. But the working directory points to the master local branch

1. **Git checkout <branchname>** -> this command is used to move from one branch to another
2. **Git checkout –b <branchname>** -> this command is used to create a new branch and move to the new branch
3. **Git branch -d <branchname>** -> this command is used to delete a local branch

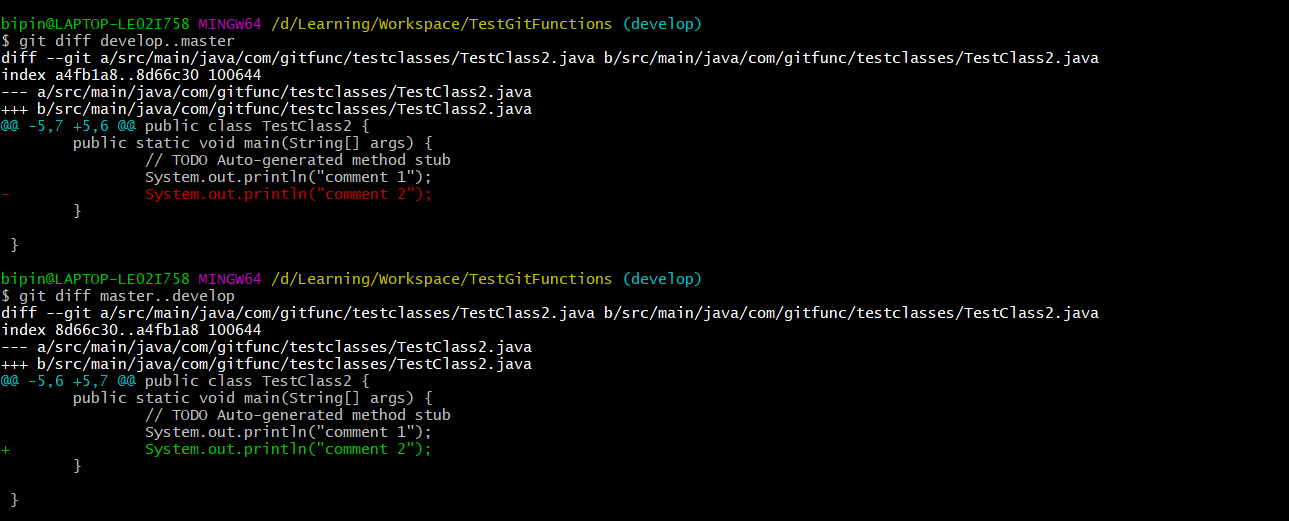
**Note** – sometimes if there are untracked changes in a branch then git will not permit the user to shift to another branch without either of the below 3 options

User must commit the untracked changes / user must remove the untracked changes using the checkout --\* command / user must stash the untracked changes which can be later used

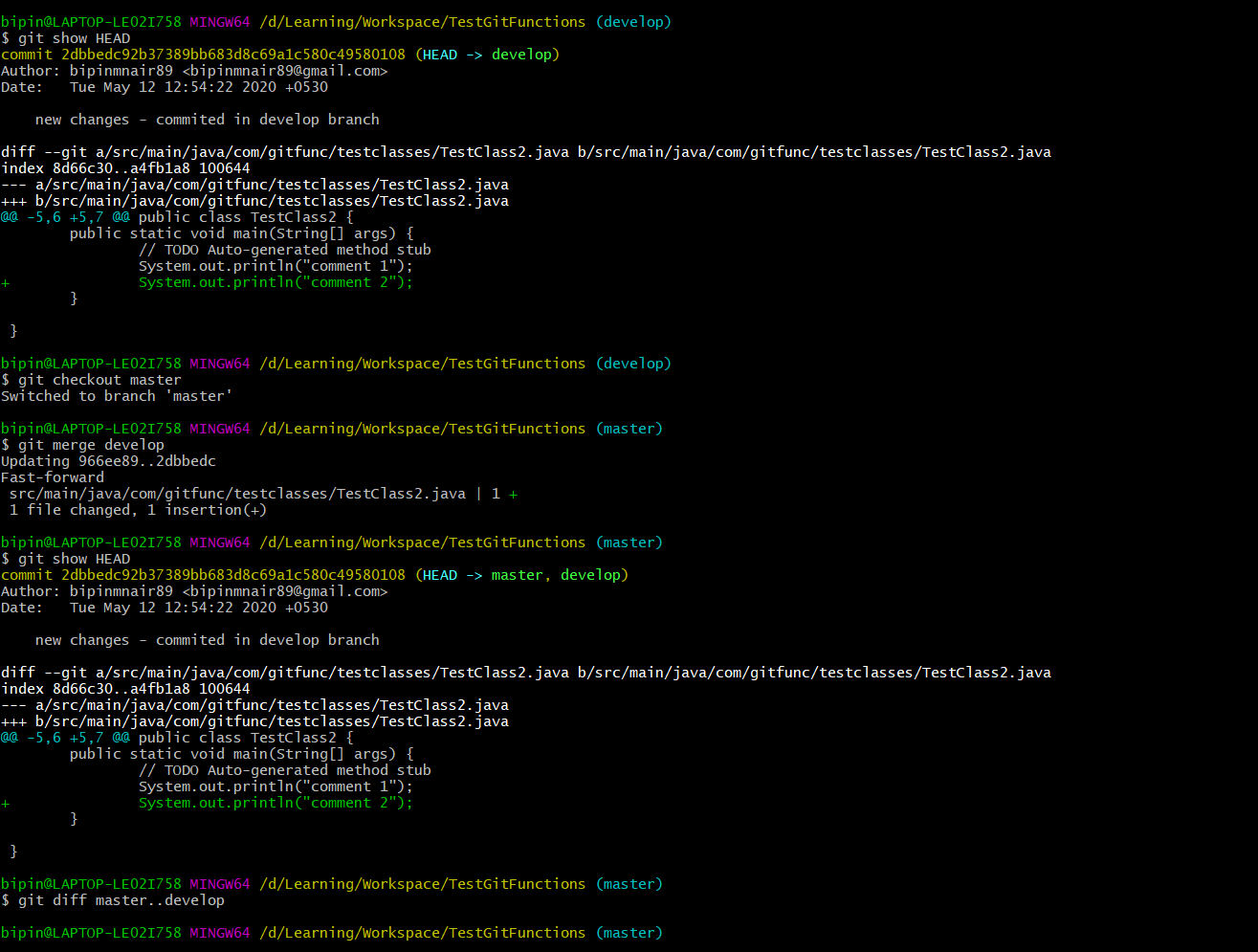
**Note** – Also the user will not be able to delete the branch on which the user has currently checked out

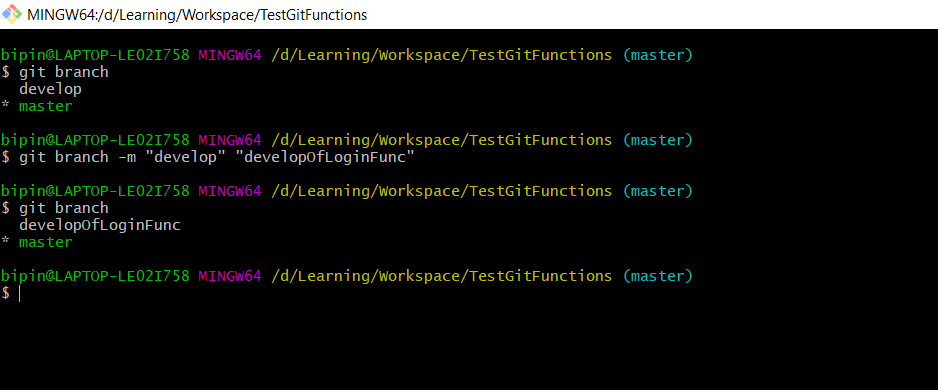
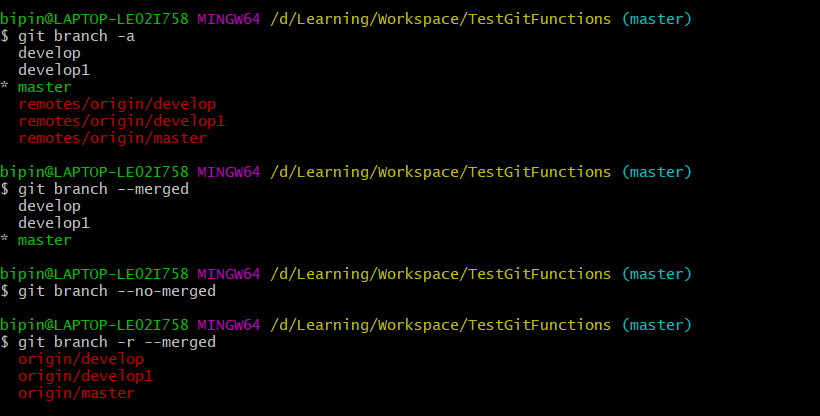
1. **Git branch –D <branchname>** 🡪 this command is used to forcefully delete a local branch when there are changes yet to be merged to master branch and hence –d command doesn’t work and –D comes into place
2. **Git diff branch1..branch2** 🡪 this command is used to compare the changes between branch1 and branch2. This command will show what all are missing in branch2 compared to branch1

The preferred way is to give the old branch first and then the new branch like master..develop

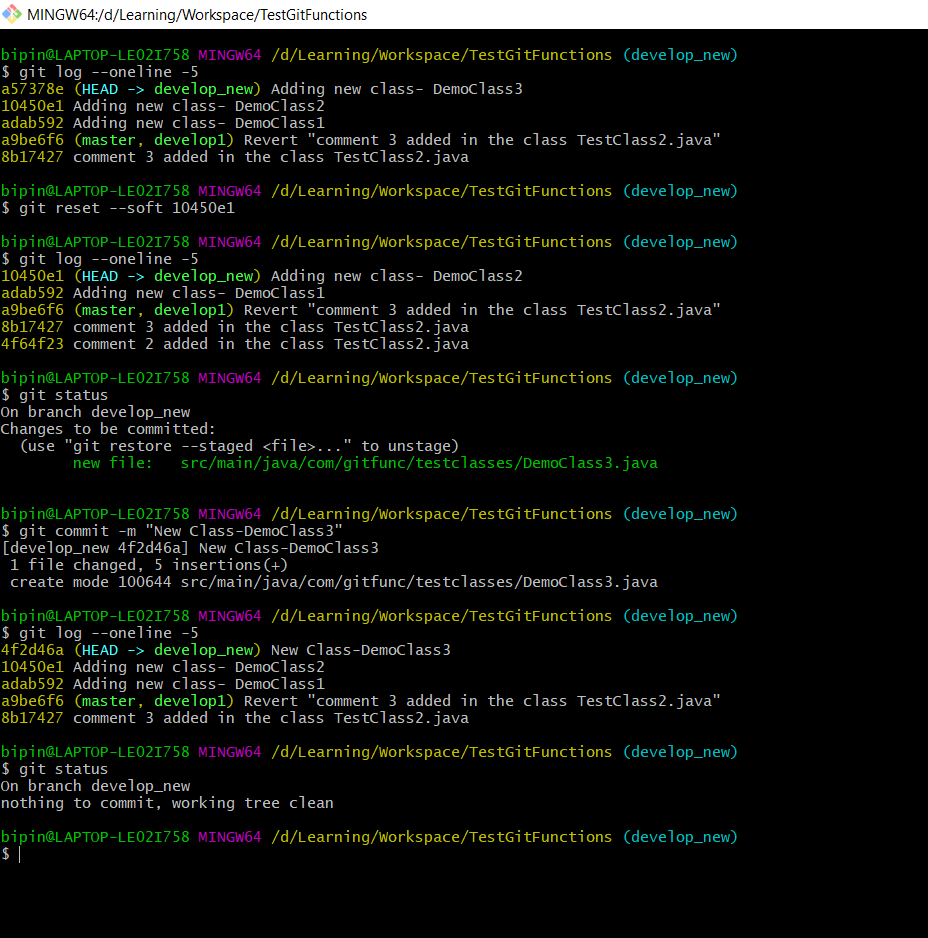


1. **Merging 2 local branches** 🡪 Once all your work is completed in develop branch and its time to merge it back to master branch. Then first checkout to the master branch and then from their do the below command

**Git merge develop** -> this command will merge all the changes of develop to master and later we can do ‘git diff’ command if there are any changes between the 2 branches which technically should be null

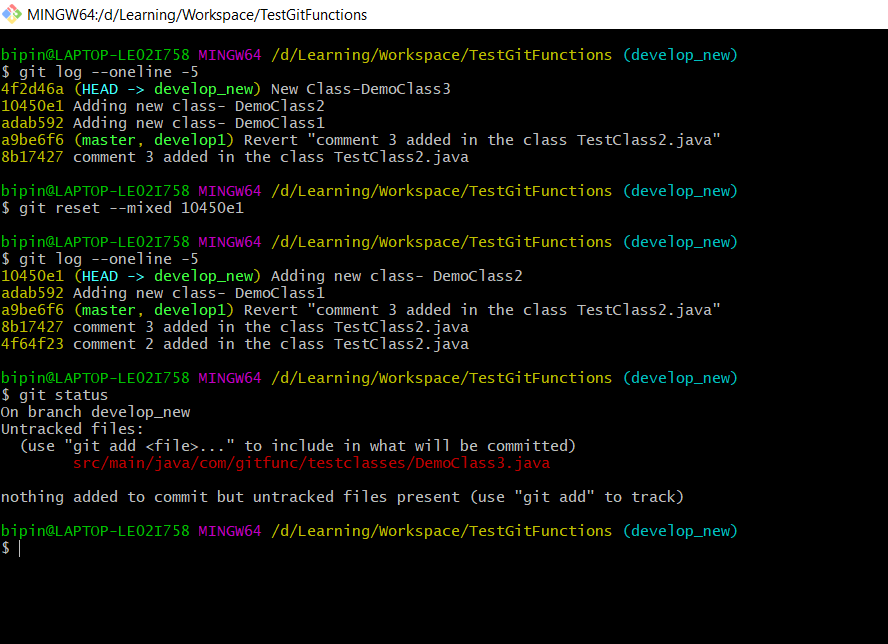
1. **Git branch –m “oldbranchname” “newbranchname”** 🡪 this command is used to rename a local branch
2. **Git branch --merged 🡪** this command will list down all the merged local branches
3. **Git branch --no-merged 🡪** this command will list down all the local branches which are yet to be merged
4. **Git branch –r --merged 🡪** this command will list down all the remote branches which are merged 
5. **Git reset** 🡪 This command is used to reset (rollback) the commits. Primarily it allows us to move the HEAD pointer to a particular location in the <tree-ish>. There are 3 types of reset in Git
6. **Git reset -- soft <tree-ish>** 🡪 This command would reset the HEAD pointer to a specific <tree-ish>.

Staging Area – All the changes in the commits after the specific tree-ish will get moved from the local repo back to the staging area which can later be used or discarded.

Working directory – Will remain unchanged 

1. **Git reset -- mixed <tree-ish>** or **Git reset <tree-ish>** 🡪 This command would reset the HEAD pointer to a specific <tree-ish>.

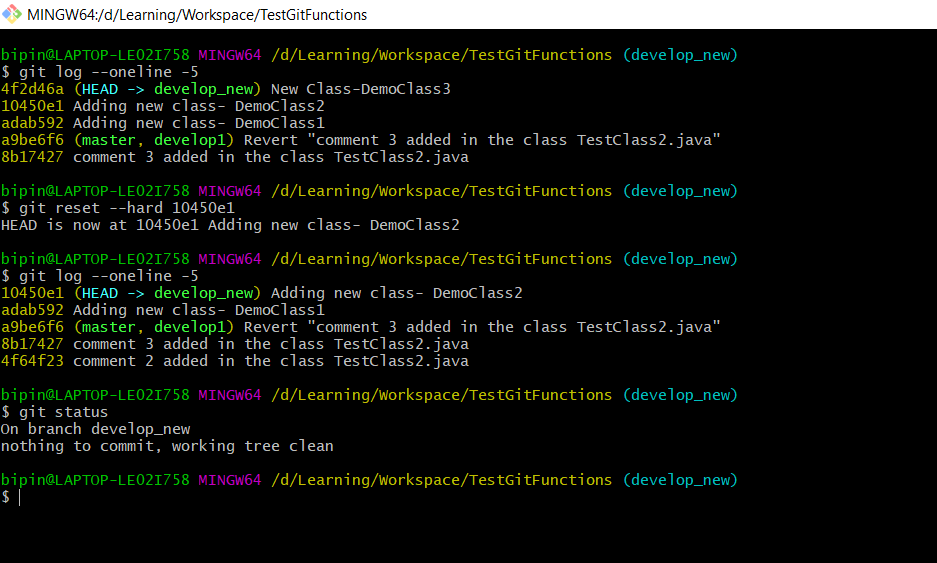
Staging Area – No data would be retained in the staging area

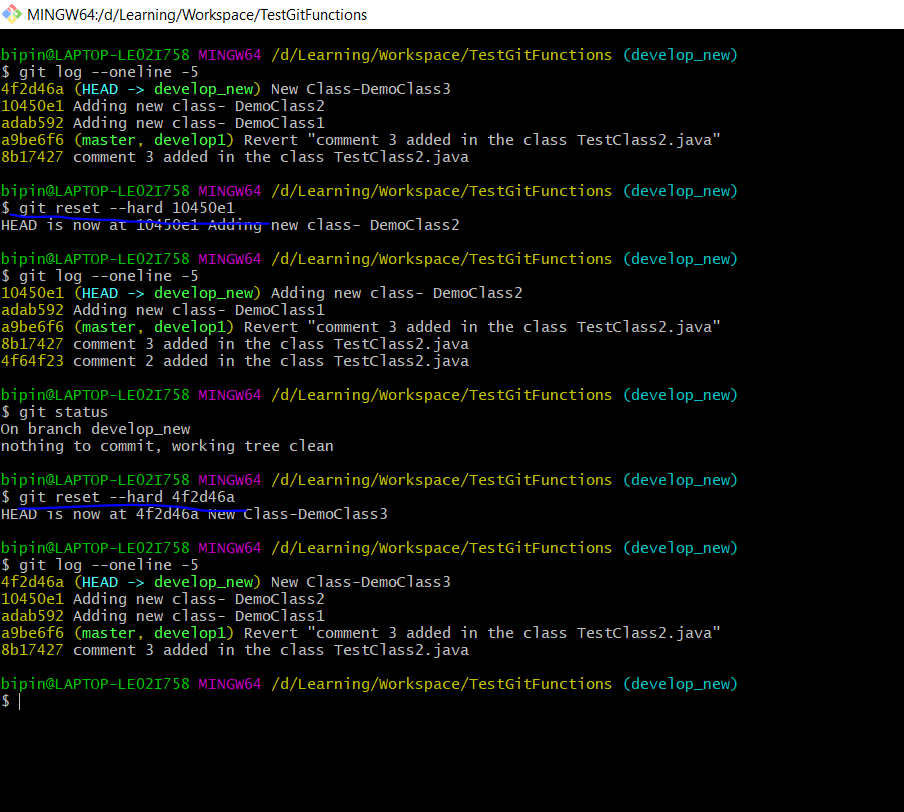
Working directory – All the changes in the commits after the specific tree-ish will get rolled back to the working directory from where we can again add to the staging area or ignore

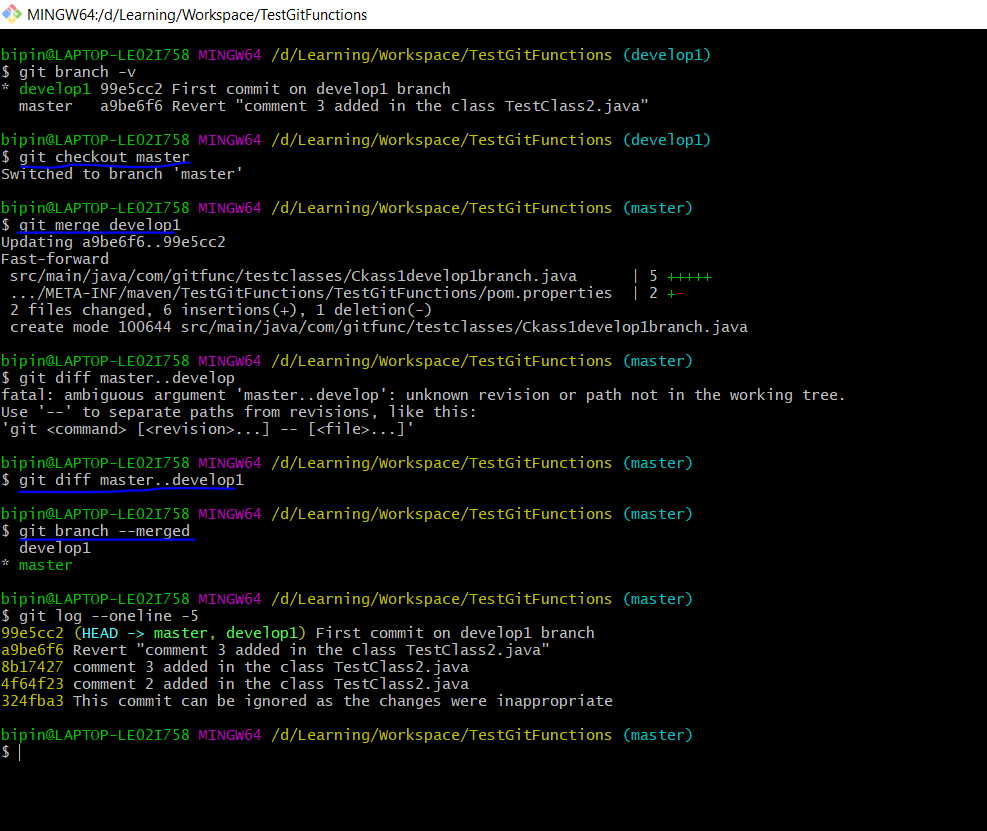
1. **Git reset -- hard <tree-ish>** 🡪 This command would reset the HEAD pointer to a specific <tree-ish>.

Staging Area – No data would be retained in the staging area

Working directory – No data would be retained in the working directory



**Note** : Also note that Git allows us to reset back to the latest commit that we have again after going to a previous commit for a certain specific time until the garbage collector comes into picture.

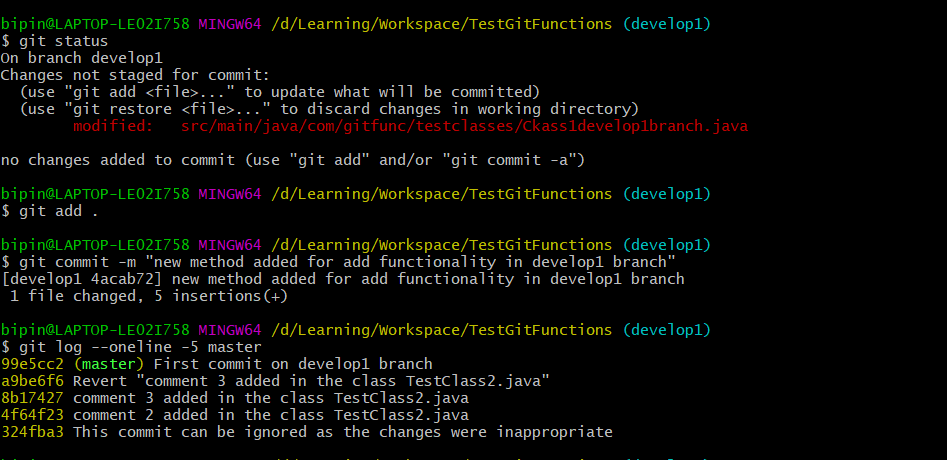
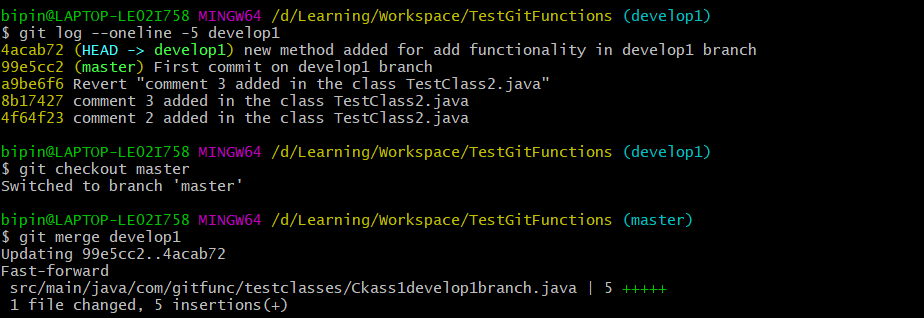
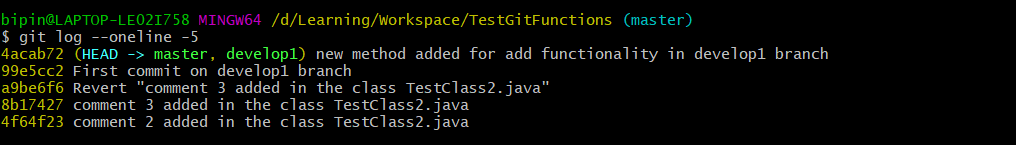
1. **Merging in GIT** -> When merging develop branch to master, it is recommended to first checkout into master branch and then give the merge command. Also it is recommended to have a clean working directory before merging branches so that the complication of merge issues arriving later would be less. 

In the above screenshot after merging we can see that there is no diff between both branches and ‘git branch --merged’ command shows both branches are merged. Also the HEAD pointer now points to the commit which was done in develop1 branch which both master and develop1 branch shares.

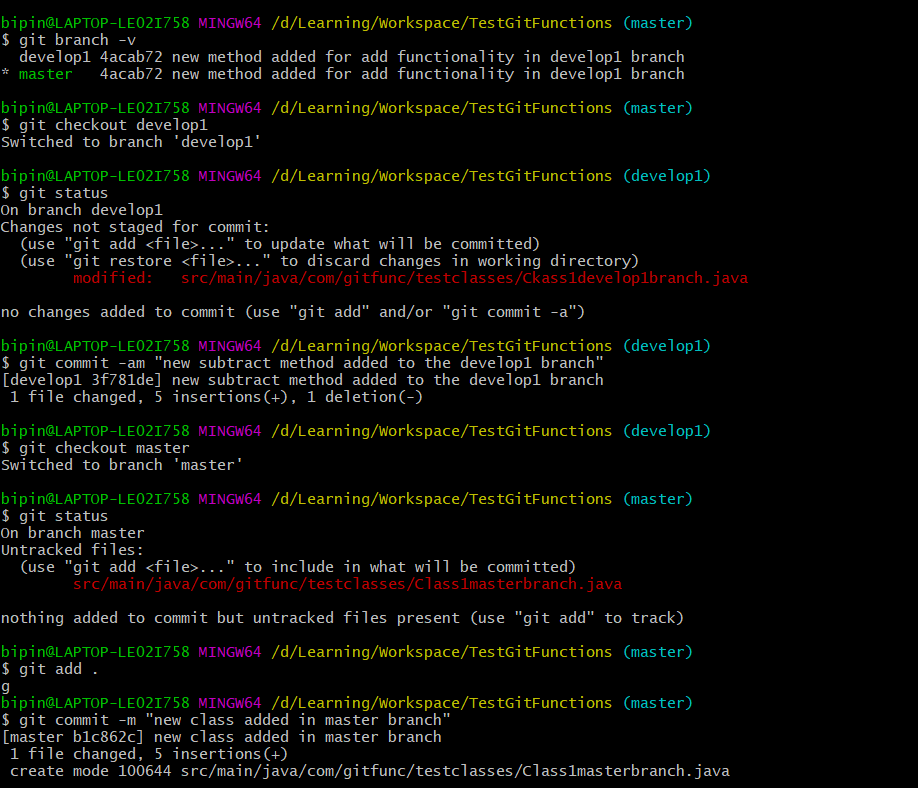
1. **Fast-Forward merge vs True Merge**

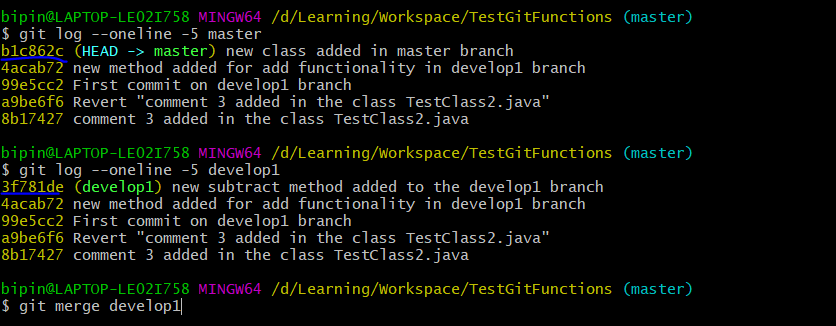
**Fast-Forward merge** 🡪 when develop branch is merged back to master branch but significance here is that the master branch didn’t have any new commits after the develop branch branched off, then the merge is a Fast-Forward merge

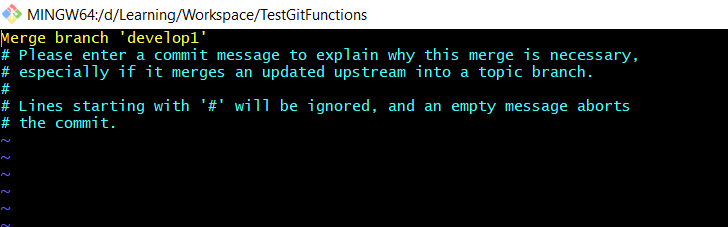
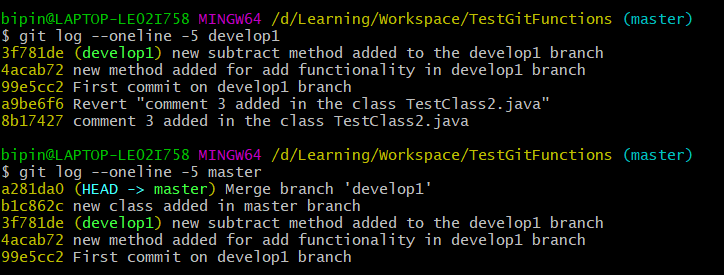
**True merge** 🡪 when develop branch is merged back to master branch but significance here is that the master branch did have new commits after the develop branch branched off, here while merging back the merge itself would be considered as a new Commit, then the merge is a True merge

**Fast forward merge**   

In the above screenshot we can see that while merging it shows that the merge is a Fast-forward merge. And the commit 4acab72 joins infront of the commit 99e5cc2 which was the last commit in branch master before the develop branch branched out.

**True Merge** 

In the above screenshot first both branches are at same point, and there is one commit in the develop1 branch and also later one commit in the master branch 

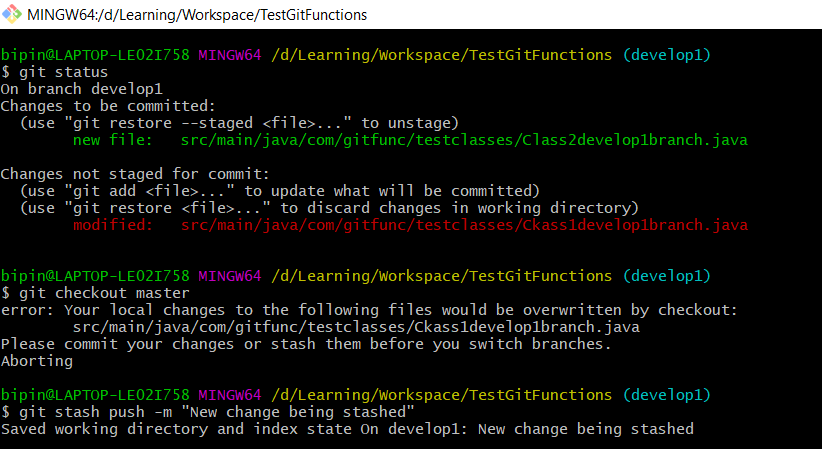
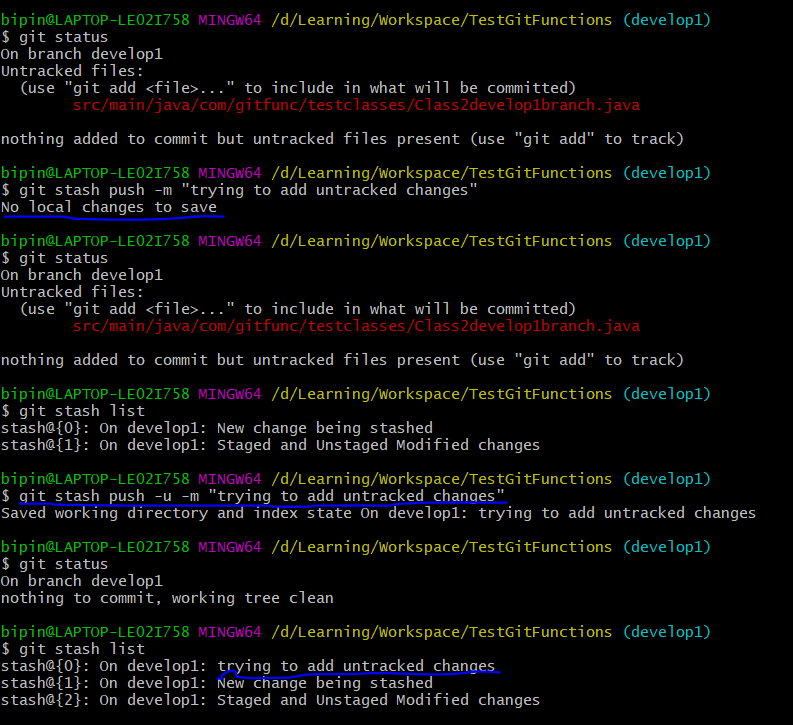
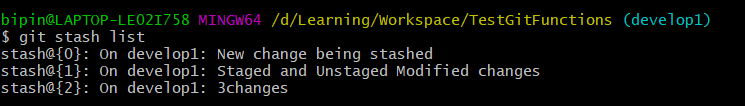
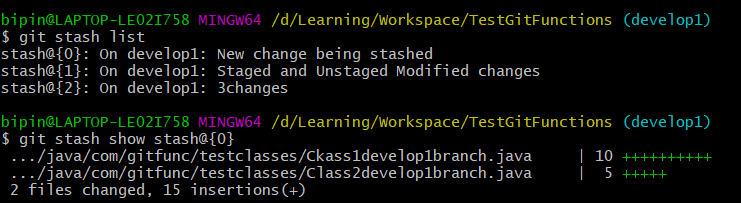
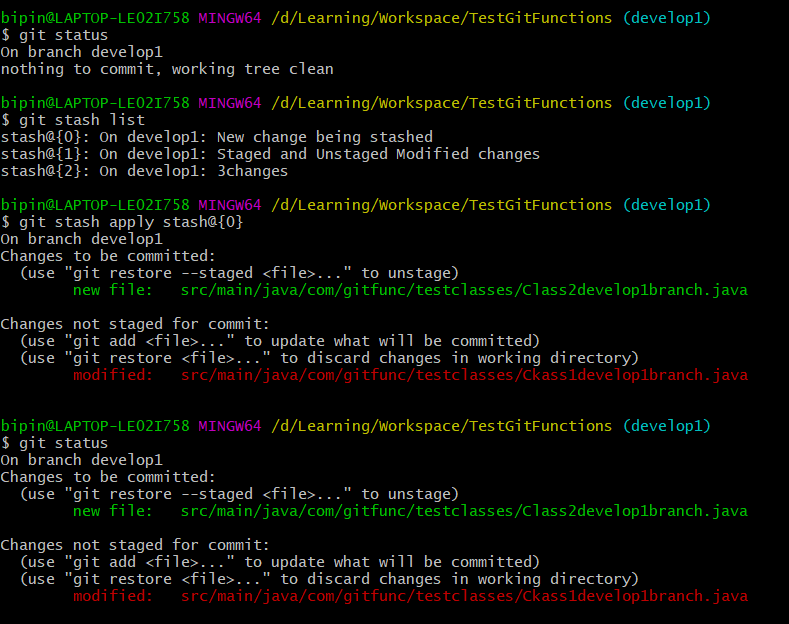
 

Once the merge is made below are the notable changes

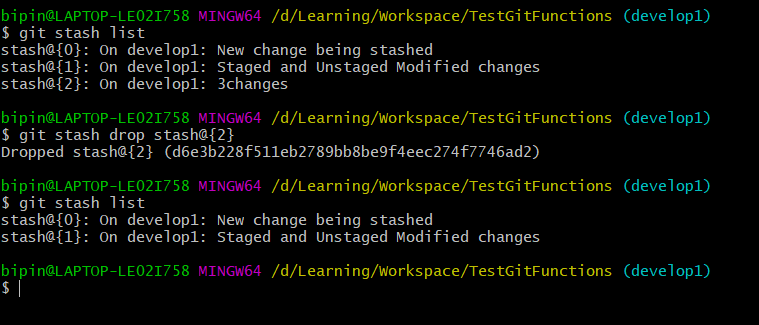
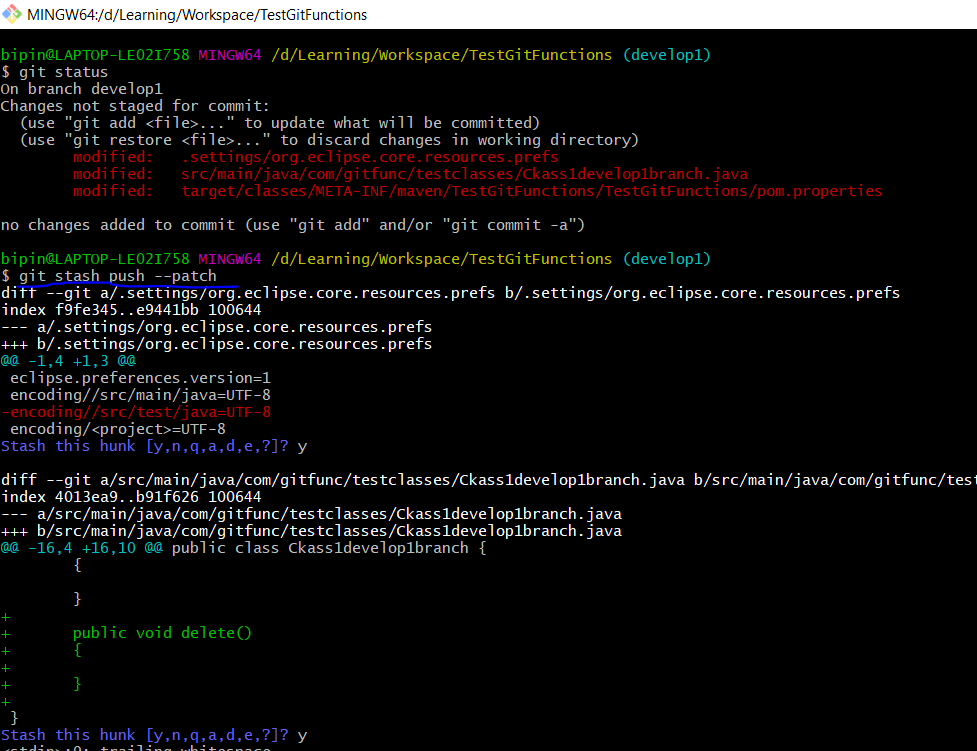
* The HEAD pointer moves to the new commit formed while merging in Master
* The develop branch remains as such with the last commit made there
* The master branch all commits gets added sequentially, the new develop branch commit gets added to the master branch first followed by the change which happened in the master itself followed by the New Merge commit.

1. **Stashing in GIT** 🡪 Stashing is the process of moving the uncommited changes (in stage and in working directory) to a temporary space which can be later retrived and worked upon.

Stashing is required if there is a requirement to switch between branches when there are untracked or unsaved changes which needs to be retained and can be used for a later purpose.

1. **Git stash push –m “<msg>”** 🡪 This command will stash the modified changes in working directory and the staged changes 
2. **Git stash push –u –m “<msg>”** 🡪 This command will stash the untracked changes as well 
3. **Git stash list 🡪** This command is used to list the stashed changes 
4. **Git stash show <stash-name> 🡪** This command is used to list the stashed changes 
5. **Git stash apply <stash-name> 🡪** This command is used to retrive the changes which was stashed 

We can also use **Git stash pop <stash-name>** 🡪 This command will also retrieve the stashed changes but it will delete the entry in the stash list

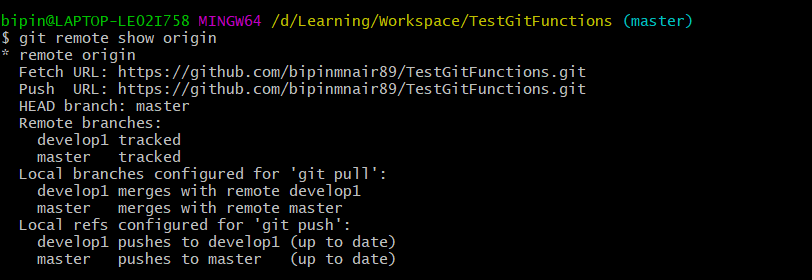
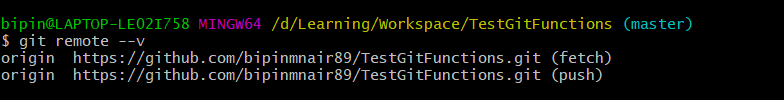
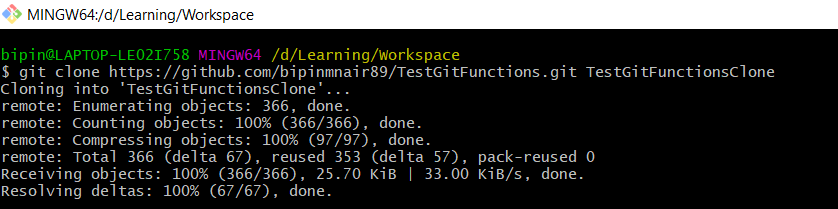
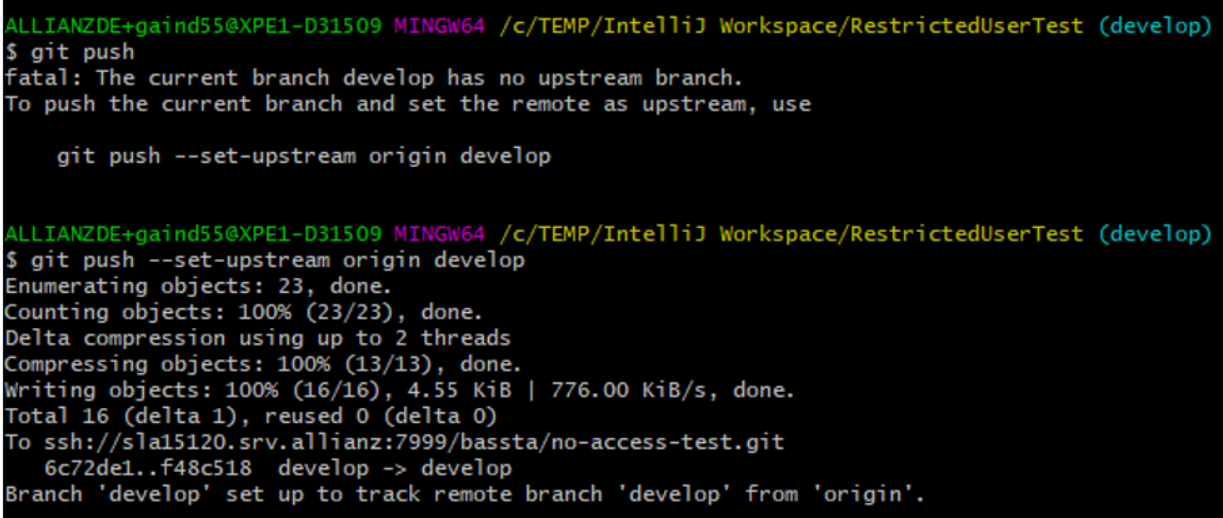
1. **Git stash drop <stash-name>** 🡪 This command will delete the stash entry 
2. **Git stash clear 🡪** This command will delete all the items in the stash
3. **Git stash push --patch** 🡪 This command will give the user the flexibility to choose which changes needs to be added to the stash list 
4. **Remote in GIT** 🡪 Remote is the process of moving the code to a remote server which can be accessed by other collaborators.

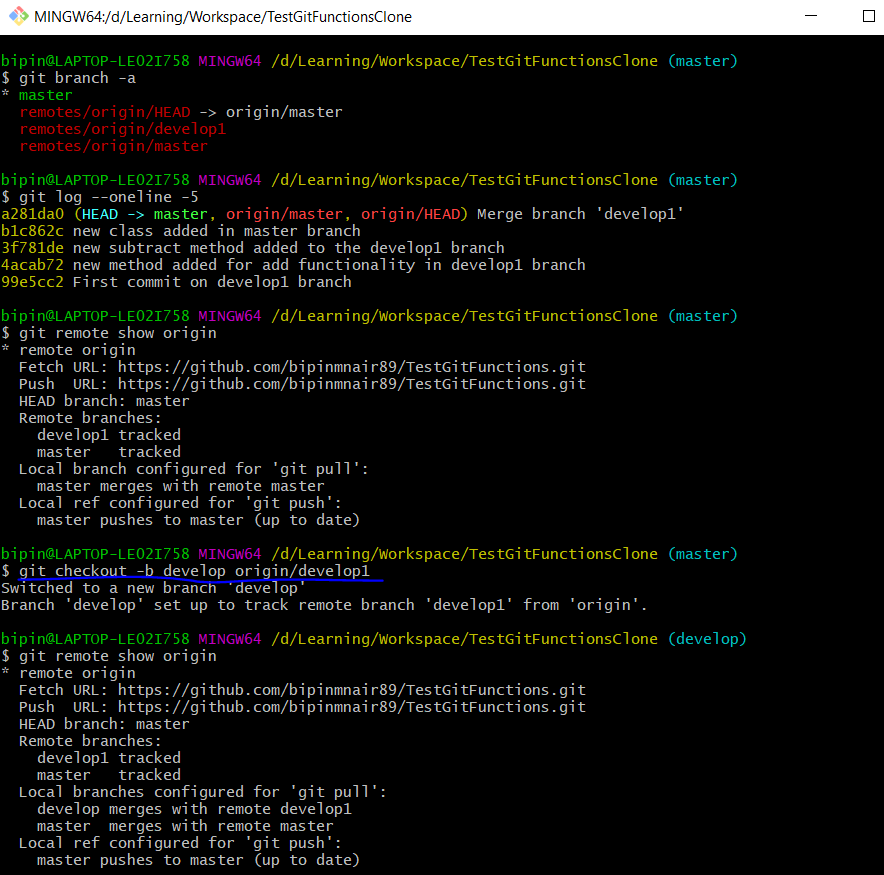
While pushing the code from **develop local branch** to a remote repository, Git will automatically create a new copy of the **develop remote branch** in local and name it as **origin/develop** which will have all the changes that we have pushed as Git is a distributed version control system.

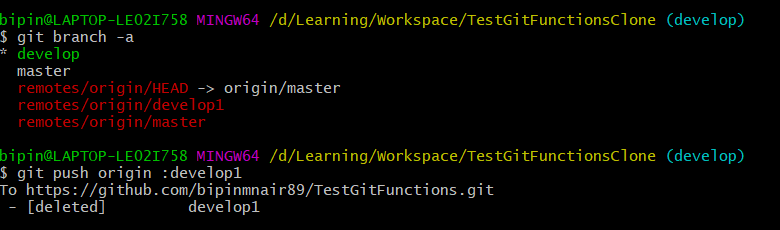
But when there are new changes committed by others in **develop remote branch** then the git **fetch** command will synch those change

s to the **origin/develop** branch in local and the git **merge** command will merge it with the develop local branch. These two processes can be done together using Git **pull** command.

All these above processes are managed by pointers.. just like the HEAD pointer in the local branch the origin/develop branch will also have a separate pointer which tracks the changes of the commits pushed and pulled from remote.

1. **Git remote add origin “url of github project”** 🡪 To add and setup a new remote
2. **Git remote set-url origin “url of github project” 🡪** To change the remote url
3. **Git remote show origin** 🡪 To get the details of the remote urls. 
4. **Git remote --v** 🡪 To get the details of the remote 
5. **Git remote rm origin 🡪** To delete the remote
6. **Git clone “url of github project”** 🡪 to clone the project into the working directory from remote
7. **Git clone “url of github project” <newprojectname> 🡪** to clone the project into the working directory from the remote and also give it a new name. 
8. **Git push –u origin master 🡪** The –u option when used with the push command helps to track the master branch in local with the master branch in remote. Basically it is like setting up a link
9. **Git push –set-upstream origin develop 🡪** This command will help to push the commited changes along with creating the link between develop and origin/develop
10. **Git checkout –b <localbranchname> <remotetrackingbranchname>** 🡪 this command is used to make a copy of the remote tracking branch to a local branch

Ex : Git checkout –b develop1 origin/develop 

1. **Git push origin :develop** 🡪 this command is used to delete the remote branch develop 
2. **Git push origin --delete develop** 🡪 this command is also used to delete the remote branch develop
3. **Git push --force 🡪** this command is used when something horrible has gone wrong in the remote repository and without merging you want to force push your version to replace the changes which are wrong in the remote repository. 