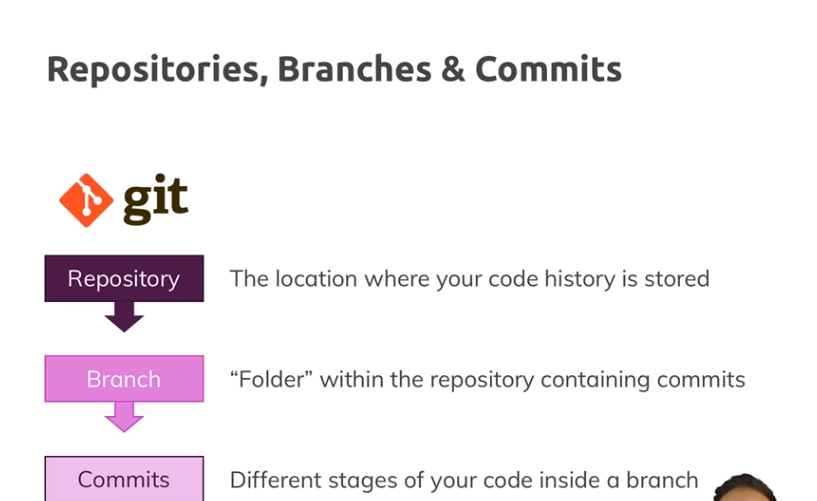
Repository- the location where your code history is stored. We have to tell git that it has to manage certain projects. For that we have to add project files to our repository. Inside such a repository, the code ofcourse is also structured in efficient or logical way, because inside the repository, we have branch. The branch is basically “folder”( we have “” because it is actually not a folder) inside repository which contains commits. Commit stands for different stages, the versions of our source code that we save inside a branch.



So in the end we are able to save our code, lets say we have starting code and ending code, so 2 versions of code in 2 commits. These 2 commits are saved inside a so called branch and this branch is part of so called repository.

Lets see how to place our project under git. Navigate to root directory of your project

1. Run **git init**. This will initialize a git repositorty and will tell git to place this folder under control of git. We can see that .git folder is created now.
2. When we ran git init, we can see msg- **Initialized empty Git repository**. But if we run **git status** we can see that there are untracked files in our folder. we also see red color.so git knows that we have some files in folder. but we have to tell git explicitly, which files to add to our repository, which files we want git to track.
3. Run-  **git add .**(git add dot) .it will tell git to track all files in that folder. we want to add particular files we do this- **git add filename**. Now if we run **git status**, we can see that now we have text -**new file: name of file** , text is of green color. green color means that we have staged this fie(means git is aware of this file and it is tracking it).
4. Run – **git comit –m”**your message**”.** “**-**”is shortcut for doing 2 things. You first tell git that it should commit these changes. So basically add this code right here as a version of code that we kind of want to save in our so called branch and then we simply add **–m**  to give that commit a name. all commits have to be named because this will allow us later to easily see what exactly changed in this commit that we did. Lets run this-

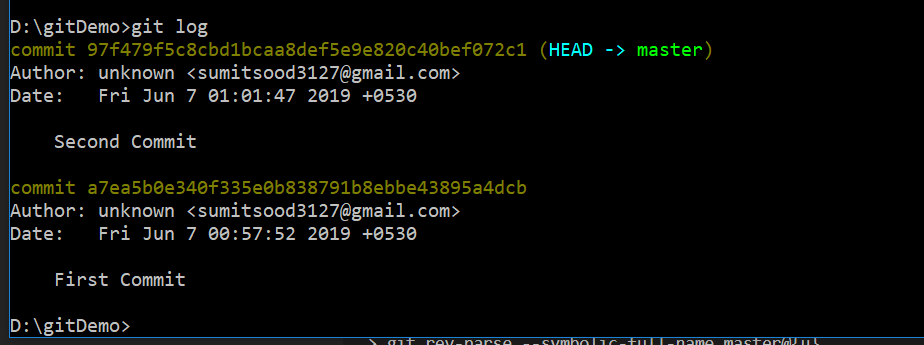
**git commit –m”starting code”**

if we run  **git status** , we will get msg nothing to commit.

5)if we run **git branch.**  You can see that git automatically created master branch. This is first thing that you have to keep in mind from structural perspective. When we ran git commit, we committed our code and added it to our repository and with automatically created this master branch. So this commit is now saved inside master branch inside our repository in this folder.

Lets say we make a change in file. now if we run **git status**. It will show us modified files in red color. now we have to add files , that we want to add to this commit.so run **git add . ,**  it will include all files. Now run **git commit –m”div added”** . so now we also added this new right here to our repository, to our branch and we also(that’s important) saved these 2 snapshots.

6)if we run **git log**. We can see our 2 commits. With comments that we mentioned.



**commit de32fb1ccc1dfffc54f9fcc3005d1cf229d0586c (HEAD -> master)**

**Author: unknown <sumitsood3127@gmail.com>**

**Date: Sat Jun 2 14:53:17 2018 +0530**

**div added**

**commit 952d440db5ad58af5f2bdadd4c516e331ea1df6a**

**Author: unknown <sumitsood3127@gmail.com>**

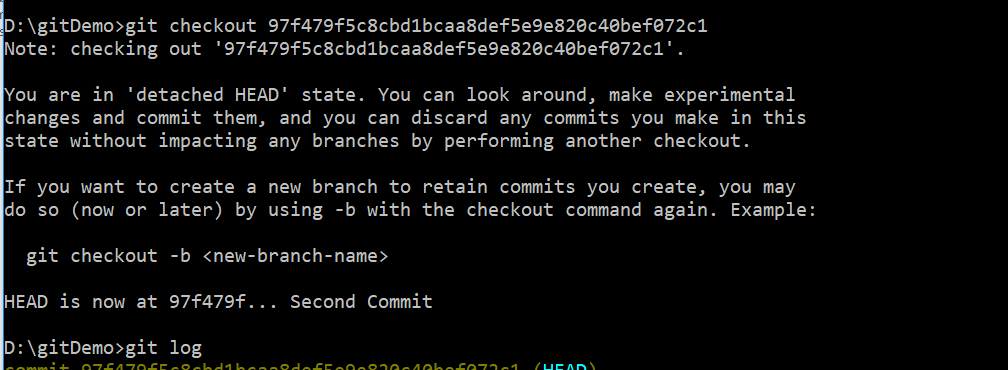
**Date: Sat Jun 2 14:13:23 2018 +0530**

**starting code**

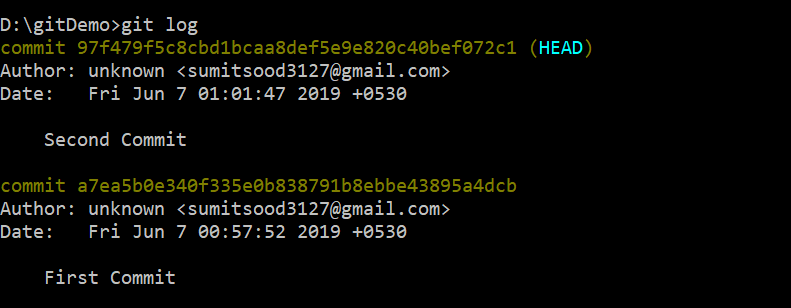
before we continue one important information. Git does not create copies of each commit that you create here. Git simply tracks changes, we saw that in **git status.**  So when we add **git add**, git has a look at our different files in the code and sees if we changed something and if we commit that in the end, these changes are saved but git is not creating copies of code again and again. So git is very smart.

If we have look at our commits, we can see that in last commit we have Head -> master written. Important part is head. Lets make new change in our file and commit that change. Then if we run **git log**, then only last commit has Head->master written on it. So last commit that we made is new head. So this is important, whenever you create a new commit, this last commit will be head in your branch. Why this is important? Lets say you want to go back to previous commit. Lets say commit with comment- div added. So we copy id of commit and run-

**git checkout idOfCommit**

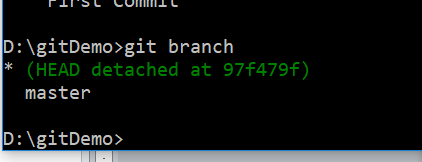


now lets say we have migrated to commit 2.now we are back at our old commit. It means we will have code that we had when we performed this commit. Now head is also at this commit but , that’s important, this commit is no longer part of our master branch.



We can see that head is at second commit and we cannot see third commit.

And below we can see that we are no longer in master branch.



Basically this commit is in middle of no where. You can use checkout feature to have a look at previous code. Now lets say you want to go back to this commit. That is you want to undo all commits that are done after commit2. Note now you are in this commit just for viewing purpose. You can see this message-

**D:\testGit>git checkout 626cbf8917b9c672ac4c2a4abb4ae2af6e9f317c**

**Note: checking out '626cbf8917b9c672ac4c2a4abb4ae2af6e9f317c'.**

**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 performing another checkout.**

**If you want to create a new branch to retain commits you create, you may**

**do so (now or later) by using -b with the checkout command again. Example:**

**git checkout -b <new-branch-name>**

**HEAD is now at 626cbf8... kl**

now if you run **git branch**, you will get this-

**D:\testGit>git branch**

**\* (HEAD detached at 626cbf8)**

**Master**

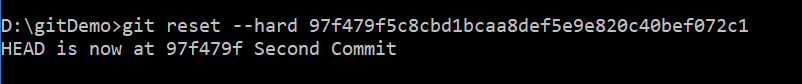
So you can see that we are in kind of new branch.

so how to go back to previous state? First you need to come to your current commit. run-

**git checkout master**

now you can see by running  **git status** that head is at last commit that was done in master branch. Now we want to go back to second commit, so run this-

**git reset –hard commitID**



commitID is id of commit where you want to go. Now if you run **git log**, the you can see all commits before the commit that we migrated to all gone. So now we have successfully migrated to a previous commit. now head is at commit where we have migrated. It means we have deleted commit. so this jump is permanent. We cannot go back to commit that we have deleted. That was one approach.

To see more about git reset-

<https://gist.github.com/tnguyen14/0827ae6eefdff39e452b>

ousing git reset we can discard staged chnage

6)Now lets say we want to migrate to previous commit, make some changes. Now we want these changes to persist, but we also want to continue on master branch. For this you need to create new branch. Here is how it can be done. Lets say we are in master branch.

Lets say we have 3 commits in master branch. First go to second commit by-

**git checkout "idOfCommit"**

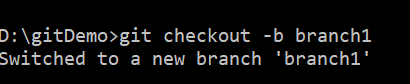
now you are in detached state(as explained above). Make any changes, stage them and commit them.

**D:\git\_Test>git commit -m"This is commit in previous commit"**

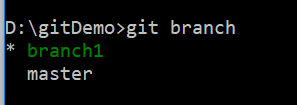
**[detached HEAD 7a247e4] This is commit in previous commit**

**1 file changed, 4 insertions(+)**

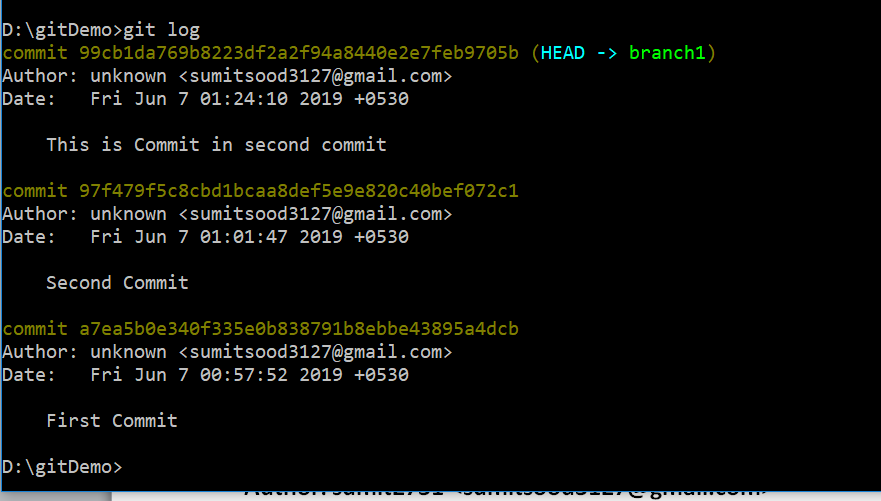
Now you want these changes to persist, but you also want your third commit to persist. So what can be done is you can have changed done to previous commit in a separate branch.



Now if you run **git branch**, you can see that-

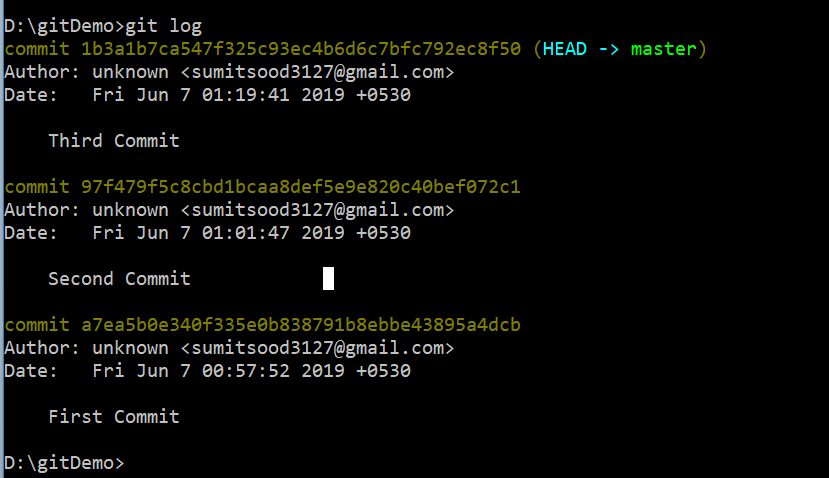


It means we are in branch1. If you run git log in branch1, you can see 2 commits of master branch, after that you will see this commit that you have done on previous commit.



Now you can go back to master branch by –

**git checkout master**



you will see code where you exactly left it. If you want to go back to bracnh1 , you can do that by-

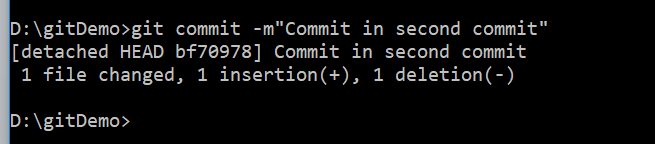
**git checkout branch1**

note that if you push your code to github, branch1 will not be pushed into github.

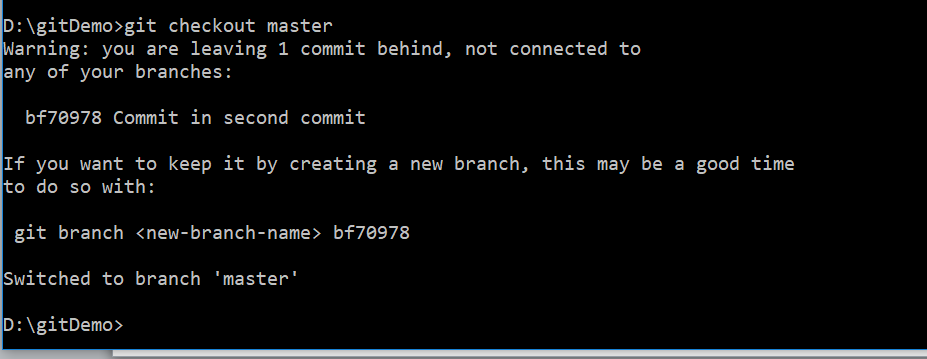
That is one approach.

Alternative approach can be-

After running this-

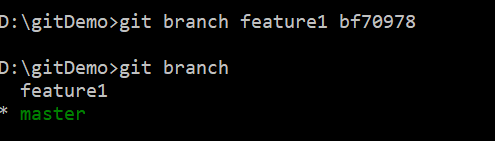


We have not created new branch, instead we moved to master branch by –

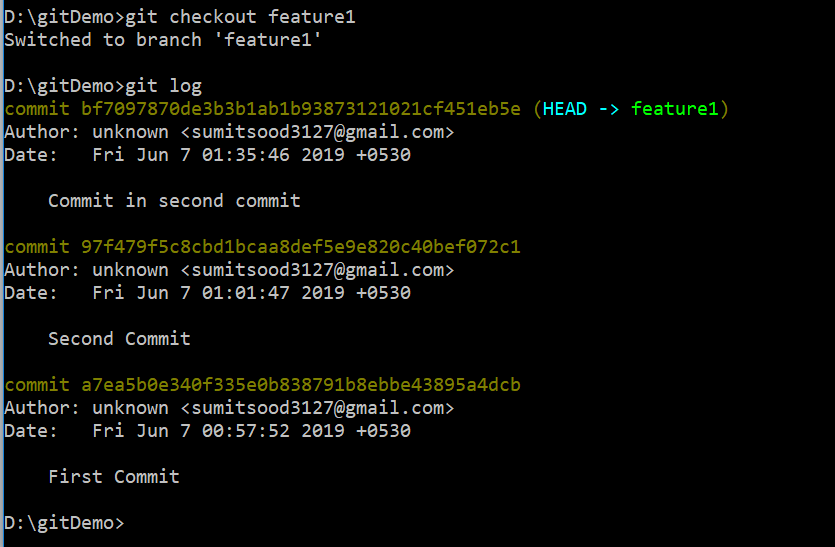


Now commit made in second commit will be lost. And we are in back in master branch. But if we run-

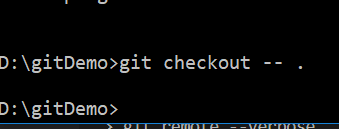
**git branch <new-branch-name> 9d18ff7**



then a new branch will be created, it will contain changes that we have made in second commit



7)Now lets say after our last commit, we are working in our project. Something went wrong and I want to go to code that I have last committed. note that changes were not staged(we have not included files by **git add** ). so what we want to do is to revert back all unstaged changes and go back to our last commit.



if you want to discard unstaged changes of particular file, run this-

**git checkout <name of** file**>**

see this link-

<https://dev.to/neshaz/when-to-use-git-reset-git-revert--git-checkout-18je>

with this we covered all core functionalities of commit area, there is lot more to dive into, but to get started this is basically everything you need to play around with commits.

But we also had this branches. now we will see branches. If we run-

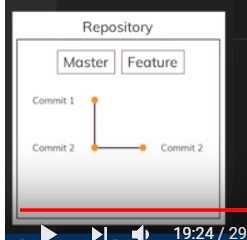
**git branch**

we will see master along with \*. \* means currently we are in this branch. This master branch was created automatically when we ran **git commit**  for first time. branch basically contains the code that we have, all the different commits, all the different versions of it.

Lets see what is use of branch. Lets say we are working in master branch and we have deployed website. It is deployed and working fine. But now we want to add new feature. Now we do not want to crash the actual web site. Then you can also create the another branch for a specific feature. Now you can continue working in both branches. You can fix some bugs in master branch and at same time you or some other coder can implement that new functionality in that separate branch. after that new feature is completed you can merge both branches. So lets try it out. Lets create a new branch.

**git checkout –b new-feature**

here we have created a new branch named new-feature. Now if you run **git branch** you can see all branches(master and newly created branch)and you can also see that you are in newly-created branch(indicated by \*). If you run **git log**.yo can see that this newly created branch has all the commits that we had in master branch.



Beside last commit we can see this –

**(HEAD -> new-feature, master)**

This means it is head for both master and newly created branch. Lets add a new file-style.css, to this newly created branch. Lets commit this file with comment – css added. If we run **git log**, we get-

**commit 67f7cf935a3608b26b691cf9e839e5265ec03748 (HEAD -> new-feature)**

**Author: unknown <sumitsood3127@gmail.com>**

**Date: Sun Jun 3 11:35:35 2018 +0530**

**css added**

**commit ebc7a28ee93d796a14480dcff4664e1a320062b0 (master)**

**Author: unknown <sumitsood3127@gmail.com>**

**Date: Sat Jun 2 19:29:43 2018 +0530**

**div added**

**commit d6a31b3ddcee0a0b1b02c734ad3ecfdc3c82b189**

**Author: unknown <sumitsood3127@gmail.com>**

**Date: Sat Jun 2 19:27:57 2018 +0530**

**initial code**

it means for newly created branch our last commit is head but for master branch head is second commit. that is why we can see **(master)**  after second commit. it means this commit is head in master branch. This new commit is only added to new-feature branch.

Lets go back to maser branch. To change branches you have 2 options. First is-

**git checkout branchName**

for master branch you can use this shortcut- **git master .**  as we switch to master branch we can see that master branch css file is not there. So lets merge these 2 branches to make sure we have new feature in master branch also. To do that we need to be in master branch which we are. so run-

**git merge new-feature**

now we can see that all changes of new feature branch are in master branch. Of you run **git log**  you can see that last commit is head of both master and new-feature.this is case only when we have no merge conflicts. In this case all commits that were added in new branch, will be added to master branch.

But if we have added a new file in master, it wont appear in new-feature. So new-feature is unchanged.

Now we do not need new-feature branch. So we can delete it-

**git branch –D new-feature**

we can confirm it by running **git branch** , we can see that only branch that we have is master.

So this is how we can use branches.

There is one more thing I want to show you. Lets create a new branch-

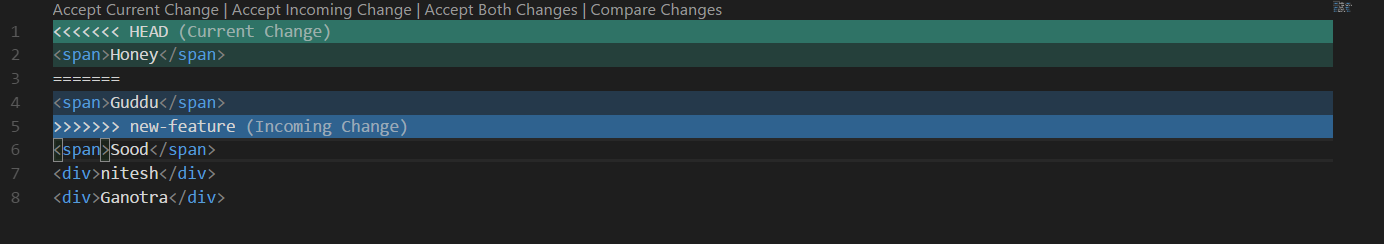
**git checkout –b new-feature**

this will create a new branch and we will migrate to that new branch.in new branch we will replace sumit with guddu in index.html. then commit this change. Now go back to master branch. Here we can see sumit text.now in masterbranch, replace sumit with honey. It is important that you change the same element. Commit your change.

Now question is what happens if I try to merge these branches? Os run-

**git merge new-feature**

we can see that git has identified conflict-



Normally git tries to solve these conflics on its own but in this case how will git know what to do. on my ide I can see optons like – accept current change.(to keep master code), accept incoming change(to use new-feature code), acccept both changes(to have both changes), compare changes.

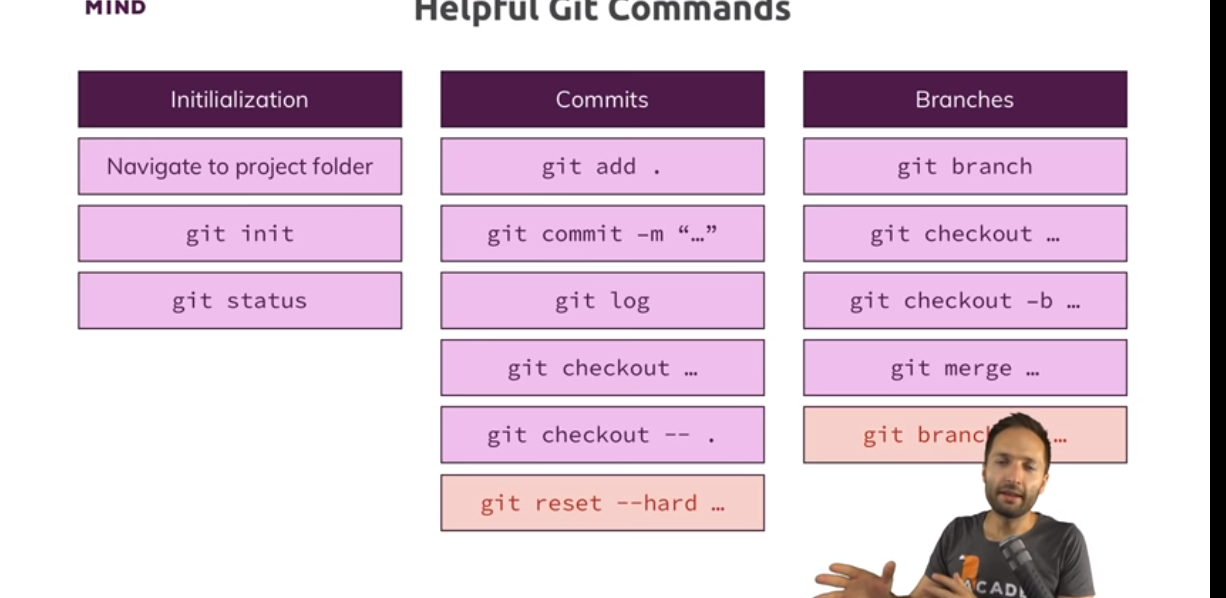
Now lets say I do not want to rely on my code editor. You can also manually delete changes ,lets say we want to keep guudu(new-feature code). So delete all extra code. So that’s it , we have merged it. Commit your change name it conflict resolved. Now if we do

**git log**

in master branch we can see that, all master branch commits, then commit of feature1 branch and finally commit in which we resolved the conflict.

Now lets say in master branch we have text sumit, now we create new branch i.e branch1 from master. In branch we replace sumit with sood. When we will merge our branch1 with master, git will automatically replace sumit with sood without giving conflict,this is because mater branch is not changed after we created new branch. Moreover after merging if we do **git log** in master branch, we can see commits that we performed in branches, they will be sorted by time in which they were performed.

Overviews of command-



To dive deep into git, go to officials docs.

git checkout –ours, git checkout theirs

<https://dev.to/willamesoares/git-ours-or-theirs-part1-agh>