1. mkdir deeDemo (Creates a folder named deeDemo)
2. cd deeDemo (goes inside deeDemo)
3. git init ( added a new .git directory means it creates a git repository in short repo)
4. Config user name and email –
5. Global: -

git config --global user.name “Deepak Arora”

git config --global user.email [deepak.arora@milwaukeetoolc.om](mailto:deepak.arora@milwaukeetoolc.om)

or

1. Local: -

git config --local user.name “Deepak Arora”

git config --local user.email [deepak.arora@milwaukeetoolc.om](mailto:deepak.arora@milwaukeetoolc.om)

1. git config –list (shows us the that our name and email are set
2. vi file1 (Add a file named file1 in folder.

Put some lines

Esc 🡪 :wq

1. git status
2. git add file1
3. git status
4. git commit -m “added file1”
5. git status
6. git log (shows commit graph)
7. cp file1 file2 (Add another file in folder file2 by copying file1)
8. vi file2 (Edit file2 to make a difference from file1, then save and exit)

add a line at the start Line0

Esc 🡪 wq

1. vi file1 (Edit file1, then save and exit)

add a line at the start Line0

Esc 🡪 wq

1. git status
2. git diff (shows the difference between tracked files in the working tree and the file in staging area)
3. git add file1 file2 (To add both files in staging area)

or

git add . (To add all modified and new files in staging area)

or

git add file\* (use of wild card character)

1. git status (working tree and staging area are on same page)
2. git diff –staged (shows the difference between files in staging area and most recent commit or in other words it shows what we are about to commit)
3. git commit -m “added file2 and edited file1”
4. git log (most recent commit is at the top)
5. git log -p (shows what changes has made with each commit)
6. git rm file2 (It does two things remove file from working tree and remove file from staging area as well)
7. git status (deleted file2)
8. git commit (if we don’t give -m than it will take us to default file editor and we can put multiline commit there)
9. git log (most recent commit is at the top)

**Undo a working tree change**

1. vi file1 (Edit file1)

add a new line at the end Line4

1. Esc 🡪 :wq
2. git diff (shows the difference between tracked files in the working tree and the file in staging area)
3. git status (file1 is modified but not staged)
4. git checkout -- file1 (discard the changes, notice the space after ‘--')
5. git diff (this time it will display nothing because working tree and staging area matched)
6. git status
7. more file1 (display content of file1)

**Undo Staging a file**

1. vi file1 (Edit file1)

add a new line at the end Line4

1. Esc 🡪 :wq
2. git diff (shows our changes between tracked files in the working tree)
3. git add file1 (stage all the changes)
4. git diff (this time it will display nothing because working tree and staging area matched)
5. git diff –staged (shows the difference between files in staging area and most recent commit or in other words it shows what we are about to commit)
6. git status (shows file1’s latest changes are staged and ready to commit)
7. git reset HEAD file1 (To un-stage, working tree still has modifications; here HEAD refers to the most recent commit in History)
8. git status
9. git checkout -- file1 (undo working tree changes also ☺, notice the space after ‘--')
10. git status

**Restore file from earlier commit**

1. git log -- file2 (show commits which affected file2, notice the space after ‘--')
2. git checkout <first five char from commit id from above command output> -- file2 ( to get back file 2 from snapshot with given commit id)
3. ls (shows the list of files)
4. git status (shows that the file2 is not only back in working tree but also in staging area)
5. git commit -m “restored file2”

**.gitignore**

1. Create some files and folders

touch myapp.pyc (create a file named myapp.pyc)

mkdir logs (create a dir named logs)

touch logs/log1.log (create a file named log1.log)

touch logs/log2.log (create a file named log2.log)

1. git status
2. create .gitignore file and add details (we can put whole file name or we can use wild card charectors e.g. myapp.pyc or \*.pyc entire directory use “logs/”)

vi .gitignore

\*pyc

logs/

Esc 🡪 :wq (to exit vi editor)

1. git status (myapp.pyc and logs folder are not displayed, but .gitignore is)
2. git add .
3. git status
4. git commit -m “added .gitignore file”

|  |  |  |
| --- | --- | --- |
| Working Tree  File1  File2  .gitignore | Staging Area  (index)  File1  File2  .gitignore | History  (Commit graph)  . |

**Introduction to Git - Branching and Merging**

1. mkdir deeDemo
2. cd deeDemo
3. git init
4. vi file1

line1

line2

line3

Esc 🡪 :wq

1. git add file1
2. git commit -m “created file1”
3. cp file1 file2
4. git add file2
5. git commit -m “created file2”
6. git log ( to see our two commits)
7. git status (it says we are on master branch, master branch gets created automatically by git)

by default master branch is created by git.

Master is pointing to the latest commit.

Head is a pointer that normally points to our branch, so far we have only first branch i.e. the master branch therefor Head is pointing to that. Since Head usually point to our branch and not directly to a commit, it is sometimes called a symbolic pointer, in git terminology the Head pointer tells us what we have checked out, so right now we know that we have the master branch checked out



1. git log --all --decorate –online --graph (shows we are on the second commit due to the HEAD pointer, The HEAD pointer is pointing to the master branch, this tells us that we have master branch checked out. This is a usefull command so we can create a command alias to shorten it.)
2. alias graph=”git log --all --decorate –online --graph”
3. graph (same output as 12.)
4. git branch dev (creates a new branch dev from where there HEAD pointer is pointing)
5. git branch qa (creates a new branch qa from where there HEAD pointer is pointing)
6. git branch (shows all the branches, \* and green font color shows that branch is currently checked out branch more precisely the HEAD pointer is pointing to that branch)
7. graph (all three branches are pointing to the same commit, HEAD is attached to master branch since we have master is checked out)

Checkout and work on branches

1. git checkout dev (moves us to dev branch)
2. graph (it shows that HEAD has moved and it now points to dev branch)
3. git status (also show the same as above “On branch dev”)
4. vi file1 (edit file1, add line 0)

line0

Esc 🡪 :wq

1. git add file1; git commit -m “dev for file1” (stage and commit)
2. graph (shows our updated commit graph)

our dev branch has moved, so our reference our newest commit and HEAD stays with it

1. cat file1 (to view the file1’s content, it show the line0 which we added)
2. git checkout qa (moves us to dev branch)
3. git branch (shows all the branches, \* and green font color shows that branch is currently checked out branch more precisely the HEAD pointer is pointing to that branch)
4. graph ( we can see how the checkout command moved the HEAD pointer, it left the dev branch and now attached with qa branch)
5. cat file1 (to view the file1’s content, it show the line0 which we added has gone, git replaced our working tree and staging area to match the commit of associated with, at this earlier commit we don’t have line0 in file1, to see that again use below)
6. git checkout dev
7. cat file1 (it revels that we have line0 back, git updated our working tree and index again to reflect this commit)
8. git checkout qa (back to the qa branch)
9. vi file1 (Edit file1 and add line4 at the end)

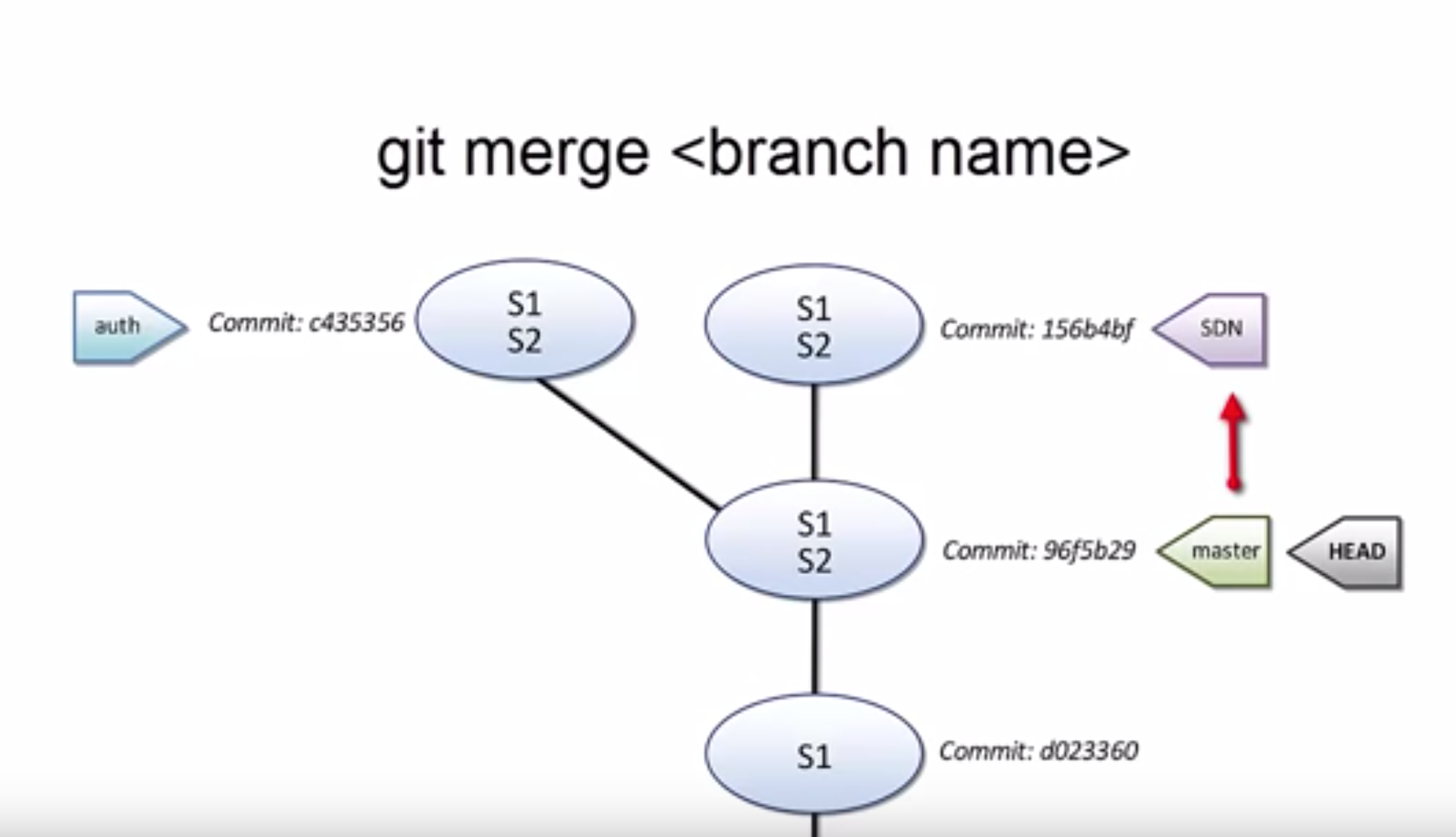
line4

Esc 🡪 :wq

1. git status (shows that we have modified file1 in the working tree)
2. git commit -a -m “qa for file1” (shortcut ‘-a’ is for stage and commit)
3. graph

we are done with our work withing dev and qa branch – now time to merge

Fast-Forward merge



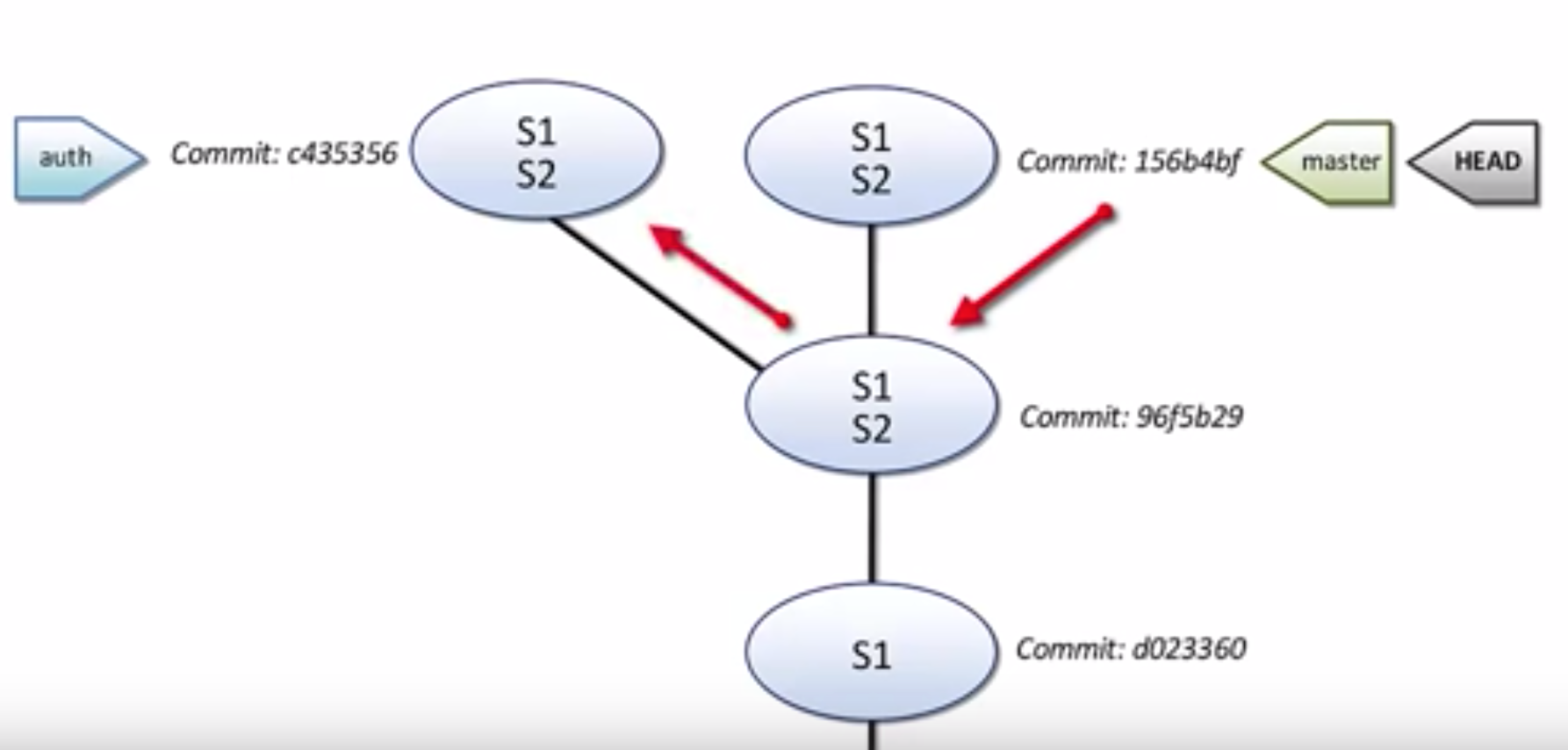
1. git status (reminds we are on qa branch)
2. git checkout master (moves us to master branch)
3. git diff master..qa (shows what will chang when we merge qa into master)
4. git merge qa (from the master branch merge qa into master)
5. cat file1 (shows that line4 is added into master branch, it come from qa branch)
6. graph (it shows that master branch is caught up with qa branch now, git move the pointer to the same commit where qa is)

now we as we merged qa into master we don’t need qa branch

Delete branches

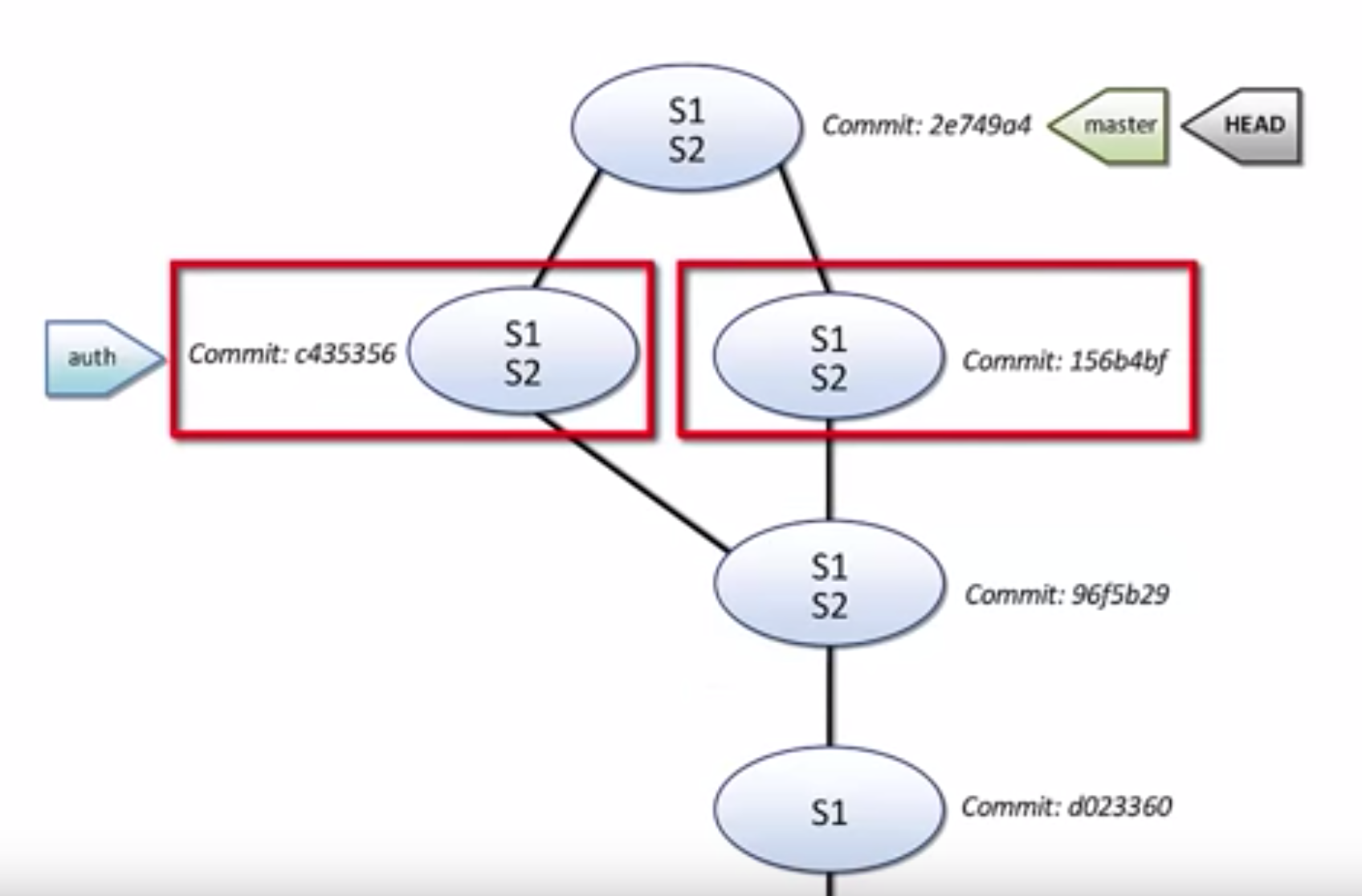
1. git branch –merged (shows that our master branch has been merged with qa branch, so now we can safly delete the qa branch, note dev branch is not listed here as we haven’t merge dev branch into master branch yet)
2. git branch -d qa (deleted branch qa, ‘-D’ can forcefully delete the branch)
3. git branch -d dev (error: - The branch ‘dev’ is not fully merged.)

Not a direct path from master branch to dev branch, so git cannot do a fast-forward merge this time, for this case a 3-Way merge happens



3-Way merge

To merge master with dev we cannot move the pointer just from master to dev, if we do so we’ll loose our qa branch changes, we need to merge these branches together into a new commit called the merge commit, to make this merge commit



To make this merge commit git looks at 3 commits first base commit frome where the two branches started from, then the last commit of each branch

1. git status (we are on master)
2. git merge dev (starts the merge, since we are making a merge commit, we are going to need a commit message, we can accept git’s default message here, save and exit, now it doesn’t say fast-forward, now it says Merge made by the ‘recursive’ strategy.)
3. graph (we can see merge commit joining the two branches)
4. git branch –merged (shows that our master branch has been merged with dev branch, so now we can safly delete the dev branch)
5. git branch -d dev (deleted branch dev, ‘-D’ can forcefully delete the branch)

Merge conflicts

Conflicts occurs when we change same lines in the same file.

1. git checkout -b dev (one line shortcut to create and checkout new branch)
2. vi file1 (create conflict, change line1 and line3 to lineI and lineIII)

lineI

lineIII

Esc 🡪 :wq

1. git diff (show our new modifications)
2. git commit -a -m “updated file1 line1 and 3” (shortcut ‘-a’ is for stage and commit)
3. git checkout master
4. vi file1 (create conflict, change line0, line1 and line3 and delete line4 to row0, row1 and row3)

row0

row1

rowIII

<delete line4>

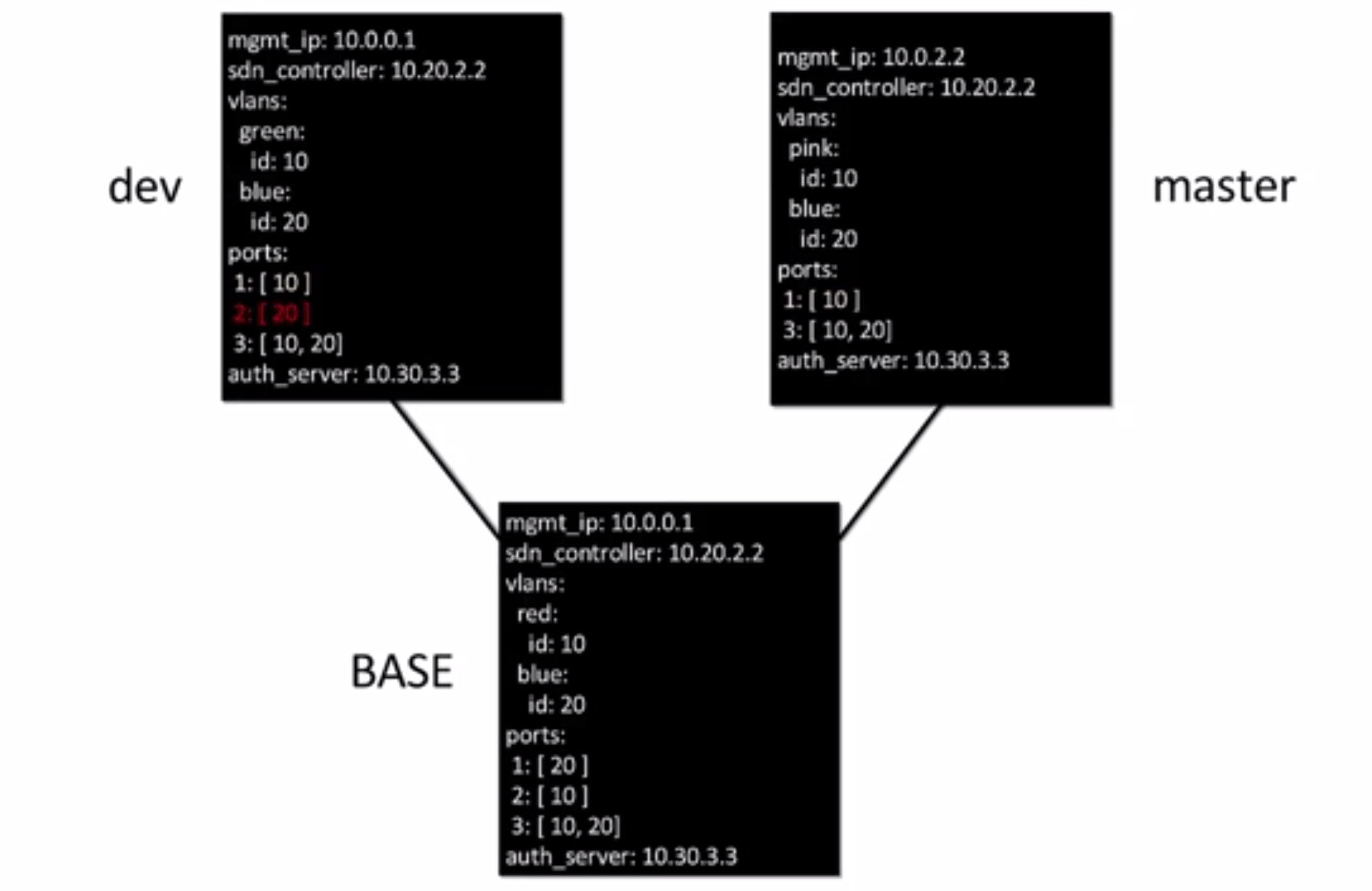
Esc 🡪 :wq

1. git commit -a -m “updated file1 in master” (shortcut ‘-a’ is for stage and commit)
2. graph (shows our two branches are diverged, so we know that it can’t be fast-forward, this is going to be a three way merge,and there will be conflicts)

line0 zero change will win from master

line1 changed in both master and dev so git can’t guess which version we want to keep

line3 has exact same change on moth dev and master so no conflict



1. git status (shows we are on master)
2. git merge dev (Merge conflict in file1)
3. git status (you have unmerged paths, if we don’t want to deal with the conflicts here, we can abort by using “git merge --abort” to abort the merge)

assuming we started with clean working tree and staging area, we can do this

1. git merge --abort
2. git status (shows we are out of the merge process)
3. graph (shows us we are right back we were before)
4. git merge dev (Merge conflict in file1)
5. git status (you have unmerged paths, if we don’t want to deal with the conflicts here, we can abort by using “git merge --abort” to abort the merge)
6. vi file1 (to see conflicts)

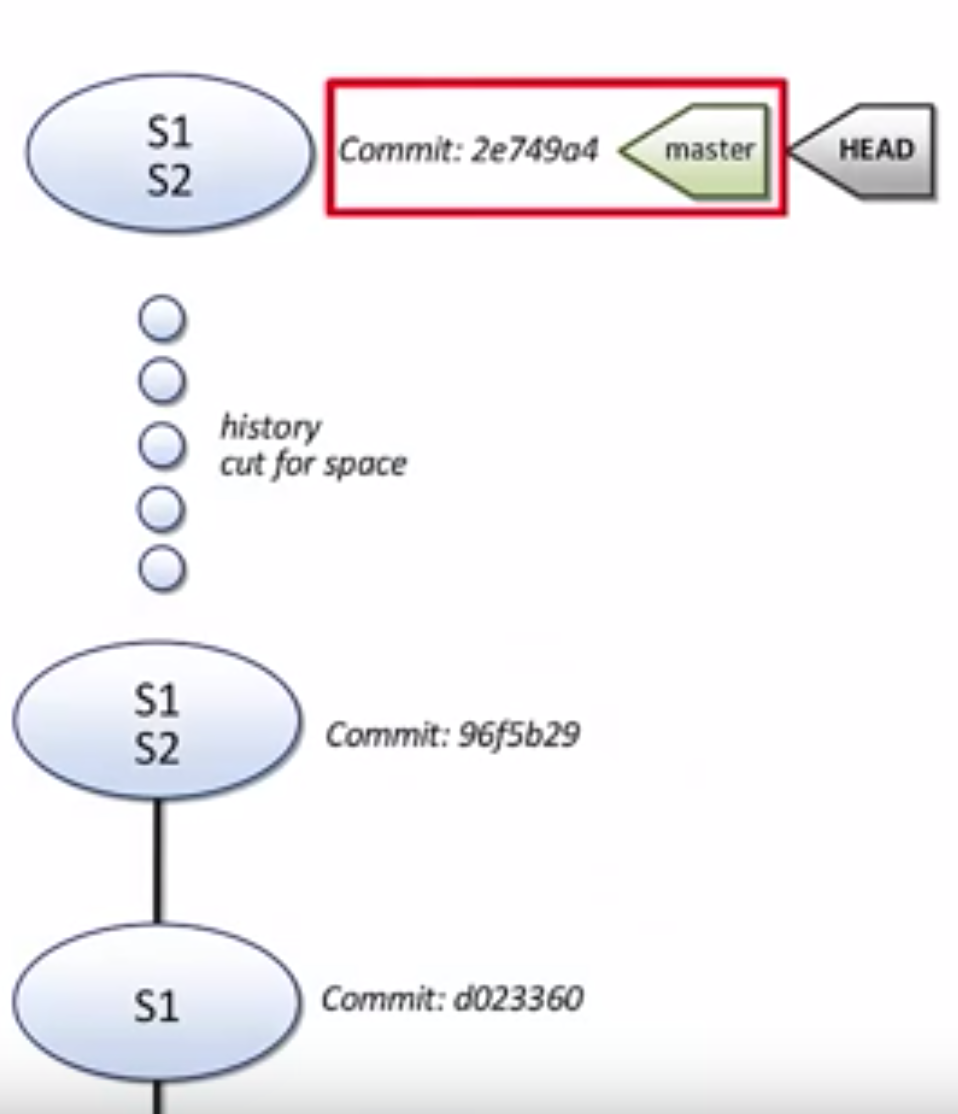
(set of equal signs ====== separates out the state of the file in the two branches, on HEAD which currently pointing to master branch line1 changed to row1 below the equal signes shows that on that commit dev branch points to line1 changed to lineI, we need to decide how our merged file should look, once decided we need to delete git markers(<<< HEAD and equal signs) and the not needed code.)

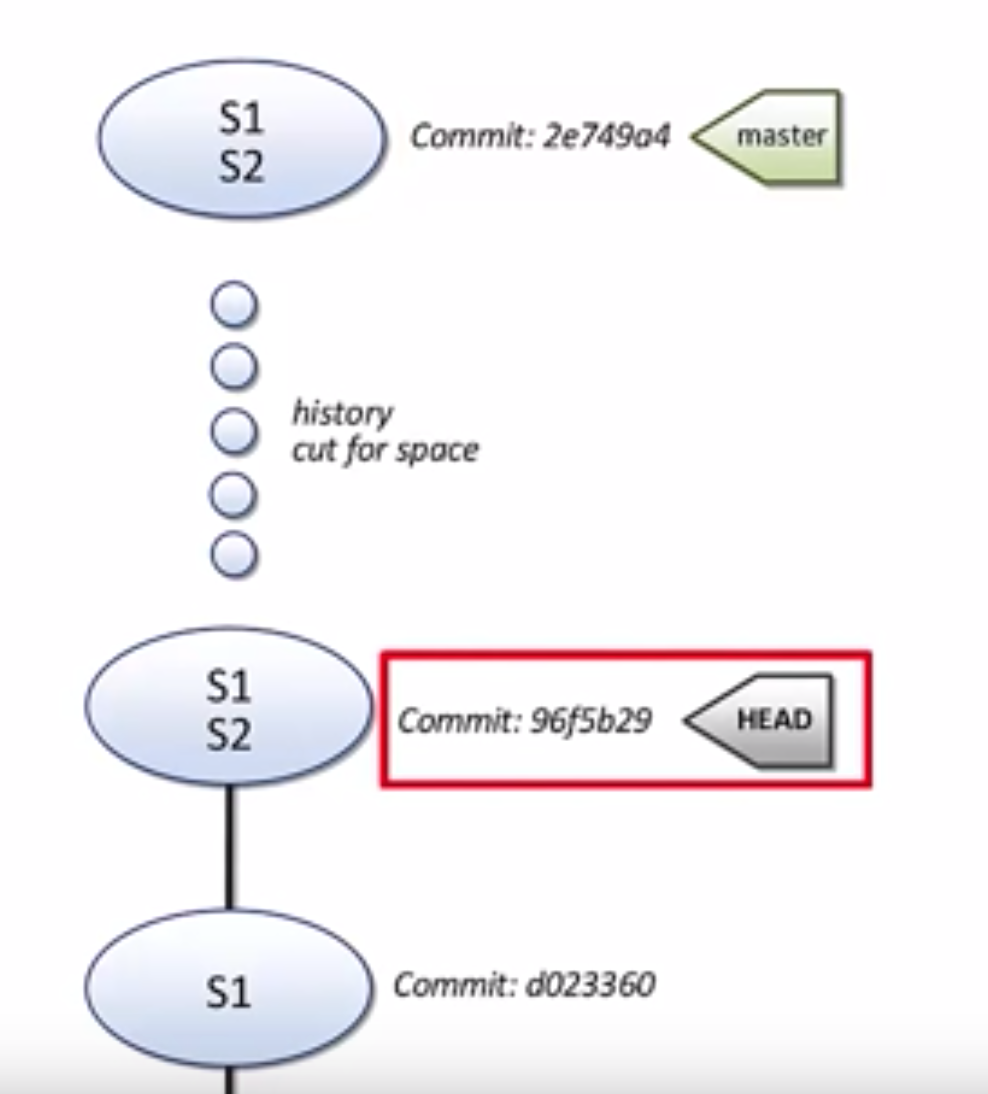
Esc 🡪 :wq (save and exit from vi)

1. git add file1 ()
2. git status (all conflicts fixed but you are still merging)
3. git commit (default merge commit message is acceptable, so save and exit)
4. graph (shows our merge commit)
5. git branch -d dev (we can now safely delete our dev branch)

Detached HEAD

Usually HEAD points to our branch which inturn points to a commit, but when HEAD is instead pointing directly to a commit, we have the Detached HEAD state



Lets’ checkout a commit instead of branch

1. git log (shows our commit history)
2. git checkout <initial 5 characters of commit hash (by this we are checking out directly a commit, now git gives us some warnings and graph shows us the situation, take it from commit message created file2)
3. graph (shows us the current situation, we have he head pointer directly referencing to a commit and not a branch, this is a detached HEAD)

one way to get out of this state is just checkout a branch again.

1. git checkout master
2. graph (now we don’t have a detached HEAD state)

lets see another way to handle detached HEAD state

1. git checkout <initial 5 characters of commit hash (by this we are checking out directly a commit, now git gives us some warnings and graph shows us the situation, take it from commit message created file2)
2. graph (shows us the current situation, we have he head pointer directly referencing to a commit and not a branch, this is a detached HEAD)

another way to get out of this detached state is we can put a new branch label here

1. git branch newbranch (creates newbranch here)
2. graph (shows we have our new branch, but HEAD isn’t attached to it.)
3. git checkout newbranch (checksout our new branch)
4. graph (shows HEAD pointer is no longer detached, it is attached to our new newbranch, from here we can proceed normally)

Git stash

1. git checkout master
2. vi file1 (edit file1)

change line2 to row2

Esc 🡪 :wq

1. git status
2. git checkout newbranch (git blocks us doing this, if itndon’t then we’ll loos our changes on master, to proceed either commit changes on master or stash)
3. git stash (saved our new changes to file1 so we can apply them back later)
4. git status (shows that we are back to a clean state, now we can freely checkout any branches and perform any merge)

another stash

1. vi file1

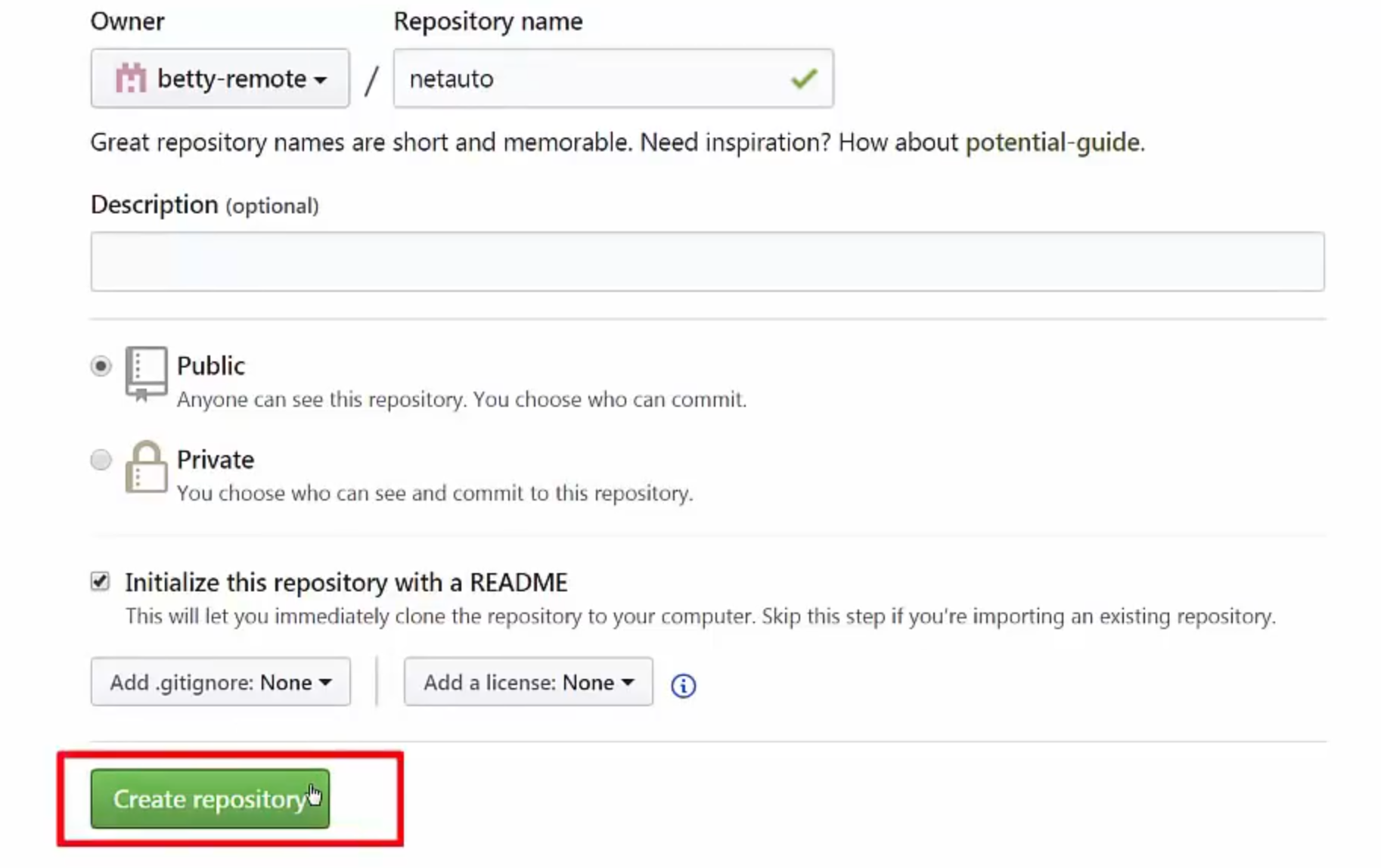
remove last line

wa

1. git stash (to save these changes)
2. git stash list (we can see our two stashes)
3. git stash list -p (with ‘-p’ option we can see the edits which occurred at each stash points, we can reapply these stashes at any time)
4. git stash apply (reapply the most recent stash, ‘apply’ don’t remove/pop the stash from list so we can reuse them, to remove/pop from stash list use “git stash pop”)
5. git diff ( we can see the removal of last line)
6. git commit -a -m “removed line4”
7. git stash list (we still have two stash points, we didn’t pop/removed the most recent stash, so we can reuse stashes as needed, if we really want to apply stash and remove it from stash list then use pop instead of pop, to use other stash other than most recent stash, we can called out the labels)
8. git stash apply stash@{1} (apply the stash with label stash@{1})
9. git diff (we can see the change)
10. git stash list
11. git stash save “removed line4” (in the output of git stash list we may want a better reminder for what each stash contains, we can provide a message with the stash with we can provide a message with stash )
12. git stash list (now list show the message with stash)

**Remotes**

1. Create a GitHub Repository
2. Login to GitHub
3. Click Start a project
4. Give a name “deeDemo”
5. Select Public/Private
6. Check Initialize repository with Readme.md (Git will create README.md file, it’s a markdown type file)
7. Click Create Repository button



Repository will be create with single commit (Initial commit)

Click on Clone or download button and copy the ssl/https address

Git Clone

Laptop1

1. git clone <remote address copied above>
2. cd deeDemo
3. Config user name and email –

git config --local user.name “Deepak Arora”

git config --local user.email [deepak.arora@milwaukeetoolc.om](mailto:deepak.arora@milwaukeetoolc.om)

(

1. Global: -

git config --global user.name “Deepak Arora”

git config --global user.email [deepak.arora@milwaukeetoolc.om](mailto:deepak.arora@milwaukeetoolc.om)

or

1. Local: -

git config --local user.name “Deepak Arora”

git config --local user.email [deepak.arora@milwaukeetoolc.om](mailto:deepak.arora@milwaukeetoolc.om)

)

1. git remote (display our git remots, we have a single remote and its name is origin after clone it the default name for our first remote)
2. git remote -v (display full locations for fetch and push addresses)
3. git log --all --decorate --online --graph (our deeDemo repo has single commit “Initial Commit” created by GitHub in his commit history, locally we have only master branch, also we have our symbolic HEAD pointer, which tells us that we have master branch checked out, in addition to local pointer/references we have something new- we have origin/master. (origin/master is a specialized branch, it is called a remote tracking branch, the job of remote tracking branch is to tell us wht the master branch looks like at origin, it is telling us that origin/master at GitHub and master branch at local are pointing the same commit in other words our local branch master and remote branch origin/master is in sync.

shows we are on the second commit due to the HEAD pointer, The HEAD pointer is pointing to the master branch, this tells us that we have master branch checked out. This is a useful command so we can create a command alias to shorten it.)

1. alias graph=”git log --all --decorate --online --graph”
2. graph (same output as 12)
3. checkout ‘origin/master’ (remote tracking branch is not same as local branch, checking out remote tracking branch works but we’ll end up detached HEAD state, this is fine if we want to find how things are stand at our remote, however we do want to stay on our local branches.
4. git checkout master

**git fetch and git merge**

1. GitHub (User1)
2. Click “Create new file”)
3. Give a name ‘file1’
4. Paste some data in it(

Line1

Line2

Line3

)

1. Commit new file (we can leave default commit message, this creates our second commit on remote, we don’t have it on our local machine yet.)
2. git status (git does not know about the second commit/changes at origin/master, it says we are up to date with origin/master, however we know that this is not true.
3. Graph (In the commit history we see the same thing)
4. git fetch origin (to be updated with origin we use this command, our system reached out to GitHub it finds the new commit and brings it down)
5. git status (here now it says our local branch is behind by origin/master by 1 commit)
6. graph (we see more details, two commits at remote we hve two commit while locally we have one commit.
7. git merge origin/master (we are merging commits referenced by origin/master into our local master branch
8. ls (it shows file1 is in our local repository)
9. graph (shows we are in sync again, our local branch and remote branch are at same place)
10. git pull (it is same as git fetch + git merge, it is not preferred to use ‘git pull’ to refrain ourselves to get into unexpected merge results, it is good to use separate git fetch and git merge)

git push

1. vi file1

change Line1 to Row1

Esc 🡪 :wq

1. git commit -a -m “new Row1 for file1” (shortcut ‘-a’ is for stage and commit)
2. graph (we can see our new commit, we see we are locally one commit ahead by orgin/master)
3. git status (it says the same as above)
4. git push origin master (we are pushing our edit/changes to our remote named origin into the master branch at origin, this push relies on github authontitain
5. graph ( now we see our tracking branch is updated)
6. Look at GitHub (it is updated now we have three commits)

Create a github fork

1. Login to GitHub (user2)
2. Find/look User1’s repo (deeDemo)
3. Click on ‘Fork’ button (it creates the fork and take us directly on the fork under user2’s account, we see the note that we currently on fork)
4. Get the clone address
5. git clone <clone address from above> (at user2’s local system, will clone the remote fork/repo to the user2’s local system)
6. git cd deeDemo
7. git remote -v (shows us the git remotes, here origin shows user2’s fork deeDemo, display full locations for fetch and push addresses)

Add a git remote

1. git remote add upstream (We are giving an alia or short name ‘upstream’, the location of this remote is user1’s deeDemo repo)
2. git remote -v (shows two remotes now origin pointing to user2’s fork and upstream pointing to user1’s repo)
3. If we ever want to remove a repo e.g. “git remote remove upstream” (git remote remove <remote name>

Sync with upstream

1. At user2’s local system
2. Graph (3 commits, local and origin are at the same commit, notice that we don’t have a remote tracking branch for our new remote named upstream, we have to run git fetch for this)
3. git fetch upstream
4. graph (now graph show a new remote tracking branch upstream/master, it reflects the state of user1’s master branch at github, origin/master reflects the state of user2’s fork at git hub)
5. At user1’s local system
6. cp file1 file2 (copy file)
7. vi file2

change Row1 to Row0

Esc 🡪 :wq

1. git status (shows the new file)
2. git add file2
3. git commit -m “added file2”
4. graph (shows user1’s local master branch has moved up, it out of sync with its origin, we can push)
5. git push origin master
6. graph ( now graph shows that user1 is in sync with it’s origin)
7. At user2’s local system
8. graph (graph show that user2 doesn’t see the new commit yet, he won’t see until he do git fetch upstream)
9. git fetch upstream
10. graph (this time graph shows new commit the tracking branch upstream/master has moved up, both user2’s local repo and fork are out of sync of user1’ repo)
11. git merge upstream/master (merges commits from upstream in the master branch into our local master branch
12. graph (show now we are finally caught up with locally, the local master is at the new commit where upstream/master is, the only thing lagging behind is user2’s fork presented by origin/master)
13. git push origin/master (
14. graph (now graph shows that we have all three repos in sync)

Remote Branches

(User2 will create a new local branch, push it upto his fork and create a pull request to the User1’s original deeDemo reop. The request is for user1 to merge our new edit-file2 branch into the master branch of his repo, we are effectly asking user1 to merge our new changes into the main project, if user1 do merge then we can clean up work, we can delete our local branch locally and from our fork in GitHub, we can also resync our master branch locally and in GitHub)

1. User2’s system
2. git checkout -b “edit-file2” (create and checkout new branch with one command)
3. git branch (shows our new branch, notice it doesn’t display any remote tracking branches)
4. git branch -a (to see local and remote tracking branches use ‘-a’)
5. git branch -r (to see remote tracking branches only use ‘-r’)
6. vi file2 (change in file2 and save it.)

change last Line3 to Row3

Esc 🡪 :wq

1. git commit -a -m “updated file2” (shortcut ‘-a’ is for stage and commit)
2. graph (we see our new commit and our new branch pointing to that commit
3. git push origin edit-file2 (we are pushing our new branch edit-file2 up to origin or user2’s fork)
4. graph (it now shows that we have our remote tracking branch origin/edit-file2, it is at the same commit as our local edit-file2 branch that we have checked out)
5. Got to GitHub and see at User2’s GitHub account in the deeDemo project we are in the master branch, we can see our new branch by clicking on branch dropdown ans select new branch “edit-file2” git shows that this is one commit ahead of user1’s master branch, from the “edit-file2” branch we can make a New pull request” by using “New pull request” button, we are looking to merge our edit-file2 branch into user1’s master branch, update commit message, press the “Create pull request” button, GitHub then send a notification to user1 about the pull request, user1 take different acteions including reviewing the change/adding comments/requesting alterations/asking others to review/approving and merging the request/closing the request to deny, when the pr (pull request) is done, we can clean up work, we can delete our local branch locally and from our fork in GitHub, we can also resync our master branch locally and in GitHub