(2):An Important Note About GitHub Naming Conventions

As of late 2020, GitHub has renamed its default branches from **master** to **main.** New repositories will always use **main** as the name of their default branch, older repositories may still use **master** for the default branch. Which naming convention you will use depends on when you created your repository, when the repositories you are working on were created, and your version of git. As these videos were recorded before the naming convention was updated, Colt uses commands like **git checkout master** in the videos in this unit. When following along, you may need to use commands like **git checkout main**. For more information about this updated naming convention, read the GitHub documentation linked here.

Basically whenever he used master ,use main instead.

(3):Branching

everything so far in git, has been on a single branch.

Main is the default branch created whenever you initiate a git(a repository) in a file.

can make other branches whenever

can make a new branch for example when you want to make some serious edits that are risky and you are not sure will work.

- if it does not work, you can delete that branch. If it does work, then you can add that back into the **main branch**(can merge it in).
- On most developer teams, most devs do not work in the master branch, but off branches.
- if you want to see which branches exist in your repository, use git branch

how to make a branch:

- to make a new branch do: **git branch name**, when you type git branch, it will show the new branch.
- cannot switch to other branch while work is not committed

- to switch you have to do, **git checkout name**
- any changes made on one branch will not affect the other branches or the main branch
- any commits made on one branch will not be shown on the other branches except the master branch(commits made on the master branch will show on the other branches but not the other way around).
- you can do git log –oneline to show it on one line.
- when you go back to the main branch after having made a ton of changes and new files on the off branches, the main branch will have none of that.

(4):More Branching

git branch nameOfBranch will make a new branch

git checkout -b NameOfBranch: this will make a new branch, but will also switch to it immediately.

the current location that you make the new branch on is important. Whatever branch you are currently on will be used as the basis for the new branch(all of the logs of that branch and the branch that that branch was branched off of will transfer as well as the files).

(5,6):Deleting Branches, Merging

there are a few reasons to delete a branch:

- the branch has been merged with main branch
- or you don't want those changes, so you delete it.

you can delete the branch by doing: **git branch -d name_of_branch**

- cannot delete the branch that you are currently on, have to do it from another branch.
- if you delete without merging you will get a warning and it will say to use **git branch -D name_of_branch,** but if you merge it, it will delete using -d.

it is always a good idea to check git status before making a branch or merging ect, to make sure that the working tree is clean.

git merge branch_name: this will merge one branch with another.

- you have to use git merge in the branch that you want the other branch to merge into. So if you git merge in the main branch, the branch that you git merge will merge into the main branch(because you are in that branch).
- if you: git merge try-purple in the main branch, the try-purple branch will merge into the main branch. This will add the commits in the branch to the main branch and the files.
- merging is not so simple, sometimes might run into **merge conflicts**.

(7):Merge Conflicts

Depending on the history of your commits, when you merge, it will either be a(you do not specify these, it is the strat that git will use behind the scene):

1. Fast forward merge
    - git will use this when it can tell what order the commits happened in.
    - git will use this when it is simple
2. Recursive merge
    - happens when the commits are in a complex order.

merge conflicts occur when there are conflicting changes on branches and you are trying to merge them together.

when there is a merge conflict, you have to manually resolve it.

- simple ex of a merge conflict: when you make a file on the off branch and later make a file on the main branch that has the same name and you try to merge. Since you have 2 different files with the same name, there will be a conflict.
- if you look at the file when merged, git will place in the file the conflicting text so that you can decide which one to keep(it will clearly show the 2 different contents in the file). you have to decide how to fix the conflict, then commit it.
- make sure that the commits are named well.