# What is Version Control:

For Content, teams and Agility.

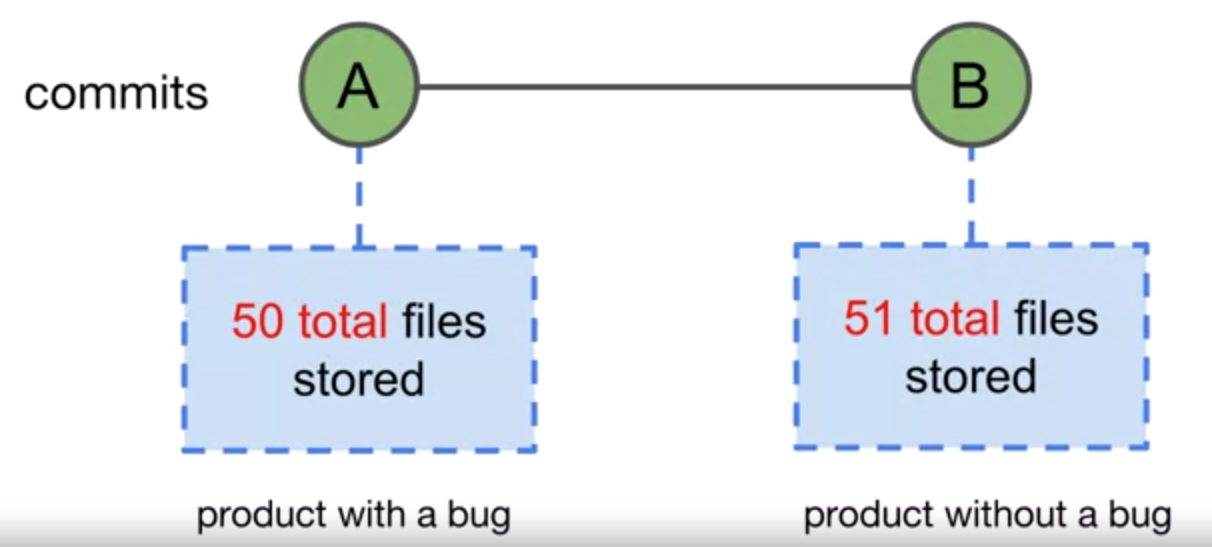


# Git:

Maintains full history of project.

Each commit is a snapshot of entire project.

But done is an efficient way: Each unique file is stored only once



Best Practice: Create a separate branch for every feature/change and then commit to the main branch.

# Basic Git Syntax



## Git Command Example:

**git status**

## Git Command-- flags Example:

**git status –short**

## Git command --flags [arguments]

**git add file.txt**

**git add .**

# Git Locations

## Working Tree

A single commit’s directory and files

## Staging Area or Index

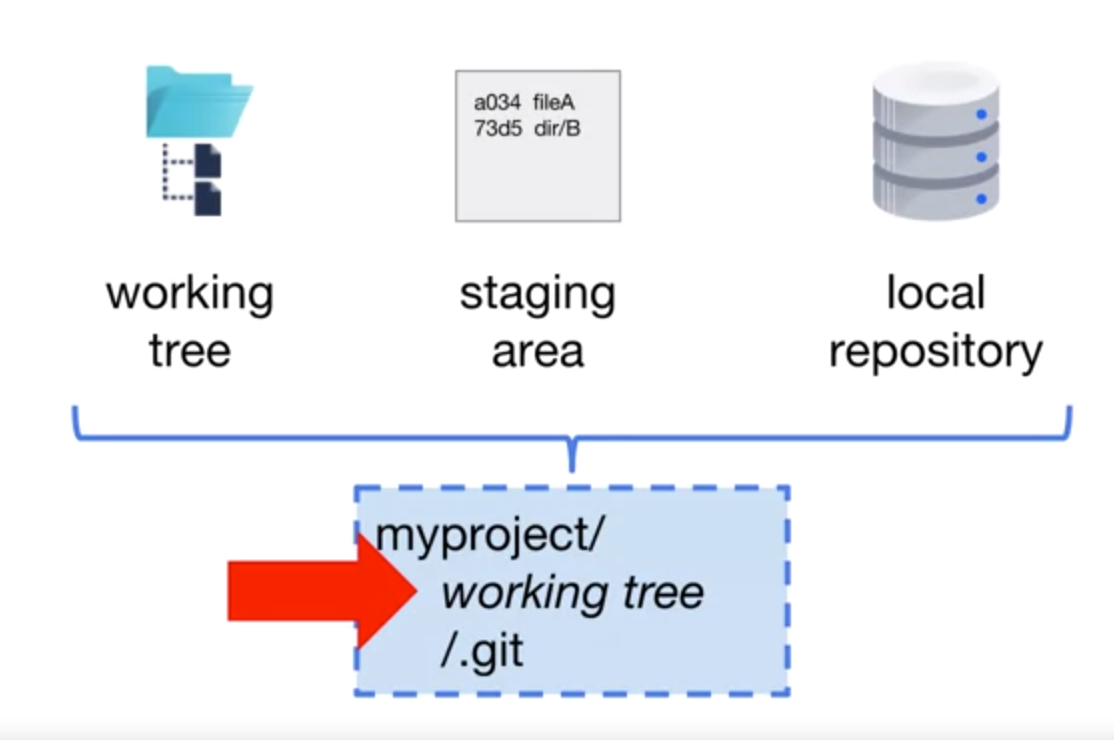
Files that are planned/ready for the next commit. **Add** command is used to achieve this.

## Local Repository

All the commits that have been made for the project. These commits contain the version history of the project. Also **contains staging area.**

## Project Directory

Contains all the project that is being handled. It contains **working tree** (files of single commit of the project – usually latest)and **.git** folder that contain staging area and all the commits of projects i.e. **Local Rep.**



## Remote Repository

All the commits just like Local Rep but kept online/cloud. It is single **source of Truth**.

# Commit to a local repository

## Git Status

view the status of files in the working tree and specifically **staging area**. If you **add** anything it will appear in status

Enter following command to show staus (-s for short details)

**git status -s**

When a file is in working dir but not added to stage



**git add <filename>**

ORs

**git add .**

When a file is added to stage



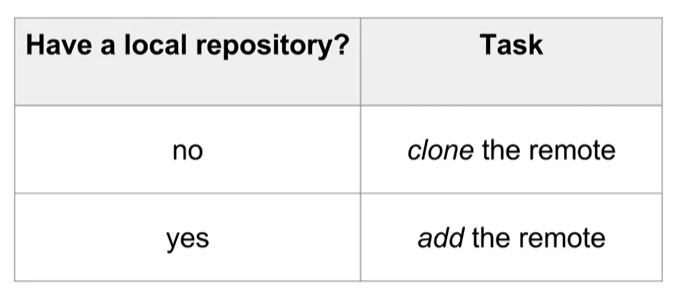
**echo “Someting” > FileA.txt**

When something is modified in staged file



Add again to get latest changes into stage

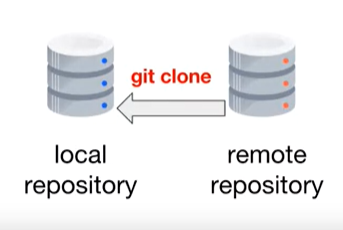
# Push to Report Repo



## Git Clone

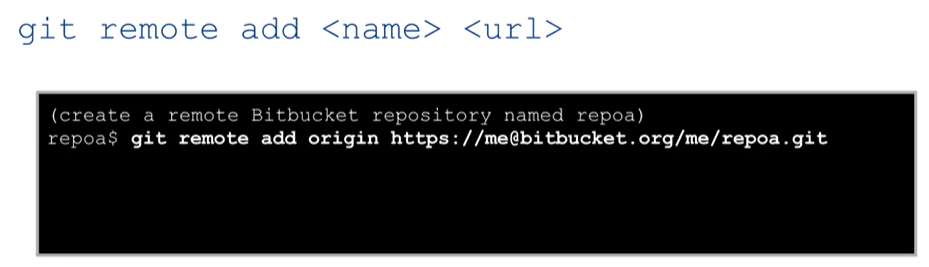


If project name specified (as in option in command) it will be used as project directory. If not specified, then name in the URL (minus the .git) will be used.



## Git Remote Add

When you have local repo already set and committed



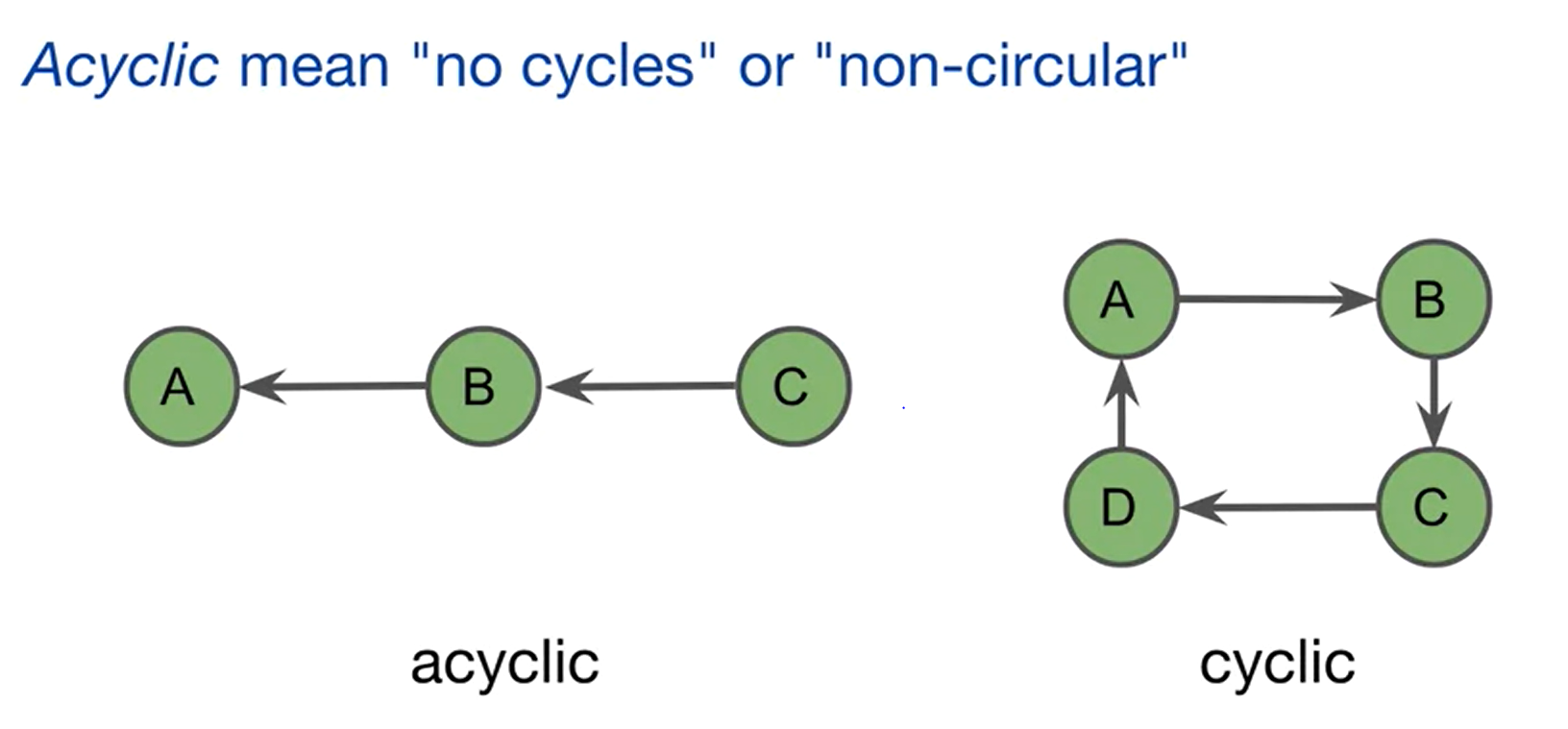
# Git Graph

## Directed Graph:

A graph that has nodes pointing to a specific direction

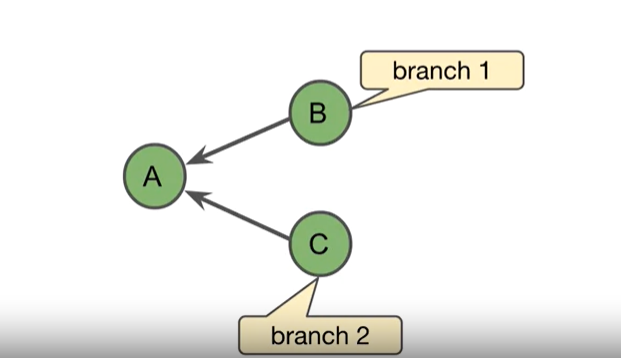
## Acyclic

Cannot end where it starts



## Branch

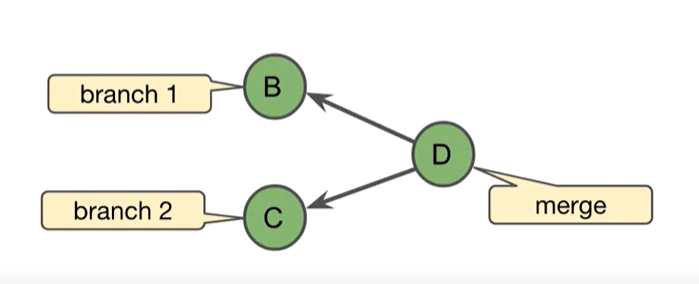
Branch occurs when a Commit has 2 (or more) child



In this example B and C have branched off A

## Merge

A merge occurs when a Commit has 2 or more parents.



# Git Objects

Internally Git uses objects to store 4 types of things. Git keeps these objects in something called Object Store (or similar).

## Commit Object:

A text file containing:

* commit user info
* commit message
* a ref to commit parent/parents
* ref to the root tree of project

These all elements git need to build full **Commit**

## Annotated Tag:

Ref to a specific commit.

## Tree

An object that contains a list of file names and directories inside a directory. Mostly internal object i.e. developer usually do not interact with it and Git usually manages it.

## Blob

An object that stores the content of **Files that are being managed by Git.** Mostly internal object i.e. developer usually do not interact with it and Git usually manages it.

# Git ID

Git ID is the name of a specific Git object. Also known as **SHA-1, Checksum Object ID** and **Hash**



Above is a git id of Commit Object (explained above)

Git uses SHA-1 to generate IDs based on content of files. Same git id (SHA-1 Value) for same content every time.

## SHA-1

Even it is mostly done at the backend by git itself, you can go ahead and create/see git id for any file or content. Below command:

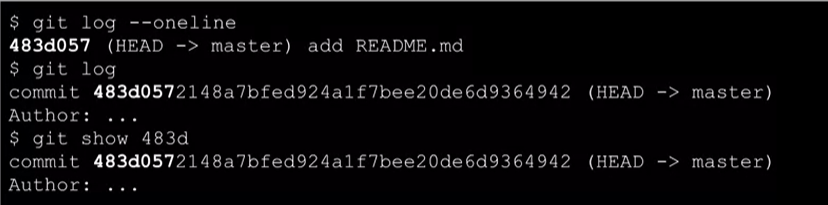
**git hash-object <file>**



Hash-object command is one of git’s low level commands, called **Plumbing Commands**. It helps when you are scripting or to get an advanced level understanding of low level git workings.



## Short IDs

We can shorten the large 40 character git SHA ids. 

The **--oneline** option shows first 7 characters of a commit object’s id.

We can also search an object using few first characters (minimum 4). The first four or more characters of the Git ID are required when executing Git commands. More characters may be required if two objects have similar names

# Git References

A reference is a label given to any SHA-1 value / Git object Id

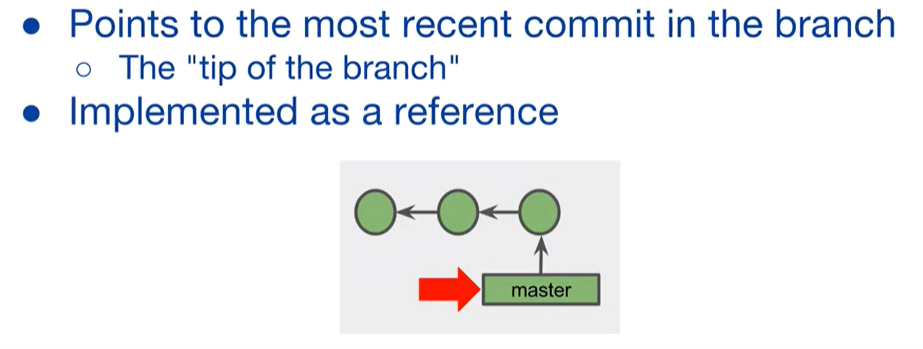
Git references help easily identify objects and retrieve information about hat object. For example, in above **git show <object id>** command, we can see that it has 2 References (HEAD & master).

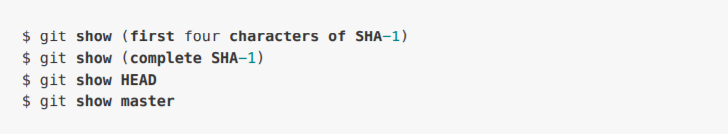
We can just do **git show HEAD** to see all the information about that object just as we can see with ID. Here HEAD is a reference to the branch label master.

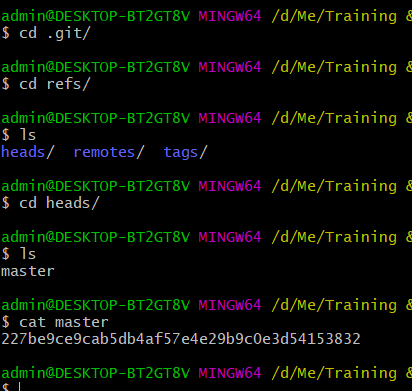
## Branch label

Git Branches use this technique to efficiently identify objects.

Called Branch Labels, it is only labeled with the latest commit of a branch, as shown below



***Remember that commits timeline is in reverse order of arrows, it means that commit points to its parent. So last commit is latest.*** 



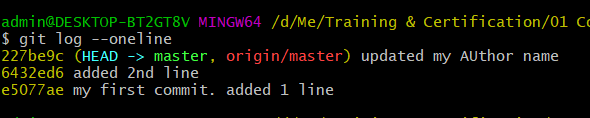
Above is an example of reference file, how git maintains it.

This is how git makes association between **SHA-1** hash and ’tip of the branch’

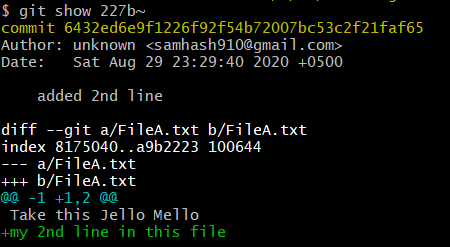
# Git references using TILDE character (~)

Tilde (read Tilda) points to the previous commit of any commit.

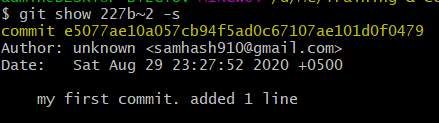
Below is whole history of commits, there are total 3 commits done.



Getting info of commit previous than the 227be

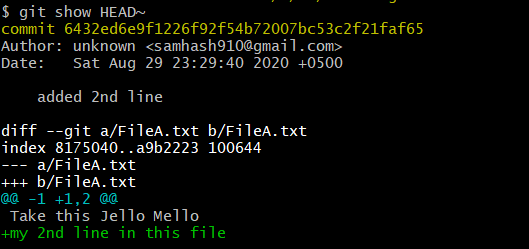


Getting info of 2 commits back from latest commit. (*-s shows short information)*



We can do above things either from Object ID, as done above, or by Branch label (master) or Reference (symbolic reference)

As below output is same as above



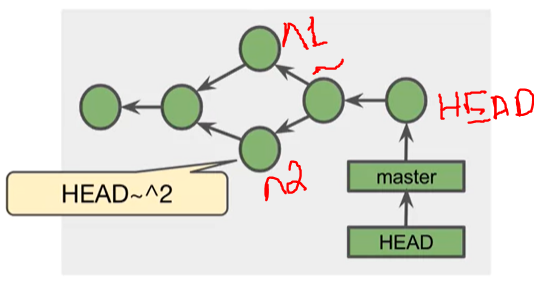
# Git references using Carrot character (^)

Carrot works same as Tilde (~), it just points to the parent in a merge commit. Usually in merge commit, a commit has multiple parents (different branches merging into 1)

Same as **master~** will point to the previous commit than the latest, **master^** will go to parent of 1st commit, which are same in this case.

**master^^** goes to 1st parent of 1st parent which is same commit as **master~~**

BUT, **master^2** will not go to parent of parent, instead, it goes to 2nd parent of first commit. In above example, we do not have multiple parents. So using graph example below:



**HEAD~^2** will break down like this:

* HEAD is a symbolic ref to master so it actually starts from whatever master is pointing. As we know *master* has only the reference of OBJECT ID of latest commit.
* ~ will go to 1 commit back from latest commit
* ^2 will look for 2nd parent of that commit.

## Git Tags

### Light Weight tags:

**git tag <tag name>**

### Annotations:



**git tag -a v1.0**

If -m is not specified, it will open the default editor (set at the time of installation) to enter the comment. An Annotation tag and can be signed and verified with **GNU Privacy Guard**

# Branches

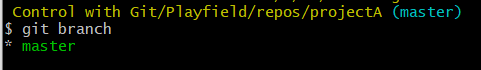
Flexible way to extend a project. usually there is 1 master branch and others are small features branches. Branch is just a reference in a file, so it is an edition of just 1 file to the project, making it super-efficient.

* Easiest way to maintain multiple versions of the project.
* Enable experimentation
* Enable Team development

There could be branches that can run till the end of project, like **master, developer, release**

Feature, bugfix, hotfix branches are short lived and called **Topics**

We use **git branch** to show all the branches

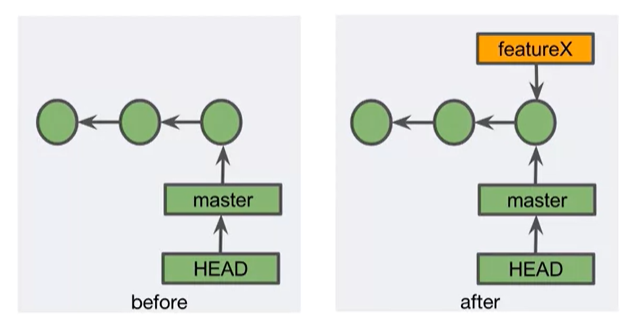


The asterisk (\*) shows which branch one is current branch

## Create a branch

Creating a branch has same command as showing branch, just add branch name to it and it will create that branch **on most recent commit**

**git branch featureX**



Notice that the HEAD is still referencing to the master branch label. Read ahead to learn how to change that.

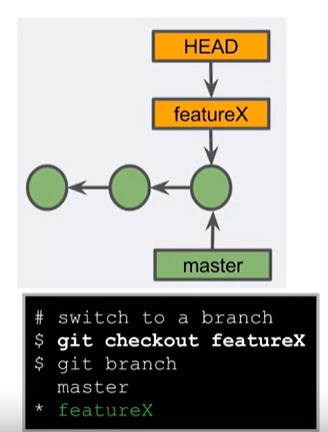
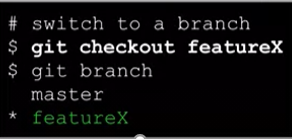
## Checkout

To switch HEAD from master (or any other branch or commit) to the branch/commit of your desire.

When we checkout, it updates the **working tree** to the files of the commit/branch you checked out to. So when you do a **ls**, it will show files for the newly checked-out branch or commit.

**git checkout <branch name / commit SHA-1>**

Compare below image than above one.



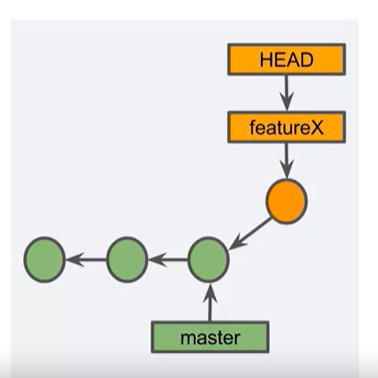
### Create and Checkout in same command

IT IS ONLY FOR NEW BRANCHES - FAILS FOR EXISTING BRANCHES

**git checkout -b feature**

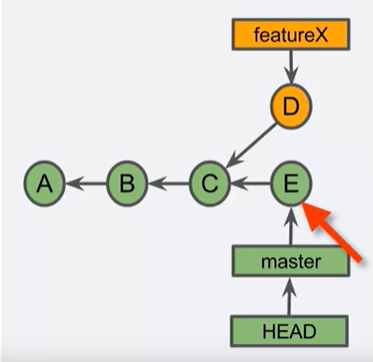
### The Next commit after Checkout

The next commit will add a commit to the checked-out branch but the master or other branches will remain pointing to the old branch. HEAD will also move with the checked-out branch.



### The Next commit on Master

If we check back to master and do a commit there, the master will follow its own path.

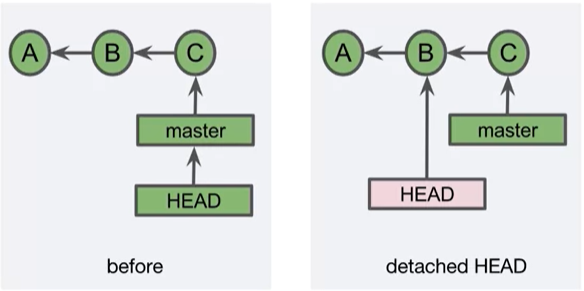


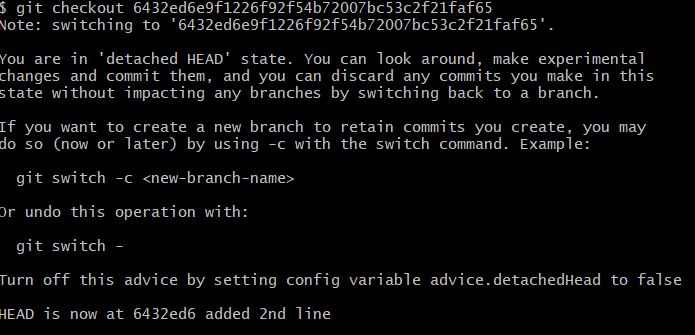
Neither master nor featureX knows about other branches, so they won’t interfere others.

## Detached Head (Pointing to a commit)

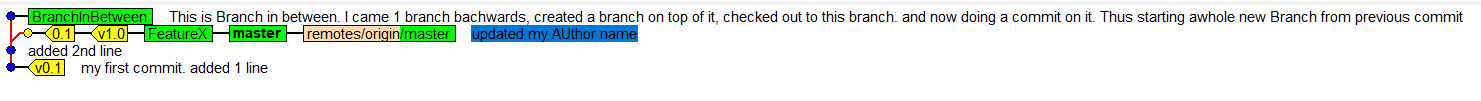
Usually you check out a branch, but what if we checkout a commit? One might want to go to the older version of the project and see that, or start a separate feature/branch on top of it.

Detached Head happens when we checkout to a commit instead of a branch, using **git checkout SHA-1**





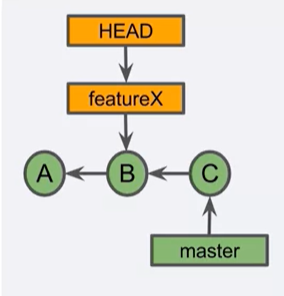
But this is not normal, so to make it normal, one should create a branch on top of it, and checkout to that branch.



In the above diagram, I checkout to a previous commit from master (featureX points to master too). Then made a new branch there named **BranchInBetween**, did a commit to file there. The commit message shows detail.

Now If I checkout back to master or feature branch, my working tree will show other version of branch, the one that does not contain BranchInBetween’s commit.

Above tree like visual structure is shown by GIT GUI using **gitk --all**



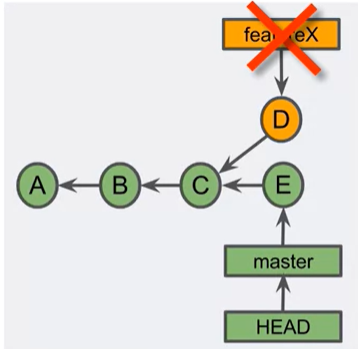
## Deleting Branches & Dangling Commits

**git branch -d <branchname>**

When you a branch that is already merged, git just deletes the label.

**But**, if one tries to delete a branch with a commit that is not merged or pointed by any other branch. Git gives an error.

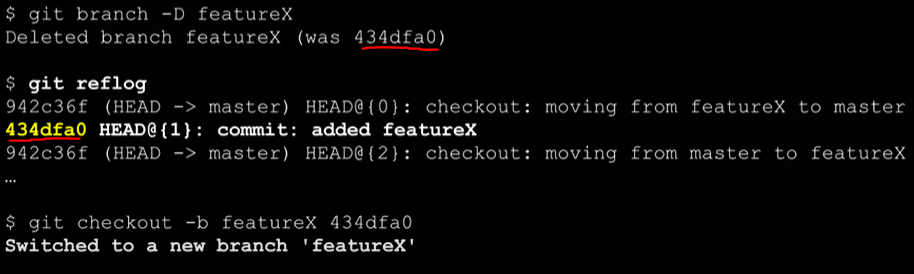
To forcefully delete that branch, use **git branch -D <branch-name>**



Notice that this will create a **Dangling Commit** in shape of D, and git will periodically look for garbage collection, hence deleting this commit resulting in potential loss of code.

## UNDO Branch Delete

We can do that with the help of **git reflog**, that shows all the recent HEAD checkouts



In the above diagram, a branch was forcefully deleted using -D. The delete info shows SHA-1 of the commit the branch was pointing to.

git reflog shows there that SHA-1 was pointing to a branch called featureX and HEAD was pointing to it.

We then create a new branch on top of the dangling commit using its SHA-1

This creates a fresh branch label on top of the dangling commit. Remember that branch label is nothing but just a label, so it doesn’t matter if its new or old. The actual change is in the commit.