# What is Version Control:

For Content, teams and Agility.

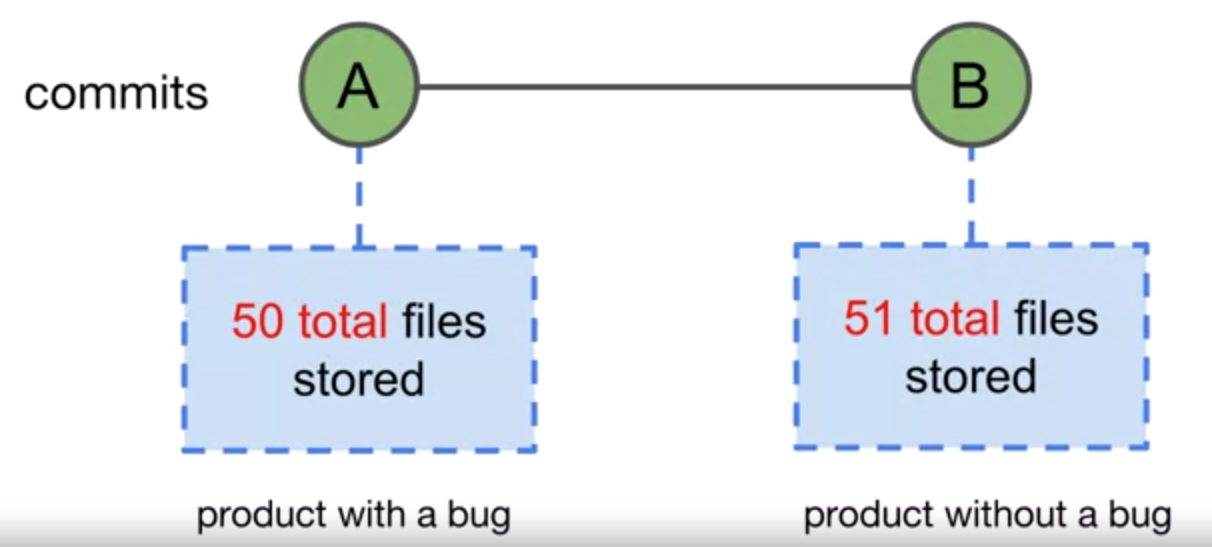


# Git:

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

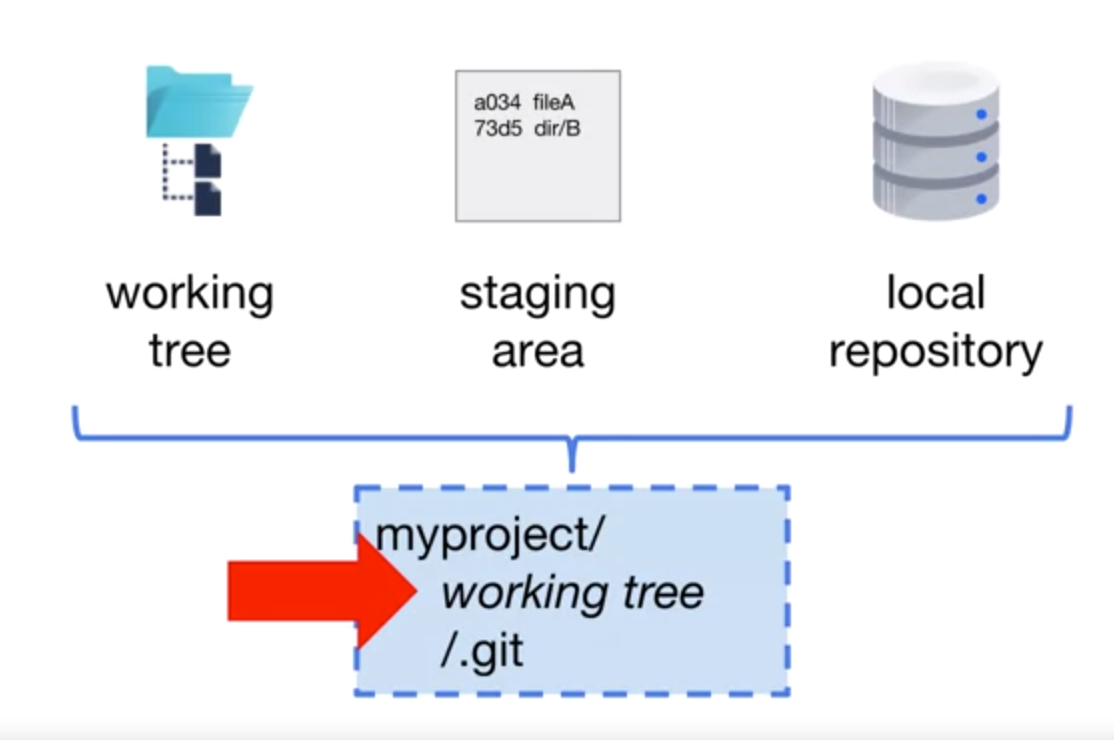
Files that are planned/ready for the next commit. **Add** command is used to achieve this.

## 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



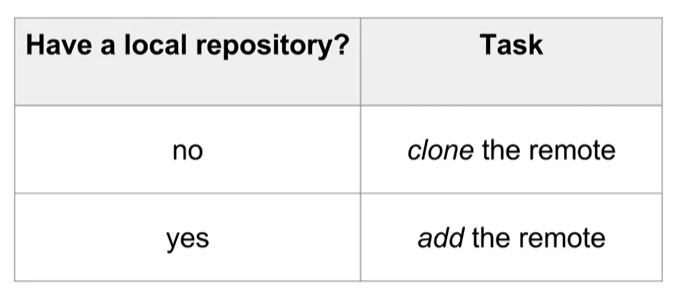
**echo “Someting” > FileA.txt**

When something is modified in staged file



Add again to get latest changes into stage

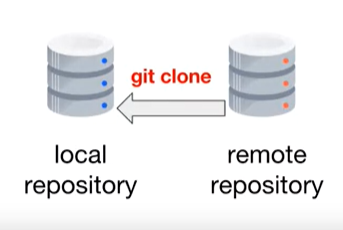
# Push to Report Repo



## Git Clone

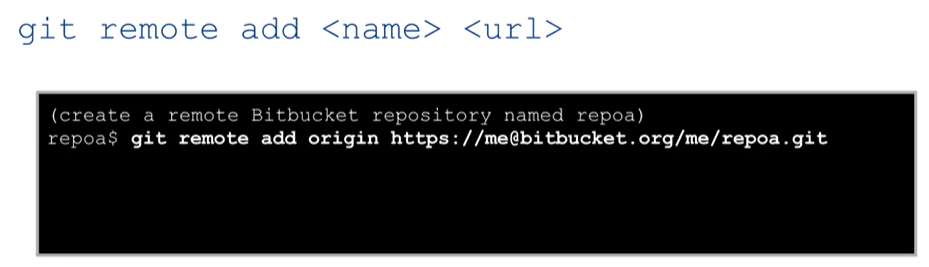


If project name specified (as in option in command) it will be used as project directory. If not specified, then name in the URL (minus the .git) will be used.



## Git Remote Add

When you have local repo already set and committed



# Git Graph

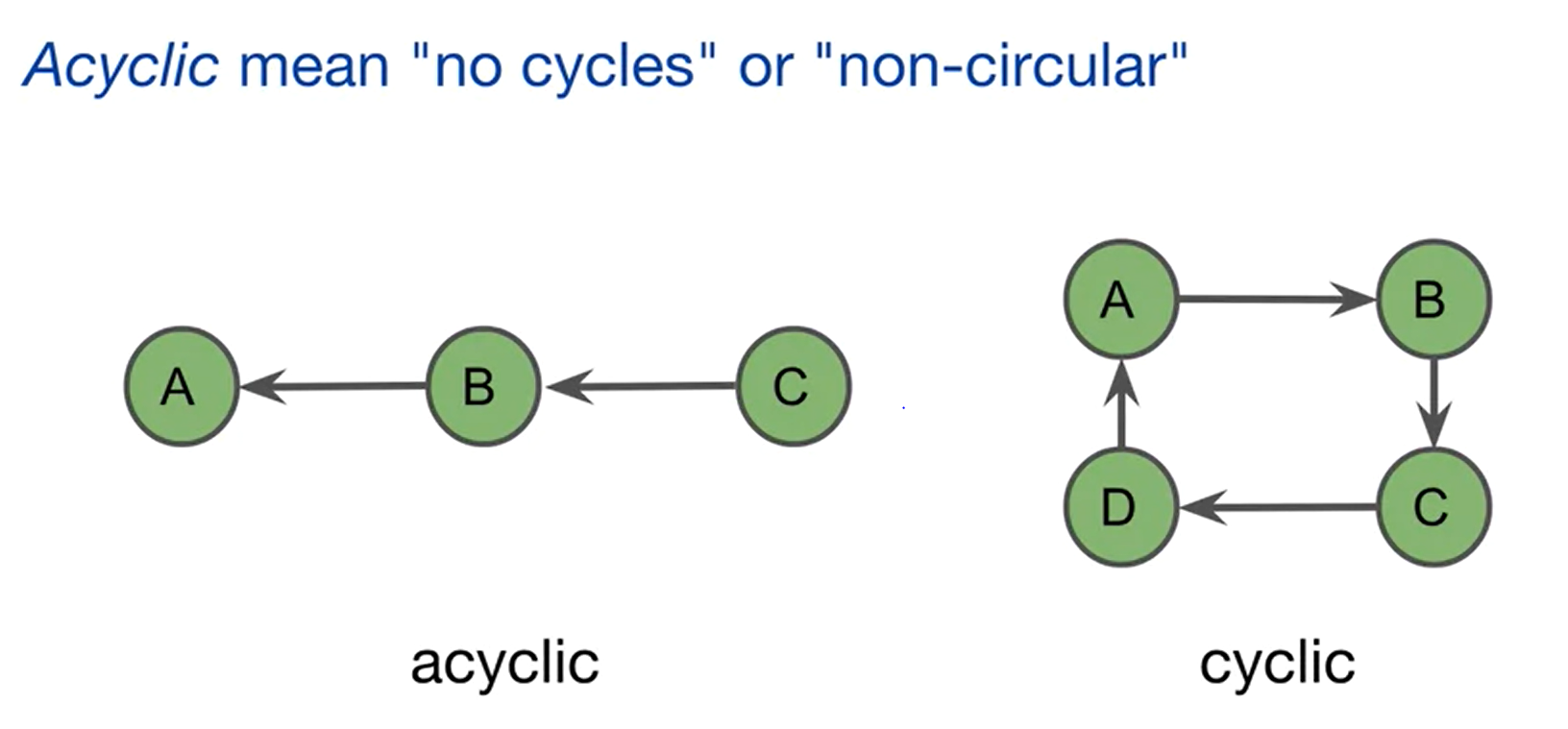
**gitk --all**

## Directed Graph:

A graph that has nodes pointing to a specific direction

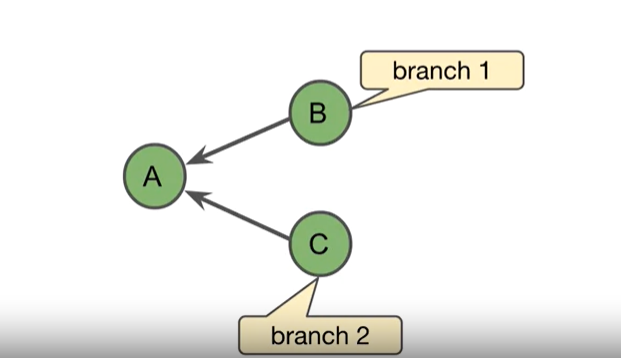
## Acyclic

Cannot end where it starts



## Branch

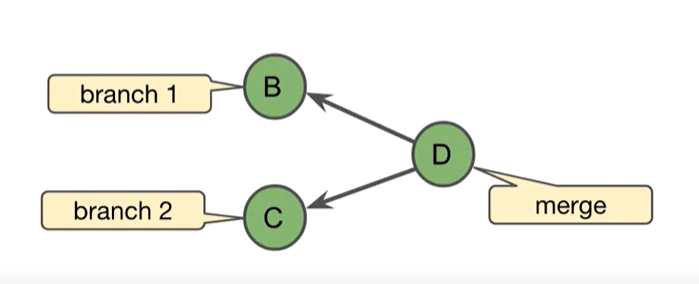
Branch occurs when a Commit has 2 (or more) child



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.

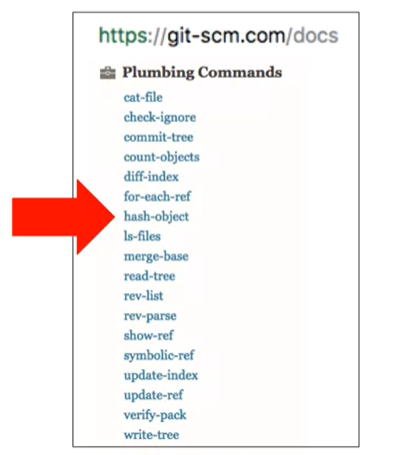
## SHA-1

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:

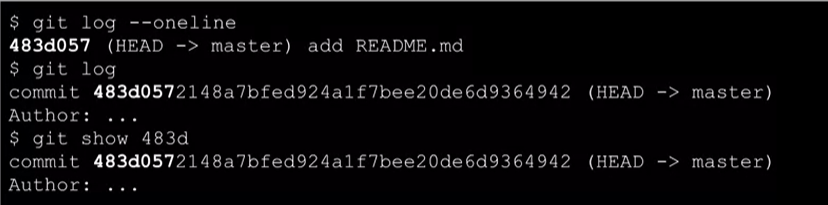
**git hash-object <file>**



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

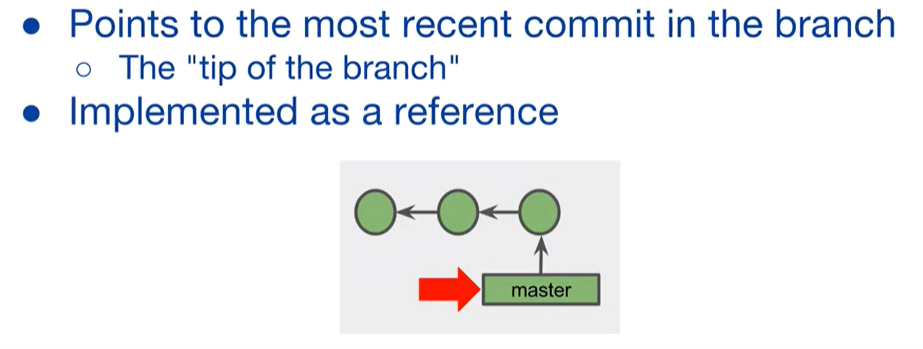
Git references help easily identify objects and retrieve information about hat object. For example, in above **git show <object id>** command, we can see that it has 2 References (HEAD & master).

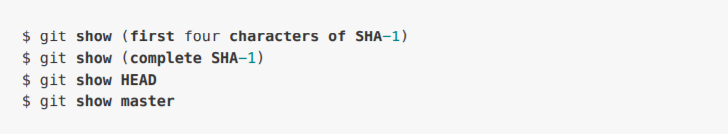
We can just do **git show HEAD** to see all the information about that object just as we can see with ID. Here HEAD is a reference to the branch label master.

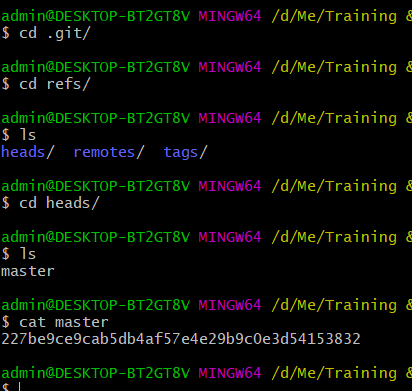
## Branch label

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.*** 



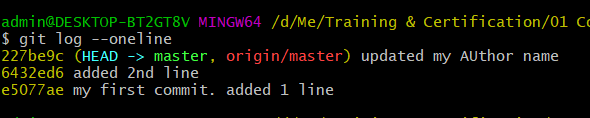
Above is an example of reference file, how git maintains it.

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

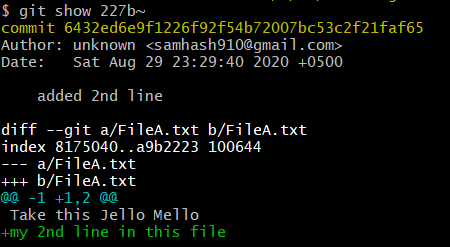
# Git references using TILDE character (~)

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

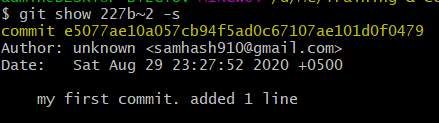
Below is whole history of commits, there are total 3 commits done.



Getting info of commit previous than the 227be

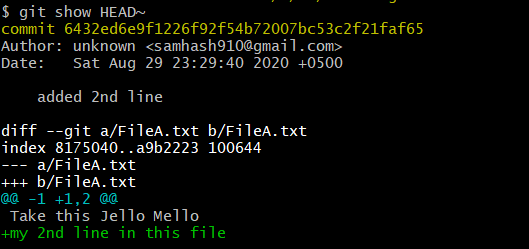


Getting info of 2 commits back from latest commit. (*-s shows short information)*



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



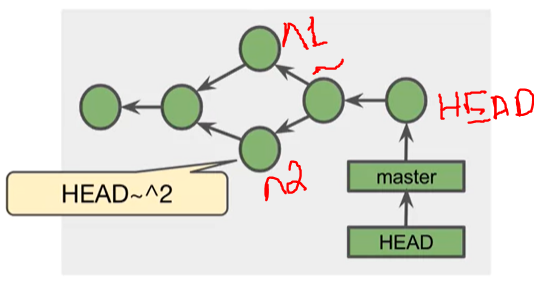
# Git references using Carrot character (^)

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

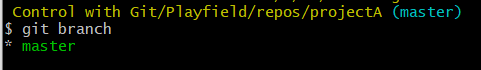
Flexible way to extend a project. usually there is 1 master branch and others are small features branches. Branch is just a reference in a file, so it is an edition of just 1 file to the project, making it super-efficient.

* 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

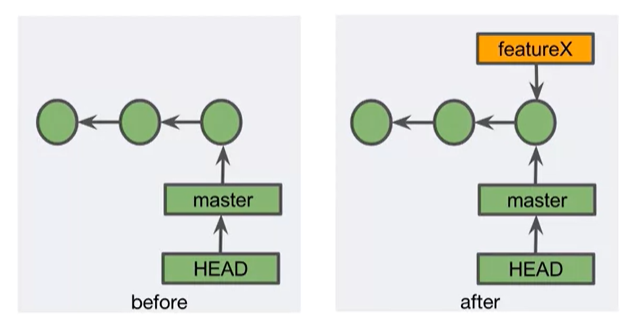


The asterisk (\*) shows which branch one is current branch

## 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.

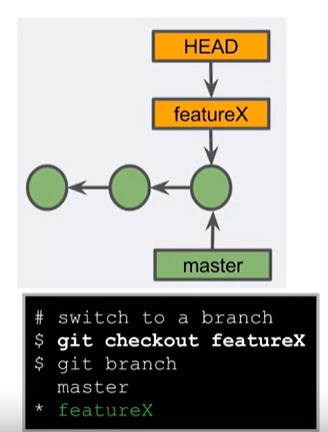
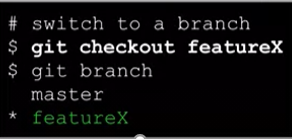
## Checkout

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.



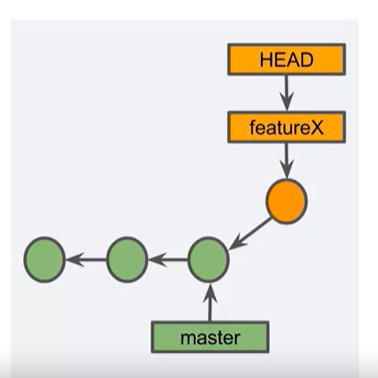
### Create and Checkout in same command

IT IS ONLY FOR NEW BRANCHES - FAILS FOR EXISTING BRANCHES

**git checkout -b feature**

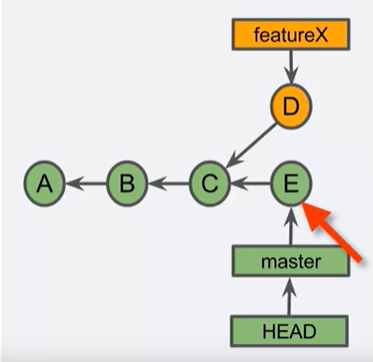
### The Next commit after Checkout

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.

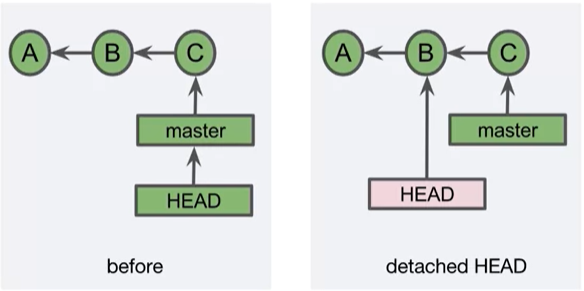


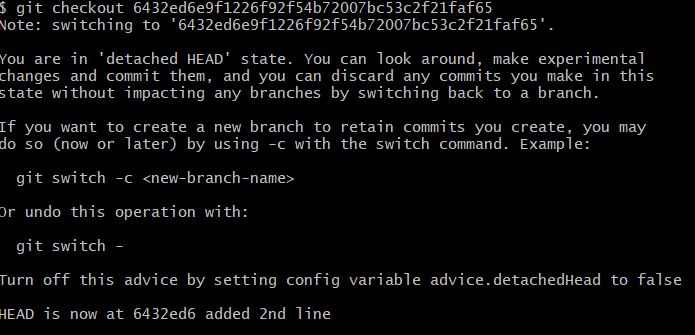
Neither master nor featureX knows about other branches, so they won’t interfere others.

## Detached Head (Pointing to a commit)

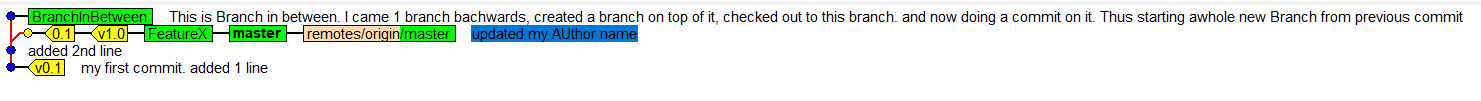
Usually you check out a branch, but what if we checkout a commit? One might want to go to the older version of the project and see that, or start a separate feature/branch on top of it.

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





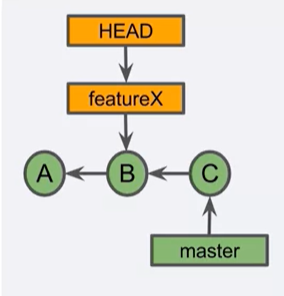
But this is not normal, so to make it normal, one should create a branch on top of it, and checkout to that branch.



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**



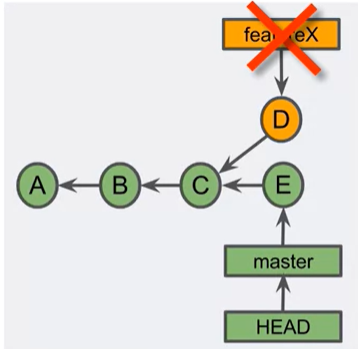
## Deleting Branches & Dangling Commits

**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.

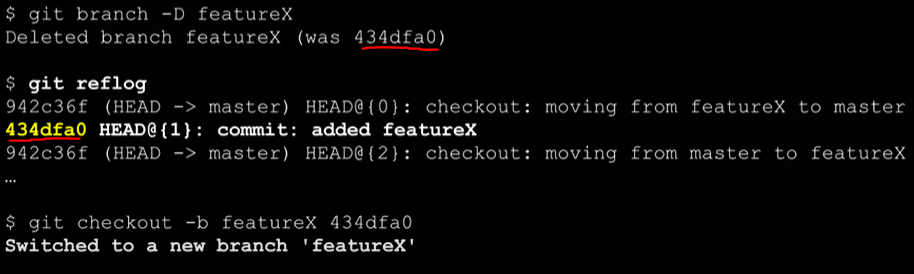
To forcefully delete that branch, use **git branch -D <branch-name>**



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.

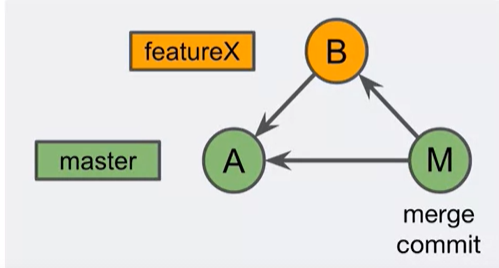
git reflog shows there that SHA-1 was pointing to a branch called featureX and HEAD was pointing to it.

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.

# Merge

Merging is Git's way of putting a forked history back together again. The git merge command lets you take the independent lines of development created by git branch and integrate them into a single branch

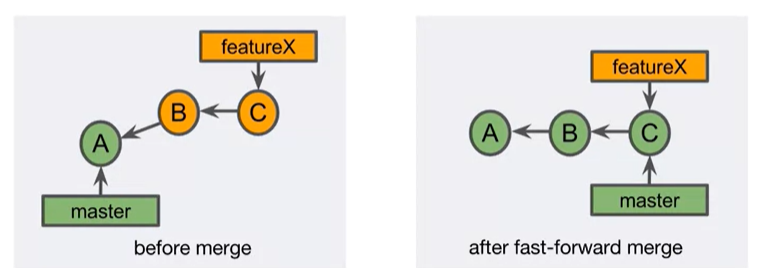


4 Types of merges

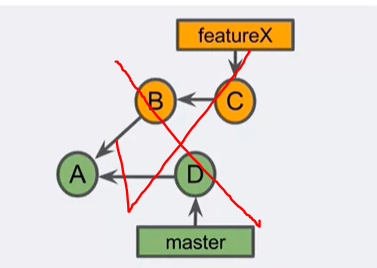
1. Fast-Forward merge
2. Merge commit
3. squash merge
4. Rebase

## Fast Forward Merge

It’s a type of merge in which the master branch starts directing towards latest commit of a forked out branch. This is only possible when master branch itself did not have any commits/changes time when separate branch was forked off and now. Otherwise master commits can be lost, so git does not allow it (that master or any other that is being merged has commits prior to forking off a separate branch).



Following scenario is not possible in Fast Forward Merge

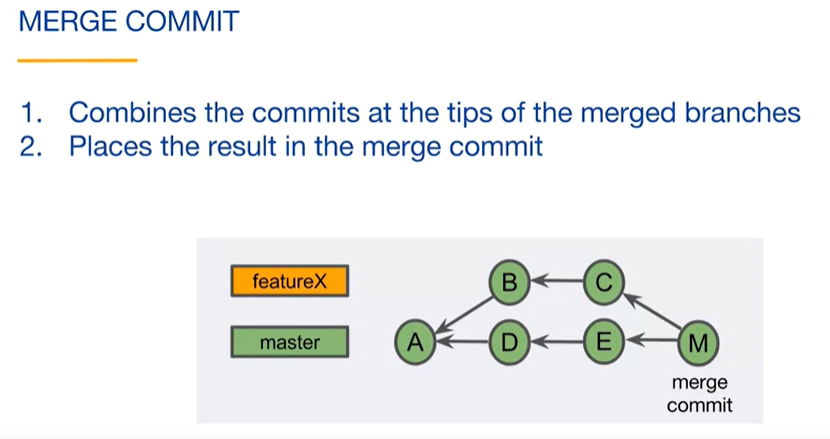


If we do fast-forward on it, the commit C will become the latest commit of master, destroying commit D in process.

Fast-forward merge is default merge of git unless any other is specified.

**Fast-forward merge is linear, there are no commits that have multiple parents**

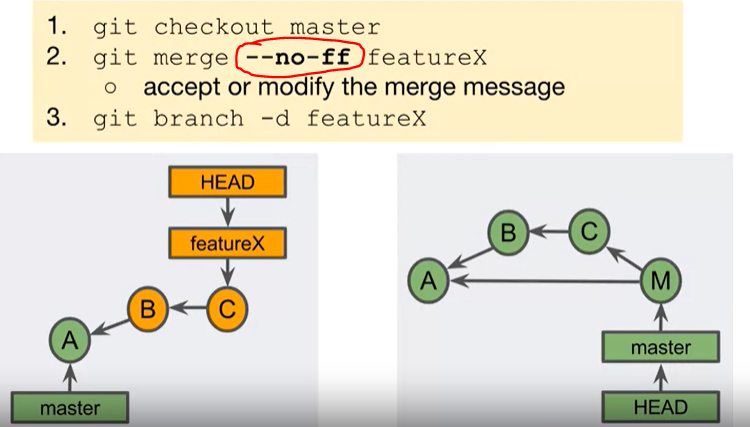
## Merge Commit



Always has multiple parents. Can result into some conflicts as FeatureX’ C commit can change something that was part of commit A. So when merged, conflict will arise. conflicts -> *Later in the document*

## Avoiding Merge commit

There could be some cases when company/project policy says that a commit can only be merge commit and not FF commit.



specifying **--no-ff** will force git to do a merge commit and make a different commit for merge as in above pic.

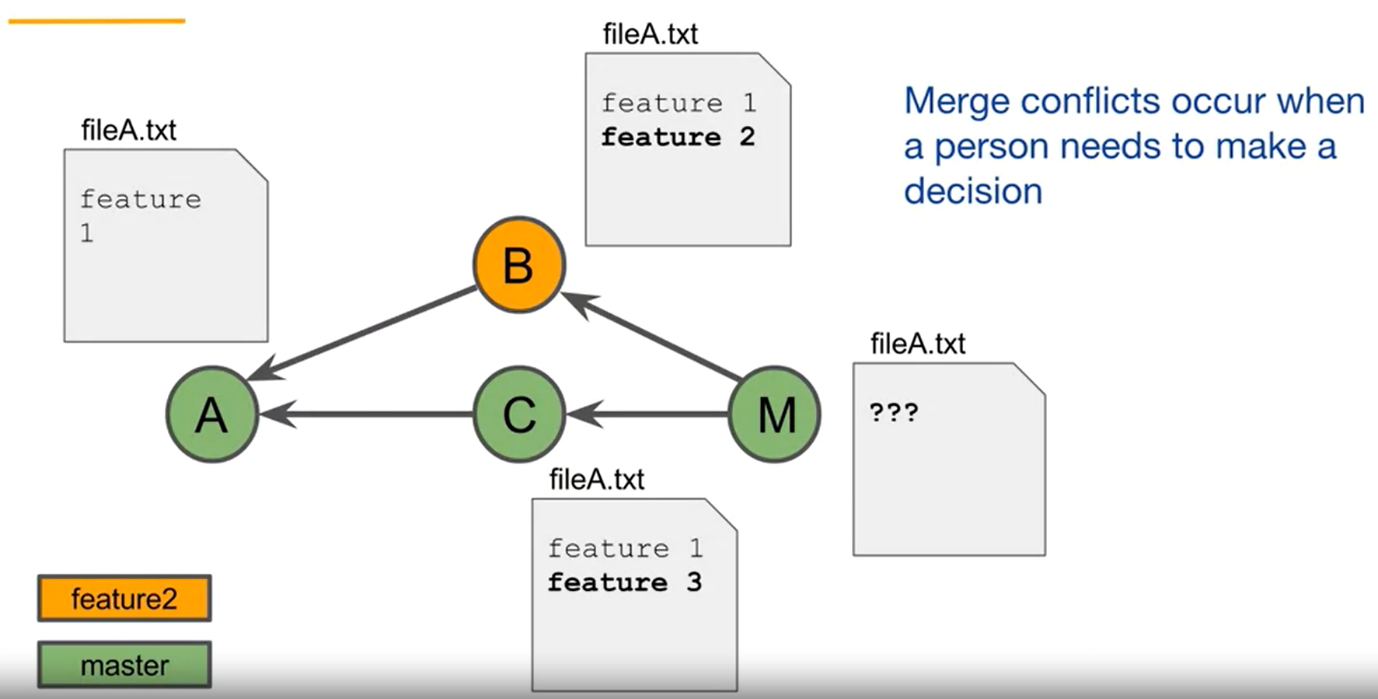
# Merge Conflicts

*Most important thing you have to remember: It is not essential that when there are changes within same file from different commits, either all the changes would have to be manually added to 1 branch or you will select only one (ours, theirs). NO. This will not always happen! Read further to know why…*

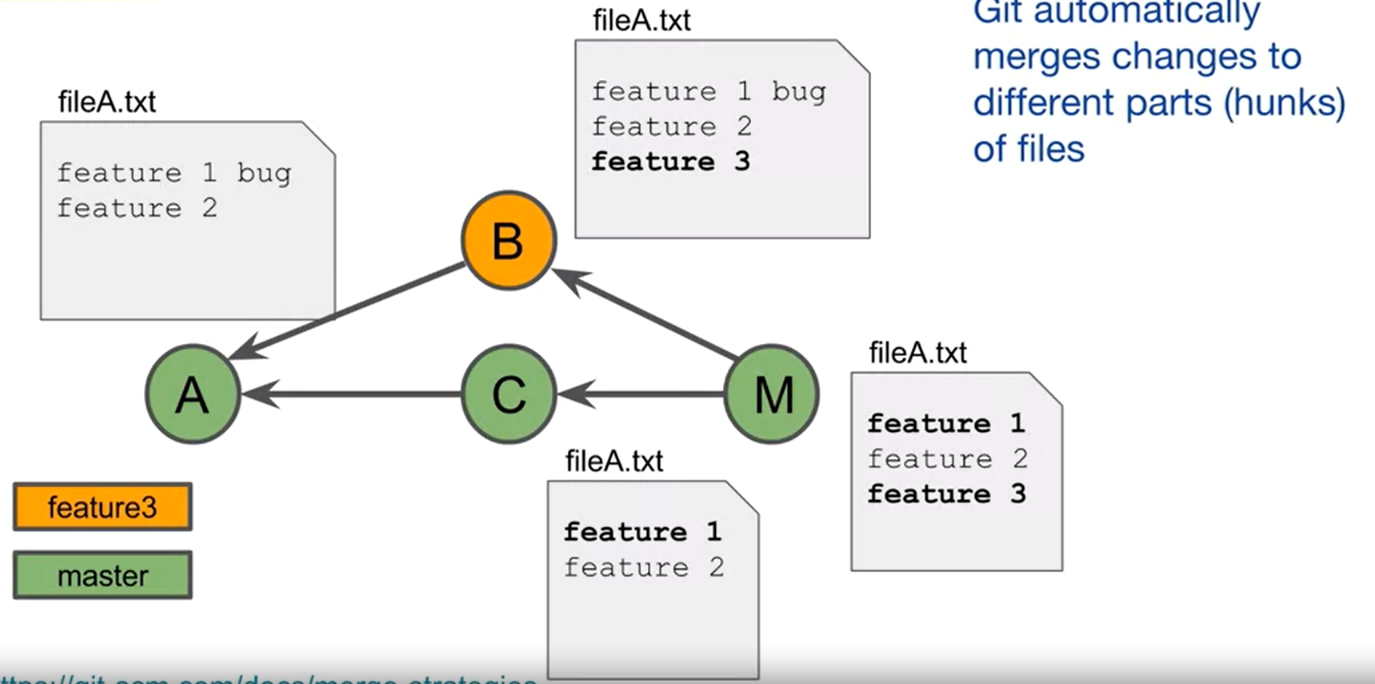
**A typical Merge Conflict scenario:**

A conflict occurs when same file or hunk of file has been changed in 2 commits and user tries to merge them. **Remember:** **In Conflict, user has to make a decision**

fileA was changed for both commits that will be merged i.e. C and B. User will make decision*,* either all the changes would have to be manually added to 1 branch (branch’s fileA) or you will select only one (ours or theirs)



Now below scenario also seems like a conflict and should result into a conflict. But it does not. This answers the question raised in 1 paragraph of this section.

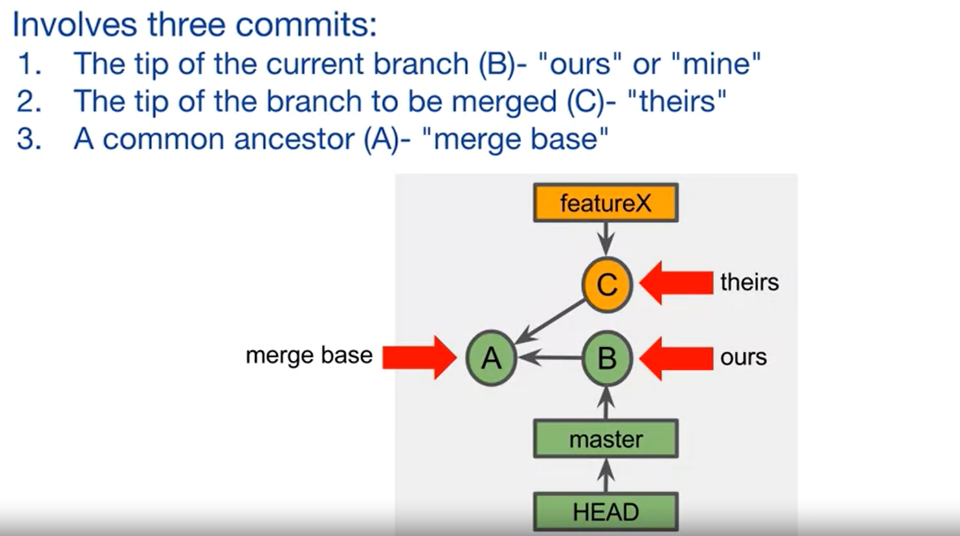


In this scenario:

* User has 2 features in first commit, 1 with bug.
* A new branch is made from here that implements feature 3.
* In the meantime, the bug in master branch was fixed and committed.
* Now when we merge, git will not raise a conflict.

That’s because Git can automatically merge changes to different part of a file. This **part** is called a **Hunk.**

## Resolving a Conflict



Remember the 3rd point, it’s an important concept that is usually forgotten. A merge is consisted of all 3 above commits.

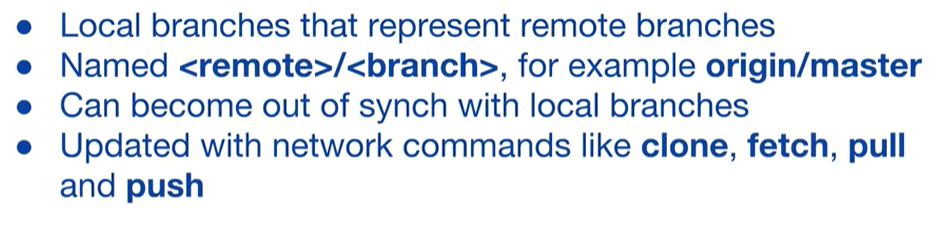
### Steps to Resolve:



You don’t need to checkout to master, it can be any base branch that you are maintaining. For example, you have a Developers branch which contains all the developed code. Every coder makes own branch for a feature/bug fix and then it is merged in this base branch after careful review (pull request, learned ahead)

# Tracking Branches

Tracking branches are:



**Tracking branches are automatically made/updated when we use network commands like pull, fetch, clone**

This command creates a new branch(not checked-out) that track a remote branch

**git branch -u upstream/foo foo**

Below command pushes a branch to remote (origin) and creates a tracking branch by using **-u**

**git push -u origin branch-name**

note that we have note specified remote branch as <remote>/<branch>. We learn that why we did that ahead.

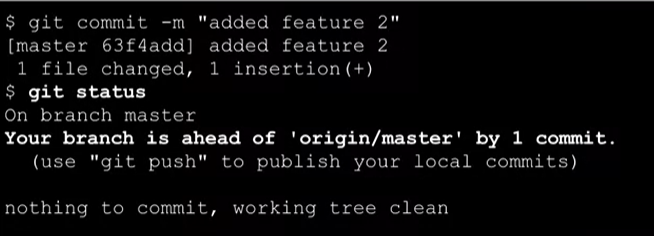
In below diagram, a remote repository was either cloned to local or a local repo was pushed using **-u** argument.

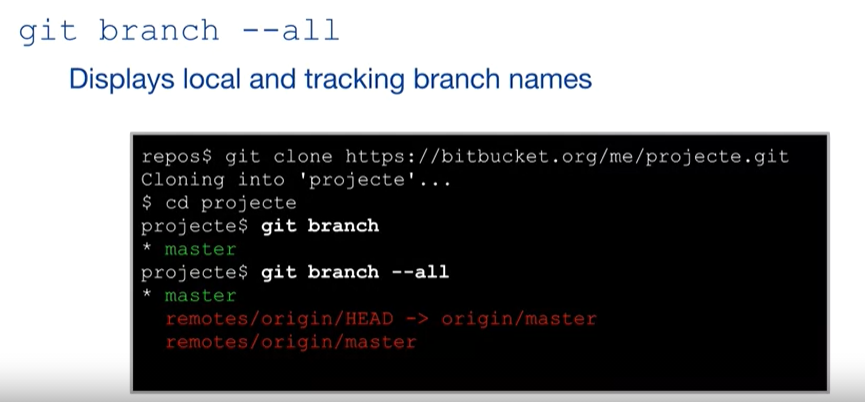
After that, on local repo, a new commit was created. And on remote repository, someone from the team made a different commit.

But the tracking branch i.e. origin/master remains where it was cloned/pushed. Note that when we make a change on local after creating tracking branch, the tracking branch will get 1 commit behind and it will also show you that when you do a **git commit.**



A local repo and tracking branch will not know of any commits to remote branch so it will not show you that it is now 1 or more commits behind when someone makes now commits on remote.



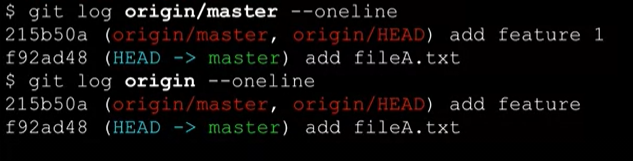


**git branch --all** command show 3 branches:

only **master** and **remotes/origin/master** are real branches in which master is normal and remote/origin/master is tracking branch, the 2nd branch in above output is a symbolic reference (reference to a reference)

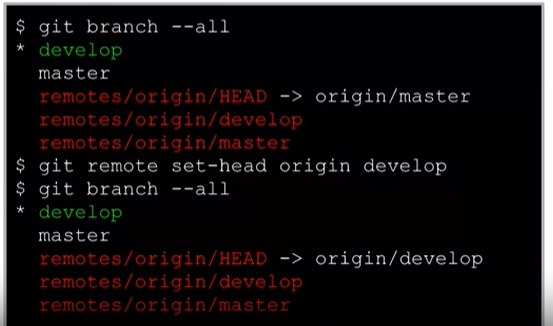
## Set Default Remote Tracking Branch

The output below shows similar logs for **origin** and **origin/master.** That is because the master branch of is set as default remote tracking branch (see which branch HEAD points to).



Below command changes the default remote tracking branch:

**git remote set-head <remote> <branch>**



**But this just sets the default remote branch on LOCAL. What if we want to set it for all users of a remote repo?**

**We would change default Remote Branch for all users in GITHUB/GITLAB/BITBUCKET settings.**

# Network Commands

## Fetch

**git fetch** command does not do any merges from remote repo but gets all the latest information like latest tracking branch, commit etc.

Fetch command usually automatically updates local tracking branch according to the remote and if there was any new commit on remote, running **git status** after **git fetch** will tell you how many commits local repo is behind origin (but that branch should be checked out i.e. if you have created -u tracking on a different branch than checkedout, it will not show desired results).

## Pull

Combines **git merge** fetch and **git merge FETCH\_HEAD**

FETCH\_HEAD is alias for tip of the tracking branch, so actually it is just merging the latest commit of tracking branch.

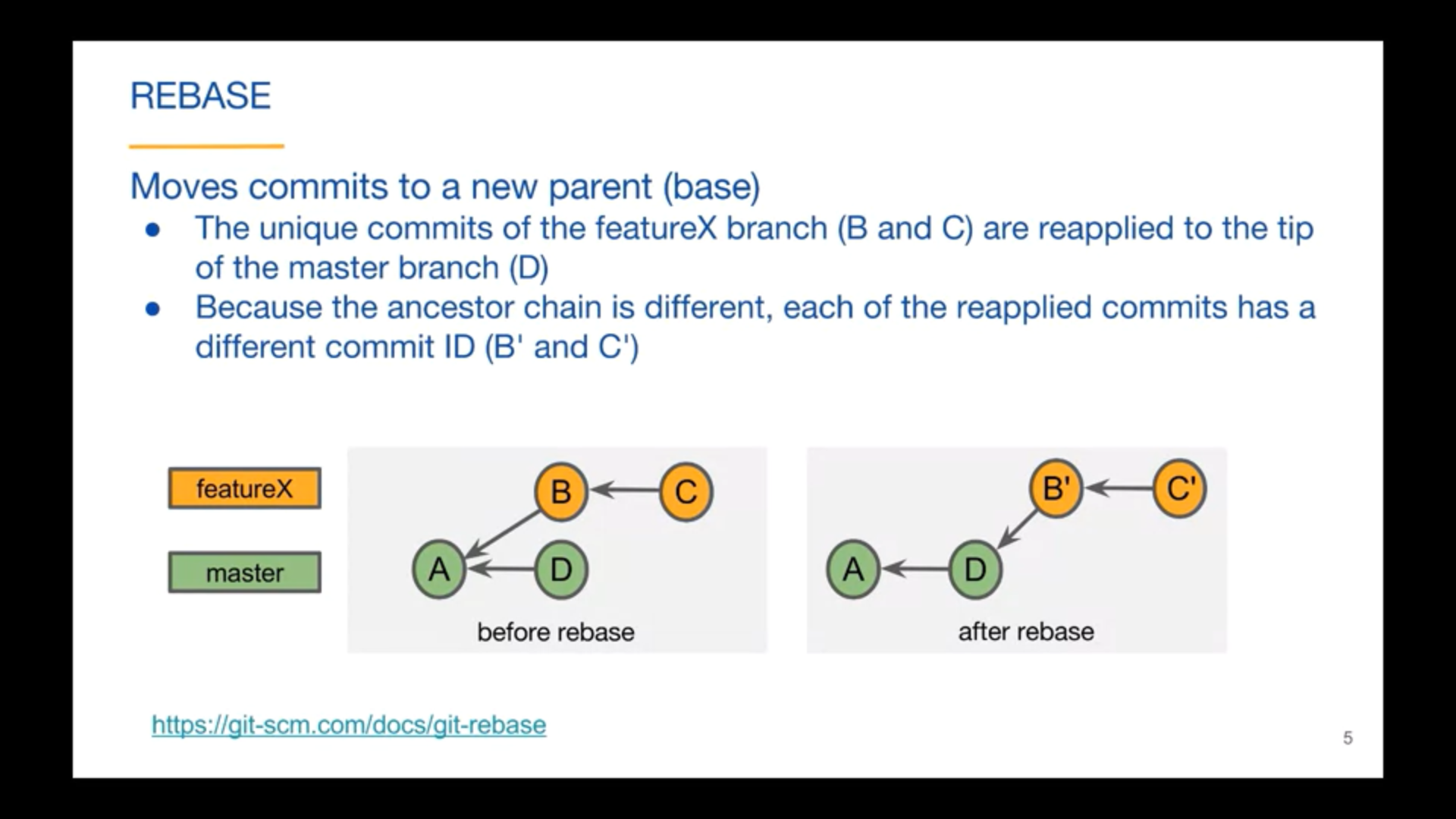
## Push

**git push -u repo branch**

This command pushes changes of checked-out local branch to a remote repo’s specific branch and also creates a tracking branch locally. Specifying -u, repo and repo’s branch is optional.

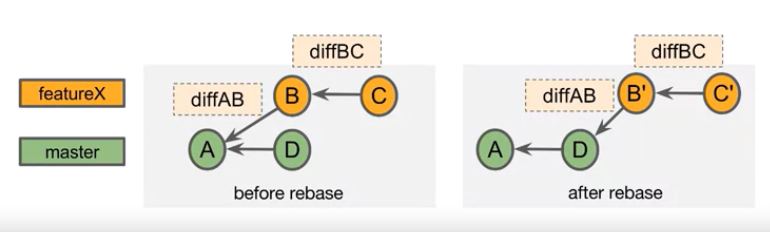
# Rebase

**Take Extreme Caution!**



Rebase does not preserve commit history, it rewrites it, so if one has shared its commit history with someone working on same branch, it will cause problems as commit IDs are changed in rebase.

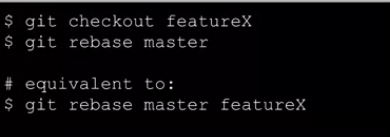
Rebase also does merge at the backend so Conflicts can also arise. n above diagram it is merging Diff of AB with D



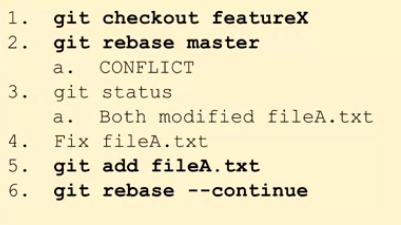
**Pros:**

Clean commit history. Makes a final merge a fast forward thus avoiding unnecessary merge commits.

It can be done 2 ways, either on checked-out branch, or without checking out to it:

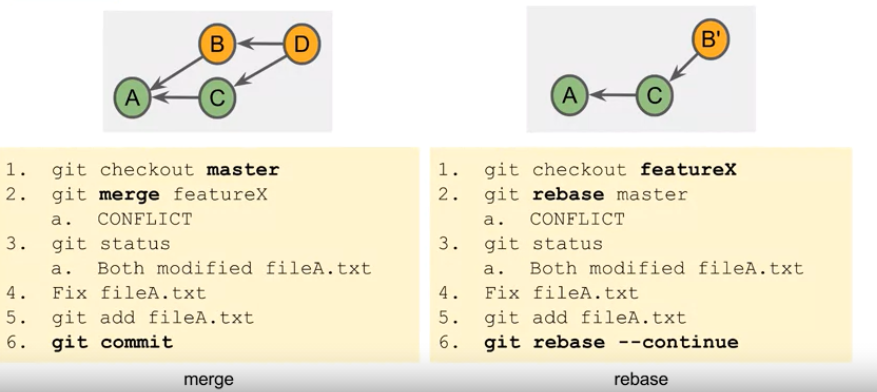


## Rebase with Merge conflict



You can also **abort** the rebase by instead of doing **point 6** above and typing **git rebase --abort**

## Conflicts - Difference in Rebase and Merge



Keep in mind that one can always **abort** on last point, by typing **git commit --abort** or **git rebase --abort**

# Interactive Rebase

## Amend

Change an already done commit by changing its content and not creating a new commit. This will certainly change the SHA-1 of the amended commit but will not create a new commit in the commit graph.

**git rebase -i <after-this-commit SHA-1>**

## Change previous Commit message (1 commit before latest)

Suppose you created a file, committed it, then later it you know there was a simple mistake in it. You can modify the file, do an **add** .

Then do a **git commit --amend -m “very small change”**

It will change the SHA-1 of the 1 previous commit from latest and apply the changes.

**Rename File:**

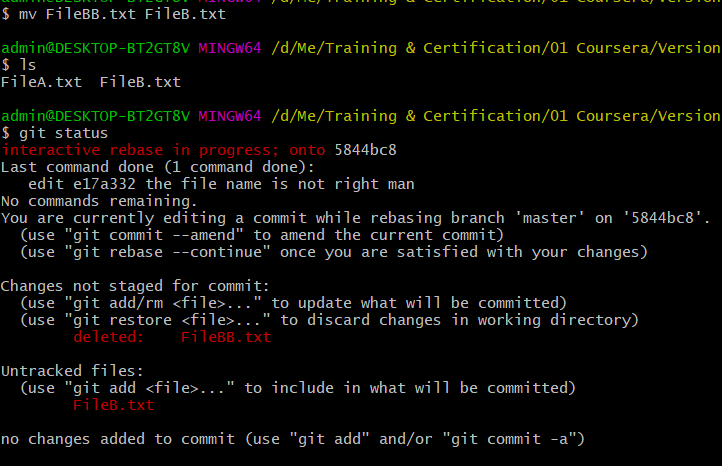
Renaming the file name that is committed. Use interactive rebase.

start from a specific SHA-1 of 1st commit by **git rebase -i <SHA-1>** command

It will open your default editor. Change the **Pick** to **Edit** in front of desired commit



close and save the editor. Now do your changes, for example changing the file name.



Above, file name is changed using **mv** command. Then we see **status** and it shows what we have done.

We now **Add** the latest changes. And do a **git rebase --continue**

Before doing continue, you can also do an **amend** to change the commit message. Use **git commit --amend -m “add fileB.txt”**

And then do a **continue.**

## Delete

Start with SHA-1 of the 1st commit



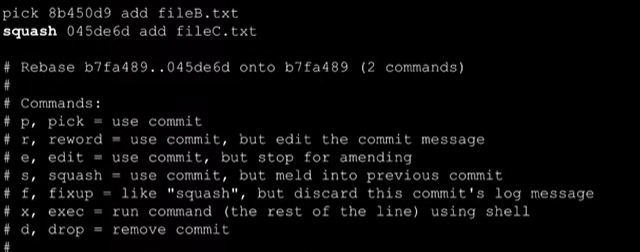
One can delete a commit if there is a mistake. By repeating the above procedure and just doing **Drop** instead of an **edit**

## Squash

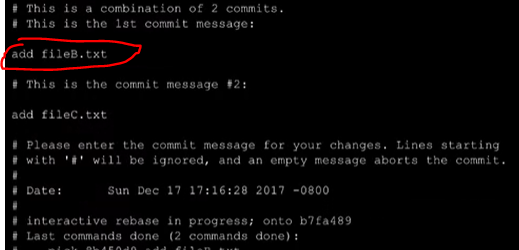
Squash combines work of 2 commits (latest and 1 before it) and creates a single commit in result. Thus replacing 2 commits with 1

The work of both commits is included.

Start with SHA-1 of the 1st commit



Just use **squash** instead of **pick**. After closing this a new window will open that will allow you to change the commit message.



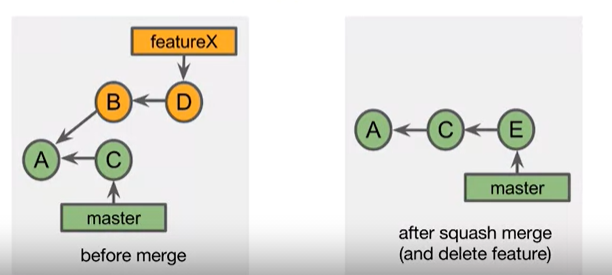
You have to change the 1st message i.e. of previous commit to show that on git logs with commit. The commit message #2 will be lost.

## Squash Merge

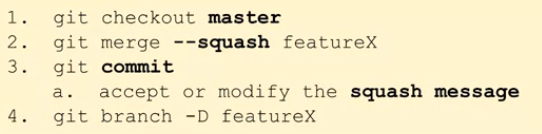
Merges the tip of **Feature** branch (any branch other than base) to the base branch (master, develop, release etc.)

There is a change of a merge conflict.

This command places the result into the **Staging Area**, which then needs to be **committed.**



To achieve above graph, do this:



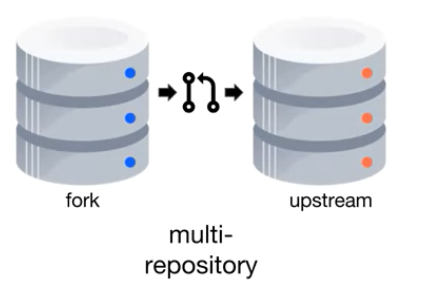
# Pull Request

Ultimate purpose of pull request is to merge a branch into the project

* Enables Teamwork and team communication related to work.
* Used for feedback and comments
* Notifications - sent to team members
* Approval of content - Can act as a form of **Code Review**

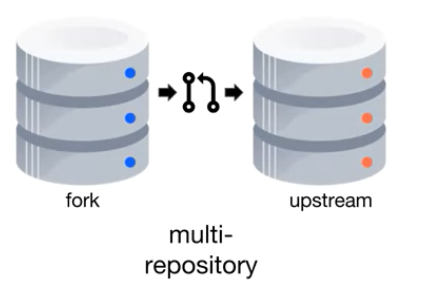
**Types:**

Single Repository type involves only 1 remote repository and developers make request to merge the branch in the remote repository from local.

In multi-repository involves 2 remote repositories. 1 repository act as fork. A pull request here is merging a branch from **fork** repository to **upstream.** Upstream is considered “Source of Truth” for project.

Common approach if the submitter does not have write access to the upstream repository. Very commonly used on open source projects where contributors have their own online repo and there is 1 repo of main project. The admins of main projects use fork repo to merge in main software code that lie on upstream.

## Forking



Copying a remote repo to your own online account. Both repos are remote, whereas in case of **clone** copy is made on your local repo.

### Fork Used for:

* experiment / Learning from upstream repo
* Issue pull request to upstream repo - contribute to projects.
* Create different source of truth - Start a different project.