GIT:

Version control:

* IT is also known as source control, is the practice of tracking and managing changes to files or software code.
* Version control software keeps track of every modification to the code in a special kind of database. If a mistake is made, developers can turn back the clock and compare earlier versions of the code to help fix the mistake while minimizing disruption to all team members.

Purpose of Version Control:

* Multiple people can work simultaneously on a single project. Everyone works on and edits their own copy of the files and it is up to them when they wish to share the changes made by them with the rest of the team.
* It also enables one person to use multiple computers to work on a project, so it is valuable even if you are working by yourself.
* It integrates the work that is done simultaneously by different members of the team. In some rare cases, when conflicting edits are made by two people to the same line of a file, then human assistance is requested by the version control system in deciding what should be done.
* Version control provides access to the historical versions of a project. This is insurance against computer crashes or data loss. If any mistake is made, you can easily roll back to a previous version. It is also possible to undo specific edits that too without losing the work done in the meanwhile. It can be easily known when, why, and by whom any part of a file was edited.

Functions and benefits of VCS:

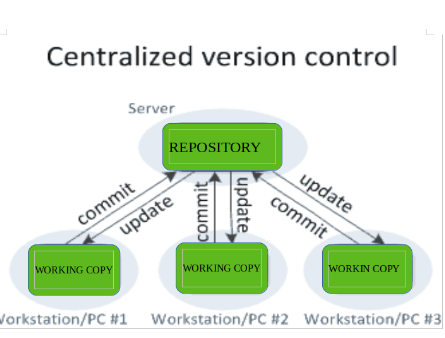
* Allows developers to work simultaneously.
* Does not allow overwriting each other’s changes.
* Maintains a history of every version
* Reduce possibilities of errors and conflicts meanwhile project development through traceability to every small change.
* Employees or contributors of the project can contribute from anywhere irrespective of the different geographical locations through this VCS.
* For each different contributor to the project, a different working copy is maintained and not merged to the main file unless the working copy is validated.
* Helps in recovery in case of any disaster or contingent situation.
* Informs us about Who, What, When, Why changes have been made.

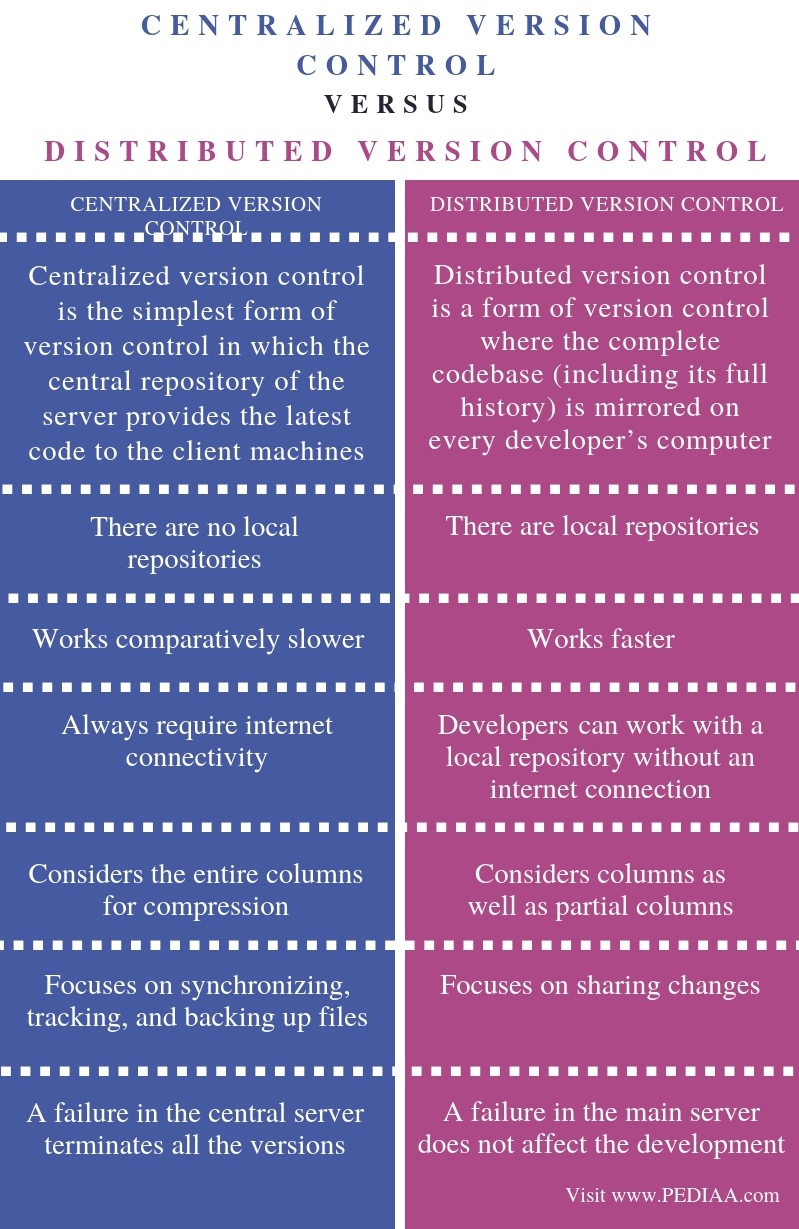
Types of VCS:

* Centralized version control system (CVCS).
* Distributed/Decentralized version control system (DVCS).

Centralized version control system (CVCS) uses a central server to store all files and enables team collaboration. But the major drawback of CVCS is its single point of failure, i.e., failure of the central server. Unfortunately, if the central server goes down for an hour, then during that hour, no one can collaborate at all. And even in a worst case, if the disk of the central server gets corrupted and proper backup has not been taken, then you will lose the entire history of the project.

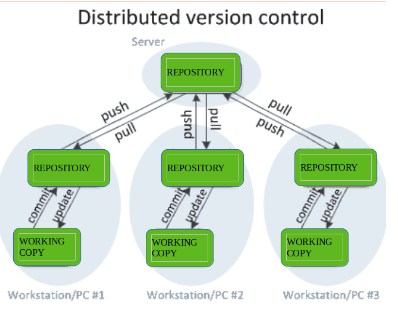
* The **benefit** of CVCS (Centralized Version Control Systems) makes collaboration amongst developers along with providing an insight to a certain extent on what everyone else is doing on the project.





Distributed version control systems contain multiple repositories. Each user has their own repository and working copy. If the sever goes down, then the repository from any client can be copied back to the server to restore it. Every checkout is a full backup of the repository. Git does not rely on the central server and that is why you can perform many operations when you are offline. You can commit changes, create branches, view logs, and perform other operations when you are offline. You require network connection only to publish your changes and take the latest changes.

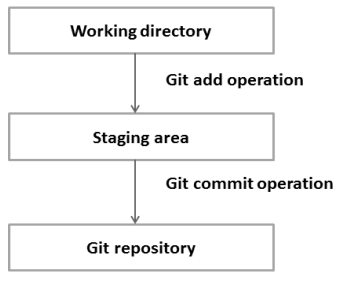
* Just committing your changes will not give others access to your changes. This is because commit will reflect those changes in your local repository and you need to push them in order to make them visible on the central repository. Similarly, when you update, you do not get others’ changes unless you have first pulled those changes into your repository.



Local Repository: Every VCS tool provides a private workplace as a working copy. Developers make changes in their private workplace and after commit, these changes become a part of the repository. Git takes it one step further by providing them a private copy of the whole repository. Users can perform many operations with this repository such as add file, remove file, rename file, move file, commit changes, and many more.

Working Directory and Staging Area or Index The working directory is the place where files are checked out. In other CVCS, developers generally make modifications and commit their changes directly to the repository. But Git uses a different strategy. Git doesn’t track each and every modified file. Whenever you do commit an operation, Git looks for the files present in the staging area. Only those files present in the staging area are considered for commit and not all the modified files. Let us see the basic workflow of Git.

* Step 1: You modify a file from the working directory.
* Step 2: You add these files to the staging area.
* Step 3: You perform commit operation that moves the files from the staging area. After push operation, it stores the changes permanently to the Git repository.

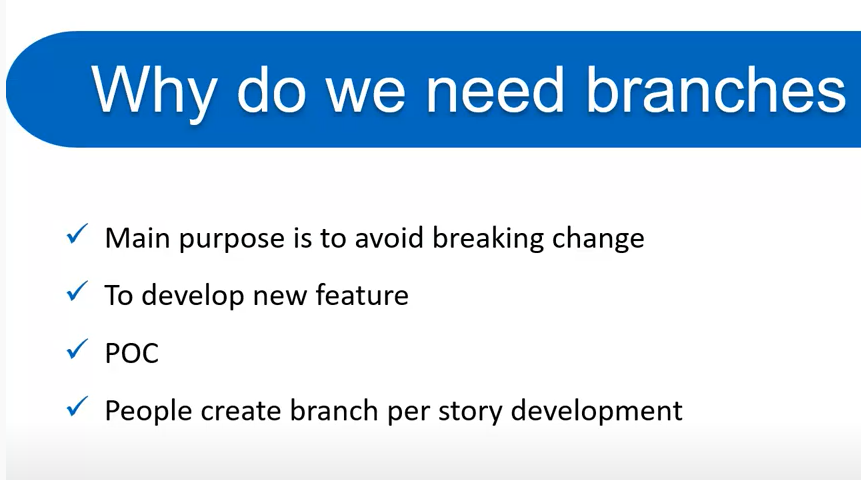


* Suppose you modified two files, namely “sort.c” and “search.c” and you want two different commits for each operation. You can add one file in the staging area and do commit. After the first commit, repeat the same procedure for another file.

Branches:

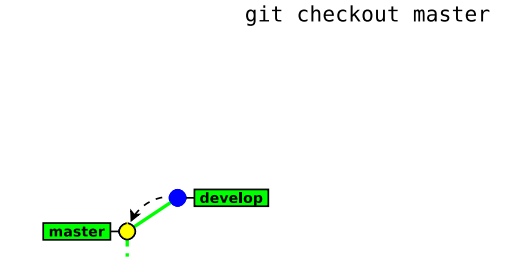
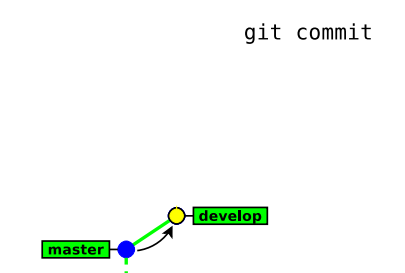


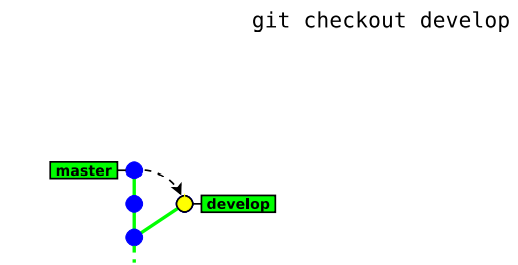
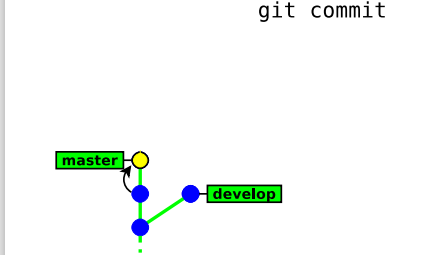
* You may have multiple variants of the same software, materialised as branches.
* VC tools will help you to handle multiple branches concurrently.
* Merge changes from a branch into another one.

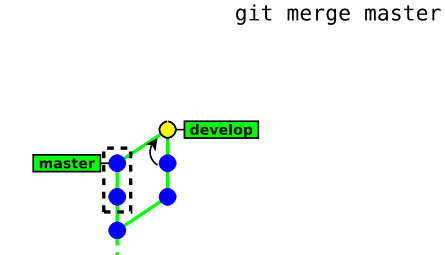
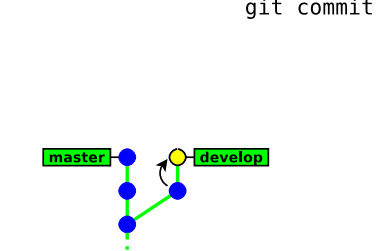


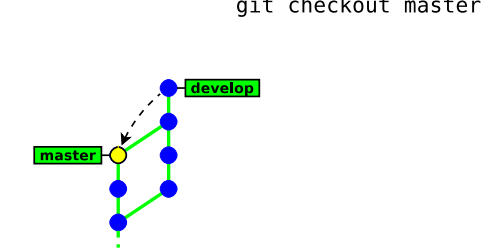
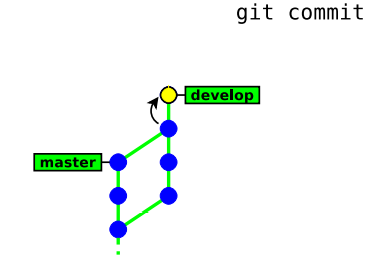
* Branching allows each developer to branch out from the original code base and isolate their work from others. It also helps Git to easily merge versions later on.
* Creating separate Branch helps developers to work on features and bugs independently.

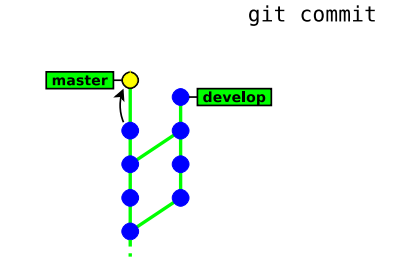


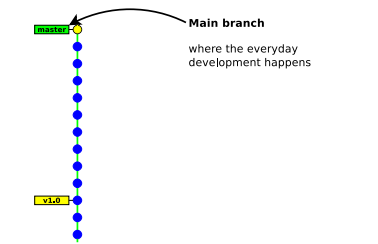
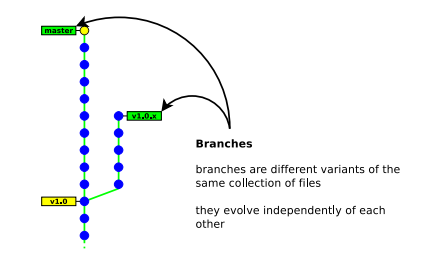


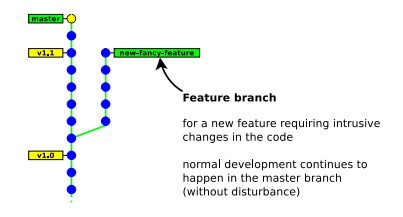
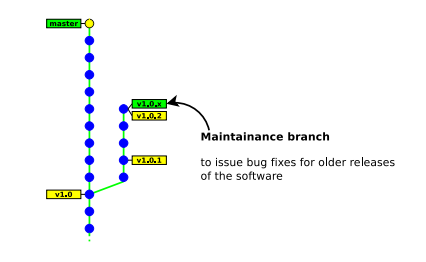


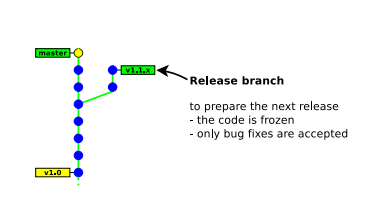
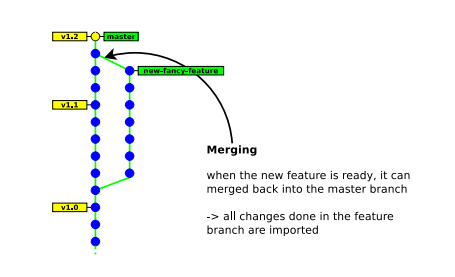


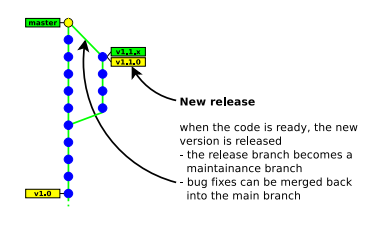
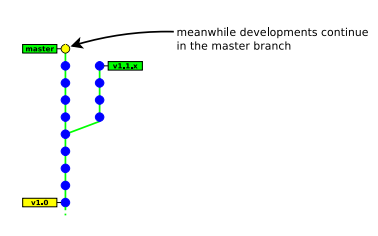










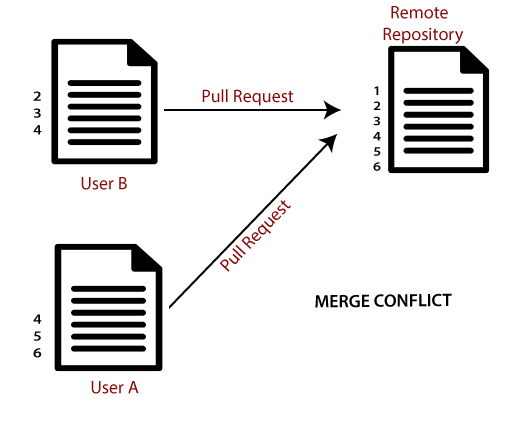


* **Creating a new branch:** git checkout -b new branch [ starting point ]
* **Switching between branches:** git checkout [-m] branch name  
  Note: it may fail when the working copy is not clean. Add -m to request merging your local changes into the destination branch.
* **Merging a branch:** This will merge the changes in other branch into the current branch.
  + The result of git merge is immediately committed (unless there is a conflict)
  + The new commit object has two parents. → the merge history is recorded
  + git merge applies only the changes since the last common ancestor in the other branch
  + if the branch was already merged previously, then only the changes since the last merge will be merged.

**Merging files:** f the same file was independently modified in the two branches, then Git needs to merge these two variants.

## **Git Merge Conflict**

When two branches are trying to merge, and both are edited at the same time and in the same file, Git won't be able to identify which version is to take for changes. Such a situation is called merge conflict. If such a situation occurs, it stops just before the merge commit so that you can resolve the conflicts manually.

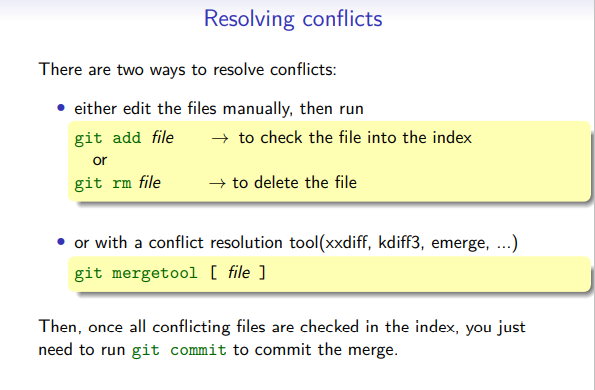


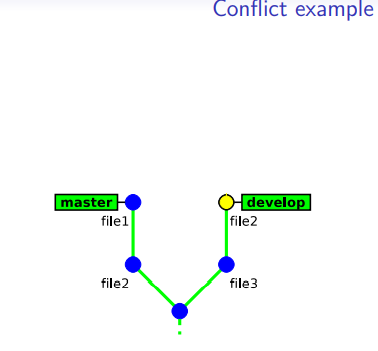
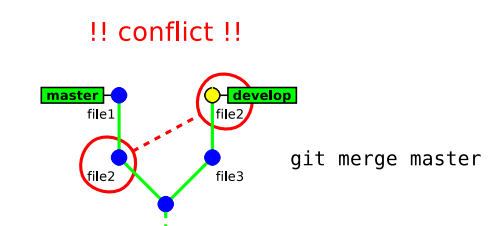
## Resolve Conflict:

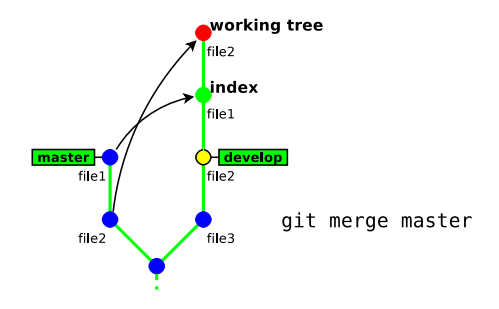
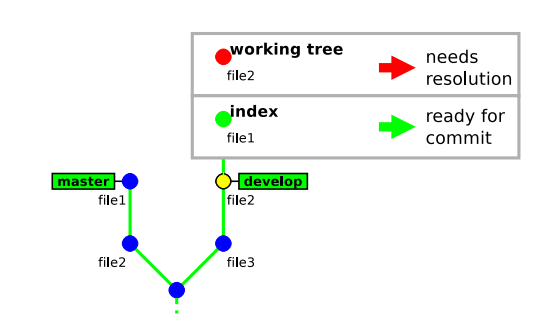
To resolve the conflict, it is necessary to know whether the conflict occurs and why it occurs. Git merge tool command is used to resolve the conflict. The merge command is used as follows:

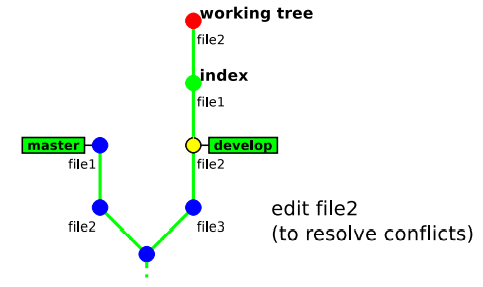
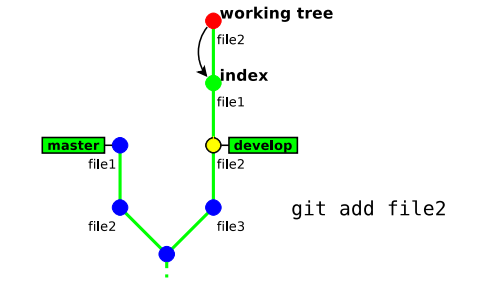
**$ git mergetool**

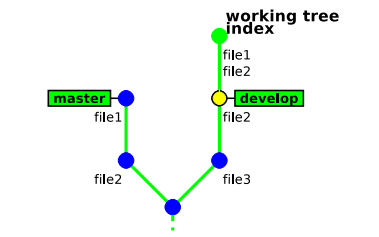
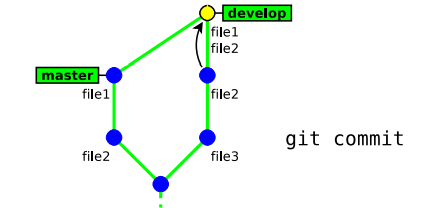
* textual files are merged on a per-line basis:
* lines changed in only one branch are automatically merged
* If a line was modified in the two branches, then Git reports a conflict. Conflict zones are enclosed within <<<<<<< >>>>>>>
* Binary files always raise a conflict and require manual merging.
* In case of a conflict:
  + unmerged files (those having conflicts) are left in the working tree and marked as “unmerged”10
  + the other files (free of conflicts) and the metadata (commit message, parents commits, ...) are automatically added into the index (the staging area)



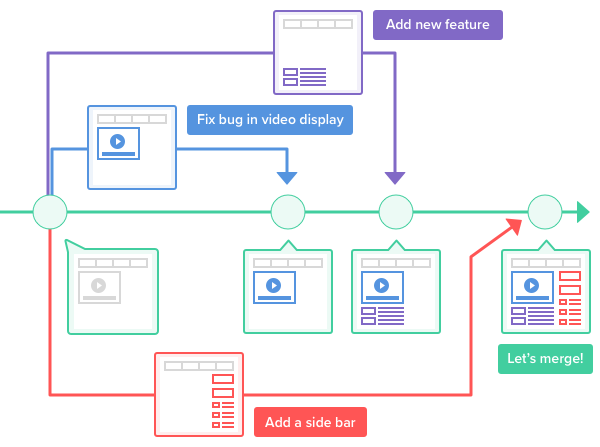
 

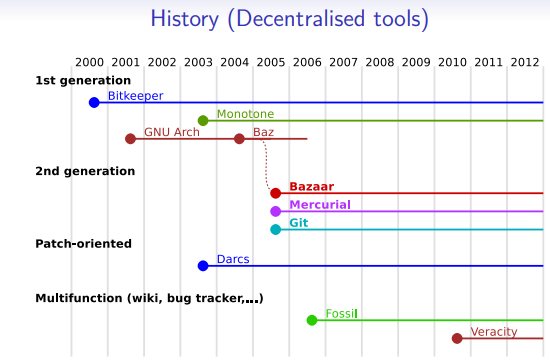
* **git branch -d branch name -** This command has some restrictions, it cannot delete:, the current branch (HEAD),a branch that has not yet been merged into the current branch

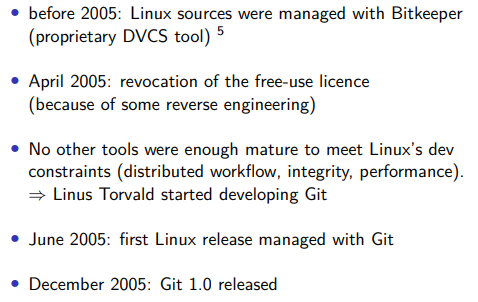
Pull Requests:

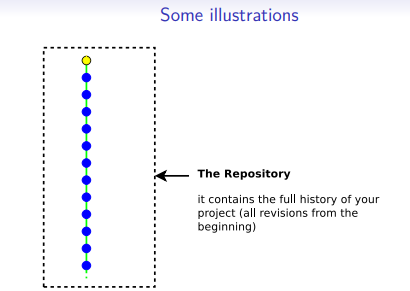
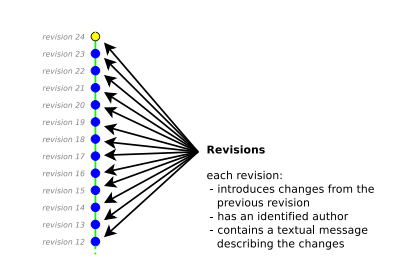
* Merging a branch in to main branch is possible by Pull requests
* Pull req is used to merge a branch with other branch
* If any merging code is faulty we can back out the changes from there it self

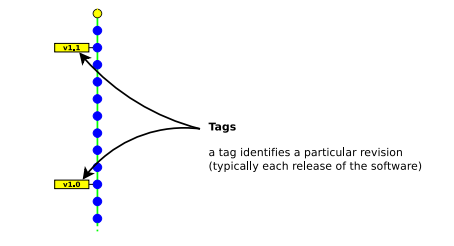
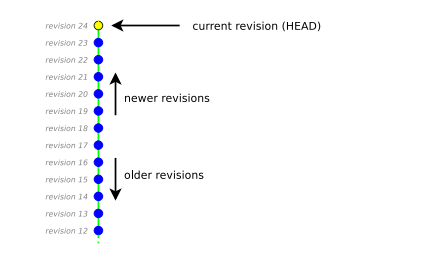


GIT History:









Creating Repository and Committing:

* **git init myrepository**
* This command creates the directory myrepository.
* The repository is located in myrepository/.git
* the (initially empty) working copy is located in myrepository/.

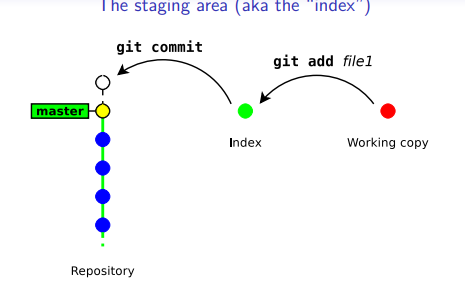
**Note**: “master” is the name of the default branch created by git init.

**Note:** The /.git/ directory contains your whole history, do not delete it unless your history is merged into another repository.

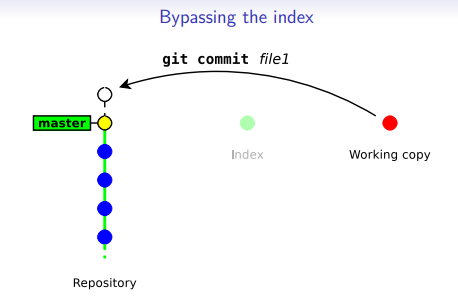
* **git add file -** To add the file
* **git commit [ -m message ] –** To committhe file.

**Staging Area:**

* Usual version control systems provide two spaces:
  + The repository (the whole history of your project)
  + The working tree (or local copy) (the files you are editing and that will be in the next commit)
* Git introduces an intermediate space: the staging area (also called index).
* The index stores the files scheduled for the next commit:
  + git add files → copy files into the index
  + git commit → commits the content of the index



* Running git add & git commit for every iteration is tedious (too long or slow).
* GIT provides a way to bypass the index.
  + **git commit file1 [ file2 . . . ]-**This command commits files (or dirs) directly from the working tree.
* Note: when bypassing the index, GIT ignores new files:
* “git commit .” commits only files that were present in the last commit (updated files)
* “git add . && git commit” commits everything in the working tree (including new files)

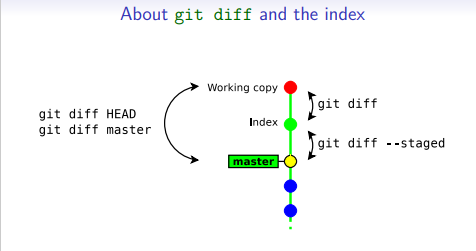


Delete files

* **git rm file** → remove the file from the index and from the working copy
* **git commit** → commit the index

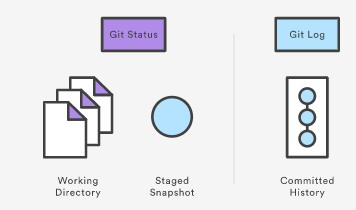
Showing difference: ----- Later

* **git diff [ rev a [ rev b ] ] [ -- path . . . ]** → shows the differences between two revisions rev a and rev b (in a format suitable for the patch utility)
  + by default rev a is the index
  + by default rev b is the working copy
* **git diff --staged [ rev a ] [ -- path . . . ]-** shows the differences between rev a and the index.
  + **B**y default rev a is HEAD (a symbolic references pointing to the last commit).



**Other commands:**

* **git status** → show the status of the index and working copy
* **git show** → show the details of a commit (metadata + diff)
* **git log** → show the history ,display the entire commit history using the default formatting. If the output takes up more than one screen, you can use Space to scroll and q to exit.
  1. **git log -n <limit>-** Limit the number of commits by . For example, git log -n 3 will display only 3 commits.
  2. **git log <file>-** display commits that include the specified file
  3. **git log --author="koushik" -p hello.py -**This will display a full diff of all the changes koushik has made to the file hello.py.
  4. **git log –oneline**
  5. **git log –stat**
  6. **git log –p -**Display the patch representing each commit. This shows the full diff of each commit, which is the most detailed view you can have of your project history.
  7. **git log --author="<pattern>"-** search commits by a particular author.
  8. **git log --grep="<pattern>" -** Search for commits with a commit message that matches , which can be a plain string or a regular expression.
  9. **git log <since>..<until>-**Show only commits that occur between < since > and < until >. Both arguments can be either a commit ID, a branch name, HEAD
  10. **git log --graph --decorate –oneline -**The --graph flag that will draw a text based graph of the commits on the left hand side of the commit messages. --decorate adds the names of branches or tags of the commits that are shown. --oneline shows the commit information on a single line making it easier to browse through commits at-a-glance.
  11. **git log --oneline main..some-feature-** displays a brief overview of all the commits that are in some-feature that are not in main.
* **git mv** → move/rename a file
* **git reset**-> cancels the changes in the index (and possibly in the working copy)
  1. git reset drops the changes staged into the index8 , but the working copy is left intact
  2. **git reset --hard** drops all the changes in the index and in the working copy
* **git checkout - -path -> T**his command restores a file (or directory) as it appears in the index (thus it drops all unstatged changes)
* **git tag** → creating/deleting tags (to identify a particular revision)



Commands:  
  
Git config: This command configures the user. The Git config command is the first and necessary command used on the Git command line. This command sets the author name and email address to be used with your commit.  
  
$ git config --global user.name "XXXXXXXXx"

**$ git config --global user.email "xxxxxxxx**[**@gmail.com**](mailto:Himanshudubey481@gmail.com)**"**   
**Git Init :** This command is used to create a local repository

**$ git init Demo**

Git clone used to make a copy of a repository from an existing URL

**$git clone** [**https://geico.visualstudio.com/xxxxxxxxxxxxxxxxxx**](https://geico.visualstudio.com/xxxxxxxxxxxxxxxxxx)

Git add used to add one or more files to staging (Index) area.

**$ git add Filename or $ git add \***  
**$git rm file --> to delete**

Git commit used to commit the files. It records or snapshots the file permanently in the version history with a message.

**$ git commit -m " Commit Message"**

Git status The status command is used to display the state of the working directory and the staging area. It allows you to see which changes have been staged, which haven't, and which files aren?t being tracked by Git

**$ git status**

Git push It is used to upload local repository content to a remote repository. Pushing is an act of transfer commits from your local repository to a remote repo

**git push --set-upstream origin "branchname"**

Git pull It fetches and merges changes on the remote server to your working directory.

**$ git pull**

Git Merge used to merge the specified branch?s history into the current branch

**$ git merge BranchName**

Git checkout used to switch the branch in git

**$ git checkout -b <new\_branch>**

**$ git checkout <existing\_branch>**

Or   
$ git switch <existing\_branch>   
$ git switch -c <non\_existing\_branch>

Git log used to check the commit history

* + **$ git log**   
    **git log -n <limit>-** Limit the number of commits by . For example, git log -n 3 will display only 3 commits.
  + **git log <file>-** display commits that include the specified file
  + **git log --author="koushik" -p hello.py -**This will display a full diff of all the changes koushik has made to the file hello.py.
  + **git log –oneline**
  + **git log –stat**
  + **git log –p -**Display the patch representing each commit. This shows the full diff of each commit, which is the most detailed view you can have of your project history.
  + **git log --author="<pattern>"-** search commits by a particular author.
  + **git log --grep="<pattern>" -** Search for commits with a commit message that matches , which can be a plain string or a regular expression.
  + **git log <since>..<until>-**Show only commits that occur between < since > and < until >. Both arguments can be either a commit ID, a branch name, HEAD
  + **git log --graph --decorate –oneline -**The --graph flag that will draw a text based graph of the commits on the left hand side of the commit messages. --decorate adds the names of branches or tags of the commits that are shown. --oneline shows the

Git remote used to connect your local repository to the remote server. This command allows you to create, view, and delete connections to other repositories.

**$ git remote add >>>>>>>>>>.**

Git Revert The git revert command is used to apply revert operation

**$ git revert <commit-ish>**

Git diff used to show changes between commits, commit, and working tree, etc.

* **git diff [ rev a [ rev b ] ] [ -- path . . . ]** → shows the differences between two revisions rev a and rev b (in a format suitable for the patch utility)
  + by default rev a is the index
  + by default rev b is the working copy
* **git diff --staged [ rev a ] [ -- path . . . ]-** shows the differences between rev a and the index.
  + **B**y default rev a is HEAD (a symbolic references pointing to the last commit).
* **git mv** → move/rename a file
* **git reset**-> cancels the changes in the index (and possibly in the working copy)
  1. git reset drops the changes staged into the index8 , but the working copy is left intact
  2. **git reset --hard** drops all the changes in the index and in the working copy

GITHUB: To provide Internet hosting for version control and software development. GitHub is a service hosted on the web.  
Gitstash: Let’s say you're a developer and you want to switch branches to work on something else. The issue is you don’t want to make commits in uncompleted work, so you just want to get back to this point later. The solution here is the Git stash.

Git stash takes your modified tracked files and saves it on a stack of unfinished changes that you can reapply at any time. To go back to the work, you can use the stash pop.

**$ git stash**    
To stash a change with a message

**$ git stash save "<Stashing Message>"**

To check the stored stashes,

**$ git stash list**

Popping your stash removes the changes from your stash and reapplies them to your working copy.

**git stash pop**

e **HEAD** points out the last commit in the current checkout branch

GIT fetch : Git Fetch is the command that tells the local repository that there are changes available in the remote repository without bringing the changes into the local repository. (yet doesn’t do any file transferring. It’s more like just checking to see if there are any changes available).

git clean command removes the untracked files from the working directory.

Disks:   
git cherry-pick used to move specific commits from one branch of a repository to another.   
Cherry picking is a technique used in version control systems, such as Git, to select and apply specific commits from one branch to another. It allows you to choose individual commits and apply them to a different branch without merging the entire branch history.

**git reflog** This command tracks every single change made in the repository references

* open Git and clone with azure devops(by copying azure devops link and pasting in cmd using. **Git clone link**