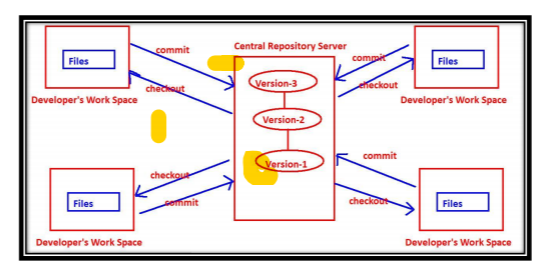
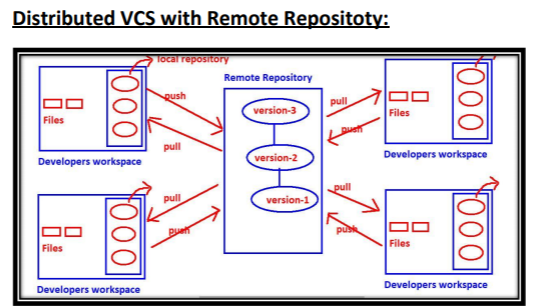
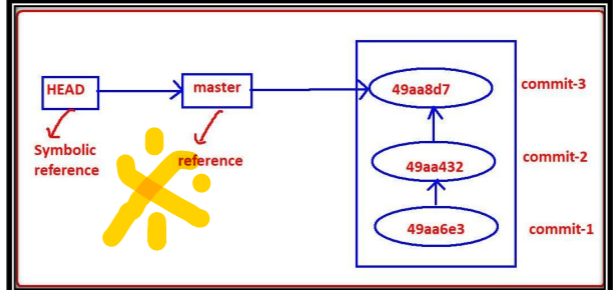
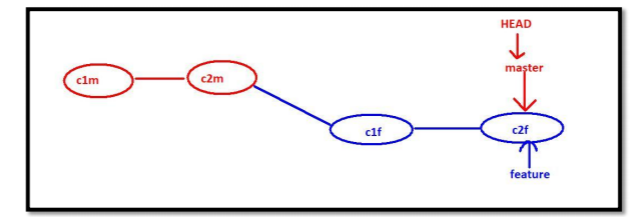
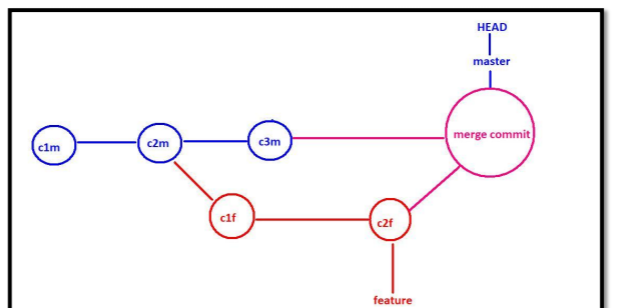
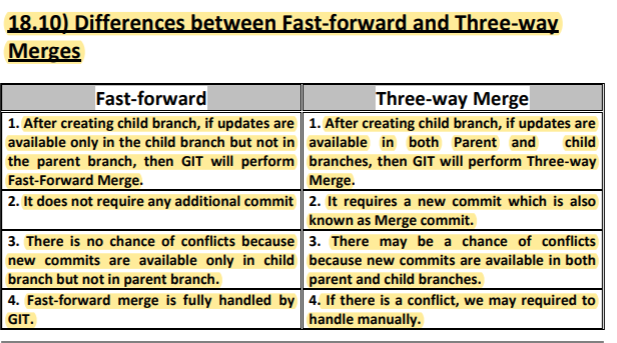
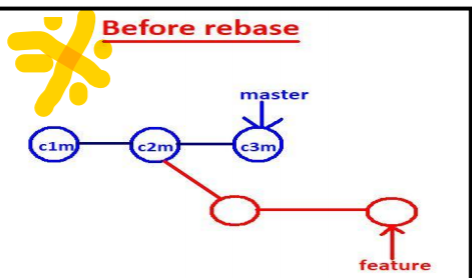
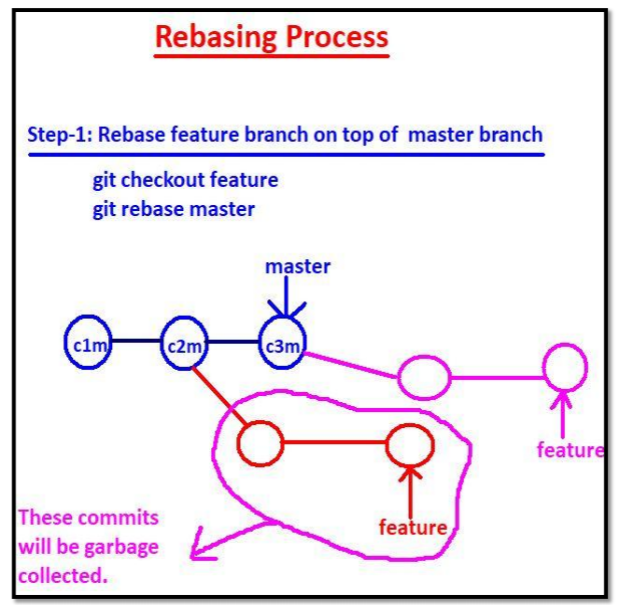
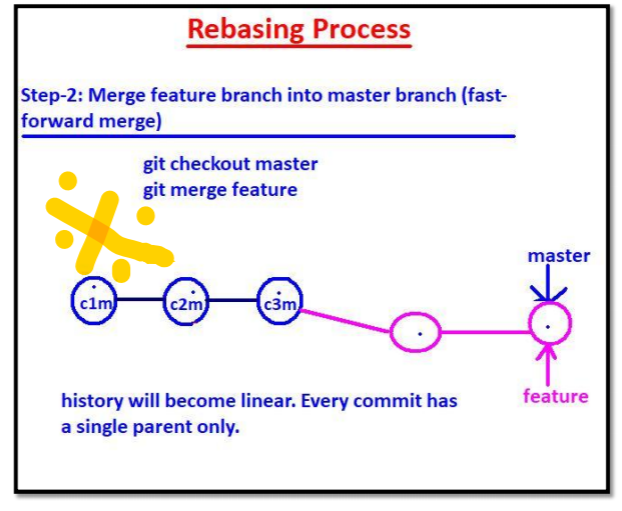
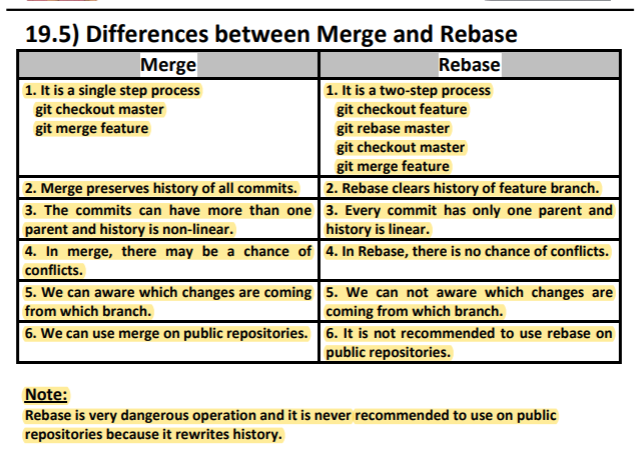
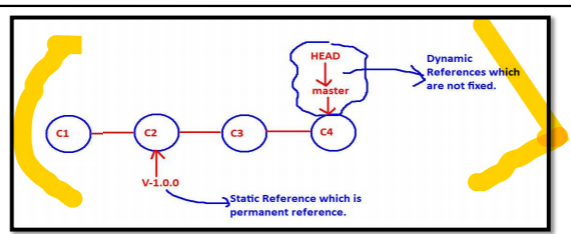
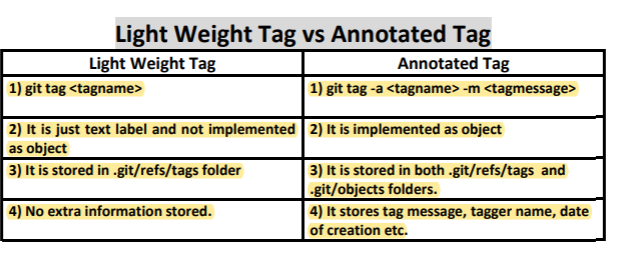
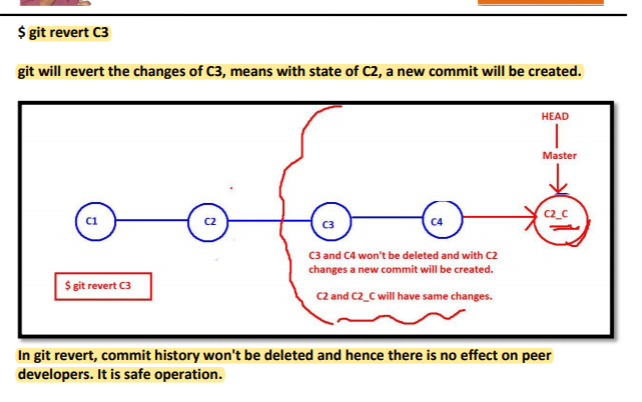
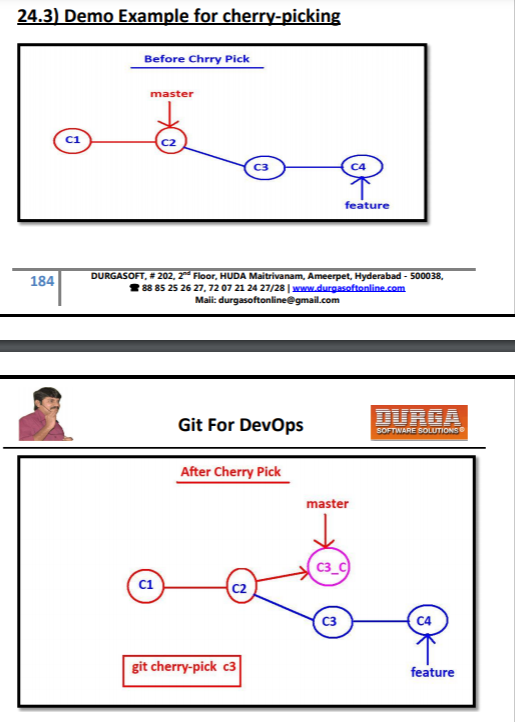
GIT Commands:

* Devops: • It is a new culture or process to develop,release and maintain software products/projects/applications with high quality in very faster way.
* Devops vs Agile Models: Devops and Agile, both are not same. Similarities: 1) Both are software development methodologies. Agile is there in the market for the last 20 years, but devops is recent methodology. 2) Both models concentrating on rapid development of software project. Differences: 1) The differences between these models will starts after development of the project. Agile methodology always talks about software development,testing and deployment. Once deployment completed agile methodology has no role. But Devops model will continue after deployment also and it is also responsible for operations and monitoring. 2) In Agile Model, separate people are responsible for developing, testing, and deploying the software. But, in DevOps, the DevOps engineer is responsible for everything; development to operations, and operations to development. 3) Agile model won't force us to use automation tools. But devops model is completely based on automation. 4) Agile model always giving highest priority for speed, where as Devops giving proirity for both speed and automation. 5) In Agile, client is responsible to give the feedback for the sprint. But in Devops, immediate feedback is available from the monitoring tools
* What is Devops? Devops is not a new Tool/Technology in the market. It is a new culture or process to develop,release and maintain software products/projects/applications with high quality in very faster way with automation tools. Devops is combination of development and operations. The main objective of devops is to implement collaboration between development and operations teams. It is the process of continuous development, continuous build, continuous test, continuous release of the software with high quality in very faster way with automation tools
* Types of Version Control Systems: There are 2 types of VCSs 1) Centralized Version Control System 2) De Centralized/Distributed Version Control System
* The name itself indicates that, this type contains only one central repository and every developer should be connected to that repository. The total project code will be stored in the central repository. If 4 developers are there, still we have only one repository.
* 
* Problems with Centralized VCSs: 1) Central Repository is the only place where everything is stored, which causes single point of failure. If something goes wrong to the central repository then recovery is very difficult. 2) All commit and checkout operations should be performed by connecting to the central repositoty via network. If network outage, then no version control to the developer. i.e in this type, developer work space and remote repository server should be connected always. 3) All commit and checkout operations should be performed by connecting to the central repositoty via network and hence these operations will become slow, which causes performance issues. No local operations and every version control operation should be remote operation. 4) Oranization of central repository is very complex if number of developers and files increases.
* 2.5.2) Distributed Version Control Systems: The name itself indicates the respository is distributed and every developers workspace contains a local copy of the repository. There is no question of central repository.
* Note: 1) commit and checkout operations will be performed between workspace and repository. work space – commit ◊ Repository Repository – checkout ◊ workspace 2) push and pull operations will be performed between repositories. one repository ---push ◊ other repository one repository ⇓ pull----other repository
* 
* Remote Repository is not Central Repository: 1) Every developer has his own local copy of repository. It is not centralized and it is distributed only. 2) commit and checkout operations won't be performed on remote repository and these will be performed on local repository only. The main job of remote repository is just to share our work to peer developers. High availability, Speed and there is no single point of failure are main reasons for popularity of this model. Eg: Git, Mercurial, Fossil
* GIT Architecture:
* 
* GIt contains 2 types of repositories: 1) Local Repository 2) Remote Repository For every developer, a separate local repository is available. Developer can perform all checkout and commit operations wrt local repository only. To perform commit operation, first he has to add files to staging area by using git add command, and then he has to commit those changes to the local repository by using git commit command. Hence commit in GIT is a 2-step process. commit is applicable only for staging area files but not for working directory files. If the developer wants to share his work to the peer developers then he has to push his local repository to the remote repository by using git push command. Remote repository contains total project code, which can be accessible by all developers. New developer can get local repository by cloning remote repository.For this we have to use git clone command. A developer can get updates from the remote repository to the local repository by using git pull command.
* git add -> To add files from working directory to staging area. git commit -> To commit changes from staging area to local repository. git push -> To move files from local repository to remote repository. git clone -> To create a new local repository from the remote repository. git pull -> To get updated files from remote repository to local repository.
* GIT Life Cycle: 1)Untracked: The files which are newly created in working directory and git does not aware of these files are said to be in untracked state. 2)Staged: ✽ The files which are added to staging area are said to be in staged state. ✽ These files are ready for commit. 3)In Repository/ Committed: Any file which is committed is said to be In Repository/Committed State. 4)Modified: Any file which is already tracked by git, but it is modified in working directory is said to be in Modified State.
* git init -> This command will provide empty repository for our working directory, so that version control is applicable for our workspace. The name of the empty directory is .git, which is hidden directory.
* Git Configurations before 1 st Commit: Before first commit, we have to configure user name and mail id, so that git can use this information in the commit records. We can perform these configurations with the following commands **git config --global user.email** **"durgasoftonline@gmail.com"** **git config --global user.name "Durga"**
* **git status:** It shows the current status of all files in each area, like which files are untracked, which are modified, which are staged etc. Note: We can get concise information by using -s option git status –s
* git add: To add files from working directory to staging area for tracking/commiting purpose, we have to use git add command
* i) To add all files present in current working directory git add . ii) To add one or more specified files git add a.txt git add a.txt b.txt iii) Even we can use pattern also git add \*.txt git add \*.java
* git commit: If we want to commit staged changes, then we have to use git commit command. For every commit, a unique commit id will be generated. It is of 40-length hexadecimal string.
* The first 7 characters also unique, by using that also we can identify commit.
* This unique id is considered as hash, which is generated based on content of files
* The advantages of this hash are 1) Data inside our local repository is more secure. 2) git requires less space to store contents of files.
* while using git commit command, commit message is mandatory. git commit -m "commit message"
* We can add files to staging area and we can commit changes by using a single command git commit -a -m "commit message", But this command will work only for tracked files but not for new files.
* git log: It shows history of all commits. It provides commit id, author name,maild , timestamp and commit message.
* git config: We can use this command to configure git like user name, mail id etc git config --global user.email "durgasoftonline@gmail.com" git config --global user.name "Durga"
* \*\*\*Note: global means these configurations are applicable for all repositories created by git. If we are not using global then it is applicable only for current repository.
* git config --list To list out all git configurations $ git config user.name To display user name
* git ls-files This command will list out all files which are tracked by git
* How to see Log Information of a Particular File: git log git log file1.txt
* Option-1: --oneline Option to get brief Log Information Bydefault git log command will provide detailed output. If we want concise output then we should go for --oneline option
* Option-2: -n Option to Limit the Number of commits to Display We can limit the number of commits in the git log command output. For this we have to use -n option. $ git log -n 2
* Option-3: --grep Option to search based on given Pattern in commit Message: We can search based on given pattern in commit message. git log --grep="pattern"
* Option-4: Show commits more recent than a specific Time --since= --after= Show commits more recent than a specific date. git log --since="5 minutes ago" git log --since="2020-05-17"
* Option-5: Show commits Older than a specific Time --until= --before= Show commits older than a specific date.
* git log --until="5 minutes ago" git log --before="2020-05-17" display all commits on or before 17th.
* Option-6: Show commits based on Author --author= git log --author=Ravi –oneline
* Git log –graph
* It is very common requirment to find differences between the content of a particular file or all files 1) Between working directory and staging area 2) Between working directory and last commit 3) Between staged area and last commit 4) Between working directory and a particular commit 5) Between staged area and a particular commit 6) Between two specified commits For this we required to use git diff command. diff means difference.
* ) git diff file1.txt To compare working directory copy with staged copy
* git diff HEAD file1.txt To compare working directory copy with last commit copy
* git diff --staged file1.txt git diff --cached file1.txt git diff --staged HEAD file1.txt git diff --cached HEAD file1.txt To compare staged copy with last commit copy
* git diff <commit\_id> file1.txt To compare working directory copy with the specified commit copy.
* git diff –staged <commit\_id> file1.txt To compare staged copy with the specified commit copy
* git diff <source\_commit\_id> <dest\_commit\_id> file1.txt To compare content in the file between two commits
* git diff master test It shows all differences between master branch and test branch
* git diff master origin/master It shows all differences between master branch in local repository and master branch in remote repository.
* We can use P4Merge tool for both comparison and merging purposes.
* Difftool Configurations: git config --global diff.tool p4merge
* git config --global difftool.p4merge.path "C:\Program Files\Perforce\p4merge.exe"
* git config --global difftool.prompt false
* Mergetool Configurations: git config --global merge.tool p4merge
* git config --global mergetool.p4merge.path "C:\Program Files\Perforce\p4merge.exe"
* git config --global mergetool.prompt false
* git config --global –list
* git rm file1.txt ◊ It will remove file from both working directory and staging area 2) git rm --cached file1.txt ◊ It will remove file only from staging area but not from working directory 3) rm file1.txt ◊It will remove file only from working directory but not from staging area.
* We can use checkout command to discard unstaged changes in the tracked files of working directory. Observe the 3 words: 1) Only for working directory 2) To discard unstaged changes(The changes which are not added to staging area) 3) In the tracked files (The files which are already added to staging area/commit) It is something like undo operation. It will copy contents of the file from index area(staging area) to working directory.
* Note: git checkout is applicable only for the files which are already tracked by git. It is not applicable for new files
* Summary: git checkout -- file.txt To discard changes in working directory copy. git checkout To discard changes in all tracked files of working directory. git checkout If we are not passing any argument, then this command will show the list of eligible files for checkout. Note: git checkout command can be used in branching also.
* For most of the commands (like git log, git diff etc) we have to provide commit id as argument. But remembering commit id is very difficult, even 7 characters also. Git provides some sample names for these commit ids. We can use these names directly. These are just pointers to commit ids. These sample names are called references or refs. References are stored in .git/refs directory as text files. There are multiple types of references like heads,tags and remotes.
* What is master 1) master is the name of the branch. 2) It is a reference(pointer) to last commit id. Hence where ever we required to use last commit id, simply we can use reference master. 3) This information is available in .git/refs/heads/master file
* What is HEAD? HEAD is a reference to master. If any reference pointing to another reference, such type of reference is called symbolic reference. Hence HEAD is symbolic reference. Bydefault HEAD is always pointing to branch(master).
* HEAD is stored in root of .git directory but not in .git/refs directory.
* ****
* Detached HEAD: Sometimes HEAD is not pointing to the branch name, such type of head is considered as Detached HEAD.
* There are 2 utilities of git reset command. Utility-1: To remove changes from staging area Utility-2: To undo commits at repository level
* Utility-1: To Remove Changes from staging Area We can use git reset to remove changes from staging area. Changes already added to staging area, but if we don't want to commit, then to remove such type of changes from staging area, then we should go for git reset. It will bring the changes from staging area back to working directory. It is opposite to git add command.
* git rm --cached vs git reset: git rm --cached file1.txt The file will be removed completely from staging area. git reset file1.txt The file won't be removed from staging area, but reset to previous state(one step back).
* Utility-2: To undo Commits at Repository Level We can also use reset to undo commits at repository level. Syntax: git reset Moves the HEAD to the specified commit, and all remaining recent commits will be removed. mode will decide whether these changes are going t0 remove from staging area and working directory or not.
* The allowed values for the mode are: --mixed --soft –hard
* To discard commit-3: git reset --mixed 86d0ca3 git reset --mixed HEAD~1 git reset HEAD~1
* Note: 1) It is not possible to remove random commits. 2) --mixed will work only on repository and staging area but not on working directory. 3) whenever we are using --mixed, we can revert the changes, because changes are available in working directory.
* 2)reset with --soft Option: It is exactly same as --mixed option, but changes are available in working directory as well as in staging area. It won't touch staging area and working directory. As changes already present in staging area, just we have to use commit to revert back.
* Use Cases: 1) If some files are missing in the last commit, then add those files and commit again. 2) We forgot to add defect number in commit message.
* reset with --hard: It is exactly same as --mixed except that Changes will be removed from everywhere (local repository,staging area,working directory) It is more dangerous command and it is destructive command. It is impossible to revert back and hence while using hard reset we have to take special care.
* --mixed vs --soft vs --hard 1. --mixed: changes will be discarded in local repo and staging area. It won't touch working directoy. Working tree won't be clean. But we can revert with git add . git commit 2. --soft Changes will be discarded only in local repository. It won't touch staging area and working directory. Working tree won't be clean. But we can revert with git commit 3. --hard Changes will be discarded everywhere. Working tree won't be clean. No way to revert
* Note: If the commits are confirmed to local repository and to discard those commits we can use reset command. But if the commits are confirmed to remote repository then not recommended to use reset command and we have to use revert command.
* Alias means nickname or short name or other alternative name. In Git we can create our own commands by using aliasing concept. This is something like alias command in Linux. If any git command is lengthy and repeatedly required, then for that command we can give our own convenient alias name and we can use that alias name every time
* Q1) Create alias Name 'one' to the following git Command? git log –oneline
* Test whether alias Name already used OR not? First we have to check whether the name 'one' is already used or not. $ git one git: 'one' is not a git command. See 'git --help'. We can use 'one' as alias name.
* Creating alias Name: We can create alias name by using git config command. Syntax: git config --global alias.aliasname "original command without git" Eg: git config --global alias.one "log --oneline"
* Note: After creating alias name, we can use either alias name or original name.
* Q2) Create alias Name 's' to the following git Command? git status
* Where these aliases will be stored? All alias names will be stored inside .gitconfig file. This file will be available in user's home directory.
* It is very common requirement that we are not required to store everything in the repository. We have to store only source code files like .java files etc. README.txt ◊ Not required to store log files ◊ Not required to store We can request git, not to consider a particular file or directory. We have to provide these files and directories information inside a special file .gitignore .gitignore File: We have to create this file in working directory
* nothing added to commit but untracked files present (use "git add" to track) .gitignore: # Don't track a.txt a.txt #Don't track all .txt files \*.txt #Don't track log files logs/ #Don't track any hidden file .\*
* No special treatement for directories. Git always consider only files but not directories. Git never give any importance for the directories. Whenever we are adding files from the directory, implicitly directory also will be added.
* git branch - It will show all branches in our local repository. - By default we have only one branch: master - master is the default name provided by GIT.
* Note: There is another way to check on which branch currently we are working, for this we have to use git status command.
* 2) How to Create a New Branch: We can create a new branch by using git branch command. Syntax: git branch brach\_name
* 3) How to Switch from one Branch to another Branch? We have to use git checkout command. We used git checkout command already to discard unstaged changes in working directory. git checkout brach\_name
* \*\*\*4) Short-cut Way to Create a New Branch and switch to that Branch: We have to use -b option with checkout command. git checkout -b new2branch
* Important Conclusions: 1. All branches are isolated to each other. The changes performed in master branch are not visible to the new branch and the changes performed in the new branch are not visible to the master branch
* 2. In GIT branching, logical duplication of files will be happend. For every branch, new directory won't be created.
* 3. In Git, if we switch from one branch to another branch just HEAD pointer will be moved, beyond that no other work will be happend. Hence implementing branching concept is very easy and very speed.
* Note: In GIT Branching, new directory won't be created and files won't be copied and just HEAD pointer will be changed. Hence to implement branching zero affort is required in GIT.
* 18.6) Advantages of Branching: 1. We can enable Parallel development. 2. We can work on multiple flows in isolated way. 3. We can organize source code in clean way. 4. Implementing new features will become easy 5. Bug fixing will become easy. 6. Testing new ideas or new technologies will become easy
* 18.8) What is Fast-forward Merge? After creating child branch, if we are not doing any new commits in the parent branch, then git will perform fast-forward merge. i.e updations(new commits) happened only in child branch but not in parent branch. In the fast-forward merge, git simply moves parent branch and points to the last commit of the child branch.
* ****
* If same file modified by both parent and child branches then conflicts will be raised. In fast-forward merge there is no chance of any conflicts, because updations happened only in child branch and we didn't touch parent branch
* Note: After creating child branch if parent branch also contains some new commits, then fast-forward merge won't be happend and Three-way merge will be happed.
* Three-way merge creates a new commit which is also known as merge commit. Parent branch will pointing to the newly created merge commit.
* ****
* ****
* Merge Conflicts and Resolution Process In the case of 3-way merge, if the same file updated by both Parent and child branches then may be a chance of merge conflict. If there is a conflict then GIT stops the merge process and provides conflict message. We have to resolve the conflict manually by editing the file. Git will markup both branches content in the file to resolve the conflict very easily. Once we completed editing of the file with required final content, then we have to add to the staging area followed by commit. With that merging process will be completed
* 18.12) How to Delete a Branch? Once we completed our work we can delete the branch. Deletion of the branch is optional. The main objective of deleting branch is to keep our repository clean. We can delete a branch by using git branch command with -d option. Syntax: $ git branch –d, $ git branch -d feature
* If we want to combine all commits of feature branch into a single commit and merge that commit to the master branch, then we should go for squash option. git merge --squash feature
* Rebase is alternative way to merge changes of two branches togther. rebase = re + base ◊ re arrange base
* Process of rebasing: It is a two step process. Step-1: We have to rebase feature branch on top of master branch. A. Checkout feature branch **git checkout feature** B. Rebase feature branch on top of master branch git rebase master Step-2: We have to merge feature branch into the master branch(fast-forwar merge will be happend) A. checkout master branch git checkout master B. Merge feature branch into master branch git merge feature
* ****
* Step-1: We have to rebase feature branch on top of master branch. git checkout feature git rebase master Whatever new commits are there in the feature branch will be duplicated by git. Here everything(like commit message,timestamp,author name and mail) is same except that commit ids will be changed. The base commit of the feature branch(duplicate copy) will be updated as last commit of parent branch(master branch).
* Step-2: We have to merge feature branch into the master branch (fast-forwar merge will be happend) git checkout master git merge feature The master branch pointer will be changed to last commit(duplicate copy) of the feature branch. In this case Fast-forward merge will be happend.
* ****
* ****
* 19.3) Advantages of rebasing: 1. Rebase keeps history linear. In 3-way merge, a commit can have multiple parents. But in Rebase every commit has a single parent only. Hence history will be liner. 2. Clear work flow (Linear) will there. Hence easy to understand for the developers. 3. Internally git performs Fast-forward merge and hence there is no chance of conflicts. 4. No extra commit like merge commit. 19.4) Disadvantages of rebasing: 1. It rewrites history. We cannot see history of commits what we did in feature branches 2. We does not aware which changes are coming from which branch.
* ****
* 20.1) What is git stash: The git stash command takes our uncommitted changes (both staged and unstaged), saves in some temporary location. After completing our urgent work, we can bring these stashed changes to our current working directory.
* Note: 1. Stashing concept is applicable only for tracked files but not for newly created files. 2. To perform stashing, atleast one commit must be completed. $ git stash You do not have the initial commit yet $ git stash No local changes to save
* 20.3) How to list all available stashes: lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashing (master) $ git stash list stash@{0}: WIP on master: 0323e16 2 files added
* 20.4) How to check the contents of stash: git show stash@{0}
* 20.5) How to perform unstash? We have to bring files from temporary location to our working directory. For this we have to perform unstash operation. We can perform unstashing in 2 ways: 1. by using git stash pop 2. by using git stash apply 1. by using git stash pop: git stash pop stash@{0} It will bring stashed changes from temporary location to working directory. The corresponding entry will be deleted. 2. by using git stash apply: git stash apply stash@{0} It will bring stashed changes from temporary location to working directory. But, the corresponding entry won't be deleted, so that we can use this stash in other branches to continue their work
* $ git stash list
* 20.6) Partial Stash: Assume we have multiple files, but we want stash only for some files. It is possible and this concept is called partial stash. We can perform partial stash by using the following command: $ git stash –p
* We can have any number of stashes. Based on our requirement, we can delete all stashes or a particular stash. git stash clear To delete all stashes git stash drop stashid To delete a particular stash
* 21.1) Need of Remote Repositories: Upto this we created multiple files and we did several changes. We used multiple git commands. All these operations happend in local repository. If we required to share our code to the peer developers/team members then the remote repository concept will come in the picture. As GIT is distributed version control system, every developer has his own local repository. Every developer can share his code to the peer developers/team members. By using push operation, developer can share his code the peer developer. By using pull operation, developer can get the code of peer developer.
* 1)git remote: We can use git remote command to configure remote repository to our local repository. $ git remote add <alias\_name> remote\_repository\_url Eg: $ git remote add origin https://github.com/durgadevops88/github\_project.git Here, origin is alias name of the remote\_repository\_url. By using this alias name only we can communicate with remote repository. Instead of origin we can use any name, but it is convention to use origin. By using git remote command we can also view remote repository information.
* $ git remote It just provides the alias names of remote repositories
* $ git remote -v It provides remote repository urls also.
* 2)git push: We can use git push command to send our changes from local repository to remote repository. ie to push our changes from local repo to remote repo. git push Eg: git push origin master
* Note: Whenever we are using remote repository the communication will be happend between local and remote repositories only.
* 3)git clone: cloning means creating exactly duplicate copy. Already there is a project on remote repository. Being a new team member, we may required complete remote repository into our local repository. For this purpose we have to use git clone command. i.e we can use git clone command to create local repository with remote repository files. All the files and commit history will be copied from remote repository into local repository. Syntax: git clone Eg: git clone https://github.com/durgadevops777/github\_project.git Now the project name will become repository project name. Note: Based on our requirement we can provide a new name for the project. git clone Eg: git clone https://github.com/durgadevops777/github\_project.git my\_project
* 4)git fetch Command: We can use get fetch command to check whether any updates available at remote repository or not. This command will retrieve only latest meta-data info from the remote repository. It won't download updates from remote repository into local repository. Syntax: $ git fetch origin
* git pull Command: We can use git pull command to download and merge updates from remote repository into local repository. git pull = fetch+merge Syntax: $ git pull <remote> <branch> $ git pull origin master
* How Developer A will get these updates from Remote repo? Developer has to use git fetch and git pull commands. git fetch ◊ To check whether updates are available or not git pull ◊ To download and merge those updates with local repo. We can use directly git pull without using git fetch.
* Note: 1. To perform push operation, local repository and remote repository should be in sync. ie the local repository should be upto date to the remote repository. If any updates in remote repo, git push is not allowed. It will ask us to pull first. 2. If any conflicts are there, we have to sit with peer developer to resolve the conflicts.
* Introduction to Tagging: In our repository there may be a chance of multiple commits. HEAD/Branch is always pointing to latest commit and its value keep on changing as new commits happend. Sometimes, we have to mark significant events or mile stones(like version numbering) with some label in our repository commits. We can do this labeling by using git tag command. Tag is nothing but a label or mark to a particular commit in our repository. Tag is static and permanent reference to a particular commit where as HEAD/Branch is a dynamic reference. In general Tags can be used for release verions
* ****
* There are two types of tags 1) Light Weight/Simple Tags 2) Annotated Tags [Tags with Information]
* Creation of a Light Weight Tag: We have to use git tag command. Syntax: $ git tag Eg: $ git tag V-1.0.0.0 V-1.0.0.0 tag acts as a permanent reference to the current latest commit. But based on our requirement we can define a tag for old commits also.
* The tags will be stored in our repository inside tags folder(.git/refs/tags). We can list available tags by using -l or --list option. $ git tab -l $ git tab --list We can use tag directly where ever commit id is required in our commands. Eg: git show For the specified commitid, if any tag is defined then we can use directly that tag instead of commit id. git show tag\_name
* 22.3) How to delete a Tag? We have to use git tag command with -d or --delete option. git tag –d
* Summary: git tag git tag --list git tag –delete
* Symantic Versioning: Almost all software products follow Symantic Versioning. V 1.2.12 Where 1 is considered as Major version. 2 is considered as Minor version. 12 is considered as Patch version. We can get more info on semantic versioning from the following url: https://semver.org/ Note: First Alpha Release and then Beta Release and then finally original major version release.
* Limitation of LightWeight Tags: LightWeight Tag is simply text reference to the specified commit. It won't maintain any information like tagger name, date of creation, message etc.
* If we want maintain such type of information, we should go for Annotated Tags.
* Annotated Tags (Tags with Information) Annotated Tag is exactly same as Light weight tag except that it maintains information like tagger name, date of creation, description etc. Annotated Tag internally maintained as object form in git repository. Annotated tags will be stored in .git/refs/tags folder and .git/objects folder. We can create annotated tag by using git tag command with -a option. $ git tag -a Here -a indicates that it is annotated tag. Now default editor will be opened for tag message. $ git tag -a –m
* For light weight tag(V-1.0.0), no extra information available. But for Annotated tag(V-1.1.0) extra information is avaialble.
* LightWeight tag is not implemented as object but Annotated tag implemented as object.
* Annotated tag information will be stored in .git/refs/tags folder and .git/objects folder
* $ cat .git/refs/tags/V-1.1.0 378a767274b8e2991b6fcfa9d987fe6ec25fd34b This is hash of tag object.
* Note: Extra information maintained by Annotated Tag when compared with Light Weight tag and hence it is recommended to use Annotated Tag.
* ****
* How to Tag a previous Commit? Sometimes we may forgot to tag a particular commit. It is possible to tag for previous commits. $ git tag -a –m
* How to update an existing Tag? Sometimes we tag for a wrong commit. It is possible to update the tag to the correct commit. We can do this by using the following 2 ways. 1) Delete the tag and Recreate tag with corresponding correct commit id. 2) By using -f or --force option to replace an existing tag without deleting. $ git tag -a -f –m
* Note: For the same commit we can define multiple tags also based on our requirement.
* How to Push Tags to remote Repository? Bydefault push command wont push tags to the remote repository. We have to push tags separately.
* Whenever we are using push command, only code pushed to the remote repository but not tags.
* How to push a Single Tag? git push origin <tag\_name>
* How to push all Tags? We have to use --tags option with git push command. $ git push origin master –tags
* How to delete a Tag from the Remote Repository? git push origin :V-1.0.0-beta
* Need of revert Command: In the last videos we covered already git reset command. $ git reset c2 After c2, all the next commits will be deleted
* Suppose if this code already pushed to the remote repository and already several people pulled this code and starts working on that code,git reset command may create big problem. git reset command deletes commit history and hence it is destructive command and not recommended to use on public repositories. To overcome this problem we have to go for git revert command. git revert command won't delete commit history. It reverts the required commit by creating a new commit. i.e it will undo a particular commit without deleting commit history
* ****
* Note: With git revert we can use -n option also. -n or --no-commit Usually the git revert command automatically creates some commits with commit log messages stating which commits were reverted. This flag applies the changes necessary to revert the named commits to your working tree and the index, but does not make the commits. In addition, when this option is used, your index does not have to match the HEAD commit. The revert is done against the beginning state of your index.
* 24.1) Need of Cherry-Picking: Assume that we have two branches are there like master and feature. If we merge feature branch to the master branch, then all commits and the corresponding changes will be added to master branch. Insted of all commits, we can pick an arbitraty commit of the feature branch and we can append that commit to the master branch. It is possible by using cherry-pick command
* 24.2) Use cases of cherry-pick: 1. Cherry-pick allows to share code between branches. 2. We can use for bug hot fixes. Assume that a developer has started working on a new feature. During that new feature development he identify a pre-existing bug. The developer creates an explicit commit patching this bug. This new patch commit can be cherry-picked directly to the master branch to fix the bug before it effects more users.
* ****
* git cherry-pick 2ca8bd2
* Note: 1. When every we are using cherry-pick command, a brand new commit object will be created but with same changes of the picked commit of feature branch. 2. Cherry-pick may cause duplicate commits and hence in most of the scenarios we can use merge operation instead of cherry-pick.
* . We can use -n or --no-commit option with cherry-pick, in that case new commit won't be created. This is helpful whenever we are doing cherry-picking multiple commits. With all these changes we can create a single commit. $ git checkout master $ git cherry-pick -n commitid1 $ git cherry-pick -n commitid2 $ git cherry-pick -n commitid3 $ git add .;git commit -m 'single commit for multiple cherry-pick options
* Topic-25: git reflog Command reflog ◊ means reference log We can use git reflog command to display all git operations what ever we performed on local repository. We can use git reflog command to know only local repository operations but not remote repository operations. Syntax: $ git reflog It will show all git operations performed on local repository. $ git reflog It will show all git operations performed on local repository related to that particular branch. Note: reflog will maintain 90 days git operations history bydefault. We can use reflog command to go to specific state of the repository. By mistake if we did any unwanted destruction operation( like git reset --hard), still we can recover by using git reflog command.
* git reflog –stat
* $ git reflog –oneline
* Note: Whatever options we can use with log command, all those options can be used for reflog also