Git has its roots in the development environment for the Linux kernel. In the early 2000s, the team working on the kernel began using a proprietary distributed source control system called BitKeeper which was a commercial product.  
Later on, Development began in early April of 2005.

**git config --global user.name "abhay khati"**

**git config --global user.email** [abhaykhati@gmail.com](mailto:abhaykhati@gmail.com)

**to see the list of options set in .gitconfig file use following command  
git config --global –list**

**to see the location of global git config file use**

**git config --global --list --show-origin or when you start gitbash terminal usually you can find the .gitconfig at below location it is the one which is used while setting with –global option for core editor, diff and merge tool custom configuration**

**C:\Users\<windows user who logged in>\.gitconfig**

**to add a remote repository use following**

**git remote add origin** [**https://github.com/abhaykhati/myRepoFromExistingSource.git**](https://github.com/abhaykhati/myRepoFromExistingSource.git)

**git remote –**v to check which git remote repo it is referring to

Removes a file from working tree and index means before commit no checkout direct delete

**git-rm**

**to initialize an empty git repository use**:

**git init** // please ensure the base directory is entered already into which you wish to initialize git repository.

To get help for git commands  
**git help or git help –a or git help –g**

**Git help <command name> or git help glossary**

**Git clone to clone an existing repository from git hub  
git add <filename> (provide space if adding multiple files or can use \*.\* wildcard)**

**Use "git rm --cached <file>..." to unstage any file which is added to staging in last step   
(git add)**

**Git status -- long (long status) or git status –s (short status) to check status of staged files  
git reset HEAD <file>..." to unstage**

**To use express commit option without using add command:**

**git commit –am “ msg”**

**git log –oneline to check commit log with only one line**

**To check commit log of a particular file use following**

**git log <filename> // give space in case of multiple files logs**

If you wish to see commit history between 2 commits then use following command

Step 1:

git log –oneline

Step 2: copy commit id for **since param** you wish to start the history search

Step 3: git log <**since param**>..(These 2 dots are compulsory) copy another commit id for till parameter

So the command syntax is git log **since Param..till Param  
note: while showing result of above command it excludes since param  
to see last few commits such as i.e. last 3 or n commits then**

**Git log -n 3 –oneline**

**How to see a current branch in git?**

**git branch   
how to see which are the branches are there in remote and current repo**

**git remote show origin  
to create a branch in git  
git branch <new branch name>   
to switch from master branch to custom branch**  
**git checkout <new branch name>**  
**To express commit means skipping git add command use following but it works only if the file is earlier committed to local repo i.e. commit area**

**git commit –am “message to commit”**

**Use of checkout command:**

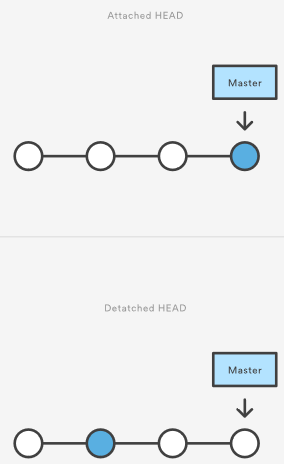
**Reverting changes in git repository for a particular file  
git revert head**

**to commit checkout,**

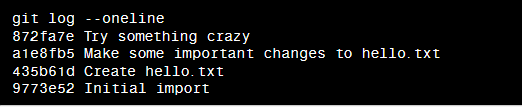
In case if you made changes to files and committed them, later on you want to revert your commit to a particular commit id which we get by using then It will put your head in detached state by using

**<git checkout selected commitId till which we want to revert>**in order to root out detached head state use **<git checkout master>  
git checkout -b <new-branch-name>** to put new changes of commit to another branch while in detached head state.   
  
The point is, your development should always take place on abranch—never on a detached HEAD. A fun metaphor is to think of Git as timeline management utility. Commits are snapshots of a point in time or points of interest along the timeline of a project's history. Additionally, multiple timelines can be managed through the use of branches. When 'undoing' in Git, you are usually moving back in time or to another timeline where mistakes didn't happen.

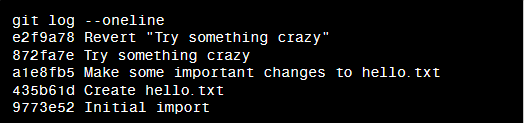
When you have found a commit reference to the point in history you want to visit, you can utilize the git checkout command to visit that commit. Git checkout is an easy way to “load” any of these saved snapshots onto your development machine. During the normal course of development, the HEAD usually points to master or some other local branch, **but when you check out a previous commit, HEAD no longer points to a branch—it points directly to a commit. This is called a “detached HEAD” state**.

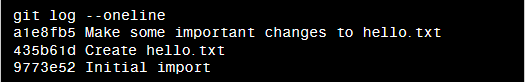
****

You can view all commits across all branches by executing **git log --branches=\***

**Undoing a committed snapshot with example  
**we will focus on undoing the **872fa7e** Try something crazy commit. Maybe things got a little too crazy. Checking out a specific commit will put the repo in a "detached HEAD" state. This means you are no longer working on any branch. In a detached state, any new commits you make will be orphaned when you change branches back to an established branch. Orphaned commits are up for deletion by Git's garbage collector. The garbage collector runs on a configured interval and permanently destroys orphaned commits. To prevent orphaned commits from being garbage collected, we need to ensure we are on a branch

**How to undo a public commit with git revert**

Let's assume we are back to our original commit history example. The history that includes the **872fa7e** commits. This time let's try a revert 'undo'. If we execute **git revert HEAD** Git will create a new commit with the inverse of the last commit. This adds a new commit to the current branch history and now makes it look like:  


At this point, we have again technically 'undone' the 872fa7e commit. Although 872fa7e still exists in the history, the new e2f9a78 commit is an inverse of the changes in 872fa7e. Unlike our previous checkout strategy, we can continue using the same branch. This solution is a satisfactory undo. This is the ideal 'undo' method for working with public shared repositories. **If you have requirements of keeping a curated and minimal Git history this strategy may not be satisfactory**.  
 **Undo a commit with git reset**git reset is an extensive command with multiple uses and functions. If we invoke **git reset --hard a1e8fb5** the commit history is reset to that specified commit. Examining the commit history with git log will now look like  
****  
the log output shows the e2f9a78 and 872fa7e commits no longer exist in the commit history. At this point, we can continue working and creating new commits as if the 'crazy' commits never happened. This method of undoing changes has the cleanest effect on history. **git reset moves both the HEAD and branch refs to the specified commit.** **Doing a reset is great for local changes however it adds complications when working with a shared remote repository. If we have a shared remote repository that has the 872fa7e commit pushed to it, and we try to git push a branch where we have reset the history, Git will catch this and throw an error. Git will assume that the branch being pushed is not up to date because of its missing commits. In these scenarios, git revert should be the preferred undo method.**   
**Undoing the last commit**   
in the previous section, we discussed different strategies for undoing commits. These strategies are all applicable to the most recent commit as well. **In some cases though, you might not need to remove or reset the last commit. Maybe it was just made prematurely. In this case you can amend the most recent commit. Once you have made more changes in the working directory and staged them for commit by using** [**git add**](https://www.atlassian.com/git/tutorials/saving-changes)**, you can execute git commit --amend. This will have Git open the configured system editor and let you modify the last commit message. The new changes will be added to the amended commit.**

**to checkout branch,   
git checkout <branchName>  
to checkout file to an old version of an individual file**

Let’s say if you have 5 commits of a particular file, then if you wish to root out last 3 commits of this particular file then use

**git checkout <commit Version> filename** it will set the pointer to 2nd commit version leaving behind the last 3 commits.

If you wish to keep that all the 5 commits of an above mentioned file and set the pointer to 5th commit which is the last commit of that file then use, this is what you called if you wish to revert or fallback to original commit head then use

**git checkout head <filename>**

If there is only single commit of a file, and if you want to revert it then use  
**git checkout filename**

**-------------------------------------------------------**

**Git Clean**

Git clean is to some extent an 'undo' command. The git clean command operates on untracked files. Untracked files are files that have been created within your repo's working directory but have not yet been added to the repository's tracking index using the git add command. By default, Git is globally configured to require that git clean be passed a "force" option to initiate. This is an important safety mechanism. When finally executed git clean is not undo-able. When fully executed, git clean will make a hard file system deletion, similar to executing the command line rm utility. Make sure you really want to delete the untracked files before you run it.  
  
**Git clean options:**

**-n** option will perform a “dry run” of git clean This will show you which files are going to be removed without actually removing them. It is a best practice to always first perform a dry run of git clean.  
**-f or –force** to remove files forcefully which is untracked by git  
**git clean –dn** Here we have executed a 'dry run' using the -dn combination which outputs untracked\_dir is up for removal  
**git clean –df** forceful deletion of untracked directory  
**git clean –di** "**interactive**" mode that you can initiate by passing the -i option

**Git Merge and Diff Tools**  
to see available editors for merge and diff use  
**git mergetool --tool-help**

To set p4merge as a tool for git diff **git config --global diff.tool p4merge #to set diff.tool as p4merge <p4merge is a name of the .exe>**

**git config --global diff.tool p4merge.path "C:/Program Files/Perforce/p4merge.exe" # to set the path for p4merge installation wihtout setting PATH Environment Variable in WINDOWS**

**git config --global difftool.prompt false # to disable prompt everytime the diftool is launched**  
  
to set core editor for git in order to edit messages and files   
**git config --global core.editor "'C:/Program Files (x86)/Meld/Meld.exe' -multiInst -nosession"**

**To set p4merge as a tool for git diff and git merge  
diff.tool=p4merge** #to set diff.tool as p4merge <p4merge is a name of the .exe>  
**difftool.p4merge.path=C:/Program Files/Perforce/p4merge.exe** # to set the path for p4merge installation and setting PATH Environment Variable in WINDOWS  
**difftool.prompt=false** # to disable prompt every time the difftool is launched  
  
**merge.tool=p4merge** #to set merge.tool as p4merge <p4merge is a name of the .exe>

**mergetool.p4merge.path=C:/Program Files/Perforce/p4merge.exe** # to set the path for p4merge installation and setting PATH Environment Variable in WINDOWS  
**mergetool.prompt=false** # to disable prompt every time the difftool is launched

**Git Diff commands:**

**Git diff <filename> or git diff** is used to show difference between working Directory and staging area  
**Git diff head** is used to see difference between working directory and last commit  
in order to see any difference via visual tool use  
**git difftool <filename> or git difftool** to see all difference between only one file **visually  
to see difference between working directory and staging area use**  
**git diff <filename> or git diff**   
or to see using visual editor  
**git difftool <filename with or without>**  
**To see the difference between working directory and commit area (Local Repository)  
git diff head <with /without filename>**or to see using visual editor  
**git difftool head <filename with or without>  
To see the difference between staging are and commit are (Local Repository)  
means the changes which is added to the staging area by using git add command but yet to be committed to the commit area (Local Repository).  
git diff --staged head or git diff –staged  
or  
git difftool --cached head or git difftool --cached   
To see the difference between 2 arbitrary (random) commits (preferably between last commit -1)  
git diff <commit id> head or git difftool <commit id> head use it for last commit -1  
git difftool <commit id of earliest commit> <commit id of latest> use it for arbitrary commit**

**In addition to the above command to select commit id for head-1 we can use caret symbol (^) or tilde (~)   
git diff head^ head or git difftool head^ head**

**git diff head~ head or git difftool head~ head  
in order to compare changes between last commit head -1 vs. working directory**

**Git difftool head~ (use either tilde~ or caret ^)**

**To see the difference of only few or one file instead of all files in commit use  
git diff or difftool -- <committed file name>  
to see the difference between local repository and remote repository  
git difftool master origin/master or git diff master origin/master**where **master** refers to local repo and **origin/master** refers to git remote repo **to see the difference between any 2 branches like master branch and feature branch   
git diff <master> <feature\_branch>**

