working-With-Git-Local

```
scenario 1: Create Empty Local repository & add some files, modify
files then add to git staging area & commit all changes together

    mkdir ProjectA ; cd ProjectA
    git init ( will initialize the empty project if the directory is
empty )
    touch abc.java new.java sample.java ( creates empty files if not
exists )
    vi sample.java (opens the file in editor - enter some text & save
it )
    git status -- to check changes made
    git add abc.java new.java sample.java ( adds the files to git
staging area so that git can track the changes )
    git commit -m "any message" ( saves the files to git repository
permanently  & genereates a commit id )

================================================================================
==========================================
scenario 2: in a git local repository add more files, edit some files,
remove somefiles & add to staging area then commit the changes for
every change accordingly

    cd ProjectA ( switch into the project direcotry in which git init
command was ran )
    touch heloworld.java readme.txt ( creates new empty files )
    git status -- to check changes made
    git add . ( we can give the individual file names OR just give dot
(.) to stage all the changes in the project )
    git commit -m "added files"
    vi heloworld.java ( opens the file in editor - enter some text &
save it )
    git status -- to check changes made
    git add .
    git commit -m "modified heloworld.java"
    rm abc.java ( removes the files from project ProjectA direcotry )
    git status -- to check changes made
    git add .
    git commit -m "deleted abc.java"

    Note: like above you can add & commit each change you do OR you do
all changes in a repository then at the end you can add & commit them
all together also, its completely how you want to do it.

================================================================================
==========================================
scenario 3: check all the commits made so far & who commited at what
time etc...
```

```
    cd ProjectA
    git log ( will show all the commits made to project so far )
    git log --oneline ( will show all the commits in short form )

================================================================================
========================================
scenario 4: it a git local repository remove somefiles permanently &
remove some files only from git repository ( means the file should
exist on machine but git should not track )

    cd ProjectA
    git rm readme.txt ( removes file from project & stages the changes
to git ( means we do not need to run git add for this change ) )
    ls -ltr -- to see file is removed
    git status -- to check changes made
    git commit -m "deleted readme.txt"

    git rm --cached heloworld.java -- removes the file only from git
repository, means the file is still present in the project but git
stops tracking the changes made to heloworld.java
    ls -ltr -- to see the file is still present
    git status -- to check changes made
    git commit -m "unstaged heloworld.java"

================================================================================
========================================
scenario 5: ignoring some files in a git repository permanently ( .
gitignore )

    cd ProjectA
    touch .gitignore ( creates a .gitignore file )
    vi .gitignore ( opens the file for editing enter below )
        logs/   ( completely ignore the direcotry & content inside )
        *.class ( completely ignores files ending with .class in the
reposiotry )
        *.txt   ( comletely ignores files ending with .txt in the
reposiotry )
        save & close the file
    git add .
    git commit -m "new .gitignore"
    git status -- to see any chagnes - make sure it shows "working tree
is clean" then

    touch abc.class new.txt naresh.class
    mkdir logs/
    cd logs
    touch devops.java git.java helo.txt
    cd ..
    git status -- should still show you working tree is clean as we
```

```
added all the above pattern in .gitignore file

        Note: .gitignore is a hidden file in linux ( means a dot (.)
infront of any file name becomes hidden file in linux )
        hidden files can not be listed with normal ls -ltr command --
to see hidden files do "ls -al"

================================================================================
==========================================

scenario 6: work with git tags

    Tagging is used to capture a point in history that is used for a
marked version release (i.e. v1.0.1), in other words git tags are
simply aliases for commit ids. tags are always created against specific
commit ids

    git tag -- lists all available tags
    git show v1.2 -- shows which commit id tagged, who commited & what
is commited

    git tag --a v1.2 -m "any message" --- by default git tags the
recent commit id / last commit id -- in other terms the HEAD

    git tag --a v1.3 <commit id> -m "any message" -- tags a specific
commit id provided

================================================================================
==========================================

scenario 7: reverting or resetting the changes made in a git reposiotry

    git reset --hard < commit id >
        resets to the commit id provided, means all commits after given
commit id are completely removed from git repository & also removes any
staged changes in the reposiotrory. so it recomended to check the git
status before & ensure the working tree is clean.
    git commit -m "Reverting to the state to <commit id>"
    git log -- verify the last commit id is now the commit id provided


    git revert < commit id >
        reverts to the commit id changes & provides a new commit id &
in git history log we will still have the deleted commit id for
reference
    git log -- verify the last commit id is a new commit id with
changes of commit id provided also we can see the commit id reverted
from.

================================================================================
```

```
==========================================
Working with git branching

scenario 8: create a new branch, switch to the branch make changes &
commit, delete a branch.

    git branch
        shows all the branches available ( * in front of branch name
refers the current branch we are working on )
        the default branch is called as "master" - it creates when we
do git init

    git branch <branch name>
        creates a new branch & copies content of current branch from
where we are creating it.
    git checkout -b BRANCH_NAME
        creates a new branch, copies content of current branch from
where we are creating it & switches to the newly created branch

    git branch
        to see the branch created successfully

    git checkout <branch name>
        to switch to the provided/existing branch
    git branch
        check whether the * mark is in front of the branch provided or
we switched to. which means our current working branch is now provided
branch

    git branch -d branch_name  -- to delete a branch
    git branch -D branch_name  -- to delete a branch forcefully

================================================================================
==========================================

scenario 9: create a branch,make changes, merge changes to master
branch.

    on current branch check the files ( ls -ltrh )
    git branch defect ( creates a new branch defect )
    git checkout defect ( switches to branch defect )
    git branch -- ensure/check the * is in front of defect branch,
which means the current working branch.
    make some changes in current branch
    touch abcd.java google.java facebook.java
    vi google.java ( enter some text & save it )
    git add .
    git commit -m "new java files"
    git checkout master ( switch to master branch )
    ls -ltrh the content of master branch ( changes made at defect
```

```
branch will not be available here )
    git branch -- ensure/check it switched to master branch
    git merge defect ( it merges the changes made in defect branch into
master branch )
    git status

================================================================================
=========================================

scenario 10: merge conflict ( occures when same file modified at both
branches )

    git branch defect
    git checkout defect
    git branch -- to see we checked out successfully to defect branch (
ensure * infront of defect )
    touch china.java  -- creates a empty file
    vi chian.java ( opens the file for editing, enter some text & save
the file )
    git add .
    git commit -m "new china.java at defect"

    git checkout master ( switch back to master branch )
    git branch -- to see we checked out successfully to master branch (
ensure * infront of master )
    touch china.java
    vi china.java ( opens for editing, enter some text & save it )
    git add .
    git commit -m "new china.java at master"

    now we have same china.java file exist & modifed at both master &
defect branches. this usually creates a merge conflict when we merge
defect branch with master branch

    root@ubuntu:/projectA# git merge defect
    Auto-merging china.java
    CONFLICT (add/add): Merge conflict in china.java
    Automatic merge failed; fix conflicts and then commit the result.

    root@ubuntu:/projectA# git status
    On branch master
    You have unmerged paths.
        (fix conflicts and run "git commit")
    Unmerged paths:
        (use "git add <file>..." to mark resolution)

        both added:      china.java

    In order to resolve the merge conflict, open the file for editing
at master ( vi china.java )
```

```
    root@ubuntu:/projectA# vi china.java
    <<<<<<< HEAD -- always delete this line
    at master
    ======= -- always delete this line
    at defect
    >>>>>>> defect -- always delete this line
```

    In above output we see the content on top put at master branch, below is put at defect branch. keep whatever is needed, it can be both changes or either of one. dicuss with the people who made those changes. we must always delete lines with (<<<< , >>> , ====)

    after decided what to keep, just save the file & close it. then

```
    git add .
    git commit -m "merge conflict resolved"
    git status -- working tree/directory should be clean
```

==========

scenario 11: understanding merge & rebase (merge & rebase both are used to combine the changes from one branch into another)

lets assume we have two branches master & feature with some commits we want to merge one into another

    assuming master has two commits c1 & c2, then we created a new branch feature, so here the feature is created with c1 & c2 and we call c2 as the base for feature branch now.

```
    master            c1   c2
    feature           c1   c2
```

    assume we started working on feature branch and create some commits f1 & f2 - mean while someone else also started working on master branch and created a commit c3. it will look like below now

```
    master            c1   c2  c3
    feature                c2  f1  f2
```

so what are the differnt ways to merge these two branches & what would be the results. lets try to understand

case1: merge changes with -- git merge

    git merge feature (we should be in master branch while running this command). once we issue this is how the git log history looks like at master

```
    master            c1   c2  c3  f1  f2  mc
    feature           c1   c2  f1  f2
```

    so this would add / forward all commit id from feature branch into master along with a brand new commit id called merge commit (mc above)
    this mc does not have any significance in history making some ambiguity those who look at the commit history.

case 2: merge changes with -- git merge --squash

    git merge --squash feature (we should be in master branch while running this command). once we issue this is how the git log history looks like at master

```
    master            c1   c2  c3  mc
    feature           c1   c2  f1  f2
```

    this case it will combile all the commit ids from feature branch & creates a brand new commit id at master with all changes combined from feature.

case 3: merge changes using -- git rebase

```
        master      c1  c2   c3
        feature         c2   f1   f2
```

        swith to feature branch and do "git rebase master", which would result as below

```
        master       c1  c2  c3
        feature      c1  c2  c3  f1  f2  ( it changes feature branch base to match last commit id of master )
```

        switch back to master (git checkout master) & do "git rebase /merge feature" , which would result as below.

```
        master       c1  c2  c3  f1  f2
        feature      c1  c2  c3  f1  f2
```

        making both branches with clean history of commits.

there are pros & cons using reabse -- we need to understand the situation before using & if neccessary use merge or rebase accordingly.