Git For DevOps Study Material

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

INDEX

Detailed Index

Option-1: --oneline option to get brief log information
Option-2: -n option to limit the number of commits to display
Option-3: --grep option to search based on given pattern in commit message
Option-4: Show commits more recent than a specific time.
Option-5: Show commits older than a specific time
Option-6: Show commits based on author
Option-7: --decorate option to display extra information

Topic-9: The Complete Story of git diff command
Case-1: To see the difference in file content between working directory and staging area
Case-2: To see the difference in file content between working directory and last commit
Case-3: To see the difference in file content between staged copy and last commit
Case-4: To see the difference in file content between specific commit and working directory copy
Case-5: To see the difference in file content between specific commit and staging area copy
Case-6: To see the difference in file content between two specified commits
Case-7: To see the difference in file content between last commit and last butone commit
Case-8: To see the differences in all files content between two specified commits
Case-9: To see the differences in content between two branches
Case-10: To see the differences in content between local and remote repositories

Topic-10: Helix Visual Merge Tool(p4merge) For Checking Differences

Topic-11: Removing files by using git rm command
Case-1: To remove files from working directory and staging area(git rm)
Case-2: To remove files Only from staging area (git rm --cached)
Case-3: To remove files Only from Working Directory (rm command)

Topic-12: Undo changes with git checkout command

Topic-13: Git References (master and HEAD)

Topic-14: Git reset command
Utility-1: To remove changes from staging area
Utility-2: To undo commits at repository level(--mixed, --soft, --hard modes)

Topic-15: Git Aliases - Providing our own convenient names to git commands
Topic-16: Ignoring unwanted files and directories by using .gitignore file
Topic-17: Any Special Treatment for directories by Git ???

Topic-21: Working with Remote Repositories

- 21.1) Need of Remote Repositories
- 21.2) How to work with GitHub?
- 21.3) How to create account in GitHub?
- 21.4) How to create a new reposioty in Github?
- 21.5) How to work with Remote Repository?
  - 1) git remote
  - 2) git push
  - 3) git clone
  - 4) git pull
  - 5) git fetch

Topic-22: Git Tagging (git Tag Command)

- 22.1) Introduction to Tagging
- 22.2) Creation of a Light Weight Tag
- 22.3) How to delete a tag?
- 22.4) Annotated Tags (Tags with Information)
- 22.5) How to tag a previous commit?
- 22.6) How to update an existing tag?
- 22.7) How to compare Tags
- 22.8) How to push tags to remote repository?

Topic-23: git revert Command

- 23.1) Need of revert command
- 23.2) Demo Example to revert last commit
- 23.3) Demo example to revert a particular commit

Topic-24: Cherry-Picking (git cherry-pick Command)

- 24.1) Need of Cherry-Picking
- 24.2) Use cases of cherry-pick
- 24.3) Demo Example for cherry-picking

Topic-25: git reflog Command

Topic - 1

Introduction to DevOps

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

Topic - 1: Introduction to DevOps

1.1) What is Devops?
1.2) Water Fall Model
1.3) Agile Model
1.4) Water fall vs Scrum
1.5) Devops vs Agile Models
1.6) Top Important points about DevOps

1.1) 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.
- We can achieve this in devops by using several automation tools.
- For any software development,release and maintenance, there are two groups of engineers will work in the company.

1) Development Group
2) Non-Development Group or Operations Group or Administrators Group.

Again this classification can be divided into small sets of groups.

1) Development Group:

The people who are involving

1) planning
2) coding
3) build
4) Testing

are considered as Development Group.

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

Eg:
Business Analyst(BA)
System Analyst(SA)
Design Architech(DA)
Developers/coders
Build Engineer
Test Engineers/QA

2) Operations Group:

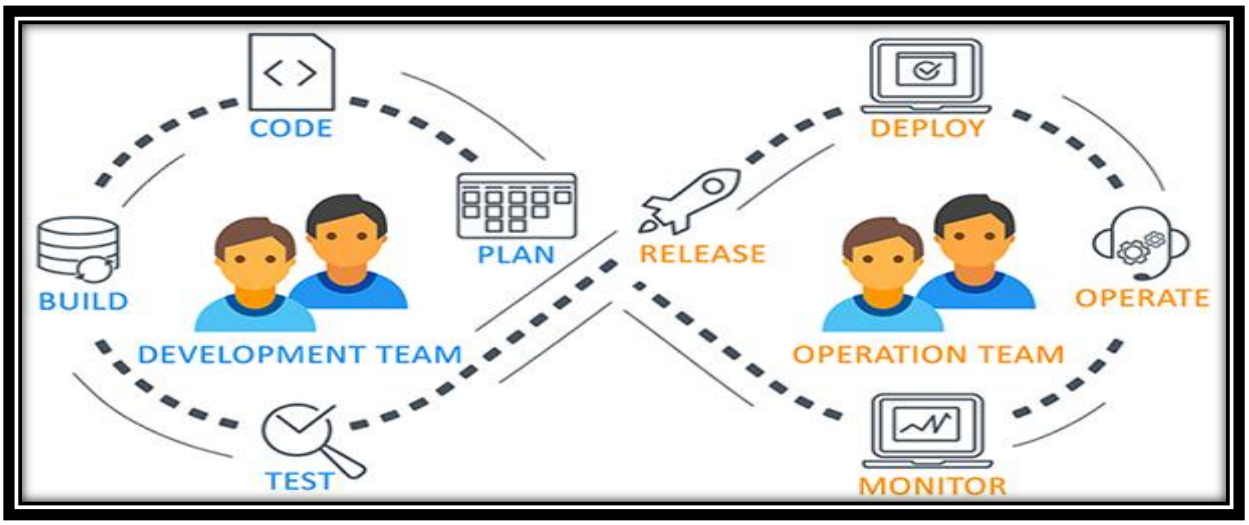
The people who are involving

1) **Release**
2) **Deploy**
3) **Operate**
4) **Monitor**

are considered as Operations Group.

Eg:
Release Engineers
Configuration Engineer
System Admin
Database Admin
Network Admin
etc

Devops is combination of development and operations.
The main objective of devops is to implement collaboration between development and operations teams.



DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

To understand this new Devops culture, we have to aware already existing SDLC Models.

SDLC → Software Development Life Cycle

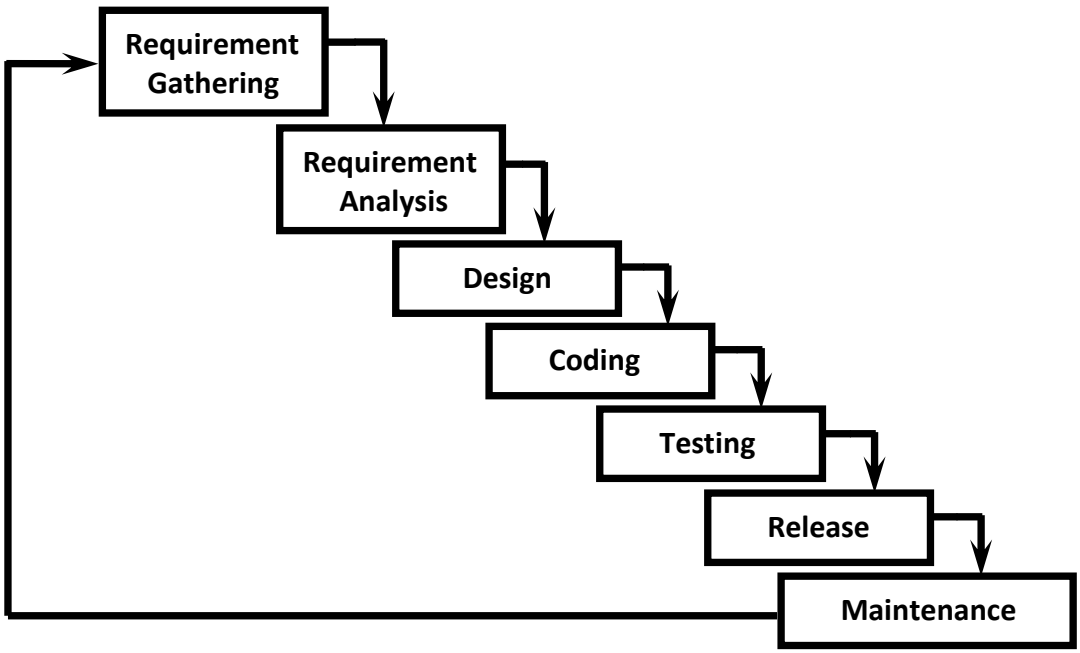
1) **Waterfall Model**
2) **Prototype Model**
3) **Incremental/Iterative Model**
4) **Spiral Model**
5) **RAD Model**
6) **Big-Bang Model**

7) **Fish Model**
8) **V Model**
9) **Agile Model**

10) **Devops Culture**

1.2) Water Fall Model:

- **It is the oldest SDLC Model.**
- **It is also known as Linear sequential Model.**
- **In this model,each phase must be completed before the next phase can begin and there is no overlapping of phases. i.e all phases will be performed one by one just like flowing of water fall downwards.**



DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

Advantages:

1) It is very simple and easy to implement.
2) Phases won't be overlapped and hence there is no ambiguity.
3) All phases will be executed one by one which gives high visibility to the project managers and clients about the progress of the project.
4) Best suitable if the requirements are fixed.
5) Best suitable for small projects.

Disadvantages:

1) It is very rigid model b'z it won't accept requirement changes in the middle.
2) Client satisfaction is very low because most of the times client will add new requirements in the middle,which won't be supported.
3) Total project development time is more because testing should be done after complementing development only.
4) The cost of bug fixing is very high because we cannot identify bugs in the early stages of life cycle.
5) Not suitable if the requirements keep on changing.
6) Not suitable for large projects.

1.3) Agile Model:

This is the most frequently used and hot cake model for software development.

Agile Model is divided into several sub models

1) Rational Unify Process (RUP)
2) Adaptive Software Development (ASD)
3) Feature Driven Development (FDD)
4) Crystal Clear
5) Dynamic Software Development Method (DSDM)
6) Extream Programming (XP)
7) Scrum

   etc

Among all these models Scrum model is the most popular and frequently used model.
Scrum is derived from Rugby Game.

- It is light weight process.
- It is an iterative /incremental model and it accepts changes very easily.
- It is people based model but not plan based model.
- Team Collaboration and Continuous feedback are strengths of this model.

1.4) Water fall vs Scrum:

1) In water fall model ,before starting next phase,the previous phase should be completed. It is very rigid model and won't accept requirement changes in the middle.
2) But scurm model is not linear sequential model. It is iterative model. Total software will be developed increment by increment and each increment is called a sprint. Sprint is a deliverable/shippable product in scrum model.

Points to Remember:

1) Scrum is an agile model that allows us focus on delivering highest quality software in shortest time.
2) In this model software developement follows increment by increment
3) Each increment will take one to 3 weeks duration.
4) 7 to 9 members are responsible in every sprint.

The art of doing the twice work in half time is nothing but scrum model → Juff sutherland

Advantages of Scrum Model:

1) There is maximum chance for quality
2) It ensures effective use of time and money
3) Requirement changes will be accepted so that maximum chance for client satisfaction
4) There is a possibility for the client involvment in every stage.

5) Project status Tracking is very easy
6) Team gets complete visibility through scrum meetings.

Limitations:

1) The chances of project failure is very high if individuals are not committed or cooperative
2) Adapting scrum model for large teams is very big challenge
3) Must required experienced and efficient team members
4) If any team member leaves in the middle of project,it can have a huge negative impact on the project.

1.5) 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:

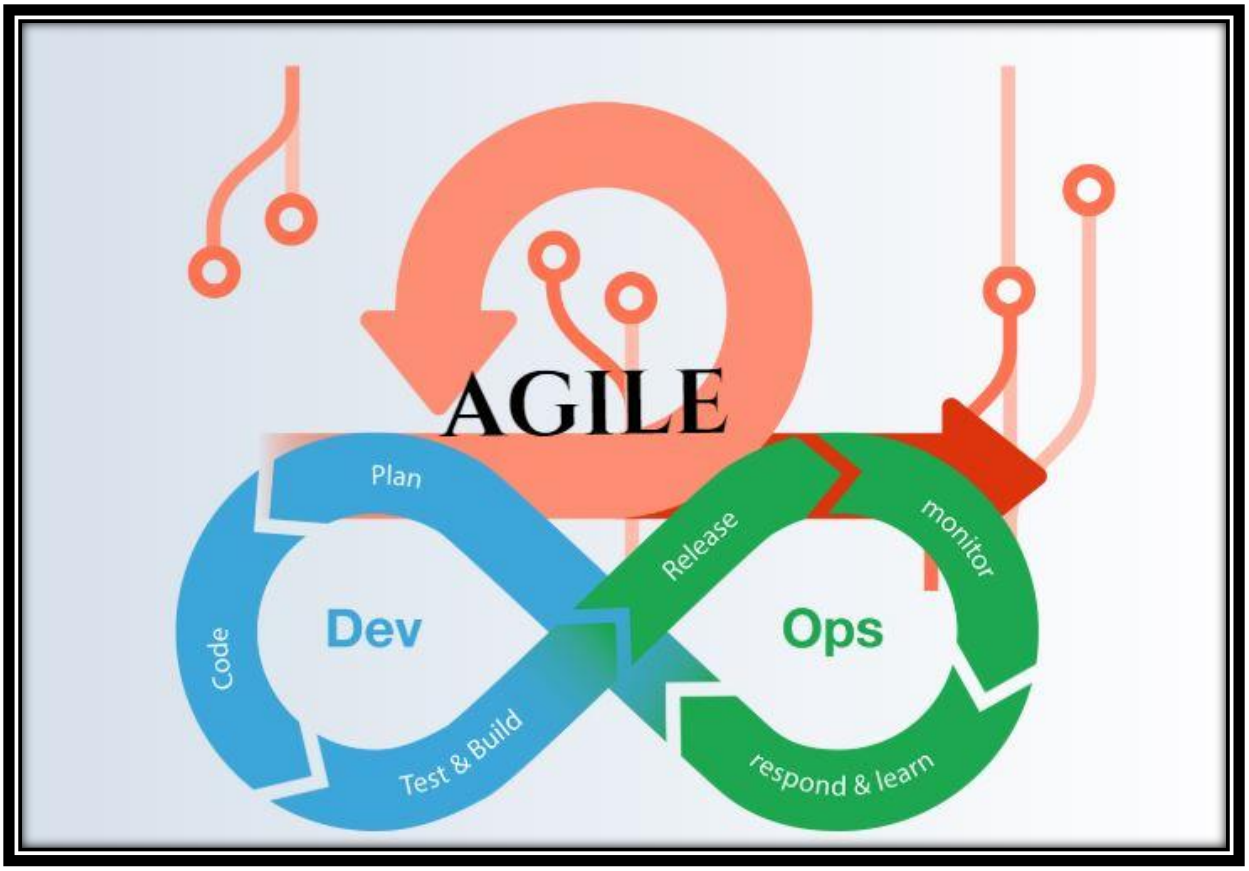
1) The differences between these models will starts after development of the project.
   Agile methodology always talks about software development,testing and deployment.
   Once deployment completed agile methodology has no role.
   But Devops model will continue after deployment also and it is also 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.

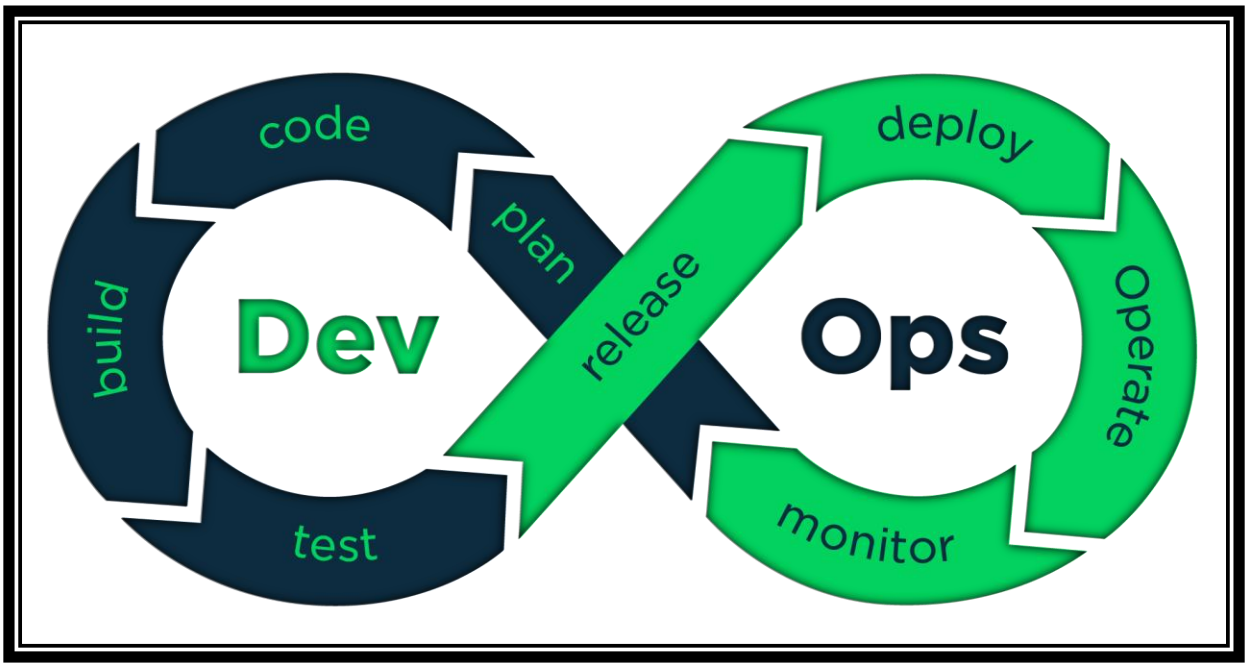
It is a new culture or process to develop,release and maintain software products/projects/applications with high quality in very faster way with automation tools.

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.

1.6) Top Important points about DevOps:

1) Devops is not a new Tool/Technology in the market.
2) It is a new culture or process to develop,release and maintain software products.
3) DevOps is combination of Development and Operations.
4) The main objective of devops is to implement collaboration between development and operations teams.
5) The beauty of DevOps is everything is automated and we can use several automation tools for development and operations.
6) Devops Engineer is All Rounder. He should aware everything. Hence his role is considered as Devops Generalist.
7) Devops is not Agile model and it is more than that because it covers both Development and operations, where as Agile covers only Development but not operations.
8) Devops Cycle is an Infinite Loop where everything is continuous.



TOPIC – 2

Introduction to Version Control System

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

Topic-2: Introduction to Version Control System

Version Control System is also known as Software Configuration Management (SCM) OR Source Code Management (SCM) System.

2.1) Need of Version Control System?

Being a developer we have to write several files which contains source code.

Developer → Write Code → Files

Client gave requirement to Durga to develop a project

client project
  |--100 files developed
  |- client suggested some changes
  |- I changed some files source code to meet client requirement
  |- I gave the demo and client suggested some more changes
  |- I changed some files source code to meet client requirement
  |- I gave demo 3rd time
  |- Client asked for first version only
  |- My Face with big ????

We should not overwrite our code.
Every version we have to maintain.

1) Maintaining multiple versions manually is very complex activity.
2) Dev-A and Dev-B working on the code. At last we have to merge the code developed by both developers and we have to deliver to the client. If both developers developed a file named with Util.java, then one copy will overwrite with another copy, which creates abnormal behaviour. We should not overwrite our code.

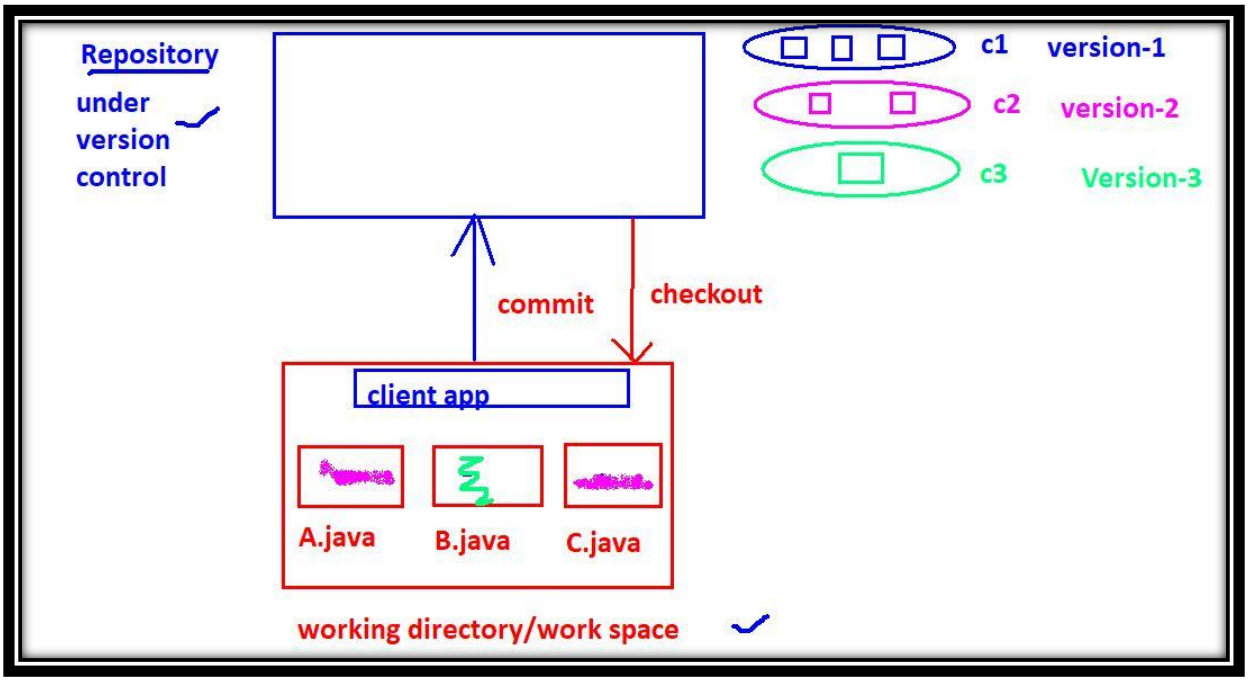
DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

2) Every change should be tracked like
   who did the change
   when he did the change
   which changes he did etc
   and all changes should be maintained.
3) Overwriting of the code should not be happend.
4) Developers have to share their code to peer developers, so that multiple developers will work in collaborative way.
5) Parallel development must be required

2.2) How Version Control System will work?

Version control system always talks about files which contain source code.
Everyone required version control system to maintain different versions of their document.

tester → To maintain different versions of test script
Architect → To maintain different versions of Documents
Project Manager → To maintain different versions of Excel sheets
etc



DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

2.3) The Basic Terminology of Version Control System:

Working Directory:

Where developers are required to create/modify files.
Here version control is not applicable. Here we won't use the work like version-1, version-2 etc

Repository:

Where we have to store files and metadata.
Here version control is applicable.
Here we can talk about versions like version-1, version-2 etc

Commit:

The process of sending files from working directory to the repository.

Checkout:

The process of sending files from repository to working directory.

2.4) Benefits of Version Control System:

1) **We can maintain different versions and we can choose any version based on client requirement.**
2) **With every version/commit we can maintain metadata like**
   - **commit message**
   - **who did changes**
   - **when he did the change**
   - **what changes he did**
3) **Developers can share the code to the peer developers in very easy way.**
4) **Multiple developers can work in collaborative way**
5) **Parallel development.**
6) **We can provide access control like**
   - **who can read code**
   - **who can modify code**

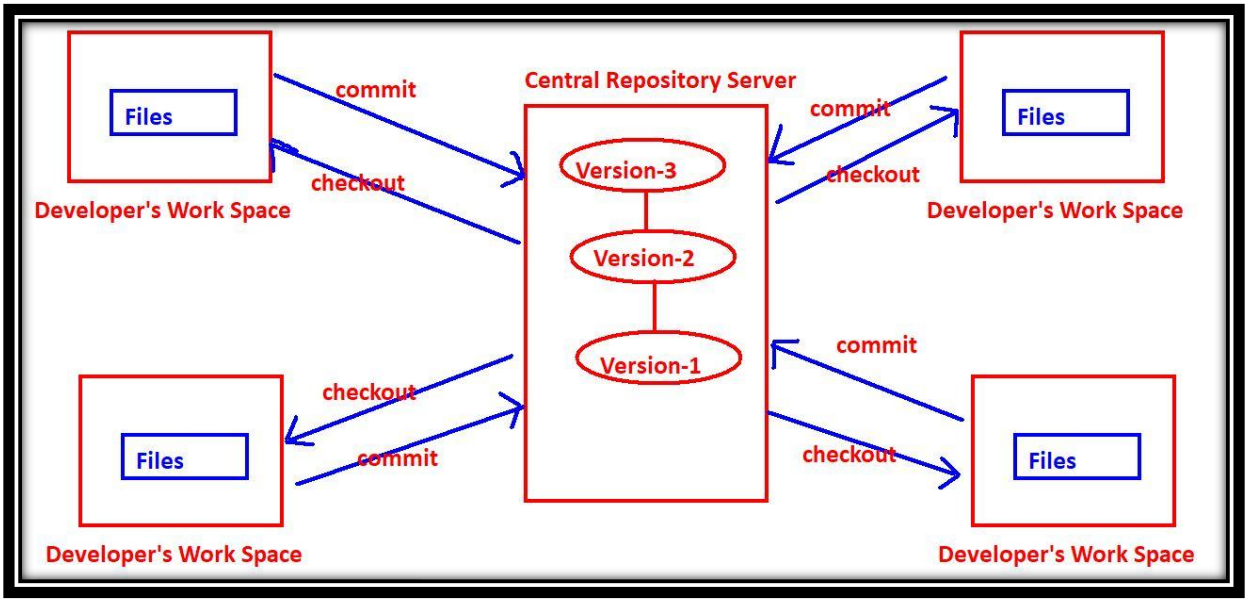
2.5) Types of Version Control Systems:

There are 2 types of VCSs

1) **Centralized Version Control System**
2) **De Centralized/Distributed Version Control System**

2.5.1) Centralized 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.



This type of VCS is very easy to setup and use.

Eg: CVS, SVN, Perforce, TFS, Clearcase etc

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.**

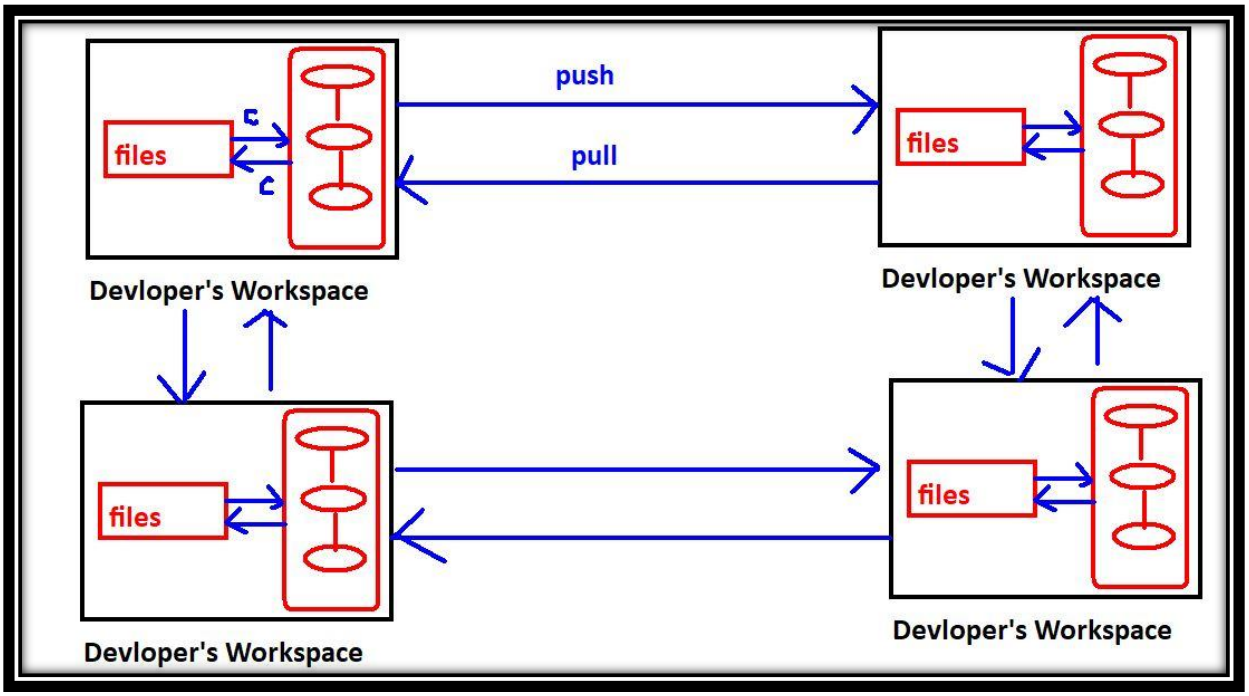
2) **All commit and checkout operations should be performed by connecting to the central repositoty via network. If network outage, then no version control to the developer. i.e in this type, developer work space and remote repository server should be connected always.**

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.**
   etc

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.



If 4 developers are there then 4 repositories will be there.

1) **The checkout and commit operations will be performed locally. Hence performance is more.**
2) **To perform checkout and commit operations network is not required. Hence if there is any network outage, still version control is applicable.**
3) **If something goes wrong to any repository there is a chance to recover. There is no question of single point of failure.**
4) **To perform push and pull operations network must be required, but these operations are not most common operations and we are performing very rarely.**

Note:

1) **commit and checkout operations will be performed between workspace and repository.**

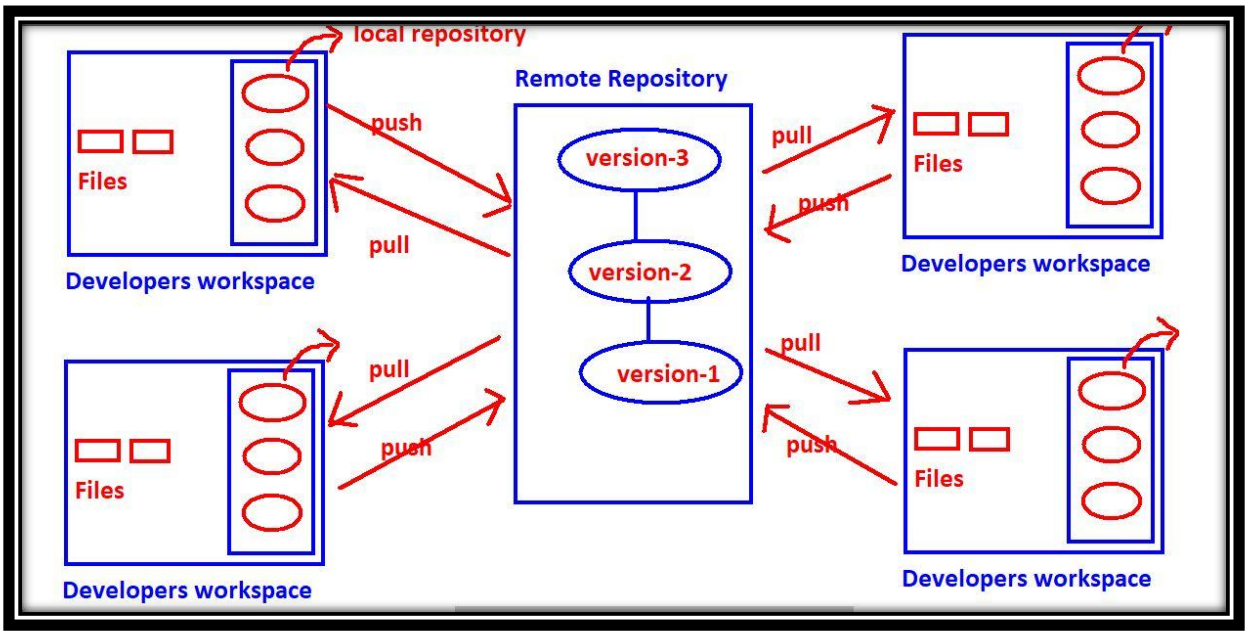
   work space – commit → Repository
   Repository – checkout → workspace

2) **push and pull operations will be performed between repositories.**

   one repository ---push → other repository
   one repository ← pull----other repository

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

Distributed VCS with Remote Repositoty:



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

TOPIC – 3

Features And Architecture of GIT

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

Topic-3: Features and Architecture of GIT

3.1) What is GIT?
3.2) Features of GIT
3.3) GIT Architecture

3.1) What is GIT?:

- Git is Distributed Version Control System Tool.
- Git is not acronym and hence no expansion.But most of the people abbreviated as
- "Global Information Tracker".
- GIT is developed by Linus Torvalds(Finnish software engineer), who also developed Linux Kenel.
- Most of the companies like Microsoft,Facebook,Yahoo,LinkedIn,Intel using Git as Version Control System Tool.

3.2) Features of GIT:

Git is very popular because of the following features:

1) Distributed

Git is developed based on Distributed Version Control System Architecture.
Because of Distributed Architecture it has several advantages:

A) Every Developer has his own local repository. All the operations can be performed locally.Hence local repo and remote repo need not be connected always.

B) All operations will be performed locally, and hence peformance is high when compared with other VCSs. i.e it is very speed

C) Most of operations are local. Hence we can work offline most of the times.

D) There is no single point failure as Every Developer has his own local repository.

E) It enables parellel development & automatic-backups.

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

2) Staging Area:

It is also known as index area.
There is logical layer/virtual layer in git between working directory and local repository.

Working Directory → Staging Area → Local Repository

We cannot commit the files of working directory directly. First we have to add to the staging area and then we have to commit.

This staging area is helpful to double check/cross-check our changes before commit.

This type of layer is not available in other Version Control Systsem Tools like CVS, SVN etc

Git stores files in repository in some hash form, which saves space.
GIT will uses internally snapshot mechanism for this. All these conversions and taking snapshots of our data will be happened in staging area before commit.

Eg: If a sample repository takes around 12 GB space in SVN where as in GIT it takes hardly 420 MB.

Hence Staging Area is the most important Strength of GIT.

3) Branching and Merging:

We can create and work on multiple branches simultaneously and all these are branches are isolated from each other. It enables multiple work flows.

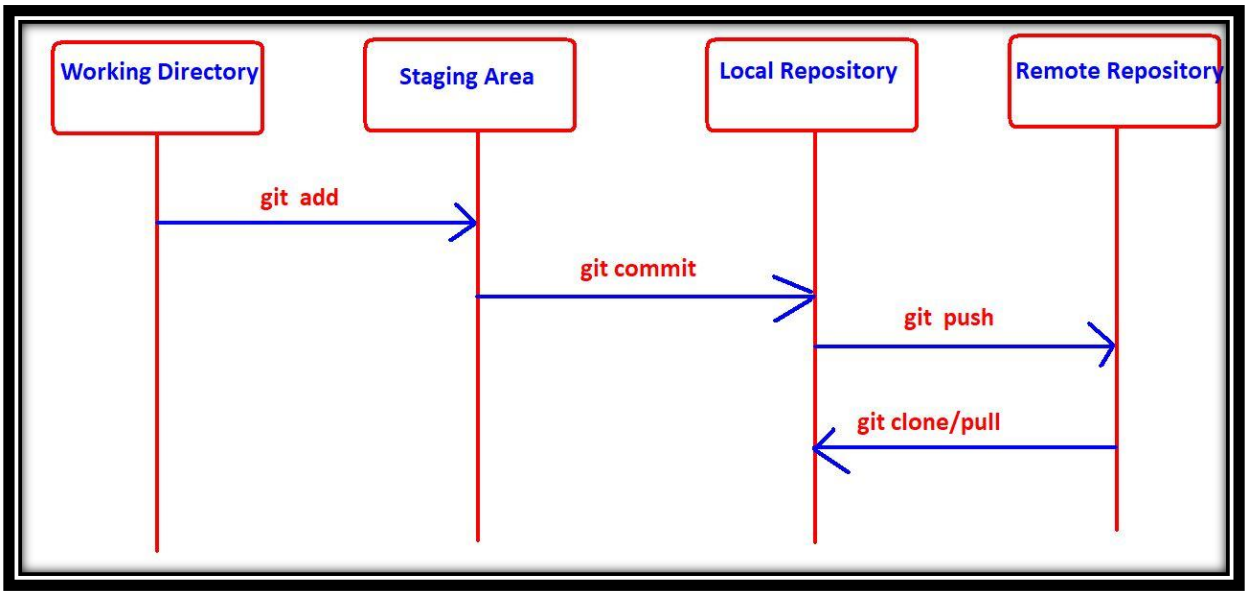
We can merge multiple braches into a single brach. We can commit branch wise also.

4. Moving files in GIT is very easy as GIT automatically tracks the moves. Whereas in other VCS we need to create a new file & then delete the old one.

5. Freeware and Open Source

6. It provides support for multiple platforms.

3.3) 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.

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

TOPIC – 4

Life Cycle of File in GIT

Topic - 4: Life Cycle of File in GIT

Every file in GIT is in one of the following states:

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:

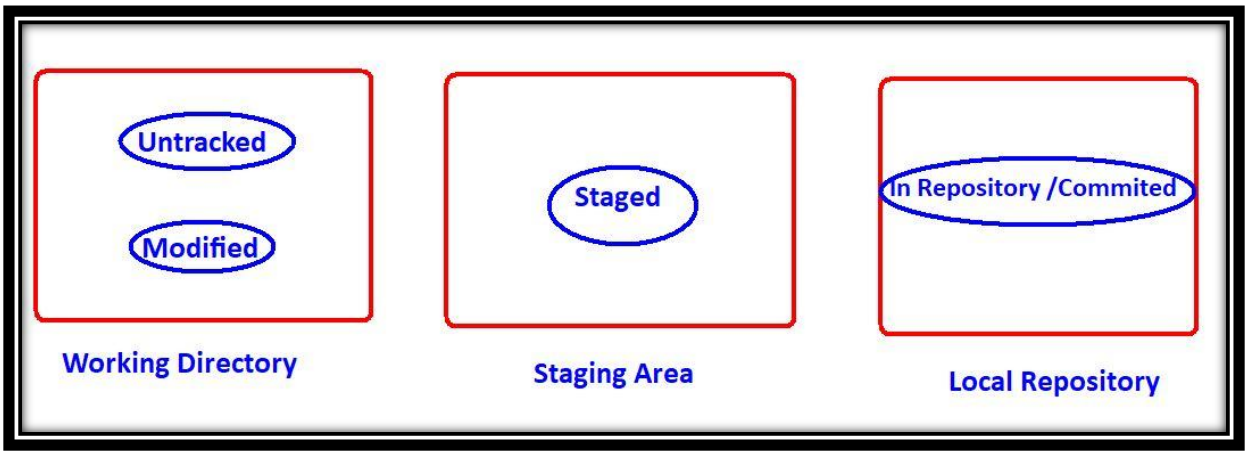
- **The files which are added to staging area are said to be in staged state.**
- **These files are ready for commit.**

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.



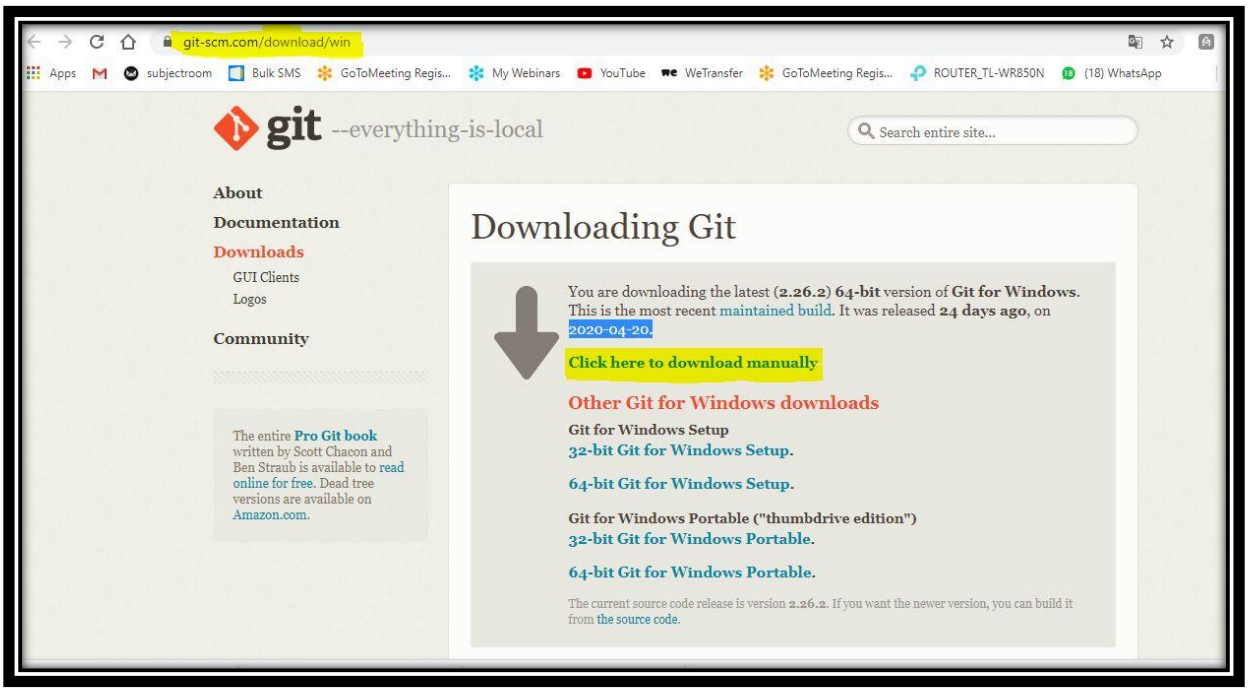
DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

TOPIC - 5

Git Installation On Windows

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

Topic-5: Git Installation On Windows



https://git-scm.com/download/win
2.26.2
Git-2.26.2-64-bit.exe

Q) How to check git installed OR not?

$ git --version
git version 2.26.2.windows.1

If we just type git, then we will get complete options available with git command.

```
lenovo@DESKTOP-ECE8V3R MINGW64 ~
$ git
usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]
```

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
- clone — Clone a repository into a new directory
- init — Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)
- add — Add file contents to the index
- mv — Move or rename a file, a directory, or a symlink
- restore — Restore working tree files
- rm — Remove files from the working tree and from the index
- sparse-checkout — Initialize and modify the sparse-checkout

examine the history and state (see also: git help revisions)
- bisect — Use binary search to find the commit that introduced a bug
- diff — Show changes between commits, commit and working tree, etc
- grep — Print lines matching a pattern
- log — Show commit logs
- show — Show various types of objects
- status — Show the working tree status

grow, mark and tweak your common history
- branch — List, create, or delete branches
- commit — Record changes to the repository
- merge — Join two or more development histories together
- rebase — Reapply commits on top of another base tip
- reset — Reset current HEAD to the specified state
- switch — Switch branches
- tag — Create, list, delete or verify a tag object signed with GPG

collaborate (see also: git help workflows)
- fetch — Download objects and refs from another repository
- pull — Fetch from and integrate with another repository or a local branch
- push — Update remote refs along with associated objects

'git help -a' and 'git help -g' list available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.
See 'git help git' for an overview of the system.

TOPIC – 6

Example-1 To Understand Working Directory, Staging Area and Local Repository

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

Topic-6: Example-1 To Understand Working Directory, Staging Area and Local Repository

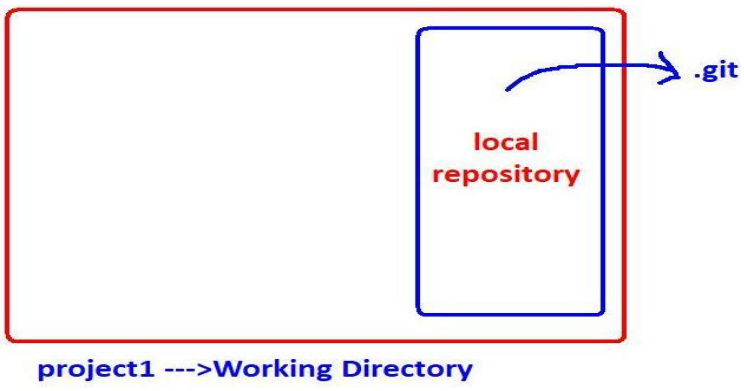
1) Creating workspace
2) git initialization
3) Creating files with some content in the working directory
4) Adding these files to staging area
5) Git Configurations before first commit
6) Commit those changes to local repository

```
lenovo@DESKTOP-ECE8V3R MINGW64 ~
$ cd d:
lenovo@DESKTOP-ECE8V3R MINGW64 /d
$ mkdir gitprojects
lenovo@DESKTOP-ECE8V3R MINGW64 /d
$ cd gitprojects
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects
$ mkdir project1
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects
$ cd project1
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project1
```

Now project1 acts as working directory. We have to request git, to provide version control for this directory. For this we have to use git init command.

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.

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project1
$ git init
Initialized empty Git repository in D:/gitprojects/project1/.git/
```



DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

Note:

1) If our working directory contains any files, then these files won't be added to the local repository bydefault, we have to add explicitly.
2) If our working directory already contains local repository(.git), still if we call git init command, then there is no impact.

Creating Files with some Content and adding to staging Area and then commit:

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project1 (master)
$ git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project1 (master)
$ cat > a.txt
First line in a.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project1 (master)
$ cat > b.txt
First line in b.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project1 (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        a.txt
        b.txt

nothing added to commit but untracked files present (use "git add" to track)

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project1 (master)
$ ls
a.txt  b.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project1 (master)
$ git ls-files
```

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project1 (master)
$ git add a.txt b.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project1 (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   a.txt
        new file:   b.txt
```

Git Configurations before 1st 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"

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project1 (master)
$ git commit -m "Added two files a.txt and b.txt"
[master (root-commit) 9a33a5b] Added two files a.txt and b.txt
 2 files changed, 2 insertions(+)
 create mode 100644 a.txt
 create mode 100644 b.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project1 (master)
$ git status
On branch master
nothing to commit, working tree clean

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project1 (master)
$ git log
commit 9a33a5b2e0d1c90eff544a3710b599be3c22665e (HEAD -> master)
Author: Durga <durgaadvjava@gmail.com>
Date:   Thu May 14 22:16:59 2020 +0530

    Added two files a.txt and b.txt
```

If we modify the Content in working Directory:

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project1 (master)
$ cat >> a.txt
Second Line

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project1 (master)
$ cat >> b.txt
Second Line

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project1 (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   a.txt
        modified:   b.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Adding these modified Files to the staging Area and then commit:

```
git add a.txt b.txt
git commit -m "Both files modified"

We can combined these two commands into a single command
git commit -a -m "Both files modified"

But make sure this option will work only for modified files, but not for newly created files.

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project1 (master)
$ git commit -a -m "Both files modified"
[master df4bb05] Both files modified
 2 files changed, 2 insertions(+)

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project1 (master)
$ git log
commit df4bb05e36e672698251e05e09d92ba45ea1fc47 (HEAD -> master)
Author: Durga <durgaadvjava@gmail.com>
Date:   Thu May 14 22:31:17 2020 +0530

    Both files modified
```

commit 9a33a5b2e0d1c90eff544a3710b599be3c22665e
Author: Durga <durgaadvjava@gmail.com>
Date: Thu May 14 22:16:59 2020 +0530

Added two files a.txt and b.txt

TOPIC – 7

The 6 Git Commands With Example

1) init
2) status
3) add
4) commit
5) log
6) config

Topic-7: The 6 Git Commands With Example - init, status, add, commit, log and config

1) git init

Once we creates workspace, if we want version control, then we require a local repository. To create that local repository we have to use git init command.

```
$ git init
Initialized empty Git repository in D:/gitprojects/project1/.git/
```

.git is an empty repository, which is hidden directory.

2) 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.

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   a.txt
        modified:   b.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Note: **We can get concise information by using -s option.**

```
$ git status -s
M  a.txt
M  b.txt
A  c.txt
```

3) 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
```

4) 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.

```
$ echo -n "df4bb05e36e672698251e05e09d92ba45ea1fc47" | wc -c
40
```

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.**
   (If SVN repository required 12GB, but for same content git requires 420MB)

while using git commit command, commit message is mandatory.

```
git commit -m "commit message"
```

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project1 (master)
$ git log
commit 9a33a5b2e0d1c90eff544a3710b599be3c22665e
Author: Durga <durgasoftonline@gmail.com>
Date:   Thu May 14 22:16:59 2020 +0530

    Added two files a.txt and b.txt
```

For every commit, git records author name,mail id, timestamp and 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"

-a means adding files to staging area
-m means commit message

But this command will work only for tracked files but not for new files.

git commit -a -m "commit message"
git commit -am "commit message"→ Valid
git commit -ma "commit message"→ won't work, because order is important.

5) git log:

It shows history of all commits.
It provides commit id, author name,maild , timestamp and commit message.

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project1 (master)
$ git log
commit 9a33a5b2e0d1c90eff544a3710b599be3c22665e
Author: Durga <durgasoftonline@gmail.com>
Date:   Thu May 14 22:16:59 2020 +0530

    Added two files a.txt and b.txt
```

There are multiple options available with git log command.
git log --help

6) 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 config user.email
To display user email

We can change user name and mail id with the same commands

git config --global user.email "durgasoftonline@gmail.com"
git config --global user.name "Durga"

7) $git ls-files

This command will listout all files which are tracked by git.

8) $ls

This command will listout all files present in workspace

Q) What is create mode 100644 ?

The first 3 digits describe the type of file.
The next 3 digits describe the file permissions.
100 → Means it is an ascii text file.
644 → File permissions (rw-r--r--)

TOPIC - 8

The Complete Postmortem of git log Command

Topic-8: The Complete Postmortem of git log Command

8.1) How to see History of all commits in Local Repository:

If we want to see history of all commits in local repository, then we have to use git log command. It is the most commonly used command in git.

git log and git log → Both are same

```
$ git log
commit 48437a7ad2ada6e18a26b127ca101c0ebf45b19e (HEAD -> master)
Author: Durga <durgaadvjava@gmail.com>
Date:   Thu May 7 21:09:33 2020 +0530

    This is 3rd commit for file2.txt

commit 3a8051f59110f9696f4e0f922f438cbb6bb7694d
Author: Durga <durgaadvjava@gmail.com>
Date:   Thu May 7 21:06:58 2020 +0530

    Second commit for file2.txt

commit 4b77312160c82d76395558da415a96b2a8b36072
Author: Durga <durgaadvjava@gmail.com>
Date:   Thu May 7 21:05:25 2020 +0530

    This is second commit related to file1.txt

commit 93d297b69e048046b8ff5dba140b5889f1b47500
Author: Durga <durgaadvjava@gmail.com>
Date:   Thu May 7 20:59:22 2020 +0530

    This is my second commit

commit d49f79120beecb2ea9e34b8398b4ee78bf662bf4
Author: Durga <durgaadvjava@gmail.com>
Date:   Thu May 7 20:52:12 2020 +0530

    This is my first commit
```

8.2) How to see Log Information of a Particular File:

git log <filename>
git log file1.txt

```
$ git log file1.txt
commit 4b77312160c82d76395558da415a96b2a8b36072
Author: Durga <durgaadvjava@gmail.com>
Date:   Thu May 7 21:05:25 2020 +0530

    This is second commit related to file1.txt

commit d49f79120beecb2ea9e34b8398b4ee78bf662bf4
Author: Durga <durgaadvjava@gmail.com>
Date:   Thu May 7 20:52:12 2020 +0530

    This is my first commit
```

Note: There are multiple options are availble for git log command to see the history.
git log --help

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.

Output:
7 characters of commit id + commit message

```
$ git log --oneline

48437a7 (HEAD -> master) This is 3rd commit for file2.txt
3a8051f Second commit for file2.txt
4b77312 This is second commit related to file1.txt
93d297b This is my second commit
d49f791 This is my first commit
```

***This option is very helpful if we have lot of commits and to identify commit based on message.**

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.

Syntax:

-<number>
-n <number>
--max-count=<number>
Limit the number of commits to output.

```
$ git log -n 2
commit b7bd0cfecb7cd64128f209a1de4cc0ffefdd9310 (HEAD -> master)
Author: Ravi <durgasoftonline@gmail.com>
Date:   Sat May 16 21:23:23 2020 +0530

    new file added

commit 44fe2785f2e3f30ebcf733ffdc278ce240364488
Author: Durga <durgasoftonline@gmail.com>
Date:   Sat May 16 21:05:07 2020 +0530

    file1.txt got modified
```

Note: we can use -n and --oneline options together also.

```
$ git log -n 2 --oneline
b7bd0cf (HEAD -> master) new file added
44fe278 file1.txt got modified
```

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"

It shows all commits which has given pattern in the commit message.

git log --grep="added" --oneline

```
$ git log --grep="added" --oneline
b7bd0cf (HEAD -> master) new file added
```

dcb4108 New files added

*** This option is very helpful if we follow a particular structure for the commit message. We can use this option to find all commits related to a particular request number or defect number etc.

git log --grep="defect_number" --oneline

Option-4: Show commits more recent than a specific Time

--since=<date>
--after=<date>
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=<date>
--before=<date>
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=<pattern>

git log --author=Ravi --oneline

$ git log --author=Ravi
commit 73e3bc5c0dd6c17c76cc50adc322545b2ba1efab (HEAD -> master)
Author: Ravi <durgasoftonlinetraining@gmail.com>
Date: Sun May 17 19:43:50 2020 +0530

commited a.txt

commit b7bd0cfecb7cd64128f209a1de4cc0ffefdd9310
Author: Ravi <durgasoftonline@gmail.com>
Date: Sat May 16 21:23:23 2020 +0530

new file added

Option-7: --decorate Option to display extra Information

This option will print some extra information like branch information,head information, tags information etc

```
$ git log --decorate --oneline
b7bd0cf (HEAD -> master) new file added
44fe278 file1.txt got modified
dcb4108 New files added
```

Note: There are multiple options are availble for git log command to see the history of all commits.

```
git log --help
```

TOPIC - 9

The Complete Story of git diff Command

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

Topic-9: The Complete Story of git diff Command

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.

Demo Example:

file1.txt

First line in file1.txt
Second line in file1.txt

file2.txt

First line in file2.txt
Second line in file2.txt

first commit: 2 files and each file contains 2 lines

file1.txt

First line in file1.txt
Second line in file1.txt
Third line in file1.txt
Fourth line in file1.txt

file2.txt

First line in file2.txt
Second line in file2.txt
Third line in file2.txt

Fourth line in file2.txt

2nd commit : 2 files and each file contains 4 lines.
Now we are adding new line in file1.txt in working directory

file1.txt

First line in file1.txt
Second line in file1.txt
Third line in file1.txt
Fourth line in file1.txt
Fifth line in file1.txt

We are adding file1.txt to staging area
git add file1.txt

Again we are adding a new line in file1.txt of working directory

file1.txt

First line in file1.txt
Second line in file1.txt
Third line in file1.txt
Fourth line in file1.txt
Fifth line in file1.txt
sixth line in file1.txt

Case-1: To see the difference in File Content between Working Directory and staging Area

```
$ git diff file1.txt
diff --git a/file1.txt b/file1.txt
index 0e17c9d..e3e329f 100644
--- a/file1.txt
+++ b/file1.txt
@@ -3,3 +3,4 @@ Second line in file1.txt
 Third line in file1.txt
 Fourth line in file1.txt
 Fifth line in file1.txt
+sixth line in file1.txt
```

1) diff --git a/file1.txt b/file1.txt
a/file1.txt means source copy which means staging area
b/file1.txt means destination copy which means working directory copy

2) index 0e17c9d..e3e329f 100644
0e17c9d → hash of source file content
e3e329f → hash of destination file content
100644 → git file mode
First 3 characters(100) represents the type of file.
100 means ASCII text file.
Next 3 characters represents the file permissions.
644 → rw-r--r--

3)--- a/file1.txt
--- means missing lines in staged copy

4) +++ b/file1.txt
+++ means new lines added in working directory version

5) @@ -3,3 +3,4 @@
-3,3
- means source version

from 3rd line onwards total 3 lines
+3,4
+ means destination version
from 3rd line onwards total 4 lines

If any line prefixed with space means it is unchanged.
If any line prefixed with + means it is added in destination copy.
If any line prefixed with - means it is removed in destination copy.

@@ -3,3 +3,4 @@
Second line in file1.txt
Third line in file1.txt
Fourth line in file1.txt
Fifth line in file1.txt
+sixth line in file1.txt

Clear indication that one line added in the working directory copy when compared with staged copy.

+sixth line in file1.txt

Case-2: To see the difference in File Content between Working Directory and Last Commit

The last commit can be referenced by HEAD.

git diff HEAD file1.txt
It shows the differences between working copy and last commit copy.

```
$ git diff HEAD file1.txt
diff --git a/file1.txt b/file1.txt
index cadd0e1..e3e329f 100644
--- a/file1.txt
+++ b/file1.txt
@@ -2,3 +2,5 @@ First line in file1.txt
 Second line in file1.txt
 Third line in file1.txt
 Fourth line in file1.txt
+Fifth line in file1.txt
+sixth line in file1.txt
```

Case-3: To see the difference in File Content between staged Copy and Last Commit

We have to use --staged option or --cached option.

git diff --staged HEAD file1.txt
It shows the differences between staged copy and last commit copy.
Here HEAD is optional. Hence the following 2 commands will produce same output

git diff --staged HEAD file1.txt
git diff --staged file1.txt

```
$ git diff --staged HEAD file1.txt
diff --git a/file1.txt b/file1.txt
index cadd0e1..0e17c9d 100644
--- a/file1.txt
+++ b/file1.txt
@@ -2,3 +2,4 @@ First line in file1.txt
 Second line in file1.txt
 Third line in file1.txt
 Fourth line in file1.txt
+Fifth line in file1.txt
```

Case-4: To see the difference in File Content between specific Commit and Working Directory Copy

git diff 7chracters_of_specified_commitid filename

$ git log --oneline
6745461 (HEAD -> master) 2 files and each file contains 4 lines
e5705a6 2 files and each file contains 2 lines

Eg:

```
$ git diff e5705a6 file1.txt
diff --git a/file1.txt b/file1.txt
index d4effe0..e3e329f 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1,2 +1,6 @@
 First line in file1.txt
 Second line in file1.txt
+Third line in file1.txt
+Fourth line in file1.txt
+Fifth line in file1.txt
+sixth line in file1.txt
```

Case-5: To see the difference in file content between specific commit and staging area copy:

git diff --staged e5705a6 file1.txt

```
$ git diff --staged e5705a6 file1.txt
diff --git a/file1.txt b/file1.txt
index d4effe0..0e17c9d 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1,2 +1,5 @@
 First line in file1.txt
 Second line in file1.txt
+Third line in file1.txt
+Fourth line in file1.txt
+Fifth line in file1.txt
```


DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

Case-6: To see the difference in File Content between 2 specified Commits:

$ git log --oneline
6745461 (HEAD -> master) 2 files and each file contains 4 lines
e5705a6 2 files and each file contains 2 lines

```
$ git diff e5705a6 6745461 file1.txt
diff --git a/file1.txt b/file1.txt
index d4effe0..cadd0e1 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1,2 +1,4 @@
 First line in file1.txt
 Second line in file1.txt
+Third line in file1.txt
+Fourth line in file1.txt
```

```
$ git diff 6745461 e5705a6 file1.txt
diff --git a/file1.txt b/file1.txt
index cadd0e1..d4effe0 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1,4 +1,2 @@
 First line in file1.txt
 Second line in file1.txt
-Third line in file1.txt
-Fourth line in file1.txt
```

Case-7: To see the difference in File Content between Last Commit and Last butone Commit

git diff HEAD HEAD^ file1.txt
git diff HEAD HEAD^1 file1.txt
git diff HEAD HEAD~1 file1.txt

HEAD → Reference to last commit
HEAD^ or HEAD^1 or HEAD~1 → Reference to last but one commit

```
$ git diff HEAD HEAD^ file1.txt
diff --git a/file1.txt b/file1.txt
index cadd0e1..d4effe0 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1,4 +1,2 @@
```

First line in file1.txt
Second line in file1.txt
-Third line in file1.txt
-Fourth line in file1.txt

Case-8: To see the differences in all Files Content between 2 specified Commits

$git commit -m '5th line added to file1.txt'
and removed 3rd and 4th line from file2.txt

$ git log --oneline
be5256c (HEAD -> master) 6th line added to file1, 3rd and 4th lines removed from file2
8ceda5e 5th line added to file1.txt
6745461 2 files and each file contains 4 lines
e5705a6 2 files and each file contains 2 lines

$ git diff 6745461 be5256c
diff --git a/file1.txt b/file1.txt
index cadd0e1..e3e329f 100644
--- a/file1.txt
+++ b/file1.txt
@@ -2,3 +2,5 @@ First line in file1.txt
Second line in file1.txt
Third line in file1.txt
Fourth line in file1.txt
+Fifth line in file1.txt
+sixth line in file1.txt
diff --git a/file2.txt b/file2.txt
index ad87203..3495851 100644
--- a/file2.txt
+++ b/file2.txt
@@ -1,4 +1,2 @@
First line in file2.txt
Second line in file2.txt
-Third line in file2.txt
-Fourth line in file2.txt

Case-9: To see the differences in Content between 2 Branches

$ git diff master test

It shows all differences between master branch and test branch

Case-10: To see the differences in Content between Local and Remote Repositories

$ git diff master origin/master

It shows all differences between master branch in local repository and master branch in remote repository.

Summary:

git diff <path>

Shows the differences in the content of working directory, staging area and local repostiory.

we can use in the following ways

1) **git diff file1.txt**
   To compare working directory copy with staged copy

2) **git diff HEAD file1.txt**
   To compare working directory copy with last commit copy

3) **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

4) **git diff <commit_id> file1.txt**
   To compare working directory copy with the specified commit copy.

5) **git diff --staged <commit_id> file1.txt**
   To compare staged copy with the specified commit copy.

6) **git diff <source_commit_id> <destination_commit_id> file1.txt**
   To compare content in the file between two commits

7) **git diff HEAD HEAD~1 file1.txt**

To compare content in the file between last commit and last but one commit.

8) **git diff <source_commit_id> <destination_commit_id>**

To compare content of all files between two commits.

9) **git diff master_test**

It shows all differences between master branch and test branch

10) **git diff master_origin/master**

It shows all differences between master branch in local repository and master branch in remote repository.

TOPIC - 10

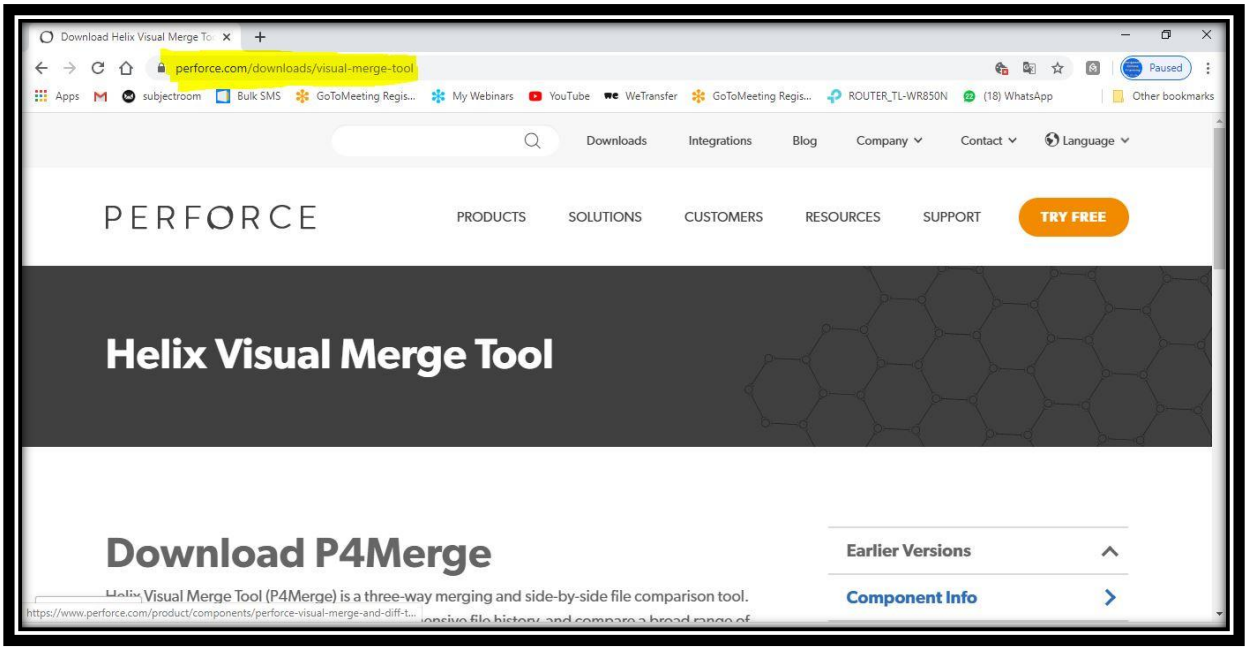
Helix Visual Merge Tool (p4merge) For Checking Differences

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

Topic-10: Helix Visual Merge Tool (p4merge) For Checking Differences

There are multiple tools are available like Helix Visual Merge Tool (P4Merge), meld etc.

How to download and install P4Merge:



We can use P4Merge tool for both comparison and merging purposes.
https://www.perforce.com/

downloads

Helix Visual Merge Tool (P4Merge)

Select our required platform
Windows 64-bit

skip registration

We will get the following exe file.
p4vinst64.exe file

P4MERGE will provide multiple utilities, But we require only Merge and Diff Tool.

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

Select only Merge and Diff Tool.

$ p4merge
bash: p4merge: command not found

We have to set path explicitly.
C:\Program Files\Perforce
This location contains our required p4merge application: p4merge.exe

How to Connect p4merge with git:

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
```

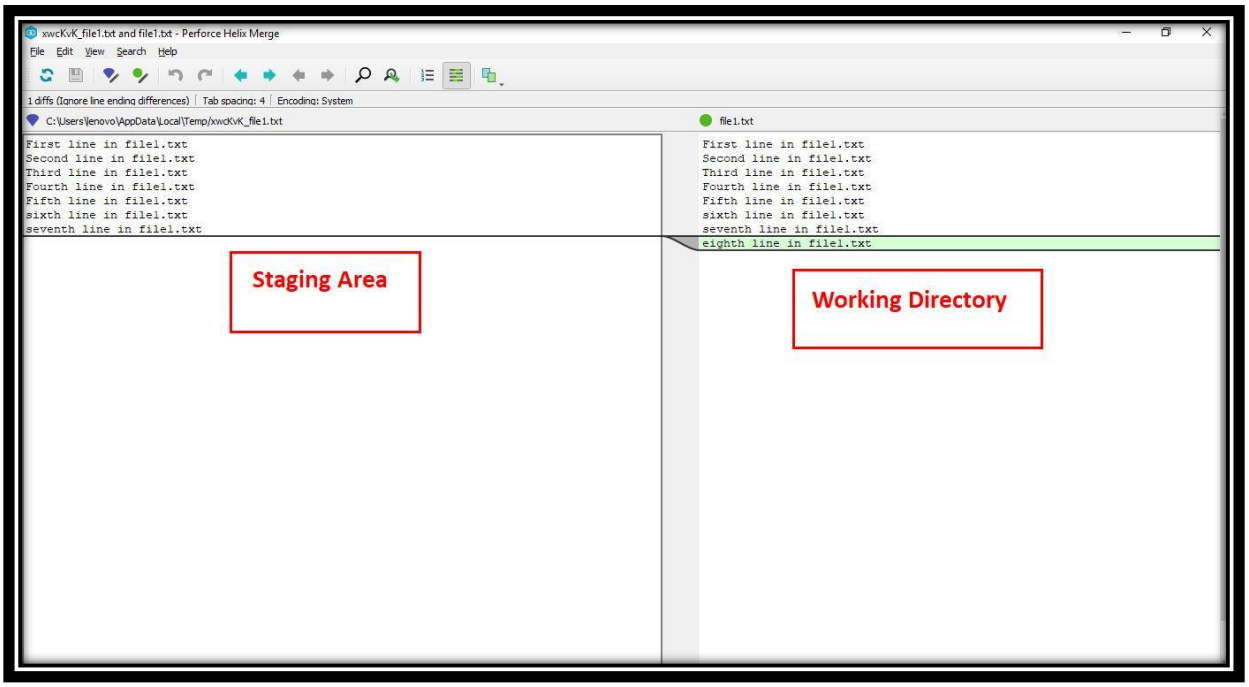
```
$ git config --global --list
user.name=Ravi
user.email=durgasoftonlinetraining@gmail.com
core.autocrlf=true
diff.tool=p4merge
difftool.p4merge.path=C:\Program Files\Perforce\p4merge.exe
difftool.prompt=false
merge.tool=p4merge
mergetool.p4merge.path=C:\Program Files\Perforce\p4merge.exe
mergetool.prompt=false
```

Continuition of Our Previous Example:

file1.txt → 7th line added and staged
file1.txt → 8th line added in working directory

Eg 1: Working Directory vs Staging Area

git diff file1.txt
git difftool file1.txt



Eg 2: Staging Area vs Last Commit

git diff --staged HEAD file1.txt
git difftool --staged HEAD file1.txt

Eg 3: Between 2 specified Commits

$ git log --oneline
be5256c (HEAD -> master) 6th line added to file1, 3rd and 4th lines removed from file2
8ceda5e 5th line added to file1.txt
6745461 2 files and each file contains 4 lines
e5705a6 2 files and each file contains 2 lines

git diff 6745461 be5256c file1.txt
git difftool 6745461 be5256c file1.txt

Note: p4merge tool can be used to compare only one file at a time.

TOPIC - 11

Removing Files by using git rm Command

Topic-11: Removing Files by using git rm Command

It is very common requirement to remove files from working directory and staging area. For these removals we can use the following commands

git rm file1.txt
git rm --cached file1.txt
General Linux rm command

Case-1: To Remove Files from Working Directory and staging Area (git rm)

If we want to remove a file from working directory and from staging area then we should go for gir rm command.

git rm file1.txt

file1.txt will be removed from staging area and from working directory

Note: for git rm command argument is mandatory
$ git rm
fatal: No pathspec was given. Which files should I remove?

$ git rm .
fatal: not removing '.' recursively without -r
It won't work because we didn't use -r option.

$ git rm -r .
It will remove all files

Case-2: To Remove Files Only from staging Area (git rm --cached)

If we want to remove the file only from staging area but not from working directory then we should use git rm --cached command.

git rm --cached file4.txt
file4.txt will be removed only from staging area but not from working directory

```
$ ls
file4.txt  file5.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project5 (master)
$ git ls-files
file4.txt
file5.txt

$ git rm --cached file4.txt
rm 'file4.txt'

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project5 (master)
$ ls
file4.txt  file5.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project5 (master)
$ git ls-files
file5.txt
```

Note: If we are not passing any argument,

```
$ git rm --cached
fatal: No pathspec was given. Which files should I remove?
```

Case-3: To Remove Files Only from Working Directory (rm Command)

We can use general linux command rm to remove files from working directory.

```
$ ls
file1.txt  file2.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project5 (master)
$ git ls-files
file1.txt
file2.txt

$ rm file1.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project5 (master)
$ ls
file2.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project5 (master)
$ git ls-files
```

file1.txt
file2.txt

Note:

1) **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.**

TOPIC - 12

Undo Changes with git Checkout Command

Topic-12: Undo Changes with git Checkout Command

We can use checkout command to discard unstanged 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.

git checkout -- filename

Eg:

```
$ git checkout -- file1.txt
```

It will discard any unstaged changes made in file1.txt.
After executing this command, staged copy content and working directory content is same.

```
$ cat file1.txt
first line in file1.txt
second line in file1.txt
This is third line in file1.txt
This is fourth line in file1.txt

$ git diff file1.txt
diff --git a/file1.txt b/file1.txt
index f718d29..862edcf 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1,2 +1,5 @@
 first line in file1.txt
 second line in file1.txt
+This is third line in file1.txt
+This is fourth line in file1.txt

$ git checkout -- file1.txt
```

```
$ cat file1.txt
first line in file1.txt
second line in file1.txt
```

Note: git checkout is applicable only for the files which are already tracked by git. It is not applicable for new files.

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project2 (master)
$ git checkout -- file4.txt
error: pathspec 'file4.txt' did not match any file(s) known to git
```

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.

TOPIC – 13

Git References (master and HEAD)

Topic-13: Git References (master and HEAD)

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.

Eg:

```
$pwd
/d/gitprojects/project6/.git/refs/heads
$ cat master
49aa8d79a9bab4c0d72dec217c0c6d5d96d604ce
```

Most of the times, we have to use the most recent commit id.
For such type of most commonly used commit ids git provides default references.

What is master?

```
$ git status
On branch 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.

The following two commands will produce same output.

```
$ git show 49aa8d7
$ git show master
```


DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

What is HEAD?

HEAD is a reference to master.

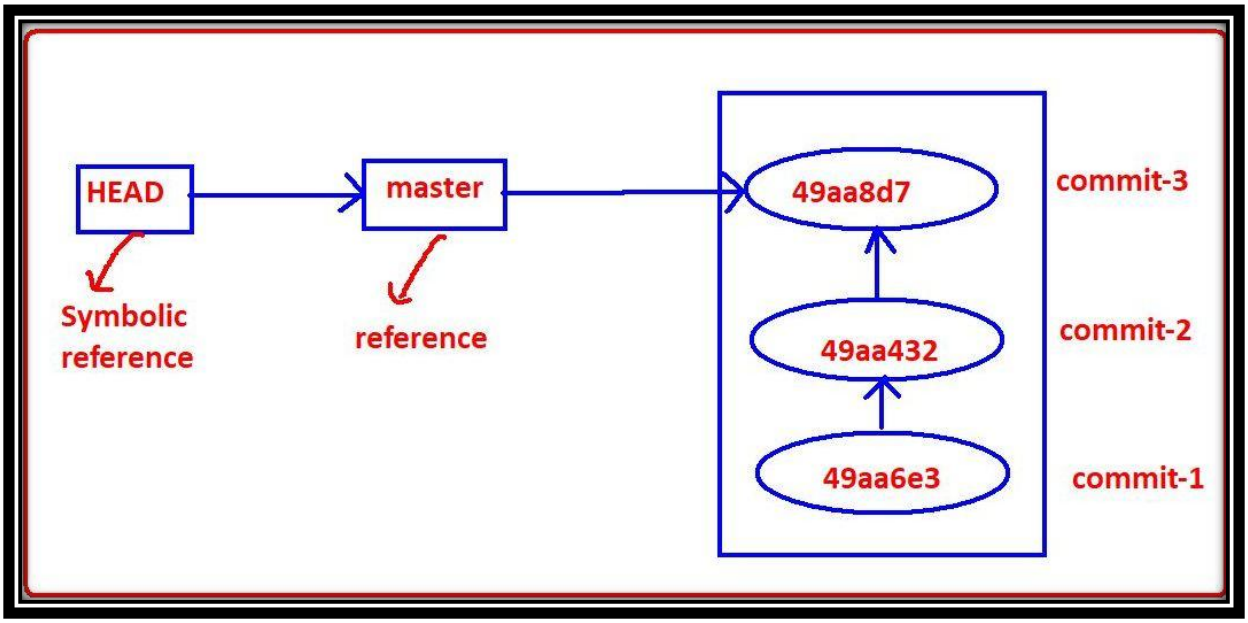
If any reference pointing to another reference, such type of reference is called symbolic reference. Hence HEAD is symbolic reference.

Bydefault HEAD is always pointing to branch(master).

$ git log --oneline
49aa8d7 (HEAD -> master) both files added

HEAD is stored in root of .git directory but not in .git/refs directory.

$ cat HEAD
ref: refs/heads/master



Detached HEAD:

Sometimes HEAD is not pointing to the branch name, such type of head is considered as Detached HEAD.

TOPIC - 14

Git reset Command

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

Topic-14: Git reset Command

git reset command is just like reset settings in our mobile.

There are 2 utilities of git reset command.

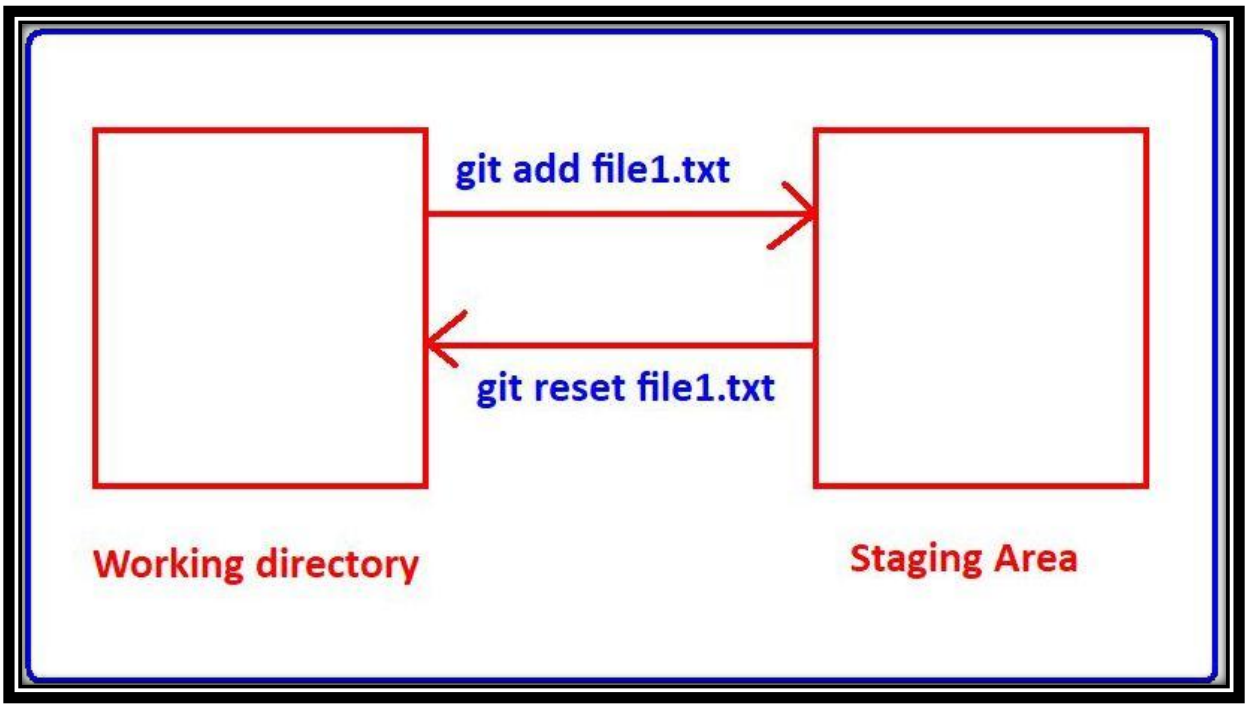
Utility-1: To remove changes from staging area
Utility-2: To undo commits at repository level

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.



DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project7 (master)
$ vi file1.txt
  First line in file1.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project7 (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file1.txt

nothing added to commit but untracked files present (use "git add" to track)

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project7 (master)
$ git add file1.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project7 (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   file1.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project7 (master)
$ git reset file1.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project7 (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file1.txt

nothing added to commit but untracked files present (use "git add" to track)
```

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).

We can see difference by using ls and git ls-files

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project7 (master)
$ ls
file1.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project7 (master)
$ git ls-files
file1.txt
```

Q) We modified the content of the file1.txt and added to staging area. But we want to ignore those changes in staging area and in working directory. For this requirement which commands we required to use?

git reset file1.txt

To ignore changes in staging area

git checkout -- file1.txt

To ignore changes in working directory

Utility-2: To undo Commits at Repository Level

We can also use reset to undo commits at repository level.

Syntax:

git reset <mode> <commitid>

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**
- **--keep**
- **--merge**

1)--mixed Mode:

It is the default mode.
To discard commits in the local repository and to discard changes in staging area we should use reset with --mixed option.
It won't touch working directory.

Example:

```
vi file1.txt
  First line in file1.txt

git add file1.txt; git commit -m 'file1 added'

vi file2.txt
  First line in file2.txt

git add file2.txt; git commit -m 'file2 added'

vi file3.txt
  First line in file3.txt

git add file3.txt; git commit -m 'file3 added'

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (master)
$ git log --oneline
6fcc300 (HEAD -> master) file3.txt added
86d0ca3 file2 added
9165d34 file1 added
```

To discard commit-3:

```
git reset --mixed 86d0ca3
git reset --mixed HEAD~1
git reset  HEAD~1

Now HEAD pointing to 86d0ca3
```

After undo commit-3:

The changes will be there in working directory.
option-1: To discard changes in working directory also
  git checkout -- filename
  But make sure this file should not be new file and should be already tracked by git.
option-2: If we want those changes to local repository

```
git add file3.txt; git commit -m 'file3 added again'
```

```
$ git log --oneline
59e6cd7 (HEAD -> master) file3 added again
86d0ca3 file2 added
9165d34 file1 added
```

To discard commit-2 and commit-3:

```
git reset --mixed 9165d34
git reset --mixed HEAD~2
git reset  HEAD~2

$ git log --oneline
9165d34 (HEAD -> master) file1 added

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (master)
$ git ls-files
file1.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (master)
$ ls
file1.txt  file2.txt  file3.txt

$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file2.txt
        file3.txt

nothing added to commit but untracked files present (use "git add" to track)
```

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.

```
$ git log --oneline
1979e61 (HEAD -> master) file3 added again
4d32eb3 file2 added again
9165d34 file1 added
```

To discard the latest commit:

```
git reset --soft 4d32eb3
git reset --soft HEAD~1
```

Now HEAD is pointing to 4d32eb3

The commits will be discarded only in local repository, but changes will be there in working directory and staging area

```
$ git ls-files
file1.txt
file2.txt
file3.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (master)
$ ls
file1.txt  file2.txt  file3.txt
```

To Revert Changes we have to do Just

```
git commit -m "added again"
```

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.

```
$ git log --oneline
3d7d370 (HEAD -> master) file3 added again
4d32eb3 file2 added again
9165d34 file1 added
```

To remove recent two commits permanently:

git reset --hard 9165d34
git reset --hard HEAD~2

Now changes will be removed from everywhere.

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (master)
$ git log --oneline
9165d34 (HEAD -> master) file1 added

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (master)
$ git ls-files
file1.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (master)
$ ls
file1.txt

--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.

Mode Name	Discard Changes in working directory	Discard Changes in Staging Area	Discard Changes in Local Repository
--mixed	NO	YES	YES
--soft	NO	NO	YES
--hard	YES	YES	YES

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.

TOPIC - 15

Git Aliases –
Providing our own Convenient Names
to
git Commands

Topic-15: Git Aliases - Providing our own Convenient Names to git Commands

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"**

Using alias Name:

$ git one
bb26af3 (HEAD -> master) two files we added
257073d file1 added

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

```
$ git s
git: 's' is not a git command. See 'git --help'.
$ git config --global alias.s "status"
$ git s
On branch master
nothing to commit, working tree clean
```

Note: If we use git in original command while creating alias name, what will happend?

```
$ git config --global alias.ss "git status"

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (master)
$ ss
bash: ss: command not found
```

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.
In the windows it will be available in C:\Users\lenovo

.gitconfig:

```
[user]
        name = Ravi
        email = durgasoftonlinetraining@gmail.com
[core]
        autocrlf = true
[diff]
        tool = p4merge
[difftool "p4merge"]
        path = C:\\Program Files\\Perforce\\p4merge.exe
[difftool]
        prompt = false
[merge]
        tool = p4merge
[mergetool "p4merge"]
        path = C:\\Program Files\\Perforce\\p4merge.exe
[mergetool]
        prompt = false
[alias]
        one = log --oneline
        s = status
```

We can perform any changes in alias commands based on requirement.
one = log

```
$ git one
commit bb26af3c6875a480ee0f92883ba85af5048eec6f (HEAD -> master)
Author: Ravi <durgasoftonlinetraining@gmail.com>
Date:   Tue May 26 19:40:13 2020 +0530

    two files we added

commit 257073dcecf4364b77e8c64dbd7386a71f4071a2
Author: Ravi <durgasoftonlinetraining@gmail.com>
Date:   Tue May 26 12:38:38 2020 +0530

    file1 added
```

TOPIC – 16

Ignoring unwanted Files
And
Directories by using .gitignore File

Topic-16: Ignoring unwanted Files and Directories by using .gitignore File

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.

```
# Don't track abc.txt file
abc.txt
# Don't track all .txt files
*.txt
# Don't track logs directory
logs/
#Don't track any hidden file
.*

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (master)
$ touch a.txt b.txt Customer.java

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (master)
$ mkdir logs

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (master)
$ touch logs/server.log logs/access.log

$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        Customer.java
        a.txt
            b.txt
        logs/
```


DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

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
.*
```

TOPIC – 17

Any Special Treatment For Directories by Git?

Topic-17: Any Special Treatment for Directories by Git?

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 status
On branch master
nothing to commit, working tree clean

$ mkdir dir1
```

Even though we created dir1, GIT won't give any importance for this directory because it does not contain any files.

```
$ git status
On branch master
nothing to commit, working tree clean

$ touch dir1/{a..d}.py
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        dir1/

nothing added to commit but untracked files present (use "git add" to track)

$ git add .
$ git status
On branch master
Changes to be committed:
(use "git restore --staged <file>..." to unstage)
        new file:   dir1/a.py
        new file:   dir1/b.py
        new file:   dir1/c.py
        new file:   dir1/d.py

$git commit -m 'all python files added'
```

Topic-18

Branching And Merging

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

18.1) What is Branching?

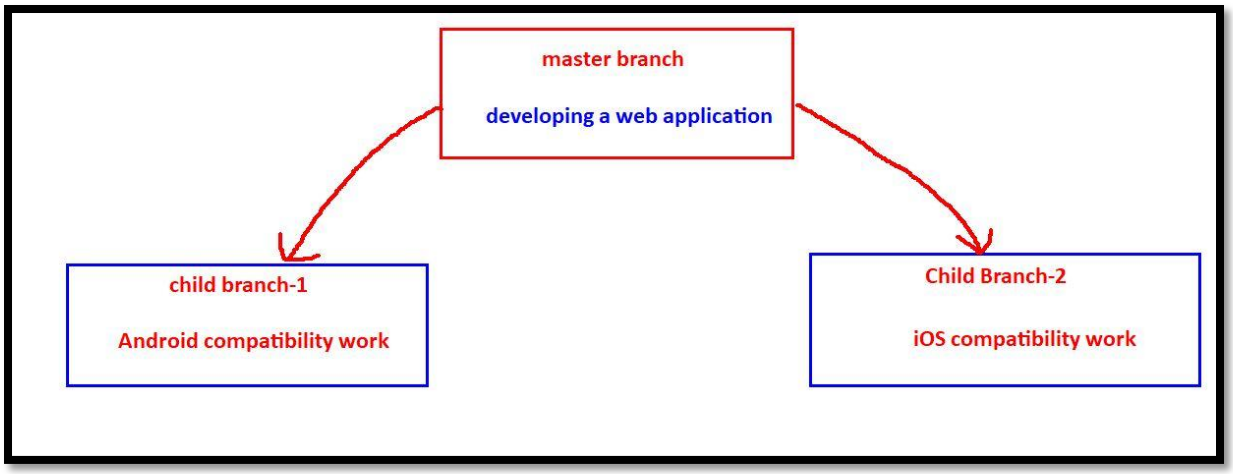
Branching is one of very important concept in version control systems.

While working on real time projects code base, branching is one of mandatory and unavoidable concept.

Till now whatever files created and whatever commits we did, all these happend in master branch.
master branch is the default branch/ main branch in git.
Generally main source code will be placed in master branch.

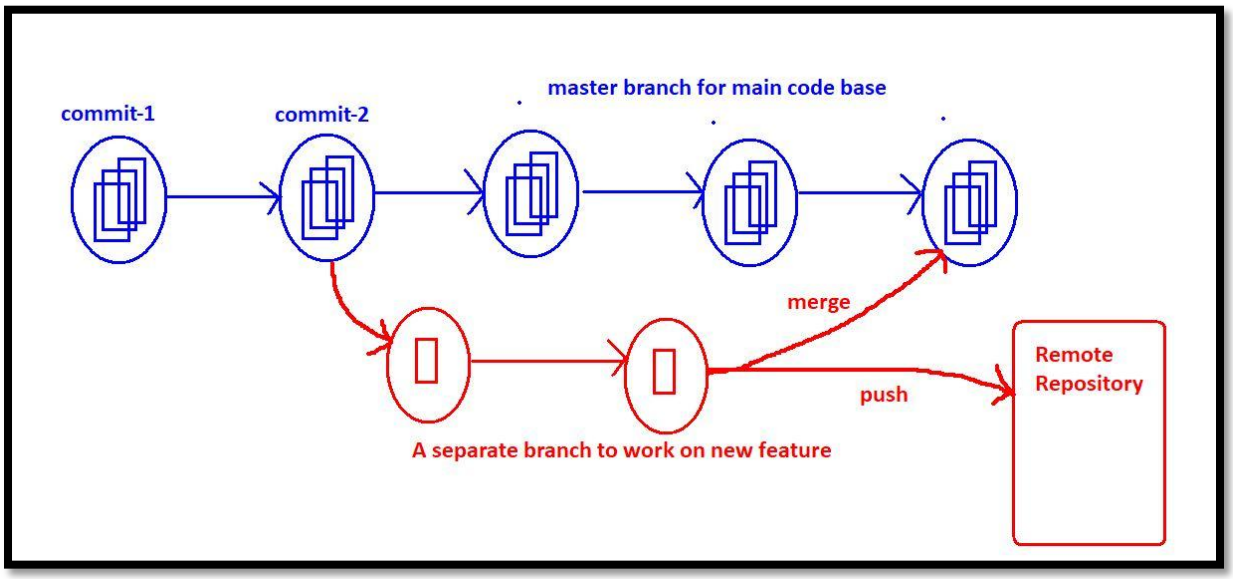
18.2) Need of creating a New Branch:

Assume we required to work on new requirements independently, then instead of working in the master branch, we can create a seperate branch and we can work in that branch, related to that new requirement without affecting main branch.



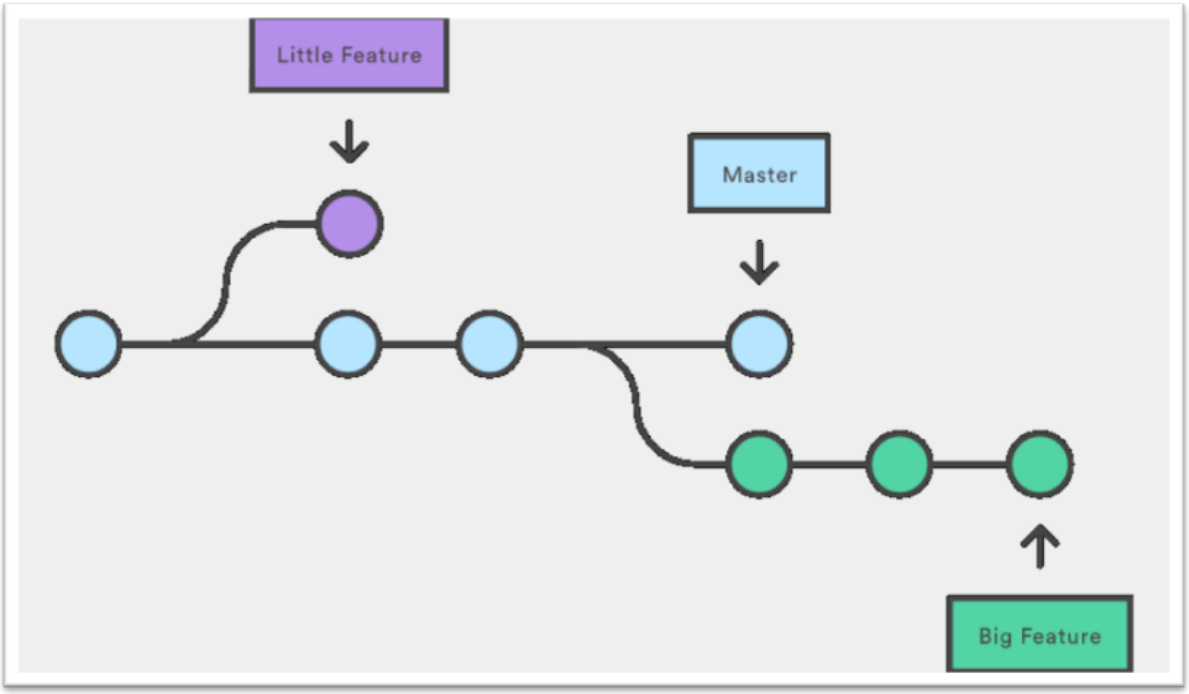
master branch → devlop web application
branch-1 → develop Android compatibility work
branch-2 → develop iOS compatibility work

A Branch is nothing but an independent flow of development and by using branching concept, we can create multiple work flows.



Based on our requirement, we can create any number of branches.

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com



Conclusions:

1. Once we creates a branch all files and commits will be inherited from parent branch to child branch. Branching is a logical way of duplicating files and commits. In the child branch we can create new files and we can perform new commits based on our requirements.

2. 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.

3. Once the work completed in new branch then we can merge that new brach to the main branch or we can push that branch directly to the remote repository.

18.3) Various Commands used in branching:

1) To View Branches:

To know all available branches in our local repository, we have to use git branch command.

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.**

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

$ git branch
*** master**
*** indicates that master is current active branch.**

Note:
There is another way to check on which branch currently we are working, for this we have to use git status command.

$ git status
On branch master
nothing to commit, working tree clean

2) How to Create a New Branch:

We can create a new branch by using git branch command.
Syntax: git branch brach_name

Eg:
$ git branch new1branch
It will create a new branch: new1branch

$ git branch
*** master**
new1branch

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

Eg:
$ git checkout new1branch
Switched to branch 'new1branch'

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project6 (new1branch)
$ git branch
master
*** new1branch**

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project6 (new1branch)
$ git status
On branch new1branch
nothing to commit, working tree clean.

***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

```
$ git checkout -b new2branch
Switched to a new branch 'new2branch'

$ git branch
master
new1branch
* new2branch
```

18.4) Demo Example for branching:

```
$ touch a.txt b.txt c.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (master)
$ git add a.txt;git commit -m 'a.txt added'
[master (root-commit) e74f011] a.txt added
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 a.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (master)
$ git add b.txt;git commit -m 'b.txt added'
[master 68e47e4] b.txt added
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 b.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (master)
$ git add c.txt;git commit -m 'c.txt added'
[master 78e7f07] c.txt added
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 c.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (master)
$ git status
On branch master
nothing to commit, working tree clean

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (master)
$ git log --oneline
78e7f07 (HEAD -> master) c.txt added
```

```
68e47e4 b.txt added
e74f011 a.txt added

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitproject
$ git checkout -b test
Switched to a new branch 'test'

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (test)
$ git status
On branch test
nothing to commit, working tree clean
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (test)
$ ls
a.txt  b.txt  c.txt

$ git log --oneline
78e7f07 (HEAD -> test, master) c.txt added
68e47e4 b.txt added
e74f011 a.txt added

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (test)
$ touch x.txt y.txt z.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (test)
$ git add .;git commit -m 'new files added'
[test 5a63a15] new files added
 3 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 x.txt
 create mode 100644 y.txt
 create mode 100644 z.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (test)
$ git log --oneline
5a63a15 (HEAD -> test) new files added
78e7f07 (master) c.txt added
68e47e4 b.txt added
e74f011 a.txt added

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (test)
$ git checkout master
Switched to branch 'master'

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (master)
$ ls
a.txt  b.txt  c.txt
```

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (master)
$ git log --oneline
78e7f07 (HEAD -> master) c.txt added
68e47e4 b.txt added
e74f011 a.txt added

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (master)
$ touch d.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (master)
$ git add d.txt;git commit -m 'd.txt added'
[master d8009dd] d.txt added
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 d.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (master)
$ ls
a.txt  b.txt  c.txt  d.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (master)
$ git log --oneline
d8009dd (HEAD -> master) d.txt added
78e7f07 c.txt added
68e47e4 b.txt added
e74f011 a.txt added

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (master)
$ git checkout test
Switched to branch 'test'

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (test)
$ ls
a.txt  b.txt  c.txt  x.txt  y.txt  z.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (test)
$ git log --oneline
5a63a15 (HEAD -> test) new files added
78e7f07 c.txt added
68e47e4 b.txt added
e74f011 a.txt added
```

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.

But in other version control systems like SVN, if we want to create a branch, first we have to create a new directory and we have to copy all files manually to that directory which is very difficult job and time consuming job.

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.

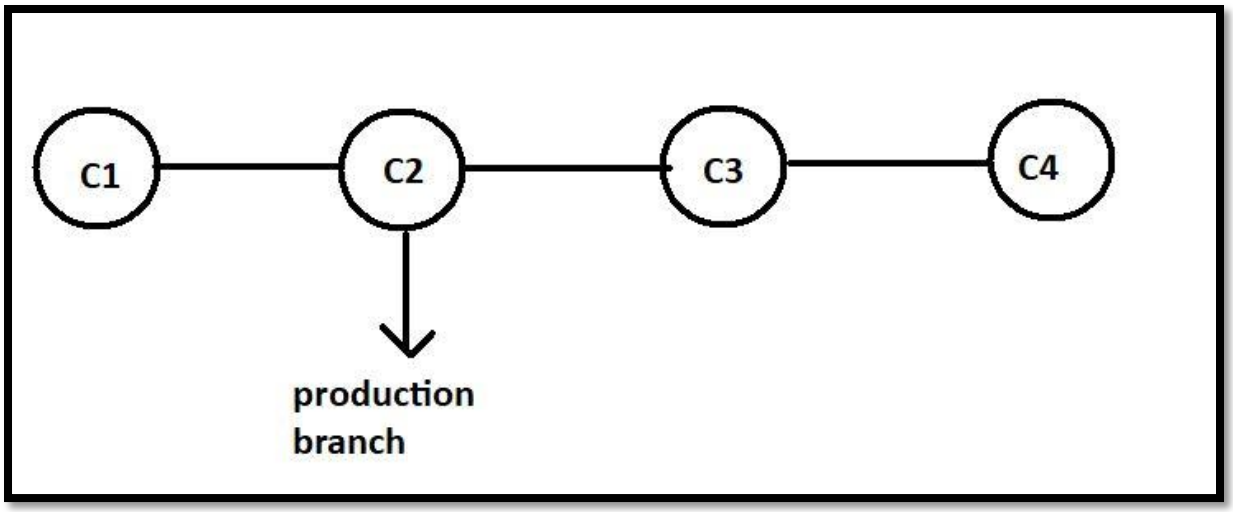
```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (master)
$ cat .git/HEAD
ref: refs/heads/master
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (master)
$ git checkout test
Switched to branch 'test'

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (test)
$ cat .git/HEAD
ref: refs/heads/test
```

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.

18.5) Multiple Use Cases where branching is required:

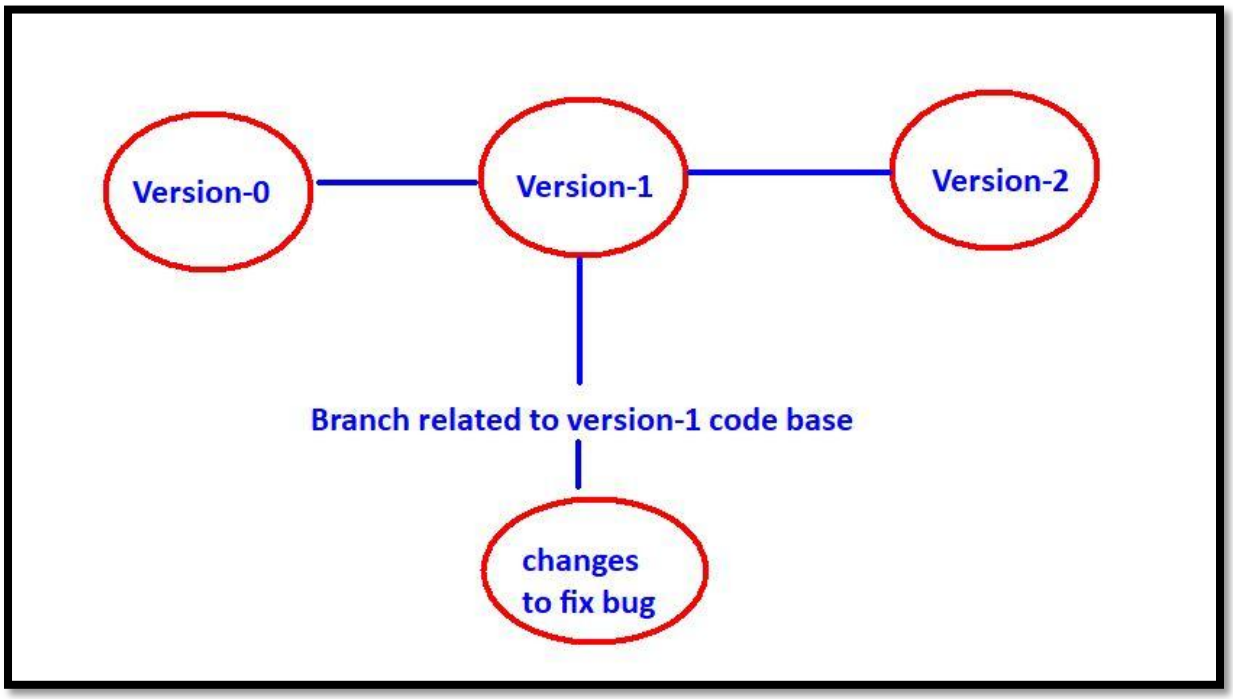
1. If we are working on a new feature of the project, and if it is required longer time then we can use branching. We can create a separate branch for Implementing new feature. It won't affect main code (master branch).

2. If we required to work on hot fixes of production code, then we can create branch for the production code base and we can work on that branch. Once work completed then we can push the fixed code to the production.
Most of the real time projects have a separate production branch to handle this type of requirements.



3. To support multiple versions of same code base, branching is required.

For every version, a separate branch will be there. If we want to fix any bugs or performance issues or any changes in a particular version, then we can work in that branch and we can push changes to the production.



4. To test new technologies or new ideas without effecting main code base, branching is the best choice.

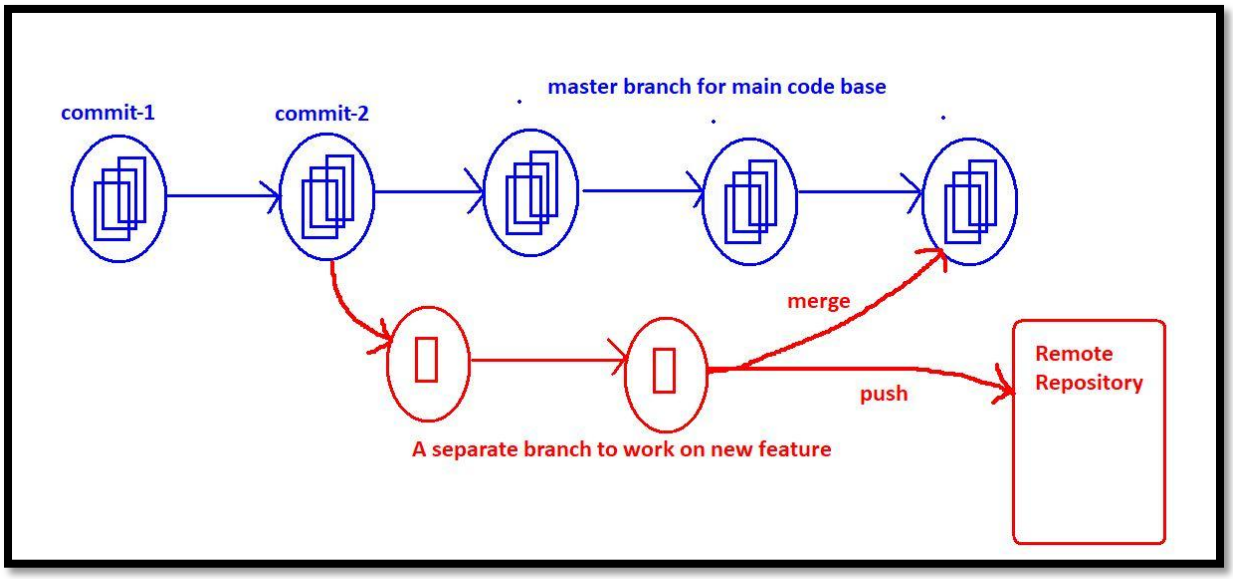
DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

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.

18.7) Merging of a Branch:

We created a branch to implement some new feature and we did some new changes in that branch, once work completed we have to merge that branch back to parent branch.



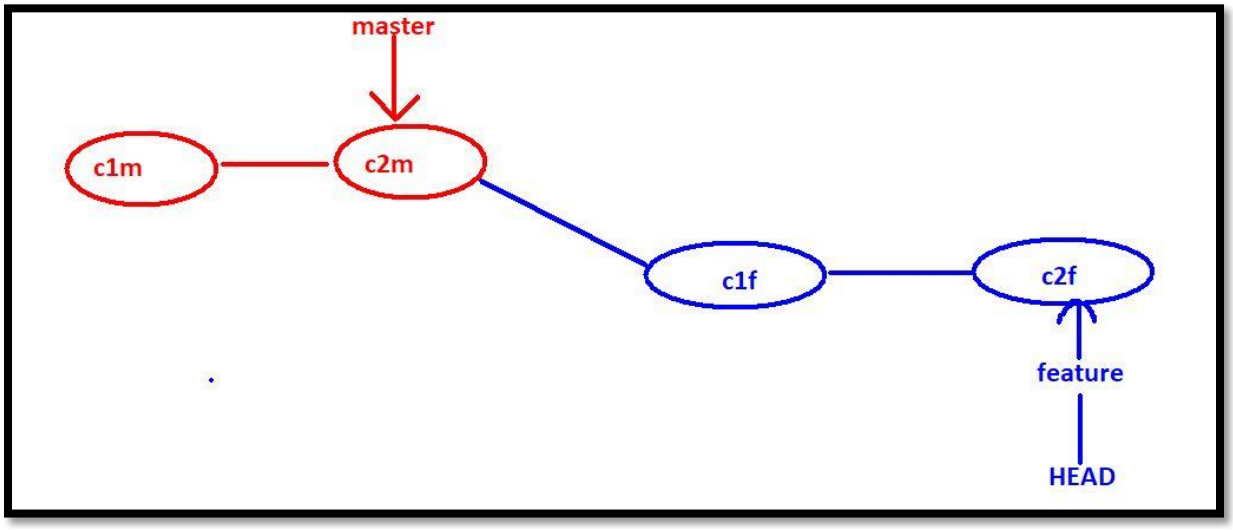
We can perform merge operation by using git merge command.
We have to execute this command from parent branch.

Demo Example:

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project9
$ touch a.txt b.txt
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project9 (master)
$ git add a.txt; git commit -m 'c1m'
[master (root-commit) 2164c45] c1m
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 a.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project9 (master)
$ git add b.txt; git commit -m 'c2m'
[master 99d500e] c2m
```

```
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 b.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project9 (master)
$ git log --oneline
99d500e (HEAD -> master) c2m
2164c45 c1m

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project9 (master)
$ git branch
* master

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project9 (master)
$ git checkout -b feature
Switched to a new branch 'feature'

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project9 (feature)
$ git branch
* feature
  master

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project9 (feature)
$ touch z.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project9 (feature)
$ git add z.txt;git commit -m 'c1f'
[feature 87701ef] c1f
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 z.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project9 (feature)
$ touch x.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project9 (feature)
$ git add x.txt;git commit -m 'c2f'
[feature 85d15fd] c2f
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 x.txt

$ git log --oneline master
99d500e (master) c2m
2164c45 c1m

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project9 (feature)
$ git log --oneline feature
```

85d15fd (HEAD -> feature) c2f
87701ef c1f
99d500e (master) c2m
2164c45 c1m



Assume new feature implemented properly, We want to merge feature branch with master branch.

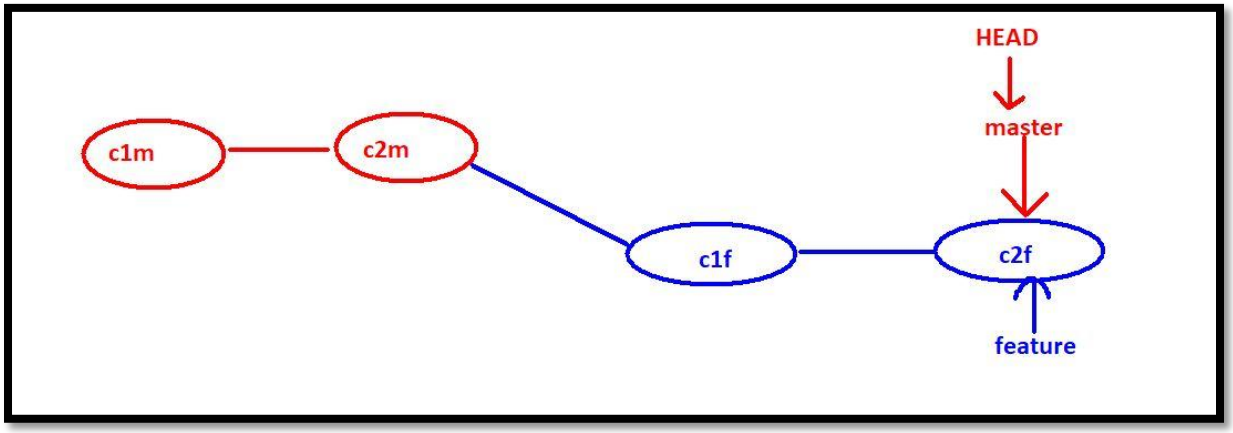
```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project9 (feature)
$ git checkout master
Switched to branch 'master'

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project9 (master)
$ git merge feature
Updating 99d500e..85d15fd
Fast-forward
 x.txt | 0
 z.txt | 0
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 x.txt
 create mode 100644 z.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project9 (master)
$ ls
a.txt  b.txt  x.txt  z.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project9 (master)
$ git log --oneline
85d15fd (HEAD -> master, feature) c2f
87701ef c1f
```

99d500e c2m
2164c45 c1m

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.



Demo Example to Demonstrate no Chance of raising conflicts in fast-forward Merge:

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project11 (master)
$ touch a.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project11 (master)
$ echo "first line" > a.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project11 (master)
$ git add a.txt ; git commit -m 'c1m'
[master (root-commit) f127932] c1m
 1 file changed, 1 insertion(+)
 create mode 100644 a.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project11 (master)
$ echo "secon line" >> a.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project11 (master)
$ git add a.txt ; git commit -m 'c2m'
```

```
[master 5aebe46] c2m
 1 file changed, 1 insertion(+)

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project11 (master)
$ git checkout -b 'feature'
Switched to a new branch 'feature'

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project11 (feature)
$ ls
a.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project11 (feature)
$ echo "third line" >> a.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project11 (feature)
$ git add a.txt ; git commit -m 'c1f'
warning: LF will be replaced by CRLF in a.txt.
The file will have its original line endings in your working directory
[feature edd47fd] c1f
 1 file changed, 1 insertion(+)

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project11 (feature)
$ git checkout master
Switched to branch 'master'

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project11 (master)
$ git merge feature
Updating 5aebe46..edd47fd
Fast-forward
 a.txt | 1 +
 1 file changed, 1 insertion(+)

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project11 (master)
$ cat a.txt
first line
secon line
third line
```

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.

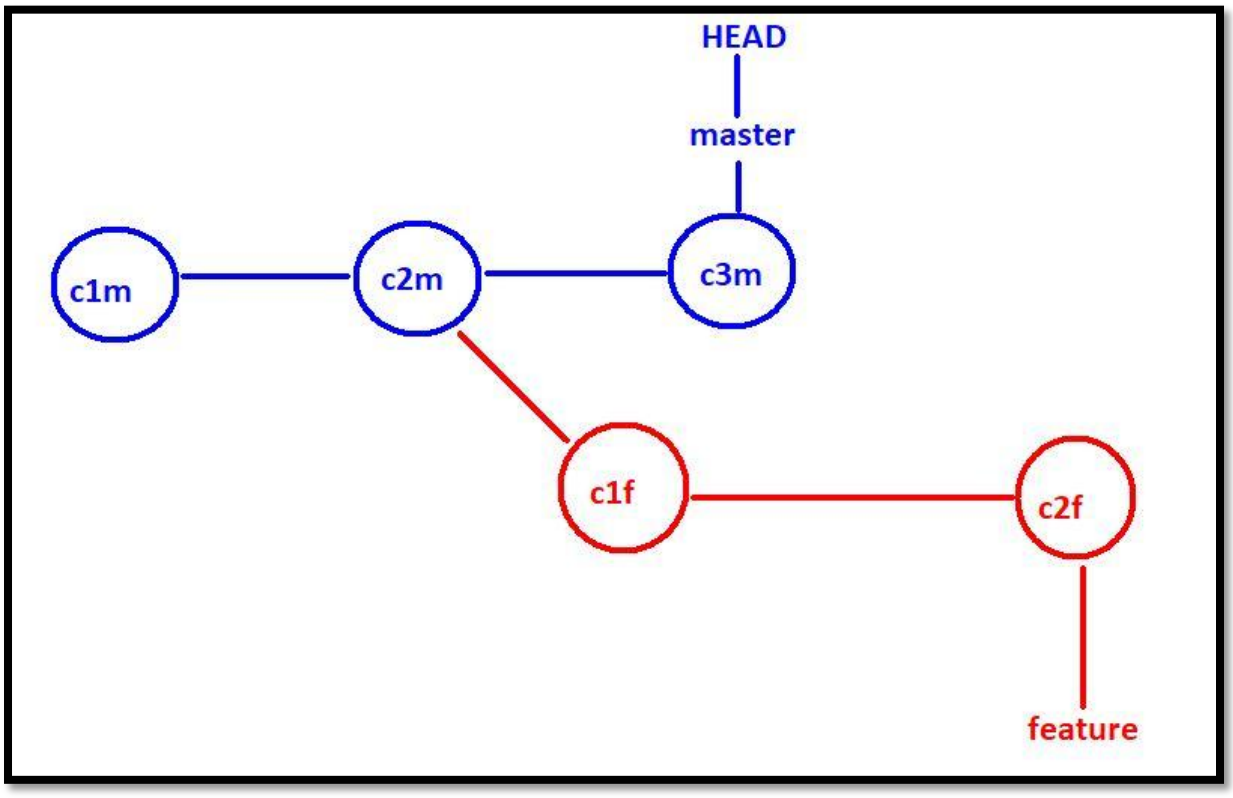
18.9) What is Three-Way Merge?

If changes present in both parent and child branches and if we are trying to perform merge operation, then git will do three-way merge.

Demo Example To Demonstrate Three-way Merge:

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project10 (master)
$ touch a.txt b.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project10 (master)
$ git add a.txt;git commit -m 'c1m'
[master (root-commit) 9e65e9f] c1m
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 a.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project10 (master)
$ git add b.txt;git commit -m 'c2m'
[master 56e0980] c2m
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 b.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project10 (master)
$ ls
a.txt  b.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project10 (master)
$ git log --oneline
56e0980 (HEAD -> master) c2m
9e65e9f c1m

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project10 (master)
$ git checkout -b feature
Switched to a new branch 'feature'

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project10 (feature)
$ touch x.txt z.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project10 (feature)
$ git add x.txt ; git commit -m 'c1f'
[feature 488588b] c1f
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 x.txt
```

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project10 (feature)
$ git add z.txt ; git commit -m 'c2f'
[feature 6a9b808] c2f
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 z.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project10 (feature)
$ git branch
* feature
  master

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project10 (feature)
$ git log --oneline master
56e0980 (master) c2m
9e65e9f c1m

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project10 (feature)
$ git log --oneline feature
6a9b808 (HEAD -> feature) c2f
488588b c1f
56e0980 (master) c2m
9e65e9f c1m
```

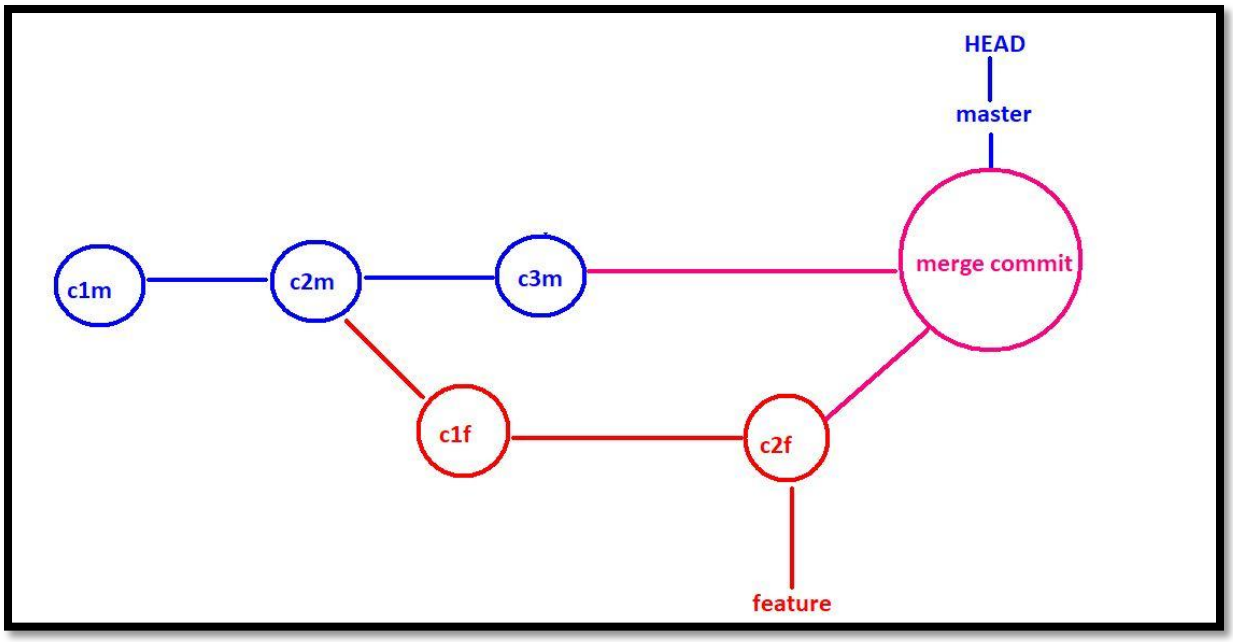


```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project10 (feature)
$ git checkout master
Switched to branch 'master'

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project10 (master)
$ touch c.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project10 (master)
$ git add c.txt ; git commit -m 'c3m'
[master 56fccfa] c3m
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 c.txt
```

In this case new commits are available in both parent and child branches.
If we are trying to perform merge operation, git will do three-way merge.

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project10 (master)
$ git merge feature
Merge made by the 'recursive' strategy.
 x.txt | 0
 z.txt | 0
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 x.txt
 create mode 100644 z.txt
```

Three-way merge creates a new commit which is also known as merge commit.
Parent branch will pointing to the newly created merge commit.



DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project10 (master)
$ git log --oneline --graph
*   65afca1 (HEAD -> master) Merge branch 'feature'
|\
| * 6a9b808 (feature) c2f
| * 488588b c1f
* | 56fccfa c3m
|/
* 56e0980 c2m
* 9e65e9f c1m
```


```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project10 (master)
$ git log --oneline --graph feature
* 6a9b808 (feature) c2f
* 488588b c1f
* 56e0980 c2m
* 9e65e9f c1m
```

18.10) Differences between Fast-forward and Three-way Merges

Fast-forward	Three-way Merge
1. After creating child branch, if updates are available only in the child branch but not in the parent branch, then GIT will perform Fast-Forward Merge.	1. After creating child branch, if updates are available in both Parent and child branches, then GIT will perform Three-way Merge.
2. It does not require any additional commit	2. It requires a new commit which is also known as Merge commit.
3. There is no chance of conflicts because new commits are available only in child branch but not in parent branch.	3. There may be a chance of conflicts because new commits are available in both parent and child branches.
4. Fast-forward merge is fully handled by GIT.	4. If there is a conflict, we may required to handle manually.

18.11) 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.

Demo Example:

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects
$ mkdir project12

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects
$ cd project12

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project12
$ git init
Initialized empty Git repository in D:/gitprojects/project12/.git/

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project12 (master)
$ echo "First Line Added" > a.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project12 (master)
$ git add a.txt ; git commit -m 'c1m'
[master (root-commit) dd727c4] c1m
 1 file changed, 1 insertion(+)
 create mode 100644 a.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project12 (master)
$ echo "Second Line Added" >> a.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project12 (master)
$ git add a.txt ; git commit -m 'c2m'
[master 1a42e6d] c2m
 1 file changed, 1 insertion(+)

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project12 (master)
$ git checkout -b feature
```

```
Switched to a new branch 'feature'

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project12 (feature)
$ echo "New Data Added By Feature Branch" >> a.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project12 (feature)
$ git add a.txt ; git commit -m 'c1f'
[feature c5bf898] c1f
 1 file changed, 1 insertion(+)

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project12 (feature)
$ git checkout master
Switched to branch 'master'

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project12 (master)
$ echo "New Data Added By Master Branch" >> a.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project12 (master)
$ git add a.txt ; git commit -m 'c3m'
[master 603072f] c3m
 1 file changed, 1 insertion(+)

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project12 (master)
$ git diff master feature
diff --git a/a.txt b/a.txt
index adefdaf..044a856 100644
--- a/a.txt
+++ b/a.txt
@@ -1,3 +1,3 @@
 First Line Added
 Second Line Added
-New Data Added By Master Branch
+New Data Added By Feature Branch

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project12 (master)
$ git merge feature
Auto-merging a.txt
CONFLICT (content): Merge conflict in a.txt
Automatic merge failed; fix conflicts and then commit the result.

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project12 (master|MERGING)
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
```

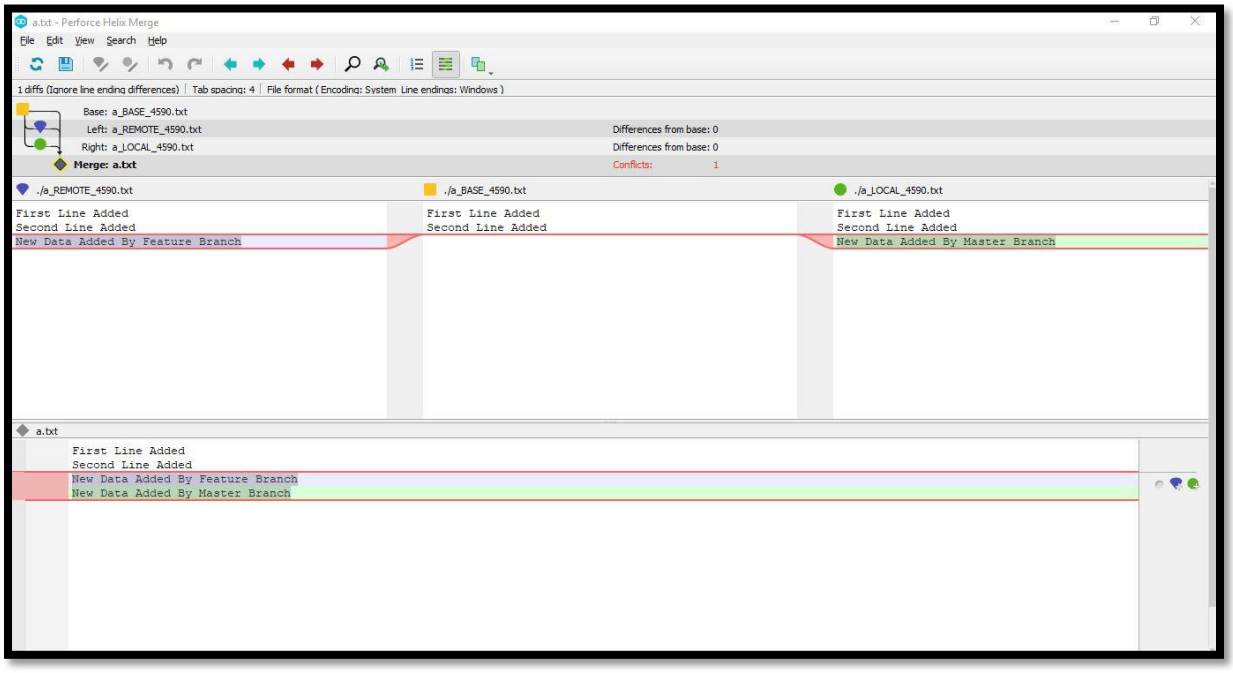
(use "git merge --abort" to abort the merge)

Unmerged paths:
(use "git add <file>..." to mark resolution)
both modified: a.txt

no changes added to commit (use "git add" and/or "git commit -a")

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project12 (master|MERGING)
$ cat a.txt
First Line Added
Second Line Added
<<<<<<< HEAD
New Data Added By Master Branch
=======
New Data Added By Feature Branch
>>>>>>> feature

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project12 (master|MERGING)
$ git mergetool

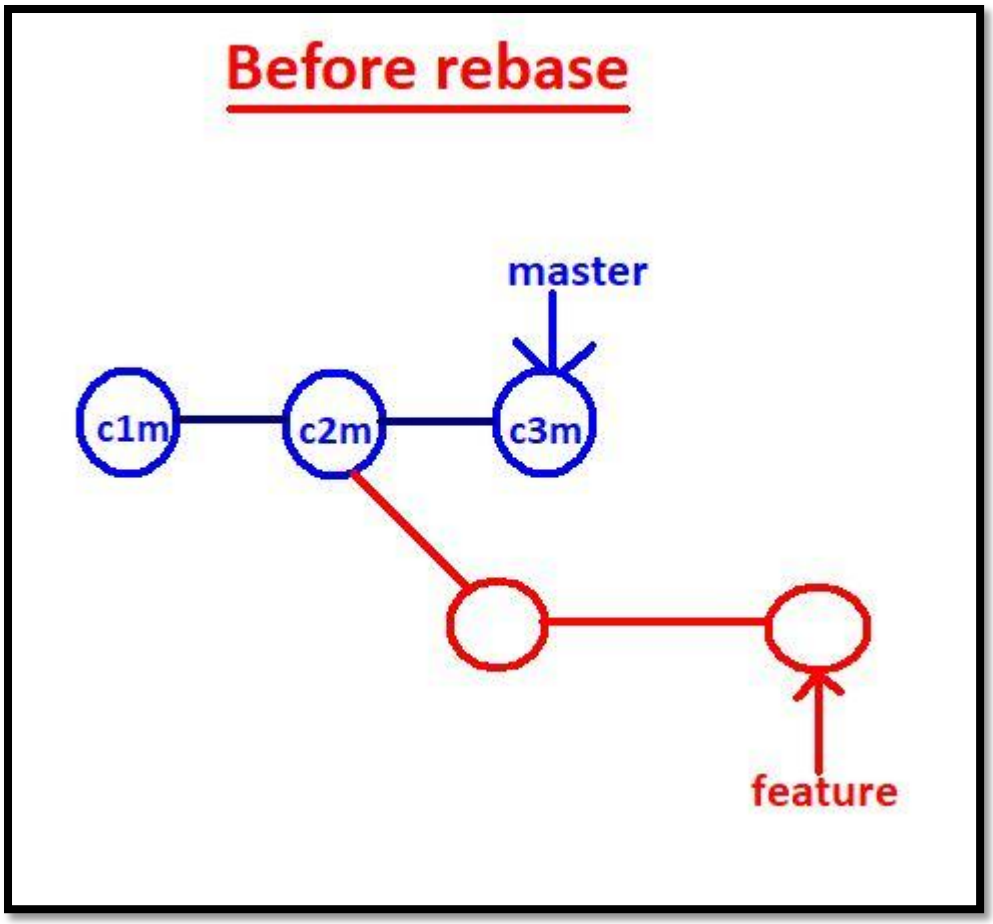


lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project12 (master|MERGING)
$ vi a.txt

We have to edit a.txt to decide final content.

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project12 (master|MERGING)
$ git add a.txt ; git commit -m 'Resolved Merge Conflicts'
[master 63f541a] Resolved Merge Conflicts

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project12 (master)
$ git status
On branch master
nothing to commit, working tree clean

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project12 (master)
$ git log --oneline --graph
*   63f541a (HEAD -> master) Resolved Merge Conflicts
|\
| * c5bf898 (feature) c1f
* | 603072f c3m
|/
* 1a42e6d c2m
* dd727c4 c1m

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project12 (master)
$ cat a.txt
First Line Added
Second Line Added
New Data Added By Master Branch
New Data Added By Feature Branch
```



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 <branch_name>**

Eg:

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project12 (master)
$ git branch
  feature
* master

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project12 (master)
$ git branch -d feature
Deleted branch feature (was c5bf898).

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project12 (master)
$ git branch
```


DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

*** master**

After deleting the branch, still files and commits are available because the changes are merged to master branch.

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project12 (master)
$ git log --oneline --graph
*   63f541a (HEAD -> master) Resolved Merge Conflicts
|\
| * c5bf898 c1f
* | 603072f c3m
|/
* 1a42e6d c2m
* dd727c4 c1m
```

Note:

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
```

Topic-19

Merging By using Rebase

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

19.1 Process of rebasing
19.2. Demo Example for rebasing
19.3. Advantages of rebasing
19.4. Disadvantages of rebasing
19.5 Differences between Merge and Rebase

Rebase is alternative way to merge changes of two branches togther.
rebase = re + base → re arrange base

19.1) Process of rebasing:

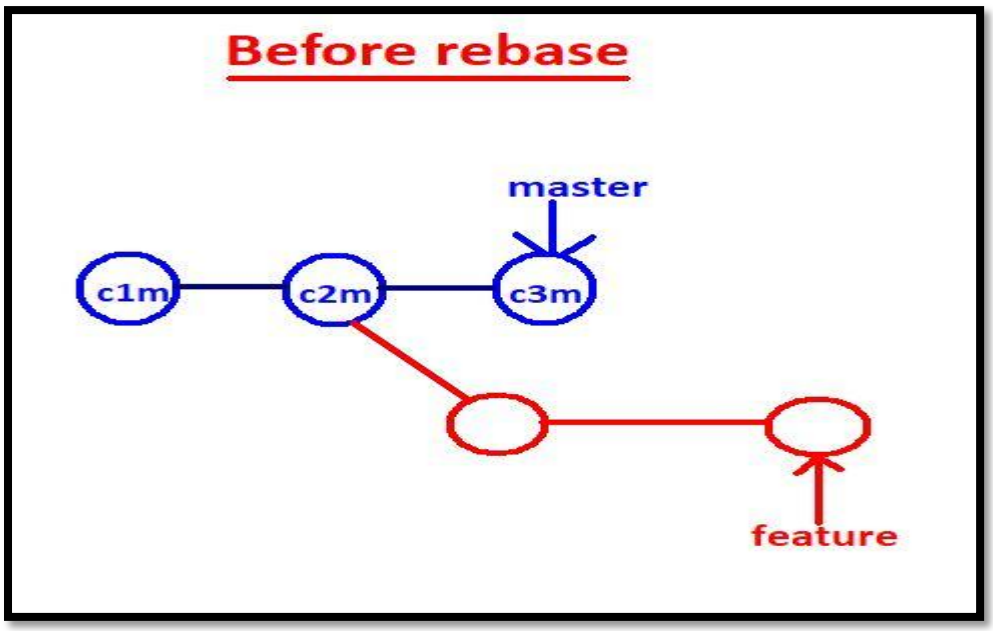
It is a two step process.

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



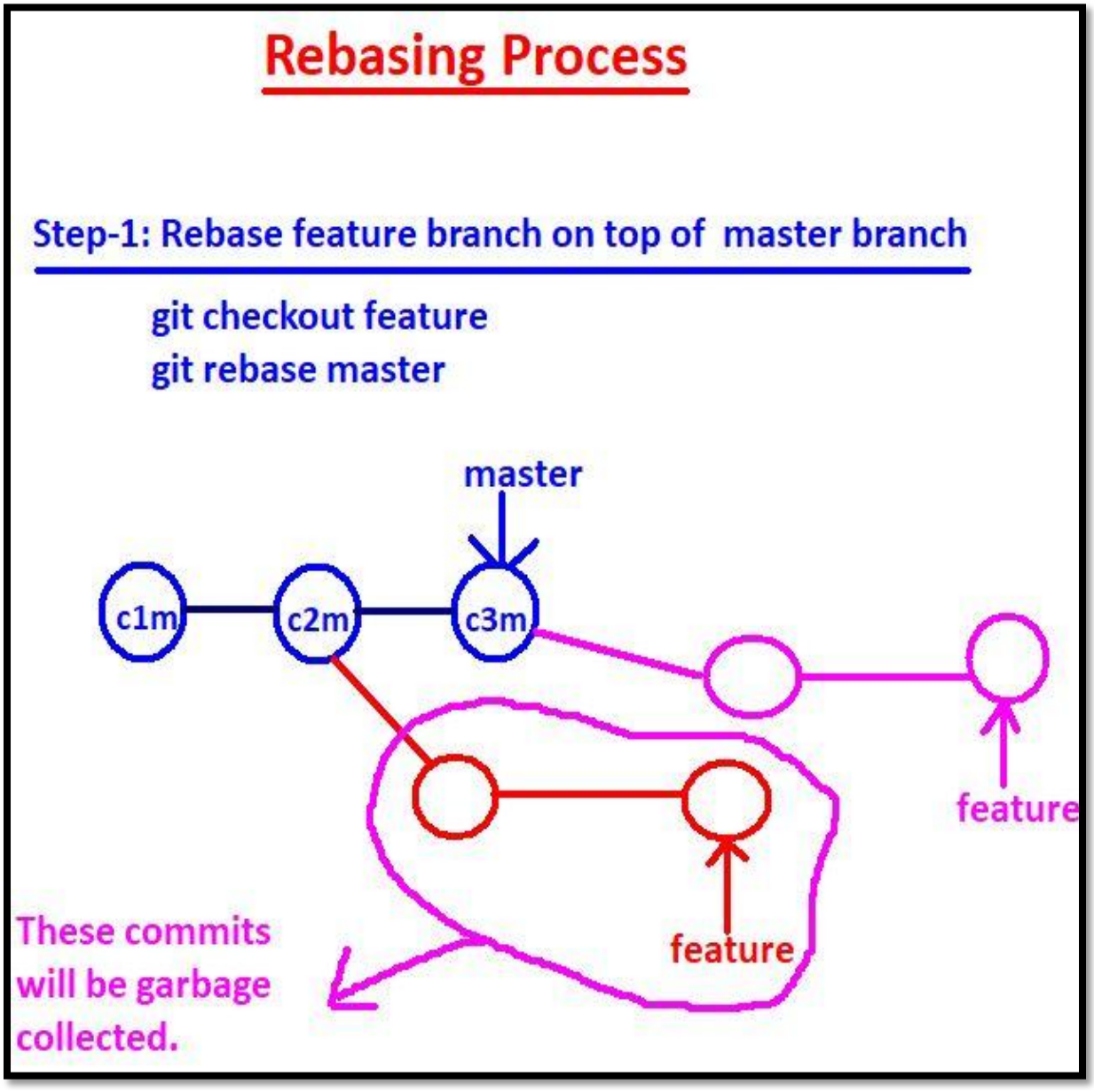
DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

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.

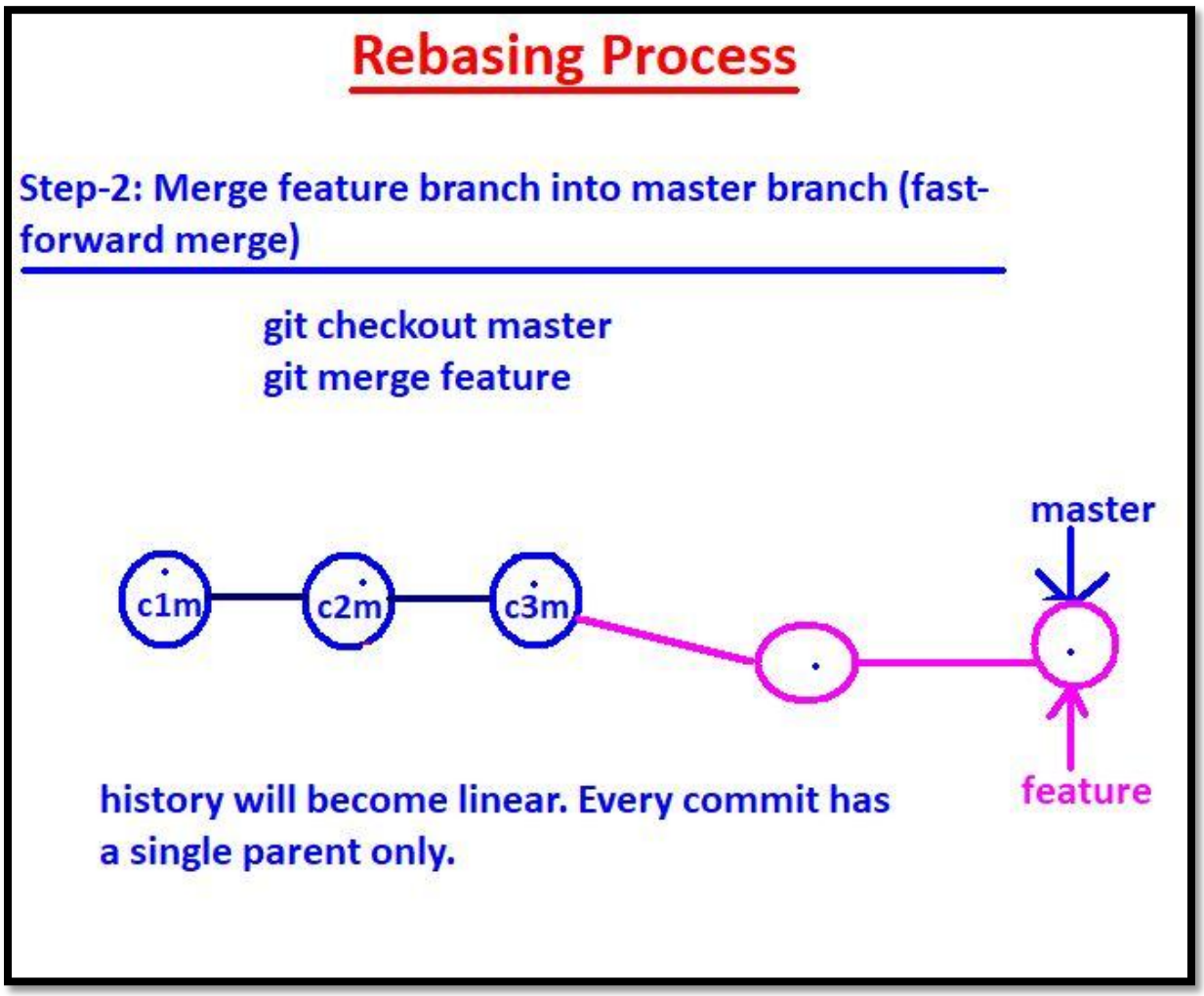
The base commit of the feature branch(duplicate copy) will be updated as last commit of parent branch(master branch).



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.



DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

19.2) Demo Example for rebasing:

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects
$ mkdir rebasing

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects
$ cd rebasing

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/rebasing
$ git init
Initialized empty Git repository in D:/gitprojects/rebasing/.git/

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/rebasing (master)
$ touch a.txt b.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/rebasing (master)
$ git add a.txt;git commit -m 'c1m'
[master (root-commit) 27458a4] c1m
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 a.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/rebasing (master)
$ git add b.txt;git commit -m 'c2m'
[master d2369f5] c2m
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 b.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/rebasing (master)
$ git checkout -b feature
Switched to a new branch 'feature'

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/rebasing (feature)
$ touch x.txt y.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/rebasing (feature)
$ git add x.txt;git commit -m 'c1f'
[feature a7de761] c1f
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 x.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/rebasing (feature)
$ git add y.txt;git commit -m 'c2f'
[feature 56661b6] c2f
 1 file changed, 0 insertions(+), 0 deletions(-)
```

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/rebasing (feature)
$ git checkout master
Switched to branch 'master'

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/rebasing (master)
$ touch c.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/rebasing (master)
$ git add c.txt;git commit -m 'c3m'
[master 76f925d] c3m
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 c.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/rebasing (master)
$ git log master
commit 76f925d5bc1f019c0d37aeb2019428c02a1bd9a2 (HEAD -> master)
Author: Ravi <durgasoftonlinetraining@gmail.com>
Date:   Wed Jun 3 12:41:39 2020 +0530

    c3m
commit d2369f5d777eff029551b018fd4800d3471fbebf
Author: Ravi <durgasoftonlinetraining@gmail.com>
Date:   Wed Jun 3 12:39:44 2020 +0530

    c2m
commit 27458a4980b808ac346efa86ede08ea49c3f1719
Author: Ravi <durgasoftonlinetraining@gmail.com>
Date:   Wed Jun 3 12:39:33 2020 +0530

    c1m
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/rebasing (master)
$ git log feature
commit 56661b62bd8b81cbe5347ac4e6bc9cc57a8850d8 (feature)
Author: Ravi <durgasoftonlinetraining@gmail.com>
Date:   Wed Jun 3 12:40:59 2020 +0530

    c2f
commit a7de761b18a07f44c34cd88a01c204d4c68df697
Author: Ravi <durgasoftonlinetraining@gmail.com>
Date:   Wed Jun 3 12:40:39 2020 +0530

    c1f
commit d2369f5d777eff029551b018fd4800d3471fbebf
Author: Ravi <durgasoftonlinetraining@gmail.com>
Date:   Wed Jun 3 12:39:44 2020 +0530
```

c2m
commit 27458a4980b808ac346efa86ede08ea49c3f1719
Author: Ravi <durgasoftonlinetraining@gmail.com>
Date: Wed Jun 3 12:39:33 2020 +0530

c1m
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/rebasing (master)
$ git log --oneline master
76f925d (HEAD -> master) c3m
d2369f5 c2m
27458a4 c1m

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/rebasing (master)
$ git log --oneline feature
56661b6 (feature) c2f
a7de761 c1f
d2369f5 c2m
27458a4 c1m

Step-1: We have to rebase feature branch on top of master branch.

git checkout feature
git rebase master

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/rebasing (master)
$ git checkout feature
Switched to branch 'feature'

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/rebasing (feature)
$ git rebase master
Successfully rebased and updated refs/heads/feature.

$ git log --oneline feature
d00c558 (HEAD -> feature) c2f
d96d3ae c1f
76f925d (master) c3m
d2369f5 c2m
27458a4 c1m

$ git log feature
commit d00c5589350abc457fd511c560ee24dad2ed12a3 (HEAD -> feature)
Author: Ravi <durgasoftonlinetraining@gmail.com>
Date: Wed Jun 3 12:40:59 2020 +0530

c2f
commit d96d3ae369b5d9394615baa4ea5fd1393016cc04
Author: Ravi <durgasoftonlinetraining@gmail.com>
Date: Wed Jun 3 12:40:39 2020 +0530

c1f
commit 76f925d5bc1f019c0d37aeb2019428c02a1bd9a2 (master)
Author: Ravi <durgasoftonlinetraining@gmail.com>
Date: Wed Jun 3 12:41:39 2020 +0530

c3m
commit d2369f5d777eff029551b018fd4800d3471fbebf
Author: Ravi <durgasoftonlinetraining@gmail.com>
Date: Wed Jun 3 12:39:44 2020 +0530

c2m
commit 27458a4980b808ac346efa86ede08ea49c3f1719
Author: Ravi <durgasoftonlinetraining@gmail.com>
Date: Wed Jun 3 12:39:33 2020 +0530

c1m

Step-2: We have to merge feature branch into the master branch (fast-forwar merge will be happend)

```
git checkout master
git merge feature
```

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/rebasing (feature)
$ git checkout master
Switched to branch 'master'

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/rebasing (master)
$ git merge feature
Updating 76f925d..d00c558
Fast-forward
 x.txt | 0
 y.txt | 0
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 x.txt
 create mode 100644 y.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/rebasing (master)
$ git log --oneline --graph master
* d00c558 (HEAD -> master, feature) c2f
```

```
* d96d3ae c1f
* 76f925d c3m
* d2369f5 c2m
* 27458a4 c1m

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/rebasing (master)
$ git branch -d feature
Deleted branch feature (was d00c558).

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/rebasing (master)
$ git log --oneline --graph master
* d00c558 (HEAD -> master) c2f
* d96d3ae c1f
* 76f925d c3m
* d2369f5 c2m
* 27458a4 c1m
```

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.

19.5) Differences between Merge and Rebase

Merge	Rebase
1. It is a single step process git checkout master git merge feature	1. It is a two-step process git checkout feature git rebase master git checkout master git merge feature
2. Merge preserves history of all commits.	2. Rebase clears history of feature branch.
3. The commits can have more than one parent and history is non-linear.	3. Every commit has only one parent and history is linear.
4. In merge, there may be a chance of conflicts.	4. In Rebase, there is no chance of conflicts.
5. We can aware which changes are coming from which branch.	5. We can not aware which changes are coming from which branch.
6. We can use merge on public repositories.	6. It is not recommended to use rebase on public repositories.

Note:

Rebase is very dangerous operation and it is never recommended to use on public repositories because it rewrites history.

Topic-20

Stash in GIT

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

20.1 What is git stash?
20.2 Demo Example for stashing
20.3 How to list all available stashes?
20.4 How to check the contents of stash?
20.5 How to perform unstash?
20.6 Partial Stash
20.7 How to delete the stash?

Stashing is a bit advanced concept in GIT.
Most of the people may not aware about this topic.

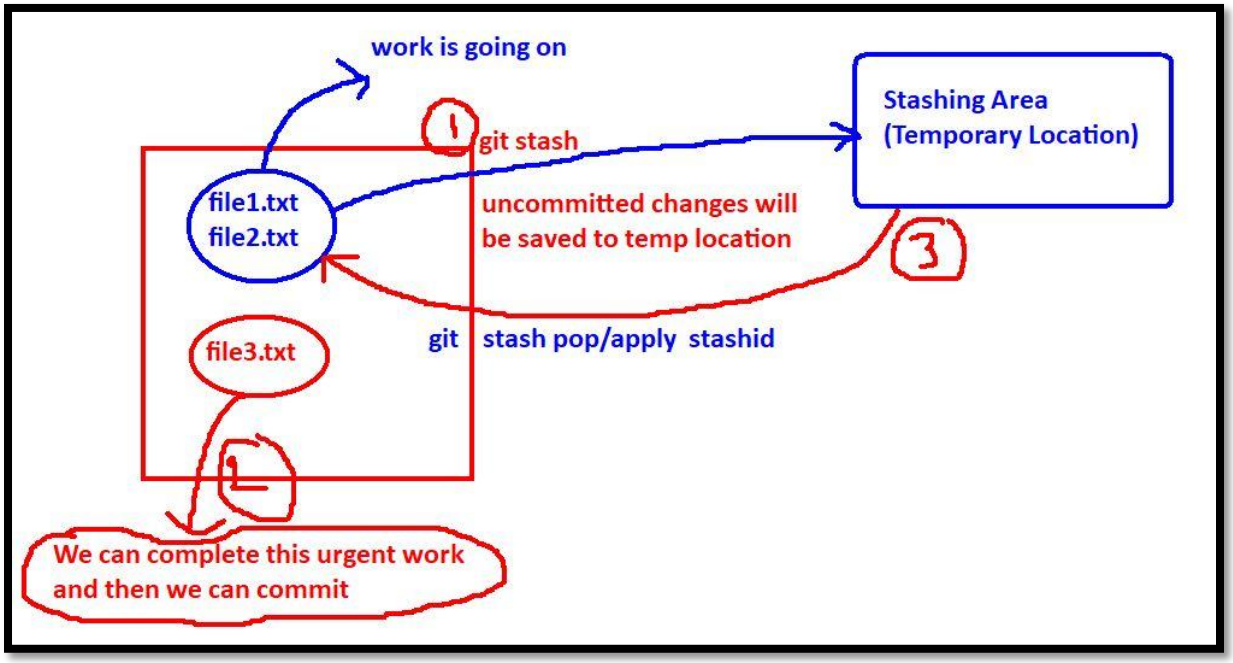
Eg-1:
Stash → Store (something) safely in a hidden or secret place.
Eg: Most of billionaire shashed their welath in Swiss banks"

Eg-2:
Doctor → op is going on (6PM to 9PM)
An urgent case came → Doctor will pause op and then he look into urgent case

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.



DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 88 85 25 26 27, 72 07 21 24 27/28 | www.durgasoftonline.com
Maii: durgasoftonline@gmail.com

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
```

20.2) Demo Example for stashing:

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects
$ mkdir stashing

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects
$ cd stashing

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashing
$ git init
Initialized empty Git repository in D:/gitprojects/stashing/.git/

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashing (master)
$ echo "First Line in File1" > file1.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashing (master)
$ echo "First Line in File2" > file2.txt
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashing (master)
$ git add file1.txt file2.txt;git commit -m '2 files added'
[master (root-commit) 0323e16] 2 files added
 2 files changed, 2 insertions(+)
 create mode 100644 file1.txt
 create mode 100644 file2.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashing (master)
$ vim file1.txt
First Line in File1
Work is going on...

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashing (master)
$ vim file2.txt

First Line in File2
Work is going on ...
```

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashing (master)
$ git add file2.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashing (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
      modified:   file2.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
      modified:   file1.txt
```

Assume we required to create and work on file3.txt and and this file changes needs to be committed immediately.

To work on file3.txt, we have to save uncommitted changes of file1.txt and file2.txt to some temporary location, because we don't want to include these changes in the current commit. For this we should go for git stash command.

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashing (master)
$ git stash
Saved working directory and index state WIP on master: 0323e16 2 files added

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashing (master)
$ git status
On branch master
nothing to commit, working tree clean

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashing (master)
$ cat file1.txt
First Line in File1

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashing (master)
$ cat file2.txt
First Line in File2
```

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}

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashing (master)
$ git show stash@{0}
commit f4777b1cf3f5d182a5a85b0041bcfc512f0c22a6 (refs/stash)
Merge: 0323e16 474e39d
Author: Ravi <durgasoftonlinetraining@gmail.com>
Date:   Wed Jun 10 20:47:46 2020 +0530

    WIP on master: 0323e16 2 files added

diff --cc file1.txt
index 5fd239d,5fd239d..2a0ac40
--- a/file1.txt
+++ b/file1.txt
@@@ -1,1 -1,1 +1,2 @@@
  First Line in File1
++Work is going on...

Now we can work on our urgent requirement..
---------------------------------------------------------------
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashing (master)
$ echo "Urgent work needs to be committed immediately"> file3.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashing (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file3.txt

nothing added to commit but untracked files present (use "git add" to track)

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashing (master)
$ git add file3.txt; git commit -m 'urgent work completed'
[master e92524a] urgent work completed
 1 file changed, 1 insertion(+)
 create mode 100644 file3.txt
```

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.

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashing (master)
$ git stash list
stash@{0}: WIP on master: 7548594 2 files added

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashing (master)
$ git stash pop stash@{0}
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file1.txt
        modified:   file2.txt

no changes added to commit (use "git add" and/or "git commit -a")
Dropped stash@{0} (775eb1616fe54fec28cd0e0a3a7b52fabba34d21)

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashing (master)
$ git stash list

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashing (master)
$ git status
On branch master
Changes not staged for commit:
```

```
 (use "git add <file>..." to update what will be committed)
 (use "git restore <file>..." to discard changes in working directory)
       modified:   file1.txt
       modified:   file2.txt

no changes added to commit (use "git add" and/or "git commit -a")
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashing (master)
$ cat file1.txt
First Line in File1
Work is going on...

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashing (master)
$ cat file2.txt
First Line in File2
Work is going on...

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashing (master)
$ git add file1.txt file2.txt ; git commit -m '2 files added'
[master 8bac6b1] 2 files added
 2 files changed, 2 insertions(+)

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashing (master)
$ echo "Some more work is going on" >> file1.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashing (master)
$ echo "Some more work is going on" >> file2.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashing (master)
$ git stash
Saved working directory and index state WIP on master: 8bac6b1 2 files added

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashing (master)
$ git stash list
stash@{0}: WIP on master: 8bac6b1 2 files added

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashing (master)
$ git stash apply stash@{0}
On branch master
Changes not staged for commit:
 (use "git add <file>..." to update what will be committed)
 (use "git restore <file>..." to discard changes in working directory)
       modified:   file1.txt
       modified:   file2.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashing (master)
$ git stash list
  It is empty
```

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
```

Demo Exaample for Partial Stash:

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects
$ mkdir partialstash

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects
$ cd partialstash

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/partialstash
$ git init
Initialized empty Git repository in D:/gitprojects/partialstash/.git/

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/partialstash (master)
$ echo 'First Line' > file1.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/partialstash (master)
$ echo 'First Line' > file2.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/partialstash (master)
$ git add .;git commit -m 'first commit'
[master (root-commit) 9484eab] first commit
 2 files changed, 2 insertions(+)
 create mode 100644 file1.txt
 create mode 100644 file2.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/partialstash (master)
$ echo "work is going on" >> file1.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/partialstash (master)
$ echo "work is going on" >> file2.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/partialstash (master)
$ git status
On branch master
```

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file1.txt
        modified:   file2.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Partial Stash of only file1.txt:

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/partialstash (master)
$ git stash -p
diff --git a/file1.txt b/file1.txt
index 603cb1b..0ac9556 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1 +1,2 @@
 First Line
+work is going on
(1/1) Stash this hunk [y,n,q,a,d,e,?]? y

diff --git a/file2.txt b/file2.txt
index 603cb1b..0ac9556 100644
--- a/file2.txt
+++ b/file2.txt
@@ -1 +1,2 @@
 First Line
+work is going on
(1/1) Stash this hunk [y,n,q,a,d,e,?]? n

Saved working directory and index state WIP on master: 9484eab first commit

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/partialstash (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file2.txt

no changes added to commit (use "git add" and/or "git commit -a")

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/partialstash (master)
$ git stash list
stash@{0}: WIP on master: 9484eab first commit
```

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/partialstash (master)
$ git show stash@{0}
commit 76459eecb1b350caf56349b94cb2f091a371a7d0 (refs/stash)
Merge: 9484eab 30f190a
Author: Ravi <durgasoftonlinetraining@gmail.com>
Date:   Thu Jul 16 20:32:05 2020 +0530

    WIP on master: 9484eab first commit

diff --cc file1.txt
index 603cb1b,603cb1b..0ac9556
--- a/file1.txt
+++ b/file1.txt
@@@ -1,1 -1,1 +1,2 @@@
  First Line
++work is going on

We can continue our urgent work

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/partialstash (master)
$ echo 'urgent work' > file3.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/partialstash (master)
$ git add .;git commit -m 'commit related to urgent work'
[master 5e040c2] commit related to urgent work
 2 files changed, 2 insertions(+)
 create mode 100644 file3.txt
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/partialstash (master)
$ git status
On branch master
nothing to commit, working tree clean

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/partialstash (master)
$ git stash pop stash@{0}
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file1.txt

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (76459eecb1b350caf56349b94cb2f091a371a7d0)
```

20.7) How to delete the stash:

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

Demo Example:

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects
$ mkdir stashdelete

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects
$ cd stashdelete

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashdelete
$ git init
Initialized empty Git repository in D:/gitprojects/stashdelete/.git/

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashdelete (master)
$ echo 'First Line'> file1.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashdelete (master)
$ echo 'First Line'> file2.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashdelete (master)
$ echo 'First Line'> file3.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashdelete (master)
$ git add .;git commit -m '3 files added'
[master (root-commit) df0ba71] 3 files added
 3 files changed, 3 insertions(+)
 create mode 100644 file1.txt
 create mode 100644 file2.txt
 create mode 100644 file3.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashdelete (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
```

```
        modified:   file1.txt

no changes added to commit (use "git add" and/or "git commit -a")

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashdelete (master)
$ git stash
Saved working directory and index state WIP on master: df0ba71 3 files added

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashdelete (master)
$ git stash list
stash@{0}: WIP on master: df0ba71 3 files added

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashdelete (master)
$ git show stash@{0}
commit 9632ddb69d4aa33e7b804a38f39032c3f657a461 (refs/stash)
Merge: df0ba71 2bd956b
Author: Ravi <durgasoftonlinetraining@gmail.com>
Date:   Thu Jul 16 20:42:44 2020 +0530

    WIP on master: df0ba71 3 files added

diff --cc file1.txt
index 603cb1b,603cb1b..b4660ea
--- a/file1.txt
+++ b/file1.txt
@@@ -1,1 -1,1 +1,2 @@@
  First Line
++Work is going on

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashdelete (master)
$ git status
On branch master
nothing to commit, working tree clean

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashdelete (master)
$ echo 'urgent work is going on file2' >> file2.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashdelete (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file2.txt
```

```
no changes added to commit (use "git add" and/or "git commit -a")

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashdelete (master)
$ git stash
Saved working directory and index state WIP on master: df0ba71 3 files added

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashdelete (master)
$ git status
On branch master
nothing to commit, working tree clean

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashdelete (master)
$ git stash list
stash@{0}: WIP on master: df0ba71 3 files added
stash@{1}: WIP on master: df0ba71 3 files added

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashdelete (master)
$ echo 'more urgent work on file3' >> file3.txt

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashdelete (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file3.txt

no changes added to commit (use "git add" and/or "git commit -a")

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashdelete (master)
$ git stash
warning: LF will be replaced by CRLF in file3.txt.
The file will have its original line endings in your working directory
Saved working directory and index state WIP on master: df0ba71 3 files added

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashdelete (master)
$ git stash list
stash@{0}: WIP on master: df0ba71 3 files added
stash@{1}: WIP on master: df0ba71 3 files added
stash@{2}: WIP on master: df0ba71 3 files added

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashdelete (master)
$ git stash drop stash@{1}
Dropped stash@{1} (33f3c2b23ae7f4b9df5979d898b5fbbe7b93e272)
```

```
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashdelete (master)
$ git stash list
stash@{0}: WIP on master: df0ba71 3 files added
stash@{1}: WIP on master: df0ba71 3 files added

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashdelete (master)
$ git stash clear

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashdelete (master)
$ git stash list
```

FAQs:

1) What is stash?
2) How to perform stash?
3) How to perform unstash?
4) What is the Difference between pop and apply?
5) what is partial stash?
6) How to delete all stashes?
7) How to delete a particular stash?