

# VERSION CONTROL



git

By; Ahmed EL-Mahdy

LNKD: [bitlylink.com/ooLxn](https://bitlylink.com/ooLxn)

FB: [bit.ly/2Utbjkz](https://bit.ly/2Utbjkz)

# PART 1

---

- What is Git?
- Why is Git used?
- Git Lifecycle
- Git Installation
- Environment setup
- Common Git Commands-Local and Remote

# WHAT IS GIT

---

**Git** helps to keep track of the history of codes files by storing them in versions on its own server repository

EXAMPLE: GitHub.

**Git** has:

- Functionality
- Performance
- security
- flexibility

# WHY GIT

---

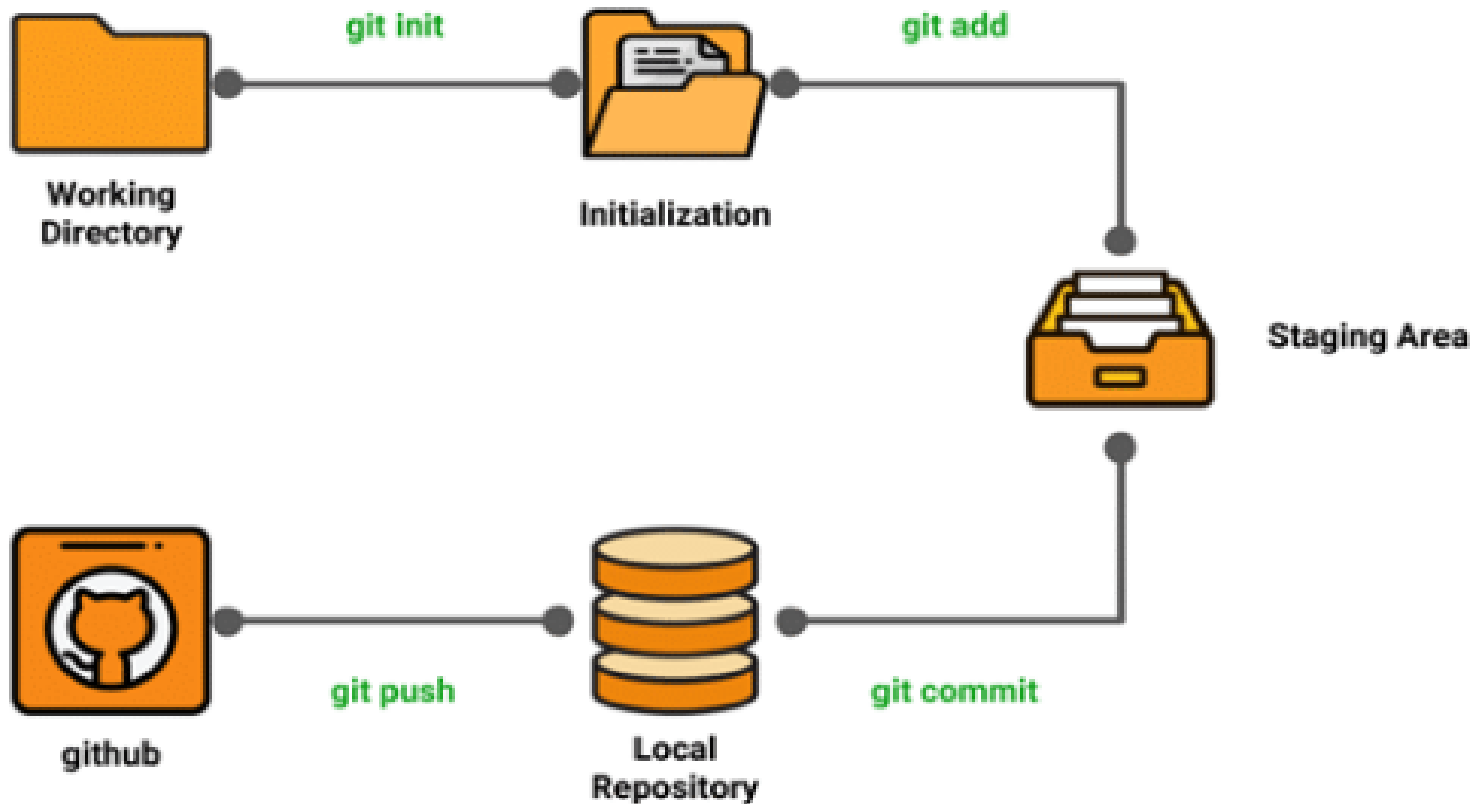
**Git** is so popular for those reasons:

- Work offline
- Undo your mistakes
- Restore deleted commits
- Secure
- Performance
- Flexibility

# GIT LIFE CYCLE

---

- Local working directory



# GIT LIFE CICLE

---

- Git lifecycle
  - Initialization
  - Staging area
  - Commit

# INSTALL GIT

---

## Linux :

- Debain and Ubuntu  
`sudo apt-get install git`
- CentOS  
`sudo yum install git`
- Fedora  
`sudo yum install git-core`
- Arch Linux  
`sudo pacman -Sy git`
- Gentoo  
`sudo emerge --ask --verbose dev-vcs/git`

## Windows:

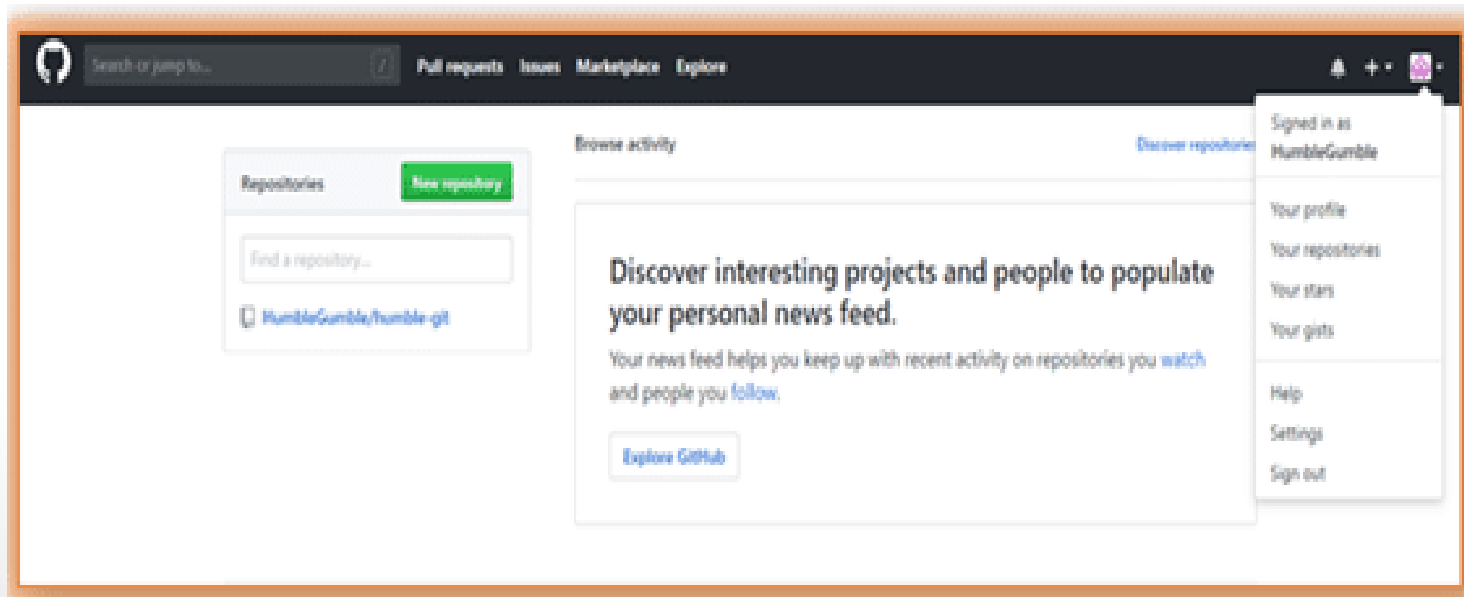
<https://bit.ly/2OSPzzb>

# ENVIRONMENT SETUP

---

## Create a GitHub account

- Go to <https://Github.com>
- Create one GitHub account
- Login






# ENVIRONMENT SETUP

---

## Configuring Git

To tell Git who you are, run the following to commands

A terminal window with a black background and orange border. It shows two commands being executed in a Windows environment (MINGW64). The first command sets the global user email to 'debashis.intellipaat@gmail.com'. The second command sets the global user name to 'HumbleGumble'.

```
INTELLIPAAT@DESKTOP-186LH03 MINGW64 /c/git/ProjectGit (master)
$ git config --global user.email "debashis.intellipaat@gmail.com"

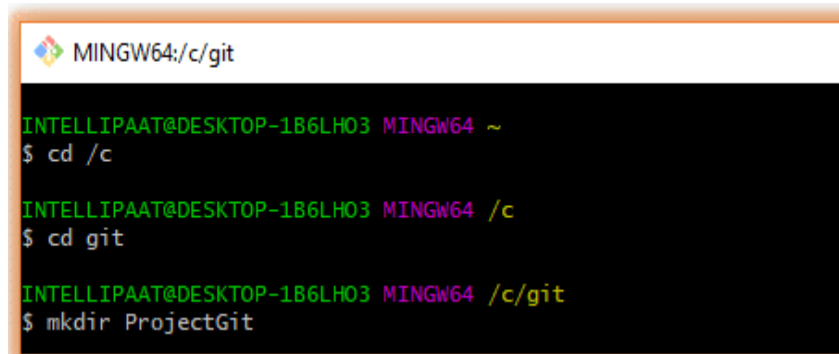
INTELLIPAAT@DESKTOP-186LH03 MINGW64 /c/git/ProjectGit (master)
$ git config --global user.name "HumbleGumble"
```

# ENVIRONMENT SETUP

---

## Creating a Local Repository

- Open a terminal and move to wanted place for the project on the local machine using `cd` (change directory)
- Example

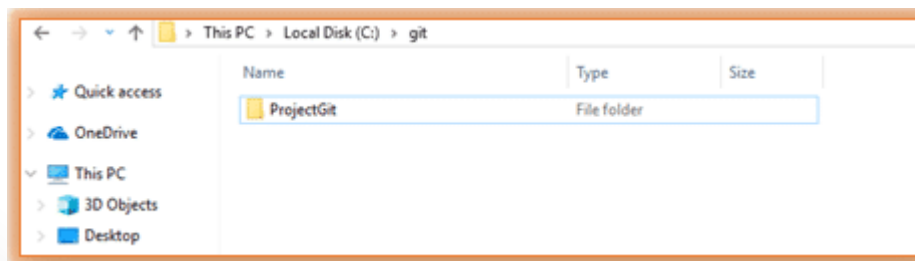


```
MINGW64:/c/git

INTELLIPAAT@DESKTOP-1B6LH03 MINGW64 ~
$ cd /c

INTELLIPAAT@DESKTOP-1B6LH03 MINGW64 /c
$ cd git

INTELLIPAAT@DESKTOP-1B6LH03 MINGW64 /c/git
$ mkdir ProjectGit
```



# GIT COMMANDS

## LOCAL COMMENDS

---

### Git init [repository name]

- Navigate to project directory
- type the command **git init** to initialize a Git repository for local project folder.
- Git will create a hidden **.git** directory and use it for keeping its files organized in other subdirectories.

```
intellipaat@DESKTOP-SPC6JQB MINGW64 ~/ProjectGit1
$ cd

intellipaat@DESKTOP-SPC6JQB MINGW64 ~
$ cd ProjectGit1

intellipaat@DESKTOP-SPC6JQB MINGW64 ~/ProjectGit1
$ git init
Initialized empty Git repository in C:/Users/intellipaat/ProjectGit1/.git/

intellipaat@DESKTOP-SPC6JQB MINGW64 ~/ProjectGit1 (master)
$ |
```

# GIT COMMANDS

## LOCAL COMMENDS

---

### Git touch

To add files to a Project

- Create a new file with command touch
- Will see the files present in the master branch

```
INTELLIPAAT@DESKTOP-1B6LHO3 MINGW64 /c/git/ProjectGit (master)
$ touch humble.txt

INTELLIPAAT@DESKTOP-1B6LHO3 MINGW64 /c/git/ProjectGit (master)
$ ls
humble.txt
```

# GIT COMMANDS

## LOCAL COMMENDS

---

### Git status

After creating the new file, you can use the Git status command and see the status of the files in the master branch.

```
INTELLIPAAT@DESKTOP-1B6LHO3 MINGW64 /c/git/ProjectGit (master)
$ git status
On branch master

No commits yet

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

        humble.txt

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

# GIT COMMANDS

## LOCAL COMMENDS

---

### Git add [file(s) name]

This will add the specified file(s) into the Git repository

*i.e.*, into the staging area where they are already being tracked by Git and now ready to be committed.

```
intellipa@DESKTOP-SPC6JQB MINGW64 ~/ProjectGit1 (master)
$ git add 234master.txt
warning: LF will be replaced by CRLF in 234master.txt.
The file will have its original line endings in your working directory.
```

### git add . , git add \* or git add -A

This will take all our files into the Git repository

*i.e.*, into the staging area.

```
intellipa@DESKTOP-SPC6JQB MINGW64 ~/ProjectGit1 (master)
$ git add .
warning: LF will be replaced by CRLF in 234.txt.
The file will have its original line endings in your working directory.
```

# GIT COMMANDS

## LOCAL COMMENDS

---

### Git status

This command will show the modified status of an existing file and the file addition status of a new file, if any, that have to be committed.

```
intellipaate@DESKTOP-SPC6JQB MINGW64 ~/ProjectGit1 (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   234.txt
        new file:   ls
```

# GIT COMMANDS

## LOCAL COMMENDS

---

### Git commit-m "message"

This command records or snapshots the file permanently in the version history.

All the files which are there in the directory right now are being saved in the git file system.

```
intellipaat@DESKTOP-SPC6JQB MINGW64 ~/ProjectGit1 (master)
$ git commit -m "Committing 234master.txt in master"
[master (root-commit) be73fc3] Committing 234master.txt in master
2 files changed, 2 insertions(+)
create mode 100644 123master.txt
create mode 100644 234master.txt
```



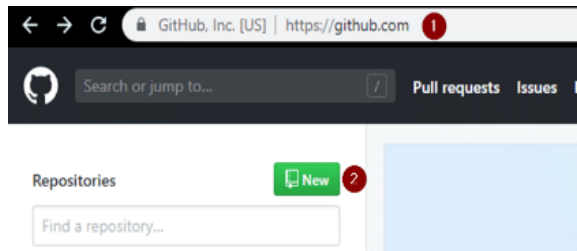
# GIT COMMANDS

## REMOTE COMMENDS

Once everything is ready on our local system, we can start pushing the code to the remote (central) repository of the project.

### Create new Repo

1. Go to GitHub account
2. New
3. Create a new repository



### Create a new repository

A repository contains all the files for your project, including the revision history.

Owner

HumbleGumble ▾

Repository name

humbleRepo ✓

Great repository names are short and memorable. Need inspiration? How about curly-fiesta.

Description (optional)

My new repository



Public

Anyone can see this repository. You choose who can commit.



Private

You choose who can see and commit to this repository.

☐ Initialize this repository with a README

This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None ▾

Add a license: None ▾



Create repository

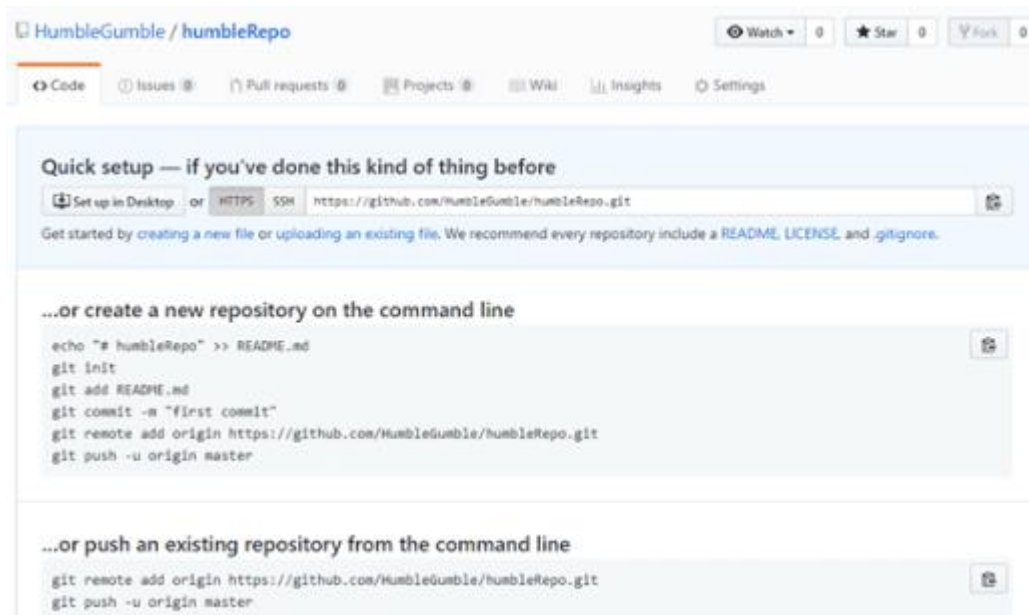
# GIT COMMANDS

## REMOTE COMMENDS

---

### Create new Repo

land on



# GIT COMMANDS

## REMOTE COMMENDS

---

### **Git remote add origin “[URL]”**

Operate remote commands in the repository

(copy repo url and paste as below)

```
intellipaat@DESKTOP-SPC6JQB MINGW64 ~/ProjectGit1 (master)
$ git remote add origin https://github.com/prabhpreetk/humbleRepo1.git
```

# GIT COMMANDS

## REMOTE COMMENDS

---

### **git push origin [branch name]**

By using the command 'git push', the local repository's files will be synced with the remote repository on Github.

(in case changes in the files need to push to remote repo)

```
intellipaat@DESKTOP-SPC6JQB MINGW64 ~/ProjectGit1 (branch1)
$ git push origin branch1
fatal: HttpRequestException encountered.
  An error occurred while sending the request.
Username for 'https://github.com': prabhpreetk
Counting objects: 7, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (7/7), 729 bytes | 0 bytes/s, done.
Total 7 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), done.
remote:
remote: Create a pull request for 'branch1' on GitHub by visiting:
remote:   https://github.com/prabhpreetk/humbleRepo1/pull/new/branch1
remote:
To https://github.com/prabhpreetk/humbleRepo1.git
 * [new branch]      branch1 -> branch1
```

# GIT COMMANDS

## REMOTE COMMENDS

---

### **git pull**

When it comes to syncing remote repository pull command comes in handy.

Let us take a look at the steps that leads to pulling operation in Git.

- remote origin
- pull master

```
INTELLIPAAT@DESKTOP-1B6LHD3 MINGW64 /c/git/ProjectGit (master)
$ git remote set-url origin "https://github.com/HumbleGumble/humbleRepo.git"

INTELLIPAAT@DESKTOP-1B6LHD3 MINGW64 /c/git/ProjectGit (master)
$ git pull origin master
From https://github.com/HumbleGumble/humbleRepo
 * branch            master      -> FETCH_HEAD
+ 5ad410f...169cfba master      -> origin/master (forced update)
Already up to date.
```

# GIT COMMANDS

## REMOTE COMMENDS

---

**How to work on exist project !**

# GIT COMMANDS

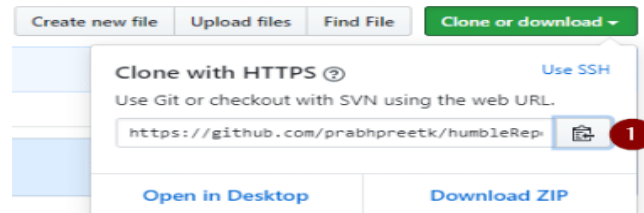
## REMOTE COMMENDS

---

### **git clone [url]**

It imports the files of project from the remote repository to the local system.

- Click Clone or Download.
- Copy the link and paste it on the terminal with git clone command.



- Create a local folder  
`mkdir [directory- name]`  
`cd [directory- name]`  
`git clone [URL]`

# GIT COMMANDS

## REMOTE COMMENDS

---

`git clone [url]`

```
intellipaate@DESKTOP-SPC6JQB MINGW64 ~/test
$ git clone https://github.com/prabhpreetk/humbleRepo1.git
Cloning into 'humbleRepo1'...
```

**Note:** Don't use the git remote add origin command because now if you push any new file , it knows where it has to go .



# GIT COMMEND

---

## WRAPUP!

# WRAPUP GIT COMMEND

---

## **Get Documentation For a Command**

`$ man git-log`

OR

`$ git help log`

## **introduce yourself to Git with your name and public email address**

`$ git config --global user.name "Your Name Comes Here"`

`$ git config --global user.email you@yourdomain.example.com`

# WRAPUP GIT COMMEND

---

## Importing a new project

Assume you have a tarball `project.tar.gz` with your initial work. You can place it under Git revision control as follows.

```
$ tar xzf project.tar.gz
```

```
$ cd project
```

```
$ git init
```

Git will reply

Initialized empty Git repository in `.git/`

You've now **initialized** the working directory—you may notice a new directory created, named `".git"`.

Next, tell Git to take a snapshot of the contents of all files under the current directory (note the `.`), with **git add** :

```
$ git add .
```

This snapshot is now stored in a temporary staging area which Git calls the `"index"`. You can permanently store the contents of the index in the repository with **git commit**:

```
$ git commit
```

This will prompt you for a commit message. You've now stored the first version of your project in Git.

# WRAPUP GIT COMMEND

---

## Making changes

Modify some files, then add their updated contents to the index:

```
$ git add file1 file2 file3
```

You are now ready to commit. You can see what is about to be committed using git diff with the --cached option:

```
$ git diff --cached
```

(Without --cached, git diff will show you any changes that you've made but not yet added to the index.) You can also get a brief summary of the situation with git status:

```
$ git status
```

On branch master

Changes to be committed:

Your branch is up to date with 'origin/master'.

(use "git restore --staged <file>..." to unstage)

```
modified: file1
```

```
modified: file2
```

```
modified: file3
```

# WRAPUP GIT COMMEND

---

## Making changes

If you need to make any further adjustments, do so now, and then add any newly modified content to the index. Finally, commit your changes with:

```
$ git commit
```

This will again prompt you for a message describing the change, and then record a new version of the project.

instead of running git add beforehand, you can use

```
$ git commit -a
```

which will automatically notice any modified (but not new) files, add them to the index, and commit, all in one step.

## Git tracks content not files

Many revision control systems provide an **add** command that tells the system to start tracking changes to a new file. Git's **add** command does something simpler and more powerful: git add is used both for new and newly modified files, and in both cases it takes a snapshot of the given files and stages that content in the index, ready for inclusion in the next commit.

# WRAPUP GIT COMMEND

---

## Viewing project history

At any point you can view the history of your changes using

```
$ git log
```

If you also want to see complete diffs at each step, use

```
$ git log -p
```

Often the overview of the change is useful to get a feel of each step

```
$ git log --stat --summary
```

## Using Git for collaboration

Suppose that Alice has started a new project with a Git repository in `/home/alice/project`, and that Bob, who has a home directory on the same machine, wants to contribute.

Bob begins with:

```
bob$ git clone /home/alice/project myrepo
```

This creates a new directory "myrepo" containing a clone of Alice's repository. The clone is on an equal footing with the original project, possessing its own copy of the original project's history.

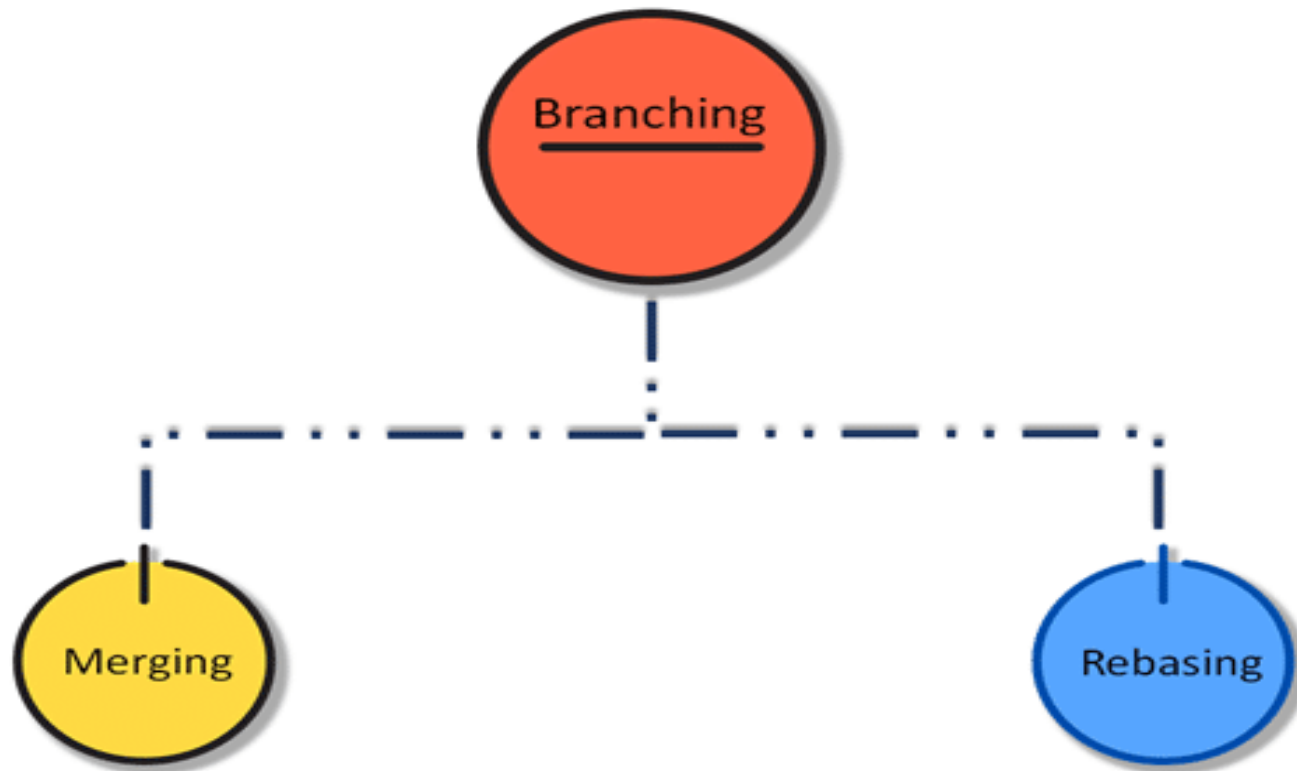
# PART 2

---

- Branching
- Merging
- Rebasing “self study”
- Merging Conflicts
- Rebasing Conflicts “self study”
- Solving Merging Conflicts
- Git Workflows

# BRANCHING

---





# BRANCHING

---

If any modify needed with out change in the main project take a branch out of the master branch.

It handles the workspace of multiple developers

**Git checkout -b [name-of-the-new-branch]**

**Git branch [name-of-the-branch]**

To create a new branch to add a feature in the main project

```
INTELLIPAAT@DESKTOP-1B6LHO3 MINGW64 /c/git/ProjectGit (master)
$ git checkout -b branch1
Switched to a new branch 'branch1'

INTELLIPAAT@DESKTOP-1B6LHO3 MINGW64 /c/git/ProjectGit (branch1)
$ git branch
* branch1
  master
```

# BRANCHING

---

**Get branch -D [name –of –the- branch]**

To delete a branch

**Note:** Now, you must be wondering how to change branch in git to master branch right? Use the following command:

**git checkout master**

# MERGING

---

The merge commit represents every change that has occurred on feature since it branched from master.

**Note:** Even after merging we can go on with our work on both the master and feature branch independently.

Create a new branch called branch2

```
INTELLIPAAT@DESKTOP-1B6LHO3 MINGW64 /c/git/ProjectGit (branch1)
$ git checkout -b branch2
Switched to a new branch 'branch2'
```

create a new file in that branch

```
INTELLIPAAT@DESKTOP-1B6LHO3 MINGW64 /c/git/ProjectGit (branch2)
$ touch newFile.txt
```

# MERGING

---

Add changes from all tracked and untracked files

*Note: refer the following command **Git add** attributes*

```
-n, --dry-run      dry run
-v, --verbose      be verbose

-i, --interactive  interactive picking
-p, --patch        select hunks interactively
-e, --edit         edit current diff and apply
-f, --force        allow adding otherwise ignored files
-u, --update       update tracked files
--renormalize      renormalize EOL of tracked files (implies -u)
-N, --intent-to-add record only the fact that the path will be added later
-A, --all          add changes from all tracked and untracked files
--ignore-removal   ignore paths removed in the working tree (same as --no
11)
--refresh          don't add, only refresh the index
--ignore-errors    just skip files which cannot be added because of error

--ignore-missing   check if - even missing - files are ignored in dry run
--chmod (+|-)x    override the executable bit of the listed files
```

Git add -A

```
INTELLIPAAT@DESKTOP-1B6LHO3 MINGW64 /c/git/ProjectGit (branch2)
$ git add -A

INTELLIPAAT@DESKTOP-1B6LHO3 MINGW64 /c/git/ProjectGit (branch2)
$ git commit -m "Added a file in branch2"
[branch2 59ce99d] Added a file in branch2
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 newFile.txt
```

# MERGING

---

## Git log

used to check the log for every commit detail in the repo

It show the log of the current branch

## Git log-3

Check the last 3 log

```
intellipaat@DESKTOP-SPC6JQB MINGW64 ~/ProjectGit1 (master)
$ git log -3
commit be73fc3e802f8afece5a9f12cea4415665e36bf4 (HEAD -> master, origin/master)
Author: prabhpreetk <prabhpreet.intellipaat@gmail.com>
Date:   Wed Jun 19 14:56:33 2019 +0530

    Committing 234master.txt in master

intellipaat@DESKTOP-SPC6JQB MINGW64 ~/ProjectGit1 (master)
$ git checkout branch1
Switched to branch 'branch1'

intellipaat@DESKTOP-SPC6JQB MINGW64 ~/ProjectGit1 (branch1)
$ git log -3
commit deae5df00b52e75abe175f9f5bdcfde84feb6dd8 (HEAD -> branch1, origin/branch1)
Author: prabhpreetk <prabhpreet.intellipaat@gmail.com>
Date:   Wed Jun 19 15:43:54 2019 +0530

    123master.txt file modified from feature branch

commit bbf434bc2eceaca5d1742664638a9bd05630636d
Author: prabhpreetk <prabhpreet.intellipaat@gmail.com>
Date:   Wed Jun 19 15:41:09 2019 +0530

    123branch1.txt filein feature branch; 1st commit in feature branch

commit be73fc3e802f8afece5a9f12cea4415665e36bf4 (origin/master, master)
Author: prabhpreetk <prabhpreet.intellipaat@gmail.com>
Date:   Wed Jun 19 14:56:33 2019 +0530

    Committing 234master.txt in master
```

# MERGING

---

## Get fetch

- git gathers any commits from the target branch that don't exist in the current branch stores them in our local repository.
- it does not merge them with the current branch.

```
intellipaata@DESKTOP-SPC6JQ8 MINGW64 ~/ProjectGit1 (branch1)
$ git fetch --all
Fetching origin
```

- This is particularly useful when we need to keep our repository up to date but are working on something that might break if we update our files. To integrate the commits into our master branch, we use merge. It fetches all of the branches from the repository. It also downloads all the required commits and files from the other repository.

# MERGING

---

## Git pull

### Git pull origin master

The git pull command first runs 'git fetch' which downloads the content from the specified remote repository and then immediately updates the local repo to match the content.

```
intellipaate@DESKTOP-SPC6JQB MINGW64 ~/ProjectGit1 (master)
$ git pull origin master
From https://github.com/prabhpreetk/humblerepo1
* branch      master      -> FETCH_HEAD
Already up-to-date.
```

# MERGING

---

## Git checkout master

To get inside the master branch

## Git merge branch2

To perform merging

```
INTELLIPAAT@DESKTOP-186LH03 MINGW64 /c/git/ProjectGit (branch2)
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.

INTELLIPAAT@DESKTOP-186LH03 MINGW64 /c/git/ProjectGit (master)
$ git merge branch2
Updating 169cfba..59ce99d
Fast-forward
 newFile.txt | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 newFile.txt
```



# MERGING

---

## Git merge

### Git merge [another-file-name]

- This command will combine multiple sequences of commits into one unified history.
- In the most frequent use cases, git merge is used to combine two branches.
- The git merge command takes two commit pointers, usually the branch tips, and finds a common base commit between them.
- Once Git finds a common base commit, it will create a commit sequence.

```
intellipaat@DESKTOP-SPC6JQB MINGW64 ~/ProjectGit1 (branch1)
$ git checkout master
switched to branch 'master'

intellipaat@DESKTOP-SPC6JQB MINGW64 ~/ProjectGit1 (master)
$ git merge branch1
updating be73fc3..f538c12
Fast-forward
 123branch1.txt | 1 +
 123master.txt  | 2 ++
 1s             | 0
3 files changed, 3 insertions(+)
create mode 100644 123branch1.txt
create mode 100644 1s
```

# MERGING

---

**Git log -graph**

**Git log -graph**

**Git log -graph --pretty=oneline**

**Selfstudy**

# MERGING

---

## Git stash

### Git stash

To save work without staging or committing the code to git repo and want to switch between branches

```
intellipa@DESKTOP-SPC6JQ8 MINGW64 ~/ProjectGit1 (master)
$ git checkout branch1
Switched to branch 'branch1'

intellipa@DESKTOP-SPC6JQ8 MINGW64 ~/ProjectGit1 (branch1)
$ ls
123branch1.txt 123branch2.txt 123master.txt 234master.txt
$ ls
123branch1.txt 123branch2.txt 123master.txt 234master.txt

intellipa@DESKTOP-SPC6JQ8 MINGW64 ~/ProjectGit1 (branch1)
$ echo "123master.txt is getting modified, will be stashed!">>123master.txt

intellipa@DESKTOP-SPC6JQ8 MINGW64 ~/ProjectGit1 (branch1)
$ git stash
warning: LF will be replaced by CRLF in 123master.txt.
The file will have its original line endings in your working directory.
saved working directory and index state WIP on branch1: deae5df 123master.txt file modified fr
```

# MERGING

---

## Git diff

### Git diff [commit-id-of-version-x] [commit-id-of-version-y]

- Diffing is a function that takes two input datasets and outputs the changes between them.
- It's multi-use Git command which when executed runs a diff function on Git data sources.
- These data sources can be commits, branches, files, and more.
- It's used along with git status and git log commands to analyze the current state of a Git repository.
- Use **git log** to get the details of commit IDs.

```
intellipaat@DESKTOP-5PC6JQB MINGW64 ~/ProjectGit1 (master)
$ git diff f538c123a0f0900d717db8f342f690d9304eee07
diff --git a/123master.txt b/123master.txt
index c656711..f6ffadf 100644
--- a/123master.txt
+++ b/123master.txt
@@ -1,3 +1,2 @@
 123.txt file in master branch; 1st commit in master
 123.txt file modified from feature branch
-123master.txt is getting modified, will be stashed!
```

# REBASEING

---

**Advantage of rebasing**

**Disadvantage of rebasing**

**Self study**

# GIT MERG CONFLICT

---

## Simple example

<pre>main() {     program statements     calling function 1 }  Function1() {     statements } Function2() {     statements }</pre> <p>Developer A</p>	<pre>main() {     program statements     calling function 1 }  Function1() {     statements } Function3() {     statements }</pre> <p>Developer B</p>
-------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------

Developer-A added a function called function2 in our main code.

Developer-B added a different function called function3 to the same code file.

How to merge these two modifications now? That is where our merging conflict occur.

# SOLVING CONFLICTS

---

## **Solving merging conflicts:**

- Manually resolve it in merging tool.
- File locking doesn't allow different developers to work on the same piece of code simultaneously. It helps to avoid merge conflicts, but slows down development.

**rebasing conflicts** “Self study”

# ROLL BACK

---

## Git reset

**Git reset --hard[SOME-COMMIT]**

return the *entire* working tree to the last committed state

```
intellipaat@DESKTOP-SPC6JQB MINGW64 ~/ProjectGit1 (master)
$ git reset --hard
HEAD is now at 03aaec4 Committing chnges

intellipaat@DESKTOP-SPC6JQB MINGW64 ~/ProjectGit1 (master)
$ git log --graph --pretty=oneline
* 03aaec45355ea6fc9a94895a02d4e32f2c7918d9 (HEAD -> master) committing chnges
* f538c123a0f0900d717db8f342f690d9304eee07 (branch1) 123master.txt file is being committed after
* deae5df00b52e75abe175f9f5bdcfde84feb6dd8 (origin/branch1) 123master.txt file modified from f
* bbf434bc2eceaca5d1742664638a9bd05630636d 123branch1.txt file in feature branch; 1st commit in
* be73fc3e802f8afece5a9f12cea4415665e36bf4 (origin/master) Committing 234master.txt in master
```

## Git will:

- Make our current branch (typically master) back to point <SOME-COMMIT>
- Then, make the files in our working tree and the index (“staging area”) the same as the versions committed in <SOME-COMMIT>



# ROLL BACK

---

## Git revert [commit id]

- It's an 'undo' type command.
- It figures out how to invert the changes introduced by the commit and appends a new commit with the resulting inverse content.

```
intellipaat@DESKTOP-SPC6JQ8 MINGW64 ~/ProjectGit2 (master)
* 3fe7411e9309c2a5bf8445211cb094b141e2cb86 (HEAD -> master) Revert "Master humble.txt"
* 788f1b68a399f02ef3069637a942e15e435f51c9 Master humble.txt
* 32e3b572663e41daa2708bd7adac96fe5bba165f Humble.txt
* d9ca5e44e6363e82c803eb032d24021bf8a0e1b6 Humble Gumble3
* dc82b9972a06d87b0d17c174365b710fd05653f3 Humble Gumble2
* 22ec55c68c3bc5e71ecf7c45766d53902715195d Committing Stash
* df25c6aabde6c4486caef8aea6444795cd5d034c Adding file in newFile
* b161ec560c79c9cd2c25df15ff8d65f8bf4cb041 (origin/master, branch1) What is humble gumble?
```

# Enabling required reviews for pull requests

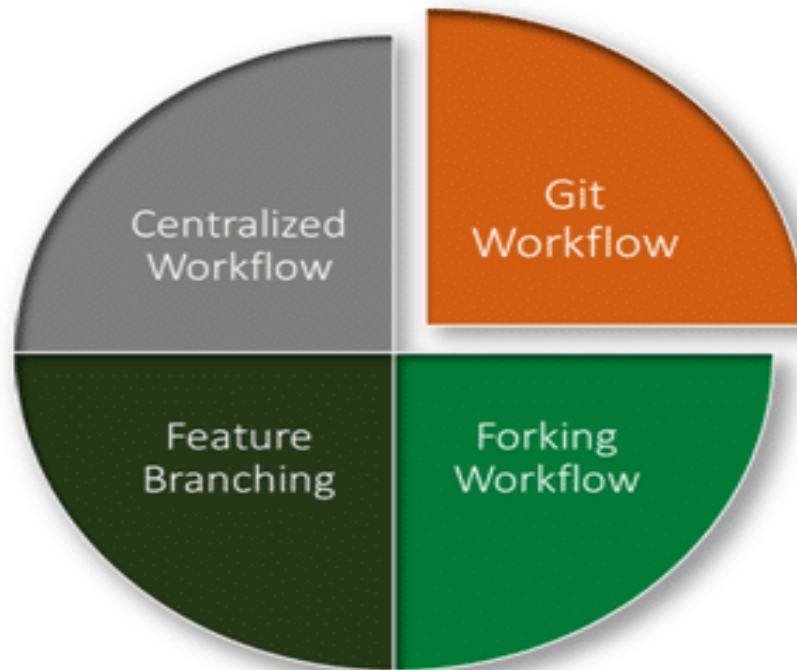
---

**Self study**

# GIT WORKFLOWS

---

Depending upon the team size, choosing the right Git workflow is important for a team project to increase the productivity.



# GIT WORKFLOWS

---

## Centralized workflow:

- Only **master** branch is there.
- All changes are committed into the master branch itself.

## Feature Branching Workflow:

- feature development takes place only in a dedicated feature branch.

## Git Workflow:

- In this workflow uses two branches instead of a single master branch.
- master branch stores the official release history.
- develop branch acts as an integration branch for features.

## Forking Workflow:

- contributor has two Git repositories: one private local and one on public server-side.

# MERG, REBAS, FETSH & PULL

---

## WRAPUP!

# FETCH, MERGE & PULL

---

**git fetch** and **git merge origin/master** will fetch & integrate remote changes.

Let's explain a common scenario:

origin/master is at C. Someone pushed D. You worked on E & F. Note that you will not see D in your local repository until you run **git fetch**.

origin/master

v

A-B-C-E-F < master

\

(D) < master on remote

Now you run **git fetch**. Now you can see D, and origin/master is updated to match the remote repository that it's tracking.

A-B-C-E-F < master

\

D < origin/master, master on remote

# FETCH, MERGE & PULL

---

Now you run **git merge**, giving you this:

A-B-C-E-F

\ \

D---G < master

^

origin/master, master on remote

now you've integrated your changes on master (E, F) with the new commits on origin/master (D).

**git pull** is simply a shortcut for the above steps.

# FETCH, MERGE & PULL

---

## **git merge without fetching**

Running **git merge origin/master** without the **git fetch** is pointless.

Without a **git fetch**, your local repository is unaware of any potential changes on the remote repository and origin/master will not have moved.

So you're at this state, where D is only on the remote and not present locally:

origin/master

v

A-B-C-E-F < master

\

(D) < master on remote

Since your local repository does not have D, a **git merge origin/master** will simply yield:

Already up-to-date.

Because, as far as your local repository is concerned, master already has everything in origin/master.



# FETCH, MERGE & PULL

---

## What's best?

None of the above. :)

`git fetch`

`git rebase origin/master master`

or a shortcut, **git pull -r**,

Prefered to see the changes before I rebase.

This will replay your changes on master (E, F) on top of origin/master (D) without a yucky merge commit. It yields:

A-B-C-D-E'-F' < master

^

origin/master, master on remote

**Note** how everything is in a single line, you're ready to push, and the history doesn't look like a friendship bracelet

# FETCH, MERGE & PULL

---

## One warning –

- Never rebase any commits that have already been pushed. Note that E & F became E' & F' after rebasing.
- The commits are entirely rewritten, with a new SHA and everything.
- If you rebase commits that are already public, developers will have their history re-written for them when they pull. And that's awful, and everyone will give you evil eyes and shun you.

# COMMANDS

---

## **Creat new branch**

Git -b (name of branch)

## **Creat new branch and switch it**

Git checkout -b (name of branch)

## **Delete branch**

Git branch -D [name -of -the- branch]

# COMMANDES

---

## Managing branches

A single Git repository can maintain multiple branches of development. To create a new branch named "experimental", use

```
$ git branch experimental
```

If you now run

```
$ git branch
```

you'll get a list of all existing branches:

```
experimental
```

```
* master
```

# COMMANDES

---

The "experimental" branch is the one you just created, and the "master" branch is a default branch that was created for you automatically. The asterisk marks the branch you are currently on; type

```
$ git switch experimental
```

to switch to the experimental branch. Now edit a file, commit the change, and switch back to the master branch:

```
(edit file)
```

```
$ git commit -a
```

```
$ git switch master
```

# COMMANDES

---

Check that the change you made is no longer visible, since it was made on the experimental branch and you're back on the master branch.

You can make a different change on the master branch:

(edit file)

```
$ git commit -a
```

at this point the two branches have diverged, with different changes made in each. To merge the changes made in experimental into master, run

```
$ git merge experimental
```

# COMMANDES

---

If the changes don't conflict, you're done. If there are conflicts, markers will be left in the problematic files showing the conflict;

```
$ git diff
```

will show this. Once you've edited the files to resolve the conflicts,

```
$ git commit -a
```

will commit the result of the merge. Finally,

At this point you could delete the experimental branch with

```
$ git branch -d experimental
```

This command ensures that the changes in the experimental branch are already in the current branch.

If you develop on a branch crazy-idea, then regret it, you can always delete the branch with

```
$ git branch -D crazy-idea
```

# COMMANDES

---

## Exploring history

Git history is represented as a series of interrelated commits. We have already seen that the git log command can list those commits. Note that first line of each git log entry also gives a name for the commit:

```
$ git log
commit c82a22c39cbc32576f64f5c6b3f24b99ea8149c7
Author: Junio C Hamano <junkio@cox.net>
Date: Tue May 16 17:18:22 2006 -0700
```

```
merge-base: Clarify the comments on post processing.
```

```
,
```

We can give this name to git show to see the details about this commit.

```
$ git show c82a22c39cbc32576f64f5c6b3f24b99ea8149c7
```

Get last 3 comment

```
$ git log - 3
```



# COMMANDS

---

But there are other ways to refer to commits. You can use any initial part of the name that is long enough to uniquely identify the commit:

\$ git show c82a22c39c	# the first few characters of the name are # usually enough
\$ git show HEAD	# the tip of the current branch
\$ git show experimental	# the tip of the "experimental" branch

Every commit usually has one "parent" commit which points to the previous state of the project:

\$ git show HEAD^	# to see the parent of HEAD
\$ git show HEAD^^	# to see the grandparent of HEAD
\$ git show HEAD~4	# to see the great-great grandparent of HEAD

# COMMANDS

---

Note that merge commits may have more than one parent:

```
$ git show HEAD^1 # show the first parent of HEAD (same as HEAD^)
```

```
$ git show HEAD^2 # show the second parent of HEAD
```

You can also give commits names of your own; after running

```
$ git tag v2.5 1b2e1d63ff
```

you can refer to 1b2e1d63ff by the name "v2.5". If you intend to share this name with other people (for example, to identify a release version), you should create a "tag" object, and perhaps sign it; see `git-tag[1]` for details.

# COMMANDS

---

Any Git command that needs to know a commit can take any of these names. For example:

```
$ git diff v2.5 HEAD    # compare the current HEAD to v2.5
```

```
$ git branch stable v2.5 # start a new branch named "stable" based  
                        # at v2.5
```

```
$ git reset --hard HEAD^ # reset your current branch and working  
                        # directory to its state at HEAD^
```

The git grep command can search for strings in any version of your project, so

```
$ git grep "hello" v2.5
```

searches for all occurrences of "hello" in v2.5.

If you leave out the commit name, git grep will search any of the files it manages in your current directory. So

```
$ git grep "hello"
```

is a quick way to search just the files that are tracked by Git.

```
$ git revert
```

Git revert comment id

Return back as undo ctrl+z

# COMMANDS

---

Many Git commands also take sets of commits, which can be specified in a number of ways. Here are some examples with git log:

```
$ git log v2.5..v2.6      # commits between v2.5 and v2.6
```

```
$ git log v2.5..          # commits since v2.5
```

```
$ git log --since="2 weeks ago" # commits from the last 2 weeks
```

```
$ git log v2.5.. Makefile  # commits since v2.5 which modify  
                           # Makefile
```

You can also give git log a "range" of commits where the first is not necessarily an ancestor of the second; for example, if the tips of the branches "stable" and "master" diverged from a common commit some time ago, then

```
$ git log stable..master
```

will list commits made in the master branch but not in the stable branch, while

```
$ git log master..stable
```

will show the list of commits made on the stable branch but not the master branch.

# COMMANDS

---

most commands that take filenames will optionally allow you to precede any filename by a commit, to specify a particular version of the file:

```
$ git diff v2.5:Makefile HEAD:Makefile.in
```

You can also use `git show` to see any such file:

```
$ git show v2.5:Makefile
```

# TOOLS

---

There is a plenty of tools that can solve conflicts with GUI, to show all possible files and select what you want to keep or remove.

One of these tools is **KDIFF3** (<http://kdiff3.sourceforge.net/>)

## Link KDIFF3 mergetool with GIT

You have to add the path of your tool to the global GIT configuration

```
> git config --global --add merge.tool kdiff3
> git config --global --add merge.tool.kdiff3.path "/usr/bin/kdiff3" //replace with your path
> git config --global --add mergetool.kdiff3.trustExitCode false
> git config --global --add diff.guitool kdiff3
> git config --global --add difftool.kdiff3.path "/usr/bin/kdiff3" //replace with your path
> git config --global --add mergetool.kdiff3.trustExitCode false
```

**Thank you**