



# Git Notes

**Git:**

World most popular version control system.

**Version Control System:**

Version Control is software that track and manages changes to file over time. Version Control System generally allow user to revisit earlier version of file, compare Change between version, undo changes and a whole lot more.

**Git is just one of the VCS.**

- There are many versions control system available today
- Like subversion, CVS and mercurial.
- According to survey, Git is Widely used.

**Git help us:**

- Track changes across multiple files.
- Compare version of projects.
- “Time Travel” back to old version.
- Revert to a previous version.
- Collaborative and share changes.
- Combine changes.

Eg: I am playing game (GTA). After every mission I save the progress and if I die at some point, I simply load the game.

Git is referred as **Global information tracker** or even as **Stupid content tracker**.

**Git and GitHub:****Git:**

Git is a version control software that runs locally on your machine. You don't need to register for an account. You don't need internet to use it. You can use Git without ever touching GitHub.

**GitHub:**

GitHub is a service that hosts our git repositories in cloud and makes it easier to collaborate with other people. You need to signup for an account to use GitHub. Its and online Place to share work that is done using git.

Project on my local machine, I can keep track of changes of my project and I can upload my history to GitHub to share with other people from other also can contribute to your project.

- ❖ Git was used or created to run on unix based interface (like Bash).
- ❖ Windows has default command line interface called command prompt.
- ❖ So fortunately, we have **Git Bash**. Git bash is a tool that emulates a bash experience on a windows machine and it comes with git too.

You can download setup file from their website

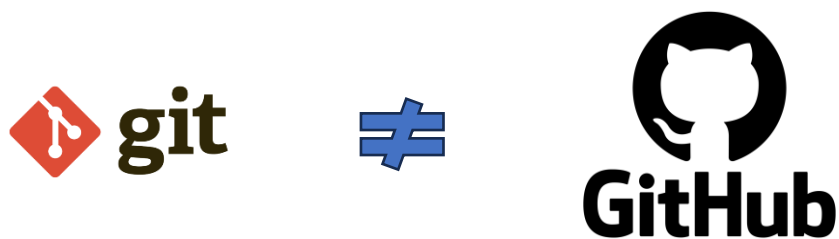
### Configurations:

We need to provide our name and mail. The name and mail is require to se who made what changes in the project.

**To configure Git:**

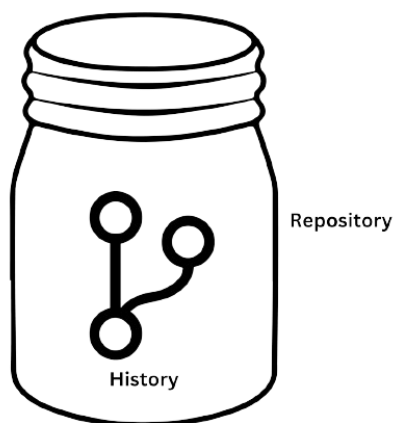
**Note:** When we go to GitHub, you will want your git email address to match your GitHub account.

**It is better to use same email address at both places.**



## Repository:

A “Git Repo” is a workspace which tracks and manages files within a folder.



## Git Status:

It gives information on the current status of a git repository and its content

`git status.`

## Git init:

It is used to create new git repository. we must initialize a repo first. This is something we do once per project.

`git init.`

## .git Folder:

It is Hidden Folder. It stores the history like what changes made by whom, when etc. If you delete this folder, you loose your history and it will no longer be a git repository.

## Committing:

Basic Git workflow

Step 1:

**Work on stuff:** making new file, edit, delete file.

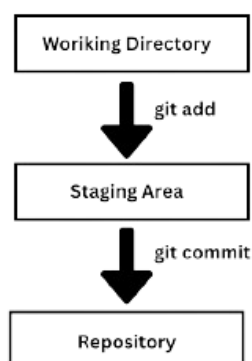
Step 2:

**Add changes:** Group specific changes together in preparation of commit.

Step 3:

**Commit:** Commit everything that was previously added.

Git captures a snapshot of project currently staged changes.



#### **Git add:**

A git add command adds a change in working directory to staging area. It tells git that you want to include updates to a particular file in next commit.

**git add .** : Adds all files in the directory

**git add Filename1 Filename2.**

#### **Staging Area:**

When you are ready to save changes, you first add them to staging area. Its like preparing your changes for snapshot. You can think of it's a changes you want to include in your next commit. Staging Area act as a preview area where you can review your changes before officially saving them to your project history.

#### **Git Commit:**

Git commit command actually commit changes from staging area. When making commit , we need to provide a commit message that summarizes changes and work snapshotted in commit.

**git commits.**

### `git commit -m "Message"`

`-m` flag allows us to pass inline commit msg rather than launching a text editor.

`git commit -a -m "Msg"` : to stage changes and commit in one command.

### **Git log:**

Git log will display a chronological list of commits starting with most recent commit and working backwards through history of your project.

Each Commit has:

- Commit hash: A unique identifier for commit, usually a long string of character.
- Author: The name and mail of person who made commit.
- Date and Time: When commit was made.
- Commit Message: A brief description of changes made in the commit.

### `git log`

`git log --online`: Displays commit in 1 line.

`git log --graph`: display text-based graph of branch.

Commits are present or placed on one above one like Stack.

**Commit 3**

**Commit 2**

**Commit1**

### **Atomic Commits:**

When possible, a commit should encompass a single feature, change or fix. In other words, try to keep each commit focused on single thing. This makes much easier to undo or rollback changes later on. It also makes your code or project easier to review.

### **Writing Commit Message:**

Present or pass tense?

It should be in imperative for like passing command or request.

It can be either in present or past tense both.

### **Amending Commits:**

Suppose you just made commit and then realized you forgot to include file or maybe your made a typo in commit message that you want to correct.

Rather than making one more commit, you can “Redo” the previous commit using the `--amend` option.

**Note: It only works with recent commits not with the older commits.**

### **Git ignore:**

#### **Ignoring Files.**

We can tell git which files and directories to ignore in a given repository using a `.gitignore` file.

This is useful for files you know you have never want to commit including.

- Secrets, API keys, Credentials etc
- Operating system files (.DS store in mac)
- Log Files
- Dependencies and packages.

A git ignore file is a simple text file used by git to specify intentionally untrack the file that git should ignore. This file typically contains pattern that match file names or path in your project that you don't want git to track.

This is useful for files that are generated during build process, temporary files or sensitive info like passwords that you don't want to be included in your repo.

A gitignore file:

#### **#ignore compiled build**

`*.class`

`*.o*`

`*.pyc`

#### **#ignore directories and files**

`/build, Secrets.txt`

## **Branching**



## Branches:

Branches are an essential part of git

Think of branches as alternative timeline for a project.

They enable us to create separate context when we can try to new things or even work on multiple idea in parallel.

**If we make change to 1 branch, they do not impact the other branches (unless we merge the changes).**

The Master Branch:

In git, we are always working on a branch. The default branch name is “Master.

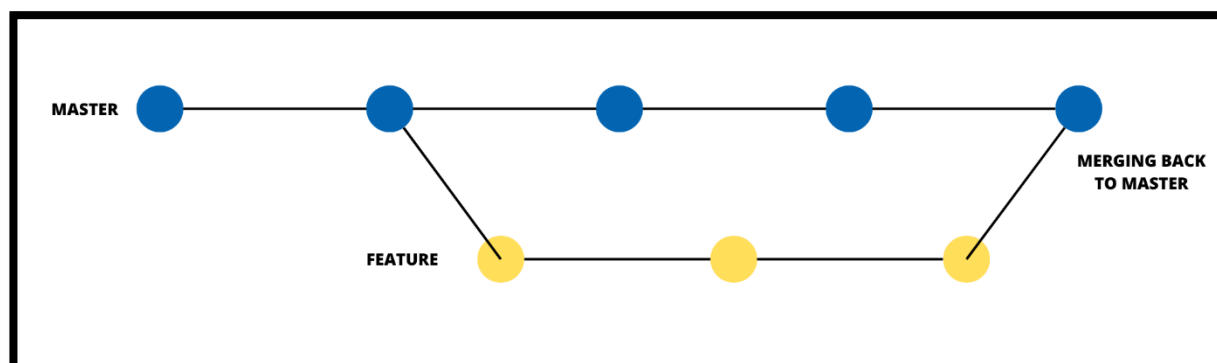
It doesn’t do anything special or have fancy power. Its just like any other branch

### Master

Many people designate the master branch as their “source of truth” or “Official Branch” for their codebase, but that is left to you to decide.

**In 2002, GitHub renamed the default branch from ‘Master’ to ‘Main’. The default git branch name is still master though a git team is exploring a potential change.**

## A common Workflow



What is (Head -> Master)

Head is simply a pointer that refer to current location in your repository. It points to a particular branch reference. So far , HEAD always points to latest commit you made on the master branch but soon we'll see that we can move around & HEAD will change.

**Head can be seen as an indicating pointer on which branch you are working.**

### Viewing Branches:

Use `git branch` to view your existing branches. The default branch in every git repo is master, though you can configure it.

**Look for \* which indicates the branch you are currently on.**

`git branch`

```
PS D:\VipulGit\branch> git branch
* master
```

### Creating branch:

Use `git branch <branch-name>` to make a new branch based upon current HEAD. This just creates branch. It doesn't switch you to that branch (the HEAD stays the same)

```
PS D:\VipulGit\branch> git branch dev
PS D:\VipulGit\branch> git branch
dev
* master
```

### Switching Branch:

Once you have created a new branch, use `git switch <branch name>` to switch to it

**`git switch -c <branch name>` :** Creates a branch and switches instantly to it.

```
PS D:\VipulGit\branch> git switch dev
Switched to branch 'dev'
```

```
PS D:\VipulGit\branch> git switch -c Feature
Switched to a new branch 'Feature'
PS D:\VipulGit\branch> git branch
* Feature
dev
master
```

### Another way of switching?

**Git checkout <branch name> to switch branches. The checkout command does a million additional things, so the decision was made to add a standalone switch command which is much simpler.**

### Switching Branches with Unstaged changes?

- So, if you are in branch1 and switch to branch2, its going to throw warning of losing the work.

```
ASUS@VipsyVrx MINGW64 /d/VipulGit/branch (dev)
$ git switch master
error: Your local changes to the following files would be overwritten by checkout:
    abc.txt
Please commit your changes or stash them before you switch branches.
Aborting
```

- If the changes do not conflict with files in the master branch, Git will allow you to switch branches seamlessly.
- If you are in branch1 and create another branch from it and create a file in it. Then you switch branch to other branch but it will still have untracked change with u. even after switching the branch.

### Deleting and renaming the branches:

#### For deleting:

1. You should be not on same branch, change the branch.
2. When u try to delete the branch, it might give you warning.

**git branch -d <branch name>**

```
ASUS@VipsyVrx MINGW64 /d/VipulGit/branch (master)
$ git branch -d feature
Deleted branch feature (was 1c52c13).
```

**git branch -D <branch name>:** Force Delete.

```
ASUS@VipsyVrx MINGW64 /d/VipulGit/branch (master)
$ git branch -d dev
error: the branch 'dev' is not fully merged.
If you are sure you want to delete it, run 'git branch -D dev'
```

```
ASUS@VipsyVrx MINGW64 /d/VipulGit/branch (master)
$ git branch -D dev
Deleted branch dev (was 570253b).
```

#### For Renaming:

1. You should be on the branch which you want to rename

**git branch -m <new branch name>**

```
ASUS@VipsyVrx MINGW64 /d/VipulGit/branch (master)
$ git switch -c name
Switched to a new branch 'name'

ASUS@VipsyVrx MINGW64 /d/VipulGit/branch (name)
$ git branch -m newname
```

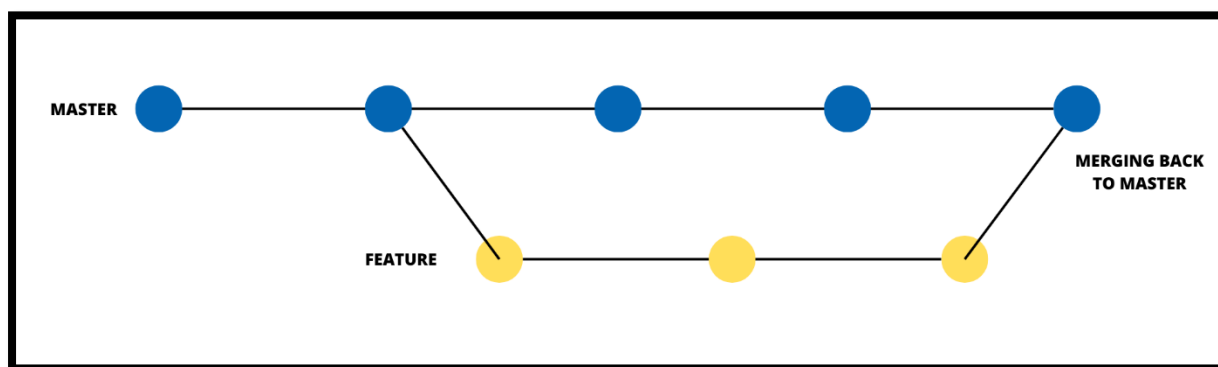
### Merging Branches:

## ✚ Merging:

Branching makes it super easy to work with self-contained contexts, but often we want to incorporate changes from one branch into another.

We can do this using **git merge** command.

### A Common workflow:



1. There is 1 master/main/Trunk branch which has a proper working application code etc.
2. To add feature, user create a feature branch from master branch and works on it & if all the works is good and Feature, I working then it is merged in master branch.

### Merging:

The merge command can sometime confuse student early on. Remember these two merging concepts.

- We merge branches not specific commits.
- We always merge to the current HEAD branch

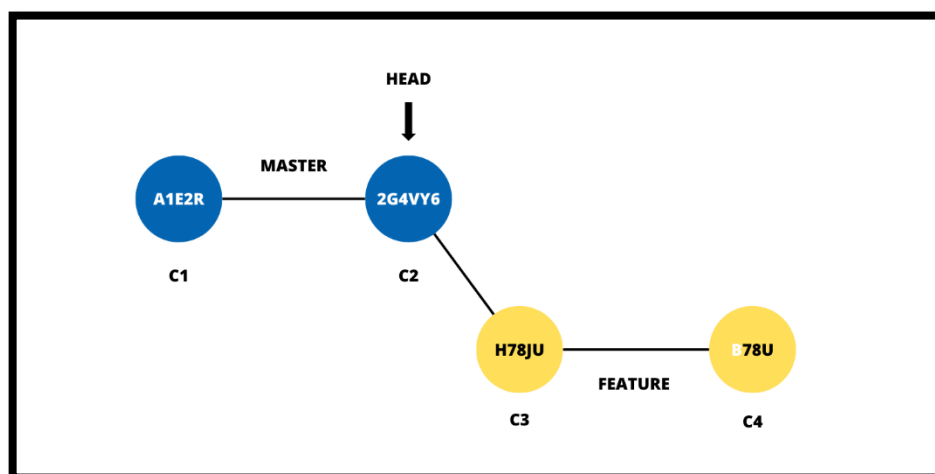
### Merging Made Easy:

To merge, follow these basic steps:

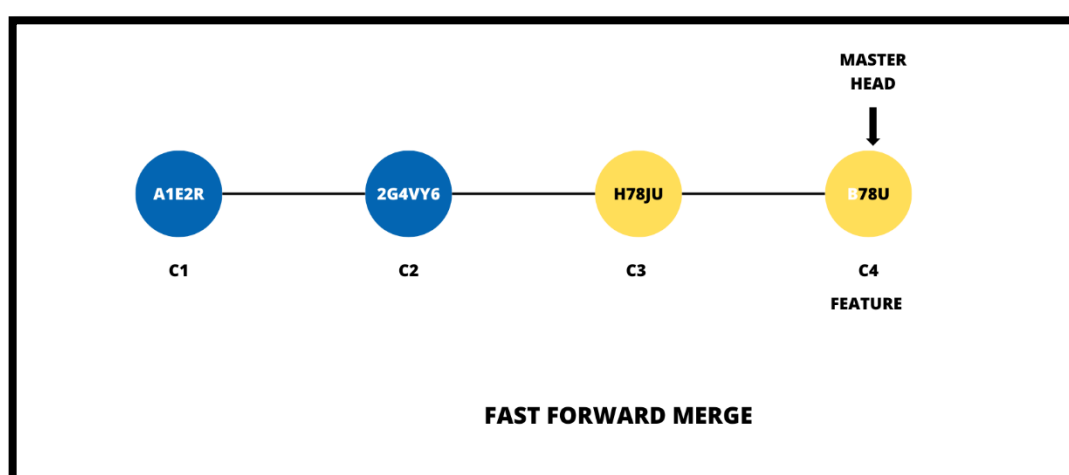
1. Switch to or checkout the branch you want to merge the changes into ( the receiving branch)

2. Use the git merge command to merge changes from a specific branch into current branch.

Visual representation:



1. There are two commits on master **C1** and **C2** and HEAD is present on master branch.
2. Now we create Feature branch from Master Branch and have 2 commit **C3** & **C4**.
3. After working on feature is done, we simply checkout to master branch and run **git merge feature** command.



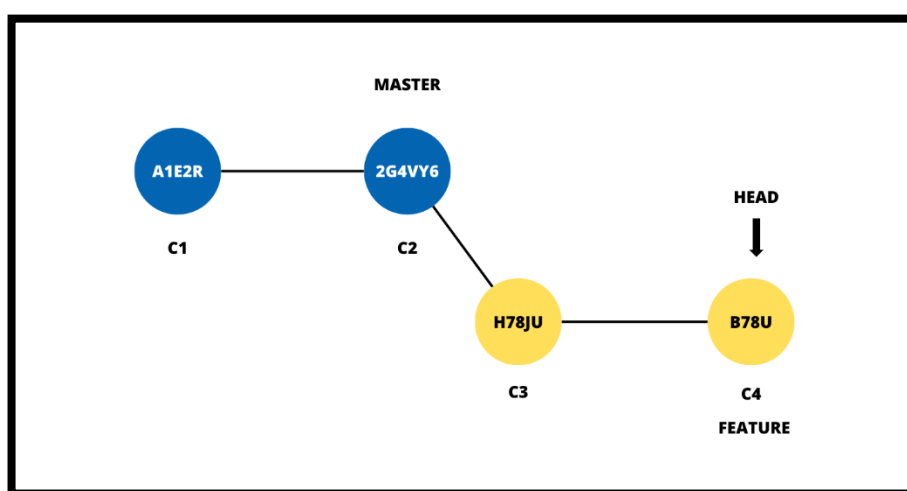
Above is called as Fast Forward Merge

## Fast Forward Merge:

A fast forward merge can be performed when there is direct linear path from source branch to target branch. In fact, forward merge, git simply moves the source branch pointer to target branch pointer without creating an extra merge commit.

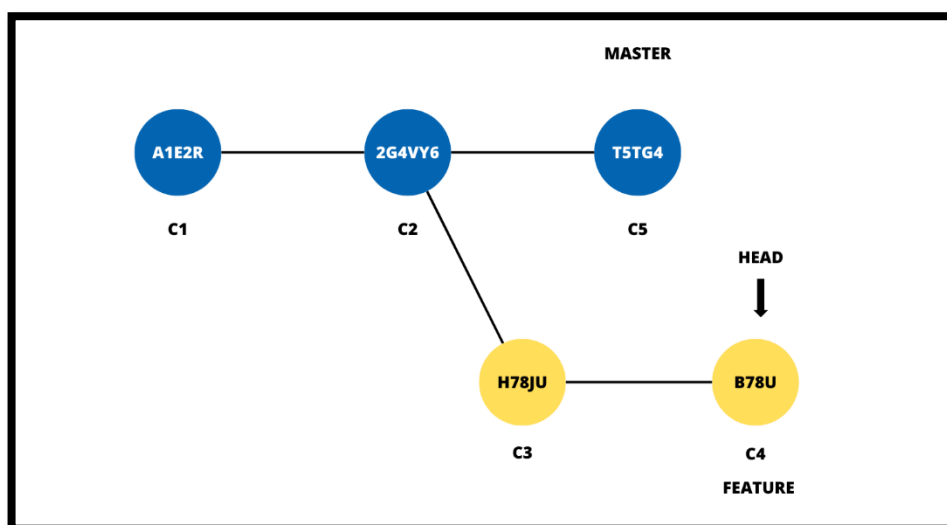
### Not All Merge are fast Forward

1. We have branched off of master, just like before

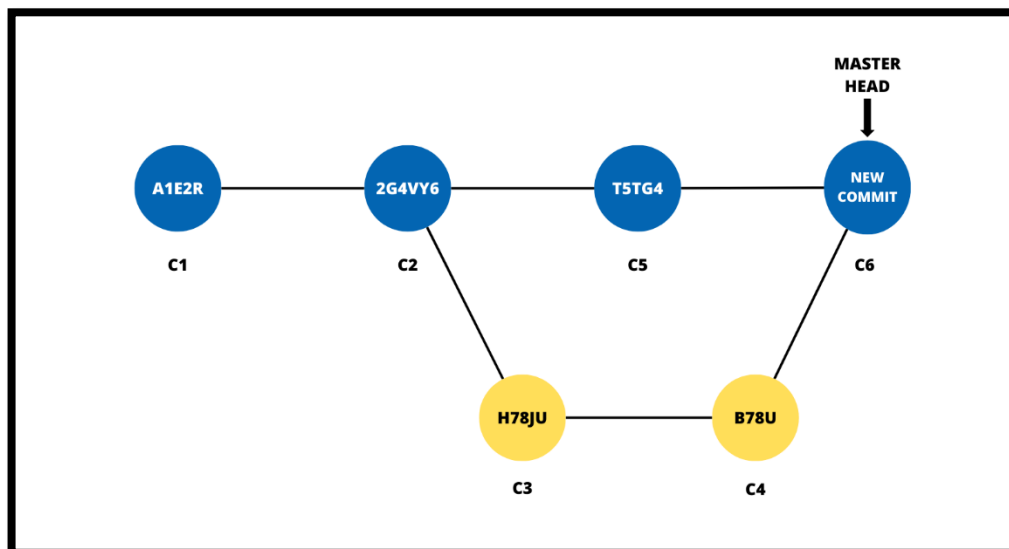


2. What if we add a commit on master?

This happens all the time! Imagine one of your teammates merged in a feature or change to master while you were working on branch.



Rather than performing a simple fast forward, git performs a “merge commit”. We end up with a new commit on master branch. Git will prompt you a message.



```

Merge branch 'Feature'
# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
~
~
~
~
~
~

```

```

ASUS@VipsyVrx MINGW64 /d/vipulGit/Merge (master)
$ git log --graph
* commit 48f36a6a9d464d8fa68418b6855b9dfda1b69866 (HEAD -> master)
  Merge: 5a77d9e 9d90c94
  Author: Vipul <vipulgaikwad567@gmail.com>
  Date: Fri Mar 22 11:13:26 2024 +0530

    Merge branch 'Feature'

* commit 9d90c9488fac08e8954c8d58d23bc9b21bf2b1e3 (Feature)
  Author: Vipul <vipulgaikwad567@gmail.com>
  Date: Fri Mar 22 11:10:09 2024 +0530

    Create B file

* commit 5a77d9ed532daab13d2e17d341f485812999f304
  Author: Vipul <vipulgaikwad567@gmail.com>
  Date: Fri Mar 22 11:11:29 2024 +0530

    Create Function.txt

* commit 01bc71d80c63578a648c415705774a4c3d4604e1
  Author: Vipul <vipulgaikwad567@gmail.com>
  Date: Fri Mar 22 11:08:51 2024 +0530

    Create A.txt file

```

Depending on specific change you are trying to merge, git may not be able to automatically merge. This result in merge conflict, which you need to manually resolve.

**>conflict (content): Merge conflict in abc.txt**

Automatic merge failed; fix conflict and then commit the result.

When you encounter a merge conflict, Git warns you in console that it could not automatically merge.

It also changes the content of your file to indicate the conflict that it wants you to resolve

Merge conflicts can occur in both situations, whether you change existing lines or add new lines, depending on the circumstances:

- **Changing Existing Lines:** If two branches modify the same lines of a file differently, Git will consider it a conflict. For example, if in the feature branch you change line 3 to say "X" and in the dev branch you change line 3 to say "Y", Git will not know which change to apply and will flag a conflict.
- **Adding New Lines:** If both branches add new lines to the same section of a file, Git generally handles this without conflict because it can combine the additions. However, if the added lines interfere with each other or with existing lines in a conflicting manner, Git may still raise a conflict.

So, while conflicts are more common when modifying existing lines, they can still occur when adding new lines, especially if the additions conflict in some way. It's always good practice to review changes carefully during merges to catch any conflicts early on

```
PS D:\VipulGit\Merge> git merge new_branch_to_merge_later
warning: Cannot merge binary files: merge.txt (HEAD vs. new_branch_to_merge_later)
Auto-merging merge.txt
CONFLICT (content): Merge conflict in merge.txt
Automatic merge failed; fix conflicts and then commit the result.
PS D:\VipulGit\Merge>
```

The screenshot shows the Git merge conflict resolution interface in VS Code. It displays a conflict in the file 'merge.txt' between the 'new\_branch\_to\_merge\_later' branch (Incoming) and the 'master' branch (Current). The Incoming branch has 'totally different content to merge later' on line 1. The Current branch has 'this is some content to mess with' on line 1 and 'content to append' on line 2. The Result section shows the current state of the file with the conflict highlighted. A 'Complete Merge' button is visible at the bottom right.



If Git performed a fast-forward merge instead of indicating a merge conflict, it suggests that the changes made in the branch could be applied directly on top of the master branch without conflicting with any existing changes.

This typically happens when: -

- The commits on the dev branch are ahead of the commits on the master branch.
- There are no conflicting changes between the branches.

In your case, it seems that the changes made in the dev branch were compatible with the changes in the master branch, so Git opted for a fast-forward merge, which simply moves the master branch pointer to the latest commit on the dev branch.

While fast-forward merges are convenient, they do not occur if the commit history of the branches diverges or if there are conflicting changes. In those cases, Git would perform a regular merge and potentially indicate conflicts for manual resolution.

If you want to experience a merge conflict, try making conflicting changes in the same lines of the same file on different branches and then attempt to merge them again.

## Resolving Conflict:

Whenever, you encounter merge conflict, follow these steps to resolve them.

- Open up the file (s) with merge conflict.
- Edit the file (s) to remove conflicts, decide which branch content you want to keep in each conflict or Keep content from both.
- Remove the conflict marker in document.
- Add your Changes And then make a commit.

## Git Diff

We can use git diff command to view changes between commits, branches, files, our working directory and more.

We often use git diff alongside command like git status and git log to get better picture of repository and how it has changed over time.

Without additional option, git diff lists all the changes in our working directory that are not staged for our next commit.

### Comparing Staging area and Working Directory

Last commit	New Changes
a/Animal	b/Animal
Tiger	Tiger
Lion	Lion
Fox	Fox
	Bear
	Elephant

```
PS D:\VipulGit\Diff> git diff
diff --git a/Animal.txt b/Animal.txt
index 922cf44..4937c06 100644
--- a/Animal.txt
+++ b/Animal.txt
@@ -1,3 +1,5 @@
  Tiger
  Lion
- Fox
\ No newline at end of file
+ Fox
+ Bear
+ Elephant
\ No newline at end of file
PS D:\VipulGit\Diff> █
```

### 1. Compared Files:

**Diff -git a/Animal/txt b/Animal.txt**

For each comparison, git explains which file it is comparing. Usually this is two version of same file

Git also declares one file as A and other file as B

## 2. File Metadata:

**Index 922cf44...4937c06 100644**

First two number are hashes of two files being compared. The last number is an internal file mode identifier.

## 3. Marker:

**--- a/Animal.txt**

**+++ b/Animal.txt**

File A and File B are each assigned a symbol

- File a get a minus sign (-).
- File B gets a plus sign (+).

## 4. Chunks:

**@@ -1,3 +1,5 @@**

**..**

**..**

**+Elephant**

A git diff won't show you entire content of file, but instead it only shows portions or "chunks" that were modified.

A chunk also includes some unchanged lines before and after a change to provide some context.

## 5. Chunks Header:

AA -1,3 +1,5 @@

Each chunk starts with chunk Header, Found between @@ & @@.

From File a, 3 lines are extracted starting from line 1

From File b, 5 lines are extracted starting from line 1

## 6. Changes:

Tiger

..

..

+Elephant

Every Line that changed between the two file is marked either with a + or – symbol.

Line begins with – comes from file A.

Line begins with + comes from File B.

## Different git diff Usage:

- Normal git diff:

**git diff**

git diff is used to compare changes that are not staged for next commit.

- Git diff Head:

**git diff HEAD**

git diff HEAD lists all changes in the working tree since your last commit.

Difference between git diff and git diff HEAD is that git diff compares changes that are not staged for next commit and git diff HEAD compares both staged and unstaged all changes since your last commit.

```

PS D:\VipulGit\Diff> git add .
PS D:\VipulGit\Diff> git diff
PS D:\VipulGit\Diff> git diff HEAD
diff --git a/Animal.txt b/Animal.txt
index 922cf44..4937c06 100644
--- a/Animal.txt
+++ b/Animal.txt
@@ -1,3 +1,5 @@
Tiger
Lion
-Fox
\ No newline at end of file
+Fox
+Bear
+Elephant
\ No newline at end of file
PS D:\VipulGit\Diff>

```

After adding changes to stage, the git diff command didn't work but git diff HEAD command worked.

- **Git diff --staged or --cached:**

**Git diff --staged**

**Git diff --cached**

It will list the changes between staging area and our last commit. "Show me what will be included in my commit if I run git commit right now"

**Show the changes between my stage changes and last commit.**

- **Diffing Specific Files:**

**Git diff HEAD [Filename]**

**Git diff --staged [Filename]**

**We can view changes withing specific Files by providing git diff with a file name.**

```
git diff HEAD Animal.txt
```

- **Comparing Branches:**

```
Git diff branch ..branch2
```

Will list changes between tips of branch1 and branch2.

```
Git diff master..Feature
```

- **Comparing Changes across Commit:**

```
git diff commit1..commit2
```

to compare two commits, provide git diff with the commit hashes of commit in questions.

```
PS D:\VipulGit\Diff> git diff 8f4be19..4563054
diff --git a/Animal.txt b/Animal.txt
index 922cf44..4937c06 100644
--- a/Animal.txt
+++ b/Animal.txt
@@ -1,3 +1,5 @@
 Tiger
 Lion
-Fox
 \ No newline at end of file
+Fox
+Bear
+Elephant
 \ No newline at end of file
PS D:\VipulGit\Diff> █
```

Commits are compared using hash id of commit.

## Git Stashing

- Suppose I am on master and commit 2 changes.
- Now I switch to other branch and made some changes and now my friend asked me to check on master branch which he has merged.
- If i switch back to master then there will be 2 scenarios:  
My changes come with me to destination branch.

```
PS D:\VipulGit\Diff> git switch -c Industry
Switched to a new branch 'Industry'
PS D:\VipulGit\Diff> git status
On branch Industry
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        Fruits.txt

nothing added to commit but untracked files present (use "git add" to track)
PS D:\VipulGit\Diff> git switch master
Switched to branch 'master'
PS D:\VipulGit\Diff> git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        Fruits.txt

nothing added to commit but untracked files present (use "git add" to track)
PS D:\VipulGit\Diff> █
```

Git won't let me switch If it detects potential conflict.

```
PS D:\VipulGit\Diff> git switch master
error: Your local changes to the following files would be overwritten by checkout:
        Fruits.txt
Please commit your changes or stash them before you switch branches.
Aborting
PS D:\VipulGit\Diff> █
```

Git provides an easy way of stashing these uncommitted changes so that we can return them later without having to make unnecessary commit.

### Git stash:

Git Stash is super useful command that help you save change's that you are not ready to commit. You can stash changes and then comeback to them later.

Running git stash will take all uncommitted change's (staged and Unstaged) and stash them, reverting the changes in your working copy.

git stash,

git stash save "Message" : It is better to give message to stash for understanding

```
PS D:\VipulGit\Diff> git stash
Saved working directory and index state WIP on Industry: d43b429 Fruit Created
PS D:\VipulGit\Diff>
```

## Git Stash Pop:

use git stash pop to remove the most recently stashed changes in your stash and re-apply them to your working copy.

```
PS D:\VipulGit\Diff> git stash pop
On branch Industry
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:   Fruits.txt

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (4fa43a0c77c12006125da286b7dc9bb23f440cdb)
PS D:\VipulGit\Diff>
```

## Git stash Apply:

You can use git stash apply whatever is stashed away without removing it from stash. This can be useful if you want to apply stashed changes to multiple branches.

## Git stash apply

Git stash will not pop anything it will be their inn stash area. It directly applies it and you can commit then.

```
PS D:\VipulGit\Diff> git stash apply
On branch Industry
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:   Fruits.txt

no changes added to commit (use "git add" and/or "git commit -a")
PS D:\VipulGit\Diff> git stash list
stash@{0}: WIP on Industry: d43b429 Fruit Created
PS D:\VipulGit\Diff>
```



## Stashing Multiple Times:

You can add multiple stashes into stack of stashes. They will be stashed in order u add them.

### Git stash

Do some stuff

### Git stash

Do some stuff

### Git stash

Do some stuff

```
PS D:\VipulGit\Diff> 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:   cars.txt

no changes added to commit (use "git add" and/or "git commit -a")
PS D:\VipulGit\Diff> git stash
Saved working directory and index state WIP on master: b6cca3e Colors creat
PS D:\VipulGit\Diff> 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:   Colors.txt

no changes added to commit (use "git add" and/or "git commit -a")
PS D:\VipulGit\Diff> git stash
Saved working directory and index state WIP on master: b6cca3e Colors creat
PS D:\VipulGit\Diff> git stash list
stash@{0}: WIP on master: b6cca3e Colors created
stash@{1}: WIP on master: b6cca3e Colors created
PS D:\VipulGit\Diff> █
```

## Viewing Stashes:

Run `git stash list` to view all stashes.

## Applying specific Stashes:

Git assumes you want to apply the most recent stash when you run `git stash apply`, but you can also specify a particular stash like `git stash apply 'stash@{2}'`

```
PS D:\VipulGit\Diff> git stash list
stash@{0}: WIP on master: b6cca3e Colors created
stash@{1}: WIP on master: b6cca3e Colors created
stash@{2}: WIP on master: b6cca3e Colors created
PS D:\VipulGit\Diff> git stash apply 'stash@{1}'
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:   Colors.txt

no changes added to commit (use "git add" and/or "git commit -a")
PS D:\VipulGit\Diff>
```

### Dropping Stashes:

To delete a particular stash, you can use `git stash drop <drop id>`.

`Git stash drop 'stash@{2}'`

```
PS D:\VipulGit\Diff> git stash drop 'stash@{2}'
Dropped stash@{2} (f88030f06b098b70ad1c042fc52ead7b314f94aa)
PS D:\VipulGit\Diff>
```

### Clearing the stash:

To clear out all stashes, run `git stash clear`.

```
PS D:\VipulGit\Diff> git stash list
stash@{0}: WIP on master: b6cca3e Colors created
stash@{1}: WIP on master: b6cca3e Colors created
PS D:\VipulGit\Diff> git stash clear
PS D:\VipulGit\Diff> git stash list
PS D:\VipulGit\Diff>
```

## Git Checkout

The Git checkout command is like a sit Swiss army knife. Many developers think it is over loaded which is what led to addition of **git switch** and **git restore** commands.

We can use checkout to create branches, switch to new branches, restore files and undo History.

We can use **git checkout commit<commit-hash>** to view a previous commit.

We can use the git log command to view commit hashes. We just need first 7 digit of a commit hash.

```
git checkout d8194d6
```

When you run this command

### Detached HEAD?

You are in 'Detached HEAD' state. You can look around, make experimental changes and commit them and you can discard any commit you make in this state without impacting any branches by switching back to a branch.

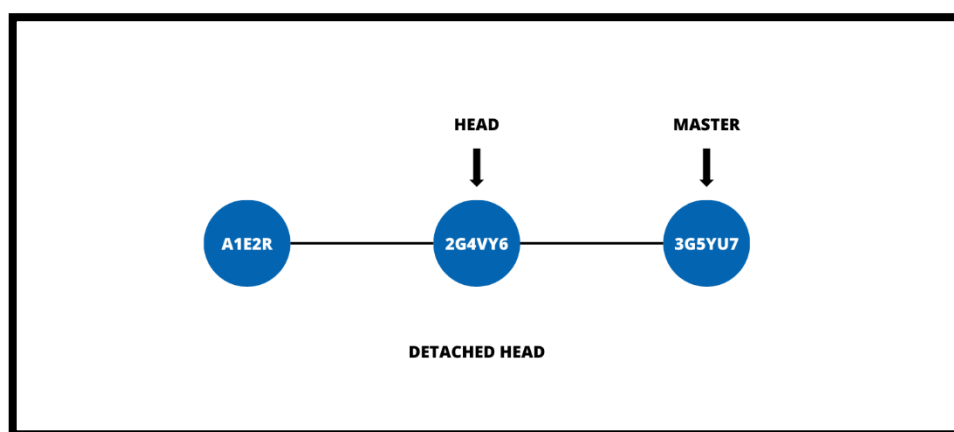
Usually HEAD points to the branch reference rather than a particular commit

How it works:

- HEAD is a pointer to current branch reference.
- Branch Reference is a pointer to the last commit made on particular Branch

When we checkout a commit, HEAD points at commit rather than at branch pointer.

**Git checkout 2G4VY6**



## Detached HEAD

**Git checkout <commit hash>**

We have a couple options:

- Stay in detached HEAD to examine the content of old commit. Poke around, view file.
- Leave and go back to wherever you were before reattaching the HEAD.
- Create a new branch and switch to it. You can now make and save changes. Since HEAD is no longer detached.

## What we can do and can't do in detached HEAD:

### Re-Attaching our detached HEAD:

#### For Viewing purpose

If you have detached head of a branch and you want to reattach it simply use git switch command on same branch you are.

```
PS D:\VipulGit\Checkout> git checkout 467ee49
Note: switching to '467ee49'.
```

You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may do so (now or later) by using -c with the switch command. Example:

```
git switch -c <new-branch-name>
```

Or undo this operation with:

```
git switch -
```

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 467ee49 Content added in abc.txt

```
PS D:\VipulGit\Checkout> git switch -
```

Previous HEAD position was 467ee49 Content added in abc.txt

Switched to branch 'master'

```
PS D:\VipulGit\Checkout> █
```

## For making Changes

If you have detached head of branch and want to try something new with it then you can Re-attach the head by creating new branch.

Your master branch changes and commit are present and when you switch branch in detached state it will Reattach to new branch created.

```
PS D:\VipulGit\Checkout> git checkout 467ee49
Note: switching to '467ee49'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

    git switch -c <new-branch-name>

Or undo this operation with:

    git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 467ee49 Content added in abc.txt
PS D:\VipulGit\Checkout> git switch -c bugfix
Switched to a new branch 'bugfix'
PS D:\VipulGit\Checkout> git log
commit 467ee49346effdafb5be59e400beaef121b29841 (HEAD -> bugfix)
Author: Vipul <vipulgaikwad567@gmail.com>
Date:   Fri Mar 29 11:48:34 2024 +0530

    Content added in abc.txt

commit e6b7ea610083b0240372f0d088ec0bd008115c11
Author: Vipul <vipulgaikwad567@gmail.com>
Date:   Fri Mar 29 11:47:23 2024 +0530

    abc.txt created
```

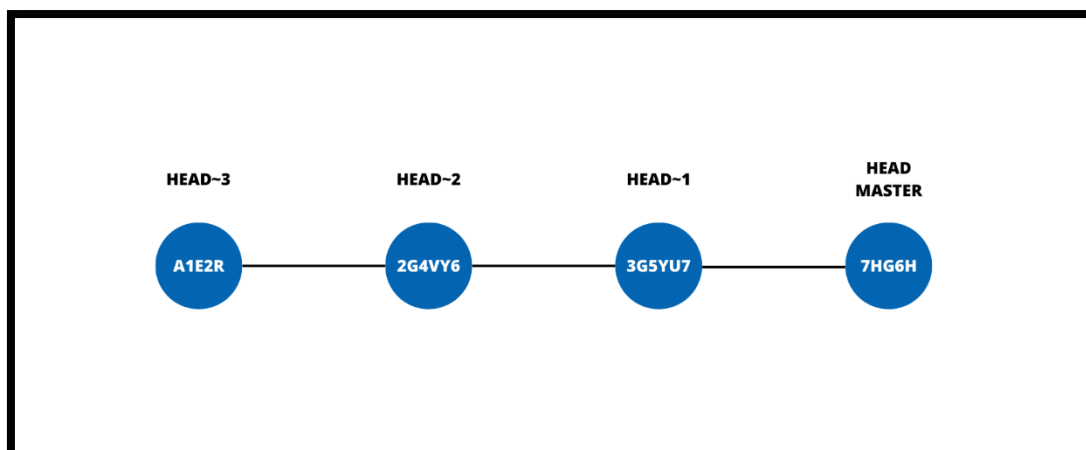
## Referencing commit relative to HEAD:

Git checkout supports a slightly odd syntax for referencing previous commit relative to a particular commit.

**HEAD~1 refer to commit before HEAD (parent)**

**HEAD~2 refer to commit before HEAD (grandparent)**

**git checkout HEAD~1**



It is just for easy usage instead of using Hashid id

If you are in detached state and don't remember which branch you are in, you can use:

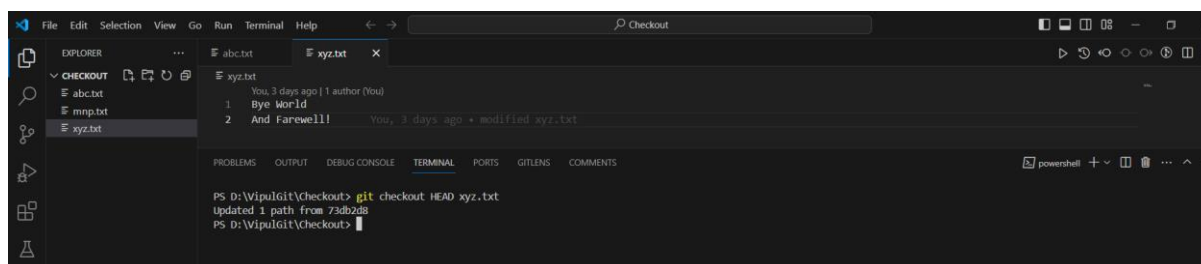
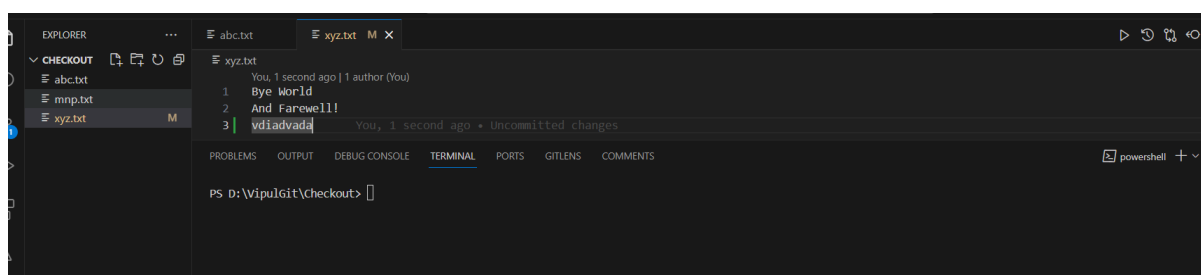
**git switch –**

### Discarding changes with git checkout:

Suppose you made some changes to a file but don't want to keep them. To revert the file back to whatever it looked like when you made last commit you can use:

**Git checkout HEAD <filename>**

To discard any changes in that file, reverting back to HEAD.



If you want to discard changes since last commit you can use this command

## Another Option

**git checkout --<file>**

It is used to discard changes which are not staged or committed.

## Git Restore

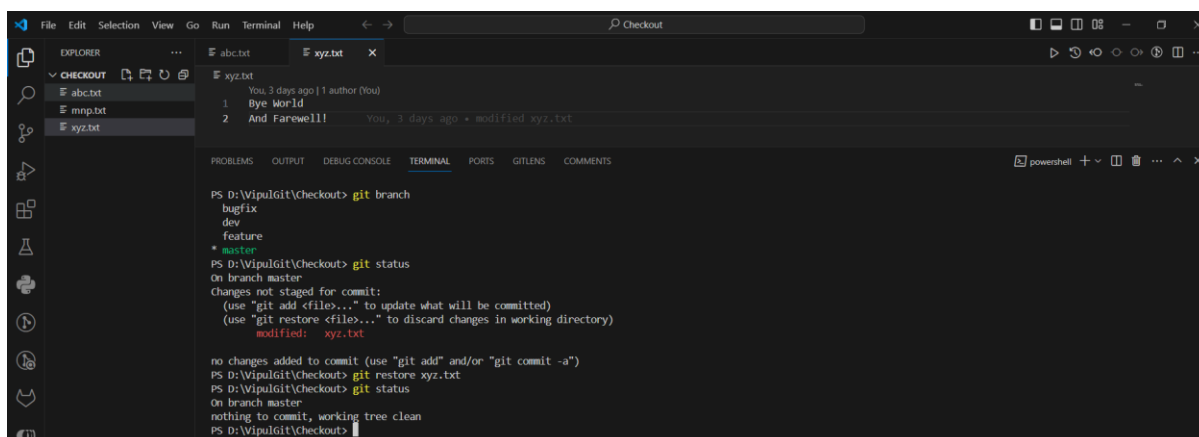
Git restore is a brand-new git command that helps with undoing the operations.

### Unmodifying Files with Restore:

Suppose you have made some changes to a file since your last commit. You have saved the file but then realize you definitely do not want those changes anymore

To restore the file back to content in HEAD. Use

**git restore <filename>**



```

PS D:\VipulGit\Checkout> git branch
bugfix
dev
feature
* master
PS D:\VipulGit\Checkout> 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:   xyz.txt

no changes added to commit (use "git add" and/or "git commit -a")
PS D:\VipulGit\Checkout> git restore xyz.txt
PS D:\VipulGit\Checkout> git status
On branch master
nothing to commit, working tree clean
PS D:\VipulGit\Checkout>

```

Note: The above command is not “undoable” if you have uncommitted changes in the file , they will be lost.

git restore<filename> restore using HEAD as the default source, but we can change that using the – source option.

For example, **git restore –source HEAD~1 home.html** will restore content of home.html to its state from commit prior to HEAD. You can also use a particular commit hash as source.

**git restore –source HEAD~1 app.js. use git restore filename to go back to latest changes.**

## Unstaging changes with Git Restore:

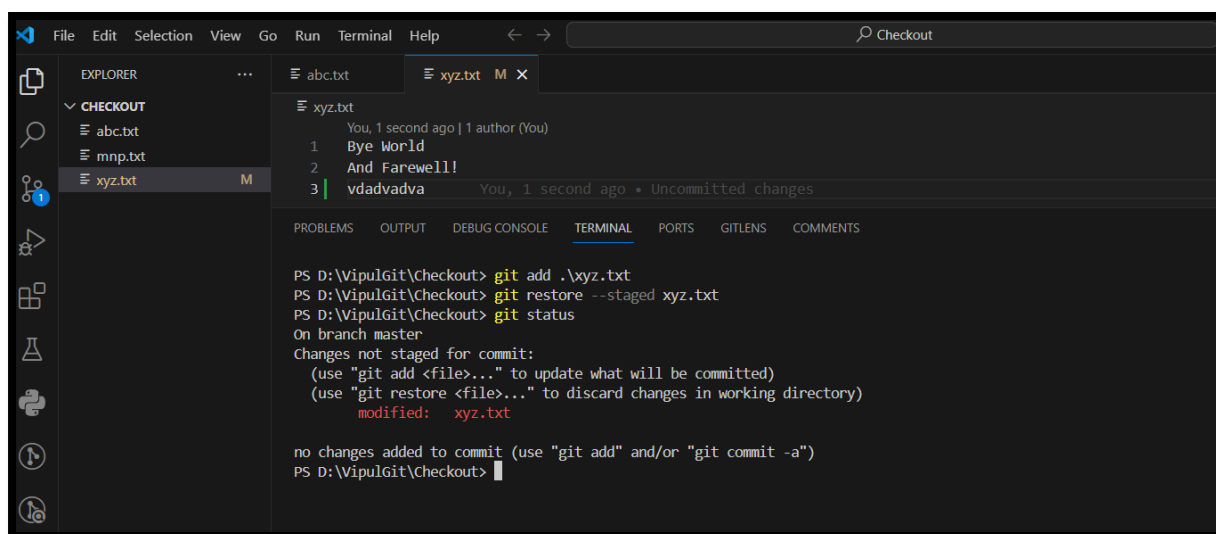
If you accidentally added a file to your staging area with git add and you don't wish to include it in next commit, you can use git restore to remove it from staging.

Use the `--staged` option like this:

**Git restore --staged app.js**

**Git restore --staged <filename>**

git restore is used for both for discarding unmodified changed and staged changes.



The screenshot shows the Visual Studio Code interface with a terminal window open. The Explorer pane on the left shows a file named `xyz.txt` with a green 'M' icon, indicating it is modified. The terminal window displays the following commands and output:

```
PS D:\WipulGit\Checkout> git add .\xyz.txt
PS D:\WipulGit\Checkout> git restore --staged xyz.txt
PS D:\WipulGit\Checkout> 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:   xyz.txt

no changes added to commit (use "git add" and/or "git commit -a")
PS D:\WipulGit\Checkout>
```

## Git Reset

Suppose you have just made a couple of commits on the master branch, but you actually meant to make them on a separate branch instead. To undo those commit you can use **git reset**

**git reset <commit hash>** will reset repo back to a specific commit. The commit is gone.

There is Regular reset and had reset.

A plain Reset:



```

PS D:\VipulGit\Checkout> git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    mmo.txt

nothing added to commit but untracked files present (use "git add" to track)
PS D:\VipulGit\Checkout> git log --oneline
eea3cd1 (HEAD -> master) Merge branch 'dev'
a50089a (bugfix) Added content in file mmp
6935525 Create mmp.txt
ec82c1b (dev) modified xyz.txt
44ac424 Modified abc.txt
363932c Added content in xyz.txt
bb8333e xyz.txt create
467ee49 Content added in abc.txt
e6b7ea6 abc.txt created
PS D:\VipulGit\Checkout> git reset ec82c1b
PS D:\VipulGit\Checkout> git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    mmo.txt
    mmp.txt

nothing added to commit but untracked files present (use "git add" to track)
PS D:\VipulGit\Checkout> git log --oneline
ec82c1b (HEAD -> master, dev) modified xyz.txt
44ac424 Modified abc.txt
363932c Added content in xyz.txt
bb8333e xyz.txt create
467ee49 Content added in abc.txt
e6b7ea6 abc.txt created
PS D:\VipulGit\Checkout>

```

When we perform git reset commit id it simply reset the repo back to particular commit & commit are gone, but the changes are still

### Present in working directory:

It helps when you mistakenly do certain commit in a branch which you want to do on other branch so you can use this **git reset <commit id>**.

## A Hard Reset:

### Reset - - hard

If you want to undo both the commit AND the actual changes in your file. You can use **--hard** option

E.g. **Git reset --hard HEAD~1** will delete the last commit & associated changes.

**When we hard reset commit & change both gets deleted.**

```

PS D:\VipulGit\Checkout> git log --oneline
ec82c1b (HEAD -> master, dev) modified xyz.txt
44ac424 Modified abc.txt
363932c Added content in xyz.txt
bb8333e xyz.txt create
467ee49 Content added in abc.txt
e6b7ea6 abc.txt created
PS D:\VipulGit\Checkout> git reset --hard 467ee49
HEAD is now at 467ee49 Content added in abc.txt
PS D:\VipulGit\Checkout> git log --oneline
467ee49 (HEAD -> master) Content added in abc.txt
e6b7ea6 abc.txt created
PS D:\VipulGit\Checkout> git status
On branch master
nothing to commit, working tree clean
PS D:\VipulGit\Checkout>

```

## Git Revert:

Another similar sounding and confusing command that has to do with undoing changes.

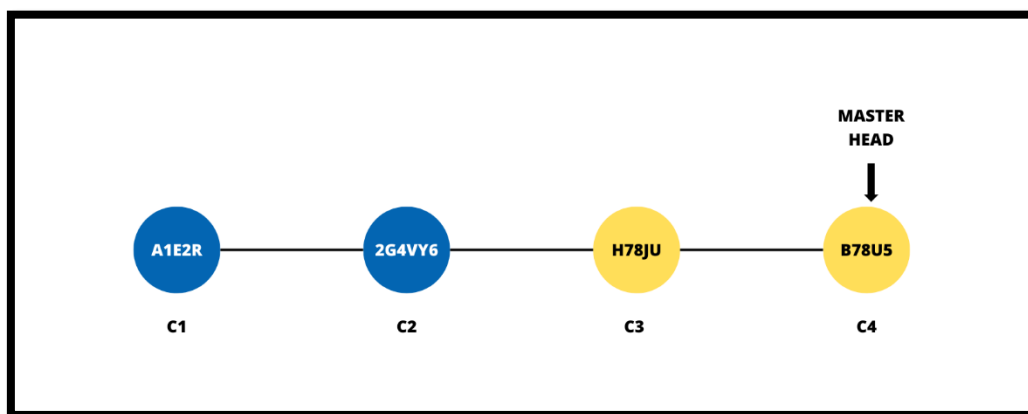
Git revert is simply to git reset in that they both “Undo” changes, but they accomplish it in different ways.

git reset actually moves the branch pointer backwards eliminating commit.

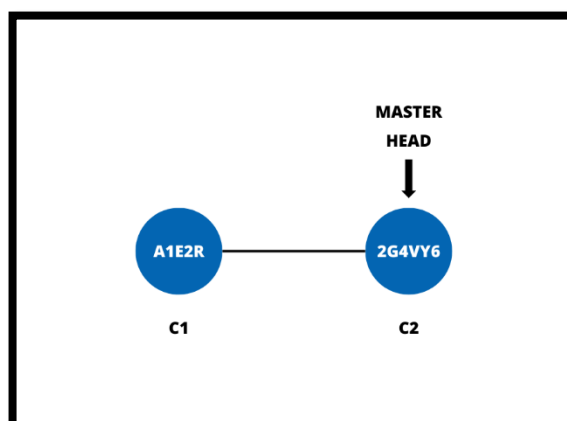
Git revert instead creates a branch new commit which reverse/undo the changes from a commit. Because it results in a new commit, you will be prompted to enter a message.

`git revert <commit-hash>`

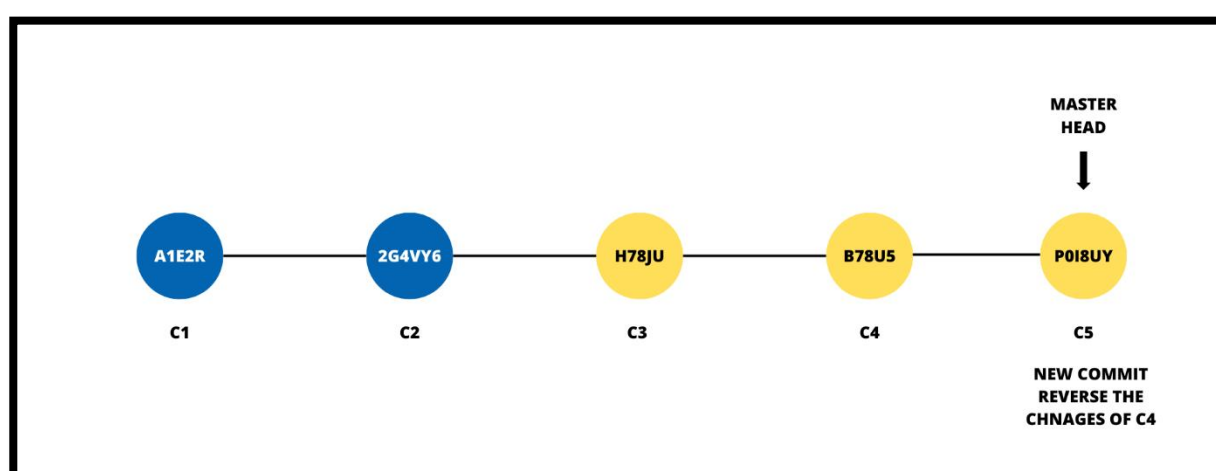
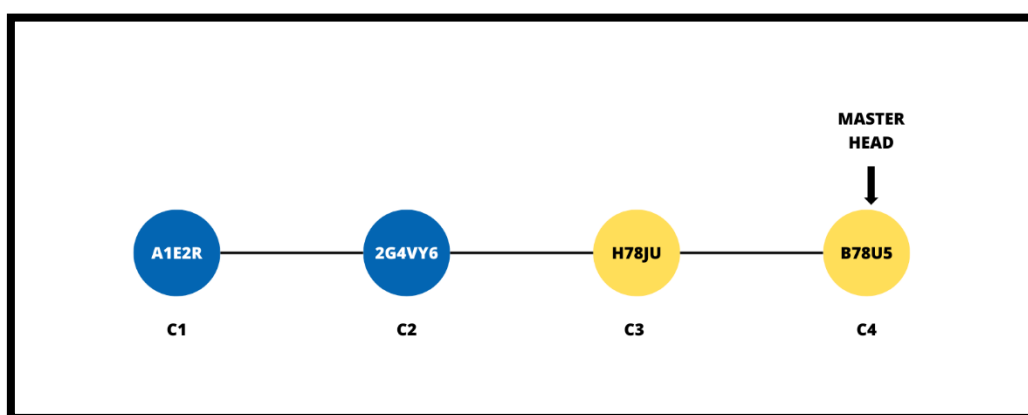
undoing using reset:



Branch pointer is moved to an earlier commit, erasing the 2 later commit.



Undoing using revert: -



It removes changes but keep it in history of commit.

**Which one should I use?**

Both git reset and git revert help us reverse changes, but there is a significant difference when it comes to collaboration.

1. If you want to reverse some commit that other people already have on their machine. You should use revert.
2. If you want to reverse commit that you haven't shared with other, use reset and no one will ever know.

## **Github: (Microsoft)**

Github is a hosting platform for git repositories. You can put your own Git repos on Github and access them from anywhere and share them with people around the world.

Beyond hosting repos, Github also provides addition collaboration features that are not native to git (but are super useful), Github helps people share and collaborate on repos.

### **Why you should use Github:**

#### **Collaboration:**

If you ever plan on working on a project with at least one another person, Github will make your life easier. Whether you are building a hobby project with you friend or you're collaborating with entire world, Github is essential.

#### **Open-Source Projects:**

Today GitHub is the home of open-source projects on internet. Project ranging from React to swift are hosted on Github. If you plant on contributing to open-source project, you will need to get comfortable working with Github.

#### **Exposure:**

Your Github profile showcase your own projects & contributions to other projects.

If act as a sort of resume that many employees will consult in hiring process. Additionally, you can gain some details of platform for creating or contributing to popular projects.

#### **Stay up to date:**

Being active on Github is the best way to stay up to date with project and tool you rely on. Learn about upcoming changes and the decisions/debate behind them.

## Cloning:

So far, we've created our own git repos from scratch, but often we want to get a local copy of an existing repository instead.

To do this, we can clone a remote repository hosted on Github or similar website. All we need is URL that we can tell Git to clone For Use.

## Git clone:

To clone a repo, simply run `git clone<url>`.

Git will retrieve all files associated with the repository and will copy them to your local machine.

In addition, Git initialize a new repository on your machine, giving you access to full git history of cloned Project.

`git clone<url>`

make sure you are not inside of repo when you clone

## Permission:

Anyone can clone a repository from Github. Provided the repo is public. You do not need to be an owner or collaborator to clone the repo locally to your machine. You just need URL from Github.

Pushing up your changes to Github repositories .. that's another story entirely! You need

Permission to do that.

## Setup Github Account

So, create your Github account.

Note: Create it with same mail and username which you had configured in git. It will be easy.

After that

## Setting up SSH keys

### SSH:

Secure shell protocol is method for secure remote login from one computer to another computer.

SSH enables secure system administration and file transfer over insecure networks using encryption to secure the connection between end points.

### SSH Keys:

It is an access credential similar to a password, used in ssh protocol. SSH Key grant, automate and enable remote access to a digital core of nearly every enterprise. They major financial institution, global industrial, tech giants and government to function securely.

You need to be authenticated on Github to do certain operations like pushing up code from your local machine. Your terminal will prompt you every single time for your Github email and password. Unless you generate and configure an SSH Key! Once configured, you can connect to Github without having to supply your username/Password.

**You can use Github docs for it to setup the SSH keys.**

[Connecting to GitHub with SSH - GitHub Docs](#)

## Create First Github Repo

### 1. First Option:

If you have already had an existing repo locally that you want to get on Github.

- Create a new repo on Github.
- Connect your local Repo (Add a remote)
- Push up your changes to Github.

### 2. Second Option:

If you haven't begun work on your local repo, you can

- Create a branch new repo on Github
- Clone it down to your machine.
- Do some work locally
- Push up your changes to Github.

## Remote:

Before we push anything to Github, we need to tell git about our remote repository on Github. We need to setup a "destination" to push up to.

In Git, we refer to this destination as remote, each remote is simply URL where a hosted repository lives.

## Viewing Remote:

To view any existing remote for your repository, we can run **git remote** or **git remote -v**. This just displays a list of remote. if you haven't added any remote yet, you won't see anything.

**git remote -v**

## Adding a new Remote:

A remote is really two things: a **URL** and a **Label**.

To add anew remote, we need to provide a both to git.

**git remote add <name> <URL>**

eg: `git remote add origin https://github.com/blas/repo.git`

okay git anytime. I use the name origin; I'm referring to this particular Github repo URL.

## ORIGIN?

Origin is a conventional Git remote name, but it is not at all a special. It's just a name for a URL.

When we clone a Github Repo, default remote name setup for us in is origin. you can change it.

Most people leave it.

### Other commands:

They are not commonly used, but there are commands to rename and delete remotes if needed.

`git remote rename <old><new>`

`git remote remove <name>`

## Pushing:

### Git Push:

Now that we have a remote set up, lets push some work up to Github:

We need to push using command, `git push`.

We need to specify the remote we want to push up to and the specific local branch. We want to push up to that remote.

`git push <remote> branch`

e.g.: `git push origin master`.

`Git push origin master` tell git to push up to master branch to our origin remote.

### Git push in detail:

#### IMP:

While we often want to push a local branch up to a remote branch of same name, we don't have to!

Suppose I have branch named pancake in my local and I have branch named waffle in my remote repo.

`git push <remote><local name>:<remote-branch>`

`git push origin pancake: waffle`.

We can even push the changes of one branch into another branch directly.



Git push origin empty: master

### The -u option:

The -u option allows us to set the upstream of branch we're pushing. You can think of this as link connecting our local branch to a branch on Github.

Running **git push -u origin master** sets the upstream of local master branch so that it tracks master branch on origin repo

**git push -u origin master**

It is like using -u you create upstream link from local to remote. So, when you push next time, you can simply use 'Git push' instead of 'Git push origin master'. It is just to make it easier and fast.

### Main and Master:

- Earlier Master was the to be considered as default branch.
- Later Github started "Main" as their default branch.
- Still some companies use master as default.
- While creating new repo in Github, you can change then name of default repo to whatever u want.
- Even u can use  
git branch -M Main: to change the branch name locally.
- If you are working on repo and it has master as default and after some work it is more like to change to main and u want to make it default then u can do it in using repo settings.

### Fetching and Pulling:

#### Remote Tracking Branches:

"At time, you last communicated with this remote repository, here is where X branch was pointing. They follow this pattern <remote>/<branch>

- **Origin/master:** reference state of master branch on remote repo named origin.

- **Upstream/logo Redesign:** reference state of logoRedisgn branch on remote named upstream (a common remote name).

## Remote Branches: -

Run `git branch -r` to view remote branches

`git branch -r`

### Scenario:

1. You cloned a repo, now both master (local) and origin/master (remote) are on same commit.
2. Now I make changes locally and commit my master(local) will move forward and origin/master(remote) will stay there until I push it to remote. Basically your local master moves but Remote reference doesn't move.
3. Using git status can help you to see it

**>git status**

On branch master

Your branch is ahead of 'Origin/master' by 2 commits.

(use "git push" to publish your local commit)

4. You can do `git checkout origin/master` to see your remote repo state. It will give you better idea.

## Working with Remote Repos:

1. So, when we clone a repo and we want to see branches. It doesn't mean it all comes in my workspace.
2. The Github repo has a branch called puppies, but when I run `git branch`, I don't see it on my machine. All I see is master branch.
3. By default, my master branch is already tracking origin/master.
4. **I want to work on puppies' branch locally!** I could checkout `origin/puppies`, but puts me in detached head. I want my own local branch called puppies and I want it to be connected to `origin/puppies`, just like my local master branch is connected to `origin/master`.

## 5. Its super easy!

Run `git switch <remote branch name>`

To create a new local branch from the remote branch of same name.

Git switch puppies makes a local puppies branch and sets up to track the remote branch origin/puppies

`git switch puppies`

### Note:

The new command `git switch` makes this super easy to do!. It used to be slightly more complicated using `git checkout`.

Git checkout –track origin/puppies.

## Fetching:

- Fetching allows us to download changes from a remote repository, but these changes will not be automatically integrated into our working files.
- It lets you see what other have been working on without having to merge those changes into your local repo.
- Think of it as “please go and get the latest information from Github, but don’t screw up my working directory”.

### Git Fetch:

`git fetch <remote>` command fetches branches & history from a specific remote repository. It only updates remote tracking branches.

`git fetch origin` would fetch all changes from origin remote repository.

`>git fetch <remote>`

If not specified, `<remote>` defaults to origin.

e.g: `git fetch origin master`

I now have those changes on my machine, but if I want to see them I have to checkout origin/master. My master branch is untouched.

### Git Fetch Demo:

1. I have a repo in my local & collaborator made/committed some changes to branch.

Branch: Origin/Main

Origin/Food

Origin/Movies

They made changes to Food and Movies.

2. Now I want to see those changes without disturbing those changes.

3. `>git fetch origin` #to fetch whole repo

`>git fetch origin branch name` #to fetch particular branch.

`>git branch -r`

`>git checkout origin/Food.`

This will fetch remote food branch and I can see all changes git log too see the commits

`>git switch -`

To get out of detached state.

### Pulling:

#### Git pull:

Git pull is another command we can use to retrieve changes from remote repository. Unlike fetch, pull actually update our HEAD branch with whatever changes are retrieved from remote.

**“go and download data from Github and immediately update my local repo with those changes”**

**Git pull = Git fetch + Git Merge.**

#### Git pull:

TO pull, we specify particular remote & branch we want to pull using `git pull <remote> <branch>`. just like with git merge, it matter Where we run this command from. Whatever branch we run it from is where the changes will be merged into.

Git pull origin master would fetch latest information from origins master branch and merge those changes into our current branch.

**Git pull <remote> <branch>**

**Pull can result into merge conflict**

**Git pull Demo:**

**Same scenario as fetch.**

**>git pull origin master.**

It will bring all changes present into remote repo into your workspace.

**>git log**

**Git pull and Merge conflict:**

1. Sometimes it may be possible that collaborator had made changes to same file on which you are working
2. And when you make git pull It can lead it into merge conflict.
3. You can resolve the conflict commit it & push it into remote.

**An Easier Syntax for all:**

If we run git pull without specifying a particular remote or branch to pull from, git assumes following:

- Remote will default to origin.
- Branch will default to whatever tracking connection is configured for your current branch.

Note: This behavior can be configured & tracking connections can be changed manually. Most people don't mess with that stuff.

It is like similar to git switch

If you use git pull on branch it will work and will pull the branch you are currently on.

Git Fetch	Git Pull
Get changes from remote branch	Get changes from remote branch
Updates the remote tracking branches with new changes.	Update current branches with new changes, merging them in
Does not merge changes onto your current.	Changes result in merge conflict.
Safe to do anytime	Not recommended if you have Uncommitted changes.

## Github Grab Bag: Odds and Ends:

### Public vs Private Repos:

#### Public repo:

Public repos are accessible to everyone on the internet. Anyone can see the repo on Github.

#### Private Repo:

Private repos are only accessible to the owner and people who have been granted access.

### Deleting the Github Repos:

Repos can be deleted by owner at any point of time at his wish.

### Adding Collaborators:

You can give access top collaborator using repo setting in Github

### READMEs:

READMEs file is used to communicate important information about repository including:

- What Project Does.
- How to run the project.
- Why its Noteworthy.
- Who maintains the Projects

If you put READMEs in the root of your project Github will recognize it and automatically display it on repo's Home page.

### **README.md:**

READMEs are Markdown file, ending with the .md extension. Markdown is a convenient syntax to generate formatted text. Its easy to pick up!

### **Markdown Crash Course:**

It is used to generate formatted text for ReadMe page.

[Markdown-it.github.io](https://markdown-it.github.io)

Write your own ReadMe.

### **Github Gists:**

Github gist's are a simple way to share code snippet and useful fragments with other. Gists are much easier to create, but offer for fewer than a typical Github repository.

### **Github Pages:**

Github pages are public webpages that are hosted and published via Github. They allow you to create a website simply by pushing your code to Github.

Github Pages is a hosting Service For Static webpages so it does not Support Server side code like python, Ruby or Node. Just HTML/CSS/JS.

## 2 Flavors of Github Pages:

### 1. User Site:

You get one user site per Github account. This is where you could host a portfolios site or some form of personal websites. The default URL is based on your Github username. Following this pattern : `username.github.io` though you can change this.

### 2. Project Site:

You get unlimited Project site! Each Github repo can have an corresponding hosted website. It as simple as telling Github which specific branch contains web content. Default url follows this pattern **Username.github.io/repo name**

## Hosting Github Pages:

1. Go to repo settings
2. Choose the branch which has pages.
3. Choose Folder mainly its root only.
4. And done you will see URL
5. You can use it for demonstrations in Readme file also by providing link.

## Github Collaboration Workflows

### 1. Centralized Workflow:

AKA Everyone works on Master/Main.

AKA The most Basic Work flow Possible



The simplest Collaborative workflow is to have everyone work on master branch(or main, or any other SINGLE BRANCH)

Its Straight Forward and can Work for tiny teams but it has quite a few shortcoming.

#### **SCENARIO:**

- 3 people working on same master branch.
- A made some changes and pushed it to remote. Throws them a error because they are not having update work in their local repo.
- B and C also made but cannot push it will throw them error because they are not having update work in their local Repo.
- Sometimes it can also lead to merge conflict.

#### **Problem:**

While its nice and easy to only work on master branch, this lead to some serious issues on teams.

- Lots of time spent resolving conflict and merging code, especially as team size scale up
- No one cane work on anything without disturbing the main codebase.
- The only way to collaborate on a feature together with another teammate is to push incomplete code to master. Other teammates now have broken code

## **2. Feature Branch Workflow:**

Rather than working directly on master/Main all new development should be done on separate branches!

- Treat master/main branch as the official project history.

- Multiple teammates can collaborate on single feature and share code back and forth without polluting the master/main branch.
- Master/main branch won't contain broken code (or at least, it won't unless someone messes Up).

### **Merging in feature Branches:**

At some point new th work on feature branches will need to be merged into master branch:

There are couple of options for how to do this:

- Merge at will, without any sort of discussion with teammate. JUST DO IT WHENEVER YOU WANT.
- Send an email or chat message or something to your team to discuss if changes should be merged in.
- Pull Request.

### **3. Pull Requests: (Used in organizations)**

Pull request are a feature built in to product like Github and bitbucket.

They are not native to git itself.

They allow developers to alert team mates to new work that need to be reviewed. They Provide a mechanism to approve or reject work on a given branch. They also help facilitate discussion and feedback on specified commit

**“I have this new stuff I want to merge into master branch. What do you all think about it?”**

**Workflow:**

- I. Do some work locally on your feature branch. Push up the feature branch.
- II. Push up the feature branch to Github.
- III. Open a pull request using feature branch just pushed up to Github.
- IV. Wait for PR to be approved and merged. Start discussion on PR. This part depends on the team structure.

You can create pull request on Github and add comment. The owner will come and see the changes will discuss and then decide whether to merge or not.

First you have to compare your branch with the main branch.

You can restrict the merge request action by changing setting

Sometimes while merging the request there will be fast forward merge and sometime it will be merging conflict present merge pull request with conflict

**4. Merge Pull request with conflict:**

- I. Once I make requested changes, my boss (or whoever is in the charge of merging) can merge in my pull request
- II. Just like any other merge, sometimes there are conflict that need to be resolved when merging a pull request. This is fine. Don't panic.
- III. You can perform merge and fix conflict on command line like normal or you can use Github interactive editor.

**Boss can merge the branch and resolve the conflict locally.**

1. Switch to branch in question. Merge in master and resolve conflict.

>git fetch origin

>git switch my new feature

>git merge master

>fix conflicts

2. Switch to master. Merge on feature branch (now with no conflict).Push changes upto Github.

>git switch master

>git merge my new feature

>git push origin master.

## 5. Configuring branch protecting rules:

You can protect your branches by adding rules.

- You can set no commit on main branch by collaborator.
- Code review and vote system for merging request.

## 6. Fork and Clone:

### Another Workflow:

The 'Fork and Clone' workflow is different from anything we've seen so far. Instead of just one centralized Github repository, every developer has their own Github repository in addition to "main" repo. Developer makes changes and push to their own fork before making pull requests. Its very commonly used on large open-source project and where there may be thousands of contributors with only a couple maintainer.

### Forking:

Github (similar tool) allows us to create personal copies of other people's repositories. We call those copies a 'Fork' of original.

When we fork a repo, we're basically asking Github **"Make me my own copy of this repo please"**.

As with the pull request, Forking is not a git feature the ability to fork is implemented by Github.

### **Now what**

Now that I've forked, I have to very own copy of repo where I can do whatever I want. I can clone my fork and make changes, add feature and break things without fear of disturbing the original repository.

If I don't want to share my work, I can make a pull request from my fork to original repo.

1. Fork the project
2. Clone the fork
3. ADD Upstream remote
4. Do some work
5. Push to origin
6. Open Pull Request.

This "Fork and clone" workflow might seem complicated but its extremely common for good reason. It allows a project maintainer to accept contributors from developer all around world without having to add them as actual owner of main projects repository or worry about giving them all permission to push to repo(which could be disastrous)

**Demo:**

1. You Fork a project
2. You clone it too local
3. Add remote
4. Add upstream remote
5. Now we have 2 remotes

`git remote add upstream`

`Git pull upstream main`

**To pull changes of original repository.**

6. You can make changes in local
7. You push to forked repo
8. You create pull request to original repo

**Rebasing:**

There are two ways to use git rebase command:

- As a alternative to merge
- As a cleanup tool

**Rebasing:**

We can instead rebase feature branch into master branch. This moves the entire feature branch so that it **BEGINS** at the tip of master branch. All the work is still there, but we have re written the history

Instead of using merger a commit, rebasing rewrite history by creating new commits for each original feature branch commits.

>git switch feature

>git rebase master

### Rebasing Workflow:

1. It is basically similarly to merge.
2. So instead of merging you can use rebase.
3. Suppose you have master and feature branch you have 2 commit on master and made feature branch and you worked on feature branch
4. Now somebody made Changes in master branch and you want it in your feature branch.
5. You can switch to feature and use cmd

### Git rebase master

6. So our feature branch commits are recreated and it all begins at the tip of master branch.

### When not to rebase:

Rebasing Helps in reading the history.

- NEVER REBASE commit that have been shared with other. If you have already pushed commits up to Github..... Do not rebase them unless, you are positive. No one on team is using those commits.
- You do not want to rewrite any git history that other people already have. It's a pain to reconcile the alternative histories.

### Handling conflict and rebasing:

1. You can face merge conflict while rebasing
2. & it will ask you to solve the merge conflict.
3. And then it will ask you for git rebase --continue to continue the rebase
4. Or it will simply ask you to skip (Abort the rebase) git rebase --abort

**git rebase --continue**

**git rebase --skip**

## **Introducing Interactive Rebase:**

### **Rewriting History:**

Sometimes we want to rewrite delete, rename or even reorder commits (before sharing them) we can do this using git rebase.

### **Interactive rebase:**

Running a git rebase with -i Options will enter the interactive mode, which allows us to edit, modify, add files, drop commits etc. Note that we need to specify how far back we want to rewrite commit.

Also, notice that we are not rebasing onto another branch. Instead we are rebasing a series of commits onto HEAD that currently based on

**Git rebase -i HEAD~4**

### **Now what?**

In our text editor, we'll see a list of commits alongside a list of command that we can choose from. Here are Couple of more commonly we command

- Pick: use the commit



- Reword: use commit, but edit commit message.
- Edit: use commit ,but stop for amending.
- Fixup: use commit contents but meld it into previous commit and discard commit message
- Drop: remove commit.

You use command, use choose type of work you want to do and you save it make changes in file and save it, it will get updates.

### **Fixing up and Squashing commit with interactive Rebase:**

- When we use fixup it just squashes commit in previous commit basically delete the commit from history by adding it ti previous commit. The work or changes made by that commit are sill present, it only deletes the commit from commit history.
- Dropping commit with interactive rebase  
So in this we not only remove the commit message but also remove the changes done by commit  
We will use drop command here

## Git Tags:

Tags are pointer that refer to particular points in git history. We can mark a particular Moment in time with a tag. Tags are most often used to mark version releases in projects (v4.1.0, v4.1.1 etc.)

Think of tags as branch reference that do NOT CHANGE. Once a tag is created, it always refers to same commit. It's just a label for a commit.

**Tag is basically aging a particular commit in git repository more like of versions**

## Two types of Git tags:

There are two types of gits tags we can use:

### Lightweight and annotated tags

**Lightweight tags** are ...lightweight. They are just a name/label that point to particular commit.

**Annotated tags** store extra meta data including the authors name and mail, the date and a tagging a message (like a commit).

## Semantic Versioning

**The** semantic versioning spec outline a standardized versioning system for software release. It provides a consistent way for developer to give meaning to their software releases.

Versioning consists of three numbers separated by periods:

1. Major version when you incompatible API changes.

2. Minor version when you add functionality in backwards compatible manner.
3. Patch version when you make backwards compatible bug fixes

**Initial releases: 1.0.0**

### **Patch Releases:**

Normally contain new feature or significant changes. They typically signify bug fixes and other changes that do not impact how code is used.

### **Minor Releases:**

Minor releases signify that new feature or functionality have been added, but project is still backwards compatible. No breaking changes. The new Functionality is optional and should not force user to rewrite their own code.

### **Major Release:**

Major Releases signify significant changes that is no longer backward compatible. Feature may be removed or changed substantially.

### **Viewing Tag:**

**>git tag**

Git tag will print a list of all the tags in current repository.

We can search for tags that match a particular pattern by using git tag -l and then passing in wildcard pattern.

For example: git tag-l “\*beta\*” will print list of tags that include beta in their name

**>git tag-l “\*beta\*”**

## Comparing tag with git diff:

### Checking out tags:

To view the states of repo at a particular tag, we can use `git checkout<tag>`.

This put us in detached HEAD!

```
>git checkout <tag>
```

Git checkout 16.1.0

You go in detached HEAD now

```
>git switch -c Branch_from_tag
```

Now you can see files in that tag

You can see the changes between two tags

```
>git diff v.17.0.0 v17.0.3
```

## Creating Lightweight Tags:

To create a lightweight tag, use `git tag <tagname>`. By default, Git will create tag referring to commit that HEAD is referencing.

```
git tag <tag name>.
```

### Create Annotated tags:

Use `git tag-a` to create new annotated tag. Git will then open your default text editor and prompt your additional information

Similarity to `git commit` we can also use the `-m` option to pass a message directly and forges opening of text editor.

```
git tag -a <tag name>
```

```
>git tag -a 17.1.3
```

```
>git show 17.1.3
```

**Tagging previous Commits:**

We can also tag an older commit by providing the commit hash

```
git tag -a <tagname><commit-hash>
```

**Replacing Tag with FORCE:**

**Forcing TAG:-**

Git will yell at us if we try to use a tag that is already referencing to a commit.

If we use -f options, we can FORCE our tag through

```
>git tag -f <tag name>
```

```
>git tag v17.0.3 696e736be
```

Fatal tag already existed.

```
>git tag v17.0.3 699ef726be -f
```

Updated tag

**Deleting Tag:**

To deleting a tag, use **git tag -d <tagname>**

```
>git tag -d v17.0.3
```

## Pushing Tags:

By default, the git push command doesn't transfer tags to remote server. If you have a lot of tags that you want to push up at once, you can use the `-tags` option to the git push command. This will transfer all of your tags to the remote server that are not already there.

**>git push -tags.**

You need to push tags they are not pushed automatically.

Git Behind the scenes:

### 1. Config:

The config file is for configuration, we have seen how to configure global settings like our name and email across all git repos, but we can also configure things on a per repo basis.

Watch git book reference for more:

You can set color for username and many more things.

### 2. Refs Folder:

Refs/Head contains one file per branch in a repository. Each file is named after branch and contains the hash of commit at tip of branch.

Refs also contains a refs/tags folder which contains one file for each tag in repo.

Reference ref

It has a pointer to commit, branch tag stored in it.

### 3. Head file:

- Head is just a text file that keeps track of where HEAD POINTS
- If it contains refs/heads/master, this means
- That HEAD is pointing to master branch.
- In Detached HEAD, the HEAD file contains a commit hash instead of branch reference

### 4. Object Folder

Object Directory contains all the repo files. This is where Git stores the backup of files, the commit in a repo and more.

The files are all compressed and encrypted so they won't look like much.

There is binary file present in it which stores whole snapshot like history of commit and much more they all are hashed.

### **Hashing Function:**

Hashing Function are functioning that map input data of some arbitrary size to fixed sized output values.

Git uses SHA-1 hash function to generate commit hashes.

### **Git Database:**

Git is a key-value data store. We can insert any kind of content into a git repository, and Git will hand us back a unique Key. we can later use to retrieve that content.

These Keys that we get back are SHA-1 checksum.

### **Hashing with Git Hash-Object:**

#### **Echo 'hello' | git hash-object -stdin**

The `-stdin` option tells git hash-object to use content from stdin rather than a file. In our example it will hash the word "hello"

The echo command simply repeats whatever we tell it repeat to the terminal. We pipe the output of echo to git hash-object.



## Let's try hash

### Git hash – object <file>

Git hash-object command takes some data stores in our .git/object directory and give us back the unique SHA-1 hash that refer to that data object.

In simplest form, git simply take some content and return unique key that WOULD be used to store our object. But it doesn't not actually store anything.

### Working:

**Echo "asadadad Asdas"**

**Echo "hi" | git hash-object -sdtin**

45b983be36b73c0788dc9cbcb76cbb80fc7bb057

### Retrieving Data with git Cat file:

git cat-file -p <object-hash>

now that we have data stored in our git object database, we can try retrieving it using git cat-file command.

The -p option tell git to pretty print content of object based on its type.

**Object: blobs**

**Object: Trees**

**Object: Commits**

**Reflog:**

Git keep a record of when the tips of branch and other reference were updated in repo.

We can view and update these reference logs using the git reflog command.

**Limitations:**

Git only keep reflog on your local activity. They are not shared with collaborators.

Reflog also expires. Git cleans out old entries after around 90 days, though this can be configured.

The git reflog command accepts subcommand show, expire, delete and exists. Show is only commonly used variant and it is default subcommand.

**Git reflog show** will show the log of a specific reference (it defaults to HEAD)

For example: to view the logs for tip of main branch we could run **git reflog show main**

**Git reflog show HEAD**

**Git reflog**

**Reflog Reference:**

We can access specific git refs is name @ {qualifies}

We can use this syntax to access the specific refs pointer and can pass them to other command including checkout, reset and merger.

name@ {qualifier}

**git reflog show HEAD@{24}**

`git checkout HEAD@{24}`

it will take you into detached state now

`git diff HEAD@{24} HEAD@{1}`

### Timed Reference:

Every entry in reference log has a timestamp associated with it. We can filter reflog entries by the time/date by using time qualifier like

- 1 day
- 3 minutes ago
- Yesterday
- Fri 12 feb 2023 14:06:21 – 0800

`>git reflog master@{one week ago}`

`>git checkout bugfixes@{2 days ago}`

`>git diff main@{0} main@{yesterday}`

### Rescuing Lost commit with reflog:

When you hard reset any commit, it is still present in Reflogs

Use `git reset --hard master@{1}`

To remove it from reflog too

Undoing a rebase/reflog: