GIt is used for:

* Tracking code changes
* Tracking who made changes
* Coding collaboration

What does Git do?

* Manage projects with **Repositories**
* **Clone** a project to work on a local copy
* Control and track changes with **Staging** and **Committing**
* **Branch** and **Merge** to allow for work on different parts and versions of a project
* **Pull** the latest version of the project to a local copy
* **Push** local updates to the main project

Working with Git

* Initialize Git on a folder, making it a **Repository**
* Git now creates a hidden folder to keep track of changes in that folder
* When a file is changed, added or deleted, it is considered **modified**
* You select the modified files you want to **Stage**
* The **Staged** files are **Committed**, which prompts Git to store a **permanent** snapshot of the files
* Git allows you to see the full history of every commit.
* You can revert back to any previous commit.
* Git does not store a separate copy of every file in every commit, but keeps track of changes made in each commit!

Log in with git hub -

* git config --global user.name "Gihan4"
* git config --global user.email "roey.gihan@yahoo.com"

Once you have navigated to the correct folder, you can initialize Git on that folder:

You just created your first Git Repository!

git init

Then we check the Git status and see if it is a part of our repo:

git status

Now Git is **aware** of the file, but has not **added** it to our repository!

**Staged** files are files that are ready to be **committed** to the repository you are working on

git add index.html

The file should be **Staged**.

Now add all files in the current directory to the Staging Environment:

git add --all

Using --all instead of individual filenames will stage all changes (new, modified, and deleted) files.

Since we have finished our work, we are ready move from stage to commit for our repo.

Adding commits keep track of our progress and changes as we work. Git considers each commit change point or "save point". It is a point in the project you can go back to if you find a bug, or want to make a change.

When we commit, we should **always** include a **message**.

git commit -m "First release of Hello World!"

The commit command performs a commit, and the -m "message" adds a message.

Commit without the stage:

git commit -a -m

Sometimes, when you make small changes, using the staging environment seems like a waste of time. It is possible to commit changes directly, skipping the staging environment. The -a option will automatically stage every changed, already tracked file.

git status –short

M index.html

**Note:** Short status flags are:

* ?? - Untracked files
* A - Files added to stage
* M - Modified files
* D - Deleted files

To view the history of commits for a repository, you can use the log command:

Git log

Git clone <url>

Of the existing reposertry.

Takes the file to my computer.

But what if you just want to update your local repository, without going through all those steps?

pull is a combination of fetch and merge. It is used to pull all changes from a remote repository into the branch you are working on.

Token:

ghp\_cxfotJnkTs8sXZrkJs8mdIjP1ZYRUp391p7t

**Note:** branch -r is for remote branches only.

**Branch**: A branch is a parallel version of a repository, allowing you to work on different features, bug fixes, or experiments without affecting the main codebase (usually called the "master" or "main" branch). When you create a branch, it starts as an identical copy of the original branch, and you can make changes to it independently. Branches are commonly used for collaborative development, enabling multiple people to work on different tasks simultaneously. Once the changes made on a branch are complete, they can be merged back into the main branch.

**fork**: Forking a repository means creating a separate copy of an entire repository under your GitHub account. Forking is typically used when you want to contribute to someone else's project or create your own version of an existing project. Forking allows you to freely experiment and make changes without affecting the original project. You can make modifications to your forked repository, such as adding new features or fixing bugs, and then submit these changes as pull requests to the original repository to propose your modifications for inclusion.

To summarize:

- Branching is about creating different versions of the code within the same repository.

- Forking is about creating an independent copy of an entire repository under your account.

Both branching and forking are powerful mechanisms that facilitate collaboration and code management in GitHub, but they serve different purposes.

We are working in our local repository, and we do not want to disturb or possibly wreck the main project.

So we create a new branch:

git branch hello-world-images

Now we created a new branch called "hello-world-images"

Let's confirm that we have created a new branch:

git branch

hello-world-images

\* master

the \* beside master specifies that we are currently on that branch.

git checkout hello-world-images

Now we have moved our current workspace from the master branch, to the new branch

Switch to the main branch and then you can merge branches.

git merge emergency-fix

git fetch – takes the current state of the remote repository and syncs it to my local git. In all of the branches. But it doesn’t pull the commits!

you still need to do pull to take the changes.

git-remote - Manage set of tracked repositories

-v show remote url after name

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

1. **git remote update**: This command updates all your remote tracking branches with the latest changes from the remote repository. It fetches the latest commits, branches, and tags from the remote repository.
2. **git fetch**: This command fetches the latest changes from the remote repository without merging them into your current branch. It updates your remote tracking branches, allowing you to see the changes made in the remote repository.
3. **git pull**: This command fetches the latest changes from the remote repository and automatically merges them into your current branch. It is equivalent to running **git fetch** followed by **git merge**.

In Git, the term "tree" is used to refer to the fundamental data structure that represents the contents of a directory. There are three types of trees in Git: the working tree, the index (or staging area), and the commit tree (or object tree).

1. Working Tree:

The working tree is the current state of your project's files and directories on your local machine. It's the directory structure you see and interact with when you're working on your project. Any modifications or additions you make to the files in this tree are not yet tracked by Git.

For example, let's say you have a Git repository with a file named `example.txt` in the working tree. If you make changes to this file, such as adding new content or modifying existing content, those changes will be reflected in the working tree until you stage and commit them.

2. Index (Staging Area):

The index, also known as the staging area, acts as a middle ground between the working tree and the commit tree. It's a snapshot of the changes you want to include in the next commit. When you make modifications to files in the working tree, you need to explicitly add those changes to the index before they can be committed.

For example, let's continue with the previous scenario. After making changes to `example.txt` in the working tree, you can use the `git add` command to add those changes to the index. The file is now staged and ready to be committed.

3. Commit Tree (Object Tree):

The commit tree, also known as the object tree or the Git history, represents the series of commits that form the history of your project. Each commit points to a snapshot of the entire project at a specific point in time, including the relevant metadata such as the author, timestamp, and a reference to its parent commit(s).

For example, let's say you commit the changes made to `example.txt` from the staging area. Git creates a new commit object that represents the snapshot of the project at that particular moment. This commit is then added to the commit tree, becoming the latest commit in the project's history.

Overall, these three trees work together to manage the version control of your project in Git. The working tree holds your current project files, the index serves as a staging area for changes to be committed, and the commit tree represents the history of your project with a series of snapshots.

Git log -p

Retrieves the last committed items with info of what has changed.

Q to exit view.

<git help log> a helpful manual.

So after git log I see all of these different recent commits. So what if I want to get back to a specific commit?

Git reset!

$ git reset <hashtag> c12344634

And if you want to change the order of the commits of the text you can do it with <git rebase> which will lead you to a vim.

To add a new branch and to switch to it:

$ git switch -c newbranch

HEAD- refers to what is the current reality.

Merge conflict is when you try to merge the new branch with the added changes, but meanwhile the main branch was already changed. To the conflict has to be resolved. You go to the conflicted file by nano and see that there is a conflict with HEAD(main) and the new branch.

When you clone a repository with git clone, it automatically creates a remote connection called origin pointing back to the cloned repository. This is useful for developers creating a local copy of a central repository, since it provides an easy way to pull upstream changes or publish local commits. This behavior is also why most Git-based projects call their central repository origin.

The --rebase option can be used to ensure a linear history by preventing unnecessary merge commits. Many developers prefer rebasing over merging, since it’s like saying, "I want to put my changes on top of what everybody else has done." In this sense, using git pull with the --rebase flag is even more like svn update than a plain git pull.

Merge –

Incorporates changes from the named commits (since the time their histories diverged from the current branch) into the current branch. This command is used by **git pull** to incorporate changes from another repository and can be used by hand to merge changes from one branch into another.

Assume the following history exists and the current branch is "master":

A---B---C topic

/

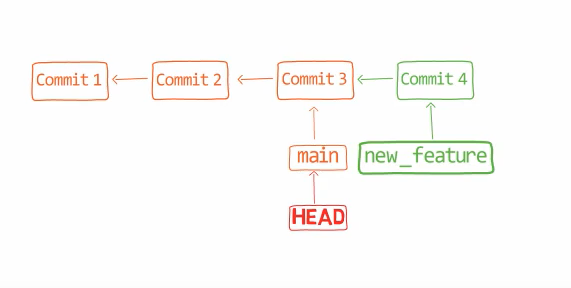
D---E---F---G master

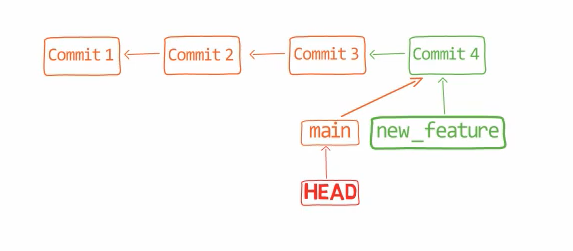
Then "git merge topic" will replay the changes made on the topic branch since it diverged from master (i.e., E) until its current commit (C) on top of master, and record the result in a new commit along with the names of the two parent commits and a log message from the user describing the changes. Before the operation, ORIG\_HEAD is set to the tip of the current branch (C).

A---B---C topic

/ \

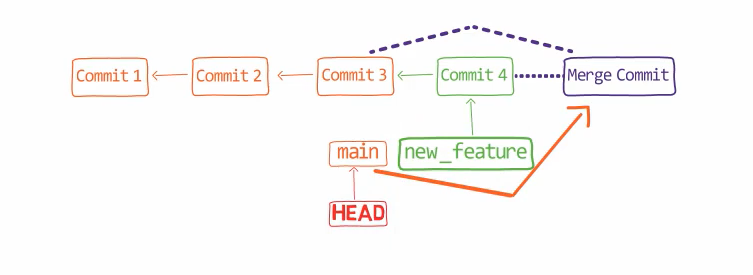
D---E---F---G---H master

Fast forward merge – alters the pointer 



So, if I want to merge without t he fast forward, to merge it completely, and not just to alter the pointer:

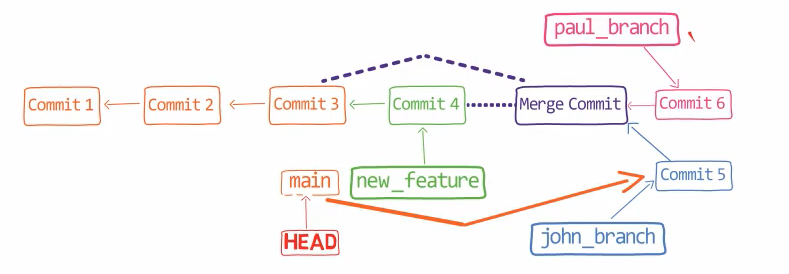
$ merge new\_branch –no—ff <-m “ ”>



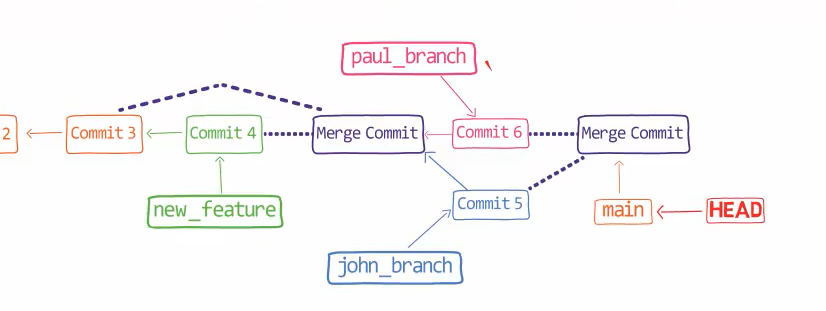
It created a new commit for us “merge commit”

A regular <git merge branch> is in default fast forward.

$ git merge john\_branch



so now if Paul also wants to merge his branch to the main branch, the fast forward merge wont do, because commit 6 is not in line with the tree. So the merge will have to make a branch commit.



3-way merge:

1. Locate the common ancestor – merge base

the first commit that is reachable from both branches.

2. Calculate diffs:

\*merge base -> first branch

\*merge base -> second branch

(generate patches).

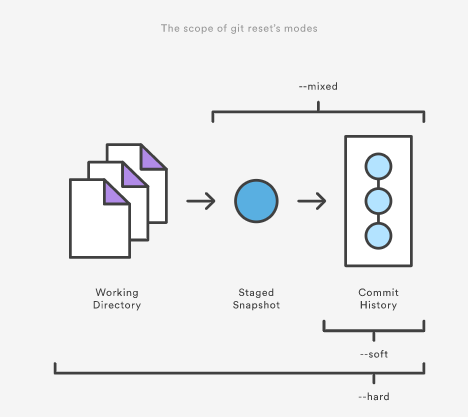
3. apply both patches together.

So for 1 the merge base is “merge commit”

Git reset is a command used in Git version control to move the current branch pointer to a specific commit, effectively changing the state of the repository

git reset, moves both the HEAD and branch refs to the specified commit.

When git checkout only moves the HEAD the git reset moves the main branch and the HEAD.



There are 3 types of reset:

--hard – this is the most direct. All the trees are updates to that specific commit. Any previously pending changes to the Staging Index and the Working Directory gets reset to match the state of the Commit Tree. This means any pending work that was hanging out in the Staging Index and Working Directory will be lost.

What we didn’t commit will be lost.

--mixed – this is the default mode. The Staging Index is reset to the state of the specified commit. Any changes that have been undone from the Staging Index are moved to the Working Directory. Let us continue.

What we didn’t add will be lost

The Staging Index has been reset and the pending changes have been moved into the Working Directory. Compare this to the --hard reset case where the Staging Index was reset and the Working Directory was reset as well, losing these updates.

--soft – only touches the commit history. It doesn’t care about the staging and the working directory.

1. **Soft Reset:** **git reset --soft <commit>**: This form of reset moves the branch pointer to the specified commit, but keeps the changes made after that commit in the staging area (index). It effectively "undoes" the commits after the specified commit while preserving the changes in the working directory. You can then modify the changes in the staging area and create a new commit.
2. **Mixed Reset:** **git reset --mixed <commit>** (default): This is the default behavior of **git reset** if you don't specify a reset type. It moves the branch pointer to the specified commit and resets the staging area (index) to match the commit, effectively unstaging any changes made after the specified commit. The changes made after the specified commit are preserved in the working directory, allowing you to review the changes before committing them again.
3. **Hard Reset:** **git reset --hard <commit>**: This is the most powerful and potentially dangerous form of reset. It moves the branch pointer to the specified commit and resets both the staging area and the working directory to match the commit. This means any changes made after the specified commit are permanently discarded. It's essential to exercise caution when using **git reset --hard** as it can result in data loss.