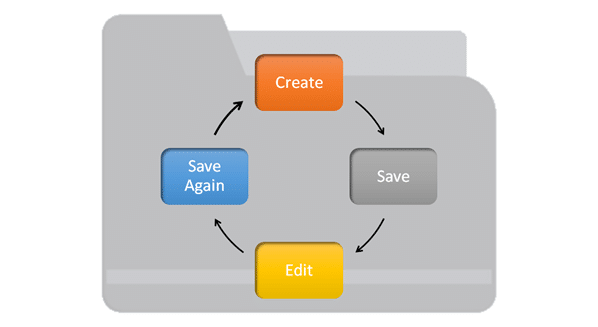
**Overview of Git**

Be it a designer, a creator, or a developer, all deal with projects and files every day. Our primary work cycle revolves around creating a file, saving it, editing or making required changes, and saving it again. In this Git tutorial for beginners, we will be focusing on Git basics and thereby understanding the life of a developer.  


[***Git***](https://intellipaat.com/blog/what-is-git/) is a small yet very efficient version control tool. It helps both programmers and non-programmers keep track of the history of their project files by storing different versions of them.

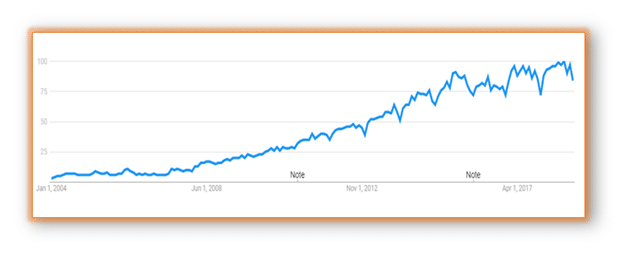
Git has become significantly popular among developers in the last few years in such a way that if one is part of the developers’ world, it is expected that he/she knows Git.

You must have come across the term ‘Git’ many times already, but don’t worry if you have never used Git because, in this tutorial, we will guide you through:

* [What is Git?](https://intellipaat.com/blog/tutorial/devops-tutorial/git-tutorial/#_What_is_git)
* [Why is Git used?](https://intellipaat.com/blog/tutorial/devops-tutorial/git-tutorial/#_Why_is_git)
* [Git Life Cycle](https://intellipaat.com/blog/tutorial/devops-tutorial/git-tutorial/#_Git_life)
* [Git Installation](https://intellipaat.com/blog/tutorial/devops-tutorial/git-tutorial/#_Git_install)
* [Environment Setup](https://intellipaat.com/blog/tutorial/devops-tutorial/git-tutorial/#_setup)
* [Common Git Commands: Local and Remote](https://intellipaat.com/blog/tutorial/devops-tutorial/git-tutorial/#_command)
  + [Local Commands: init, status, add, commit, and clone](https://intellipaat.com/blog/tutorial/devops-tutorial/git-tutorial/#_Local)
  + [Remote Commands: remote, push, and pull](https://intellipaat.com/blog/tutorial/devops-tutorial/git-tutorial/#_Remote)
* [Branches in Git: checkout and branch](https://intellipaat.com/blog/tutorial/devops-tutorial/git-tutorial/#_branch)
* [Merging in Git](https://intellipaat.com/blog/tutorial/devops-tutorial/git-tutorial/#_marging)
* [Rebasing in Git](https://intellipaat.com/blog/tutorial/devops-tutorial/git-tutorial/#_rebash)
* [Merging Conflicts and Rebasing Conflicts](https://intellipaat.com/blog/tutorial/devops-tutorial/git-tutorial/#_merr)
  + [Solving the Conflicts](https://intellipaat.com/blog/tutorial/devops-tutorial/git-tutorial/#_solve)
* [Git Workflows](https://intellipaat.com/blog/tutorial/devops-tutorial/git-tutorial/#_workflow)

**What is Git?**

Git helps developers keep track of the history of their code files by storing them in different versions on its own server repository, i.e., [***GitHub***](https://intellipaat.com/blog/what-is-github-github-tutorial/). Git has all the functionality, performance, security, and flexibility that most of the development teams and individual developers need.

The below-given graph depicts the rate of popularity that Git has gained over the years (2004–2017):  
  
Now, let us go ahead in this Git tutorial and learn why Git is used and what the factors are that make Git so unique.

**Why Git Version Control?**

Below are some of the facts that make Git so popular:

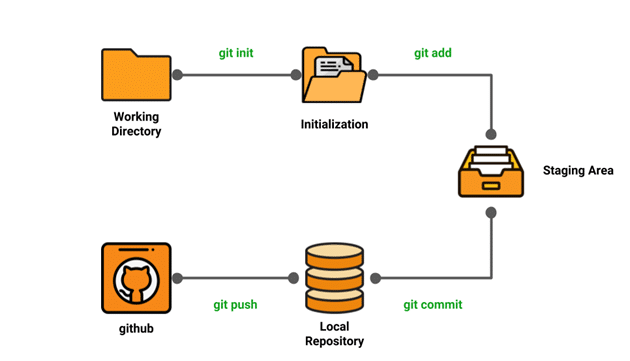
* **Works offline**: Git provides users with very convenient options such as allowing them to work both online and offline. With other version control systems like SVN or CVS, users need to have access to the Internet to connect to the central repository.
* **Undoes mistakes**: Git allows us to undo our commands in almost every situation. We get to correct the last commit for a minor change, and also we can revert a whole commit for unnecessary changes.
* **Restores the deleted commits**: This feature is very helpful while dealing with large projects when we try out some experimental changes.
* **Provides security**: Git provides protection against secret alteration of any file and helps maintain an authentic content history of the source file.
* **Guarantees performance**: Being a distributed version control system, it has an optimized performance due to its features like committing new changes, branching, merging, comparing past versions of the source file, etc.
* **Offers flexibility**: Git supports different nonlinear development workflows, for both small and large projects.

**How does Git work?**

Now that we have learned about Git in this Git tutorial for beginners, we will move ahead and discuss how Git works. Let us start learning the Git basics, starting with the life cycle of Git.

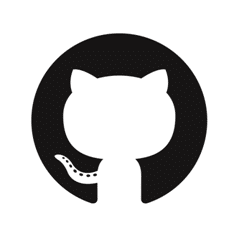
**Git Life Cycle**

* **Local working directory**: The first stage of a Git project life cycle is the local working directory where our project resides, which may or may not be tracked.



* **Initialization:** To initialize a repository, we give the command git init. With this command, we will make Git aware of the project file in our repository.
* **Staging area:** Now that our source code files, data files, and configuration files are being tracked by Git, we will add the files that we want to commit to the staging area by the git add command. This process can also be called indexing. The index consists of files added to the staging area.
* **Commit:** Now, we will commit our files using the git commit -m ‘our message’ command.

We have successfully committed our files to the local repository. But how does it help in our projects? The answer is, when we need collaborative projects, files may have to be shared with our team members.

This is when the next stage of the Git life cycle occurs, i.e., in GitHub, we publish our files from the local repository to the remote repository. And how do we do that? We do that by using the git push command.  
So far, we have discussed the life cycle of a Git project, where we came across some commands such as git init, git add, git commit, git push, etc. Further in this Git tutorial, we will be explaining these common commands in detail.

But before that, let’s go through the installation and configuration of Git.

**Installing Git**

[***Git installation on Windows***](https://intellipaat.com/mediaFiles/2018/12/GIT-INSTALLATION-WINDOWS.pdf).

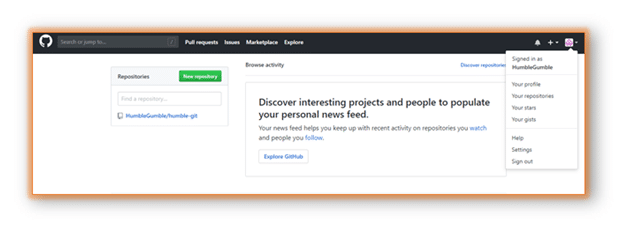
We have now successfully installed Git on our system. Let us set up the environment on our system in the next section of this Git for beginners tutorial.

**Environment Setup**

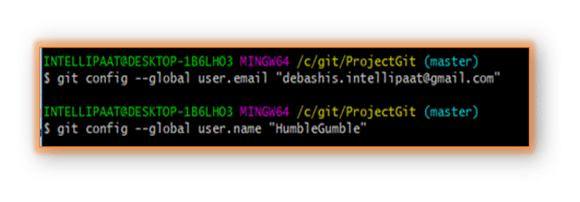
* Create a GitHub account
* Configure Git
* Create a local repository

**Creating a GitHub account:**

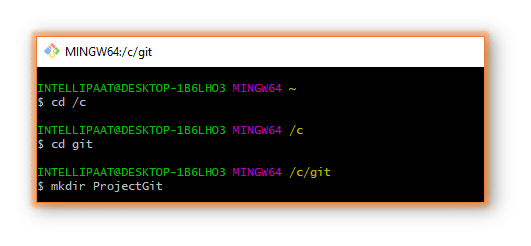
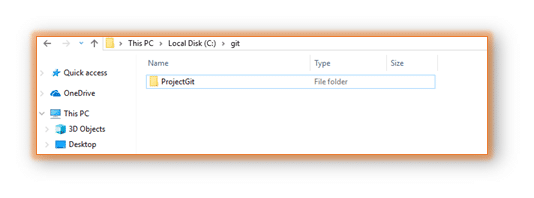
* Go to https://Github.com
* Create a GitHub account
* Login

After accomplishing these steps, our GitHub page should look like this:  


**Configuring Git:**

To tell Git who we are, run the following two commands:  


**Creating a local repository:**

To begin with, open a terminal and move to where we want to place the project on our local machine using the cd (change directory) command. For example, if we have a ‘projects’ folder on our desktop, we would do something like below:  
We can go to the Git folder and see the new directory that we have just created.  
Finally, we are all set to start working on [Git commands](https://intellipaat.com/blog/tutorial/devops-tutorial/git-commands/).

***Learn more about DevOps in this***[***DevOps Training in Sydney***](https://intellipaat.com/devops-certification-training-sydney/)***to get ahead in your career!***

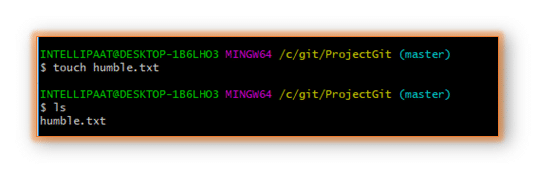
**Common Git Commands**

We will divide the common Git commands into two primary categories:

* **Local**: git init, git touch, git status, git add, git commit, and git rm
* **Remote**: git remote, git pull, and git push

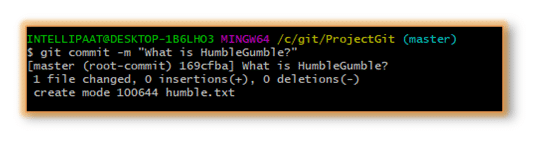
Next in Git and GitHub tutorial, we will discuss the local Git commands.

**Local Git Commands**

* **git init**: We use the git init command to initialize a Git repository in the root of a folder.
* **git touch**: To add files to a project, we can use git touch. Let us see how we can add a file to the Git repository we have just created  
  **Step 1**: Create a new file with the command touch  
  **Step 2**: See the files present in our master branch  
  
* **git status**: After creating a new file, we can use the git status command and see the status of the files in the master branch.  
  
* **git status**: Now, we will add the humble.txt file to the staging environment by using git add, before going ahead to the staging to see the change using git status.



* **git commit**: Now we will use the git commit command as shown below:

Thus, we have successfully created our first commit.

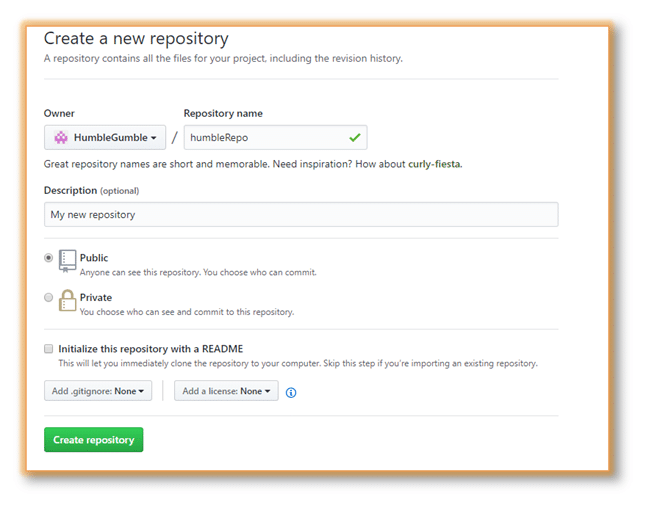
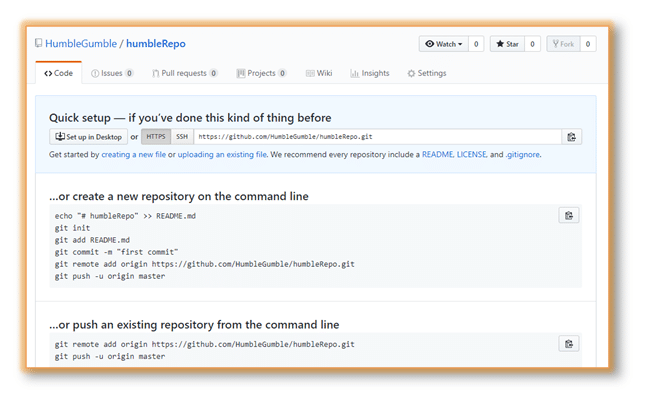
* **Git clone:**



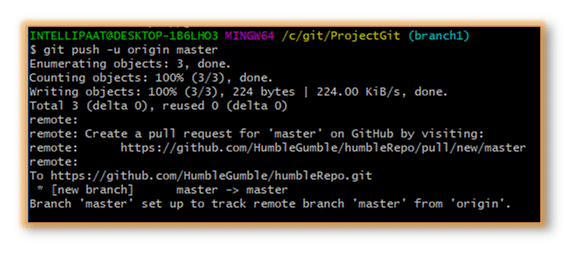
**Remote Commands**

Let us now discuss the remote commands such as remote, push, and pull in Git. For that, we will create a new repository in our GitHub account. How do we do that? Follow the below steps:

**Step 1**: Go to the GitHub account  
**Step 2**: Create a new repository

Once we are done with filling up the new repository form, we should land on a page like the following:  
Now, we are ready to operate remote commands in our repository that we have just created.

To push an existing repository to a remote repository from a command line, follow the commands given below.

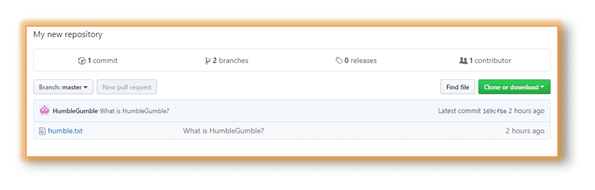
* **git remote:**After this, we will provide the following command:
* **git push**

git push origin EnterBranchName

Don’t get confused by the word ‘origin’. What exactly is this command referring to?

Instead of writing the whole Git URL, like **git push Git@Github.com:Git/Git.Git ourbranchname**, we can just use the following command, assuming that our brach name is ‘branch1’:

git push origin branch1

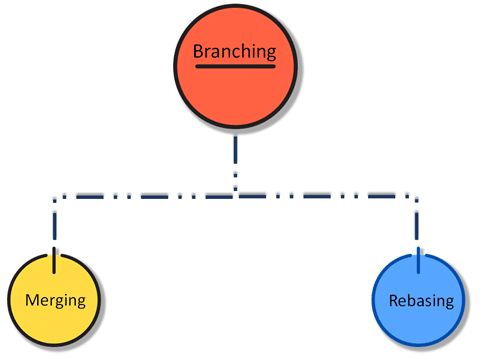
Now, let us look at our repository:  


* **git pull**

When it comes to syncing a remote repository, the pull command comes in handy. Let us take a look at the commands that can leads to the pulling operation in Git.

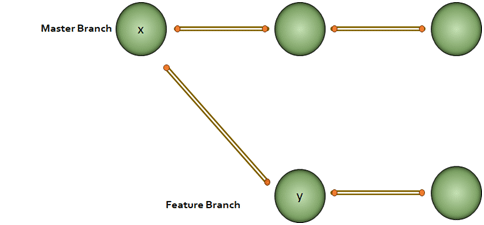
* remote origin
* pull master

We will use these as shown below:  
Now that we have learned the workflow of Git with the help of the common Git commands, let us take a look at branching, merging, and rebasing and also at how and when to use them.

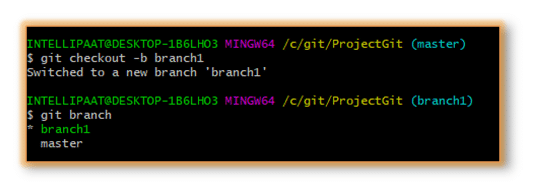


**Branching in Git**

We are almost familiar with various Git commands now. Let’s take a step further and discuss one of the most important operations in Git, namely, branching. In branching, we mainly make use of the **git checkout** and **git branch** commands.

  
Every node in the figure above represents commits. We have to note that for the ‘y’ node in the feature branch, ‘x’ is the base.

Why would we do branching in the first place? Say, we want to modify something but don’t want to make any change in the main project. This is when we make a branch out of the master branch, i.e., if we want to create a new branch to add a feature to the main project, we will make a branch out of it with the help of the following steps:

**Step 1**: We will run **git checkout -b <new branch name>**  
**Step 2**: Then, we will use the git branch command to confirm that our branch is created  
We can see from the above image that we have created a branch called ‘branch1’ and we automatically got landed in the new branch. Again, just to see which branch we are currently in, we run another command, git branch.

**Note:** The **\*** mark before the branch name shows that it is the current branch.

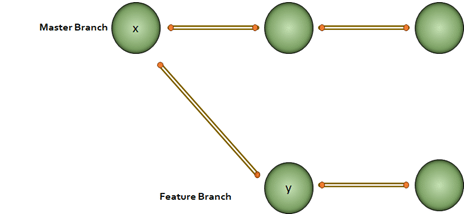
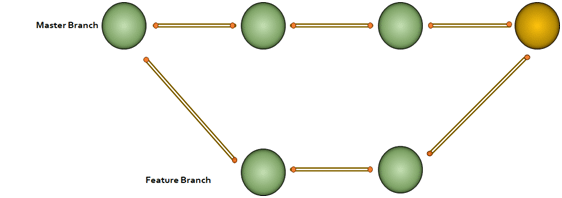
Now, how to change a branch to the master branch? For that, we use the following command:

git checkout master

**Merging in Git (git checkout, git add, git log, git merge, merging conflicts, and rebasing)**

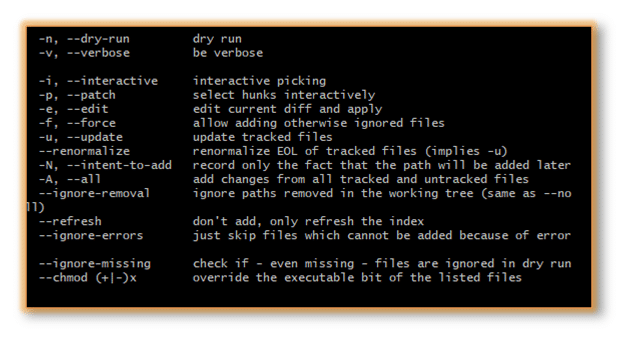
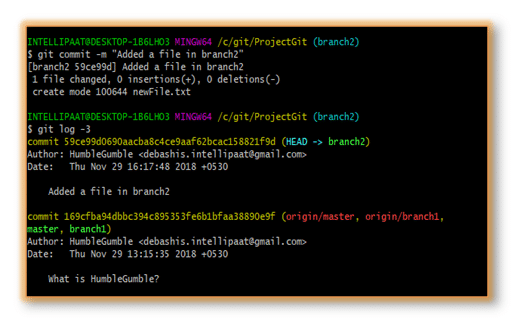
Now that we have learned how to create a branch and work on it, let us take a look at the merge feature in Git by merging the branch we created to the master branch.

Let’s take the above example. Say, we have a master branch and a feature branch.

The merge commit represents every change that has occurred on the feature branch since it got branched out from the master.  
**Note**: Even after merging, we can go on with our work on both the master and the feature branches independently.

Let us see how to perform merging:

**Step 1**: Create a new branch called ‘branch2’  
**Step 2**: Create a new file in the branch  
**Step 3**: Add changes from all tracked and untracked files

**Note**: Refer to the following **git add** attributes  
In our case, we have given the command as **git add -A** and after that, we will commit one sentence as shown below:  
**Step 4**: Check the last three logs by running the command: **git log -3**  
We have created another branch on our master branch. Now, we will see how to perform **merging**.

Let us get inside the master branch using the following command:

git checkout master

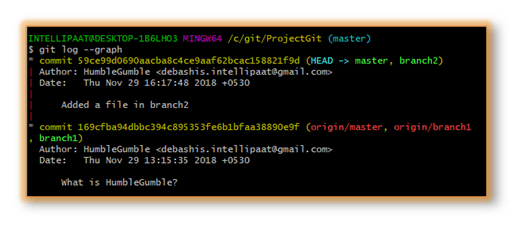
After that, we will perform merging with the help of the below command:

git merge branch2

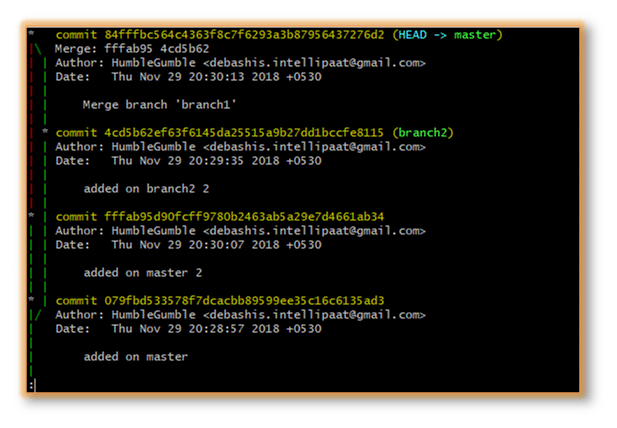


Now, we have successfully merged the feature branch into the master branch. Let us take a look at how it looks inside the master branch using the following command:

git log --graph

We can see the graph in the above image on the left. But **why the graph is linear?**

In our ProjectGit master branch, we did branching and committing. But after the branching, the master branch had not encountered any more commits. Hence, after merging, we have a linear graph.

Let us perform**git log –graph** with more than one commits in both master and feature branches that we have created.  
Here, we can see the graph with commits on both master and feature branches. The red part of the graph indicates the merging operation.

**Advantages of merging:**

* Merging allows parallel working in collaborative projects.
* It saves the time that is consumed by manual merging.

**The disadvantage of merging:**

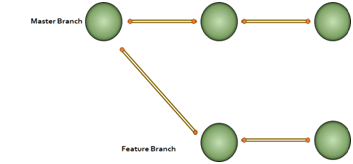
* Merging conflicts may occur while merging branches.

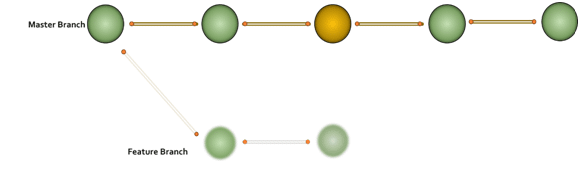
Now that we have successfully learned to branch and merge with Git and GitHub, further in this best Git tutorial, let us look at yet another important Git operation, i.e., rebasing.

**Git Rebase**

When our project becomes relatively large, the commit log and the history of the repository become messy. Here, we use rebasing. Rebasing will take a set of commits, copy them, and store them outside our repository. This helps us maintain a linear sequence of commits in our repository.

Let us take the same example. Here, we will rebase the master branch and see what happens.

**Note**: In rebasing, the base of the feature branch gets changed and the last commit of the master branch becomes the new base of the feature branch.

Now, let us perform rebasing in Git Bash.



**The advantage of rebasing:**

* Rebasing provides a cleaner project history.

**The disadvantage of rebasing:**

* In a collaborative workflow, re-writing the project history can be potentially catastrophic.

Now that we understood what branching, merging, and rebasing are, next in this Git tutorial, we will see where to use merging and rebasing.

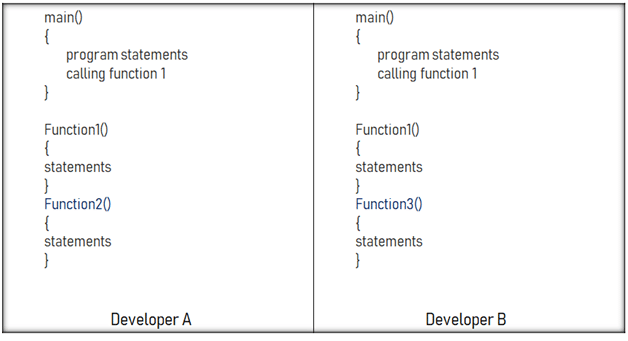
**The golden rule of Git merging and rebasing:**

* **When to use git merge**: We can use git merge while working on the public branches.
* **When to use git rebase**: We will use git rebase while working on the local branches.

As we have already mentioned earlier in this Git commands tutorial, one major disadvantage of merging is merge conflicts, which can be backbreaking for a team project. Let us understand how merge conflicts occur and how to resolve them.

**Git Merge Conflict and Rebase Conflict**

This drawback of the merging operation in Git can be explained with a simple example shown below:

Say, we have two branches of the master branch, branch1 and branch2, and two developers are working on these two branches independently but on the same code file.

As we have seen in the above image, the developers have made the below-mentioned modifications:

* Developer A added a function called ‘function2’ to the main code.
* Developer B added a different function called ‘function3’ to the same code file.

How to merge these two modifications? That is where the merging conflict occurs.

Similarly, git rebase also exhibits conflicts

**Solving the Conflicts**

We can manually resolve the merge conflict using the merging tool. We can use **file locking**which doesn’t allow different developers to work on the same piece of code simultaneously. It helps avoid merge conflicts but slows down the development process.

The rebasing conflict can also be solved with the help of the Git merging tool.

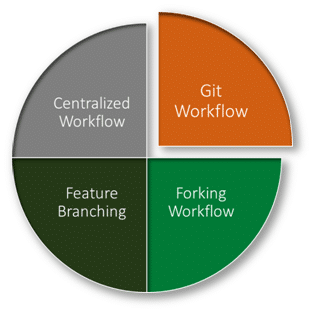
Now, let us look at a few Git workflows briefly before we end this Git tutorial so that we get an idea of which Git workflow to choose for our team project.

**Git Workflows**

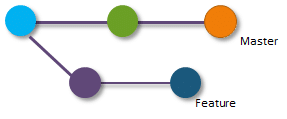
In this section, we will get introduced to different workflow options available in Git. Once we get an idea of these workflows, we can choose the right one for our team project.

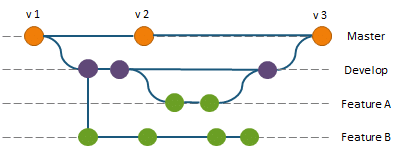
  
**But, why is it important to choose the right Git workflow?**

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

Now in this Git tutorial, let us look at the different Git workflows:  


* **Centralized workflow:**In the Git centralized workflow, only one development branch is there, which is called ‘master’ and all changes are committed into this single branch.
* **Feature branching workflow:**In the feature branching workflow, feature development takes place only in a dedicated feature branch. The below-given image in this Git tutorial depicts the feature branching workflow:



* **Git workflow**: Instead of a single master branch, the Git workflow uses two branches. Here, the master branch stores the official release history, whereas the second ‘develop’ branch acts as an integration branch for features. The below-given image depicts the Git workflow:
* **Forking workflow:**In the case of the forking workflow, the contributor has two Git repositories: one private local repository and the other public server-side repository.

This brings us to the end of the Git tutorial. In this Git tutorial, we have gone through the version control systems and its different types, the basics of Git, terminologies related to Git, Git installation in Windows, Linux, and on macOS systems, setting up and working on the GitHub repository, and various commands used in Git. We have also discussed some important operations in Git toward the end of this Git tutorial.

**Git and Its Popularity**

Git is one of the most important version control systems that has experienced a significant growth rate and popularity over the years. Git plays a key role in both developers’ and non-developers’ world. If we are part of the software developers’ world, it is expected that we know Git. So, having a good grasp of Git is very beneficial for us to get into a lucrative career.

