SVN: Apache Subversion is a software versioning and revision control system distributed as open source under the Apache License. Software developers use Subversion to maintain current and historical versions of files such as source code, web pages, and documentation

* Track changes to your code base (i.e. who changed what), and roll back to previous revisions
* Collaborate with your teammates by having the version control system merge your changes with changes made by other developers on the same file(s)
* Create branches of your code, allowing you to maintain different versions of your project in parallel

**GIT:**  Git is an open-source version control system,but what does that mean? When developers create something (an app, for example), they make constant changes to the code, releasing new versions up to and after the first official (non-beta) release.

Version control systems keep these revisions straight, storing the modifications in a central repository. This allows developers to easily collaborate, as they can download a new version of the software, make changes, and upload the newest revision. Every developer can see these new changes, download them, and contribute.

**difference between GIT and SVN**

a)      Git is less preferred for handling extremely large files or frequently changing binary files while SVN can handle multiple projects stored in the same repository.

b)      GIT does not support ‘commits’ across multiple branches or tags.  Subversion allows the creation of folders at any location in the repository layout.

c)        Gits are unchangeable, while Subversion allows committers to treat a tag as a branch and to create multiple revisions under a tag root

**advantages of using GIT:**

a)      Data redundancy and replication

b)      High availability

c)       Only one.git directory per repository

d)      Superior disk utilization and network performance

e)      Collaboration friendly

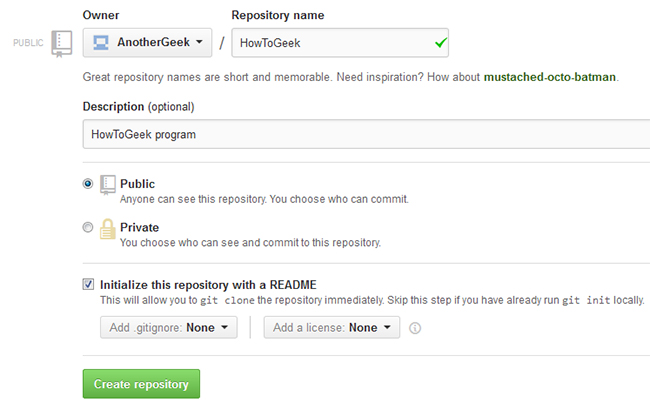
f)       Any sort of projects can use GIT

**Why GIT better than Subversion:**

**.** GIT is an open source version control system; it will allow you to run ‘versions’ of a project, which show the changes that were made to the code overtime also it allows you keep the backtrack if necessary and undo those changes.  Multiple developers can checkout, and upload changes and each change can then be attributed to a specific developer.

**GITHUB:** GitHub is a web-based hosting service for version control using Git. It is mostly used for computer code. It offers all of the distributed version control and source code management functionality of Git as well as adding its own features.

**Repository:** A repository (usually abbreviated to “repo”) is a location where all the files for a particular project are stored. Each project has its own repo, and you can access it with a unique URL



**Forking a Repo:**

“Forking” is when you create a new project based off of another project that already exists. This is an amazing feature that vastly encourages the further development of programs and other projects. If you find a project on GitHub that you’d like to contribute to, you can fork the repo, make the changes you’d like, and release the revised project as a new repo. If the original repository that you forked to create your new project gets updated, you can easily add those updates to your current fork.



### Pull Requests: You’ve forked a repository, made a great revision to the project, and want it to be recognized by the original developers—maybe even included in the official project/repository. You can do so by creating a pull request. The authors of the original repository can see your work, and then choose whether or not to accept it into the official project. Whenever you issue a pull request, GitHub provides a perfect medium for you and the main project’s maintainer to communicate.

### 

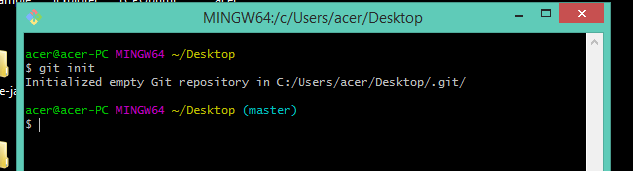
### CREATE ,UPLOAD,FIND FILES:

### 

**GIT:**

**What is a repository in GIT:** A repository contains a directory named .git, where git keeps all of its metadata for the repository. The content of the .git directory are private to git.

. In Git, to create a repository, create a directory for the project if it does not exist, and then run command “git init”. By running this command .git directory will be created in the project directory, the directory does not need to be empty.

****

**GIT CLONE:**

When you create a repository on GitHub, it exists as a *remote* repository. You can clone your repository to create a *local* copy on your computer and sync between the two locations.

This procedure assumes you have already [created a repository on GitHub](https://help.github.com/en/articles/creating-a-new-repository), or have an existing repository owned by someone else you'd like to contribute to.

1. On GitHub, navigate to the main page of the repository.

**Note:** If the repository is empty, you can manually copy the repository page's URL from your browser and skip to step four.

1. Under the repository name, click **Clone or download**.

Clone or download button

1. In the Clone with HTTPs section, click  to copy the clone URL for the repository.



1. Open Git Bash.
2. Change the current working directory to the location where you want the cloned directory to be made.
3. Type git clone, and then paste the URL you copied in Step 2.

$ git clone https://github.com/*YOUR-USERNAME*/*YOUR-REPOSITORY*

1. Press **Enter**. Your local clone will be created.
2. $ git clone https://github.com/*YOUR-USERNAME*/*YOUR-REPOSITORY*
3. > Cloning into `Spoon-Knife`...
4. > remote: Counting objects: 10, done.
5. > remote: Compressing objects: 100% (8/8), done.
6. > remove: Total 10 (delta 1), reused 10 (delta 1)

> Unpacking objects: 100% (10/10), done.

HOW TO PUSH TO REMOTE REPOSITORY ?

1. On your computer, move the file you'd like to upload to GitHub into the local directory that was created when you cloned the repository.
2. Open Git Bash.
3. Change the current working directory to your local repository.
4. Stage the file for commit to your local repository.
5. $ git add .

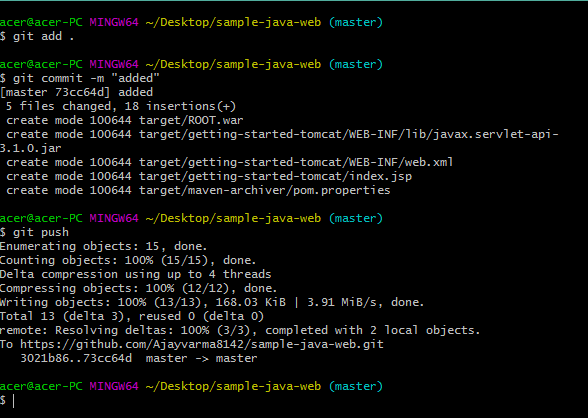
# Adds the file to your local repository and stages it for commit. To unstage a file, use 'git reset HEAD YOUR-FILE'.

1. Commit the file that you've staged in your local repository.
2. $ git commit -m "Add existing file"

# Commits the tracked changes and prepares them to be pushed to a remote repository. To remove this commit and modify the file, use 'git reset --soft HEAD~1' and commit and add the file again.

1. [Push the changes](https://help.github.com/en/articles/pushing-to-a-remote) in your local repository to GitHub.
2. $ git push origin your-branch

# Pushes the changes in your local repository up to the remote repository you specifi

****

**BRANCH:**

## In your github fork, you need to keep your master branch clean, by clean I mean without any changes, like that you can create at any time a branch from your master. Each time that you want to commit a bug or a feature, you need to create a branch for it, which will be a copy of your master branch.

When you do a pull request on a branch, you can continue to work on another branch and make another pull request on this other branch.

Before creating a new branch, pull the changes from upstream. Your master needs to be up to date.

$ git pull

Create the branch on your local machine and switch in this branch :

$ git checkout -b [name\_of\_your\_new\_branch]

Push the branch on github :

$ git push origin [name\_of\_your\_new\_branch]

When you want to commit something in your branch, be sure to be in your branch. Add -u parameter to set upstream.

You can see all branches created by using :

$ git branch -a

Which will show :

\* approval\_messages

master

master\_clean

Add a new remote for your branch :

$ git remote add [name\_of\_your\_remote] [name\_of\_your\_new\_branch]

Push changes from your commit into your branch :

$ git push [name\_of\_your\_new\_remote] [url]

Update your branch when the original branch from official repository has been updated :

$ git fetch [name\_of\_your\_remote]

Then you need to apply to merge changes, if your branch is derivated from develop you need to do :

$ git merge [name\_of\_your\_remote]/develop

Delete a branch on your local filesystem :

$ git branch -d [name\_of\_your\_new\_branch]

To force the deletion of local branch on your filesystem :

$ git branch -D [name\_of\_your\_new\_branch]

Delete the branch on github :

$ git push origin :[name\_of\_your\_new\_branch]

The only difference is the : to say delete, you can do it too by using github interface to remove branch : <https://help.github.com/articles/deleting-unused-branches>.

If you want to change default branch, it's so easy with github, in your fork go into Admin and in the drop-down list default branch choose what you want.

If you want create a new branch:

$ git branch <name\_of\_your\_new\_branch>

GIT MERGE:

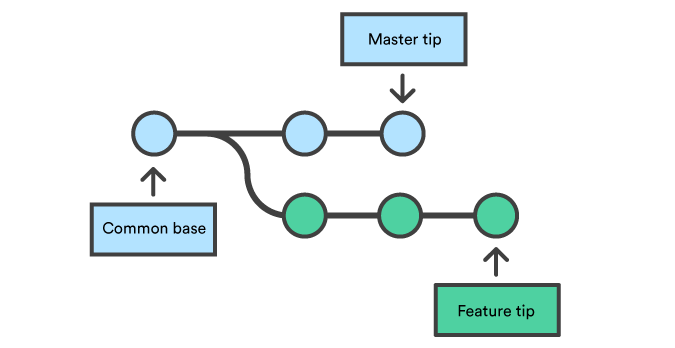
Merging is Git's way of putting a forked history back together again. The git merge command lets you take the independent lines of development created by git branch and integrate them into a single branch.

Note that all of the commands presented below merge into the current branch. The current branch will be updated to reflect the merge, but the target branch will be completely unaffected. Again, this means that git merge is often used in conjunction with git checkout for selecting the current branch and git branch -d for deleting the obsolete target branch.

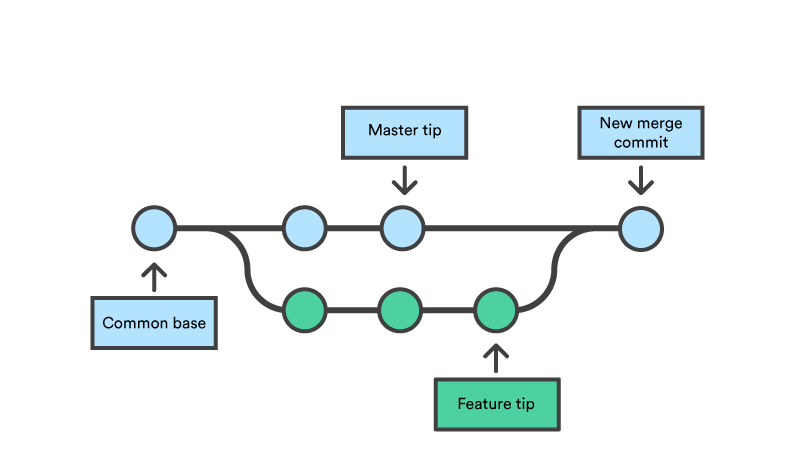
## How it works

Git merge will combine multiple sequences of commits into one unified history. In the most frequent use cases, git merge is used to combine two branches. The following examples in this document will focus on this branch merging pattern. In these scenarios, git merge takes two commit pointers, usually the branch tips, and will find a common base commit between them. Once Git finds a common base commit it will create a new "merge commit" that combines the changes of each queued merge commit sequence.

Say we have a new branch feature that is based off the masterbranch. We now want to merge this feature branch into master.



Invoking this command will merge the specified branch feature into the current branch, we'll assume master. Git will determine the merge algorithm automatically (discussed below).



Merge commits are unique against other commits in the fact that they have two parent commits. When creating a merge commit Git will attempt to auto magically merge the separate histories for you. If Git encounters a piece of data that is changed in both histories it will be unable to automatically combine them. This scenario is a version control conflict and Git will need user intervention to continue.

## Preparing to merge

Before performing a merge there are a couple of preparation steps to take to ensure the merge goes smoothly.

## Confirm the receiving branch

Execute git status to ensure that HEAD is pointing to the correct merge-receiving branch. If needed, execute git checkout <receiving> to switch to the receiving branch. In our case we will execute git checkout master.

## Fetch latest remote commits

Make sure the receiving branch and the merging branch are up-to-date with the latest remote changes. Execute git fetch to pull the latest remote commits. Once the fetch is completed ensure the master branch has the latest updates by executing git pull.

## Merging

Once the previously discussed "preparing to merge" steps have been taken a merge can be initiated by executing git merge <branch name> where <branch name> is the name of the branch that will be merged into the receiving branch.

## Fast Forward Merge

A fast-forward merge can occur when there is a linear path from the current branch tip to the target branch. Instead of “actually” merging the branches, all Git has to do to integrate the histories is move (i.e., “fast forward”) the current branch tip up to the target branch tip. This effectively combines the histories, since all of the commits reachable from the target branch are now available through the current one. For example, a fast forward merge of some-feature into master would look something like the following:

However, a fast-forward merge is not possible if the branches have diverged. When there is not a linear path to the target branch, Git has no choice but to combine them via a 3-way merge. 3-way merges use a dedicated commit to tie together the two histories. The nomenclature comes from the fact that Git uses three commits to generate the merge commit: the two branch tips and their common ancestor.

While you can use either of these merge strategies, many developers like to use fast-forward merges (facilitated through [rebasing](https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase)) for small features or bug fixes, while reserving 3-way merges for the integration of longer-running features. In the latter case, the resulting merge commit serves as a symbolic joining of the two branches.

Our first example demonstrates a fast-forward merge. The code below creates a new branch, adds two commits to it, then integrates it into the main line with a fast-forward merge.

# Start a new feature

git checkout -b new-feature master

# Edit some files

git add <file>

git commit -m "Start a feature"

# Edit some files

git add <file>

git commit -m "Finish a feature"

# Merge in the new-feature branch

git checkout master

git merge new-feature

git branch -d new-feature

This is a common workflow for short-lived topic branches that are used more as an isolated development than an organizational tool for longer-running features.

Also note that Git should not complain about the git branch -d, since new-feature is now accessible from the master branch.

In the event that you require a merge commit during a fast forward merge for record keeping purposes you can execute git merge with the --no-ffoption.

git merge --no-ff <branch>

This command merges the specified branch into the current branch, but always generates a merge commit (even if it was a fast-forward merge). This is useful for documenting all merges that occur in your repository.

## 3-way merge

The next example is very similar, but requires a 3-way merge because master progresses while the feature is in-progress. This is a common scenario for large features or when several developers are working on a project simultaneously.

Start a new feature

git checkout -b new-feature master

# Edit some files

git add <file>

git commit -m "Start a feature"

# Edit some files

git add <file>

git commit -m "Finish a feature"

# Develop the master branch

git checkout master

# Edit some files

git add <file>

git commit -m "Make some super-stable changes to master"

# Merge in the new-feature branch

git merge new-feature

git branch -d new-feature

Note that it’s impossible for Git to perform a fast-forward merge, as there is no way to move master up to new-feature without backtracking.

For most workflows, new-feature would be a much larger feature that took a long time to develop, which would be why new commits would appear on master in the meantime. If your feature branch was actually as small as the one in the above example, you would probably be better off rebasing it onto master and doing a fast-forward merge. This prevents superfluous merge commits from cluttering up the project history.

## Resolving conflict

If the two branches you're trying to merge both changed the same part of the same file, Git won't be able to figure out which version to use. When such a situation occurs, it stops right before the merge commit so that you can resolve the conflicts manually.

The great part of Git's merging process is that it uses the familiar edit/stage/commit workflow to resolve merge conflicts. When you encounter a merge conflict, running the git status command shows you which files need to be resolved. For example, if both branches modified the same section of hello.py, you would see something like the following:

On branch master

Unmerged paths:

(use "git add/rm ..." as appropriate to mark resolution)

both modified: hello.py

## How conflicts are presented

When Git encounters a conflict during a merge, It will edit the content of the affected files with visual indicators that mark both sides of the conflicted content. These visual markers are: <<<<<<<, =======, and >>>>>>>. Its helpful to search a project for these indicators during a merge to find where conflicts need to be resolved.

here is some content not affected by the conflict

<<<<<<< master

this is conflicted text from master

=======

this is conflicted text from feature branch

>>>>>>> feature branch;

Generally the content before the ======= marker is the receiving branch and the part after is the merging branch.

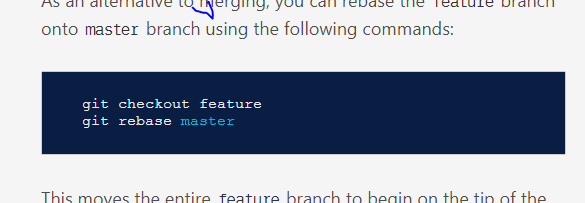
Once you've identified conflicting sections, you can go in and fix up the merge to your liking. When you're ready to finish the merge, all you have to do is run git add on the conflicted file(s) to tell Git they're resolved. Then, you run a normal git commit to generate the merge commit. It’s the exact same process as committing an ordinary snapshot, which means it’s easy for normal developers to manage their own merges.

Note that merge conflicts will only occur in the event of a 3-way merge. It’s not possible to have conflicting changes in a fast-forward merge.

REBASE:

In Git, the **rebase** command integrates changes from one branch into another. It is an alternative to the better known "merge" command.

Most visibly, rebase differs from merge by rewriting the commit history in order to produce a straight, linear succession of commits.



### Interactive Rebasing

Interactive rebasing gives you the opportunity to alter commits as they are moved to the new branch. This is even more powerful than an automated rebase, since it offers complete control over the branch’s commit history. Typically, this is used to clean up a messy history before merging a feature branch into master.

To begin an interactive rebasing session, pass the i option to the git rebase command:

git checkout feature

git rebase -i master

This will open a text editor listing all of the commits that are about to be moved:

pick 33d5b7a Message for commit #1

pick 9480b3d Message for commit #2

pick 5c67e61 Message for commit #3

This listing defines exactly what the branch will look like after the rebase is performed. By changing the pick command and/or re-ordering the entries, you can make the branch’s history look like whatever you want. For example, if the 2nd commit fixes a small problem in the 1st commit, you can condense them into a single commit with the fixup command:

pick 33d5b7a Message for commit #1

fixup 9480b3d Message for commit #2

pick 5c67e61 Message for commit #3

When you save and close the file, Git will perform the rebase according to your instructions, resulting in project history that looks like the following:

GIT CHERRY-PICK:

**Cherry picking** in **Git** is designed to apply some commit from one branch into another branch. It can be done if you eg. made a mistake and committed a change into wrong branch, but do not want to merge the whole branch. You can just eg. revert the commit and **cherry**-**pick** it on another branch.

Let's [cherry-pick](https://git-scm.com/docs/git-cherry-pick) a commit from the development and add it to feature branch by using the commit hash.

We want to pick a commit C from the master. There are a few steps you should follow to get the job done:

1. Obtain the commit hash. You can do this in two ways:
   * By typing git log --oneline, to get the log of your commits history. Make sure you are on the correct branch: git checkout master.
   * Selecting the commit hash from the GitHub page.
2. [Checkout to the branch](https://kolosek.com/git-branches/) that you want to insert the commit into, in our case this is the feature branch: git checkout feature.
3. Cherry-pick the commit: git cherry-pick C.

If you run this, will see that our cherry-picked commit at the top of the feature branch. It will have a new and different commit hash