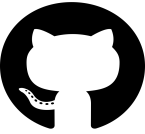
GitHub 

TABLE OF CONTENTS

1. Introduction
   1. What is GitHub?
   2. The History of GitHub
   3. The Benefits of GitHub
2. How to teach GitHub easily?
3. Git-GitHub Cheat Sheet
4. Git-GitHub Work Flow
5. Some Examples
6. Real work environment usage of GitHub
7. Additional Resources
8. Glossary

**INTRODUCTION**

**What is GitHub?**

**GitHub** is a United States-based global company that provides hosting for software development version control system using Git. It hosts your source code projects in a variety of different programming languages and keeps track of the various changes made to every iteration.

**The History of GitHub**

Github is one of the most essential tools for developers of all levels. If we want to learn GitHub, first we should learn Git. Git is a version control system, open source project originally developed in 2005 by Linus Torvalds, the famous creator of the Linux operating system kernel. But what is version control system? Think you have a project saved to your desktop and one day your system crashes. Your works just gone forever. Like you, Linus Torvalds worried about this and developed Git. Its is a version control system. You can upload your work and keeping a track of all modifications. Whenever you need , you can arrive all your works modifications. And there is another version control systems like Subversion, Bitbucket, Mercurial.



**The Benefits of GitHub**

1. It provides developers and researchers with a dynamic and collaborative environment, often referred to as a social coding platform, that supports peer review, commenting, and discussion.
2. is a very versatile tool to use. It is ideal for working on projects of any size and it is an amazing tool for web workflows.
3. Primarily you can use it for publishing your work, as a versioning control system and as a collaborative tool.
4. It is ideal for creating a backup of your work instead on saving the code on a desktop or on a corporate server.
5. GitHub helps in getting a job –So many entry-level and junior developers feel the answer to "Do I need a GitHub?" is clearly yes.

# Do prospective employers actually look at our GitHub?

Junior and intermediate coding professionals today are likely to be asked during a job search, "Can we see your GitHub?" by recruiters and hiring managers. As technology trends changed, so too did some of the concepts around hiring. The rise of the open-source movement, the drastic reduction in barriers (both cost and complexity) to hosting, general public access to publishing on mobile app markets, and the popularity of repository sites all contributed to a programmer's new ability to show a *portfolio of code* to employers.

Tech pros are routinely told that being active on GitHub gives them an advantage because it provides recruiters and hiring managers with a way to size up their skills.

For job seekers, this means that GitHub can be a powerful tool for attracting the attention of employers whose interests align closely with your own.

### HOW TO TEACH GIT?

Version control is a necessary piece of the open source community and Git has an unfortunately steep learning curve. It is easy to forget that the tools we use everyday are not actually intuitive for beginners. The purpose of this report is to help educators anticipate where their students’ confusion and questions might be coming from. Now we wil go step by step.

### What is Git?

Start by giving your students an understanding of the environment Git exists in and the problem that it solves. A lot of the confusion that occurs later on (related to remotes, branching, commit history structure) can be avoided if your beginner understands why Git exists.

When explaining Git, do not say “it’s version control”. This doesn’t help especially to beginner programmers. It only helps if they already know some other version control system.

You can say that Git is a tool that protects yourself and others from yourself and others. Git exists so you can modify/change/break/improve your code, secure in the knowledge that you can not ruin your work too badly because you created save points.

While teaching Git, it is easy to start talking about the things you think are interesting, but just focus on the part the student needs to know. They will be learning a lot of new information in this process and adding on unnecessary levels of detail will compete for attention with necessary information.

*“You’re teaching your student to drive, not how the engine works.”*

You must explain the terms before using them and then use them so that your student can Google when they get confused later. This seems basic to say, but we all fail at this regularly, so explain what the terminology means before using it. When teaching a new tool, try not to use incantations, and instead use analogous phrases or metaphors.

### The Local Workflow

### Init:

When teaching git init we have to say the word ‘initialize’, because ‘init’ sounds like ‘in it’, which can be confusing.

Make sure you explain that this will have git keep track of changes in your current directory and if they leave this directory, there is not a way for git to keep track of changes elsewhere. Showing them that git init created the subdirectory .git with ls –la is a good practice.You can even show them what happens if you delete .git. Also, this is a good time to use and explain the words ‘repository’ and ‘repo.’

#### Staging Area:

The staging area is a confusing concept. By using terms like ‘staging area,’ you’re invoking a sense of moving things spatially, so be careful and make sure your student understands that files are not being moved from one directory to another. Moving files from the ‘working directory’ to the ‘staging area’ and then the ‘repository’ does not move them but actually changes how Git is keeping track of them.

#### Status:

Encourage your students to use this command constantly. Since it won’t change the state of things, you can use it at any time whether state has changed or not to help them see. Teach them to use this a lot because it will be useful in the future when they’re confused. This command should be at the top of their list of “how do I figure out what happened” tools.

#### Add:

Sometimes students think add means “start tracking this file” rather than “include the changes from this file.” To avoid this confusion, explain the steps, and walk them through multiple commits so you can add a file which had been included in the previous commit.

**Commit:**

More than many other commands, forcing yourself to explain the concept of a commit as much as possible before using the word commit is useful. Explain that commit is a save point or a check point.

You can also talk about a commit as a collection of changes that need to be made together. Especially with beginner programmers, explain this concept with non-code examples, because it might not be clear that saving files separately could leave the code base in a broken state.

Always make sure you teach git commit -m "message" because if they forget to use the -m flag then git will open vi and ask them to enter the message and save the file. Using vi without prior experience is a confusing and potentially scary process so teach commit without -m does nothing.

A confusion is some beginners think commit means push. To avoid this problem, walk through the process of adding and committing a few times before setting up and pushing to the remote repository. This will separate git commit from git push and help show that the commands to set up the remote repo are not a part of the workflow.

**Log:**

Log is very important tool that doesn’t get enough attention early on. Many tutorials do not show how to recover old states, despite the fact that the whole point of Git is to be able to recover old versions of your code. You should teach git log on day one. Show them how to recover a previous state using the name of the commit.

On the first day of teaching Git, just tell beginners history is immutable or permanent. Each Git as if the log will never be altered and they can always get back to anything.

**Diff:**

Teaching git diff as part of the work flow right before or after git status, so that they see which files have been changed and what changed in them.

### Remote Repositories

Before teaching any command related to pushing and pulling, take some time to explain why one would use remote repos: for backups, collaboration, etc. Once again, everything will go more smoothly if they understand why they are doing these incantations, which is especially important now. since you are about to teach git remote add origin and git push -u origin master teach that these are just names assigned to the place they will push and pull from (this is easier to explain if you’ve already taught branches). The names are used so you do not have to type out the whole address every time you push and pull.

Teach these terms like environment variables. Make sure you distinguish between setting these up and pushing. The commands are often taught together but make sure they understand that in the future they don’t need to use the whole command git push -u origin master.

**Pull and Fetch:**

Git pull and git fetch are hard to explain together, especially if you haven’t covered branching yet. It is easier to just teach one of them, because they probably will not need to know both for a while. Teach the other one later. It does not matter which one you teach first, pick your favorite.

Be careful when teaching with GitHub though, because GitHub wants all use of Git to be in terms of GitHub, so it is easy to slip into a GitHub-centric workflow.  
Make sure to teach that Fork and Pull Requests are not built in to Git, they are GitHub tools.

#### Clone:

When teaching git clone for the first time, use HTTP, rather than SSH. Teaching SSH key-gen is a confusing, time consuming tangent. Understanding SSH is taking focus away from Git. SSH is about optimization, and on day one, you need to focus on keeping everything conceptually simple.

Once your students know Fork and git clone, this is an ideal time to show them that there are three ways to start a project: Fork + git clone, git clone, and starting locally with git init.

### Branching

When you start talking about branching, you can draw a graph and explain that concept. As you walk them through creating branches and merging, refer back to your diagram and show how it affects the picture.

Teach that Git automatically calls the main branch “master”. That is just a name for the “copy of the version of my code that my boss cares about”.

Show them how to make branches, checkout branches, and checkout a branch name that doesn’t exist yet. You can teach your students how to delete branches later.

#### Merge:

When you teach branches, teach them how to merge and choose an example case that will create a merge conflict.

Learning how to merge branches without resolving merge conflicts is not learning to merge. You need to teach resolving merge conflicts right away, because they will definitely encounter merge conflicts.

You need to talk about merge conflicts without using the words merge conflict, even though it sounds self explanatory.

Open up the files in question, and show them how to find the patches of code that generated the error. Git marks these problems the same every single time. You might feel compelled to explain HEAD at this time, but suppress that urge, it probably will not help clarify the situation.

Explain that Git will let a user commit if they delete the lines of arrows and equals signs, regardless of what they keep between them. Git does not care if you take pieces from one history and pieces from the other. Git only cares if a human looked at it and said “yeah this looks good”.

Once you have fixed the conflict in the two histories, commit the changes and show your student the logs again so they can see Git treats merging just like regular commits.

Also, now that your student has seen branching and merging, show the log history of both branches so they know log only shows the history of one branch.

If you have already talked about remote repositories, you can teach merge conflicts as a part of pull requests to prepare them to contribute to other projects.

## Git-GitHub Cheat Sheet

## Install

### GitHub Desktop

[desktop.github.com](https://desktop.github.com/)

### Git for All Platforms

[git-scm.com](https://git-scm.com/)

## Configure tooling

Configure user information for all local repositories

**$ git config --global user.name "[name]"**

Sets the name you want attached to your commit transactions

**$ git config --global user.email "[email address]"**

Sets the email you want attached to your commit transactions

**$ git config --global color.ui auto**

Enables helpful colorization of command line output

## Branches

Branches are an important part of working with Git. Any commits you make will be made on the branch you’re currently “checked out” to. Use git status to see which branch that is.

**$ git branch [branch-name]**

Creates a new branch

**$ git checkout [branch-name]**

Switches to the specified branch and updates the working directory

**$ git merge [branch]**

Combines the specified branch’s history into the current branch. This is usually done in pull requests, but is an important Git operation.

**$ git branch -d [branch-name]**

Deletes the specified branch

**Create repositories**

When starting out with a new repository, you only need to do it once; either locally, then push to GitHub, or by cloning an existing repository.

**$ git init**

After using the git init command, link the local repository to an empty GitHub repository using the following command:

**$ git remote add origin [url]**

Turn an existing directory into a Git repository

**$ git clone [url]**

Clone (download) a repository that already exists on GitHub, including all of the files, branches, and commits

**The .gitignore file**

Sometimes it may be a good idea to exclude files from being tracked with Git. This is typically done in a special file named .gitignore. You can find helpful templates for .gitignore files at [github.com/github/gitignore](https://github.com/github/gitignore).

**Synchronize changes**

Synchronize your local repository with the remote repository on GitHub.com

**$ git fetch**

Downloads all history from the remote tracking branches

**$ git merge**

Combines remote tracking branches into current local branch

**$ git push**

Uploads all local branch commits to GitHub

**$ git pull**

Updates your current local working branch with all new commits from the corresponding remote branch on GitHub. git pull is a combination of git fetch and git merge

**Make changes**

Browse and inspect the evolution of project files

**$ git log**

Lists version history for the current branch

**$ git log --follow [file]**

Lists version history for a file, including renames

**$ git diff [first-branch]...[second-branch]**

Shows content differences between two branches

**$ git show [commit]**

Outputs metadata and content changes of the specified commit

**$ git add [file]**

Snapshots the file in preparation for versioning

**$ git commit -m"[descriptive message]"**

Records file snapshots permanently in version history

**Redo commits**

Erase mistakes and craft replacement history

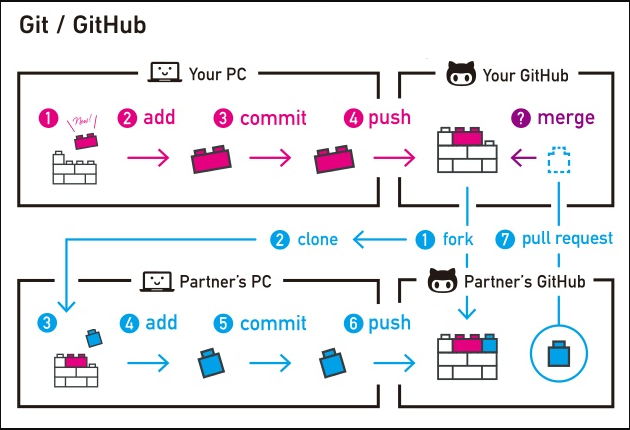
**$ git reset [commit]**

Undoes all commits after [commit], preserving changes locally

**$ git reset --hard [commit]**

Discards all history and changes back to the specified commit

**Git-GitHub Workflow**



**Some Examples**

**Local Workflow:**

**git init:** Put Git in this folder so that it keeps track of changes to files in this folder and subfolders

**Working Directory:** The directory you’re writing code in

**Staging Area:** Files are in the staging area if the changes in them will be included in the next save point

**Repository:** Everything Git is keeping track of

**git status:** Shows which files have been changed and which ones are ready to be committed

**git add filename.txt:** Include the changes to this file in the next commit

**git rm filename.txt**

**git mv filename.txt otherfile.txt**

**git commit -m “commit message”:** Wrap up all these changes and save them together with a short description of the changes

**git log:** Show a history of all commits

**git diff:** Show what is different from the last commit line by line

#### Remote Repository

**git remote add origin address-of-remote:** Make address-of-remote a new place to put my code and call it “origin”

**git push -u origin master:** Push my code to the location origin points to, on the master branch, and also in the future I will pull code from this same location

**Upstream:** Where I will pull code from in the future

**Origin:** Where I put backups or share my code

**git pull:** Grab code from another repository

**git fetch:** Grab code from another repository

**git push:** Save my history and changes in another location

**Fork:** I want a GitHub repo that looks like someone else’s repo

**Pull Request:** I made some changes that I would like you to include in your repository, please accept them

**git clone:** Give me the code at this location

**Branching**

**git branch:** What are all my branches or what are all the names of the different versions of my code?

**git branch feature:** Make a new branch/version of my code with the name feature

**git checkout feature:** Move to that branch/version of my code so I can make changes to that branch/version of my code

**master:** The name of the branch which should be the official, working, well documented, version of my code

**git merge:** Combine the history of two branches so I can have the changes from both in one place

**Merge Conflict:** Git does not know how to combine two histories and needs human assistance

Additional Resources

1. <https://git-scm.com/book/en/v2/>
2. <https://www.youtube.com/user/GitHubGuides>
3. <http://git-school.github.io/visualizing-git/>
4. <https://www.atlassian.com/git/tutorials>

**Glossary**

* **git**: an open source, distributed version-control system
* **GitHub**: a platform for hosting and collaborating on Git repositories
* **commit**: a Git object, a snapshot of your entire repository compressed into a SHA
* **branch**: a lightweight movable pointer to a commit
* **clone**: a local version of a repository, including all commits and branches
* **remote**: a common repository on GitHub that all team members use to exchange their changes
* **fork**: a copy of a repository on GitHub owned by a different user
* **pull request**: a place to compare and discuss the differences introduced on a branch with reviews, comments, integrated tests, and more
* **HEAD**: representing your current working directory, the HEAD pointer can be moved to different branches, tags, or commits when using git checkout