Getting to know Github
Contents

1 Introduction: Why use Github 1

2 Step 1: Installing Git 2
   2.1 Windows 2
   2.2 Mac OS X 2
      2.2.1 Git For Mac Installer 3
      2.2.2 Homebrew 3
      2.2.3 MacPorts 4
   2.3 Linux 5
      2.3.1 Debian/Ubuntu (apt-get) 5
      2.3.2 Fedora (dnf/yum) 5
      2.3.3 Arch (pacman) 6
   2.4 Configuration (All Systems) 7
      2.4.1 Windows 7
      2.4.2 Mac OS X 7

3 Step 2: Creating a Repository 9
   3.1 Create a local Git repository 9
   3.2 Initialize the local Git repository 10
   3.3 Create a repository on Github 10
   3.4 Connect the local repository to the Github repository 11

4 Step 3: Add files to the repository 12
   4.1 git status 13
   4.2 git add 13
4.3 git commit .................................................. 14
4.4 git push .................................................... 15

5 Step 4: Collaborate ........................................... 16

5.1 Branching .................................................. 17
  5.1.1 Creating and viewing branches ................. 17
  5.1.2 Using a branch ........................................ 18
  5.1.3 Saving a branch ...................................... 19
  5.1.4 Merging a branch ..................................... 19

5.2 Cloning .................................................... 21

5.3 Forking ..................................................... 23
  5.3.1 Forking the original project ..................... 23
  5.3.2 Cloning our version of the project ............. 24
  5.3.3 Staying connected to the original project ... 24
  5.3.4 Getting updates from the original project ... 25
  5.3.5 Opening a Pull Request ......................... 26
  5.3.6 Handling Pull Requests ......................... 28

A Quick Reference ............................................ 30

A.1 Configuration ........................................... 30
A.2 Create Repository ...................................... 30
A.3 Making Changes ........................................ 30
A.4 Branches .................................................. 31
A.5 Cloning ..................................................... 32
1 Introduction: Why use Github

Github acts as a user interface for git, a version control system. Version control systems allow us to keep track of any changes we make in our work, and revert back to an earlier version of our work, all without needing to save multiple files.

Git takes what it calls snapshots, which is essentially a saved version of our work at a specific point in time. If we then make changes we don’t like in your work, we can revert back to an earlier snapshot. This is helpful as Git handles all the snapshots, and we no longer have to have multiple files saved for every different version of our work.

Git also allows us to collaborate. A project can be stored in one place, and anyone that contributes or adds to the work can interact with the project from anywhere. A person can clone your code and have their own version. They can then make changes to that clone and ‘push’ their updated version of the code to the main project. These changes can then be evaluated by the project owner, and ‘merged’ into the main project, meaning to add the changes to the main project.

This method of storing a project in one place and editing it from anywhere using clone, push, and merge controls is incredibly beneficial. Everyone involved in the project has access to the most up to date version of the code, and they can access it from anywhere. It also allows for open-source collaboration all around the world.

Github is not equivalent to Git, but it offers us a friendlier interface than just the command line. It exists both as a website, and as a desktop application. From either, we can view our own projects, clone others’ projects, browse the enormous collection of projects by other users, as well as edit files, clone projects, and manage pull and merge requests.
2 Step 1: Installing Git

First things first, we need to install Git before we can use it! How Git is installed will vary depending on the system, so please find your appropriate operating system below and follow the instructions.

2.1 Windows

This is a very straight-forward approach to installing Git on Windows. It acts like a standard installer with a user interface, with prompts at each stage asking the user to input a path and continue until finished.

The installer can be found [here](#).

This installer package will also install Git BASH, which is a Bash emulator. Bash is the command line interface found on *NIX systems. You can access Git through both the standard Command Prompt, or through the newly installed Git Bash.

Once the installation is complete, make sure to open Command Prompt (or Git Bash) and type:

```
$ git --version
```

to ensure installation was successful. If it was successful, we will get a short message with the Git version number. If it was unsuccessful, we would get a longer message stating the command was not recognized.

2.2 Mac OS X

There are multiple ways to install Git on Mac OS X. There is also the chance you already have it installed, if you’ve previously installed XCode or its Command Line Tools.

To test if we already have Git installed, open a Terminal window and enter:
$ git --version

This should produce a Git version number if it is already installed. If it is not already installed, then we will receive an error message stating git is not a recognized command. If you receive this error, please see below for ways to install Git on Mac OS X.

2.2.1 Git For Mac Installer

This may be the most straight-forward approach to installing Git on Mac OS X. It acts like a standard installer with a user interface, with prompts at each stage asking the user to input a path and continue until finished.

The installer can be found [here](#).

Once the installation is complete, make sure to open Terminal and type:

$ git --version

...to ensure installation was successful. If installation was successful, we would receive a short message stating the Git version. If it was unsuccessful, then we would receive a longer message stating the command was not recognized.

2.2.2 Homebrew

If you have previously installed [homebrew](#) then you can use it to install Git.

Simply open the Terminal and type:

$ brew install git

Once the installation is complete, make sure to stay in Terminal and type:
to ensure installation was successful. If installation was successful, we
would receive a short message stating the Git version. If it was unsuccess-
ful, then we would receive a longer message stating the command was not
recognized.

2.2.3 MacPorts

If you have previously installed [MacPorts] then you can use it to install
Git.

First, open Terminal and ensure MacPorts is up to date by running:

```
$ sudo port selfupdate
```

Then we need to search for the latest Git ports. While still in Terminal,
enter:

```
$ port search git
$ port variants git
```

Note, if you receive an error in the above code for the word git, then
replace it with git-core in the above snippet and run again.

Finally, we can install Git. We will also add in Bash completion and the
OS X keychain helper. While still in Terminal, enter:

```
$ sudo port install git +bash_completion+credential_osxkeychain
```

Note, if you used git-core when searching, then be sure to replace git here
with git-core.

Once the installation is complete, make sure to stay in Terminal and
type:

```
$ git --version
```
to ensure installation was successful. If installation was successful, we would receive a short message stating the Git version. If it was unsuccessful, then we would receive a longer message stating the command was not recognized.

2.3 Linux

There are multiple ways to install Git on Linux, depending on the Linux flavor being used. Please find your appropriate flavor below and follow the instructions.

2.3.1 Debian/Ubuntu (apt-get)

For Debian-based flavors, we can use apt to install Git.

First, we want to make sure apt is up to date. From your preferred shell, enter:

```
$ sudo apt-get update
```

Then, we can retrieve and install the Git package. While still in the shell, enter:

```
$ sudo apt-get install git
```

Once the installation is complete, make sure to stay in the shell and type:

```
$ git --version
```

to ensure installation was successful. If installation was successful, we would receive a short message stating the Git version. If it was unsuccessful, then we would receive a longer message stating the command was not recognized.

2.3.2 Fedora (dnf/yum)

Git is also available for installation through both dnf and yum.

To install using dnf, open your preferred shell and enter:
$ sudo dnf install git

To install using yum, open your preferred shell and enter:

$ sudo yum install git

Once the installation is complete, make sure to stay in the shell and type:

$ git --version

to ensure installation was successful. If installation was successful, we would receive a short message stating the Git version. If it was unsuccessful, then we would receive a longer message stating the command was not recognized.

2.3.3 Arch (pacman)

Arch offers a package to install Git using pacman.
To install, open your preferred shell and enter:

pacman -Syu git

Once the installation is complete, make sure to stay in the shell and type:

$ git --version

to ensure installation was successful. If installation was successful, we would receive a short message stating the Git version. If it was unsuccessful, then we would receive a longer message stating the command was not recognized.
2.4 Configuration (All Systems)

Once we have installed Git, we can go ahead and create our account on Github and connect it to Git on our local system.

Head over to Github and create an account by following the prompts and verifying your e-mail address.

Once we’ve created our account, we need to connect it to our local Git installation. No matter your Operating System, open up your preferred shell (Command Prompt, Bash, Terminal, and any Bash emulators will work) and enter:

```
$ git config --global user.name "Jane Doe"
$ git config --global user.email "Jane.Doe@example.com"
```

Be sure to enter your Github username and e-mail address in place of Jane Doe in the above snippet.

If we want Git to remember our login information, we can use one of the many credential managers available. Find your appropriate Operating System and follow the instructions to install a credential manager.

2.4.1 Windows

Grab the latest .exe installer from Git-Credential-Manager-for-Windows. You can find the download link by scrolling down the page until you find the 'Download and Install' section. To install, simply follow the prompts in the installer application.

2.4.2 Mac OS X

If you installed Git using Homebrew or MacPorts, the helper application should already be installed.

To check if it is installed, open Terminal and run:
$ git credential-osxkeychain

If the output is something along the lines of the following:

$ usage: git credential-osxkeychain <get | store | erase>

then we already have the helper installed. If it is installed, then we can configure it to store our information by staying in Terminal and typing:

$ git config --global credential.helper osxkeychain

and the next time we are prompted for our user name and password, the keychain will store it for future use.

If the keychain helper was not installed, or you installed using something other than MacPorts or Homebrew, then we can install it manually using curl.

To download the keychain, open Terminal and enter:

$ curl -O http://github-media-downloads.s3.amazonaws.com/osx/git-credential-osxkeychain
$ sudo mv git-credential-osxkeychain /usr/local/bin

Now that we have downloaded the keychain, we need to make it an executable. While still in Terminal, enter:

$ chmod u+x /usr/local/bin/git-credential-osxkeychain

Finally, we can configure the keychain to work next time we login. While still in Terminal, enter:

$ git config --global credential.helper osxkeychain

and the next time we are prompted for our user name and password, the keychain will store it for future use.
3  Step 2: Creating a Repository

A repository, often shortened to repo, is the virtual storage of our project. It can include any type of file, be as small as a few files, or as big as multiple files and multiple nested folders.

The repository is where all the information for our project will be stored, along with all the snapshots Git saves. It is through this repo that we can access our project from anywhere with internet access, as well as share our code with others and allow them to make changes.

Github offers unlimited public repositories. If you’re interested in private repositories, be sure to check out Github’s paid versions or apply for their student pack.

3.1 Create a local Git repository

If we want to start completely from scratch for a project and set it up to use Git, then the first step is to create a folder for our project.

We can do this in the standard form of opening our File Explorer, navigating to where we want to create our folder, and simply adding a new folder. Using Git is going to involve the Command Line, however, so I would recommend getting comfortable navigating and creating folders through the Command Prompt.

To create a new folder through the Command Line, open your preferred Command Prompt. Assuming your shell starts at the current user’s root directory, from here we can move to the main folders in our system, such as Documents, Downloads, Desktop, etc. Choose where you would like to create the folder for your project, change directories until you are in the desired location, and use mkdir to create the folder. Finally, change directories into the newly created folder.

For example, lets say we want to create our project folder, called myProject, in our Documents folder. We would open our Command Prompt and enter:
$ cd Documents
$ mkdir myProject
$ cd myProject/

3.2 Initialize the local Git repository

Now that we have created the folder for our project, we can initialize it. To do so, open your preferred command line interface and change directories until you are in the project folder. Then, type:

$ git init

This command only needs to be run once per project folder, and if our project contains multiple subfolders, we only need to add it to the root folder.

If you already have an existing project folder with files in it, you can change to the project directory and run git init at any point.

3.3 Create a repository on Github

Now that we have our local repository set up, we can create a repository on Github to link it to.

Navigate to Github and log in to your account. Once logged in, you’ll see a small plus symbol next to your username and profile picture in the upper right-hand corner. Click the plus symbol and select ’New repository’ from the drop down menu.

The next page will ask you to name your repository and give it a short description. You will be able to pick if you’d like the repository to be public, or private. You have an unlimited number of public repositories, but private repositories are through a paid service from Github. You can also choose to check the box to ’Initialize this repository with a README’ if you would like Github to automatically generate a blank Read Me file for your repository. You can choose to add one later if you would prefer. Finally, click Create repository.
You will be forwarded to your new (and empty) Github repository. This will look as follows:

![Quick setup — if you’ve done this kind of thing before](image)

Take note if the web address given at the very top of the page, under 'Quick setup – if you’ve done this kind of thing before'. This is the link you’ll need to connect your local repository to the Github repository we just created.

### 3.4 Connect the local repository to the Github repository

Finally, to tie it all together, we need to connect our local repository to the Github repository we just created.

The next step will be the same whether you started with an empty
project folder, or an existing project folder.

Open your preferred command line interface and change directories until you are in the project folder and type the following command:

```bash
$ git remote add origin https://github.com/MyUserName/myProject.git
```

Make sure you replace MyUserName with your own Github user name, and replace myProject with the name of your Github repository. You can copy this link from the empty Github repository when you create it.

To ensure we set our origin correctly, stay in the command prompt and type:

```bash
$ git remote -v
```

If everything was correct, you should see two lines of output that look like the following:

```
origin https://github.com/MyUserName/MyProject (fetch)
origin https://github.com/MyUserName/MyProject (push)
```

4 Step 3: Add files to the repository

Now we can begin to add files to our project folder for Git to keep snapshots of, as well as share on our Github page.

If you already have files, you can drag them into the project folder we created in the above steps. If you already have files in the project folder, that’s fine too. You can start clean and simply save files to the project folder from now on, as well.

For the sake of this example, let’s assume the project folder is completely empty. We need something to add to the repository, so let’s create a quick test file. Open your preferred command prompt and change directories into the project folder and type the following command:
$ echo "This is a test text file." >> testFile.txt

This will create a text file, called testFile.txt to our project folder. Git is aware of the file, but it isn’t aware that we want to keep track of it and add it to our Github repository.

4.1  git status

To check that Git is aware of our file, stay in the command prompt and enter:

```
$ git status
```

and notice the output:

```
On branch master
Your branch is up to date with 'origin/master'.

Untracked files:
 (use "git add <file>..." to include in what will be committed)

    testFile.txt

nothing added to commit but untracked files present (use "git add" to track)
```

Here we can see that Git notices our file, but it states that it is untracked. This means Git won’t be keeping track of the file, and thus won’t be taking snapshots of it either.

4.2  git add

To tell Git that we want to track it, we can enter the following command:

```
$ git add testFile.txt
```
If you have multiple files and want to add them all at once, you can replace the testFile.txt with a period (.) to add everything in the project folder that is currently untracked.

Now we can use git status again to make sure our new file has been added, and take note of the new output:

```
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    new file:   testFile.txt
```

This output lets us know Git has added our file to be tracked. If we accidentally added a file we don’t want tracked, we can use the command given above (git reset HEAD testFile.txt) to remove it from Git.

### 4.3 git commit

Finally, to tell Git we’re done with the file and want to save the changes made, we must commit the changes. The commit command tells Git that we want to save our changes, and keep a snapshot of what changes were made. To commit our new file, we would enter the following command:

```
$ git commit -m "Added testFile.txt from local repository."
```

The `-m` is a flag that tells Git we want to use the following section in quotes as our commit message. Commit messages explain what we’ve changed since our last commit. It is important to be descriptive, but also concise. These commit messages let us know what we’ve changed in our code, but it also lets anyone else looking at our code to know what has changed.
If we used `git status` to check the status of our local repository at this point, the output would tell us:

```
Your branch is ahead of 'origin/master' by 1 commit.
(use "git push" to publish your local commits)
nothing to commit, working tree clean
```

### 4.4 `git push`

Now that we’ve told Git we want to track our file, added it to be tracked, and saved the changes with a commit, we can finally add it to Github. To do so, stay in the command prompt and type the following command:

```
$ git push -u origin master
```

The `-u` flag here lets us later use argument-less `git push` and pulls, meaning next time we want to push our new files to Github we can just type:

```
$ git push
```

If you initialized your Github repository with a README file, you may run into an error message like below when you try to push your commit:

```
$ git push -u origin master
! [rejected] master -> master (fetch first)
error: failed to push some refs to '.../remote/

hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

This gives us a lot of information, but most importantly it gives us the line ”Updates were rejected because the remote contains work that you do not have locally”. This is saying that there are files on our Github repository, but not on our local repository. To fix this, stay in the command line and enter:
$ git pull origin master

This will grab the files in our Github repository and put them in our local repository, too. Now we can push again to save our changes:

$ git push -u origin master

If you log in to your Github page and click on your myProject repository, you’ll see our file testFile.txt is now inside our Github repository. You can click it to view its contents.

It is important to keep our local repository in sync with our Github repository. Anytime you add files, or modify any existing files in the local repository, make sure you run through the steps below:

$ git add .
$ git commit -m "My commit message."
$ git push

If you are unsure what is untracked, or what needs to be committed still, you can run the git status command at any time.

If you modify, import, or create any files through your Github repository, make sure you pull those changes into your local repository afterward by changing directories into the project folder and running the command:

$ git pull

5 Step 4: Collaborate

Git, and Github, allow us to not only save our code, but share it with others as well. The reverse is also true, we can participate with other people’s projects, as well. There are multiple ways to interact with other people’s projects, and which approach depends on what it is we want to do.
If you want to experiment with a piece of code but not fully integrate it into your project just yet, take a look at the section on Branching.

If you’ve found code on Github you want to have locally, but don’t want to change or contribute to it, then take a look at the section on Cloning.

If you’ve found a project on Github you would like to contribute to, take a look at the section on Forking.

5.1 Branching

Branching is a way for us to create a temporary version of our code that is kept entirely separate from our main project. We can then write new code, test a feature, or make changes to the branch and see how it effects our project. These changes will only effect the branch we’re working on, and our original project will remain in the same state as when we first branched it.

If we decide we want to integrate the changes we made to our branch into our main project, we can merge the two. If we decide we don’t like the changes, we can simply delete the branch and move back to our main project.

5.1.1 Creating and viewing branches

To create a branch of our main project, open your preferred command prompt and change directories into the project folder. To create a branch, type the following command:

```
$ git branch testBranch
```

In this example, testBranch is the name of our branch we just created.

If you’d like to see all branches of your master project, you can use the branch command by itself:

```
$ git branch
* master
  testBranch
```
You’ll see in the output our master project, as well as any branches we’ve created. The asterisk next to a branch name means it is the current project we’re working on.

You can also at any point run the command `git status` to view which branch you’re currently working on.

You can create as many branches as you want, and branches can be branched from existing branches.

### 5.1.2 Using a branch

Now that we’ve created a branch, we need to tell Git that we want to use it instead of our main project.

To switch to a branch, we use the `checkout` command. In command prompt enter:

```
$ git checkout testBranch
```

Now any changes we make and commit will be on our testBranch, leaving our master project untouched.

We can also use a shortcut to create a new branch and switch to it in one command:

```
$ git checkout -b testBranch
```

The -b flag tells `git` to create the branch `testBranch` and switch to work on it all in one command.

If at any point we want to switch back to our master project, we can simply use the `checkout` command again:

```
$ git checkout master
```
5.1.3 Saving a branch

We can add and commit branch changes like normal once we’ve checked out the branch. Make sure if you’ve added any changes, with git add, that you also commit the changes before trying to switch to another branch or master. Git requires staged items be committed before changing between working branches.

While adding and committing is still the same with branches, our push command will look a little different.

Once we’re done making changes for the time being, we want to save those changes to our branch (*Note - not the master project). Remember we save our changes with the git push command. When pushing a branch, rather than our master project, we need to specify that we’re using a branch. To do so, our push command would look like:

```
$ git push origin testBranch
```

The origin keyword is in reference to our Github project link we setup when we created the repository. The testBranch keyword is just the name of the branch we’re working on.

5.1.4 Merging a branch

If we decide we like the changes we made on our branch and we want to incorporate them into our master project, then we can merge the two together. To do so, checkout the master project and use the merge command with the branch we want to incorporate:

```
$ git checkout master
$ git merge testBranch
```

There are two common forms of merging Git handles.

If the output from your git merge command includes the term "Fast-forward", then our branch was directly upstream from our master project and all Git has to do is move up a step. It accomplishes this by simply using
your branch as the new master project.

If the output from your git merge command includes the term "recursive", then our branch has diverged from an older point. This can happen if we’ve branched from a branch, or if we’ve branched from an older version of our master project. In this case, Git has to do a little more work. It performs a three-way merge between the branch, the master, and a common ancestor (an older version they both have in common).

Either of the above outputs is fine, and means the merge went successful.

If the output instead is an error, such as:

```
$ git checkout master
$ git merge testBranch
Auto-merging testFile.txt
CONFLICT (content): Merge conflict in testFile.txt
Automatic merge failed; fix conflicts and then commit the result
```

then our merge was unsuccessful. This can happen for multiple reasons. It is common in recursive merges, though, when we’ve made changes to the same part of the code in both sections we’re trying to merge. Git will essentially pause the merge at this point until you have worked out the conflict.

To view the file causing conflict, we can use the git status command:

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified: testFile.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

This message tells us testFile.txt has been modified in our master project as well as the branch we’re trying to merge with it.
To view the exact part of the file causing the conflict, open the file with your preferred editor. You’ll see Git has added new markings in our file to show us exactly where, and what, the conflict is. An example could be:

```
This is our
<<<<<<< HEAD
test file
=======
file for testing purposes
>>>>>>> branch-testBranch
```

The `<<<<<<< HEAD` marker tells us this is how the conflicted line appears in our master project. The `>>>>>>> branch-testBranch` line tells us this is how the conflicted line appears in the branch we’re trying to merge. The `=======` is simply a separator between the two.

To fix this merge conflict, we can decide which line we want to keep and remove the other line, as well as remove the Git conflict markers. We can then save our file, run `git status` again to ensure there aren’t any other conflicts we missed, then add the file and commit it as normal.

Now that we’ve merged our branch, we can delete it because we no longer need it. To delete a branch, we use the `branch` command with the `-d` flag:

```
$ git branch -d testBranch
```

You can keep branches around for as long or as short of an amount of time as you’d like.

### 5.2 Cloning

Cloning has multiple purposes that will be further discussed in the Forking section.

For this section, though, we will be discussing cloning as a means of obtaining existing code from Github so that we can use it on our computer.

This means we want to copy the code so we can run it, but that we don’t want to make changes and/or contribute to the existing code.
First, find the Github page for the project you want to copy. Once on the main page for the project, you’ll see a green button that says "Clone or download" along the top right of the page, directly on top of the list of the files in the repository. Click the button, and you’ll see a small drop down like so:

![Clone button screenshot](image)

You’ll see it has given us a direct link to the project’s .git. This is what we’ll use to clone the repository, so go ahead and copy this link address.

Now, we’ll open our preferred command prompt and change directories to where we want to save the repository to. Let’s say we want to clone the repo into our Documents folder. First we’ll cd into Documents, then we’ll use the clone command and paste in the clone link we just copied, like so:

```
$ cd Documents
$ git clone https://github.com/github/git-sizer.git
```

This will create a copy of the repository contents into a folder called git-sizer, inside of our Documents folder.

Git will automatically name the folder we clone the repository to after the original repository name. If you’d like to re-name the folder to something else, you can add a desired folder name after the clone command and repository, like so:
$ git clone https://github.com/github/git-sizer.git myFolder

Once you have run the clone command, you should see all of the repository files are now copied into the folder created on your computer.

If you don’t intend to change any code or interact with the original repository, it’s a good idea to remove the connection. To do so, open up your preferred command prompt and change directories until you are in the cloned folder. Then, to remove the Git connection, enter the following command:

$ rm -rf oldRepoName.git

Make sure you’re replacing oldRepoName with the actual name of the old repository.

Cloning can also be used to retrieve your own code that was created remotely, meaning it exists on Github but not on your computer. The above steps would be the same, but you would probably want to leave your connection to Git intact.

5.3 Forking

Forking allows us to work on someone else’s project. We can fork a repository if we’d like to contribute to the project, if we want to fix a bug in the project, or if we’d like to use the project as a basis for our own personal project.

5.3.1 Forking the original project

First, we need to find a project we want to contribute to. Once we’ve done so, look at the top right-hand corner of the main project page and you’ll notice three buttons, like shown below:
The first button, labeled 'Watch', allows us to follow the project and be notified if any changes are made. It is a good idea to watch any repositories you fork, so that you can know when it has been updated.

The second button, labeled 'Star', is a way for us to essentially bookmark a repository. We can save it for later, and show our appreciation for the project, but unlike the 'Watch' button, we won’t be notified of any updates on starred projects.

The last button is the one we’re most interested in for this section, the one labeled 'Fork'. When we click this button Github will make a copy of the repository remotely on our Github page. It will be found under our username, but it will have the same name as the original repository.

Go ahead and click fork, then navigate to your Github repository page. You’ll see your own copy of the repository we just forked. This repository was created remotely, meaning right now it only exists on Github, not on our computer.

5.3.2 Cloning our version of the project

To obtain a copy of the code on our computer so we can work on it, we must first clone it. Navigate to the newly created repository on your Github page (not the original repository page) and click the 'Clone or download' button. Copy the link it gives you, then open your preferred command prompt and navigate to where you want to download the project to. Enter the following command to clone the project:

```
$ git clone https://github.com/Your_Username/Forked_Repo_Name
```

Once complete, you’ll see all the project files are in the newly created folder.

5.3.3 Staying connected to the original project

When we fork a project we grab a copy of the project exactly as it exists when we press the 'Fork' button. This means any changes made to the original repository we forked won’t be saved to our fork, as well. To make sure
we can grab any updates to the original repository and add them to our copy, we need to set it as the upstream repository. This means we can grab any changes made to the original repository, add them to our repository, and still push our changes to our own repository.

To set the original repository as upstream, first we need to navigate to the original project’s repository page on Github. Click the 'Clone or download' button and copy the link it gives you. Now, open your preferred command prompt and change directories until you are in the folder containing our version of the project code. Enter the command:

```
$ git remote add upstream https://github.com/OriginalUser/OriginalName.git
```

To make sure we’ve set up everything correctly, use the git remote command:

```
$ git remote -v
origin https://github.com/Your_Username/Forked_Repo_Name.git (fetch)
origin https://github.com/Your_Username/Forked_Repo_Name (push)
upstream https://github.com/OriginalUser/OriginalName.git
upstream https://github.com/OriginalUser/OriginalName.git
```

We’ll see our repository as the first two origin listings, and the original repository as the bottom two upstream options.

5.3.4 Getting updates from the original project

Now, let’s say we’ve been working on our version of the project and we notice the original owner of the project has implemented some updates. If we want to grab those updates and add them to our code, without losing the work we’ve done, we can use the fetch and merge commands.

First, we need to grab the updates from the original repository. To do so, open your preferred command prompt and change directories until we are in the project folder we’ve been working on. Then we’ll use the fetch command with the upstream repository we set up in the previous step. Enter the following command:
Any changes that were made the original project will exist in a branch called upstream/master. This is the branch we want to implement with our code, but still keep the changes we've made as well. To do so, we'll use the merge command. Still in command prompt, enter:

```bash
$ git checkout master
$ git merge upstream/master
```

The first command, git checkout, makes sure we’re on the main branch of our project. The second command, git merge, tells Git we want to combine the updated code from the original repository with our version of the project.

If you have any issues or questions regarding the merge command, please refer to section 5.1.4 on Merging a Branch.

### 5.3.5 Opening a Pull Request

Once we’re done adding a feature, fixing a bug, or making any of the changes to our version of the project we can tell the owner of the original repository and ask if they’d like to implement our changes with their original project.

To do this, we need to make what is called a pull request. There are multiple ways we can do this. One way is to navigate to the Github page for our version of the project. You’ll see there’s a new banner at the top of our code with a green button that says 'Compare pull request’. We can click this to start the process. The other way to initiate a pull request is to navigate to the original project’s Github page. There will be a grey button that says 'New pull request’ on the left-hand side right above the files of the project. We can click this to start the process.

Either way we start the process, the next steps will be the same. You’ll be taken to a page that says 'Compare changes’. There will be two drop down options, one that says base and one that says compare. We want to select the original project as our base, and usually we want to select the 'master’ version. For compare, we want to select the fork we created at the beginning.
Once we’ve selected both, the page will display if the files can be merged, and a space for us to explain our changes, as seen below:

The smaller text box at the top is like our standard commit message. We want to put a short, but descriptive explanation of what we added or changed.

The larger text box is where we want to leave a longer message, explaining the changes we made and why we think they should be implemented into the original project. This is our chance to be as descriptive as possible, as well as argue our case to why our code should be used.

The last step is to simply click the green ‘Create pull request’ button.

The creator of the original project will be notified of our request, and they can then look at our changes and our description of why we want our
changes implemented. If they decide they like the changes and want to keep them, they will accept the pull request and merge our changes with the original project.

The creator may decide not to accept our pull request, though. Do not take this personally, there could be plenty of reasons why they didn’t accept it, and they can respond telling you why as well. This could be because we didn’t write our code in the same format as the original project, and they may ask us to look at the contribution guide and re-write our code then make another pull request. The original creator may also already be working on the changes we made, so they could deny the pull request. This can be very common in large projects.

It may also take awhile for the original creator to respond to our pull request, to either accept or deny it. This is particularly true for very large projects, as the creator is probably already dealing with a ton of pull requests. Be patient, and wait for a response.

5.3.6 Handling Pull Requests

We’ve discussed how to make pull requests for other people’s projects, but what if someone makes a pull request for our own project?

Github makes handling pull requests incredibly simple and straightforward. On the main page for your repository, you’ll always see a banner on top of the files that looks like below:

You’ll notice there is a tab called Pull requests. If we click on this tab we’ll be taken to a page for our pull requests. If there are any open, it will be listed here. Click on it, and we’ll be taken to a page where we can decide to accept or deny the request. This will look like below:
Here we can look at the commits made by the person requesting the pull request. After the list of commits, you’ll notice a green button that says "Merge pull request". If we’d like to accept the pull request, and implement the changes into our project, then we can simply click this green button.

If we decide we don’t want to accept the pull request, however, we can scroll toward the bottom where you’ll notice a text box. This is where we can enter an explanation for why we are rejecting the pull request. Finally, to deny the request we will click the grey button that says 'Close pull request'.

ProTip! Add `.patch` or `.diff` to the end of URLs for Git’s plaintext views.
A Quick Reference

A.1 Configuration

$ git config --global user.name "UserName"
$ git config --global user.email "email@email.com"

A.2 Create Repository

Initialize local repository:

$ git init

Adding connection to Github:

$ git remote add origin https://github.com/user/repo.git

A.3 Making Changes

Check status of project:

$ git status

Add file to staging area:

$ git add fileName.ext

Add all files to staging area:

$ git add .

Commit files:
$ git commit -m "Commit message"

Saving changes to Github repo:

$ git push

### A.4 Branches

Create branch:

```
$ git branch branchName
```

Switch to branch:

```
$ git checkout branchName
```

Create branch and switch to it in one command:

```
$ git checkout -b branchName
```

Merging a branch:

```
$ git checkout master
$ git merge branchName
```

Pushing to a branch:

```
$ git push origin branchName
```

Deleting a branch:

```
$ git branch -d branchName
```

Viewing all branches:
A.5 Cloning

Clone existing project:

$ git clone https://github.com/user/repository.git

Add upstream:

$ git remote add upstream https://github.com/user/repository.git

Obtain updates from original repository:

$ git fetch upstream

Merge updates with current project:

$ git merge upstream/master