A Guide to Git

Andrew Au

Riley Shenk

Bethany Wong

University of Maryland, College Park

ENGL393 Section 0101

Table of Contents

Introduction

If you are a Computer Science major at the University of Maryland who has completed at least CMSC131, then you have been forced to work alone on all of your coding assignments. Consequently, if you were to go to your first internship or job and were asked to work with a team of software engineers, this would be a completely new experience for you. Though you may not be intimidated at first, you will quickly realize this introduces a lot of complexities to your job that you never experienced while working alone in school. By following this Guide to Git, you will learn the tools needed to positively contribute to the group from day one. Please note that the setup is geared specifically towards Mac users. However, git commands are universal for any operating system.

Git is a version control system often used in software development. Now, what is a "version control system"? Similar to the concept of Google Docs, a group of people can collaborate on an assignment, update different versions of a document to a common space, and keep track of changes. In git, users are able to take open source code, make changes, and update back into the repository (the common space) for collaborators to view and see previous versions.

This manual will provide step by step instructions on the basics of using git, specifically the installation process, creation of a repository, and necessary terminal and git commands. A simple practice project is incorporated for you to work on with a group of people to make sure you are able to apply the knowledge as you learn. Depending on your programming abilities, this manual may take approximately 1-2 hours to complete.

By the end of this manual, you can expect to become adept in git and be able to work on a programming project with a group of people. Ultimately, you will be prepared for the increasingly competitive job market, have valuable experience to boost your resume, and give off a good impression to recruiters.

After interviewing multiple upperclassmen and alumni who have been exposed to programming as a profession, they have emphasized that the single best thing you can do for your resume is to get involved in the open-source community by working on existing projects in GitHub. This shows employers that you are genuinely interested in software and that you love coding enough to work on projects that you are not paid to do. To do this, you have to know how to collaborate on a team made up of people you have never met before, probably will never meet, and people who potentially aren't even in the same time zone as you. Through completing this guide, you will have the tools necessary to start collaborating on any project that sparks your interest on GitHub.

Please do not use git or GitHub for your school's programming projects because this violates the school's academic integrity policy for sharing code. Getting caught doing so will lead to an XF for the course, which will deter recruiters and will revoke your diploma if you graduated and uploaded code from a major requirement course. Further details can be found here.

Git Informed

Git & GitHub

Git is a command line tool created by Linus Torvalds in 2005. A free tool that was very similar to git used to exist and was called BitKeeper. However, the relationship between the owners of BitKeeper and the open source community broke down, so BitKeeper became a paid service. In the wake of this, Linus Torvalds created an open source version of BitKeeper and called it git. Later, git became an open source project just like his earlier creation: Linux. Contrary to what people typically think, the word git does not stand for anything; git is merely one of the shortest pronounceable commands not yet taken by operating systems for other uses.

Github is a server for git repositories founded by PJ Hyett, Chris Wanstrath, and Tom Preston-Werner. Github's website was launched in 2008 and it has quickly grown to have 11 million developers contributing towards 28 million different software projects. Github provides a graphical user interface to use git, server space for users to store their code publicly for free, and paid server space to store code privately. Unlike git, which limits users to the command line, GitHub is available as a website, mobile app, and desktop application.

The Structure of Git

At a glance, git seems like a fairly simple concept: make changes on a project and publish these changes to a repository for others to elaborate on. However, once you start using git, things may get a bit confusing with all the commands, branches, and stages. In fact, git has been so frustrating for beginners to learn that comics such as **Figure 1**, have begun emerging to make stabs at its usability.

In this section, you will get a better understanding of the basic structures in git, specifically branching and merging. After understanding the foundation of how everything works, git can be less complex to work with.

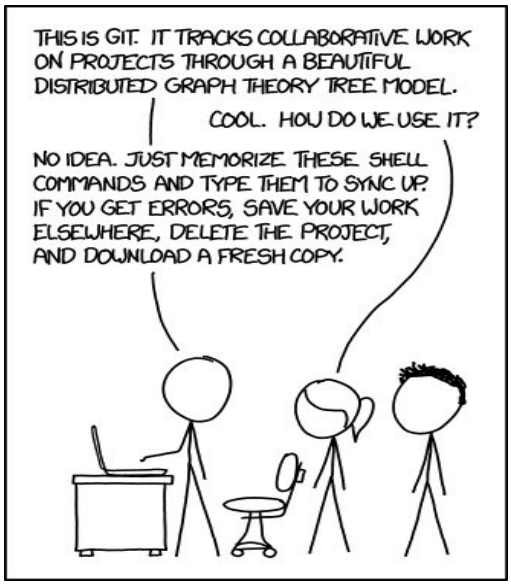

Figure 1: xkcd.com comic in reference to using git.

Branching

The main feature of git that makes it so widely used is its fast and memory efficient branching model. To make a **branch** means to create a copy of all the files within the repository that you are currently at. The new branch that has been created will be

independent of the original files within the directory until you merge the changes in the new branch to your original files. You could compare this to copy and pasting the folder that you are currently working on with a new name, except you can only open and make changes to one at a time. Afterwards, you can drag and drop your edited files to the original folder and replacing those files. This is much different than a tool like Google Docs where the changes you make are made immediately visible to everyone you are sharing your files with.

You initially begin in a master branch that is the root of your program and contains the original copies of the source code that everyone sees. When you are ready to start working on the repository you can make as many branches as you would like. **Figure 2,** illustrates this point by showing what it would look like to have a develop branch along with individual topic branches. When you are happy with the changes made in each of your branches git makes it easy to merge your changes back to the master branch. This guide will teach you how and when to make a branch and merge it back to the *master* branch.

Figure 2: A graphic representation of the branching model. Retrieved from git-scm.com.

The advantages of branching consist of the following:

- **Easier to test and experiment code.** Since branches are independent of each other, you can create branches purely for experimenting different code implementations. The best part is that if the code doesn't work, you can simply delete the branch and move on. Otherwise, with a few commands, you can apply your work to the main source code.
- **Efficiently collaborate on programming.** Instead of having to compare which lines you and your partner(s) changed or to work on the same and only copy of your program, you and your partner(s) can create branches with your own copy of the program. After completing your set of changes, git allows you to easily merge and identify changes among you and your partner(s) work.
- **Organize development.** Branches can be assigned roles, in which each branch can be dedicated to working on a feature of the program. That way, you do not

need to worry about making a feature runnable before being able to work on another feature.

Merging

We talk a lot about git being able to merge changes together, but how exactly does it work? There are two places you will need to merge: branches and the repository. In order to understand the mechanics of merging, you will first need to understand the levels in git and the different states of your file.

Levels in Git

Git is structured into three general levels that files go through: working directory, staging area, and repository.

- **Working directory.** This would be the branch you are working in. Here, you are creating additional files, implementing new code, or editing existing code.
- **Staging area.** Git refers to this area as the "intermediate area" where changes can be reviewed before committing it to the repository.
- **Repository.** This is where all your project files are stored and final updates are published to. Please refer to the note below.

Note: Throughout this guide, we will be using the terms "local repository" and "remote repository." The local repository is referring to your computer's main copy of the project files. Your local repository is only visible to you until you merge it with the remote repository. The remote repository is located on GitHub where your peers will be able to access the project files and view changes.

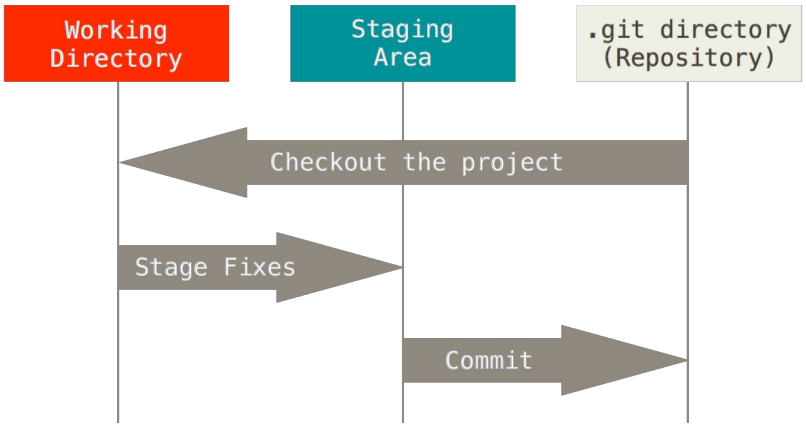

Figure 3.1: The different levels of git gives you the control to ensure that you add the changes you want and commit all the changes added to the repository. Retrieved from git-scm.com.

Figure 3.1 shows an example of how the levels relate to each other. Once you complete your changes in the working directory, you will add these altered files into the staging area for further review. After you are sure you want those changes to go through, your files leave the staging area once you execute a commit. Committing the files will sync the changes

to the existing files in your local repository and replace the old files with your new files and add any additional files.

States of Your File

Your files can enter four states before getting to the staging area: untracked, unmodified, modified, and staged. The file is either untracked or tracked, where tracked files are divided into unmodified, modified, and staged states.

- **Untracked.** Files in your working directory that were not added to the staging area in the previous commit.
- **Unmodified.** These are the files that have not been edited at all.
- **Modified**. These are the files that have been edited.
- **Staged.** Once you stage the modified files by adding them, the files are considered as staged and commits the staged changes.

Figure 3.2 provides a visual representation of the file state cycle and how the states interact with each other.

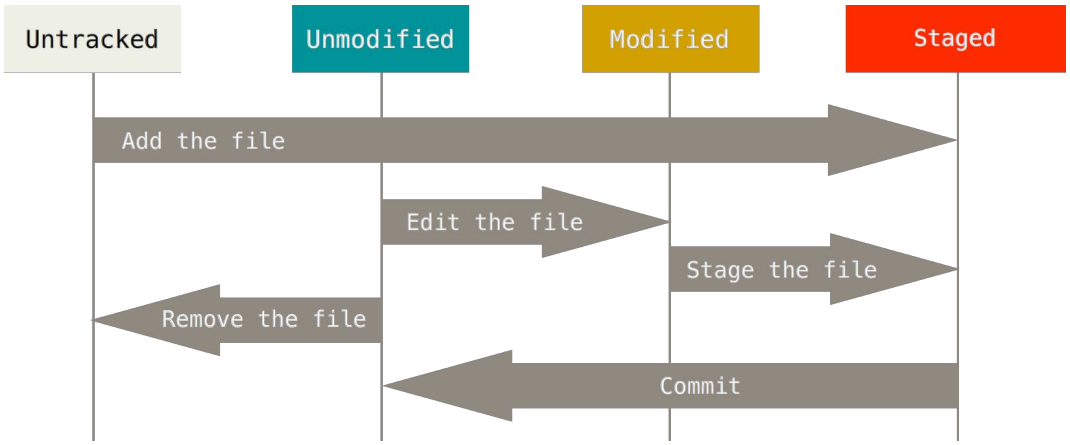

Figure 3.2: File state life cycle. Retrieved from git-scm.com.

Merging Branches

Now that you understand what happens when you make changes, you can learn how these changes are updated and synchronized with the main copies of the files. Before making your changes public to your peers, you need to merge your changes within your computer (your local repository). Changes within branches are not considered as "saved" until the master branch of your local repository is updated with those changes. In other words, the edits must leave the staging area and enter the repository. After committing the files from the working directory into the staging area, you can execute a merge among branches to synchronize the changes as shown in **Figure 2**. Ultimately, you will need to merge all desired changes to the master branch.

However, merging branches may not go through smoothly every time; you may encounter errors that say you have **merge conflicts** causing the merge to not go

through. These occur when you update a section in a file in one of your branches, but make a different update in the same section for the same file in another branch and git becomes unsure of which update to use when merging. You will need to address these conflicting changes manually in order to successfully merge the branches.

Merging Repositories

Your changes are still not published for your peers to view. In order to make all your changes public and accessible, you must merge your local repository with the remote repository. Once you have your changes saved in your local developer repository, you are ready to update your shared remote repository on GitHub.

After merging branches, the changes you have made within your local repository are unsaved by git in the remote repository. These changes are only saved in your local developer repository but not in the shared repository as shown in **Figure 3.3.**

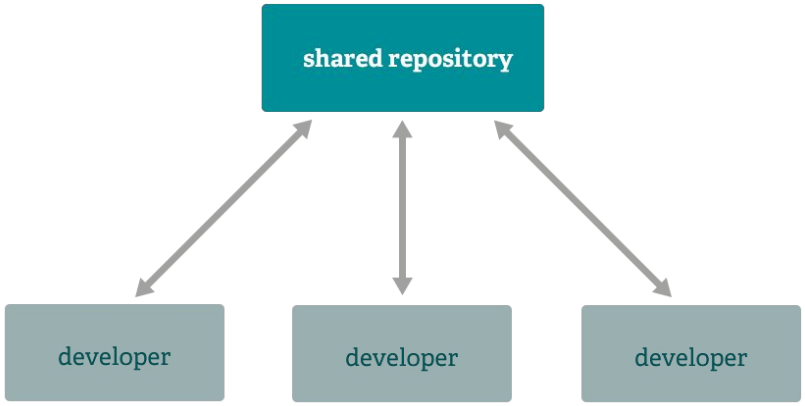

Figure 3.3: Each developer has their local copy of the repository which they can later sync with the shared or remote repository, which in most cases is stored on github.com. Retrieved from git-scm.com.

You can publish your local repository by executing a push to the remote repository. However, you may also encounter merge conflicts as you would when merging branches. In order to resolve these merge conflicts, you must retrieve the latest files in the remote repository by executing a pull. By retrieving these files, you will be able to see which files have a conflict and can edit them as you please.

Once all merge conflicts are resolved, you can successfully push your changes to the remote repository. If you forked a copy of the project files, this will consequently generate an optional **pull request**. A pull request is when you would like to merge your project files with the original project files you forked from. The owner of the project must approve of your request before your changes are merged with the origin.

Setting Up Your Computer To Use Git

Before you start learning how to use git, you will need to have the following installed and set up for your Mac:

- Ruby
- Homebrew
- Git
- GitHub account
- Java
- Text editor

Operating System - Version Verification

1. Check the version of your operating system by clicking on the Apple logo in the upper left corner of your screen.
2. In the drop down menu, click "About This Mac." Refer to **Figure 4.1.**

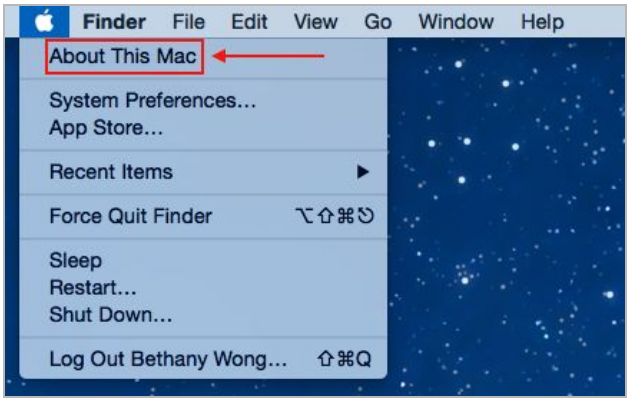

Figure 4.1: Follow step 1 and 2 to get to your system's information

   a. A window will appear with your system's information. As identified in **Figure 4.2**, your Mac OS X version is displayed at the top of the window.

Figure 4.2: Your system's information and specs are located here. You can see which version your system is running.

3. Take note of your version number as it will be used when installing git.

Terminal

Mac has a built in command line interface referred to as terminal. Terminal allows you to interact with your computer system's files. Thus, you will be using terminal to enter git commands (a command line tool) and interact with the project files saved onto your computer. You need to be familiar with how to open terminal in order to advance any further in this guide.

1. Search in Spotlight for the application called terminal. You can do so by pressing the following keys: ⌘ + `Space`, type "terminal," then hit `Enter`. You can also go into your applications to open terminal.

All commands you type into your terminal have the following format: `$ command`. Please type everything after the `$`.

Ruby

Ruby is an object oriented scripting language. For the sake of this guide, you will only be using Ruby to install a package management software identified in the next section.

1. Enter the following command to check if you already have Ruby installed.

```
$ ruby --version
```

2. If `command not found` prints, then install Ruby by entering the following command.

```
$ xcode-select --install
```

    a. Enter the following command to check that you have properly installed Ruby. A version number should appear to show that you have properly installed Ruby. Otherwise, restart your computer and repeat the process of installing Ruby from step 2.

```
$ ruby -version
ruby 2.0.0p481 (2014-05-08 revision 45883) [universal.x86_64-darwin14]
```

Homebrew

Homebrew is a package management software for Mac OS X. Homebrew allows you to easily install a variety of software. You will be using Homebrew (also referred to as "brew") within the Terminal to install the remaining software needed for this guide.

1. Paste the following command to download a Ruby script that will automatically install Homebrew:

```
$ ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

   a. Enter the following command to check that you have properly installed Homebrew. A version number should appear to show that you have properly installed Homebrew. Otherwise, restart your computer and repeat the process of installing Homebrew from step 1.

```
$ brew --version

0.9.5
```

2. Review the list of commands that brew has by entering:

```
$ brew -h
```

Note: The two brew commands you will need to know for this manual are search and install:

```
$ brew search $SOFTWARE_NAME
$ brew install $SOFTWARE_NAME
```

Git

You will need to install git, if not already installed, to import git commands to your system.

1. Refer to your Mac version noted in step 3 of Operating System - Version Verification. If your version of Mac OS X is 10.9 or higher, follow step 2. Otherwise, skip to step 4.
2. Enter the following command:

```
$ git
```

   a. Follow the instructions in the git OS X installer window (**Figure 5**) when it automatically appears. If it does not appear, skip to step 4a. Otherwise, skip to the GitHub section.

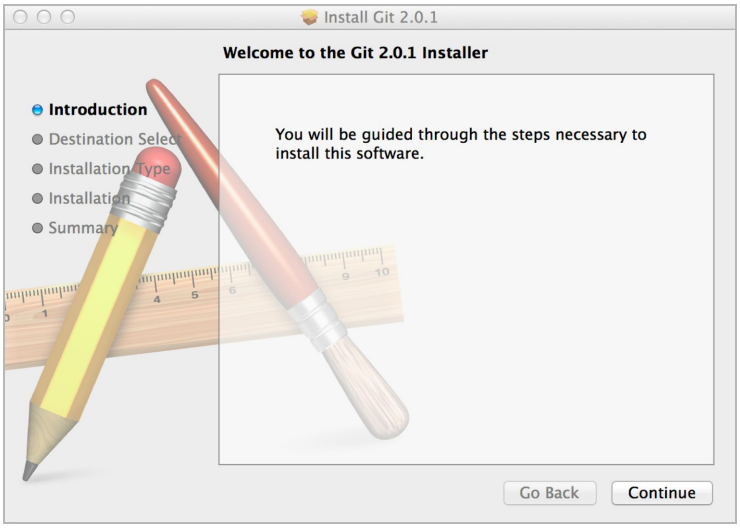

Figure 5: Git OS X installer.

3. Enter the following command to ensure that git is available for installation. This should display a list of available software including git:

```
$ brew search git
```

Note: A list of software containing "git" in each name will appear. Though all the names displayed are valid to install, you only need to install git.

4. Enter the following command to install git:

```
$ brew install git
```

    a. Enter the following command to check that you have properly installed git. A version number should appear to show that you have properly installed git. Otherwise, restart your computer and repeat the process of installing git from step 4.

```
$ git --version
git version 2.5.0
```

GitHub

GitHub is where you access your local repository for the project. You will retrieve the project files for this guide from the origin repository on GitHub. Beforehand, you need to create an account for GitHub, if you do not have one already, in order to access the files.

1. Go to github.com.
2. On the right of the frontpage, enter your desired username, email, and desired password.
3. Click "Sign up for GitHub."

Java

Java is an object oriented programming often used in software development. The guided project you will be going through with your team to practice git requires you to have Java installed to compile the program within the terminal.

1. Enter the following command into your terminal to check if you already have Java installed. If the terminal displays `java: command not found`, then continue to the next step. Otherwise, move onto to the Text Editor section.

```
$ java -version
```

2. Enter the following command to install the latest version of Java:

```
$ brew install Caskroom/cask/java
```

    a. Enter the following command to check that you have properly installed Java. A version number should appear to show that you have properly installed Java. Otherwise, restart your computer and repeat the process of installing Java from step 2.

```
$ java -version
java version "1.8.0_51"
Java(TM) SE Runtime Environment (build 1.8.0_51-b16)
Java HotSpot(TM) 64-Bit Server VM (build 25.51-b03, mixed mode)
```

Text Editor

While working on the project files, you will need a text editor to work in. If you already have a text editor that you are comfortable with, feel free to skip this section after reading the warning below. Otherwise, we suggest using Sublime.

If you are using your own text editor, please note that all the instructions throughout this guide are geared towards a Sublime text editor. Therefore, for any commands that use `subl`, go to the project folder saved on your computer to open the specified files for editing in your own text editor.

Note: For CMSC216, you might want to have a nice text editor to work with for your projects. It may be nice to set this up now.

1. Enter the following command into your terminal. This should show you that Caskroom/cask/sublime-text is available to install.

```
$ brew search sublime-text
```

2. Install sublime by entering the following command:

```
$ brew install Caskroom/cask/sublime-text
```

    a. Enter the following command to check that you have properly installed Sublime. A version number should appear to show that you have properly installed Sublime. Otherwise, restart your computer and repeat the process of installing Sublime from step 2.

```
$ subl -version
Sublime Text 2 Build 2221
```

Guided Project

Welcome to the Real World

You and your group are now going to be guided through a project that will simulate what it will be like to graduate from working on your own as a computer science major to now working in the real world on a team with members assigned to different parts of the product. You will be shown when to use the different git commands to accomplish your goal as a team. To get started on the job posting below you will need to decide a group leader. This leader will have a specific role to play in getting the project started for your team. Please note that the guide will only have you edit a single file (names.txt) from the project folder. If you would like, you and your group can practice more on the file, Human.java, to try and create a running program.

Job Posting

A young startup is trying to simulate population growth amongst an ethnically diverse isolated community. They made it off to a great start but once the source code grew large enough the simulation software grinded to a halt due to the team's inability to work together. They are looking for a new team who is motivated to pick up the unfinished simulation software and carry the project across the finish line.

The unfinished simulation is currently on GitHub here waiting to be finished. In the GitHub repository, there is a `Simulator` and `Human` class. The `Simulator` class is ready to go once a few functions in the `Human` class are finished. Before implementing these functions, you and your team will have to add your own names to the text file containing a list of the names of the project collaborators. You are required to use git to collaborate.

Things You Should Know

If you are already able to create, change, remove, list directories, and open files in a text editor using the terminal then you can skip this section.

Most of your work will be done in the terminal as well as the editor of your choice. These basic commands will help you navigate the terminal.

pwd	**p**rint **w**orking **d**irectory **pwd** will show you the directory that you are currently in.
ls	**l**i**s**t **ls** will list all of the directory contents. In most cases this will just be directories and files.
cd *$DIRECTORY*	**c**hange **d**irectory **cd** will allow you to move from your current directory into another directory. This is equivalent of double clicking on a folder in finder. You can also use `cd ..` to move to the parent directory of your current directory, like pressing the back button. You can also enter just `cd` to go back to your home directory.
mkdir *$DIR_NAME*	**m**a**k**e **dir**ectory **mkdir** will create a new directory in the current directory that you are in.
subl *$FILE_NAME*	**subl**ime text editor **subl** is the text editor that we are using in this guide, you can type it by itself to open the editor or follow up with a filename in order to open that file in sublime.
rm *$FILE_NAME*	re**m**ove **rm** will remove the file that you specify from your current directory. `rm` will also remove directories but you will need to add a the recursive flag to do this. Example: `rm -r $DIRECTORY_NAME`.

Cloning the Project to Your Personal Workspace

1. Choose a team leader in the group to be the owner of the repository about to be created. Only the team leader will complete step 2 while the others watch.

Fork will create a repository in your github account that is completely separate from the original repository.

2. ***Fork*** the project from the main repository.
   a. Go to **github.com** and log in.
   b. Go to the project's repository on GitHub located here: **https://github.com/rjshenk/git-guide**
   c. Click on the button that says "Fork" identified in **Figure 6.1.**

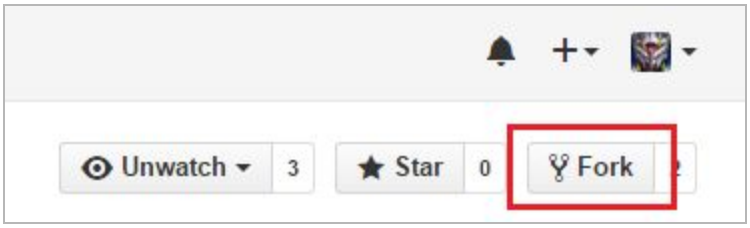

Figure 6.1: Fork located in the upper right hand corner below your profile picture.

   d. Open the settings tab in your copy of git-guide by clicking on the gear in step 0 of **Figure 6.2.**
   e. Follow step 1, step 2, and step 3 shown in **Figure 6.2** to add your team members to the list of collaborators. Once you have added all members, click on the tab labeled **<>** in the side menu where you found the setting tab. You should then be brought to your own repository under your profile.

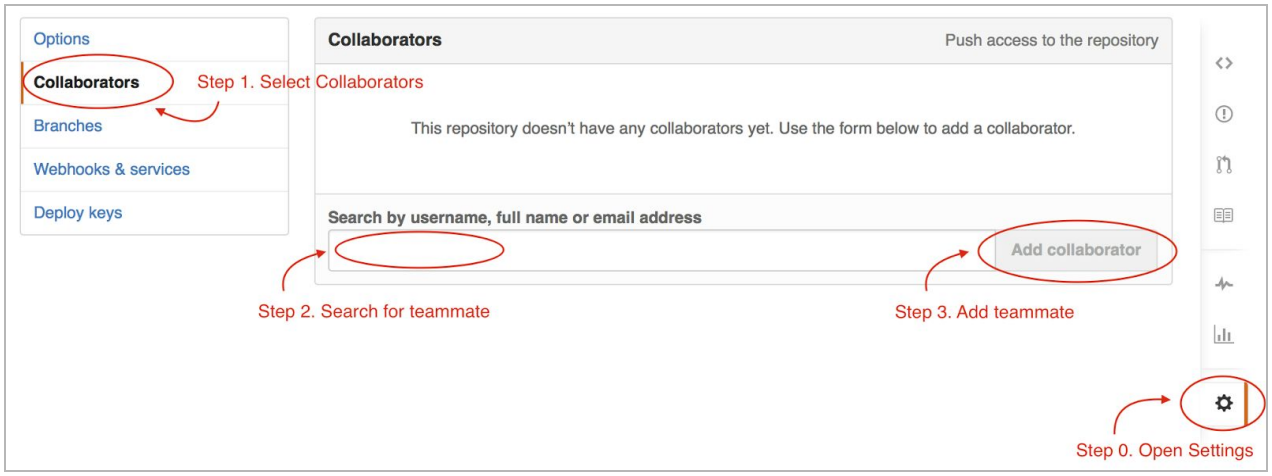

Figure 6.2: Steps needed to add a teammate. Repeat Step 2 and Step 3 until all team mates have been added

f. Locate the "**HTTPS** clone URL" link shown in **Figure 6.3** and click the "Copy to clipboard" button next to the URL.

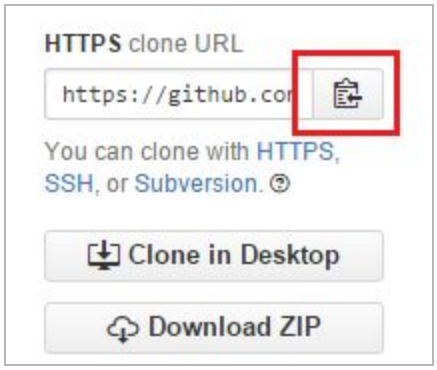

Figure 6.3: Copy to clipboard is located on the right side of the website.

g. Inform your teammates to copy the link identified in step 2f.

Clone will make a copy of the entire repository and put it in a folder your current directory. You can do everything with a cloned repository except push your changes to the original repository.

3. ***Clone*** the project to your personal computer.
   a. Open the terminal. (Shortcut: ⌘ + `Space`, type "terminal," then hit `Enter`)
   b. Change your current directory within the terminal to a working directory that you can save your project in by using the commands listed in **Things You Should Know**
   c. Enter the following command, but replace *`$URL`* with the url copied from step 2f and 2g.

```
$ git clone $URL
```

4. Move to local repository folder.
   a. List out the contents in your current directory to ensure that you have "git-guide" listed. Otherwise, change directories to wherever you download it (the project folder should be downloaded in whatever folder you performed step 3 in).
   b. Change your current directory to "git-guide".

Create Branches for Development

Branch can both list all the branches that have already been created and create a new branch under the name specified by *$BRANCH_NAME*.

1. Check current branches in working directory.
    a. Enter the following command to retrieve a list of branches created.

```
$ git branch
```

2. Create a branch from your current branch.
    a. Enter the following command, but replace *$BRANCH_NAME* with the name of your new branch.

```
$ git branch $BRANCH_NAME
```

Checkout will move you to the branch identified in *$BRANCHNAME*. After executing checkout, you will be able to work inside the specified branch.

3. ***Checkout*** the branch to actually work in it.
    a. Enter the following command, but replace *$BRANCH_NAME* with the name of your new branch created in step 2a.

```
$ git checkout $BRANCH_NAME
```

Open and Edit Project Files

1. Open the file that you wish to edit
    a. Display all the project files using `ls` in the command line.
    b. Enter the following command to open `names.txt` and then add your own name to the top of the file.

```
$ subl names.txt
```

    Note: If the filename given in the above command does not exist, one will be created for you with that name. For example, `subl new_file.txt` will create a new file called `new_file.txt` since it did not already exist in the directory.

2. Save and exit. (Sublime shortcut: ⌘ + s, ⌘ + w)

Publish Changes to Local Repository

Status will show files that have been added to the current index as well as those that are different from the most recent commit on the current branch.

Diff will display the type of changes at specific lines in the file in comparison to the original copy before making the changes.

1. View changes that you have made and check file states.
    a. Enter the following command to view all the changes you have made as well as the files' states. Since you only edited `names.txt`, you should see the following:

```
$ git status
On branch $BRANCH_NAME
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)

        modified:   names.txt

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

    b. Enter the following command to show the lines that were added, modified, or removed within `names.txt`.

```
$ git diff names.txt
```

Add adds files that have been listed to the staging area to be committed.

2. ***Add*** files to staging area
    a. Enter the following command to make `Human.java` not compile.

```
$ echo "unfinished code" >> Human.java
```

b. Enter the command, `git diff Human.java`, to view changes you made to `Human.java`. Now when you try `git status` you should see that both `names.txt` and `Human.java` have been modified.

```
$ git diff Human.java
git diff Human.java
diff --git a/Human.java b/Human.java
index ebed451..363541f 100644
--- a/Human.java
+++ b/Human.java
@@ -257,3 +257,4 @@ public class Human {
     }

 }
+unfinished code
```

c. Enter the following command to save the changes you made to `names.txt` and `Human.java`.

```
$ git add names.txt Human.java
```

Note: Initially, git users usually do not understand why git has the option to add changes separately instead of all at the same time. For future reference, there is an command to add everything using `git add --all`.

Reset removes the identified file from the staging area. This command is used after the file has been added to the staging area.

3. Display the files that are ready to commit.
   a. Enter the following command, `git status`, to observe that `names.txt` and `Human.java` are now in the staged section.

```
$ git status
```

   b. Enter the following command to remove `Human.java` from the commit queue because the last line is unfinished code that will not compile.

```
$ git reset Human.java
```

Commit Records the changes of files you've use git add on into your local repository on your computer. This will not update your remote repository located at GitHub.com.

4. ***Commit*** your changes.
    a. Enter the following command along with a brief and meaningful message explaining what changes you made in quotations marks as done below.

```
$ git commit -m "Added my name to the list of names"
```

5. Switch back to your master branch so that you can be in the right place to merge your changes.
    a. Checkout (switch to) the master branch by entering the following commands.

    Note: `git checkout master` is the command that will actually make you switch branches. To prove that this is true, use `git branch` before and after `git checkout master` to see how your current branch changed.

```
$ git branch
$ git checkout master
$ git branch
```

Merge combines the differences in the current branch with the branch name you give. As long as there are no conflicts, the merge will be done immediately.

6. ***Merge*** branches.
    a. Enter the following command to synchronize the changes in your current branch with the branch you worked in to edit the files.

```
$ git merge $BRANCH_NAME
```

    b. Enter the following command if you have no need for the current branch anymore to delete it. *$BRANCH_NAME* refers to the name of the branch you want to delete.

```
$ git branch -d $BRANCH_NAME
```

Retrieve and Update References in Remote Repository

Pull retrieves the latest changes in the shared remote repository and merges your old master files together with these new files.

1. Retrieve updated files from peers.
    a. Enter the following command to download all changes from the remote repository to your local repository.

Note: The following output assumes that one of your team members already made it past this point. If you are the first on your team to get to this command you will not have any conflicts. That is, if `git pull` does not generate the following output with a conflict in it, skip to step 2.

```
$ git pull
remote: Counting objects: 13, done.
remote: Compressing objects: 100% (12/12), done.
remote: Total 13 (delta 4), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (13/13), done.
From https://github.com/rjshenk/git-guide
   ae22458..c06fe70  master     -> origin/master
CONFLICT (modify/delete): simulation_output.txt deleted in
c06fe70cc4a7dd9103f3a4256dac4c3d49f8180b and modified in HEAD. Version
HEAD of simulation_output.txt left in tree.
Automatic merge failed; fix conflicts and then commit the result.
```

    b. Resolve the merge conflicts by entering the following command to view the first few lines of the file that has the conflicts and open it up for editing.

```
$ head names.txt
<<<<<<< HEAD
riley
=======
andrew
>>>>>>> 834f1bdb2d486be73343aa721524093955ad6dc2
aaron
abdul
abe
abel
abraham
abram
adalberto
adam

$ subl names.txt
```

c. Choose the changes that you made above the `=======` line or the changes that your group members made below the `=======` line. In this case you will want to choose a combination of both and delete the `<<<<<<< HEAD`, `=======`, and `>>>>>>> 834f1...`
d. Verify that names.txt does not have any `=======`, `<<<<<<<`, or `>>>>>>>`.

```
$ head names.txt
riley
andrew
aaron
abdul
abe
abel
abraham
abram
adalberto
adam
```

2. Add and commit names.txt again.

Push sends the changes that you've made to your local repository to the remote repository that you've pulled from.

3. ***Push*** your local references to the remote repository.
   a. Synchronize your local references to your remote repository such as your branches, commits, and changes using the following command:

```
$ git push
```

Submitting Changes to Original Owner

You must perform `git push` beforehand in order to submit a pull request. Otherwise, GitHub will see that you have nothing new to synchronize with the original copy of the project.

1. Create pull request.
    a. Go to your project's repository on GitHub.
    b. Click on the link labeled "Pull requests" as shown in **Figure 7.1.** This will bring you to a Pull Request manager.

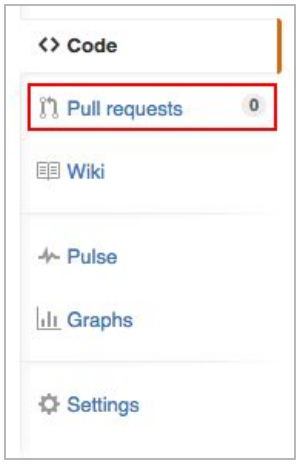

Figure 7.1: "Pull requests" link located on the right of the screen.

    c. Click on the button labeled "New Pull Request" as shown in **Figure 7.2.** You should now see a page comparing the changes between your master branch and the original copy's master branch as shown in **Figure 7.3.**

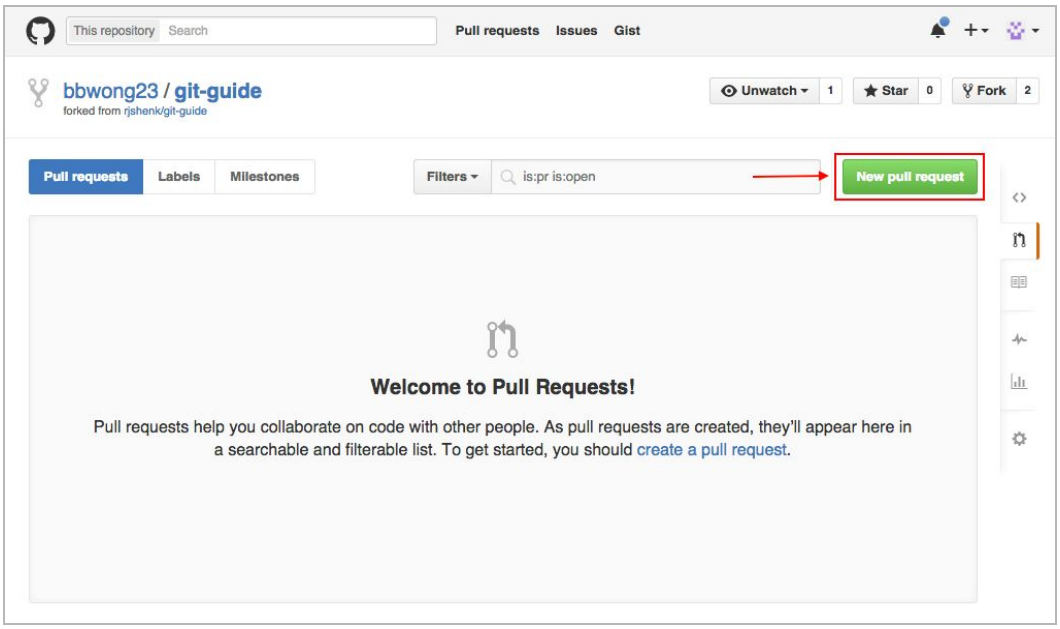

Figure 7.2: "New Pull Request" button located on the upper right corner of Pull Request Manager.

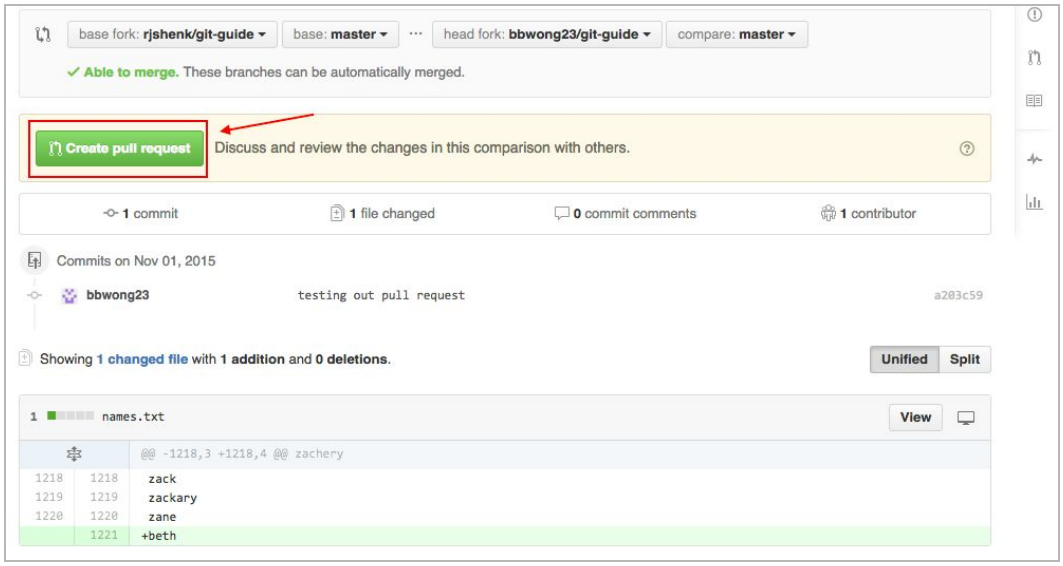

Figure 7.3: You can view your changes in comparison to the original project files. You must submit a pull request to the original owner to merge your changes with the original project.

e. Click on the "Create Pull Request" button as shown in **Figure 7.3** to request the collaborator of the original copy to merge your changes.
f. Name your pull request and enter a brief comment about the changes you have made for the owner to understand what you changed. Click "Create Pull Request" to complete the pull request. Refer to **Figure 7.4.**

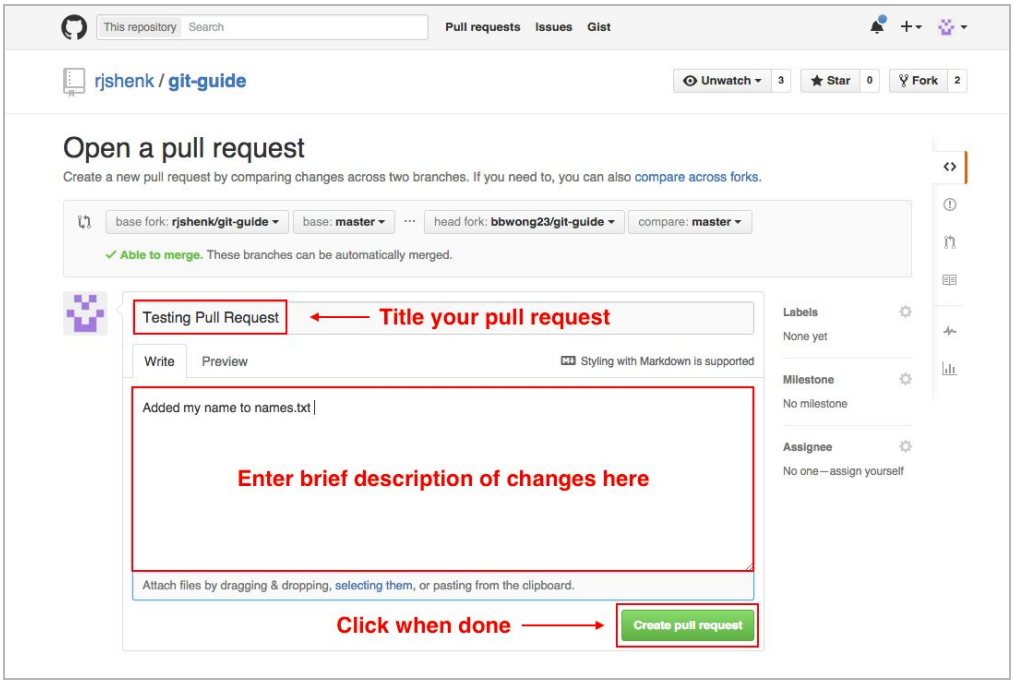

Figure 7.4: Pull request form.

2. Wait for the original owner to accept your pull request. For educational purposes, below is the set of steps need to accept a pull request.
    a. Follow step 1a and 1b.
    b. Click on the title of the pull request listed in the Pull Request Manager to view details of the pull request. Once you click the title, you can see the changes made in comparison to your current copy of the files.
    c. Click the "Merge Pull Request" button as shown in **Figure 7.5** and then click "Confirm Merge" to complete the merge.

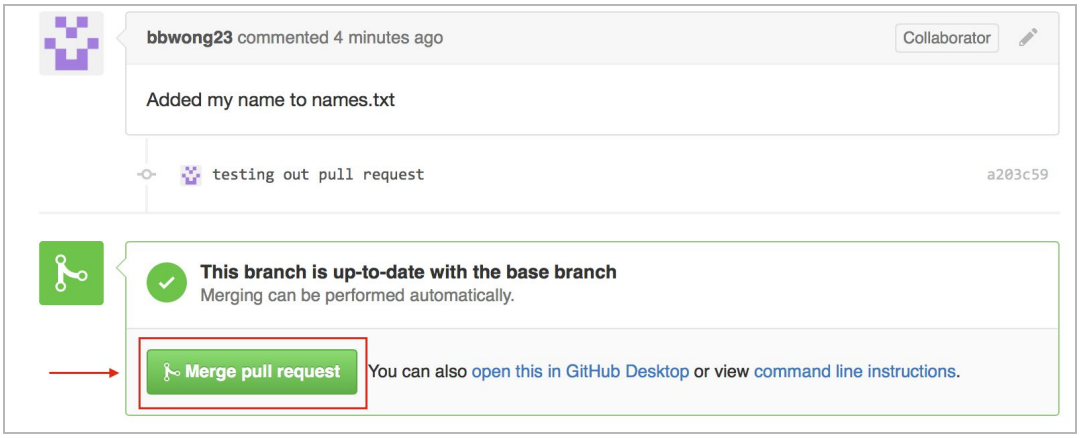

Figure 7.5: After approving of the changes, your project files will automatically update with the changes..

Note: As a collaborator, you also have the power to deny pull requests or just ignore them if you do not want to merge the changes in the pull request.

Project Specs

Project Directions

Your team will need to implement a few functions that were left empty by the previous development team. After having cloned a repository from your team leader's fork you will need to branch off of that master branch and name it *$FIRST_AND_LAST_INITIALS*.

Because there should be three team members it is recommend that one person work on the `introduce()` and `marry()` functions, another does `divorce()` and `giveBirth()`, while the last does `getJob()` and `leaveJob()`.

In working on a function, you are expected to branch off of your *$FIRST_AND_LAST_INITIALS* branch for each function that you are working on and name that branch *$FUNCTION_NAME*. You will make changes and commits to this branch for the function you are working on.

You can check for compilation errors in your code by typing `make` within the terminal and seeing `java Simulator > simulation_output.txt`.

Once you believe that you have a working version of that function, go back to your *$FIRST_AND_LAST_INITIALS* branch and merge with the *$FUNCTION_NAME* branch that you have completed.

Finishing Up

Once you have implemented the functions that were designated to you and merged their respective branches into your master, type `make` into your terminal one last time to see if everything is compiling properly. This is important because you do not want to send non-compiling code to the remote repository that you and your groupmates share.

You will then want to follow the steps to push your local repository into your group's remote repository. Once everyone has done this, you may pull from the remote repository all of the changes your group members have made and type `make` again.

To see the creation of your group's completed project you can view the output by typing `subl simulation_output.txt`.

Human Class

This is the class in which you will be implementing functions. Before you start, these are the instance variables that you will be using throughout your implementations:

`public String name`
: The name of the current `Human` instance, this can be a first name or a first and last name.

`private int age`
: The current age of the current `Human` instance, babies are born at age 0, otherwise Humans may be created at any age.

`public int health`
: A value that randomly increases and decreases as the `Human` ages, if this value is less than or equal to 0, the `Human` dies

`public String ethnicity`
: The ethnicity of the current `Human` instance, babies will be either the mother's or the father's i.e. no mixed ethnicity

`public String gender`
: The gender of the current `Human` instance, baby's gender are randomly decided between male/female, otherwise the human can be created with any gender.

`public Human spouse`
: The spouse of the current `Human` instance, this project does not discriminate and so allows same sex marriage.

`public LinkedList<Human> children`
: A list of the children that belong to the current `Human` instance.

`public LinkedList<Human> friends`
: A list of the friends that the current `Human` instance has.

`public boolean employed`
: The status of employment for the current `Human` instance, employment is normally determined by if salary is greater than `0`.

`public int salary`
: The amount of money the current `Human` instance is making per year.

Human Functions

You must implement the functions described below. Notice that many functions are already given to you and only need to change what is described below. Note that that functions do not depend on each other and so when working on each one you will only need to reference the `Human` constructor.

1. `public String introduce()`

   This function will return the `String` "Hi, my name is *`$NAME`* and I'm *`$AGE`* years old" where *`$NAME`* and *`$AGE`* represents the name and age of the `Human` calling `introduce()`.

2. `public String marry(Human toWed)`

   This function will set the spouse field of both the `Human` calling `marry()` and the `Human, toWed,` to equal each other if both do not yet have a spouse and both `Humans'` age is greater than `18`.

   This function will return one of four `Strings`:

   - "*`$NAME`* is too young to be married" if either the `Human` calling `marry()` or the `Human` being passed in is less than `18` where *`$NAME`* will take priority on the `Human` calling `marry()`.
   - "*`$NAME`* is trying to marry (himself/herself)!! Unfortunately, (he/she) is still single" where *`$NAME`* is the `Human` calling `marry()` while (himself/herself) and (he/she) is chosen based on the gender of the `Human` calling `marry()`.
   - "*`$NAME`* is already married" where *`$NAME`* will be the name of the `Human` that is already married where *`$NAME`* will take priority on the `Human` calling `marry()`.
   - "*`$NAME1`* and *`$NAME2`* are now married" if the marriage was successful where *`$NAME1`* is the name of the `Human` calling `marry()` and *`$NAME2`* is the name of the `Human` being passed in.

3. `public String divorce()`

   This function will set the `spouse` field of both the `Human` calling `divorce()` and his/her spouse to `null` if they are already married.

   This function will return one of two `Strings`

   - "*`$NAME1`* and *`$NAME2`* divorced" if the divorce was successful where *`$NAME1`* is the name of the `Human` calling `divorce()` and *`$NAME2`* is the name of their spouse.
   - "*`$NAME`* isn't even married" if the divorce is not successful where *`$NAME`* will be the name of the `Human` calling `divorce()`.

4. `public String giveBirth(Human baby, ArrayList<Human> kids)`

   This function will welcome a new `Human` into the world; this can only happen when the `Human` calling `giveBirth()` is a female that is married to a male. The new `baby` will need to be added to the the mother and father's `children` list.

   This function will return one of three `Strings`:
   - "*`$NAME`* has to be a female to give birth" if the `Human` giving birth is a male where *`$NAME`* will represent the `Human` calling `giveBirth()`.
   - "*`$NAME`* can only give birth if her spouse is a male" if the spouse of the `Human` giving birth is a female where `$NAME` will represent the `Human` calling `giveBirth()`.
   - "*`$NAME1`* and *`$NAME2`* gave birth to a baby named *`$NAME3`*" if `giveBirth()` is successful where *`$NAME1`* is the name of the `Human` calling `giveBirth()`, *`$NAME2`* is the name of the spouse, and *`$NAME3`* is the name of the `baby`.

5. `public String getJob(int money)`

   This function will set the `employed` and `salary` fields of the `Human` calling `getJob()` to `true` and `money` respectively if that `Human` does not already have a job.

   This function will return one of two `Strings`:
   - "*`$NAME`* found a job that is paying (him/her) *`$SALARY`* per year" if `getJob()` is successful where *`$NAME`* is the name of the `Human` calling `getJob()` and *`$SALARY`* is the amount of `money` that the job is paying; the `String` being returned should also use only either(him/her) depending on the `Human's` gender.
   - "*`$NAME`* already has a job" is returned if the `Human` calling `getJob()` is already employed where *`$NAME`* is the name of the `Human` calling `getJob()`.

6. `public void leaveJob()`

   This function will set the `employed` and `salary` fields of the `Human` calling `leaveJob()` to `false` and `0` respectively if that `Human` already has a job to leave.

   The function will return one of two `Strings`:
   - "*`$NAME`* has left (his/her) job" if the `Human` is employed where *`$NAME`* is the name of the `Human` calling `leaveJob()`.
   - "*`$NAME`* does not have a job (he/she) can leave" if `leaveJob()` is unsuccessful where *`$NAME`* is the name of the `Human` calling `leaveJob()`. In both cases (his/her) and (he/she) should be chosen depending on the gender of the `Human`.

Glossary

Terms

Branch

A branch is a separate copy of the repository that exists in parallel with the other branches. When creating a branch, git creates a new copy of each file in the repository that you can edit independently without affecting the original source code.

Clone

A clone is a copy of a repository that lives on your computer instead of on a website's server somewhere. Because this is a local copy, you will need to sync with the online copy through the use of push and pull.

Collaborator

Someone who has access to make changes to the main development repository of a project.

Commit

A commit is an individual change to a file (or set of files). Commits will have a unique ID that also store the user who made a commit as well as the changes that were made.

Contributor

Someone attempting to add to a project but does not have permission to do so and has to request permission to make changes from a collaborator.

Fork

A fork is a personal copy of another user's repository that lives on your account. This allows you to be the collaborator of your own repository instead of a contributor of the repository you forked off of.

Git

An open source program for tracking changes in text files mostly used in collaboration between programmers.

GitHub

A social and user interface presented as a website that is built on top of git.

Homebrew

A package management software for Mac OS X systems written in Ruby. This software allows users to easily install other software to the Mac operating system using a Ruby script.

Java

An object oriented programming language that you will be using to implement your code for the project integrated into this guide.

Master

The main branch of a repository; eventually, all branches will merge back into this one.

Merge

Taking the changes from one branch and applying them to another.

Merge Conflict

When changes to the same location in a file are made by multiple users or branches, merge conflicts will develop in the form of:

```
<<<<<<<<<<<< $BRANCH_NAME

$YOUR_CHANGES

============

$TEAMMEMBER_CHANGES

>>>>>>>>>>>> $COMMIT_ID
```

You will be tasked with deleting everything except either $YOUR_CHANGES or $TEAMMEMBER_CHANGES. A combination of the two changes is also possible.

Modified

One of the four file stages to describe files that have been edited and modified, including newly created files within the working directory.

Pull Request

When working on a forked copy of a set of project files, you can submit a pull request for the original owner of the project to pull in your changes and merge them together with the original files.

Repository

A project folder that contains all project versions and branches. This is what collaborators and contributors work on.

Ruby

An object oriented programming language. For the sake of this guide, you will only be using Ruby to install the software, Homebrew.

Staged

One of the four file stages to describe files added to the staging area and are ready to be committed to the repository.

Staging Area

The “intermediate area” where changes can be reviewed before committing them to the repository.

Sublime

Our recommended text editor that you can use for writing and editing code.

Terminal

Mac’s command-line interface that allows users to interact with the computer system files.

Unmodified

One of the four file stages to describe files that have not been edited.

Untracked

One of the four file stages to describe files in your working directory that had to be removed during the last commit to the staging area.

Working Directory

The lowest level of development in git. Refers to the branches where you are creating new files, implementing code, or editing existing files.

Commands

git add $FILE_NAME_1 $FILE_NAME_2 $FILE_NAME_3...

Add all files listed to the staging area to be committed.

git branch $BRANCH_NAME

Creates a copy of the current branch that you are working in and names it *`$BRANCH_NAME`*.

git checkout

Switches to another branch; normally used to work on different features of a project.

git commit -m "$COMMENT"

Records the changes of files you've use git add on into your repository, this will not update remote repositories.

git diff

Shows the changes between your current files and your last staged files. Adding `HEAD` to the end of the command will show the differences between your current file and the most recent commit.

git merge $BRANCH_NAME

Combine the current branch with *`$BRANCH_NAME`*, such that changes made to the current branch are reflected in *`$BRANCH_NAME`*.

git pull

Retrieve the latest updates from the remote repository that have been made by other developers working on the same project.

git push

Sends the changes that you've made to your current repository to the remote repository that you've pulled from.

git reset $BRANCH_NAME $FILE_PATHS

Moves the current head to another branch so that all files added to the staging area are no longer indexed anymore. Will also remove files from the staging area if a branch is not given or `HEAD` is given in its place.

git status

Shows files that have been added to the current index as well as those that are different from the most recent commit on the current branch.

References

"Git." Git. N.p., n.d. Web. 01 Nov. 2015.