Anyone working a programming job will be familiar with GitHub and similar online repositories and workflows. GitHub is easily one of the most recognizable and popular of these online software version control services with added features such as bug tracking, community engagement through feature requests, continuous integration, wikis, tutorials, guidelines, and more. By providing all of these features in cohesive website (or desktop application through GitHub Desktop) with the power of Git, GitHub has secured itself as a mainstay for all programmers, and will likely stay this way with it’s continued support from Microsoft and open source programmers worldwide.

When it comes to utilizing the tools provided by GitHub, it is important to be aware of how exactly it works. The first step is always to initialize the space where Git will record changes. This will typically be in a form of a specific folder, and can include nearly all file types within it. By initializing the workspace, Git will track the data within the folder using a hidden *.git* directory. However, changes are only stored when the user commits the data. The user must select which files are to be committed by adding them to the staging area using the *git add …* command. Once the stage is set, the changes can be committed by using the *git commit -m <"message">* command. The user must include a descriptive message with every commit. In this process, it is important to continuously check the status of each folder to avoid potential conflicts by using the *git status* command and the *git log* command to track previous changes.

When committing code, these commits are staged within a specified branch. For simple projects, the user may usually only work with the master branch, since they are the only one working on the code. For more advanced projects, there are likely to be many different branches which operate on different versions of the source code. For example, one programmer may be trying to deliver a hotfix for a website as another programmer is attempting to develop the site further, unaware of the hotfixes. In this scenario, the programmer working on a hotfix can create an individual branch to be added later, while the developer can implement their work on a different branch for when the website is ready to be updated. Thus, branches are used to access different versions of the source code at different intervals. They can be initialized using the *git branch <branch-name>* command. To quickly change to an existing branch, use the *git checkout <branch-name>* command, or *git checkout -b <new-branch-name>* to create a new branch and enter it simultaneously. When using GitHub integration, the user can pull code from a branch to access the data within it or can push their code to the branch to upload their changes.

Conflicts can arise, however, when attempting to merge these branches into the main source code, also known as the master or main branch. Merge conflicts typically arise when branches contain overlapping changes of specific code. When these conflicts occur, there will usually be a description of the error to help guide the resolution. As previously stated, many of these conflicts can be caught by using the *git status* command frequently. In some cases, it might also be beneficial to ignore some files by using the *git ignore <file>* command. To help remedy these conflicts, it is wise to stop the merge from happening by using the *git merge –abort command*. If necessary, it is also possible to return to the previous saved state by using the *git reset* or *git revert* commands. It is important to coordinate with fellow programmers to ensure that these conflicts do not arise. By learning more about these conflicts, programmers can hope to avoid these conflicts altogether and create a stronger collaborative environment.