­­­

Tutorials to help you succeed in version control software

Clint Goodman

git Going

Contents

[Setting up 2](#_Toc419405599)

[What is Git? 2](#_Toc419405600)

[GitHub – Registering and Cloning 4](#_Toc419405601)

[Commits 6](#_Toc419405602)

[Commits – Part 1 6](#_Toc419405603)

[Commits – Part 2 8](#_Toc419405604)

[Reverting commits 10](#_Toc419405605)

[Branching 11](#_Toc419405606)

[New branches 11](#_Toc419405607)

[Merge Conflicts 13](#_Toc419405608)

[Pull Requests 14](#_Toc419405609)

[Increase understanding 15](#_Toc419405610)

[Searching history 15](#_Toc419405611)

[Stashes 16](#_Toc419405612)

[Remotes 17](#_Toc419405613)

# Setting up

## What is Git?

Git is a version control program that allows you to save and track changes of files. It is particularly helpful when many people are working on the same set of files, and you want to keep track of who has done what, and whose code to accept. It is also helpful to rollback to any previous point in time and for programmers to be able to integrate their code well as a team.

Perhaps the best way to understand the importance of version control is to show what life would be like without version control. Let’s take the example of programming newbies you are trying to program a project together. To work together, they need to share files between each other easily. Obviously, they could email each other their updated files and replace them on your local machine, but doing so could create the following issues:

* 1. What if teammate A overrode some code that teammate B had worked on? How would either of them know when that happened, and how could they restore accidental overwrites? They can’t with email.
  2. What if 3 people emailed each other at the same time with the same file - how do they know which one is the newest, latest and greatest? They won’t.
  3. If what somebody did broke the whole code, how could they roll back to the latest working point? They can’t.

These same problems occur whether you use email, Google Drive, or any method of sharing besides version control.

You will see as you proceed through the tutorials that Git solves all of these problems, and provides dozens of great benefits and tools to make programming easier for everyone. Don’t get frustrated if you don’t see the bigger picture of Git at first – focus on understanding the next few lessons, and the bigger picture will fall into place.

There are two pieces of software that we will be using in these tutorials. One is Git itself, and the other is a user interface friendly program that executes Git statements so that you don’t have to use a command-line interface. Of course, everything in Git is available through the command line, but I strongly suggest using Atlassian SourceTree – a free software tool – to help you easily see and use Git.

To install Atlassian SourceTree, go to: <http://sourcetreeapp.com/>. Accept default installation settings, which includes installing Git. SourceTree uses Git, it means nothing without Git.

One last thing to note: every company that is evenly slightly coherent at programming uses some type of source control. Learning these skills is not just a good idea – if you ever want to be useful in this field, it is essential to learn the basics of version control. You will not succeed in a programming career unless you understand version control. So let’s git going!

## GitHub – Registering and Cloning

Git stores your project of code in “repositories”. Repositories can reside in two places: locally (on your computer) and on a remote server. You make changes to your project locally, and then you will “push” these updates to the remote server so that other team members can “pull” those updates. This is all terminology that you will learn in the next few tutorials.

Many companies host their own remote server code repositories. But as students, we will take advantage of a free hosting service known as GitHub, which will act as our free remote server. GitHub is a very common online resource, and the free version allows all Git features, except for private repositories.

First, we need to register an account on GitHub. This is possible by following the steps online at: <https://github.com/>. Pause this tutorial, and complete the registration before continuing.

[break]

Now let’s create a repository for our team project right now.

* Go to: <https://github.com/>
* Click on “+ New Repository”
* Enter in a repository name. Name it after the name of your team’s project, keeping in mind that the name must be web safe, so spaces and other characters will be escaped.
* Give your project a brief description. This will be used to automatically create a project README.md file, if you click that option.
* .gitignore files act like exemption index files. These can be very useful. On GitHub, there are several options depending on the coding language you will be coding in. I suggest that you pick your project type here, and allow Git to create that code-specific .gitignore file for you. Later on we will learn about editing our .gitignore files, and how to get the most out of them, so don’t worry about making a permanent, irreversible decision here.
* Click on “Create repository”

Your repository is now hosted online at GitHub.

Now open up SourceTree on your machine. You will notice that the program seems blank, because you have no repositories open to work on. It is time to get a local copy of the repository from GitHub so that you can make changes to it.

First, we will need the remote address of your GitHub repository:

* Go to: <https://github.com/>
* Click on your newly created repository
* Notice on the bottom left the link to your remote repository. Copy this address.

We will now clone this repository to your local machine:

* Return to SourceTree
* Click on “File” -> “Clone / New”
* Paste in the remote address to your GitHub repository in the “Source Path / URL:” box
* In the destination box, pick a local directory that you want to store this project in. Make the decision now where you want this repository to reside all semester. I do my development all in a directory I created at “C:\development\”, but you may do whatever fits your preference.
* Make sure that “Bookmark this repository” is checked
* Click on “Clone”

SourceTree will then download all the files you have created online to your local machine (which should only be two at this point – the .gitignore and the readme.md).

What we have just done is called “cloning a repository”. Cloning is used when you and your teammates want to work on the same project together. Another process you will hear of is called “forking a project”, which is similar to cloning a project, except that the repository will first be cloned to your personal account before you can then clone it. We will not be forking in these tutorials, but I did want you to hear the term as you will come across it in online threads and information databases.

Now you have a repository initialized on a remote server, and have cloned that repository to your computer. Typically a single repository holds a single project, and it is possible that in your careers you will be downloading several repositories and making changes to them separately.

Last, I want to show you what is happening to the directory you chose to be your local destination. Go to that directory on your computer. Once you deemed that directory to be your local repository container, Git is always checking to see if any changes are made to that directory. Changes include adding files, editing files, or removing files. Renaming files, according to Git, is classified as removing a file, and then adding a new file with a different name. If any changes are made anywhere inside of the project directory (even many subfolders deep), then Git/SourceTree will notify you of those changes, and allow you to either discard or save (called committing) those changes. This will be covered in the next tutorial.

The last term to understand of this tutorial is the term “HASH”. Hashes are unique identifiers of some data that has just gone through an algorithm. Hashes are much smaller than the original data, resulting in anywhere from a few to several characters long strings.

The way that Git knows if you have changed a directory is by using hashes. Right now, your directory and everything inside of it (the two files at this point) have been hashed, producing a hash code. The hashing algorithm that Git uses is the SHA-1 hashing algorithm. If you make a change to that directory, then the hash will be different from what it was at the latest saving point (known as a commit). Git becomes aware that something has changed from the difference of the hash, and allows you to decide what to do with that change. Every commit in Git has an associated hash, which we may be useful to you as you search your repository in later tutorials.

# Commits

## Commits – Part 1

Wouldn’t it be nice if every time you pushed save in Microsoft Word, a version of your document was benchmarked, so that you could go back and revert your document back to any point in time that you wanted? This is one of the most important advantages that Git gives you – the ability to make save/restore points, so that you can restore your file back to any time in history if it ends up breaking something, or if you accidentally delete it. In this section, we will be talking about the basics of committing files in Git. This will be best done by showing you an example.

Right now I have cloned my project from GitHub, as shown in the last tutorial. Currently, the repository is empty. I’m going to create a project in NetBeans, and store the project in the directory of my repository. Once I do, Git notices that something has changed within our directory, and notifies me of the different file changes. In this case, each file is new and did not exist before, which is indicated by this blue question mark. Git is telling us “I have never seen this file before. Would you like to add this file to your repository, and begin tracking changes of it?” Until you add this file to the repository, it will not begin tracking changes on it. So we want to add it as soon as possible to the repository to take advantage of Git’s ability to track changes and make restore points.

Notice at the bottom of SourceTree that they are two areas: the staging area, and the unstaging area. We’ll be working in this section to commit changes. The steps to add a file to a repository are:

1. Select the file(s) in the “Unstaged files” list
2. Drag them to the “Staged files” area. The staged area is similar to a “print preview” – it is showing you the change you are about to commit to the repository.
3. Click on the “Commit” button
4. You will notice that on the right is a line by line preview of the files you are adding. At the bottom is a space for you to put in comments about your update. You should, and are sometimes required, to add a comment for every commit that you every make. This is good practice that you give a brief summary of the change you are making.
5. Click on the “Commit” button on the bottom right

What you have just done is take a snapshot of your code at this moment, and a preview of the changes of that snapshot are shown to you in your tree explorer. You will notice that your name is attached to that commit, as well as the comment and the unique SHA-1 value for that commit (that’s not too important right now).

You can add more files by doing the same process. It is not required that you commit all the changes available to you in the “Unstaged files” list – only select the files and changes that you wish to commit. At this point, if you are seeing many more files than my list is showing in the “Unstaged files”, it is because your .gitignore was setup differently than mine. That is fine. We will cover how to customize that later.

So now let’s say that we make some changes to our Java class, and have reached a logical savepoint, and we wish to commit this snapshot. First of all, you will notice that Git has noticed that you have changed your file. Unlike the blue question mark, it now shows yellow dots. This is because your file has been added to the repository, and Git is now tracking changes made to that file. You will notice that if you click on the filename in the “Unstage files” list, you will see a preview of the differences between what you have done and what was last committed. Red lines indicate lines of code that have been deleted, and green lines indicate lines of code that are being added. Git tracks a file line by line, so you will only see the individual lines that have been changed. This is a big advantage to you as a programmer.

I can see in this preview that I made a mistake in my file, and I don’t want to save that in this commit. I can go back to the file in NetBeans, edit it, and remove these lines of code. But to make things simple, SourceTree allows you to select the lines that I don’t like, and click on “Discard lines” (“Reset” on a Mac). “Discard lines” does not mean it is going to get rid of those lines forever, but rather to get rid of the changes that are occurring to those specific lines. In this case, since I was adding these lines, “Discard lines” will simply remove the change to add them, which deletes them from my file. You will see now that if I go back to my file, those lines are now gone, so be careful about doing this, because you cannot get back changes that weren’t committed to the repository.

Now that I have the code at a better save point, I’m going to do a similar process as we did to add the files to the repository. I will click on commit, drag the changes I want to the “Staged files” area. I will add a commit message and click on commit. Notice that there is a new commit added to your tree explorer, and that those changes are no longer showing up in the “Unstaged files” section. That is because the “Unstaged files” section only shows differences between your current files and what was last committed, and at this point we haven’t changed the files at all since we last committed. The moment that we change one of these files, it will show that change in the “Unstaged files” list, and we can do similar things to discard or commit these changes. If we don’t like what we have done with the entire file since we last committed, we can simply right click the file in the “Unstaged files” and click on “Discard” (“Reset” on a Mac), which doesn’t mean that it will delete the entire file, but rather it will discard all changes made to the file since the last commit.

Let’s say you want to delete a file. If you delete the file from your computer, it is not actually “gone” from the Git repository. We can stage this change, and commit it just as we would any other change to the file. If we do so, you will notice that a new commit is added to the repository. However, remember that once a file has been added to the repository at any point in time, it is possible to recover that file by simply undoing this commit of deleting the file. This will be explained to you in a future tutorial, but you should feel safe in knowing that if the file you deleted was a mistake, you can always restore the file again, thanks to Git.

As I click on the different commits in the tree explorer, we can see how Git functions. It tracks all these changes, and simply makes a snapshot of the change. We can see what my username has done, the attached message explaining my work, and a preview of each change.

## Commits – Part 2

First, let’s review everything that we have learned about committing up to this point.

We learned that a file needs to be added to a repository before individual changes to the file can be tracked. We add a file by moving it from the unstaged area to the staged area. After previewing and making sure that we like the changes we are about to save, we add a message explaining what we have done and commit this update to the repository.

As we make changes on the file, we can commit these individual changes to the repository, or discard the changes, or delete the file. All of these changes are saved in the repository when we commit them. We can see the list of commits in the tree explorer.

Up to this point, what you have learned will benefit you greatly as a programmer, as you will be able to restore you file back to any previous time if you find that what you have done is a mistake. But the most valuable part of Git is its ability to share your code effectively with other people on your team. We will learn how to do that in this tutorial.

First, we remember that this Git repository is located in two places. It is located on your local machine, but it is also located on a remote server (in this case, our remote server is GitHub). The remote server is also known as the “origin”, meaning the origin from which we cloned and downloaded this repository. If two team members cloned the same repository, wouldn’t it be nice if team member A could copy over the commits that team member B made to their repository? This is possible by pulling and pushing your commits to the remote origin.

Let’s say that we have this case where team member A has committed 3 times, and team member B has not committed at all. In order for team member B to get the commits from team member A, team member A must do the following:

1. Right click your local branch on the left hand side of SourceTree (until you learn about branching later, just remember that the default branch for every repository is “master”)
2. Click on “Pull origin/master (tracked)”. This will pull from the origin the latest commits on the server. In this case, nothing is new on the server for team member B, but it is important to get in the habit of pulling before you push, because most of the time Git will require you to do so if you don’t have the latest stuff.
3. Right click you branch again, and click on “Push to origin/master (tracked)”. This may take some time, because Git is going to send all the commits that you have in your repository to the remote server. All the commits, that is, since you last pushed your branch to the server.

Now the remote server has the latest code from team member A. Team member B can get those latest changes by:

1. Right clicking the local branch “master” on the right hand side of SourceTree
2. Clicking on “Pull” option. This will download the latest commits and automatically commit them to this user’s local repository.

It is that simple! Now both team members are looking at the same code. To review what has just happened, I will demonstrate another example.

After getting the latest code, team member B makes a change to the code. To get this change down to team member A, team member B:

1. Pulls from the origin
2. Pushes to the origin
3. Then team member A pulls from the origin

The process of pulling, then pushing is the way that you will learn to live as you collaborate with other team members on the project. You will notice their commits show in your tree explorer, and you can see their username and message attached to each commit.

You don’t have to pull and push with every commit. You can push and pull multiple commits at a time. You should do this frequently in order to ensure that you are working off of the latest and greatest code. Once you have pushed or pulled a branch once, SourceTree will then keep track of the differences between your local repository and the remote repository, and will tell if you if there are updates waiting for you on the remote server, of if your local repository has updates that are not yet on the remote server. Remember that the remote server is the common access point between you and your team members.

## Reverting commits

Don’t like what your partner has committed and pushed? Or did you make a mistake yourself, and what to undo it? These tasks are made easy by Git.

We have already learned that if have not yet committed your change to the repository, you may easily discard changes by selecting the files in the “Unstaged files”, right clicking them and either selecting “Discard” or “Remove” (“Reset” on a Mac), depending on if the change was an edit or a new file.

But how do you undo a change that was committed to the repository?

1. Find the commit you wish to undo
2. Right click on the commit in the Tree Explorer
3. Select “Reverse commit…”
4. Confirm the action

That easy. You will notice that Git did something you might not have expected at first, but you will be glad that Git did it this way. Instead of simply “deleting” that commit forever from all of Git history, what Git actually did was create a new commit with all the actions that would undo your commit (ie add lines you deleted or delete lines you added). This way you can still properly see the full history of your file, and if you would like to restore those changes you just reversed that is possible. In Git it is a little bit tricky to actually delete something forever from memory (it is possible though), but what reversing a commit does is simply add a new commit to your repository to reverse your changes.

If you reverse a commit, don’t forget to push your changes to the origin so that your teammate can have those latest changes.

# Branching

## New branches

Now it’s time to branch out and learning something very useful – branches. Why is our Git tool called Source**Tree** anyway?

On very large projects, you will have many different people working on specific enhancements and bugs. Before they begin programming, they know what feature they are going to build or repair, and that feature will be a collection of commits in their repository over the next few days or weeks. To better keep track of a set of changes, as well as to provide a means of allowing what we will later learn called “Pull Requests”, we can use branches in our Git workflow.

We will demonstrate this with an example. Let’s say that my next task on my project is to add some style to our GUI. BEFORE I touch a single line of code, I first want to create a new branch.

To create a branch, I first select the branch that I want to base off of. Master, which we have been using throughout these tutorials, is nothing more than a branch. When I say “I want to base my new branch off of the master branch”, what I am saying is that I want Git to copy over all the code and every commit from the master branch, and create a new branch that has a copy of everything in the master branch. To create this new branch based off of master, you can:

1. Click on the branch button located at the top of SourceTree
2. Name your new branch. Naming conventions popularly use an Agile Card #, followed by a brief description of the enhancement you will be working on. If you have not yet taken a project management class, and do not know what Agile development is, simply enter in a brief description of your enhancement.
3. Select “Create Branch”

Let me point out a few things here.

1. If you look at the top left corner of SourceTree, you will see that we now have two branches – “master”, and the name of your new branch. You will also notice that your new branch is in bold, indicating that the current branch your repository is “on” is your new branch, and not the master branch. You will always know which branch you are on by looking at this section.
2. Your new branch has all the commits from “master”

Now I am going to make the changes to my code. I’m ready to commit these to Git. I commit them using the normal procedure, and I’m going to use the shortcut here to push these changes to the origin (this can also be accomplished by right clicking the branch, and selecting the push option).

I want to show you what has just happened. That commit that we just made is only in our new branch. To prove this, I am going to double click on the “master” branch. That is how you switch between branches in Git – double clicking on the branch you want to switch to. Note that if you try to switch branches but have uncommitted changes, Git may not let you switch branches until you have either discarded or committed those changes (circumstances vary).

Now I am in the master branch. You will notice all the commits that I have done, but you will notice that the latest commit is not in the master branch. That is because that commit was made in the new branch. As I switch back to the new branch, you see the new commit(s) again. I will continue to make commits to this new branch until I have finished fixing the bug, or adding the enhancement that I was coding. When I am done, I am going to want to merge the branch that I am on with the master branch, so that the master branch will have the latest integrated code. To do this, I follow these steps:

1. Switch to the main branch, into which I want to integrate other branches
2. Right click on the branch that you want to merge into the branch you are currently in
3. Click on “Merge \_\_\_\_\_\_ into current branch”

Notice how the branch’s commits are shown below. Each commit is transferred over to the main branch, as well as a new commit that shows that the merge has taken place.

## Merge Conflicts

Merge conflicts can happen frequently, depending on how your git flow is setup, and which files you are working on in correspondence with others.

When Git sees that you and another developer have changed the same line of code, and it doesn’t know which line to keep, it will warn you as you try to merge your branch into the main integration branch. An example looks like this:

I have been working on this branch, and I am now ready to merge it back into the master branch. I switch to the master branch, right click on the branch I want to merge, and click to merge that branch. But wait, Git informs me that there are merge conflicts that need to be addressed before the merger is complete.

Sourcetree shows me the items that have been merged, and the items that have merge conflicts are indicated by the warning sign. There are several options to resolve this merge conflict, and I will show you two of them.

The first way is to manually open the files, and edit the code.

1. Open up the files that have warning signals on them
2. You will notice that the “conflict areas” are indicated by Git messages
   1. The code between the lines

<<<<<<< HEAD

…Code…

=======

…Code…

>>>>>>> branch-name

Is the conflict area

* 1. The first half of the code (the code before the equal sign line) is the code that is in the current branch that you have checked out.
  2. The second half of the code (the code after the equal sign line) is the code that is in the branch that you are merging into your currently checked out branch.

The second way is to use the “use theirs” or “use mine” built in tools of Atlassian SourceTree. If you want to completely accept the code found in the new branch, or inversely if you want to only accept the code in the branch you have currently checked out, you can do so by:

1. Right clicking the file with the warning sign
2. Clicking on “Accept mine” will accept the code from the checked out branch, and discard the code from the branch trying to be merged in
3. Clicking on “Accept theirs” will accept the code from the branch trying to be merged in, and discard the code in your currently checked out branch

## Pull Requests

Typically a project will have a main integration branch. For our project, that is the master branch. This branch has the code that has been peer reviewed, QA’d and approved. Team leads will want tighter control over that branch. They may want to ensure that before you merge your feature branch into the main integration branch, he has a chance to review your feature branch. This is why the pull request function of Git was made.

To submit a pull request means that you are requesting permission to merge a branch into another branch. This could be the case when you want to merge your feature branch into the main integration branch. To request that your branch be merged into another branch, follow these steps:

1. Right click on the branch that you want to merge
2. Click on “Create pull request…”
3. Click on “Create Pull Request On Web”
4. The “base branch” will be the integration branch that you want to merge into. The “compare branch” will be the branch that you wish to merge.
5. Write a comment explaining your feature branch
6. Notice the commits that make your branch different than the integration branch (and these are the commits that will be merged in)
7. Submit the pull request by clicking on “Create pull request”

You are done! The owner of the repository, or your partner, can then approve the pull request by going to the link, previewing the changes, and clicking “Merge pull request”. Once that is done, your branch will then be merged into the integrated branch.

This process may seem somewhat redundant, but that is because the context of this class does not demand this sort of Git workflow happens. However, odds are your future employer will have a Git flow that requires pull requests, so we have taken the time to introduce you to the concept here.

# Increase understanding

## Searching history

Git provides you with much information about the history of your project, and the individual files in them. There are several ways to review this history, which may be helpful for you in determining what previous programmers were intending for the files in the project.

You can scroll through the tree visualization, and see the different commits that have taken place in that branch. You can see the changes to the files of each commit, the author of the change, and the comments that they provided in their commit message.

You can also search through the repository history in SourceTree. You can search by commit messages or authors.

You can see the history of a single file throughout its history (since it was added to the Git repository), by right clicking the file in the preview area, and selecting “Log Selected…”. This will show you the different commits that this file has been affected by, and has been a very useful tool for me in my career so far. Before I make a change to a file on the big project, I first lookup the history of the file, and determine what previous programmers were working on. This can be a useful tool for tracking down the introduction of bugs and other coding issues that may be present.

## Stashes

Stashes are good for local development. Here is the situation in which you use stashes the most:

Let’s say that you have checked out the feature branch that you are working on. Then, you want to either switch or work on a different branch. But, you have a list of changes that you have not committed yet, but want to delay committing for whatever reason (for instance, you are running into issues with your new code and don’t want to commit bad code). Most of the time, Git will not let you switch branches if you have uncommitted changes (this is due to possible merge conflicts – a warning message will prevent you from switching the branch).

You can stash the uncommitted changes, which means save a temporary snapshot of the uncommitted changes, then discard them so that you can change branches. Now you may switch branches, because your uncommitted branches have been stashed away, and are not currently present.

The whole purpose of creating stashes is so that you can restore them at a future date. You can restore a stash by:

1. Right clicking on the stash you wish to restore (you can preview that stash by left clicking on it)
2. Click on “Apply Stash”

What has just happened is that your uncommitted changes have been restored, and you once again have uncommitted changes. It is important to note that when you restore a stash, you are not making a permanent commit. Stashes involve no commits, so they are not permanent. In order to commit the changes into the repository, you must follow the typical commit process after restoring the stash. You may then delete the stash if you wish.

In summary, stashes are for saving, temporarily, uncommitted changes to your local machine. Stashes are not committed to the repository, nor available for others.

## Remotes

Remember that your Git repository is stored in two places – locally on your machine, and on a remote server. The remote server we have been using for these tutorials is GitHub. The remote server is the common access point for you and all of your teammates. Commits need to be pushed and pulled from the remote server, in order for collaboration of code to happen.

In SourceTree it is easy to tell what branches exist locally, and which branches exist on the remote server. Odds are there will be more remotes on the server than there will be on your local machine, because you do not want to download other team members branches to your local machine most of the time. Typically, the integration branch of your team will contain the merged branches of your teammates, after they have submitted a pull request and it has been accepted and merged.

If there is a branch on the remote server that you want on your local machine, you can double click on the remote branch, and SourceTree will proceed to download this branch to your local machine, and switch to that branch (called “Checking out a branch”). You can also delete remote branches, but be careful when doing this.

Git keeps track of the differences between your local branch and the remote branch, and will let you know if your local machine is ahead or behind the remote server.

If, for whatever reason, the remote server moves, you can tell SourceTree where the new origin of your remote server is.

You can view all your branches, the code, the commits, history, and everything else that SourceTree can show you online. If you go to your repository URL, you can navigate your project easily. This could be helpful if you wish to email links to your code for sharing on the web.

## Quick Tips

Sometimes it can be confusing to see other people’s branches and your branches changes at the same time. To avoid seeing the whole project, and focus on your branch, select “Current Branch” here, instead of “All Branches”.

You can push branches using the button on the top, by right clicking on the branch, or by selecting the checkbox here in the commit preview.

You can pull by selecting the button on the top, and by richt clicking on the branch.

Make sure to always select “Track changes”.

SourceTree allows you to see the complete file the way it existed at a specific commit. To see this, instead of just the small difference shown in the tree explorer, right click on the file and select “Open Selected Version”.

You can quickly open the file from SourceTree by either clicking on “Open” or “Show in Explorer”.

Sometimes I mess up a branch by accidentally committing things I didn’t mean to, messing up, or other weird git problems. One way to resolve this is to make sure that you DO NOT push your changes to the remote server (origin). If you do, it will be a little bit more difficult, but still possible. If you have not pushed your changes to the remote server, I might suggest switching branches, deleting the branch you want to (might have to force delete it), and then either rechecking out the branch from the origin, or branching off a new branch.

There are triggers, known as “hooks” in Git terminology, and they can be useful for triggering actions when you commit to your repository. That will not be covered in these tutorials.

One thing that is vital to realize is that what is showing up in your SourceTree explorer is reflected in the real files on your computer. If you switch branches, the literal files on your computer in that directory are also changed. This may take some experience and examples to see what I am talking about here.

Git is available through the command prompt. Feel free to use command prompt as well as SourceTree. Command line commands will not be covered in these tutorials.

You can change your email and username in the settings pane. You can also edit your photo online.

You do not have to have a remote server for your repository. It is possible, if you wish, to simply have a local copy of your repository. You will not be able to push or pull commits, but this may be a solution if you wish to track changes to some files by way of Git.

The easiest view in the tree explorer is the “Log/History” view. I keep it on this view most of the time.