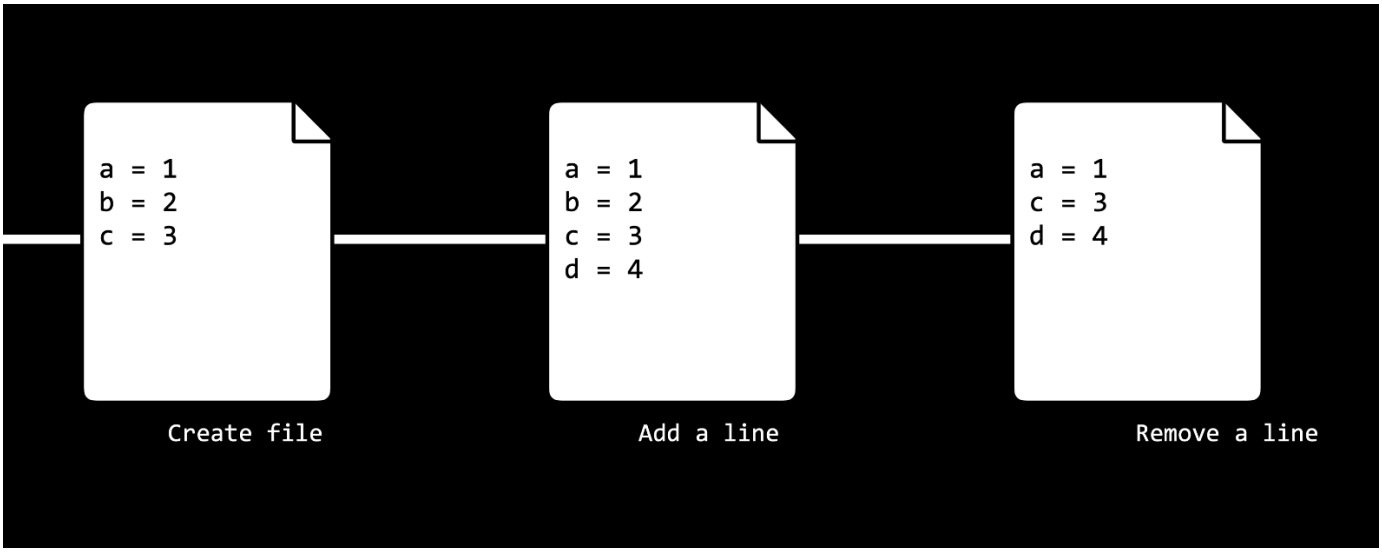
Introduction to Git and GitHub

Introduction

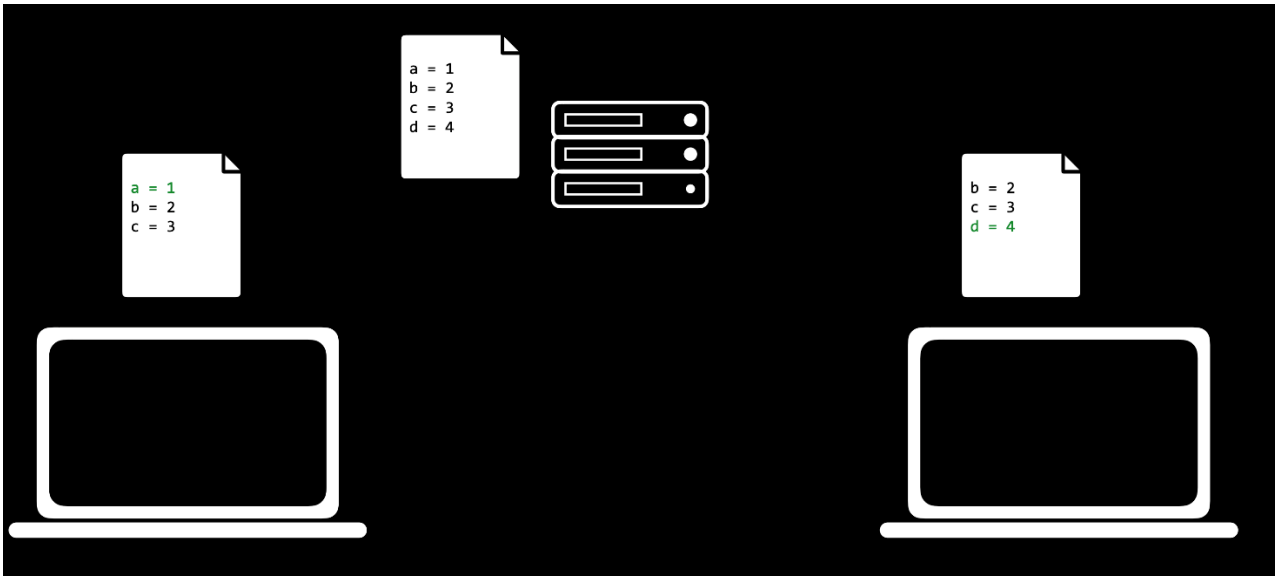
This tutorial is about Git and GitHub. It focuses only on the basics of using Git and GitHub and the goal is to help you start using these very popular tools used in developing applications. Therefore, if you want to be proficient in Gits and GitHub you may Google and learn on yourself from the abundant resources available on the Internet.

Git

- Git is a command line tool that will help us with version control in several different ways:
    - Allowing us to keep track of changes we make to our code by saving snapshots of our code at a given point in time.



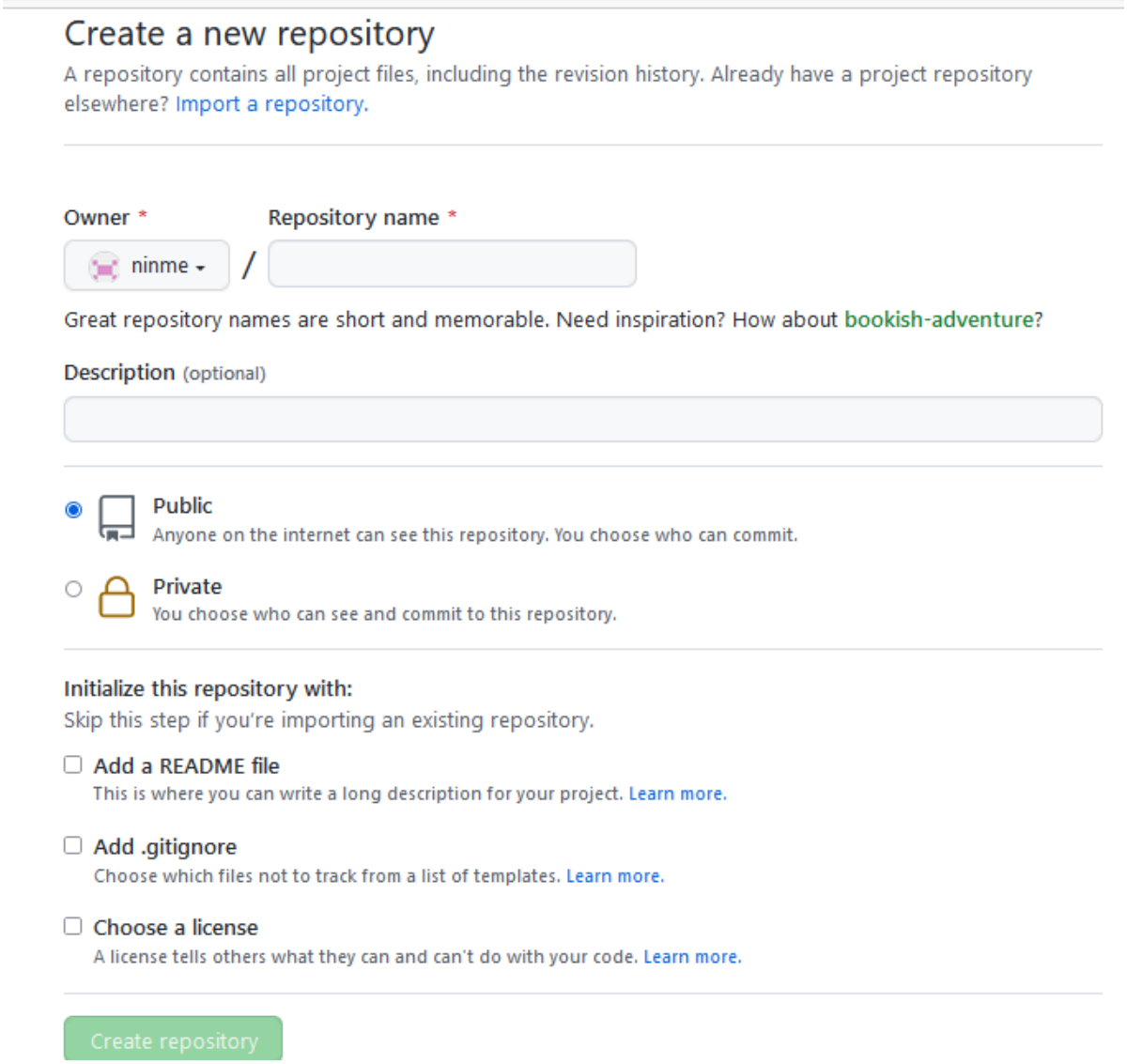
    - Allowing us to easily synchronize code between different people working on the same project by allowing multiple people to pull information from and push information to a repository stored on the web.



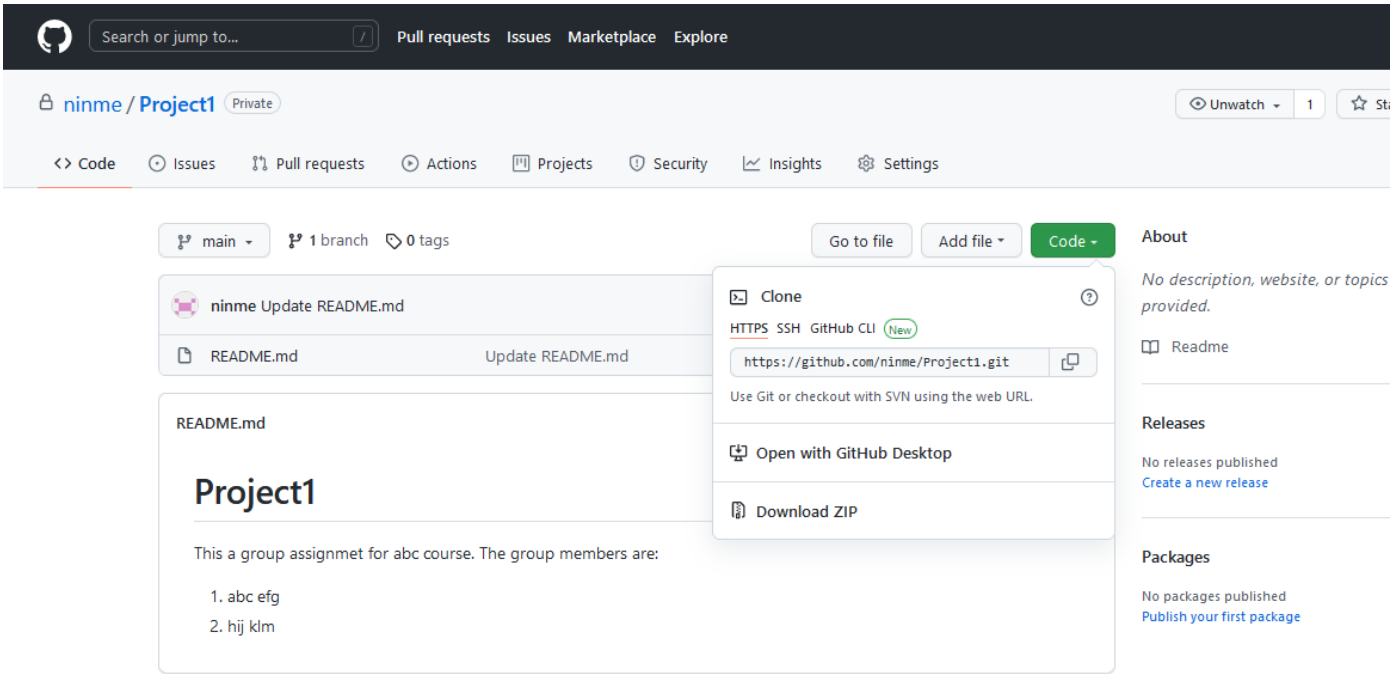
    - Allowing us to make changes to and test out code on a different *branch* without altering our main code base, and then merging the two together.
    - Allowing us to revert back to earlier versions of our code if we realize we've made a mistake.
- In the above explanations, we used the word **repository**, which we haven't explained yet. A Git repository is a file location where we'll store all of the files related to a given project. These can either be remote (stored online) or local (stored on your computer).

GitHub

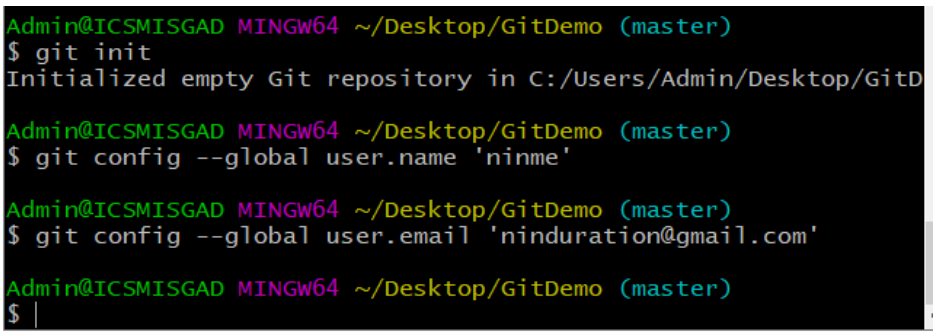
- GitHub is a website that allows us to store Git repositories remotely on the web.
- Let's get started by creating a new repository online
    1. Make sure that you have a GitHub account set up. If you don't have one yet, you can make one here.
    2. Click the + in the top-right corner, and then click "New repository"
    3. Create a repository name that describes your project
    4. (Optional) Provide a description for your repository
    5. Choose whether the repository should be public (visible to anyone on the web) or private (visible just to you and others you specifically grant access). ***You should choose private option for graded course assignments***
    6. (Optional) Decide whether you want to add a README, which is a file describing your new repository.



- Once we have a repository, we'll probably want to add some files to it. In order to do this, we'll take our newly created *remote* repository and create a copy, or clone, of it as a *local* repository on our computer.
    1. Make sure you have git installed on your computer by typing `git` into your terminal. If it is not installed, you can download it here.
    2. Click the green "Clone or Download" button on your repository's page, and copy the url that pops down. If you didn't create a README, this link will appear near the top of the page in the "Quick Setup" section.



3. Open GitBash and got to yourworking directory(local repo)
4. In your terminal, run **`git init`** to initialize your local repo
5. Then set-up the username and email of your git-hub account using the following commands



6. In your terminal, run `git clone <repository url>`. This will download the repository to your computer. If you didn't create a README, you will get the warning: `You appear to have cloned into an empty repository.` This is normal, and there's no need to worry about it.

Note: if the repo is a private repo (which is mostly the case if the repo is used as platform to submit assignment/homework solutions or something that you don't want to share with public) Instead of *`git clone https://github.com/NAME/repo.git`*

use *git clone https://personaltoken@github.com/NAME/repo.git* . To see/learn how git tokens are generated please click here.

```
Admin@ICSMISGAD MINGW64 ~/Desktop/GitDemo (master)
$ git clone https://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@github
.com/ninme/Project1.git
Cloning into 'Project1'...
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (6/6), done.

Admin@ICSMISGAD MINGW64 ~/Desktop/GitDemo (master)
$
```

7. Run `ls`, which is a command that lists all files and folders in your current directory. You should see the name of the repository you've just cloned.
8. Run `cd <repository name>` to change directory into that folder.
9. Run `touch <new file name>` to create a new file in that folder. You can now make edits to that file. Alternatively, you can open the folder in your text editor and manually add new files.
10. Now, to let Git know that it should be keeping track of the new file you've made, Run `git add <new file name>` to track that specific file, or `git add .` to track all files within that directory.

```
Admin@ICSMISGAD MINGW64 ~/Desktop/GitDemo/Project1 (main)
$ touch problem1.cpp

Admin@ICSMISGAD MINGW64 ~/Desktop/GitDemo/Project1 (main)
$ git add problem1.cpp

Admin@ICSMISGAD MINGW64 ~/Desktop/GitDemo/Project1 (main)
$
```

Commits

- Now, we'll start to get into what Git can be really useful for. After making some changes to a file, we can *commit* those changes, taking a snapshot of the current state of our code. To do this, we run: `git commit -m "some message"` where the message describes the changes you just made.
- After this change, we can run `git status` to see how our code compares to the code on the remote repository
- When we're ready to publish our local commits to Github, we can run `git push`. Now, when we go to GitHub in our web browser, our changes will be reflected.

- If you've only changed existing files and not created new ones, instead of using `git add .` and then `git commit...`, we can condense this into one command: `git commit -am "some message"`. This command will commit all the changes that you made.
- Sometimes, the remote repository on GitHub will be more up to date than the local version. In this case, you want to first commit any changes, and then run `git pull` to pull any remote changes to your repository.

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

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   problem1.cpp


Admin@ICSMISGAD MINGW64 ~/Desktop/GitDemo/Project1 (main)
$ git commit -m 'Solution to problem1 is added by abc'
[main 8693304] Solution to problem1 is added by abc
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 problem1.cpp

Admin@ICSMISGAD MINGW64 ~/Desktop/GitDemo/Project1 (main)
$ git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 295 bytes | 295.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/ninme/Project1.git
   a574d17..8693304  main -> main
```

Merge Conflicts

- One problem that can emerge when working with Git, especially when you're collaborating with other people, is something called a **merge conflict**. A merge conflict occurs when two people attempt to change a file in ways that conflict with each other.
- This will typically occur when you either `git push` or `git pull`. When this happens, Git will automatically change the file into a format that clearly outlines what the conflict is. Here's an example where the same line was added in two different ways:

```
a = 1
<<<<< HEAD
b = 2
=====
b = 3
>>>>> 56782736387980937883
c = 3
d = 4
e = 5
```

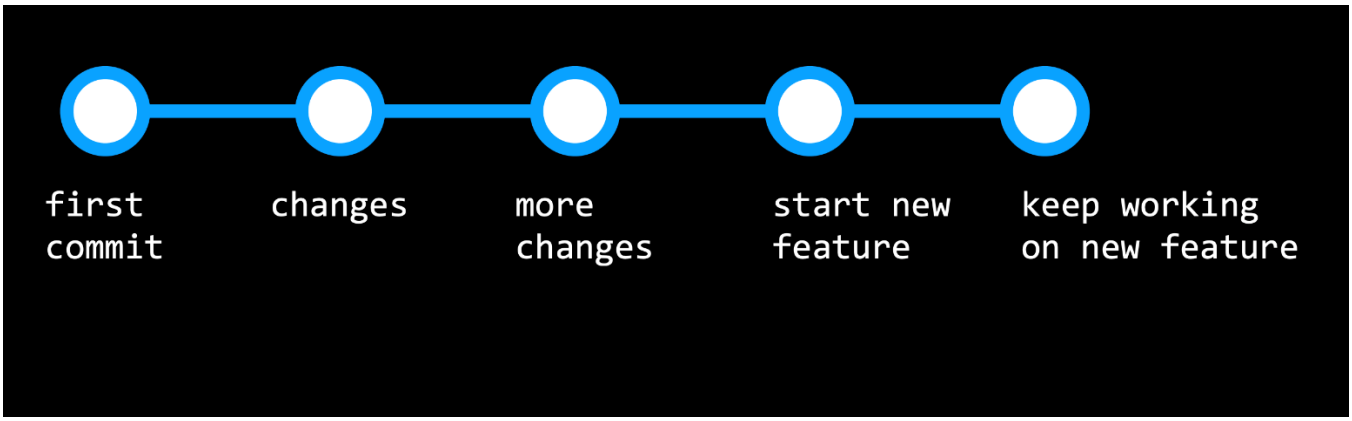
- In the above example, you added the line `b = 2` and another person wrote `b = 3`, and now we must choose one of those to keep. The long number is a *hash* that represents the commit that is conflicting with your edits. Many text editors will also provide highlighting and simple options such as “accept current” or “accept incoming” that save you the time of deleting the added lines above.
- Another potentially useful git command is `git log`, which gives you a history of all of your commits on that repository.

```
Admin@ICSMISGAD MINGW64 ~/Desktop/GitDemo/Project1 (main)
$ git log
commit 869330424f237766808d5c78896b4c36da82bff4 (HEAD -> main, origin/main, origin/HEAD)
Author: ninme <ninduration@gmail.com>
Date:   Fri Nov 26 16:38:02 2021 +0300

    Solution to problem1 is added by abc

commit a574d17ba573894c357685028b211643f3512956
Author: ninme <95081159+ninme@users.noreply.github.com>
Date:   Fri Nov 26 15:20:30 2021 +0300

    Update README.md

commit 3d44e18f007e130652ae01b057febbe79216b5f0
Author: ninme <95081159+ninme@users.noreply.github.com>
Date:   Fri Nov 26 15:16:46 2021 +0300

    Initial commit
```

- Potentially even more helpful, if you realize that you’ve made a mistake, you can revert back to a previous commit using the command `git reset` in one of two ways:
  - `git reset --hard <commit>` reverts your code to exactly how it was after the specified commit. To specify the commit, use the commit hash associated with a commit which can be found using `git log` as shown above.
  - `git reset --hard origin/master` reverts your code to the version currently stored online on Github.

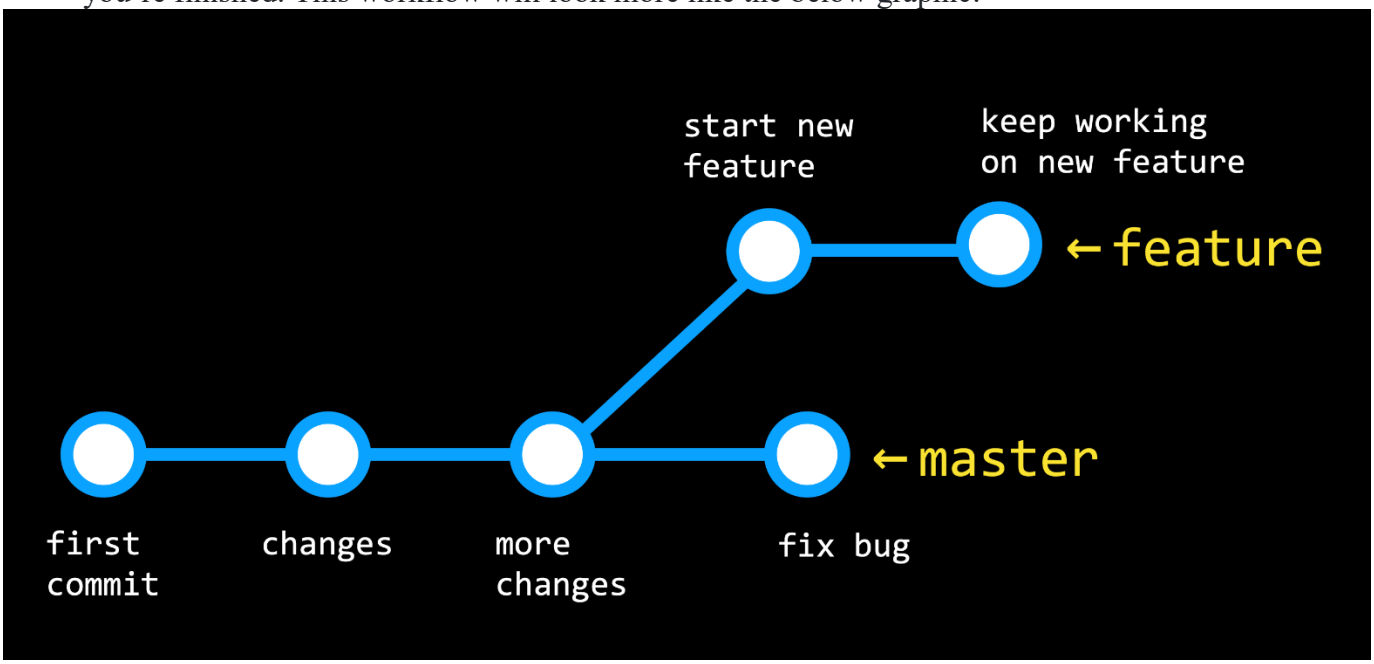
Branching

After you’ve been working on a project for some time, you may decide that you want to add an additional feature. At the moment, we may just commit changes to this new feature as shown in the graphic below



But this could become problematic if we then discover a bug in our original code, and want to revert back without changing the new feature. This is where branching can become really useful.

- Branching is a method of moving into a new direction when creating a new feature, and only combining this new feature with the main part of your code, or the main branch, once you're finished. This workflow will look more like the below graphic:



- The branch you are currently looking at is determined by the `HEAD`, which points to one of the two branches. By default, the HEAD is pointed at the master branch, but we can check out other branches as well.
- Now, let's get into how we actually implement branching in our git repositories:
    1. Run `git branch` to see which branch you're currently working on, which will have an asterisk to the left of its name.

```
Admin@ICSMISGAD MINGW64 ~/Desktop/GitDemo/Project1 (main)
$ git branch
* main

Admin@ICSMISGAD MINGW64 ~/Desktop/GitDemo/Project1 (main)
$
```

2. To make a new branch, we'll run `git checkout -b <new branch name>`.

```
Admin@ICSMISGAD MINGW64 ~/Desktop/GitDemo/Project1 (main)
$ git checkout -b newFeatures
Switched to a new branch 'newFeatures'

Admin@ICSMISGAD MINGW64 ~/Desktop/GitDemo/Project1 (newFeatures)
$ git branch
  main
* newFeatures

Admin@ICSMISGAD MINGW64 ~/Desktop/GitDemo/Project1 (newFeatures)
$
```

3. Switch between branches using the command `git checkout <branch name>` and commit any changes to each branch.
4. When we're ready to merge our two branches together, we'll check out the branch we wish to keep (almost always the main branch) and then run the command `git merge <other branch name>`. This will be treated similarly to a push or pull, and merge conflicts may appear.

Delete Files

The easiest way to delete a file in your Git repository is to execute the "git rm" command and to specify the file to be deleted.

```
$ git rm <file>

$ git commit -m "Deleted the file from the git repository"

$ git push
```

Note that by using the "**git rm**" command, the file will also be deleted from the filesystem. Also, you will have to commit your changes, "git rm" does not remove the file from the Git index unless you commit it and you have to push it to your remote repository as well.

In order to delete files recursively on Git, you have to use the "git rm" command with the "-r" option for recursive and specify the list of files to be deleted.

```
$ git rm -r <folder>

$ git commit -m "Deleted the folder from the repository"
```

```
$ git push
```

More GitHub Features

There are some useful features specific to GitHub that can help when you're working on a project:

- **Forking**: As a GitHub user, you have the ability to *fork* any repository that you have access to, which creates a copy of the repository that you are the owner of. We do this by clicking the "Fork" button in the top-right.
- **Pull Requests**: Once you've forked a repository and made some changes to your version, you may want to request that those changes be added to the main version of the repository. For example, if you wanted to add a new feature to Bootstrap, you could fork the repository, make some changes, and then submit a pull request. This pull request could then be evaluated and possibly accepted by the people who run the Bootsrap repository. This process of people making a few edits and then requesting that they be merged into a main repository is vital for what is known as *open source software*, or software that created by contributions from a number of developers.
- **GitHub Pages**: GitHub Pages is a simple way to publish a static site to the web. (You'll learn later about static vs dynamic sites.) In order to do this:
    1. Create a new GitHub repository.
    2. Clone the repository and make changes locally, making sure to include an `index.html` file which will be the landing page for your website.
    3. Push those changes to GitHub.
    4. Navigate to the Settings page of your repository, scroll down to GitHub Pages, and choose the master branch in the dropdown menu.
    5. Scroll back down to the GitHub Pages part of the settings page, and after a few minutes, you should see a notification that "Your site is published at: …" including a URL where you can find your site!

That's all for now! However, you can learn and experiment more with Git and GitHub on your own.