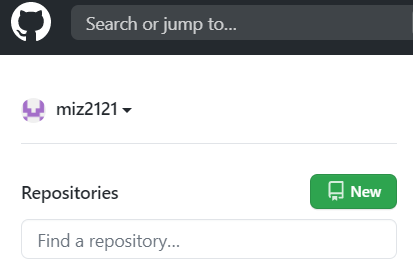
1. GitBash 설치

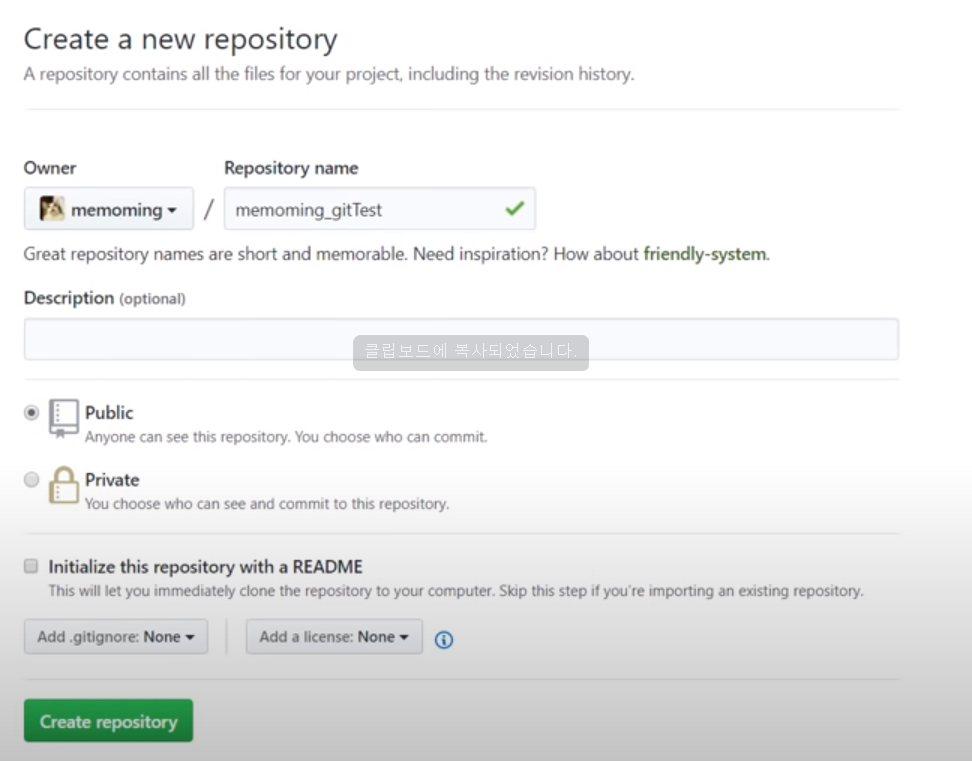
<https://gitforwindows.org/>

(for Window)

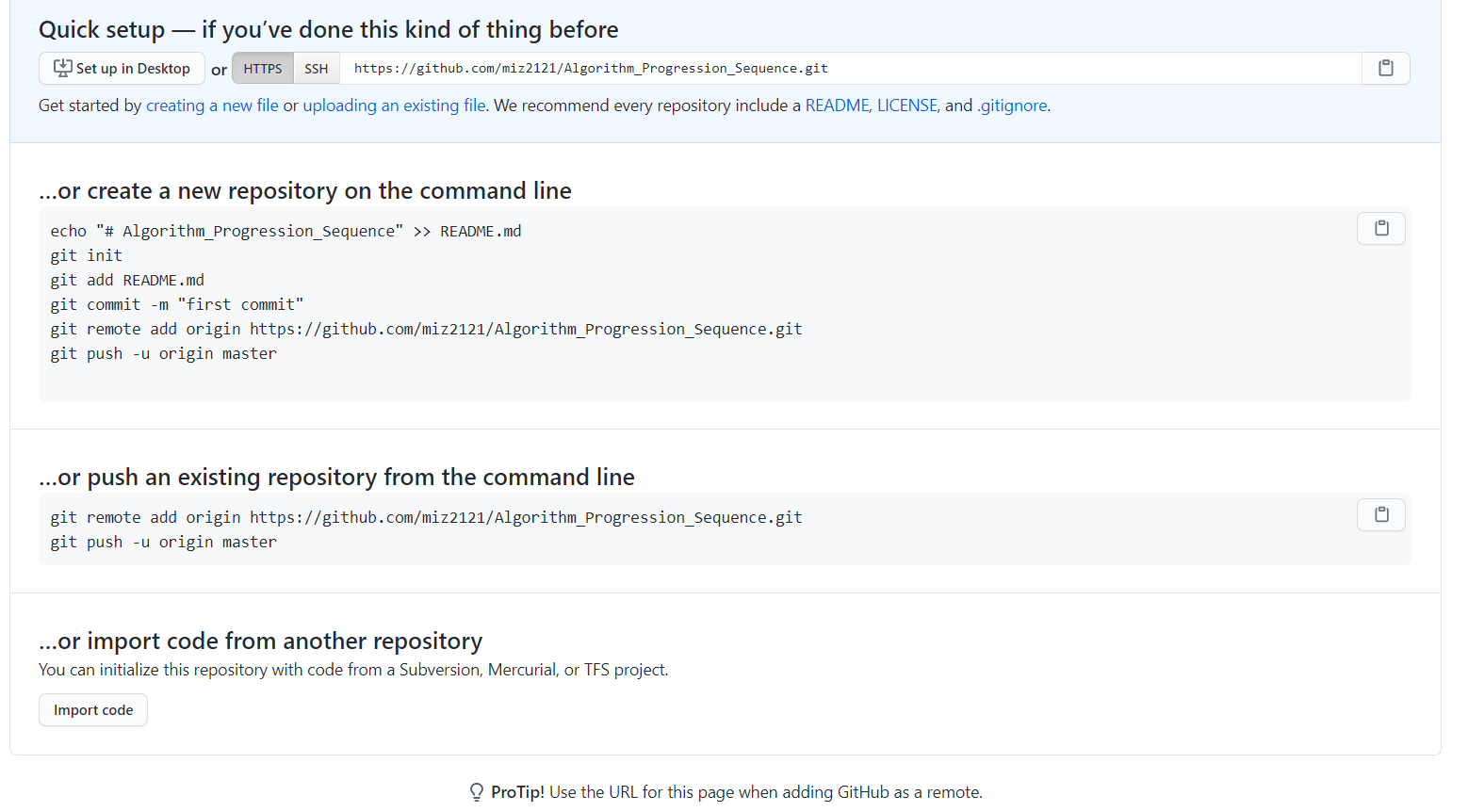
1. New Repository 생성

<https://github.com/>





Repository 이름 지어준 후 그대로 Create repository 해보자.



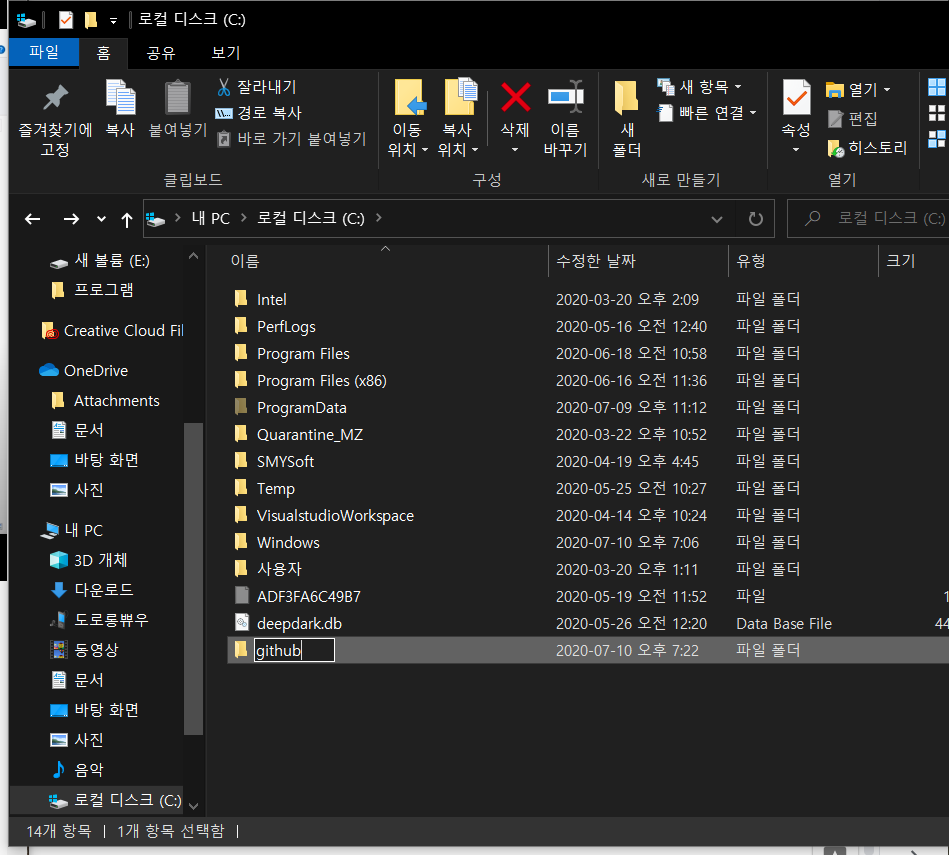
이런식의 창이 뜬다.

1. Root Folder 연결

기본폴더 → Web 상의 Git과 연결될 Local Folder

루트 폴더란, 웹 상의 git repository와 함께 연결 될 로컬 폴더를 말한다.

원하는 위치에 새 폴더를 생성한다.



* Command

git init

git remote add origin repository address

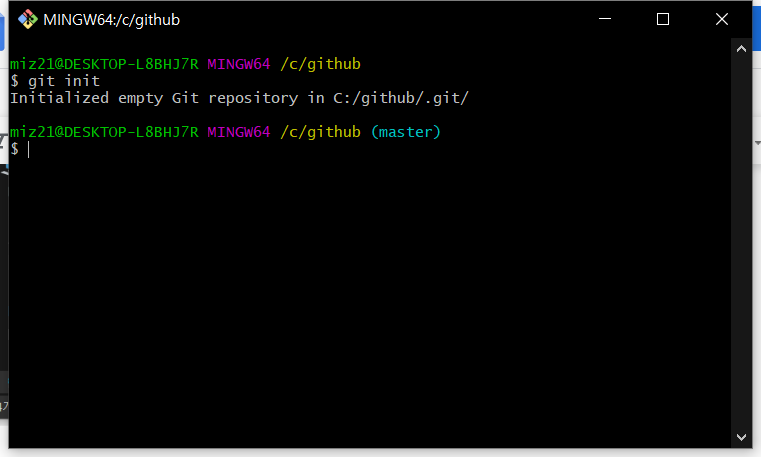
cf) git remote remove origin→ origin이라는 원격 저장소를 삭제한다.

→ git remote add origin repository address는 repository의 주소를 알려주기 위한 명령어다.

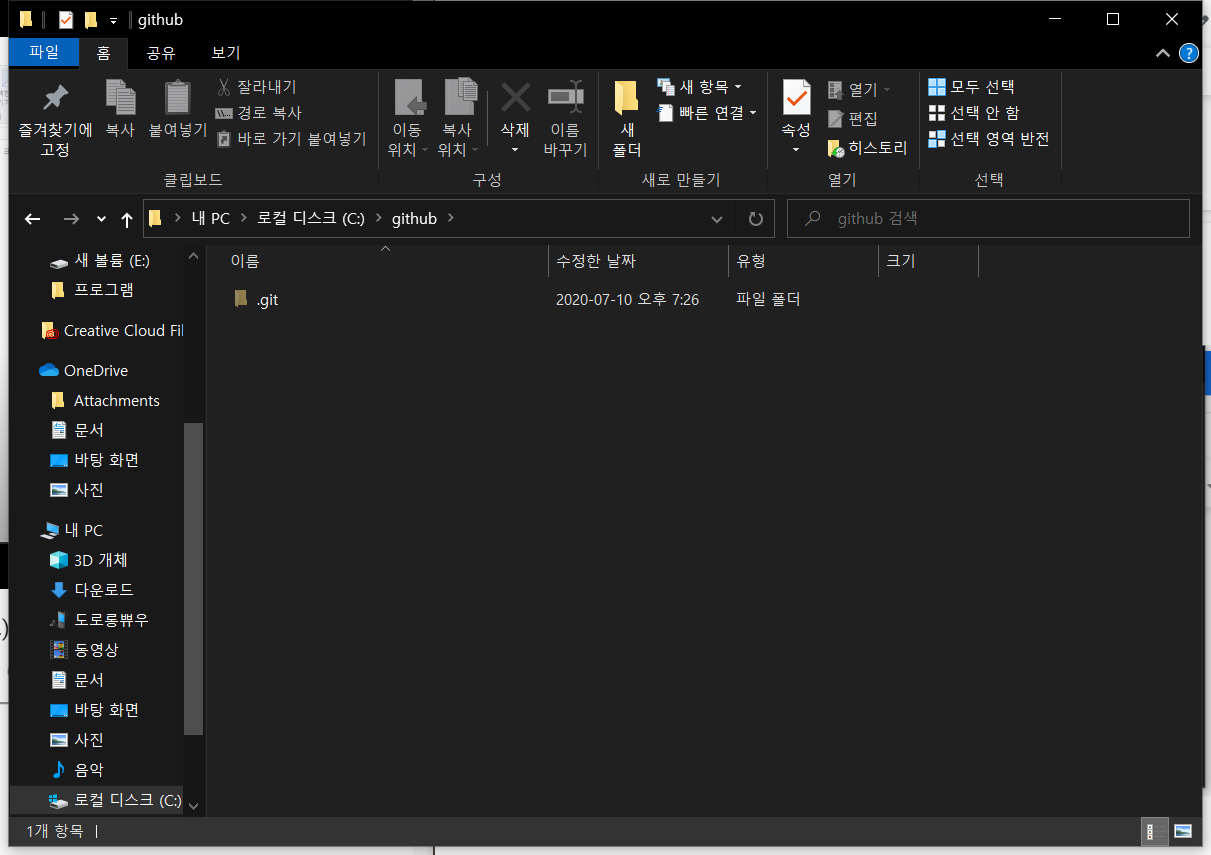
→ origin이라는 이름으로 원격저장소 주소를 등록한다.

방금 만든 폴더를 우클릭 한 후

Git Bash Here버튼을 클릭한다.

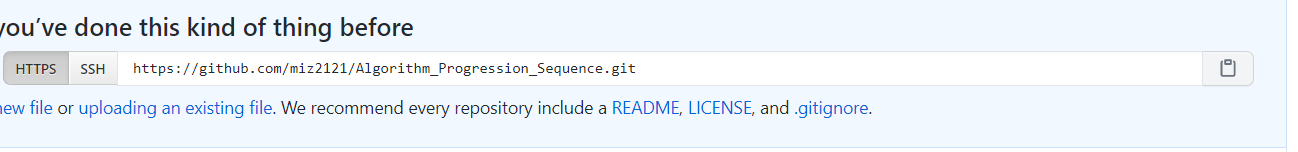


git init명령어를 실행하면

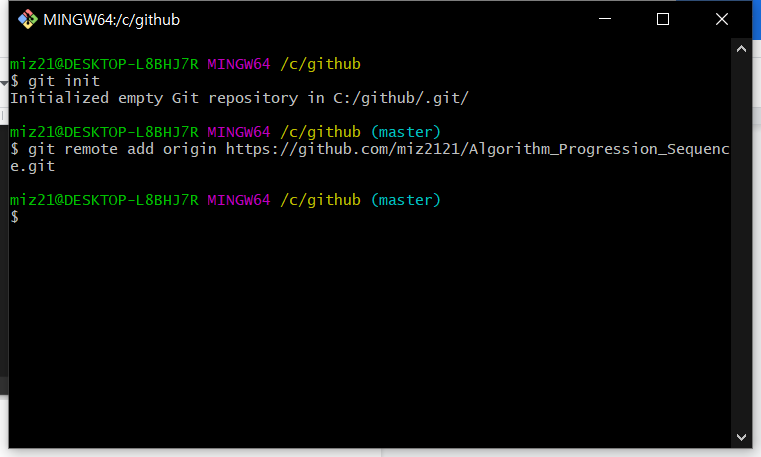


.git폴더가 생성된다.

깃허브에서



깃 주소를 복사한 후

git remote add origin 한 칸 띄고 뒤에 주소를 적는데, 우클릭 후 paste하면 된다.

아무런 말이 뜨지 않으면 정상이다.

|  |
| --- |
| cf) 아래 이미지 주소창의 주소를 paste해서 적든, 아래의 .git 주소를 paste해서 적든  상관없다.  git remote add origin https://github.com/miz2121/Github\_practice  혹은  git remote add origin https://github.com/miz2121/Github\_practice.git  상관 없다. |

1. Push 전엔 Pull이 있다.

→ 무조건은 아니지만 일단 이렇게 알아두자. 이렇게 알아둬야 날려먹는 일이 없을것이다.

Push→ git repository로 자료를 올리는 것

Pull→ 그 반대로, 당겨오는 것이다.

* Command

git pull origin master

→ 지정 해 놓은 Repository에서 해당 폴더로 repository에 있는 모든 자료를 끌어온다.

→ 정확히는, origin의 내용이 현재 사용하고 있는 컴퓨터의 master브랜치로 복사된다는 뜻이다.

git status

→ 현재 나의 로컬 폴더와 깃 과의 싱크 상태를 체크해 준다.

git add .

→ 한 칸 띄고 .까지가 모두 명령어다. 로컬폴더와 깃 레포지토리 사이의 변경된 내용을 모두 Tracking해주는 명령어다.

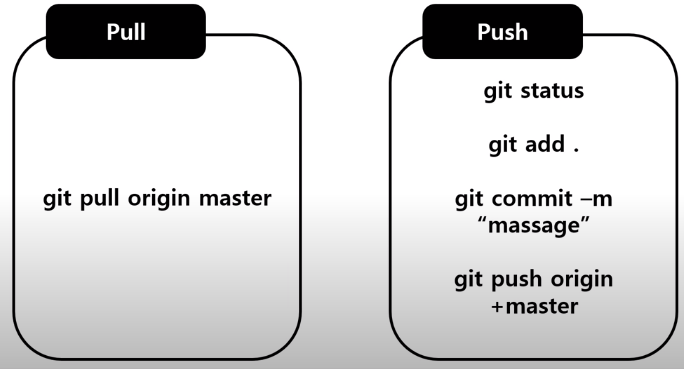
git commit -m "message"

→ 인식할 수 있는 commit을 달아준다.

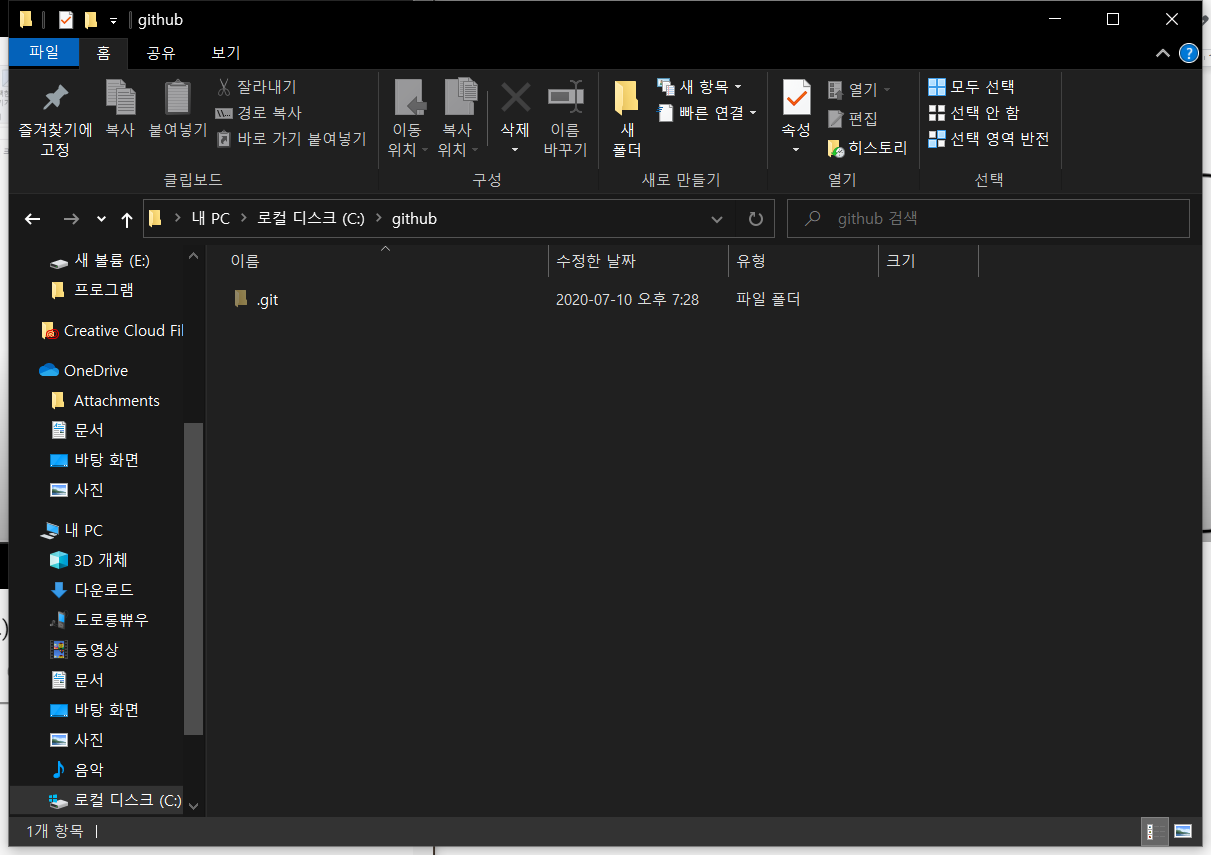
git push origin +master

→ 실제로 git에 PUSH

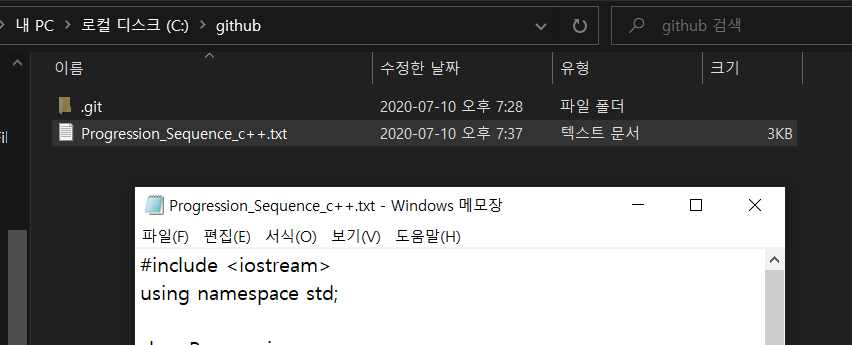
→ origin은 원격 저장소 이름이고, master는 현재 사용하고 있는 컴퓨터의 브랜치 이름이다.



cf) git push origin 하기 전에 git commit -m "message"를 해 주자. 순서가 있나보다.



앞으로 이 폴더가 repository와 정확하게 일치 될 폴더이다.

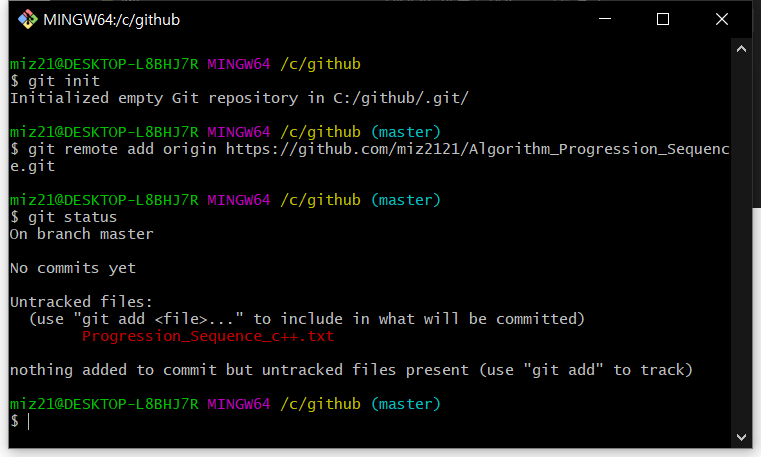


파일과 내용을 작성한다.

처음 연습해 보는거라면 파일명과 파일 내용은 아무렇게나 적어도 무방하다.

cf) 만약 Git Bash를 닫았다면, 다시 폴더를 우클릭 한 후, Git Bash Here를 클릭한다

cf) 아무런 push를 하지 않은 상태이기 때문에, pull을 하지 않고 일단 진행한다.



위 이미지를 읽어보자.

git status

→ 현재 나의 로컬 폴더와 깃 과의 싱크 상태를 체크해 주는 명령어다.

위 이미지를 읽어보면

Untracked files:가 있다.

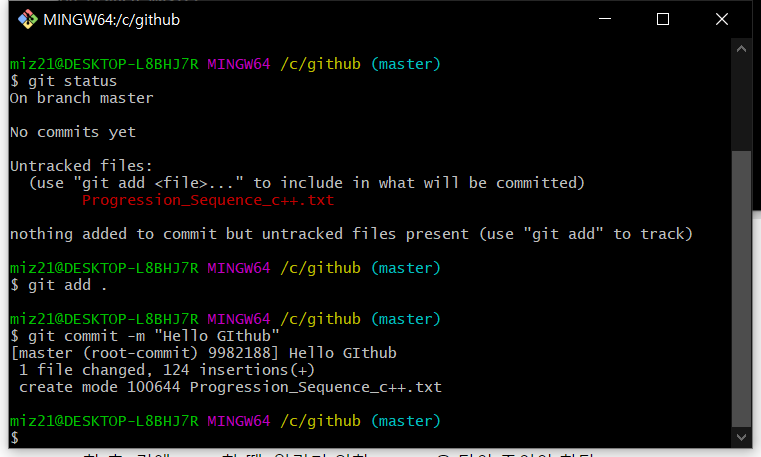
트래킹 돼 있지 않은 파일이 있다.

이 파일을 트래킹 해 주기 위해 필요한 명령어는 git add .이다.

. (온 점)명령어는 (지금은 아직 하나의 파일이지만) 여러 파일들이 있을 경우, 모두 트래킹 하겠다는 명령어이다.

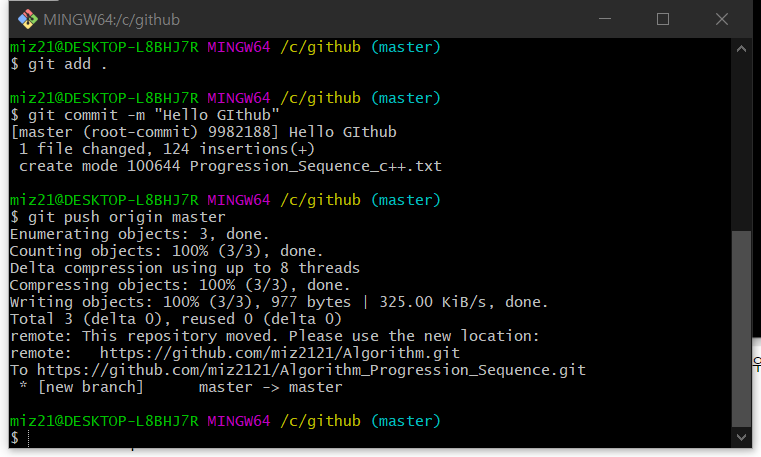
add 한 후, 깃에 push 할 때, 알리기 위한 commit을 달아 주어야 한다.

git commit -m "Hello Github"

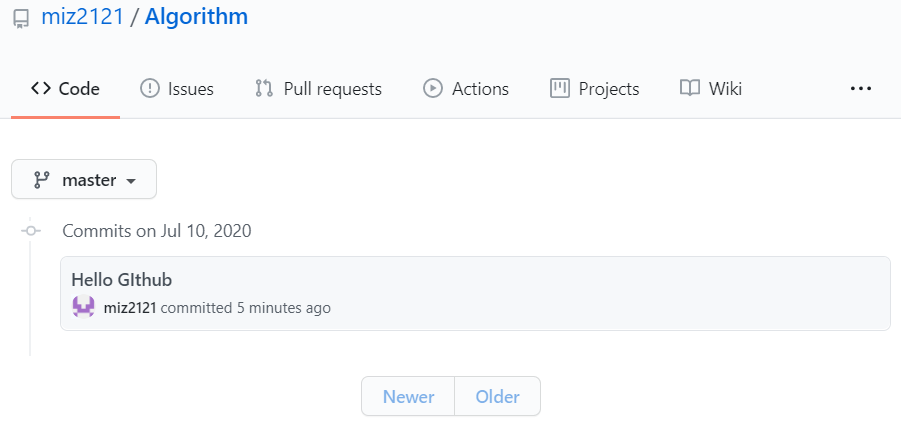


내용을 읽어보면, 1개의 파일이 바뀌었고, 124개의 추가된 내용을 확인할 수 있다. 줄 단위의 갯수이다.

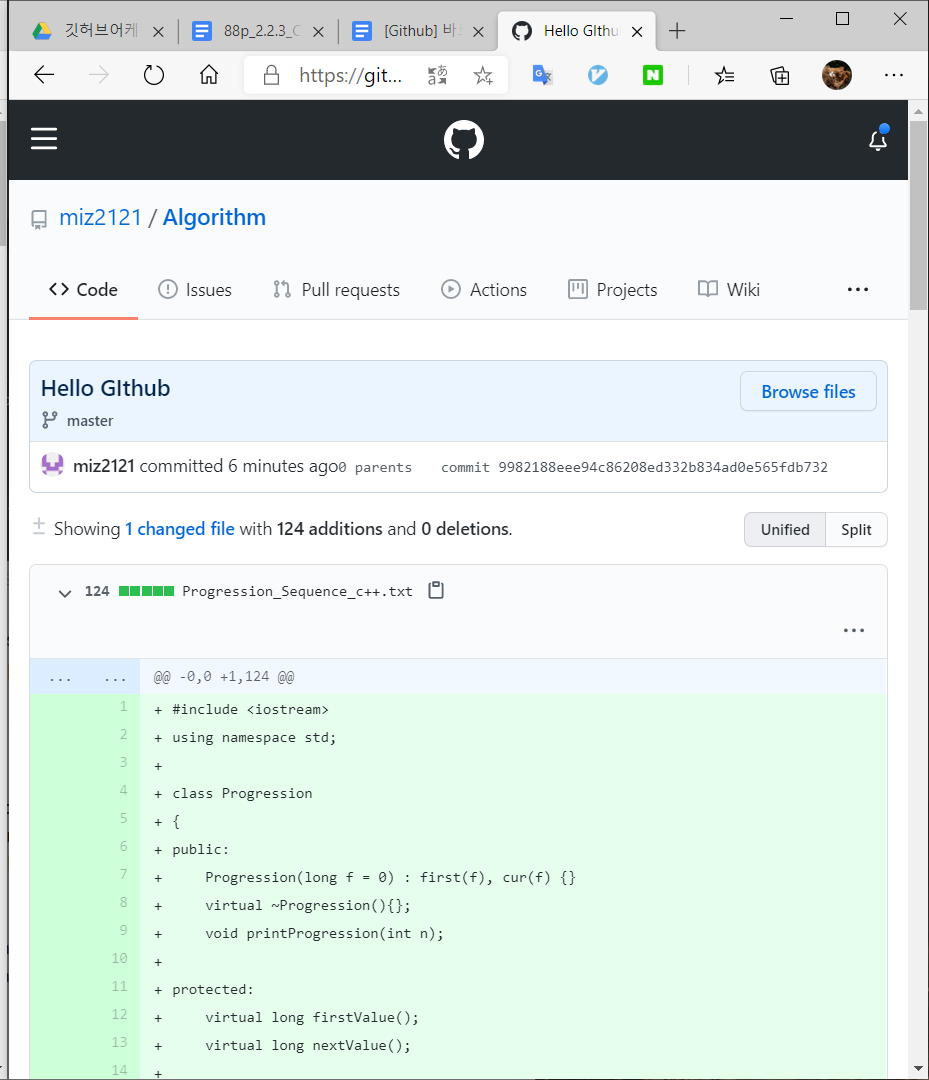
이제 깃에 push해야 내용이 올라간다.



|  |
| --- |
| cf) 혹시 아래처럼 로그인하라고 하면 로그인하자    아이디를 입력하면    이렇게 비밀번호 입력창이 뜬다고 한다. |



깃허브를 확인 해 보면 Hello Github란 commit과 함께 업로드 된 걸 확인 할 수 있다.



클릭했더니 내용이 보인다.

1. Mission:: 다른 사람 git 가져오기

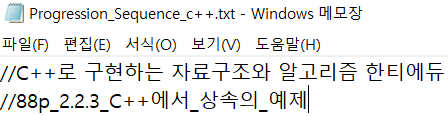
<https://github.com/memoming/memoming_gitTest>

위 주소(유뷰트 메모밍 채널의 강의 깃허브 주소입니다)에서 내용을 가져오자.

풀이!!

새 폴더 만든 후, git init, git remote add origin 위 주소, git pull origin master를 한다.

만약! 추가된 내용이 있다면!



주석 두 문장을 위 처럼 추가했다!

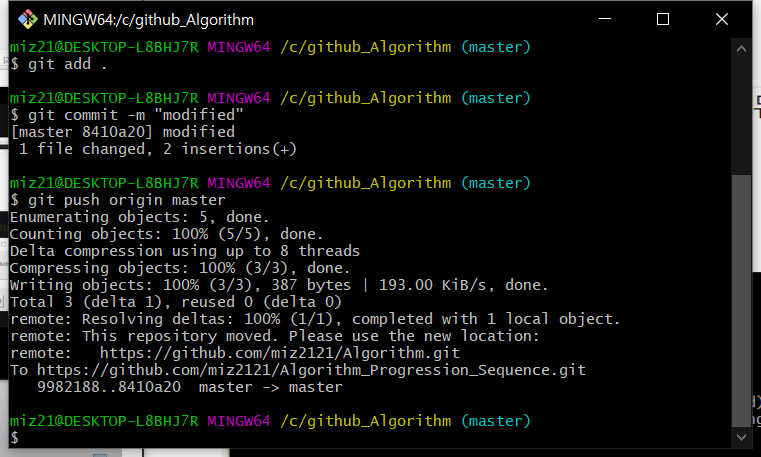
다시 폴더를 우클릭 한 후 Git Bash Here를 누른다.

일단 git status로 체크 해 보자.



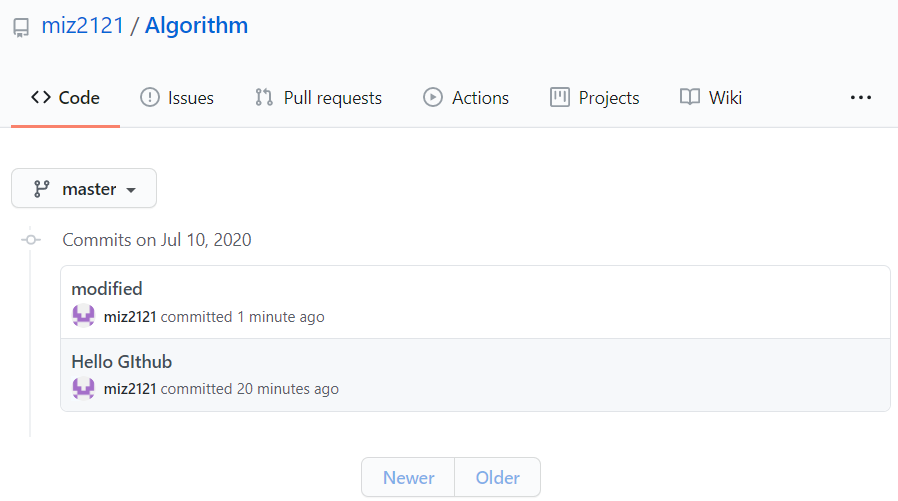
수정된 파일이 존재하고 아직 git에 올라갔지 않았다고 뜬다.

git add .을 해 주자.

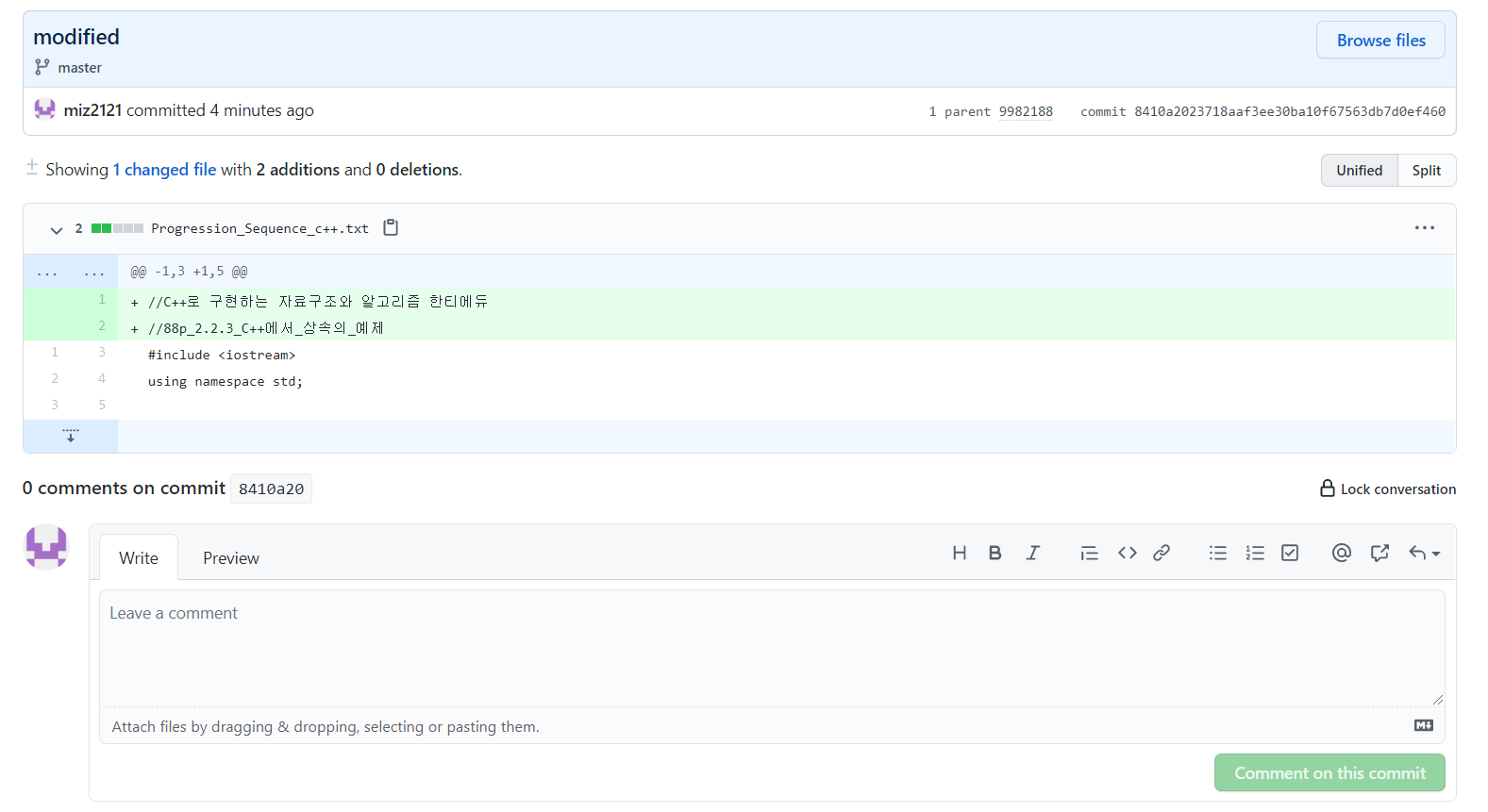


위 이미지의 명령어는 잘 읽어보도록 하자.

위 과정이 완료되면



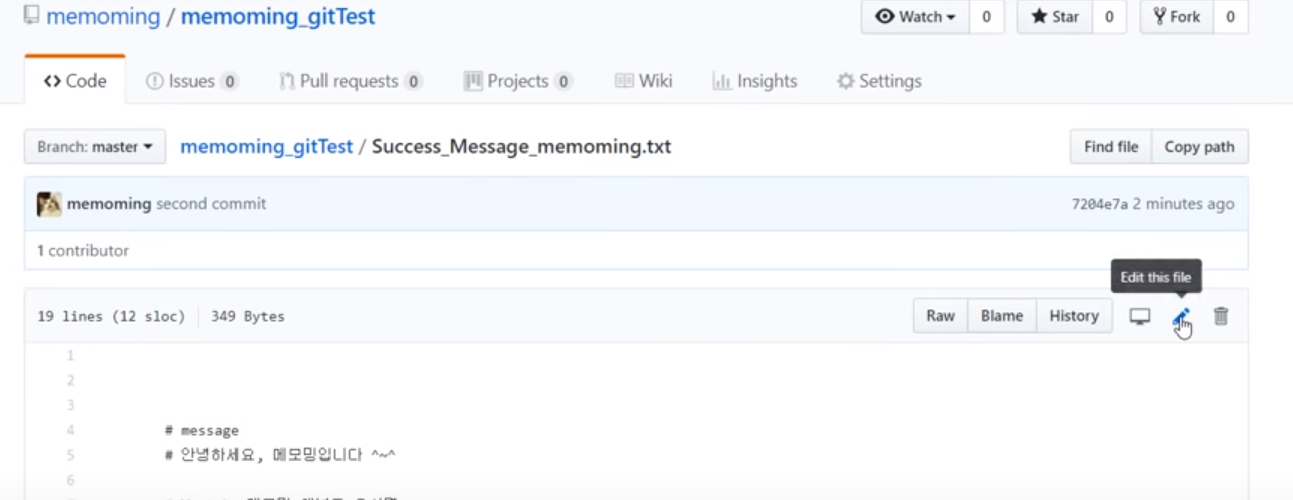
이렇게 수정된 것을 확인할 수 있다.



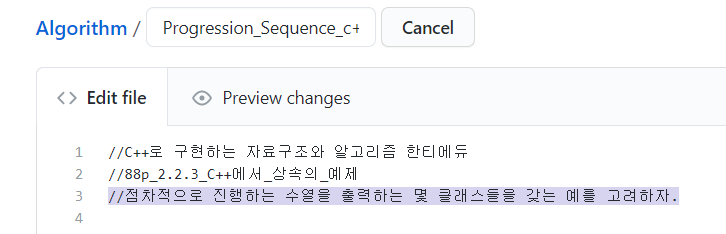
코멘트도 달 수 있다. 왜 이 파일을 수정했는지? 어떤 부분을 같이 회의해야 하는지? 이런 메시지들을 남길 수 있을 것이다.

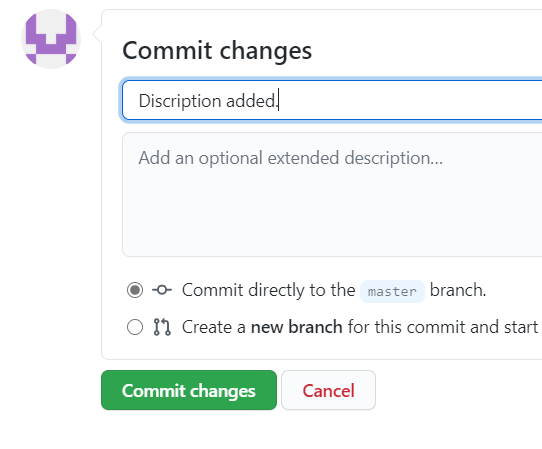
그리고 녹색 라인이 추가된 라인이다.

아래처럼 웹 상 수정도 가능하다.

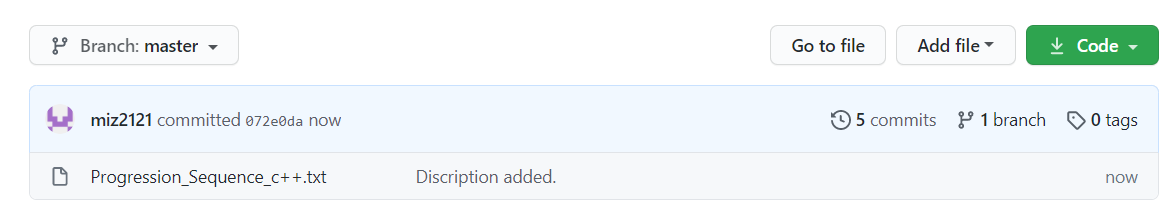


만약 웹 상에서 수정(연필모양, Edit this file) 후 커밋을 입력했다면!



이렇게 수정 하고

이런 커밋을 달았다면!



이렇게 커밋과 함께 표시된다.

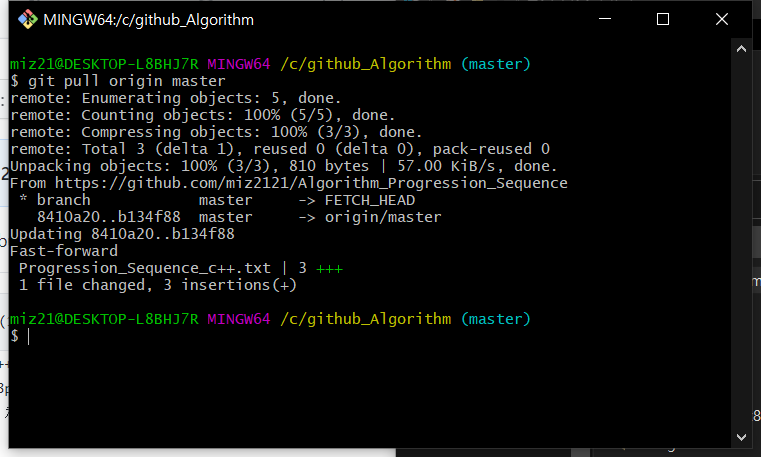
1. 지금 상황은! git repository는 수정 됐지만 내 local folder는 수정되지 않았다. 더 최신 파일이 있는 곳이 github인 상황이다.

|  |
| --- |
|  |

위 이미지를 기억하자.

pull 명령어로 최신 자료를 당겨오지 않고 작업하면! 다른 사람이 작업한 내역 그 자체의 repository에다가 지금 명령어를 입력하면 그대로 덮어쓰게 된다.

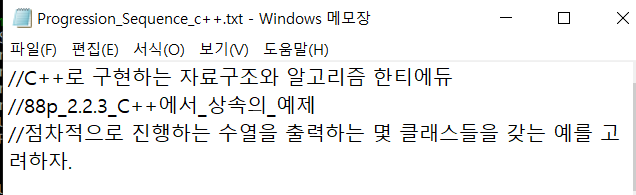
그렇기 때문에 하나의 파일을 같이 수정할 경우 반드시 pull부터 해 주자.



이렇게!

이미지를 읽어보자. 1 file changed, 3 insertions.가 있다.

이 명령어 만으로 local folder 내의 파일이 변경된다. 확인 해 보자.



세 번째 주석이 추가됐다.

클라이언트상에서 아무것도 없을 때 서버의 프로젝트를 내려받는 명령어

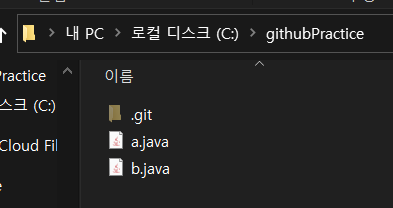
git clone [저장소 주소]

→ git pull과 비슷한 명령어다.

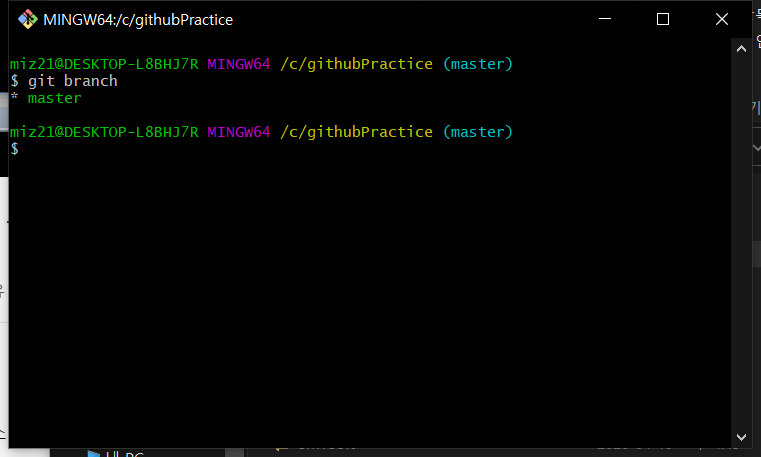
→ 내용을 자동으로 다운로드 받고 init도 된다.

브랜치

git branch



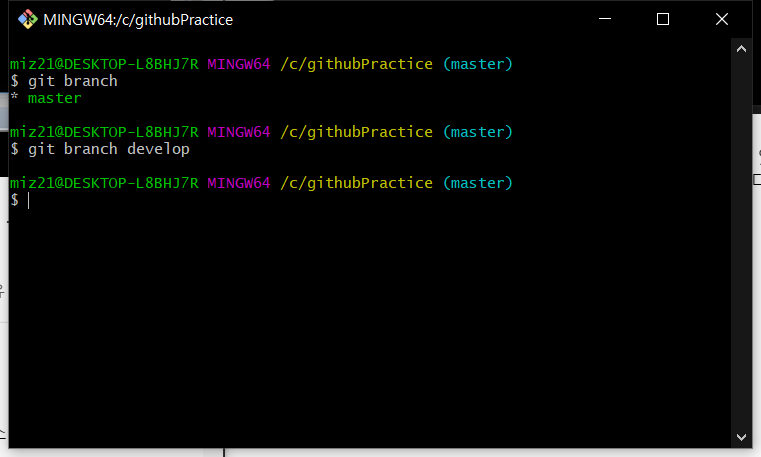
이런 상황이라고 하자.



기본적으로 위 명령어를 실행하면

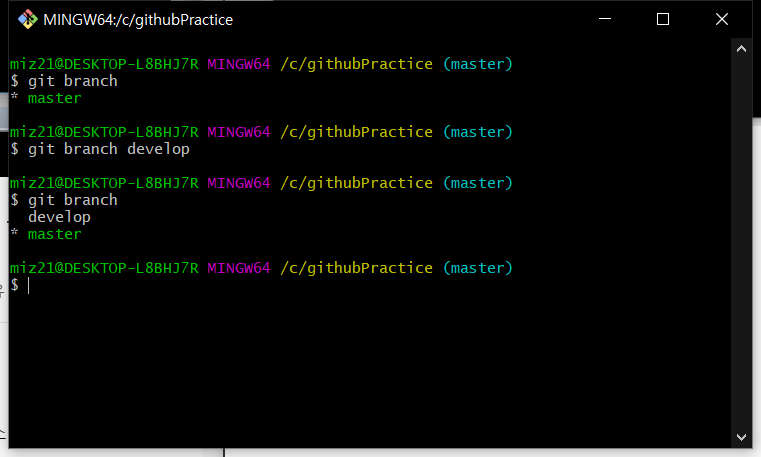
$ git branch 문장 아래에 몇 개의 브랜치가 있는 지 확인할 수 있다.

별도의 브랜치를 만들지 않았으므로 마스터 브랜치만 존재한다.



develop 브랜치를 만들었다.

이제 다시 git branch 명령을 실행하면



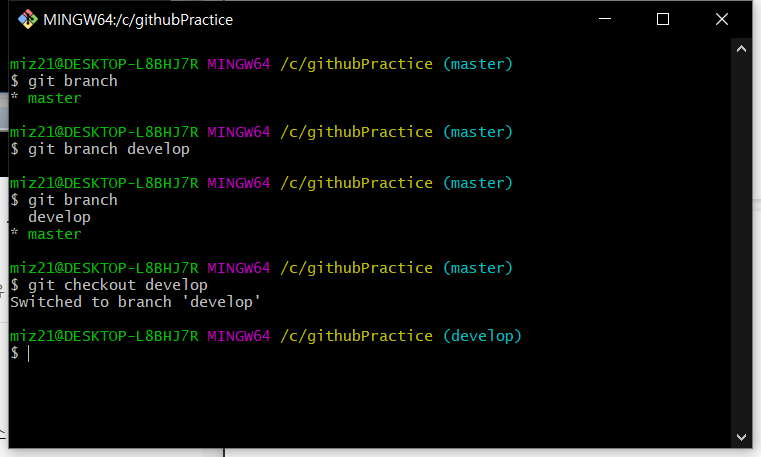
develop 브랜치를 확인할 수 있다.

develop 브랜치로 head를 움직임으로써 다른 브랜치를 확인할 수 있다.

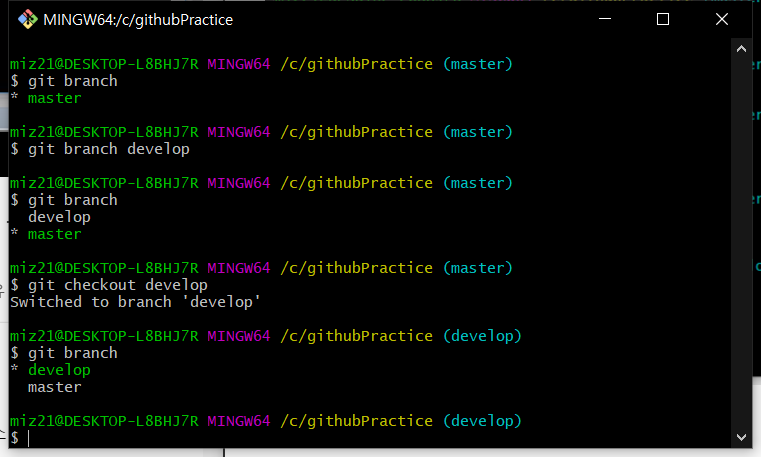
다른 브랜치로 접속하고자 할 땐

git checkout

명령어를 쓴다.

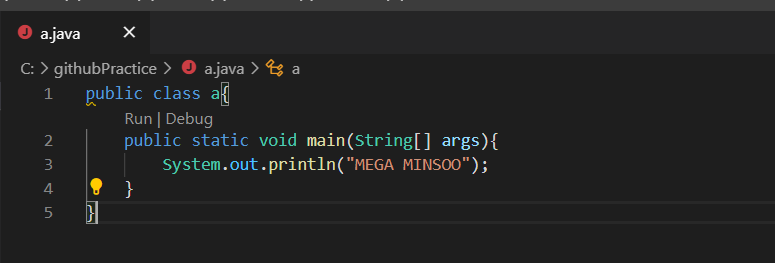


다시 브랜치를 확인 해 보면

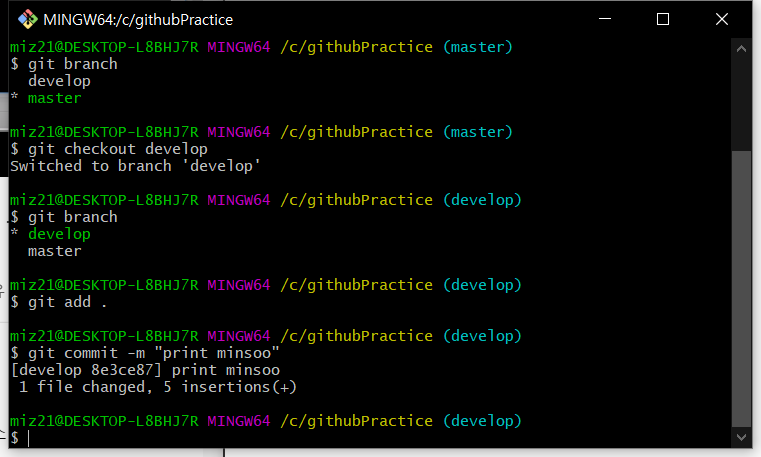


헤드가 바뀌어 있는 걸 확인할 수 있다.

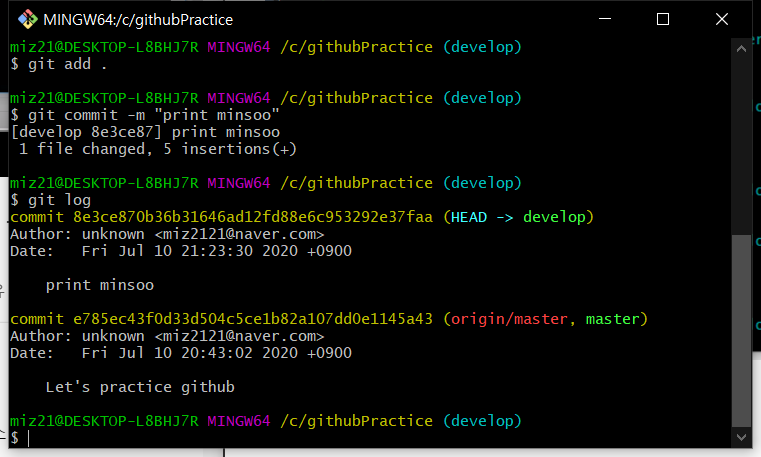
이 상태에서 파일에 내용을 추가해 보자.



이 상태에서 git add ., git commit -m "print minsoo"를 수행해 보자.

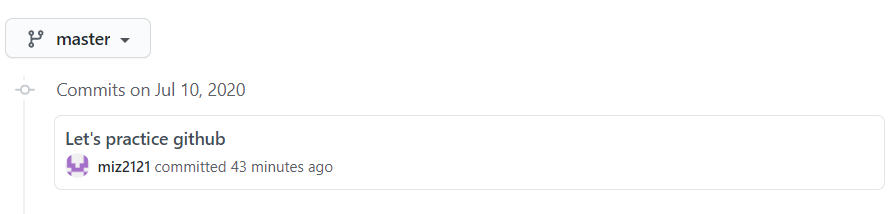


여기서 log를 찍어보면



헤드가 develop을 가리키고 있고, develop 브랜치에서의 커밋을 확인할 수 있다.

깃허브 원격지 주소는(origin/master, master) 마스터 브랜치를 가리키고 있다.

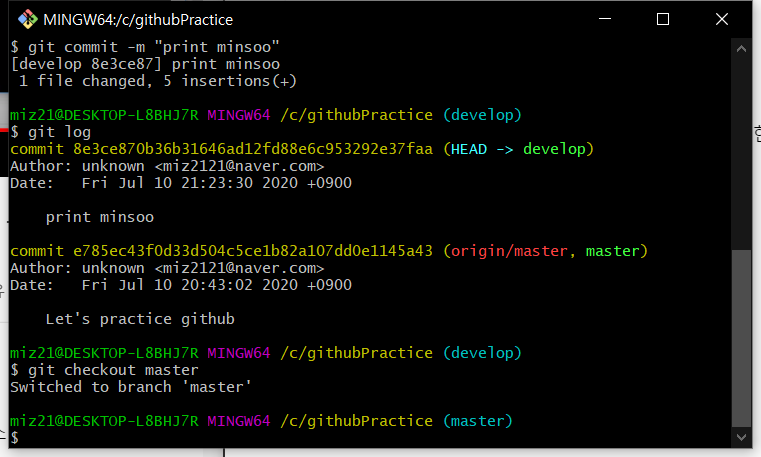


새 변경사항이 반영되지 않았음을 확인할 수 있다.

통합하기

브랜치를 merge하기 위해선 다시 마스터 브랜치로 이동해야 한다.

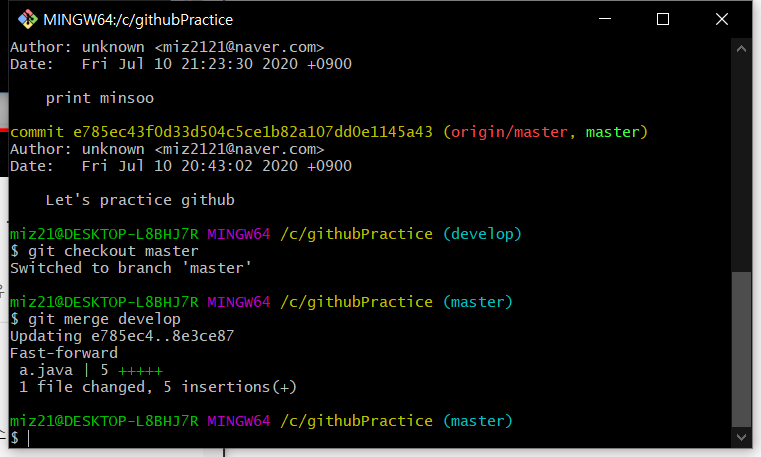
git checkout master를 입력하자.



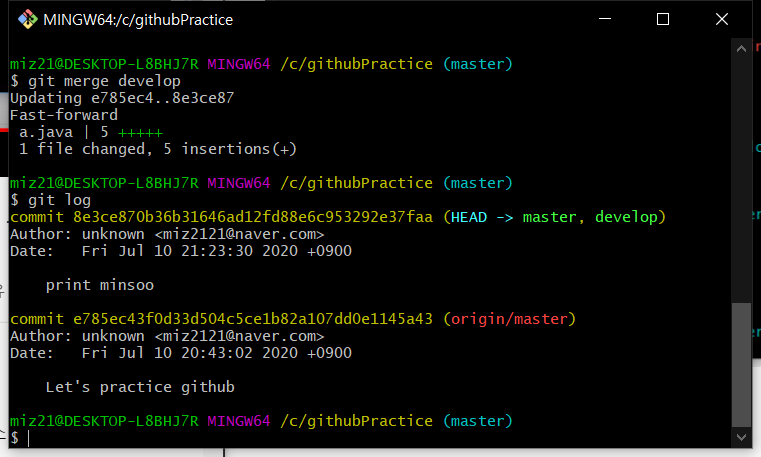
git merge

git merge develop

명령어를 입력하자.

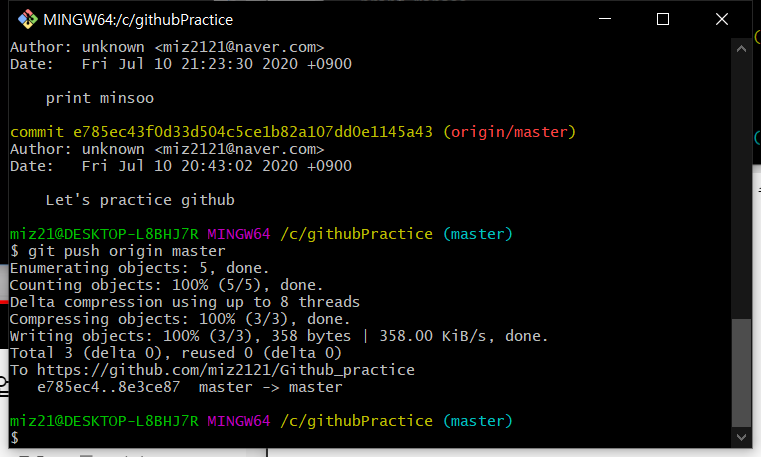


성공했따! 다시 log를 찍어보자

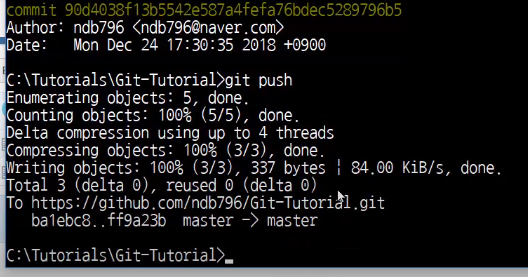


HEAD가 마스터를 가리키고 있고, 반영이 된 걸 확인할 수 있다.

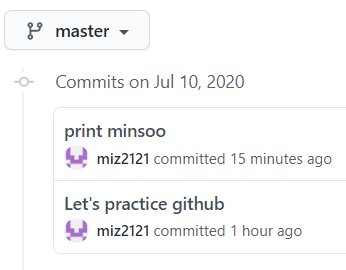
git push를 해주자.



유튜브에선 아래처럼 git push명령어만 입력해도 반영되었다.



아래 이미지처럼 print minsoo커밋이 반영된걸 확인할 수 있다.

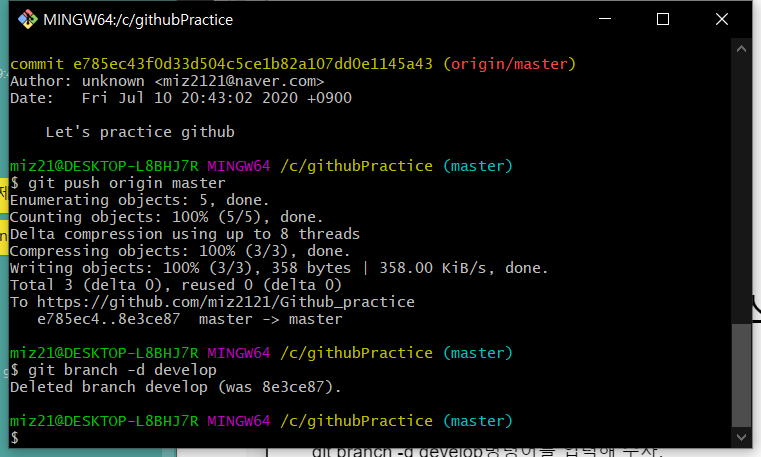


병합이 끝난 브랜치는 삭제 해 주자.

기능이 다 했으므로 삭제해 주어야 한다.

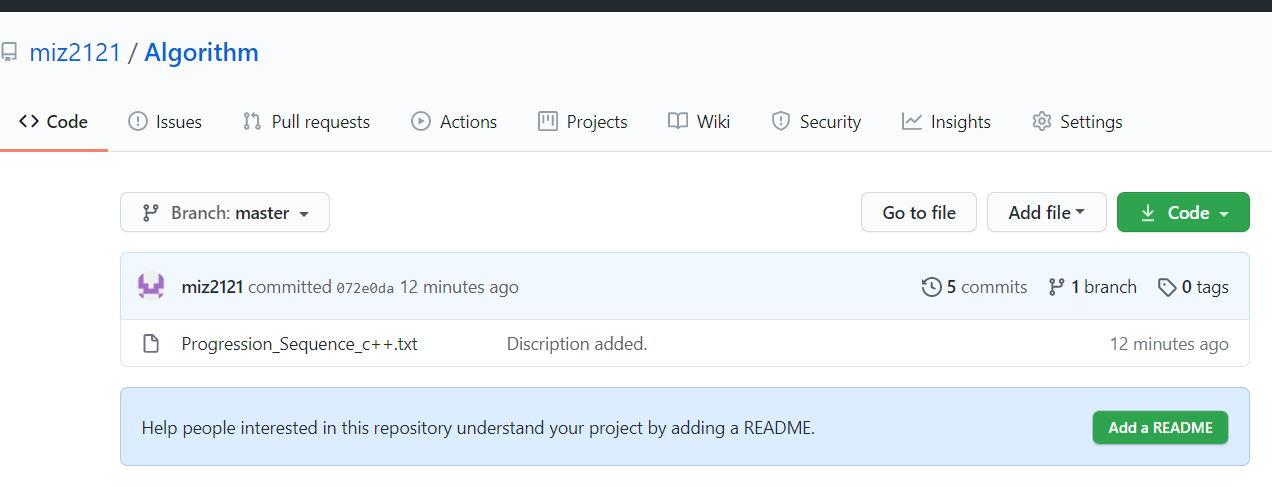
git branch -d

git branch -d develop명령어를 입력해 주자.



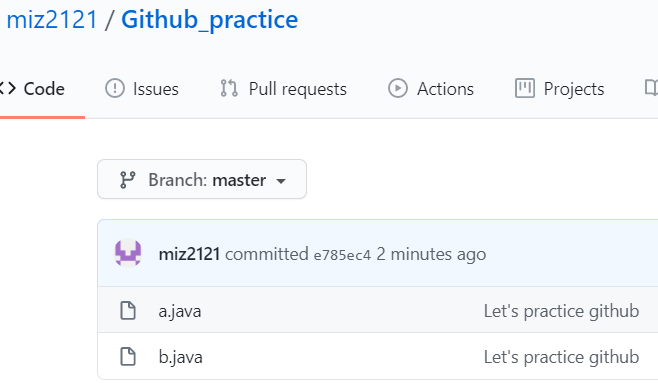
* git brach는 왜 필요할까? 아래는 브랜치 개념

깃허브에 새 프로젝트를 생성하면 일단 모든 프로젝트는 마스터 브랜치만 생성된다.



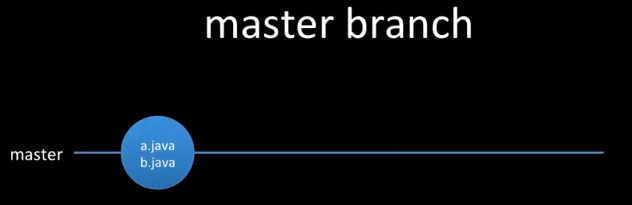
마스터 브랜치만 사용한다면 git branch의 개념을 몰라도 되지만,

보통 직접 작업하는 일은 없다.



현재 프로젝트 마스터 브랜치에 a.java, b.java 파일이 있다고 하자.

그림으로 표현한다면

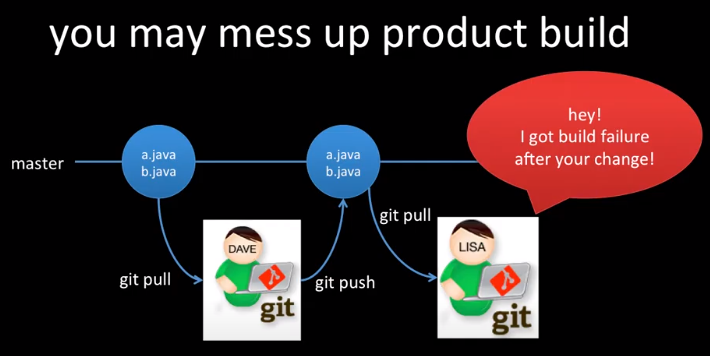


한 개발자가 마스터 브랜치에서 a.java파일을 수정한다고 가정하자.

이 개발자는 아무런 제약이 없이 파일을 수정하고 마스터 브랜치에 본인의 업데이트 된 파일을 올릴 수 있다.

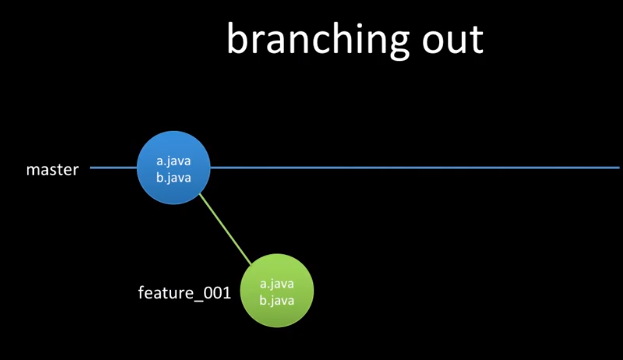
이 개발자의 부주의로, 컴파일 에러가 나는 파일을 push했을 땐 어떻게 될까?

다른 개발자가 마스터브랜치에서 pull하자마자 컴파일도 못하는 상황이 발생한다!

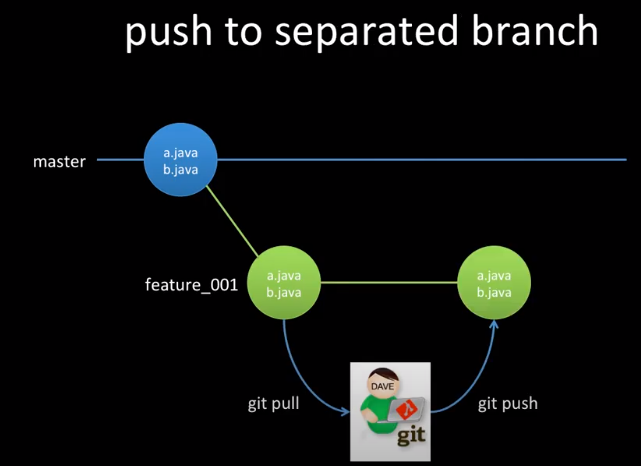


한 명의 실수로 여러 개발자들의 작업을 망칠 수 있다.

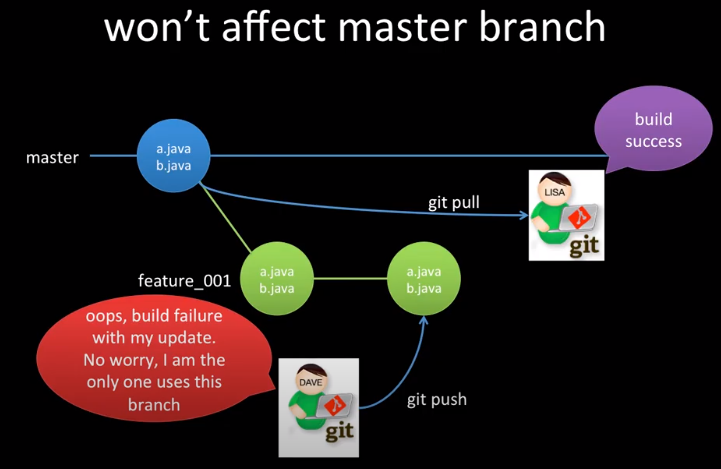
이런 여러가지 이유들로 마스터브랜치에서 직접 수정/개발하는 경우는 거의 없다.

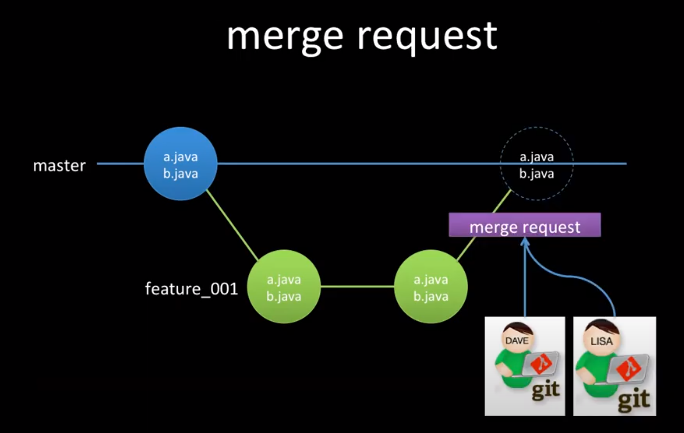


brancing out이란, 현재의 브랜치에서 나만의 브랜치를 생성하는 것을 말한다.

나만 쓰는 브랜치이기 때문에 내 맘대로 수정해도 남들에게 절대 영향을 끼치지 않는다.

만약 컴파일 에러가 나는 파일을 push해도 나만의 브랜치에만 문제가 생겼기 때문에 남들의 일은 중단되지 않을 것이다.

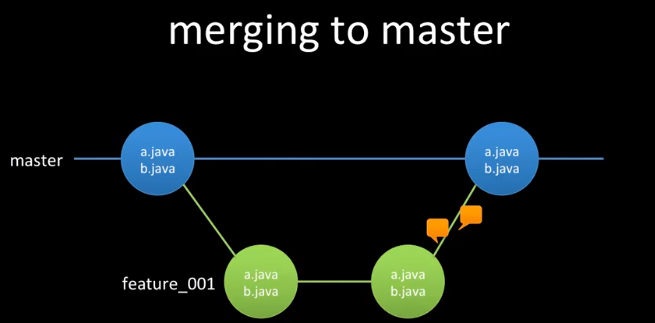


기능 개발이 완료되면, 최종적으로 나만의 브랜치를 전체 브랜치에 적용해 주어야 한다.

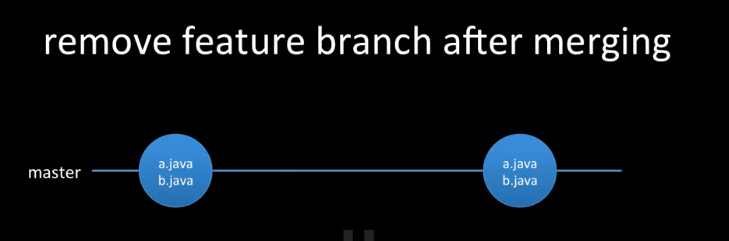
merge request에 특별한 기능이 있는데, code review기능이다.

개발자 a가 개발자 b에게 merge request를 보내면 개발자 b는 code review를 한다.

읽어보고 마음에 들지 않으면 커밋을 남기고 코드 수정이 있을 때 까지 merge하지 않을 것이다. 마음에 든다면 master branch로 merge 할 것이다.



merge가 완료되면 나만의 브랜치를 깃허브에서 지우게 된다.



ex) 아래 이미지를 참고하자.

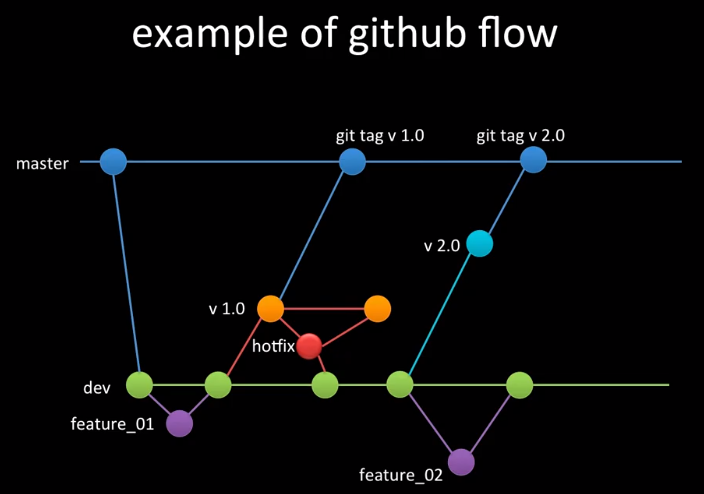
마스터 브랜치에서 보통 개발브런치를 바로 생성시키고 마스터브랜치는 버전관리용으로만 쓴다. → master brach는 배포가 가능한 수준의 안정화된 버전이다.

개발 브랜치에서 개발자들은 보통 task별로 branching out 해서 작업하고(feature)

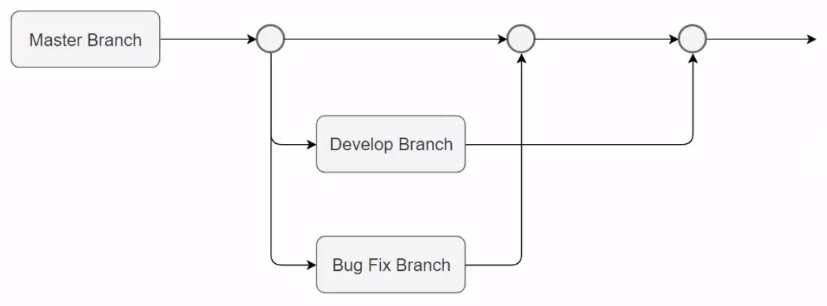
작업 완료 이후엔 본인의 브랜치를 삭제할 것이다.

버전 릴리즈 할 땐 버전 브랜치를 개발 브랜치(dev)에서 branching out 하고 버전 브랜치를 마스터 브랜치에 머지하여 마스터 브랜치에서 각각의 버전을 태그로 관리한다.

만약 릴리즈된 버전에 긴급한 버그수정이 필요할 때에는 hotpix 브랜치를 릴리즈 브랜치에서 생성하고 버그수정이 완료 됐을 땐 버전 브랜치 및 개발브랜치(dev)로 수정을 fetch하면 된다.



간단하게는 아래와 같다.



cf) 통합 브랜치: 배포가 가능한 수준의 브랜치로 일반적으로 마스터 브랜치를 의미한다

cf) 토픽 브랜치: 특정한 기능을 위해 만들어진 브랜치로 일반적으로 마스터 브랜치 이외의 다른 브랜치를 의미한다.