

codingOn X posco

K-Digital Training 신재생에너지 활용 IoT 과정

Git의 기초

오늘 수업은?

- 지난 시간에 Git을 설치하였습니다.
- 이번 시간에는 개발을 진행하면서 Git을 왜 사용하고 어떻게 사용하면되는지 자세히 알아보겠습니다.

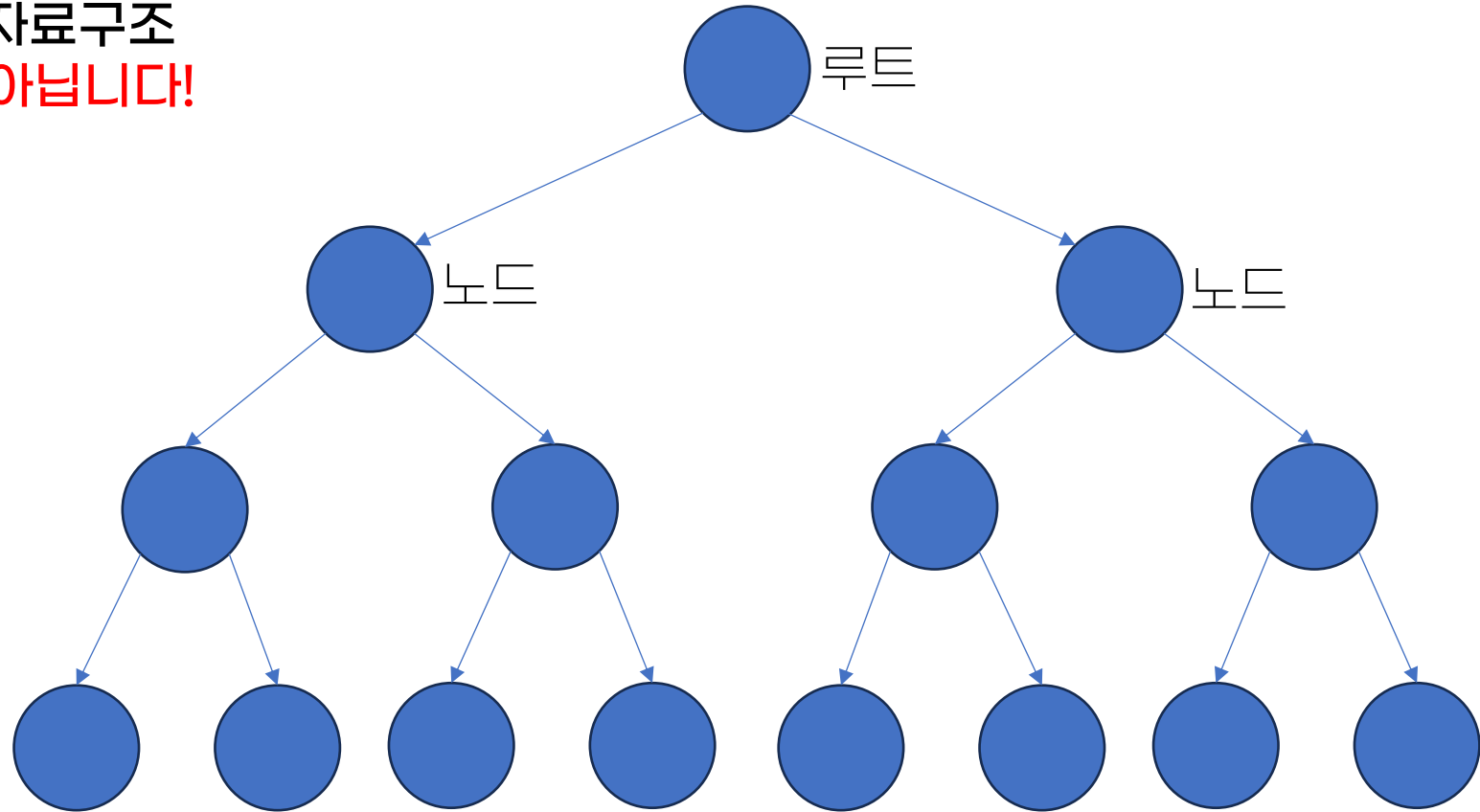
학습목표

- 트리형태의 폴더구조에 대해서 이해한다.
- github에 원격저장소를 생성할 수 있다.
- 원격저장소에 코드를 올리고 받을 수 있다.
- 브랜치를 만들고 다시 합칠 수 있다.
- 충돌을 해결 할 수 있다.

폴더 구조 이해

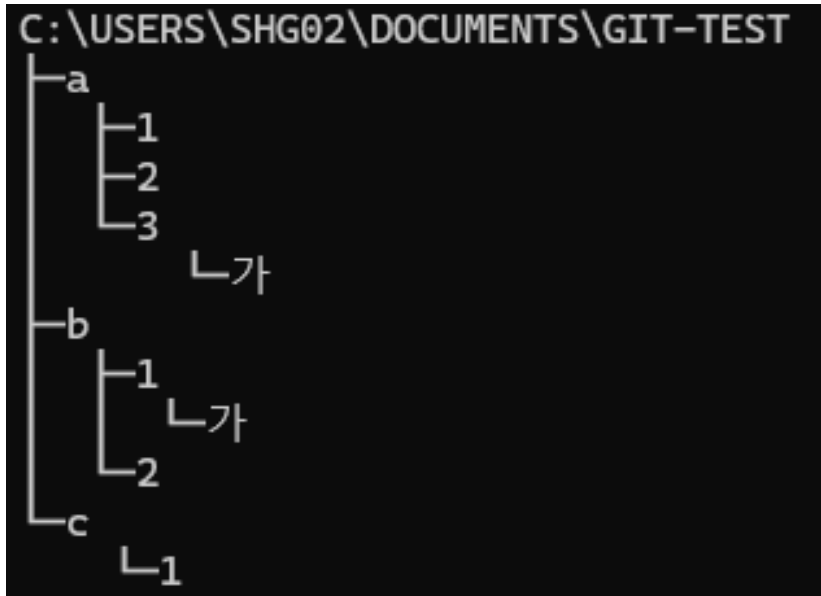
폴더 구조의 이해

- 트리구조
 - 자료구조에서 사용하는 용어로 계층적 관계를 나타내는데 사용하는 비선형 자료구조
 - => **우리는 자료구조를 하려는게 아닙니다!**
- 앞으로 개발을 진행하면서 폴더 구조에 대한 이해를 돕기위한 형태
- 루트가 프로젝트 소스코드를 담기위한 최상위 폴더
- 루트폴더 하나가 프로젝트 단위



폴더 구조의 이해

- 프로젝트에서 폴더 구조는 매우 중요
- 루트가 되는 폴더는 Git 저장소와 연결됨
- 현재 작업하는 파일의 폴더가 어디인지 꼭 확인하는 버릇이 필요



- 프로젝트 폴더 구조 예시
 - 루트 폴더: git-test
 - 하위 폴더: a, b, c
 - a 하위 폴더 : 1, 2, 3
 - b 하위 폴더 : 1, 2
 - c 하위 폴더 : 1
 - a > 3 하위 폴더 : 가
 - b > 1 하위 폴더 : 가

폴더 구조 확인하기

- (window) cmd 또는 git bash 실행
- (mac) 터미널 실행
- 폴더 구조 확인 명령어(구조를 볼 폴더의 부모 폴더에서)
 - tree 폴더명
- 명령어 실행이 안된다면?
 - (mac) brew install tree
 - (window) <https://gnuwin32.sourceforge.net/packages/tree.htm> 접속
 - 중간 Download의 Binaries에 zip 파일 다운
 - Binaries [Zip](#) 41328
 - 압축해제하면 bin폴더 안에 tree.exe가 존재
 - tree.exe
 - C:\Program Files\Git\usr\bin에 tree.exe파일을 옮겨넣어 줌

프로젝트 루트 폴더 생성

- (window) git bash 실행
- (mac) 터미널 실행
- 코드를 저장할 루트 폴더를 생성(루트 폴더와 원격저장소가 연결)
 - 터미널 주요 명령어
 - ls : 폴더에 있는 모든 파일 검색(상세 검색은 ls -al)
 - cd 폴더명 : 폴더로 이동
 - 예) cd Documents
 - mkdir 폴더명 : 폴더 생성
 - 예) mkdir git-test
 - pwd : 현재 위치한 전체 경로

Git 설정

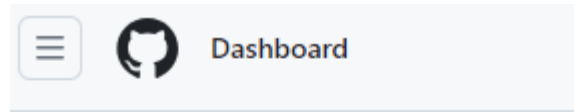
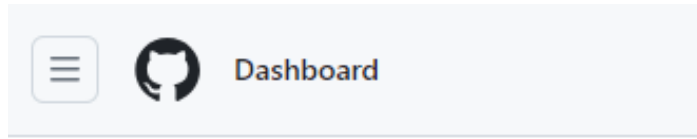
Git 설정

- (window) cmd 또는 git bash 실행
- (mac) 터미널 실행
- 설정확인 : `git config --global --list`
- 이름등록 : `git config --global user.name "프로필 이름"`
 - 예) `git config --global user.name "codingon"`
- 메일등록 : `git config --global user.email "이메일 주소"`
 - 예) `git config --global user.email "codingon@gmail.com"`
- default 브랜치가 main이 아니라면
 - `git config --global init.defaultBranch main`

원격저장소 생성

원격저장소 생성

<https://github.com/>



메인화면에서



나의 저장소 리스트 화면에서

원격저장소 생성

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere?

[Import a repository.](#)

Required fields are marked with an asterisk ().*

Owner *

Repository name *

/ 저장소 영문명

Great repository names are short and memorable. Need inspiration? How about [legendary-octo-goggles](#) ?

Description (optional)



Public

Anyone on the internet can see this repository. You choose who can commit.



Private

You choose who can see and commit to this repository.

원하는 저장 방식을 선택

Create repository

원격저장소 생성

Quick setup — if you've done this kind of thing before



Set up in Desktop

or

HTTPS

SSH

`https://github.com/[username]/git-test.git`



Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

```
echo "# git-test" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/[username]/git-test.git
git push -u origin main
```



ctrl + v 복사


원격저장소와 내 폴더 연결

- (window) git bash 실행
 - (mac) 터미널 실행
1. 위에서 만든 프로젝트 루트 폴더로 이동
 - 예) `cd git-test`
 2. Git 과 연결(명령어 차례대로 입력)
 - `git init`
 - `git remote add origin https://github.com/이름/저장소명.git`

```
MINGW64 ~/Documents/git-test
$ git init
Initialized empty Git repository in C:/Users/shg02/Documents/git-test/.git/

MINGW64 ~/Documents/git-test (main)
$ git remote add origin https://github.com/[redacted]/git-test.git
```


명령어 살펴보기

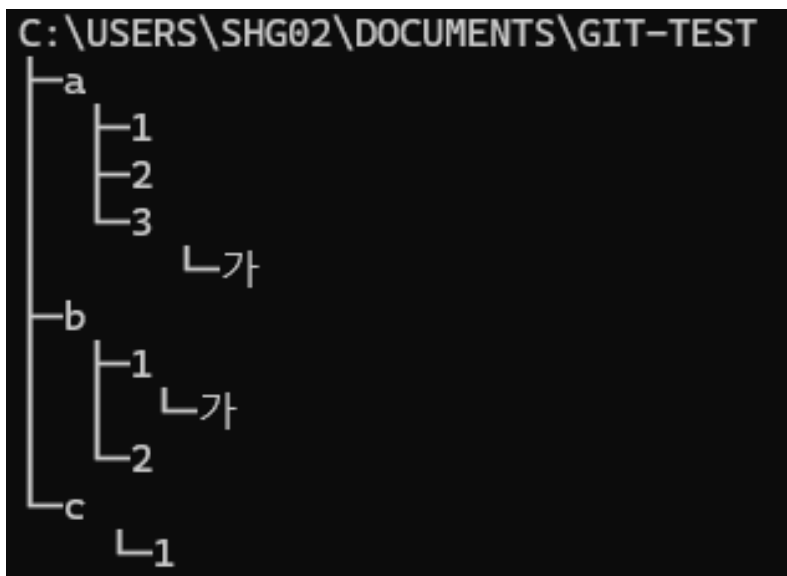
- git init
 - Git 로컬저장소가 생성됨
 - main 브랜치 생성
 - 해당 명령어 실행 후 폴더에 .git 폴더가 숨김폴더로 생성됨  .git
 - 숨김 폴더 확인하기
 - (window) 탐색기에서 보기 -> 표시 -> 숨김항목 체크
 - (mac) Finder에서 Cmd + Shift + .
- git remote add origin <https://github.com/이름/저장소명.git>
 - 위 초기화된 .git폴더에 원격저장소 연결
- 원격저장소를 변경하고 싶으면 위 숨김폴더인 .git폴더 삭제 후 다시 git init과 git remote add origin 저장소 주소.git을 입력하면 됨
- 정상적으로 연결이 되었다면 `~/Documents/git-test (main)` 폴더명 뒤에 main이 보여짐

- CLI란?
- CLI는 Command Line Interface의 약자로 터미널 창에서 명령어를 입력하여 실행하는 것을 뜻함
- CLI를 사용하는 것이 처음에는 어려울 수 있음
- 이 어려운 것을 해결하기 위해 GUI(Graphic User Interface)형태의 프로그램도 존재
 - 예) 소스트리
- 하지만 작업속도와 Git의 모든 기능을 사용하기 위해서는 CLI를 사용하는 것이 좋음
- CLI에 먼저 익숙해 진 후 GUI를 사용해도 무관

실수 사례

- git init은 루트 폴더에서 해야하는데 폴더 구조를 생각하지 않아서 하위 폴더에서 또 git init을 하는 경우가 발생하여 루트폴더가 push가 안되는 사례

• 예)



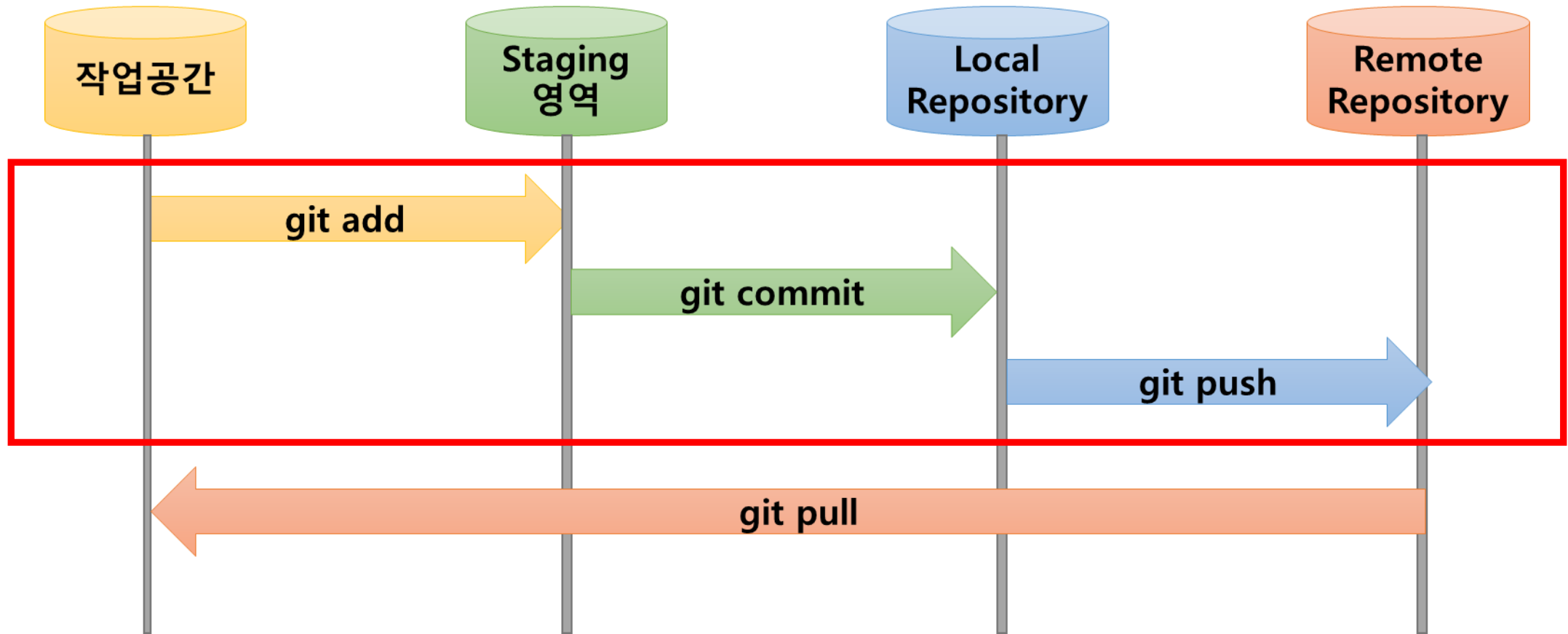
git-test폴더(루트폴더)에서 git init 진행 후 하위 폴더가 많아지고 파일이 많아지게 되면서 하위 폴더인 a폴더에서 git init을 하여 또다른 원격저장소와 연결하는 경우가 있었음

- ⇒ 이 경우 git-test폴더에서 push를 하게 되면 오류가 뜸
- ⇒ git은 .git의 내용으로 원격저장소를 찾게 되는데 git-test폴더의 안에 .git이 두개가 생기게 되면서 git은 어디로 저장되어야하는지 알 수 없게됨

- 꼭 파일이 생성되는 폴더의 위치를 잘 확인하고 작업을 진행!!!

원격저장소에 올리기

Git의 흐름도



코드를 왜 올려?

- 코드 백업 : 작업한 코드를 안전하게 저장
- 협업 용이 : 팀원들과 쉽게 코드를 공유하고 함께 작업 가능
- 버전 관리 : 코드 변경 이력을 남겨 언제든지 이전 버전으로 되돌리기 가능
- 충돌 방지 : 팀원들과 작업할 때 변경 사항을 병합하여 코드가 중복되거나 누락되는 현상(충돌)을 방지 할 수 있음
- 문제 해결 용이 : 문제 발생시 코드 버전 이력을 확인하여 변경 사항을 추적

업로드할 파일 생성

- VS Code 실행
- 파일 -> 폴더열기에서 Git에 연결했던 루트 폴더를 선택
- 팁) 터미널에서 빠르게 열기
 - 원하는 폴더 위치에서 `code .` 입력

```
MINGW64 ~/Documents/git-test (main)  
$ code .
```

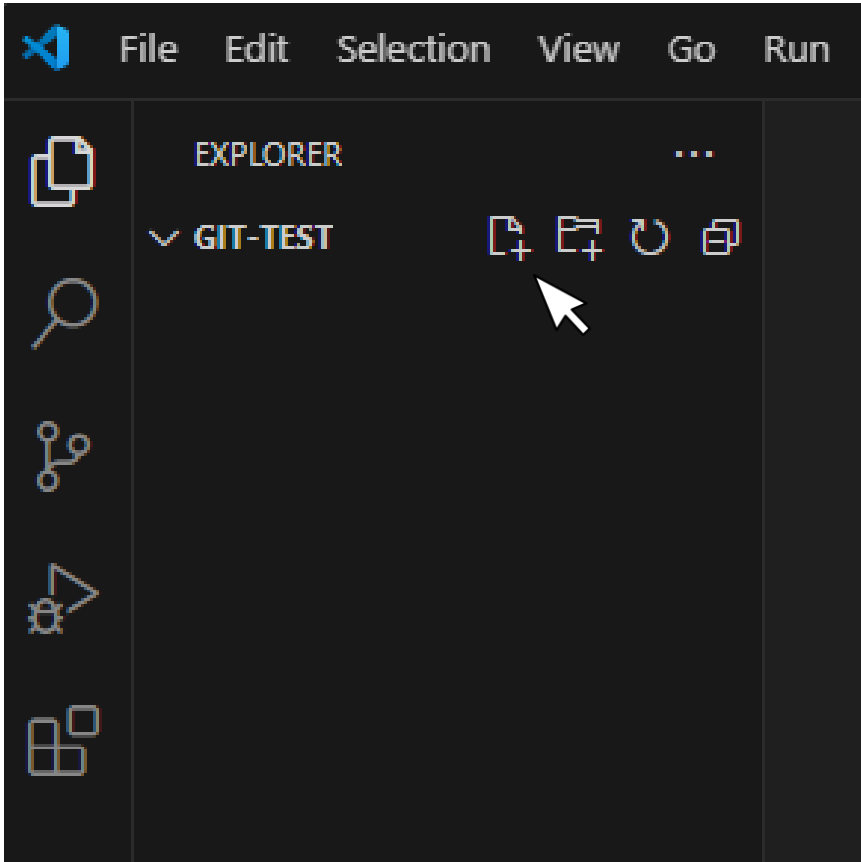
- 맥에서 위 명령어가 실행되지 않는다면?
 - VS Code 실행 후 Cmd + Shift + p 버튼 누른 후 shell command 입력 후 아래 나오는 install 'code' command in PATH 설치

```
>shell command
```

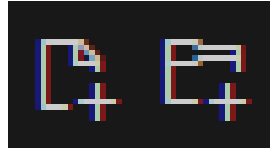
```
셸 명령: PATH에 'code' 명령 설치
```

```
Shell Command: Install 'code' command in PATH
```

업로드할 파일 생성

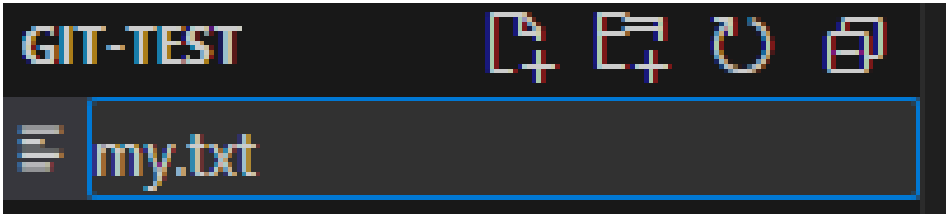


마우스 커서를 올려놓으면
아이콘이 나타납니다

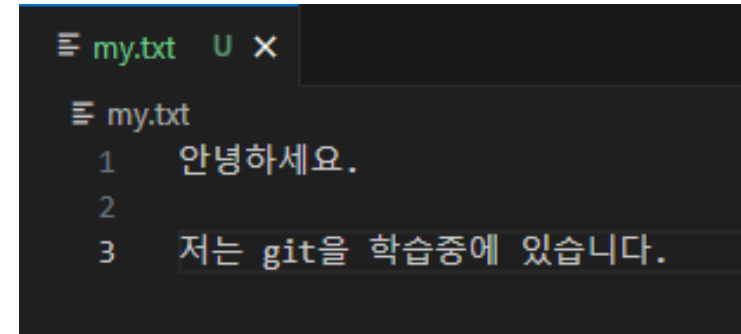


- 왼쪽 아이콘 : 신규 파일 생성
- 오른쪽 아이콘 : 신규 폴더 생성

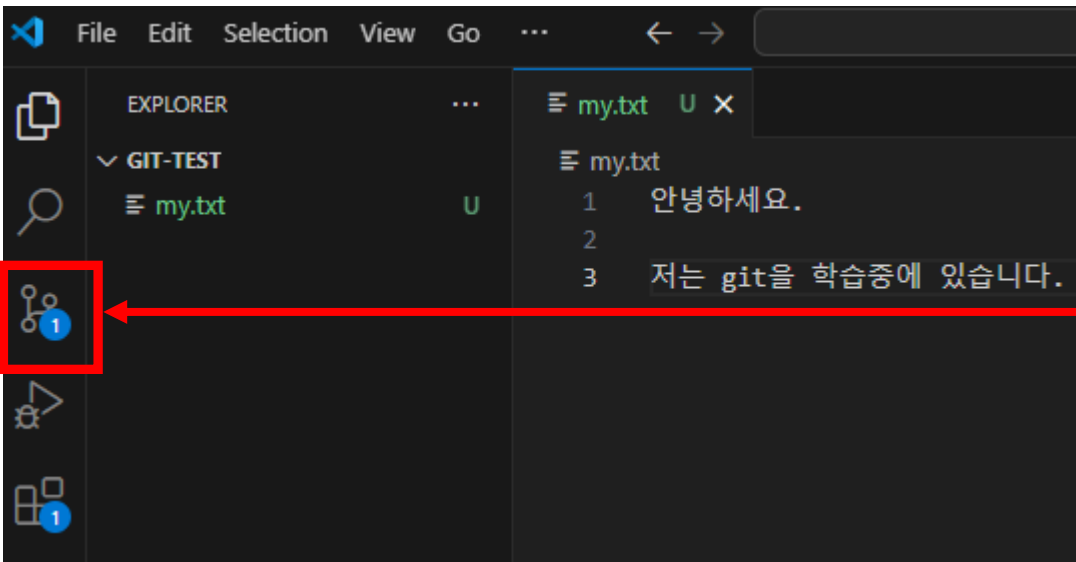
업로드할 파일 제작



파일 생성시 파일명.확장자로 생성



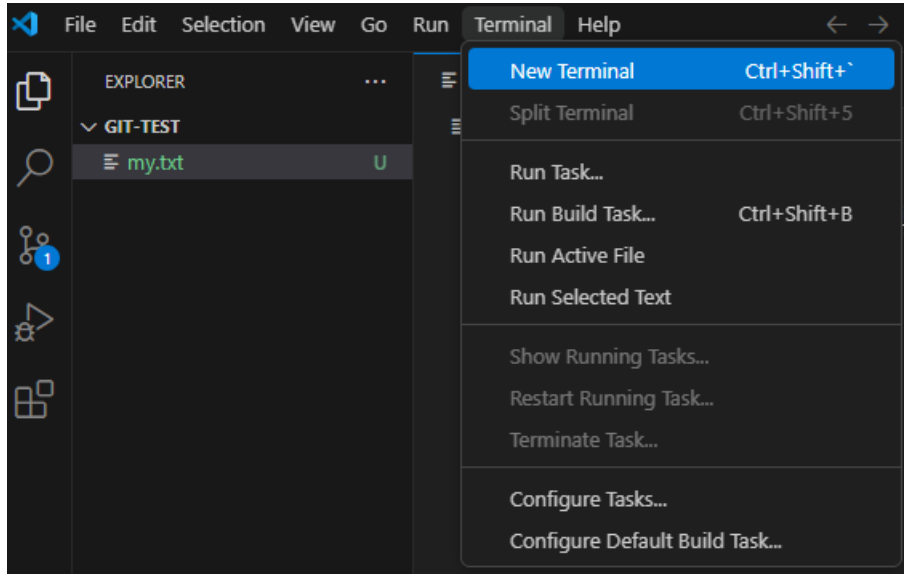
원하는 텍스트 내용 작성해주세요



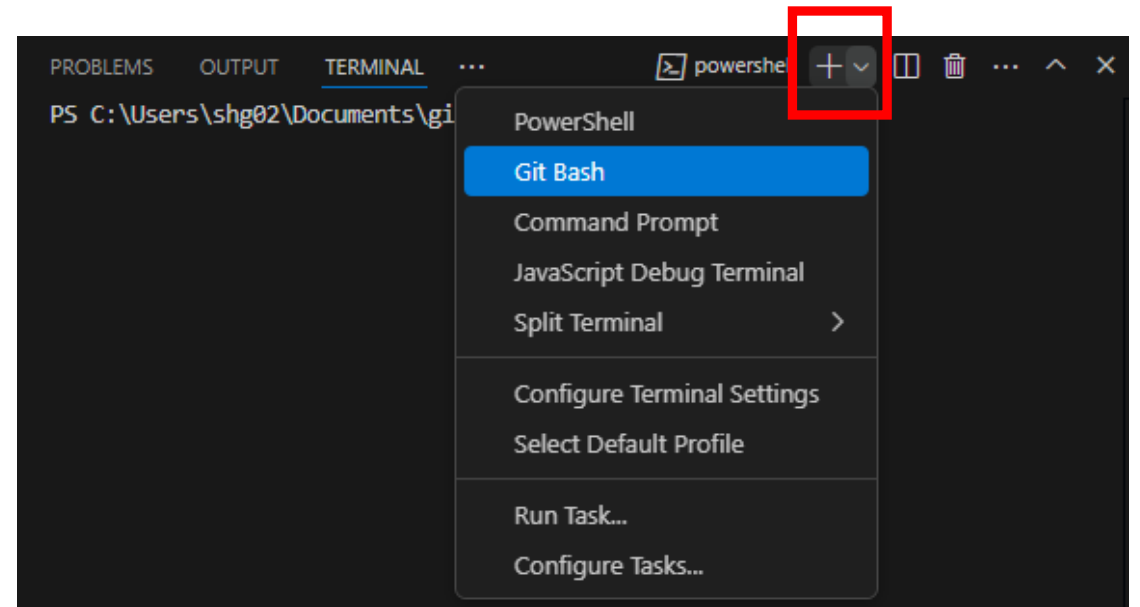
Git 연결 후 신규로 파일이 생성되거나 수정이 되면 해당 아이콘에 추가,변경된 파일의 숫자가 표시됨

터미널 준비

- (window) git bash 실행
- (mac) 터미널 실행
- VS Code에서 터미널 실행



상단 메뉴 터미널 -> 뉴 터미널



하단 터미널 창을 Git Bash로 변경

원격저장소에 파일 올리기

- Git 원격저장소로 올리기 위해서는 커밋을 해야함
- 커밋이란 버전을 뜻함. 한번 커밋하면 하나의 버전이 생성됨
- **커밋 순서(하나라도 빼놓고 진행하면 안됩니다)**
 - 커밋할 파일 추가
 - **git add 파일명** or **git add .** (. 은 한칸띄고! 폴더 전체라는 의미를 가짐)
 - ex) git add my.txt
 - 커밋 메시지 작성
 - **git commit -m "메시지 작성"**
 - ex) git commit -m "first commit"
 - 원격저장소의 브랜치로 파일 올리기
 - **git push origin 브랜치명**
 - ex) git push origin main

원격저장소에 파일 올리기

- Git 원격저장소로 올리기 위해서는 커밋을 해야함
- 커밋이란 버전을 뜻함. 한번 커밋하면 하나의 버전이 생성됨
- 커밋 순서

```
MINGW64 ~/Documents/git-test (main)
$ git add .

MINGW64 ~/Documents/git-test (main)
$ git commit -m "first commit"
[main (root-commit) 6d63845] first commit
1 file changed, 3 insertions(+)
create mode 100644 my.txt

MINGW64 ~/Documents/git-test (main)
$ git push origin main
info: please complete authentication in your browser...
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 286 bytes | 286.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/[redacted]/git-test.git
* [new branch]      main -> main
```

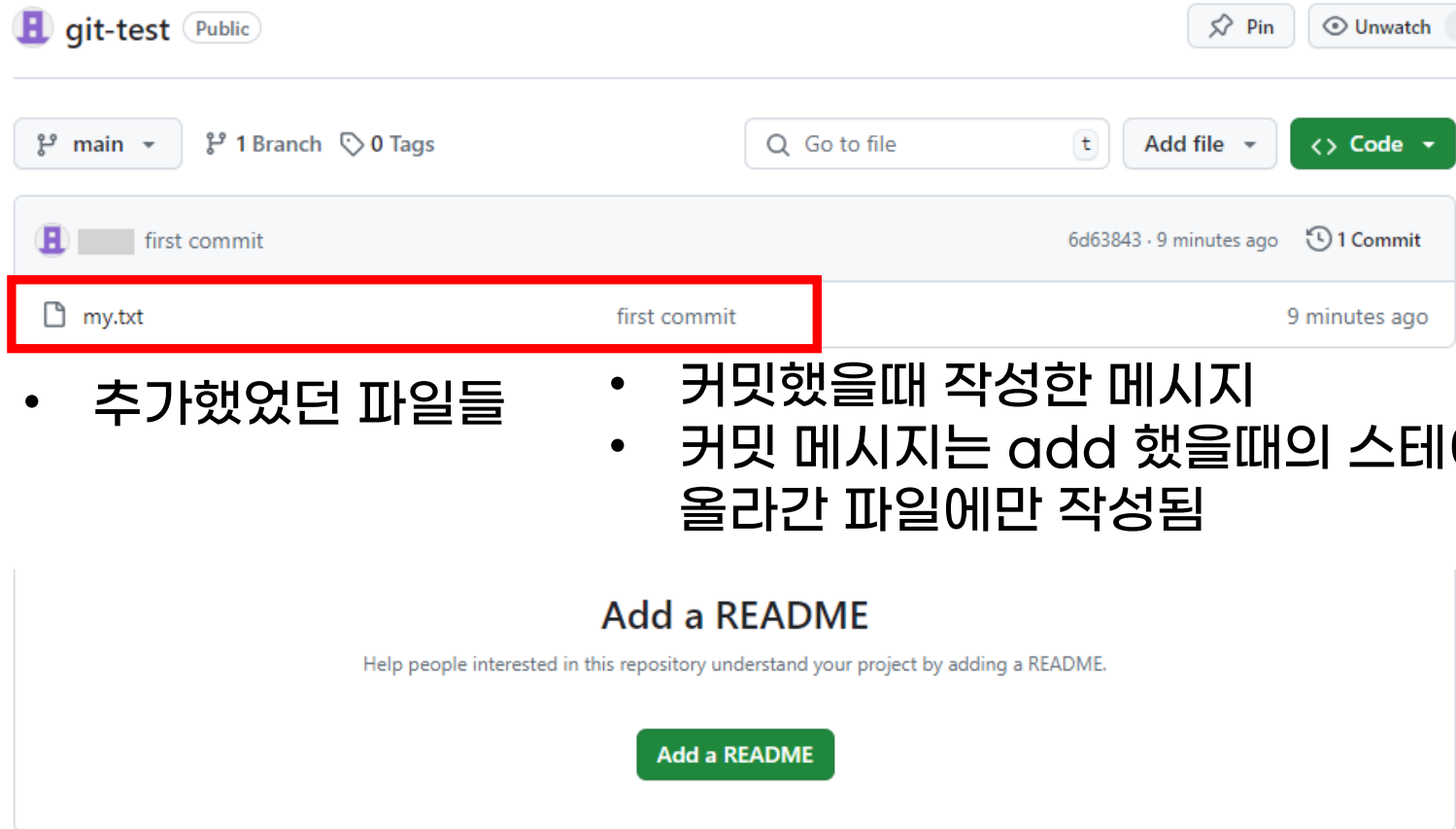
커밋할 파일 추가(스테이징에 등록)

커밋 메시지 작성(로컬저장소에 저장)

원격저장소의 브랜치로 파일 올리기

원격저장소에 파일 올리기

- 푸시 완료 후 github.com에 생성한 저장소로 이동



- 추가했었던 파일들
- 커밋했을때 작성한 메시지
- 커밋 메시지는 add 했을때의 스테이징에 올라간 파일에만 작성됨

Q & A

- 파일을 신규로 제작하고 push하게 되면 원격저장소에 파일이 업로드됨
만약 기존 파일에서 코드만 수정만 한다면?
⇒ 다시 파일이 업로드 되는게 아닌 코드만 수정됨. 스냅샷 방식
- 폴더에 아무 파일도 존재하지 않는다면?
⇒ 해당 폴더는 원격저장소에 업로드 되지 않음. 적어도 하나의 파일이 존재해야함
⇒ 폴더가 올라가는게 아닌 파일이 올라가는 것이기 때문
- 중요한 정보가 담긴 파일도 Git 저장소에 올려도 되나요?
⇒ **아니요! 절대 안됨!!!!!!!!!!** .gitignore 파일을 만들어서 파일을 무시하게 해야함

.gitignore

- 모든 파일이 전부 Git 원격저장소에 올릴 필요가 없음
- 특히 추후 배울 API key, DB 접속정보 등 외부에 노출하면 안되는 정보는 Git 원격저장소에 올리면 절대 안됨
 - 예) AWS 접속정보가 Git에 올라가면 AWS측에서 개인정보가 누출되어있다고 경고 메일이 옵니다.
- 올리면 안되는 파일들에 대한 정보를 담는 파일이 .gitignore 파일임
- .gitignore 파일은 루트폴더가 있는 곳에 있어야함

.gitignore

- 내용
- *.txt : 확장자가 txt로 끝나는 파일 모두 무시
- !text.txt : text.txt는 무시되지 않음
- test/ : 루트 폴더내 하위 폴더중 test폴더의 파일들은 무시한다는 뜻(상대 경로)
- /test : 루트 폴더내 test폴더를 무시(절대 경로)
/ 가 프로젝트 루트 기준으로 경로 지정

```
.
├── .gitignore
├── test
│   └── b.exe
└── tmp
    └── test
        └── a.exe
```

3 directories, 3 files

- 무시되는 파일 예시
- test/ : b.exe, a.exe
- /test : b.exe




실습. 원격저장소에 파일 올리기

1. 파일을 2개 더 생성하기

hello.txt, test.txt

2. 신규 생성한 파일에 텍스트로 글을 작성

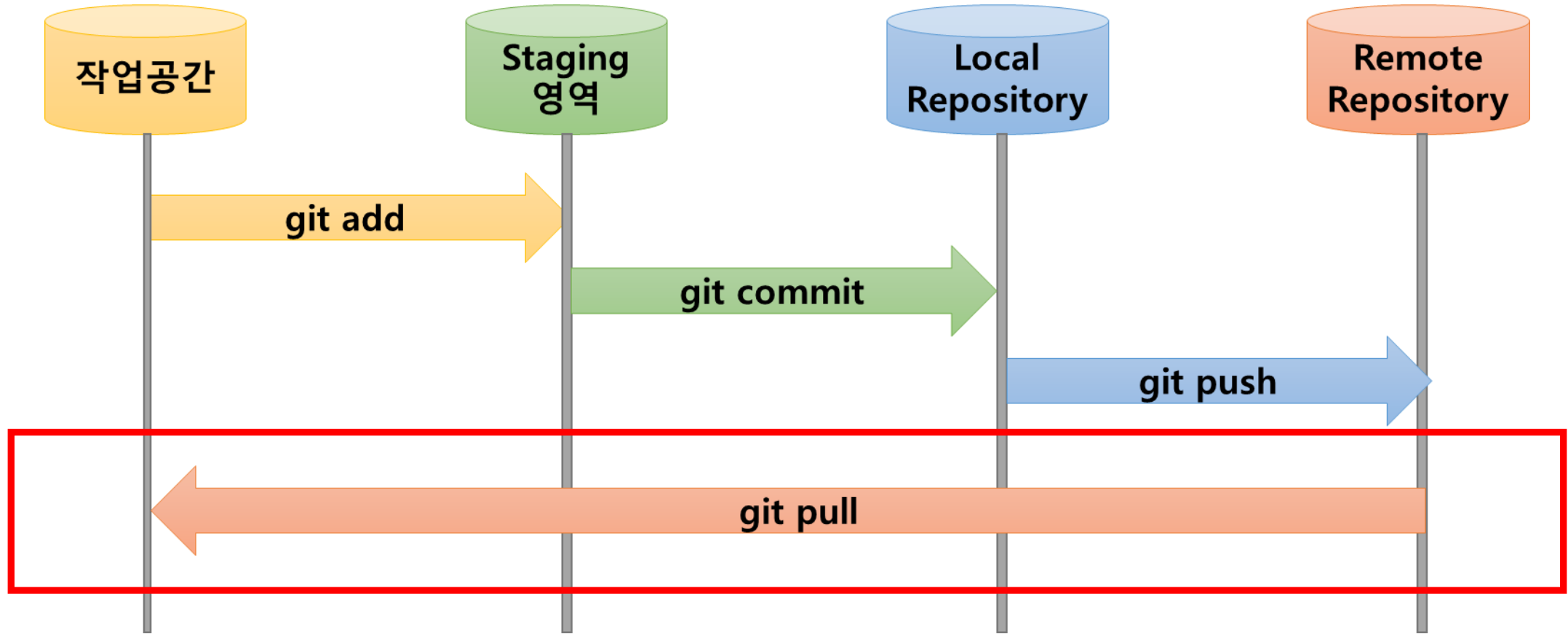
- 위 생성된 모든 파일을 원격저장소로 올려보세요

 hello.txt	실습. 원격저장소에 파일 올리기
 my.txt	first commit
 test.txt	실습. 원격저장소에 파일 올리기

실습 완료시 github에 올라간 화면 예시

원격저장소 내려받기

Git의 흐름도

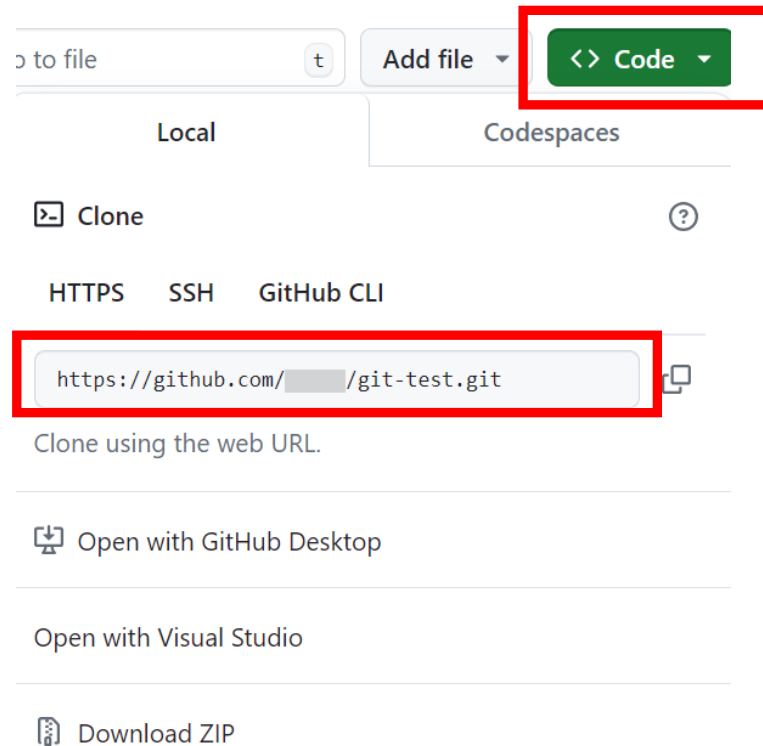


코드를 왜 받아?

- 코드 복사 : 원격저장소에 있는 코드를 내 컴퓨터에 백업 가능
- 협업 용이 : 팀원들이 작업중인 코드를 내 컴퓨터에서 작업 가능
- 최신 상태 유지: 항상 최신 버전의 코드를 가져와서 최신 상태로 작업 가능
- 빠른 시작 : 기존 프로젝트를 바로 내 컴퓨터에서 실행하고 테스트 가능
- 프로젝트 설정 : 프로젝트 처음 설정 시 필요한 파일과 폴더 구조를 한번에 가져 올 수 있음

내 컴퓨터에 로컬저장소 생성하기

- 원격저장소에 있는 프로젝트를 처음으로 내려받거나 다른 팀원의 저장소를 내 컴퓨터에 내려받을 때
- 저장소 주소 찾는 방법



내 컴퓨터에 로컬저장소 생성하기

방법1. 신규 폴더 생성 후 내려받기

- 원하는 폴더를 생성
 - mkdir 폴더명
 - ex) mkdir git-clone
- 원격저장소 복사하기(생성한 폴더로 이동 후)
 - **git clone 원격저장소 주소 .** (. 은 한칸띄고! 이 점이 중요 포인트!)
 - ex) git clone https://github.com/원격저장소 주소.git .

```
MSYS ~/Documents/git-clone  
$ git clone https://github.com/ /git-test.git .
```

• .git
• hello.txt
• my.txt
• test.txt

clone이 완료되면 폴더안에 원격저장소에있던 파일들이 전부 받아져있는 것을 확인할 수 있습니다.

내 컴퓨터에 로컬저장소 생성하기

방법2. 원격저장소의 이름으로 내려받기(이름으로 폴더가 생성)

- 원격저장소 복사하기
 - git clone 원격저장소 주소
 - ex) git clone https://github.com/원격저장소 주소.git

```
MINGW64 ~/Documents/workspace  
$ git clone https://github.com/[redacted]/git-test.git
```

```
MSYS ~/Documents/workspace  
$ cd git-test/  
MSYS ~/Documents/workspace/git-test (main)  
$
```

원격저장소 생성시 작성했던 이름이 폴더명으로 생성된 것을 확인할 수 있습니다.

원격저장소 파일 내려받기

- 숨김폴더인 .git 폴더가 있는 폴더에서 명령어 실행(루트 폴더)
 - git pull origin 브랜치명
 - ex) git pull origin main

```
MSYS ~/Documents/git-test (main)  
$ git pull origin main
```

- 터미널에 git log 명령어 작성 후 HEAD 상태 확인

```
$ git log  
commit 0f30653547eb6a6f0e6150d0d5f1194c81de1a77 (HEAD -> main, origin/main, origin/HEAD)  
Author:   
Date: Tue Aug 27 09:29:00 2024 +0900  
  
pull test
```

로컬저장소와 원격저장소의 HEAD가 최신 커밋을 가리키고 있는 상태입니다.

README.md

README

- Git 저장소에서 프로젝트의 개요와 사용법을 문서화
- Markdown 언어로 작성
- 프로젝트를 보는 사람들이 해당 프로젝트의 내용을 이해할 수 있도록 설명해주는 것이 일반적
- README를 작성하는 방법에 양식은 없음

Markdown 문법

- 제목 : # 1 개~ 6 개
- 기울임 : *기울임* or _기울임_
- 굵게 : **굵게** or __굵게__
- 기울임+굵게 : ***기울임과 굵게*** or ___기울임과 굵게___
- 취소선 : ~~취소선~~
- 이모지 : :smile:, :heart:, :rocket:
 - [이모지 이름 찾기](#)
- 줄바꿈은 엔터 두번을 눌러서 공백을 만들어 주기

Markdown 문법

- 인덱싱 : 순서 있는 인덱싱은 1번부터 번호를 입력해주면 됨
- +, *, - : 순서 없는 인덱싱 만들때 사용
 - 순서있던 없던간에 TAB을 사용하여 하위 인덱싱을 제작할 수 도 있음
- 하이퍼링크 : <url주소>
- 링크 : [링크이름](url주소, 옵션)
- 이미지 : ![이미지이름](이미지url주소)
- 인용 : > 인용내용, >> 중첩인용

Markdown 문법

- 코드 : `한줄코드`
- 여러줄 코드 : ```언어명```
- 수평선 : ---, ***, _ _ _
- 체크박스 : [] 빈체크, [x] 체크
- 표 :

헤더1	헤더2	헤더3
데이터1	데이터2	데이터3
데이터4	데이터5	데이터6

Git 주요 명령어

Git 주요 명령어

- 스테이징의 작업 상태 확인
 - git status
- 커밋 확인(최신순으로 나옴)
 - git log 또는 git log --oneline --all --graph
 - --oneline : 로그 한줄로보기
 - --all : 모든 브랜치 로그 보기, 안쓰면 현재 브랜치
 - --graph : 그래프 형태로 보기
- 로컬, 원격브랜치 전체 확인하기
 - git branch(로컬만)
 - git branch -a(로컬, 원격 둘다)
 - git branch -r (원격만)

Git 주요 명령어

- 새로운 브랜치 생성
 - `git branch` 브랜치명(브랜치는 한번에 하나씩 생성)
 - ex) `git branch test`
- 브랜치로 이동
 - `git checkout` 브랜치명
 - ex) `git checkout test`
- 브랜치 생성과 브랜치로 이동을 한번에
 - `git checkout -b` 브랜치명
 - ex) `git checkout -b test`

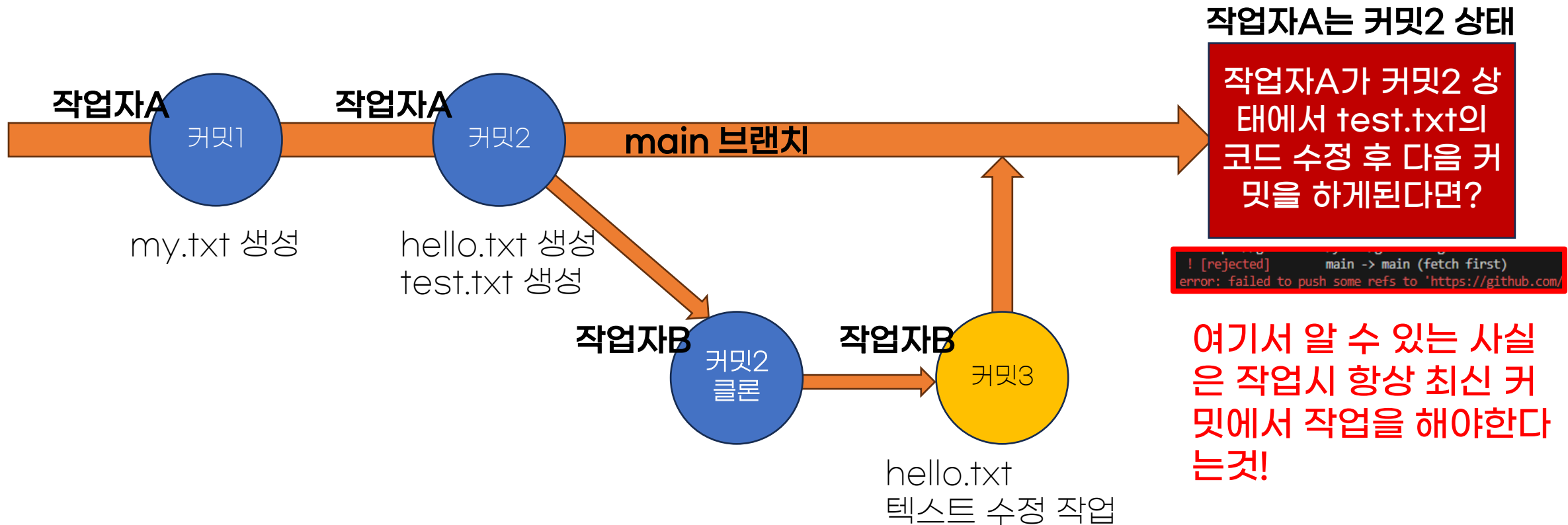
Git 주요 명령어

- 로컬브랜치 삭제(작업 중이거나 Git에 올리지 않았다면 삭제 안됨)
 - `git branch -d 브랜치명`(여러 브랜치명 가능)
 - ex) `git branch -d test hello`
- 로컬브랜치 강제 삭제(작업중여도 삭제됨. 사용시 주의)
 - `git branch -D 브랜치명`
 - ex) `git branch -D test`
- 원격브랜치 삭제하기
 - `git push origin -d 브랜치명`(여러 브랜치명 가능)
 - ex) `git push origin -d test`
- 그 밖에 명령어들은 진도를 나가면서 추가

브랜치 이해하기

브랜치를 만드는 이유

- 프로젝트 작업을 두명이서 진행한다고 가정
- 현재 우리는 main 브랜치만 존재
- main에서만 작업을 한다면?

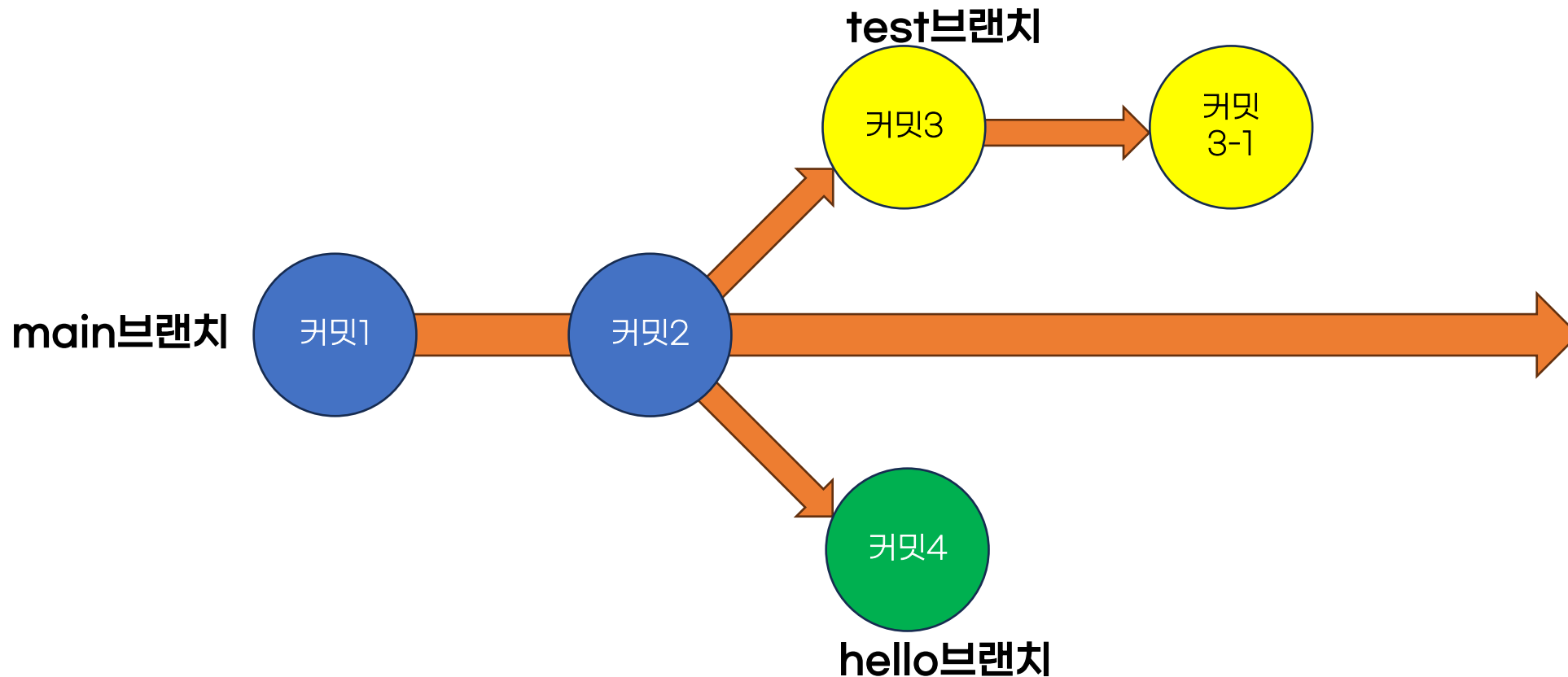


브랜치란?

- 브랜치란 저장소에서 **하나의 독립적인 작업 공간**을 뜻함
 - 우리는 지금까지 main만 사용함
- 작업 진행하면서 여러가지의 브랜치를 생성 할 수 있음
- 앞으로 브랜치에서 작업 후 다시 합치는 작업(**merge**)을 항상 해야함
- 브랜치의 역할
 - 독립적 작업 : 브랜치를 사용하면 다른 작업에 영향을 주지 않고 코드를 수정
 - 기능 분리 : 개발, 테스트, 수정 등을 기준으로 기존 코드와 분리하여 작업
 - 안전한 개발 : Git으로 배포된 메인 코드에 오류가 생기지 않도록 보호
 - 동시 작업 : 동시에 여러명의 개발자가 서로 다른 브랜치에서 작업 가능
 - 이력 관리 : 각 브랜치의 작업 이력을 관리할 수 있어서 언제, 누가 작업했는지 쉽게 추적 가능

브랜치 이해하기

- 브랜치는 나뭇가지의 뜻으로 하나의 나무에 여러 나뭇가지가 새 줄기를 생성하여 여러 갈래로 퍼지듯이 여러 데이터 흐름을 가리키며 분기라고도 함



브랜치 이해하기

- git-test 폴더에 test, hello 브랜치 생성하기

```
MINGW64 ~/Documents/git-test (main)
$ git branch test

MINGW64 ~/Documents/git-test (main)
$ git branch hello

MINGW64 ~/Documents/git-test (main)
$ git branch
hello
* main
test
```

브랜치 이해하기

- git-test 폴더의 test.txt 파일에 내용을 추가하고 커밋 후 푸쉬하기

```
MINGW64 ~/Documents/git-test (main)
$ git checkout test
Switched to branch 'test'

MINGW64 ~/Documents/git-test (test)
$ git add .

MINGW64 ~/Documents/git-test (test)
$ git commit -m "test 브랜치에서 푸쉬하기"
[test 022b548] test 브랜치에서 푸쉬하기
1 file changed, 3 insertions(+), 1 deletion(-)

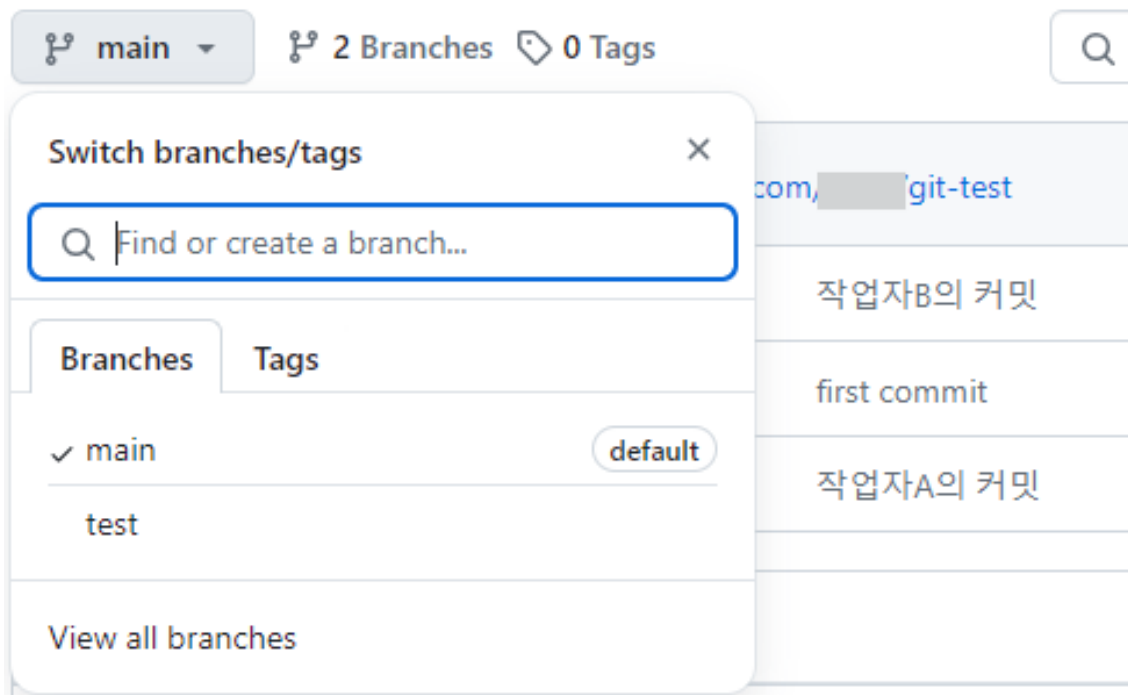
MINGW64 ~/Documents/git-test (test)
$ git push origin test
```

test 브랜치 이동

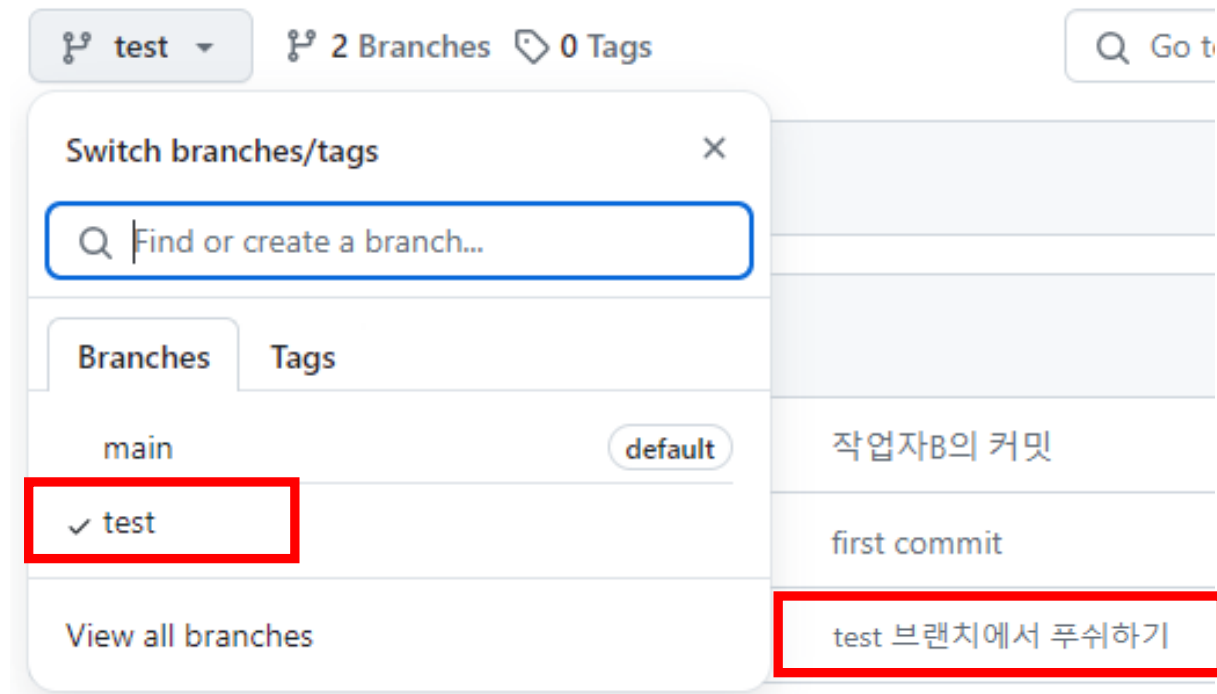
test 브랜치를 원격저장소로 푸쉬

add
commit
push
순서 외우고! 입력하기!

브랜치 이해하기



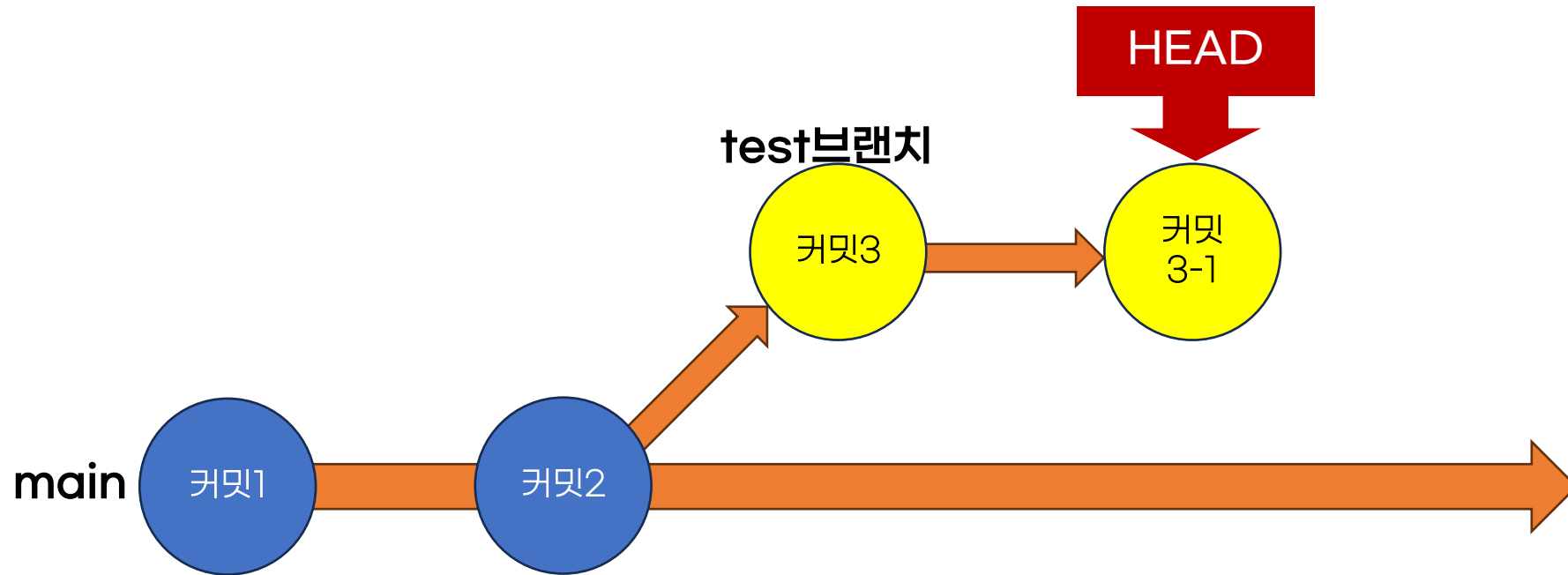
main 브랜치



test 브랜치

브랜치 이동

- 브랜치를 이동시키면서 코드를 살펴볼 수 있는 이유는 [HEAD]라는 특수한 포인터 때문. 커밋을 가리킴



로컬의 HEAD는 항상 최신의 커밋을 가리키고 있음

브랜치 이동

```

MINGW64 ~/Documents/git-test (test)
$ git log
commit 339ad3f5ef0b35ba02aa2fe59ba12282917cf5b7 (HEAD -> test, origin/test)
Author: 
Date: Tue Aug 27 17:54:13 2024 +0900

    test 브랜치에서 신규 추가 푸쉬하기

commit 022b548d5fd1003c668d8bab7cf3b4bafcd8b30
Author: 
Date: Tue Aug 27 17:49:41 2024 +0900

    test 브랜치에서 푸쉬하기

commit e2139e8d23216db3a0ddacb38744c92b8838111c (origin/main, origin/HEAD, main, hello)
Merge: 228d045 bed2312
Author: 
Date: Tue Aug 27 17:41:31 2024 +0900

    Merge branch 'main' of https://github.com/~/git-test

commit 228d0453ede414f544c2e606aa9deeee30db19ae
Author: 
Date: Tue Aug 27 17:41:12 2024 +0900

    작업자A의 커밋
  
```

- 브랜치를 생성하고 푸쉬하게 되면 로컬저장소의 HEAD는 최신 커밋을 가리킴
- 원격저장소의 origin/HEAD는 기본 브랜치인 main를 가리킴
- 새로운 브랜치를 푸쉬했다고 해서 자동으로 최신 커밋을 가리키지 않음
- 이는 새로운 브랜치는 원격저장소에 단지 추가되었을뿐 기본 브랜치에는 영향이 없기 때문
- 두 HEAD가 일치되게 하려면 병합 (merge)하면됨

브랜치 병합하기

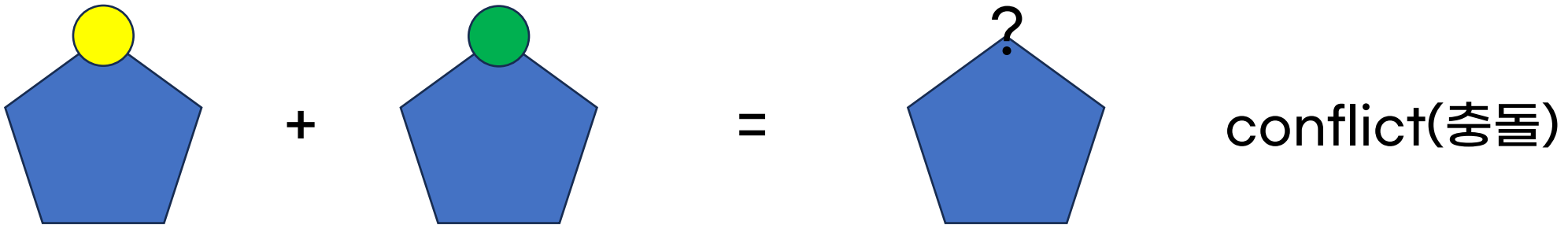
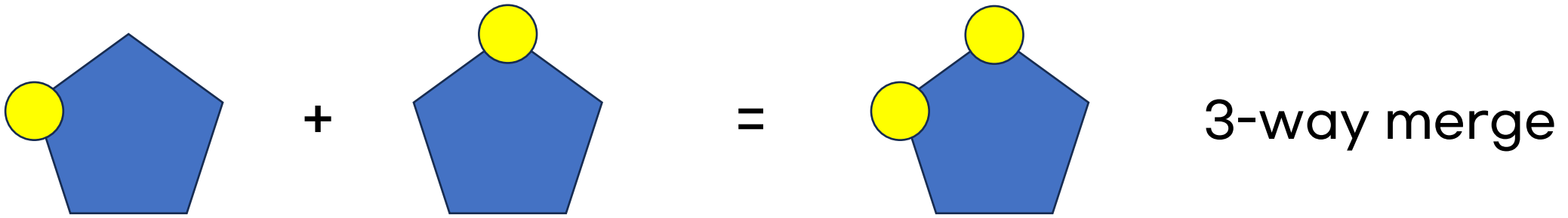
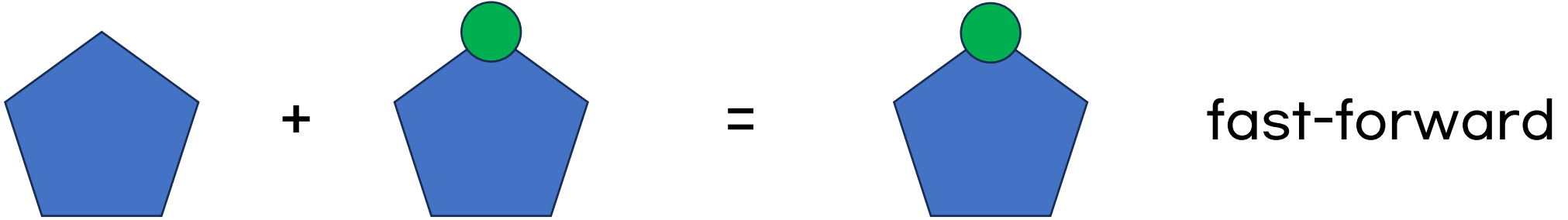
브랜치를 왜 병합해?(merge)

- 작업 통합 : 여러 분기된 브랜치들을 메인 브랜치에 통합하여 최종 결과물을 만듦
- 기능 배포 : 새로 개발한 기능이나 수정사항을 프로젝트에 반영
- 코드 일관성 : 모든 개발자의 작업을 하나로 합쳐 코드를 일관성있게 유지
- 충돌 해결 : 다른 브랜치에서 발행한 코드 충돌을 해결하고 통합
- 버전 업데이트 : 팀원들이 작업 한 부분을 모아 최종 결과물을 완성하고 프로젝트의 버전을 최신상태로 업데이트 하여 배포

병합 명령어

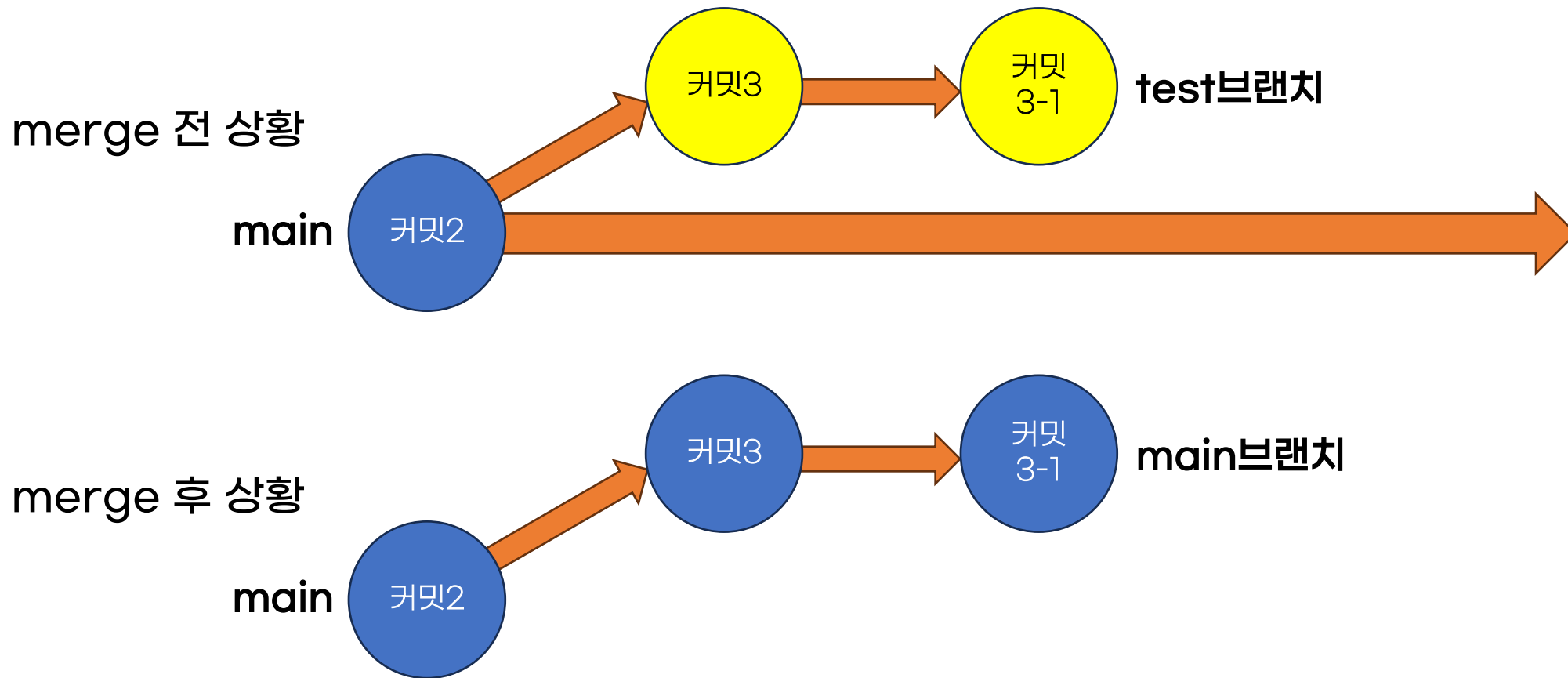
- 기준이 되는 브랜치로 이동 후
 - ex) git checkout main(메인에 합칠 경우)
- **git merge** 합쳐질 브랜치명
 - ex) git merge test
- (merge 완료 후) git push origin 기준 브랜치명
 - ex) git push origin main

병합(merge) 이해하기



fast-forward

- main브랜치에서 분기한 브랜치가 기존커밋에 영향이 없는상태에서 다시 메인으로 병합할때



fast-forward

```
MINGW64 ~/Documents/git-test (test)
$ git checkout main
Switched to branch 'main'

MINGW64 ~/Documents/git-test (main)
$ git merge test
Updating d954b5e..71b7447
Fast-forward
 branch.txt | 1 +
 test.txt   | 4 +++-
2 files changed, 4 insertions(+), 1 deletion(-)
create mode 100644 branch.txt
```

명령어 입력 후 merge에 성공하면
fast-forward merge라고 알려줌

```
MINGW64 ~/Documents/git-test (main)
$ git push origin main
```

push로 원격저장소에 반영

fast-forward

```
MINGW64 ~/Documents/git-test (main)
$ git log
commit 339ad3f5ef0b35ba02aa2fe59ba12282917cf5b7 (HEAD -> main, origin/test, origin/main, origin/HEAD, test)
Author: 
Date: Tue Aug 27 17:54:13 2024 +0900

    test 브랜치에서 신규 추가 푸쉬하기

commit 022b548d5fd1003c668d8bab7cf3b4bafcd8b30
Author: 
Date: Tue Aug 27 17:49:41 2024 +0900

    test 브랜치에서 푸쉬하기

commit e2139e8d23216db3a0ddacb38744c92b8838111c (hello)
Merge: 228d045 bed2312
Author: 
Date: Tue Aug 27 17:41:31 2024 +0900

    Merge branch 'main' of https://github.com/~/git-test

commit 228d0453ede414f544c2e606aa9deeee30db19ae
Author: 
Date: Tue Aug 27 17:41:12 2024 +0900

    작업자A의 커밋
```

병합완료 후
origin/HEAD가 최신
커밋을 가리키는 것을
확인 할 수 있음

이전 그림에서 커밋3-1

merge 완료하기

- merge 완료 한 브랜치는 삭제

```
MINGW64 ~/Documents/git-test (main)
$ git push origin -d test
To https://github.com/[redacted]/git-test.git
- [deleted]          test
```

원격저장소 브랜치 삭제

```
MINGW64 ~/Documents/git-test (main)
$ git branch -d test
Deleted branch test (was 71b7447).
```

로컬저장소 브랜치 삭제

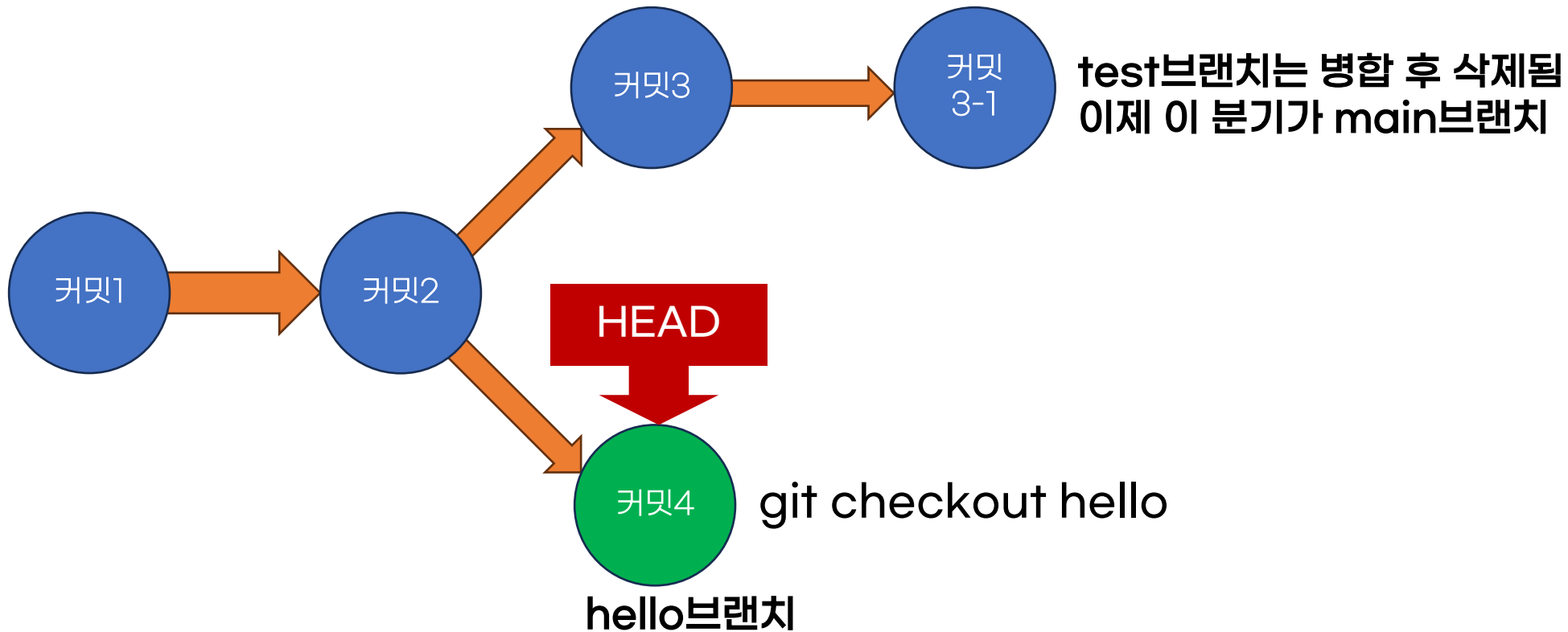
```
MINGW64 ~/Documents/git-test (main)
$ git log
commit 339ad3f5ef0b35ba02aa2fe59ba12282917cf5b7 (HEAD -> main, origin/main, origin/HEAD)
Author: [redacted]
Date: Tue Aug 27 17:54:13 2024 +0900

    test 브랜치에서 신규 추가 푸쉬하기
```

git log 확인

3-way merge

- 두 브랜치가 각각의 작업 후 하나로 합칠때
- 공통되는 조상이 동일 해야함. 여기서 커밋2는 공통 조상(Base)



3-way merge

- 브랜치 이동 후 hello.txt파일에 내용을 추가하고 커밋 후 푸쉬

```
MINGW64 ~/Documents/git-test (hello)
$ git checkout main
Switched to branch 'main'

MINGW64 ~/Documents/git-test (main)
$ git merge hello
Merge made by the 'ort' strategy.
hello.txt | 4 +++-
1 file changed, 3 insertions(+), 1 deletion(-)
```

수정된 내용을 main으로 이동 후 병합

fast-forward가 없음

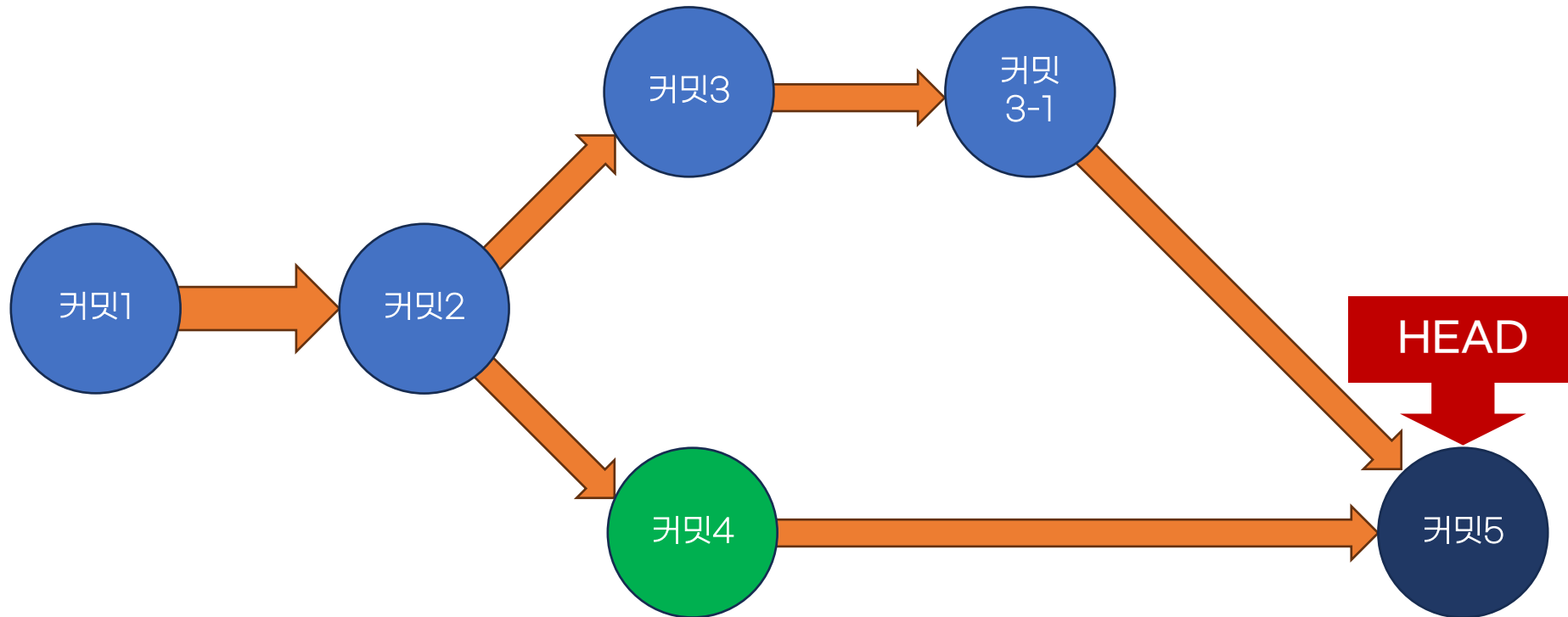
```
.git > ◆ MERGE_MSG
1 Merge branch 'hello'
2 # Please enter a commit message to explain why this merge is necessary,
3 # especially if it merges an updated upstream into a topic branch.
4 #
5 # Lines starting with '#' will be ignored, and an empty message aborts
6 # the commit.
7 hello 브랜치 merge하기
```

- git merge hello 입력시 위와 같은 탭이 열리게 됨
- 메시지 작성 후 탭을 닫게되면 merge완료
- 메시지는 작성 안해도 됨

이 메시지는 commit history에서 확인 할 수 있습니다

3-way merge

- 두 브랜치 커밋(3-1, 4)과 조상 커밋(2) 총 3개의 커밋이 merge에 관여하기 때문에 3-way라고 불리게됨
- merge완료 시 결과를 담은 커밋(5)이 생성됨(병합커밋)
- 협업에서 사용되는 merge가 3-way



실습. merge 해보기

1. 현재 main브랜치에서 dog, cat브랜치 생성
 2. dog브랜치 이동 후 my.txt 파일을 수정한뒤 git에 올리기
 3. main브랜치에서 dog브랜치 병합(dog브랜치 삭제)
 4. 병합 완료 후 cat브랜치로 이동
 5. cat브랜치에서 my.txt파일을 수정한뒤 git에 올리기
 6. main브랜치에서 cat브랜치 병합
 7. 꼭 5번째 줄을 수정!
- ⇒ cat브랜치 병합 후 무슨 메시지가 뜨는 지 확인

```
my.txt
1  안녕하세요.
2
3  저는 git을 학습중에 있습니다.
4
5  저는 강아지를 좋아합니다.
```

```
my.txt
1  안녕하세요.
2
3  저는 git을 학습중에 있습니다.
4
5  저는 고양이를 좋아합니다.
```

브랜치 충돌

브랜치 충돌 이해하기

- 두명 이상의 개발자가 **같은 파일의 같은 부분**을 작업할 때 일어나는 현상
- Git이 자동으로 병합하지 못하고 수동으로 해결해야함
- 충돌 해결 후 작업을 진행
- 오류가 아닌 코드의 충돌일 뿐

```
MINGW64 ~/Documents/git-test (main)
$ git merge cat
Auto-merging my.txt
CONFLICT (content): Merge conflict in my.txt
Automatic merge failed; fix conflicts and then commit the result.
```

충돌이 났을때는 수동으로 충돌 해결 후 진행하면 됩니다

브랜치 충돌 해결하기

my.txt

```
1  안녕하세요.
2
3  저는 git을 학습중에 있습니다.
4
5  Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
6  <<<<<< HEAD (Current Change)
7  저는 강아지를 좋아합니다.
8  =====
9  >>>>>> cat (Incoming Change)
10 저는 고양이를 좋아합니다.
```

ctrl + z (되돌리기) 해줄 수 있으니
자유롭게 눌러보셔도 됩니다

- Accept Current Change : 위 코드만 남김
- Accept Incoming Change : 아래 코드만 남김
- Accept Both Changes : 위,아래 코드 남김
- Compare Changes : 새 탭에서 두 코드 보여줌

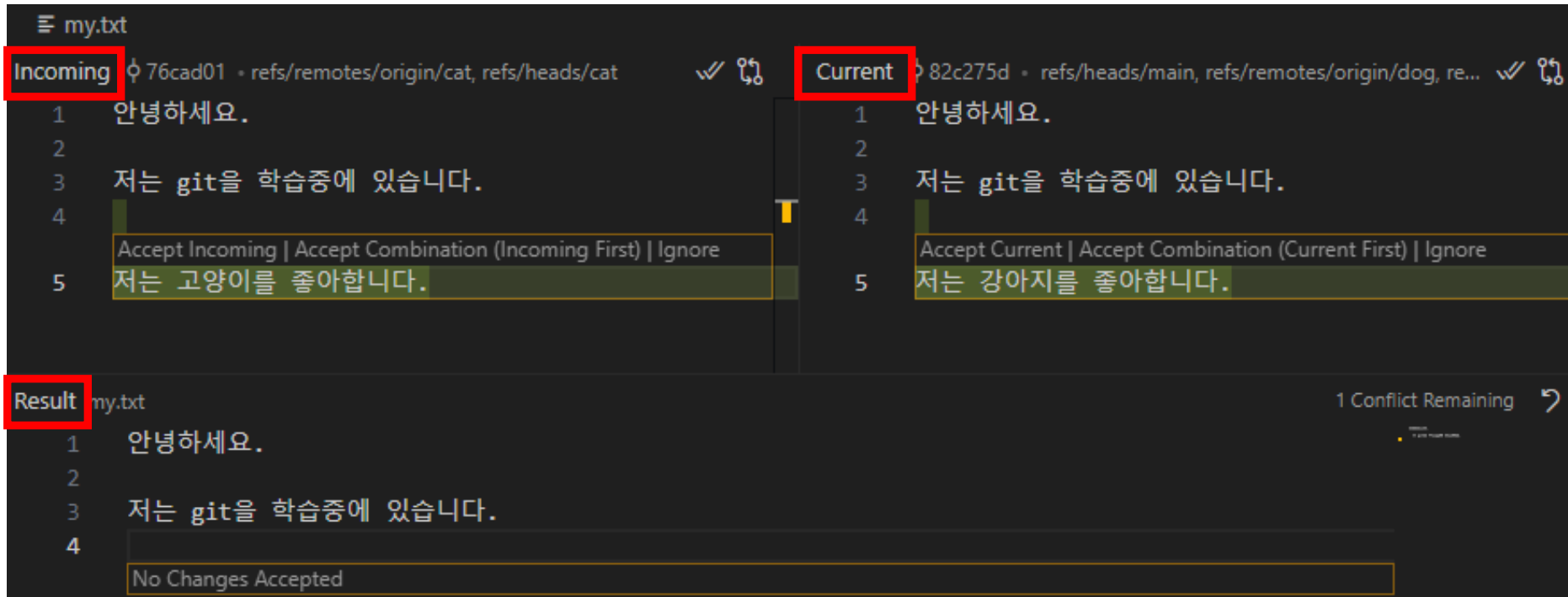
브랜치 충돌 해결하기

- 방법1 : 원하는 코드만 남기기(코드는 둘다 남긴다고 가정)
<<<<< HEAD, =====, >>>>> cat 을 수동으로 삭제 후 저장

```
my.txt
1  안녕하세요.
2
3  저는 git을 학습중에 있습니다.
4
5  저는 강아지를 좋아합니다.
6  저는 고양이를 좋아합니다.
```

브랜치 충돌 해결하기

- 방법2 : Merge Editor이용

[Resolve in Merge Editor](#)

위 두개 분할 된 화면의 Incoming과 Current를 수정하면 하단 Result에 반영 됨

- 모두 완료 후 Complete Merge 버튼 클릭

[Complete Merge](#)

브랜치 충돌 해결하기

- 모두 완료 후 Git에 add, commit, push
- 이 후 모든 브랜치 삭제

```
MINGW64 ~/Documents/git-test (main|MERGING)
$ git add .

MINGW64 ~/Documents/git-test (main|MERGING)
$ git commit -m "충돌 해결"
[main 78389ce] 충돌 해결

MINGW64 ~/Documents/git-test (main)
$ git push origin main
```

브랜치 이름 규칙

브랜치 이름 규칙

- 일반적인 규칙들(회사마다 다를 수 있음)
- main브랜치에서는 직접 작업하거나 커밋하지 않음
- 브랜치 명명 규칙

feature	기능개발	feature/기능명	신규 기능 개발
hotfix	수정	hotfix-1.0.1	버그 수정. 1.0 버전에 첫번째
release	배포	release-1.0	배포 1.0
develop	개발테스트	develop	배포 전 개발 테스트 용

- 하나의 브랜치에는 하나의 작업!!

버전?

- 버전 이름이 8.1.5 일때
- 첫번째 숫자 8은 코드에 많은 변화가 있을때 사용
 - 예) 대규모 업데이트
- 두번째 숫자 1은 새로운 기능이 추가 또는 업그레이드 되었을때 사용
 - 예) 00 기능이 변경
- 세번째 숫자 5는 패치 혹은 버그 수정시 사용
 - 예) 로그인 오류 해결

학습정리

- 폴더구조에서는 루트 폴더가 중요하다. 앞으로 모든 프로젝트는 루트 폴더에서 시작하며 루트 폴더와 Git저장소가 연결된다.
- github에 원격저장소는 원하는대로 생성이 가능하다.
- 원격저장소를 내 로컬환경에 가져오기 위해서는 clone을 해줘야 한다.
- 원격저장소에 코드를 올리는 방법은 add, commit, push이고 원격저장소에서 코드를 내려받는 방법은 pull이다.

학습정리

- 브랜치는 내가 작업할 수 있는 하나의 작업 공간이며 작업시 main 에서 작업하는것이 아닌 브랜치를 생성 후 작업한다.
- 작업이 완료 된 브랜치는 병합을 진행하여 코드를 최신 상태로 해 놓아야 한다.
- 두명 이상이 같은 파일의 같은 코드를 수정하면 충돌이 일어나게 되며 충돌을 해결 후 작업을 계속 진행하면 된다.
- 충돌은 코드의 오류가 아닌 단순히 같은 코드를 수정했을 뿐이다.

다음 수업은?

- 지금까지 혼자 코드를 수정해보고 Git에 올려보았습니다.
- 다음시간에는 팀에서 협업을 진행할 때 Git을 어떻게 사용하는지 알아보겠습니다.

복.습.철.저

수고하셨습니다