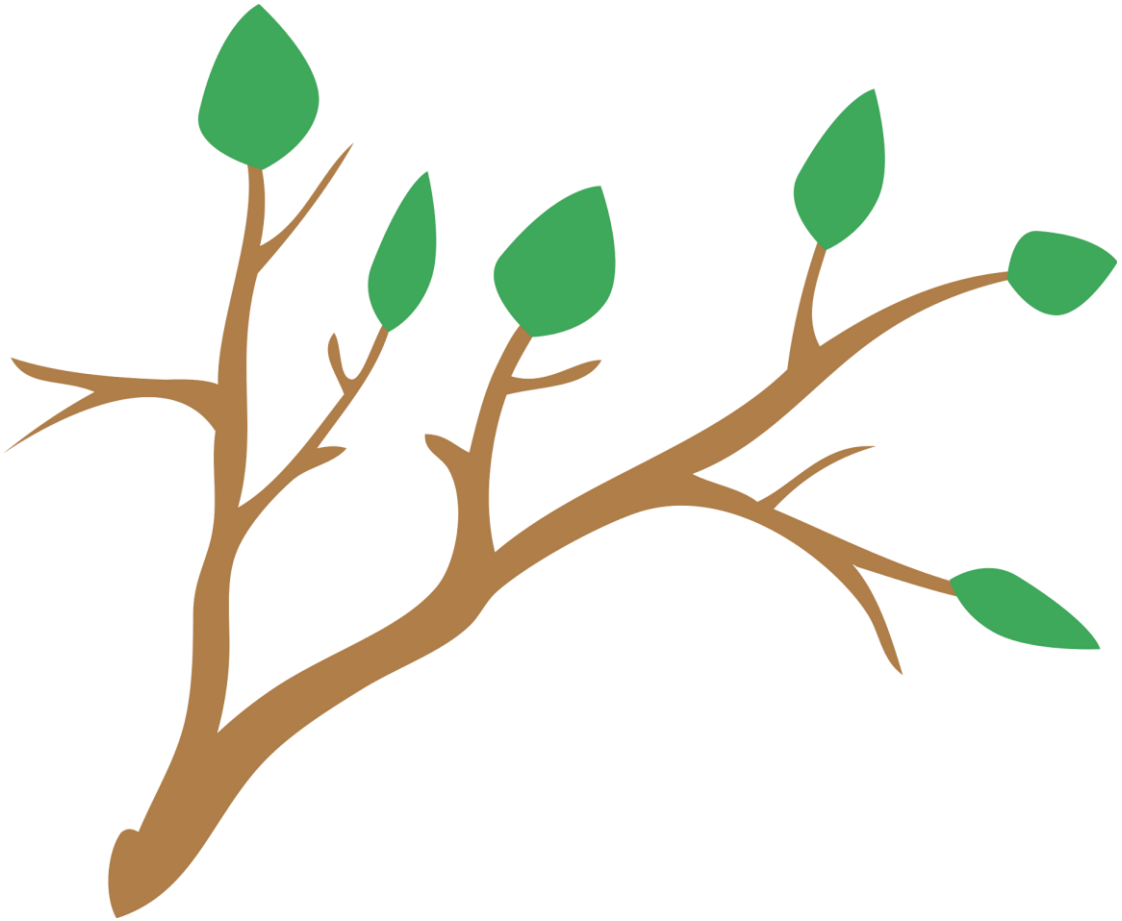
codingOn

codingOn

브랜치 이해하기

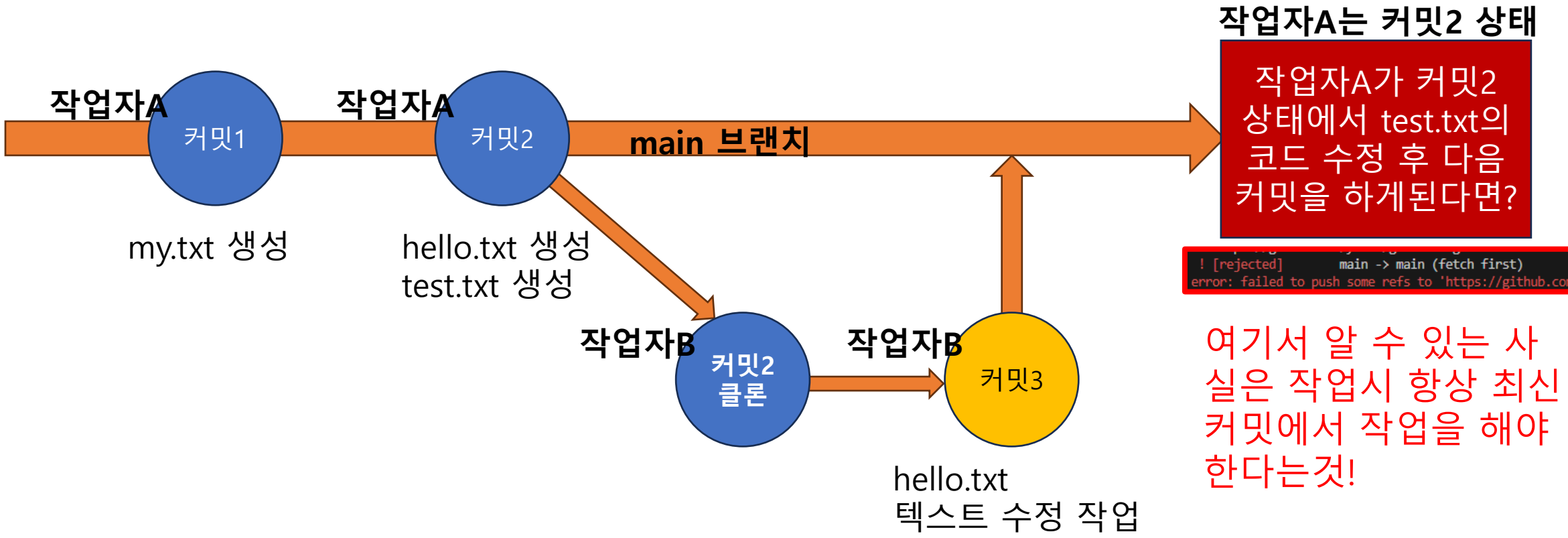
Branch?

- 나뭇가지
- 분기
- (회사 따위의) 지사
- 둘 이상으로 갈라지다



브랜치를 만드는 이유

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



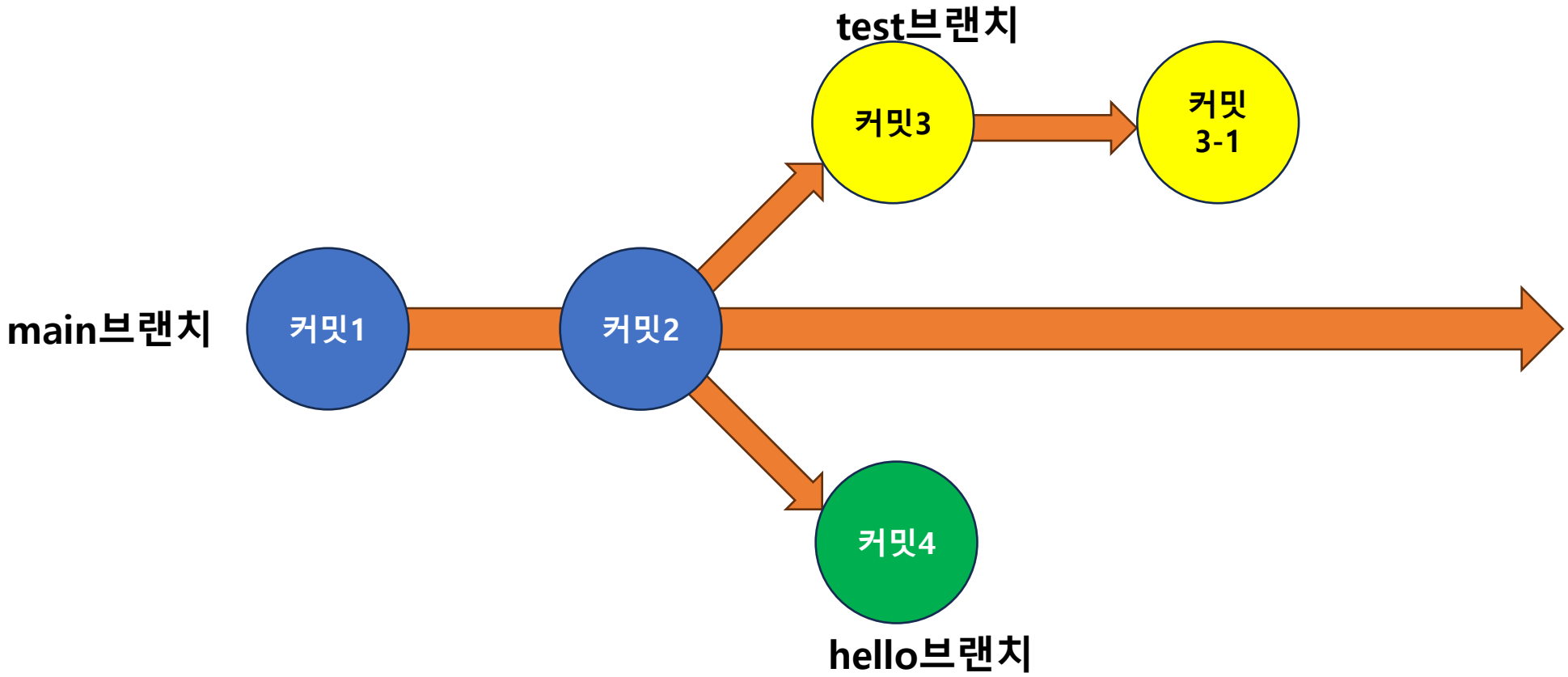
여기서 알 수 있는 사실은 작업시 항상 최신 커밋에서 작업을 해야 한다는것!

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 - 명령어

codingOn

- 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
- 로컬, 원격브랜치 전체 확인하기
  - git branch(로컬만)
  - git branch –a(로컬, 원격 둘다)
  - git branch –r (원격만)

브랜치 이해하기 - 명령어

codingOn

- 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

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.txt파일에 내용을 추가하고 커밋 후 푸쉬하기

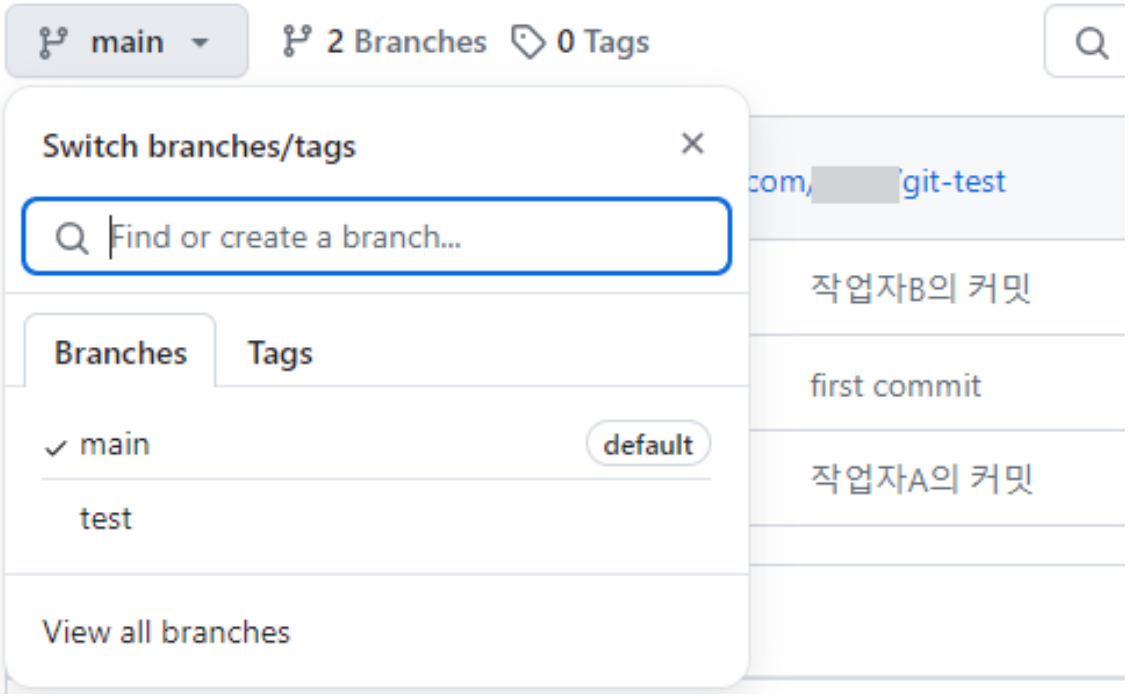
```
MINGW64 ~/Documents/git-test (main)
$ git checkout test
Switched to branch 'test'

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

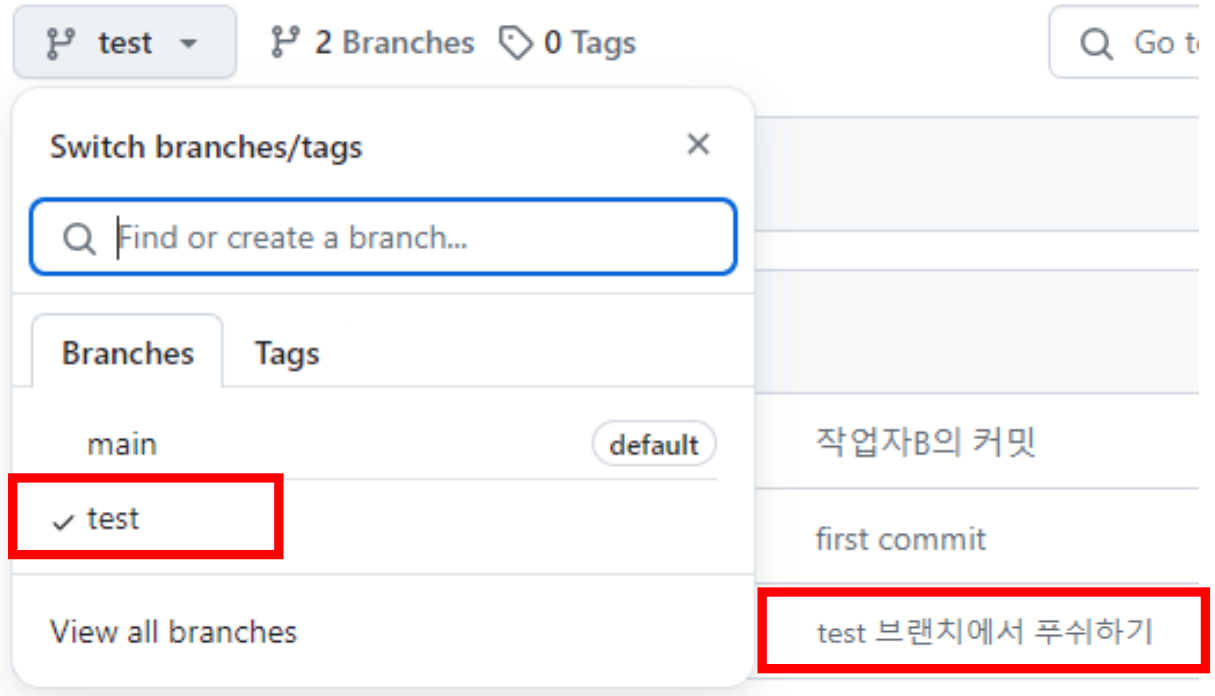
test 브랜치를 원격저장소로 푸쉬

add
commit
push
순서 외우고! 입력하기!

브랜치 이해하기



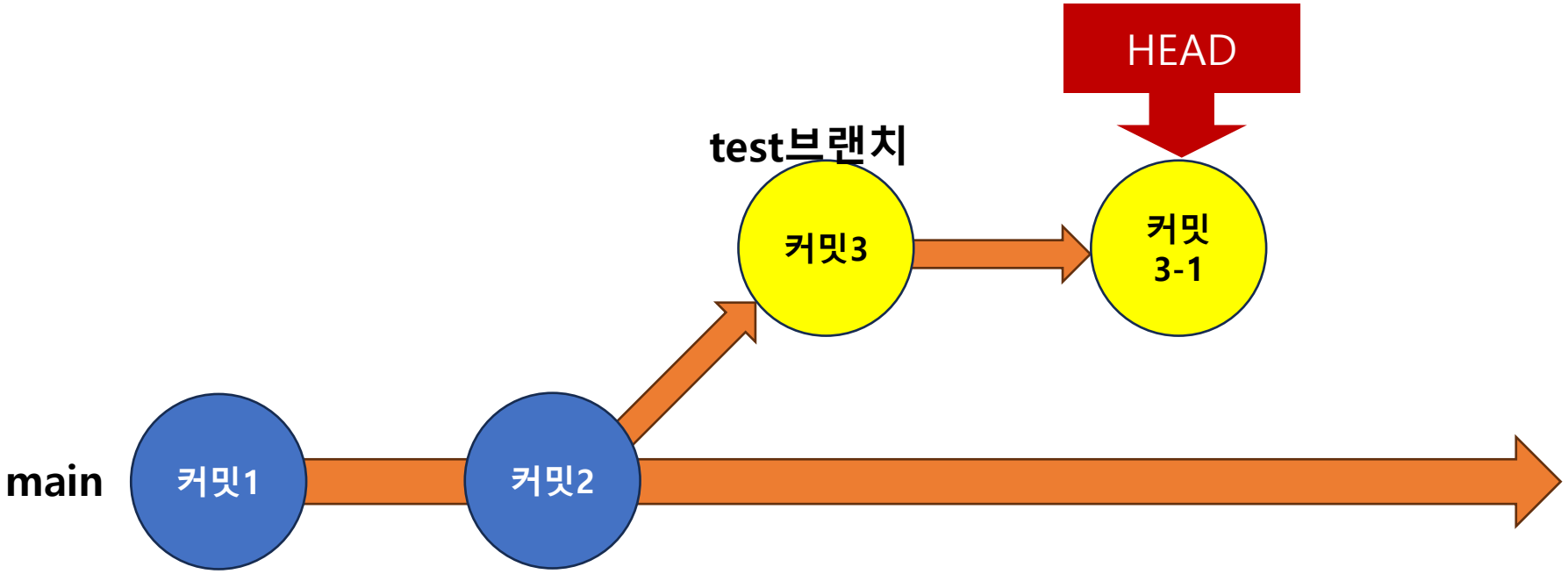
main 브랜치



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 022b548d5fd1003c668d8bab7cf3b4bafcdd8b30
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

브랜치을 왜 병합해?(merge)

- 작업 통합 : 여러 분기된 브랜치들을 메인 브랜치에 통합하여 최종 결과물을 만듬
- 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

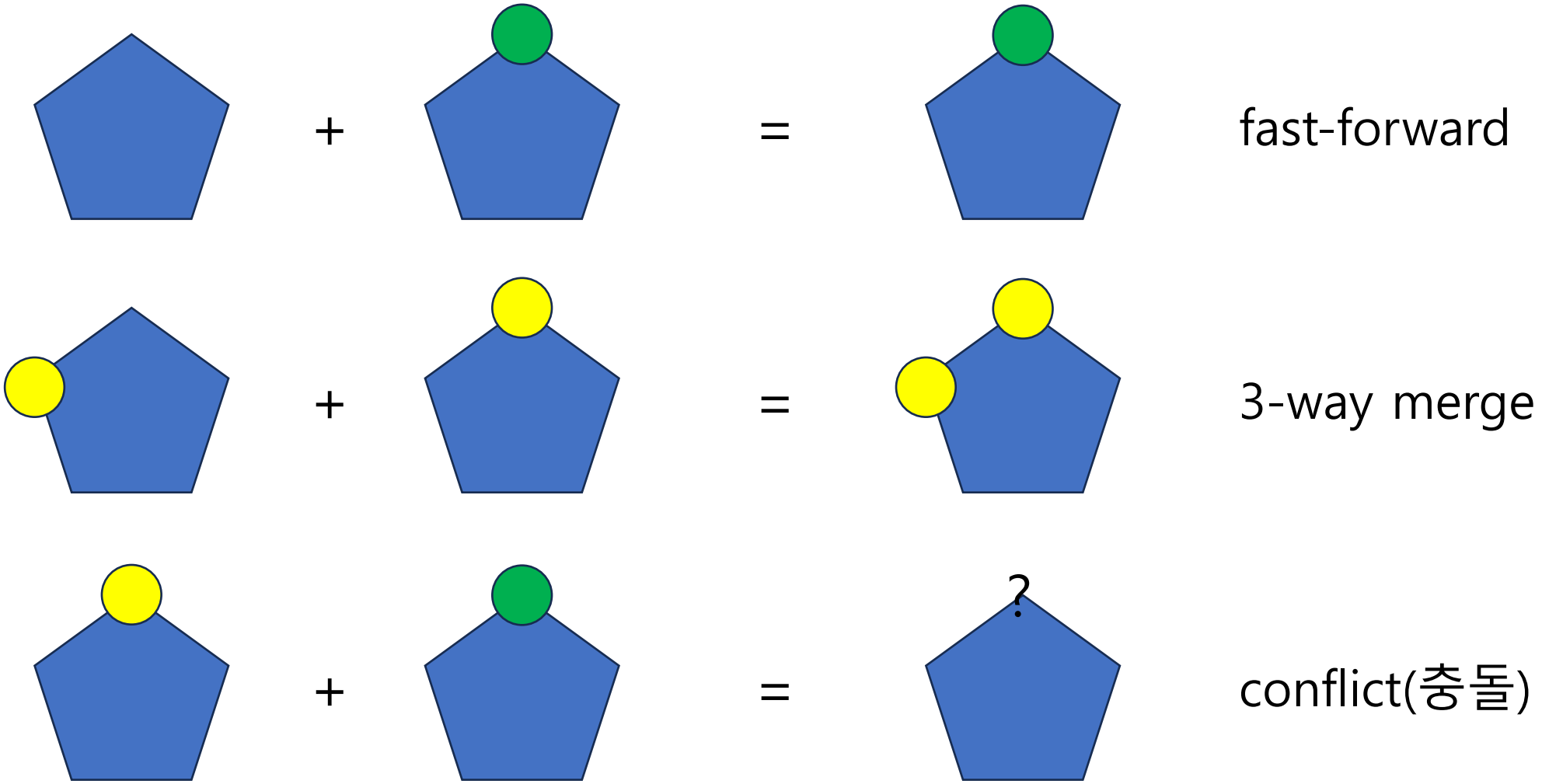
병합 명령어

codingOn

- 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

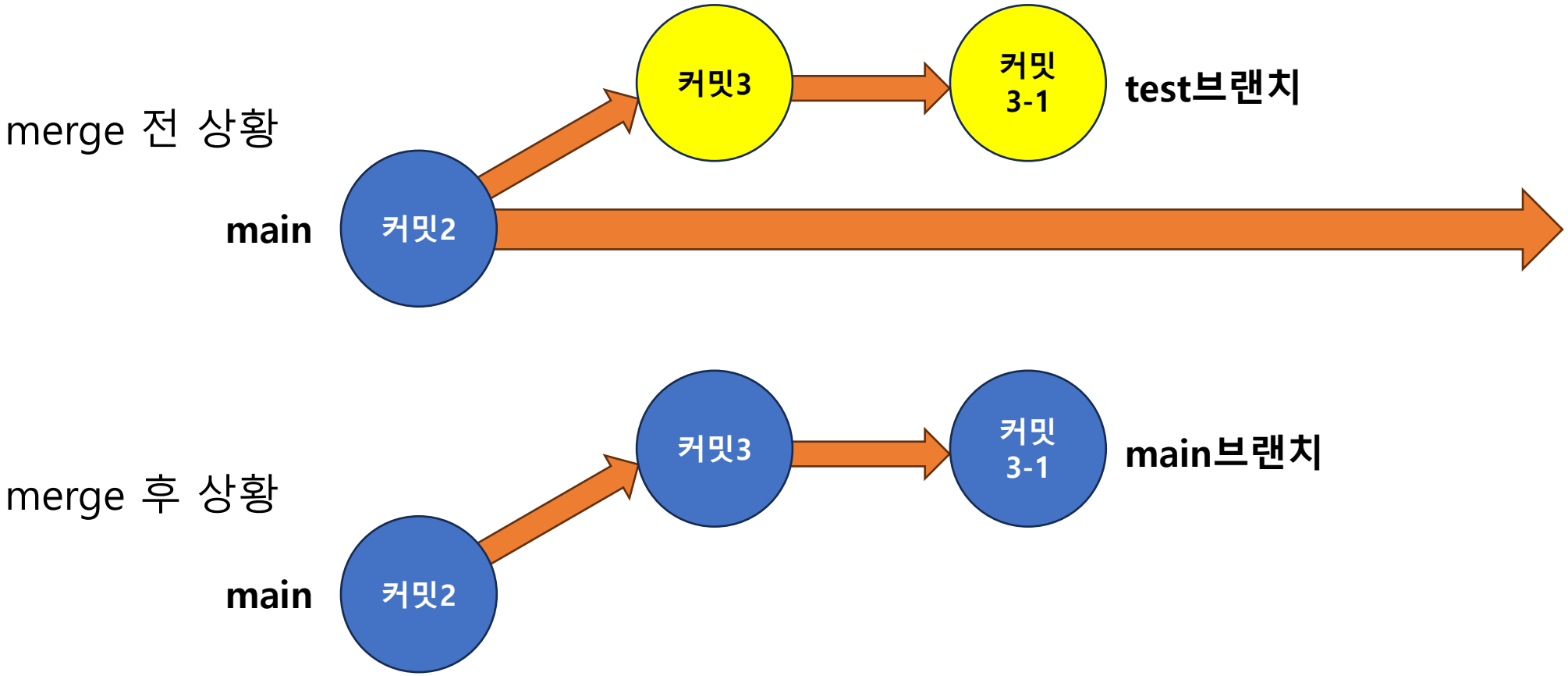
병합(merge) 이해하기

codingOn



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

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

    test 브랜치에서 신규 추가 푸쉬하기

commit 022b548d5fd1003c668d8bab7cf3b4bafcdd8b30
Author: [redacted]
Date:   Tue Aug 27 17:49:41 2024 +0900

    test 브랜치에서 푸쉬하기

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

    Merge branch 'main' of https://github.com/[redacted]/git-test

commit 228d0453ede414f544c2e606aa9deeee30db19ae
Author: [redacted]
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

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

원격저장소 브랜치 삭제

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

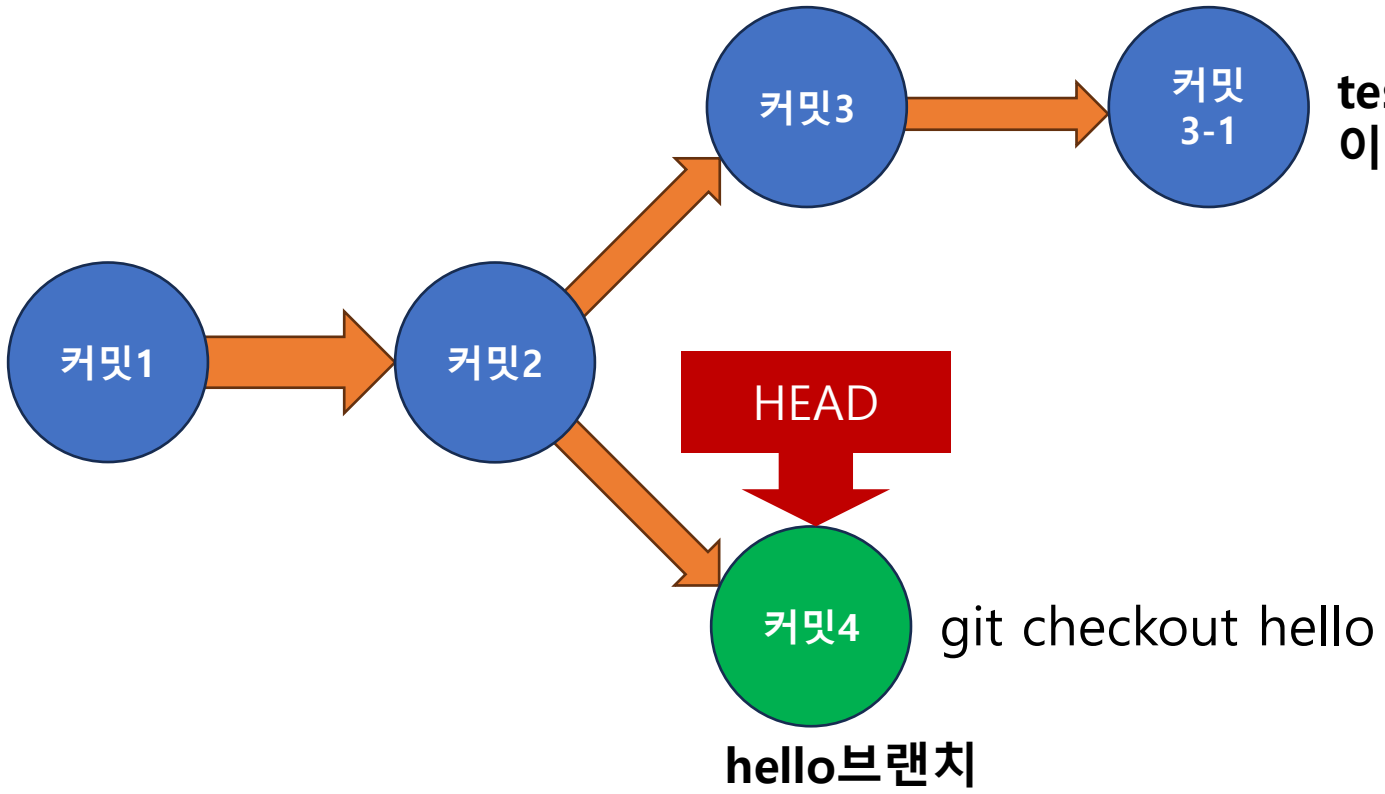
로컬저장소 브랜치 삭제

```
[redacted] 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

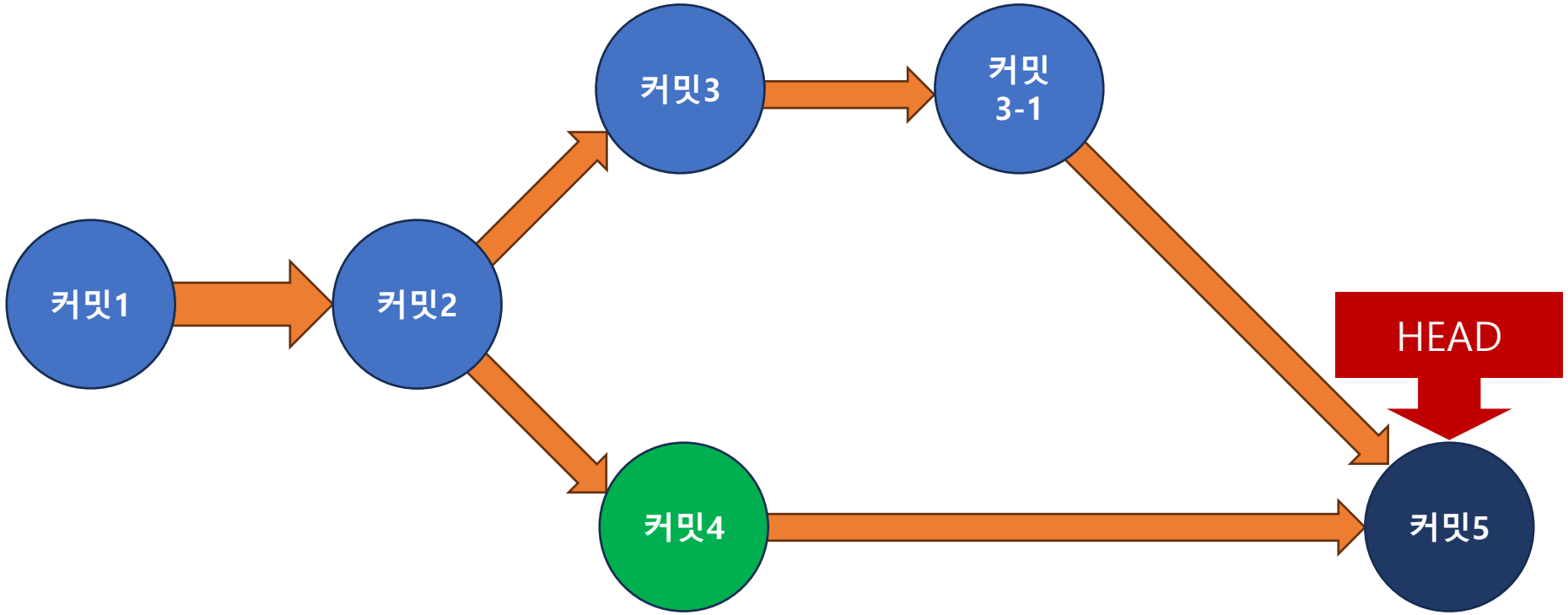
```
1  Merge branch 'hello'
2  # Please enter a commit message to explain why this merge is necessary,
3  # especially if it merges an updated upstream into a topic branch.
4  #
5  # Lines starting with '#' will be ignored, and an empty message aborts
6  # the commit.
7  hello 브랜치 merge하기
```

- 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

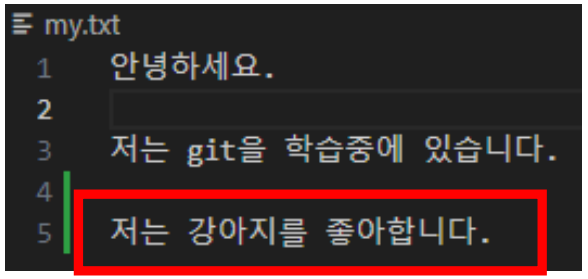
- 두 브랜치 커밋(3-1, 4)과 조상 커밋(2) 총 3개의 커밋이 merge에 관여하기 때문에 3-way라고 불리게됨
- merge완료 시 결과를 담은 커밋(5)이 생성됨(병합커밋)
- 협업에서 사용되는 merge가 3-way

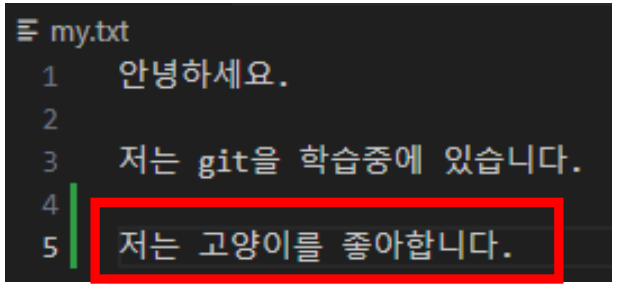


실습. merge 해보기

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





codingOn

브랜치 충돌

브랜치 충돌 이해하기

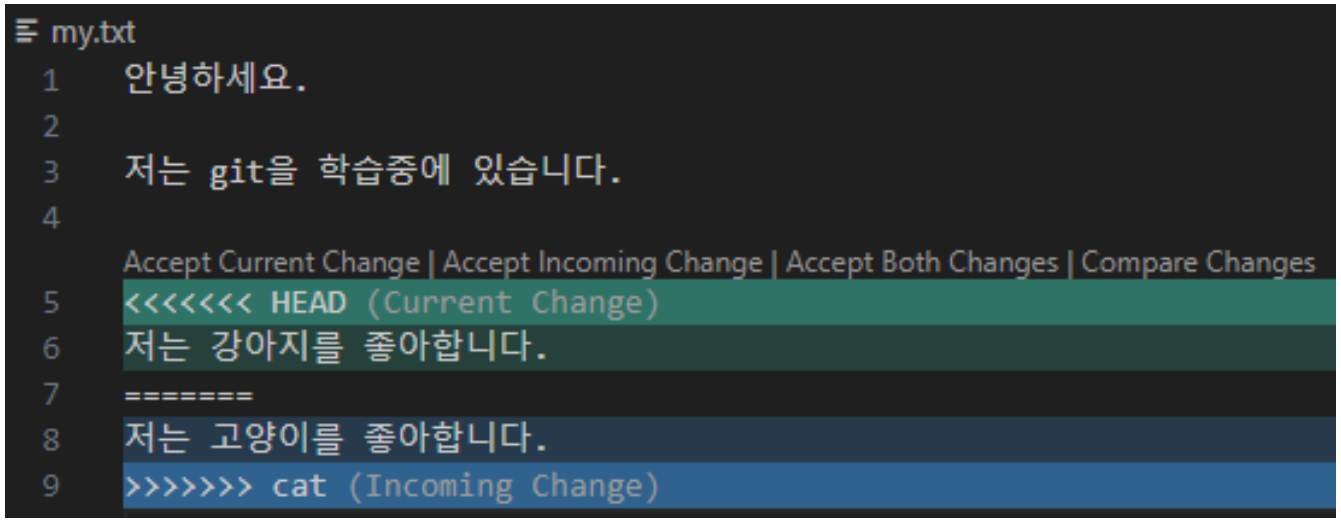
- 두명 이상의 개발자가 같은 파일의 같은 부분을 작업할 때 일어나는 현상
- Git이 자동으로 병합하지 못하고 수동으로 해결해야함
- 충돌 해결 후 작업을 진행
- 오류가 아닌 코드의 충돌일 뿐

```
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

codingOn



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
- Compage Chages : 새 탭에서 두 코드 보여줌

<<< 특정 Tool 에서 보일 때

브랜치 충돌 해결하기

- 방법 : 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

- 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

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

codingOn

Commit 되돌리기

Commit 되돌리기

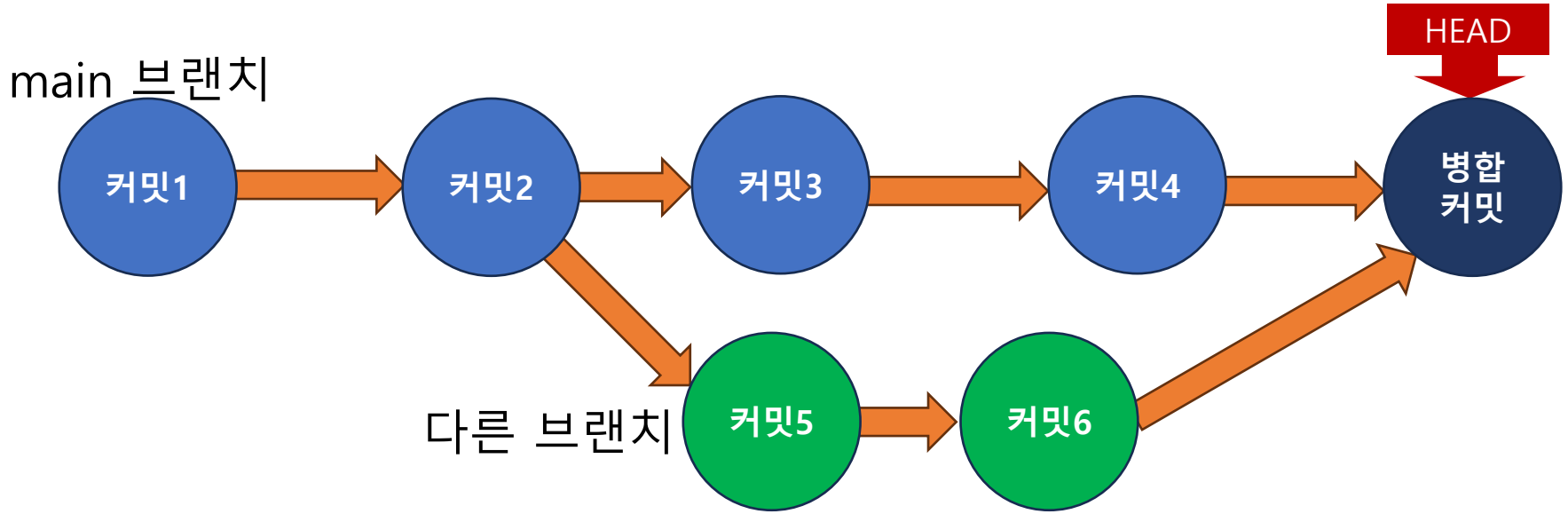
codingOn

- Commit 되돌리기는 Git이 익숙해지기 전까지 사용안해도 무관
- 자주 사용 안할 수도 있는 명령어
- Git의 커밋을 이전상태로 되돌릴 때 사용
- 코드가 모두 삭제될 수도 있으므로 사용시 주의 해야함

reset

- git reset (옵션) <이동할 커밋>
  - 예) git reset --hard HEAD^2
- 이동할 커밋
  - HEAD~n : n번째 조상 커밋으로 이동
  - HEAD^n : 병합커밋에서 사용하며 n번째 부모로 이동
- 옵션값
  - --soft : 스테이징에는 올라가 있으며 커밋만 다시 하고 싶을때
  - --mixed(기본값) : 스테이징, 커밋을 모두 취소하고 다시 하고 싶을때
  - --hard : 최근 변경사항을 모두 삭제하고 이전 커밋으로 돌아가고싶을때
- 협업시에는 사용을 안하는게 좋음

reset

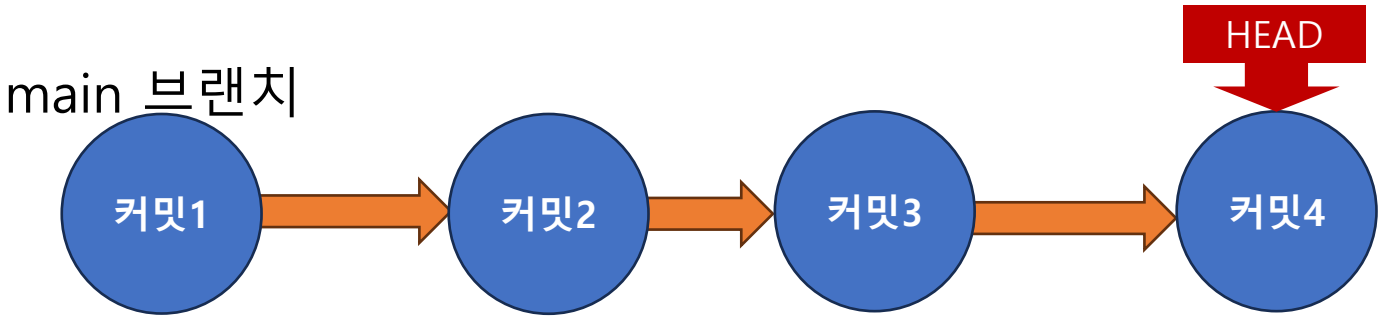


이동 명령어 예시
HEAD^1 : 병합커밋의 첫번째부모로 main 브랜치의 커밋4와 병합된 것이므로 커밋4
HEAD^2 : 병합커밋의 두번째 부모로 다른 브랜치의 커밋6과 병합된 것이므로 커밋6
HEAD~1 : 이전 커밋인 부모 커밋4
HEAD~2 : 두번째 조상 커밋인 커밋3
HEAD~3 : 세번째 조상 커밋인 커밋2

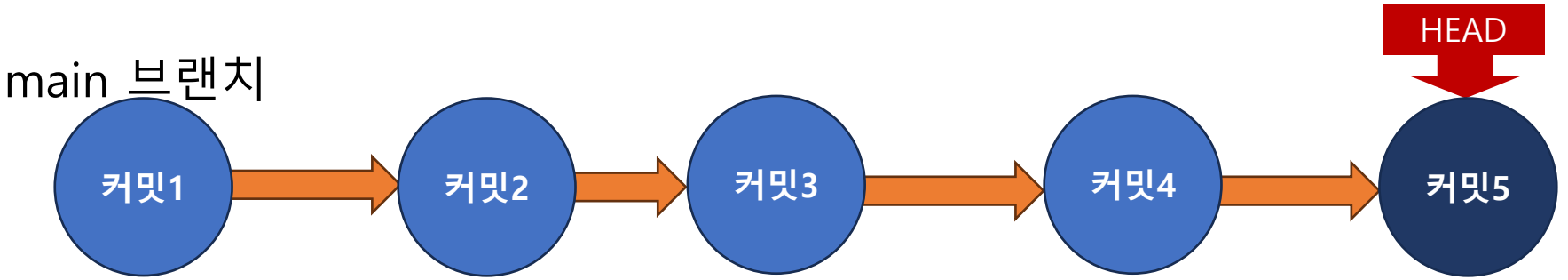
revert

- git revert <이동할 커밋>
- 특정 커밋을 취소하는 새로운 커밋 생성
- 기존 커밋 이력을 유지하면서 변경 사항을 되돌리는 새로운 커밋 생성
- 특정 커밋을 기준으로 되돌리며 필요시 여러커밋을 순차적으로 되돌리기
- 사용시에는 최근 커밋부터 되돌리기
- reset과의 차이점
  - reset은 커밋이력을 삭제하면서 이전으로 되돌림
  - revert는 기존 커밋이력을 남기고 변경된 새로운 커밋을 생성
- 협업에는 reset보다는 revert를 권장

revert



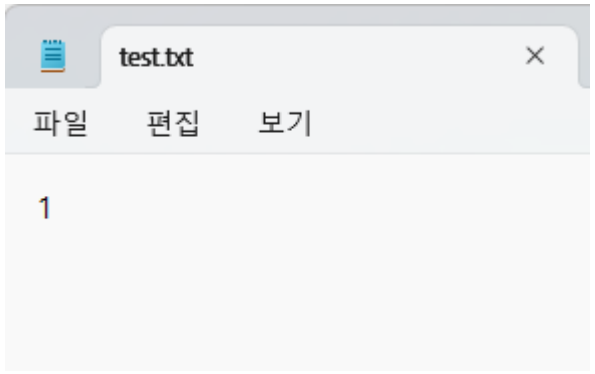
커밋3이 잘못되어 되돌린다면
=> git revert 커밋3



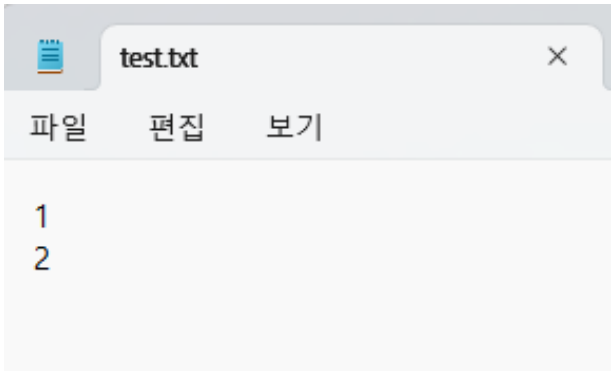
커밋5는 커밋3을 되돌린 커밋

실습. Reset, Revert

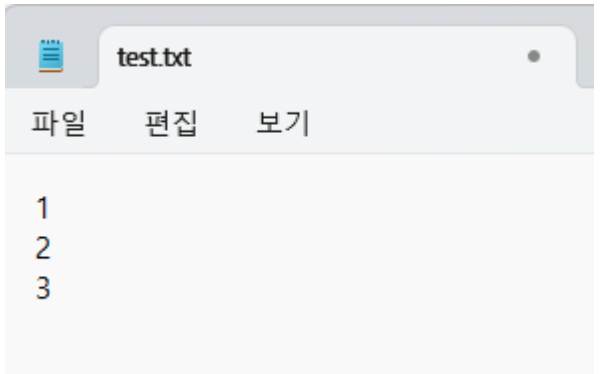
1. 아래와 같이 텍스트 파일의 내용을 변경하며 커밋 3회 수행



commit 1



commit 2



commit 3

실습. Reset, Revert

2. Revert를 사용하여 commit 1 상태로 되돌리기
3. Reset을 사용하여 commit 3 상태로 되돌리기
4. Revert를 사용하여 commit 2 상태로 되돌리기
5. git log --oneline 으로 커밋 상태를 확인하고 스크린샷으로 찍어 노션 댓글에 올리기