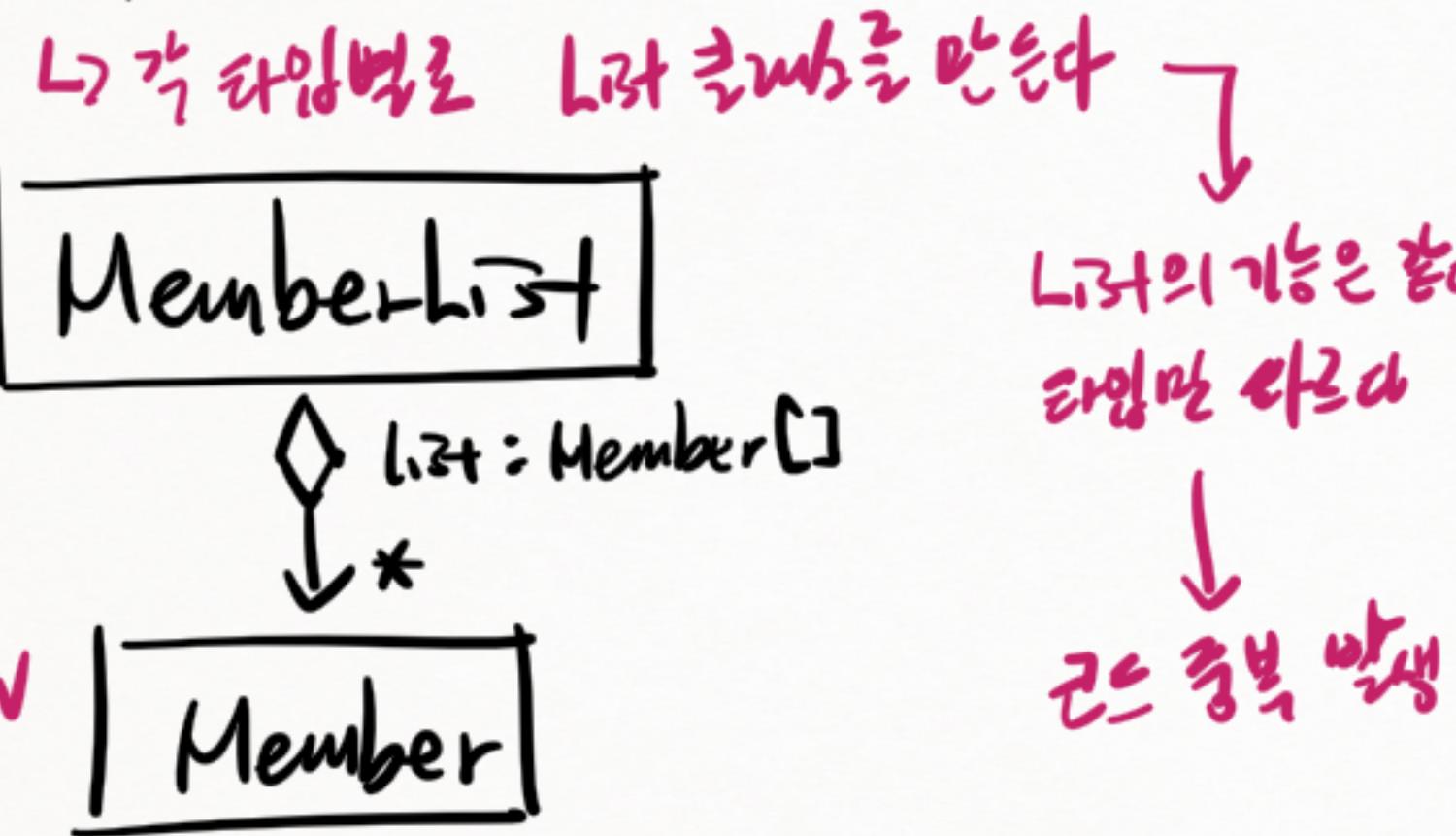


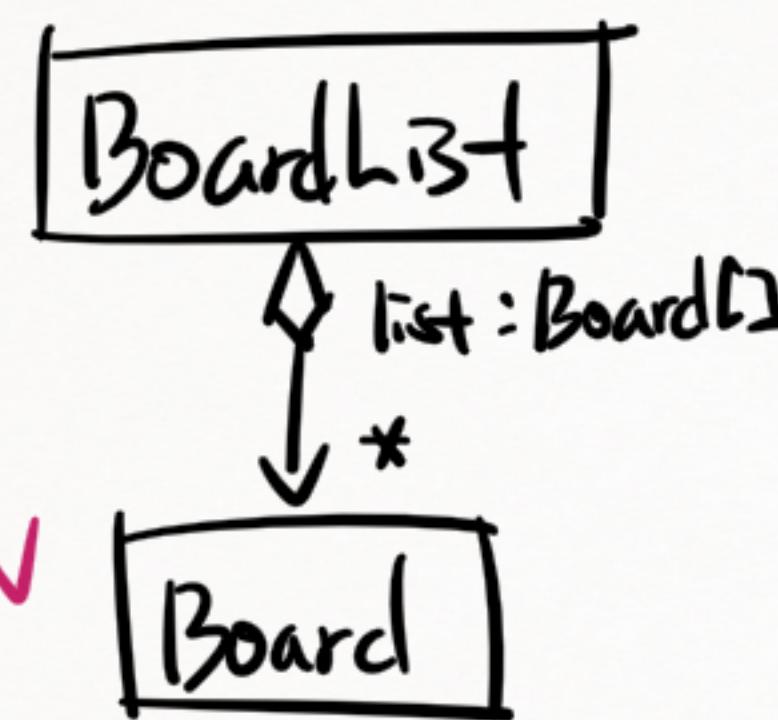
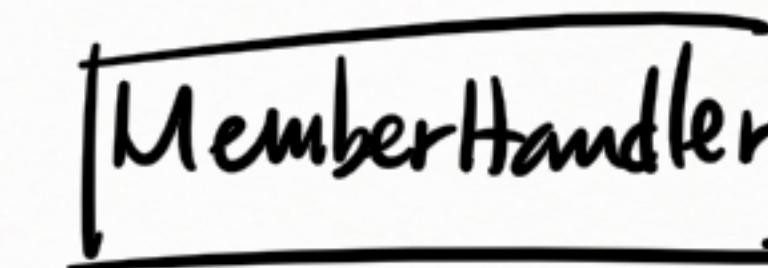
24. 제네리ك (Generic) 적용 : (object 타입처럼 다양한 타입에 대응할 수 있다.)  
특정타입으로 제한할 수 있다.

① 다형적 변수 적용 전



② 다형적 변수 적용

→ 미처 각 타입별로 클래스를 정의한 듯한  
효과를 볼 수 있다.



\* 단점 :

- ① 인스턴스를 만들어야 하다  
형변환 필요!
- ② 특정타입별로 다수로  
제한할 수 있다.

← 타입별로  
List를 만들 필요가  
없다!  
← 타입 안정성을 해친다.

. Member  
. Board  
:  
: {  
} 다양한 타입의  
인스턴스(주)를  
생성할 수 있다.

\* Generic  $\Rightarrow$  Type Parameter

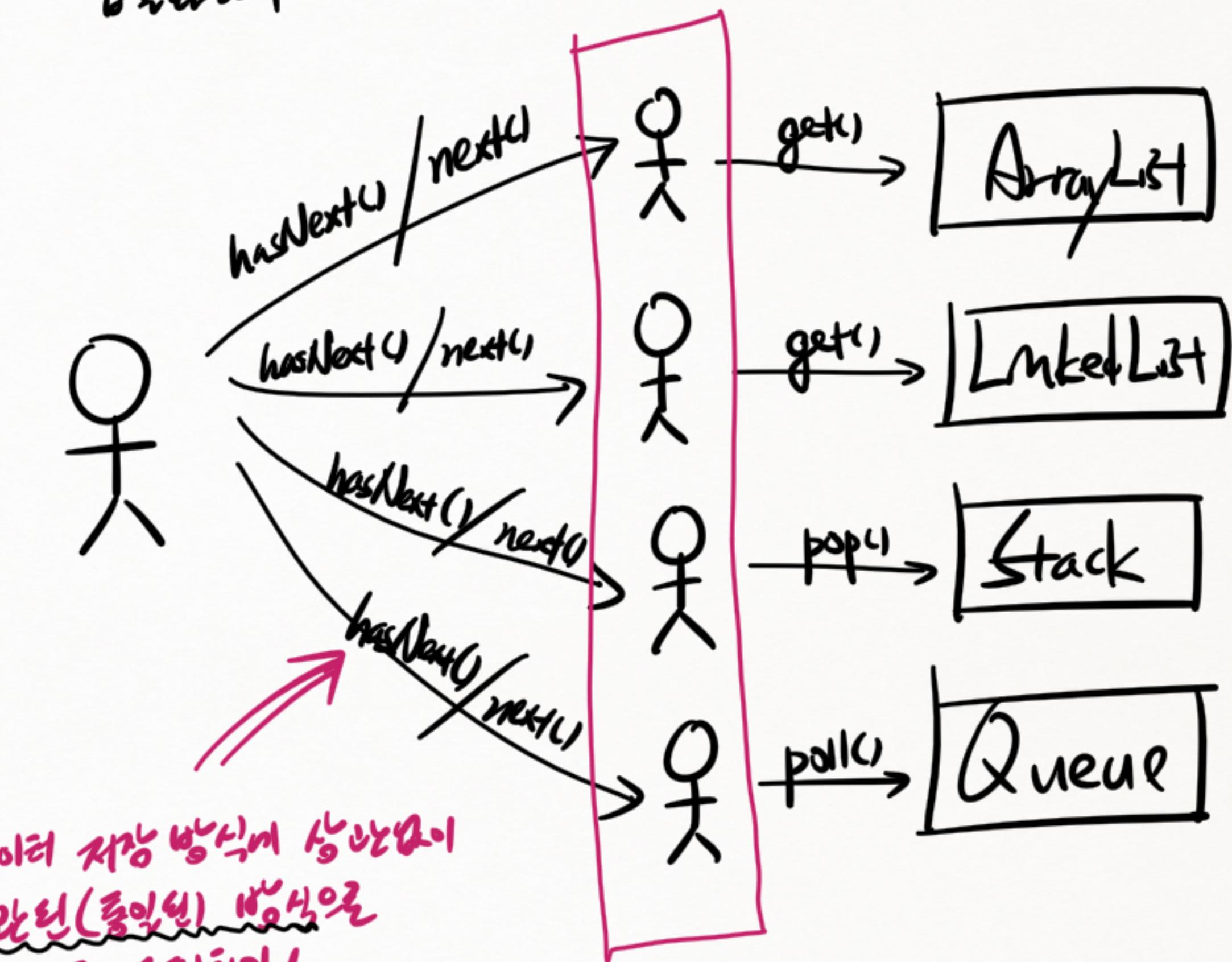
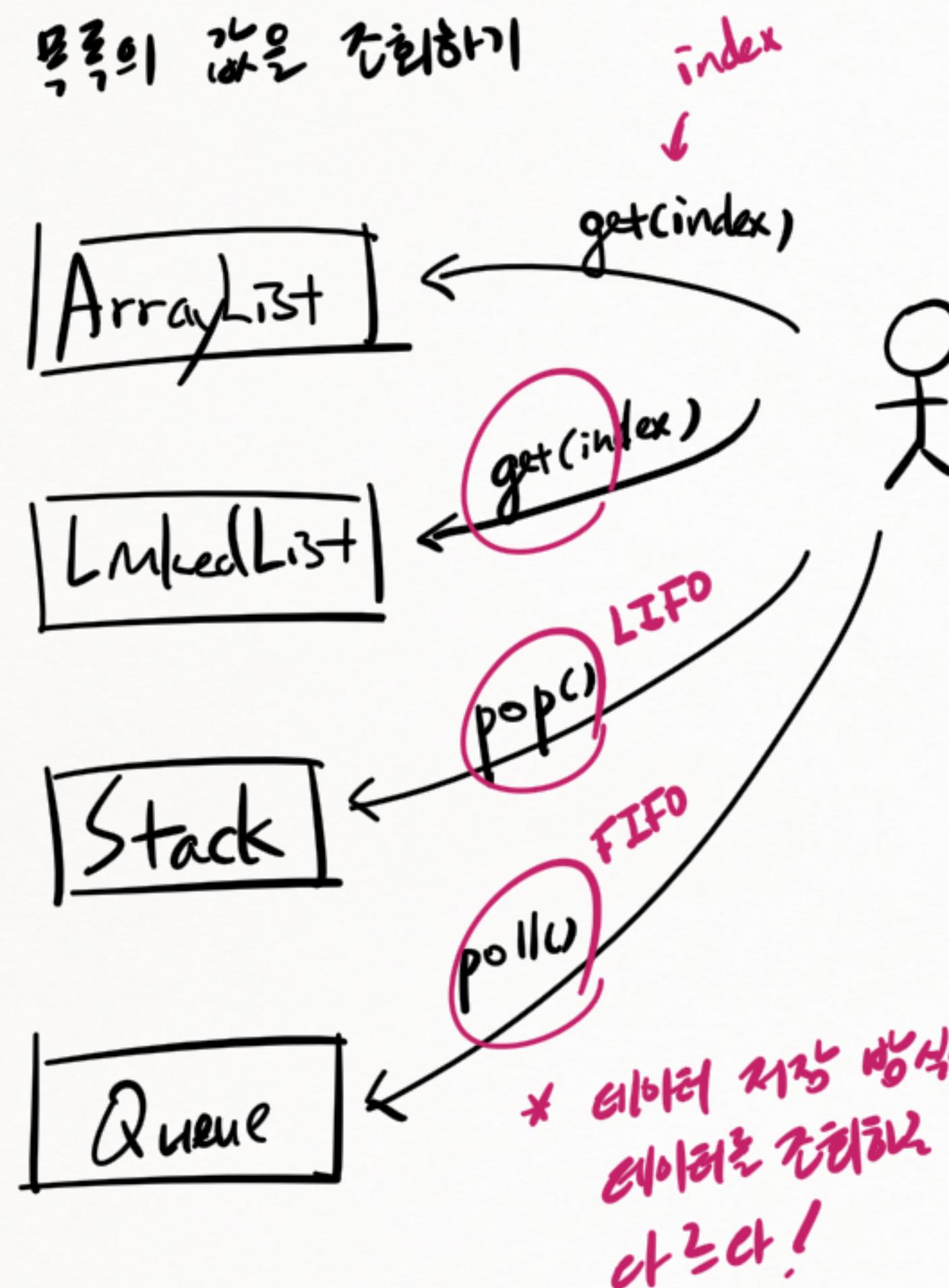
클래스의 타입 변수를 넣는다

```
class ArrayList<What> {  
    public void add(What obj) {  
        =  
    }  
    public What get(int index) {  
        =  
    }  
    :  
}
```

타입 이름을 넣을 변수 = "Type Parameter"

## 25. Iterator 대신을 활용하여 iterator 처리기능을 기존에 쓰던화하기 →변환화하기

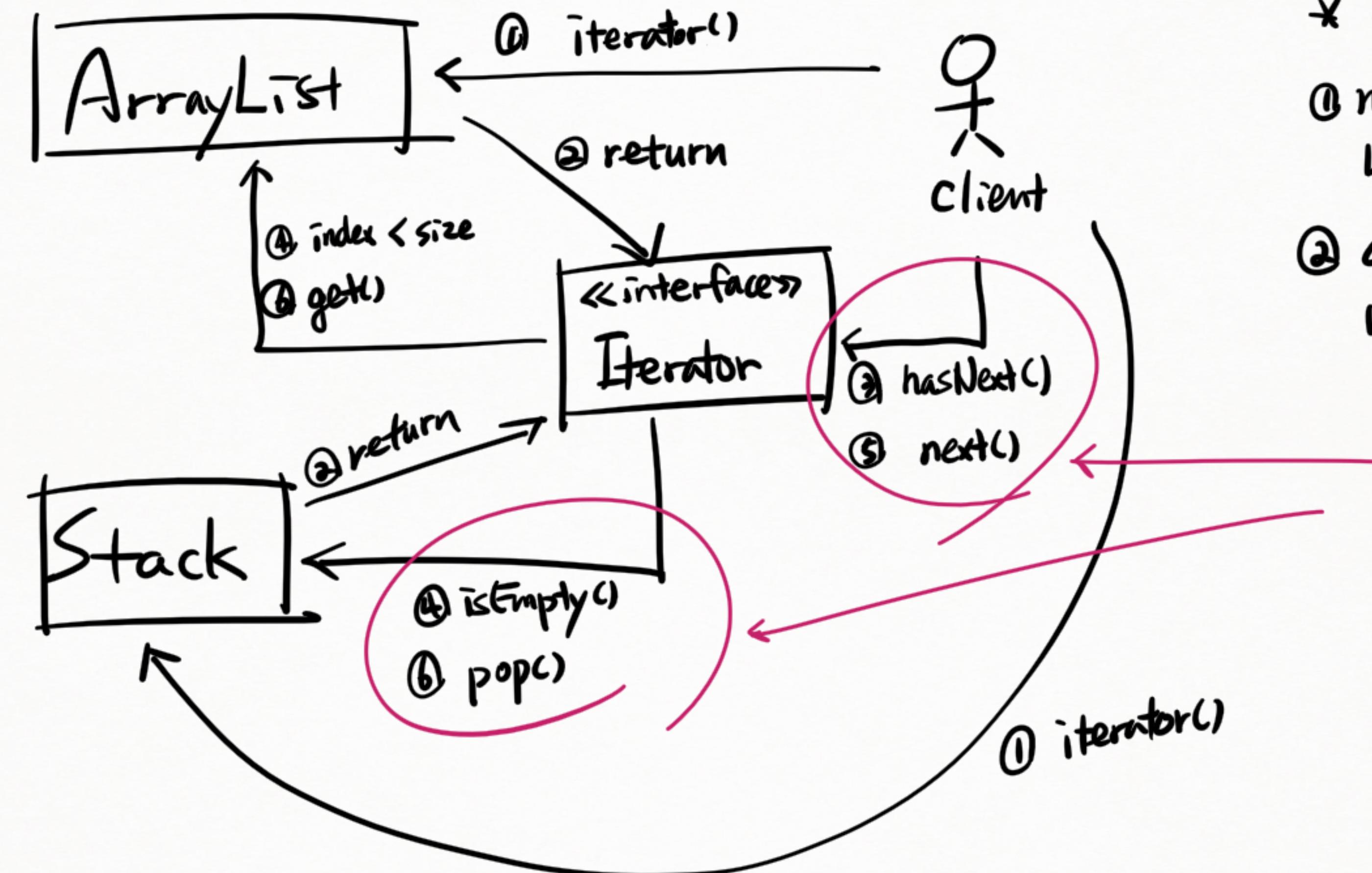
① 목록의 값을 조회하기



\* 데이터 처리 방식에 따라  
일반인(중복인) 방식으로  
데이터를 조회하기!

Iterator  
(데이터를 순회하는 일을 하는 객체)

## \* Iterator mechanism (구조원리)

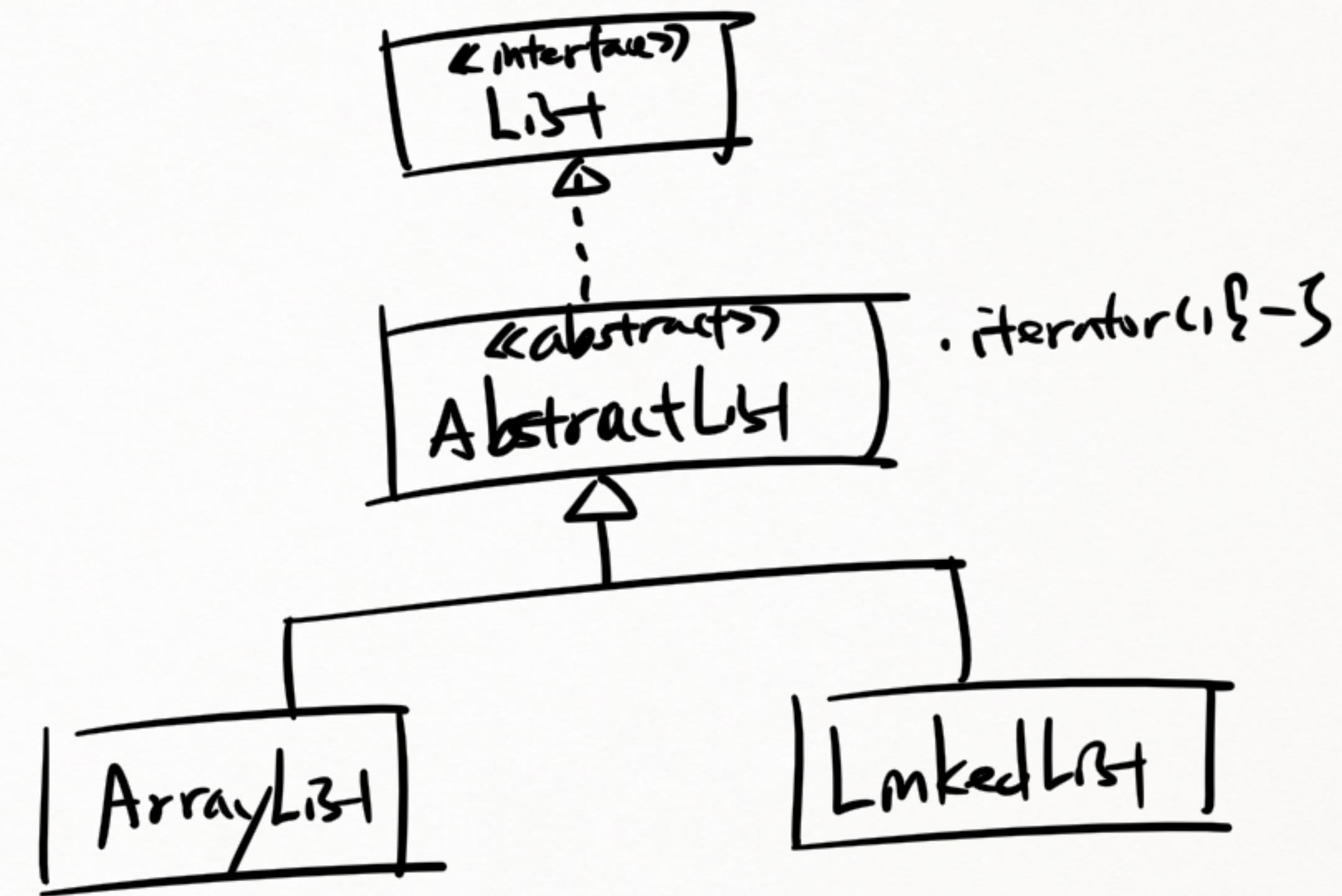
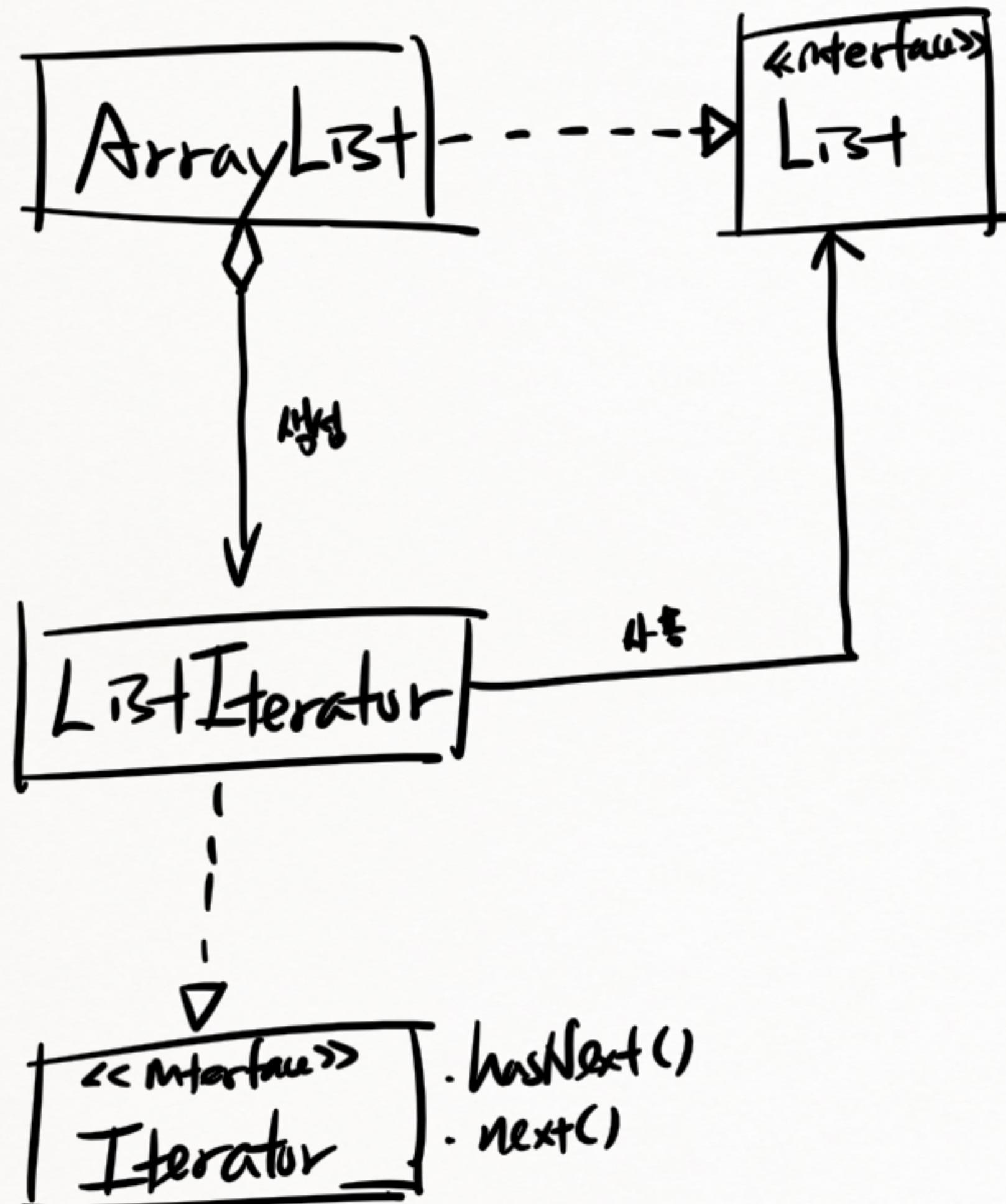


- \* client
- ① network 분야  
↳ 네트워크 요청으로 연결을 수행 S/W
- ② OOP 분야  
↳ 다른 객체를 사용하는 개체.

제작업체가 상관없이  
일관된 방식으로  
작동을 가능케 한다!

## \* Iterator 퀘션 구조

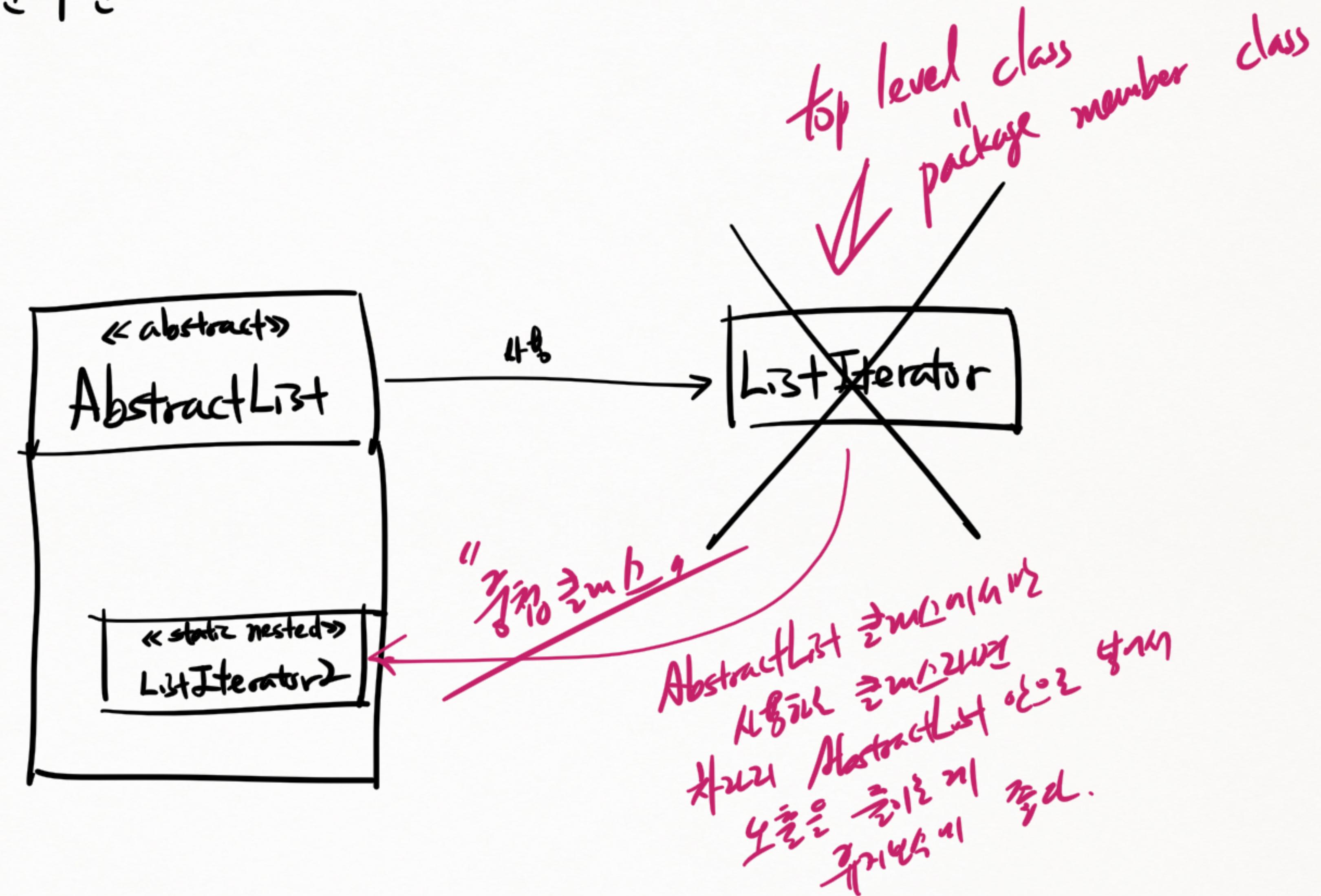
① 퀘션 - top level class



`.iterator()`

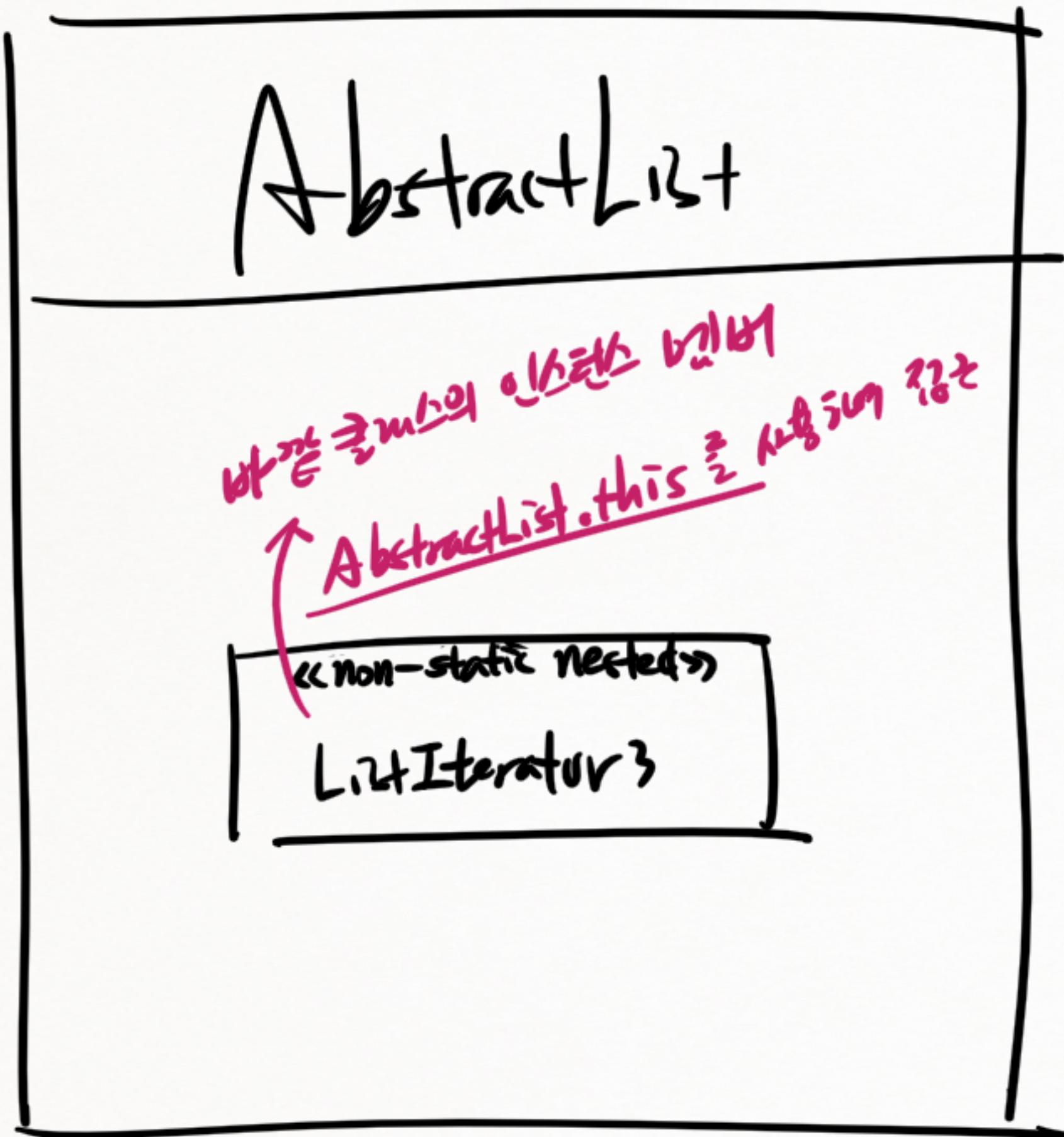
## \* Iterator 퀘션 구현

② 구현 — static nested class

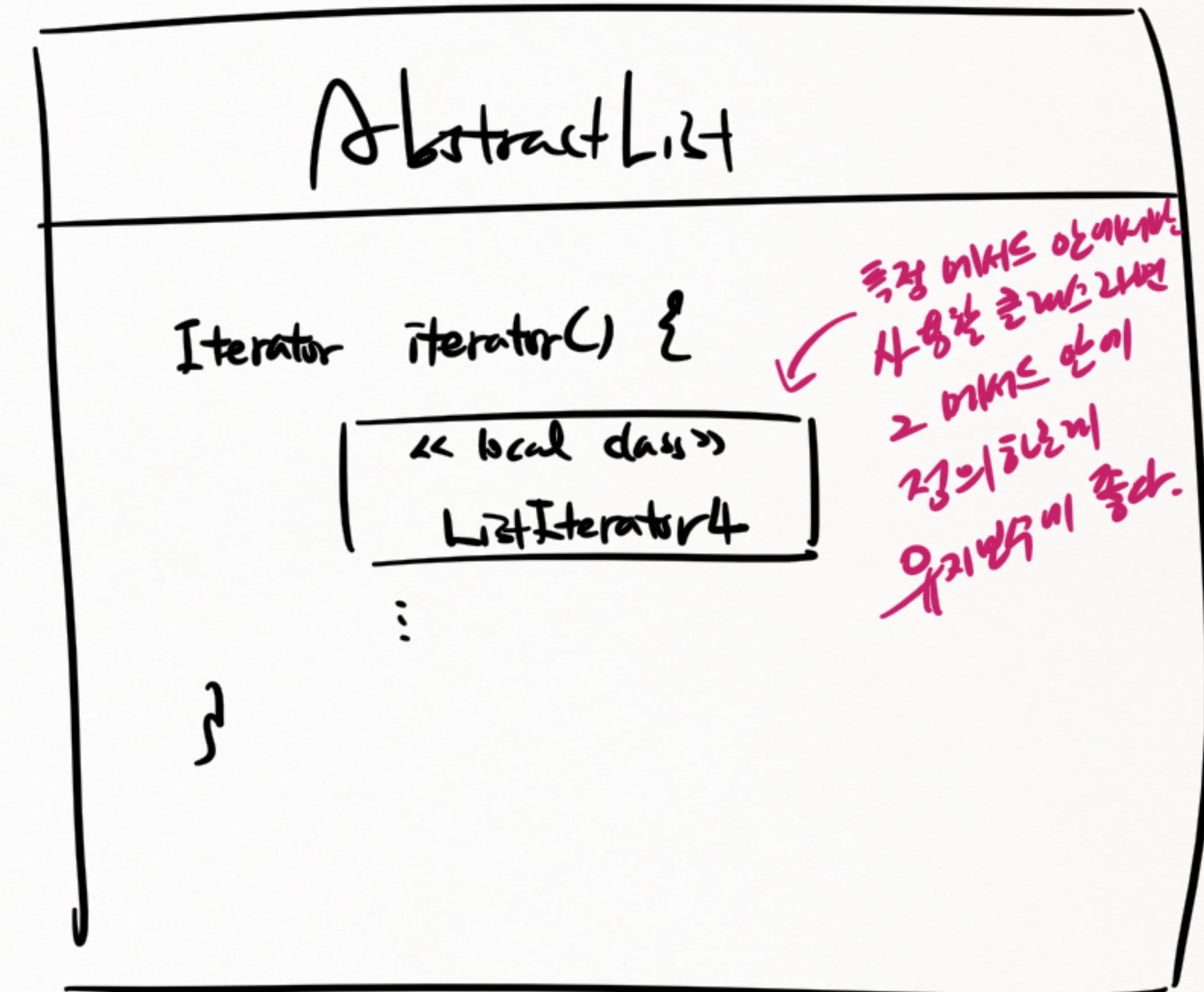


\* Iterator 파편 구현

③ 구현 - non-static nested class

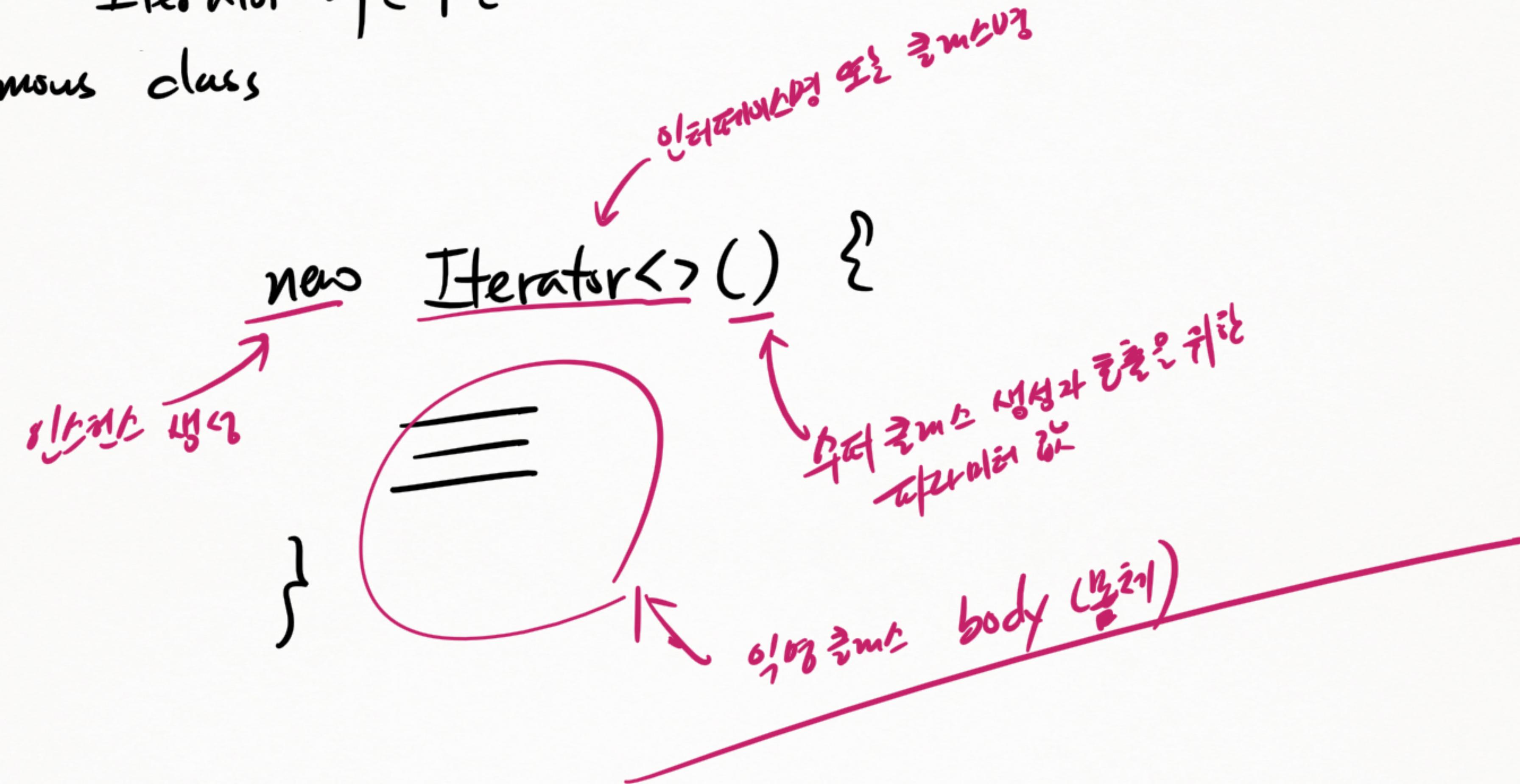


④ 구현 - local class



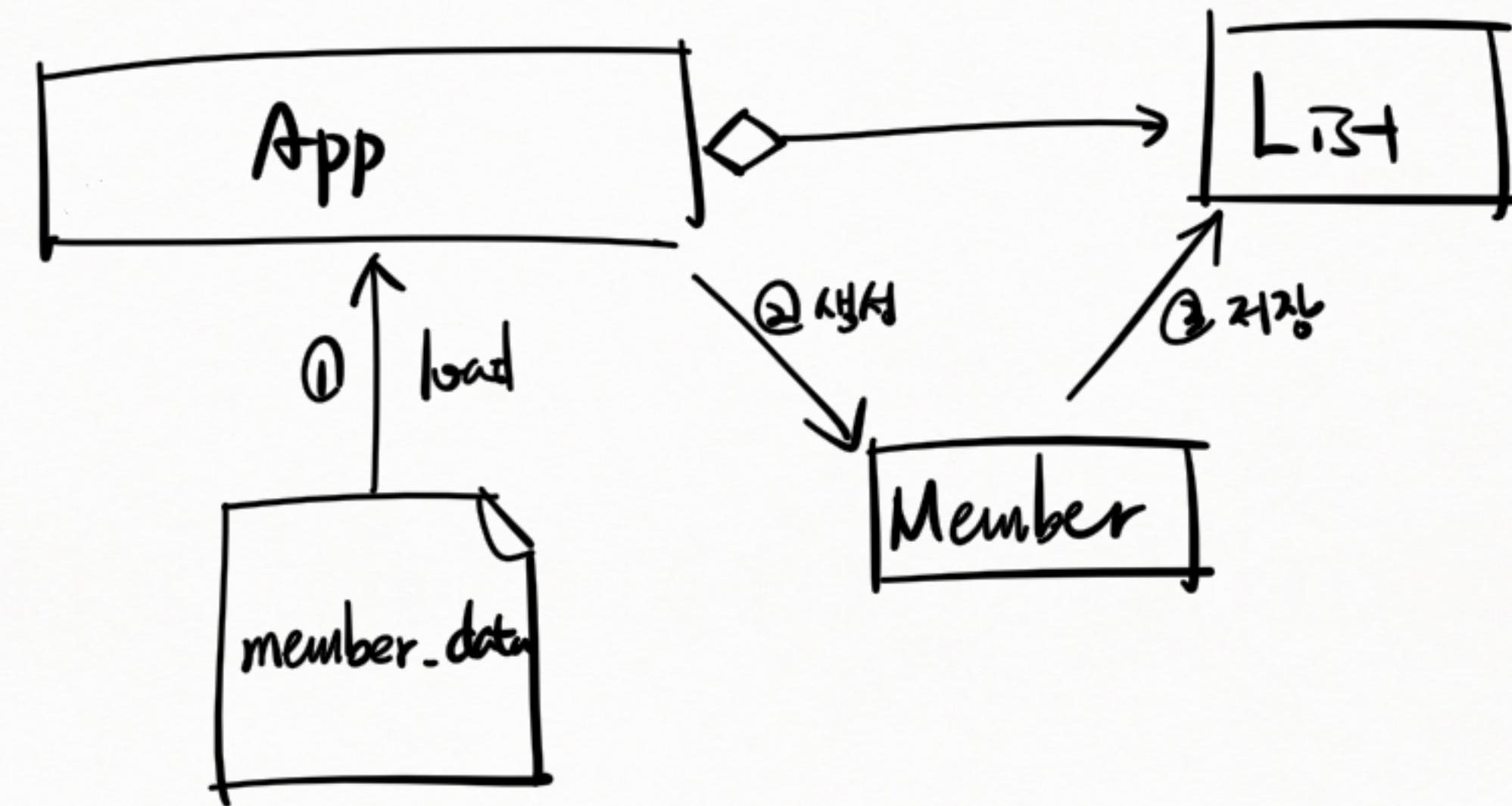
## \* Iterator 파편 구현

### ⑤ anonymous class



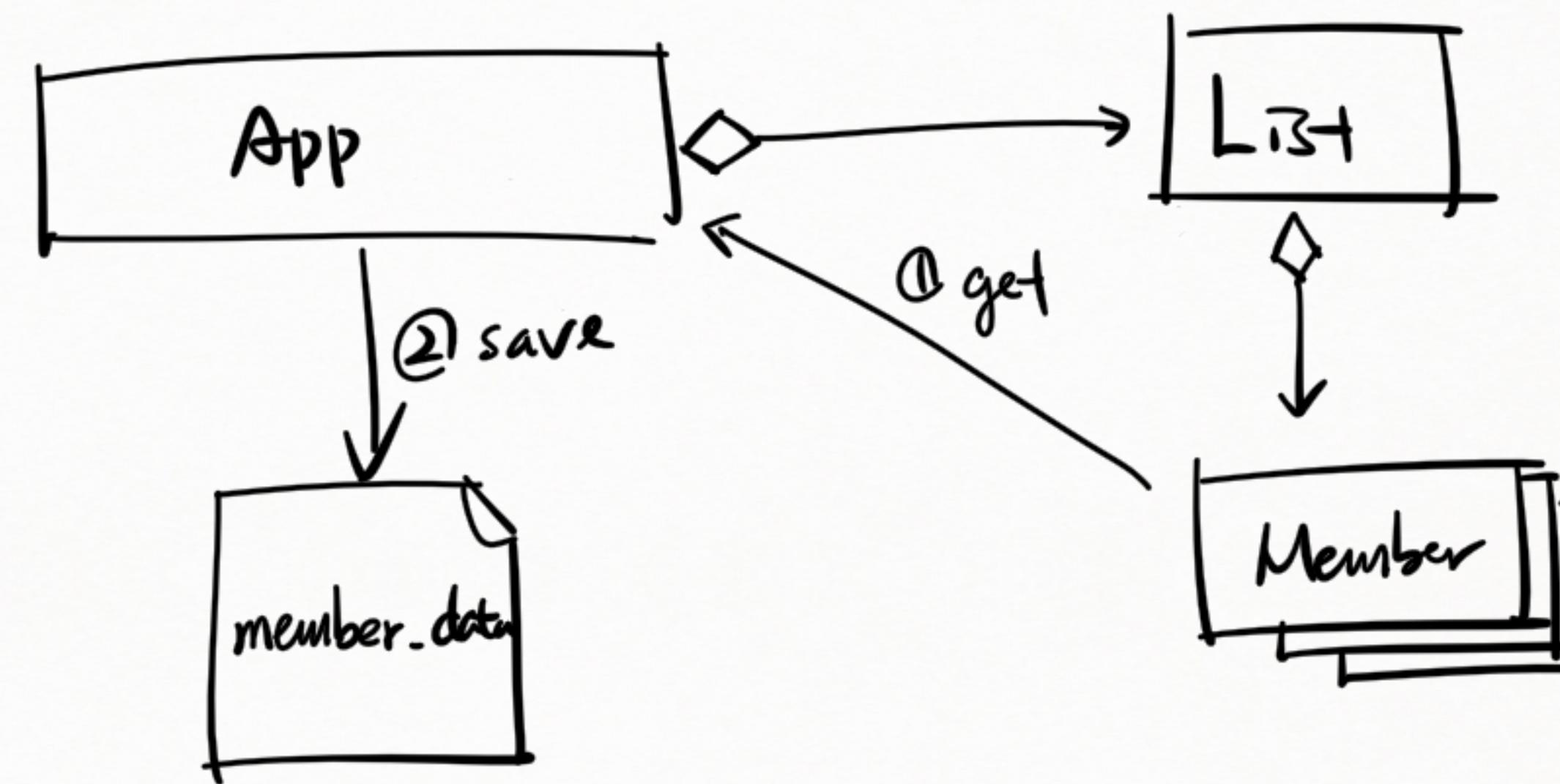
## 27. Data I/O Stream API - binary stream API 사용 ↳ Data 저장

### ① Data 읽기



## 27. Data I/O Stream API - binary stream API 사용 ↳ Data 저장

### ② Data 쓰기



```
int length;
```

```
| 00 | 00 | 00 | 03 |
```

00 03

in.read(4)

... 100 << 8

```
| .. | .. | 100 | .. |
```

! | .. | .. | 00 | .. |  
| .. | .. | 103 | .. |  
00 | .. | 100 | 03 |

61 61 61

in.read()

```
| 00 | .. | 103 | .. |
```

length  
12345678910

↓ ↓

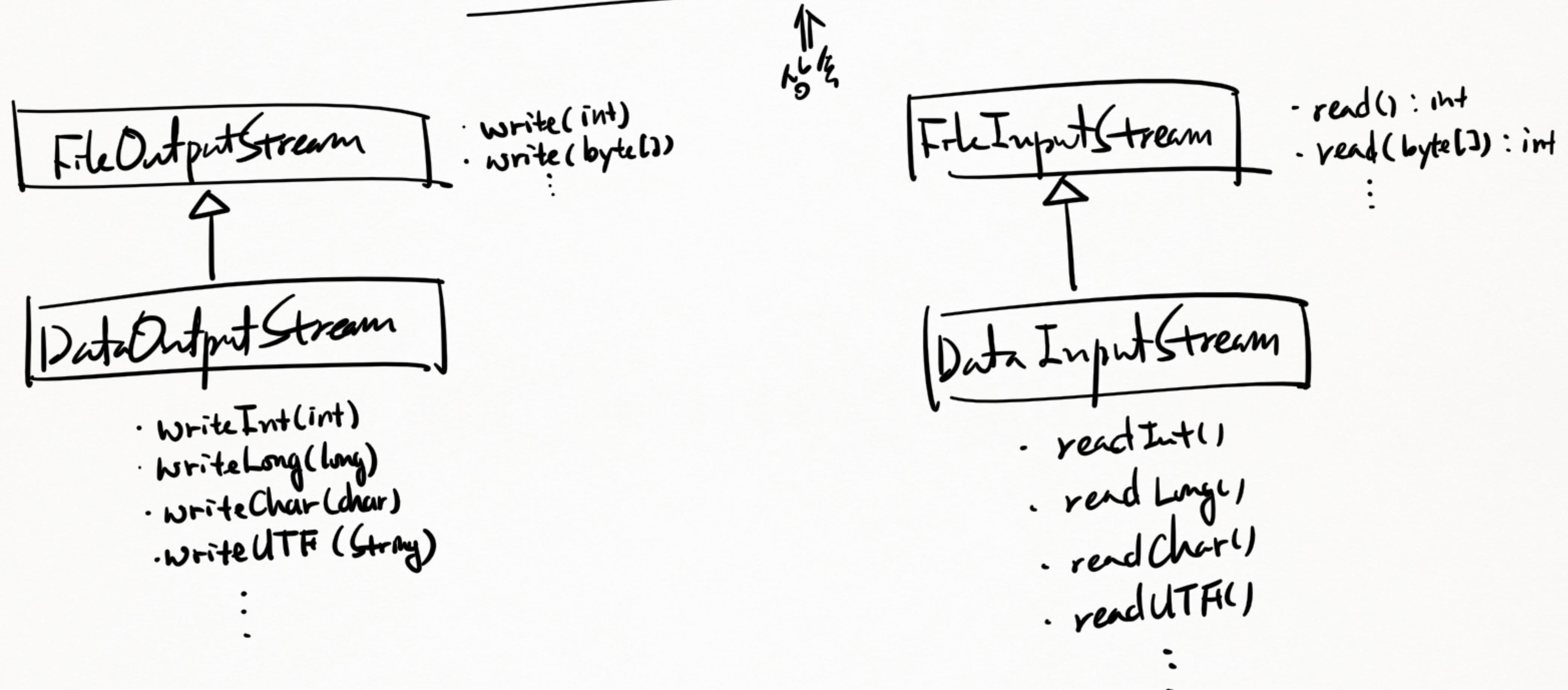
in.read(buf, 0, 3)

0 1 2  
61 | 61 | 61 | ...  
1000 bytes

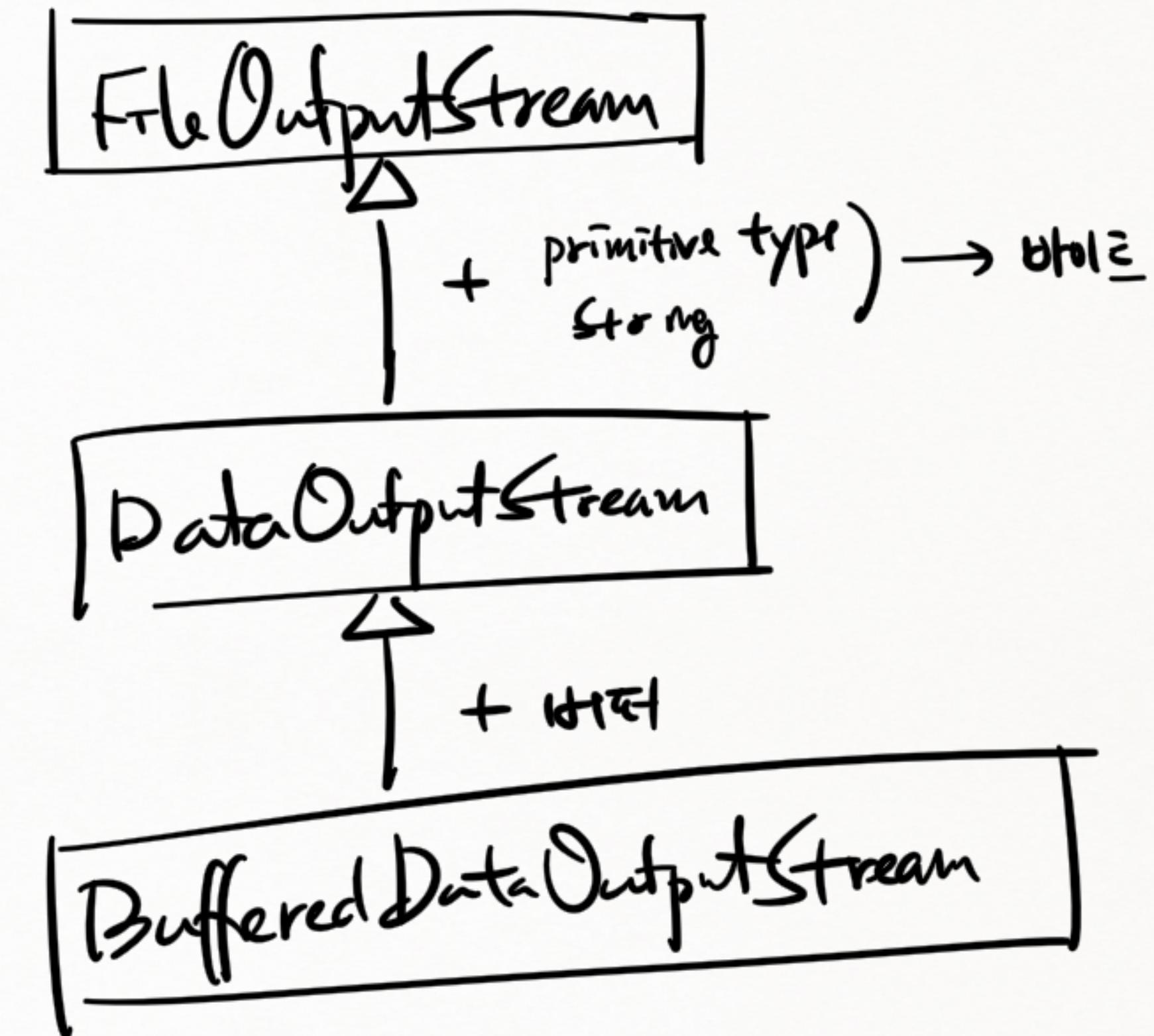
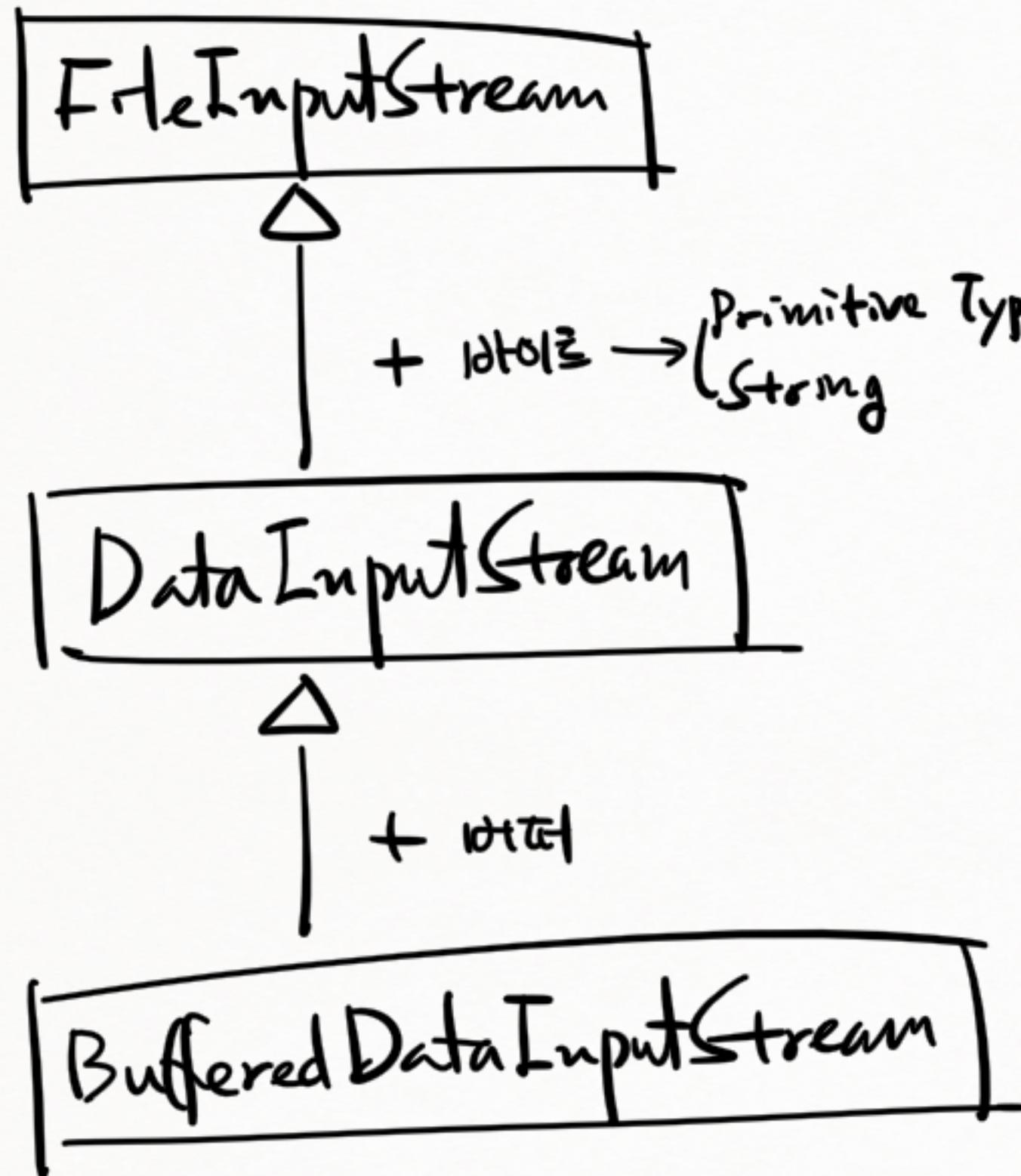
new String( , 0, count, "UTF-8")

member . setName( )

## 28. FileInputStream / FileOutputStream + (primitive type) String 입출력 기법



## 29. 파일 입출력의 버퍼링 예제 : 파일 읽기 쓰기

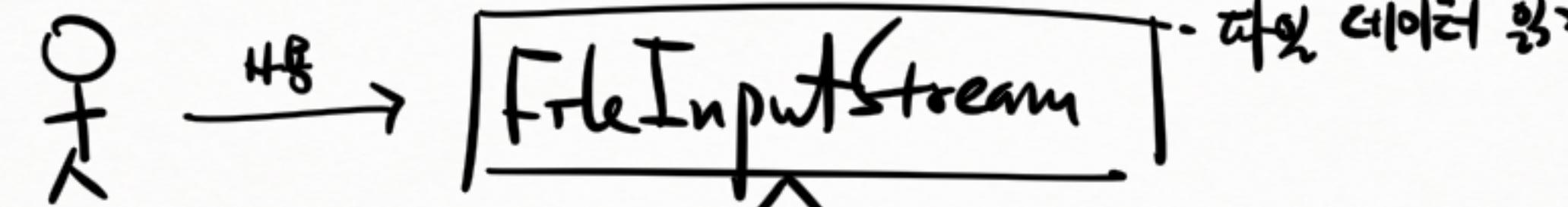


## 30. 기능을 확장할 때 상속과는 Decorator 패턴 적용

### ① 상속을 이용한 기능 확장의 문제점

- byte / byte[] 읽기

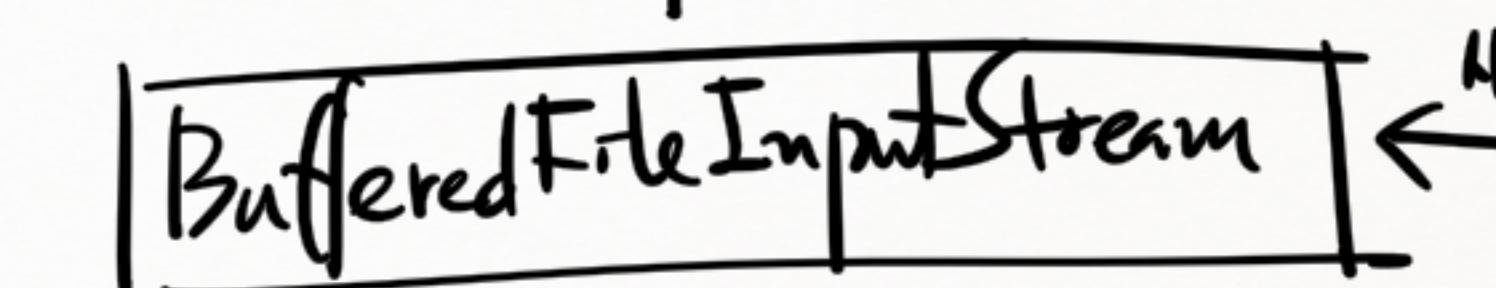
- \* Primitive Type / String 읽기 불편



- (primitive type) 읽기 편함  
String

- 바이트 단위로 읽기 대비해 대량의 데이터를 읽을 때 overhead 발생!  
(Data Seek Time)

- 버퍼를 이용해서 읽기 성능 개선

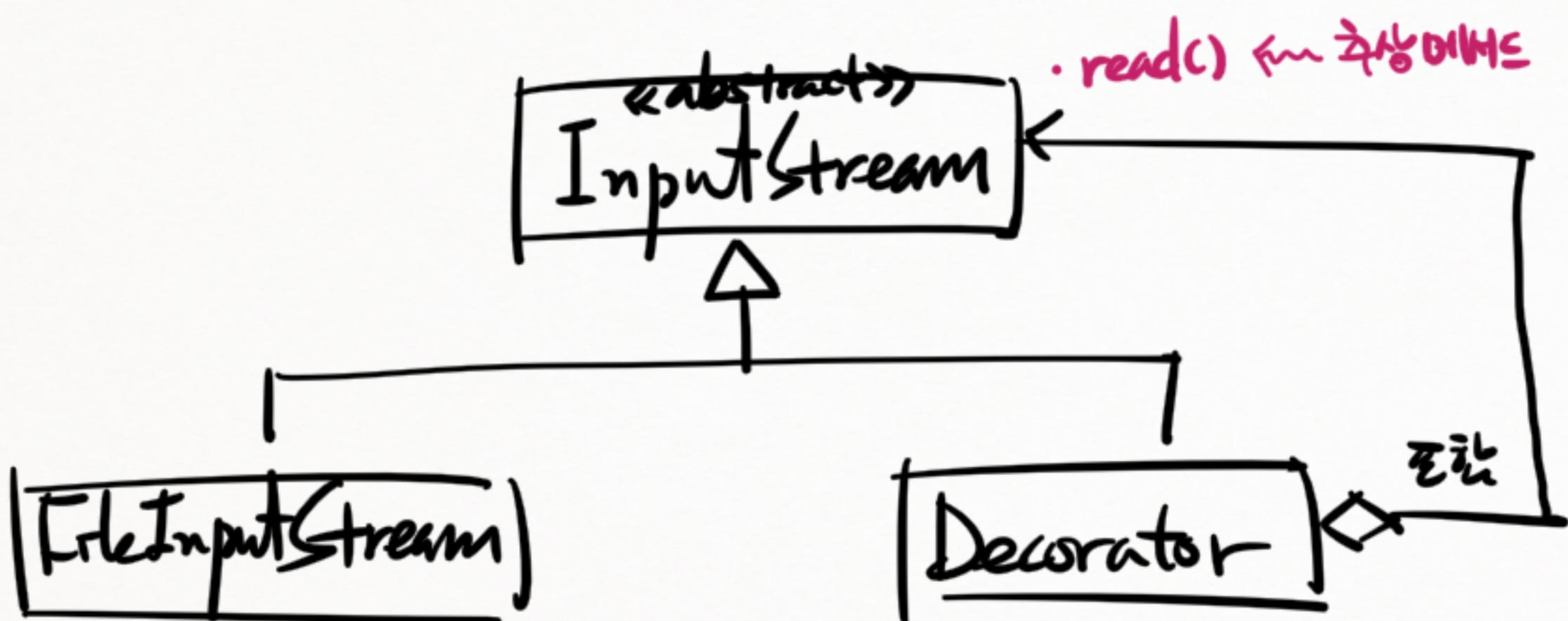


\* 상속은 좋은 유도의 기능은  
제한된 한계.  
그러나 상속의 sub 클래스는 그 상위 클래스를  
상속하는 한계의 한계!

(primitive type)의 데이터는  
읽을 때마다 많은 성능이 사용할 때  
바이트 단위로 데이터를 읽을 때  
입기 성능은 개선되는 한계.

30. 기능을 확장할 때      쌍축 with Decorator 티켓 적용

② 장식적(Decorator) 패턴 → 기능을 덧붙이고 예상하지 않을 수 있는 문제를

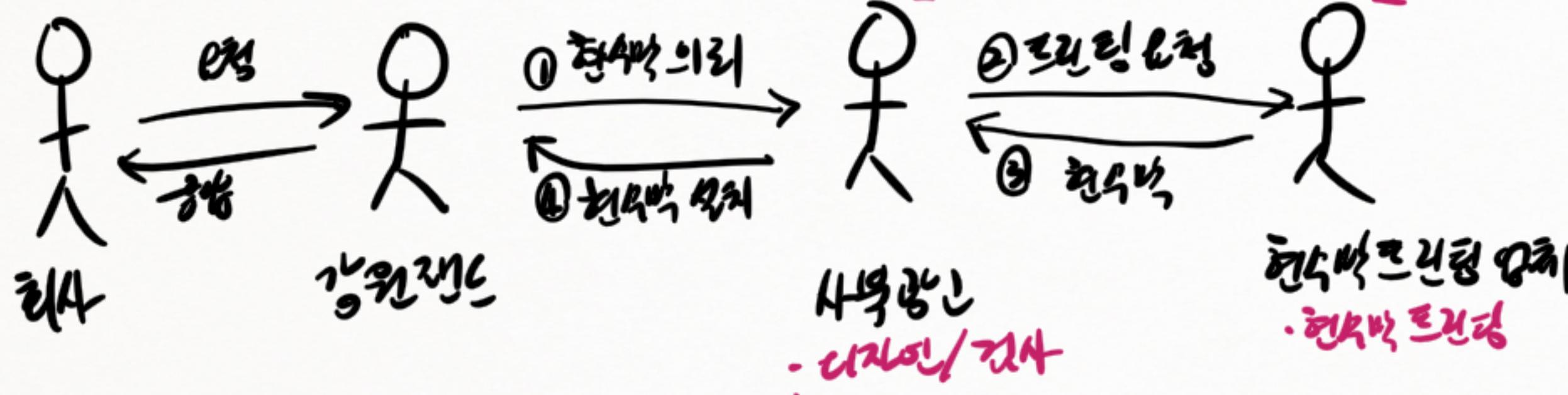


## a) DataInputStream

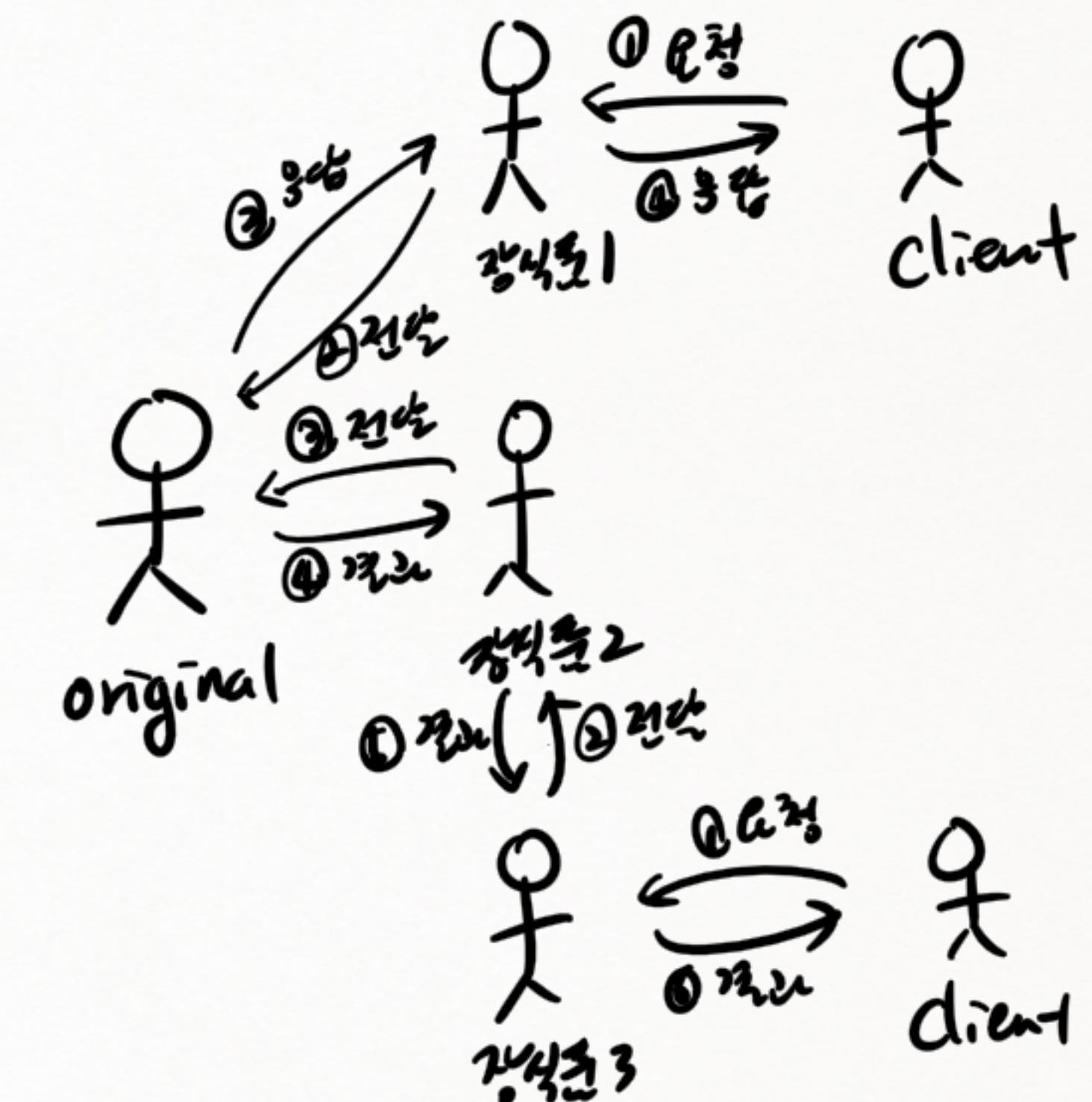
## BufferedInputStream

✓ Decorator

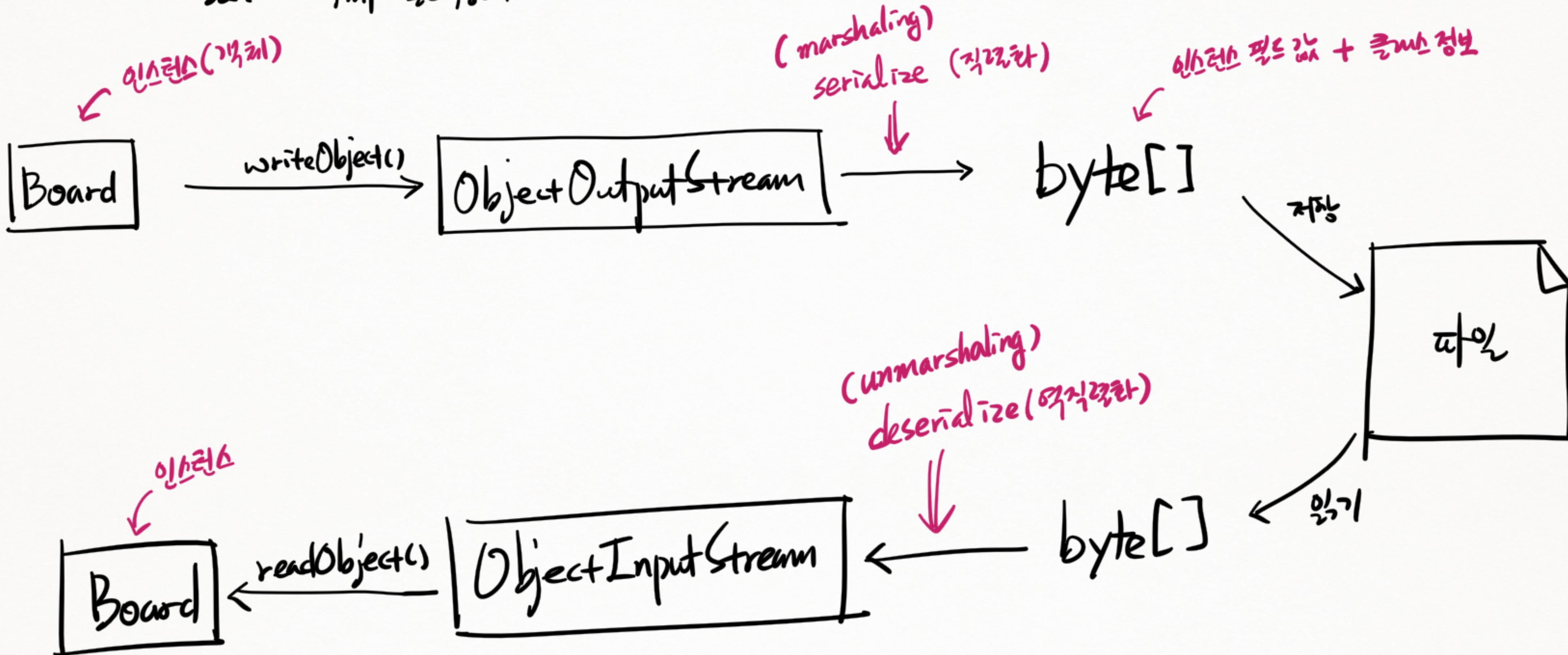
Original



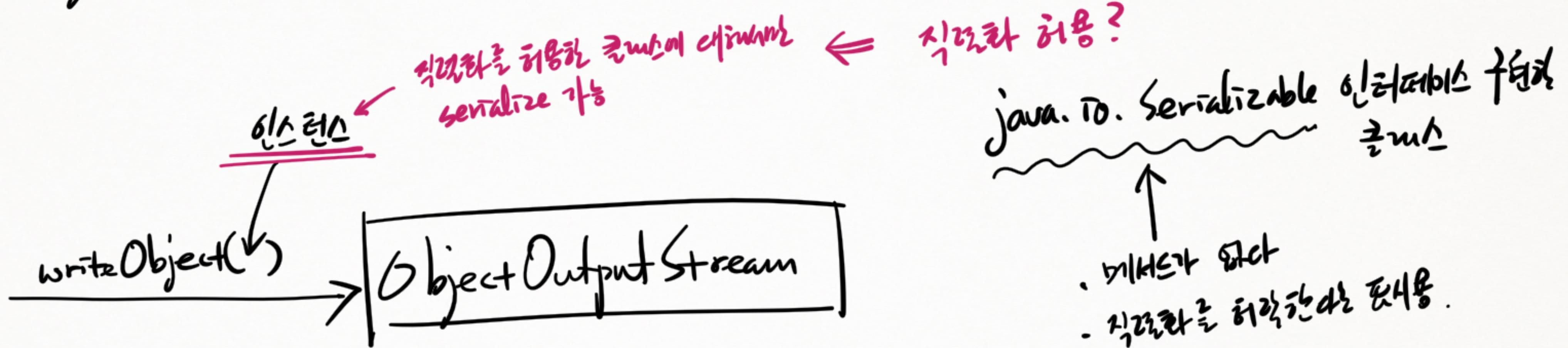
↳ Decorator 패턴 (GOF)  
(Composite 패턴과 유사)



## 32. 객체 흘러보기



## \* java.io.Serializable 인터페이스

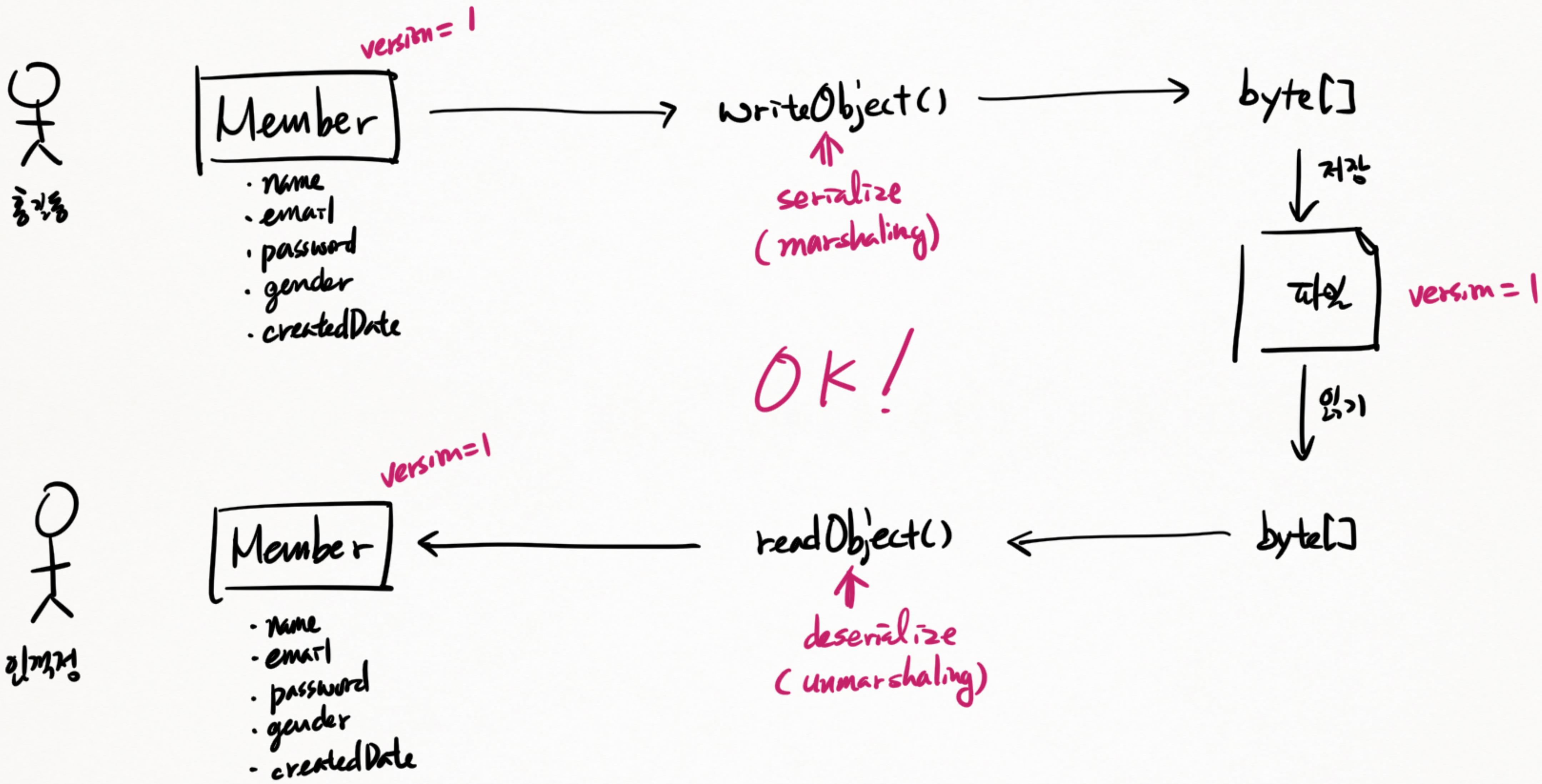


\* 직렬화는 허락 받았을 때 가능한가?

↓  
Yes

↑  
직렬화를 이용한 대량 트랜잭션  
이용은 가능하다!

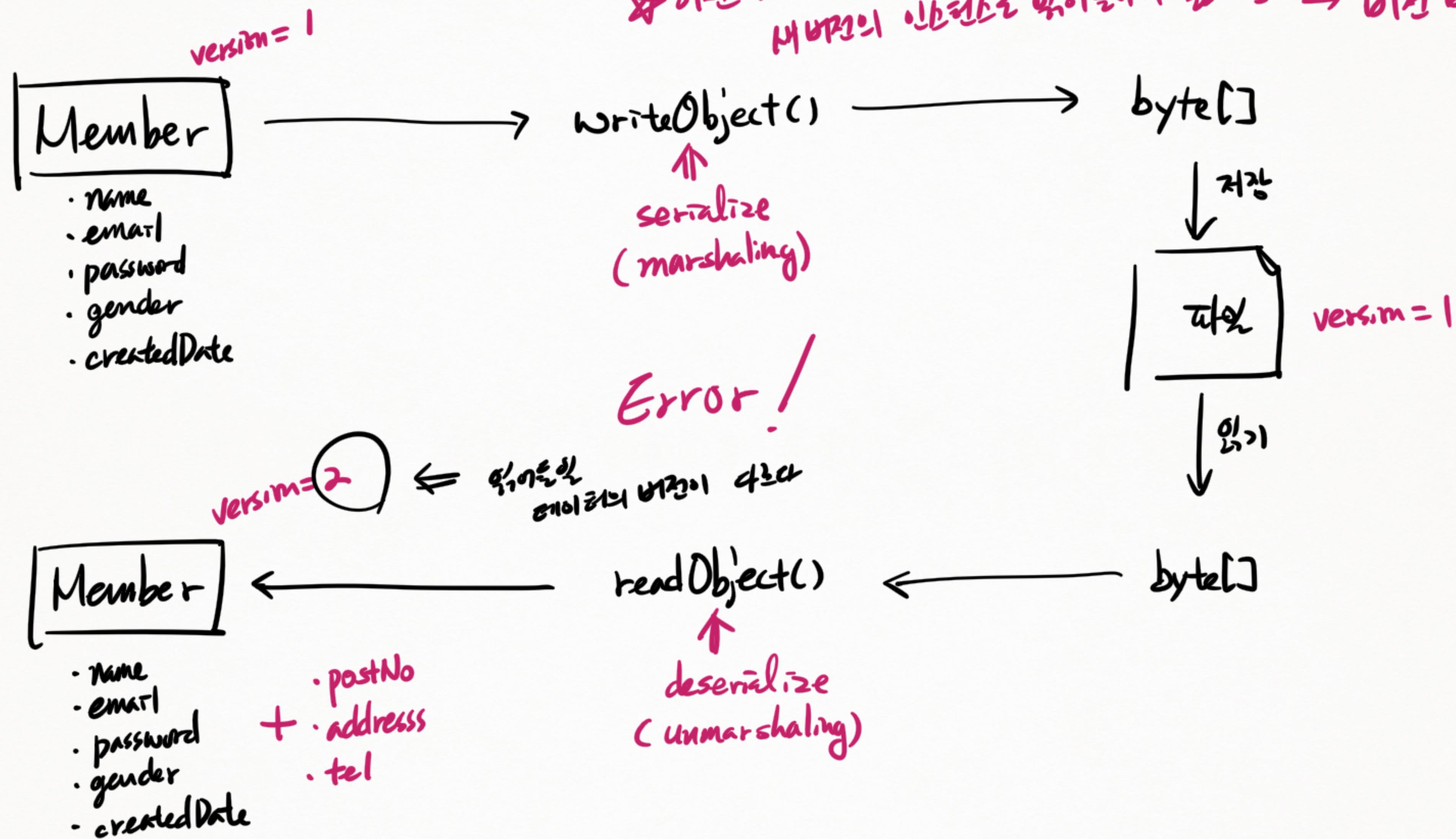
\* serialVersionUID 스태틱 필드



\* serialVersionUID 스태틱 필드

○ 흰색 풀

○ 흰색 풀



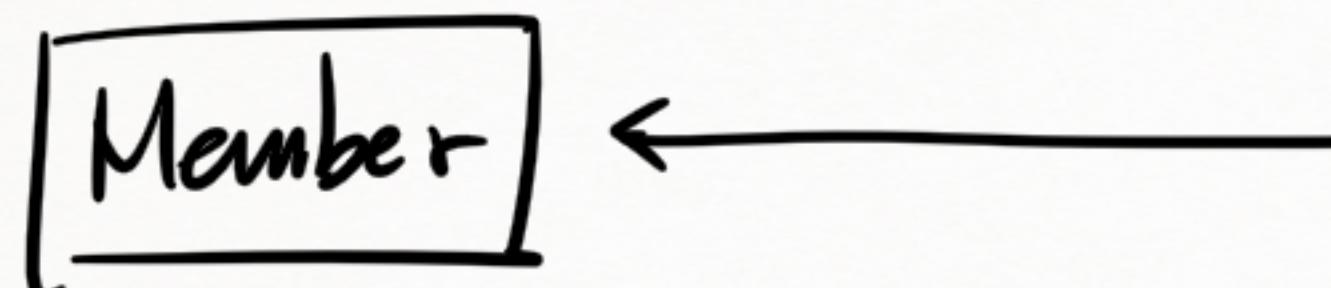
\* 이런 버전은 serialize 한 데이터를  
내부의 인스턴스로 맞이하기 쉬워 ⇒ 이런 버전은 알리하라!

\* serialVersionUID 스태틱 필드

○ 흰색  
회원 등록



○ 흰색  
회원 수정



version = 1

- name
- email
- password
- gender
- createdDate
- + postNo
- + addressss
- + tel

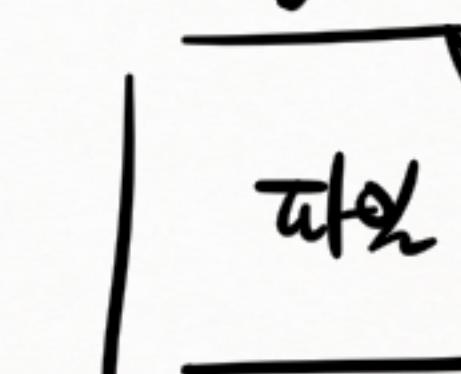
\* 이런 방식으로 serialize 할 데이터는  
내부의 인스턴스로 일어도 문제없음 => 버전 번호를  
같이 한다!



serialize  
(marshaling)

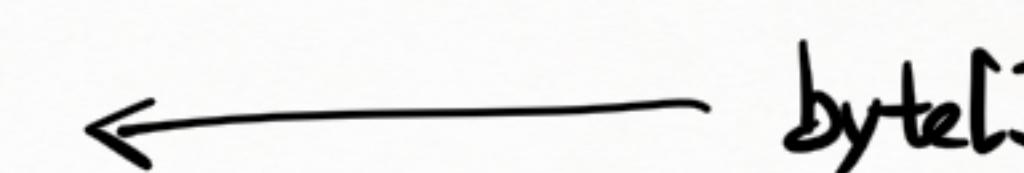
byte[]

저장



version = 1

OK!

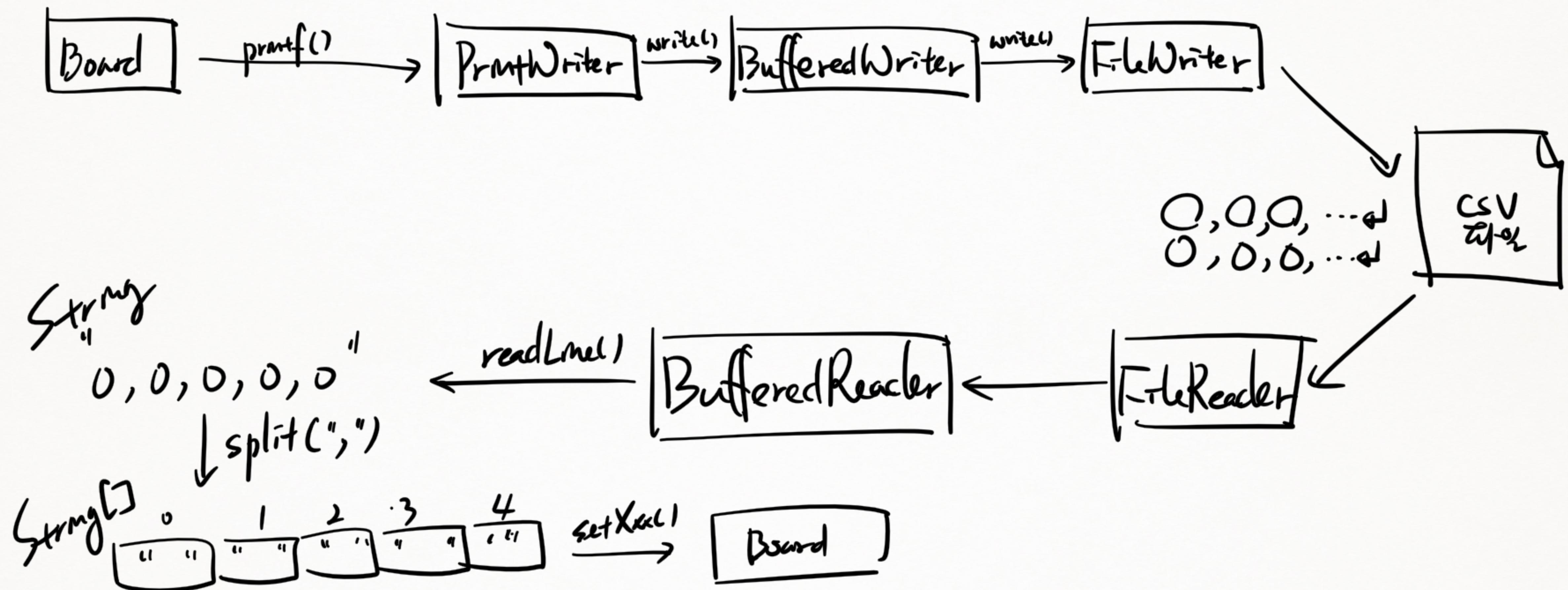


deserialize  
(unmarshaling)

readObject()

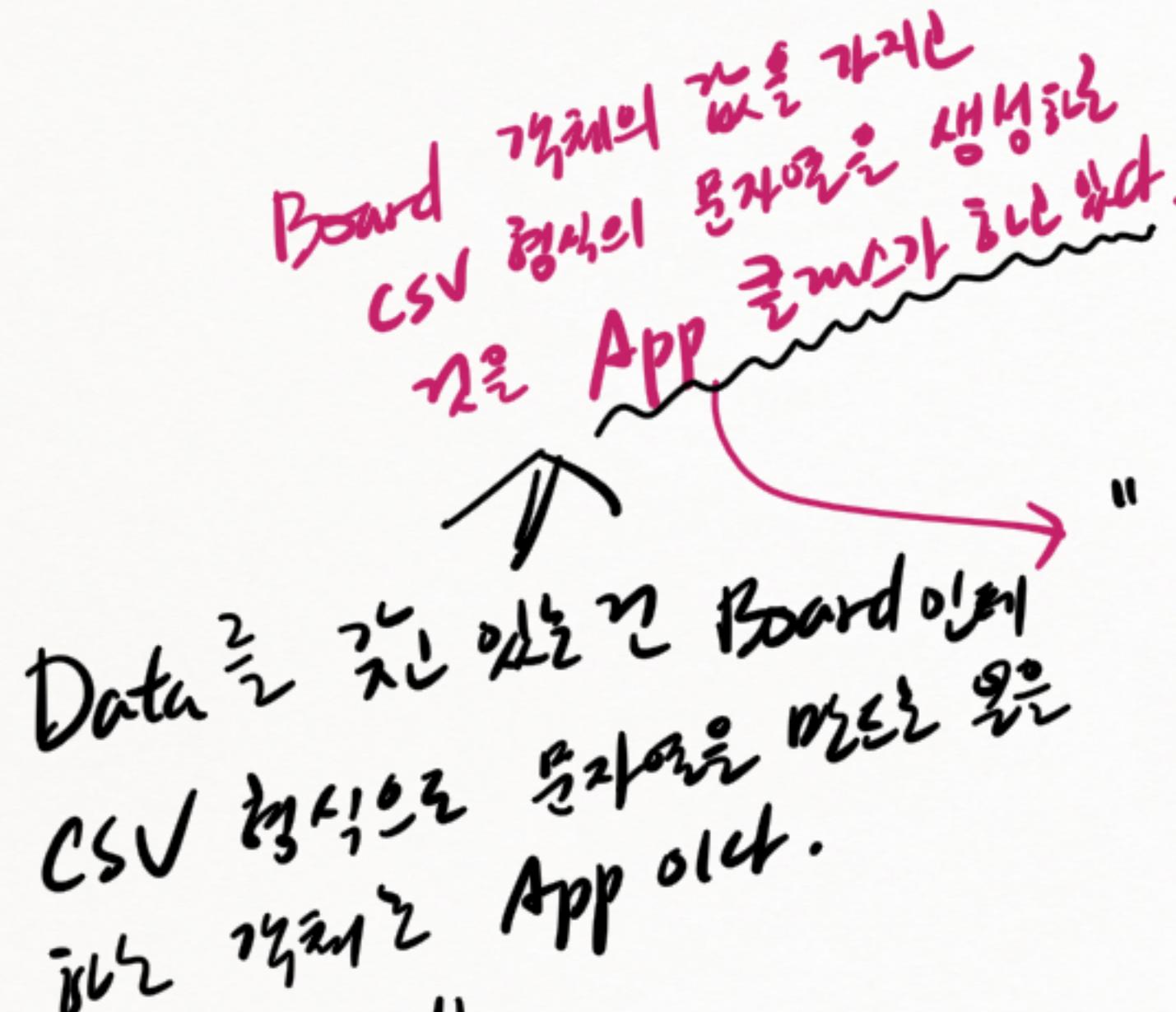
byte[]

33. 텍스트 파일 (CSV) 을 파일로 읽기

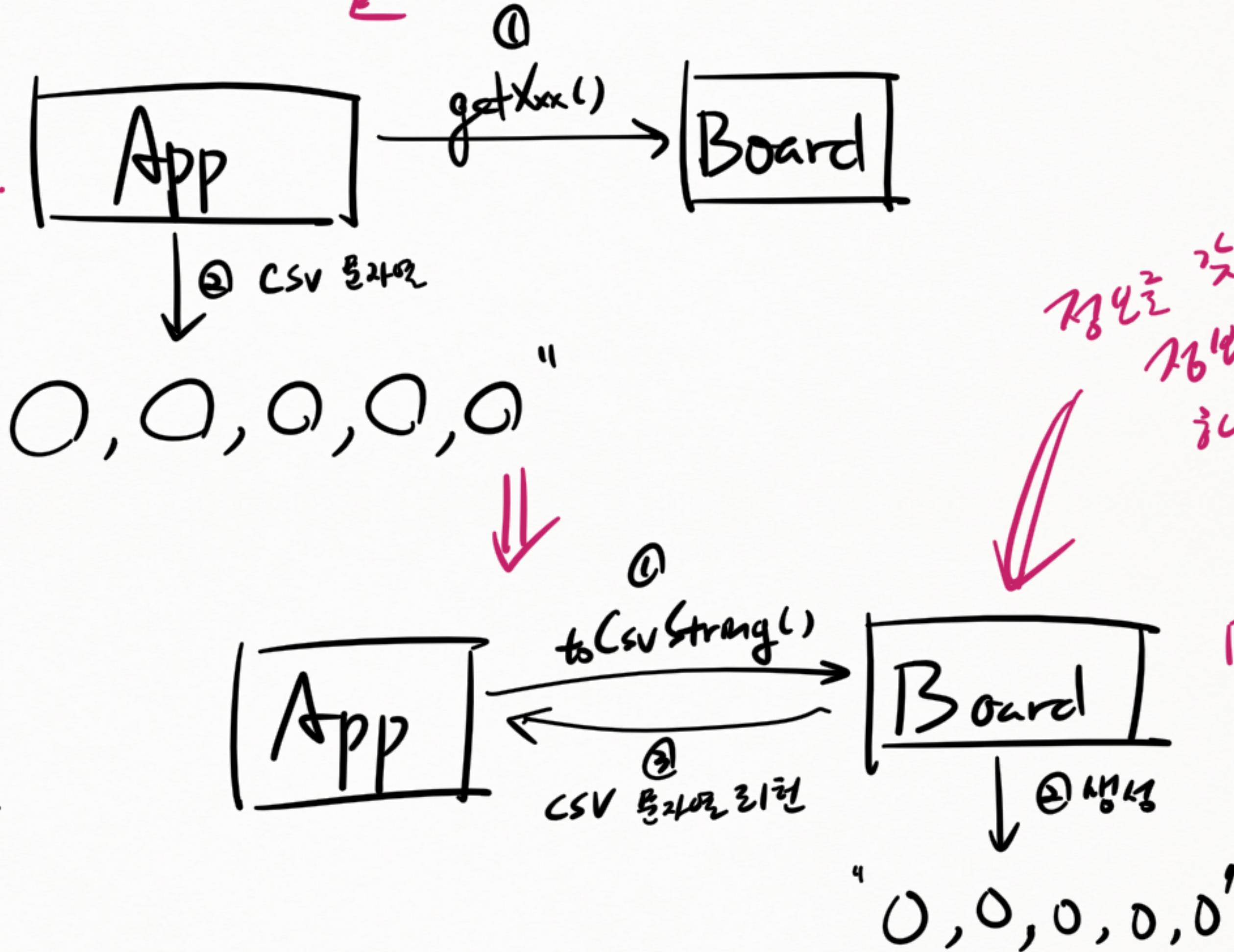


## 34 · Refactoring : ① Information Expert

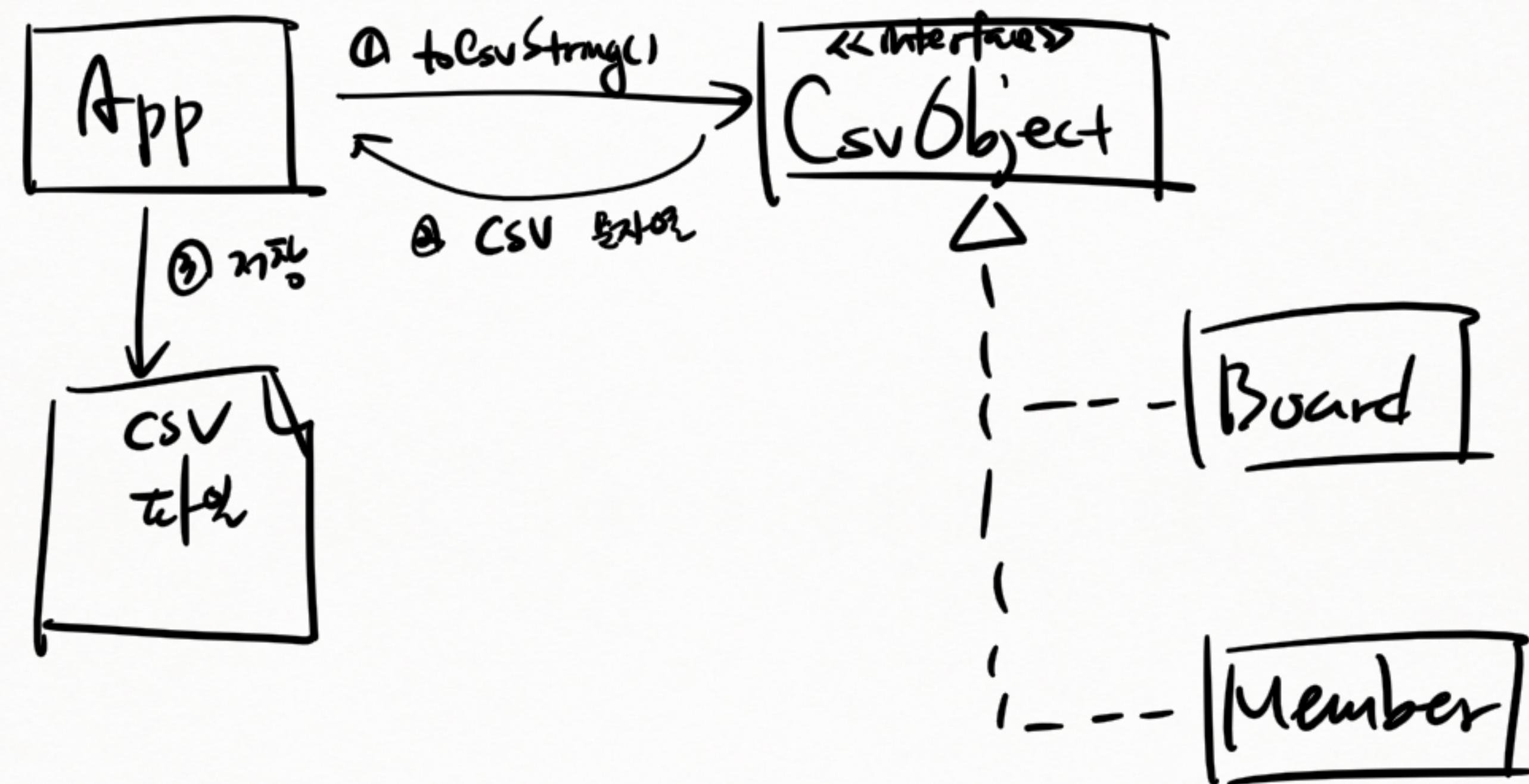
문제점: Board의 블록이 추가되거나  
삭제되며 App 클래스를 변경해야 한다



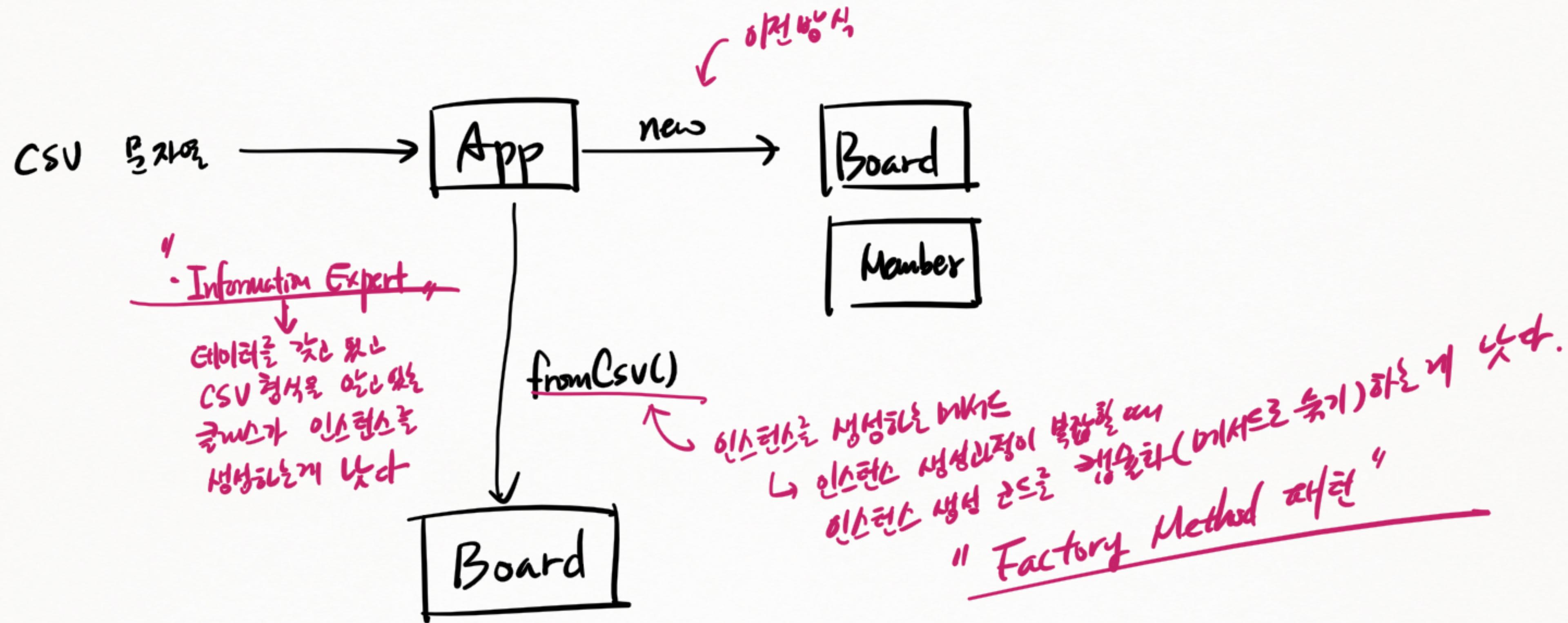
"Information Expert"  
자료로 전문가가 된 것



## \* Interface 분리의 원칙



## ② Factory Method (GOF) 패턴 적용



## ② Factory Method (GOF) 패턴 적용 → 개선

*T가 타입 파라미터임을 선언!*

```
<T> void loadCsv(String filename, List<T> list, Class<T> clazz) {
```

\* Reflect.m API를 사용하면  
매번 코드를 찾고 훔을 드립니다! + Generic을 사용하면  
다양한 타입에 대응할 수 있고  
매번 코드를 찾지 않아!

Method factoryMethod = clazz.getDeclaredMethod("fromCsv", String.class);

↑  
매번 코드를 찾고 매개변수  
타입을 정의하는  
방법입니다.

↑  
현재 클래스에서  
정의된 메소드를  
찾는다.

↑  
매서드명

↑  
파라미터 타입

(T) factoryMethod.invoke(null, line);

↑  
매소드 호출

↑  
인스턴스 주기  
(스레蚀 매소드의 경우, null을 넘긴다.)

↑  
매소드를 호출하는 단계를 파악합니다.

## 35. JSON 형식으로 입출력하기

↳ JavaScript Object Notation : 자바스크립트 객체 리터럴 문법을 모방하여 만든 텍스트 파일 형식

① 자바스크립트 객체 리터럴

문자열 - { "문자열",  
          '문자열'

숫자 - 3.14  
       3.14

논리 - true/false

문서 - {  
          name: "홍길동",  
          age: 20,  
          working: true  
        }

JSON 흔히

{

"name": "홍길동",  
"age": 20,  
"working": true

}

반드시 큰따옴표(")를 사용해서  
프로그래밍 이름을 써야 한다.

문자열은 반드시 큰따옴표 사용

XML이나  
C언어하고 차이가 있다.

JSON

또는

XML

deserialize

"

decomposing

encoding

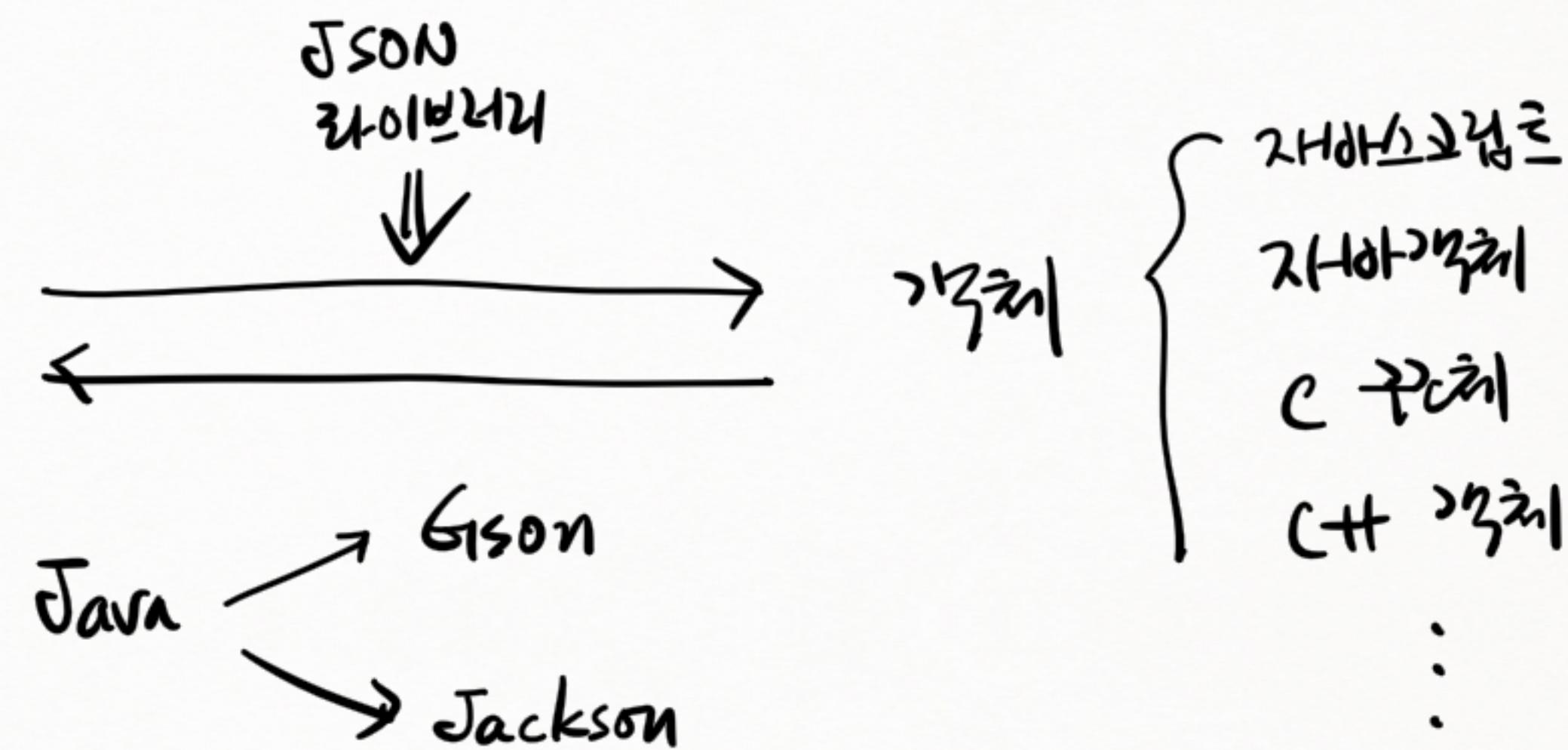
"

serialize

자바스크립트 객체

## \* JSON 인코딩 / 디코딩

JSON 형식의  
문자열



\* JSON 훔기시 필드명은 끄로처리링?

class Member {

String nm; ← field

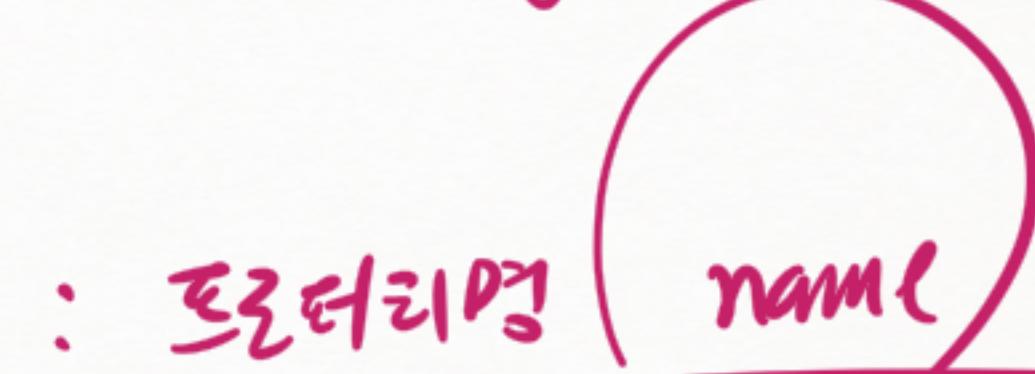
void setName(...){...}

String getName(){...}

}



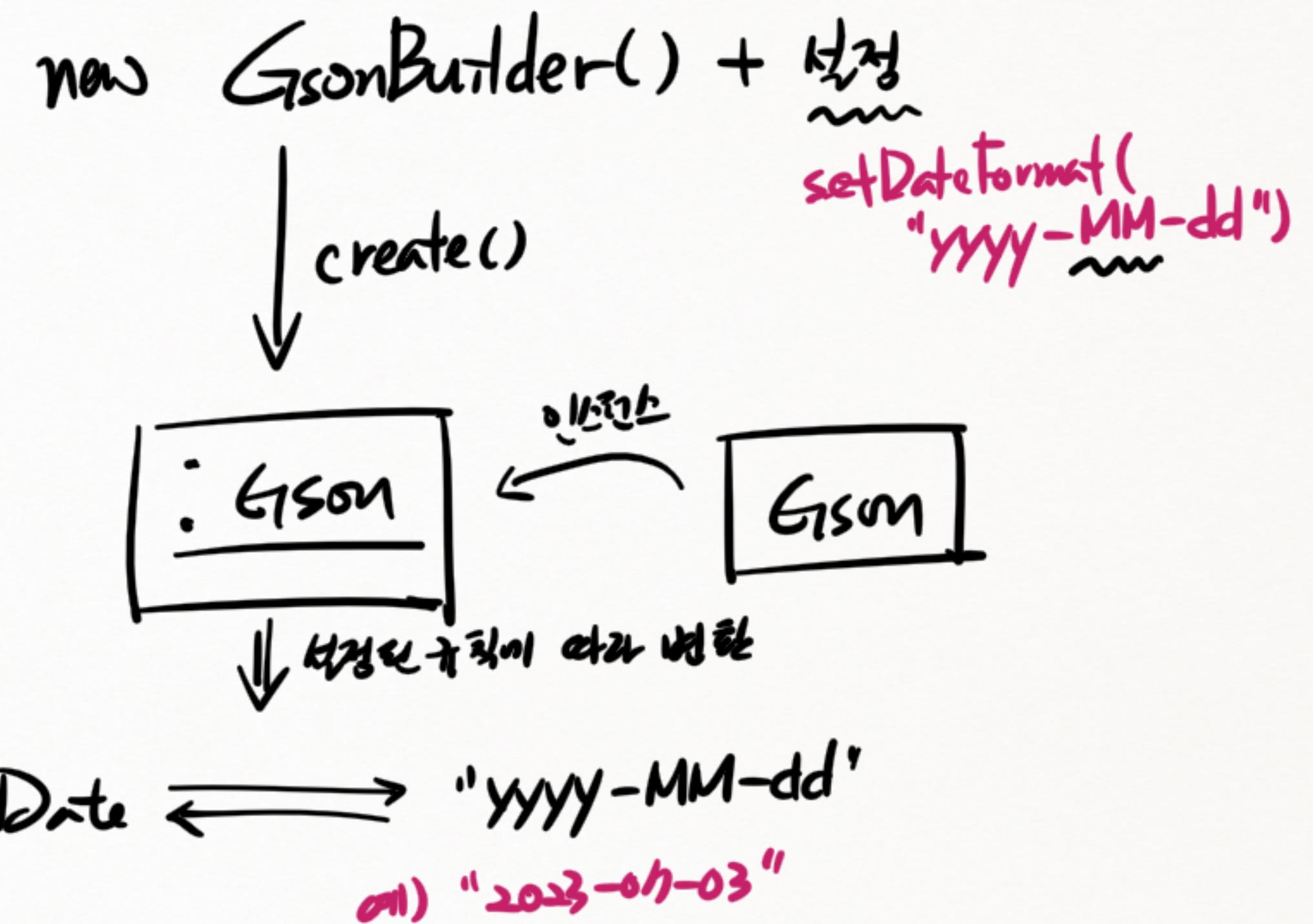
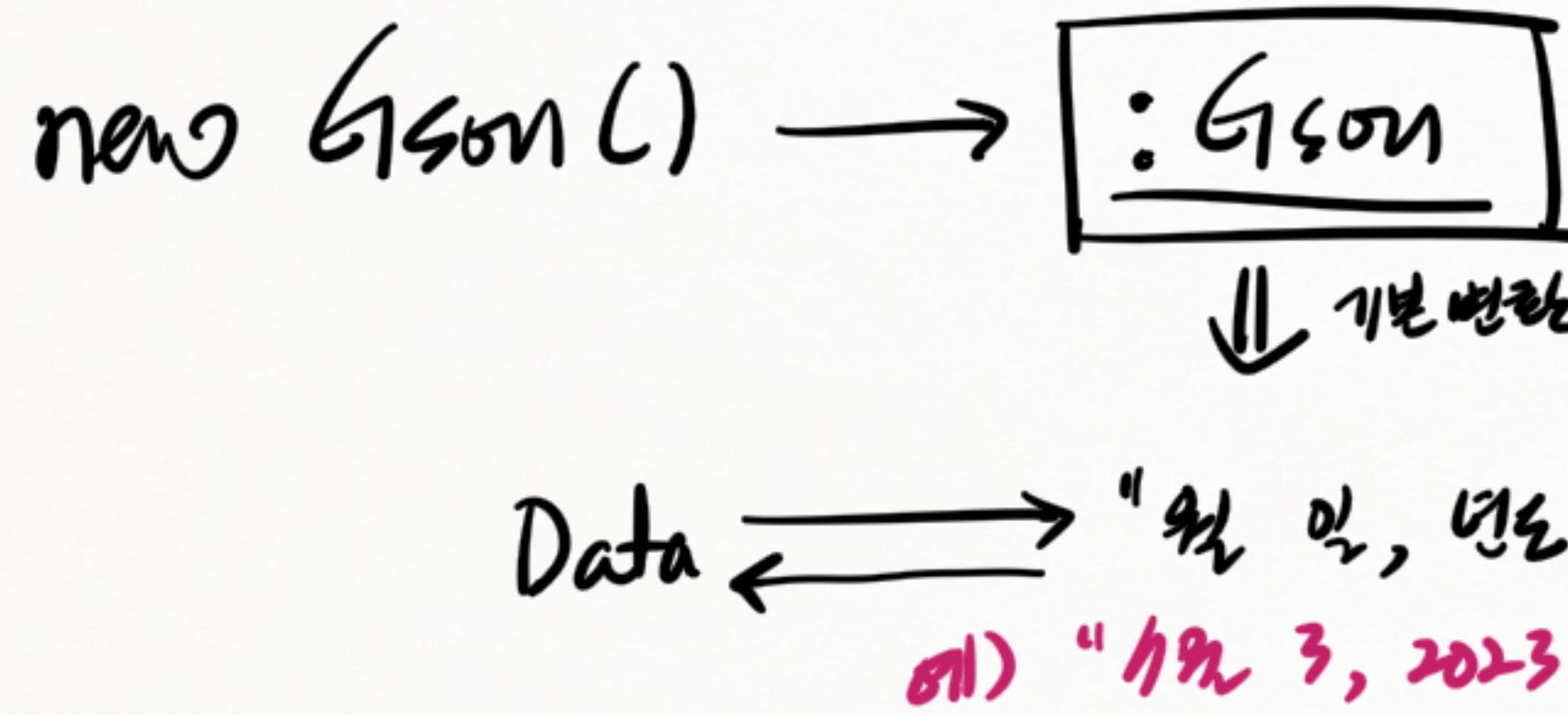
property  
setter/getter



- set/get 메서드
- 나머지 필드명이나  
첫번째 알파벳을  
소문자로 할 이는

프로퍼티와 필드명이 다른 경우  
• Gson → 필드명 사용  
• Jackson → 프로퍼티 사용  
↓  
결론!

\* Gson with 날짜 다루기

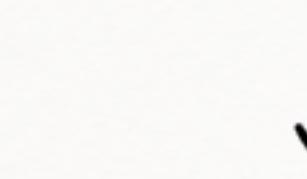


\* yyyy : 2023년  
month MM : 2023년  
dd : 2023년 01월  
HH : 2023년 01월  
mm : 2023년 01월  
ss : " 초 "

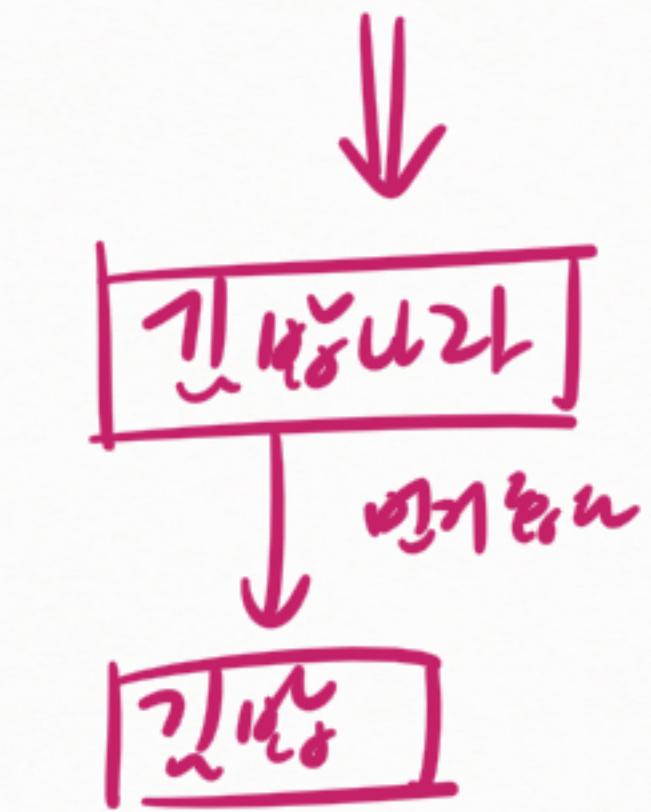
## \* Gof의 Builder 패턴

① new 뒤에  
자기만 인스턴스 만들기

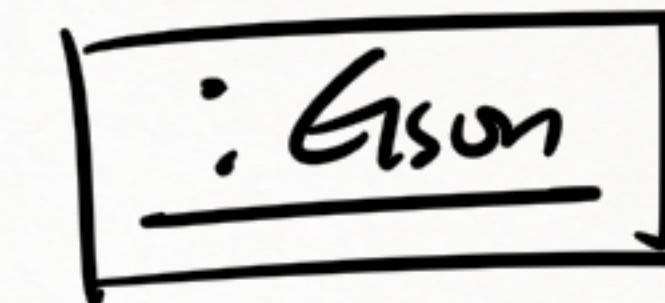
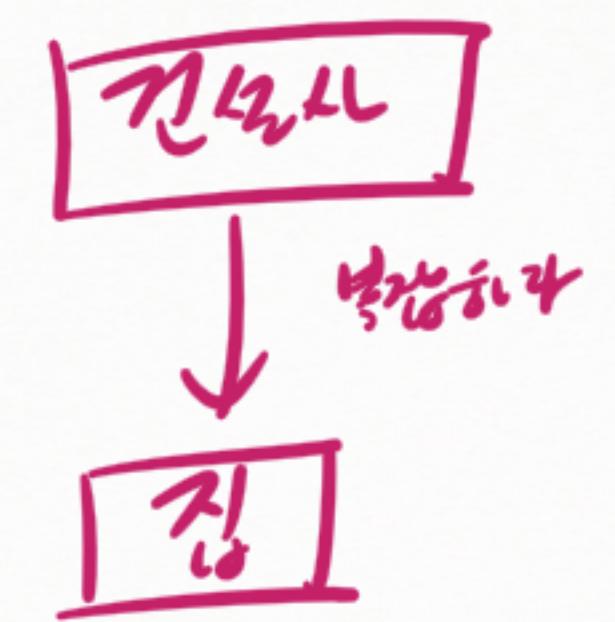
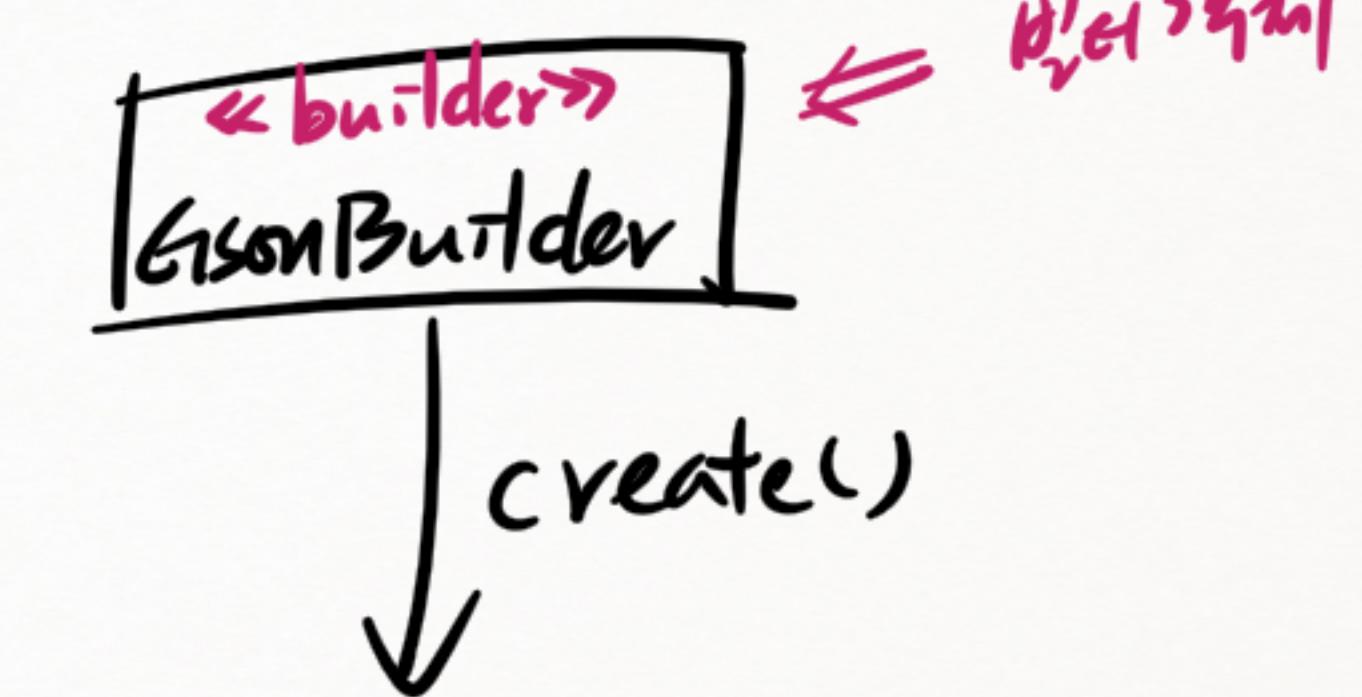
new Eison()



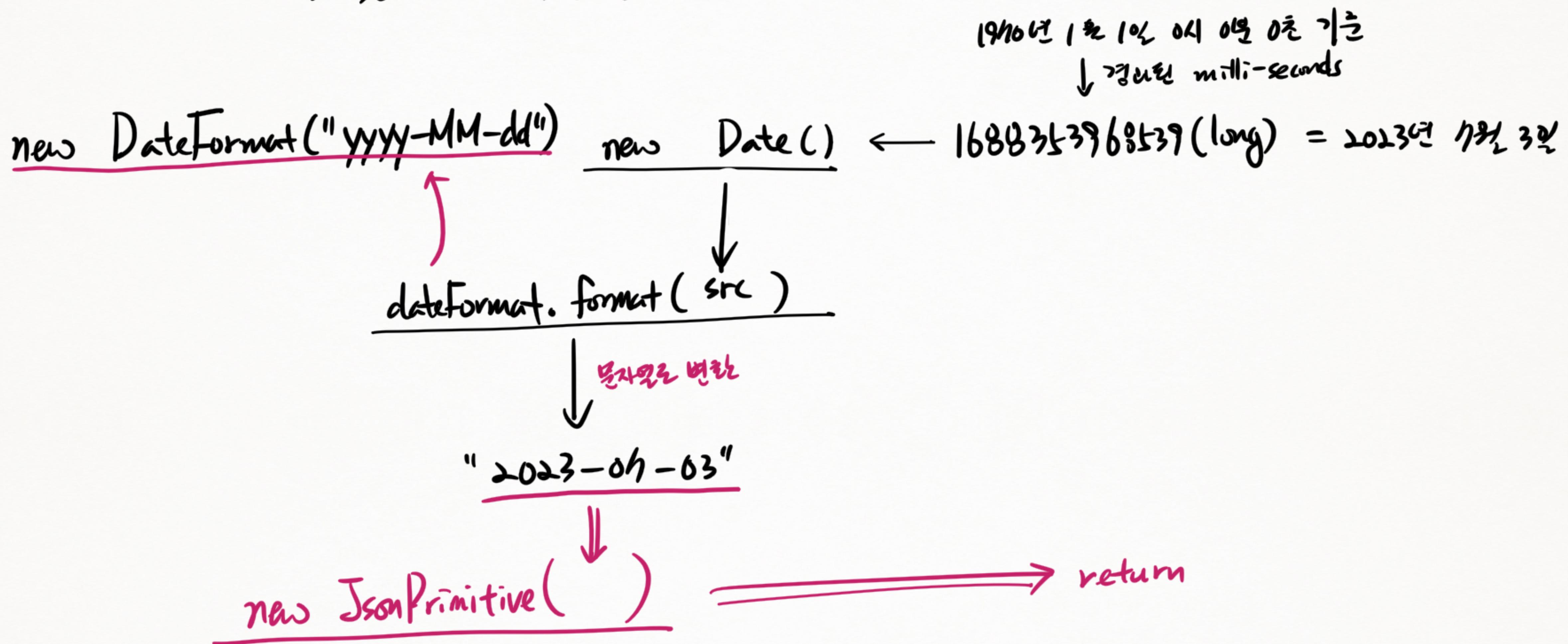
인스턴스 생성과정이 (복잡한 경우)  
반복문을 경우



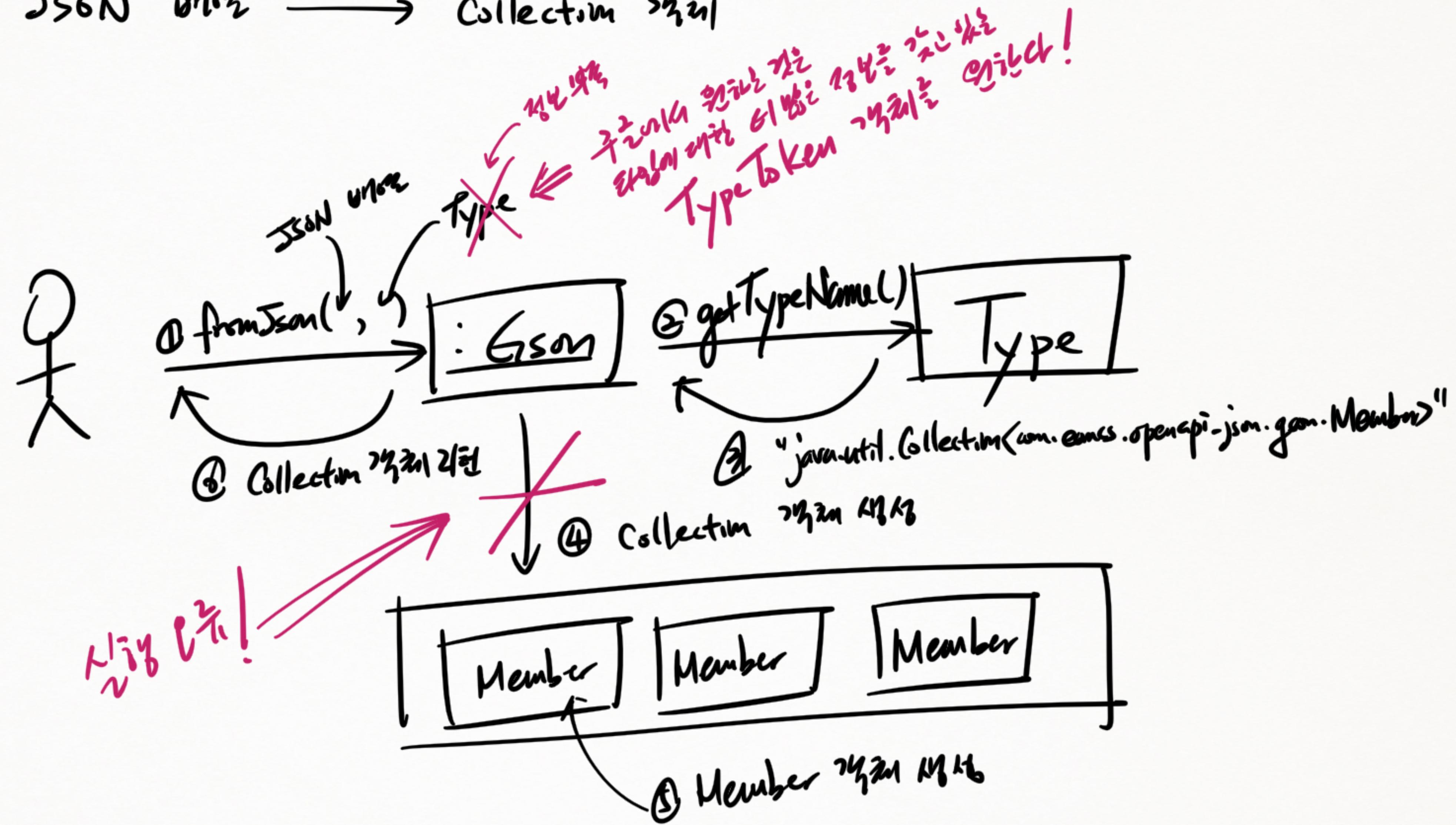
② 토끼의 낚지를 통해 인스턴스 생성



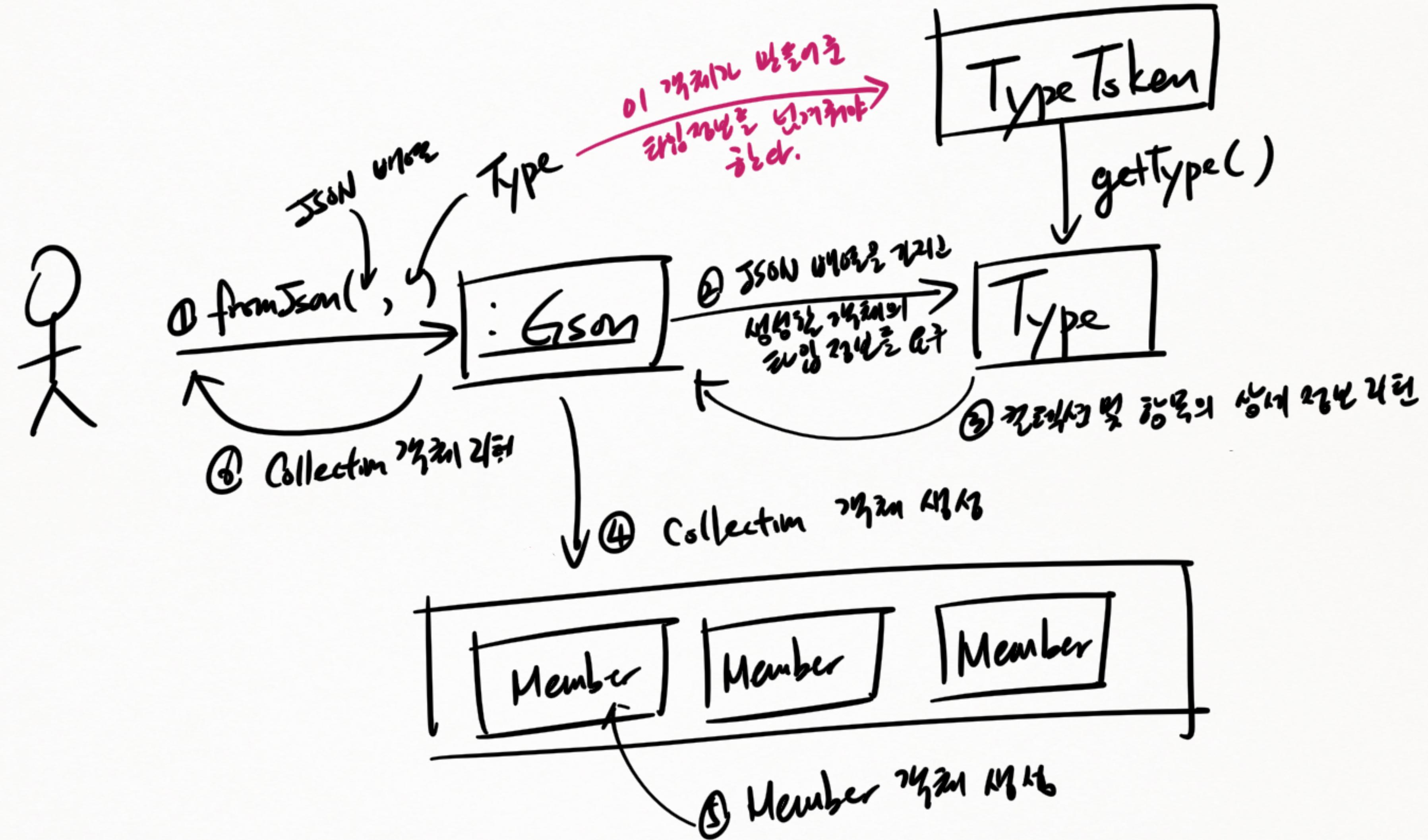
## \* JsonSerializer.serialize()



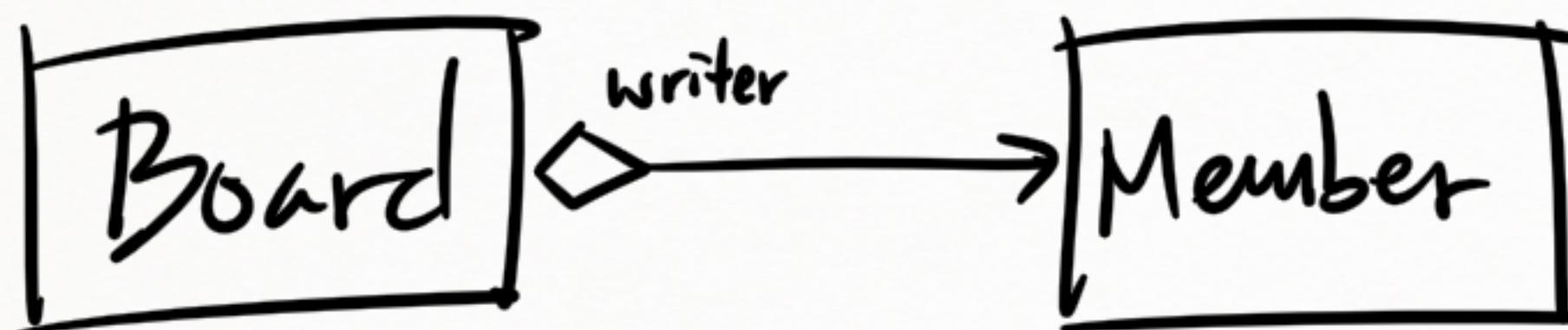
\* JSON 풀기 → Collection 풀기



\* JSON 풀기 → Collection 풀기



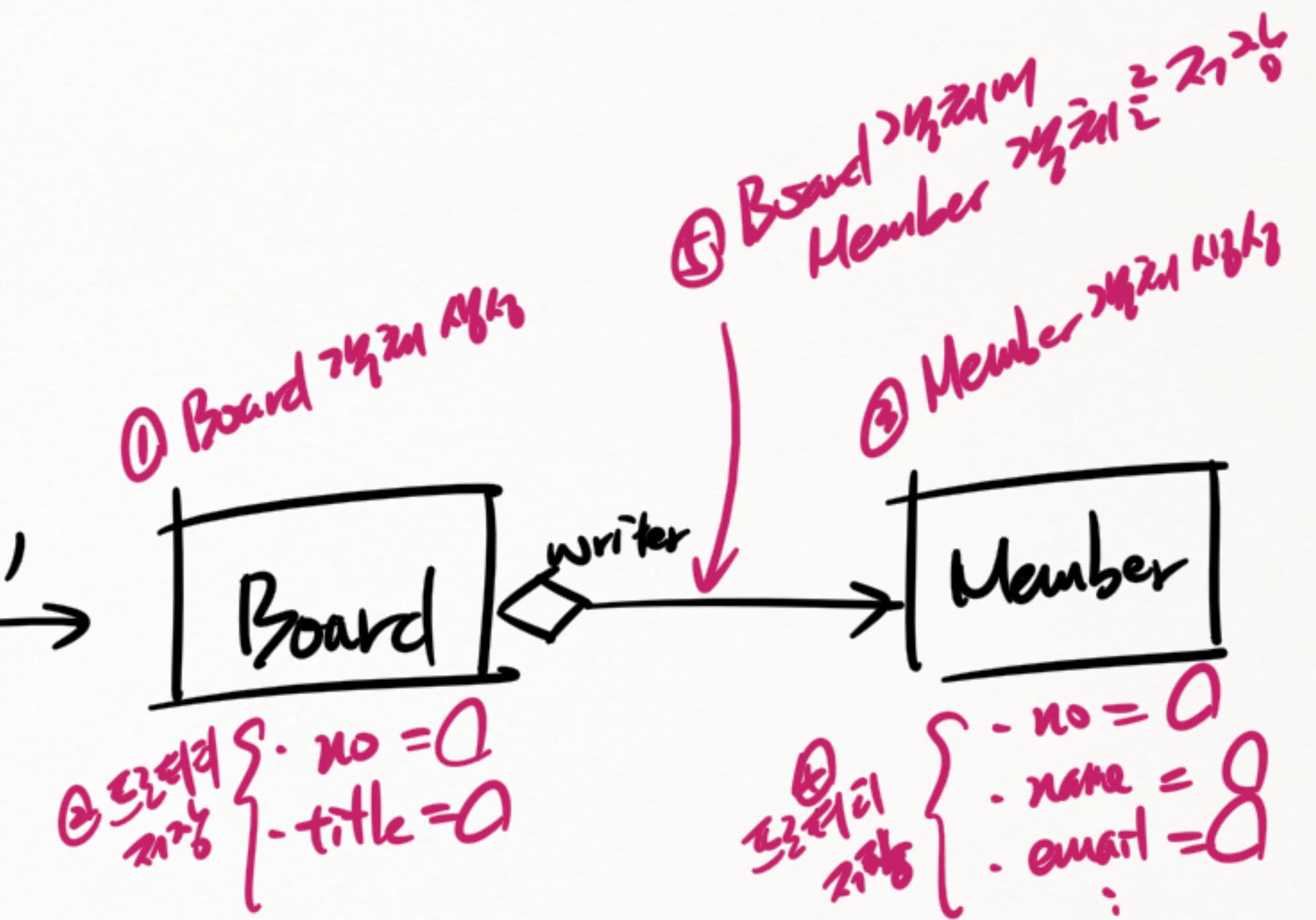
\* 다른 객체를 찾고 있는 경우



↓  
toJSON()

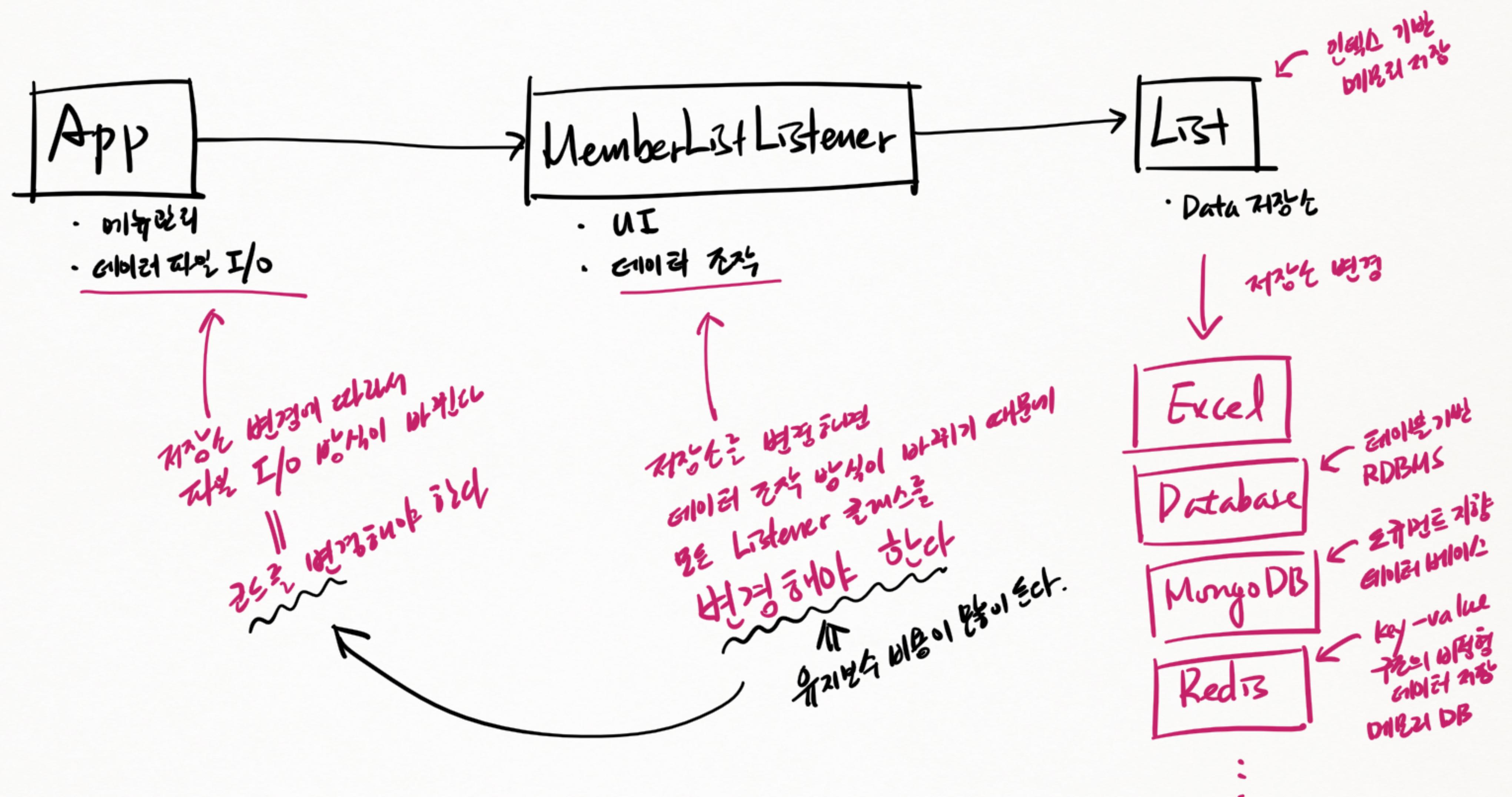
{  
 "no": 1,  
 "title": "—",  
 :  
 "writer": {  
 "no": 100,  
 "name": "김민경",  
 :  
 }  
}

fromJson()



36. DAO (Data Access Object) එක

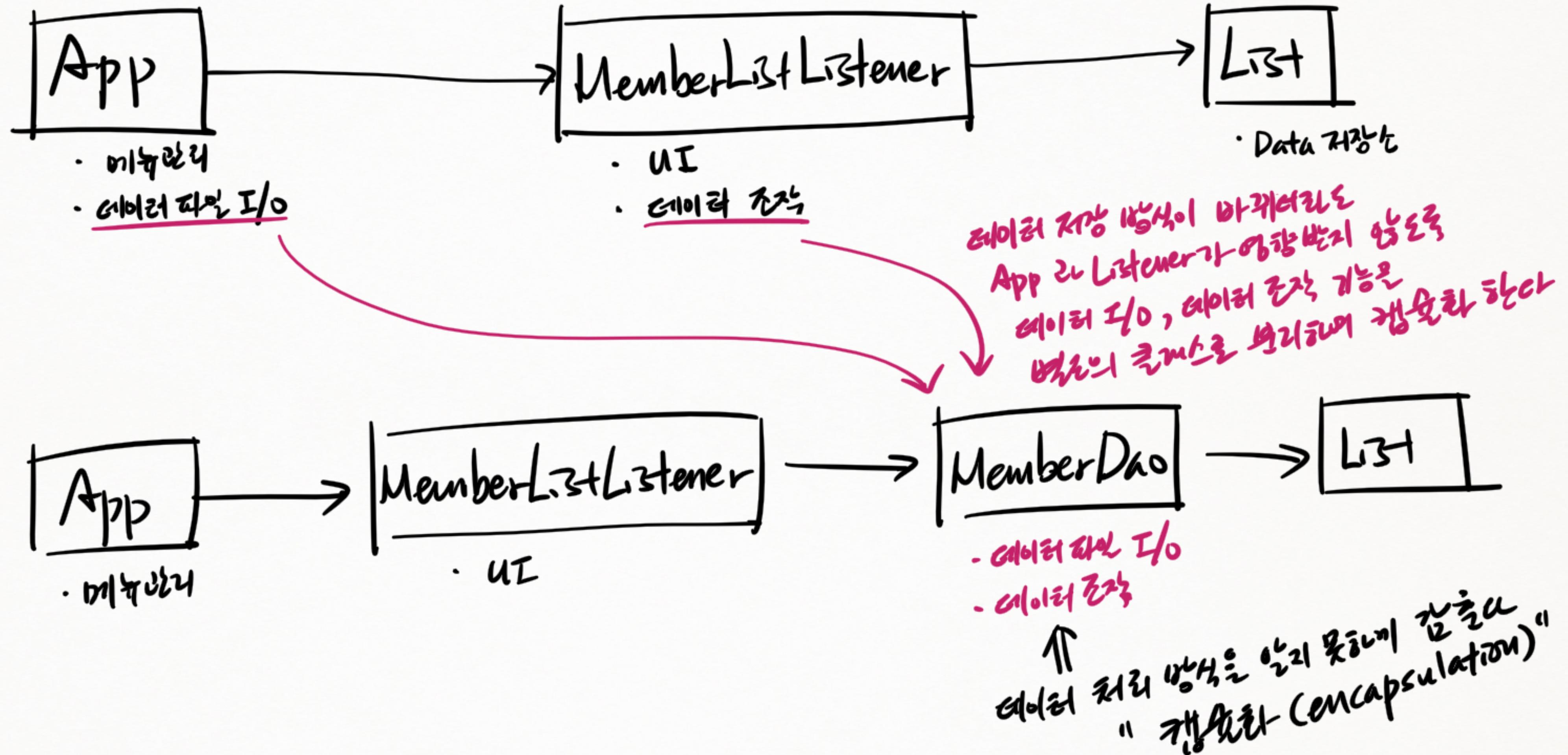
\* 현장



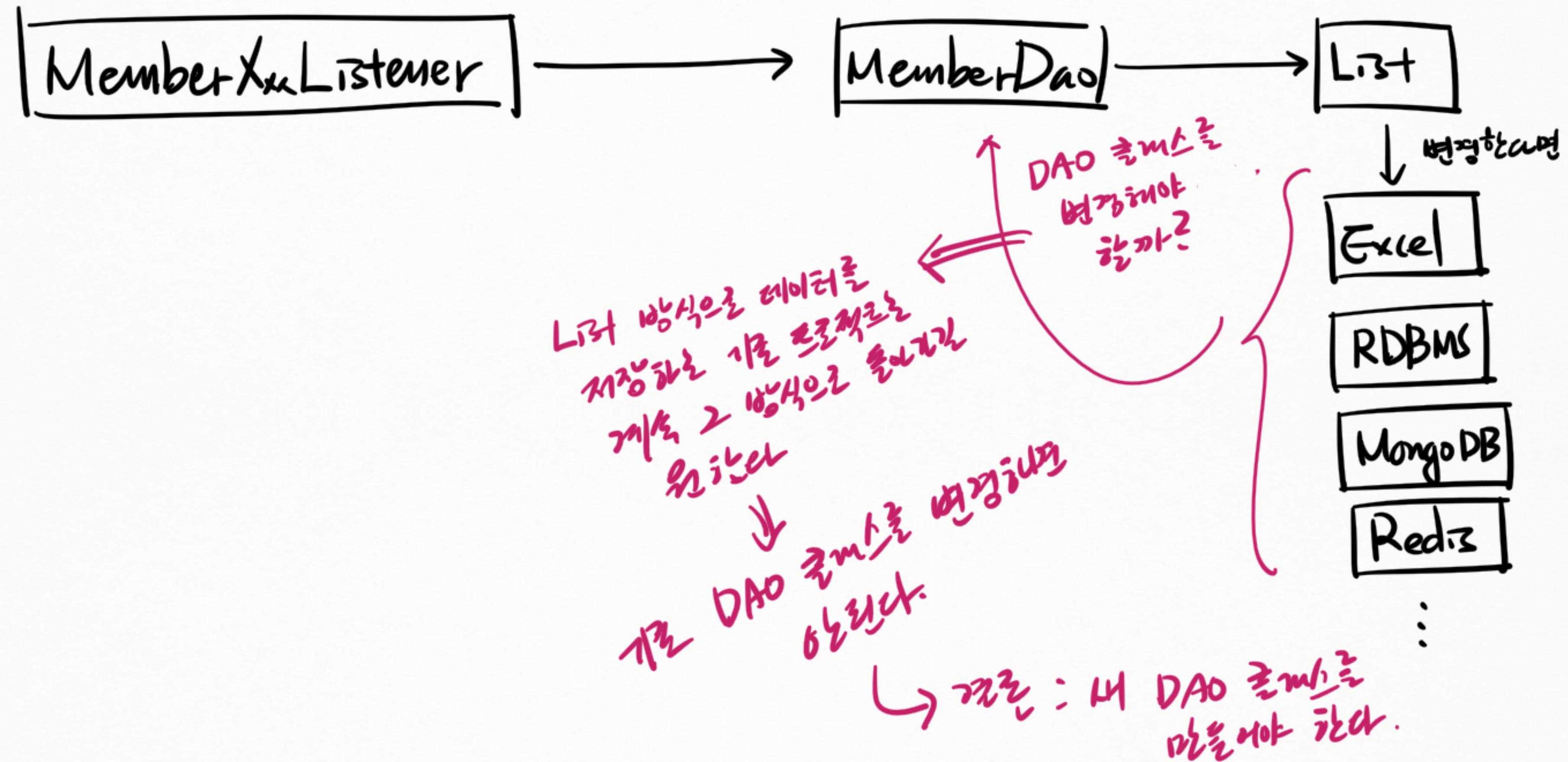
## 36. DAO (Data Access Object) 은?

↳ 데이터 저장 및 조작 단위

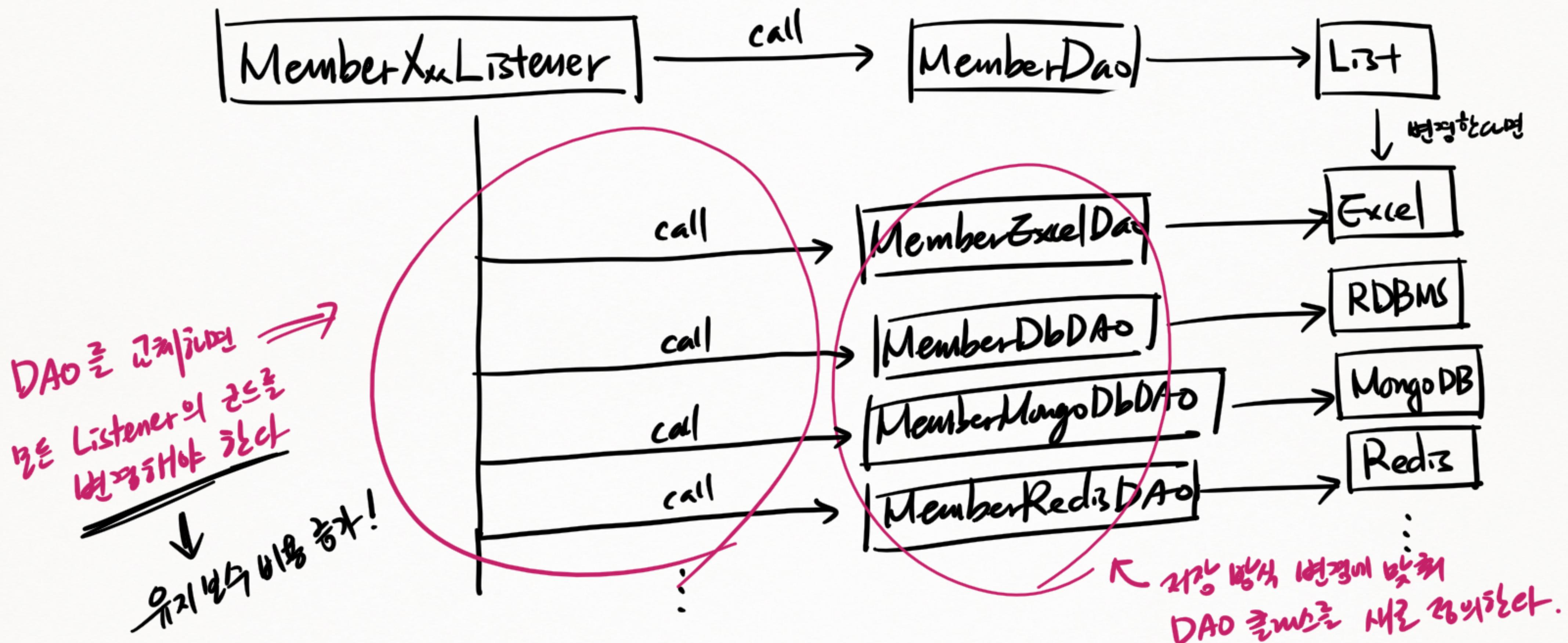
\* 개선



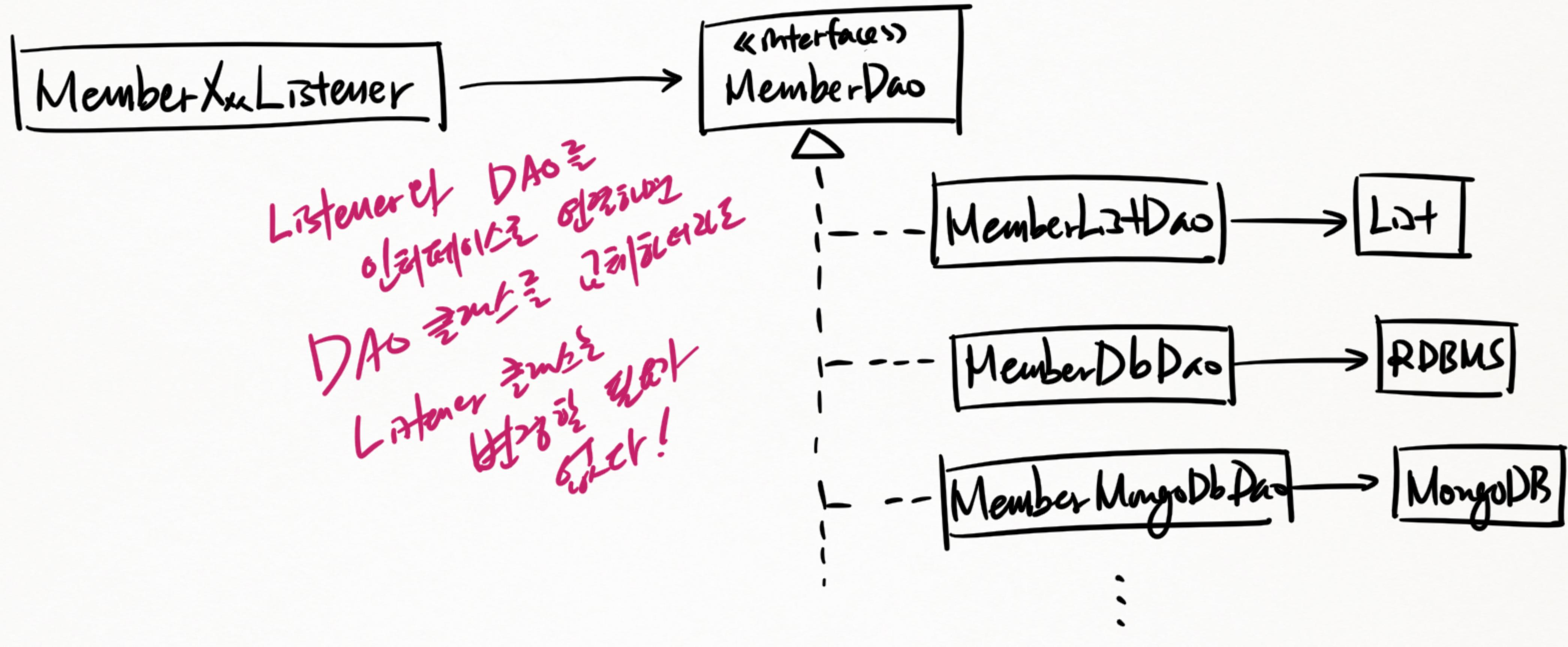
# DAO 만들기



# DAO 만들기

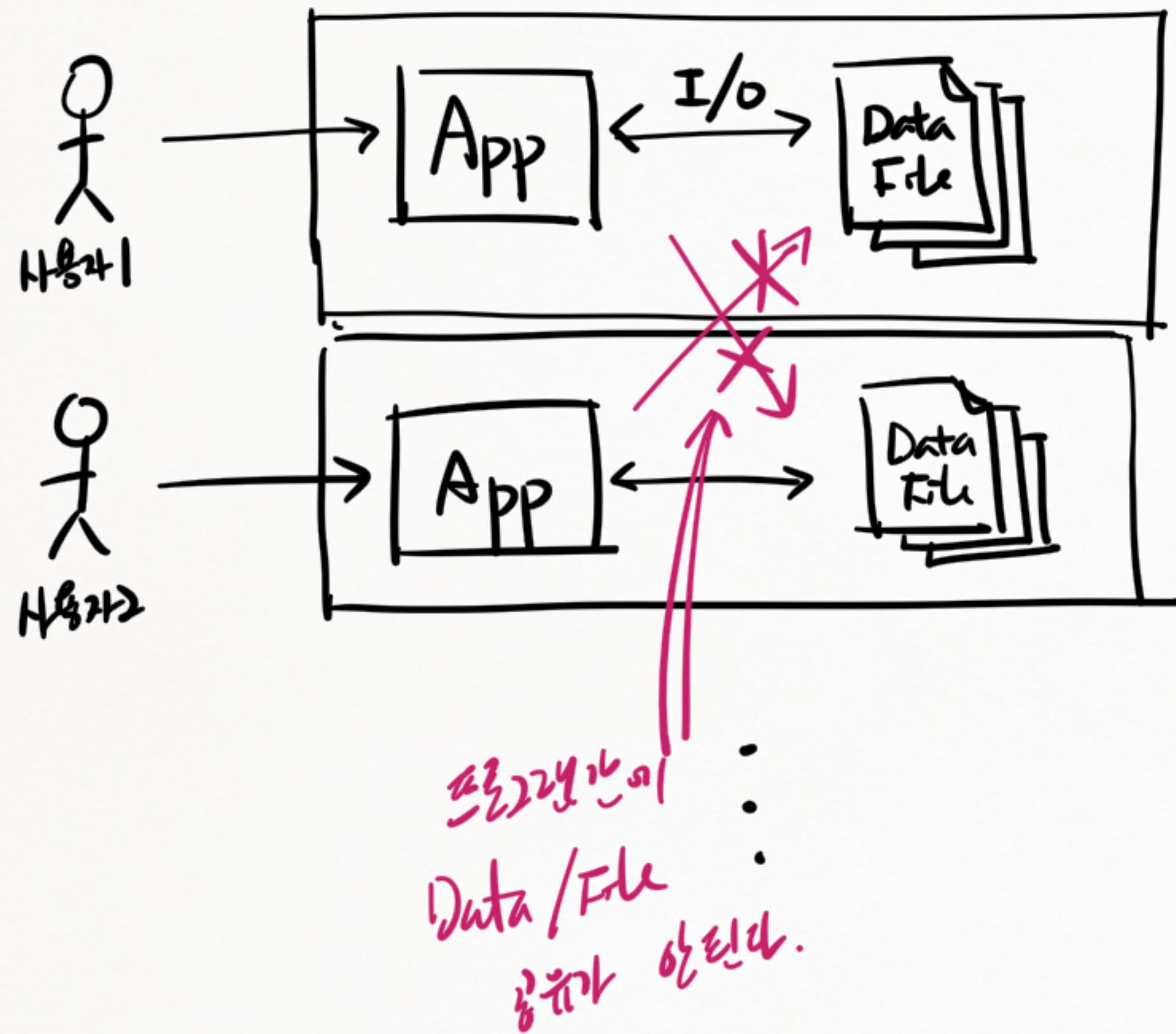


# DAO 만들기



## 37. 바이오킹을 통한 데이터 공유

### ① 현황



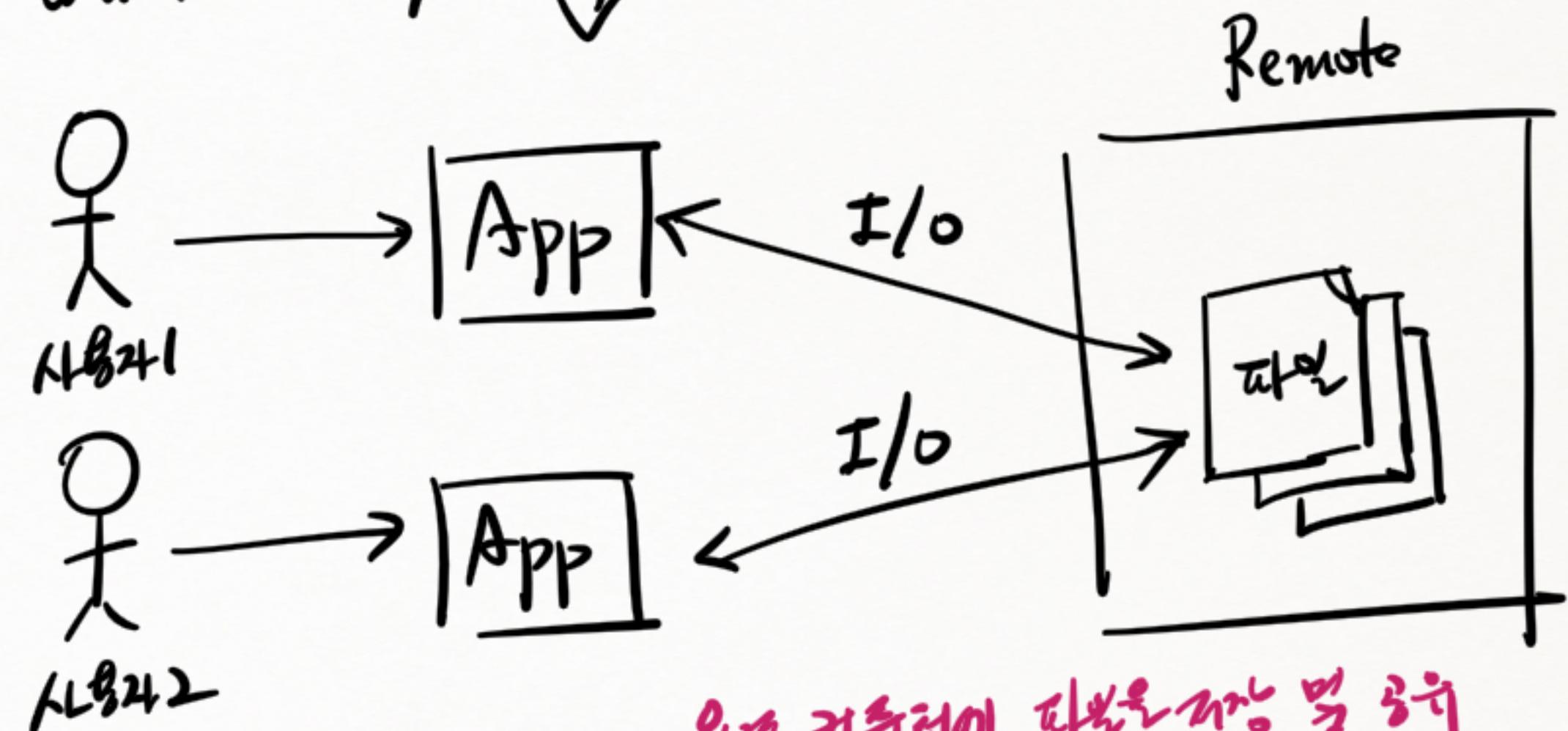
*App 을 실행하는 컴퓨터*

데이터 파일이 로컬에 저장된다.

↓

App 간에 데이터를 공유할 수 없다!

② 데이터 파일은  
별도의 컴퓨터에 분리/공유 ↓ 가능할까?

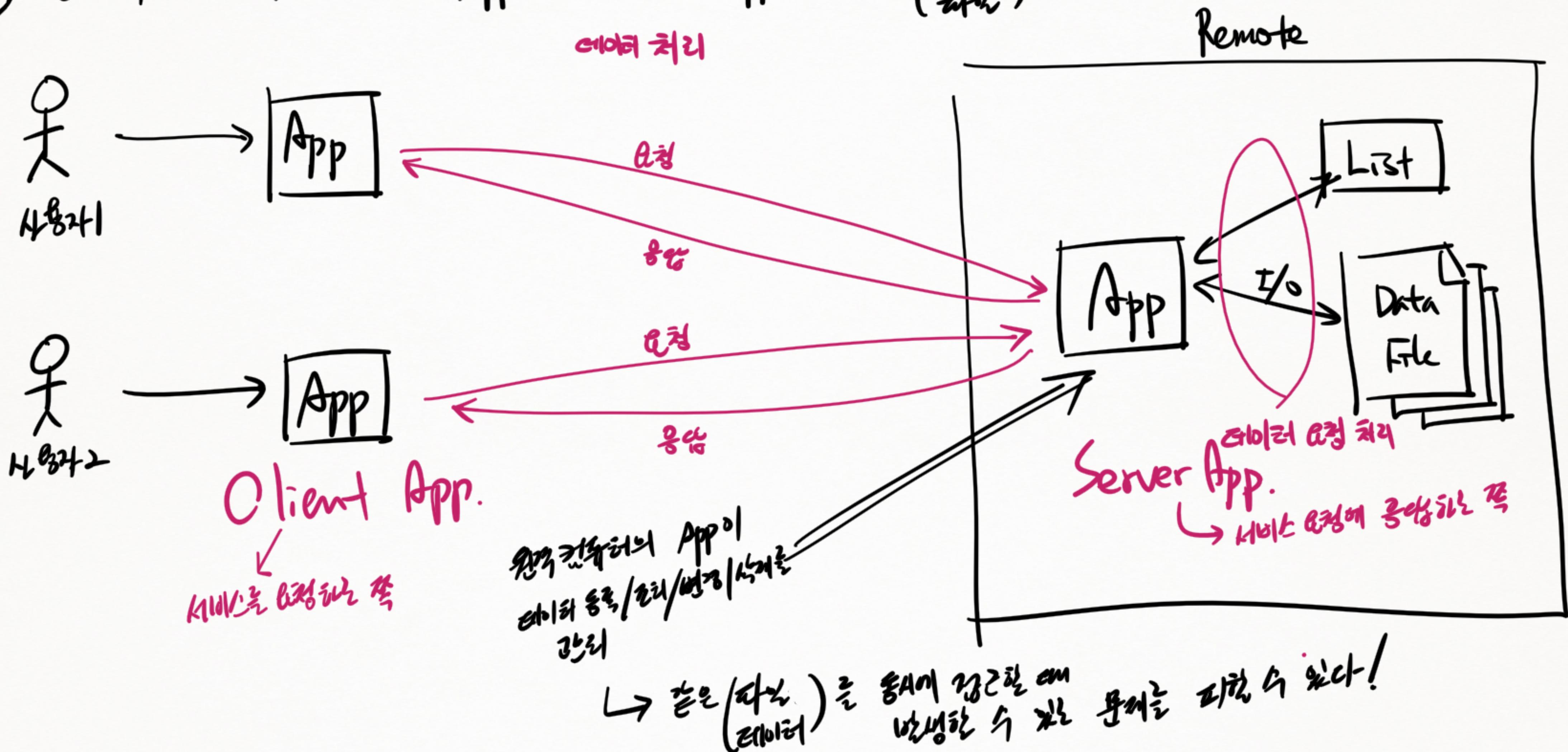


- 원격 컴퓨터에 파일을 저장 및 공유

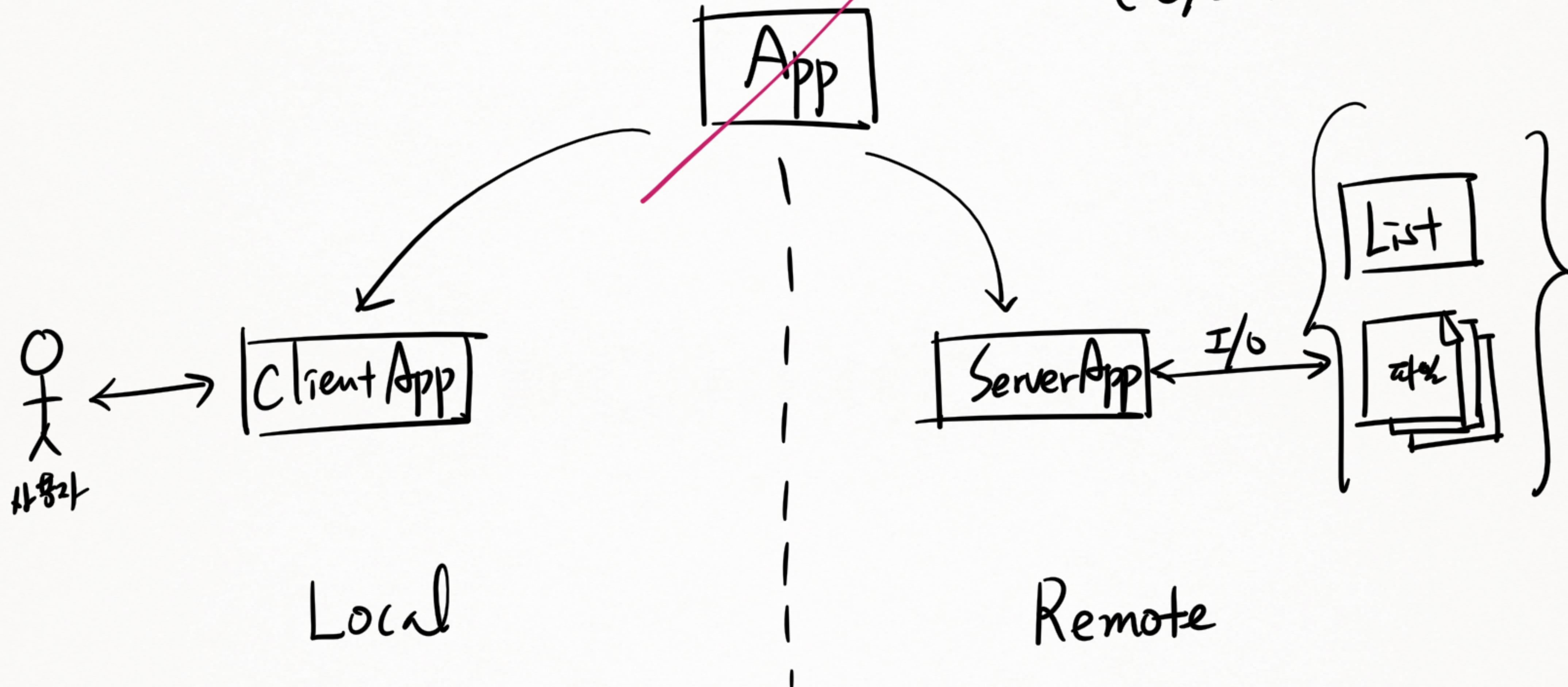
중재자

여기 App이 같은 파일을 편집하다 보면  
각자의 데이터가 깨끗해졌다.

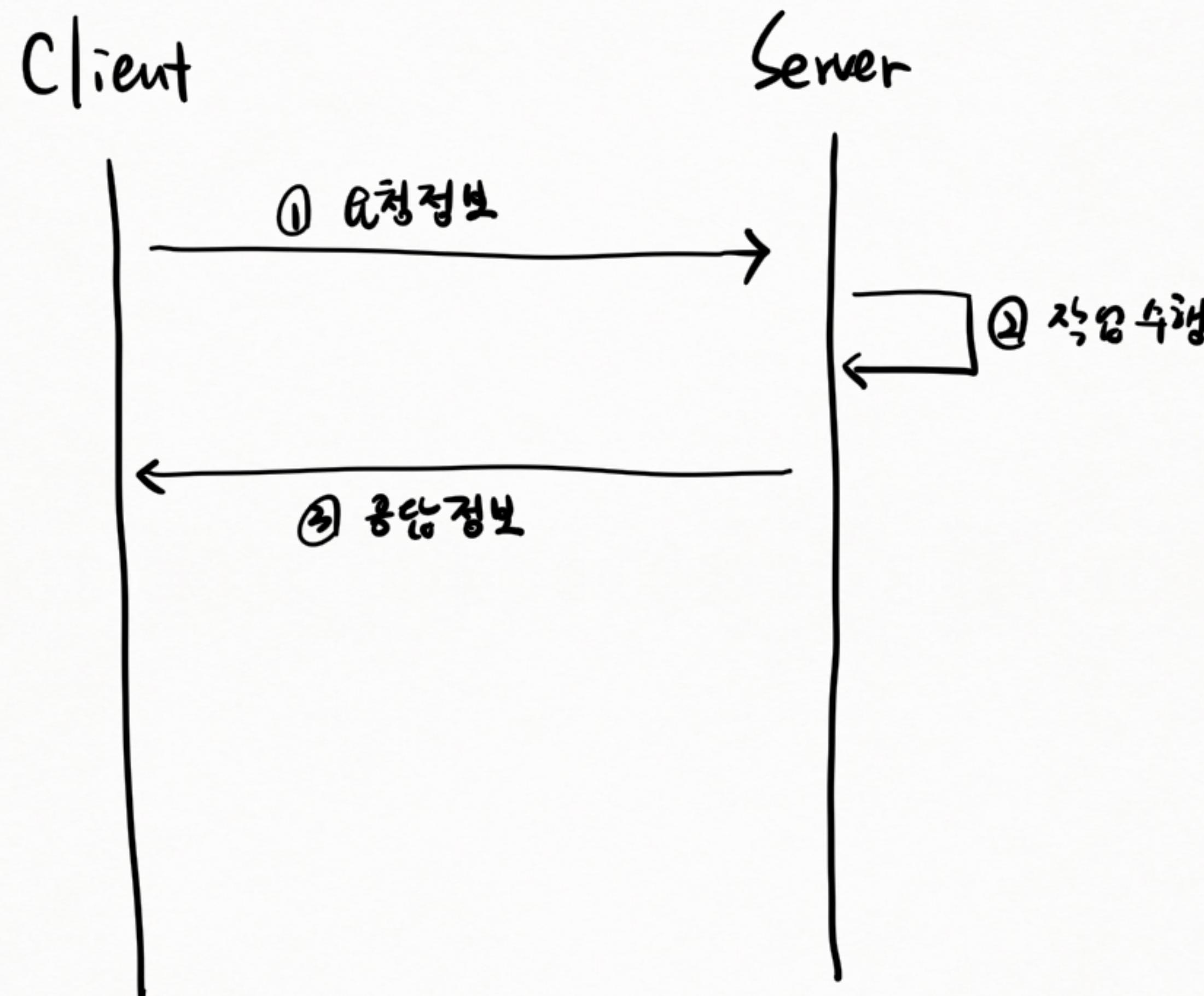
③ 데이터 관리 기능을 별도의 App. 으로 분리 - App이 직제 (데이터)  
를 접근하는 것을 막는다.



\* System Architecture : Client / Server Architecture  
(C/S Architecture)



\* client / server 통신 규칙(protocol)



\* client / server 통신 규칙 (protocol)

① 요청 정보 (JSON 문자열)

```
{  
  "command": "레이타이머 / 명령",  
  "data": "JSON 문자열"  
}
```

↓ 예

```
{  
  "command": "board/insert",  
  "data": {"title": "...",  
           "content": "...",  
           ... :  
           }  
}
```

"레이타이머 / 명령"

기사로: board  
회원: member  
독서록: reading

서버의 DAO 메서드명.

JSON 문자열

command 값: 일반 문자열  
data 값: JSON 문자열

- \* client / server 통신 규칙 (protocol)

## ② 응답 정보 (JSON 문자열)

```
{  
    "status": "실행결과",  
    "result": "JSON 문자열"  
}
```

↓ 성공 예

```
{  
    "status": "success",  
    "result": "[{"no": 12, ... }]"  
}
```

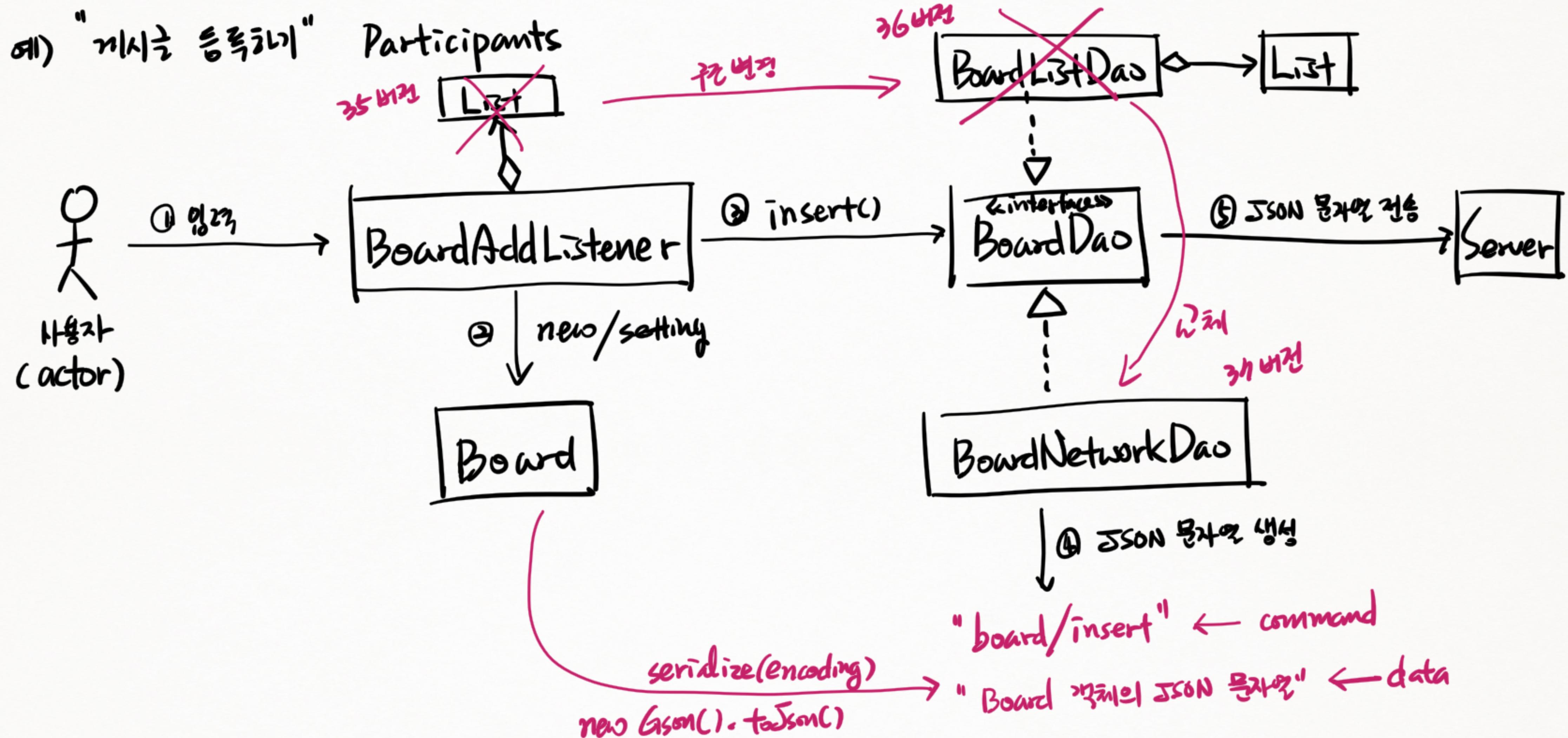
- \* 실행 결과  
"success": 성공  
"failure": 실패

↓ 실패 예

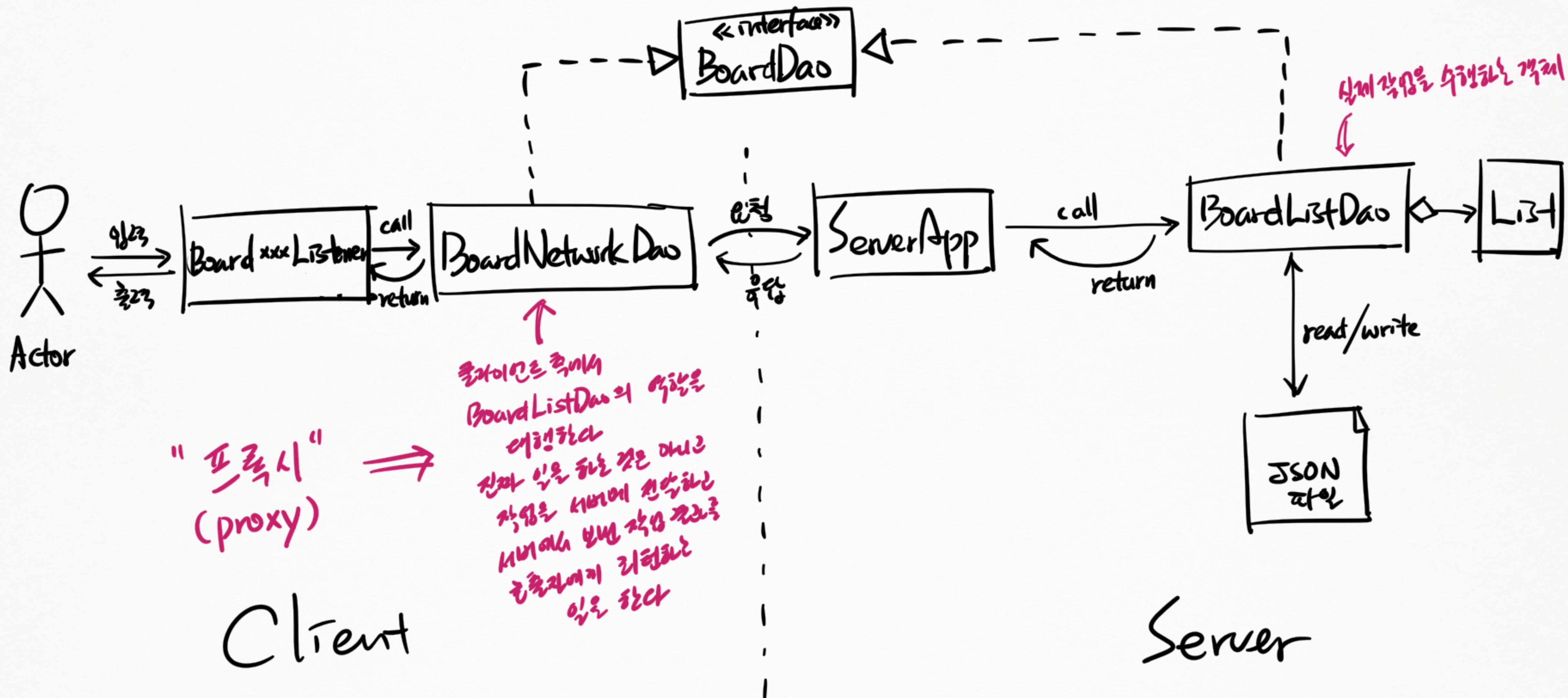
```
{  
    "status": "failure",  
    "result": "해당 번호의 데이터가 없습니다!"  
}
```

\* 요청 정보 뷰티가 작업에 참여하는 객체들과 실행흐름

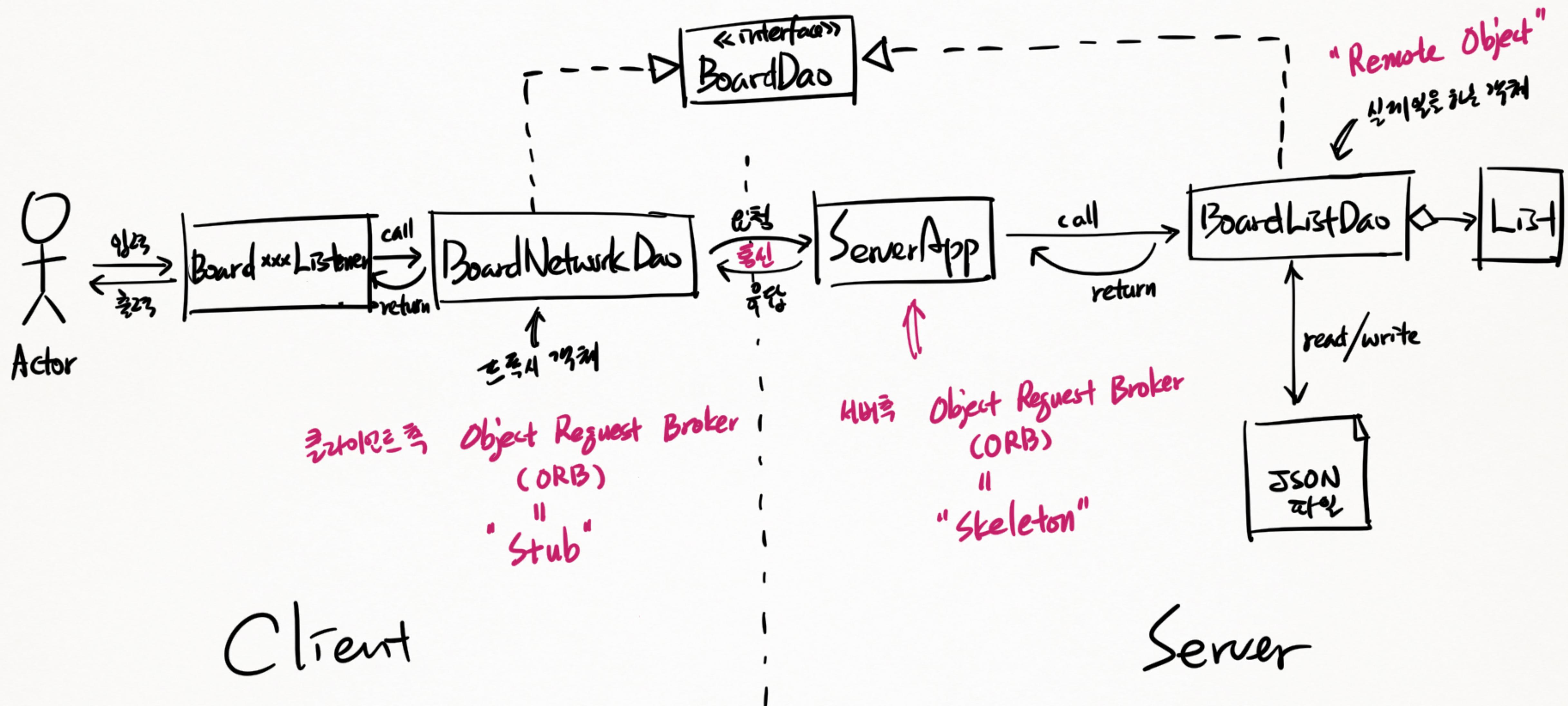
예) "게시글 등록하기" Participants



\* DAO 와 Proxy 패턴 (GoF)

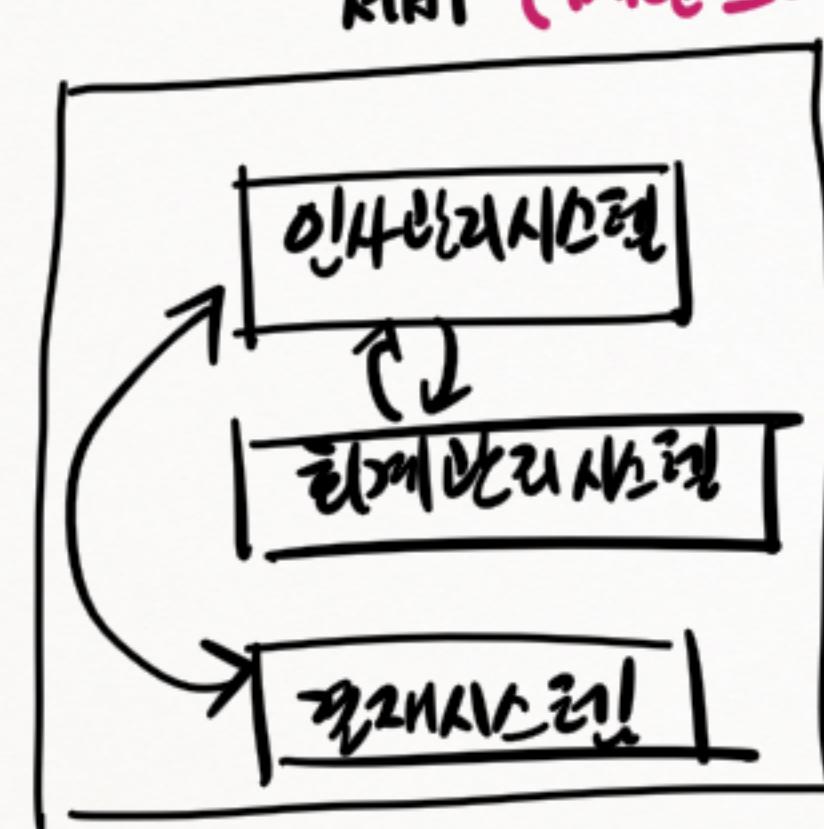


\* DAO 와 Proxy 패턴 (GoF) ↗ 예술자의 명칭



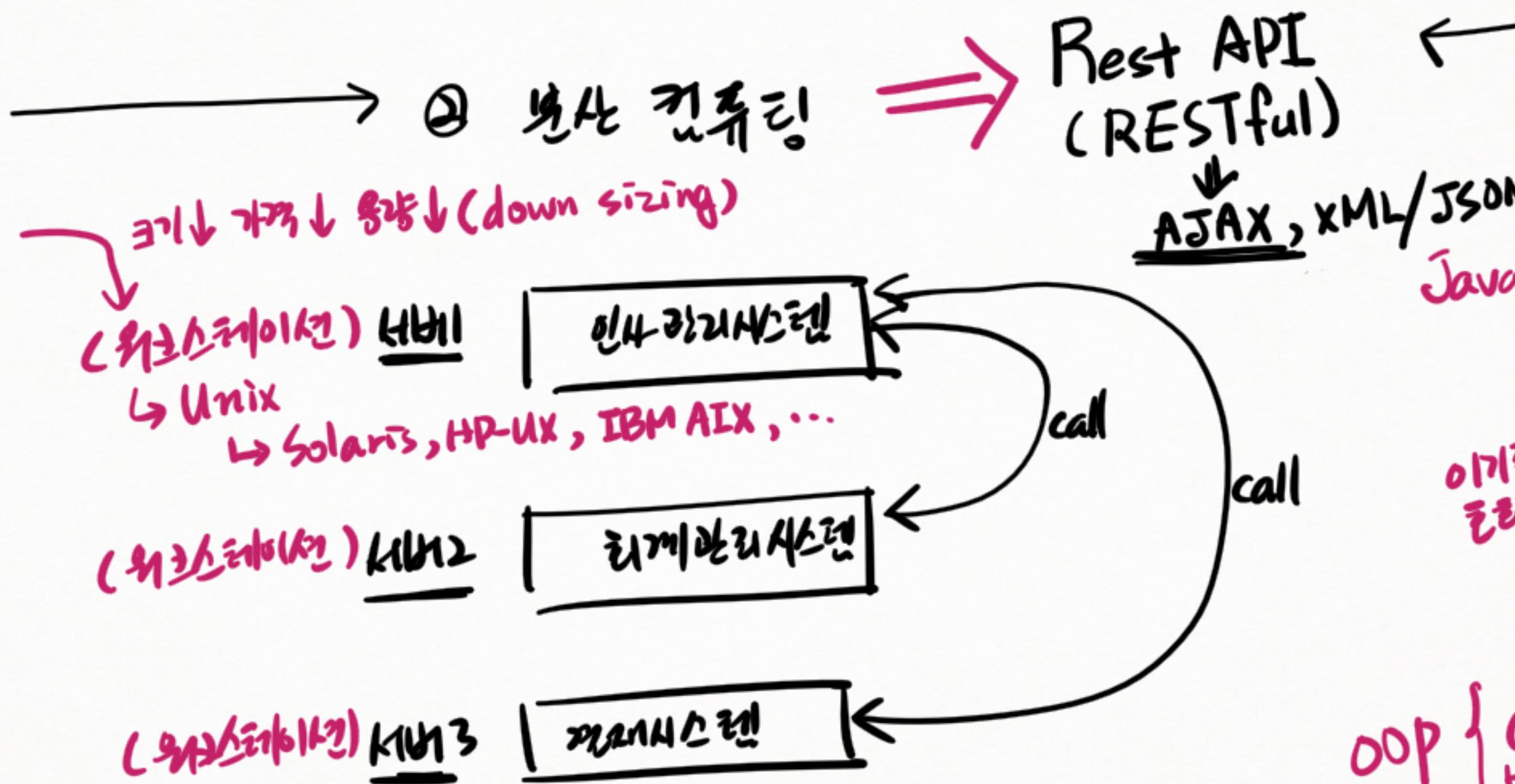
## \* 온라인 컴퓨팅

### ① 중앙 집중식 컴퓨팅



문제점

### ② 온라인 컴퓨팅



문제점

## Rest API (RESTful)

AJAX, XML/JSON

Java

EJB

이기종 호환

CORBA (Common ORB Architecture)  
· IIOP

RMI (Remote Method Invocation)

C RPC (Remote Procedure Call)

해당 없음

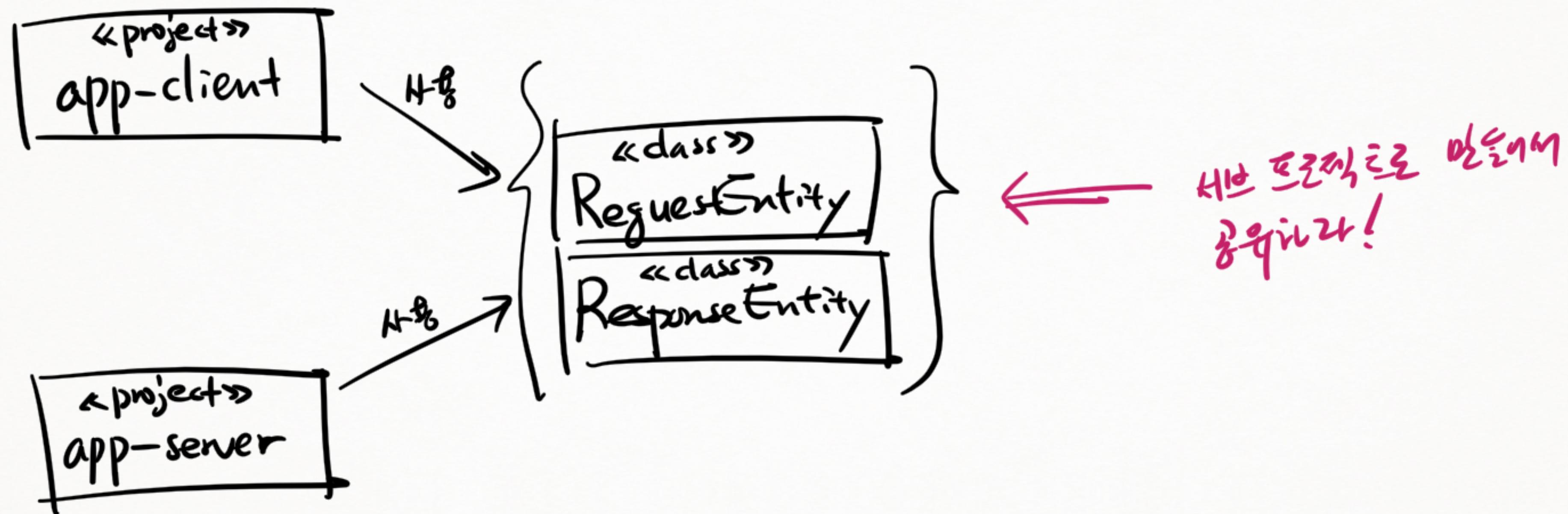
- 전용 API

Web Service

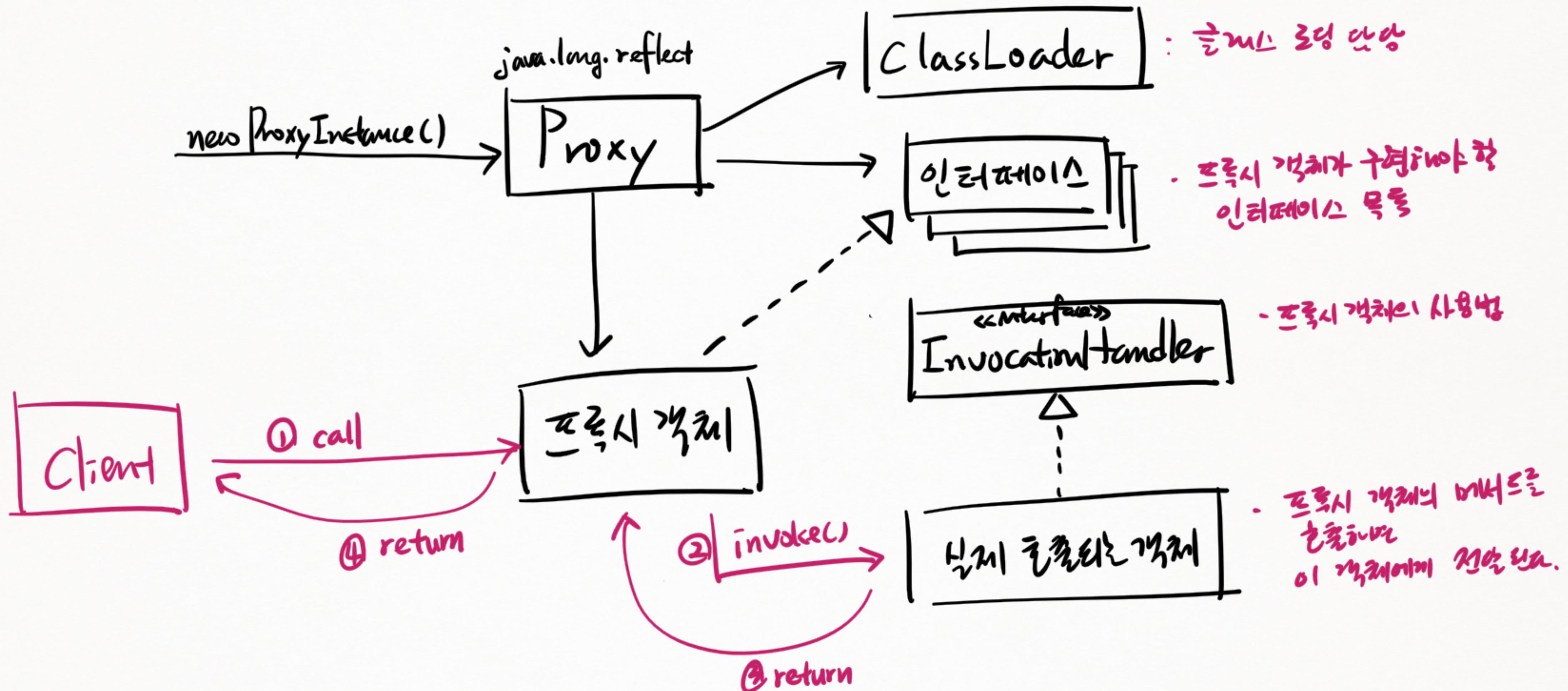
- WSDL
- SOAP
- UDDI
- XML

Web  
HTTP

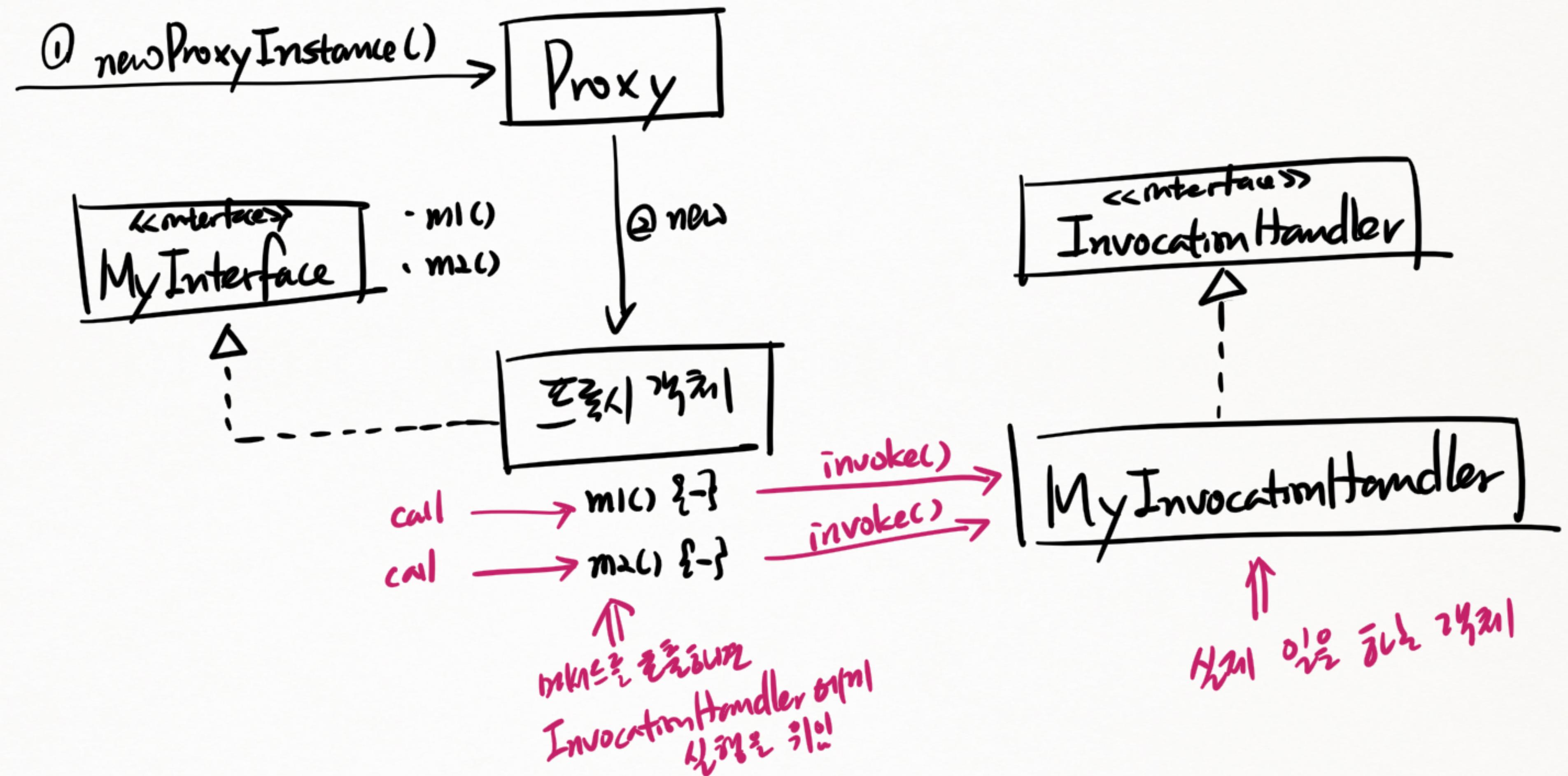
\* Project 간에 헤더 공유하기



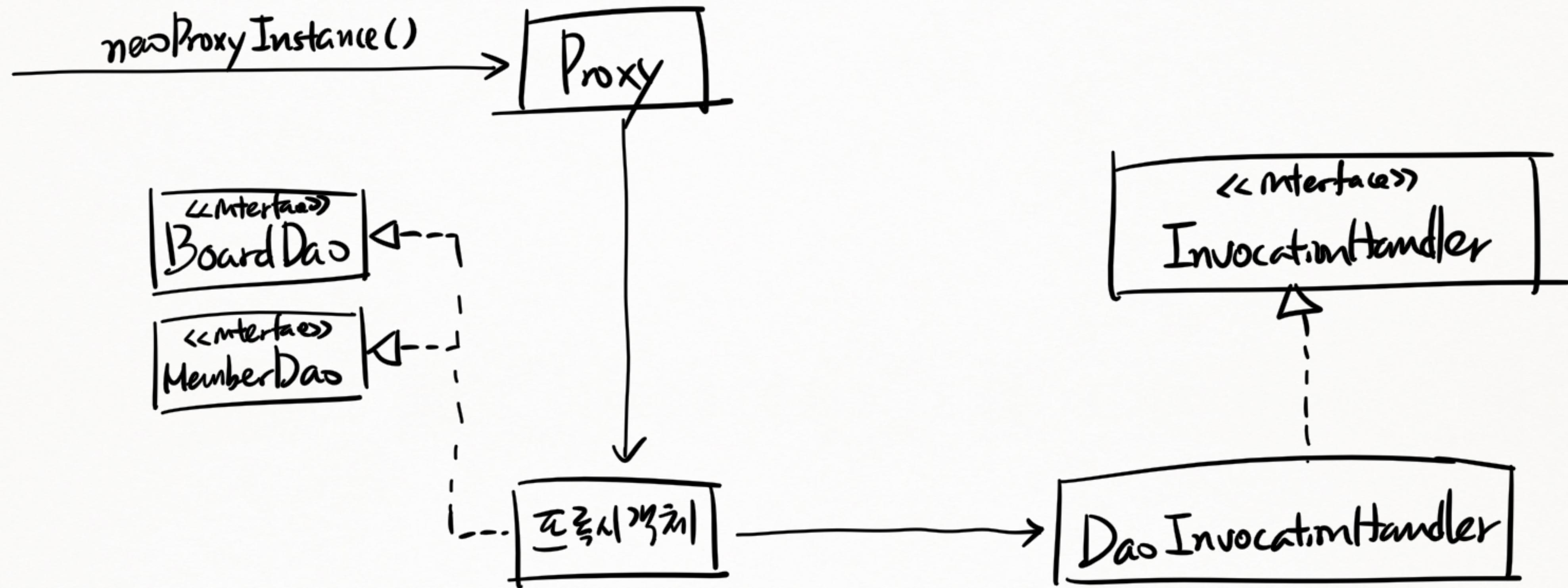
### 3.8. 프록시 객체 자동 생성



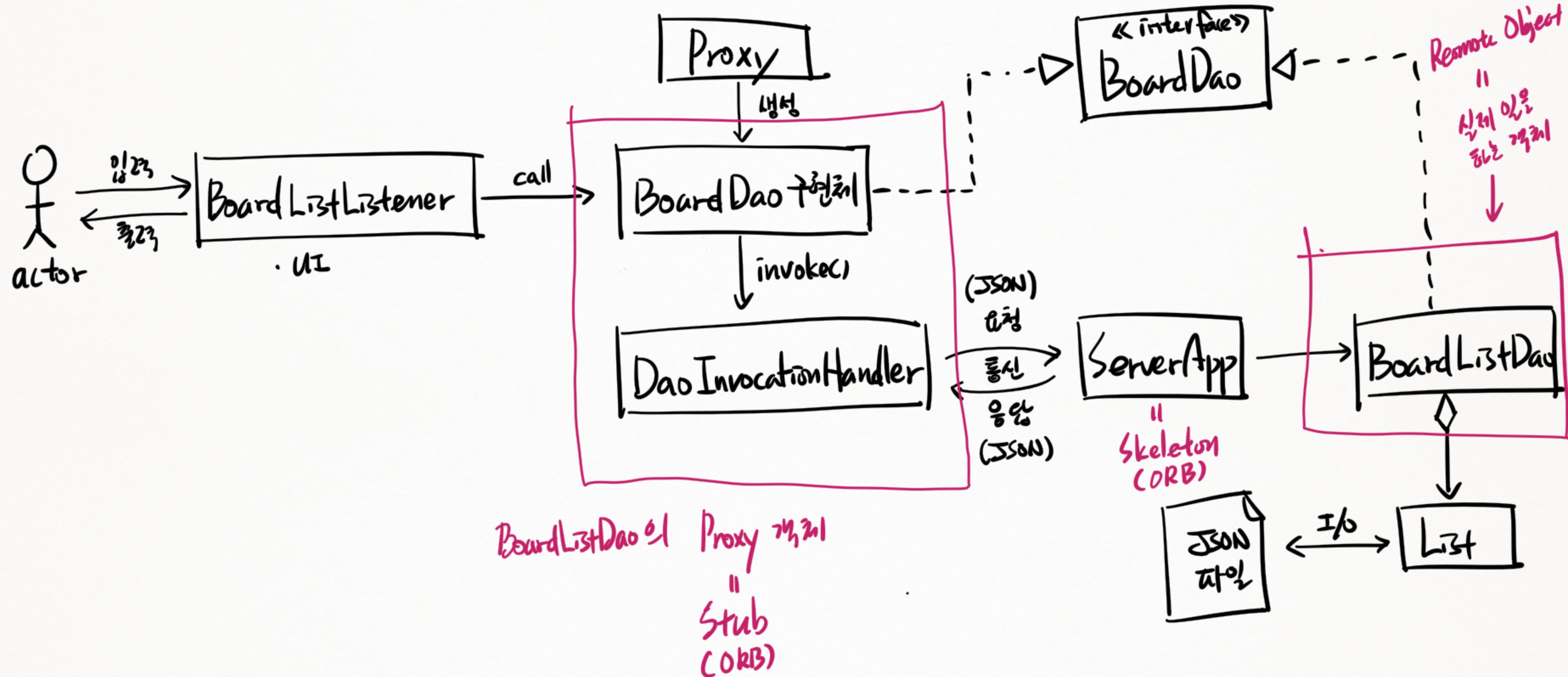
## 38. 프록시 객체 사용 예제 - ②



## 38. 프록시 패턴의 사용 예제 - Participants



## 38. 프록시 개념의 적용 예제 - Participants II



### 38. 데이터 전송/수신 기능을 활용화하기

## ① 현황

개선

↳ 통신 기능을 키워드화하여 100%의 클래스로 분리한다.

