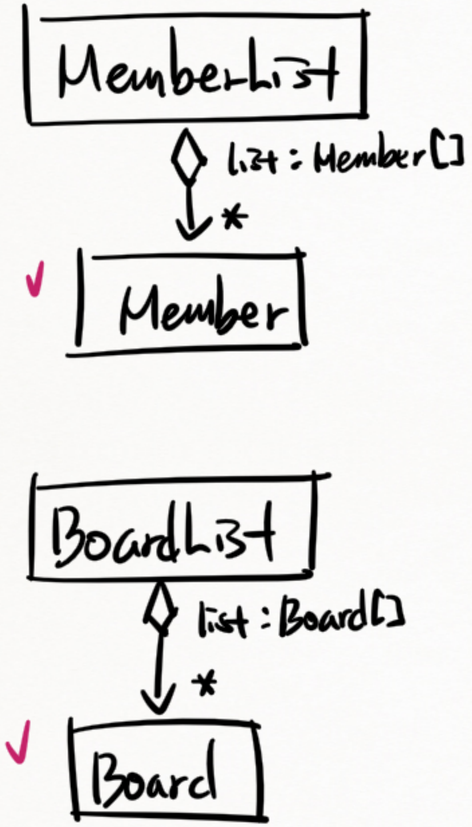
24. 제네릭(Generic) 적용 : (Object 타입처럼 다양한 타입에 대응할 수 있다. 특정타입으로 제한할 수 있다.)

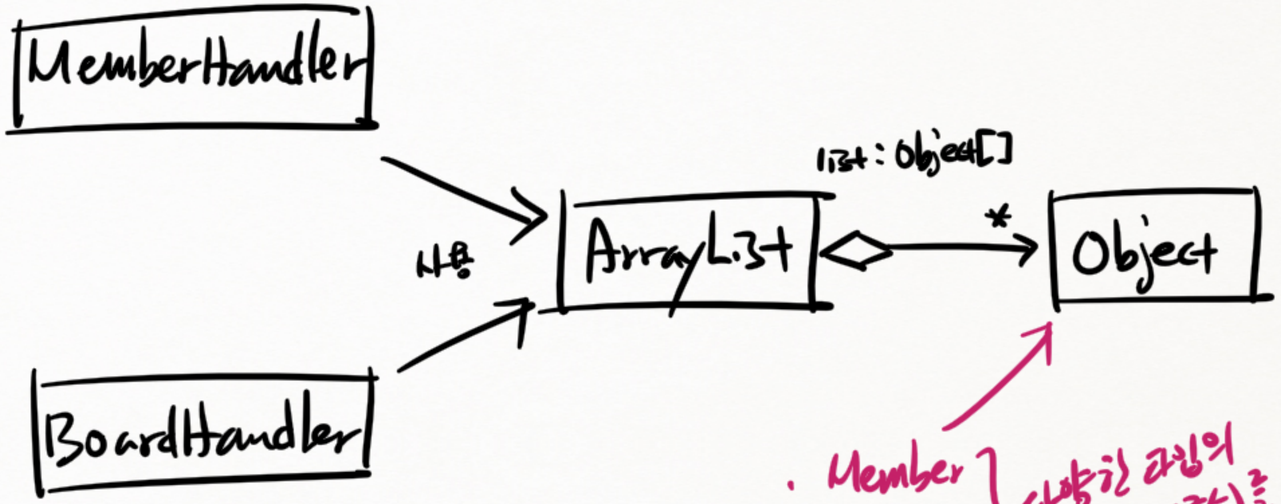
① 다형적 변수 적용 전 → ② 다형적 변수 적용

↳ 마치 각 타입별로 클래스를 정의한 듯한 효과를 볼 수 있다.

↳ 각 타입별로 List 클래스를 만든다



List의 기능은 같다
타입만 다르다
↓
코드 중복 발생



· Member
· Board
⋮

} 다양한 타입의 인스턴스(객체)를 저장할 수 있다.

* 단점 :

① 인스턴스를 꺼낼 때마다 형변환 필요!

② 특정 타입만 다루도록 제한할 수 없다.

⟸ 타입 별로 List를 만들 필요가 없다!

↳ 타입 안정성을 해친다.

* Generic ⟹ Type Parameter

클래스나 다른 타입 정보를 넘긴다

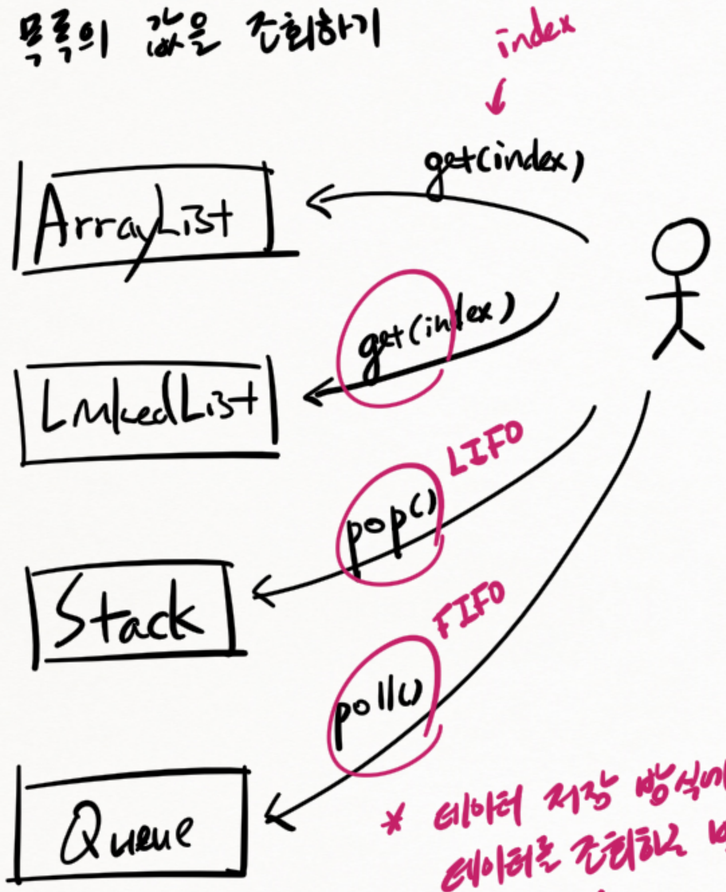
타입이름을 받을 변수 = "Type Parameter"

```
class    ArrayList<What> {

    public void add(What obj) {
        ≡
    }
    public What get(int index) {
        ≡
    }
    ⋮
}
```

25. Iterator 패턴을 활용하여 데이터 조회기능을 객체로 추출하기
→ 범용화하기

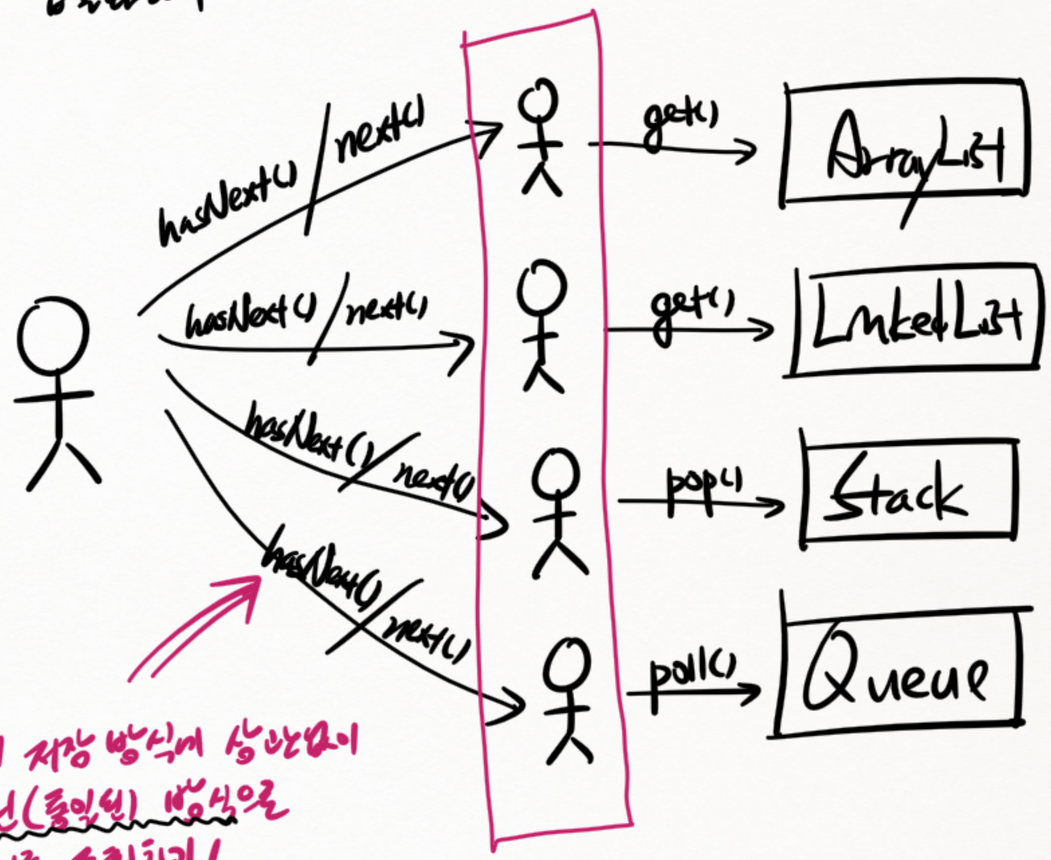
① 목록의 값을 조회하기



* 데이터 저장 방식에 따라 데이터를 조회하는 방법이 다르다!

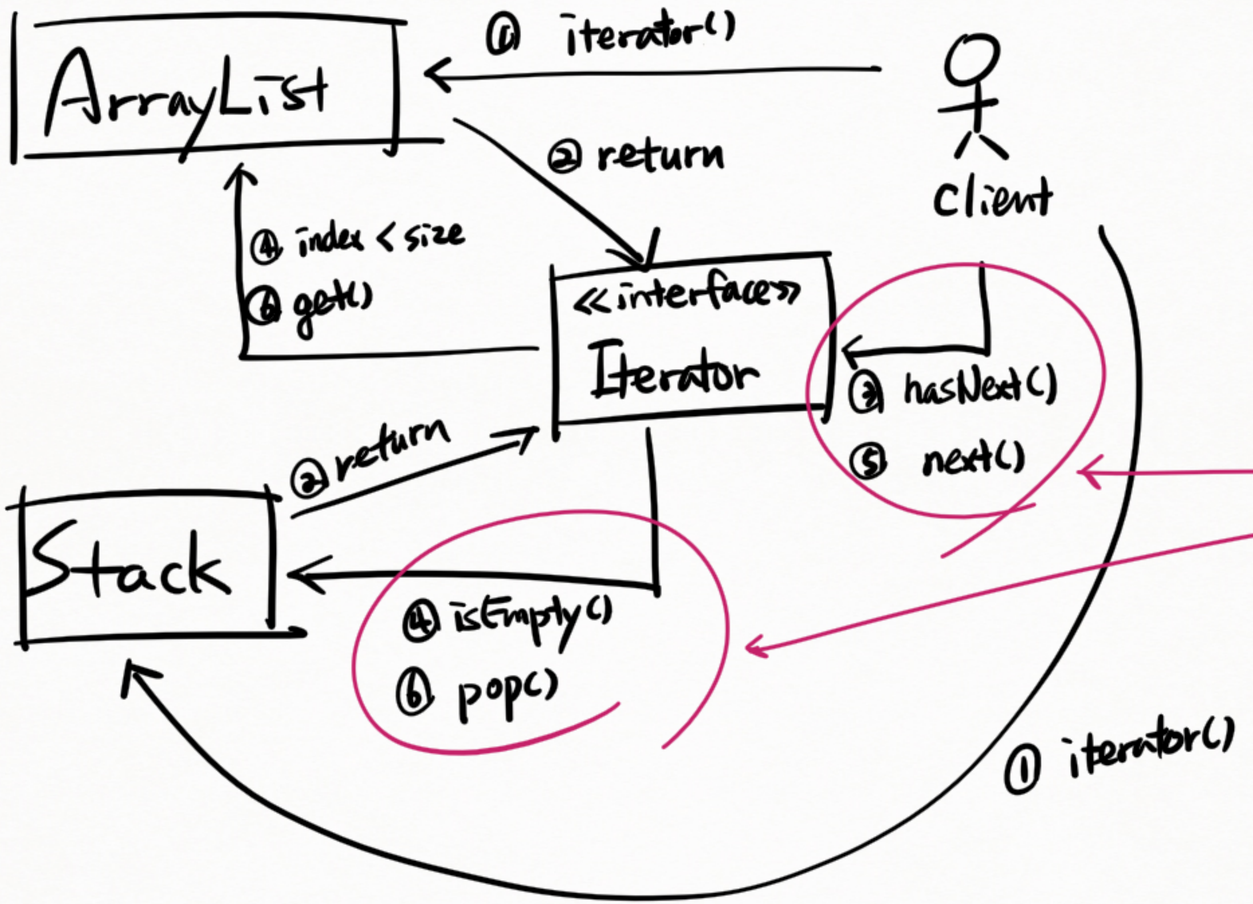
⟹

* 데이터 저장 방식에 상관없이 일관된(통일된) 방식으로 데이터를 조회하기!



Iterator
(데이터를 꺼내주는 일을 하는 객체)

* Iterator mechanism (구동원리)



* client

① network 분야

↳ 서버에 요청하는 역할을 수행 S/W

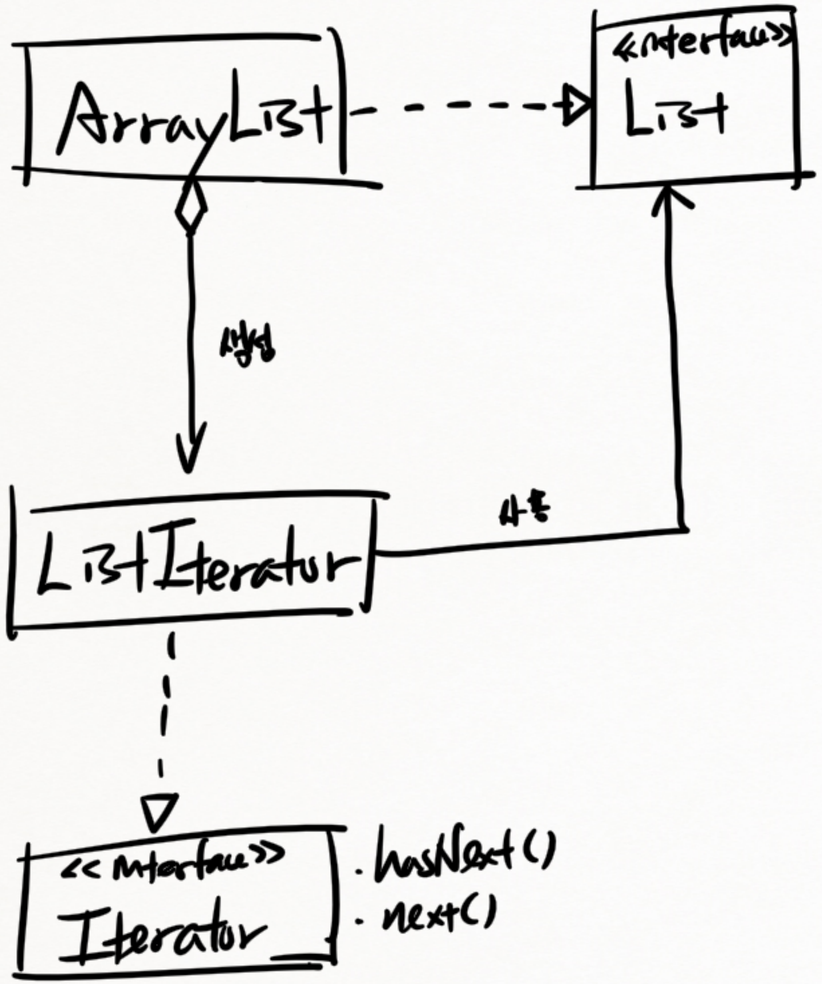
② OOP 분야

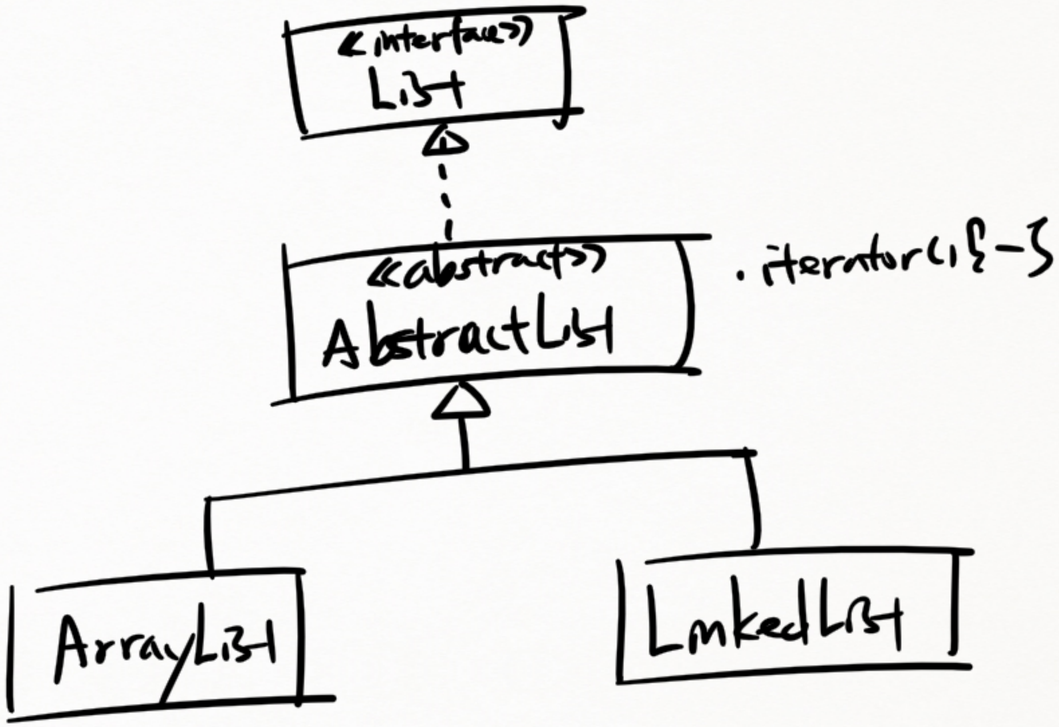
↳ 다른 객체를 사용하는 객체.

저장방식이 상관없이
일관된 방식으로
값을 꺼낼수 있다!

* Iterator 패턴 구현

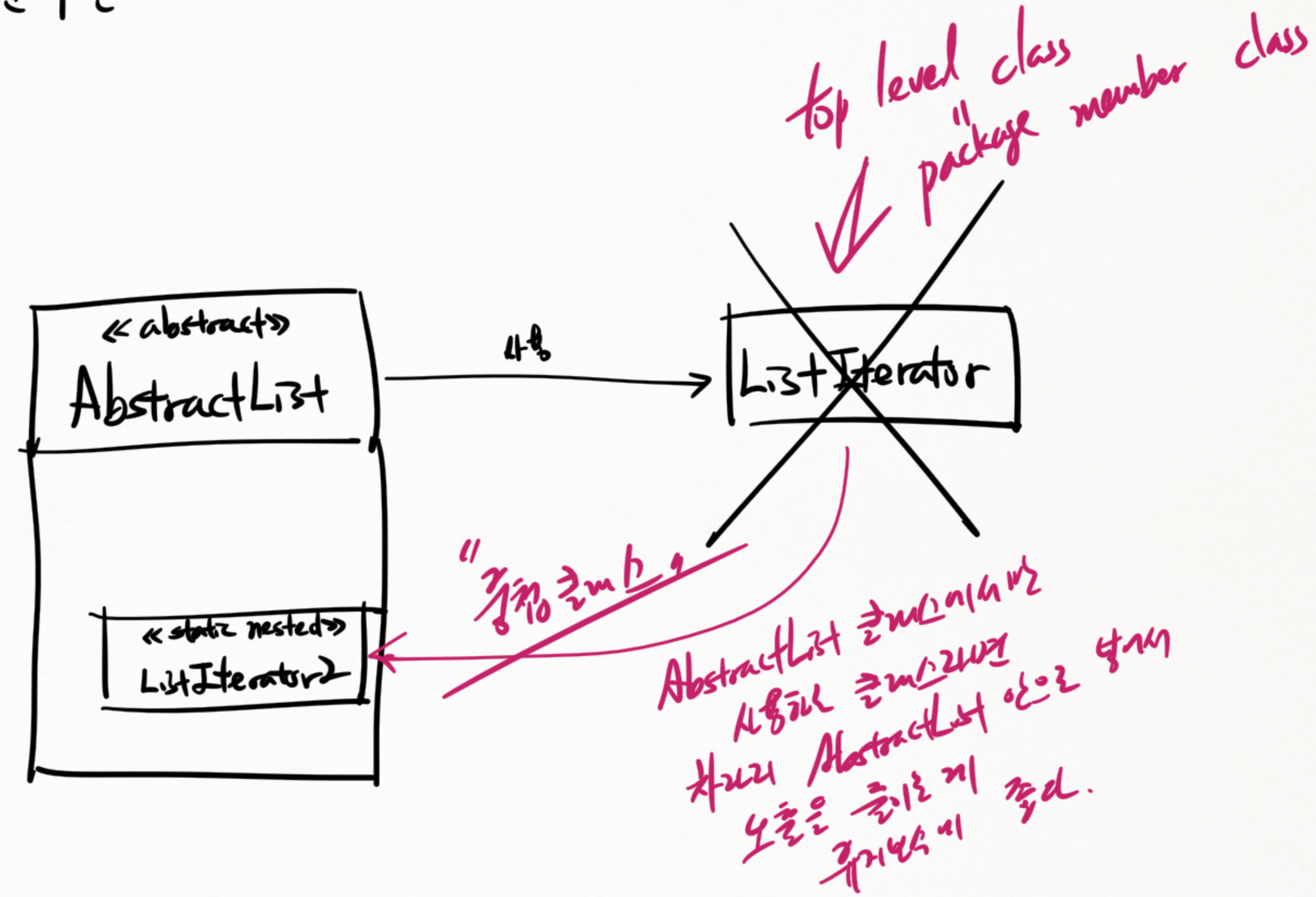
① 구현 - top level class





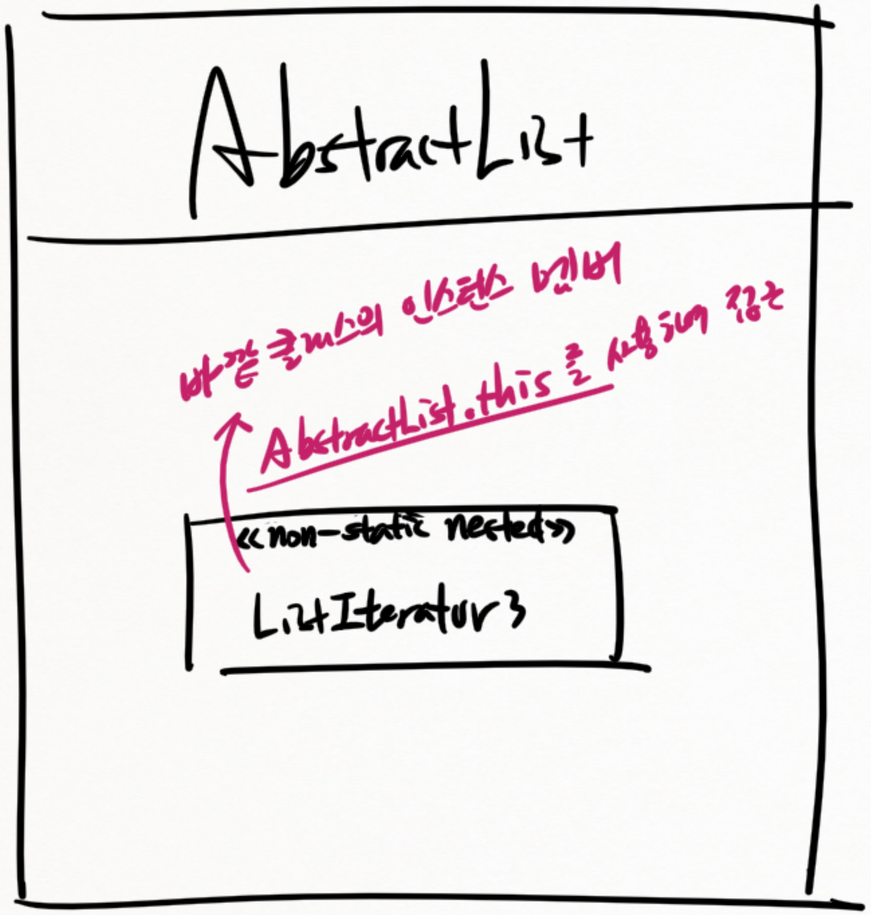
* Iterator 패턴 구현

② 구현 — static nested class

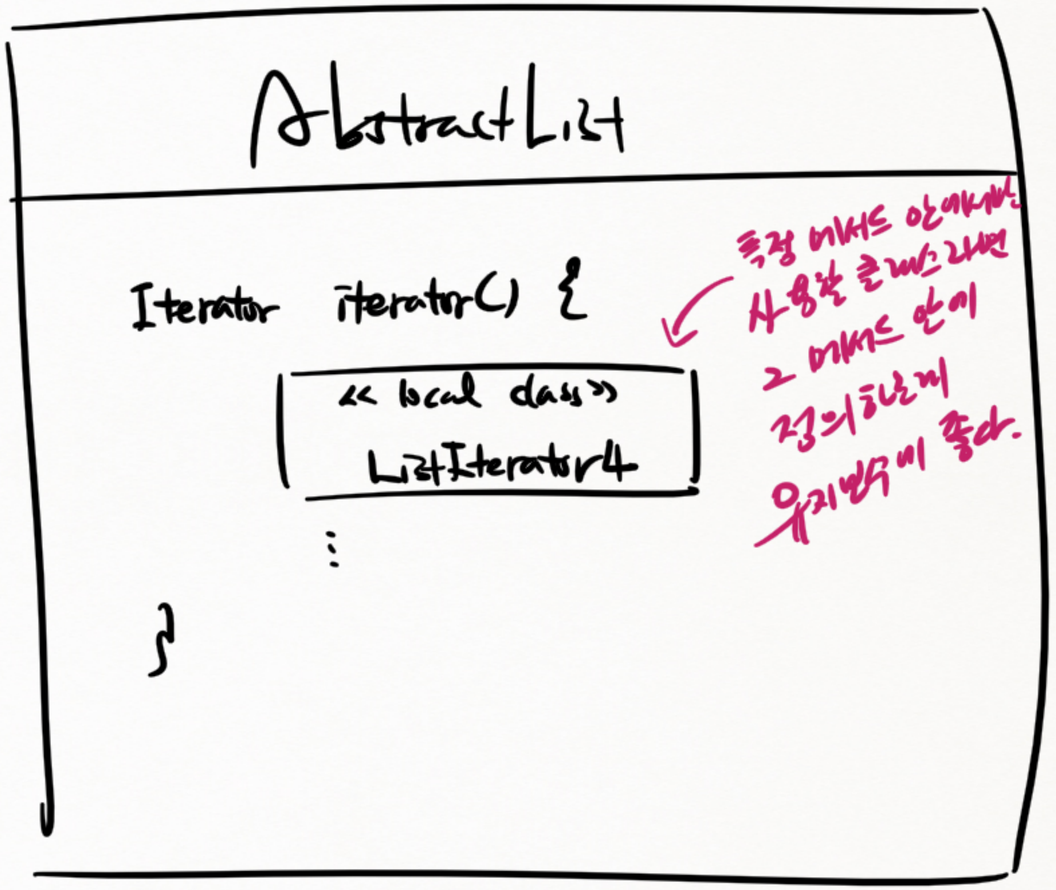


* Iterator 패턴 구현

③ 구현 — non-static nested class



④ 구현 — local class



* Iterator 패턴 구현

⑤ anonymous class

인터페이스명 또는 클래스명

```
new Iterator<>() {
    ≡
}
```

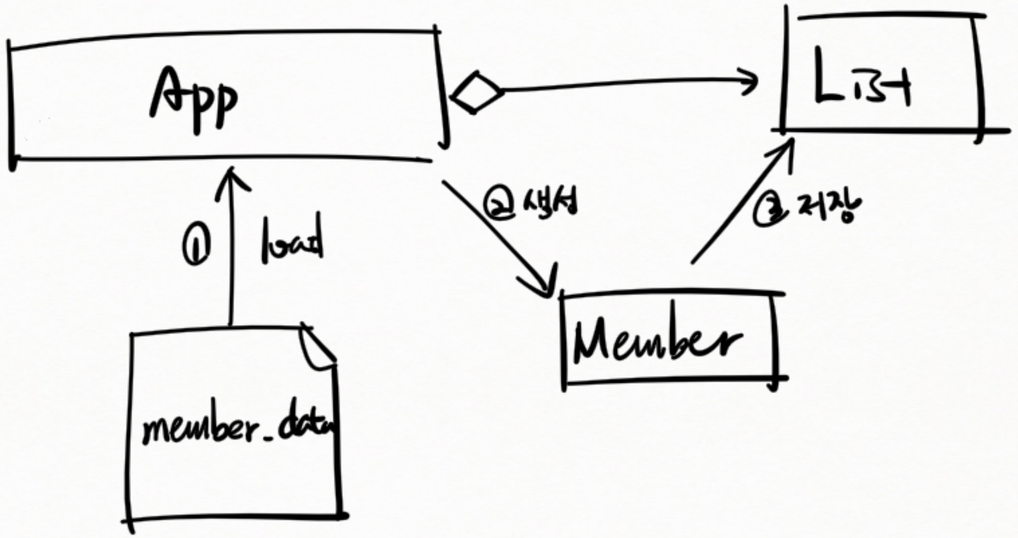
인스턴스 생성

부터 클래스 생성과 호출을 위한 파라미터 값

익명 클래스 body (본체)

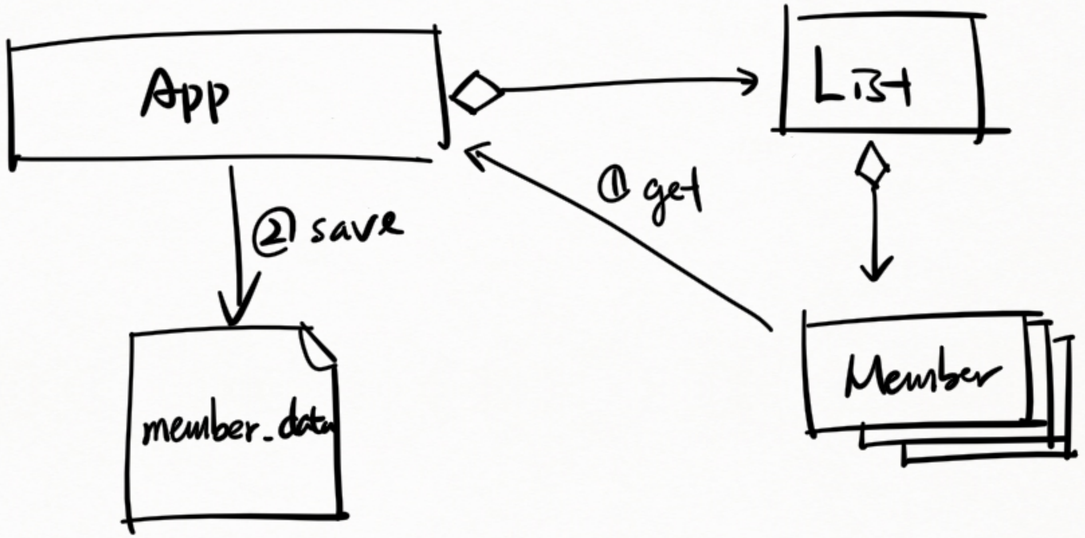
27. Data I/O Stream API — binary stream API 사용
↳ Data 저장

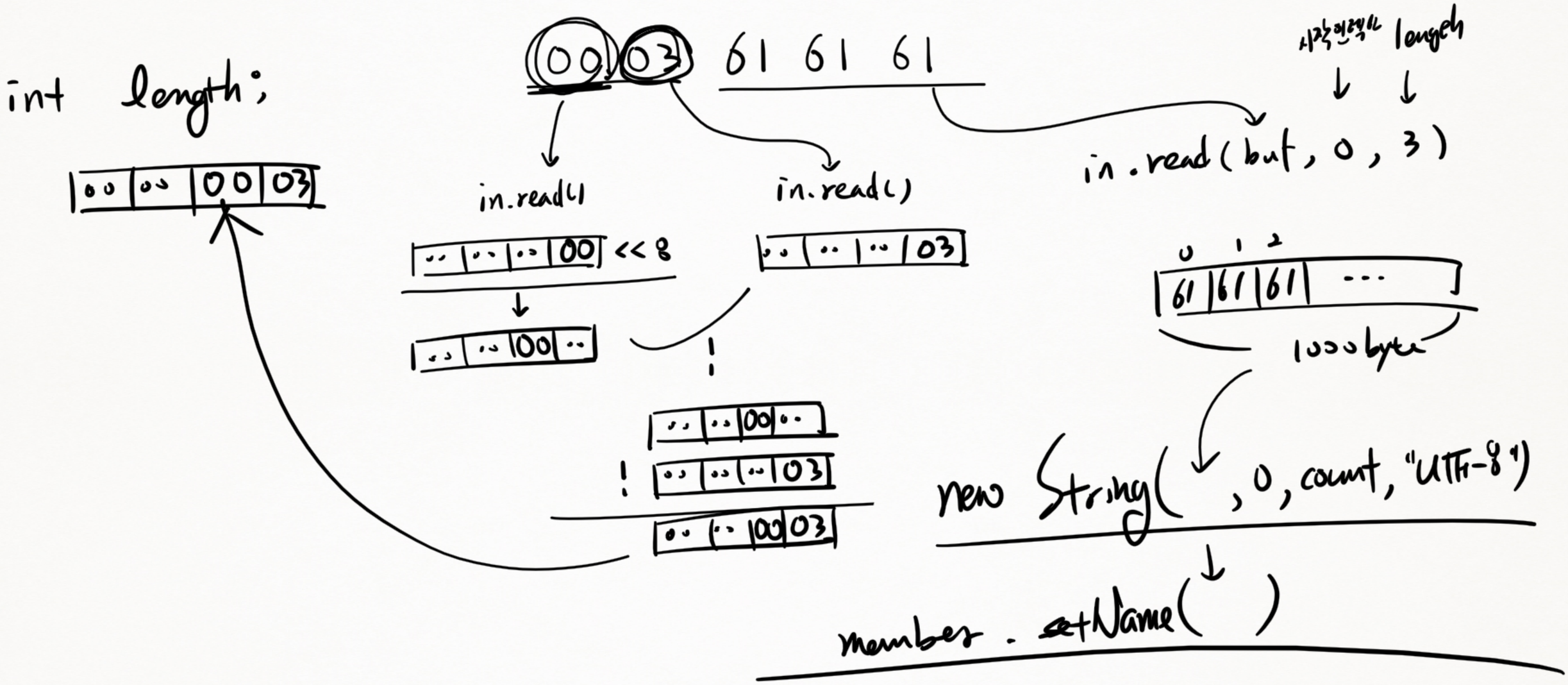
① Data 읽기



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

② Data 쓰기




int length;
00 03 61 61 61
in.read()
in.read()
in.read(buf, 0, 3)
시작인덱스
length
00 << 8
00
03
0 1 2
61 61 61 ...
1000byte
new String(, 0, count, "UTF-8")
member . setName( )

28. FileInputStream / FileOutputStream + (primitive type, String) 입출력 가능

⇑ 상속

FileOutputStream
- write(int)
- write(byte[])
- ⋮

↑

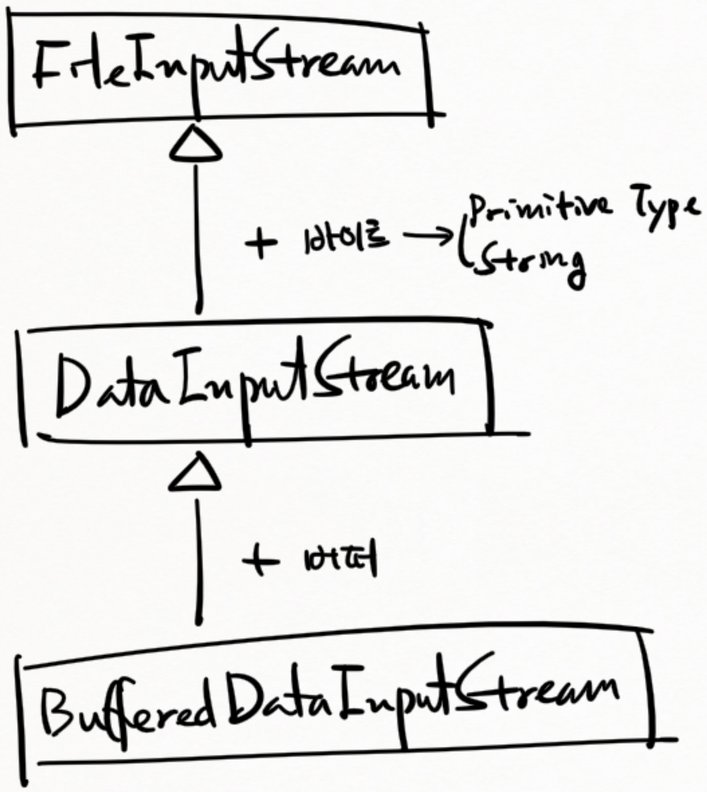
DataOutputStream
- writeInt(int)
- writeLong(long)
- writeChar(char)
- writeUTF(String)
- ⋮

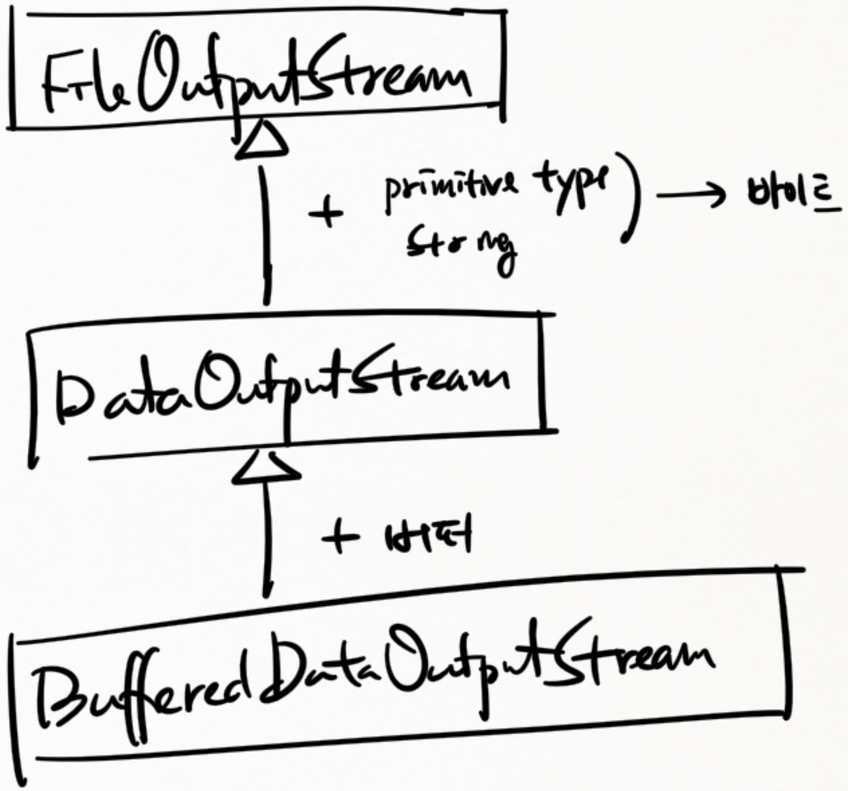
FileInputStream
- read() : int
- read(byte[]) : int
- ⋮

↑

DataInputStream
- readInt()
- readLong()
- readChar()
- readUTF()
- ⋮

29. 입출력에 버퍼 도입 : 상속 분법 활용

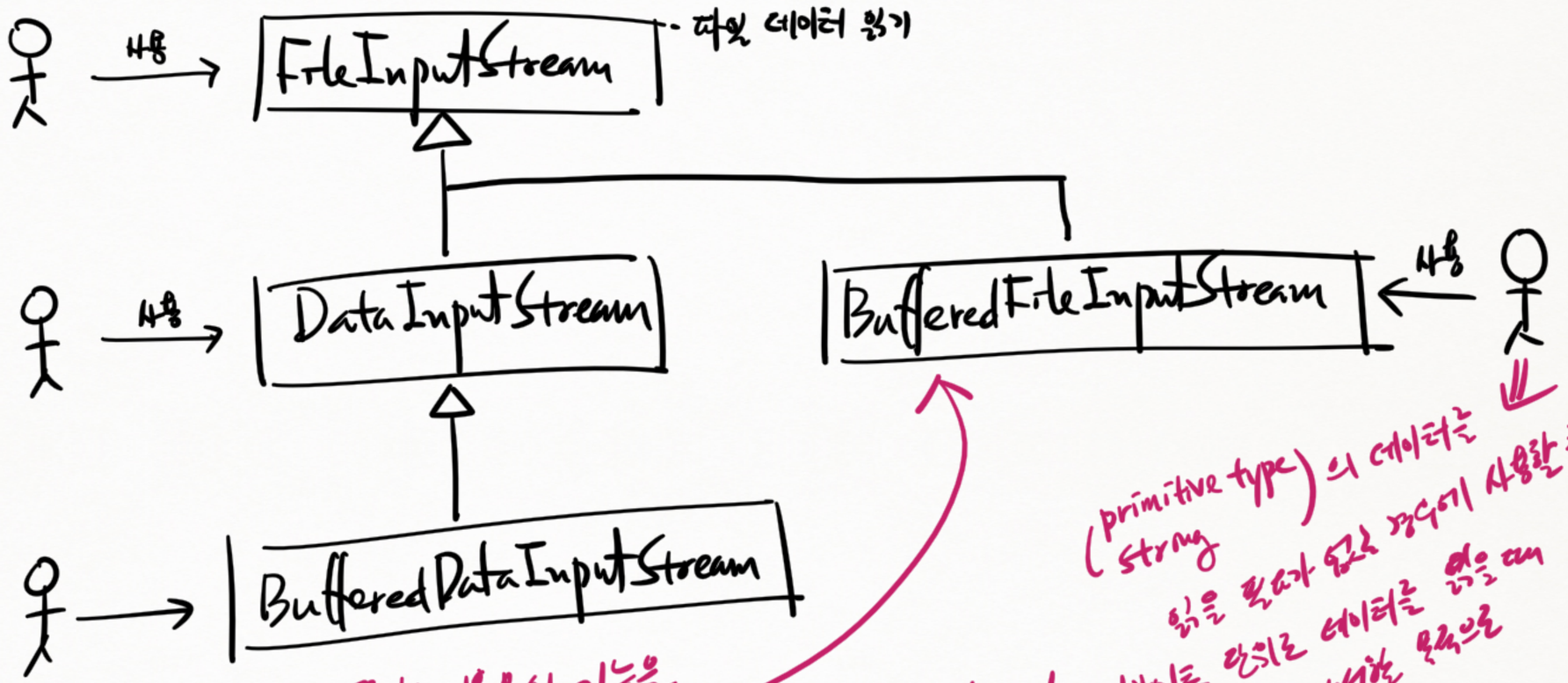




30. 기능을 확장할 때 상속 대신 Decorator 패턴 적용

① 상속을 이용한 기능 확장의 문제점

- byte/byte[] 읽기
- ☆ Primitive Type/String 읽기 불편



- (primitive type / String) 읽기 편함
- 바이트 단위로 읽기 때문에 대량의 데이터를 읽을 때 overhead 발생! (Data Seek Time)
- 버퍼를 이용해서 읽기 성능개선

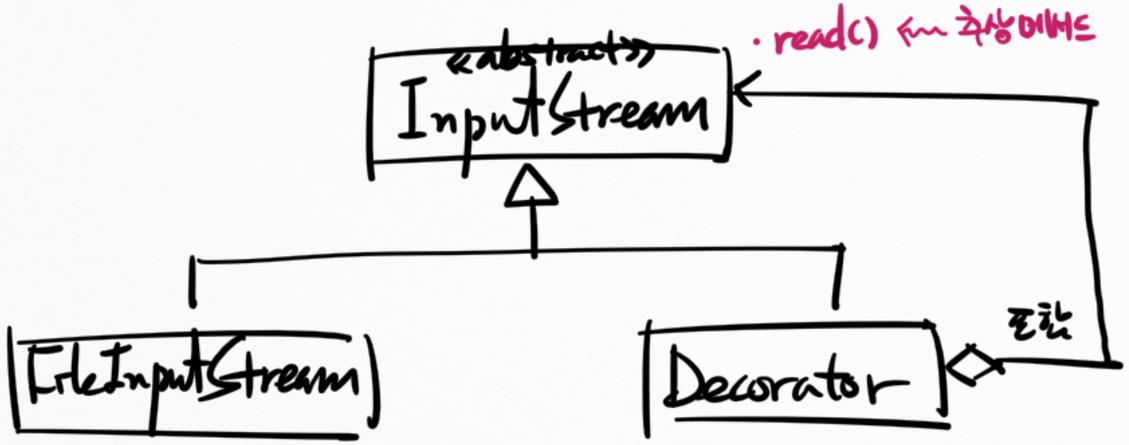
* 상속은 중간 부모의 기능을 제거할 수 없다. 그래서 별도의 sub 클래스를 또 정의해야 한다! ↳ 이것이 상속의 한계다!

(primitive type / String)의 데이터를 읽을 필요가 없는 경우에 사용할 클래스 바이트 단위로 데이터를 읽을 때 읽기 성능을 개선할 목적으로 사용.

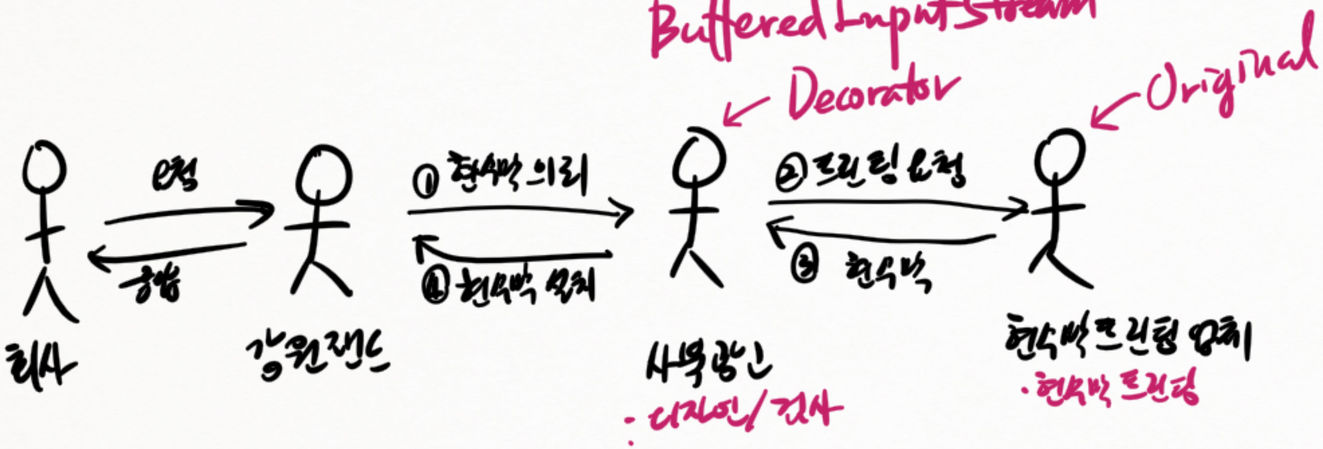
30. 기능을 확장할 때 상속 대신 Decorator 패턴 적용

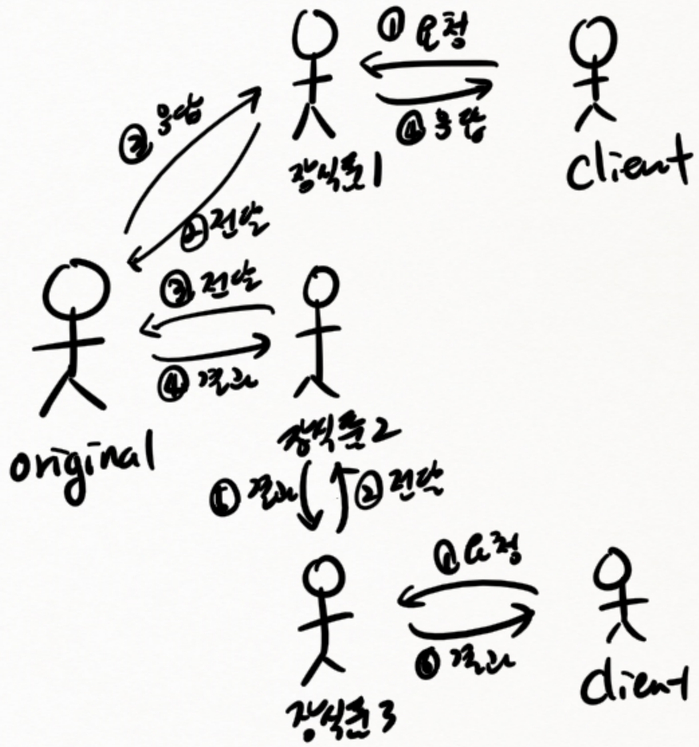
② 장식자(decorator) 패턴 기능을 덧붙이는 데에 유리하기 때문에 많이 쓰임

↳ Decorator 패턴 (GoF)
(Composite 패턴과 유사)

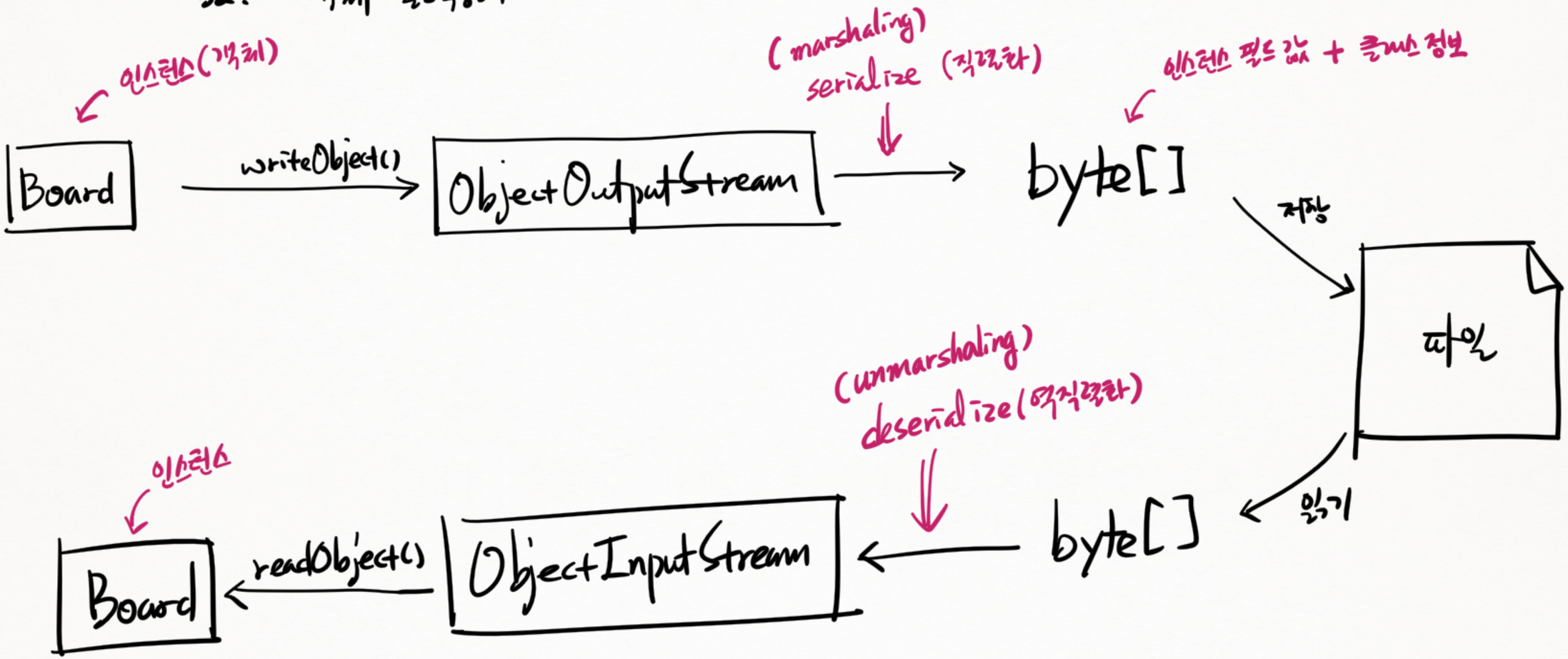


예) DataInputStream
BufferedInputStream





32. 객체 출력하기


인스턴스(객체)
Board
writeObject()
ObjectOutputStream
(marshaling)
serialize (직렬화)
인스턴스 필드 값 + 클래스 정보
byte[]
저장
파일
읽기
byte[]
(unmarshaling)
deserialize(역직렬화)
ObjectInputStream
readObject()
Board
인스턴스

* java.io.Serializable 인터페이스

직렬화를 허용한 클래스에 대해서만 serialize 가능

인스턴스

writeObject( ) → ObjectOutputStream

⇐ 직렬화 허용?

java.io.Serializable 인터페이스 구현한 클래스

- 메서드가 없다
- 직렬화를 허락한다는 표시용.

* 직렬화는 허락 받아야 할 일인가?

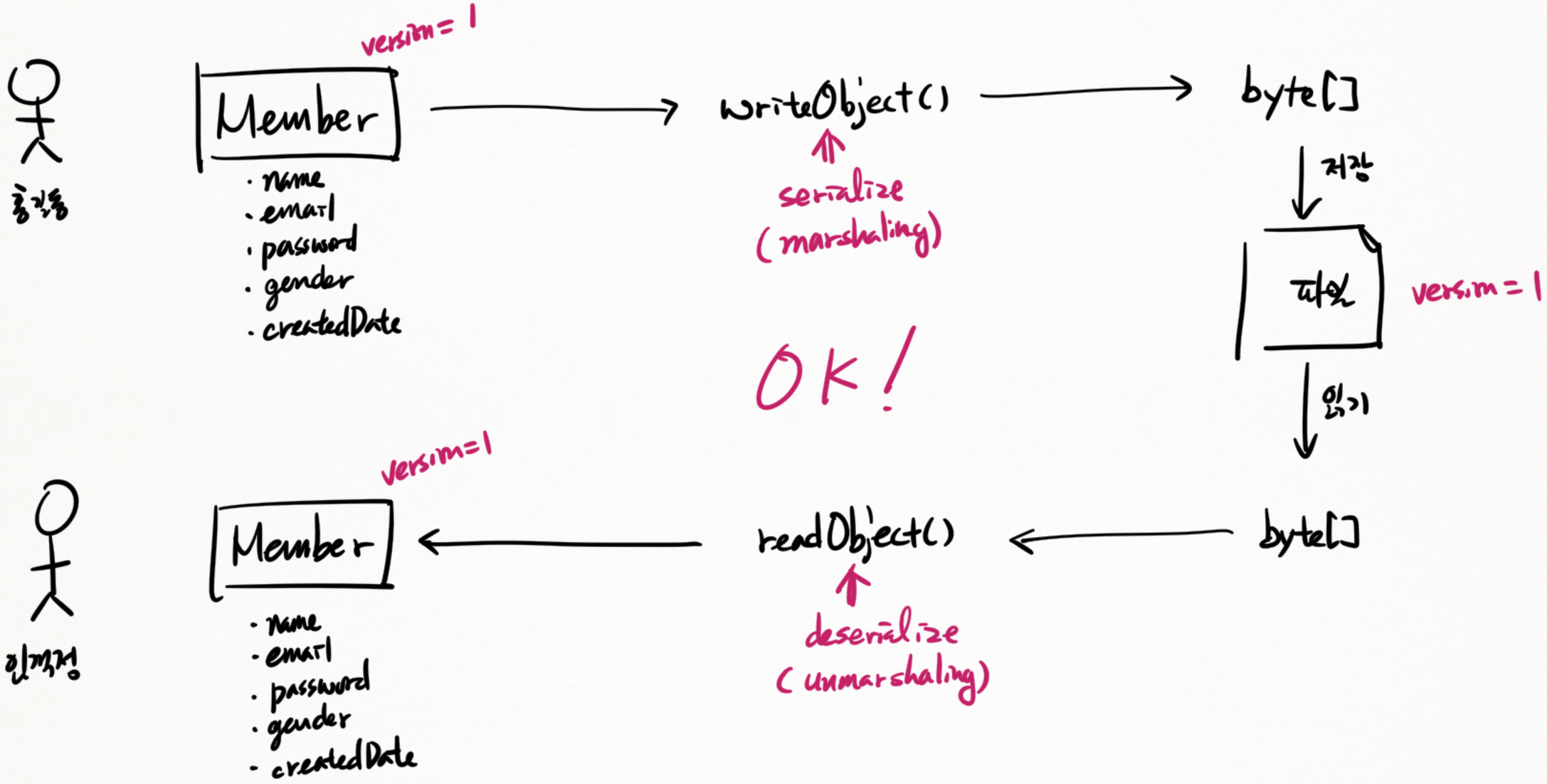
⇓

yes

⇓

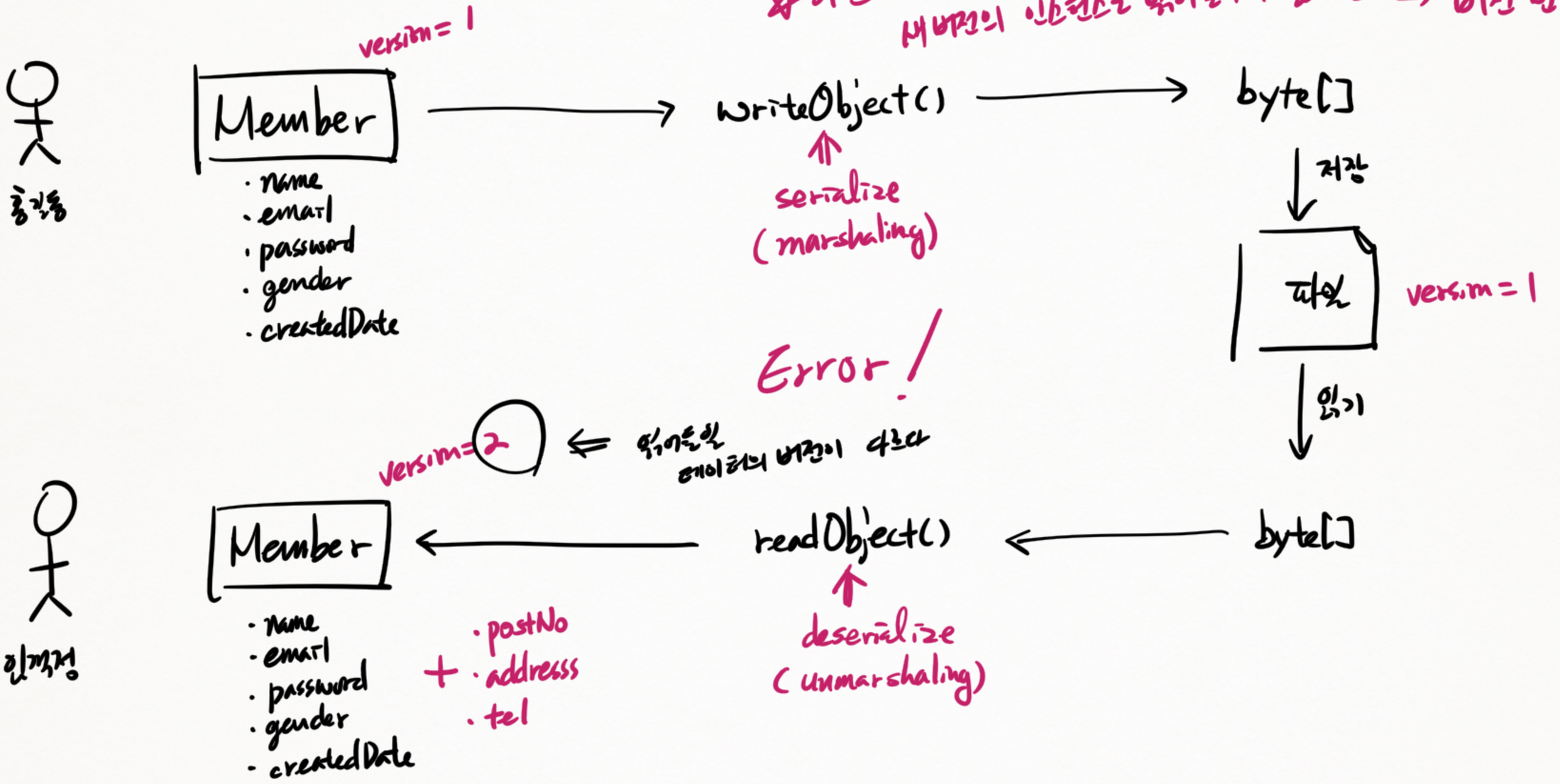
객체의 내용을 그대로 외부로 내보낸다는 것은 보안을 위협하는 행위다!

* serialVersionUID 스태틱 필드



* serialVersionUID 스태틱 필드

☆ 이전 버전으로 serialize 한 데이터를 새 버전의 인스턴스로 읽어들이기 싶다면 ⇒ 버전 번호를 관리하라!



* serialVersionUID 스태틱 필드

☆ 이전 버전으로 serialize 한 데이터를 새버전의 인스턴스로 읽어도 된다면 ⟹ 버전 번호를 같게 하라!

홍길동

version = 1

Member
- name
- email
- password
- gender
- createdDate

→ writeObject() → byte[]

writeObject() ⇑ serialize (marshaling)

byte[] ↓ 저장 → 파일 (version = 1) ↓ 읽기 → byte[]

OK!

인재정

version = 1

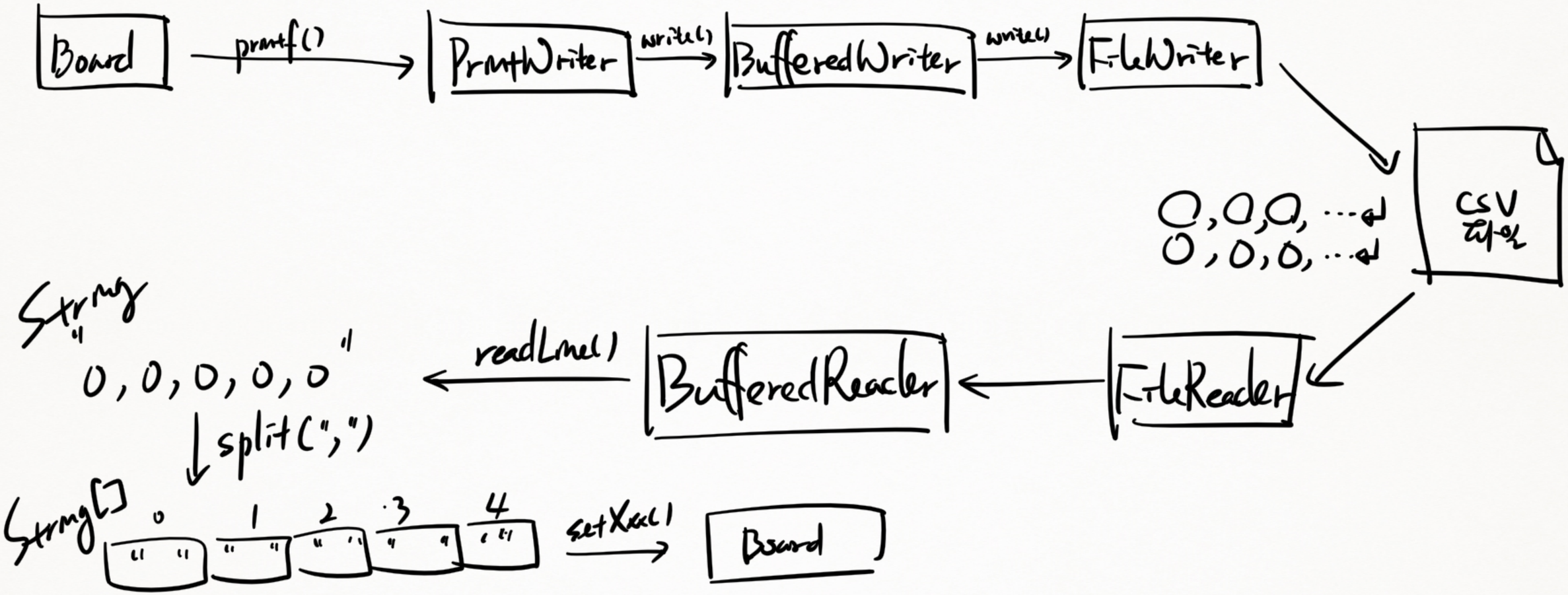
Member
- name
- email
- password
- gender
- createdDate

+
- postNo
- addresss
- tel

← readObject() ← byte[]

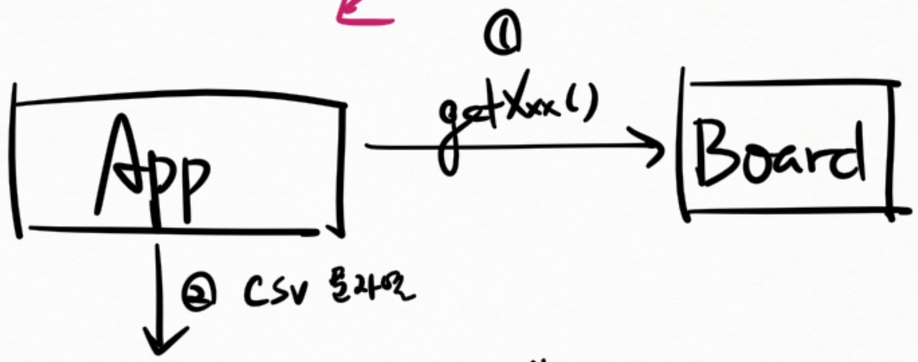
readObject() ⇑ deserialize (unmarshaling)

33. 텍스트 포맷(CSV)으로 입출력 하기



34. Refactoring : ① Information Expert

문제점: Board의 필드가 추가되거나 삭제되면 App 클래스를 변경해야 한다



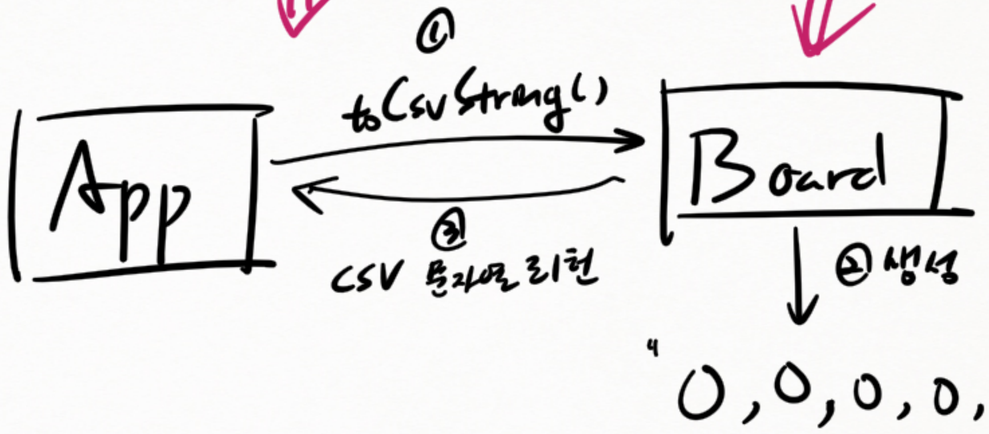
Board 객체의 값을 가지고 CSV 형식의 문자열을 생성하는 것은 App 클래스가 하고 있다.

"0,0,0,0,0"

Data를 갖고 있는 것은 Board인데 CSV 형식으로 문자열을 만드는 일은 객체는 App이다.

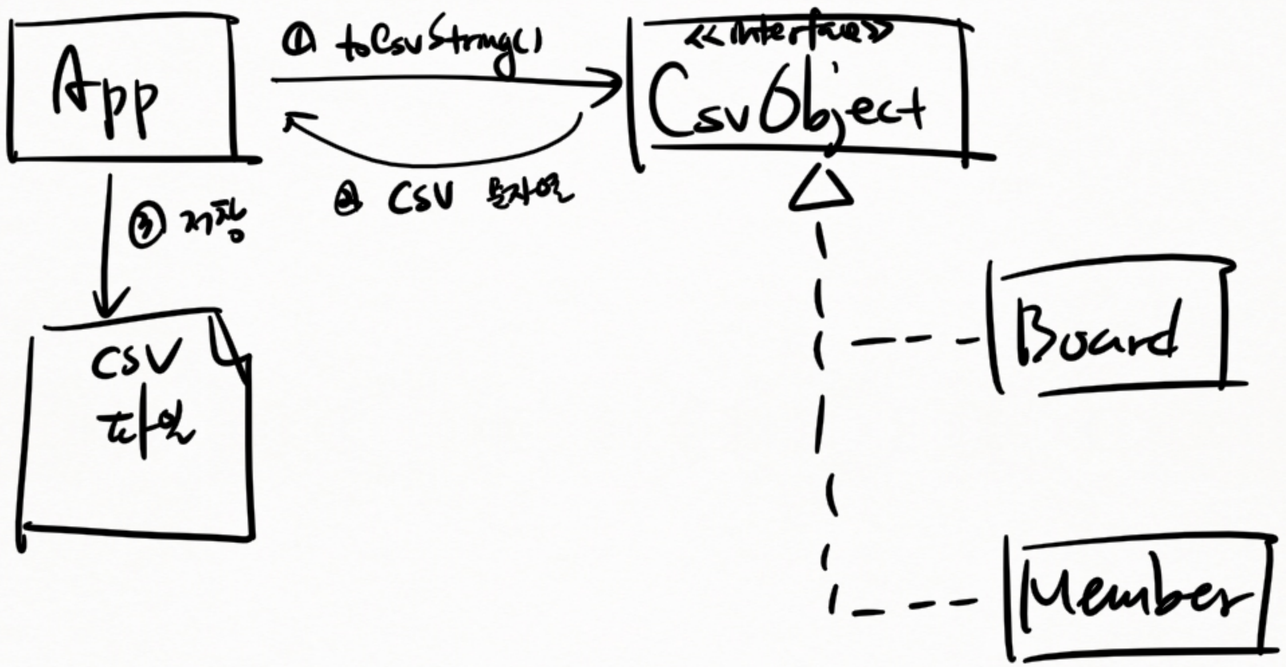
"Information Expert" 패턴으로 전환할 필요가 있다.

정보를 갖고 있는 객체가 정보를 가공하는 일을 하도록 설계 변경!

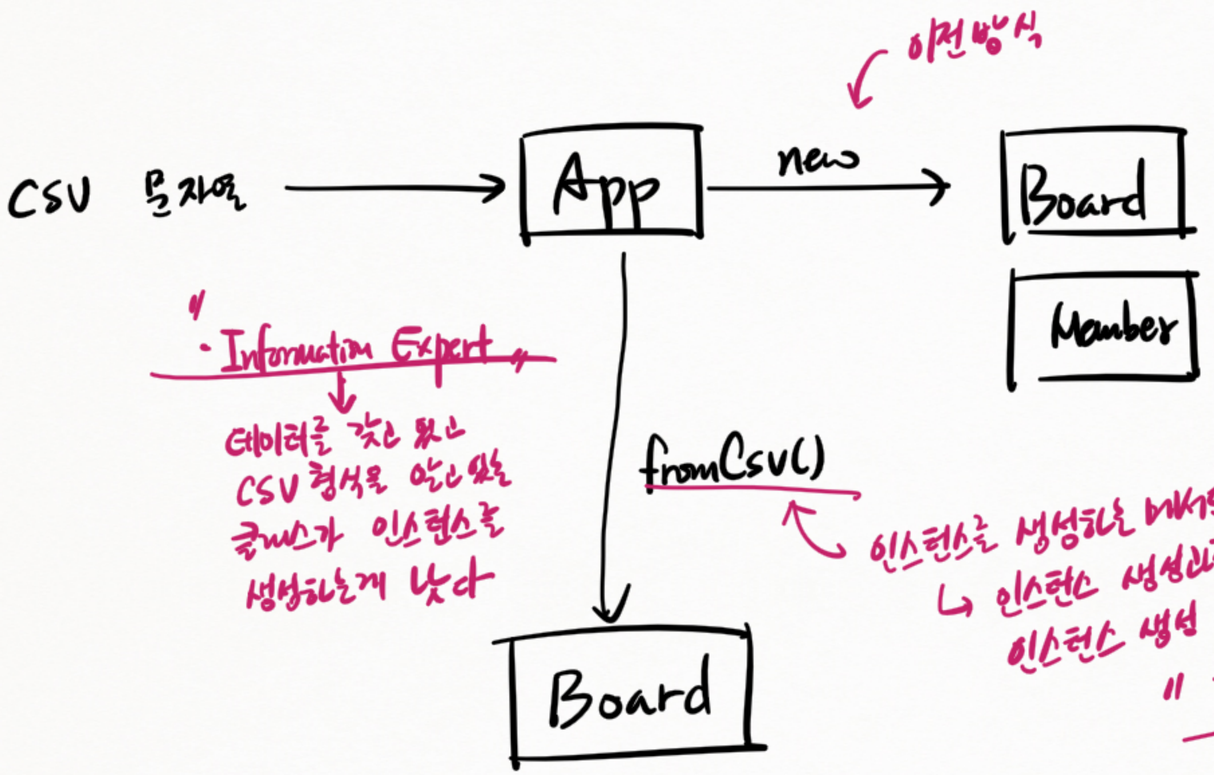


이점:
Board의 필드가 추가되거나 변경되더라도 App 클래스에는 영향을 주지 않는다.

* Interface 문법의 활용



② Factory Method (GoF) 패턴 적용



"· Information Expert"

↓

데이터를 갖고 있는 CSV 형식을 알고 있는 클래스가 인스턴스를 생성하는게 낫다

인스턴스를 생성하는 메서드

↳ 인스턴스 생성과정이 복잡할 때

인스턴스 생성 코드를 캡슐화(메서드로 숨기)하는 게 낫다.

" Factory Method 패턴 "

② Factory Method (GoF) 패턴 적용 → 개선

T가 타입 파라미터임을 선언!

☆ Reflection API를 사용하여 메서드를 찾고 호출하기! + Generic을 사용하여 다양한 타입에 대응할 수 있는 메서드를 정의!

```
<T> void loadCsv(String filename, List<T> list, Class<T> clazz) {
    Method factoryMethod = clazz.getDeclaredMethod("fromCsv", String.class);
```

메서드 정보를 다루는 객체

현재 클래스에서 정의된 메서드를 찾는다.

메서드명

파라미터 타입

리턴값 형변환

```
    (T) factoryMethd.invoke(null, line);
```

메서드 호출

인스턴스 주소 (스태틱 메서드인 경우, null을 넘긴다.

메서드를 호출할 때 넘겨줄 파라미터 값.

35. JSON 형식으로 입출력하기

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

① 자바스크립트 객체리터럴 ⟶ JSON 포맷

문자열 – { "문자열" / '문자열' }

숫자 – 314
3.14

논리 – true/false

객체 – {
  name : "홍길동",
  "age" : 20,
  'working' : true
}

property name
property value

```
{
  "name" : "홍길동",
  "age" : 20,
  "working" : true
}
```

반드시 큰 따옴표(")를 사용해서 프로퍼티 이름을 적어야 한다.

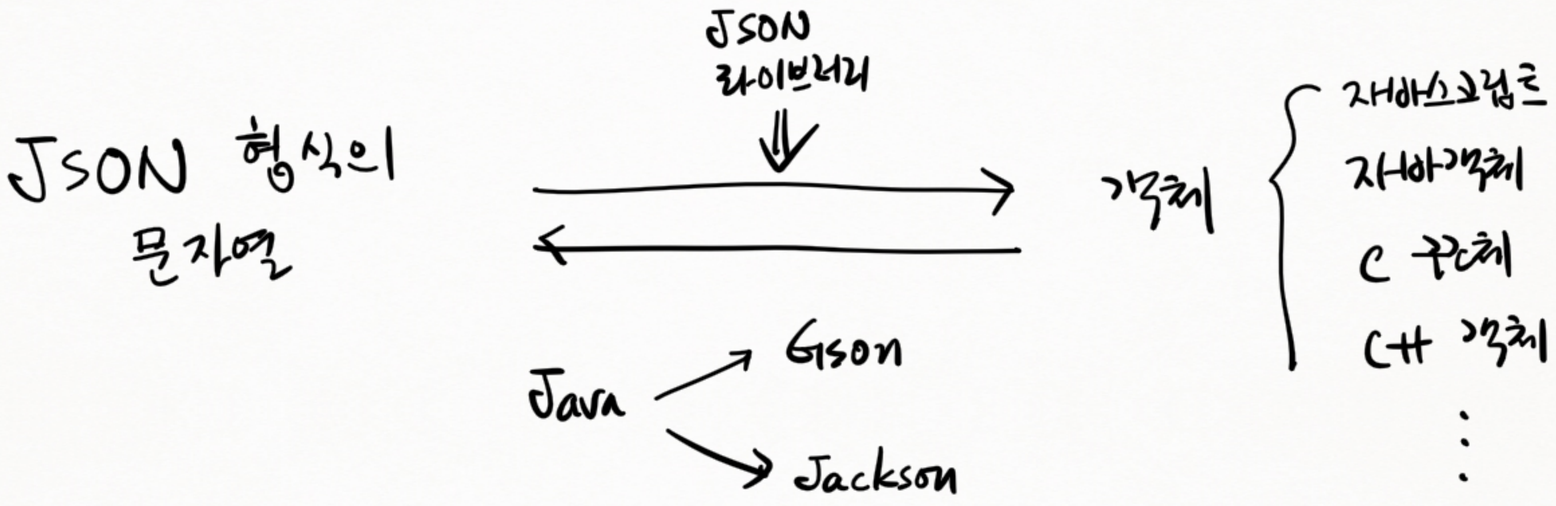
문자열은 반드시 큰 따옴표 사용

XML보다 더 간결하고 직관적이다.

JSON 또는 XML ⟶ (deserialize = decoding) ⟶ 자바스크립트 객체

JSON 또는 XML ⟵ (encoding = serialize) ⟵ 자바스크립트 객체

* JSON 인코딩/디코딩


JSON 형식의 문자열
JSON 라이브러리
객체
자바스크립트
자바객체
C 구조체
C++ 객체
⋮
Java
Gson
Jackson

* JSON 출력시 필드명과 프로퍼티명 ?

```
class Member {
    String nm;

    void setName(...){...}
    String getName(){...}

}
```

← field : 필드명 nm

← property = setter/getter : 프로퍼티명 name

- set/get 제거
- 나머지 메서드명에서 첫번째 알파벳을 소문자로 한 이름

프로퍼티명과 필드명이 다른 경우

- Gson → 필드명 사용
- Jackson → 프로퍼티명 사용

⇓

결론!

* Gson에서 날짜 다루기

new Gson() → [: Gson]

⇓ 기본 변환

Data ⇄ "월 일, 년도"

예) "7월 3, 2023"

new GsonBuilder() + 설정

setDateFormat("yyyy-MM-dd")

↓ create()

[: Gson] ← 인스턴스 — [Gson]

⇓ 설정된 규칙에 따라 변환

Date ⇄ "yyyy-MM-dd"

예) "2023-07-03"

* yyyy : 4자리 년도

month MM : 2자리 월

dd : 2자리 일자

HH : 2자리 시간

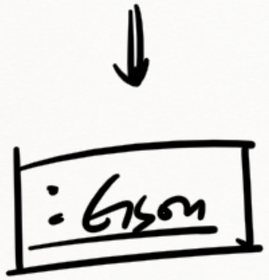
minute mm : 2자리 분

ss : 〃 초

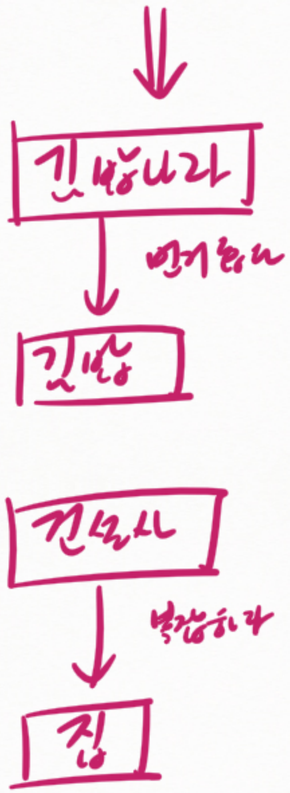
* Gson의 Builder 패턴

① new 명령으로 직접 인스턴스 생성

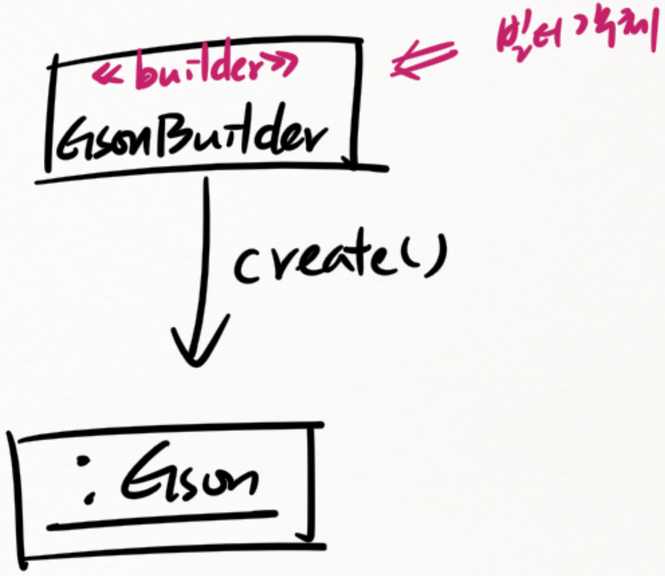
new Gson()



인스턴스 생성과정이 (변수로 주는 경우 / 복잡한 경우)



② 오직 객체를 통해 인스턴스 생성



* JsonSerializer.serialize()

1970년 1월 1일 0시 0분 0초 기준

↓ 경과된 milli-seconds

new DateFormat("yyyy-MM-dd") new Date() ← 1688354968539(long) = 2023년 7월 3일

↑ ↓

dateFormat.format(src)

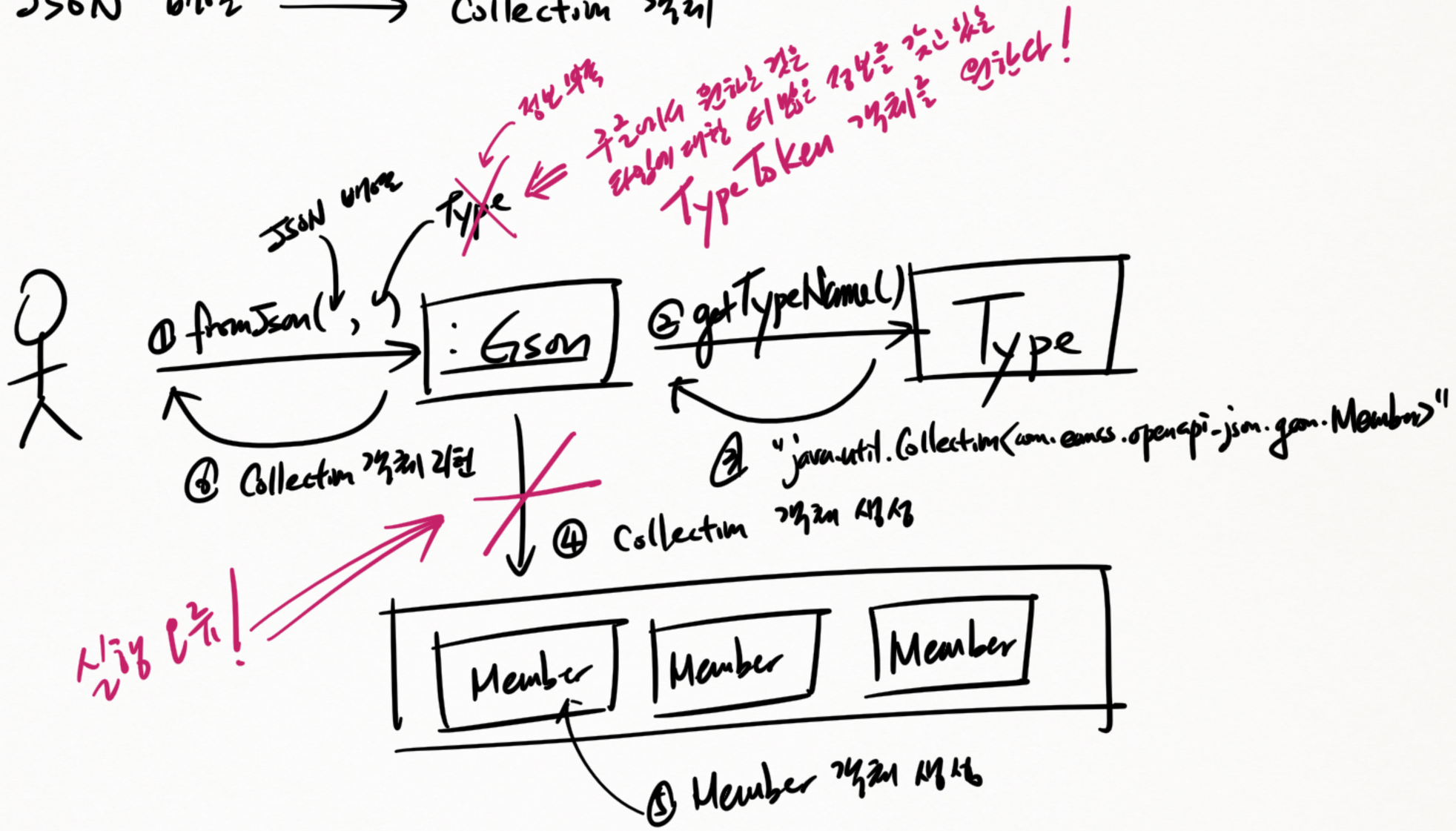
↓ 문자열로 변환

"2023-07-03"

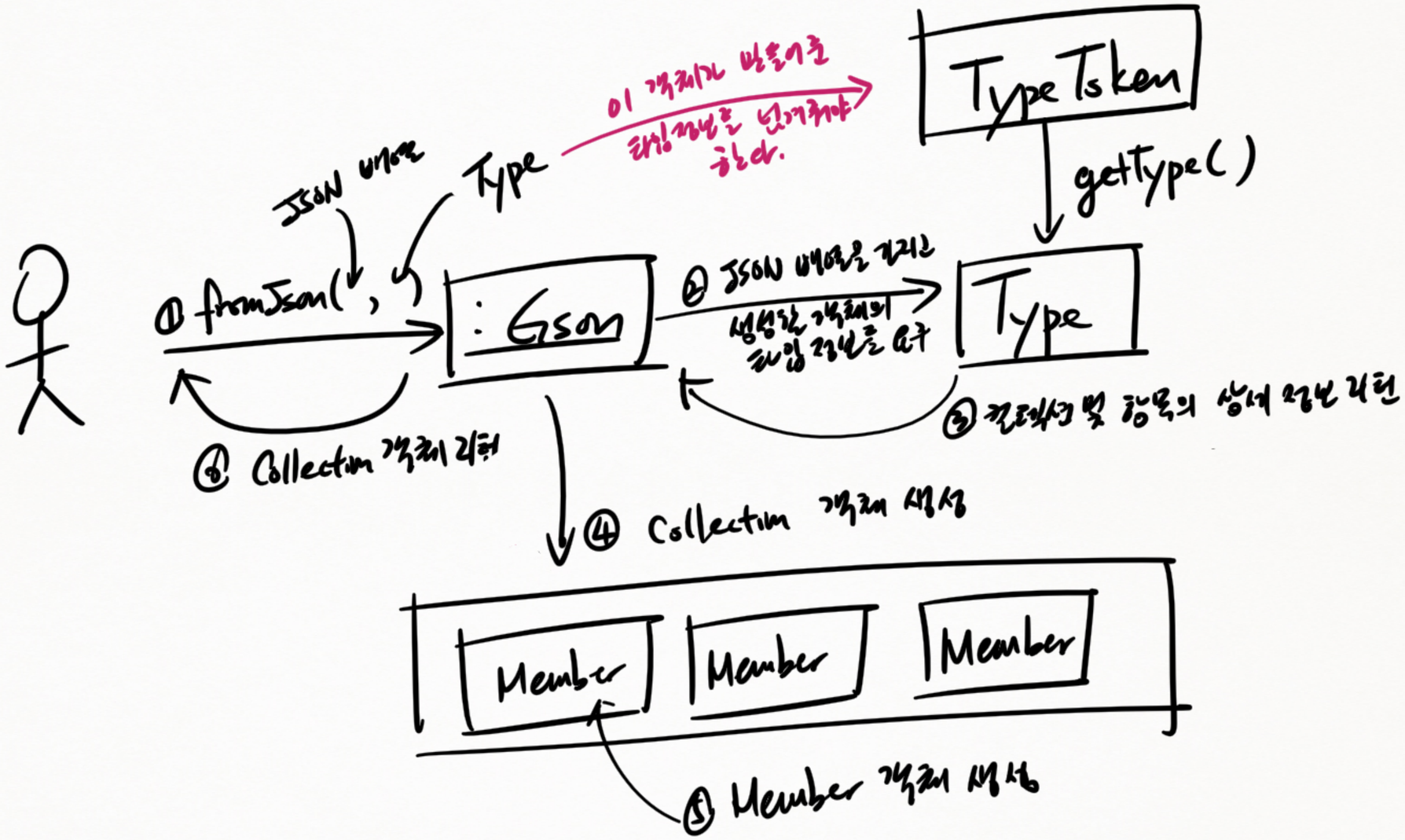
⇓

new JsonPrimitive( ) ⟹ return

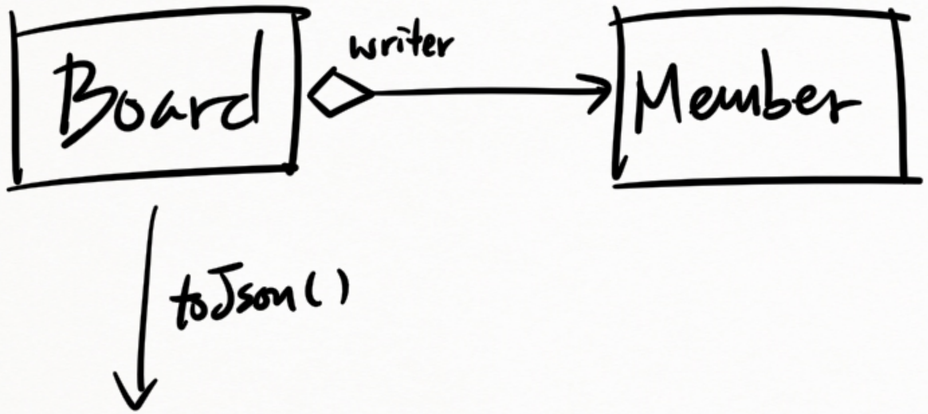
* JSON 배열 ⟶ Collection 객체



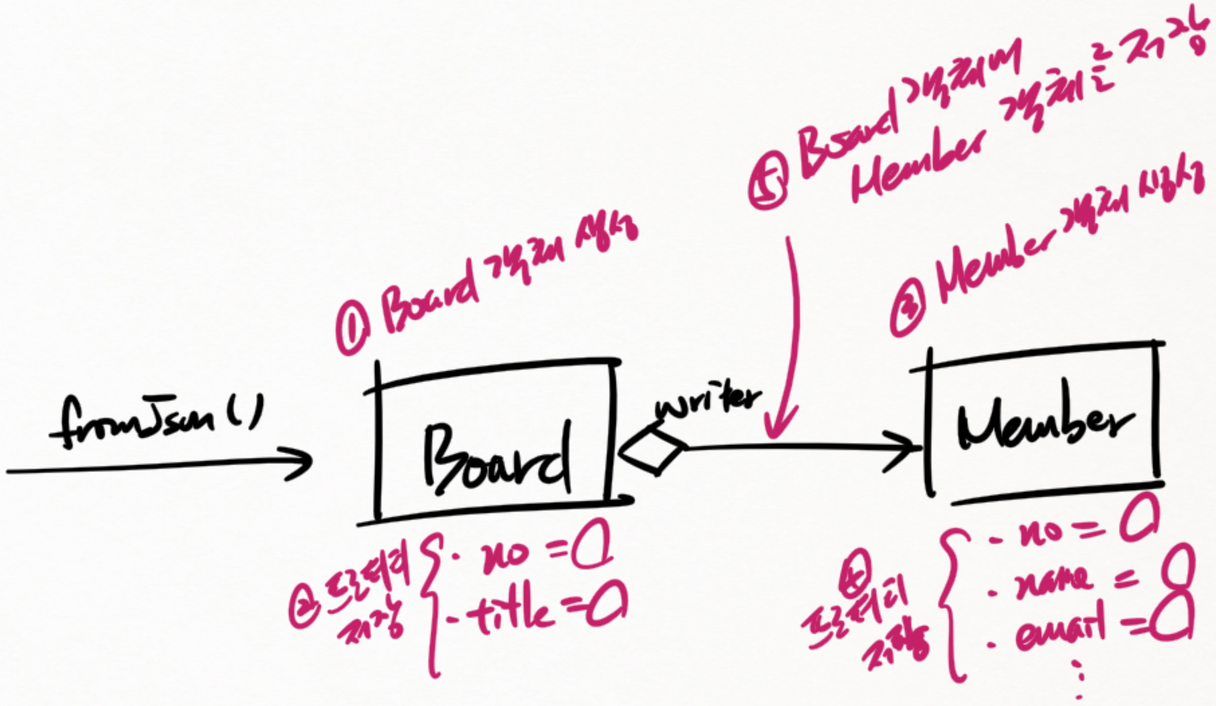
* JSON 배열 ⟶ Collection 객체



* 다른 객체를 참조 하는경우

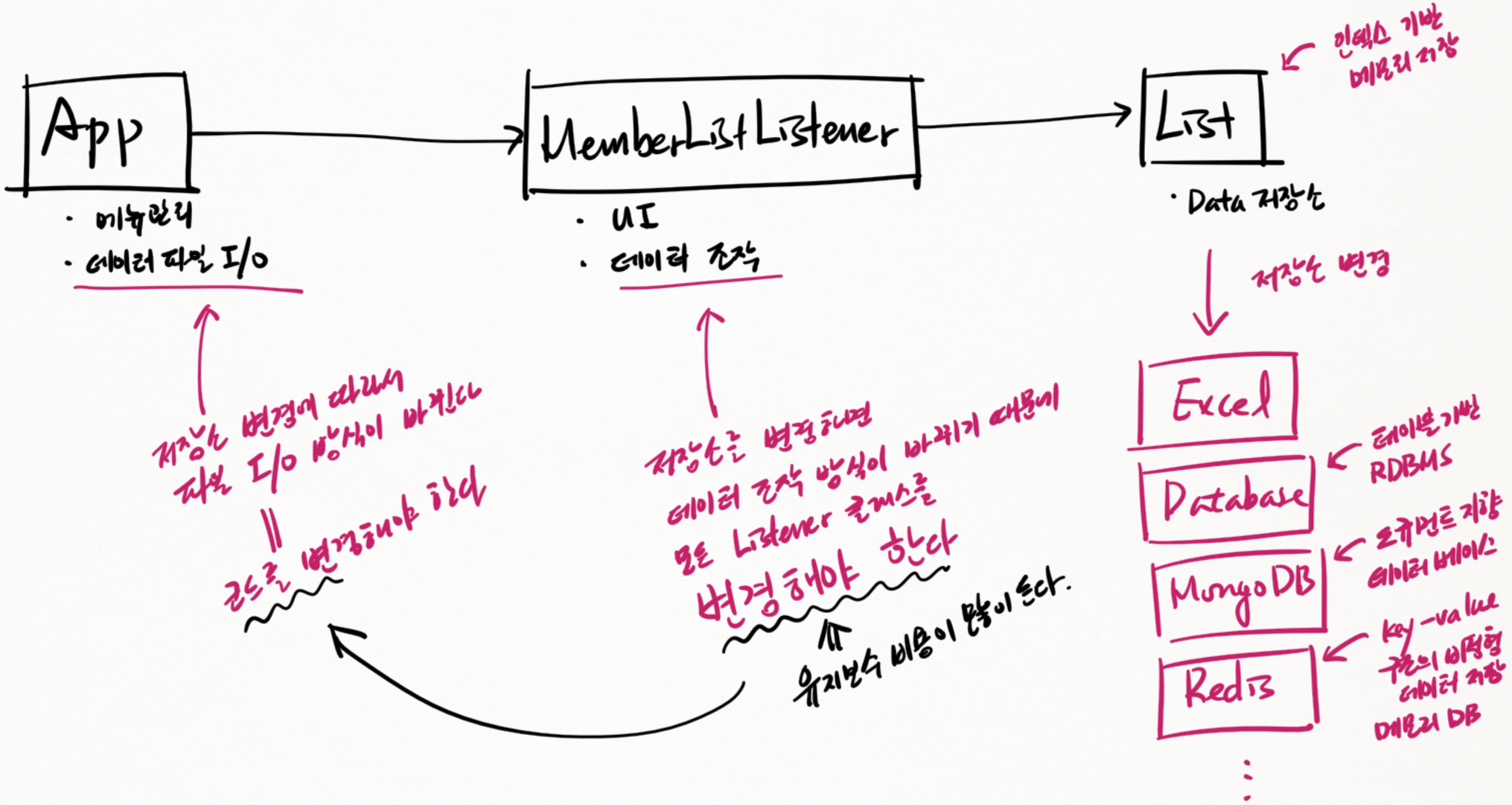



```
{
  "no": 1,
  "title": "—",
    ⋮
  "writer": {
      "no": 100,
      "name": "홍길동"
        ⋮
  }
}
```



36. DAO (Data Access Object) 도입

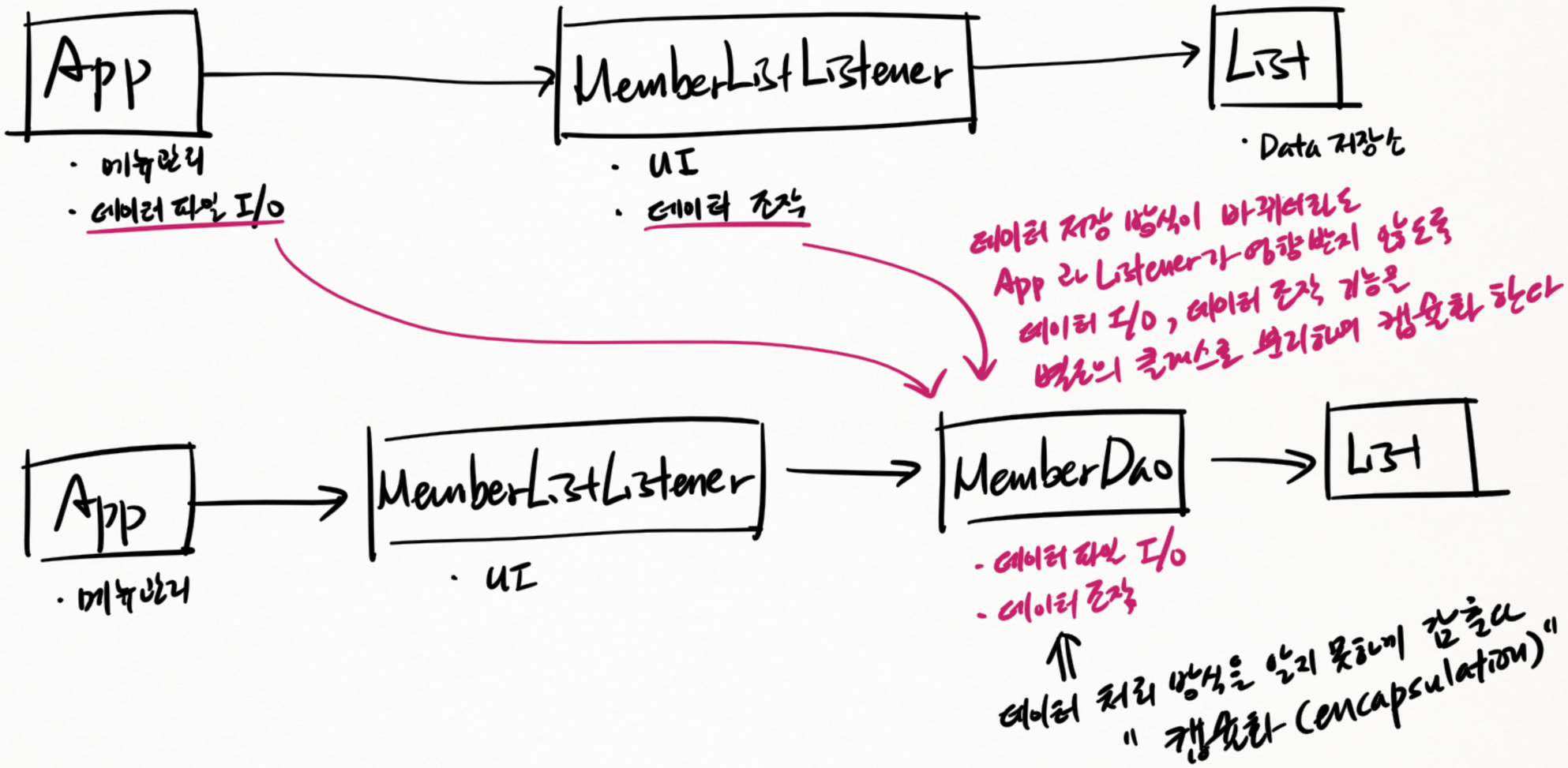
* 현황



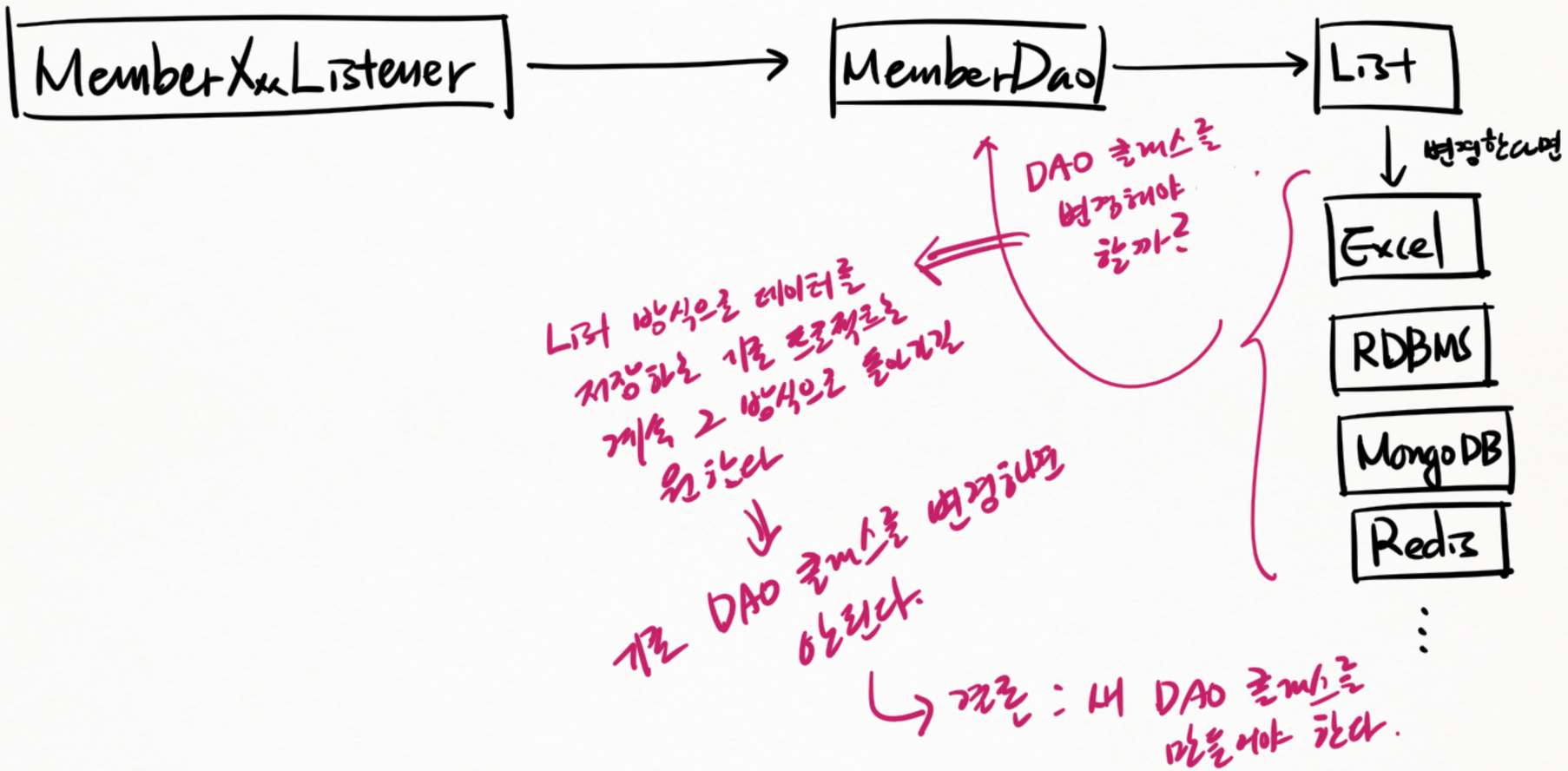
36. DAO (Data Access Object) 도입

↳ 데이터 저장 및 조작 담당

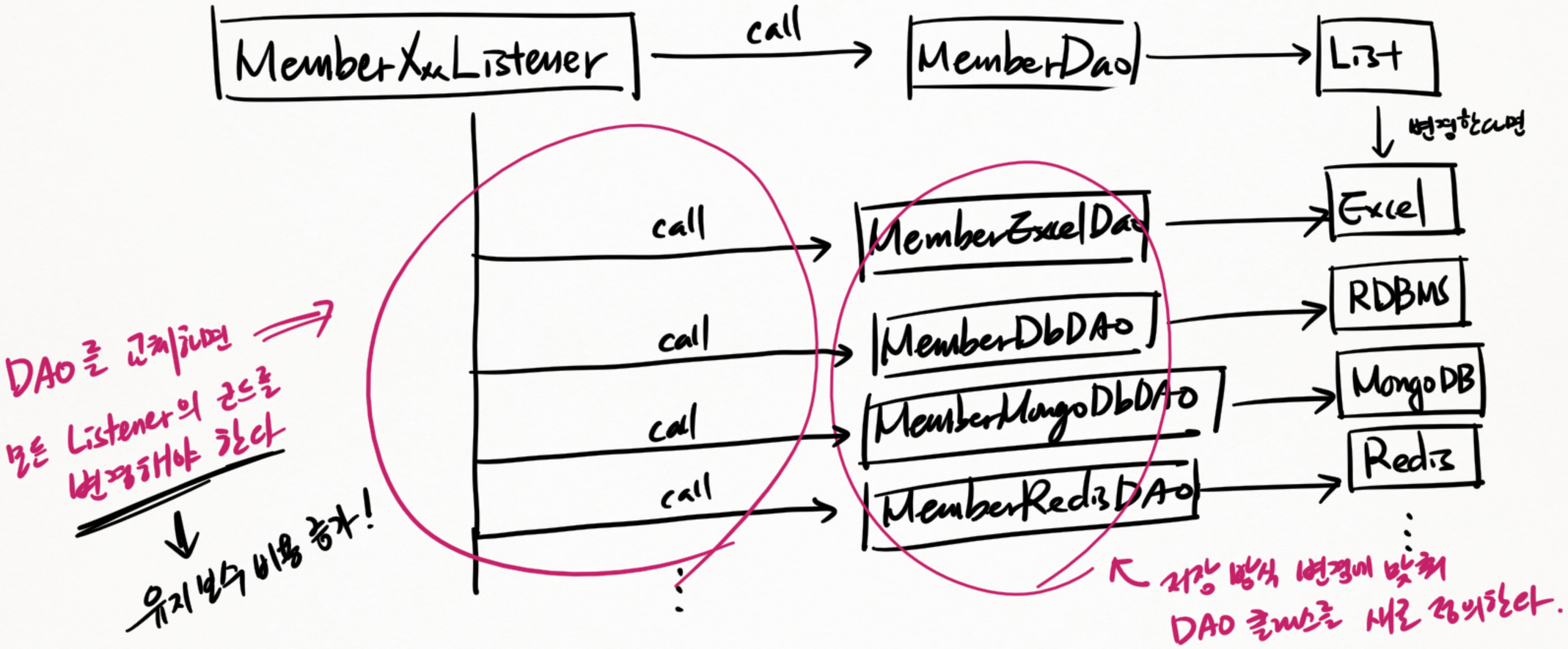
* 개선



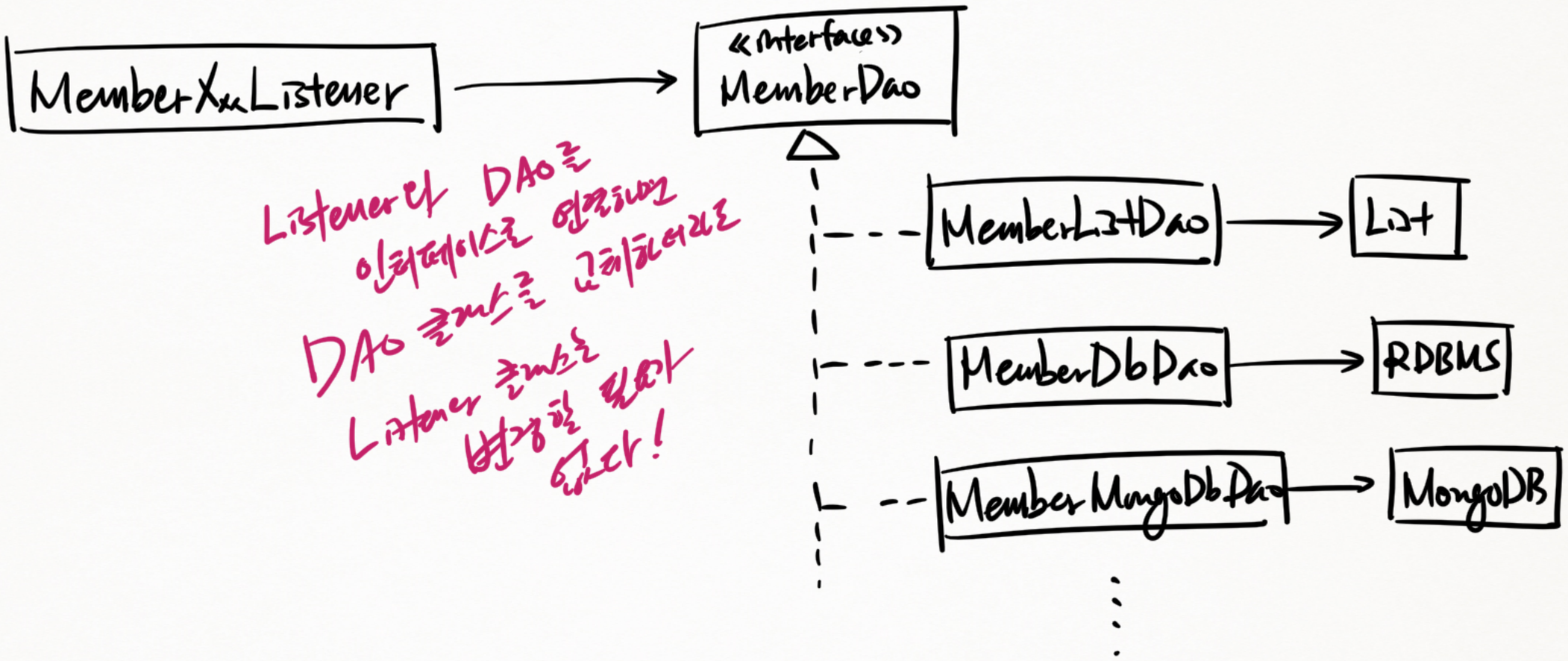
DAO 만들기


MemberXxxListener
MemberDao
List
변경한다면
Excel
RDBMS
MongoDB
Redis
⋮
DAO 클래스를 변경해야 할까?
List 방식으로 데이터를 저장하도록 기존 프로젝트를 계속 그 방식으로 흘러가길 원한다
기존 DAO 클래스를 변경하면 안된다.
결론 : 새 DAO 클래스를 만들어야 한다.

DAO 만들기


MemberXxxListener
call
MemberDao
List
변경한다면
Excel
call
MemberExcelDao
RDBMS
call
MemberDbDAO
MongoDB
call
MemberMongoDbDAO
Redis
call
MemberRedisDAO
DAO를 고치려면
모든 Listener의 코드를
변경해야 한다
유지보수 비용 증가!
저장 방식 변경에 맞춰
DAO 클래스를 새로 정의한다.

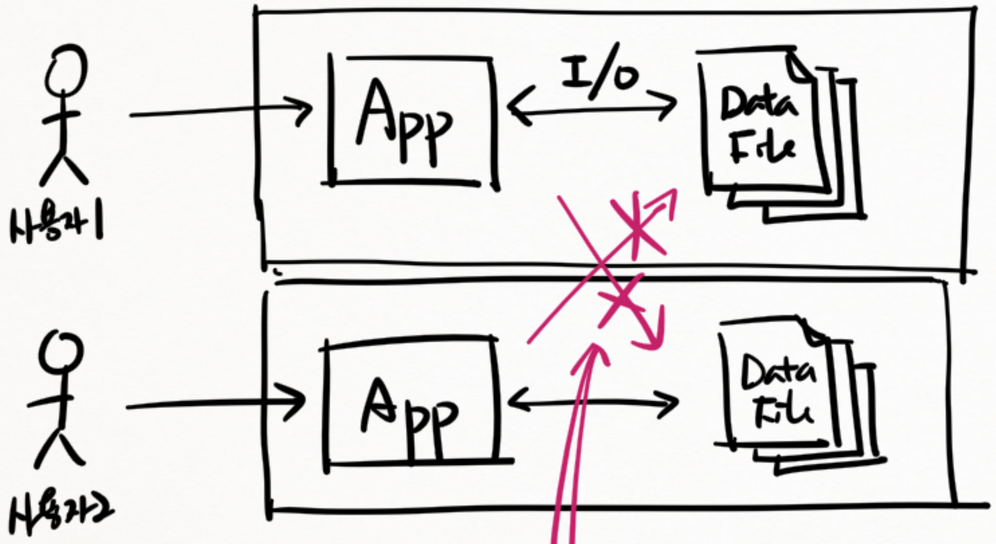
DAO 만들기



Listener가 DAO를
인터페이스로 연결하면
DAO 클래스를 교체하더라도
Listener 클래스를
변경할 필요가
없다!

37. 네트워킹을 통한 데이터 공유

① 현황



프로그램 간의 Data File 공유가 안된다.

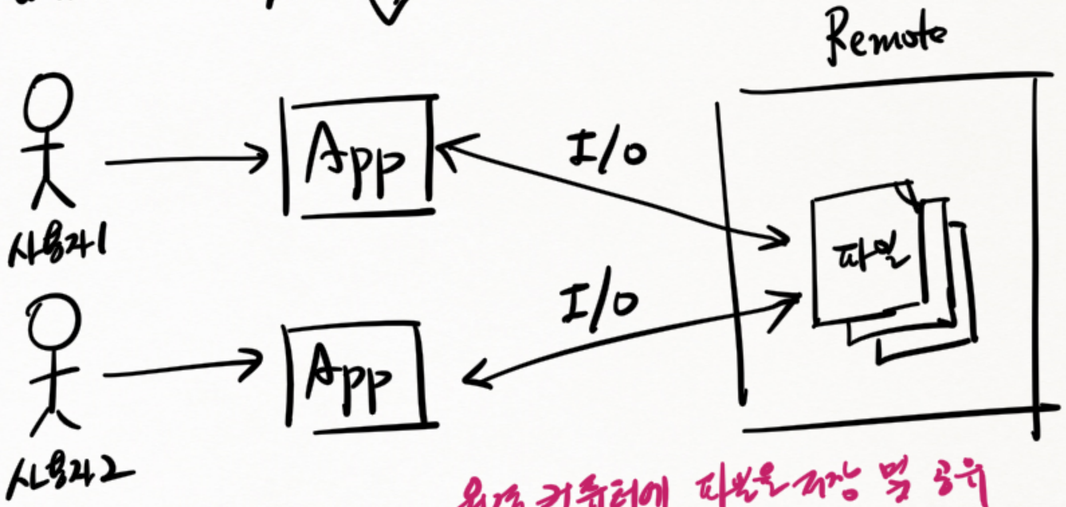
App을 실행하는 컴퓨터

데이터 파일이 로컬에 저장된다.

⇓

App 간에 데이터를 공유할 수 없다!

② 데이터 파일을 별도의 컴퓨터에 보관/공유 ⇓ 해결책?

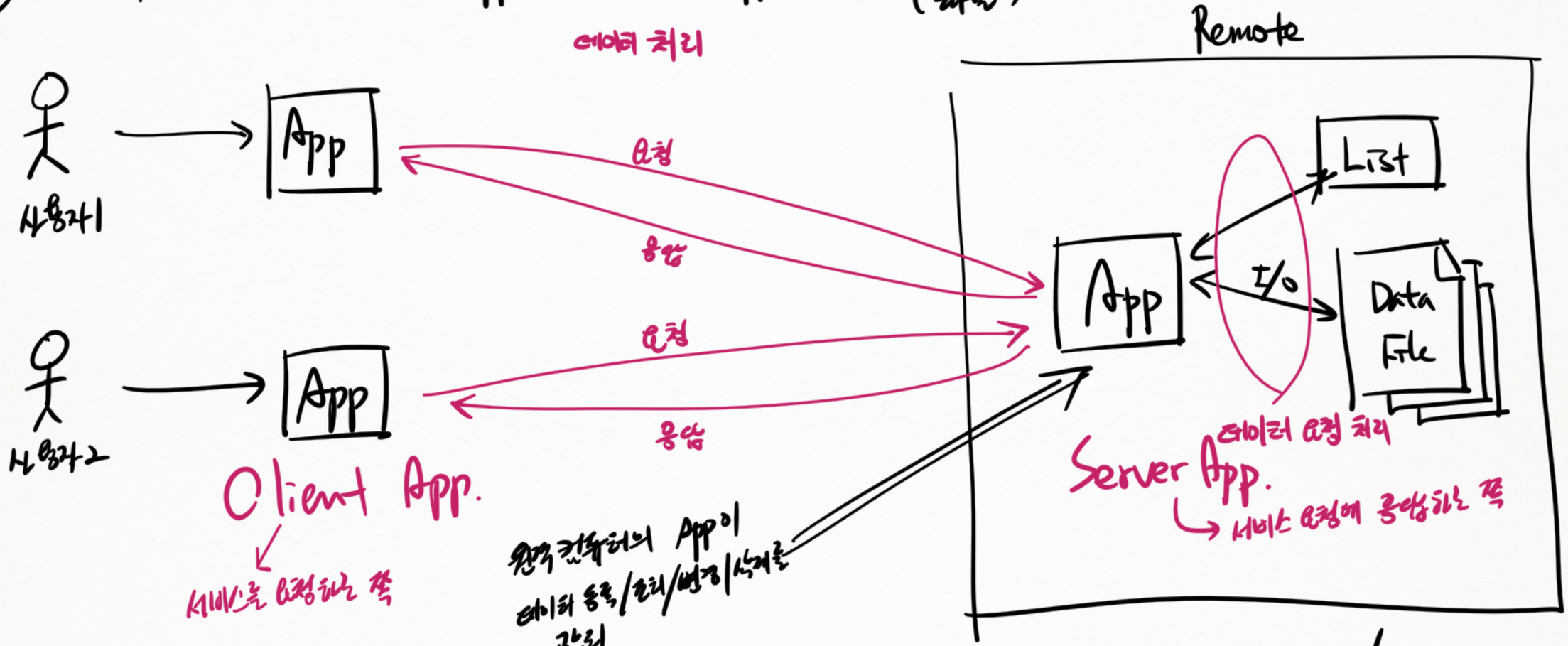


- 원격 컴퓨터에 파일을 저장 및 공유

⇓

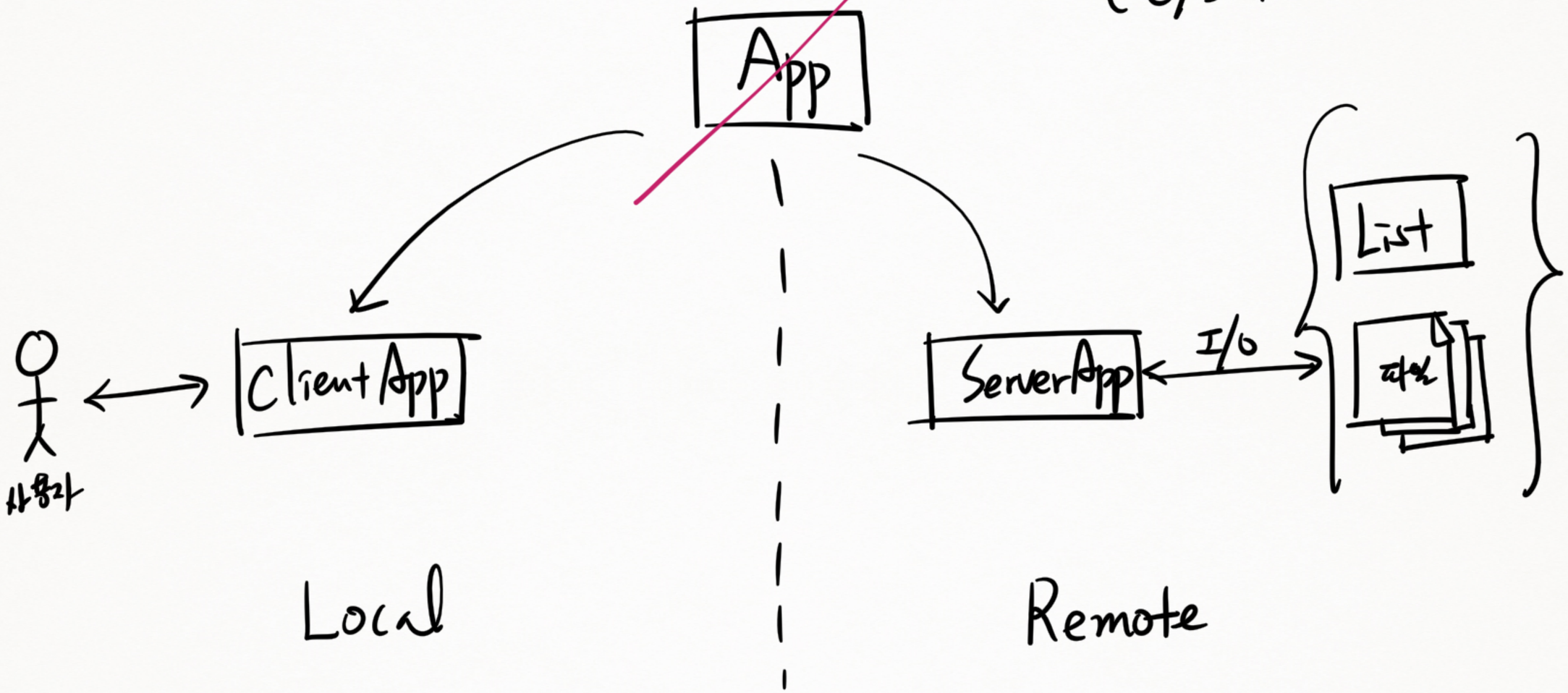
여러 App이 동시에 같은 파일을 접근한다 보면 파일의 데이터가 깨질 수 있다.

③ 데이터 관리 기능을 별도의 App.으로 분리 — App이 직접 (데이터 파일)에 접근하는 것을 막는다.

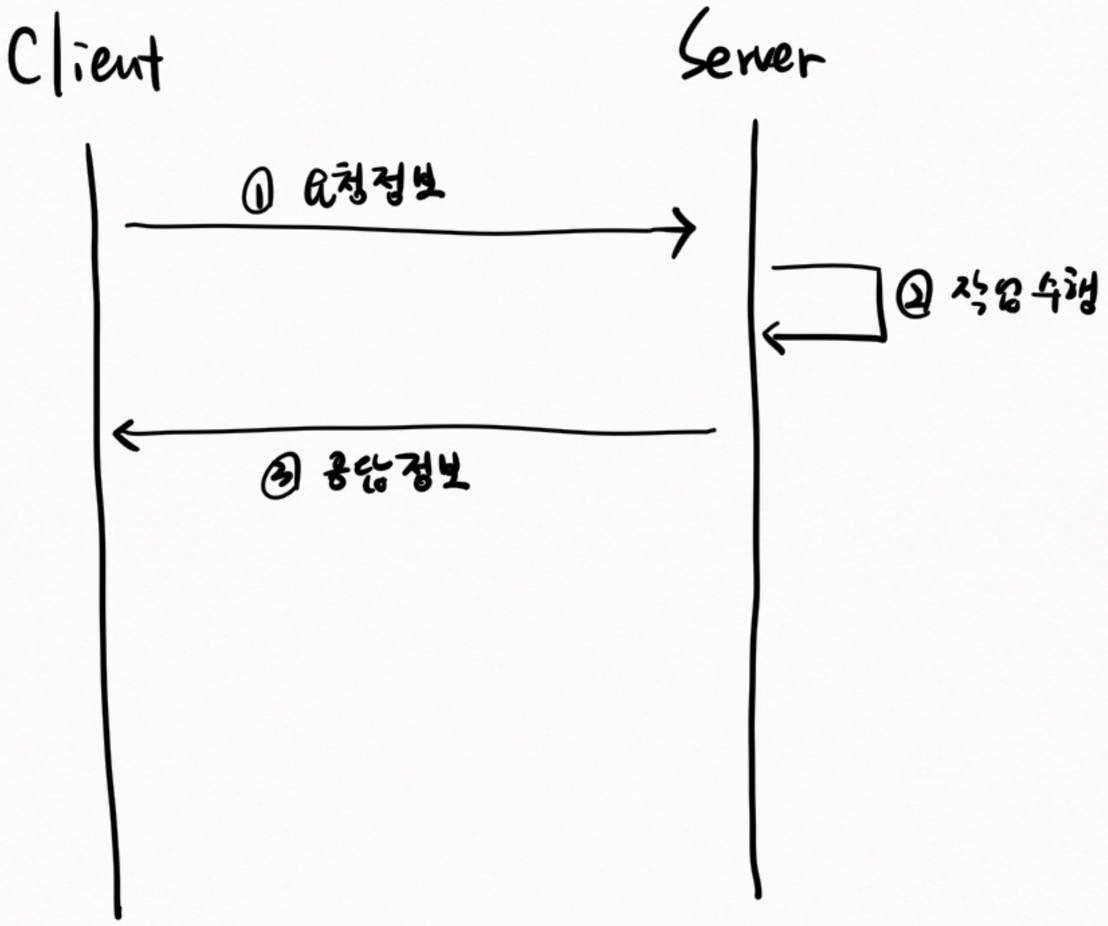


↳ 같은 (파일/데이터)를 동시에 접근할 때 발생할 수 있는 문제를 피할 수 있다!

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



* client / server 통신 규칙 (protocol)



* client / server 통신 규칙(protocol)

① 요청 정보 (JSON 문자열)

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

"테이블이름 / 명령"

게시글 : board
회원 : member
독서록 : reading

서버의 DAO 메서드명.

⇓ 예

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

⇐

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 에

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

* 실행결과

"success" : 성공

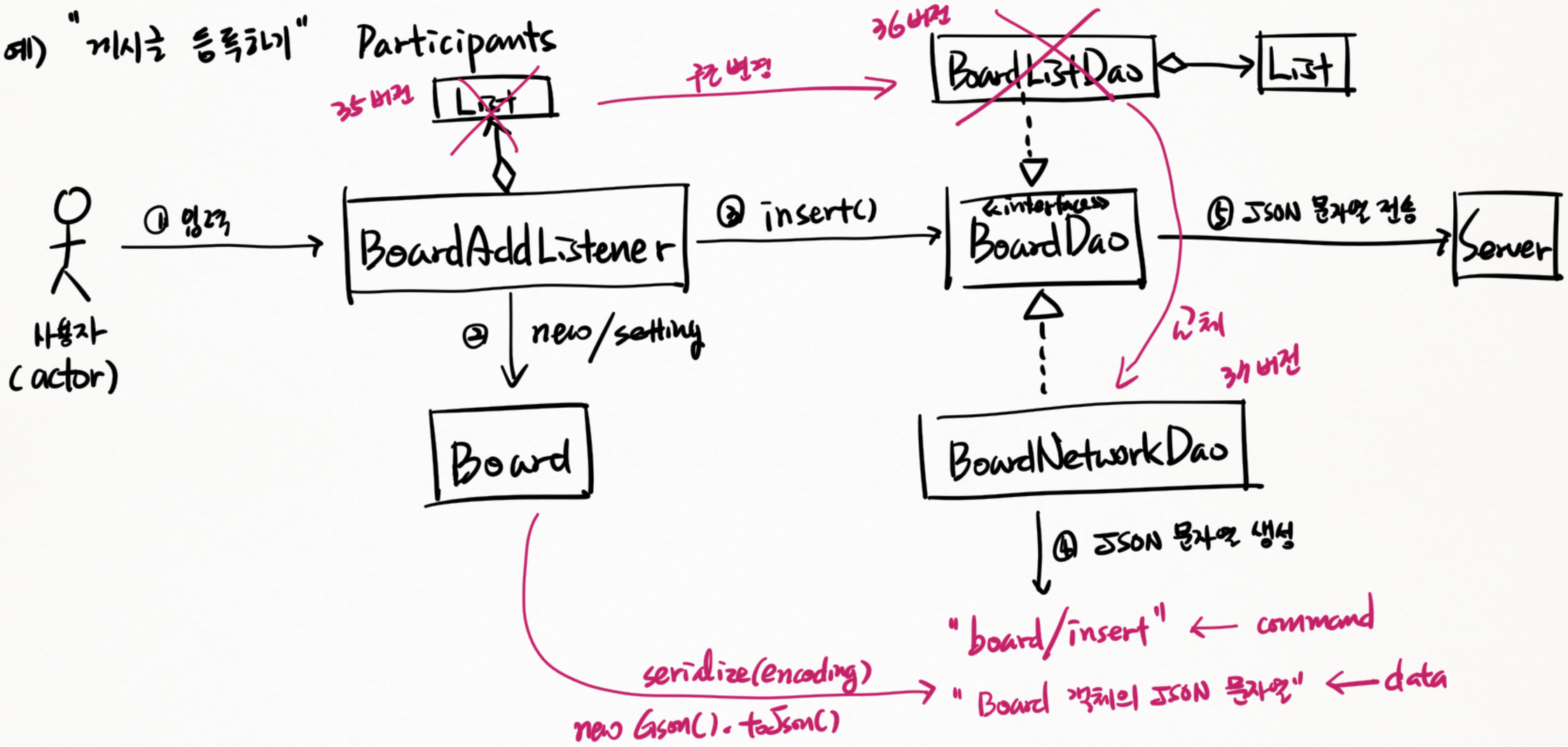
"failure" : 실패

⇓ 실패 에

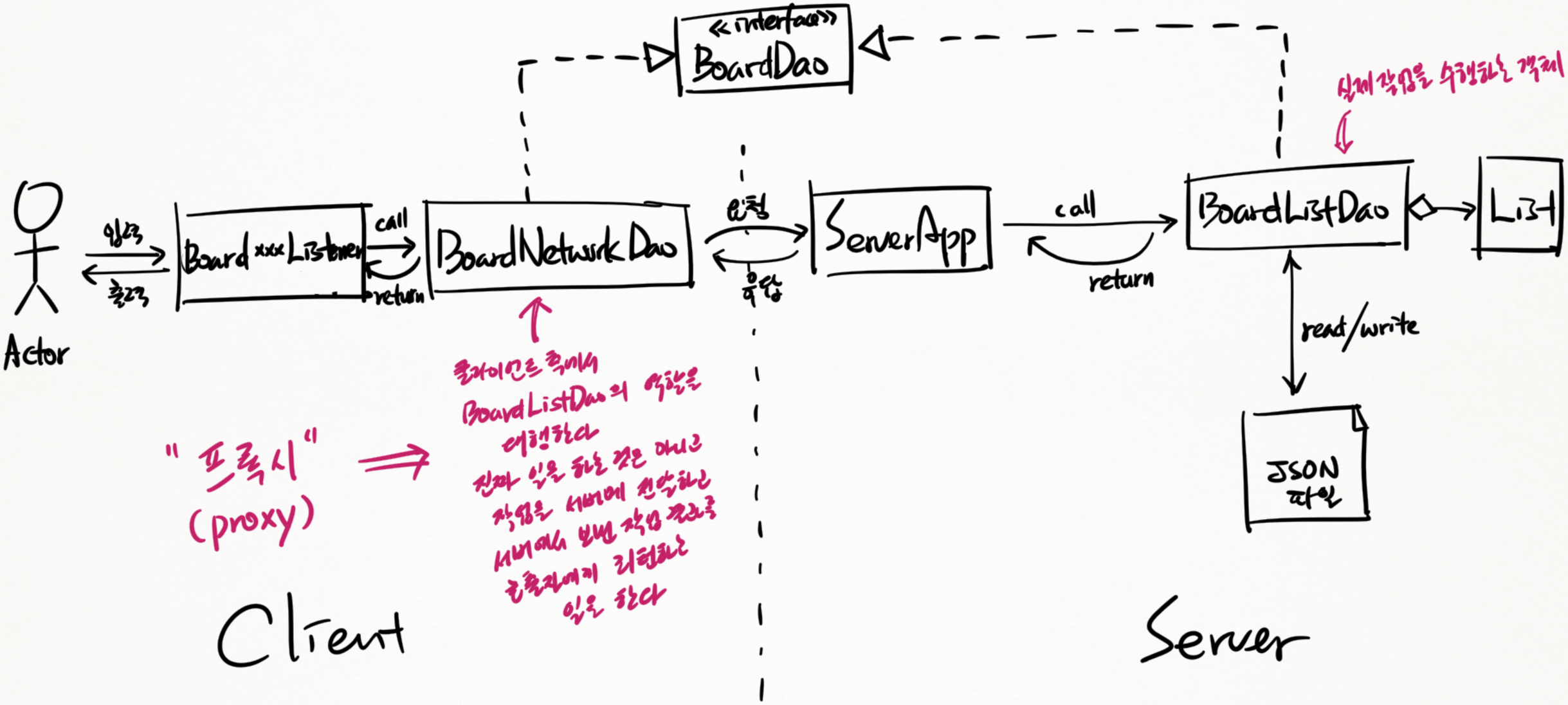
```
{
    "status" : "failure",
    "result" : "해당 번호의 데이터가 없습니다!"
}
```

* 요청 정보 보내기 작업에 참여하는 객체들과 실행 흐름

예) "게시글 등록하기" Participants

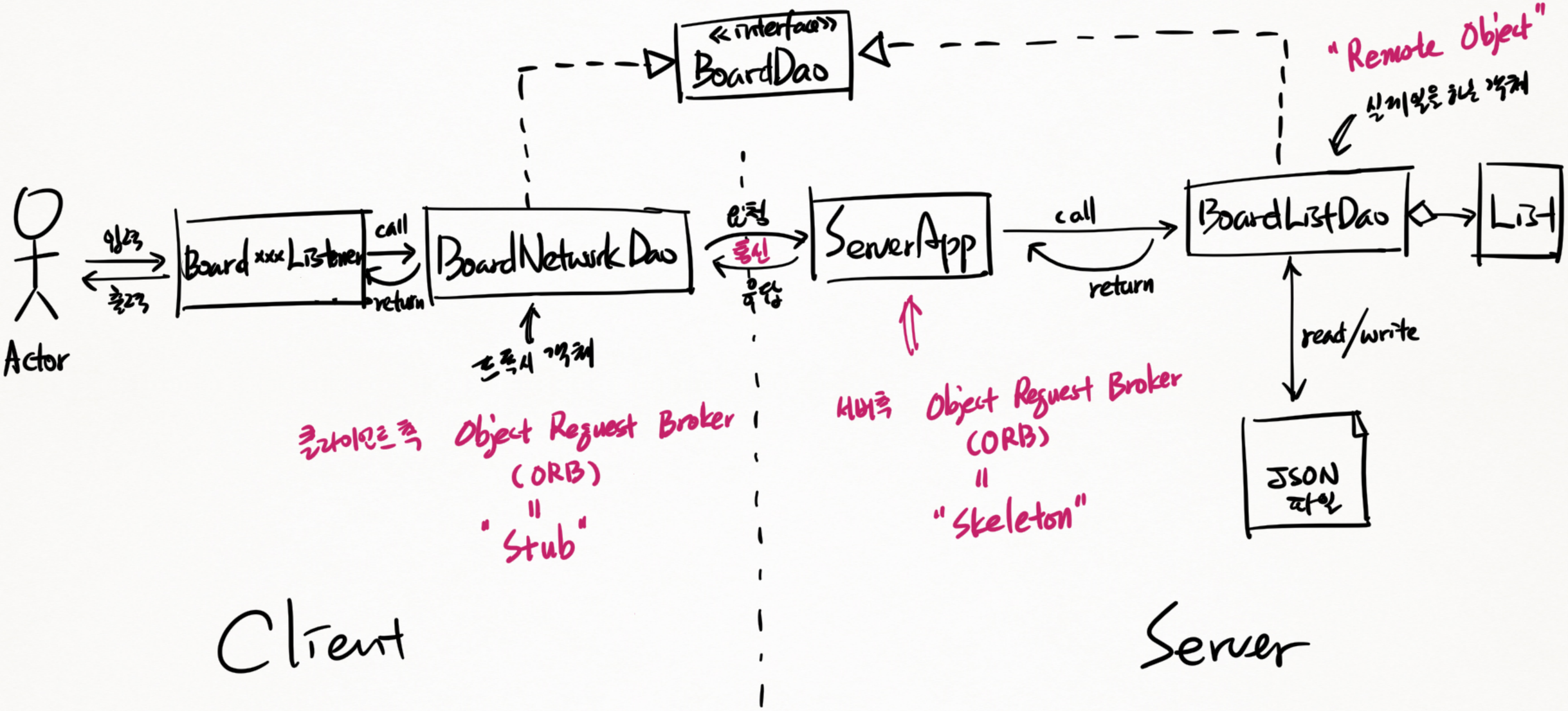


* DAO 와 Proxy 패턴 (GoF)



* DAO 와 Proxy 패턴 (GoF)

각 역할자의 명칭



* 분산 컴퓨팅

① 중앙 집중식 컴퓨팅 → ② 분산 컴퓨팅 ⇒ Rest API (RESTful)

↓

AJAX, XML/JSON

Web → HTTP

- 전용 API

Web Service
- WSDL
- SOAP
- UDDI
- XML

+Web

Java EJB

이기종 통합

CORBA (Common ORB Architecture)
- IIOP

OOP { C++, Pascal

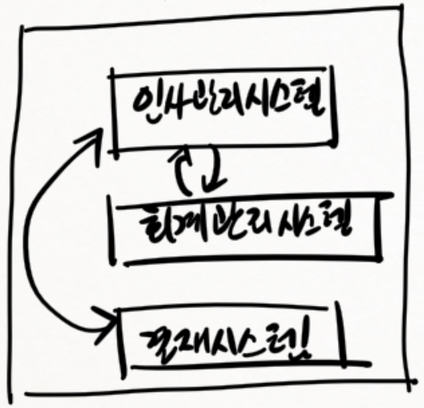
RMI (Remote Method Invocation)

C

RPC (Remote Procedure Call)

해결책

서버 (메인 프레임)



크기↓ 가격↓ 용량↓ (down sizing)

(워크스테이션) 서버1 — 인사관리시스템

↳ Unix

↳ Solaris, HP-UX, IBM AIX, ...

(워크스테이션) 서버2 — 회계관리시스템

(워크스테이션) 서버3 — 판매시스템

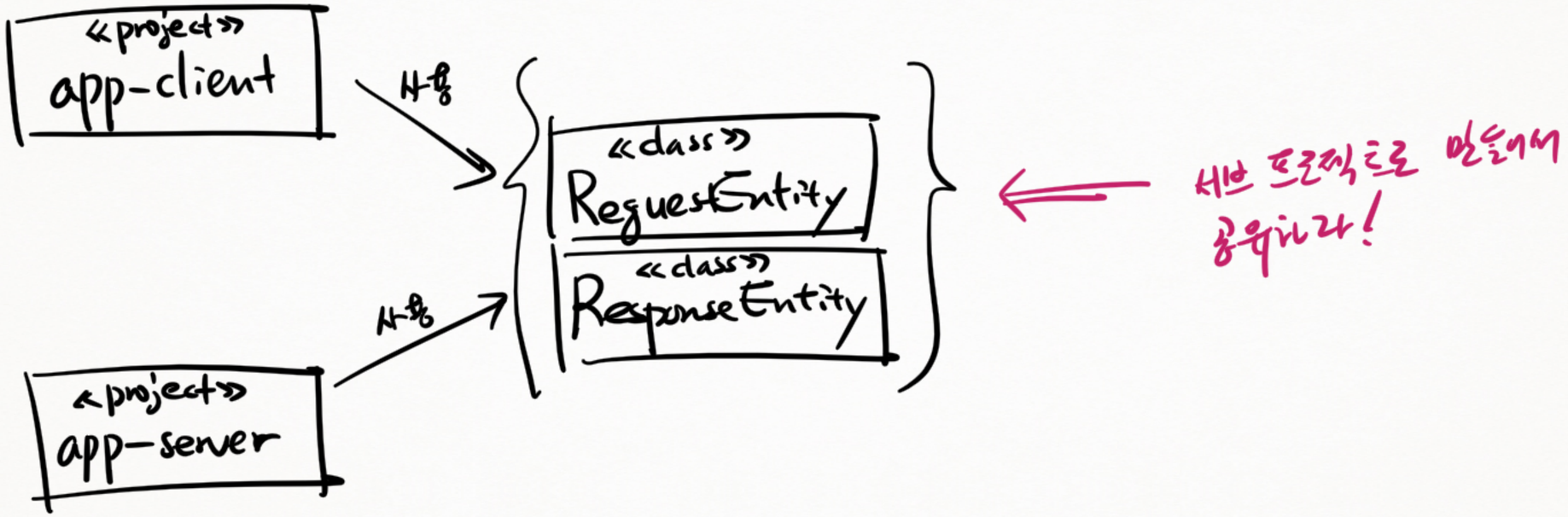
call

call

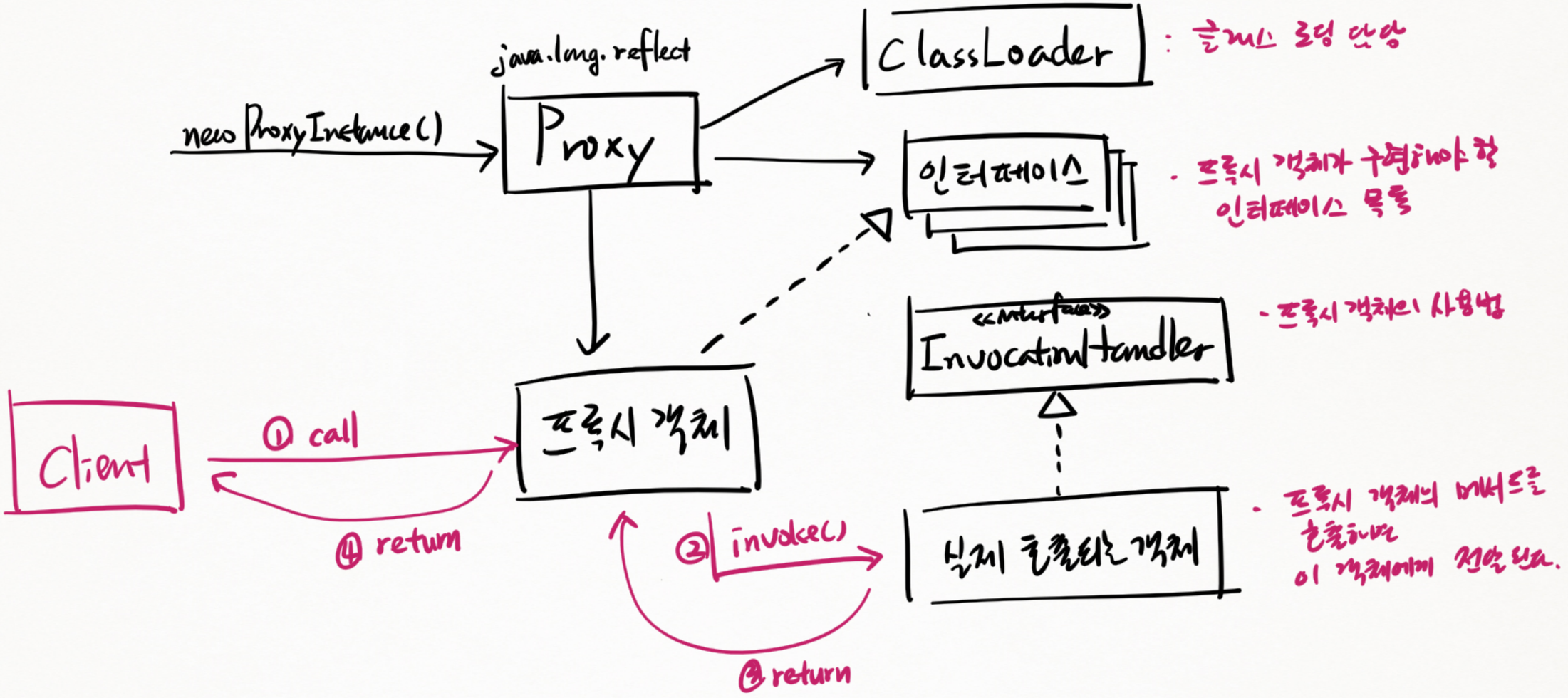
문제점
- 서버에 문제 발생
  ↓
  모든 업무시스템이 정지
- 사용자 증가 → 서버 증설 (RAM, HDD, CPU)
  ↓
  서버 장비는 고가
  ↓
  H/W 유지비가 많이든다.

문제점
- 다른 시스템의 코드를 실행할 수 없다
  ↓
  네트워크를 이용하여 데이터 통신
  ↓
  개발자/개발사 별로 구현
  ↓
  표준화 X → 유지보수 어렵다

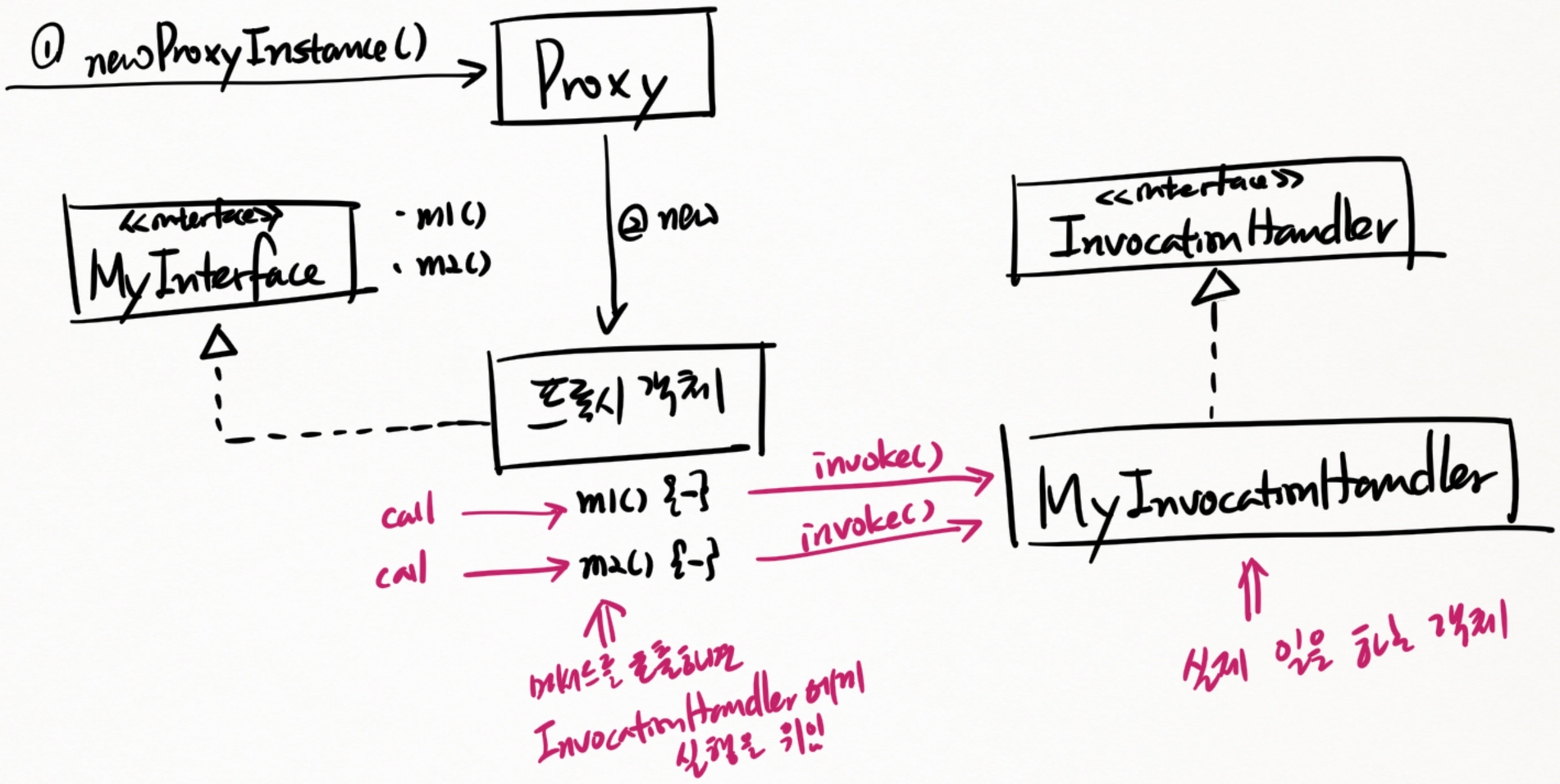
* Project 간에 클래스 공유하기



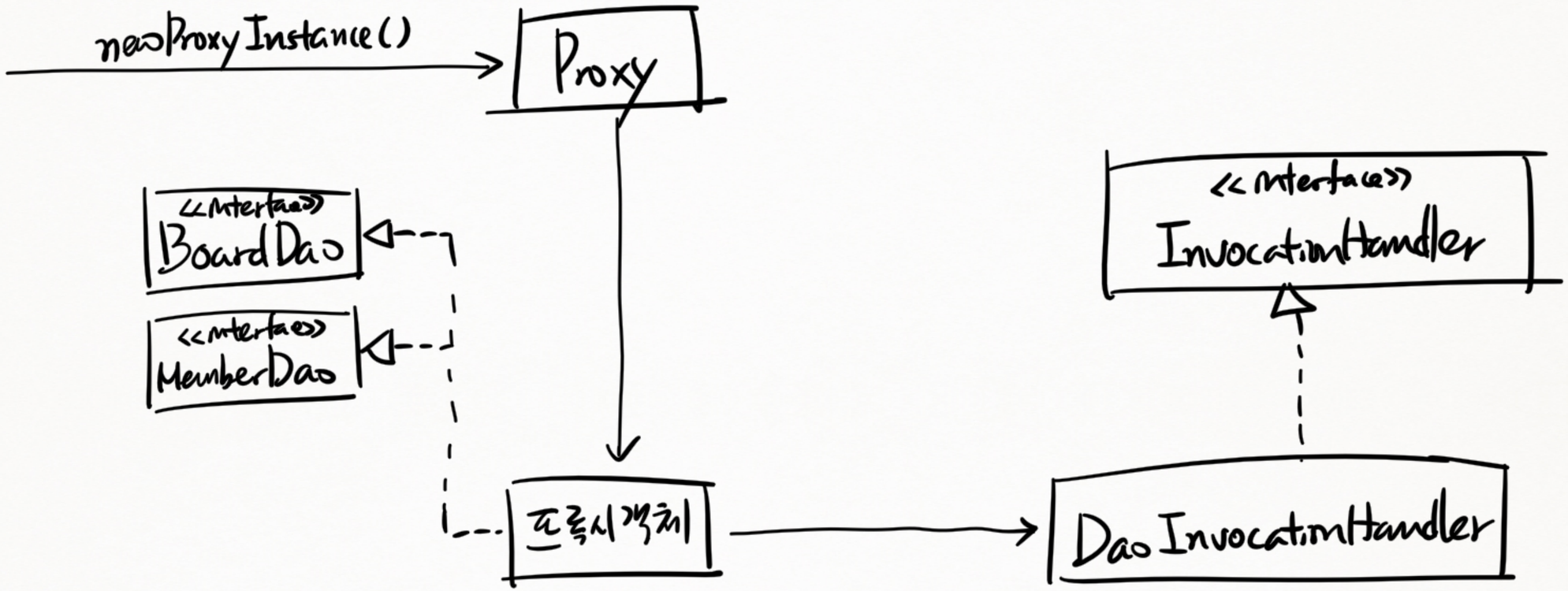
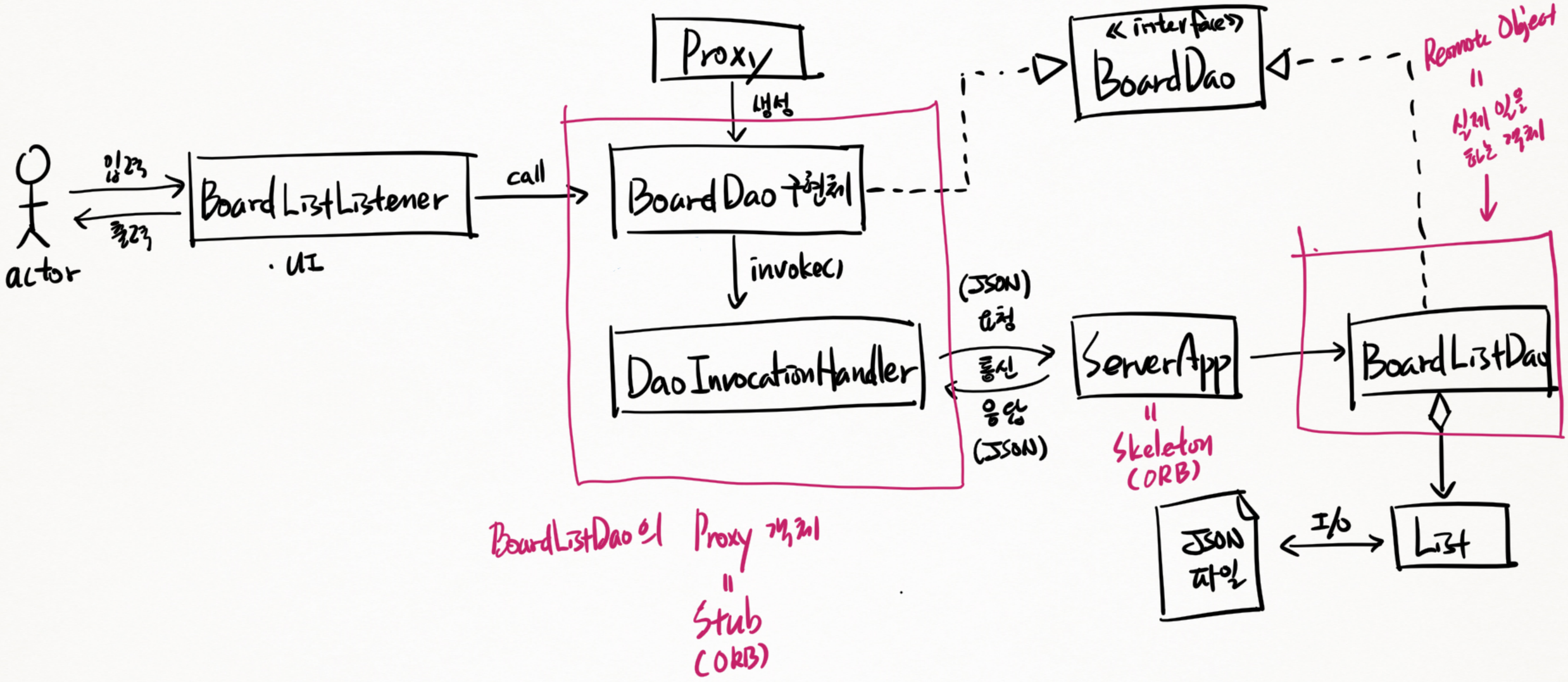
38. 프록시 객체 자동 생성



38. 프록시 객체 자동 생성 — 예1



38. 프록시 객체 자동 생성 - Participants



38. 프록시 객체 자동 생성 - Participants II