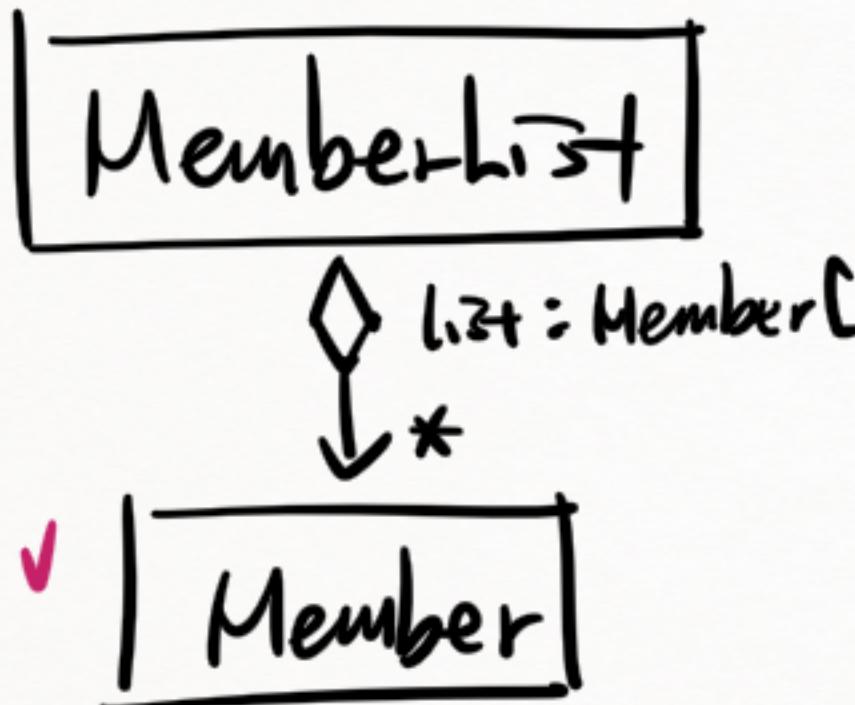


24. 제네리ك(Generic) 적용 : (Object 타입처럼 다양한 타입에 대응할 수 있다.)
특정타입으로 제한할 수 있다.

① 다형적 변수 적용 전

↳ 각 타입별로 List 클래스를 만든다



L101의 기능은 흡사
타입일 라즈다

근드 중복 말

* 08월

- * 이점 -

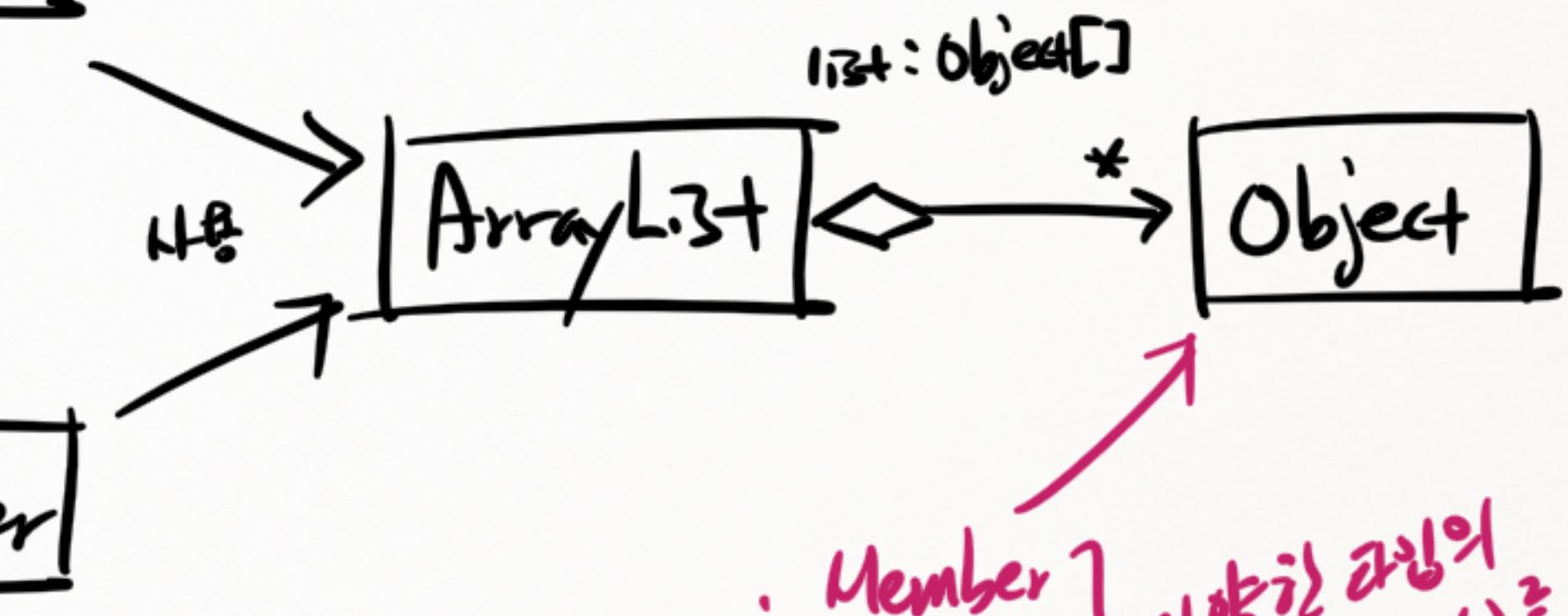
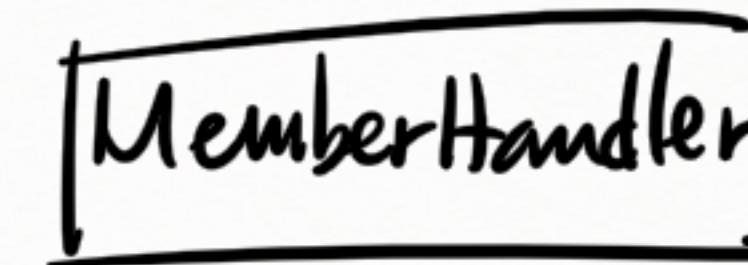
 - ① 인스턴트를 깨끗이 한다
청결한 점!
 - ② 투명 라임과 다진으로
제작할 수 있다.

↳ 타임 안정성을 해친다.

리스트로
만들어
준다!

① 다형적 변수 적용 전 —————→ ② 다형적 변수 적용

↳ 마지막 각 단원별로 클래스를 정의한 뒤에
해석을 볼 수 있다.



- Member
 - Board
 - :

} 다수인 단체의
인스턴스(각부)를
지정할 수 있다.

* Generic \Rightarrow Type Parameter

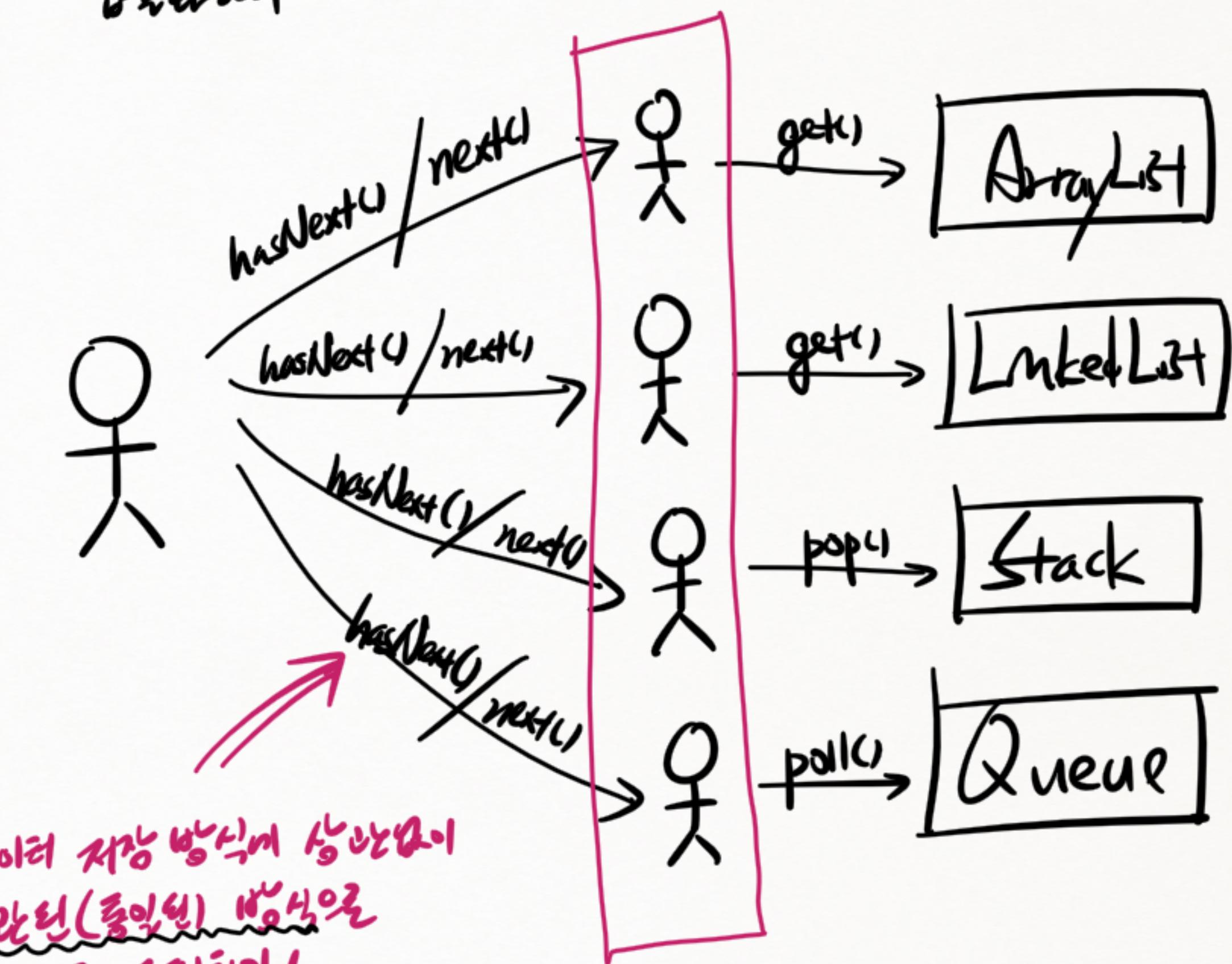
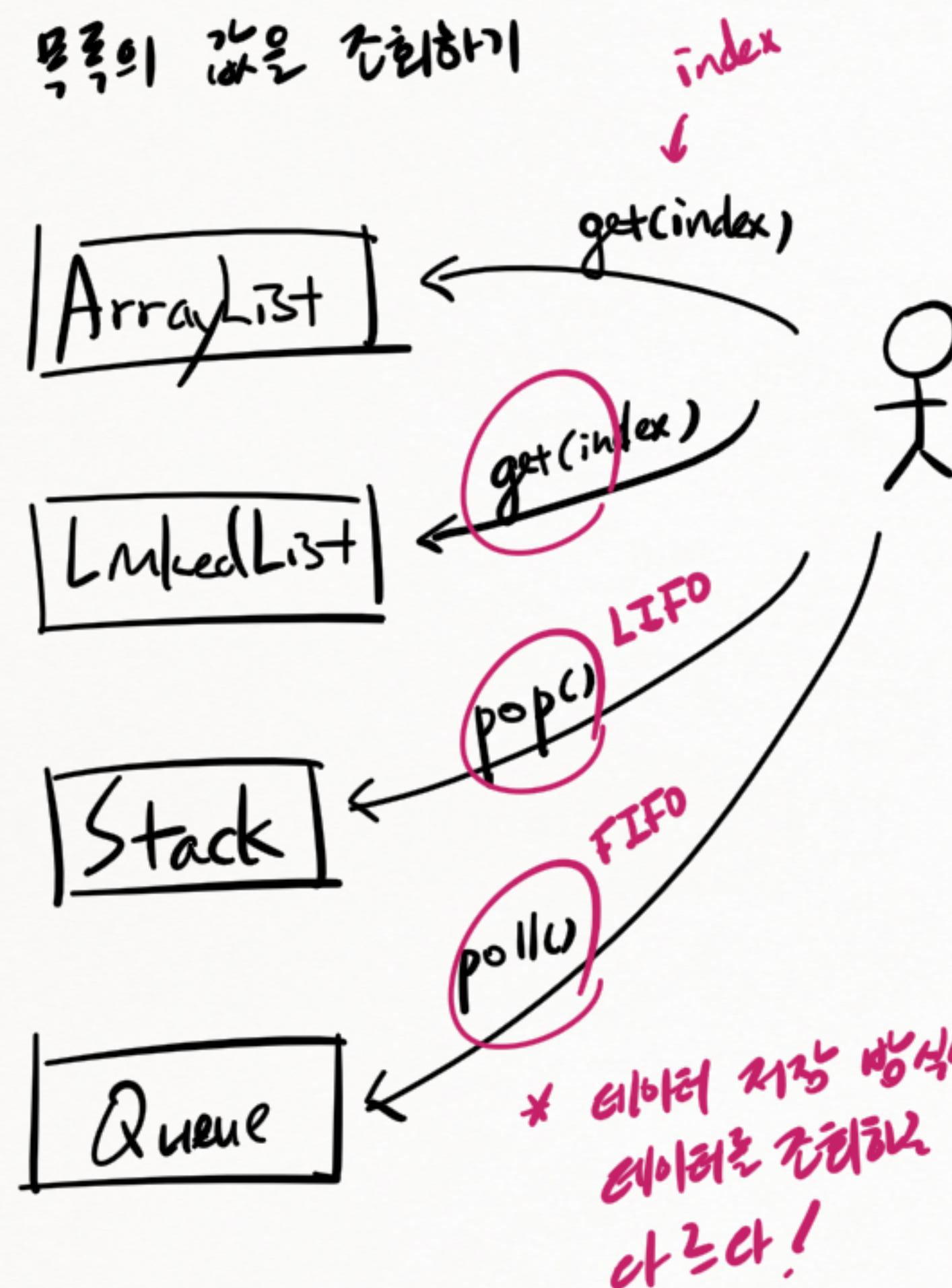
클래스의 타입 변수를 넣는다

```
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

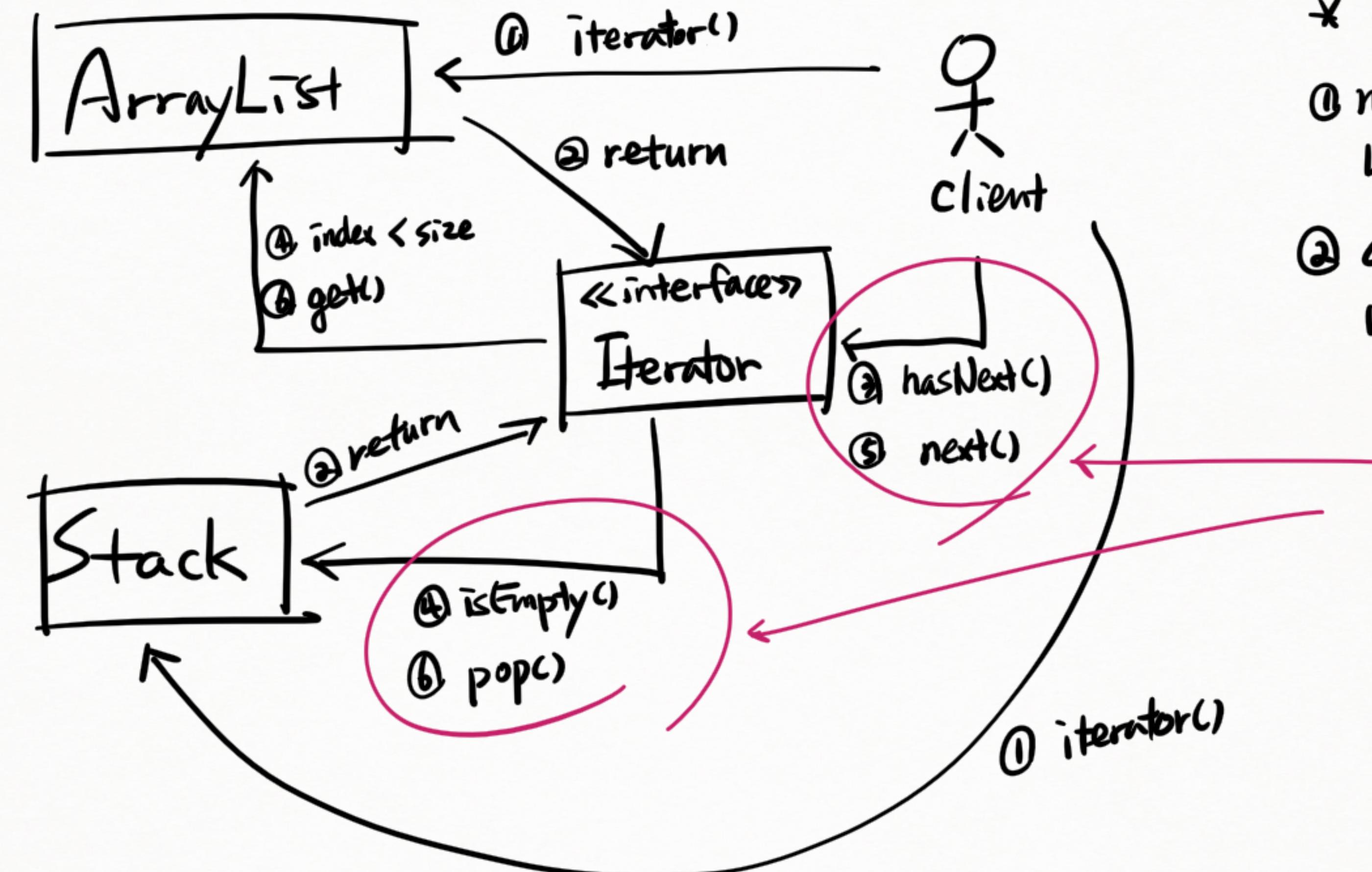
① 목록의 값을 조회하기



* 데이터 처리 방식에 따라
Iterator(증발된) 방식으로
데이터를 조회하기!

Iterator
(데이터를 순회하는 일을 하는 객체)

* Iterator mechanism (구조원리)

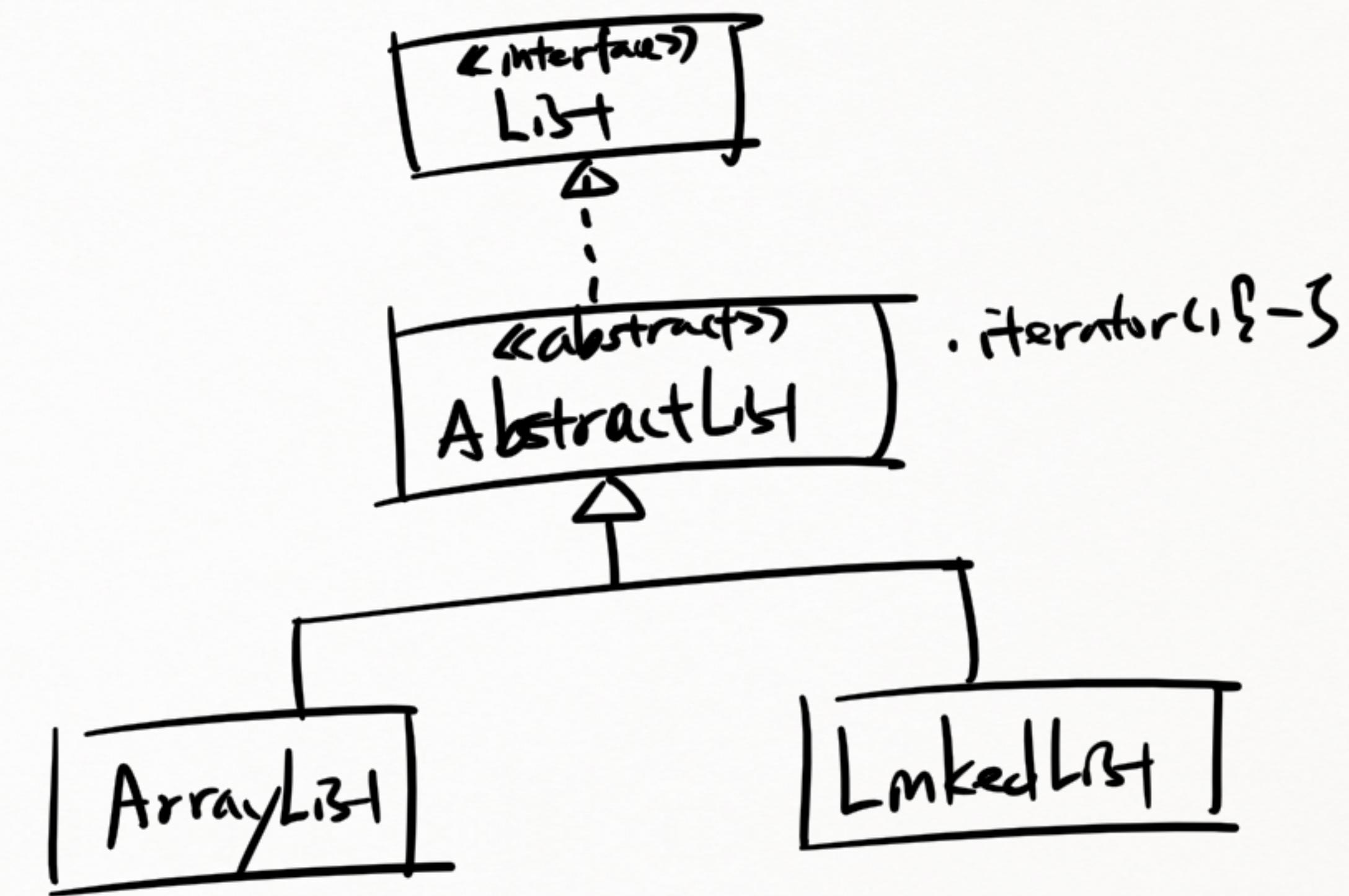
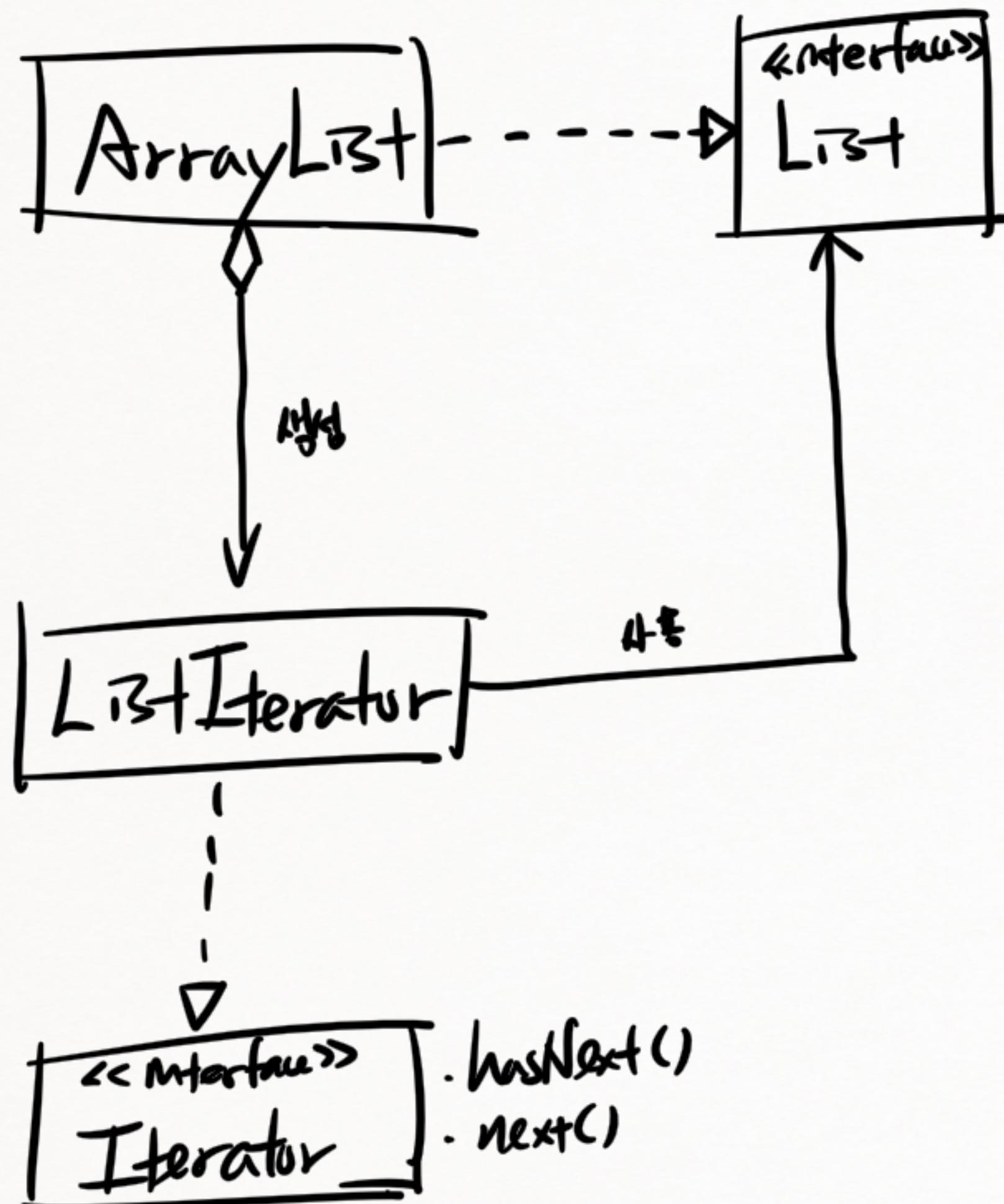


- * client
- ① network 분야
↳ 네트워크 요청으로 연결을 수행 S/W
- ② OOP 분야
↳ 다른 객체를 사용하는 개체.

제작업체가 상관없이
일관된 방식으로
작동을 가능케 한다!

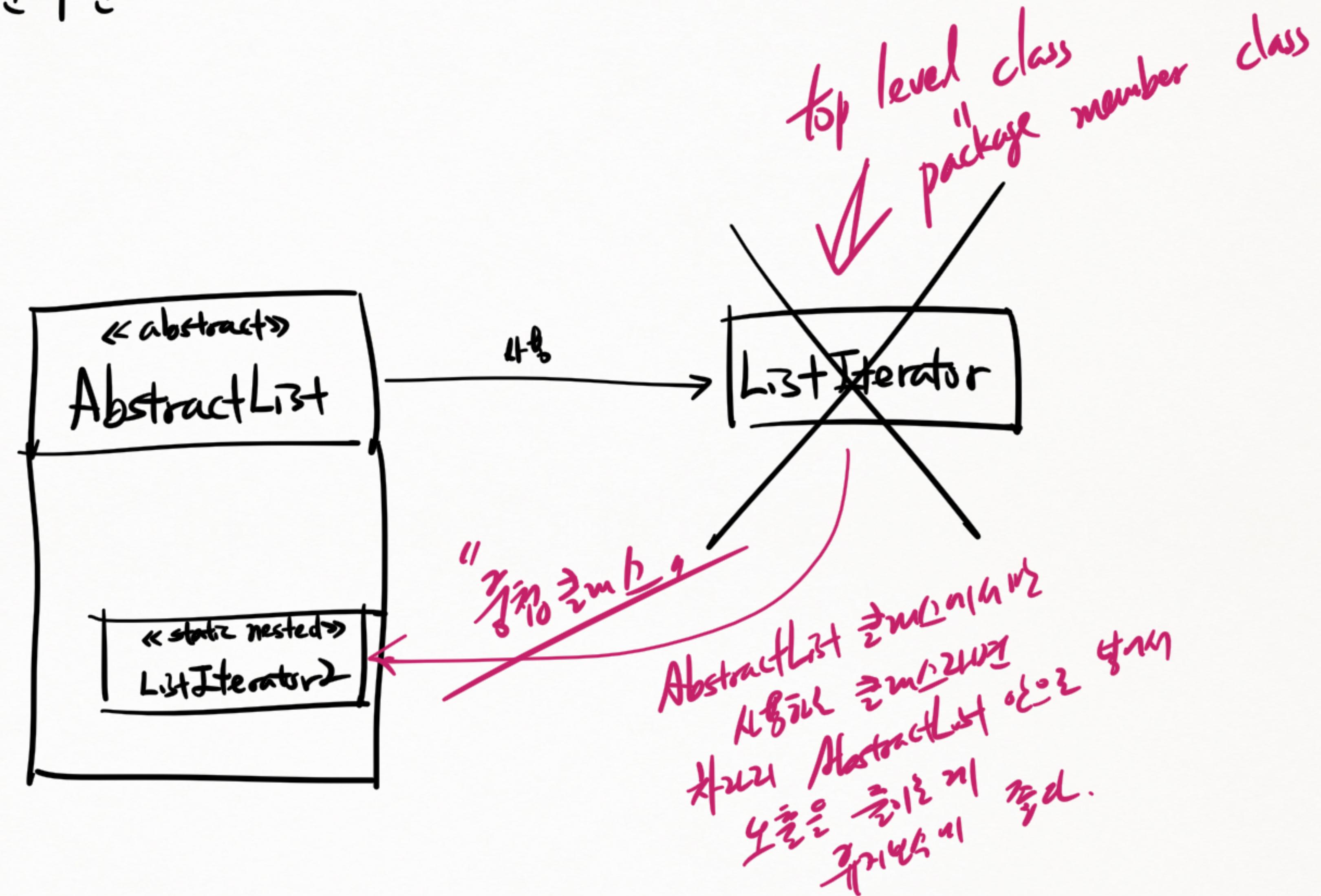
* Iterator 퀘션 구조

① 퀘션 - top level class



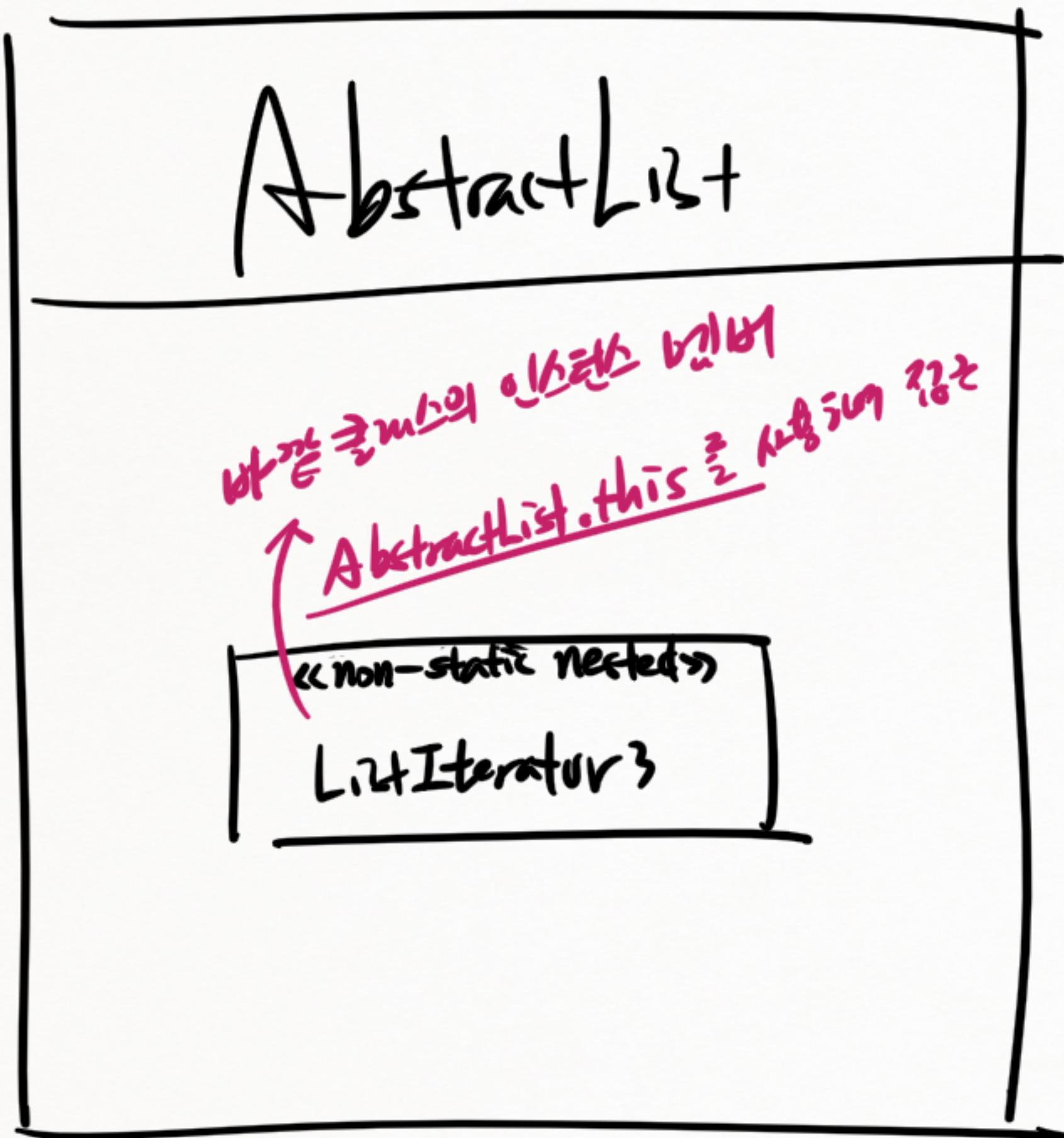
* Iterator 퀘션 구현

② 구현 — static nested class

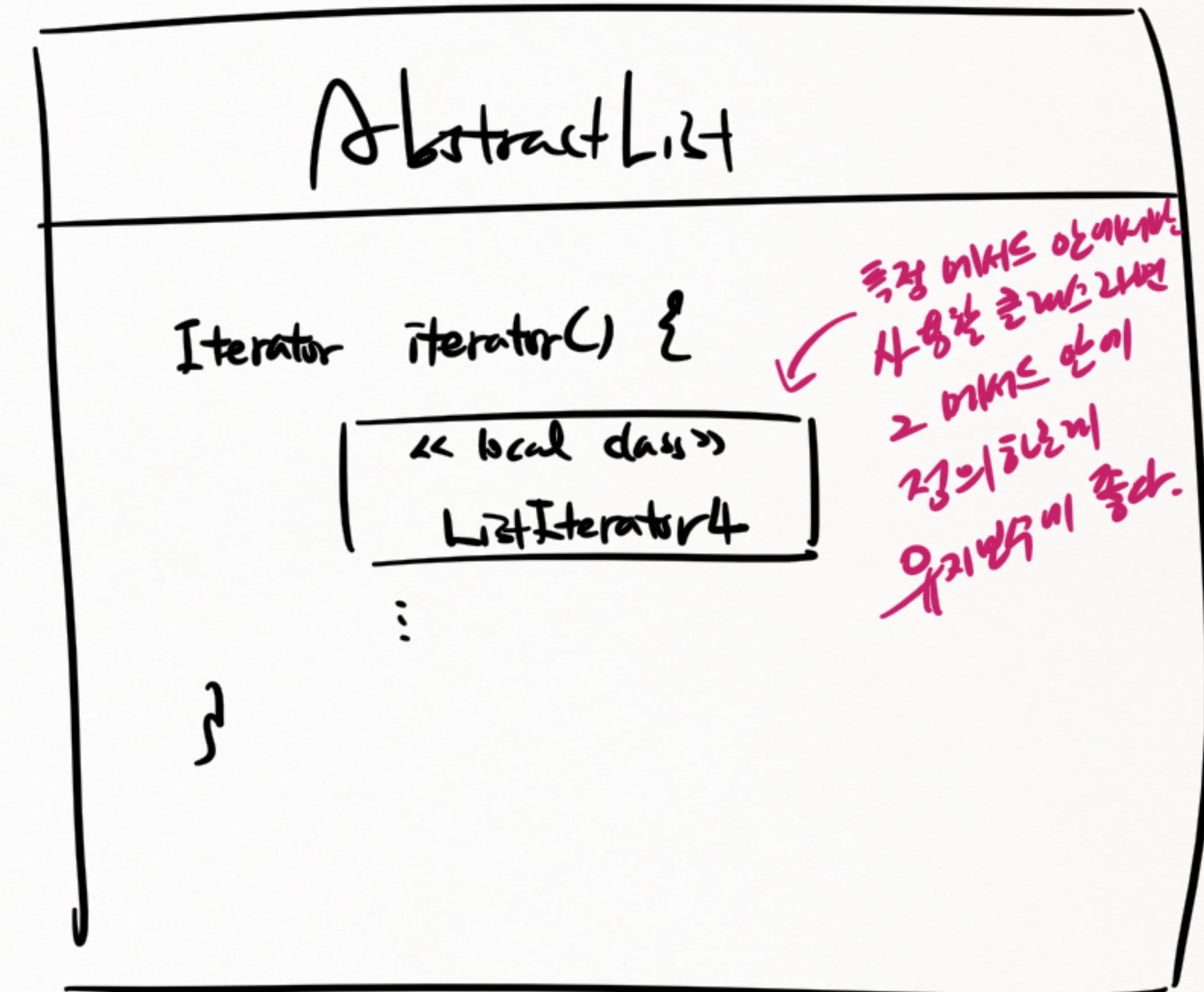


* Iterator 파편 구현

③ 구현 - non-static nested class

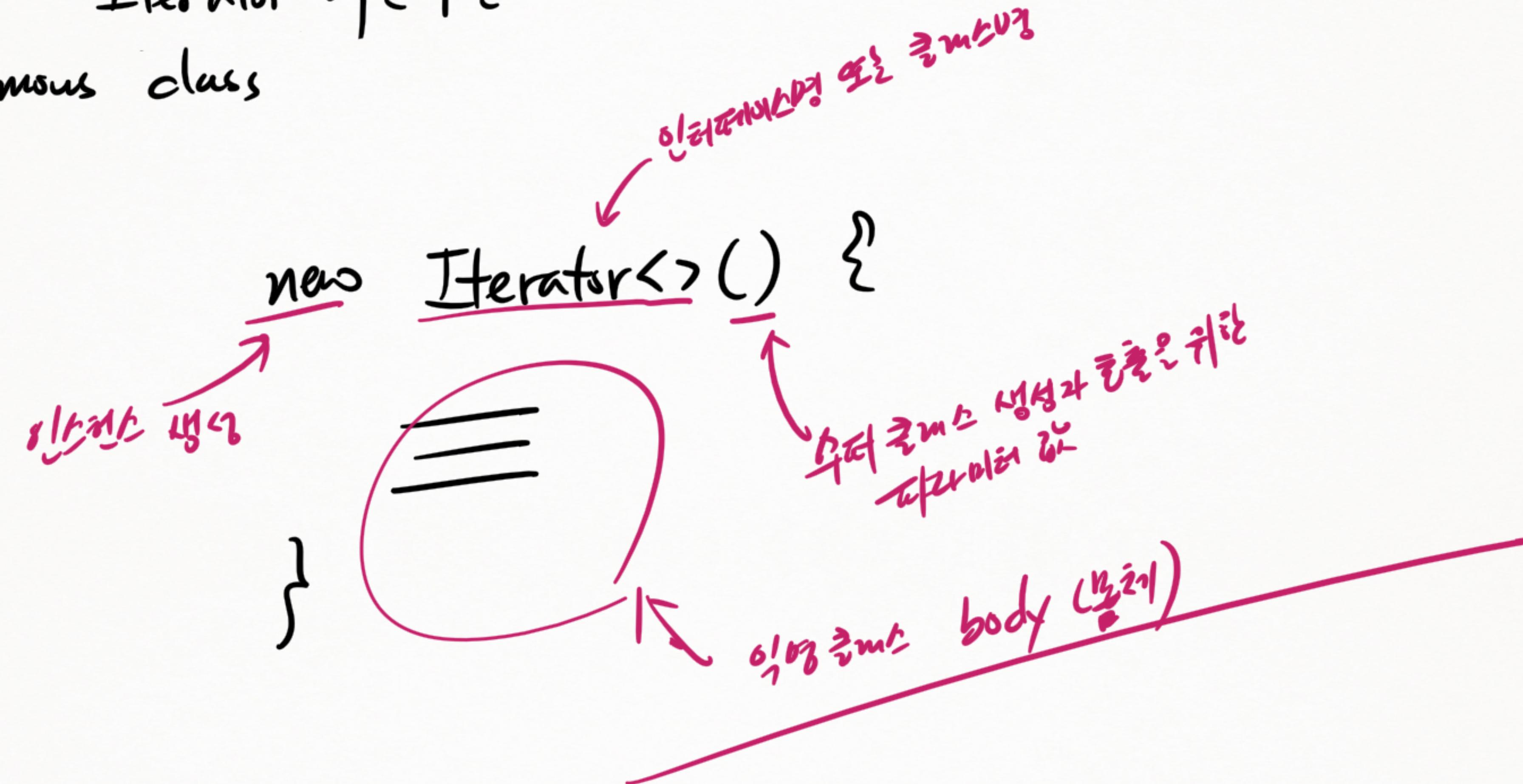


④ 구현 - local class



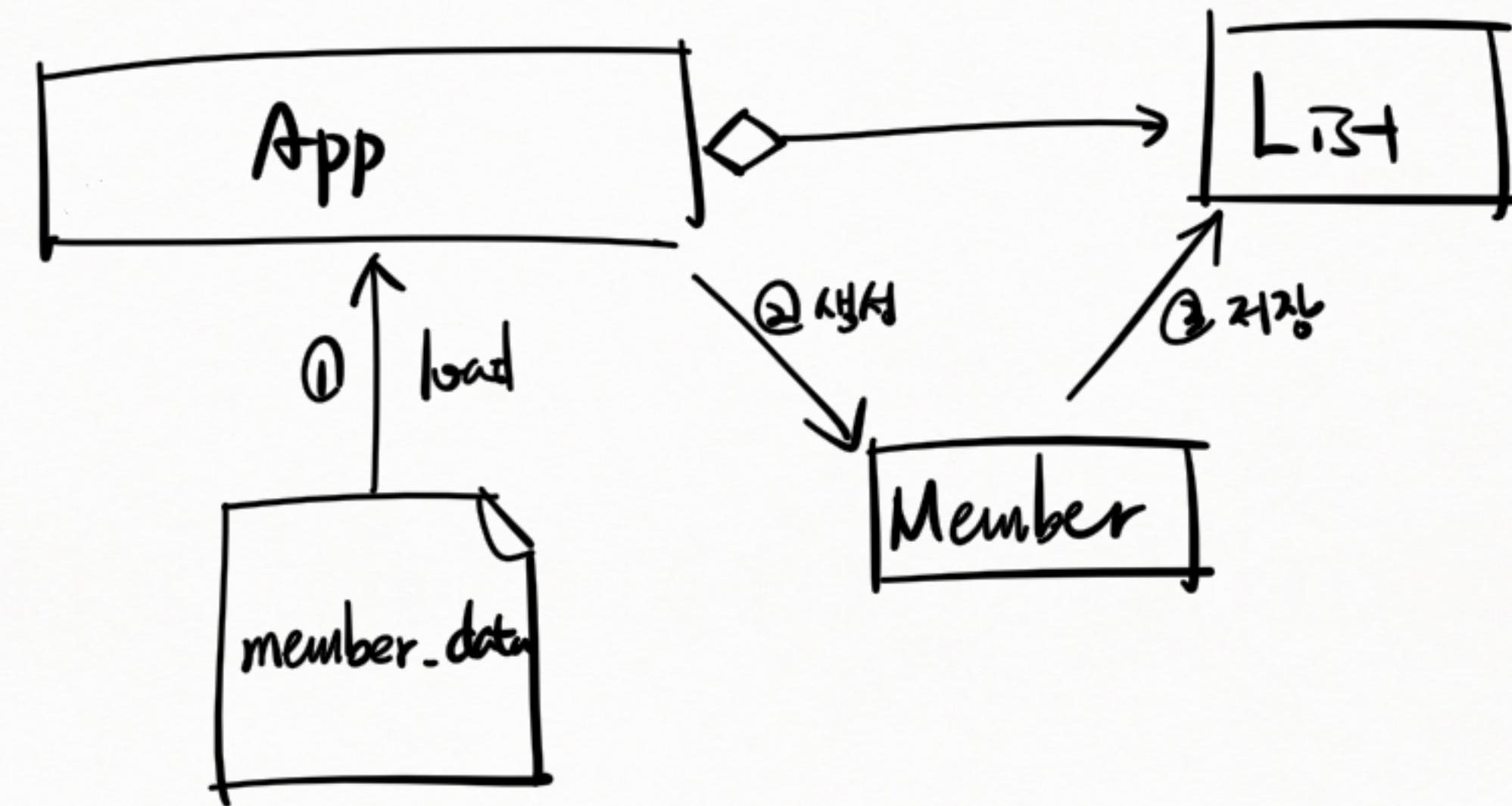
* Iterator 파편 구현

⑤ anonymous class



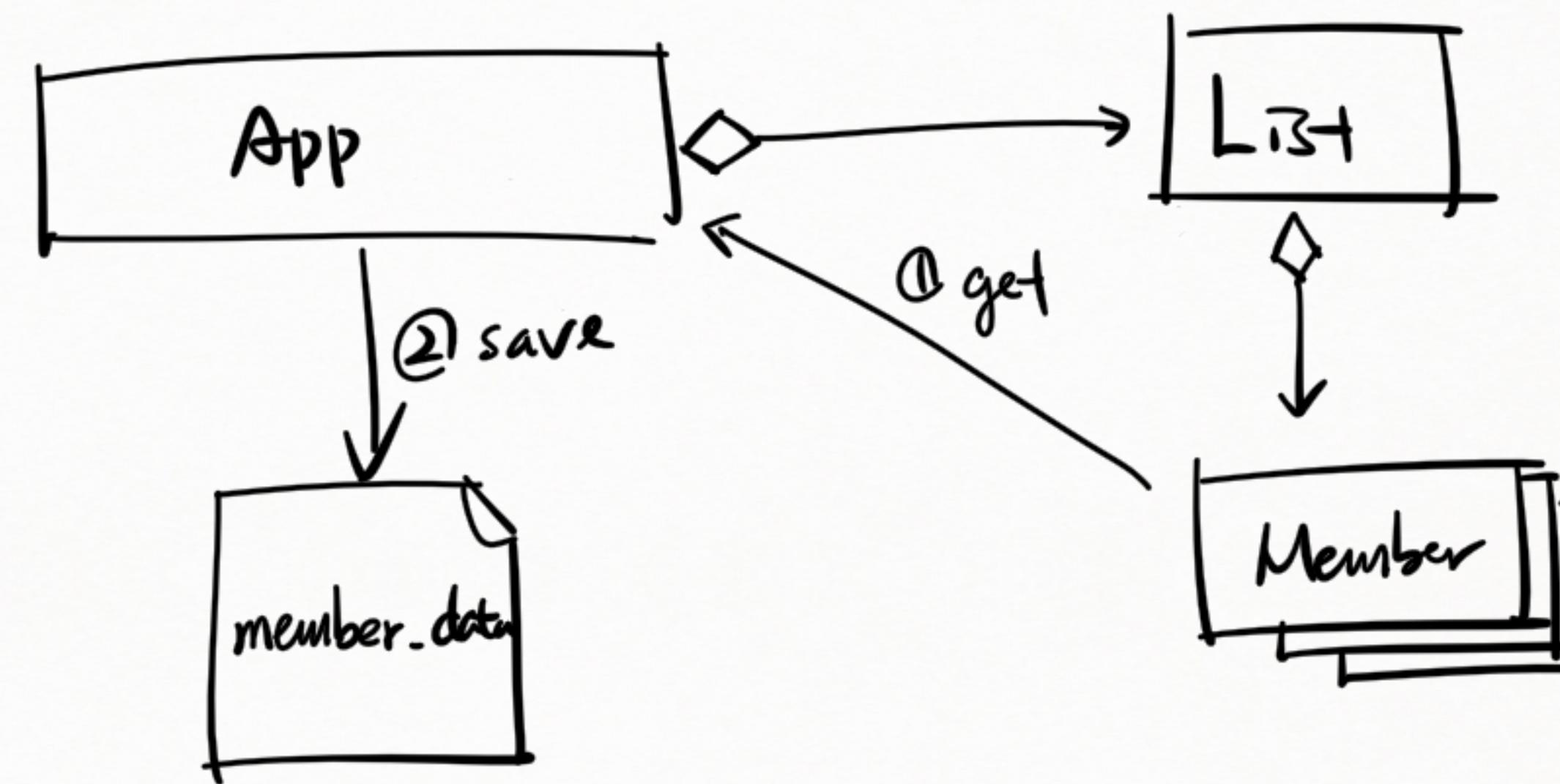
27. Data I/O Stream API - binary stream API 사용 ↳ Data 저장

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

↓ ↓

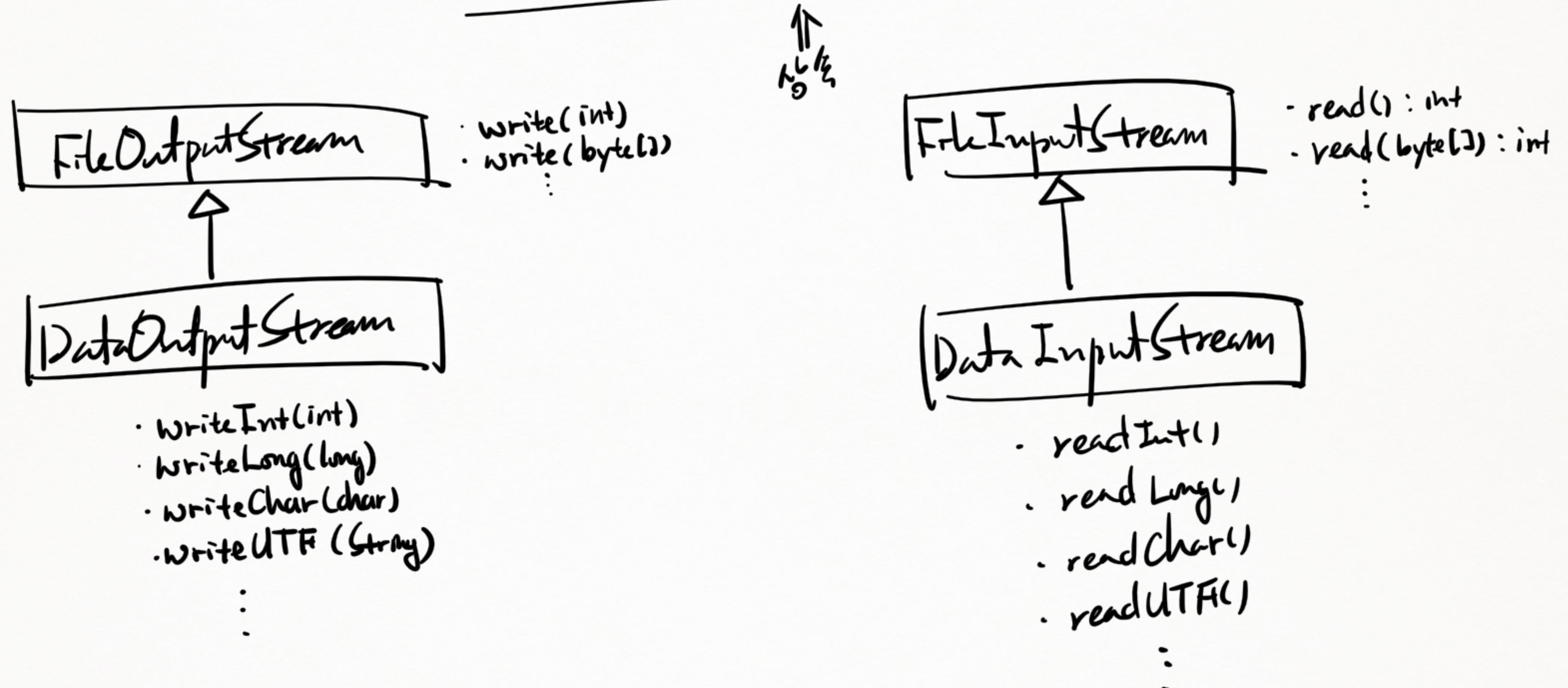
in.read(buf, 0, 3)

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

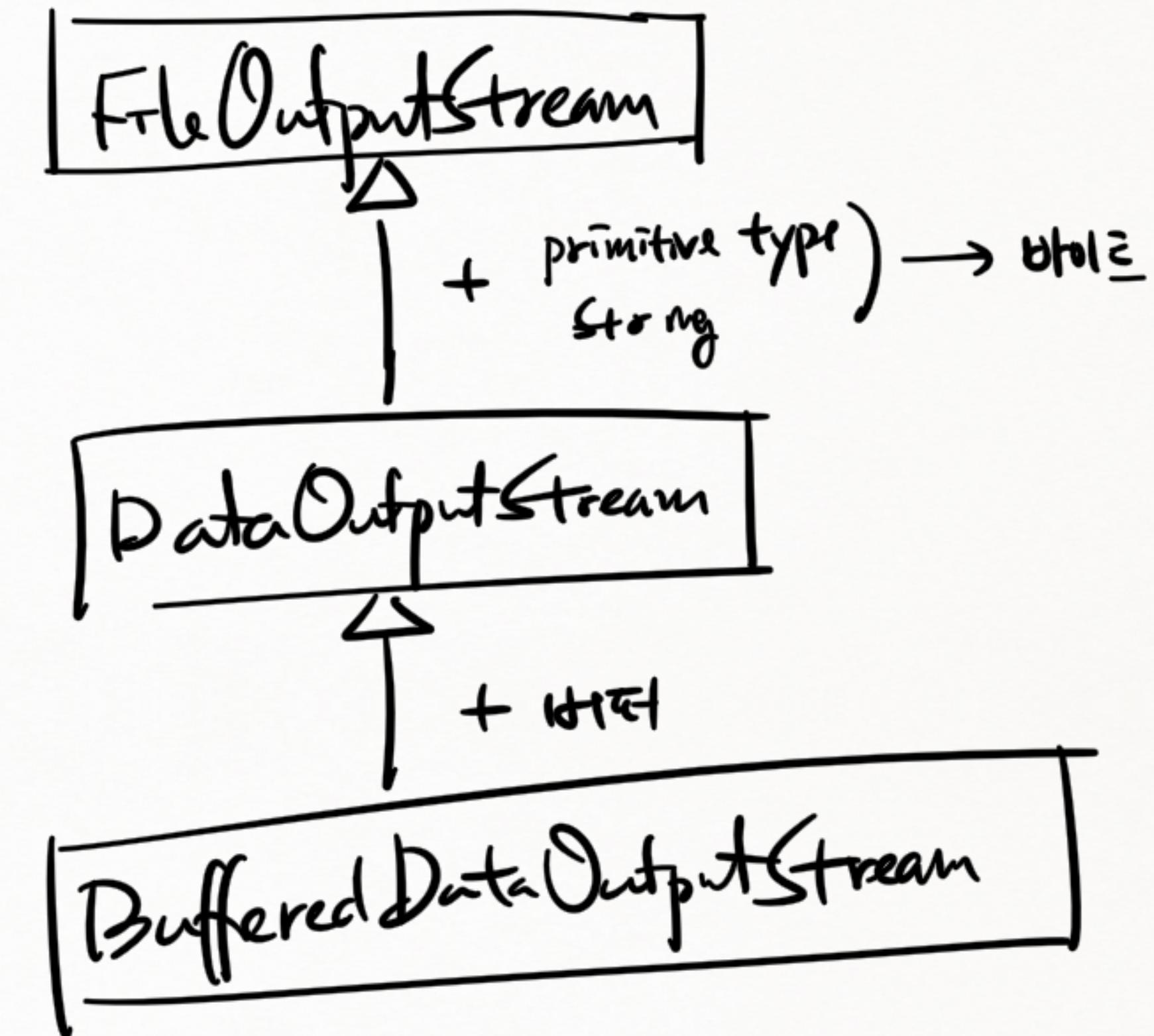
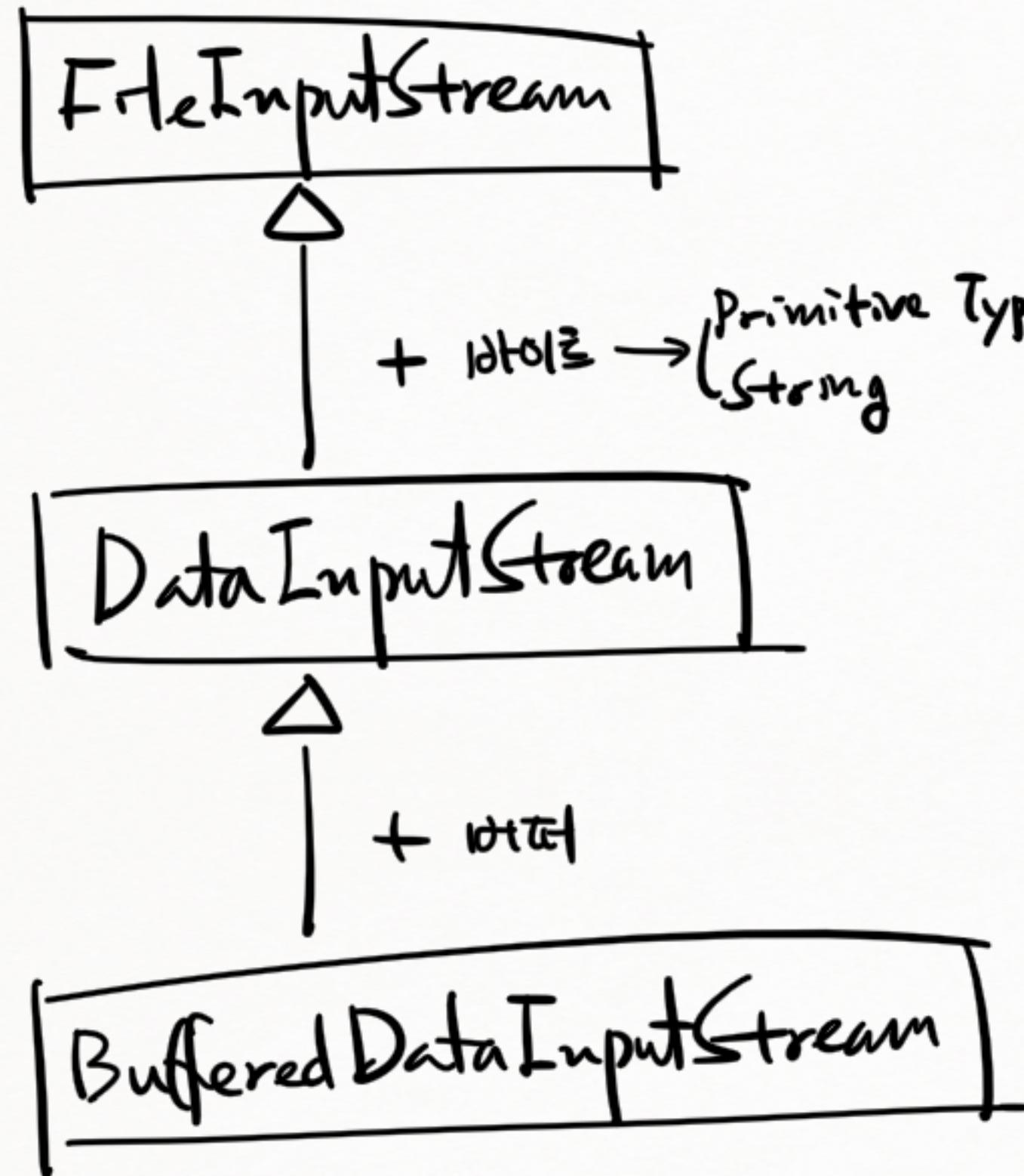
new String(, 0, count, "UTF-8")

member . setName()

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



29. 파일 입출력의 버퍼링 예제 : 파일 읽기 쓰기

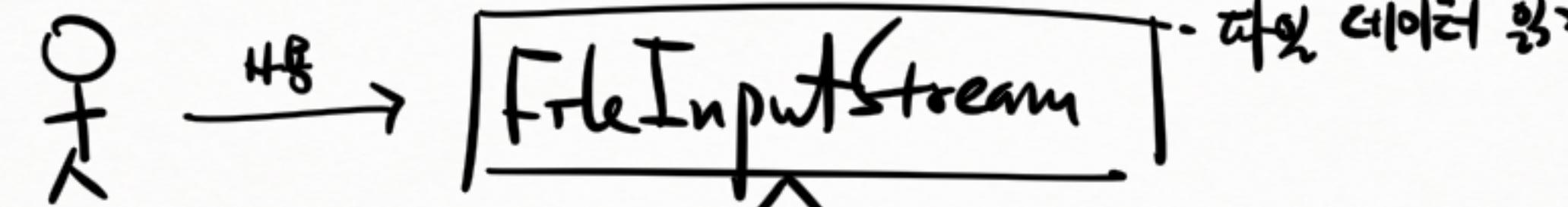


30. 기능을 확장할 때 상속과는 Decorator 패턴 적용

① 상속을 이용한 기능 확장의 문제점

- byte / byte[] 읽기

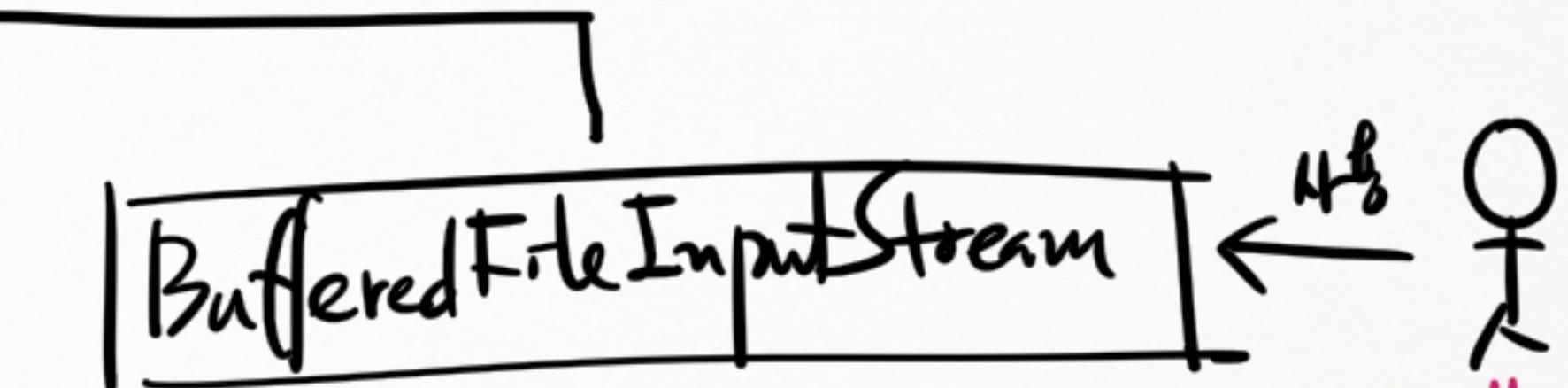
- * Primitive Type / String 읽기 불편



- (primitive type) 읽기 편함

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

- 버퍼를 이용해서 읽기 성능 개선



* 상속은 좋은 유형의 기능을
제공한 'Fact.'

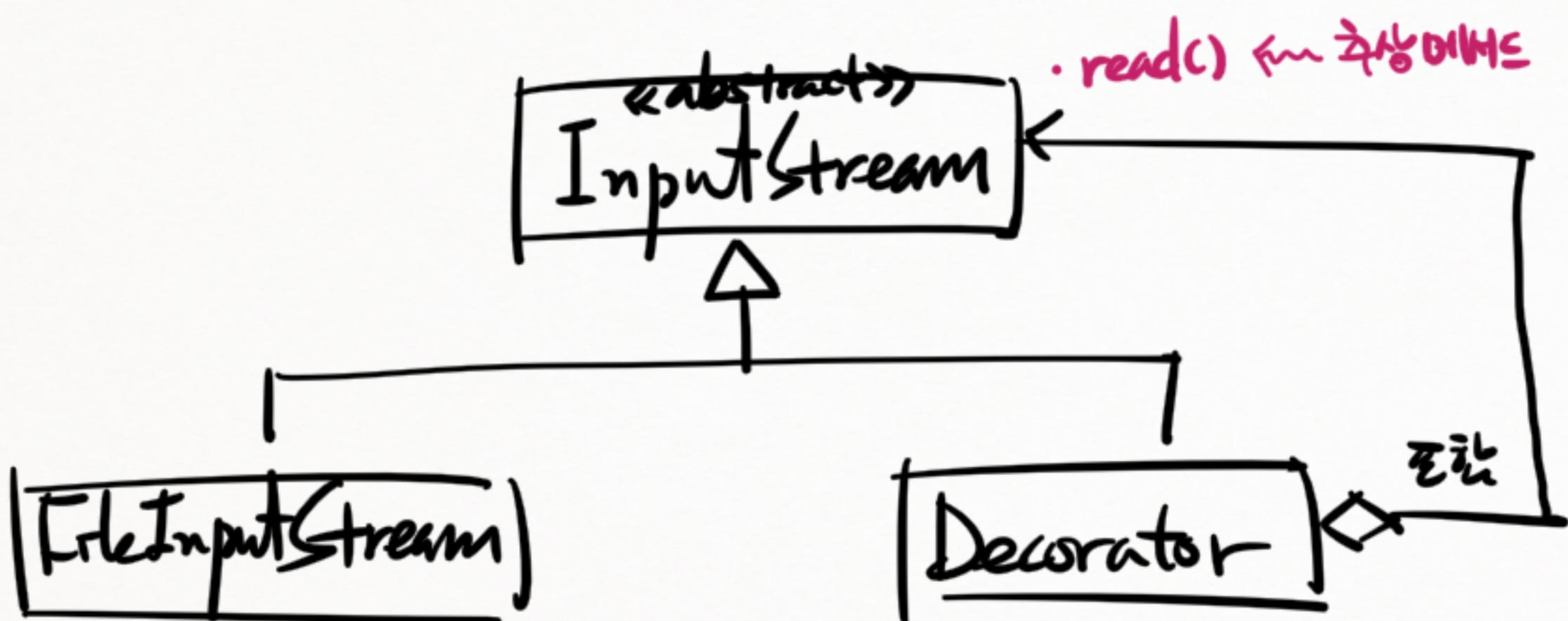
그러나 상속의 sub 타입은 오직 한 가지만 가능하다!

↳ 이것이 상속의 한계!

(primitive type)의 데이터는
읽을 때마다 많은 성능이 사용될 경우,
바이트 단위로 데이터를 읽을 때
데이터를 통해 데이터를 읽을 때
읽기 성능은 개선된다.
하지만.

30. 기능을 확장할 때 쌍축 with Decorator 티켓 적용

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

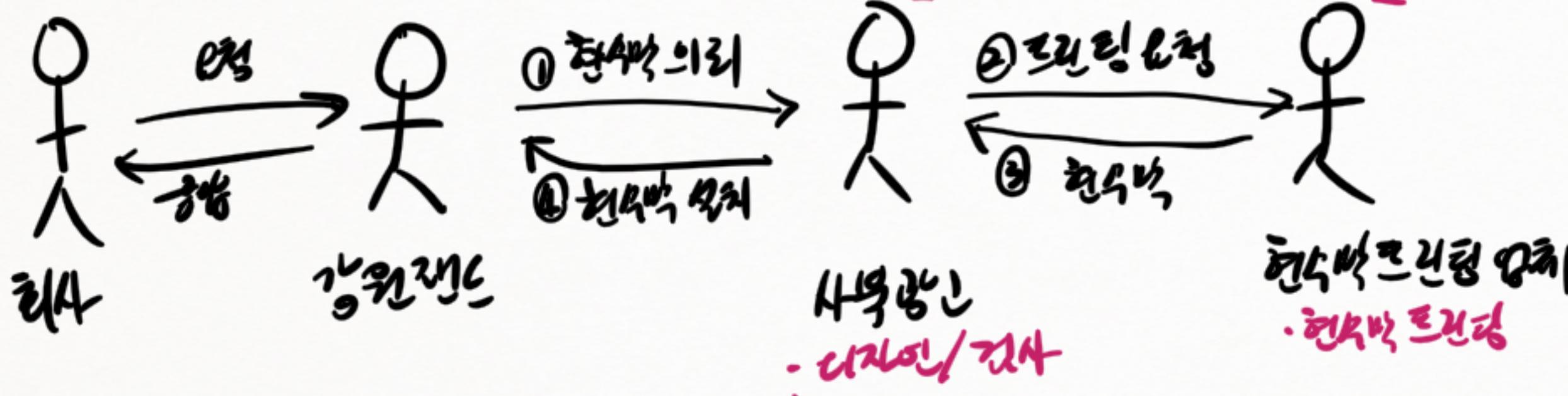


a) DataInputStream

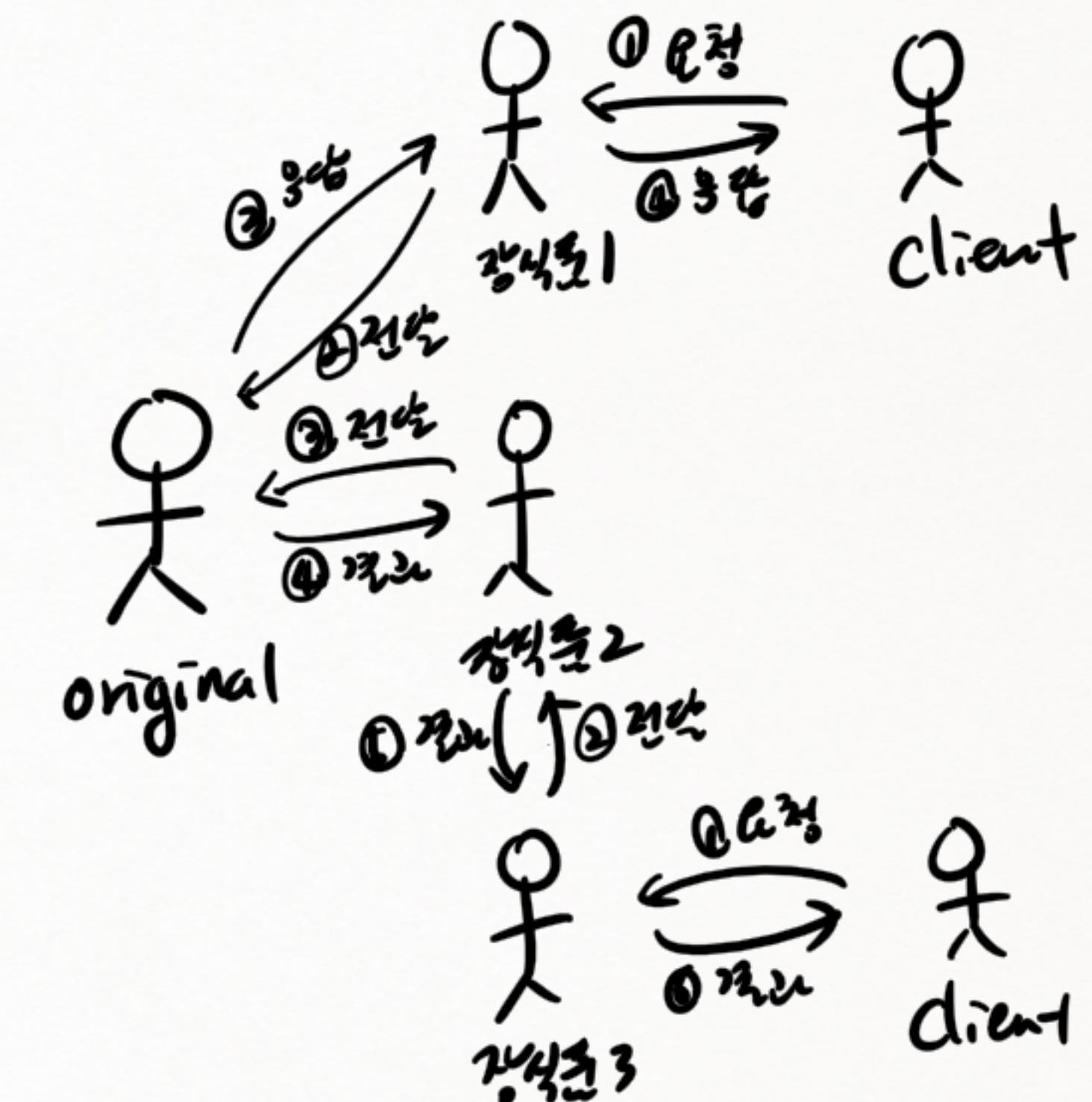
BufferedInputStream

✓ Decorator

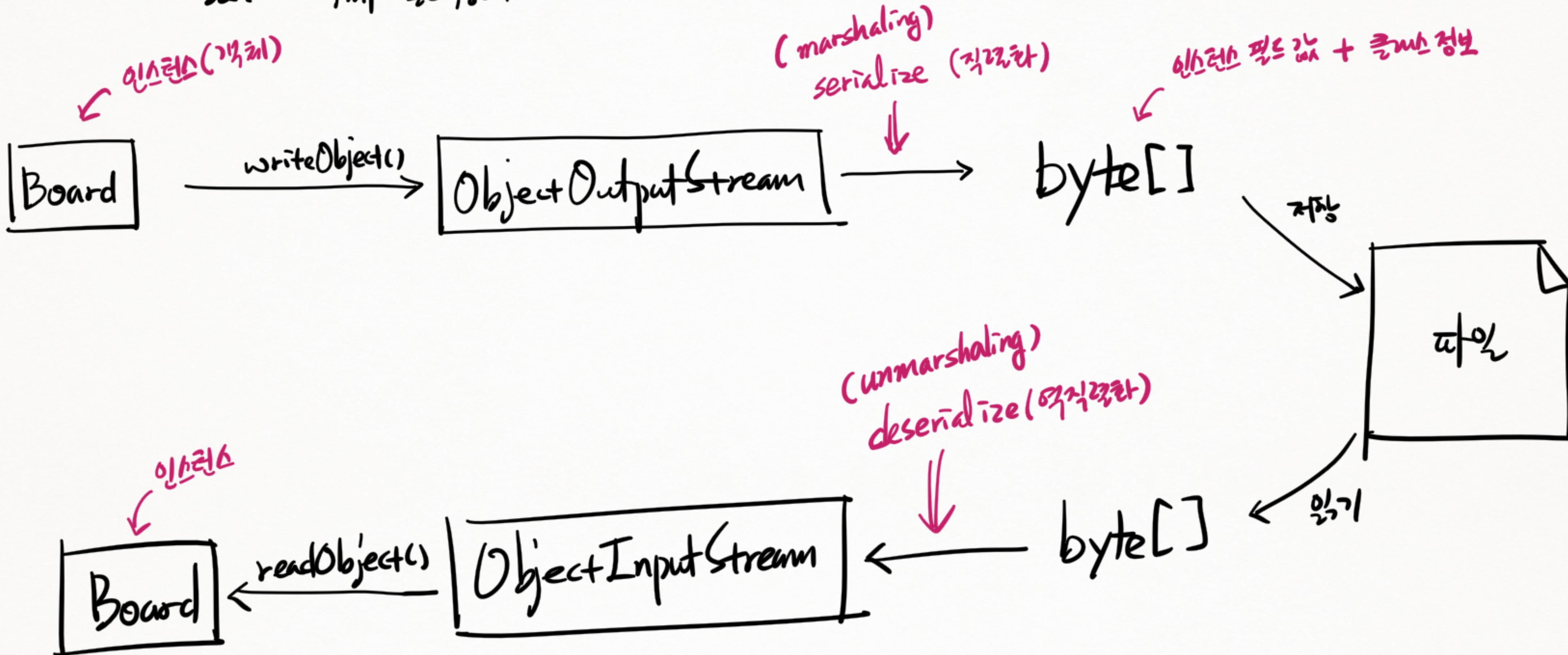
Original



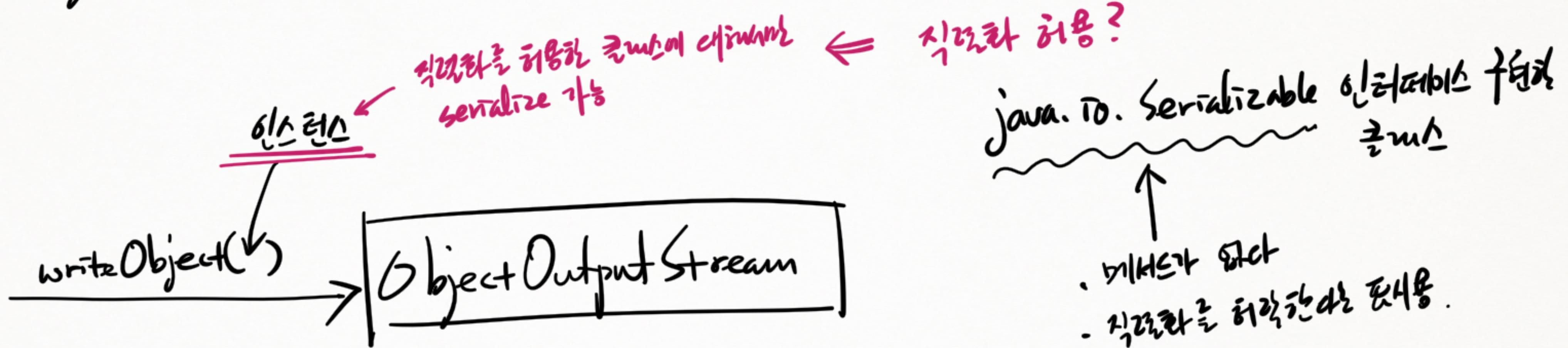
↳ Decorator 패턴 (GOF)
(Composite 패턴과 유사)



32. 객체 흘러보기



* java.io.Serializable 인터페이스

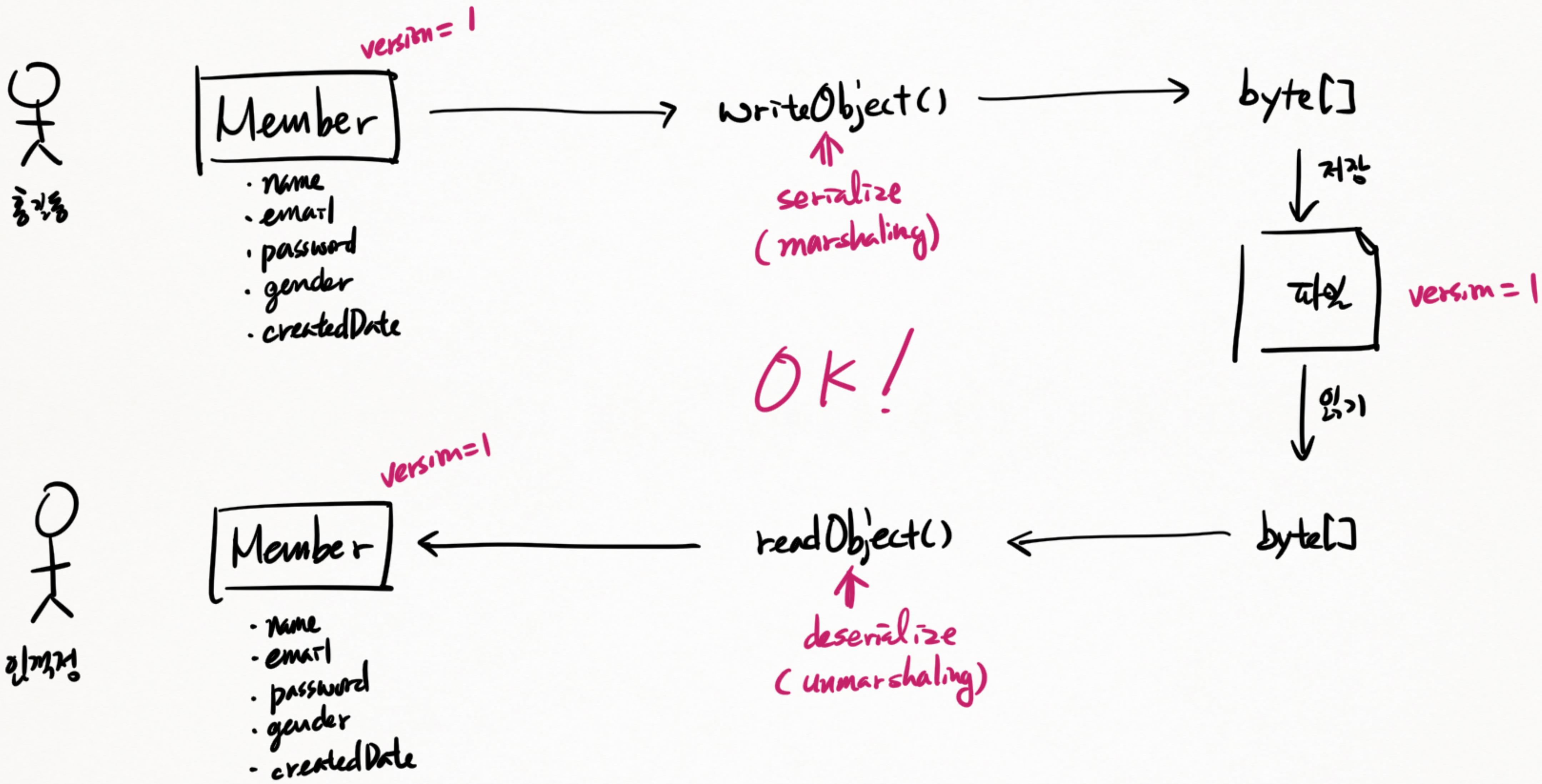


* 직렬화는 허락 받았던가?

↓
Yes

↑
"이전의 내용을 그대로 외부로 저장하는 키
값은 유통하고 있다!"

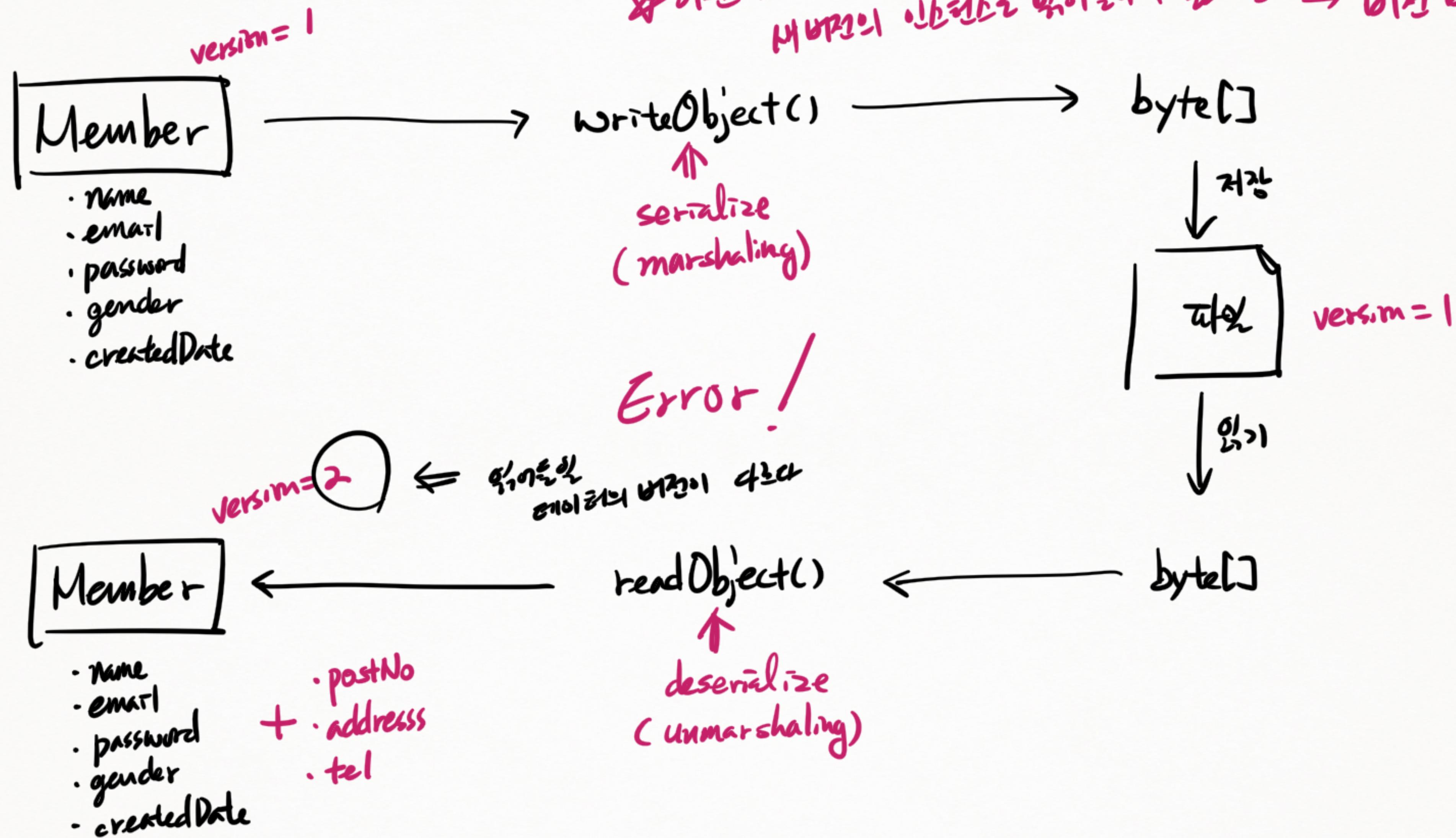
* serialVersionUID 스태틱 필드



* serialVersionUID 스태틱 필드

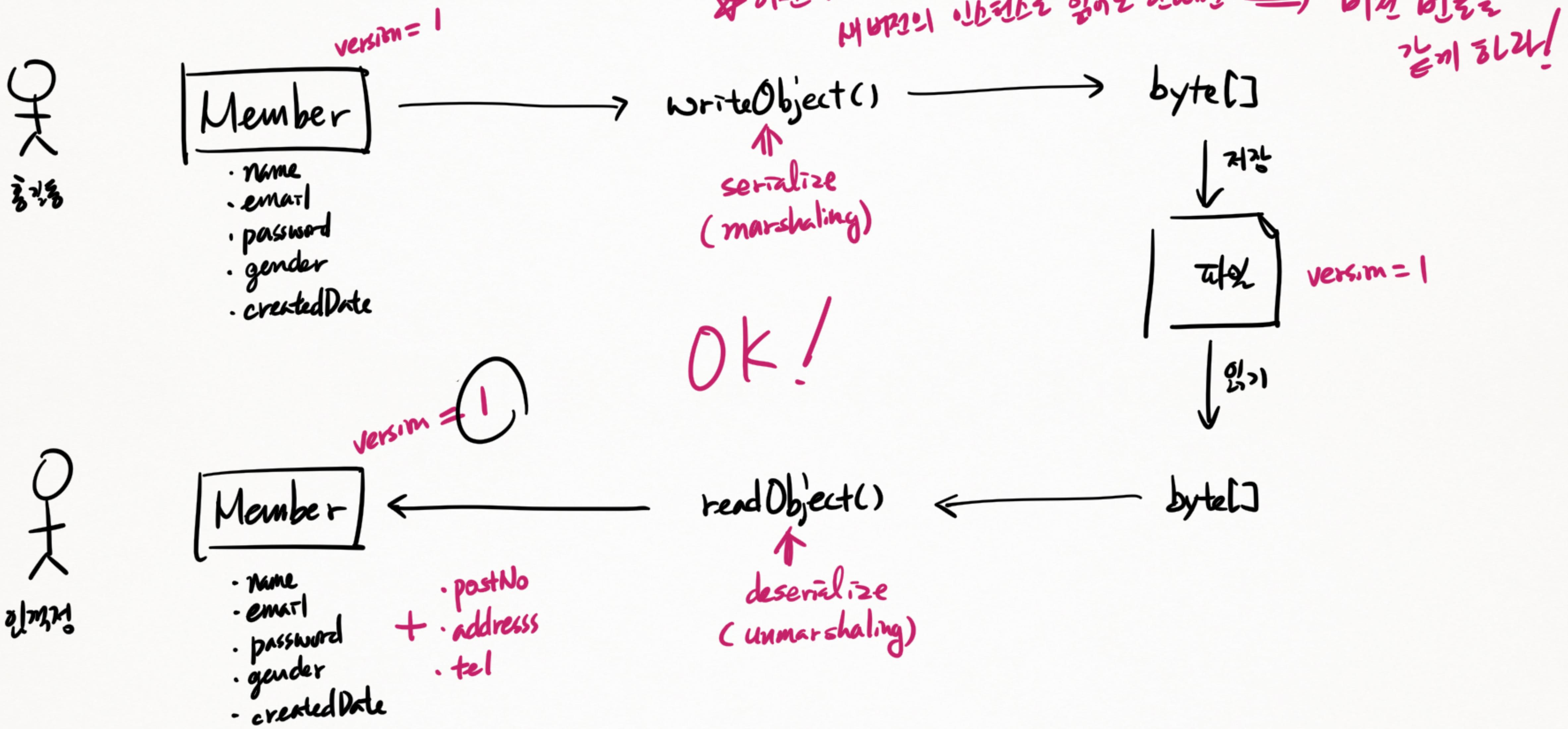
○ 흰색 풀

○ 흰색 풀

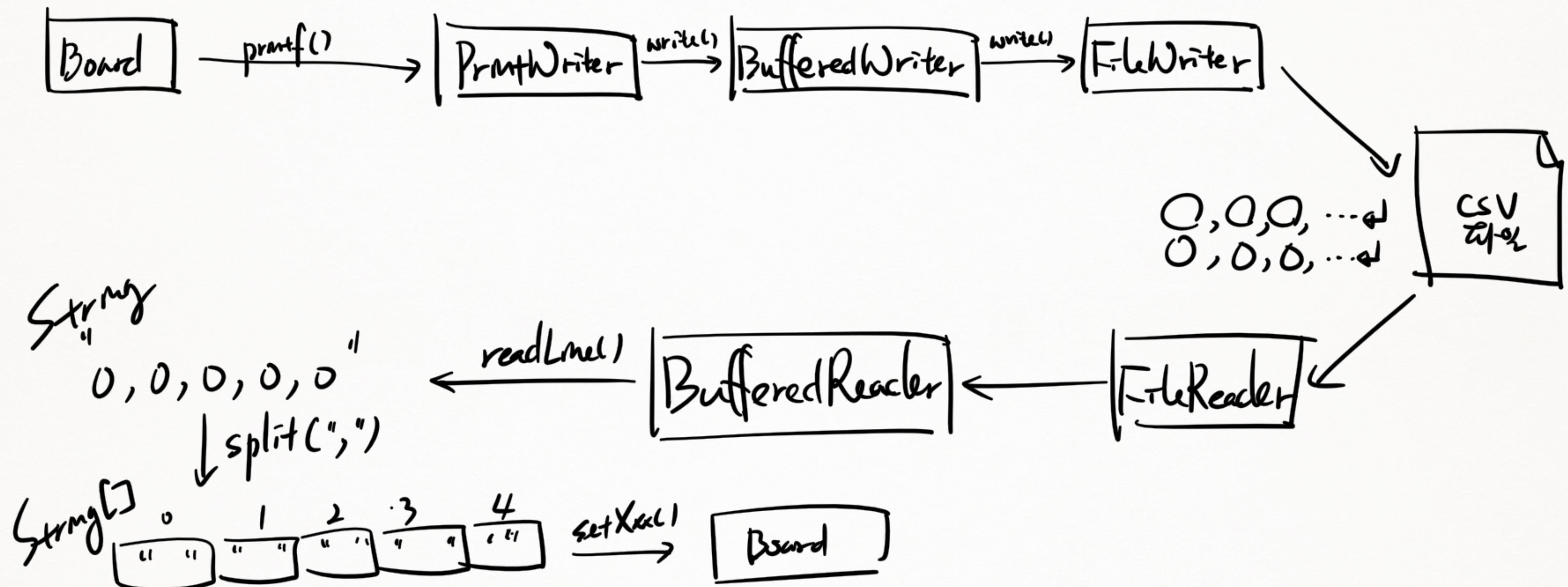


* 이런 버전은 serialize 한 데이터를
내부의 인스턴스로 맞이하기 쉬워 ⇒ 이런 버전은 알리하라!

* serialVersionUID 스태틱 필드

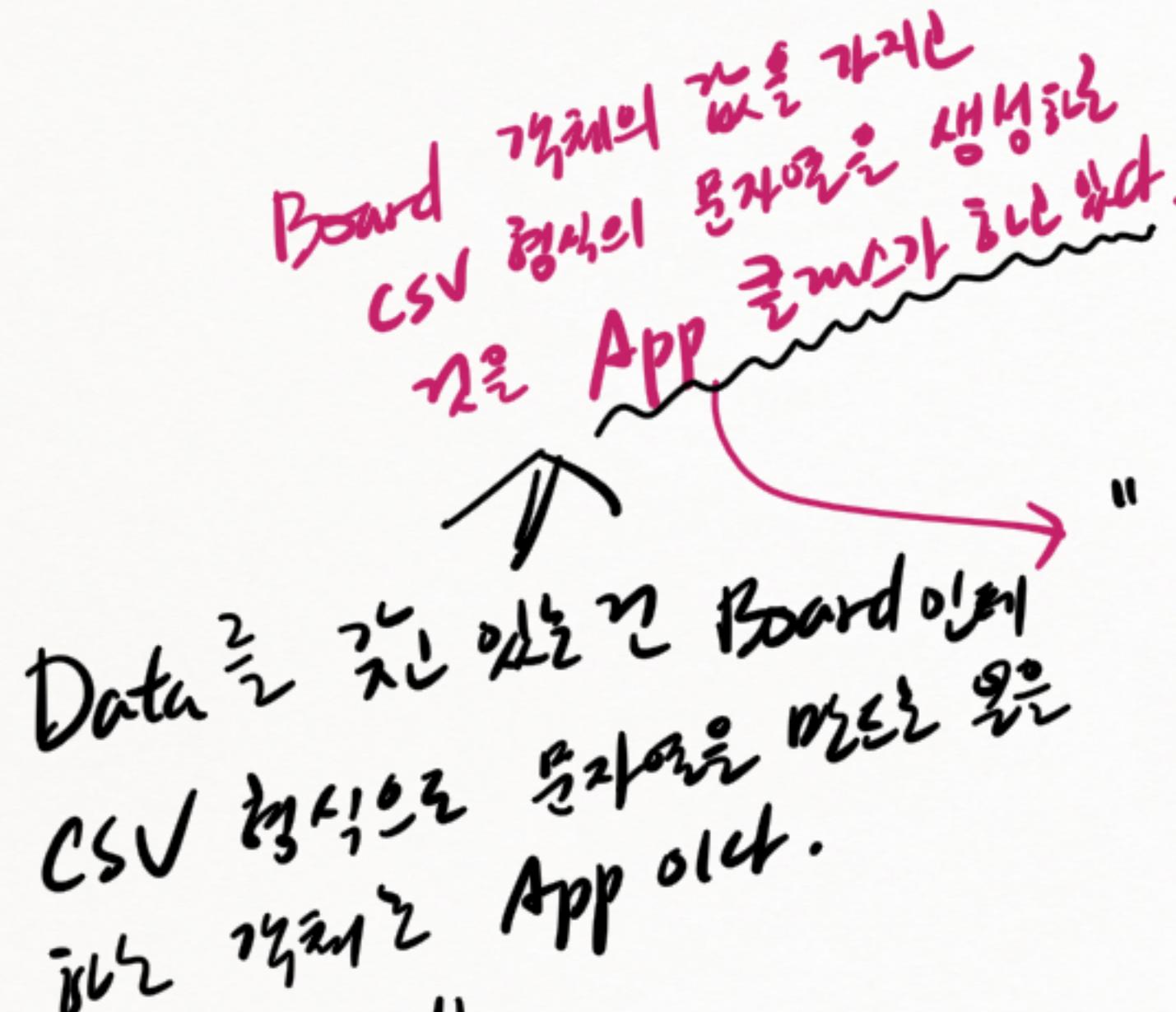


33. 텍스트 파일 (CSV) 을 파일로 읽기

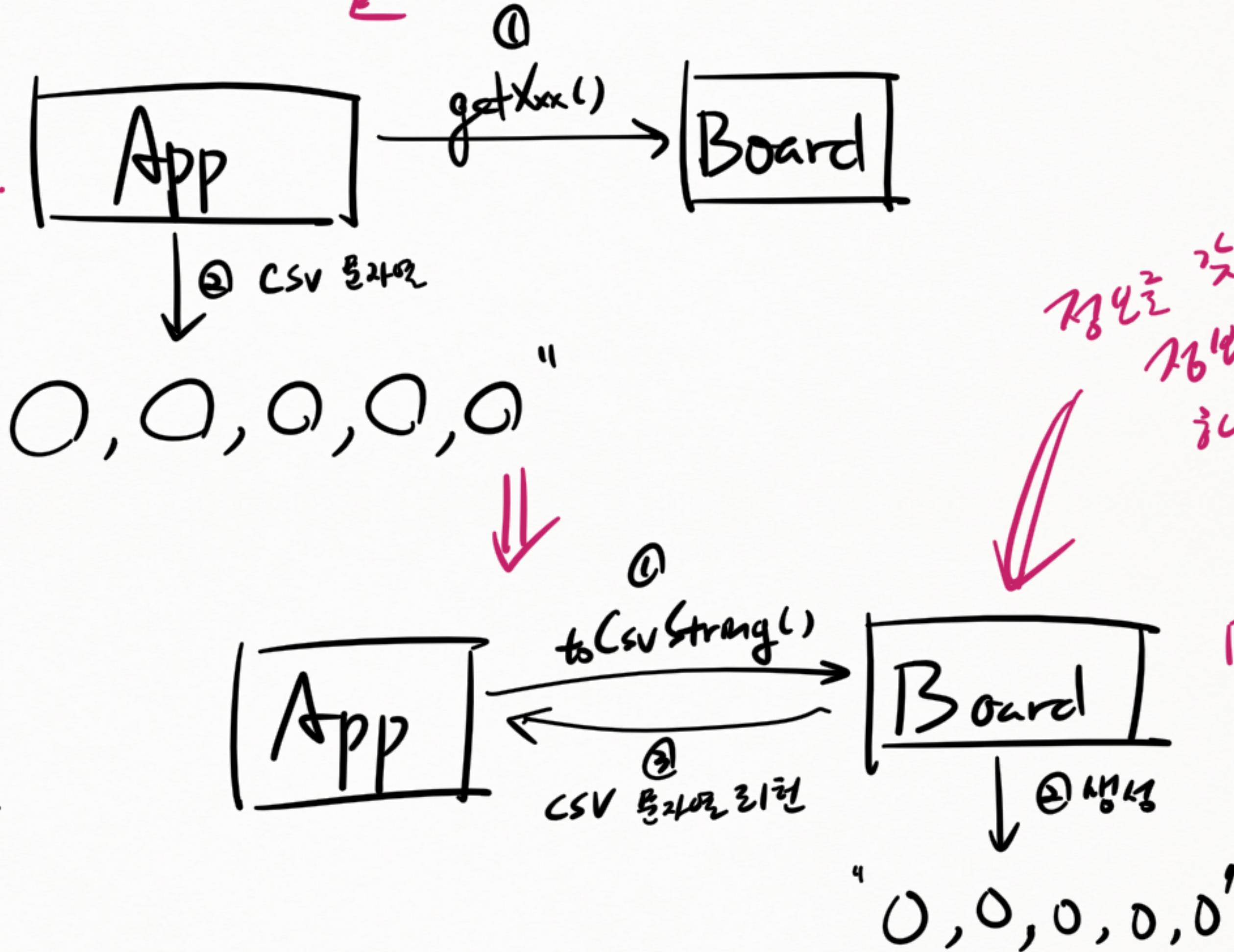


34 · Refactoring : ① Information Expert

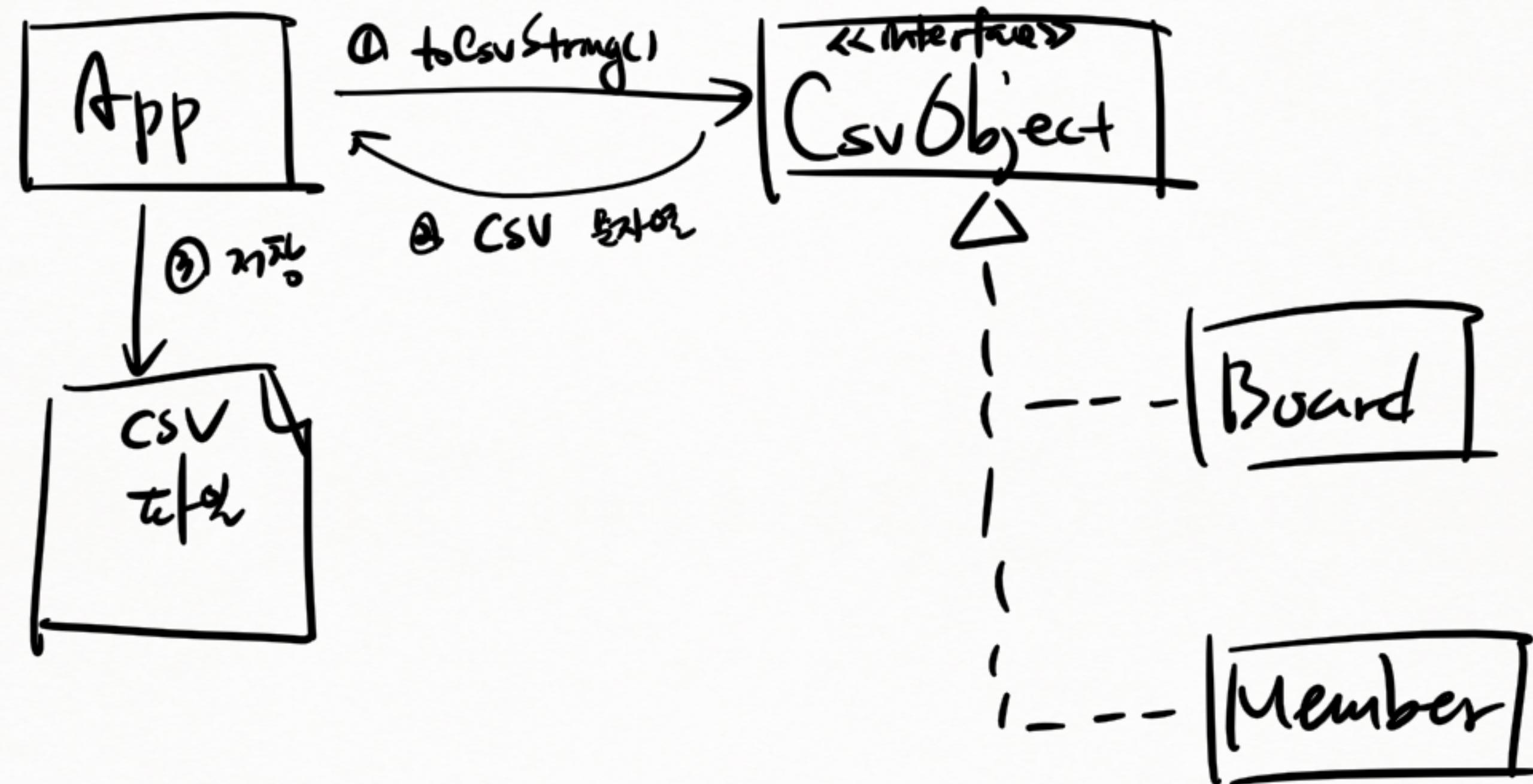
문제점: Board의 필드가 추가되거나
삭제되며 App 클래스를 변경해야 한다



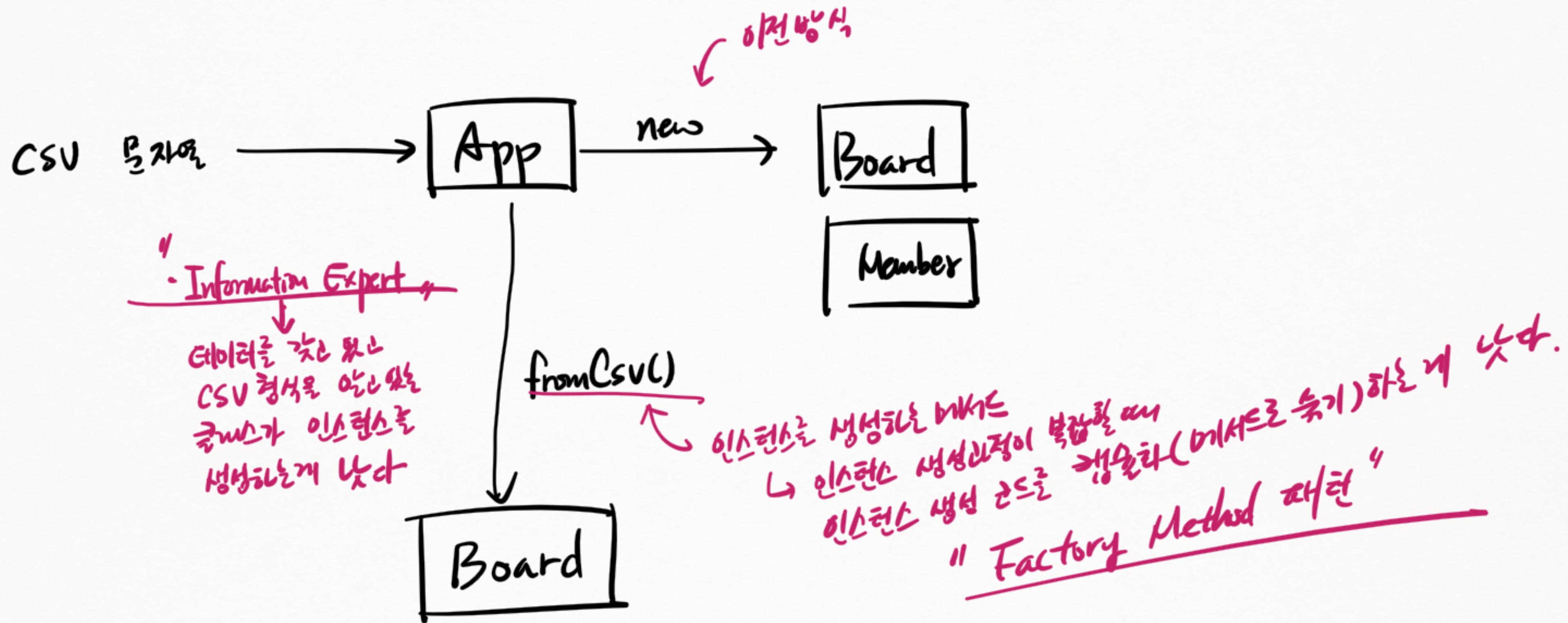
"Information Expert"
자료으로 전문가로 활동



* 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);

↑
매서드 호출

↑
인스턴스 주기
(스레蚀 매서드의 경우, null을 넘긴다.)

↑
매서드를 호출하는 단계를 파악합니다.