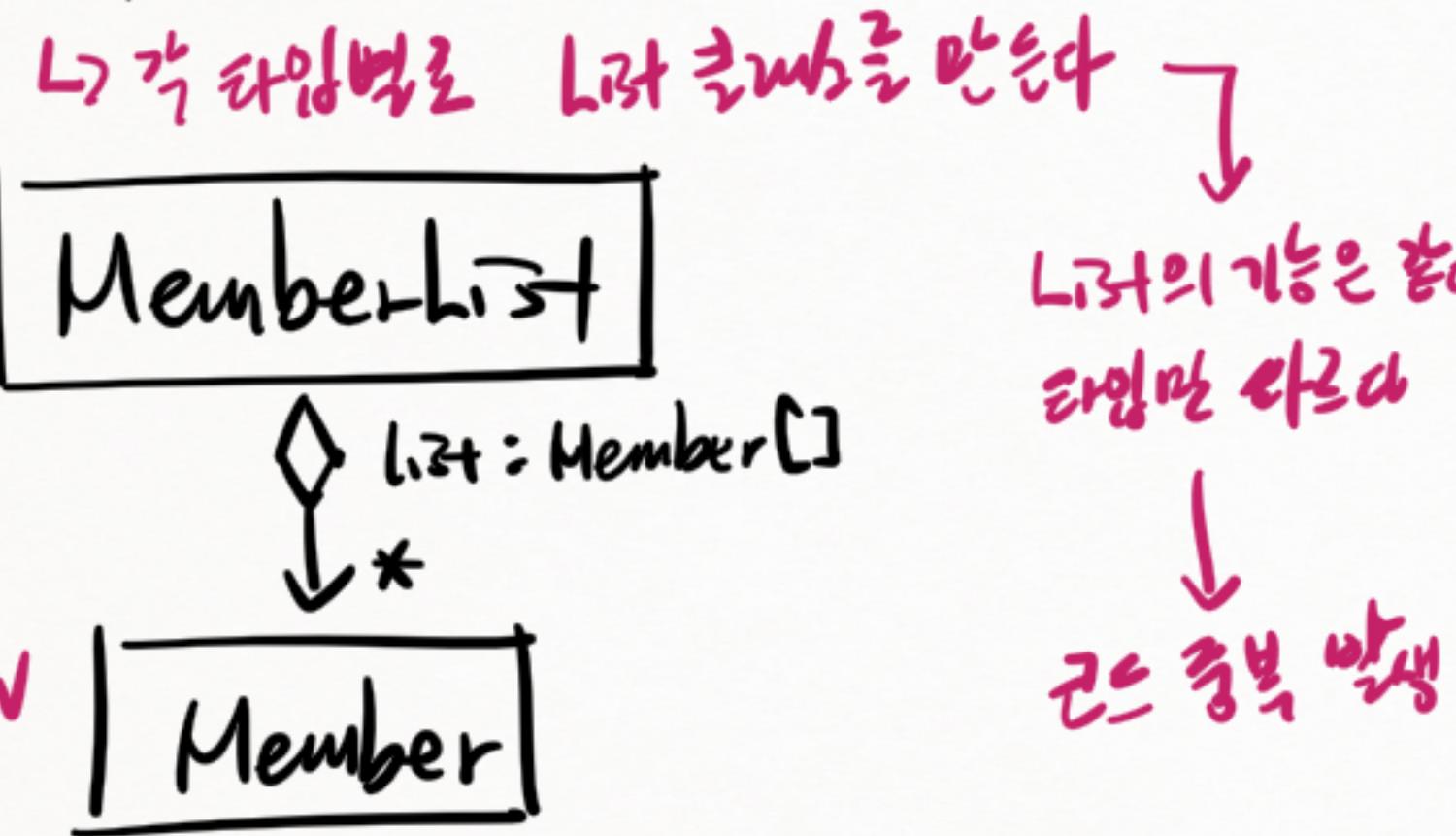


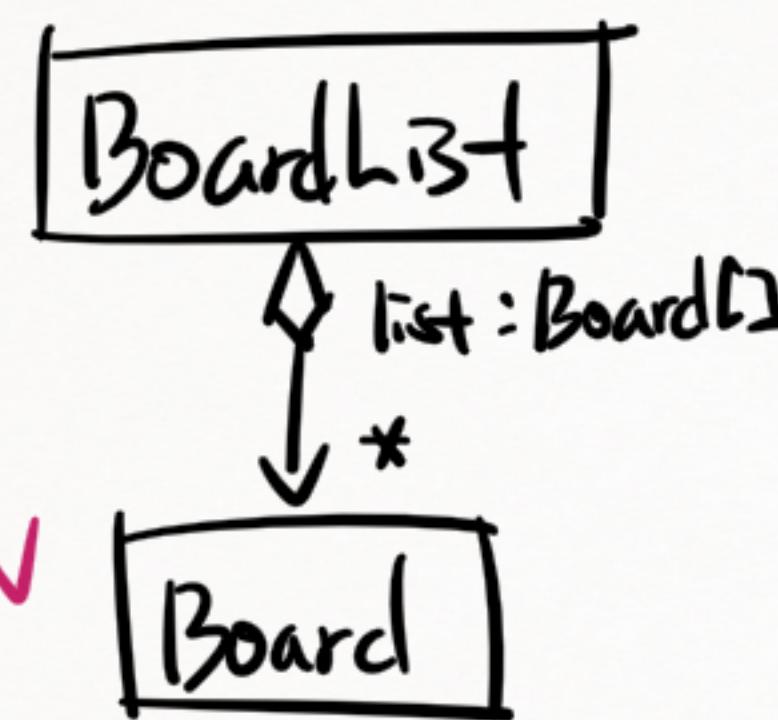
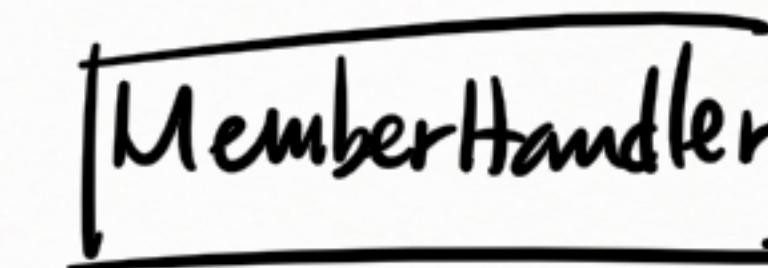
24. 제네리ك (Generic) 적용 : (object 타입처럼 다양한 타입에 대응할 수 있다.)
특정타입으로 제한할 수 있다.

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

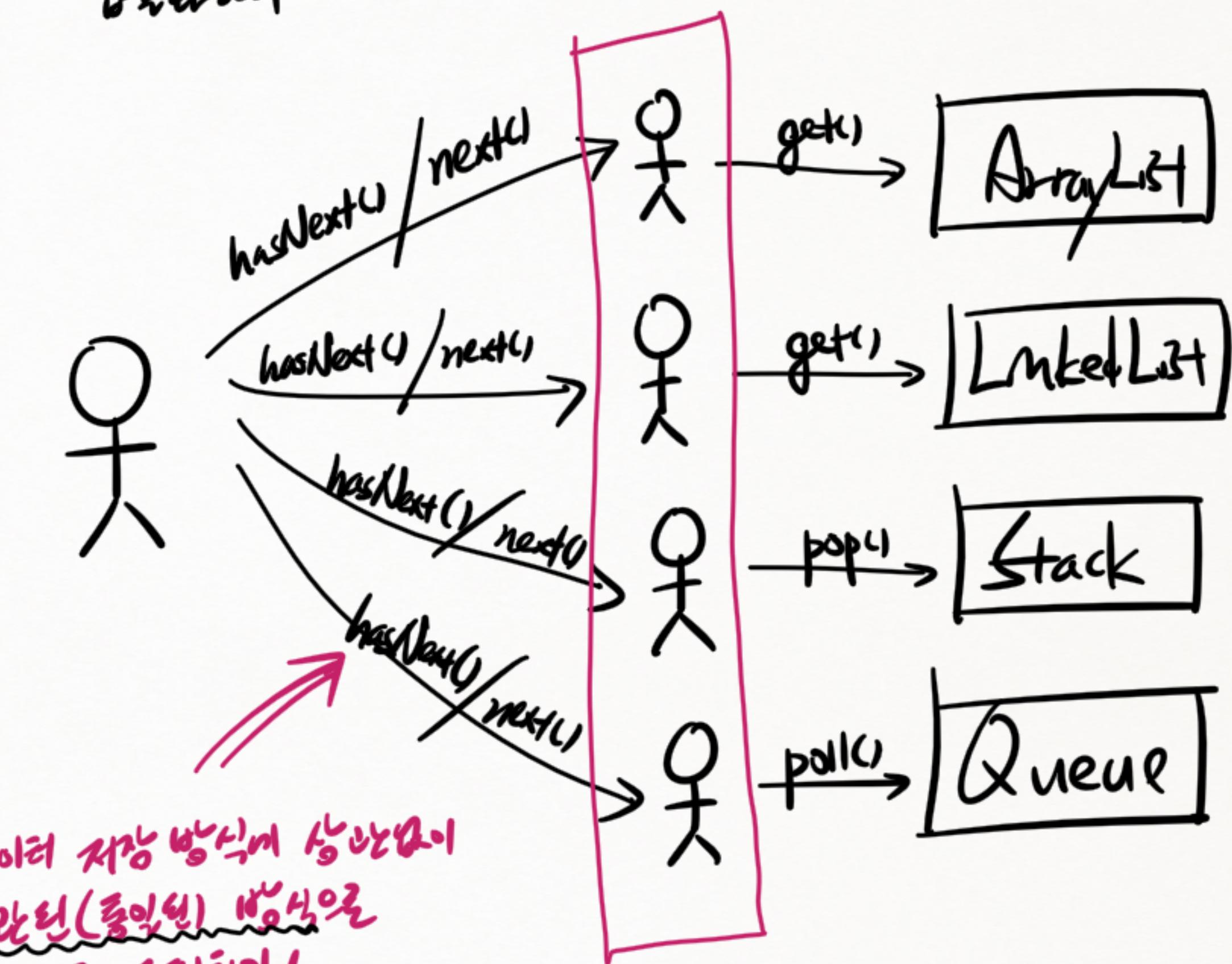
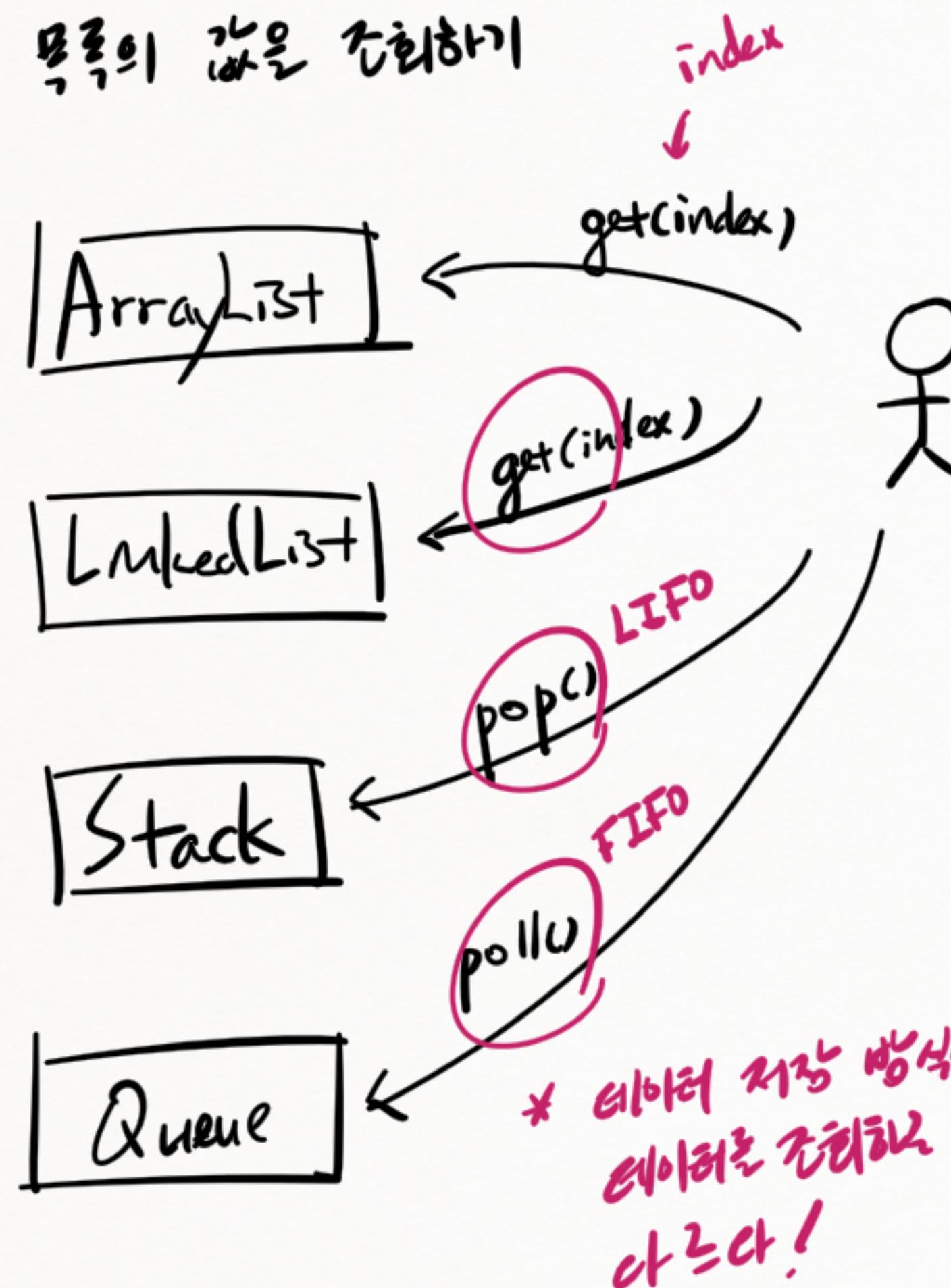
클래스의 타입 변수를 넣는다

```
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

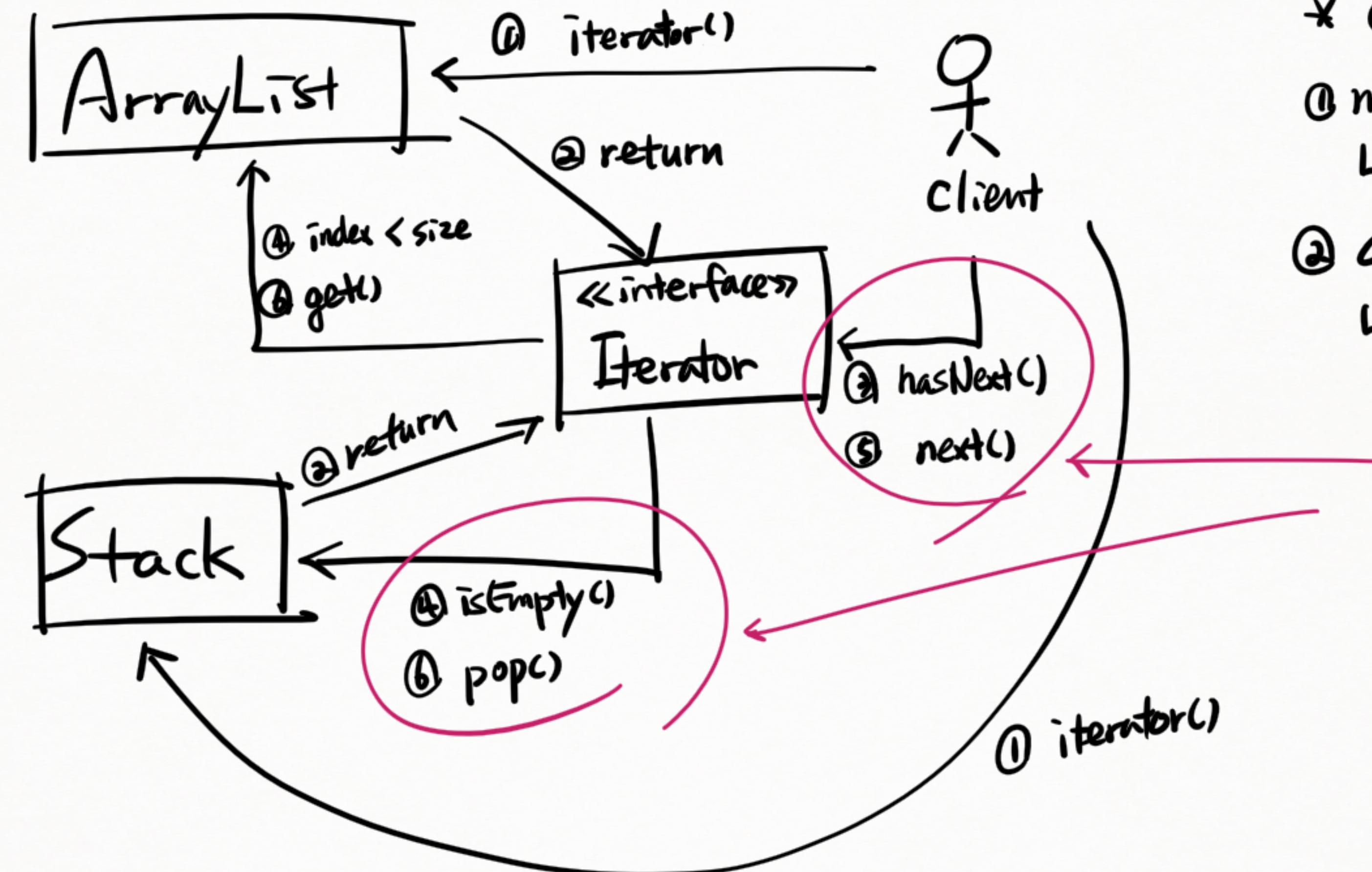
① 목록의 값을 조회하기



* 데이터 처리 방식에 따라
실란트(중복된) 방식으로
데이터를 조회하기!

Iterator
(데이터를 순회하는 일을 하는 객체)

* Iterator mechanism (구조원리)

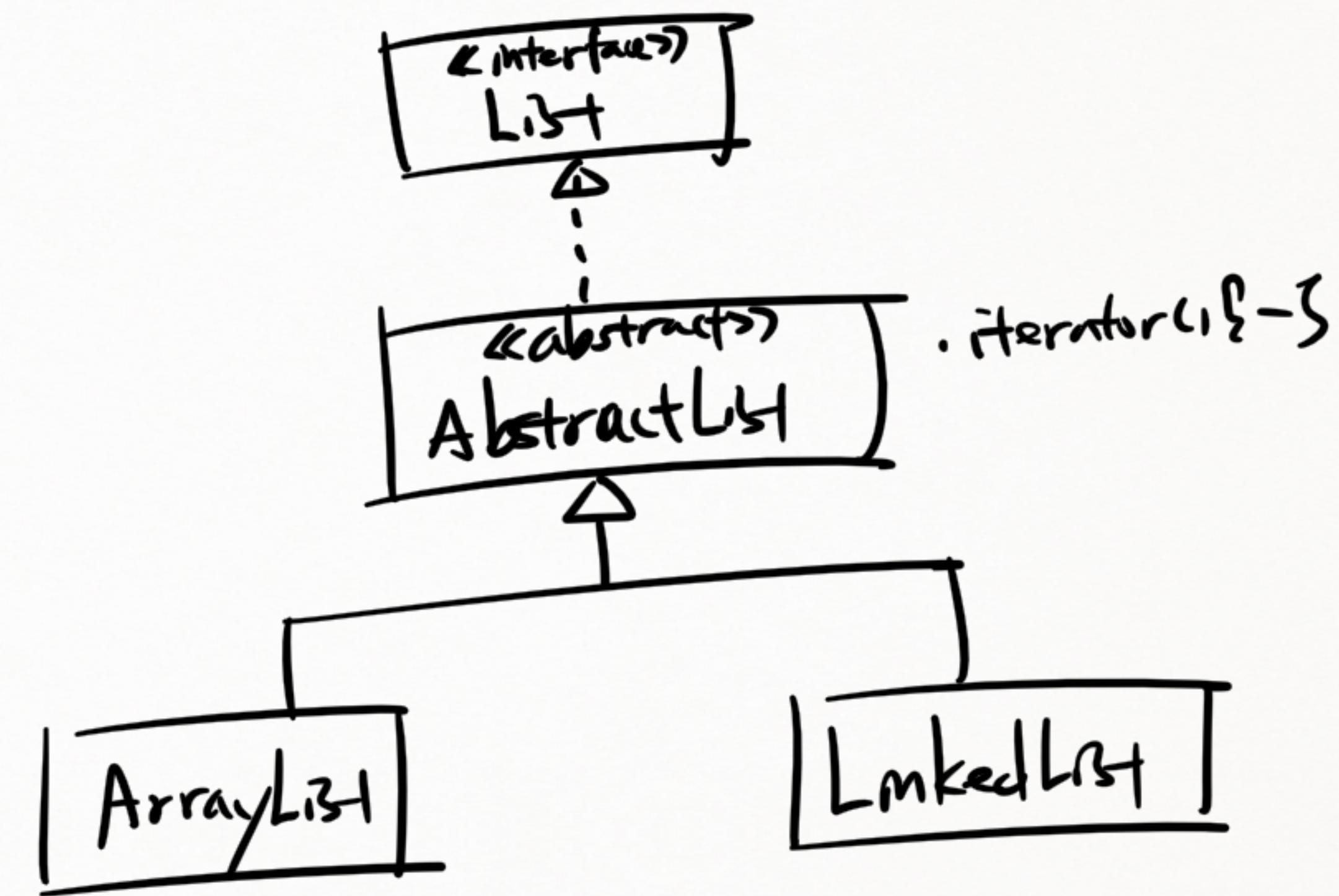
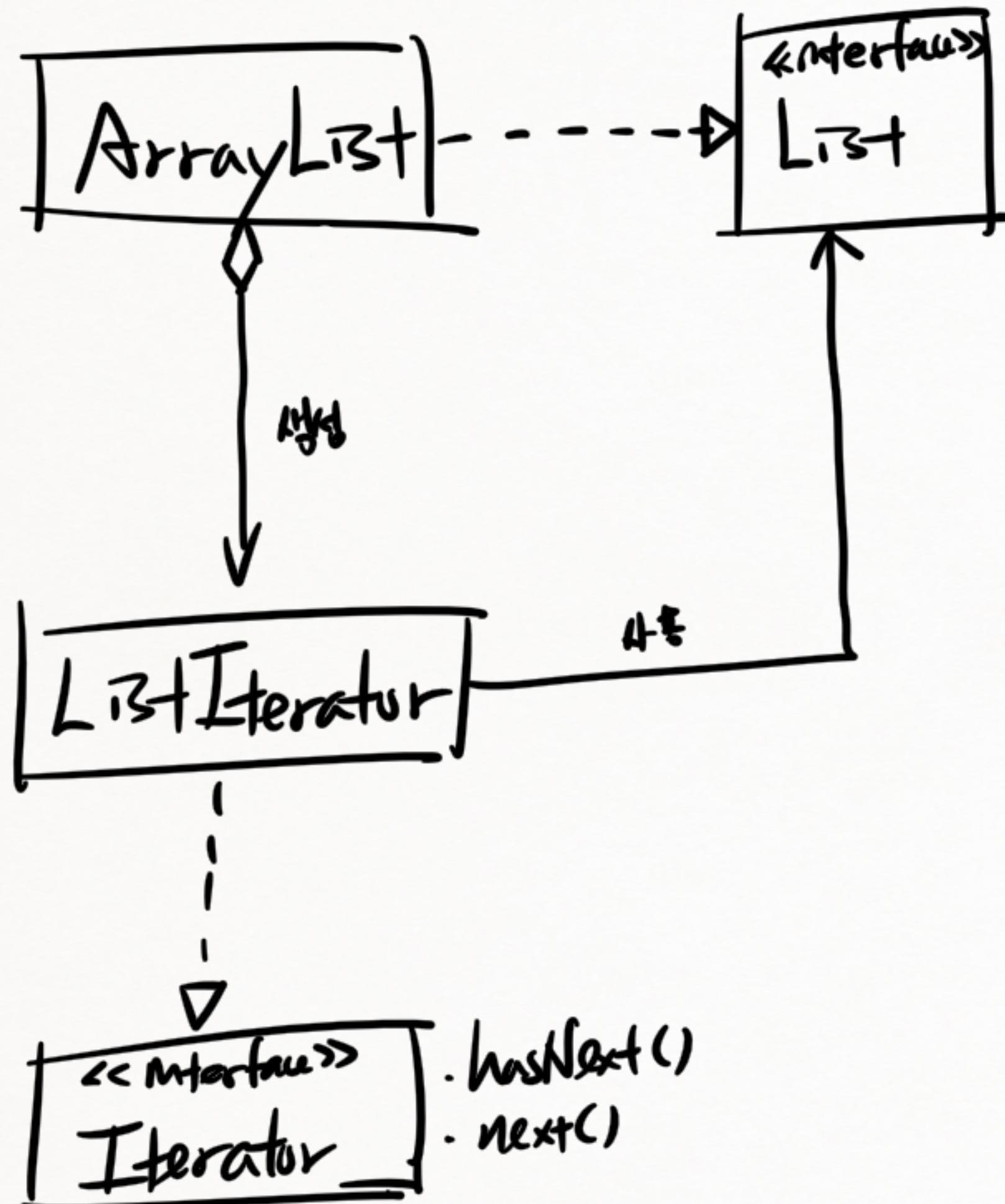


- * client
- ① network 분야
↳ 네트워크 요청으로 연결을 수행 S/W
- ② OOP 분야
↳ 다른 객체를 사용하는 개체.

제작업체가 상관없이
일관된 방식으로
작동을 가능케 했다!

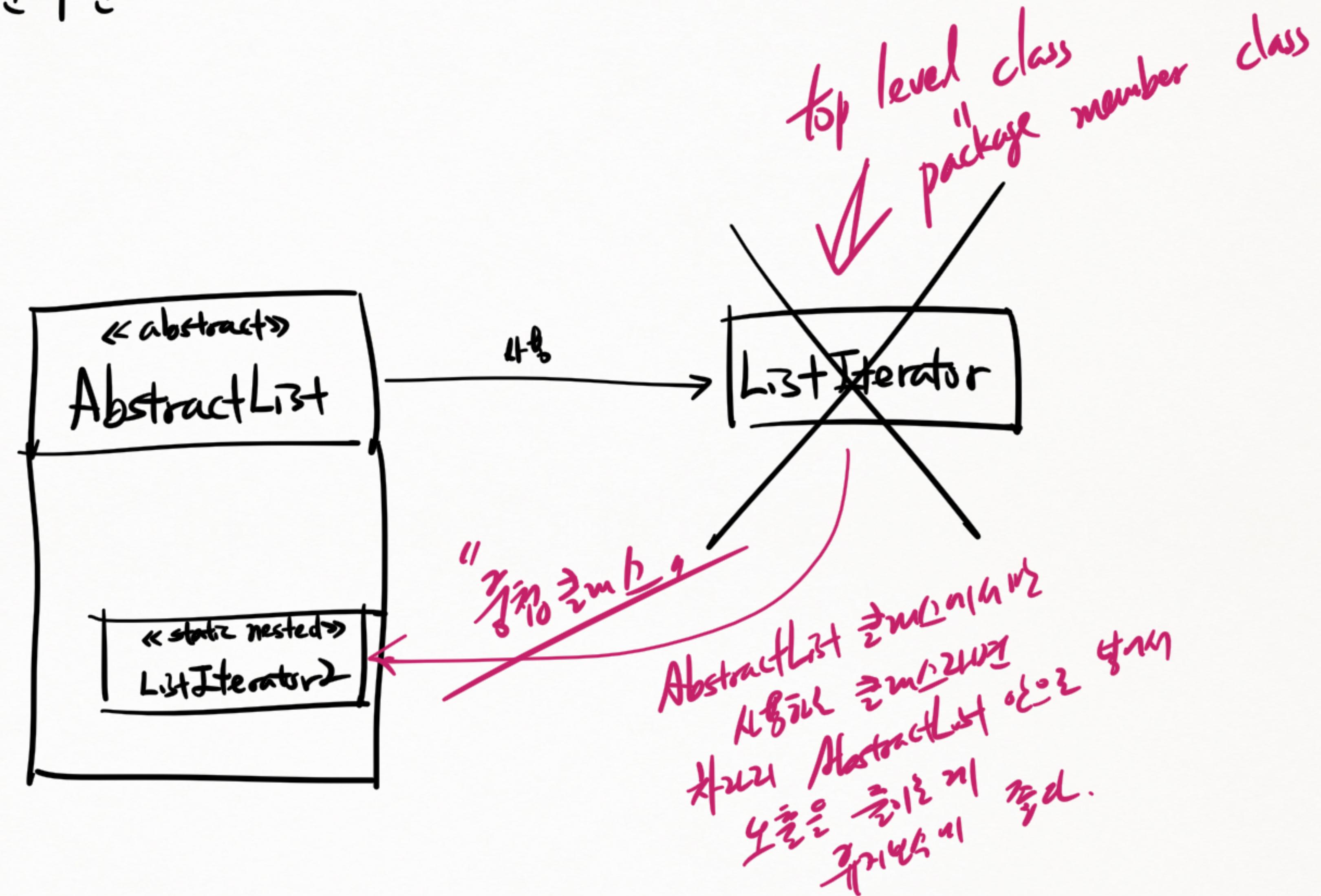
* Iterator 퀘션 구조

① 퀘션 - top level class



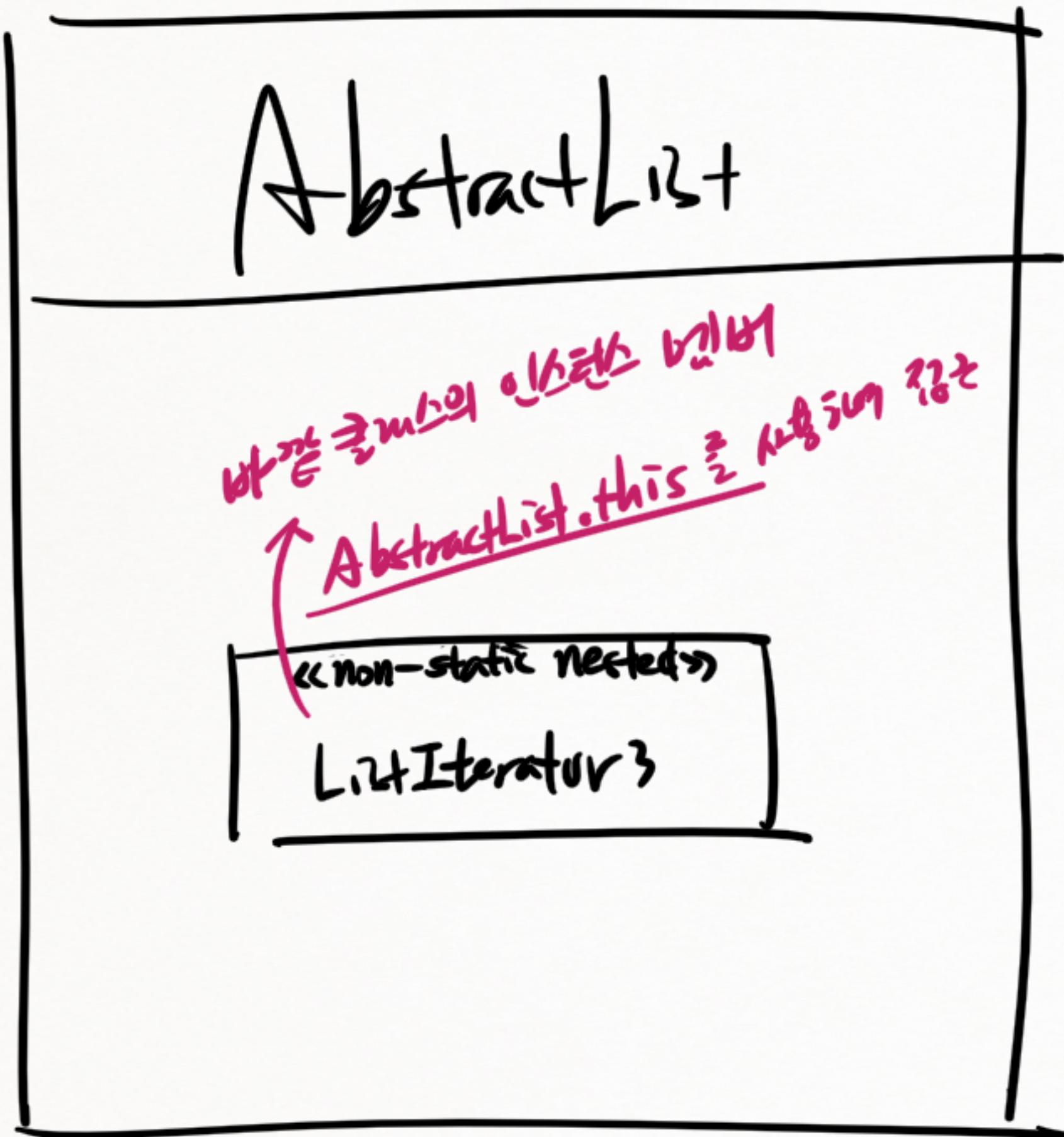
* Iterator 퀘션 구현

② 구현 — static nested class

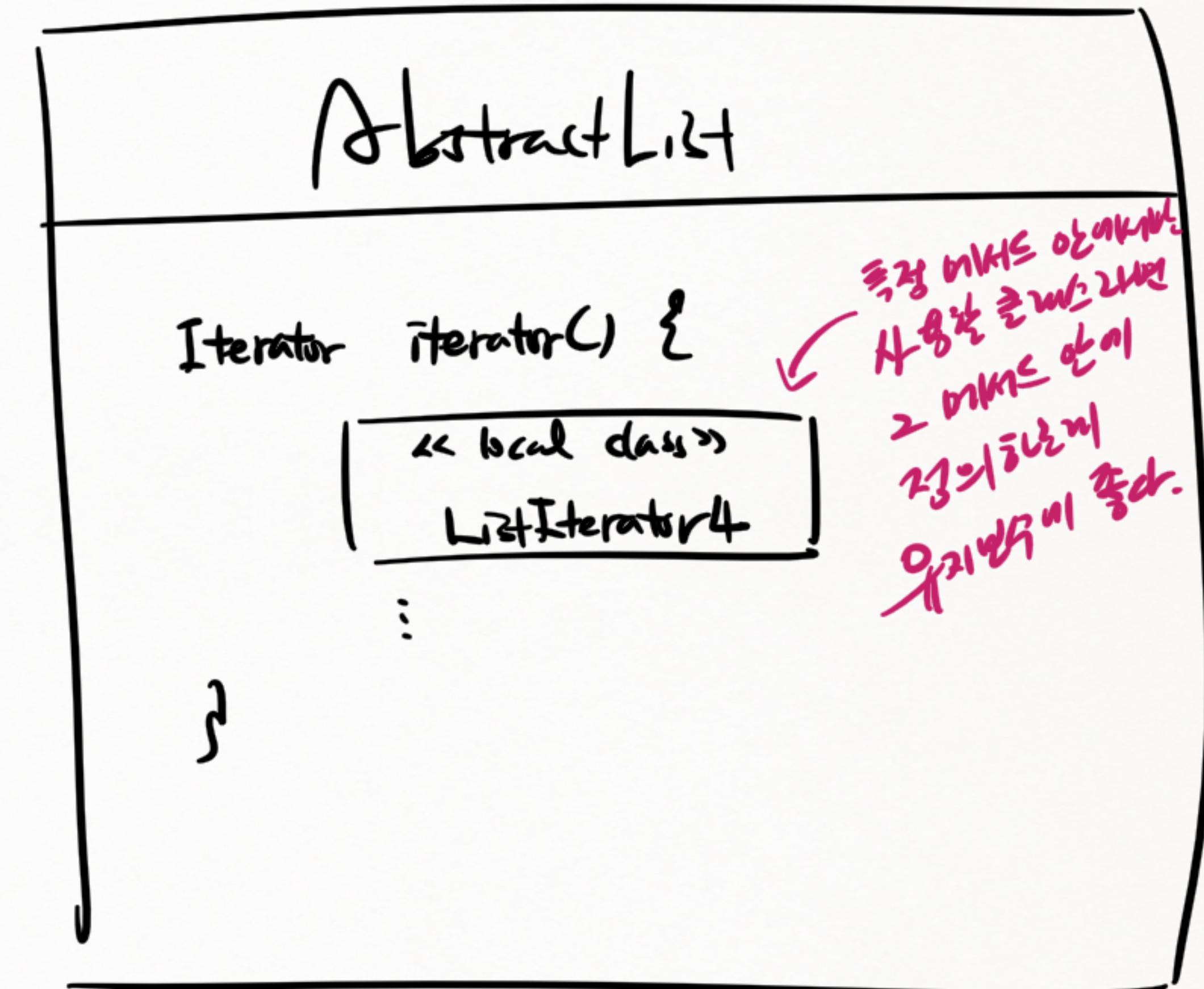


* Iterator 파편 구현

③ 구현 - non-static nested class

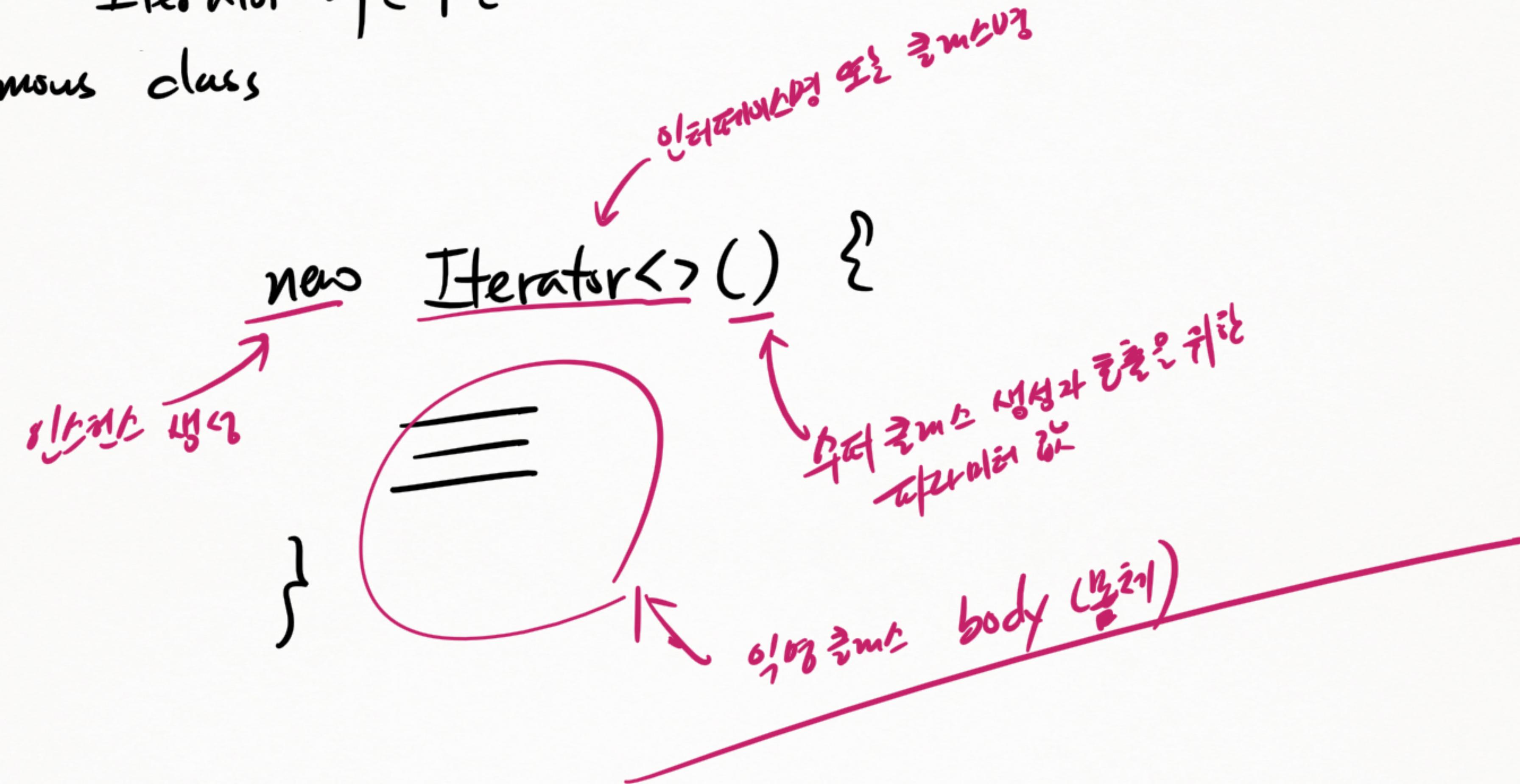


④ 구현 - local class



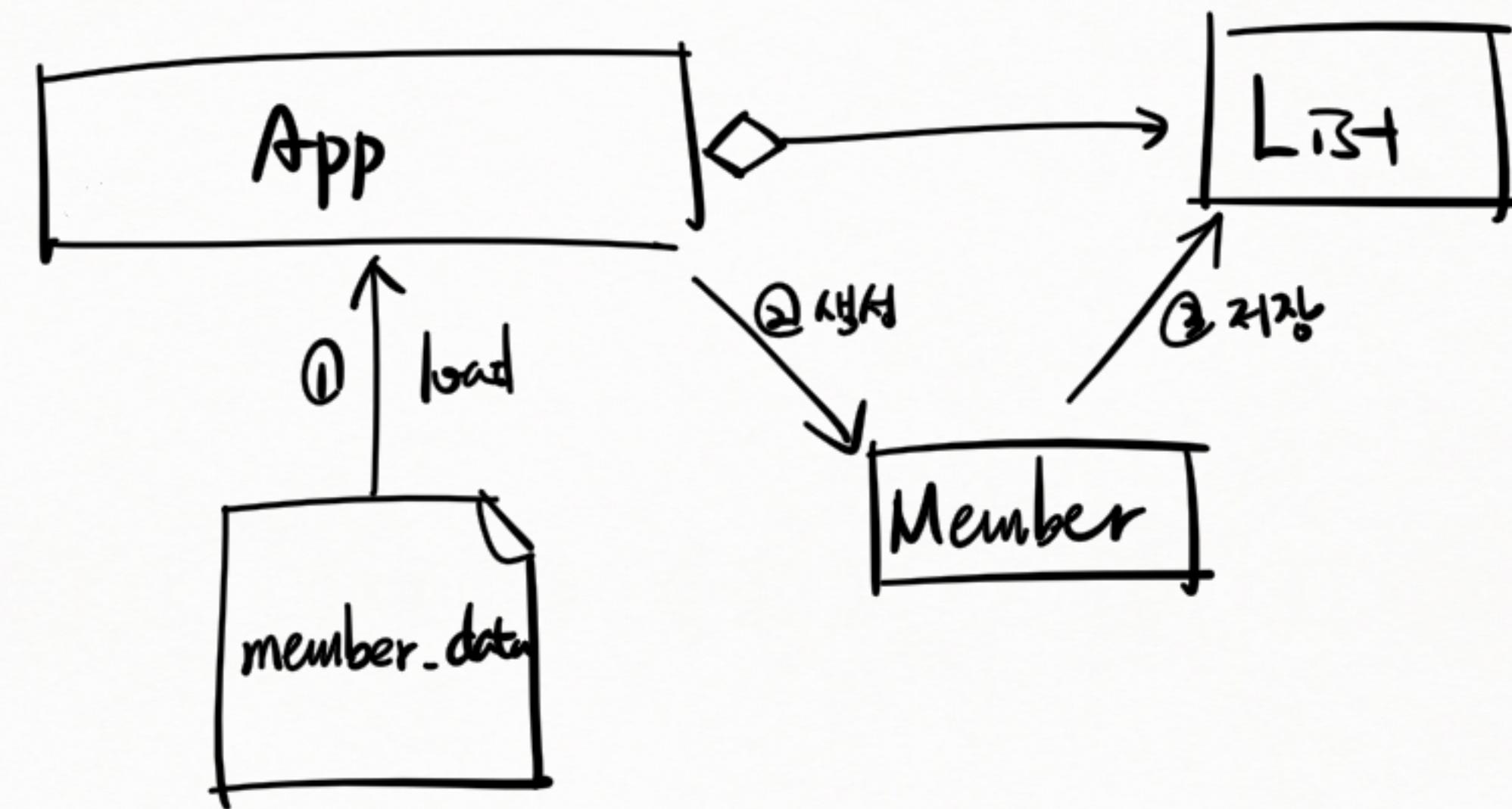
* Iterator 파편 구현

⑤ anonymous class



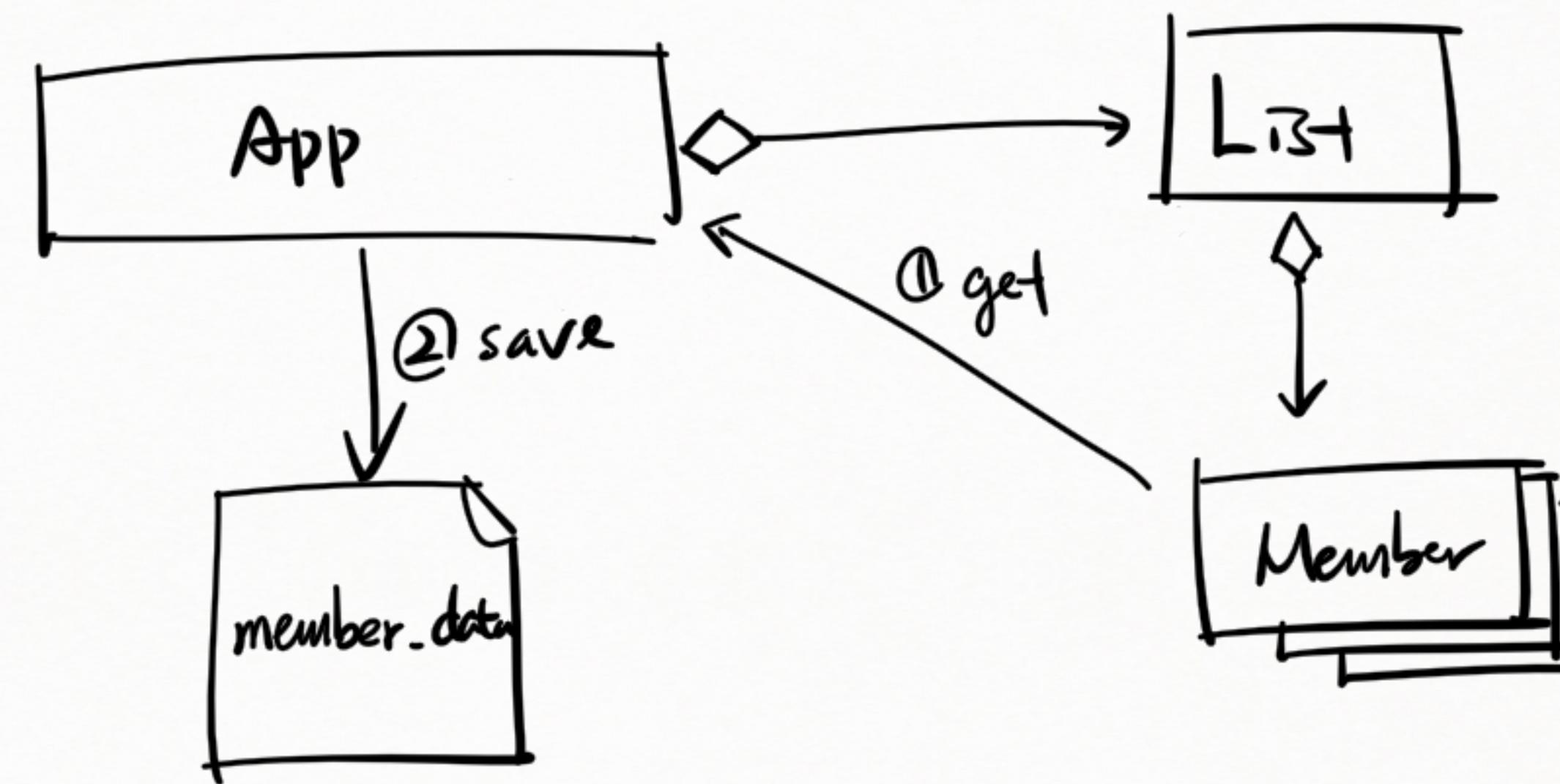
27. Data I/O Stream API - binary stream API 사용 ↳ Data 저장

① Data 읽기



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

② Data 쓰기



```
int length;
```

100|00|00|03

00|03

in.read(4)

...|00|00|00| << 8

...|00|00|00|

in.read()

...|00|03|

: ...|00|00|00|
...|00|03|
00|00|00|03|

61|61|61|

in.read()

12345678910 length

↓
↓

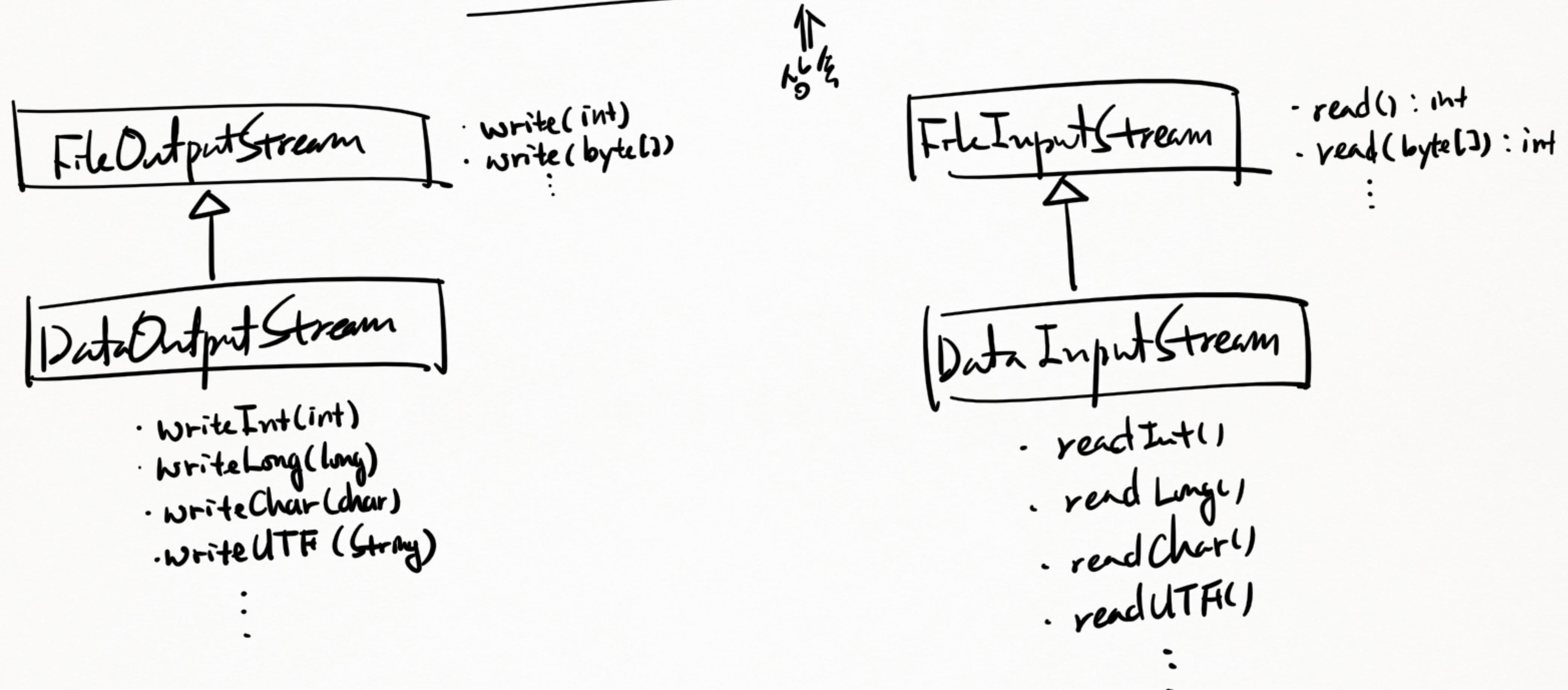
in.read(buf, 0, 3)

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

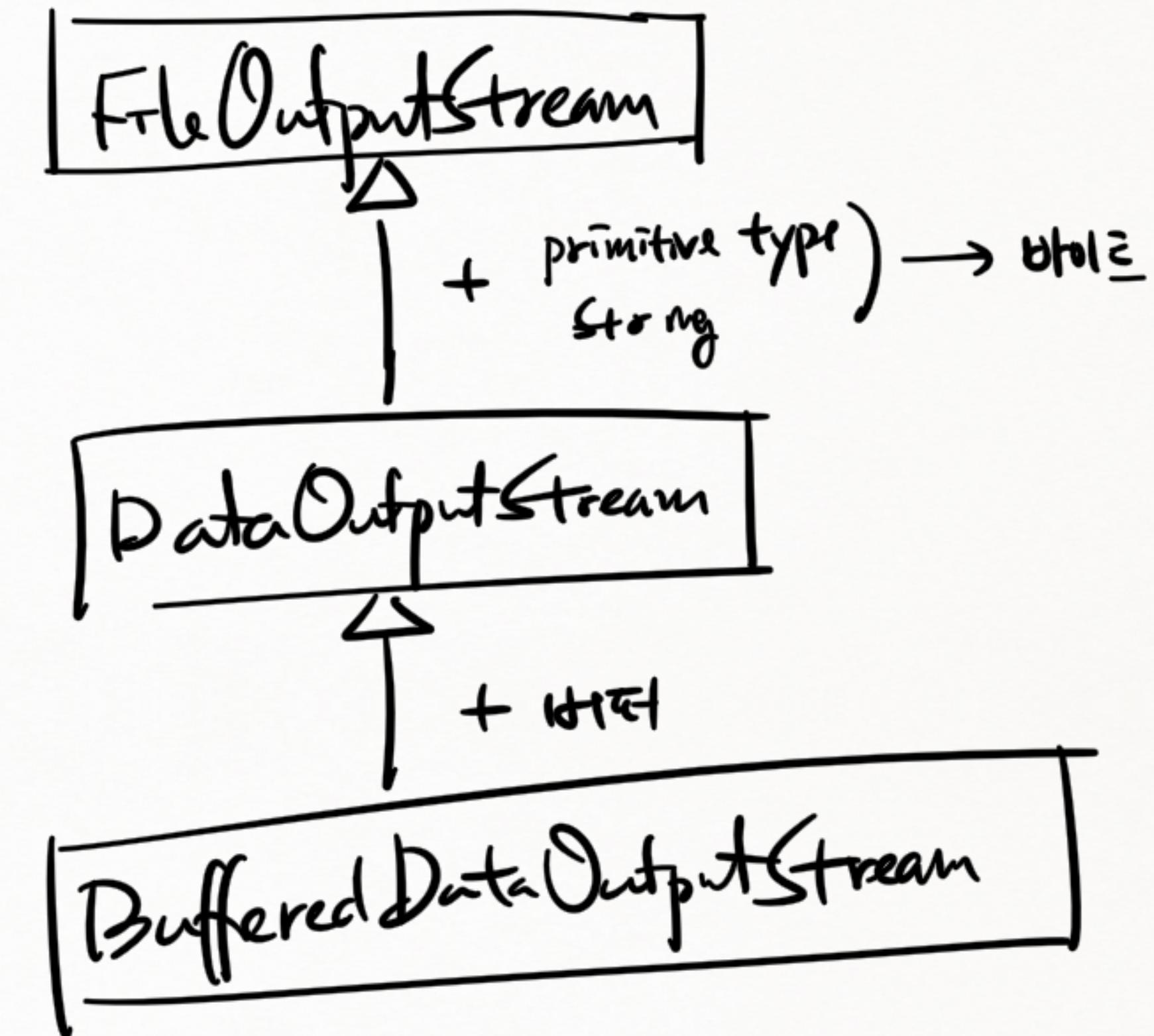
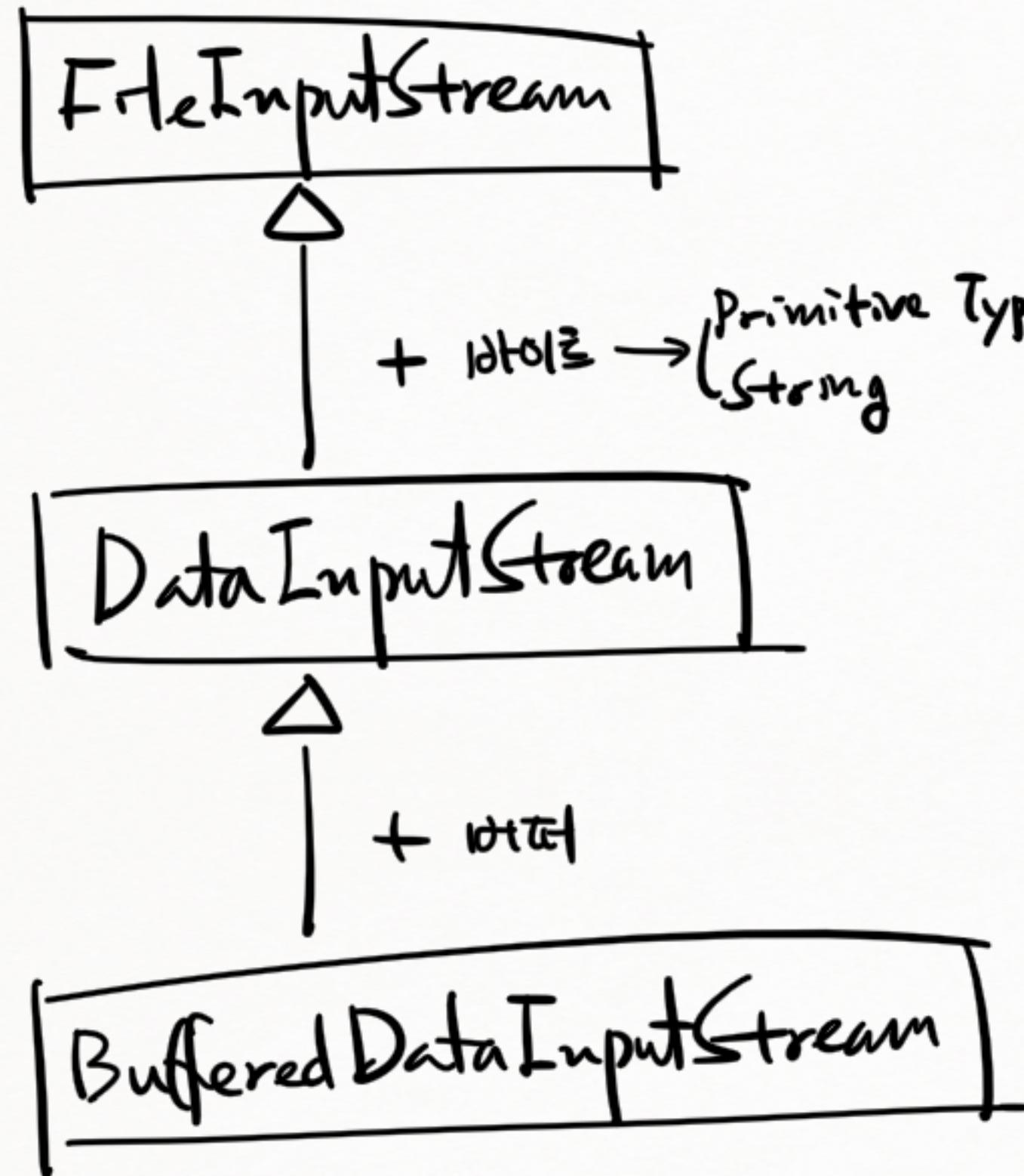
new String(, 0, count, "UTF-8")

member - setName()

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



29. 파일 입출력의 버퍼링 예제 : 파일 읽기 쓰기

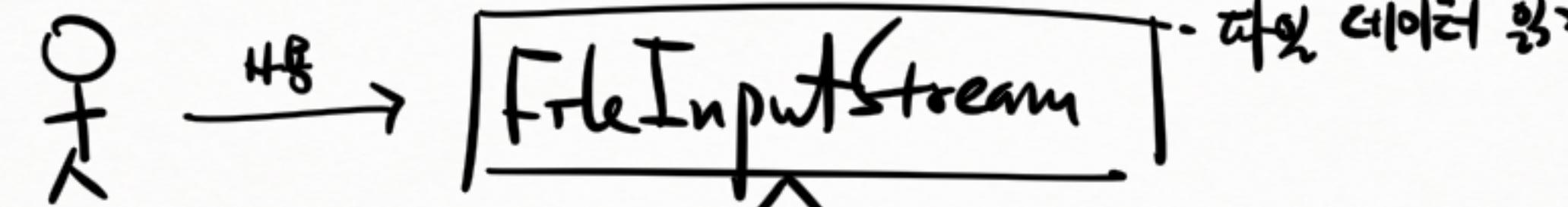


30. 기능을 확장할 때 상속과는 Decorator 패턴 적용

① 상속을 이용한 기능 확장의 문제점

- byte / byte[] 읽기

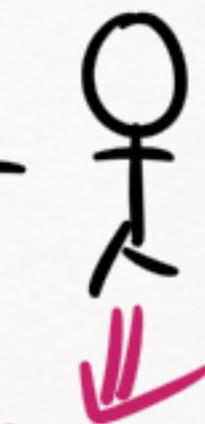
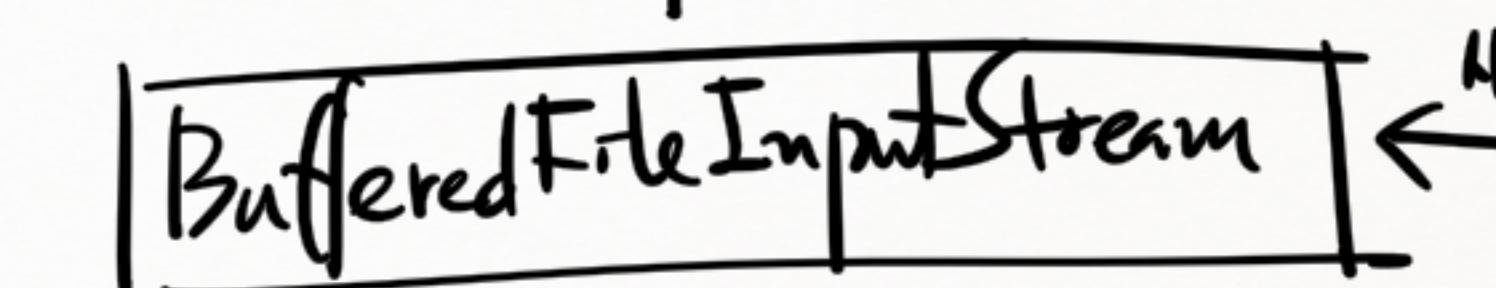
- * Primitive Type / String 읽기 불편



- (primitive type) 읽기 편함
String

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

- 버퍼를 이용해서 읽기 성능 개선

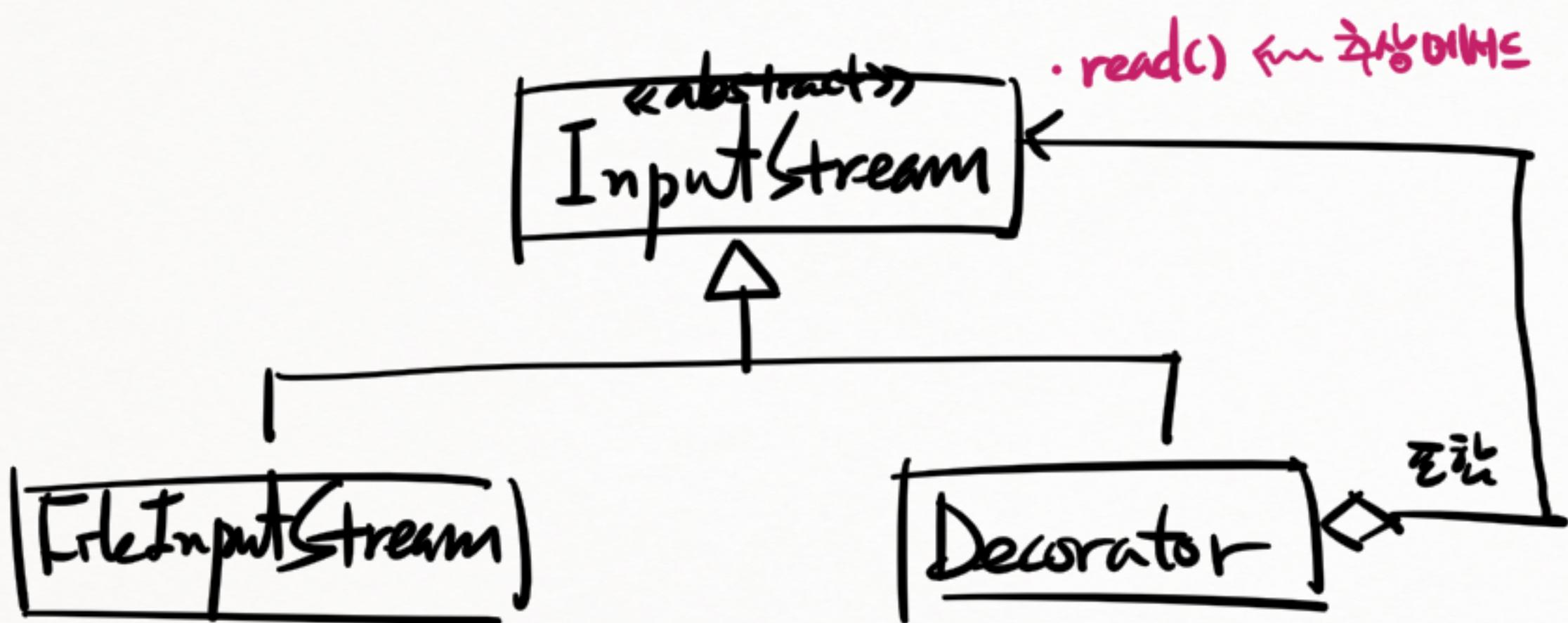


* 상속은 좋은 유도의 기능은
제한된 한계.
그러나 상속의 sub 클래스는 오버라이드 할 때
부모의 기능은 재정의 해야 한다!
↳ 이것이 상속의 한계!

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

30. 기능을 확장할 때 사용되는 Decorator 패턴 적용

② 장식 패턴 (decorator) 차별 → 이는 핵심이 예기 원하지 않은 결과를 얻게



→ Decorator 패턴 (GoF)
(Composite 패턴의 유형)

