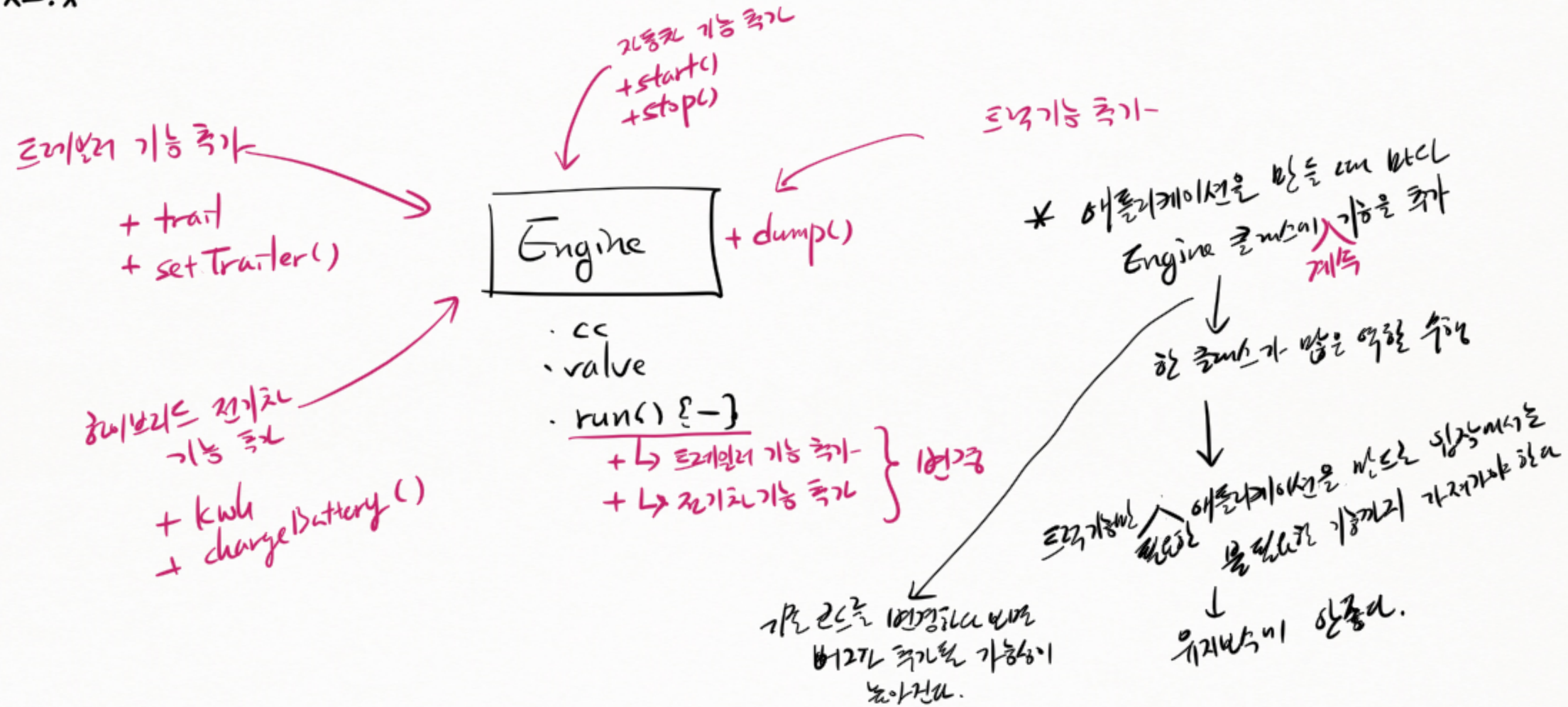
* 기능 확장 방법 1 - 기존 코드에 새 기능 추가

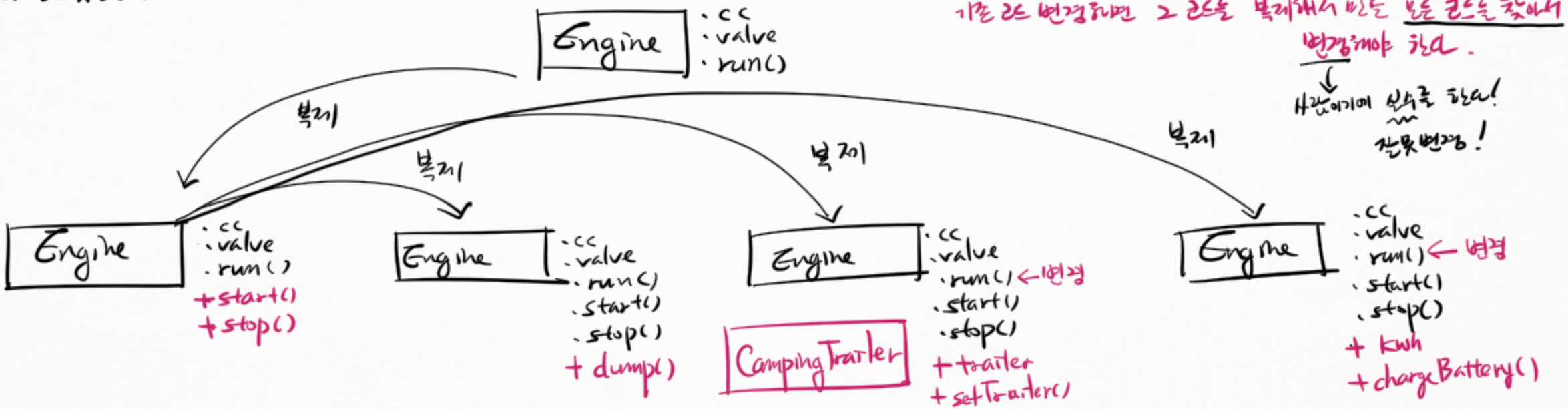
oop.ex05.x2.*



* 기능 확장 방법2 - 복제한 코드에 새 기능 추가

oop.ex05.x3.*

↓ 문제점
기존 코드 변경하면 그 코드를 복제해서 만든 모든 코드를 찾아서 변경해야 한다.
↓
새로운기에 신속히 한다!
잘못변경!



① 자동차 만들기
app1

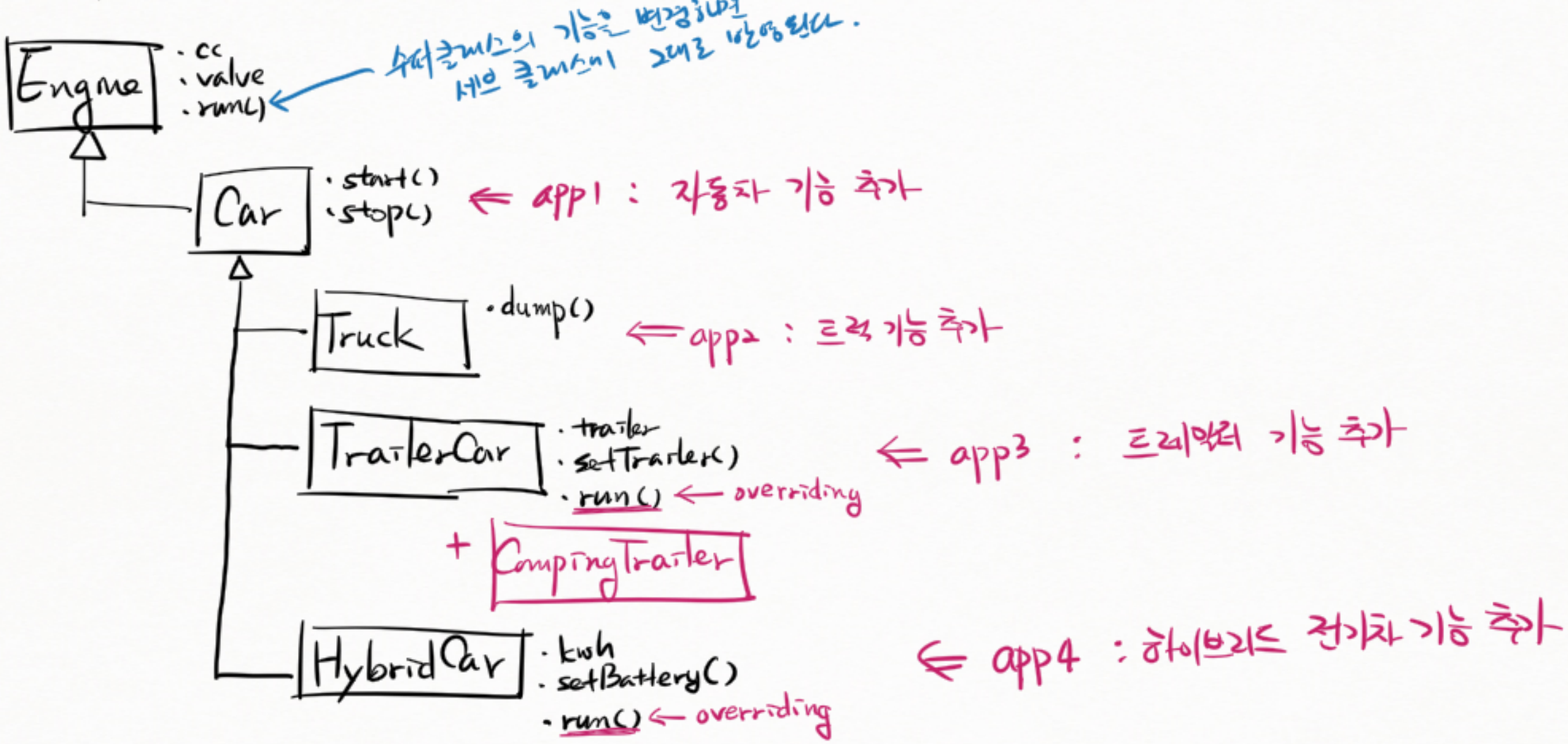
② 트럭 만들기
app2

③ 캠핑카 만들기
app3

④ 하이브리드 전기차 만들기
app4

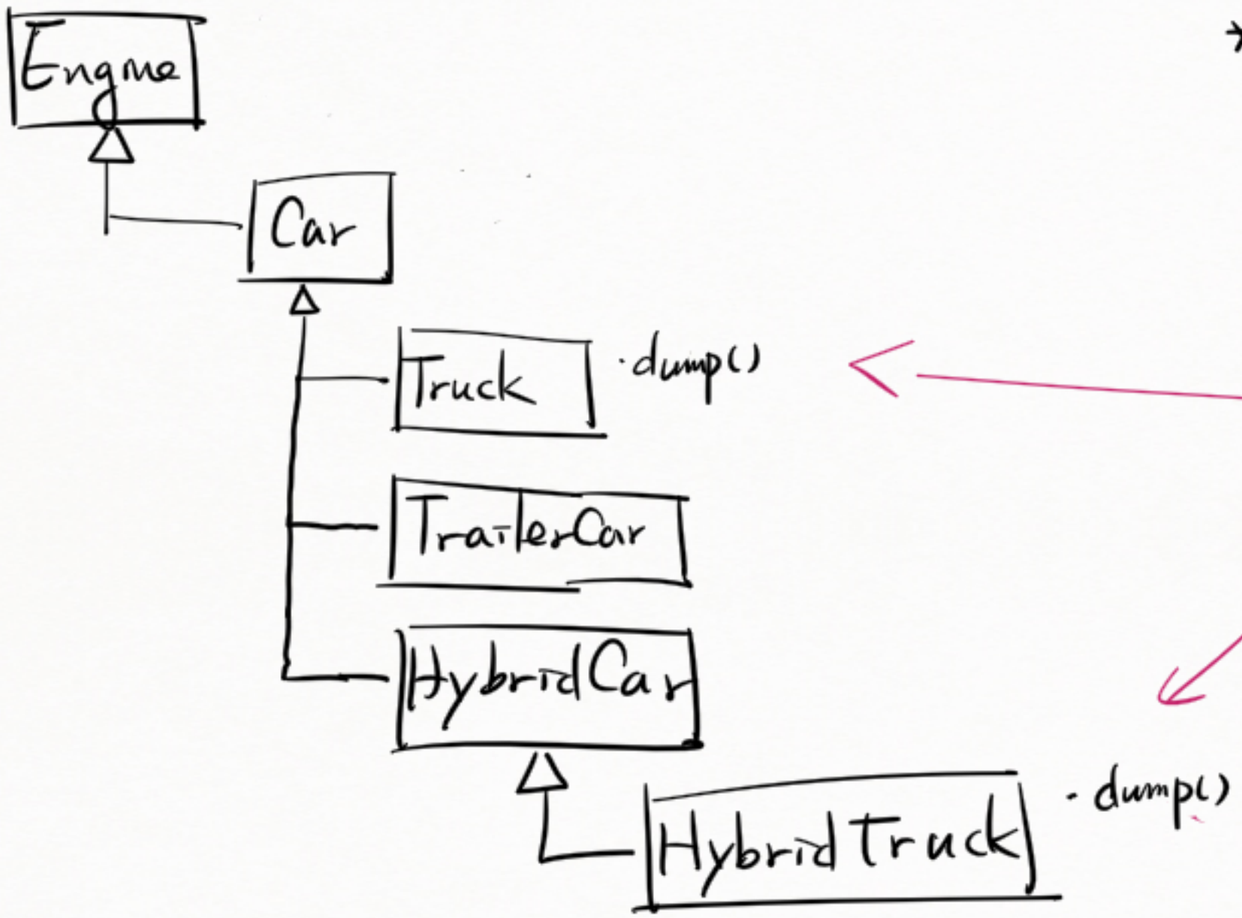
* 기능 확장 방법 3 - 상속 문법을 활용.

oop.ex05.x14.*

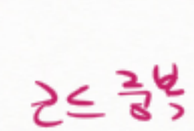


* 기능 확장 방법 3 - 상속을 통한 기능 확장의 한계

oop.ex05.x14.*

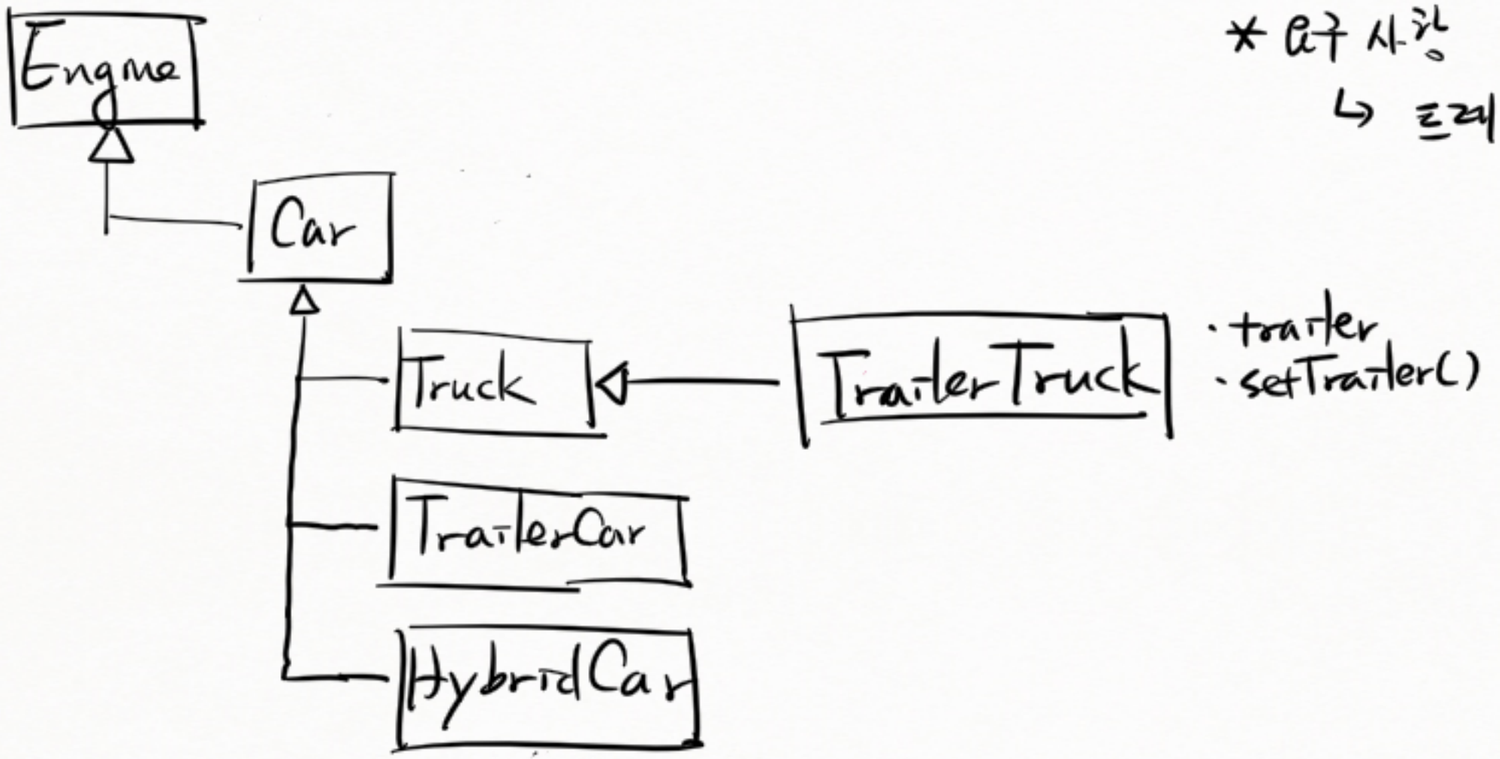


* 요구 사항
  ↳ 전기 트럭을 만들라!



* 기능 확장 방법 3 - 상속을 통한 기능 확장의 한계

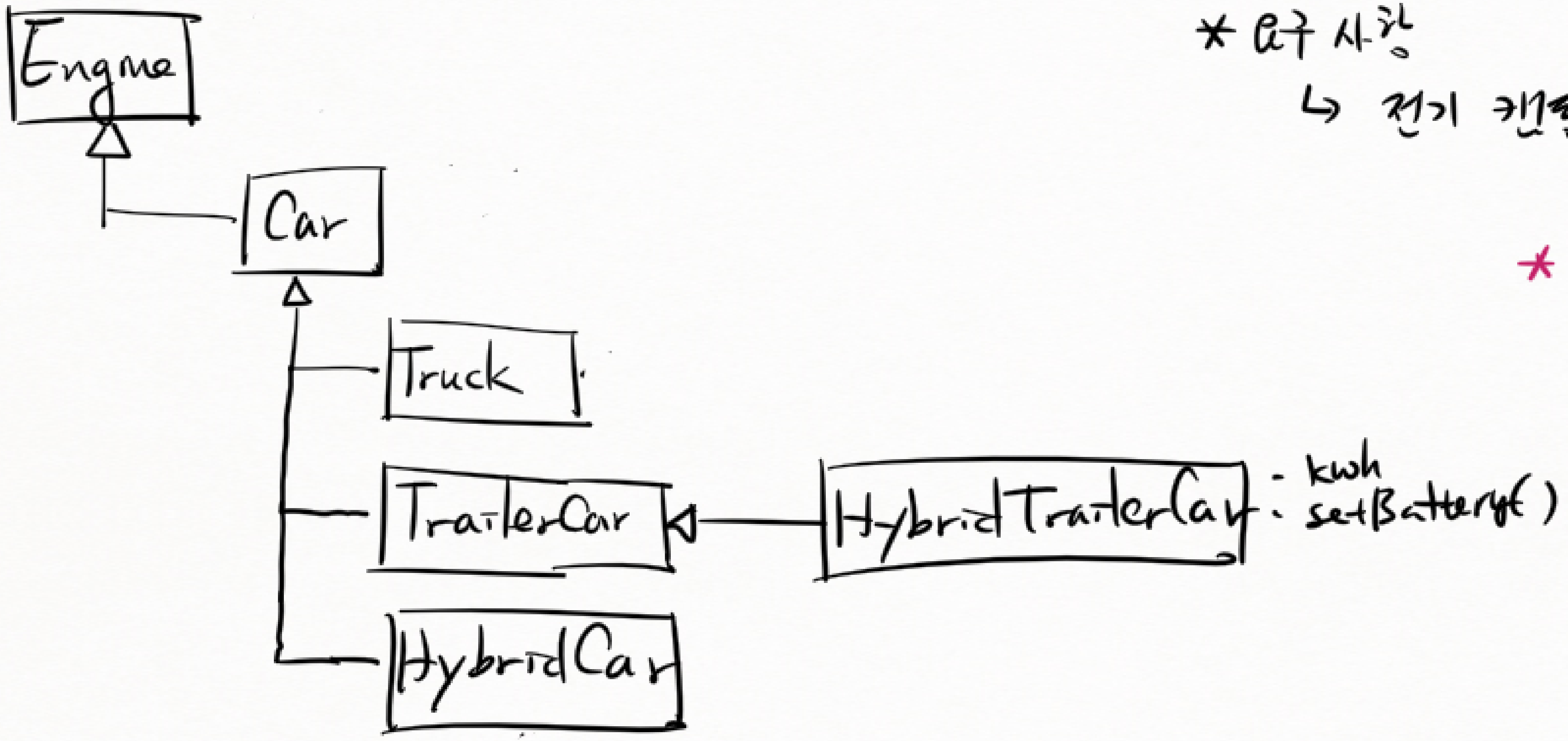
oop.ex05.x14.*



* 요구 사항
  ↳ 트레일러를 붙일 수 있는 트럭

* 기능 확장 방법 3 - 상속을 통한 기능 확장의 한계

oop.ex05.x4.*



* 요구 사항
  ↳ 전기 캠핑카

* 이렇게 기능 조합을 하다보면 수많은 서브클래스가 생성된다

↓

(상속으로 다양한 기능이 조합된 객체를 만들기 번거롭다.)

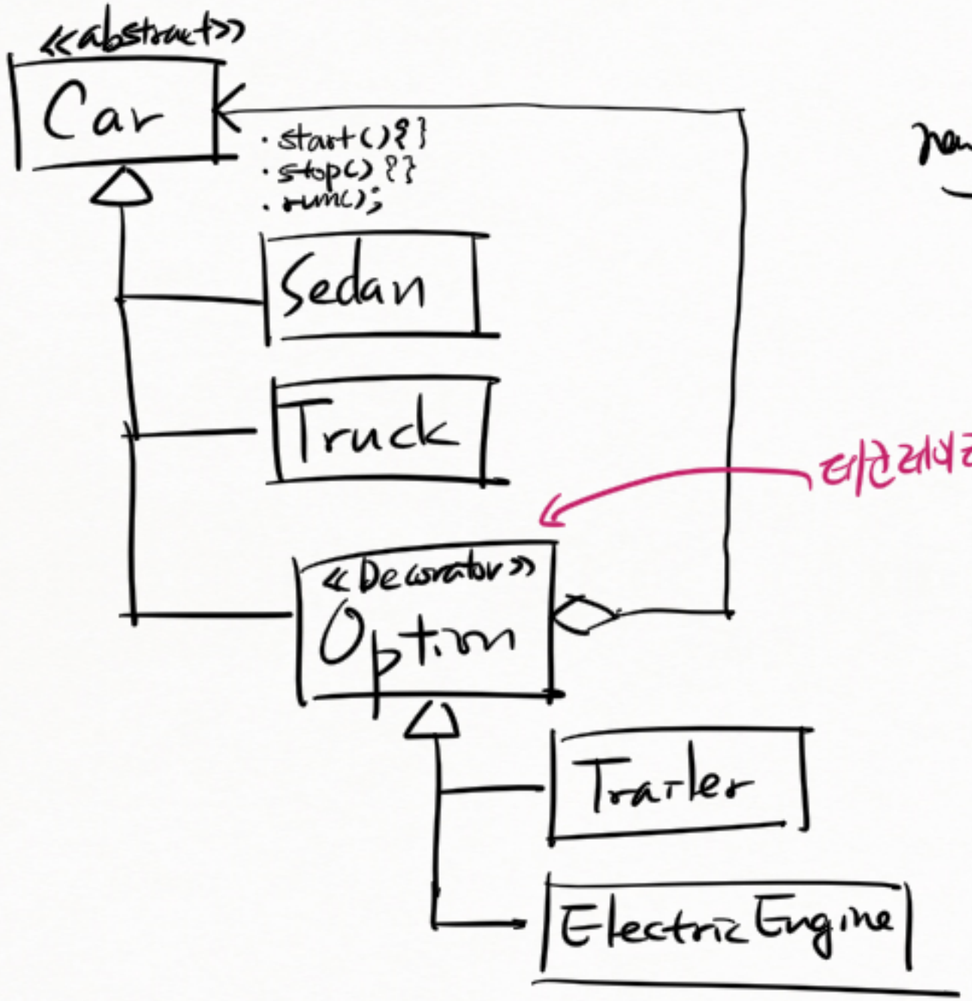
↓

유지보수가 어려워진다.

* 기능확장 방법 4 : 데코레이터 패턴 개념 이용

주 객체	보조기능
Sedan	GasEngine
Truck	ElectricEngine
	Trailer

* 데코레이터 패턴 기반



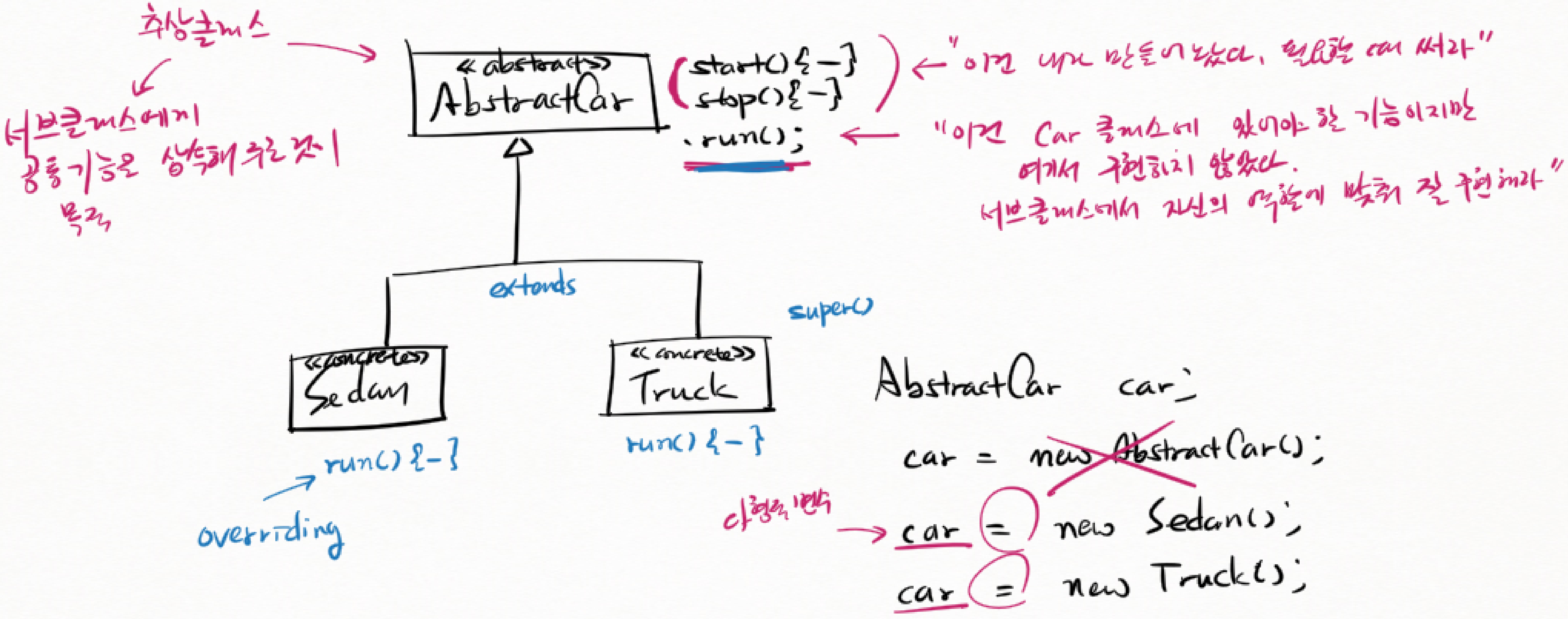
데코레이터 : 객체에게 옵션을 선택기능

new 옵션3(new 옵션2(new 옵션1(객체)))

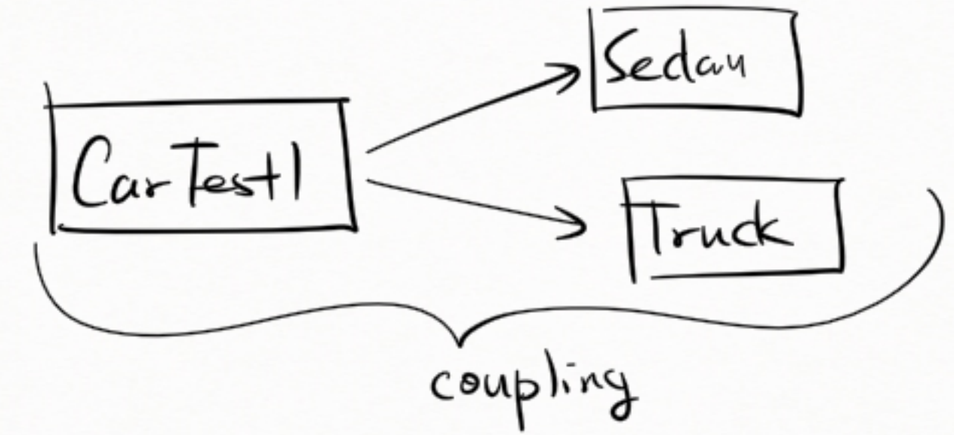
new 옵션3(객체)

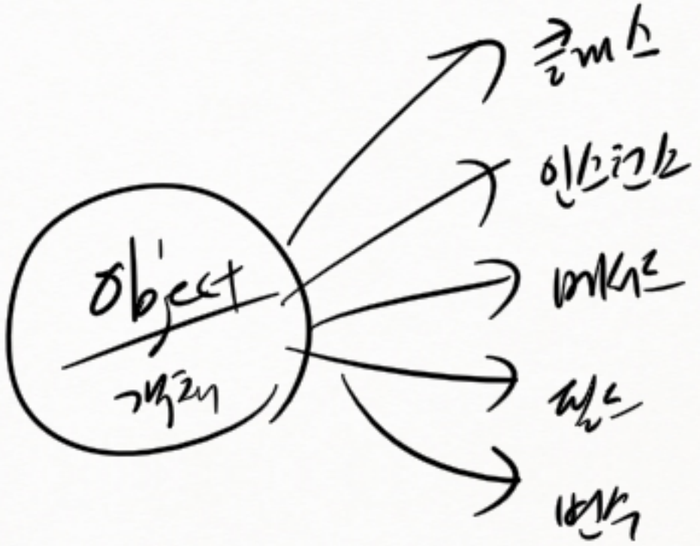
기능을 선택적으로 추가할 수 있다.

* 추상클래스 와 구현


추상클래스
서브클래스에게 공통기능을 상속해 주는 것이 목적
«abstract»
AbstractCar
start(){–}
stop(){–}
run();
"이건 내가 만들어 놨다. 필요할 때 써라"
"이건 Car 클래스에 있어야 할 기능이지만 여기서 구현하지 않았다. 서브클래스에서 자신의 역할에 맞춰 잘 구현해라"
extends
super()
«concrete»
Sedan
run(){–}
overriding
«concrete»
Truck
run(){–}
AbstractCar car;
car = new AbstractCar();
다형적 변수
car = new Sedan();
car = new Truck();

* 실전 프로그래밍 1 만제

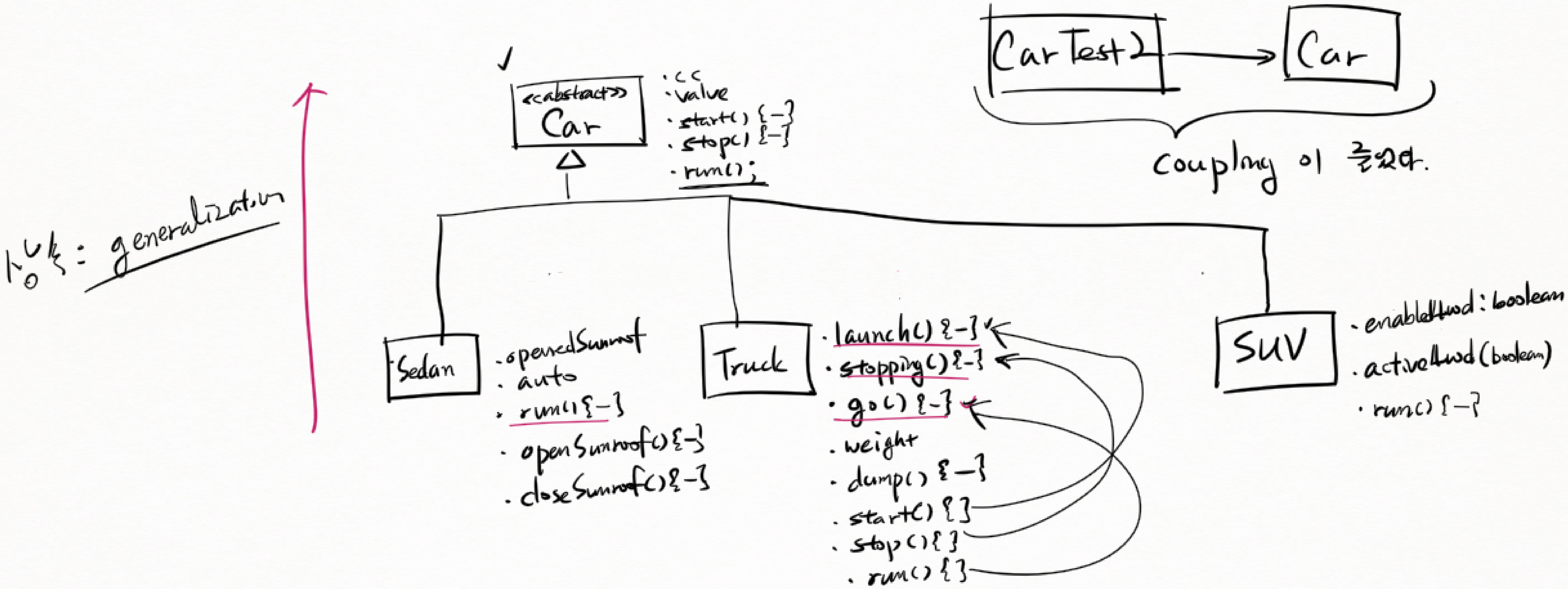




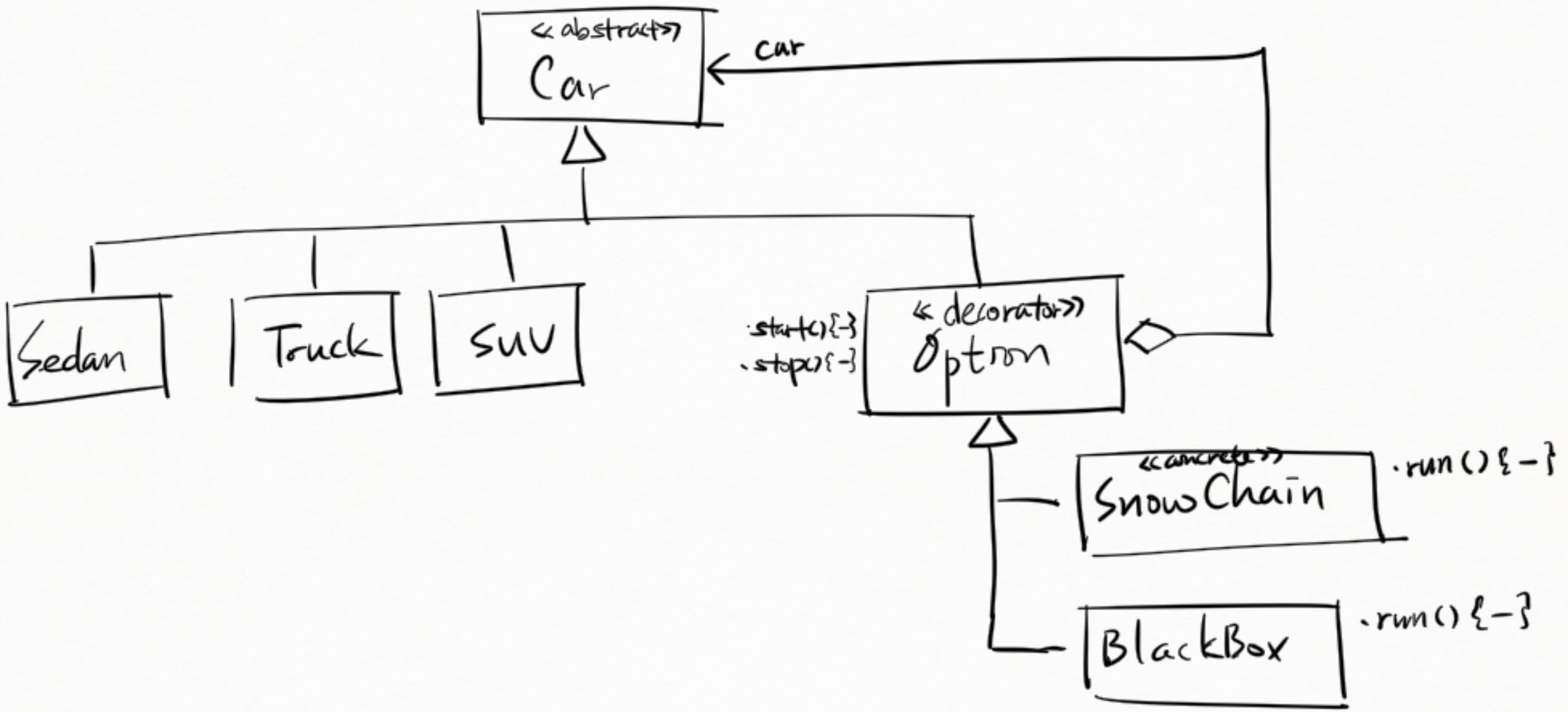
Sedan
- openedSunroof
- auto
- cc
- value
- start() {-}
- stop() {-}
- run() {-}
- openSunroof() {-}
- closeSunroof() {-}

Truck
- cc
- value
- launch() {-}
- stopping() {-}
- go() {-}
- weight
- dump() {-}

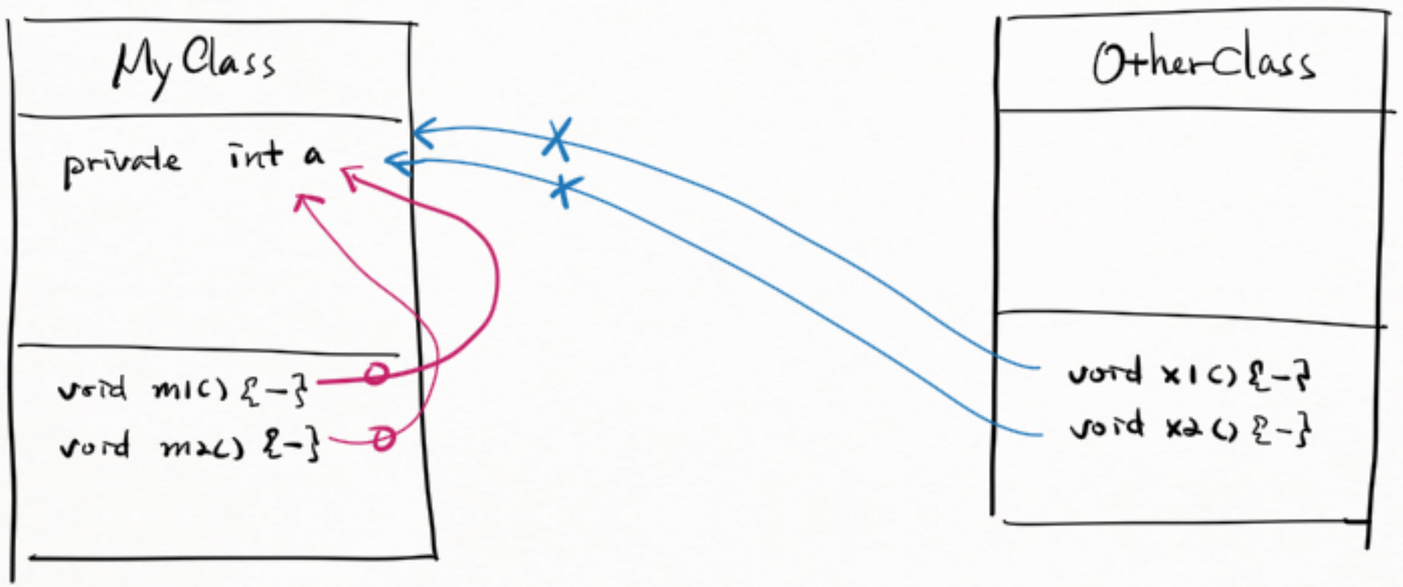
* 실전프로그래밍 2단계 - 상속, 오버라이딩, 다형성 변수



*실전프로그래밍 3단계 - 데코레이터 패턴 개념 정리



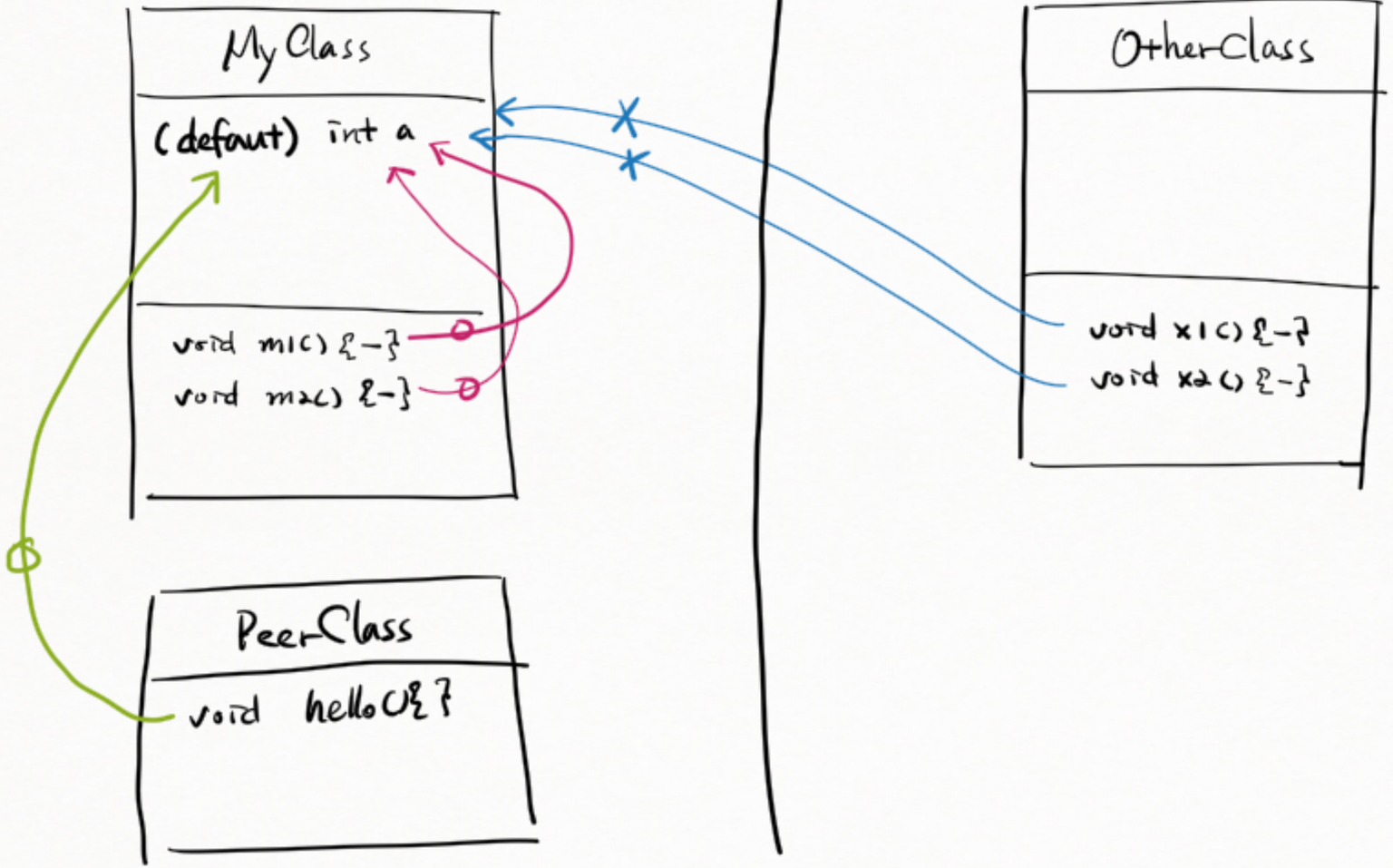
* private – 같은 클래스의 멤버만 접근 가능



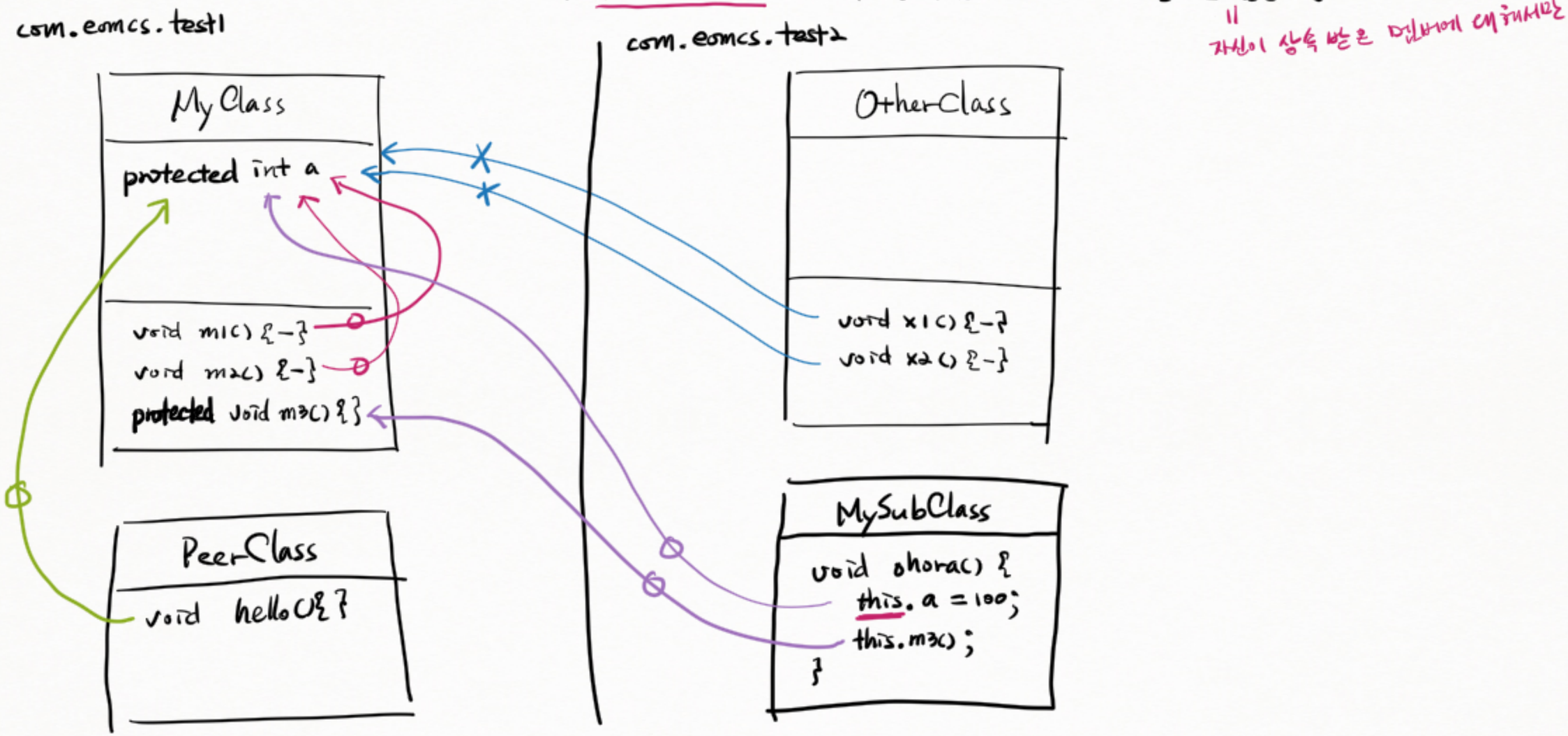
* (default) – 같은 클래스의 멤버 접근 가능
+ 같은 패키지 소속 클래스의 멤버 접근 가능

com.eomcs.test1

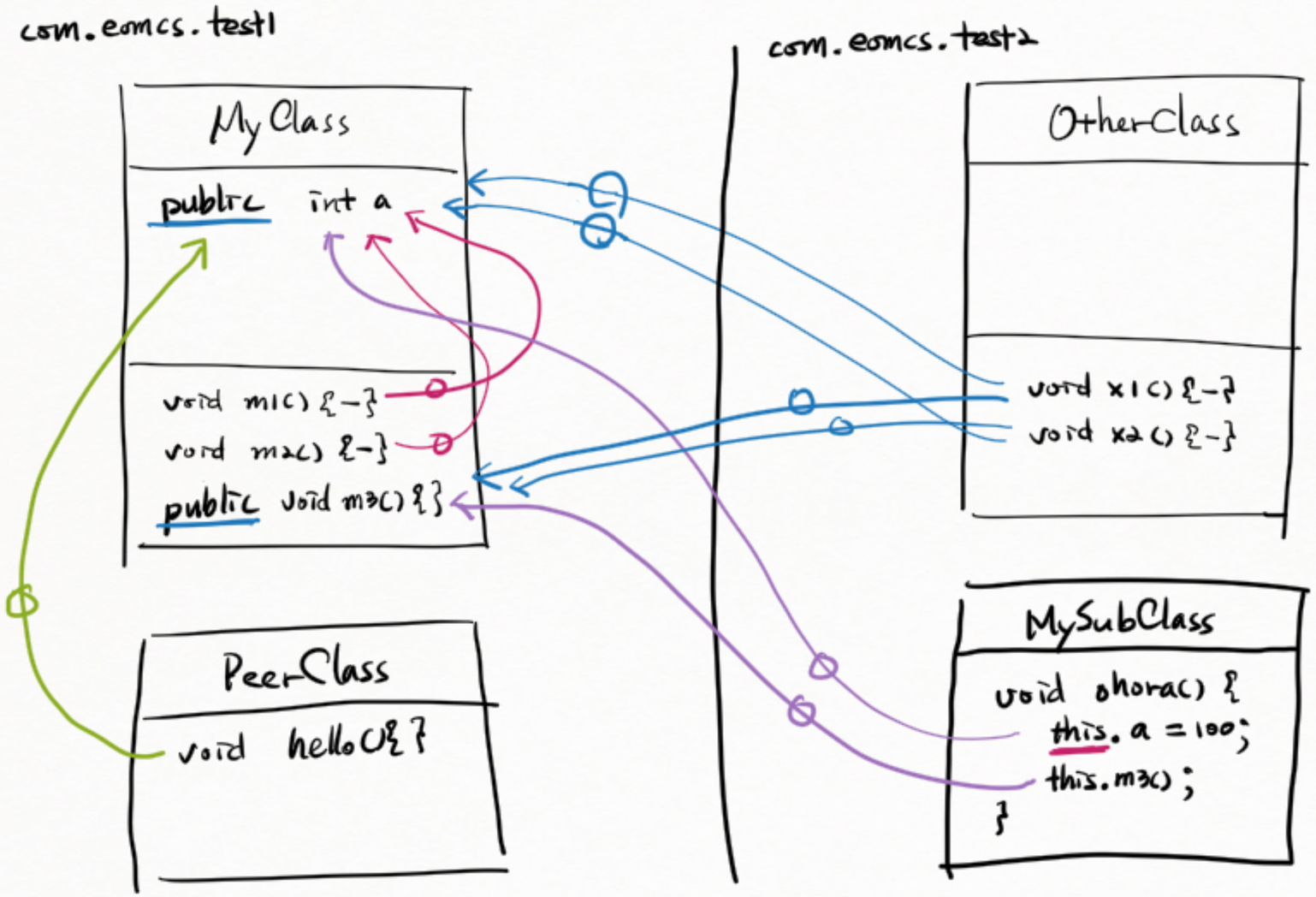
com.eomcs.test2

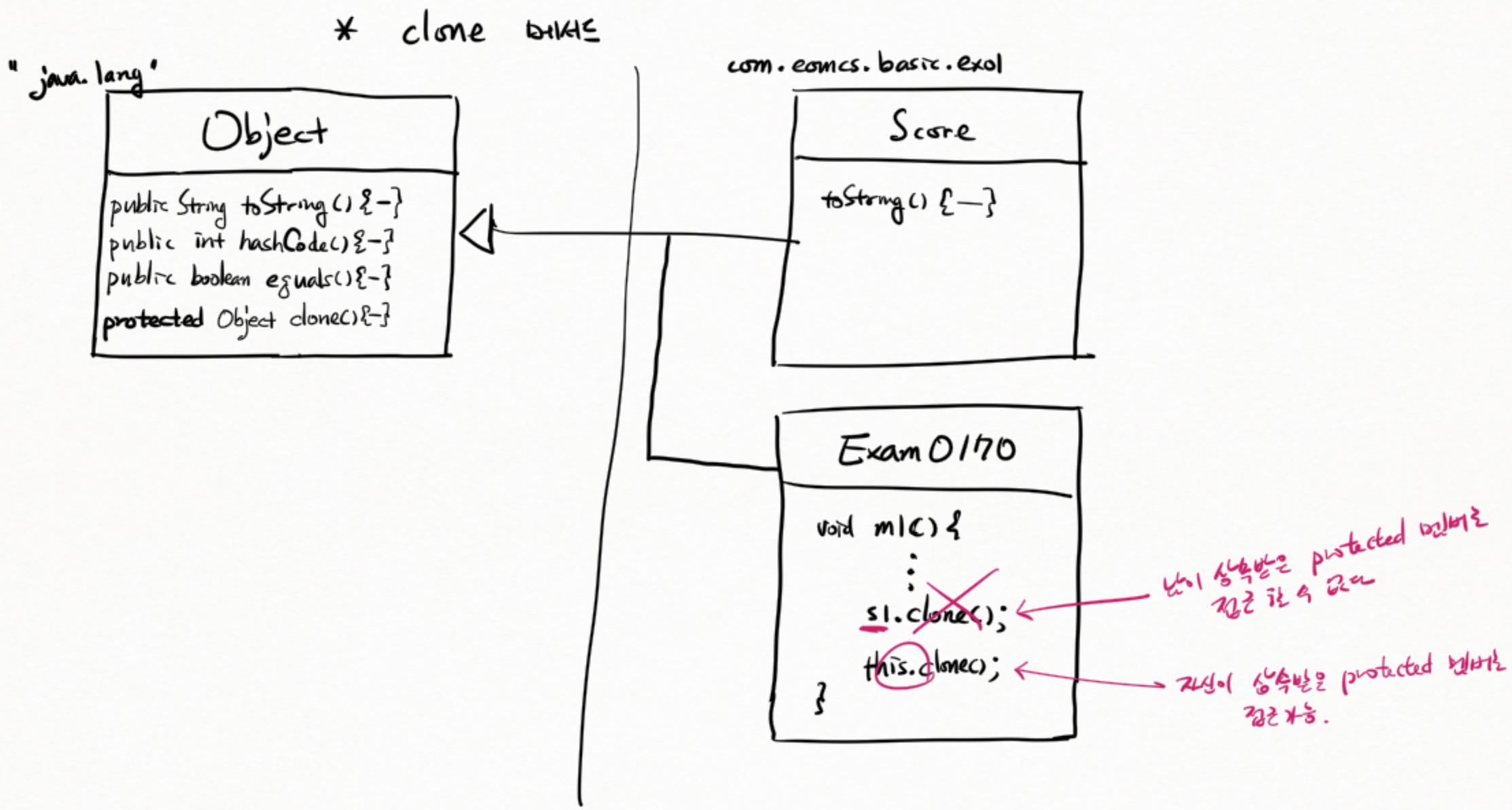


* protected – 같은 클래스의 멤버 접근 가능
+ 같은 패키지 소속 클래스의 멤버 접근 가능 + 서브클래스 접근 가능
= 자신이 상속 받은 멤버에 대해서만

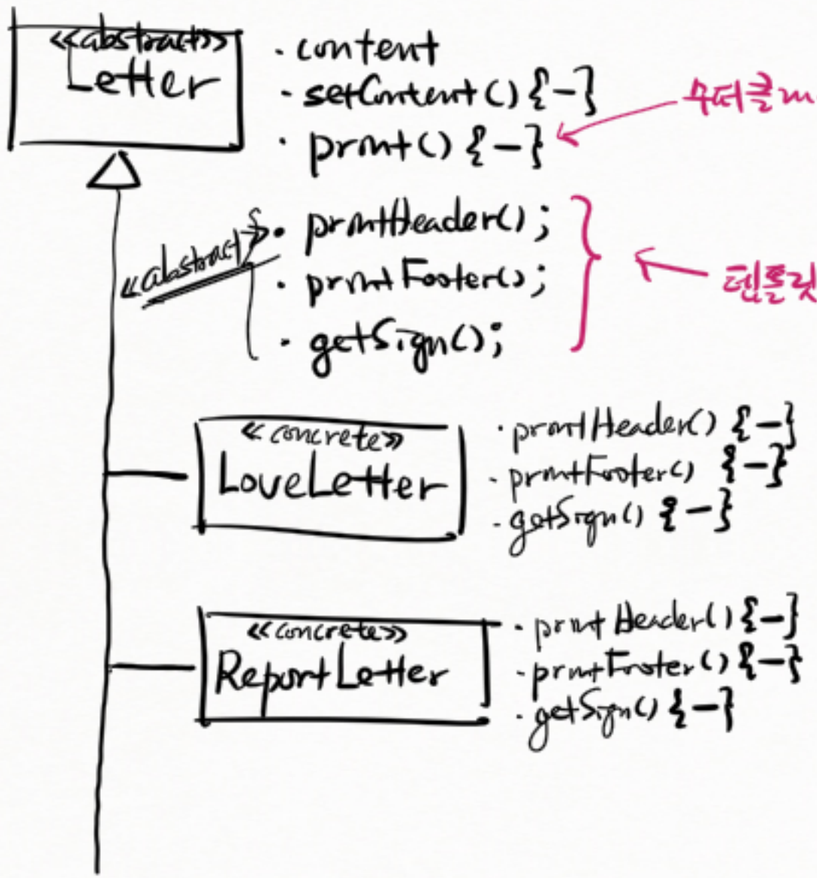


✳ public - 모든 멤버 접근 가능




* clone 메서드
"java.lang"
Object
public String toString(){-}
public int hashCode(){-}
public boolean equals(){-}
protected Object clone(){-}
com.eomcs.basic.ex01
Score
toString(){—}
Exam0170
void m1(){
s1.clone();
this.clone();
}
남이 상속받은 protected 멤버를 접근할 수 없다
자신이 상속받은 protected 멤버는 접근가능.

* 추상클래스



the skeleton of an algorithm

상위클래스에서 기능의 흐름을 정의. 틀을 구성. (템플릿) ⇐ 템플릿 메서드

템플릿 안에서 실행될 세부 동작은 서브클래스에게 구현을 맡긴다

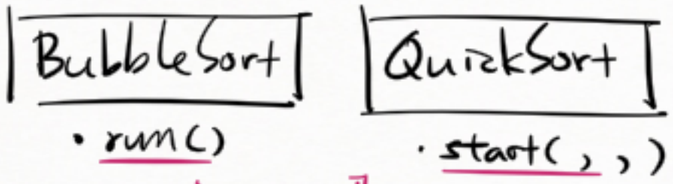
상위클래스: 실행흐름을 정의하고 서브클래스로 구체적인 동작을 정의

⇓ 범용기법

"템플릿 메서드" 패턴 (template method)

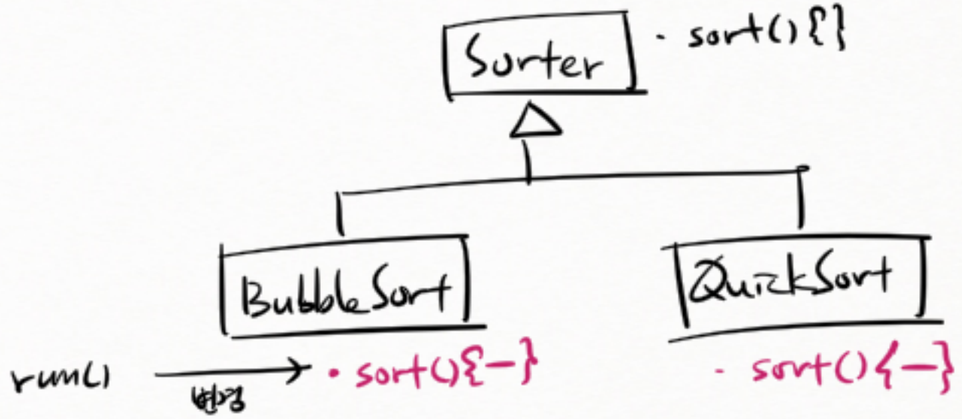
* 추상 클래스와 추상메서드의 활용

① 다른 타입의 클래스 사용 → 개선 → ② 같은 타입의 클래스로 묶기



두 개의 정렬 클래스를 같은 부류가 아니기 때문에
한 개의 print() 메서드를 사용할 수 없고
따로 사용법도 다르다.

- display (BubbleSort, int[]) {-}
- display (QuickSort, int[]) {-}



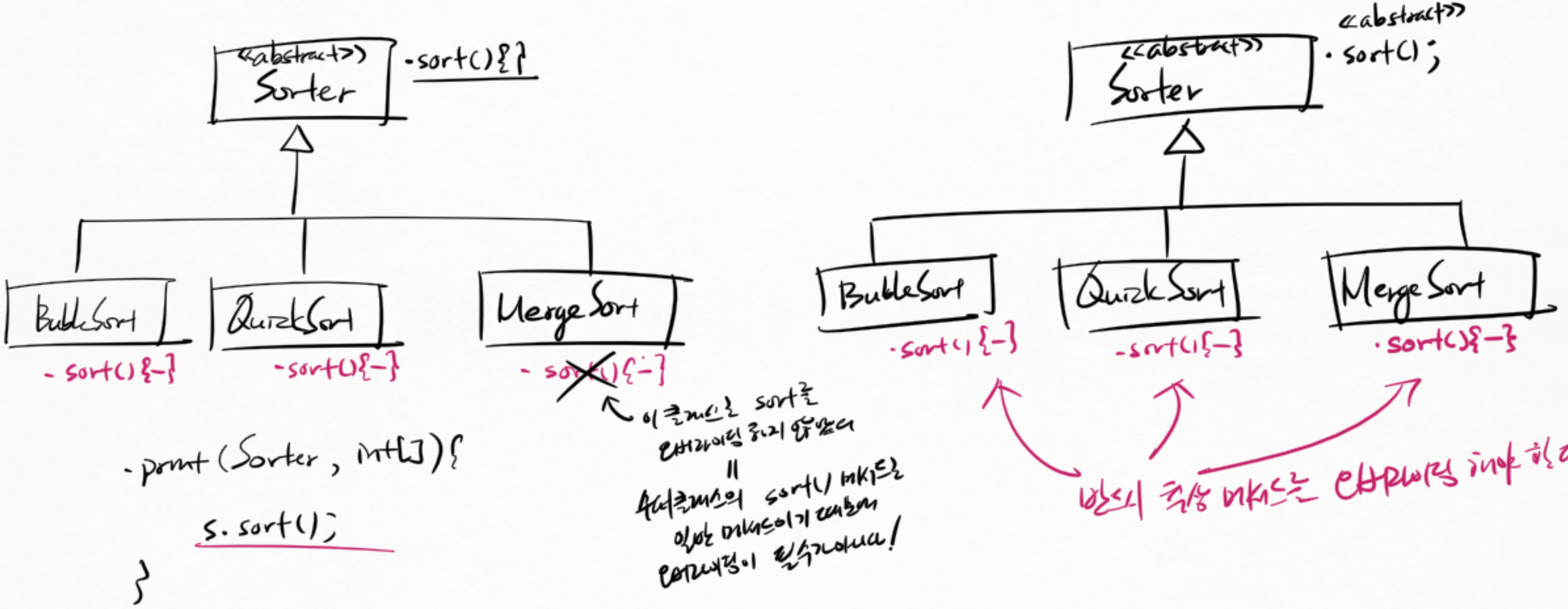
같은 타입이라서
print()를 두 개씩 만들 필요 없다.

```
• display (Sorter, int[]) {
      s.sort();
  }
```

클래스의 상관없이
같은 이름의 메서드 호출
=
일관된 사용 가능

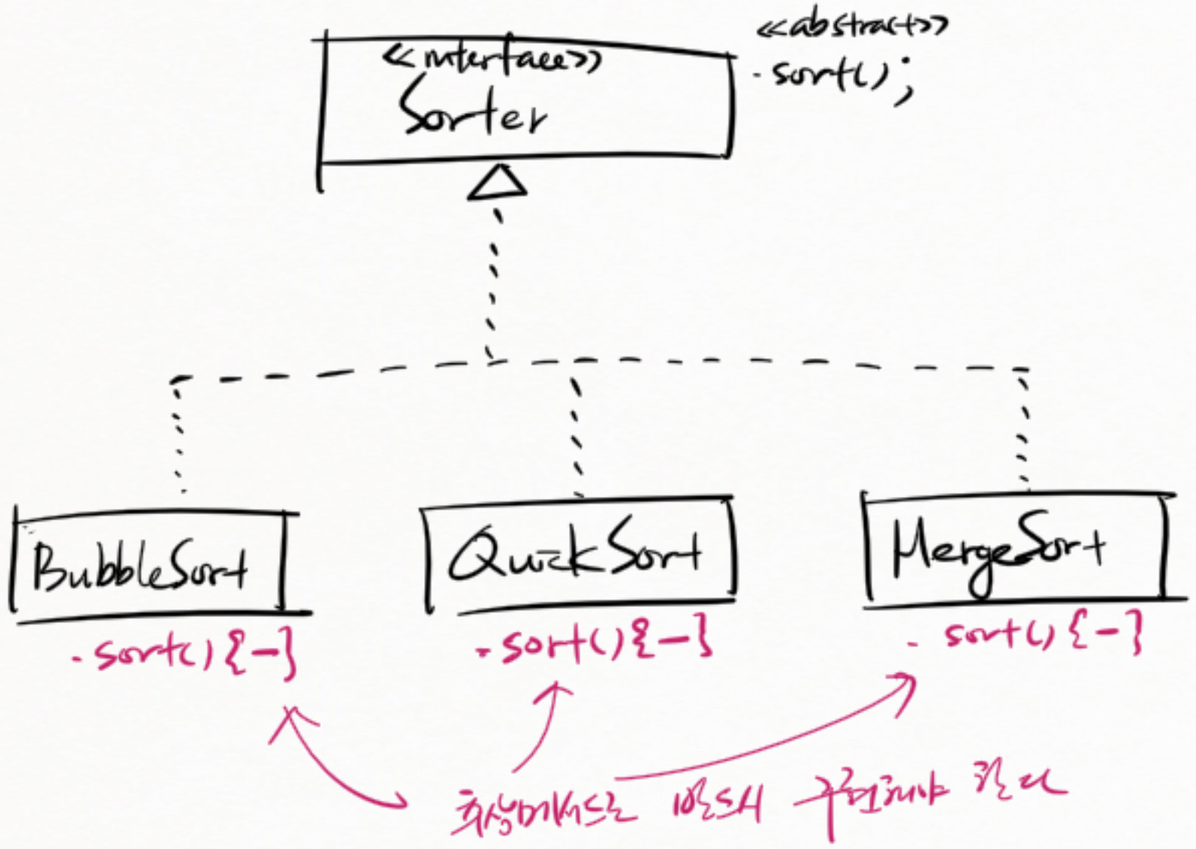
* 추상 클래스와 추상 메서드의 활용

③ 부모클래스를 추상클래스로 바꾸기 → ④ 추상메서드로 오버라이딩을 강제하기

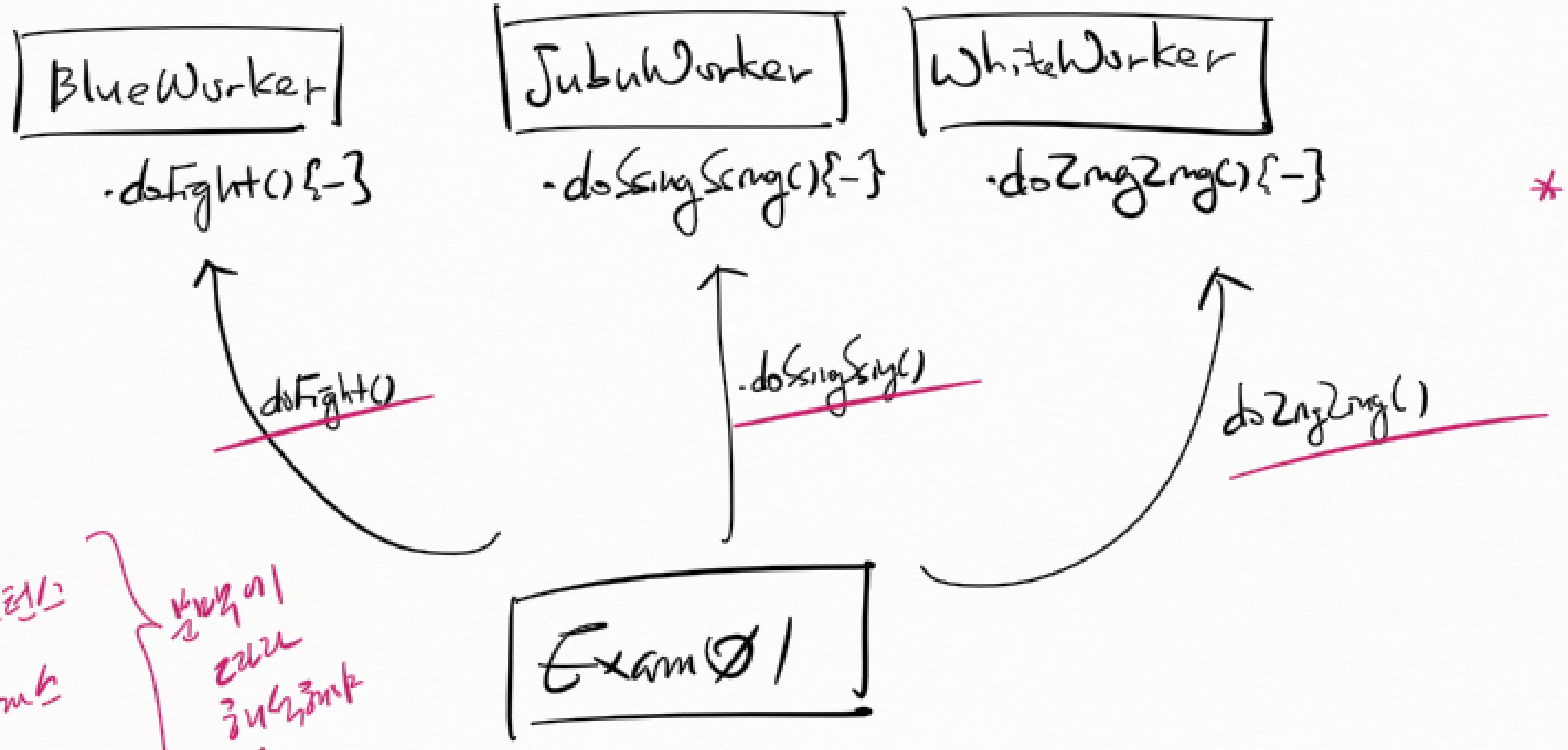


* 추상클래스 대신 인터페이스를 쓸 때

⑤ 추상 클래스 대신 인터페이스를 클래스 쓰기



* 인터페이스 사용 전 : — oop.ex09.a1.before



"객체" 용어
- → 인스턴스
- → 클래스
- → ...

문맥에 따라 해석해야 한다.

* 객체마다 (클래스) 메서드 시그너처가 다르다. (signature)

↓

객체 사용법이 다르다

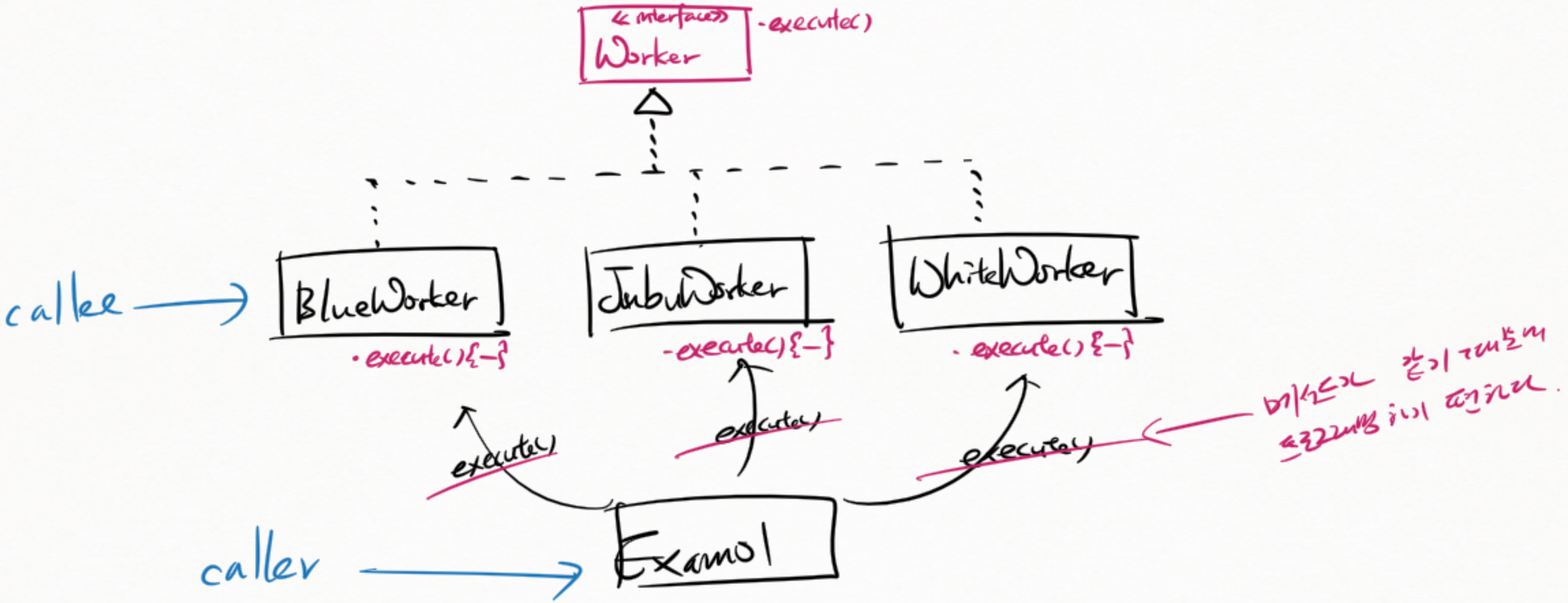
↓

프로그래밍이 불편해 진다.

↓

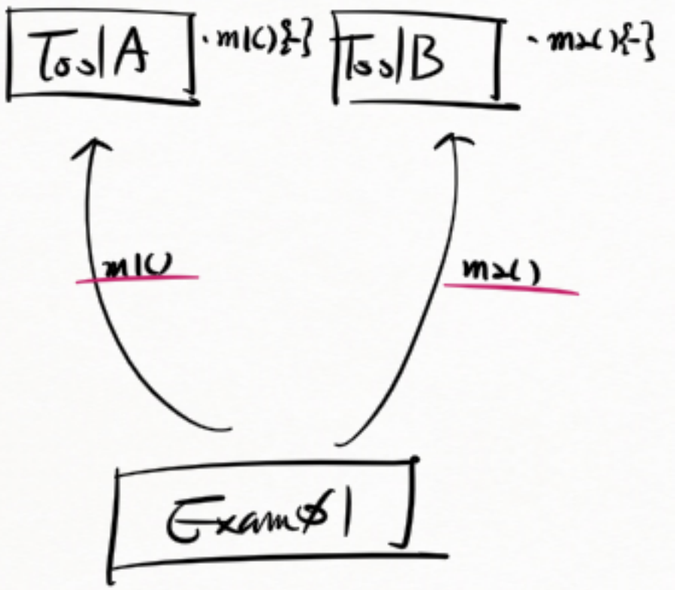
유지보수 어렵다.

* 인터페이스 사용 예 : - oop.ex09.a1.after



* 인터페이스 사용 전/후

① 사용전



도구마다 사용법이 다르니
m1(), m2()마다도 호출

↓

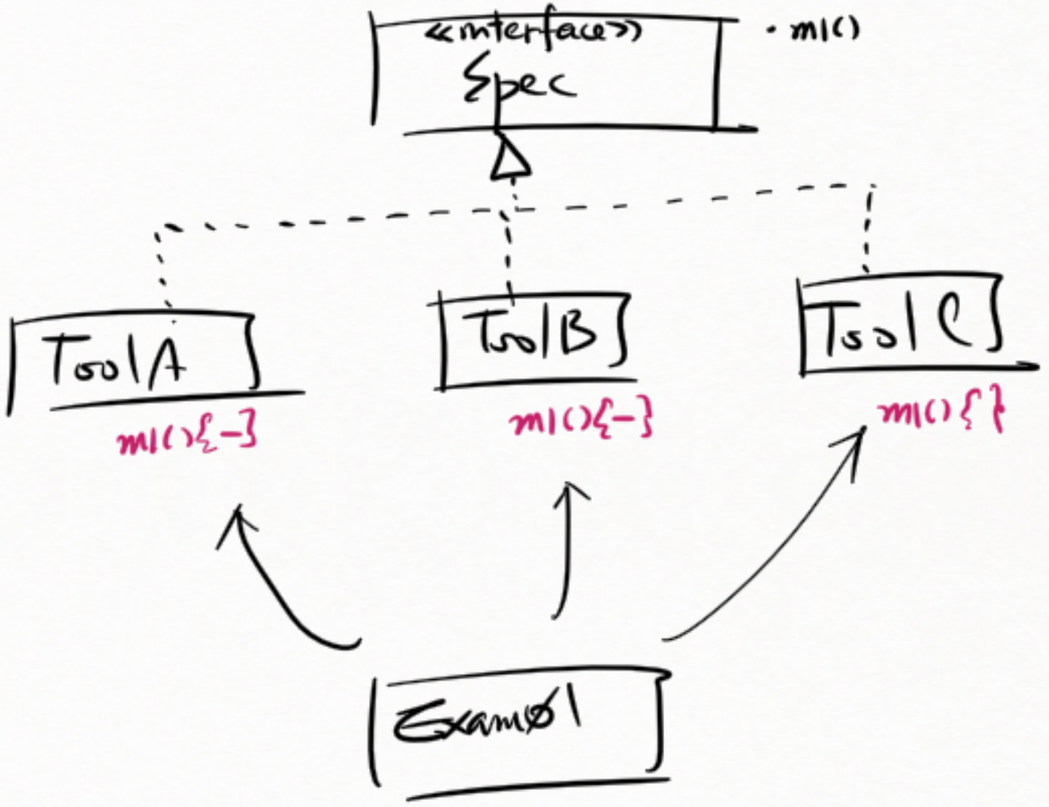
프로그램의 일관성이 없다

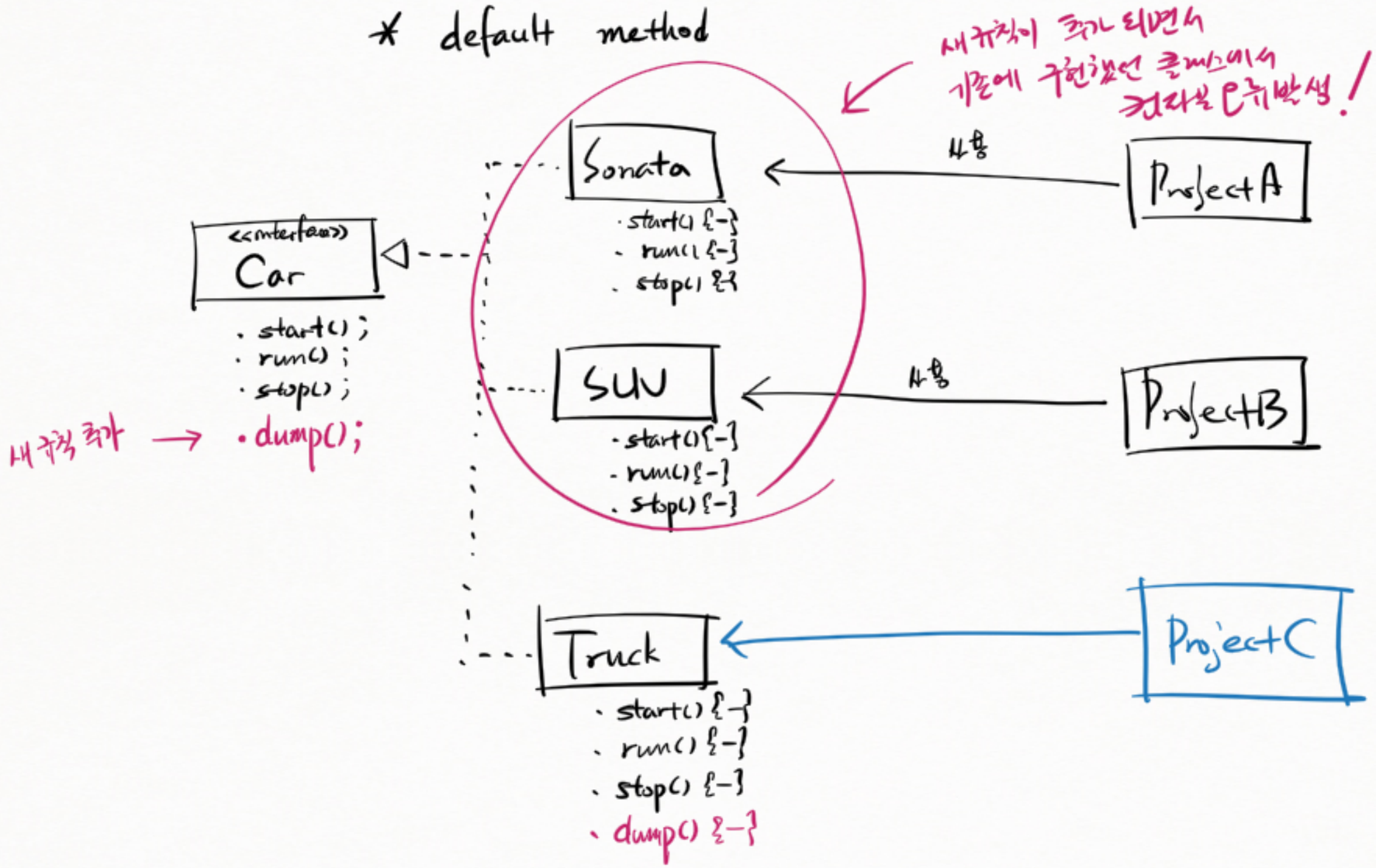
↓

유지 보수가 힘들다.

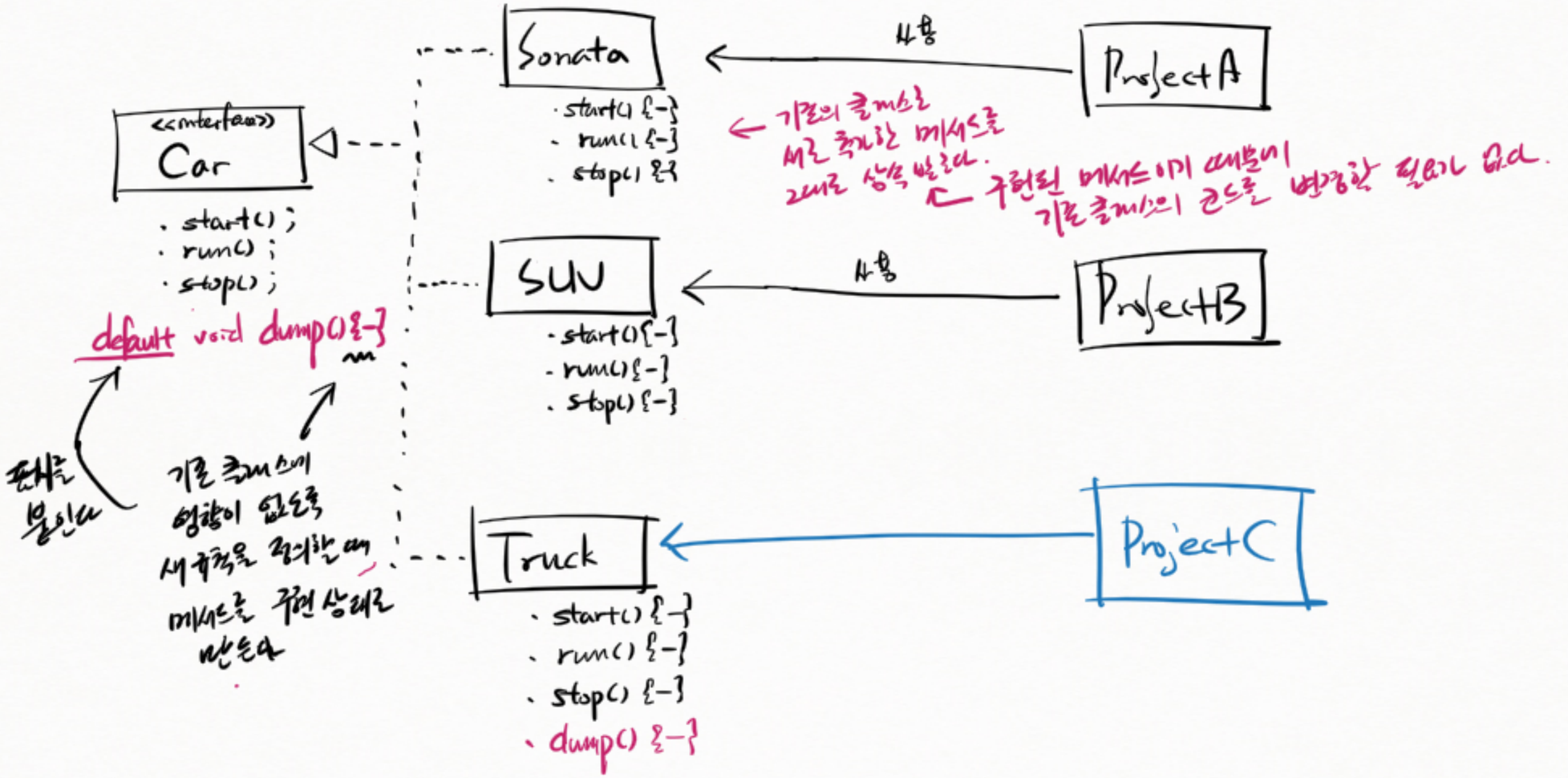
개선

② 사용후

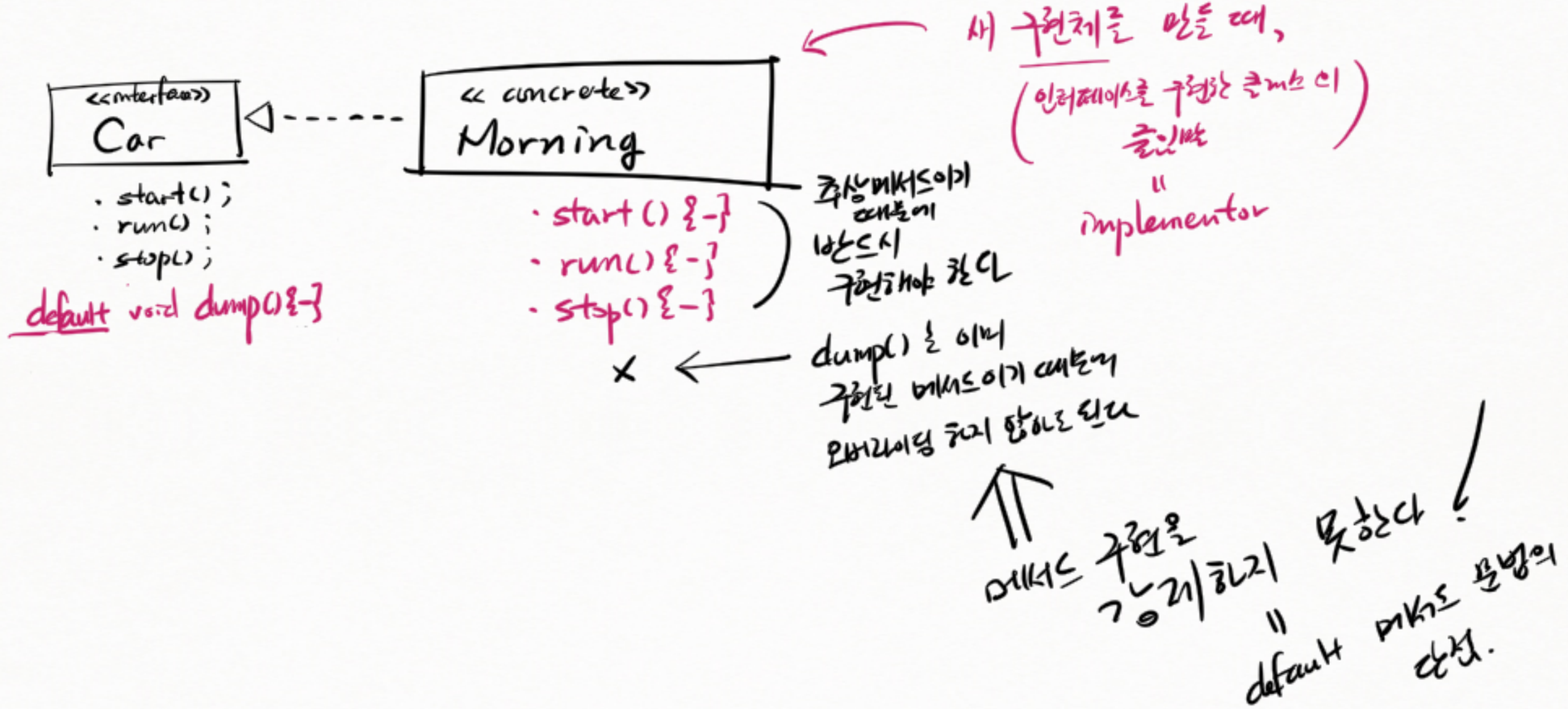



* default method
새 규칙이 추가되면서
기존에 구현했던 클래스에서
컴파일 에러발생!
Sonata
· start() {-}
· run() {-}
· stop() {-}
<<interface>>
Car
· start();
· run();
· stop();
새 규칙 추가 → · dump();
SUV
· start() {-}
· run() {-}
· stop() {-}
Truck
· start() {-}
· run() {-}
· stop() {-}
· dump() {-}
사용
사용
ProjectA
ProjectB
ProjectC

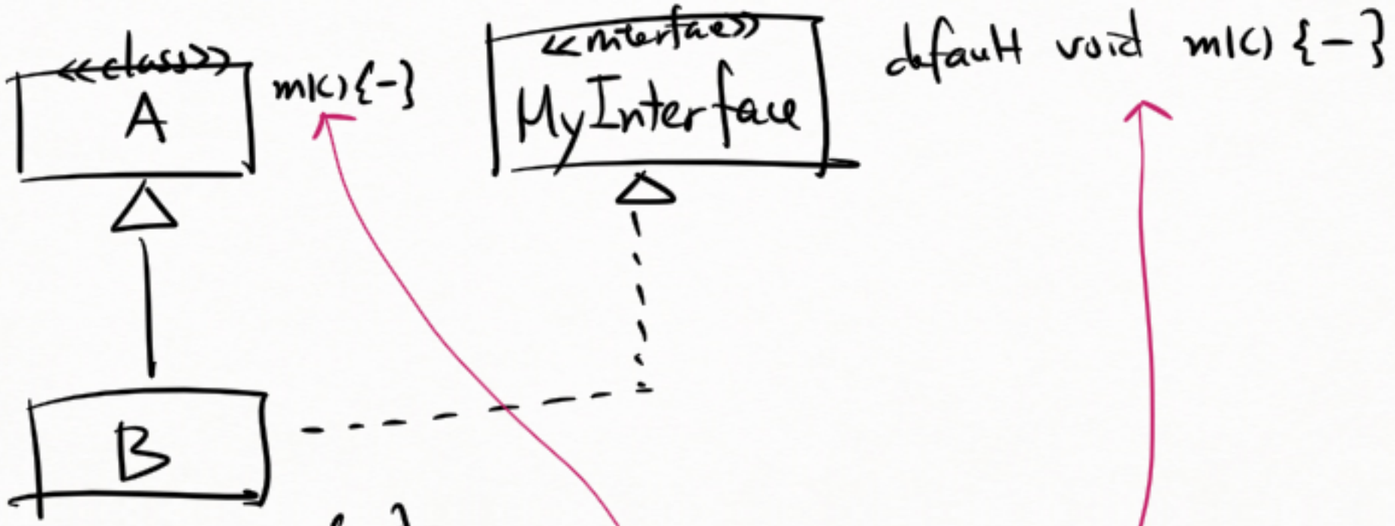
* default method ⇐ 기존의 구현 클래스에 영향을 주지 않으면서 새 규칙을 추가하고 싶을 때,



* default method 의 문제점

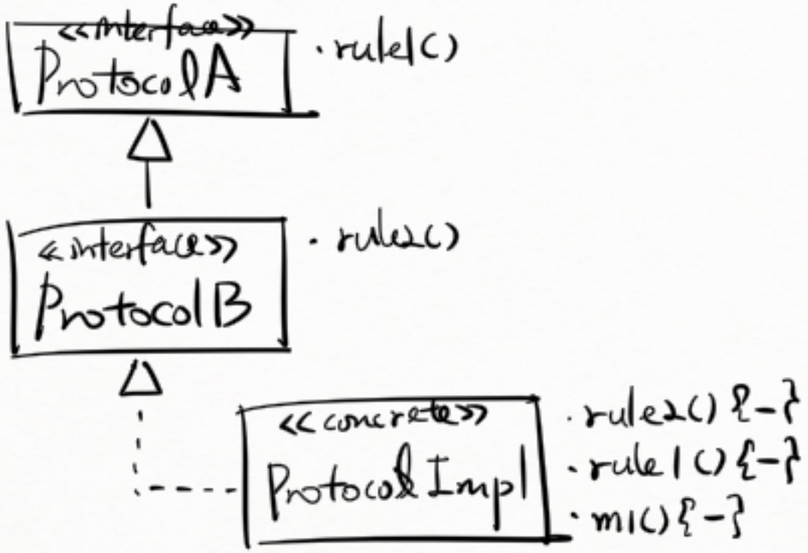


* super 와 인터페이스.super




```
m1(){-}
test(){
    this.m1();
    super.m1();
    MyInterface.super.m1();
}
```

* 인터페이스 상속 — oop.exo9.c.*

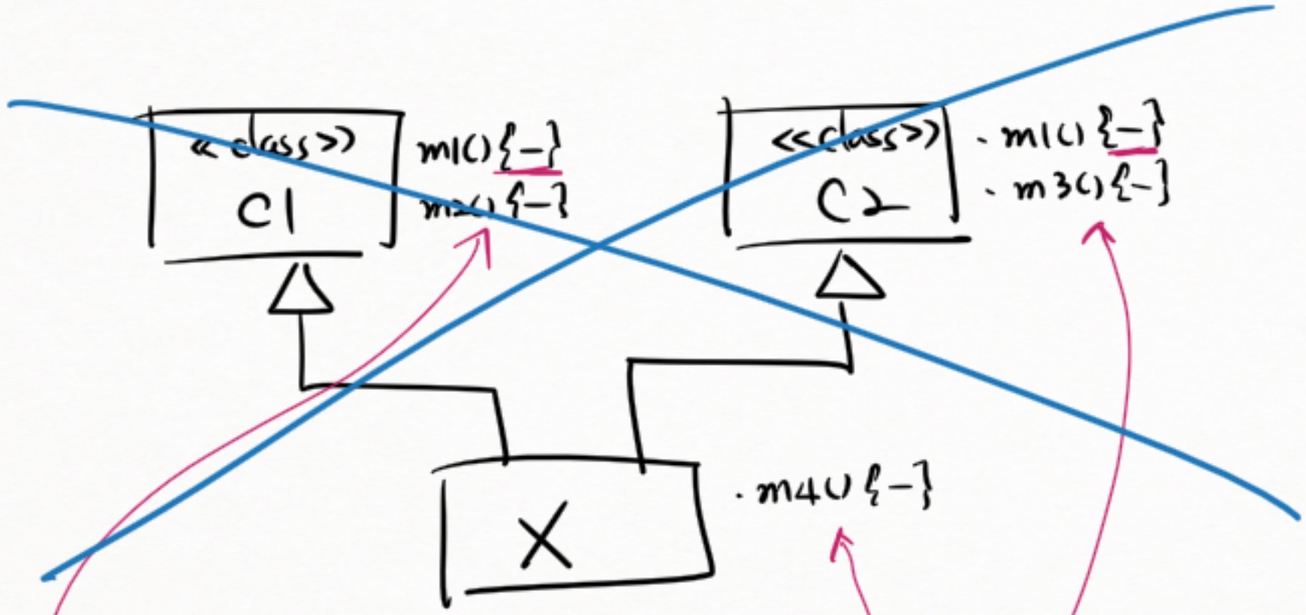



```
ProtocolImpl obj = new ProtocolImpl();
obj.m1();    // ok
obj.rule2(); // ok
obj.rule1(); // ok

ProtocolB b = obj;
b.m1();    // error
b.rule2(); // ok
b.rule1(); // ok

ProtocolA a = obj;
a.m1();    // error
a.rule2(); // error
a.rule1(); // ok
```

* 클래스 다중 상속 과 인터페이스 다중 상속




```
X obj = new X();
obj.m4();
obj.m2();
obj.m3();
obj.m1();
```

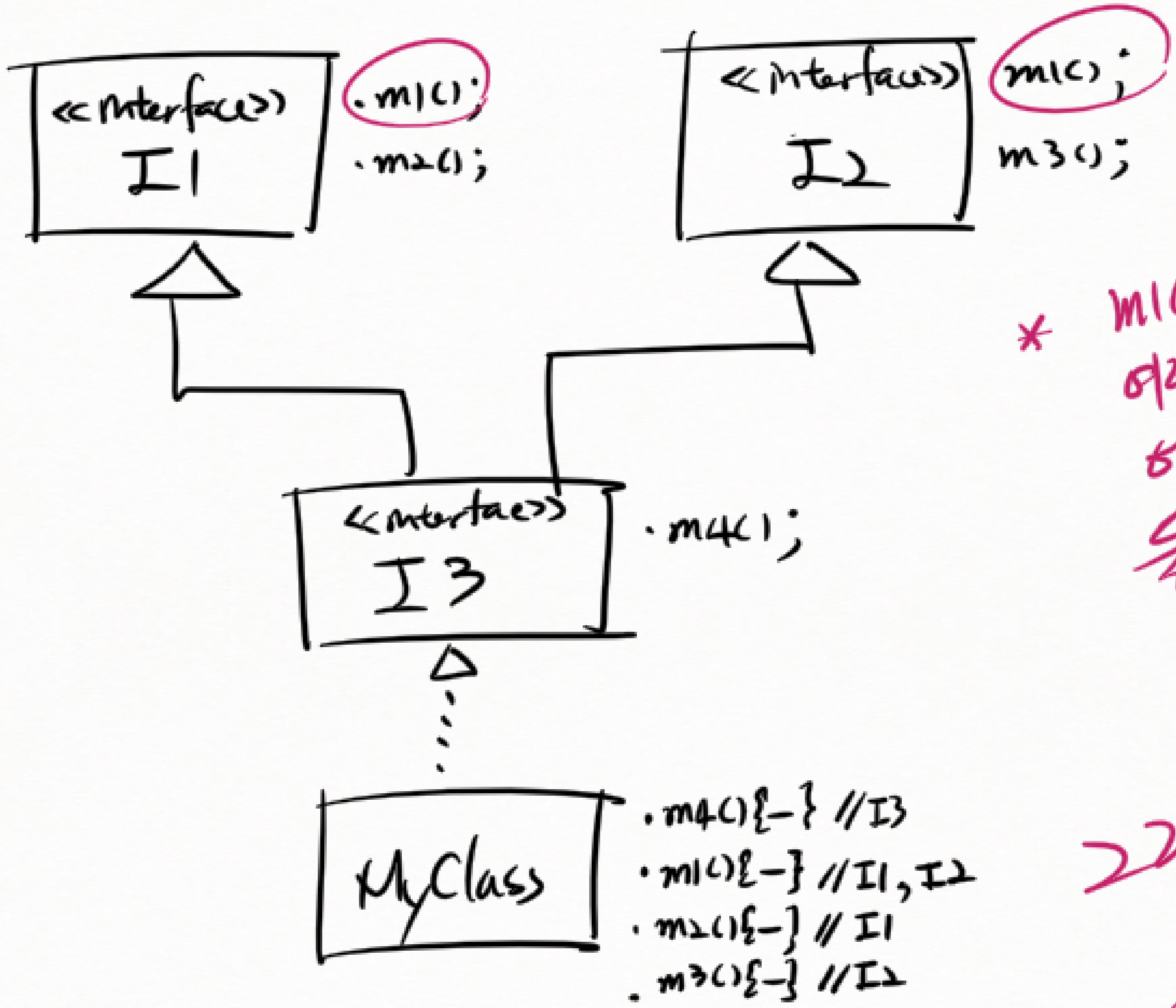
? ← → ?

C1의 m1()을 호출하는가?
C2의 m1()을 호출하는가? 의 대상

그래서 자바는 다중 상속을 허용하지 않는다.

m1()은 구현된 메서드이기 때문에 CH?

* 클래스 다중 상속 vs 인터페이스 다중상속

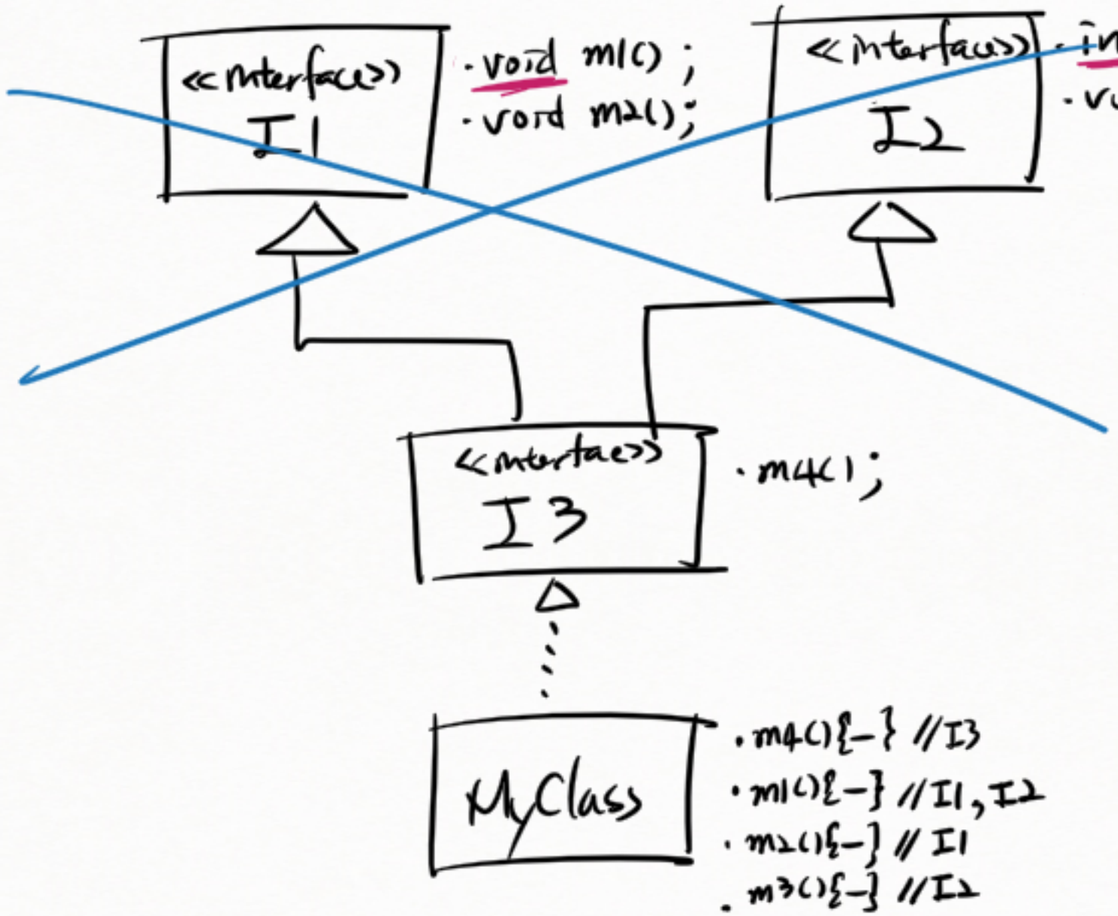


* m1() 메서드처럼 여러 수퍼클래스에 존재하더라도 어차피 "구현하지 않은 메서드"이기 때문에 둘 중 어떤것을 상속받더라도 문제될게 없다

⇓

그래서 인터페이스는 다중상속을 허락한다.

* 인터페이스 다중 상속 불가!

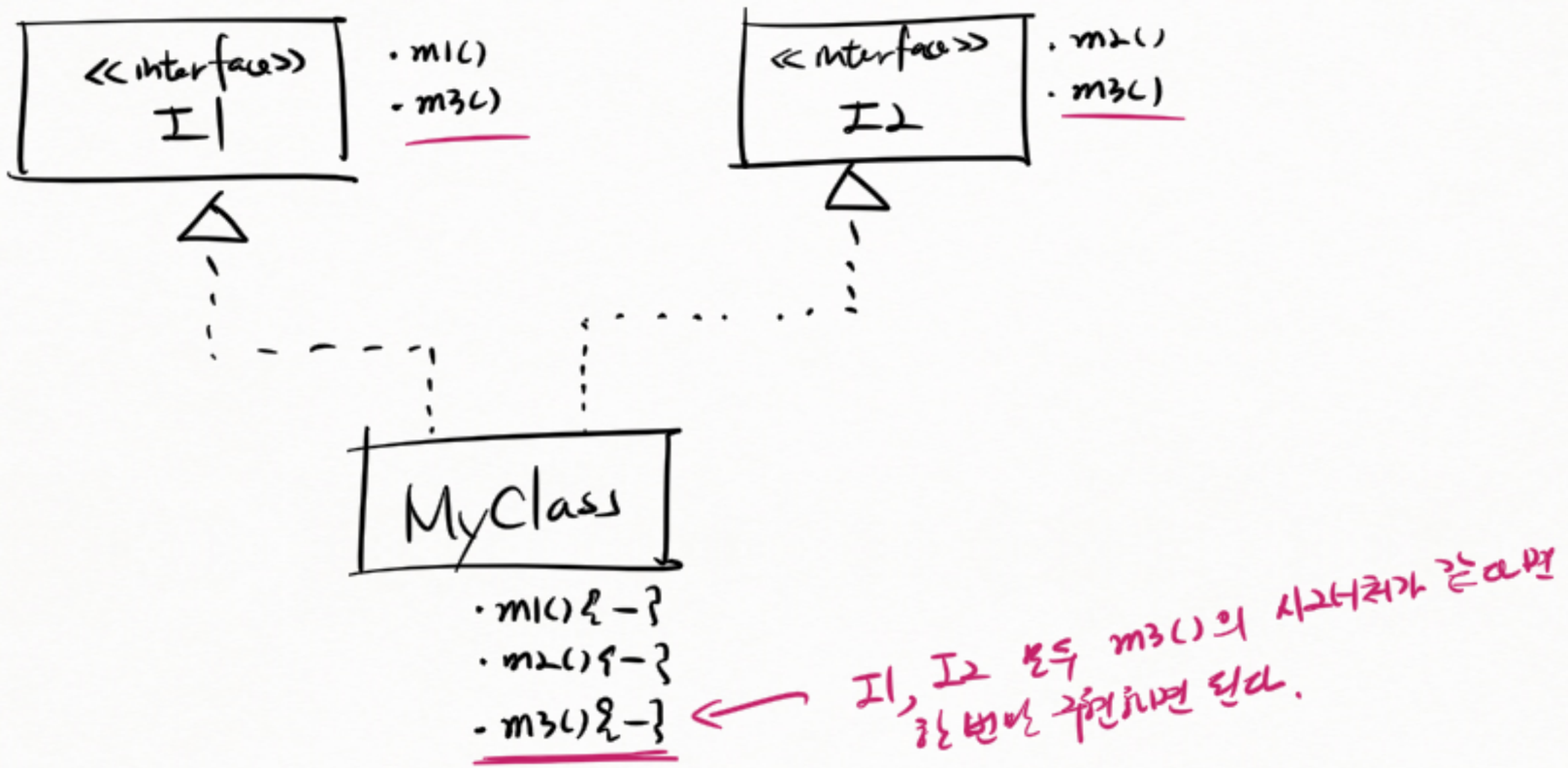


* I1의 m1()과 I2의 m1()의 메서드 시그니처가 다르다.
리턴타입이 다르다!

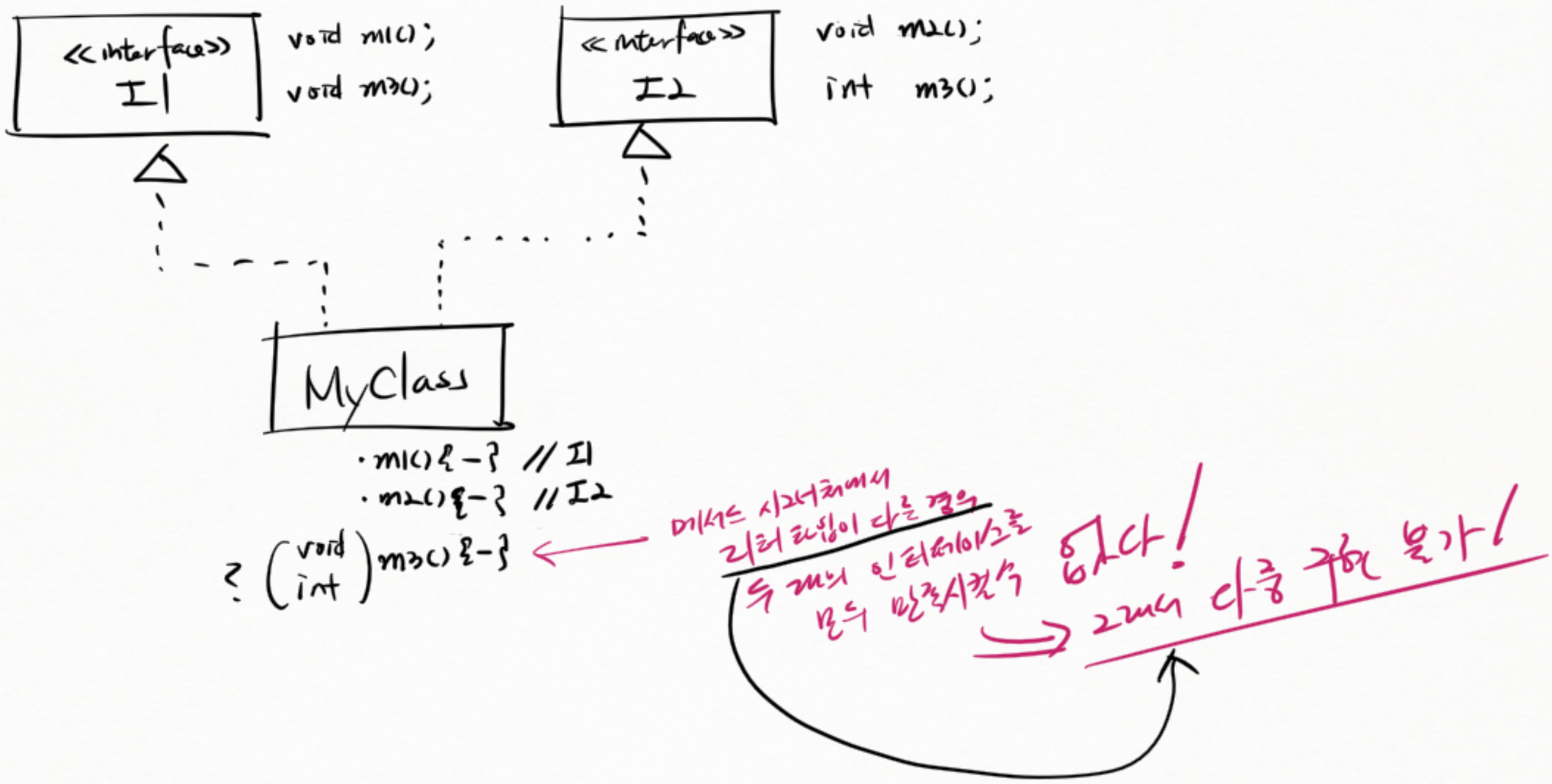
↓

이런 경우는 어떤 인터페이스를 상속받느냐에 따라 구현코드가 달라지기 때문에 혼동을 피하기위해 다중 상속을 허락하지 않는다

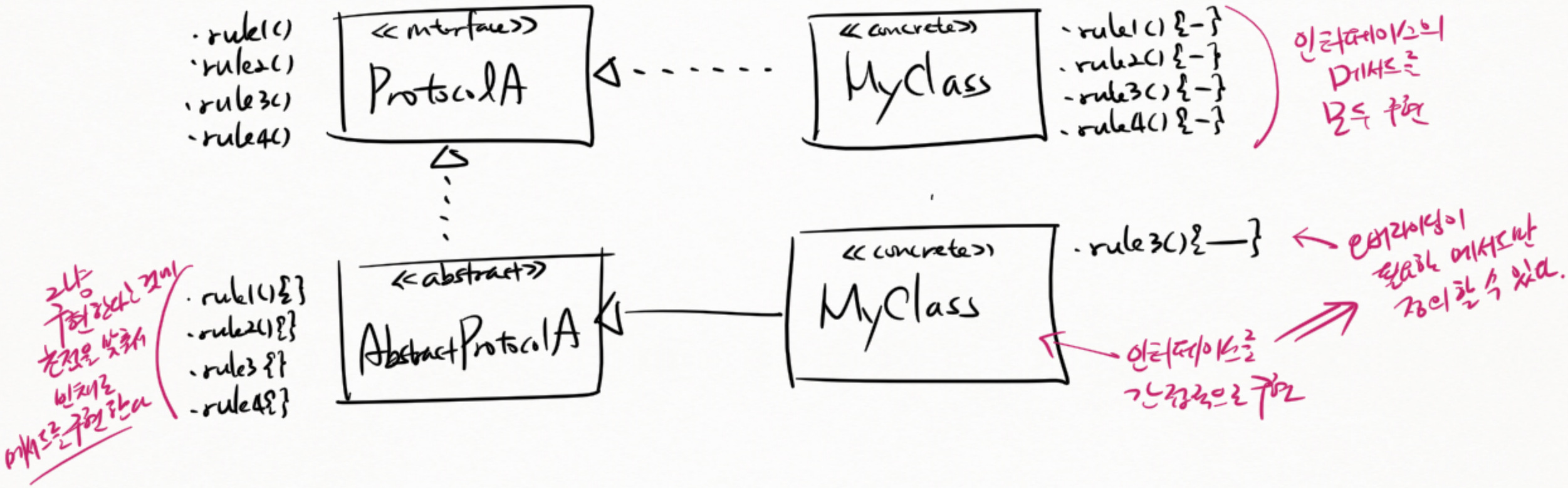
* 인터페이스 다중 구현


«interface»
I1
· m1()
· m3()
«interface»
I2
· m2()
· m3()
MyClass
· m1(){-}
· m2(){-}
· m3(){-}
I1, I2 둘 다 m3()의 시그니처가 같으면
한번만 구현하면 된다.

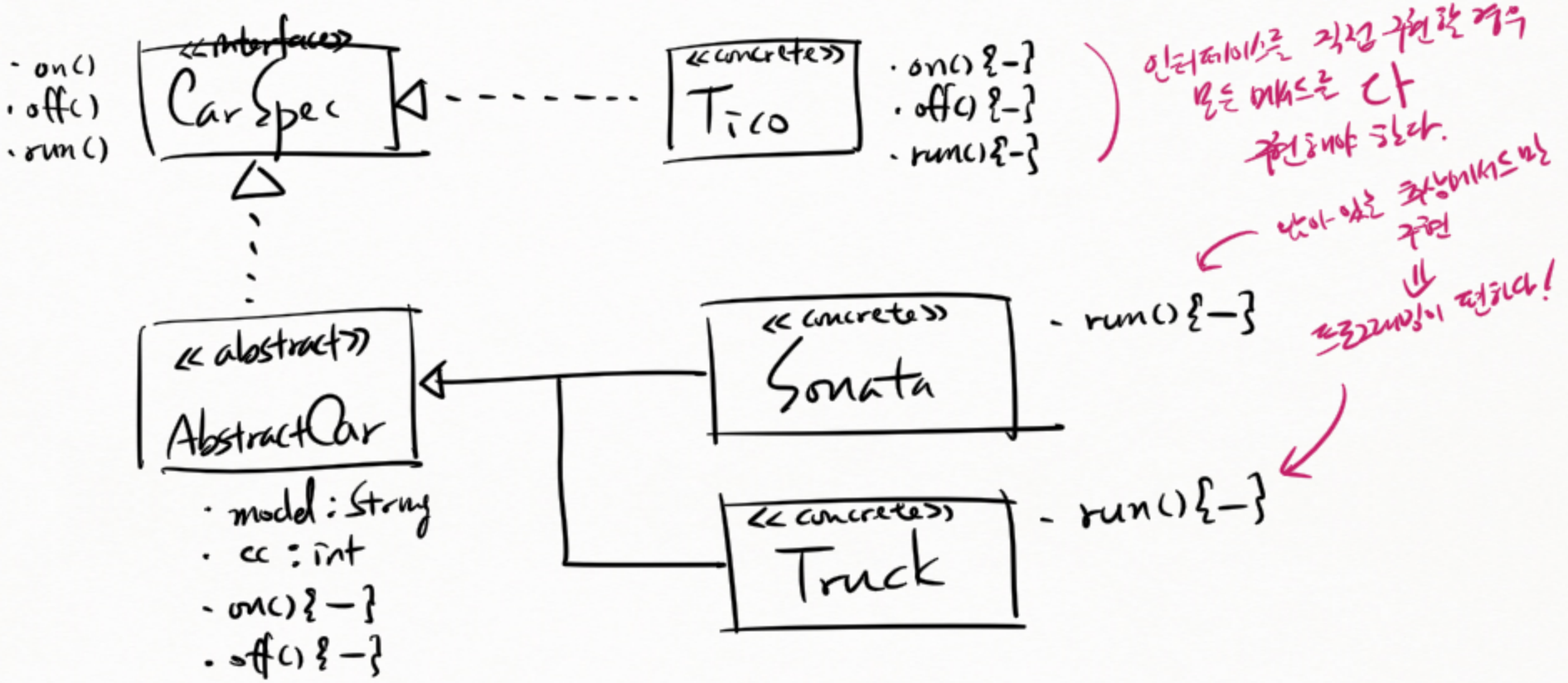
* 인터페이스 다중 구현 불가!



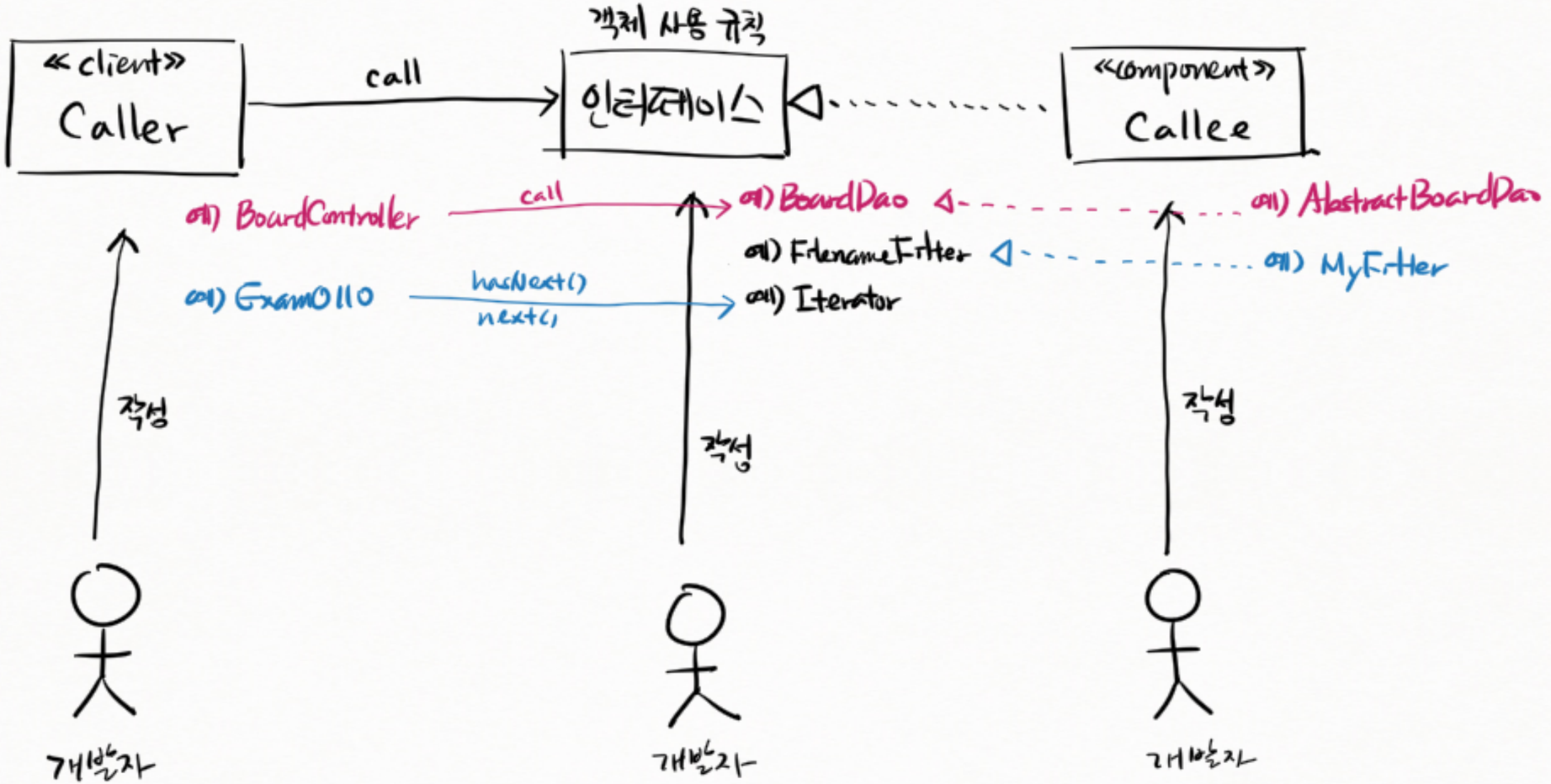
* 인터페이스 와 추상 클래스의 관계



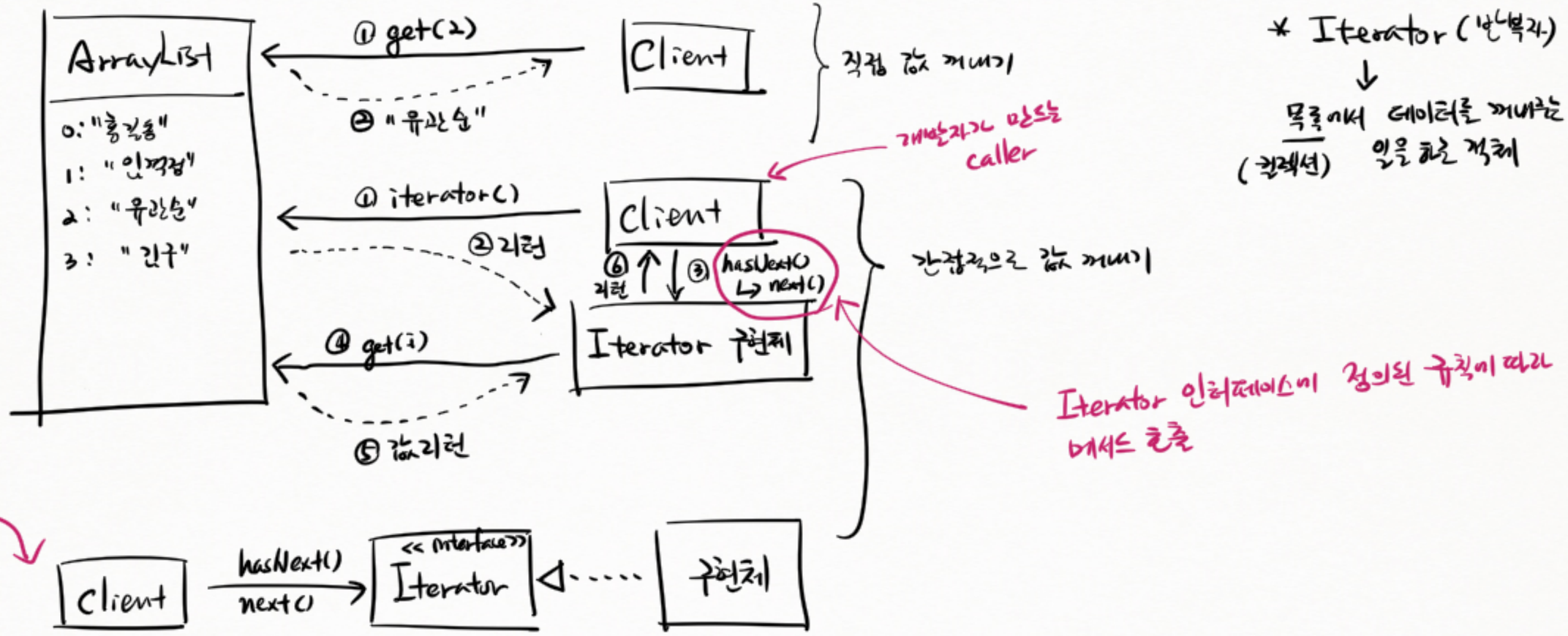
* 인터페이스와 추상클래스의 공통점 예)



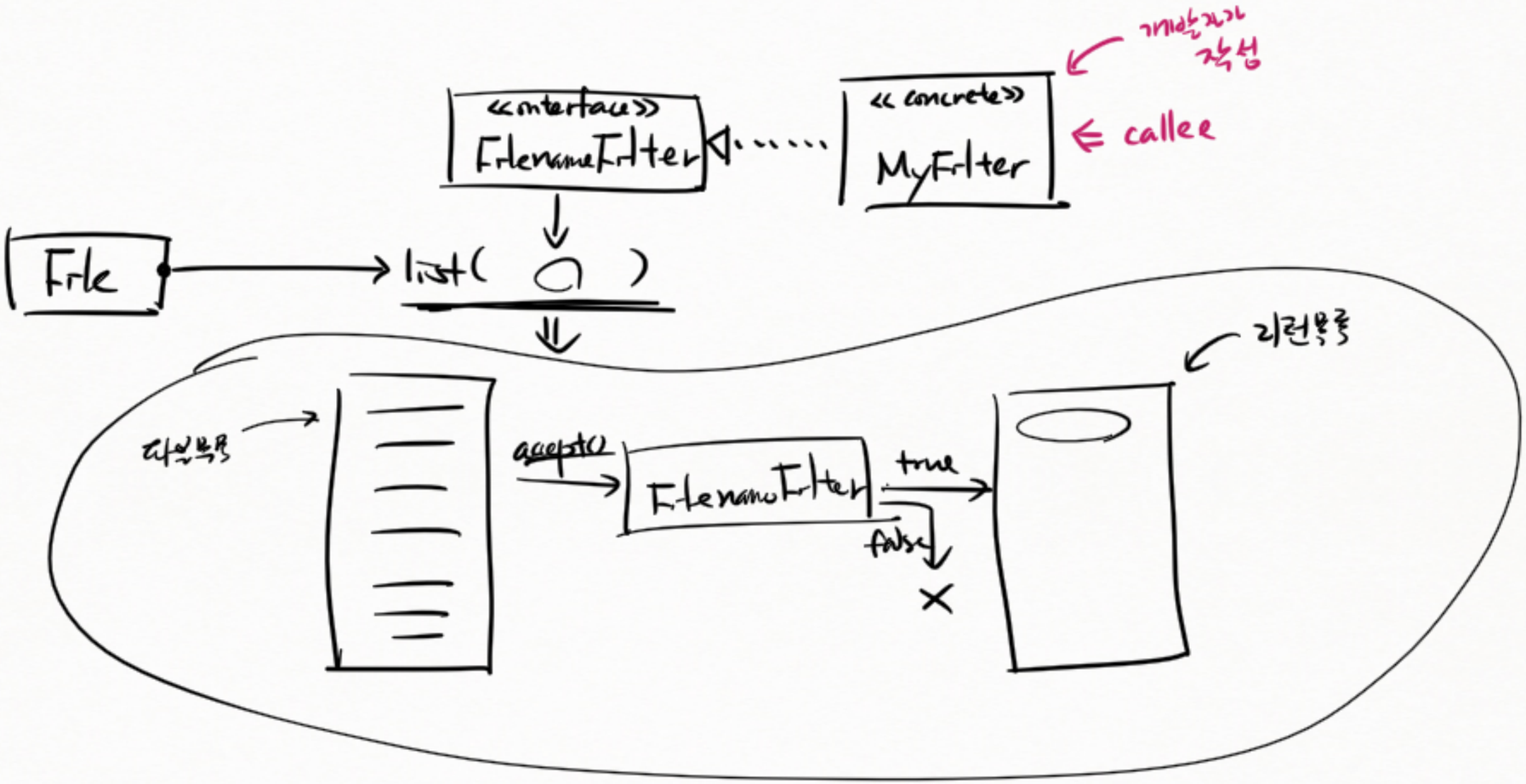
* 인터페이스와 개발자


객체 사용 규칙
«client»
Caller
call
인터페이스
«component»
Callee
예) BoardController
call
예) BoardDao
예) AbstractBoardDao
예) FilenameFilter
예) MyFilter
예) ExamOllo
hasNext()
next()
예) Iterator
작성
작성
작성
개발자
개발자
개발자

* Caller 개발 방법


ArrayList
0: "홍길동"
1: "신짱구"
2: "유관순"
3: "김구"
① get(2)
② "유관순"
Client
직접 값 꺼내기
개발자가 만드는 caller
① iterator()
② 리턴
Client
③ hasNext()
↳ next()
⑥ 리턴
Iterator 구현체
간접적으로 값 꺼내기
④ get(i)
⑤ 값리턴
Iterator 인터페이스에 정의된 규칙에 따라 메서드 호출
* Iterator (반복자)
↓
목록에서 데이터를 꺼내는 일을 하는 객체
(컬렉션)
caller를 작성
Client
hasNext()
next()
<< interface >>
Iterator
구현체

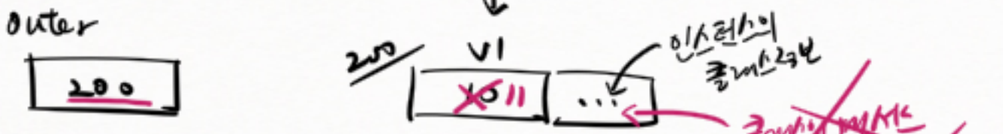
* callee 를 만드는 방법



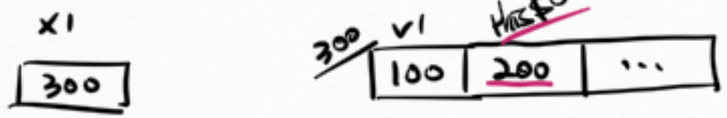
* inner 클래스의 객체
(com.eomcs.oop.ex11.c.Exam0230)

Method Area

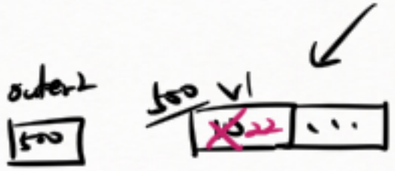
B3 outer = new B3()



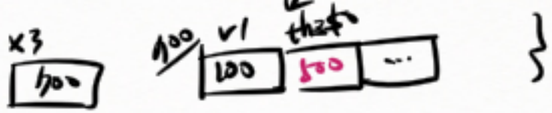
B3.X x1 = outer.new X()



B3 outer2 = new B3()

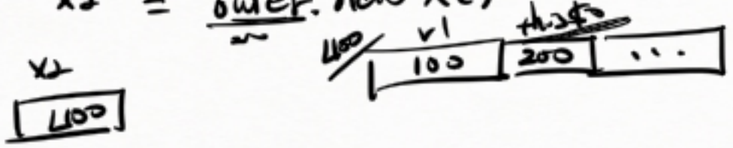


B3.X x3 = outer2.new X()



x1.test()

B3.X x2 = outer.new X()




```
void test() {
    this = 300
    v1 = 1000
}
```

출력 : v1 ⟶ 1000

출력 : this.v1 ⟶ 100

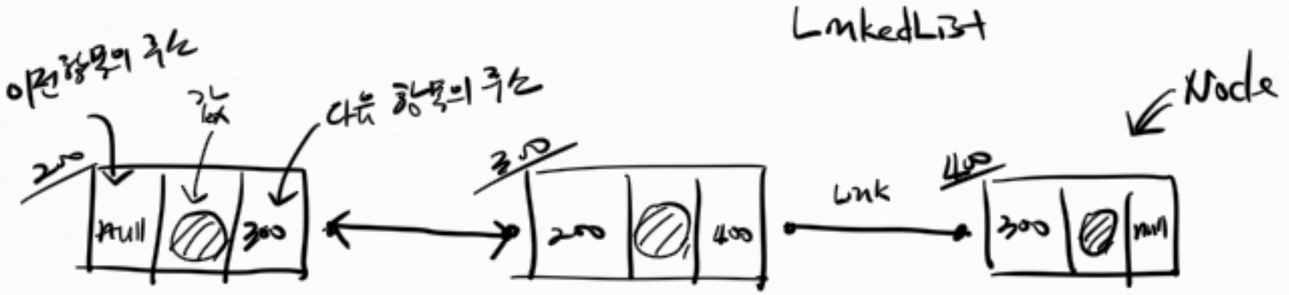
출력 : B3.this.v1 ⟶ 11

B3.this

this$0

200

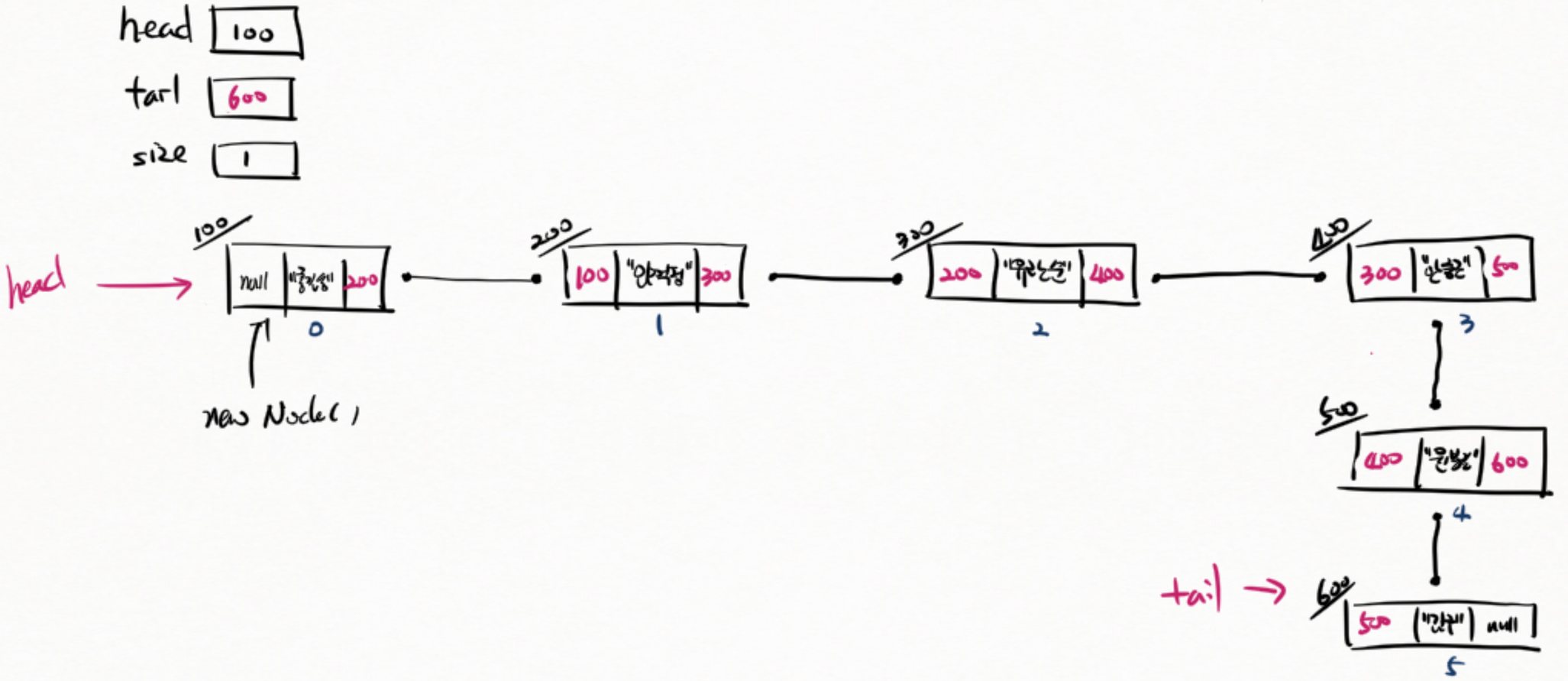
* LinkedList



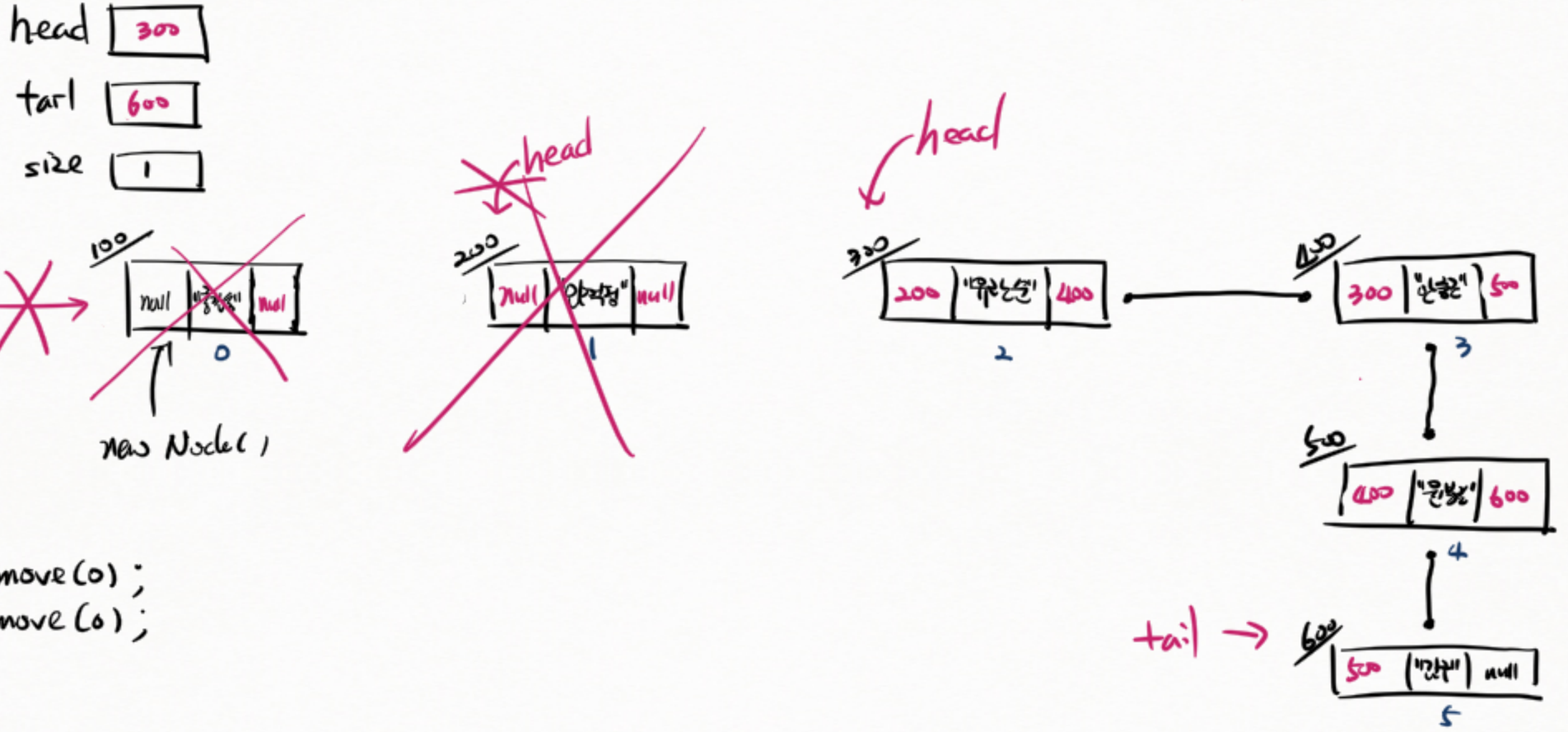
head 200

tail 400

* LinkedList - 값 추가

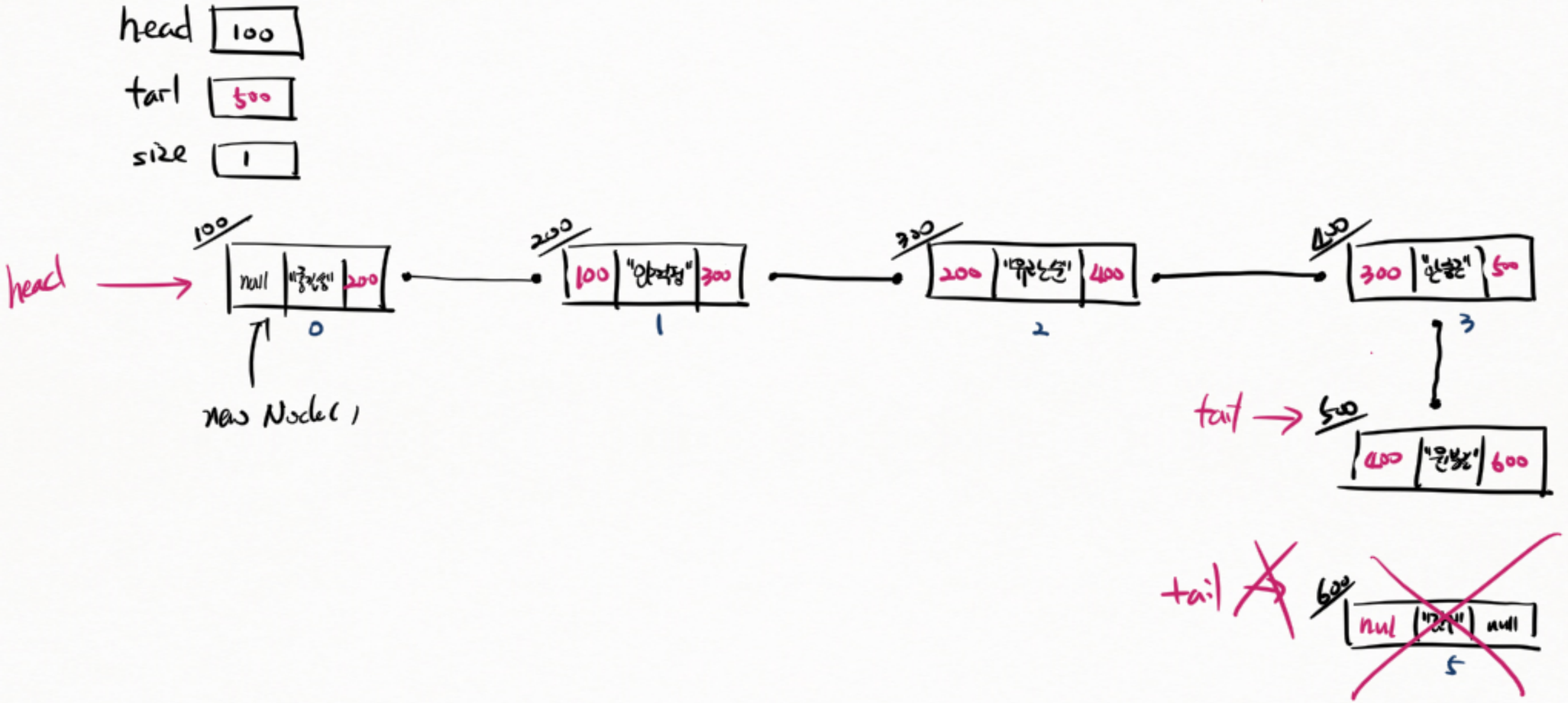


* LinkedList - 값 삭제 (맨 앞 값)

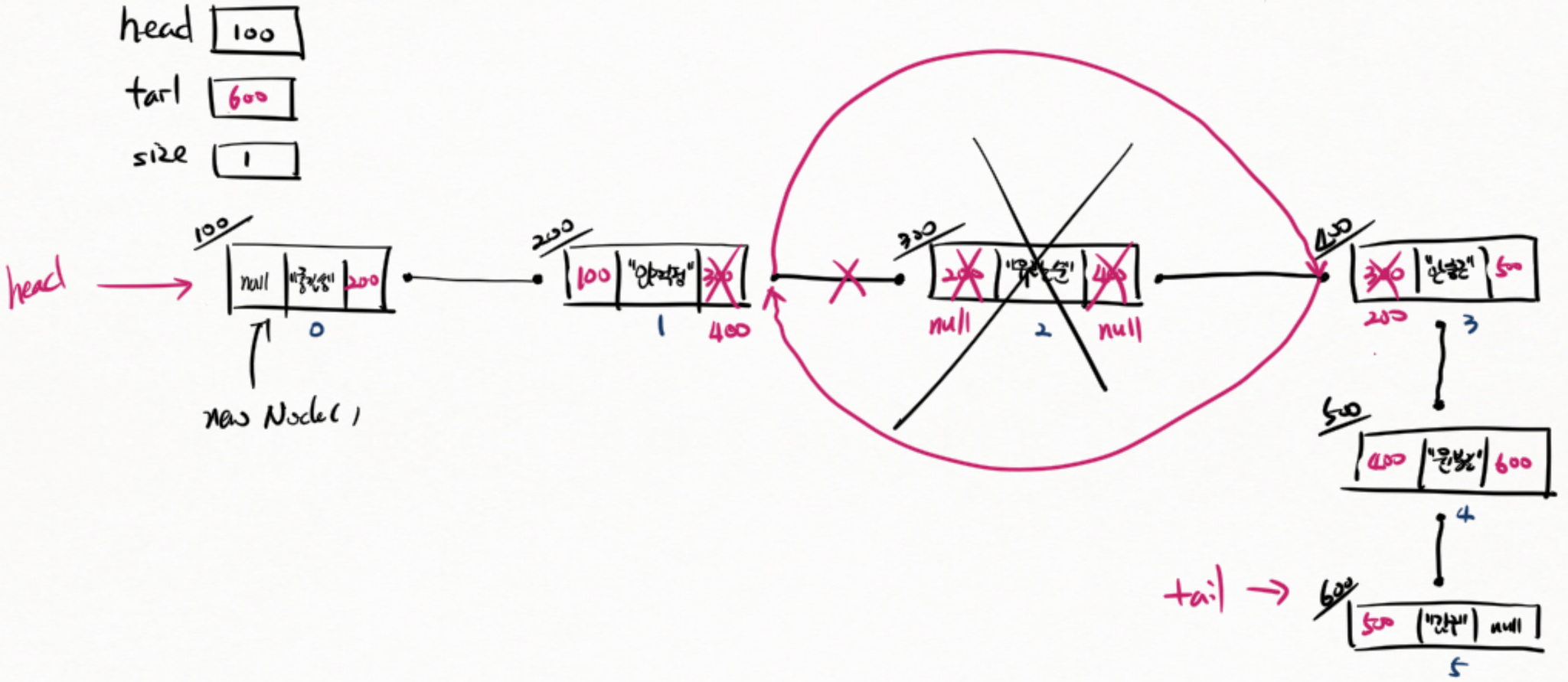



```
remove(0);
remove(0);
```

* LinkedList - 값 삭제 (뒤 위치)



* LinkedList - 값 삭제 (중간 값)



* LinkedList - 값 삽입