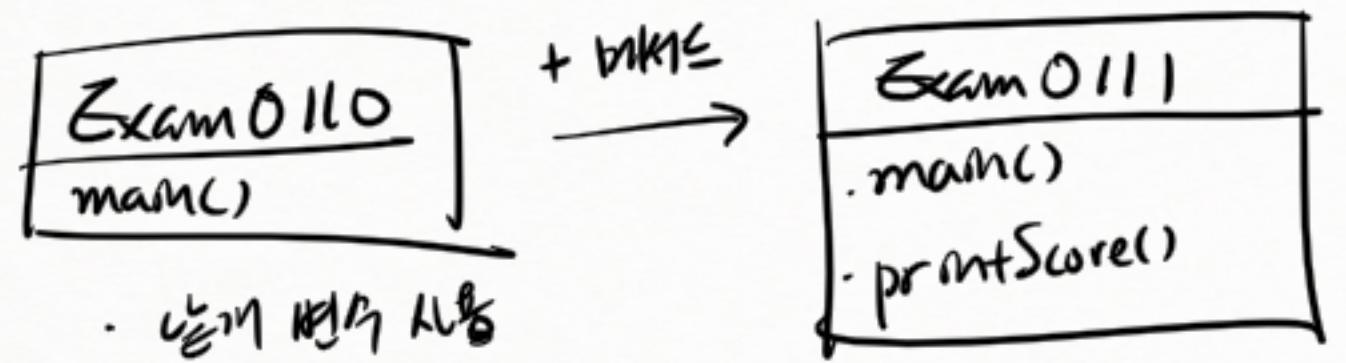


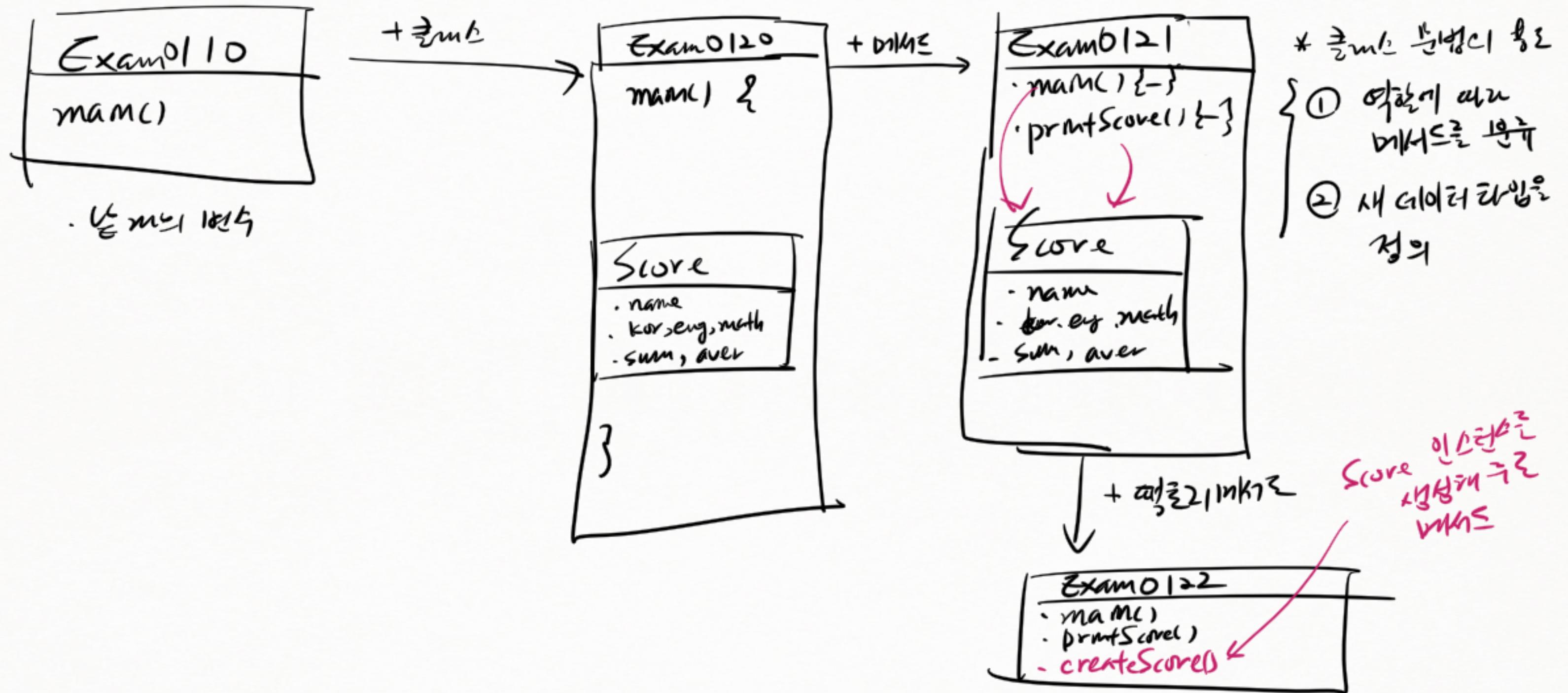
* 재미스 문법 활용 예



· 놓기 허용 사용

- 인라인 문법 활용
↳ 자바에서 사용

- ↓
 - ✓ 정복 접근하기
 - ↓
 - 코드 처리율 ↑
 - ✓ 유지보수가 쉬워짐



* 데이터를 101로 → 여러 모의 인스턴스를 다루기

Score s1, s2, s3

s1
200

s2
300

s3
1100

200	name	kor	eng	math	sum	aver
200	○	○	○	○	○	○
300	C	○	○	○	○	○
1100	○	○	○	C	○	○

s1. name = "—"'

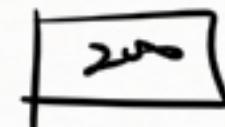
s1. kor = 100 -

:

* 예외처리 10주차

Score[] arr = new Score[3];

arr



null?
- null이면 오류!
- 접근할 수가 0으로 설정되었을 때.

arr[0] = new Score();

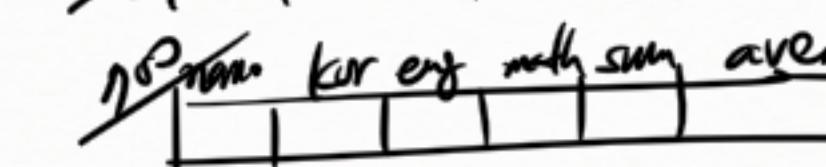
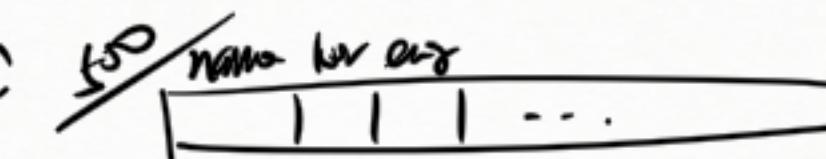
* 예외 처리 ← 자동으로 null로 초기화 된다
* 접근할 수가 초기화 되어 있다.

arr[1] = new Score();

arr[2] = new Score();

arr[3] = new Score();

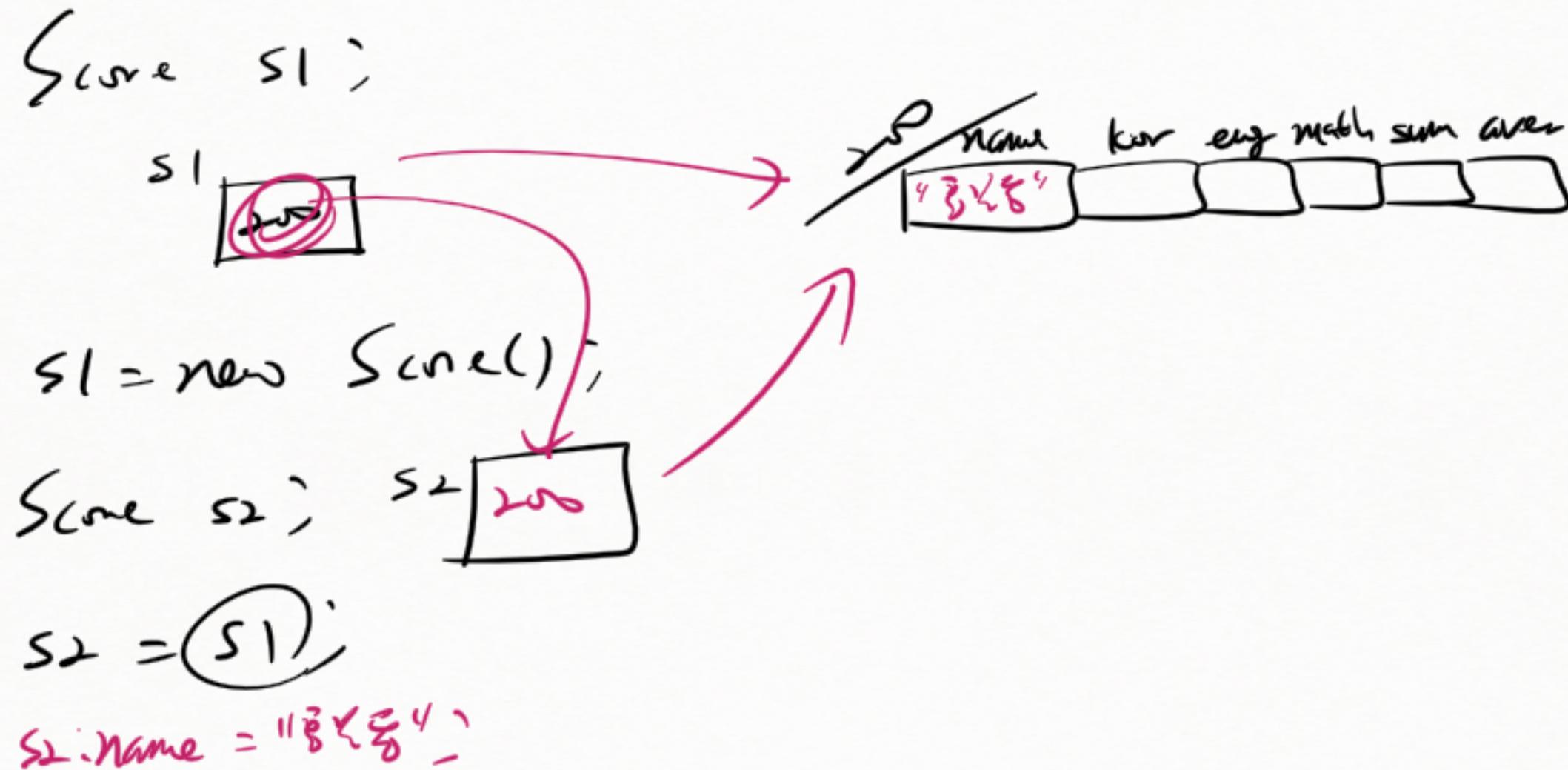
* ArrayIndexOutOfBoundsException



new Score();

Score ⇒ null 선언 했을 때
Heap에 초기화 되어 있다.
기억해두면 좋다.

* 리터럴과 인스턴스



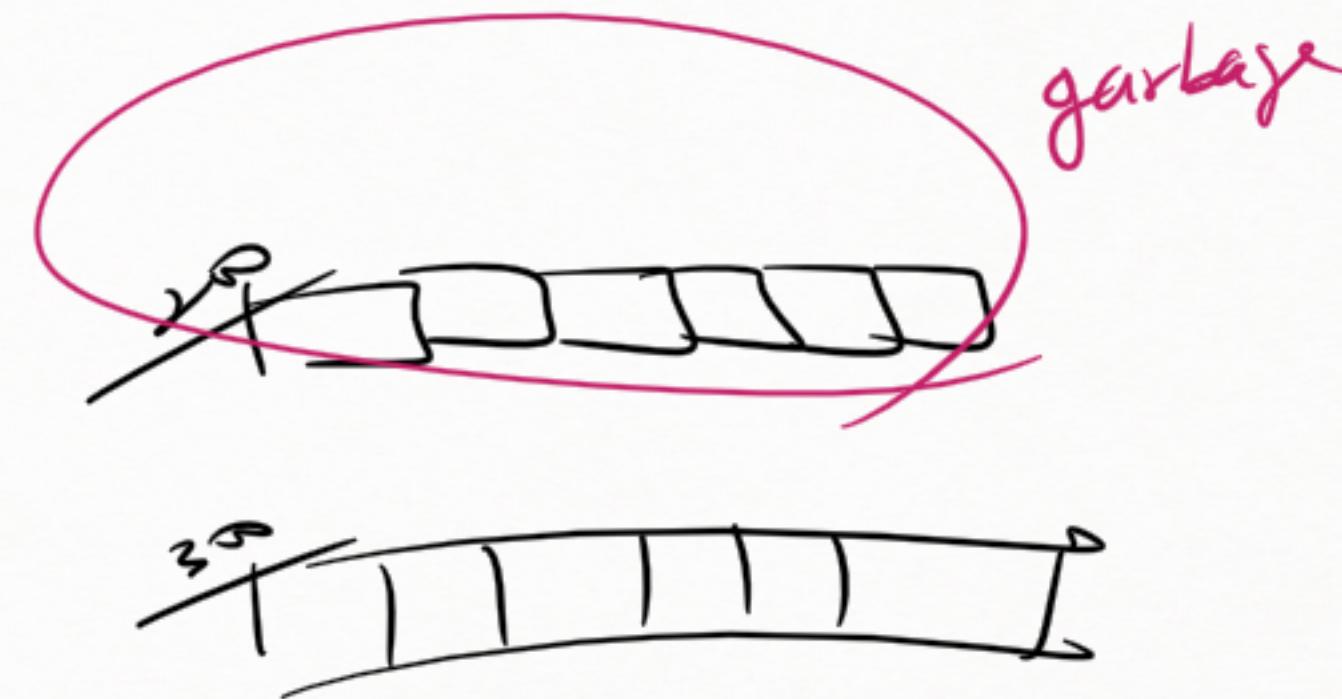
* 7110121 (garbage)

Score s1;



s1 = new Score();

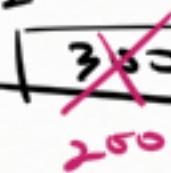
s1 = new Score();

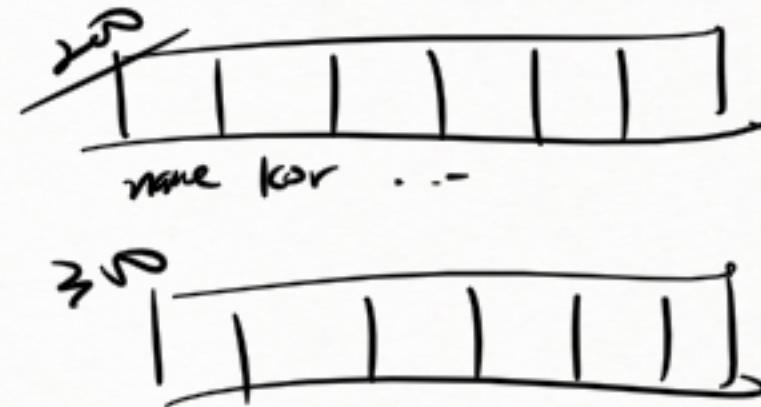


* 리터럴은 카운트와 관계

```
Score s1, s2;  
s1 = new Score();  
s2 = new Score();  
s2 = s1;
```

s1

s2




JVM이 처리

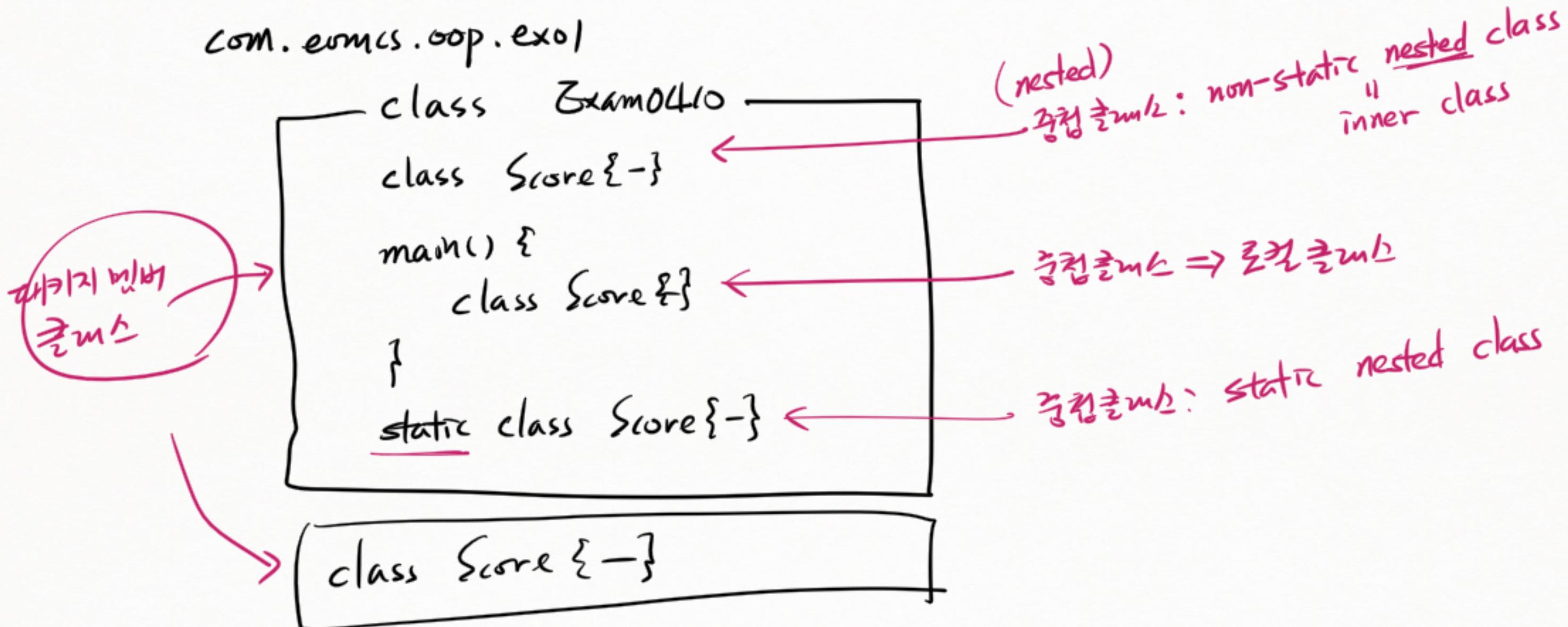
리터럴은 카운트 관계

리터널은 카운트가
0인 경우
"garbage" 가
된다.

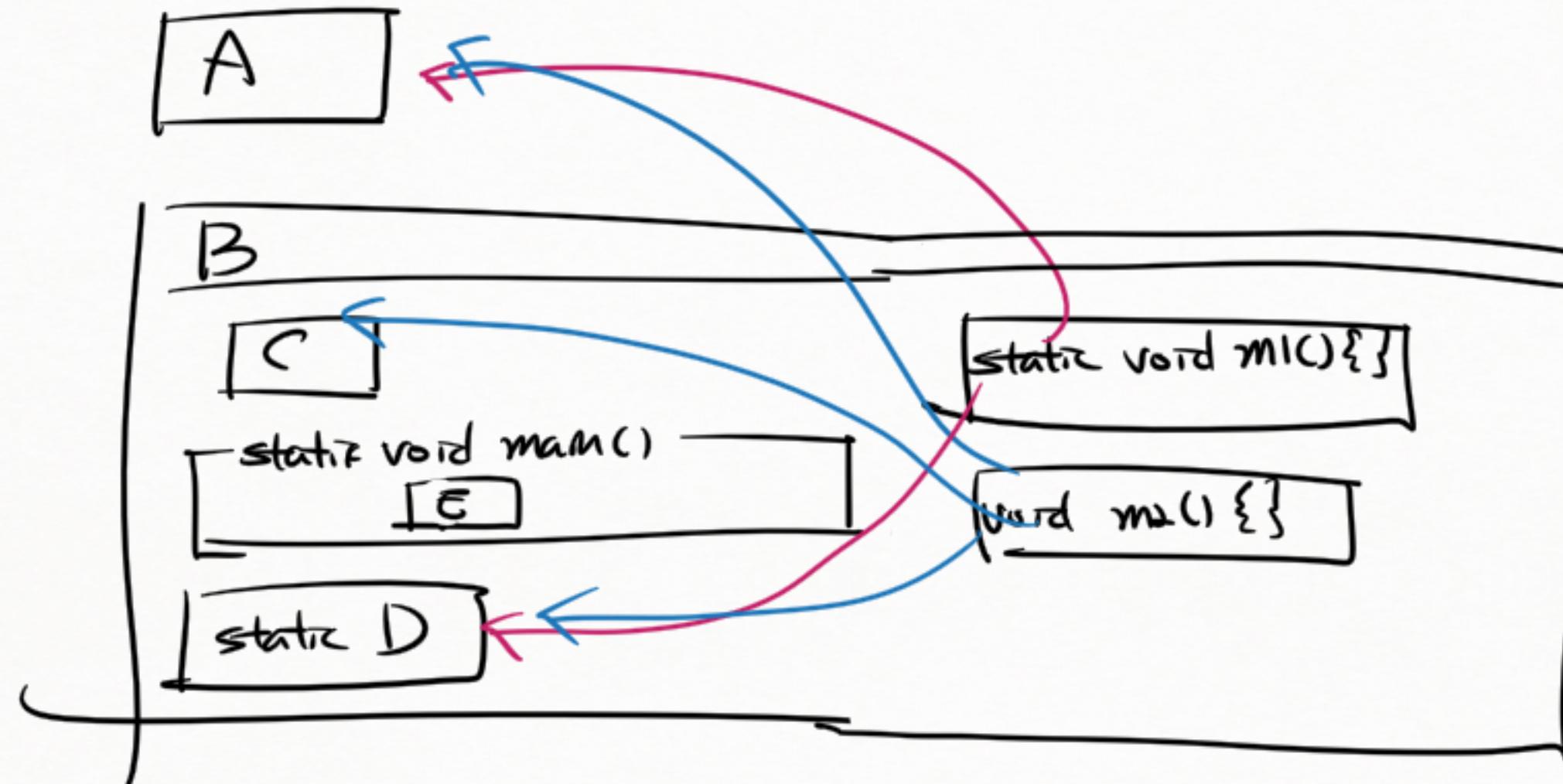
인스턴스	참조 횟수
200	X 2
300	X 0

* 클래스 구조

com.eunics.oop.ex01

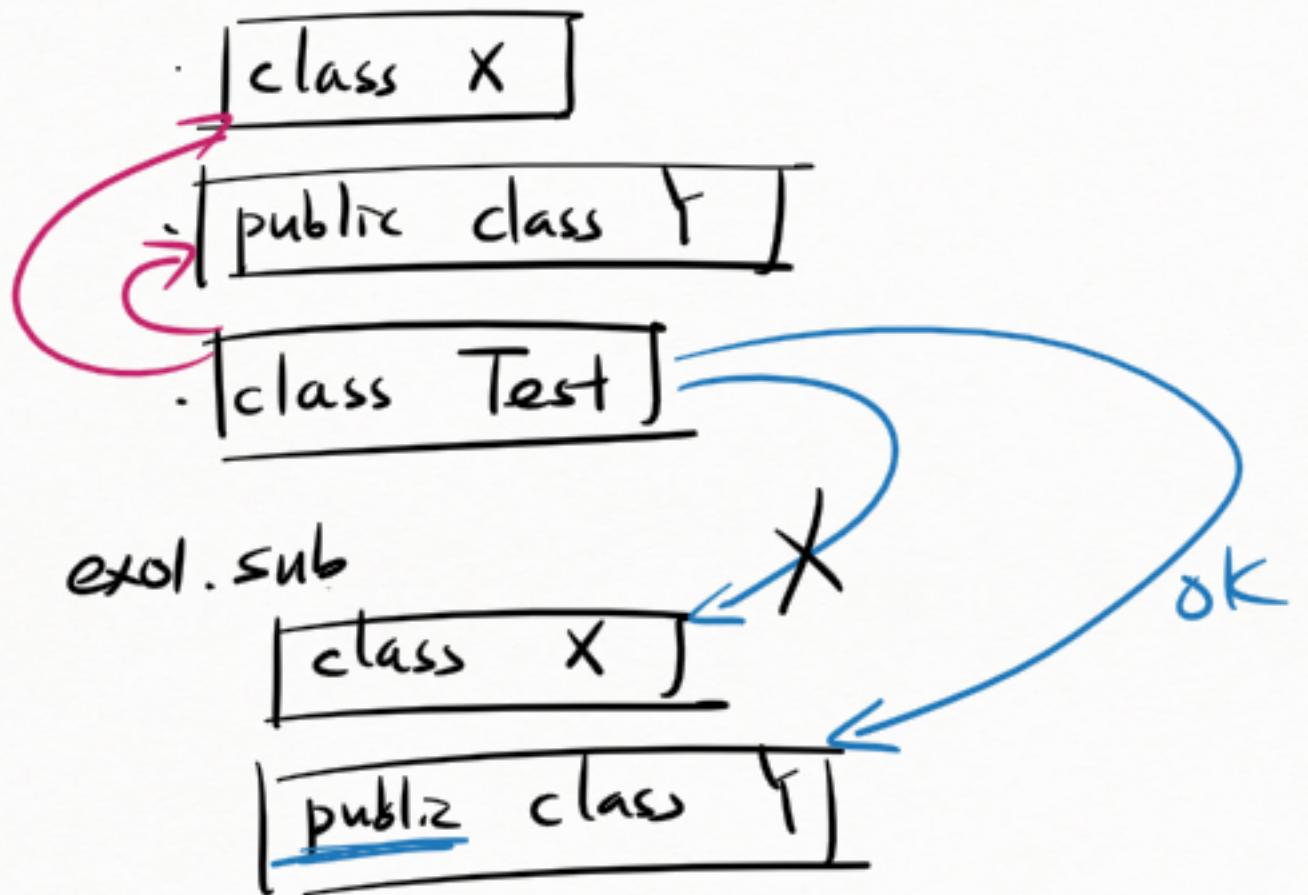


* static member non-static member



* 공개 멤버 필드

ex01



ex01.sub

* 클래스 문법의 활용 예: ① 사용자 정의 데이터 타입을 만드는 용도
User-defined Data Type
 개발자



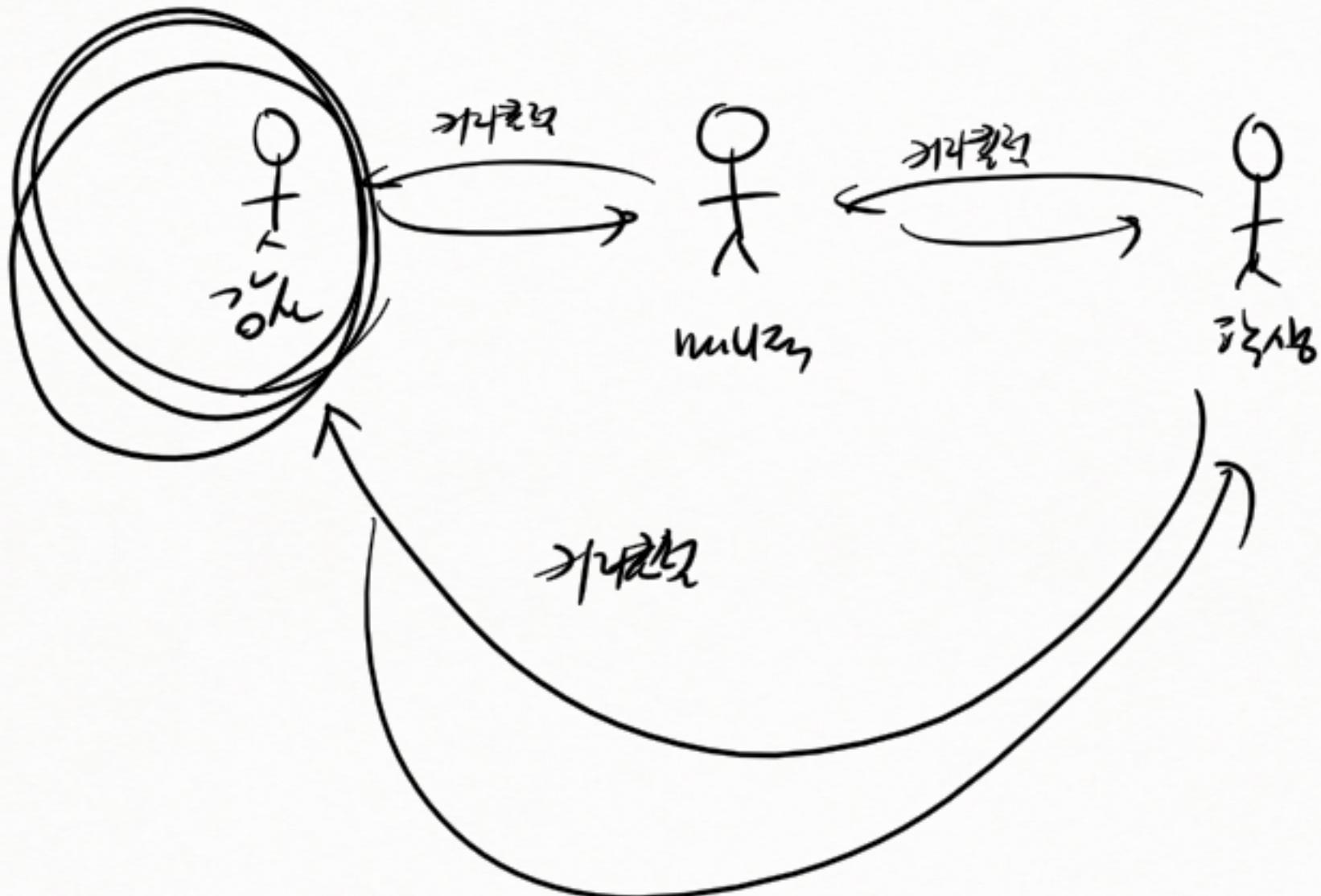
s1.name = "홍길동"

* optimizing(최적화) vs refactoring(재구조화)

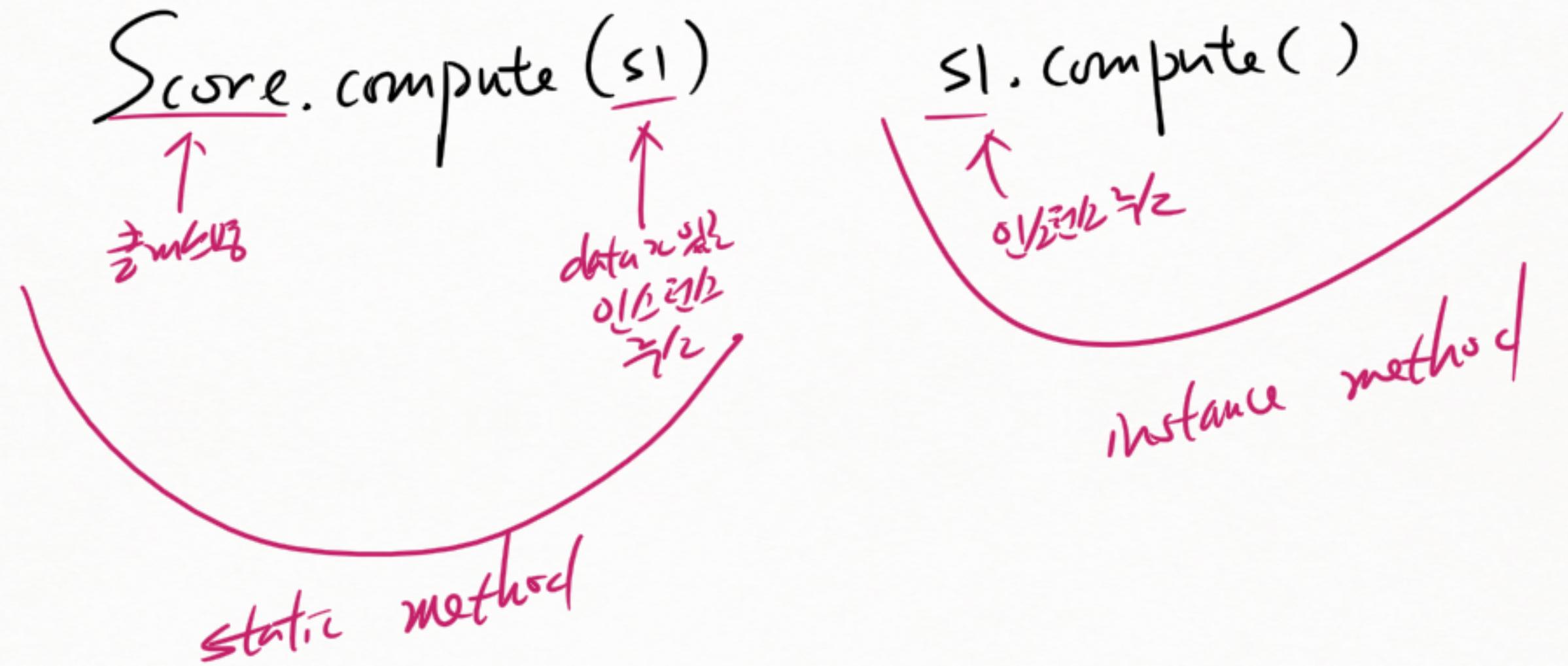
- ↓
- 속도↑
- 유지보수 힘들기↑
- 디버깅↓

- ① s1 리퍼런스에 저장된 주소로 쳐다보면서 해당 인스턴스의 name 변수 —
- ② s1 리퍼런스가 가리키는 인스턴스의 name 변수 —
- ③ s1 인스턴스의 name 변수 —
- ④ s1 객체의 name 변수 (필드)
- ⑤ s1의 name 필드 (변수)

✗ GRASP : 훌륭스며 책임 있는 의사 결정을 +



* static 데일리에 인스턴스 변수



* 인스턴스 메서드와 인자

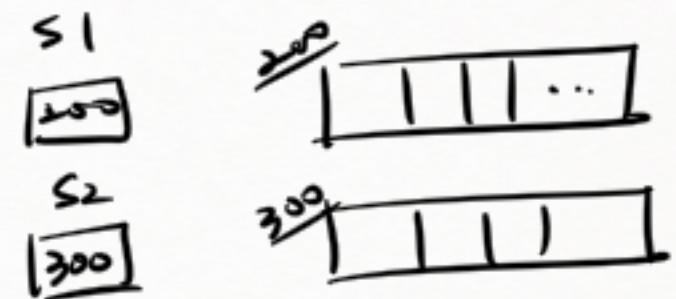
j ++ ;
 operand
 (인자)
 operator (연산자)

j ++;

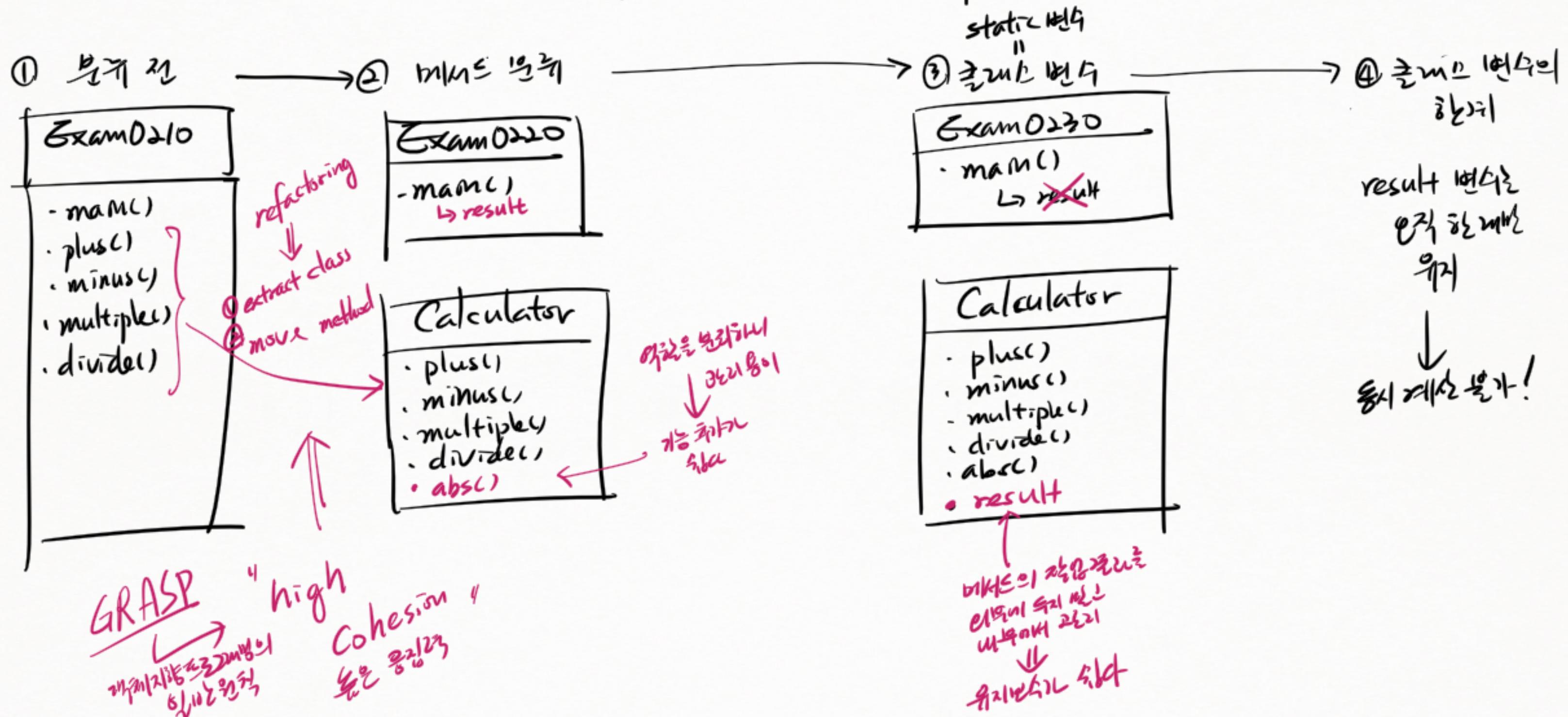
Score s1, s2 ;
 s1 = new Score();
 s2 = new Score()

인자
 ↓
s1. compute()
 ↑
 operand
 ↑
 operator

s2. compute()

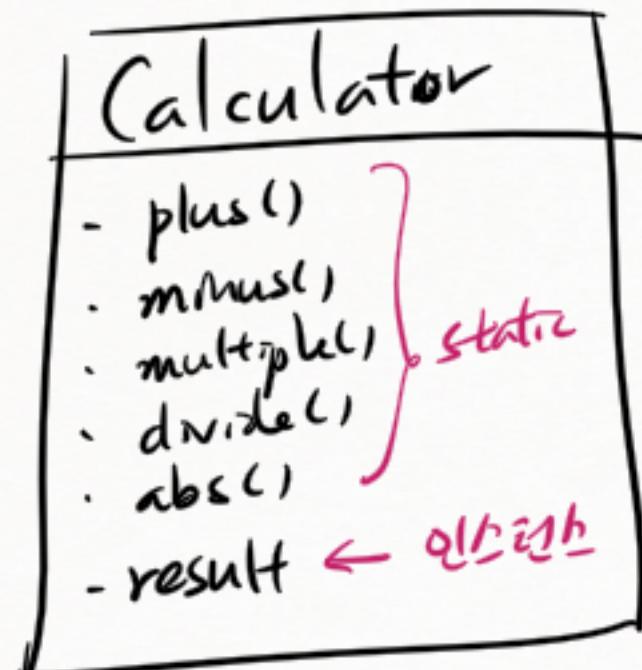
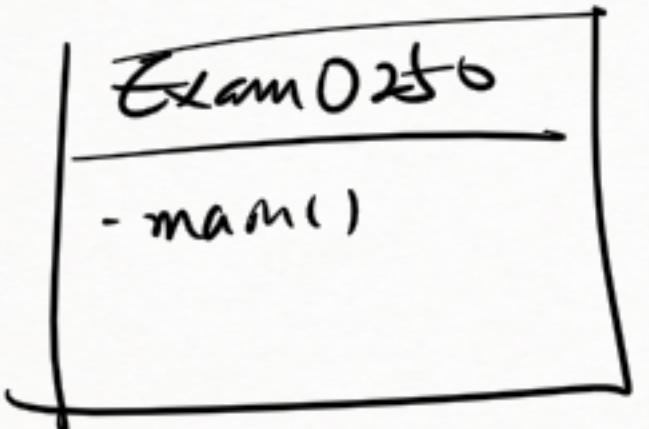


* 3단계 분기점: MMR을 끝난 분기점이



* 인스턴스 멤버 함수: 인스턴스마다 다른 결과를 반환

→ ⑤ 인스턴스 멤버 변수



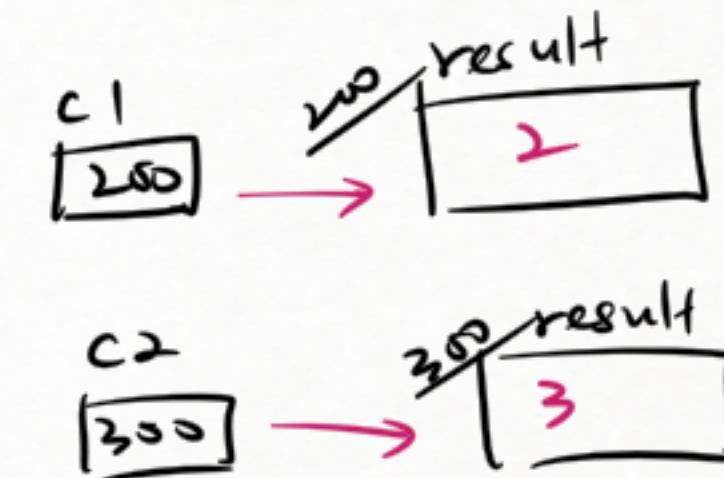
```
Calculator c1 = new Calculator();
Calculator c2 = new Calculator();
```

```
Calculator.plus(c1, 2);
```

```
Calculator.plus(c2, 3);
```

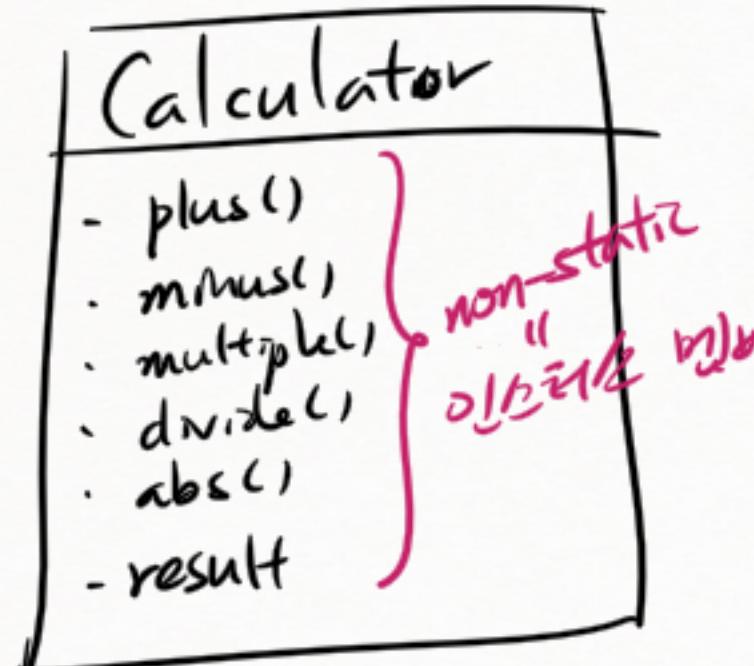
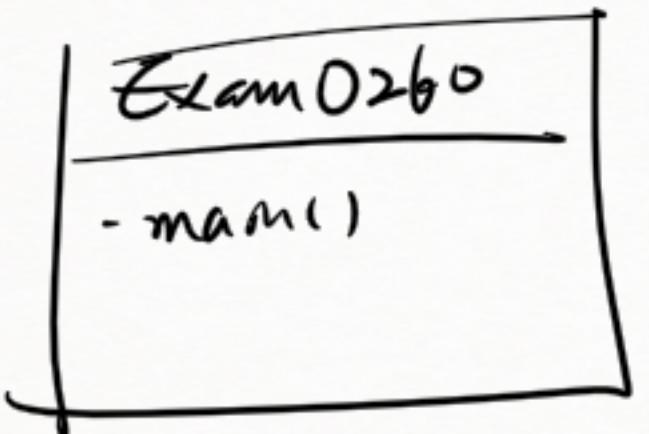
:

↑
각 인스턴스는 자신의
result 멤버 변수를
다른 값으로 초기화



* 인스턴스 멤버 변수: 인스턴스마다 다른 값을 갖다

→ ⑥ 인스턴스 변수

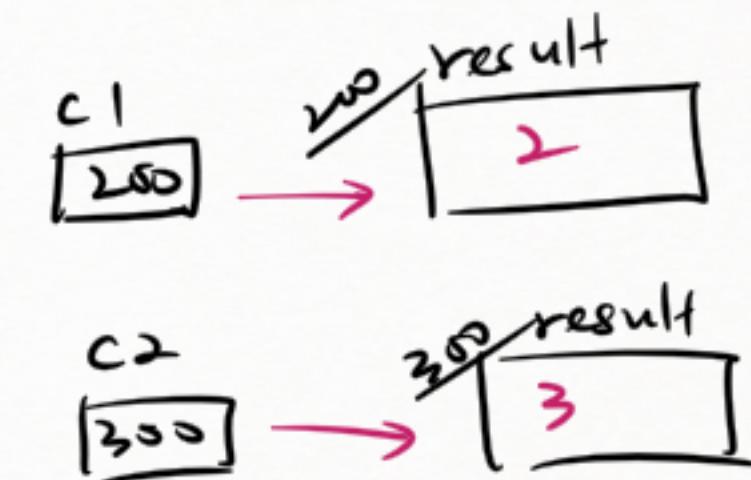


Calculator c1 = new Calculator();
Calculator c2 = new Calculator();

c1.plus(2);
c2.plus(3);

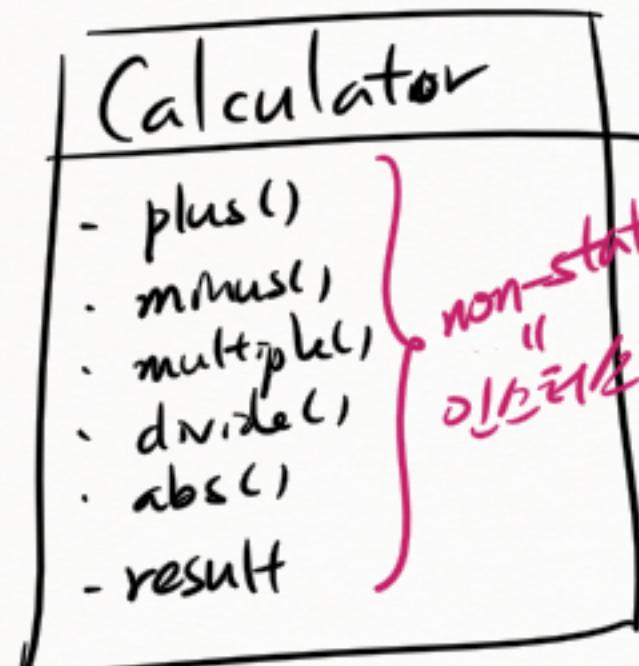
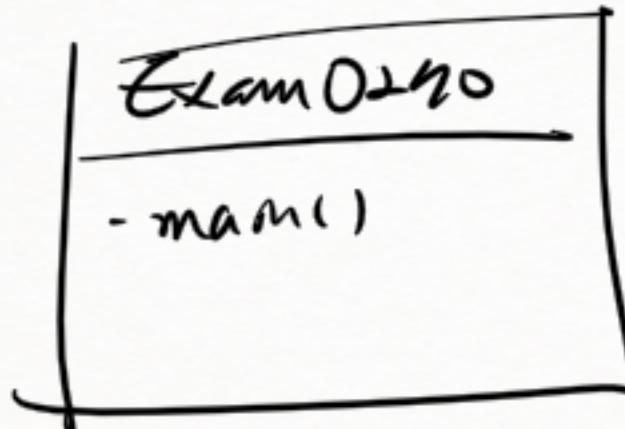
인스턴스 변수

인스턴스 변수
c1.plus(2)
+ 2

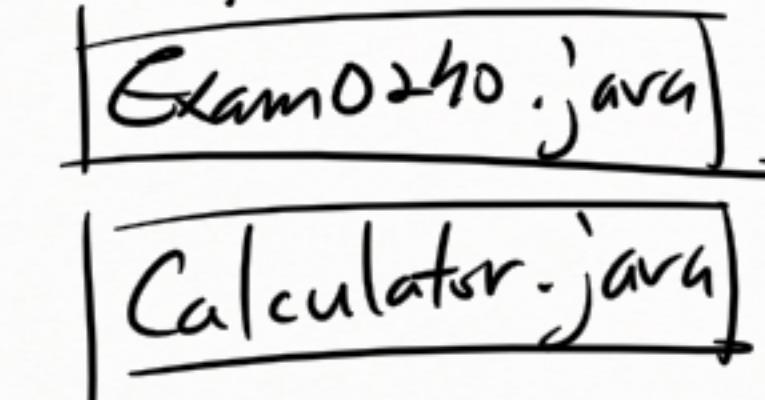


* \Rightarrow ml² ပုံမှန် ရှိခဲ့ပါ : မြတ်သွေးကို ဖြန့်မျက်

→ ① အော်လုပ်မှု \Rightarrow ml² → ② အော်လုပ်



com.eomcs.oop.ex02.



com.eomcs.oop.ex02.Exam0240

com.eomcs.oop.ex02.util.Calculator

import com.eomcs.oop.ex02.util.Calculator;

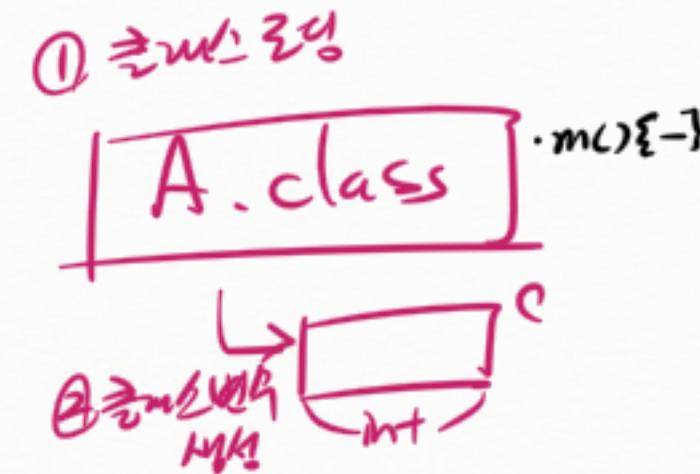
↑
 \Rightarrow ပုံမှန် ပုံမှန် ပုံမှန် ပုံမှန်

* static 멤버 와 인스턴스 멤버

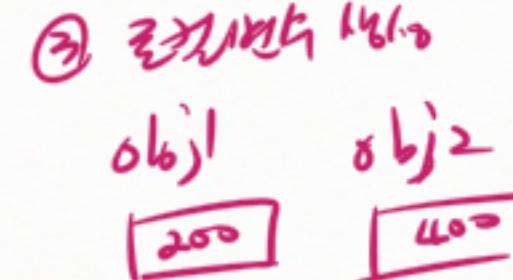
class A {
 인스턴스 멤버
 int a;
 int b;
 static int c;
 void m() {}
}

A obj1 = new A();
 A obj2 = new A();

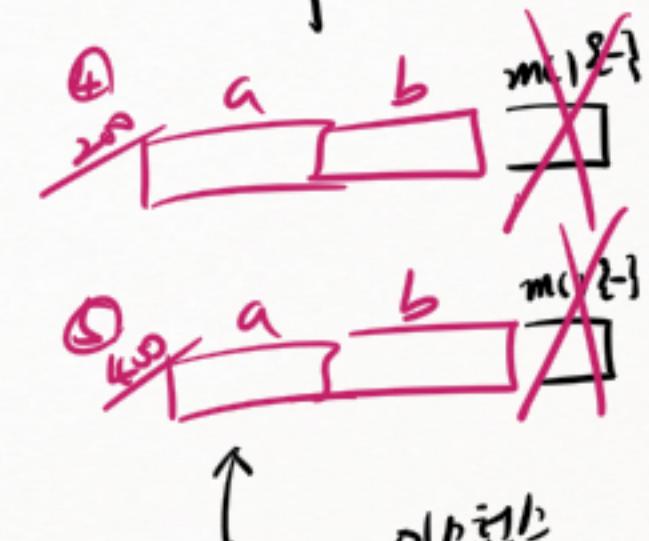
Method Area



JVM Stack



Heap



↑
 A 클래스의 인스턴스

인스턴스의 필드가 2nd.

* 클래스 멤버와 인스턴스 멤버

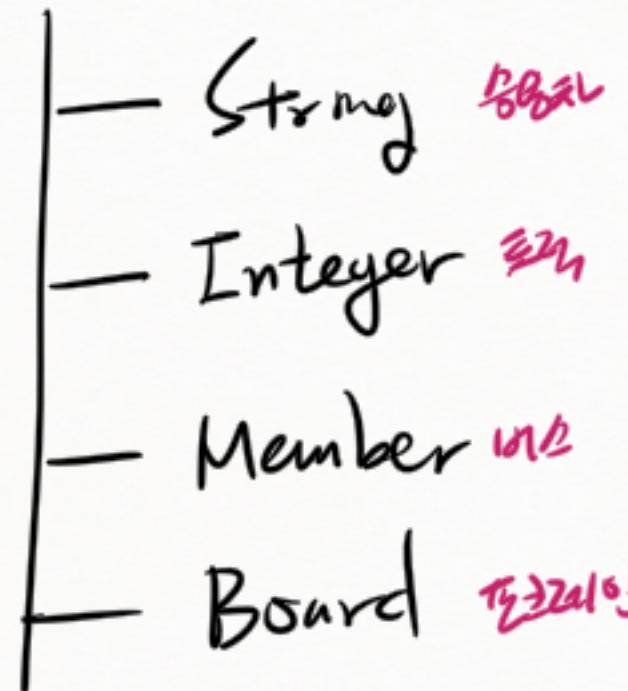
```
class Calculator {  
    int result;  
    void plus(int v){  
        result += v;  
    }  
    ...  
}
```

클래스 멤버
인스턴스 멤버를 다룰 때
연산자!
(인스턴스)

* Object 클래스
↳ 자바의 최상위 클래스

java.lang.

Object 사용

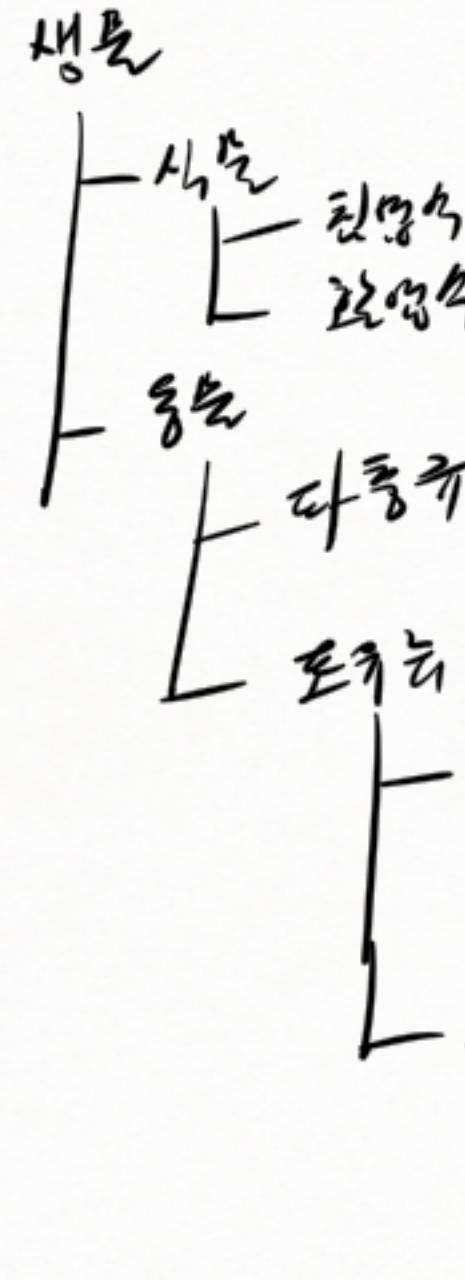


자바의 모든 것은 Object의
자식 클래스이다.

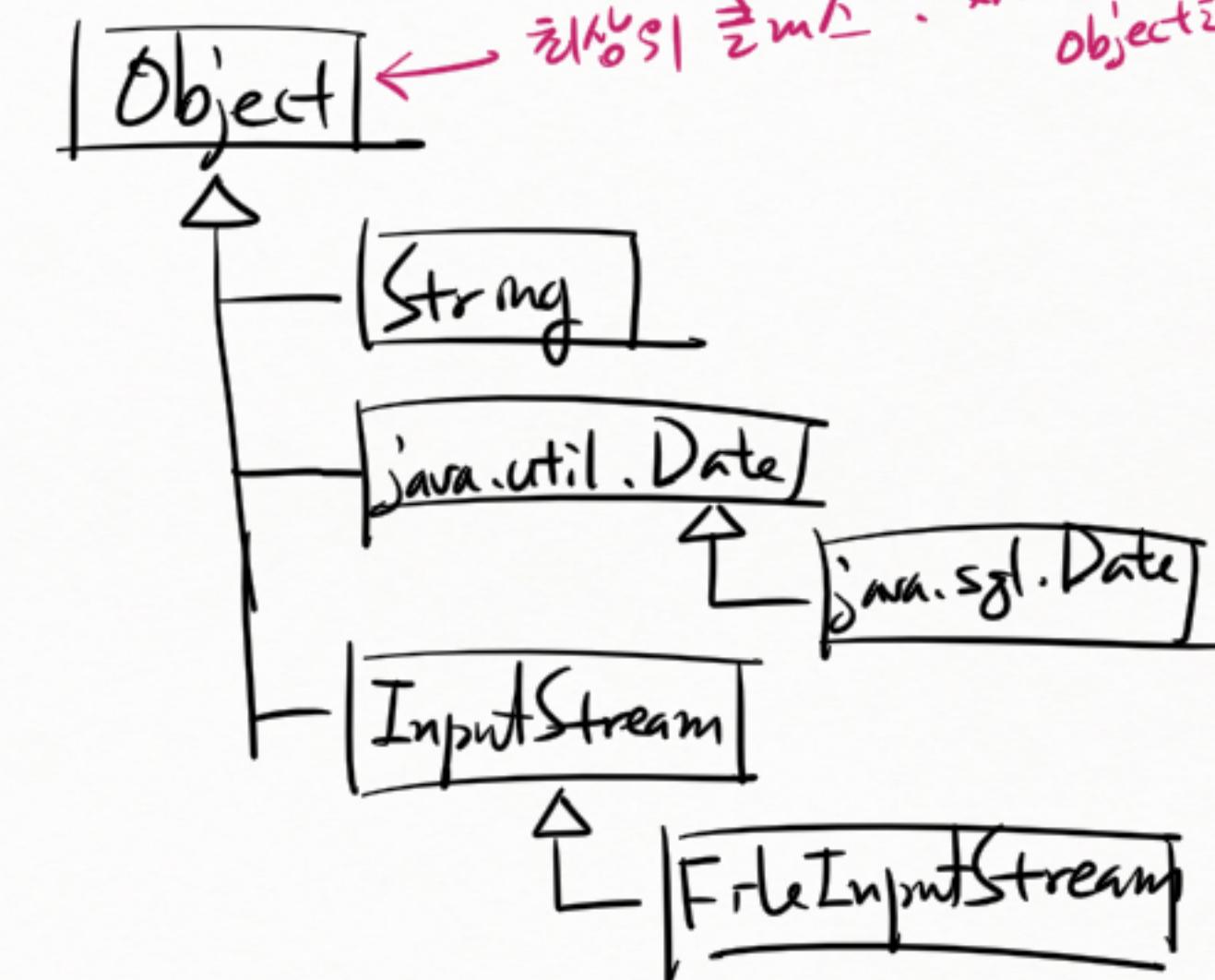
자식 (sub)

* 물류와 출판 분야

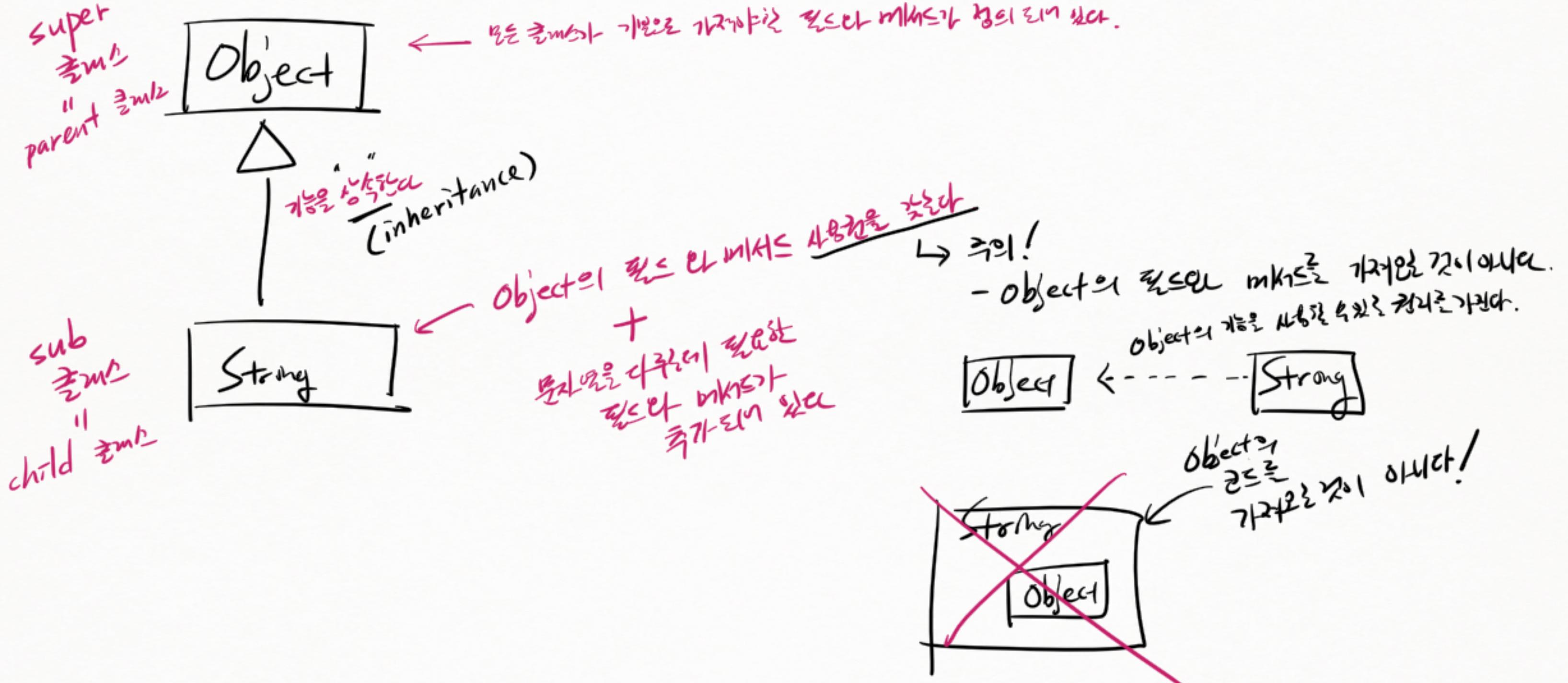
1877
MBS



제작된 제품은 물건, 화물
구성을 구성하는 있다.



* 상위 클래스와 하위 클래스 : 같은 공유를 위한!



* 향수 예법: 근드 중의 향법 중 하나.



* 상속 문법과 다형성

- ↳ 대형화 범위 *
- ↳ 오버로딩 (overloading)
- ↳ 오버라이딩 (overriding) *

Car c;

```
c = new Car();
c = new Sedan();
c = new Truck();
c = new Trailer();
c = new Dump();
```

↳ 대형화 범위

Truck t;

```
t = new Car();
t = new Sedan();
t = new Truck();
t = new Trailer();
t = new Dump();
```

상속 | 출현 | 리턴값이
하나 | 출현 | 이전은 가능
하지만 수 있다
 ↓
구현 | 출현 | 가능
가지 않을 수 있다.

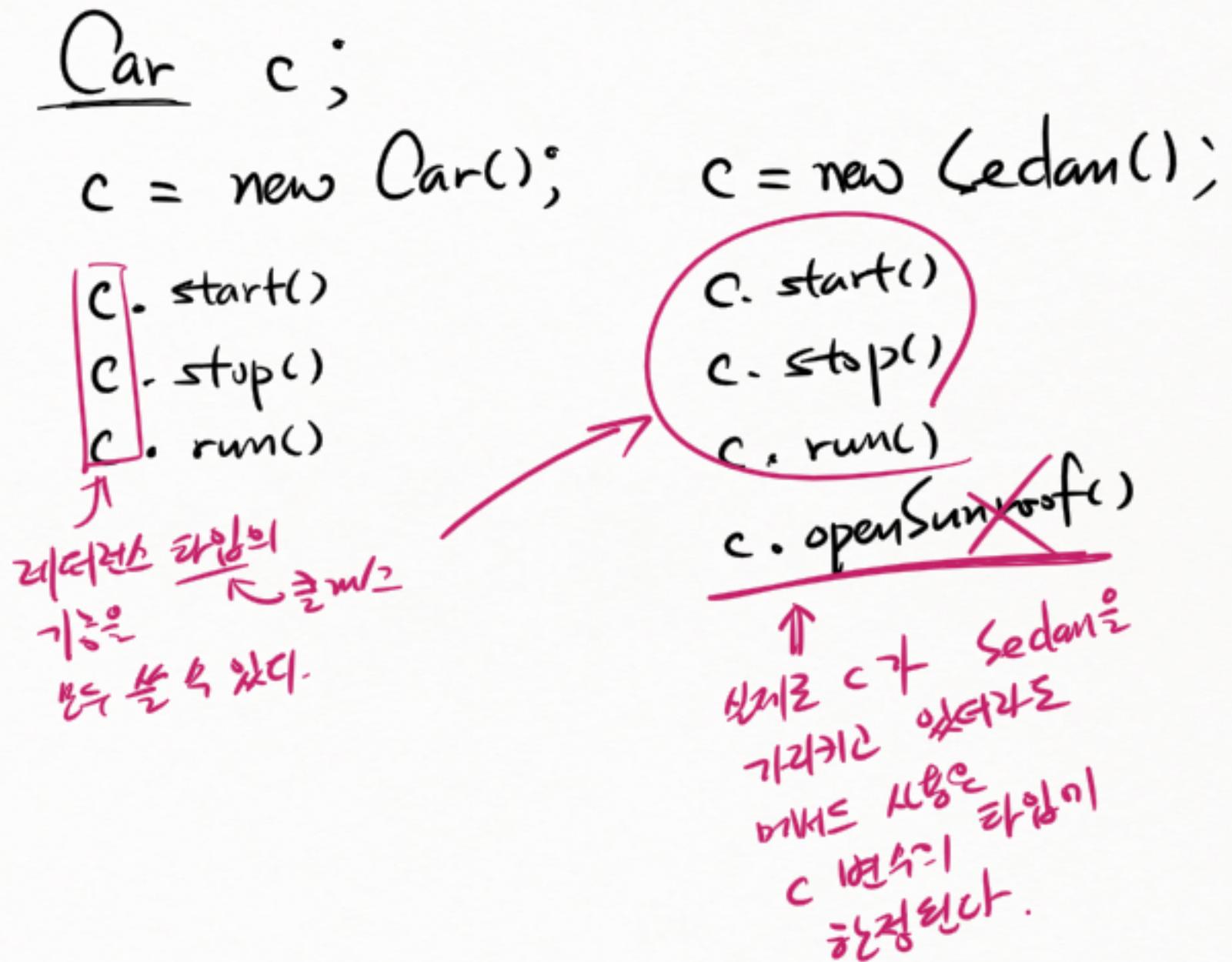
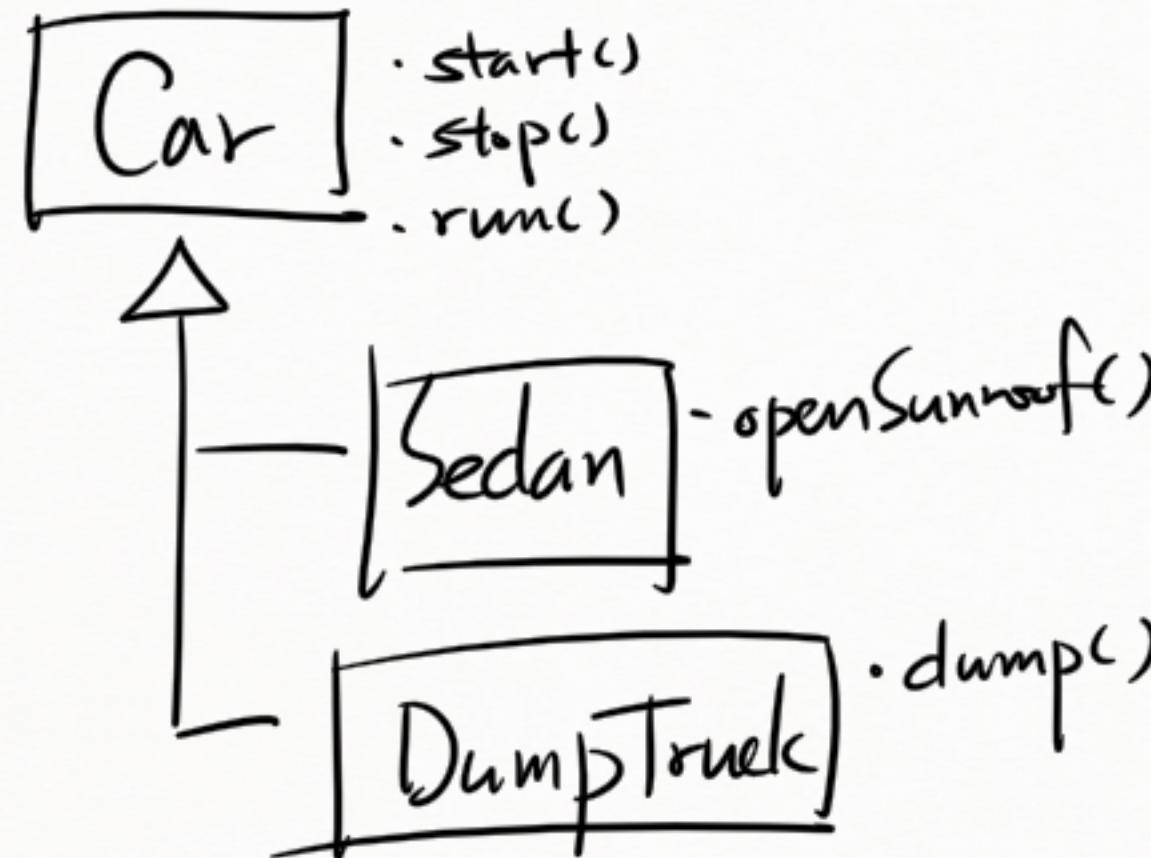
* 쿨러스 정의와 오버라이드 사용하기

```
class Board extends Object {  
    =  
}
```

(생략가능!)

```
class Member {  
    =  
}  
↑  
자체클래스를 2정의할 때  
기본으로 Object를 상속합니다.
```

* 상속과接口



Sedan s;

s = new Sedan();

s.start()
s.stop()
s.run()

s.openSunroof()

수퍼클래스의
기능은 자식
클래스에
다른

~~Car = (2m/2
primitive-type)~~

~~s = new Car()~~

s.start()
s.stop()
s.run()

~~s.openSunroof();~~

s의 인터페이스
Sedan의 기능은
모두 s가
갖고 있다.
Car는 자식
클래스에
다른

자신
클래스의
기능은
자신
클래스
만에
있다.
Car는
모두
Car
클래스
만에
있다.

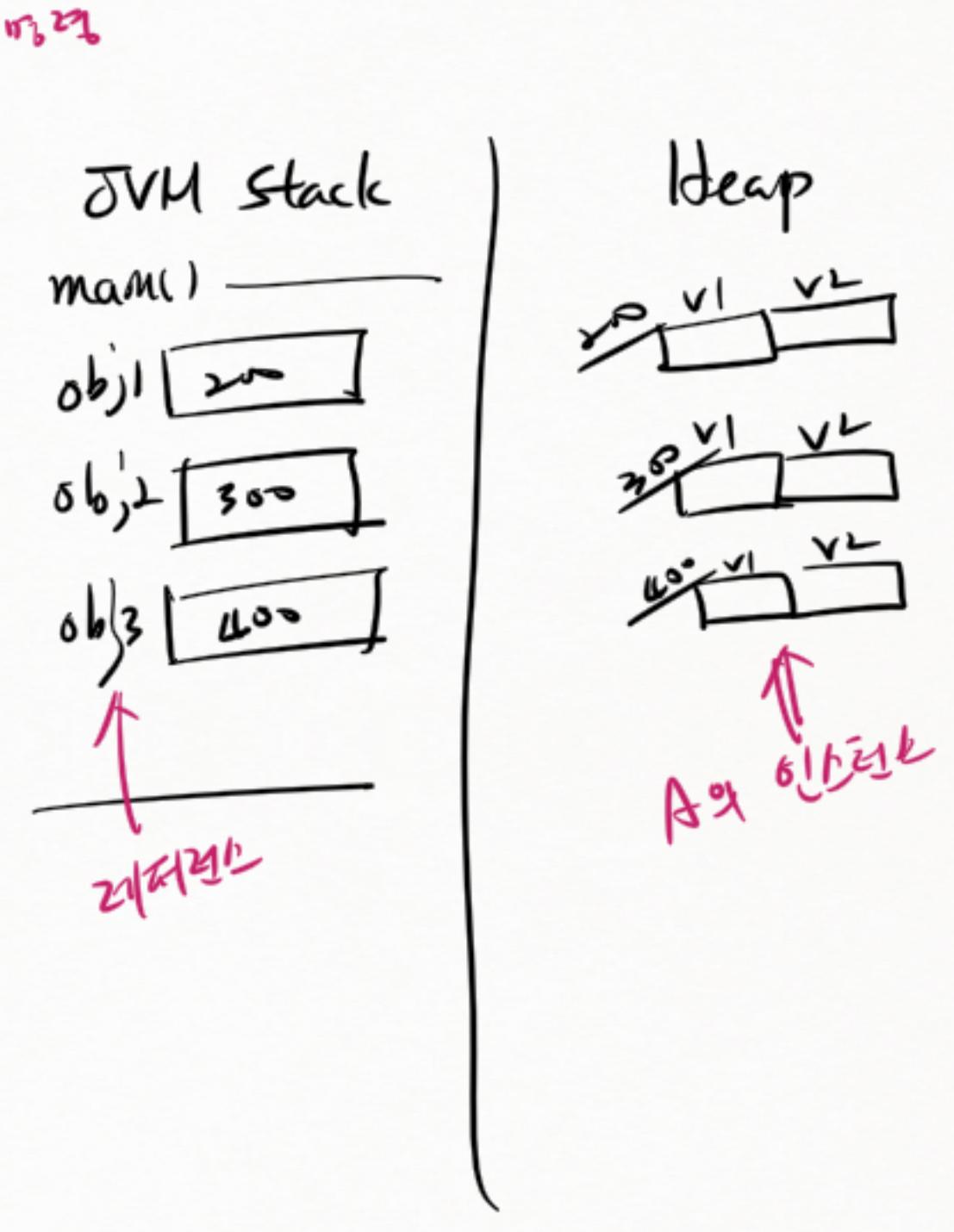
* static 필드와 non-static 필드

```
class A {  
    int a; // non-static 필드(변수) ← Heap ← Garbage Collector가  
           // 관리할 영역  
    static int b; // static 필드(변수) ← Method Area  
}
```

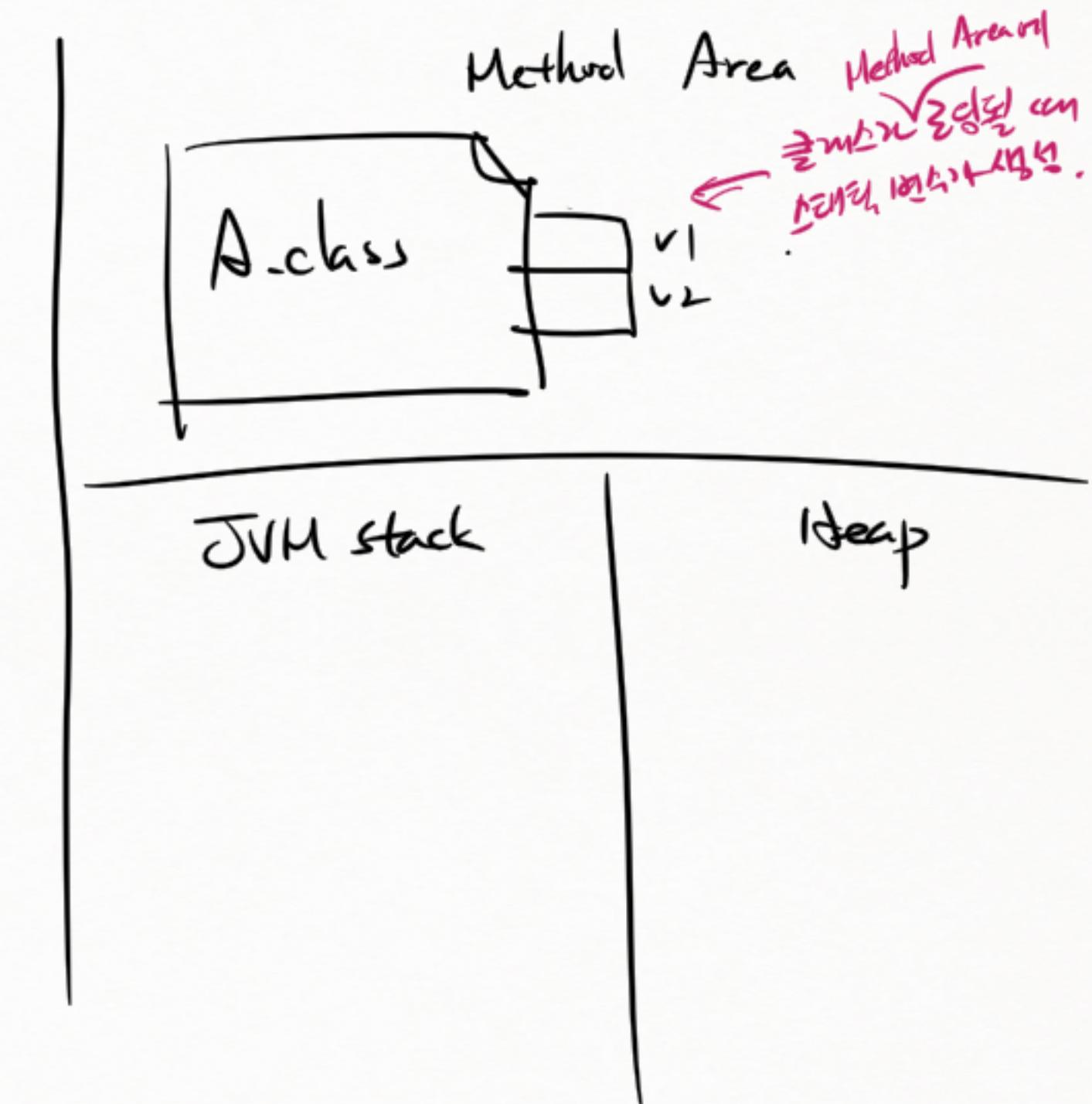
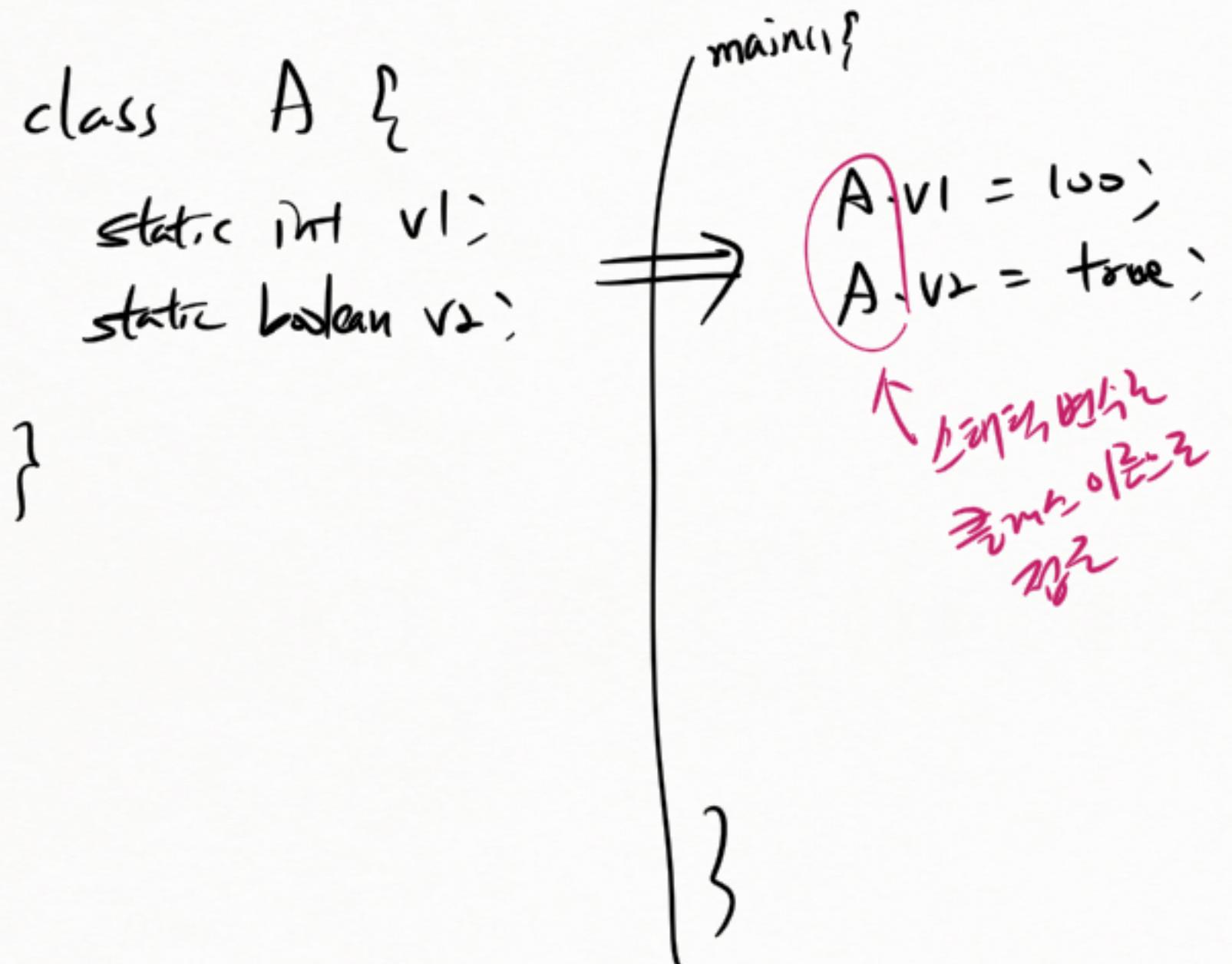
* DLR (non-static) မျှ

The diagram illustrates the execution flow between two code snippets:

- Left Snippet:** A class definition `class A { int v1; boolean v2; }` followed by a function definition `void m1() { } }`. The variable declarations `v1` and `v2` are circled in red.
- Right Snippet:** A main function `main() { }` containing three object creations: `A obj1 = new A();`, `A obj2 = new A();`, and `A obj3 = new A();`. The objects `obj1`, `obj2`, and `obj3` are circled in red. An arrow labeled `call by value` points from the `m1()` declaration to the `obj1.m1();` call in the main function. A vertical line connects the `m1()` declaration to the `obj1.m1();` call. A red box highlights the `obj1.m1();` call. Below it, a red-outlined box contains the assignment `obj1.v1 = 100;`.



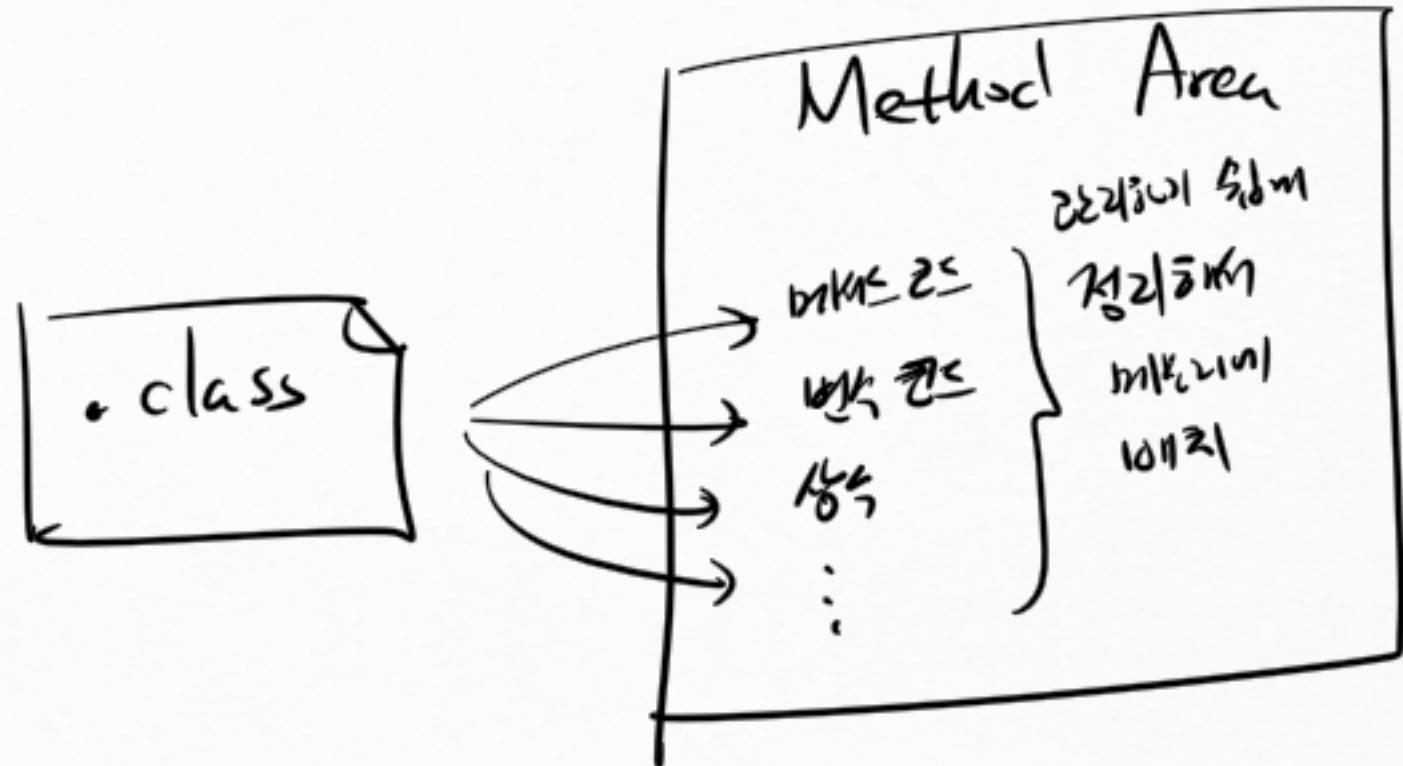
* 주소(static) 필드(값)



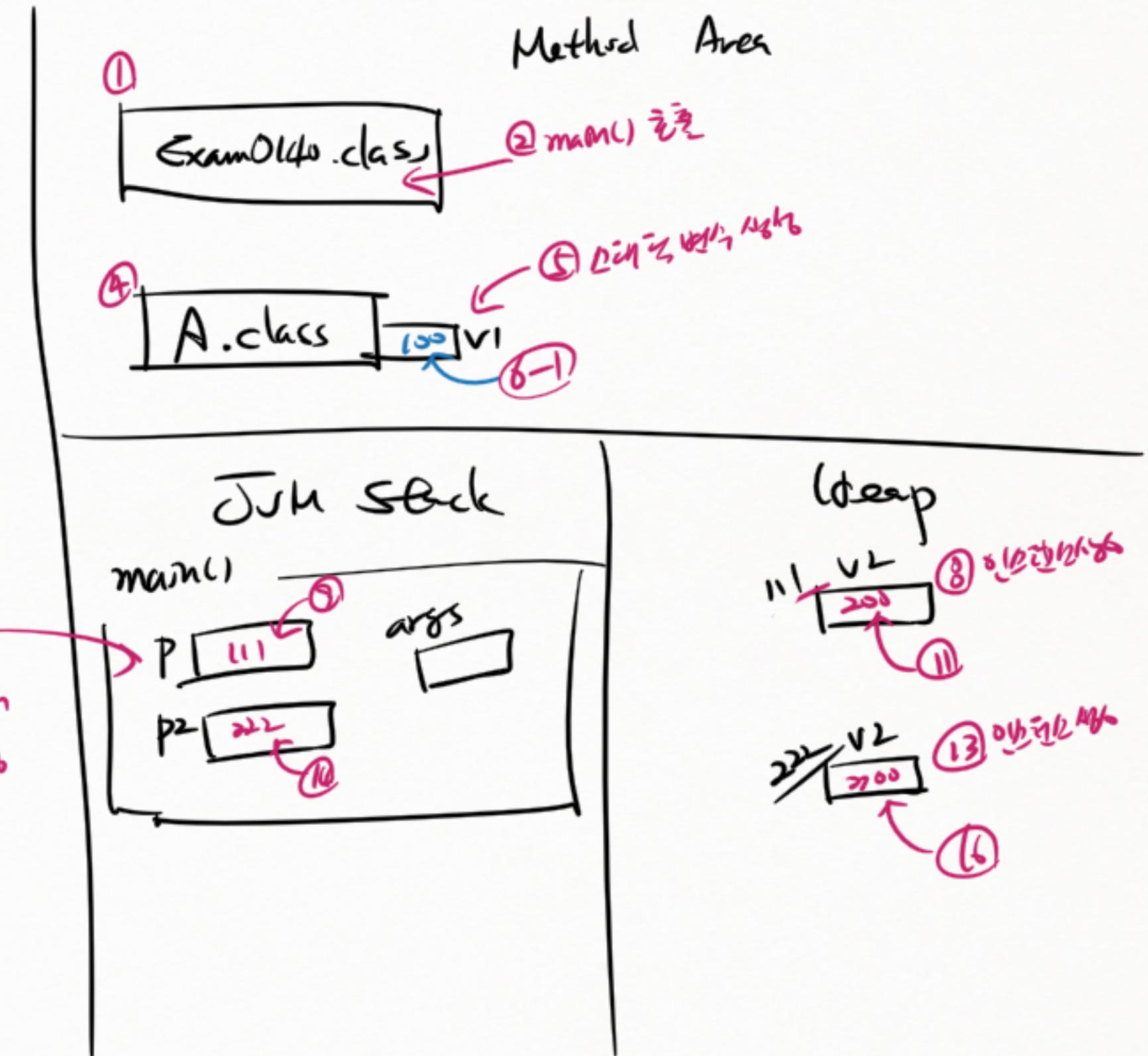
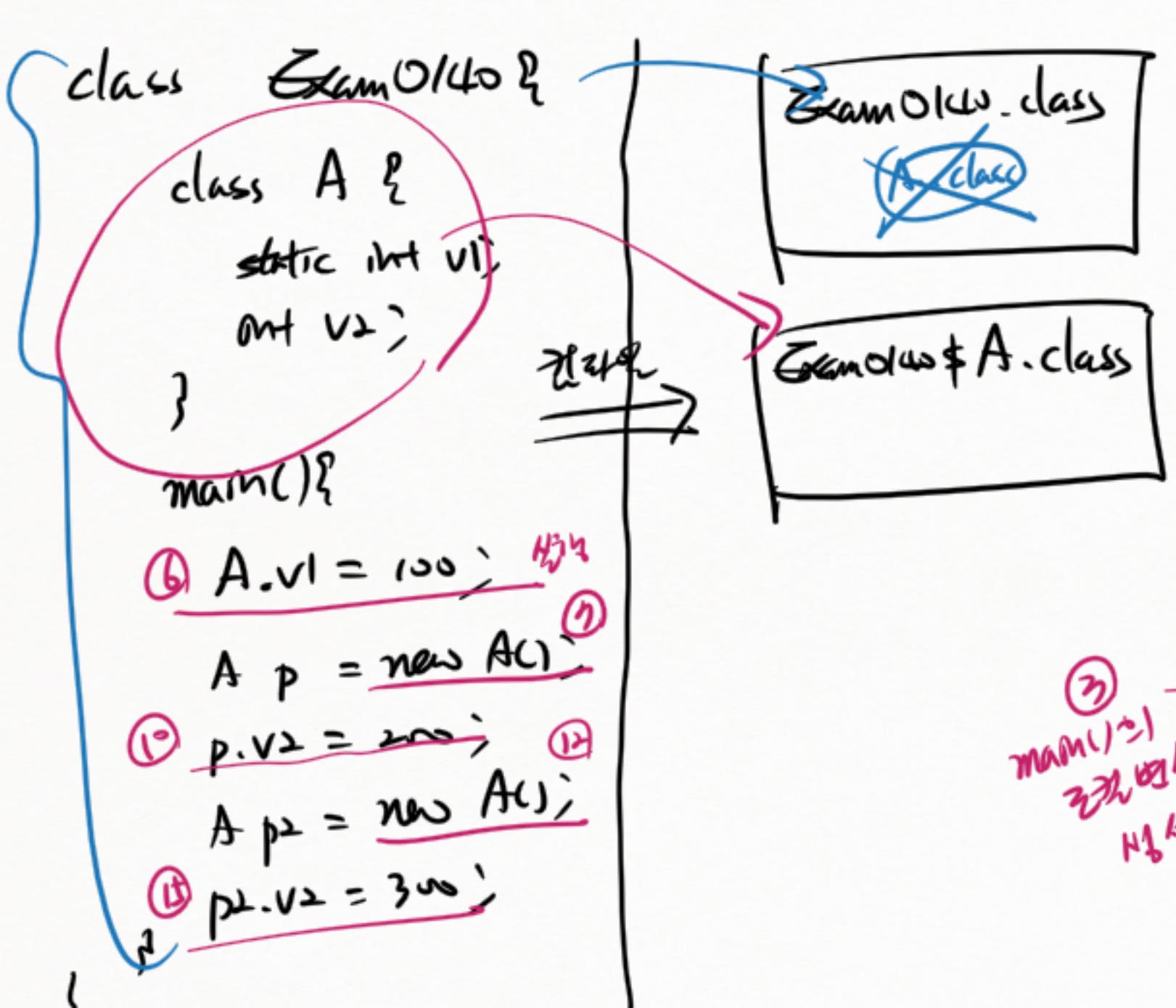
* JVM의 로딩과 실행

\$ java Hello

- ① Hello.class 찾는다
- ② Bytecode 검증
- ③ Method Area에 로딩 \Rightarrow
- ④ 스레蚀 필드 생성
- ⑤ 스레蚀 블록 실행
- ⑥ main() 호출

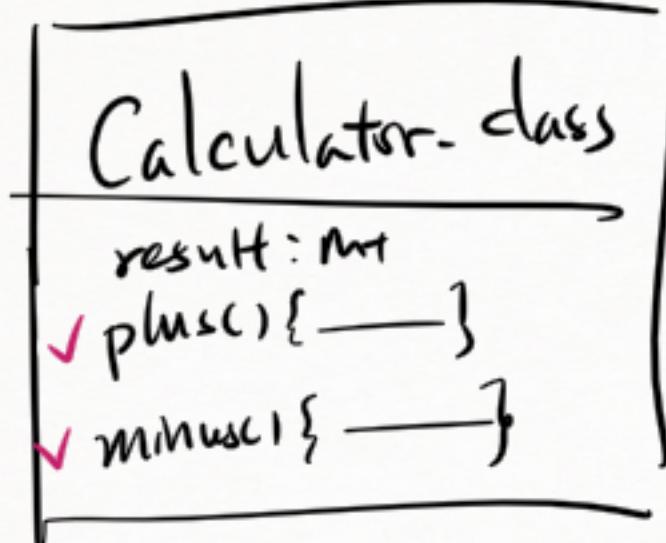
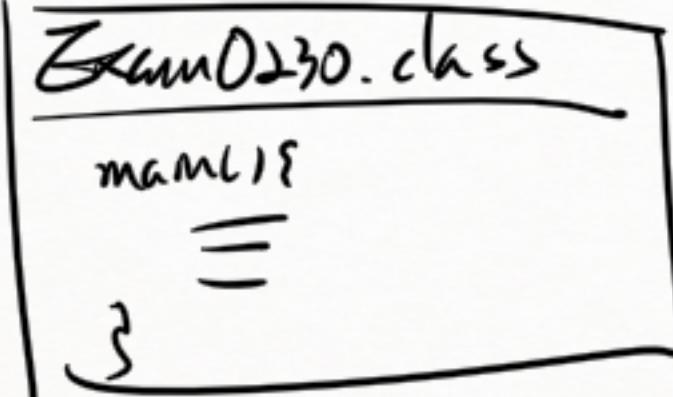


* 예제 2 문제, 소스코드 및 실행 결과 분석

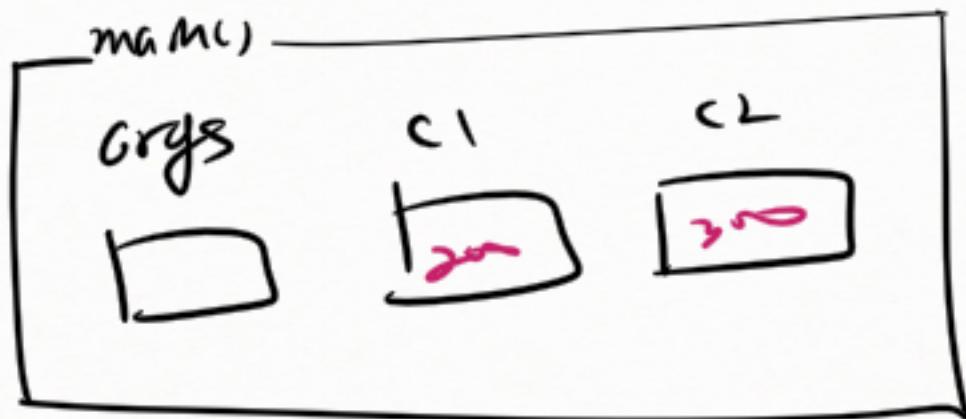


* 인스턴스 변수와 인스턴스 Method 둘다 3/21

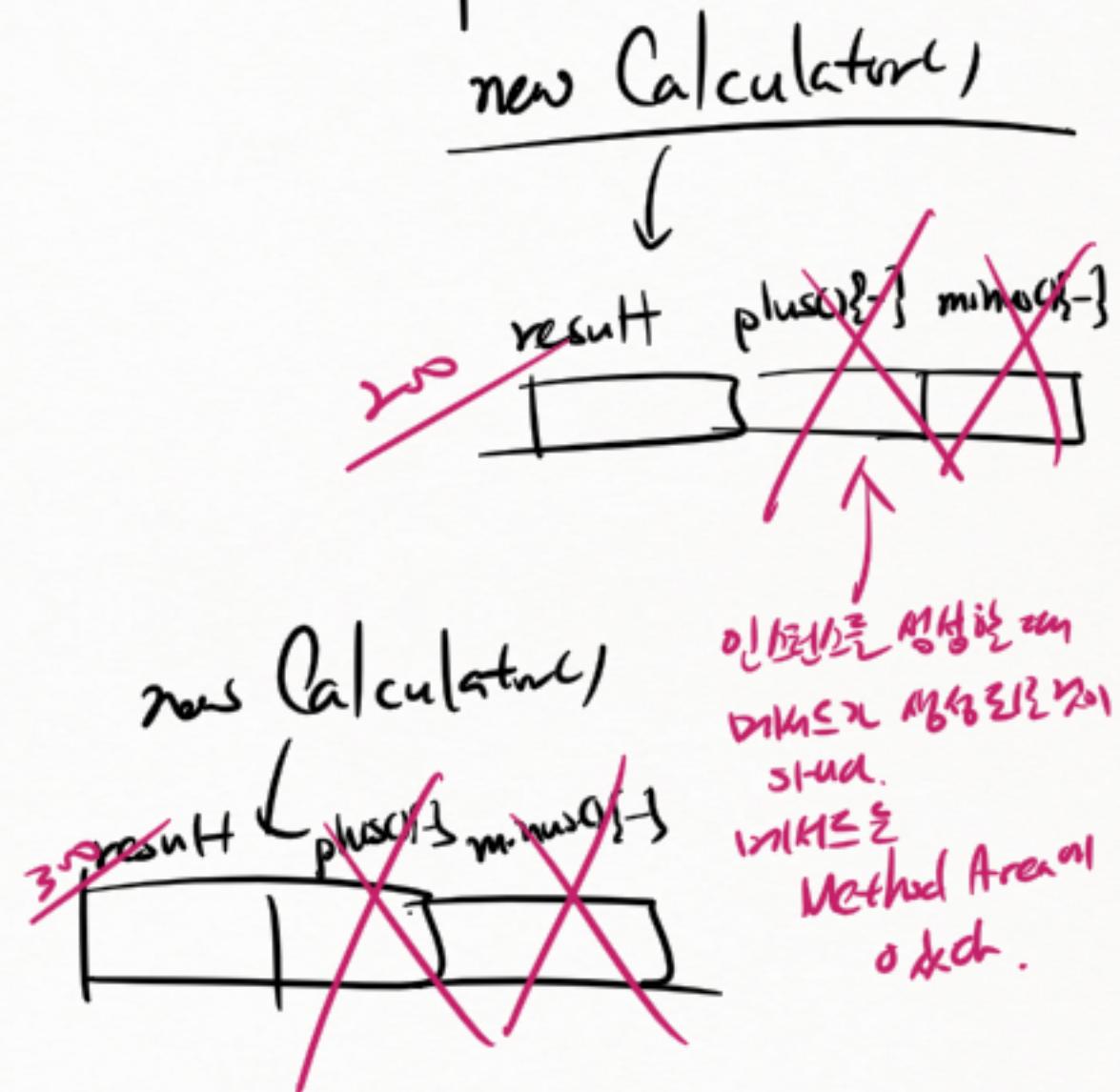
Method Area



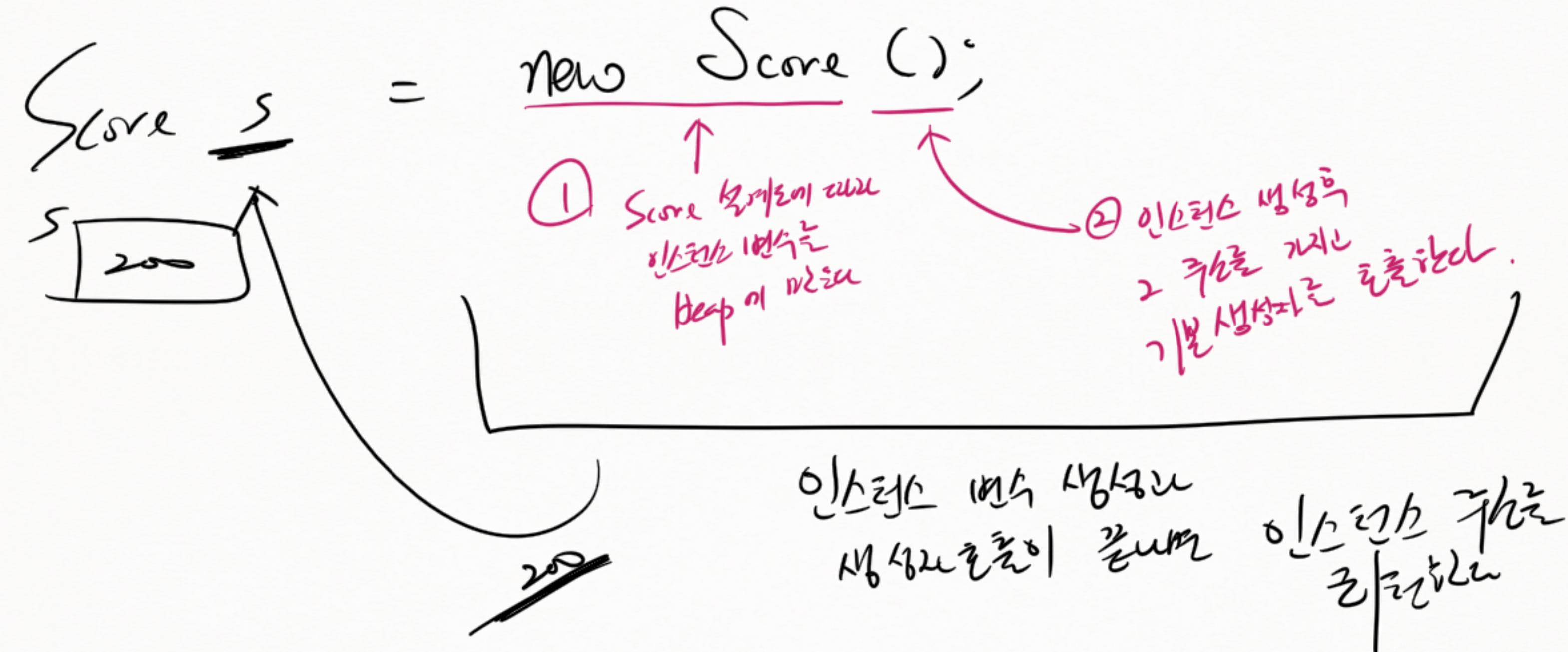
JVM Stack

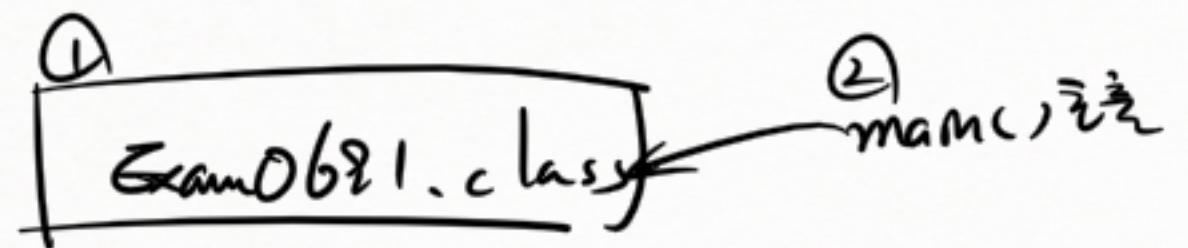


Decp



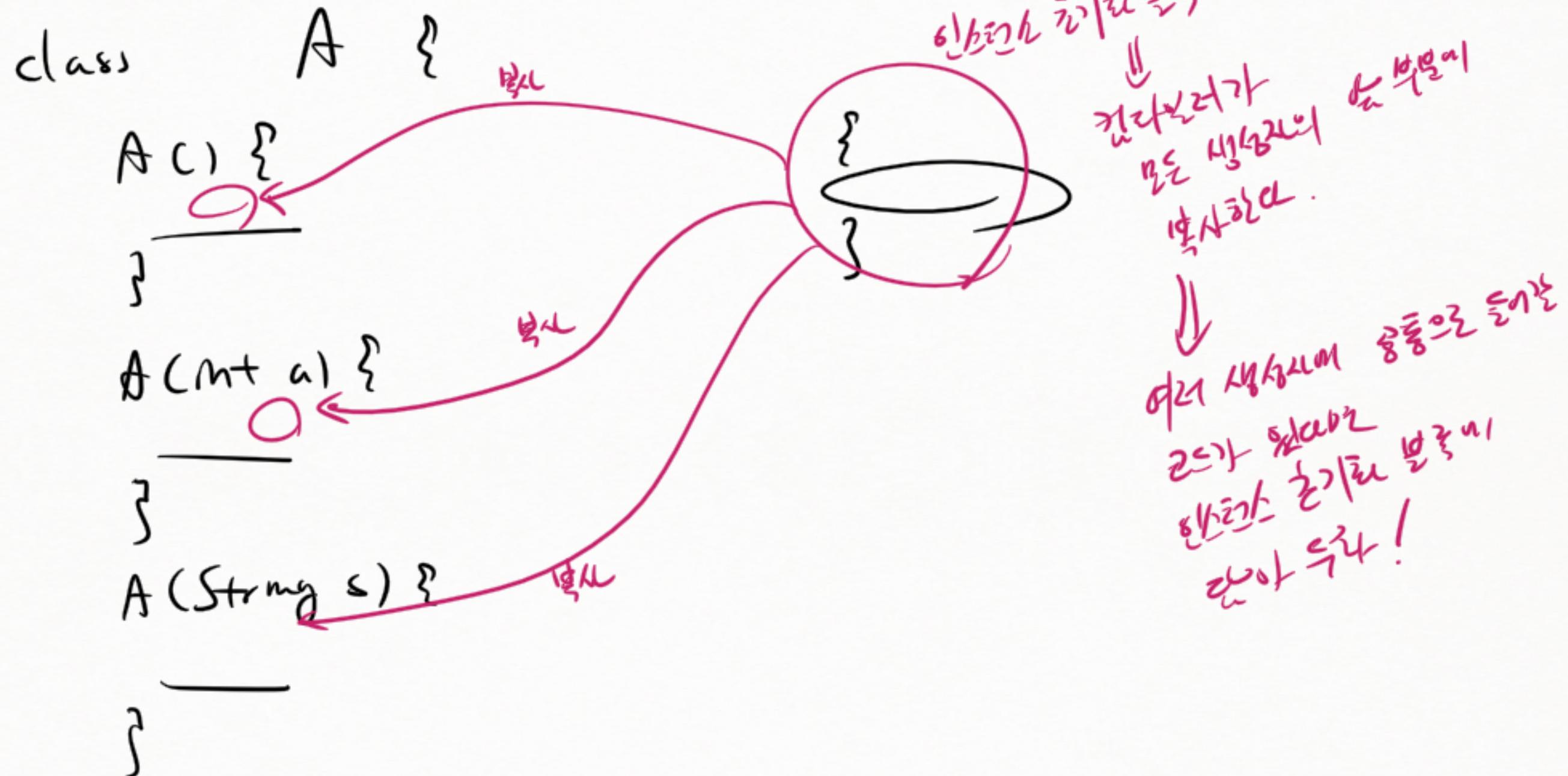
* 인스턴스 생성과 사용



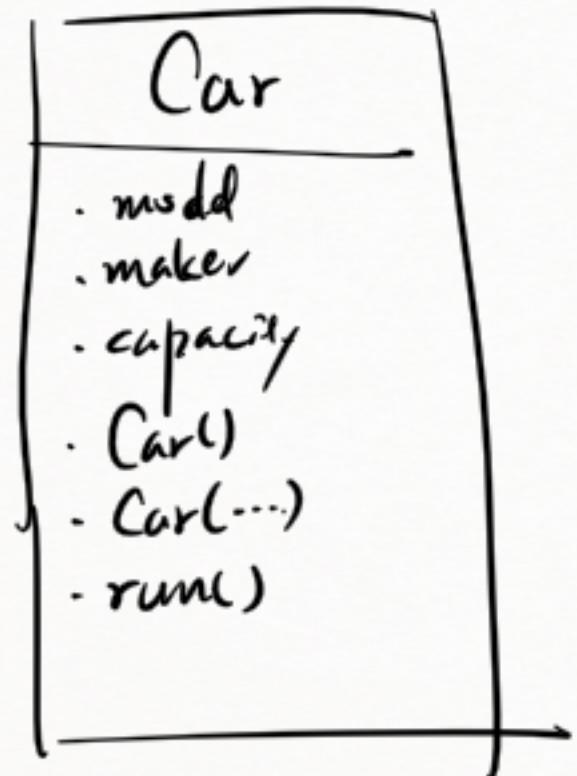


① A.static{}
② B.static{}
36
29

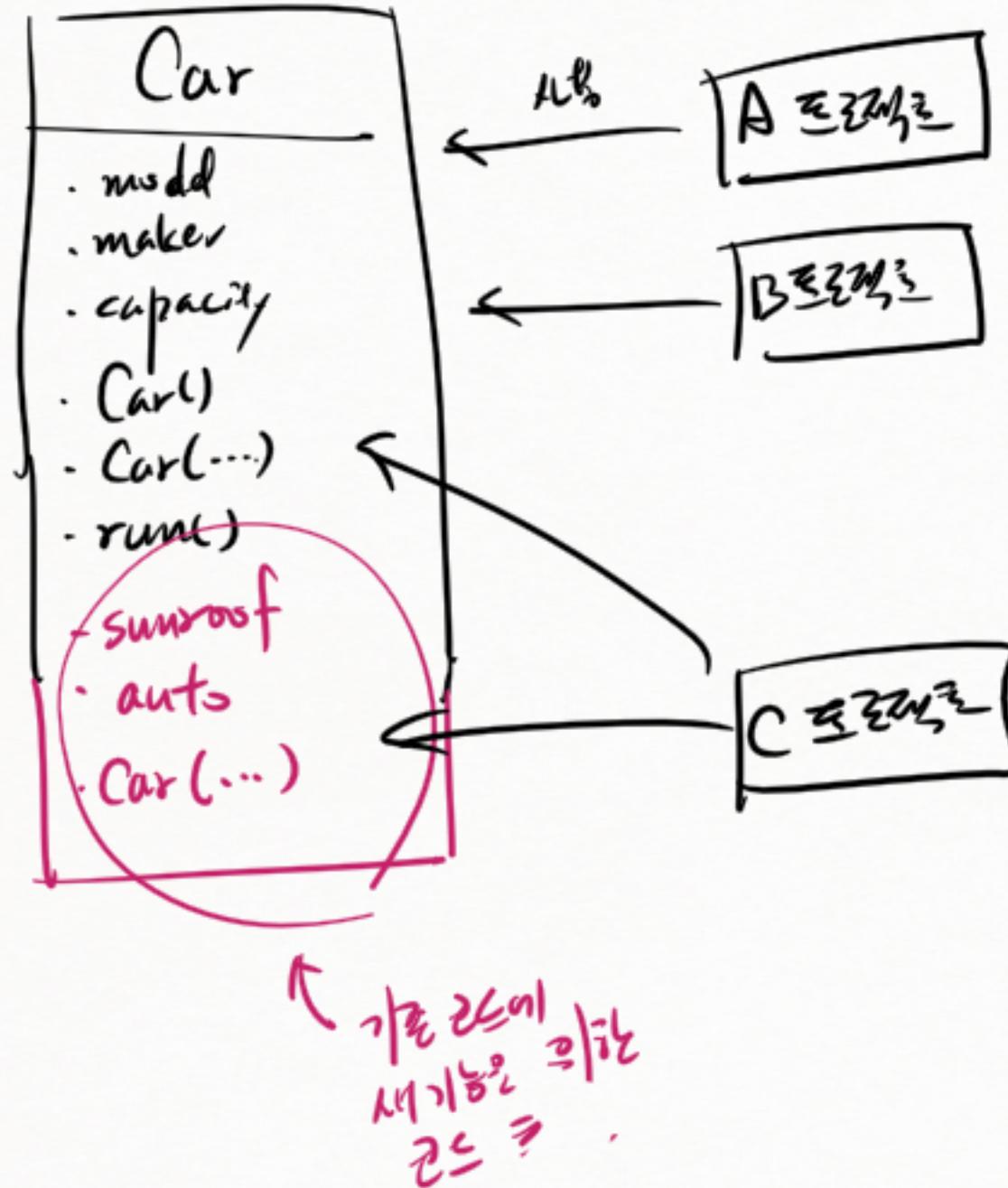
* 인스턴스 초기화 (instance initializer)



* 자료구조 예제

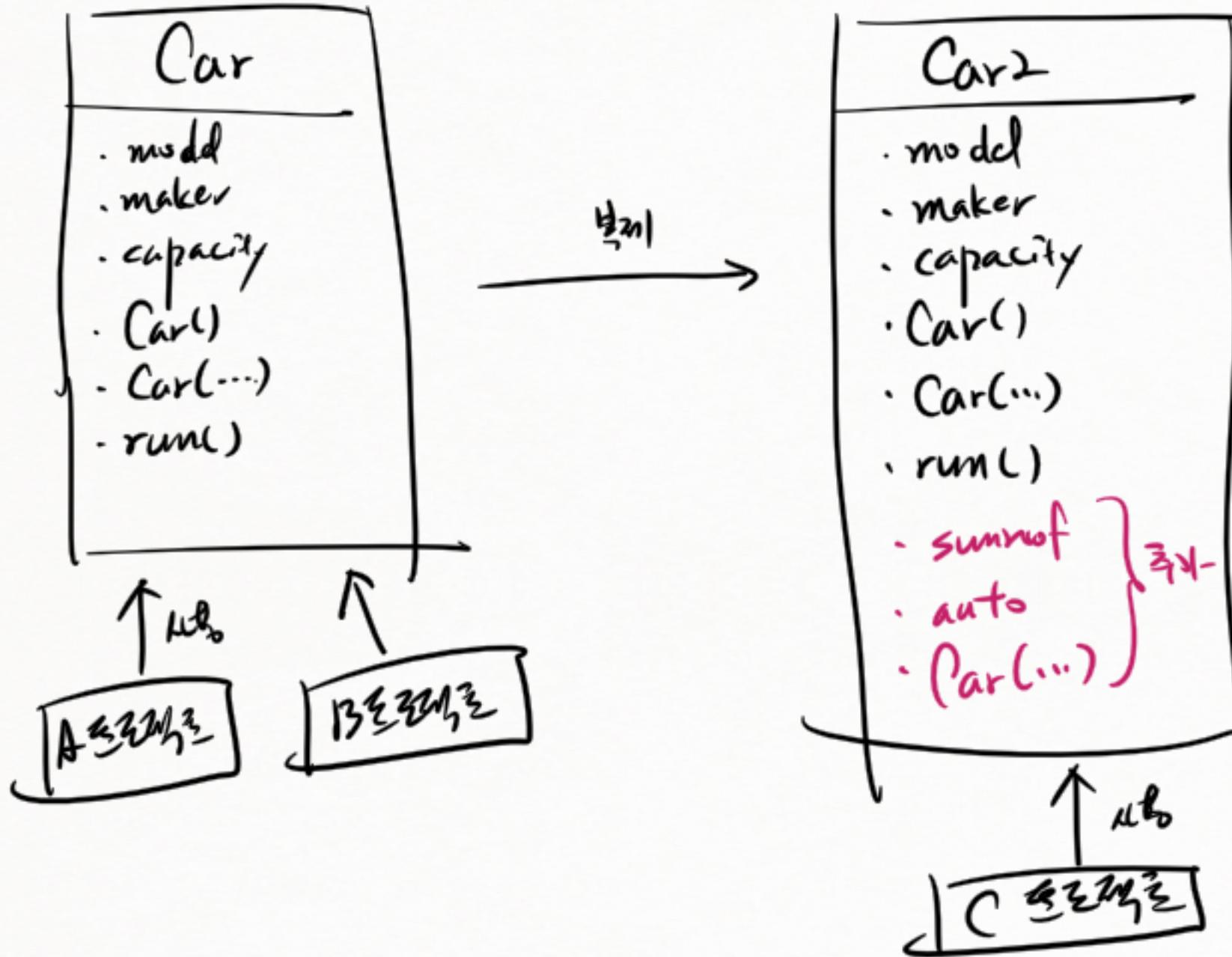


* 기능을 확장하는 방법 - ① 기존 클래스 연장



★ 예제
① 같은 예제 기능을 확장하는 경우 예전에 했던 것과
★ ② 같은 코드를 사용하는 프로젝트에 확장성을 기준해.
A와 B 프로젝트

* 자동차 클래스의 상속 - ② 자료 속성을 복제하는 경우



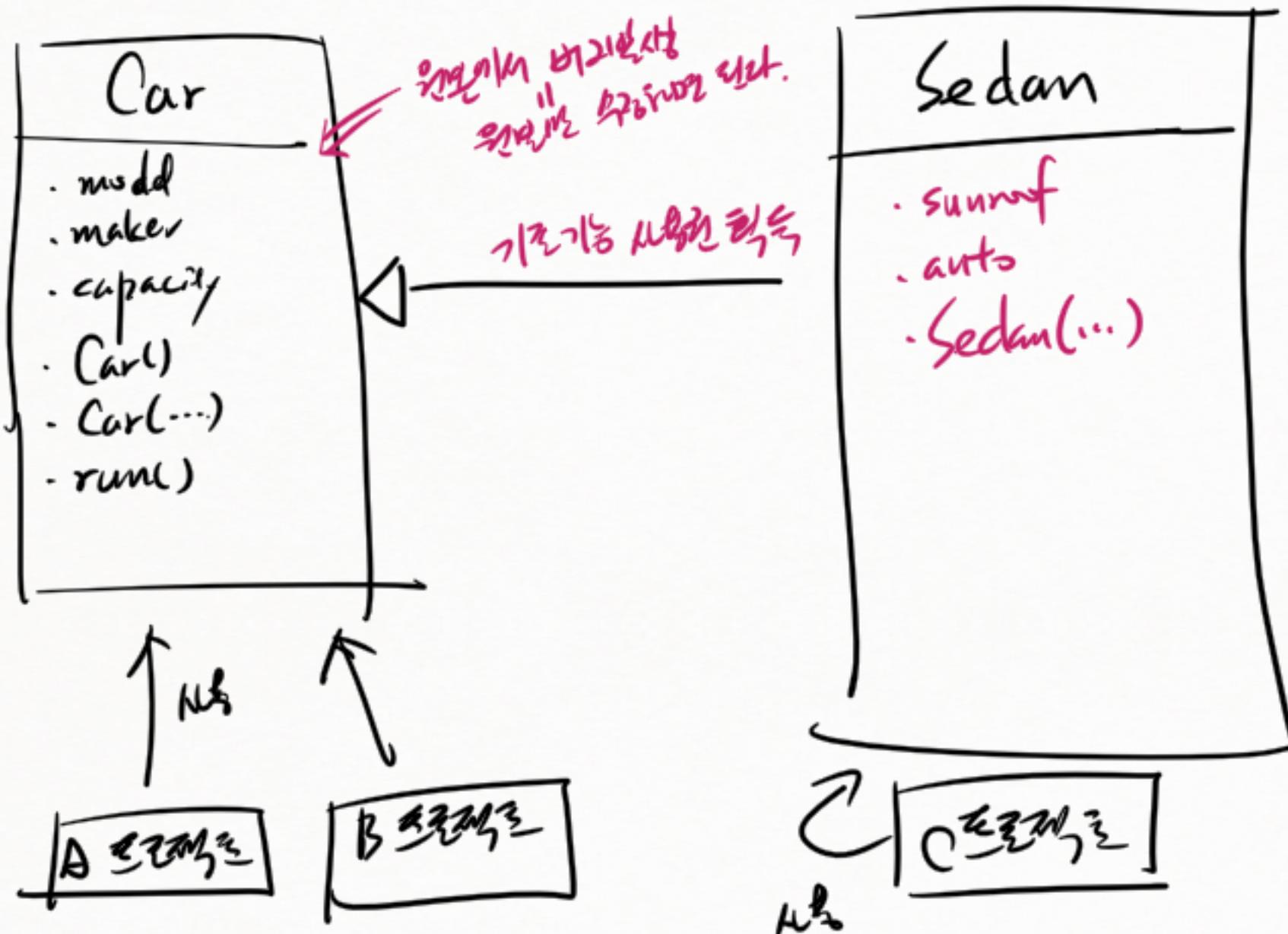
특징

- ① 자료 속성을 헤게이트로
기존 프로토 타입 패턴을 확장화.
→ 중복되는 책임

단점

- ① 복제 → 중복되는 책임
→ 가능하지 않다 → 자체적인 책임
→ 비슷하지 않다 → "유지보수를 한다."

* 기능을 확장하는 방법 - ③ 상속을 이용



특징!

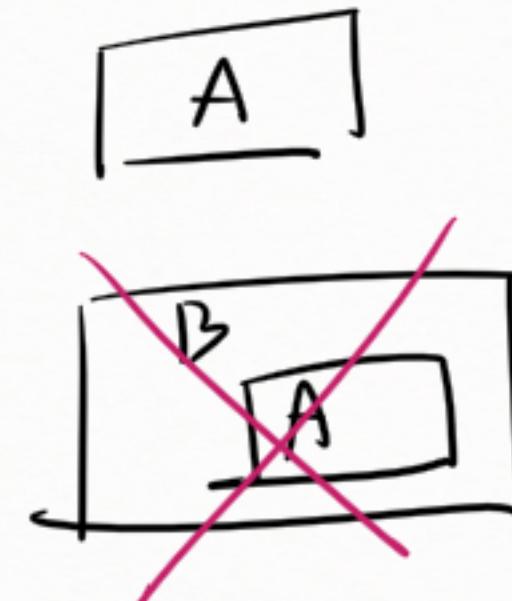
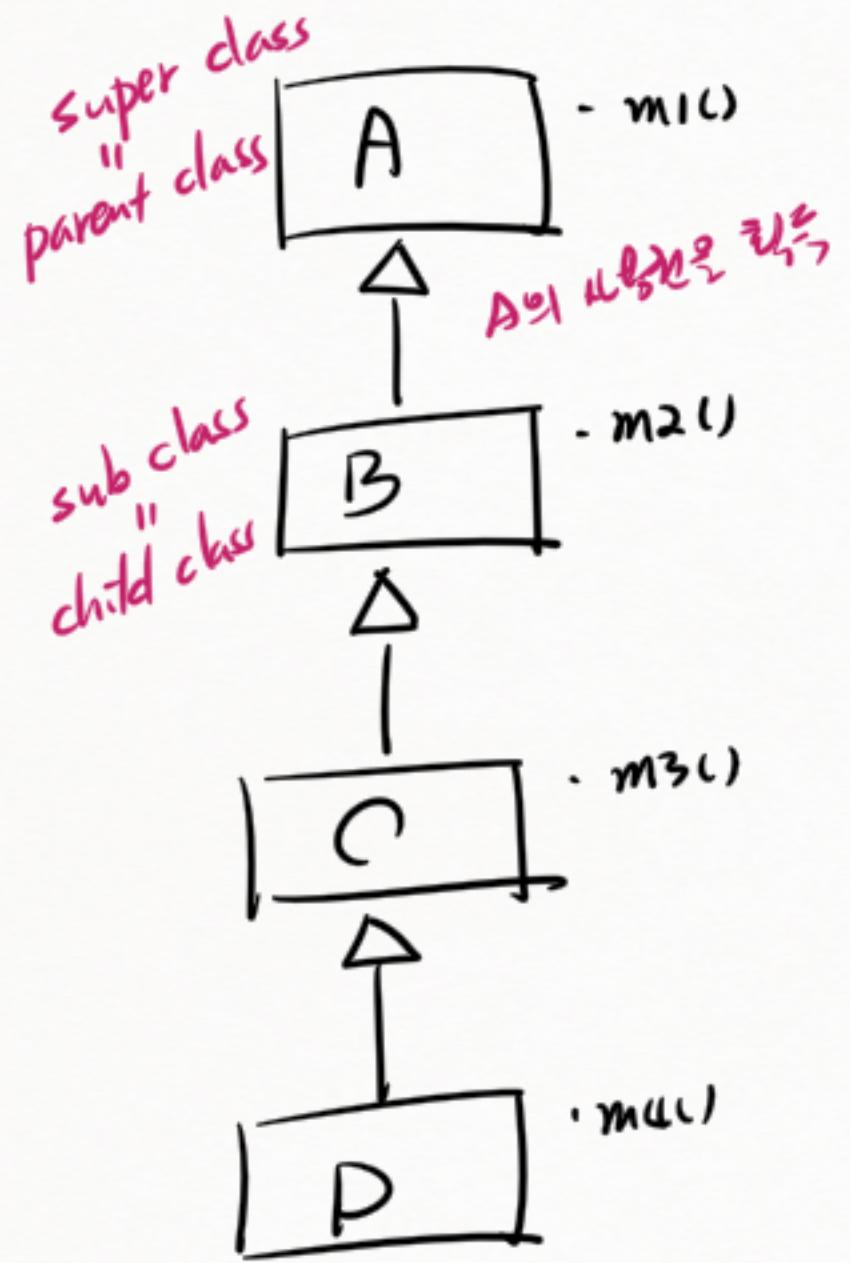
- 기존 코드를 통해 상속.
↓
이전 프로그램에 영향을 가지지 않아야
- 기존 코드를 재사용 → 비용 절감
→ 버그 추가 가능성이 ↓

단점!

코드 중복을 없애야.
↳ 버그 수정이 힘들

- 여러 단계를 거쳐서 넣어
마다 뭔가 기능은 강제로 상속받을 경우가 있다

* 부기된 멤버는



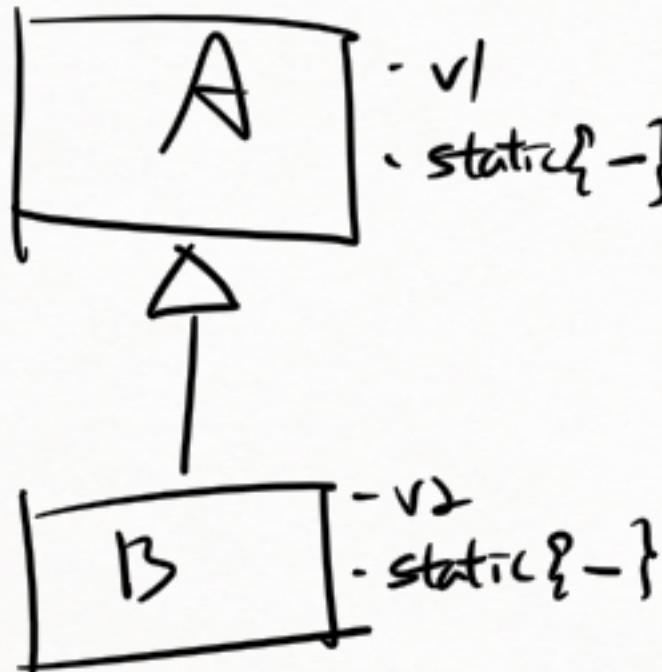
A의 주소를 가진 B의 주소를 가진 A는 허용되지 않다!

B obj = new B();

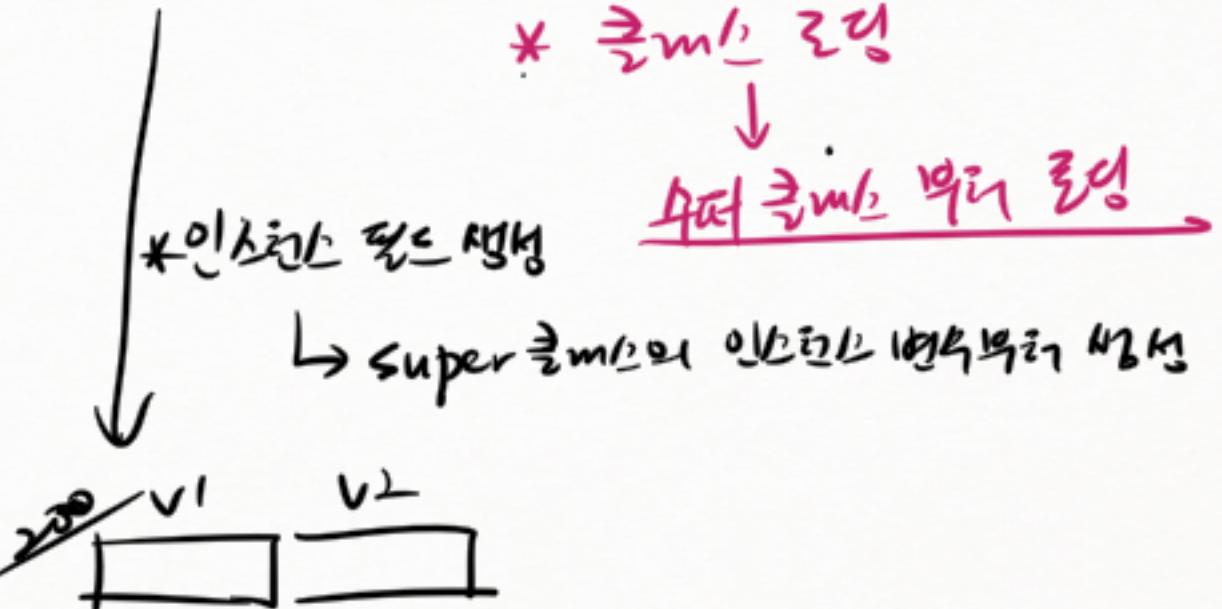
obj. m2(); // ok

obj. m1(); ← B 클래스를 통한
↑
A의 m1() 접근
*A 클래스의 멤버는
A를 통해 접근 가능*

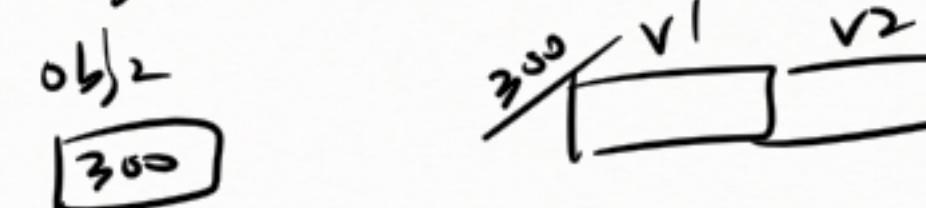
* 상속과 인스턴스 필드(변수)



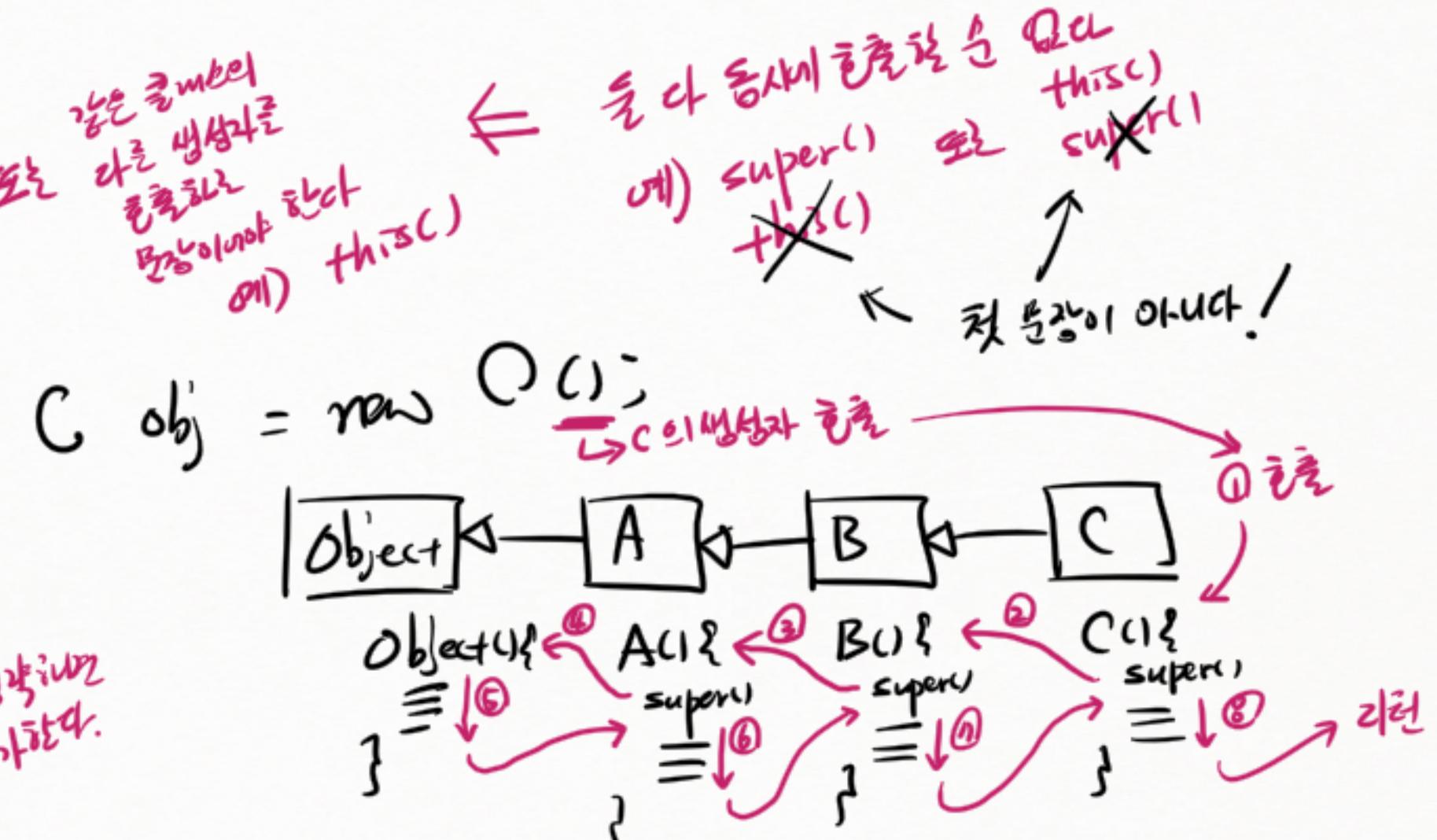
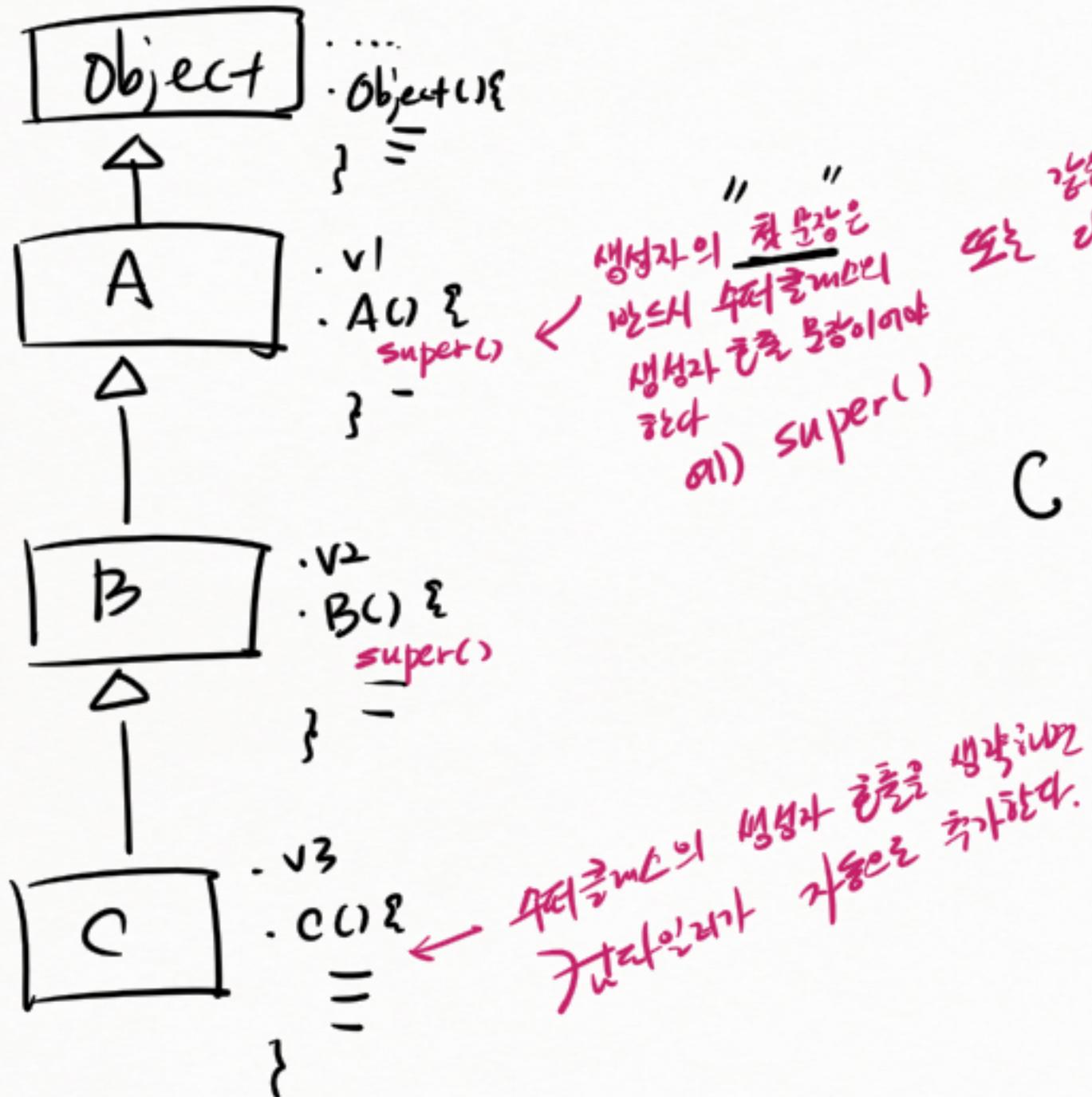
$B \ obj_1 = new B();$



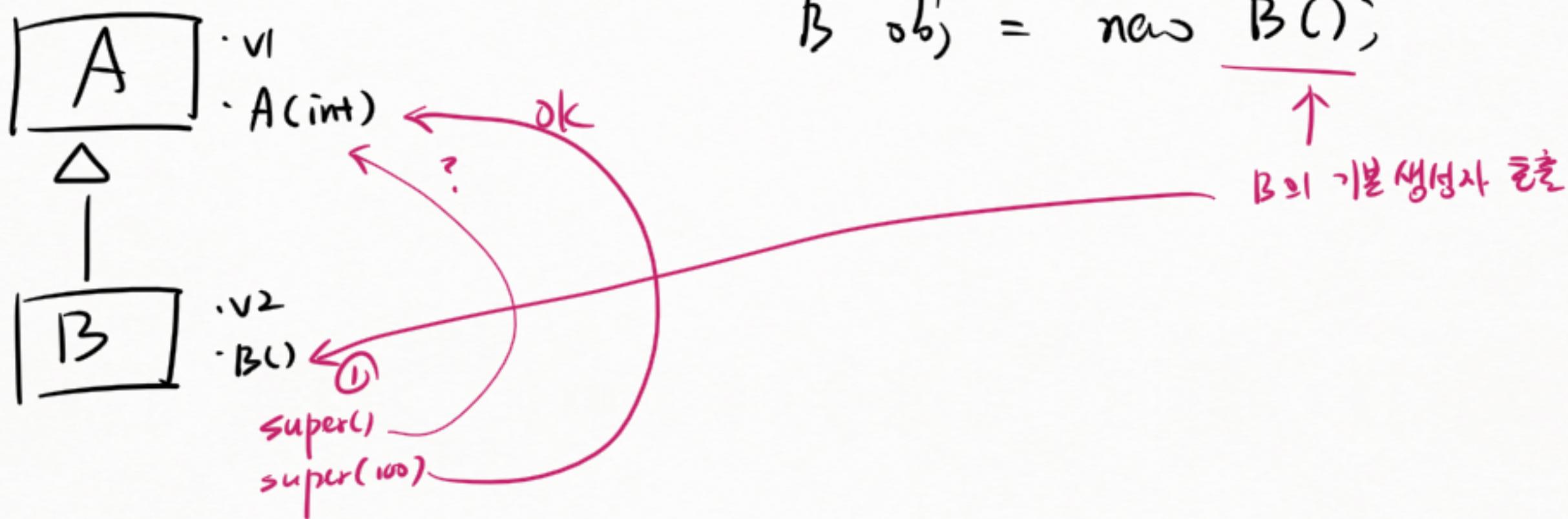
$B \ obj_2 = new B();$

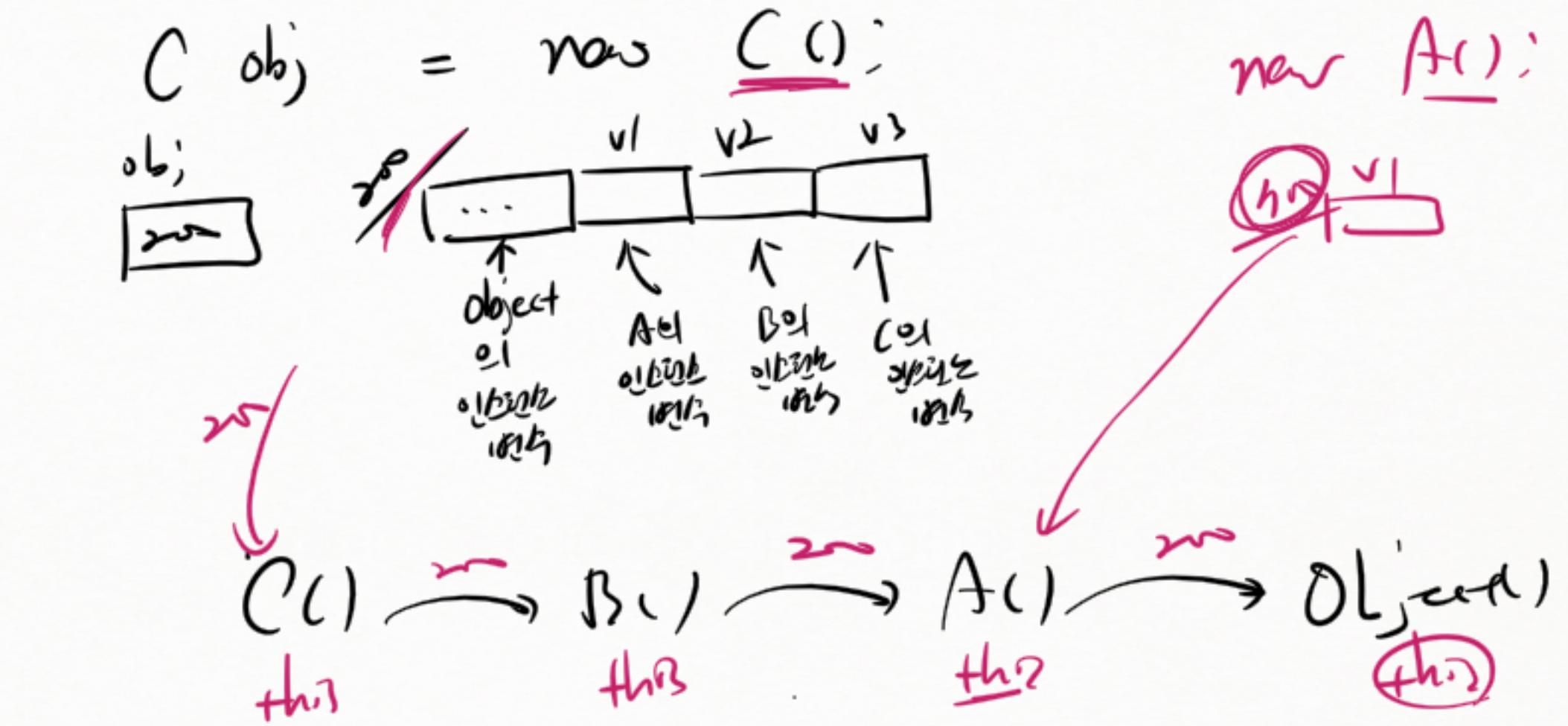


* 생성자 호출 순서

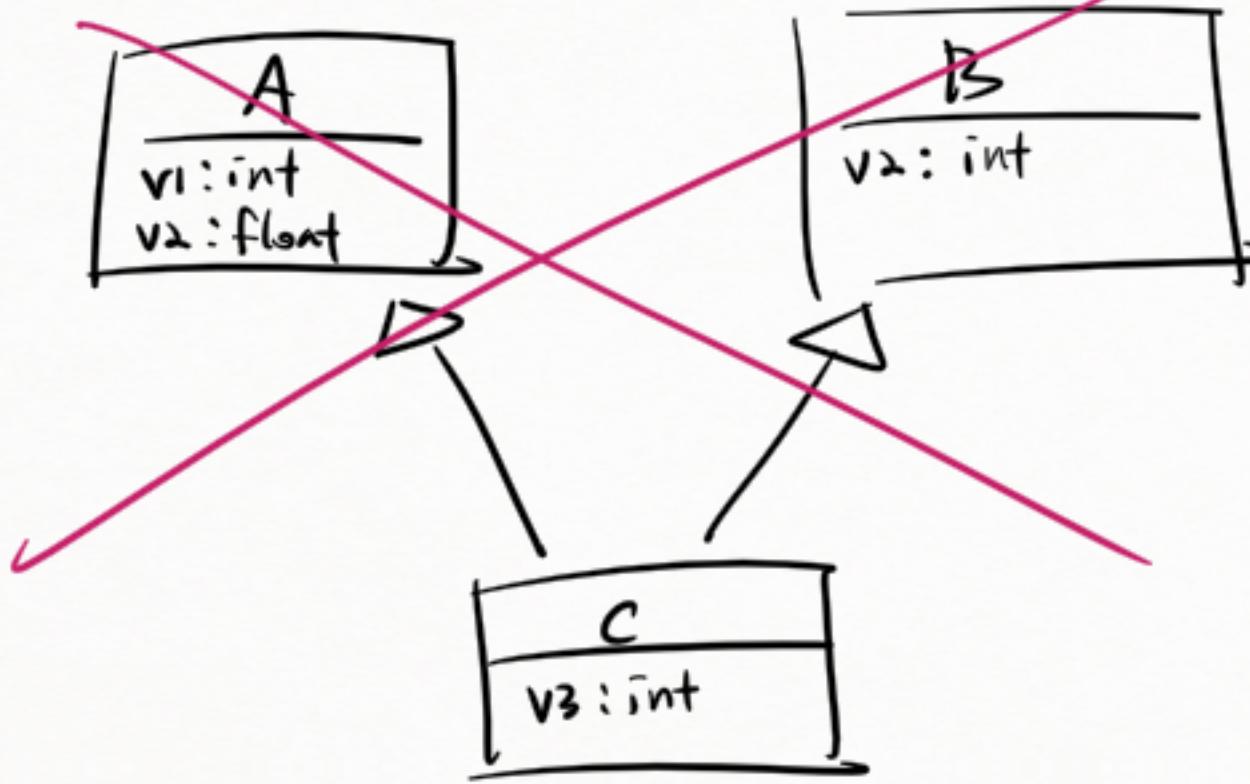


* 생성자 호출 2

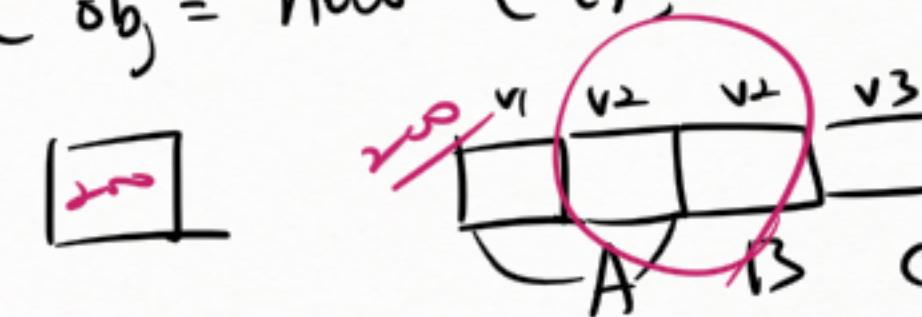




* 다중 상속 \Rightarrow 자원 누획!



$C \text{ obj}' = \text{new } C(1);$



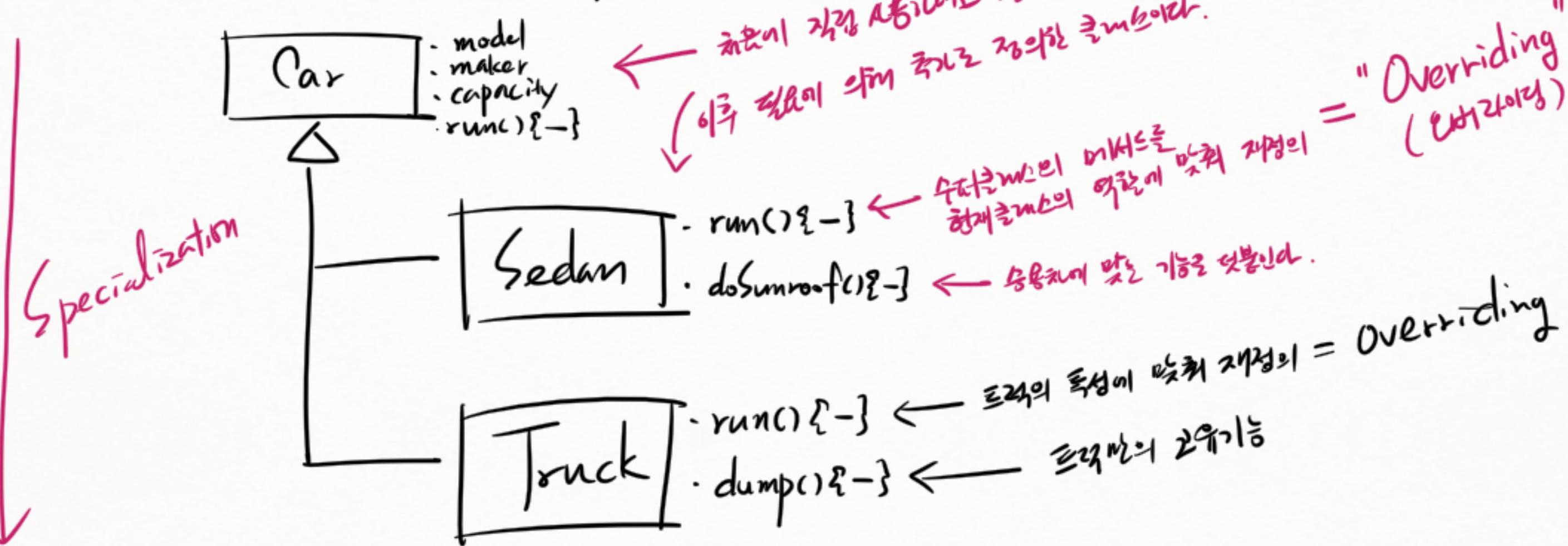
$\text{obj}' \cdot \underline{v2} = 0;$

A의 변수값
B의 변수값
구현화가 없어.
구현화된 v2
를 찾을 수 없다.

증명
증명하기 위해서는
다중 상속을 허용하는
언어를 선택해보자

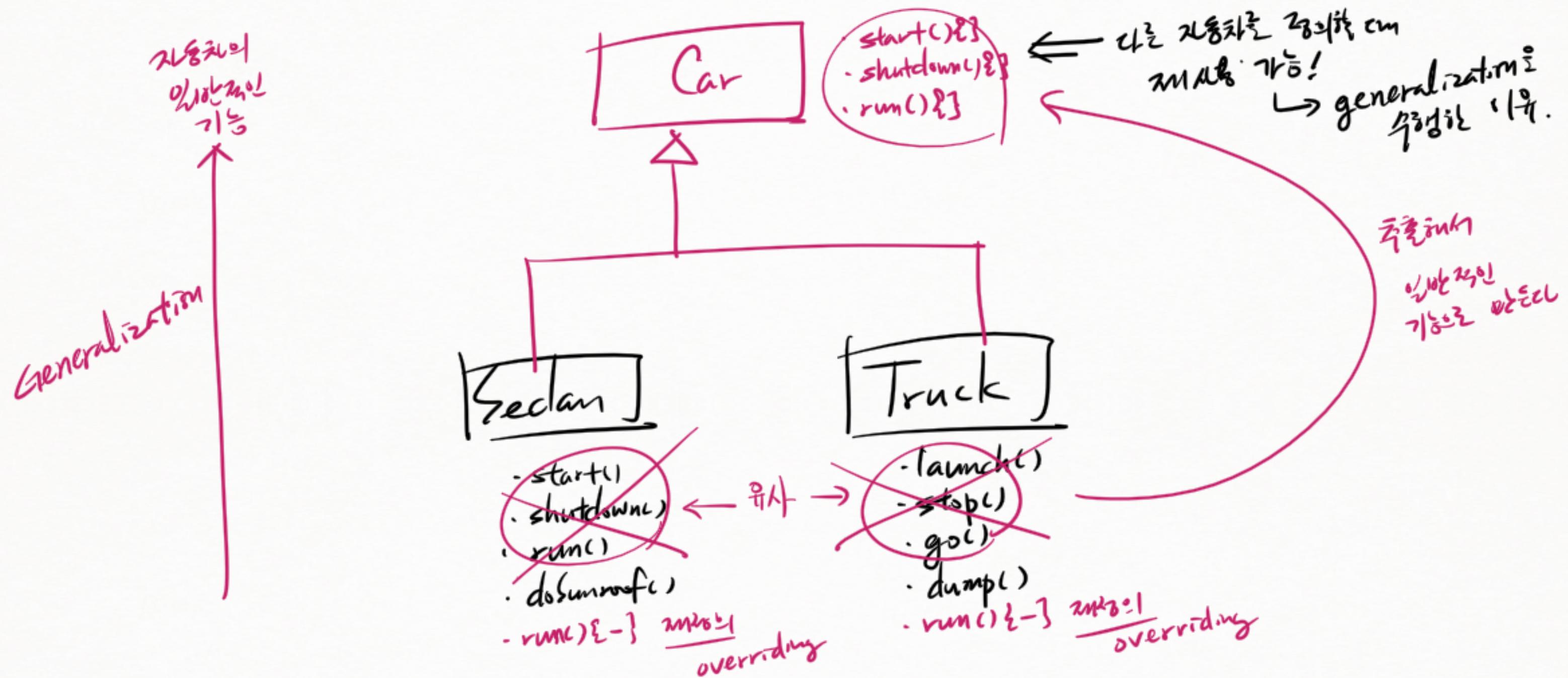
구현화된 v2
를 찾을 수 있다.
언어는 복잡
하지만 상속은 가능

* ↗ : specialization (전문화)

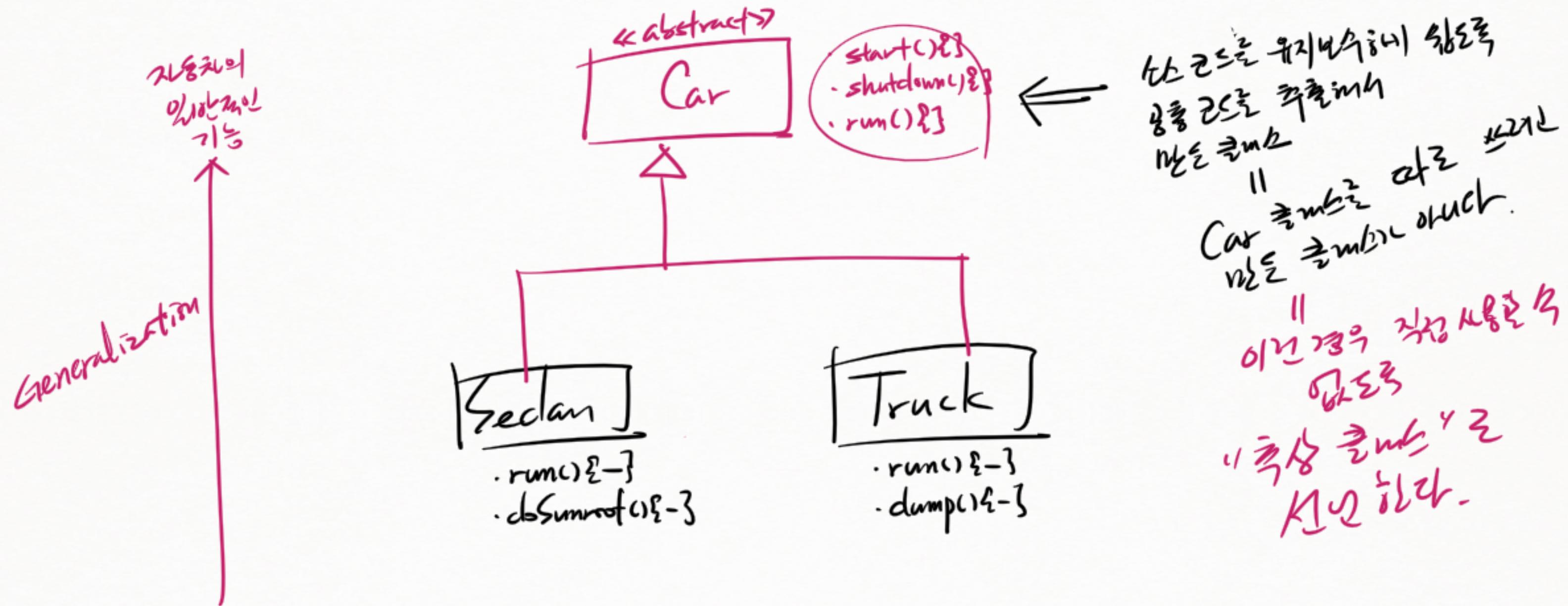


더 특별화된
기능을 만든다

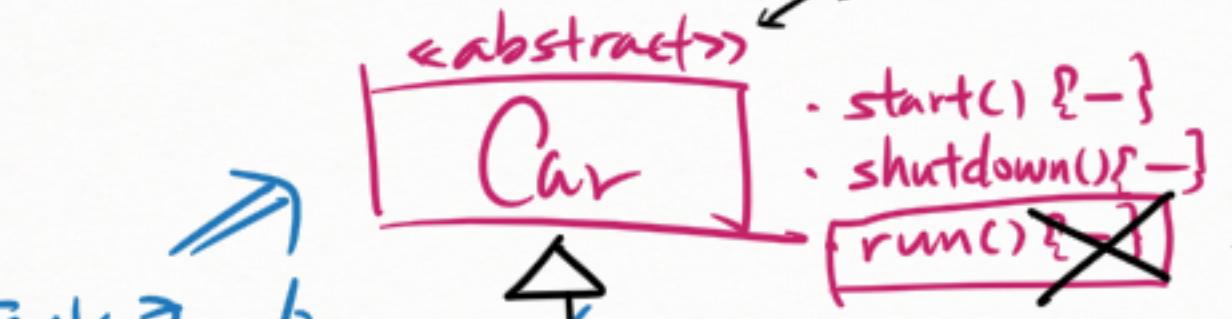
* 10: Generalization (일반화)



* 차량의 추상화



* 상속과 추상 클래스



추상 클래스

인터페이스

~~new~~ (Car)

~~obj. (Sedan)~~

"~~자식~~ 추상 클래스"
↳ 인터페이스 상속 가능



"~~자식~~ 추상 클래스"
↳ 인터페이스 상속 가능

"Method
Signature"

"function prototype"
(C/C++)

자체만 쓰기 가능
제한적

~~body~~
만들지 않아도
"추상 클래스"

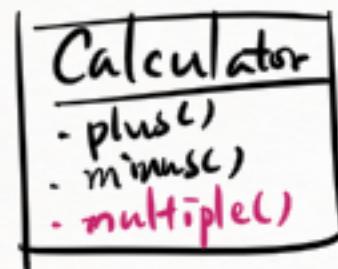
이전의
자체만 쓰기 가능
제한적

sub
인터페이스

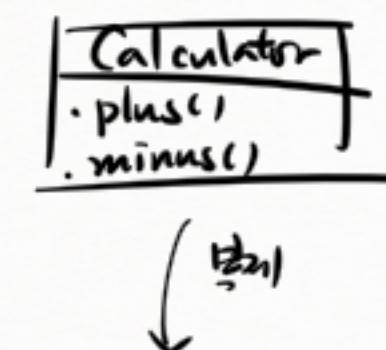
인터페이스

* 기능 학습법 cheat sheet

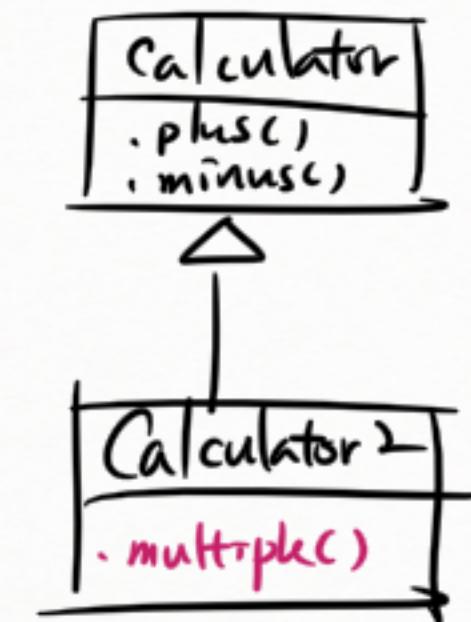
① 기능 재사용



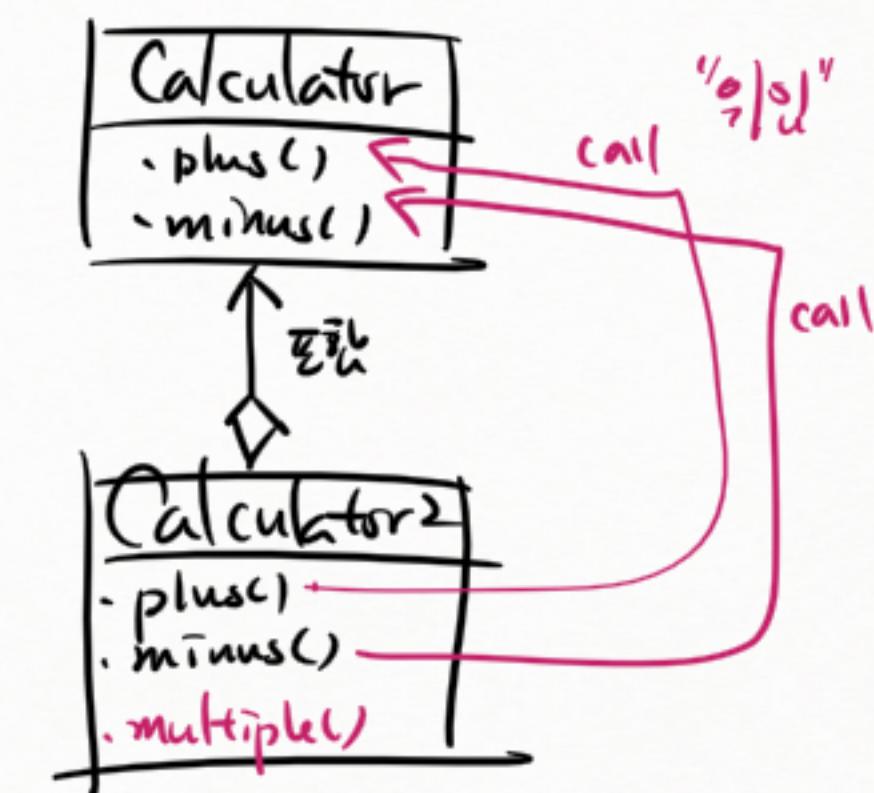
② 접근 복제하기 기능 추가



③ 상속

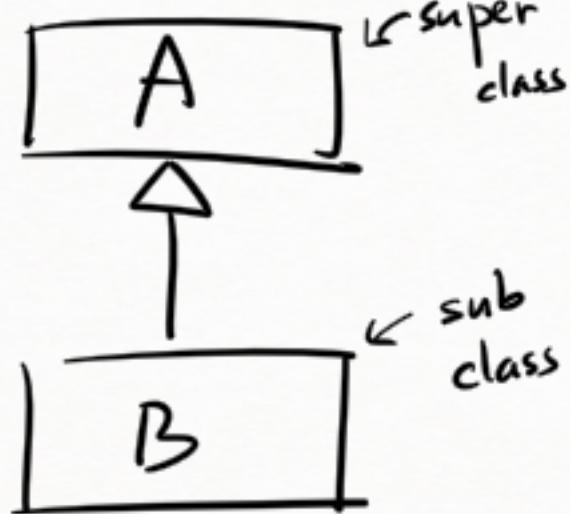


④ 오버라이드



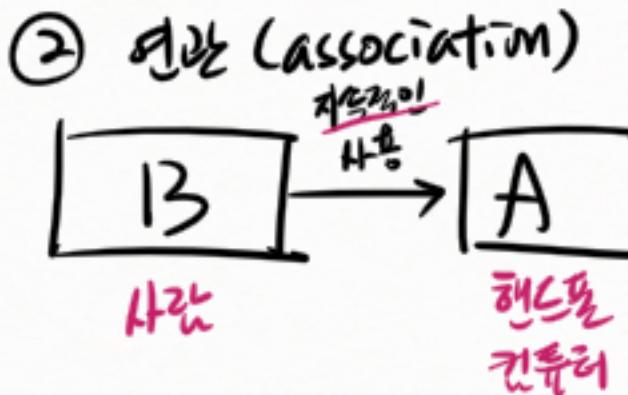
* 클래스 관계 cheat sheet

① 상속 (inheritance)



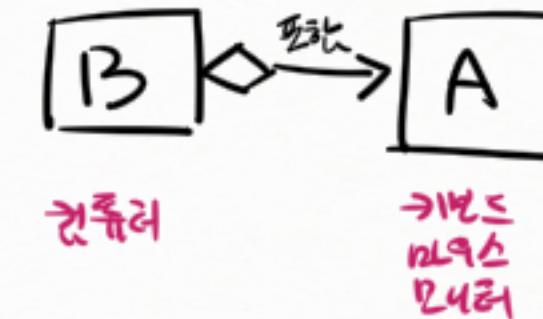
class B extends A {
≡ }

② 연관 (association)



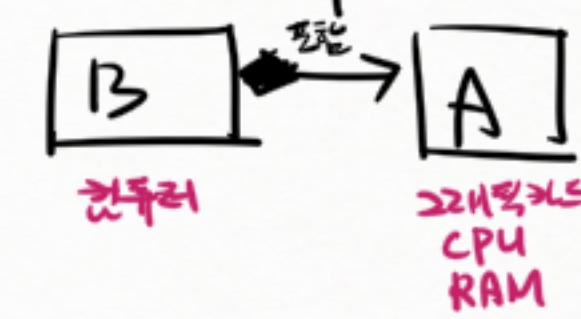
class B {
 A obj;
} ≡

③ 집합 (aggregation)



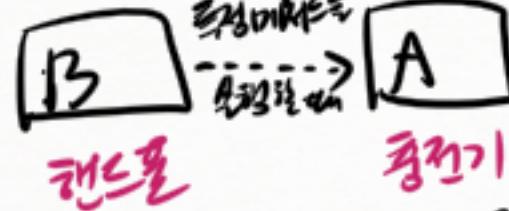
class B {
 A obj;
} ≡

④ 핍심 (composition)



class B {
 A obj;
} ≡

⑤ 의존 (dependency)



class B {
 void m(A obj){
} ≡ }

컴퓨터 ≠ 키보드
마우스
마이크

Lifecycle

컴퓨터 = $\frac{\text{CPU}}{\text{RAM}}$

Lifecycle

* String \Rightarrow 멀티 사용법.

String s1;

s1 = new String("Hello");

String s2 = new String("Hello");

String x = "Hello";

String y = "Hello";

String 풀(Pool)의
String 인스턴스는
"중복 생성하지 않는다"

JVM Stack

s1 [200]

s2 [300]

x [400]

y [500]

String Constant Pool

500 value hash
Hello —

String 풀의
인스턴트

Heap

200 Hello

300 value hash
Hello ...

String 풀의
인스턴트

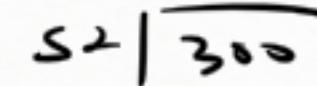
* String.intern()

String s1 = new String("Hello")

String s2 = s1.intern()

String s3 = "Hello"

JVM Stack



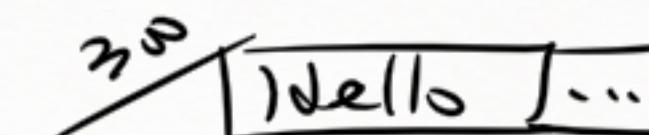
↑
s1의 주소를 문자열을 가진
String 객체는 String Pool에서 찾을 수 있는
이 생성되는 순간

문자열은
반복 사용,

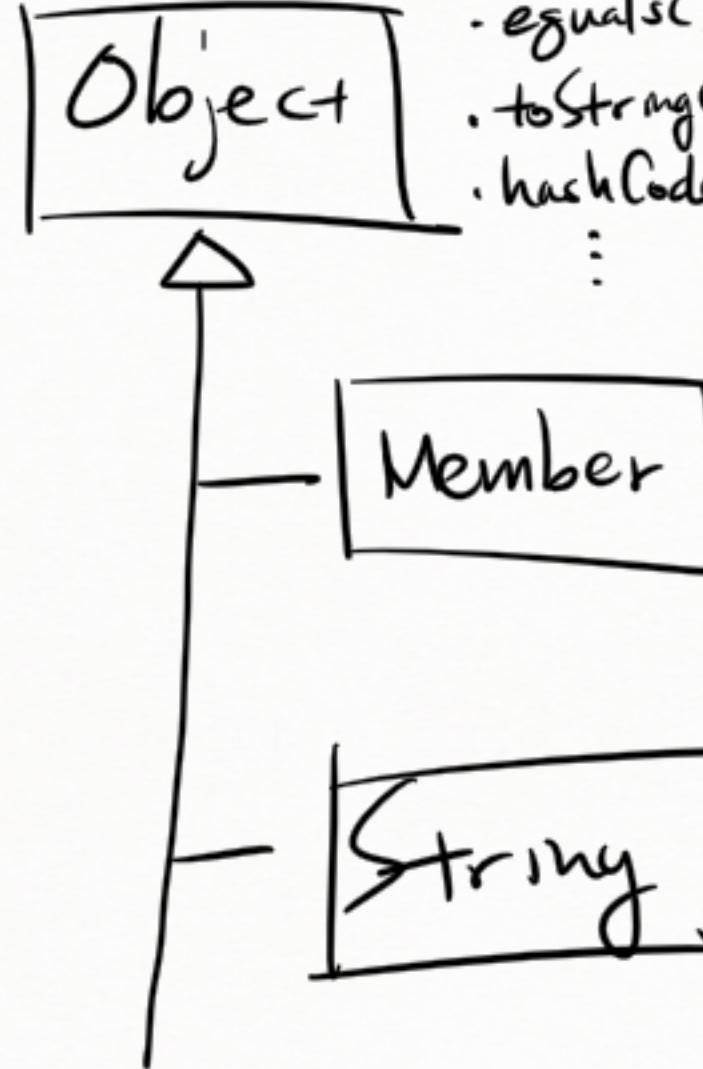
Heap



String Pool



* Object.equals()



- equals() ← 인스턴스 주소가 같은지 비교 → == 연산자와 동일하게
- toString() ← "클래스명@주소값" 리턴
- hashCode() ← 랙킹 번호 리턴

• equals() 오버라이드
↳ 문자열 비교

Member m = new Member();

m.toString(); ← Object

m.equals(); ← Object

m.hashCode(); ← Object

String s1 = new String("Hello");

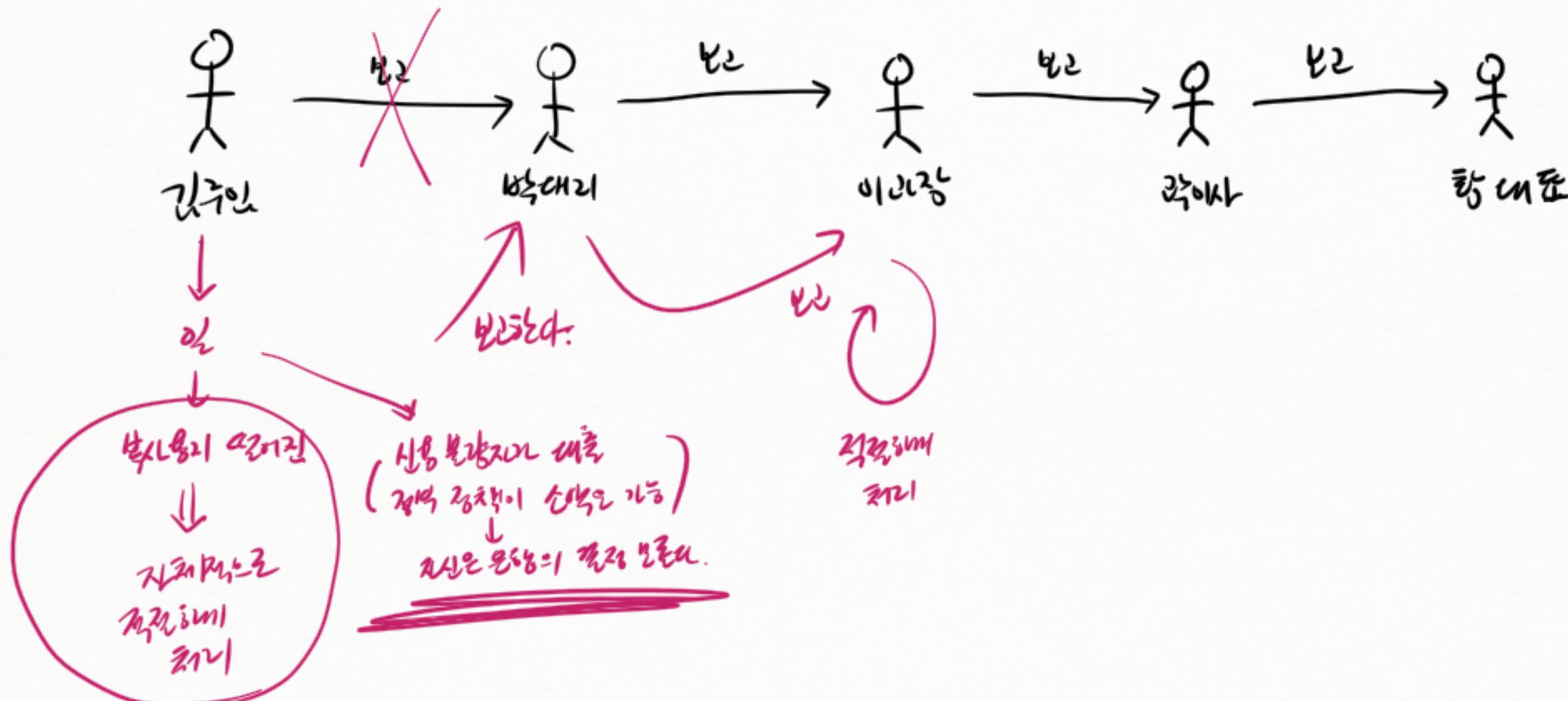
String s2 = new String("Hello");

s1 == s2 ⇒ false

s1.equals(s2) ⇒ true

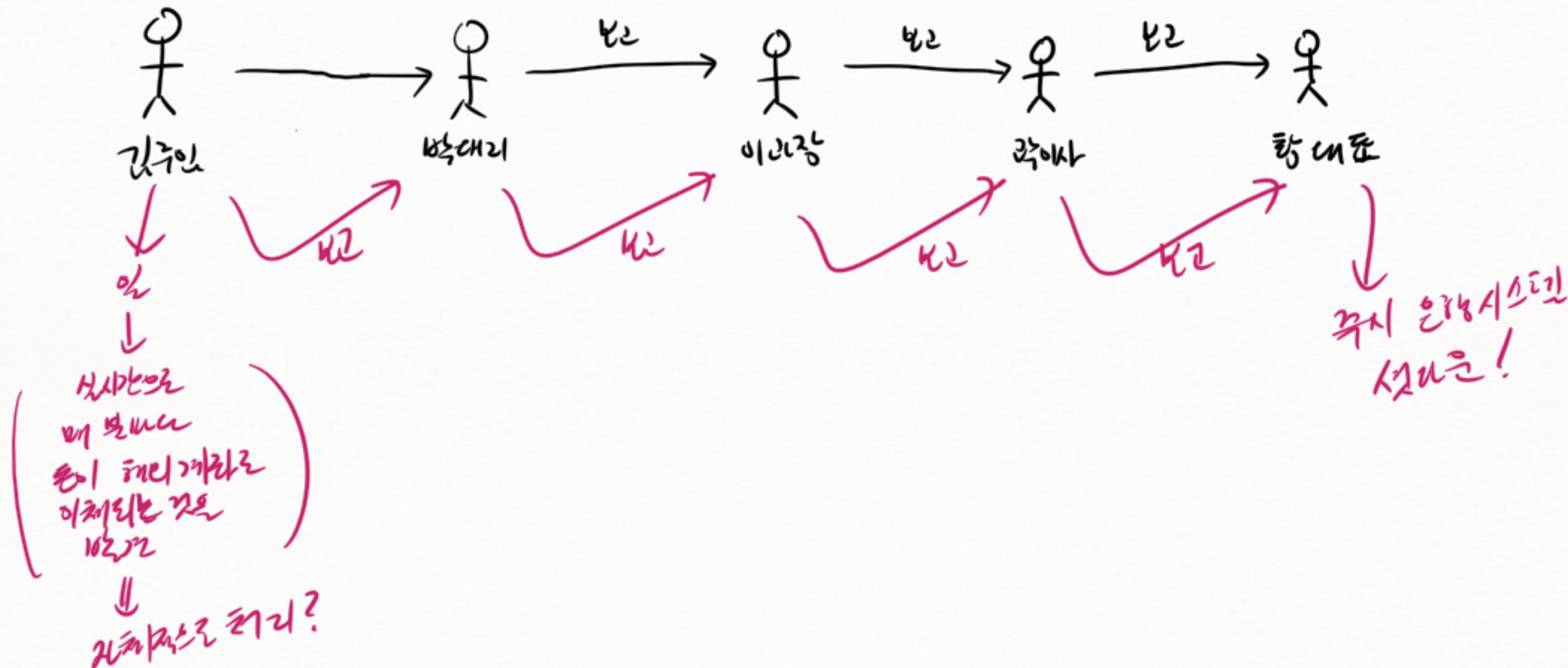
* 예외 처리 : 발생 → 보고 → 처리

① 단순 예외 흐름



* 예의 차리 : 1발생 → 보고 → 차리

② 치매환자의 상황

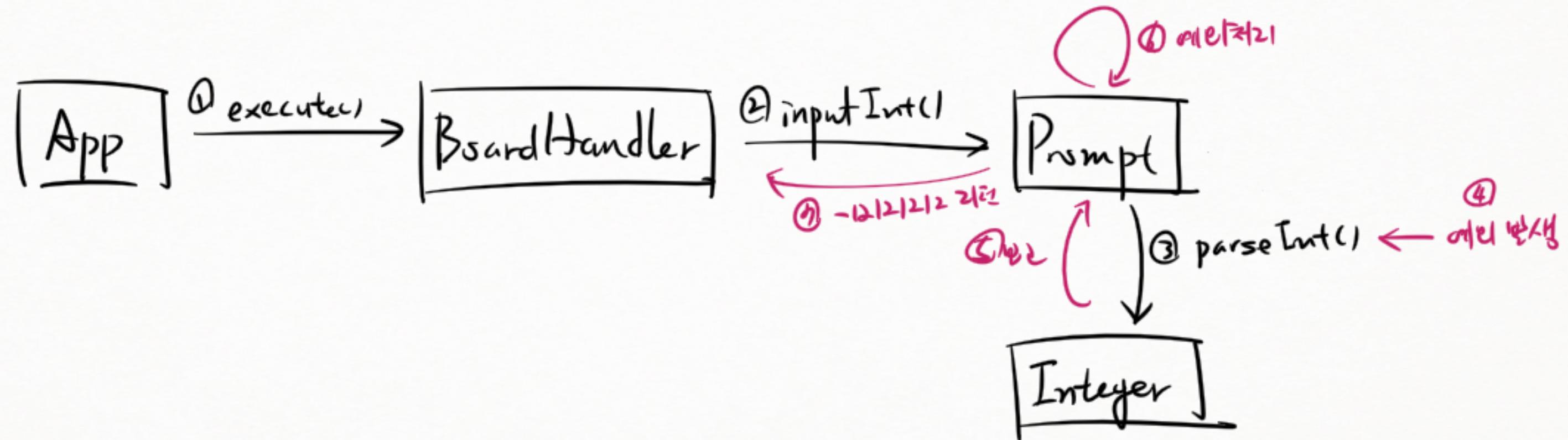


* 예외 처리 풀기

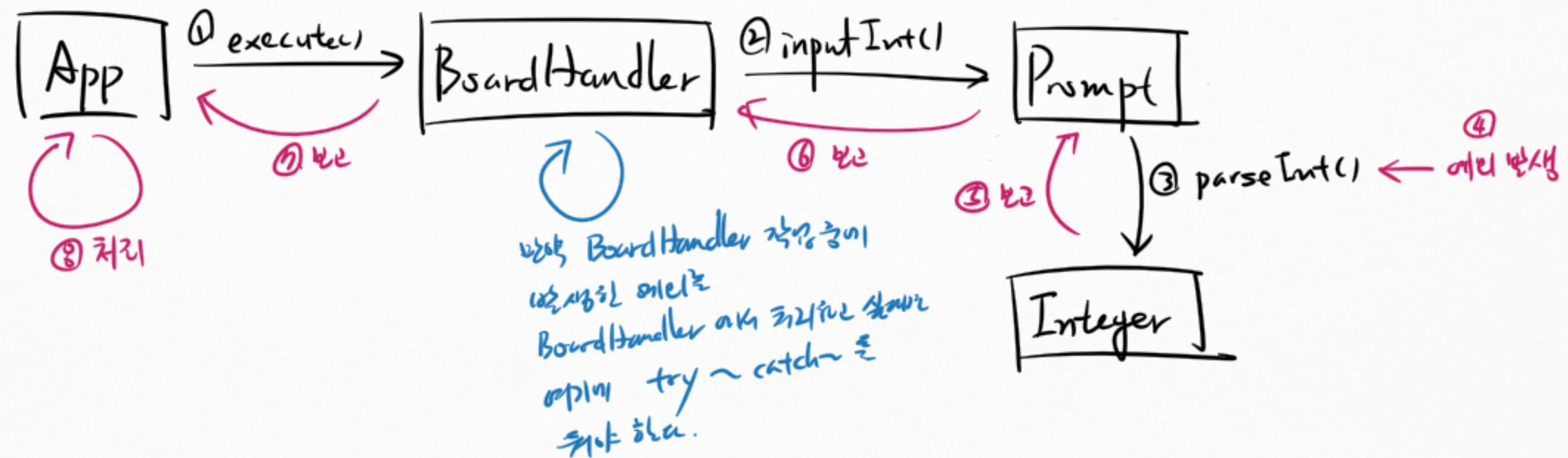
```
try {
    예외 발생할 때 실행
} catch (예외체류 타입을 다루는 것) {
    예외 처리
}
```

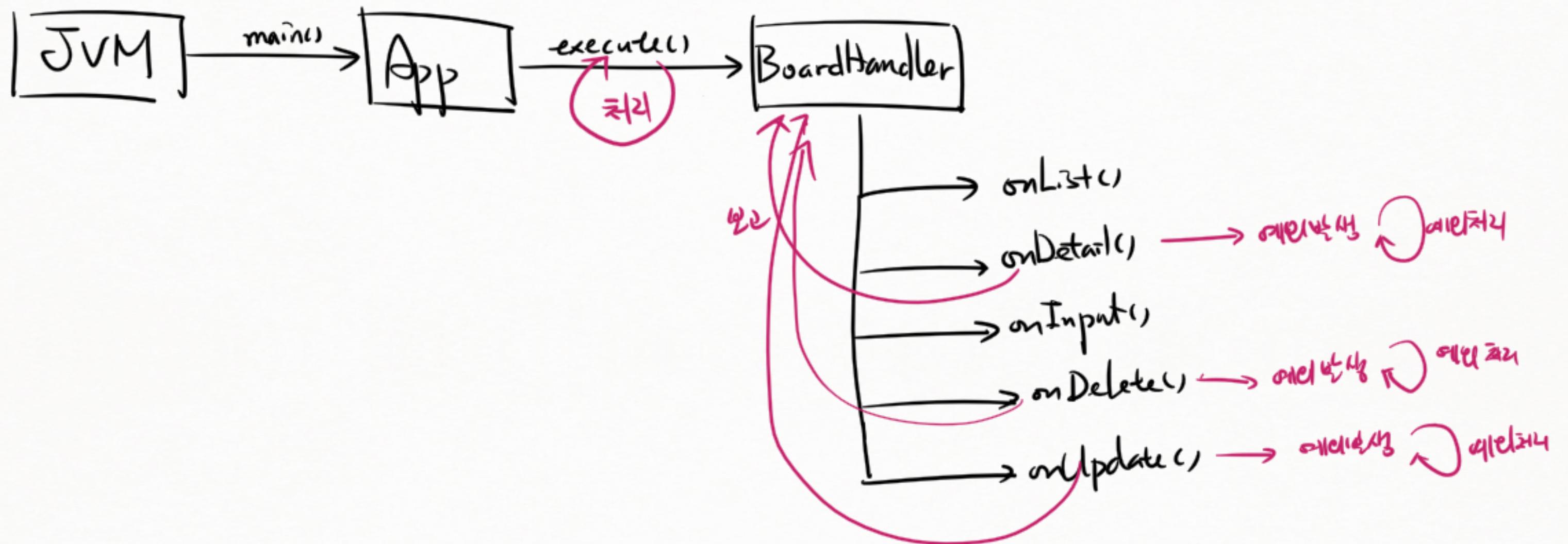
→ java.lang.Throwable

* 016. 예외 처리 1

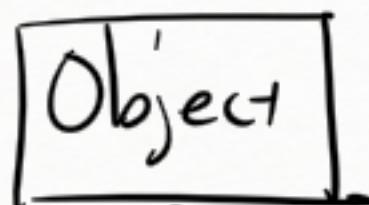


* 017. 예외 처리 2

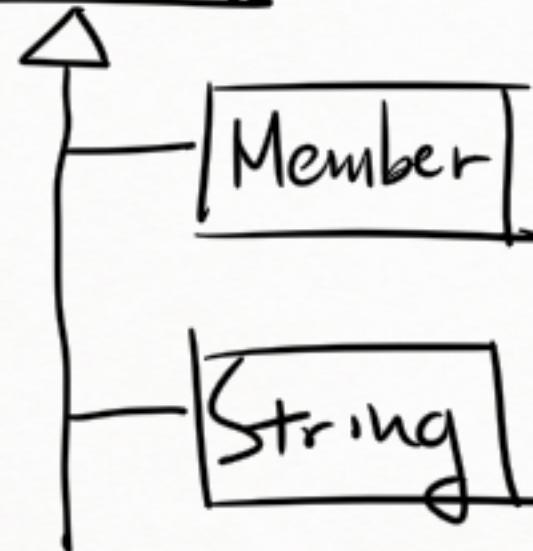




* 오버라이딩 메서드 흐름



· `toString()` ⇒ "제작된 문자열"



· `toString()` ⇒ "저장된 문자열"

"
컨타クト
오류"
↳

컨타クト는
별도의 타입이
존재하고
있기 때문에
호출 가능.
* 변수에 어떤 객체
저장되었을 때
호출 가능.

`Object x = new String("Hello");`
`x.charAt(0);` ← 컨타クト 오류!
`x.toString();` ← 컨타クト OK!

↳ JVM은 레퍼런스가 실제 가리키는
클래스에서부터 메서드를 찾아올라간다. ⇒ 틀렸어! String>!

`Object obj;`

`obj = new Object();`

`obj.toString();` ← Object

`Member m = new Member();`

`m.toString();` ← Object

`String s = new String("Hello");`

`s.toString();` ← String

018. 예외 상황 예상하기

throw

예외를 발생시키기;



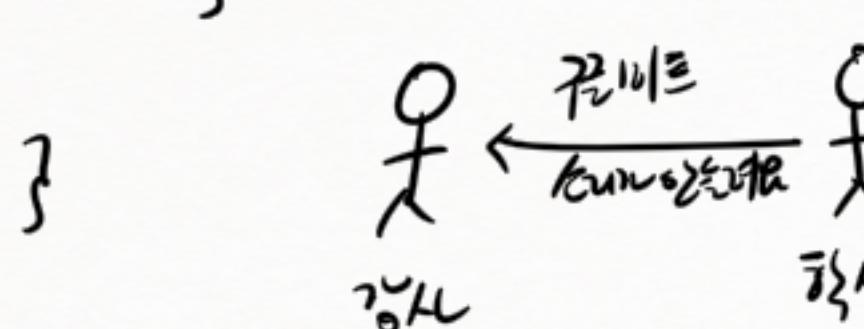
java.lang.Throwable 예상

내부 예상 가능!

x 예외 처리 방법

① 예외 처리

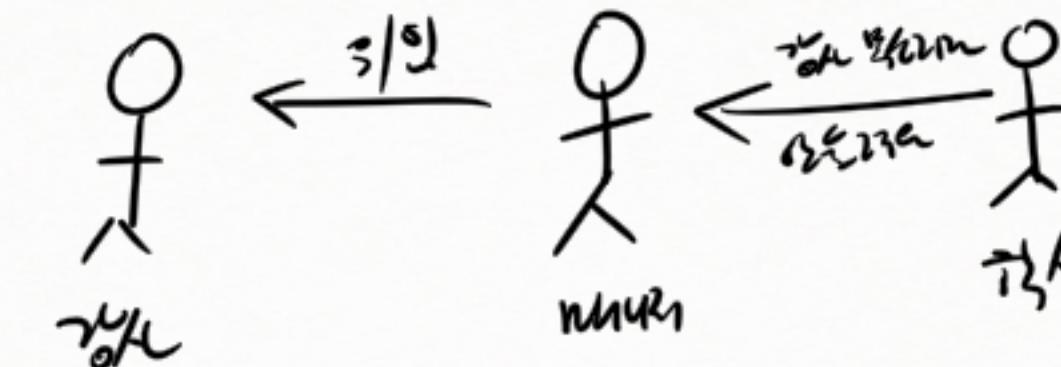
```
void m() {  
    try {  
        // 예외를 발생시킬 코드  
    } catch (Throwable ex) {  
        예외처리 코드  
    }  
}
```



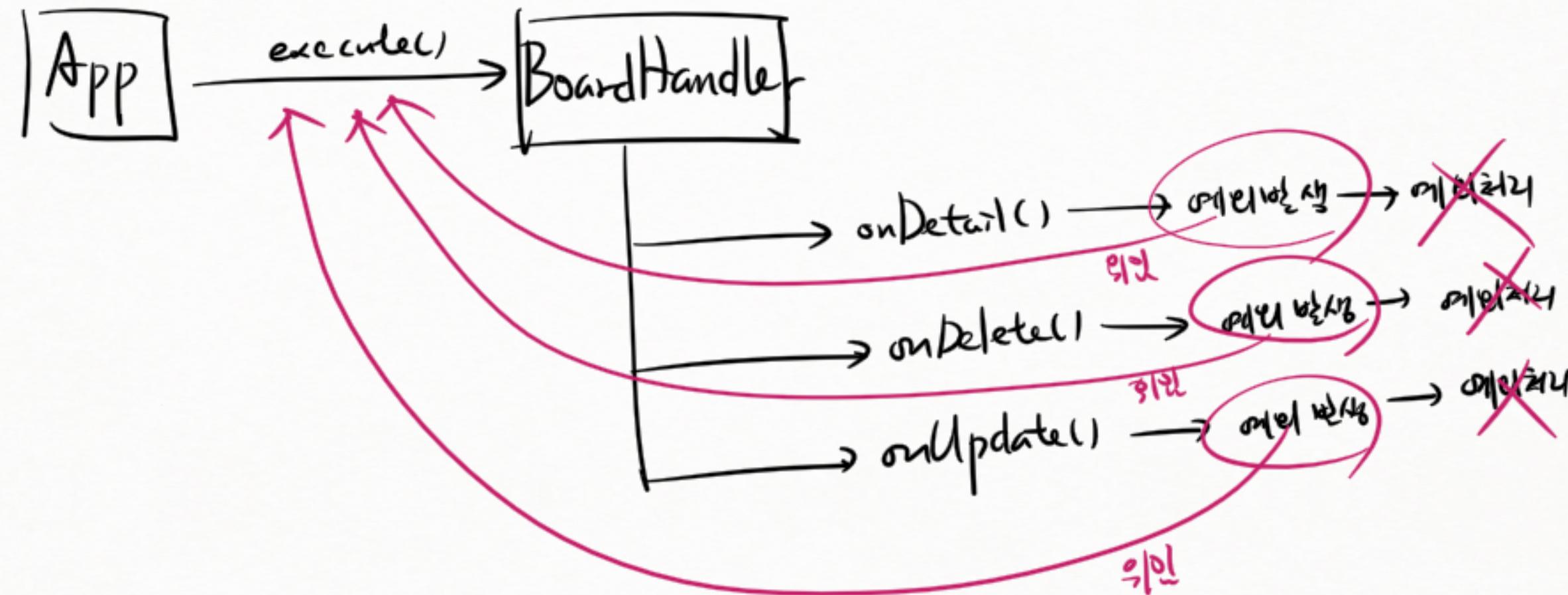
예외 처리

② 예외

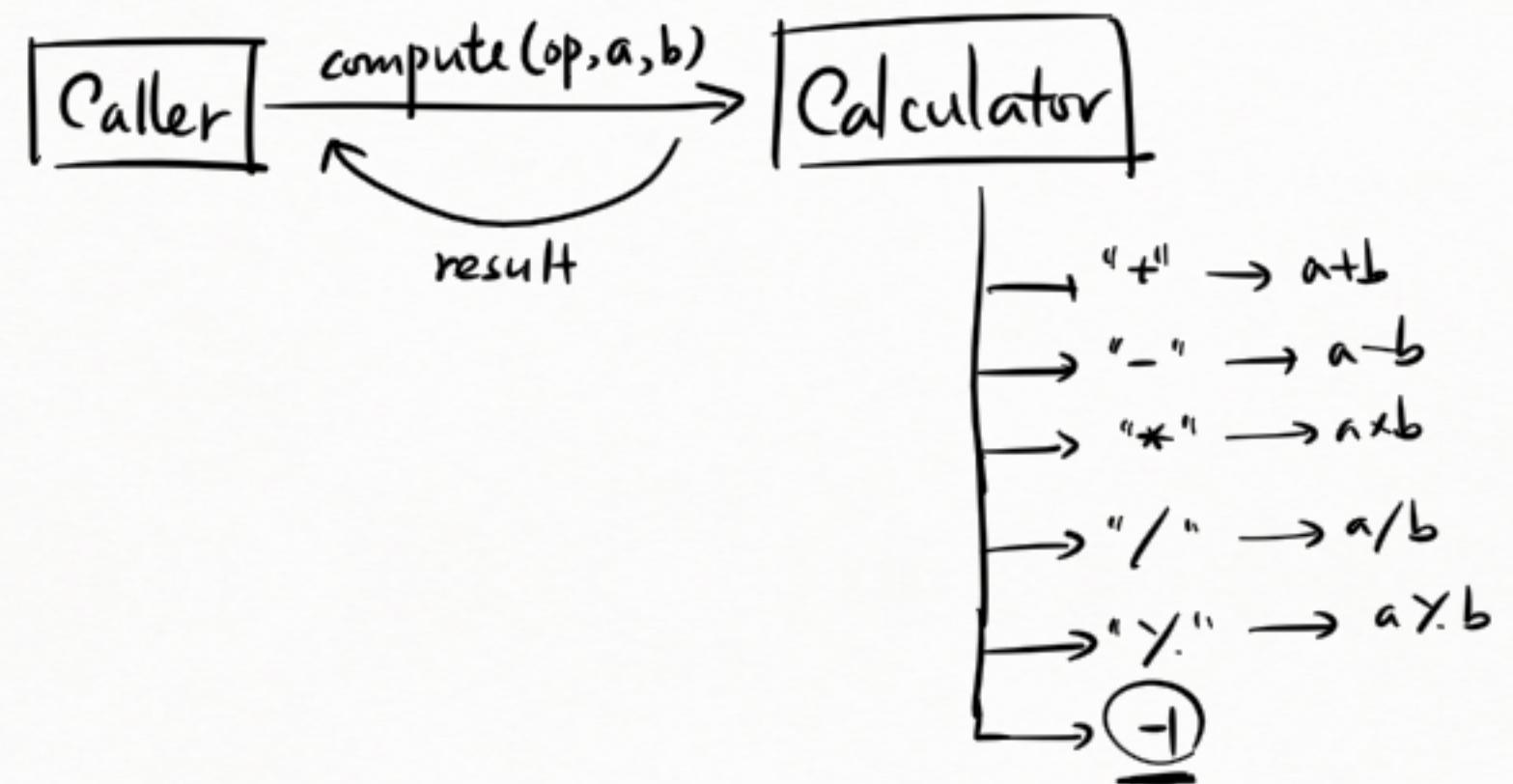
```
void m() throws 예외인정 {  
    예외를 발생시키는 코드  
}  
}
```



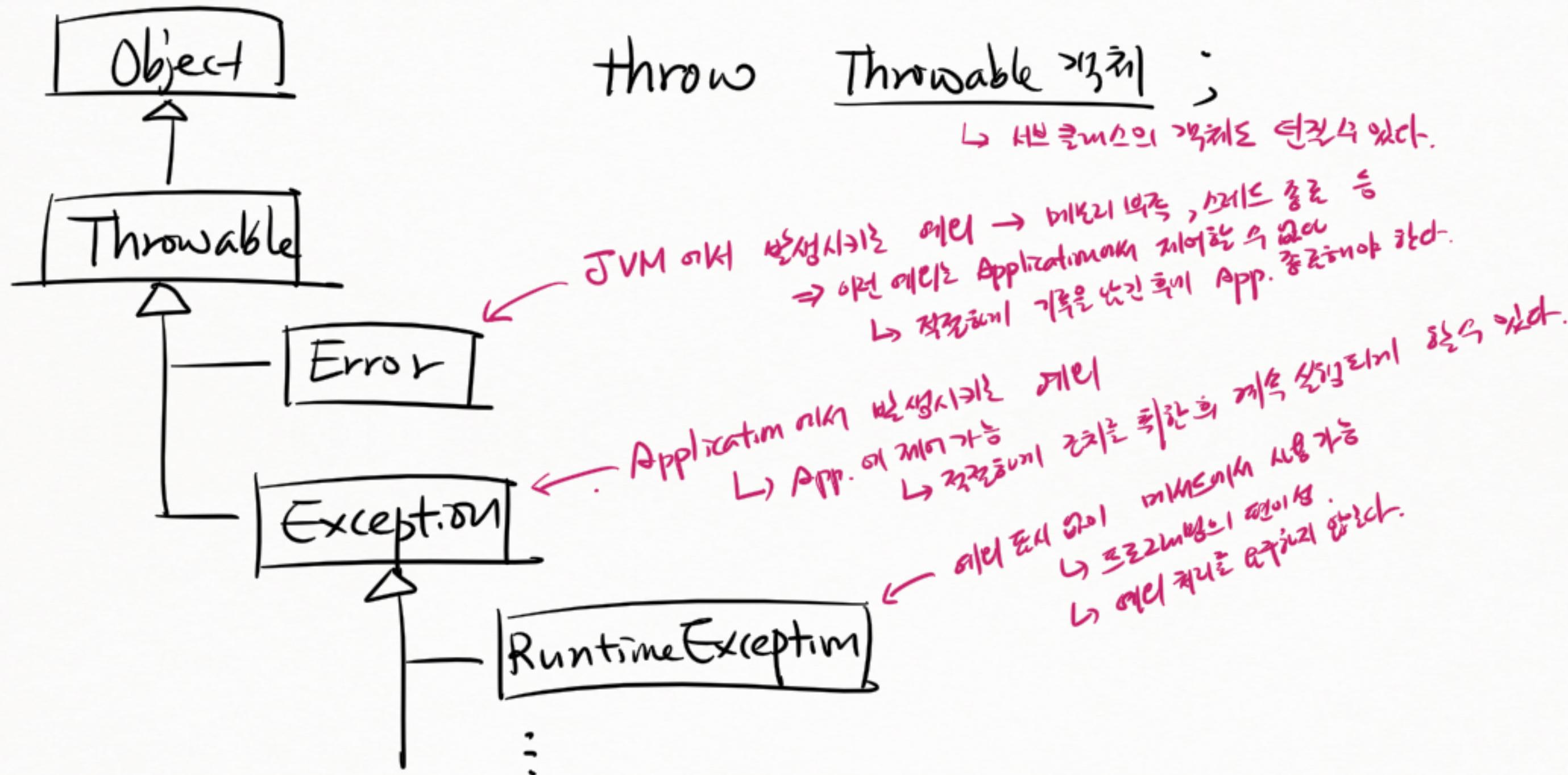
예외는 처리하기 어렵고,
예외 처리는 어렵다.
예외를 처리하는 데에
m()을 호출하는 주제가
자기 자신에게 책임지다.



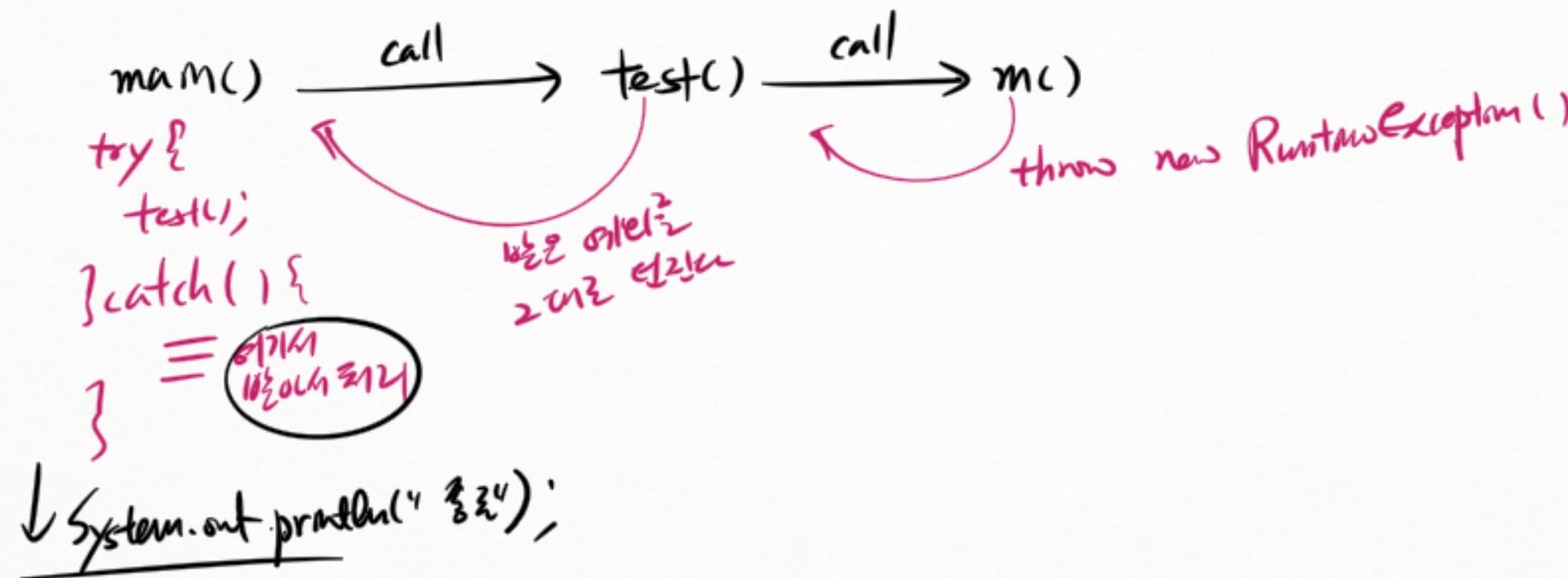
* 예외 상황을 이런 식으로 알리기



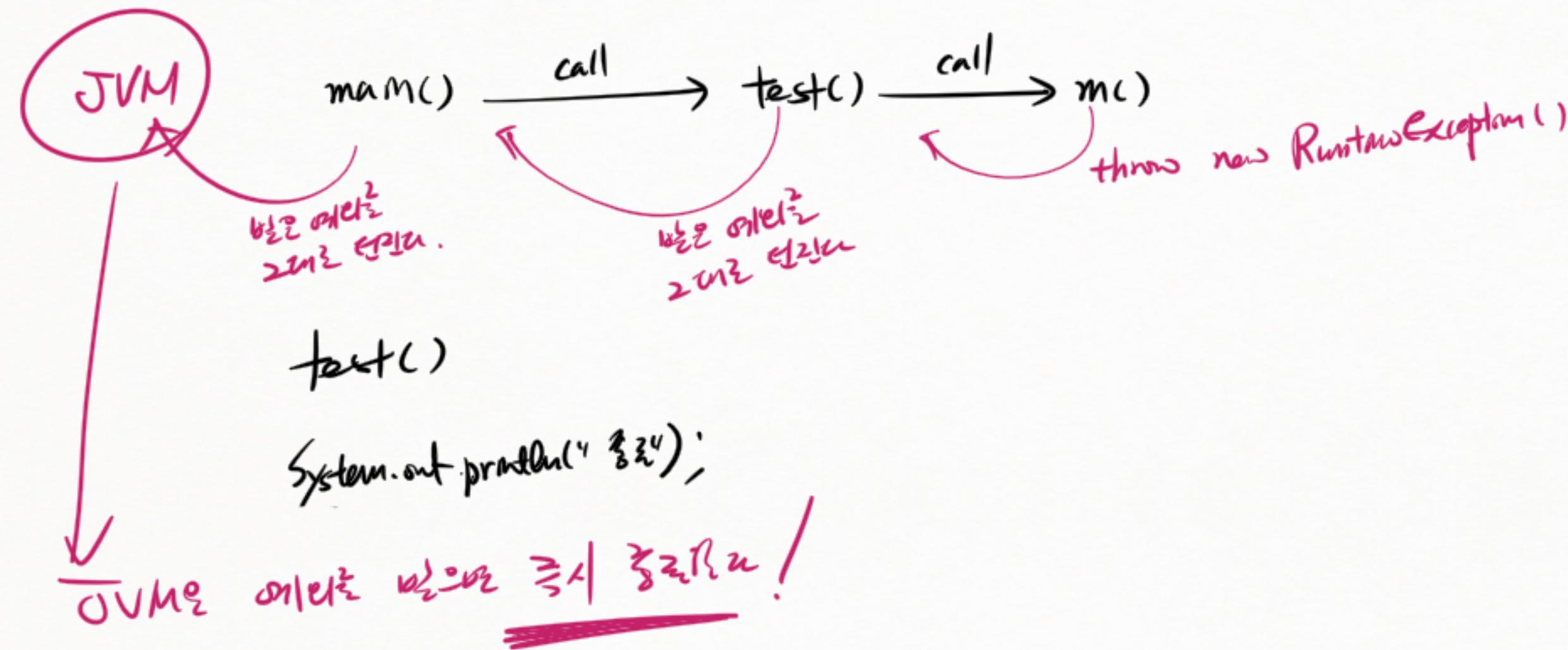
* 예외 던지기



* 예외 처리

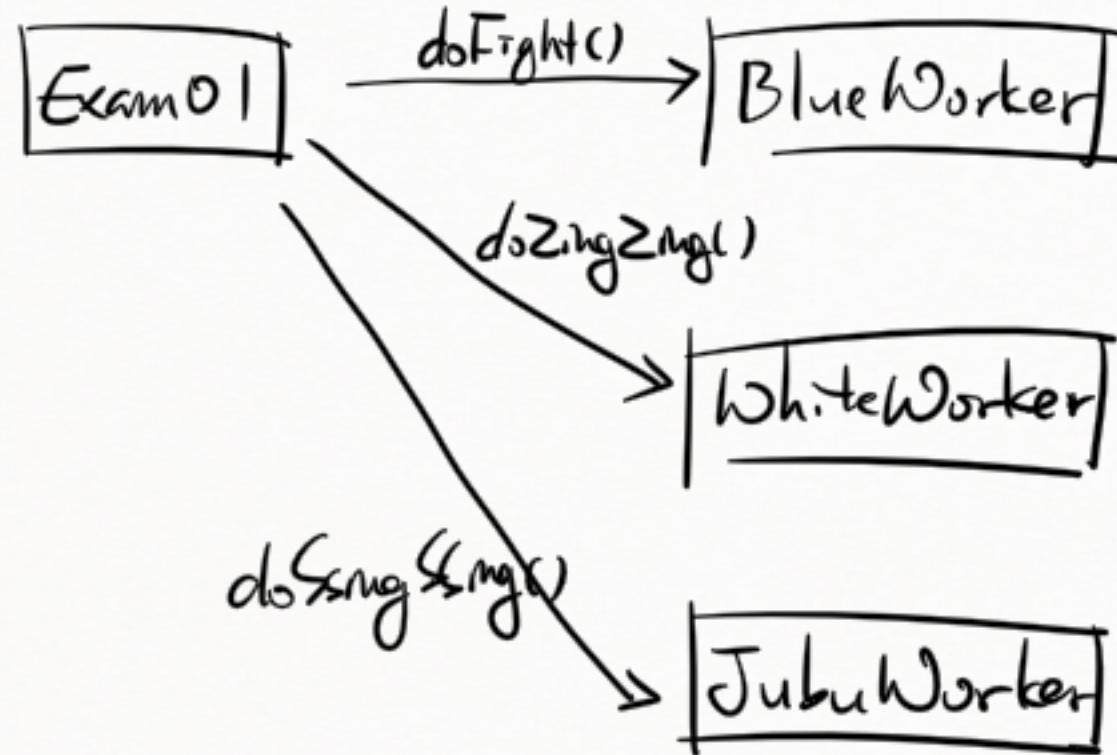


* 멤버 속성

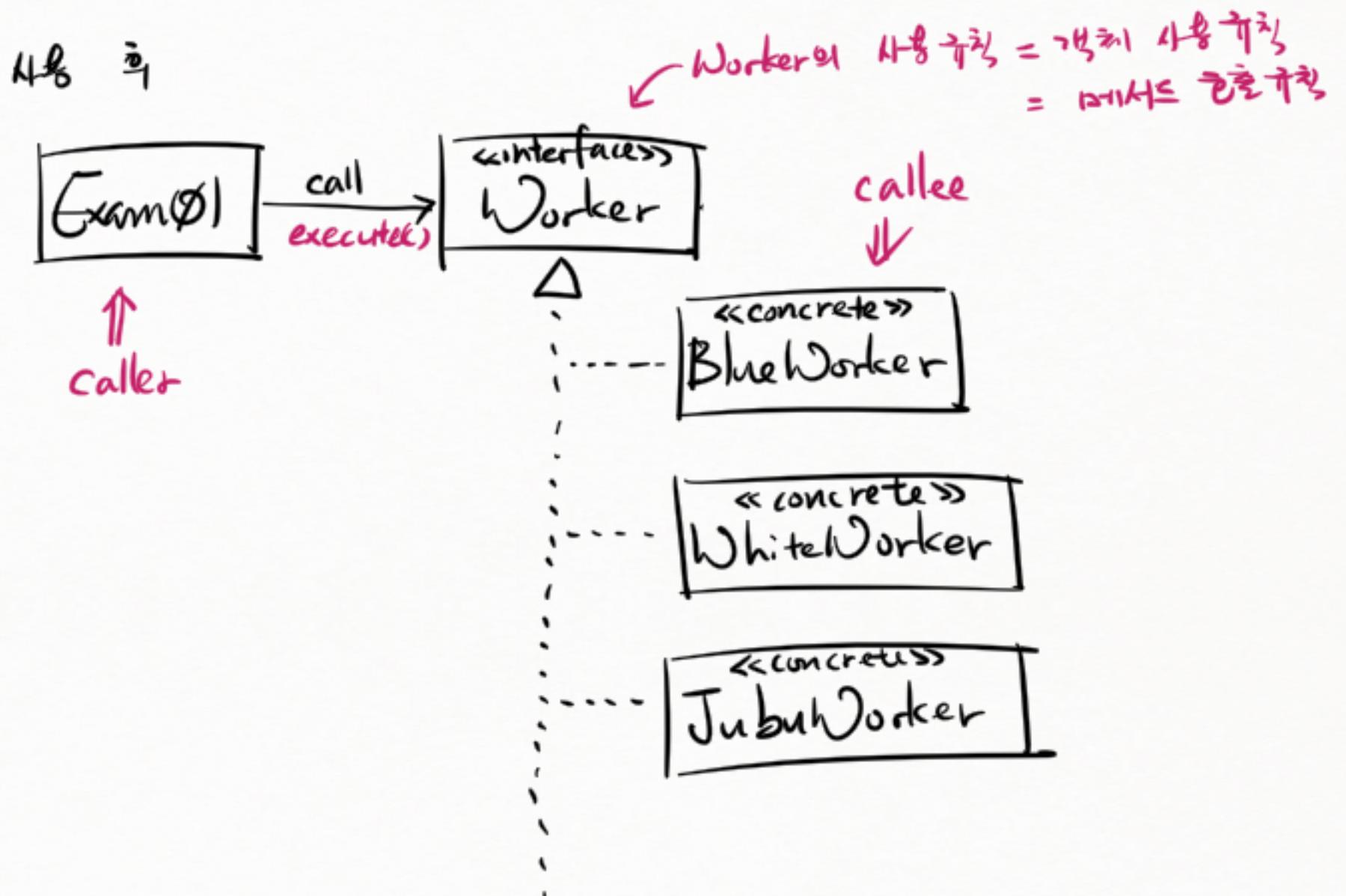


* 인터페이스

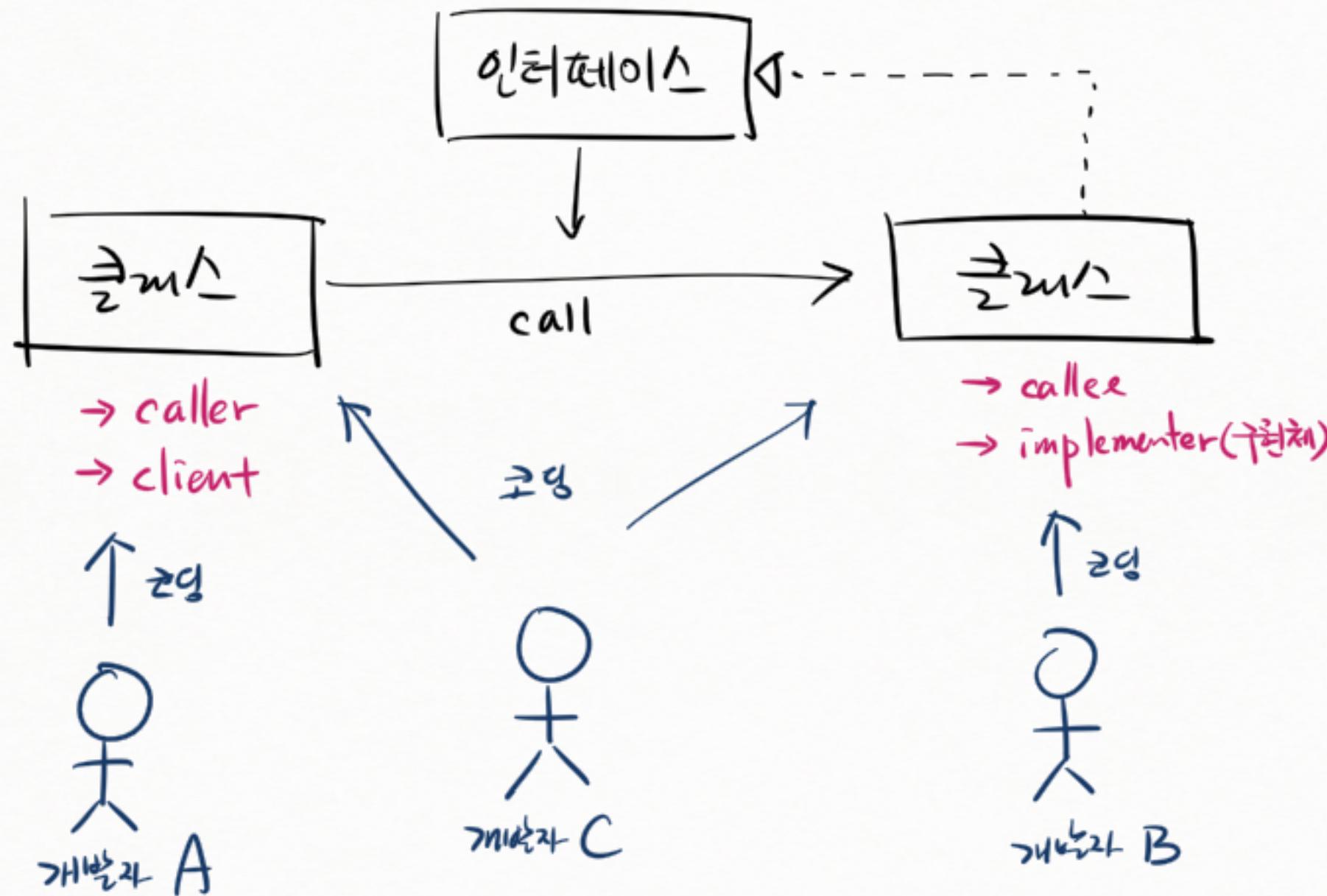
① 사용 전



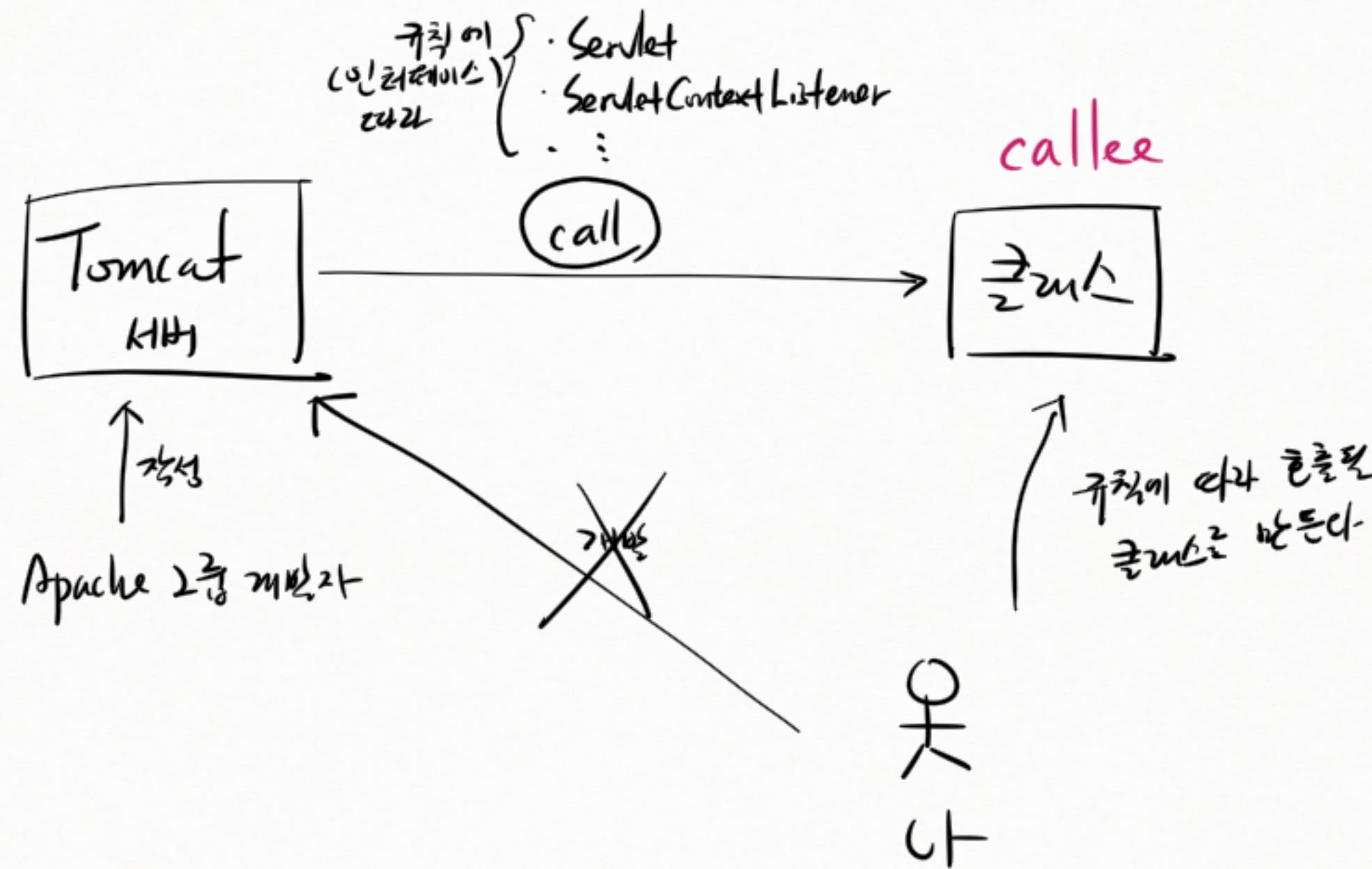
② 사용 후



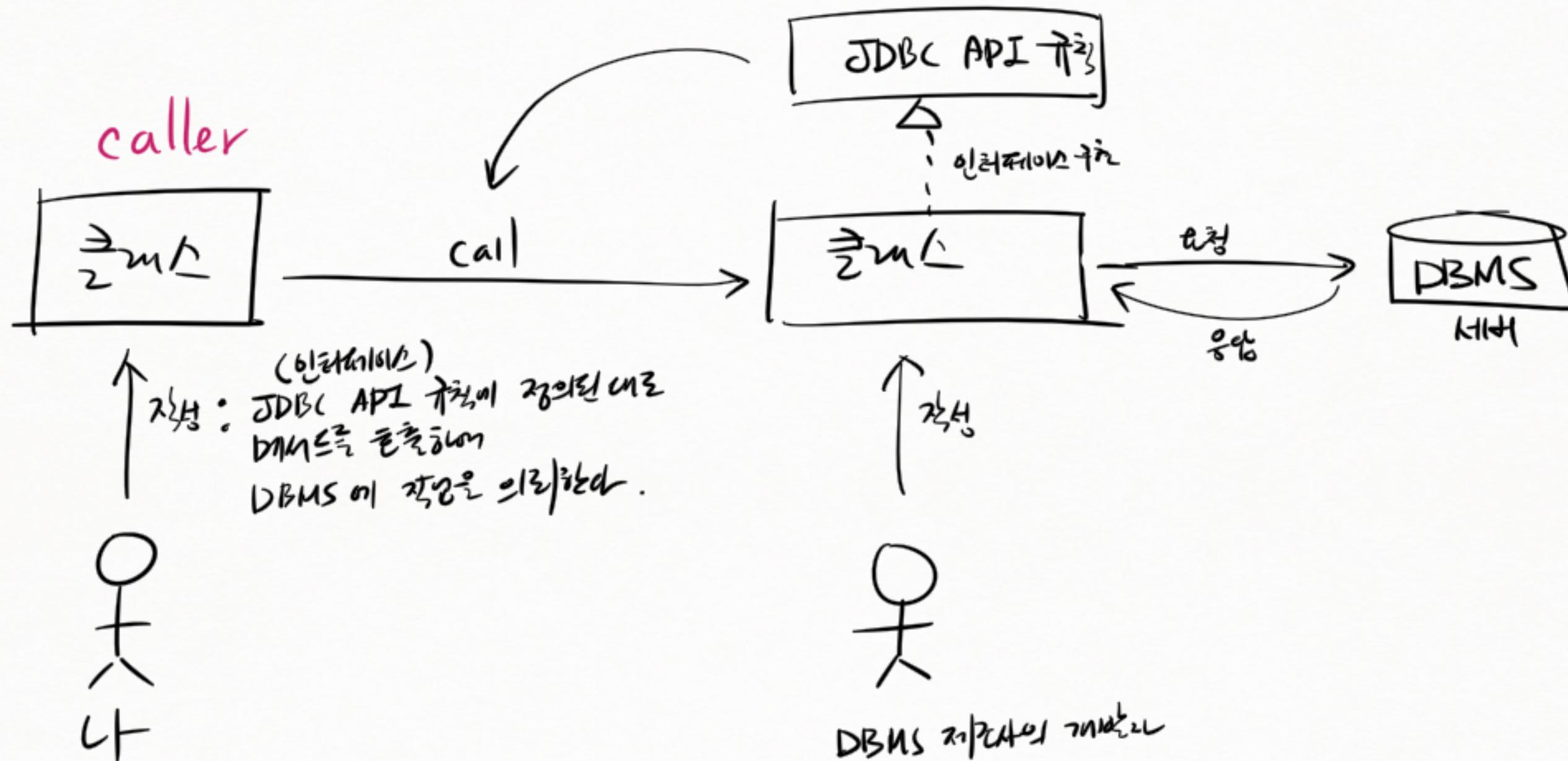
* 인터페이스와 구드 작성



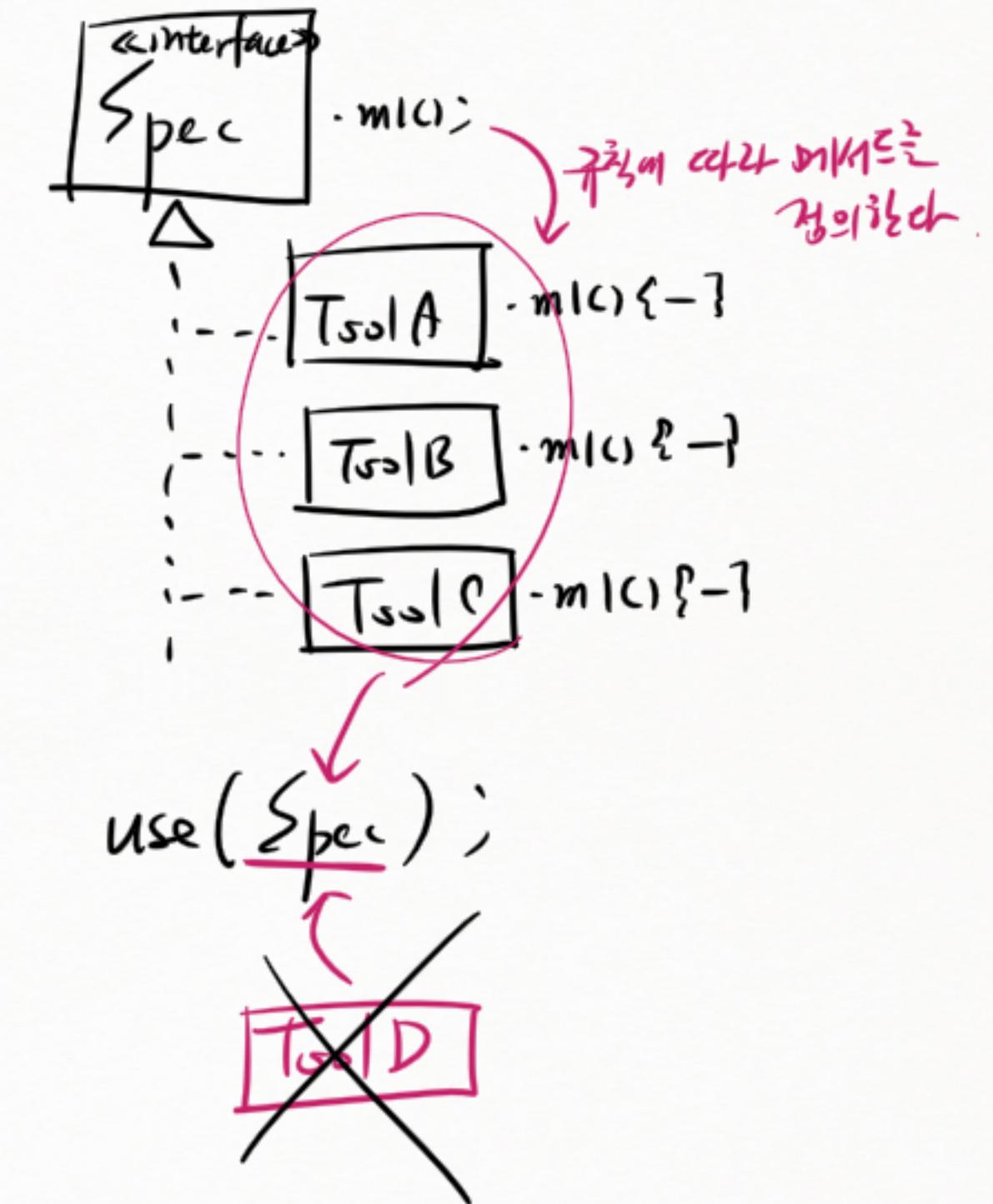
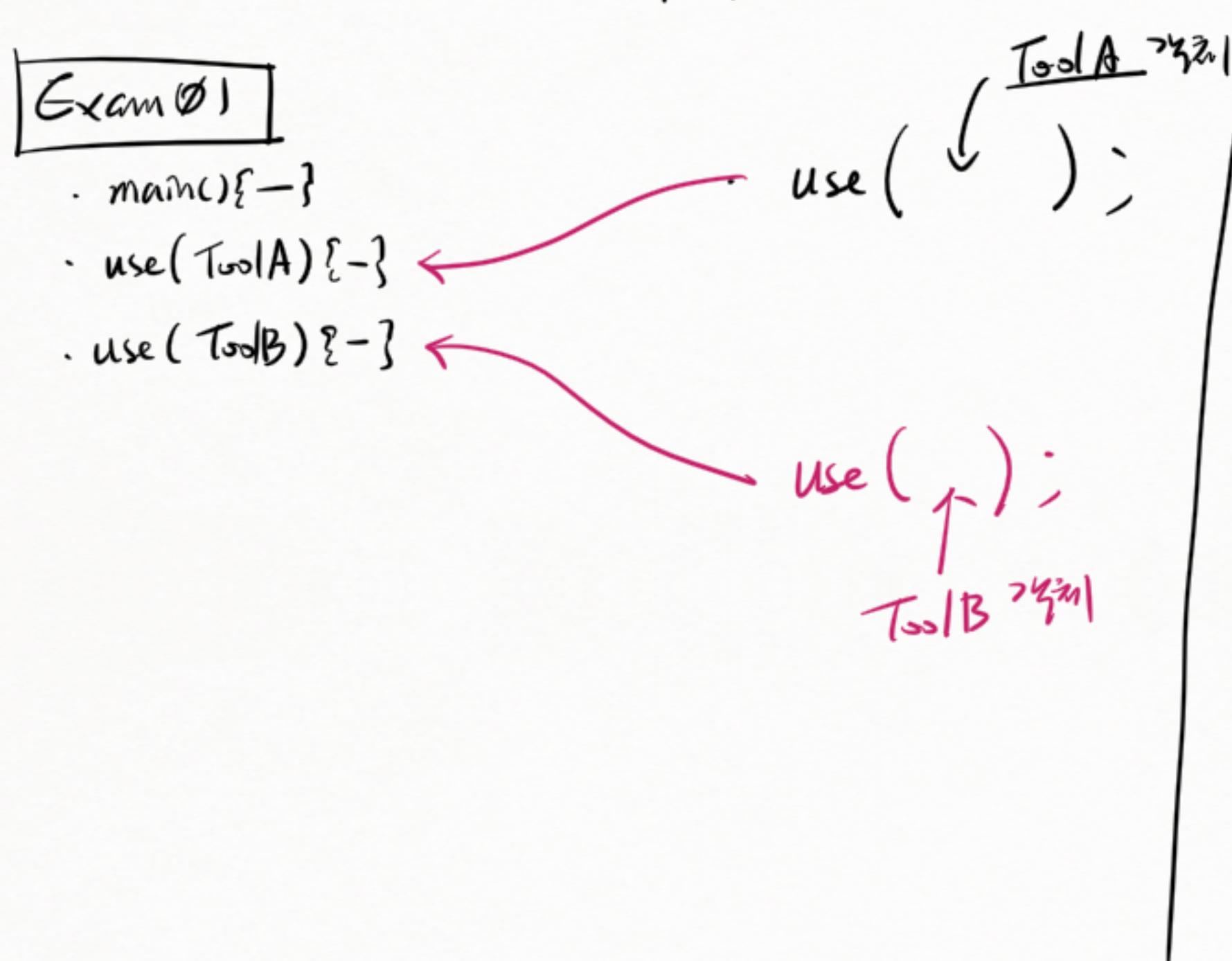
* 인터페이스와 키워드 설명



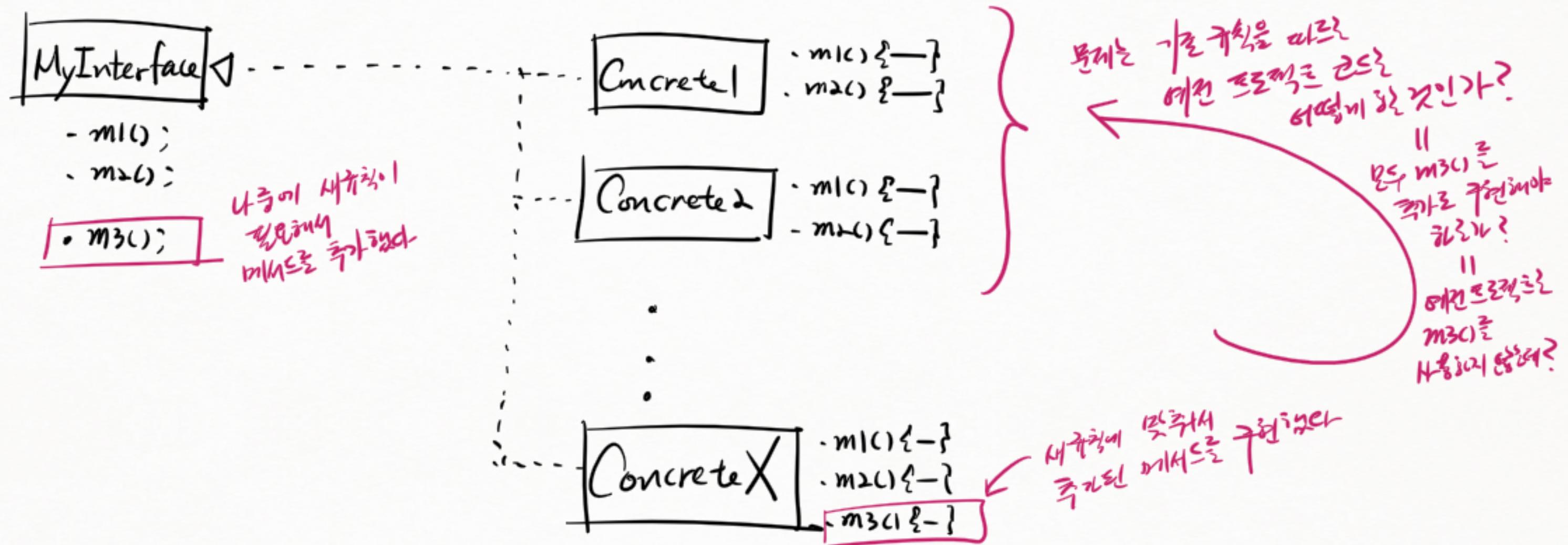
* 인터페이스와 JDBC 구조



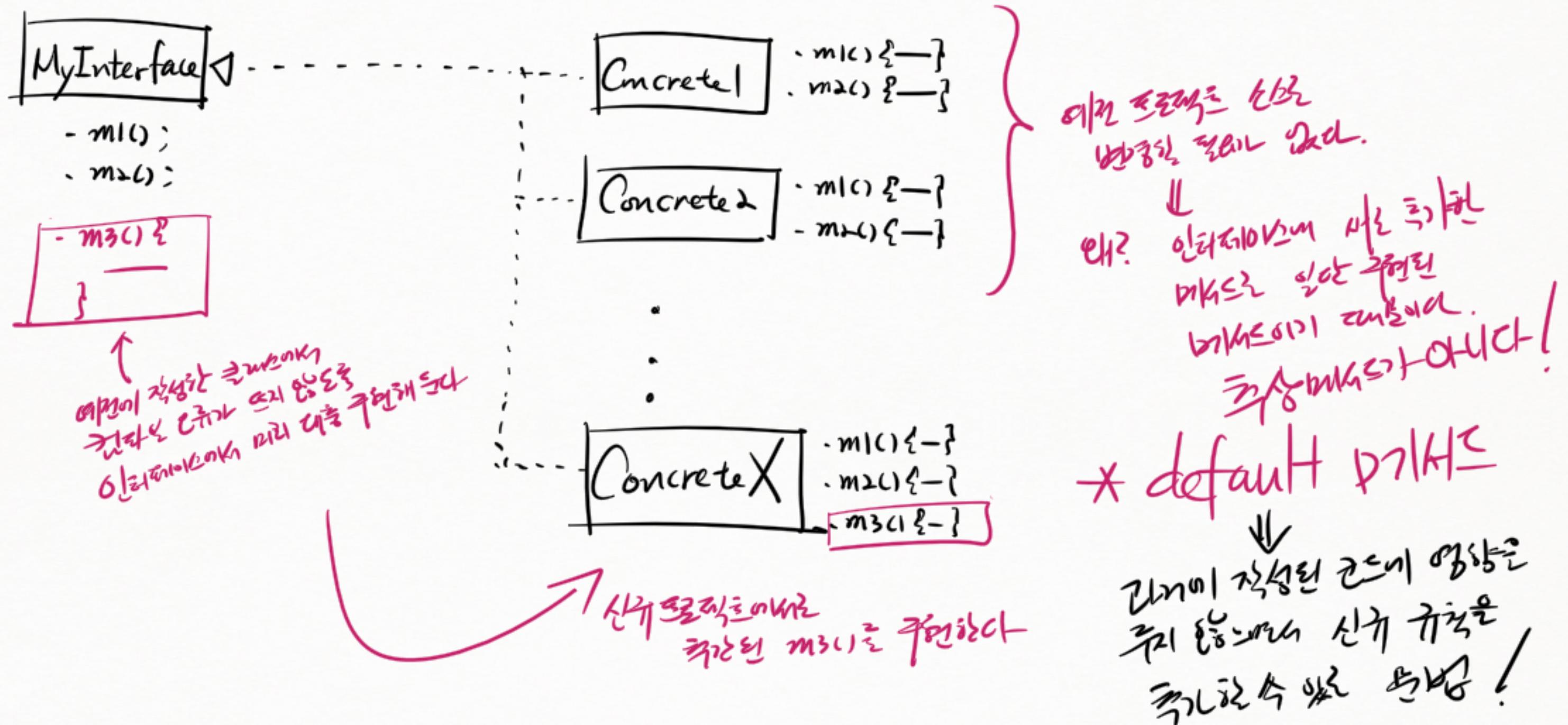
* 디자인 패러다임과 인터페이스



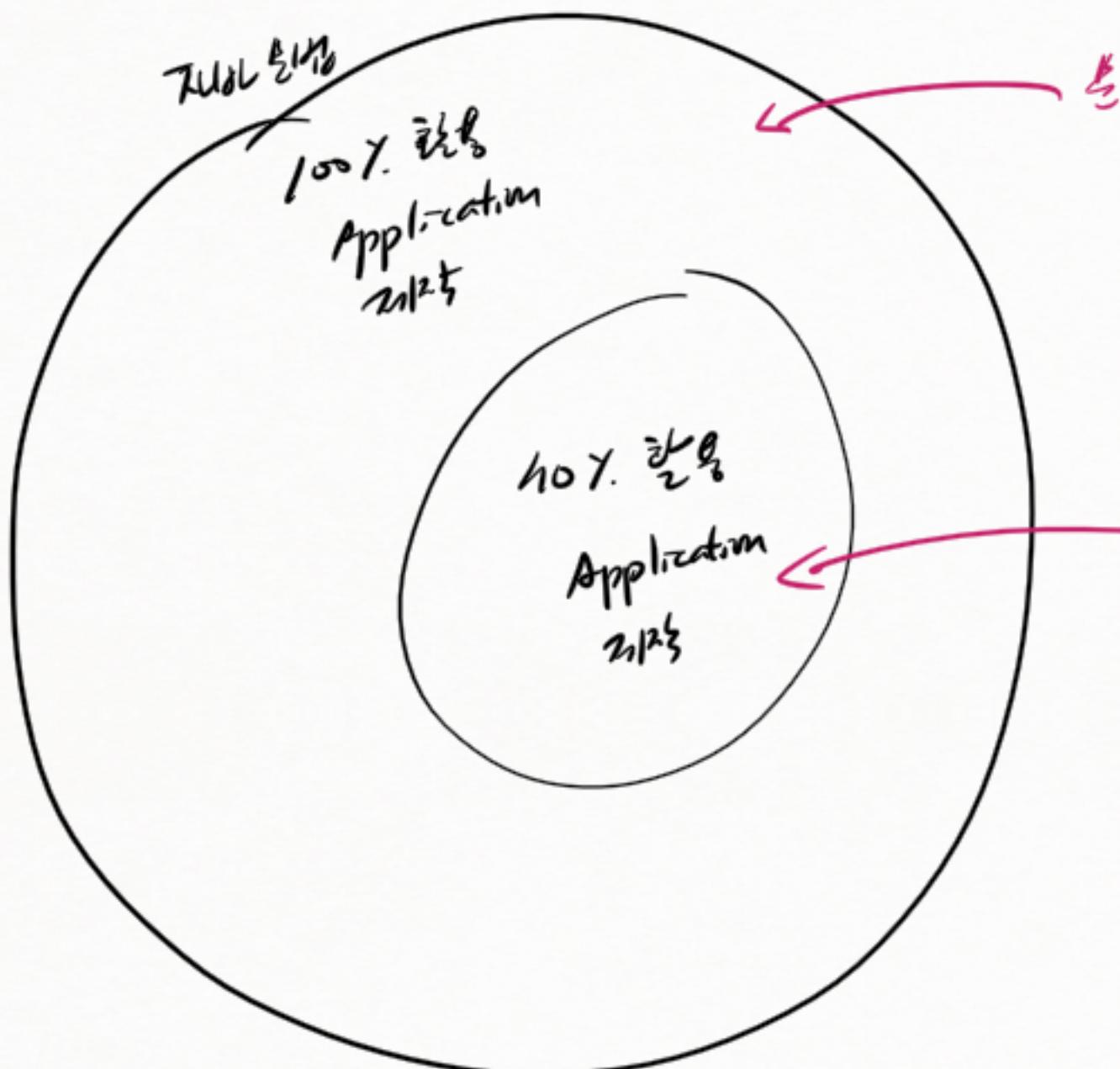
* default 인터페이스 등장 전



* default 디자인 등장 후



* 자바 문법의 활용

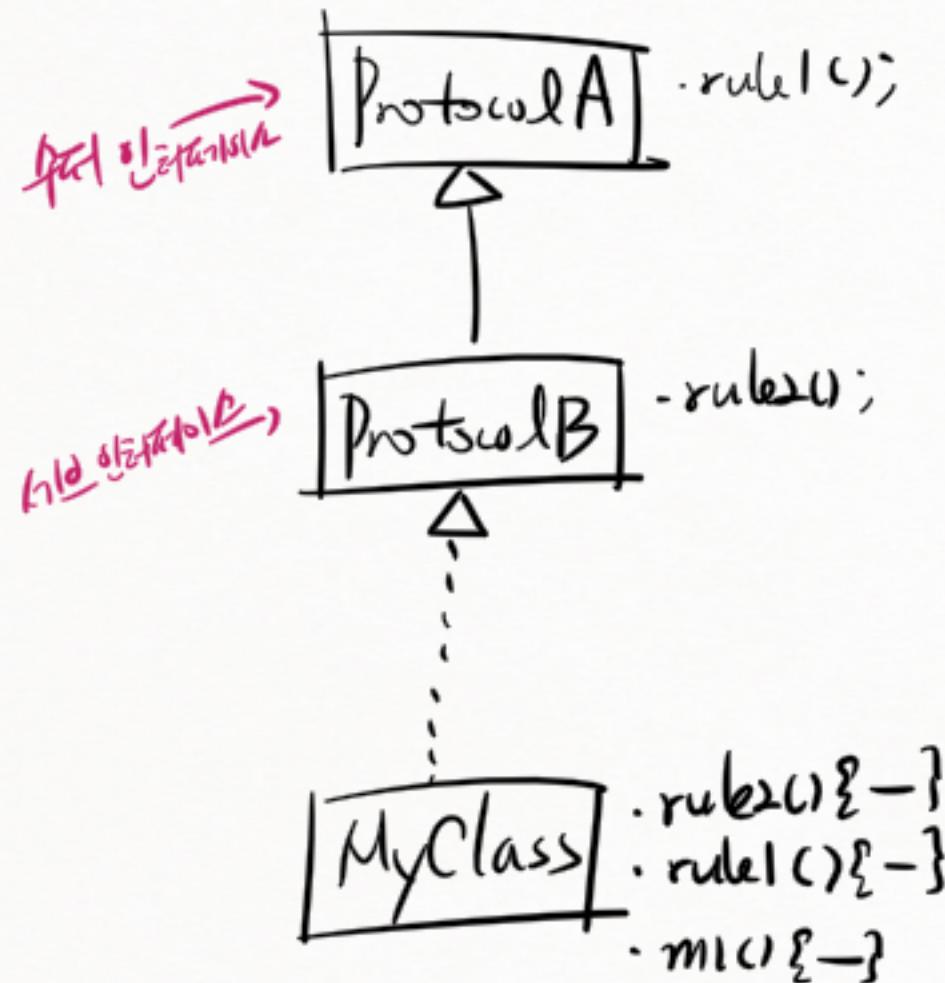


문법을 100% 활용해보자!
수행되는 코드를 보면 이해하기 쉽다!

문법을 활용하면 → 코드를 양보

대부분의 수행되는 코드에는 문법이
있음을 알면 → 유리한 편이다.

* 인터페이스의 상속과 구현



MyClass obj = new MyClass();
 obj.m1();
 obj.rule1();
 obj.rule2();

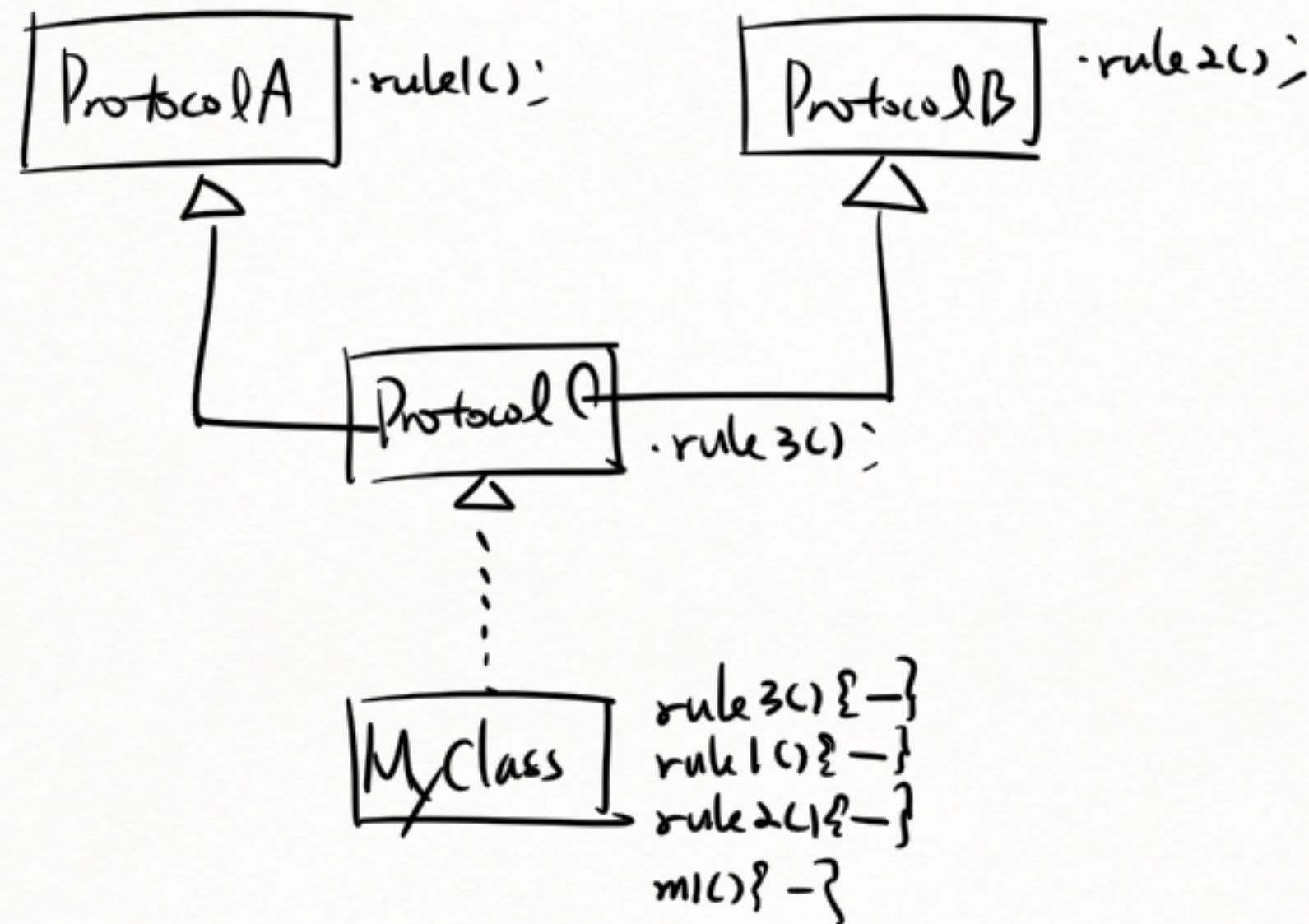
ProtocolB obj2 = obj;
 obj2.rule2();
 obj2.rule1();
 obj2.m1();

ProtocolB에 선언된
메서드가 아니다.

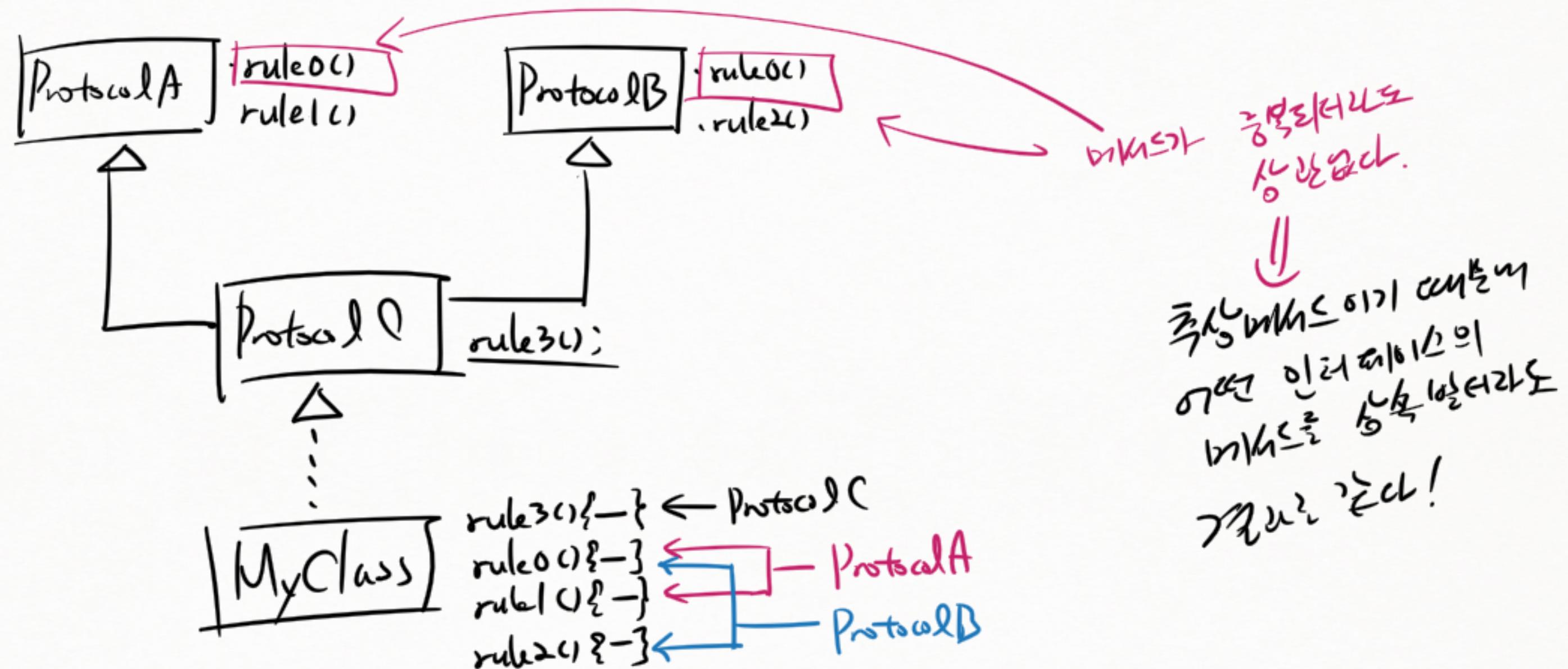
ProtocolA obj3 = obj;
 obj3.rule1();
 obj3.rule2();
 obj3.m1();

} ProtocolA의
m1()가 출력됨.

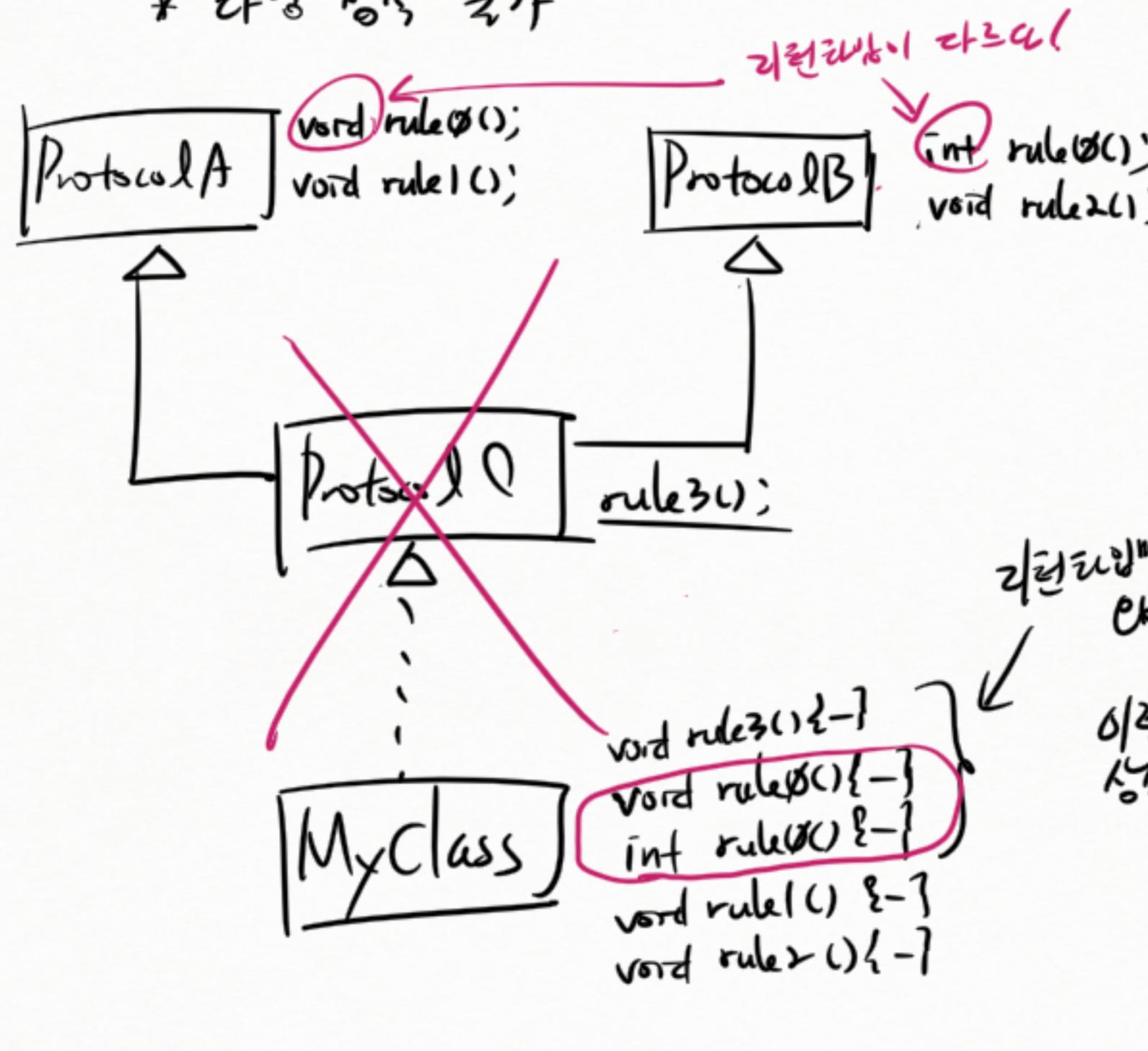
* 인터페이스의 상속과 구현 II



* 다른 상속과의 차이



* 다른 상속 불가

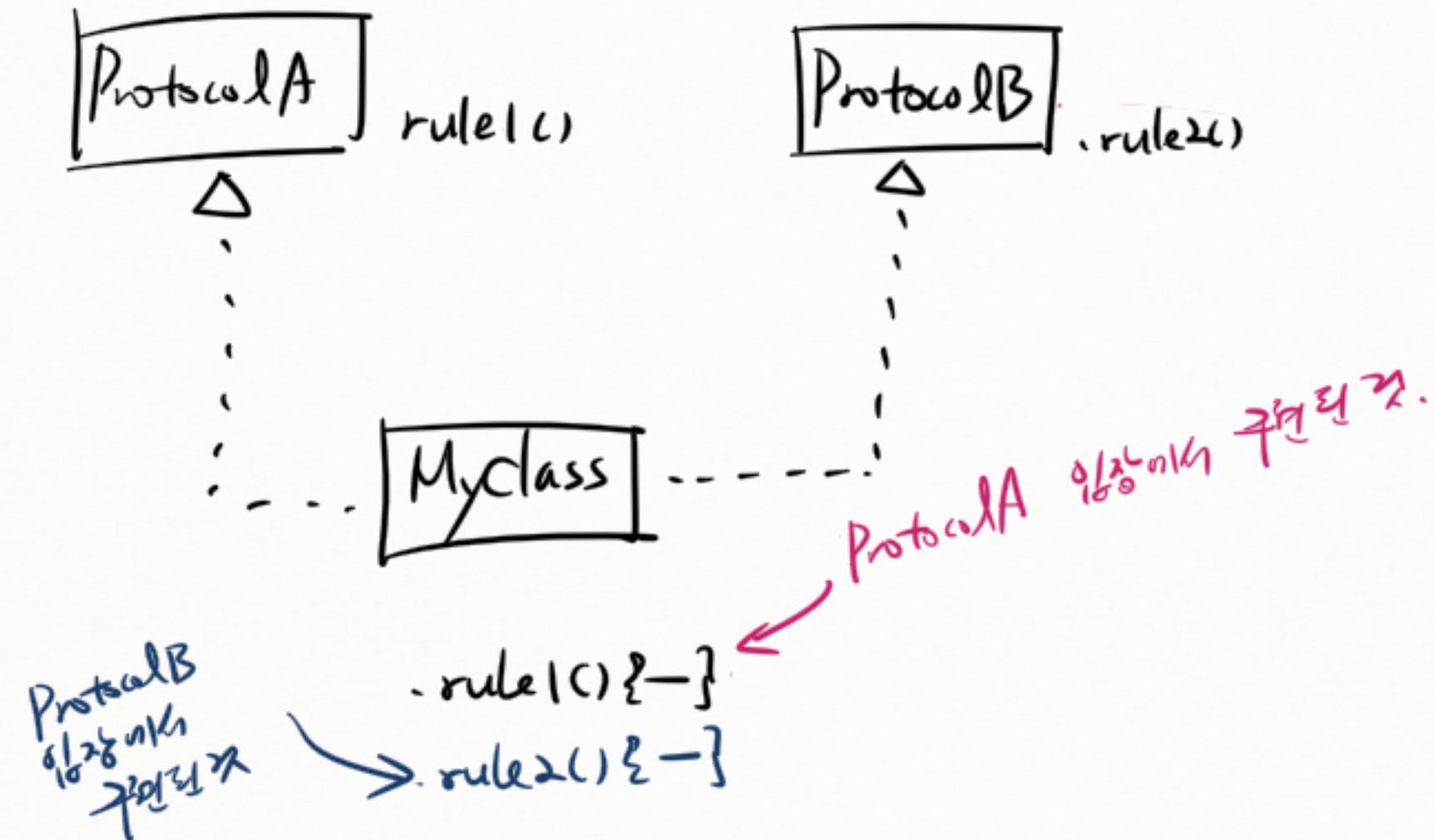


리턴 타입이 다른 경우
상속 불가능하다.

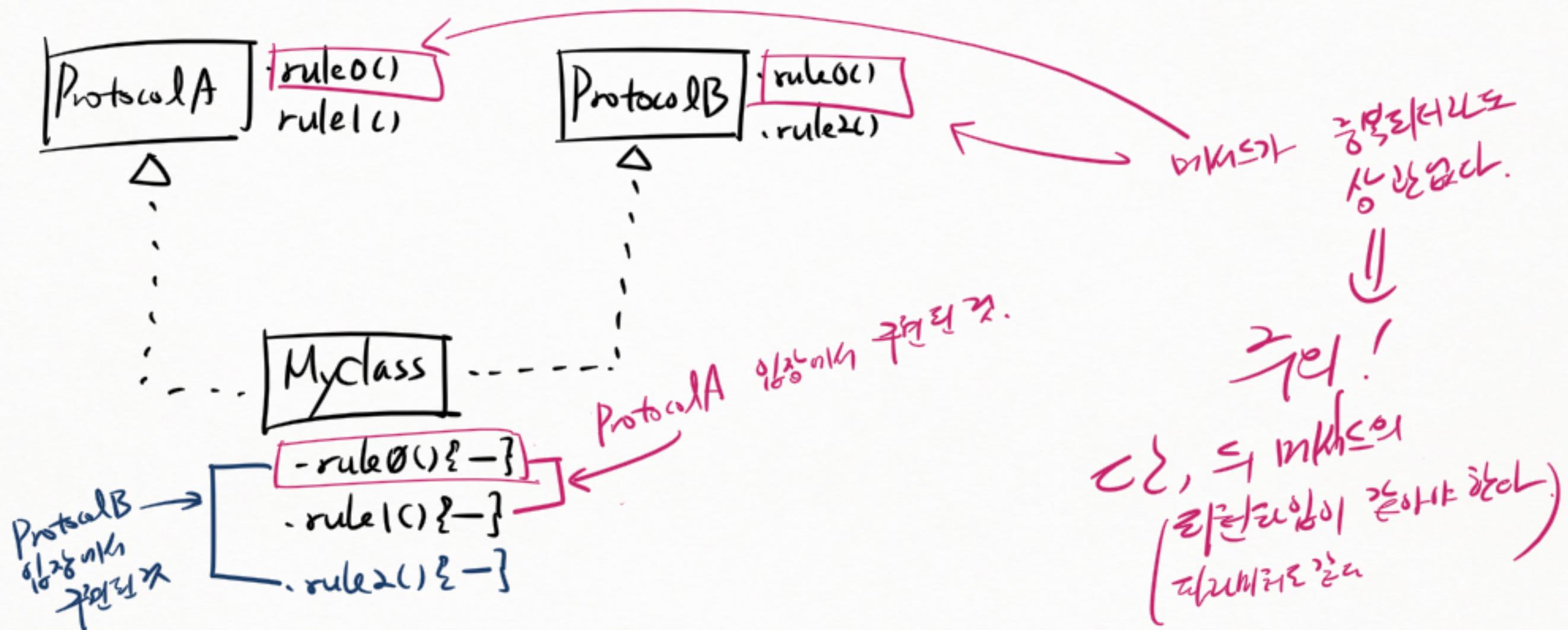
이전 경우는 상속 불가능하다.
상속 불가능한 경우에 대해서는
다음과 같이 처리된다.

1. 상속을 허용하지 않는다.
2. 상속을 허용하지만 예외로 한다.

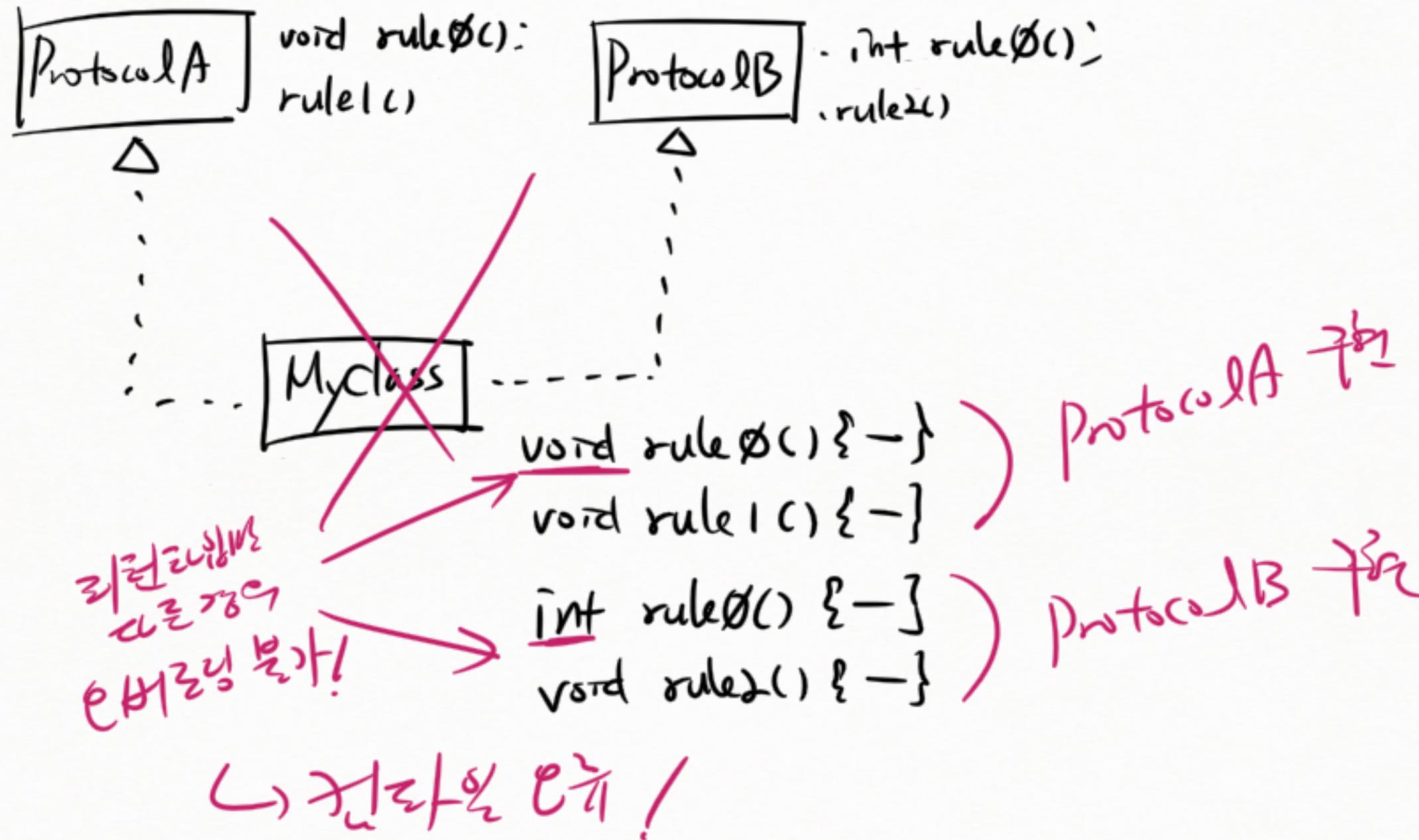
* 다중 인터페이스 구현



* 다중 인터페이스 구현 II

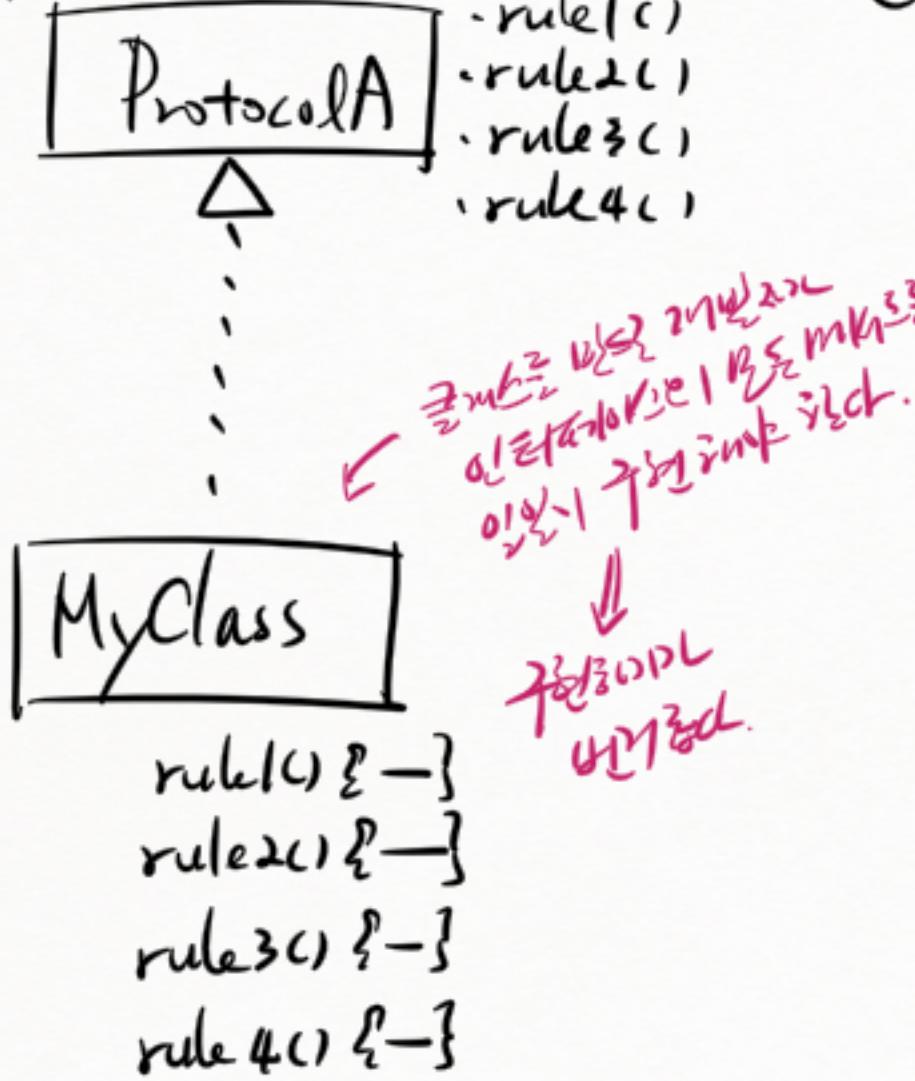


* 다중 인터페이스 구현 불가!

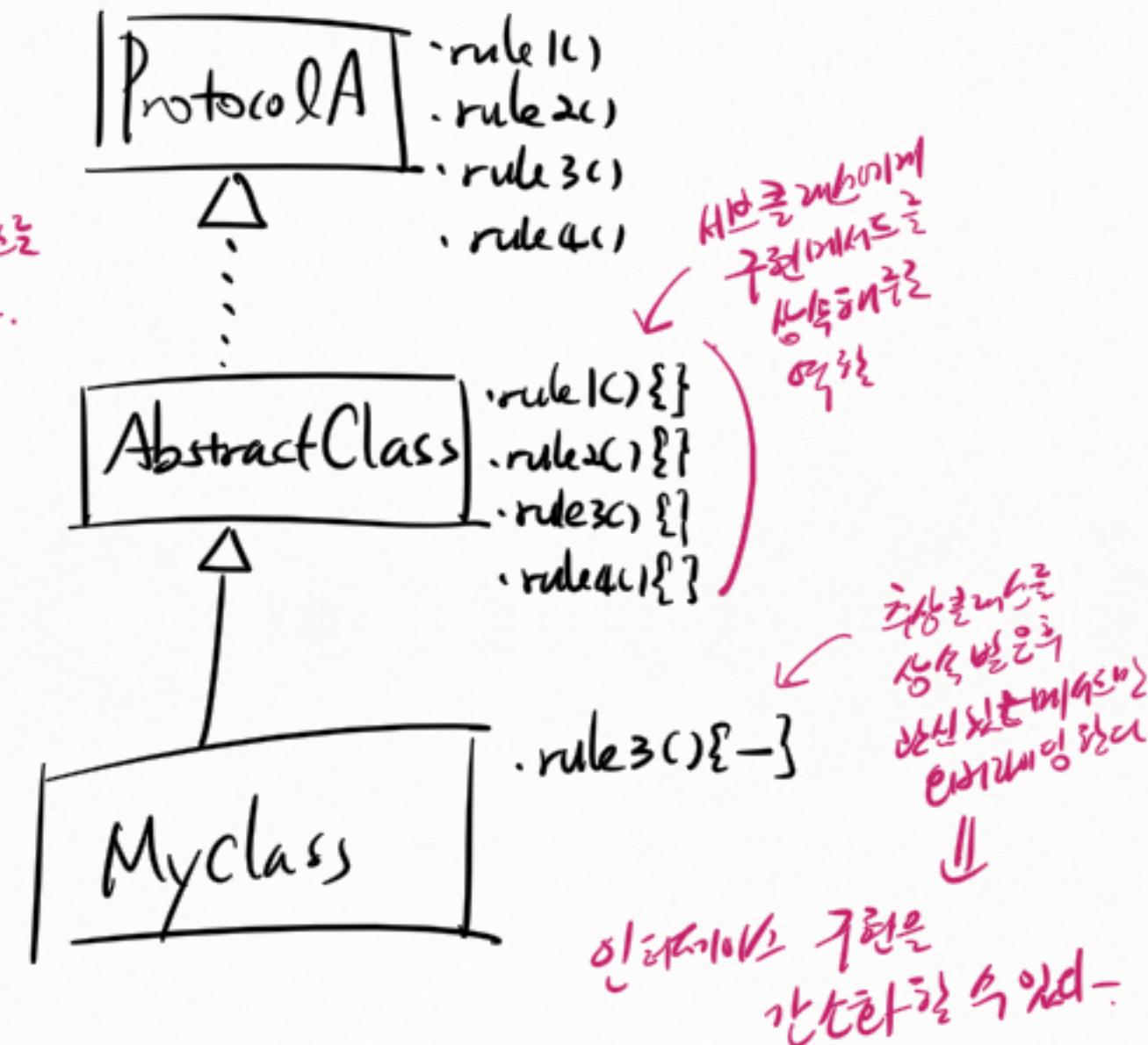


* 인터페이스 와 추상클래스

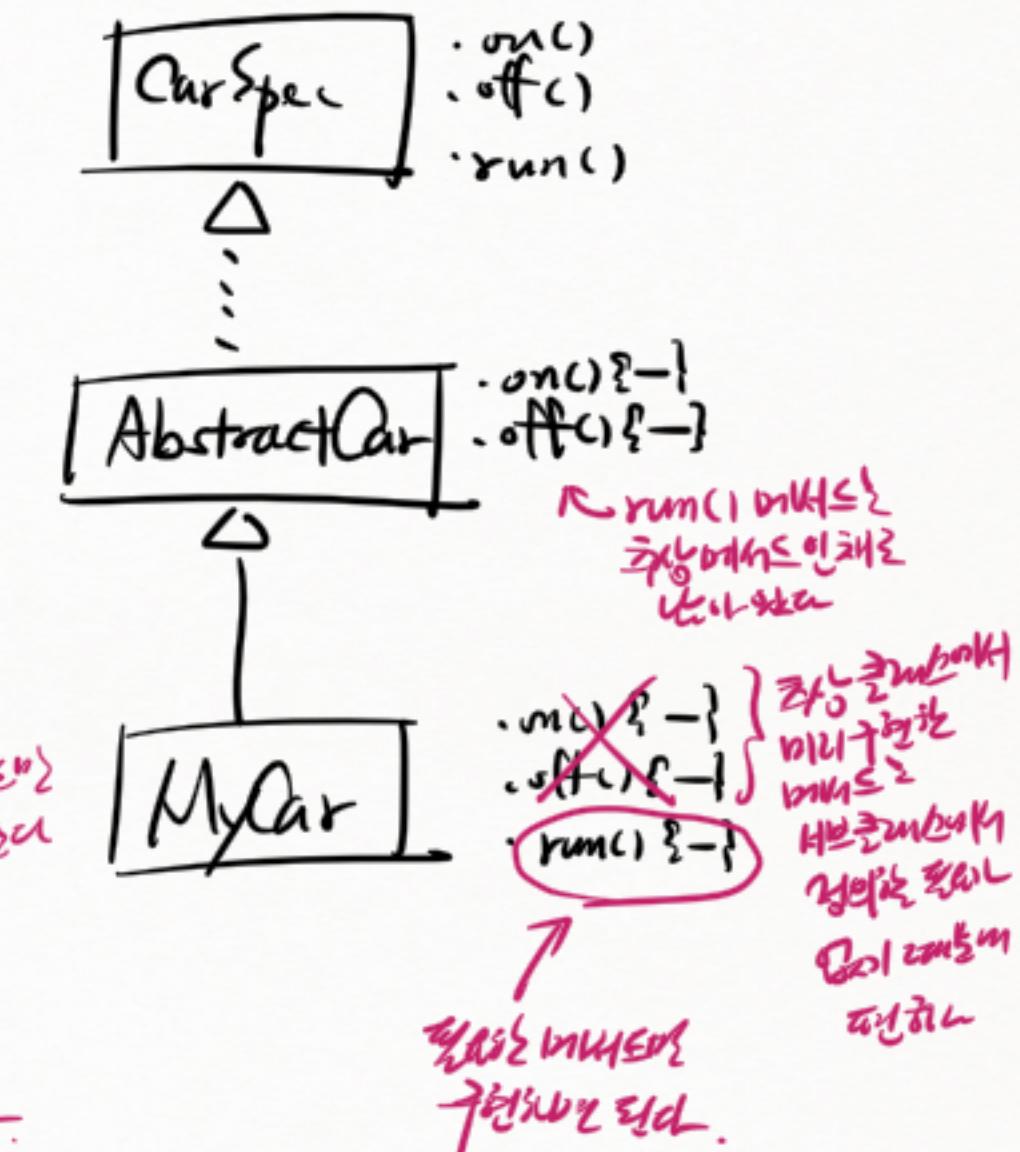
① 직립구조



② 추상클래스를 활용

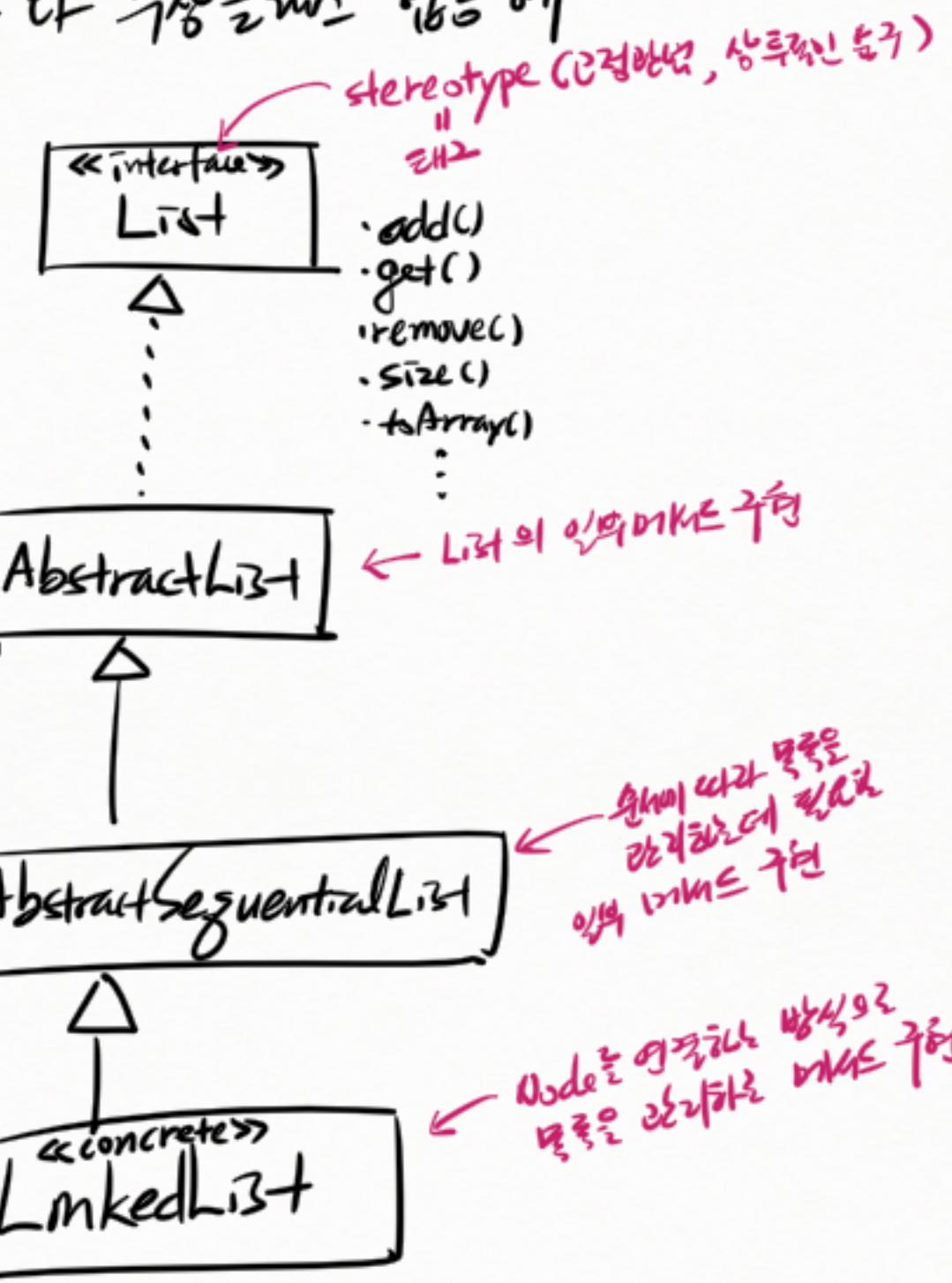


③ 실전 예



* 인터페이스와 추상클래스 협동 예

구현 예 : ⇒



구현 예

일부 구현 ⇒

구현 예

일부 구현 ⇒
+ 다음에 대해서 메서드 추가

다음과 같은 방식의
구현 예 : ⇒

* List or Set

~~100~~ ["apple" | 20]

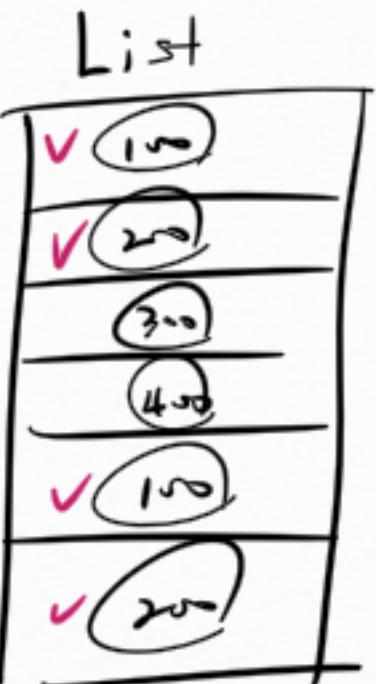
~~200~~ ["apple" | 30]

~~300~~ ["apple" | 10]

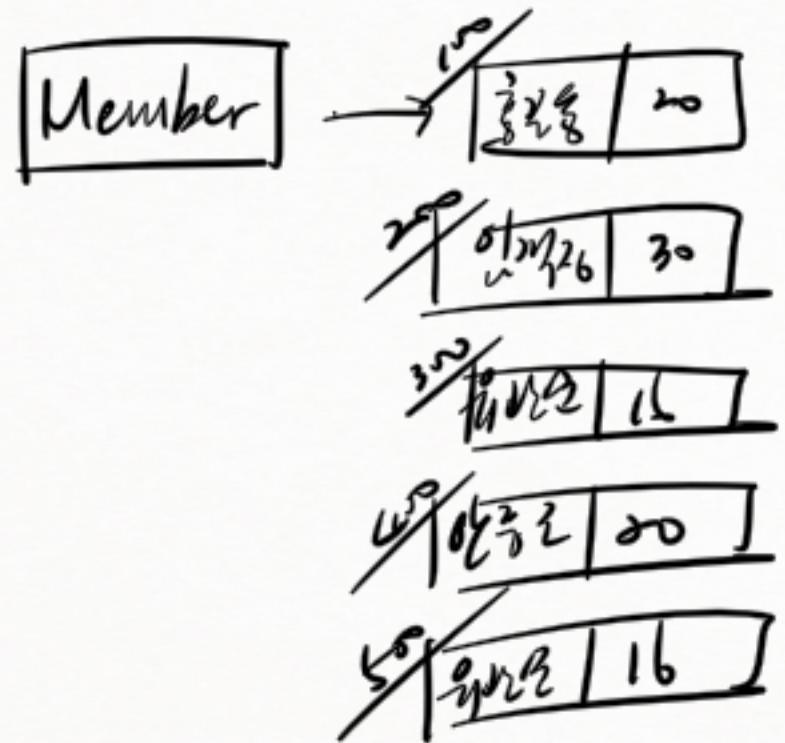
~~400~~ ["apple" | 30]

~~500~~ ["apple" | 40]

add(100)
add(200)
add(300)
add(400)
add(100)
add(200)



* HashSet zu equals(), hashCode()



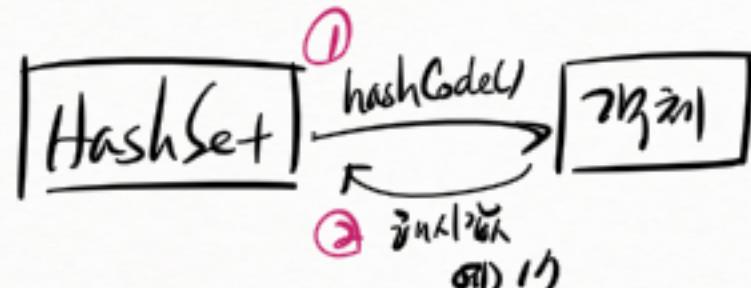
* HashSet vs HashMap

① hash함수는 객체의
② 해시값을 가지고
위치선정

17.05
11
2

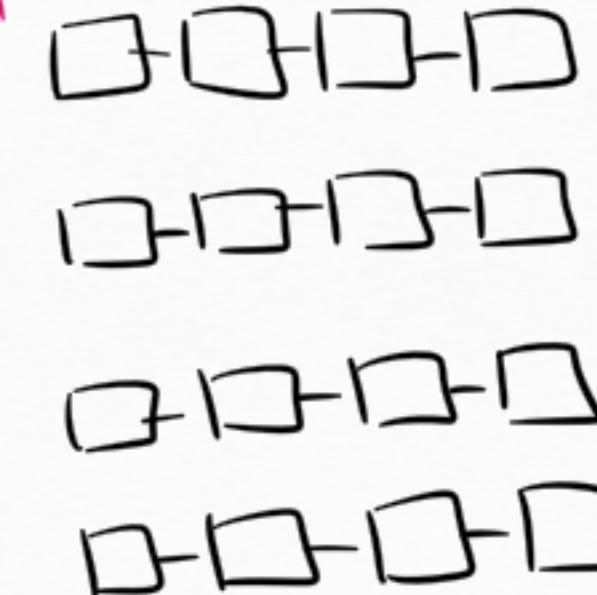
- 0 
- 1 
- 2 
- 3 
- 4 

이의의
순서로
꺼낸다
특정 객체를
꺼낼수가 있다.



③ hash함수는 객체의
해시값이 아닌,
그 객체의 key의
해시값을 가지고
위치선정

key를 가지는
특정 객체는
꼭 찾거나
꺼낼 수 있다.



key	value
"s01"	C
"s02"	-
"s03"	-
"s04"	-

