

class

* 팀스 운영의 패턴

- ① 멤버는 문구 : MemberHandler
Prompt
- ② 새 아이디 태입을 확장 : Score, Member

* 새 데이터 타입 정의

① class 정의

```
class Score {  
    String name;  
    int kor;  
    int eng;  
    int math;  
    int sum;  
    float aver;  
}
```

"인스턴스 변수" (field)

④ 인스턴스 초기화

obj. name = "홍길동"

② 인스턴스 할당

```
Score obj;
```

obj
200

obj = new Score();



③ 인스턴스 사용

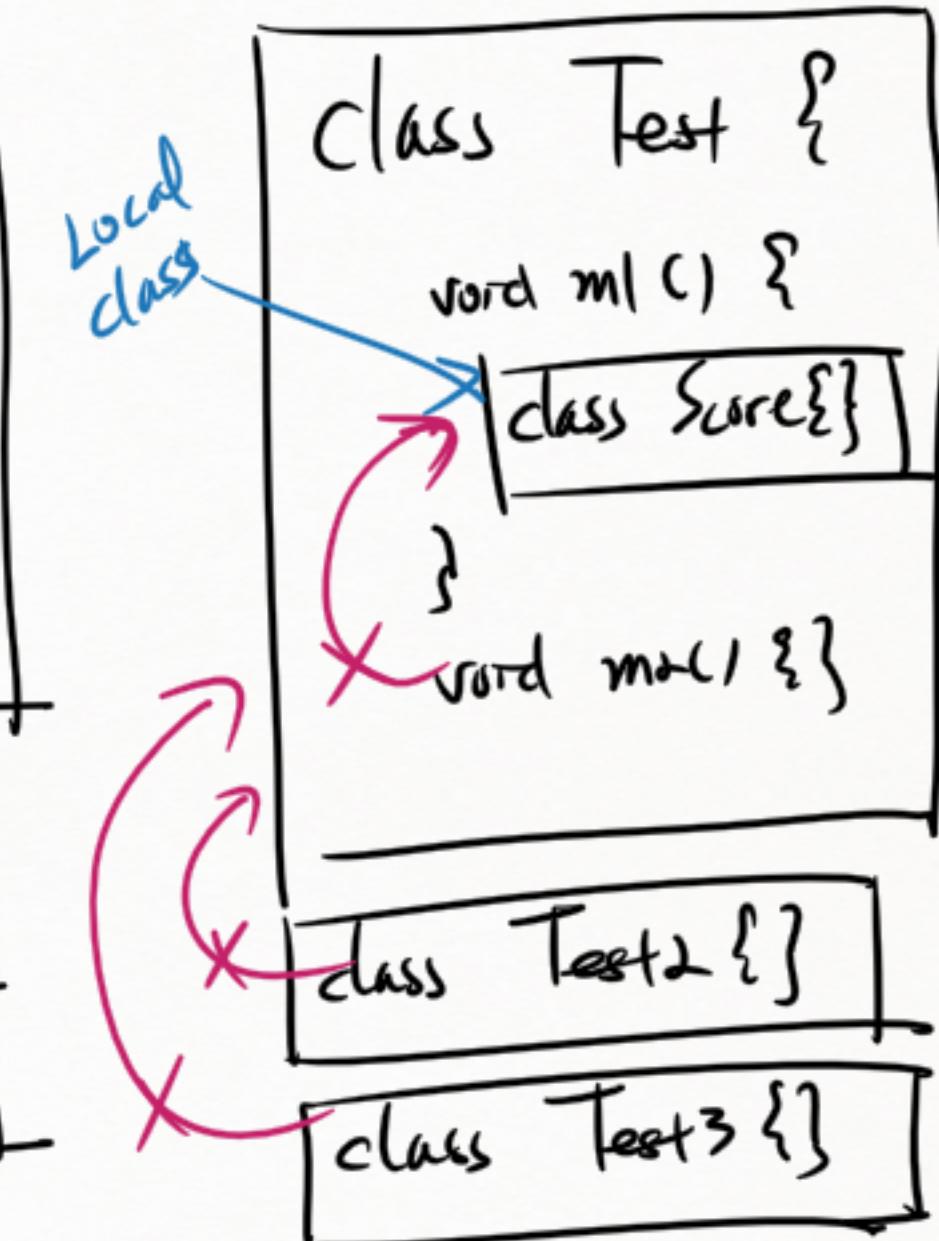
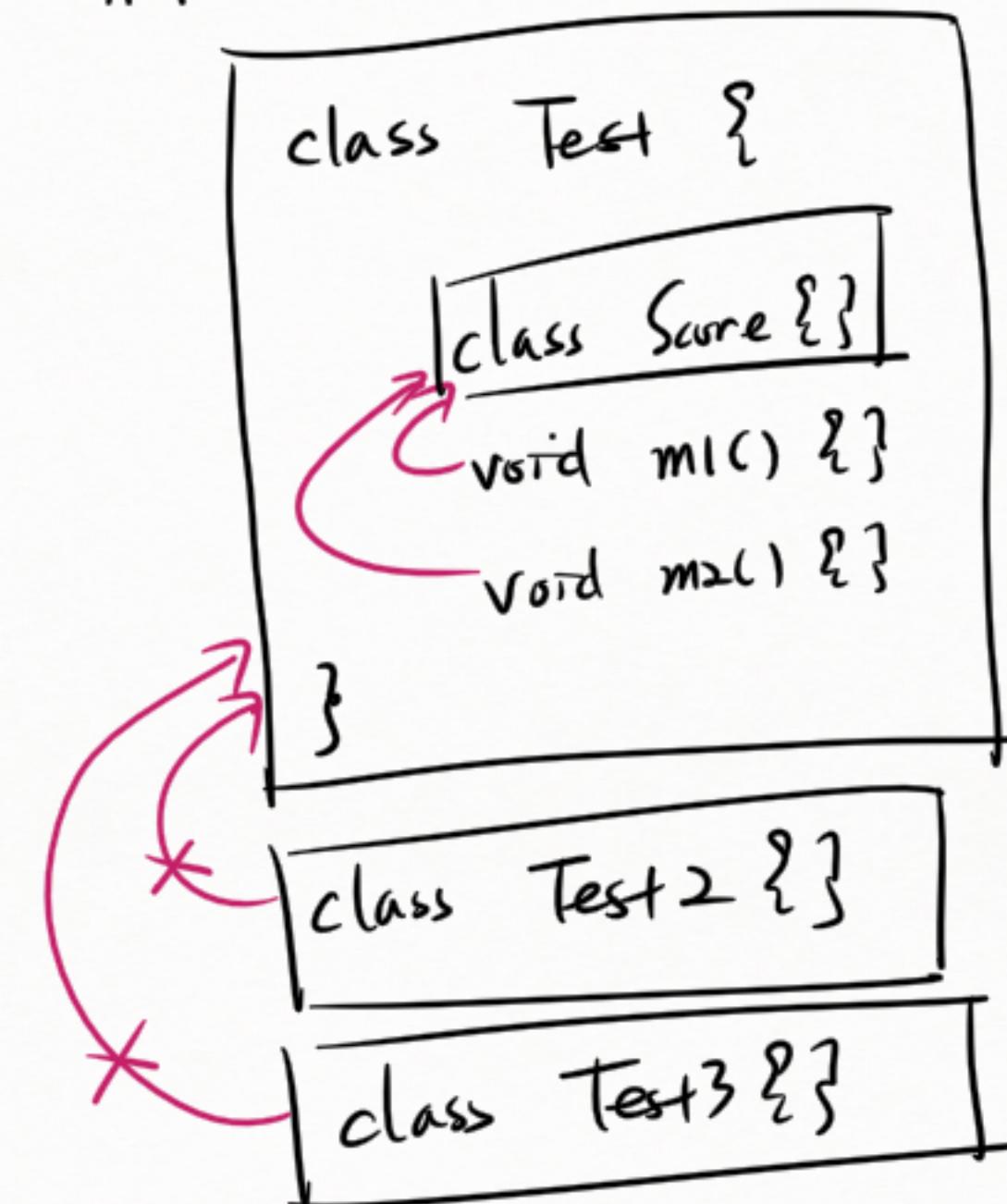
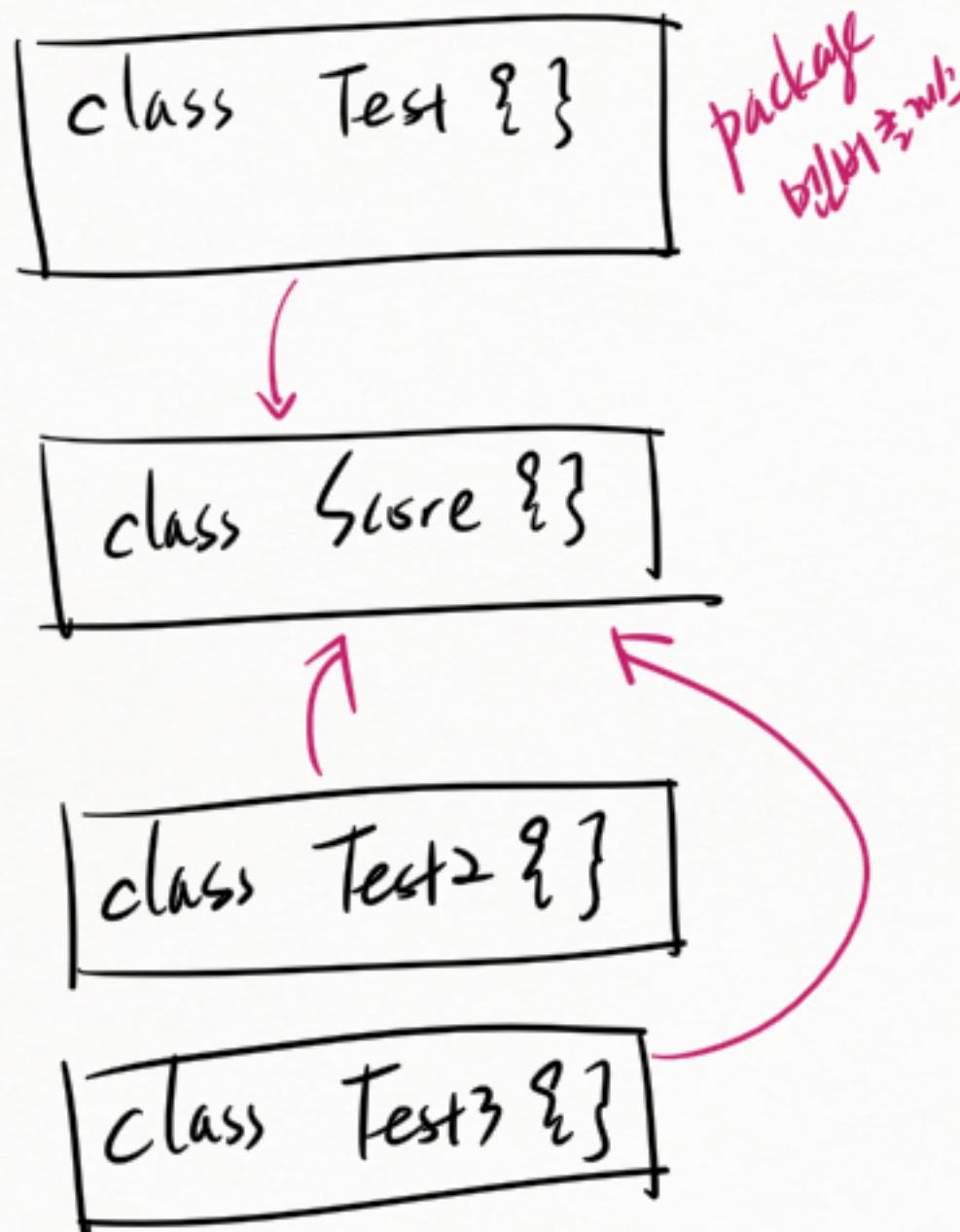
"Score의 인스턴스"
instance

"reference"

↑
Score의 인스턴스 주소를 저장하는 변수

Heap
obj → Head
of Exa.

* 클래스 정의의 유형



"Nested class"

* 인스턴스 생성, 메모리, call by reference

Score s = new Score();

s
200

200	name	kur	eng	math	sum	aver
	String	int	int	int	int	float
	한국어	100	90	80	270	90.0

s.name = "한국어";

s.kor = 100;

s.eng = 90;

s.math = 80;

s.sum = s.kor + s.eng + s.math;

s.aver = s.sum / 3f;

printScore(s);

인스턴스의 주소

printScore(Score s) {

}

System.out.printf();

}

* 디자인한 인스턴스 생성 후 출력

```

Score s = createScore("김민수", 100, 100, 100);    createScore (String name, int kor, int eng, int math) {
    s
    

|     |
|-----|
| 200 |
|-----|


    prntScore(s);
}

}
    Score s = new Score();
    s.name = name;
    s.kor = kor;           s.sum =
    s.eng = eng;           s.aver =
    s.math = math;
    return s;
}

s


|     |
|-----|
| 200 |
|-----|


    name kor eng math sum aver
    "김민수" 100 100 100 300 100.0

```

* 커스텀 클래스 사용 예

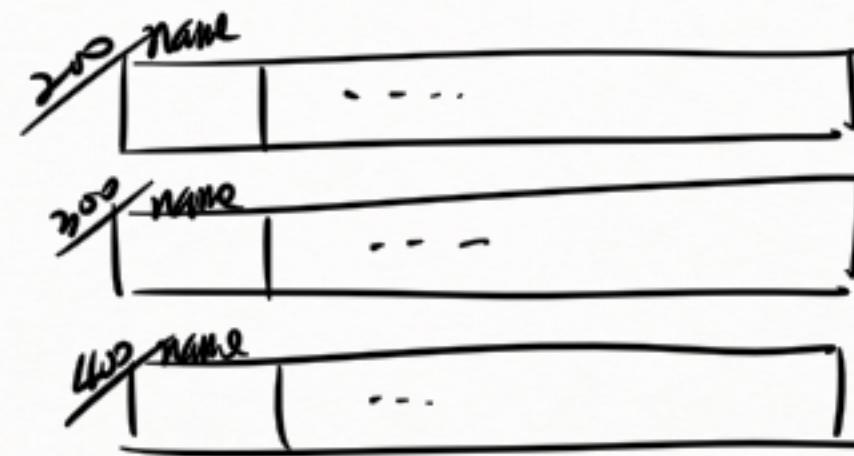
Score s1, s2, s3;

s1 | 200 s2 | 300 s3 | 400

s1 = new Score();

s2 = new Score();

s3 = new Score();



* 리퍼런스 변수 사용 후
 ↗️ 리퍼런스 변수
 ↗️ 리퍼런스들의 대체

`Score[] scores = new Score[3];`

`scores`

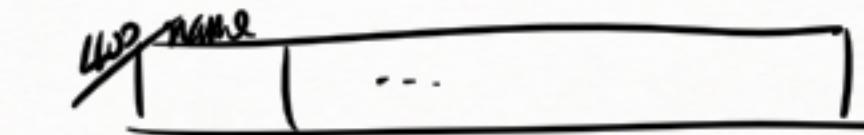
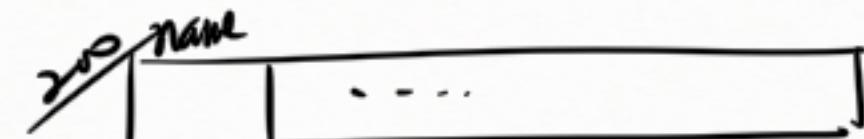
`1700`



`scores[0] = new Score();`

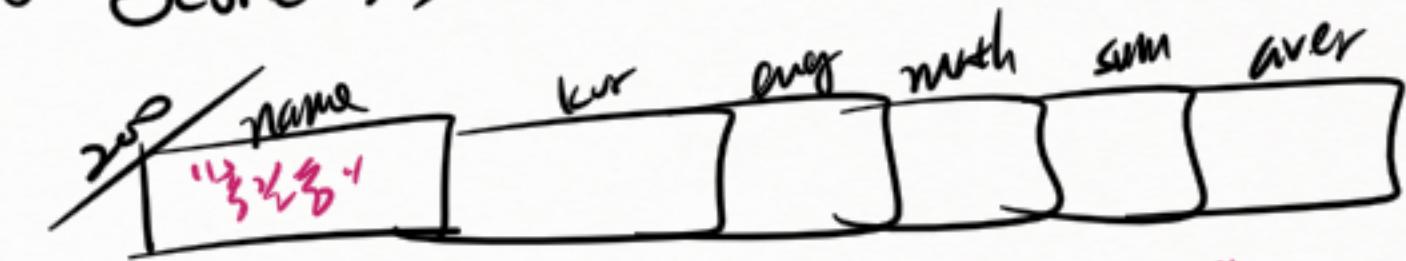
`scores[1] = new Score();`

`scores[2] = new Score();`



* 인스턴스와 메서드:

Score s1 = new Score();



"Score의 인스턴스"
개념

Score s2 = s1;

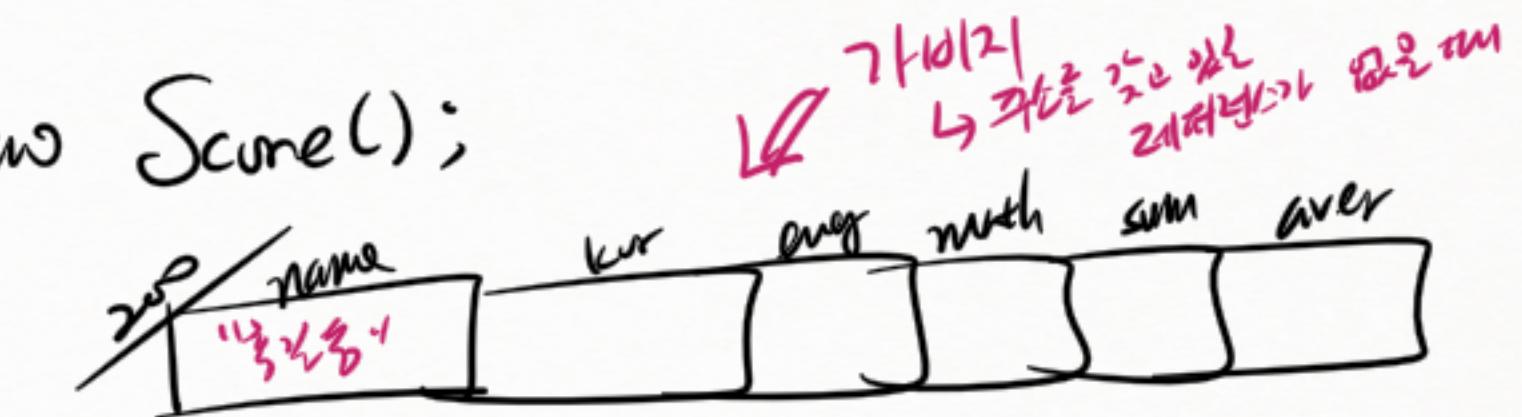


* 111011 (Garbage)

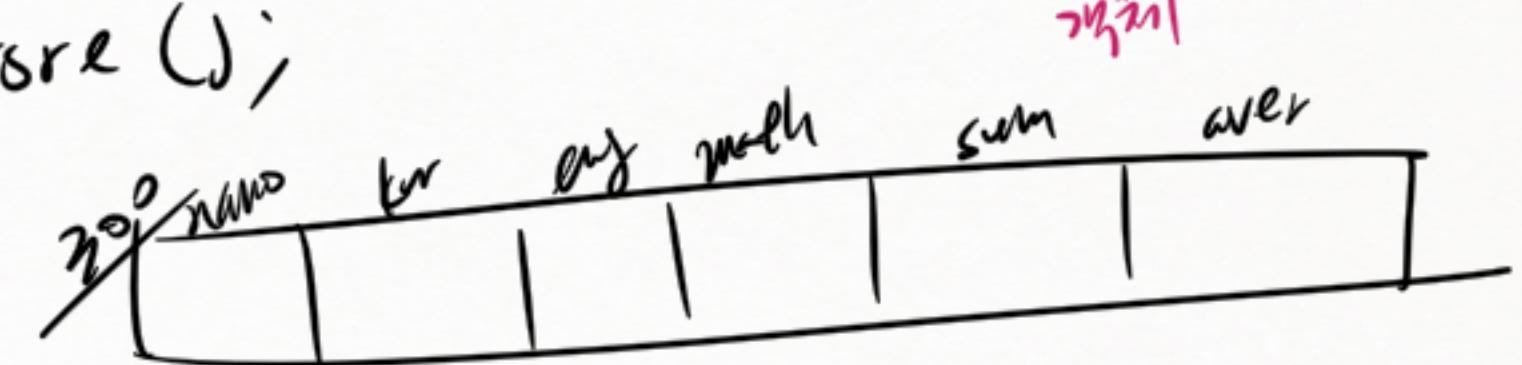
Score s1 = new Score();



s1 = new Score();



"Score의 인스턴스"

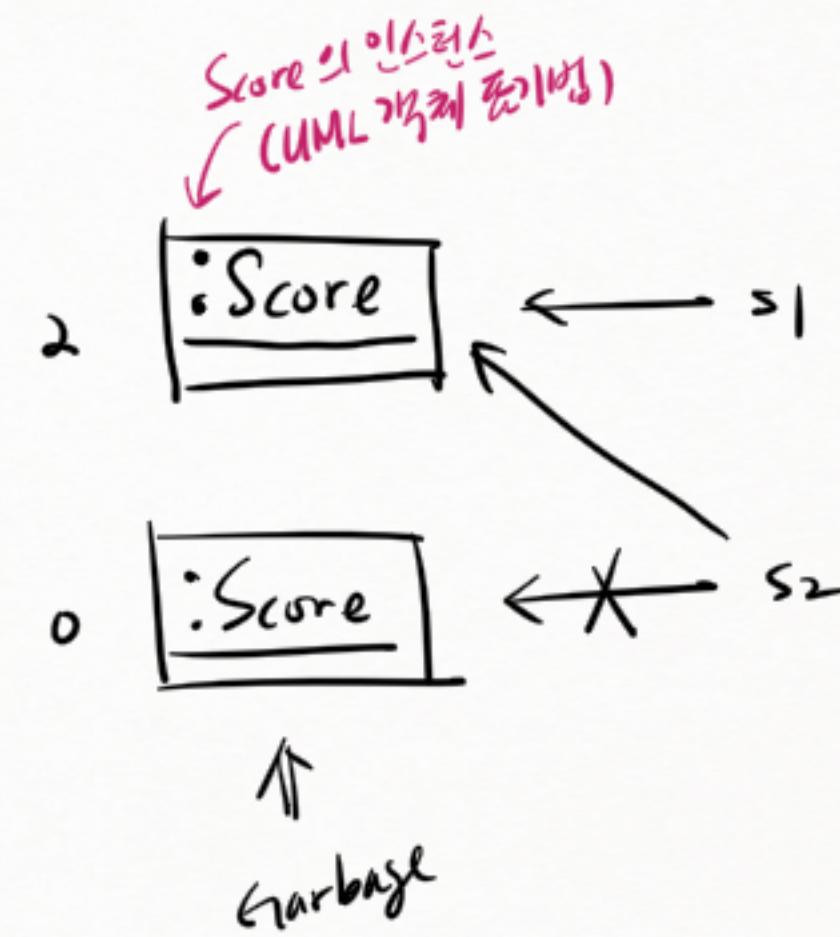


* 인스턴스와 리퍼런스 차운트

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

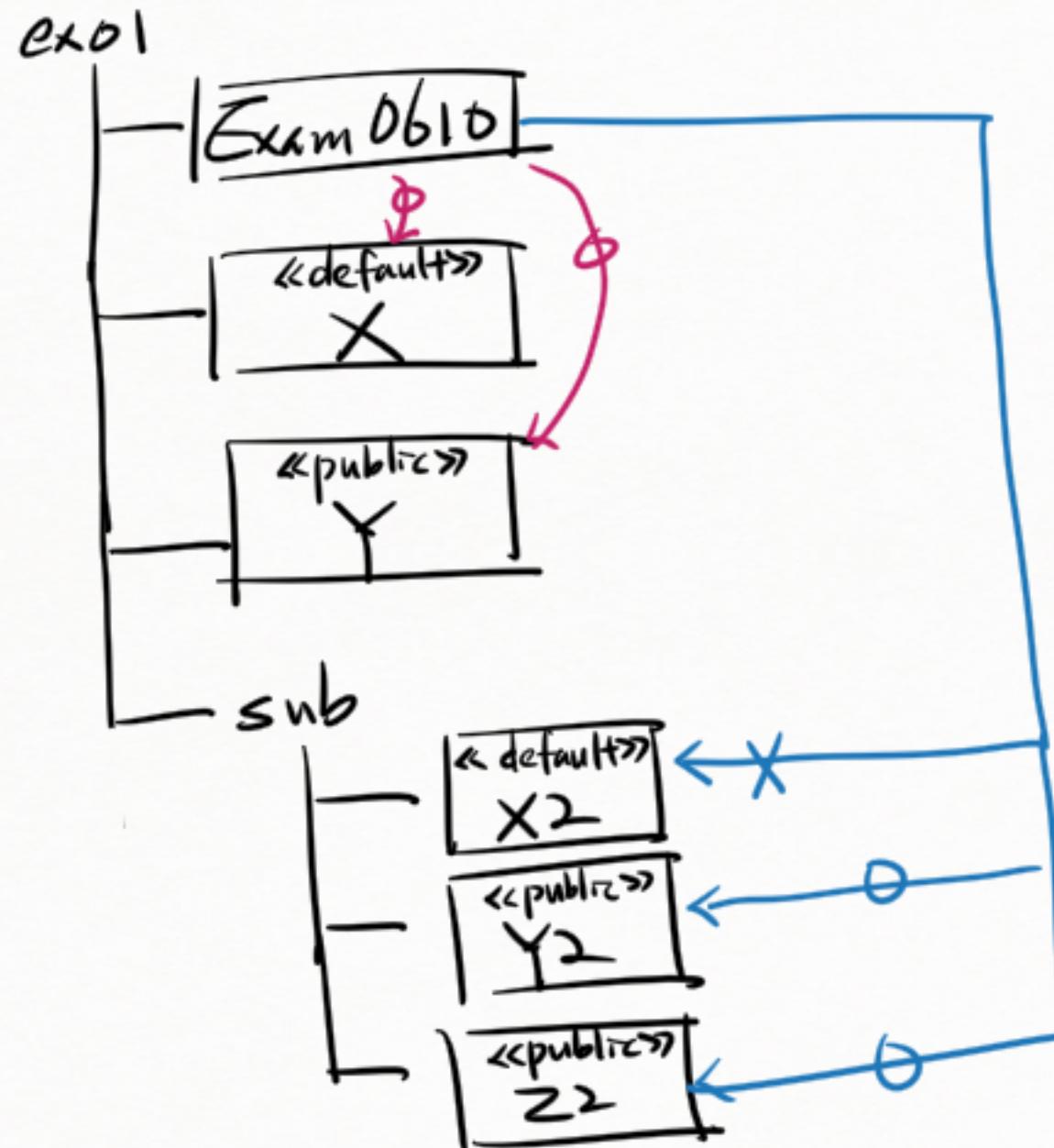
```
Score s2 = new Score();
```

```
s2 = s1;
```



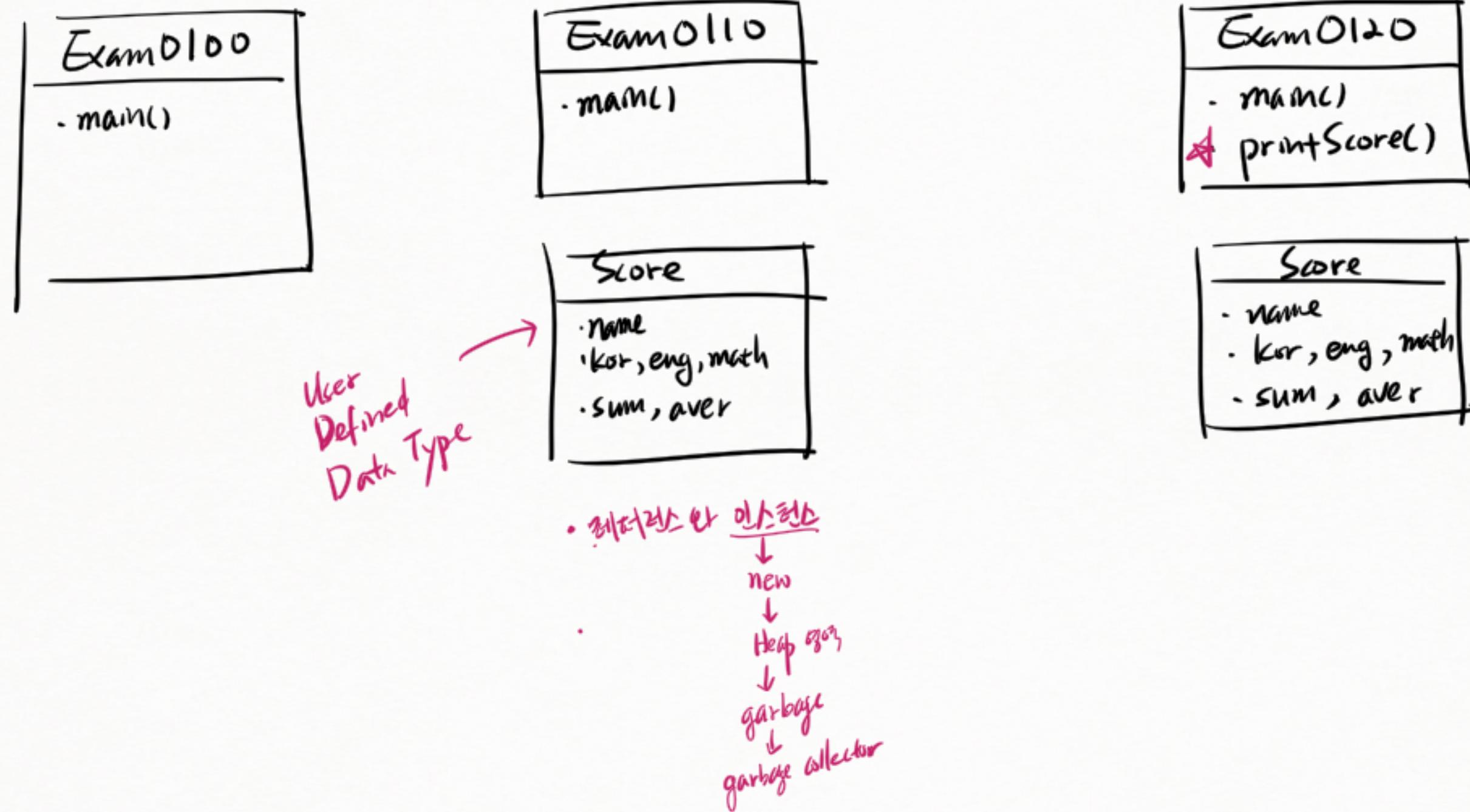
* public $\frac{3}{2}$ mLcf

default $\frac{3}{2}$ mL
(package private class)



* com. cs. oop. ex02. Exam01xx

① 놓개 변수 사용 → ② class, 블법: 새 데이터 타입 정의 → ③ method 블법: 중복코드 제거

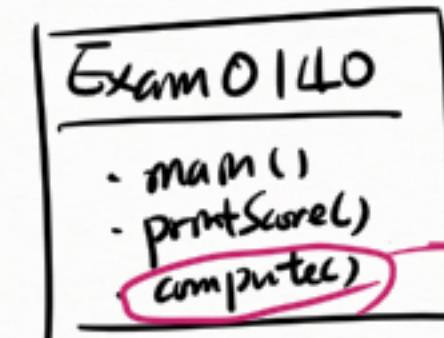
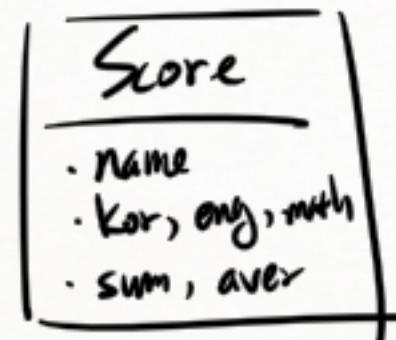
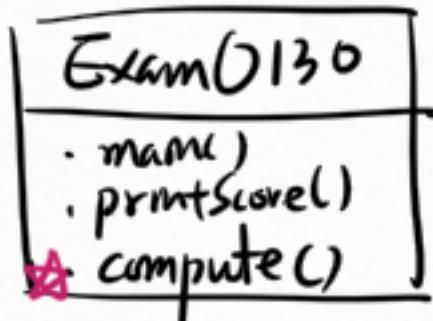


④ 리팩토링: 1기능 → 1 메서드

⑤ 리팩토링: 멤버드 이동

⑥ 인스턴스에 더 쉽게 접근하는 법: 인스턴스 메서드

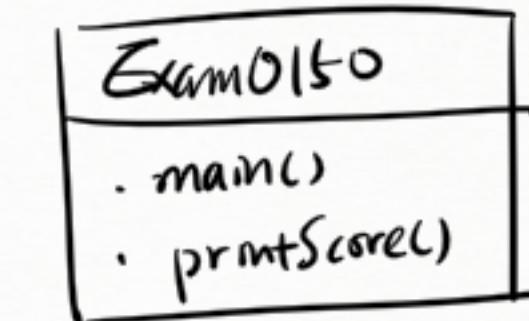
non-static



class
속성
↓
data를 정의한
그 데이터를 다루는
operator를 만든다.

GRASP의
Information
Expert

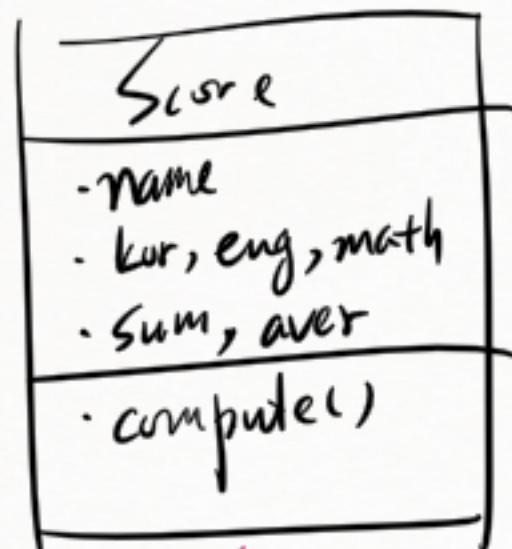
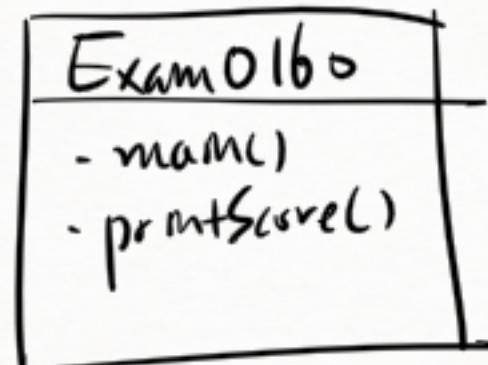
이동



non-static 으로
변경

이전 방식
Score. compute(인스턴스주소) → 변경 후
인스턴스주소. compute()

① 패키지 멤버 클래스



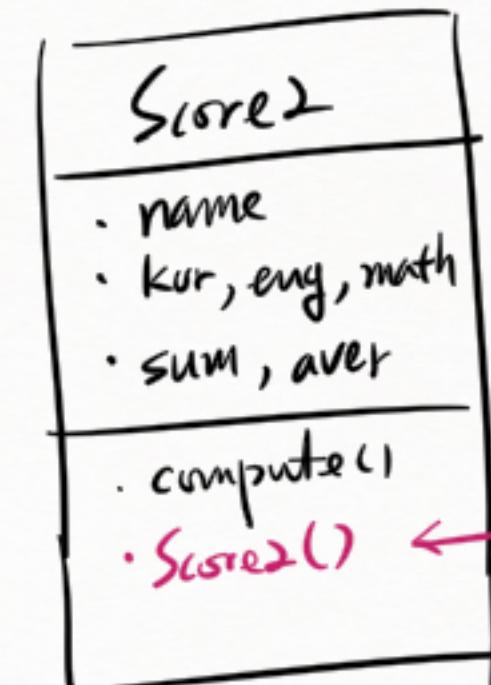
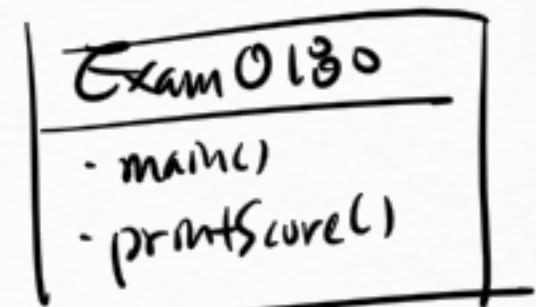
nested class → package member
변경

② 클래스를 패키지로 분리
관리 용이



- 접근제어 특성
- public : 외전용
 - protected : 내부클래스, 같은 패키지
 - (default) : 같은 패키지
 - private : 외부용

③ 객체 초기화 방법 : 생성자



* 스태틱 메서드와 인스턴스 메서드

static 메서드 \Rightarrow Score. compute(s1);
||
클래스 메서드

메서드가 소속된 클래스

non-static 메서드 \Rightarrow s1. compute();
||
인스턴스 메서드

메서드가 소속된 클래스의 인스턴스 주는
~~~ 인스턴스를 보다 쉽게 다루는  
메서드 문법

\* 인스턴스 메서드와 this

인스턴스  
메서드를 호출할 때  
this는  
매개변수로  
传여  
된다.  
이쪽에서 넘겨온 인스턴스 주소를 받는  
built-in 로컬 변수

non-static 멤버에만 존재한다.

\* this 가는  
this 멤버변수에 저장된다

↓  
인스턴스 주소를 블리지  
하고 멤버를 접근할  
수가 있다.

```
void compute() {  
    this.sum = this.kor + this.eng + this.math;  
    :  
}
```

\* 생성자

class Score {

↘ 클래스 이름과 같은 변수 있다

Score (int num, ...)

}

=

}

기원래 있어  
왔다

```
graph TD; Object(( )) -- "기원래 있어 왔다" --> Score1[Score]; Score1 --- ScoreDef["Score { int num, ... }"]; ScoreDef --- Score2[Score]; Score2 --- Equals["="]; Score2 --- BottomBrace["{"];
```

## \* 생성자呼び出し

① 이름

```
Score s = new Score();
```

```
s.name = "홍길동";
```

```
s.kor = 100;
```

```
s.eng = 100;
```

```
s.math = 100;
```

```
s.compute();
```

} Score 객체 생성  
이스턴스 생성

생성자 호출

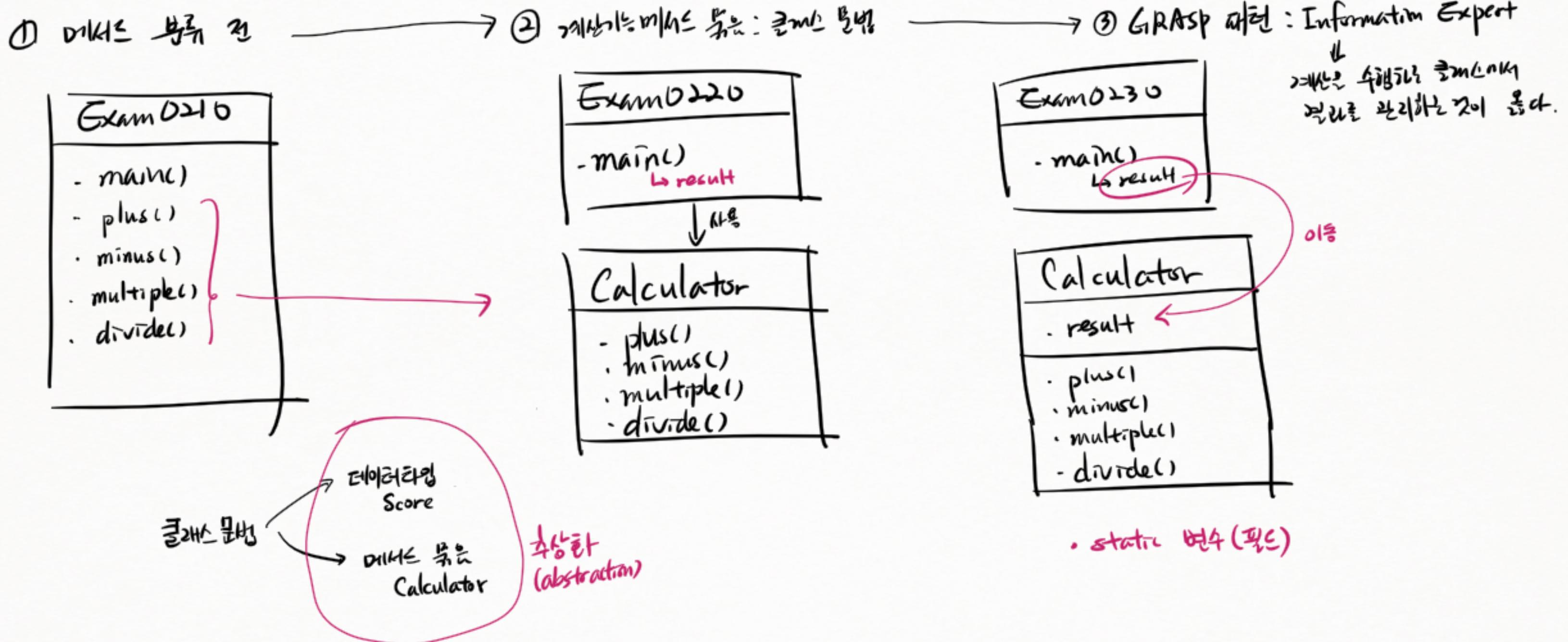
② 내용

```
Score s = new Score("홍길동", 100, 100, 100);
```

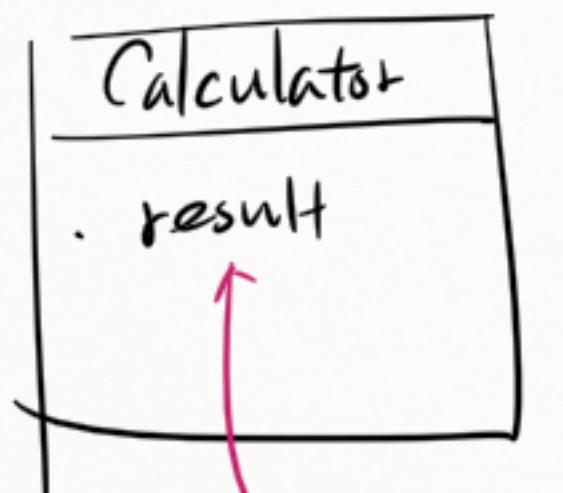
↓ 이스턴스 생성과 즉시 생성자 자동호출

```
Score (String n, int k, int e, int m){  
    this.name = n;  
    this.kor = k;  
    this.eng = e;  
    this.math = m;  
    this.compute();  
}
```

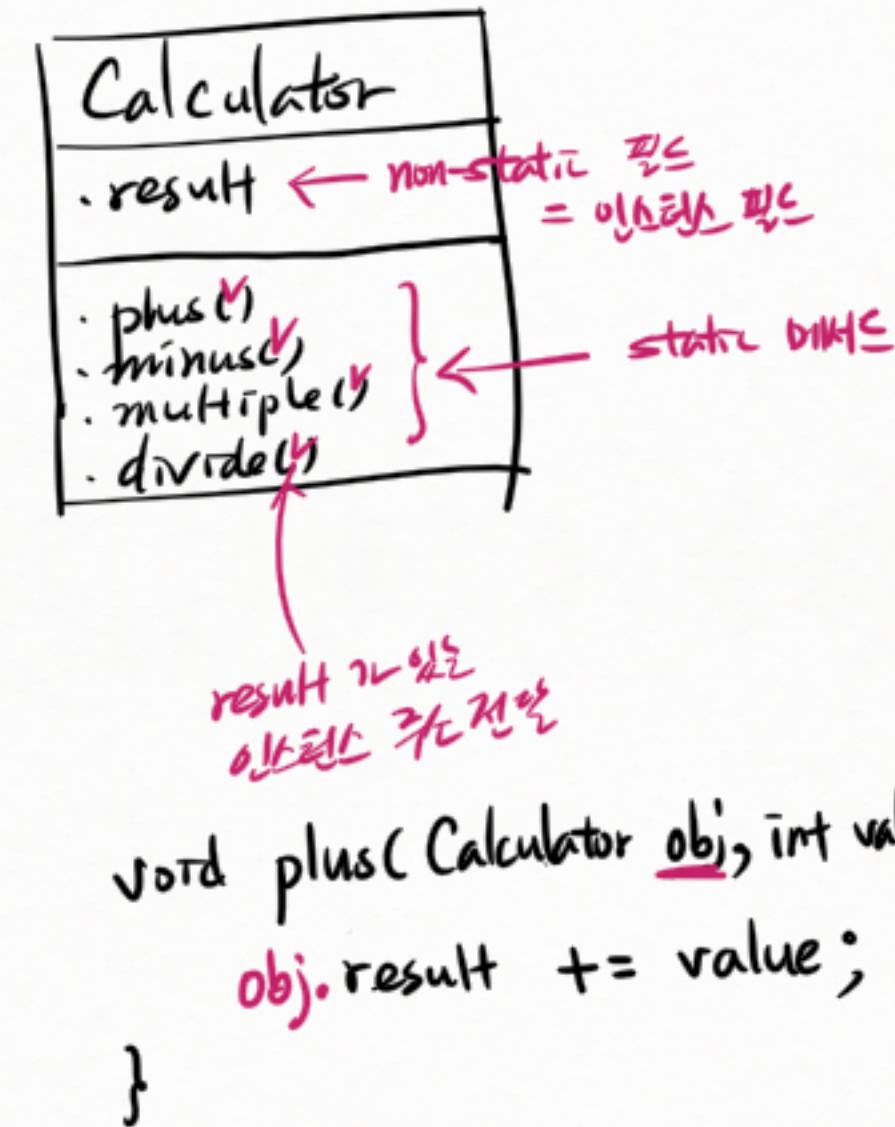
\* 스태틱 필드 → 인스턴스 필드  
 (com.eomcs.oop.p. ex02.Exam02xx)



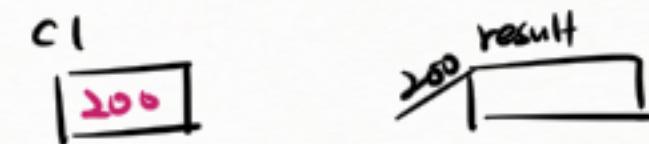
④  $\Rightarrow$  멤버 변수의 초기화  $\longrightarrow$  ⑤ 인스턴스 변수로 전환



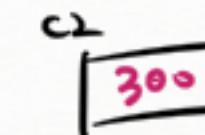
static 멤버  
멤버는 static 멤버  
인스턴스가 아니므로  
모든 인스턴스가 공유하는  
값이 됨.



Calculator c1 = new Calculator();



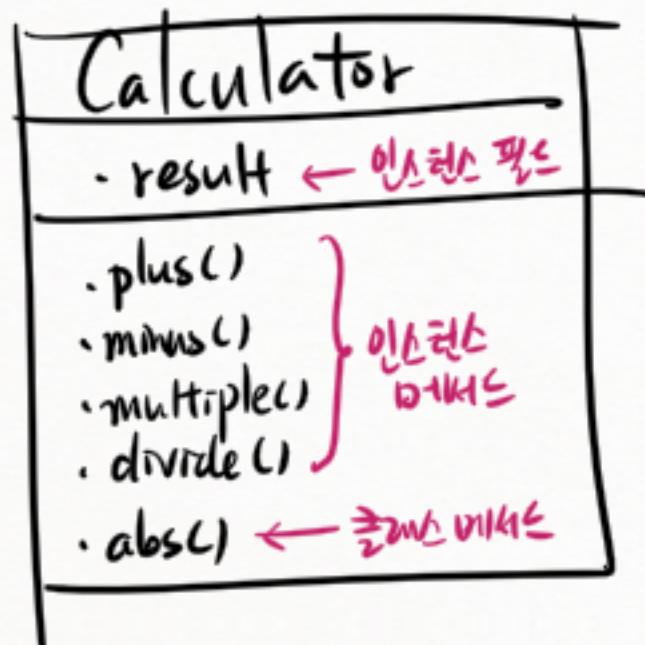
Calculator c2 = new Calculator();



Calculator.plus(c1, 2);

Calculator.plus(c2, 3);

⑥ static 멤버드 → 인스턴스 멤버드 정의



void plus( int value ) {  
 this.result += value;  
}

↑ 인스턴스 멤버드의 Built-in 변수

Calculator c1 = new Calculator();



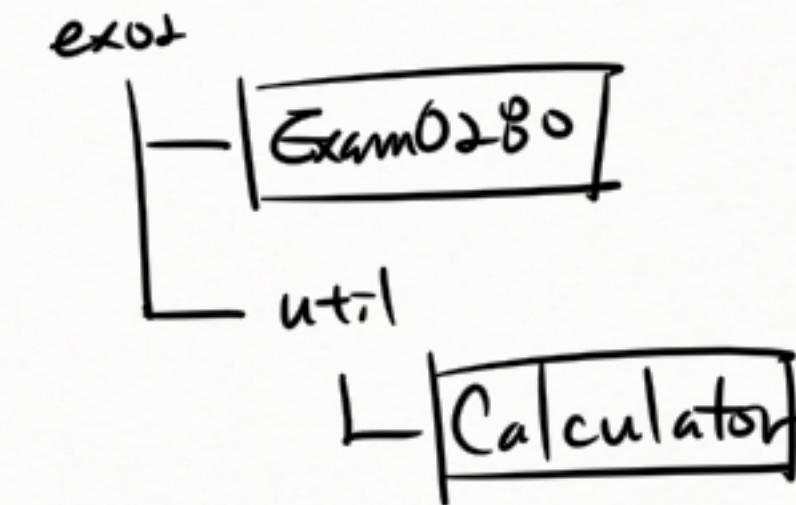
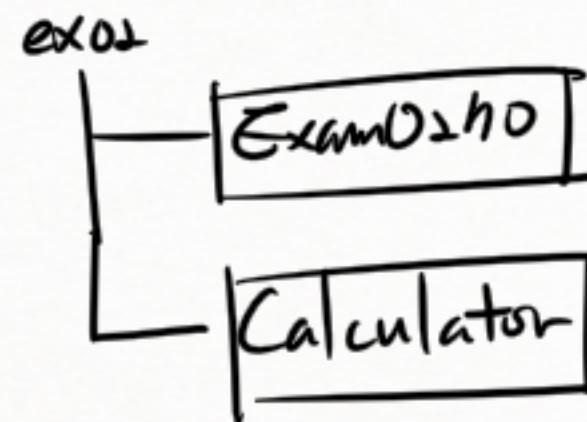
Calculator c2 = new Calculator();



c1.plus( 2 );

c2.plus( 3 );

① Nested  $\frac{2}{2}m\backslash$  → Package member  $\frac{2}{2}m\backslash$   $\rightarrow$  ②  $\text{util}$   $\frac{2}{2}$

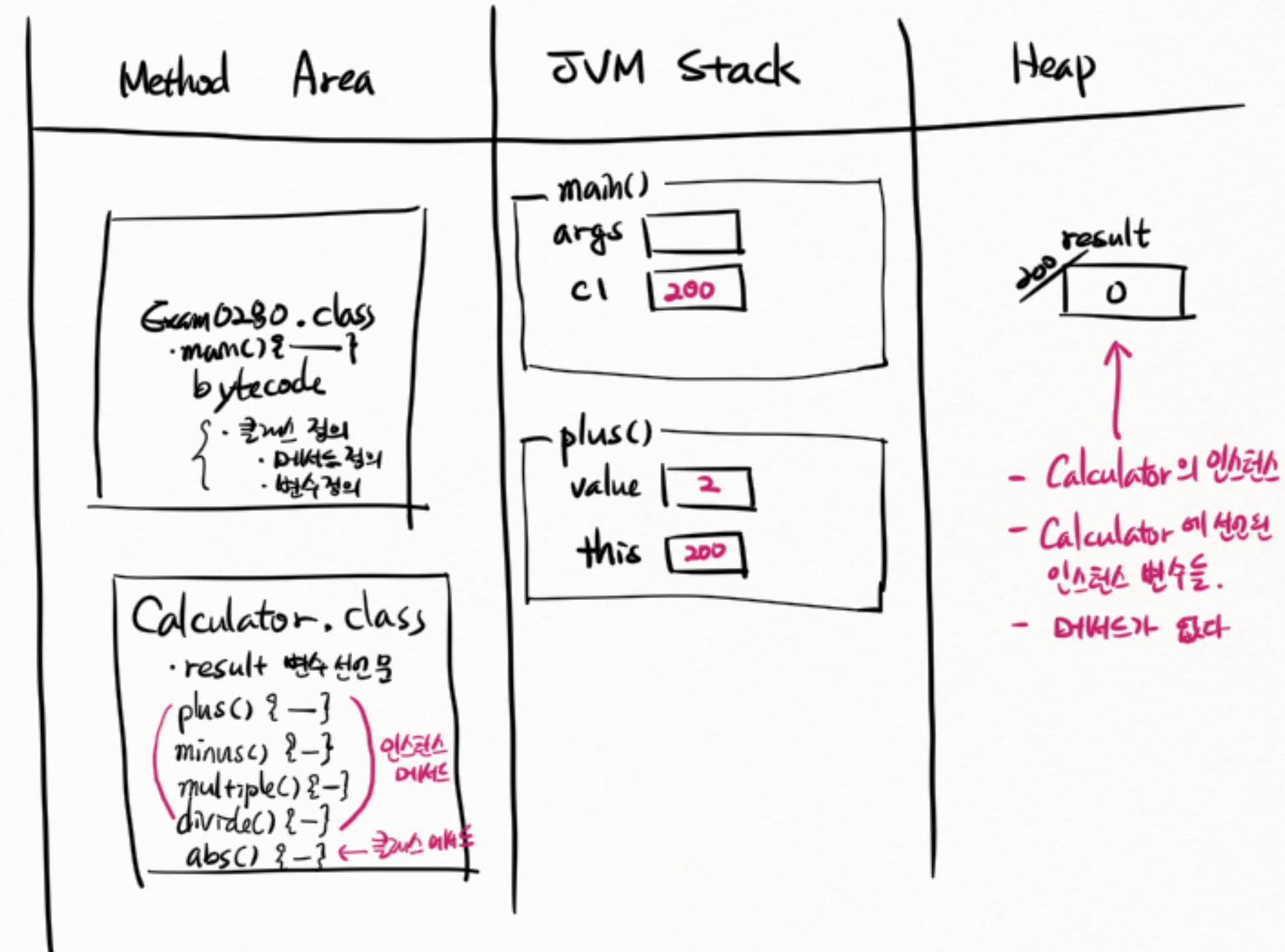


## \* JVM 디버깅 예제와 변수, 메서드

```
class Exam0280 {
    void main() {
        Calculator c1 = new Calculator();
        c1.plus(2);
    }
}
```

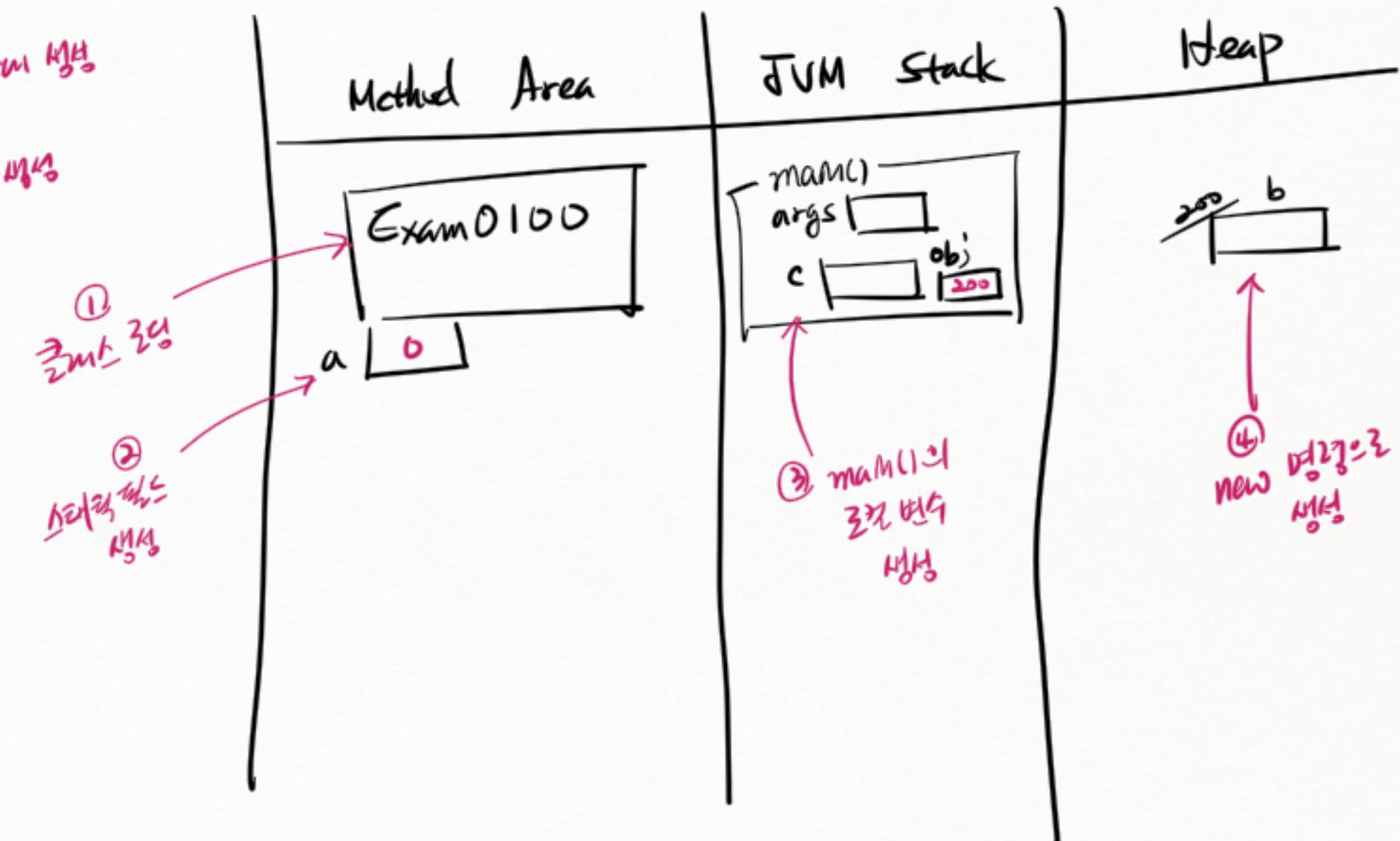
*인스턴스 주소*

*Calculator에 선언된 인스턴스 변수를 Heap에 생성한다*

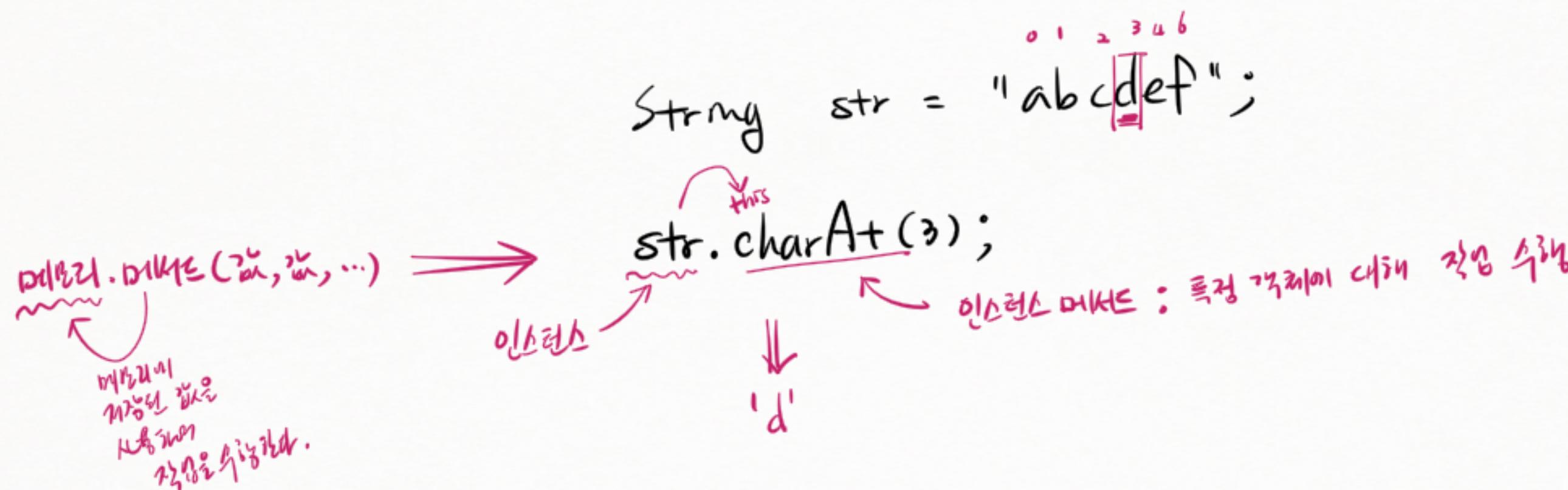
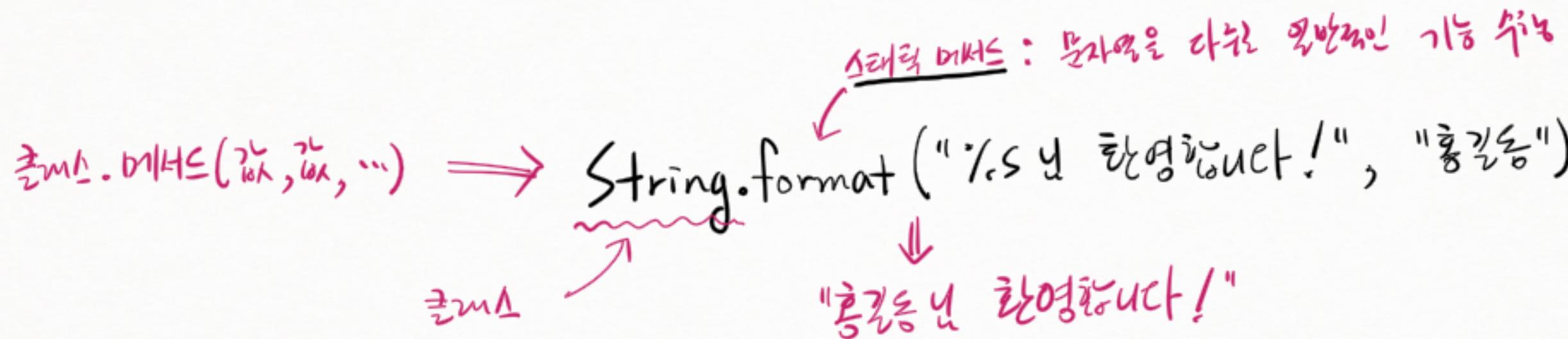


\* 스레티 필드, 인스턴스 필드, 로그 변수

```
class Exam0100 {  
    static int a; // 스레티 필드  
    int b; // 인스턴스 필드  
    void main() {  
        int c; // 로그 변수  
        Exam0100 obj;  
        obj = new Exam0100();  
    }  
}
```



## \* 스태틱 메서드 vs 인스턴스 메서드



\* static 필드, instance 필드, local ~~변수~~ 변수

```
class Exam000 {  
    static int a;  
    int b;  
    mam() {  
        int c;  
    }  
}
```

① static 필드

- 클래스가 로딩될 때 생성된다 (Method Area)

② instance 필드

- new 연산자로 인스턴스 생성할 때 만든다 (Heap)

③ local 변수

- 메소드 호출될 때 만든다 (JVM stack)

\* 생성자

class Score {

x Score() {

}

Score(string name) {

=

}

Score(string name, int kur, int eng, int math) {

=

}

:

new Score();

↑  
생성자 호출

new Score(); *default constructor*

new Score("Bob");

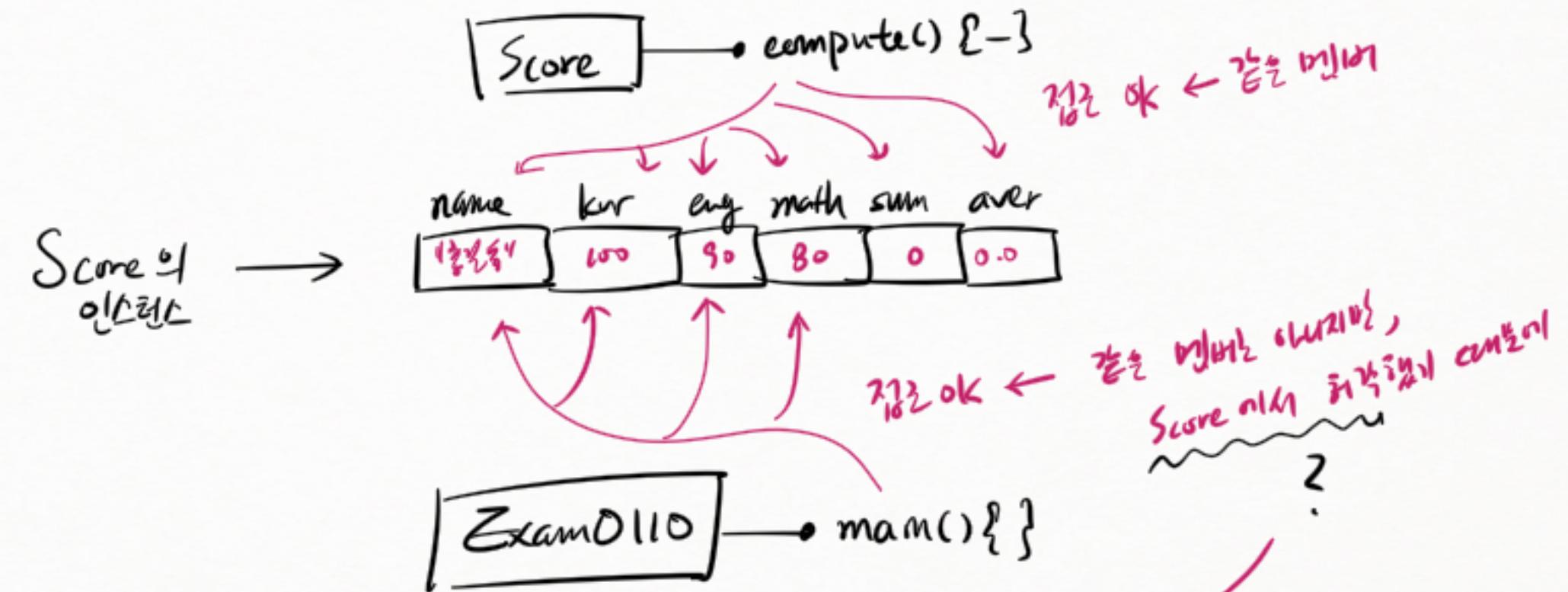
new Score("Bob", 100, 90, 85);

new Score("Bob", 100); *Illegal argument*

Encapsulation + getler / settler

\* 멤버에 대한 접근 제어 : (default)

```
class Score {  
    String name;  
    int kor;  
    int eng;  
    int math;  
    int sum;  
    float aver;  
    computer() {}  
}
```



\* 멤버 접근 제어

- private: 같은 패키지

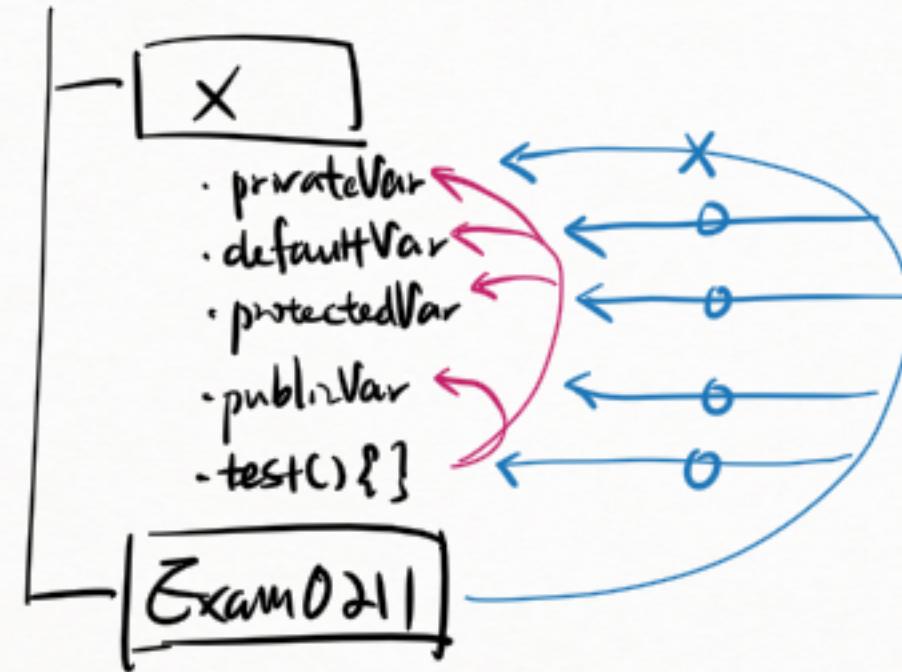
- (default): " + 같은 패키지"

- protected: " + " + 서브클래스의 멤버

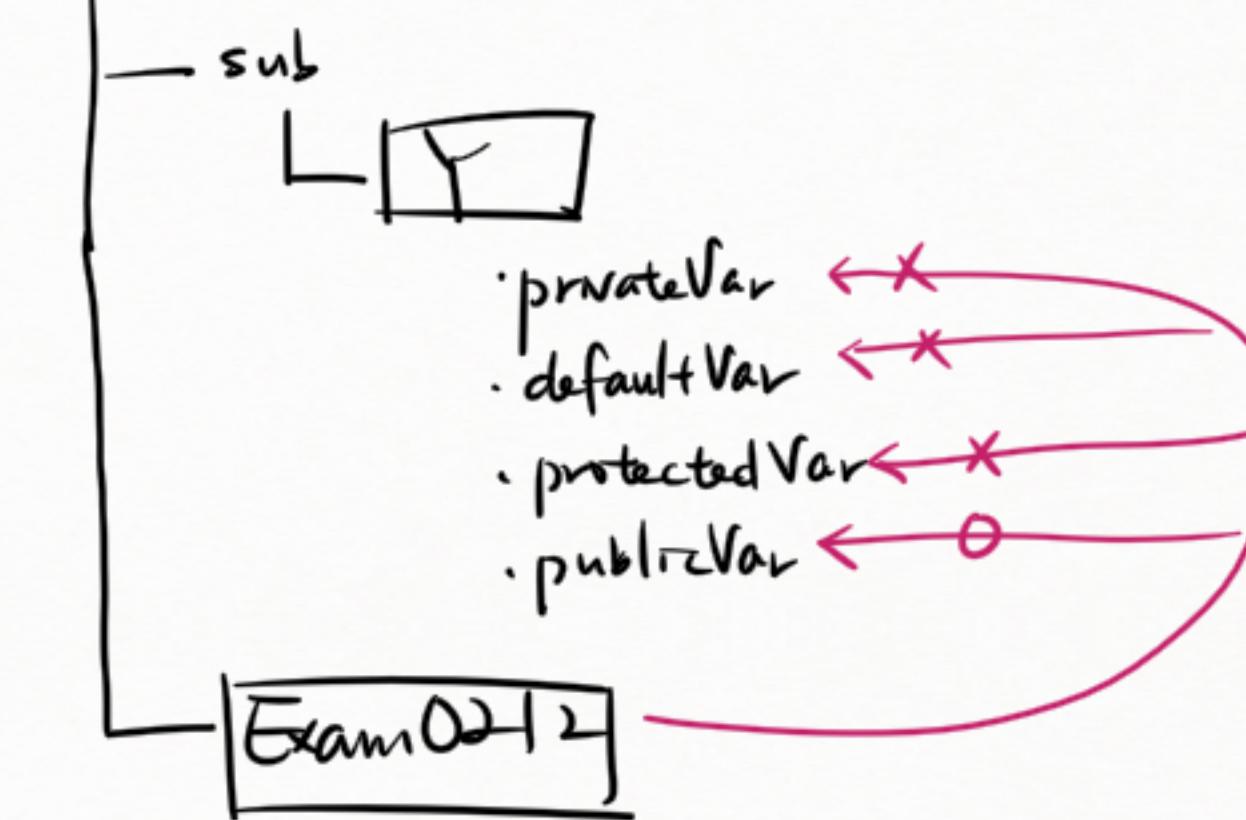
- public: 모두

## \* 2장 멤버 접근 제한

ex08

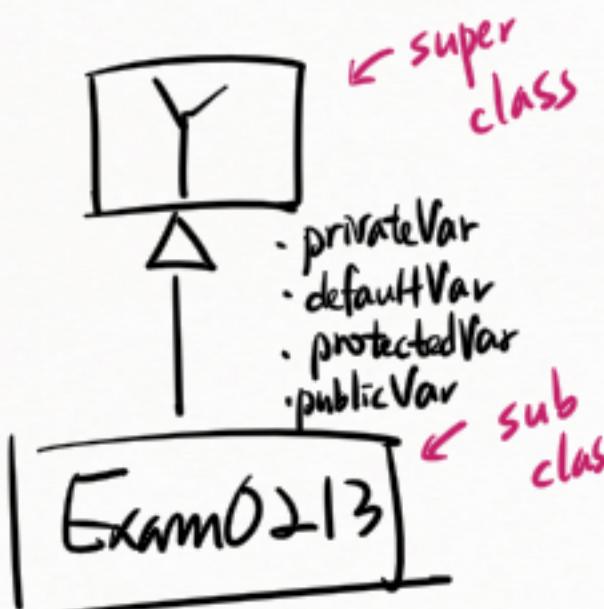
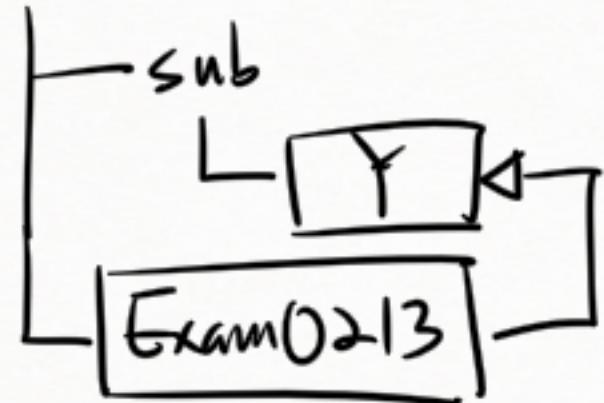


ex08



\* 접근 방지 코드 : protected 접근

exo<sup>+</sup>



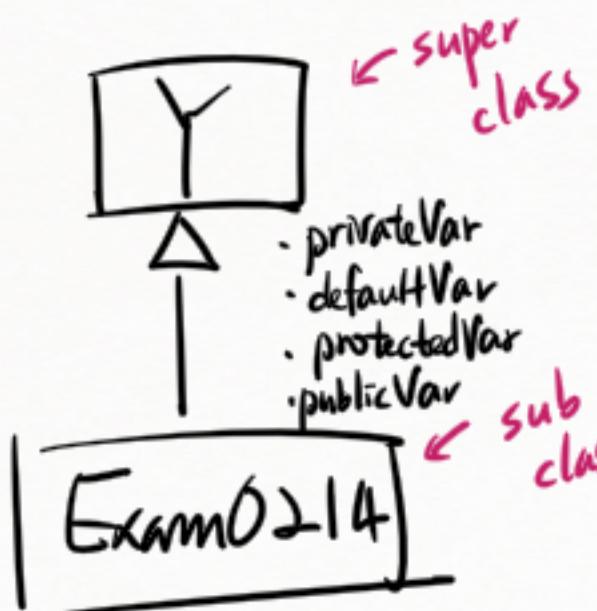
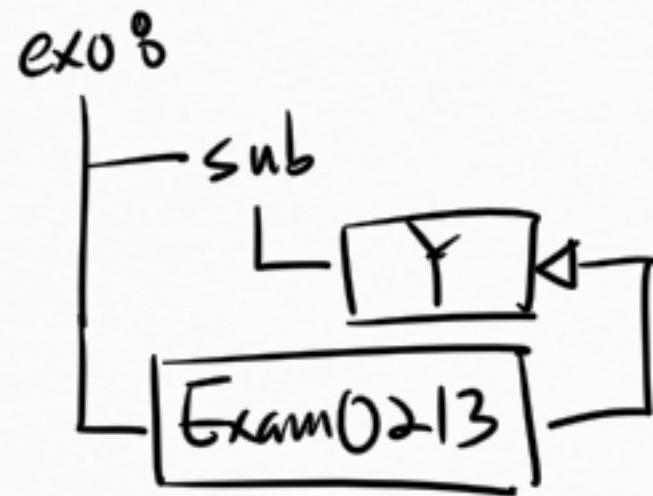
new Y()

1

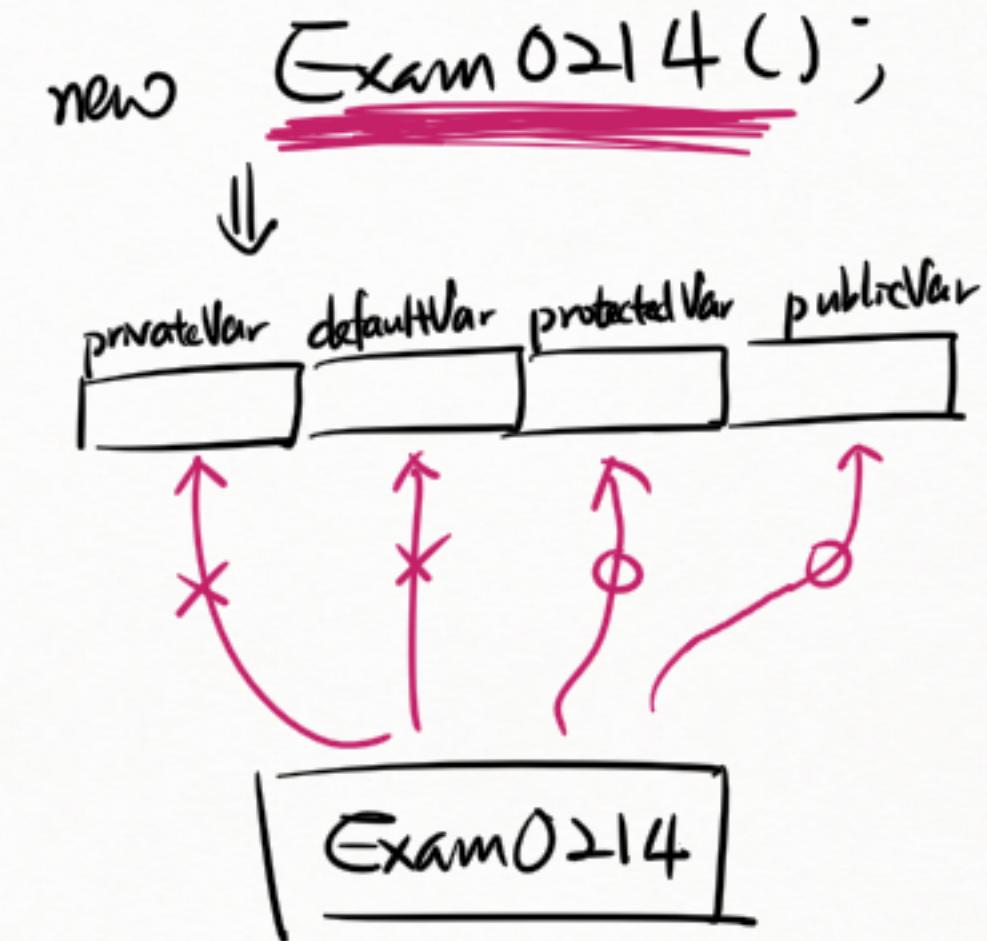
privateVar defaultVar protectedVar publicVar

서보클러스터: 서보드에 대처  
변수인 경우 가능  
접근 가능

\* 접근 범위 제한 : protected 범위 II



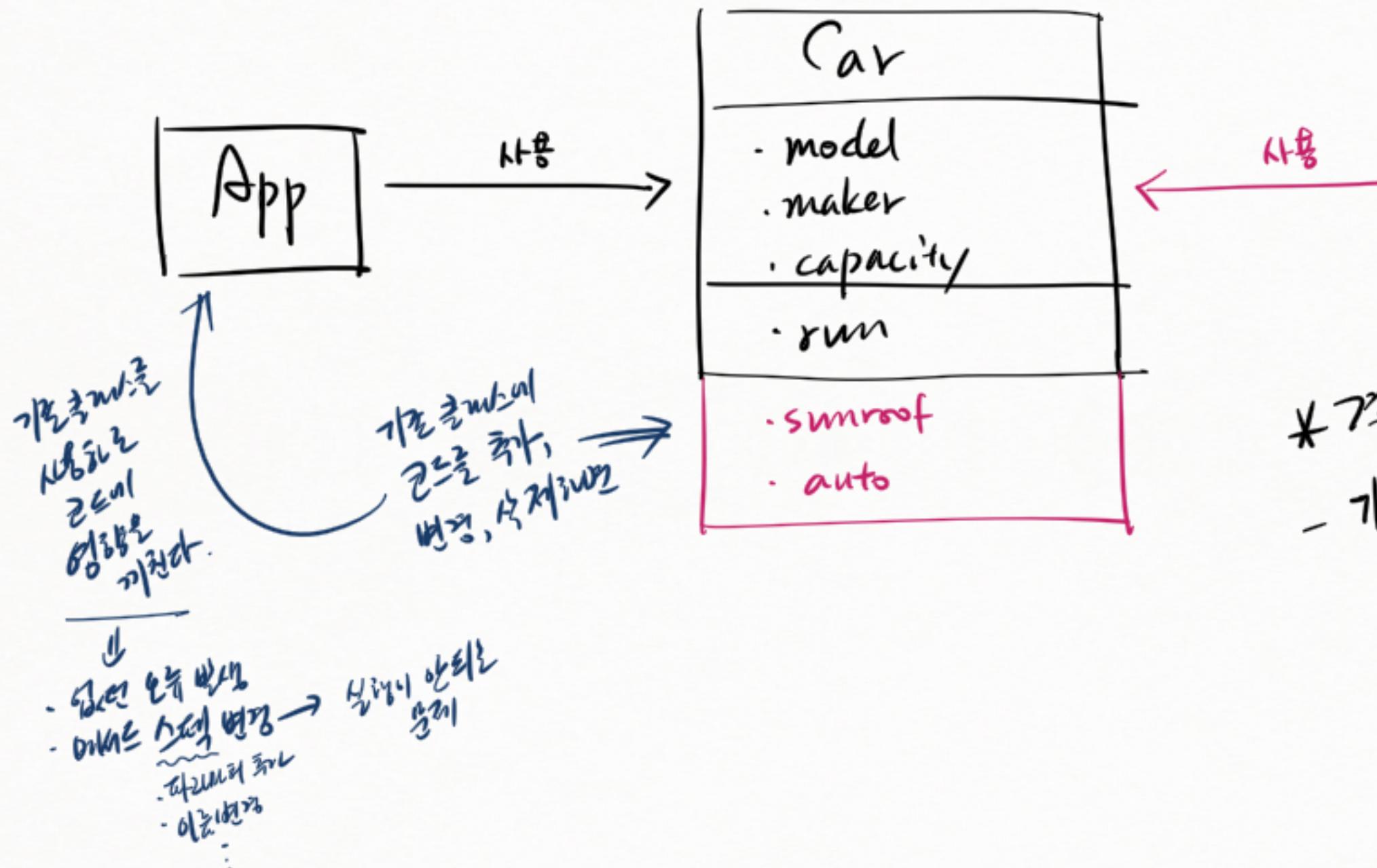
\* 접근 ?  
- 키워드로



상속 (Inheritance)

\* 기능 학장 — ① 기존 클래스를 연결

① 2021 A



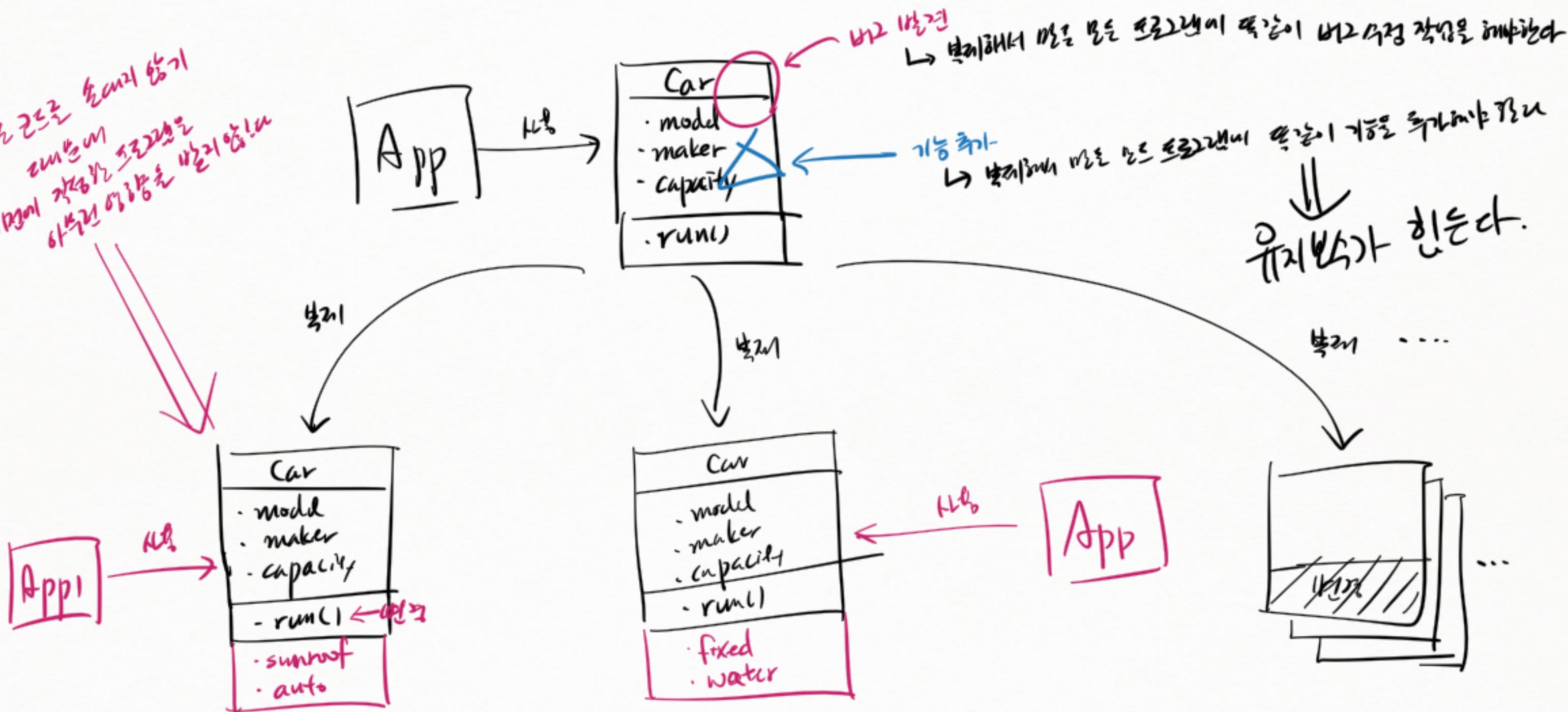
### ② 고급사 B : 기능 추가 - a/b

↓  
기존 큐레이션에 헌드 추가

- \* 기준  
- 기준 근드를 사용하는  
기준 근드를 사용하는  
기준에 영향을 미친다.

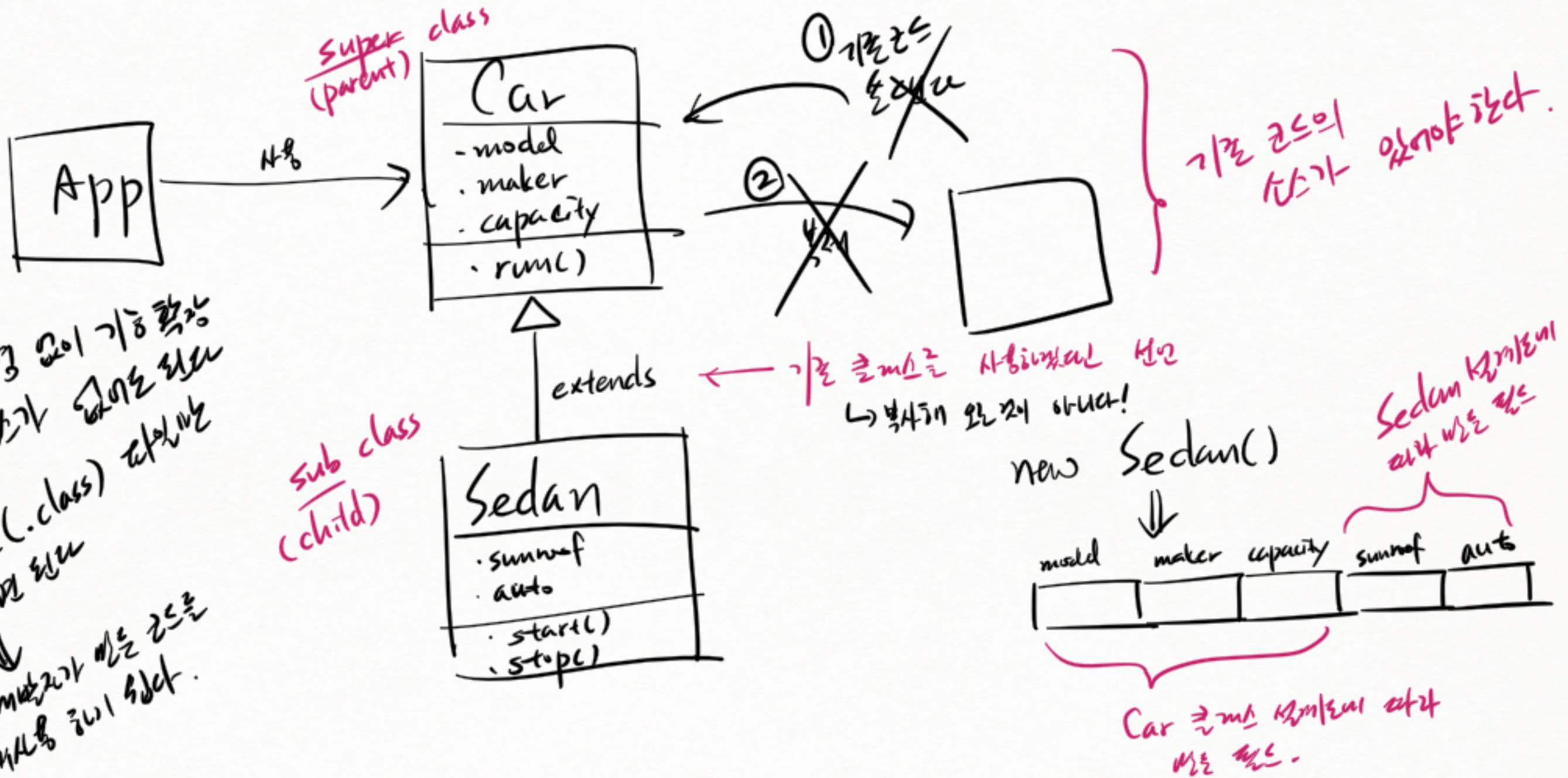
## \* 기능 확장 - ② 기존 코드를 복제한 후 기능 추가

기존 코드를 복제한 후 기능 추가  
기존 코드를 복제한 후 기능 추가  
기존 코드를 복제한 후 기능 추가  
기존 코드를 복제한 후 기능 추가  
기존 코드를 복제한 후 기능 추가  
기존 코드를 복제한 후 기능 추가

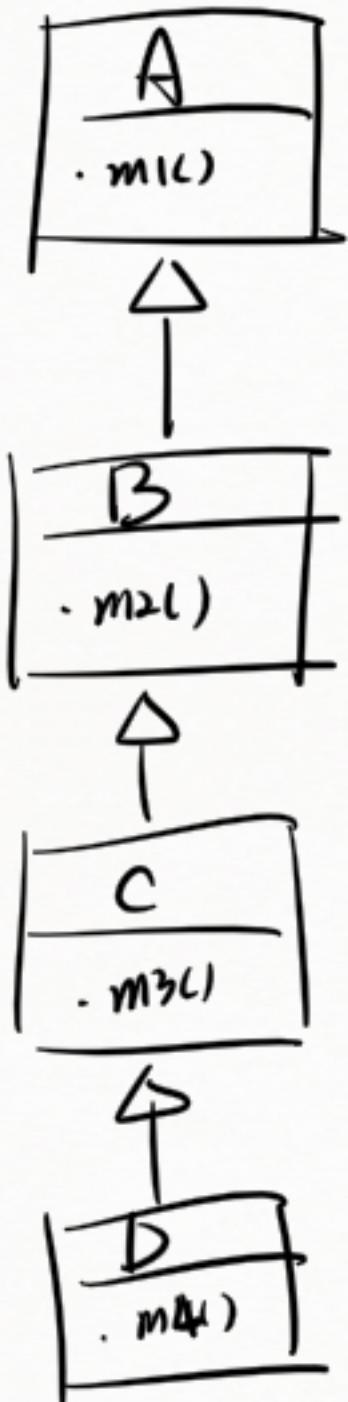


\* 가능성이 있는 - ③ 상속을 이용한 가능성을

- \* 가능성이 있는 - ③ 상속을 이용한 가능성을
- 가능성이 있는 이유는 상속을 이용해 코드 재사용이 가능해진다.
- 가능성이 있는 이유는 상속을 이용해 코드 재사용이 가능해진다.
  - 부모 클래스의 멤버를 자식 클래스에서 사용할 수 있다.
  - 부모 클래스의 멤버를 자식 클래스에서 사용할 수 있다.
  - 부모 클래스의 멤버를 자식 클래스에서 사용할 수 있다.
  - 부모 클래스의 멤버를 자식 클래스에서 사용할 수 있다.



\* 상속과 멤버드 체조.



B obj = new B();  
 obj |--- 200  
 ↓  
 ↗ 200 | ... | 300  
 ↗ 300 | 300  
 obj. m2();  
 obj. m1(); //ok  
 obj. m3();  
 obj. m2();  
 obj. m1();

D obj = new D();  
 obj |--- 400  
 ↓  
 ↗ 400 | 400  
 obj. m4();  
 obj. m3();  
 obj. m2();  
 obj. m1();

B obj = new D();  
 obj |--- 500  
 ↓  
 ↗ 500 | 500  
 obj. m4();  
 obj. m3();

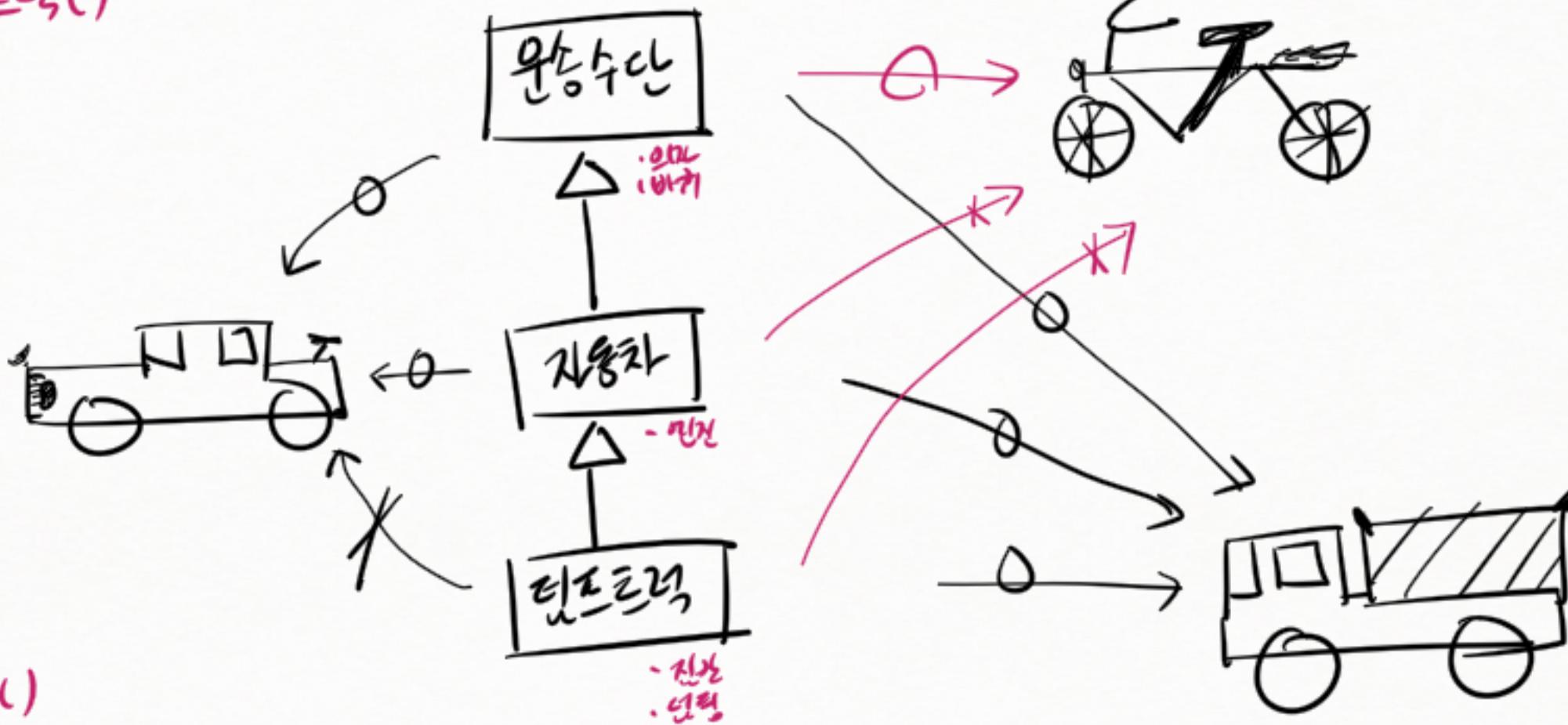
( obj. m4();  
 obj. m3(); )  
 ↗  
 컨타워려는 변수(리퍼런스)의  
 대입에서 메서드를 찾아 올라온다.  
 리퍼런스가 실제 어떤 클래스의  
 인스턴스를 가리키는 것인지  
 따지지 않는다.

obj. m2(); } ok!  
 obj. m1(); }

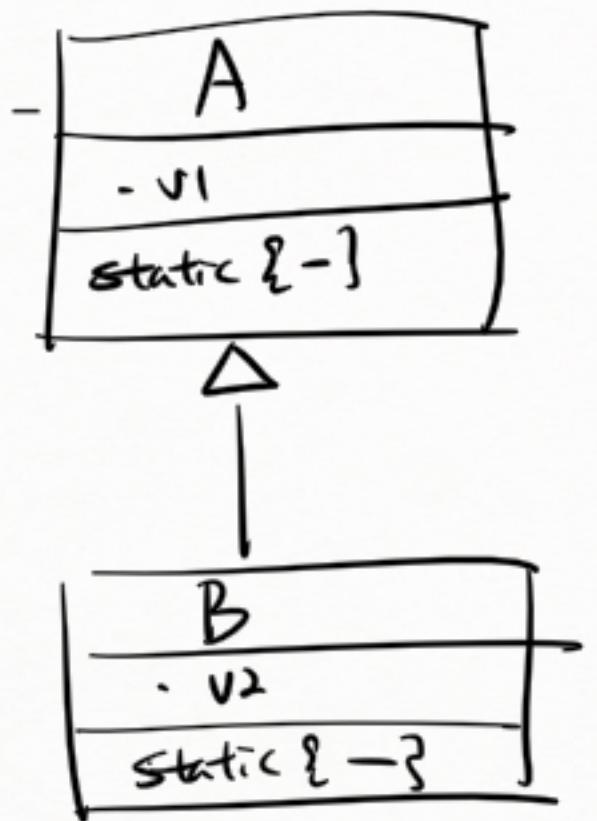
터프트럭 r1 =  
기동차 r2 = new 터프트럭()  
운송수단 r3 =

터프트럭 r1 ≠  
기동차 r2 = new 기동차()  
운송수단 r3 =

터프트럭 r1 ≠  
기동차 r2 ≠ new 운송수단()  
운송수단 r3 =

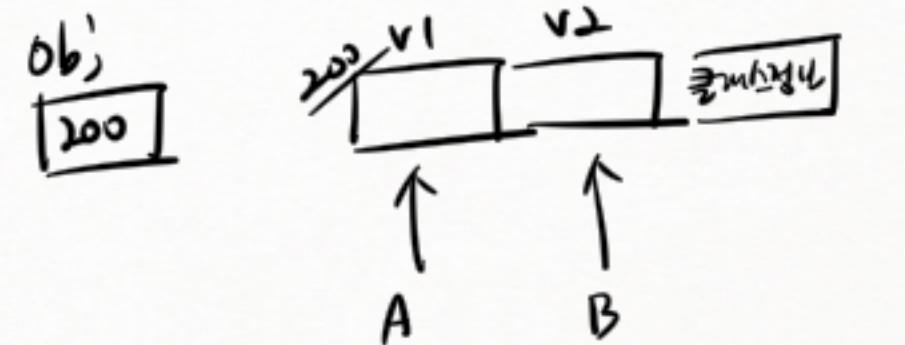


\* 상속과 상속 관계

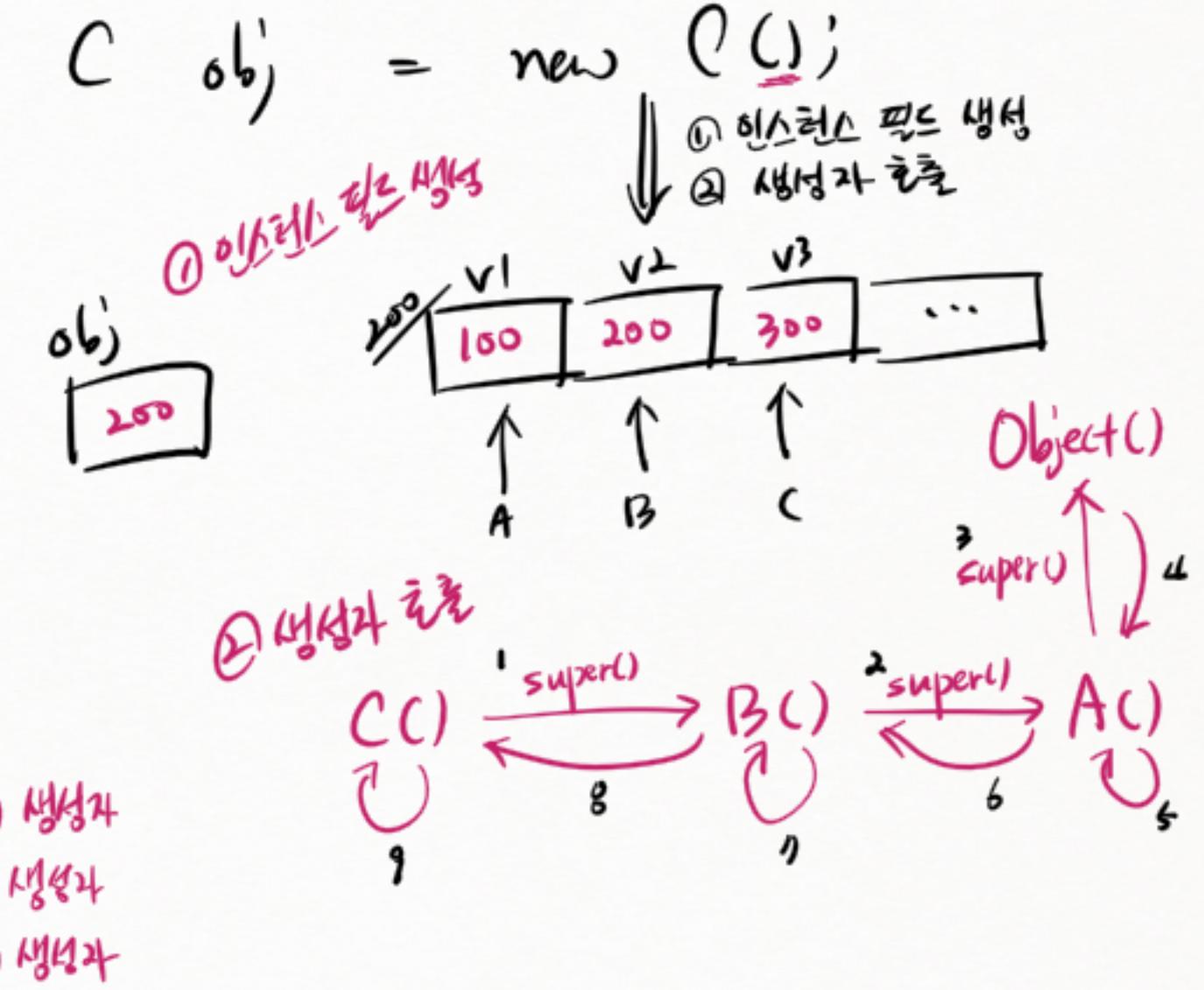
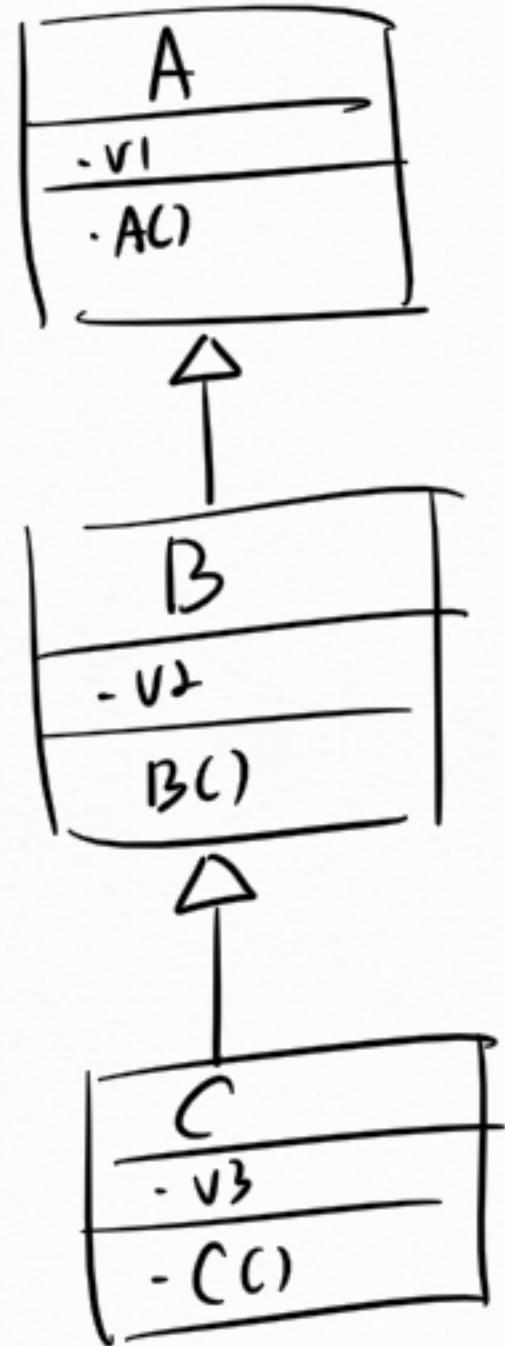


B obj = new B()

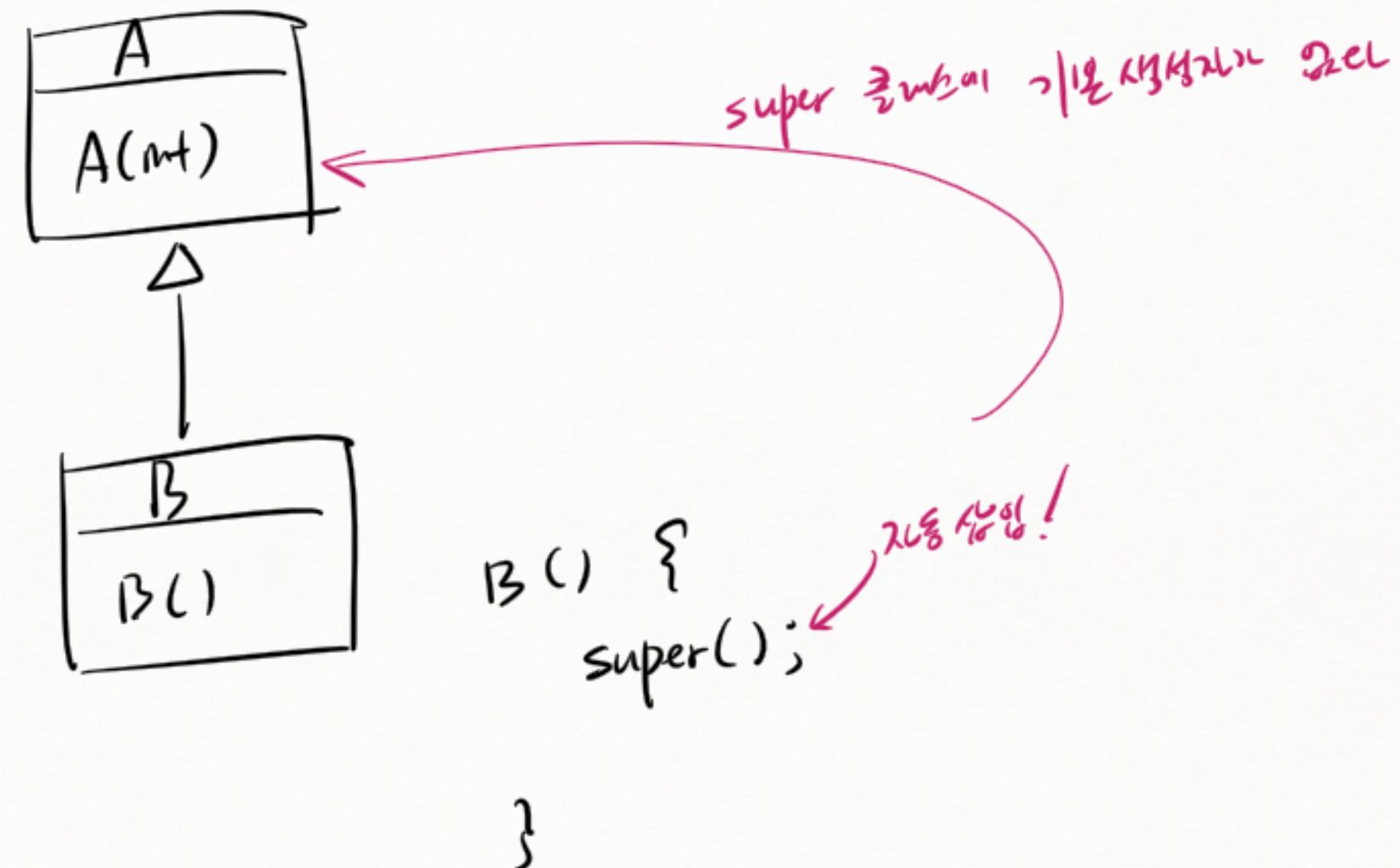
- ① 수퍼클래스부에 초기화
- ② 수퍼클래스부에 인스턴스 필드 생성



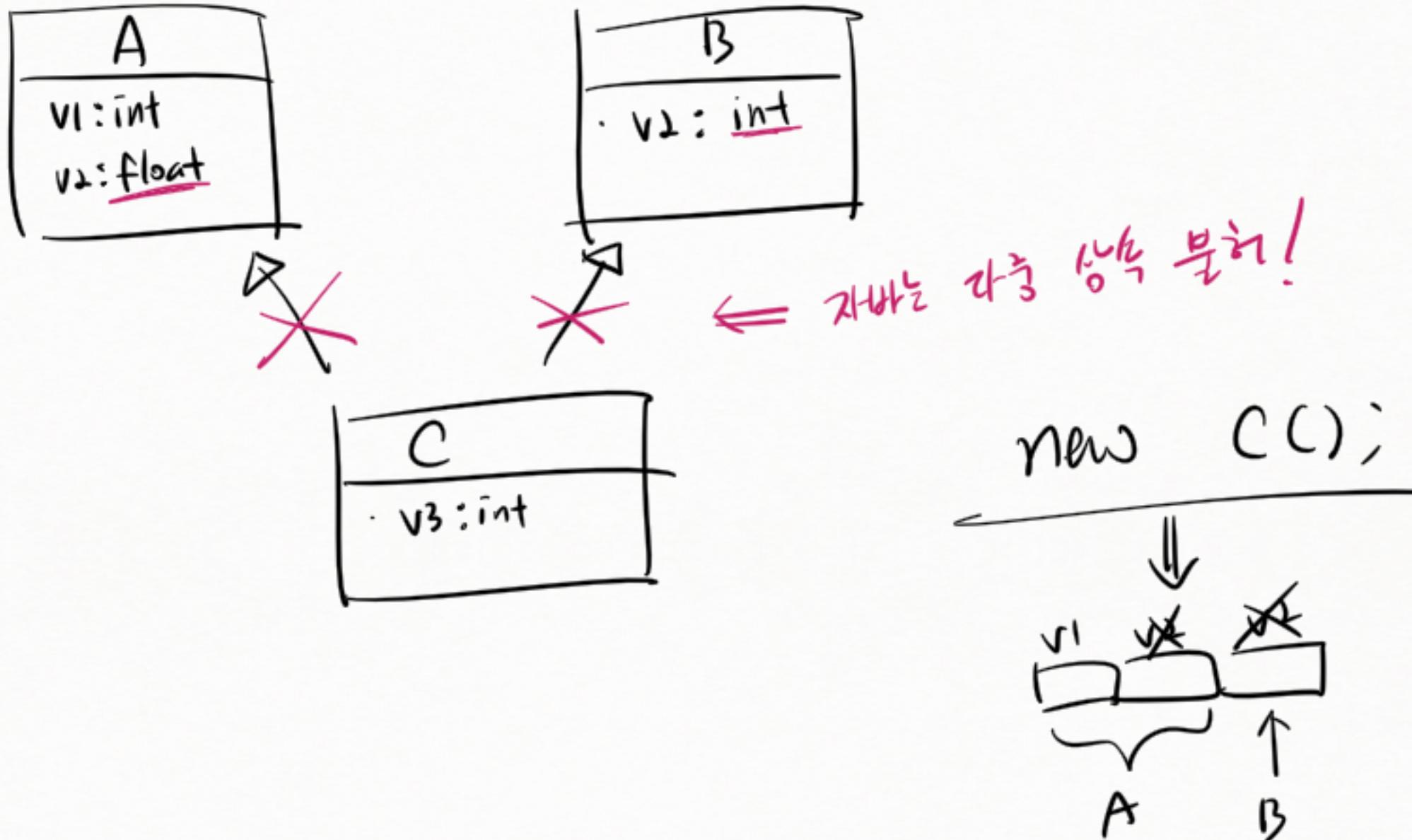
## + 생성자 호출 순서



\* 초기 층 클래스의 생성자 호출

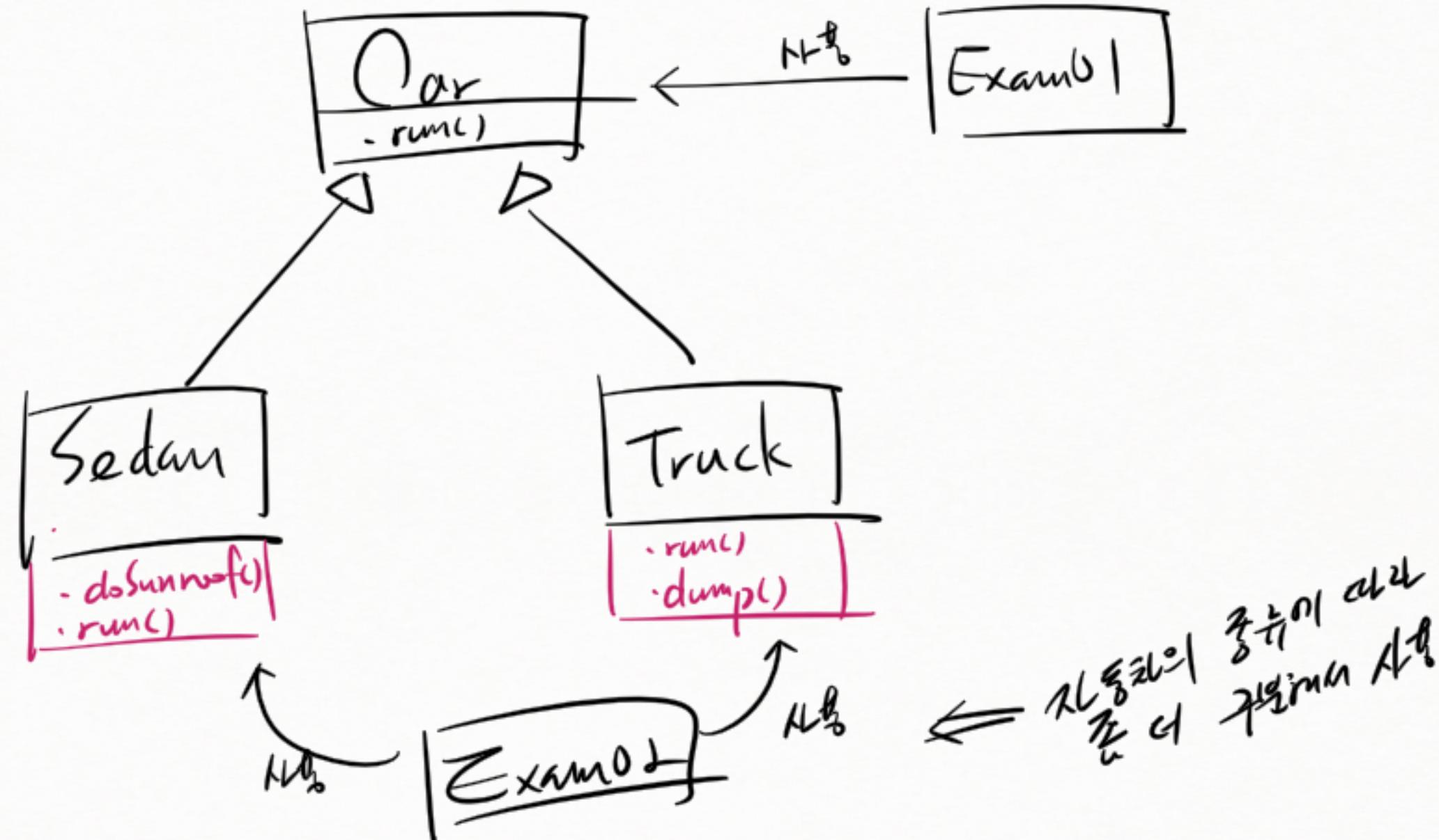


\* 다음 상속?

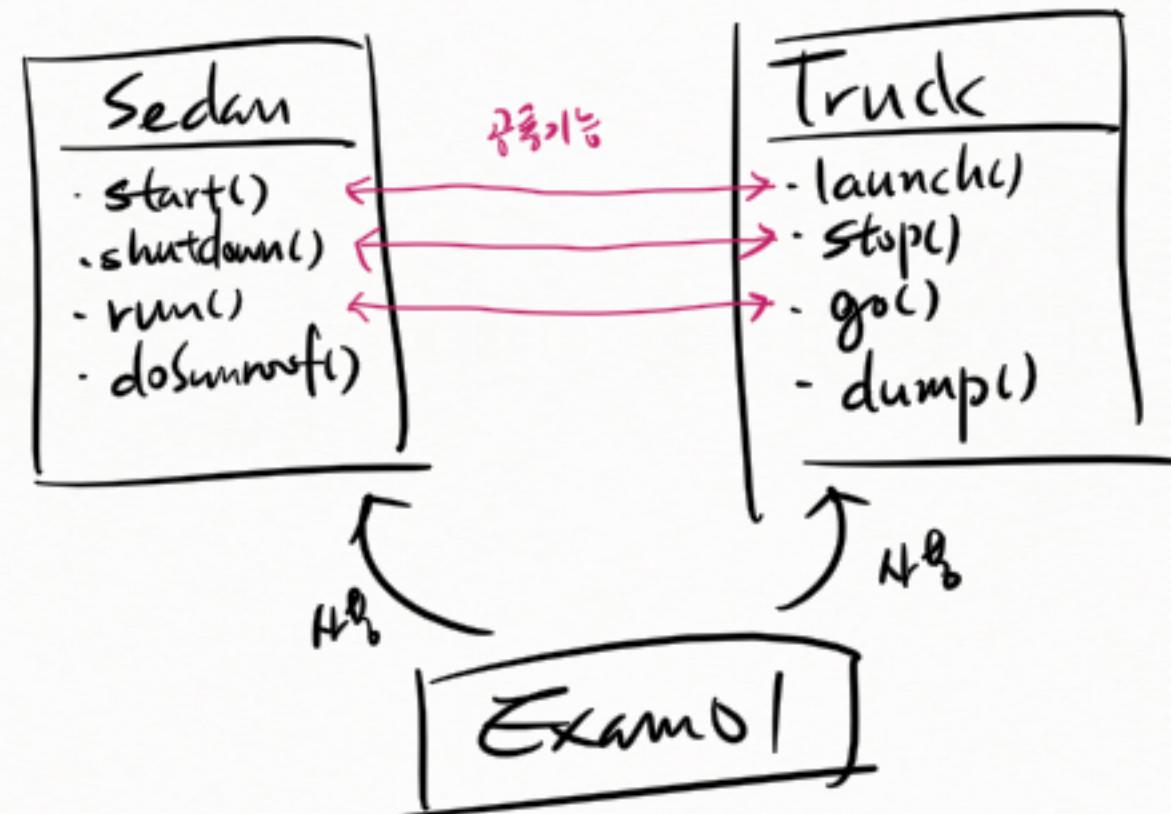


## \* Specialization

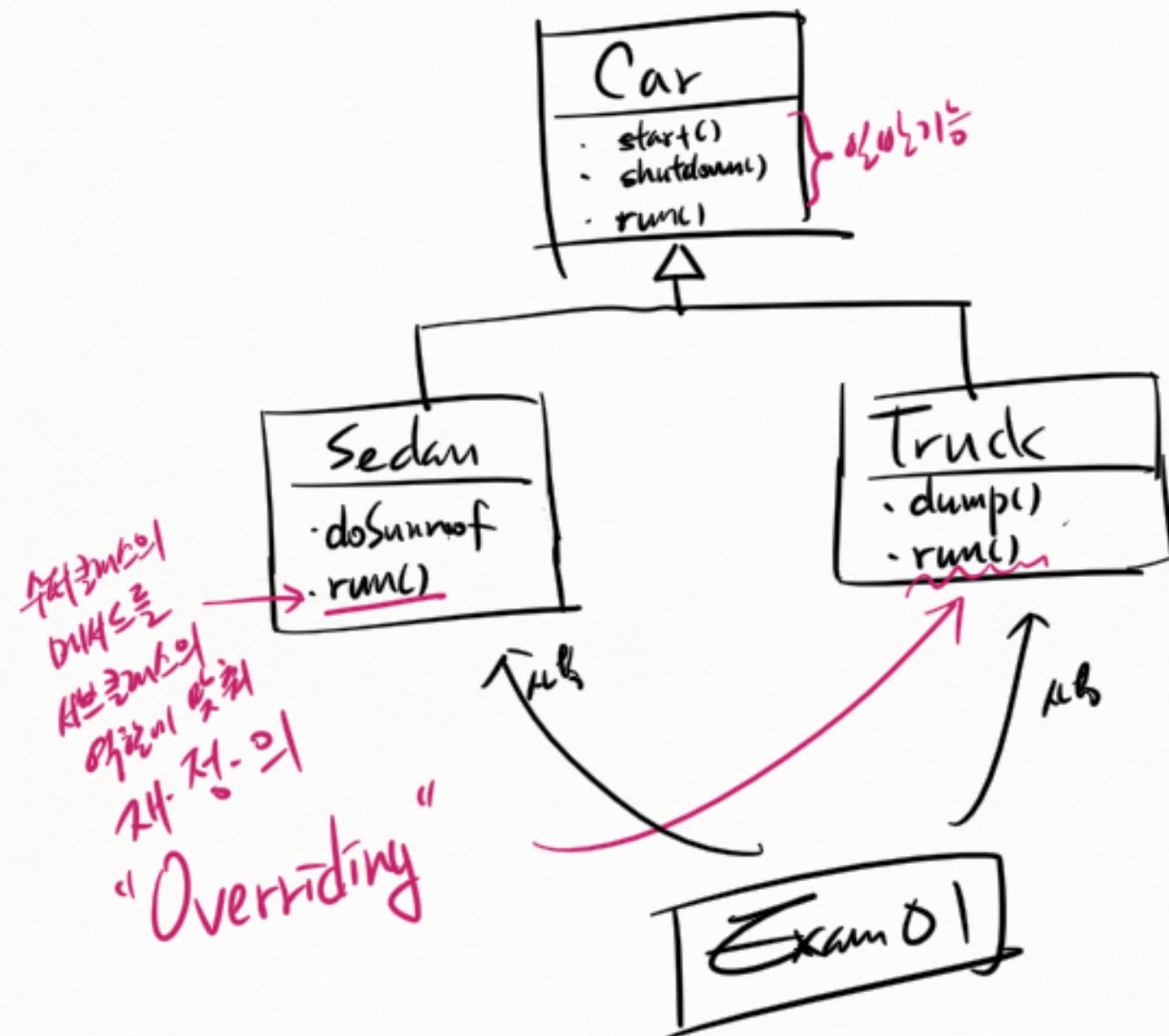
↑  
↑  
↓  
↓  
↓



\* generalization

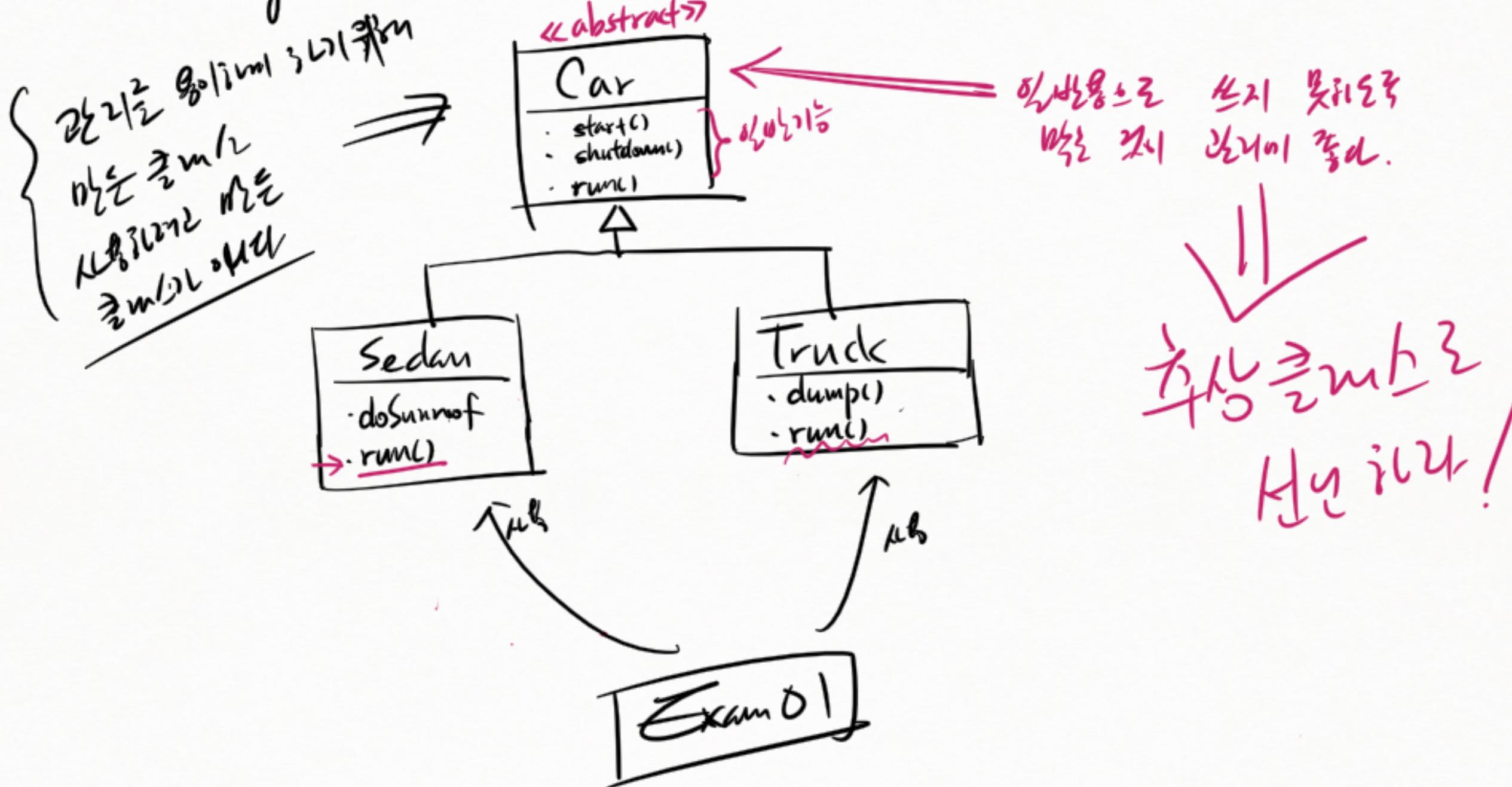


## \* generalization

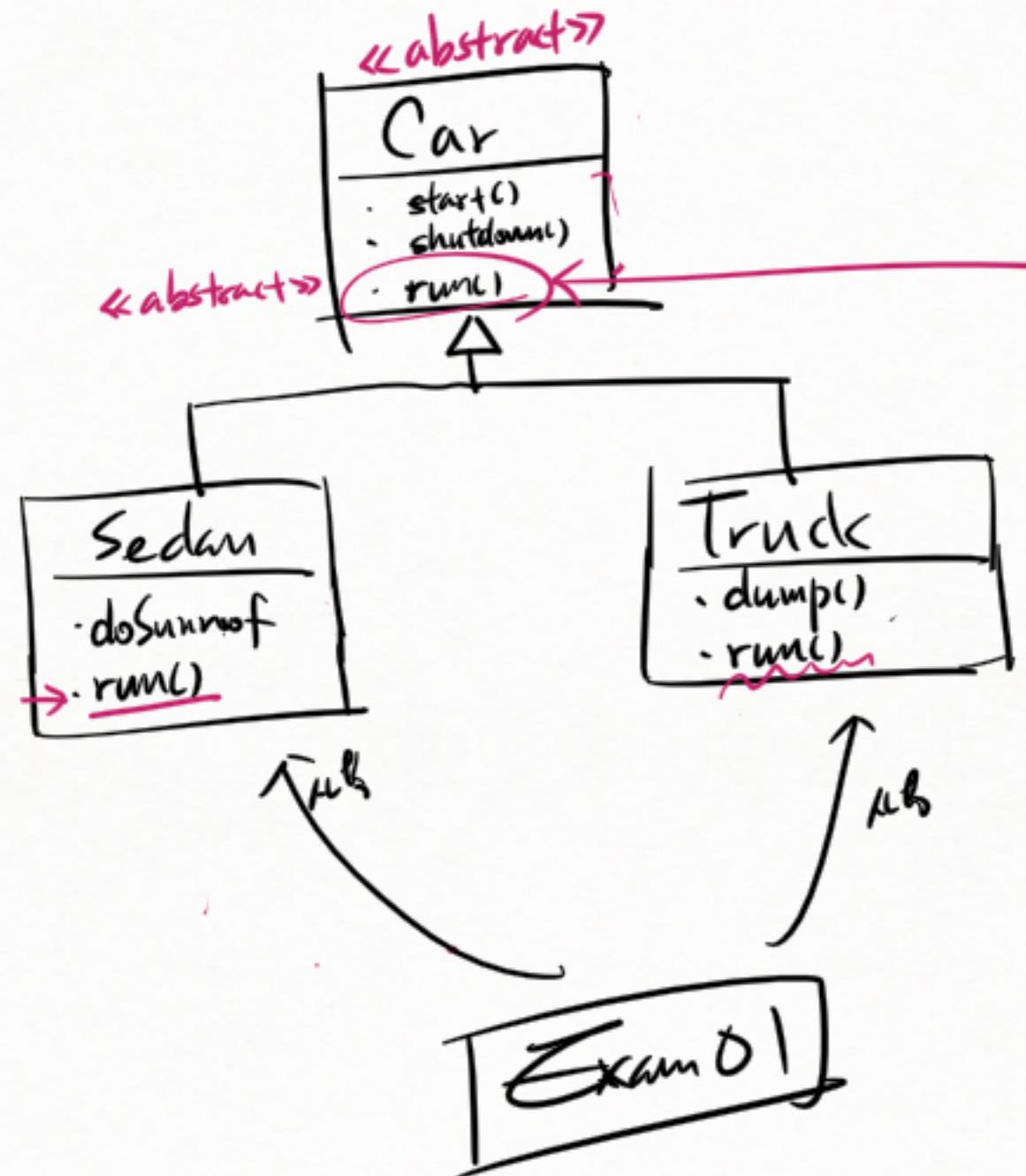


↑  
선택한 메소드를  
부모에게 물려받아  
공통 기능 / 재정의 가능  
부모의 메소드를  
재정의하는 것!  
"generalization",  
"abstraction"

\* generalization : 추상 클래스



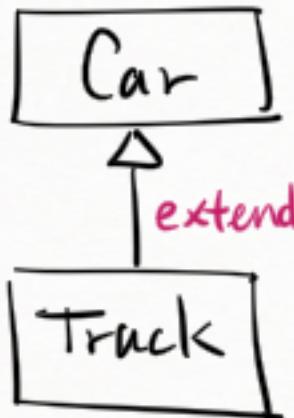
\* generalization : 추상 클래스



Abstraction  
Generalization  
Specialization  
Generalization  
Abstraction  
Generalization  
Generalization  
"Abstraction"

\* 여기서 잊기 쉬운 것들 - 클래스 간의 관계와 UML

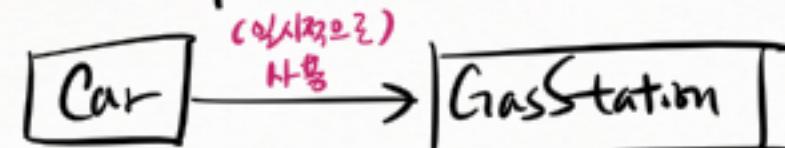
① 상속 (inheritance)



class Track  
extends Car {



⑤ 의존 (dependency) : 특정 객체에서 의존으로 사용

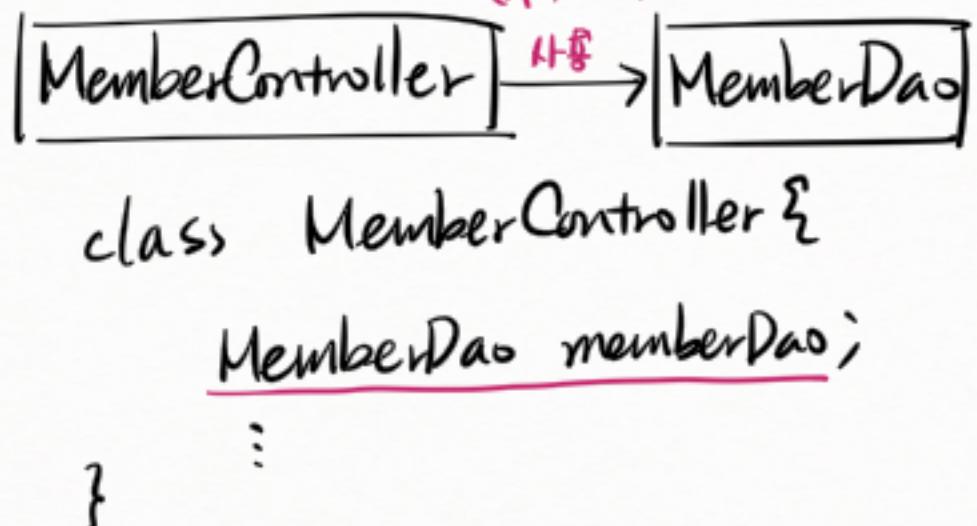


class Car {  
void oil(GasStation statin){

Container  
lifecycle  
||  
Containee  
lifecycle

(자식클래스)  
사용

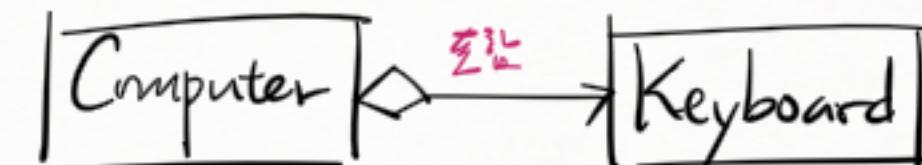
② 연관 (association)



사용

(자식클래스)

③ 집합 (aggregation) 약관

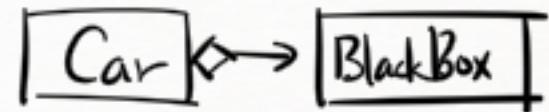


포함

class Computer {

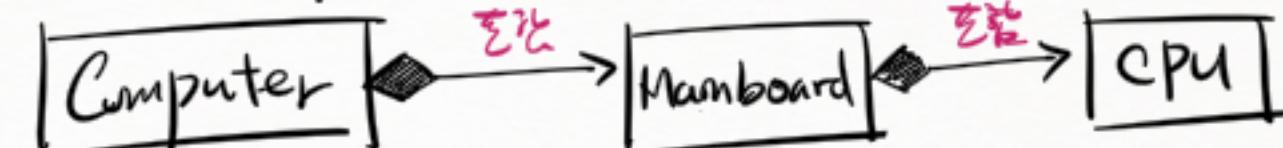
Keyboard keyboard;

:



Container ≠ Containee  
Lifecycle

④ 핍합 (composition) 강관



포함



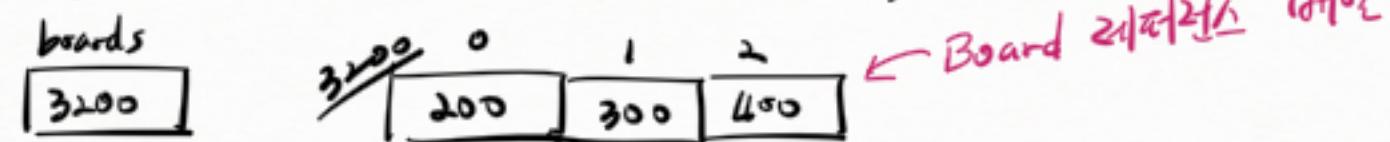
class Computer {  
Mainboard mb;

class Mainboard {  
CPU cpn;

:

\* 21/21/21 - 11:10

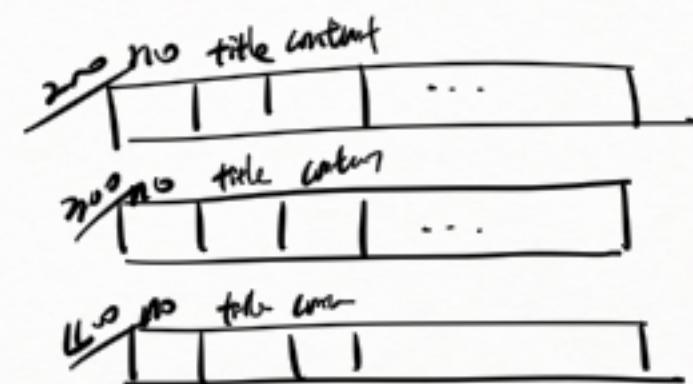
Board[] boards = new Board[3];



boards[0] = new Board();

boards[1] = new Board();

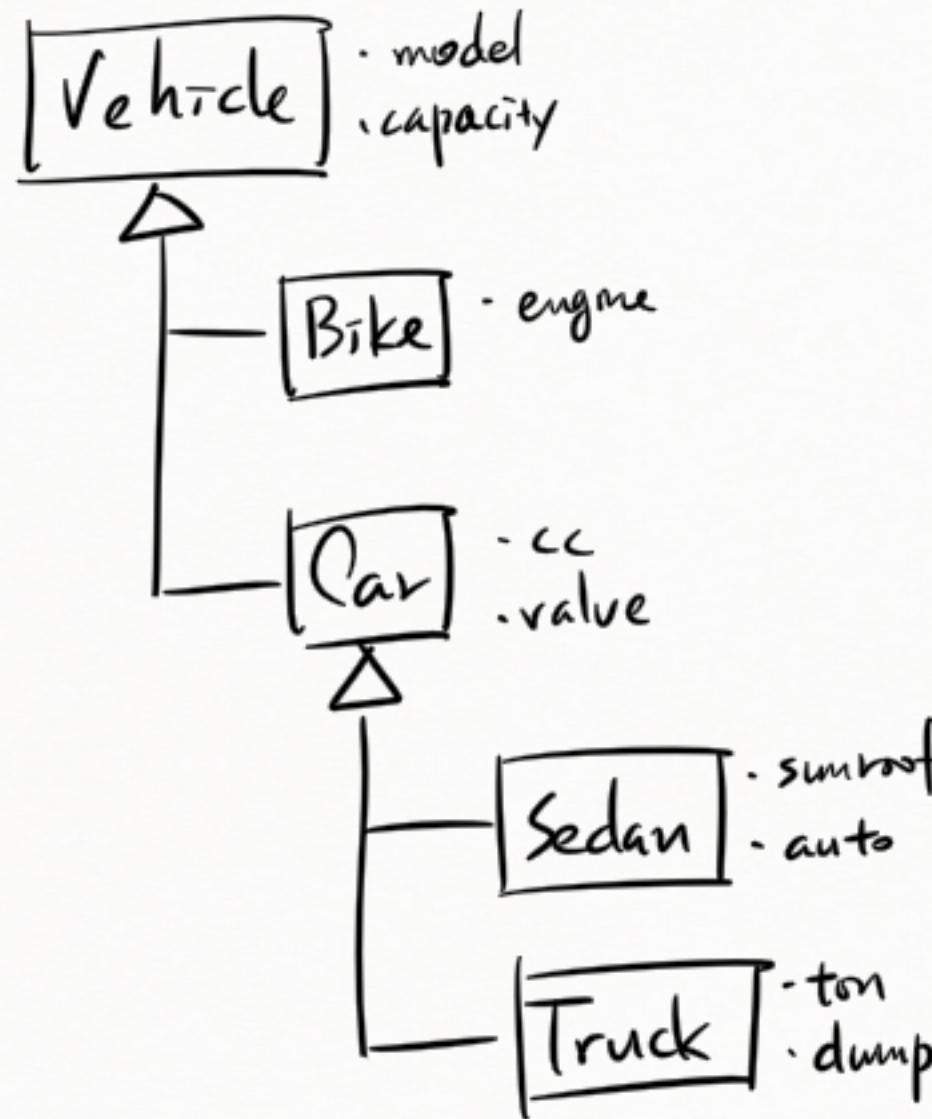
boards[2] = new Board();



## 다형성 (polymorphism)

- polymorphic variable (다형적 변수)
- overloading (오버로딩)
- overriding (오버라이딩; 메소드 재정의)

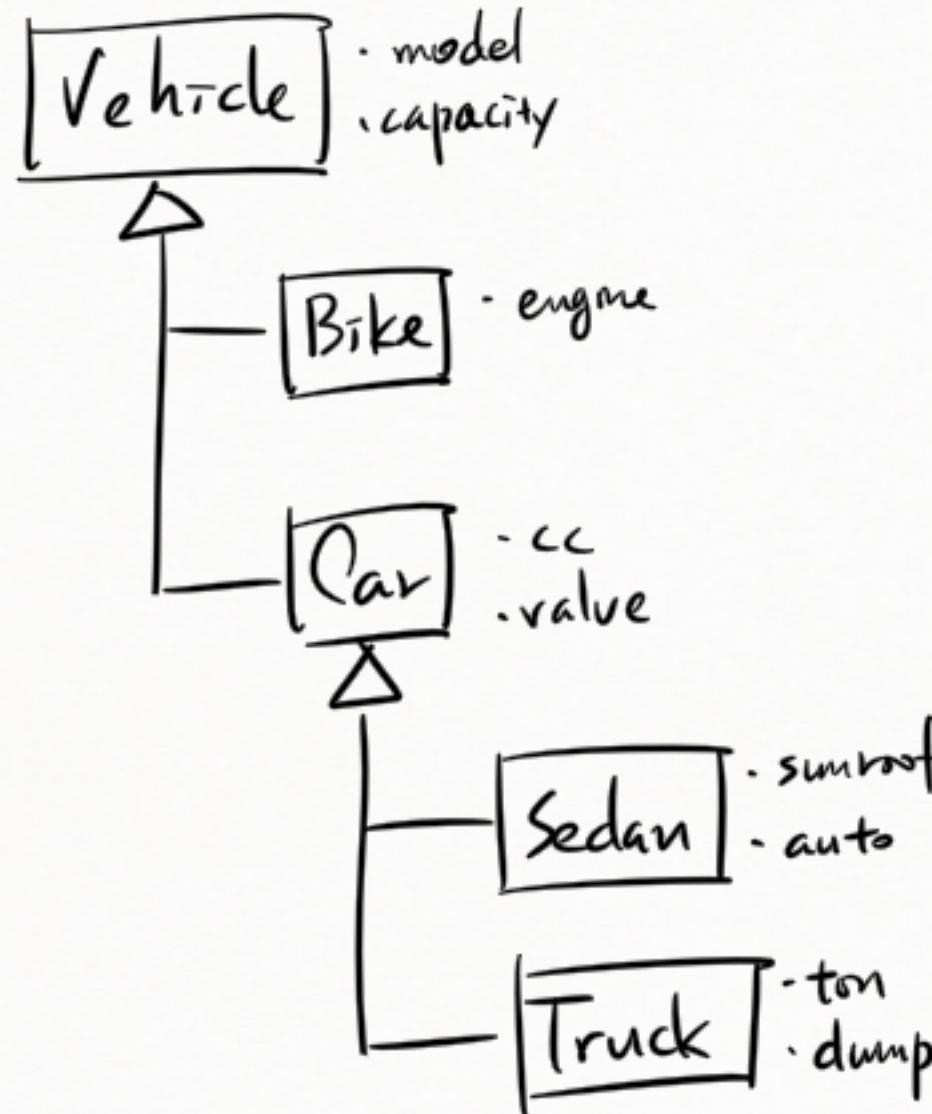
## \* polymorphic variables



sub class of  
이스 퍼스 차/트  
차량은 차/트

Vehicle v;  
v = new Vehicle();  
v = new Bike();  
v = new Car();  
v = new Sedan();  
v = new Truck();

\* 대형제 1번수와 헌법학자



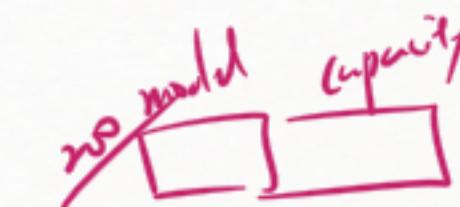
Vehicle obj

```
obj = new Car();
```

Car c = ((Car) obj)

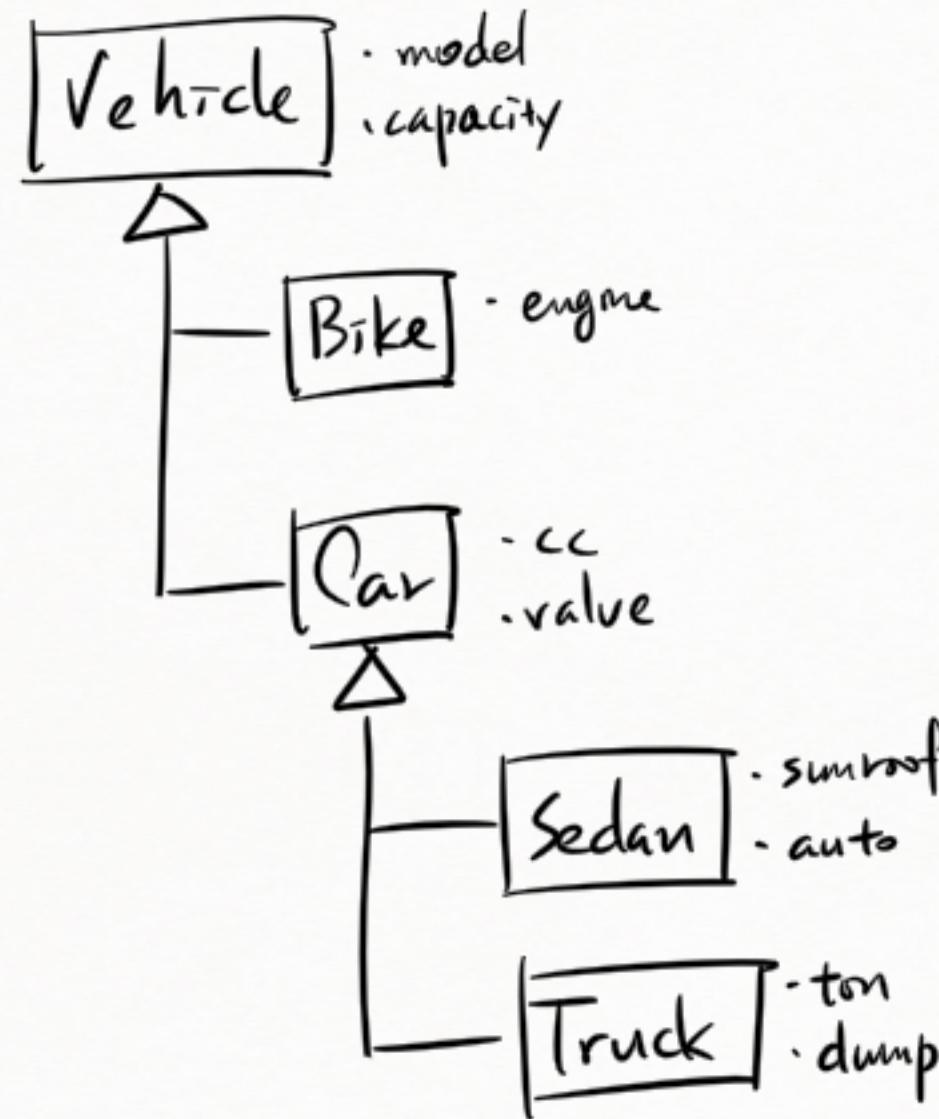
obj = new Vehicle()

~~Car c = (Car) ob~~



// 컨파일러는  
승인권자  
그리고  
JVM에서  
실행되는 cm 예제 실행.

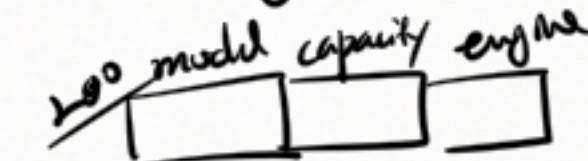
\* object oriented



Vehicle obj;

obj = new Bike();

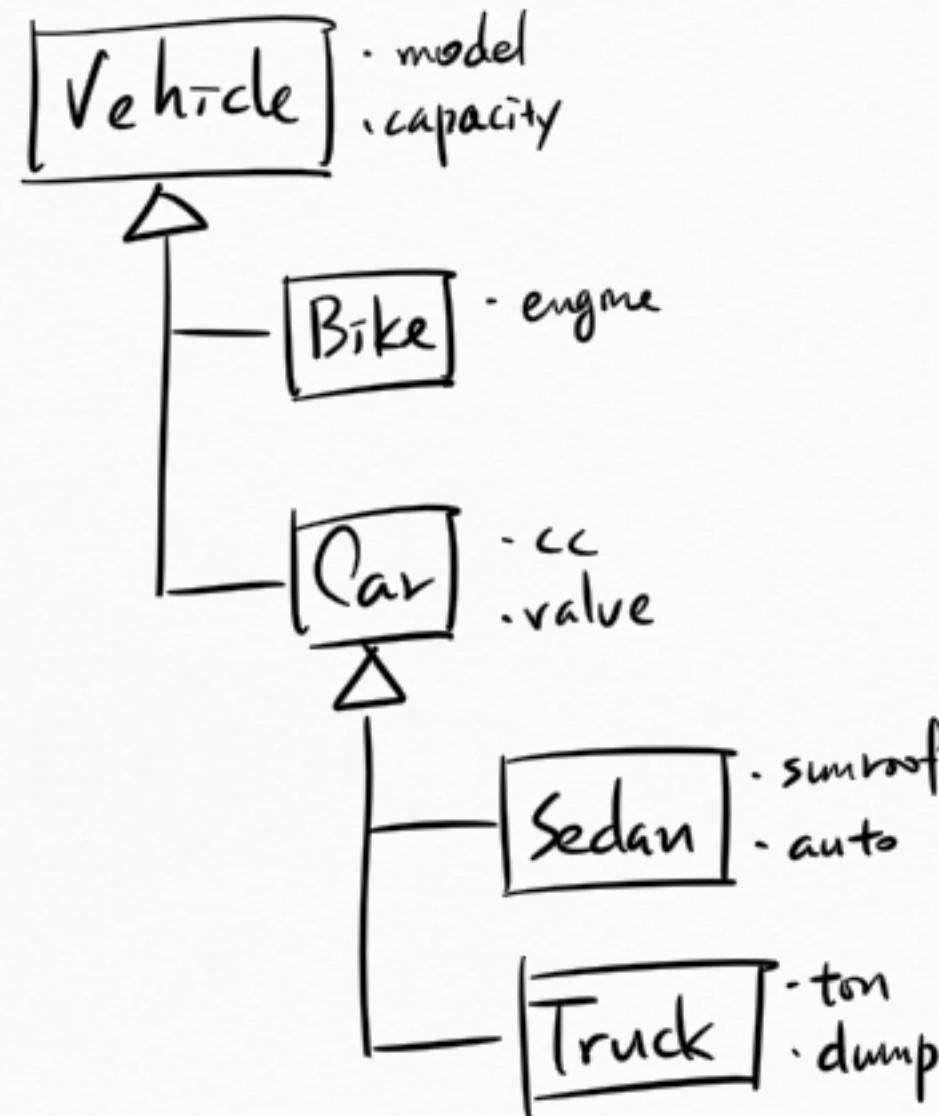
200



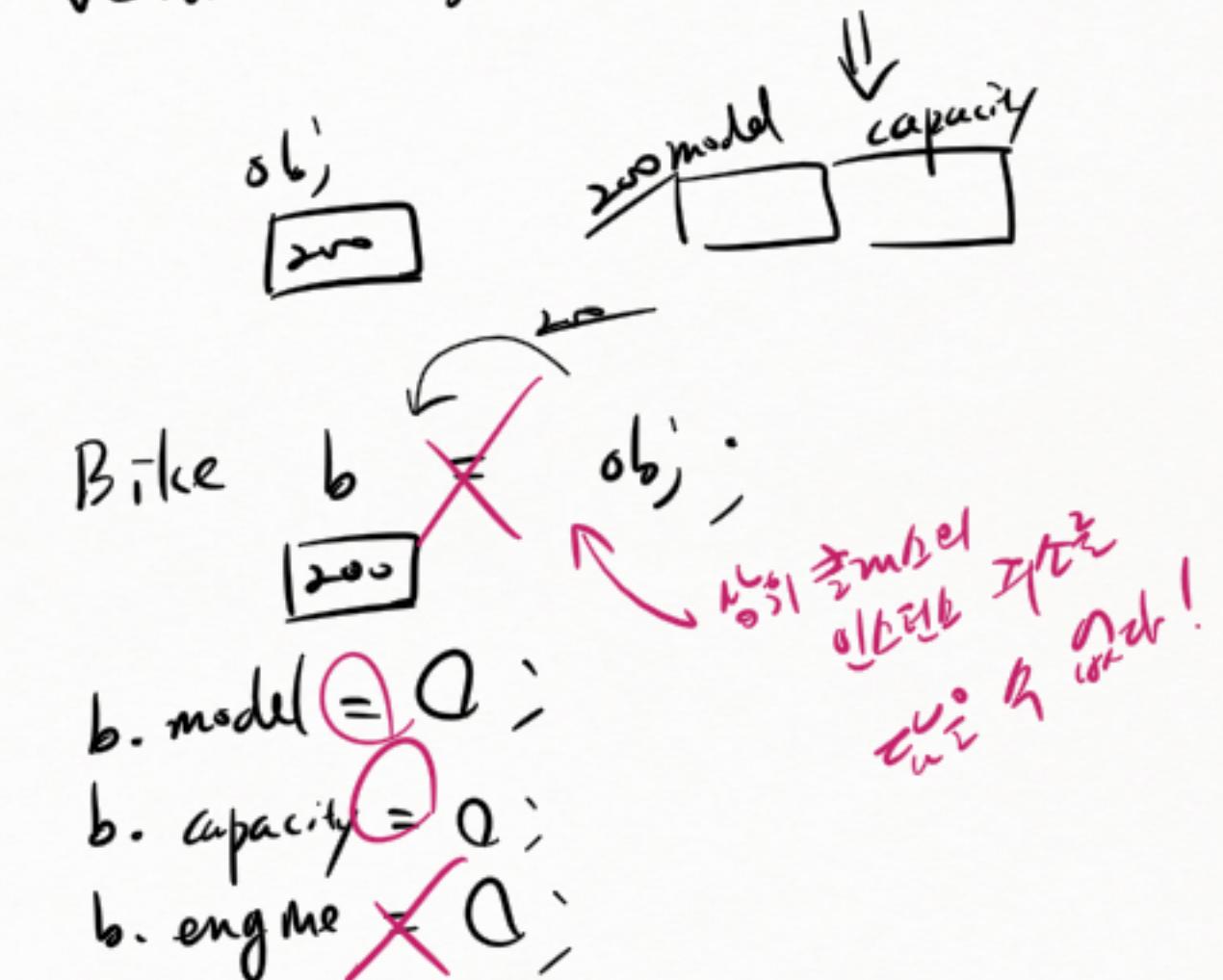
obj. model = 0;

obj. capacity = 0;

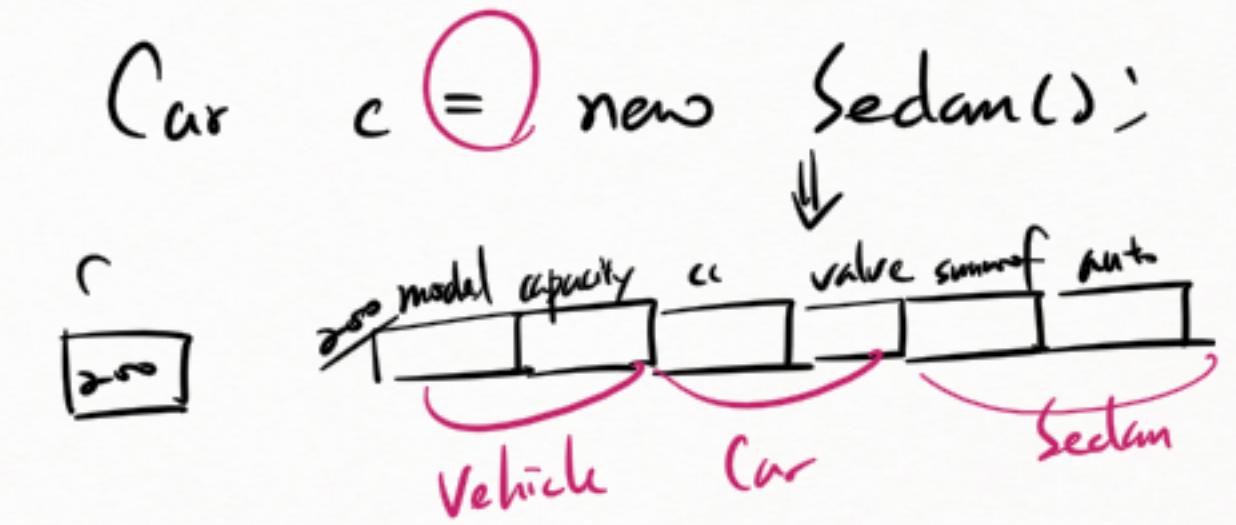
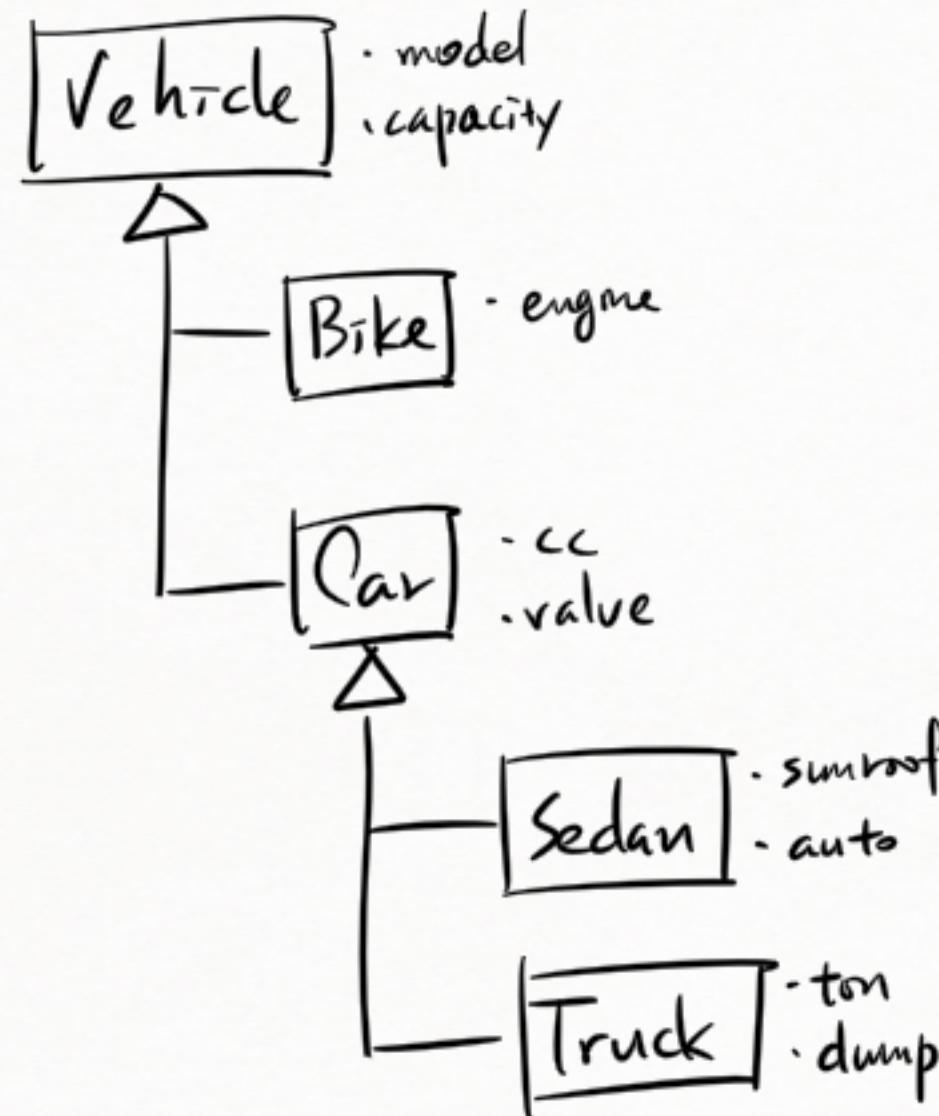
\* class의 멤버는



Vehicle obj = new Vehicle();



\* class inheritance



- c. model = 0;
- c. capacity = 0;
- c. cc = 0;
- c. value = 0;
- c. sunroof ≠ 0;
- c. auto ≠ 0;

(가장 초기화된 인스턴스에는  
생성할 때 Sedan 클래스가 초기화되었기 때문에  
생성할 때 c는 Car 초기화되었기 때문에  
Car의 초기화된 상태와 같다.)