

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

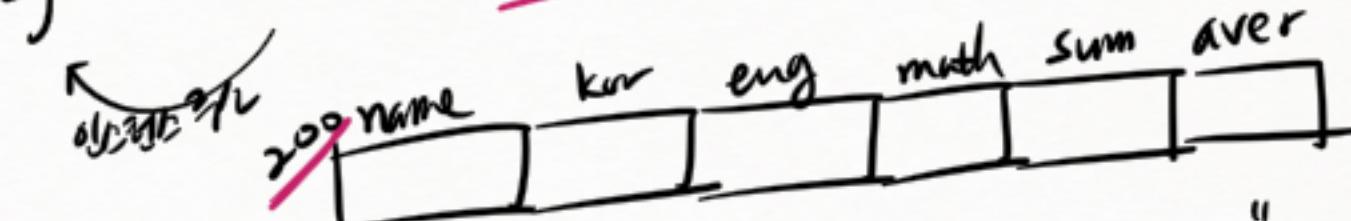
obj. name = "홍길동"

② 인스턴스 할당

```
Score obj;
```

obj
200

obj = new Score();



③ 인스턴스 사용

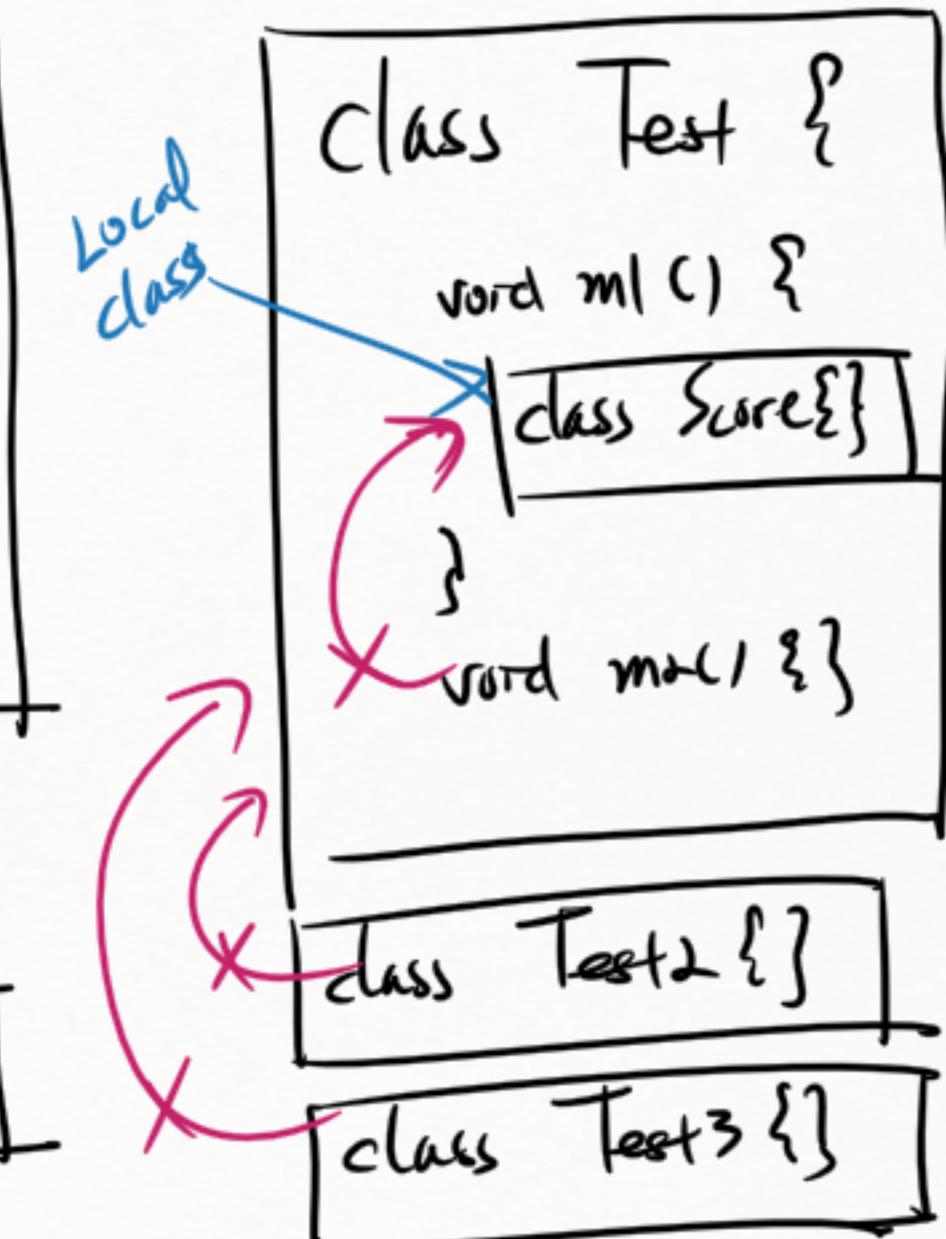
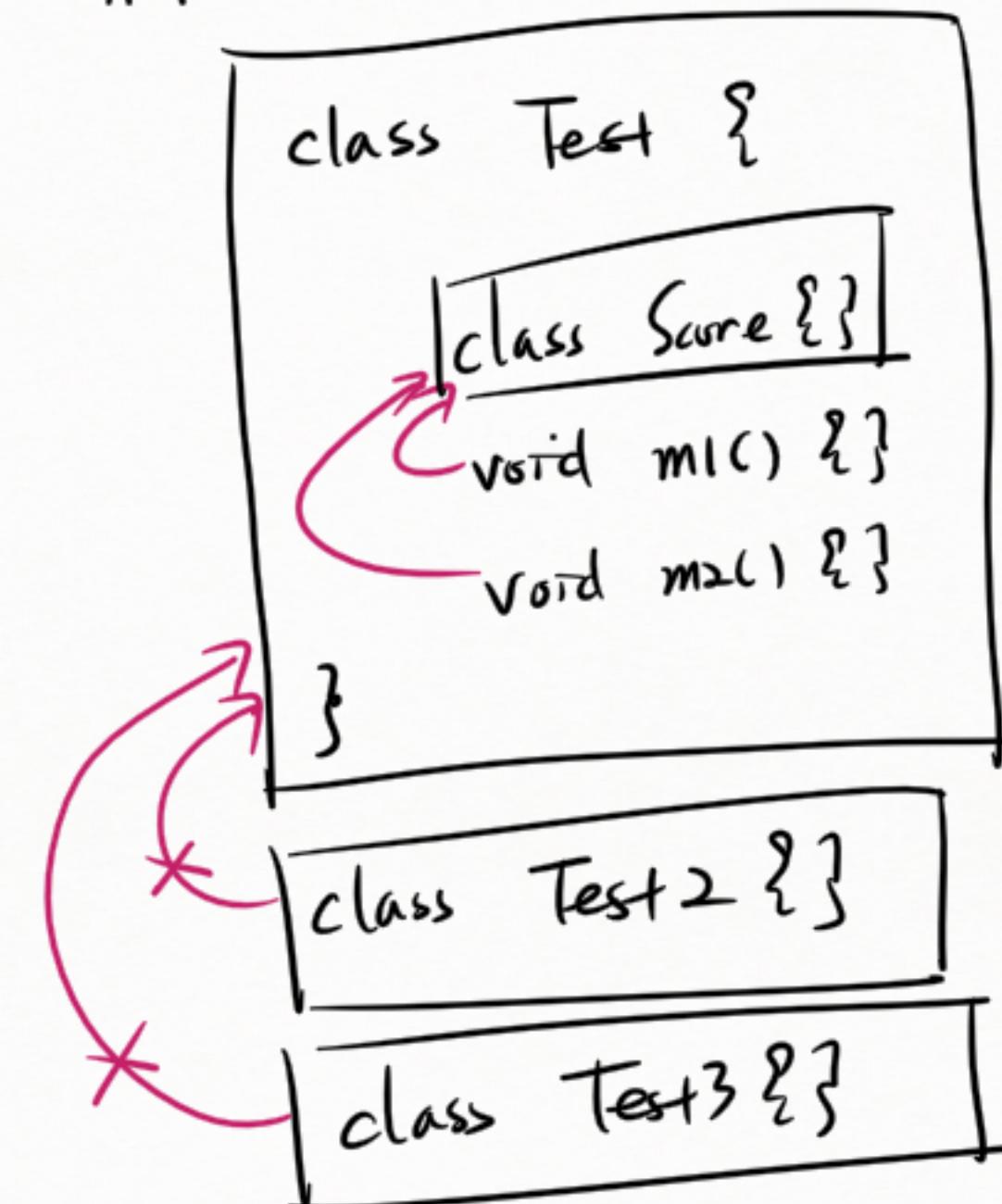
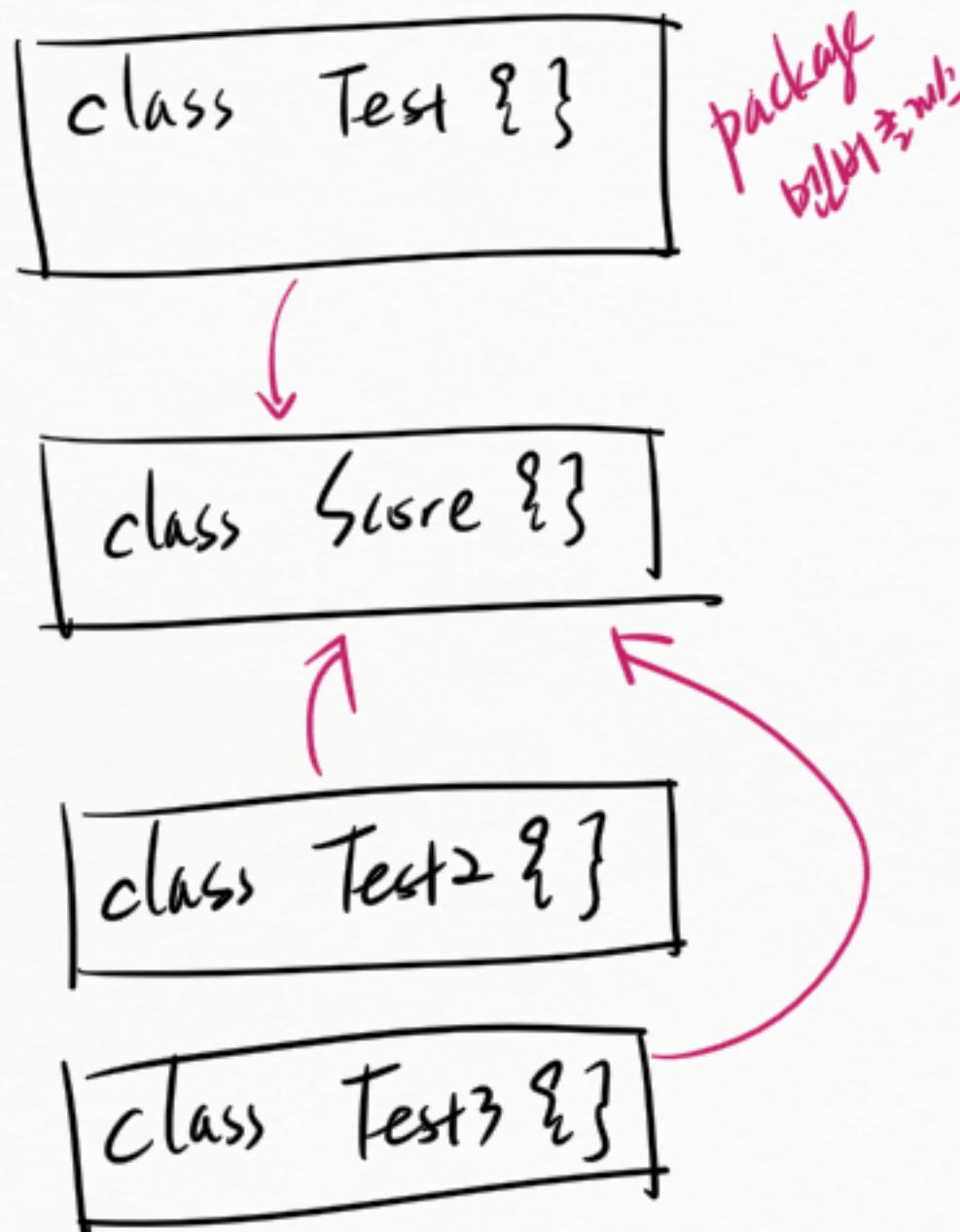
"Score의 인스턴스"
instance

"reference"

↑
Score의 인스턴스 주소를 저장하는 변수

Heap
obj → [] kor [] eng [] math [] sum [] aver

* 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

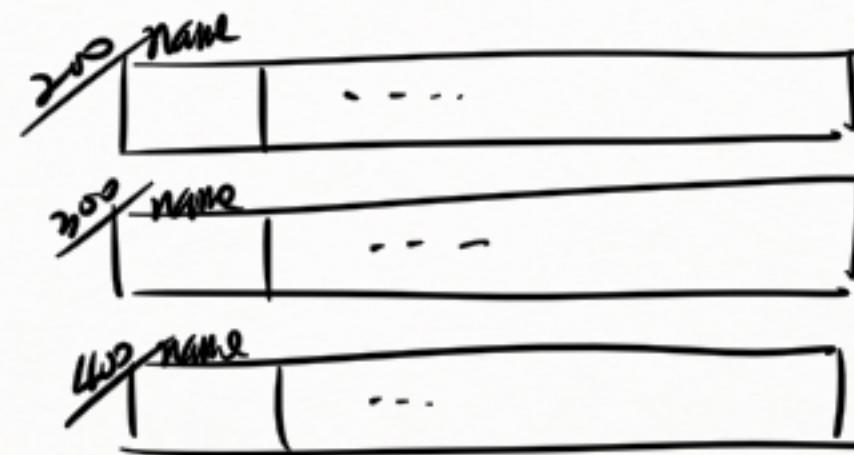
Score s1, s2, s3;

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`

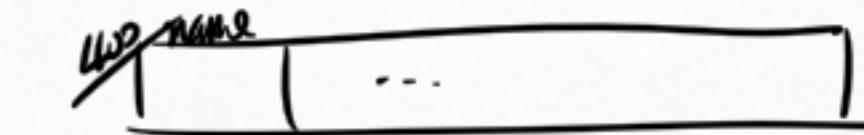
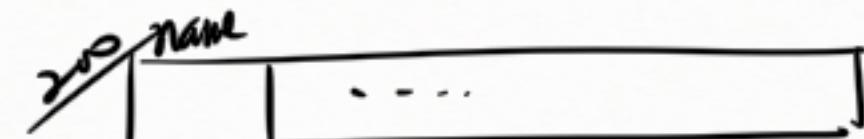
`1700`



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

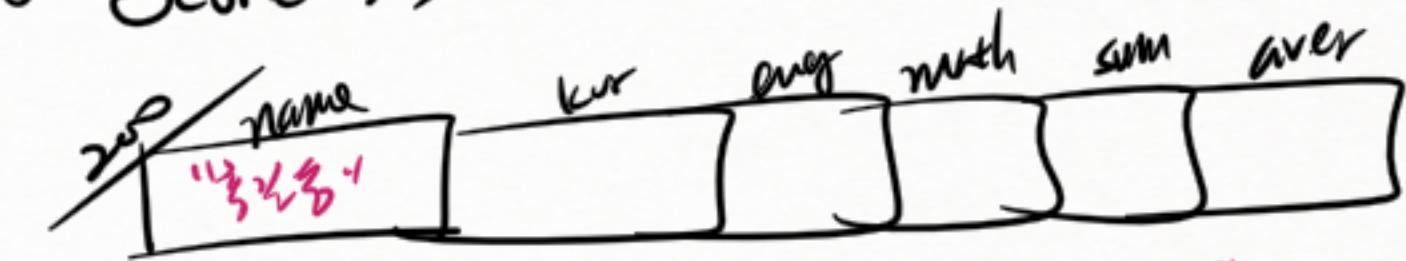
`scores[1] = new Score();`

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



* 인스턴스와 메서드:

Score s1 = new Score();



"Score의 인스턴스"
가짐

Score s2 = s1;

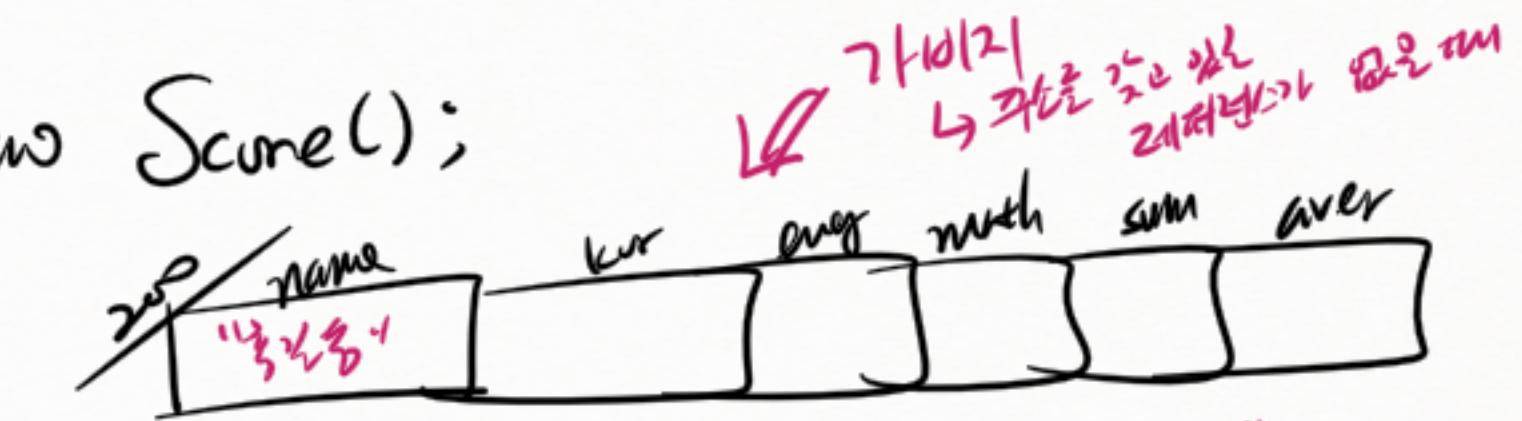


* 111011 (Garbage)

Score s1 = new Score();

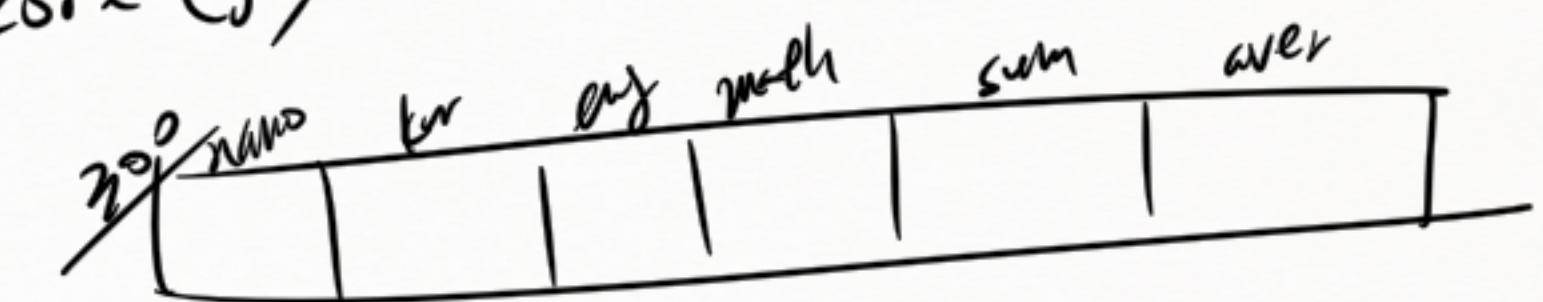


s1 = new Score();



"Score의 인스턴스"

"가장"

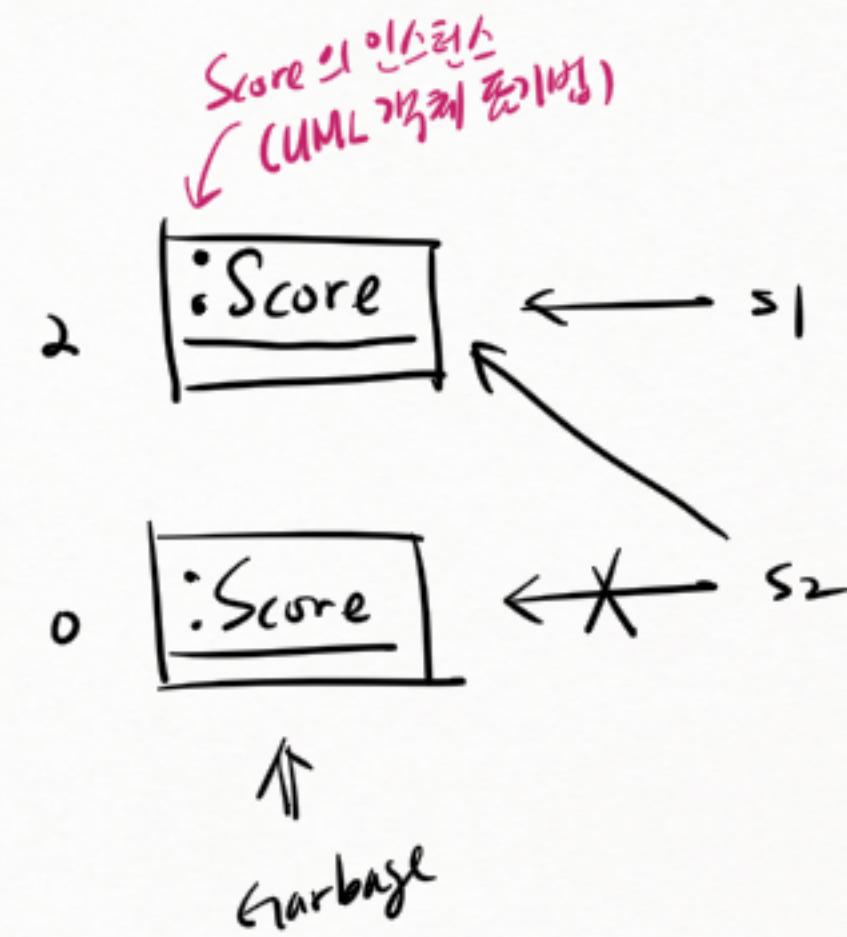


* 인스턴스와 리퍼런스 차운트

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

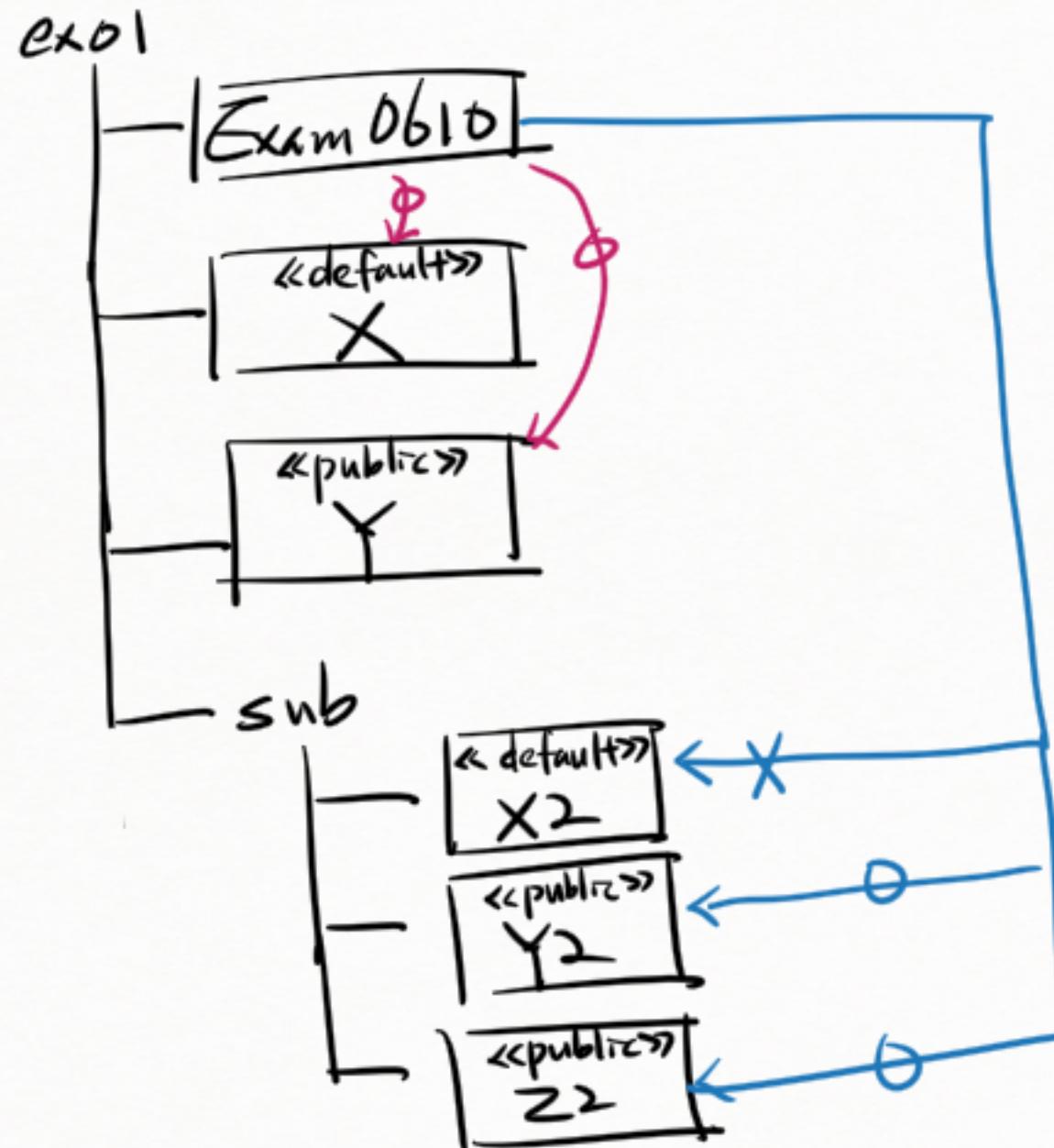
```
Score s2 = new Score();
```

```
s2 = s1;
```



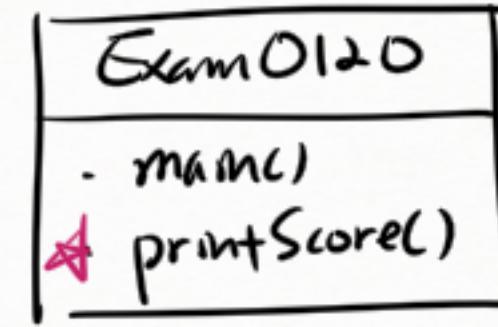
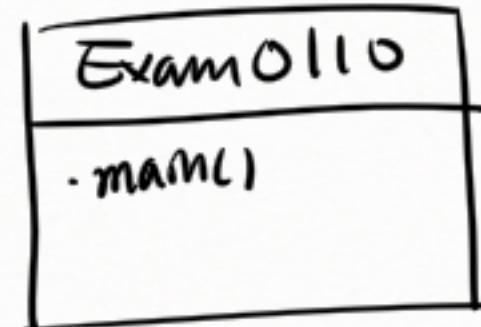
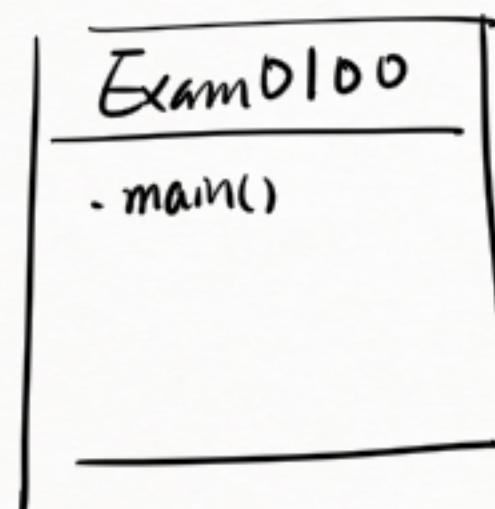
* public $\frac{3}{2}$ mLcf

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



- 메모리와 인스턴스
- ↓
 new
 ↓
 Heap 영역
 ↓
 garbage
 ↓
 garbage collector



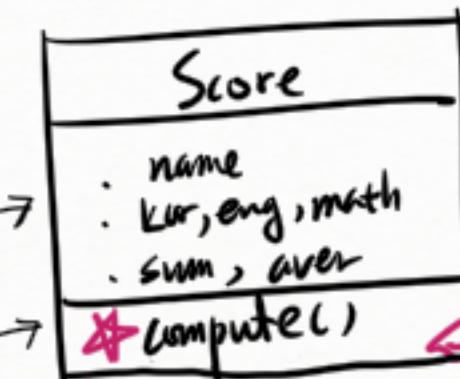
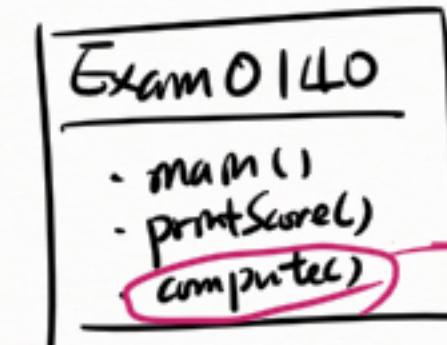
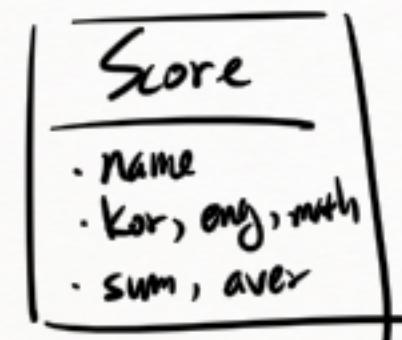
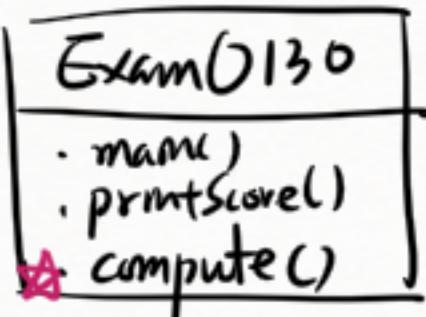
④ 리팩토링: 1기능 → 1 메서드

⑤ 리팩토링: 멤버드 이동

⑥ 인스턴스에 더 쉽게 접근하는 법: 인스턴스 메서드

~~~~~

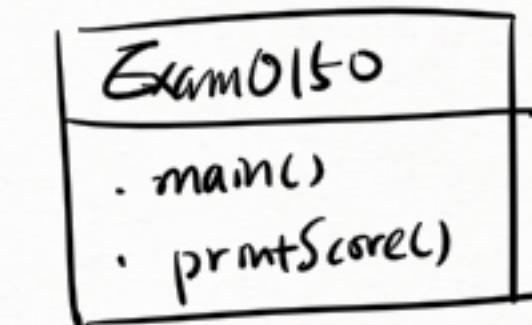
non-static



class  
속성  
↓  
data를 정의한  
2. 데이터를 다루는  
operator를 만든다.

GRASP의  
Information  
Expert

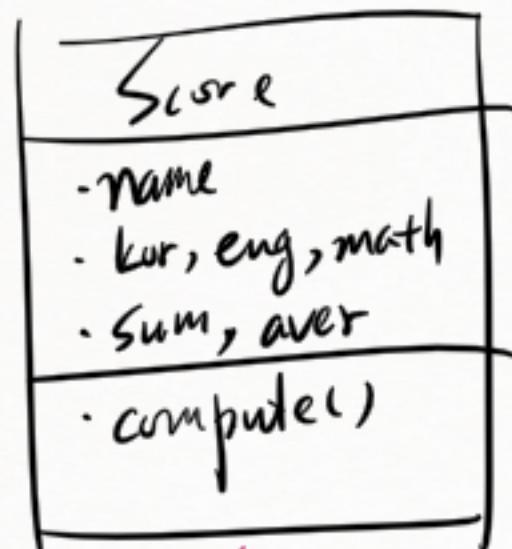
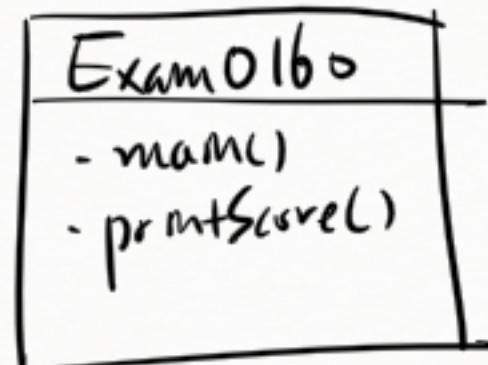
이동



non-static 으로  
변경

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

① 패키지 멤버 클래스



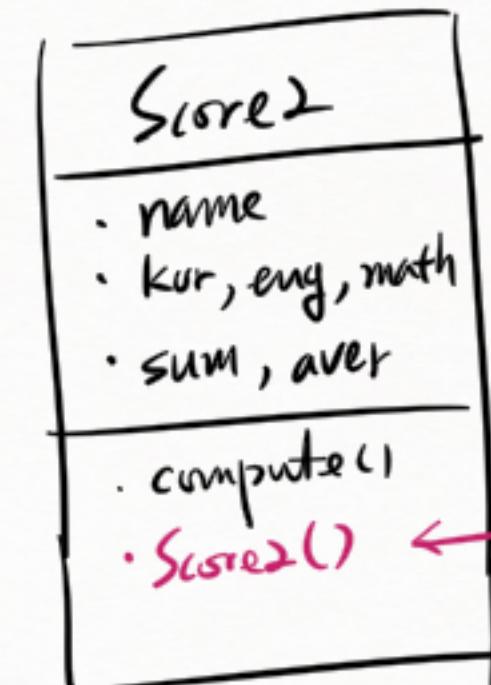
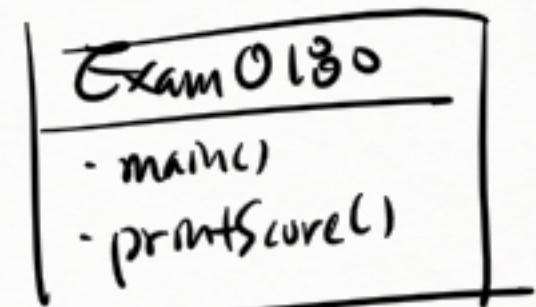
nested class → package member  
변경

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 블리어임
이로 멤버를 접근할
수가 있다.

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

* 생성자

```
class Score {  
    ↴ ← 이는 대입연산자이다  
    ← 대입연산자이다  
    → Score(π(2442, ...))  
    } =  
    }  
}
```

* 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

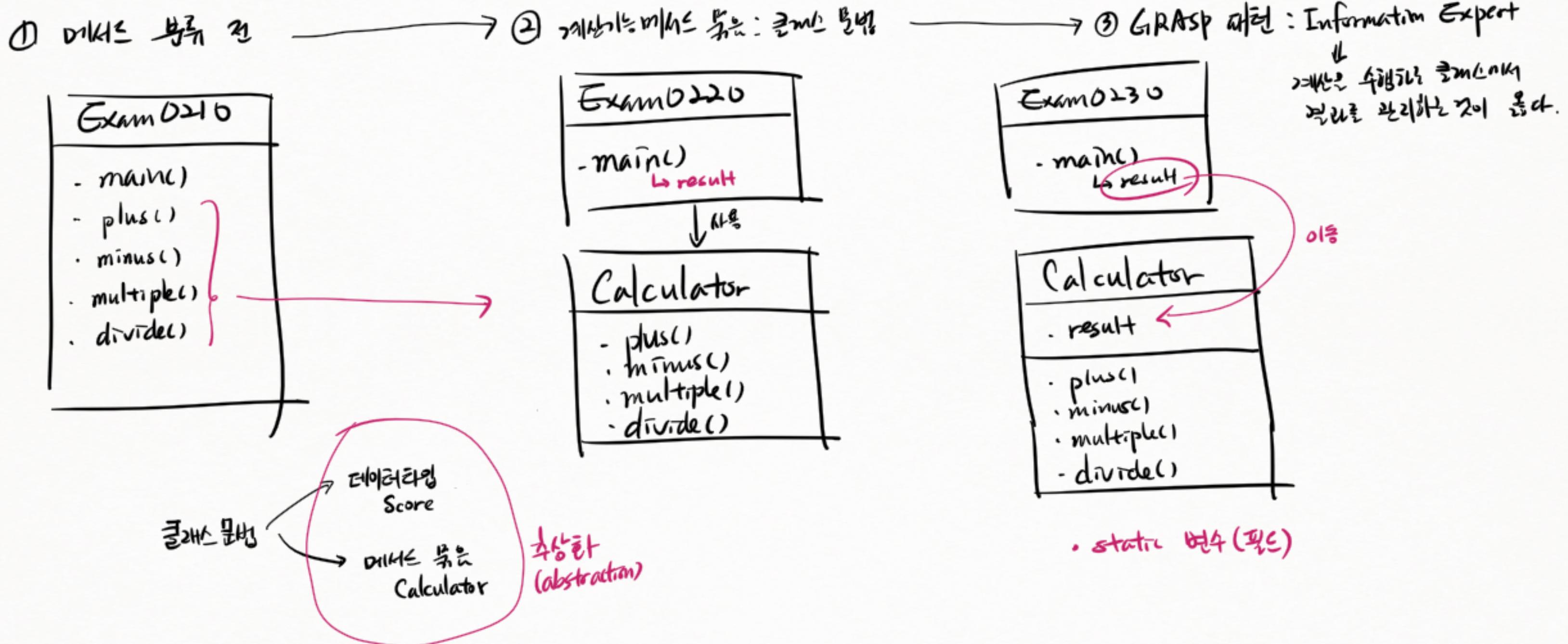
② 내용

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

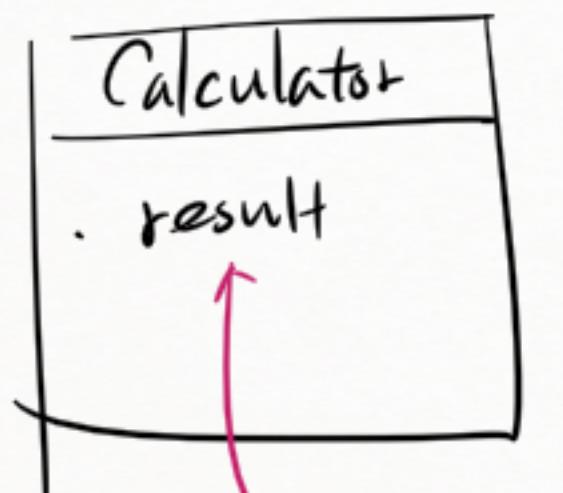
↓ 이스턴스 생성과 즉시 생성자 자동호출

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

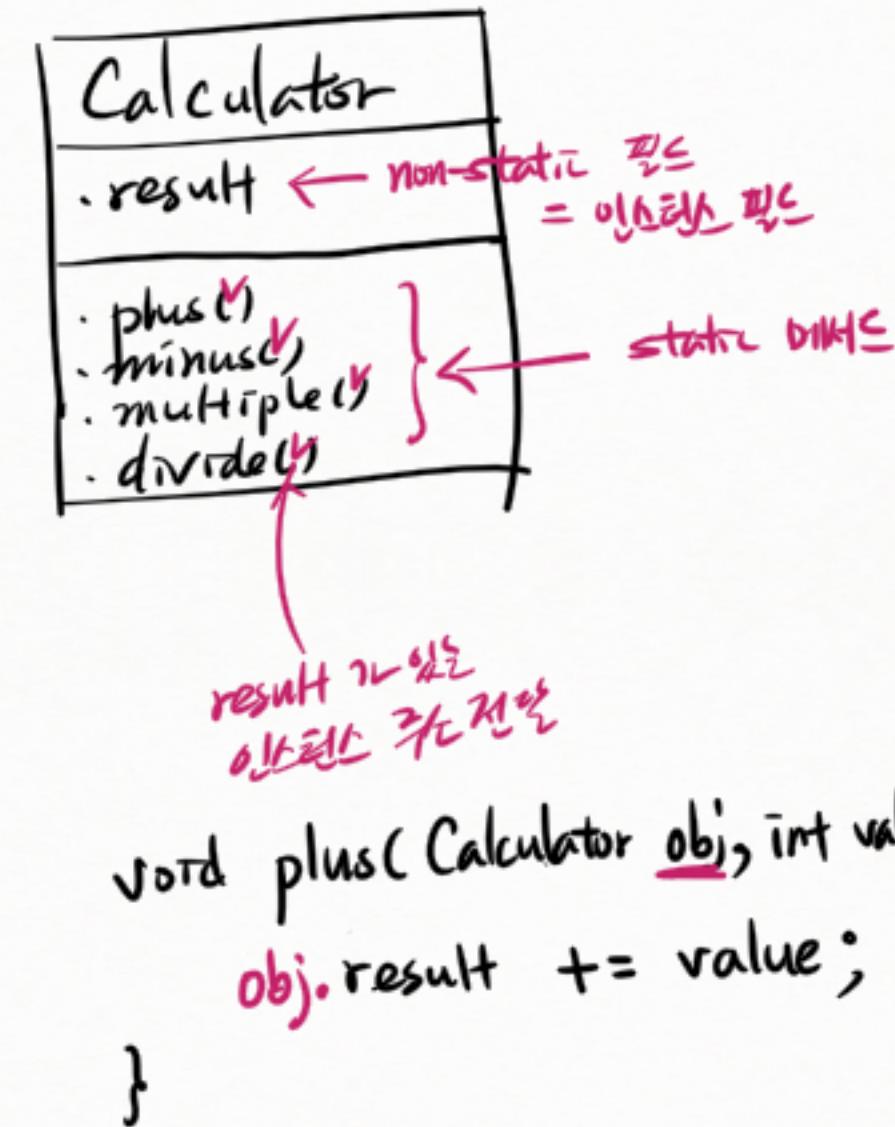
* 스태틱 필드 → 인스턴스 필드
 (com.eomcs.oop.p. ex02.Exam02xx)



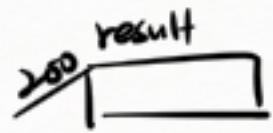
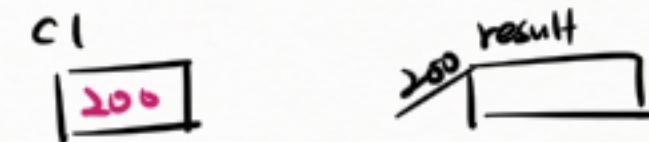
④ \Rightarrow 멤버 변수의 초기화 \longrightarrow ⑤ 인스턴스 변수로 전환



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



Calculator c1 = new Calculator();



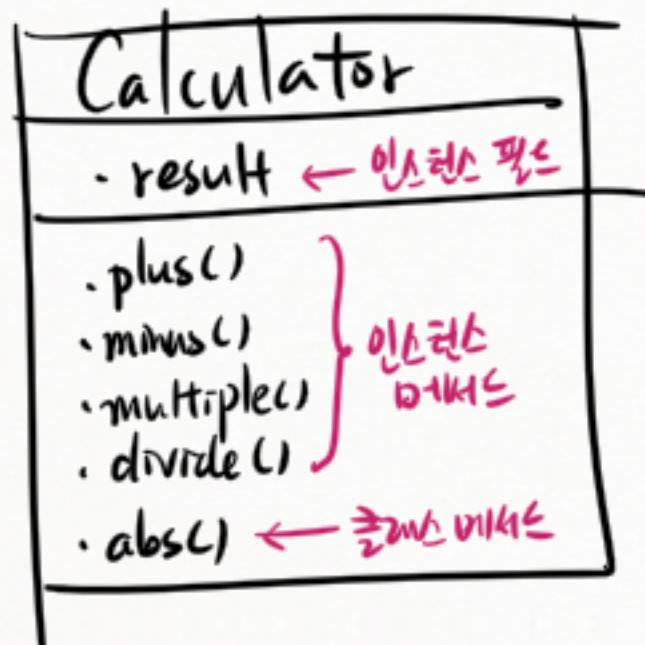
Calculator c2 = new Calculator();



Calculator.plus(c1, 2);

Calculator.plus(c2, 3);

⑥ static 멤버드 → 인스턴스 멤버드 정의



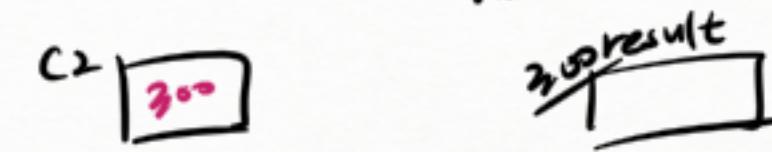
```
void plus( int value ) {  
    this.result += value;  
}
```

↑ 인스턴스 멤버드의 Built-in 변수

Calculator c1 = new Calculator();



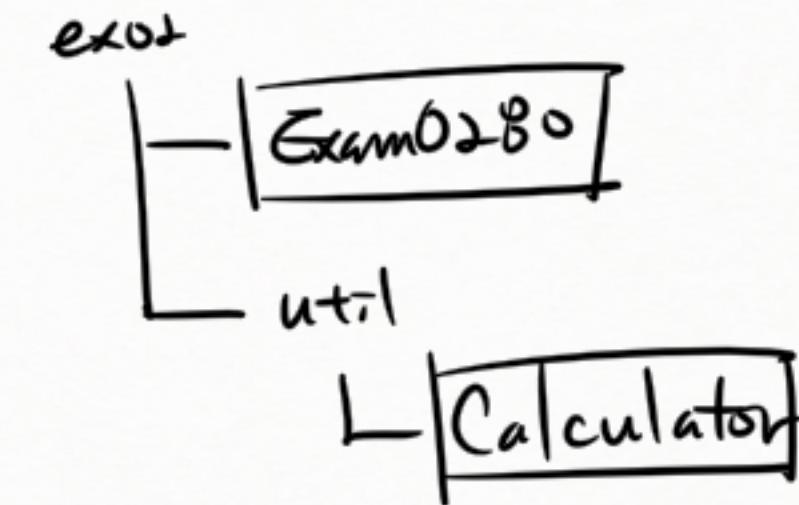
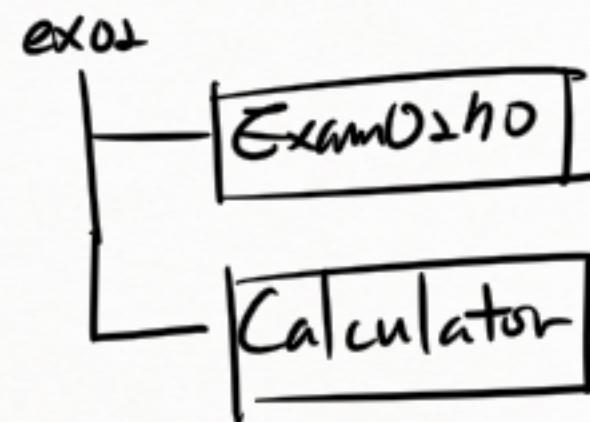
Calculator c2 = new Calculator();



c1.plus(2);

c2.plus(3);

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

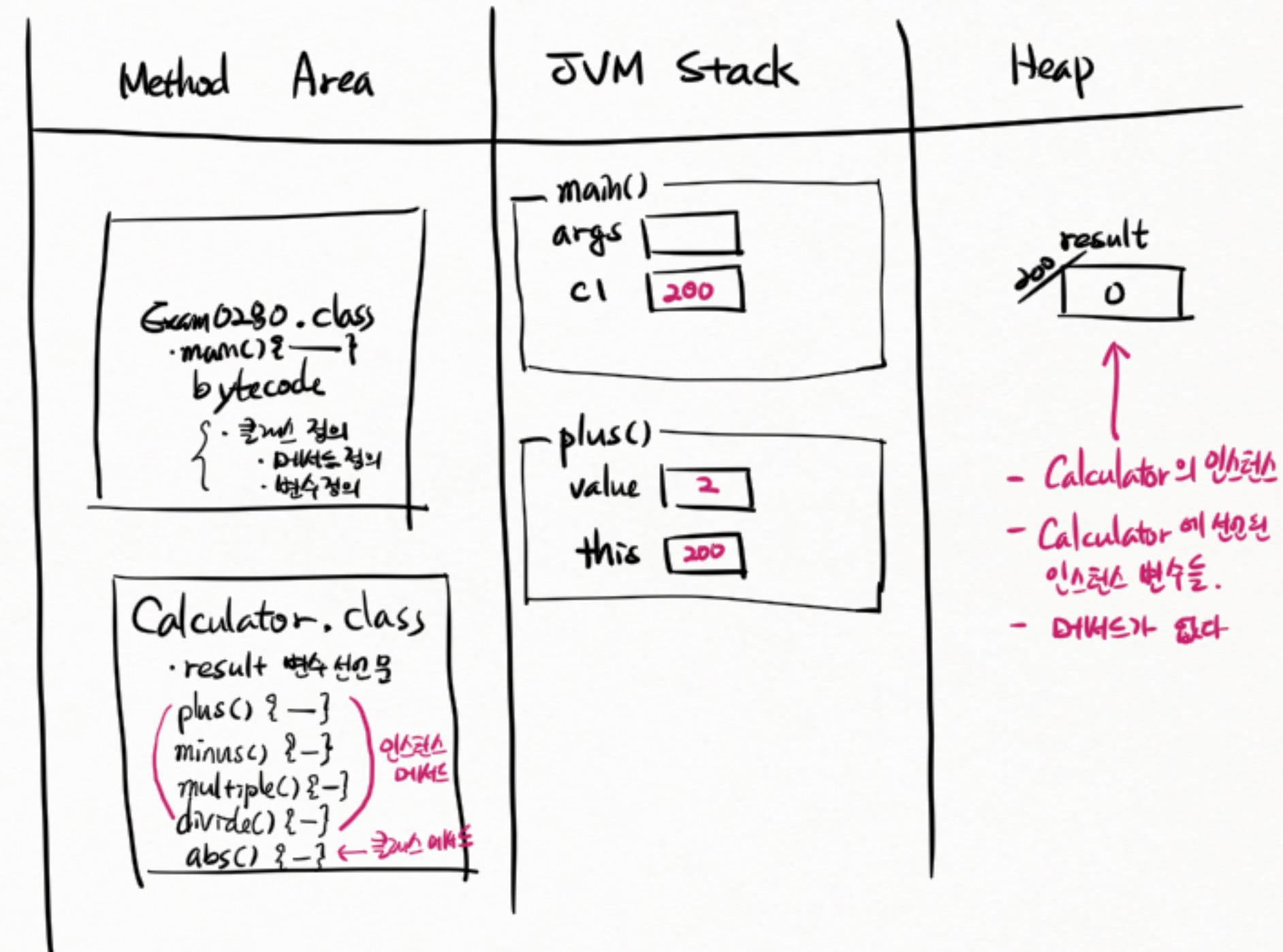


* JVM 디버깅 예제와 변수, 메서드

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

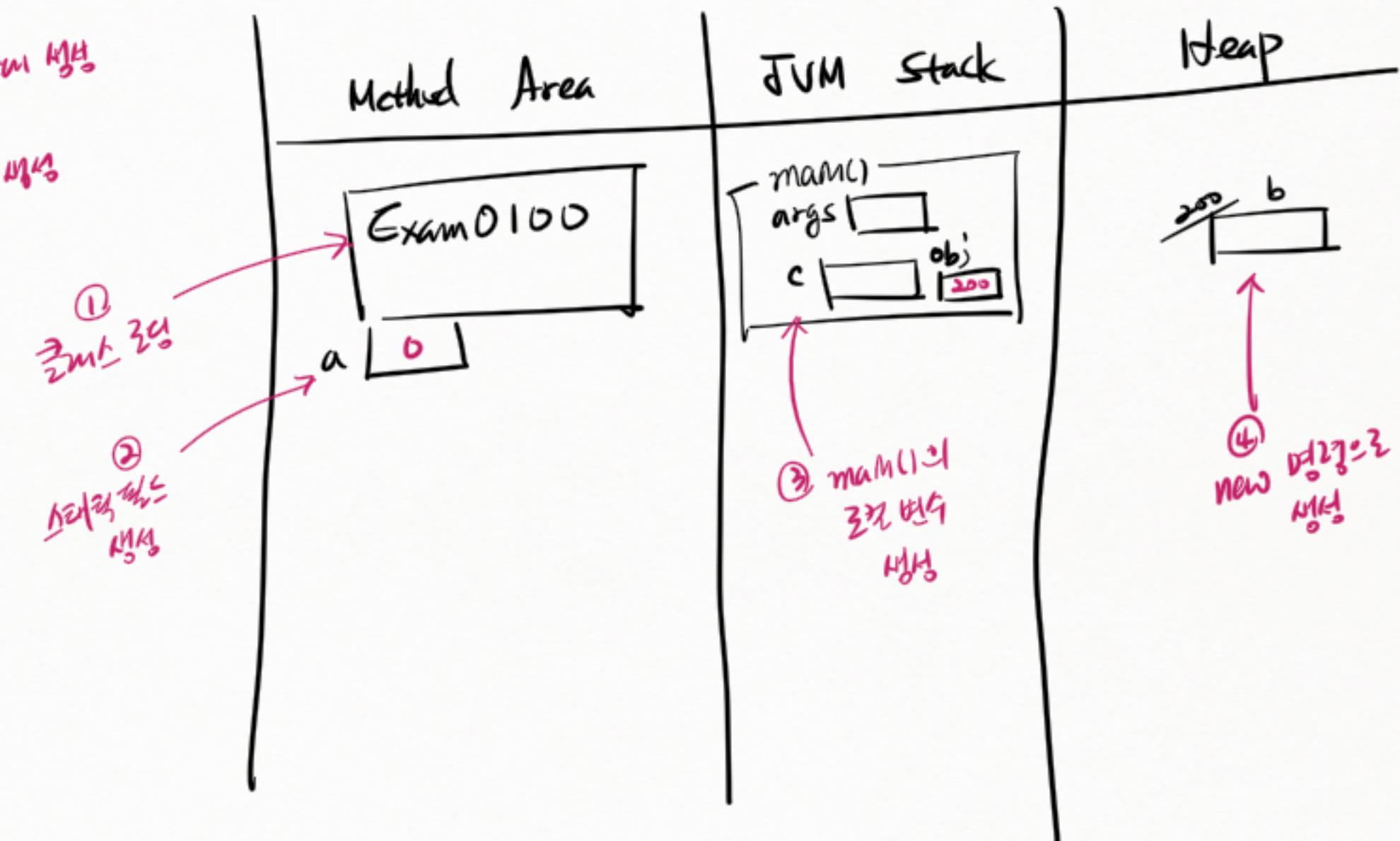
인스턴스 주소

Calculator에 선언된 인스턴스 변수를 Heap에 생성한다

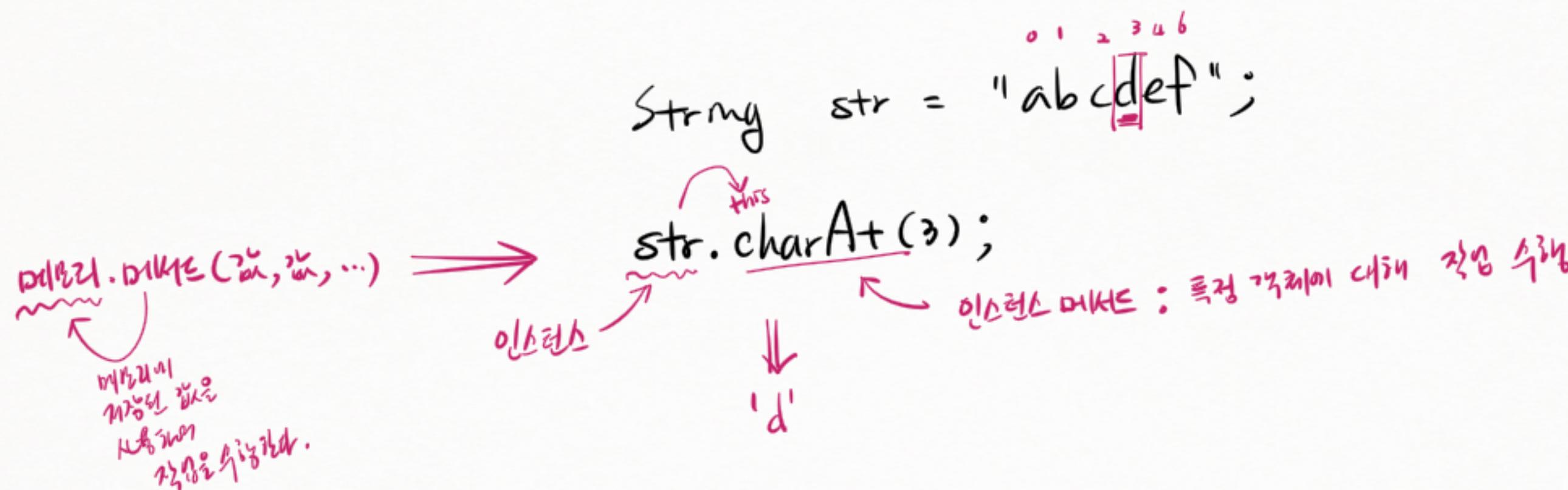
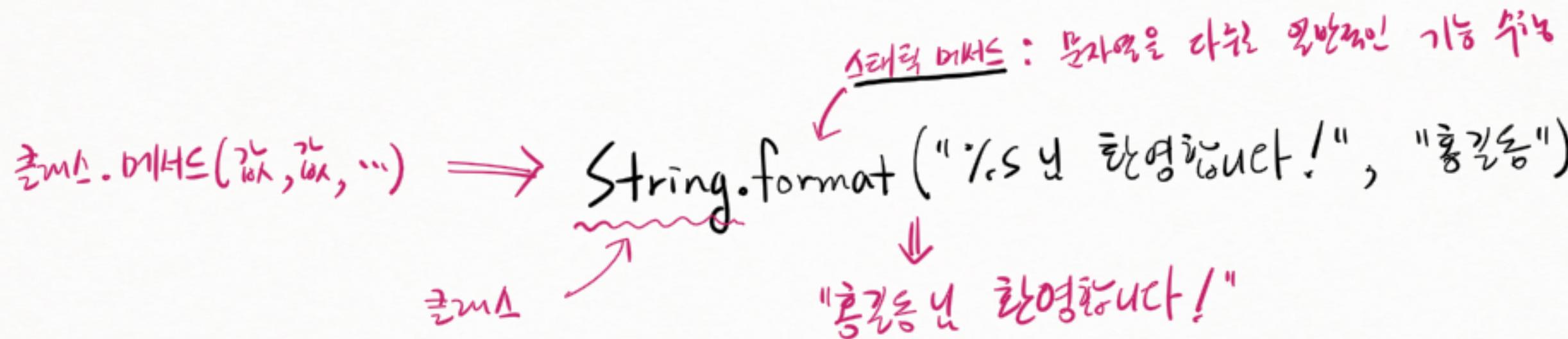


* 스레티 필드, 인스턴스 필드, 로그 변수

```
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);

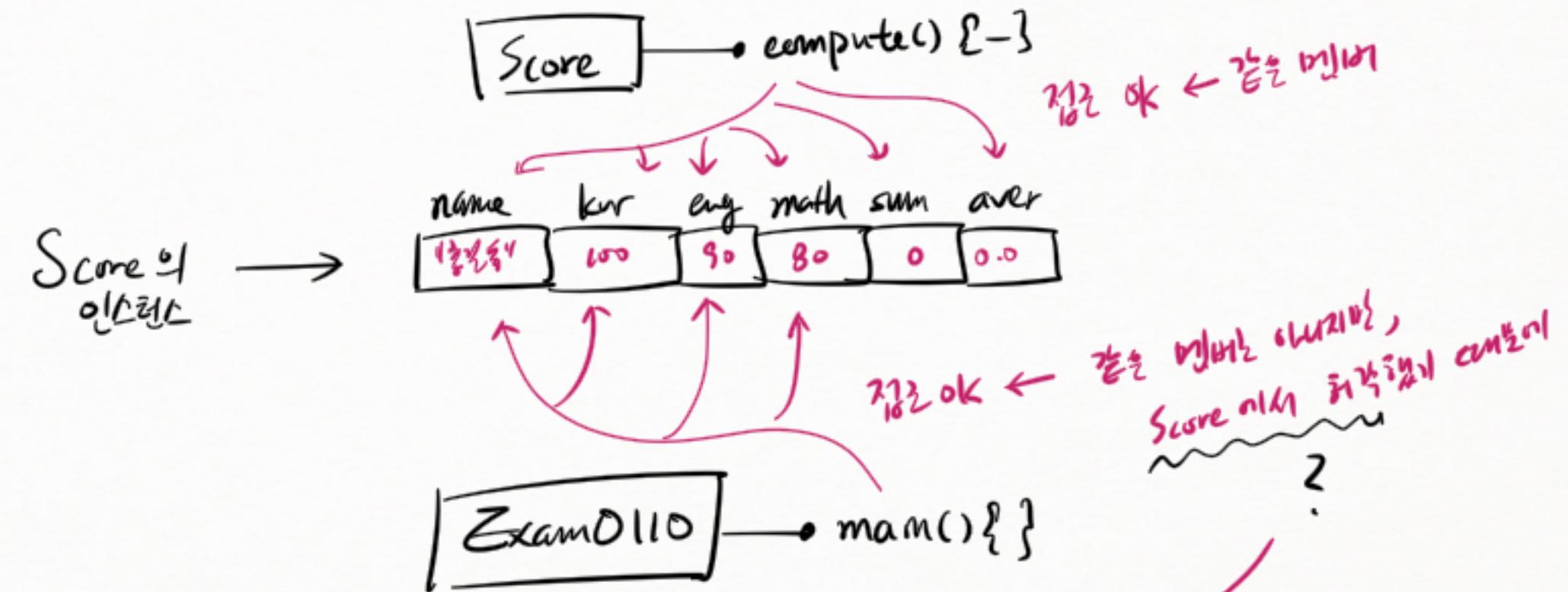
← 오류

생성자에 정의된 매개변수와 같은 이름의
기타 멤버 변수가 있는 경우
생성자에 정의된 매개변수를 사용

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 클래스내

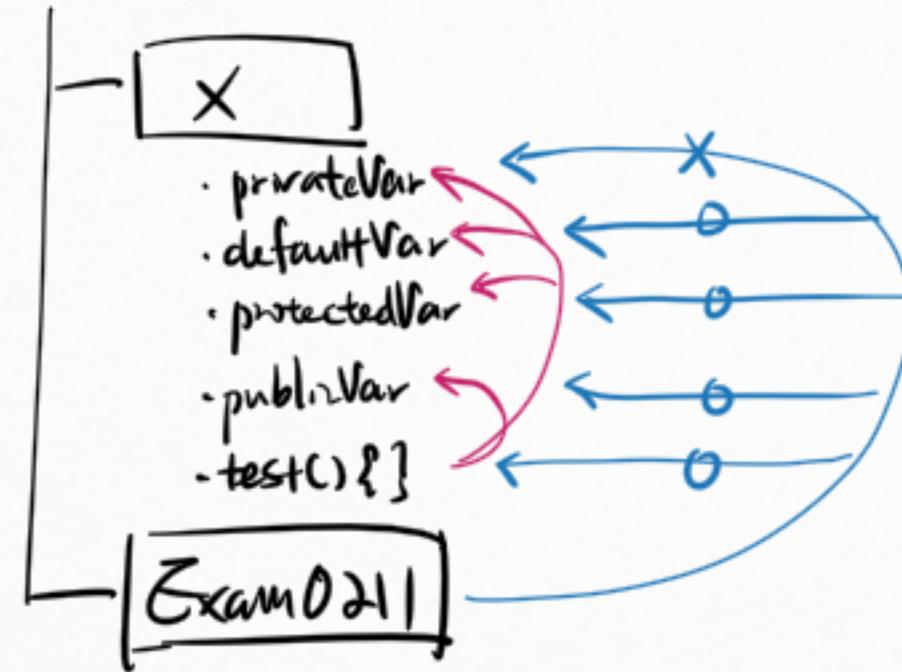
- (default): " + 같은 패키지

- protected: " + " + 서브클래스의 멤버

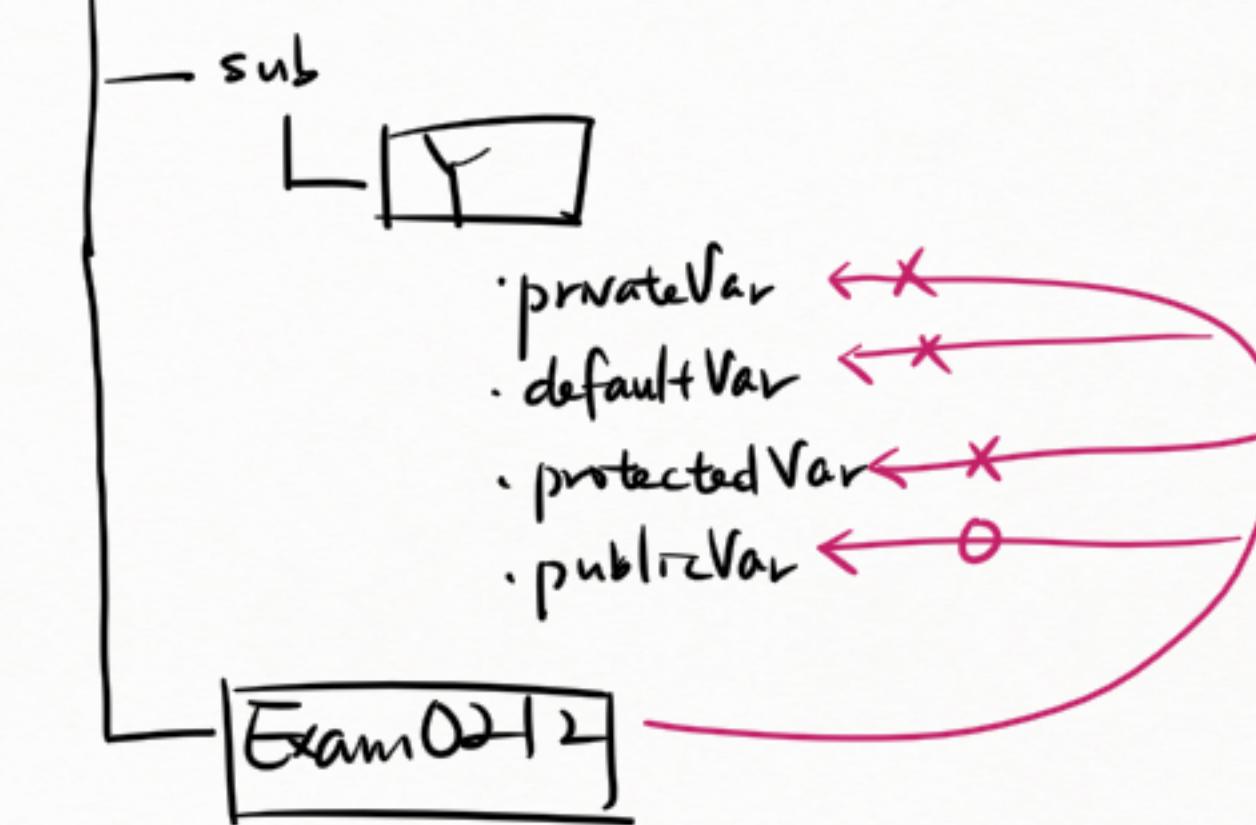
- public: 모두

* 2장 멤버 접근 제한

ex08

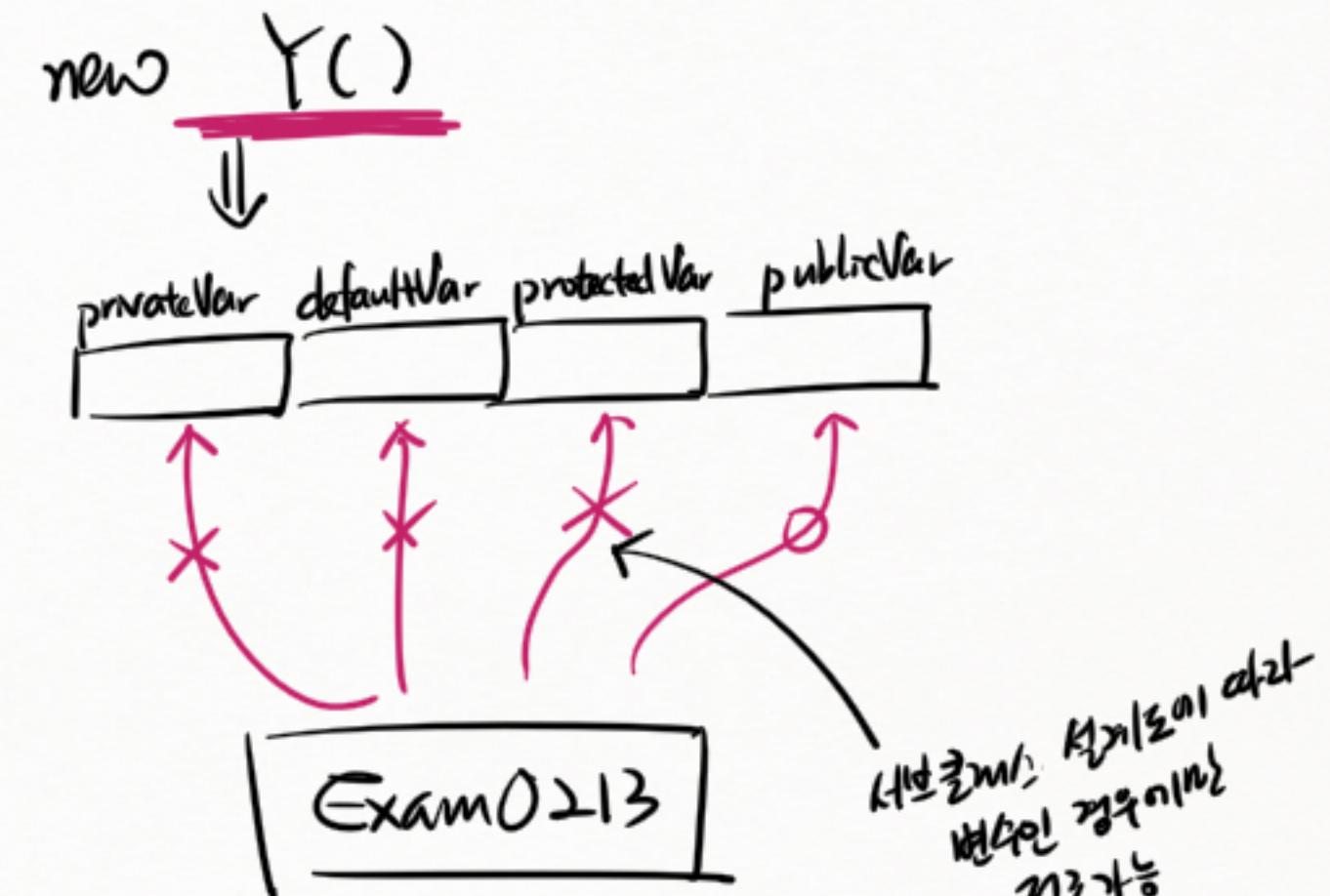
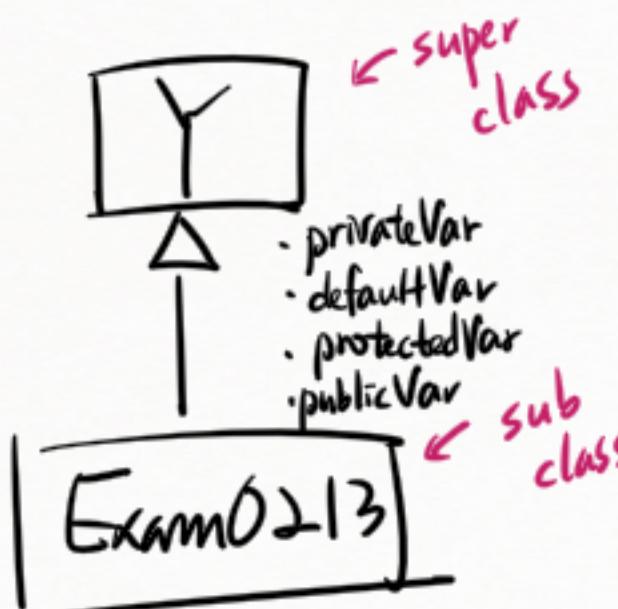
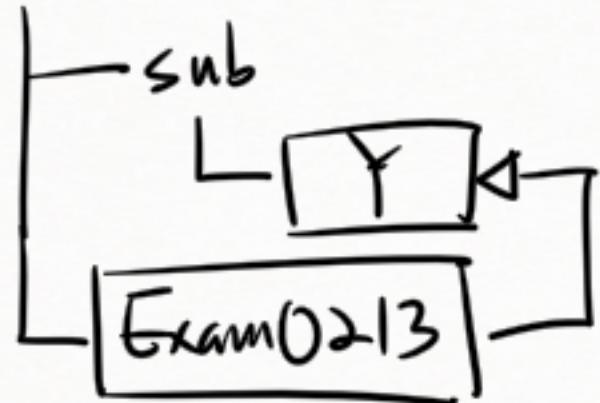


ex08

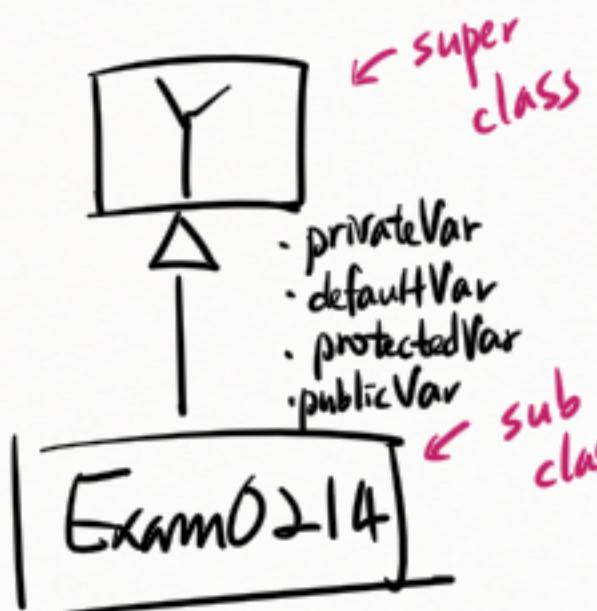
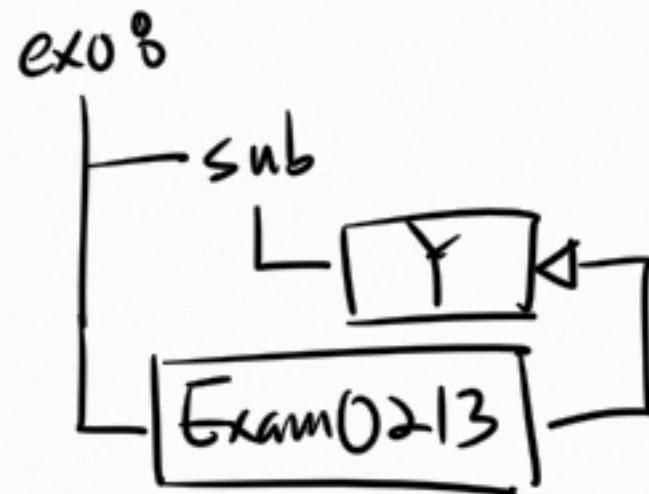


* 접근 범위 초기화 : protected 초기화

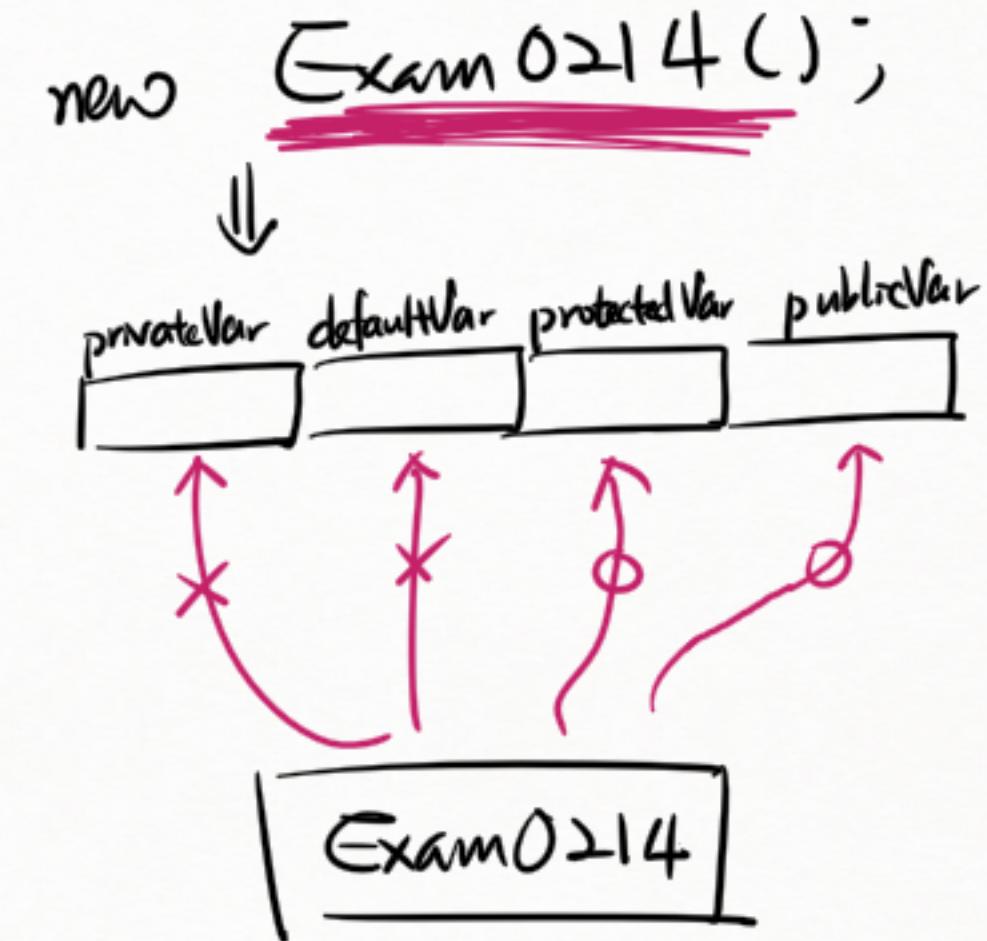
ex08



* 접근 범위 제한 : protected 범위 II



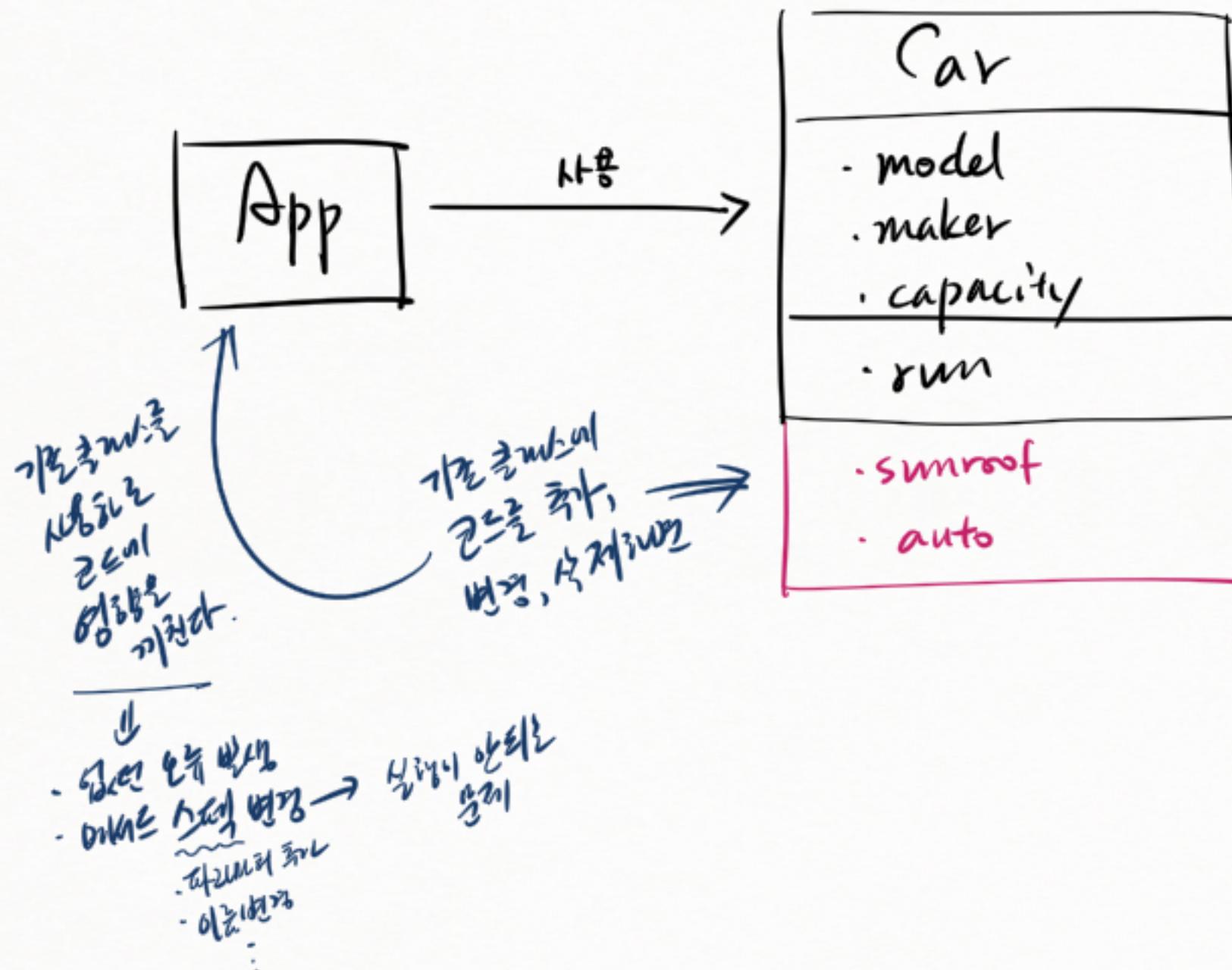
* 접근 ?
- 키워드로



상속 (Inheritance)

* 기능 학장 — ① 기존 클래스를 변경

① 2학기 A

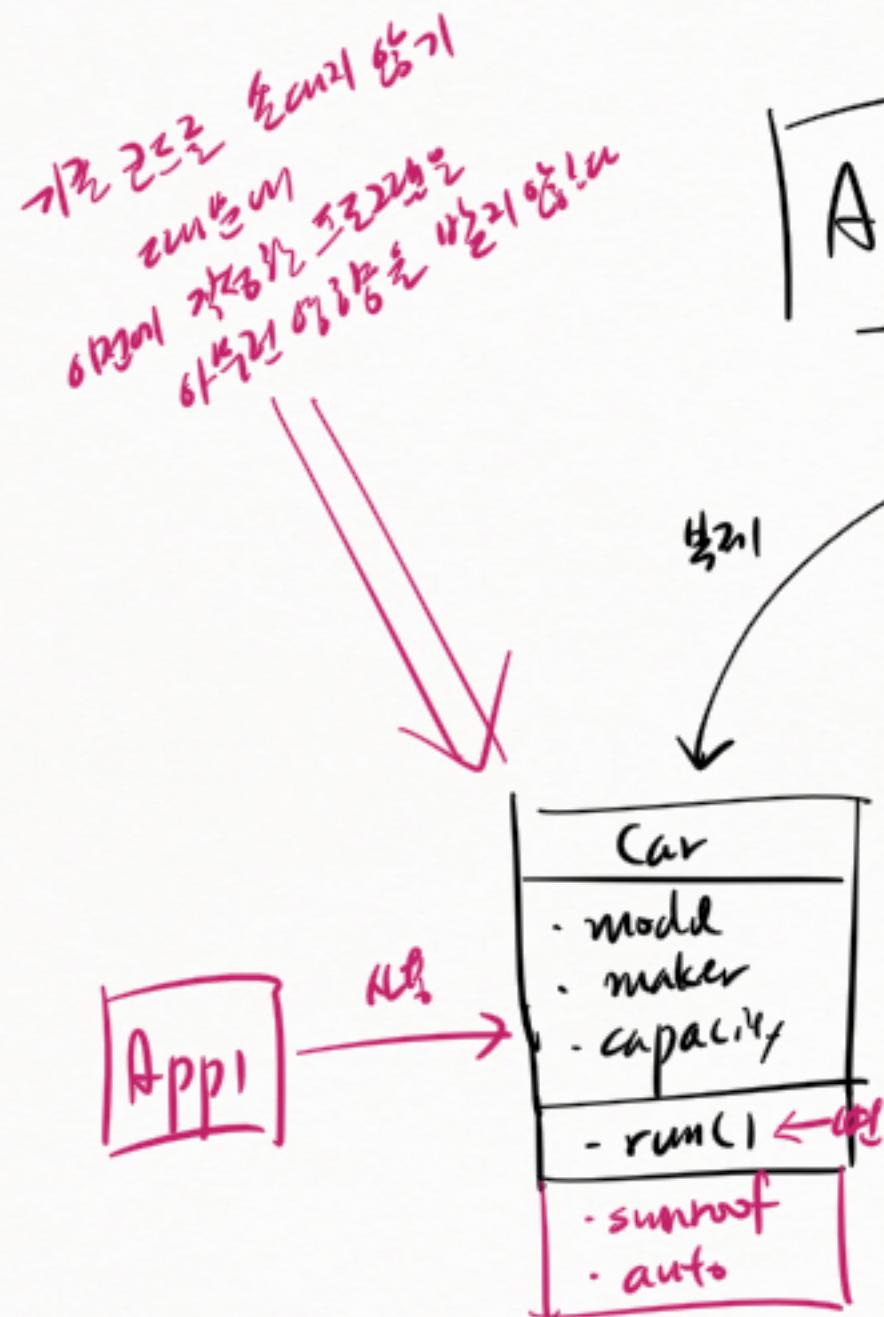


② 고급사 B : 기능 추가 - a/b

↓
기존 큐레이션에 헌드 추가

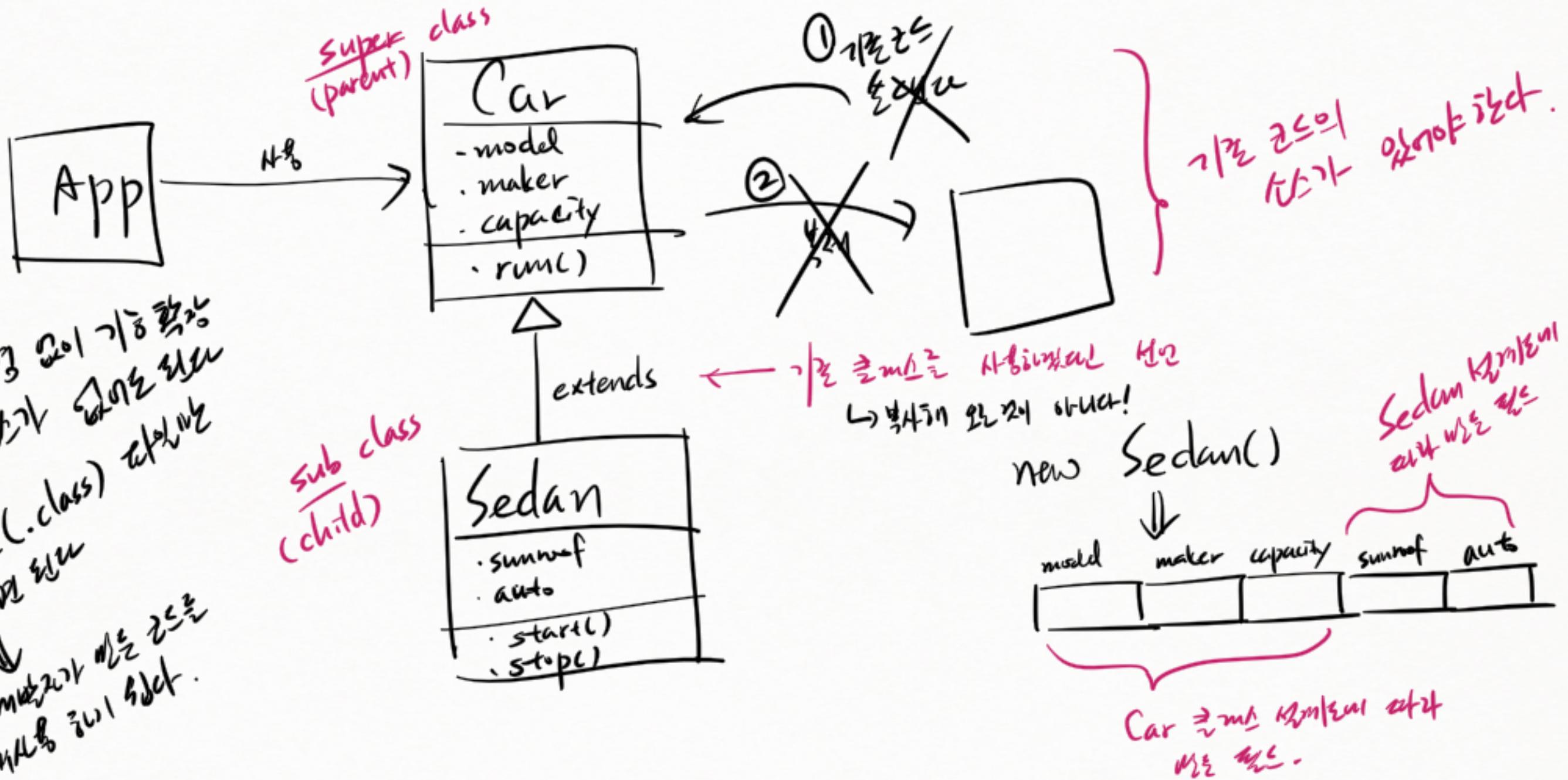
- * 7월 2일
- 기존 커스를 사용하는
1) 기존 커스를 사용하는 커스
2) 커스에 영향을 미친다.

* 가능 학습 - ② 기존 코드를 복제한 후 가능 추가

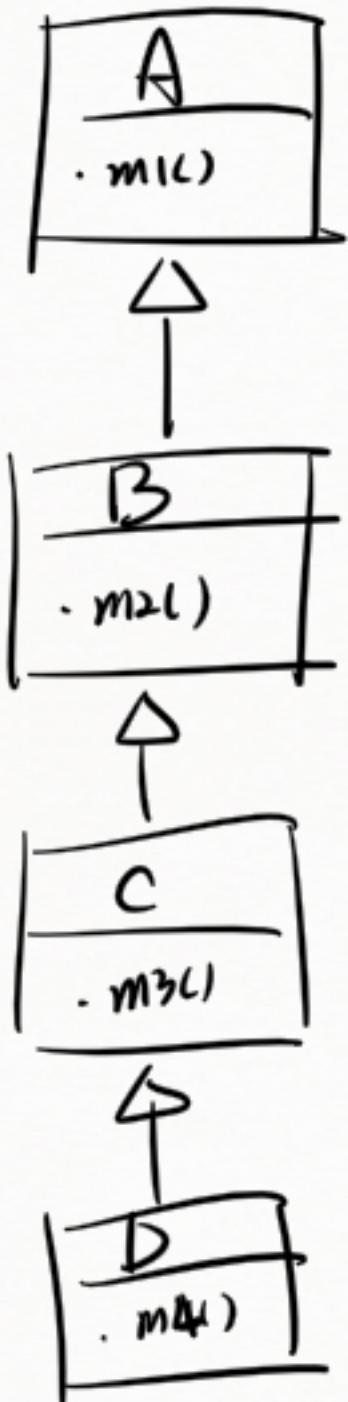


* 가능성이 있는 - ③ 상속을 이용한 가능성을

- * 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();

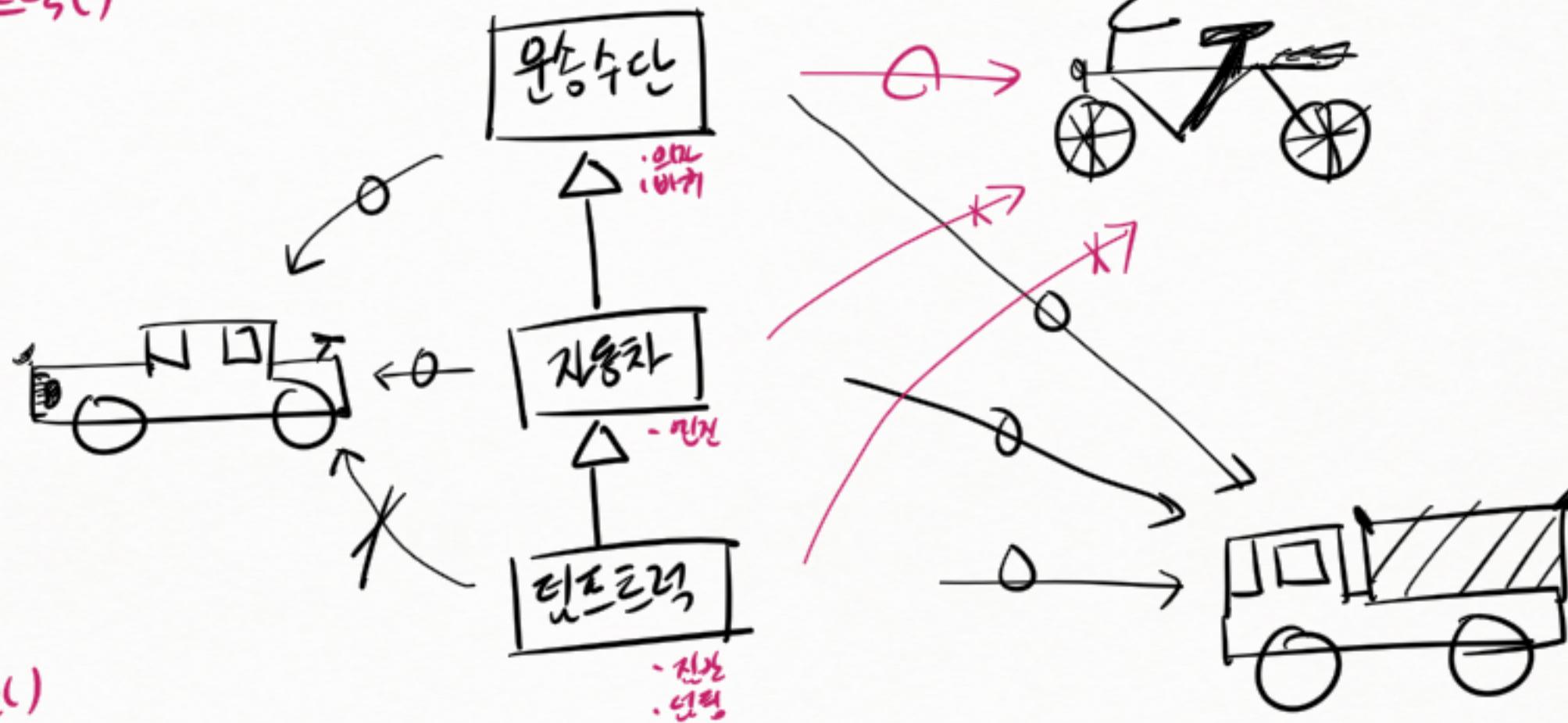
(obj. m4();
 obj. m3();)
 ↗
 컨타워려는 변수(리퍼런스)의
 대입에서 메서드를 찾아 올라온다.
 리퍼런스가 실제 어떤 클래스의
 인스턴스를 가리키는 것인지
 따지지 않는다.

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

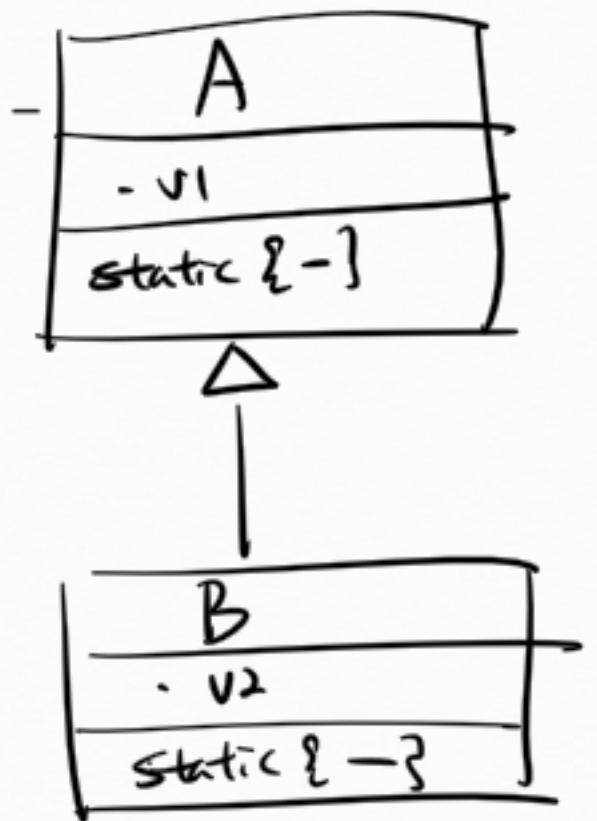
터프트럭 r1 =
기동차 r2 = new 터프트럭()
운송수단 r3 =

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

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

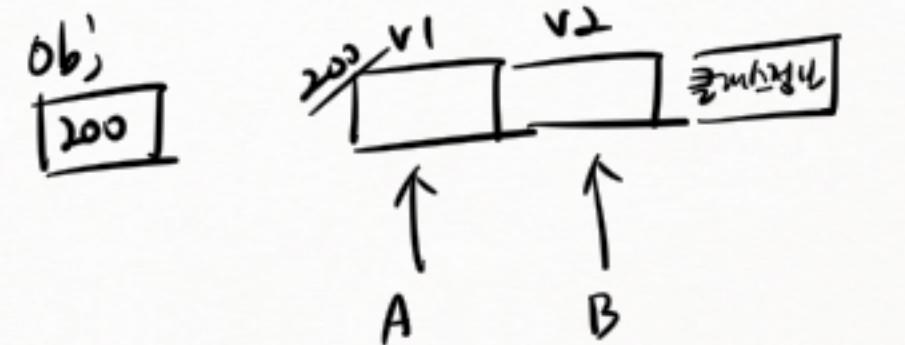


* 상속과 상속 관계

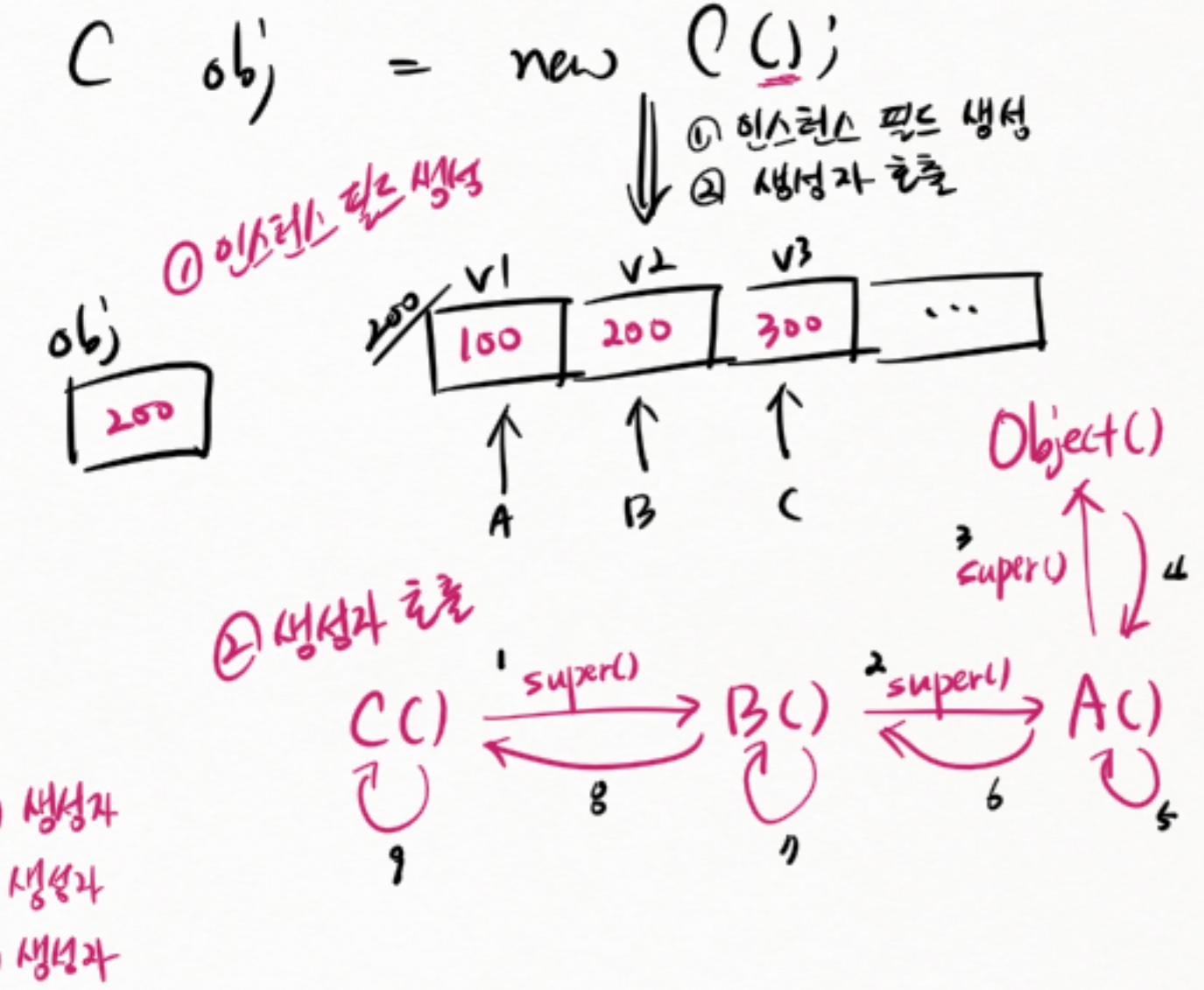
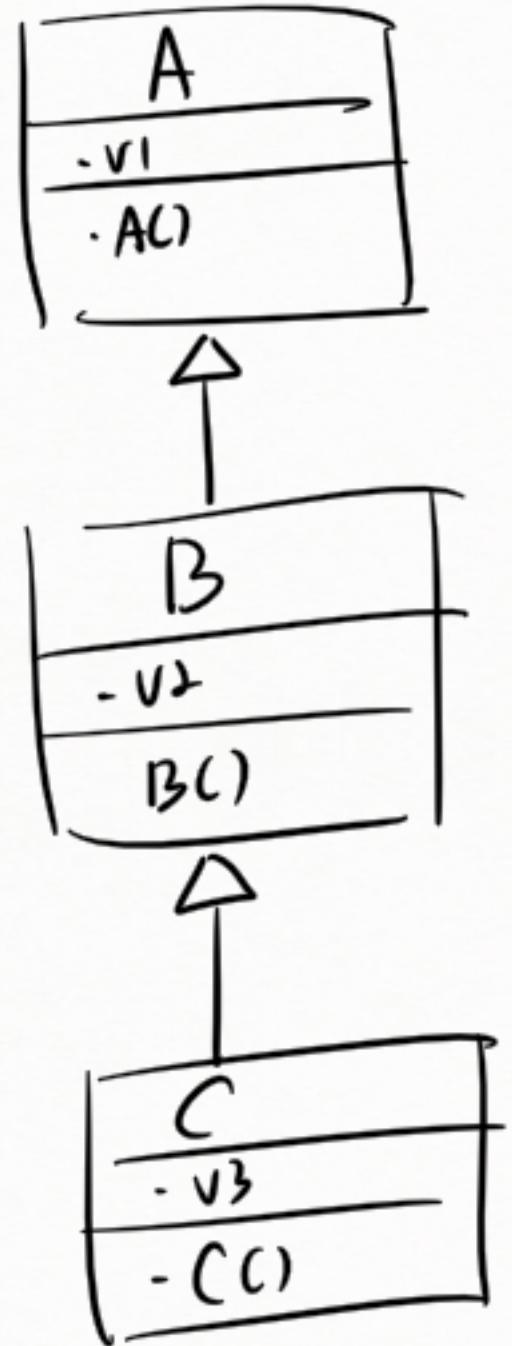


B obj = new B()

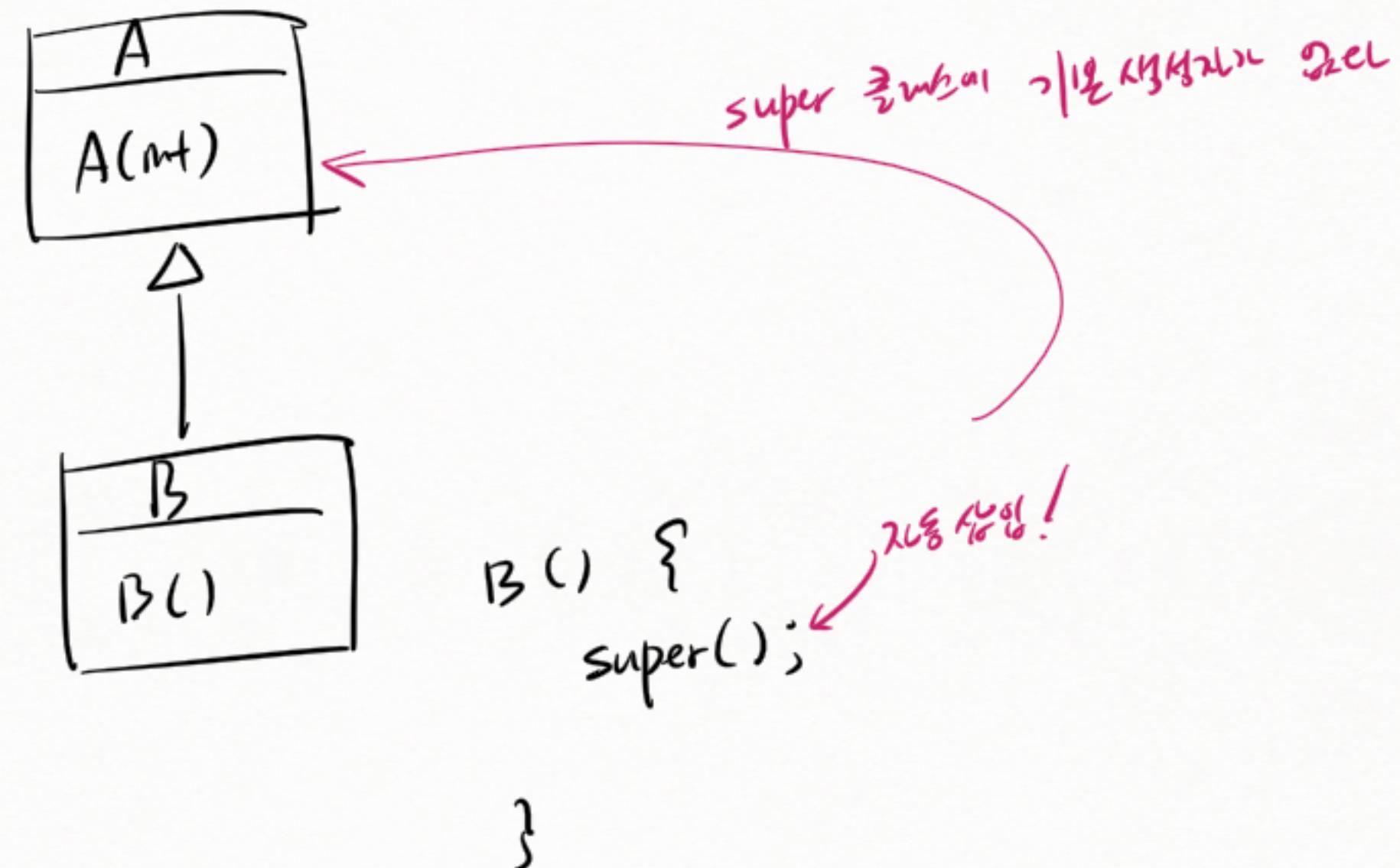
- ① 수퍼클래스부에 초기화
- ② 수퍼클래스부에 인스턴스 필드 생성



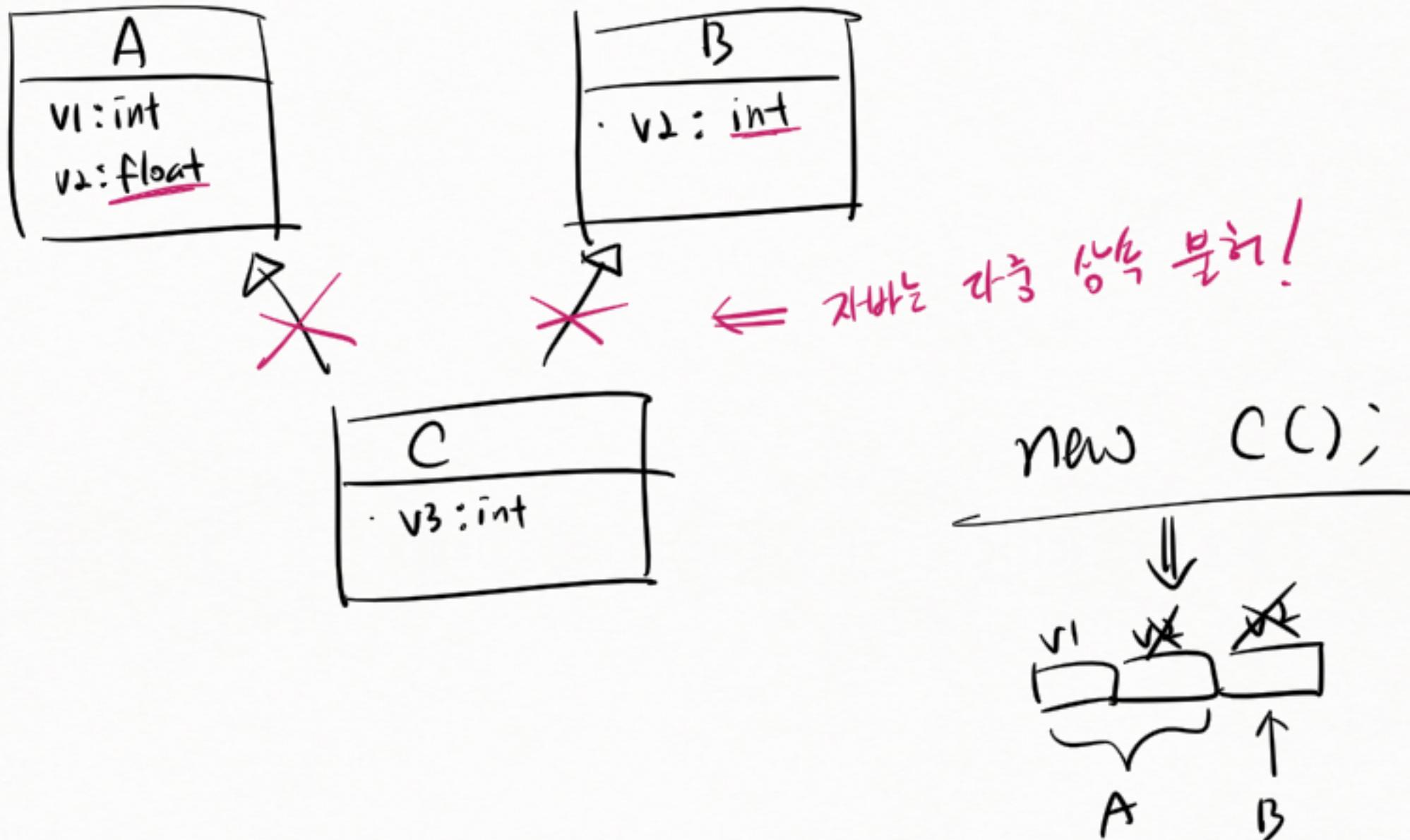
+ 생성자 호출 순서



* 초기 층 클래스의 생성자 호출

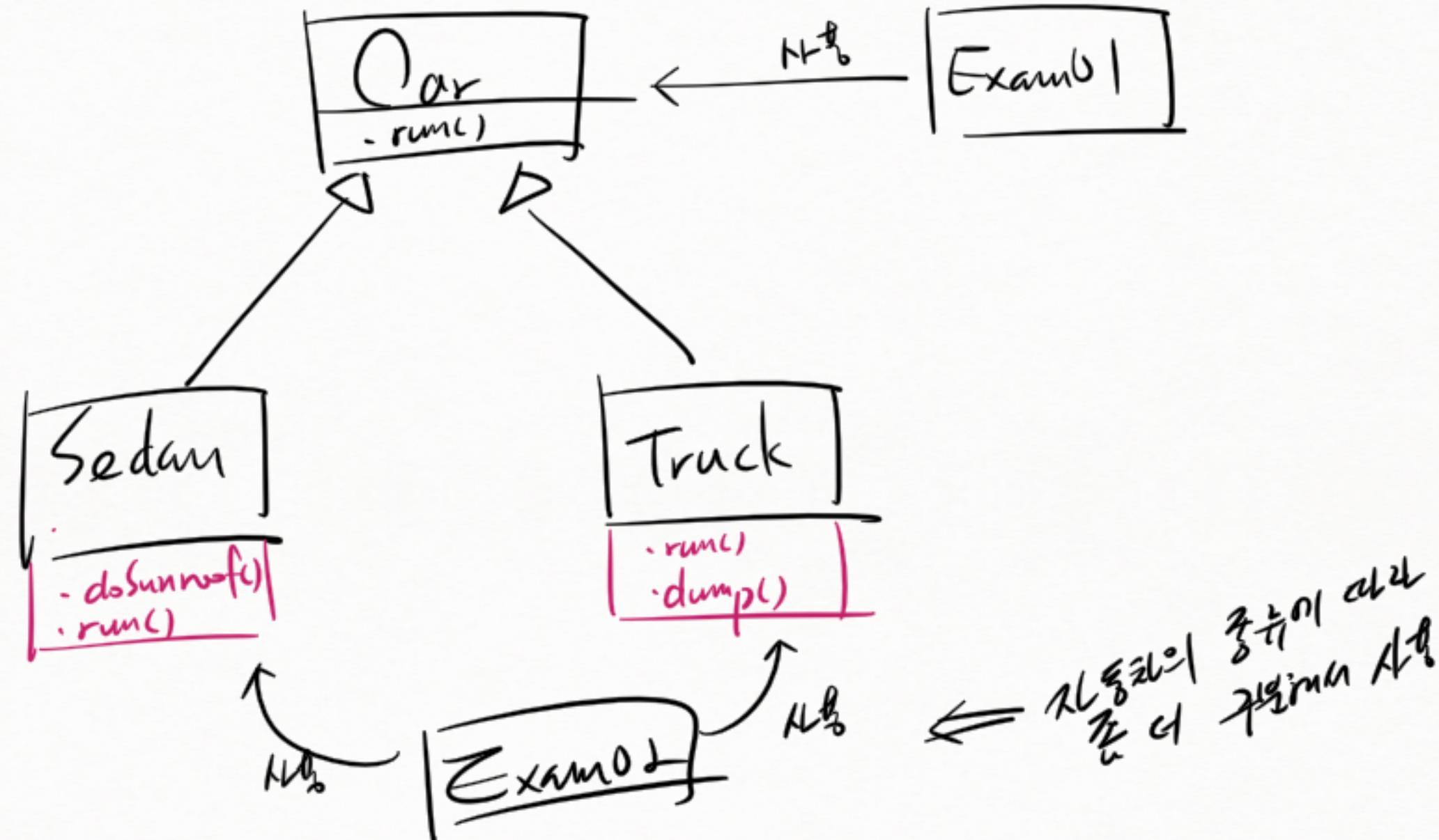


* 다음 상속?

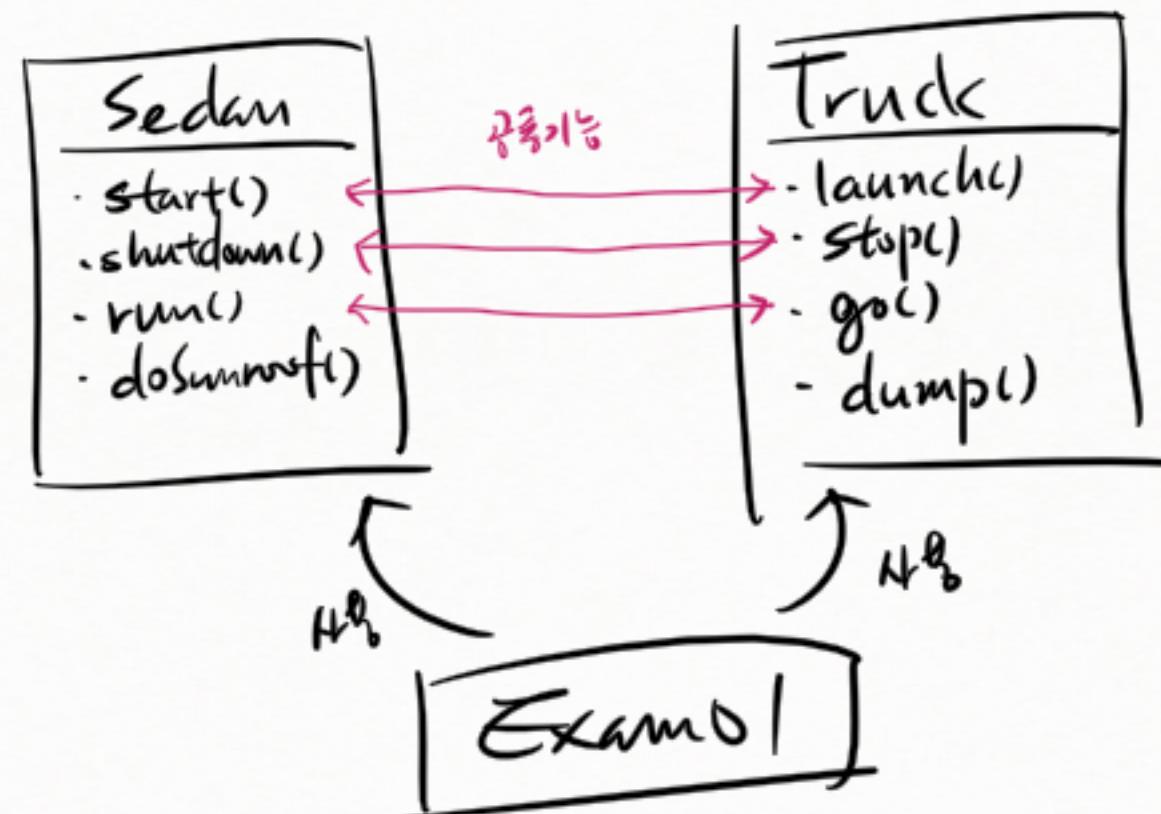


* Specialization

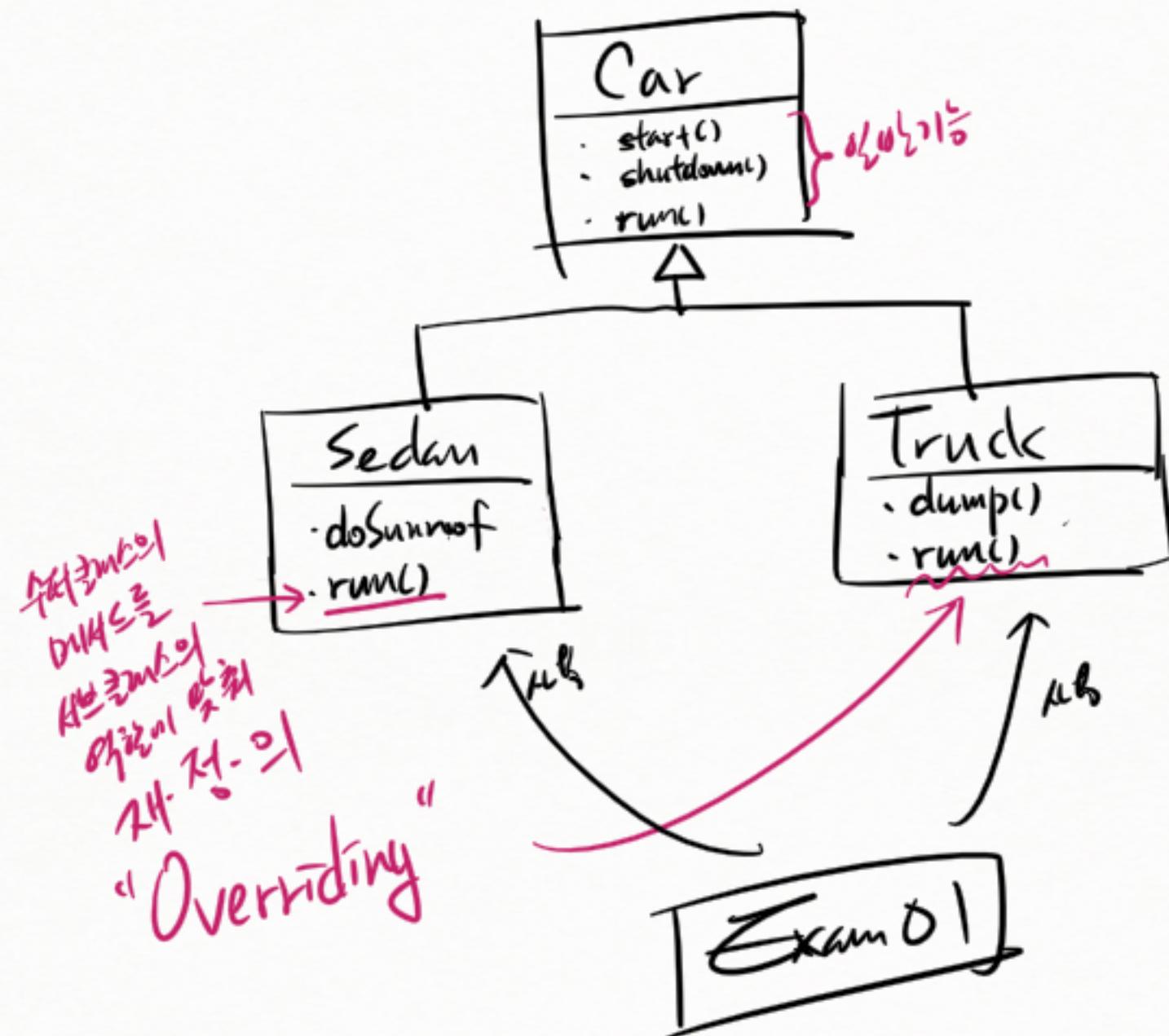
↑
↑
↓
↓
↓



* generalizat.m

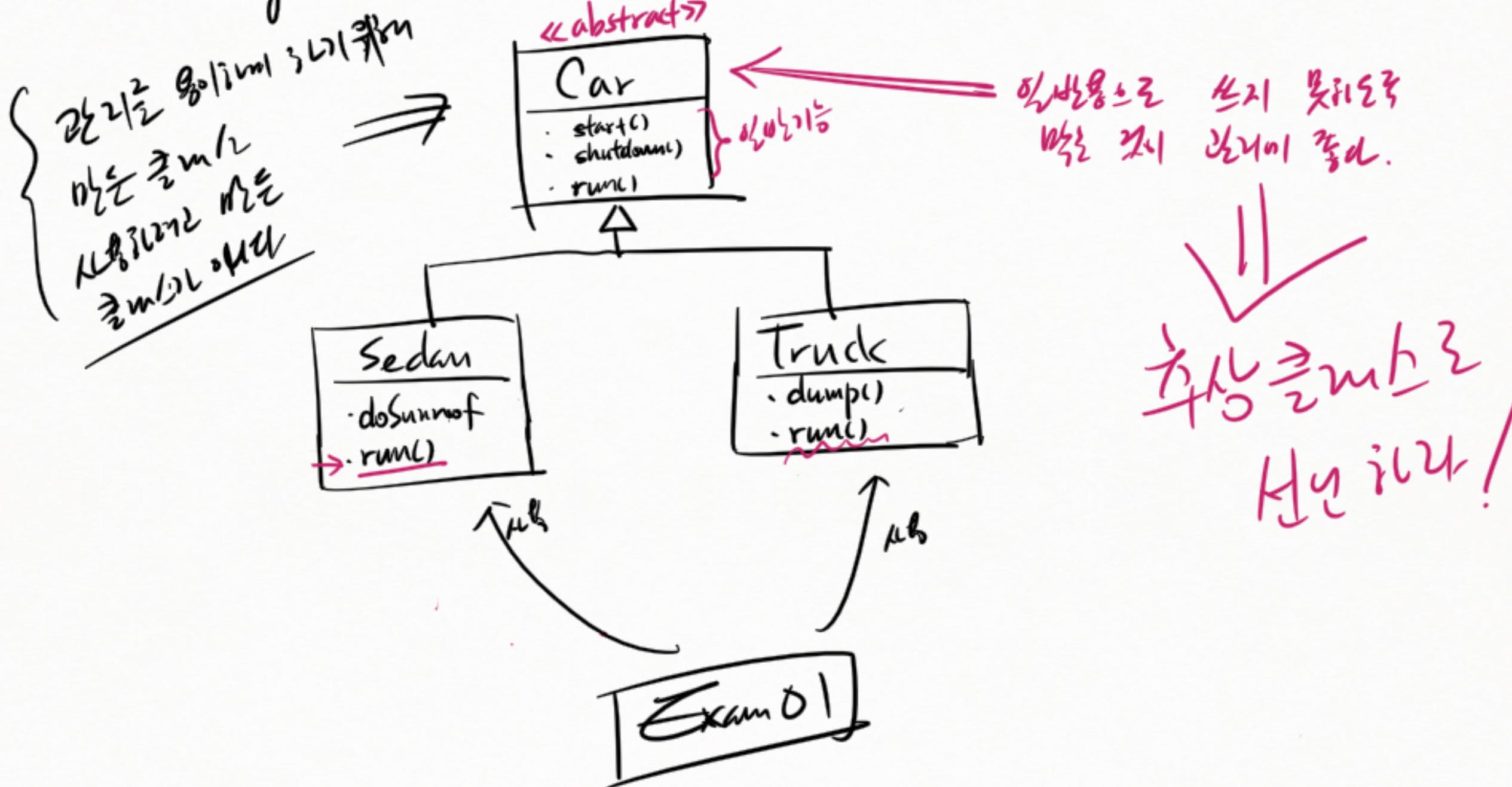


* generalization

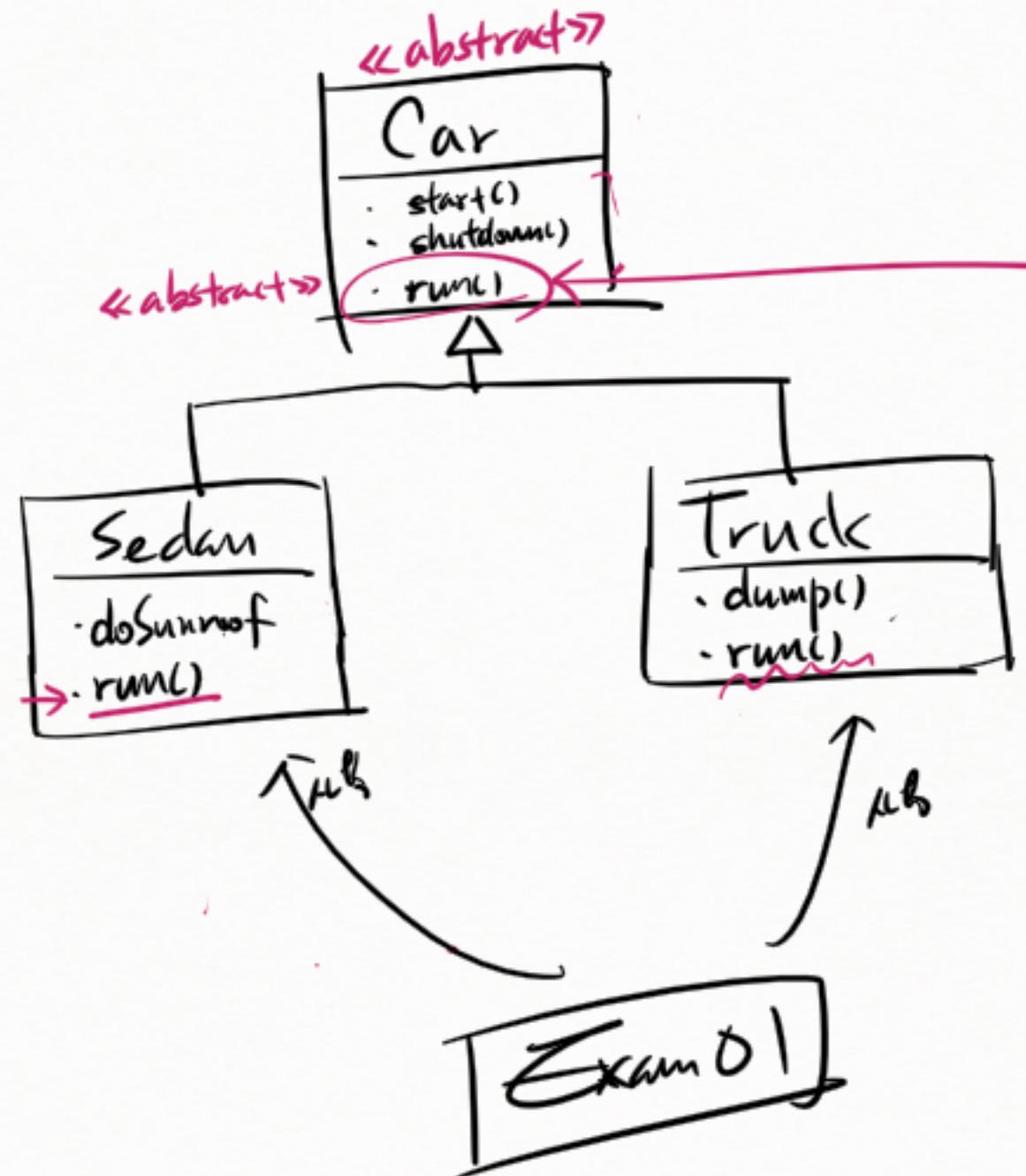


↑
부모 클래스의
공통 속성 / Method
추상화
부모 클래스
부모 메소드 상속
Overriding!
"generalization"

* generalization : 추상 클래스



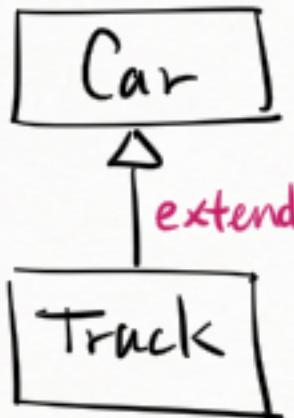
* generalization : 추상 클래스



Abstraction
Generalization
Specialization
Generalization
Abstraction
Generalization
Generalization
"Abstraction"

* 여기서 잊기 쉬운 것들 - 클래스 간의 관계와 UML

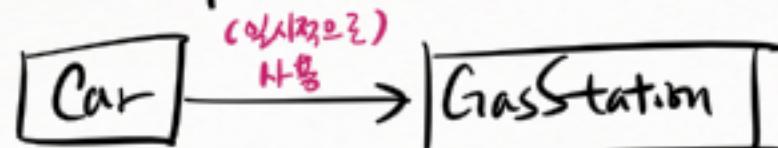
① 상속 (inheritance)



class Track
extends Car {



⑤ 의존 (dependency) : 특정 객체에서 의존으로 사용



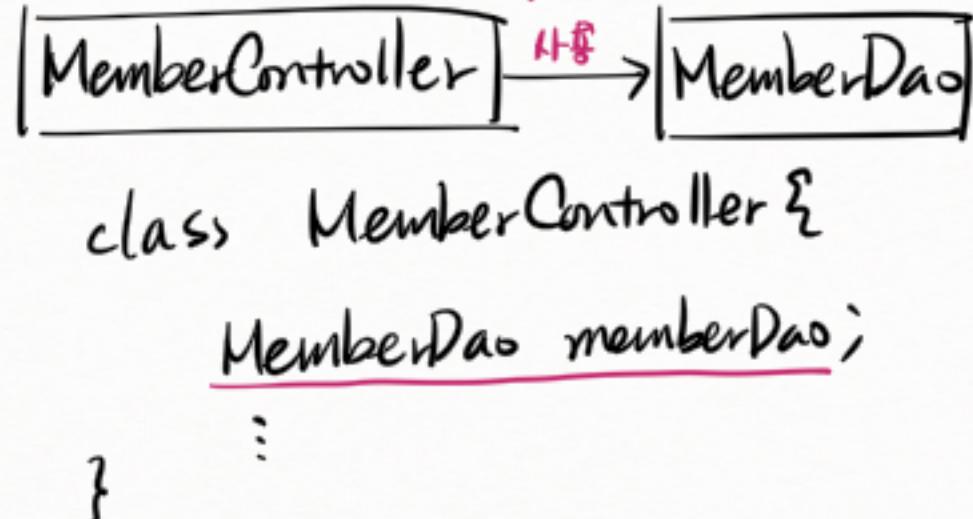
class Car {
void oil(GasStation statin){

Container
lifecycle
||
Containee
lifecycle

(자식제작)

사용

② 연관 (association)

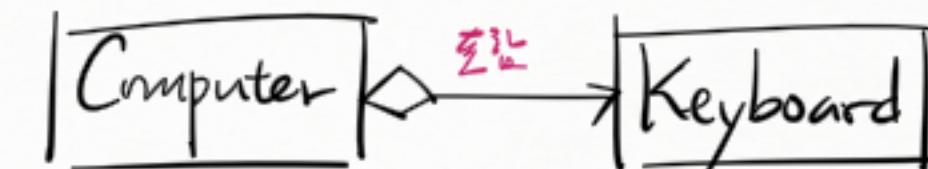


사용

제작

제작

③ 집합 (aggregation) 약관

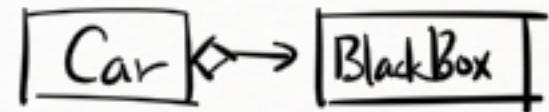


포함

class Computer {

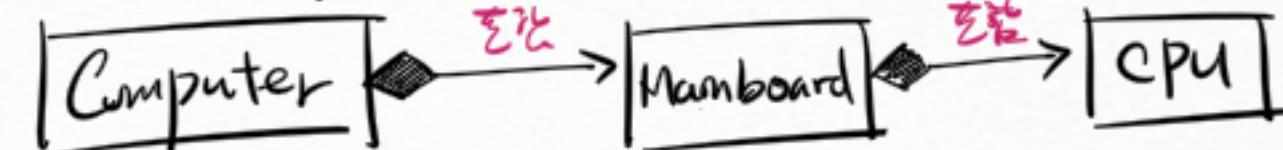
Keyboard keyboard;

:



Container ≠ Containee
Lifecycle Lifecycle

④ 핍합 (composition) 강관



포함



포함

class Computer {
Mainboard mb;

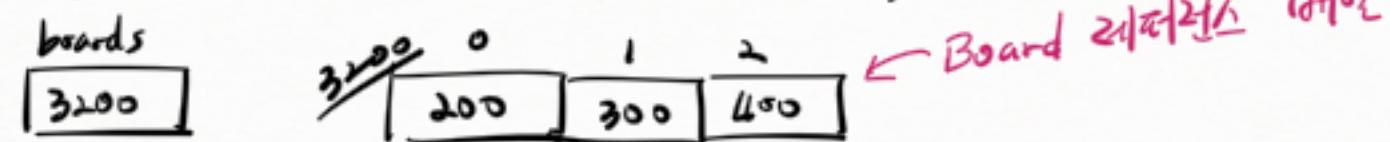
class Mainboard {
CPU cpn;

:

:

* 21/21/21 - 11:10

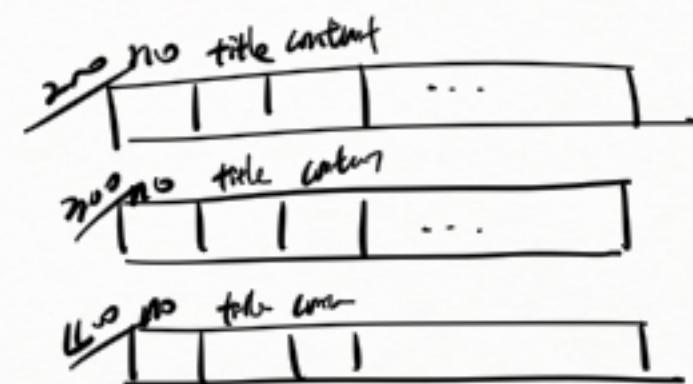
Board[] boards = new Board[3];



boards[0] = new Board();

boards[1] = new Board();

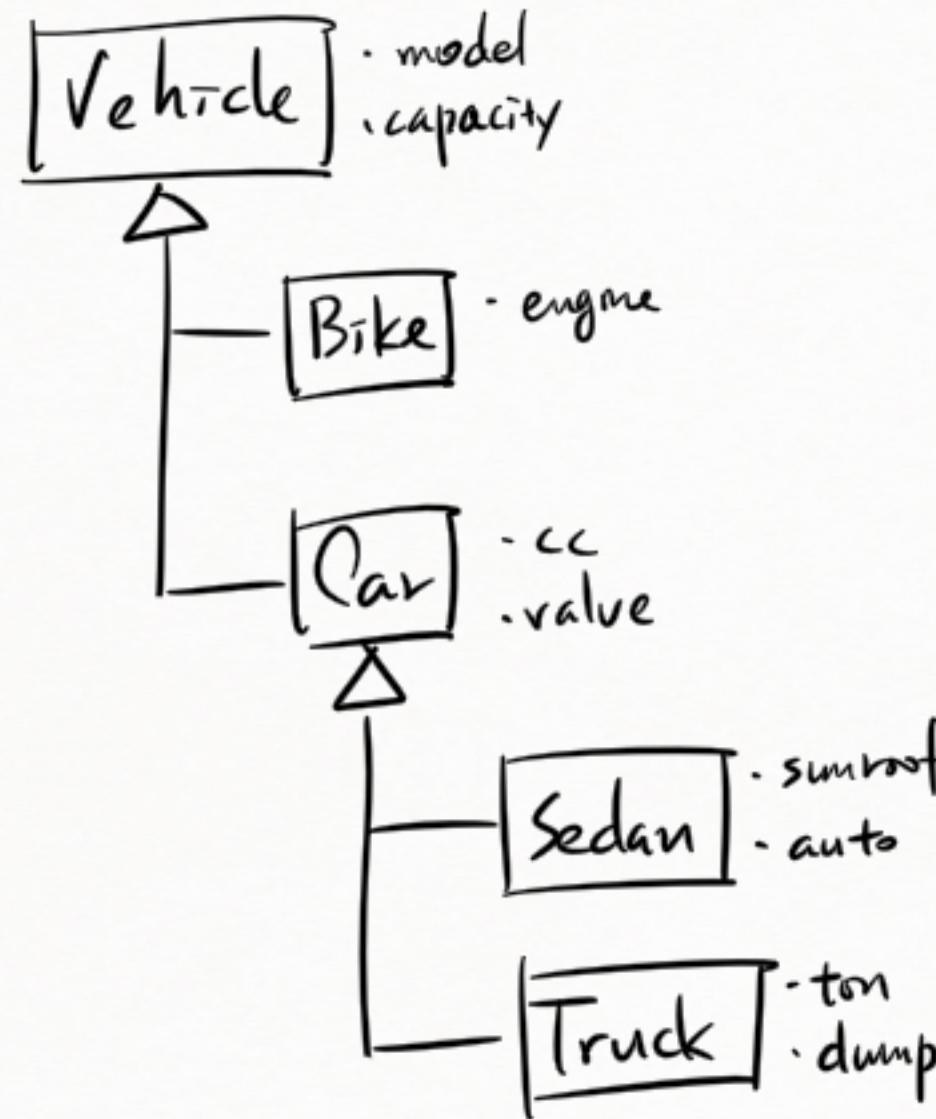
boards[2] = new Board();



다형성 (polymorphism)

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

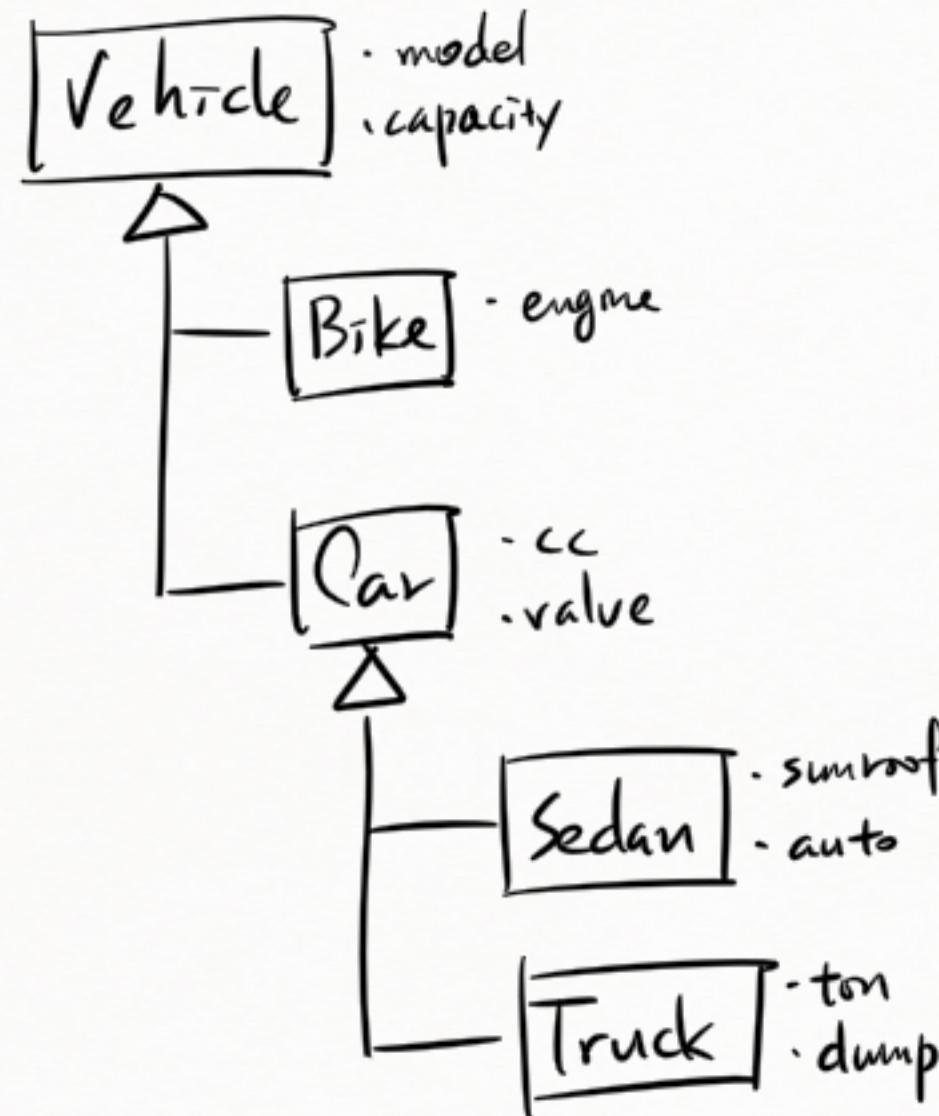
* polymorphic variables



sub class of
이스 퍼스 차/자
차량은 차/자

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

* 다형성 예제 퀴즈



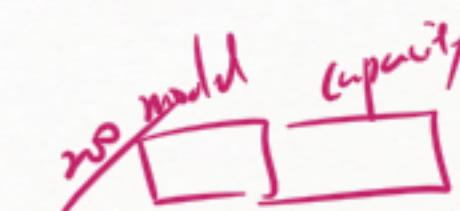
Vehicle obj;

obj = new Car();

Car c = (Car) obj;

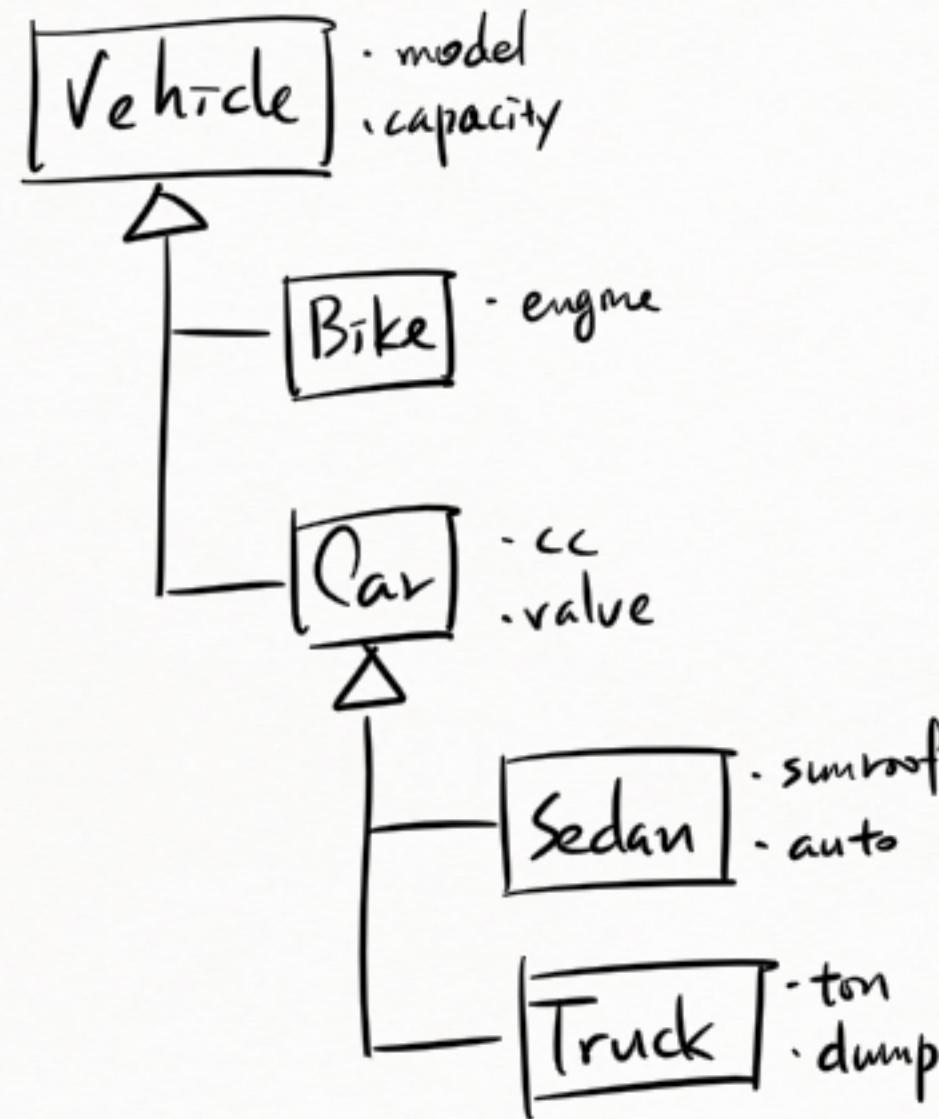
obj = new Vehicle();

Car c = (Car) obj; // 컴파일러는
// 주소를
// 주소를



// 주소를
// 주소를
// 주소를
// 주소를
// 주소를
// 주소를

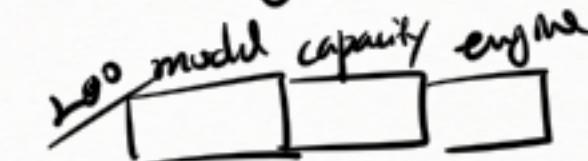
* object oriented



Vehicle obj;

obj = new Bike();

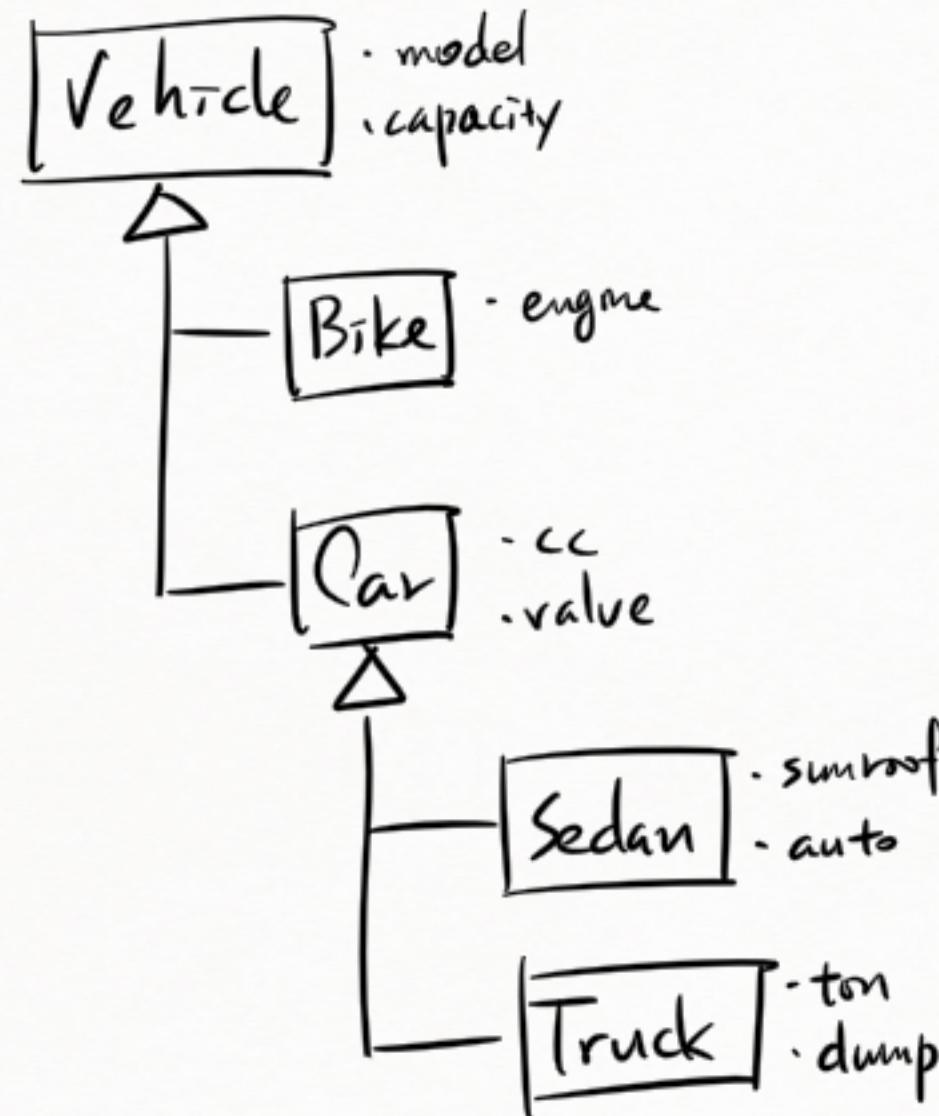
200



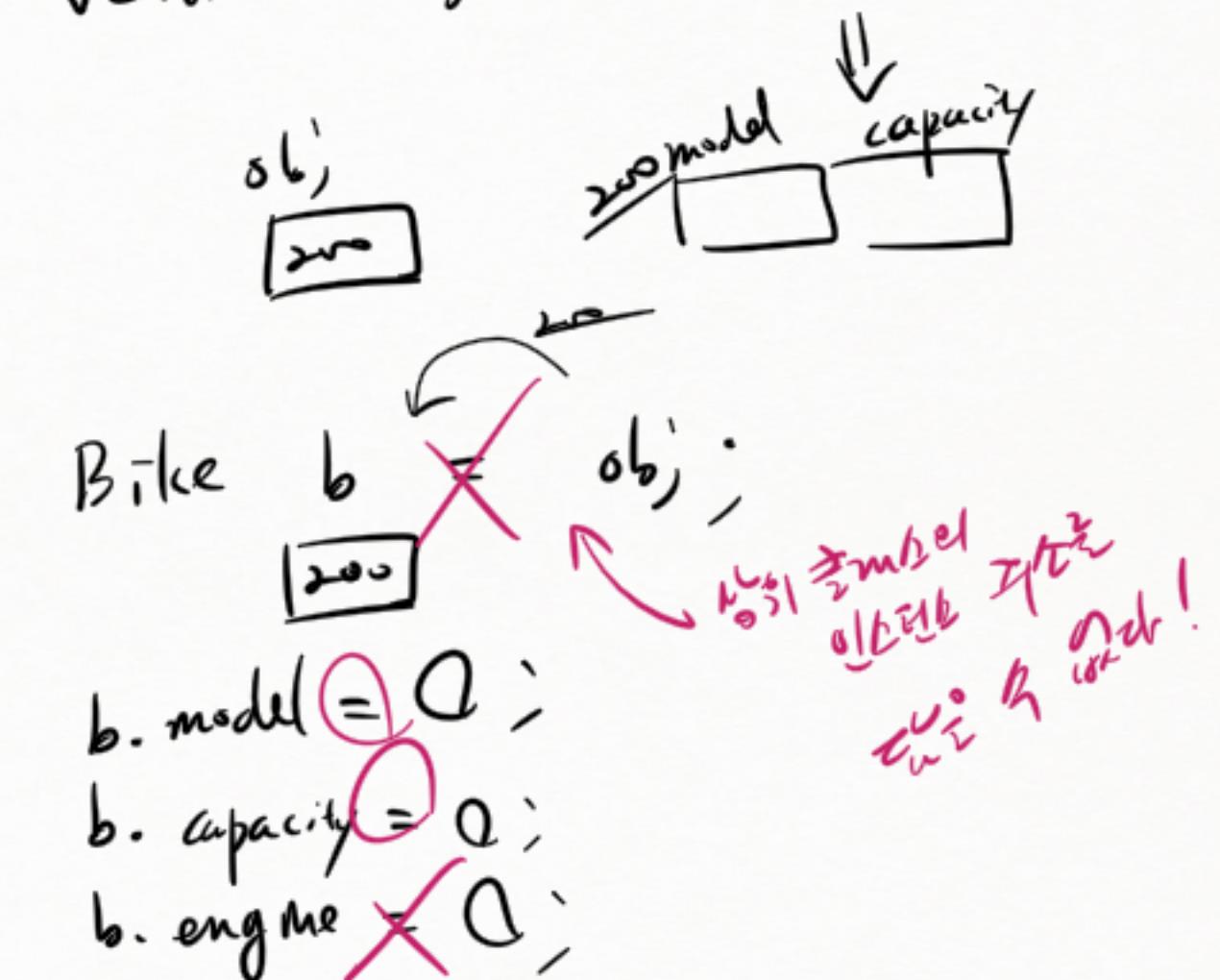
obj. model = 0;

obj. capacity = 0;

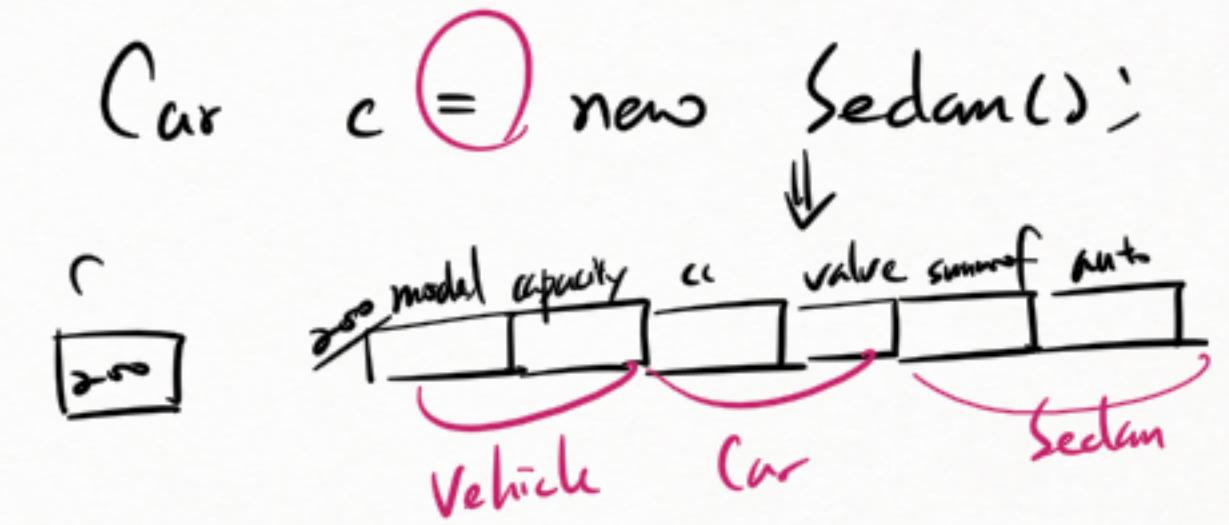
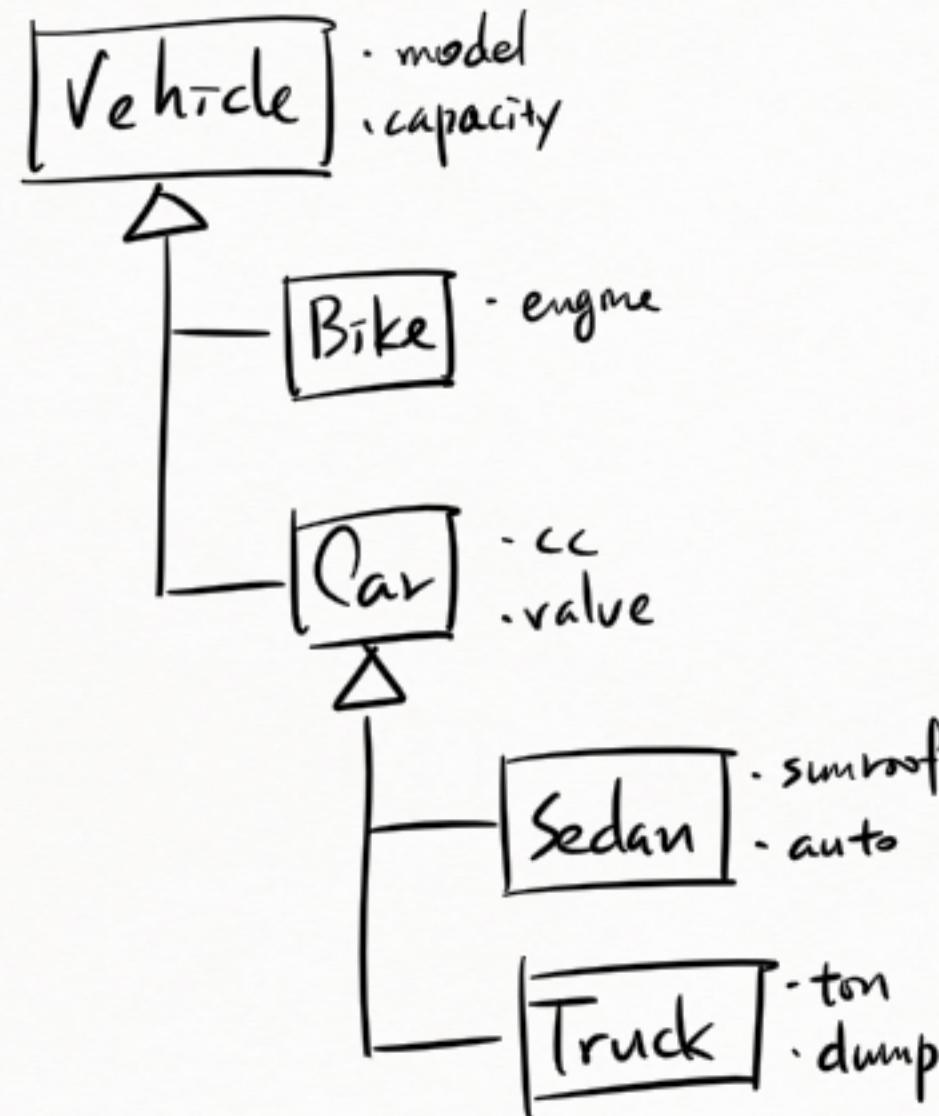
* class의 멤버는



Vehicle obj = new Vehicle();



* class inheritance

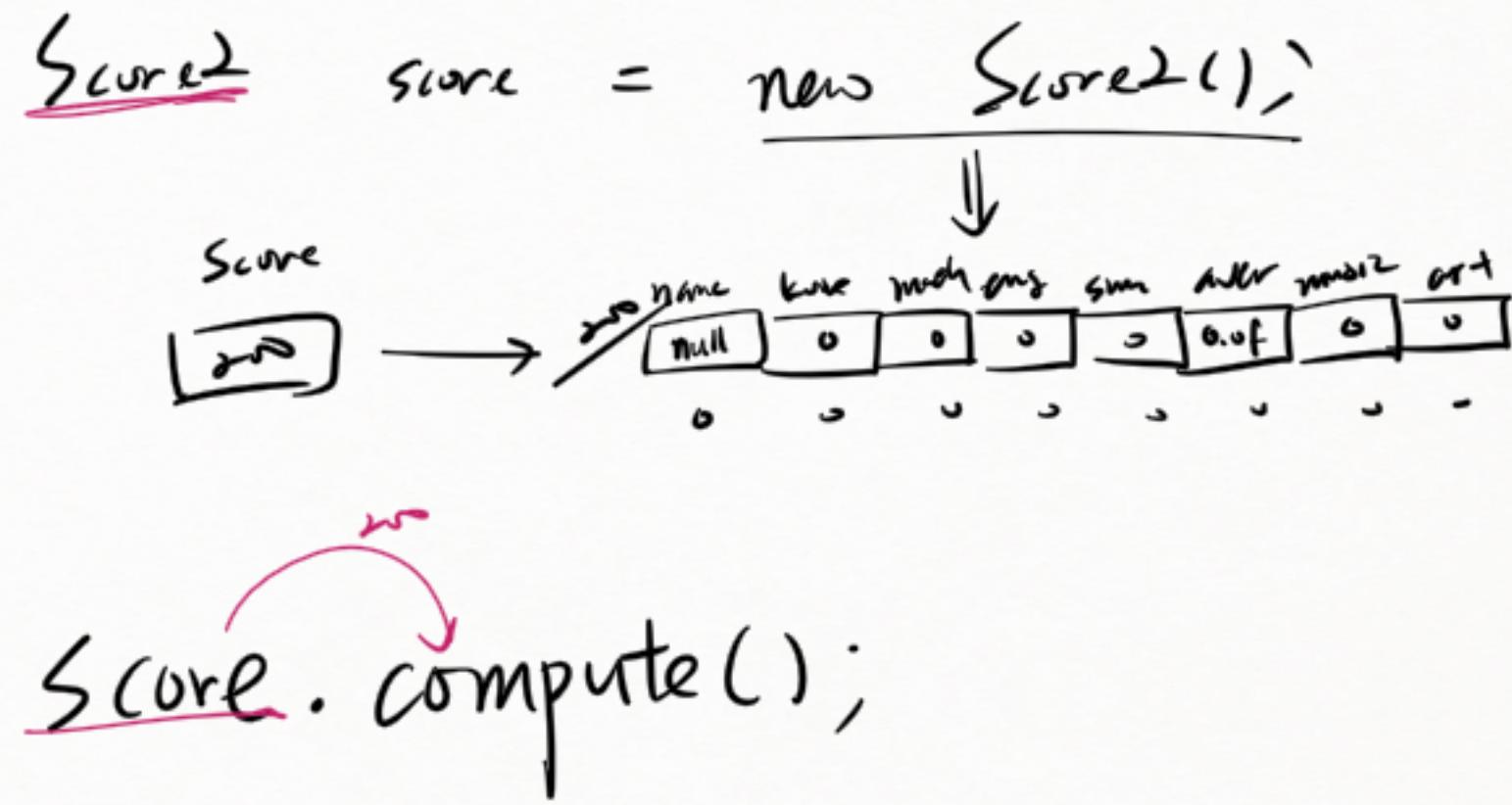
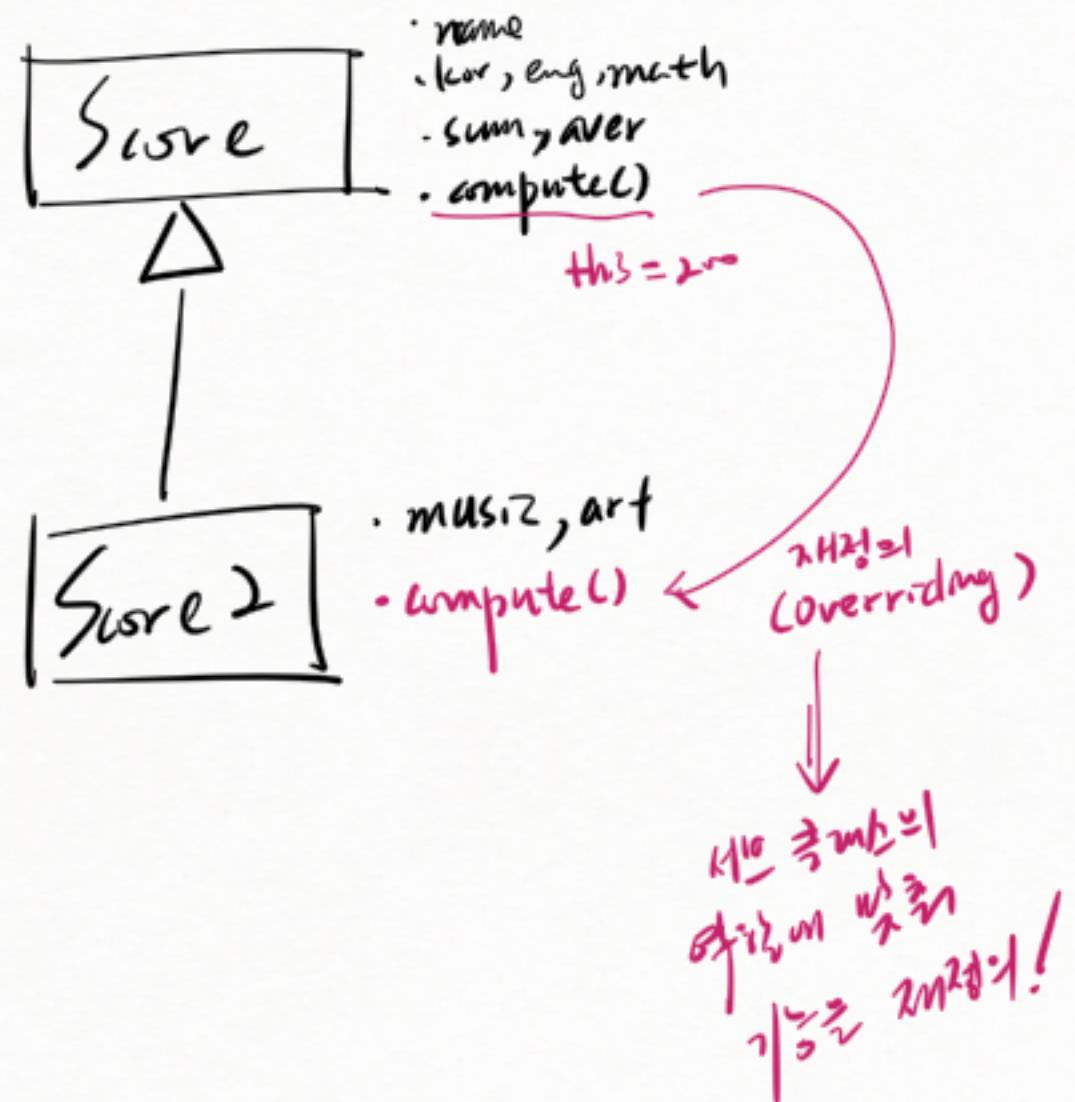


- c. model = 0;
- c. capacity = 0;
- c. cc = null;
- c. value = null;
- c. sunroof ≠ 0;
- c. auto ≠ null;

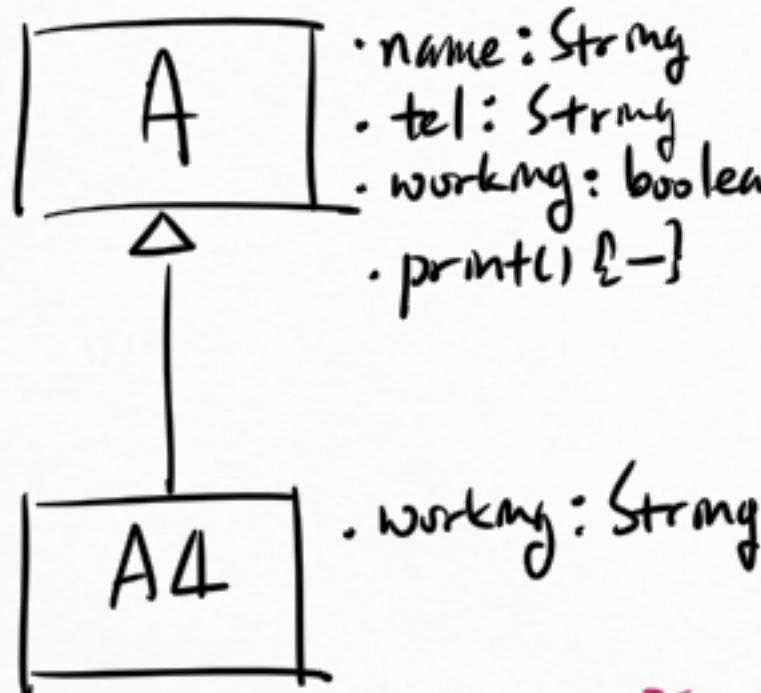
Red annotations provide additional context:

- { 가 가리키는 인스턴스에는 sunroof, sedan 필드가 초기화, sunroof는 c = Car 초기화 되었음 } (가 가리키는 인스턴스에는 sunroof, sedan 필드가 초기화, sunroof는 c = Car 초기화 되었음)
- Car 초기화 후에는 sunroof가 true.
- Car 초기화 후에는 auto가 true.

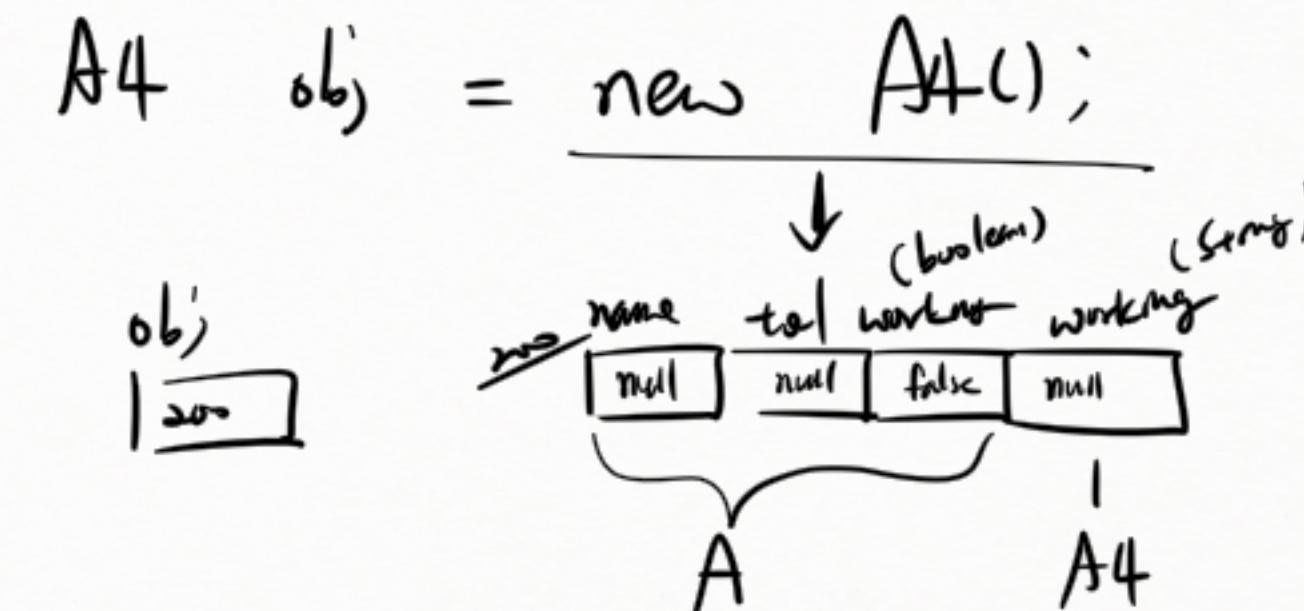
* 상속과 오버라이딩



* 접근 예제와 이팅

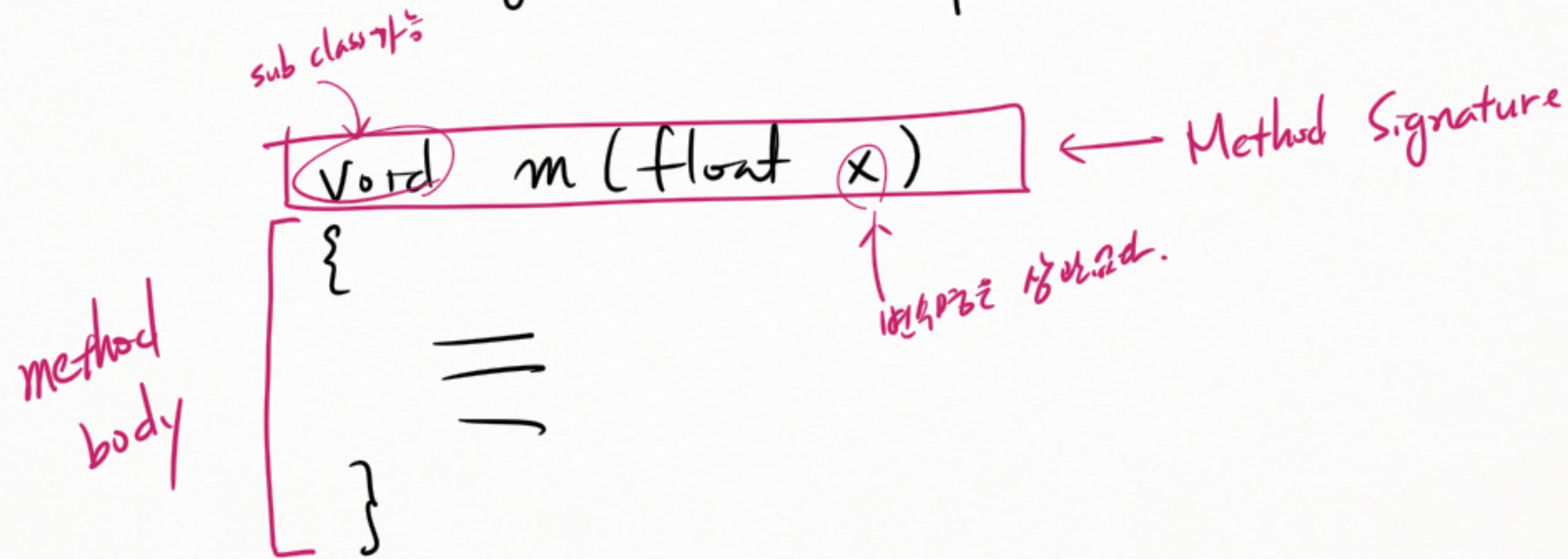


~~obj~~.²⁰ print();

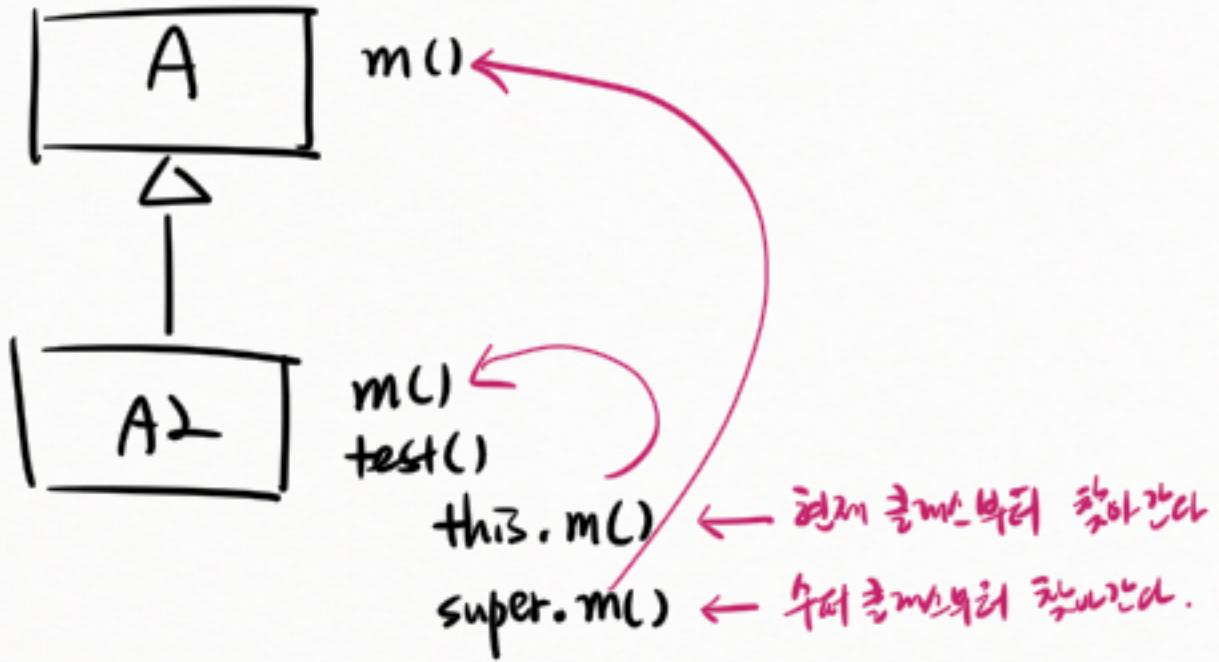


obj.name ← A.name (A의 name 인스턴스 필드라는 의미)
 obj.tel ← A의 tel
 obj.working ← A4의 working

* Method Signature = 'function prototype' in C

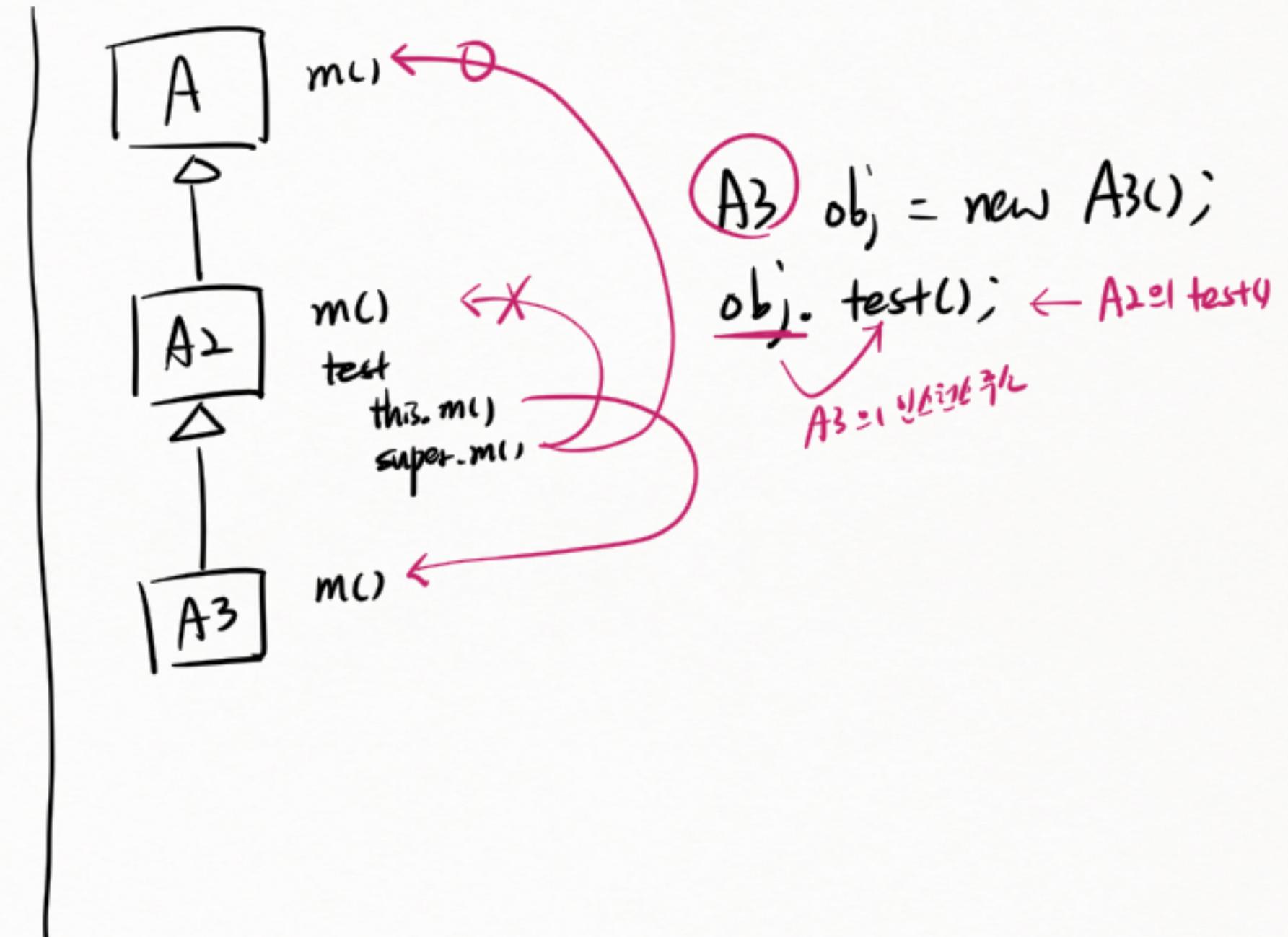


* overriding vs super 키워드



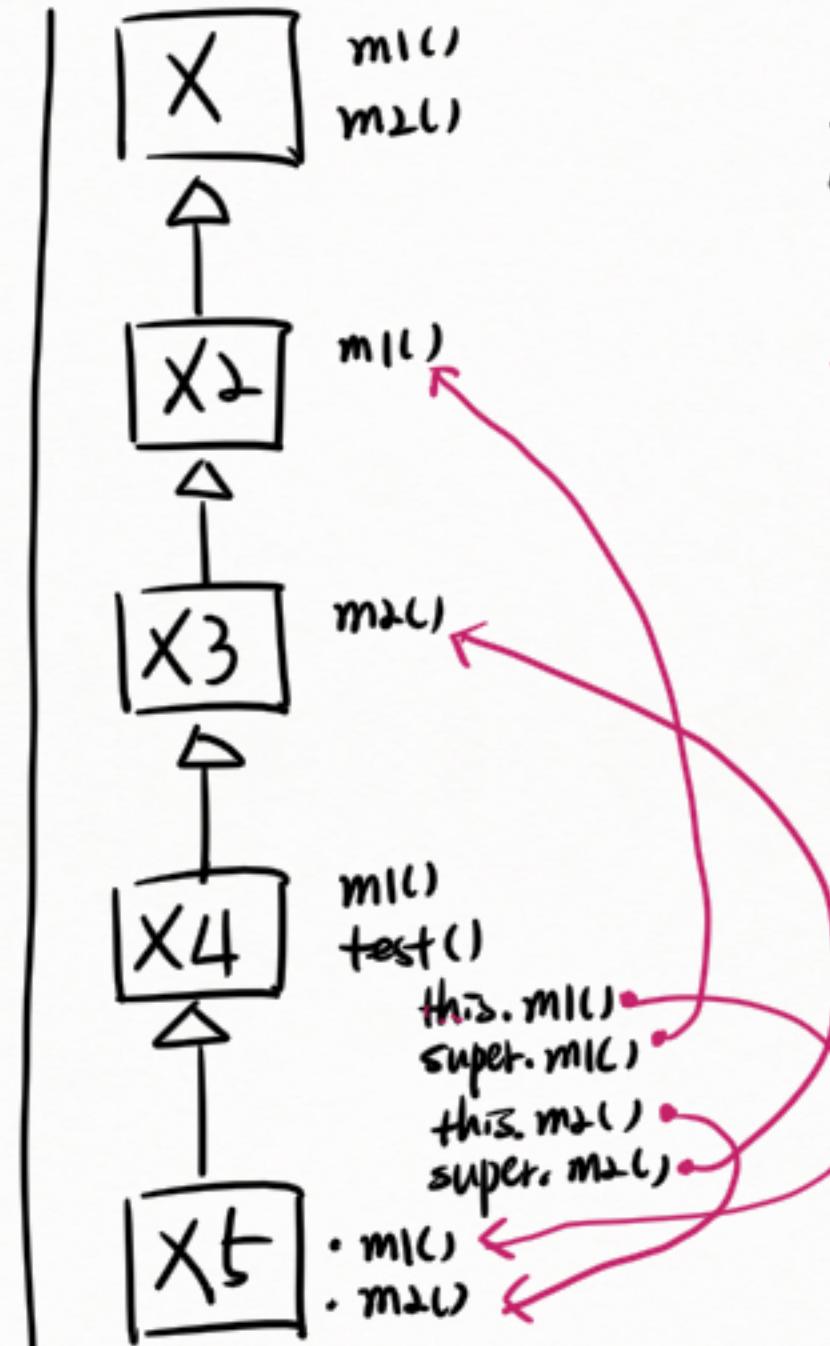
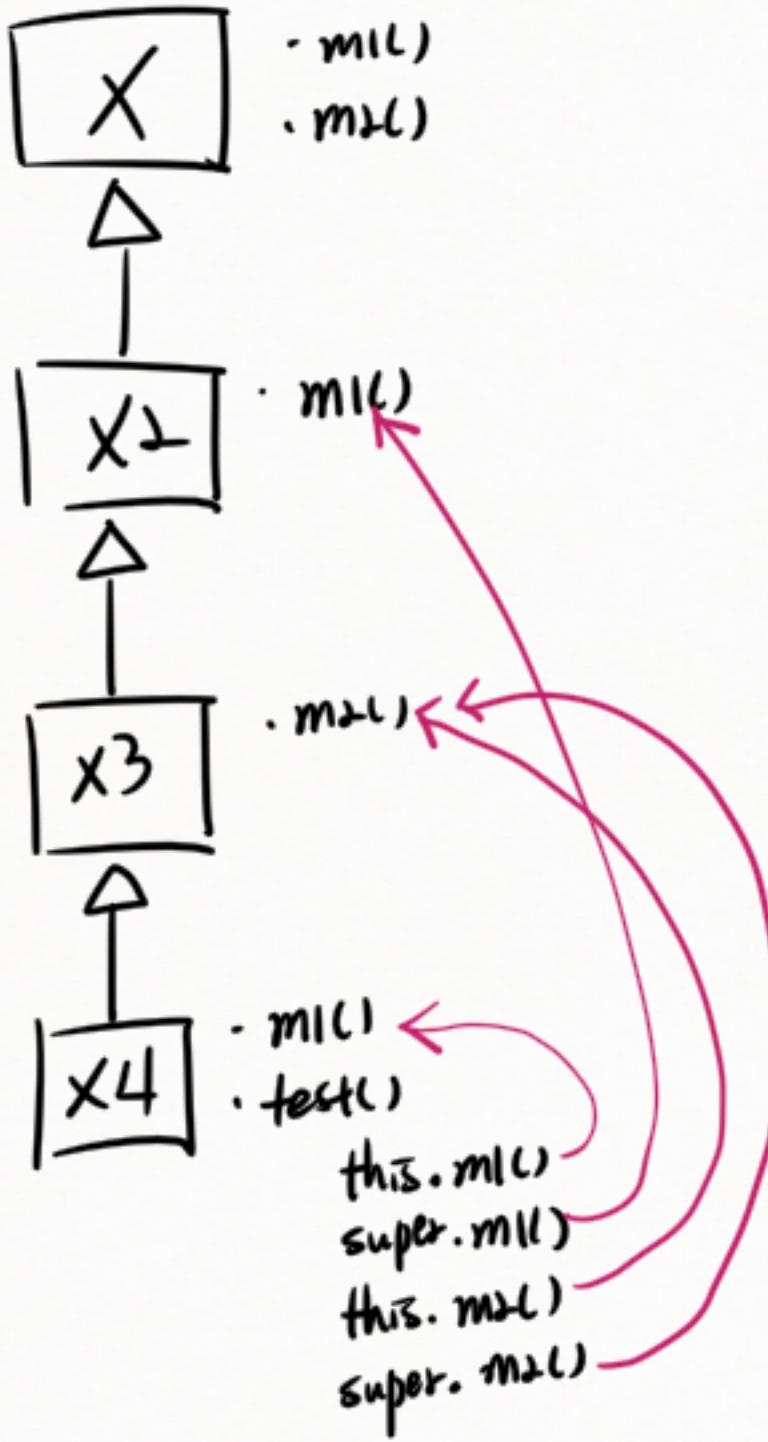
`A2 obj = new A2();`

`obj.test();`



`A3 obj = new A3();`
`obj.test(); ← A2의 test();`
`A3은 obj가 주소`

* overriding in super class

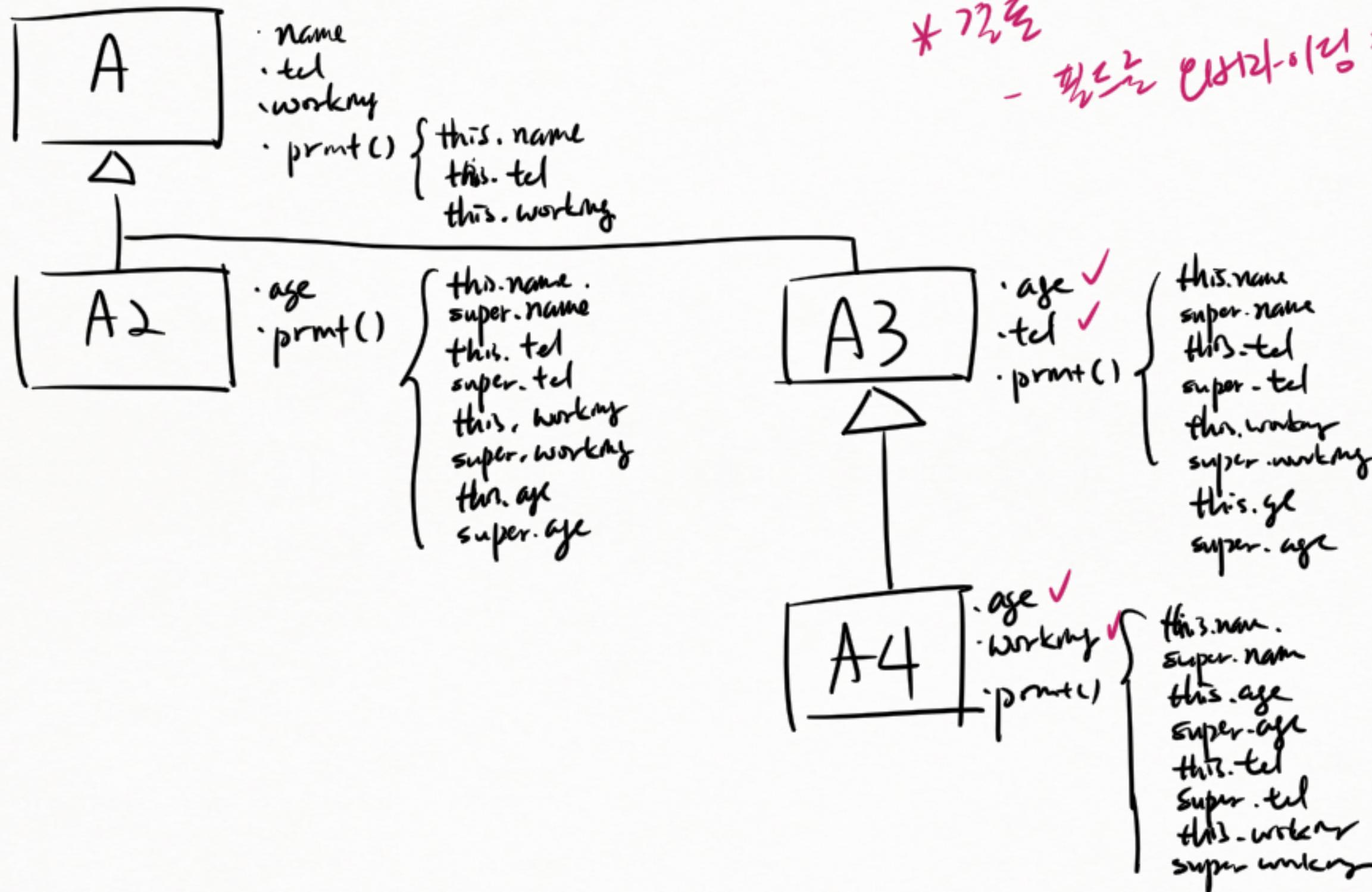


`X5 obj = new X5();`

obj.test(); ← X4의 test()

X5의 m1() ←

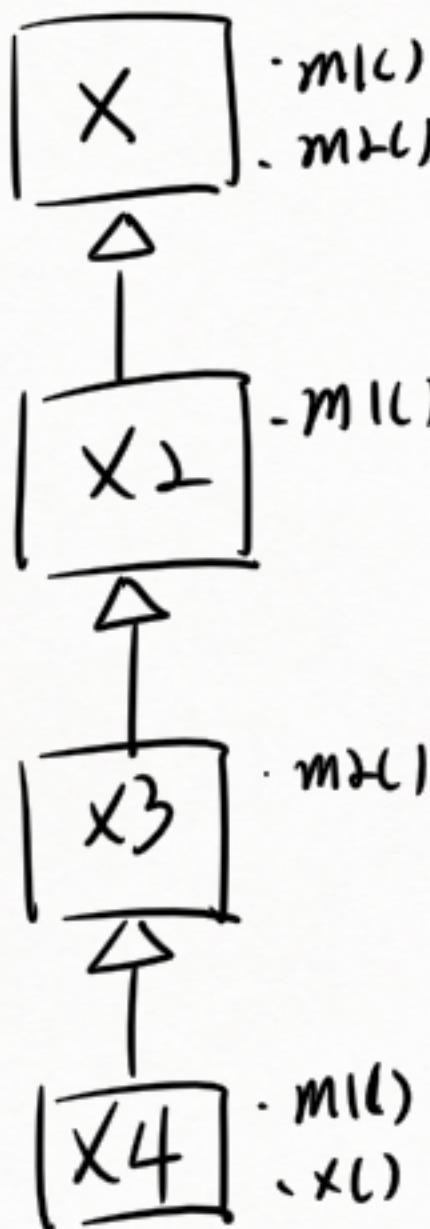
* overriding in super class



* 72번
- 부모의 print()를 상속!

* this. 출력할 때
 ↓
 부모의 print()로
 부모의 출력.
 ↓
 부모의 출력
 출력!

* 상속하고 멤버드 초기화



X4 obj = new X4()

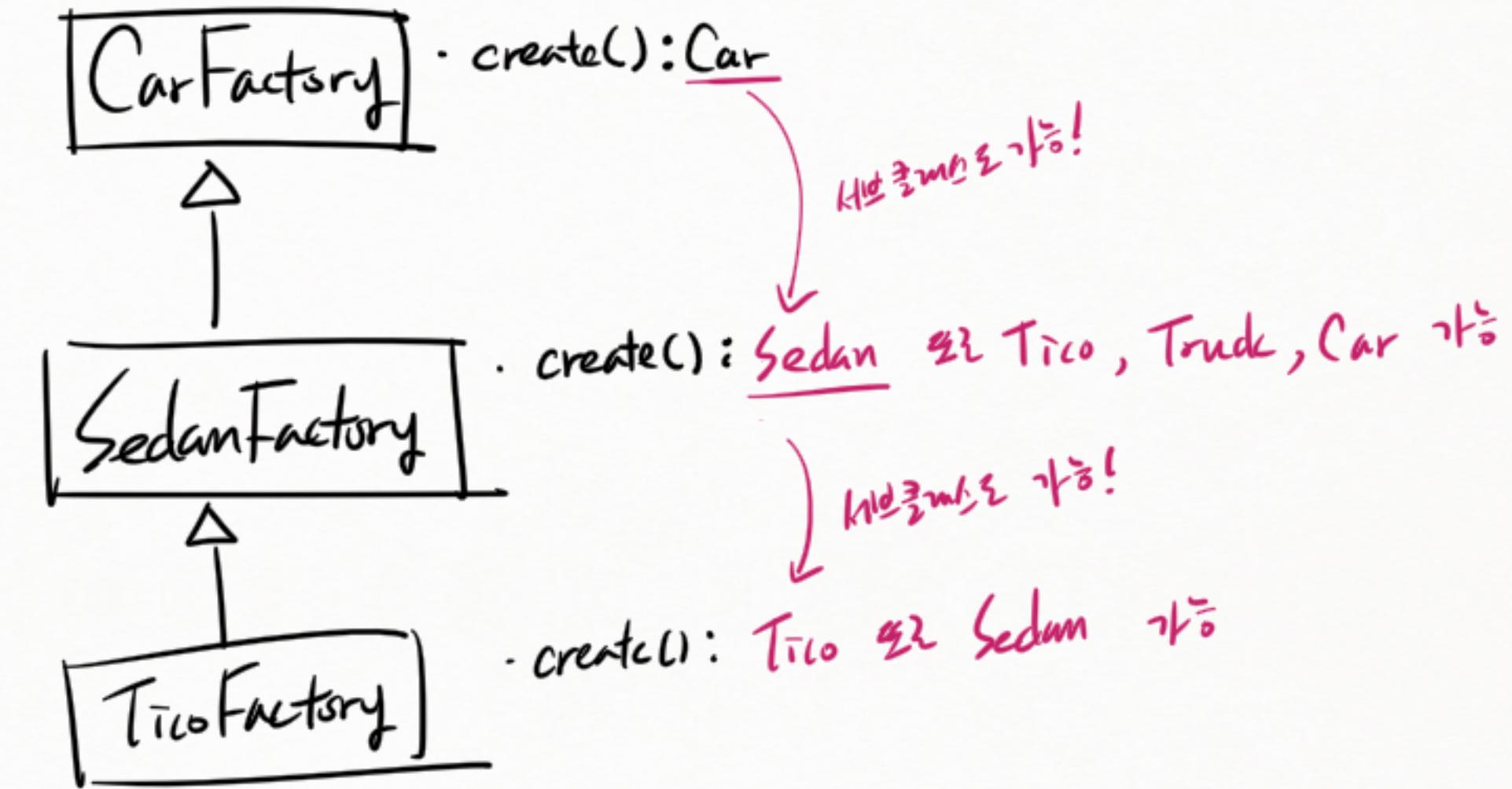
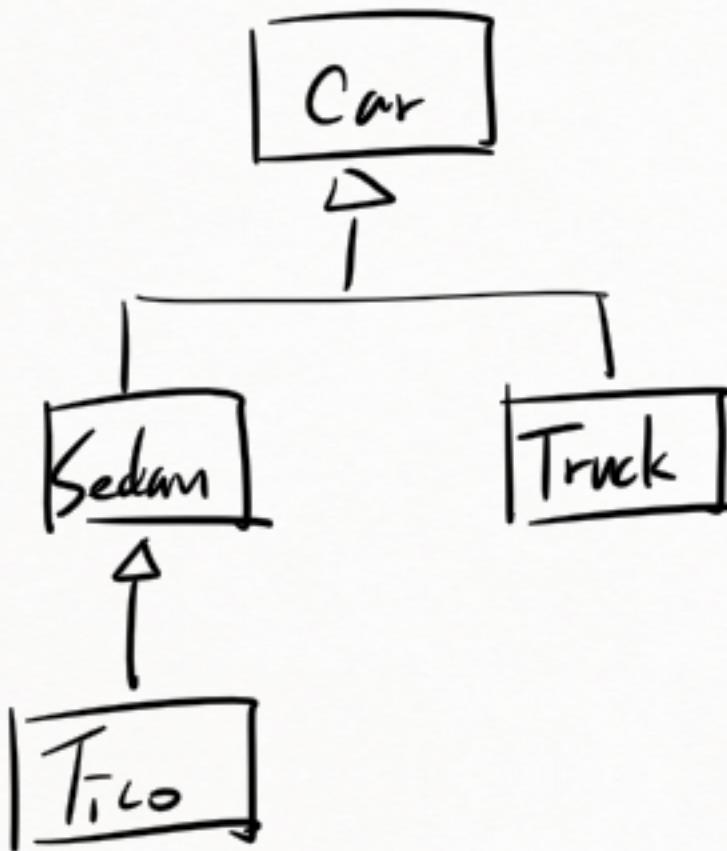
obj. m1(); ← X4의 m1()
obj. m2(); ← X3의 m2()
obj. x(); ← X4의 x()

X4의 m1() → $((X3)obj).m1();$ ← X3 계층도에서 m1()의 초기화가
계층적
obj 가 실제 가리키는 주소, 즉 부모
멤버드를 찾아 올라간다.

X3의 m2() → $((X3)obj).m2();$

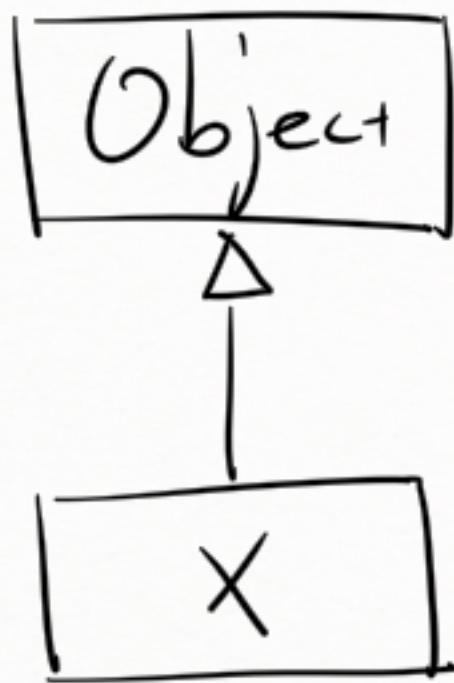
X3의 x() → $((X3)obj).x();$

* 차량 분류 구조도



Object $\cong_{\text{m.u}}$

* Object 클래스의 기타 메서드



QName
" "

FQName

- `toString()` → "Fully-Qualified Name @^{한 번째}"
파기자명 + 클래스명
각 인스턴스에 부여되는 식별번호
- `hashCode()` → 해시번호(int)
- `equals()` → 인스턴스 주소가 같은지 비교
- `getClass()` → Class 객체: 클래스정보를 담는 객체
클래스명, 필드, 메서드, 생성자, 속성, 인터페이스
- `clone()` → 인스턴스 복제 및 리턴
- `finalize()` → GC에 의해 메모리 해제 직전에 호출됨.
C++에서는 소멸자(destructor)라 부른다.
보통 오버라이딩하지 않는다.
- :

* HashSet 와 hashCode()

인스턴스(주소) 목록을 보면

| | |
|----|-------------|
| 주소 | |
| 0 | (500) |
| 1 | |
| 2 | (400) (600) |
| 3 | (200) |
| 4 | |
| 5 | |
| 6 | (300) |
| 7 | |

hashCode()

- ~~new Student("김길동", 20, false);~~ → 211 $\% 6 = 3$
- ~~new Student("김길동", 20, false);~~ → 142 $\% 6 = 6$
- ~~new Student("이민수", 21, true);~~ → 314 $\% 6 = 2$
- ~~new Student("이민수", 22, true);~~ → 208 $\% 6 = 0$
- ~~new Student("박민수", 32, false);~~ → 10 $\% 6 = 2$

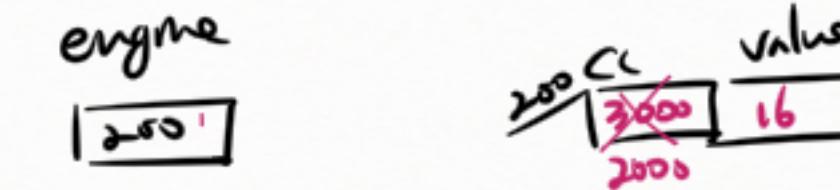
Hash + Set

같은 주소는 같은 Hash
가지는 다른 Hash

같은 Hash는 같은 주소!

* clone() में : shallow copy

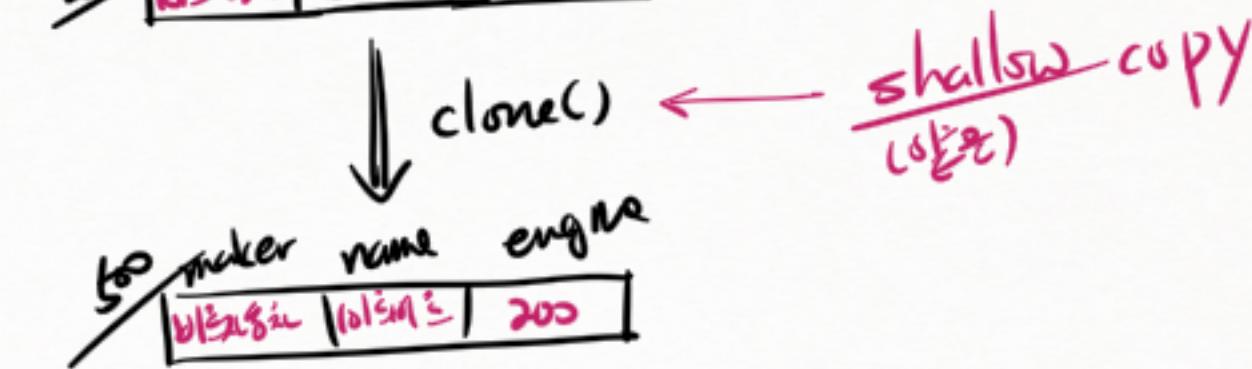
Engine engine = new Engine();
 ↓



Car car = new Car();
 ↓

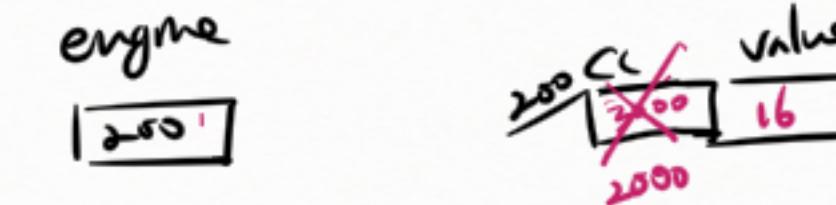


car2
| 500 |

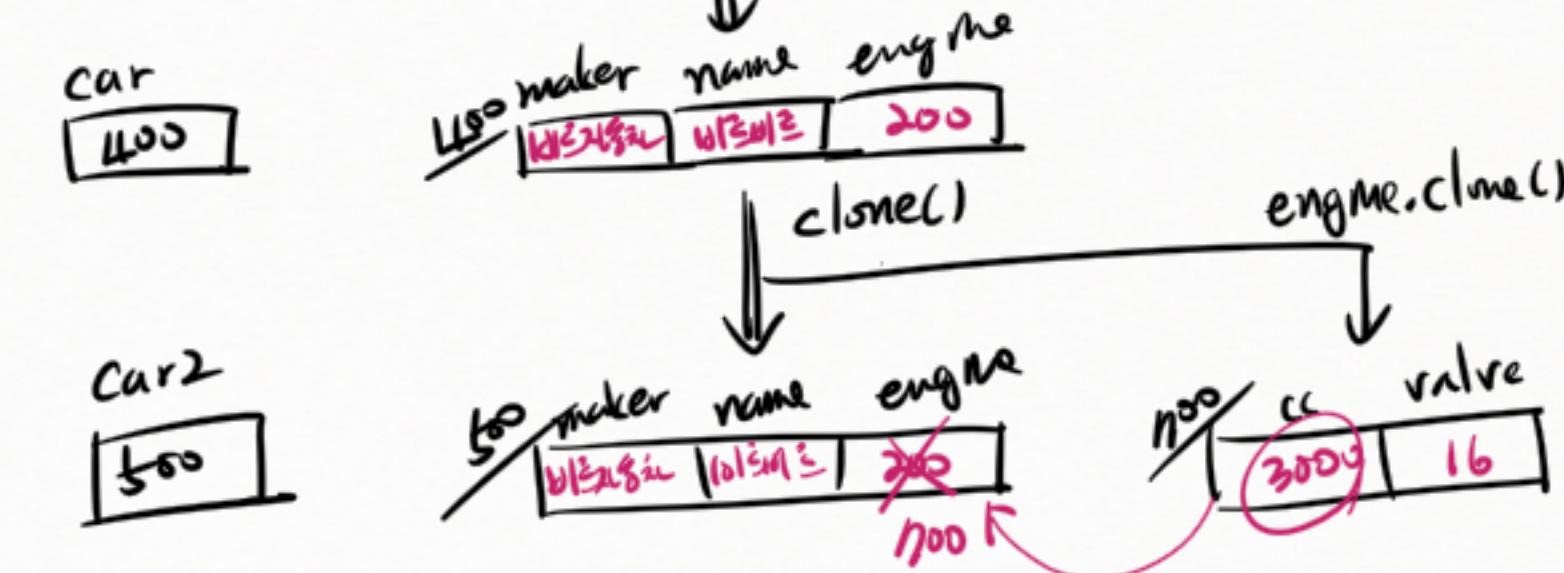


* clone() में : deep copy

Engine engine = new Engine();



Car car = new Car();



String $\frac{z}{2}^m \underline{1}$

* 21|21번스와 인터넷

String s1 = new String("Hello"); ← String 인스턴스
Heap 공간 대입.

~~200~~ | H | e | l | l | o | ...

String s2 = new String("Hello");

~~300~~ | H | e | l | l | o | ...

* 문자열 리터럴의 String 인스턴스

String s1 = "Hello";
String literal

| 200

문자열 상수를 바로 복사
할 때마다 영구히 메모리
String 인스턴스

~~Hello~~

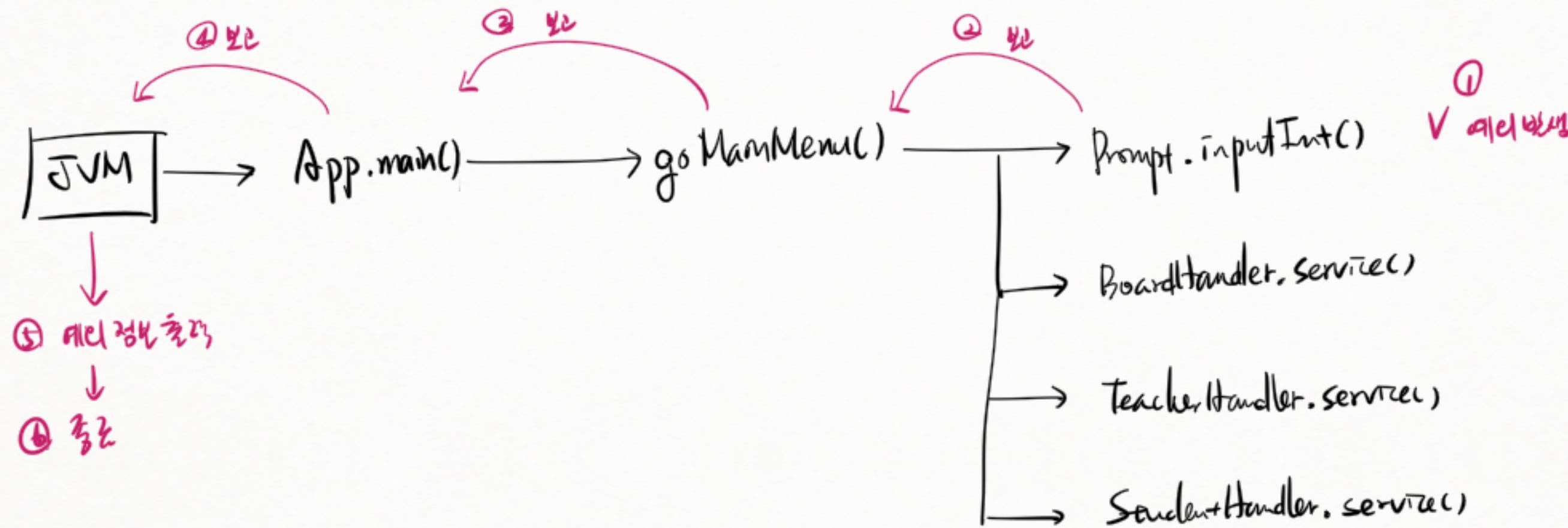
String s2 = "Hello";
String literal

| 200

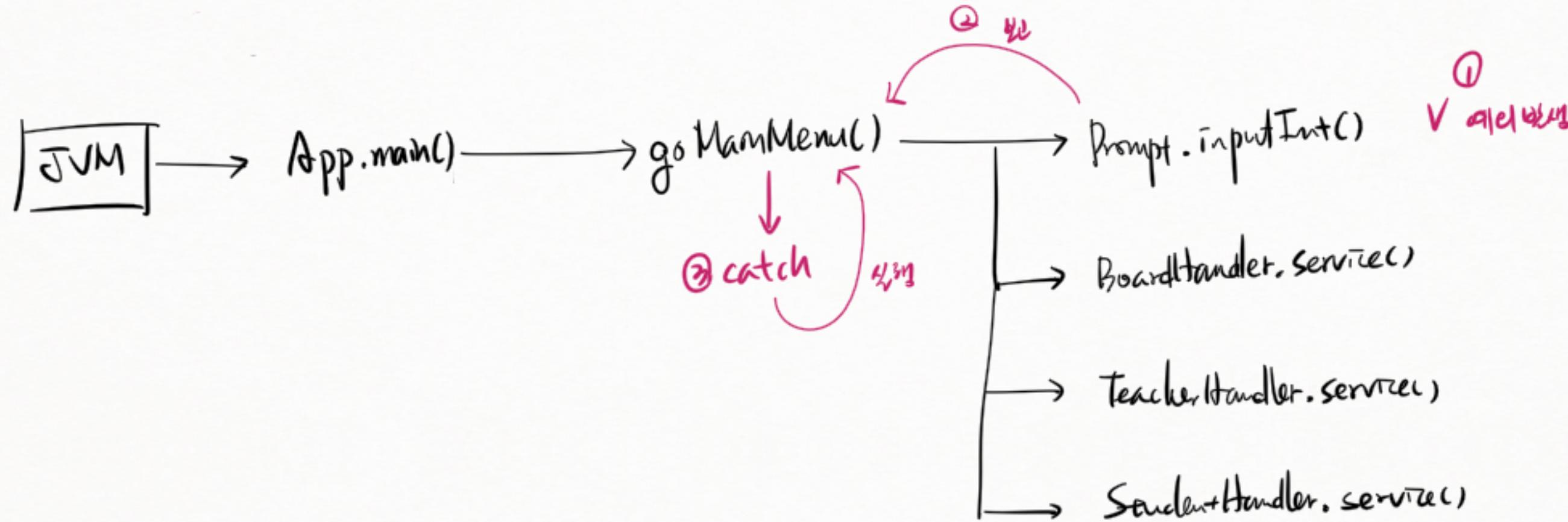
할 때마다 영구히
스트링의 주소를
기준으로 인스턴스를
복사합니다.

예외처리 (try ~ catch ~ finally)

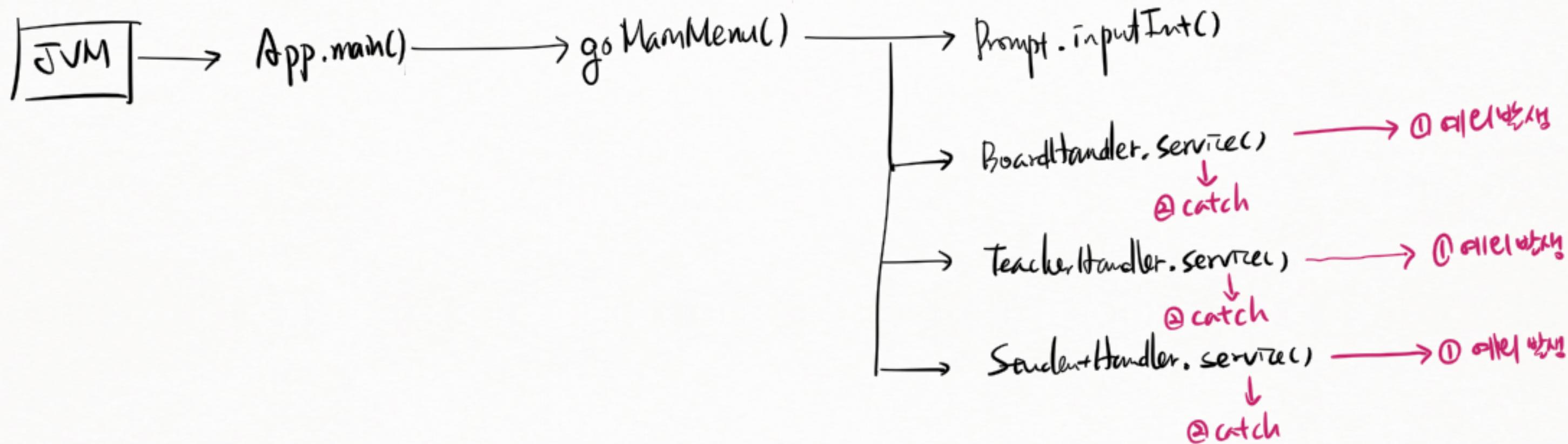
* 예외 발생과 catch



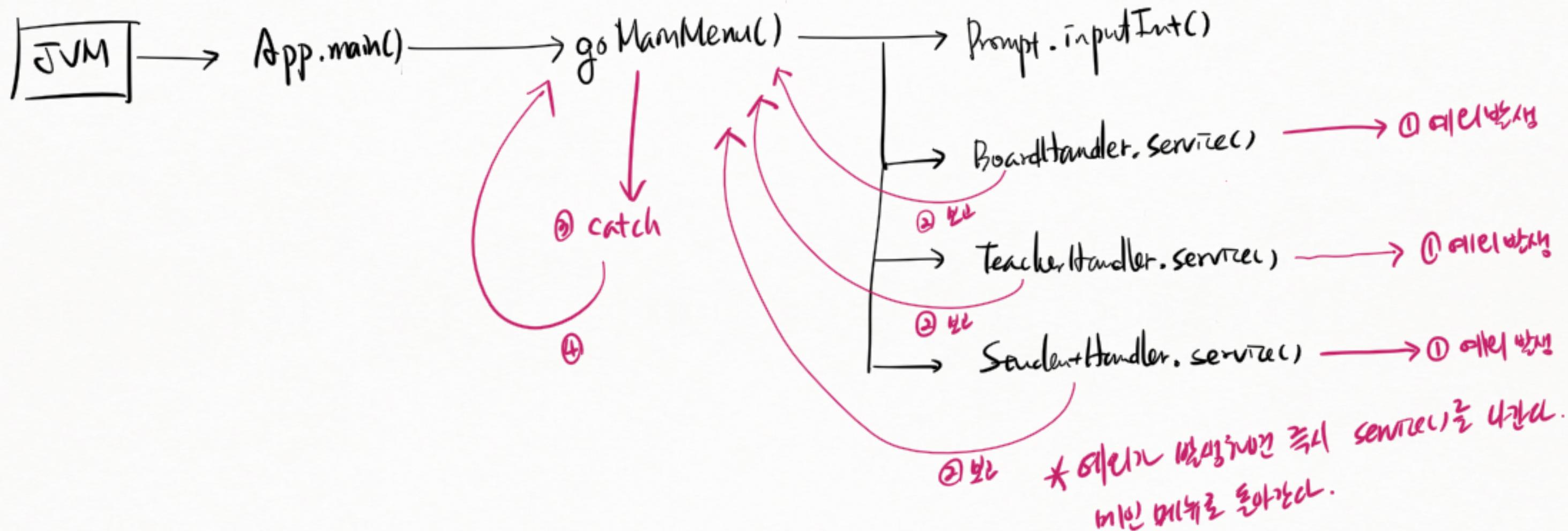
* 예외 발생과 catch



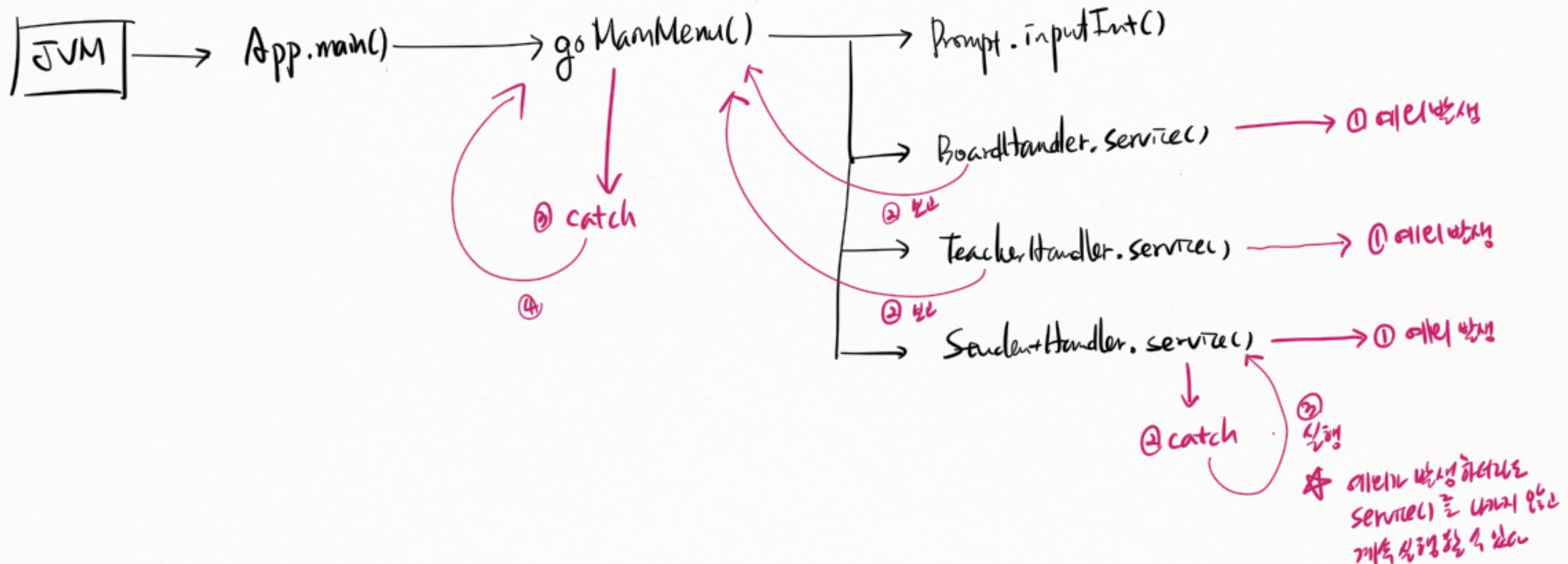
* 예외 발생과 catch



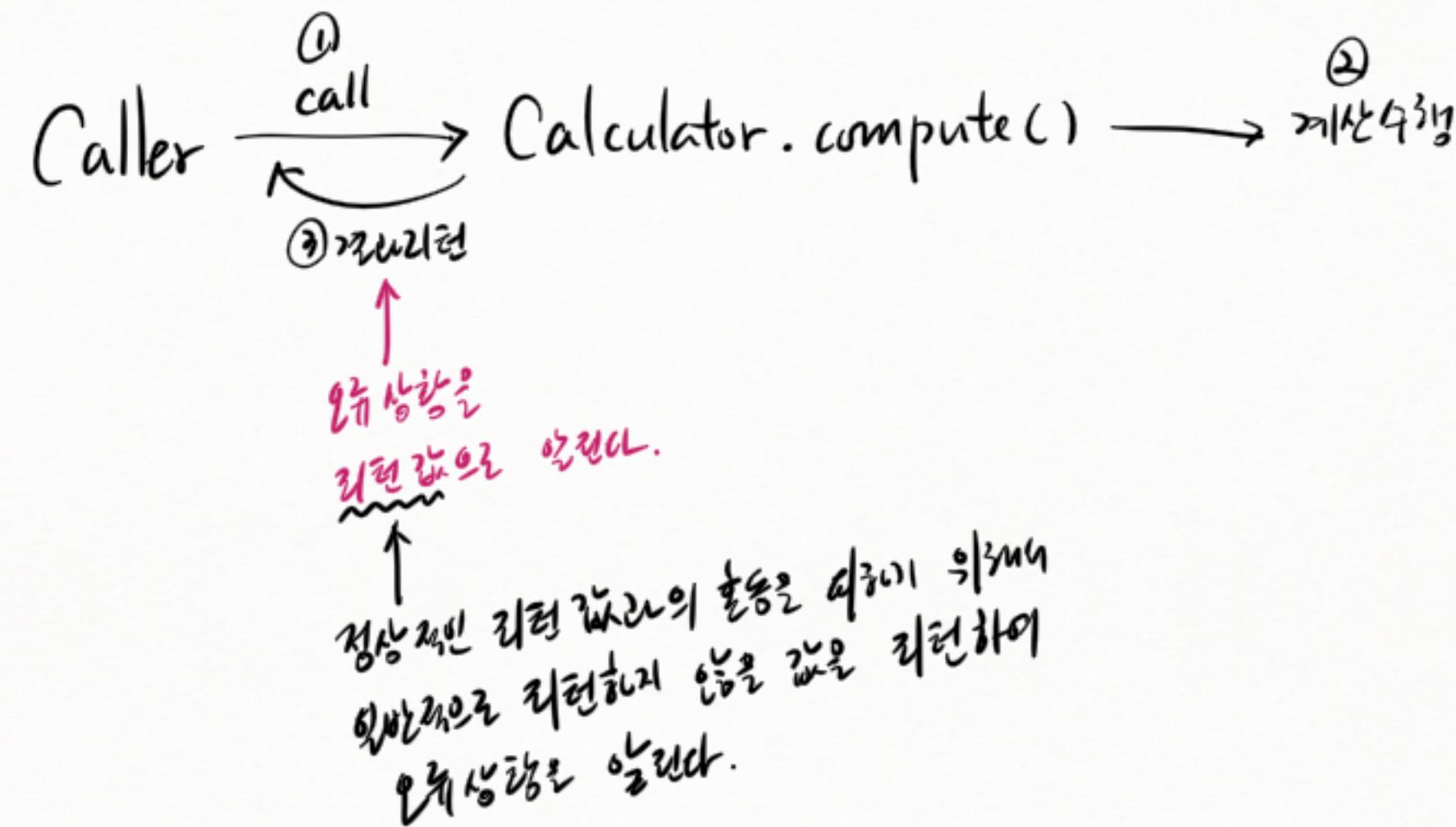
* 예외 발생과 예외 처리



* 예외 발생과 예외 처리



* 예제 1) 상황을 알리는 고전적인 방법



* 예외를 던지는 허용적인 방법

throws 예외정류를 갖는 타입;

↳ 반드시 Throwable 타입이어야 함.

예) throws new Throwable("예외 내용");

★ ↓
예외를 던지는 메서드는 선언부에 선언됨.

* 예외는 단지 한 가지의 Signature

자연적으로 throws (...) throws 예외들, ... { }

↳ Throwable 계층

int compute(...) throws Throwable { }

이면 예외는 단지 한 가지다.

* 예외는 타입은 예외

try {

예외를 처리하는 데

} catch (Throwable e) {

예외처리코드

} finally {

예외를 처리하는

정리하는 코드

예외

처리하는 코드

}

예외 처리하는 것은 자연

Throwable 타입
(+ 예외를 처리하는 코드)

