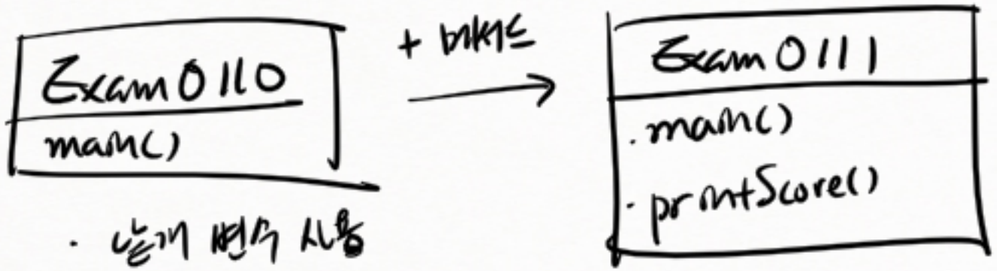
* 클래스 문법 활용 예



- 날개 변수 사용

- 메서드 문법 활용
  ↳ 파라미터 사용

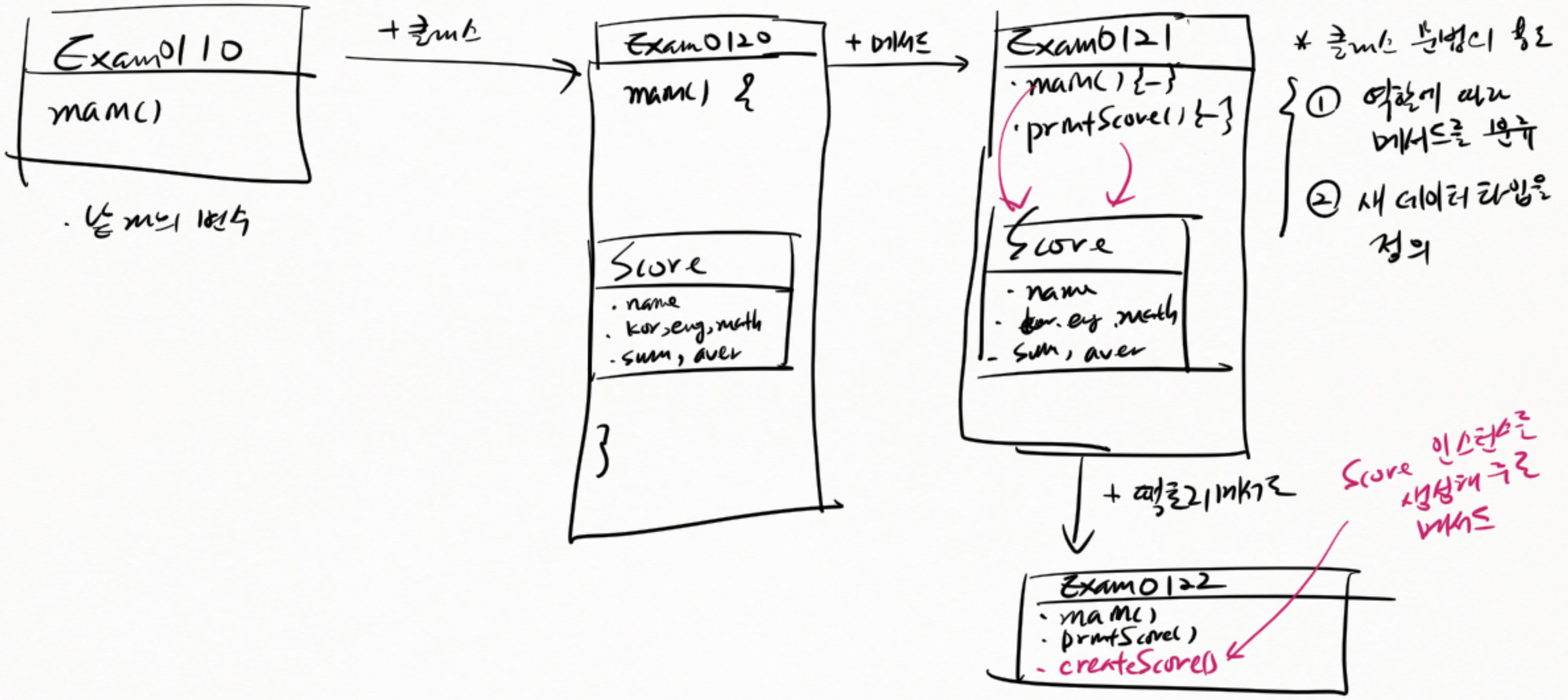
⇓

✓ 중복 코드 제거

↓

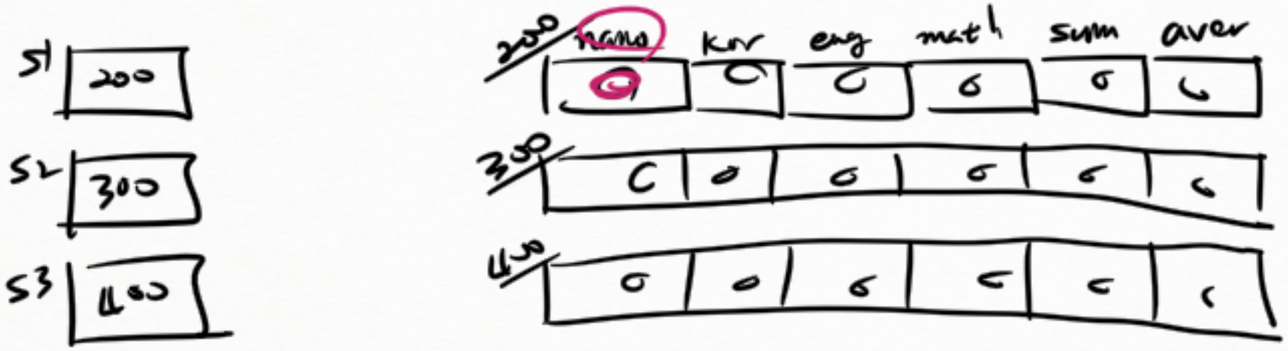
코드 재사용 ↑

✓ 유지 보수가 쉬워짐


Exam0110
main()
+클래스
Exam0120
main() {
Score
. name
. kor, eng, math
. sum, aver
}
+메서드
Exam0121
. main() {-}
. prntScore() {-}
Score
- name
- kor. eng. math
- sum, aver
* 클래스 문법의 목적
① 알맞게 따라 메서드를 호출
② 새 데이터 타입을 정의
+ 팩클리메서드
Exam0122
. main()
. prntScore()
- createScore()
Score 인스턴스를 생성해 주는 메서드

* 레퍼런스 변수 ⟶ 여러 개의 인스턴스를 다룸

Score s1, s2, s3 ;



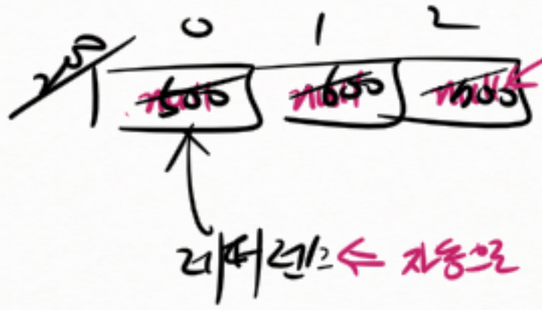
s1.name = "";

s1.kor = 100;

⋮

* 레퍼런스 배열

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

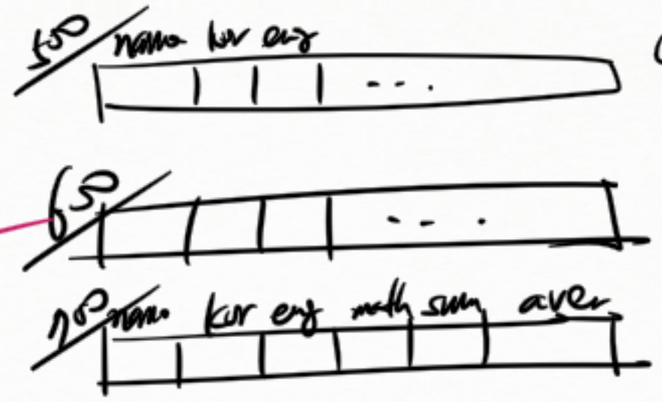


null ?
- 클래스가 아니다!
- 참조변수가 0으로 초기화되었음을 뜻한다.

레퍼런스 ⇐ 자동으로 null 값으로 초기화된다
*로컬변수는 초기화되지 않는다.

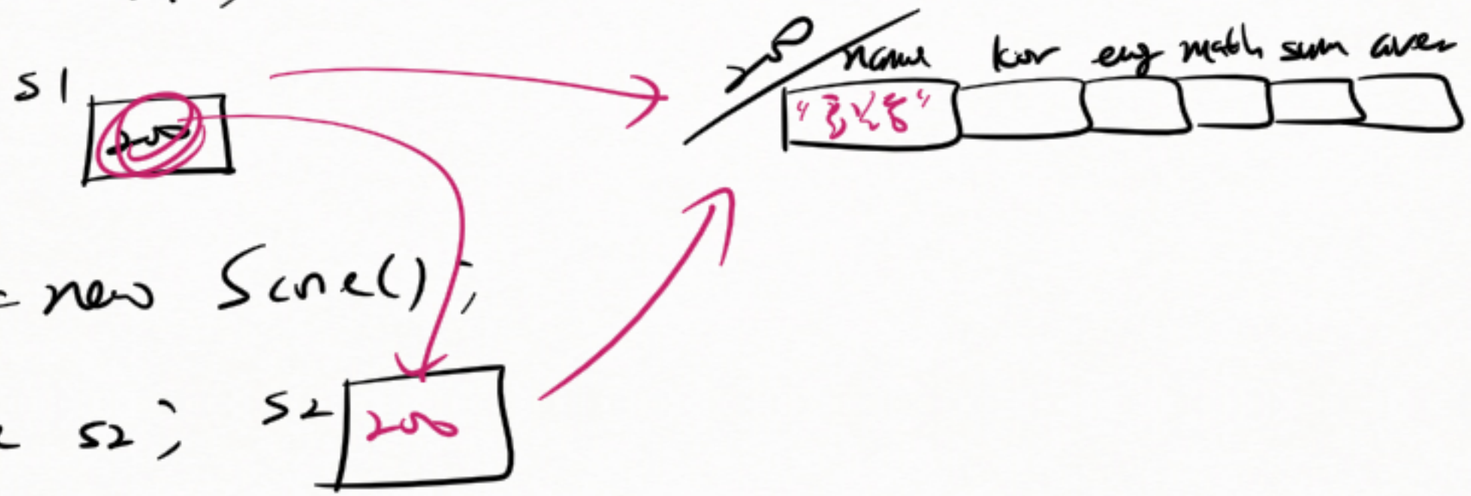
```
arr[0] = new Score();
arr[1] = new Score();
arr[2] = new Score();
arr[3] = new Score();
```

*ArrayIndexOutOfBoundsException



* 레퍼런스 와 인스턴스

Score s1;



s1 = new Score();

Score s2; s2 200

s2 = s1;

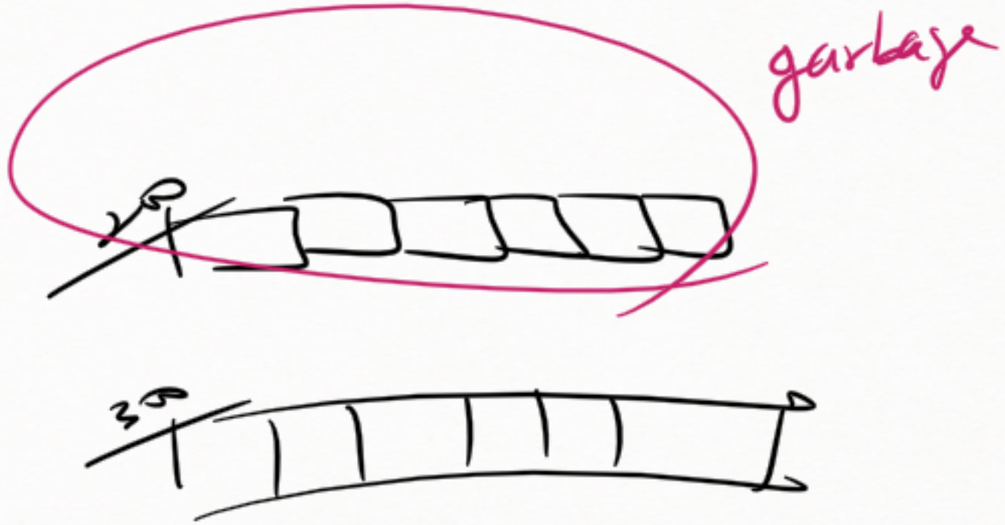
s2.name = "홍길동";

* 가비지 (garbage)

Score s1;

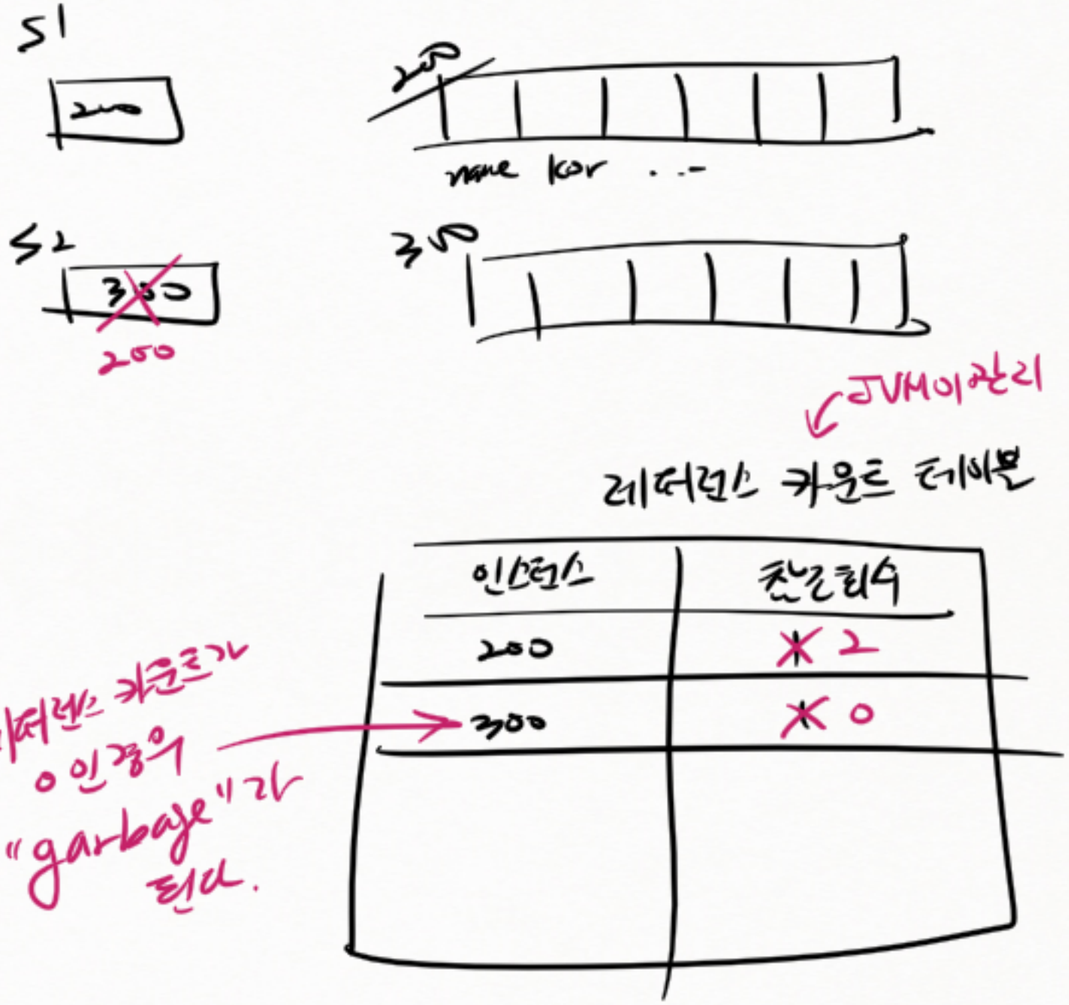
s1 = new Score();

s1 = new Score();



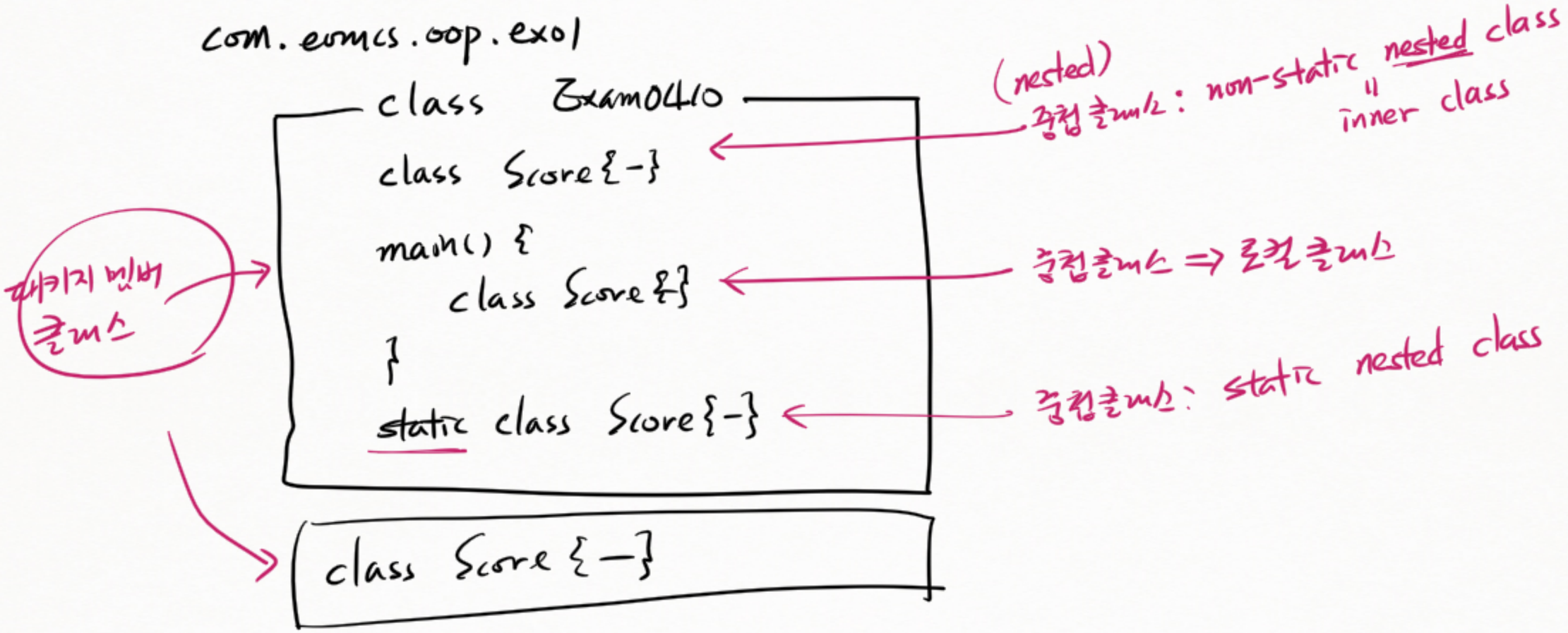
* 레퍼런스 카운트란 가비지

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

s2 = s1;
```

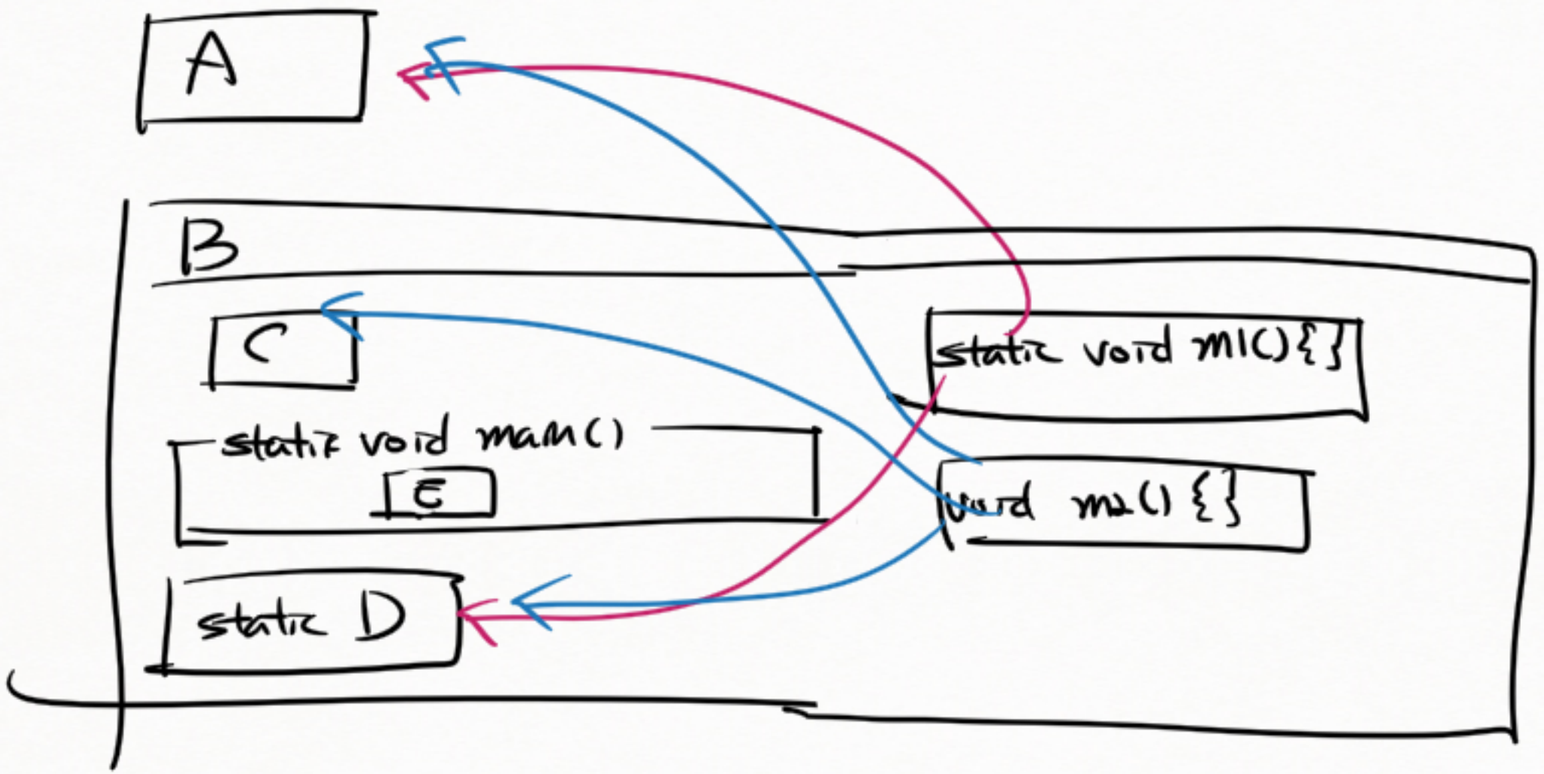


* 클래스 리뷰

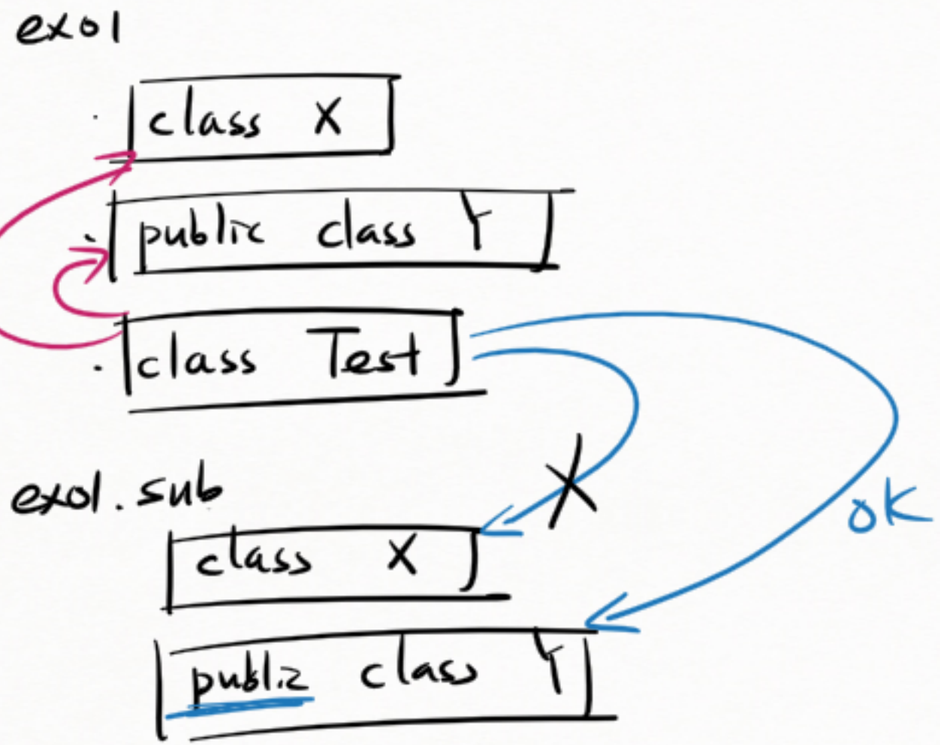
com.eomcs.oop.ex01



* static 멤버와 non-static 멤버



* 공개 클래스와 기본 클래스



* 클래스 문법의 활용 예: ① 사용자 정의 데이터 타입 을 만드는 용도

User-defined Data Type
개발자

Score s1 = new Score();

s1.name = "홍길동"

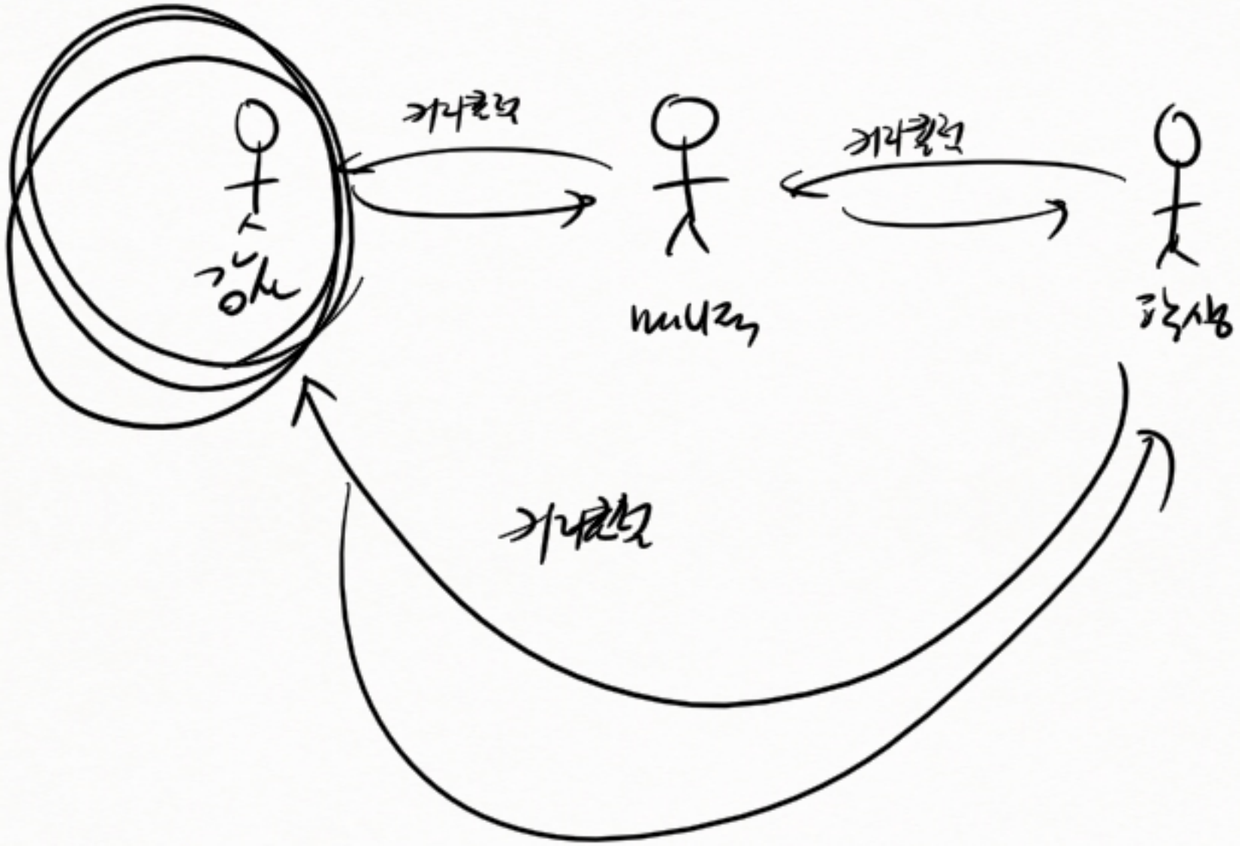
① s1 레퍼런스에 저장된 주소로 찾아가서 해당 인스턴스의 name 변수 —
② s1 레퍼런스가 가리키는 인스턴스의 name 변수 —
③ s1 인스턴스의 name 변수 —
④ s1 객체의 name 변수 (필드)
⑤ s1의 name 필드 (변수)

※ optimizing(최적화) vs refactoring(코드 깨끗)

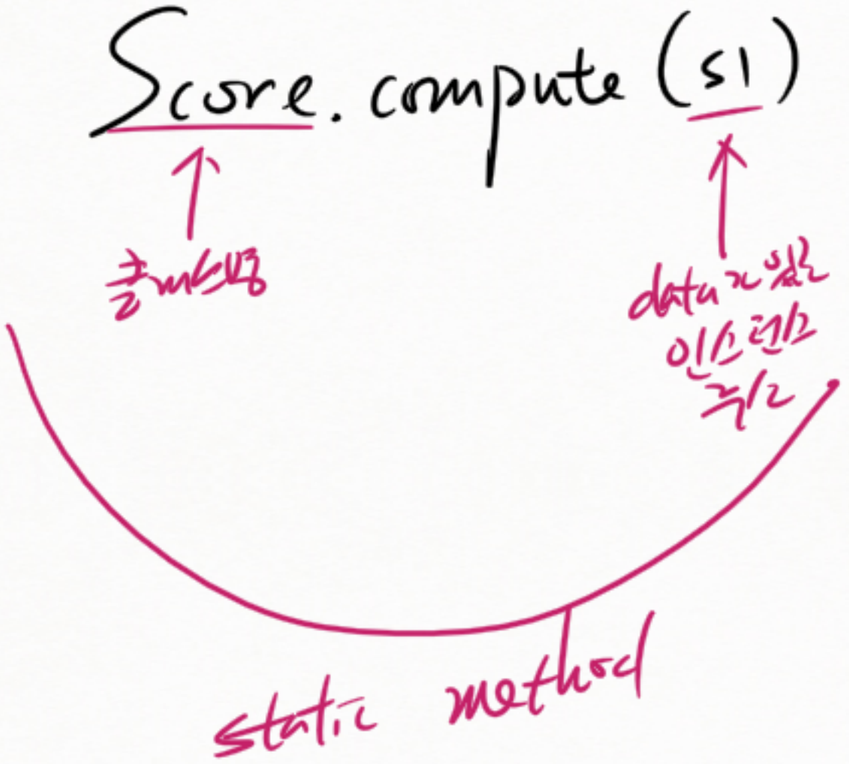
optimizing(최적화)
↓
. 속도↑
. 코드 난해 → 유지보수↓

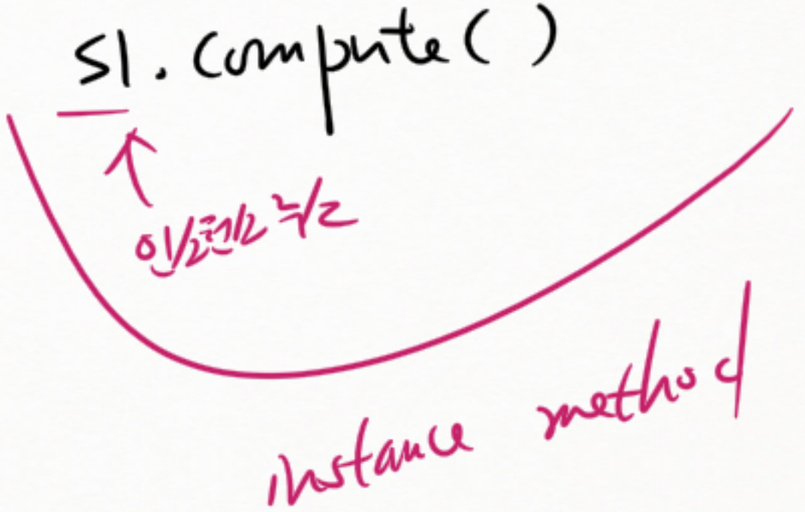
refactoring(코드 깨끗)
↓
. 유지보수 용이성↑
. 속도↓

x GRASP : 클래스의 책임 및 협력을 +



* static 메서드 vs 인스턴스 메서드





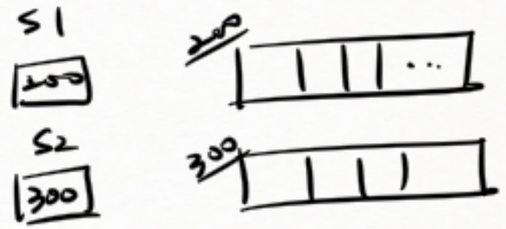
* 인스턴스 메서드 와 연산자

```
i ++ ;
```

operand (피연산자) ↗ ↖ operator (연산자)

```
j ++;
```

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



인스턴스 주소 ↘ ↙ 메서드

```
s1 . compute()
```

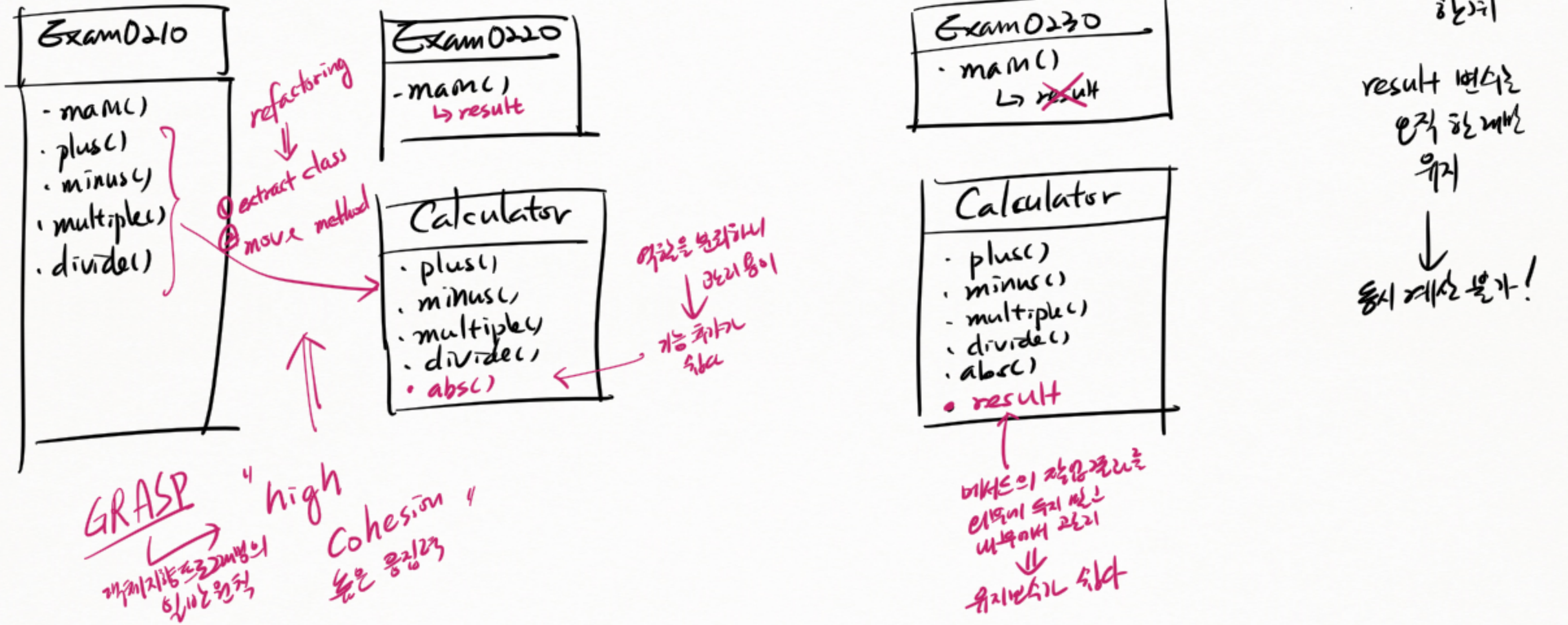
operand ↑ ↑ operator

```
s2 . compute()
```

↑

* 클래스 변경 활용 : 메서드를 위한 분리하기

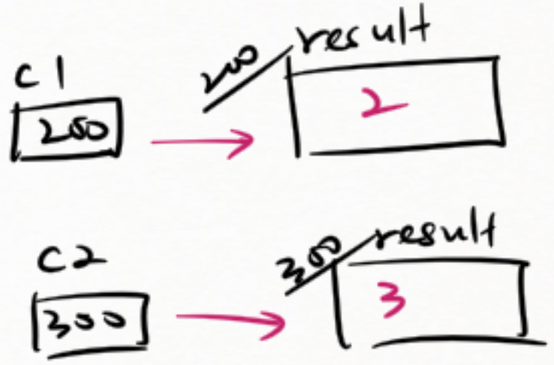
① 분리 전 → ② 메서드 분리 → ③ 클래스 변수 (static 변수 = 클래스 변수) → ④ 클래스 변수의 한계



* 클래스 분리 활용 : 메서드를 묶어 관리하기

→ ① 인스턴스 변수 도입

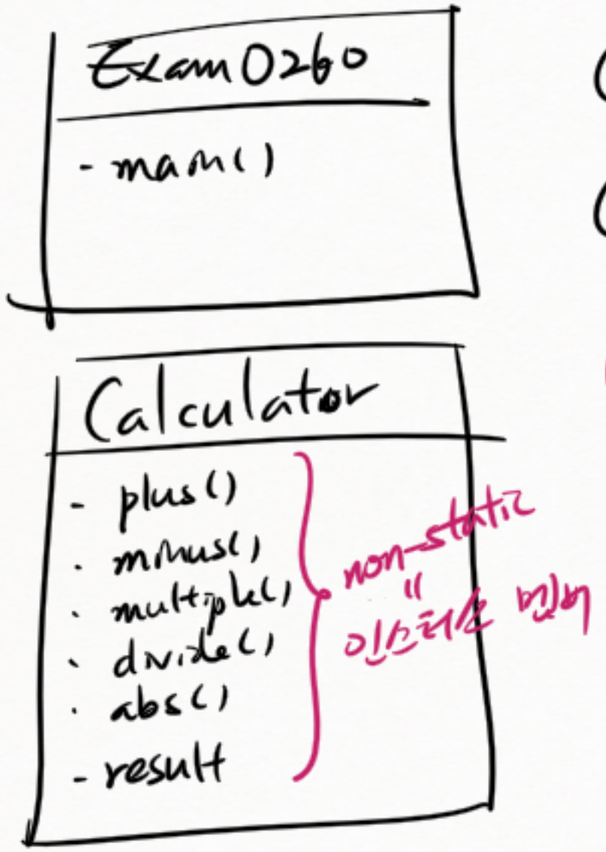

```
Calculator c1 = new Calculator();
Calculator c2 = new Calculator();

Calculator.plus(c1, 2);
Calculator.plus(c2, 3);
:
```



작업대상은 각각의 result 변수가 있는 인스턴스 주소를 알려줄거.

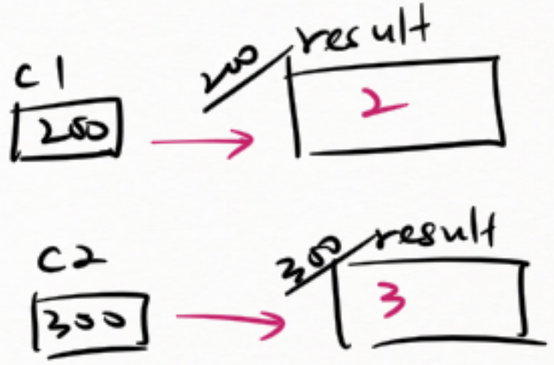
* 클래스 문법 활용 : 메서드를 통해 프로그램

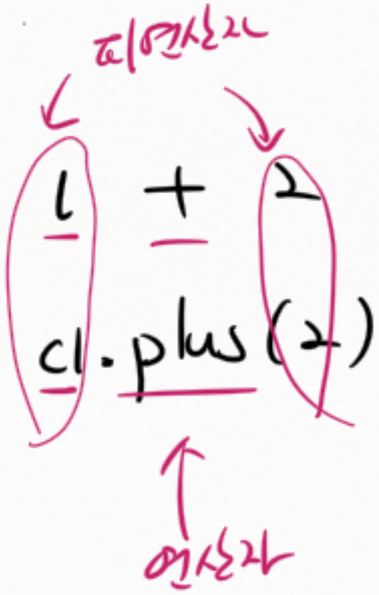
→ ⑥ 인스턴스 메서드 활용




```
Calculator c1 = new Calculator();
Calculator c2 = new Calculator();
c1.plus(2);
c2.plus(3);
```

c2 → 인스턴스 참조값





* 클래스 분배 활용 : 메서드를 묶어 관리하기

→ ① 패키지 없이 클래스

Exam0240
- main()

Calculator
- plus() - minus() - multiple() - divide() - abs() - result

non-static = 인스턴스 변수

com.eomcs.oop.ex02.

Exam0240.java
Calculator.java

→ ② 패키지 분류

com.eomcs.oop.ex02.Exam0240

com.eomcs.oop.ex02.util.Calculator

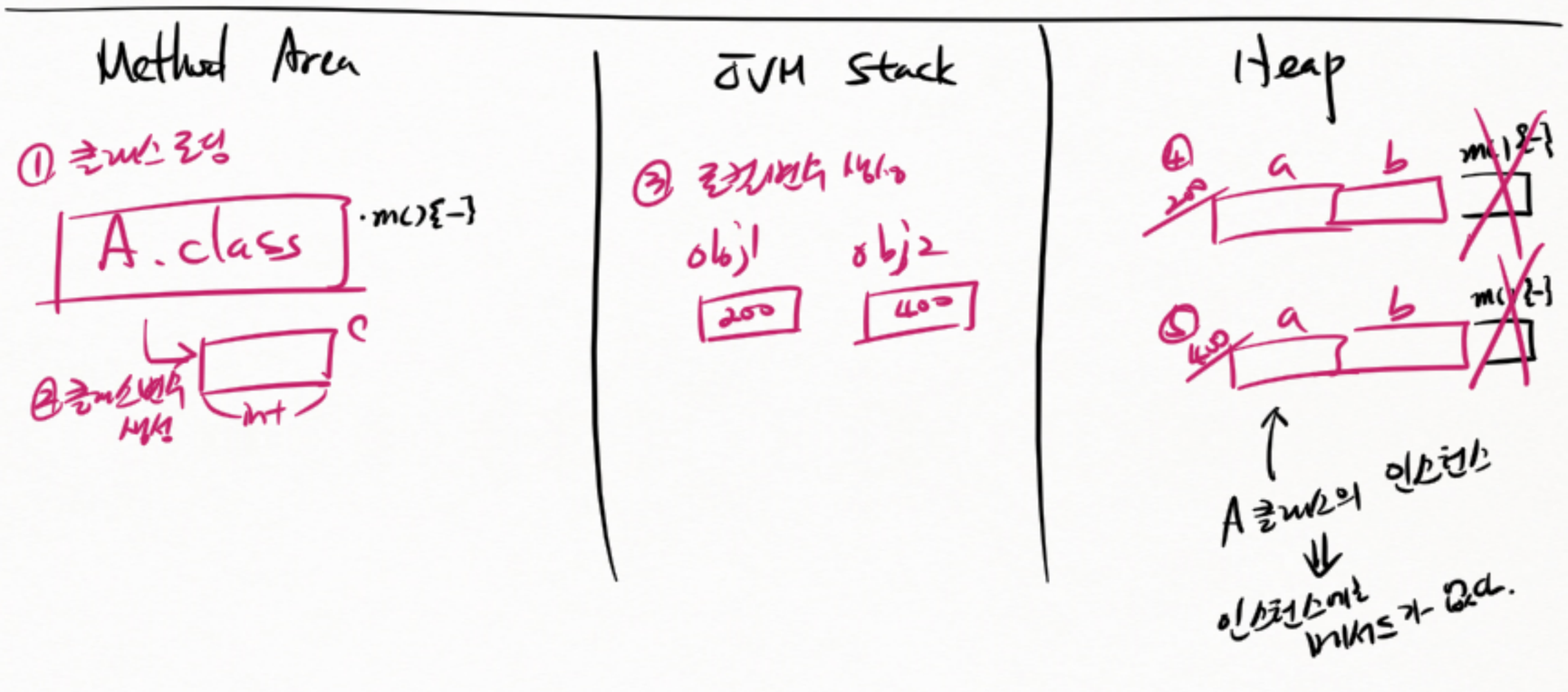
import com.eomcs.oop.ex02.util.Calculator;

클래스를 사용할 때 반드시 선언해 주어야 하는 코드.

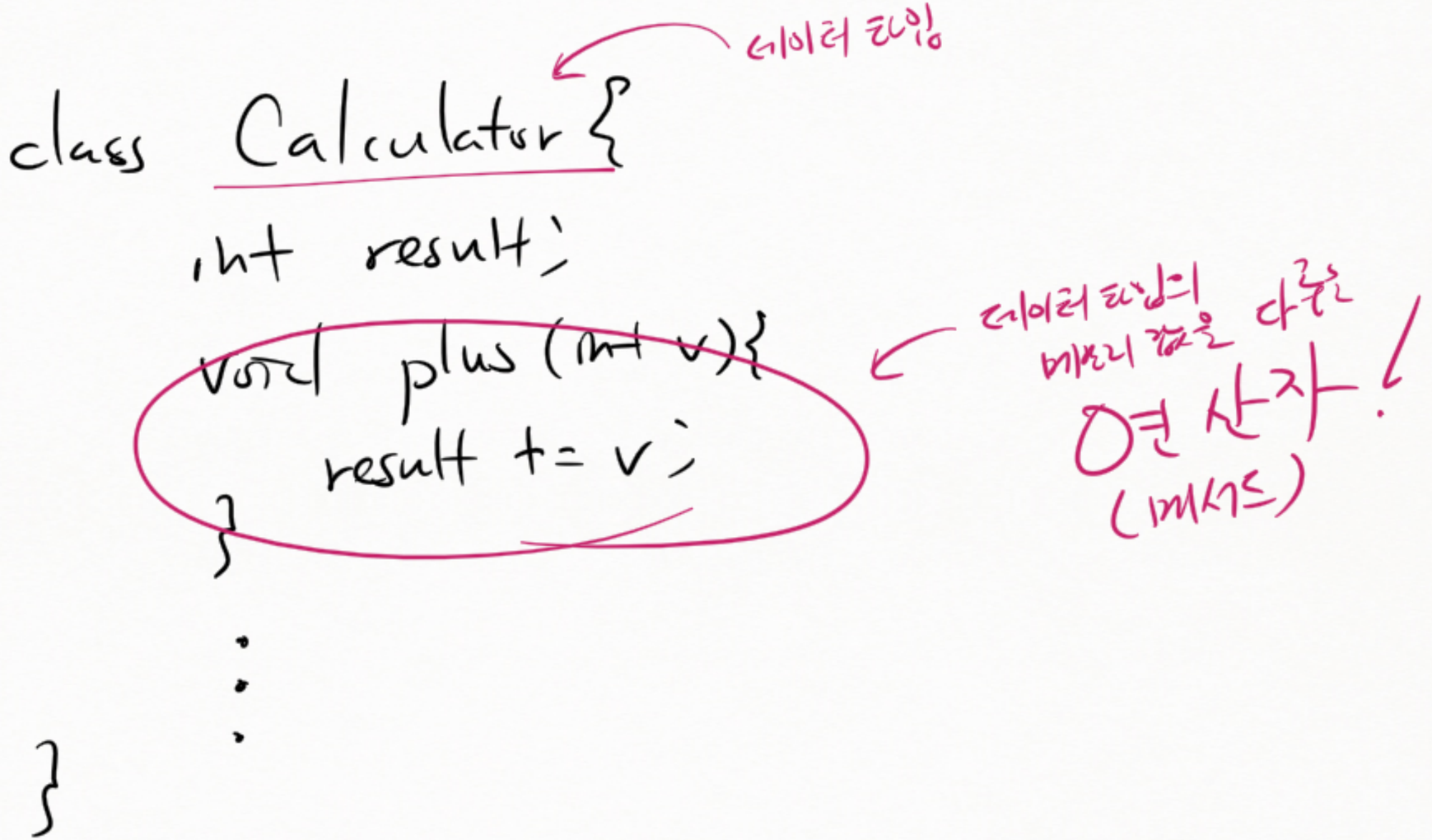
* 클래스 변수 vs 인스턴스 변수

static 변수 / non-static 변수

```
class A {
    int a;      // 인스턴스 변수
    int b;      // 인스턴스 변수
    static int c;   // 클래스 변수
    void m() {}
}

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



* 인스턴스 변수 vs 인스턴스 메서드



* Object 클래스

↳ 자바의 최상위 클래스

java.lang.

Object 자동호출

- String 문자열
- Integer 숫자
- Member 멤버
- Board 게시글

자바의 모든 클래스는 Object의

자손 클래스 이다.

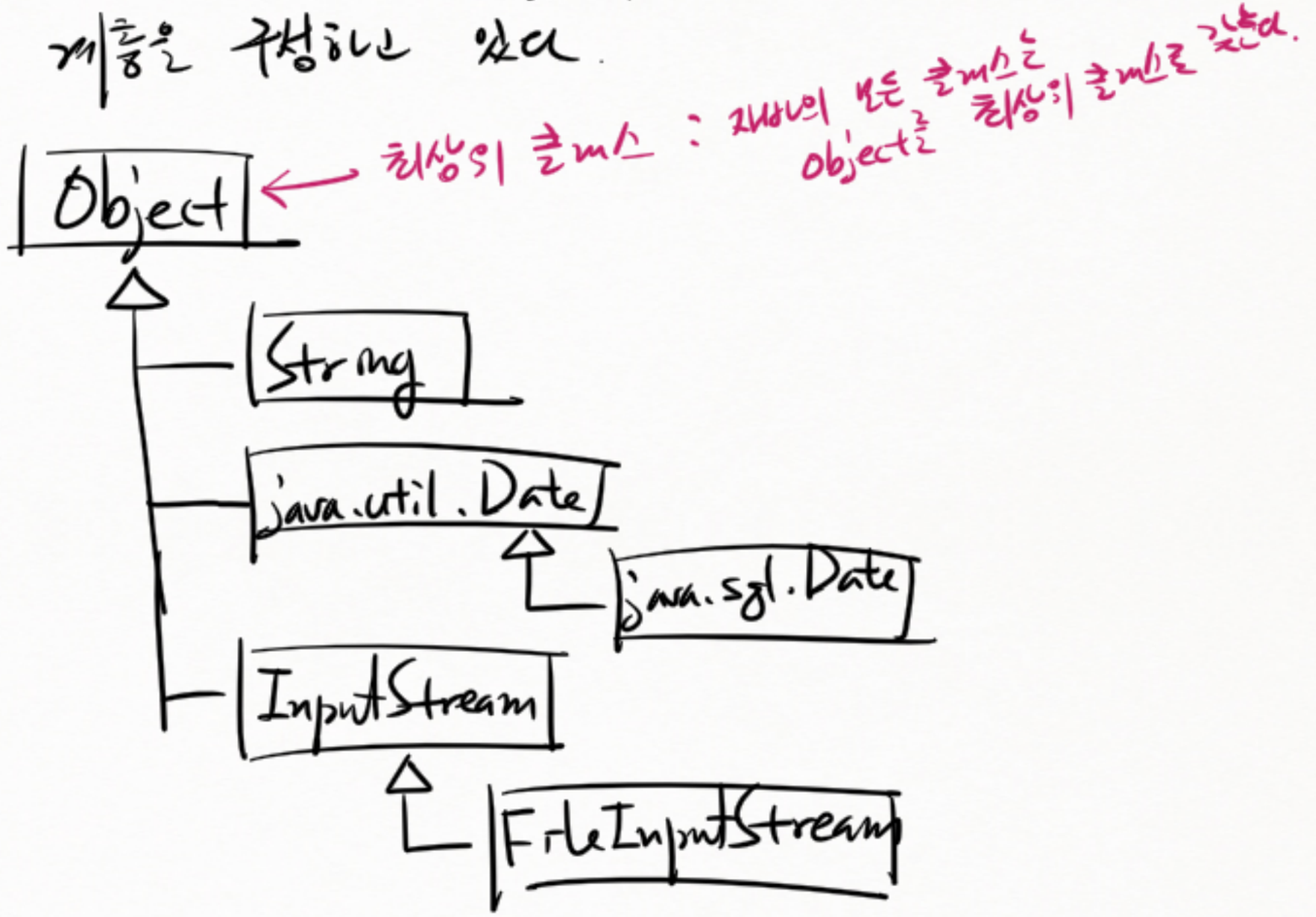
하위(sub)

* 분류 나체와 클래스

분류 계층도

생물
- 식물
  - 현영속
  - 현영속
- 동물
  - 다층류
  - 조류
    - 까마귀
      - 늑대
      - 여우
    - 고양이과
      - 호랑이
      - 사자
      - 퓨마

자바의 클래스도 상위, 하위 계층을 구성하고 있다.



* 상위 클래스와 하위 클래스 : 큰 공통을 위해!

super 클래스 = parent 클래스

Object ← 모든 클래스가 기본으로 가지고 있는 필드와 메서드가 정의되어 있다.

기능을 "상속한다" (inheritance)

sub 클래스 = child 클래스

String ← Object의 필드와 메서드 사용법을 알고 있다

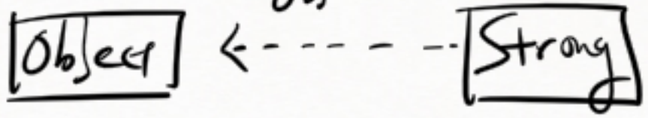
+

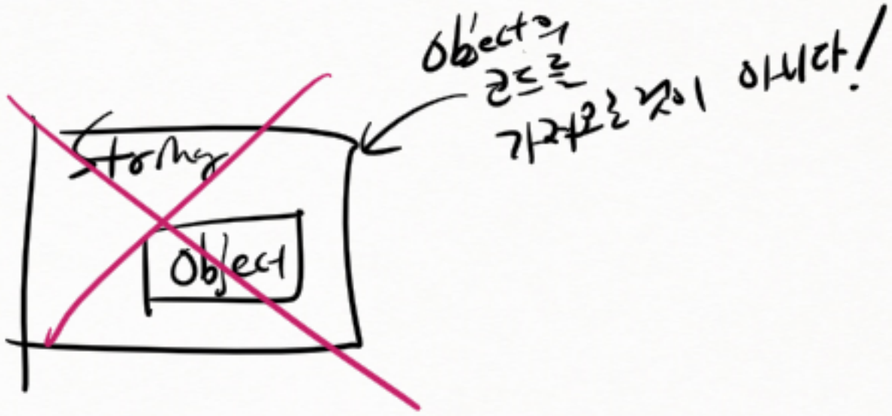
문자열을 다루기 필요한 필드와 메서드가 추가되어 있다

↳ 주의!

- Object의 필드와 메서드를 가지고 있는 것이 아니다.

Object의 기능을 사용할 수 있도록 정해준 것이다.

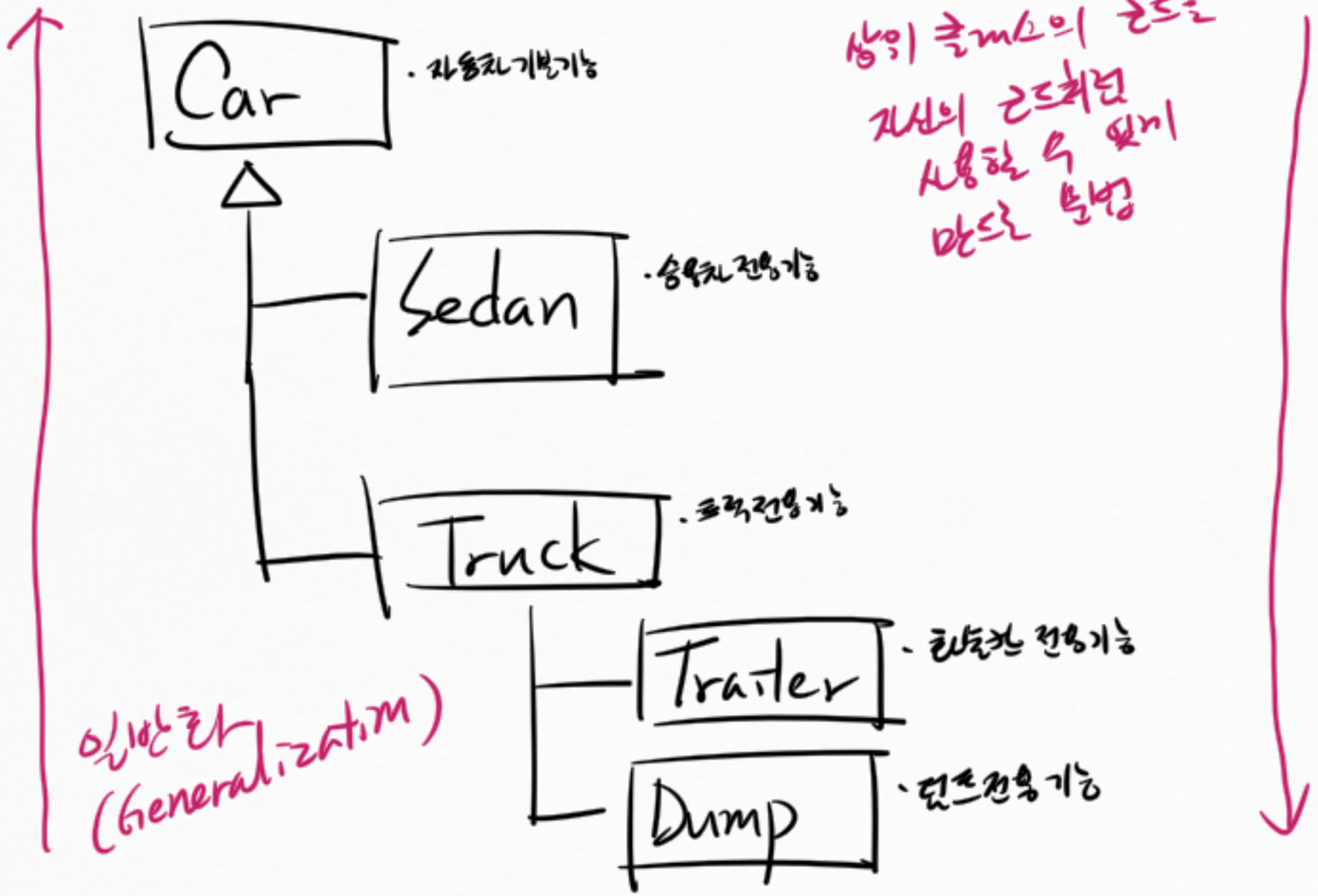




* 상속 : 코드 재사용 방법 중 하나.

↓

상위 클래스의 코드를
자신의 코드처럼
사용할 수 있게
만드는 방법



* 상속 문법과 다형성

- 다형성 변수 ★
- 오버로딩 (overloading)
- 오버라이딩 (overriding) ★

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

↑ 다형성 변수

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

상위 클래스 레퍼런스로
하위 클래스의 인스턴스 주소를
저장할 수 있다

⇓

하위 클래스의 인스턴스를
가리킬 수 있다.

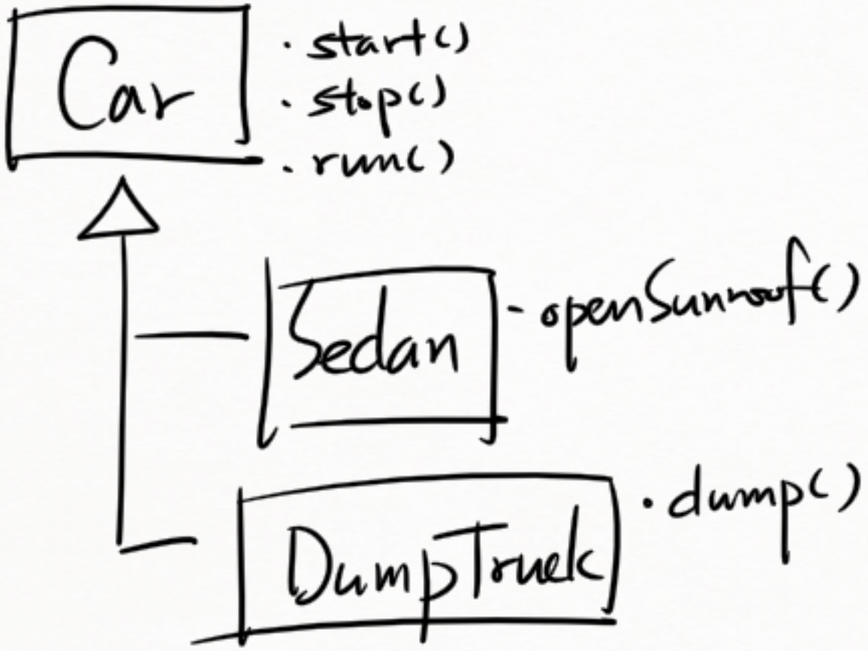
* 클래스 정의 시 부모 클래스 지정하기

```
class Board extends Object {
    ≡
}

class Member {
    ≡
}
```

생략가능!

부모 클래스를 지정하지 않으면 기본으로 Object가 부모 클래스로 설정된다.

* 상속의 주요 사용



Car c;

c = new Car();

c.start()
c.stop()
c.run()

레퍼런스 타입의 클래스 기능은 본인 것만 있다.

c = new Sedan();

c.start()
c.stop()
c.run()
c.openSunroof()

선제로 c가 Sedan을 가리키고 있더라도 메서드 사용은 c 변수의 타입이 한정된다.

타입 = 클래스 (primitive type)

```
Sedan s;

s = new Sedan();

s.start()
s.stop()
s.run()
s.openSunroof()
```

수퍼 클래스의 기능을 사용할 수 있다.

~~s = new Car()~~

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

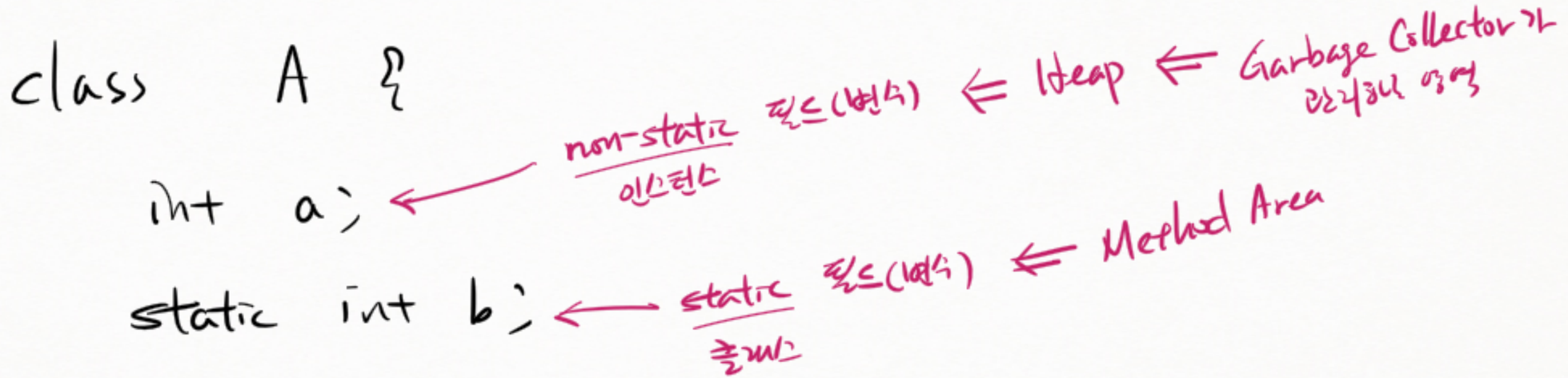
~~s.openSunroof();~~

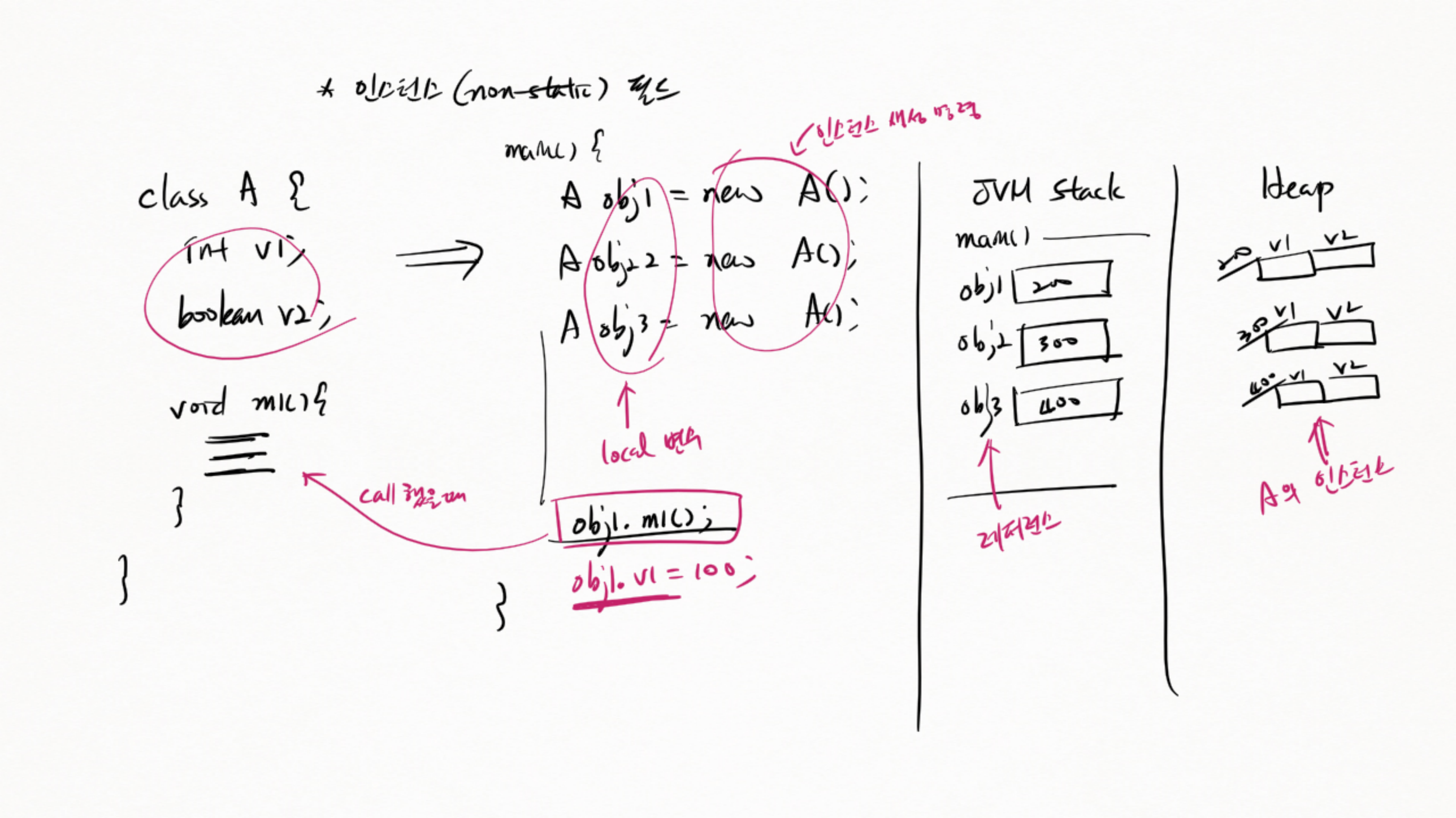
s의 타입은 Sedan이다 Sedan의 기능을 사용하려 할 것이다. 그런데 s가 Car를 가리키고 있으므로 실행 오류가 발생할 것이다.

이런 잘못된 사용을 미연에 방지하려고 하위 레퍼런스로 상위 객체를 가리키지 못하게 문법으로 막고 있다.

* static 필드와 non-static 필드

```
class A {
    int a;
    static int b;
}
```



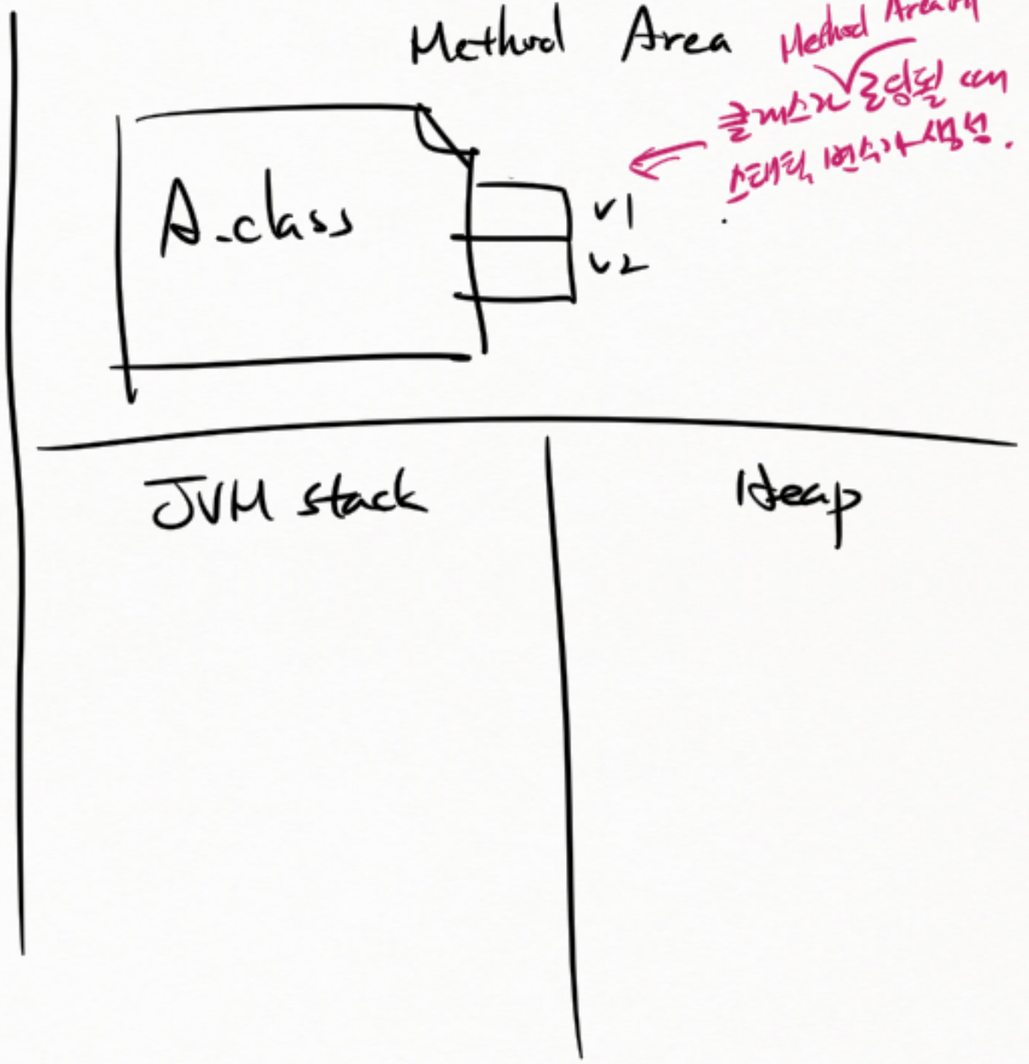

* 인스턴스 (non-static) 필드
class A {
int v1;
boolean v2;
void m1(){
}
}
main() {
A obj1 = new A();
A obj2 = new A();
A obj3 = new A();
인스턴스 생성 방법
local 변수
obj1.m1();
call 했을때
obj1.v1 = 100;
}
JVM Stack
main()
obj1
obj2
obj3
200
300
400
레퍼런스
Heap
v1
v2
200
300
400
A의 인스턴스

* 클래스(static) 필드(변수)

```
class A {
    static int v1;
    static boolean v2;
}
```

```
main(){
    A.v1 = 100;
    A.v2 = true;
}
```

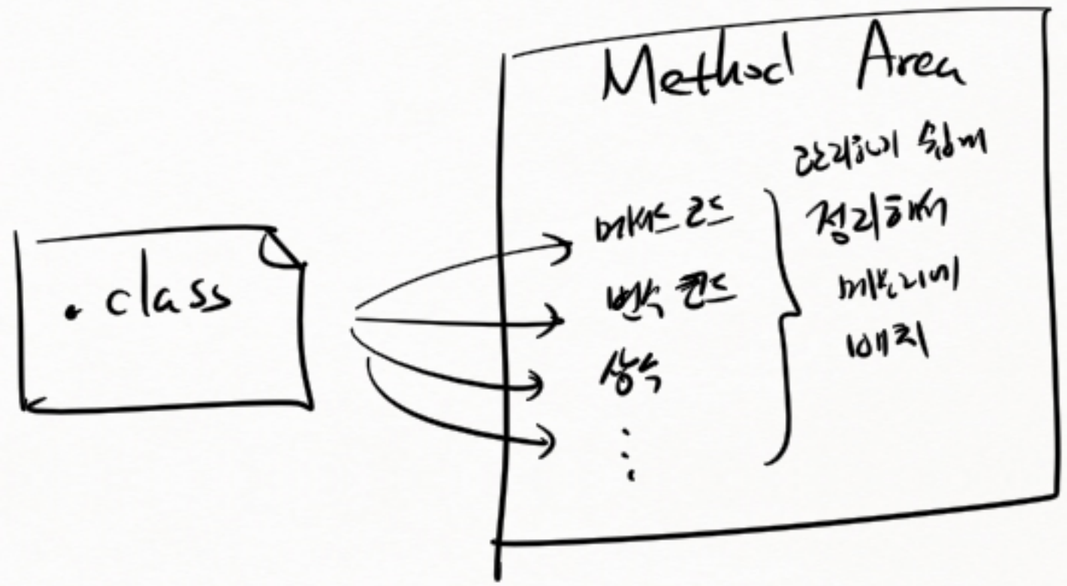
스태틱 변수는 클래스 이름으로 접근



* 클래스 로딩과 실행

$ java Hello ↵

① Hello.class 찾는다
② Bytecode 검증
③ Method Area에 로딩
④ 스태틱 필드 생성
⑤ 스태틱 블록 실행
☆ ⑥ main() 호출



* 클래스 로딩, 스태틱 변수 및 인스턴스 변수 생성

```
class ExamOl4o {
    class A {
        static int v1;
        int v2;
    }
    main(){
        A.v1 = 100;
        A p = new A();
        p.v2 = 200;
        A p2 = new A();
        p2.v2 = 300;
    }
}
```

컴파일

ExamOl4o.class

ExamOl4o$A.class

Method Area

① ExamOl4o.class ← ② main() 로딩

④ A.class — 100 v1 ← ⑤ 스태틱 변수 생성

⑥-1

Jvm Stack

main()

p 111 args

p2 222

③ main()의 로컬변수 생성

Heap

111 v2 200 ⑧ 인스턴스 생성

⑪

222 v2 300 ⑬ 인스턴스 생성

⑯

* 인스턴스 변수와 인스턴스 메서드가 놓이는 위치

Method Area

Exam0230.class

main(){

=

}

Calculator.class

result : int

plus() {——}

minus() {——}

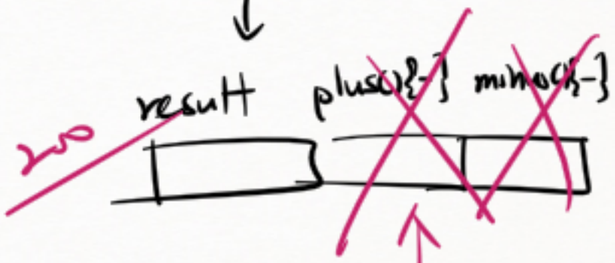
JVM Stack

main()

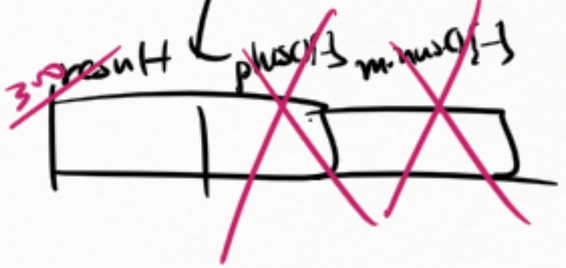
args c1 c2

Heap

new Calculator()

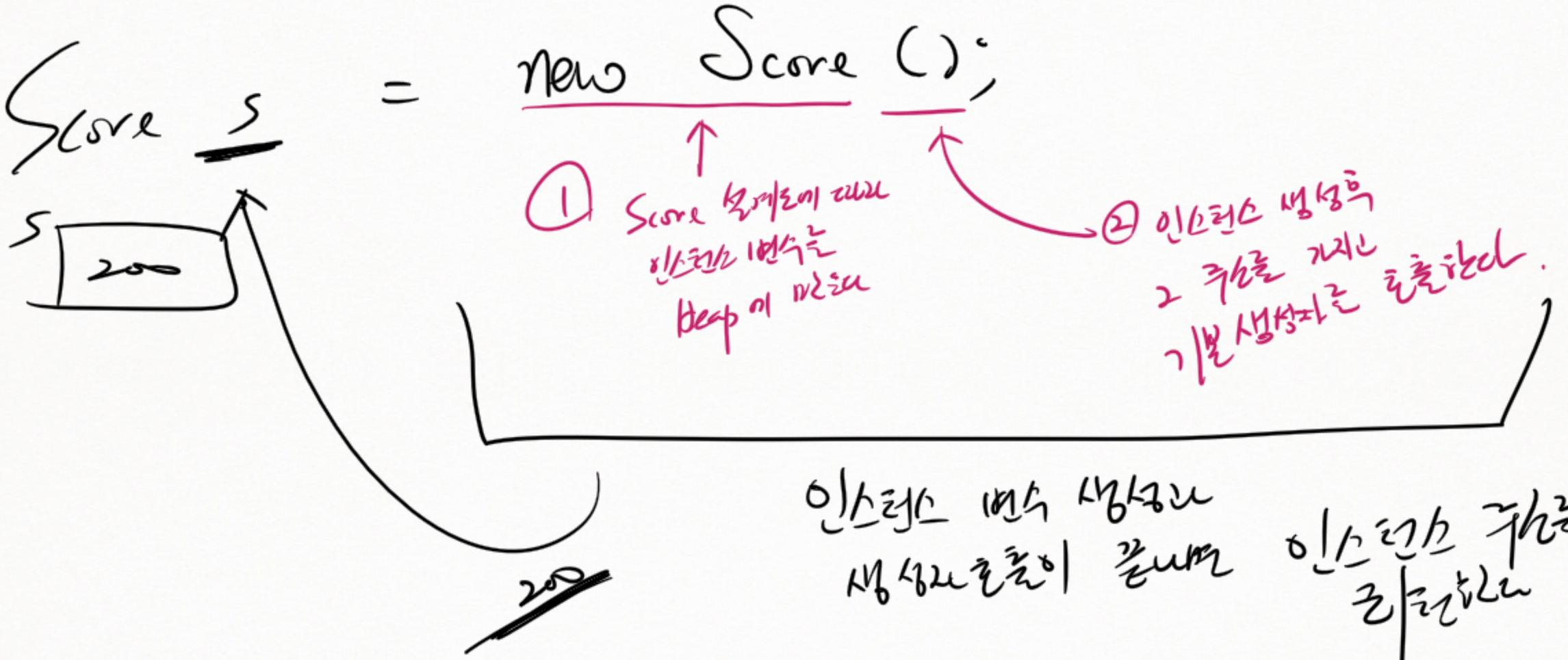


new Calculator()



인스턴스를 생성한 때 메서드도 생성되긴 하지만 실제 메서드를 Method Area에 올린다.

* 인스턴스 생성과 사용법



인스턴스 변수 생성과
생성자호출이 끝나면 인스턴스 주소를
리턴한다

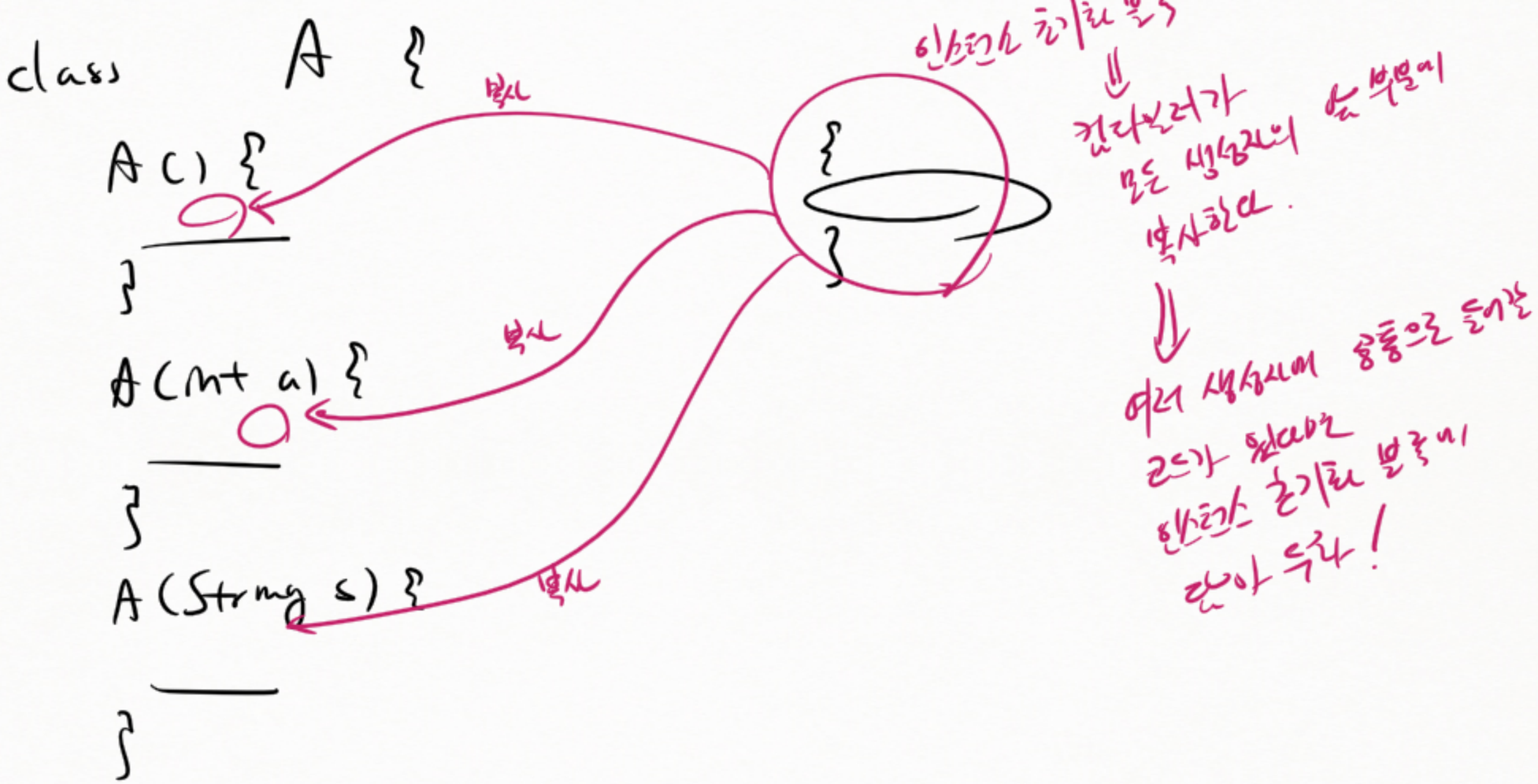
⑥ A.static{}

⑨ B.static{}

3

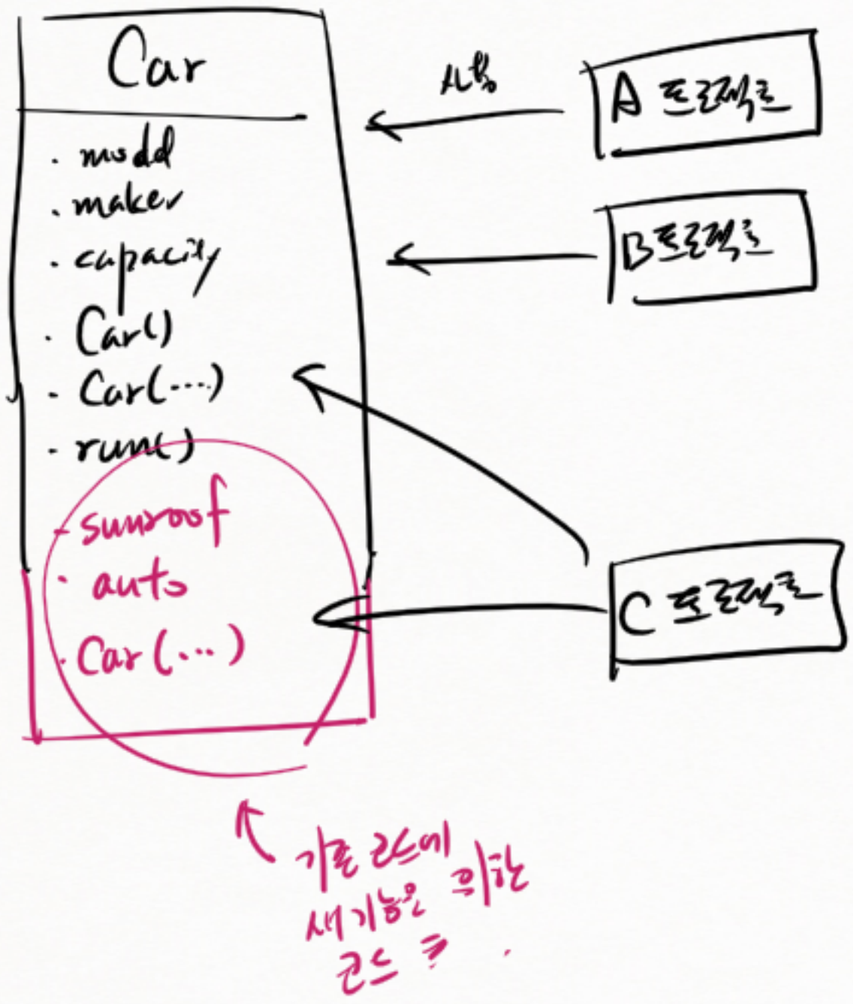
2

* 인스턴스 초기화 (instance initializer)



* 기능을 확장하는 방법

* 기능을 확장하는 방법 — ① 클래스 변경

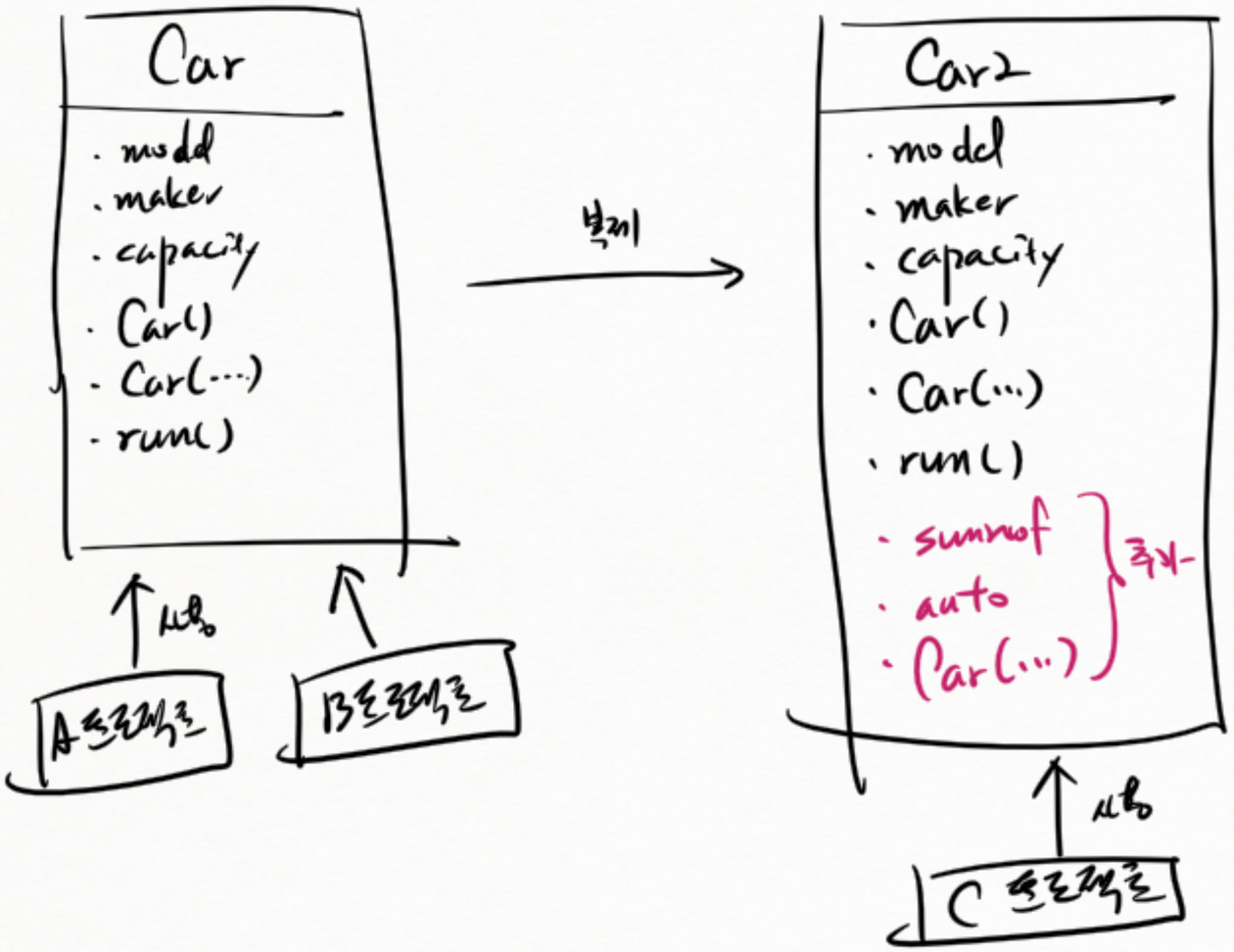


문제점

① 코드에 계속 기능을 덧붙이다 보면 유지보수가 힘들다

★ ② 기존 코드를 사용하는 프로젝트에 영향을 끼친다.
A와 B 프로젝트

* 기능을 확장하는 방법 — ② 기존 소스를 복제해서 기능 확장



특징

① 기존 코드를 손대지 않는다

↓

기존 프로젝트에 영향을 끼치지 않는다.

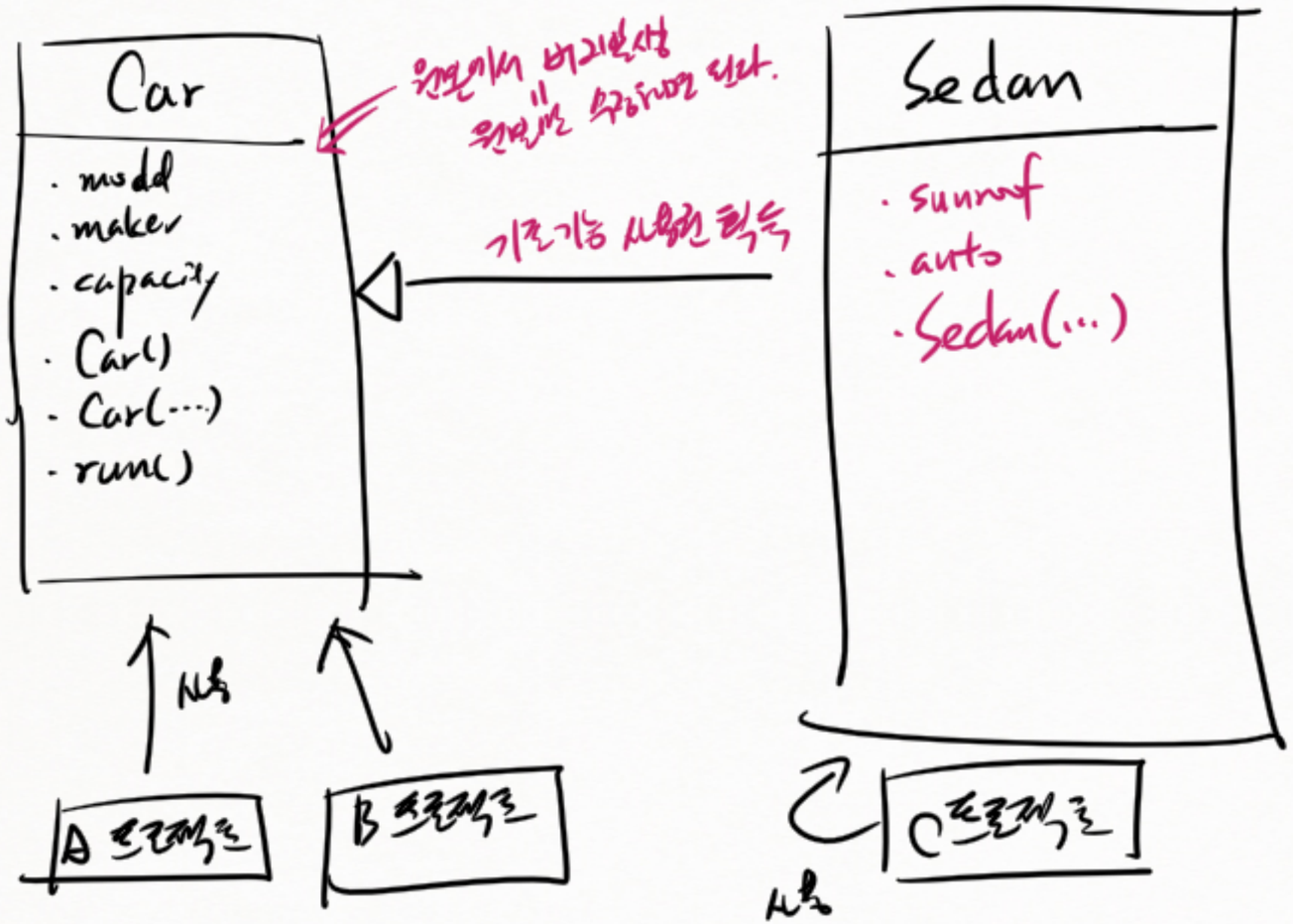
단점

① 복제 ⟶ 중복 코드 생성

↓

(기능추가시 ⟶ 전체에 적용
버그수정시 ⟶ 〃
〃 유지보수가 힘들다. 〃

* 기능을 확장하는 방법 — ③ 상속을 이용



특징

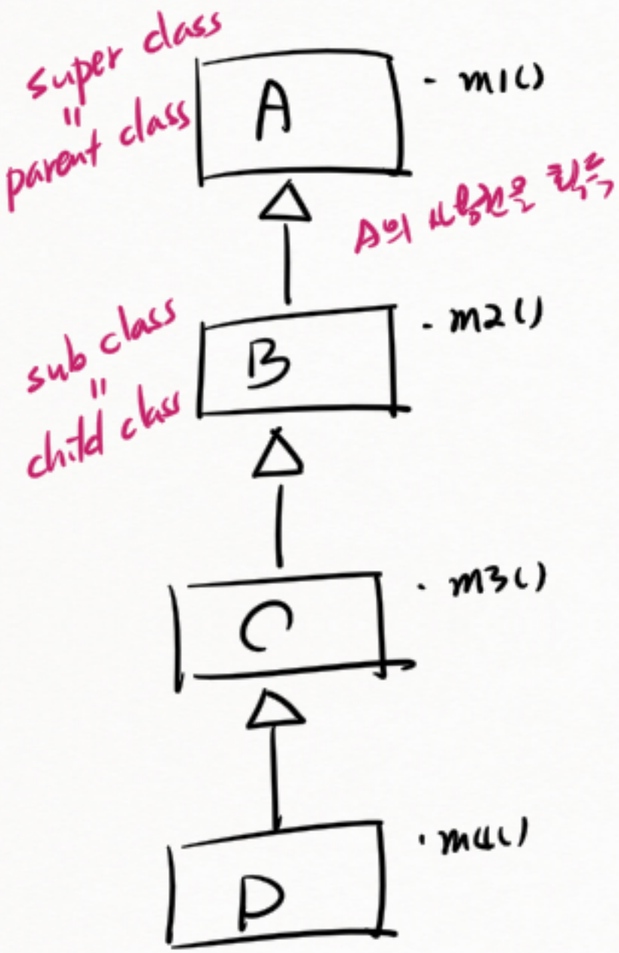
- 기존코드를 손대지 않는다.
  ↓
  이전 프로젝트의 영향을 끼치지 않는다.
- 기존 코드를 재사용 → 비용절감
  → 버그 확률 감소 ↓

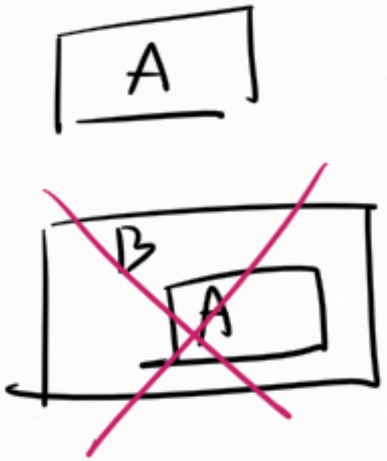
코드 중복을 없앤다.
↳ 버그 수정이 용이

한계

- 여러 단계를 거치면서
  점점 많은 기능을 감당해 상속받고 있다

* 상속과 메서드 호출



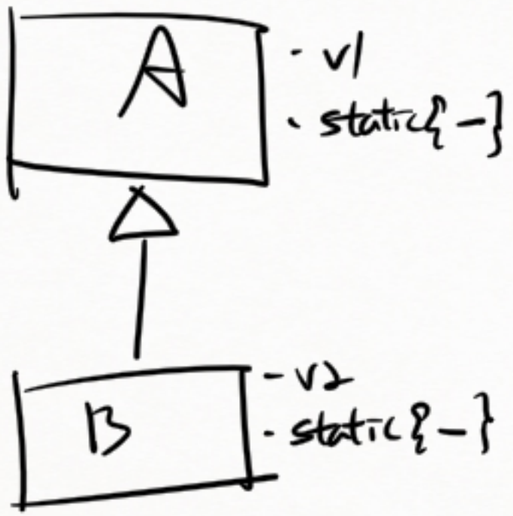


A의 존재 그대로 포함하지 않다!

```
B obj = new B();
obj.m2(); //ok
obj.m1(); ← B클래스를 통해 A클래스의 멤버를 사용할 수 있다
```

↑ A의 m1()을 호출

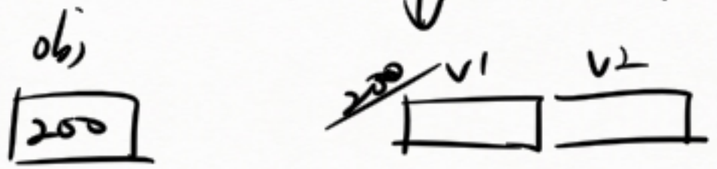
* 정적 vs 인스턴스 필드(변수)



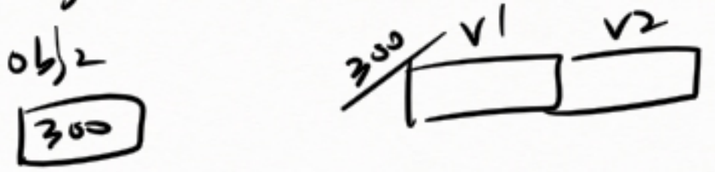
B obj = new B();

* 클래스 로딩
↓
상위 클래스 부터 로딩

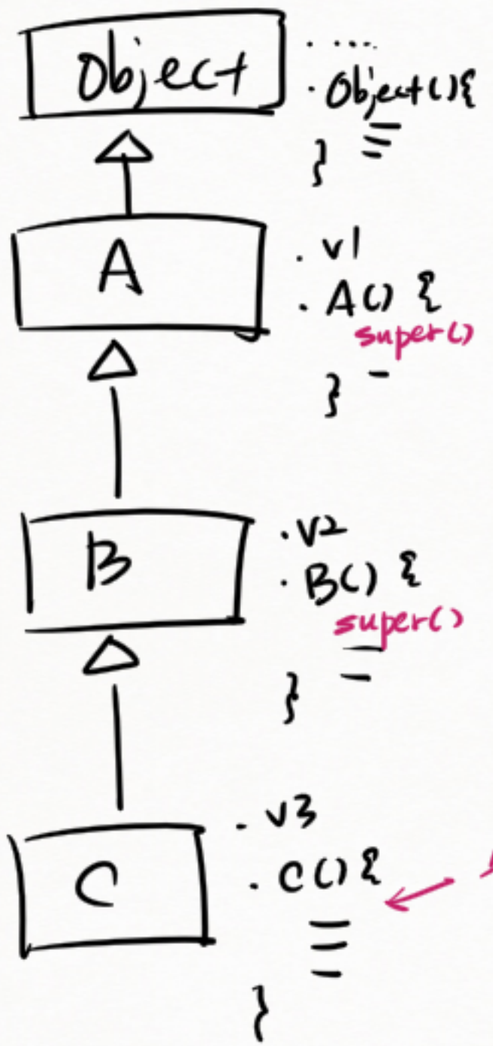
*인스턴스 필드 생성
↳ super 클래스의 인스턴스 변수부터 생성



B obj2 = new B();



* 생성자 호출 순서



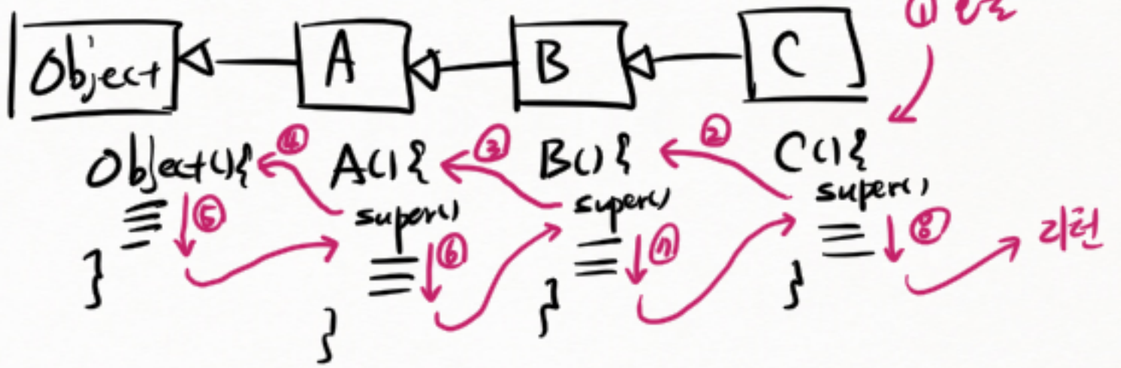
생성자의 "첫 문장은" 반드시 부모클래스의 생성자 호출 문장이어야 한다
ex) super()

또는 같은 클래스의 다른 생성자를 호출하는 문장이어야 한다
ex) this()

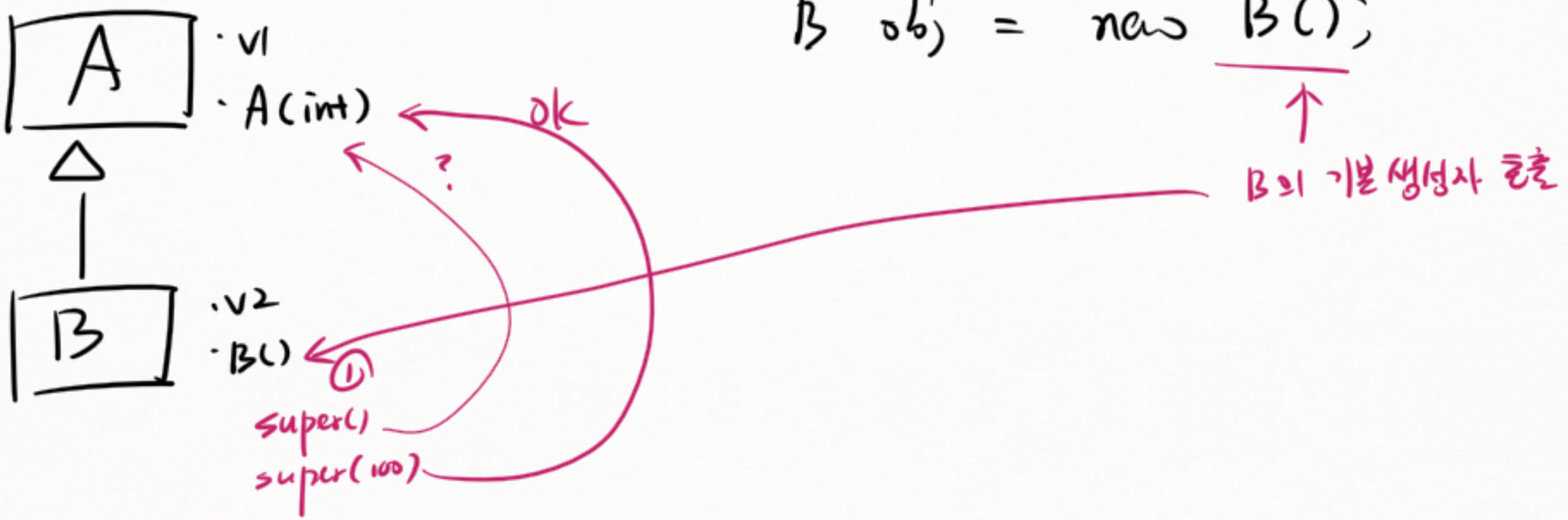
⇐ 둘 다 명시적 호출할 수 없다
ex) super() 또는 this()
this() super()
↖ 첫 문장이 아니다!

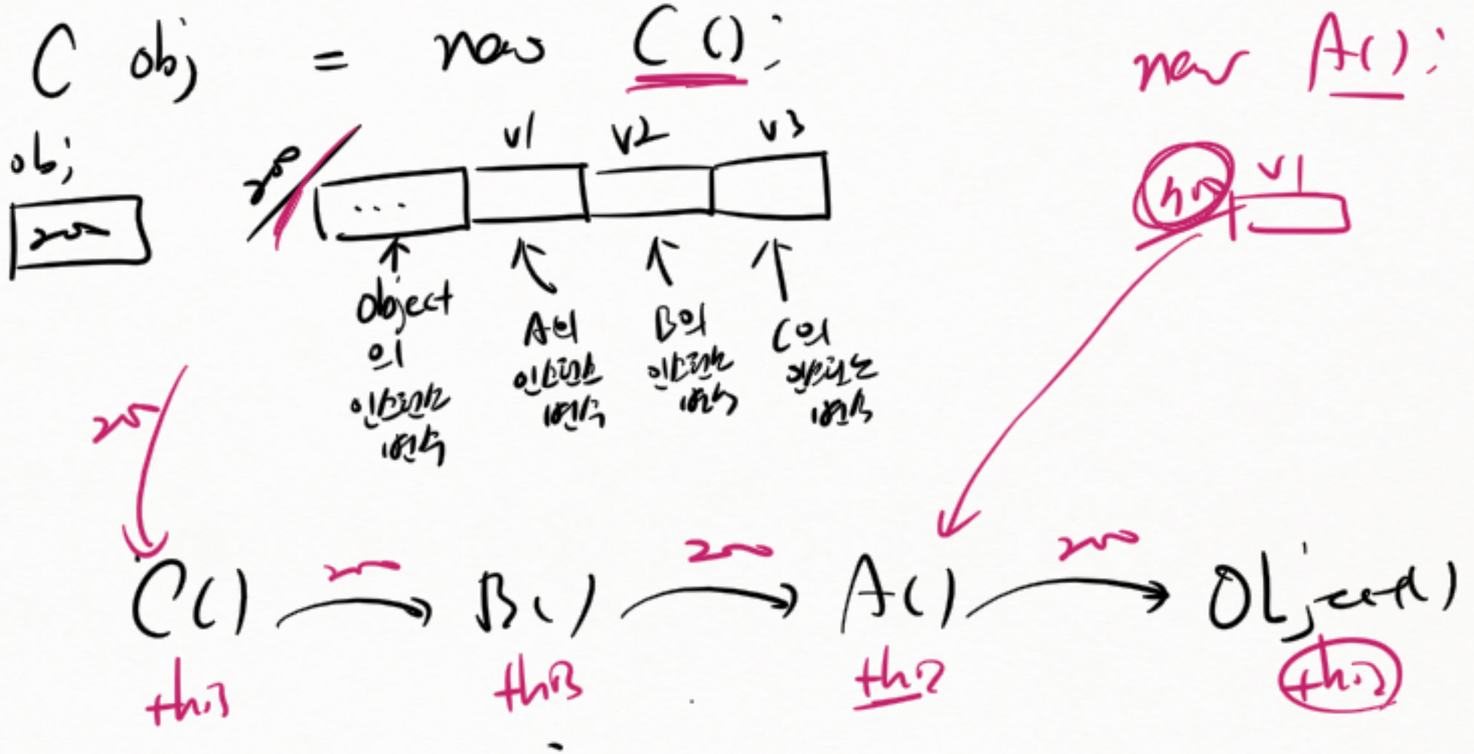
부모클래스의 생성자 호출을 생략하면 컴파일러가 자동으로 추가한다.

C obj = new C();
↳ C의 생성자 호출

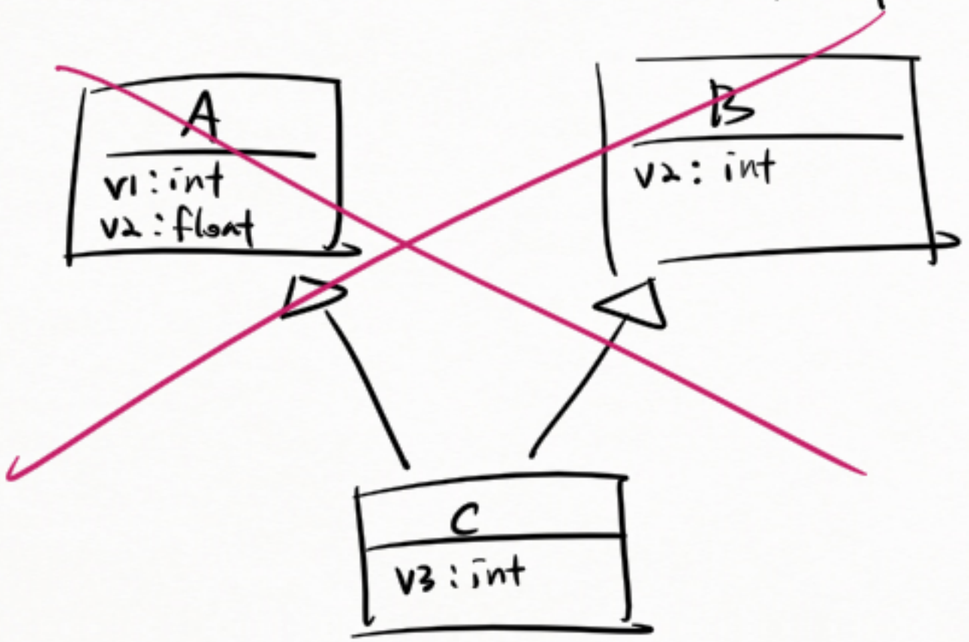


* 생성자 호출 2

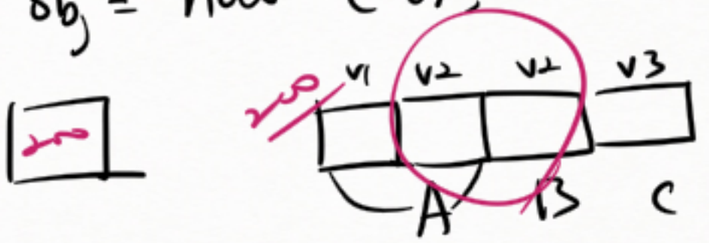



C obj = new C();
obj
new
v1
v2
v3
Object의 인스턴스 변수
A의 인스턴스 변수
B의 인스턴스 변수
C의 인스턴스 변수
new A();
new
v1
new
C()
this
B()
this
A()
this
Object()
this

* 다중 상속 ⇒ 지원 안함 !



C obj = new C();



obj.v2 = 0;

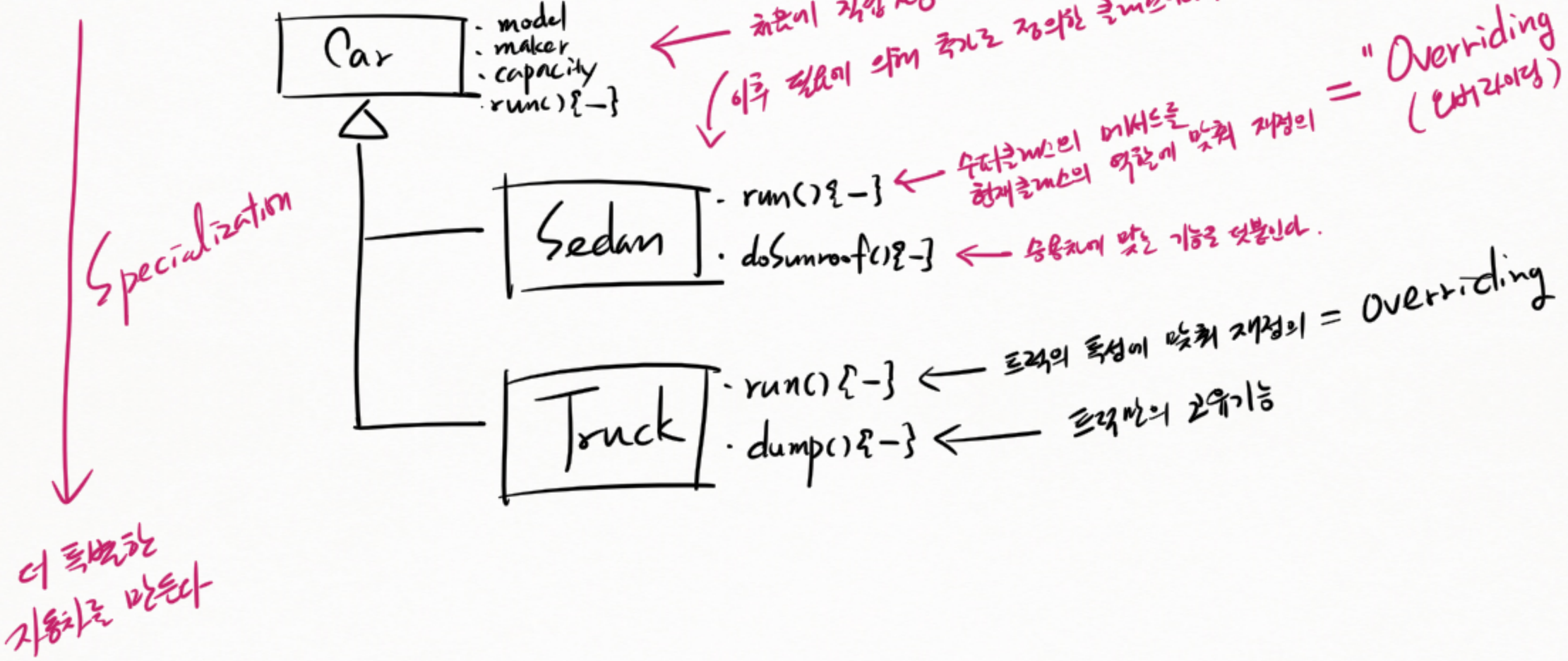
A의 변수인지
B의 변수인지
구별할 수 없다.

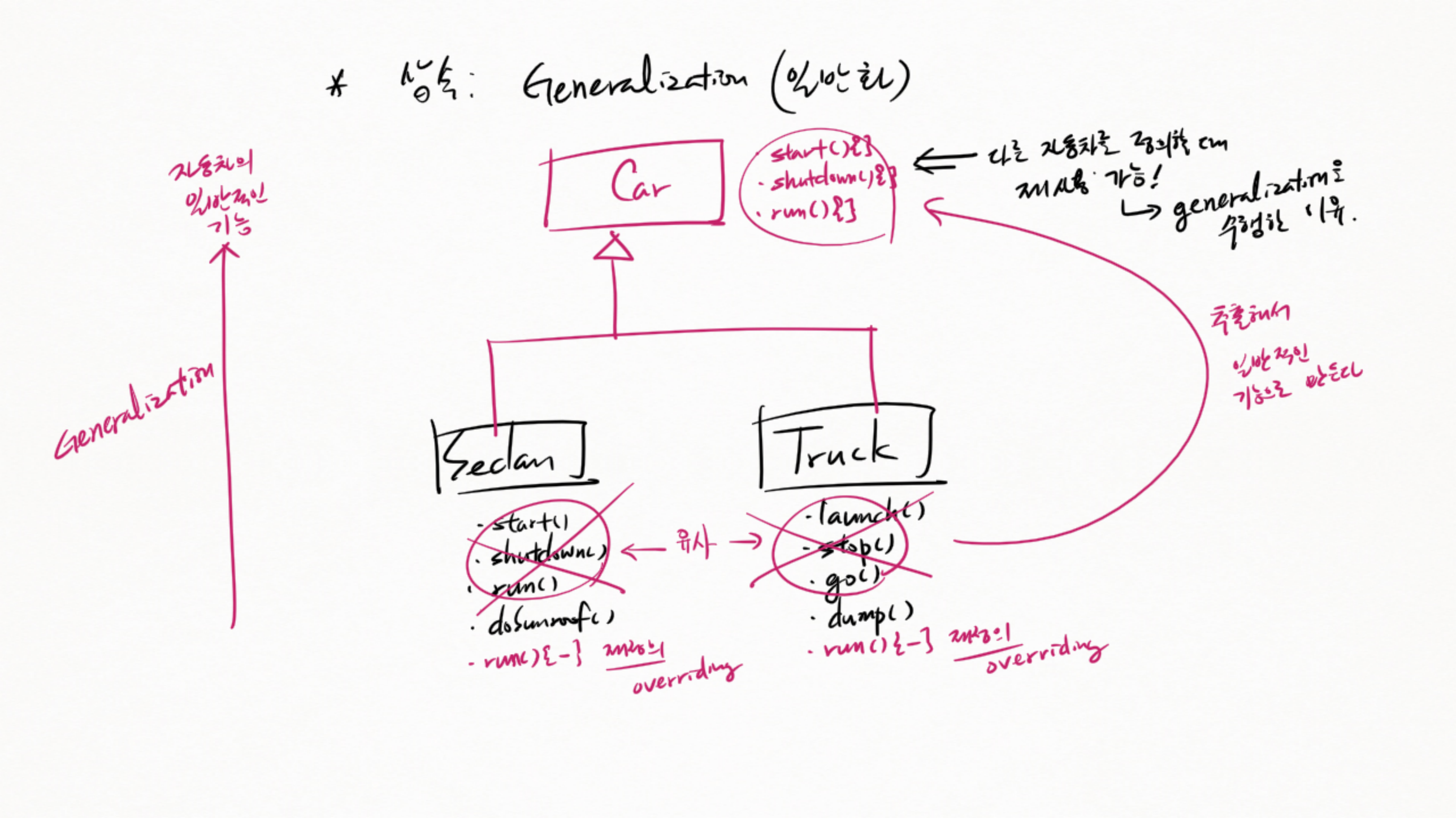
↑ 구별하려면 클래스 명을 추가

↑ 언어가 복잡

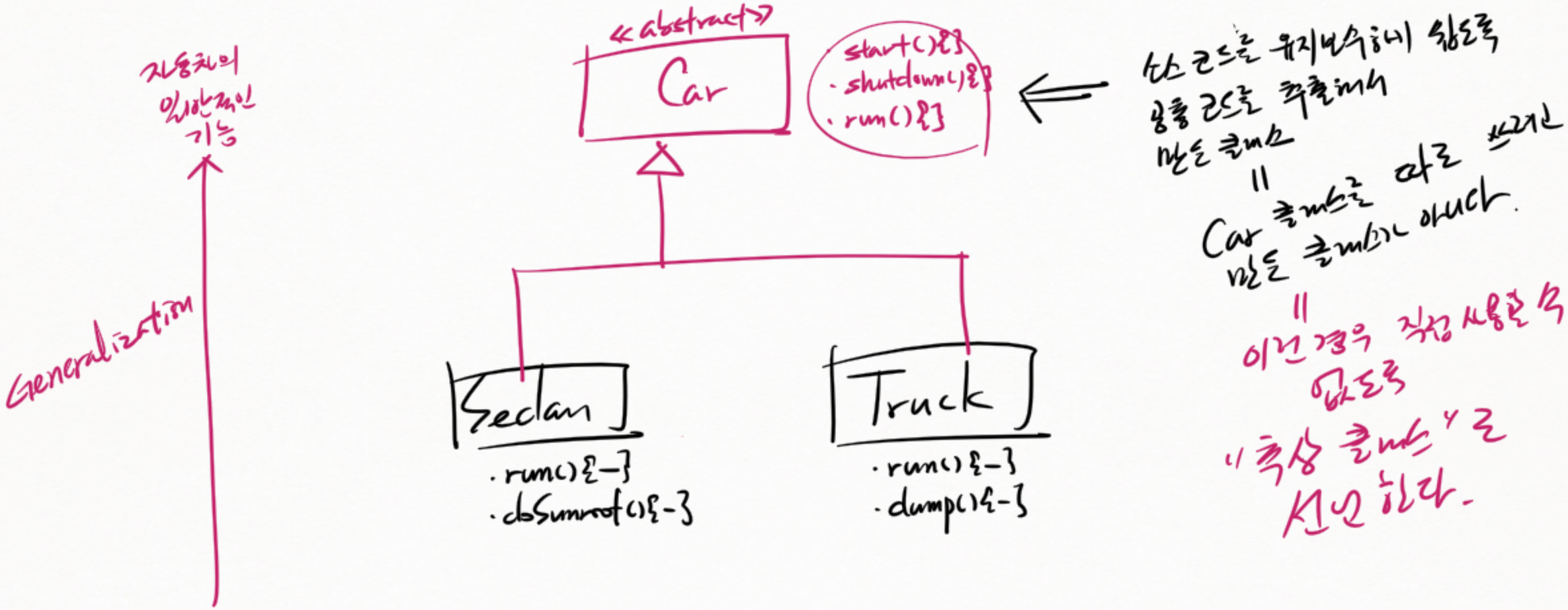
그래서 자바는 다중 상속을 허락하지 않는다

* 상속 : specialization (전문화)

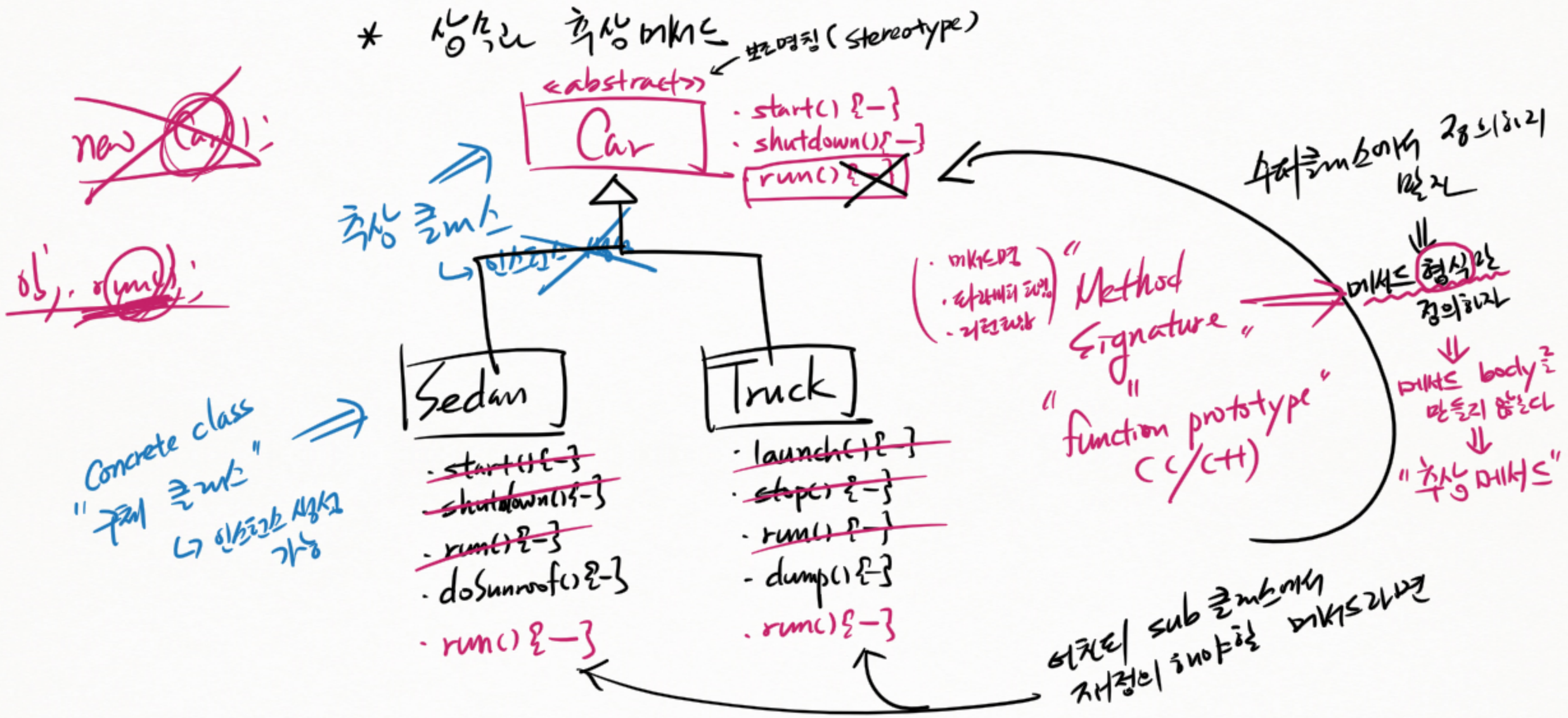



* 개념: Generalization (일반화)
자동차의 일반적인 기능
Generalization
Car
start(){}
shutdown(){}
run(){}
다른 자동차를 정의할 때 재사용 가능!
↳ generalization을 수행한 이유.
추출해서 일반적인 기능으로 만든다
Sedan
start()
shutdown()
run()
doSunroof()
run(){-} 재정의 overriding
유사
Truck
launch()
stop()
go()
dump()
run(){-} 재정의 overriding

* 상속과 추상 클래스

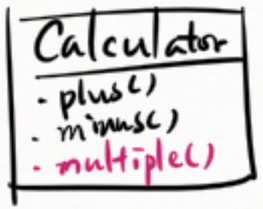


소스코드를 유지보수하기 쉽도록 공통 코드를 추출하여 만든 클래스
=
Car 클래스를 따로 쓰려고 만든 클래스가 아니다.
=
이런 경우 직접 사용할 목적으로 "추상 클래스"를 선언한다.

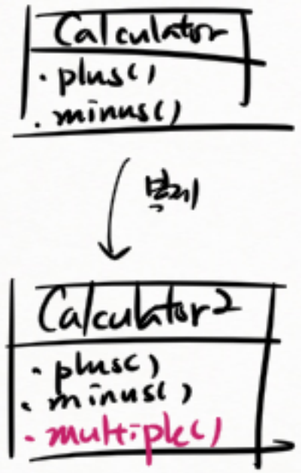

* 생략된 추상 메서드
보조명칭 (stereotype)
«abstract»
Car
· start() {-}
· shutdown() {-}
run() {-}
new Car();
이 · run();
추상 클래스
인스턴스 생성
Concrete class
"구체 클래스"
↳ 인스턴스 생성 가능
Sedan
· start() {-}
· shutdown() {-}
· run() {-}
· doSunroof() {-}
· run() {-}
Truck
· launch() {-}
· stop() {-}
· run() {-}
· dump() {-}
· run() {-}
(· 메서드명
· 파라메터 리스트
· 리턴자료형)
"Method Signature"
"function prototype"
(C/C++)
추상클래스의 경우 정의하지 않고
메서드 형식만 정의하고
메서드 body는 만들지 않는다
"추상 메서드"
하위 클래스에서 재정의 해야할 메서드라면

* 기능 확장법 cheat sheet

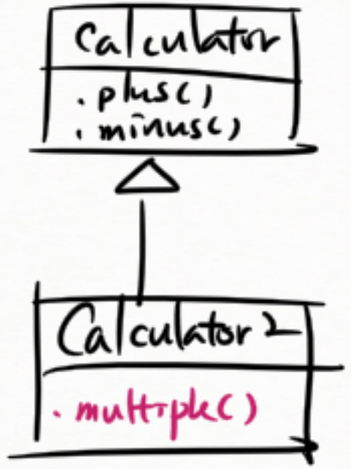
① 기존코드 변경



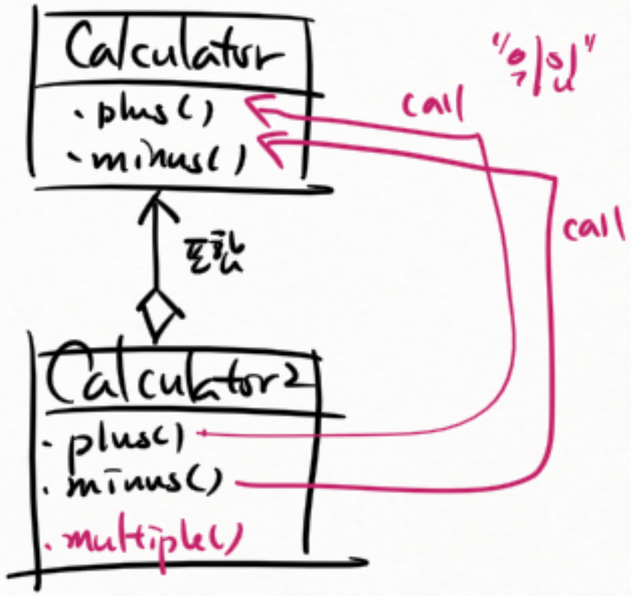
② 코드복제후 기능 추가



③ 상속

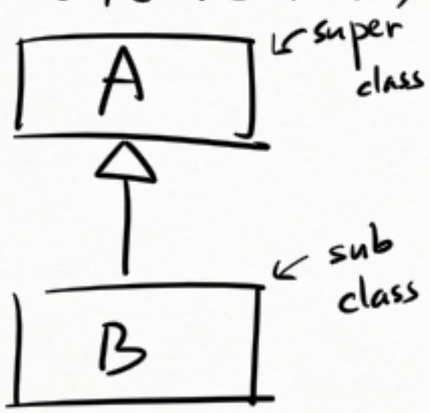


④ 위임



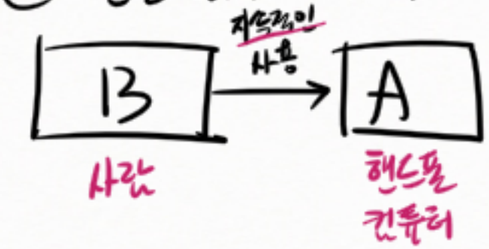
* 클래스 관계 cheat sheet

① 상속 (inheritance)




```
class B extends A{
    ≡
}
```

② 연관 (association)




```
class B {
    A obj;
    ≡
}
```

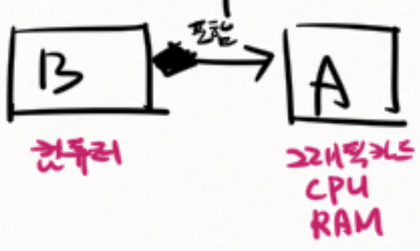
③ 집합 (aggregation)


```
class B {
    A obj;
    ≡
}
```

컨트롤러 ≠ 키보드 마우스 모니터

Lifecycle

④ 합성 (composition)

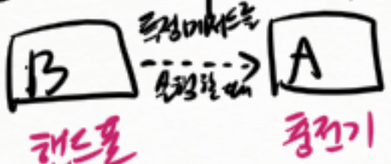



```
class B {
    A obj;
    ≡
}
```

컨트롤러 = 그래픽카드 RAM cpu

Lifecycle

⑤ 의존 (dependency)




```
class B {
    void m(A obj){
        ≡
    }
}
```