class

* 클래스 분류의 목적

① 비즈니스 분류 : MemberHandler
Prompt

② 데이터 타입을 정의 : Score, Member

★ 새 데이터 타입 정의

① 클래스 정의

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

"인스턴스 변수 (field)"

인스턴스를 만들 때 생성된 ‖ new 명령을 실행할 때 생성된.

② 레퍼런스 선언

```
Score obj;
```

"reference"

↑ Score의 인스턴스 주소를 저장할 변수

③ 인스턴스 생성

```
obj = new Score();
```

obj → 200

new Score(); → 생성되는 변수들을 Heap 영역에 만듦.

200 name | kor | eng | math | sum | aver

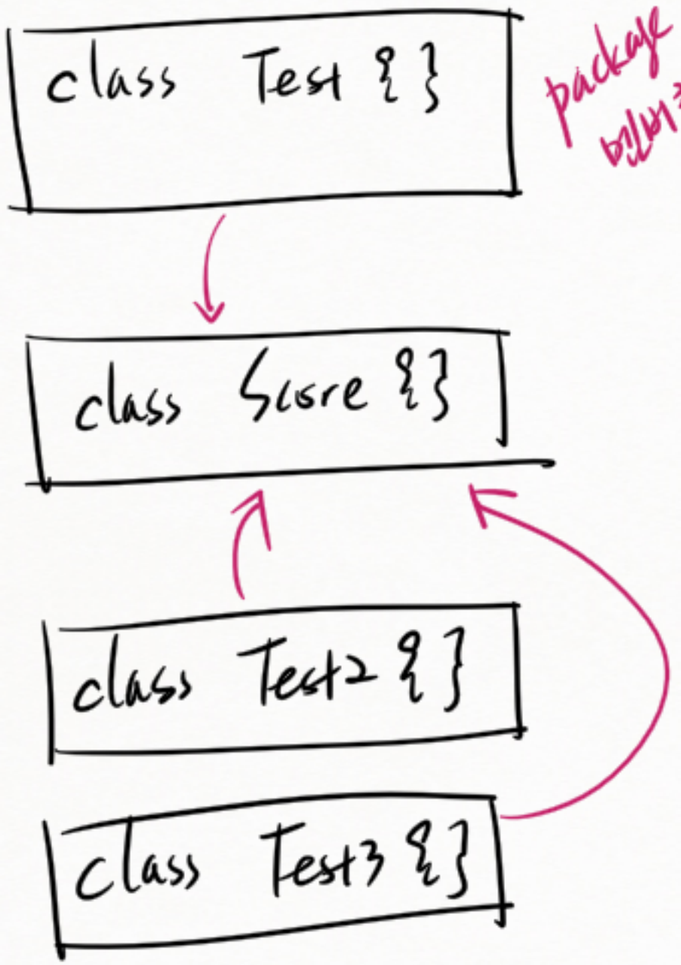
인스턴스

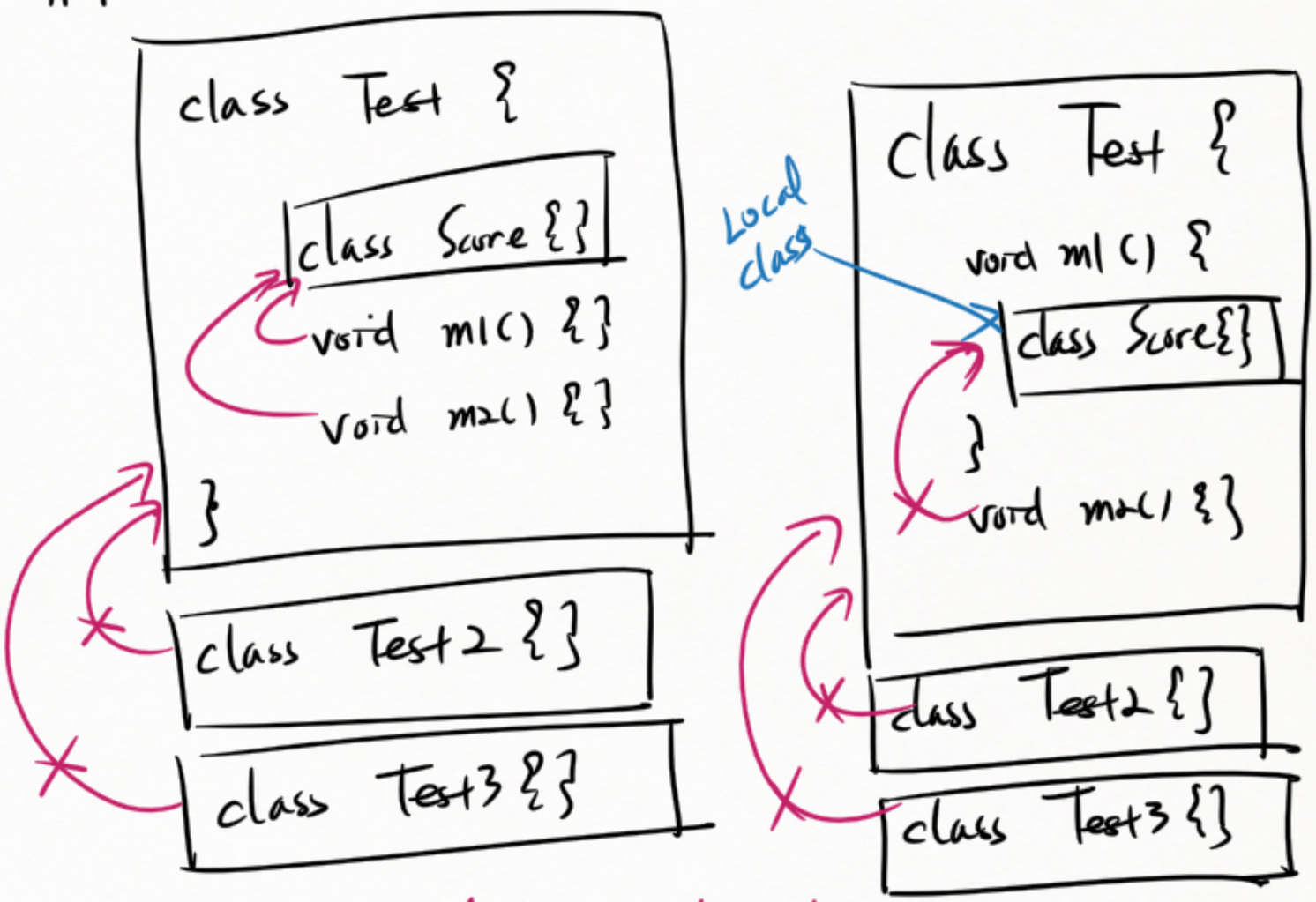
"Score의 인스턴스" instance

④ 인스턴스에 값 저장

```
obj.name = "홍길동";
```

* 클래스 정의의 위치

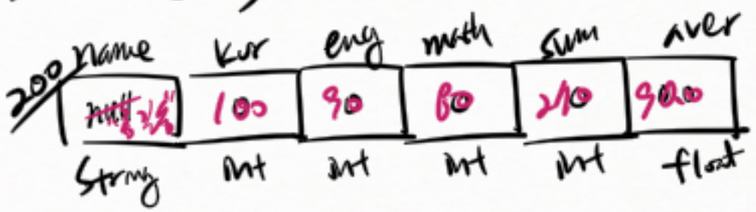




"Nested class"

* 인스턴스 생성, 레퍼런스, call by reference

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




```
s.name = "홍길동";
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 값

```
printScore(Score s) {
    System.out.printf();
}
```

200

* 메서드에서 인스턴스 생성 후 리턴

```
Score s = createScore("홍길동", 100, 100, 100);
printScore(s);
```

s → 200

```
createScore(String name, int kor, int eng, int math) {
    Score s = new Score();
    s.name = name;
    s.kor = kor;
    s.eng = eng;
    s.math = math;
    return s;
}
```

s.sum = ;

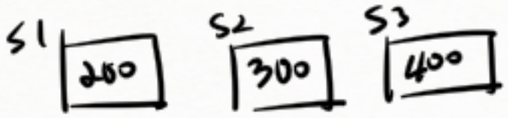
s.aver = ;

s → 200

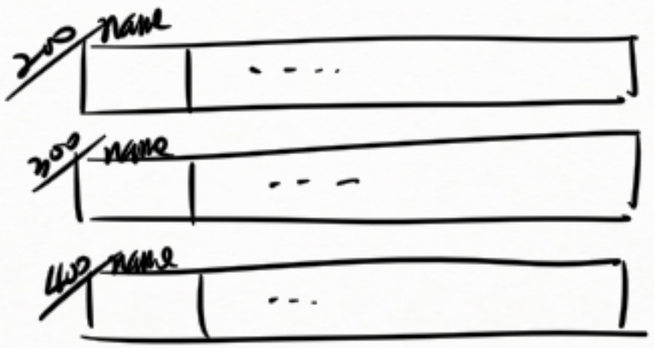
200	name	kor	eng	math	sum	aver
	홍길동	100	100	100	300	100.0

* 레퍼런스 배열 사용 전

```
Score s1, s2, s3;
```




```
s1 = new Score();
s2 = new Score();
s3 = new Score();
```



* 레퍼런스 배열 사용 후

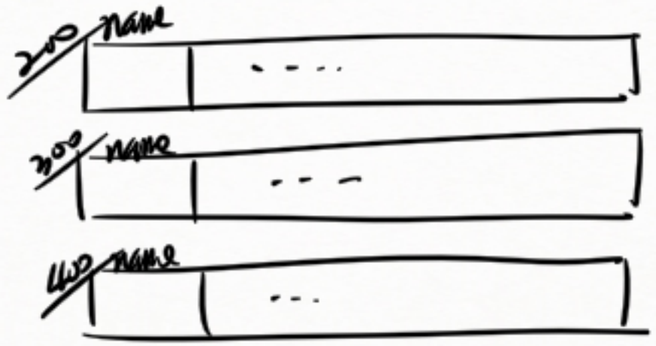
```
Score[] scores = new Score[3];
```

배열레퍼런스 — 레퍼런스들의 배열

scores

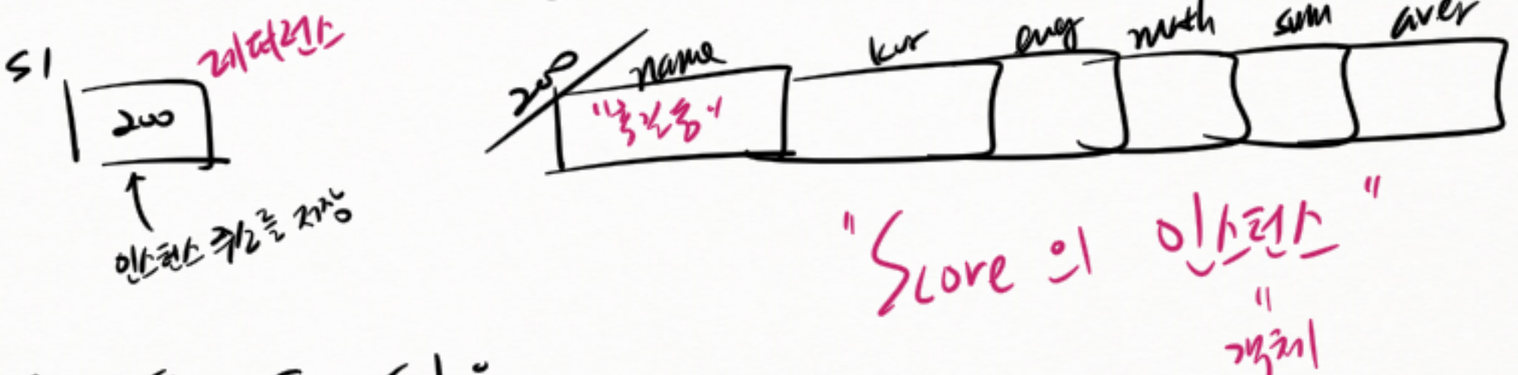
1700


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



*인스턴스 와 레퍼런스

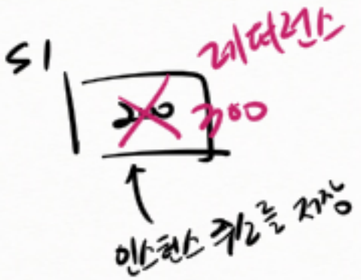
Score s1 = new Score();

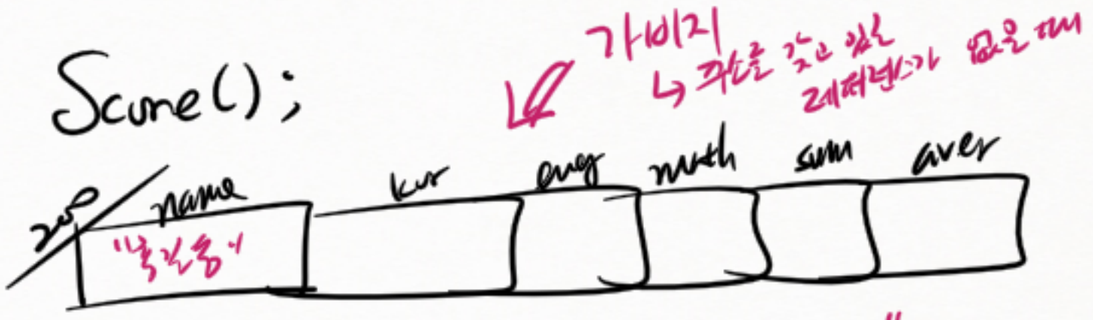


Score s2 = s1;

* 가비지(Garbage)

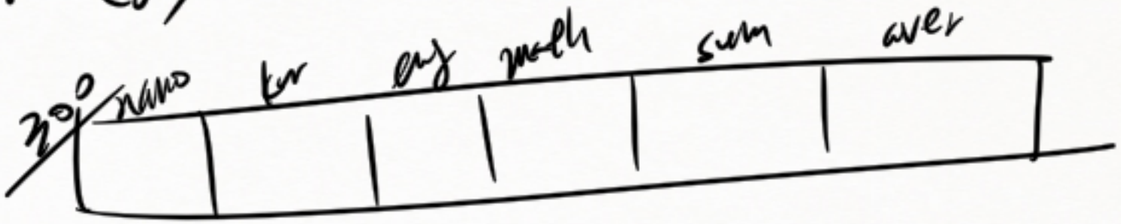
Score s1 = new Score();





"Score의 인스턴스"
"
객체

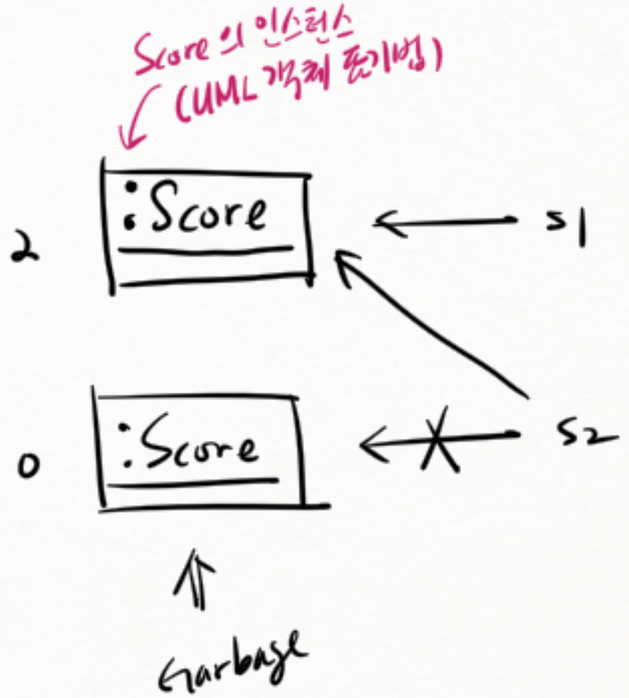
s1 = new Score();



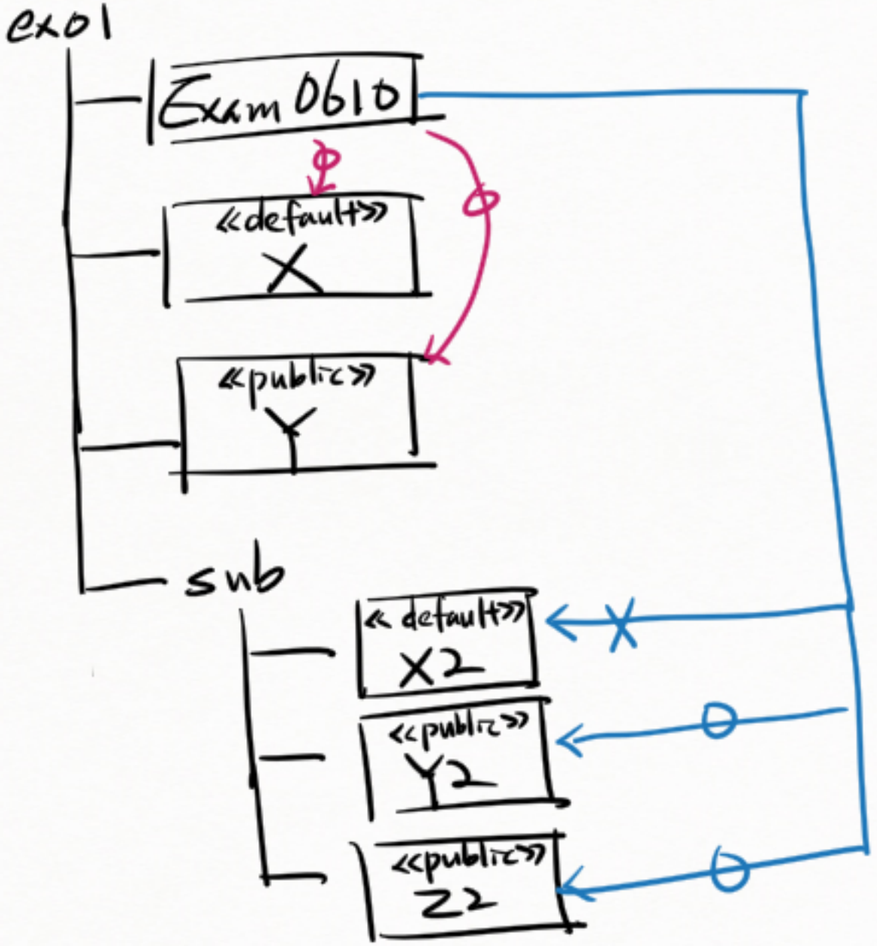
* 인스턴스 레퍼런스 카운트

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

s2 = s1;
```

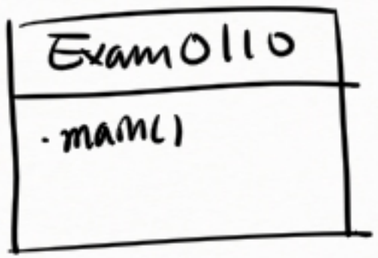


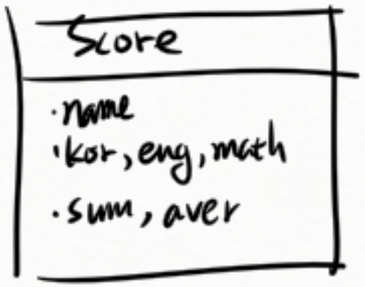
* public 클래스 — default 클래스 (package private class)



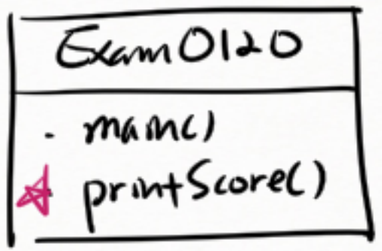
* com.comcs.oop.ex02.Exam01xx

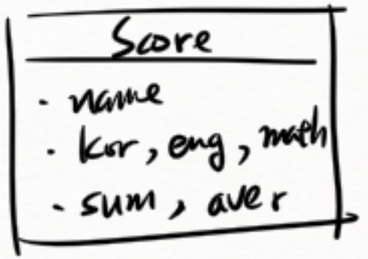
① 날개 변수 사용 → ② class 분법 : 새 데이터 타입 정의 → ③ method 분법 : 중복 코드 추출





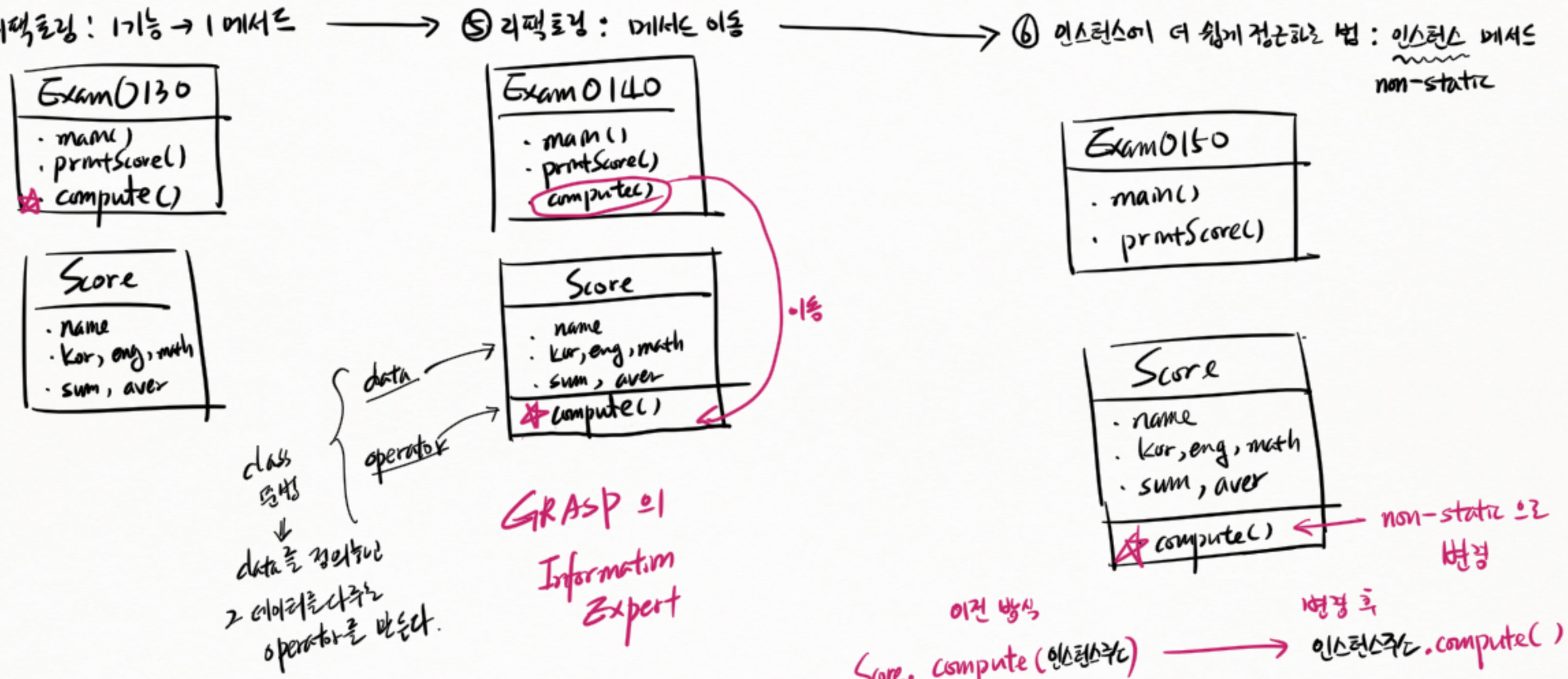
User Defined Data Type





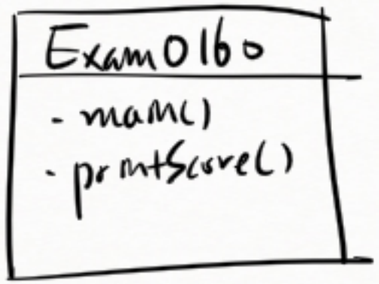
- 레퍼런스 와 인스턴스
  - new
  - Heap 영역
  - garbage
  - garbage collector

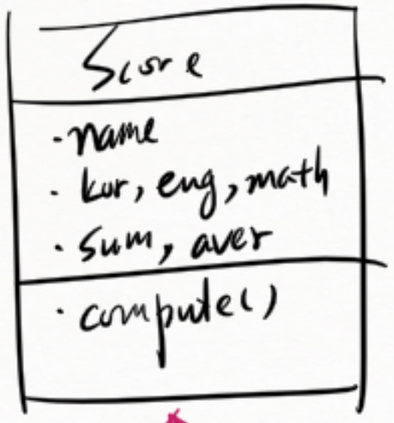
④ 리팩토링 : 1기능 → 1메서드 ⑤ 리팩토링 : 메서드 이동 ⑥ 인스턴스에 더 쉽게 접근하는 법 : 인스턴스 메서드 non-static



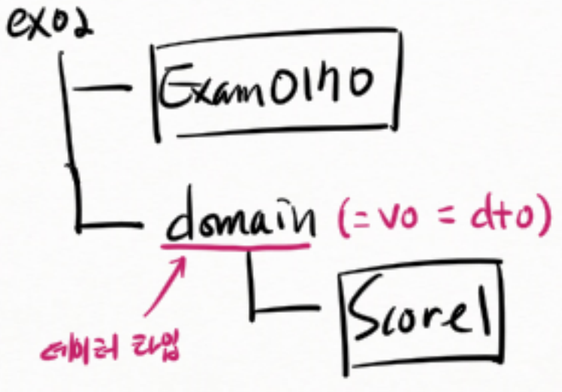
⑦ 패키지 멤버 클래스 → ⑧ 클래스를 패키지로 묶음 → ⑨ 객체 초기화 방법 : 생성자

관리 용이



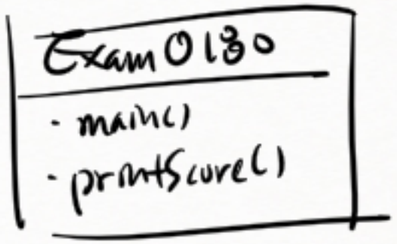


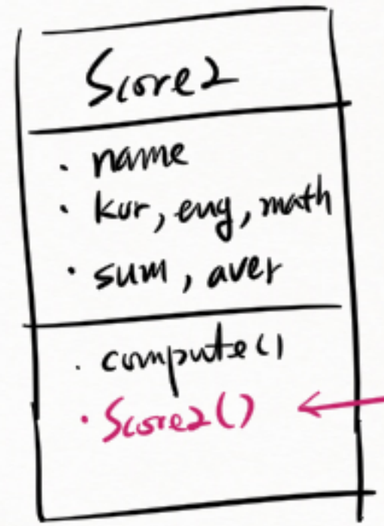
nested class → package member
변경



데이터 타입

접근제어 종류
- public : 완전공개
- protected : 서브클래스, 같은 패키지
- (default) : 같은 패키지
- private : 비공개





생성자 (constructor)

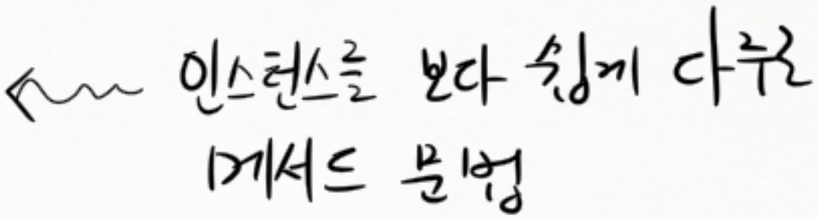
* 스태틱 메서드와 인스턴스 메서드

static 메서드 = 클래스 메서드 ⇒ `Score.compute(s1);`

- `Score` : 메서드가 소속된 클래스
- `s1` : 파라미터 값

non-static 메서드 = 인스턴스 메서드 ⇒ `s1.compute();`

- `s1` : 메서드가 소속된 클래스의 인스턴스 주소



* 인스턴스 메서드와 this

전달

```
s1.compute();
```

메서드를 호출한 cm
왼쪽에서 넘겨준 인스턴스 주소를 받는
built-in 참조 변수
⇓
non-static 메서드에만 존재한다.

* 인스턴스 주소를
this 내장 변수에 저장된다
⇓
인스턴스 주소를 받기 위해
따로 변수를 선언할
필요가 없다.

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

* 생성자

```
class Score {
    Score(파라메터, ...) {
    }
}
```

클래스 이름과 같아야 한다

리턴타입이 없다

* 생성자 사용 전/후

① 사용 전

```
Score s = new Score();
s.name = "홍길동";
s.kor = 100;
s.eng = 100;
s.math = 100;
s.compute();
```

Score 설계도에 따라 인스턴스 생성 ⇓ 생성자 호출

② 사용 후

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

↓ 인스턴스 생성 후 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();
}
```

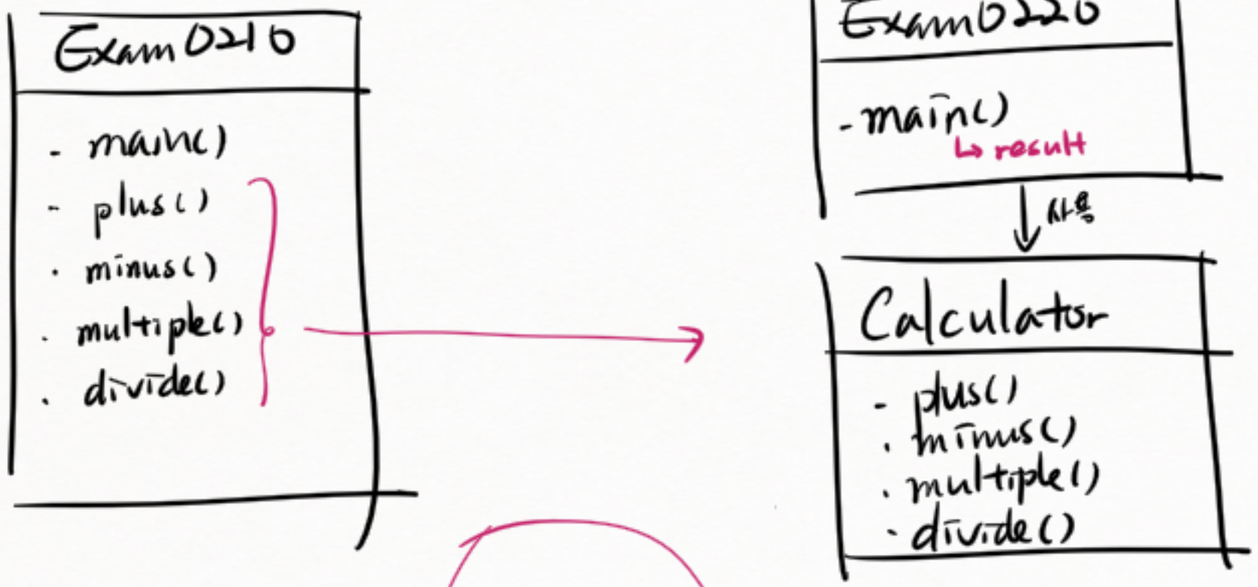
* 스태틱 필드 ⟶ 인스턴스 필드

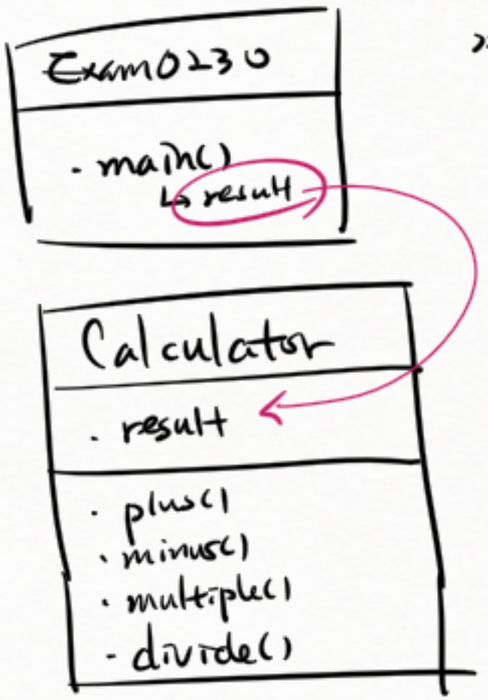
(com.eomcs.oop.ex02.Exam02xx)

① 메서드 부류 전 ⟶ ② 계산하는 메서드 묶음 : 클래스 문법 ⟶ ③ GRASP 패턴 : Information Expert

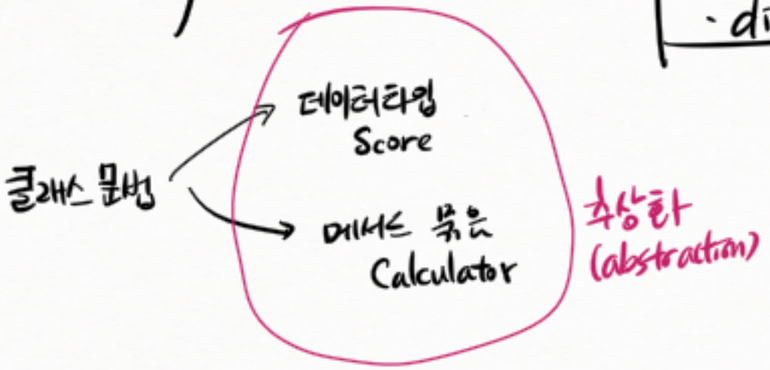
⇓

계산은 수행하라 클래스에서 필드를 관리하는 것이 좋다.

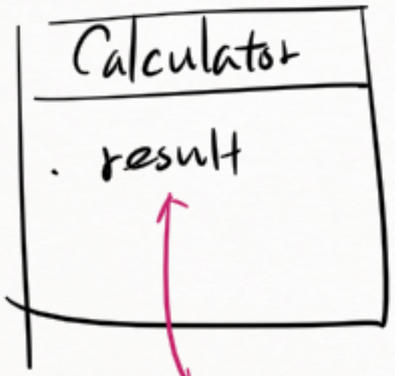




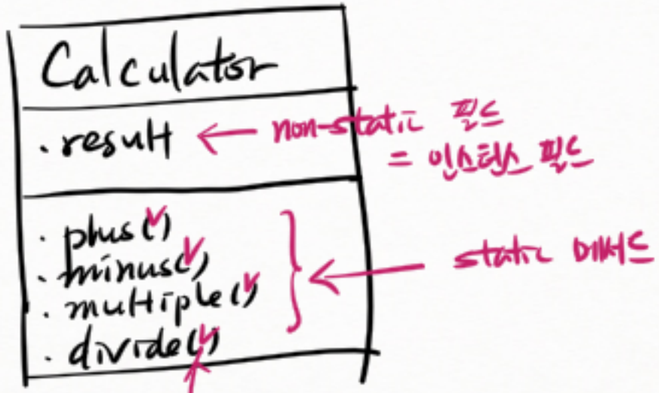
• static 변수 (필드)



④ 클래스 변수의 한계 ⟶ ⑤ 인스턴스 변수로 전환



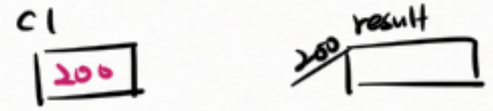
static 변수를 클래스 당 1개 생성.



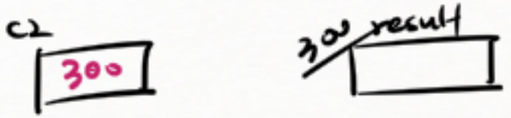
result 가 있는 인스턴스 주소 전달

```
void plus(Calculator obj, int value) {
    obj.result += value;
}
```

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




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



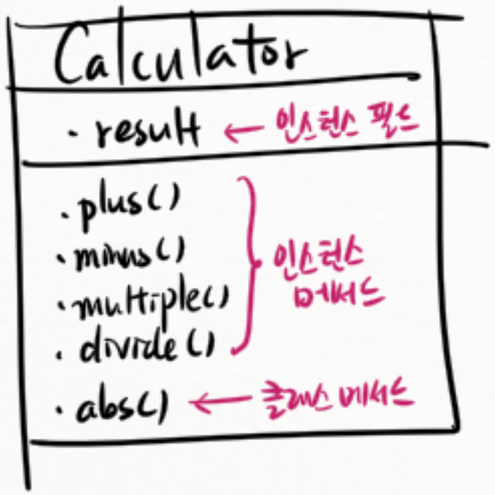

```
Calculator.plus(c1, 2);
```

200

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

300

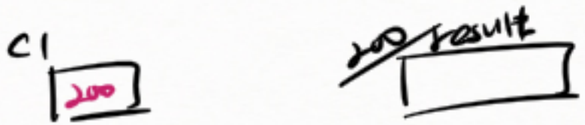
⑥ static 메서드 ⟶ 인스턴스 메서드 전환



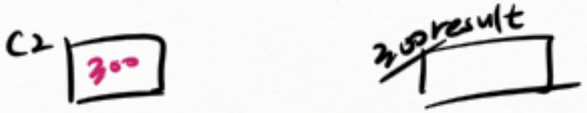

```
void plus(int value) {
    this.result += value;
}
```

↑ 인스턴스 메서드의 Built-in 변수

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




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



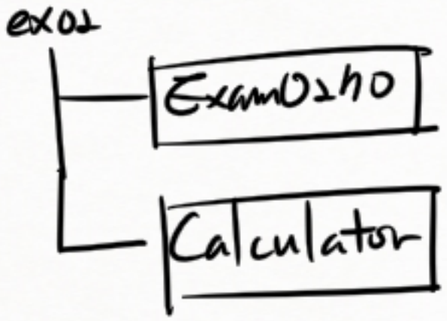

```
c1.plus(2);
```

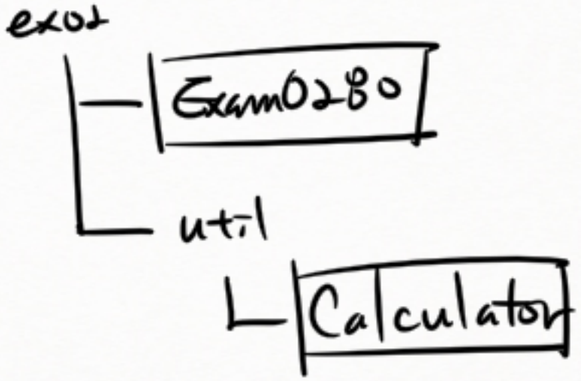
200

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

300

① Nested 클래스 → Package member 클래스 → ② 패키지 분리



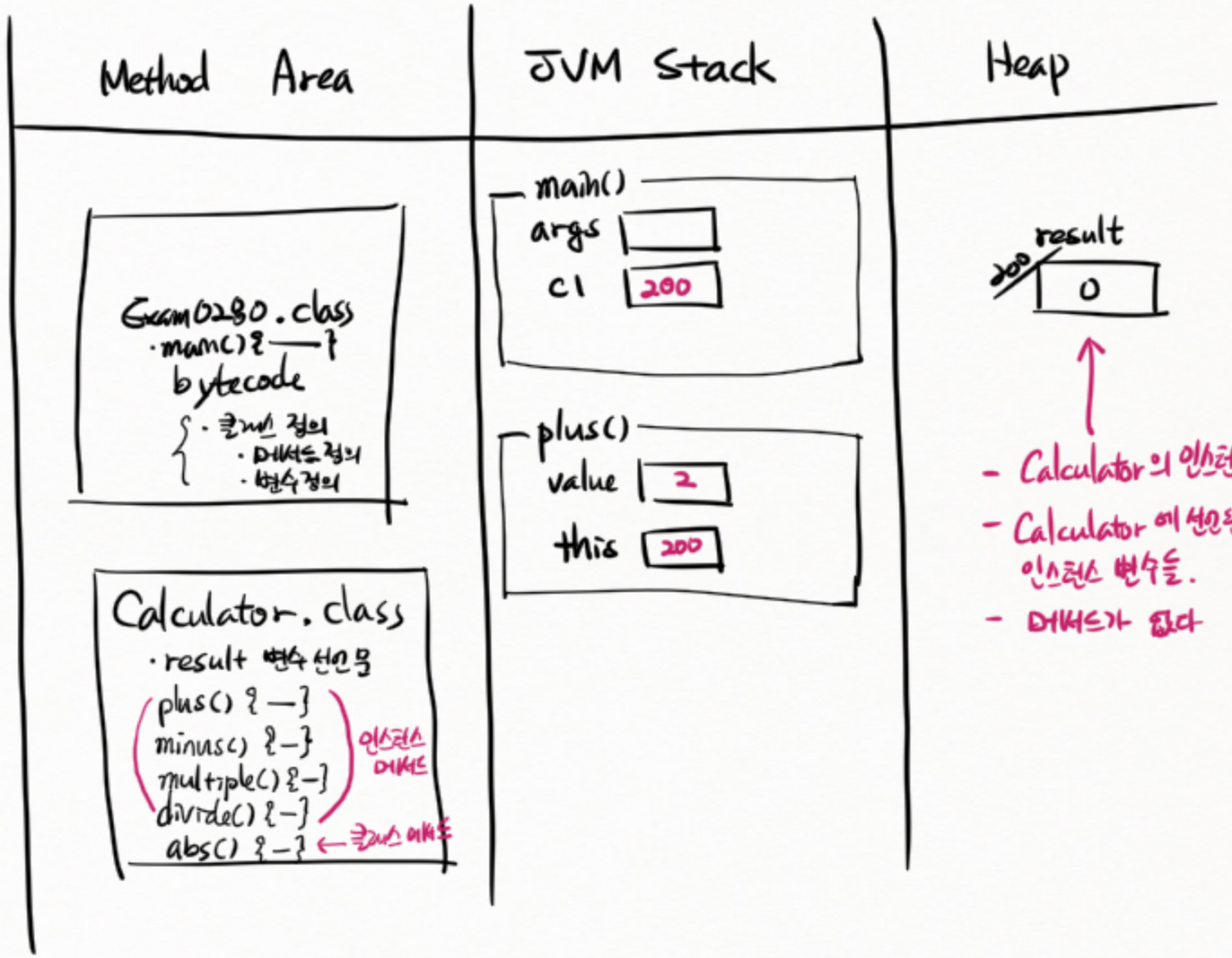


* JVM 메모리 영역 & 변수, 메서드

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

인스턴스 주소

Calculator 에 선언된 인스턴스 변수를 Heap 에 생성한다



* 스태틱 변수, 인스턴스 변수, 로컬 변수

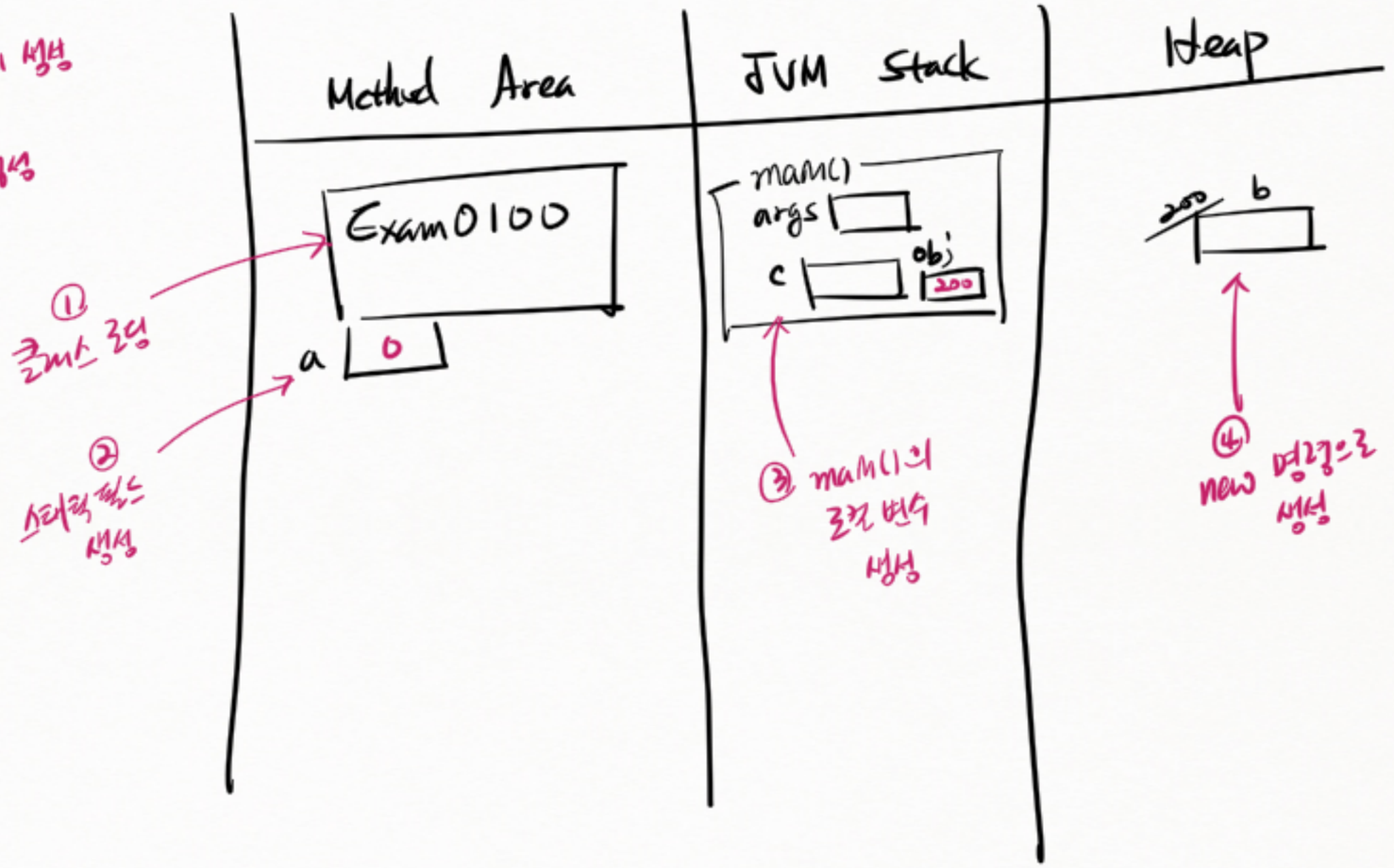
```
class Exam0100 {
    static int a;
    int b;
    void main() {
        int c;

        Exam0100 obj;
        obj = new Exam0100();
    }
}
```

스태틱 필드 → static int a; ← 클래스가 로딩될 때 생성

인스턴스 필드 → int b; ← new 명령을 실행할 때 생성

로컬 변수 → int c; ← 메서드가 호출될 때 생성



* 스태틱 메서드 vs 인스턴스 메서드

스태틱 메서드 : 문자열을 다룬 일반적인 기능 수행

클래스.메서드(값, 값, ...) ⟹ String.format("%s 님 환영합니다!", "홍길동")

클래스 → String

⇓

"홍길동 님 환영합니다!"

```
String str = "abcdef";
```

(0 1 2 3 4 5)

메모리.메서드(값, 값, ...) ⟹ str.charAt(3);

인스턴스 → str (this)

인스턴스 메서드 : 특정 객체에 대해 작업 수행

메모리에 저장된 값은 사용하여 작업을 수행한다.

⇓

'd'

* static 필드, instance 필드, local 변수

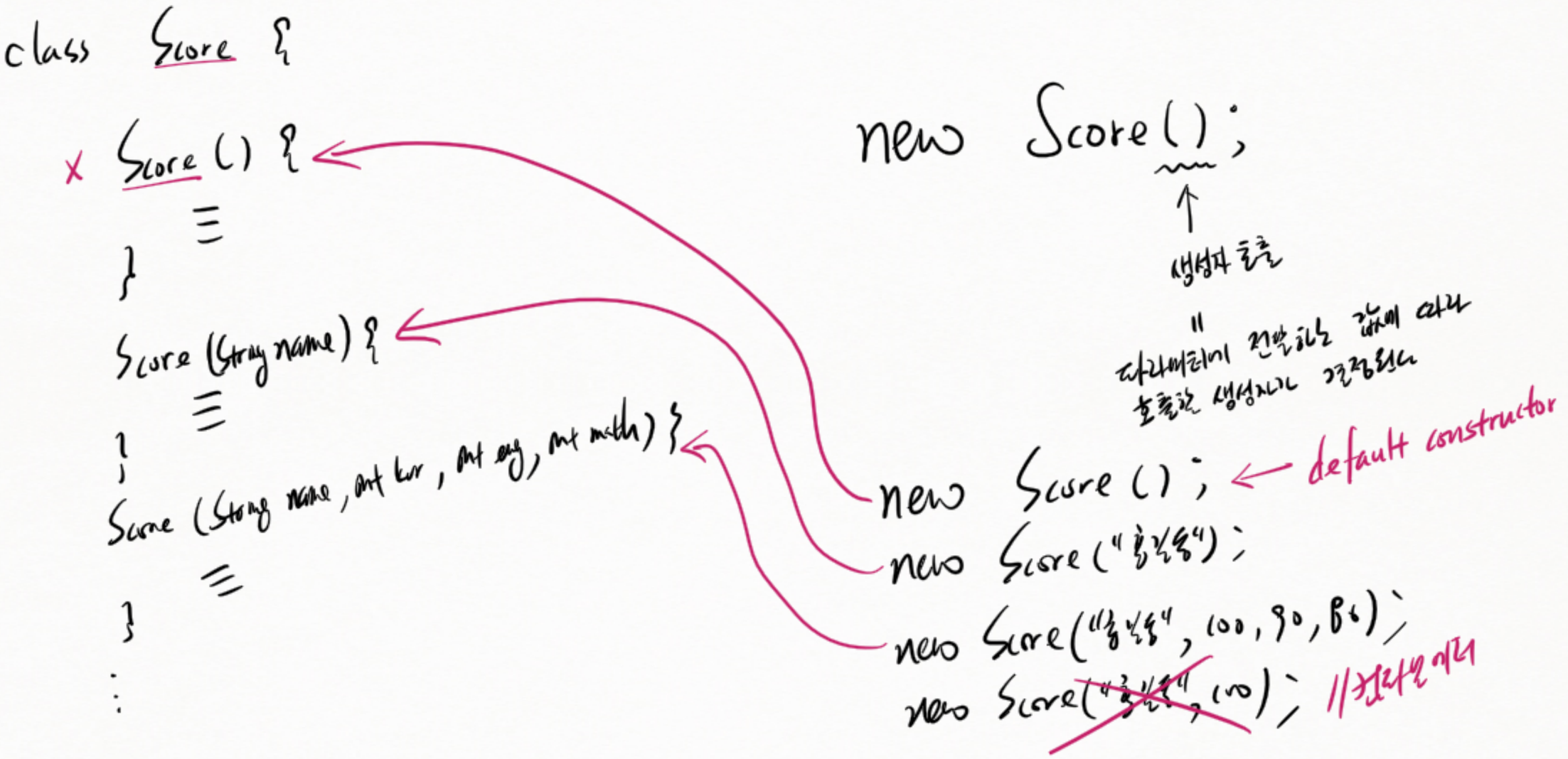
```
class Example {
    static int a;
    int b;
    main() {
        int c;
    }
}
```

① static 필드
- 클래스가 로딩될 때 생성된다 (Method Area)

② instance 필드
- new 연산자로 인스턴스 생성될 때 만들어 (Heap)

③ local 변수
- 메서드 호출될 때 만들어 (JVM stack)

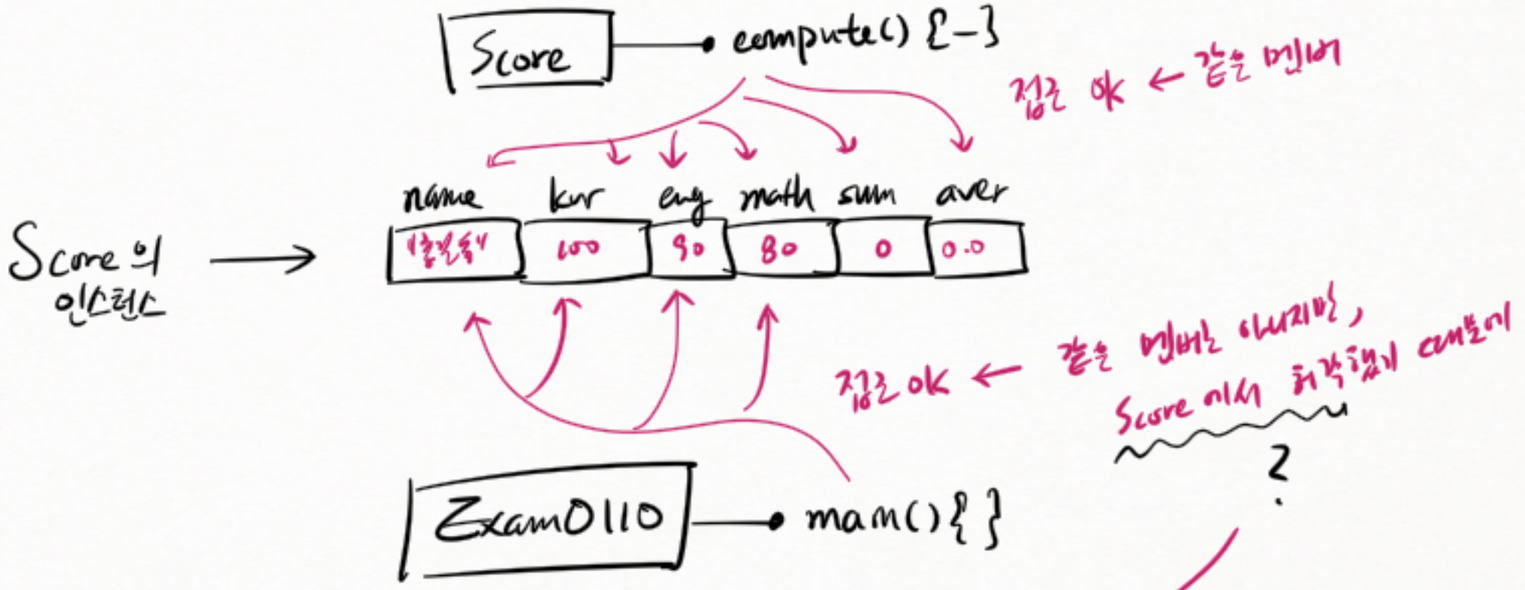
* 생성자


class Score {
Score() {
}
Score(String name) {
}
Score(String name, int kor, int eng, int math) {
}
new Score();
생성자 호출
=
다시메모리에 전달하는 값에 대한
호출한 생성자도 결정된다
new Score(); ← default constructor
new Score("홍길동");
new Score("홍길동", 100, 90, 85);
new Score("홍길동", 100); //컴파일에러

Encapsulation + getter / setter

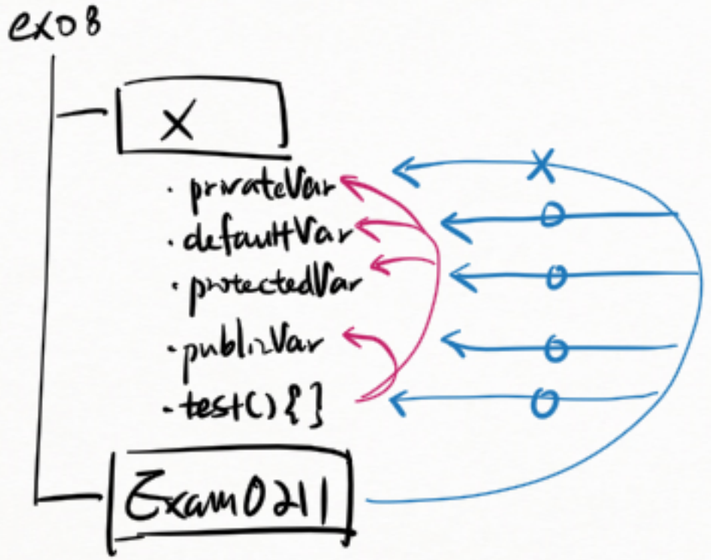
* 멤버에 대한 접근제어 : (default)

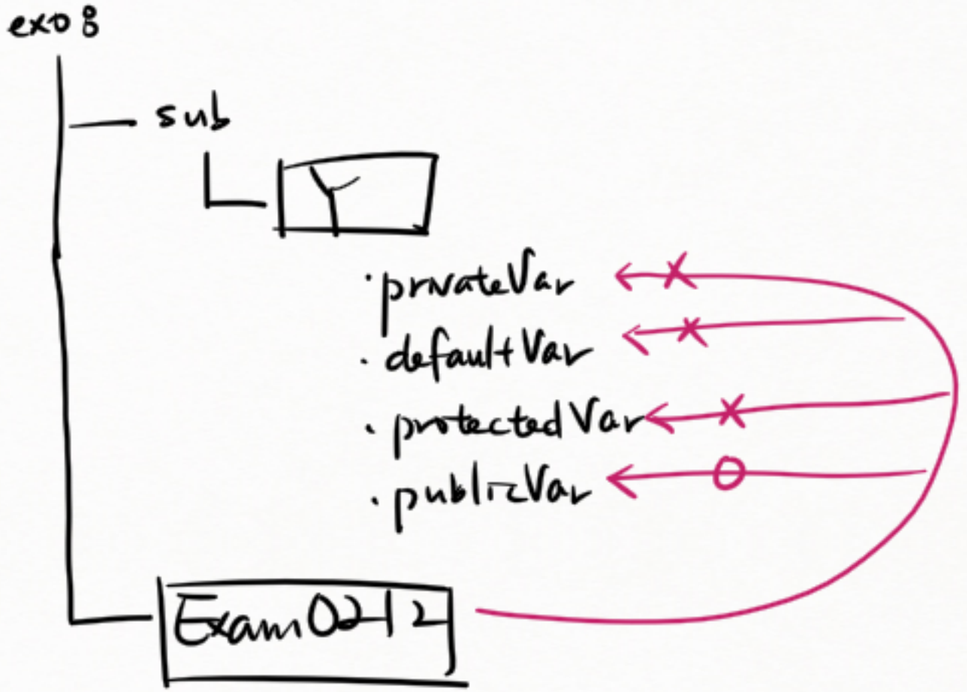
```
class Score {
    String name;
    int kor;
    int eng;
    int math;
    int sum;
    float aver;
    compute() {-}
}
```



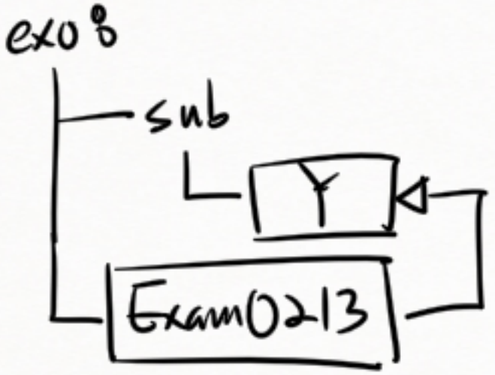
* 멤버 접근 제어
  - private : 같은 멤버만
  - (default) : 〃 + 같은 패키지
  - protected : 〃 + 〃 + 서브클래스의 멤버
  - public : 모두

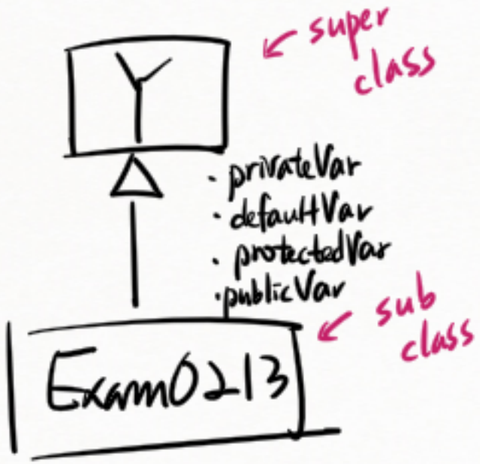
* 접근 범위 테스트

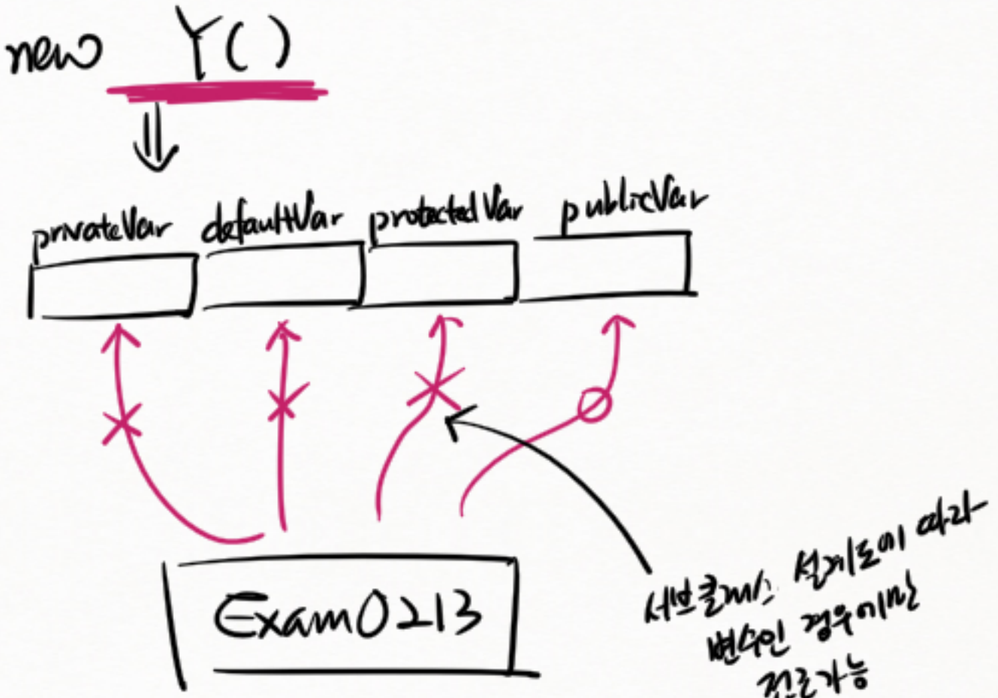

ex08
X
· privateVar
· defaultVar
· protectedVar
· publicVar
· test(){}
Exam0211


ex08
sub
Y
· privateVar
· defaultVar
· protectedVar
· publicVar
Exam0212

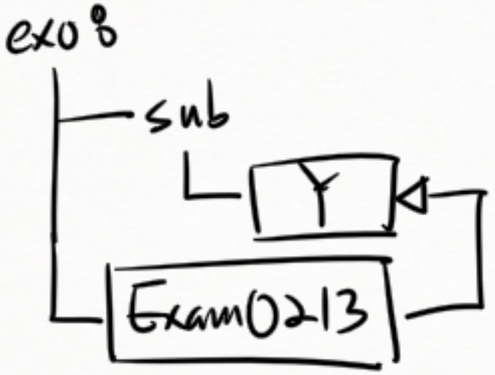
* 접근 범위 테스트 : protected 접근

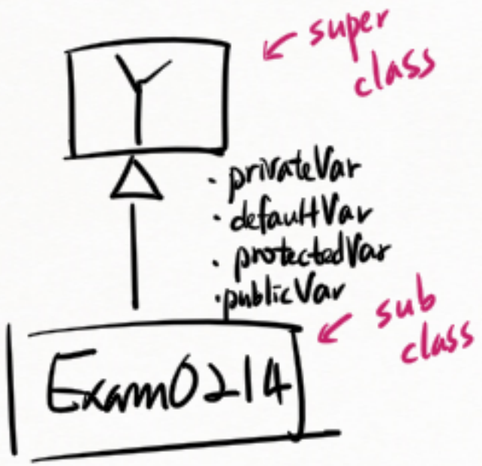






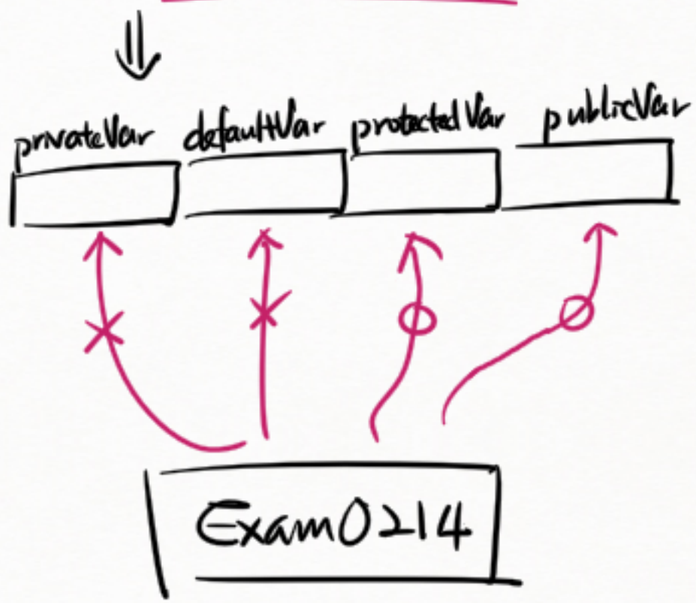
* 접근 범위 테스트 : protected 접근 Ⅱ





* 상속 ?
- 재사용 용이

new Exam0214();



상속 (Inheritance)

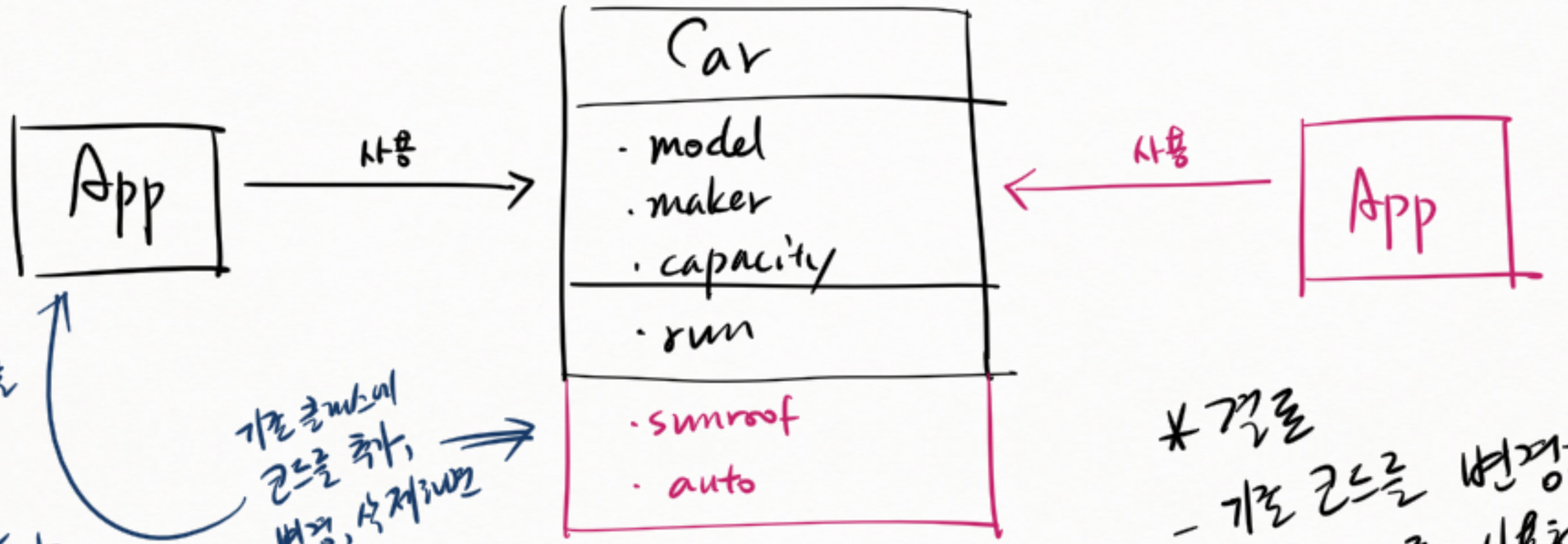
* 기능 확장 — ① 기존 클래스를 변경

① 고객사 A

② 고객사 B : 기능 추가 요구

⇓

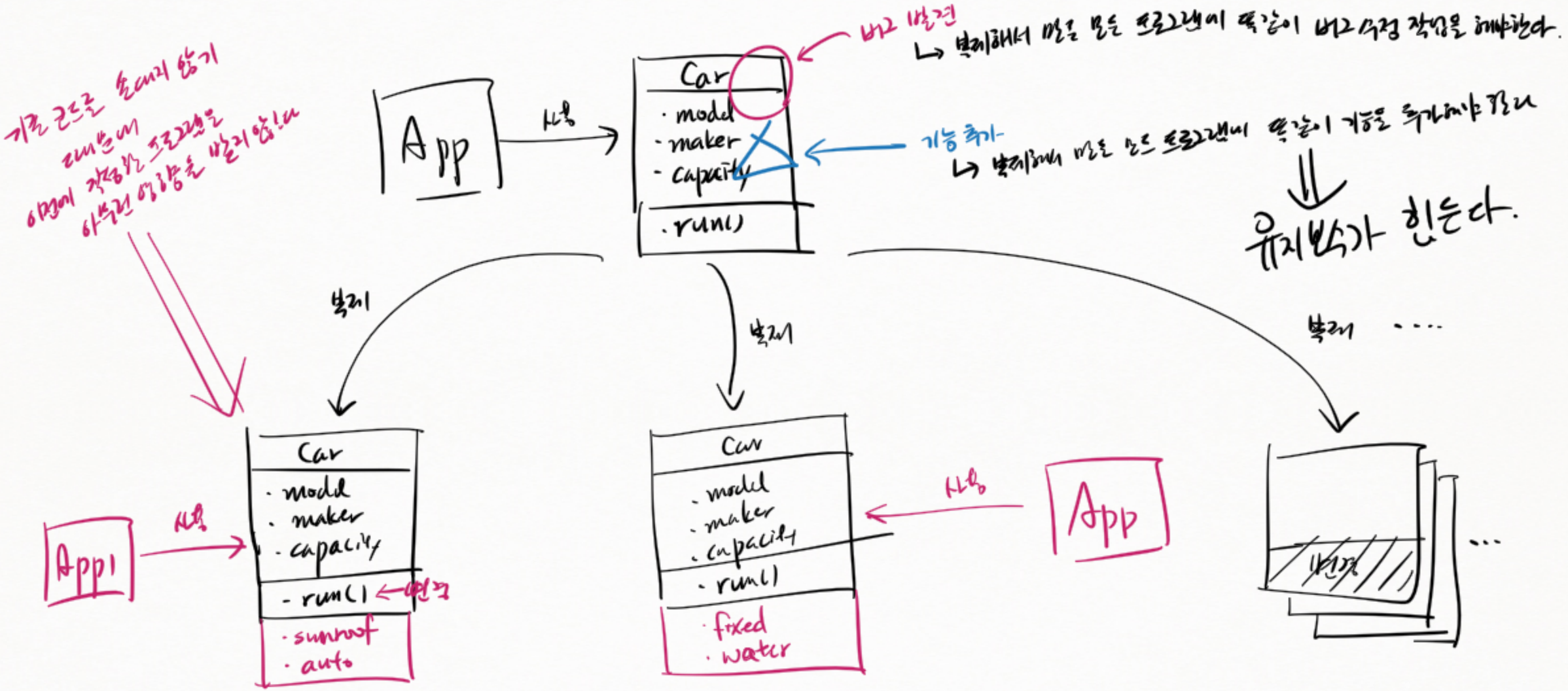
기존 클래스에 코드추가



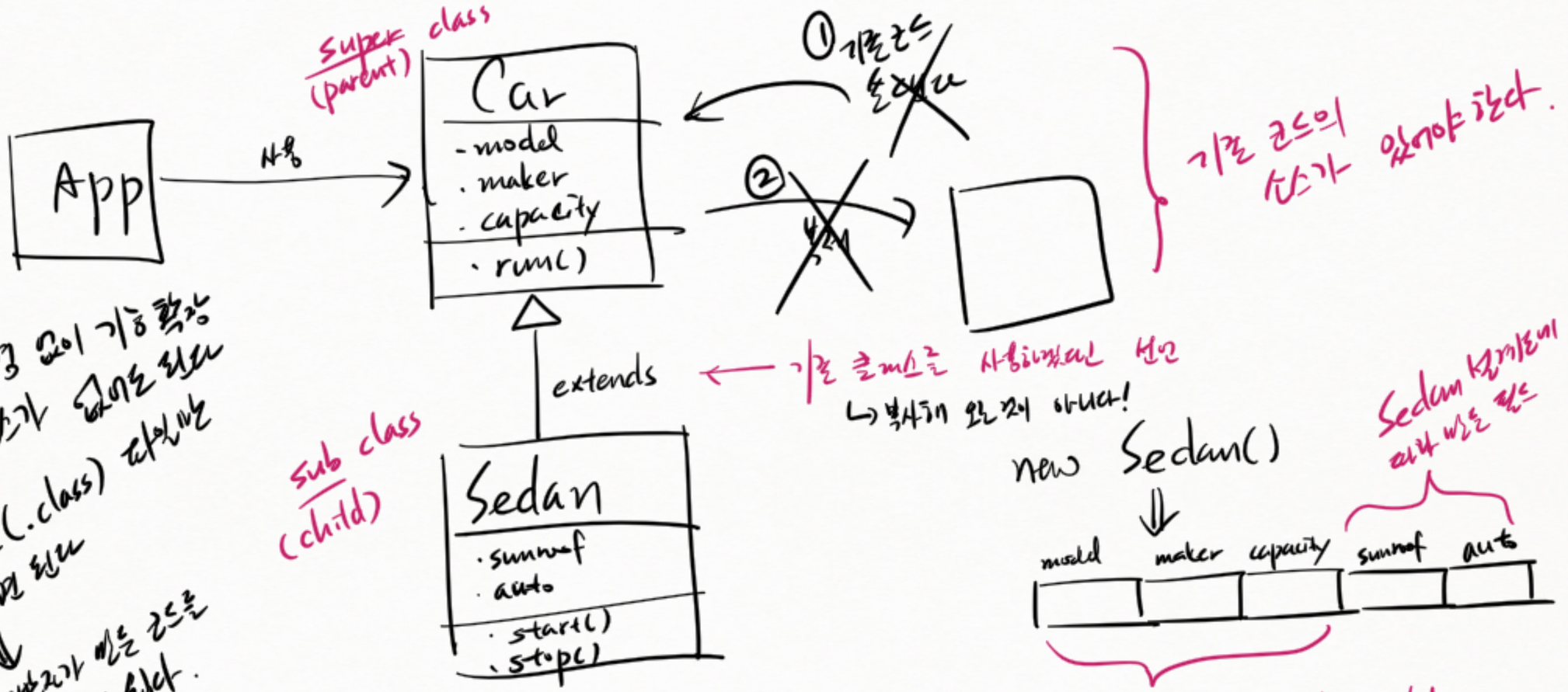
- 컴파일 오류 발생
- 메서드 스펙 변경 → 실행시 안되는 상황
  - 파라미터 추가
  - 이름변경
  - ⋮

* 결론
- 기존 코드를 변경하면 기존 코드를 사용하는 쪽에 영향을 끼친다.

*기능 확장 -② 기존 코드를 복제한 후 기능 추가

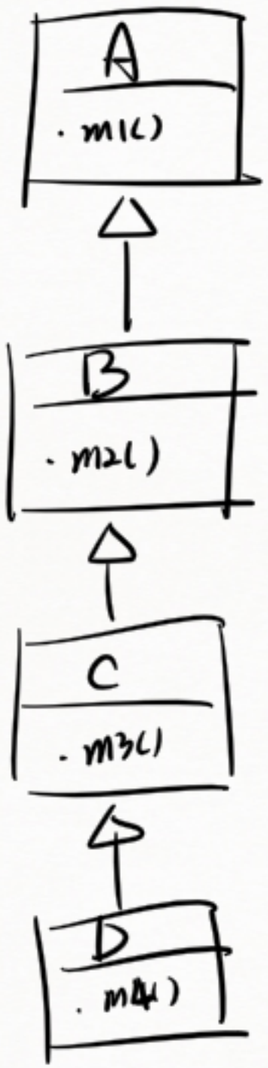


* 기능확장 - ③ 상속을 이용한 기능확장

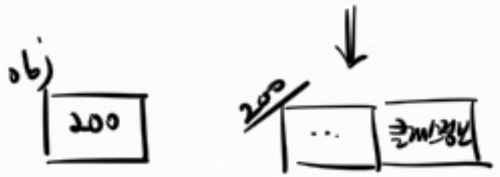


* 결론
- 기본 코드 변경 없이 기능 확장
- 기본코드의 변수가 없어도 된다
↓
bytecode(.class) 파일만 있으면 된다
↓
다른 개발자가 만든 코드를 재사용 하기 쉽다.

* 상속과 메서드 호출.



B obj = new B();



obj.m2();
obj.m1(); //ok
obj.m3();

D obj = new D();

obj.m4();
obj.m3();
obj.m2();
obj.m1();

B obj = new D();

obj.m4();
obj.m3();

컴파일러는 변수(레퍼런스)의 타입에서 메서드를 찾아올라간다. 레퍼런스가 실제 어떤 클래스의 인스턴스를 가리키는 것인지 여기서 안다.

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

덤프트럭 r1 =

자동차 r2 = new 덤프트럭()

운송수단 r3 =

덤프트럭 r1 ✗

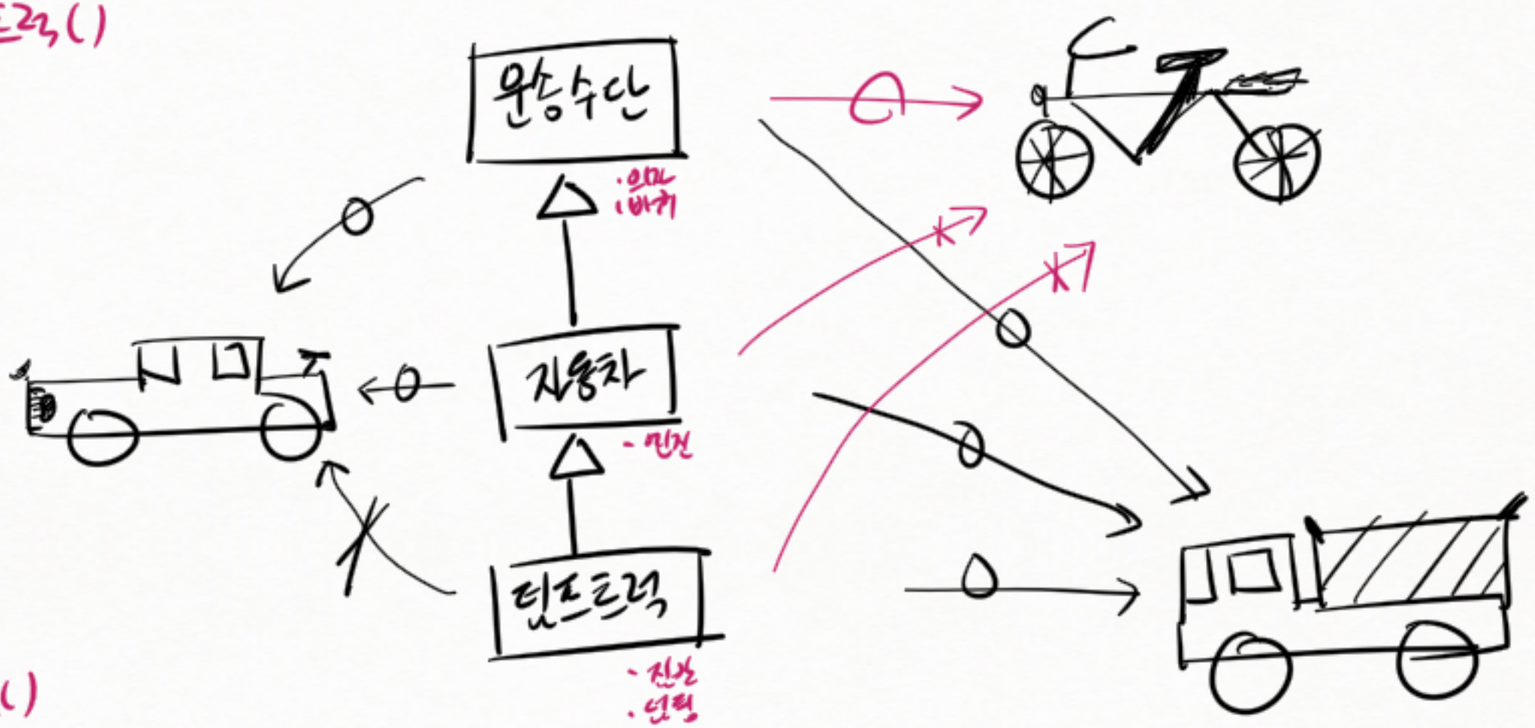
자동차 r2 = new 자동차()

운송수단 r3 =

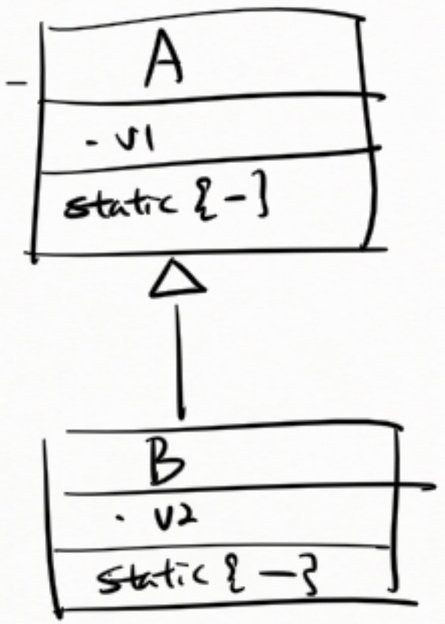
덤프트럭 r1 ✗

자동차 r2 ✗ new 운송수단()

운송수단 r3 =

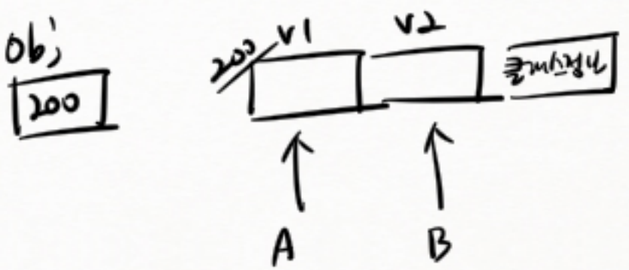


* 상속과 클래스 로딩

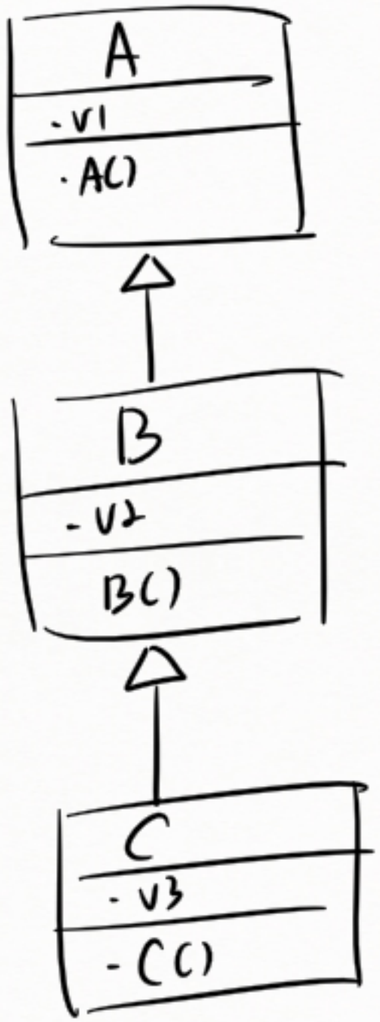


B obj = new B();

① 부터 클래스부터 로딩
② 부터클래스부터 인스턴스 필드 생성



+ 생성자 호출 순서



C obj = new C();

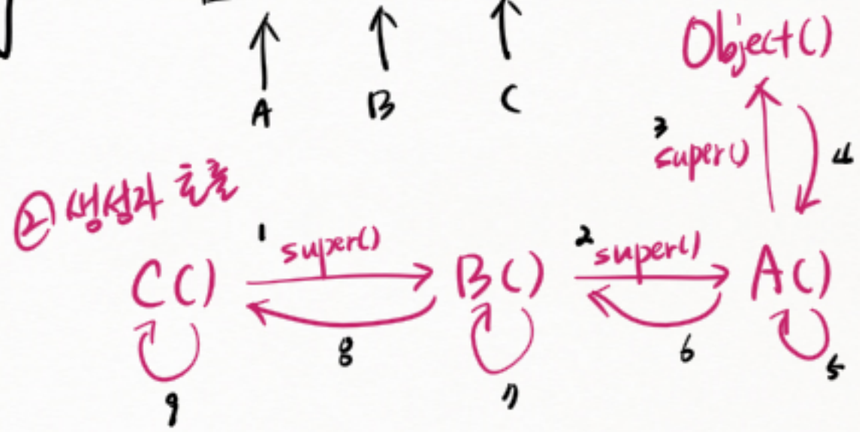
① 인스턴스 필드 생성
② 생성자 호출

① 인스턴스 필드 생성

obj
200

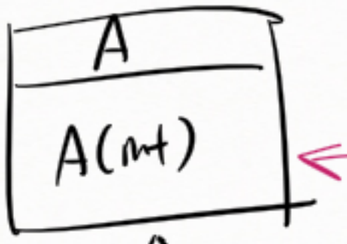
	v1	v2	v3	
200	100	200	300	...
	A	B	C	

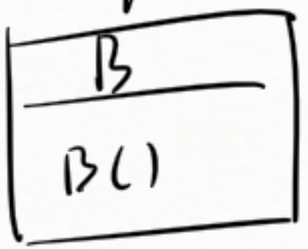
② 생성자 호출



A() 생성자
B() 생성자
C() 생성자

* 부터 클래스의 생성자 호출



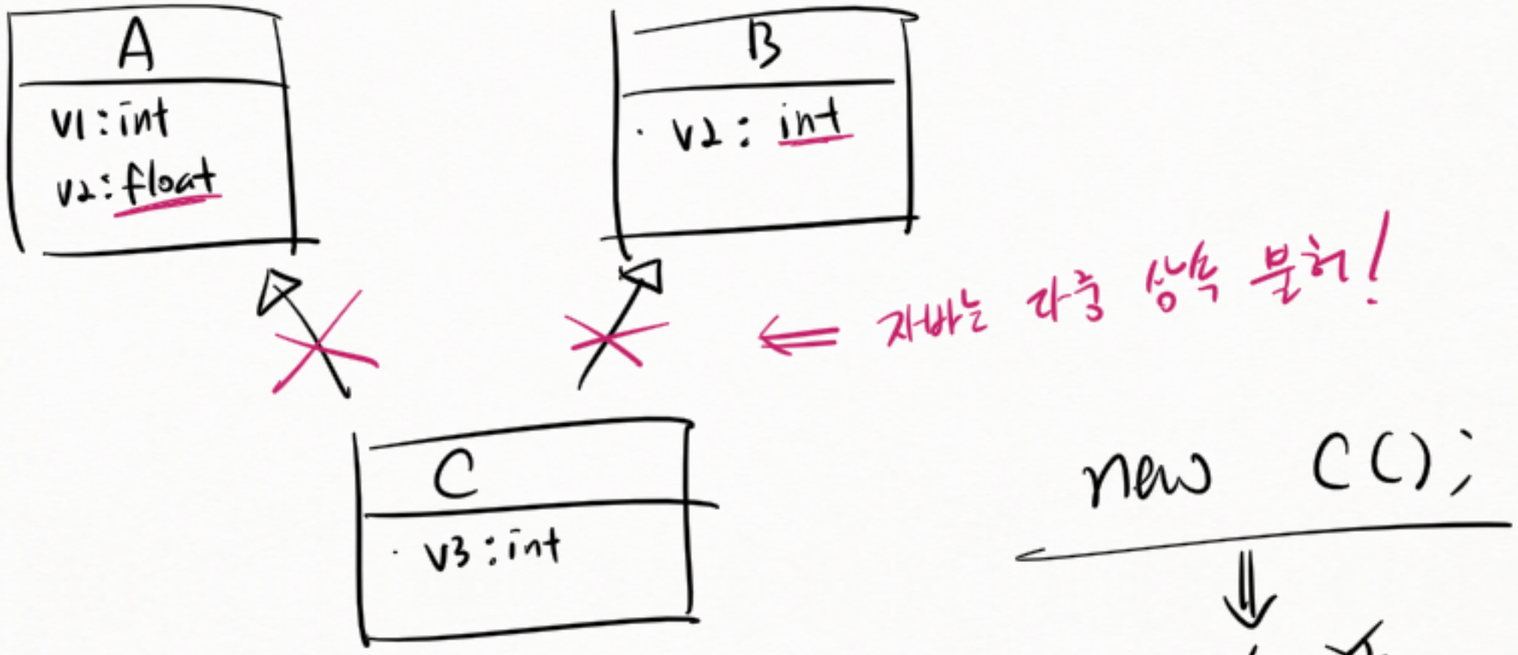


super 클래스의 기본생성자가 호출

```
B() {
    super();
}
```

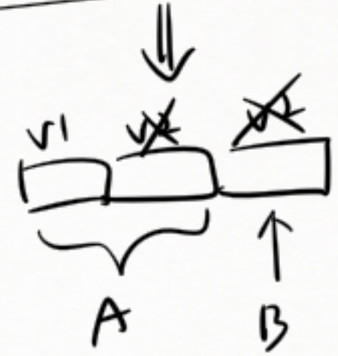
자동 생성!

* 다중 상속?

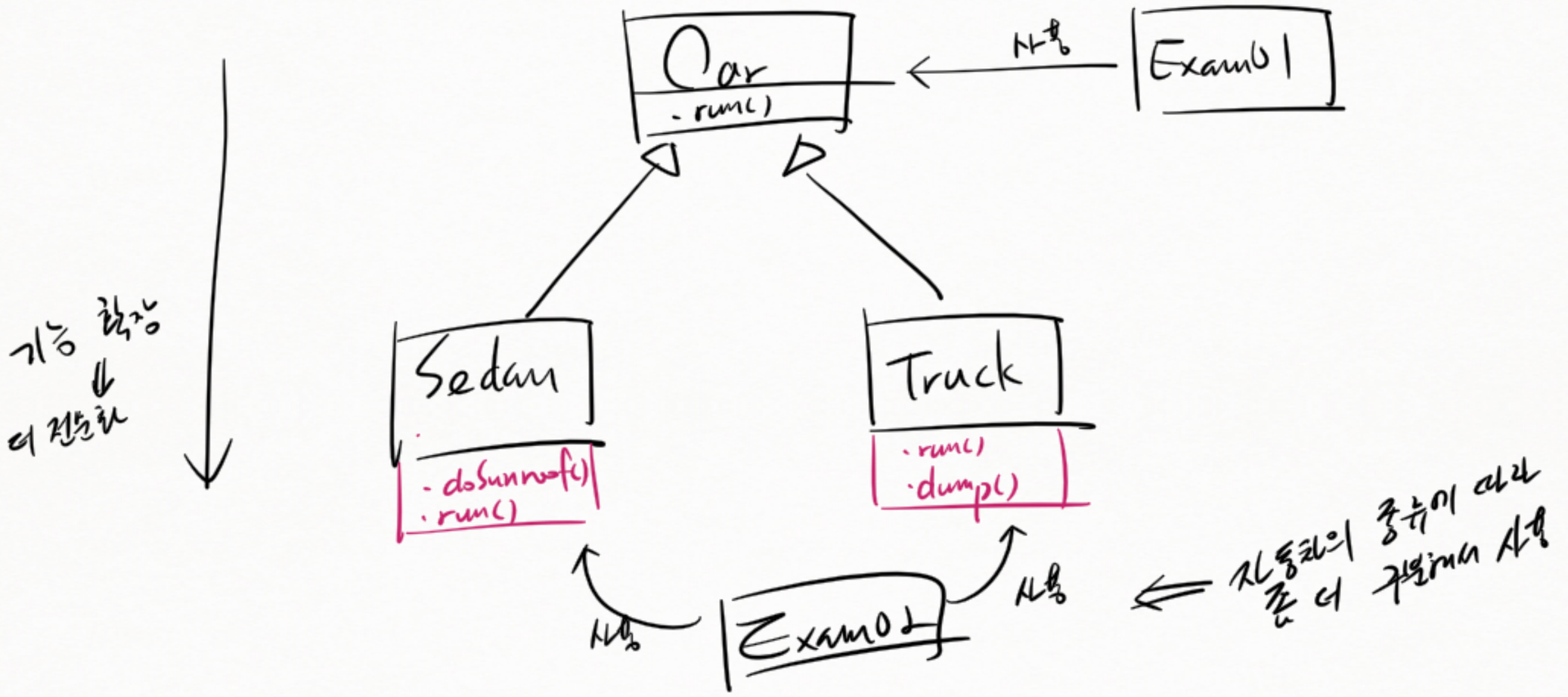


⇐ 자바는 다중 상속 불허!

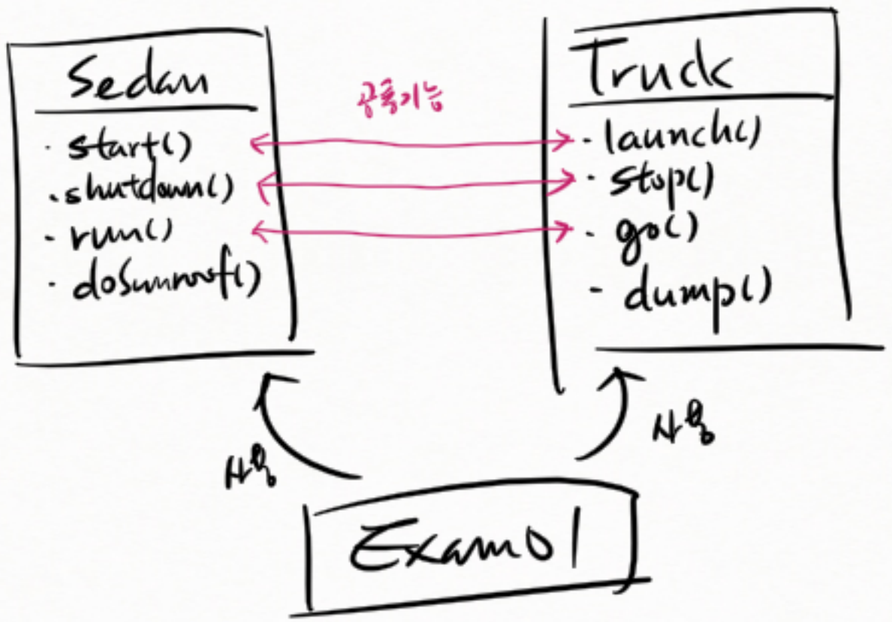
new C();



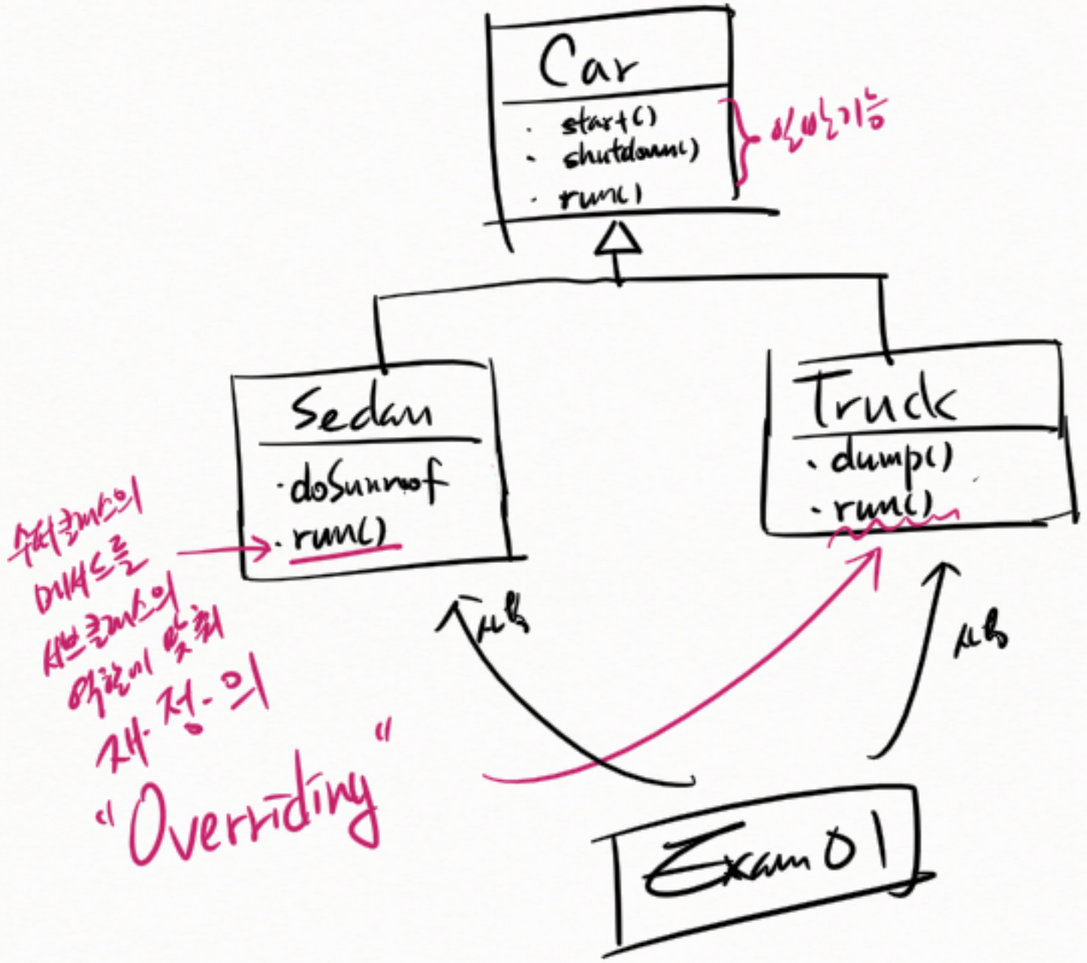
* specialization



* generalization



* generalization



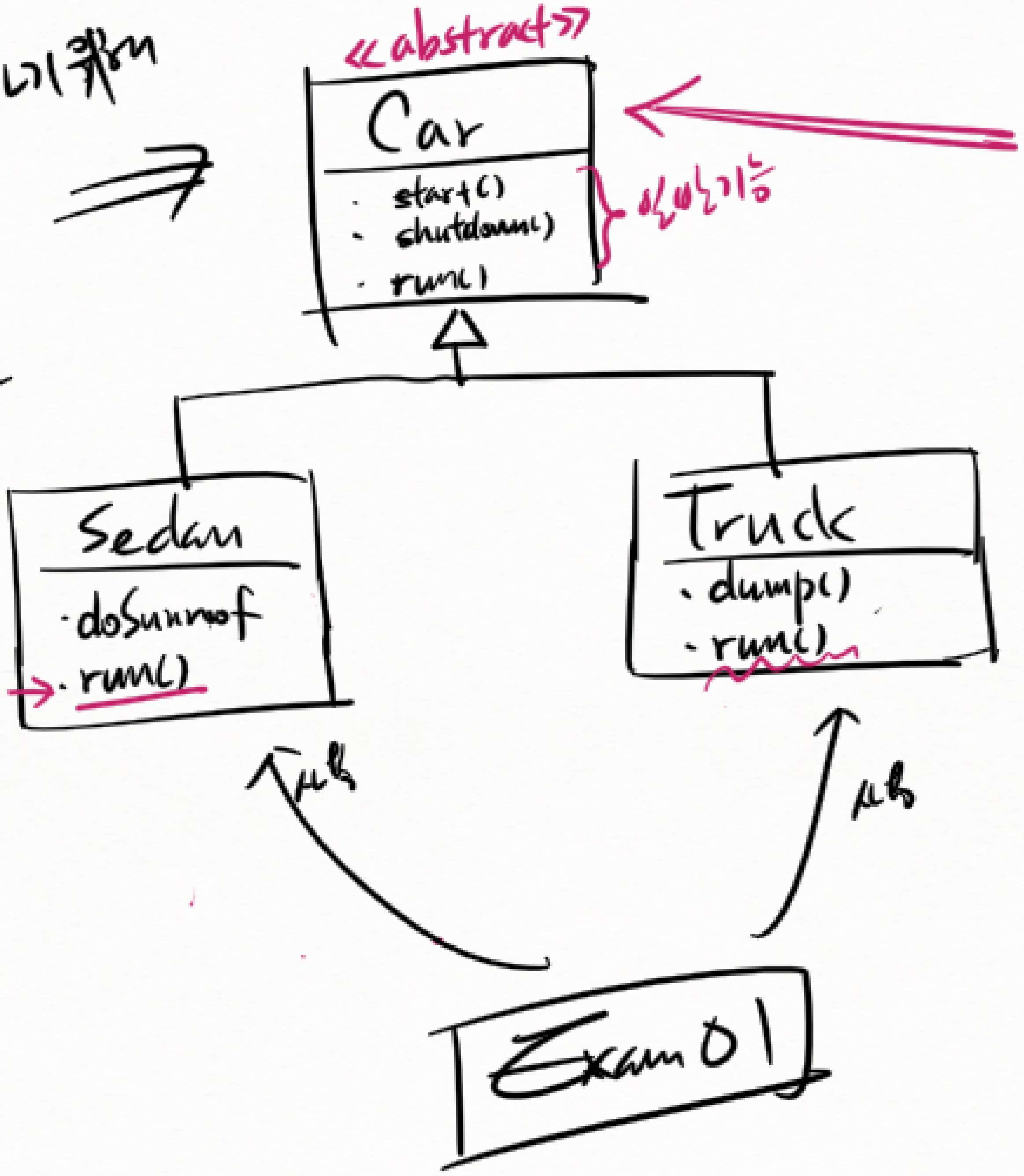
서브클래스를 통해 기능/메서드를 추가함

수퍼 클래스로 받으로 것!

"generalization"

* generalization : 추상클래스

관리를 용이하게 하기 위해
만든 클래스로
사용하려고 만든
클래스가 아니다

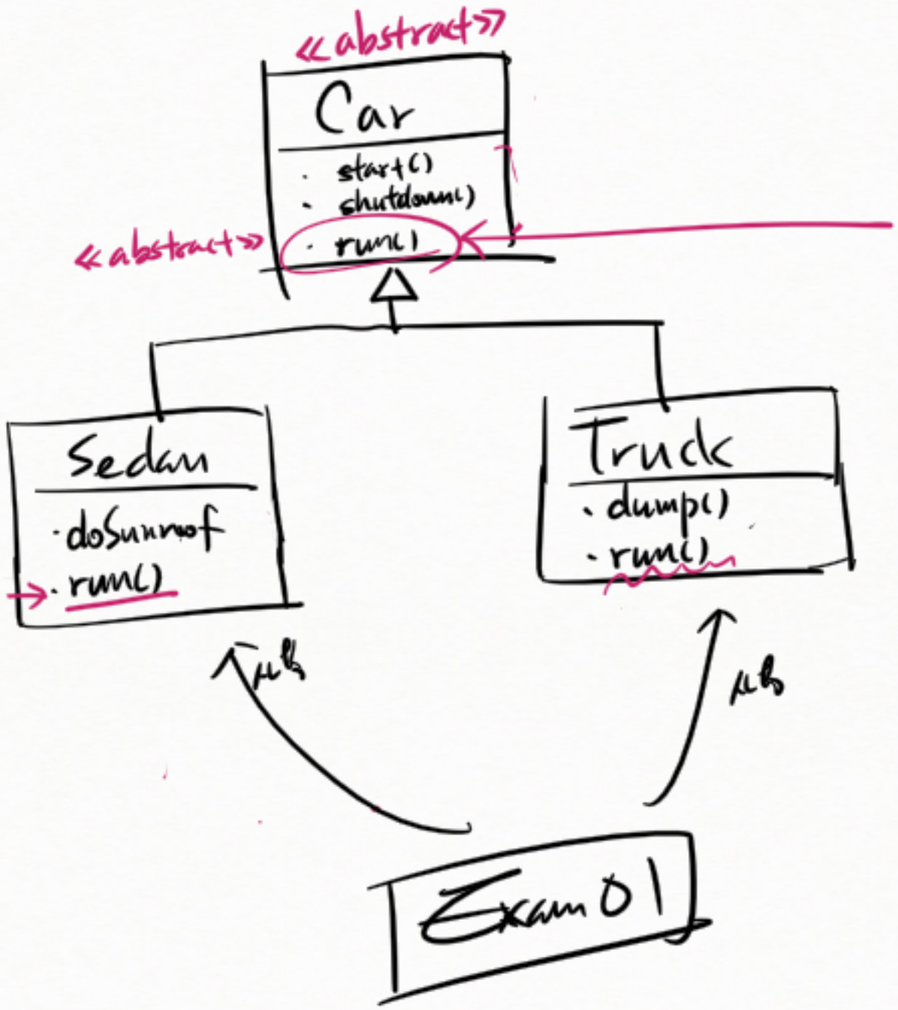


일반용으로 쓰지 못하도록
막는 것이 관리에 좋다.

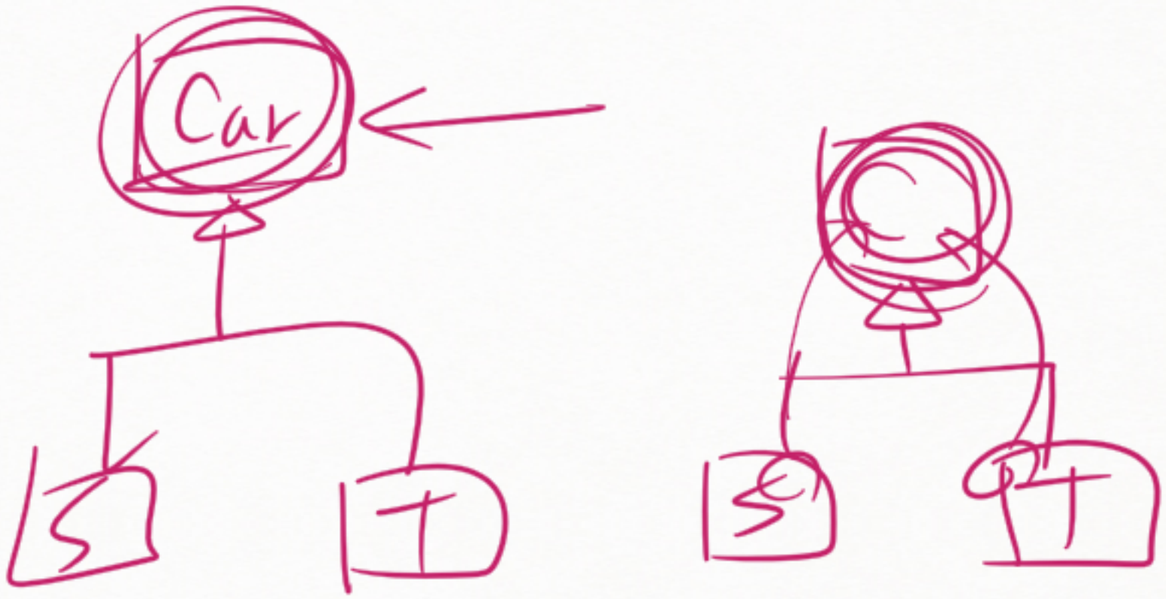
⇓

추상클래스로
선언하라!

* generalization : 추상 메서드



서브 클래스 마다 다르게 강의해야 하는 메서드라면
수퍼 클래스에서 강의하지 말고
서브 클래스에서 강의하도록
강제시키는 특별한 "추상 메서드"


Car
S
T
S
T