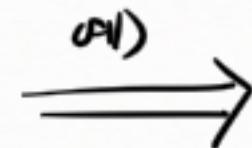


* 클래스 문법

-
- 클래스
- ① 데이터 타입 정의 (User-defined Data Type)
 ~
 개별화
 - ② 메서드 분류

* ↗ 클래스 문법 - 데이터 타입 정의

class 데이터타입 {
 변수선언
 :
}

예) 

class Contact {
 String name;
 String email;
 String tel;
 String company;
}

← 메모리
설정

||
~~new~~ 명령을 실행하면
클래스에 정의된 대로
변수가 준비된다.

* 클래스를 이용하여 새 데이터 타입의 변수 만들기

Contact c = new Contact();

Contact 클래스의 주소를 저장하는
리퍼런스 (reference)

리터리터링 = 클래스명

클래스 선언에 따라 메모리 할당 => Heap 영역

인스턴스 (instance) = 개체 (object)

c | 200

200 | name email tel company

* 레퍼런스 배열

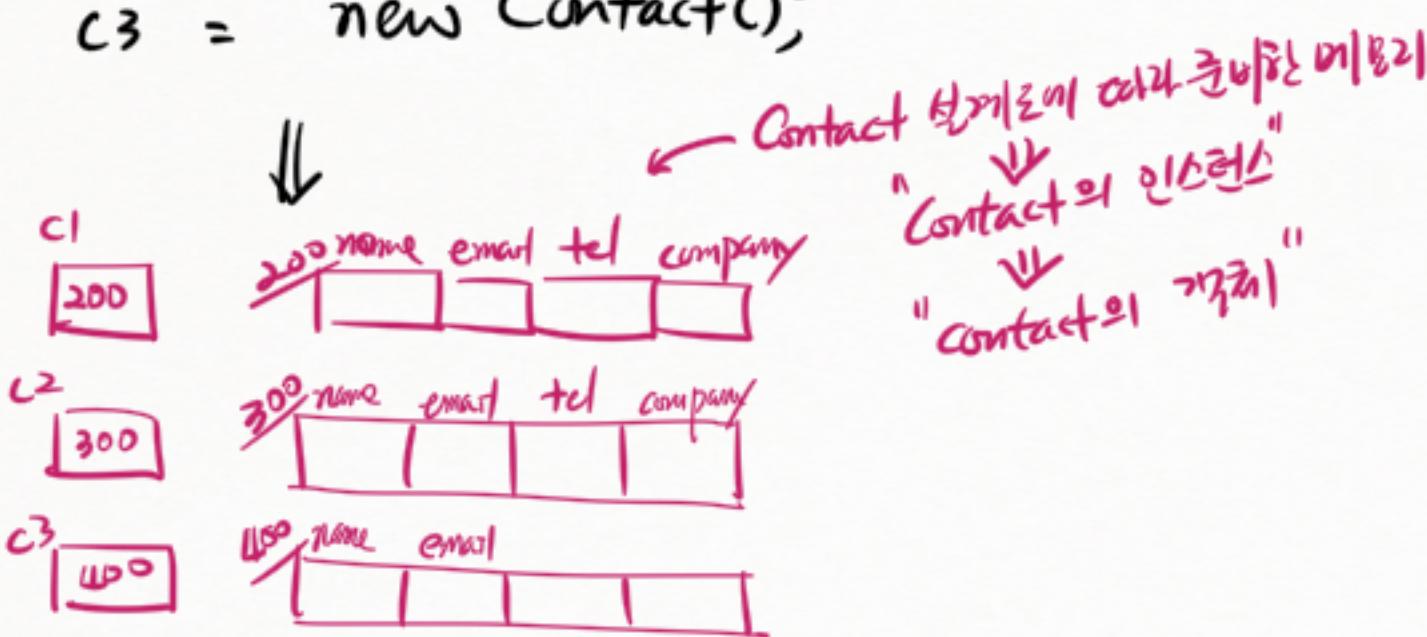
① 배포 사용 전

① 초기화 사용 선
Contact c1, c2, c3; ↪ 인스턴스의 주소를 저장하는 변수;
 "리퍼런스"
 "포인터(pointer)"

```
c1 = new Contact();
```

```
c2 = new Contact();
```

```
c3 = new Contact();
```

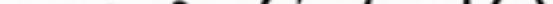


② 배포 사용 흐름

Contact[] arr = new Contact[3];
레퍼런스를 3개 만드는 명령



```
arr[0] = new Contact();
```



A hand-drawn diagram of a Contact object. It consists of a horizontal line with five boxes. The first box is crossed out with a large red X. To its right are four boxes labeled 'name', 'email', 'tel', and 'company' respectively.



```
arr[1] = new Contact();
```



arr[2] = new Contact();

* 레퍼런스와 인스턴스 변수

Contact c = new Contact()



① 인스턴스 변수가 없는 저장

c.name = "홍길동";

인스턴스 주소를
알고 있는
레퍼런스

↑
인스턴스
변수

c.email = "hong@";

c.tel = "1111";

c.company = "비트";

② 인스턴스 변경

c = new Contact()



c.name = "이꺽정";

c.email = "leem@";

c.tel = "2222";

c.company = "캐논";

기존 인스턴스의
주소를 알고 있어
레퍼런스가 한 개로 고정된
''garbage'' 가 된다.

Method Area

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

JVM Stack

```
Score s;
```

s 200

↑
Score의 레퍼런스

Heap

```
new Score()
```



200 name kor eng math sum aver

↑
Score의 인스턴스
(기록체)

* com.eomcs.oop.exam.Exam0114

Method Area

```
class Exam0114 {  
    public static void main(String[] args) {  
        System.out.println("Hello Java");  
    }  
}
```

```
class Score {  
    public static void main(String[] args) {  
        System.out.println("Hello Java");  
    }  
}
```

JVM Stack

```
main()  
args [ ] s [ ]
```

Heap

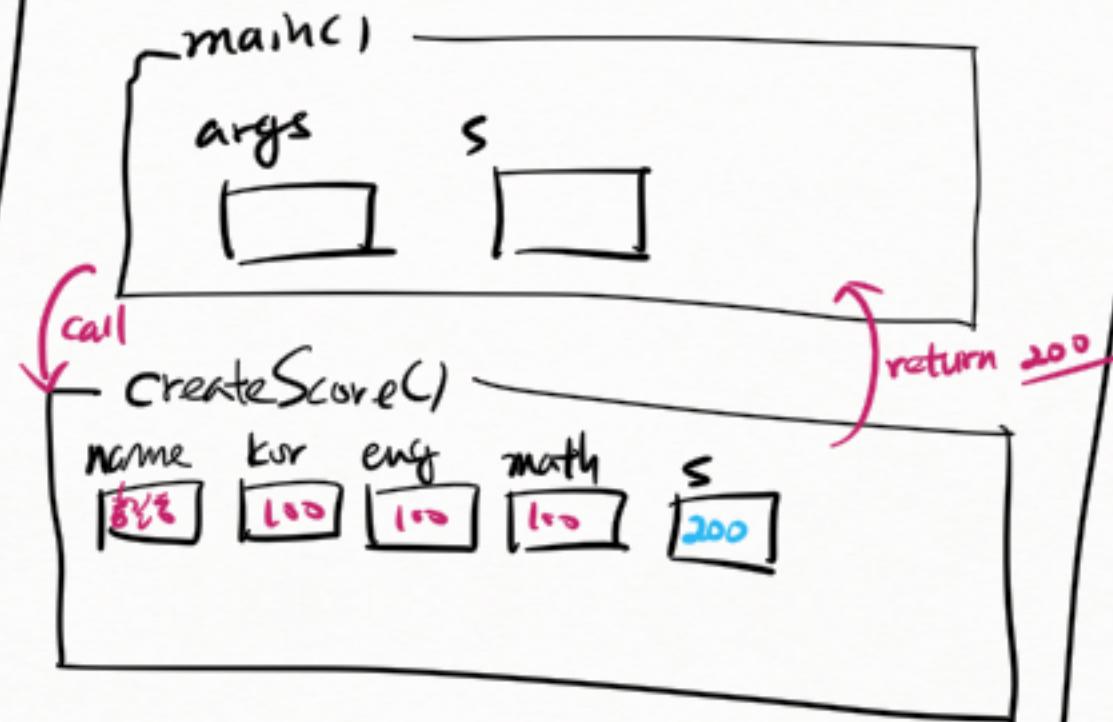
* com.eomcs.oop.ex01.Exam0114

Method Area

```
class Exam0114 {  
    public static void main(String[] args) {  
        Score s = new Score();  
        s.createScore("B/E", 100, 100, 100);  
        System.out.println(s.sum / 3);  
    }  
}
```

```
class Score {  
    private String name;  
    private int kur;  
    private int eng;  
    private int math;  
    private int sum;  
    private double aver;  
  
    public void createScore(String name, int kur, int eng, int math) {  
        this.name = name;  
        this.kur = kur;  
        this.eng = eng;  
        this.math = math;  
        this.sum = kur + eng + math;  
        this.aver = sum / 3;  
    }  
}
```

JVM Stack



Heap

name	kur	eng	math	sum	aver
3/3	100	100	100	300	100

* com.eomcs.oop.exo1.Exam0114

Method Area

```
class Exam0114 {  
    public static void main(String[] args) {  
        System.out.println("Hello Java");  
    }  
}
```

```
class Score {  
    public static void main(String[] args) {  
        System.out.println("Hello Java");  
    }  
}
```

JVM Stack



Heap

name	kor	eng	math	sum	aver
200	32.5	100	100	100	300

* com.eomcs.oop.exam. Exam0114

Method Area

```
class Exam0114 {  
    public static void main(String[] args) {  
        Score s = new Score();  
        s.printScore();  
    }  
}
```

```
class Score {  
    public void printScore() {  
        System.out.println("Hello Score");  
    }  
}
```

JVM Stack



Heap

name	kor	eng	math	sum	aver
200	72.5	100	100	100	300

name	kor	eng	math	sum	aver
200	72.5	100	100	100	300

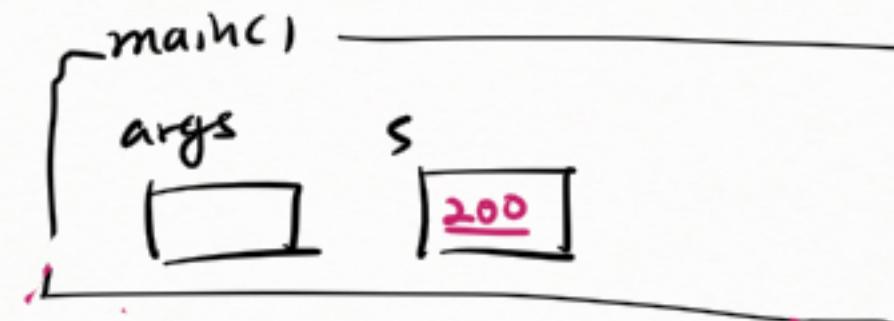
* com.eomcs.oop.exo1.Exam0114

Method Area

```
class Exam0114 {  
    public static void main(String[] args) {  
        System.out.println("Hello Java");  
    }  
}
```

```
class Score {  
    public static void main(String[] args) {  
        System.out.println("Hello Java");  
    }  
}
```

JVM Stack



Heap

name	kor	eng	math	sum	aver
200	72	100	100	300	100

* com.eomcs.001.ex01 . Exam0210

Score s_1, s_2, s_3 :

s_1 | 200

s_2 | 300

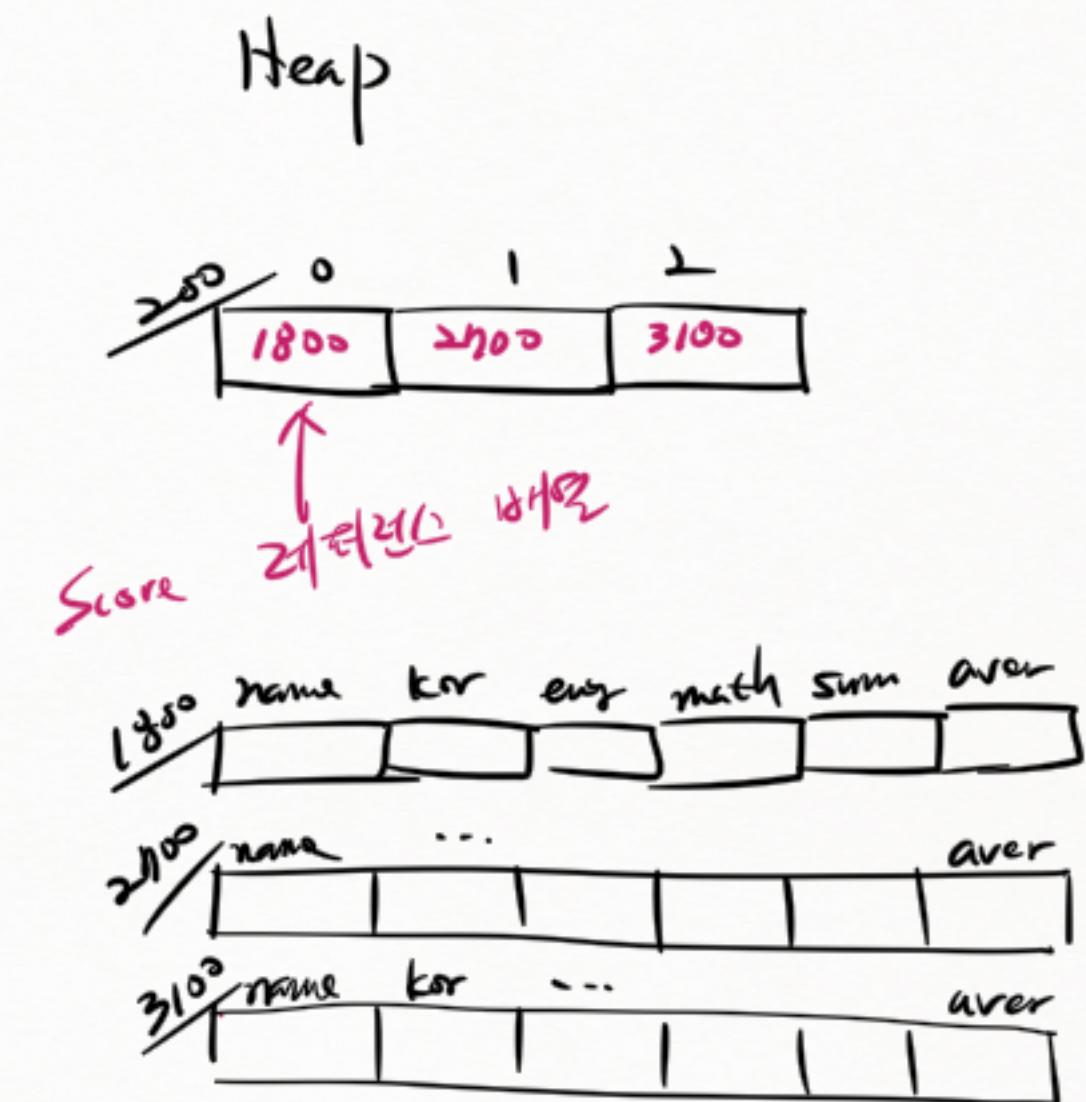
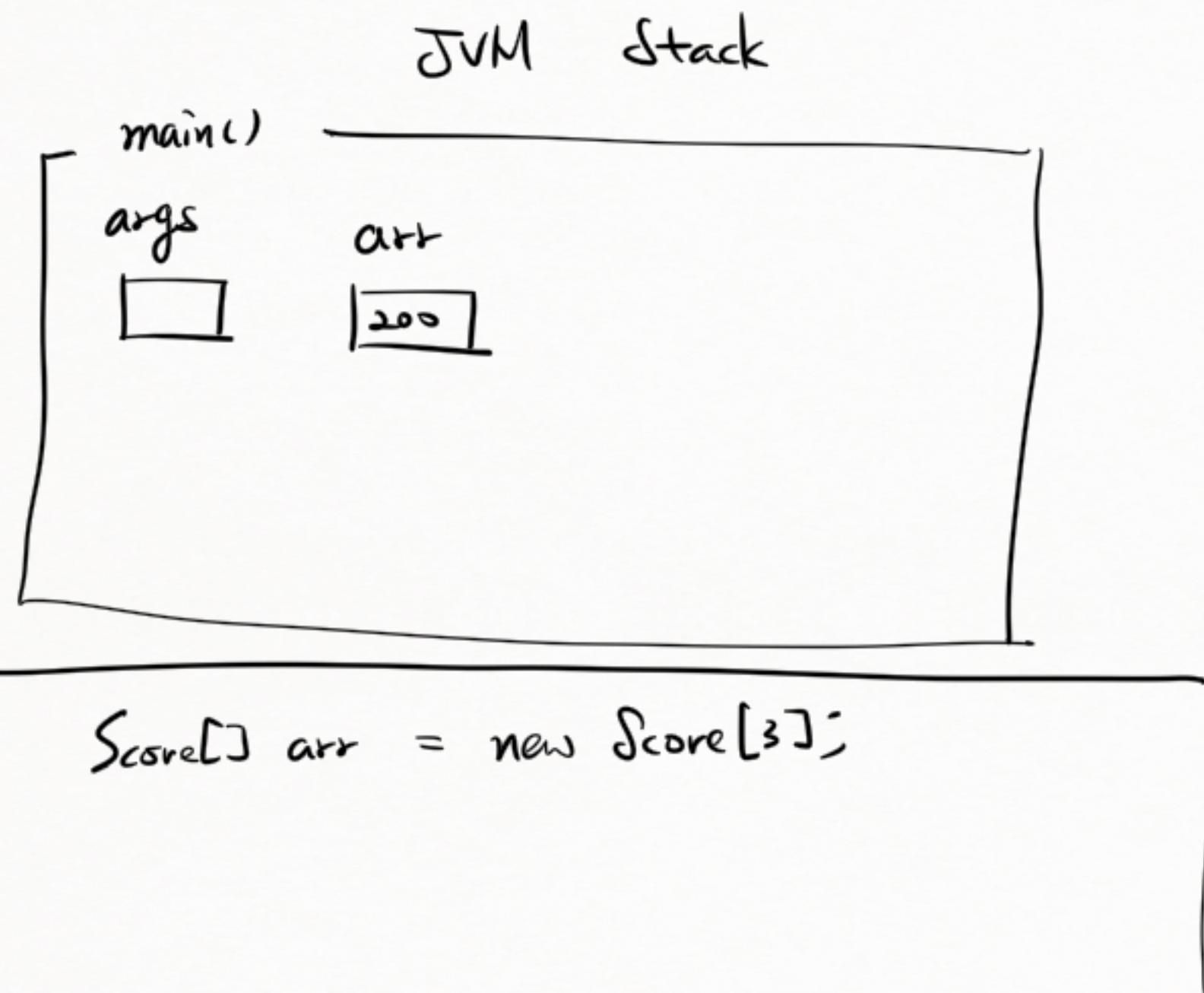
s_3 | 1100

200	name	kur	...

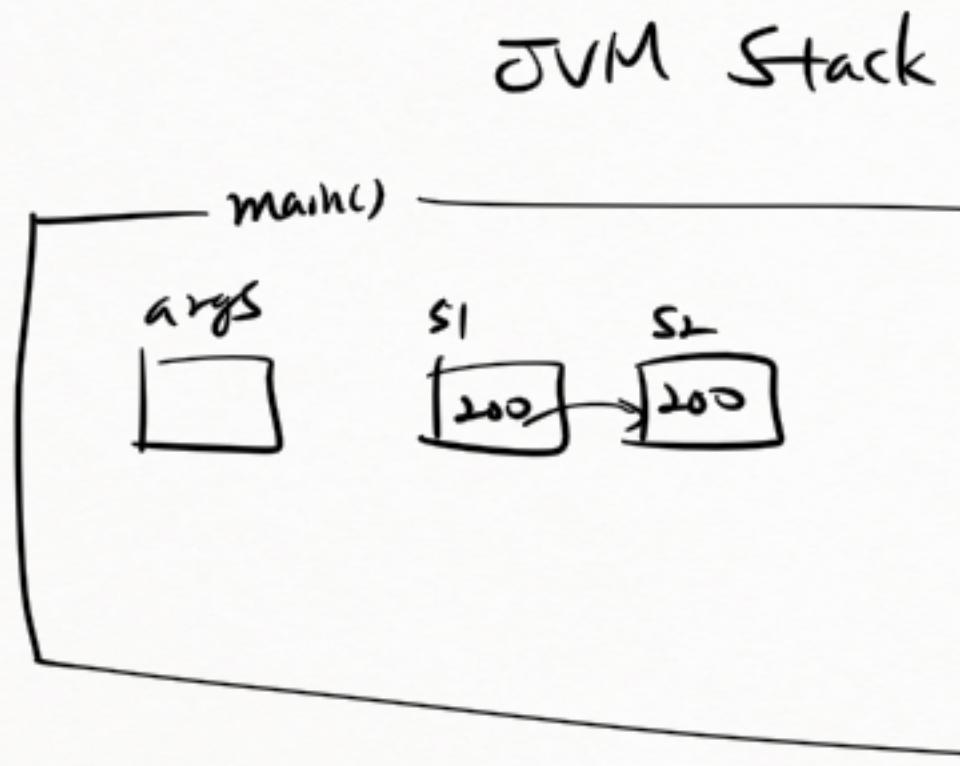
300	name	kur	

1100	name	kur	

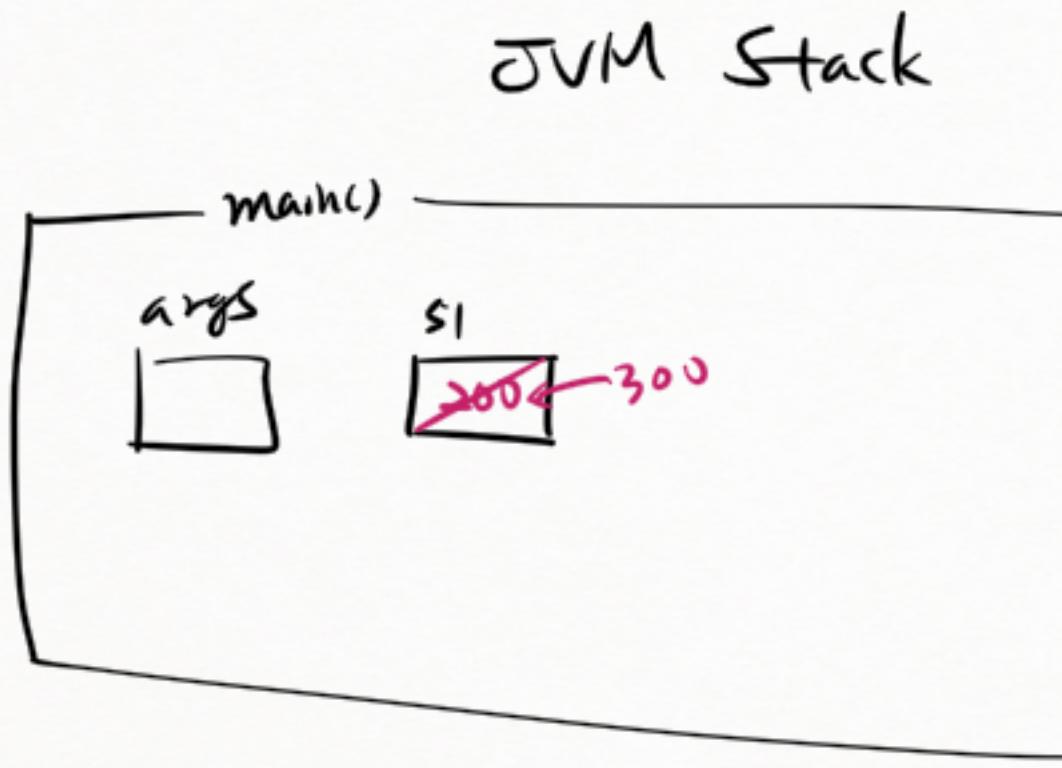
* com.eomcs.oop.ex01.Exam0220



* com.eomcs.007. ex01. Exam0310



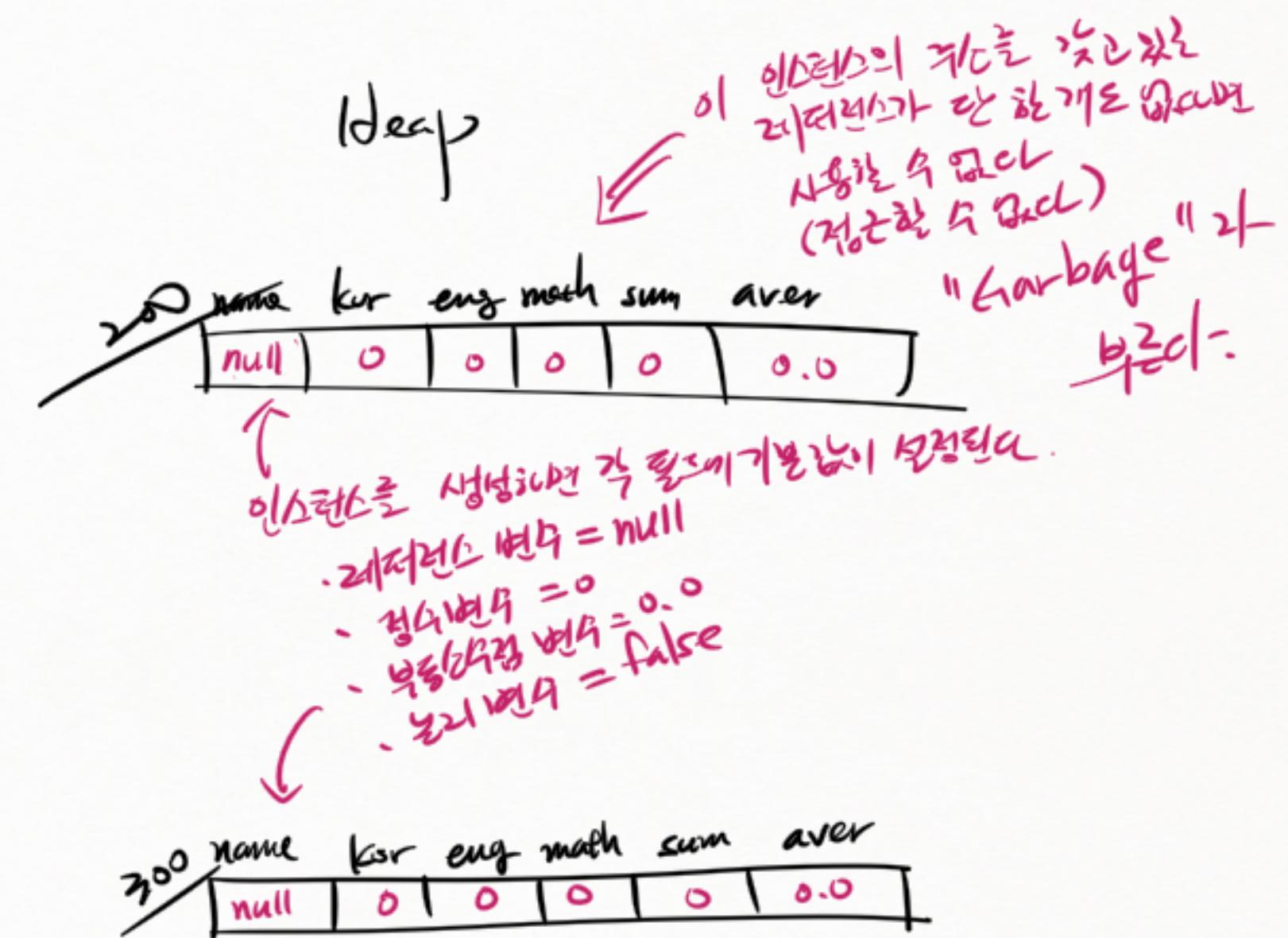
* com.eomcs.06p. ex01. Exam0320



Score s1;

s1 = new Score();

s1 = new Score();



* com.eomcs.06p. ex01. Exam0330

JVM Stack



Score s1 = new Score();

Score s2 = new Score();

s2 = s1;

인스턴스 주소는
자기주소와 같지만,
인스턴스 주소는
다른 주소로 바뀝니다.



Heap

name	kor	eng	math	sum	aver
null	0	0	0	0	0.0

인스턴스를 생성하면 각 필드 기본값이 설정된다.

- 리퍼런스 변수 = null

- 정수 변수 = 0

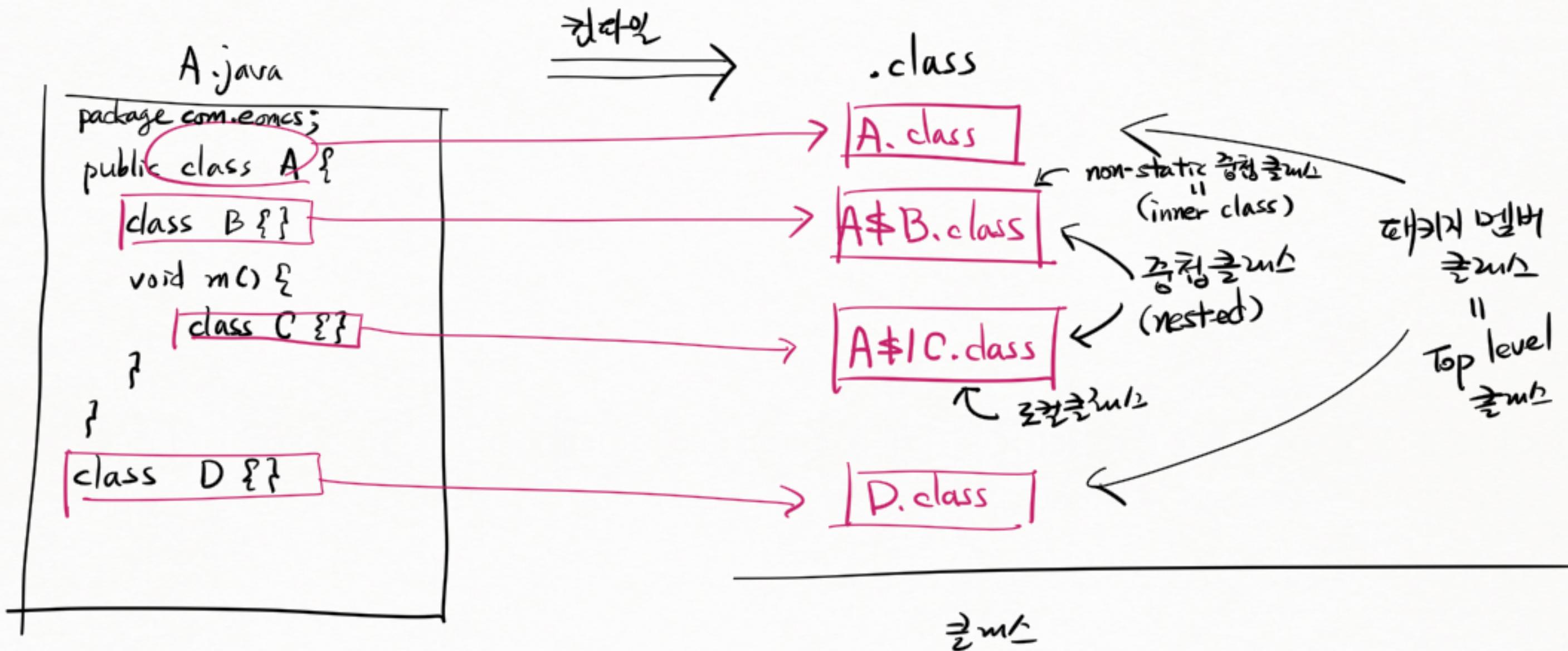
- 부동소수점 변수 = 0.0

- 논리 변수 = false

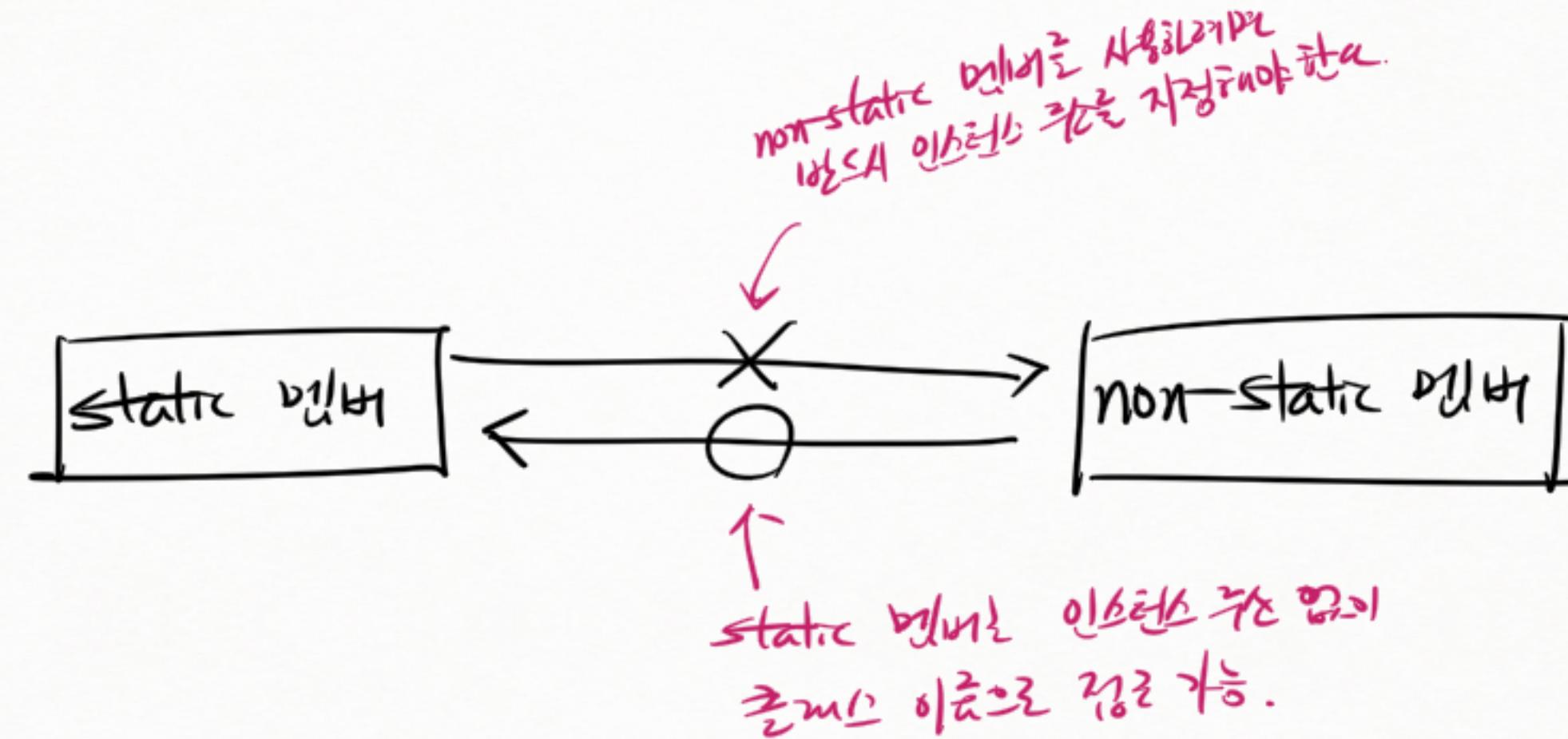
name	kor	eng	math	sum	aver
null	0	0	0	0	0.0

리퍼런스 카운트 개수가 0 이면
"가arbage(Garbage)"가 된다.

* 클래스 복수와 .class 파일



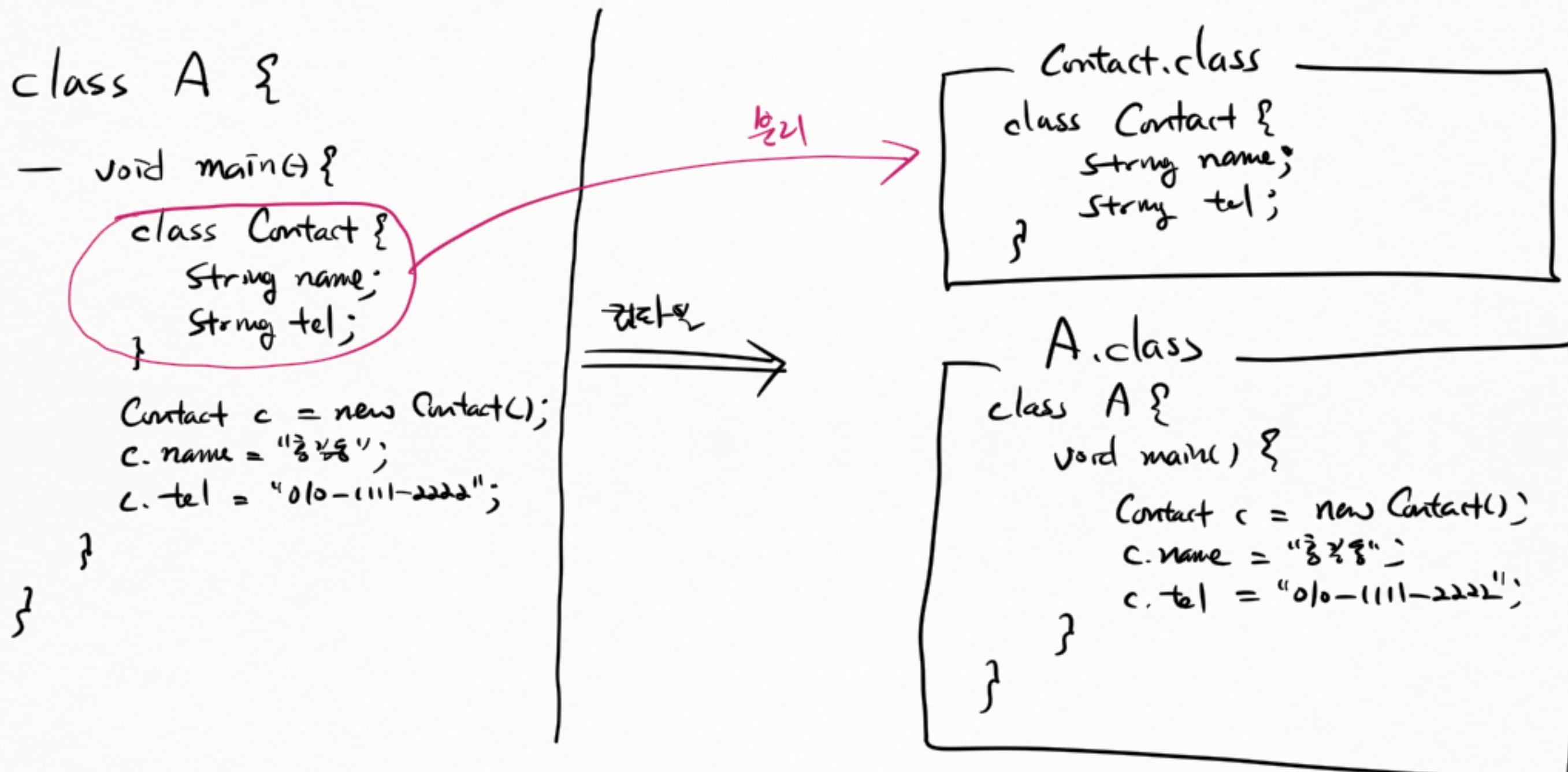
* static 멤버와 non-static 멤버 간의 접근 규칙



* 대기지 멤버 구조 : (default) vs public



* 디자인 원칙과 연결성을 정의



* 클래스 문법 - 사용자 정의 데이터 타입

① 메모리 유형설정

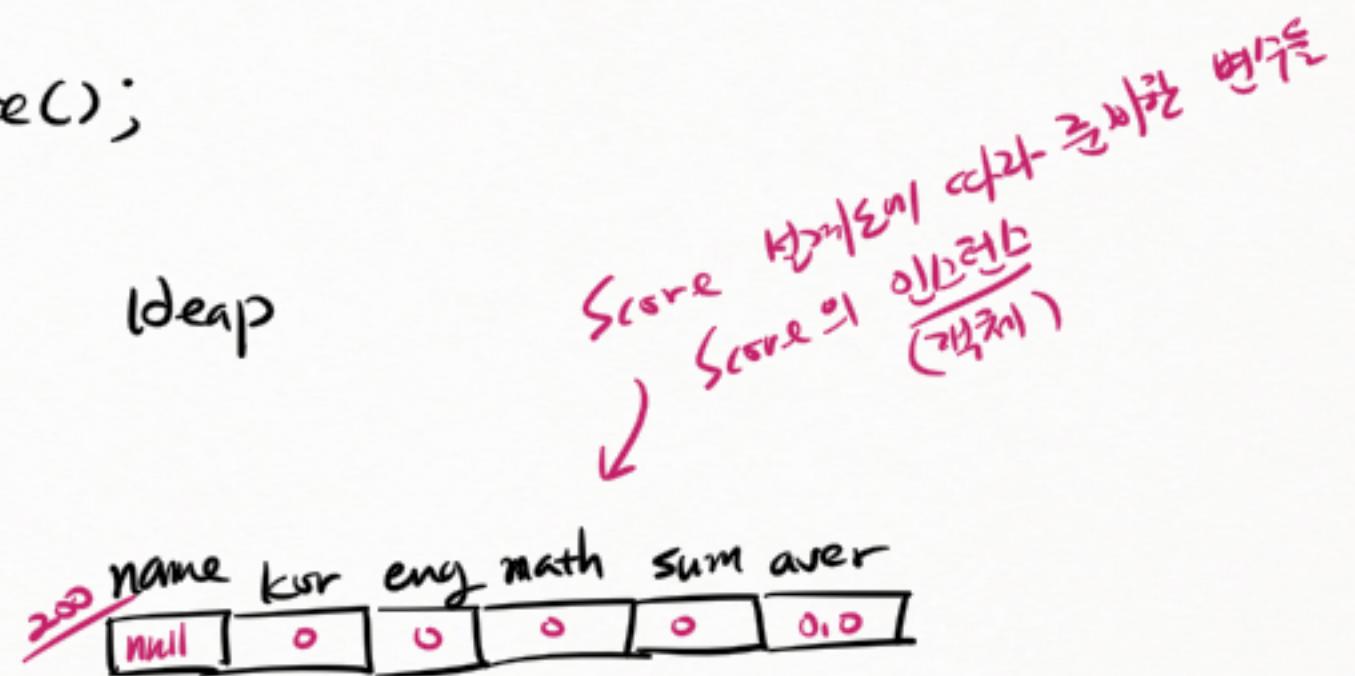
class Score {

```
String name;  
int kor;  
int eng;  
int math;  
int sum;  
float aver;
```

}



Score s = new Score();



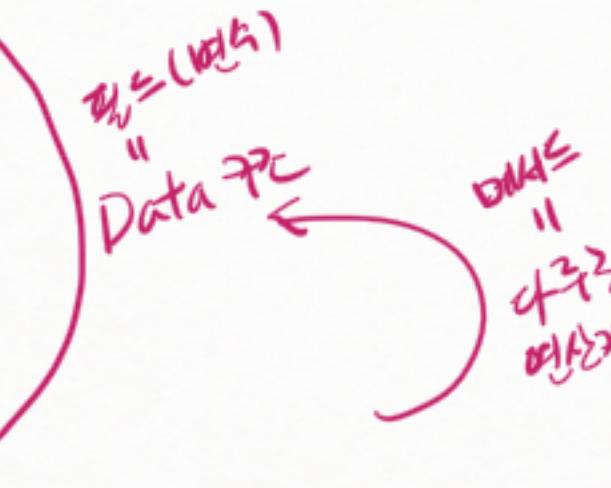
* 클래스 문법 - 사용자 정의 데이터 타입

② 데이터 구조를 설계하고 그 데이터를 다루는 인산자를 정의
스라워 메서드

class Score {

```
String name;  
int kor;  
int eng;  
int math;  
int sum;  
float aver;
```

```
static void calculate(Score score){  
    score.sum = score.kor + score.eng + score.math;  
    score.aver = score.sum / 3f;  
}
```



```
int a;  
a = -100;  ⇔  
a++;  
↑ ↑  
데이터 인산자
```

Score s = new Score();

```
s.name = "홍길동";  
s.kor = 100;  
s.eng = 90;  
s.math = 80;
```

Score.calculate(s);

설계문장 → 인산자(Operator) 대입문자

설계문장 → 메서드(method) = 함수(function)

설계문장 → 메시지(message)

* 클래스 문법 - 사용자 정의 데이터 타입

③ 레이더 구조를 설계하고 그 레이더를 다루는 인수자를 정하라

```
class Score {
```

```
String name;  
int kor;  
int eng;  
int math;  
int sum;  
float aver;
```

~~static void calculate()~~

`this.sum = this.kar + this.ang + this.math;`
`this.aver = this.sum / 3f;`

3

non-static 멤버

int a;
a = -100; ⇔
a++
↓ ↓
피연산자 연산자

기준의
연산자를
사용하는 문법은
더 비슷하다

Score s = new Score();

S. name = "李四";
S. kor = 100;
S. eng = 90;
S. math = 85;

~~Score.calculate(s)~~

`s.calculate()`

인스턴스 주소를 메시드에 전달

* 클래스 정의

데이터를 저장할
메모리 구조 설계 ← 인스턴스 필드 선언

+
새 데이터 유형을 다른
연산자 정의 ← 메서드 정의

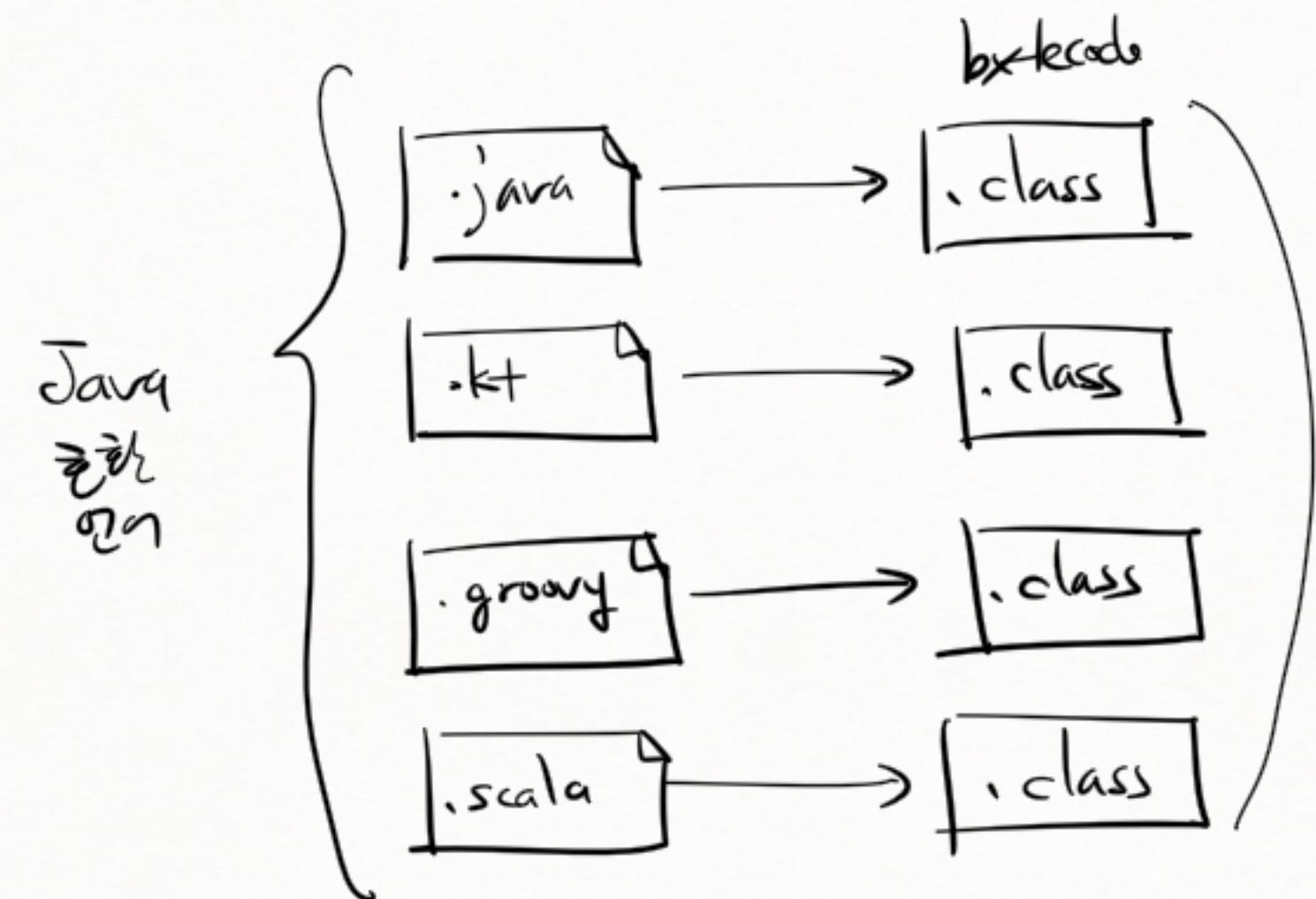
* modifier

final
static
private
⋮

int $i = 100;$

이들이 무엇을 추가하느냐에 따라-
위에 선언한 변수(또는 메서드, 클래스)의
성질(특성)을 바꾼다
||
변경을 가하는 명령 (modifier)
변경자 | 흡정자 | 제한자-

* 자바 향한 언어

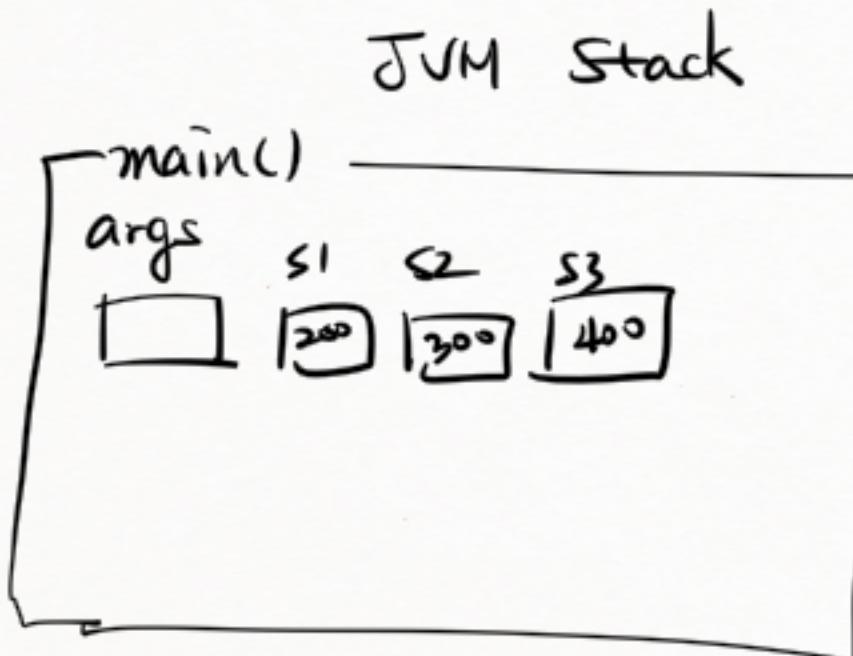


* 인스턴스 변수(필드) : - oop. ex03, Exam 0110
non-static

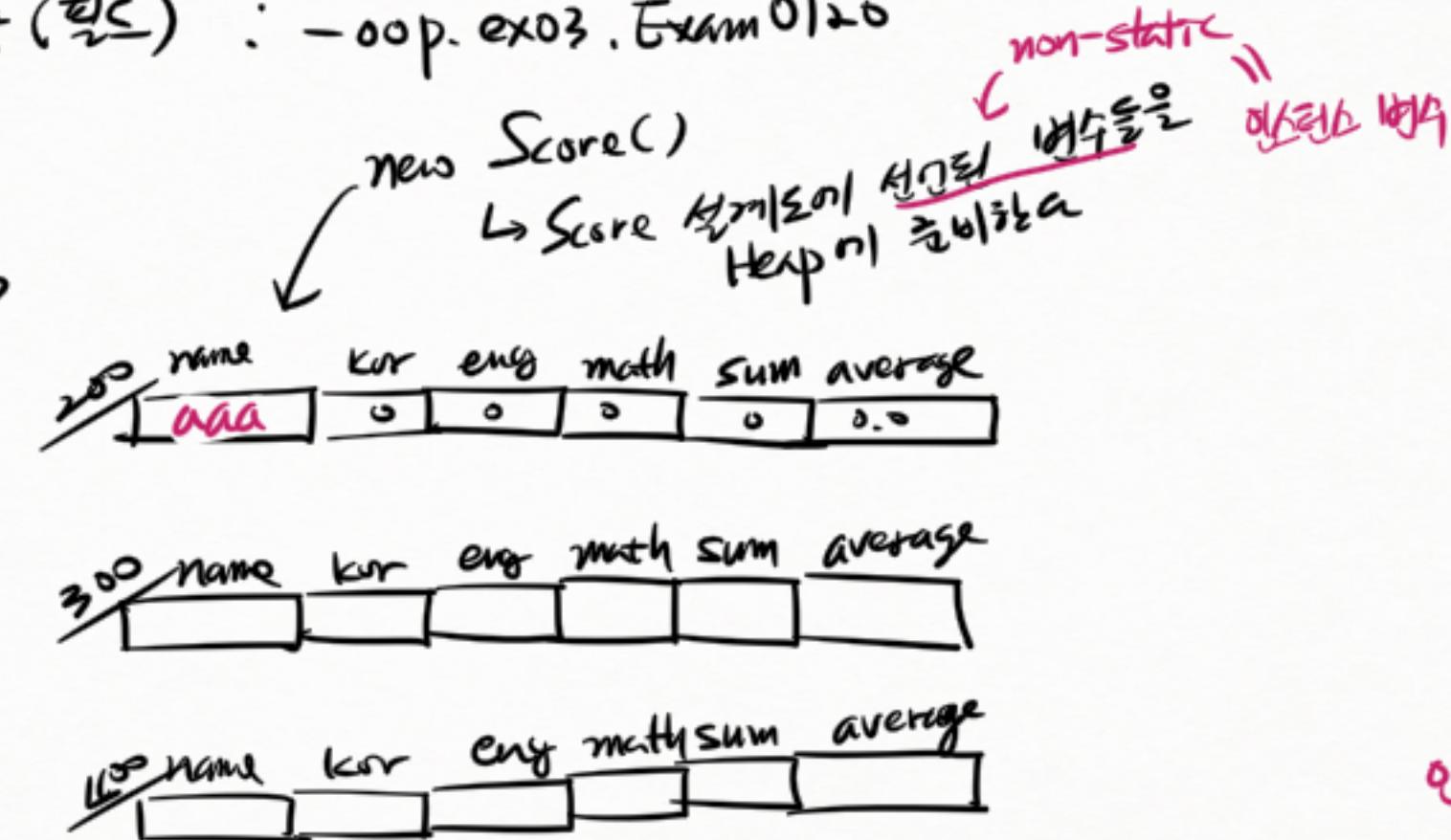


* 인스턴스 변수(필드) : - oop. ex03, Exam 0120

non-static

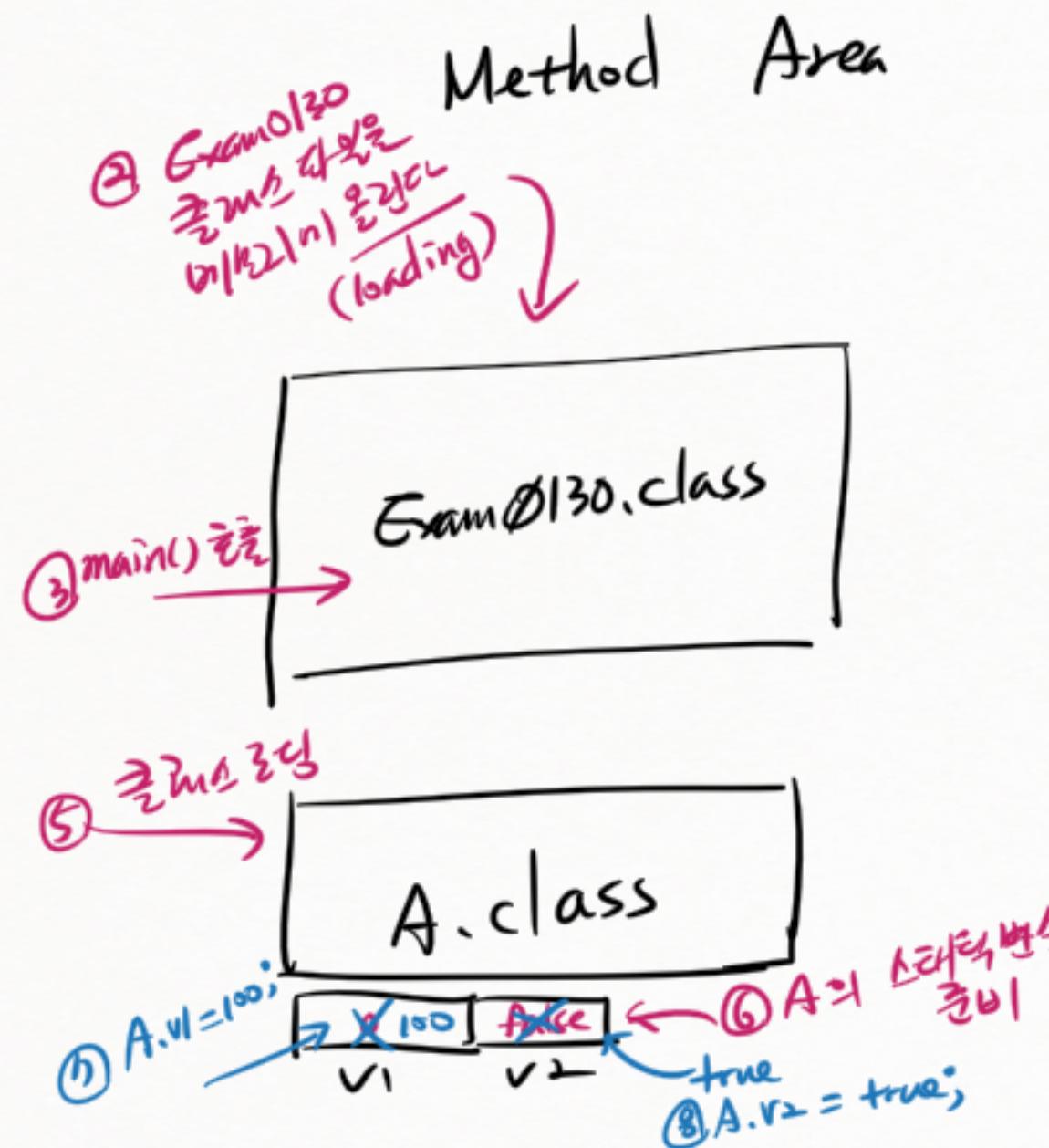


Heap



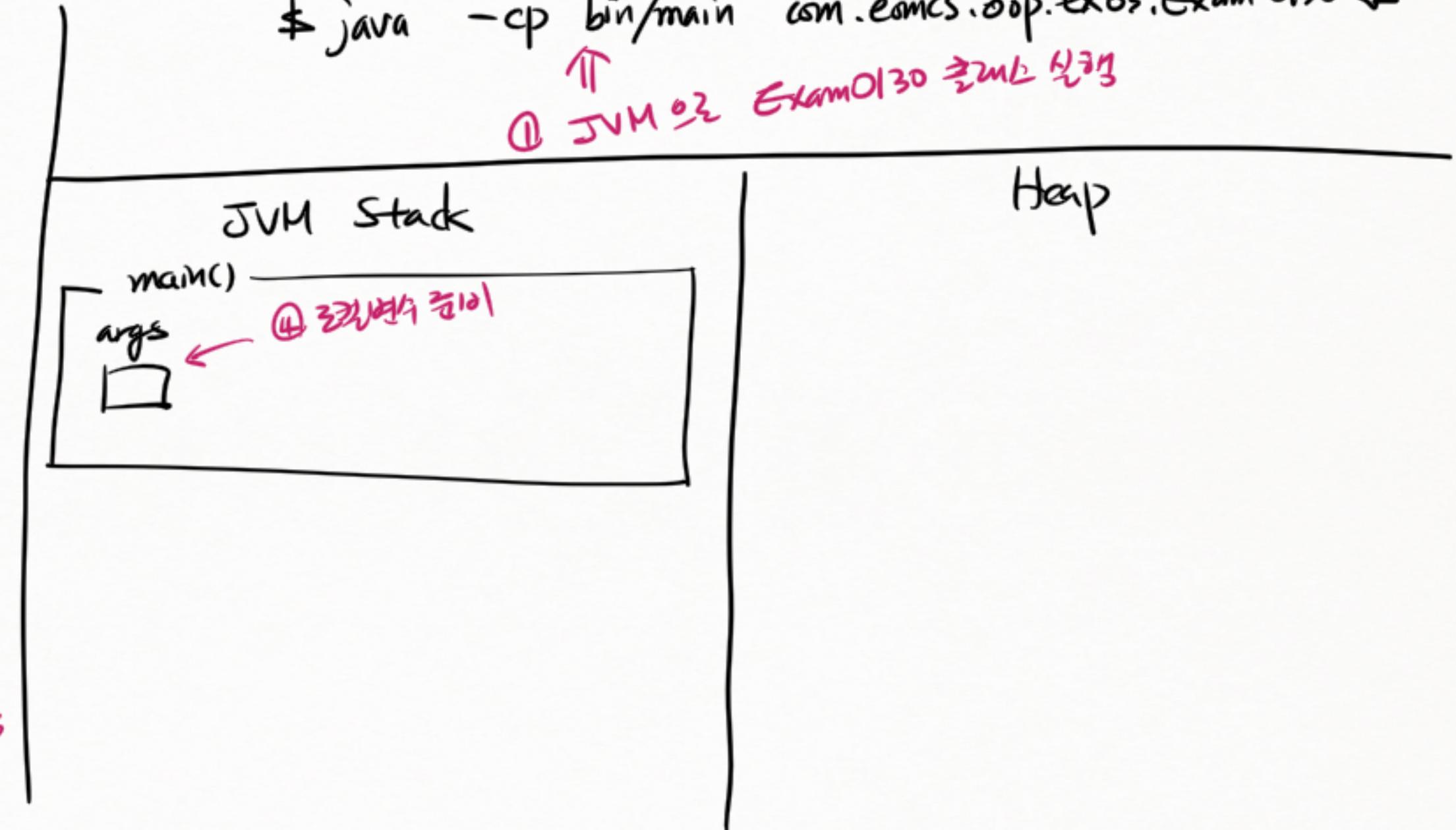
s1.name = "aaa"
mm
200
↑
인스턴스 필드

* 정적 변수(필드) : - oop.ex03.Exam0130
static

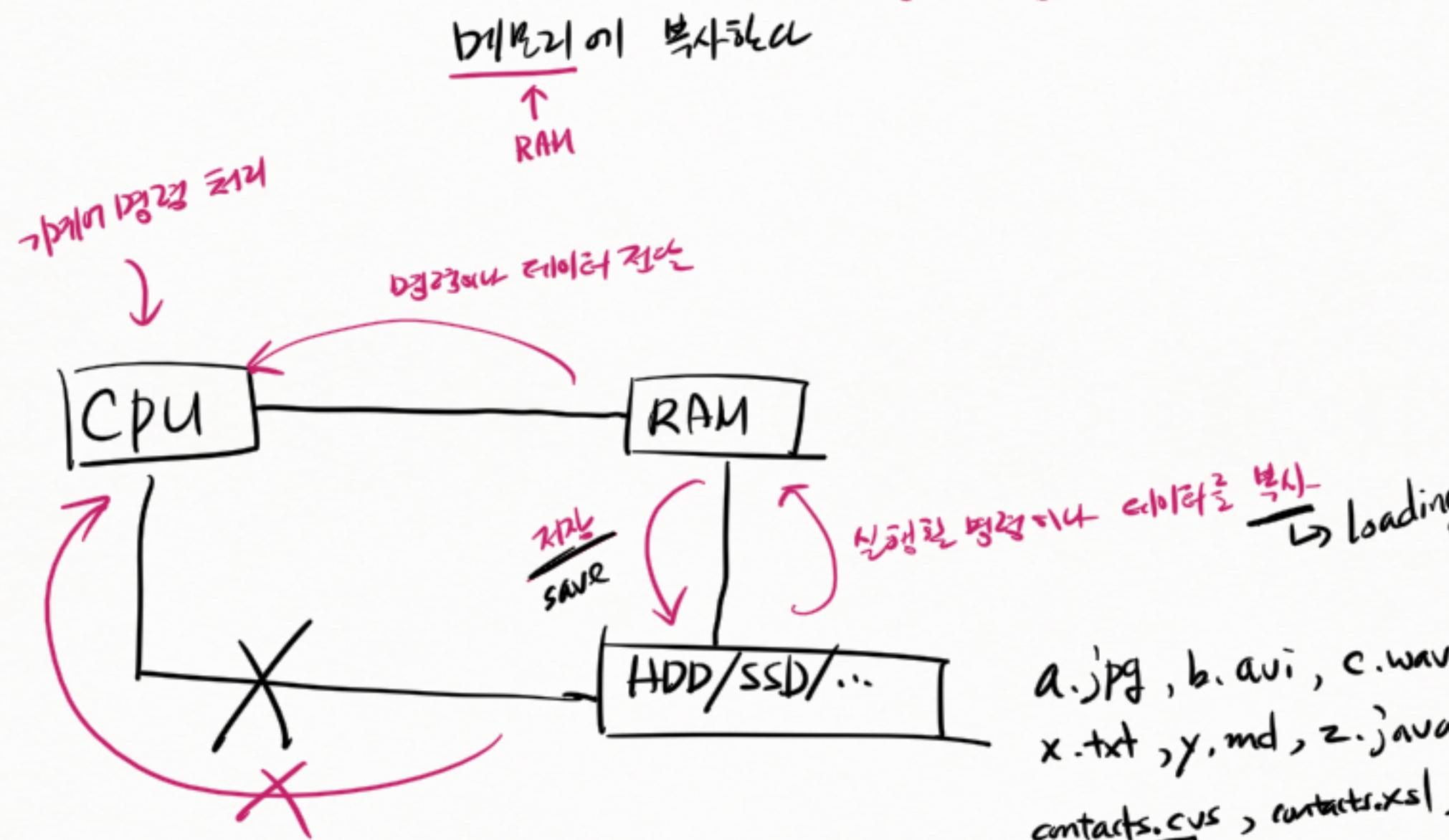


\$ java -cp bin/main com.eomcs.oop.ex03.Exam0130 ↳

① JVM으로 Exam0130 정적 변수 실행 ↳



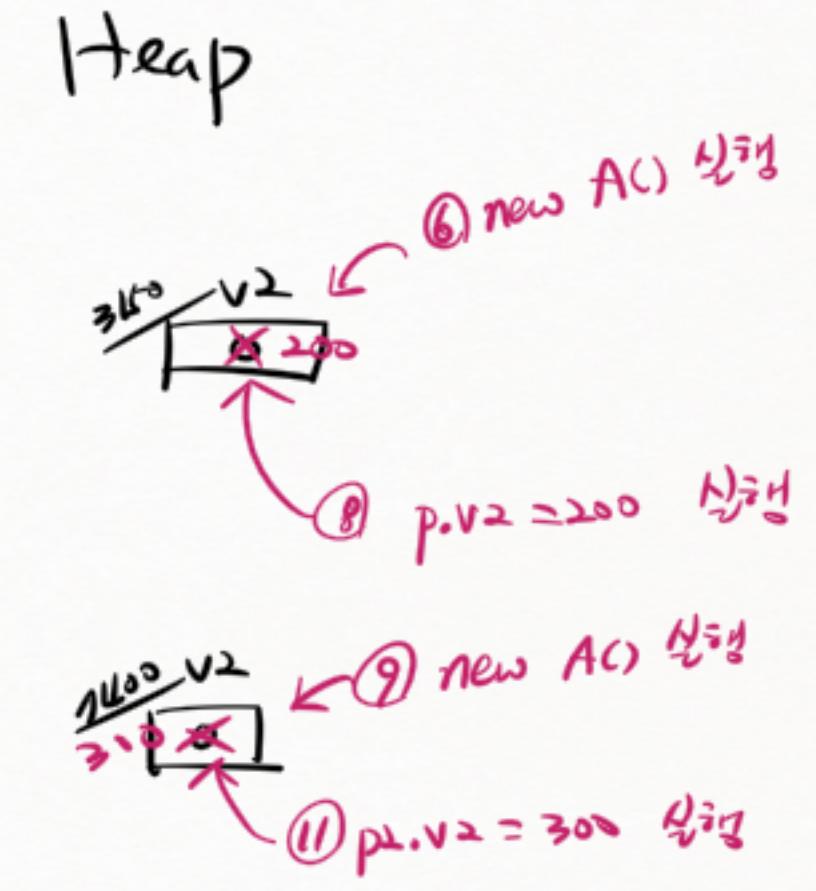
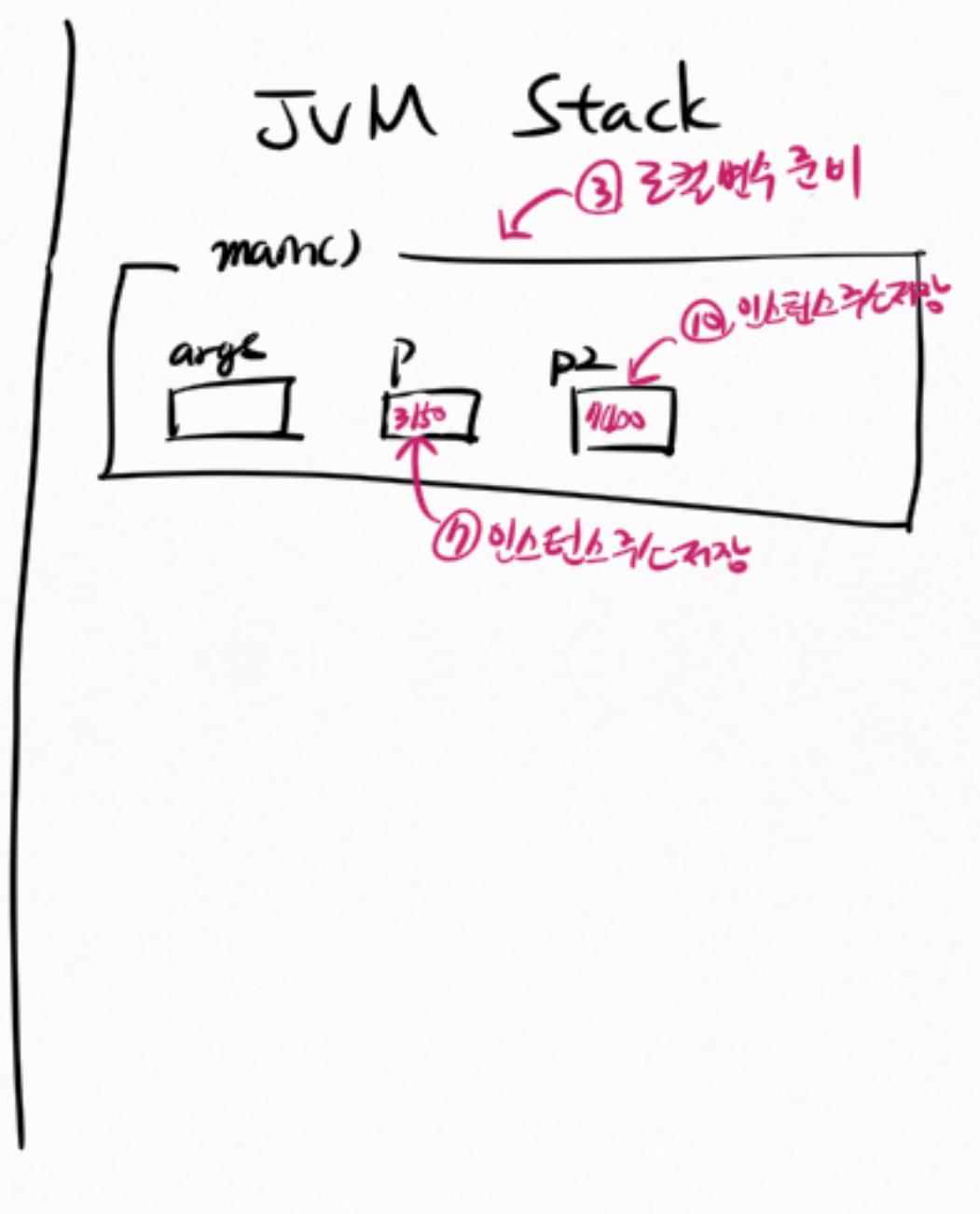
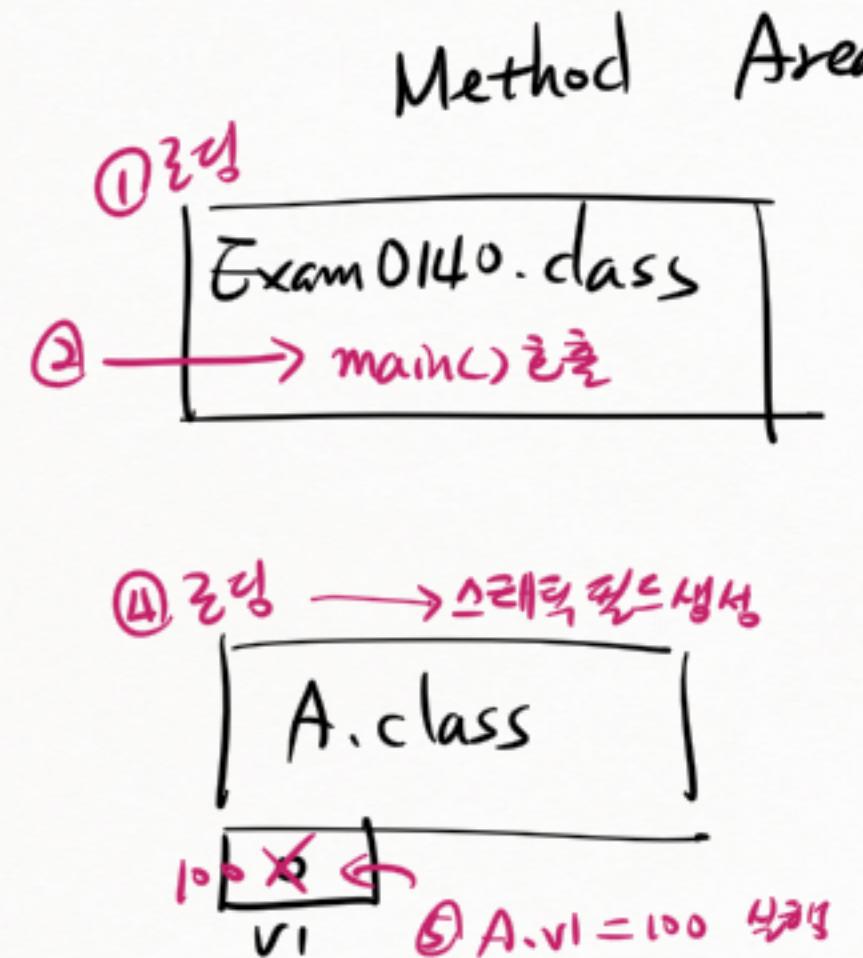
* 클러스터링 (clustering)



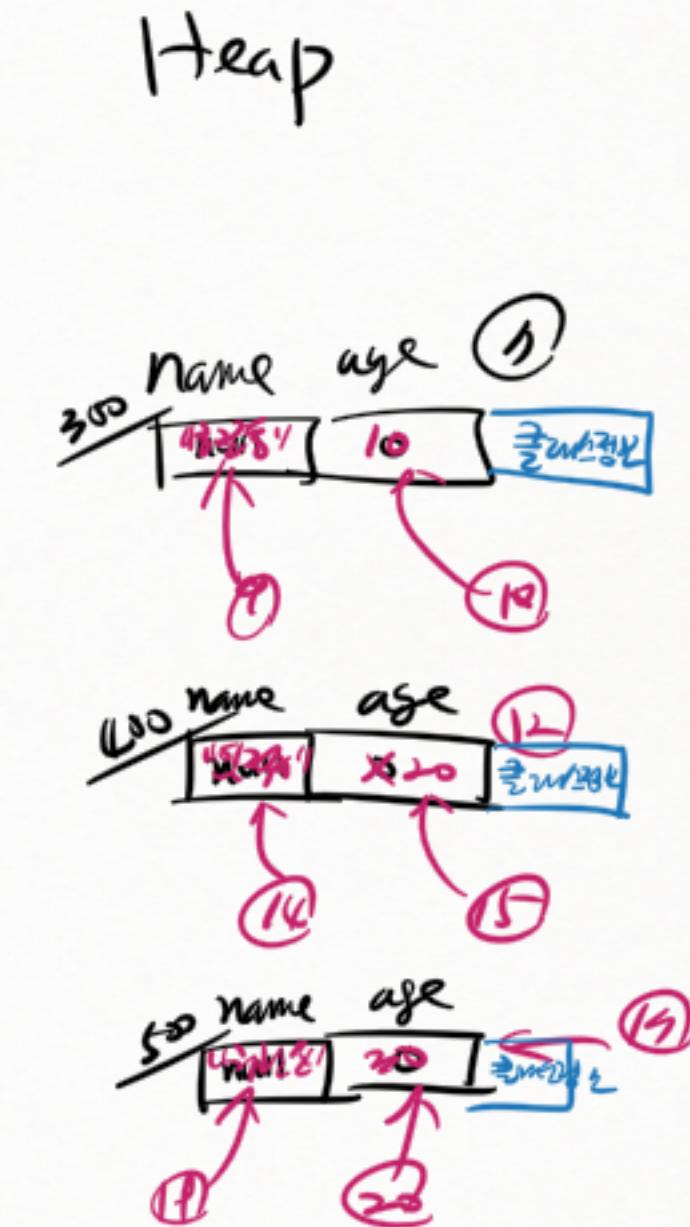
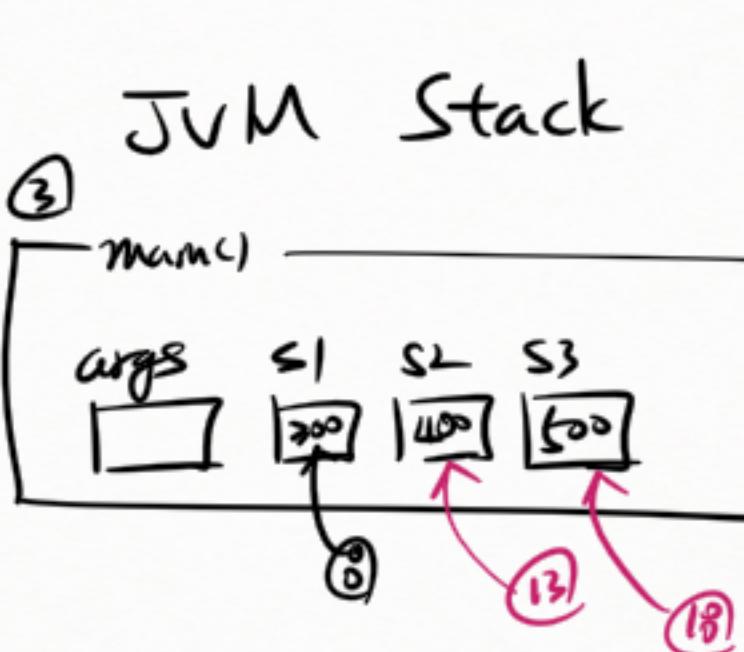
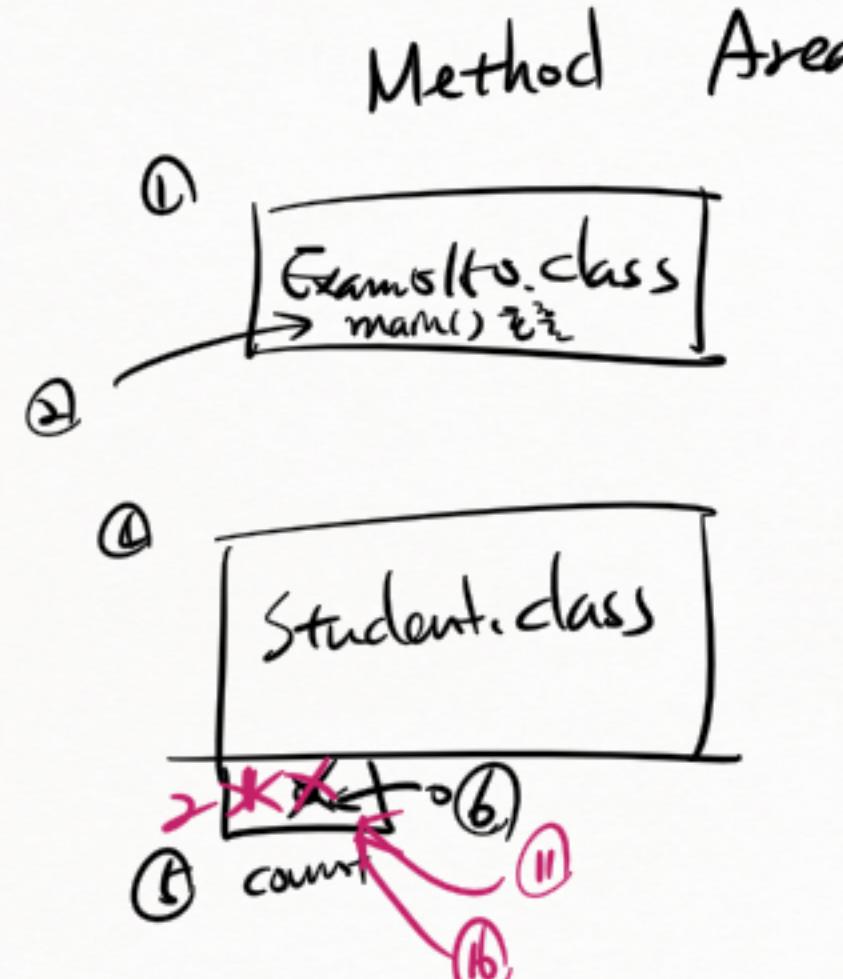
$128 \times 14 \text{ collet } \frac{2}{2} \xrightarrow{\text{부기}} \text{loading}$

- a.jpg, b.avi, c.wav, d.mp3 ...
- x.txt, y.md, z.java
- contacts.csv, contacts.xls, ...
- .class

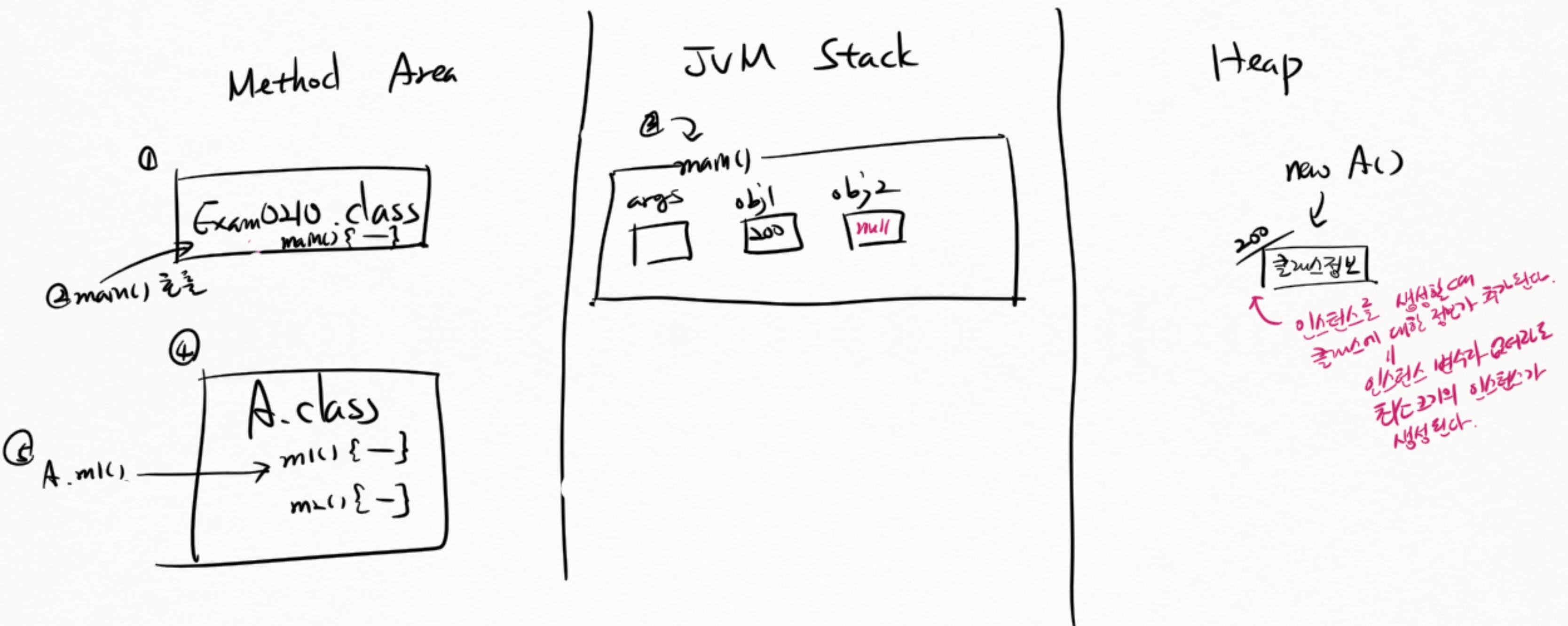
* 글래스 변수와 인스턴스 변수 : oop. ex 3. Exam0140



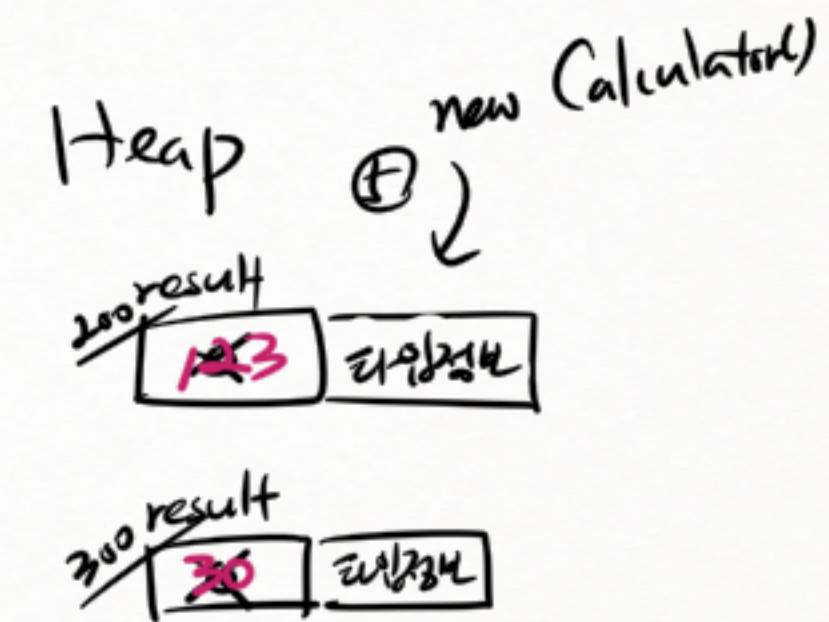
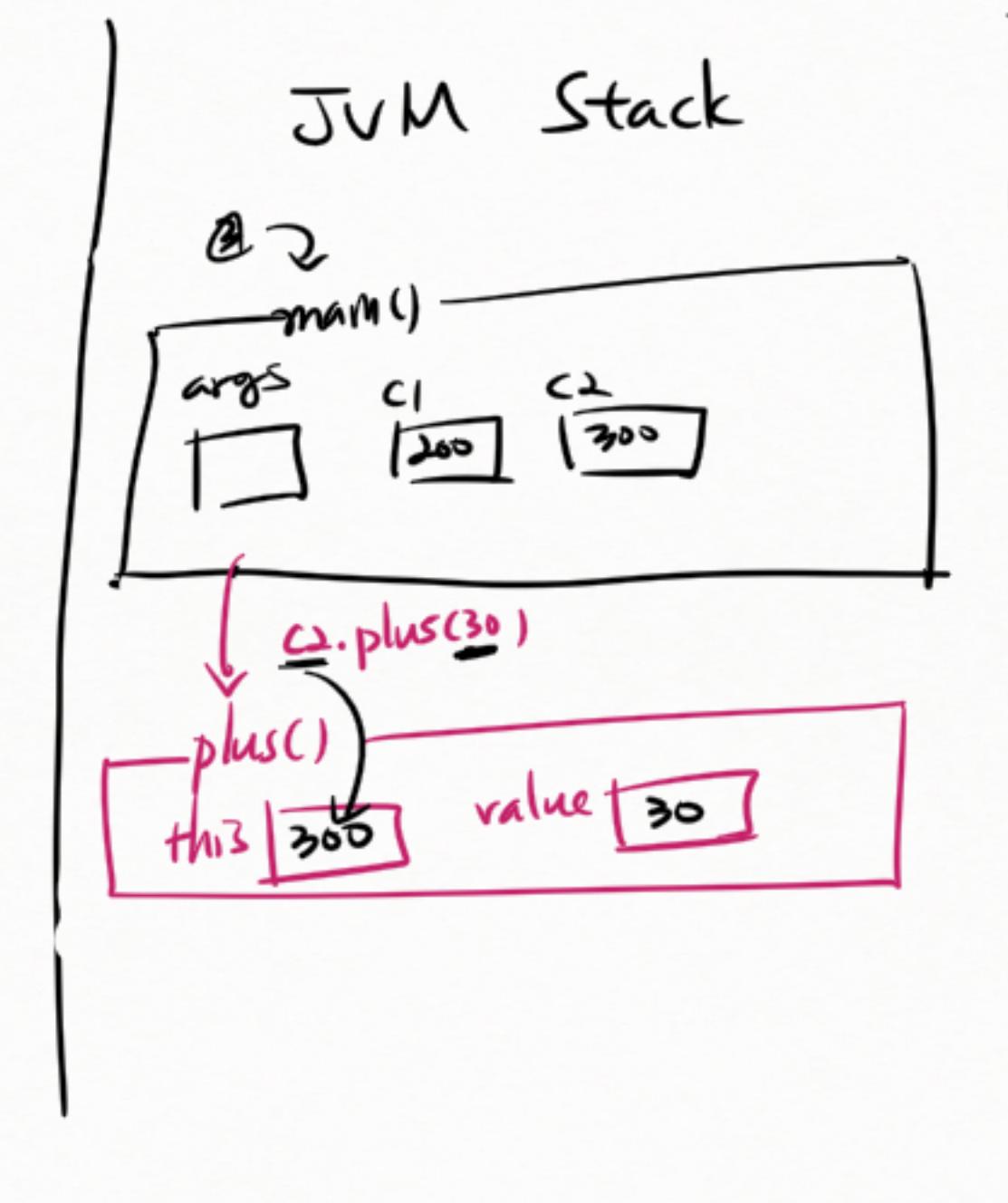
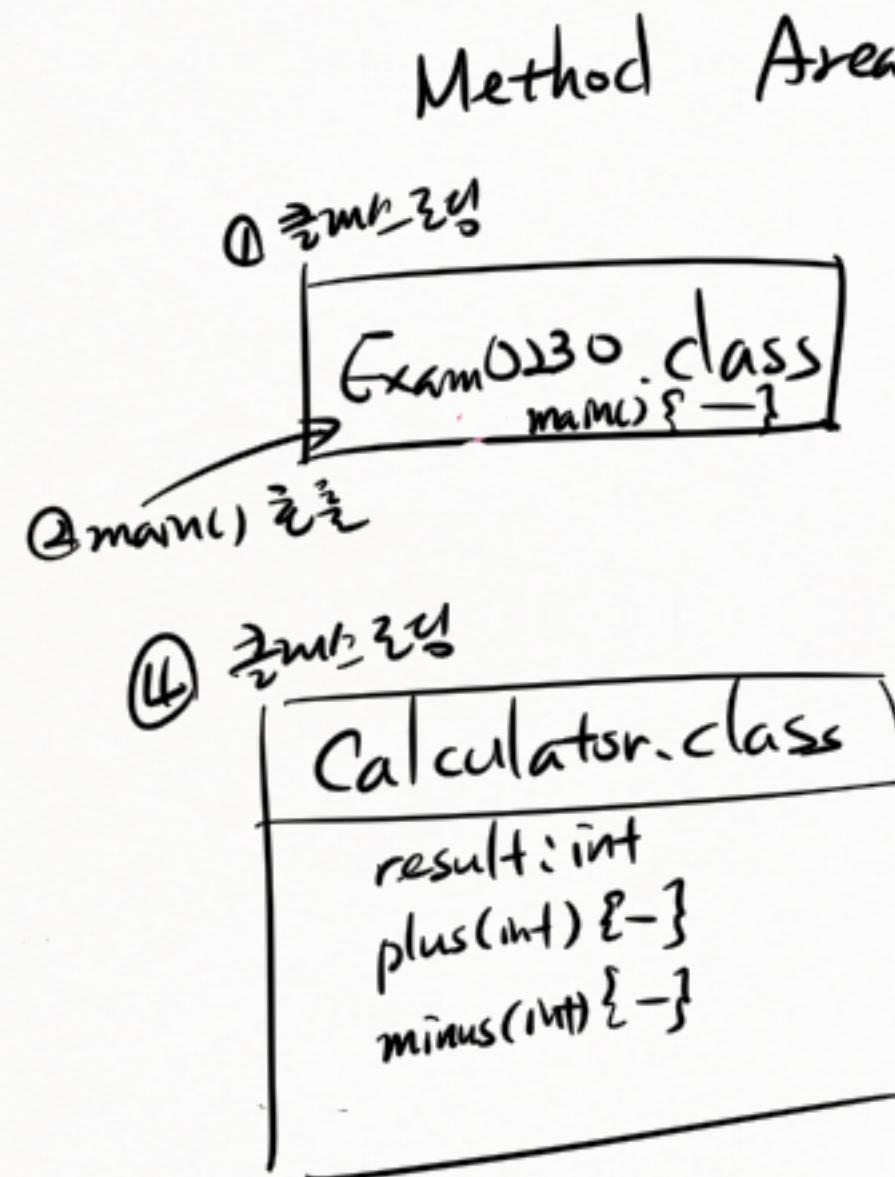
* 주소 변수와 인스턴스 변수 : oop. ex03. Exam0150



* 접근스 박스와 인스턴스 메소드 : oop. ex03. Exam0210



* 디스터스 멤버(변수와 메서드) : oop. ex03. Exam0230



* 생성자

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

~~Score()~~

) 생성자(Constructor)

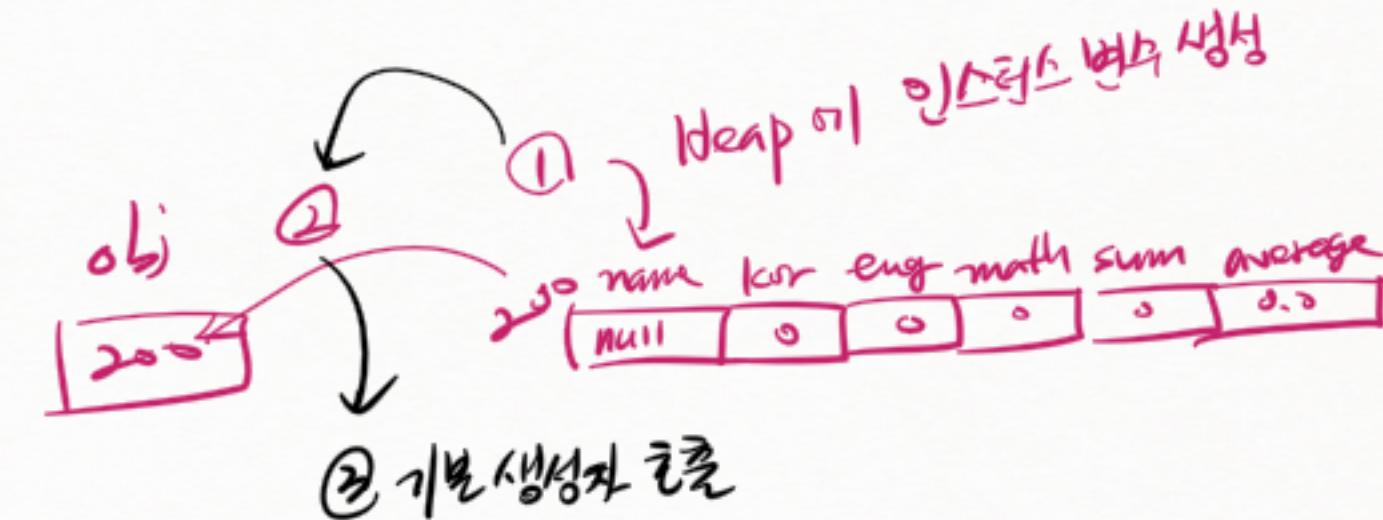
자바비전이 있는 생성자
"default constructor"

Score obj = new Score();

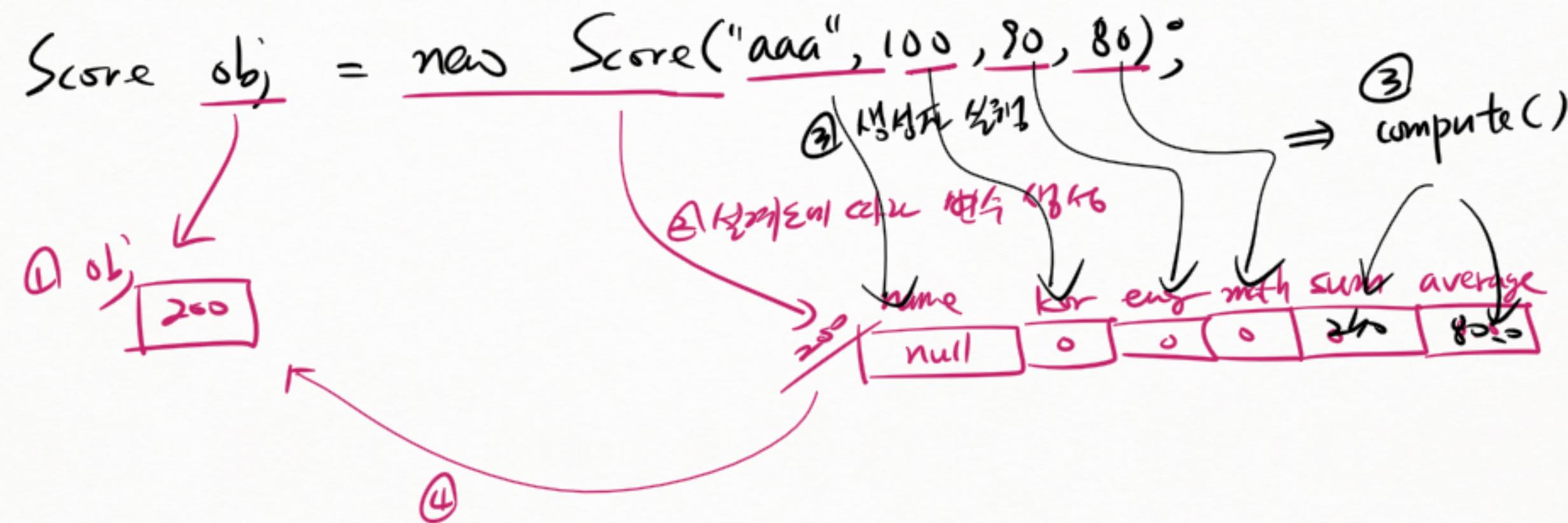
↑
클래스명
↑
자동으로 호출될 생성자를 지정.

타입 primitive type
class interface enum

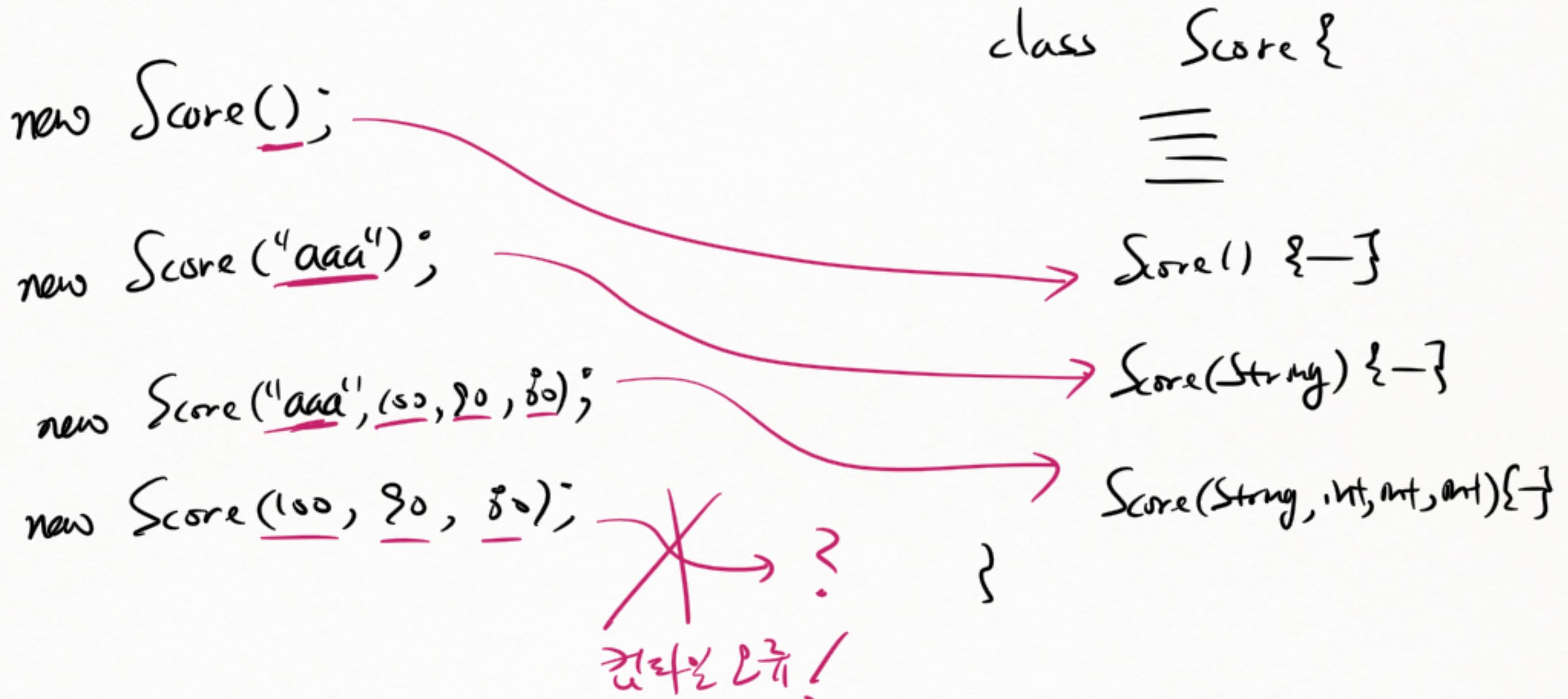
리퍼런스
" (포인터)



* 생성자 활용 \Rightarrow 인스턴스 변수는 초기화된다.

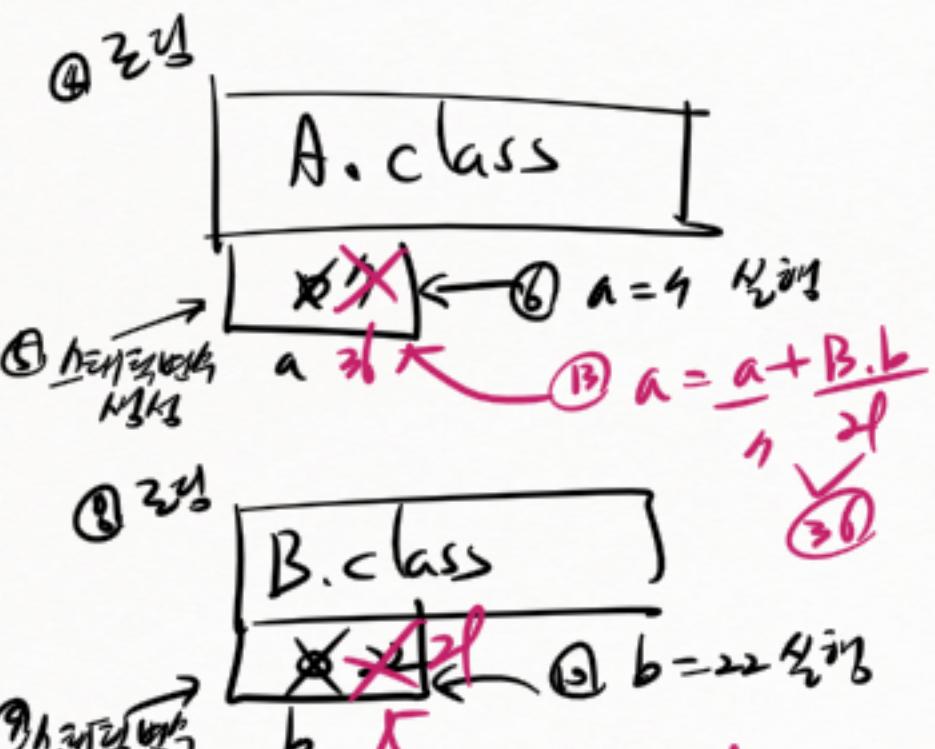
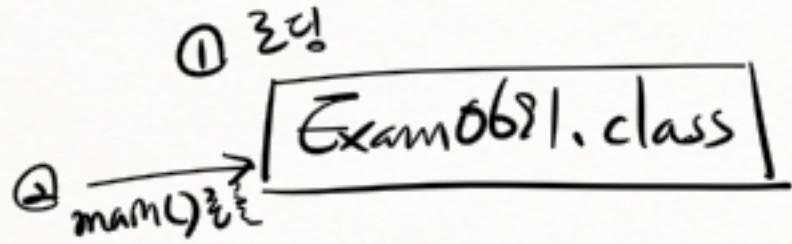


* 헤더 파일 생성자를 결정하는 방법

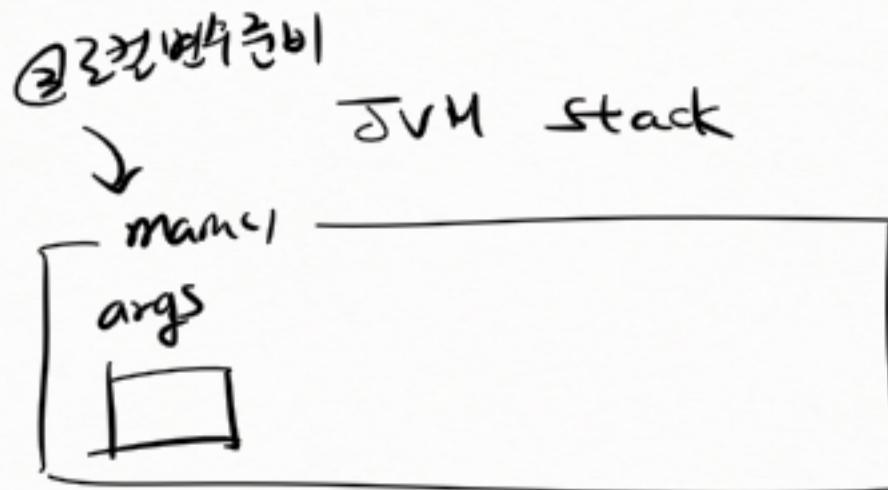


oop. ex03. Exam0910

Method Area



$$\begin{aligned} & \text{⑪ } b = \cancel{22} + \frac{A.a}{\cancel{22}} \\ & \quad \checkmark \end{aligned}$$



console \rightarrow

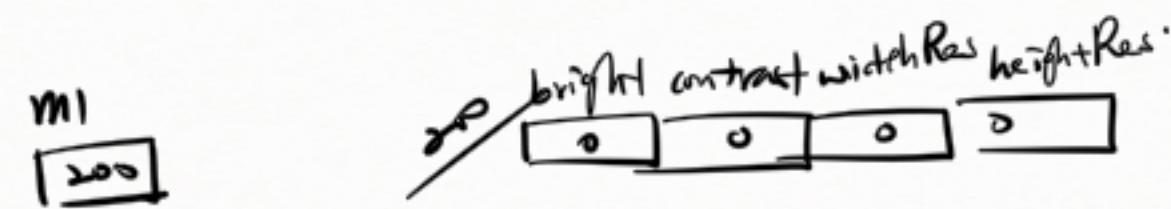
① \hookrightarrow $A.\text{static}\{ \}$

② $\xrightarrow{\text{static }} \text{ static } \frac{\text{정적 }}{\text{변수}}$

⑪ \hookrightarrow $B.\text{static}\{ \}$

⑫ $\xrightarrow{\text{static }} \text{ static } \frac{\text{정적 }}{\text{변수}}$

`new Monitor()`



m1.display();

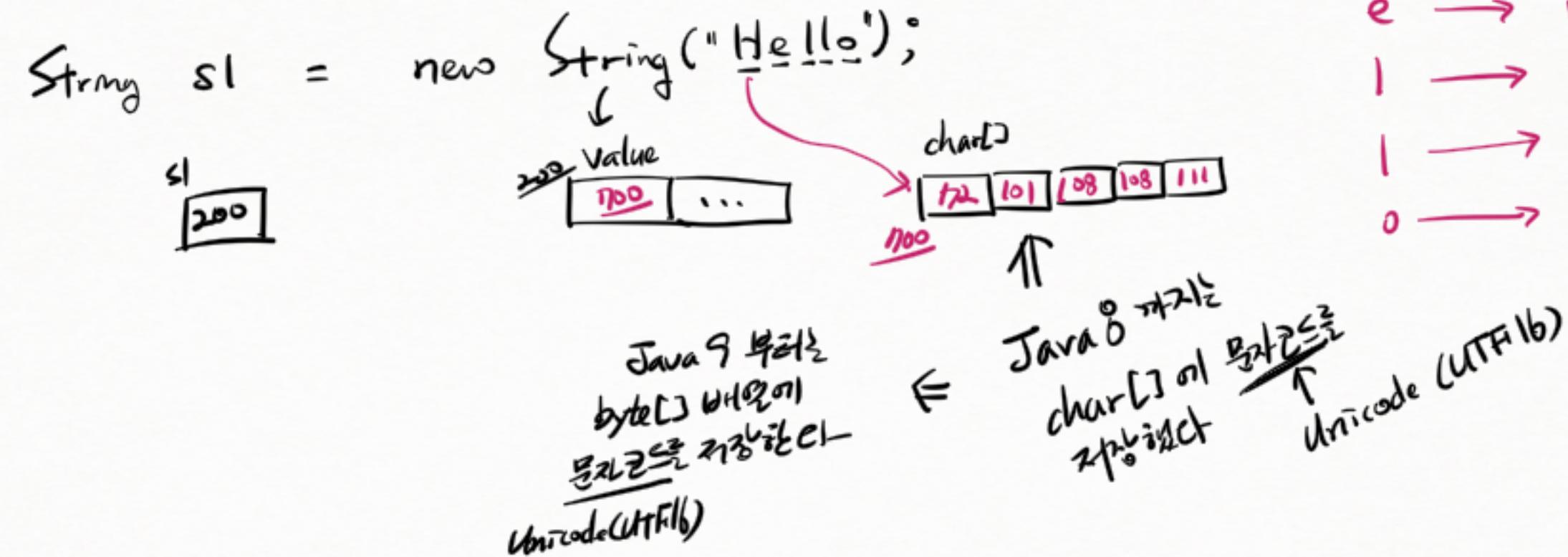
200

this



* String 클래스의 생성자

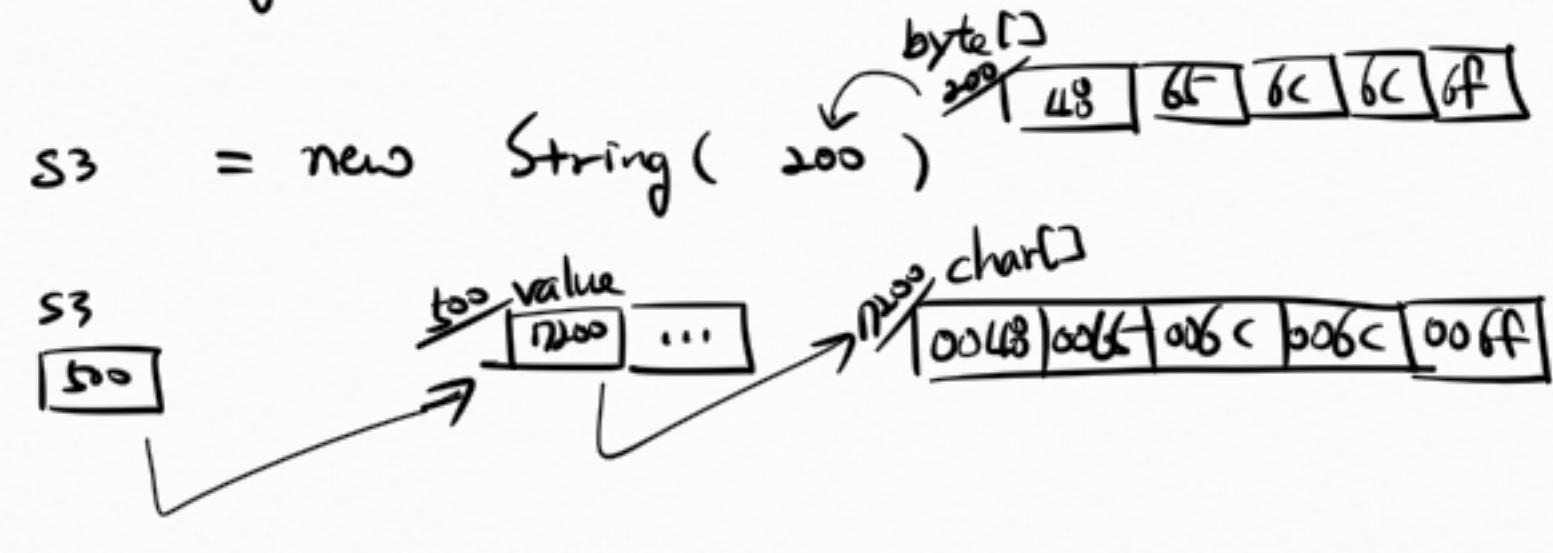
String s0 = new String();
기본 생성자 (default constructor)



문자	JVM이 사용하는 (charset)	Unicode 값 (UTF-16)
H	→	72 (0x48)
e	→	101 (0x65)
l	→	108 (0x6C)
l	→	108 (0x6C)
o	→	111 (0x6F)

* String 클래스의 생성자

String s3 = new String("Hello")



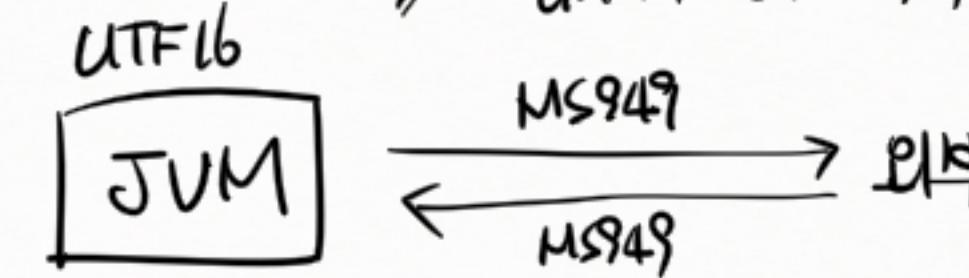
1 byte	2 byte Unicode
48	0048 (H)
65	0065 (e)
6c	006c (l)
6c	006c (l)
6f	006f (o)

* JVM 와 UTF16

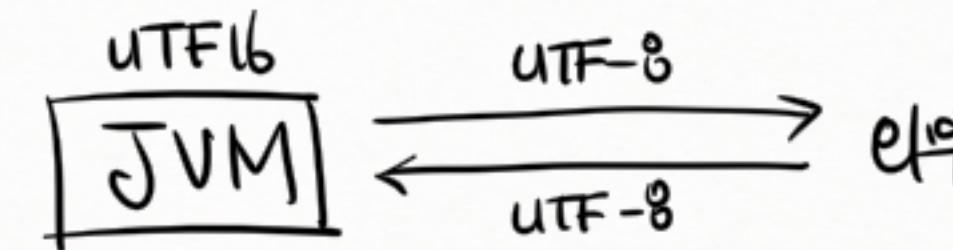
default characterset

↳ JVM이 외부의 문자코드를 내부문자로 변환할 때 사용하는 charset-
기본값은 charsets (문자변환기체)

Windows \Rightarrow

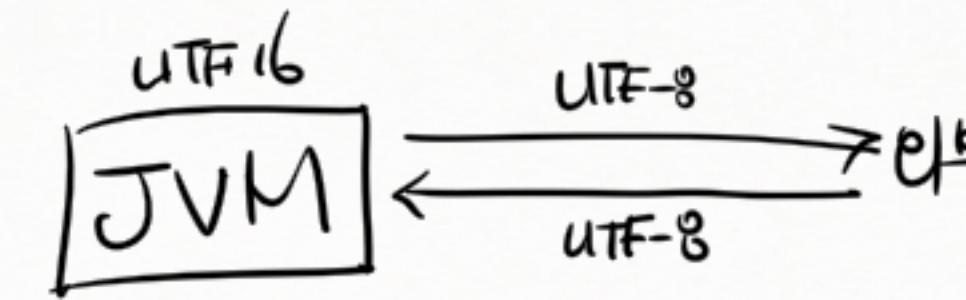


Unix/Linux \Rightarrow



Eclipse에서 \Rightarrow

App. 실행



* String 클래스와 생성자: oop.ex04.Exam0112

실행環境: Eclipse环境下

JVM이 외부 문자코드를 읽을 때
UTF-8이라 가정한다

String s

200

= new String();

200 value ...
IP 00bo 00a1 00b0 00a2 00b6 00ca 00b6 00cb
char[]

byte[]
'가' '나' '들' '을'

'가' '나' '들' '을'의 EUC-KR 코드
문자

문자	EUC-KR
가	B0A1
나	B0A2
들	B6CA
을	B6CB

String 클래스
내부로 배열에 들어가는 문자코드가 UTF-8 이라고

assumes.

↓ 문제 발생!

문자인 Unicode로 변환

한글이 깨진다.

'가'
→ B0A1 (EUC-KR)
→ ACOO (UTF-16)
→ EAB000 (UTF-8)

System.out.println(s);

* String 클래스와 생성자: oop.ex04.Exam0112

실행環境: Eclipse SDK 실행

JVM이 외부 문자코드를 읽을 때
UTF-8이라 가정한다

String s

200

= new String(, "EUC-KR");

200 value
1200 ...

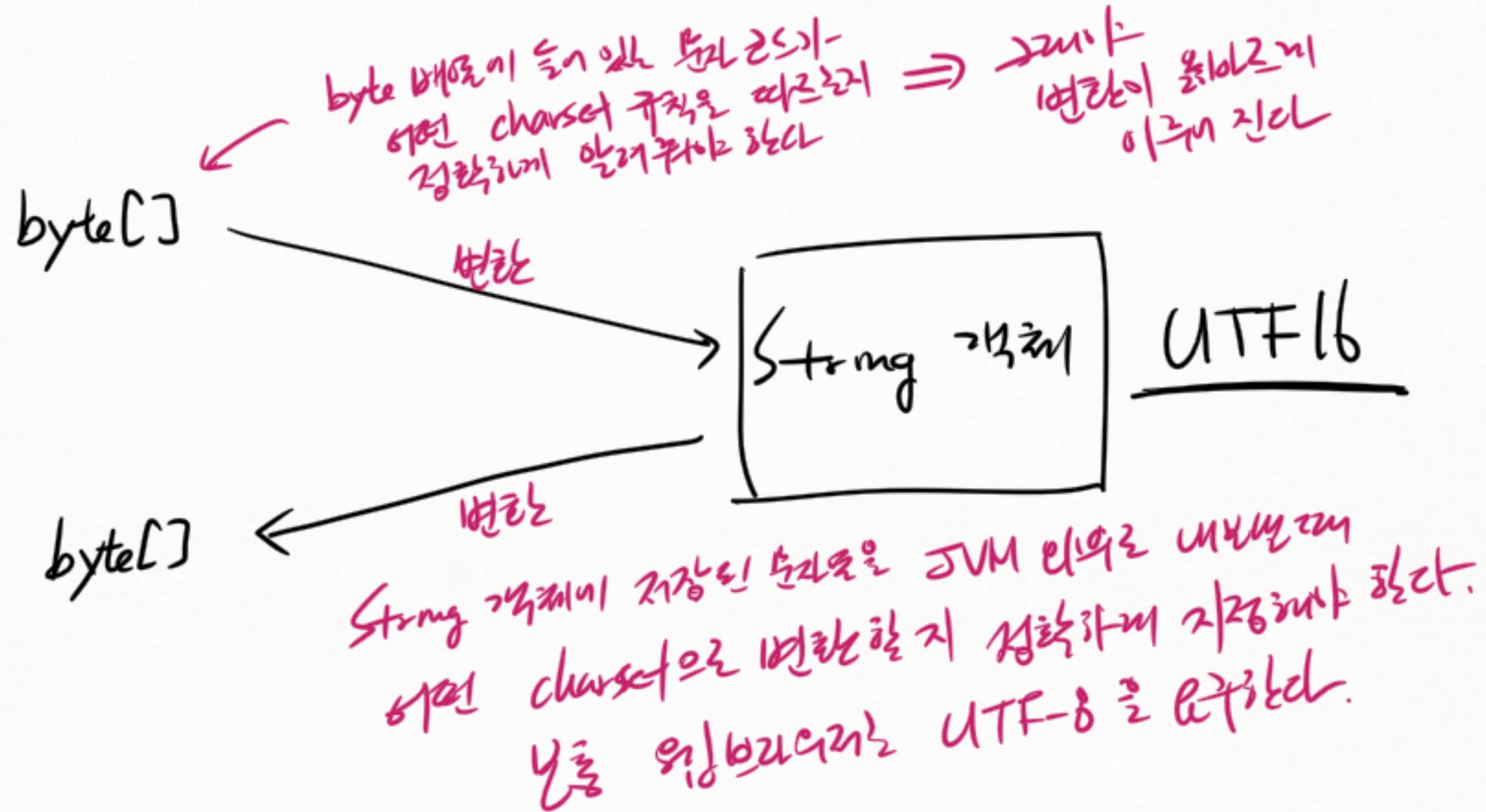
1200
AC00 AC01 B618 B625
char[]

byte[]
b0 a1 b0 a2 b6 ca bc cb
'가족동행'의 EUC-KR 코드
문자코드

byte[] 빼면 이들이 있는 값이
이전 문자집합의 규칙을 따르는지
정확하게 지정된다면
UTF16으로 정상적으로
변환하는 거다

{ EUC-KR \Rightarrow b0 a1 b0 a2 b6 ca bc cb }
UTF16 \Rightarrow AC00 AC01 B618 B625 }

* Java 한글 처리 주의!



① -Dfile.encoding 퀘션의 답은?

Windows : MS-949 ✓

Unix/Linux : UTF-8

byte[]

EUC-KR

b0	a1	b0	a2	b6	ca	b6	cb
----	----	----	----	----	----	----	----

"ABC가中华民族"



EUC-KR

(한글 111/224+α)
(한글 235자)

UTF-8, UTF-16

(한글 111/224+α)
(한글 X)

현재 byte[] 배열에 저장되어 있는
데이터 문자집합은? EUC-KR



MS-949 (EUC-KR+α)

UTF-16

'A' → 0041

'B' → 0042

'C' → 0043

'가' → AC00

'나' → AC01

String s = new String ()



UTF-16

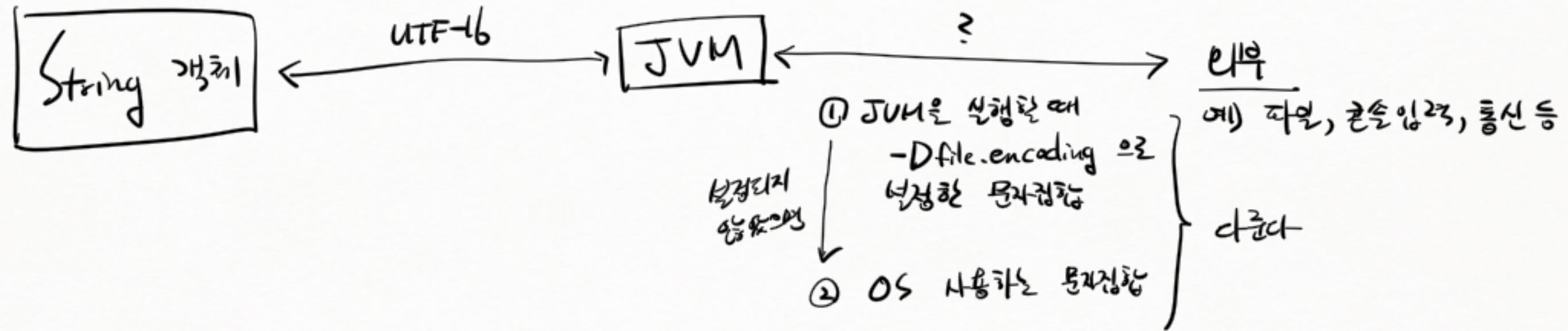
스팅 객체

char[]

--	--	--	--	--	--	--

JVM이 외부에서 문자데이터를 읽거나.
외부로 문자데이터 보낼 때
사용하는 문자변환기체
character set
(문자집합)

* JVM 내 문자 진화법



~~-Dfile.encoding=UTF-8~~ 복정 오류

java -cp bin/main com —

Windows
(MS949)

[MS949 → UTF16] ↗ 사용

byte[] $\xrightarrow{\text{영어}} \text{new String()}$

영어 EUC-KR $\longrightarrow \text{O}$

한글 UTF-8 $\longrightarrow \text{X}$

Unix/Linux
(UTF-8)

[UTF-8 → UTF16]

byte[] $\xrightarrow{\text{한글}} \text{new String()}$

EUC-KR $\longrightarrow \text{X}$

UTF-8 $\longrightarrow \text{O}$

-Dfile.encoding=UTF-8 복정 오류

java -Dfile.encoding=UTF-8 -cp bin/main —

Windows
(MS949)

byte[] $\longrightarrow \text{new String()}$

EUC-KR $\longrightarrow \text{X}$

UTF-8 $\longrightarrow \text{O}$

Unix/Linux
(UTF-8)

byte[] $\longrightarrow \text{new String()}$

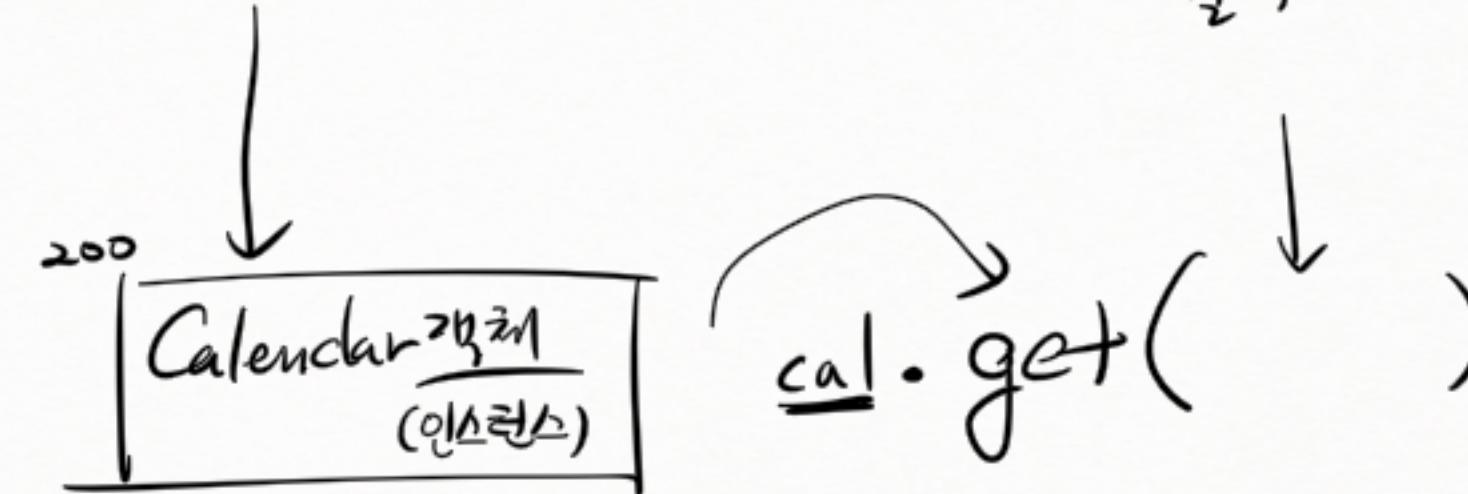
EUC-KR $\longrightarrow \text{X}$

UTF-8 $\longrightarrow \text{O}$

* Calendar 사용법

Calendar cal = Calendar.getInstance()

200



cal.get(1) → 년 을 일 시:분:초:밀리초

cal.get(Calendar.YEAR) → ①년도

같은 아이디를 갖기 때문에

그러나 소수점 뒷수에 따라 그 번호를 저장해 두고,

숫자는 문자이기 때문에 이해하기 수월.

* String : compareTo()

s1.compareTo(s2)
 ↙
 this

Hello;
Hello ← s1
Hello

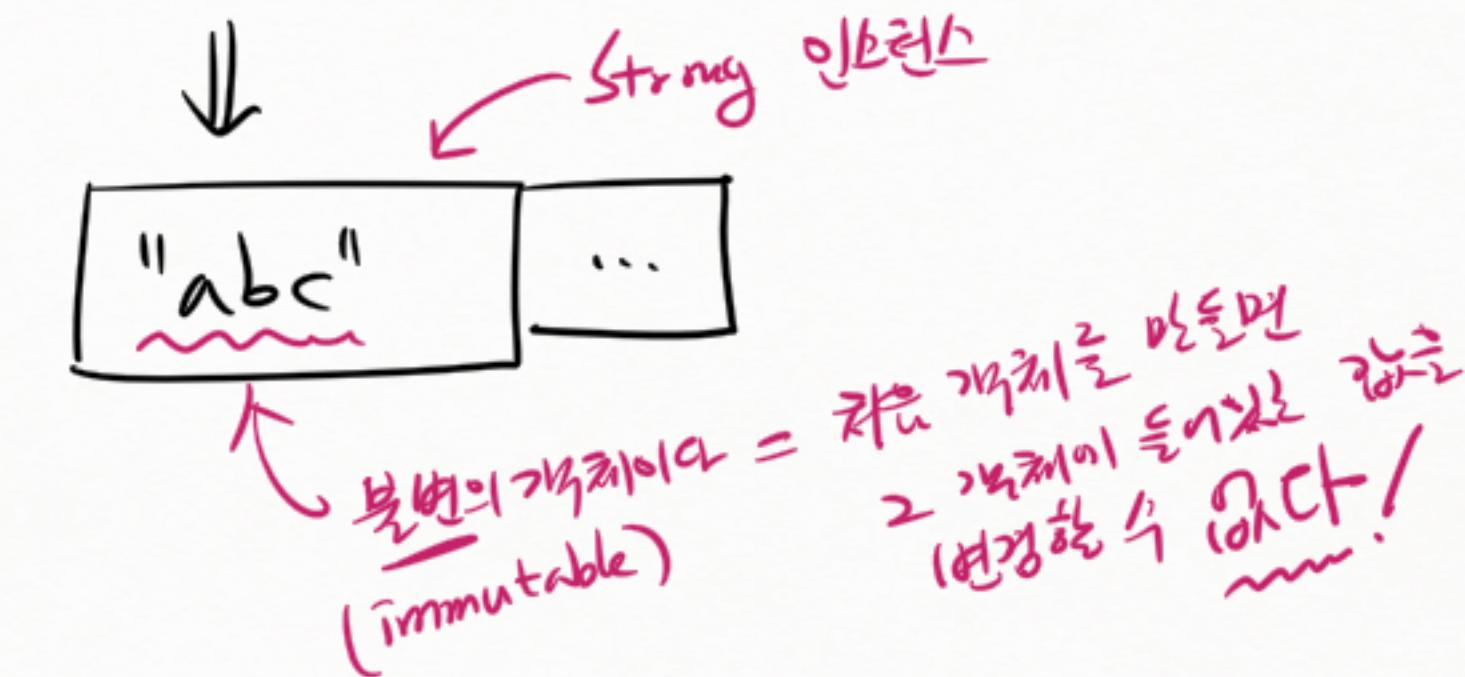
A



z

* Immutable 객체와 Strong

new String("abc")



* primitive type 2 wrapper 풀이

int → java.lang.Integer

byte → " .Byte

short → " .Short

long → " .Long

float → " .Float

double → " .Double

boolean → " .Boolean

char → " .Character

(
자료형 대체 가능
자동으로 만들 수 있음)
Wrapper

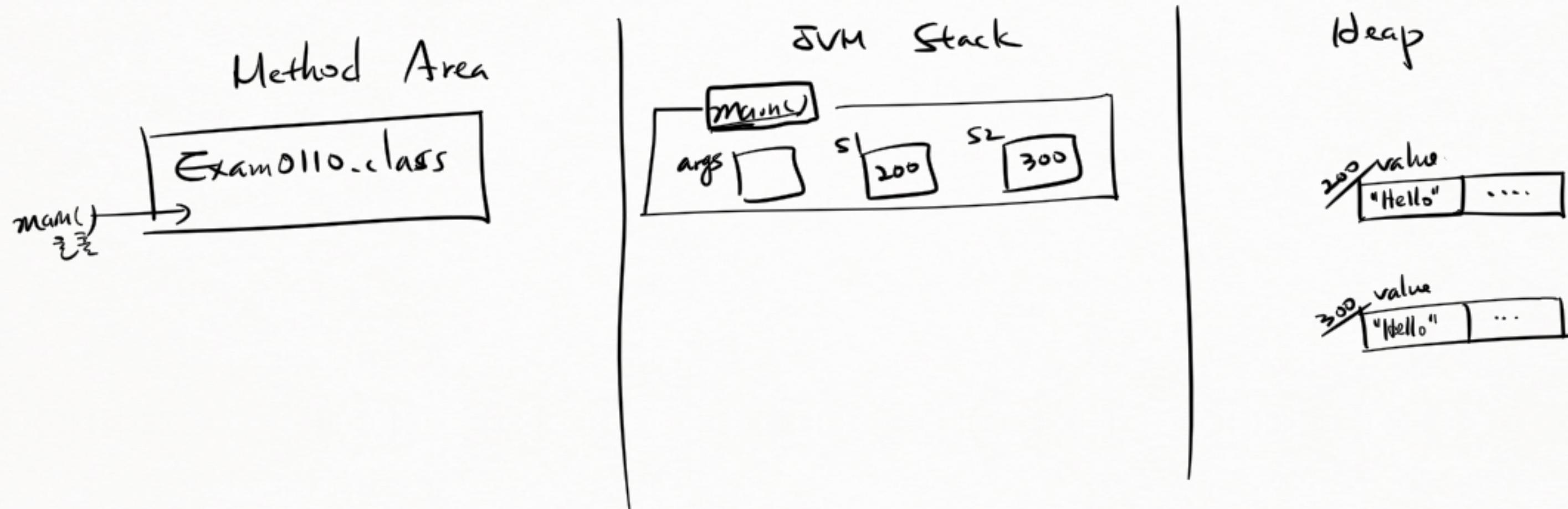
int → Integer

10

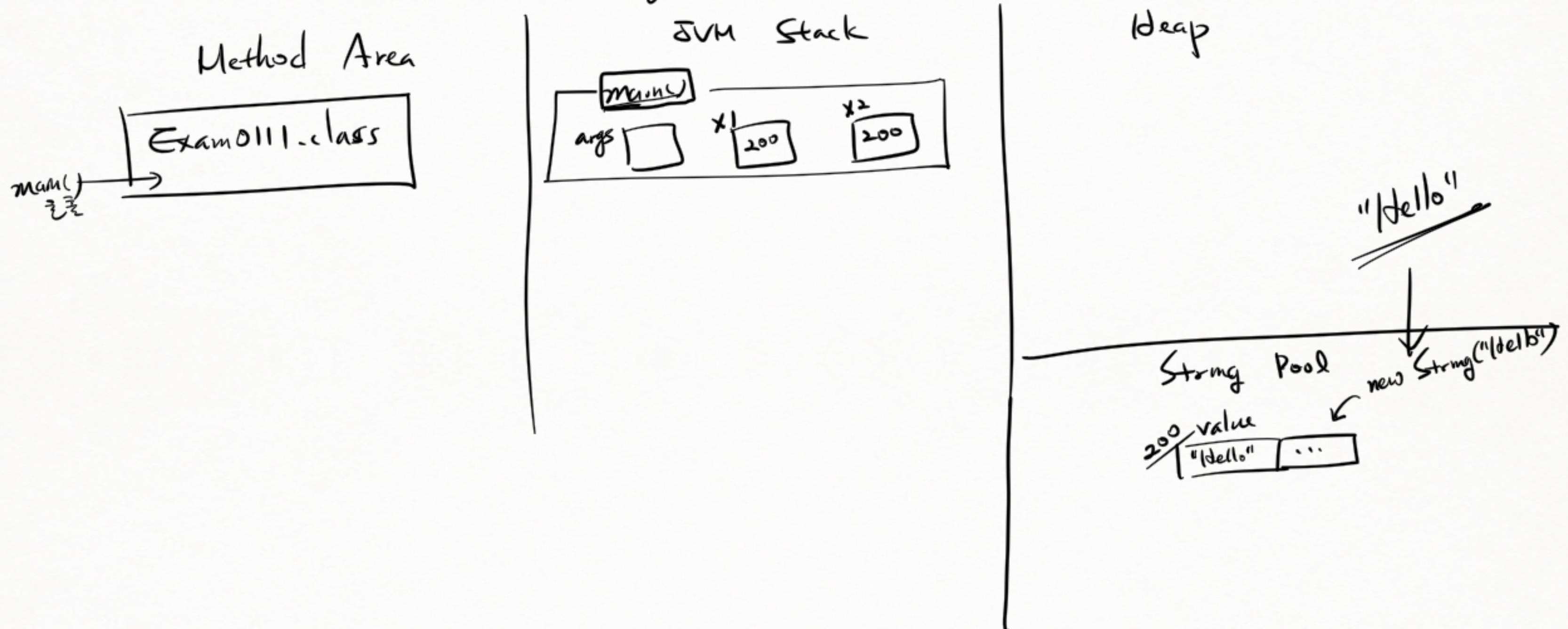
value
10 ...

인스턴스 변수가 있는 것임

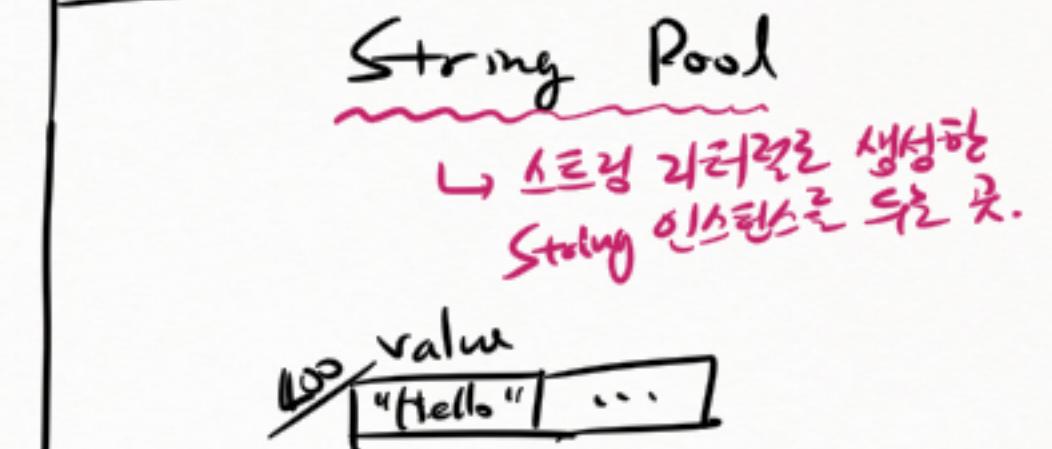
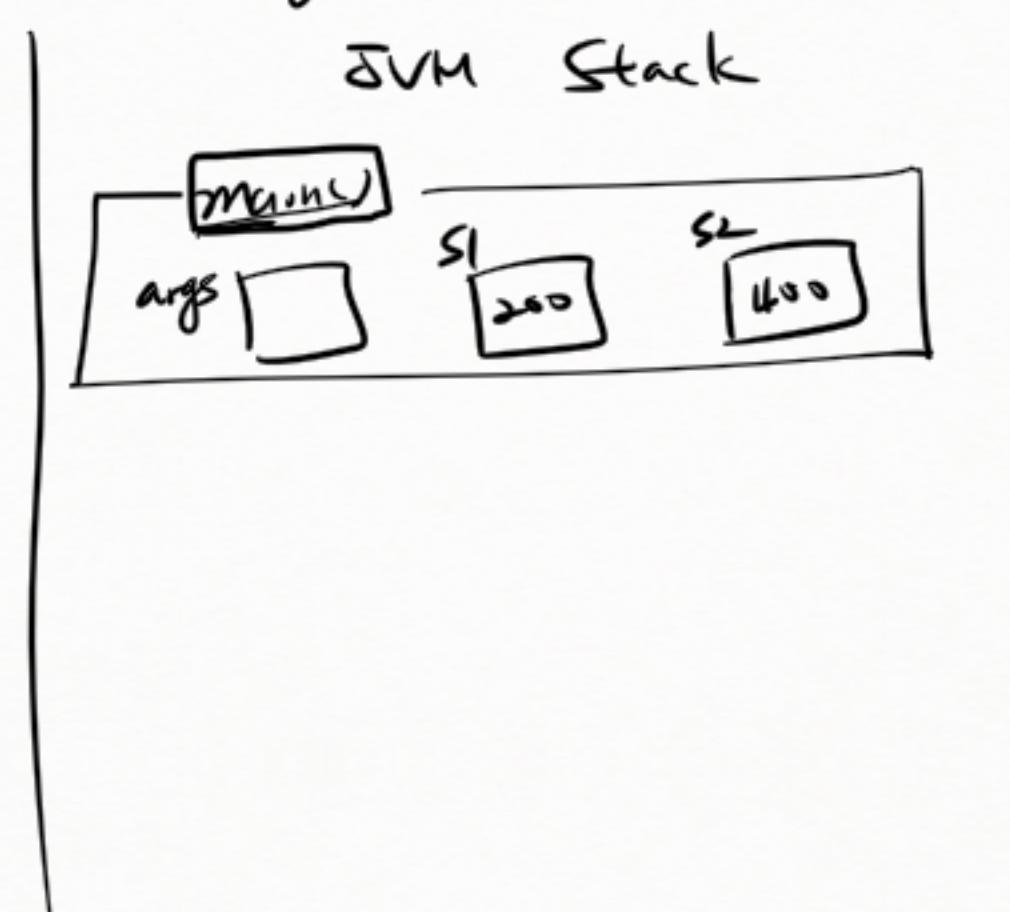
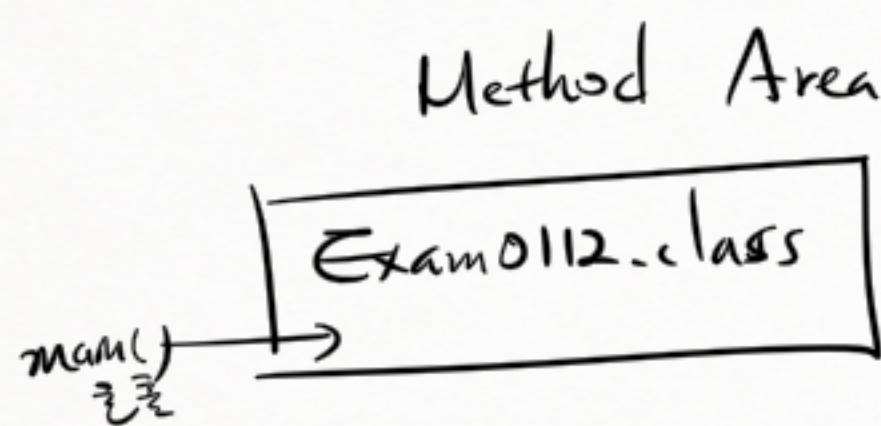
* lang. ex02 - Exam0110 - String 之 m< n & 之



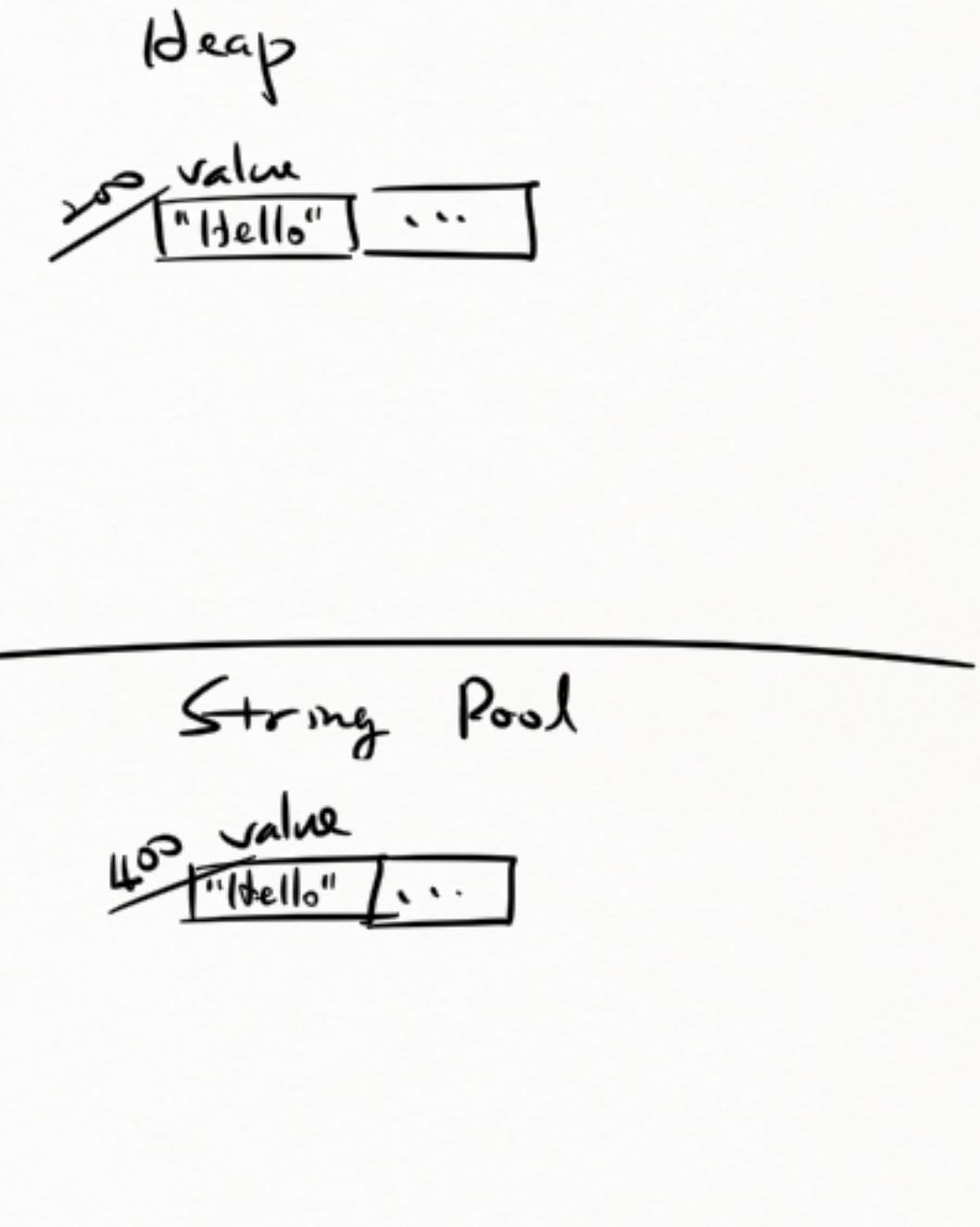
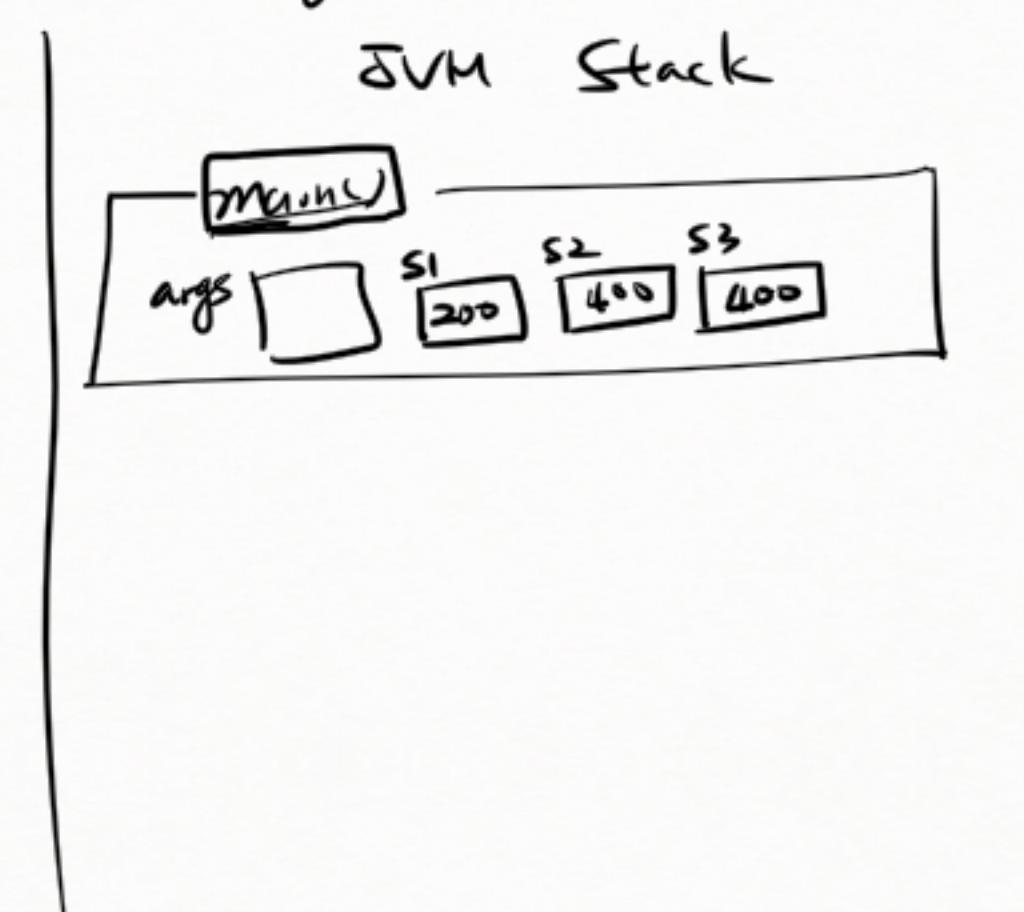
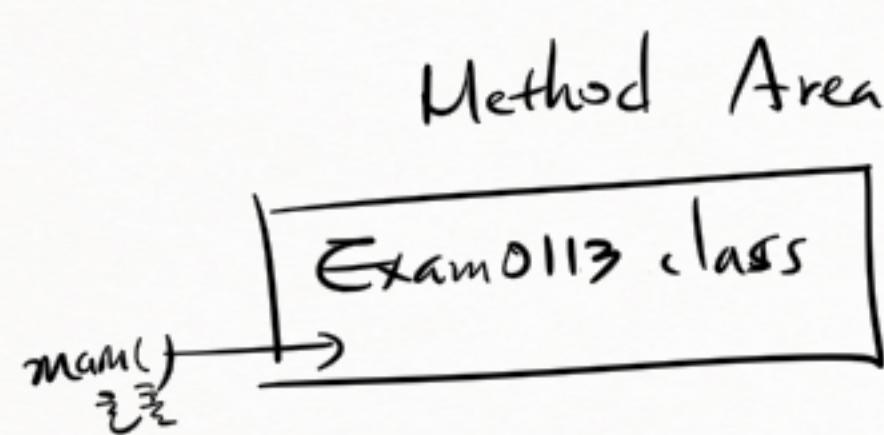
* String 은 mt. No법 - 스트링 리터럴로 인스턴스 생성하기
lang.ex02.Exam0111



* String 초기화 방법 - new vs "—"
lang.ex02.Exam0112

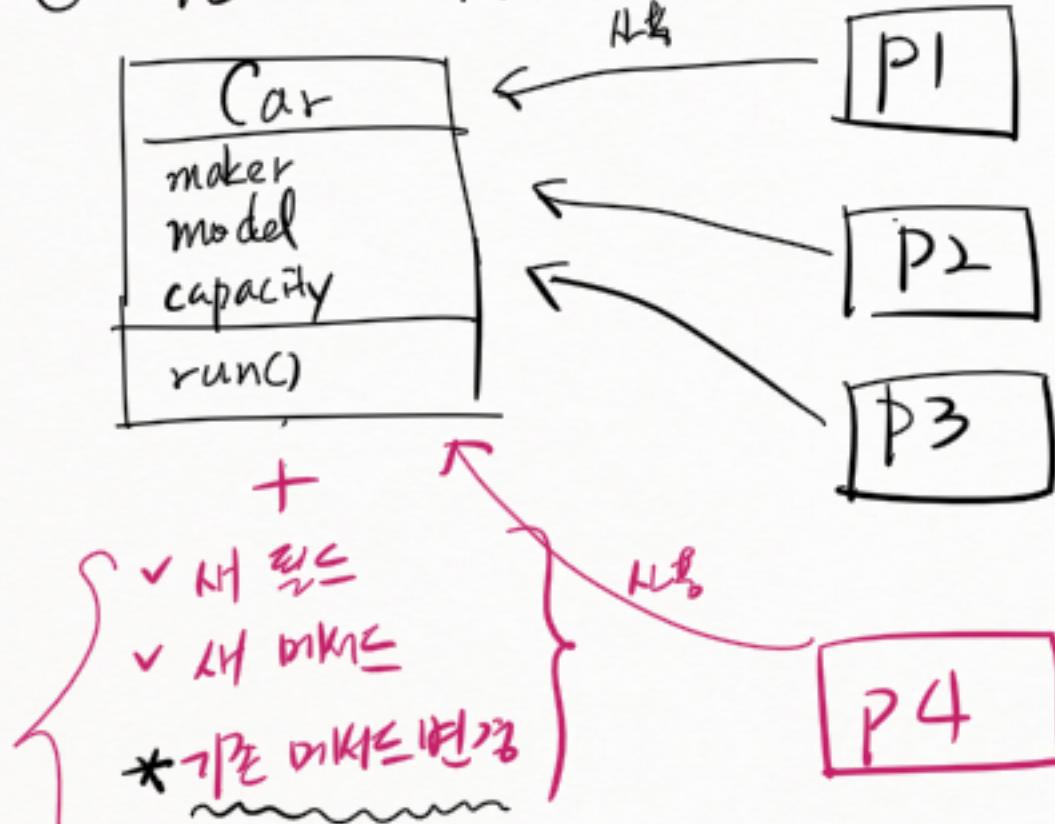


* String အဲမှု မှာ - intern()
lang.ex02.Exam0113

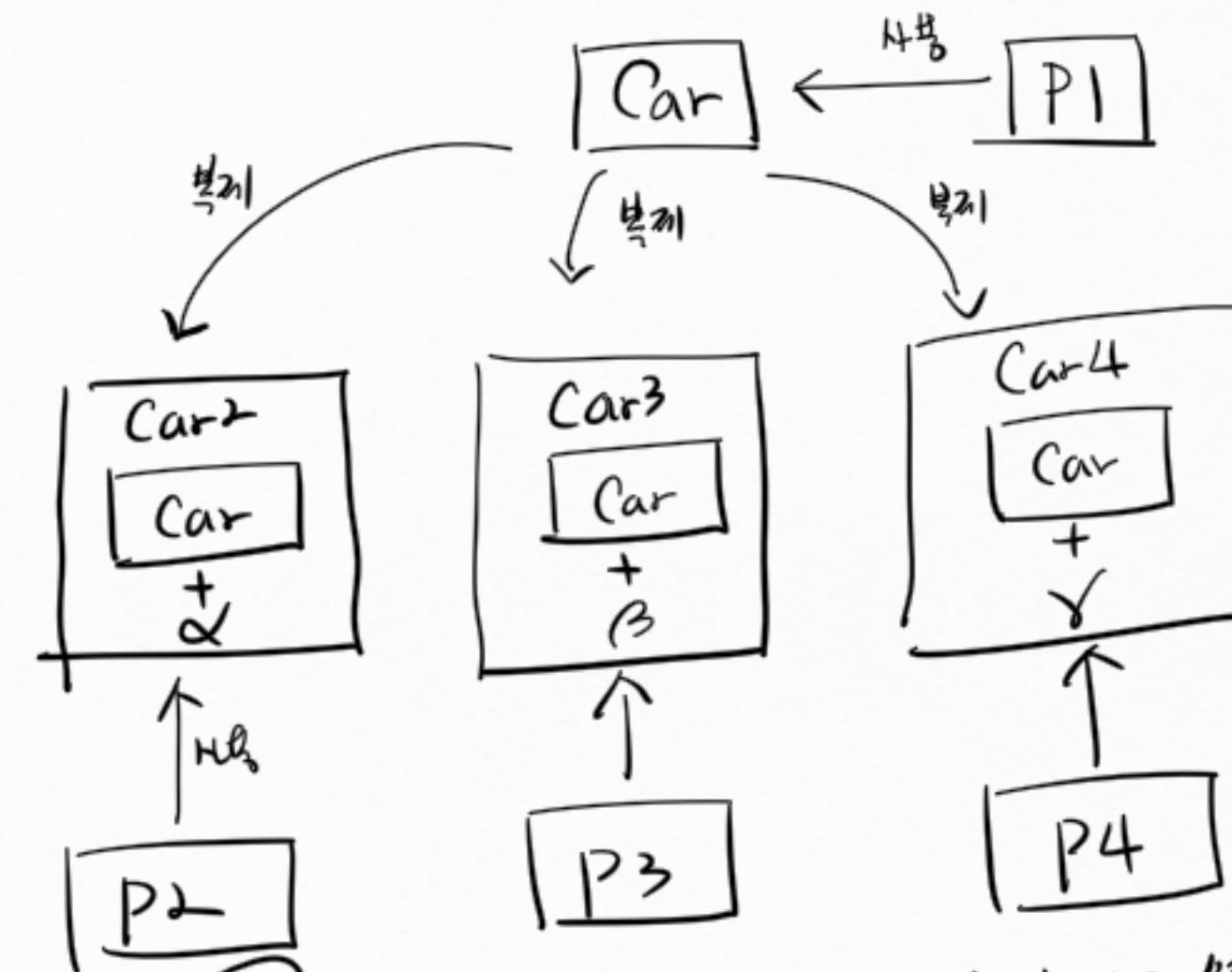


* 상속 - 기능 확장은 위한 방법 (oop.ex05)

① 기존 클래스에 새 기능 코드를 추가하는 방법



② 기존 코드를 복제하여 새 기능을 추가하는 방법



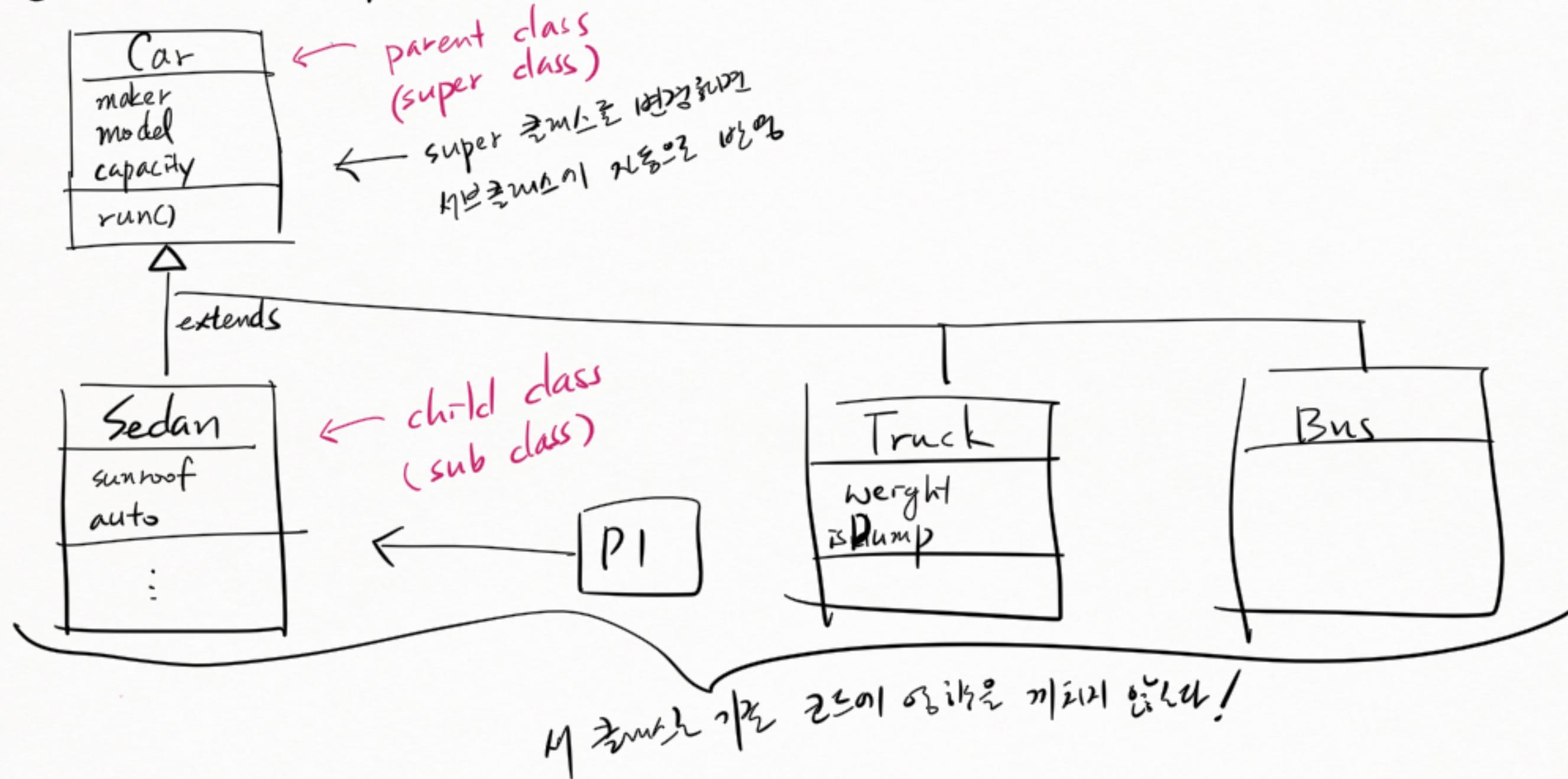
장점:
기존 코드를 손대지 않아
기존 프로젝트에 영향을 미치지 않는다.

원본 버전 수정 → 모든 복사본 버전 수정
원본 기능변경 → "

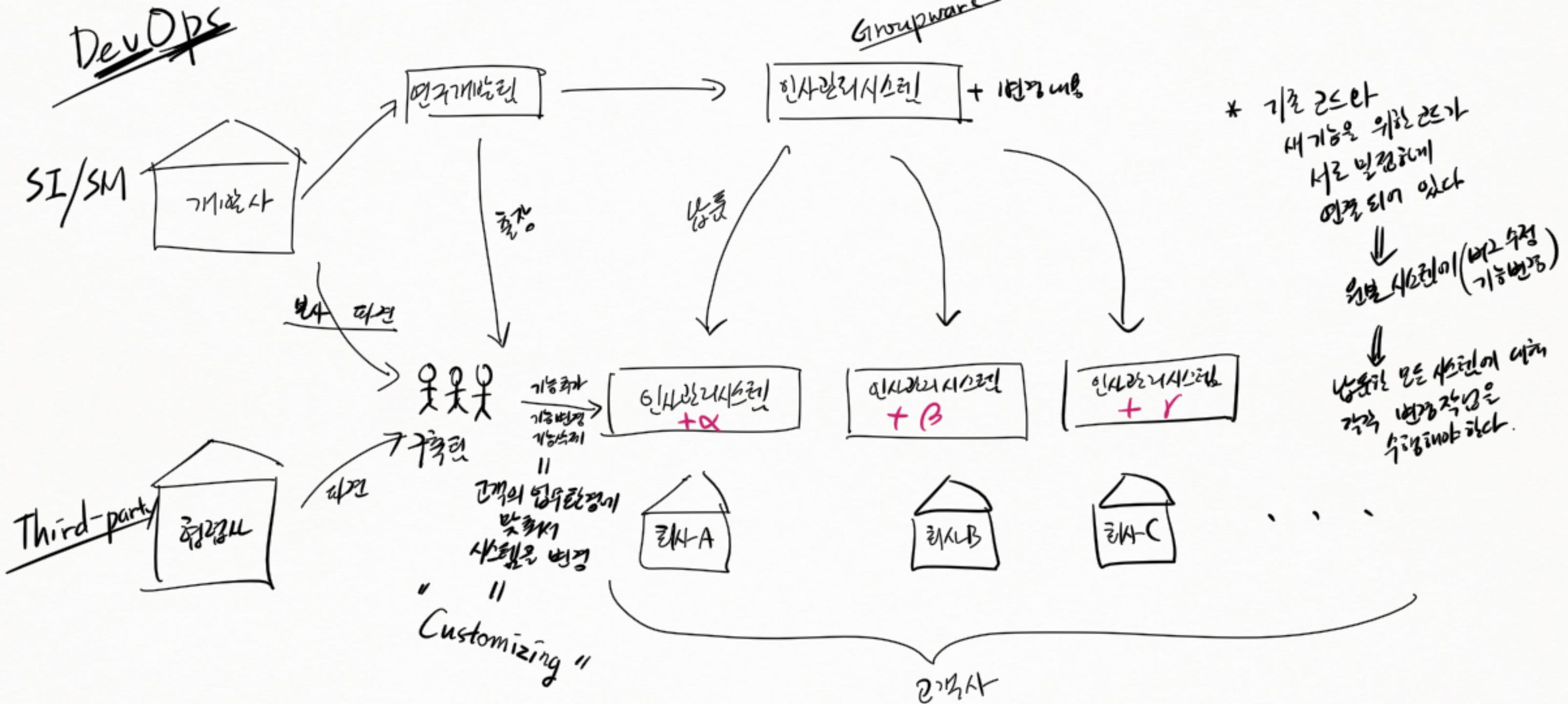
단점:

* 상속 - 기능 확장은 위한 기법 (oop.ex05)

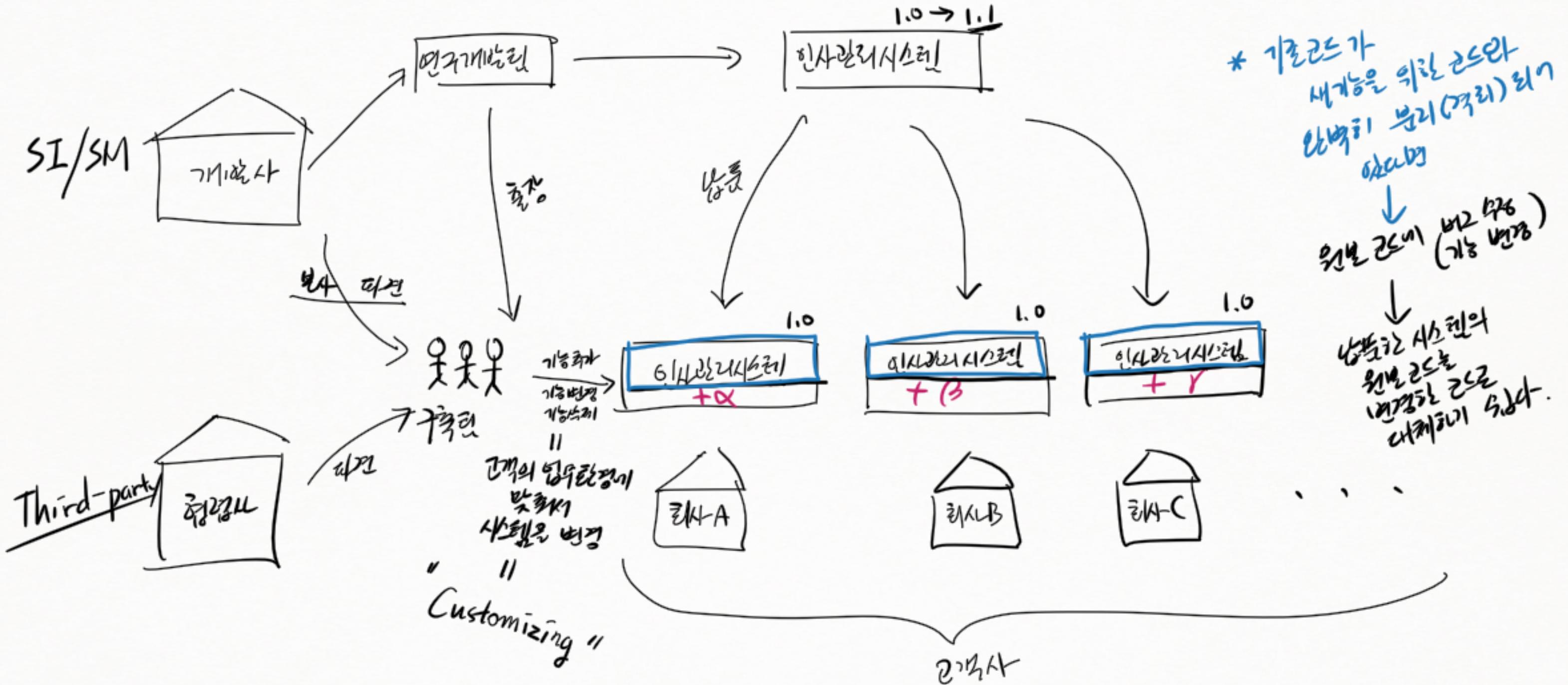
③ 상속을 이용한 기능 확장



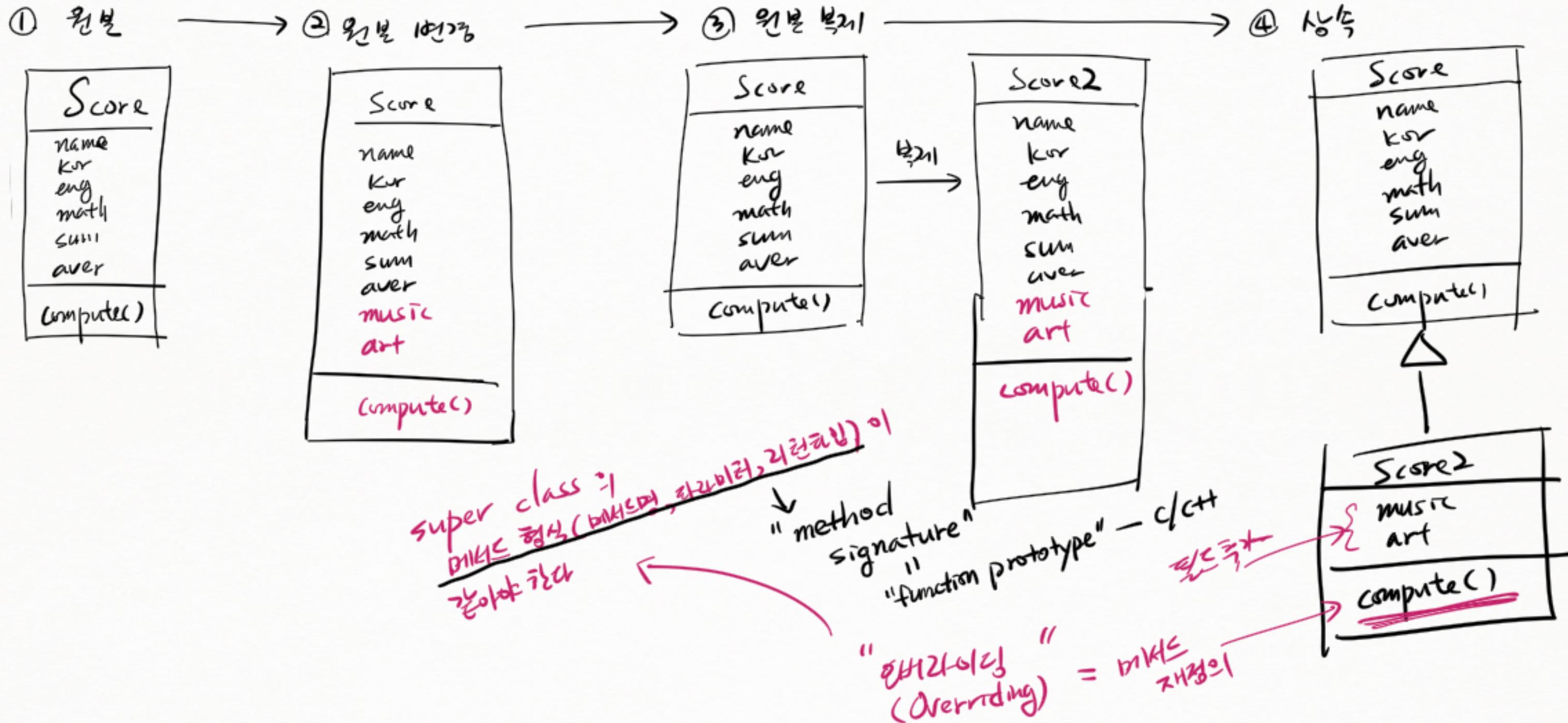
* 시스템화 - 고객사 설문을 토대로 커스터마이징 수준



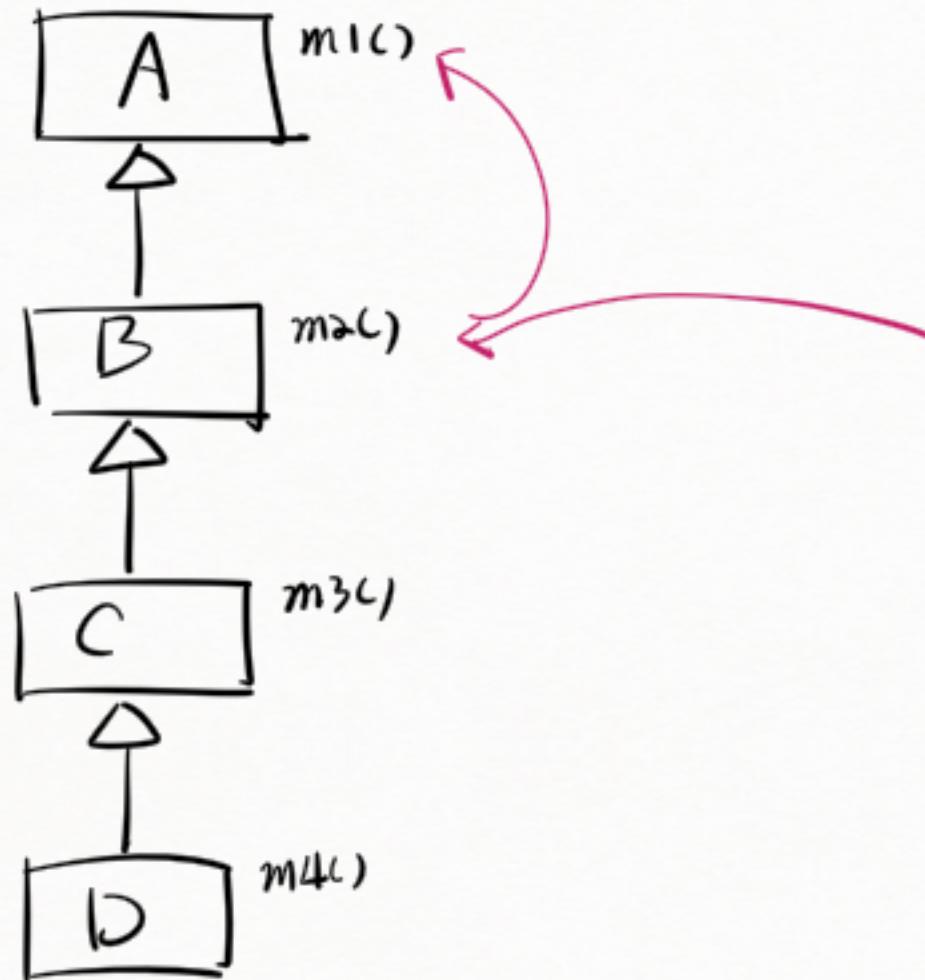
* 시스터즈 - 고객사별로 특화화 커스터마이징 가능



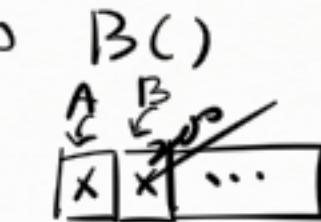
* 학습 예 - oop. ex05. a/b/c/d



* 상속 예 - oop. ex05. e

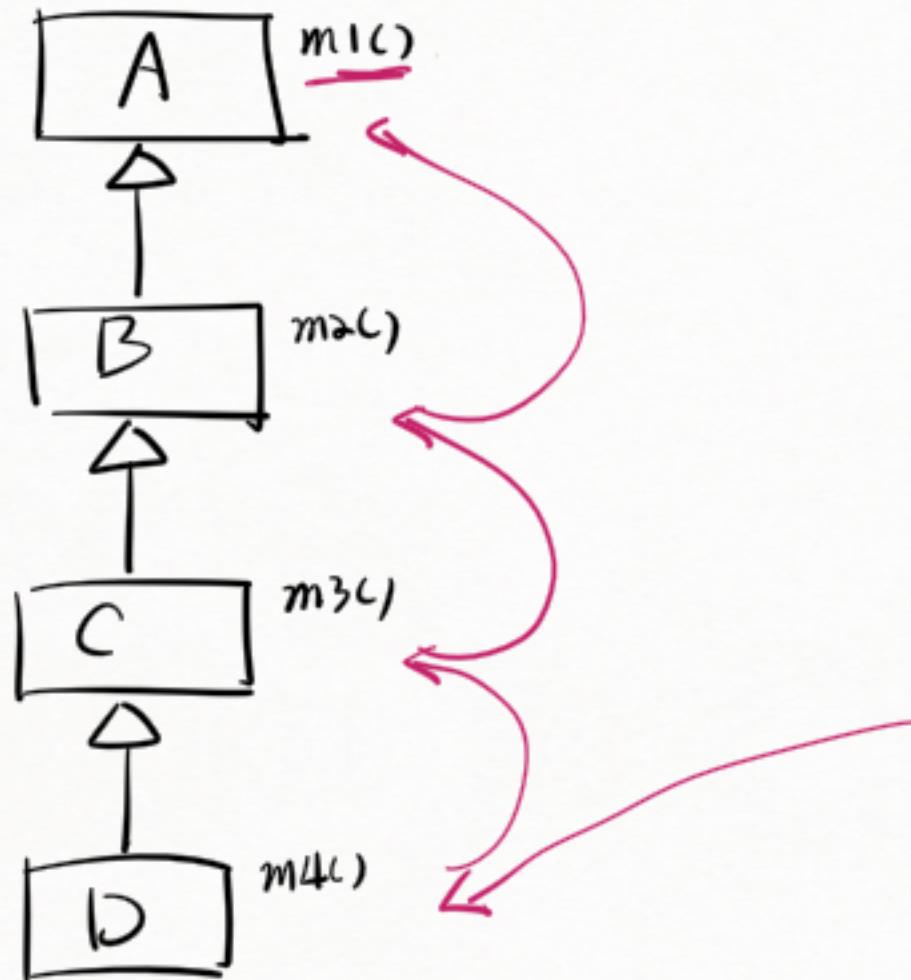


B obj = new B()
obj.m1();
obj.m2();



메서드 찾는 순서 \Rightarrow B \rightarrow A.m1() 훸 훸
 ↓
 레퍼런스의 클래스부터 찾기 시작해서
 수퍼 클래스로 따라 올라가면서 찾는다.
 B.m2() 훸 훸

* կենա - oop. exst. e



D obj = new D()

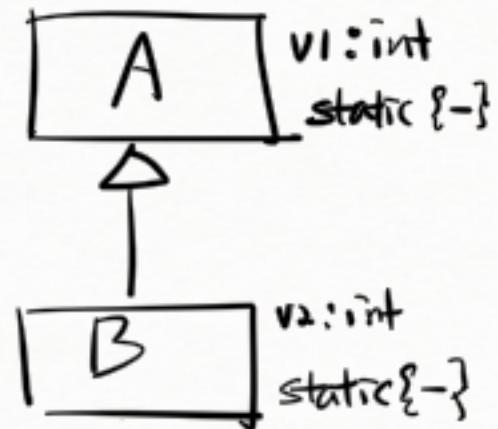
obj. m4(); D.m4() ↗

obj. m3(); D → C.m3() ↗

obj. m2(); D → C → B.m2() ↗

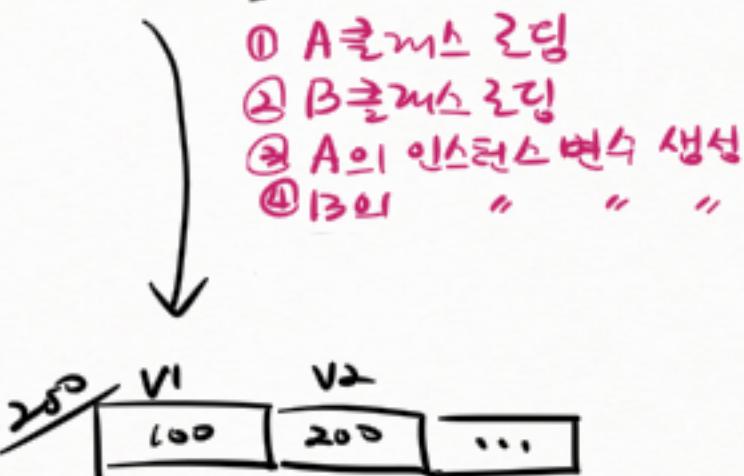
obj. m1(); D → C → B → A.m1() ↗

* 상속 예 - oop.ex05.f



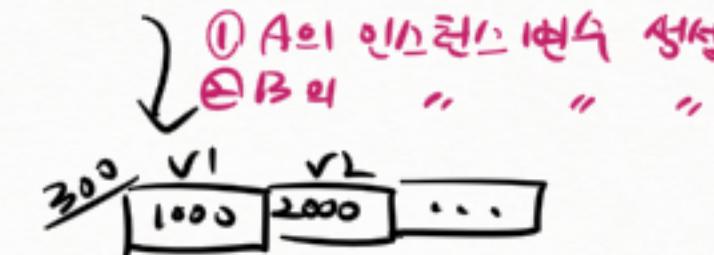
`B obj = new B();`

`obj`
[200]

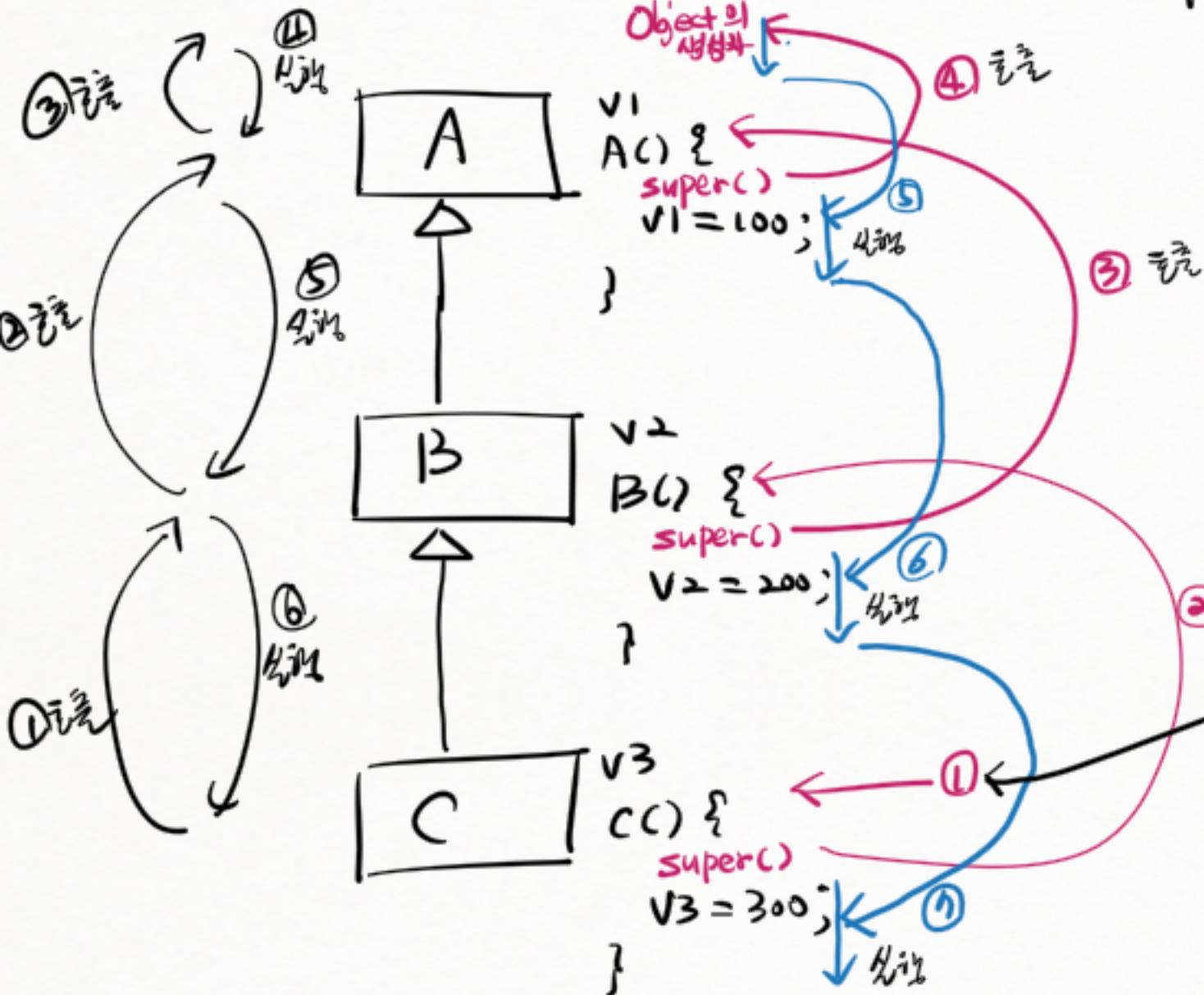


`B obj2 = new B();`

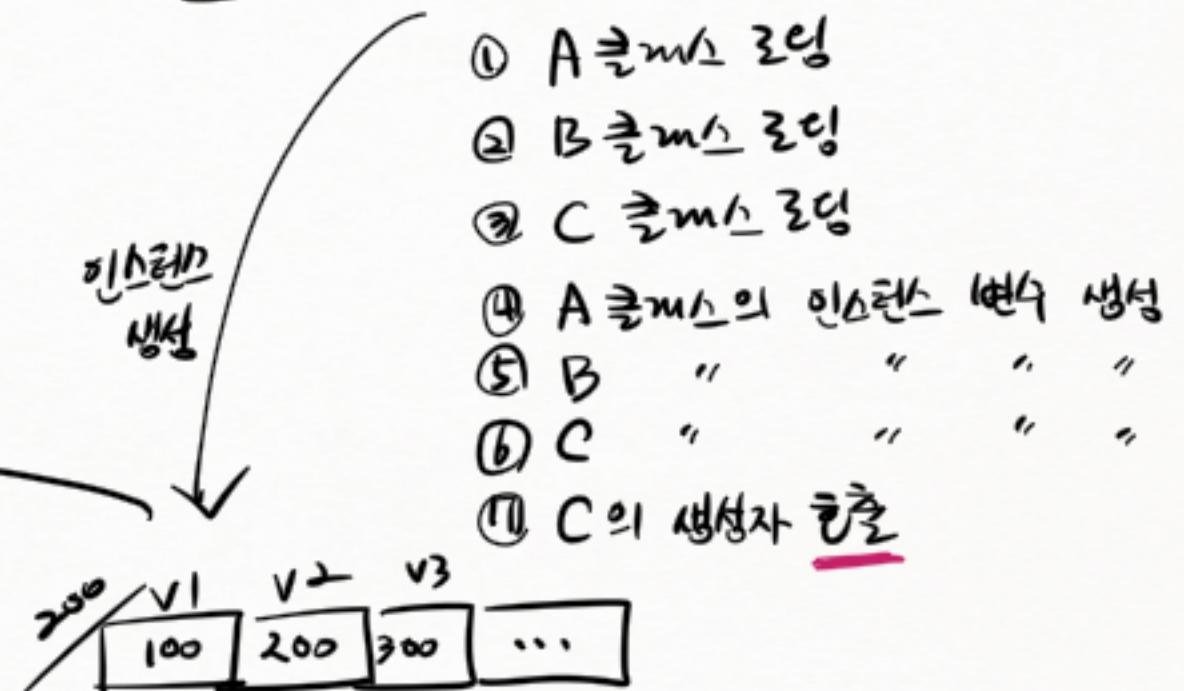
`obj2`
[300]



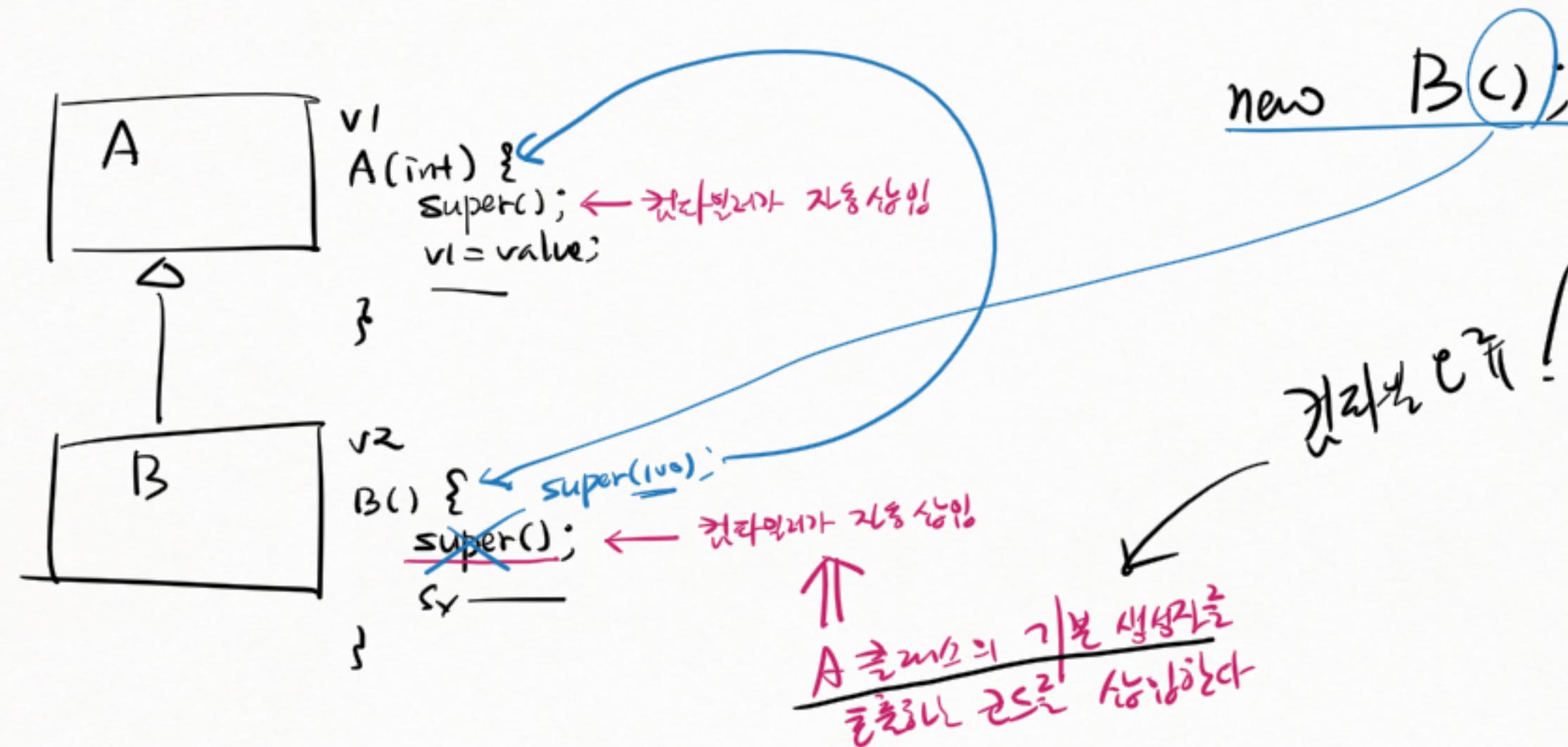
* 상속 예 - oop. ex05.g



`C obj = new C();`

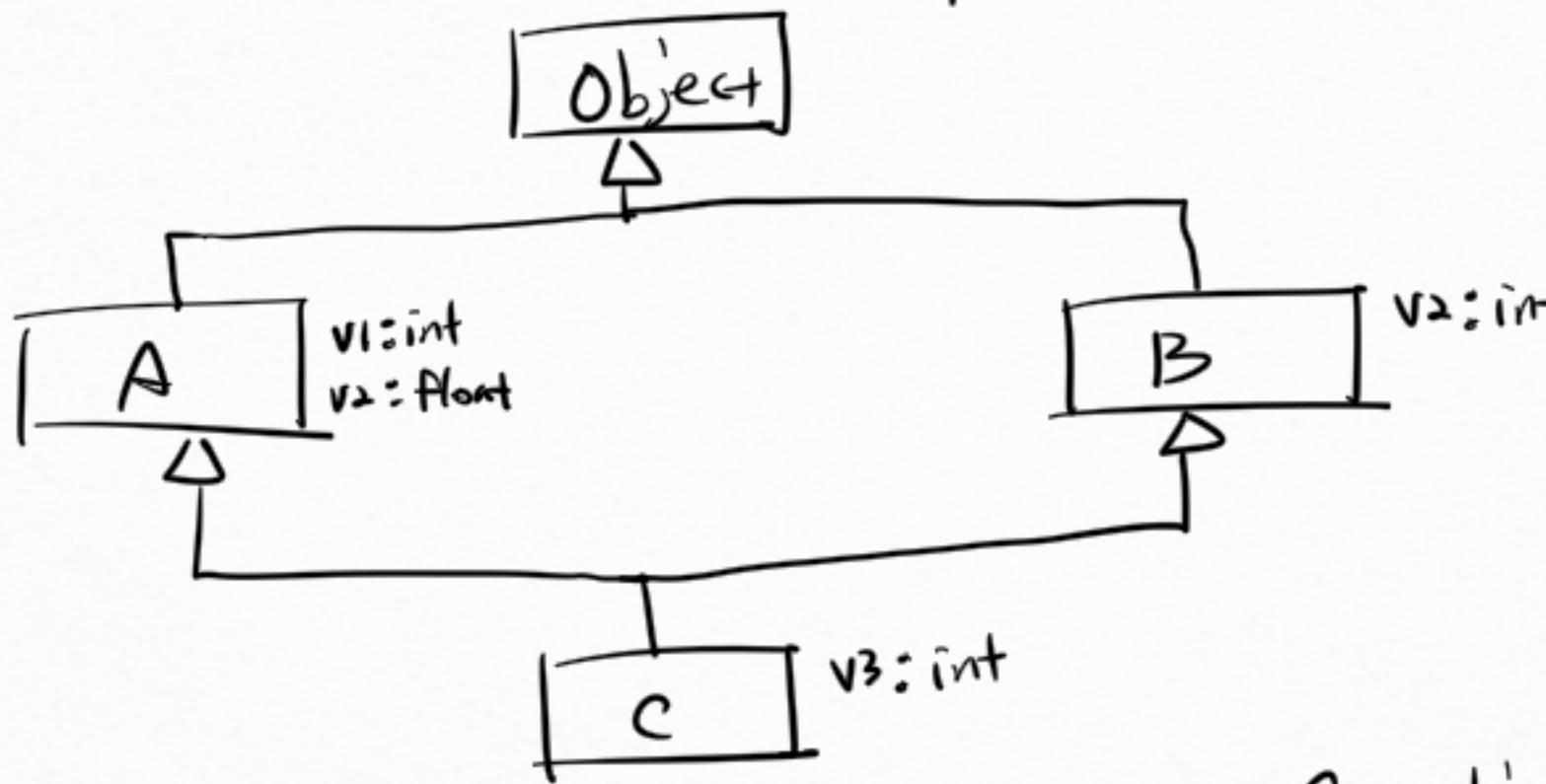


* 상속과 슈퍼클래스 생성자 (oop-ex5.h)



* 다중 상속

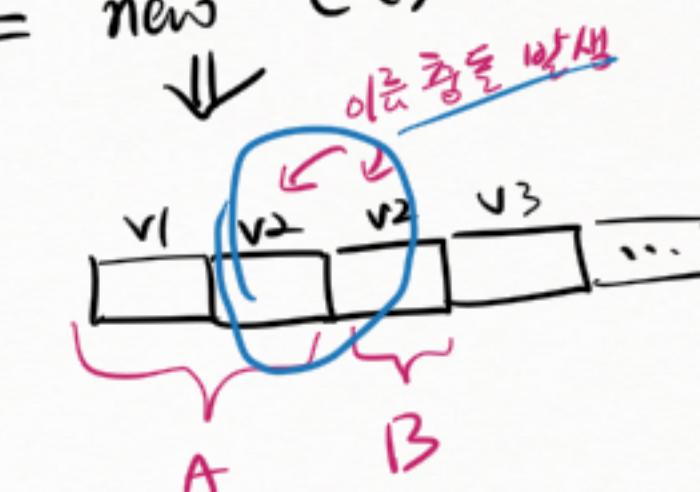
(oop. ex05. i)



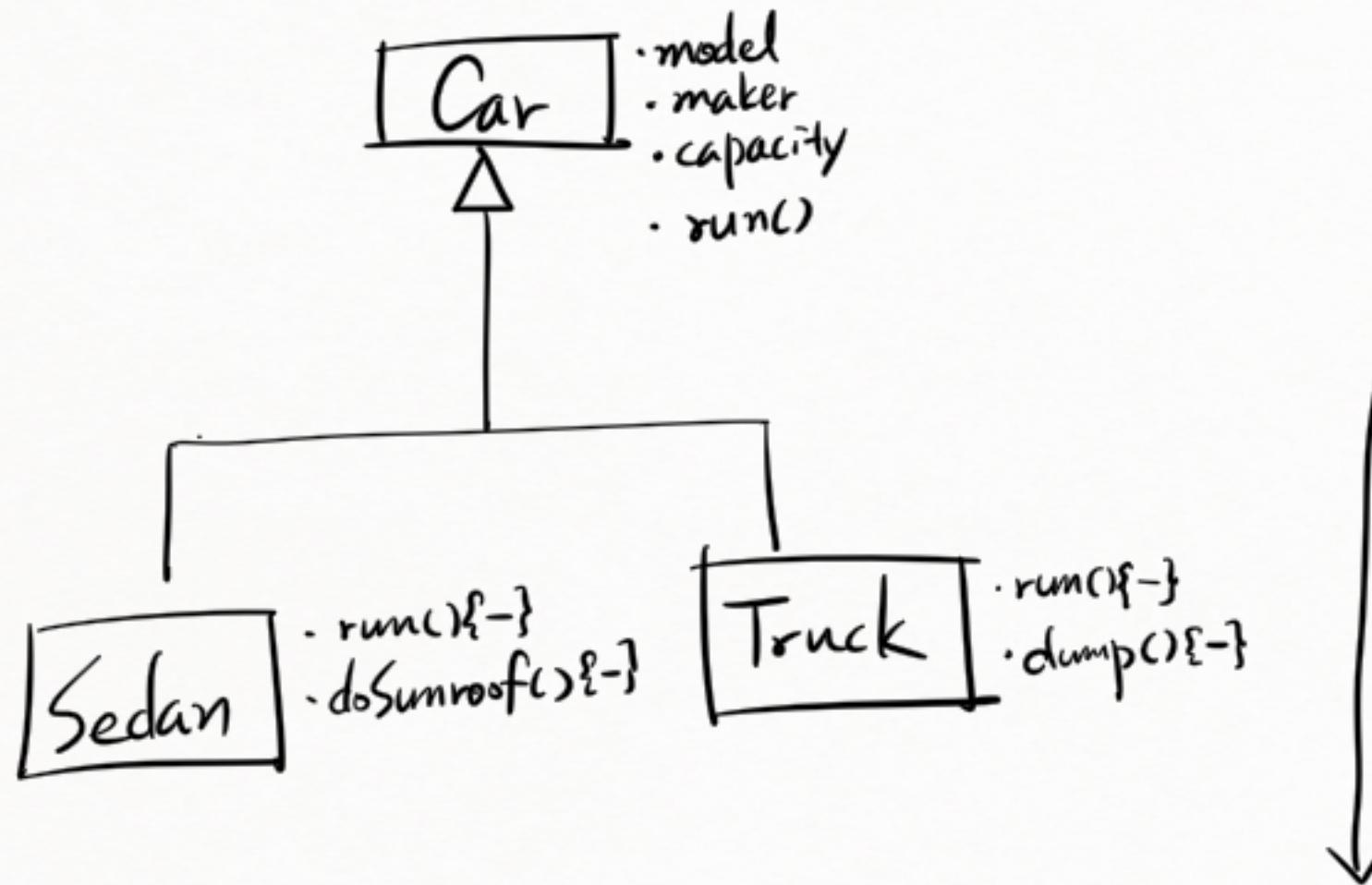
* 여러 클래스를 동시에 상속받아
변경시, 원래의 클래스에서 증설이 가능해진다.
이른바 多重継承이다.

그러나多重継承을 하면 여러 문제점이 있다.

`C obj = new C();`

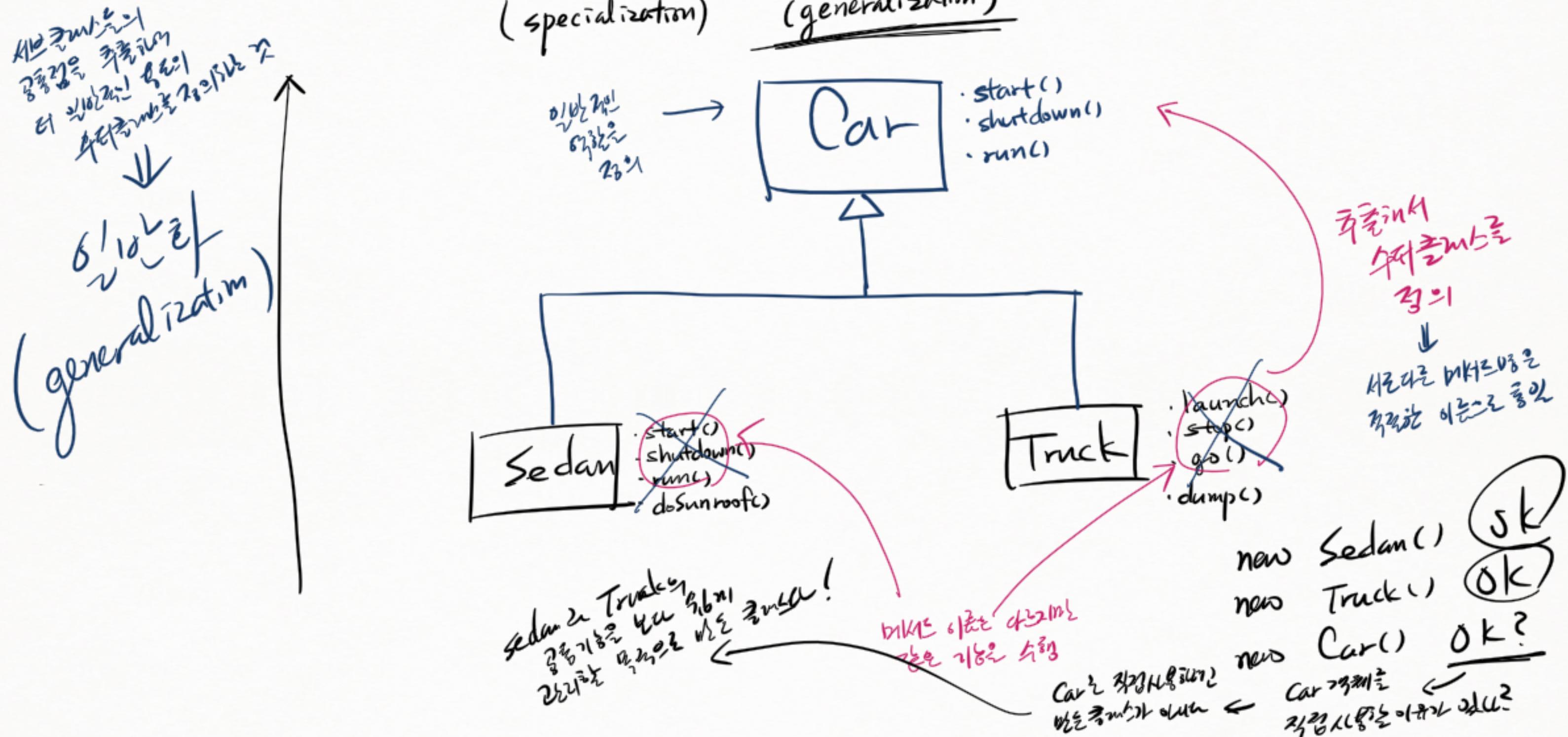


* 상속 : 전문화 와 일반화 (oop. ex05. j)
(specialization) (generalization)

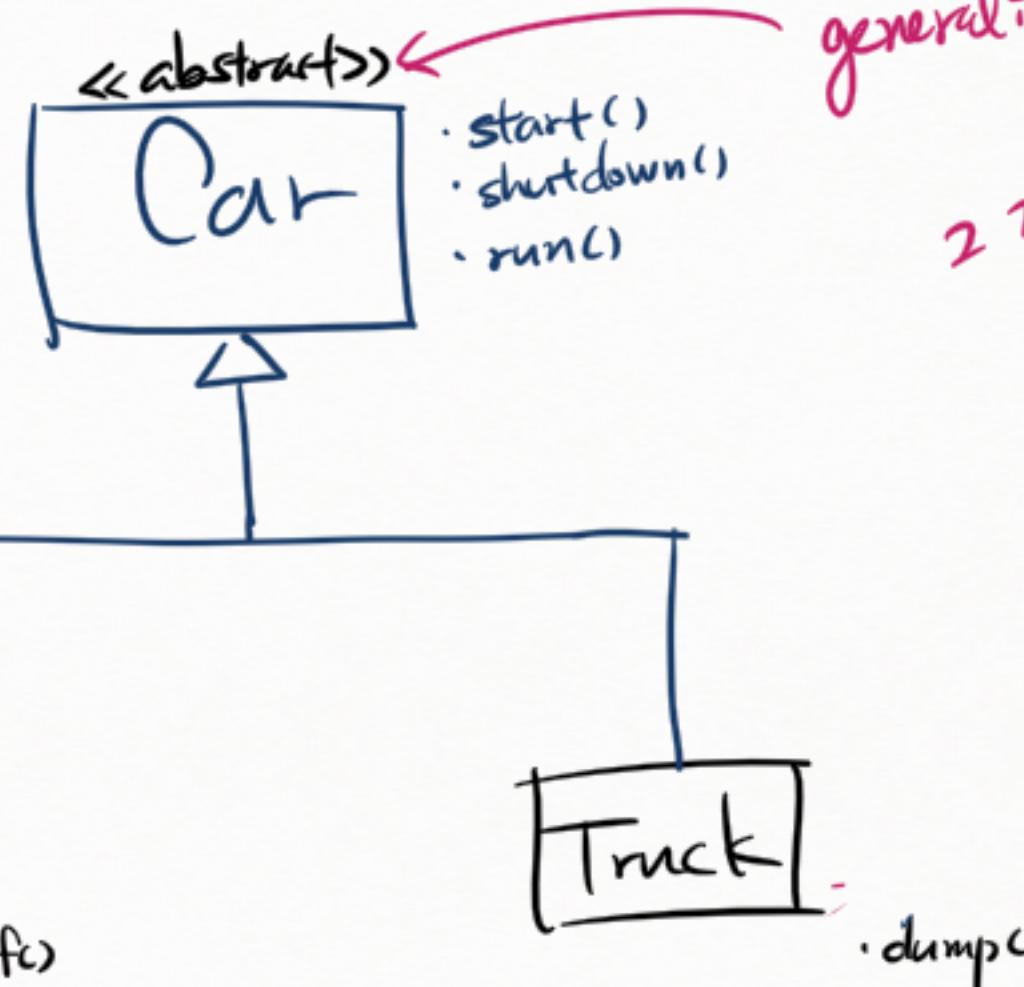
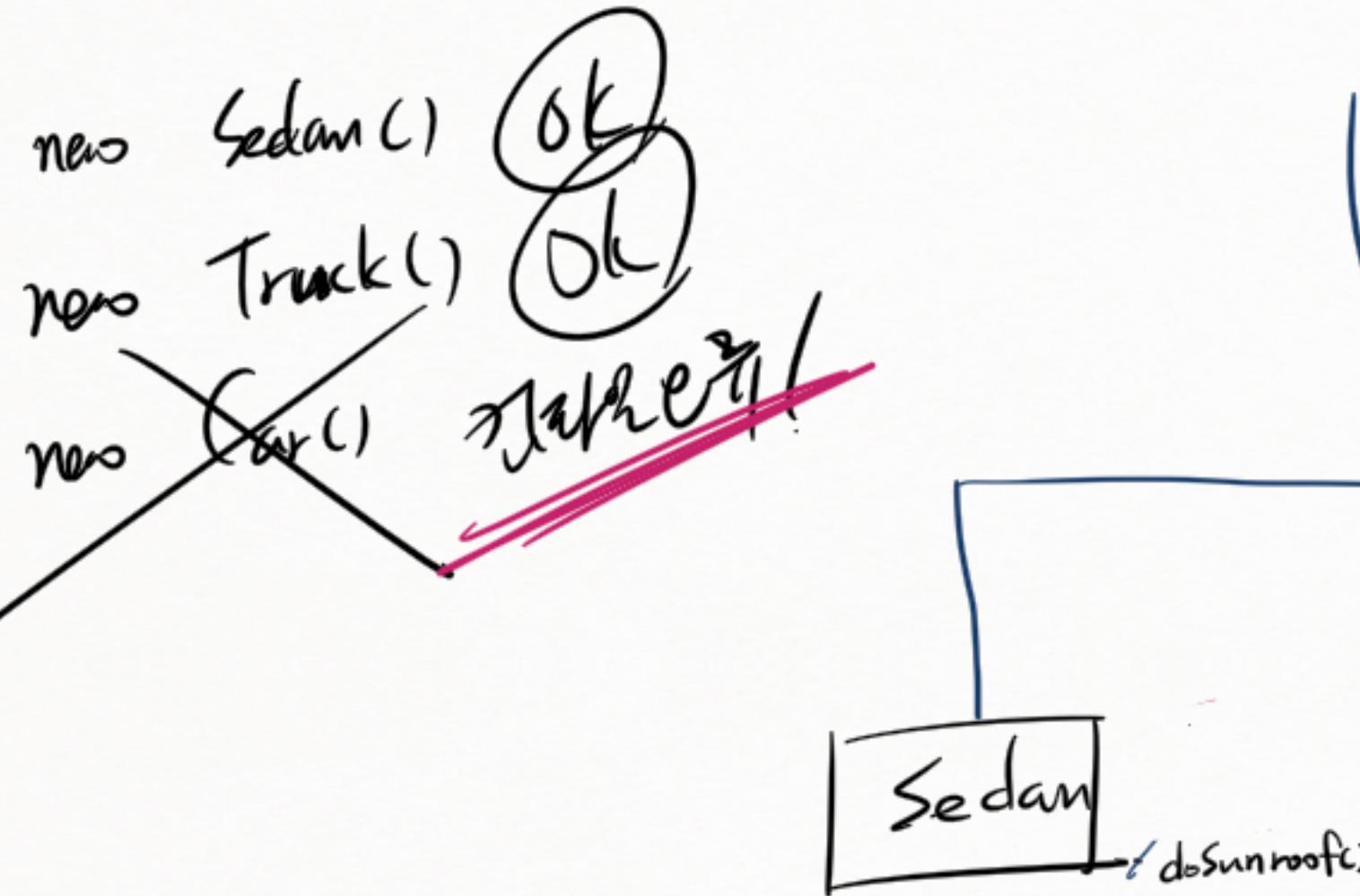


수퍼클래스를 상속 받을 때
기능을 더 물어가기
조금 특별한 예로는 가능!
A부모클래스를 상속 받을 때
↓
"Specialization
(전문화)"

* 상속 : 전문화 와 일반화 (coop. ext. k/l)
 (specialization) (generalization)



* 상속 : 일상화한 추상 클래스 (sup. class)

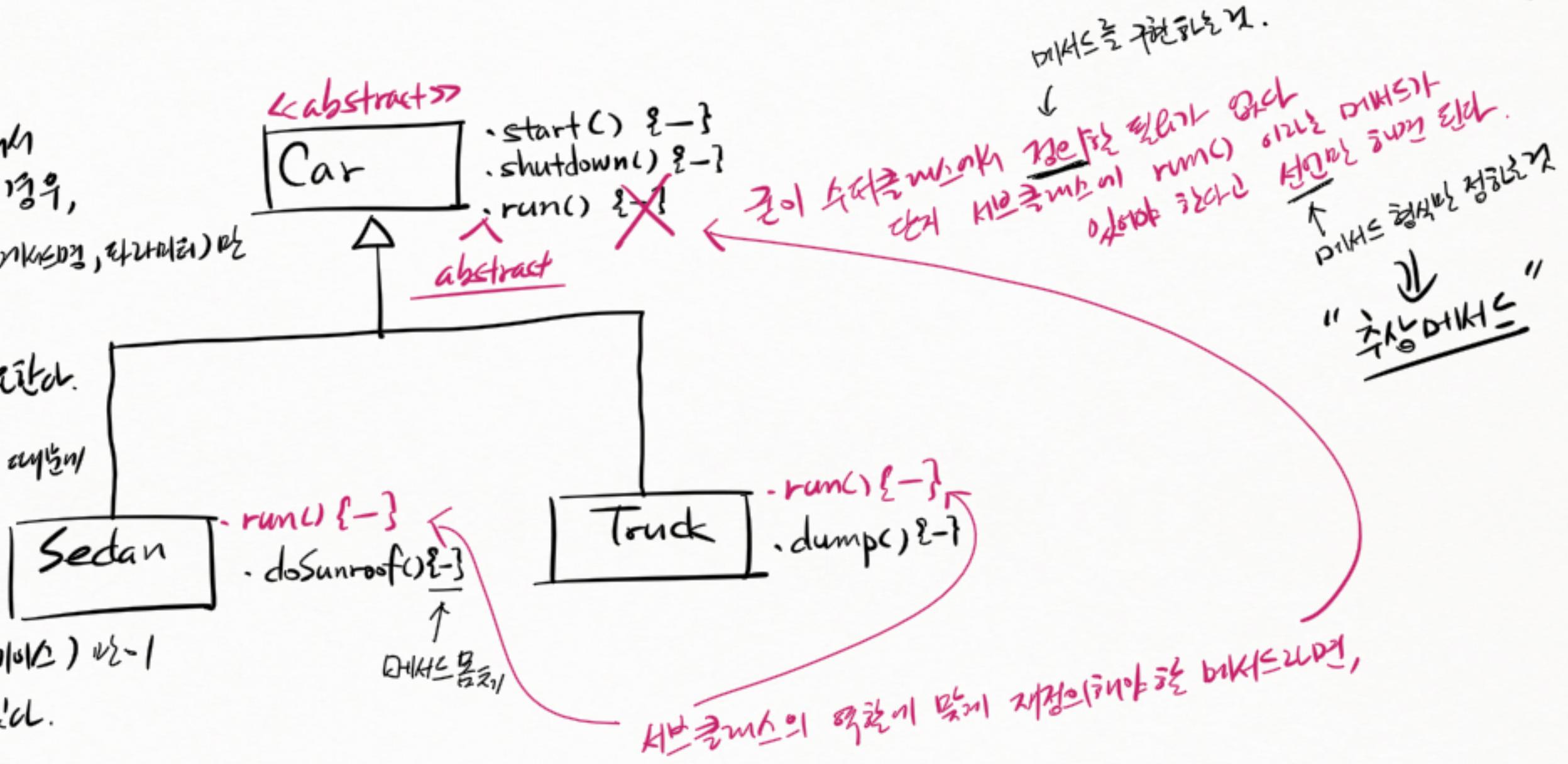


generalization을 통한 접근
추상 클래스는 실제 접근 목적으로 2 가지로 각각 다른 목적으로 활용
↓
이전 클래스는 실제로 일반화 활용되는 것은 막을 필요가 있다
↓
이제 예전으로 등장
물론이지
“추상 클래스”
(abstraction)

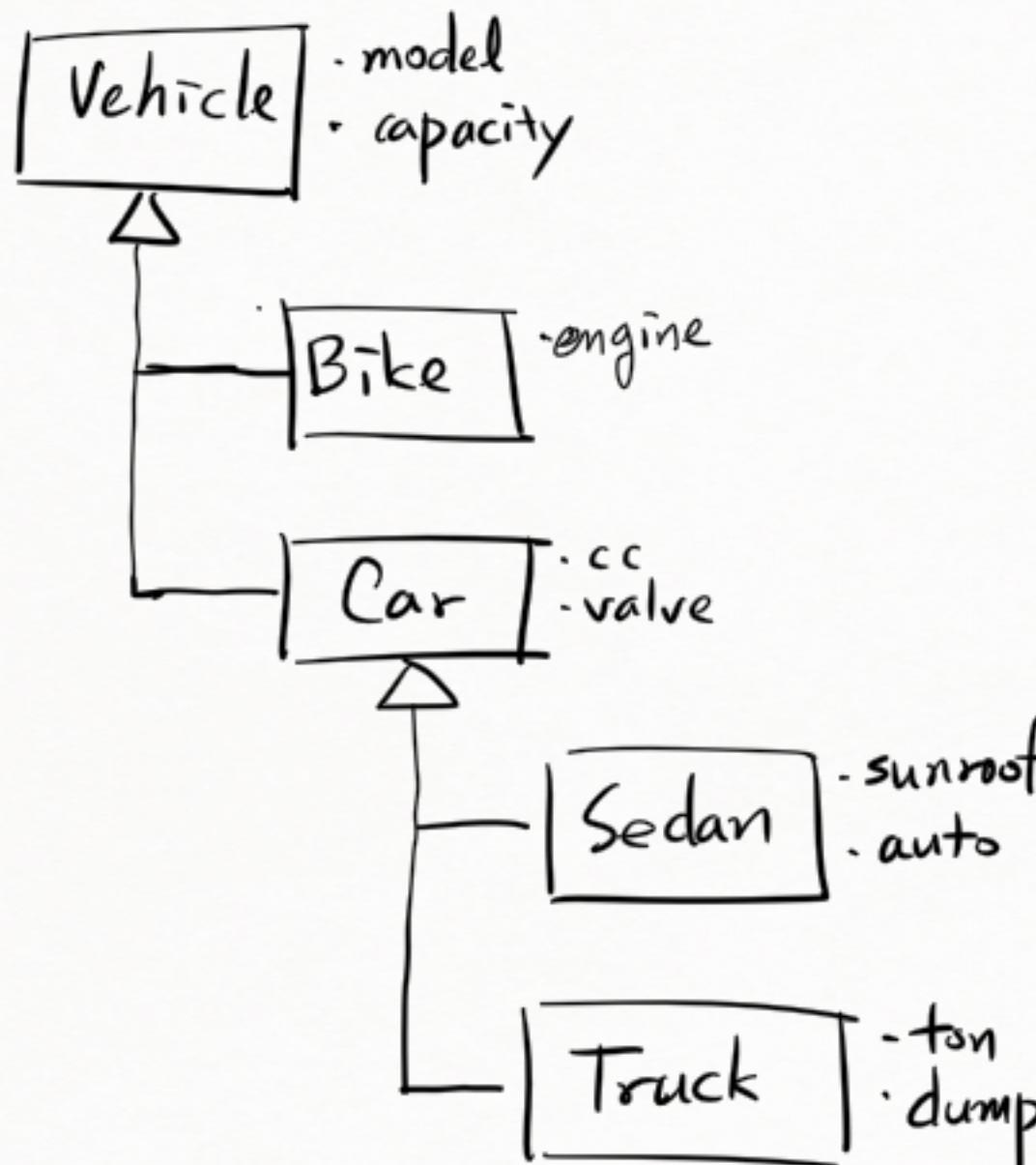
* 상속 : 추상클래스와 추상메소드 (oop.exst.n)

* 추상메서드

- 세부클래스의 역할에 맞게서
구현해야 하는 메서드인 경우,
- 메서드 구현(리턴타입, 매개변수, 파라미터)은
작성한다.
- 세부클래스에게 구현을 강제함.
- 메서드의 몸체를 만들지 않기 때문에
일반클래스 혹은 추상메서드를
선택 없다
- 오직 추상클래스(또는 인터페이스)만이
추상메서드를 가질 수 있다.



* 다형성 (polymorphism) : oop. exob. a



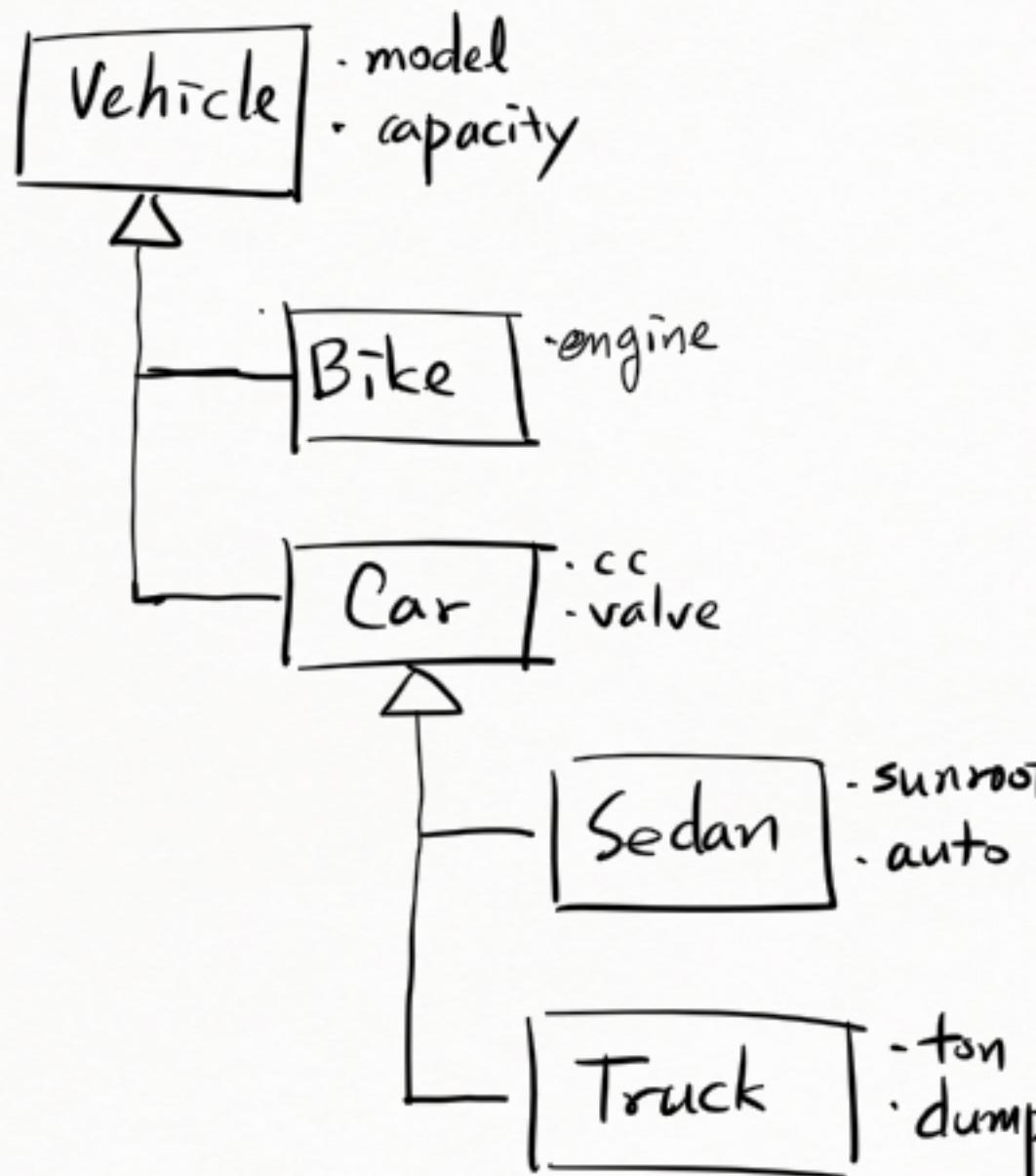
Vehicle c1;

c1 = new Vehicle(); → 

c1 = new Bike(); → 

- ① 상위 클래스의 레퍼런스는 하위 클래스의 인스턴스를 갖을 수 있다.
 ② 하위 클래스의 인스턴스는 상위 클래스의 인스턴스로 가리킬 수 있다.
 ③ 하위 클래스의 객체를 가리킬 수 있다.

* 다형성 (polymorphism) : oop. exob. a

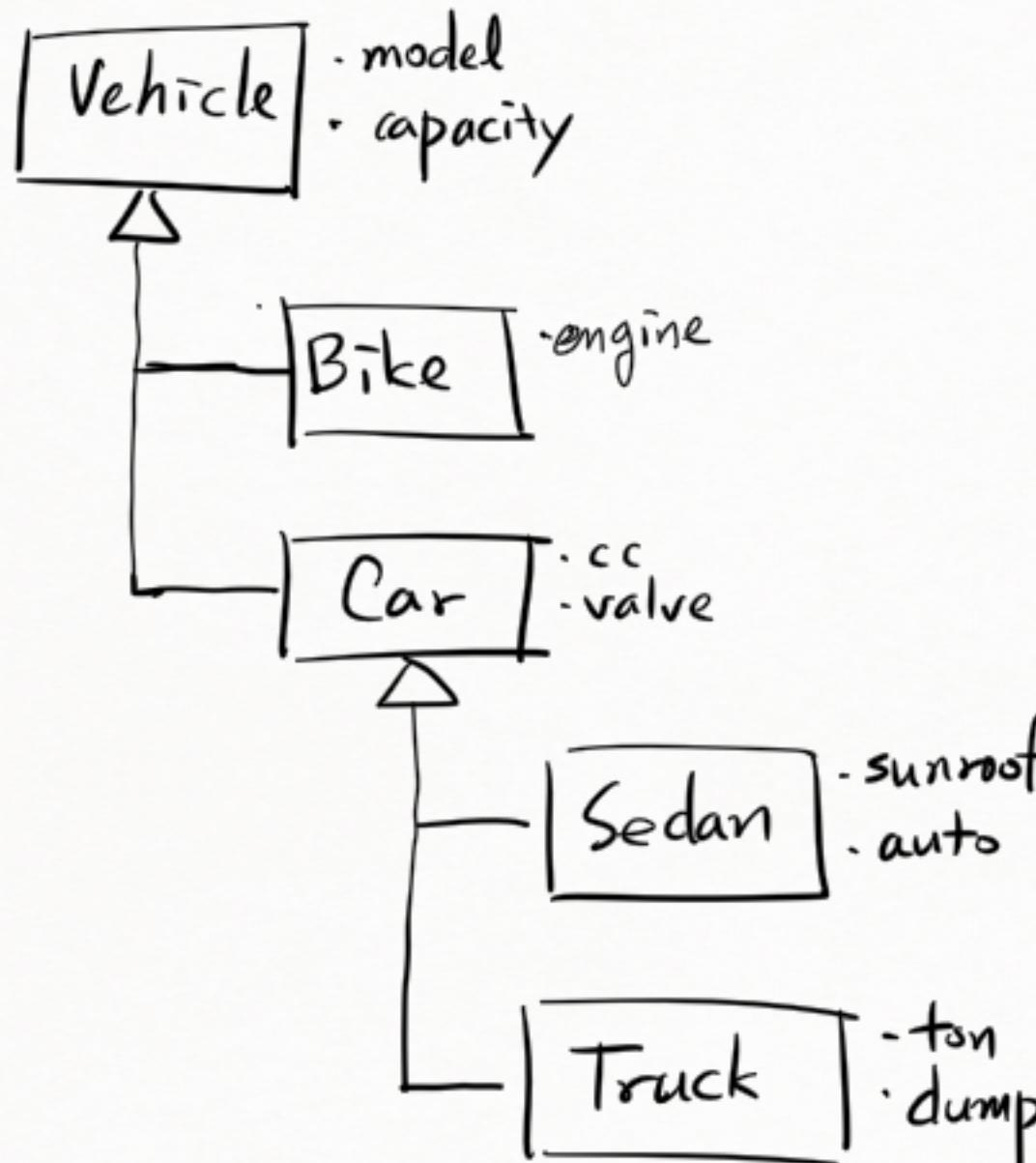


Car c1;
~~c1 = new Vehicle();~~ → **기타일 가능!**

Car의 인스턴스 멤버를 초기화하는 경우,
Car의 인스턴스 멤버를 초기화하는 경우,
~~c1.model = "t4t4";~~ } **Vehicle의 인스턴스 변수**
~~c1.capacity = 5;~~ } **?**
~~{ c1.cc = 2000;~~ } **Car의 인스턴스 변수**
~~c1.valve = 16;~~

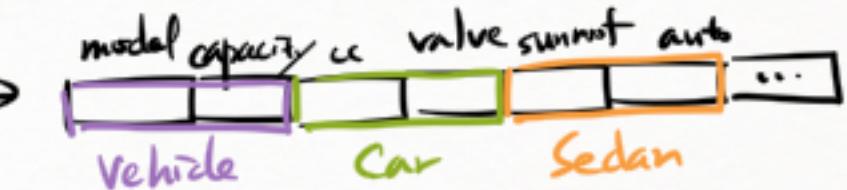
현재 초기화된 멤버는 차량 타입을
사용할 수 있기 때문에
이제 차량별로 전파될 수 있을 것
프로그램 초기화 단계에서 한다

* 다형성 (polymorphism) : oop. exob. a



Car c1;

c1 = new Sedan(); →



c1. model = "aaa"; } Vehicle
c1. capacity = 5; } Vehicle

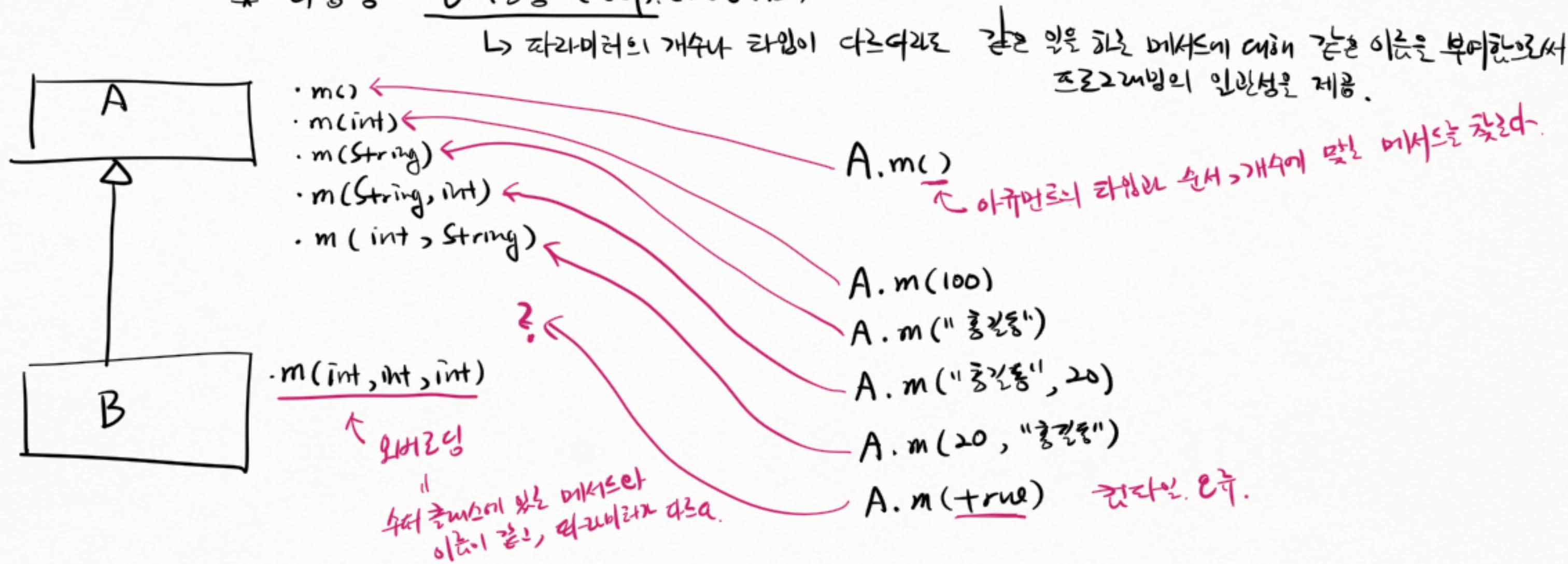
c1. cc = 2500; } Car
c1. valve = 16; } Car

c1. sunroof = true; } Sedan
c1. auto = true; } Sedan

실행 결과

- c1은 Car 클래스의 인스턴스이므로
 - 어떤 Car 클래스 변수에 할당해도 내용은 차이가 없어짐
- c1이 실제 Sedan 객체를 가지려고 오류나오면

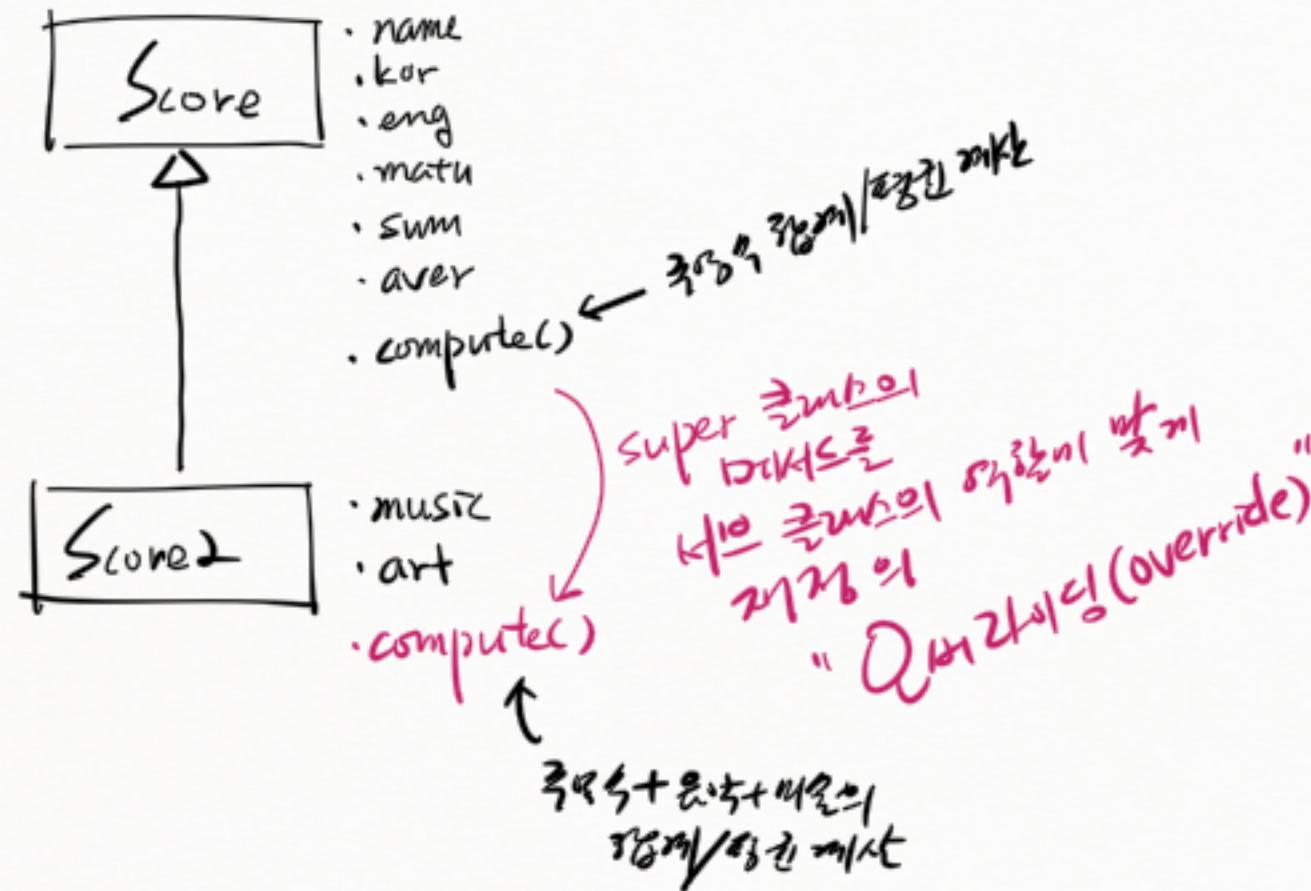
* 다형성 - 오버로딩 (Chap. ex06.b)



* 다형성 - 오버라이딩 (overriding)

com.eecs.oop.ex6.c

↳ 상속 받은 메서드를 서브 클래스의 예외에 맞춰 재정하는 것.



new Score2()

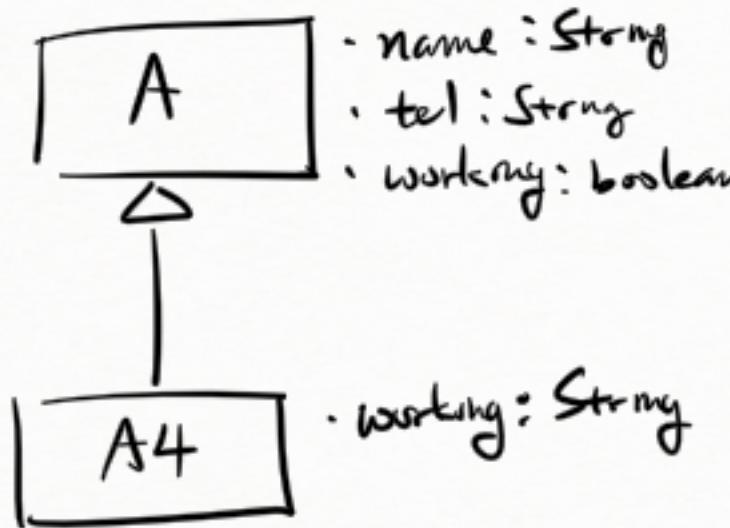


name	kor	eng	math	sum	aver	music	art	...
null	100	100	100	300	100.0	50	50	...

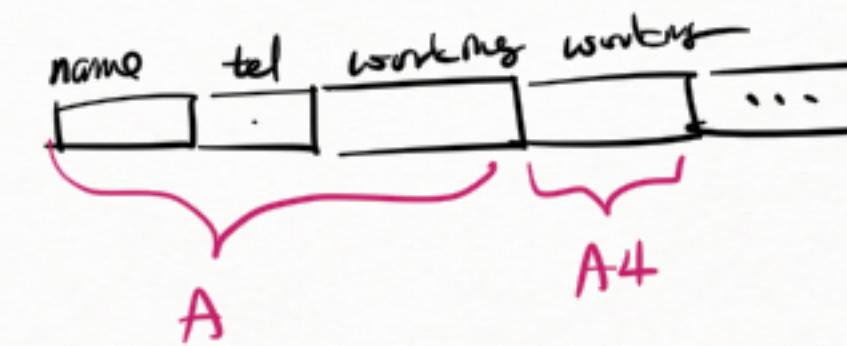
Diagram showing object state after instantiation:

- The object is named "Score2".
- Attributes: kor=100, eng=100, math=100, sum=300, aver=100.0, music=50, art=50.
- Handwritten notes in pink:
 - "`Score2`의 `compute()`는 `Score`의 `compute()`를 상속 받았습니다."
 - "`Score2`의 `compute()`는 `Score`의 `compute()`를 오버라이딩합니다."

* 초기화 - 힐드 초기화



A4 obj = new A4()

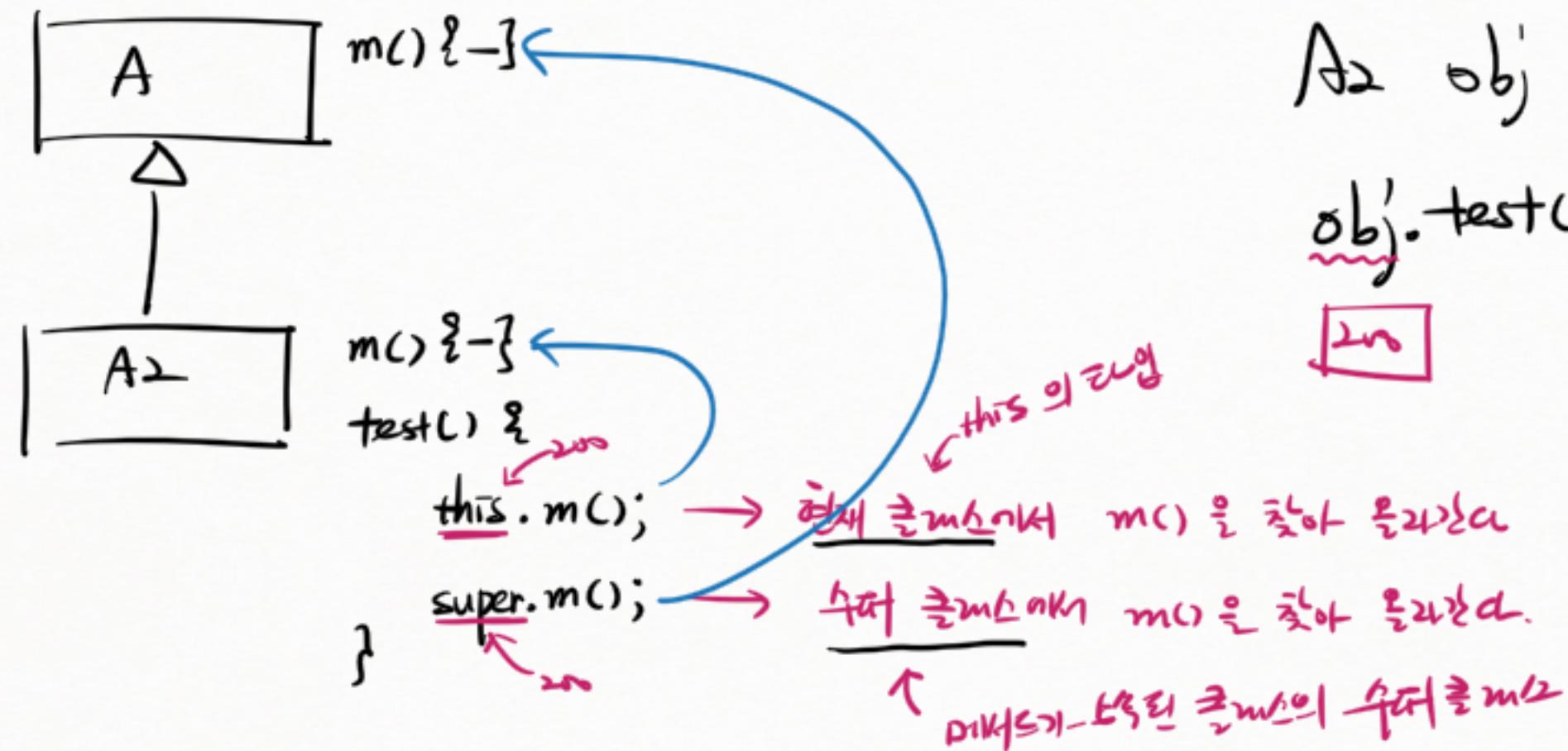


obj.name = "—";

obj.tel = "—";

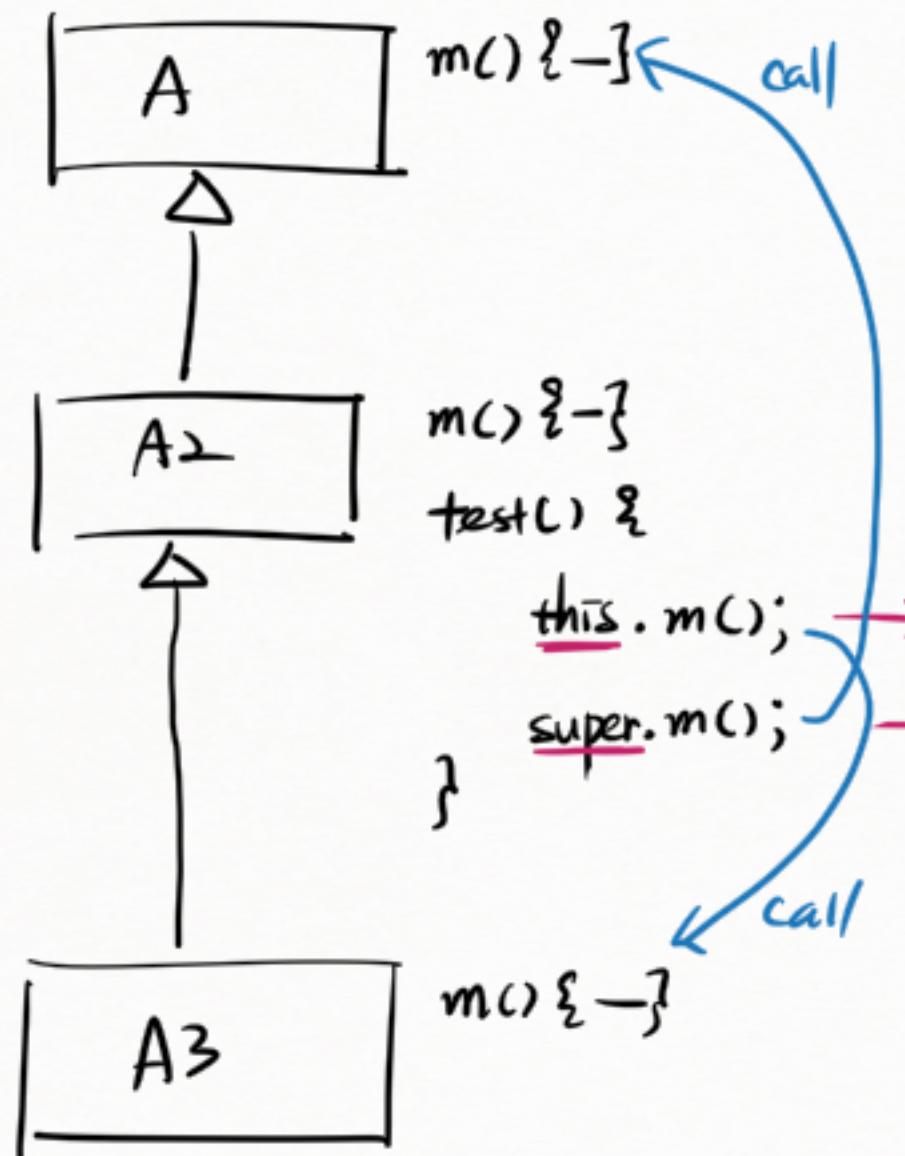
obj.working = ?

* 오버라이딩 - super 키워드 (99. ex06.c. Exam0410)



A2 obj = new A2();
obj.test();
200

* 오버라이딩 - super 키워드 (opp. ex6. c.Exam0411)



A3 obj = new A3();

obj.test();

↑ 상속받은 메서드를 호출할 경우

this의 실제 태입

↳ test()는 호출될 때 실제로 객체의 태입 (A3)

현재 클래스에서 m()을 찾아 올라간다

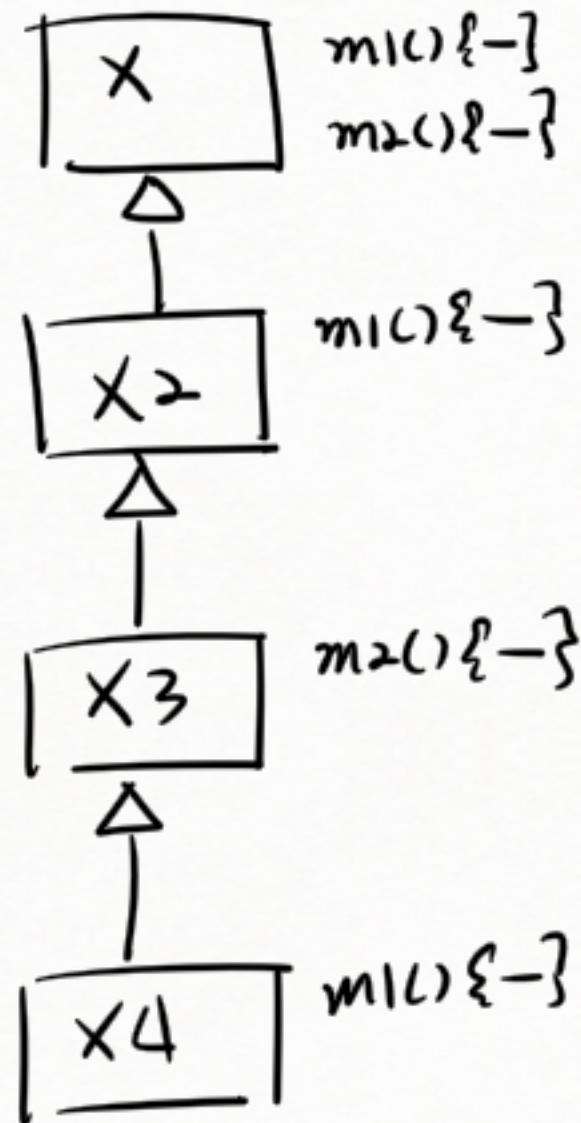
부모 클래스에서 m()을 찾아 올라간다.

메서드가 소속된 클래스의 부모 클래스

(A)

* this et super 例題

Exam 420



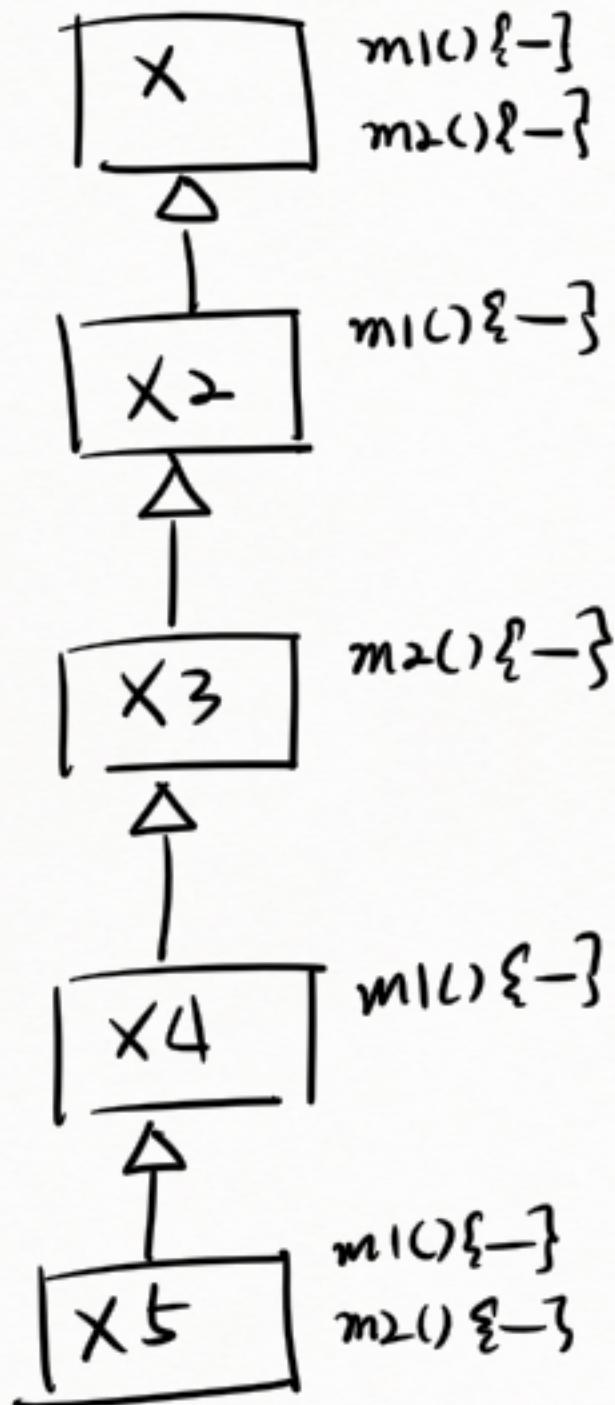
test()
 this.m1(); → X4.m1()
 super.m1(); → X2.m1()
 this.m2(); → X3.m2()
 super.m2(); → X3.m2()

X4 obj = new X4();
obj.test();
 ↑
 this

Internationalization
Localization

* this 와 super 차이

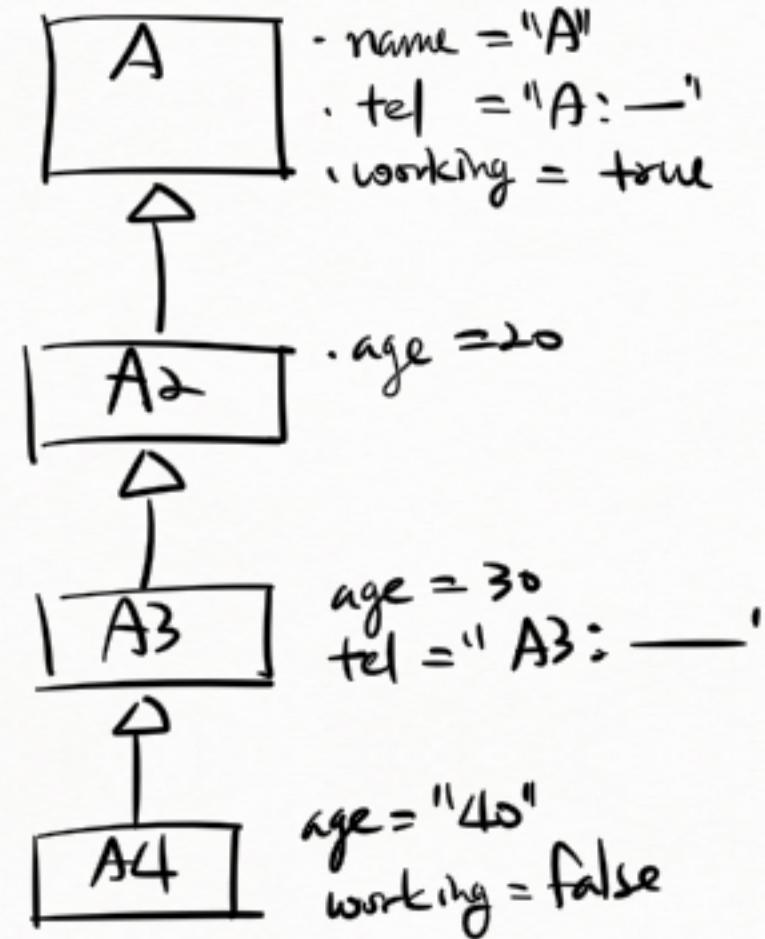
Exam 0421



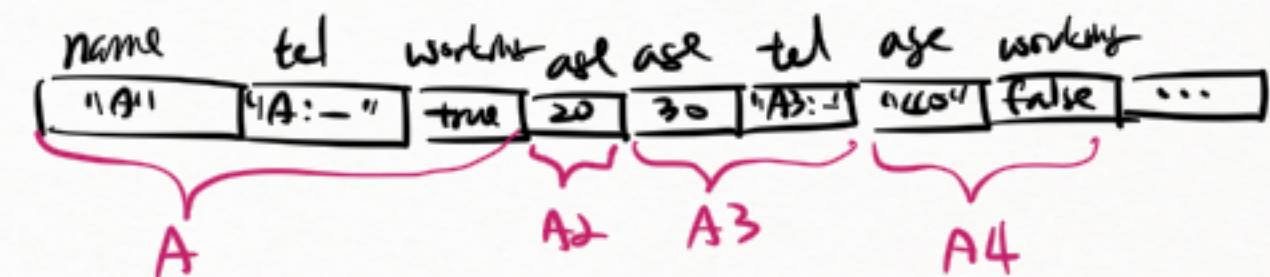
X5 obj = new X5();
obj.test();
↑
X4의 test()는?

test() {
 this.m1(); → X5.m1()
 super.m1(); → X2.m1()
 this.m2(); → X5.m2()
 super.m2(); → X3.m2()
}

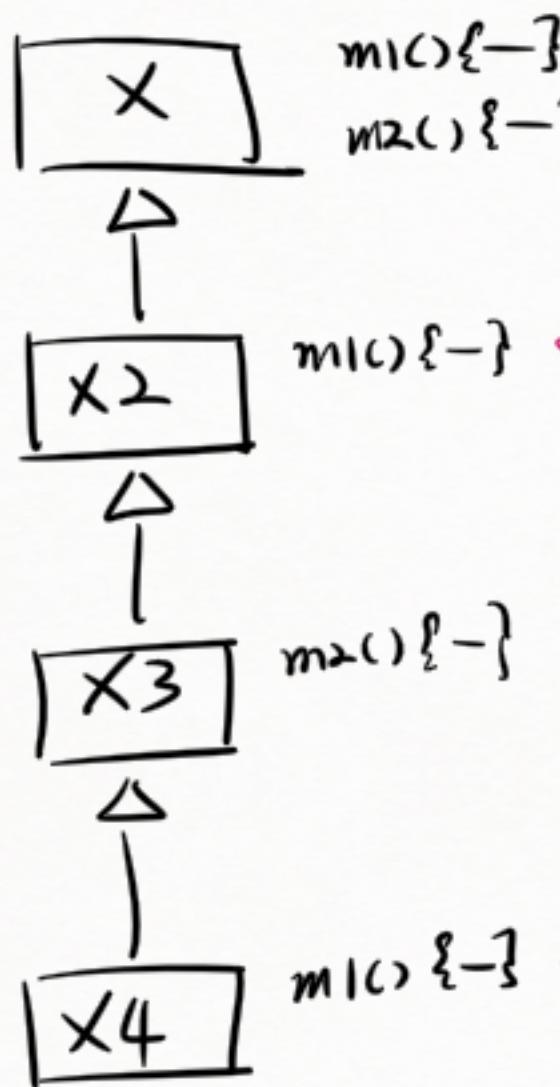
* 2) 디렉스와 퍽스



A4 a4 = new A4();



* 레퍼런스와 메서드



$x4 \quad x4 = \text{new } X4();$

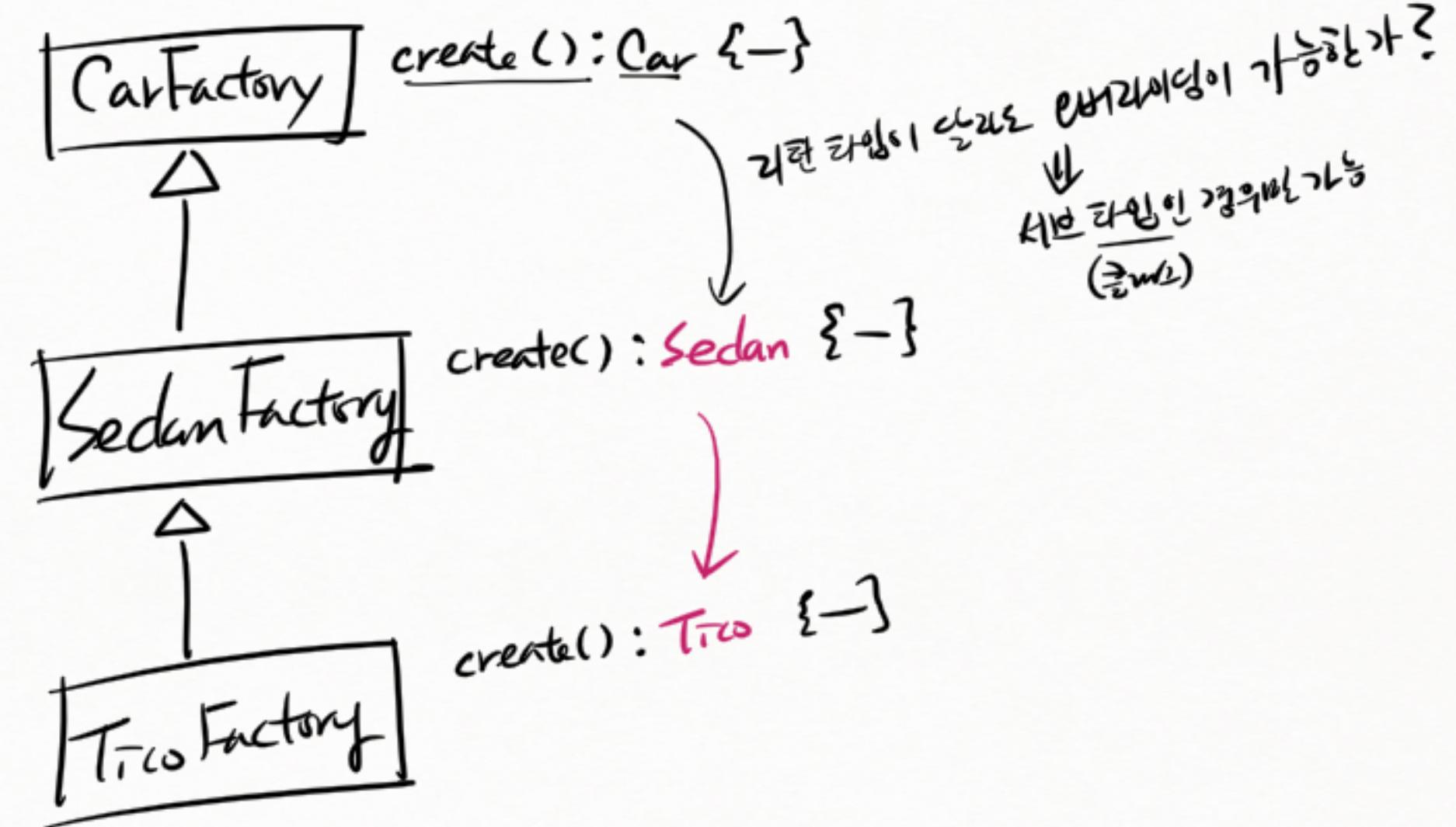
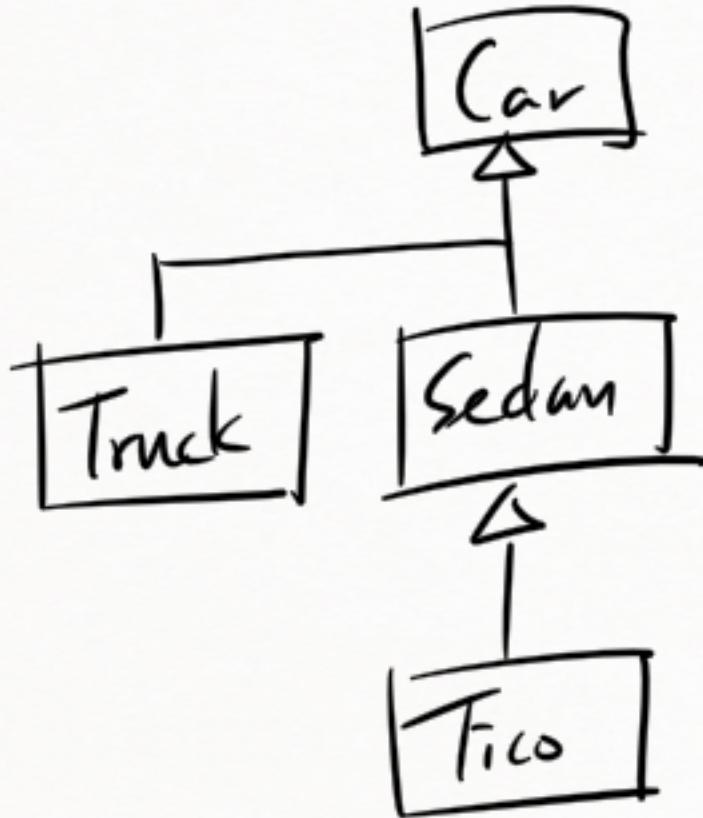
$x4.m1();$

$\underline{x2} \quad \underline{x2} = \underline{x4};$

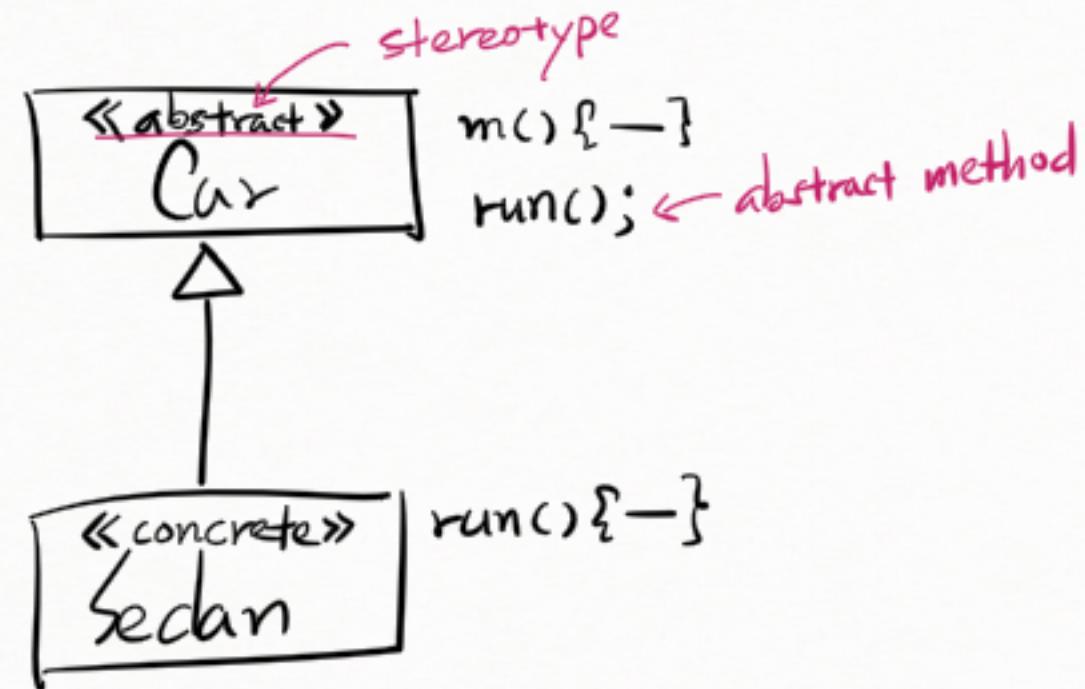
$\underline{x2.m1();}$

실제 레퍼런스가 가리키는 객체의 태입이나 초기화를 바꿀 수 있다

* 구조화된 구조화된



* 추상클래스의 추상메소드를 상속받아쓰고 레퍼런스



Car c = new ~~Car()~~;

↑
추상클래스로
인스턴스 생성 불가!

Car c = new Sedan();

c.run();

↑
하지만 c가 가리키는 객체의 빌드는 허용
(Sedan ~ m1)