* 클래스 문법 활용 예

Exam0110
main()

· 낱개 변수 사용

+ 메서드 →

Exam0111
· main()
· printScore()

· 메서드 문법 활용
↳ 파라미터 사용

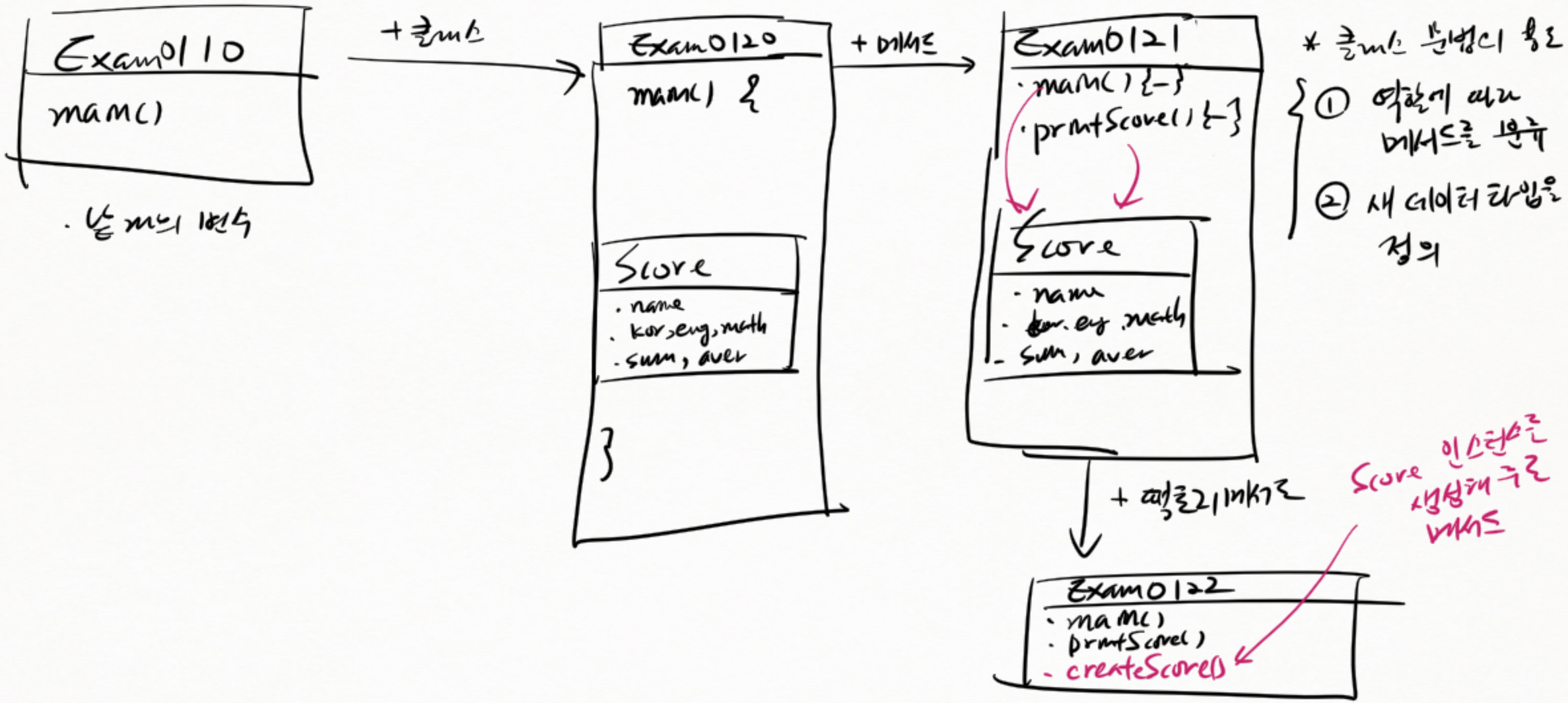
⇓

✓ 중복 코드 제거

↓

코드 재사용 ↑

✓ 유지 보수가 쉬워진


Exam0110
main()
· 날씨의 변수
+ 클래스
Exam0120
main() {
Score
· name
· kor, eng, math
· sum, aver
}
+ 메서드
Exam0121
· main() {-}
· printScore() {-}
Score
- name
- kor, eng, math
- sum, aver
* 클래스 문법의 용도
① 약속에 따라 메서드를 분류
② 새 데이터 타입을 정의
+ 팩토리 메서드
Score 인스턴스를 생성해 주는 메서드
Exam0122
· main()
· printScore()
- createScore()

* 레퍼런스 비교 → 여러 개의 인스턴스를 다루기

Score s1, s2, s3;

```
s1.name = "___";
s1.kor = 100;
   ⋮
```

* 레퍼런스 배열

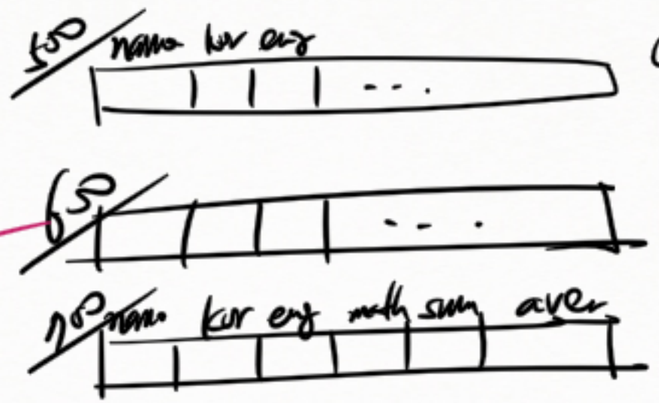
```
Score[] arr = new Score[3];
```

null?
- 물건이 아니다!
- 쥐고 있지만 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



new Score();

Score 클래스의 선언된 변수를 Heap 메모리 영역에 순서대로 만든다.

* 레퍼런스 와 인스턴스

```
Score s1;
```

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

```
Score s2;
```

```
s2 = s1;
```

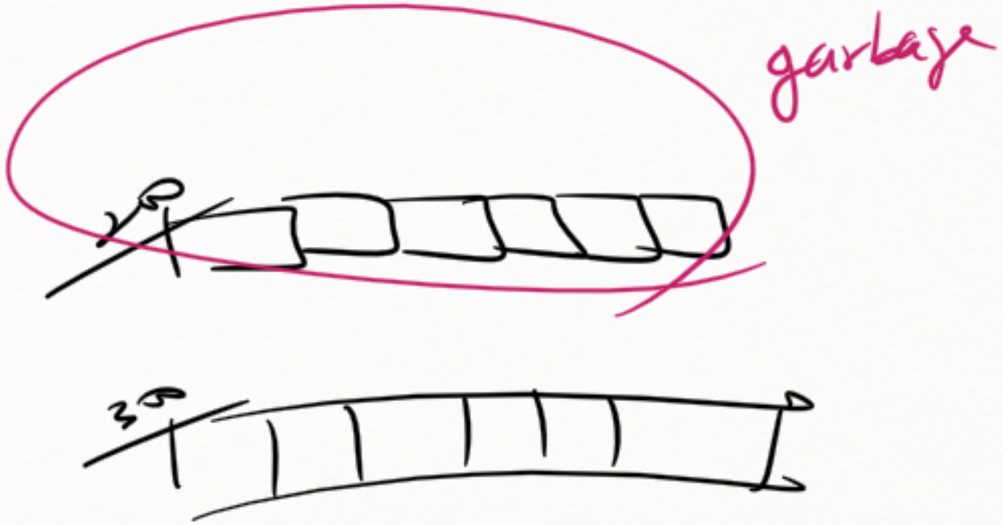
```
s2.name = "홍길동";
```

* 가비지 (garbage)

Score s1;

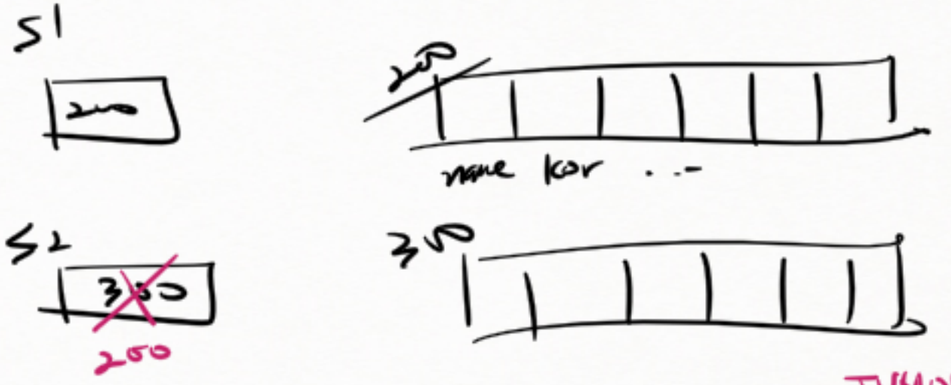
s1 = new Score();

s1 = new Score();



* 레퍼런스 카운트와 가비지

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

s2 = s1;
```



JVM이 관리

레퍼런스 카운트 테이블

인스턴스	참조회수
200	2
300	0

레퍼런스 카운트가 0인 경우 "garbage"가 된다.

* 클래스 리뷰

com.eomcs.oop.ex01

```
class Exam0410
    class Score{-}
    main() {
        class Score{}
    }
    static class Score{-}
```

```
class Score{-}
```

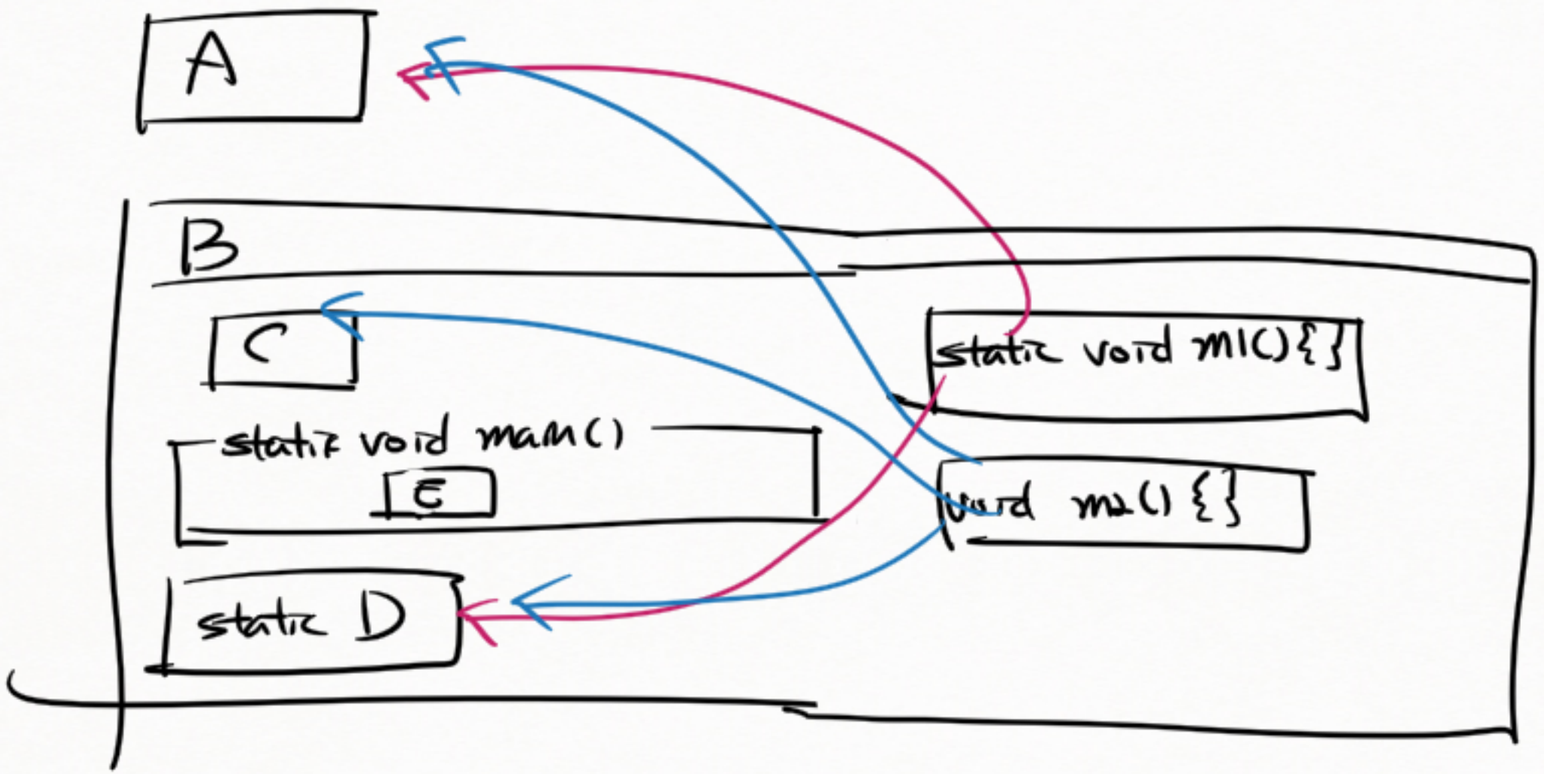
(nested)
중첩클래스 : non-static nested class
= inner class

중첩클래스 ⇒ 로컬클래스

중첩클래스 : static nested class

패키지 멤버 클래스

* static 멤버와 non-static 멤버



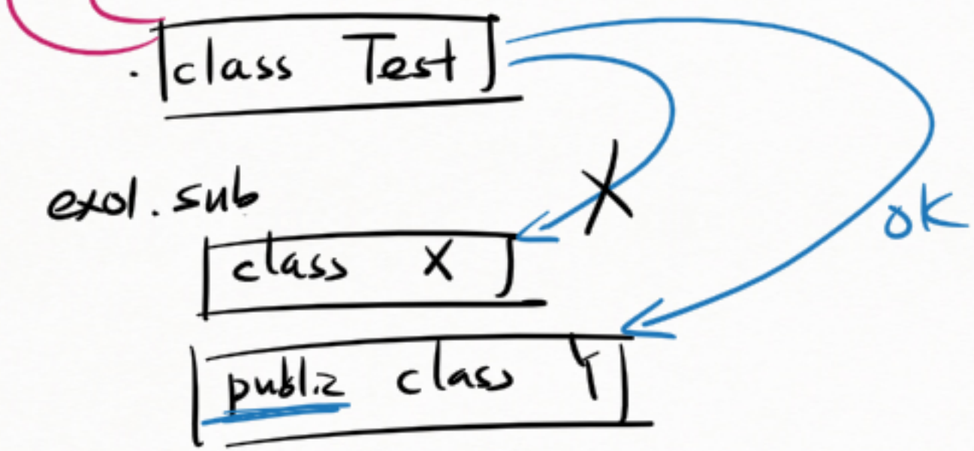
* 공개 클래스와 기본 클래스

ex01

- class X
- public class Y
- class Test

ex01.sub

- class X
- public class Y



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

User-defined Data Type
개발자

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

레퍼런스 → 200

Score의 인스턴스

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

```
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)

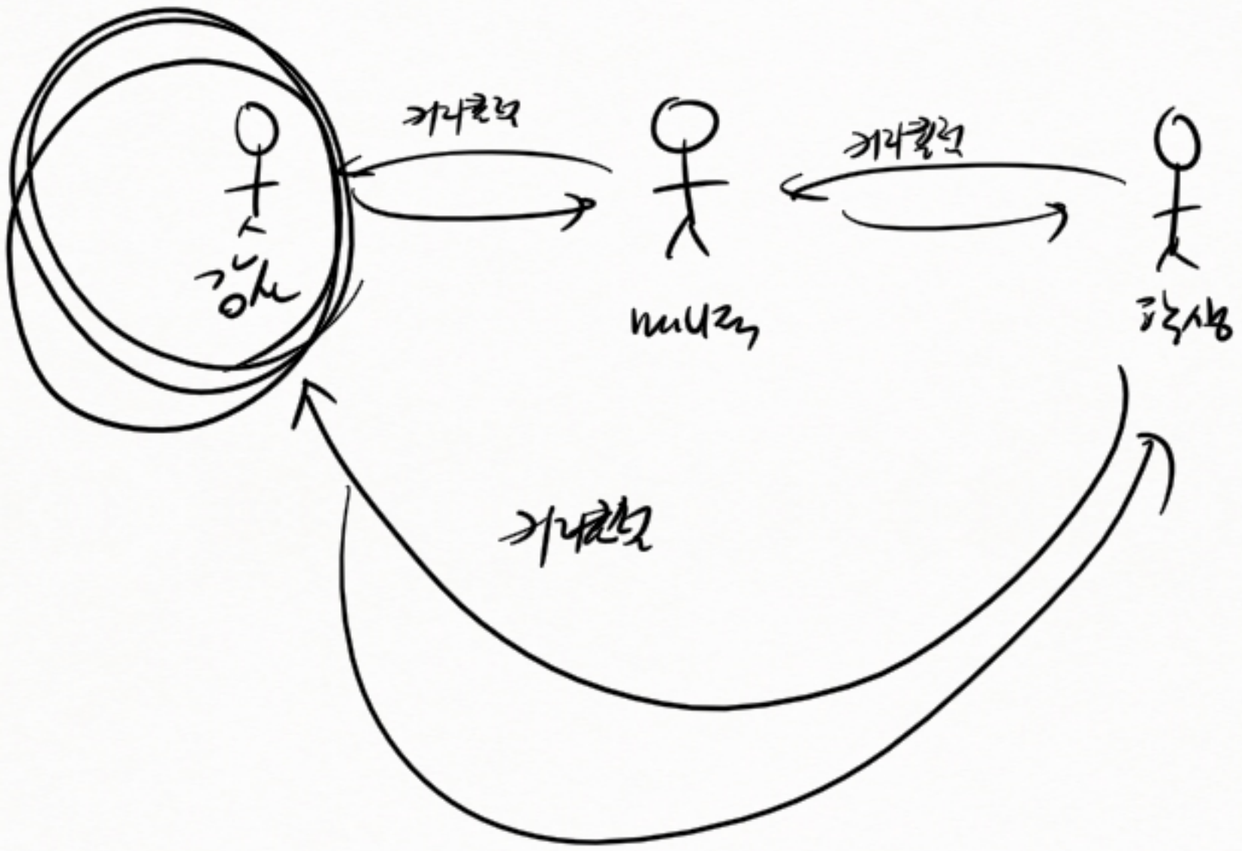
⑤ s1의 name 필드 (변수)

※ optimizing (최적화) vs refactoring (코드 개선)

optimizing ↓
- 속도↑
- 코드 난해 → 유지보수↓

refactoring ↓
- 유지보수 용이성↑
- 속도↓

x GRASP : 클래스에 책임 부여 방법론 +



* static 메서드 와 인스턴스 메서드

Score.compute(s1)

클래스명

data로 생성된 인스턴스 주소

static method

s1.compute( )

인스턴스 주소

instance method

* 인스턴스 메서드 와 연산자

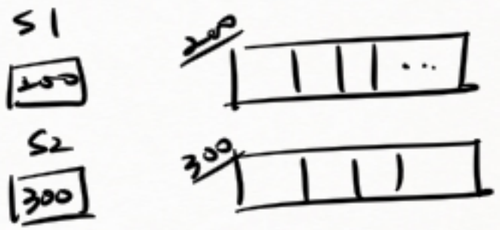
```
i ++;
```

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 많이 분리하기

① 분리 전 → ② 메서드 분리 → ③ 클래스 변수 → ④ 클래스 변수의 한계

static 변수
=
클래스 변수

Exam0210

- main()
- plus()
- minus()
- multiple()
- divide()

refactoring
⇓
① extract class
② move method

Exam0220

- main()
  ↳ result

Calculator

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

역할을 분리하니
관리 용이
↓
기능 추가도 쉬움

GRASP "high cohesion"
객체지향 프로그래밍의 일반 원칙
높은 응집력

Exam0230

- main()
  ↳ ~~result~~

Calculator

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

메서드의 작용 범위를
외부에 두지 않고
내부에서 관리
⇓
유지보수도 쉬움

result 변수를
단 한 번만
위치
↓
동시 제어 불가!

* 클래스 문법 활용 : 메서드를 묶어 분류하기

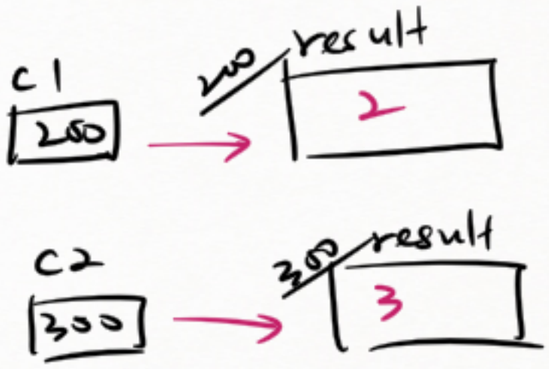
→ ① 인스턴스 변수 도입

Exam02to
- main()

Calculator
- plus() - minus() - multiple() - divide() - abs() } static - result ← 인스턴스

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

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

작업결과를 저장할 result 변수가 있는 인스턴스 주소를 알려준다.



* 클래스 문법 활용 : 메서드를 묶어 분류하기

→ ⑥ 인스턴스 메서드 활용

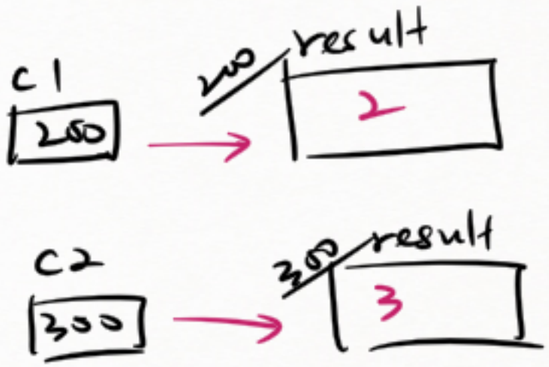
Exam0260
- main()

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

non-static
=
인스턴스 멤버

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

c1.plus(2);
c2.plus(3);
```

인스턴스 주소 전달



피연산자

1 + 2

c1.plus(2)

연산자

* 클래스 문법 활용 : 메서드를 묶어 분류하기

→ ① 패키지 없이 클래스 ⟶ ② 패키지 묶음

Exam0240
- main()

com.eomcs.oop.ex02.

Exam0240.java

Calculator.java

com.eomcs.oop.ex02.Exam0280

com.eomcs.oop.ex02.util.Calculator

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

non-static
=
인스턴스 변수

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

⇑

클래스를 쓰기 전에
패키지 선언을 컴파일러에게
알려줌.

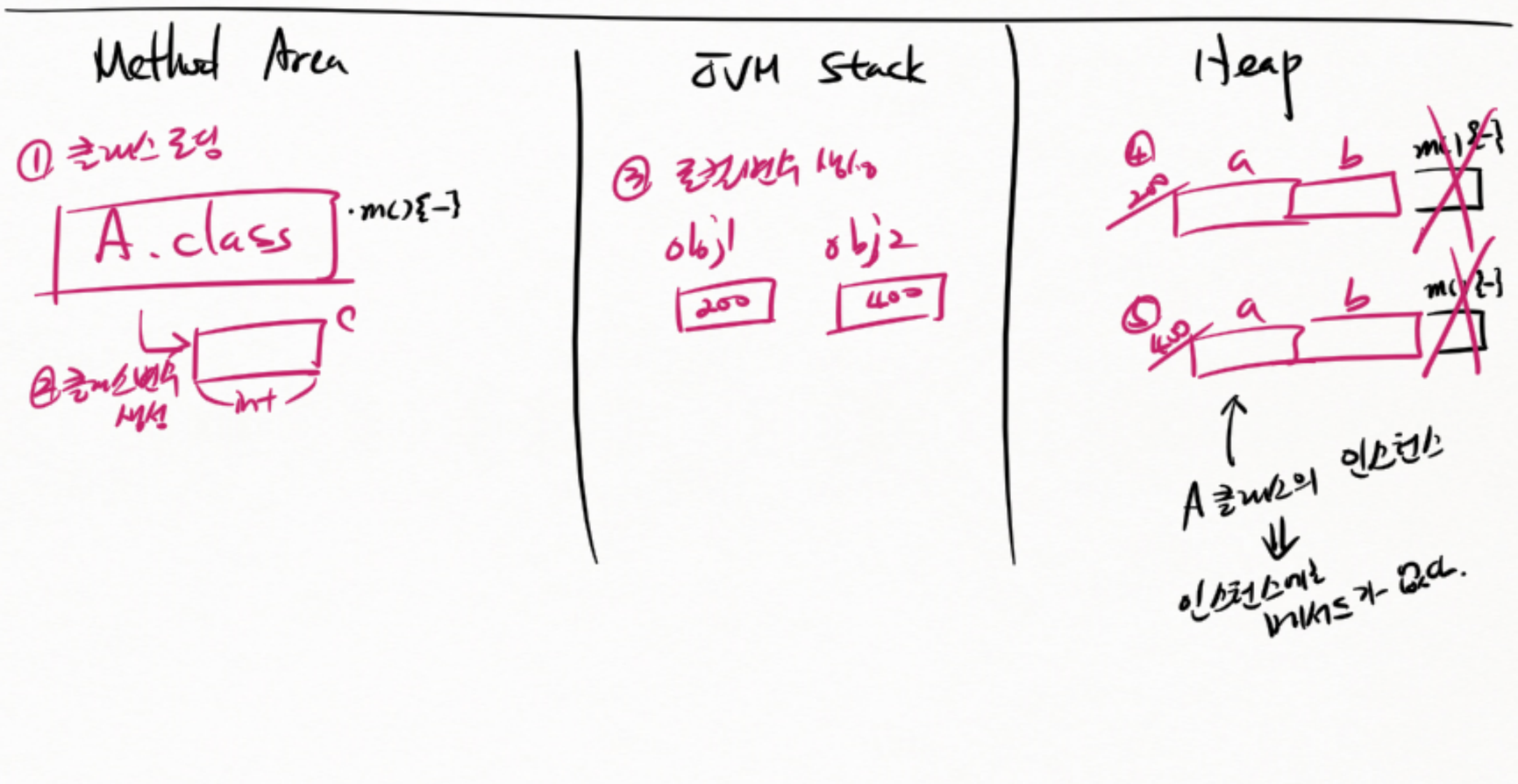
* 클래스 변수 와 인스턴스 변수

static 변수 non-static 변수

```
class A {
    int a;
    int b;
    static int c;
    void m() {—}
}

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

인스턴스 변수

클래스 변수



* 인스턴스 변수와 인스턴스 메서드

데이터 타입

```
class Calculator {
    int result;
    void plus(int v){
        result += v;
    }
    ⋮
}
```

데이터 타입의 변수 값을 다루는 연산자!
(메서드)

* Object 클래스

↳ 자바의 최상위 클래스

java.lang.

Object 자동차

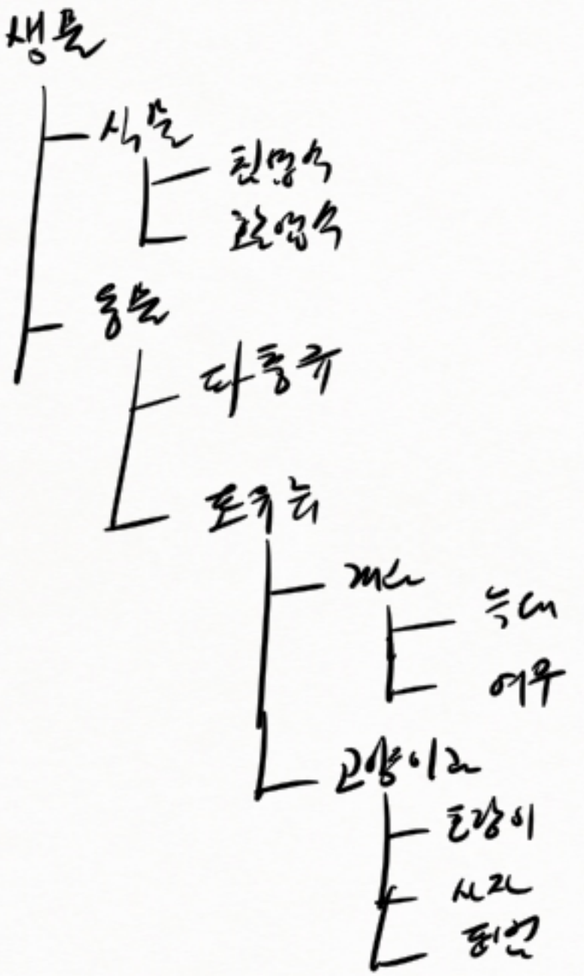
- String 문자열
- Integer 숫자
- Member 멤버
- Board 게시판

자바의 모든 클래스는 Object의 자손 클래스 이다.

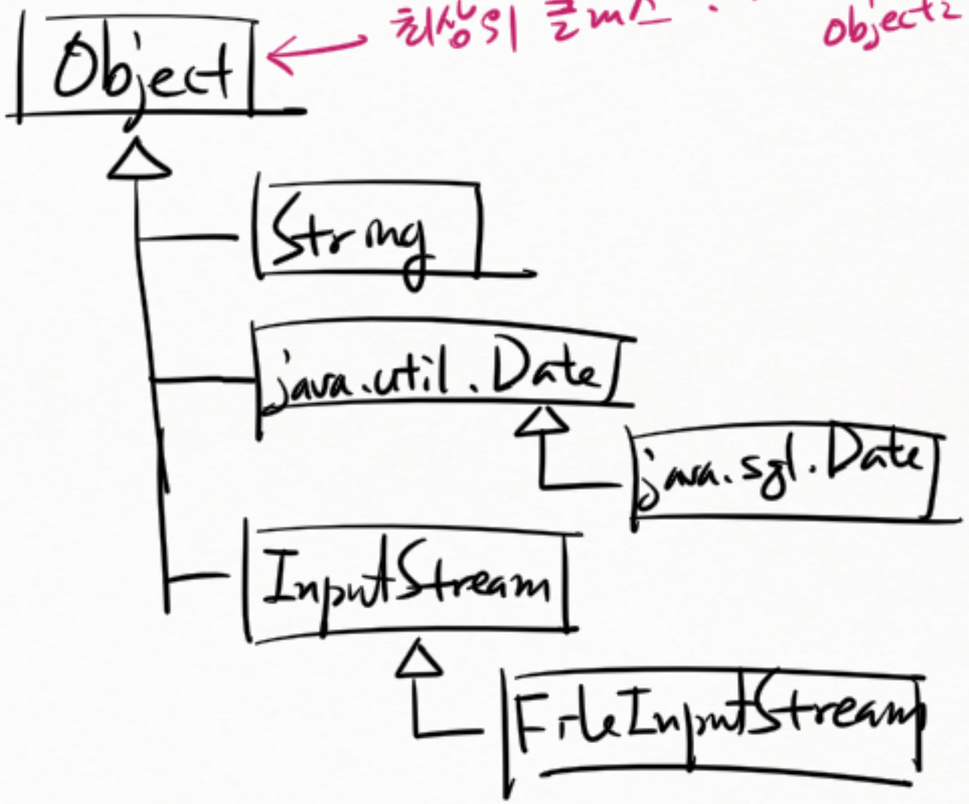
하위(sub)

* 분류 나 자바 클래스

분류 계층도



자바의 클래스로 상위, 하위 계층을 구성하고 있다.



* 상위 클래스와 하위 클래스 : 코드 공유를 위해!

super 클래스 = parent 클래스

Object ← 모든 클래스가 기본으로 가져야 하는 필드와 메서드가 정의되어 있다.

기능을 "상속한다" (inheritance)

sub 클래스 = child 클래스

String ← Object의 필드와 메서드 사용법을 알고 있다

+

문자열을 다루기 위해 필요한 필드와 메서드가 추가되어 있다

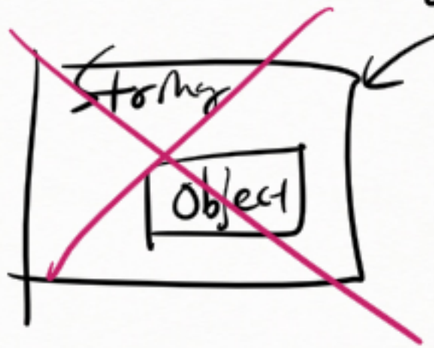
↳ 주의!

- object의 필드와 메서드를 가져온 것이 아니다.

object의 기능을 사용할 수 있도록 정리해 가진다.

Object ← - - - - - - String

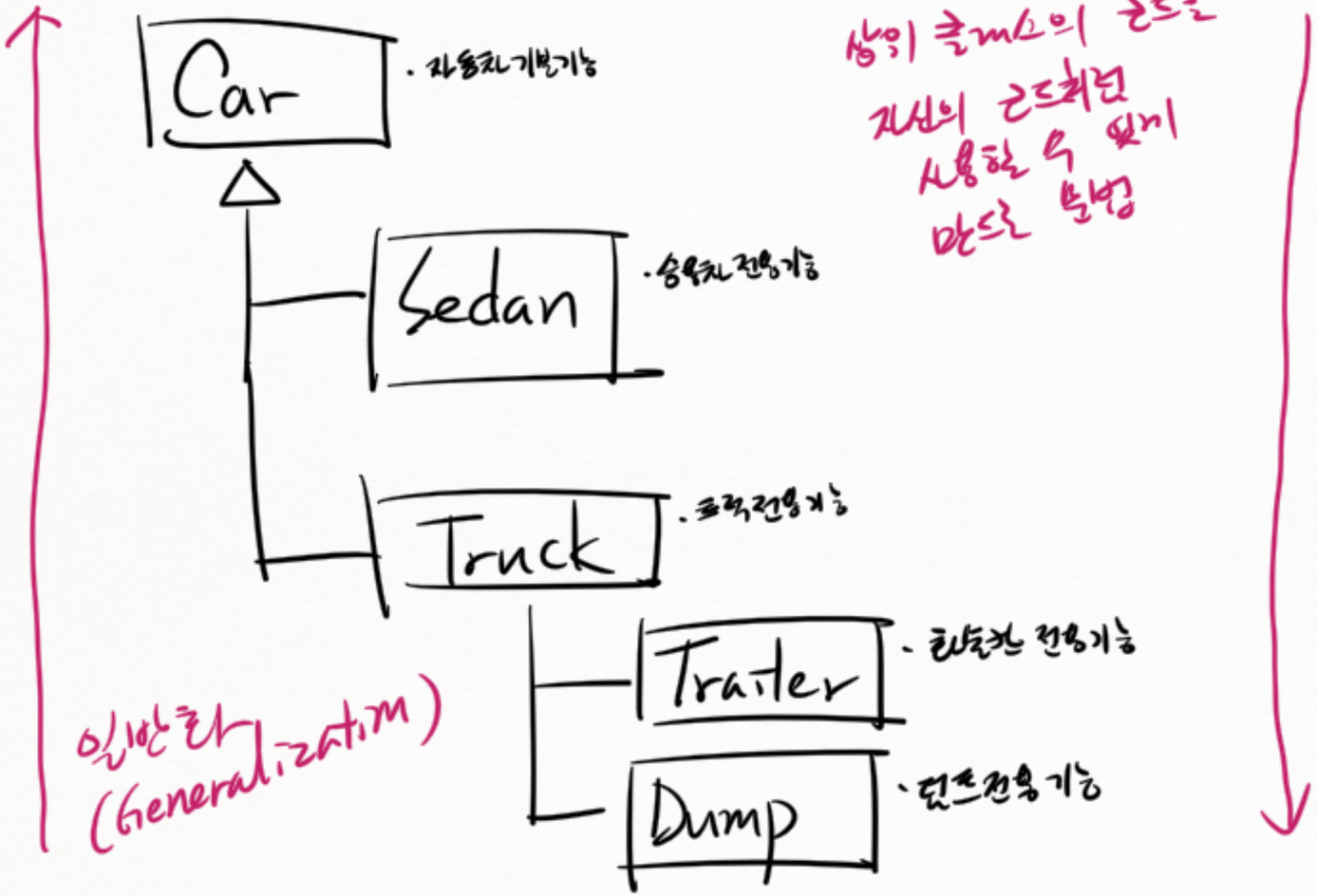
Object의 코드를 가져오는 것이 아니다!



* 상속 문법: 코드 재사용 방법 중 하나.

↓

상위 클래스의 코드를 자신의 코드처럼 사용할 수 있게 만드는 문법



전문화 (Specialization)

* 상속 문법과 다형성

- 다형적 변수 ☆
- 오버로딩 (overloading)
- 오버라이딩 (overriding) ☆

```
Car c;

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

다형적 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 수퍼클래스 지정하기

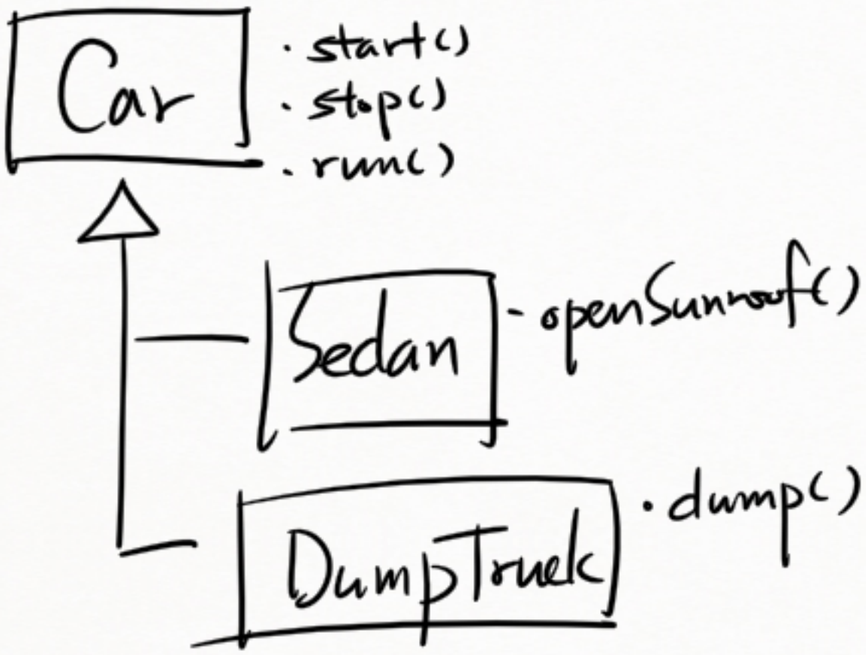
```
class Board extends Object {
    ≡
}
```

생략가능!

```
class Member {
    ≡
}
```

수퍼클래스를 지정하지 않으면 기본으로 Object가 수퍼클래스로 설정된다.

* 상속과 코드 사용



Car c;

c = new Car();

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

레퍼런스 타입의 기능은 변수 쓸 수 있다.

클래스

c = new Sedan();

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

실제로 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()
```

우리 클래스의 기능을 사용할 수 있다.

```
s = new Car()
s.start()
s.stop()
s.run()
s.openSunroof();
```

s의 타입은 Sedan이고 Sedan의 기능을 사용하려 할 것이다. 그런데 s가 Car를 가리키고 있어서 실행 오류가 발생할 것이다.

이런 잘못된 사용은 미연에 방지하기 위해 타입레퍼런스로 실제 객체를 가리키지 못하게 문법으로 막고 있다.

* static 필드와 non-static 필드

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

non-static 필드(변수) ⇐ Heap ⇐ Garbage Collector가 관리하는 영역

인스턴스

static 필드(변수) ⇐ Method Area

클래스

* 인스턴스 (non-static) 필드

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

    void m1(){
        ≡≡≡
    }
}
```

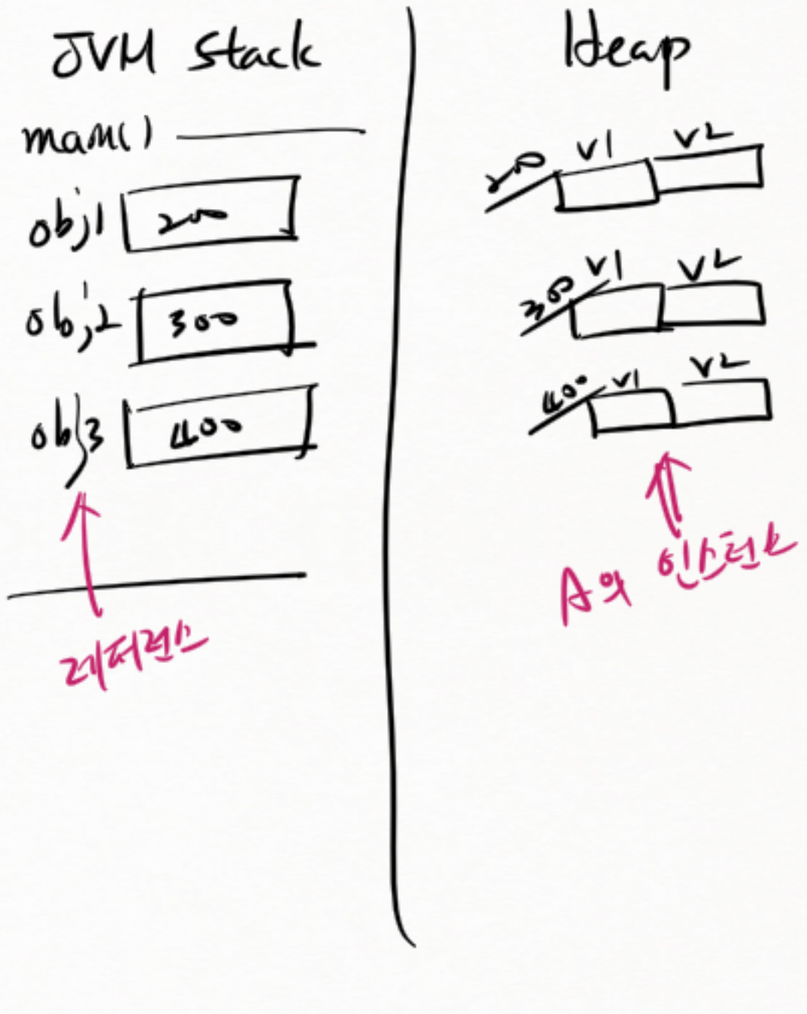
⟹

```
main() {
    A obj1 = new A();
    A obj2 = new A();
    A obj3 = new A();

    obj1.m1();
    obj1.v1 = 100;
}
```

인스턴스 생성 명령

local 변수

call 했을 때

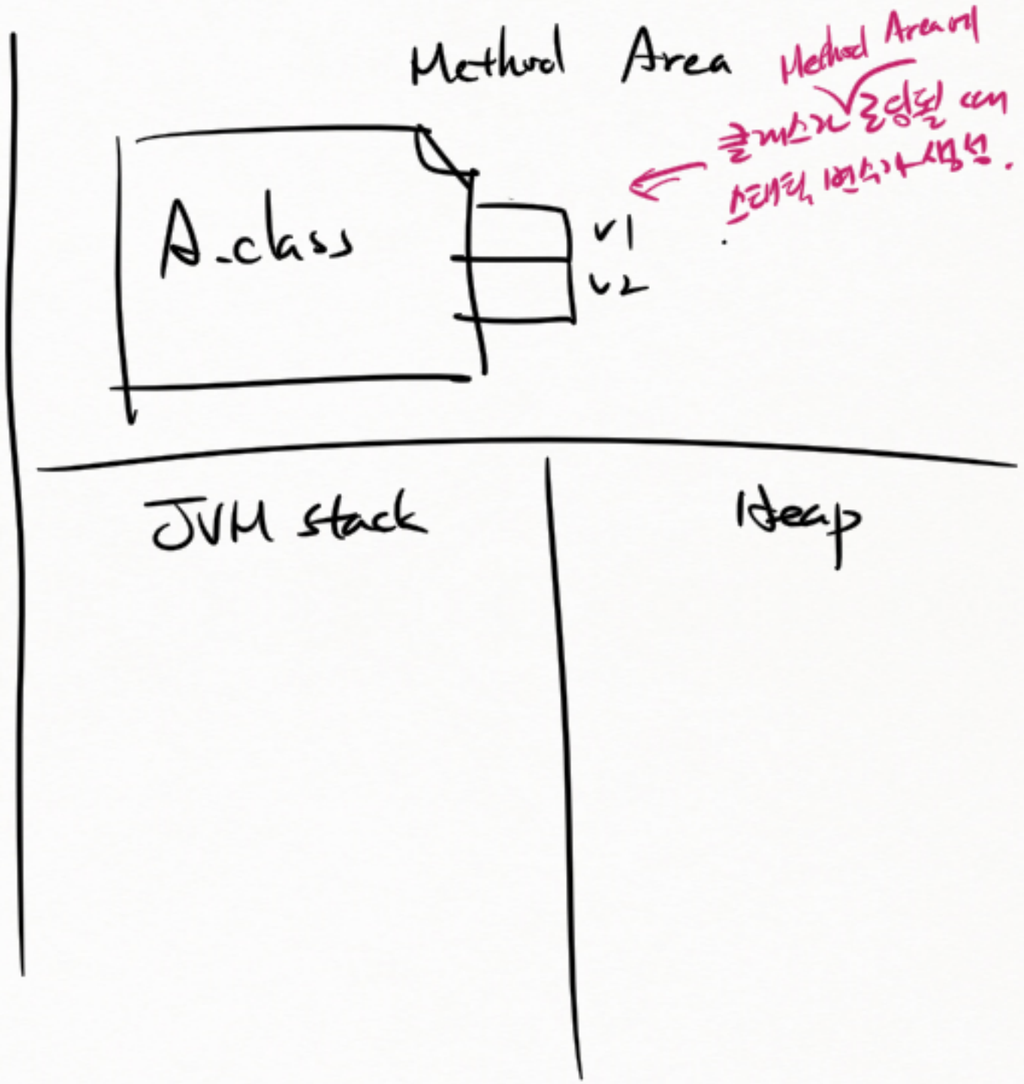


* 클래스(static) 필드(변수)

```
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

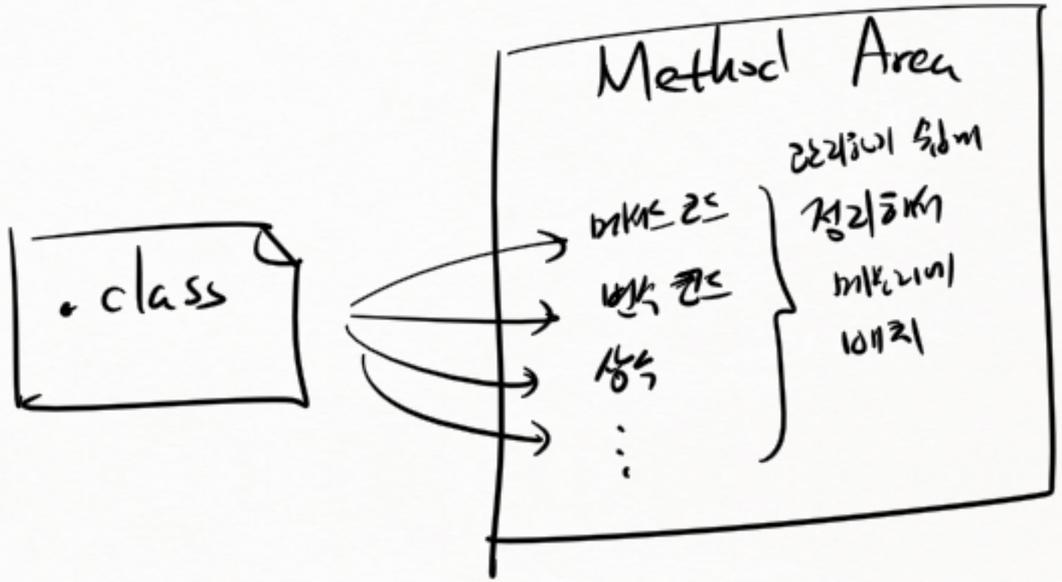
② Bytecode 검증

③ Method Area에 로딩 ⟹

④ 스태틱 필드 생성

⑤ 스태틱 블록 실행

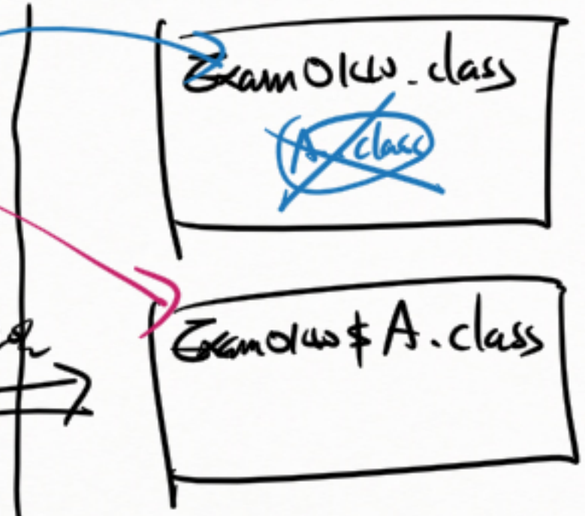
☆ ⑥ main() 호출

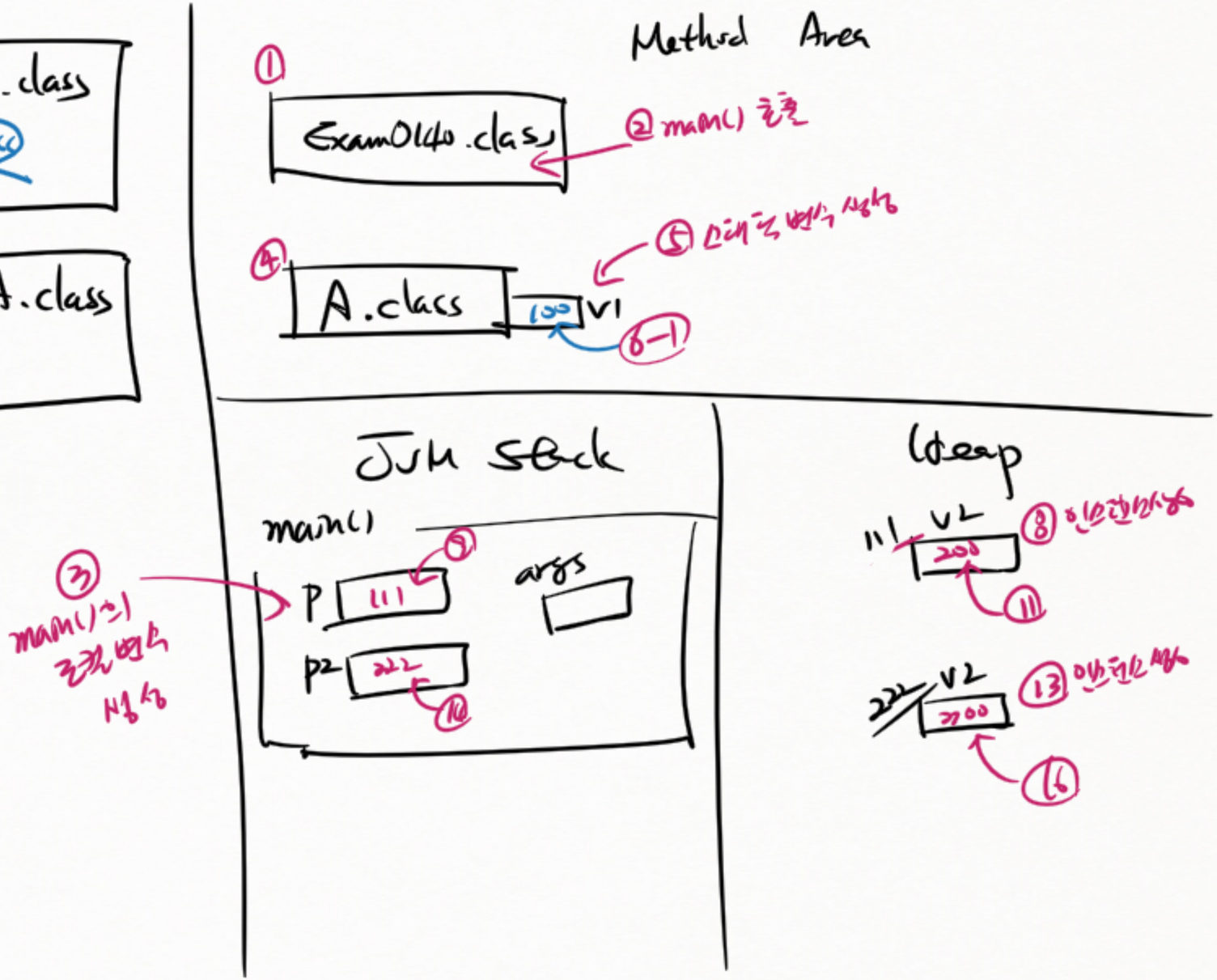


* 클래스 로딩, 스태틱 필드 및 인스턴스 필드 생성

```
class ExamOlleo {
    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 ⟹





* 인스턴스 변수와 인스턴스 메서드가 놓이는 위치

Method Area

Exam0230.class

```
main(){
  ≡
}
```

Calculator.class

```
result : int
✓ plus(){ —— }
✓ minus(){ —— }
```

JVM Stack

main()

args	c1	c2
	200	300

Heap

new Calculator()

~~200~~ result	~~plus(){-}~~	~~minus(){-}~~

new Calculator()

~~300~~ result	~~plus(){-}~~	~~minus(){-}~~

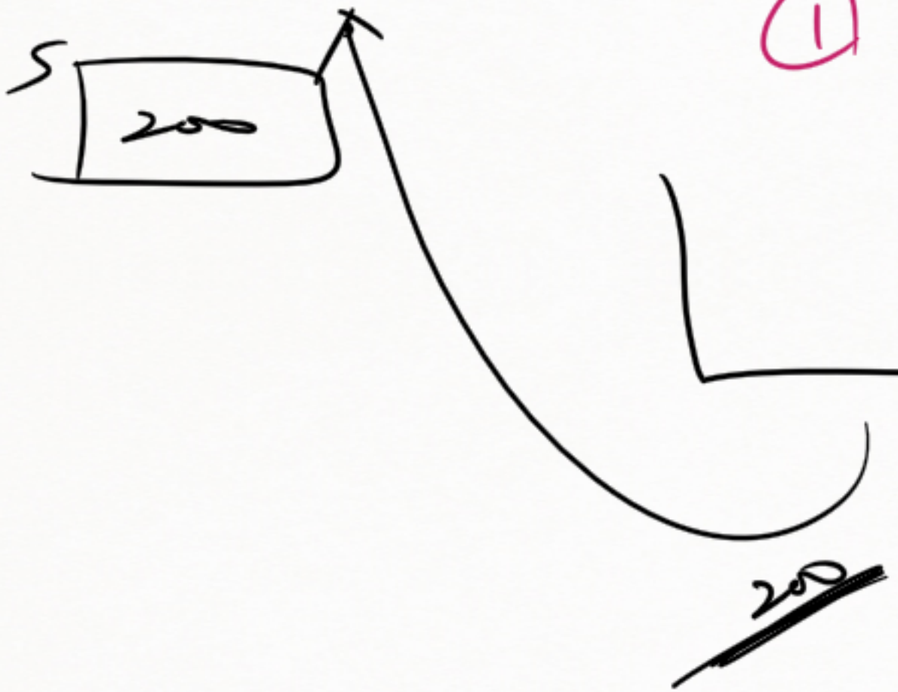
인스턴스를 생성할 때 메서드가 생성되는 것이 아니라 메서드는 Method Area에 있다.

* 인스턴스 생성과 생성자

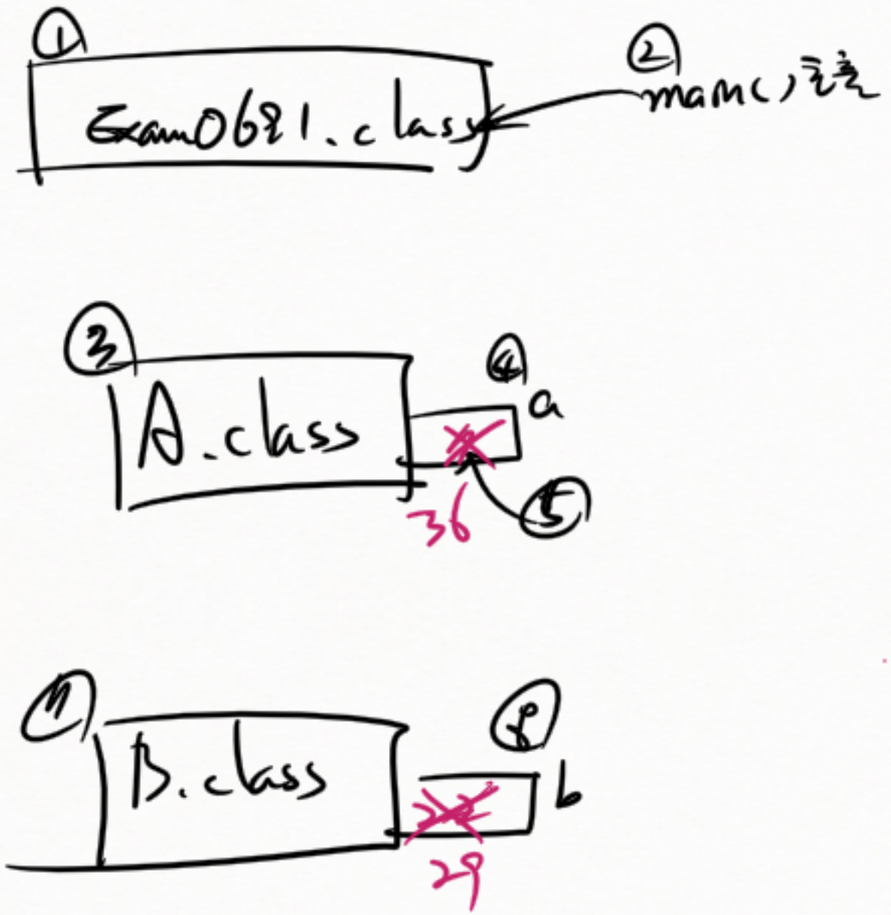
```
Score s = new Score();
```

① Score 클래스에 대해 인스턴스 변수는 Heap에 만든다

② 인스턴스 생성과 그 주소를 가지고 기본생성자를 호출한다.



인스턴스 변수 생성과 생성자호출이 끝나면 인스턴스 주소를 리턴한다



⑥
A.static{}

⑨
B.static{}

36

29

* 인스턴스 초기화 (instance initializer)

```
class A {
    A() {
        ○
    }
    A(int a) {
        ○
    }
    A(String s) {
        ○
    }
}
```

복사

복사

복사

인스턴스 초기화 블록

⇓

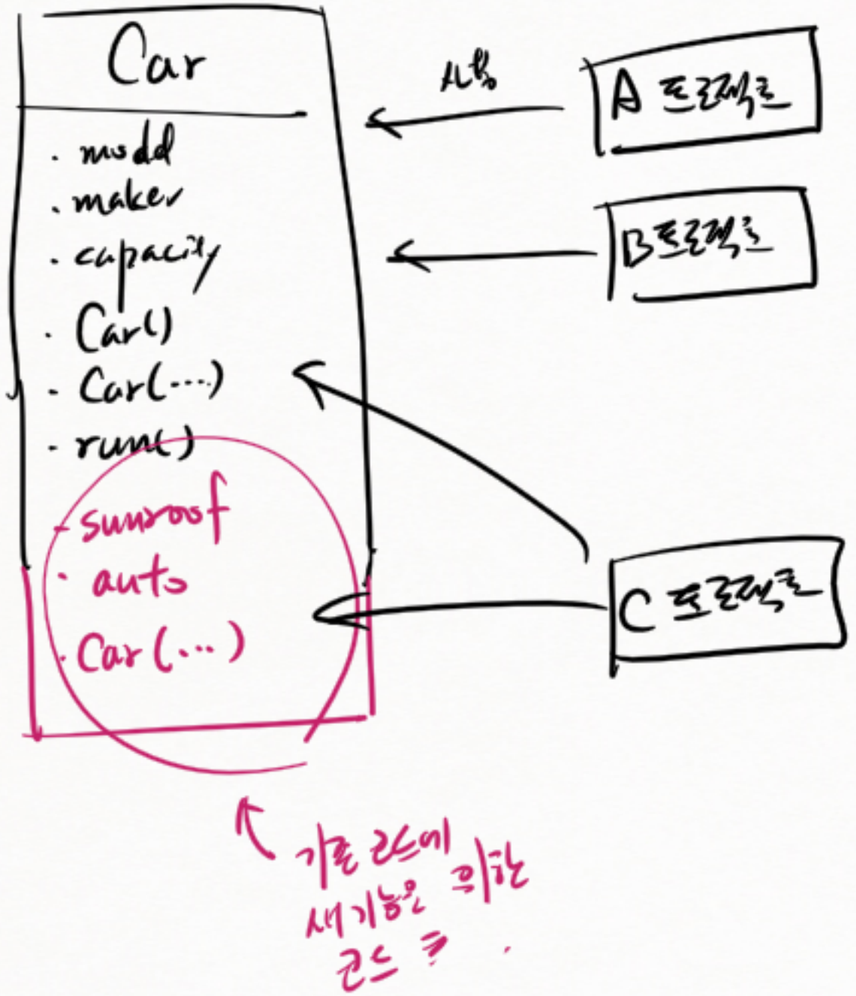
컴파일러가 모든 생성자의 앞부분에 복사한다.

⇓

여러 생성자에 공통으로 들어갈 코드가 있다면 인스턴스 초기화 블록에 넣어 두라!

* 기능을 확장하는 방법

* 기능을 확장하는 방법 — ① 기존 클래스 변경

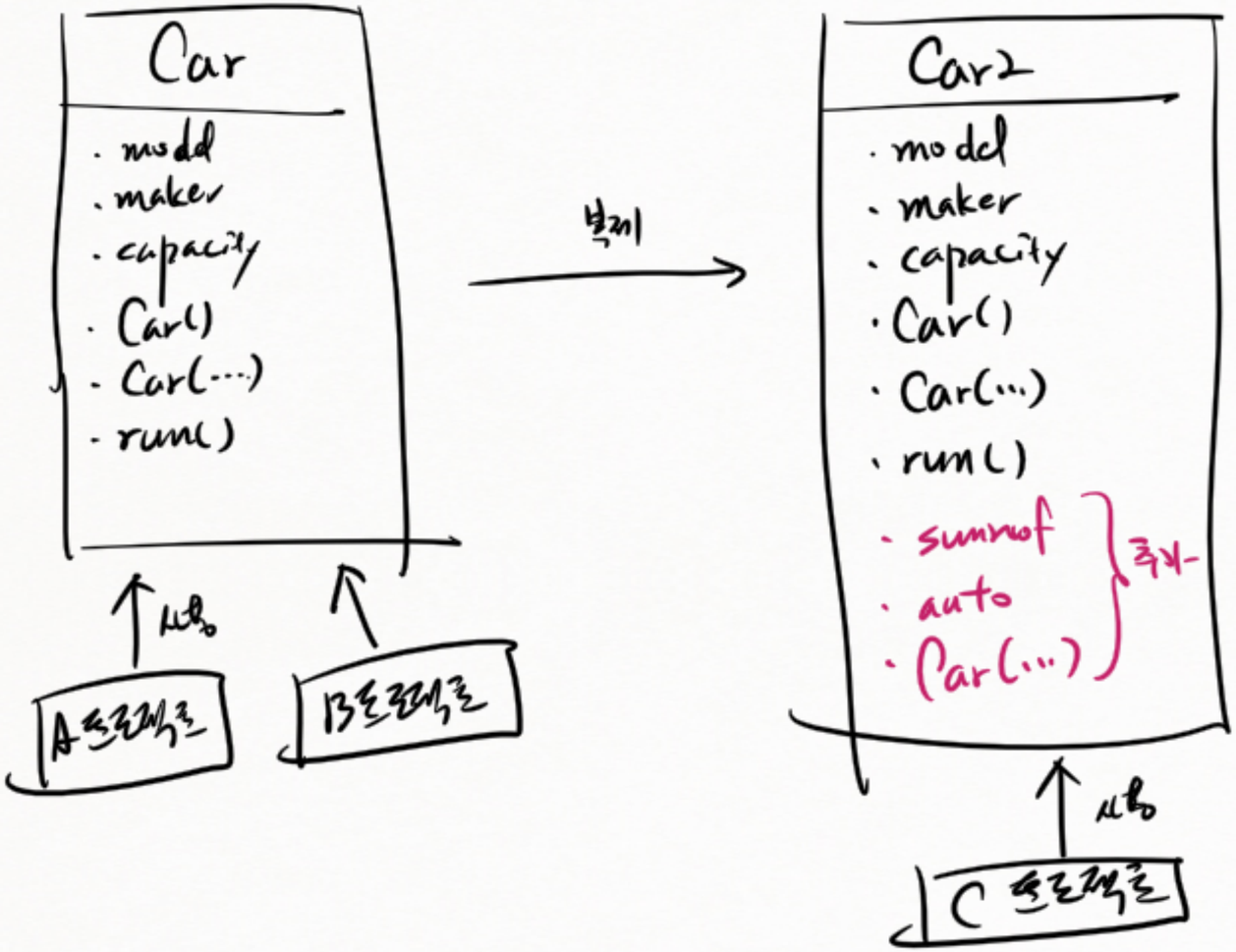


문제점

① 코드에 계속 기능을 덧붙이다 보면 누더기 코드가 된다

☆ ② 기존 코드를 사용하는 프로젝트에 영향을 미친다.
A와 B 프로젝트

* 기능을 확장하는 방법 — ② 기존 소스를 복제해서 기능 확장



특징

① 기존 코드를 손대지 않는다

↓

기존 프로젝트에 영향을 끼치지 않는다.

단점

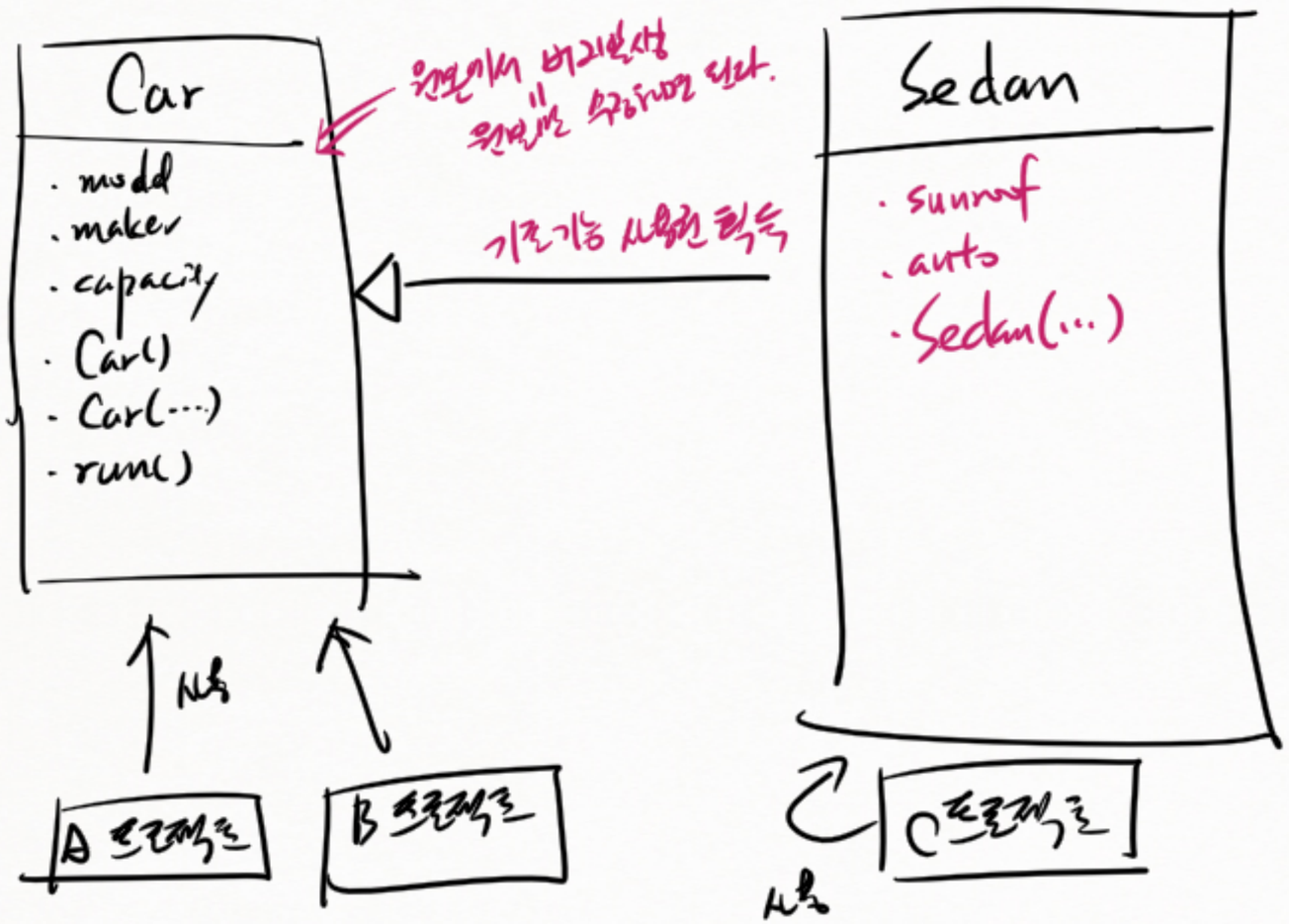
① 복제 → 중복 코드 생성

↓

(기능추가시 → 전체에 적용
버그수정시 → 〃)

"유지보수가 힘들다."

* 기능을 확장하는 방법 — ③ 상속을 이용



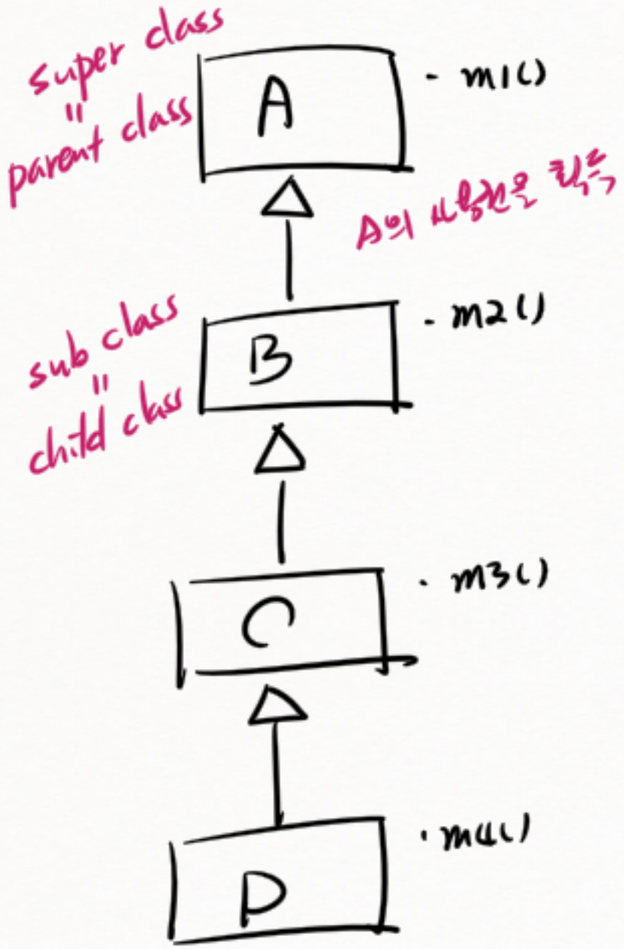
특징

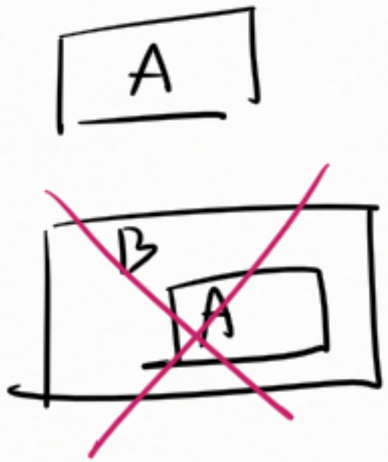
- 기존 코드를 손대지 않는다.
  ↓
  이전 프로젝트에 영향을 끼치지 않는다
- 기존 코드를 재사용 → 비용절감
  → 버그 확률 가능성 ↓
  ⇓
  코드 중복을 없앤다.
  ↳ 버그 수정이 용이

한계

- 여러 단계를 거치다 보면
  필요없는 기능도 강제로 상속받는경우가 있다

* 상속과 메서드 호출





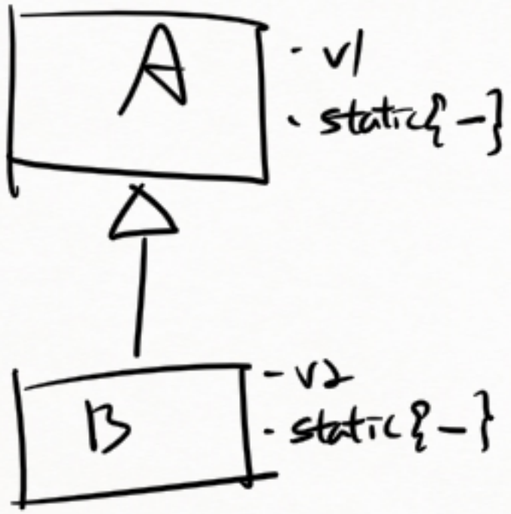
A의 근본은 그대로 포함하게 아니다-!

```
B obj = new B();
obj.m2(); //ok
obj.m1();
```

← B 클래스를 통해 A 클래스의 멤버를 사용할 수 있다

↑ A의 m1()을 호출

* 상속과 인스턴스 필드(변수)

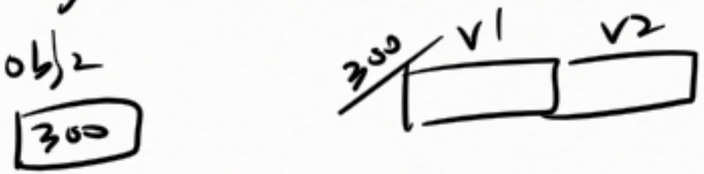


B obj = new B();

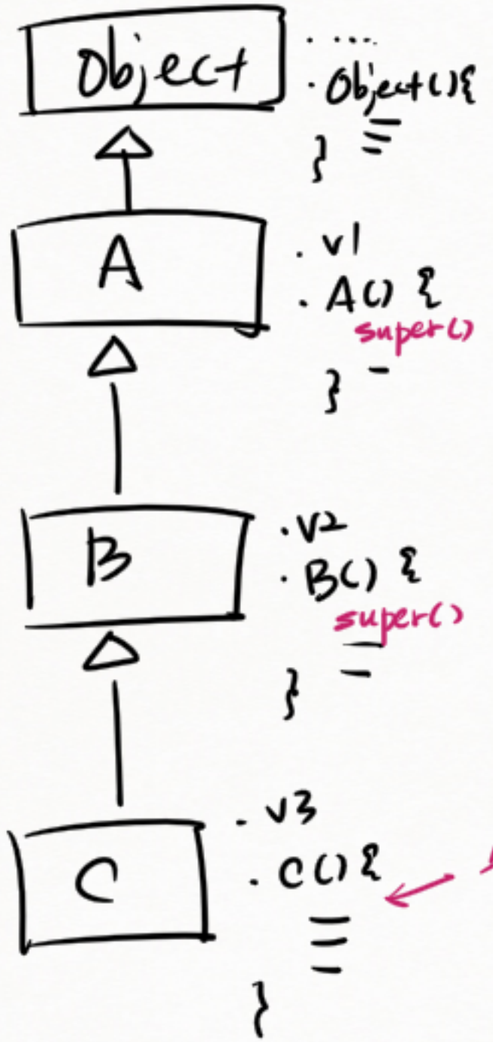
* 클래스 로딩
↓
슈퍼 클래스 부터 로딩

*인스턴스 필드 생성
↳ super 클래스의 인스턴스 변수부터 생성

B obj2 = new B();



* 생성자 호출 순서



생성자의 "첫 문장"은 반드시 수퍼클래스의 생성자 호출 문장이어야 한다
예) super()

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

⇐ 둘 다 동시에 호출할 순 없다

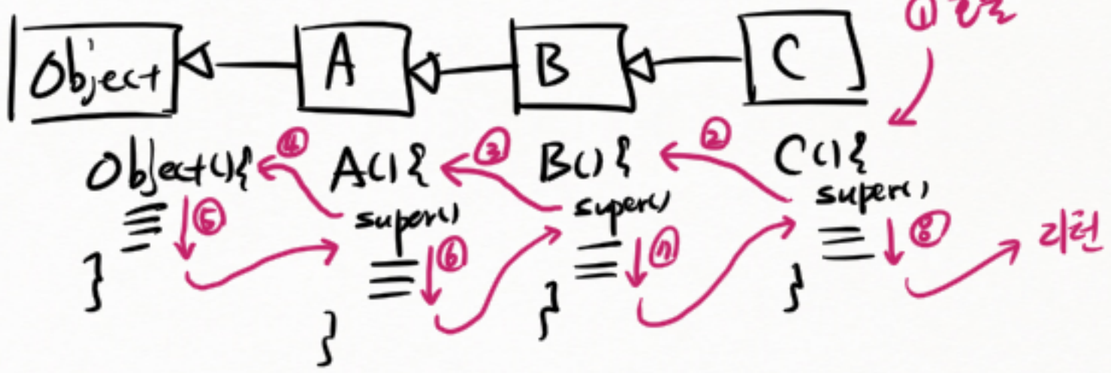
예) super() 또는 this()

~~this()~~ ~~super()~~

첫 문장이 아니다!

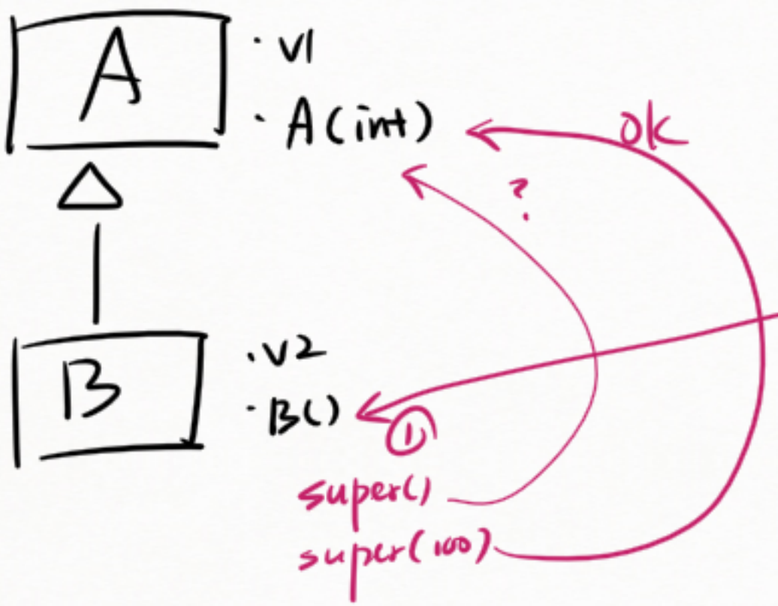
C obj = new C();

↳ C의 생성자 호출



수퍼클래스의 생성자 호출을 생략하면 컴파일러가 자동으로 추가한다.

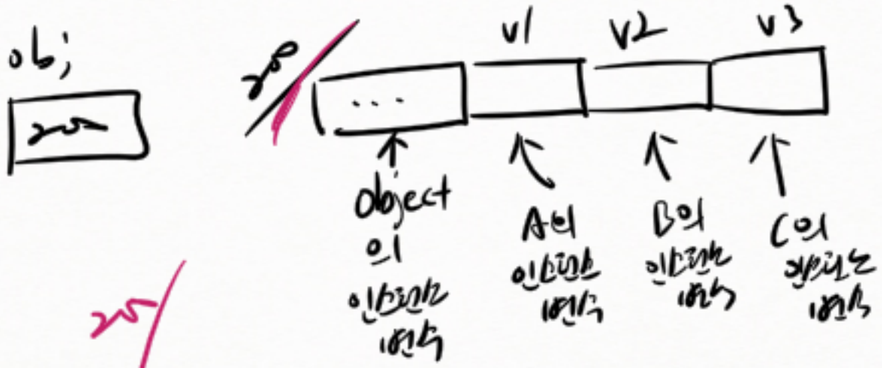
* 생성자 호출 2



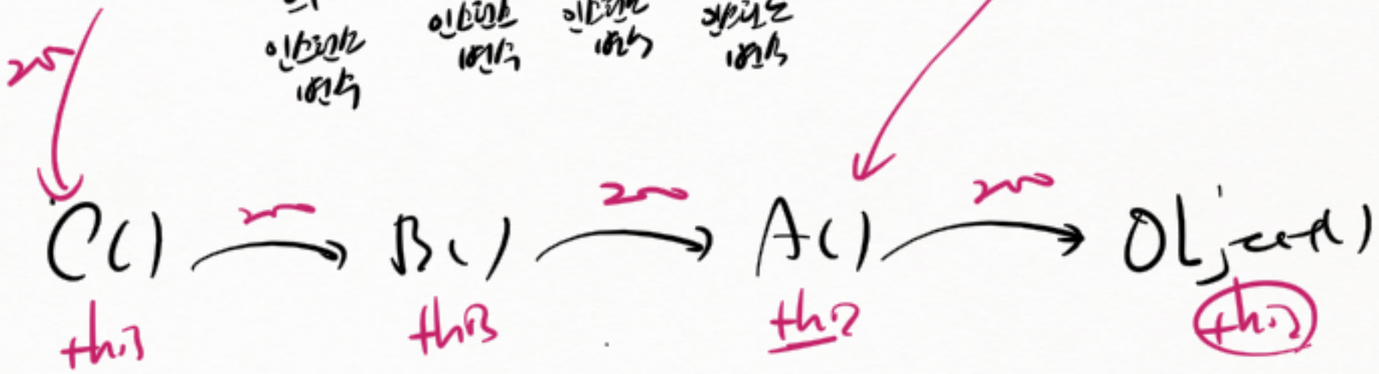

```
B obj = new B();
```

B의 기본 생성자 호출

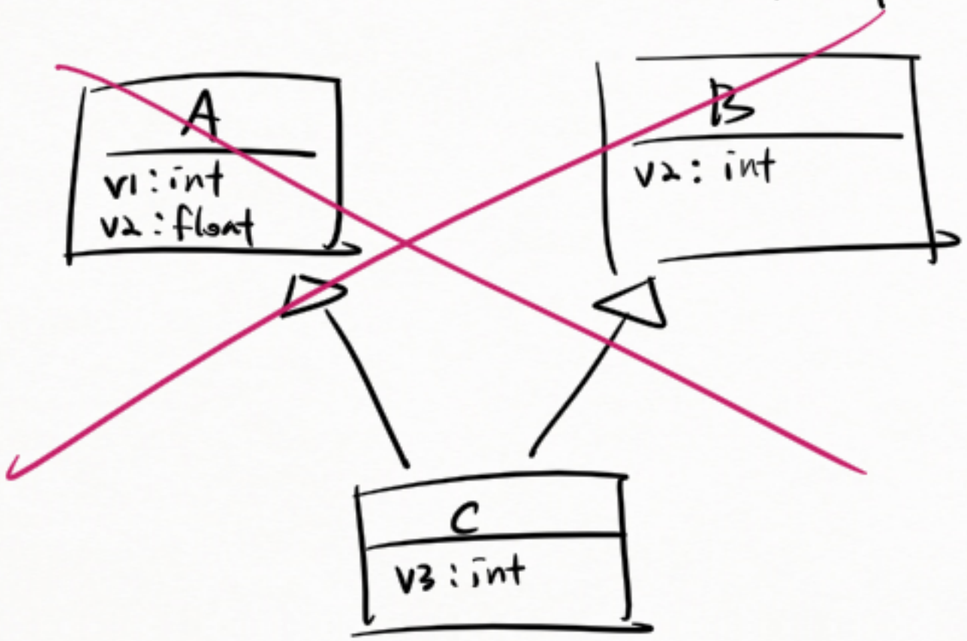
C obj = new C();



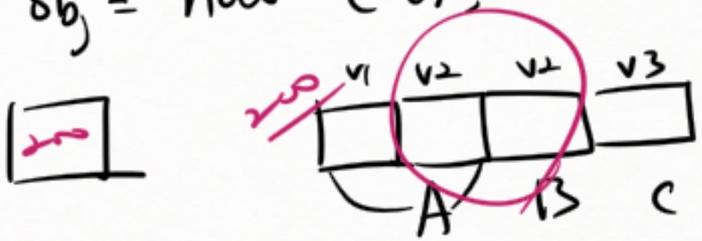
new A();



* 다중 상속 ⇒ 지원 안함!




```
C obj = new C();
```




```
obj.v2 = 0;
```

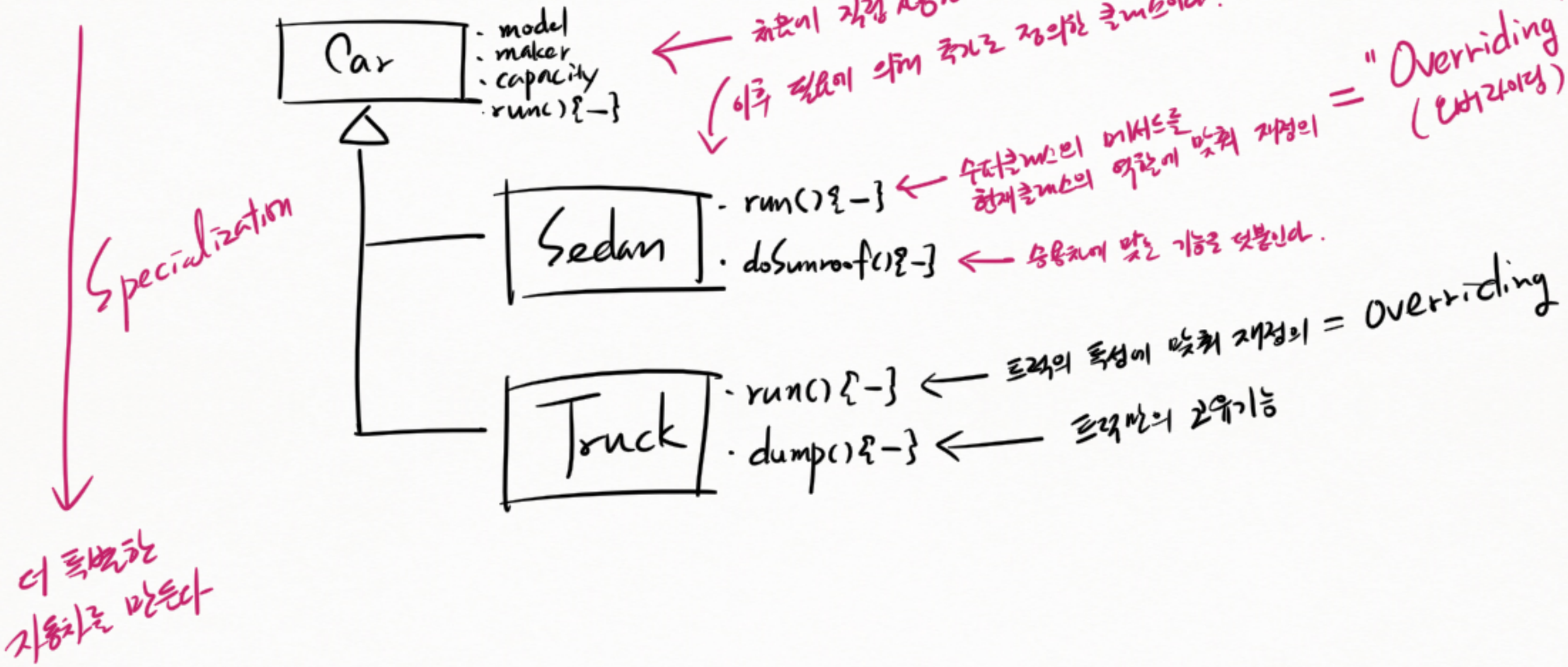
A의 변수인지
B의 변수인지
구분할 수 없다.

구분하려면
문법을 추가

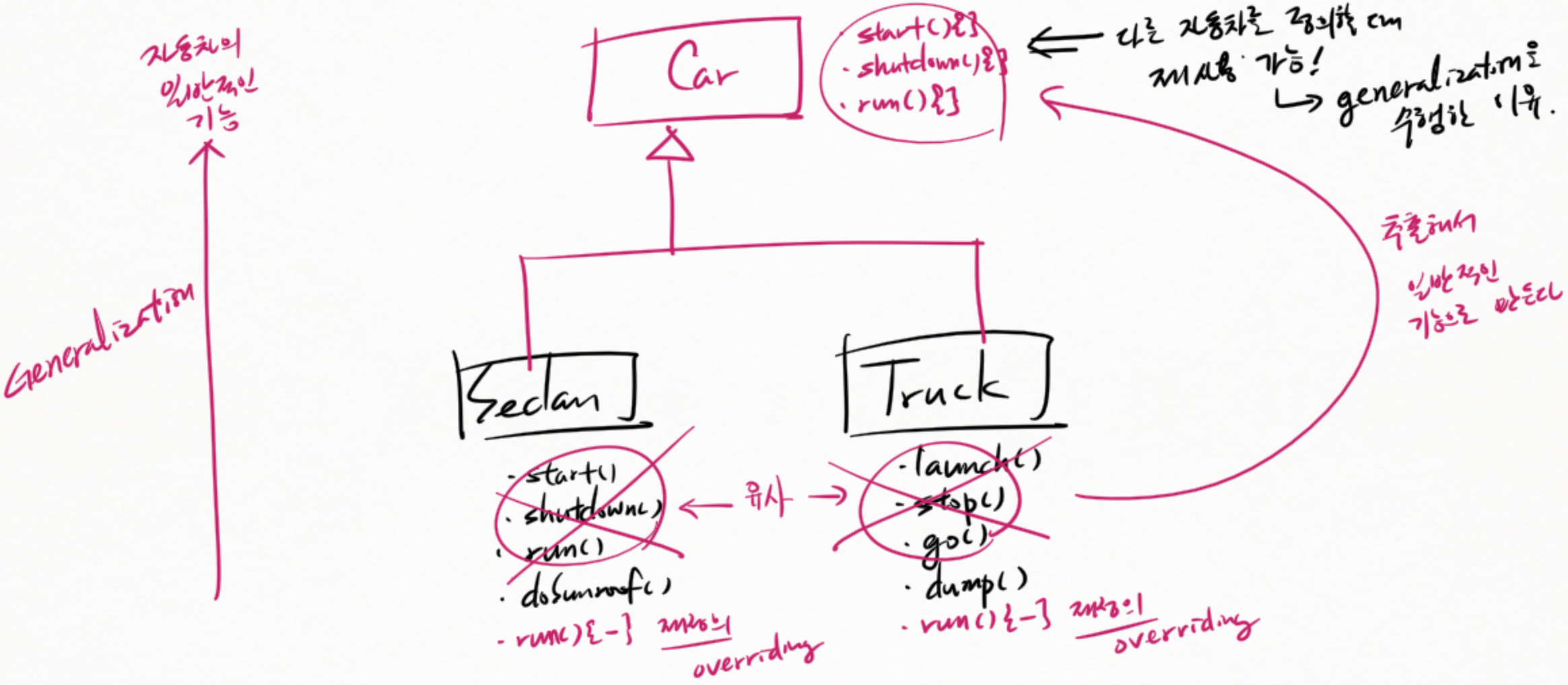
언어가 복잡
=
컴파일러 복잡

그래서
자바는
다중 상속을
허락하지 않는다

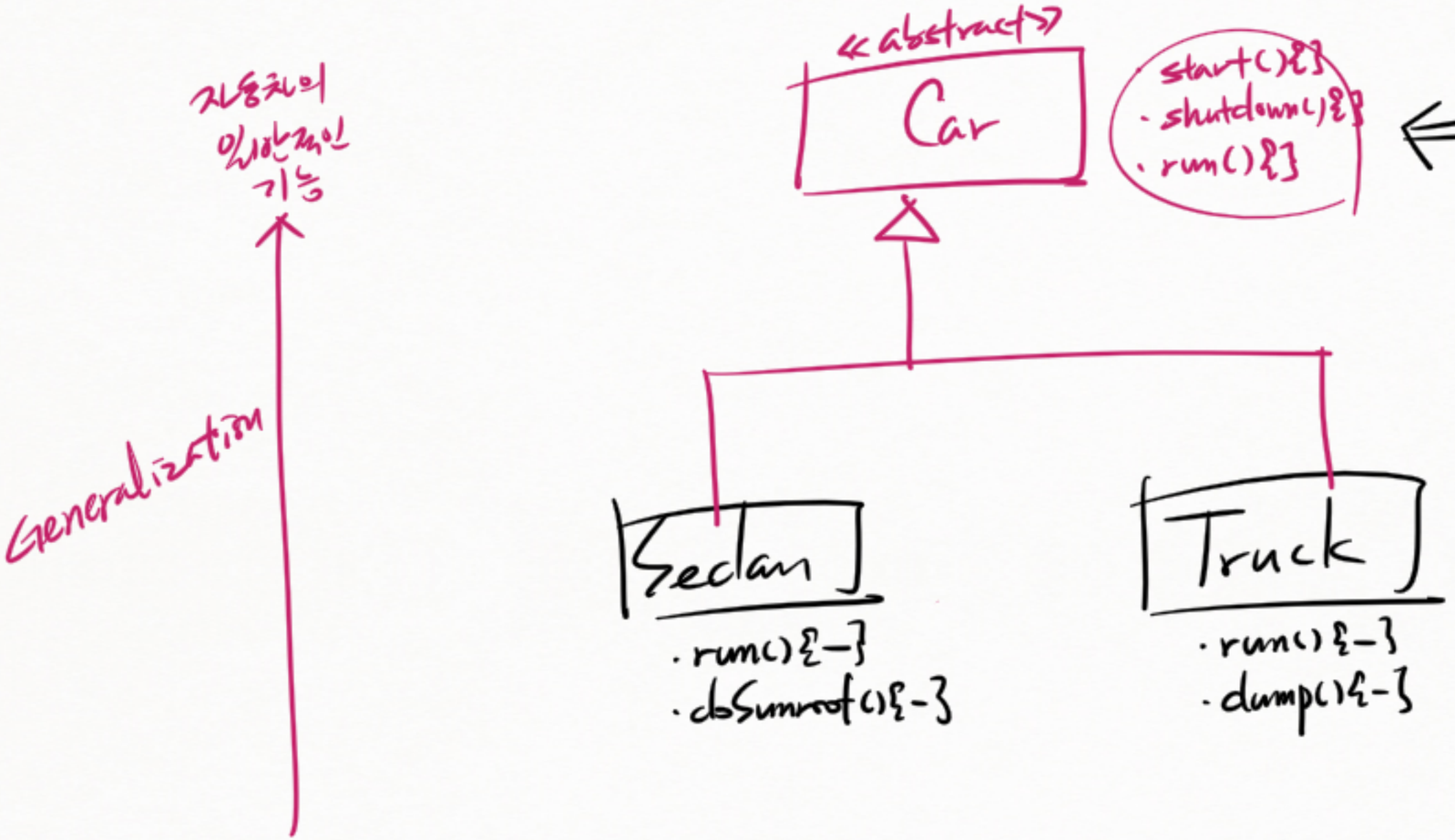
* 상속 : specialization (전문화)



* 상속: Generalization (일반화)



* 상속과 추상 클래스



소스 코드를 유지보수하기 쉽도록 공통 코드를 추출해서 만든 클래스

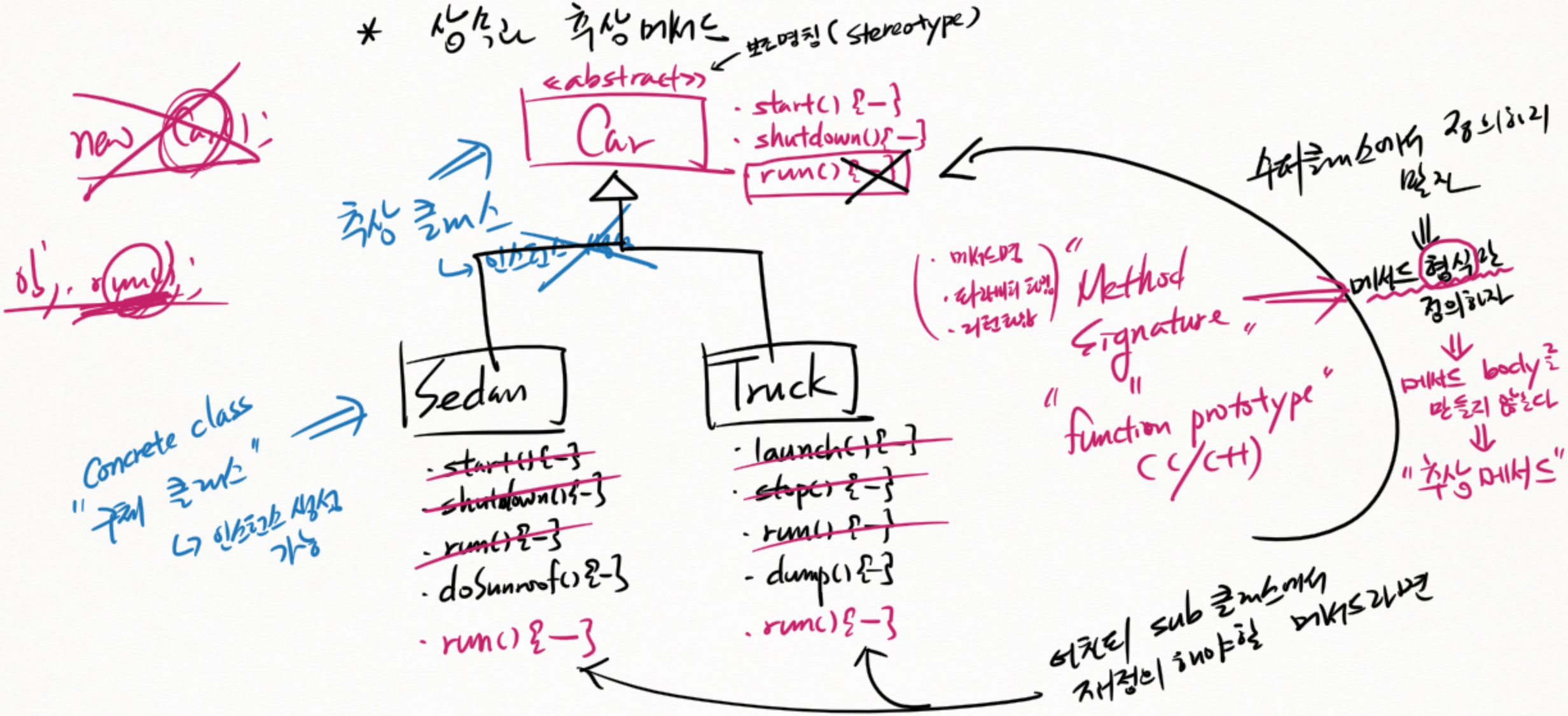
||

Car 클래스를 따로 쓰려고 만든 클래스가 아니다.

||

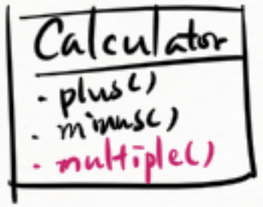
이런 경우 직접 사용할 수 없도록 "추상 클래스"로 선언한다.

* 상속과 추상 메서드

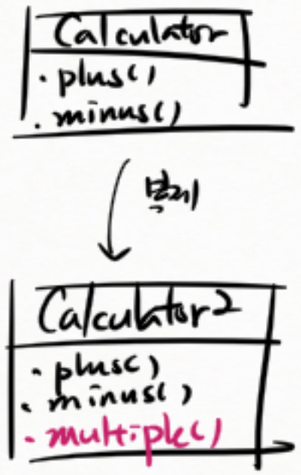

보조명칭 (stereotype)
<<abstract>>
Car
· start() {-}
· shutdown() {-}
· run() {-}
new Car();
이, run();
추상 클래스
↳ 인스턴스 생성 불가
Sedan
· start() {-}
· shutdown() {-}
· run() {-}
· doSunroof() {-}
· run() {-}
Truck
· launch() {-}
· stop() {-}
· run() {-}
· dump() {-}
· run() {-}
Concrete class
"구체 클래스"
↳ 인스턴스 생성 가능
· 메서드명
· 파라메터 리스트
· 리턴타입
"Method Signature"
"function prototype"
(C/C++)
서브클래스에서 정의하라 말고
메서드 형식만 정의하라
메서드 body를 만들지 않는다
"추상 메서드"
어차피 sub 클래스에서 재정의 해야할 메서드라면

* 기능 확장법 cheat sheet

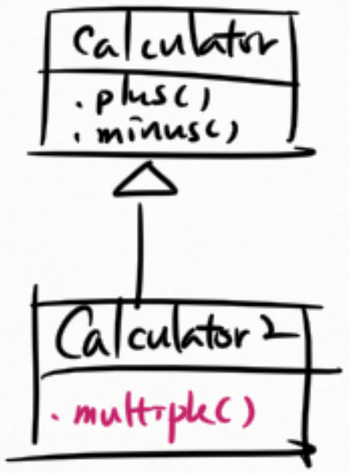
① 기존 코드 변경



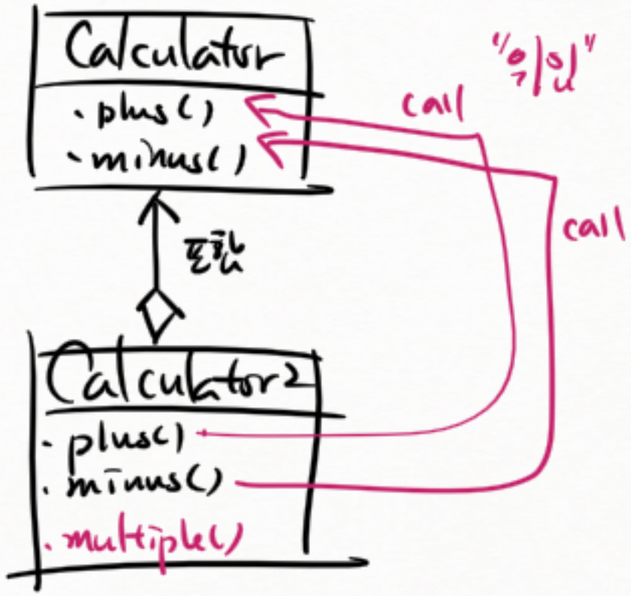
② 코드 복제후 기능 추가



③ 상속

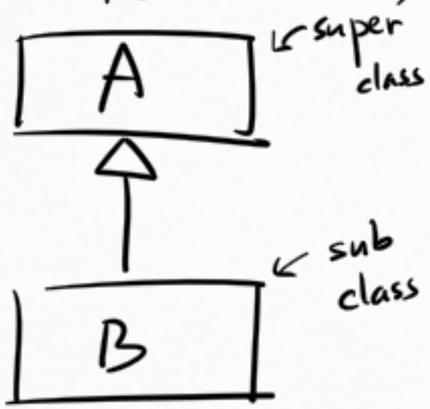


④ 위임



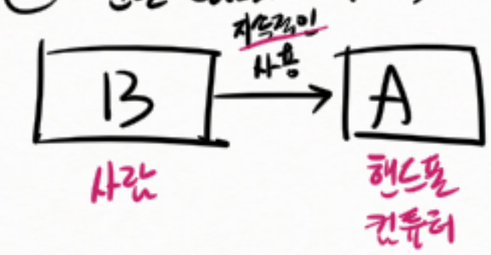
* 클래스 관계 cheat sheet

① 상속 (inheritance)




```
class B extends A {
    ≡
}
```

② 연관 (association)




```
class B {
    A obj;
    ≡
}
```

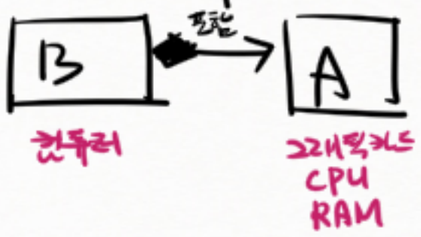
③ 집합 (aggregation)

```
class B {
    A obj;
    ≡
}
```

컨트롤러 ≠ 키보드 마우스 모니터

Lifecycle

④ 합성 (composition)

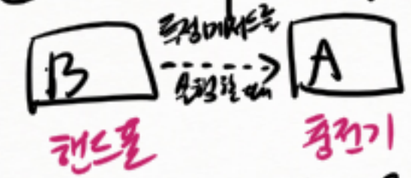



```
class B {
    A obj;
    ≡
}
```

컨트롤러 = 그래픽카드 RAM cpu

Lifecycle

⑤ 의존 (dependency)

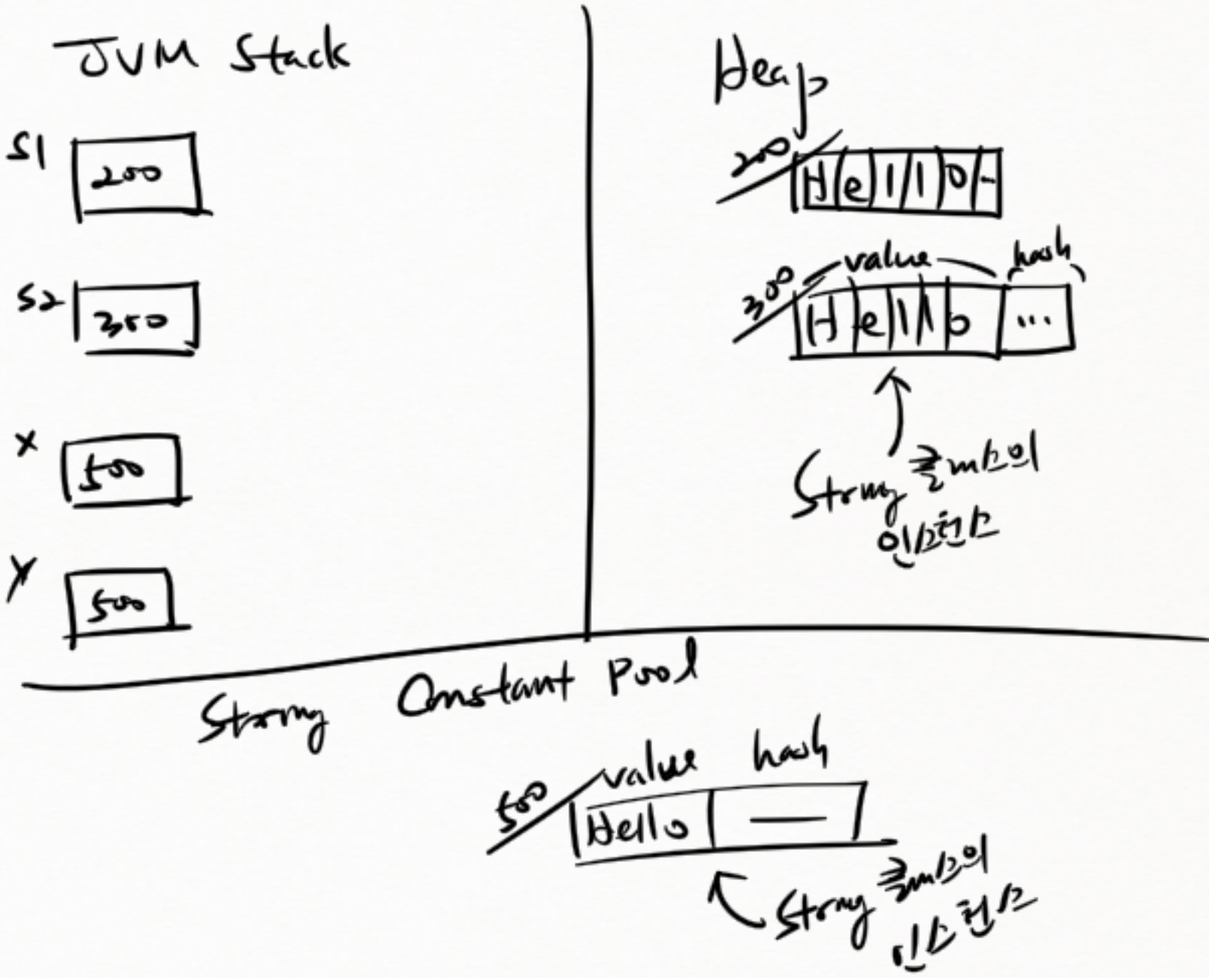



```
class B {
    void m(A obj){
        ≡
    }
}
```

* String 클래스 사용예

```
String s1;
s1 = new String("Hello");
String s2 = new String("Hello");

String x = "Hello";
String y = "Hello";
```

String 상수풀(pool)에 String 인스턴스를 생성 "중복생성하지 않는다"

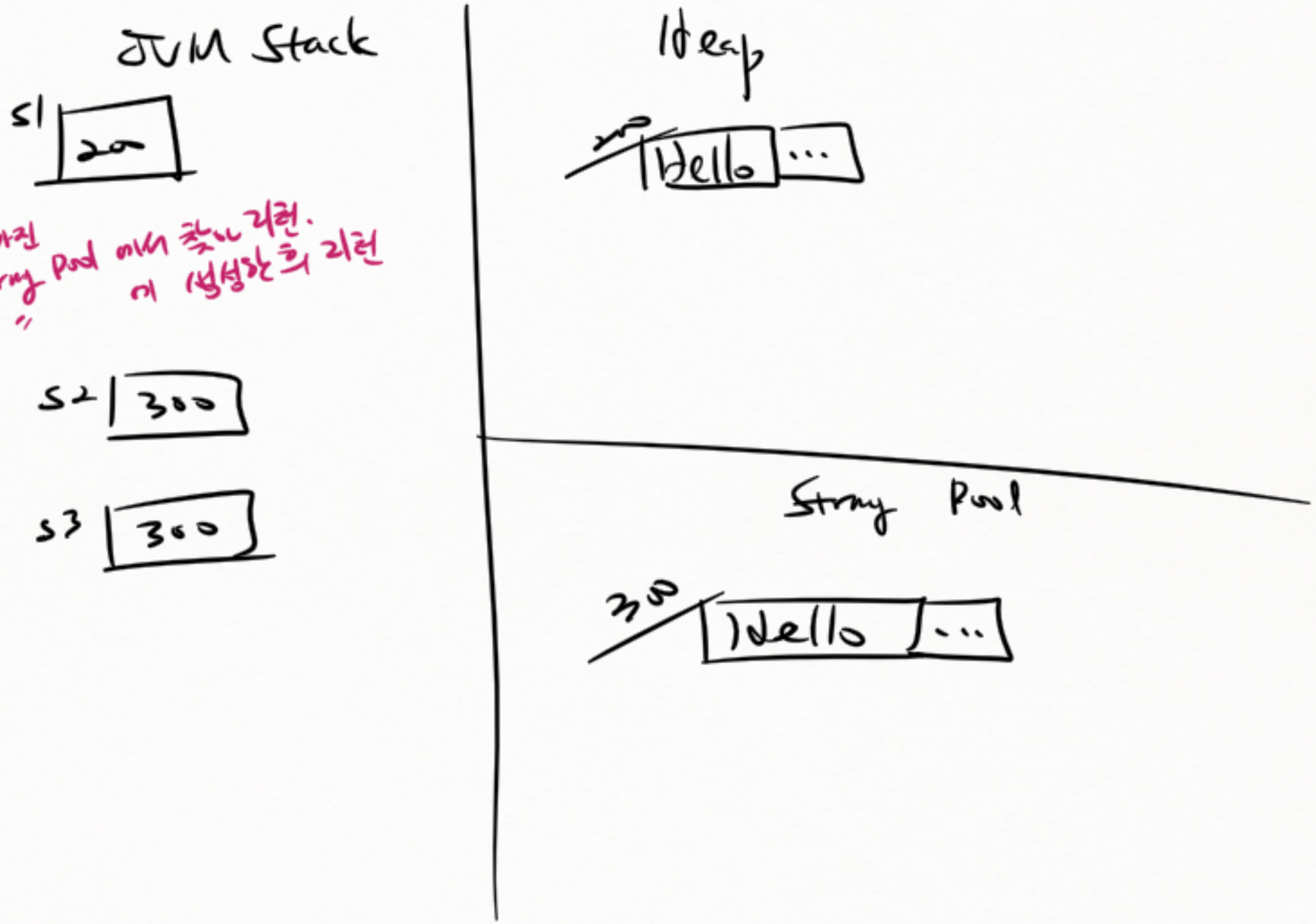


* String.intern()

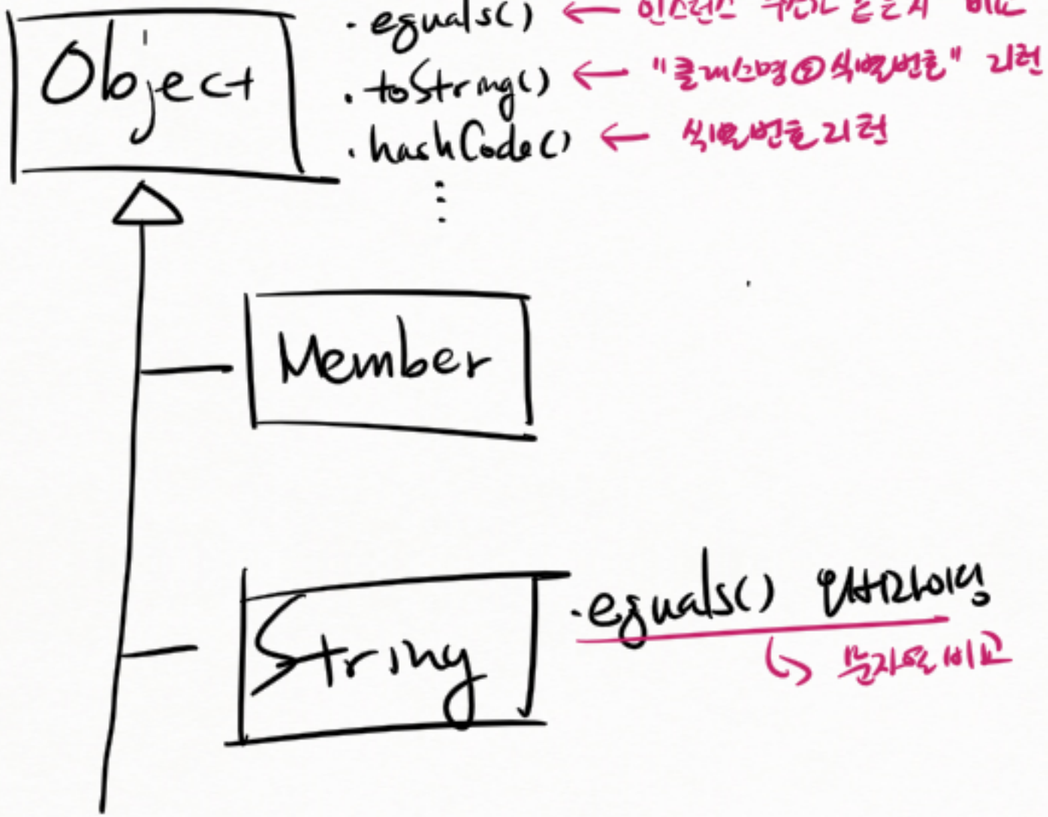
```
String s1 = new String("Hello")
String s2 = s1.intern()
String s3 = "Hello"
```

s1과 같은 문자열을 가진 String 객체를 String Pool에서 찾아 리턴. 만약 없으면, 이 생성한 후 리턴

360



* Object.equals()



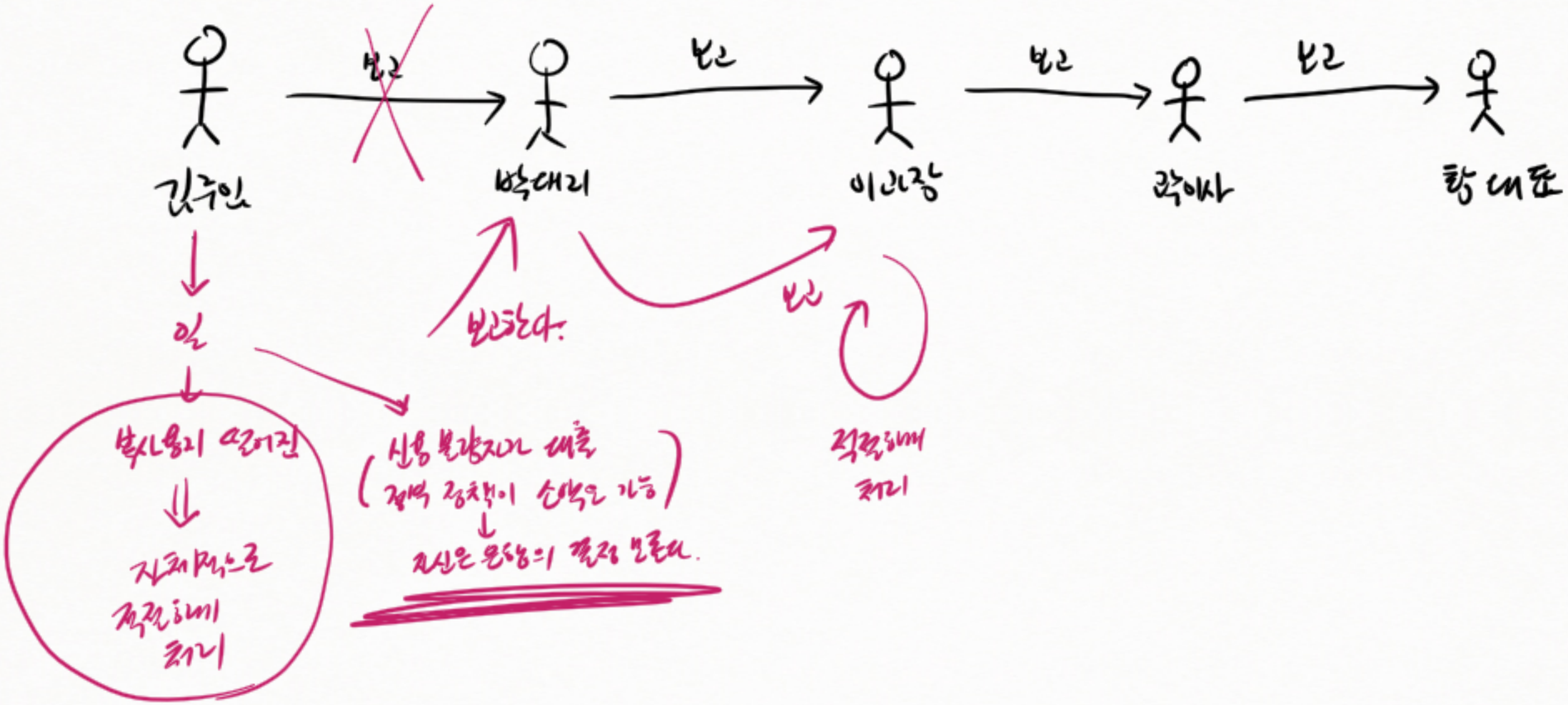

```
Member m = new Member();
m.toString();  ← Object
m.equals();    ← Object
m.hashCode();  ← Object
⋮
String s1 = new String("Hello");
String s2 = new String("Hello");
```

s1 == s2 ⇒ false

s1.equals(s2) ⇒ true

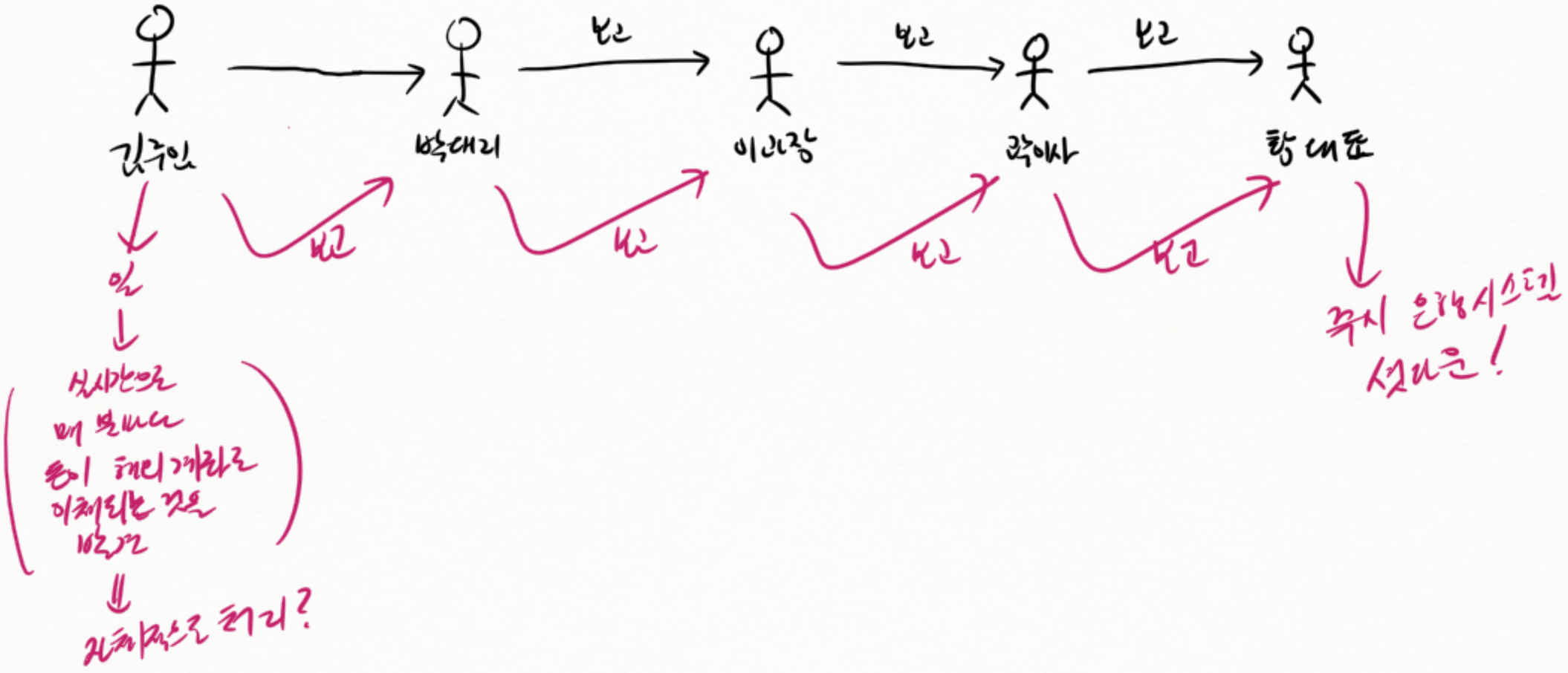
* 예외 처리 : 발생 → 보고 → 처리

① 안은 예외 상황



* 예외 처리 : 발생 → 보고 → 처리

② 치명적인 상황

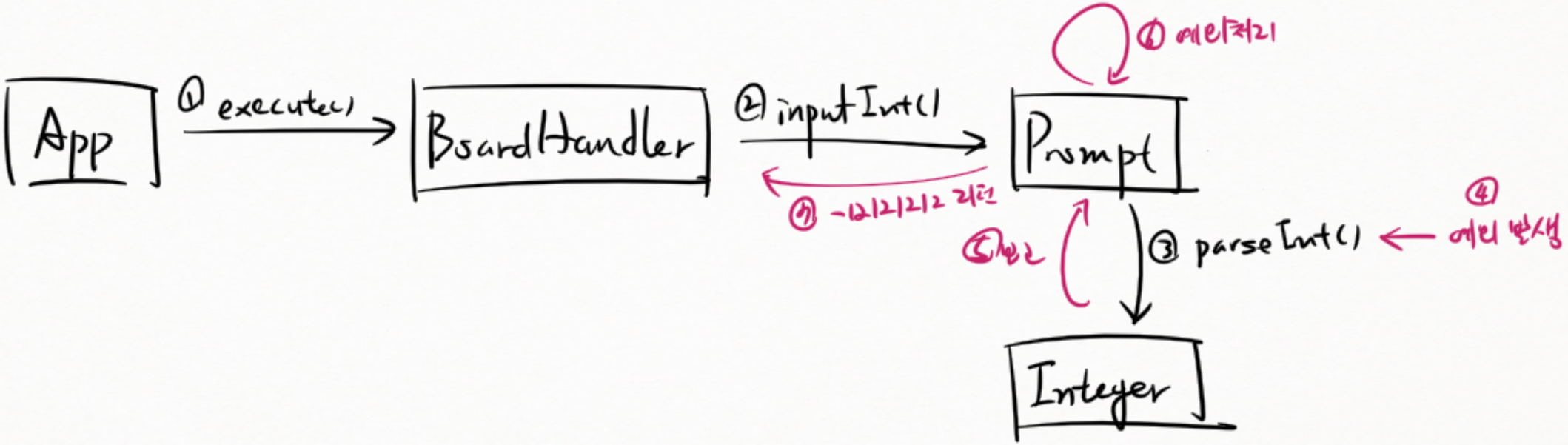


* 예외 처리 문법

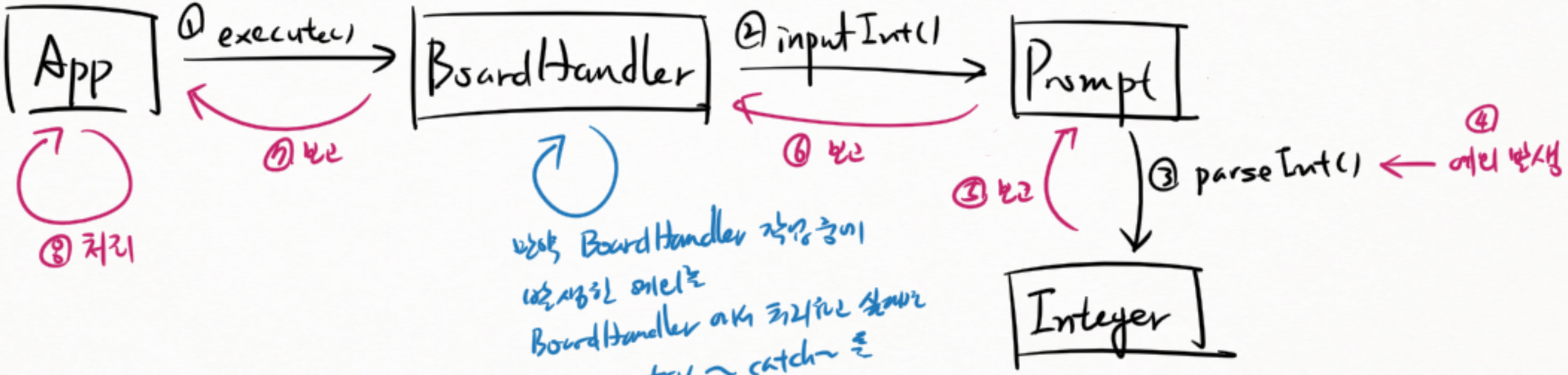
```
try {
    예외가 발생할 수 있는 코드
} catch (예외정보를 받을 파라미터) {
    예외처리 코드
}
```

→ java.lang.Throwable

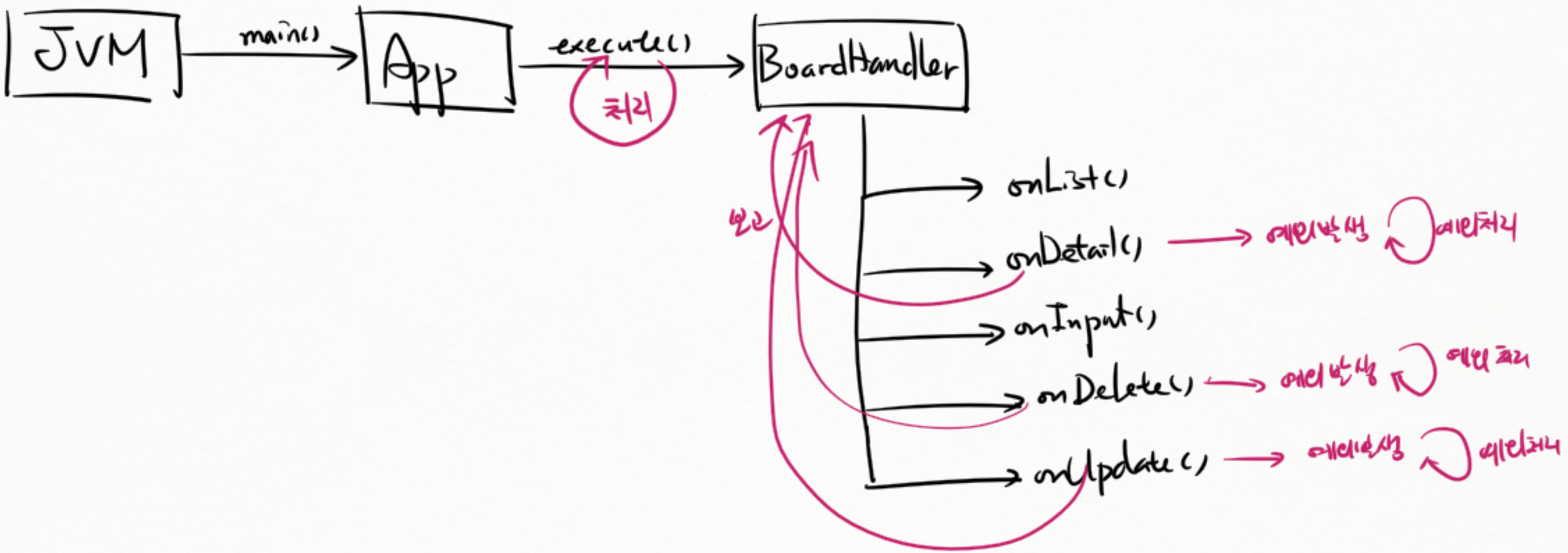
* 016. 예외 처리 1


App
① execute()
BoardHandler
② inputInt()
Prompt
⑥ 예외처리
⑦ -121212 리턴
⑤ 반복
③ parseInt()
④ 예외 발생
Integer

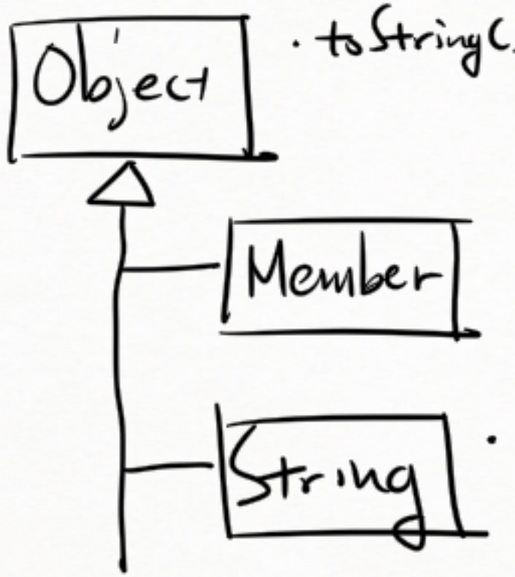
* 0117. 에러 처리 2



만약 BoardHandler 작업 중에 발생한 에러를 BoardHandler 에서 처리하고 싶다면 여기에 try ~ catch ~ 를 쓰야 한다.


JVM
main()
App
execute()
처리
BoardHandler
onList()
onDetail()
onInput()
onDelete()
onUpdate()
호출
예외발생
예외처리
예외발생
예외처리
예외발생
예외처리

* 오버라이딩 메서드 호출



· toString() ⇒ "클래스명@해시값"

· toString() ⇒ "저장된 문자열"

```
Object obj;
obj = new Object();
obj.toString();
```
← Object

```
Member m = new Member();
m.toString();
```
← Object

```
String s = new String("Hello");
s.toString();
```
← String

```
Object x = new String("Hello");
x.charAt(0);
x.toString();
```

"컴파일 오류" → x.charAt(0); ← 컴파일 오류!

컴파일러는 변수의 타입이 클래스나 메서드를 호출 가능. x 변수에 어떤 객체가 저장되었는지 관심 없다.

x.toString(); ← 컴파일 OK!

↳ JVM은 레퍼런스가 실제 가리키는 클래스에서의 메서드를 찾아올라간다. ⇒ 결론! String꺼!

016. 예외 상황 알리기

throw 예외정보를 담은 객체;

↑

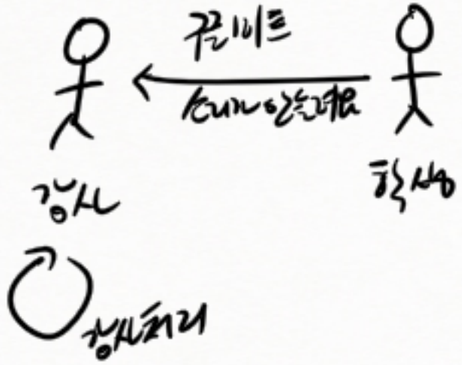
java.lang.Throwable 객체

서브 클래스 가능!

* 예외 처리 방법

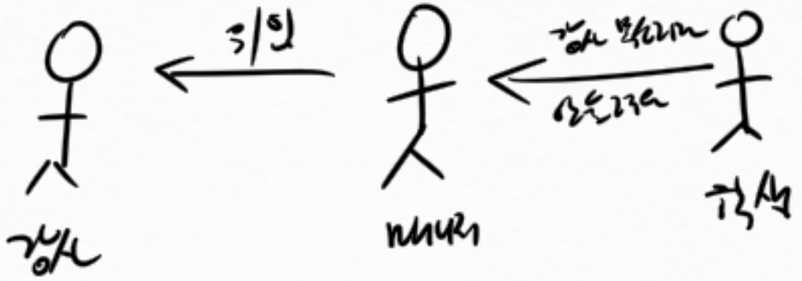
① 직접 처리

```
void m() {
    try {
        예외를 발생시키는 코드
    } catch (Throwable ex) {
        예외처리 코드
    }
}
```

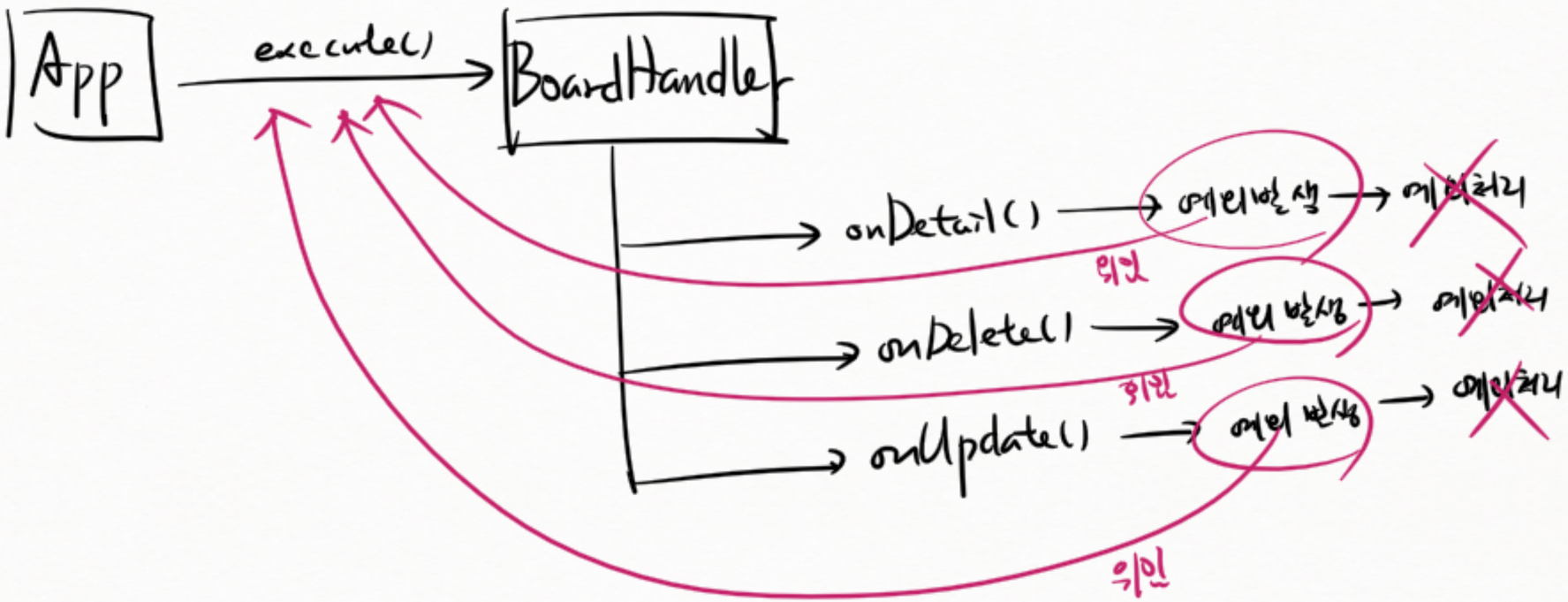


② 위임

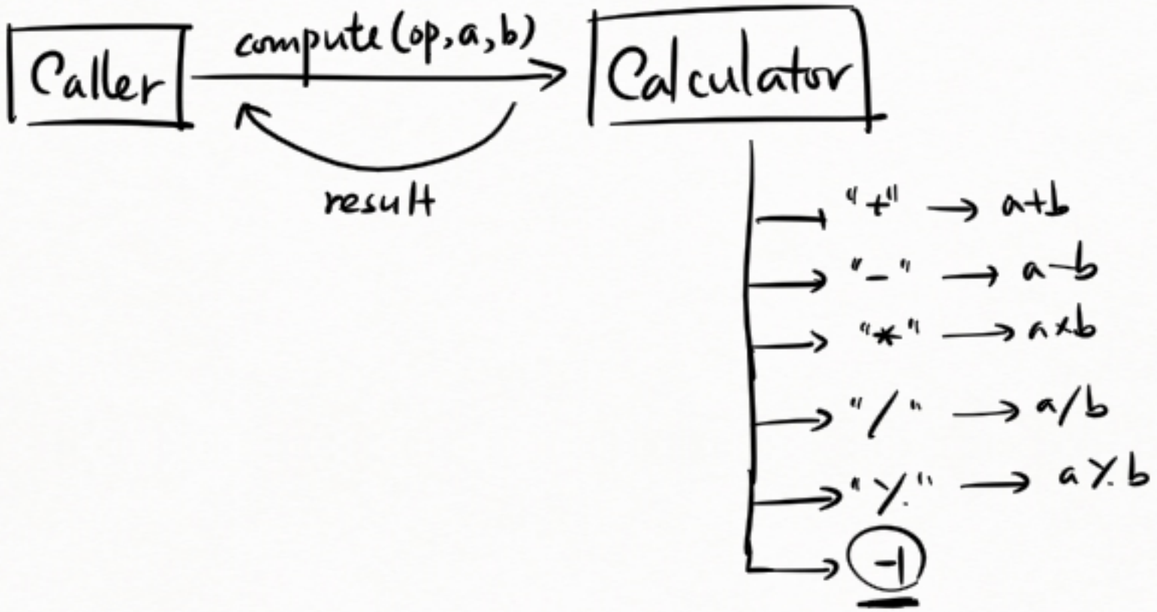
```
void m() throws 예외타입 {
    예외를 발생시키는 코드
}
```



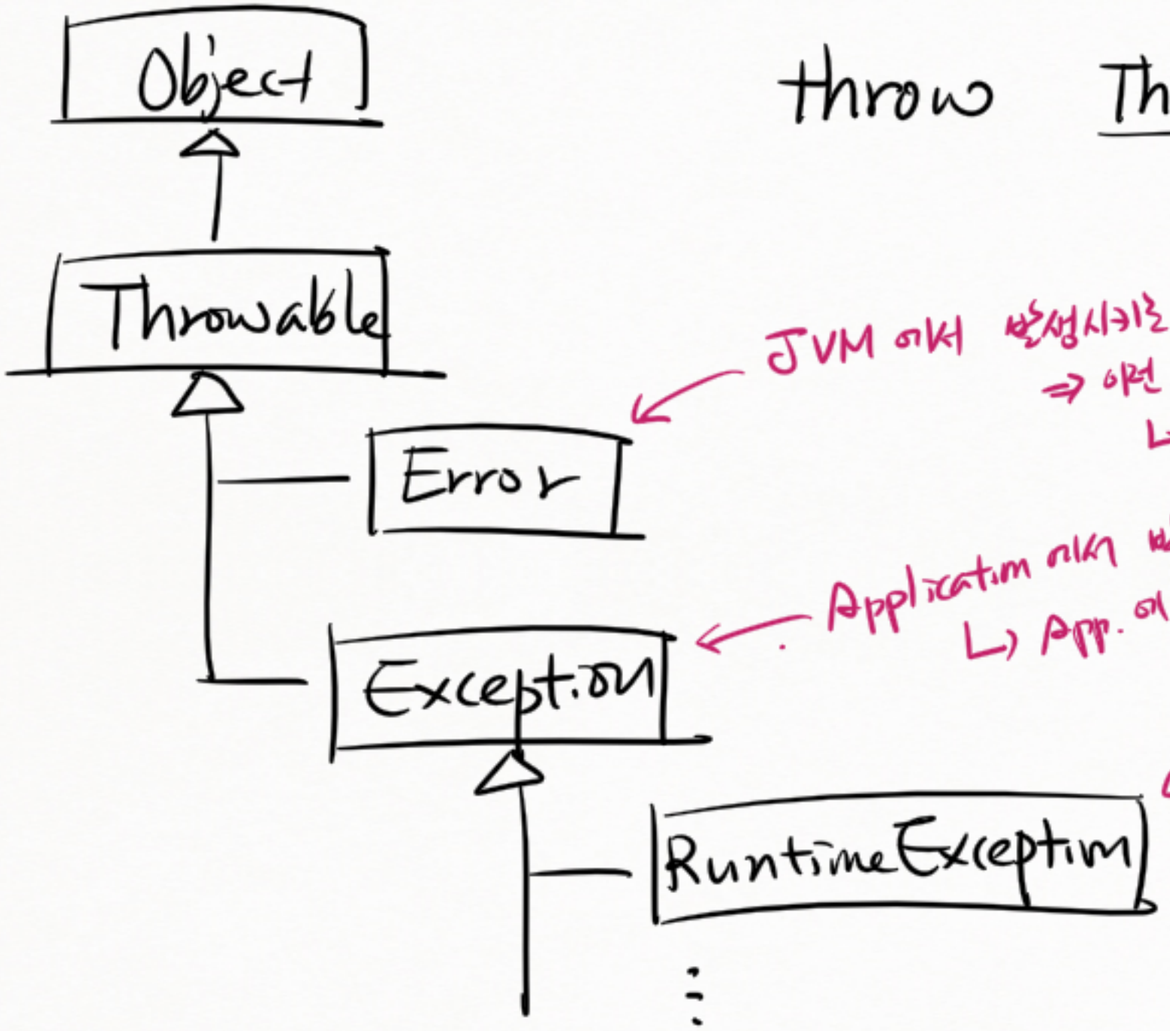
예외를 처리하지 않고, 예외 통지를 함으로써 m()을 호출한 쪽에서 처리하도록 위임한다.


App
execute()
BoardHandle
onDetail()
예외발생
예외처리
위임
onDelete()
예외 발생
예외처리
위임
onUpdate()
예외 발생
예외처리
위임

* 예외 상황을 리턴 값으로 알리기



* 에러 던지기



throw Throwable 객체 ;

↳ 서브 클래스의 객체도 던질 수 있다.

JVM 에서 발생시키는 에러 → 메모리 부족, 스레드 종료 등

⇒ 이런 에러는 Application에서 제어할 수 없다

↳ 적절하게 기록을 남긴 후에 App. 종료해야 한다.

Application 에서 발생시키는 에러

↳ App. 에 제어 가능

↳ 적절하게 조치를 취한 후 계속 실행되게 할 수 있다.

에러 또는 외의 메서드에서 사용 가능

↳ 프로그래밍의 편의성

↳ 에러 처리를 요구하지 않는다.

* 예외 위임

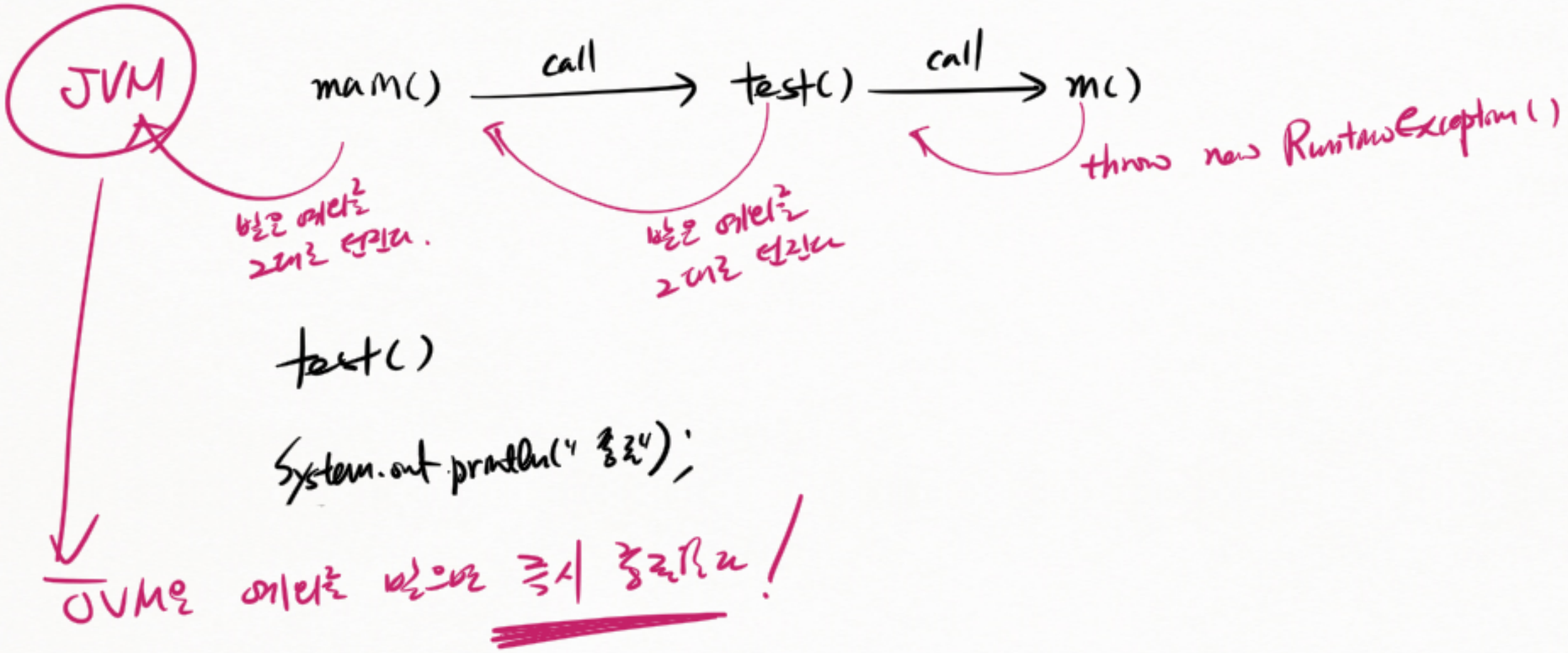
main() —call→ test() —call→ m()

throw new RuntimeException()

받은 예외를 그대로 던진다

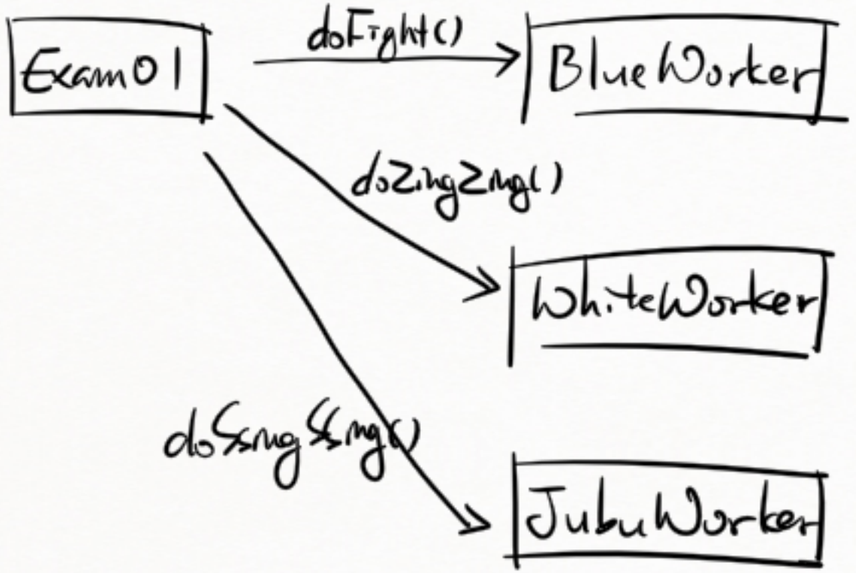
```
try {
  test();
}catch() {
  ≡ 여기서 받아서 처리
}
System.out.println("종료");
```

* 예외 처리


JVM
main()
call
test()
call
m()
throw new RuntimeException()
받은 예외를 그대로 던진다.
받은 예외를 그대로 던진다
test()
System.out.println("종료");
JVM은 예외를 받으면 즉시 종료한다!

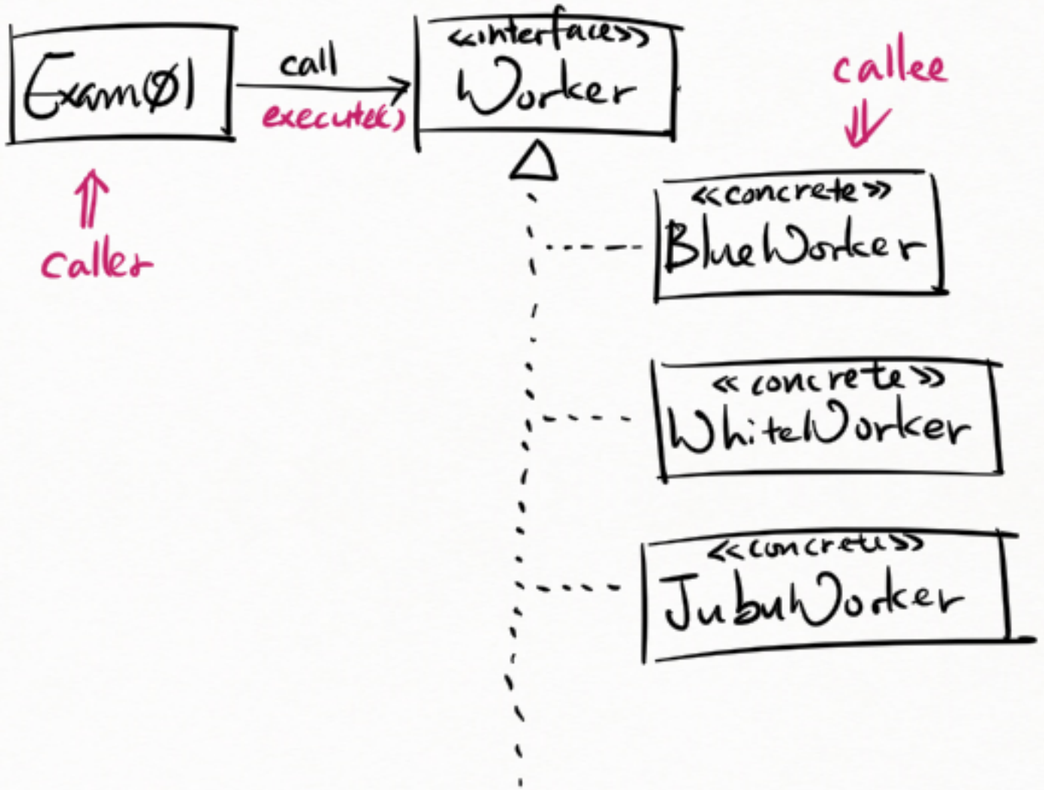
* 인터페이스

① 사용 전

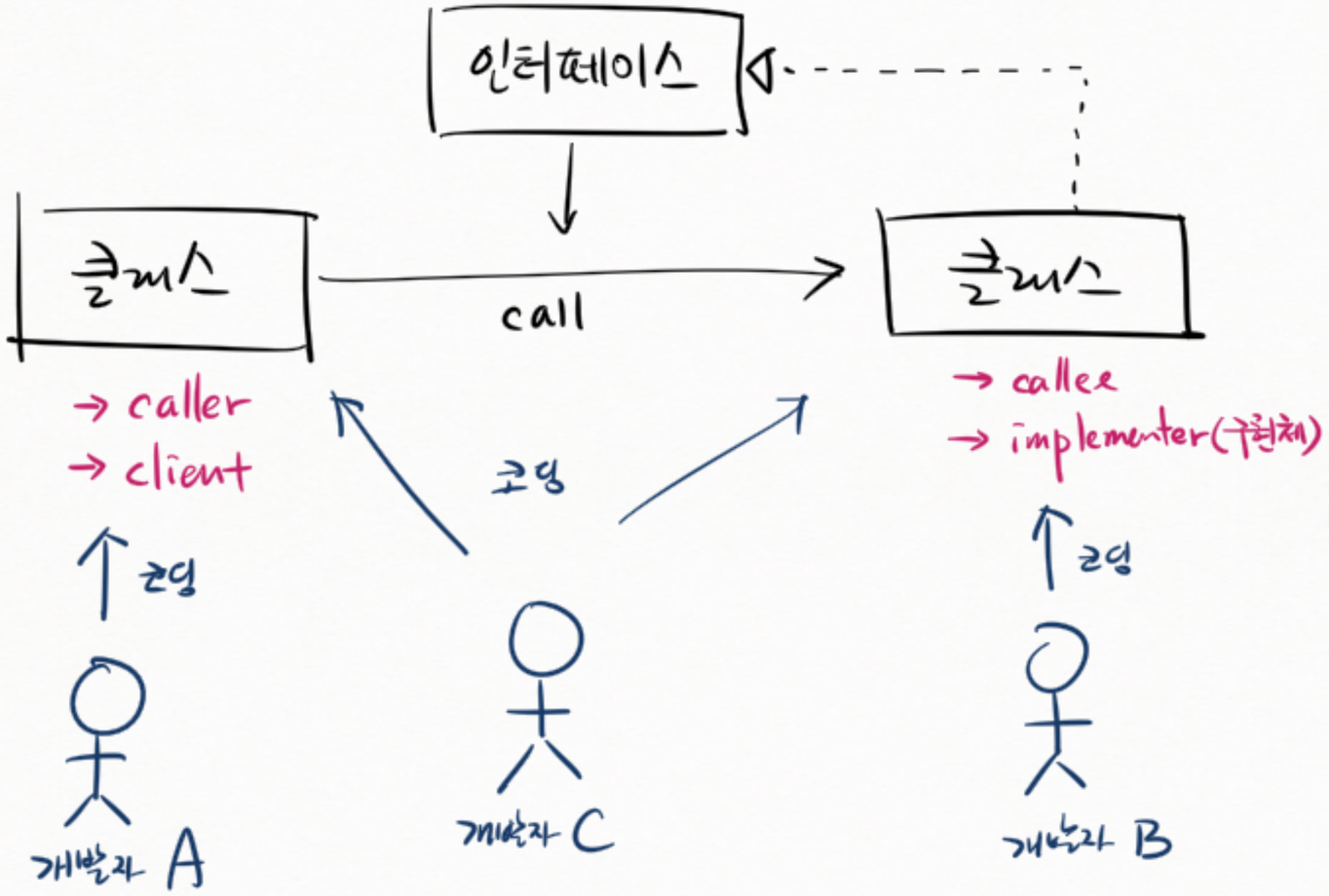


② 사용 후

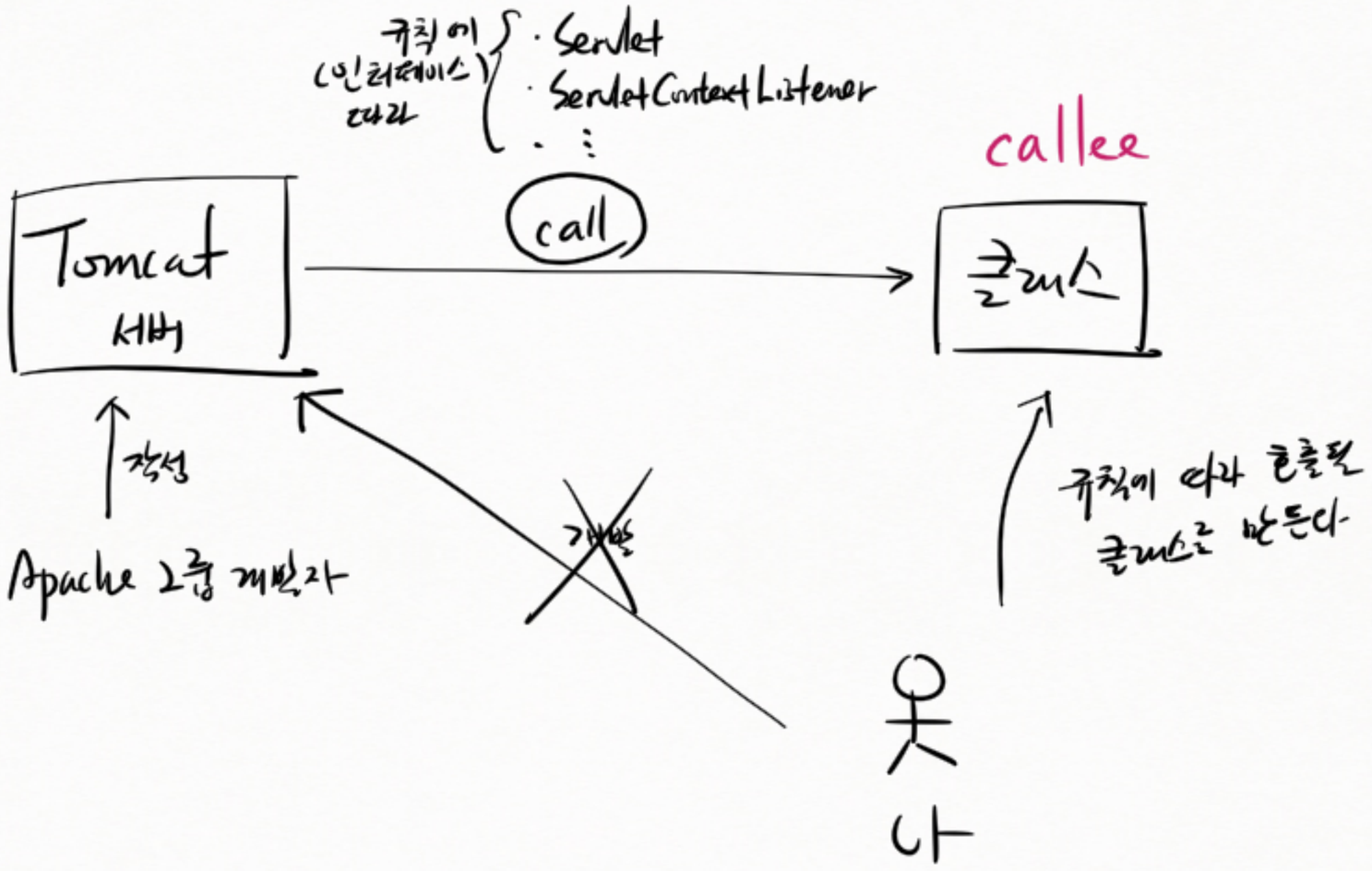
Worker의 사용 규칙 = 객체 사용 규칙
= 메서드 호출 규칙



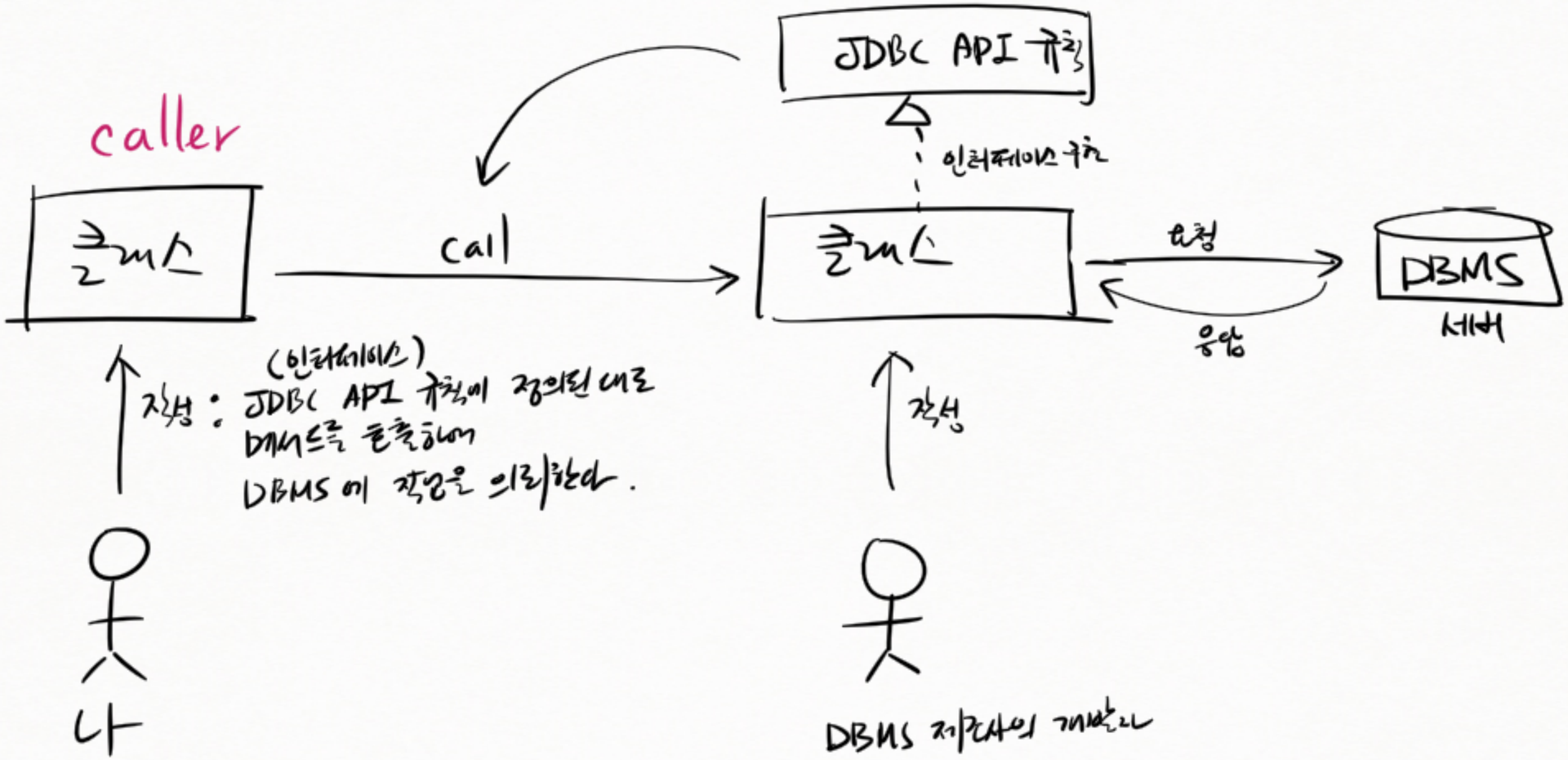
* 인터페이스와 코드 작성


인터페이스
클래스
call
클래스
→ caller
→ client
→ callee
→ implementer(구현체)
코딩
코딩
코딩
개발자 A
개발자 C
개발자 B

* 인터페이스 와 웹 프로그래밍



* 인터페이스와 JDBC 프로그래밍



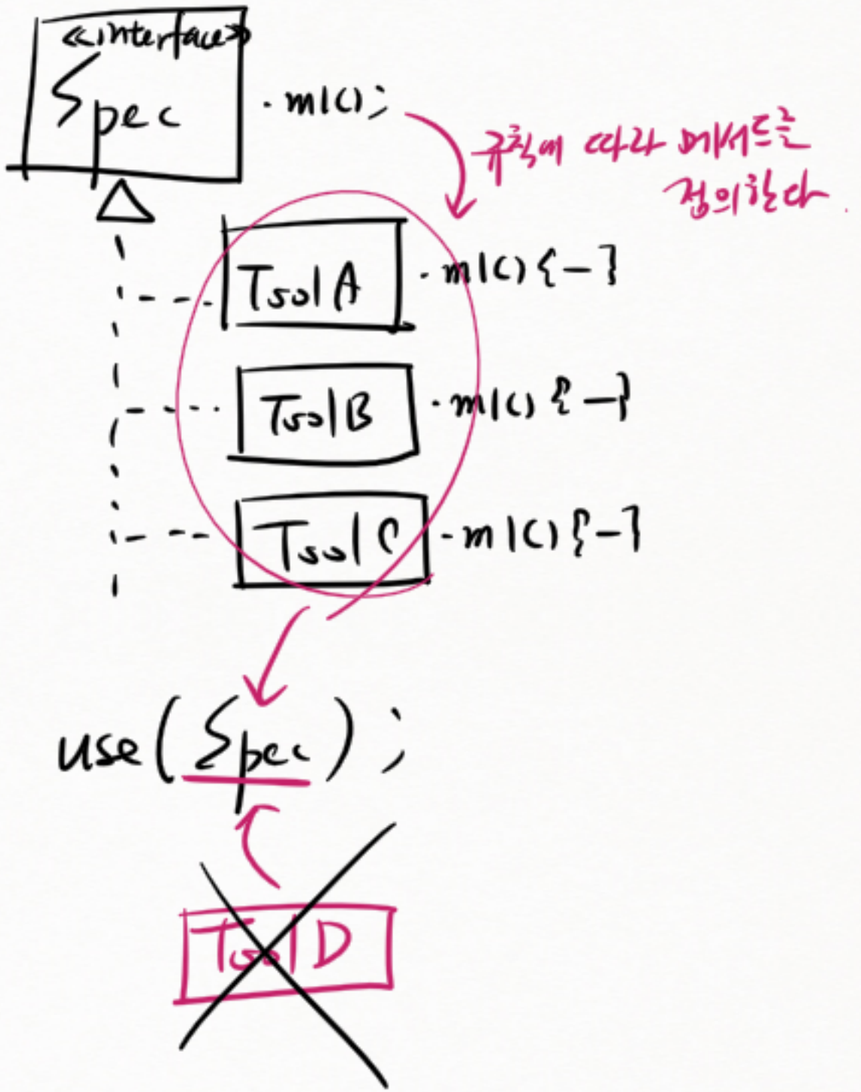
* 메서드 파라미터와 인터페이스

Exam01

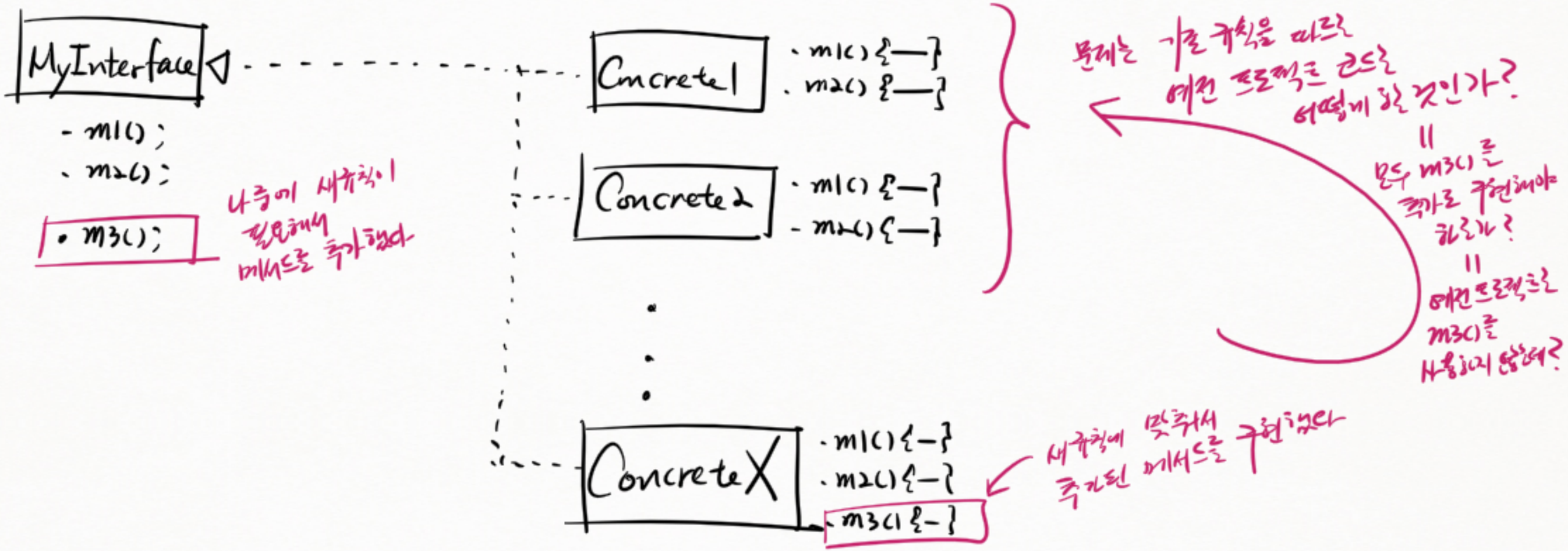
- main(){-}
- use(ToolA){-}
- use(ToolB){-}

use(ToolA 객체);

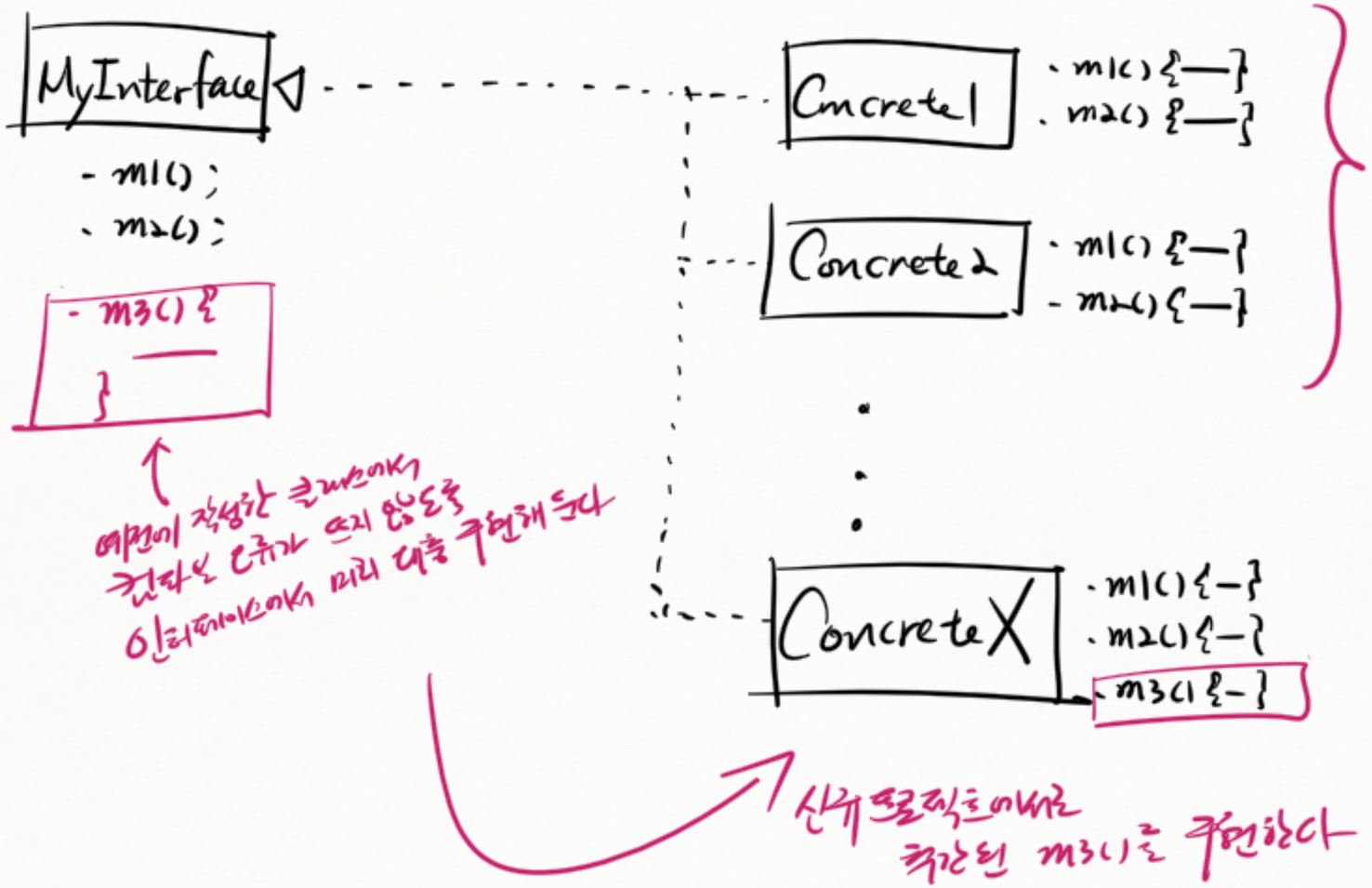
use(ToolB 객체);



* default 메서드 등장 전



* default 메서드 등장 후



예전에 작성한 클래스에서 컴파일 에러가 뜨지 않도록 인터페이스에서 미리 디폴트 구현해 둔다

신규 프로젝트에서는 추가된 m3()를 구현한다

이전 프로젝트는 소스를 변경할 필요가 없다.

⇓

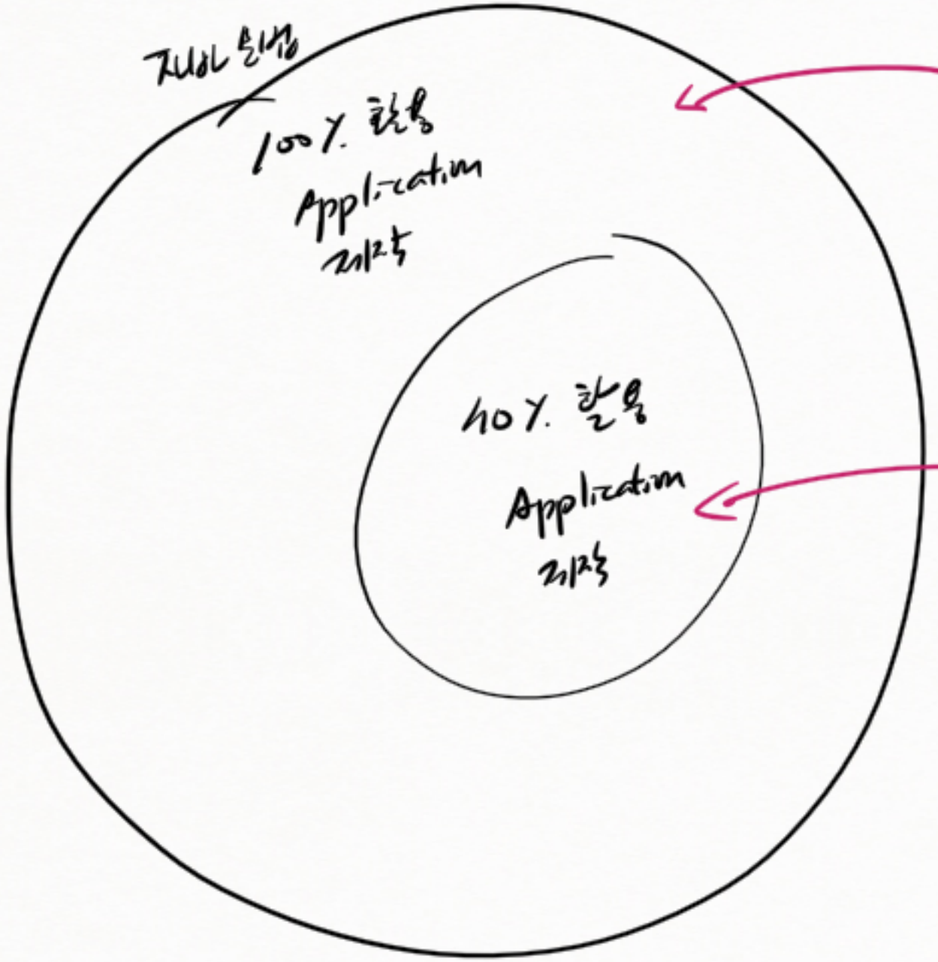
왜? 인터페이스에 새로 추가된 메서드로 일단 구현된 메서드이기 때문에 추상메서드가 아니다!

* default 메서드

⇓

과거에 작성된 코드에 영향을 주지 않으면서 신규 규칙을 추가할 수 있음!

* 자바 문법의 활용



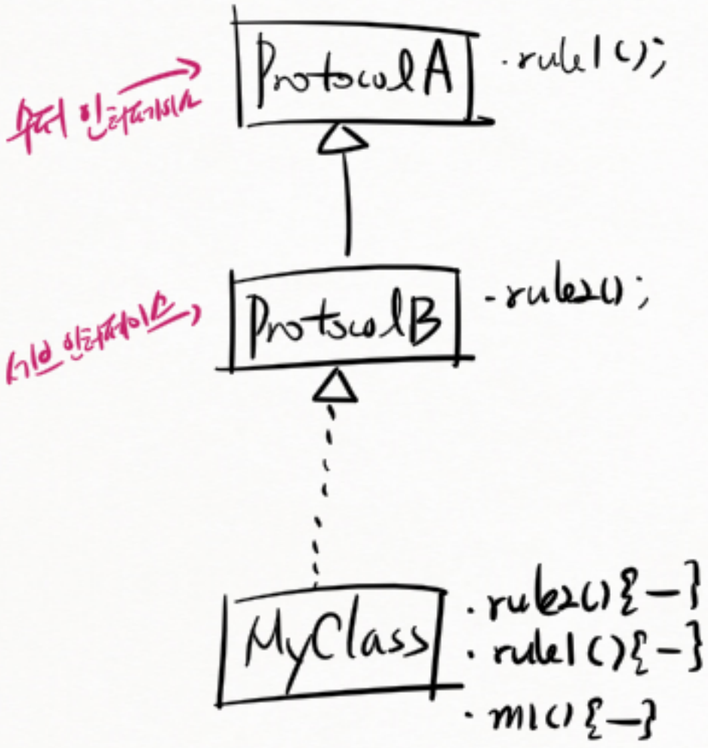
문법을 100% 완벽히 이해하여 사용하는 개발자는 많지 않다!

⇓

문법을 잘못 이해 → 코드를 엉망

대부분의 개발자는 알고 있는 문법 내에서 코딩을 작성 → 유지보수에 좋다.

* 인터페이스의 상속 과 구현

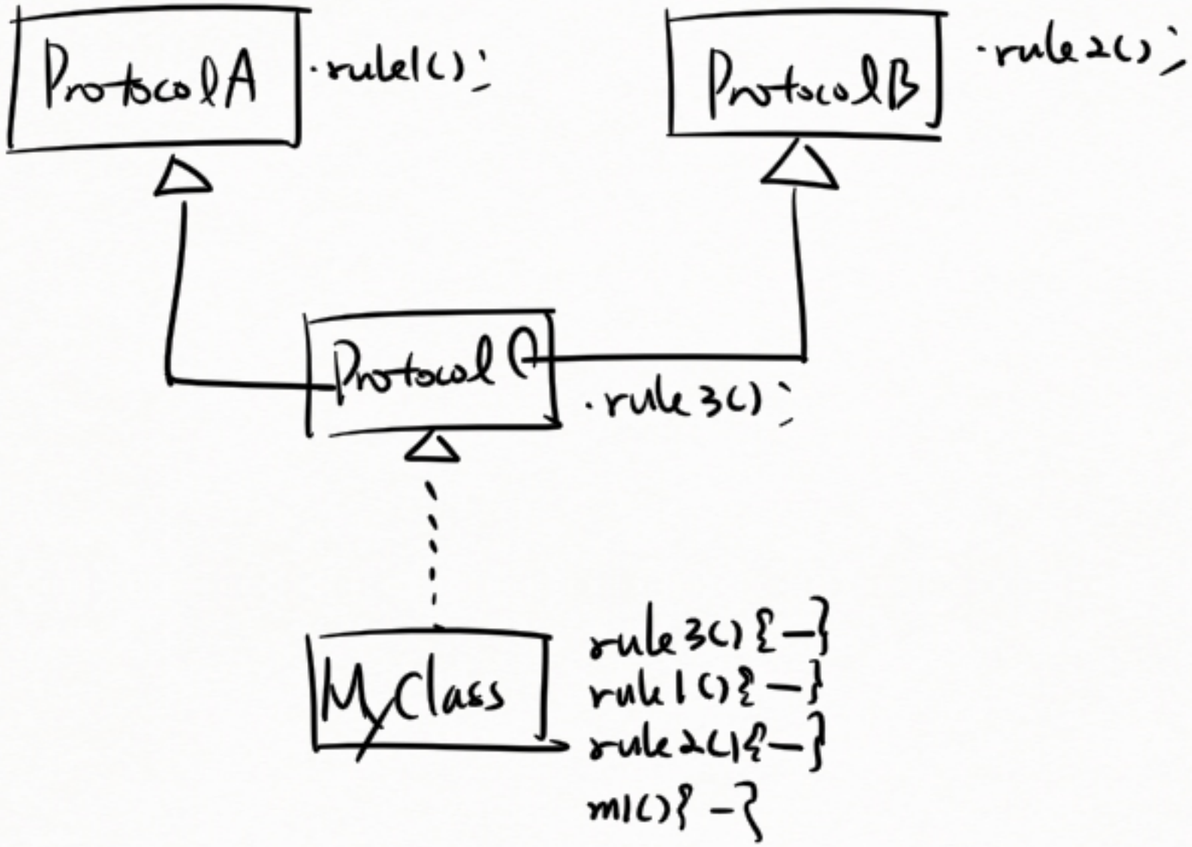



```
MyClass obj = new MyClass();
obj.m1();
obj.rule1();
obj.rule2();

ProtocolB obj2 = obj;
obj2.rule2();
obj2.rule1();
obj2.m1();

ProtocolA obj3 = obj;
obj3.rule1();
obj3.rule2();
obj3.m1();
```

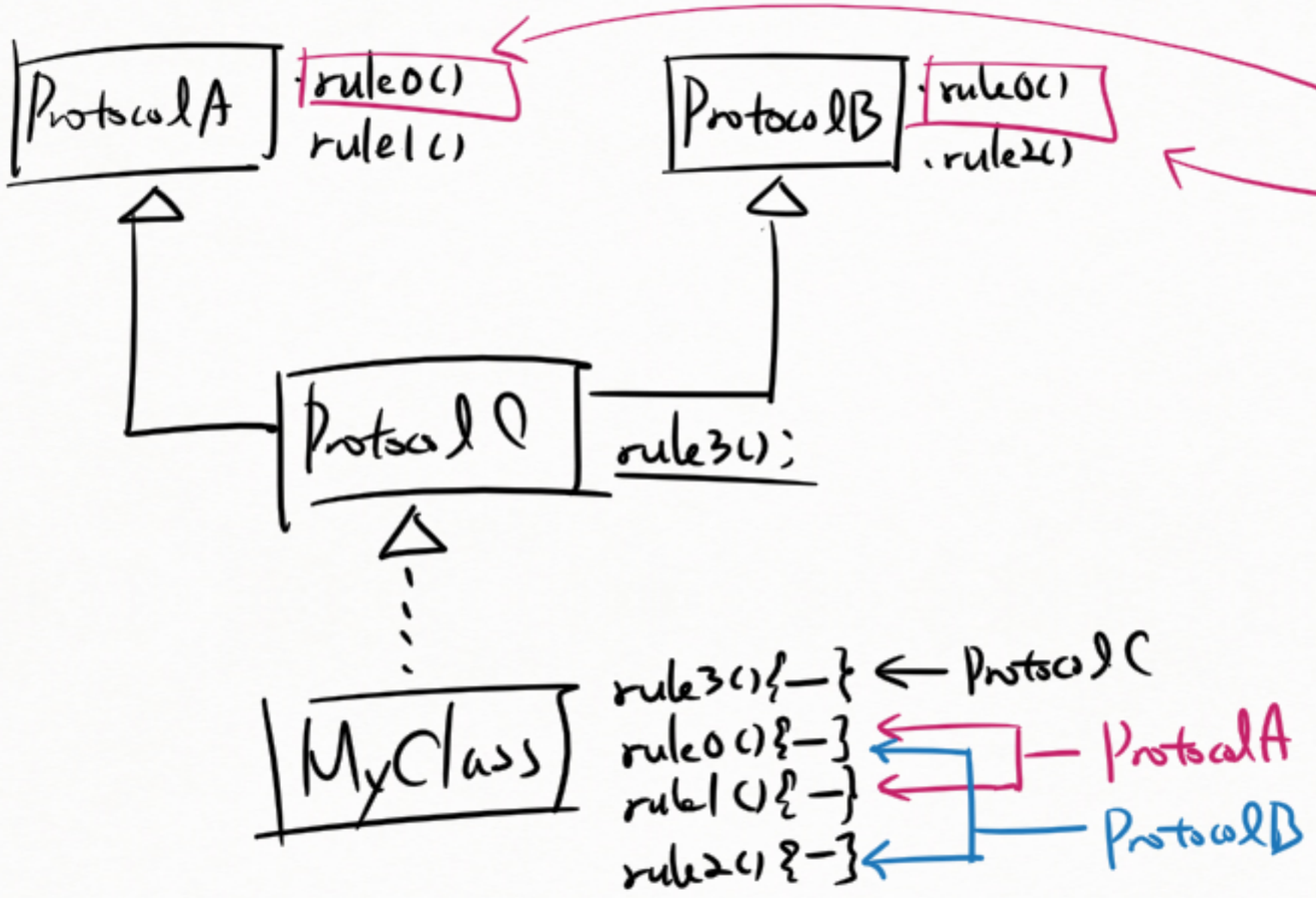
ProtocolB에 선언된 메서드가 아니다.

ProtocolA의 메서드가 아니다.

* 인터페이스의 상속 및 구현 II



* 다중 상속과 메서드 중복

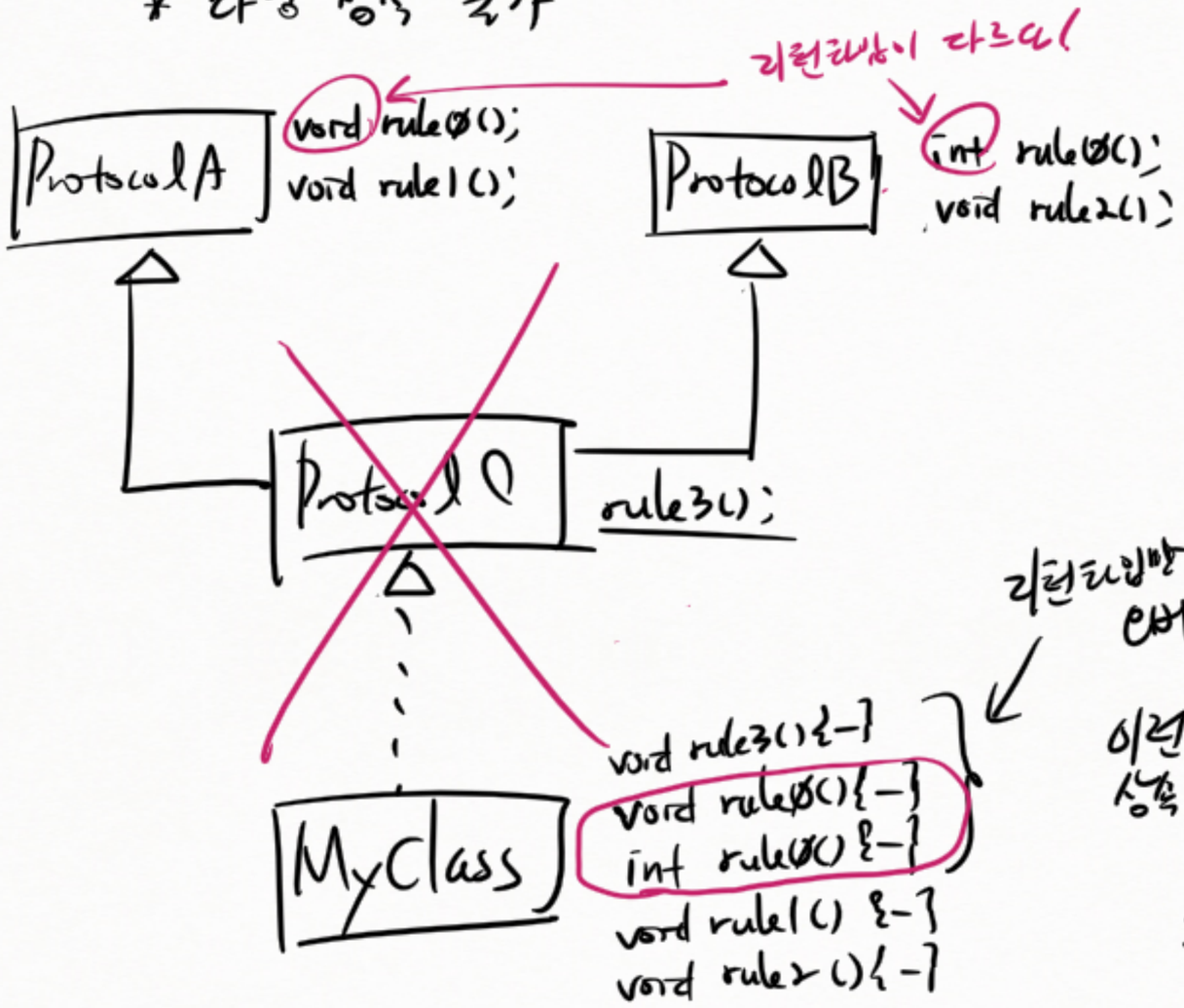


메서드가 중복되더라도 상관없다.

⇓

추상메서드이기 때문에 어떤 인터페이스의 메서드를 상속받더라도 구현과는 같다!

* 다중 상속 복가



리턴타입이 다른 경우
에러로 컴파일에 어긋난다.

⇓

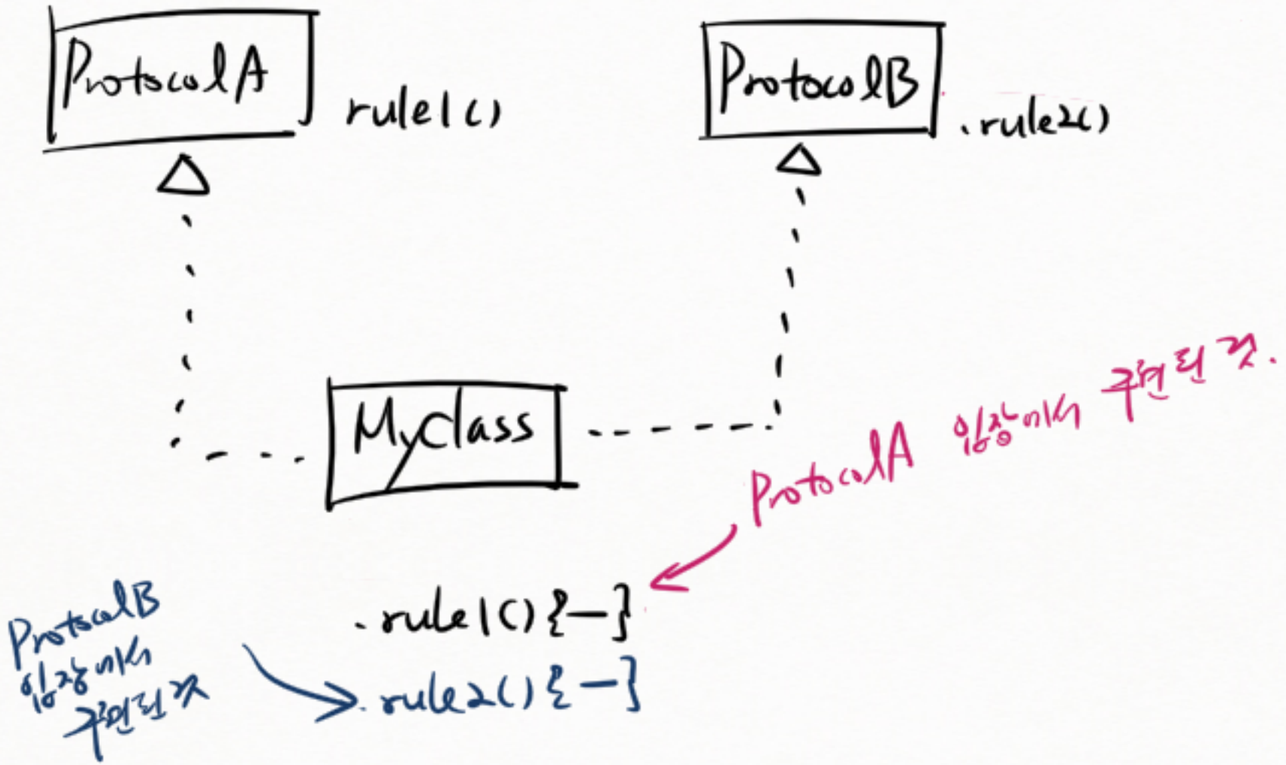
이런경우 둘 중 어떤 메서드를
상속받느냐에 따라 동작이 달라진다

⇓

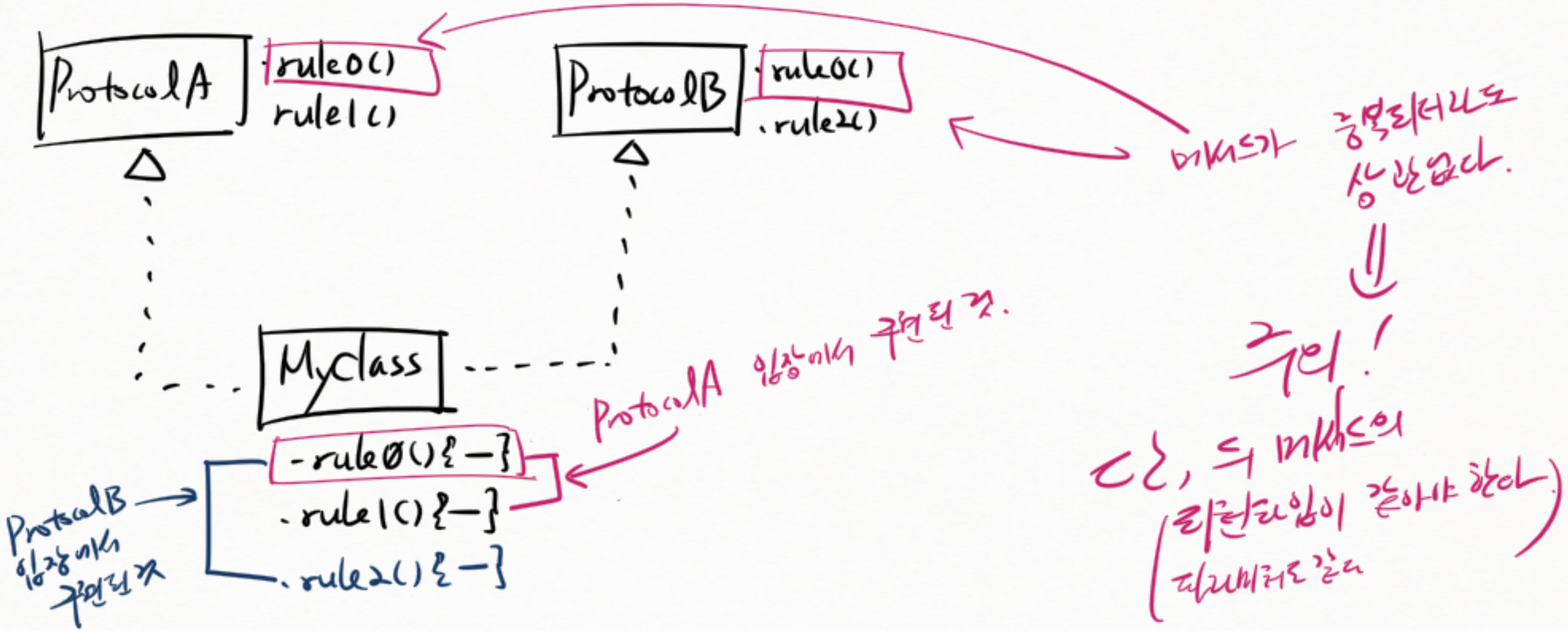
그래서 다중상속을 허용하지 않는다

이런경우

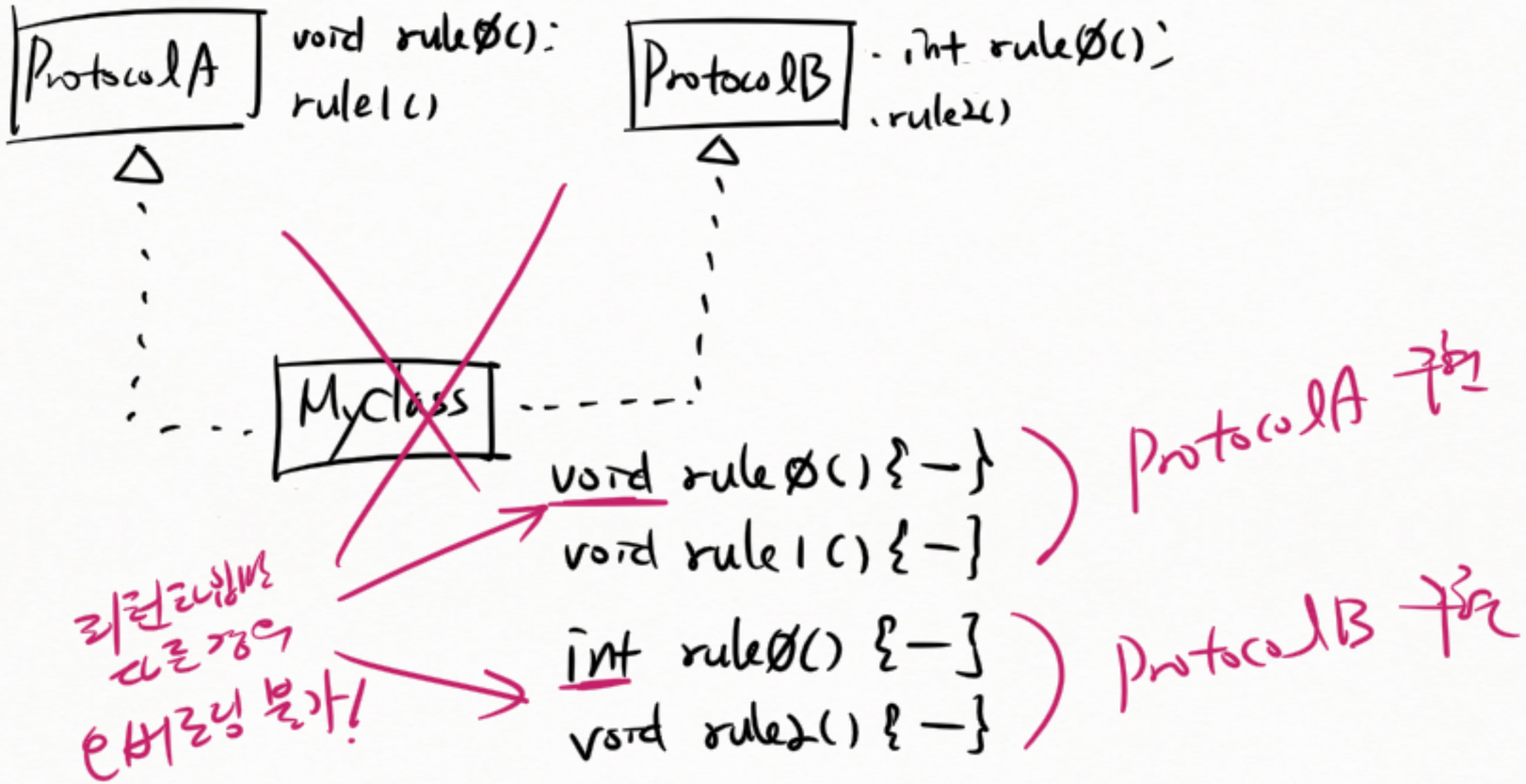
* 다중 인터페이스 구현


ProtocolA
rule1()
ProtocolB
rule2()
MyClass
ProtocolA 입장에서 구현된 것.
rule1(){—}
ProtocolB 입장에서 구현된 것
rule2(){—}

* 다중 인터페이스 구현 II

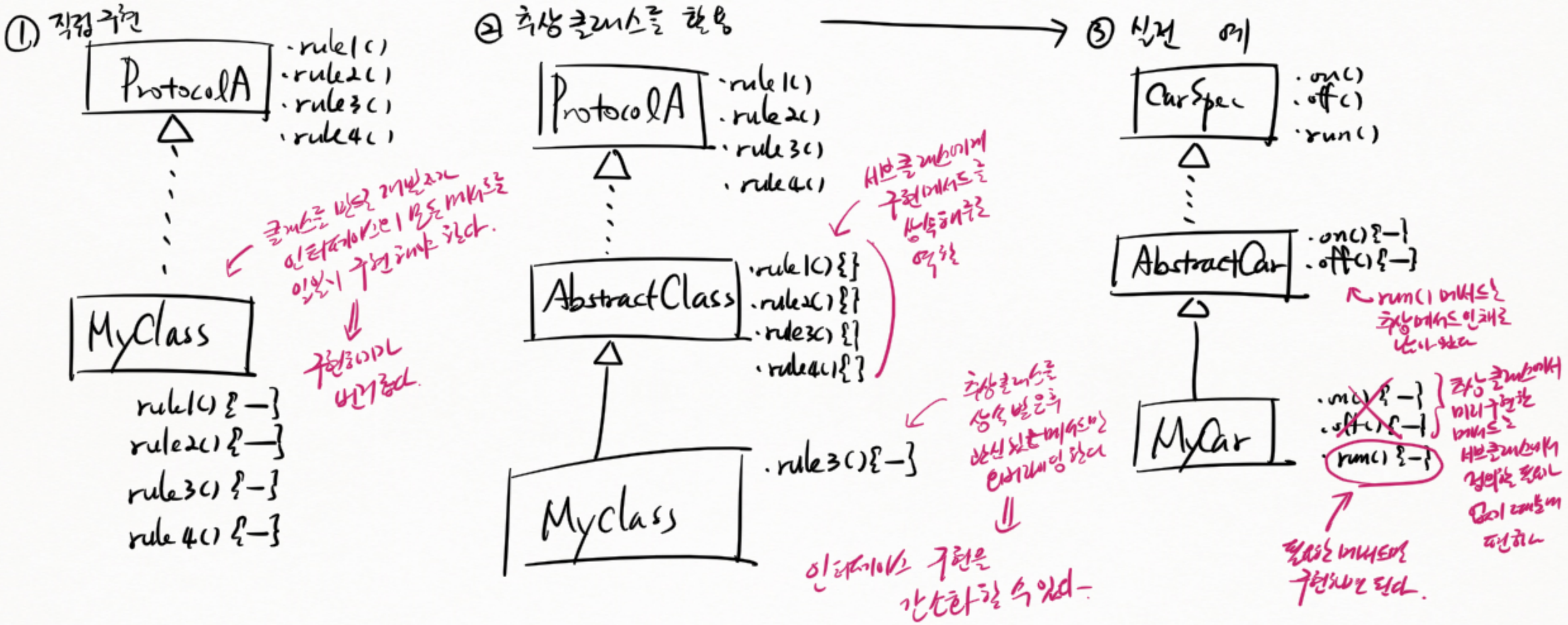


* 다중 인터페이스 구현 불가!

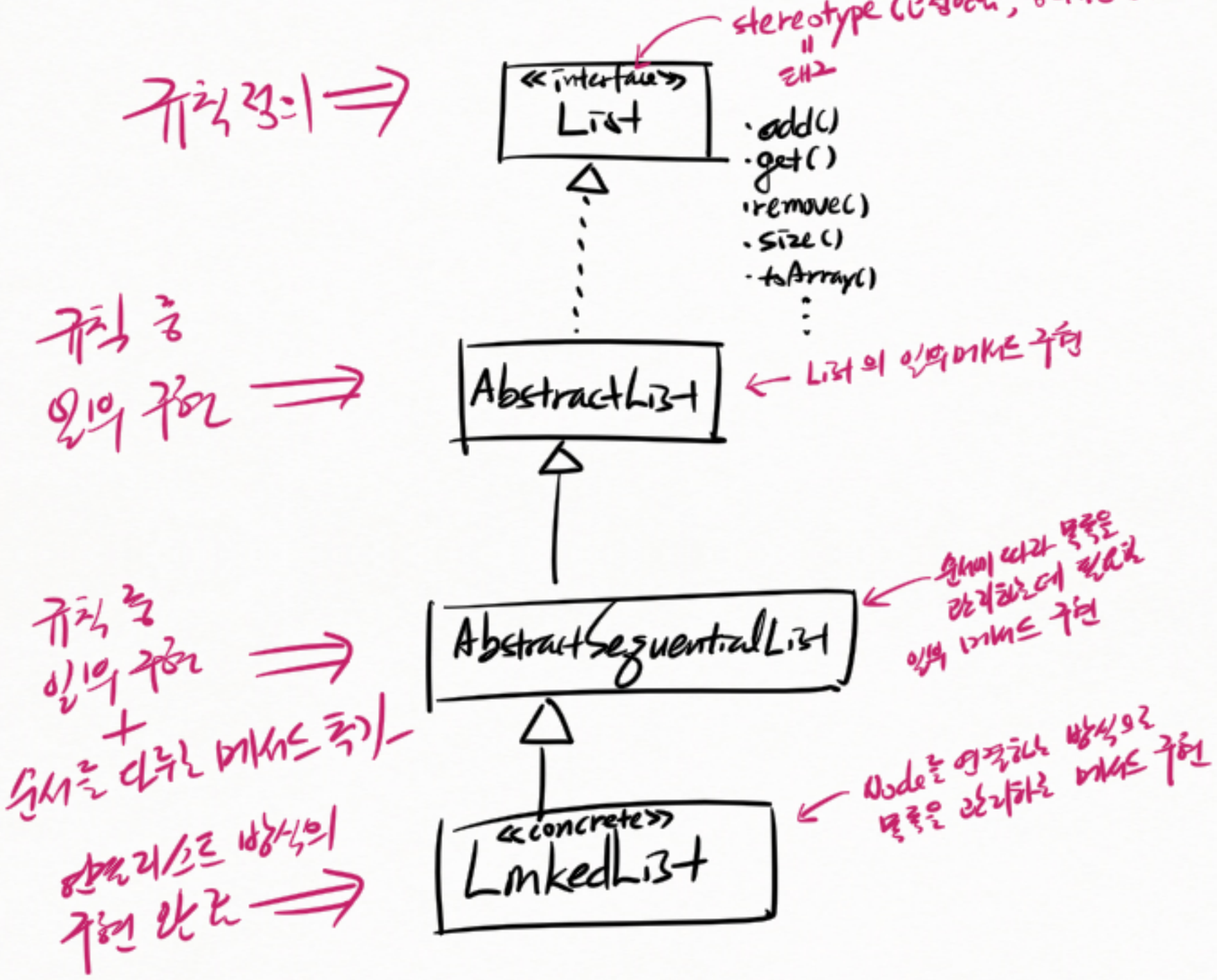


↳ 컴파일 에러!

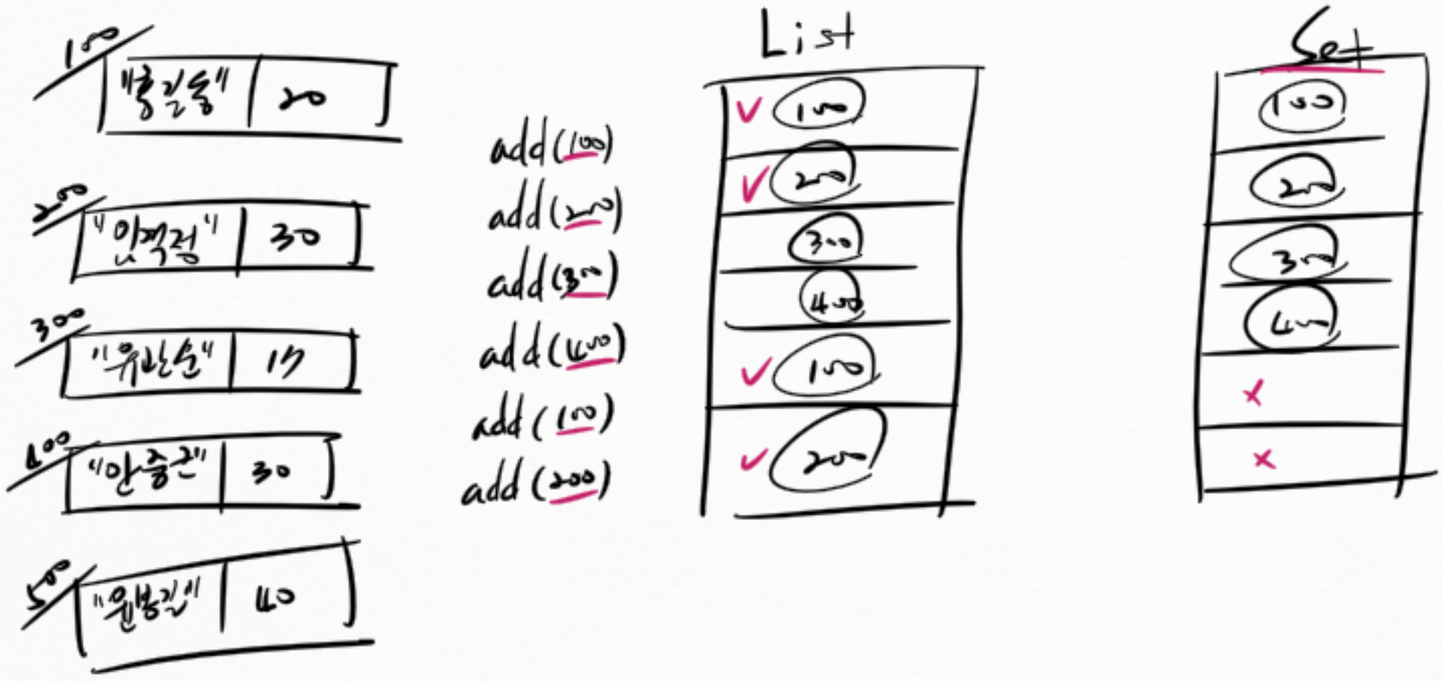
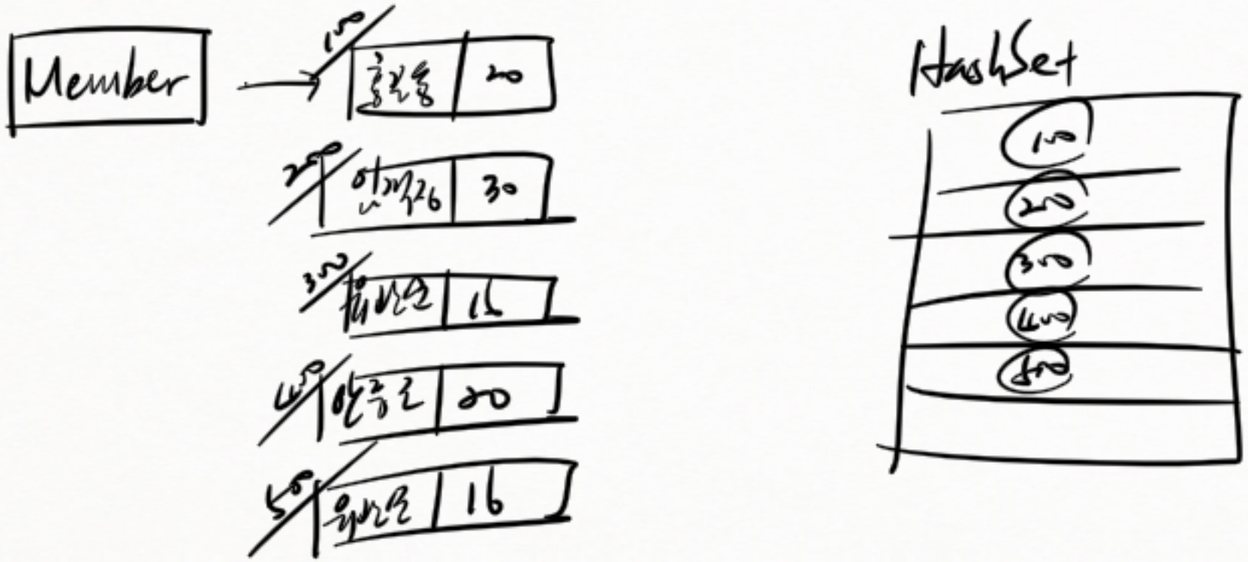
* 인터페이스 나- 추상 클래스



* 인터페이스와 추상클래스 활용 예



* List vs Set


100 "홍길동" 20
200 "임꺽정" 30
300 "유관순" 17
400 "안중근" 30
500 "윤봉길" 40
add(100)
add(200)
add(300)
add(400)
add(100)
add(200)
List
100
200
300
400
100
200
Set
100
200
300
400

* HashSet 과 equals(), hashCode()