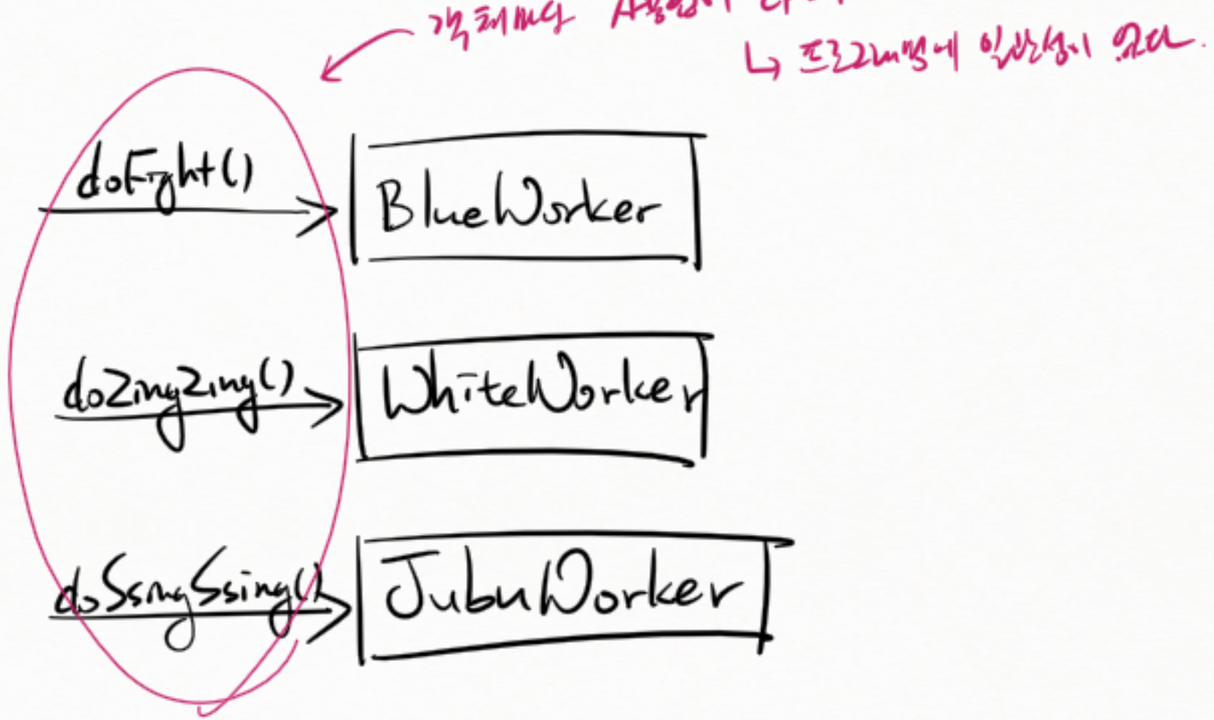
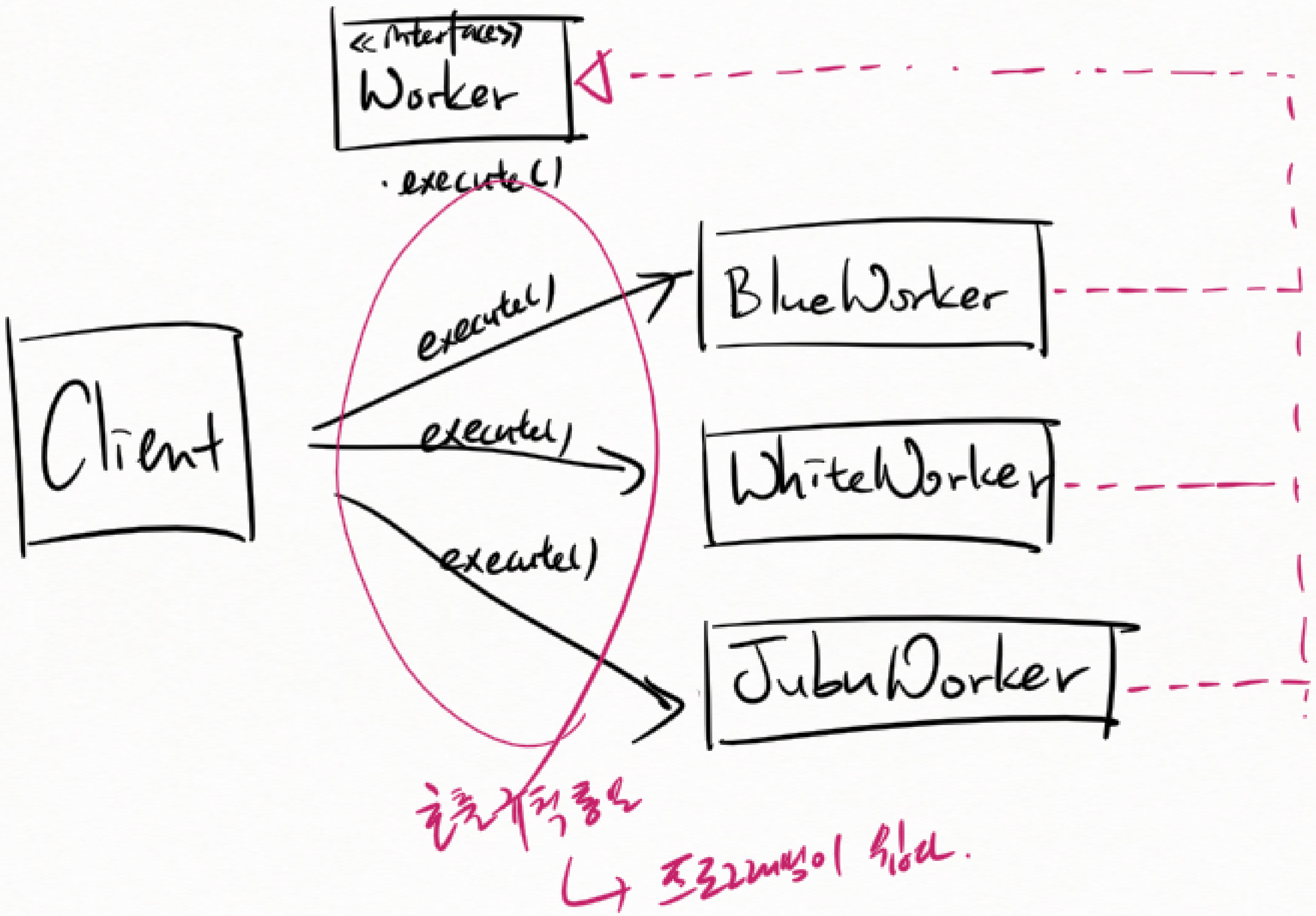
인터페이스 (interface)

* 인터페이스 사용 전

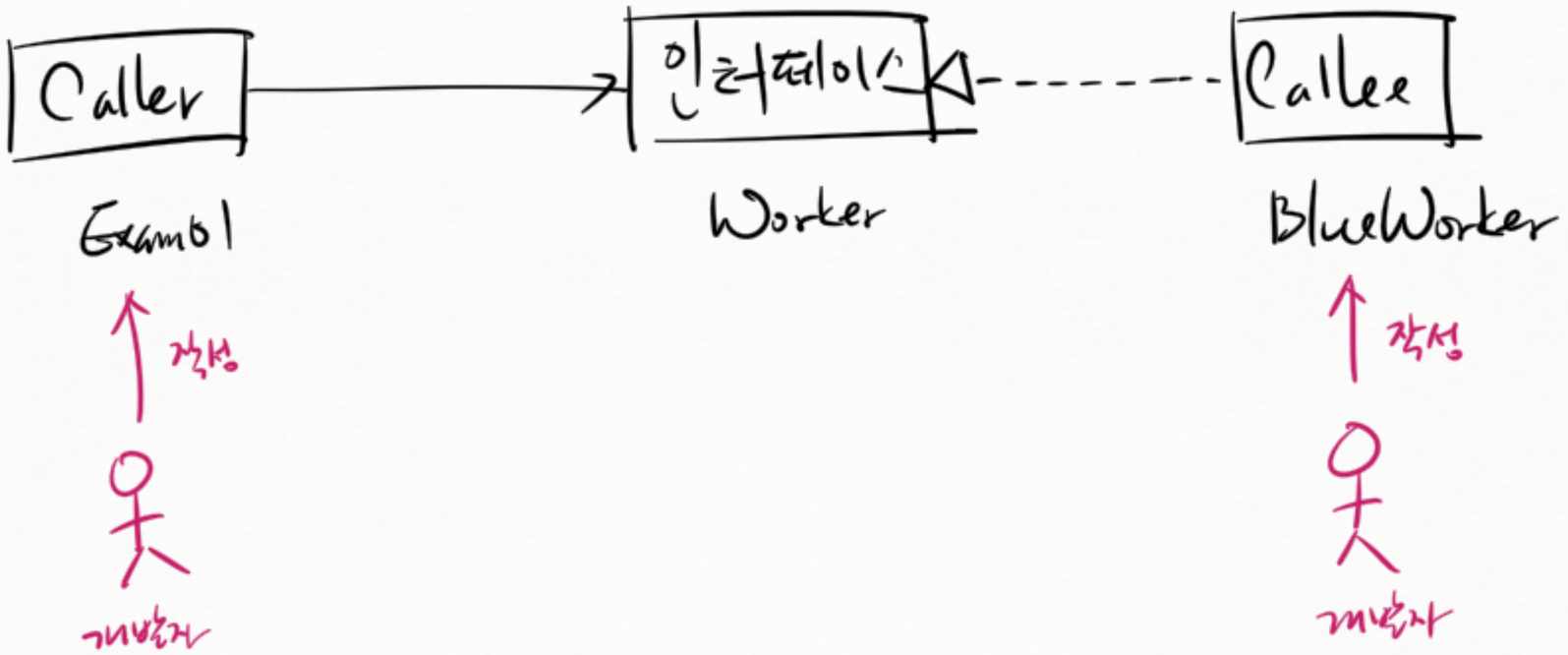

객체마다 사용방식 다르다.
↳ 프로그래밍에 일관성이 없다.
doFight()
BlueWorker
doZingZing()
WhiteWorker
doSsangSsing()
JubuWorker

* 인터페이스 사용 후

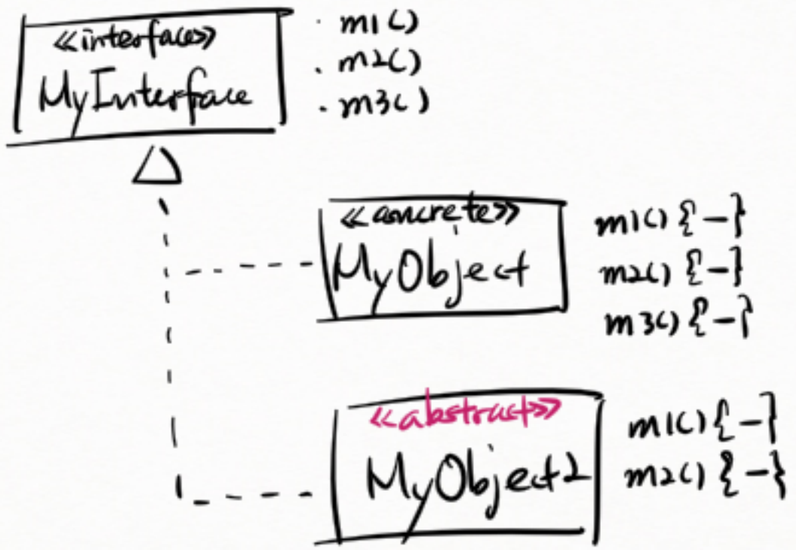



```
class BlueWorker
    implements Worker {
        ⋮
}
```

* 인터페이스 와 caller / callee



* 인터페이스의 구현



MyInterface ref;

ref = new MyObject();

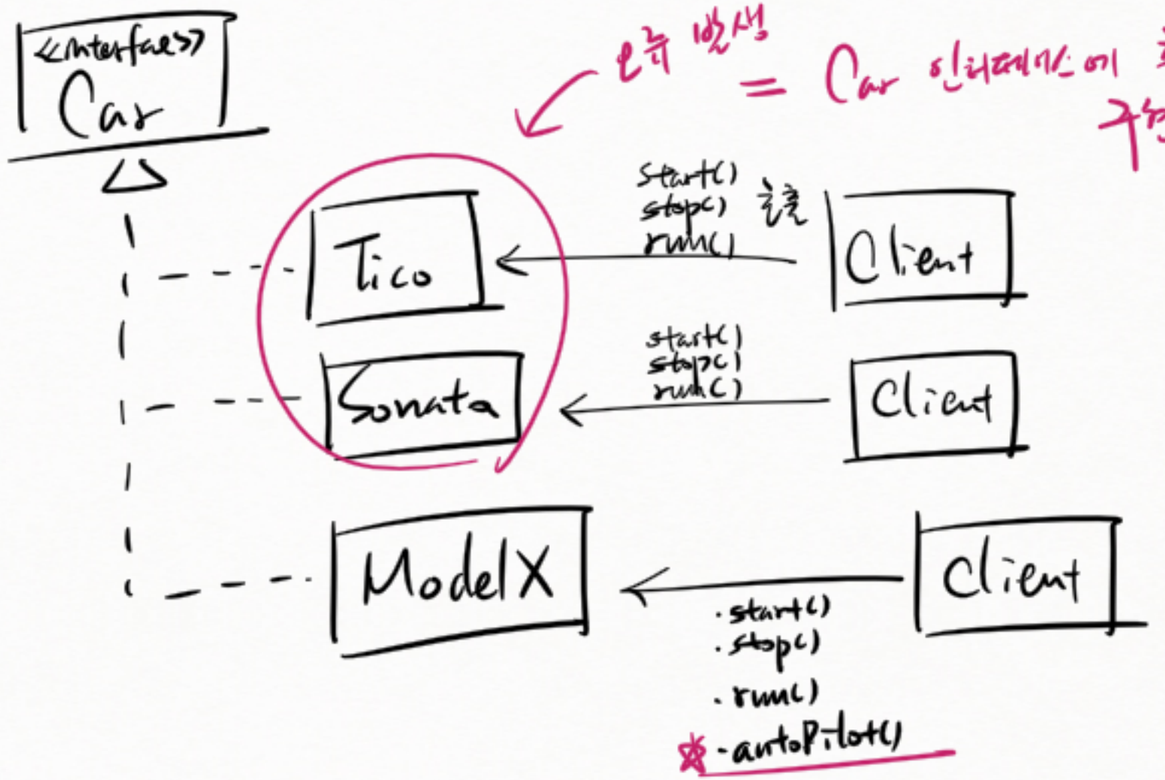
인터페이스를 구현한 클래스의 인스턴스 주소

* default 메서드 사용 전

```
interface Car {
    - start();
    - stop();
    - run();
}
```

↑ 규칙 추가

```
+ autoPilot();
```



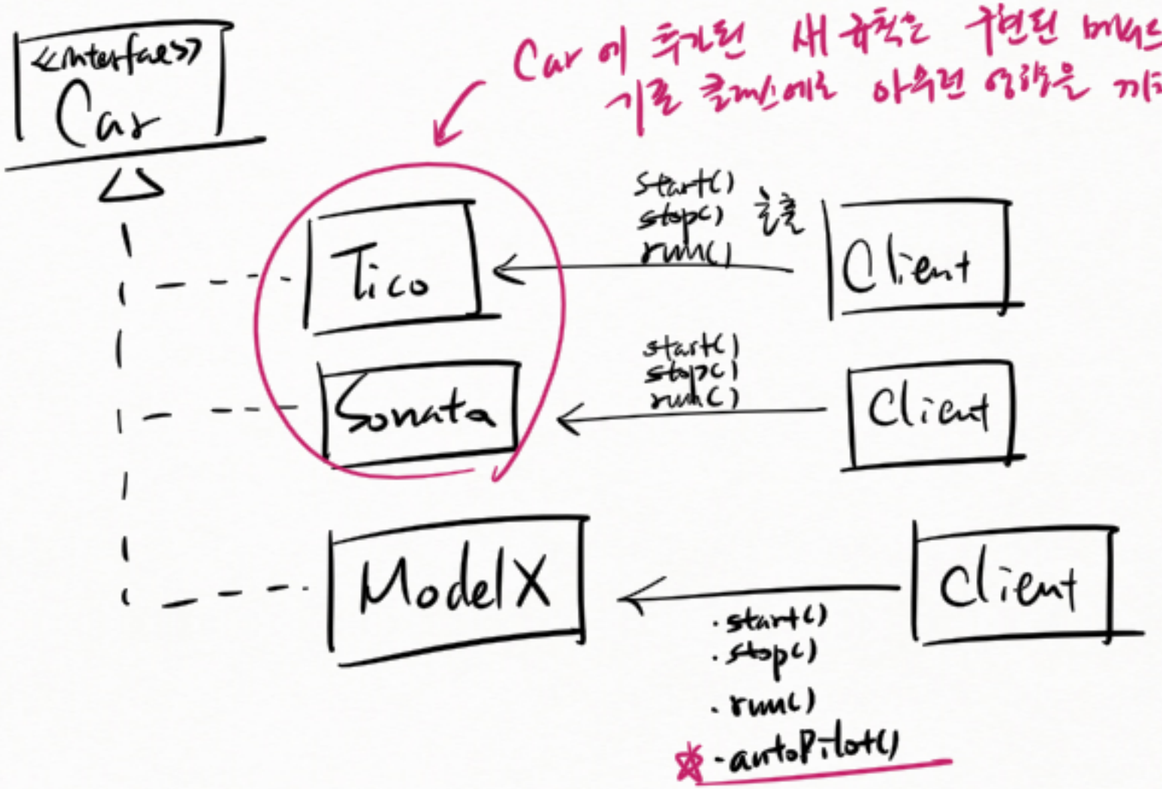
신규 발생 = Car 인터페이스에 추가된 새 규칙은 구현(정의)하지 않았기 때문이다.

* default 메서드 사용 예

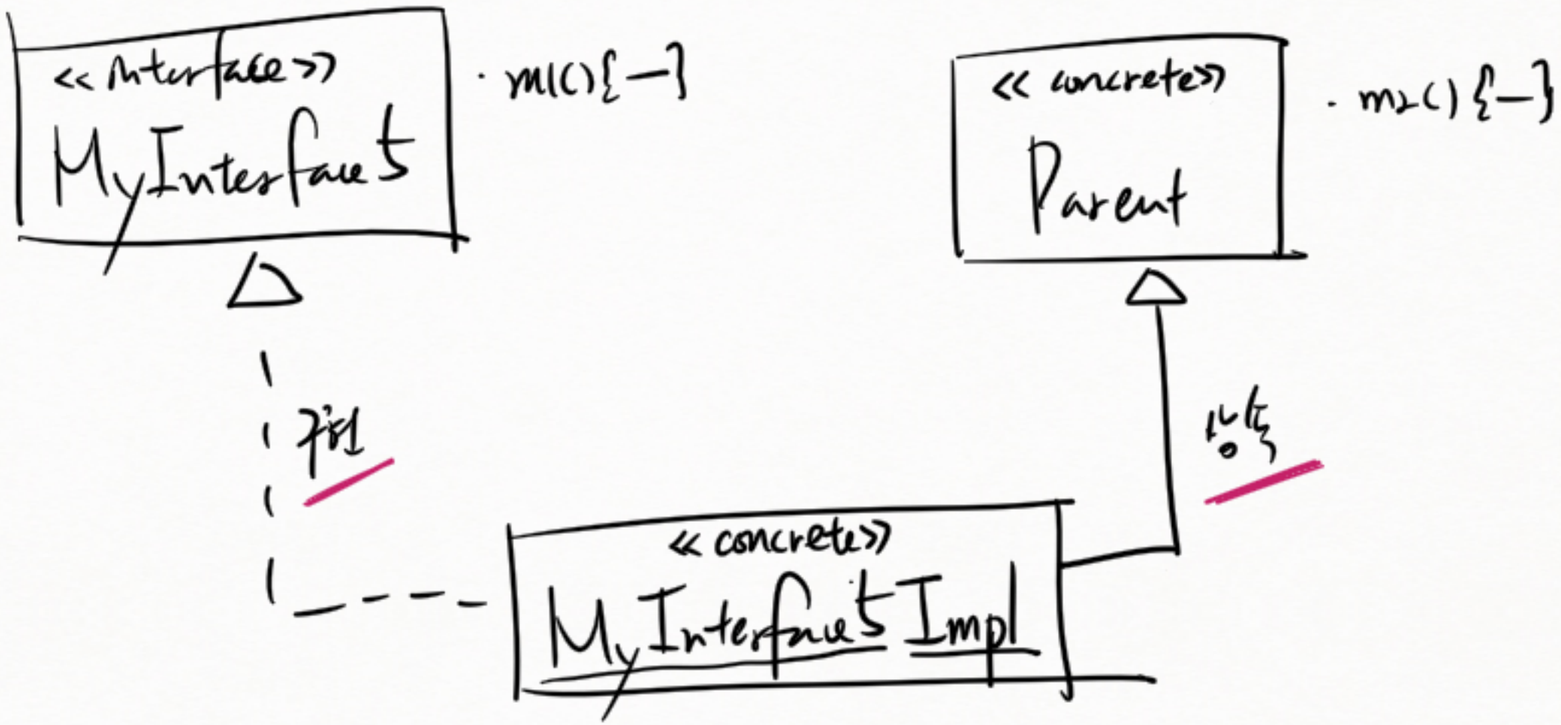
```
interface Car {
    - start();
    - stop();
    - run();
}
```

↑ 추가

```
default autoPilot() {}
```

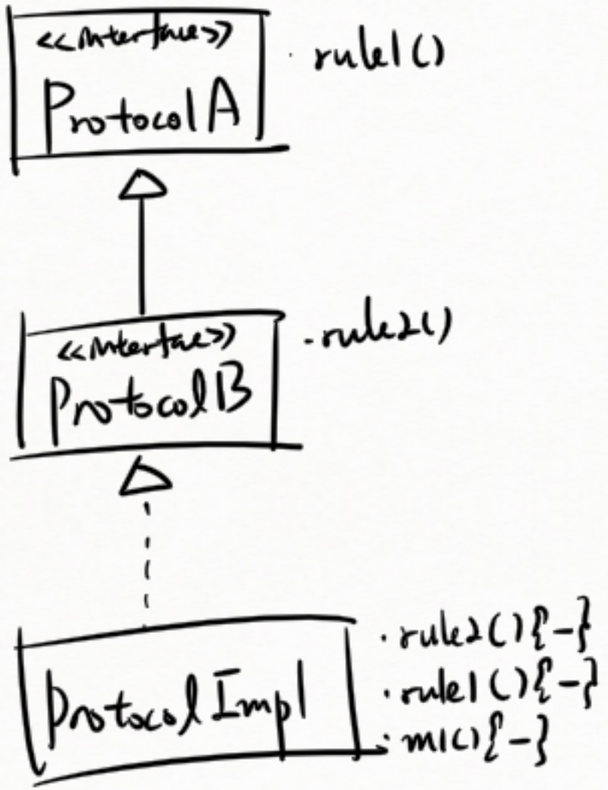


* static 바인딩


«Interface»
MyInterface
· m1() {—}
«concrete»
Parent
· m2() {—}
구현
상속
«concrete»
MyInterfaceImpl

* 인터페이스 상속 및 구현

↳ 레퍼런스를 통한 메서드 호출 법칙



ProtocolImpl obj = new ProtocolImpl();

obj.m1();
obj.rule1();
obj.rule2();

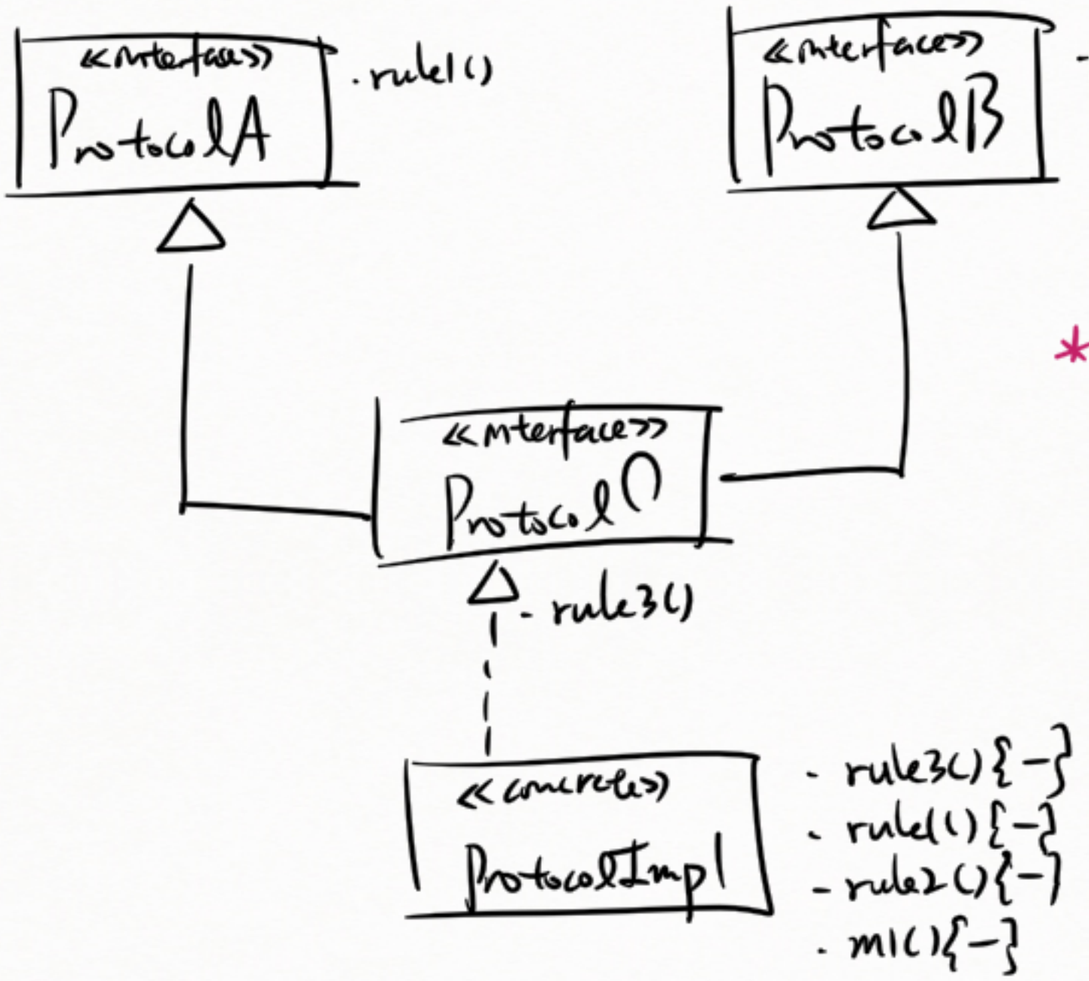
ProtocolB obj2 = obj;

obj2.m1();
obj2.rule2();
obj2.rule1();

obj3.m1();
obj3.rule2();
obj3.rule1();

ProtocolA obj3 = obj;

* 인터페이스 다중 상속



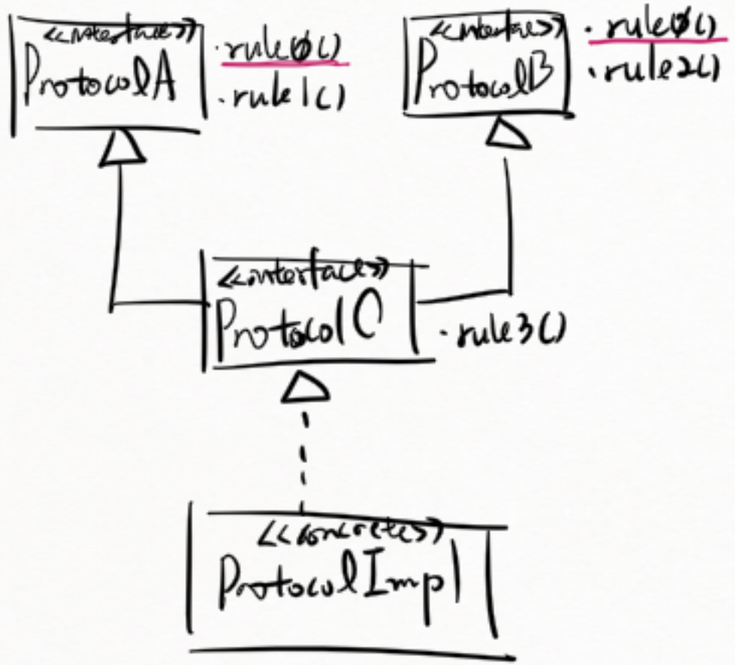
* 인터페이스의 메서드 추상메서드이기 때문에 다중상속을 해도 문제가 될 소지가 없다

⇓

다중 상속 가능!

* 인터페이스 다중상속

가능한 경우

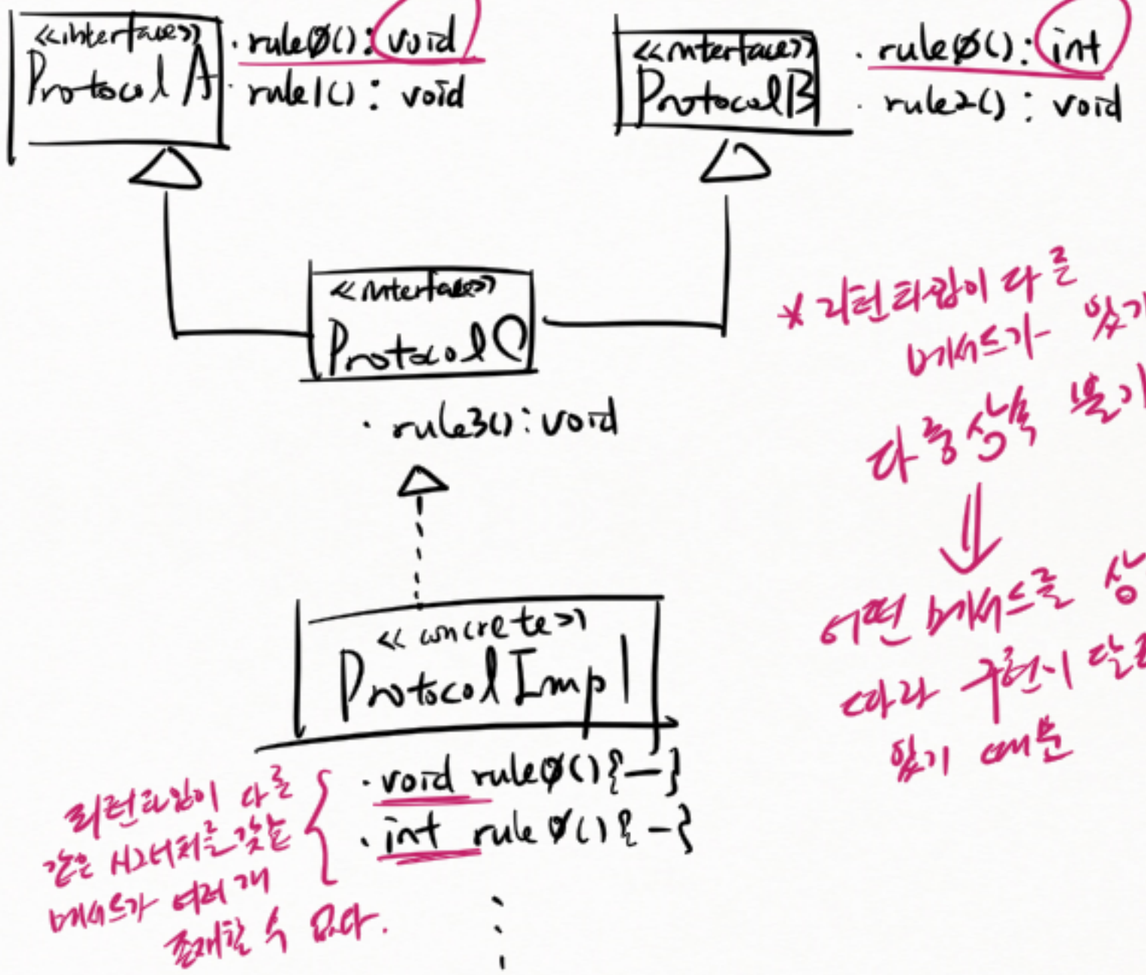


ProtocolA
- rule0(){-}
- rule1(){-}

ProtocolB
- rule0(){-}
- rule2(){-}

ProtocolC
- rule3(){-}

불가능한 경우



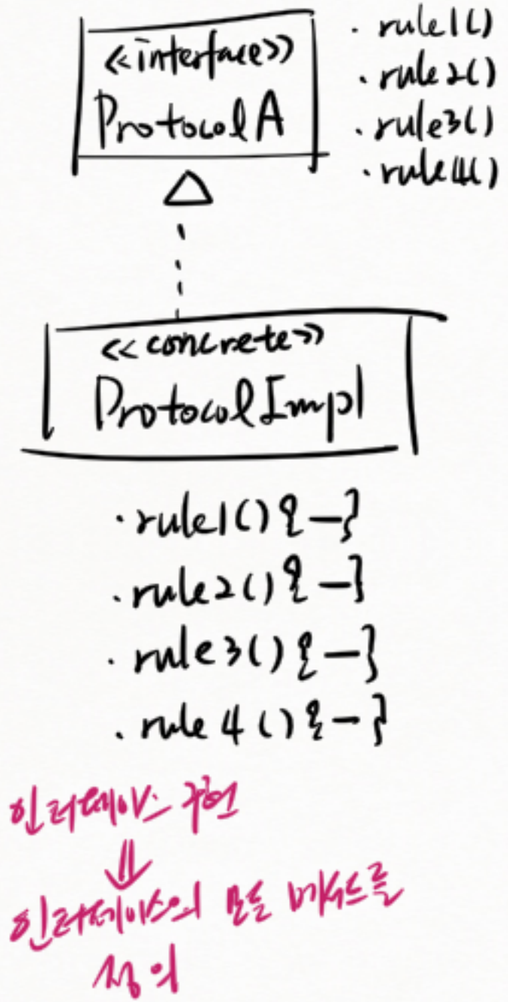
리턴타입이 다른 같은 시그니처를 갖는 메소드 여러 개 정의할 수 없다.
- void rule0(){-}
- int rule0(){-}

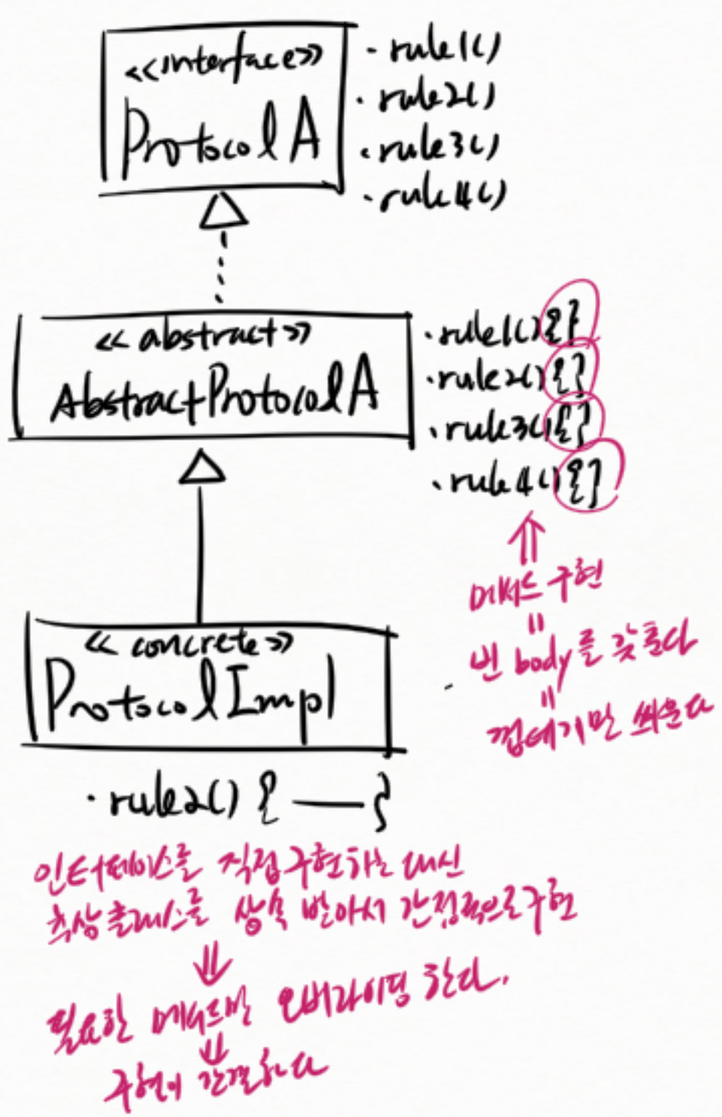
* 리턴타입이 다른 메소드가 여러 개 있을 때 다중상속 불가!

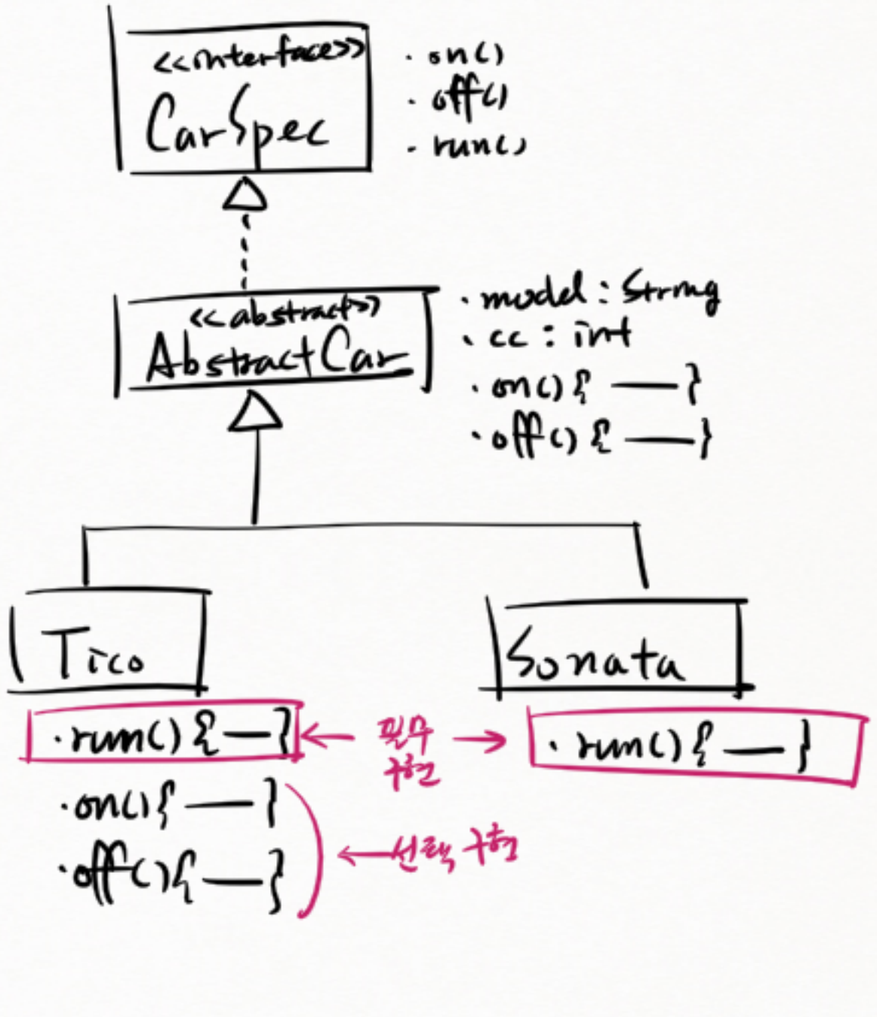
⇓

어떤 메소드를 상속받느냐에 따라 구현이 달라질 수 있기 때문

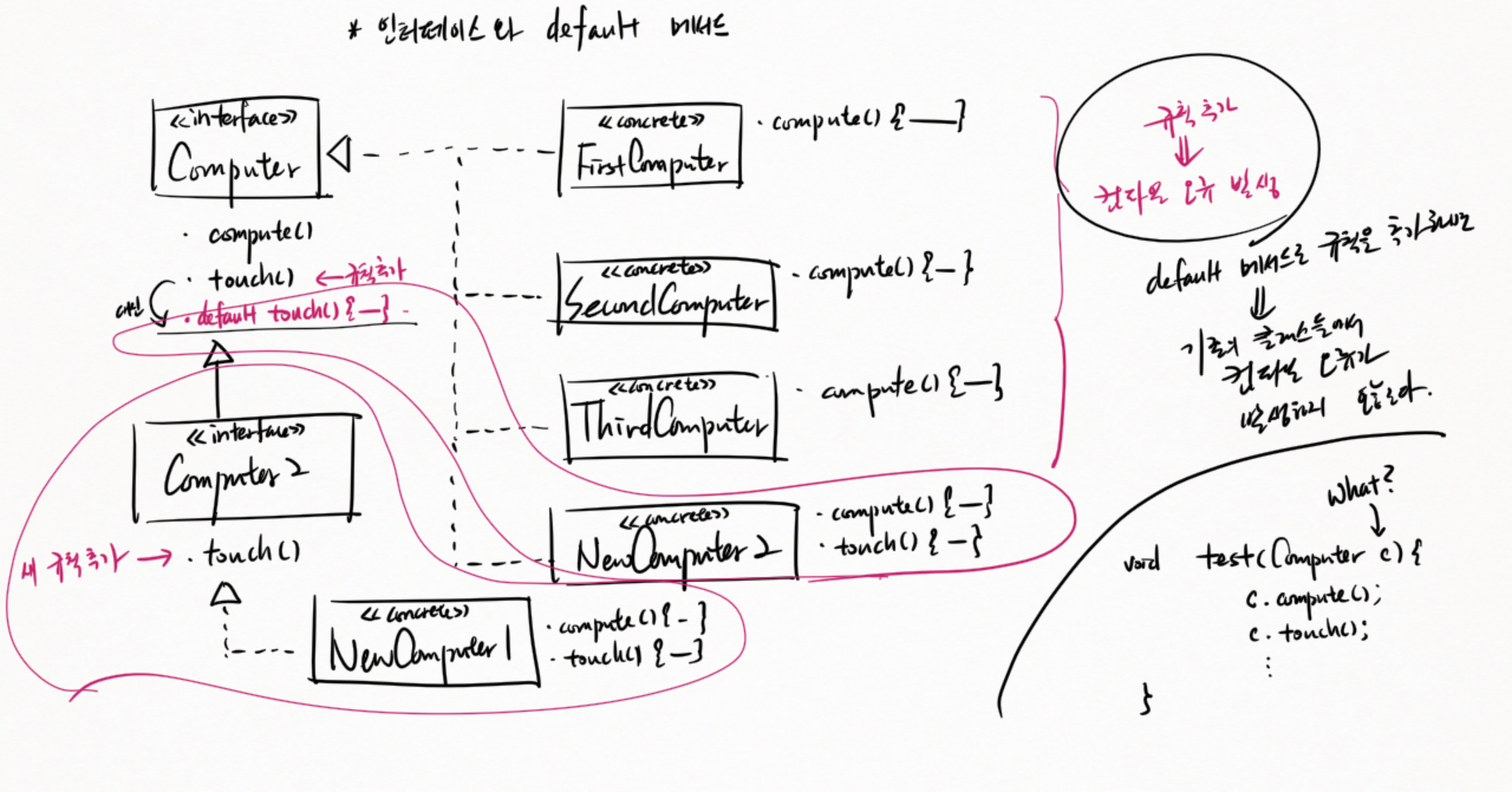
* 인터페이스와 추상 클래스







* 인터페이스와 default 메서드

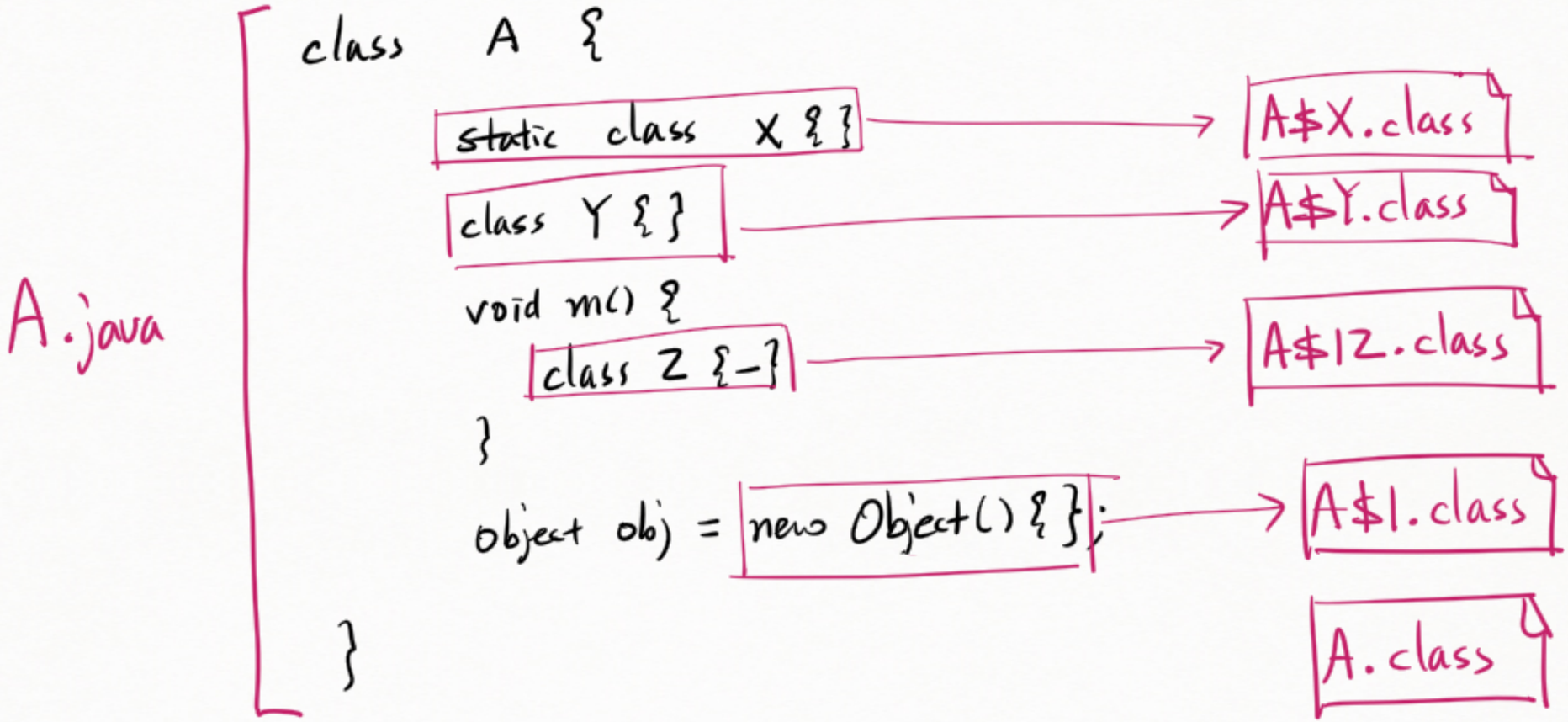


중첩 클래스 (nested class)

* 중첩 클래스의 종류
nested

```
class A {
    static class X { }   ← static nested class
    class Y { }          ← non-static nested class
    void m() {
        class Z { }      ← local class
    }
    Object obj = new Object() { };   ← anonymous class
}
```

* 중첩 클래스와 .class 파일



* static nested class 와 non-static nested class 의 차이

```
class A {
    static class X {}
    class Y {}
}
```

```
Y obj = new Y();
```

⇓

```
new Y(this);
```

컴파일러가 바깥 클래스의 객체 주소를 넘기는 코드로 자동 변경

```
static class X {
    X() { - }
}
```

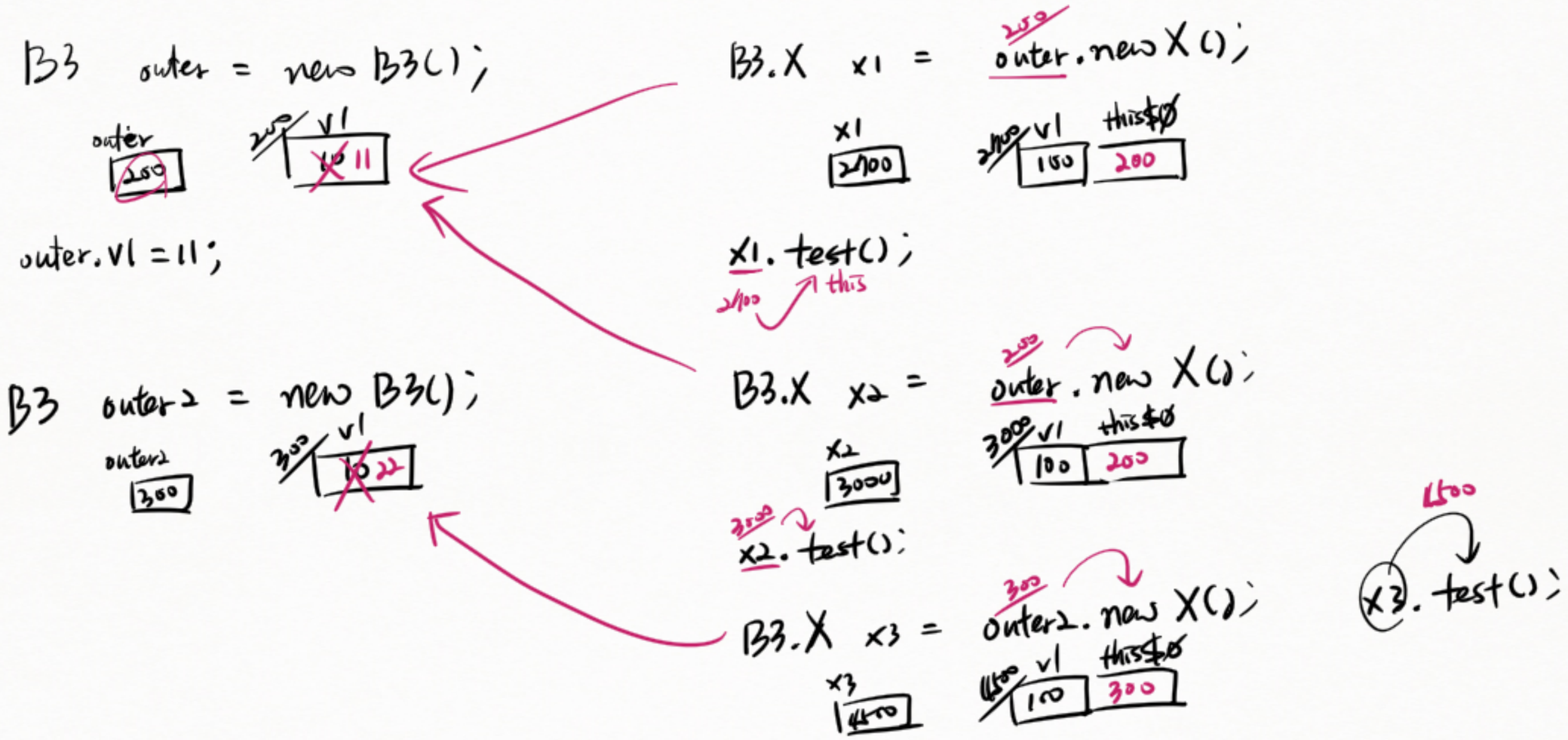
← 컴파일러가 기본 생성자 추가

컴파일러가,

```
class Y {
    A this$0;
    Y(A arg) {
        this$0 = arg;
    }
}
```

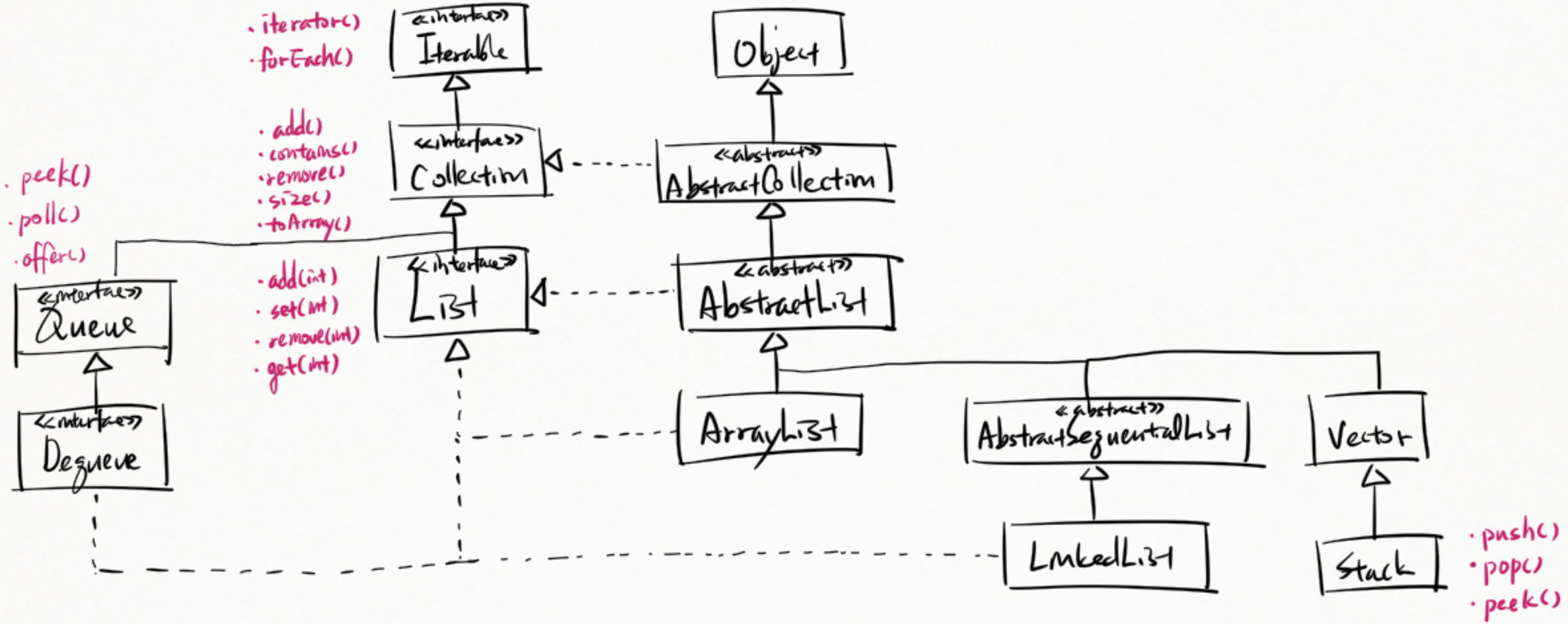
← 바깥 클래스의 객체 주소를 받은 레퍼런스를 추가.

← 생성자에서 바깥 클래스의 인스턴스를 받을 수 있도록 파라미터를 추가

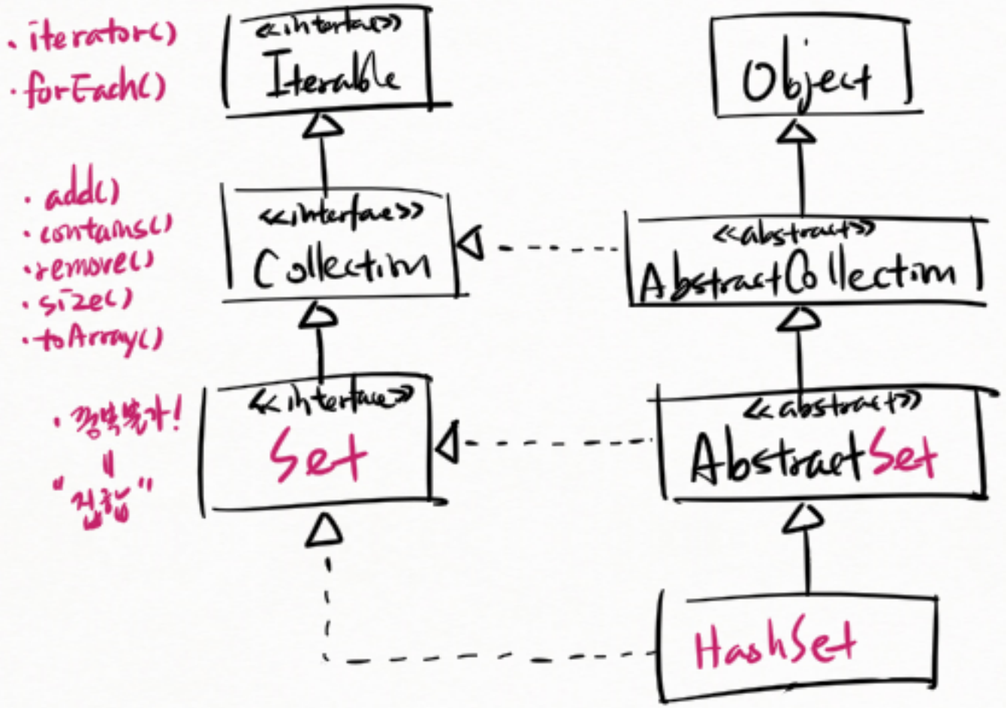

B3 outer = new B3();
outer
200
200
v1
10 11
outer.v1 = 11;
B3.X x1 = 250 outer.new X();
x1
2100
2100
v1
this$0
100
200
x1.test();
2100
this
B3 outer2 = new B3();
outer2
300
300
v1
10 22
B3.X x2 = 2500 outer.new X();
x2
3000
3000
v1
this$0
100
200
2500
x2.test();
B3.X x3 = 300 outer2.new X();
x3
4400
4400
v1
this$0
100
300
4400
x3.test();

Collection API

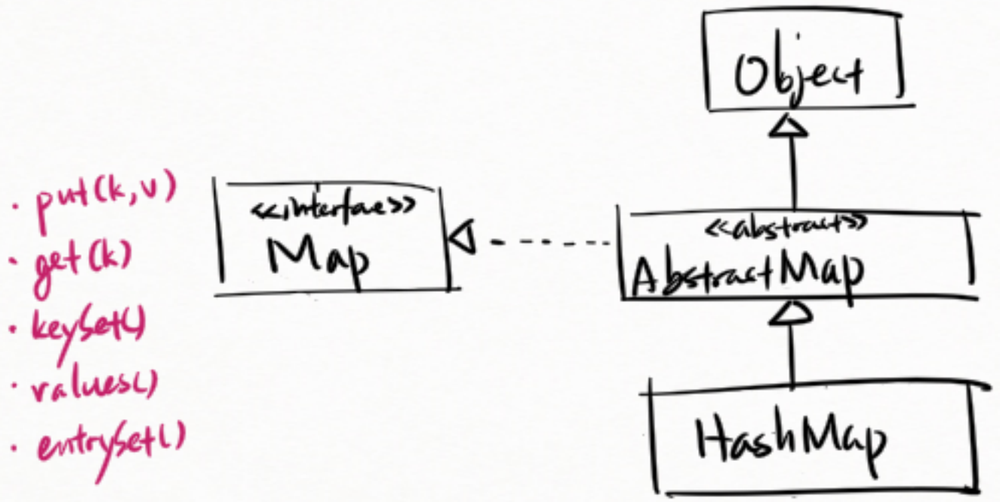
* List 관련 클래스 계층도 (hierarchy)

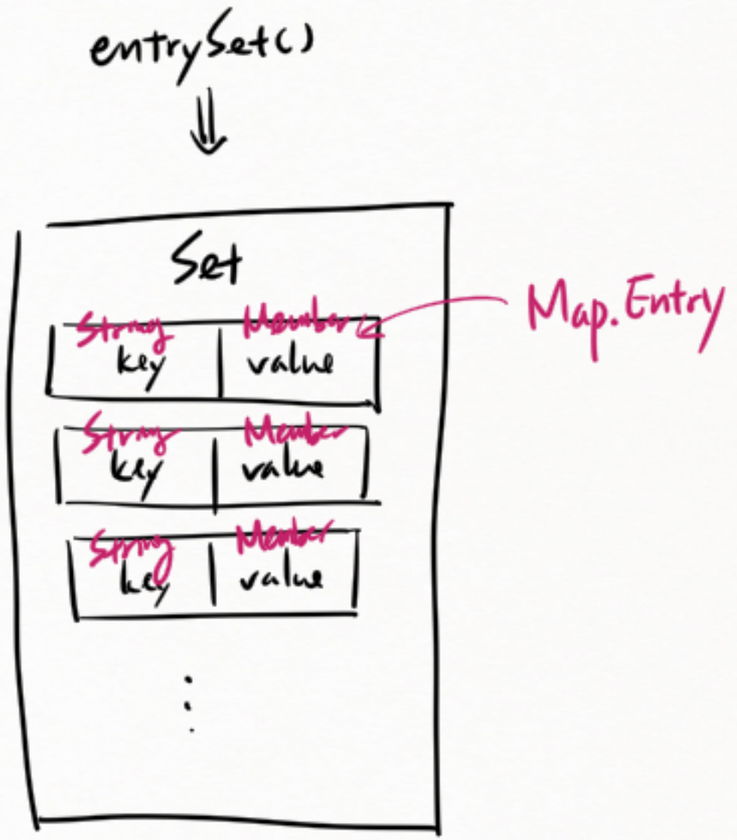


* Set 관련 클래스 계층도 (hierarchy)



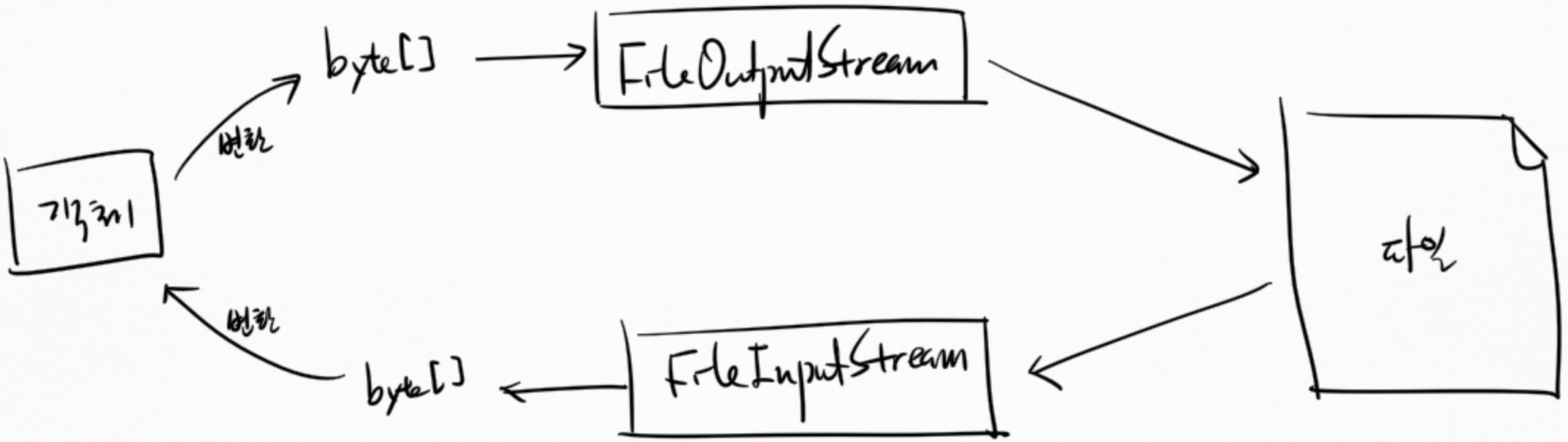
* Map 관련 클래스 계층도 (hierarchy)





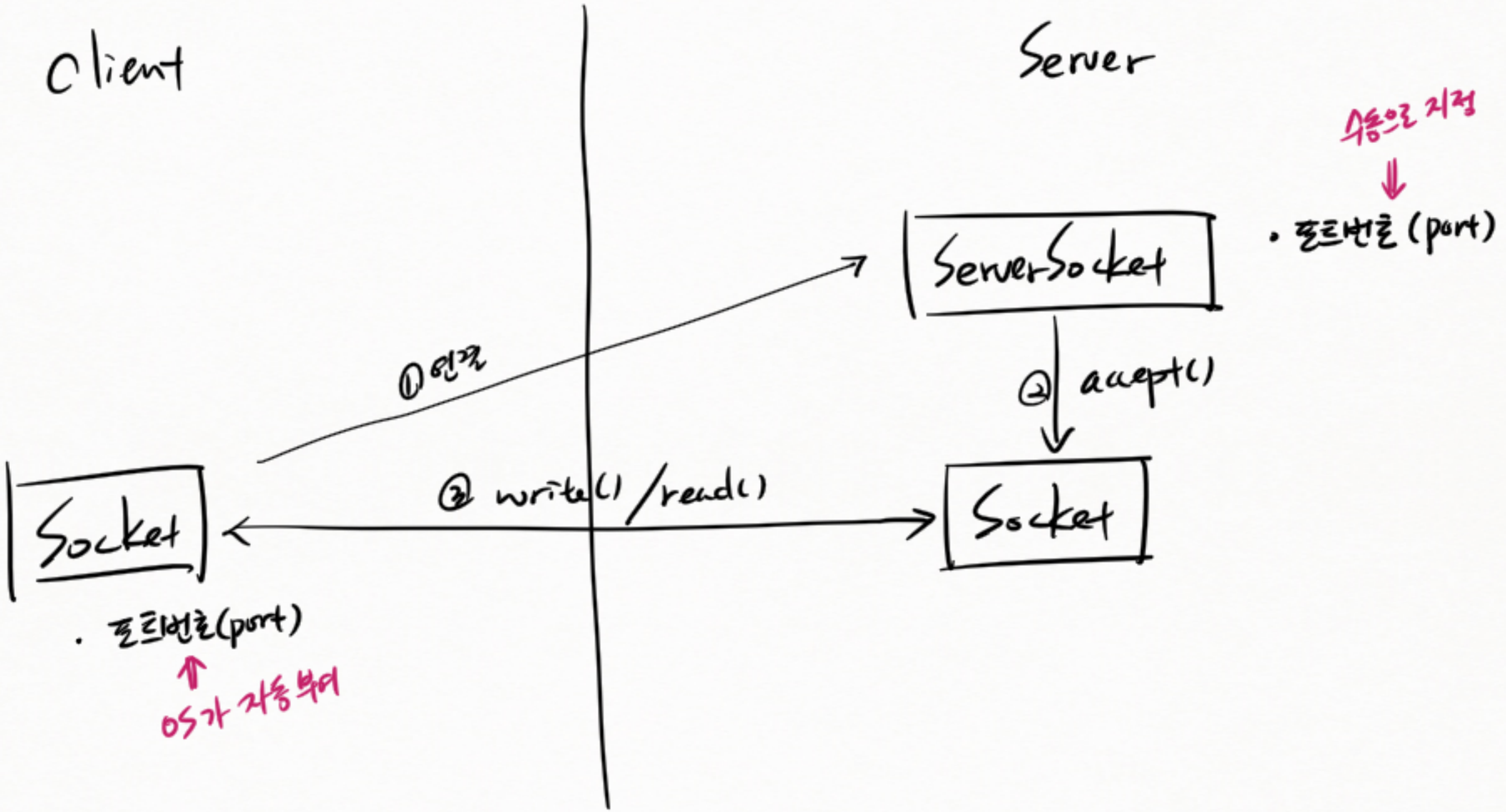
File I/O API

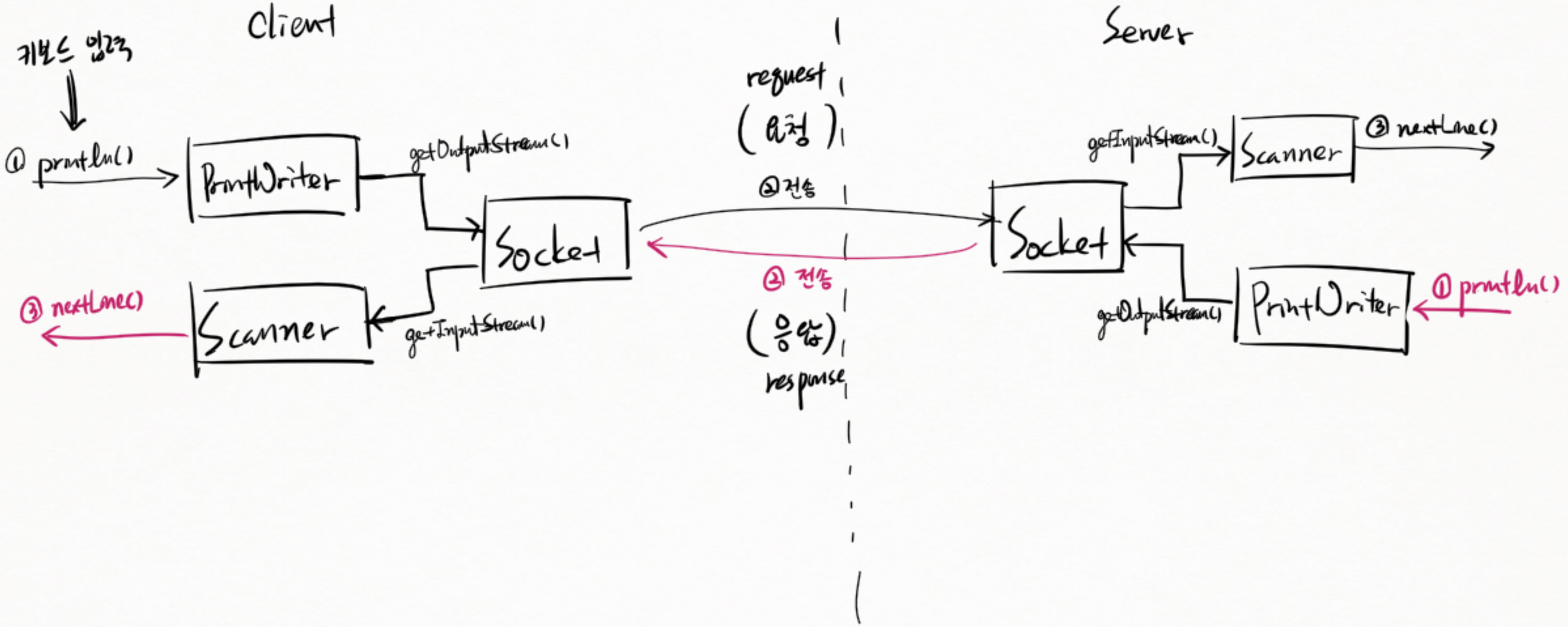
* FileOutputStream / FileInputStream



Networking

Client / Server




Client
Server
키보드 입력
① println()
PrintWriter
getOutputStream()
Socket
Scanner
getInputStream()
③ nextLine()
request
(요청)
② 전송
② 전송
(응답)
response
getInputStream()
Scanner
③ nextLine()
Socket
getOutputStream()
PrintWriter
① println()