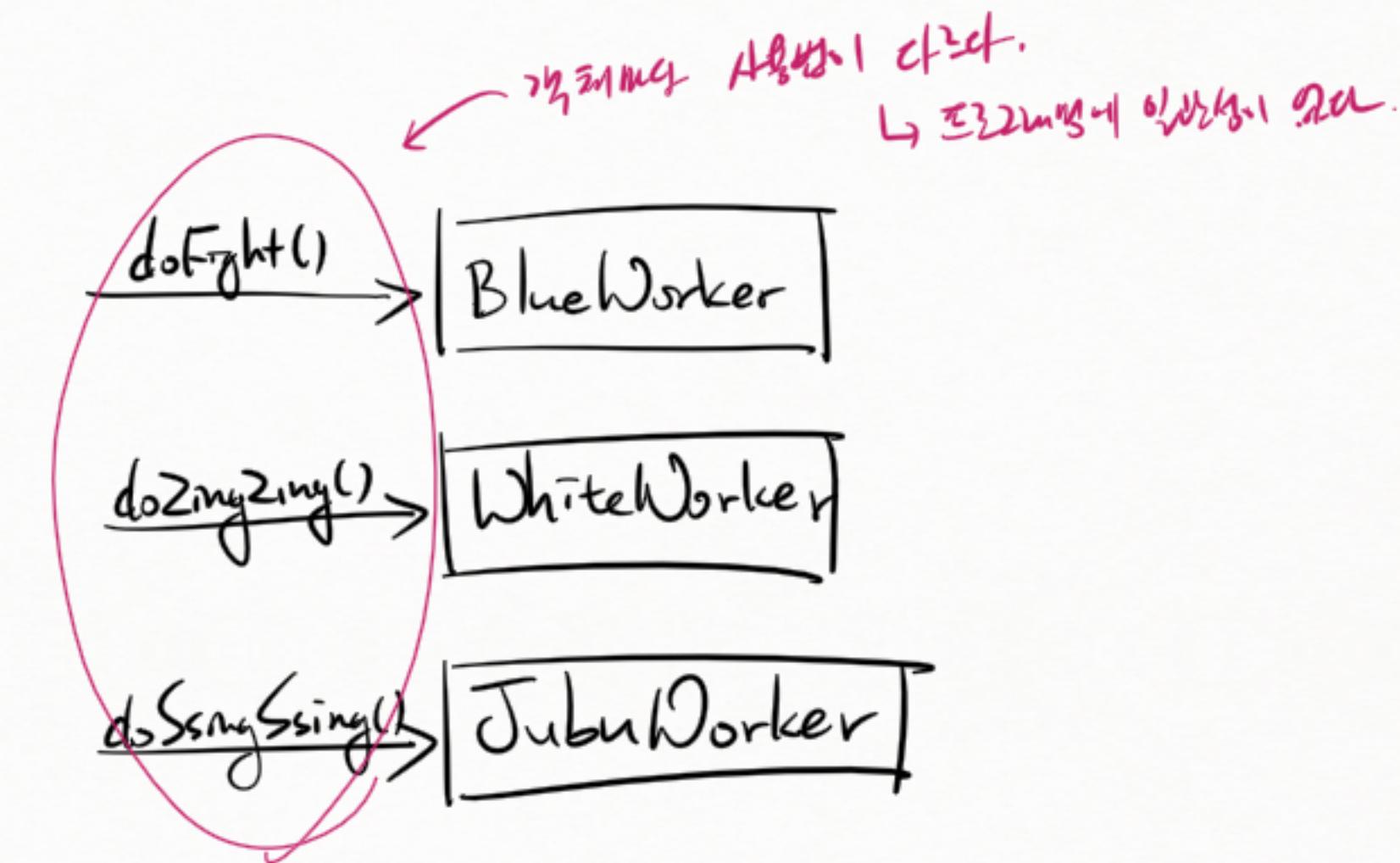
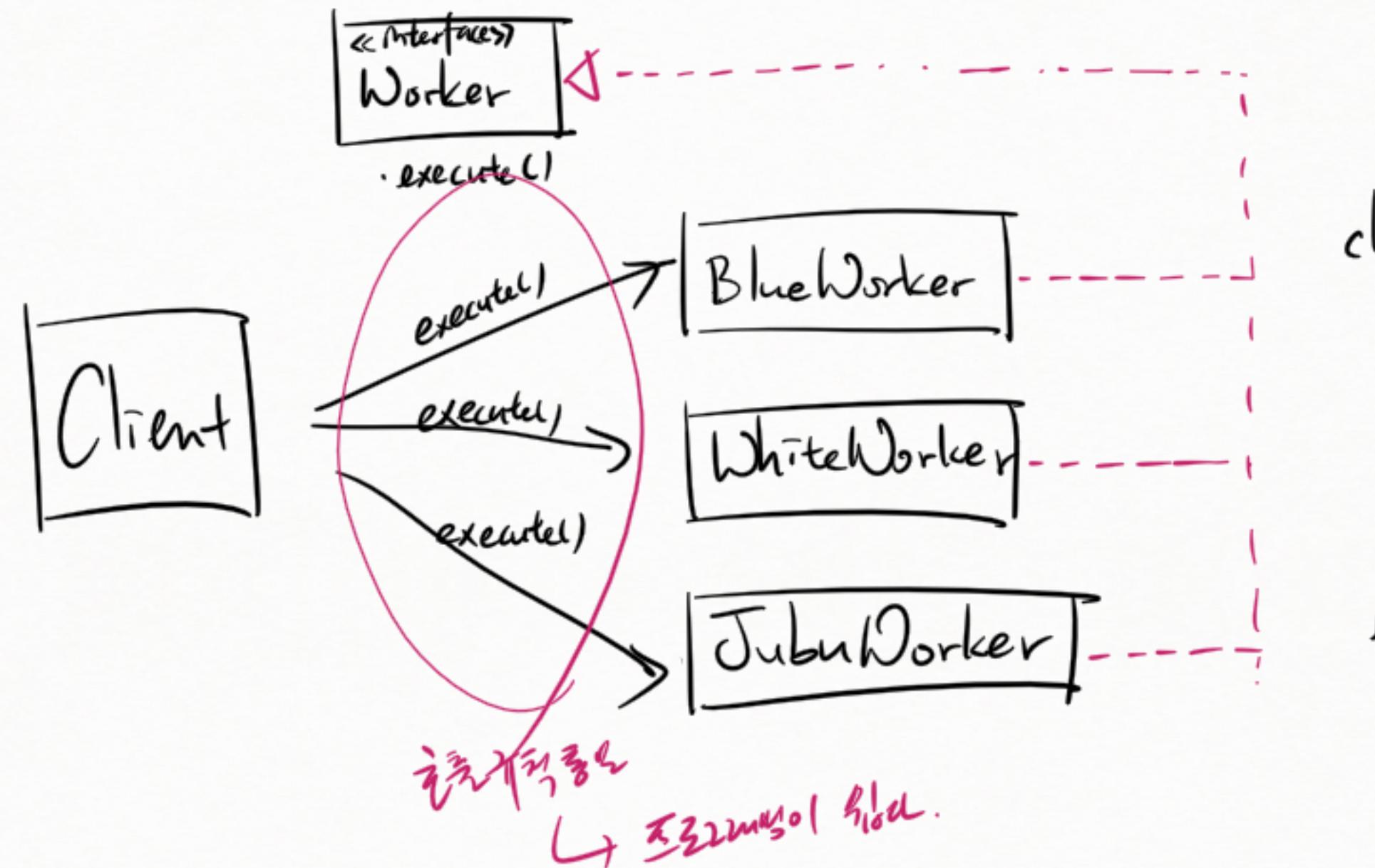


인터페이스 (Interface)

* 인터페이스 사용 전

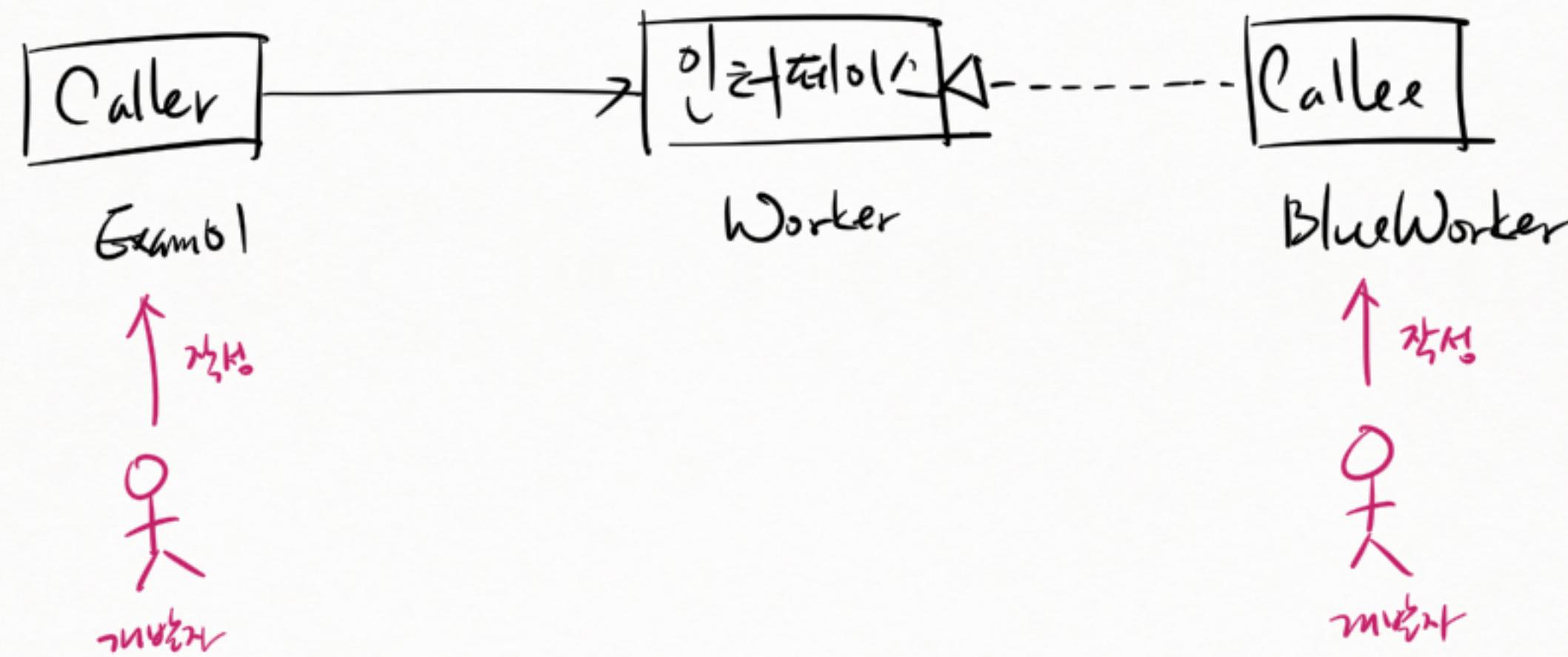


* 인터페이스 사용 후

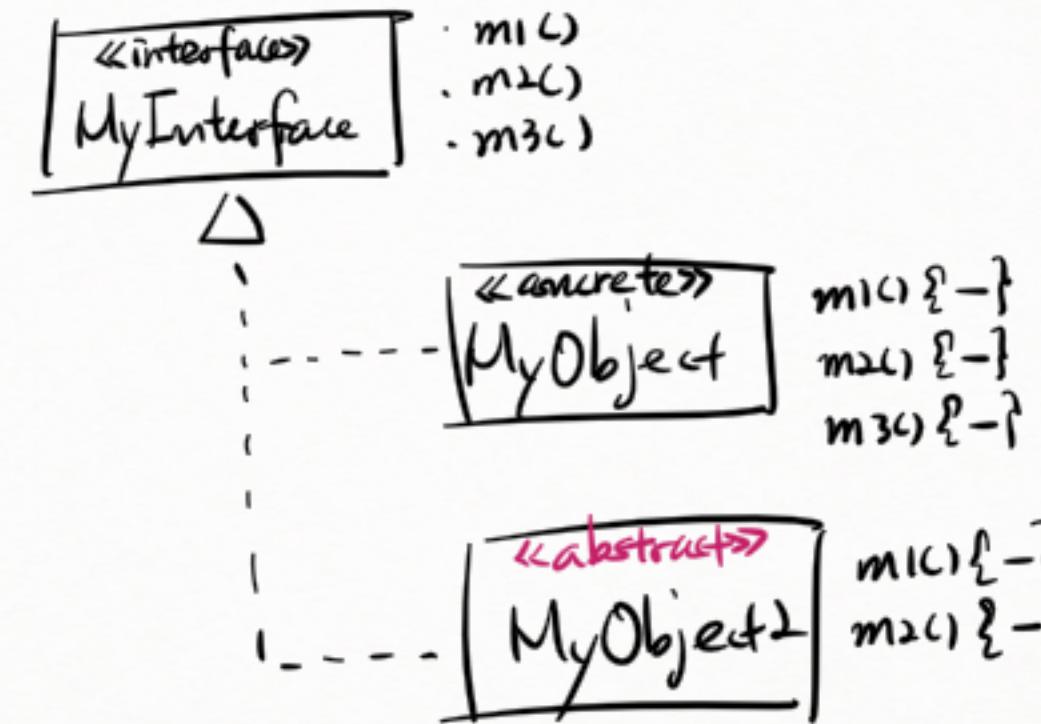


class BlueWorker
implements Worker {
 ...
}

* 인터페이스 와 caller / callee



* 인터페이스의 구현



MyInterface ref;

ref = new MyObject();

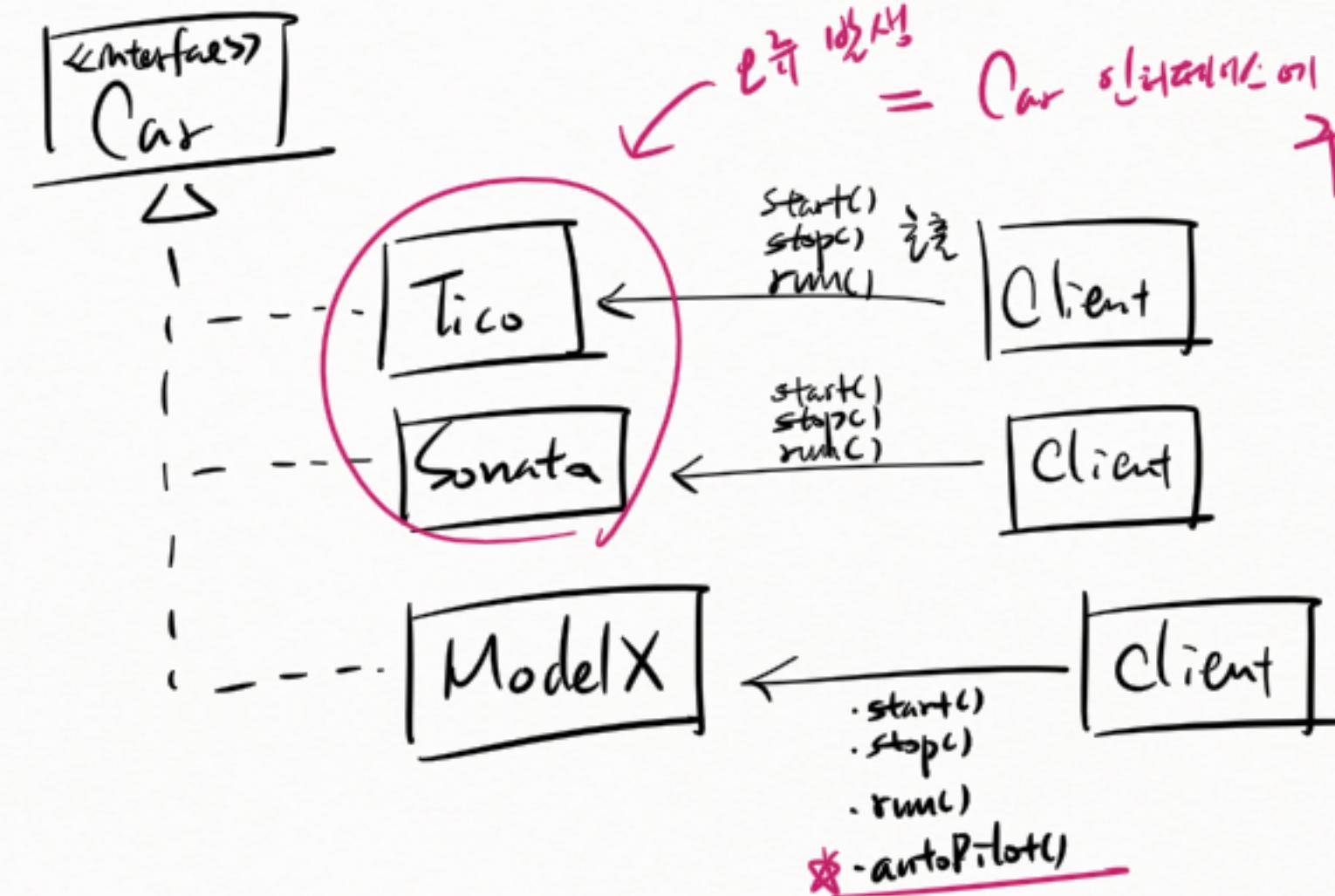


인터페이스를 구현한 클래스의
인스턴스화

* default 메서드 사용

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

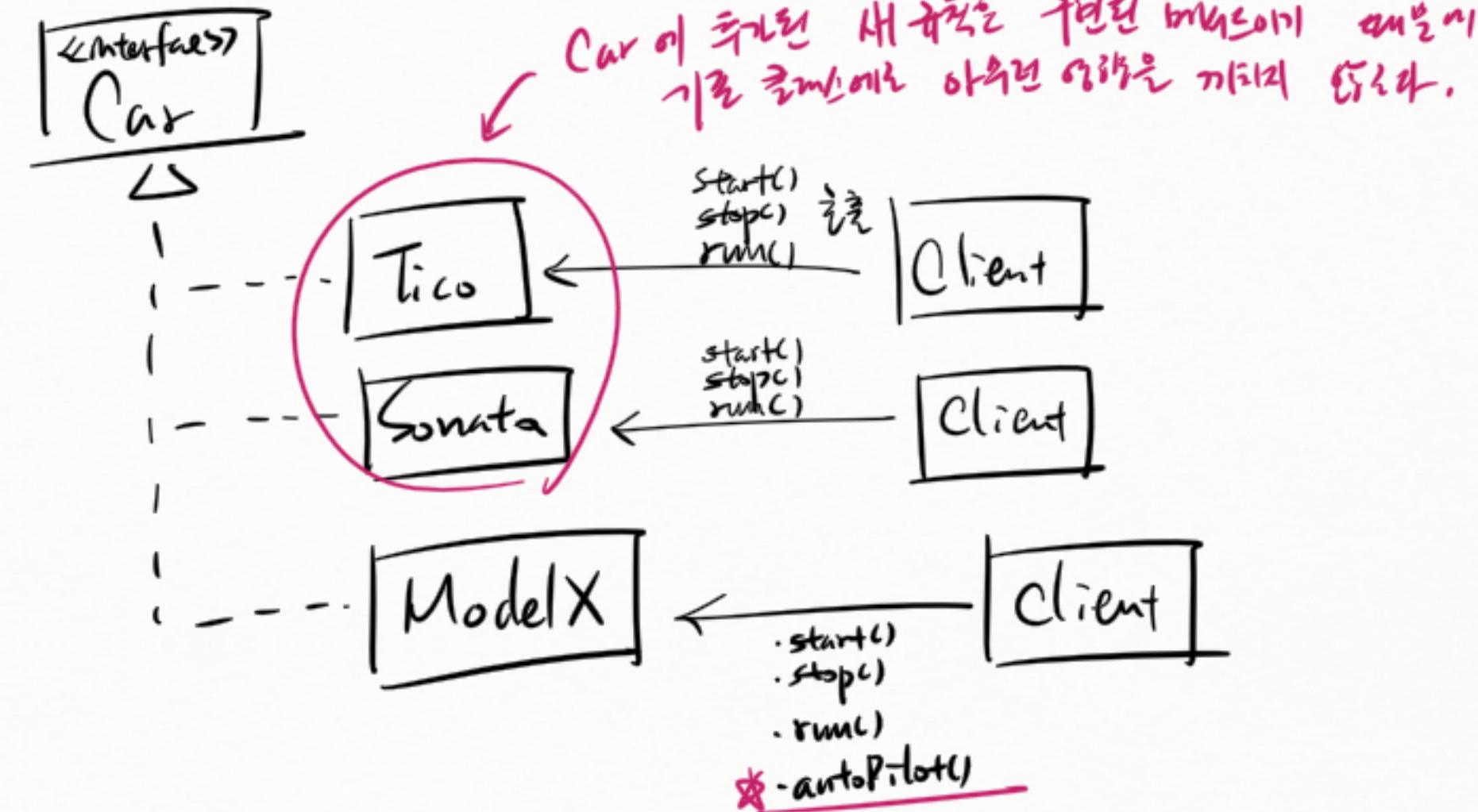
↑
구현 추가
+ autopilot();



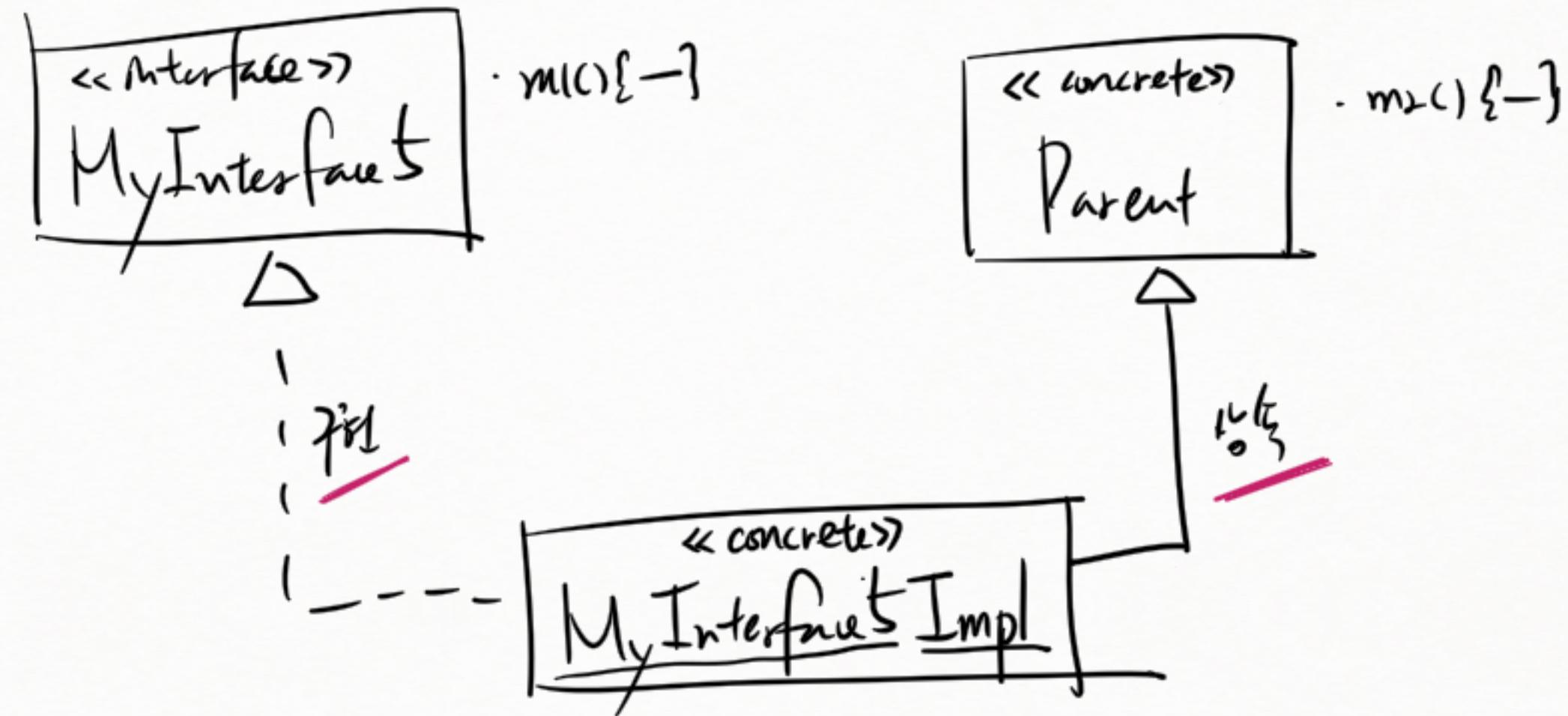
→ = Car 인터페이스에 추가된 새 기능은 구현(정의)되어 있겠지
증명이다.

* default interface 헬퍼

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

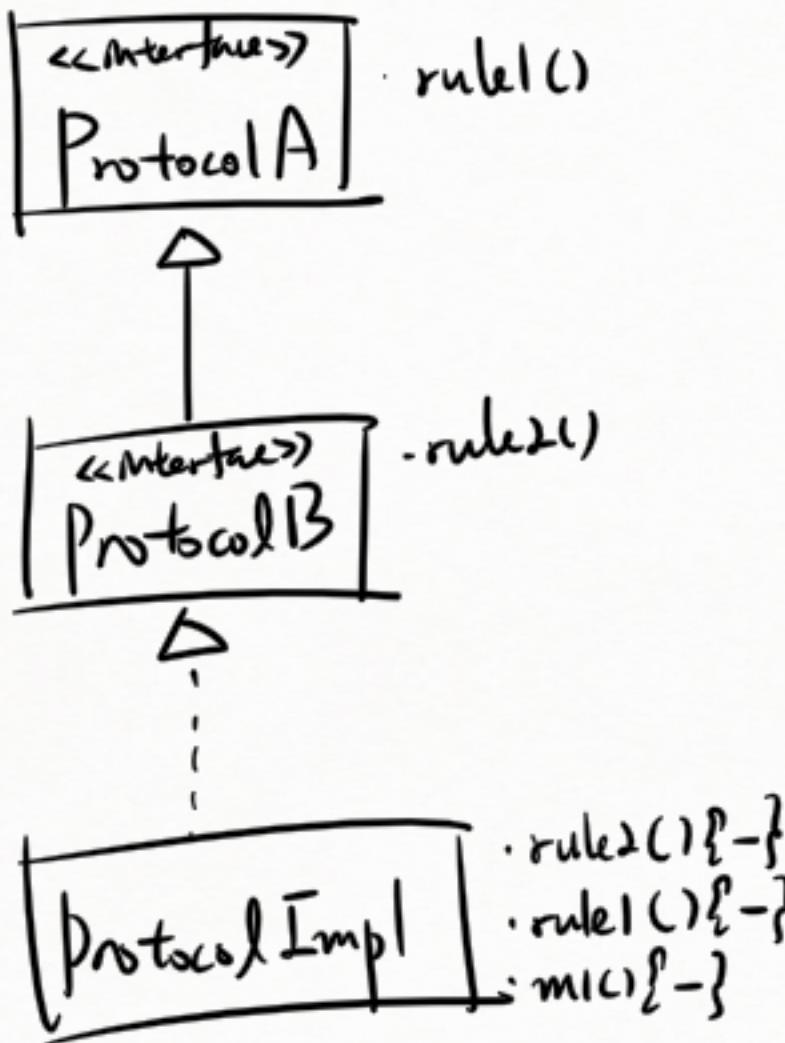


* static b1/b2



* 인터페이스 상속 구현

↳ 인터페이스를 통한 메서드 호출 방식



ProtocolImpl obj = new ProtocolImpl();

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

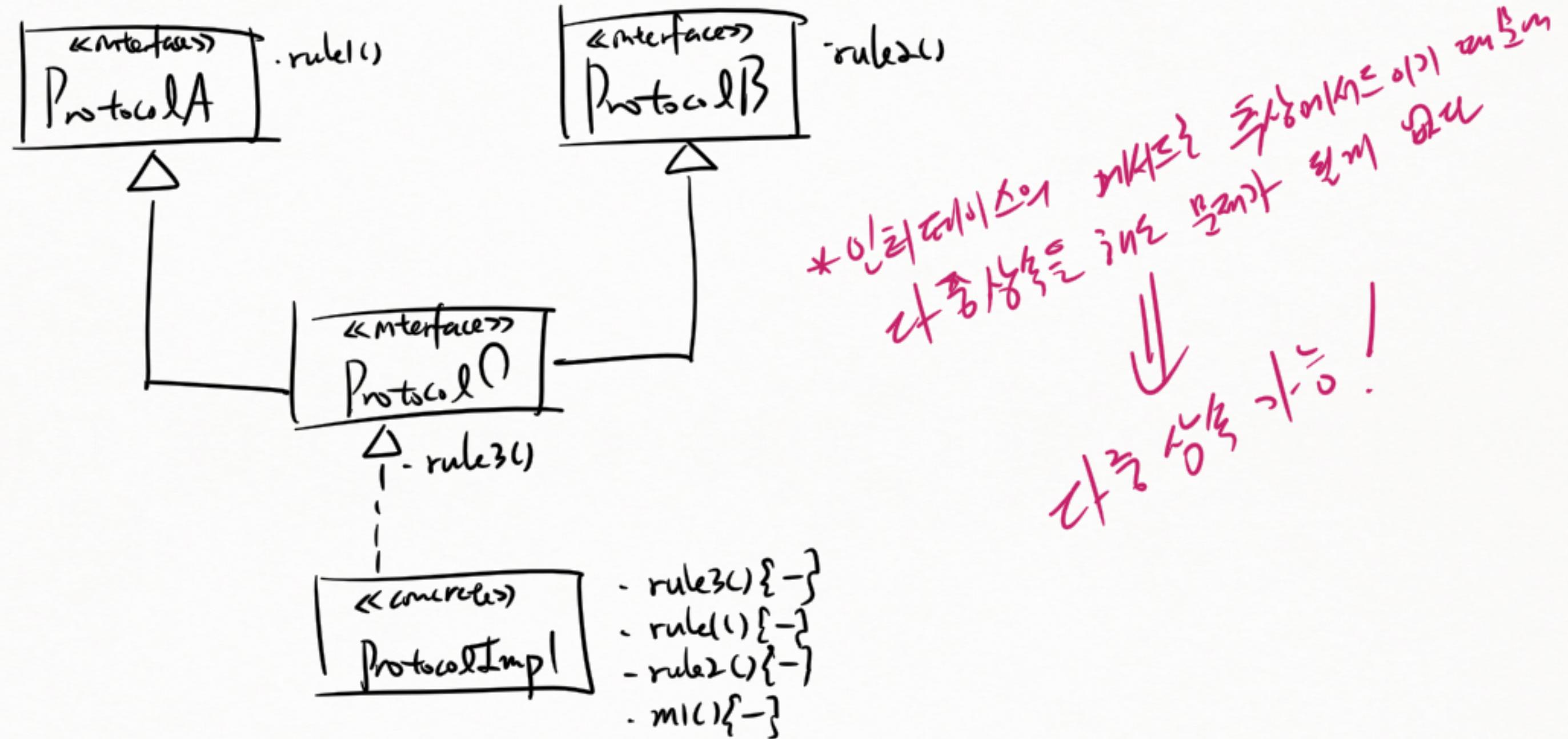
ProtocolB obj2 = obj;

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

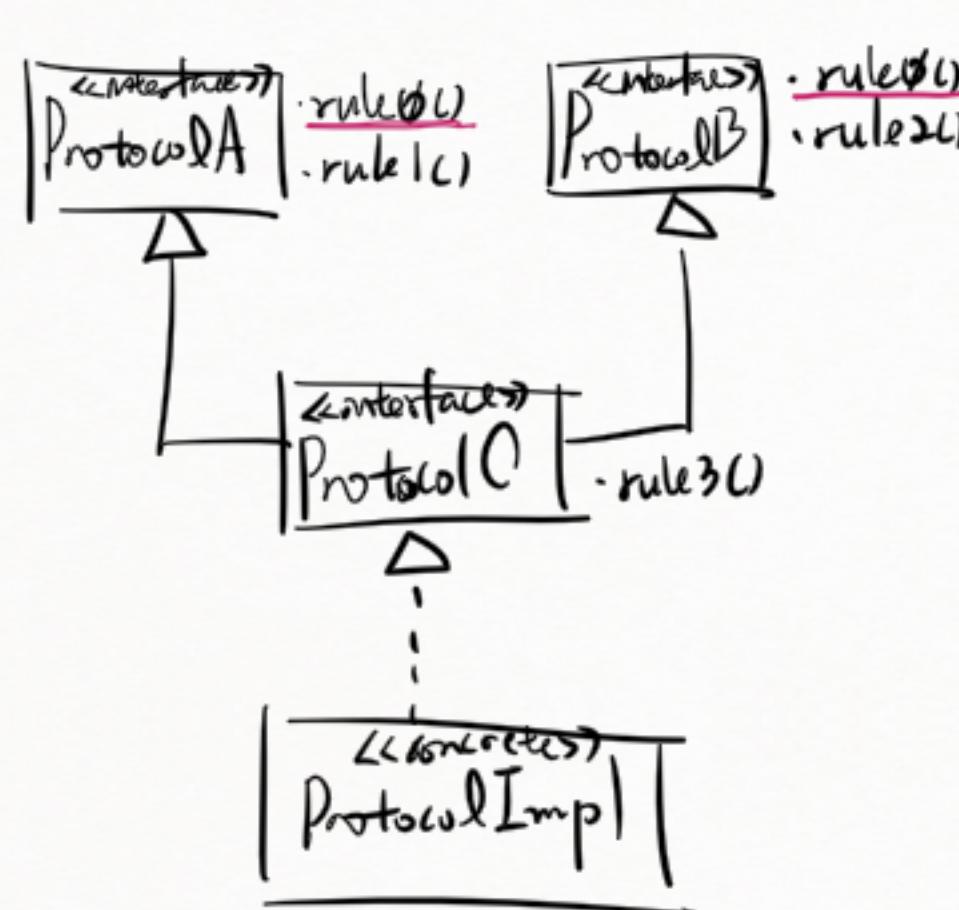
~~obj3. m1();~~
~~obj3. rule2();~~
~~obj3. rule1();~~

ProtocolA obj3 = obj;

* 인터페이스 다중 상속



* 이전에는 다음과 같은
가능한 경우

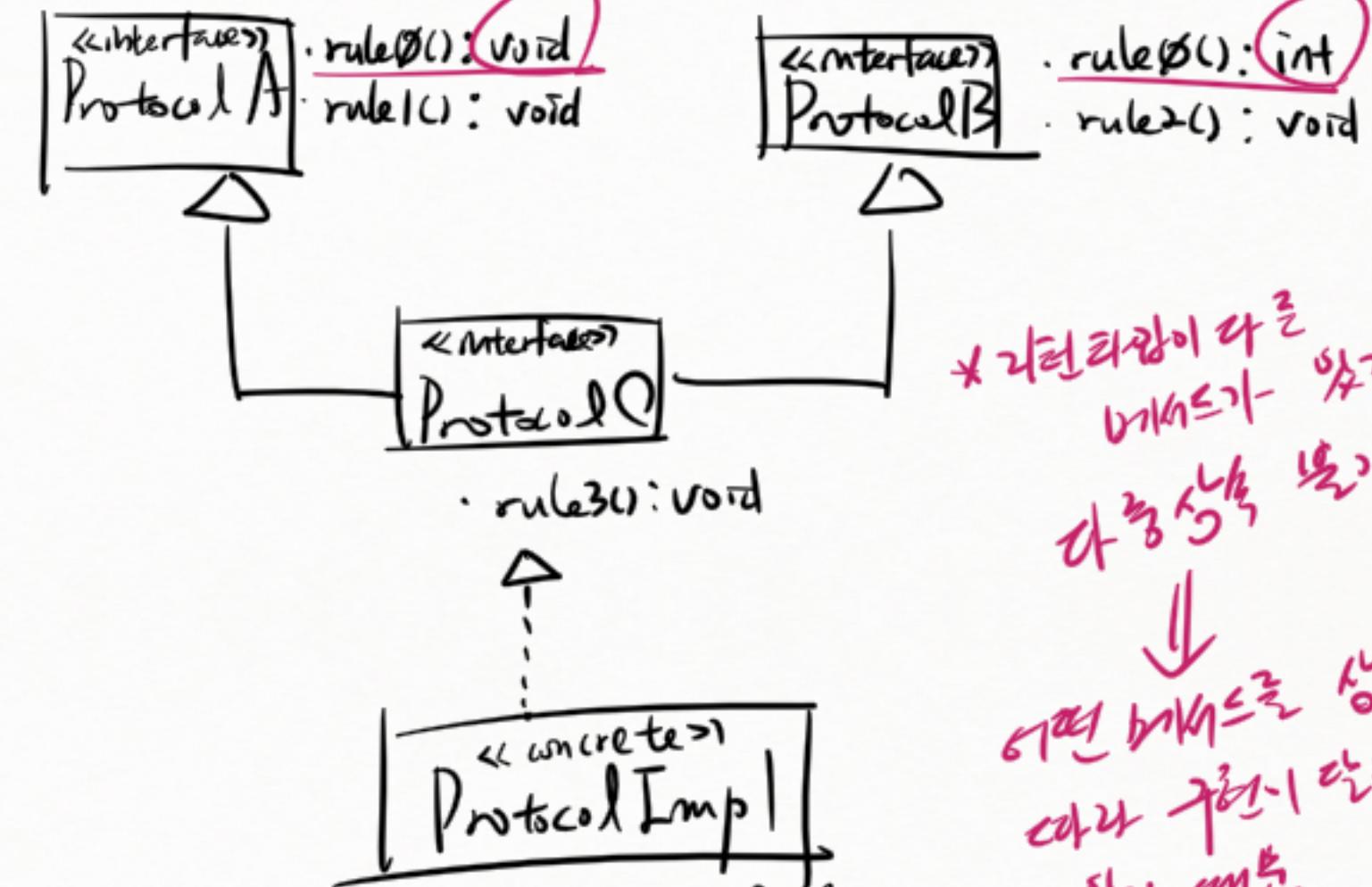


ProtocolA
rule0()
rule1()

ProtocolB
rule0()
rule2()

ProtocolC
rule3()

불가능한 경우



이전처럼 다를 수
없는 H2에 대한
부모가 다른 두
자식을 갖다.

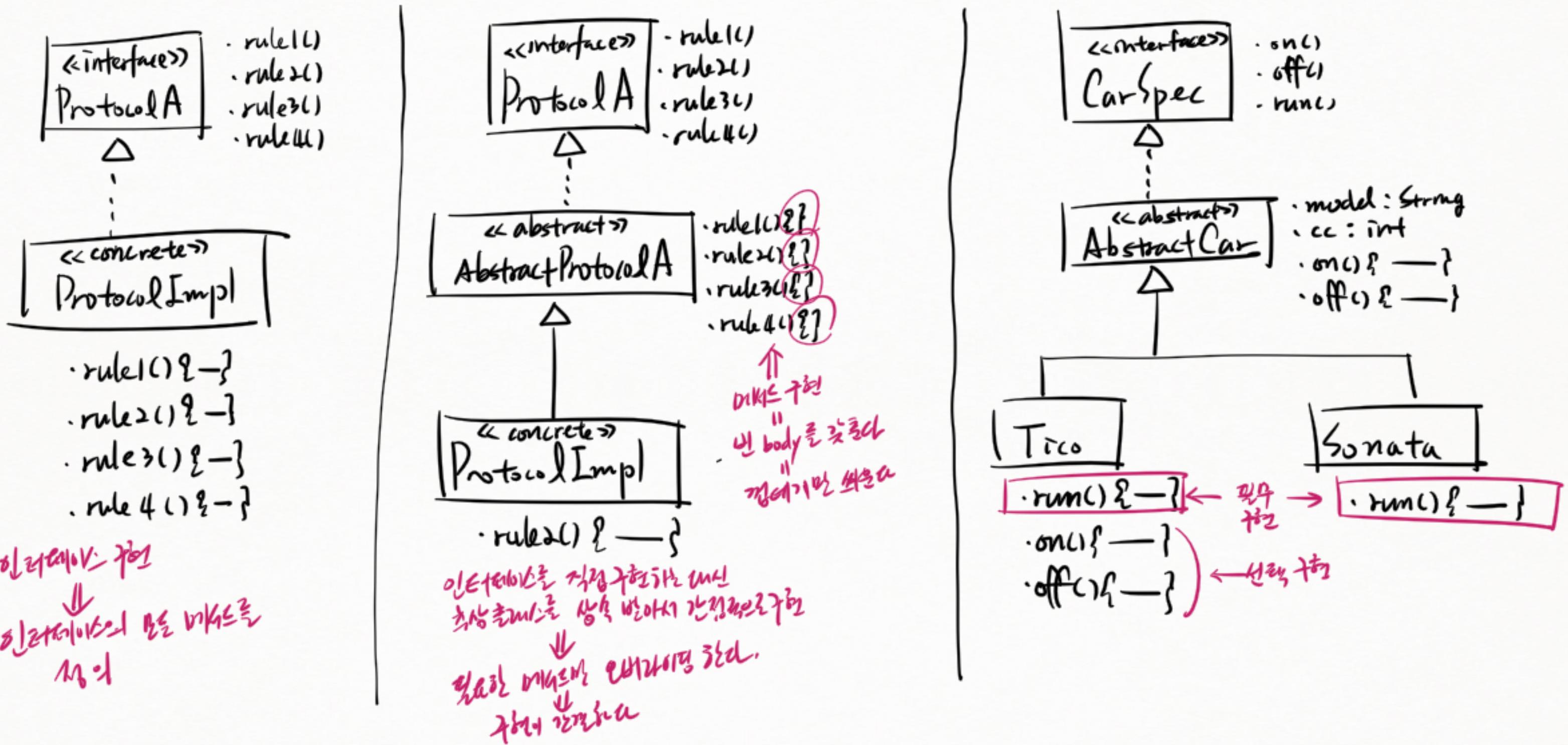
• void rule0(){ }
• int rule0(){ }

* 이런 이유로 다를 수
없는 부모가 있기 때문에
다른 자식을 두고
있지 않아!

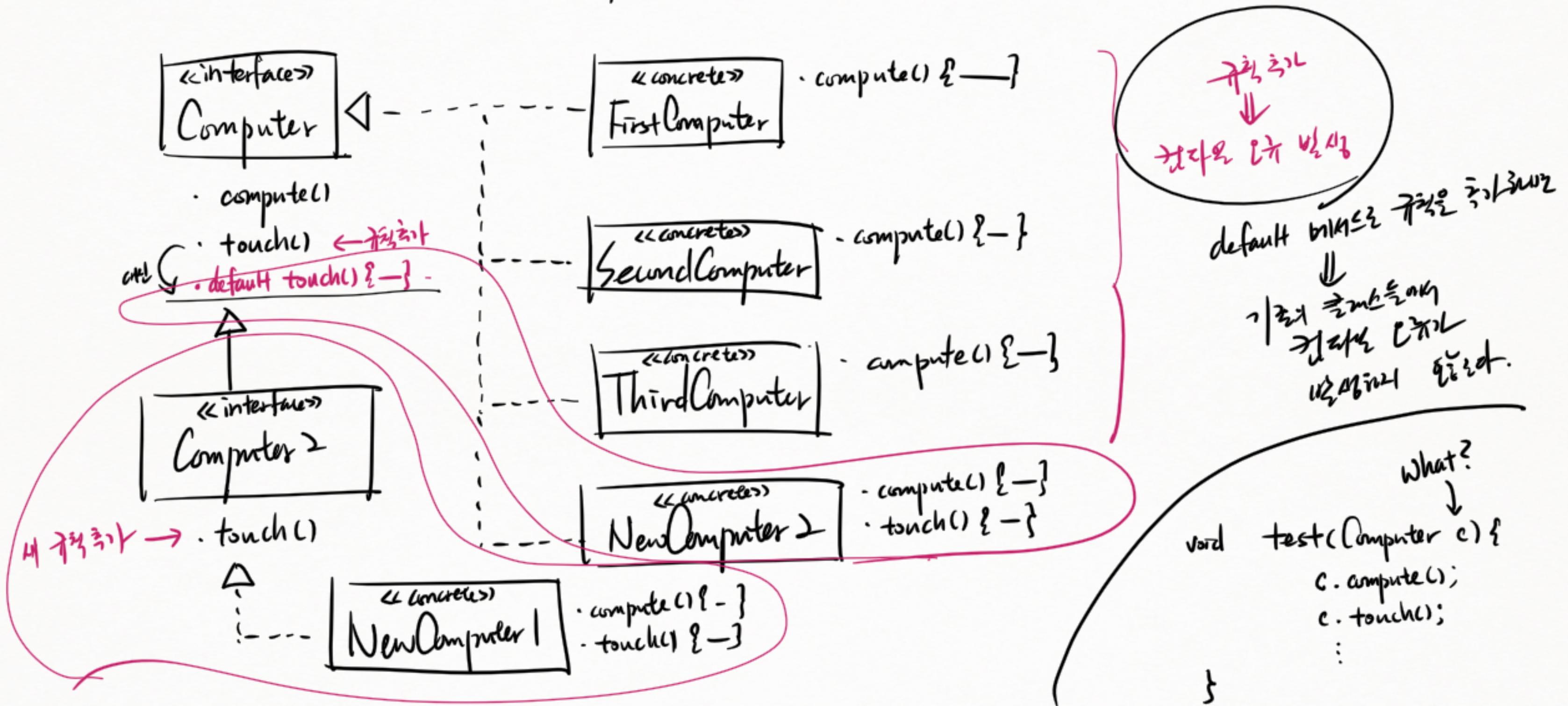
↓

여전히 부모는 상속 받는 부모
보다 자식이 더 많지
않기 때문

* 인터페이스와 추상 클래스



* 인터페이스의 default 메소드

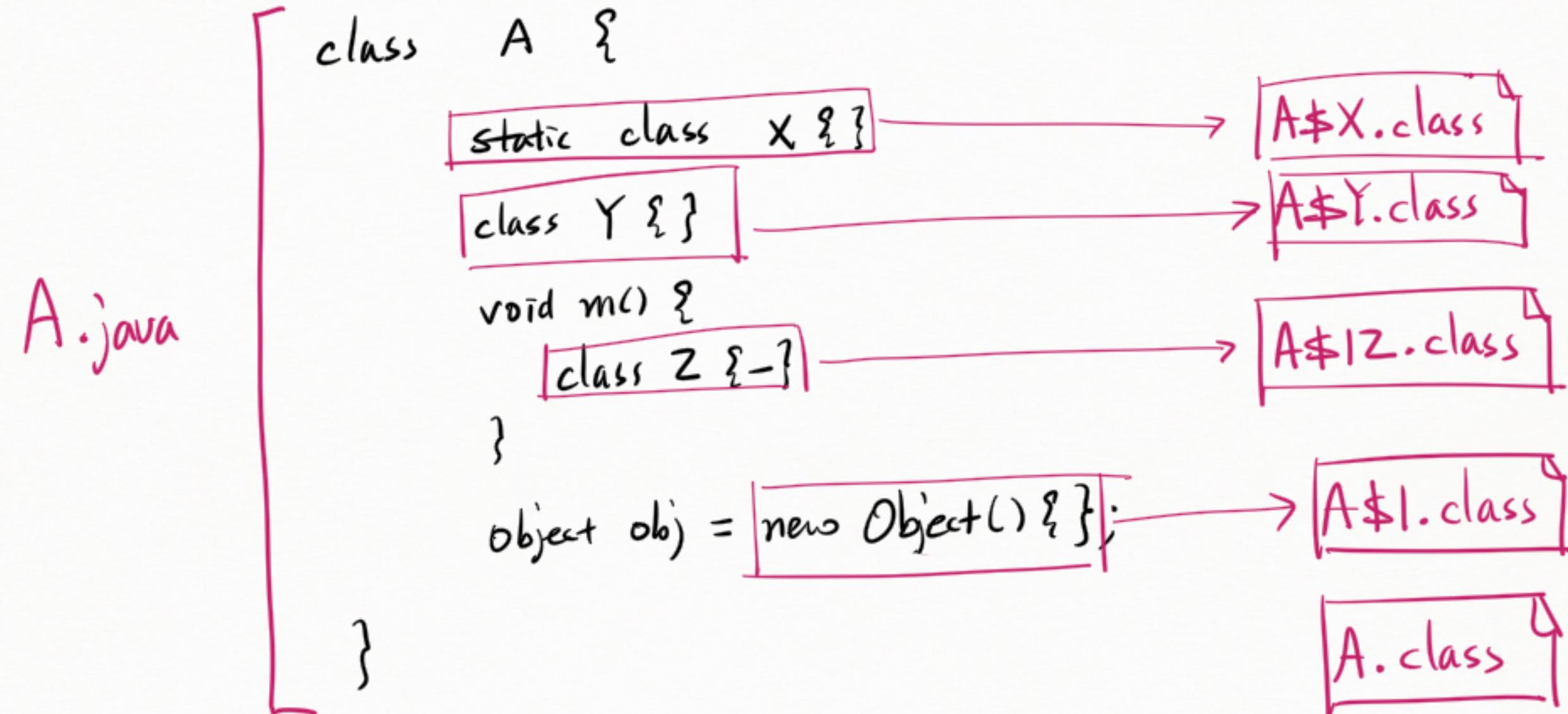


깊은
정적
클래스
(nested class)

* 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 클래스의 .class 파일



* static nested class 와 non-static nested class의 차이

class A {

 static class X { }

 class Y { }

}

Y obj = new Y();

 ↓
 new Y(this);

 ↳ Initialization
 비깥 클래스의 객체 주소
 인자는 반드시 사용 변수

static class X {
 X() { }
}

class Y {

 A this\$0;

 Y(A arg) {

 this\$0 = arg;

 }

 ↳ 전파일러가 기본 생성자 추가

 ↳ 전파일러,

 비깥 클래스의
 객체 주소를 받을
 레퍼런스를 추가.

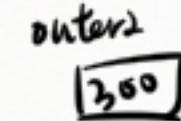
 ↳ 빙성자에서
 비깥 클래스의 인스턴스를
 받을 수 있도록
 파라미터를 추가

B3 outer = new B3();

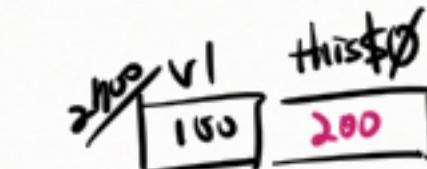
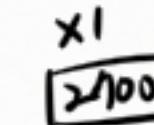


outer.v1 = 11;

B3 outer2 = new B3();



B3.X x1 = ~~outer~~.new X();



x1.test();
↑ this

B3.X x2 =

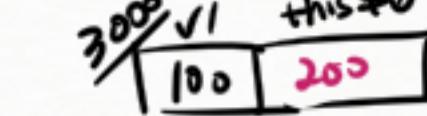


x2.test();
↑ this

B3.X x3 =



~~outer~~.new X();



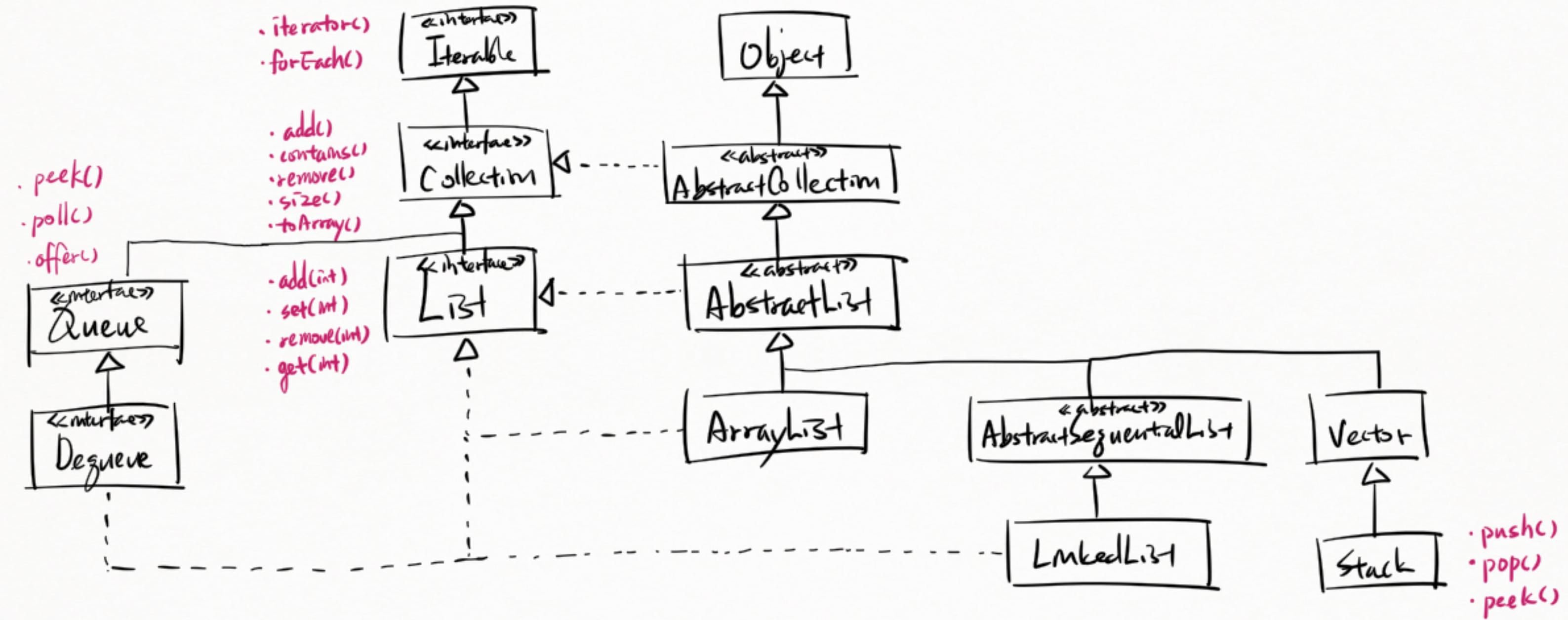
~~outer2~~.new X();



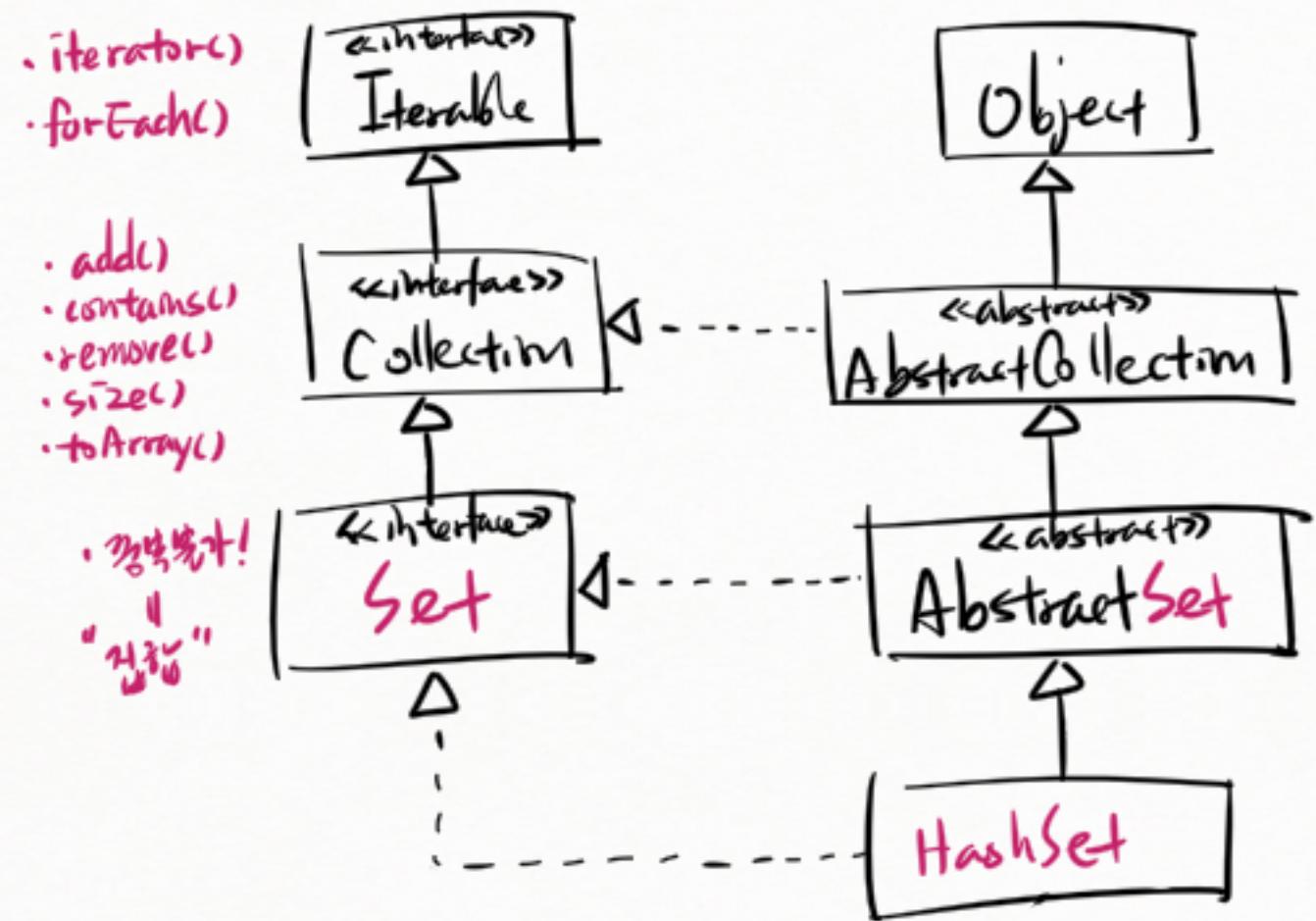
450
x3.test();

Collection API

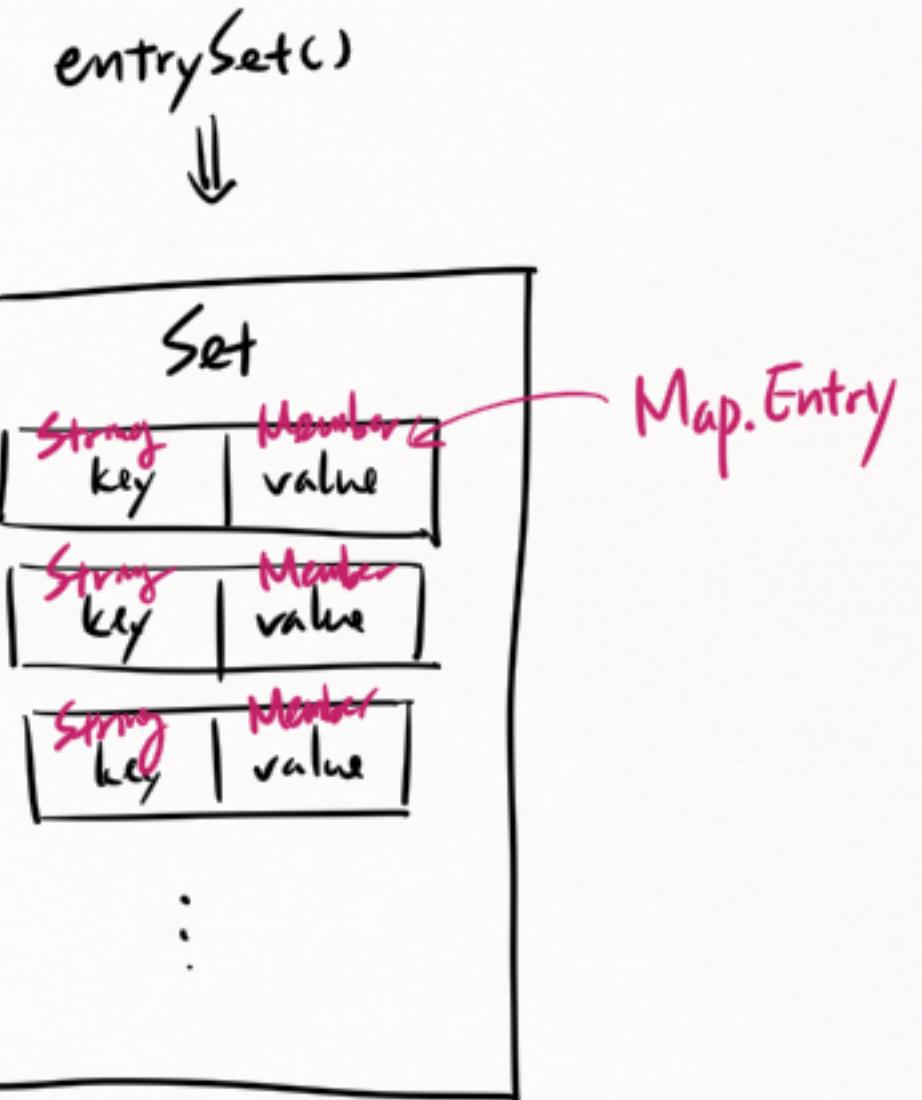
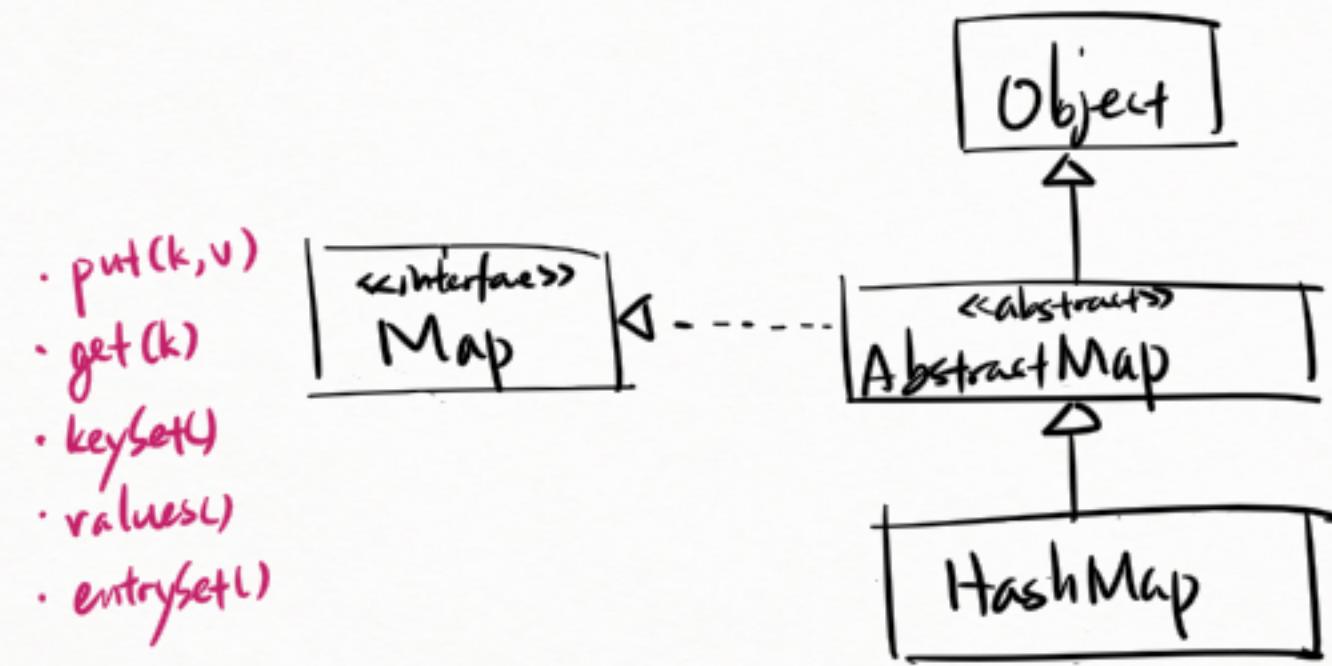
* List の構成と階層 (hierarchy)



* Set သုတေသနများ ရှိခိုင်း (hierarchy)

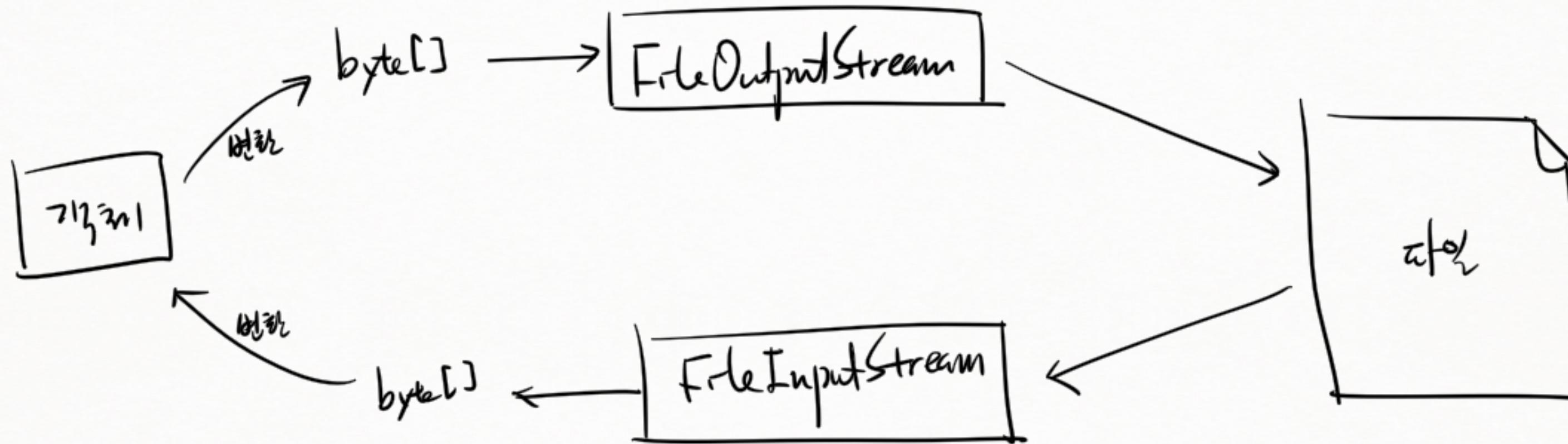


* Map の構成と階層構造 (hierarchy)



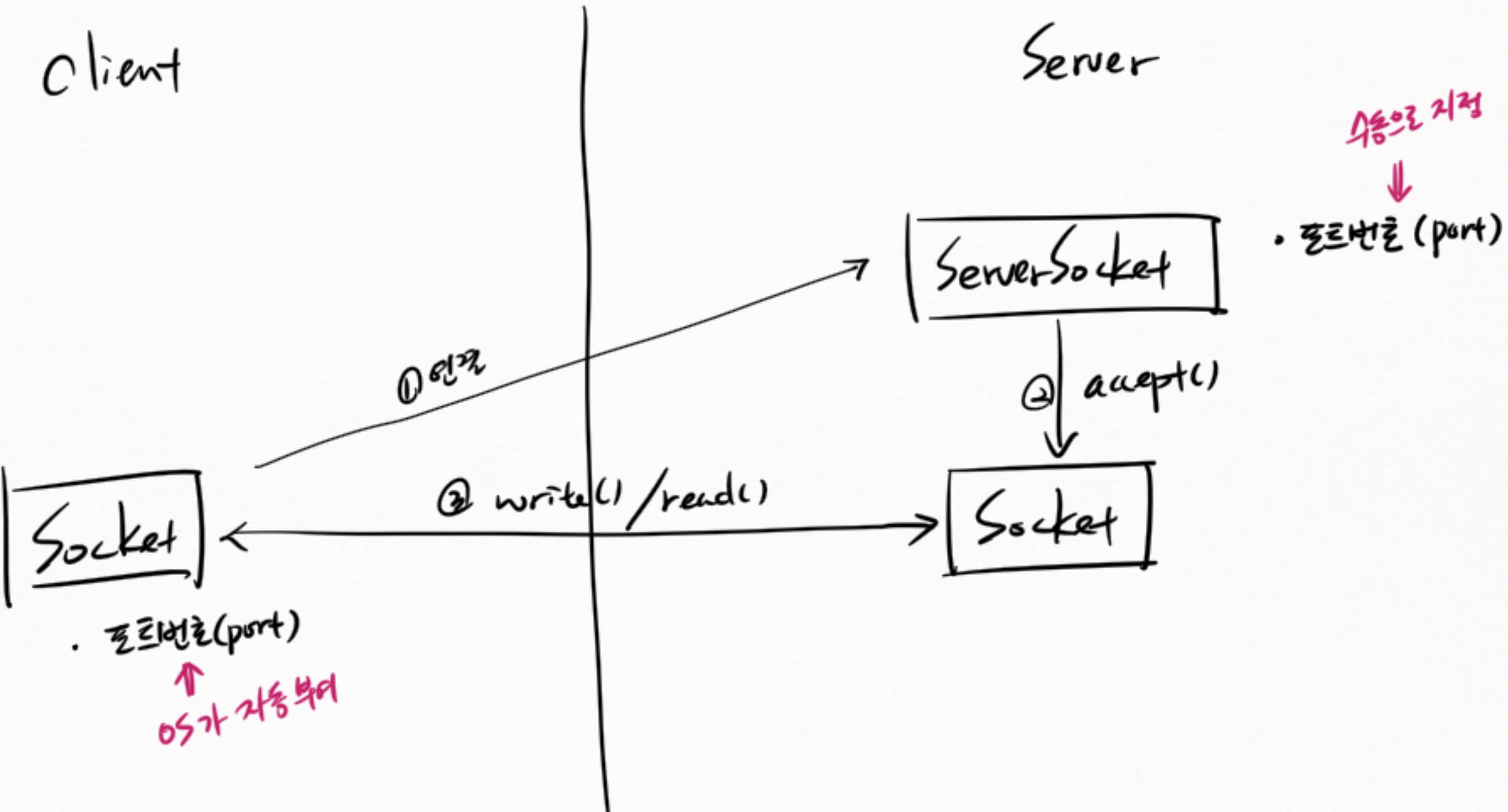
File I/O API

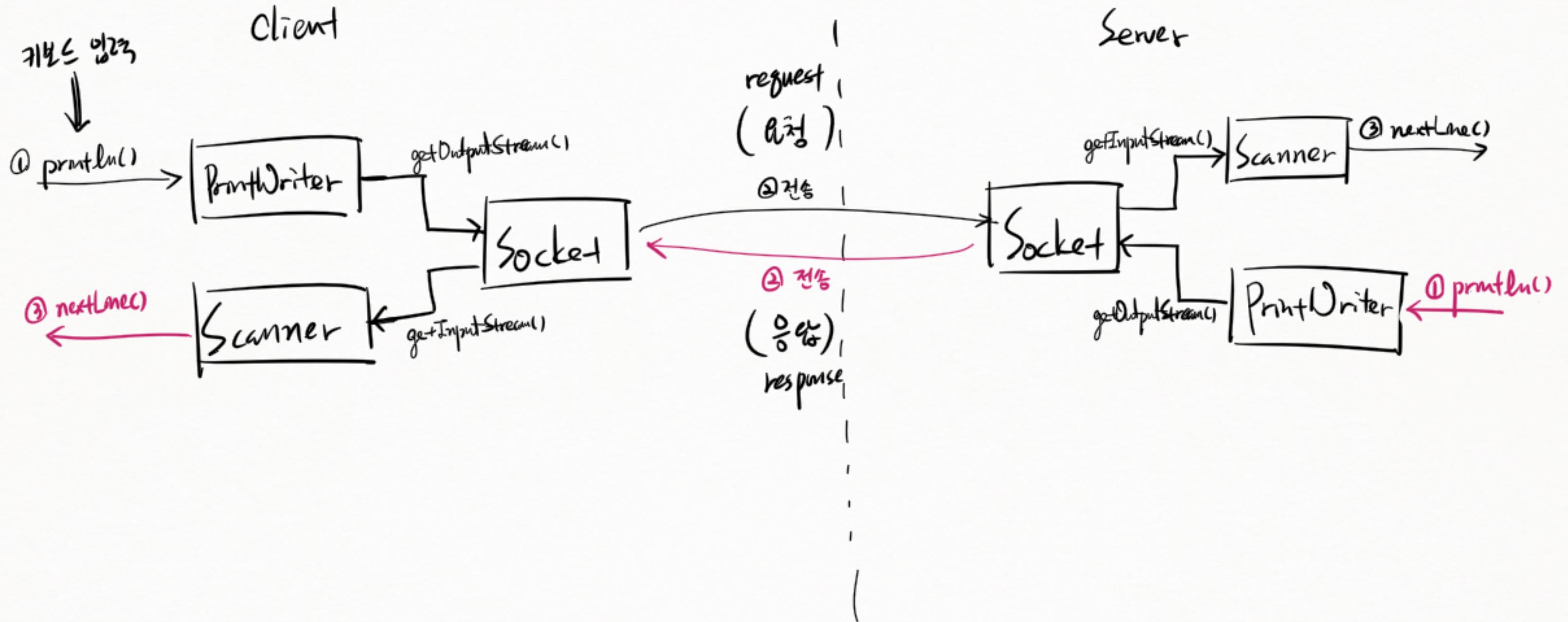
* FileOutputStream / FileInputStream



Networking

Client / Server





* Networking 연결 방식

Connection-Oriented (연결 지향)

예) 전화, WWW, LOL, 키글리드, 웹메일, 유튜브

프로토콜 \Rightarrow TCP

Connectionless (비연결 지향)

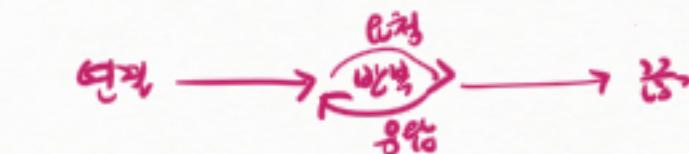
예) 네트워크, 대화, 멘시, ping

프로토콜 \Rightarrow UDP

Stateful

예) 상호전화

프로토콜: SSH, Telnet, FTP, 채팅



Stateless

예) 웹 앤내



* Connection-Oriented

- ✓ 단기적인 연결에만 사용을 한다
- ✓ 단기적인 후 데이터 전송 \Rightarrow 신뢰성 높음.

(TCP) HTTP 1/2

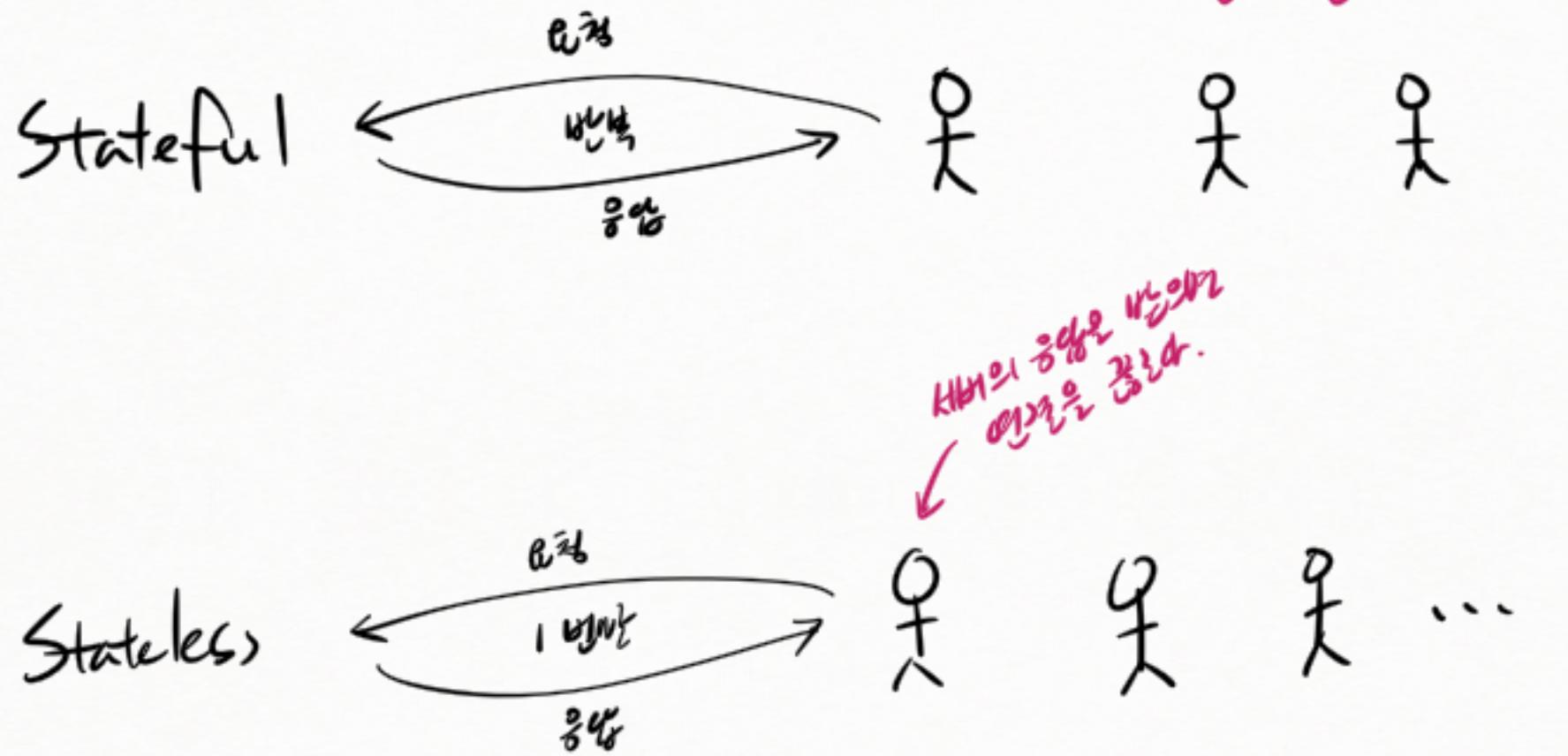
* Connectionless

↓

(UDP) HTTP 3

- ✓ 단기적인 접속 \Rightarrow 신뢰성이 낮음.
- ✓ 단기적인 접속 데이터 전송 \Rightarrow 신뢰성이 낮음.

* Stateful 허용하고 Stateless 허용하지



클라이언트의
응답을 허용하는
여기로 돌아온다.

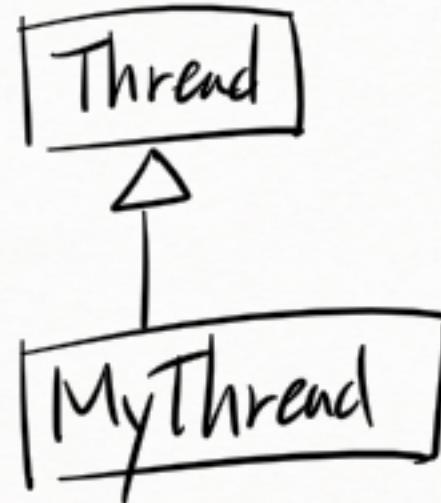
한 클라이언트가
한번 시킬 연결은 있어도
다른 클라이언트의
다른 시도가 가능하다.

한번 연결이
한번의 요청/응답만 처리
↓
다른 클라이언트의 연결 대기 시간이
있다.
↓
어떻든 클라이언트의 요청을 처리

Thread

* 스레드 생성 및 실행

① 키트레드 만들기



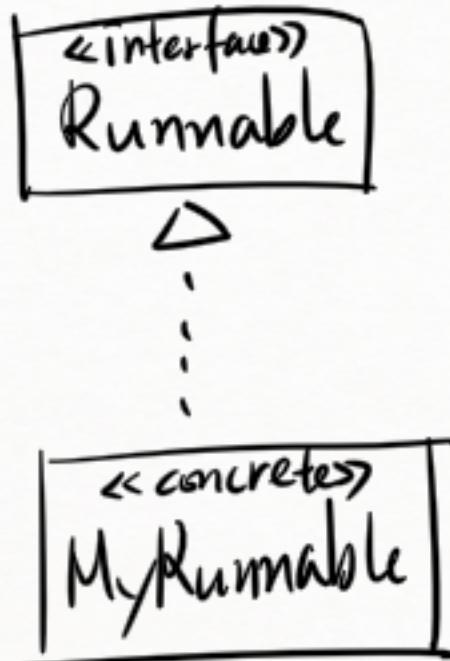
```
MyThread t = new MyThread();
t.start();
```

② 일반 클래스 키트레드 만들기

```
new Thread() {
    run() {
    }
}.start();
```

* 스레드 생성 및 실행

③ Runnable 향상 활용



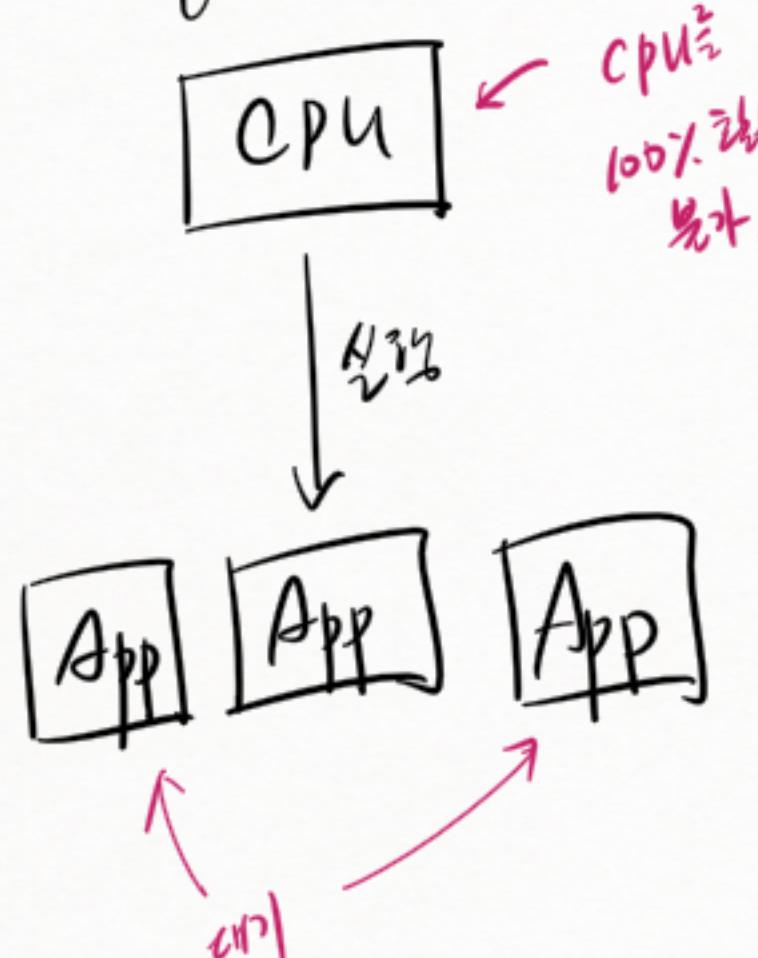
```
MyRunnable r = new MyRunnable();
Thread t = new Thread(r);
t.start();
```

④ 빌드 편의를 위한 Runnable 향상

```
new Thread( new Runnable() {
    run() {
    }
}).start();
```

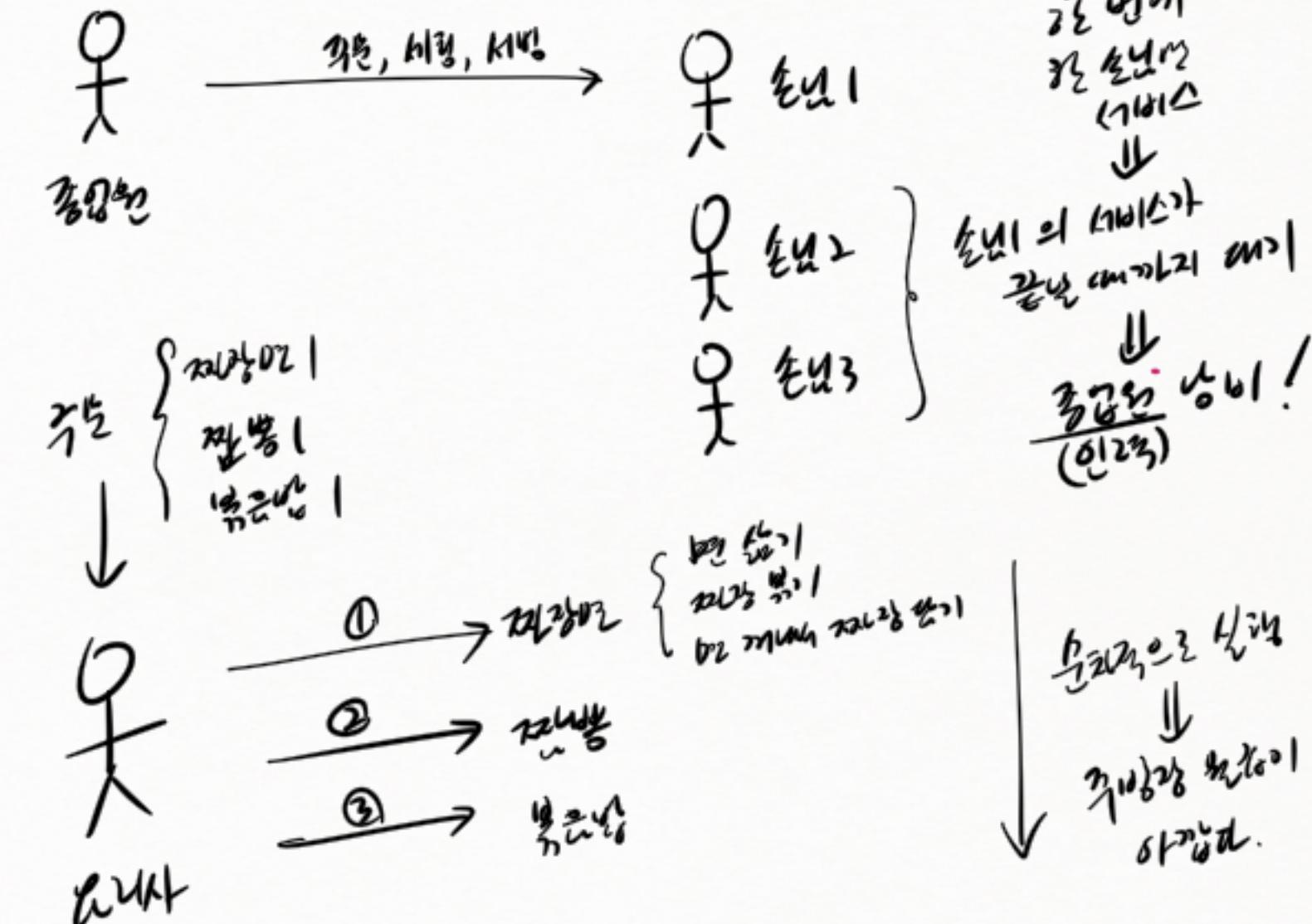
* 멀티 태스킹 (Multi-tasking)

① Single-tasking



❶ DOS

H27(1)



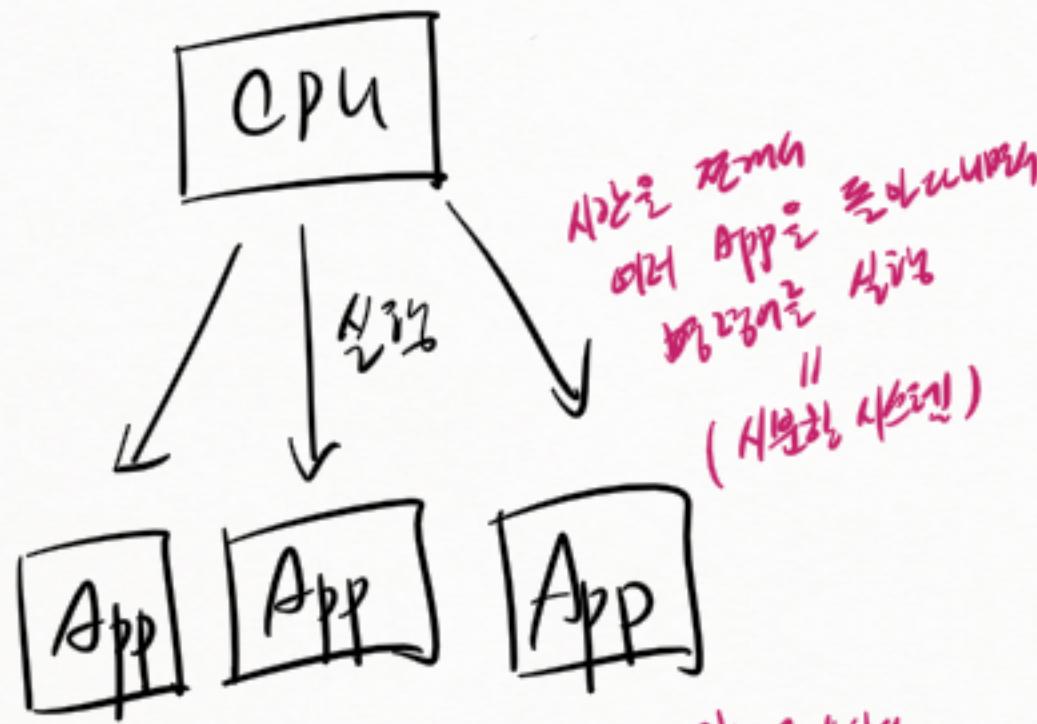
H27(2)



작업을 나누는 경우!
작업을 나누는 경우!

* 멀티 태스킹 (Multi-tasking)

② Multi-tasking



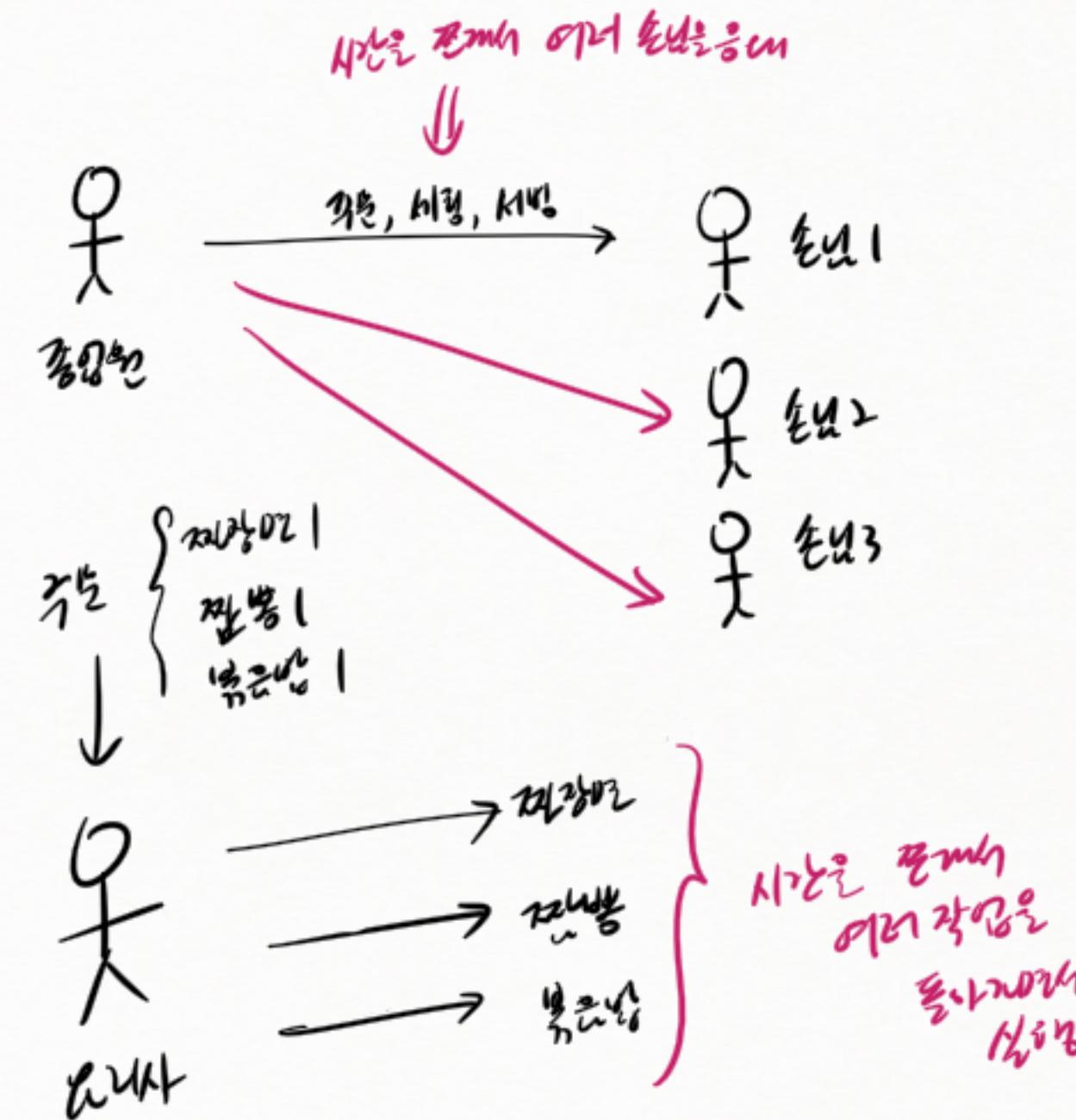
여기 App.ei 헤 놓았던 것처럼 봐야

81) Windows

Unit

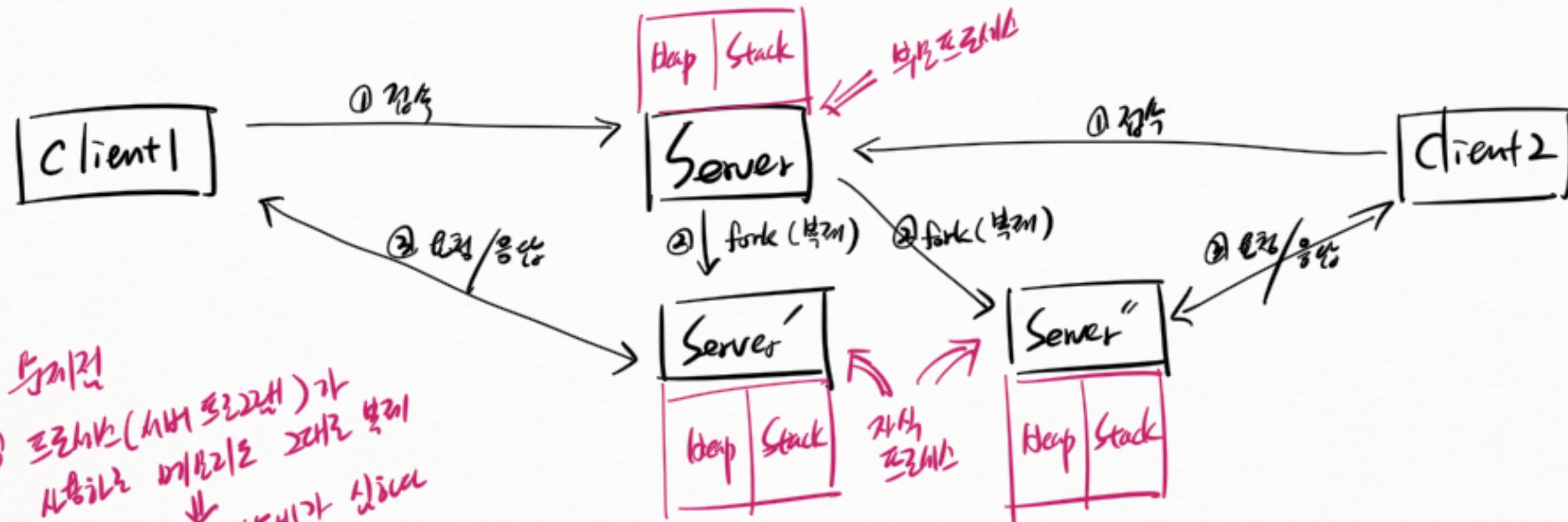
H2011

11212



* 자동 클라이언트 요청 처리에 따라 인터넷 사용

① 프로세스 복제 기법 (fork())



★ 특징

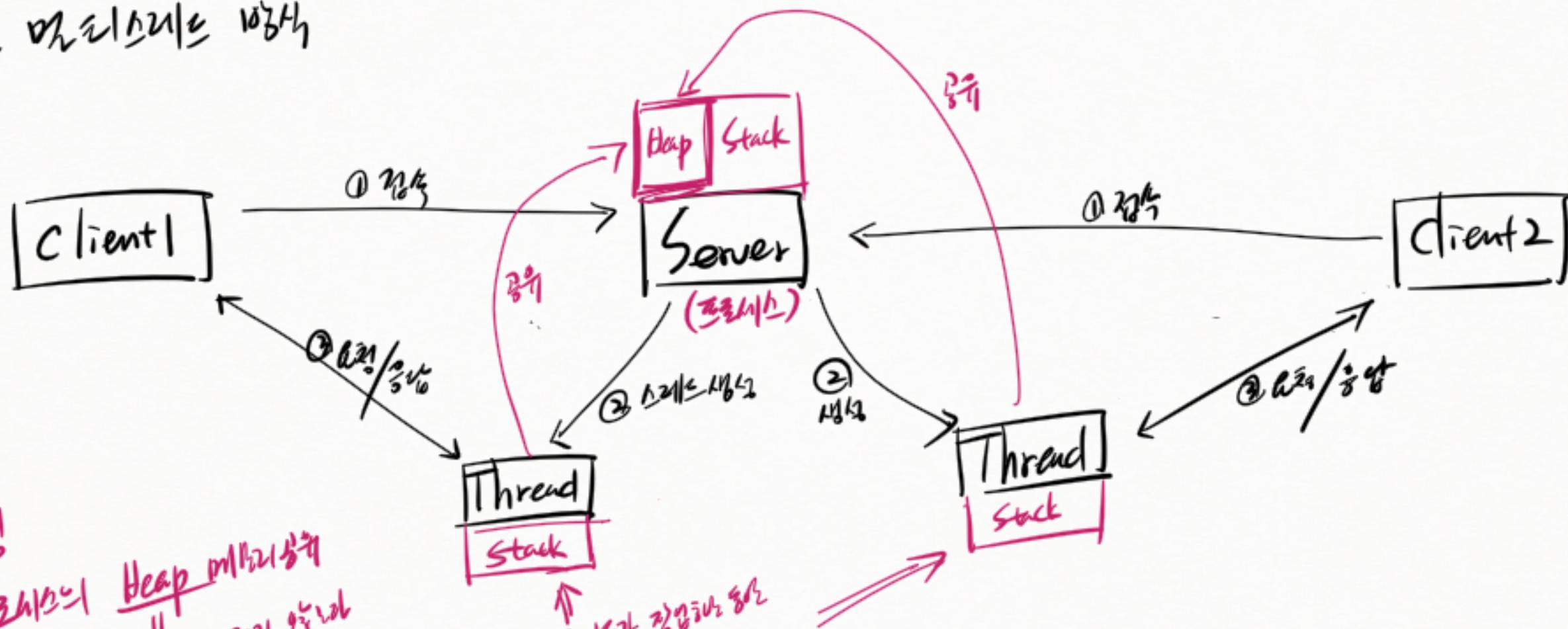
① 프로세스(서비스 프로세스)가
할당된 메모리를 그대로 물려
받게 되는 특성이 있다

기존의 쓰레드를 통한 대체로
자신 프로세스를 통한 대체로

② 부모 프로세스는 종료되거나
자신 프로세스는 종료되거나
부모와 자식 프로세스를 동일하게 만들 수 있다.

* 다음 클라이언트 요청 처리에 맞춰 구현한 내용

② 미리스리스 와트



* 320

① 프로세스의 Heap은 멘리에 있습니다.

11월의 ~~날씨~~
11월의 날씨
↓
11월의 날씨

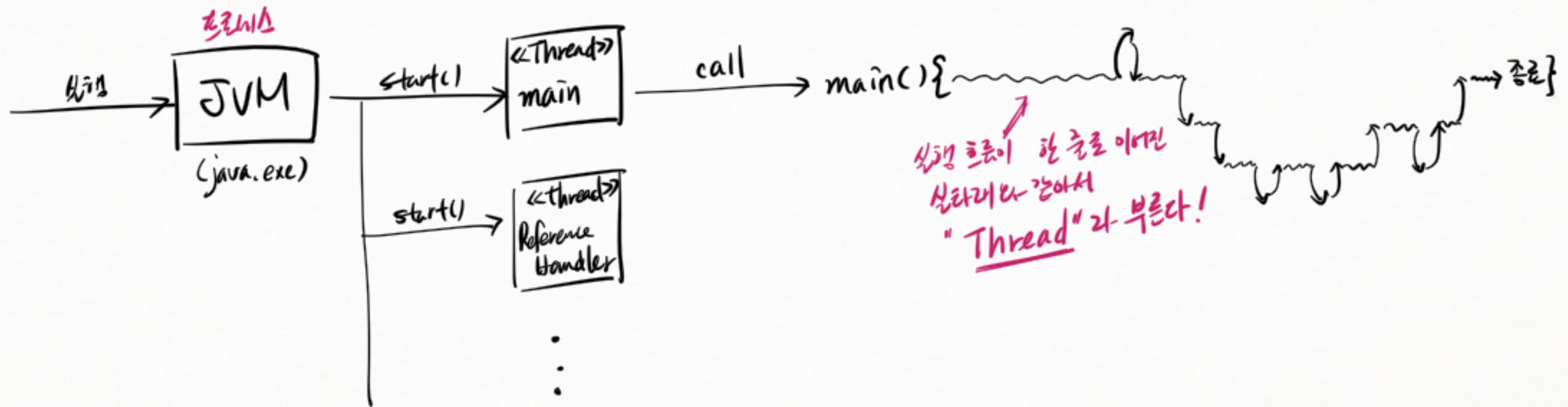
143112 304,48466
↓
143112 304,48466
→ 143112 304

\Rightarrow $\text{Date } \frac{3}{3}^{\text{rd}}$ of Oct

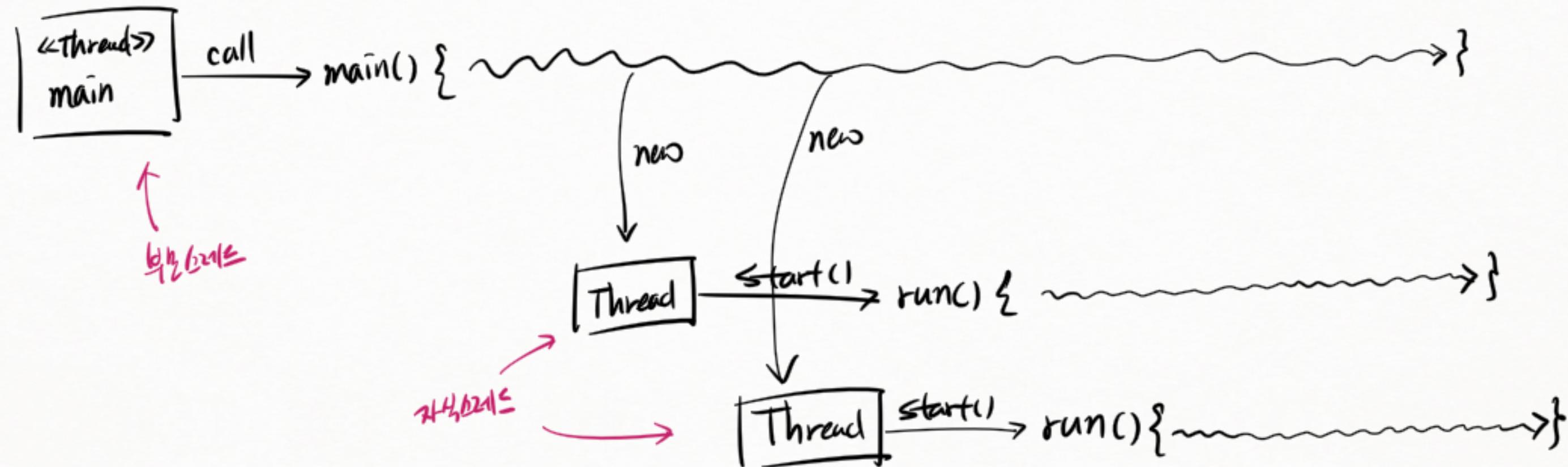
② $\text{H}_2\text{O}_2 \xrightarrow{\text{Enzyme}} \text{H}_2\text{O} + \text{O}_2$
Enzyme 총동작

H_2O_2 를 만드는 과정

* JVM 과 스레드

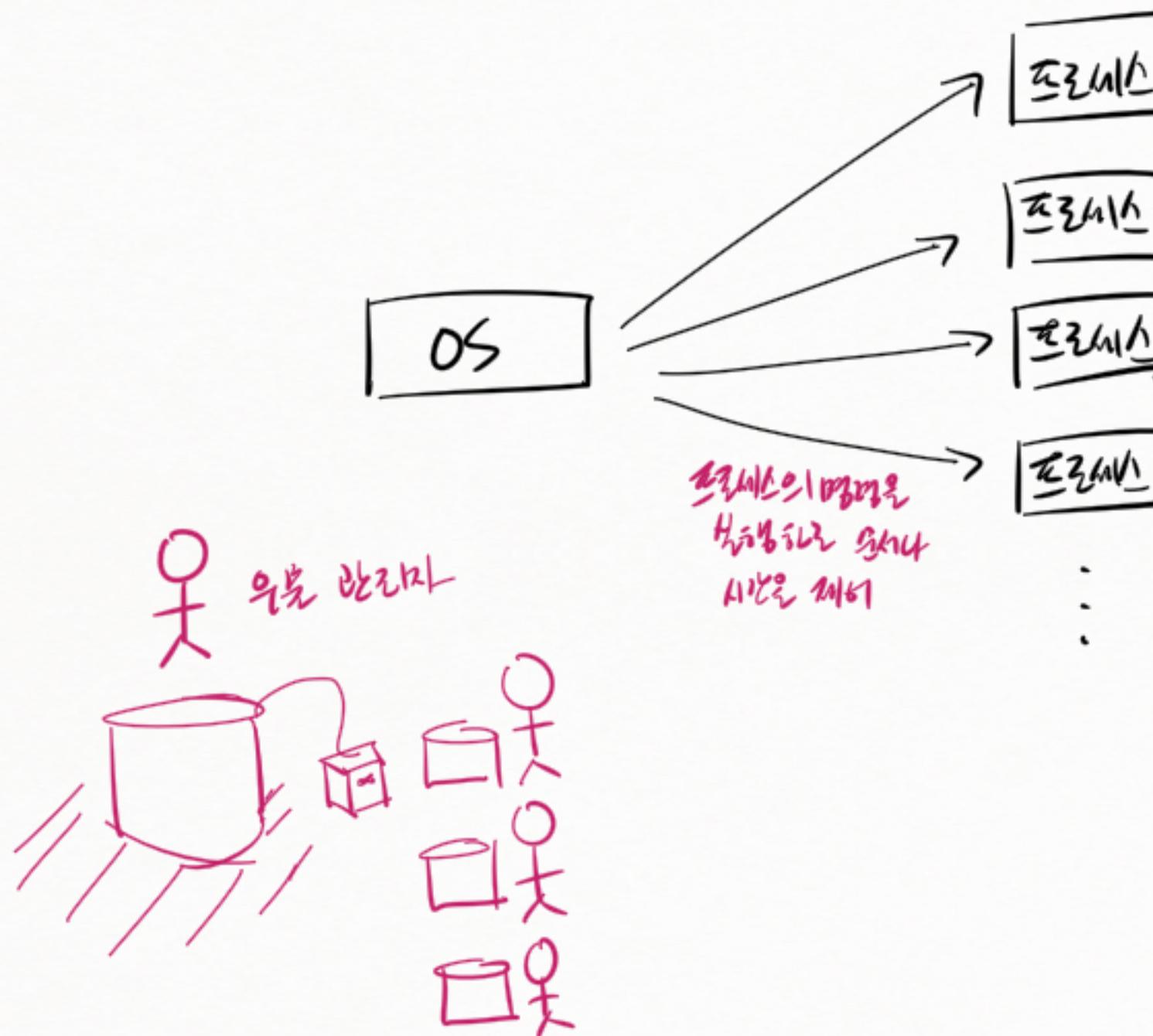


* မျှော်လုပ်ခဲ့သူတော်



* CPU Scheduling

↳ 프로세스를 CPU를 배정하는 방법.



① Round-Robin (Windows OS)

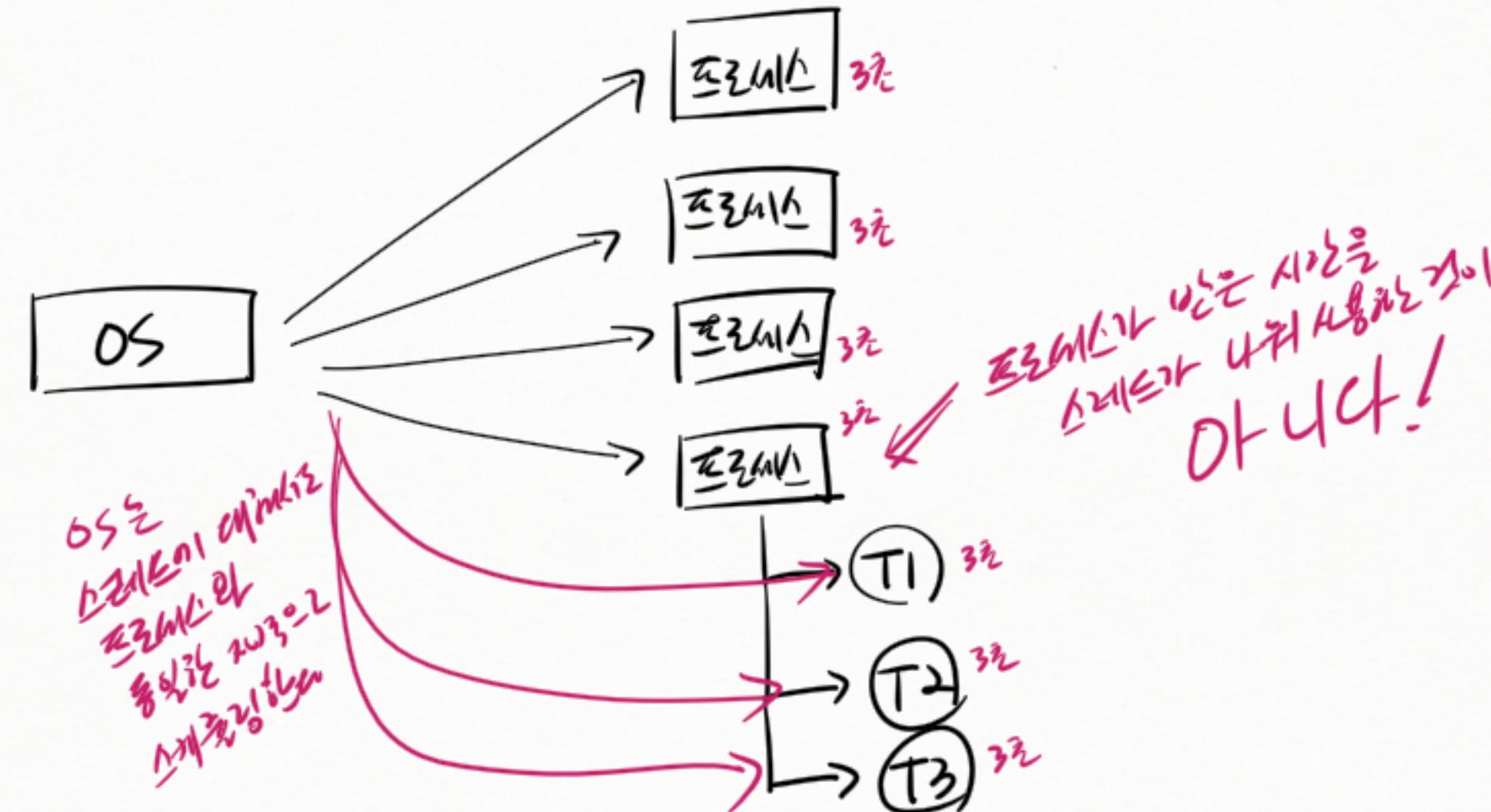
- ↳ 모든 프로세스를 동일한 시간으로 나눠 실행한다
- ↳ 우선순위가 높은가에 영향을 미친다

② Priority + Aging 기법 (Linux, Unix)

- ↳ 우선순위가 높은 프로세스에게
실행 권한이 있는 시간을 더 부여한다
- ↳ 우선순위와 실행에 영향을 미친다.

* CPU Scheduling or 스케줄링

↳ 프로세스를 CPU를 배정하는 방법.



* CPU 스위칭과 Context Switching

↳ 실행 정보와 링크를 CPU 캐시에 복사하는 것

★ Context Switching

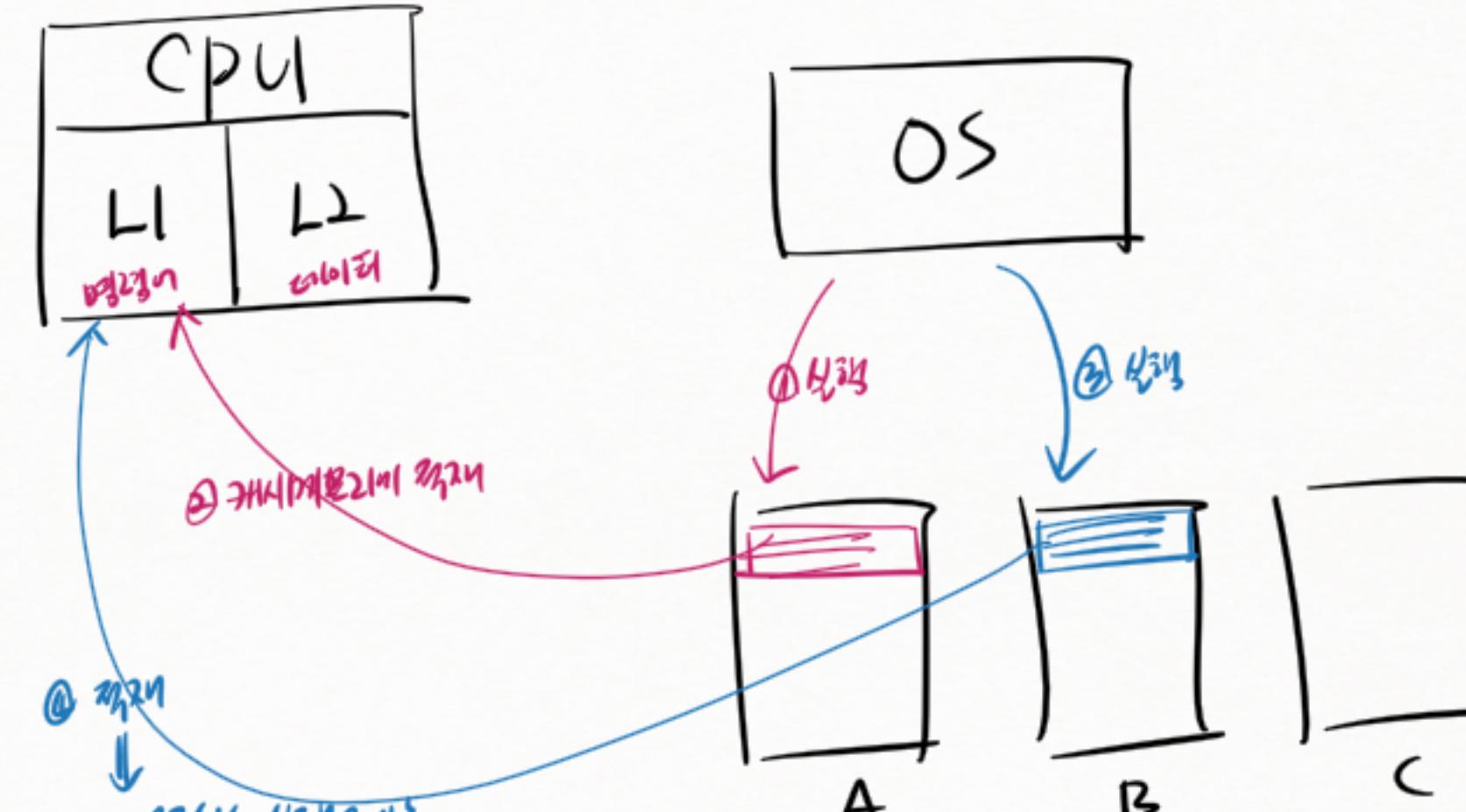
CPU 스위칭 자체에 따라
프로세스를 돌아가면서 링크를 때,

이전 프로세스의 실행 링크를 보는

다른 프로세스의 링크를 찾는다.

더 나은 성능을 위한 링크를 찾는다.
더 나은 다른 프로세스를 고려한다.

링크를 찾았을 시 간단한
Context Switching 시각화
2) 링크 찾았을 때
링크 찾았을 때
 \Rightarrow 링크를 찾았을 때
프로세스를 고려하는 것이다.



* Critical Region = Critical Section

↳ 여러 스레드가 동시에 실행할 때 충돌을 일으킬 수 있는 구간 영역.

\downarrow
Synchronized를 사용하여 처리한다
동시진입을 처리한다.

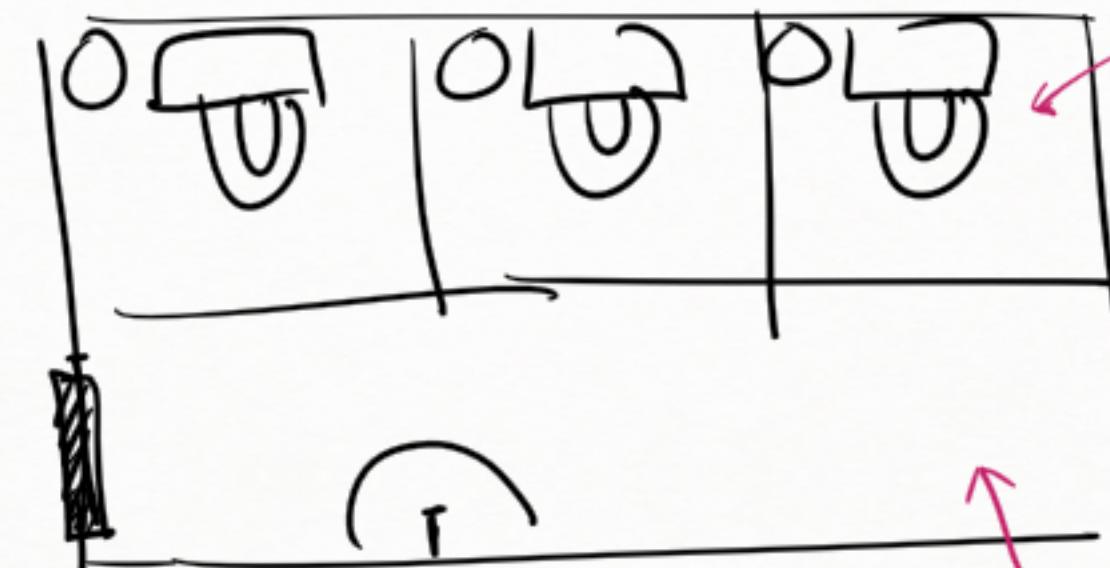


→ 같은 장소에 같은
1개의 자원이 2개의 스레드
2개의 스레드가 같은
자원에 접근하는 경우.
둘은 오직 1명만 접근.

Mutual Exclusion (상호 배제)

ex) 카드게임, TV 앱 등

Mutex
 \downarrow
한 번만 접근!



Critical Section
Semaphore (5)
"동시진입 차단"

Critical Section
Semaphore (1)
"Mutex"

Critical Section
여기서 사용이 동시에 전달되는 경우
2번 가능!