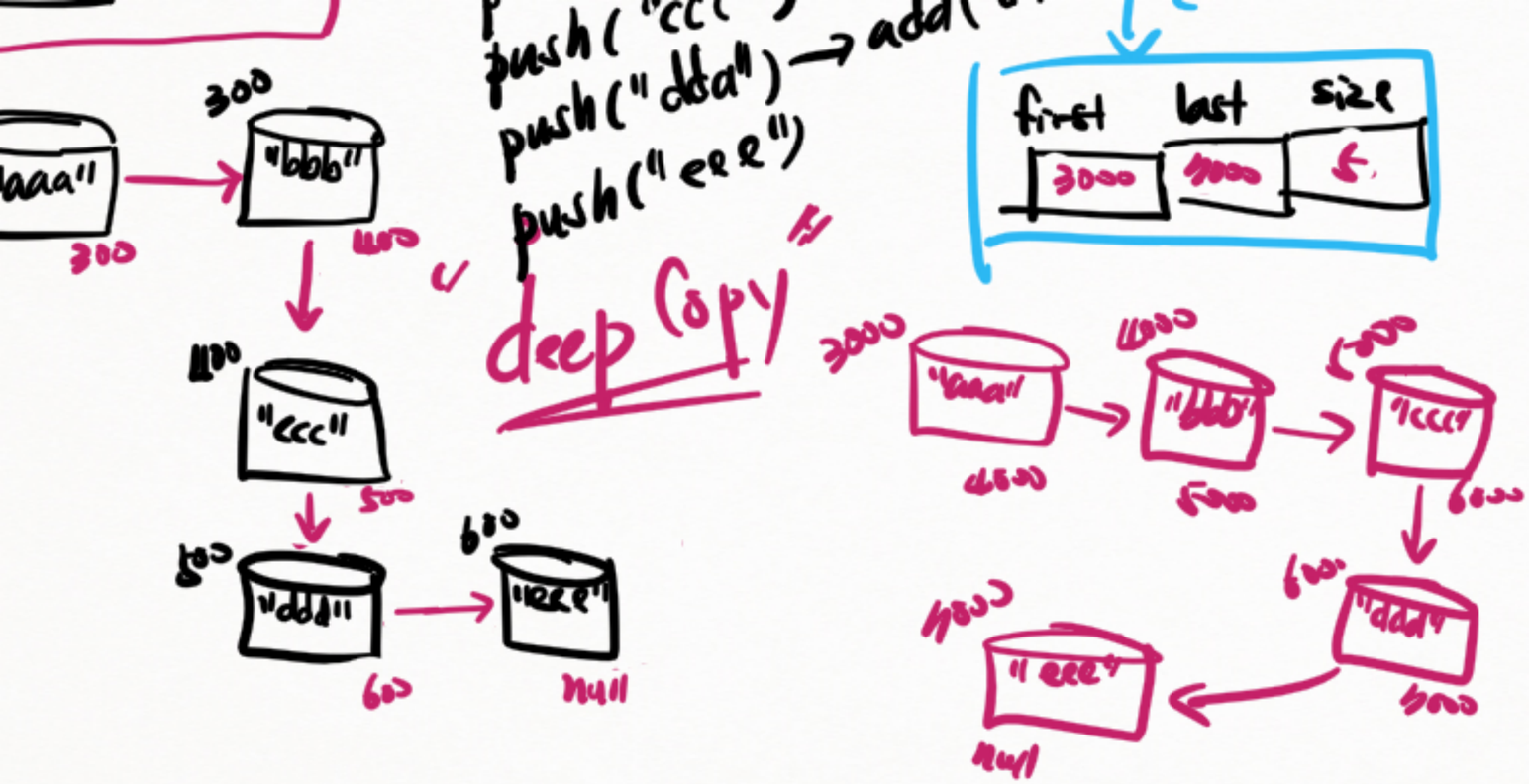
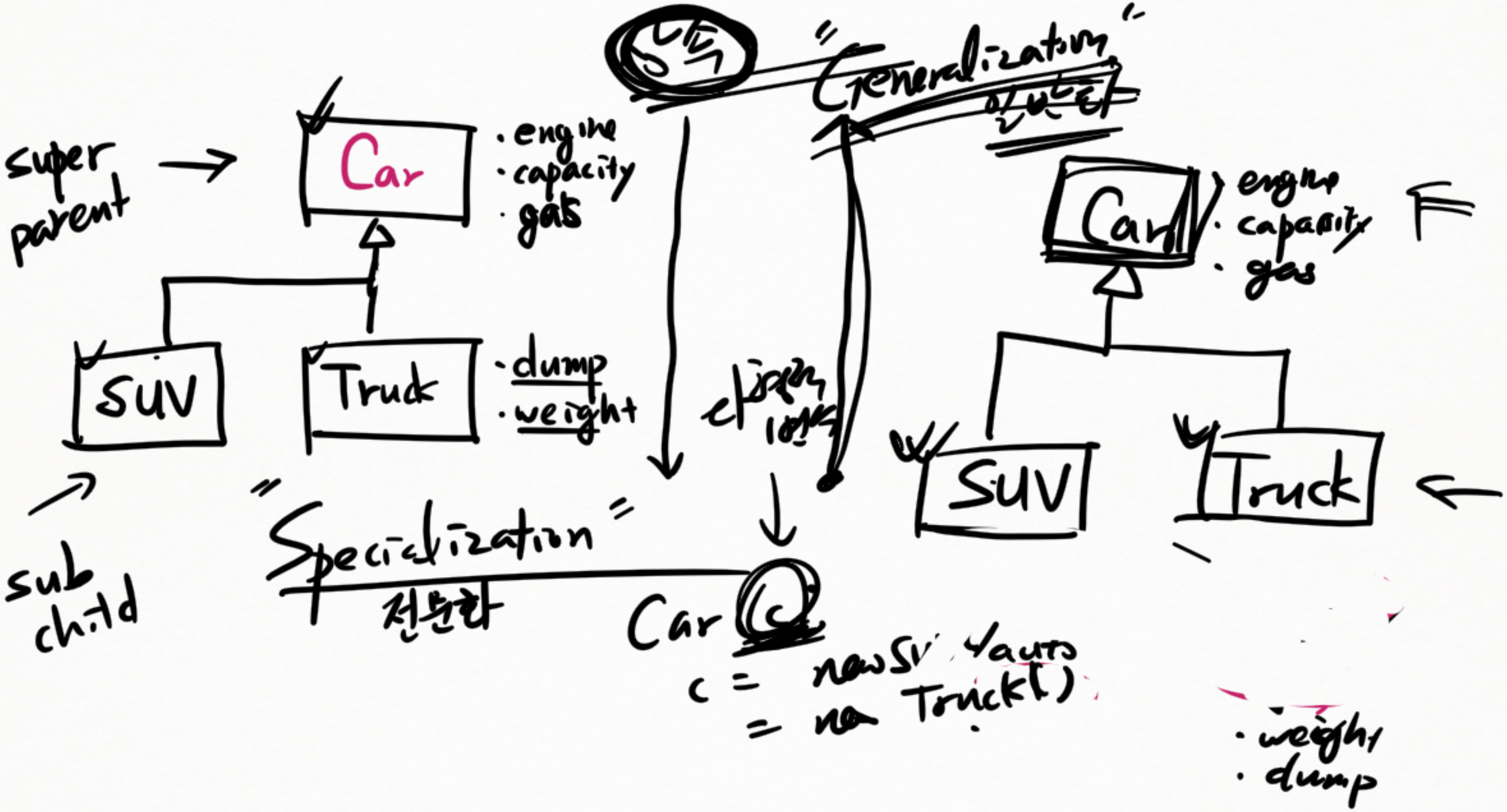
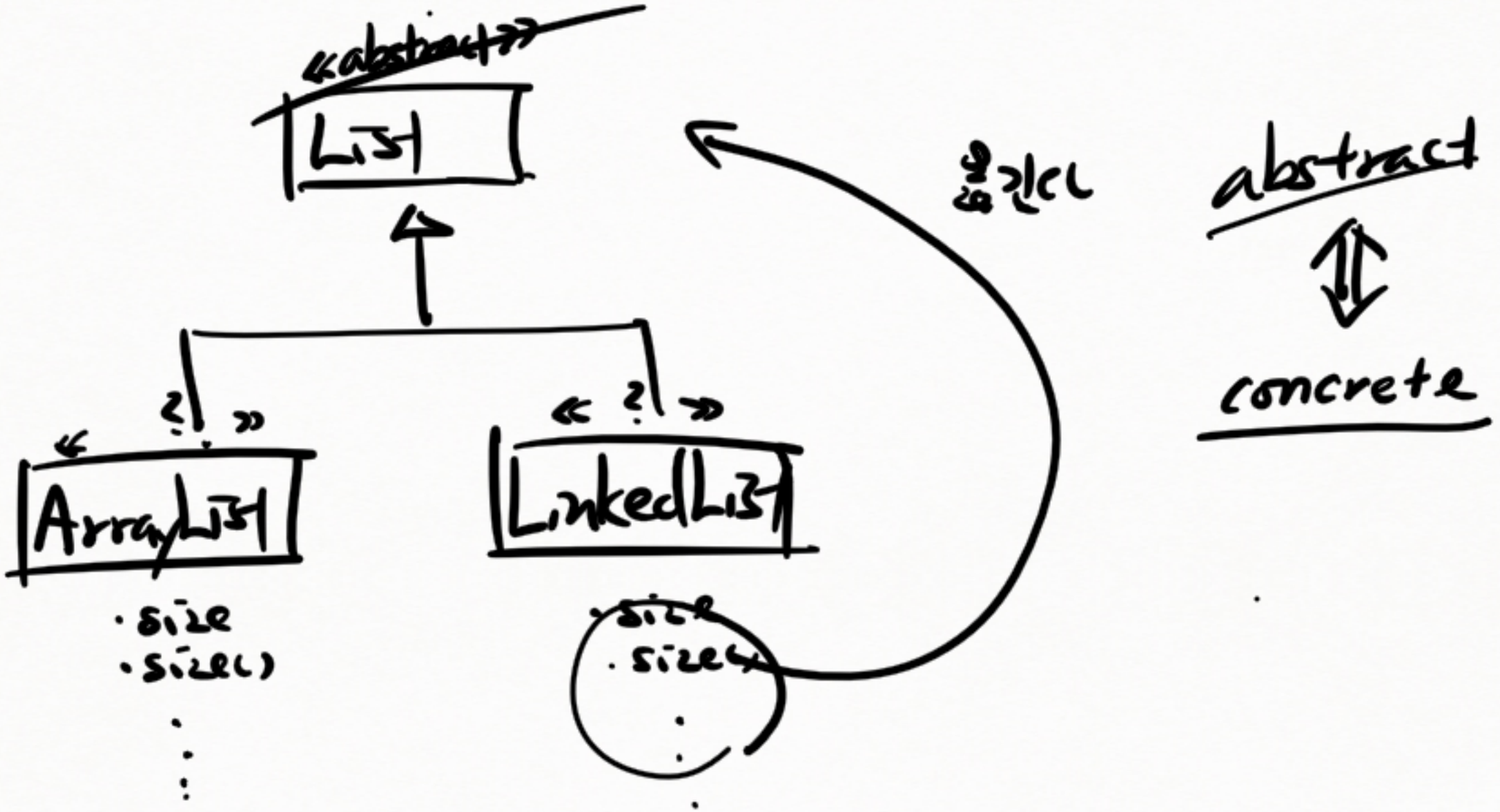
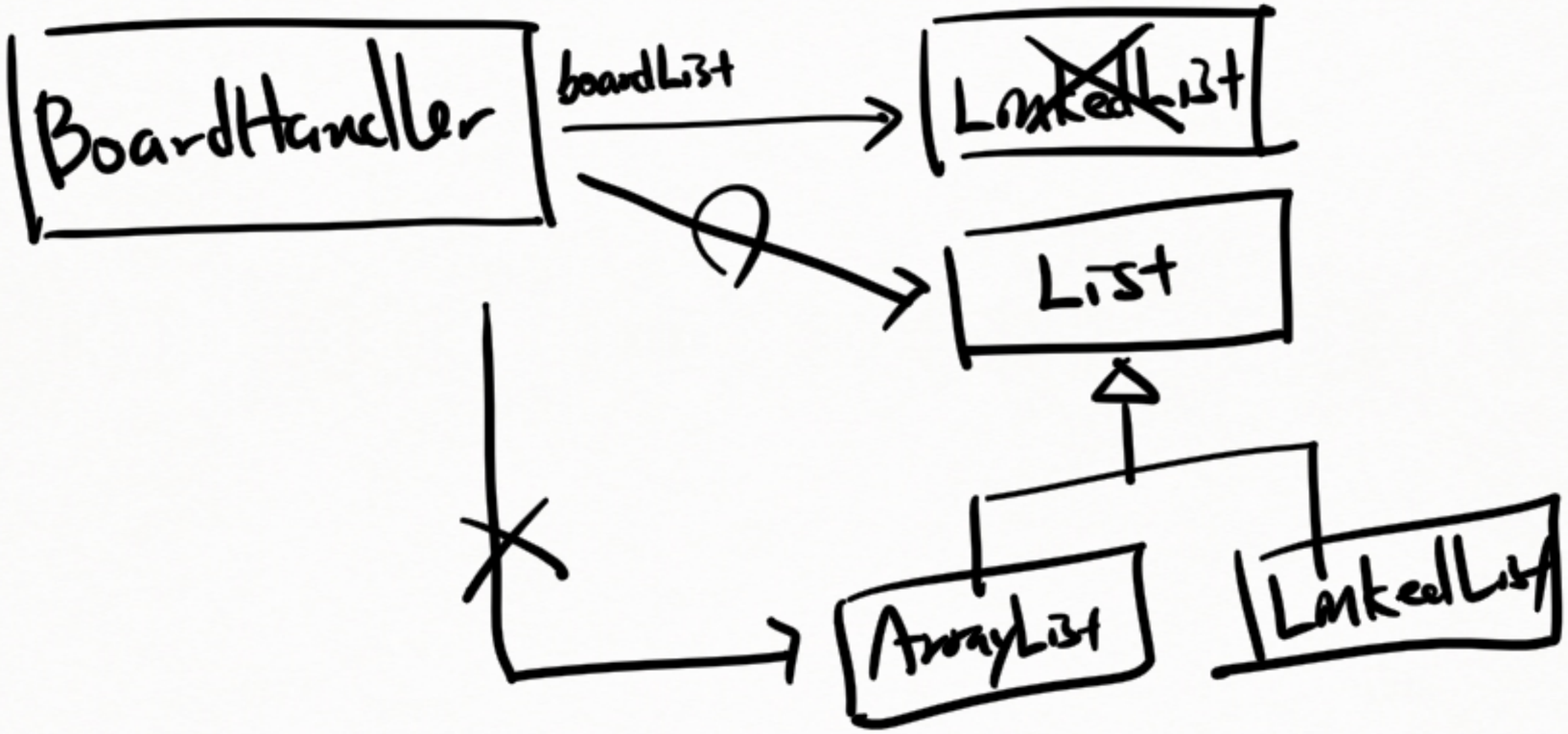
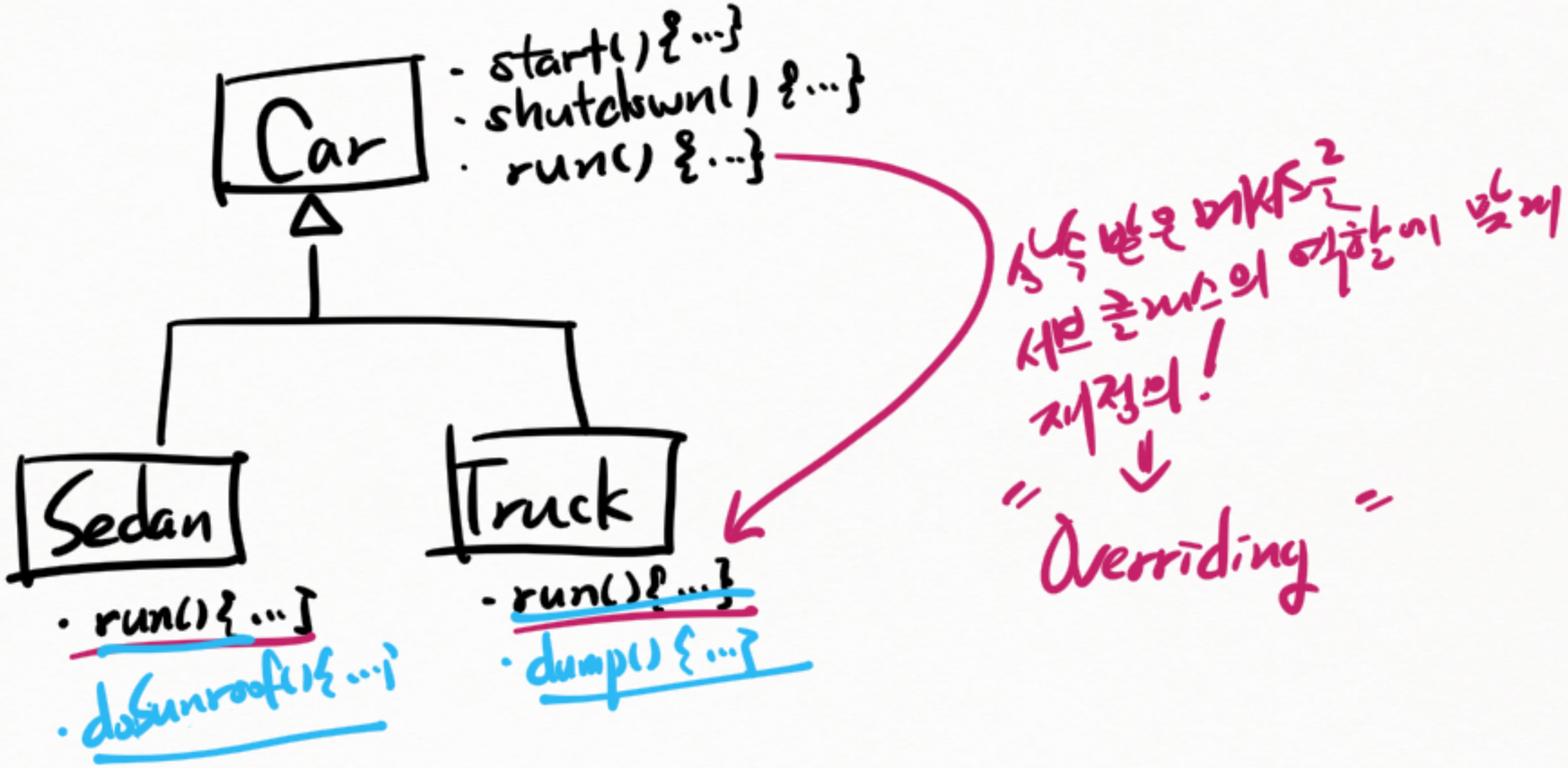
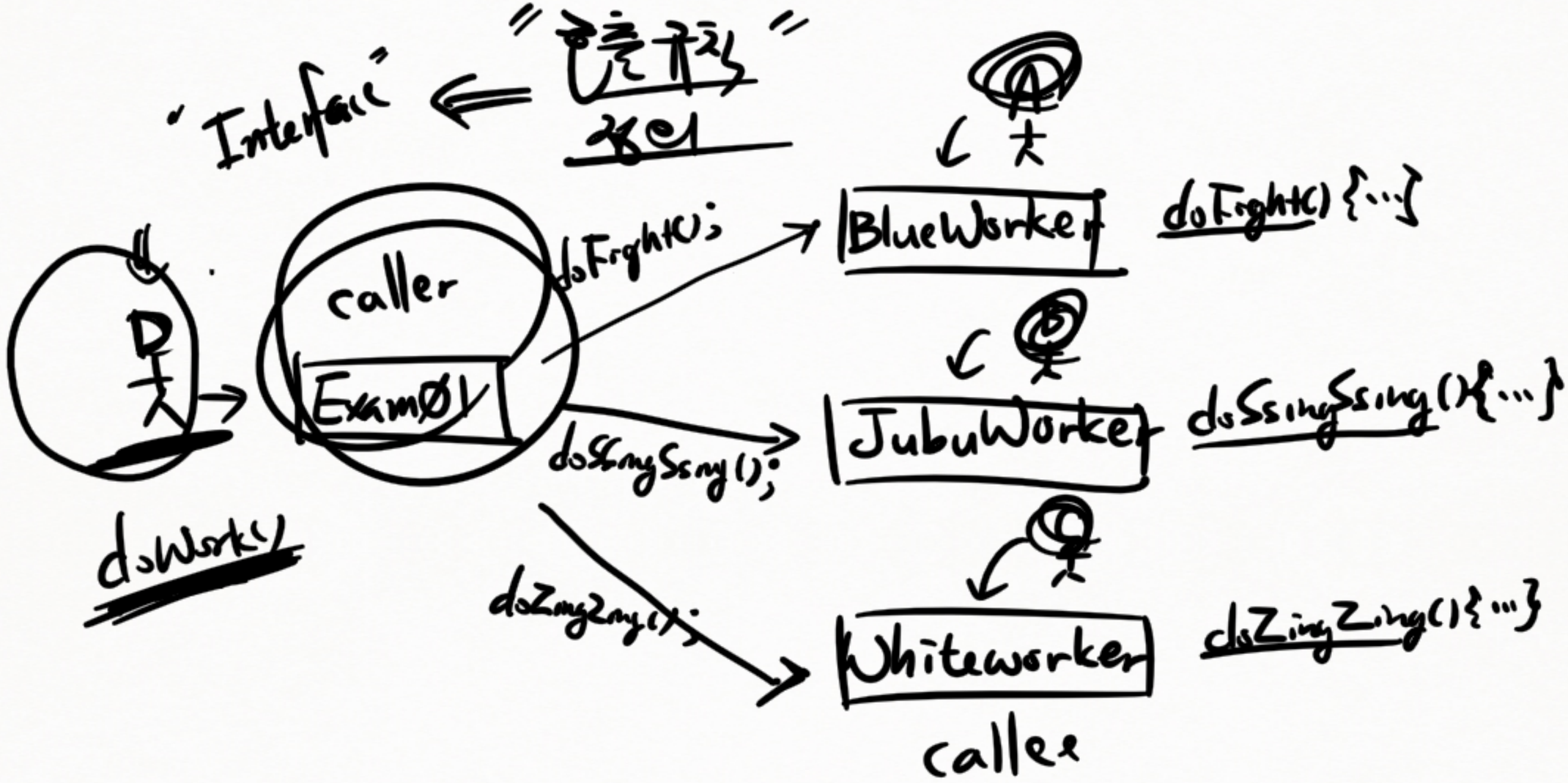

push("ccc")
push("dda")
push("eee")
first
last
size
3000
"aaa"
"bbb"
"ccc"
"ddd"
"eee"
null
deep Copy

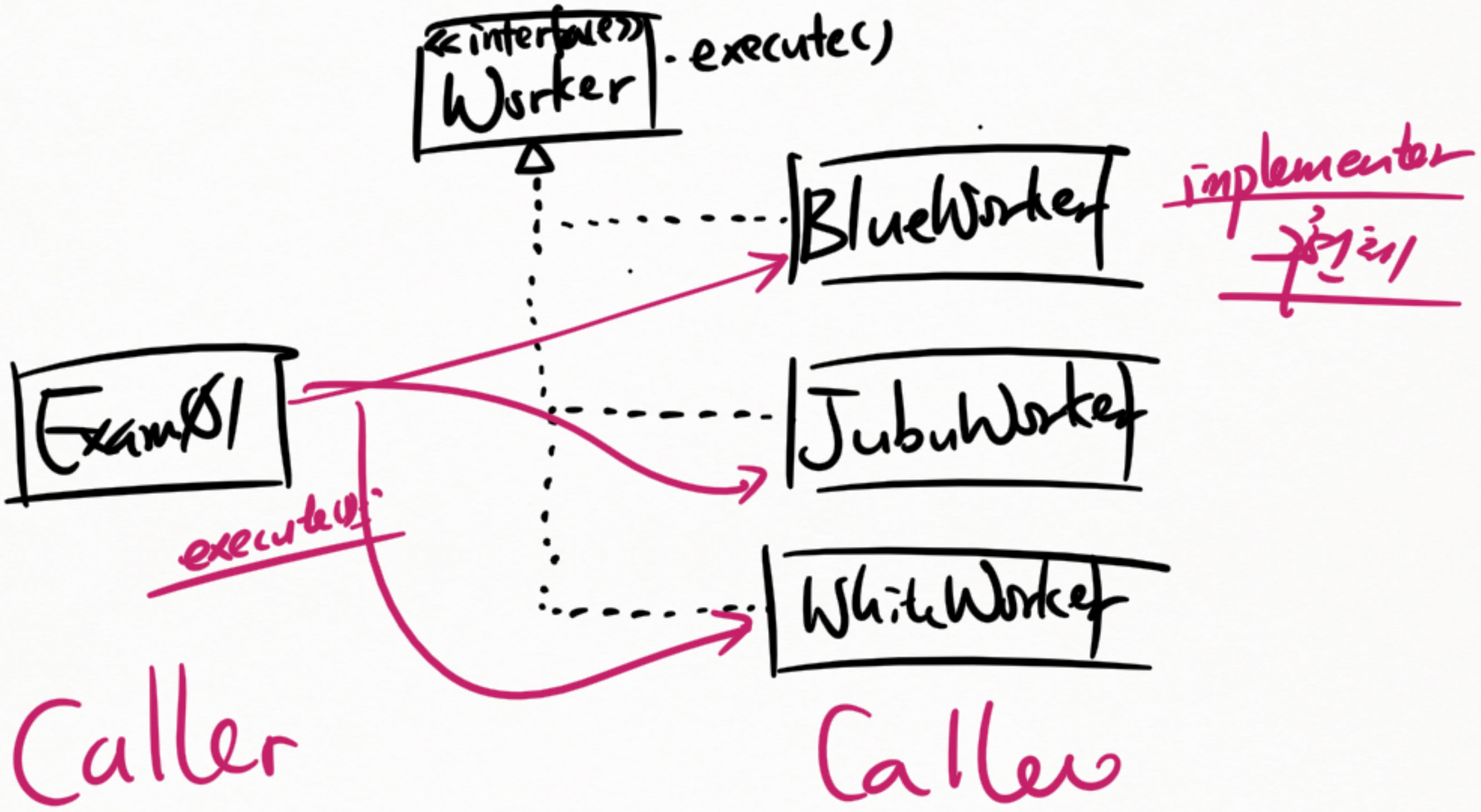

super
parent
Car
· engine
· capacity
· gas
SUV
Truck
· dump
· weight
sub
child
Specialization
전문화
Generalization
일반화
Car
c = new SUV();
= new Truck();
Car
· engine
· capacity
· gas
SUV
Truck
· weight
· dump

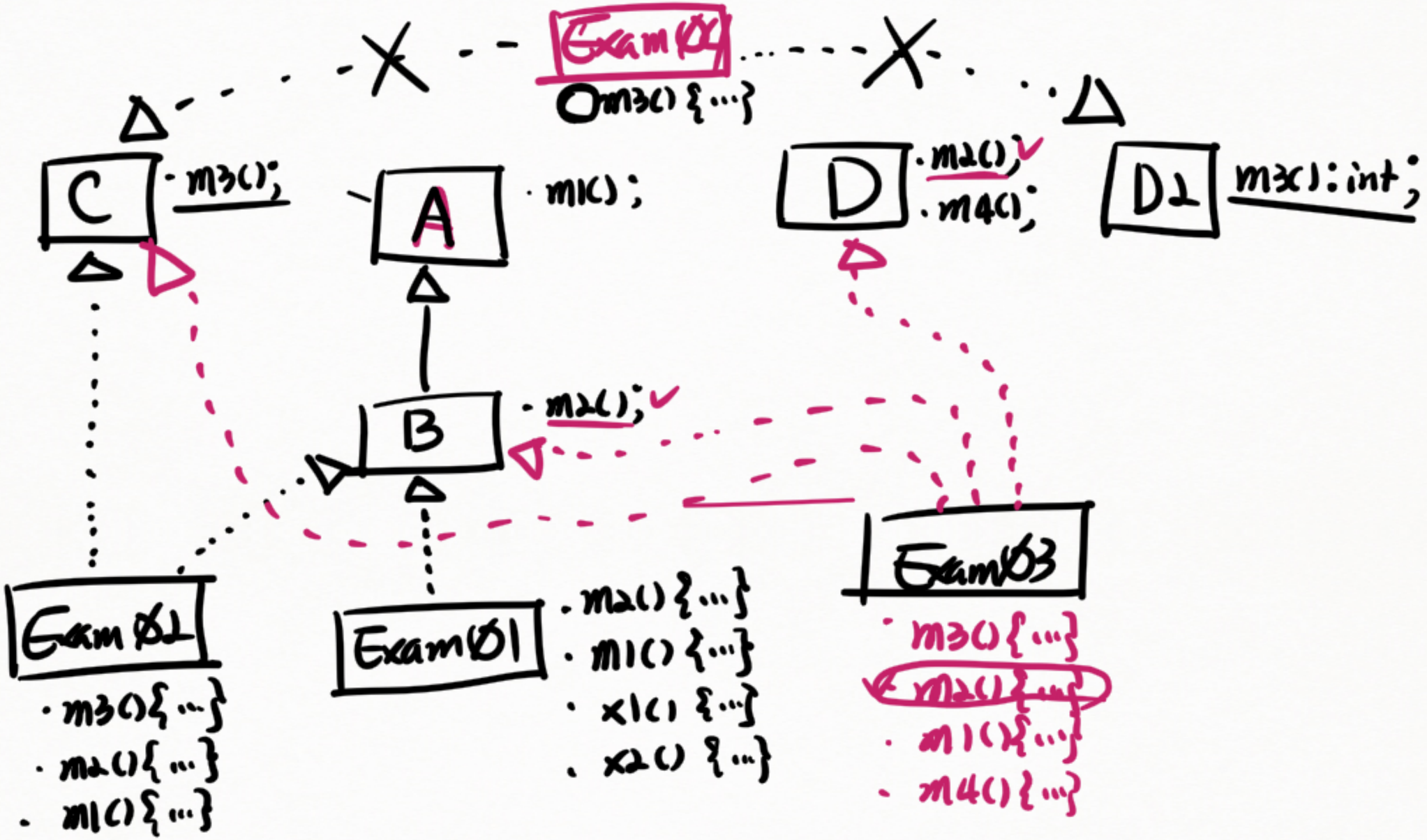

«abstract»
List
ArrayList
LinkedList
size
size()
size
sizeX
abstract
concrete

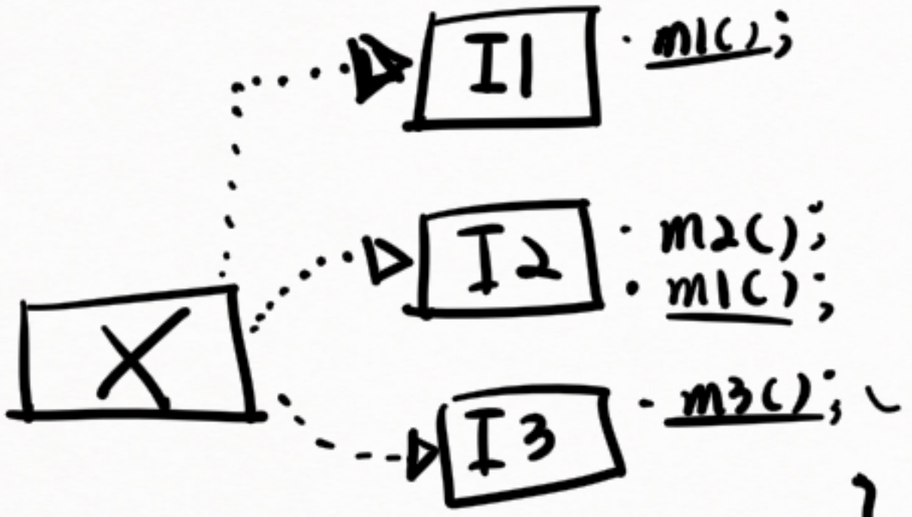

BoardHandler
boardList
LinkedList
List
ArrayList
LinkedList


Car
- start() {...}
- shutdown() {...}
- run() {...}
Sedan
- run() {...}
- doSunroof() {...}
Truck
- run() {...}
- dump() {...}
상속받은 메서드는
서브 클래스의 역할에 맞게
재정의!
"Overriding"

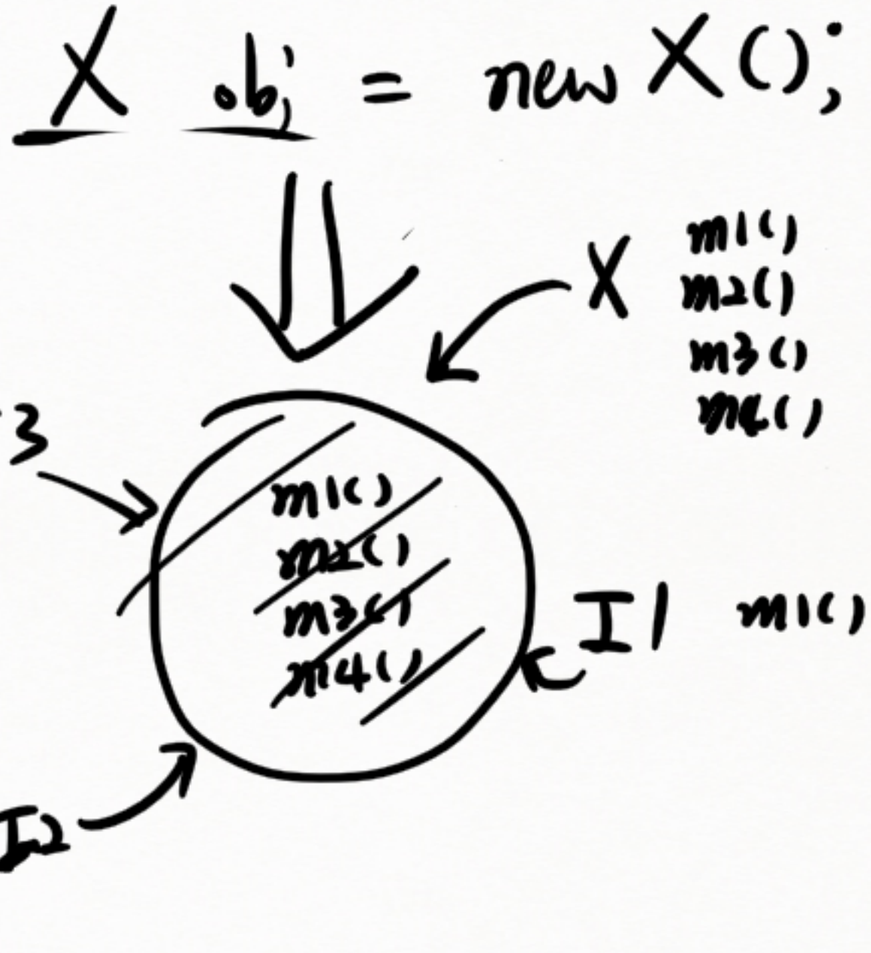

"Interface" ⇐ "클래스"
caller
ExamOI
doWork()
doFrghtC);
doSsingSsing();
doZingZing();
BlueWorker
JubuWorker
WhiteWorker
doFrghtC){...}
doSsingSsing(){...}
doZingZing(){...}
caller

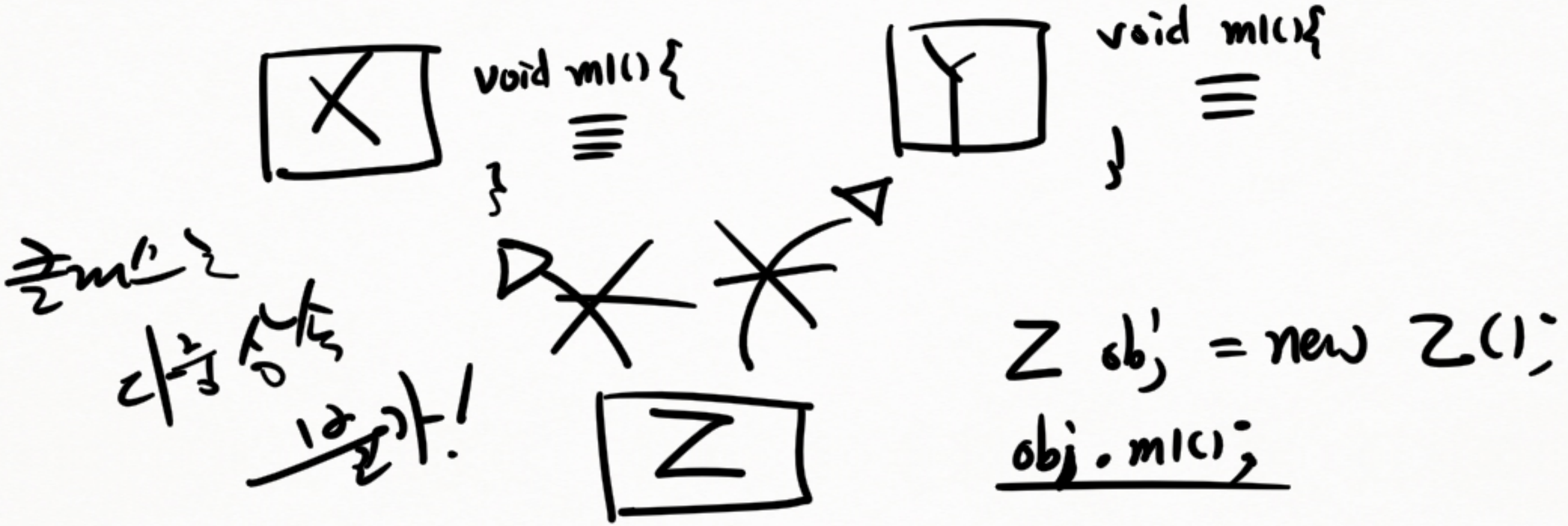

«interface»
Worker
.execute()
BlueWorker
implementer
JubuWorker
Exam01
execute()
WhiteWorker
Caller
Callee

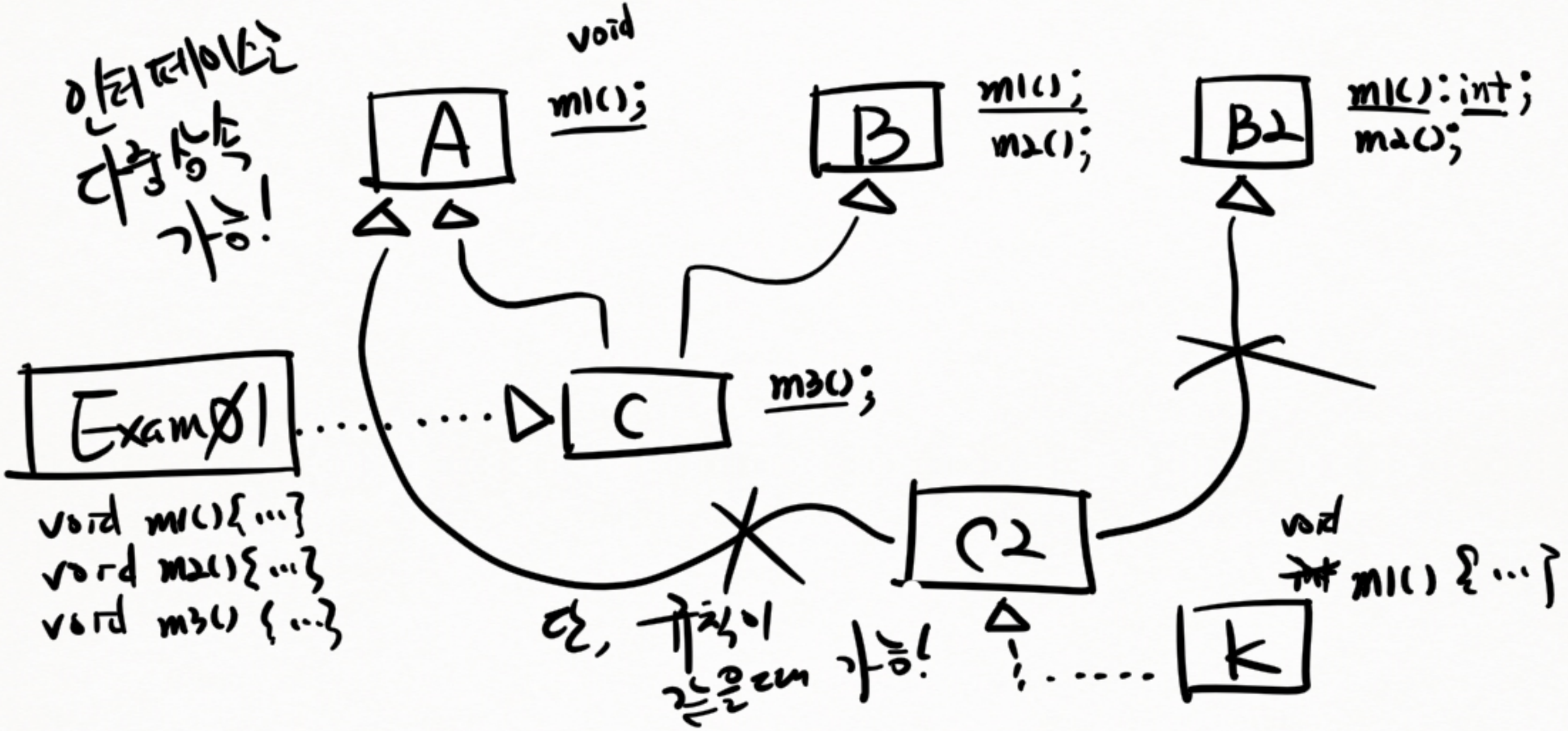

Exam00
+m3() {...}
C
-m3();
A
-m1();
D
-m2();
-m4();
D1
-m31:int;
B
-m2();
Exam02
-m3(){...}
-m2(){...}
-m1(){...}
Exam01
-m2(){...}
-m1(){...}
-x1(){...}
-x2(){...}
Exam03
-m3(){...}
-m2(){...}
-m1(){...}
-m4(){...}

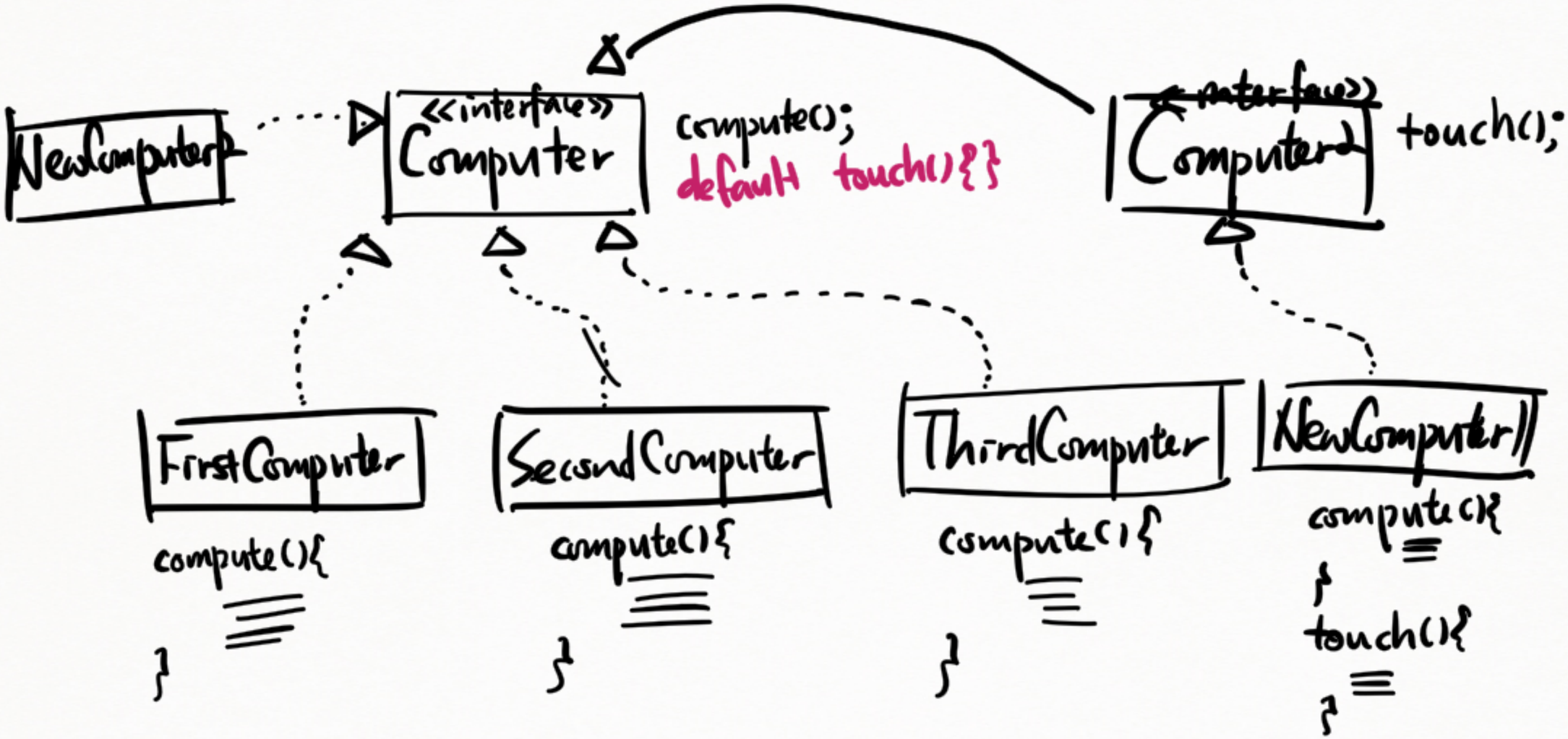


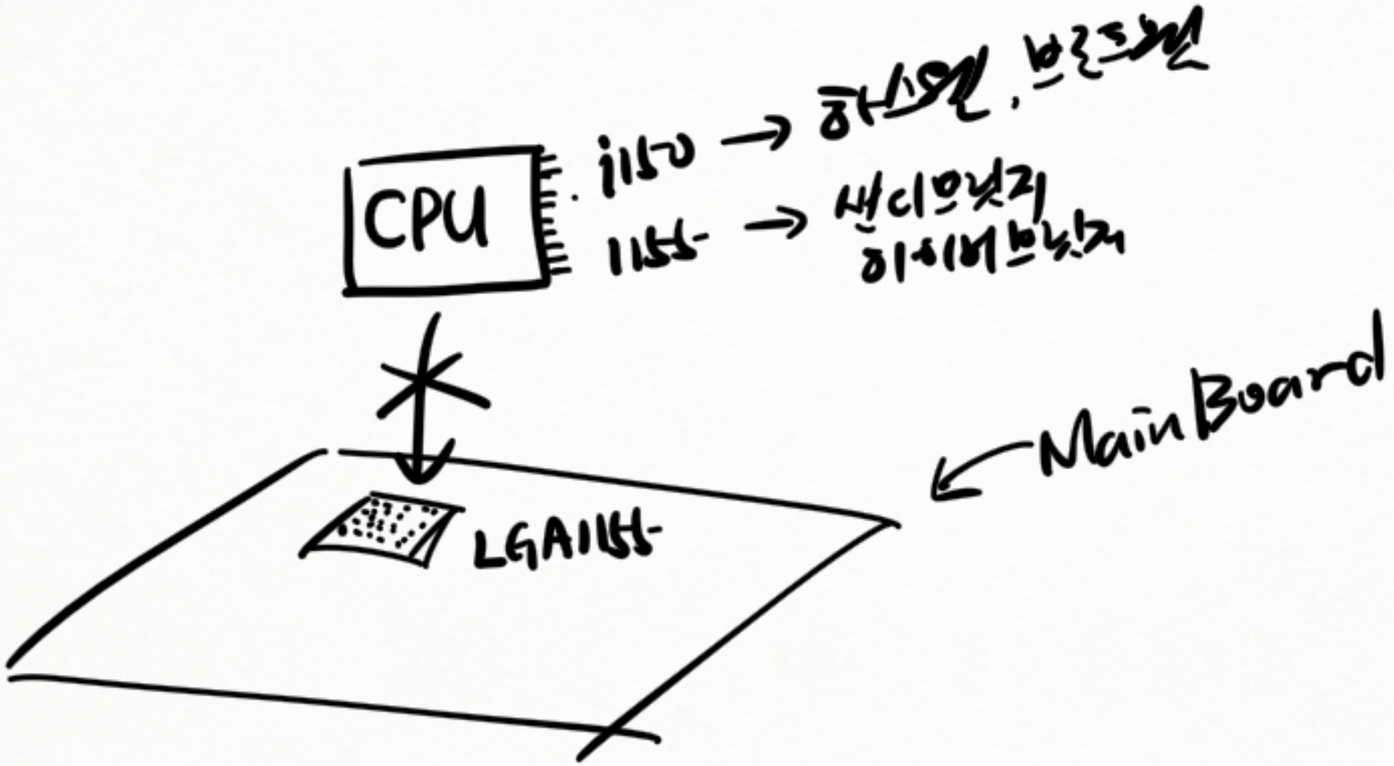

```
class X implements I1, I2, I3{
    void m1() {...}  ← I1, I2
    void m2() {...}  ← I2
    void m3() {...}  ← I3
    void m4() {...}
}
```

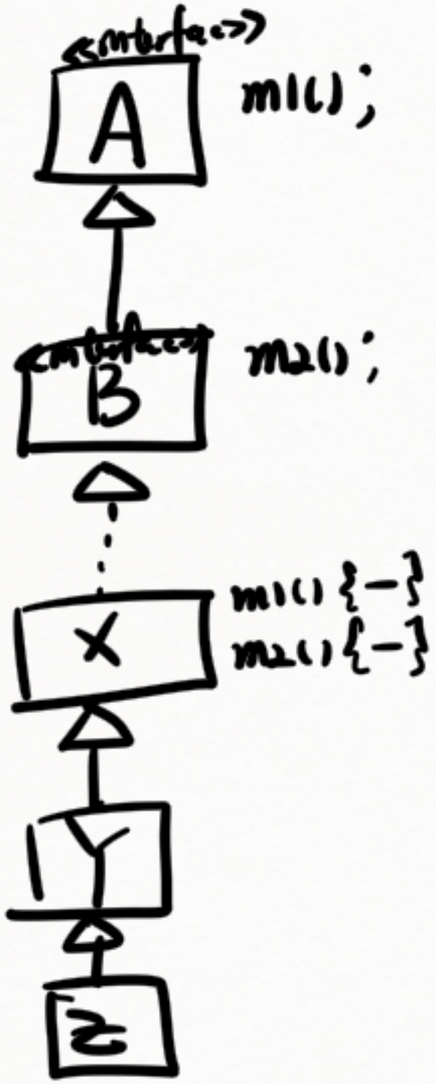


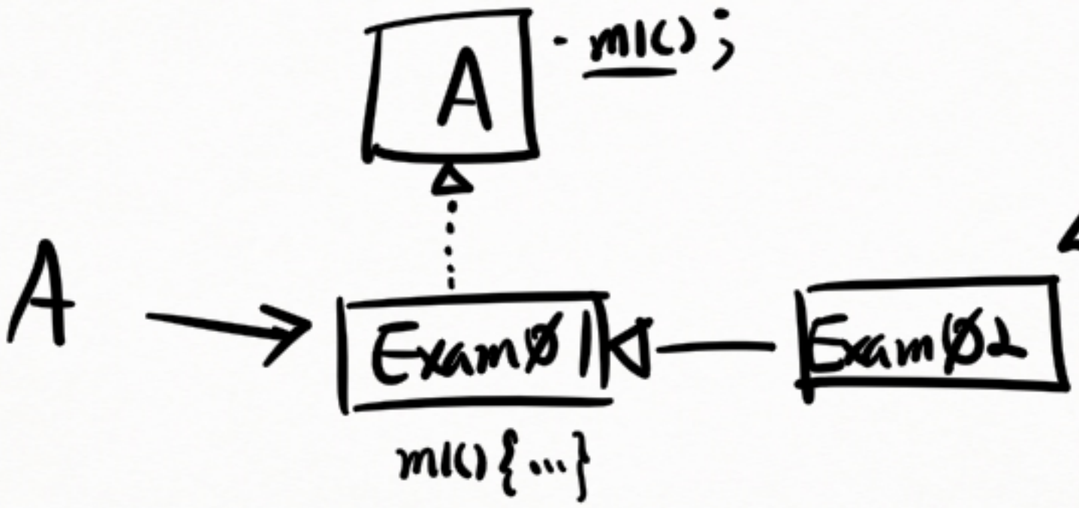

X
void m1(){
}
Y
void m1(){
}
Z
Z obj = new Z();
obj.m1();


인터페이스는 다중상속 가능!
A
void m1();
B
m1();
m2();
B2
m1():int;
m2();
Exam01
void m1(){...}
void m2(){...}
void m3(){...}
C
m3();
C2
k
void m1(){...}
단, 규칙이 있을때 가능!


NewComputer
<<interface>>
Computer
compute();
default touch(){}
<<interface>>
Computer2
touch();
FirstComputer
compute(){
}
SecondComputer
compute(){
}
ThirdComputer
compute(){
}
NewComputer
compute(){
}
touch(){
}

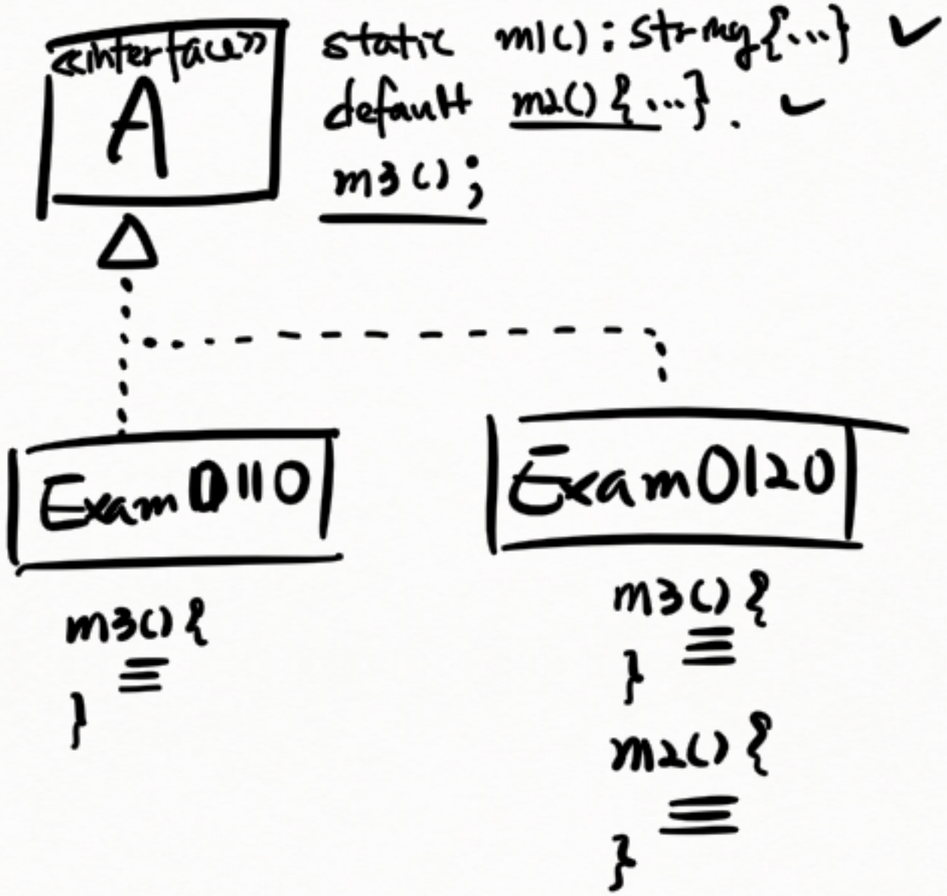

CPU
LGA1155
Main Board

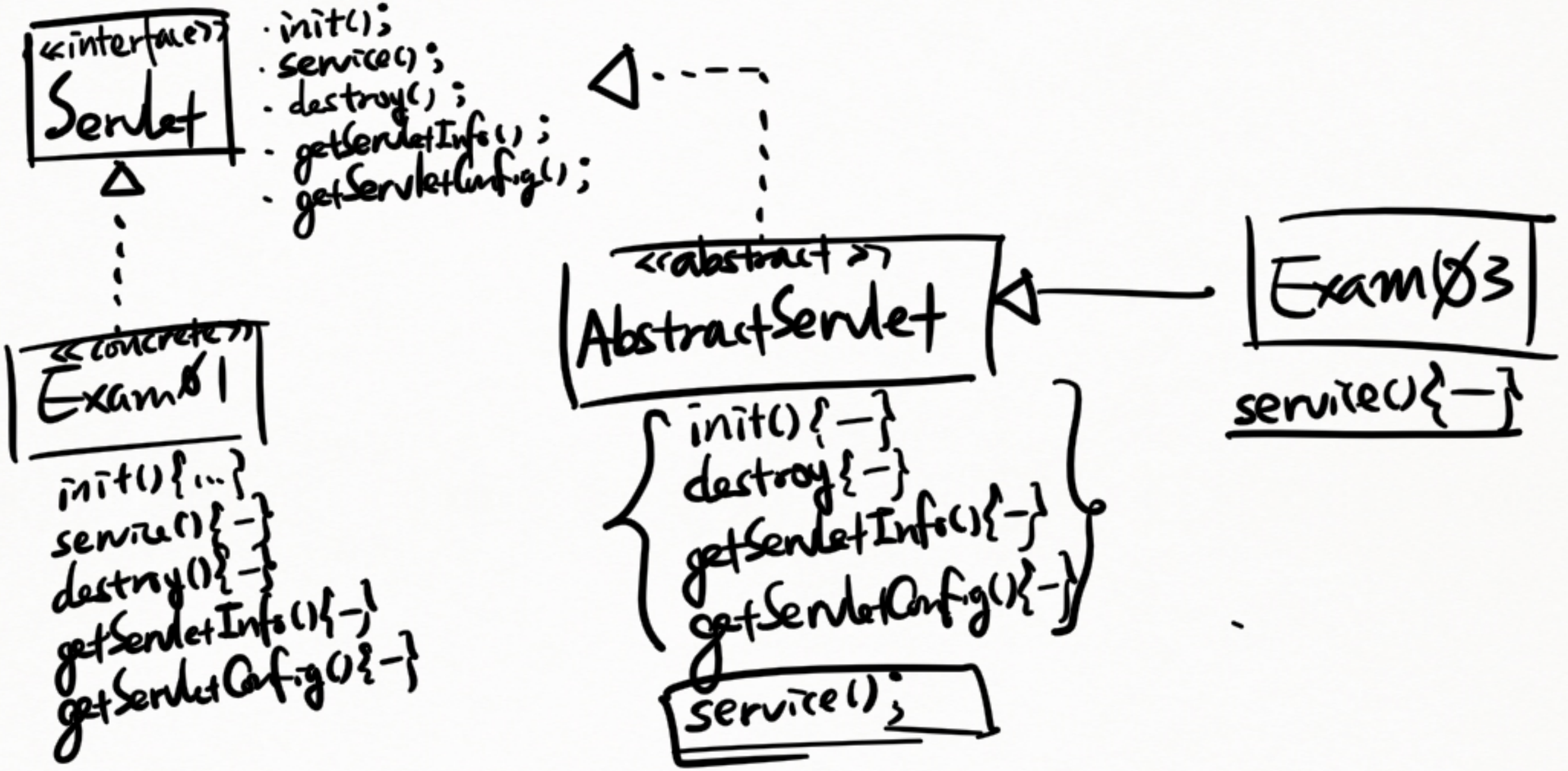


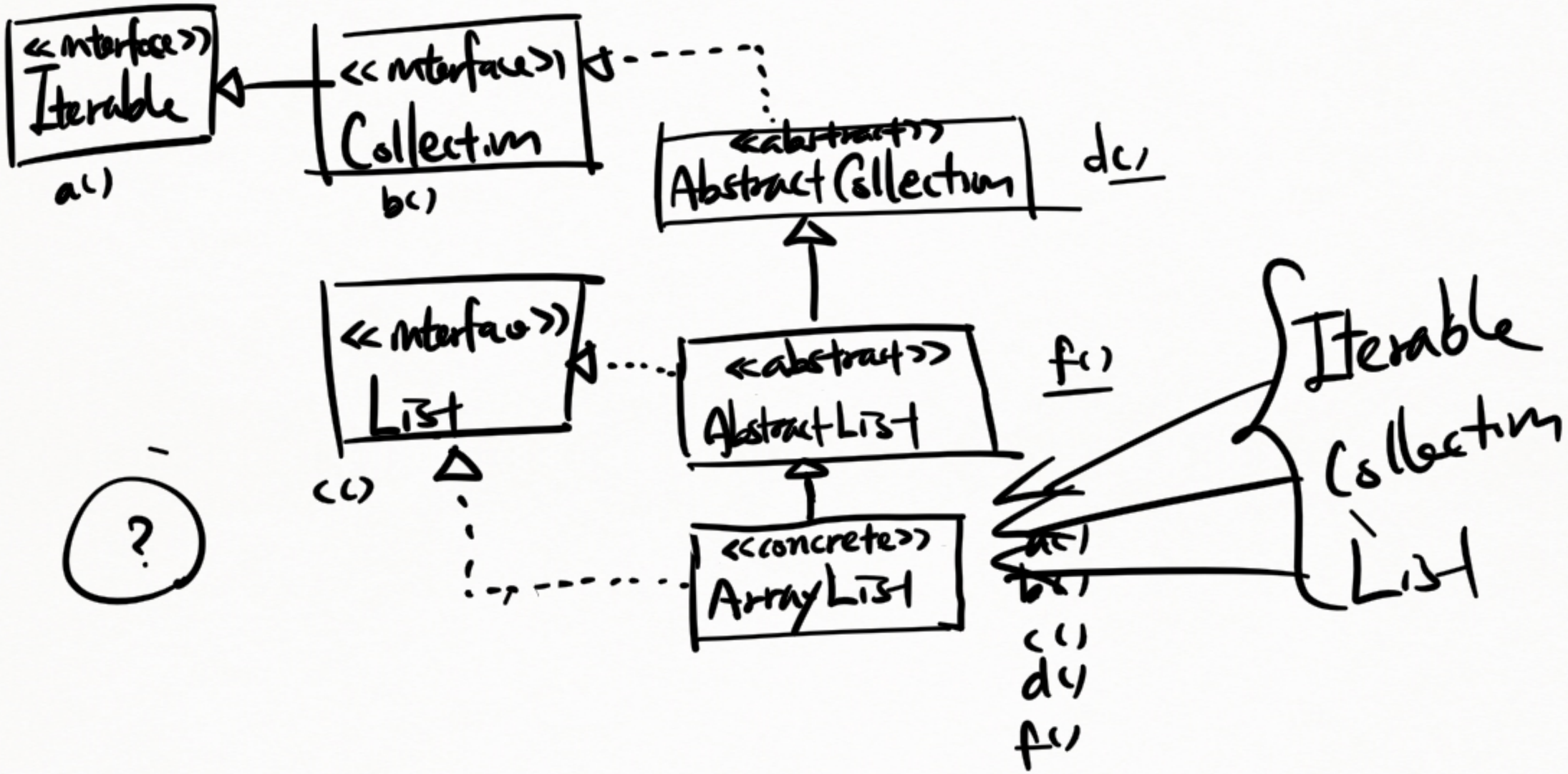


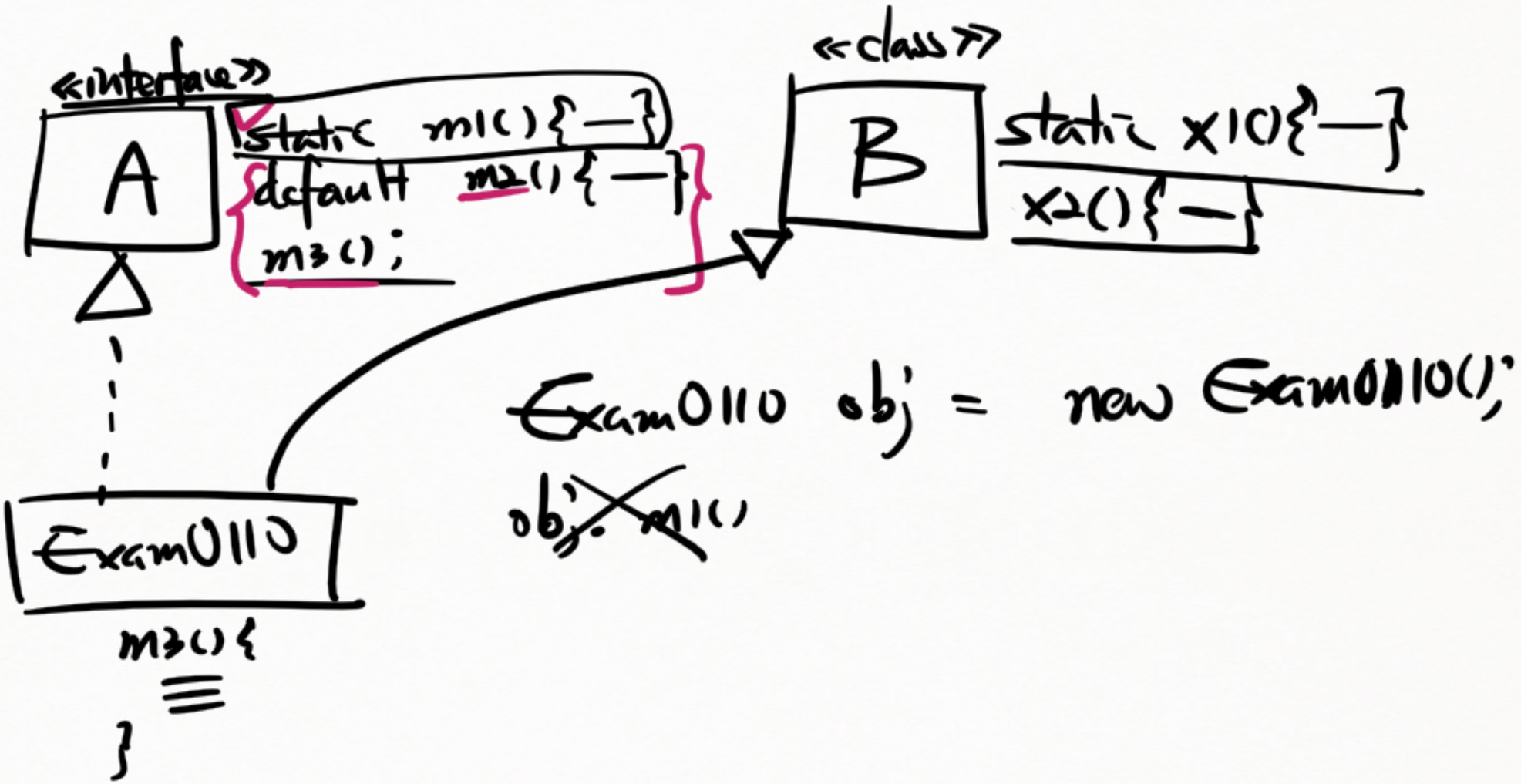

```
A a1 = new X();
     = new Y();
     = new Z();
```

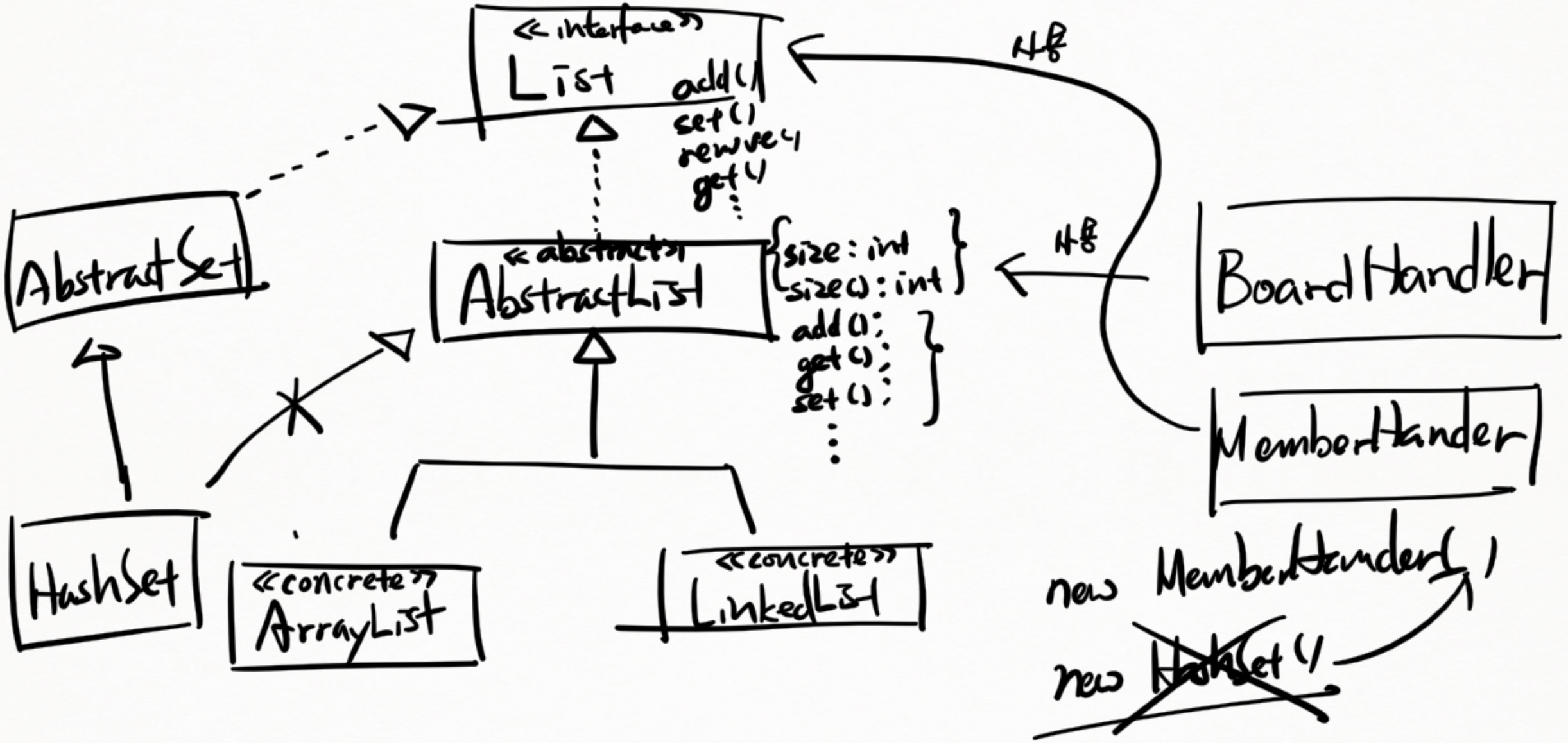
```
B a2 = new X();
     = new Y();
     = new Z();
```

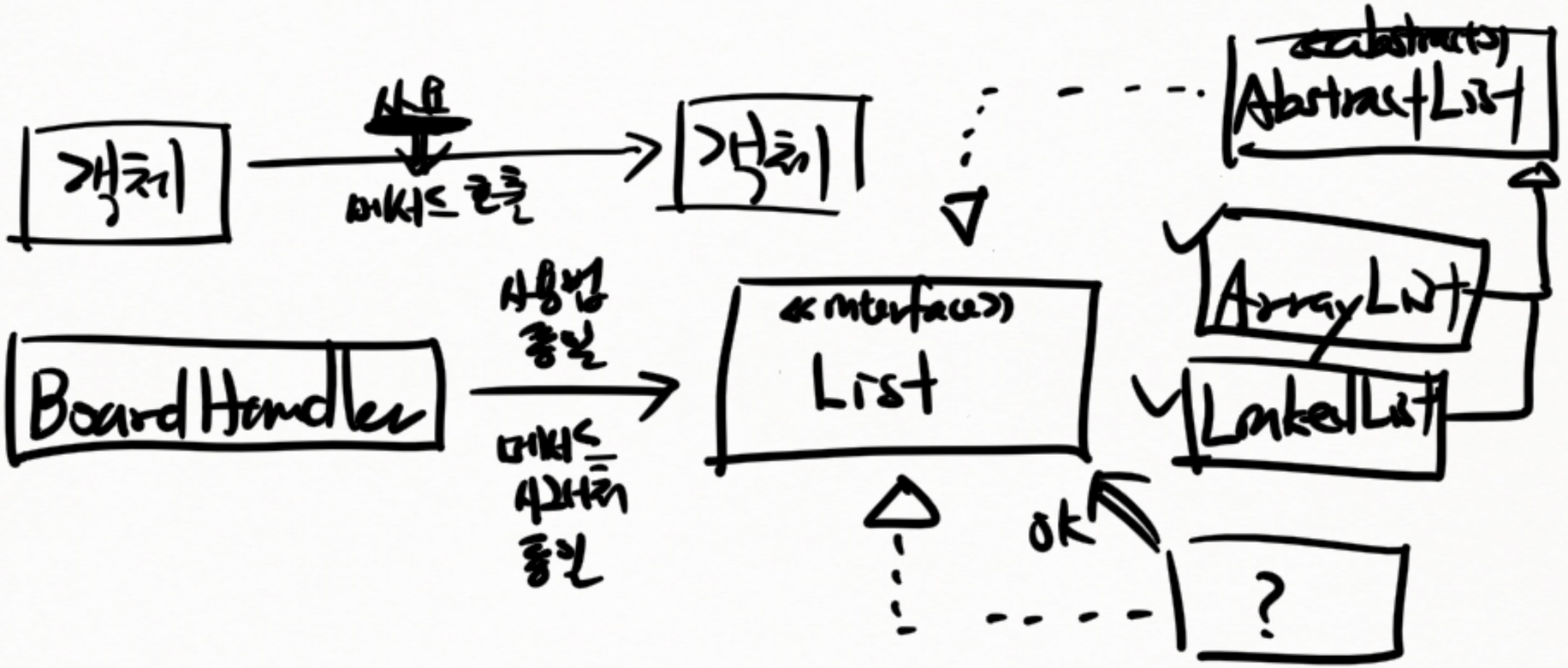

«interface»
A
static m1() : String{...}
default m2(){...}
m3();
Exam0110
m3(){
}
Exam0120
m3(){
}
m2(){
}

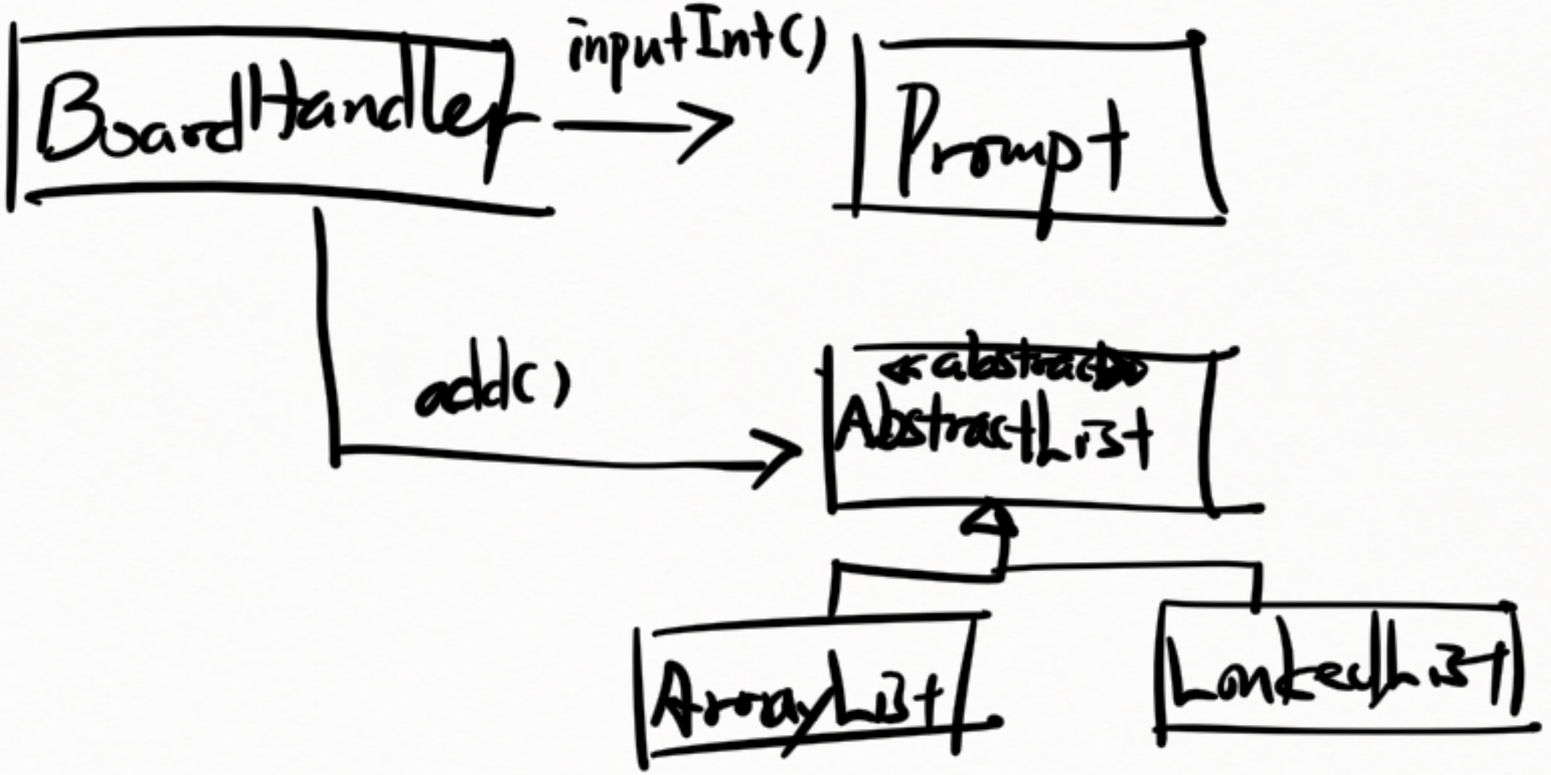

«interface»
Servlet
· init();
· service();
· destroy();
· getServletInfo();
· getServletConfig();
«concrete»
Exam01
init(){...}
service(){-}
destroy(){-}
getServletInfo(){-}
getServletConfig(){-}
«abstract»
AbstractServlet
init(){-}
destroy{-}
getServletInfo(){-}
getServletConfig(){-}
service();
Exam03
service(){-}


«interface»
Iterable
a()
«interface»
Collection
b()
«abstract»
AbstractCollection
d()
«interface»
List
c()
«abstract»
AbstractList
f()
«concrete»
ArrayList
a()
b()
c()
d()
f()
Iterable
Collection
List
?


«interface»
A
static m1(){—}
default m2(){—}
m3();
«class»
B
static x10{—}
x2(){—}
Exam0110
m3(){
≡
}
Exam0110 obj = new Exam0110();
obj.~~m1()~~


«interface»
List
add()
set()
remove()
get()
AbstractSet
«abstract»
AbstractList
size : int
size() : int
add();
get();
set();
HashSet
«concrete»
ArrayList
«concrete»
LinkedList
HB
HB
BoardHandler
MemberHander
new MemberHander();
new HashSet();

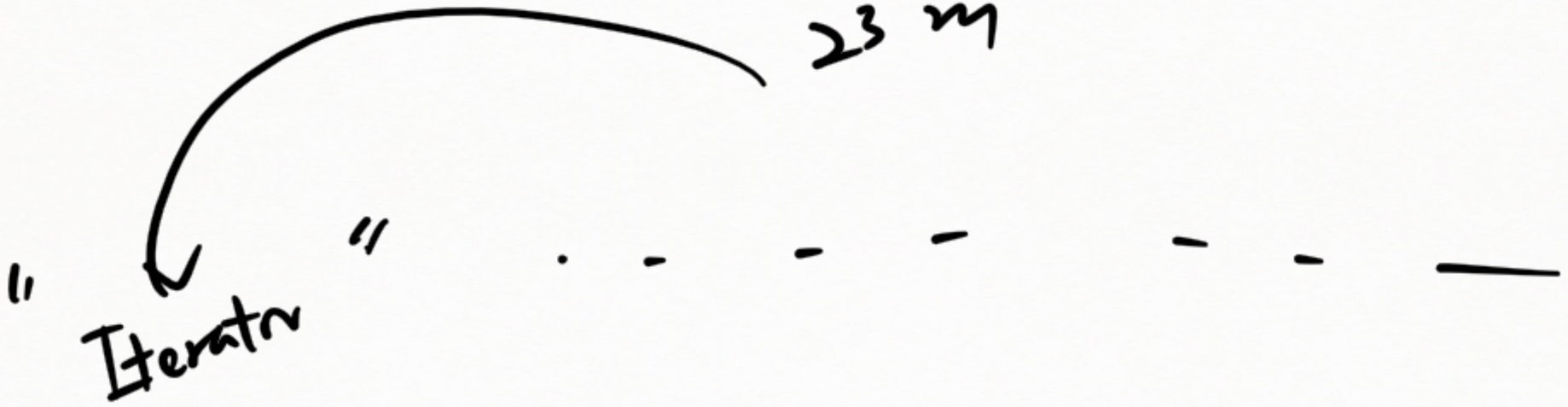

객체
API
메서드 호출
객체
<<abstract>>
AbstractList
BoardHandler
사용법
통보
메서드
사용처
통보
<<interface>>
List
ArrayList
LinkedList
OK
?

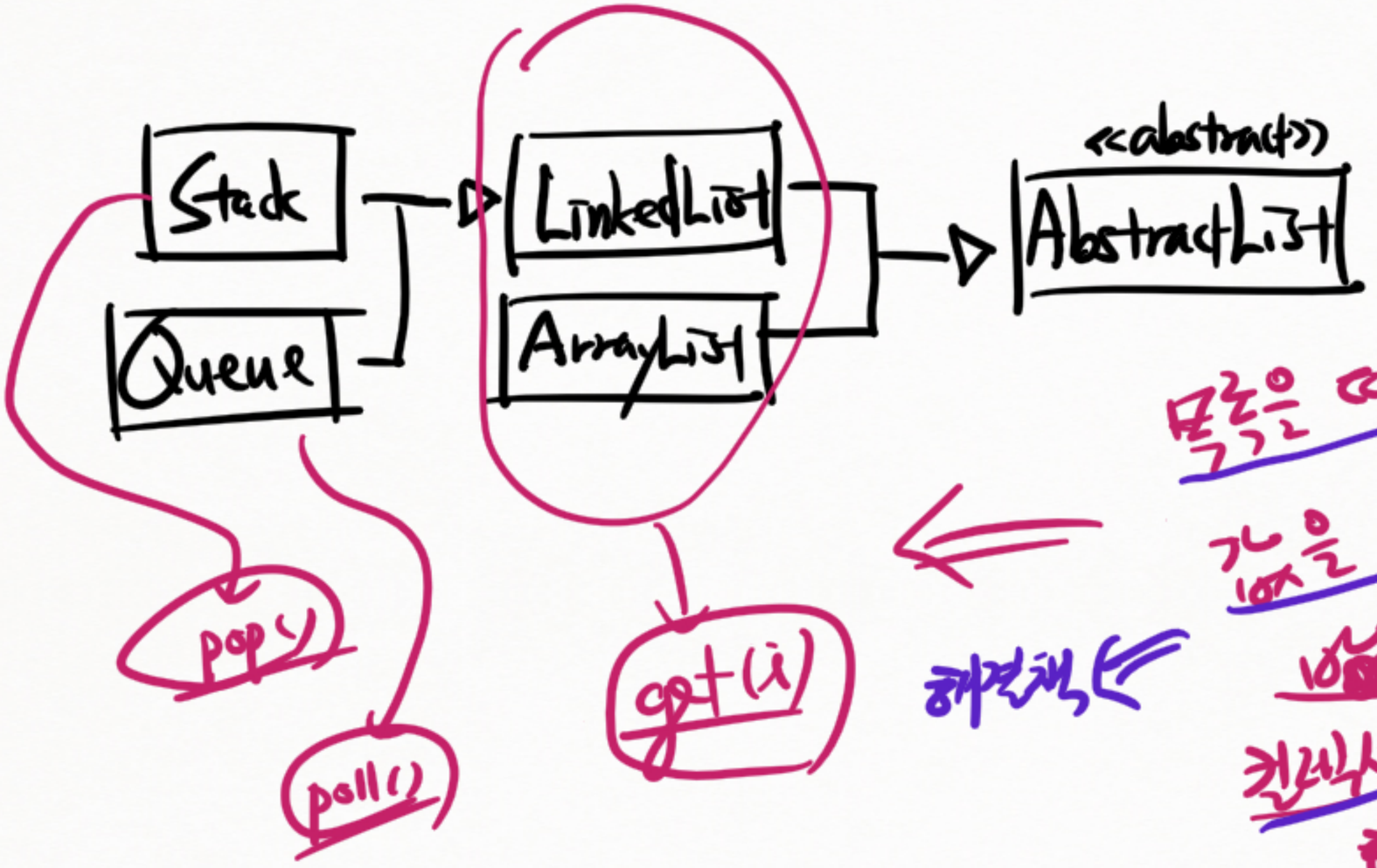

BoardHandler
inputInt()
Prompt
add()
AbstractList
ArrayList
LinkedList

"Gang of Four (GoF)

"Design Patterns"

23 nos





해결책

목록을 따라가면서
값을 조회하는
방법이
컬렉션의 종류에 따라 다르다

"순차적으로 조회"
↓
iterate

조회하는 메서드가 다르다
프로그램의 일관성이 없다

컬렉션의 타입이 다르더라도
같은 꺼내는 방법을 통일

⇊

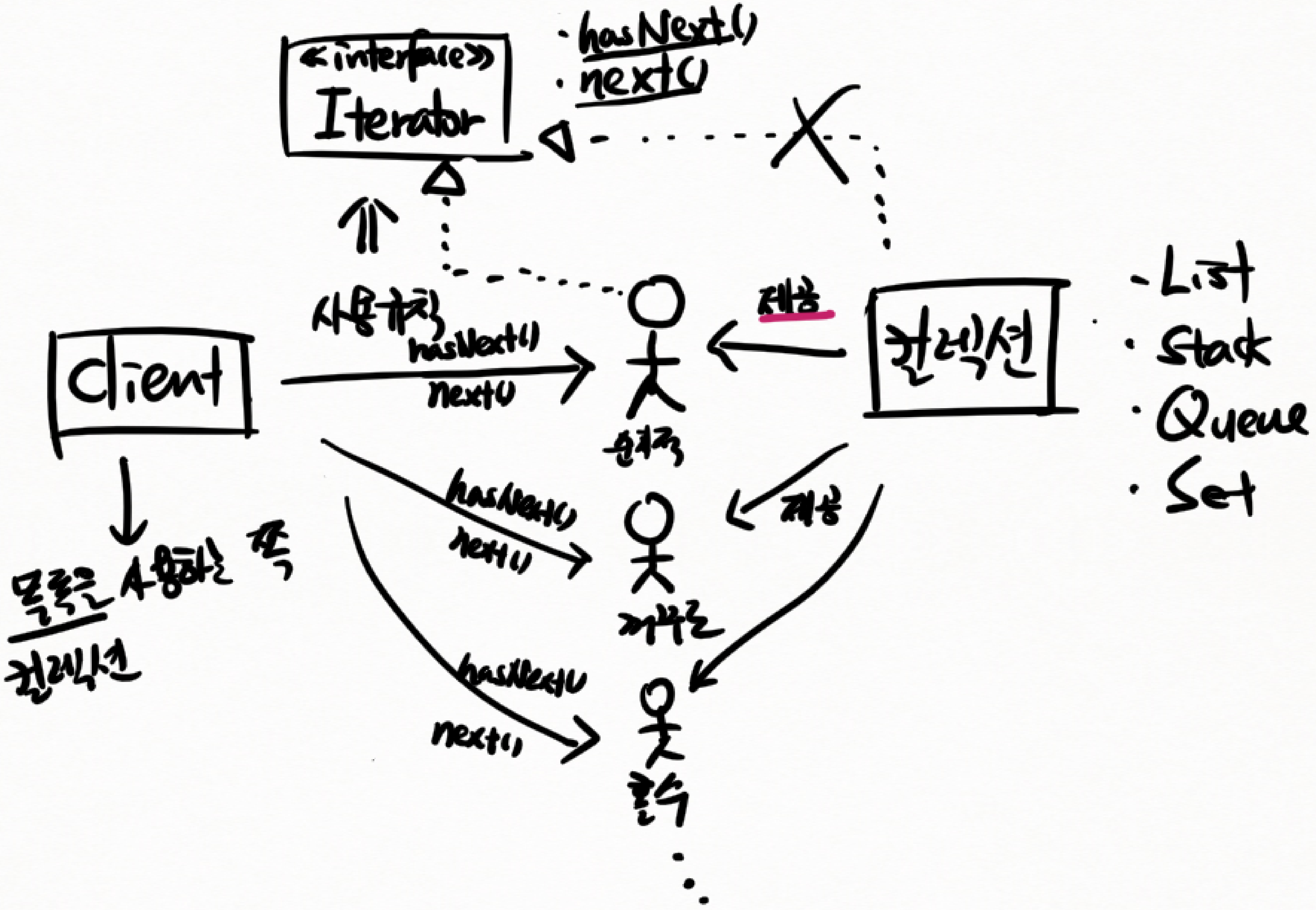
메서드 시그니처

⇊

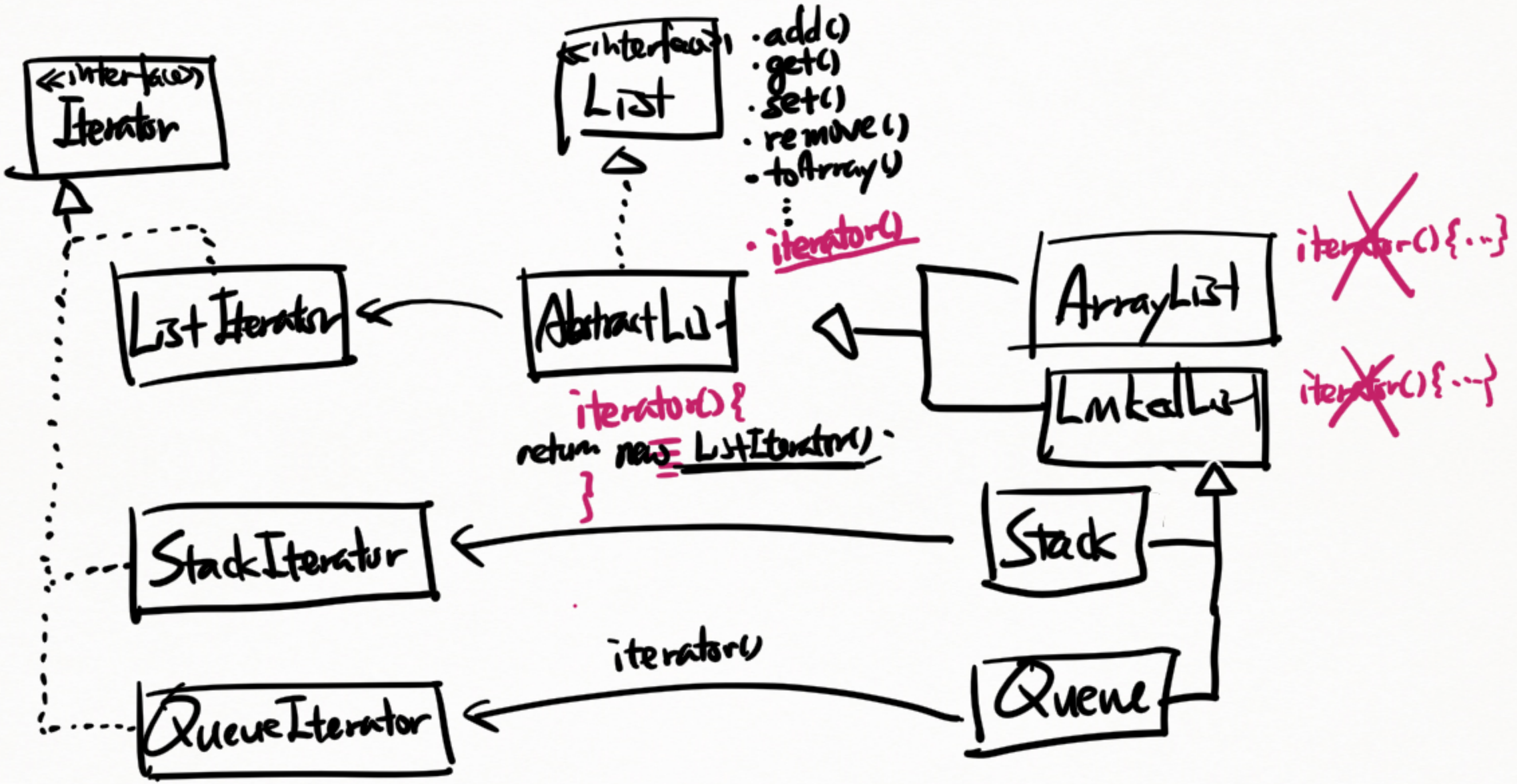
반환 규칙은 같다

⇊

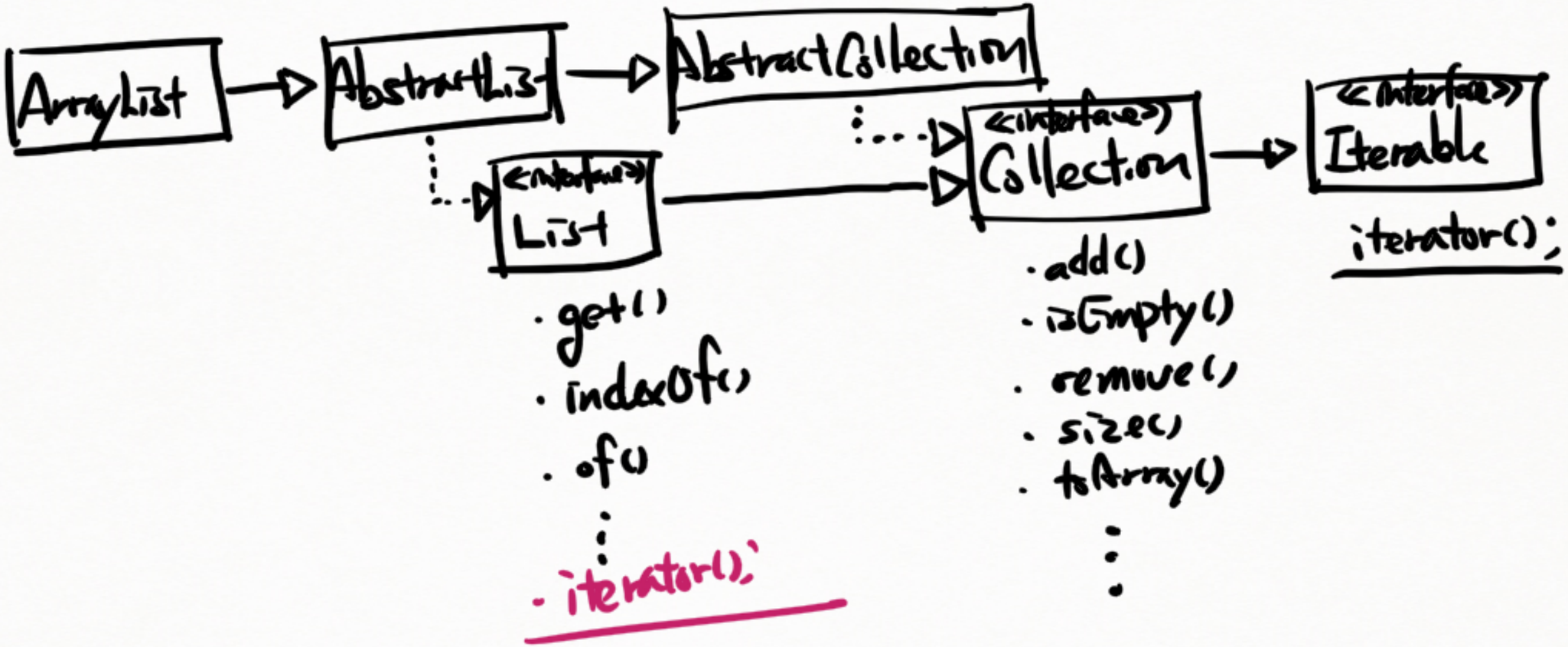
"interface"



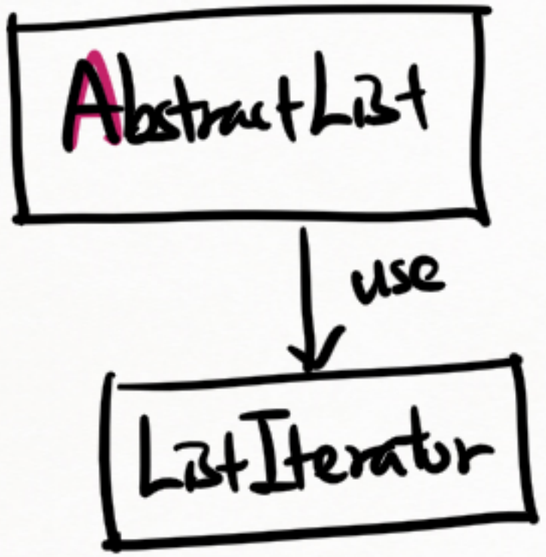
«interface»
Iterator
· hasNext()
· next()
사용규칙
hasNext()
next()
Client
순차적
제공
컬렉션
· List
· Stack
· Queue
· Set
hasNext()
next()
거꾸로
제공
hasNext()
next()
홀수
목록을 사용할 쪽
컬렉션

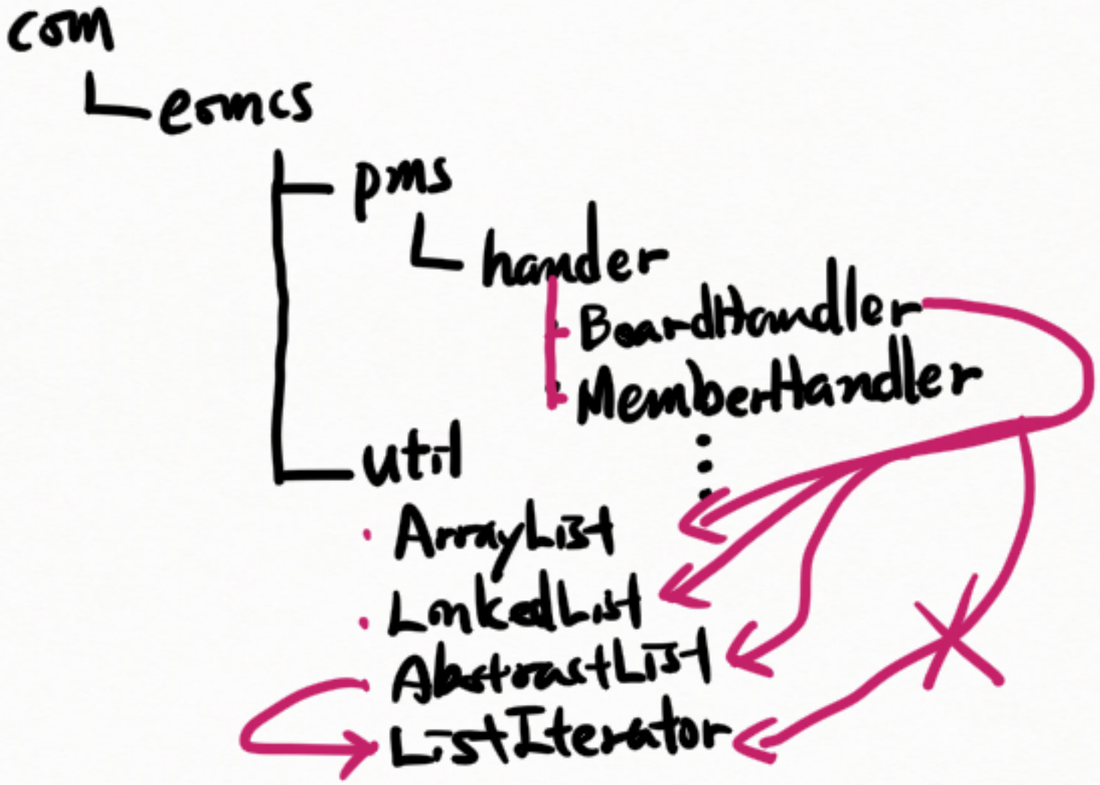

«interface»
Iterator
«interface»
List
.add()
.get()
.set()
.remove()
.toArray()
.iterator()
ListIterator
AbstractList
iterator(){
return new ListItr()
}
ArrayList
LinkedList
StackIterator
Stack
iterator()
QueueIterator
Queue

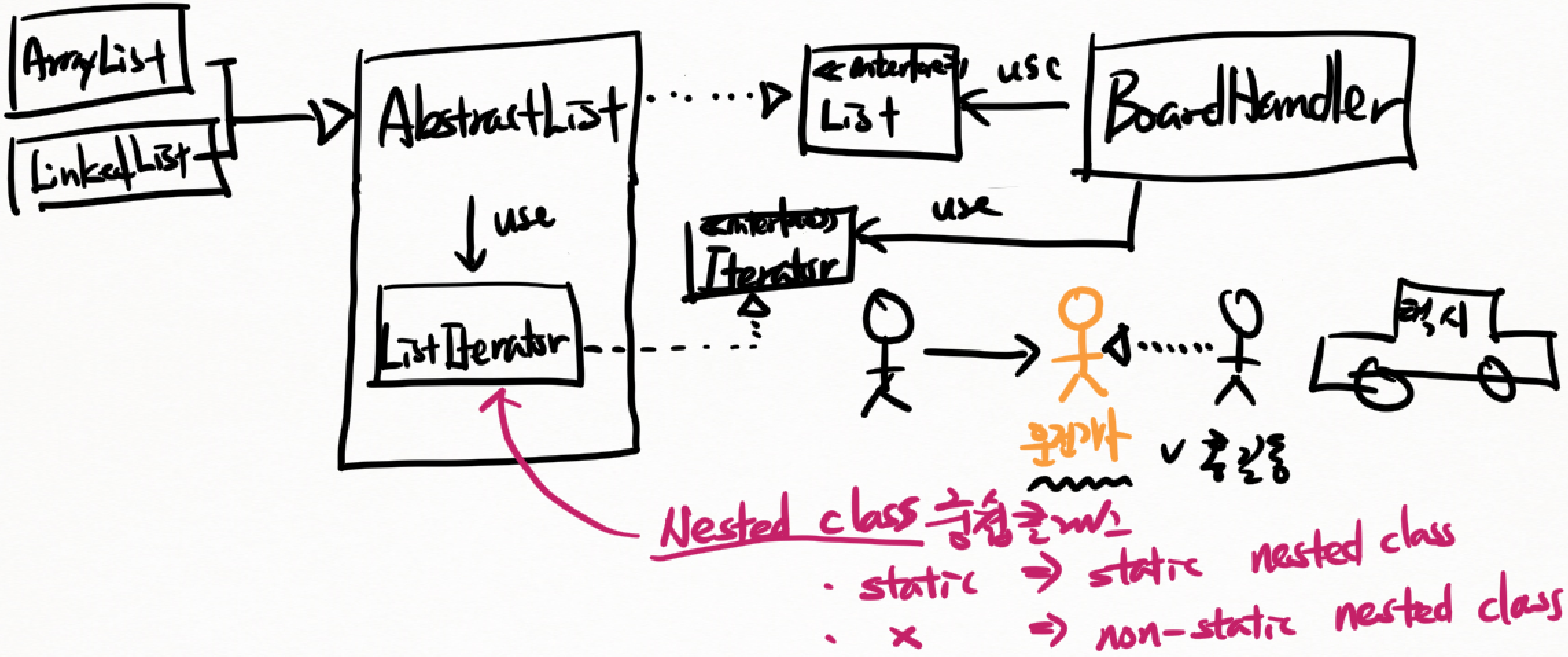
java.util.*

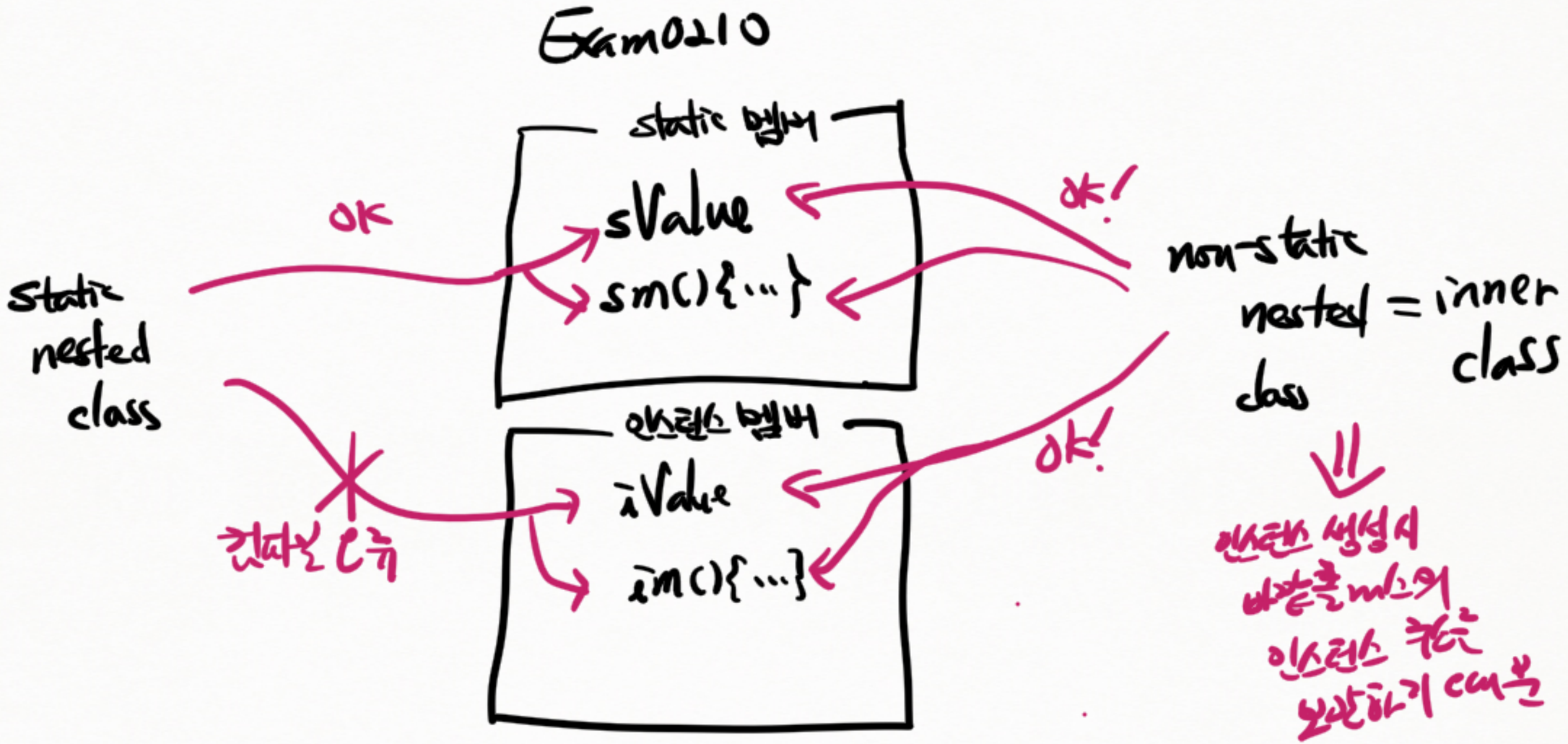

ArrayList
AbstractList
AbstractCollection
«interface»
List
«interface»
Collection
«interface»
Iterable
iterator();
.get()
.indexOf()
.of()
.iterator();
.add()
.isEmpty()
.remove()
.size()
.toArray()

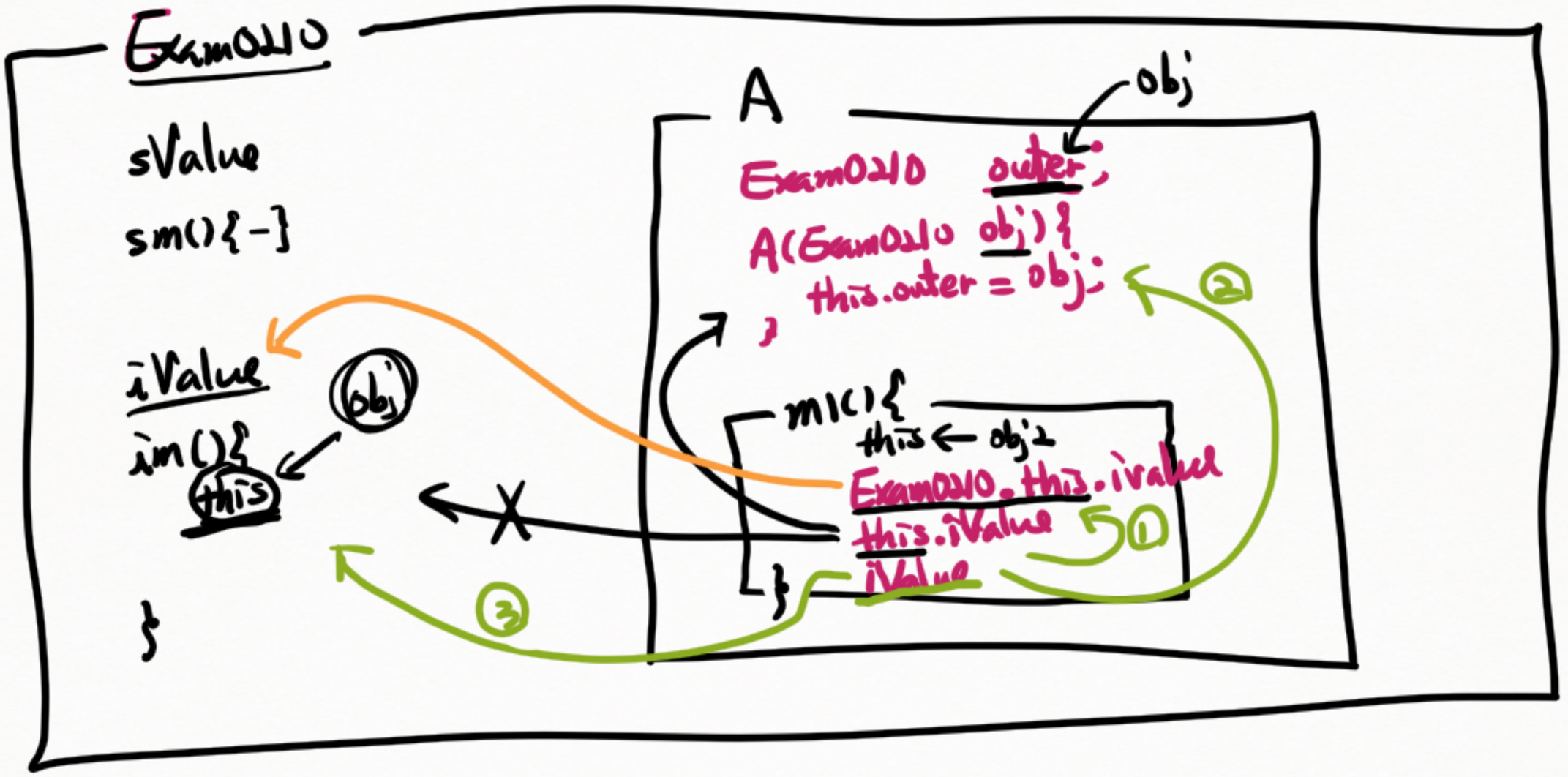
package member class
=
top-level class

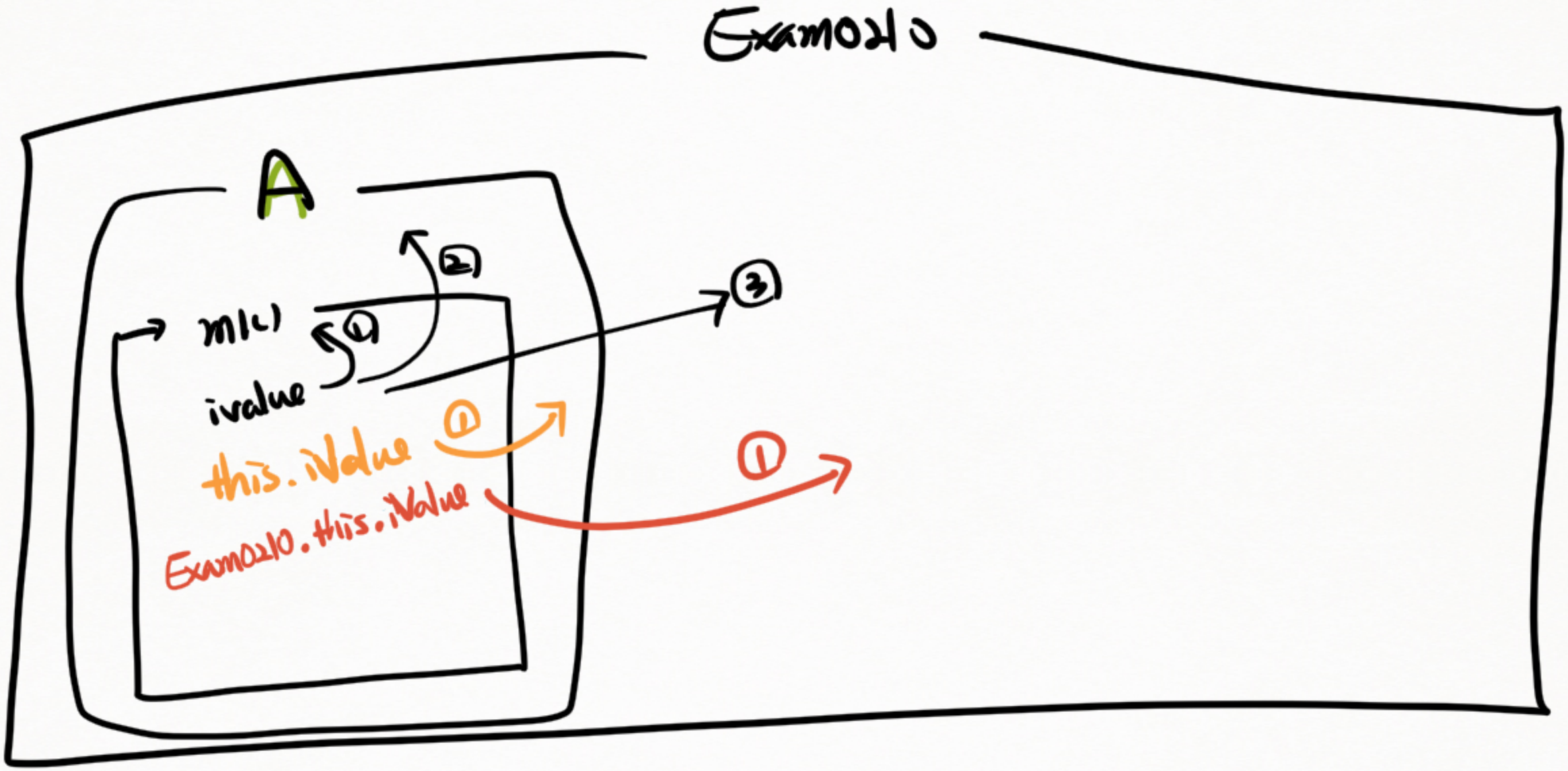





ArrayList
LinkedList
AbstractList
use
ListIterator
<<interface>>
List
use
BoardHandler
use
<<interface>>
Iterator
택시
운전기사
✓ 홍길동
Nested class 중첩클래스
· static → static nested class
· × → non-static nested class

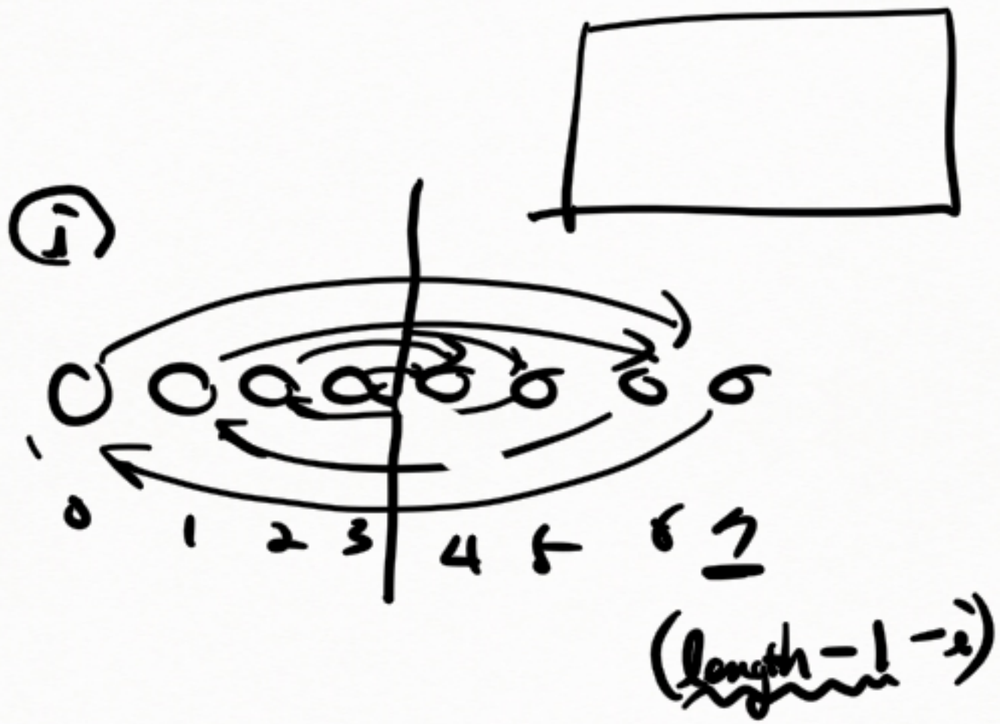

Exam0210
static 멤버
sValue
sm(){...}
인스턴스 멤버
iValue
im(){...}
ok
ok!
ok!
static nested class
컴파일 에러
non-static nested = inner class
인스턴스 생성시 바깥쪽 클래스의 인스턴스 참조 보관하기 때문

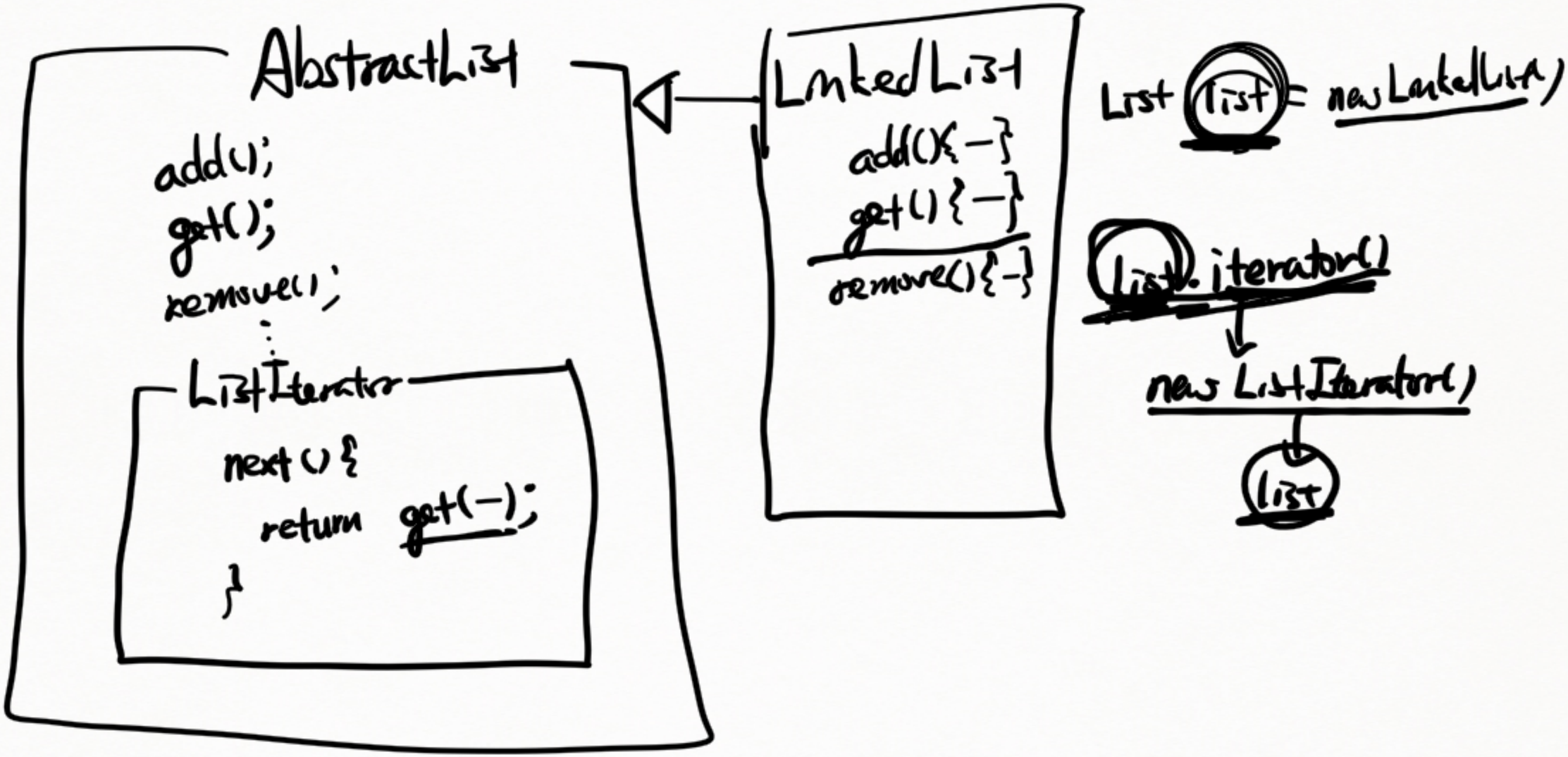

Example
sValue
sm(){-}
iValue
im(){
this
obj
}
A
obj
Example outer;
A(Example obj){
this.outer = obj;
}
m1(){
this ← obj2
Example.this.iValue
this.iValue
iValue
}
1
2
3

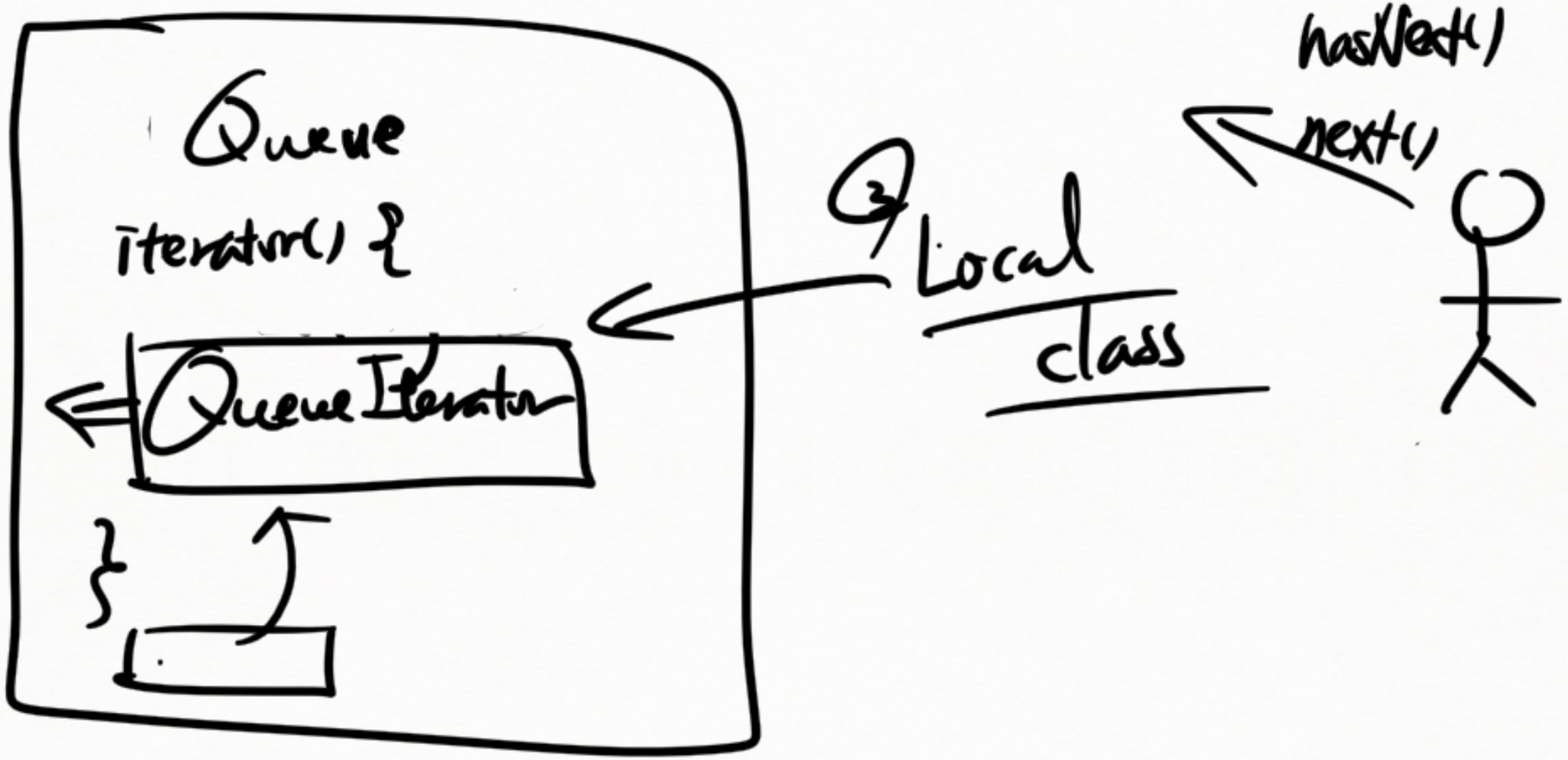

Example
A
2
main()
1
3
iValue
this.iValue
1
Example.this.iValue
1

JVM Stack

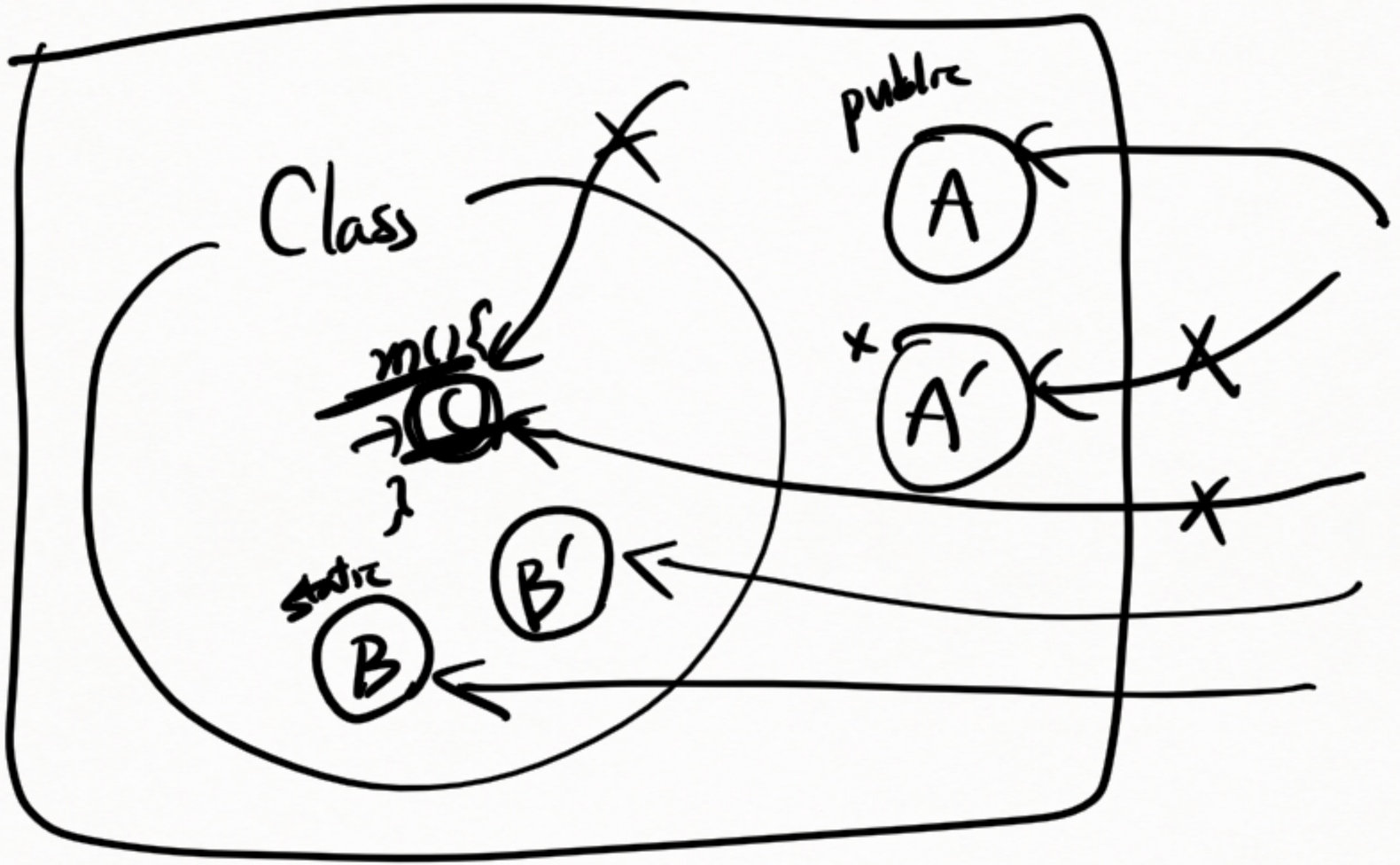
obj.m1()

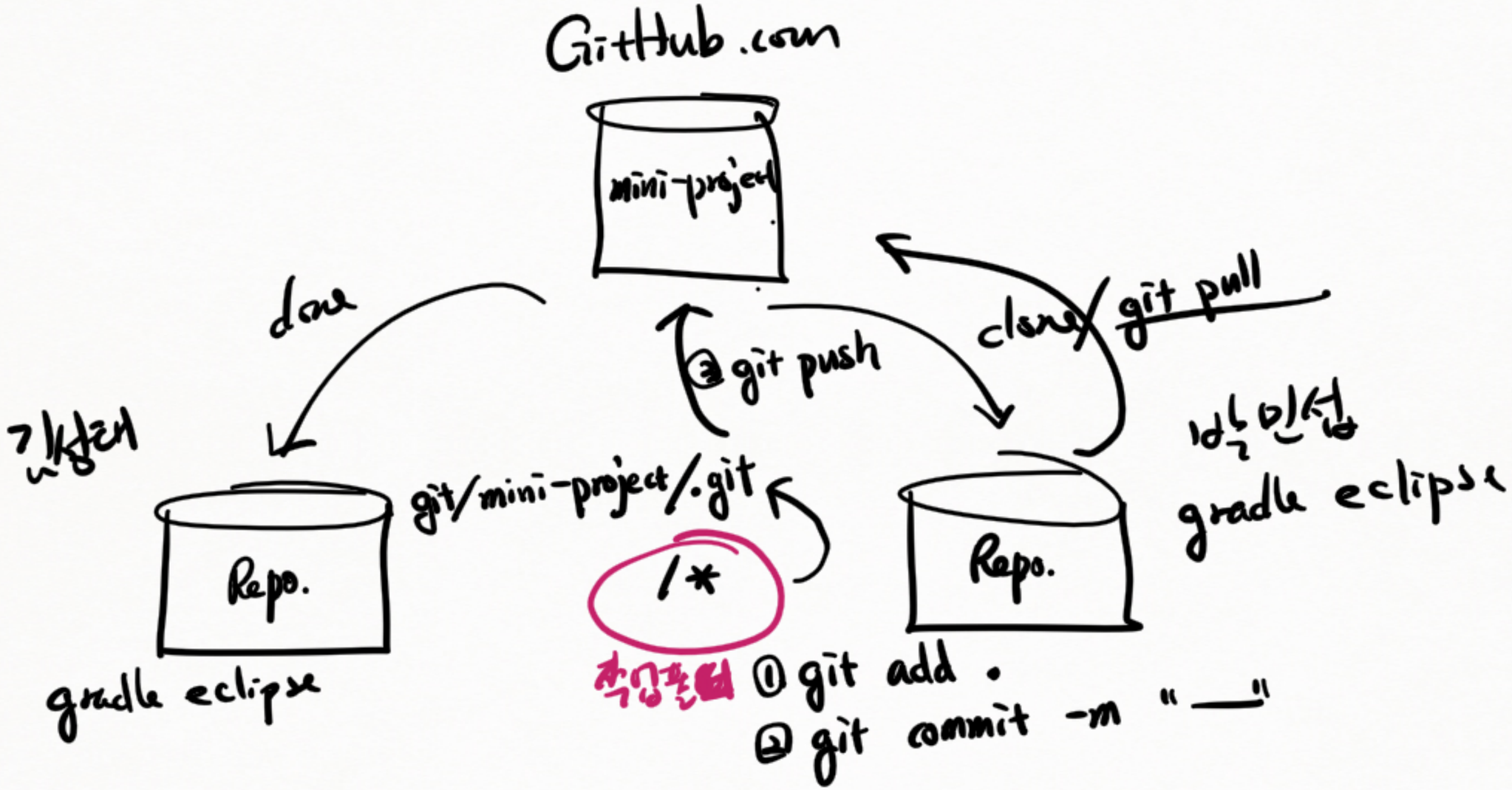


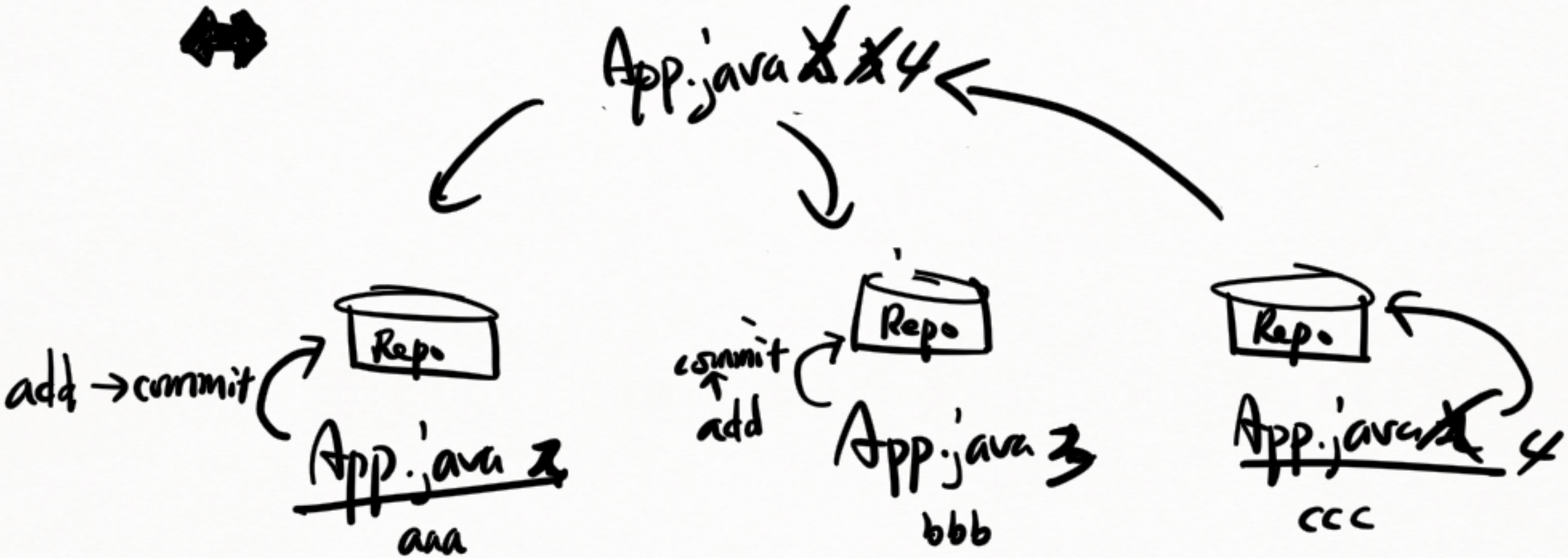

AbstractList
add();
get();
remove();
ListIterator
next() {
return get(-);
}
LinkedList
add(){-}
get(){-}
remove(){-}
List list = new LinkedList()
list.iterator()
new ListIterator()
list

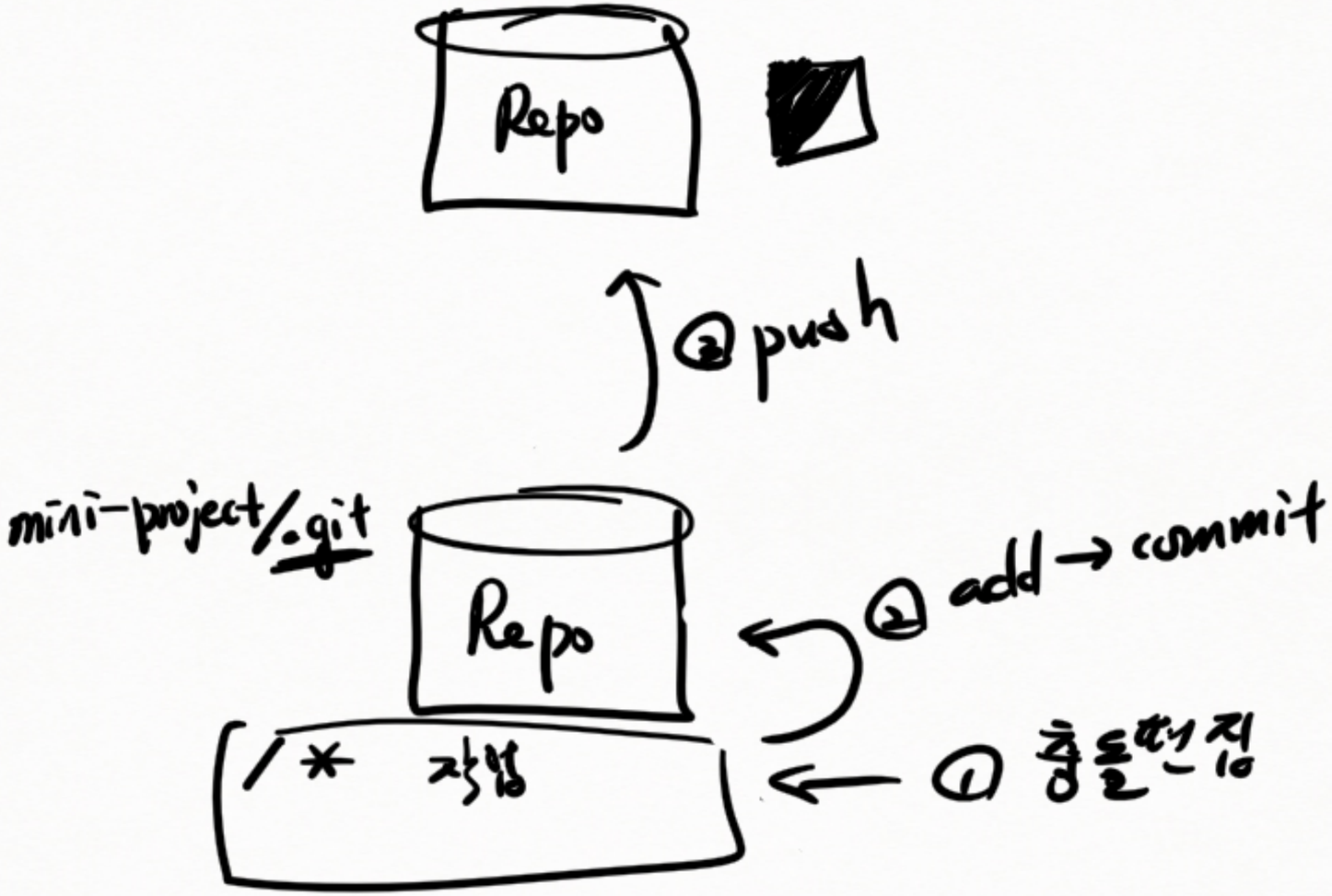

Queue
iterator() {
QueueIterator
}
Local
class
hasNext()
next()

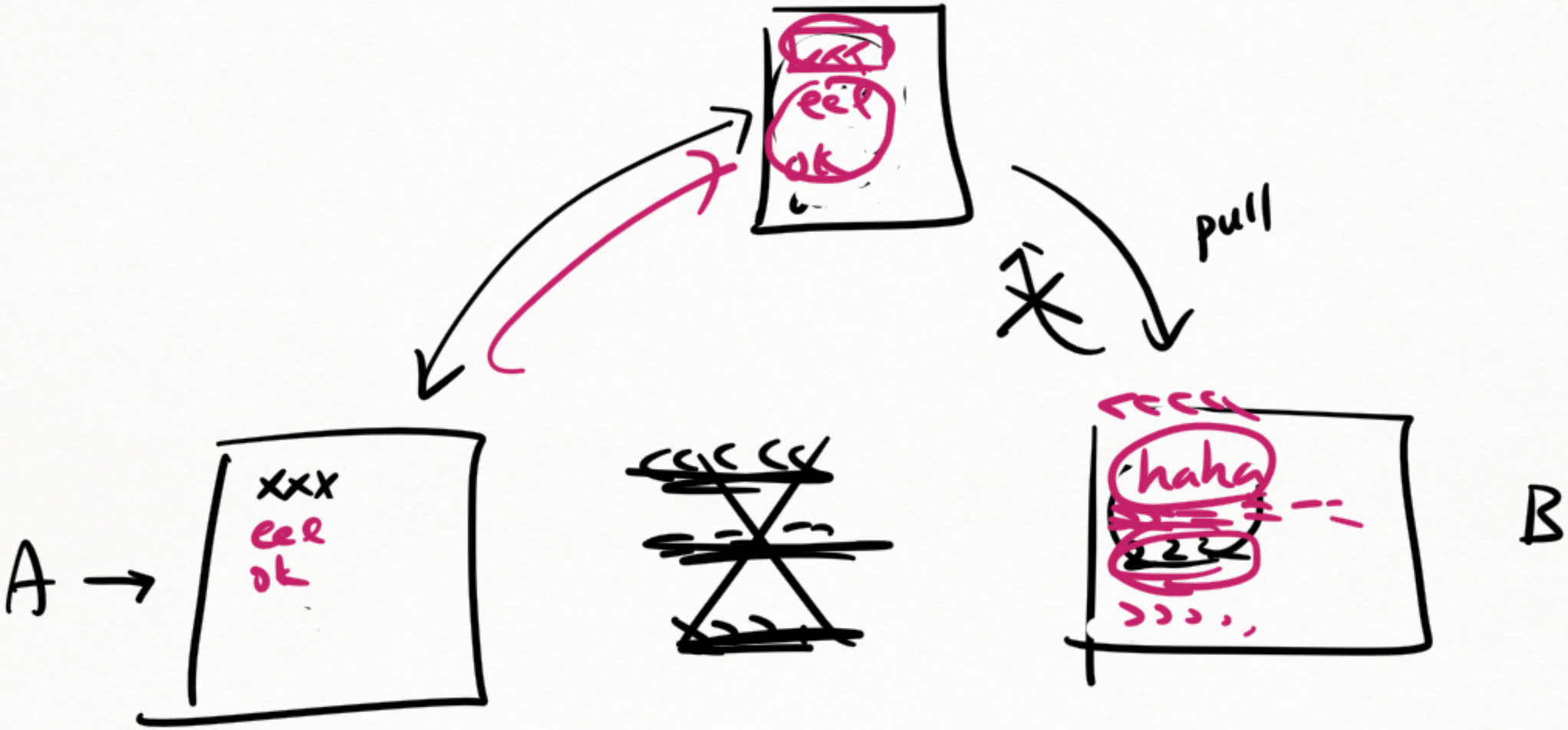
m();




GitHub.com
mini-project
clone
git push
clone/git pull
Repo.
git/mini-project/.git
Repo.
gradle eclipse
/*
① git add .
② git commit -m "—"


App.java
Repo
add → commit
App.java
aaa
Repo
commit
add
App.java
bbb
Repo
App.java
ccc

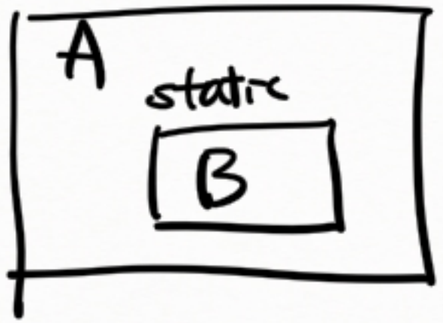

Repo
② push
mini-project/.git
Repo
② add → commit
/* 작업
① 충돌없앰

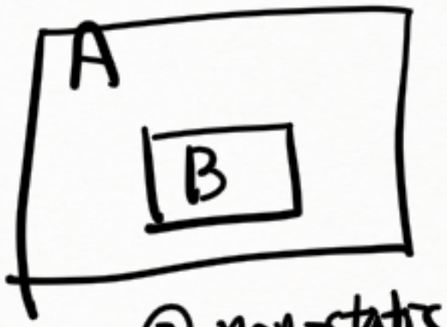

pull
A
xxx
haha
B

top-level class
=
② static nested class

com.eomcs

① top-level class { A, B, 13 }





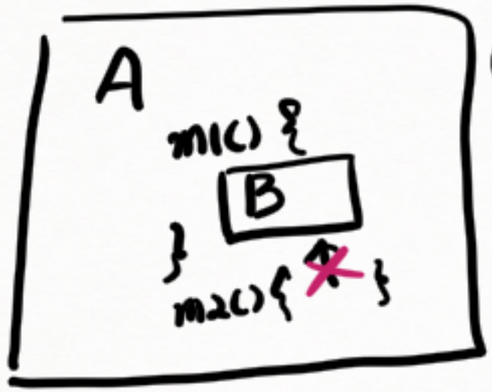
③ non-static nested class
=
inner class

new ~~com.eomcs~~.B();
import com.eomcs.B;

A.B ob = new ~~com.eomcs~~.A.B();
import com.eomcs.A.B;

```
A outer = new A();
A.B ob = outer.new B();
```

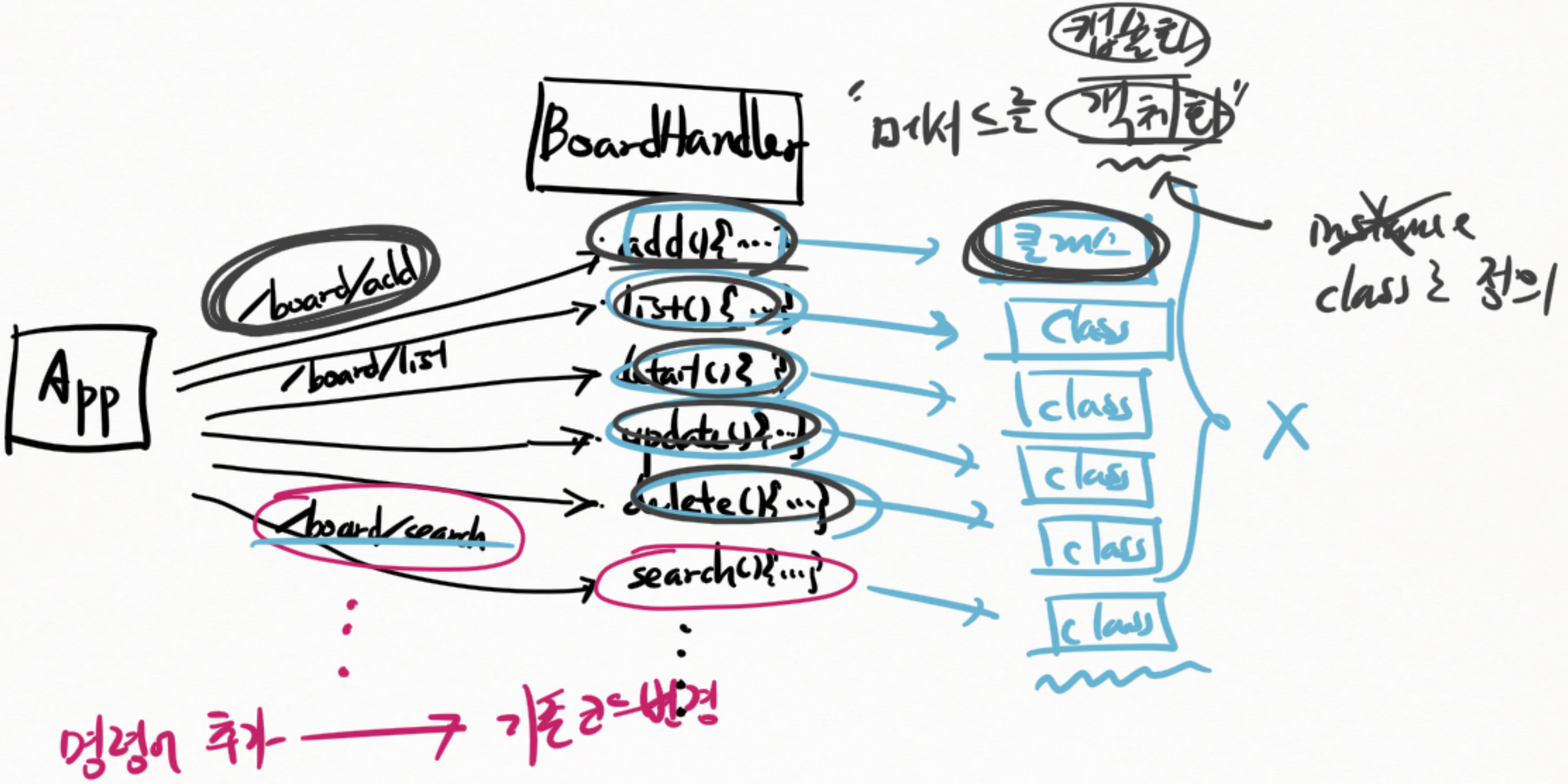
A$1B.class

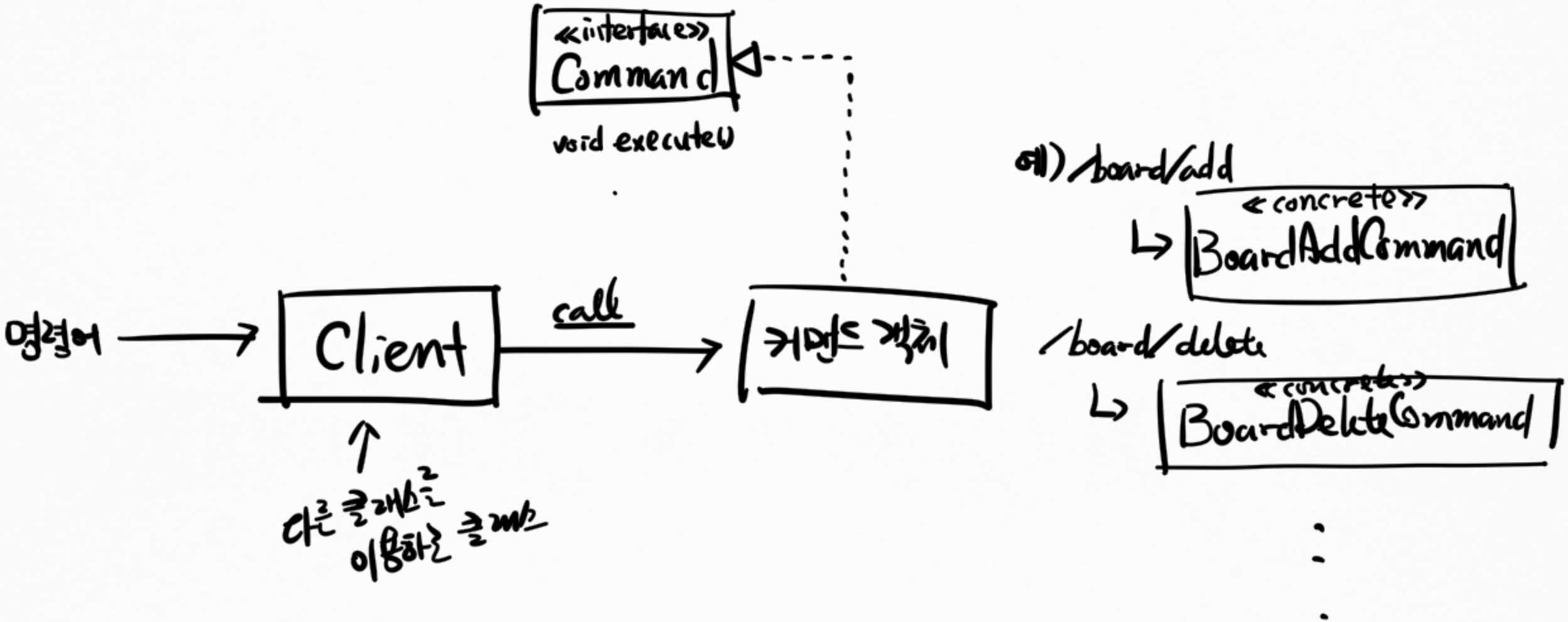


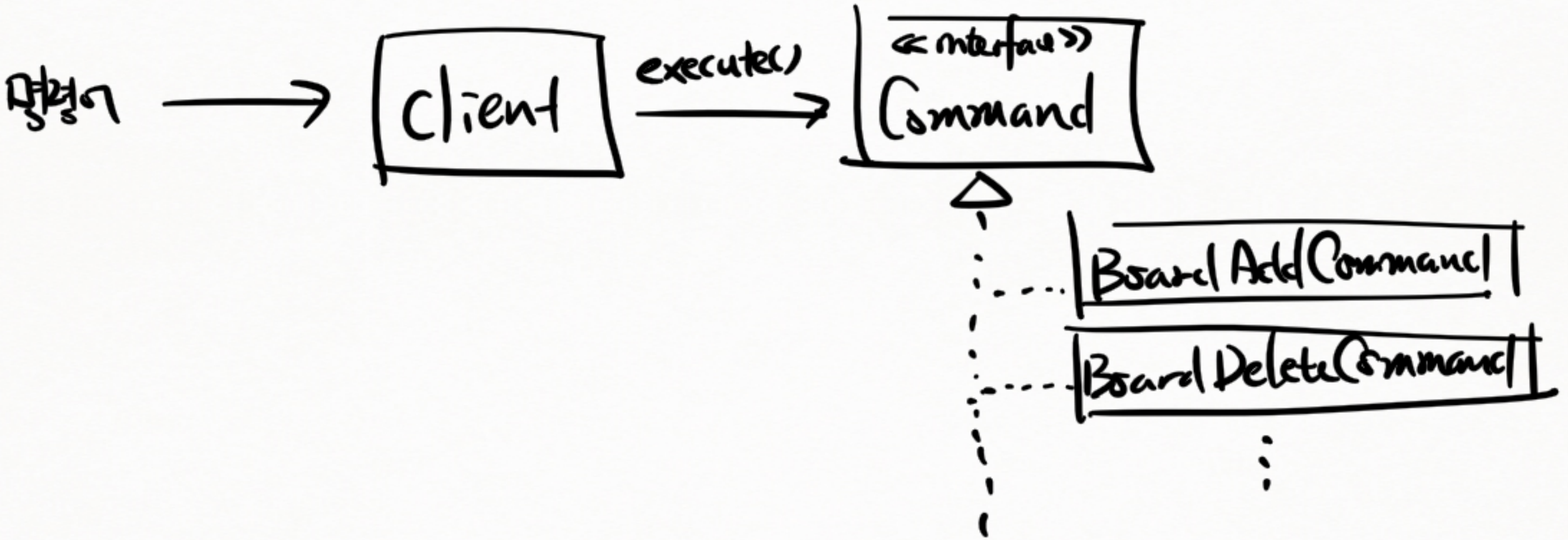
④ local class

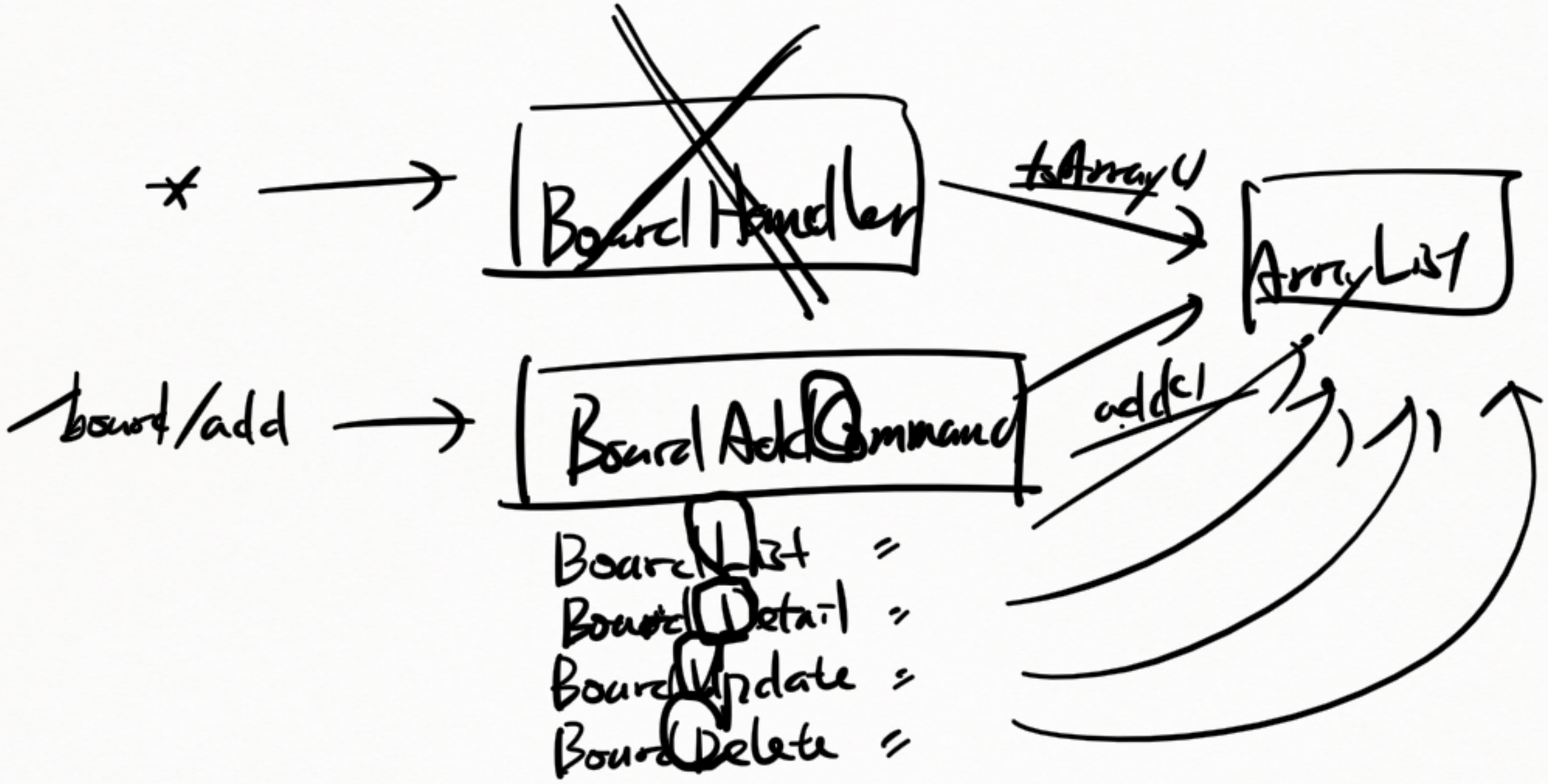
✓

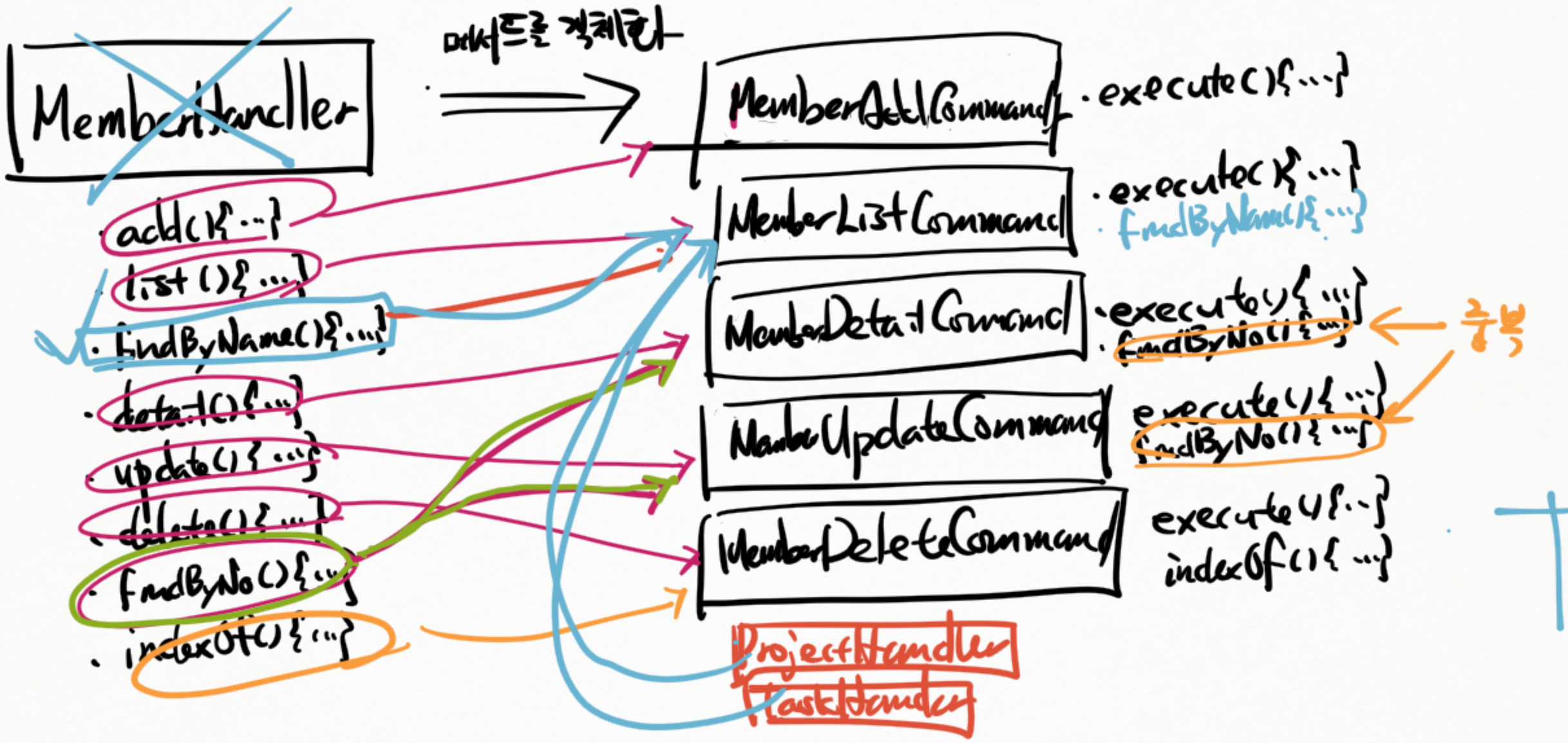
⑤ anonymous class
?


BoardHandler
"파사드를 적용한"
/board/add
/board/list
/board/search
App
add(){...}
list(){...}
start(){...}
update(){...}
delete(){...}
search(){...}
클래스
class
class
class
class
class
X
명령어 추가 → 기존코드변경


«interface»
Command
void execute()
예) /board/add
«concrete»
BoardAddCommand
명령어
Client
call
커맨드 객체
/board/delete
«concrete»
BoardDeleteCommand
다른 클래스는
이용하는 클래스


요청
Client
execute()
«interface»
Command
BoardAddCommand
BoardDeleteCommand


*
BoardHandler
toArrayV
ArrayList
board/add
BoardAddCommand
addV
BoardList =
BoardDetail =
BoardUpdate =
BoardDelete =


MemberHandler
메서드를 객체로
add(){...}
list(){...}
findByName(){...}
update(){...}
delete(){...}
findByNo(){...}
indexOf(){...}
MemberAddCommand
execute(){...}
MemberListCommand
execute(){...}
findByName(){...}
MemberDetailCommand
execute(){...}
findByNo(){...}
중복
MemberUpdateCommand
execute(){...}
findByNo(){...}
MemberDeleteCommand
execute(){...}
indexOf(){...}
ProjectHandler
TaskHandler