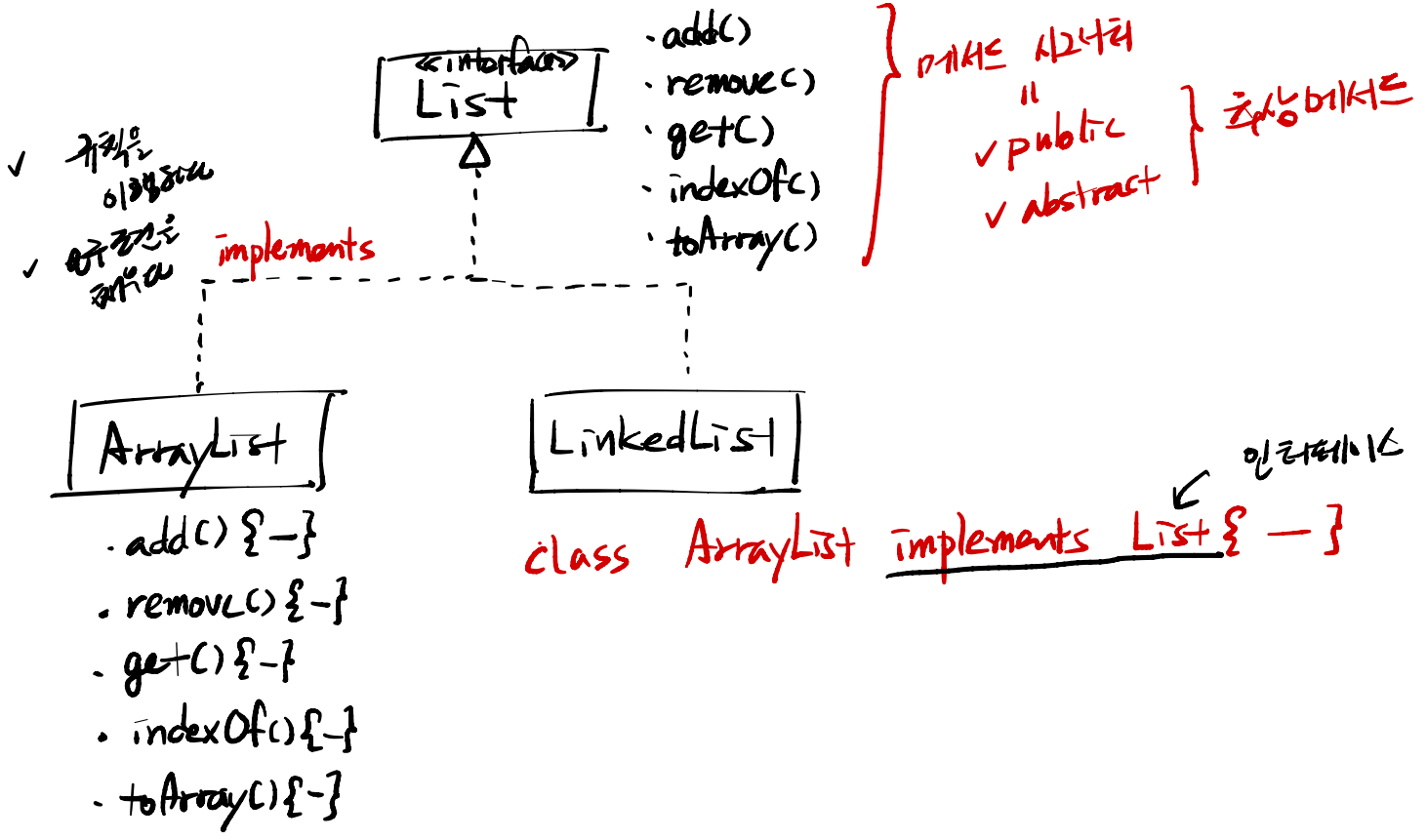
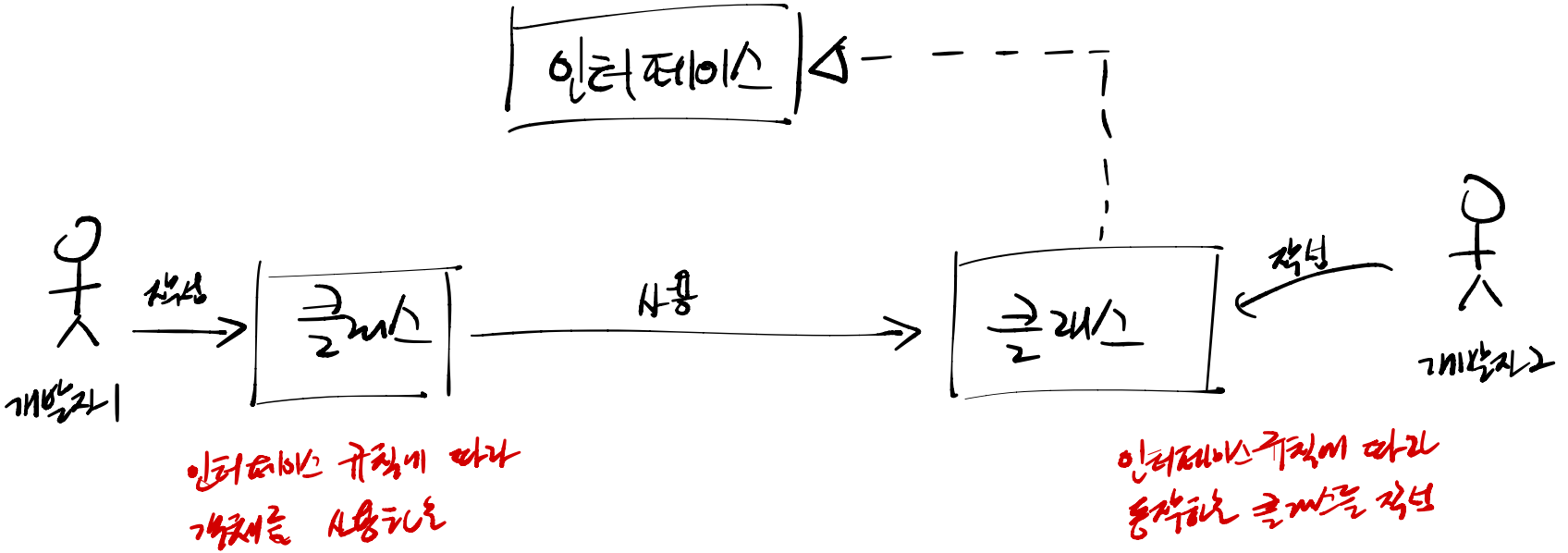
17. 인터페이스를 이용 객체의 사용규칙 정의

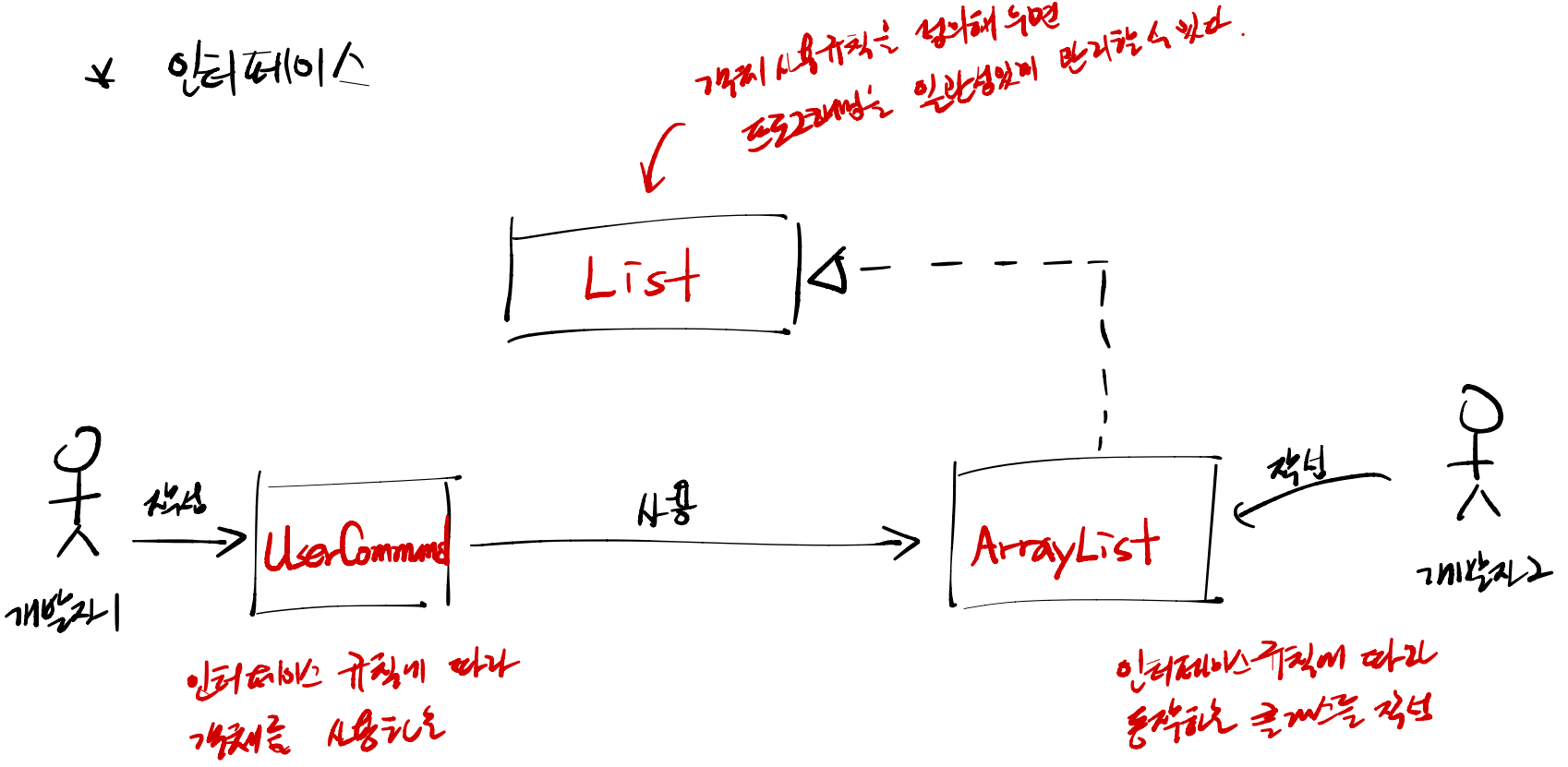


✓ 인터페이스

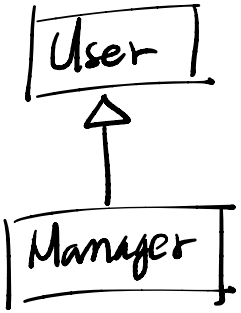
객체 사용규칙을 정의해 두면
① 프로그램을 일관성있게 관리할 수 있다.
② 고치기 쉽다



✓ 인터페이스


객체 사용규칙을 정의해 두면
프로그램을 일관성있게 관리할 수 있다.
List
개발자1
참조
UserCommand
사용
ArrayList
작성
개발자2
인터페이스 규칙에 따라
객체를 사용하는
클래스 작성
인터페이스규칙에 따라
동작하는 클래스를 작성

* instanceof vs getClass()



equals() {
≡
}

```
User u1 = new User();
String str = new String();
User u2 = new User();
Manager m = new Manager();
```

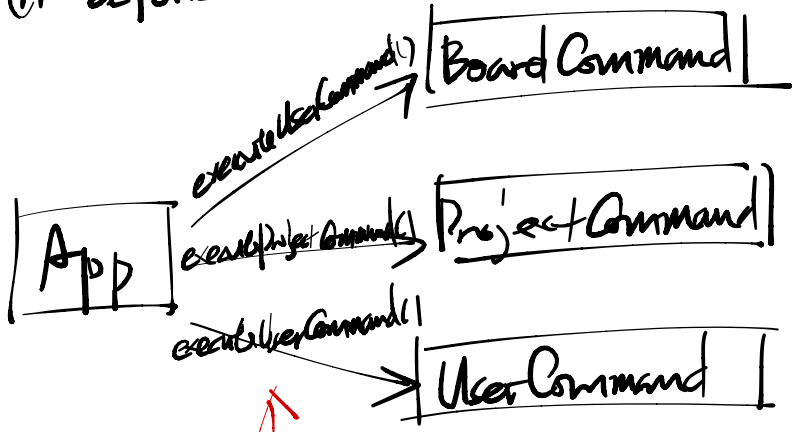
```
equals(Object obj) {
    if (this.getClass() != obj.getClass()){   // 방법 ①
        return false;
    }
    if(!(obj instanceof User)){               // 방법 ②
        return false;
    }
```

	방법 1	방법 2
u1.equals(str)	F	F
u1.equals(u2)	T	T
u1.equals(m)	F	T

* 메뉴처리 객체의 사용규칙 통일

(클래스)

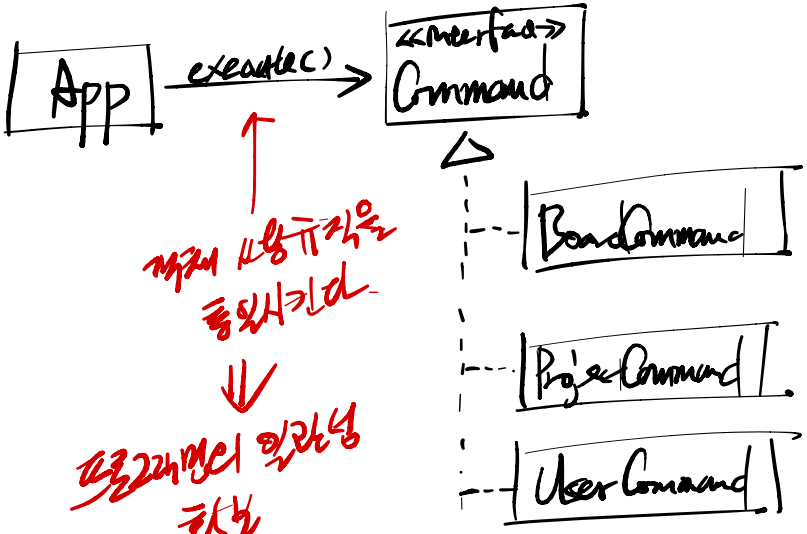
① before



사용방법이 다르다

(메서드 종류)

② after



객체 사용규칙을 통일시킨다.

⇓

프로그램의 확장성 향상

⇓

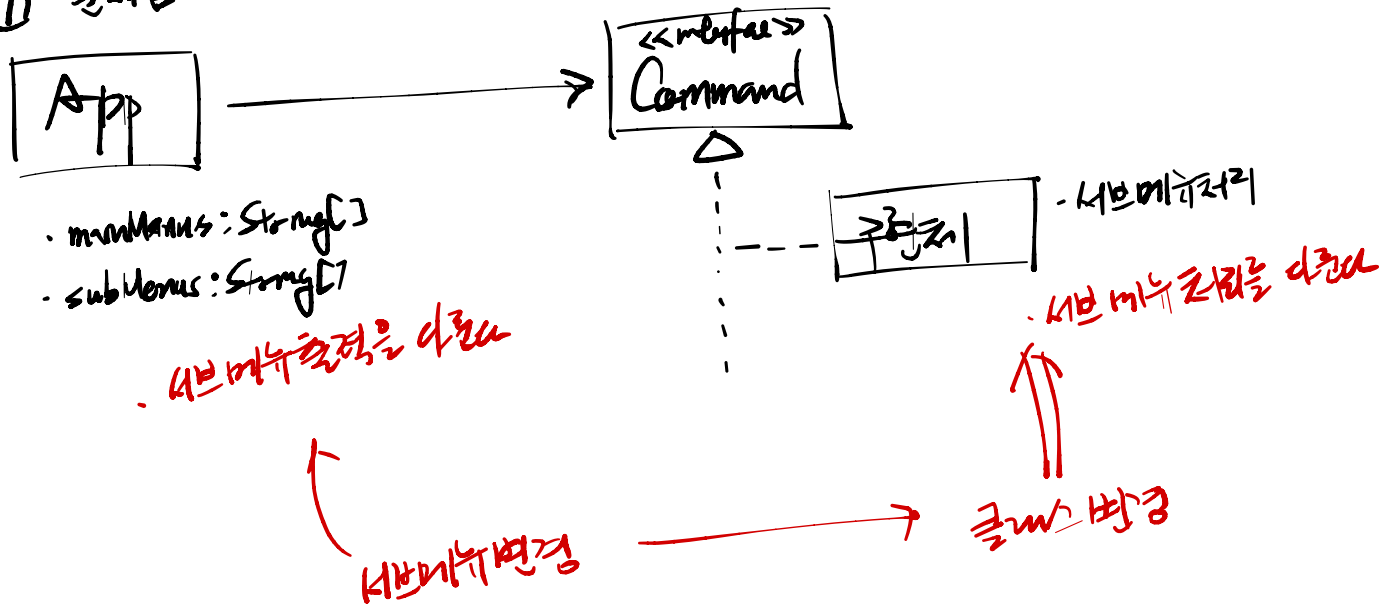
유지보수가 용이하다 = 코드를 이해하기 쉽다.

⇓

비용감소

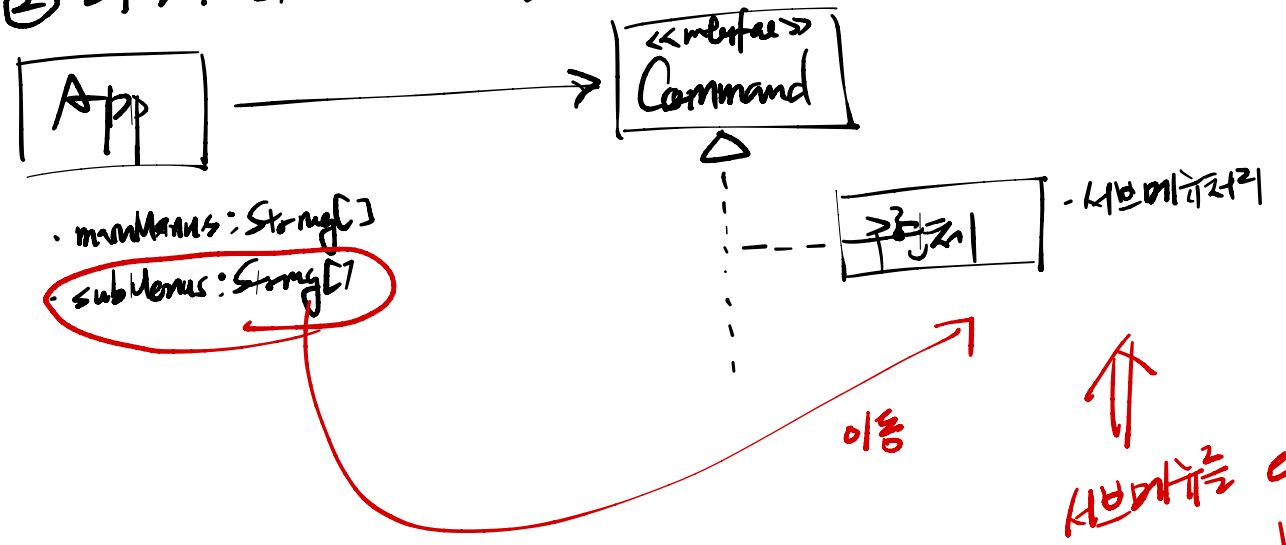
18. 리팩토링

① 특강


App
<<interface>>
Command
- mainMenus : String[]
- subMenus : String[]
- 서브메뉴 출력을 담당
- 서브메뉴 변경
- 서브메뉴 처리
- 서브메뉴 처리를 담당
- 클래스 변경

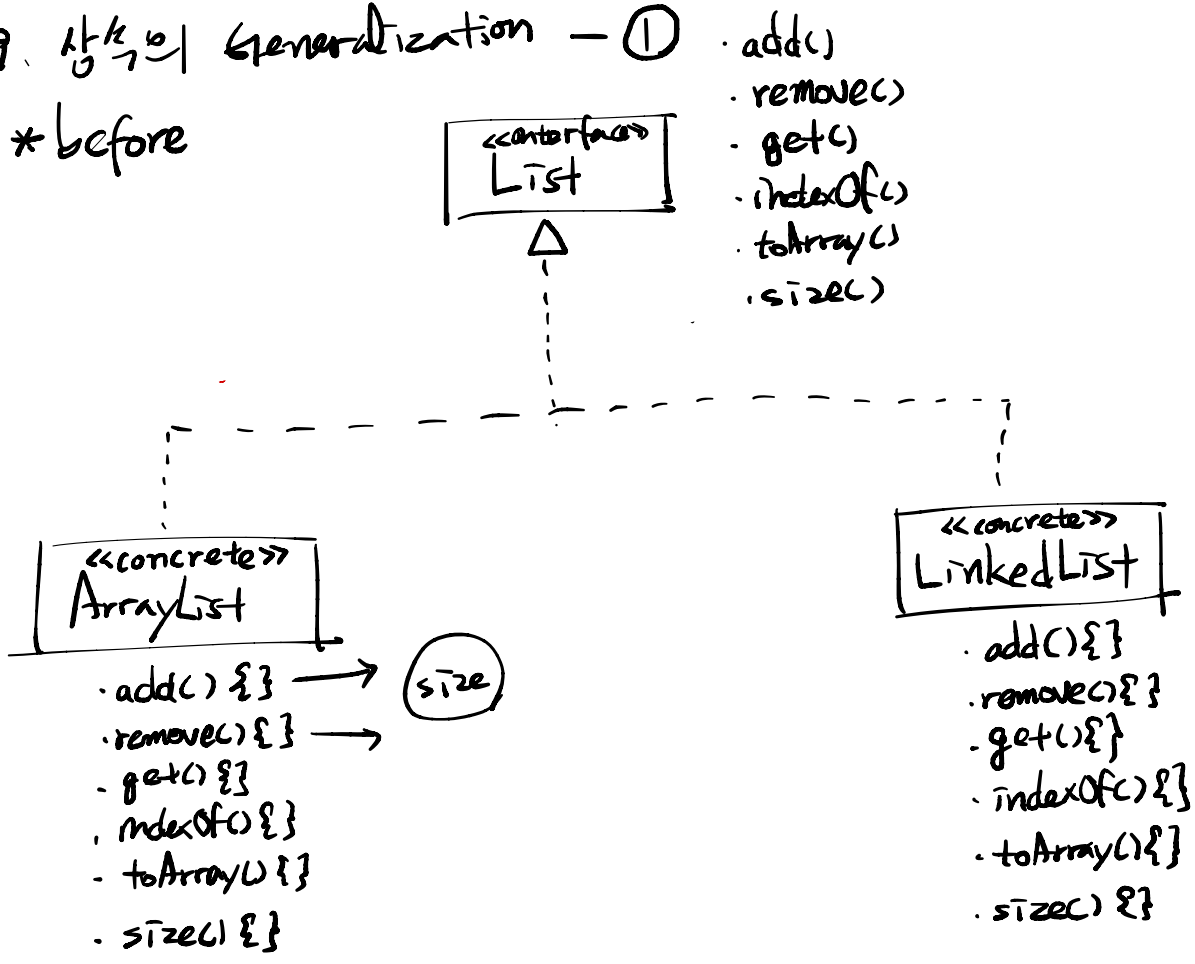
18. 리팩토링

② 해결 : GRASP의 High Cohesion 원칙 적용



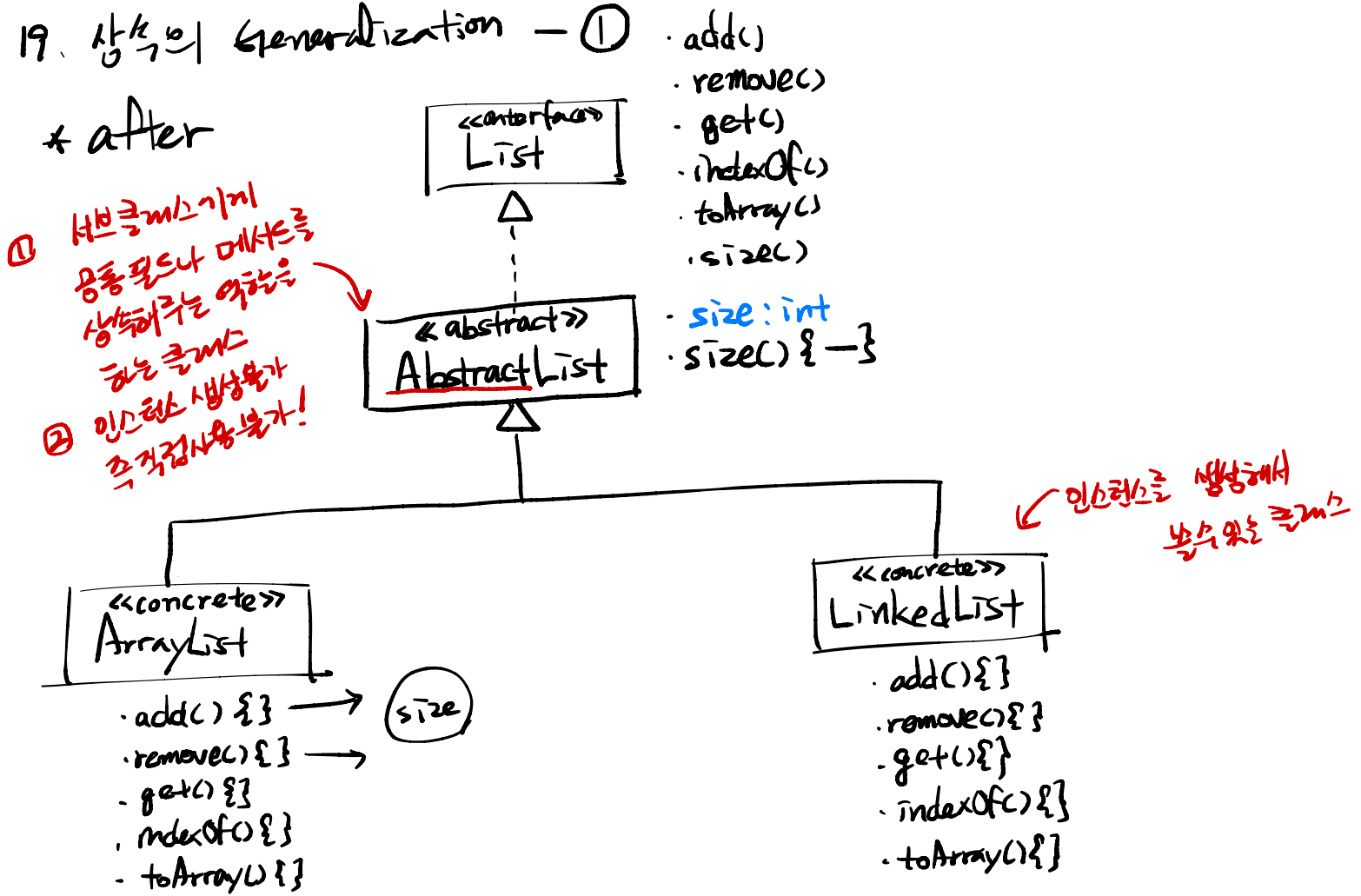
19. 상속의 Generalization — ①

*before



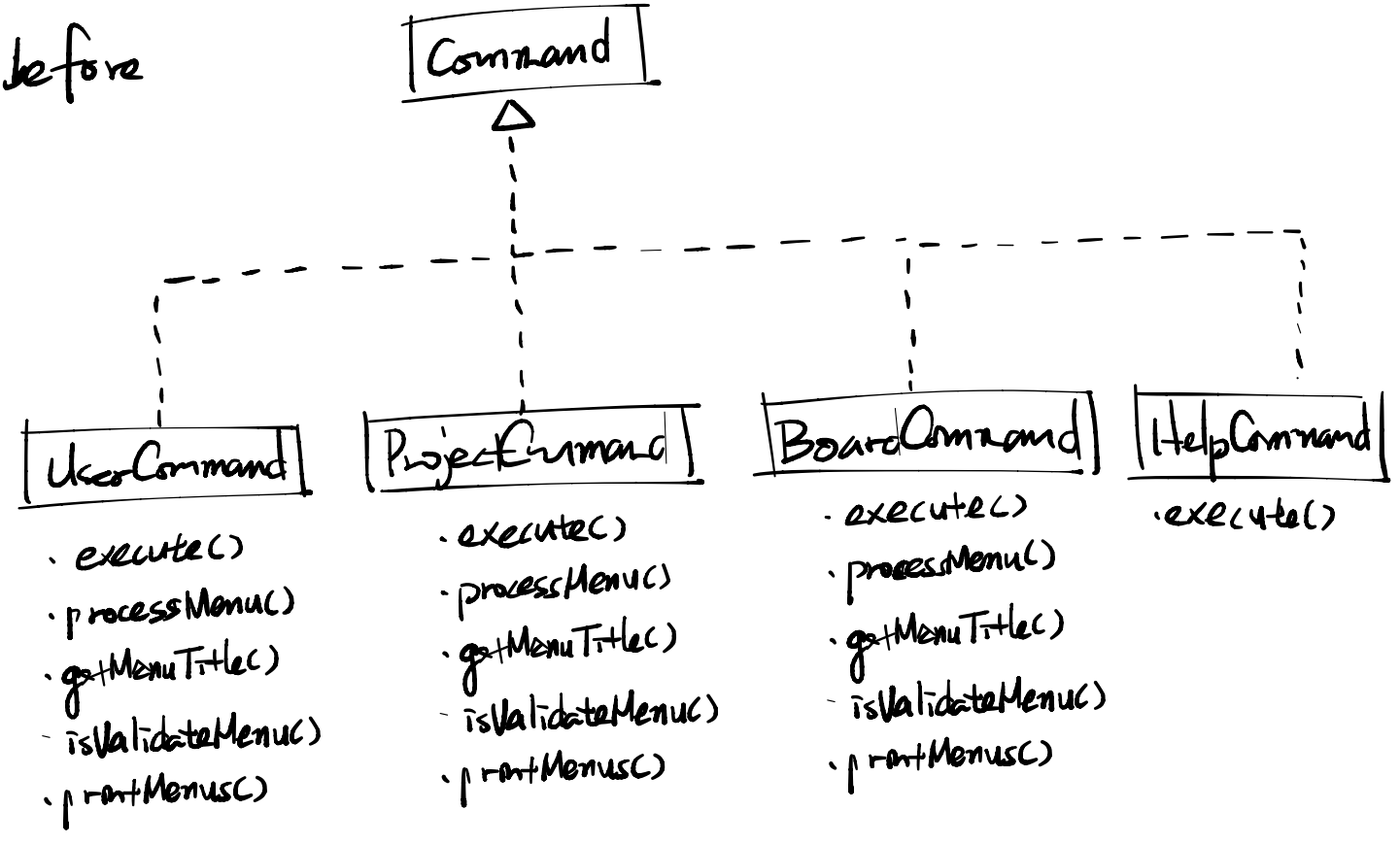
19. 상속의 Generalization — ①

* after



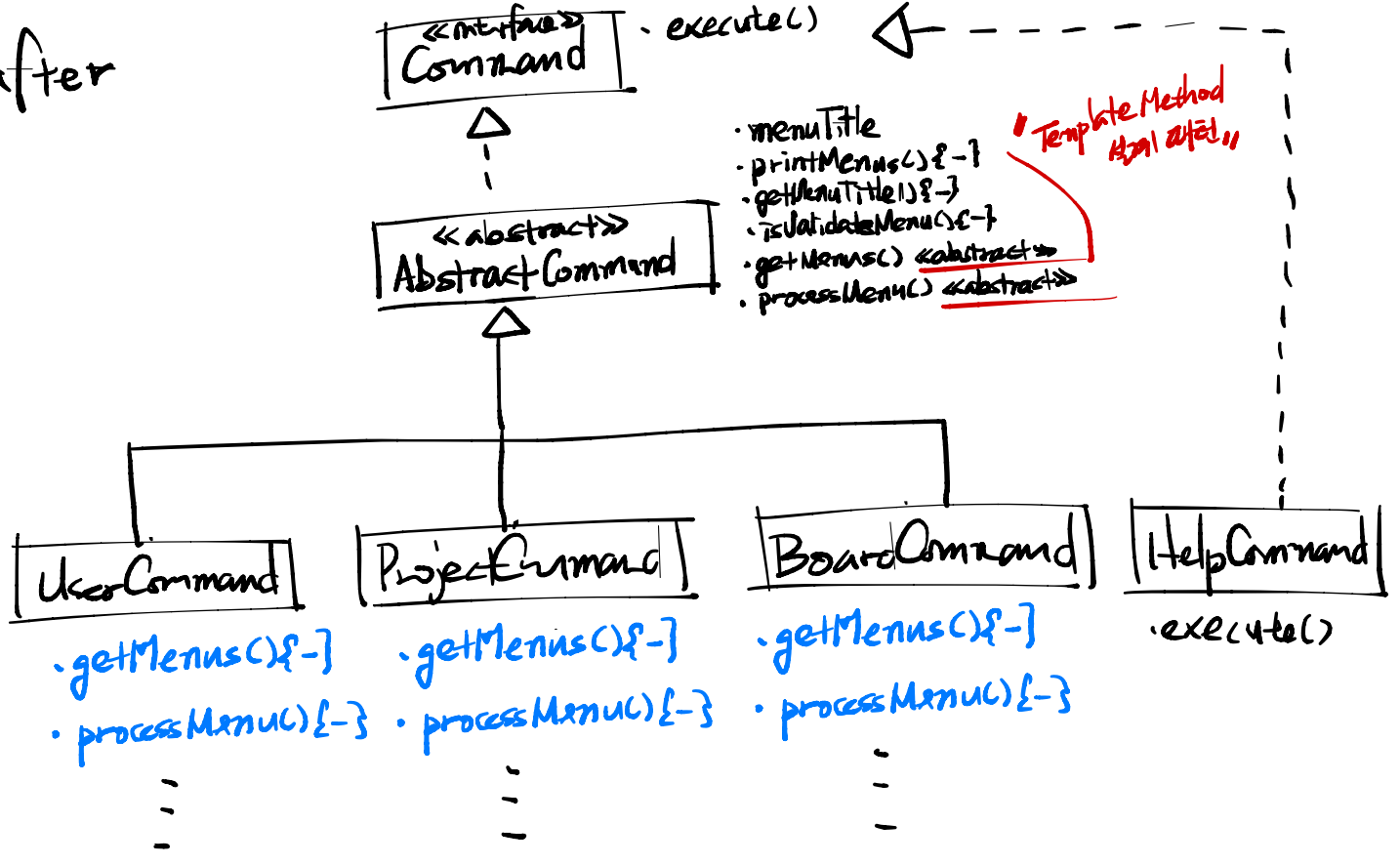
19. 상속의 Generalization — ②

* before

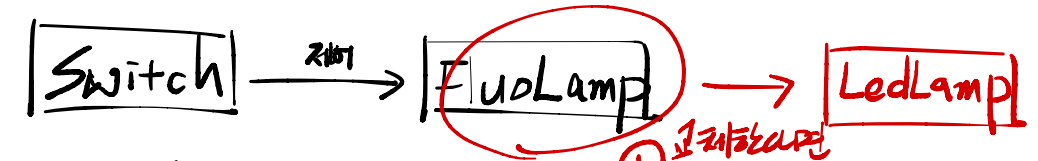


19. 상속의 Generalization — ②

* after



20. SOLID의 DIP와 GRASP의 Low Coupling




```
class Switch {
    FluoLamp light;
    ≡
}
```

② 코드를 변경해야 한다.

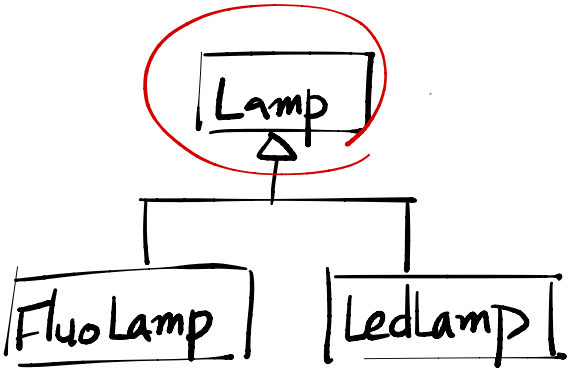
③ Switch 클래스가 FluoLamp 클래스만 사용하도록 되어있다

④ "강한 연결 상태"

⇒ 해결?

20. SOLID의 DIP와 GRASP의 Low Coupling


```
class Switch {
    Lamp light;
}
```



② 다형적 변수를 활용하여 여러 종류의 전등을 제어할 수 있다

① 두 클래스의 부모를 같게하여 같은 타입으로 묶으면

③ 전등만 제어?

Lamp

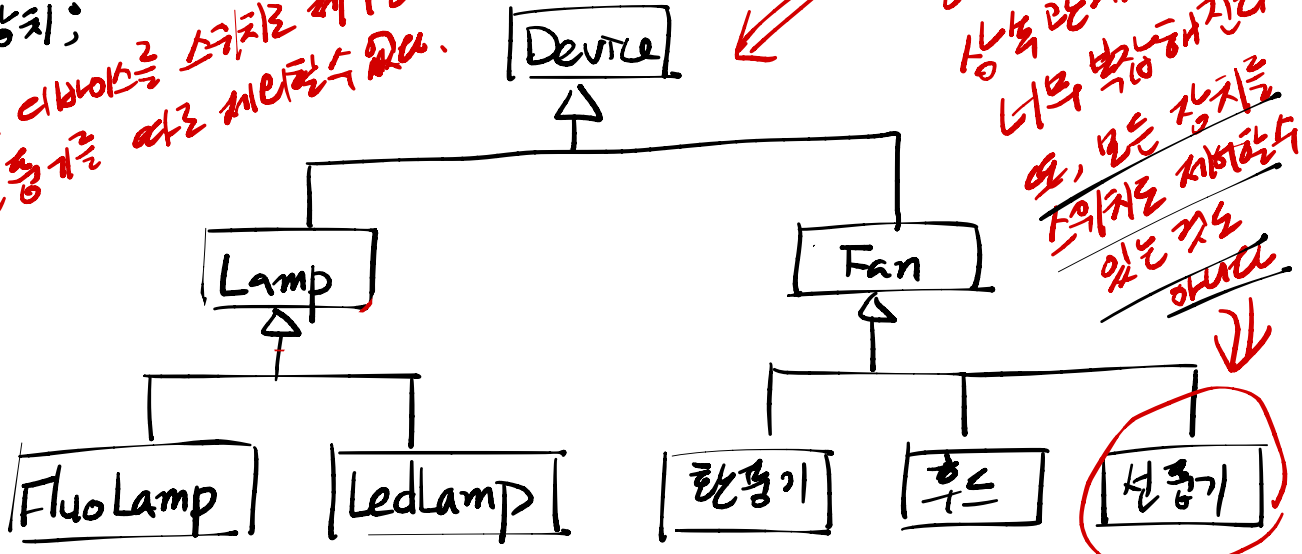
⇒ ④ 스위치로 다른 제품을 제어할 수 있어야 하는것 아닌가?

20. SOLID의 DIP와 GRASP의 Low Coupling


```
class Switch {
    Device 장치;
}
```

② 모든 디바이스를 스위치로 제어한다 가정하면 선풍기를 따로 제어할 수 없다.

① 여러 장치를 스위치로 제어하려고 같은 타입으로 묶어버면 상속 관계가 너무 복잡해진다 또, 모든 장치를 스위치로 제어할 수 있는 것은 아니다



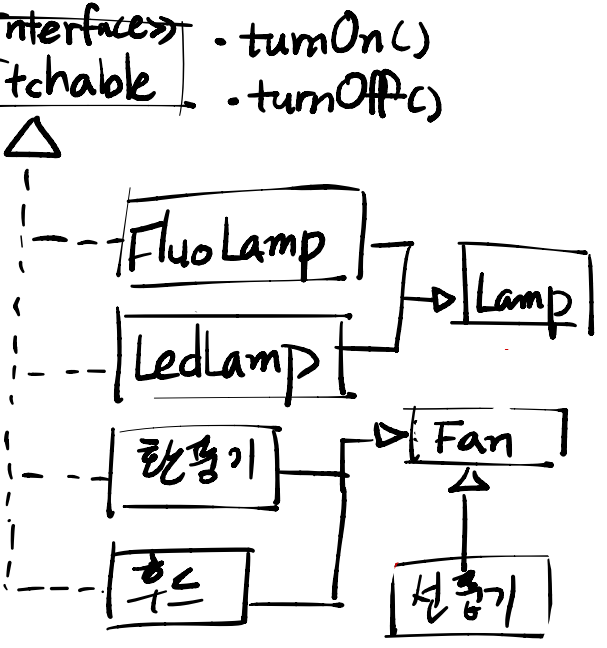
③ 상속은 "강결합"을 유지하게 된다. 유연성 부족.

20. SOLID의 DIP와 GRASP의 Low Coupling


```
class Switch {
    Switchable 장치;
    ≡
}
```

② 어떤 타입이든 Switchable 규칙에 따라 만든 제품이라면 제어가능

① 상속보다 인터페이스를 사용하면 같은 타입이 아닌 장치들을 스위치로 제어할 수 있다



↔ ③ 상속보다 더 유연하다 "약결합" = 느슨한 연결

Low Coupling

* SOLID - Dependency Inversion Principle (DIP)

↳ 의존객체를 직접 만들지 않고 외부에서 주입받는 방식.


```
class Switch {
    Switchable 장치;
}
```

① 느슨한 결합을 전환한다

```
FluoLamp light1 = new FluoLamp();
Switch switch = new Switch(light1);
```

② Switch가 의존객체를 생성하는 것이 아니라 외부에서 주입받는 방식으로 전환하면

③
✓ 재사용성이 높다
✓ 테스트가 쉽다.
↳ 의존객체를 간단히 만들어 주입함으로 스위치의 동작을 테스트할 수 있다.

더 유연해진다

* DI

SOLID

Dependency
Inversion
Principle

(의존성 역전 원칙)

(제어의 역전)

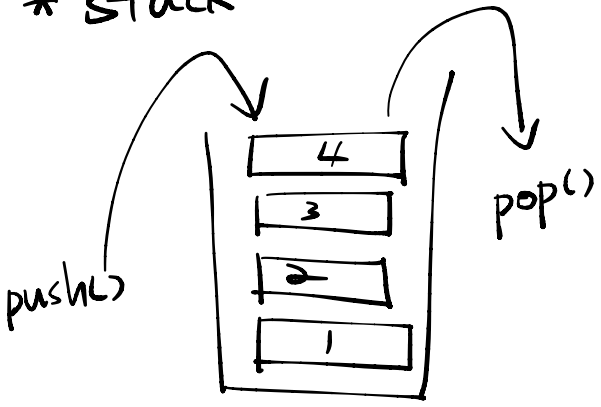
Inversion of Control (IoC)

① Dependency Injection

② Listener = event handler

21. 자료구조 - Stack 과 Queue

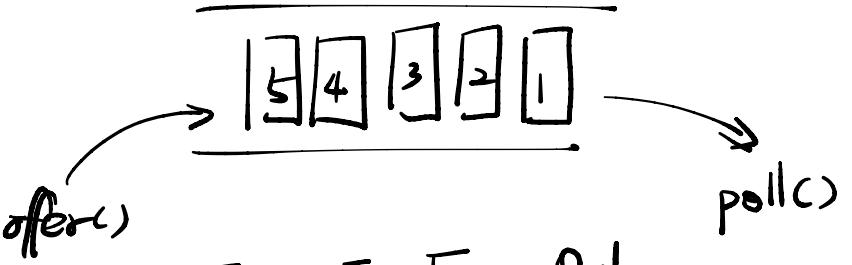
* stack



First In Last Out
(FILO)
=
Last In First Out
(LIFO)

① 방문페이지 ③ 웹브라우저 앞으로 뒤로
② 데이터 통신

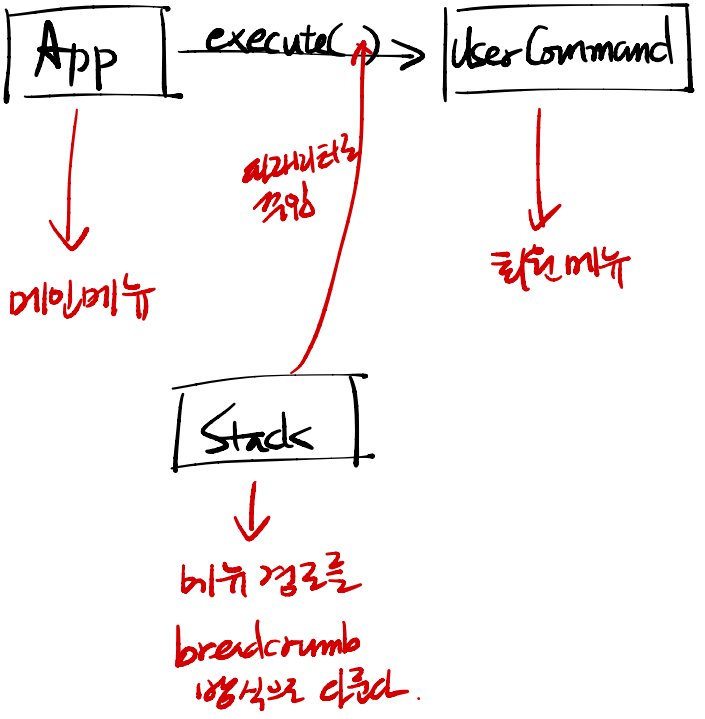
* Queue



First In First Out
(FIFO)

① 예약
② 한우스레드에서 클라이언트 요청 처리
③ 이벤트처리 = Event Queue

* 프로젝트 제목을 스택으로 다루기



* 문자열 연결

```
String str = "";
str += "aaa";
str += "bbb";
```

garbage

"" + "aaa" = "aaa" ← "새 String 객체 생성"

"aaa" + "bbb" = "aaabbb"

garbage

String immutable 객체이기 때문에 내용변경불가!
⇓
"새 String 객체 생성"

☆ 요즘 Java 컴파일러는 문자열 연결 코드를 StringBuffer 또는 StringBuilder 코드로 최적화 된다.

1아쉽다.
↳ garbage를 생성하지 않는 코드로 변경한다.

문자열을 연결하는 과정에 새 String 객체가 생성되고 기존 String 객체가 garbage가 된다. 메모리 낭비가 심해진다.
⇓
해결책? mutable 객체 사용
⇓
StringBuffer, StringBuilder

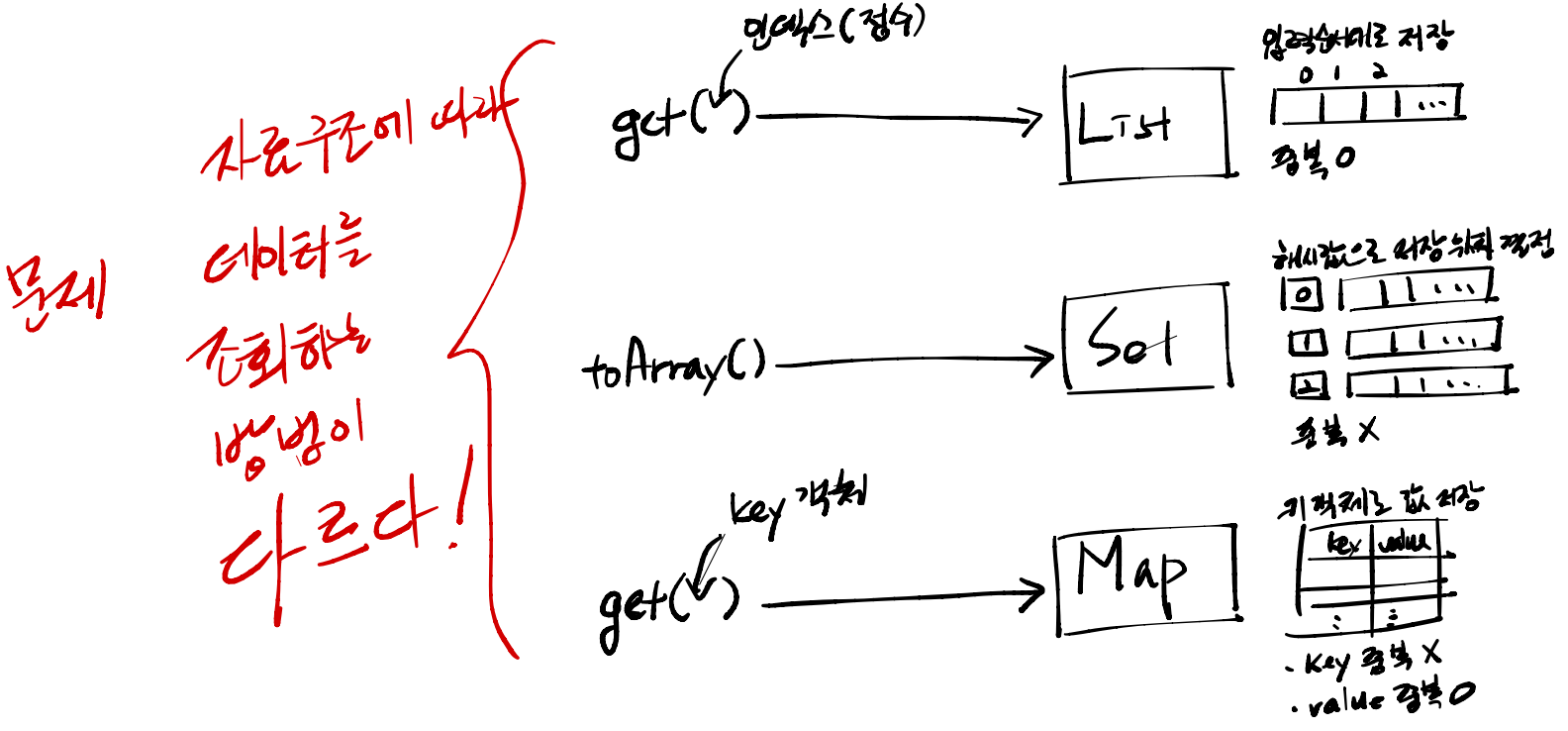
Thread-Safe
↓
멀티스레드 동시접근 차단 기능
↳ lock/unlock 하는데 시간소요
⇒ StringBuffer

thread-Safe 보장 안함 → StringBuilder
↳ 여러 스레드의 동시 접근 가능
↳ 다른 스레드의 값을 덮어 쓸수 있다
↳ lock/unlock 관리 안함
↳ 안정성↓ → 시간단축↓

22. Iterator 설계 패턴

어떤 문제에 대한 효율적인 해결 방식

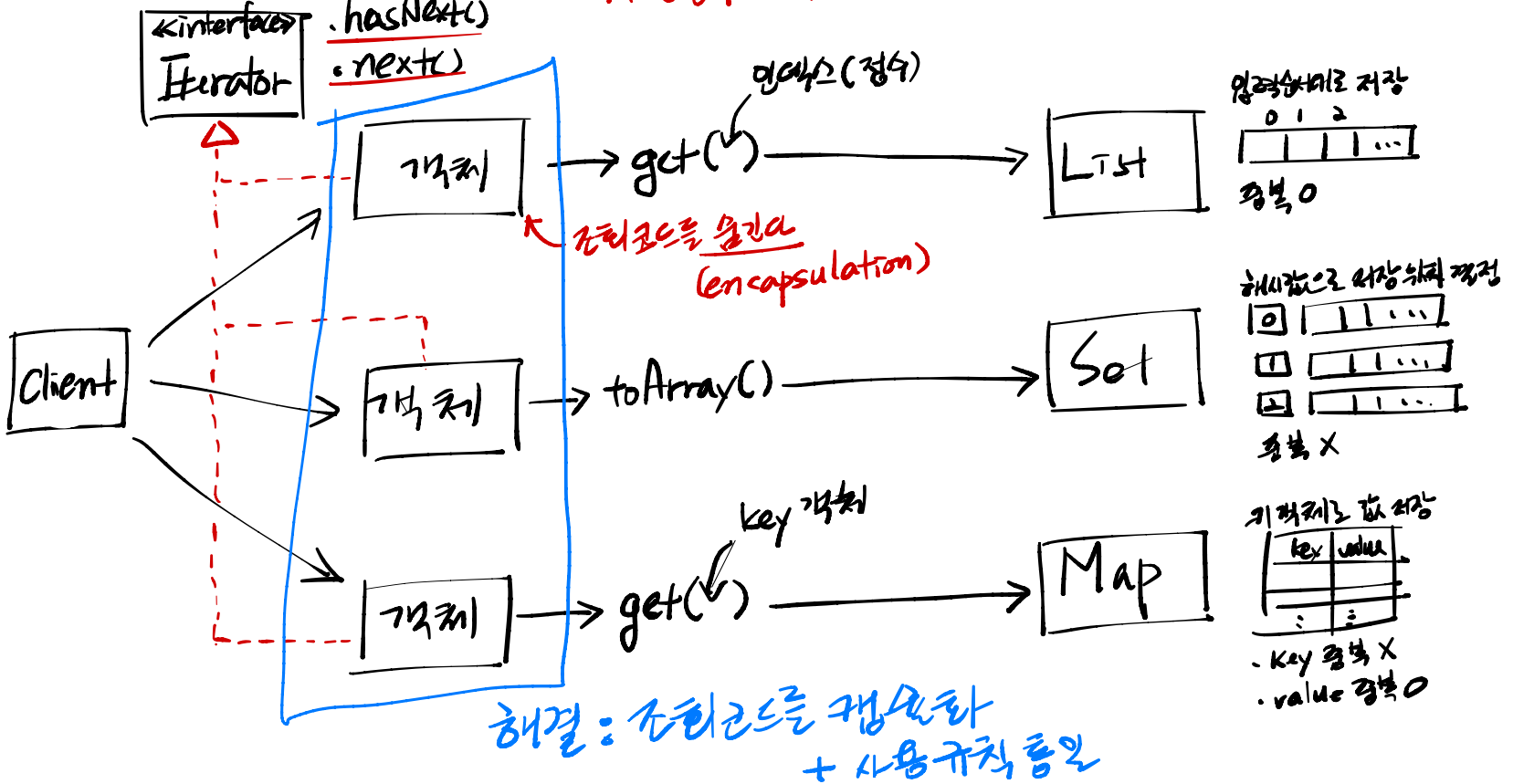
- 재사용성 ↑ · 유지보수 쉬움 · SOLID / GRASP 준수



22. Iterator 설계 패턴

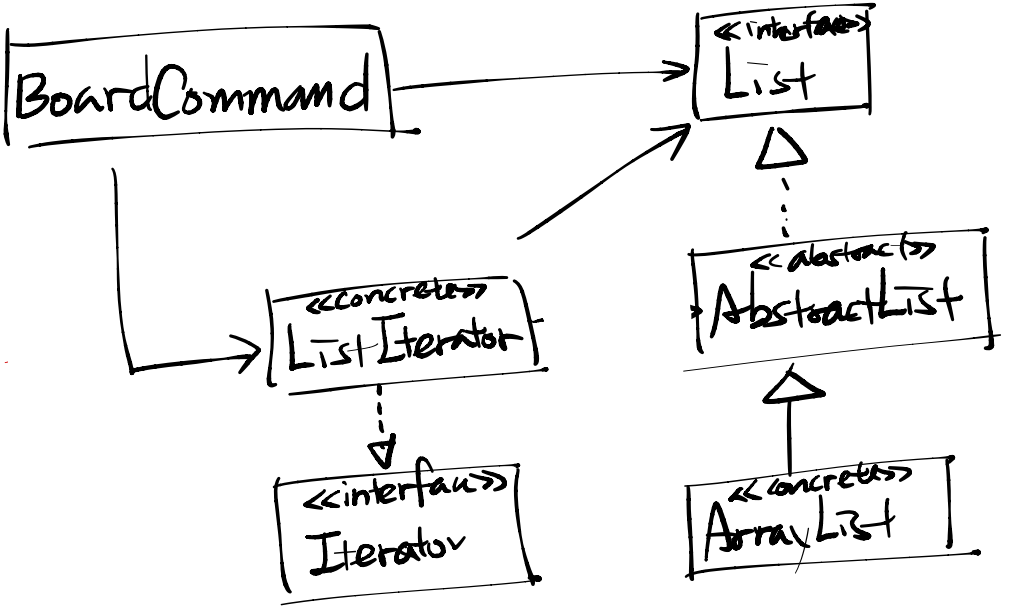
여러 문제에 대한 공통된 해결 방식

- 재사용성 ↑
- 유지보수 쉬움
- SOLID / GRASP 적용



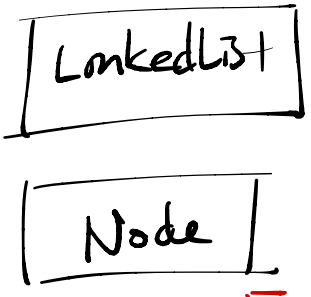
해결 : 조회코드를 캡슐화
+ 사용규칙 통일

22. Iterator 설계 패턴



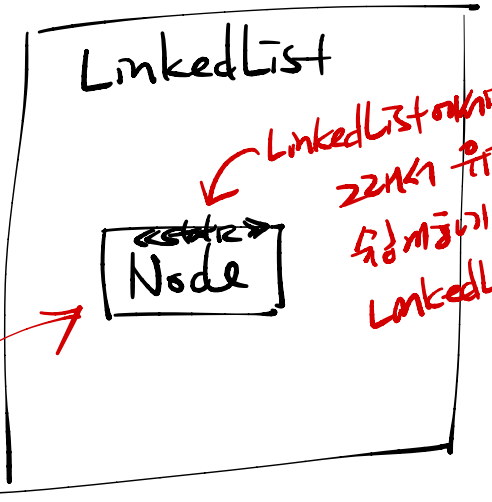
23. 중첩 클래스

before



package member class

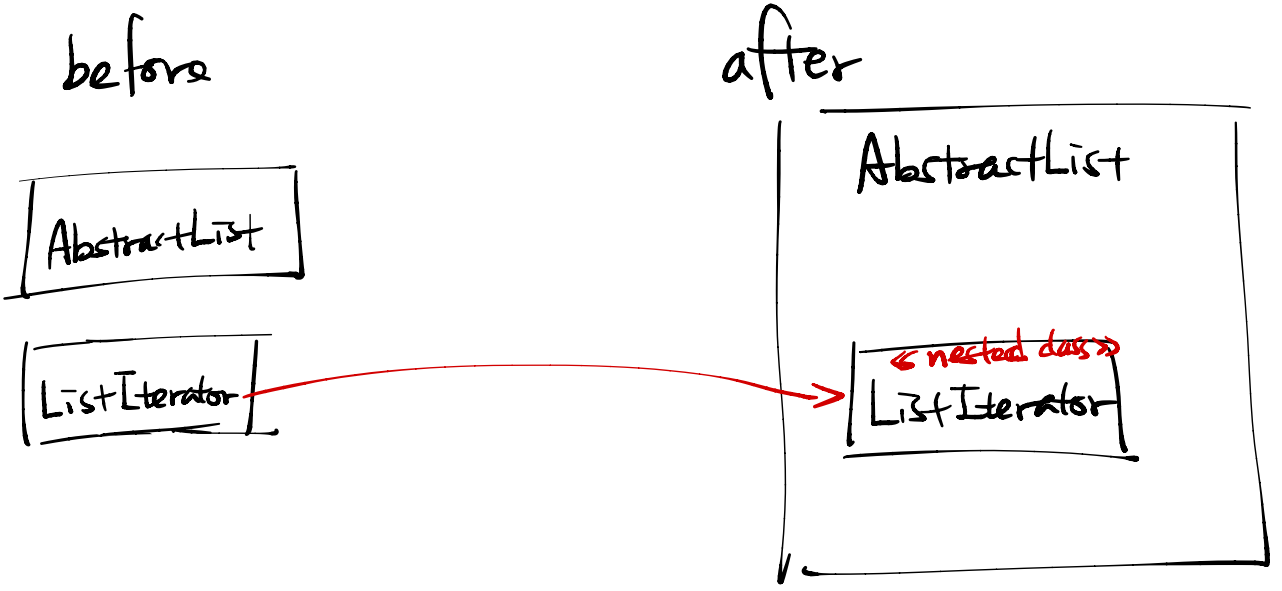
after



LinkedList에서만 사용됨.
그래서 캡슐화를
유지하기 위해
LinkedList로 옮김!

Nested class
(static nested class)

23. 중첩 클래스


before
AbstractList
ListIterator
after
AbstractList
<< nested class >>
ListIterator

* enhanced for 문장

for(변수선언 : 배열 또는 Iterable 구현체)

예) String[] names = {"홍길동", "임꺽정", "유관순"};

for(String name : names) { — }

배열

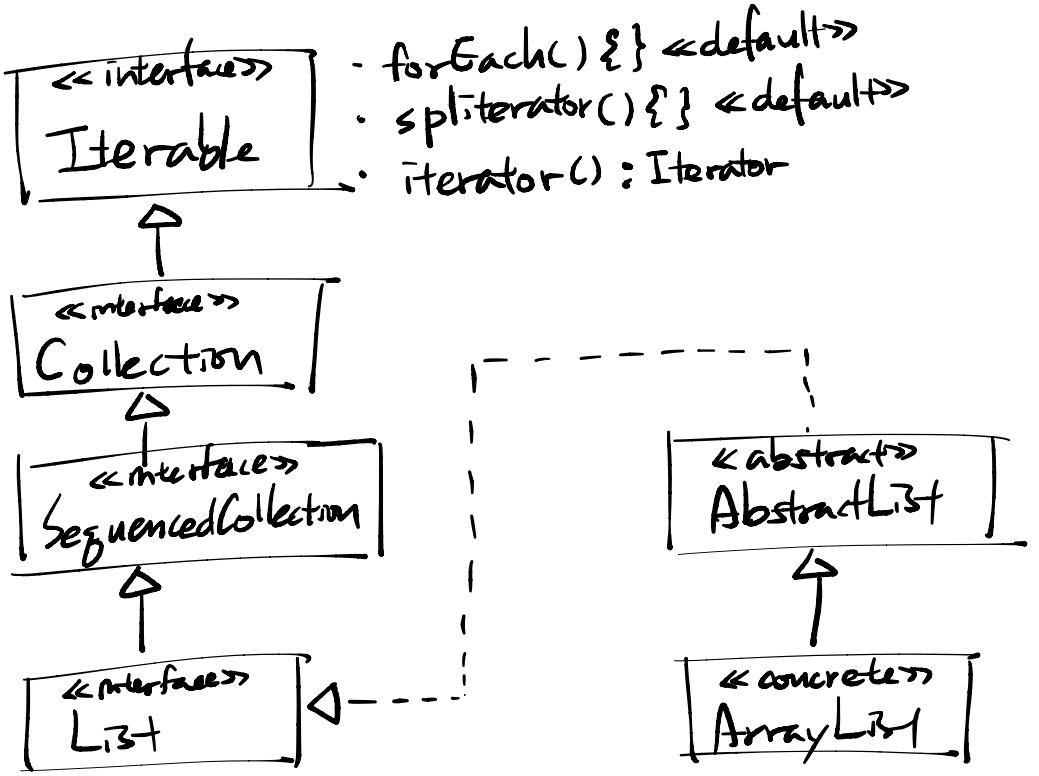
예) ArrayList list = new ArrayList();

list.add("홍길동"); list.add("임꺽정"); list.add("유관순");

for(Object item : list) { — }

Iterable 구현체

* Iterable 구현체



24. Generic 문법 사용하기

```
class Node {

    Object value;
```

```
Node n = new Node();
```

n.value ← String
n.value ← Date
n.value ← User
n.value ← Project
⋮

다형성 변수의 특성상 하위 타입의 서브클래스 객체까지 저장.

특정 타입의 객체만 저장하도록 제한할 수 없을까?

⟹ Generic 문법

24. Generic 문법 사용하기

타입 파라미터 = 타입 정보를 받을 변수

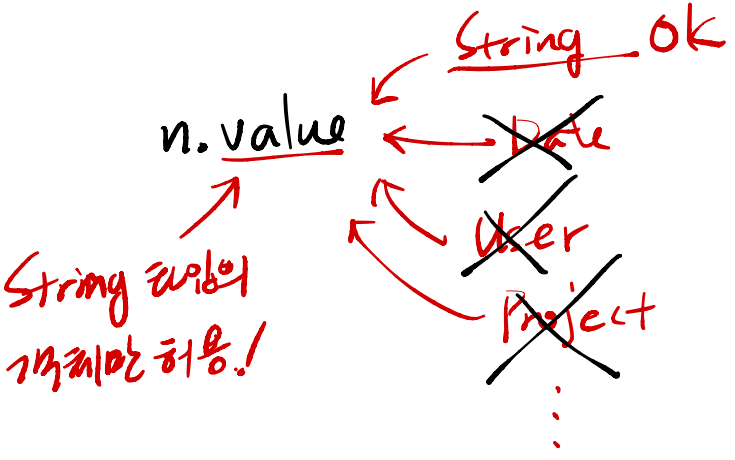
```
class Node<what> {

    what value;
```

what 이 어떤 타입인지 사용할 때 지정한다

레퍼런스에서 타입 파라미터에 들어갈 값을 지정했다면 생략가능

```
Node<String> n = new Node<String>();
```



String 타입의 객체만 허용!

* Type Parameter

↳ 타입 형변을 받는 변수

이름
- T (Type)
- E (Element)
- K (Key)
- V (Value)

S, U, V

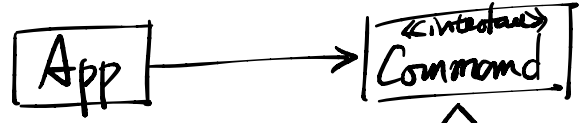
26. GoF의 Composite 설계 패턴

↳ 객체가 트리구조로 포함관계일 때.

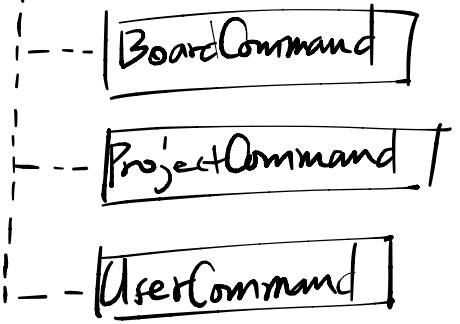
* before

문제
- 메뉴 UI 제공 코드 중복
- Command 객체의 여러 메서 일을 수행

해결
- 메뉴 UI 제공 코드를 캡슐화 → 객체로 분리
- SOLID 원리 적용
  ↳ Single Responsibility Principle



- 메뉴 UI 제공 (main)



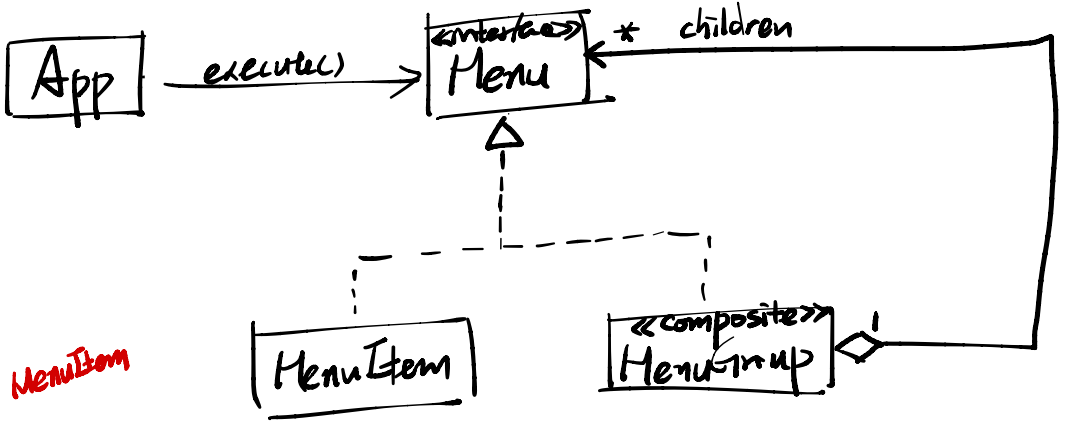
- 게시판 메뉴 UI 제공
- 게시판 CRUD 처리
- 프로젝트 메뉴 UI 제공
- 프로젝트 CRUD 처리
- 회원 메뉴 UI 제공
- 회원 CRUD 처리

26. GoF의 Composite 설계 패턴

↳ 객체가 트리구조로 표현될 때.

① 메뉴
② 주식도
③ 파일 시스템

* after

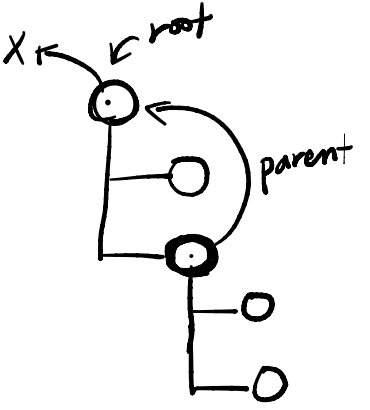


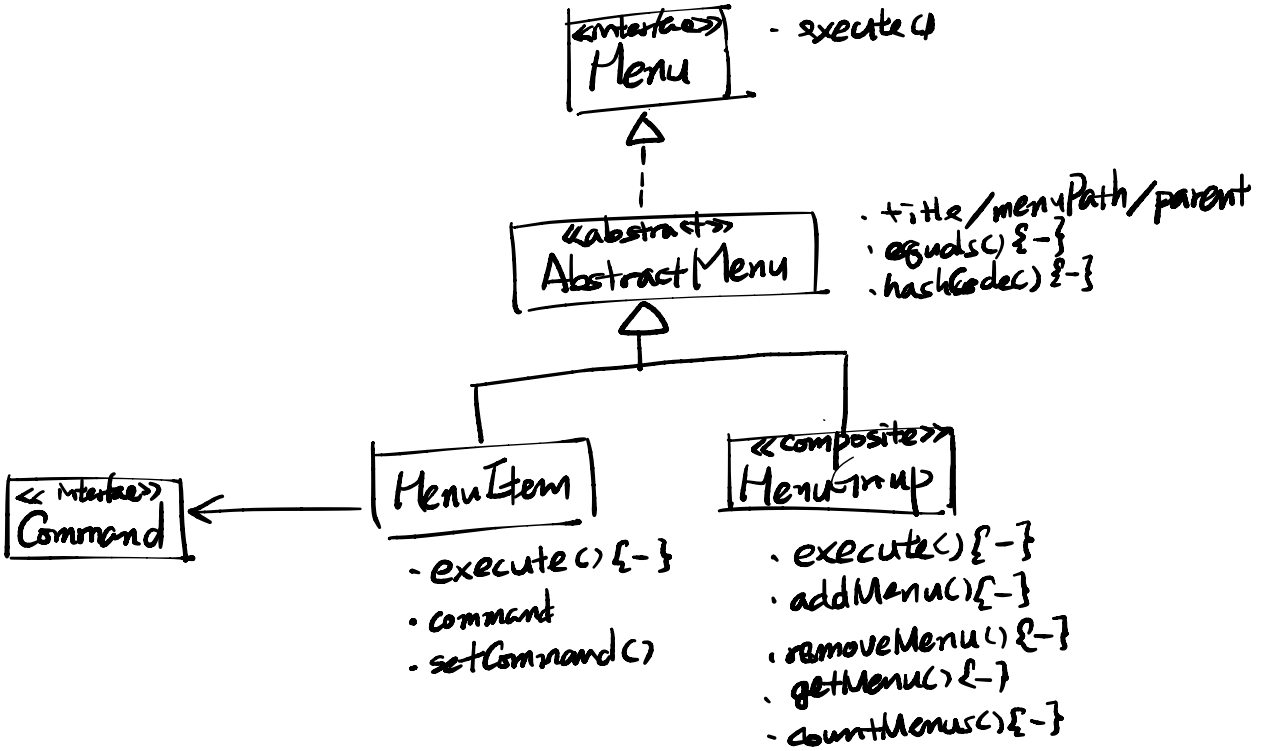
예) 메뉴

파일
- 새텍스트파일
- 새폴더
- 새창
- ⋮
- 최근항목열기 ← MenuGroup
  - ⋮

MenuItem

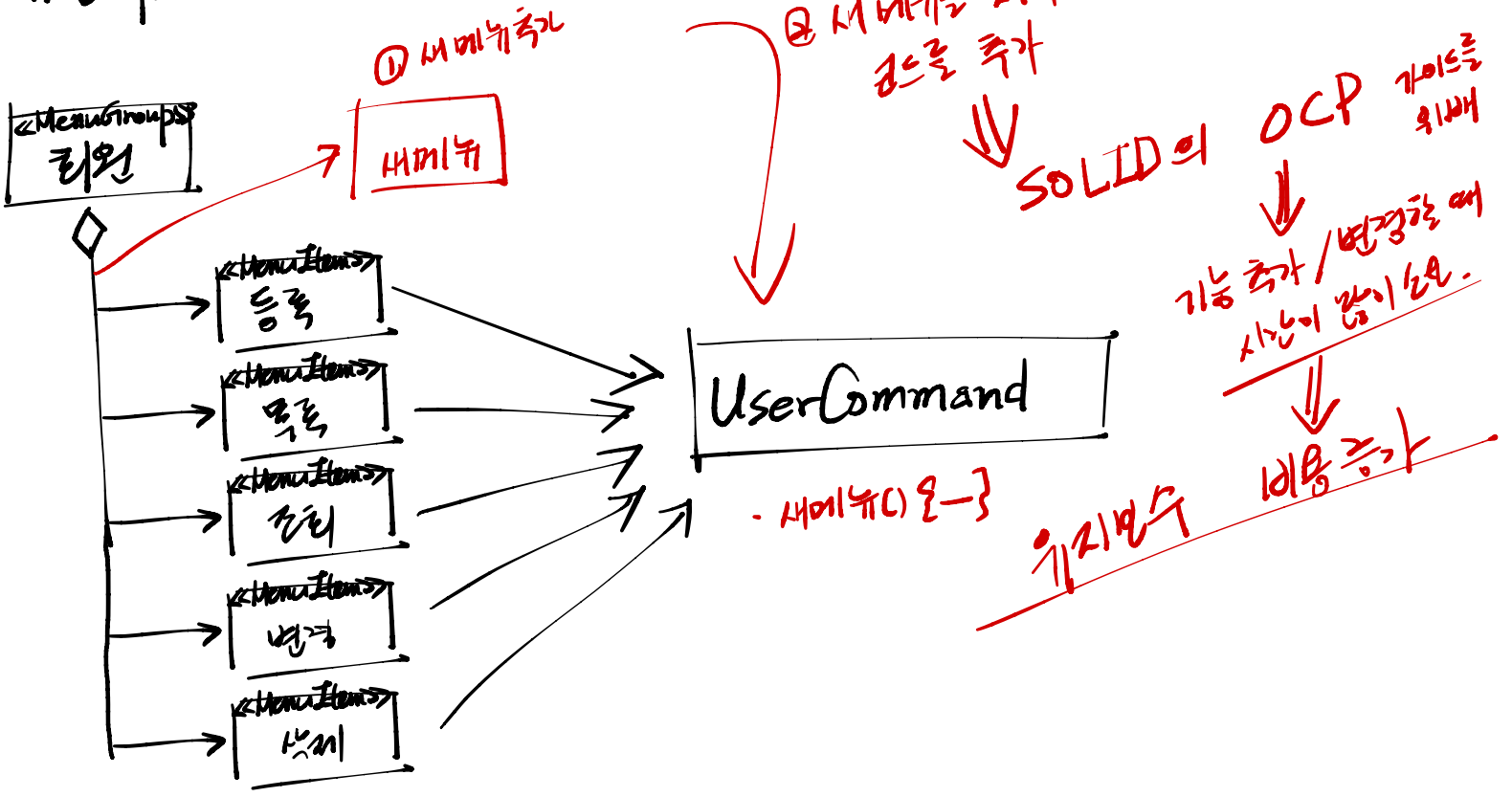
26. GoF의 Composite 패턴





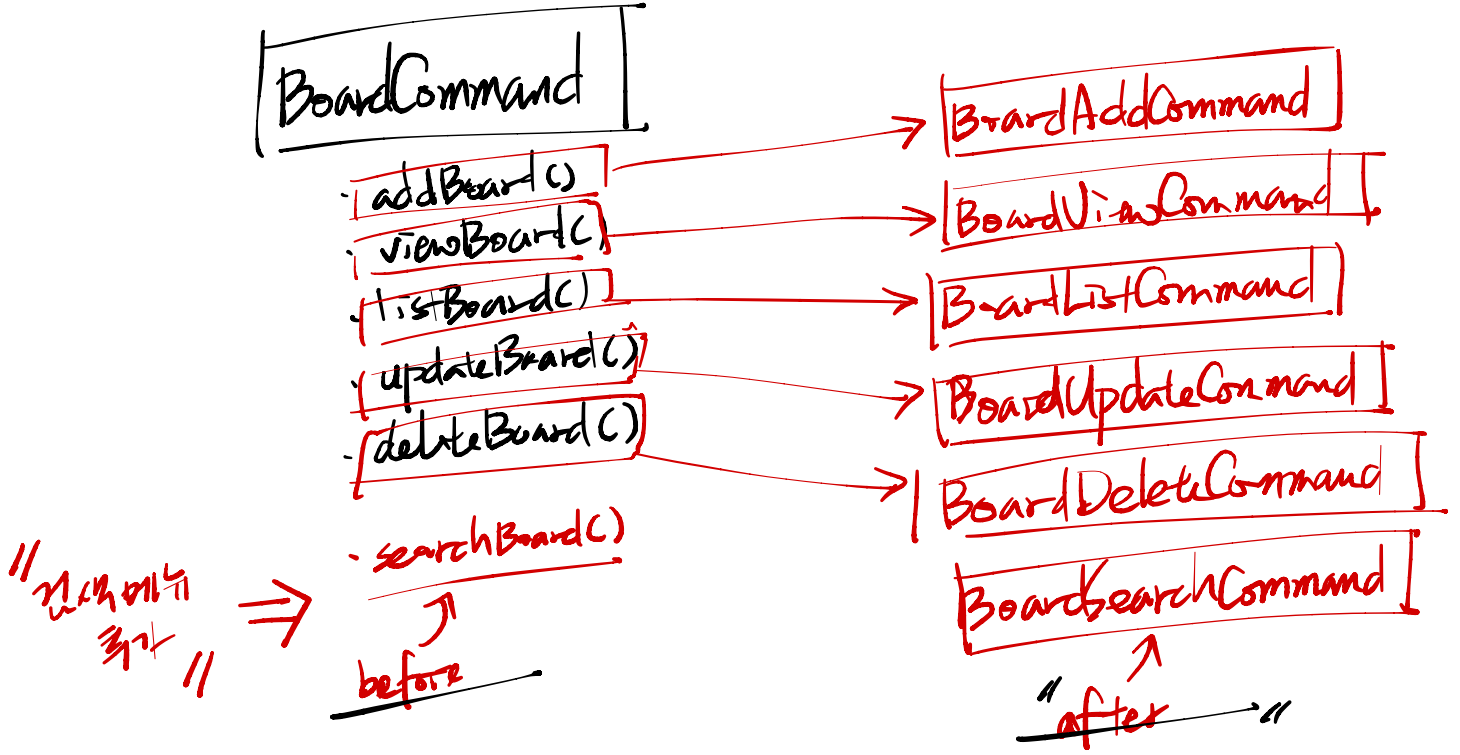
2차, 각각의 메뉴처리 기능을 객체화 : GoF의 Command 패턴 적용

* before



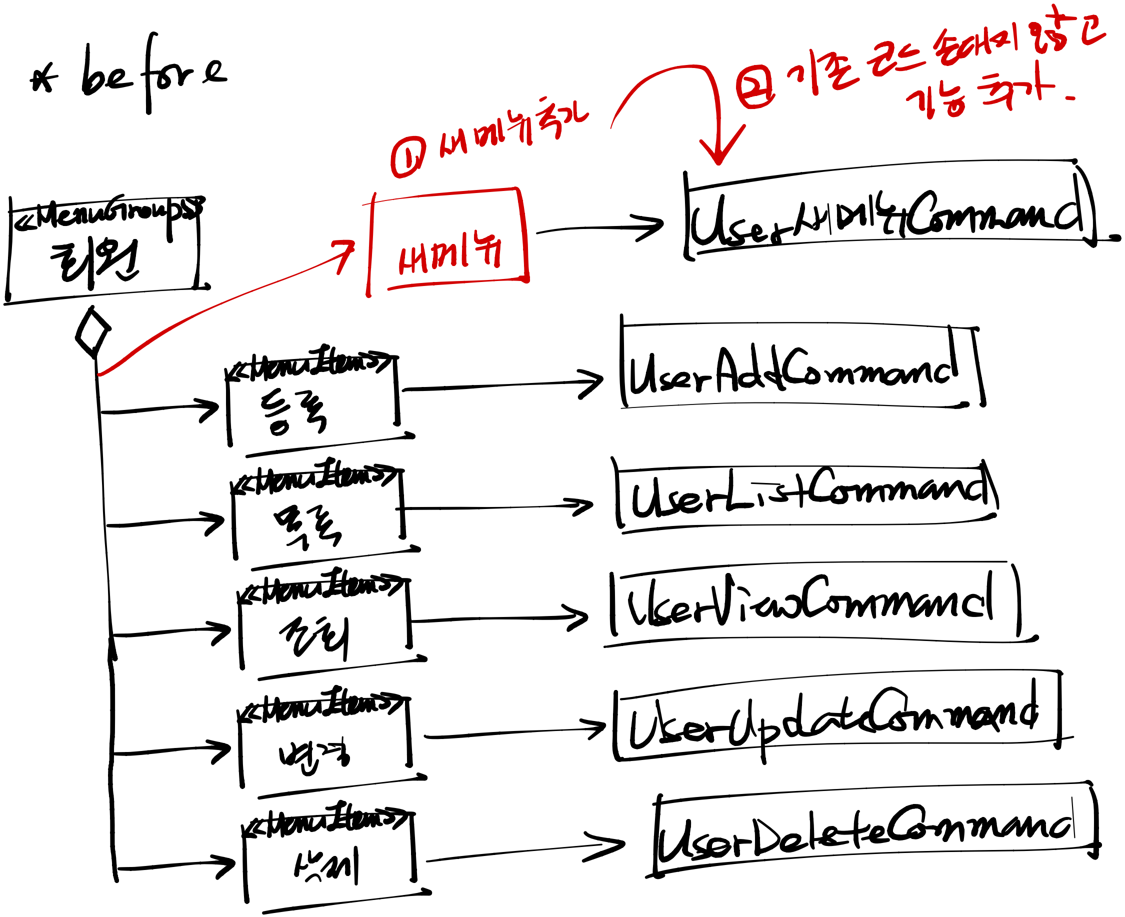
2차, 각각의 메뉴별로 기능을 객체화 : 4.0부터 Command 본격 적용

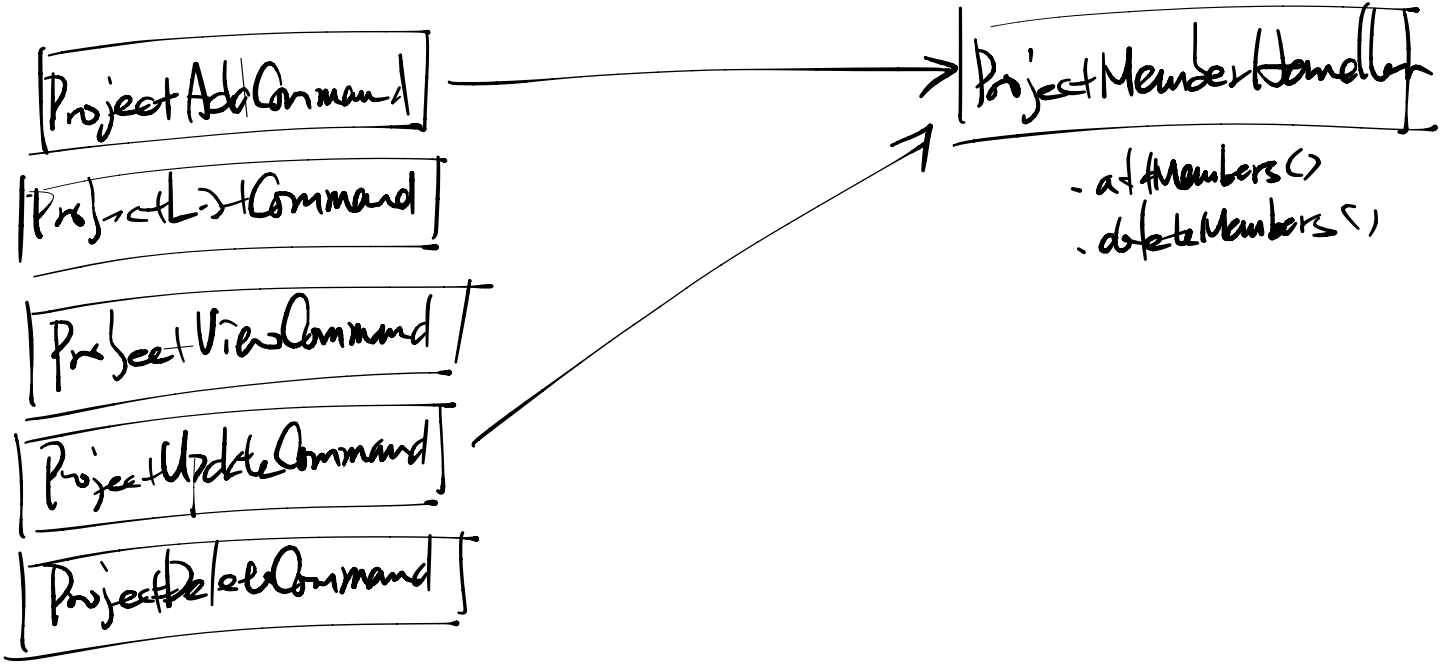
↳ 메서드 → 객체



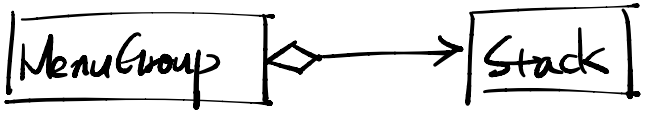
2)1. 각각의 메뉴처리 기능을 객체화 : GoF의 Command 설계 패턴

* before

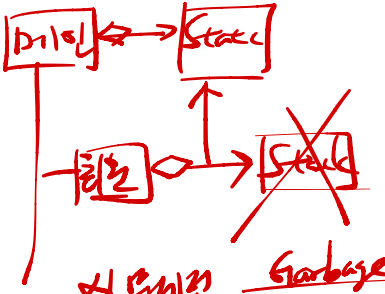



ProjectAddCommand
ProjectListCommand
ProjectViewCommand
ProjectUpdateCommand
ProjectDeleteCommand
ProjectMemberHandler
.addMembers()
.deleteMembers()

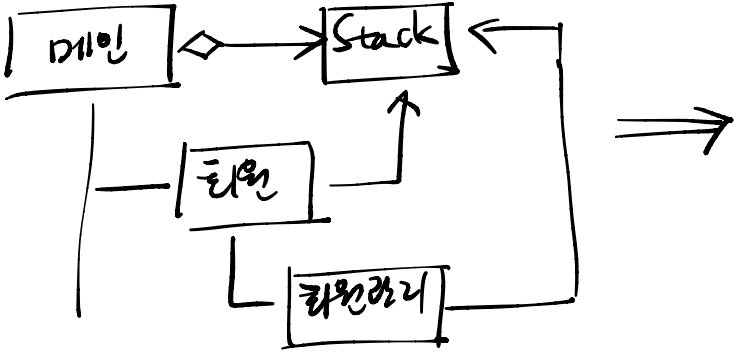
* MenuGroup 의 경로 계산을 메서드로 분리

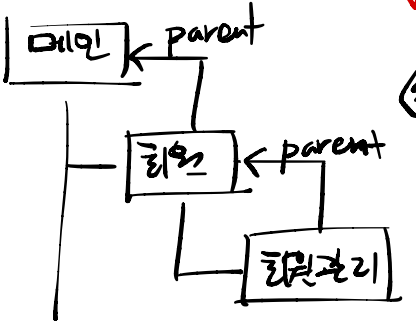


· 메뉴이름을 누적해서 계산



예)



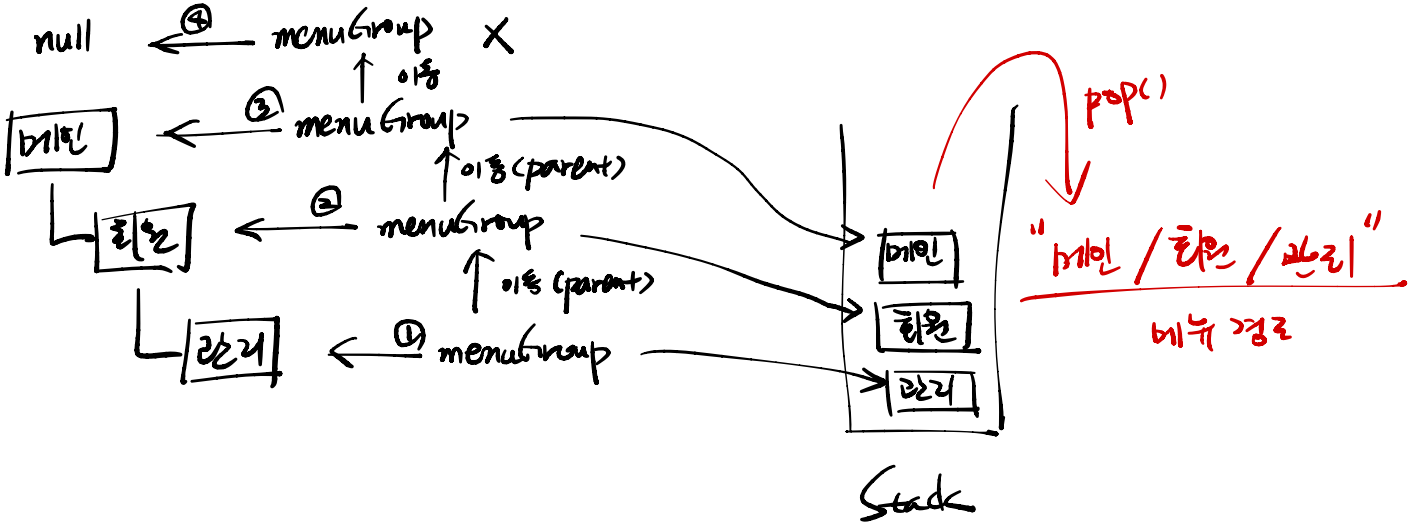


✓ getMenuPath() 메서드 추가

↑

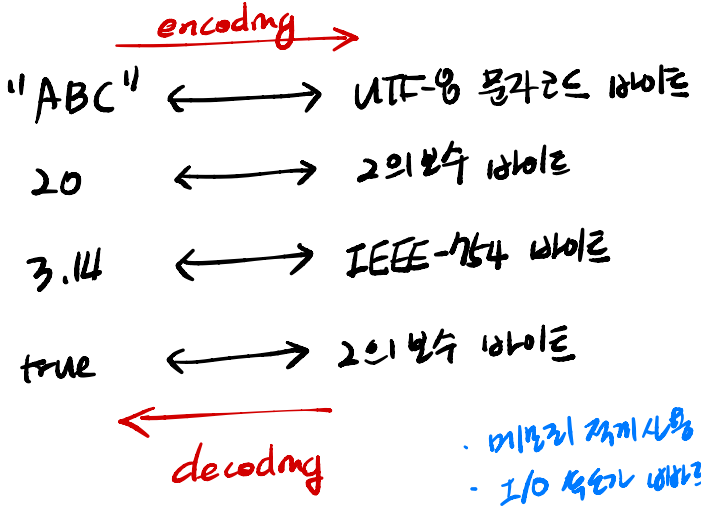
메뉴경로를 계산

* getMenuPath() 작동 원리



28. File I/O API 활용 : ① 바이너리 데이터 입출력

* binary data I/O

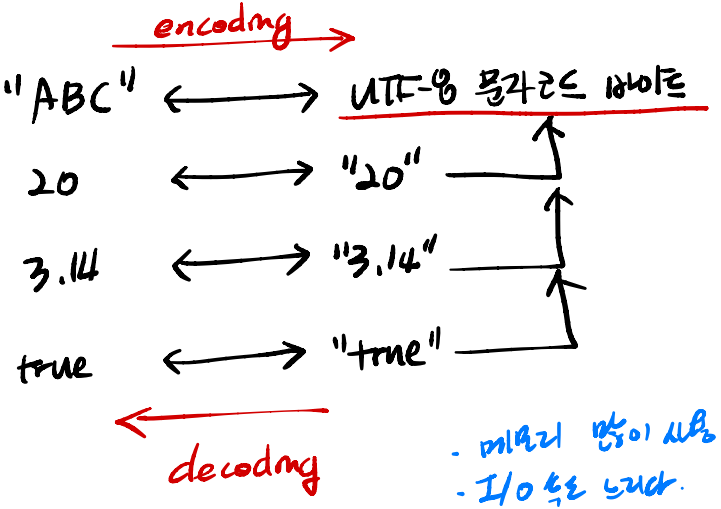


- 메모리 적게사용
- I/O 속도가 빠르다

예) PDF, PPT, DOC, GIF, JPEG, MP3, MP4, AVI, WAV, HWP, EXE 등

↳ 전용편집기를 이용해서 I/O

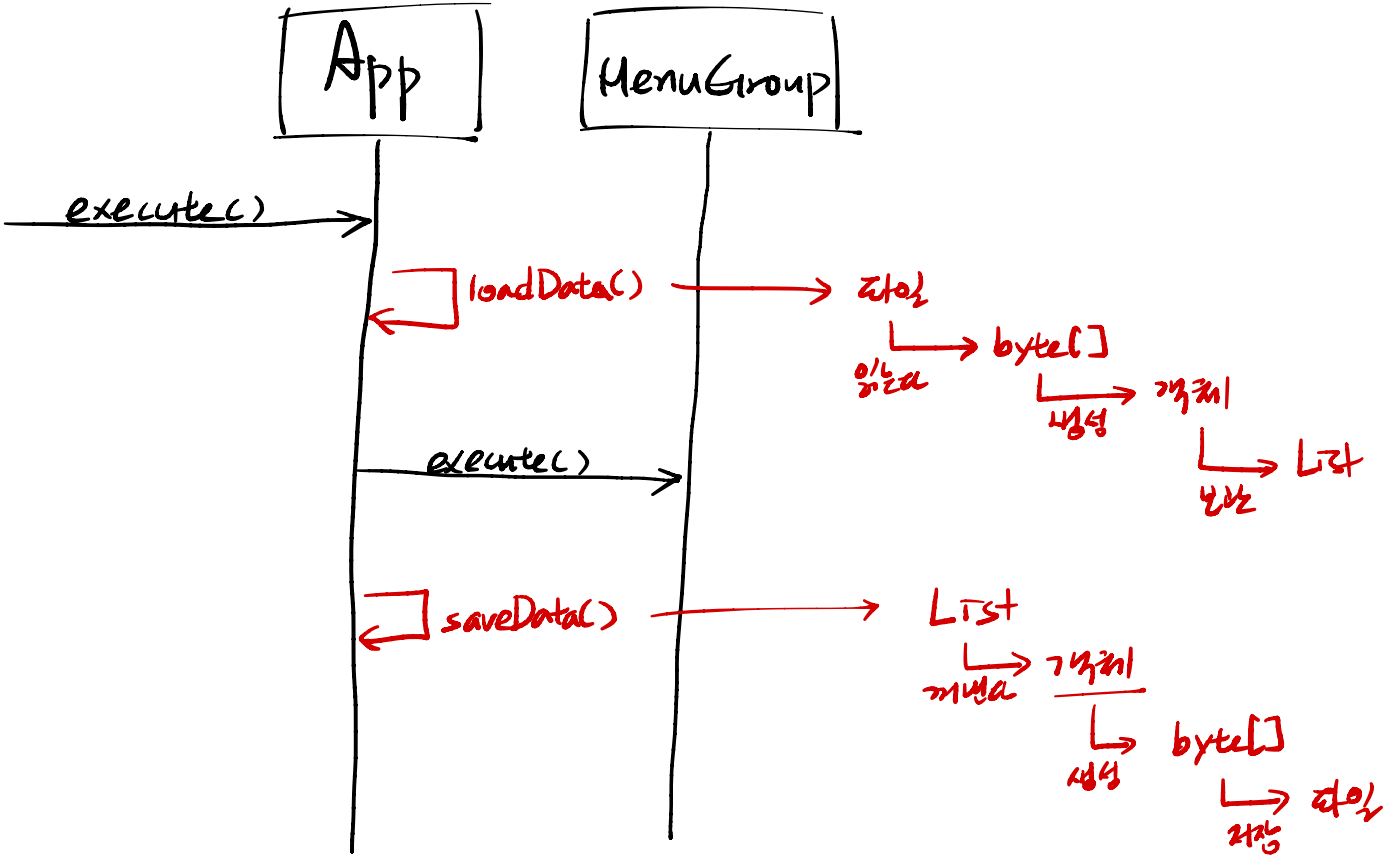
* text data I/O



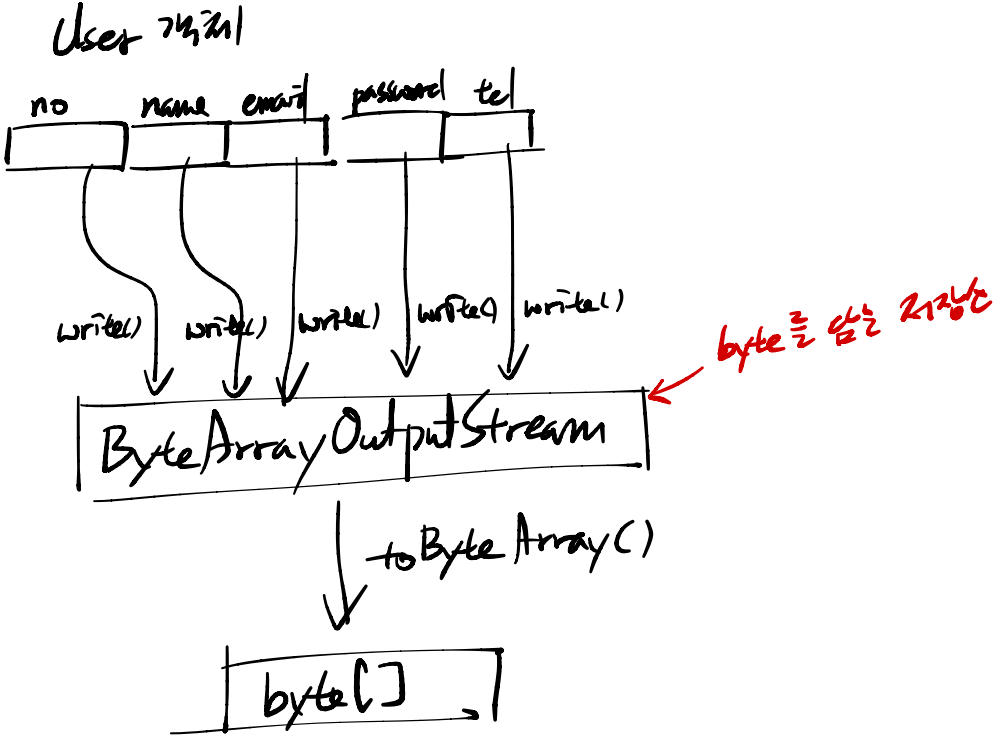
- 메모리 많이 사용
- I/O 속도 느리다.

예) TXT, HTML, CSS, JavaScript, XML, Properties, JAVA 등

↳ 텍스트 편집기를 이용해서 I/O

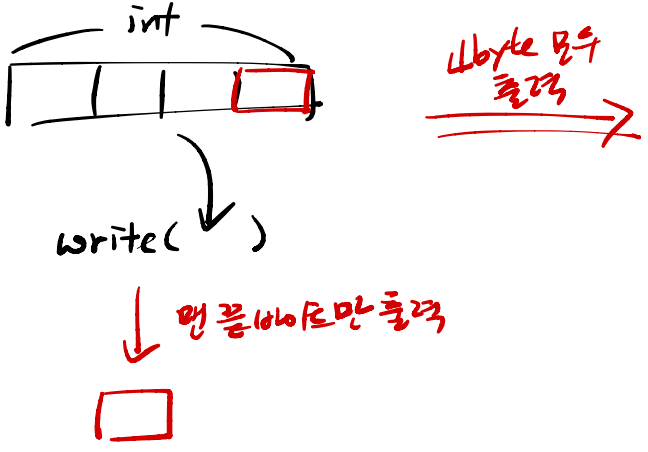

App
MenuGroup
execute()
loadData()
execute()
saveData()

* 객체의 값을 byte[]로 만들기

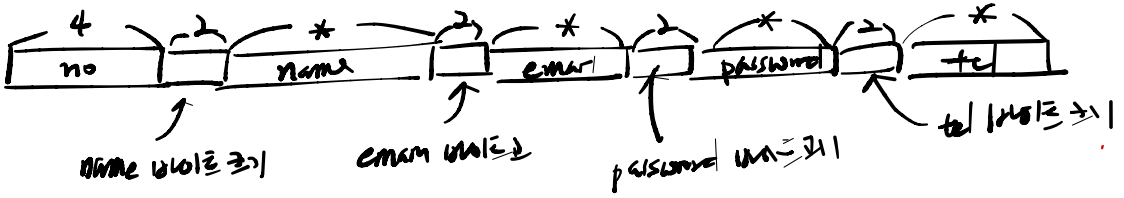


* write(int) : int 출력

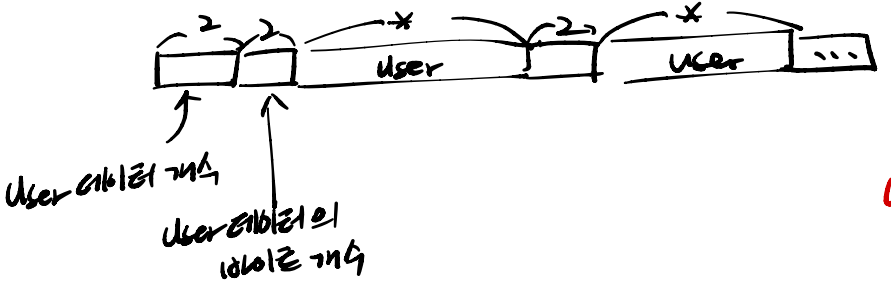
↳ 한 번에 1 바이트만 출력



* User 데이터 형식



* user.data 파일형식 (File Format)

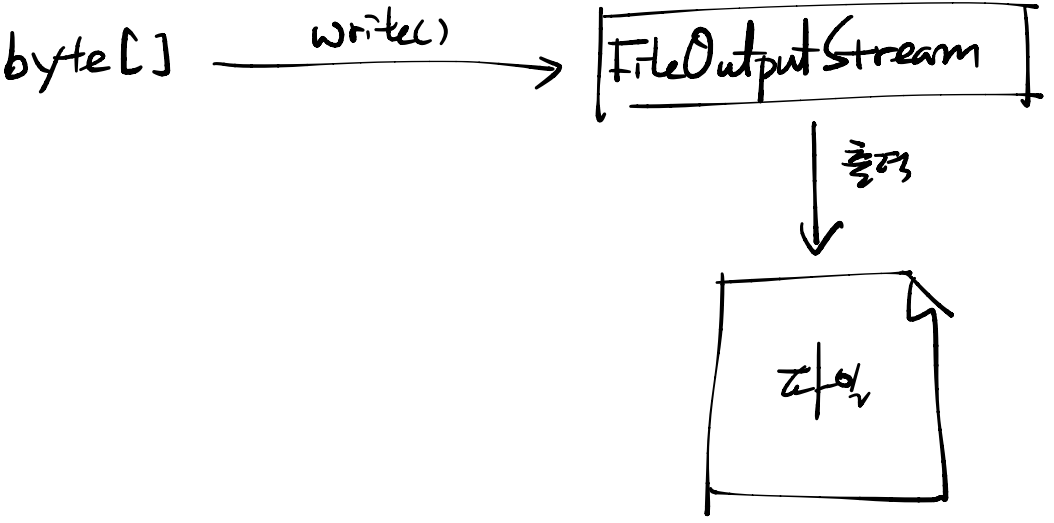


2 byte - User 데이터 개수

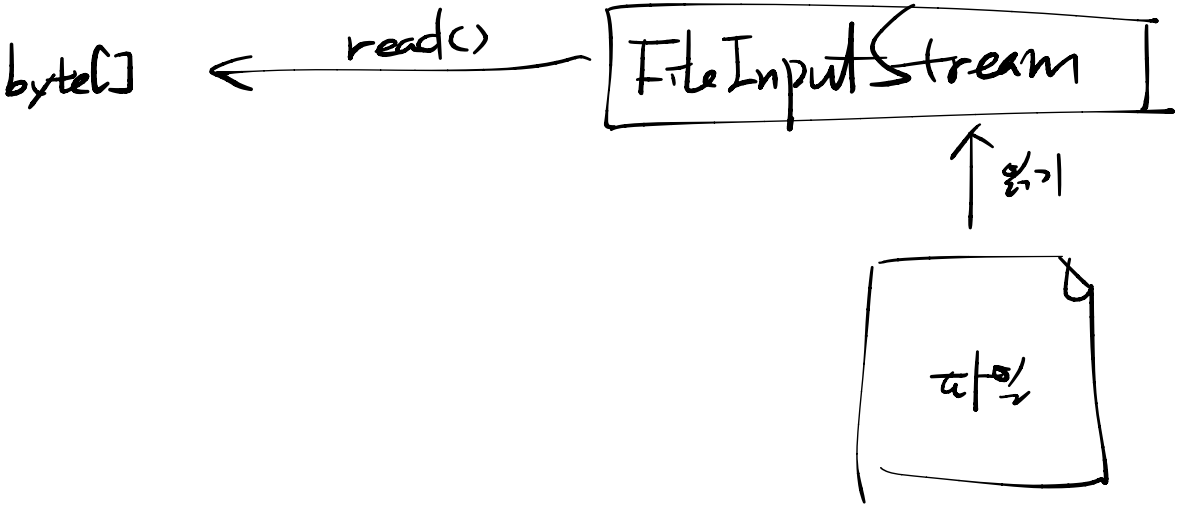
User
- 2 byte - User 데이터 바이트배열 크기
- 4 byte - no
- 2 byte - name 바이트배열 크기
- * byte - name 바이트
- 2 byte - email
- * byte
- 2 byte - password
- * byte
- 2 byte - tel
- * byte

⋮

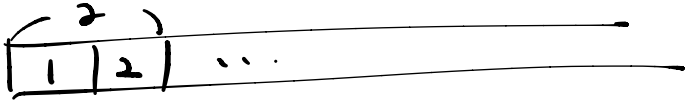
* byte[] → 출력 → 파일



* 파일 —읽기→ byte[]

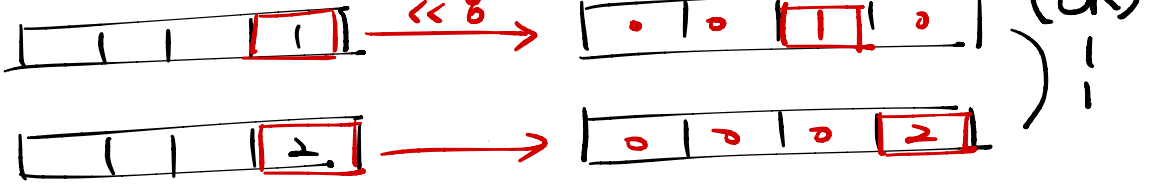


* byte[] ⟶ int



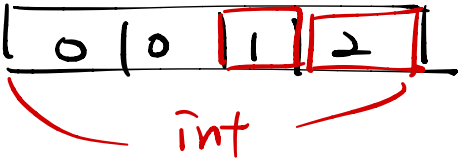
① read() ⟶

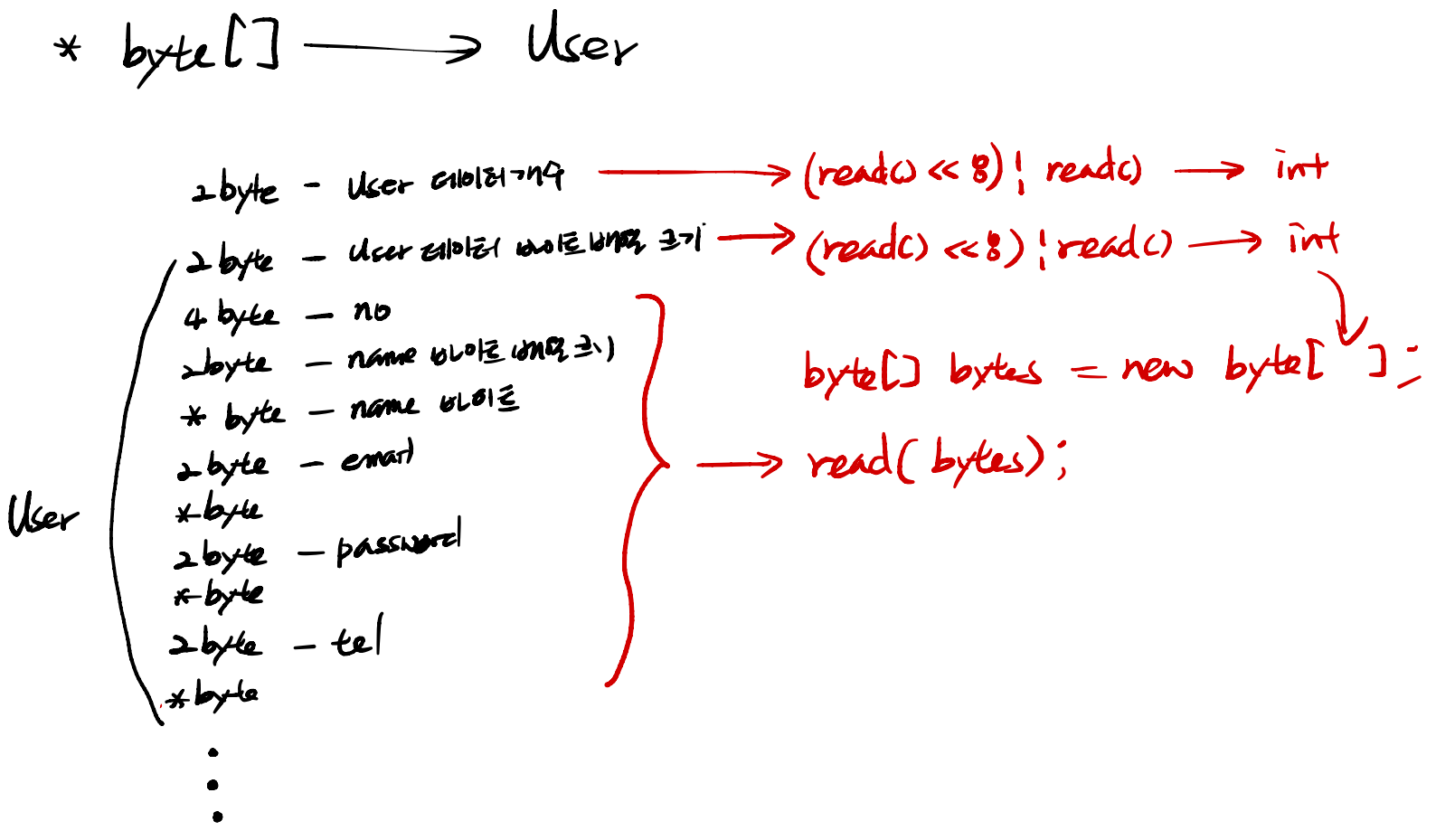
② read() ⟶



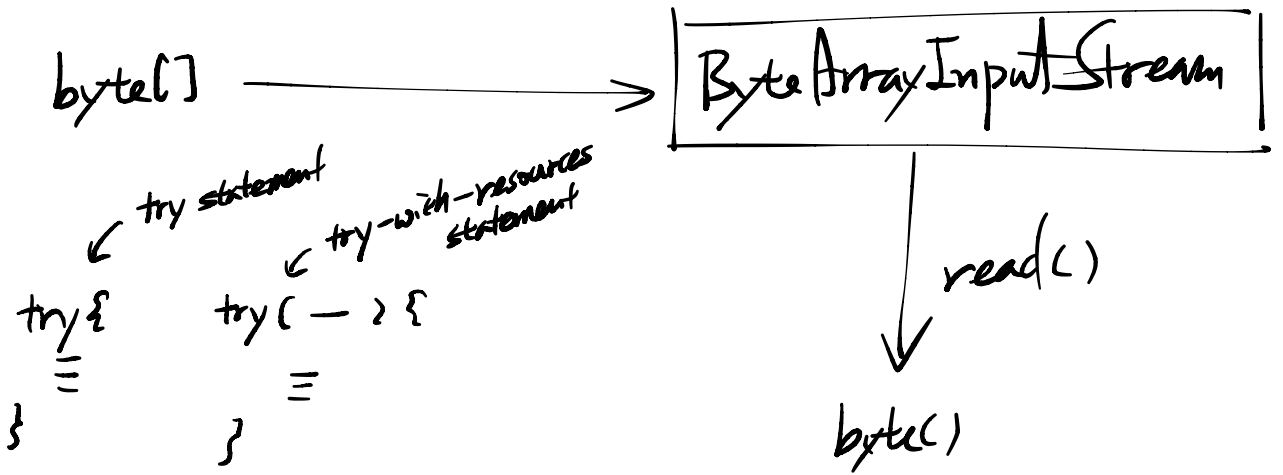
1 byte 를 읽어서

4 byte int 값으로 저장

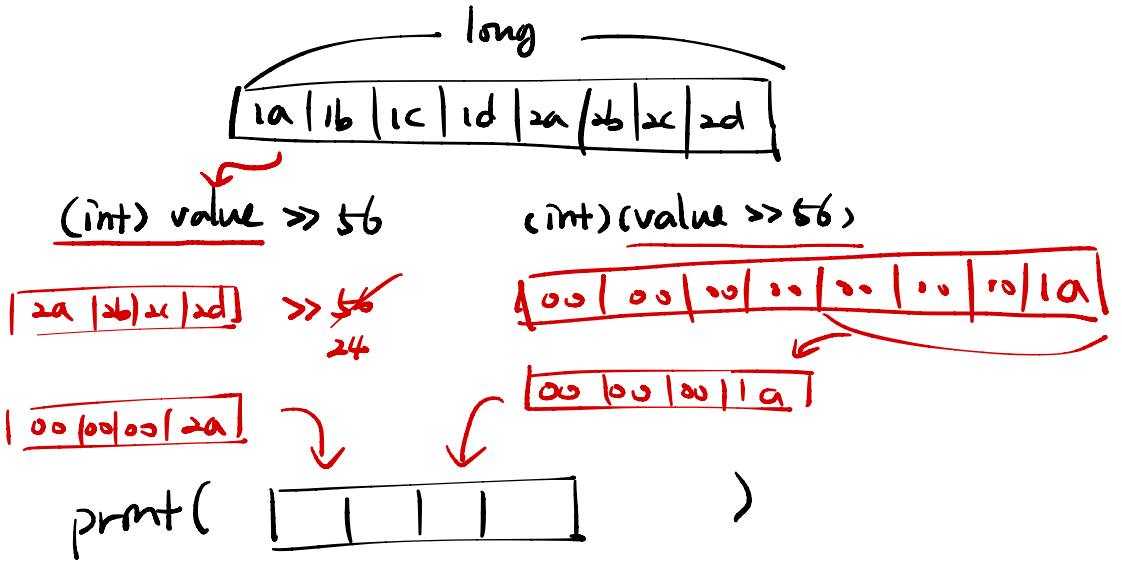



* byte[] → User
2 byte - User 데이터 개수
2 byte - User 데이터 바이트배열 크기
(read() << 8) | read() → int
(read() << 8) | read() → int
User
4 byte - no
2 byte - name 바이트배열크기
* byte - name 바이트
2 byte - email
* byte
2 byte - password
* byte
2 byte - tel
* byte
byte[] bytes = new byte[ ];
read(bytes);

* ByteArrayInputStream



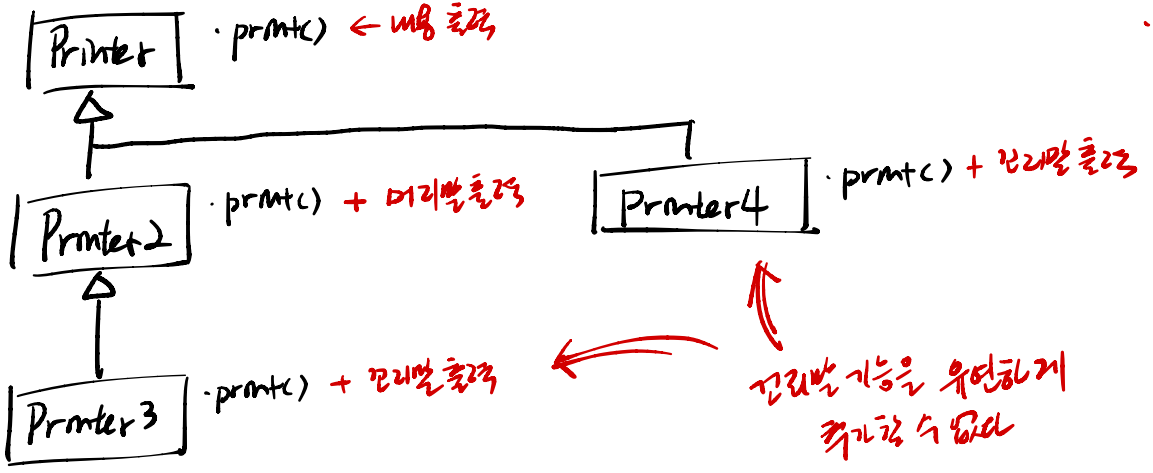
* 비트이동 시 연산자우선순위



29. File I/O API : Decorator 클래스 활용하기

기능확장

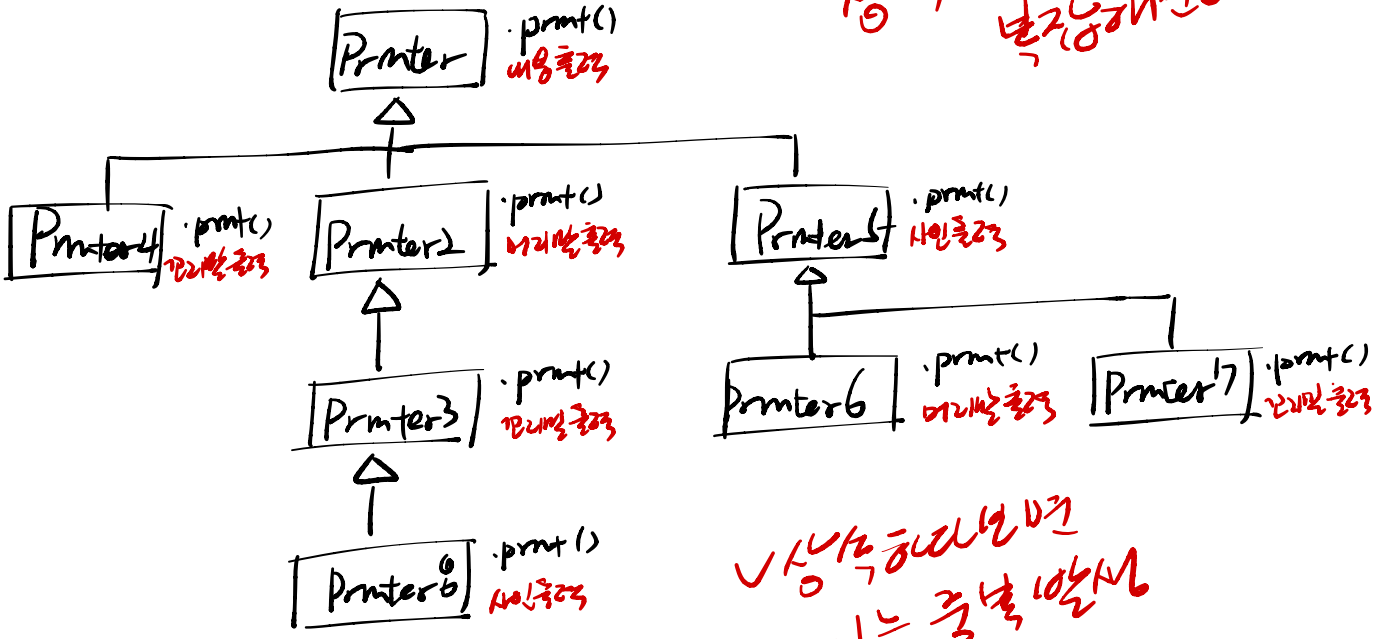
① 상속



* 단점
· 강한결합 ← 상속받은
↳ 일부기능 제외 불가

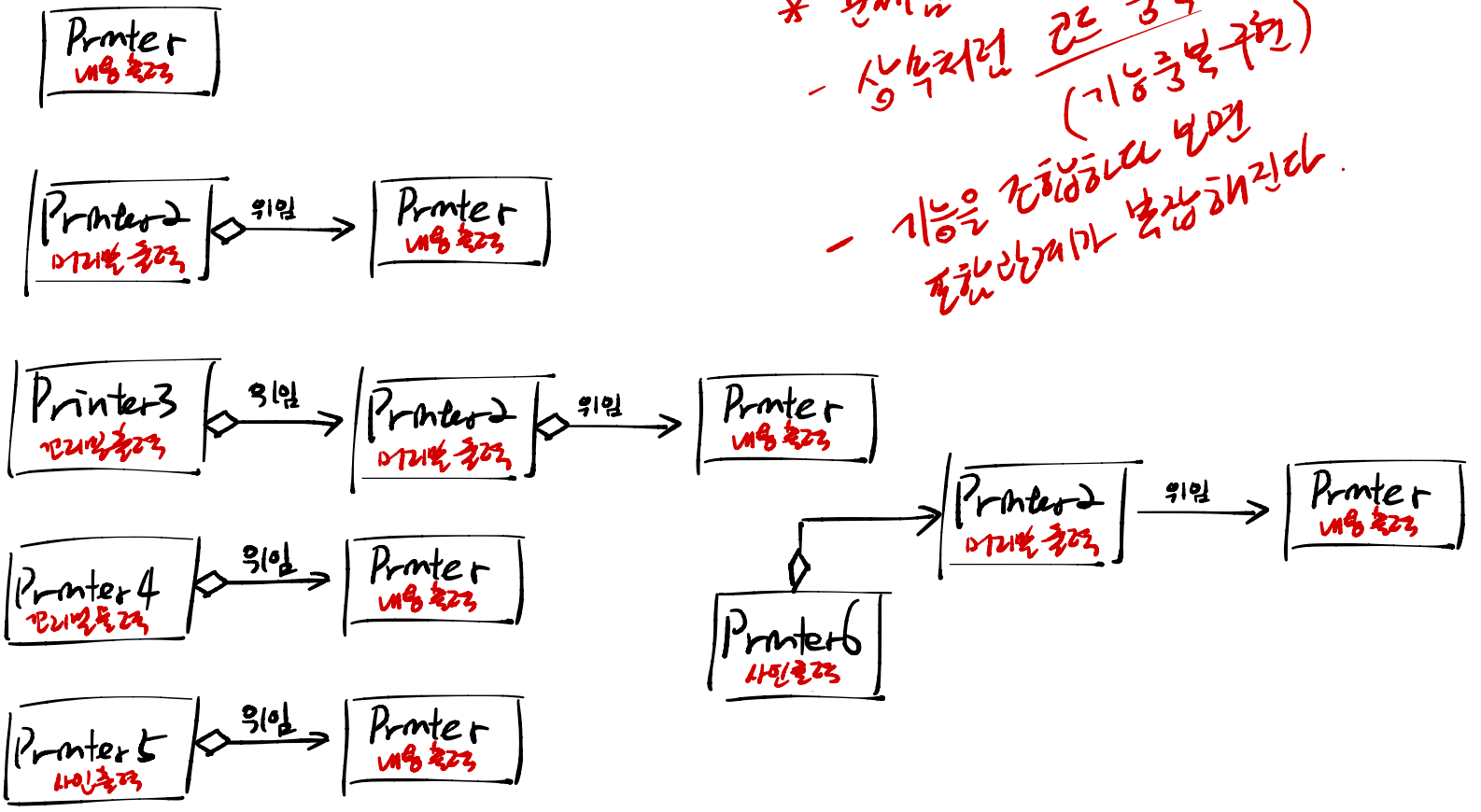
* 상속을 이용한 기능확장시 발생 상황

✓ 기능구현의 따라 상속관계가 복잡해진다



✓ 상속하면서 기능 중복 발생

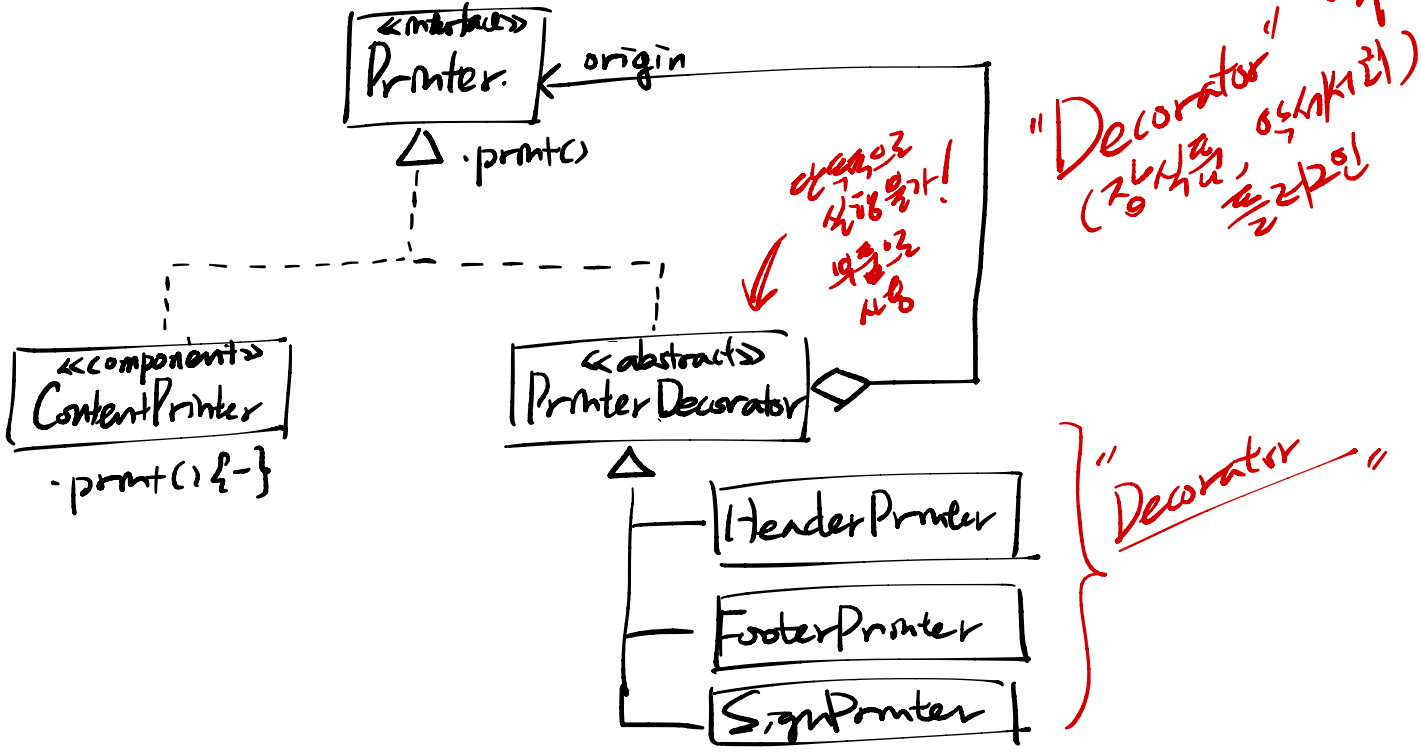
② 포함관계를 이용한 기능확장



* 문제점
- 상속처럼 큰 클래스 발생 (기능중복구현)
- 기능을 구현하는 방법 포함관계가 복잡해진다.

③ GoF의 Decorator 패턴을 이용한 기능 확장

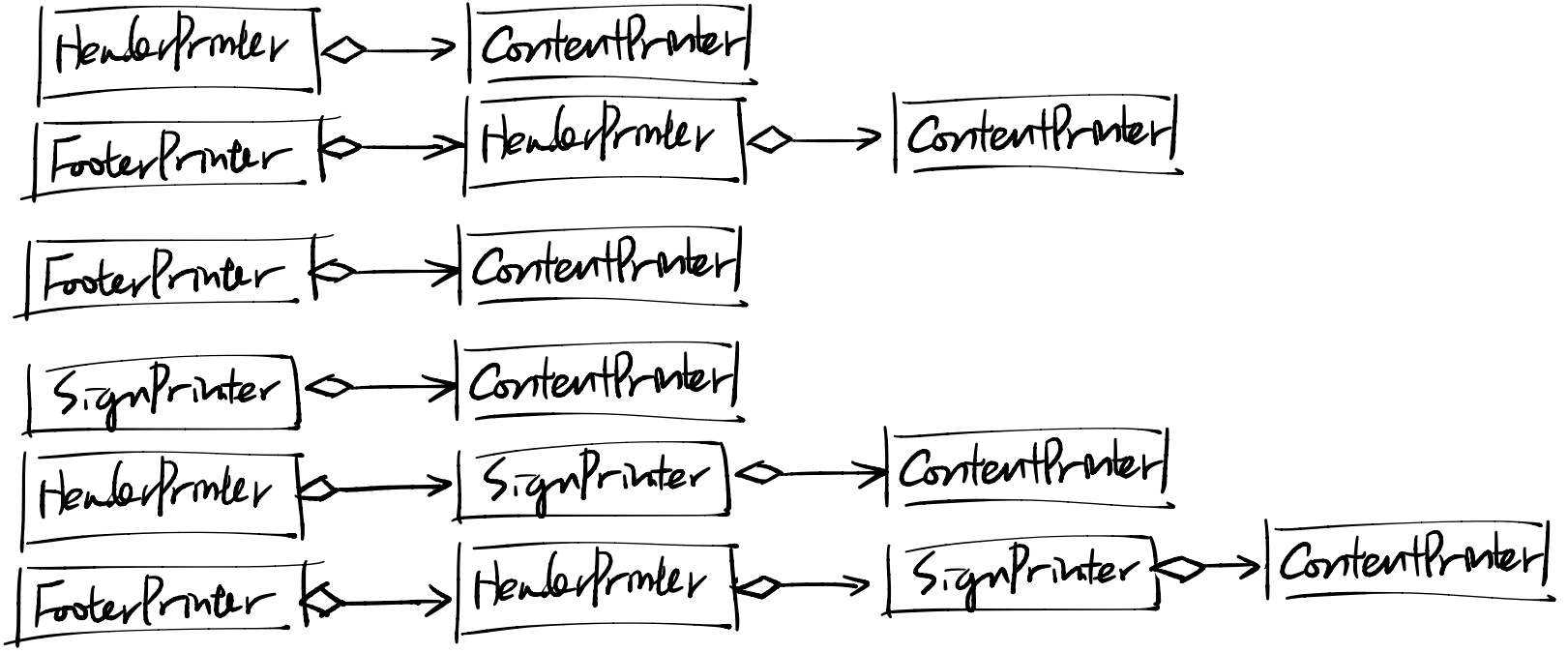
↳ 이러 기능은 상황에 따라 필요할 경우에 제한 } 필요한 만큼만 활용
↳ 기능을 붙였다 뗐다 자유롭게 할 수 있다



"Decorator"
(장식품, 액세서리)

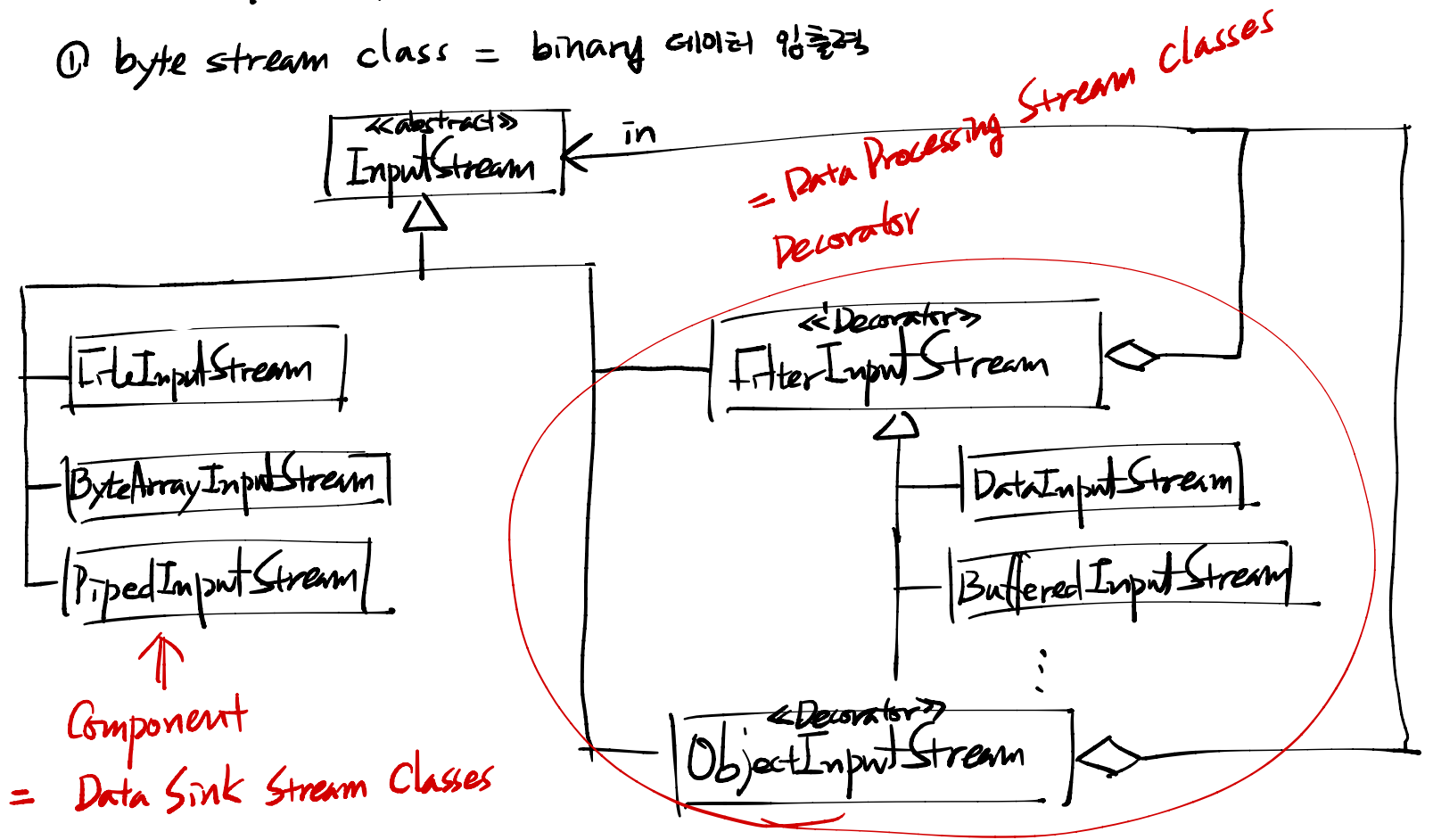
* Printer 종류


ContentPrinter
HeaderPrinter
FooterPrinter
SignPrinter

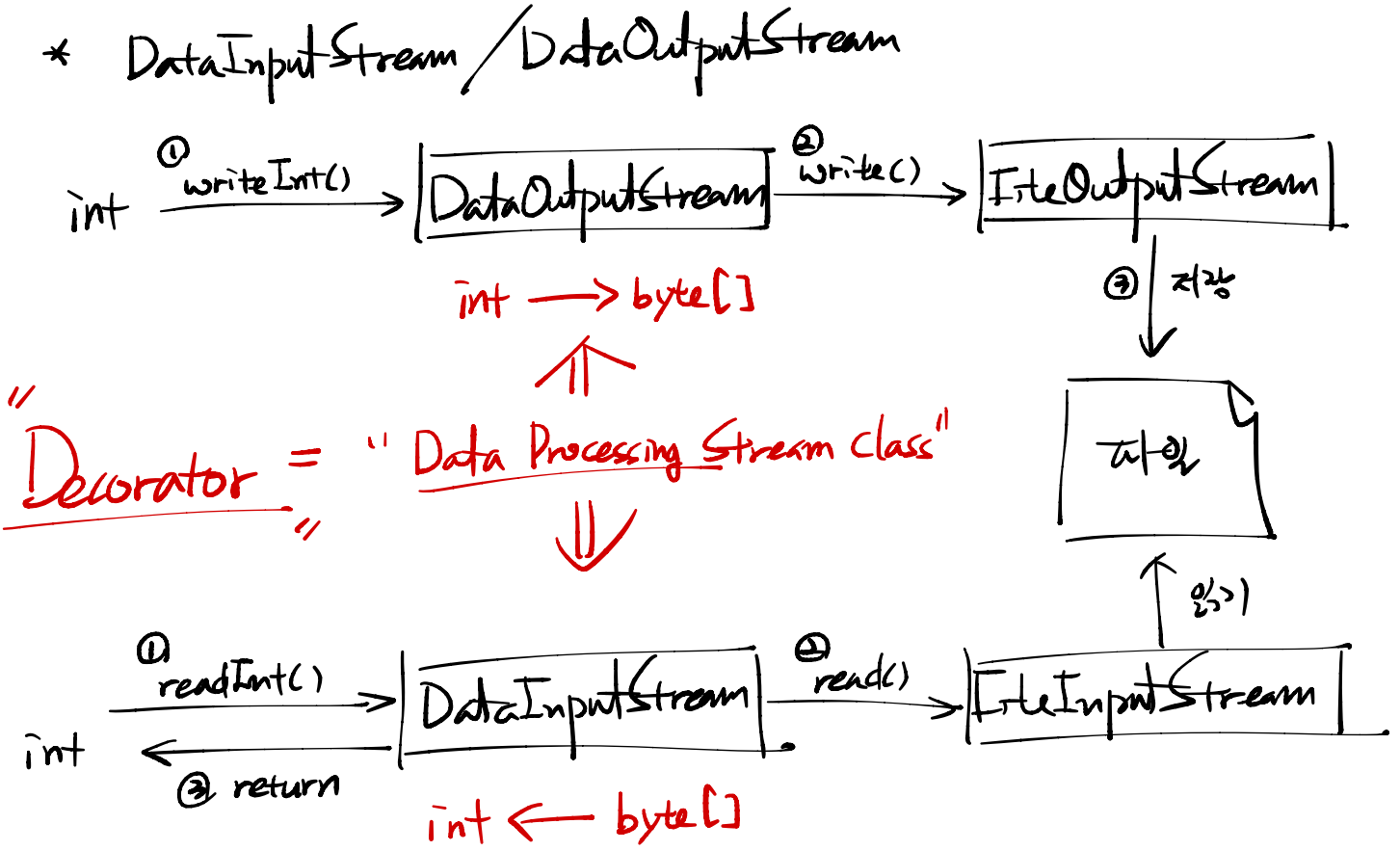

HeaderPrinter
ContentPrinter
FooterPrinter
HeaderPrinter
ContentPrinter
FooterPrinter
ContentPrinter
SignPrinter
ContentPrinter
HeaderPrinter
SignPrinter
ContentPrinter
FooterPrinter
HeaderPrinter
SignPrinter
ContentPrinter

* File I/O API 와 Decorator 패턴

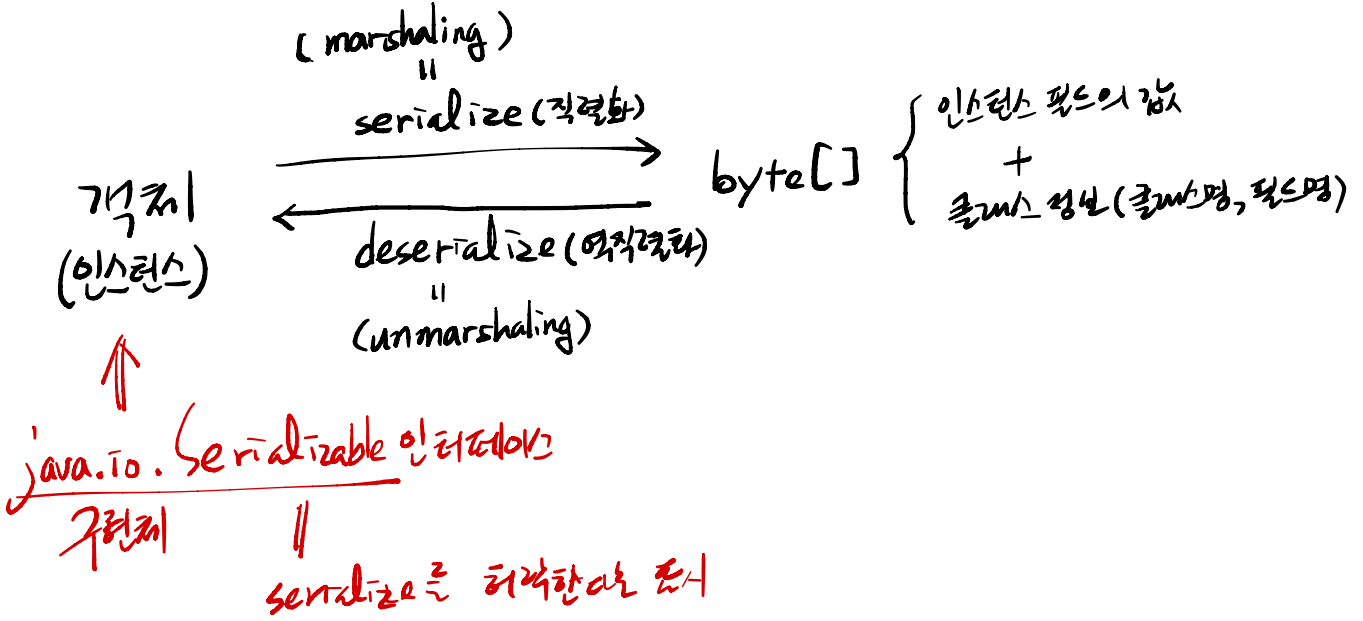
① byte stream class = binary 데이터 입출력



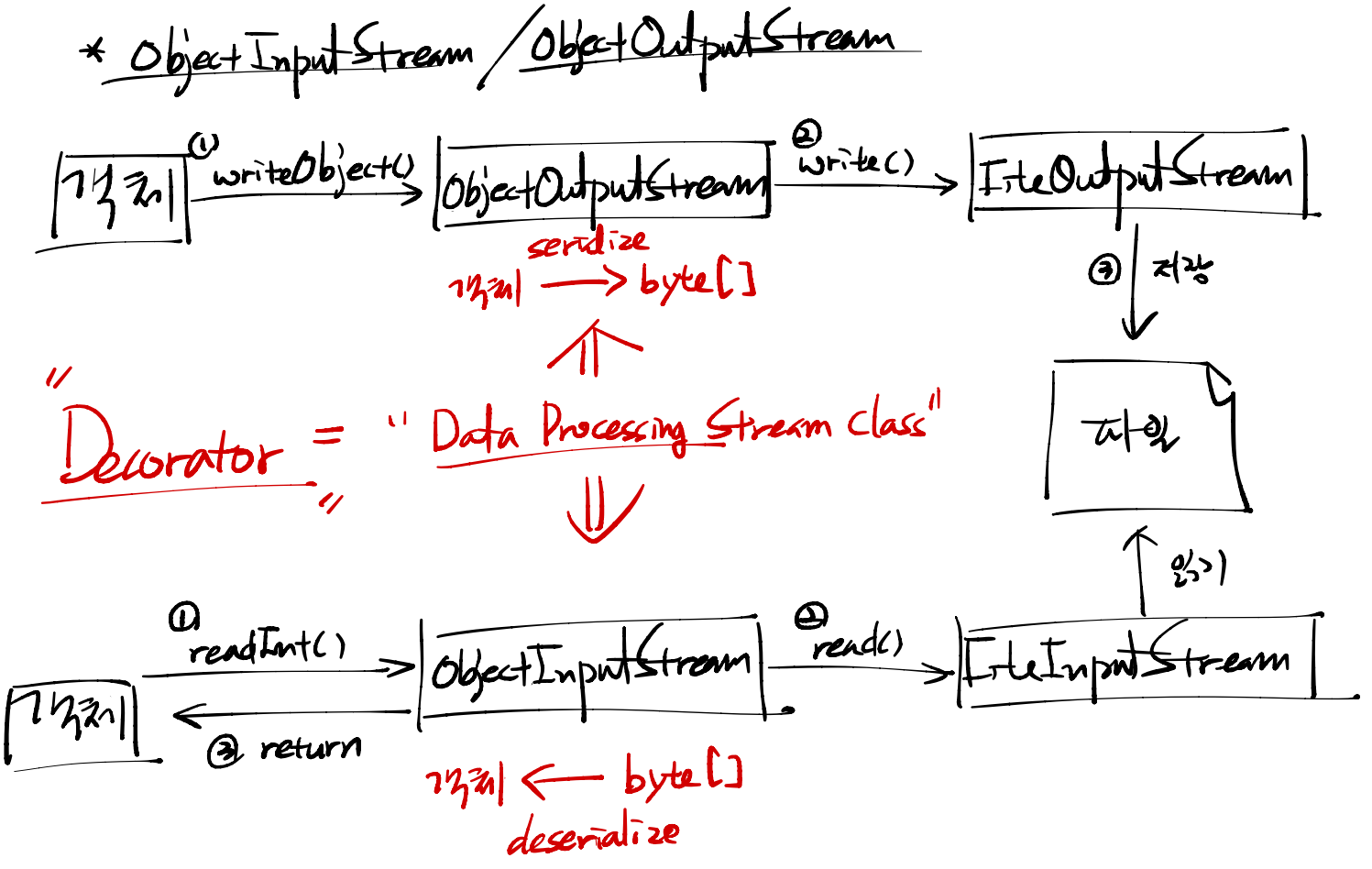
* DataInputStream / DataOutputStream



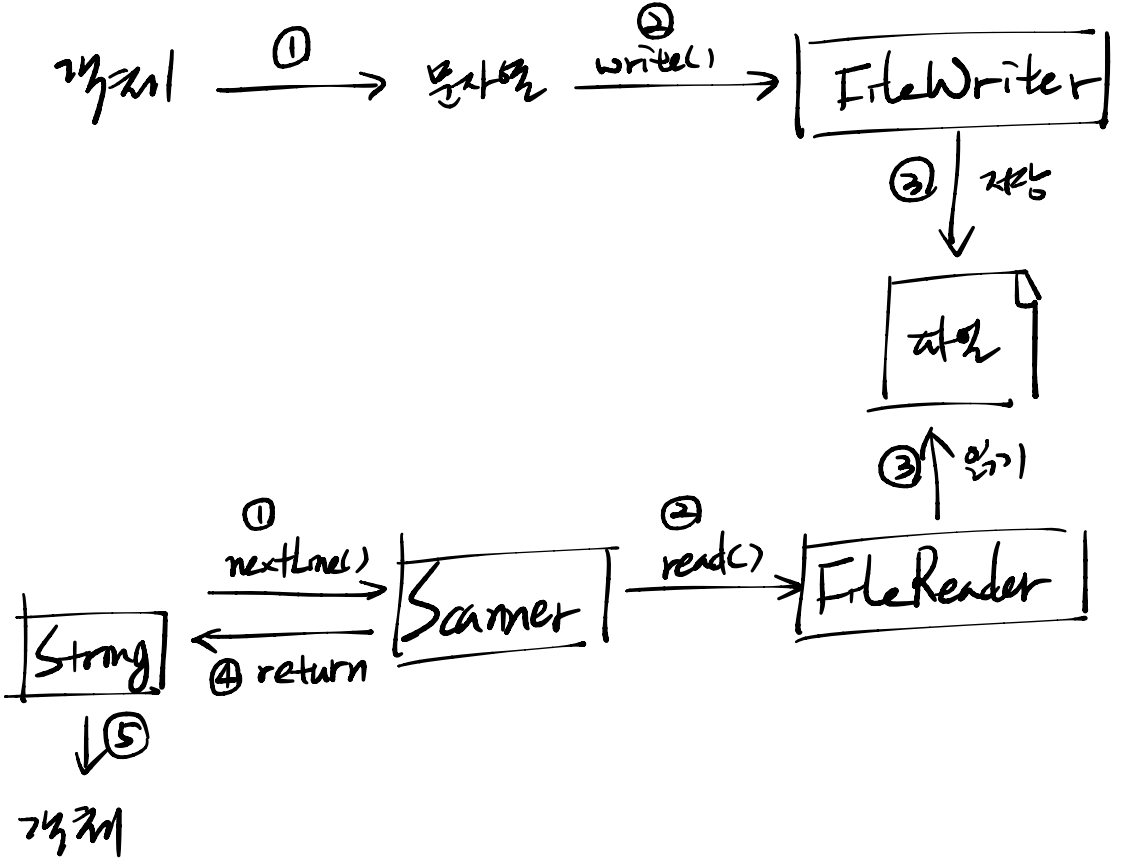
30. File I/O API : 객체 직렬화/역직렬화



* ObjectInputStream / ObjectOutputStream



31. File I/O API : 텍스트 입출력과 CSV 포맷



* CSV 포맷과 String.split() 메서드