


* app -oi -a : Gradle로 프로젝트 풀터 준비하기

사용자 홈 폴더

```
~/git/bitcamp-Study $ mkdir Study-project ↵  
$ cd Study-project  
Study-project $ gradle init ↵
```

- ① 프로젝트 유형: application 선택
- ② 프로그래밍 언어: Java 선택
- ③ 프로젝트 개수: only one 선택
- ④ 빌드 스криプ트: Groovy 선택
- ⑤ 단위 테스트: Junit4 선택
- ⑥ 프로젝트명: Study-project (Default)
- ⑦ 패키지명: com.ecomcs.pms

```
$ gradle -g run ↵
```

* app-01-a : class diagram

[미니 프로젝트 관리 시스템 PMS]

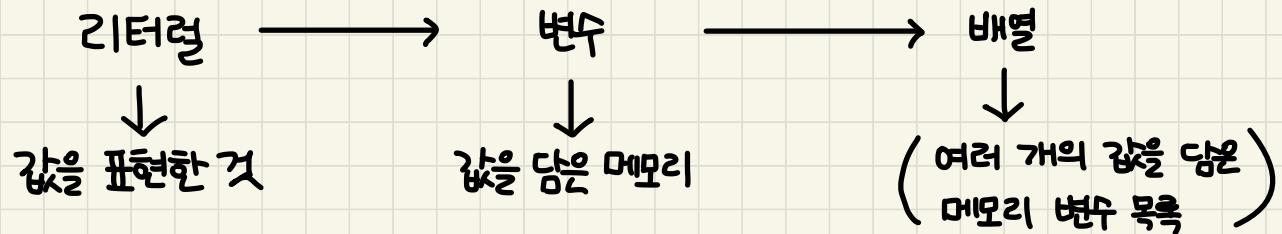
* /src/main/java



패키지명 아님! X

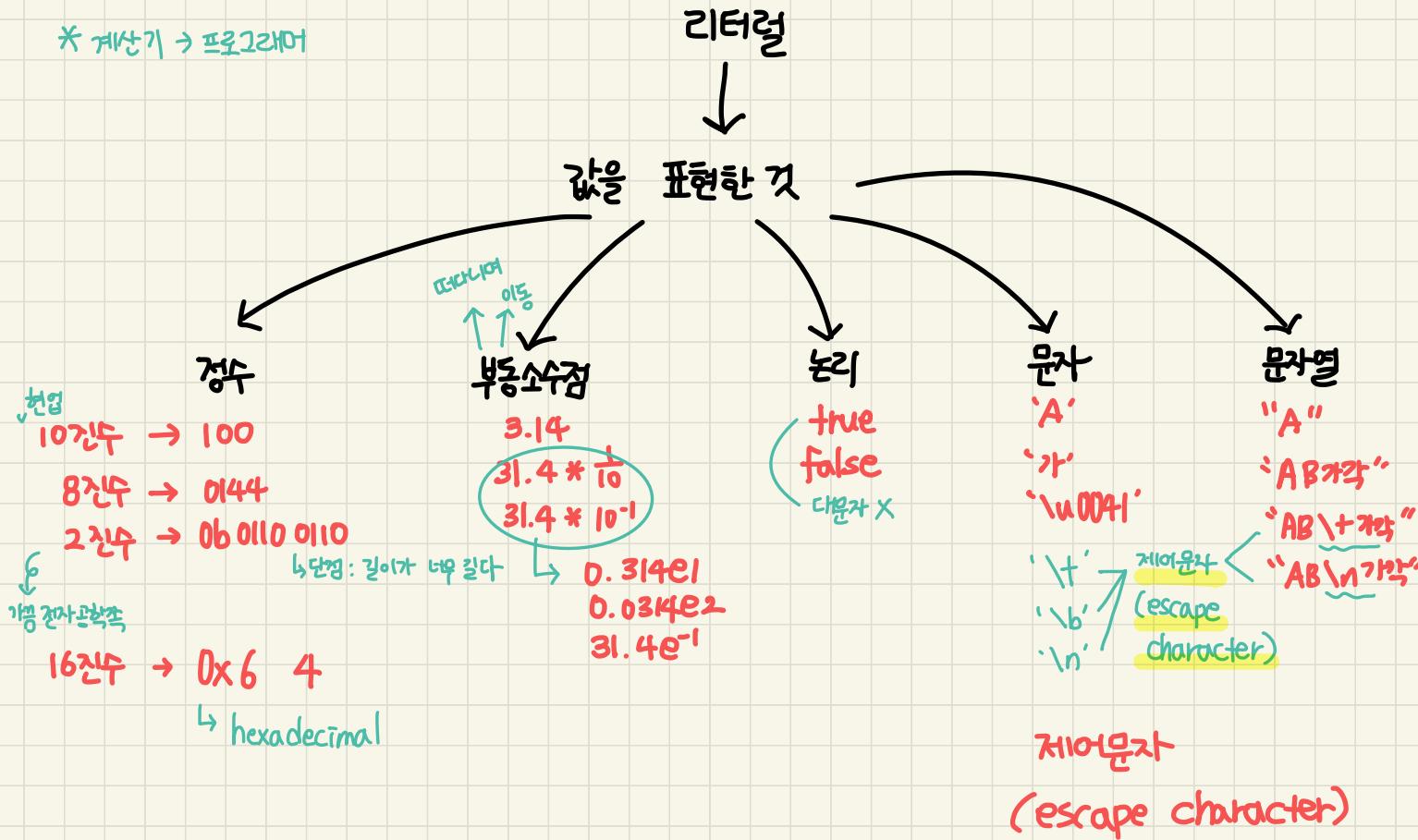
source 파일 두는 폴더!

* app - 02 : 리터럴 (literal), 변수 (variables), 배열 (array)



* app-02-a : 리터럴과 콘솔 출력

* 계산기 → 프로그래머



* app - 02 - a : 리터럴과 콘솔 출력

콘솔 출력

값체 (what)

값이나 표현식 (expression)

System.out.print() ⇒ 값 출력

What
↓
do
↓
소속
여기면
자동차
형광등

System.out.println() ⇒ 값 출력 후 줄바꿈

System.out.printf("출력형식", 값, 값...)

format

메서드 = 함수 (do)



가능수행

출력 형식에 맞춰 값 입력

%s : 문자열

%d : decimal 10진수

%x : hexadecimal 16진수

%c : character 문자

%b : boolean true/false

* 02-b : 변수와 키보드 입력

정수 → int no;

문자열 → String name;

날짜 → java.util.Date registeredDate;

new java.util.Scanner (System.in);

- import java.util.Scanner 임포트한 경우

New Scanner(System.in); 으로 생각 가능

* 02-C : 배열과 흐름제어문

데이터타입 [] 변수 = new 데이터타입 [개수];

예) int [] no = new int [5];

if (조건) {

====

}

while (조건) {

====

}

for (변수초기화; 조건; 증감문) {

=====

}

* 03-a: main() 메서드의 용도

→ entry point
출입문!

조건문, 반복문을 활용 연습

회원 입력/출력



· main() {} -3

명령어에 따라서
작업하도록 변경!

프로젝트 입력/출력



· main() {} -3



작업 입력/출력



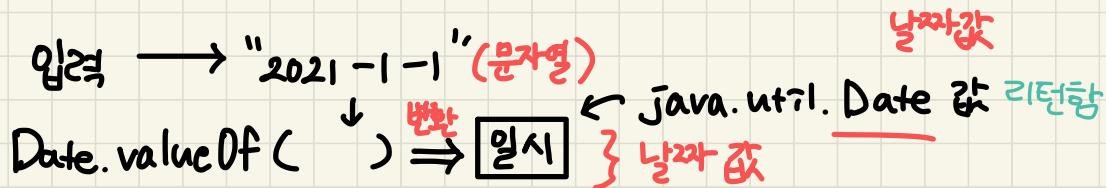
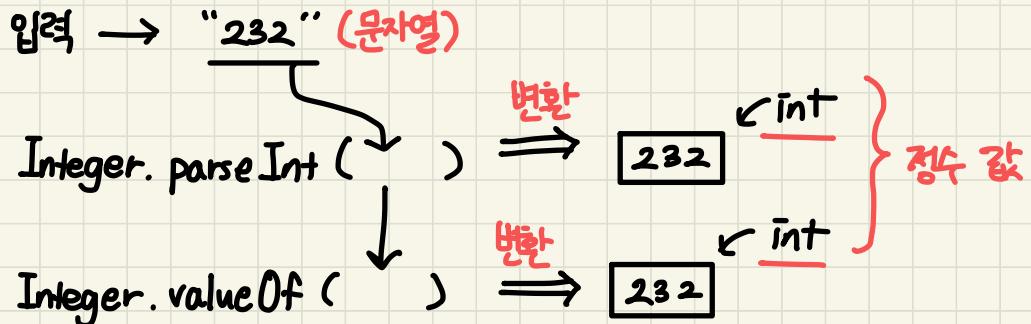
· main() {} -3



main() {}
조건

entry
point

- ✓ 회원 입력/출력
- ✓ 프로젝트 입력/출력
- ✓ 작업 입력/출력



2021-1-1 향식X
↳ 예외 발생

illegal . argument

Integer.parseInt(sc.nextLine())

사용자가 키보드로 입력한 문자열을 정수 Int값으로 리턴해준다.

```
Scanner sc = new Scanner(System.in);  
String input = sc.nextLine();
```

Scanner는 Input을 받다가
q를 입력하면 종료

↓
<or>

```
if (input.equals("q") || input.equals("Q")) {  
    break;  
}  
}
```

} else if (input.equals("/~")){

* tNo[tSize] = Integer.parseInt(sc.nextLine());

Size는 배열

addMember()

이름부여
멤버입력



tSize + 1 *

* prompt(String title)

프롬프트 실행하여면 문자열을 입력해주세요

* 03-b. 메서드 사용법 : 메서드 활용

Class A {

main() {

}

기능단위로
관리하기

+ 재사용하기가 쉽다

쉽게

메소드를
묶는다.

add Member () {

}

}

}

}

}

listMember () {

}

}

}

add Project () {

}

}

* 이렇게 기존 코드를 유지보수하기
쉬운 구조로 재작성하는 것을
“리팩토링 (refactoring)”이라
부른다

* 클래스 변수와 로컬 변수

Class X {

 Static int C = 300;

 Static void m1() {

 int a = 100;

 } == 로컬 변수

 메서드가
 끝나면
 사용할 수 없다.

 사용

}

 Static void m2() {

 int b = 200;

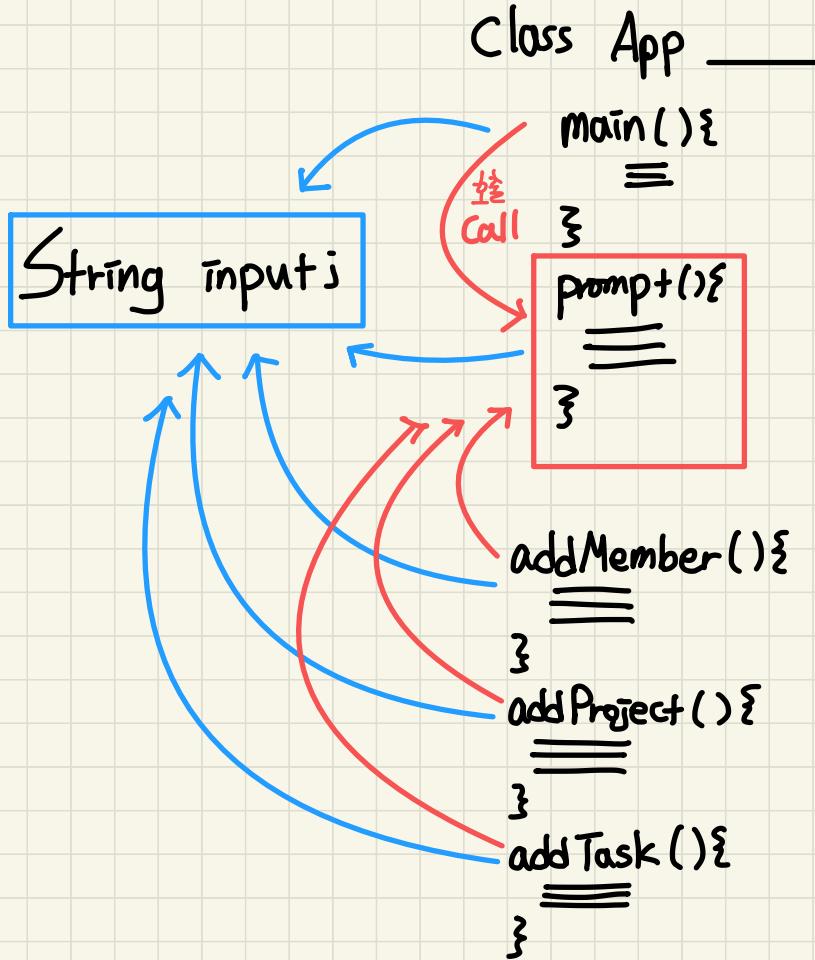
 } == 로컬 변수

}

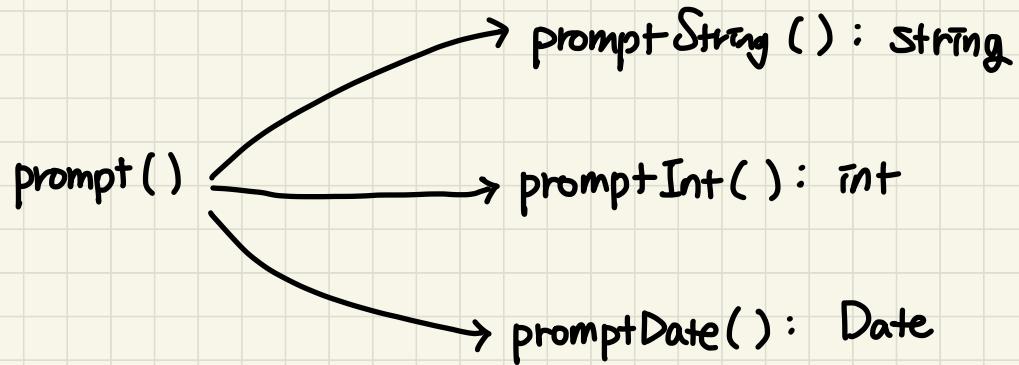
 다른 메서드의
 로컬 변수는
 접근할 수 없다.

사용

* prompt()



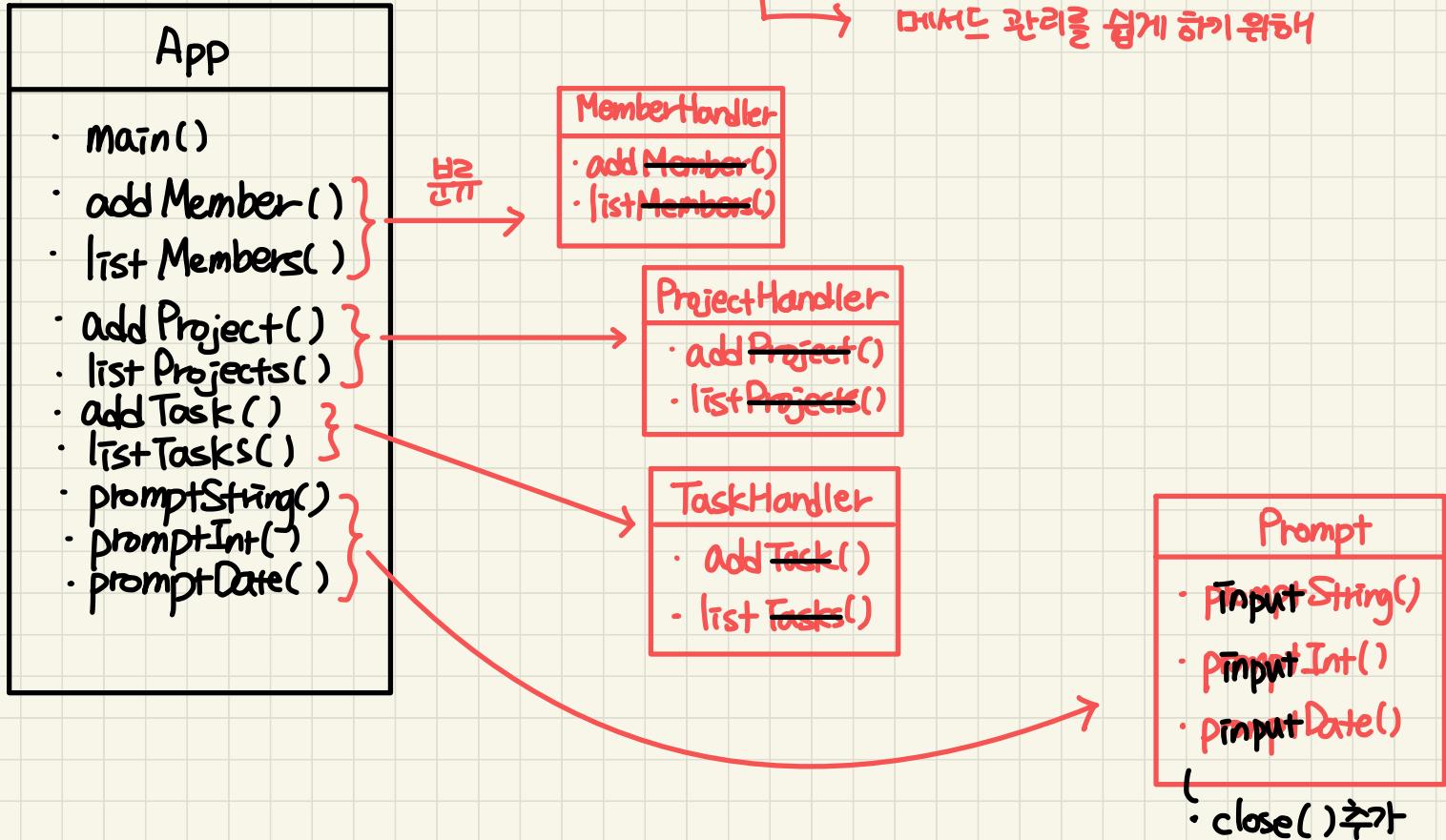
* prompt() 기능을 더 세분화



* 04-a : 클래스 문법을 이용하여 메서드를 분류하기



메서드 관리를 쉽게 하기 위해



* 클래스 문법의 용도

① 서로 관련된 일을 하는 메서드들을

관리하기 쉽게 한 클래스에 모아두는 것



역할에 따라 메서드 분류

② 복합 데이터를 저장하는 메모리를 설계



사용자 정의 데이터 타입 만들 때

개발자

(user-defined datatype)

* 클래스 문법을 사용하여 복합구조의 변수를 설계하기

```
int[ ] no;  
String[ ] name;  
String[ ] email;  
String[ ] password;  
String[ ] photo;  
String[ ] tel;  
Date [ ] registeredDate;
```



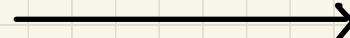
낱개의 변수로

여러 사람의

정보를 다룸



회원정보를 관리하기 쉽게



묶기



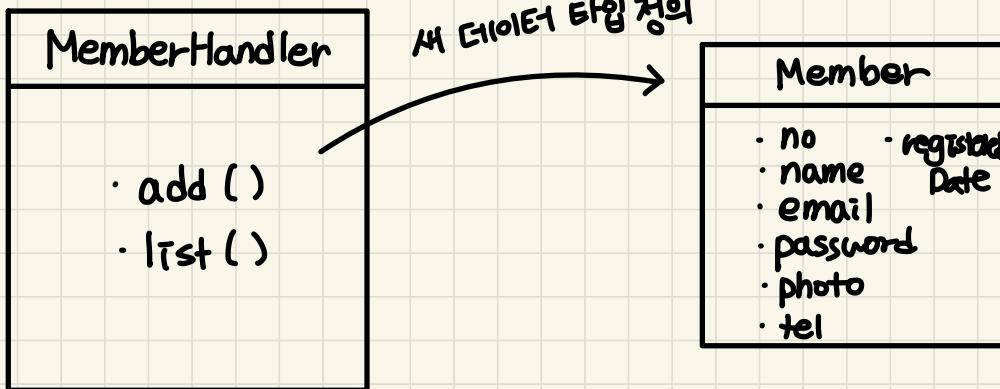
여러 개의 변수로 구성된

새로운 구조의

복합 데이터 타입을 정의

"사용자 정의 데이터 타입"

```
class Member {  
    int no;  
    String name;  
    String email;  
    String password;  
    String photo;  
    String tel;  
    Date registeredDate;  
}
```



* Member 인스턴스 만들기

클래스 설계도에 따라 Heap 영역에
변수를 만들라는 명령 *

참조변수

Member

클래스명
(변수의 데이터 타입)

member

↑
Member 인스턴스의
주소를 저장하는 변수
"

new

Member();

클래스명

"Member의 레퍼런스(reference)"라

부른다

member

3700

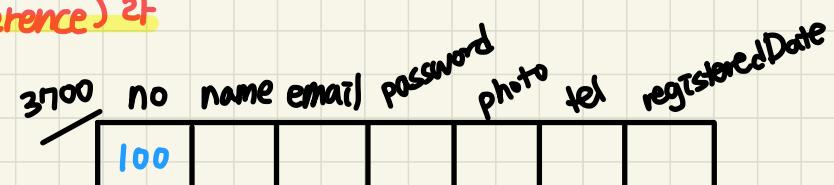
인스턴스
변수에 값 저장

member.no = 100;

인스턴스의 변수

인스턴스의 주소를 담고 있는

레퍼런스 (참조변수). (멤버변수)

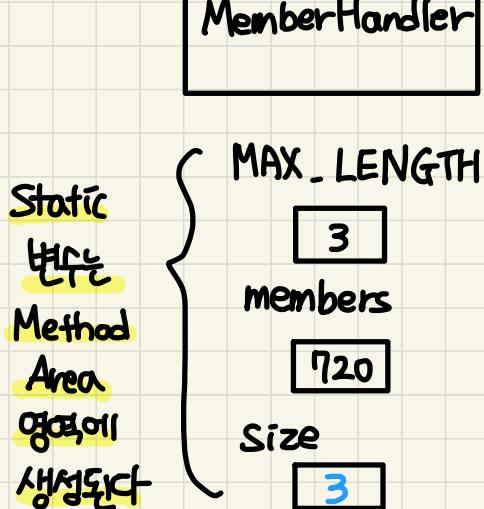


↑ Member 설계도(클래스)에 따라
만든 변수들

"Member의 인스턴스(instance)"라
부른다 !!

* MemberHandler.add()

Method Area



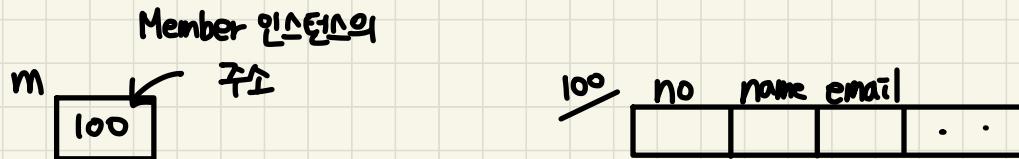
JVM Stack

Heap:

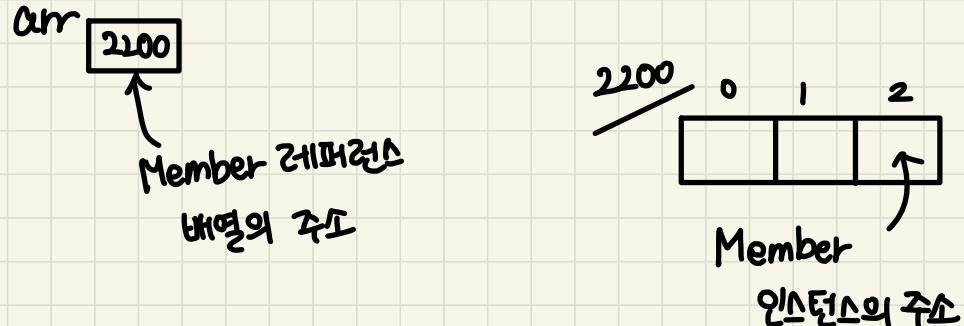
	0	1	2
720	2200	2700	3100
2200	no	name email	
	1	a a@	...
2700	no	name email	
	2	b b@	...
3100	no	name email	
	3	c c@	...
	.	.	.
			1007H

* 레퍼런스와 레퍼런스 배열 그리고 레퍼런스 배열을 가리키는 배열 레퍼런스

Member m = new Member();



Member[] arr = new Member[3];

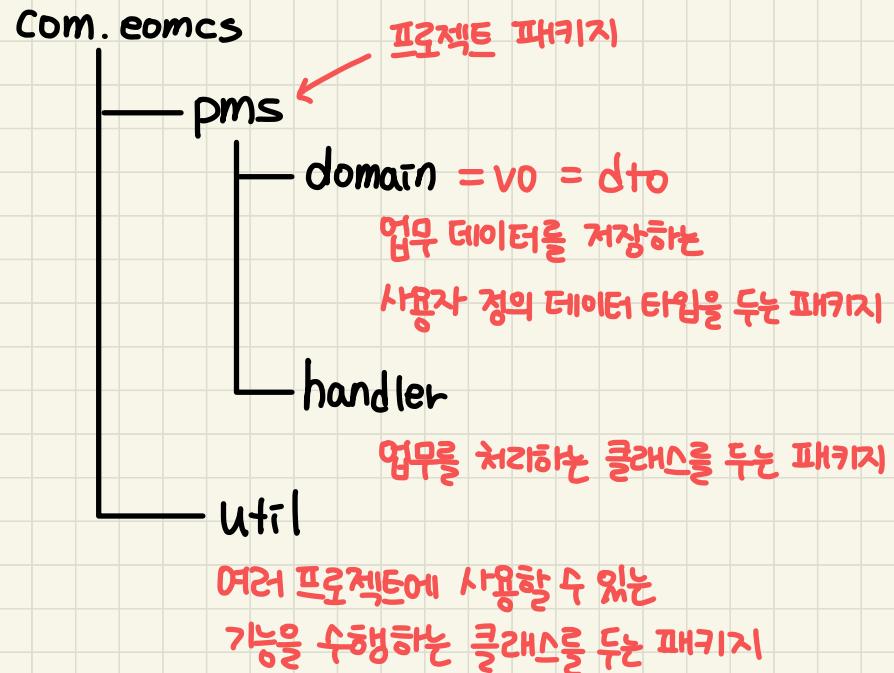


* 04-C . 패키지 적용

Com. eomcs . pms

→ 클래스를 역할이나 용도에 따라
관리하기 쉽게 분류하는 방법

- App
- Member
- MemberHandler
- Project
- ProjectHandler
- Task
- TaskHandler
- Prompt



domain 객체

- ↳ 업무에서 다루는 데이터를 표현하는 클래스
- ↳ 사용자 정의 데이터 타입을 정의하는 클래스

예) 학생 데이터 → Student 클래스

강사 " → Teacher "

주문 " → Order "

제품 " → Product "

게시글 " → Board "

↳ 값을 표현한다고 해서 "Value Object (VO)"라고도 부른다

↳ 데이터를 실어나르는 역할을 한다고 해서

"Data Transfer Object (DTO)"라고도 부른다

Member [] members;

members = new Member [5];

members [0] = new Member();

members [1] = new Member();

members

2200

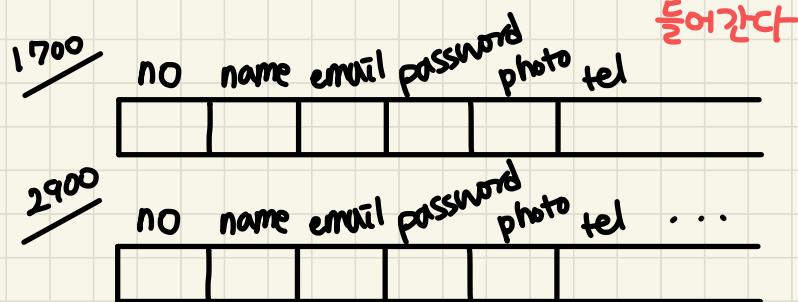
리퍼런스

배열의 주소가 들어있다

new Member [5]



리퍼런스: Member 인스턴스의 주소가 들어간다

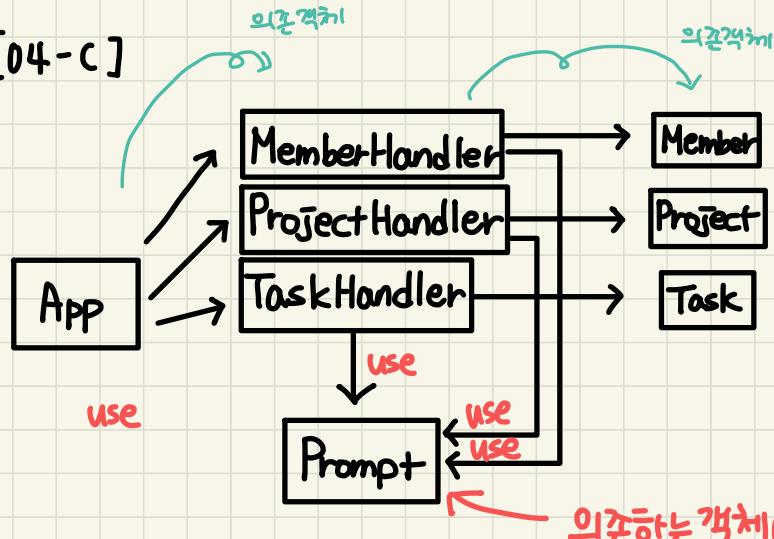


Member의
인스턴스

↑
new Member()

* 04-d 클래스 사용법 - 의존관계

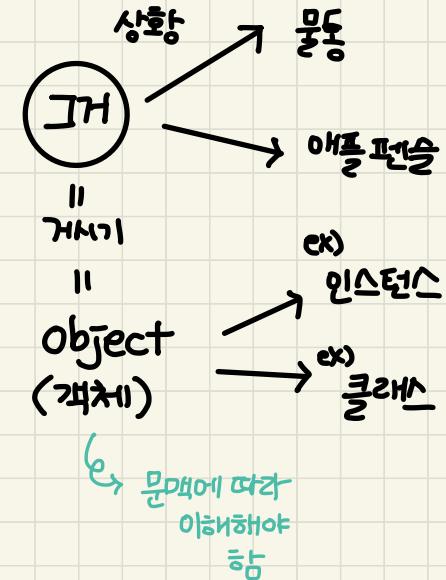
[04-c]



* App이나 XxxHandler 클래스는
Prompt를 사용한다

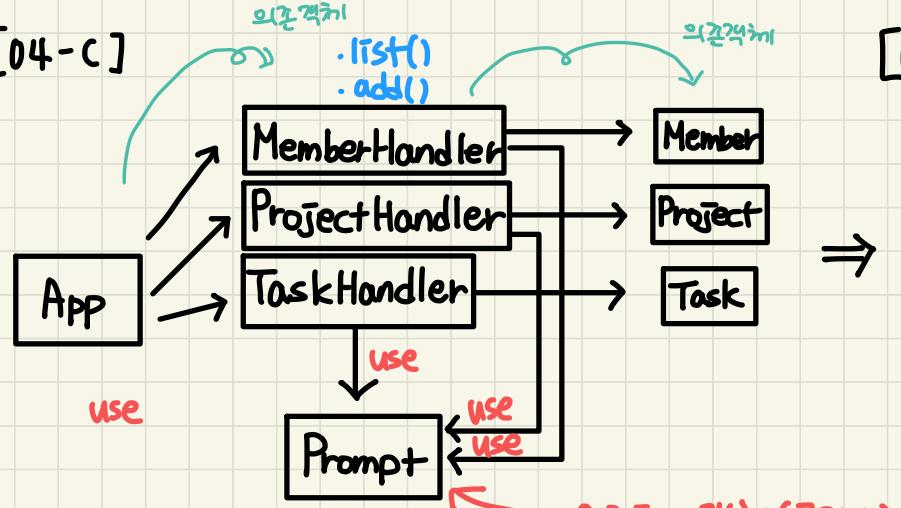
↳ “의존한다”라고 표현한다
↳ prompt 없는
동작 불가능

“의존 객체”라 부른다
(dependency)



* 04-d 클래스 사용법 - 의존관계

[04-c]



[04-d]



* App이나 XxxHandler 클래스는
Prompt를 사용한다

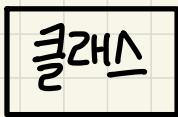
"의존 객체"라 부른다
(dependency)

↳ "의존한다"라고 표현한다

↳ prompt 없는
동작 불가능

서비스를
사용하는 입장

Client



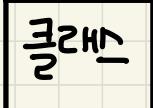
a는

Call

||

use

Supplier



b를 사용한다

ex) App

의존관계

MemberHandler

(dependency)

의존객체

Member [] members = new Member [10];

members

2000

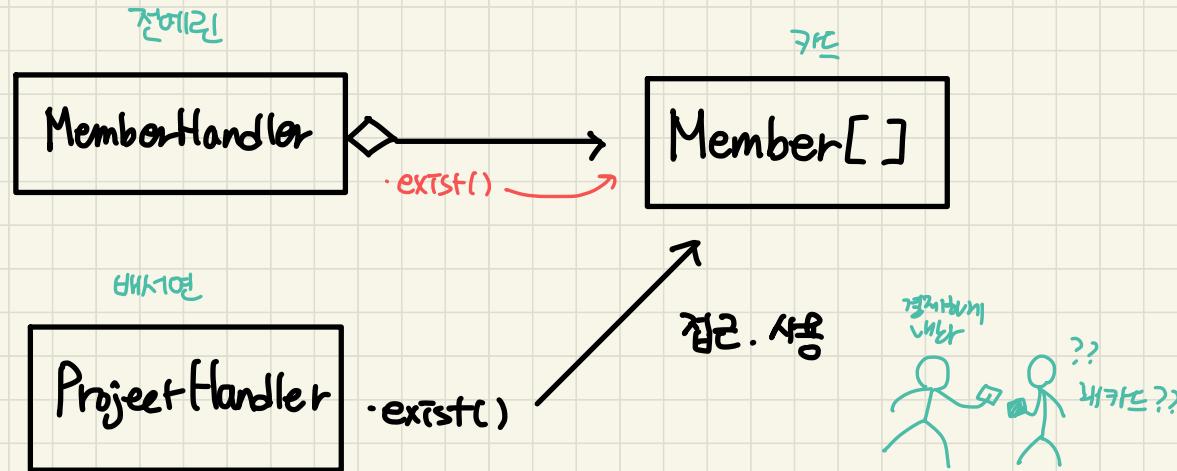
Size ~~Stellt~~

23

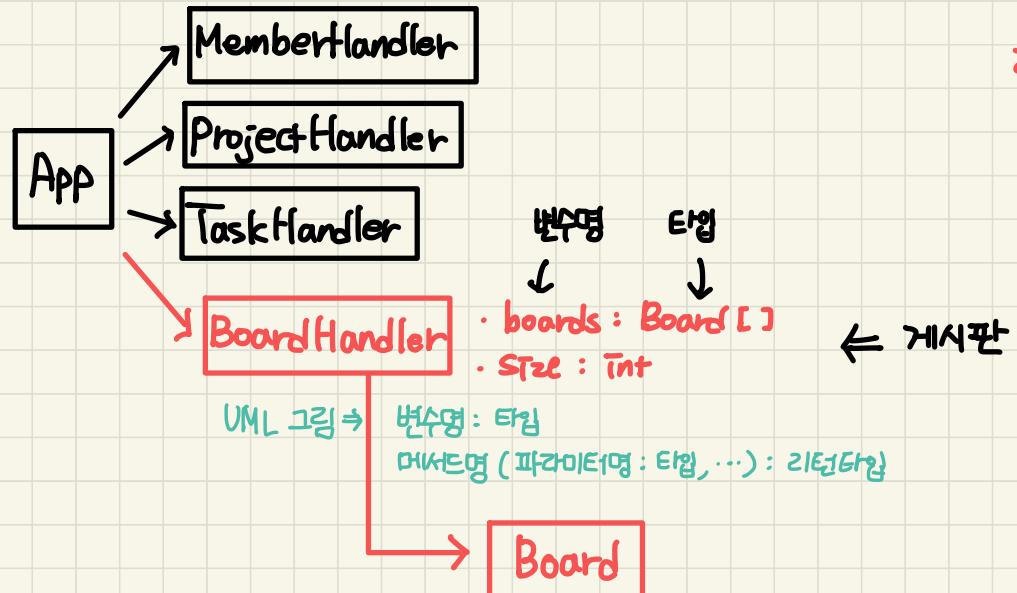
exist("bbb");



* Information Expert 설계 패턴 (GRASP)

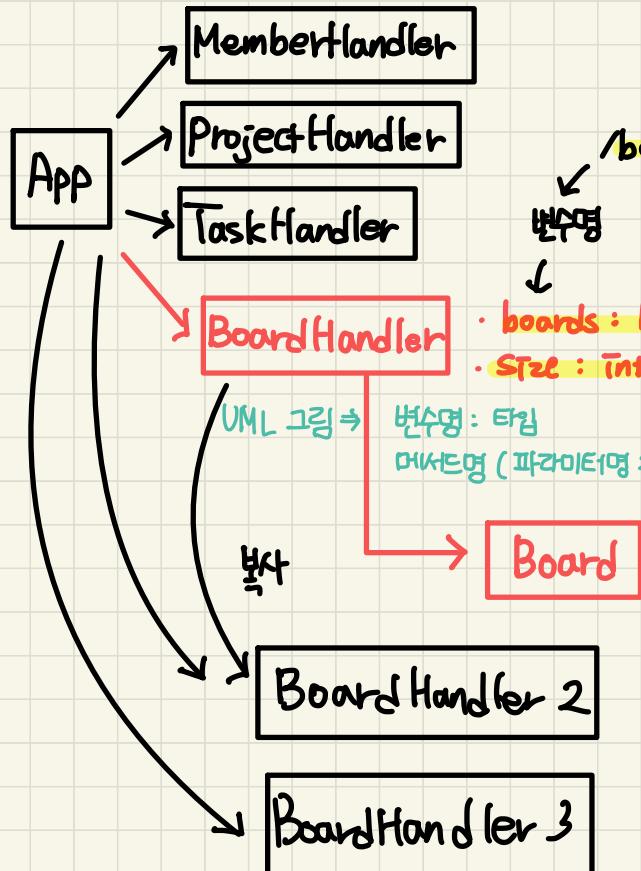


* 05-a 인스턴스 사용법 : 클래스 필드와 클래스 메서드의 한계



한 개의 개시판이 있는 경우

* 05-a 인스턴스 사용법 : 클래스 필드와 클래스 메서드의 한계



여러 개의 게시판을 만들 경우,

/board/add 명령으로 등록한 게시글을 보관한다.

변수명 템
↓ ↓
· boards: Board[]
· size: int

↳ 게시판

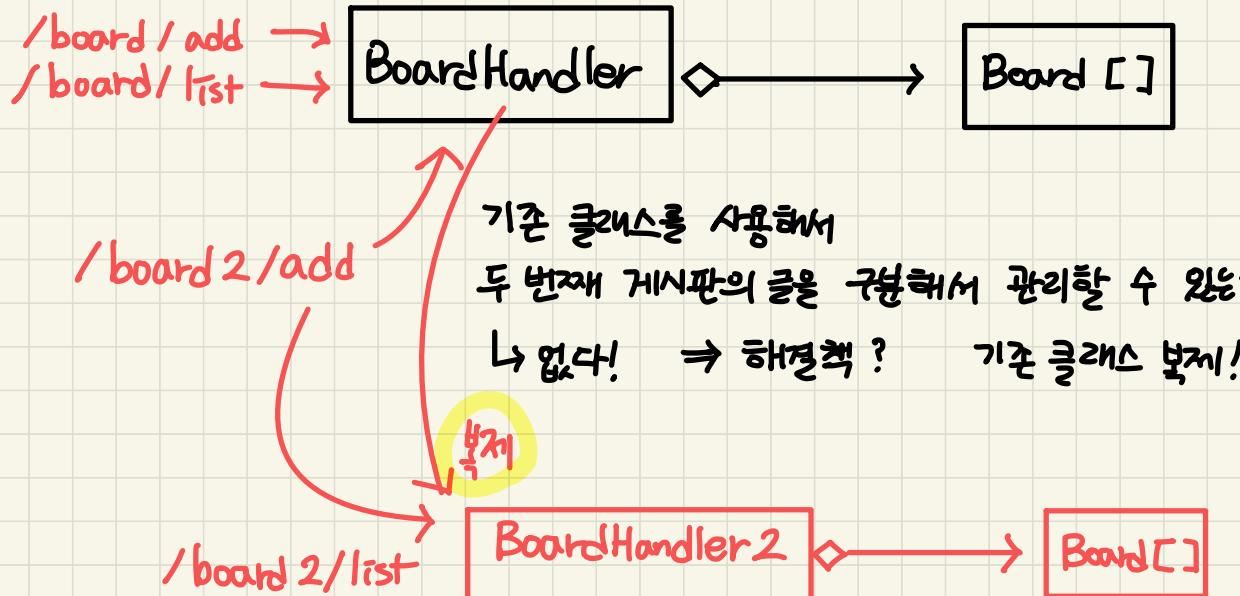
/board2/add 를 등록한 게시글은?
/board3/add " "?
/board4/add " "?

↳ 게시판을 추가할 때마다

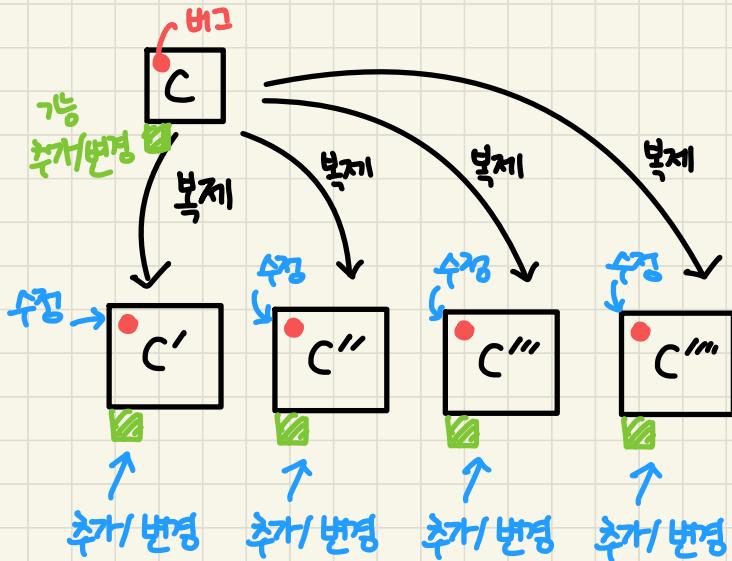
기존 클래스를 복제하여 처리한다.

↳ 왜? 한 클래스에 한 개의 배열만
있기 때문이다.

* 클래스를 복제하여 기능을 추가하기



* 클래스를 복제하여 기능을 추가하는 방식의 문제점



문제점 1. 버그가 있으면, 복제 클래스 모두 변경해야 한다.

↳ 유지 보수가 어렵다.

문제점 2. 기능(추가)하면 모든 복제 클래스를 변경해야 한다.

↳ 유지 보수가 힘들다.

* 05-b 인스턴스 사용법: 인스턴스 필드가 필요한 이유와 사용법

Class BoardHandler {

```
static final int MAX_LENGTH;  
static Board[] boards = new Board[MAX_LENGTH];  
static int size;  
:  
}
```

}
field
static 변수(필드)
" "
클래스 변수(필드)

↓
.class 파일이 'Method Area'
영역에 포함될 때 생성된다.

↓
클래스당 한 번만 생성

↓
여러 개의 개시판을
다룰 수가 없다.

Static 필드를
인스턴스 필드로 전환!
↳ new 명령으로
필요할 때마다
생성할 수 있다.

해결책 ←

* 05-b 인스턴스 사용법: 인스턴스 필드가 필요한 이유와 사용법

Class BoardHandler {

 Static final int MAX_LENGTH;

 Static Board[] boards = new Board[MAX_LENGTH];

 Static int size;

 :

 Static void add() {

 ...

boards [size+] = board;

 Static void list() {

 for (int i = 0; i < size; i++) {

 =

 }

}

}

Static 메서드는
Static 변수를 (에)
직접 접근하여
사용할 수 있다.

* 05-b 인스턴스 사용법: 인스턴스 필드가 필요한 이유와 사용법

Class BoardHandler {

 Static final int MAX_LENGTH;

~~Static~~ Board[] boards = new Board[MAX_LENGTH]; }

~~Static~~ int size;

 :

}

Static 메서드는

Static 변수를

직접 접근하여

사용할 수 있다.

모든 게시판의 배열 개수가 같기 때문에

? 이 변수는 여러 개를 만들 필요 없다.

↓
게시판마다 따로
생성해야 하는
변수만

} 인스턴스 필드로 전환

* 05-b 인스턴스 사용법: 인스턴스 필드가 필요한 이유와 사용법

Class BoardHandler {

 Static final int MAX_LENGTH;

~~Static~~ Board[] boards = new Board[MAX_LENGTH];

~~Static~~ int size;

 :

 Static void add (BoardHandler that) {

that.boards [that.size ++] = board;

 Static void first (BoardHandler that) {

 for (int i = 0; i < that.size; i++) {

 ==

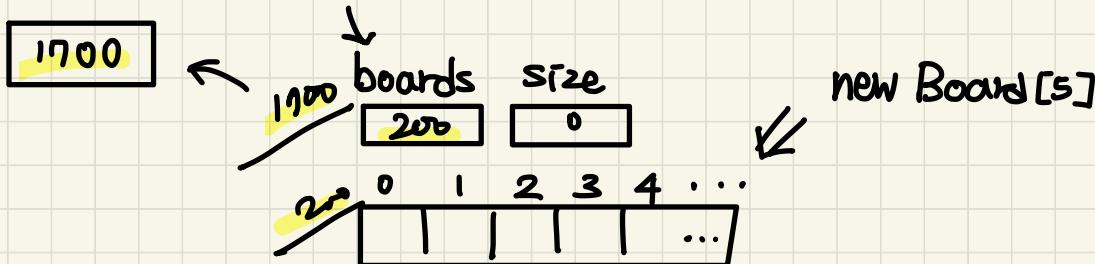
 }

}

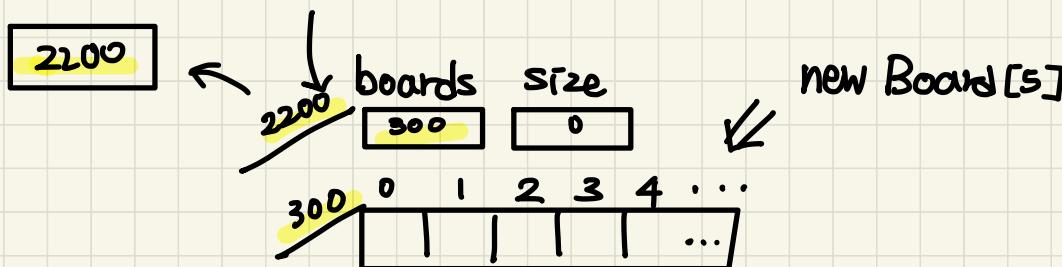
Static 메서드는
인스턴스 주소업이
인스턴스 변수에
접근 불가!

new Board Handler()

BoardHandler boardHandler = new Board Handler();

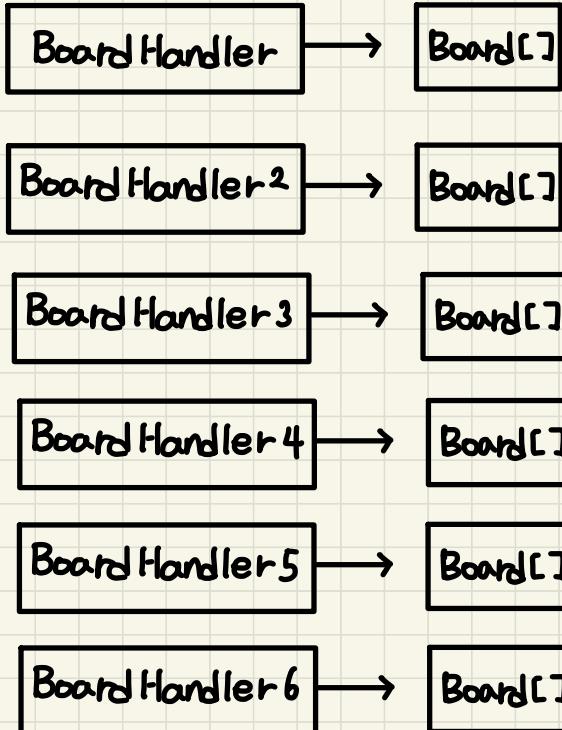


BoardHandler boardHandler2 = new Board Handler();



* 클래스 메서드와 인스턴스

자기 클래스 변수 사용



기능 변경/추가/삭제가 된다!



Board Handler

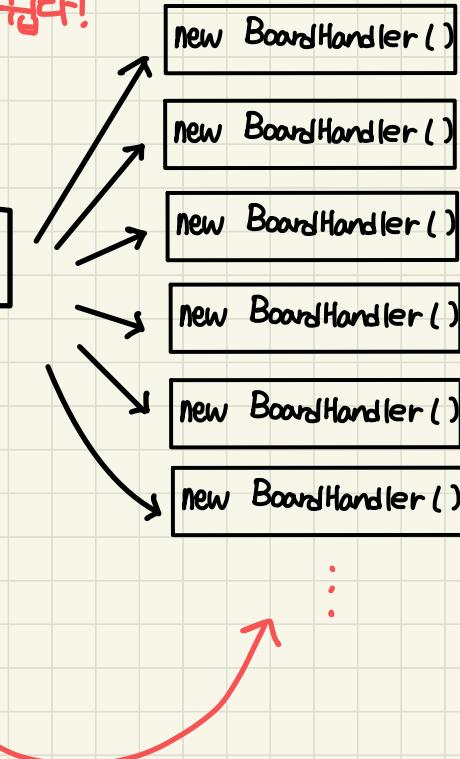
* 거시한다

별도의 클래스를

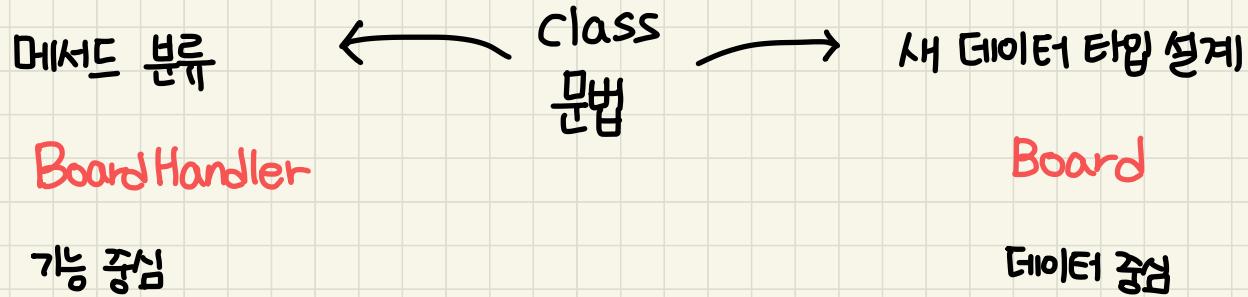
만들 필요가 없다.



대신 거시글을 보관할
배열을 별도로 만든다



new + 클래스(): static이 붙지 않은 변수는
모두 Heap에 메모리를!



new Board()
↳ 명령의 의미

- {
 - ① Board의 인스턴스 생성
 - ② Board 설계도에 따라 인스턴스 변수를 준비하라
 - ③ Board 데이터를 저장할 변수를 준비하라
- 게시글 데이터를 저장할

데이터 중심

new BoardHandler()

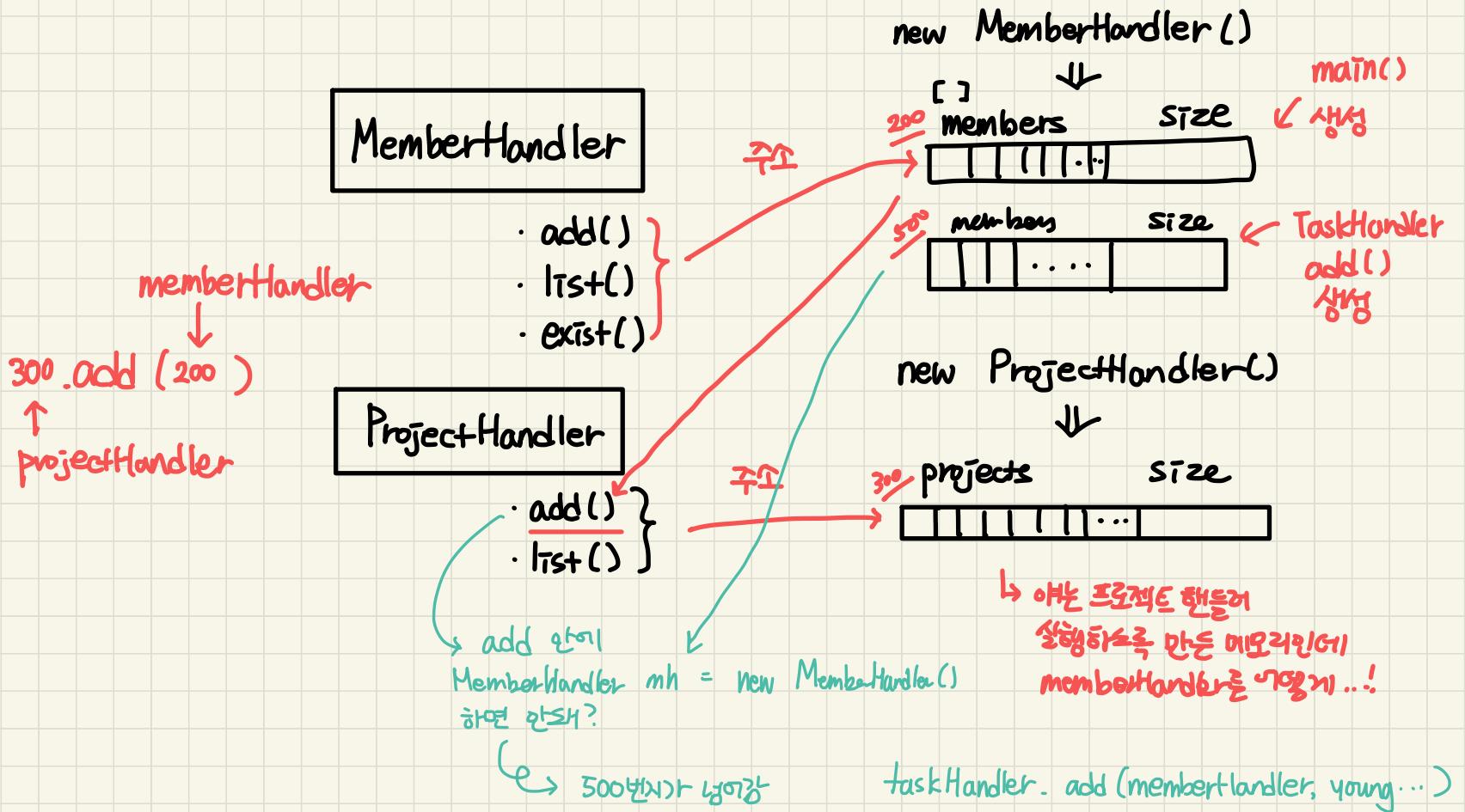
⇓ 명령의 의미

- ① BoardHandler의 인스턴스 생성
- ② BoardHandler 설계도에 따라 변수를 준비하라
- ③ BoardHandler의 메서드가 실행하는 동안 사용할 변수를 준비하라.

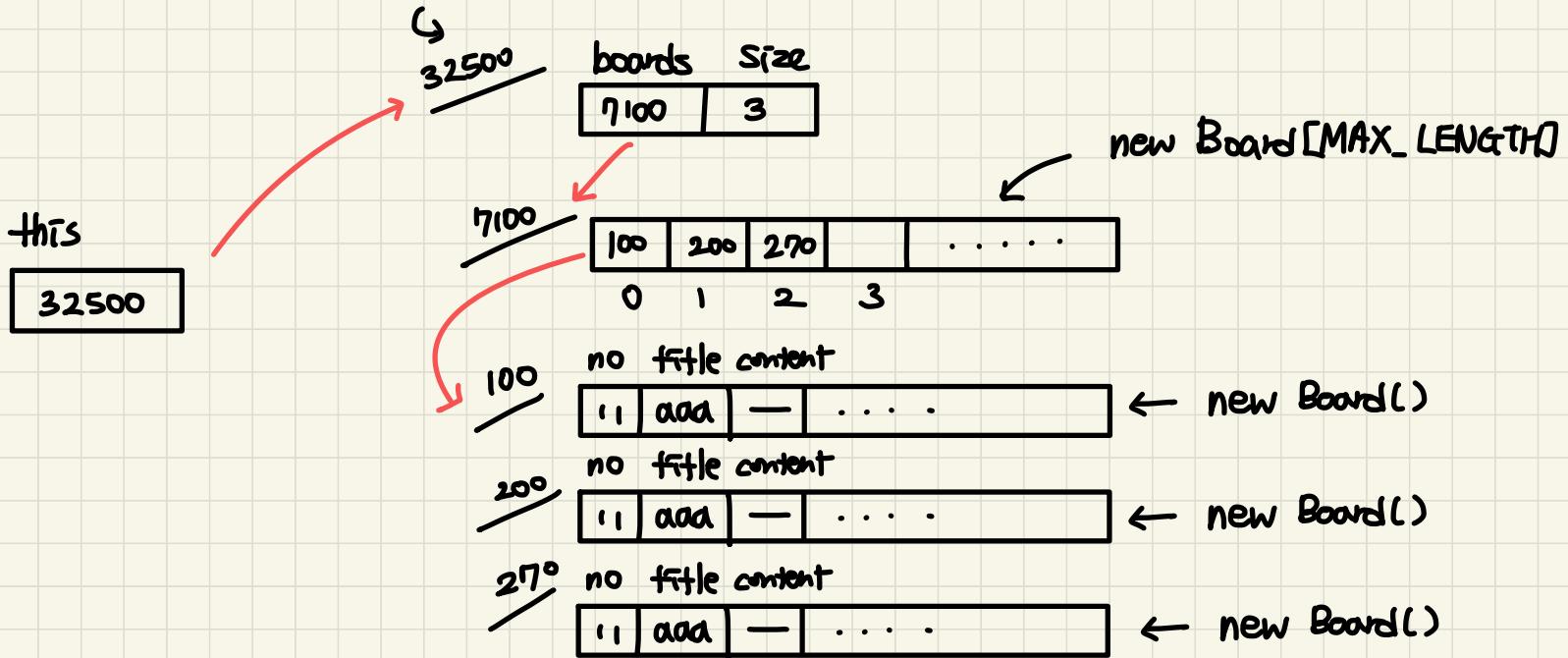
가 작업할 때

☞ 기본적인 설명임.
이상

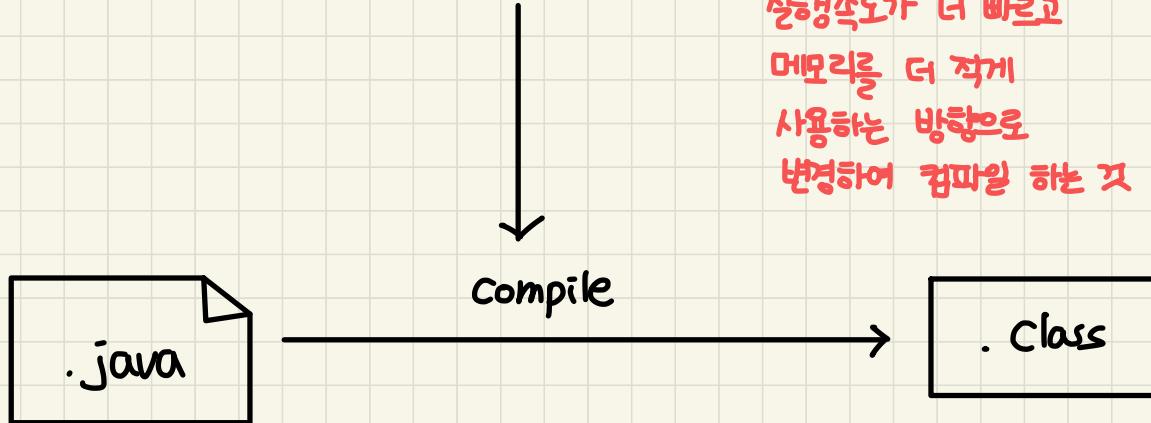
기능중심



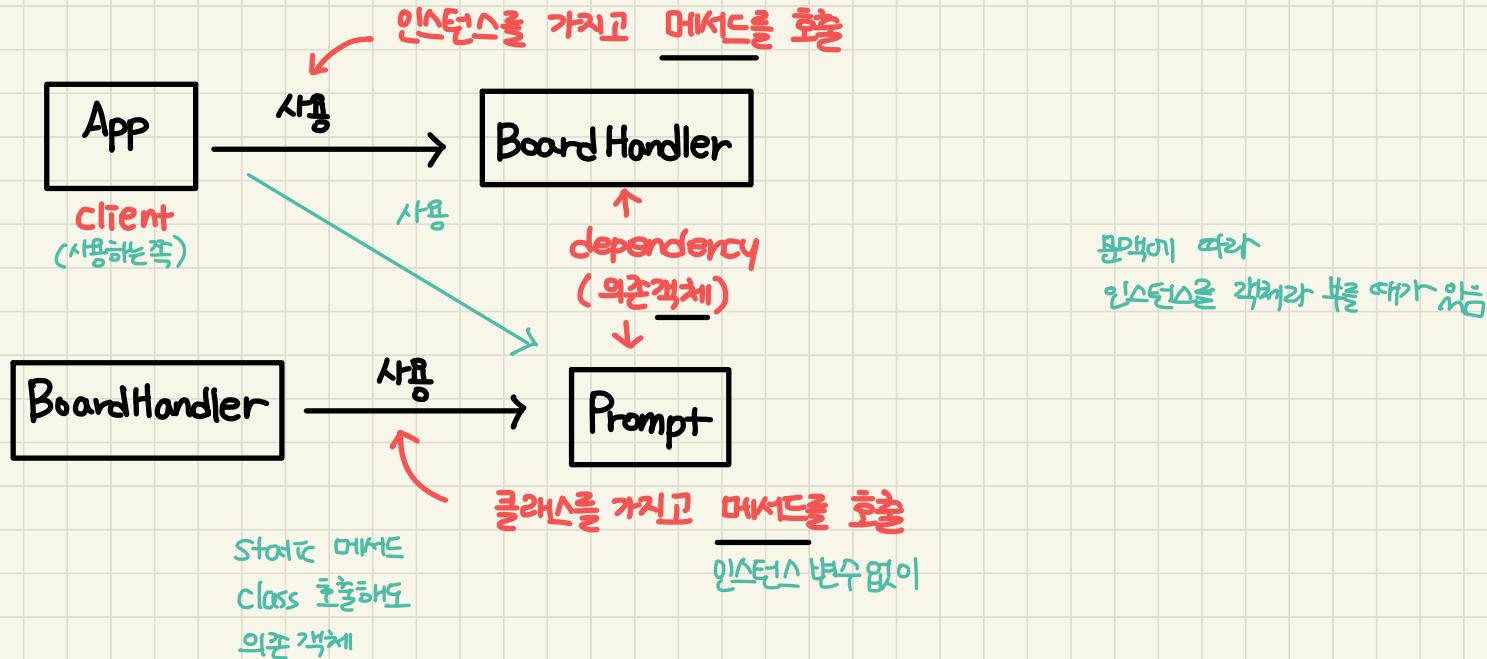
`new BoardHandler()`



최적화 (optimizing) : 개발자가 작성한 코드를
실행속도가 더 빠르고
메모리를 더 적게
사용하는 방향으로
변경하여 컴파일 하는 것



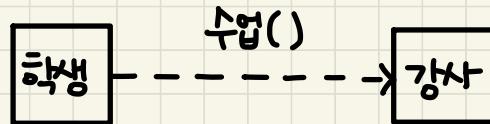
* 07 - a 의존 객체 다루기: 인스턴스 필드에 직접 주입하기



* 클래스 간의 관계

일시적인 의존 관계 (dependency)

- ↳ 특정 메서드에서 일시적으로 사용
 - ~~~~~
 - ~~~~~
 - 파라미터
 - 로컬 변수

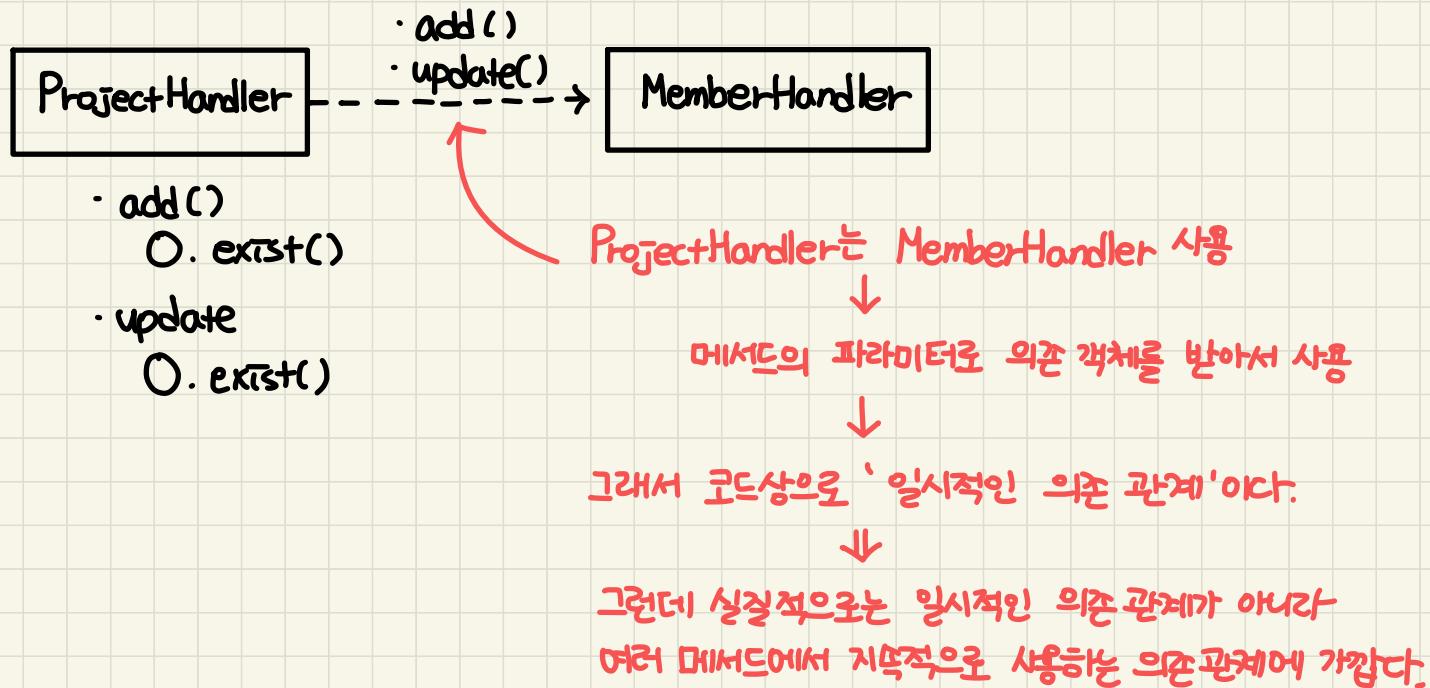


지속적인 의존 관계 (association)

- ↳ 여러 메서드에서 지속적으로 사용
- 인스턴스 필드



[06-b]



[07-a]

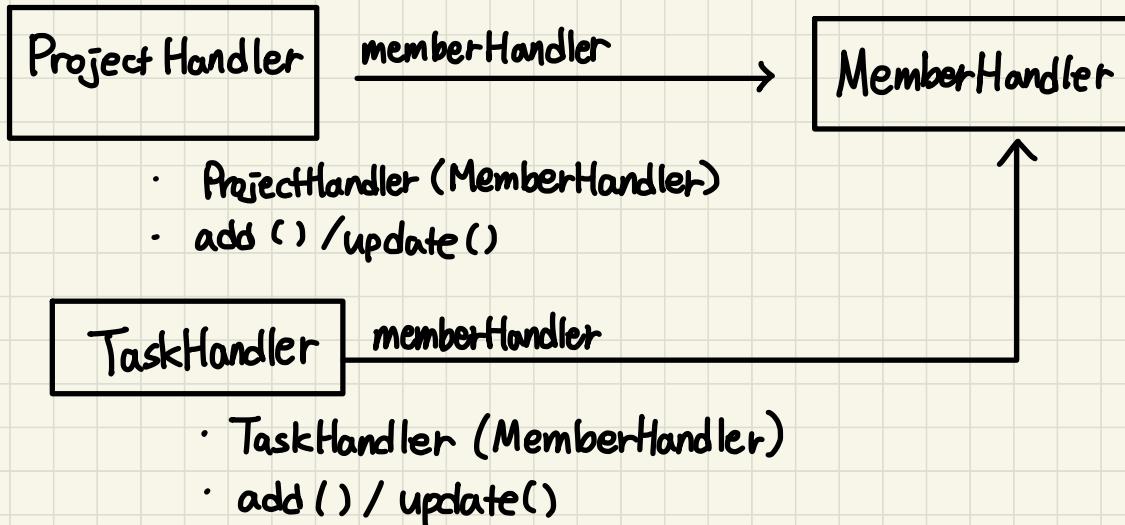
↳ 여러 메서드에서 사용하는 객체라면
임시적인 관계가 아니라 지속적인 관계로 설정하는 것이 유지보수하기가 더 쉽다.

- 메서드의 파라미터
- “로컬 변수”
- 인스턴스 필드



- add()
- update()

* 07-b. 의존 객체 다루기 : 생성자로 주입하기





* 08 - a

Composite 패턴 : 적용 전

App

- main()
- doMainMenu()
- doBoardMenu()
- doMemberMenu()
- doProjectMenu()
- doTaskMenu()



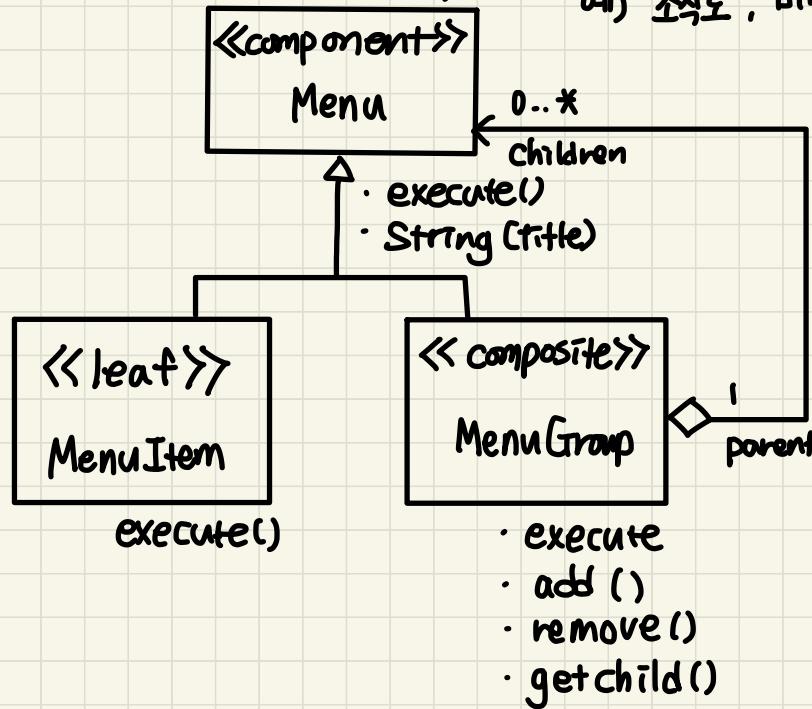
* 08 - b

Composite 패턴 · 적용 후

↳ Tree 구조를 갖는 객체를 다룰 때 유용한 설계 기법

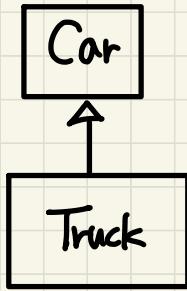


예) 조직도, 메뉴, 파일 시스템



* 클래스 관계

① 상속



class Truck
extends Car {

≡
}



② 연관 (association)

③ 집합 (aggregation)



Lifecycle
Car ≠ Navigation

class Car {
 GasStation obj;
≡ 지속적

}

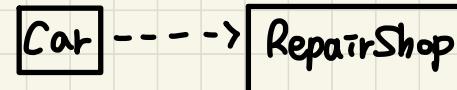
class Car {
 Navigation obj;
≡
}

④ 핍성 (composition)



class Car {
 Engine obj;
≡
}
Lifecycle
Car = Engine

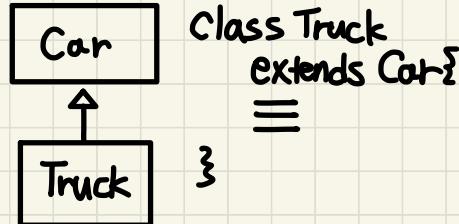
⑤ 의존 (depending)



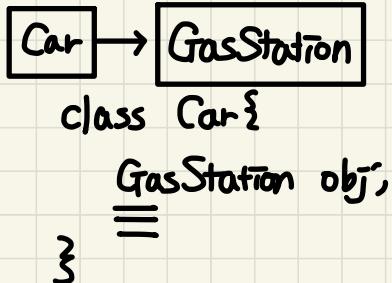
class Car {
 void fillup(RepairShop);
≡
}
일시적인 관계

* 클래스 관계

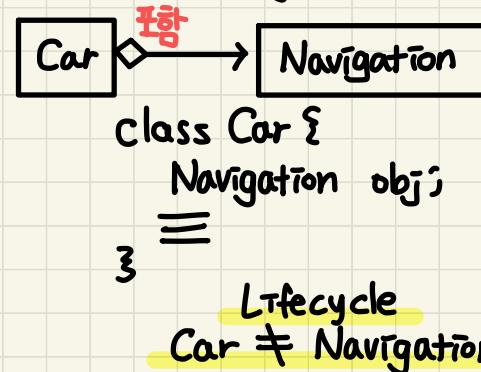
① 상속



② 연관 (association)



③ 집합 (aggregation)



④ 합성 (composition)



class Car {

Engine obj;
≡

}

Lifecycle

Car = Engine

⑤ 의존 (dependency)



class Car {

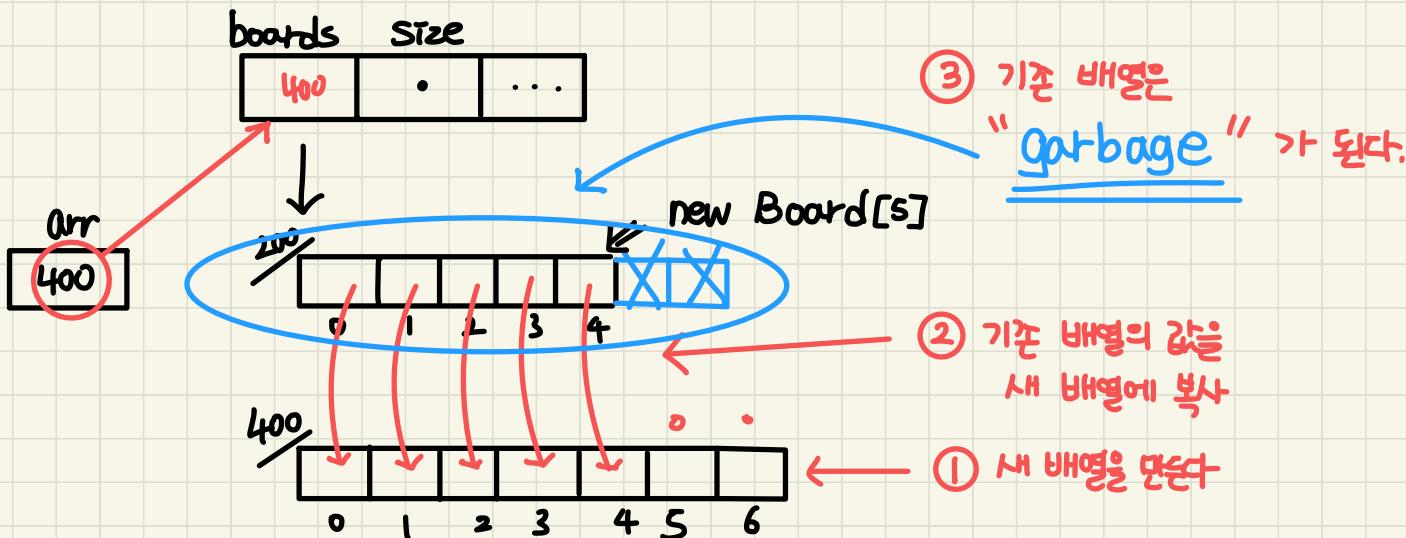
void fillup(RepairShop) {
≡

}

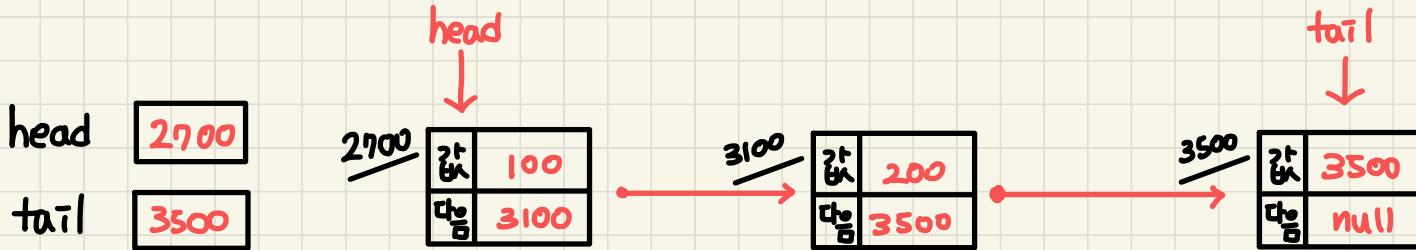
일시적인 관계

* 09-a. 배열 크기 늘리기

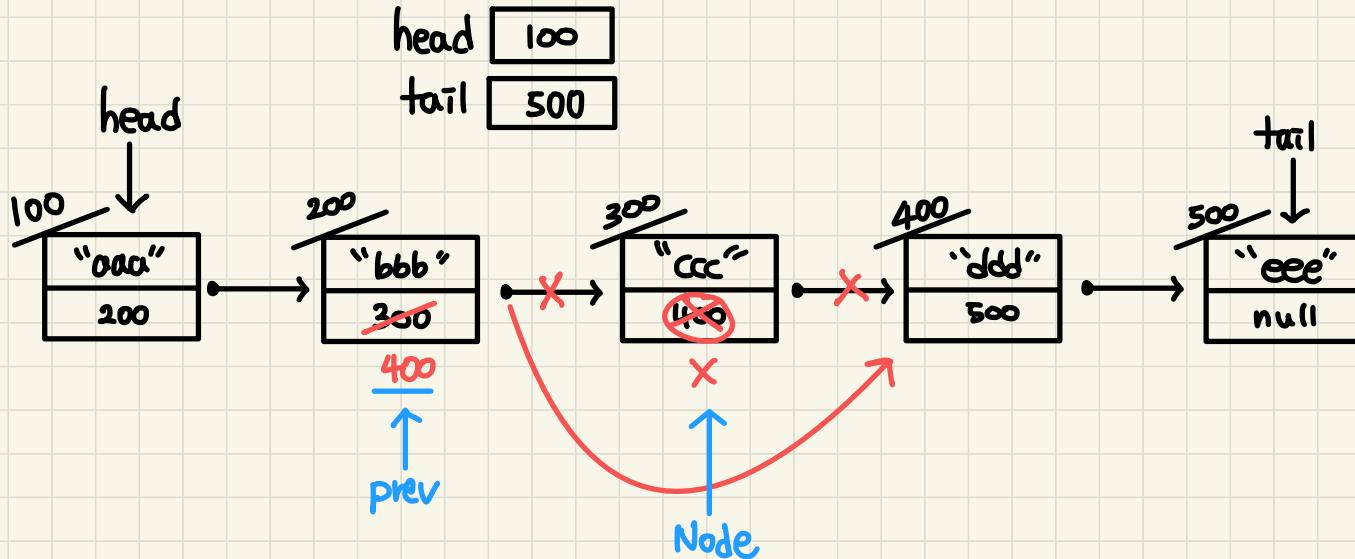
new BoardHandler()



* 09-b. 연결리스트 (LinkedList) 적용



- ① add (100)
- ② add (200)
- ③ add (300)



* 10-a. 데이터 처리 코드를 캡슐화

↳ 복잡한 코드를 분리하여 메서드에 강준다.



그 기능을 사용하는 개발자는

그 기능을 어떻게 구현했는지 자세히 알 필요가 없다.

단지 메서드만 호출하면 되기 때문에
코드가 간편해진다.



복잡한 코드를 감춰서 단순화시키자.

상세한 구현을
메서드 안에
감춘다.

`m()`

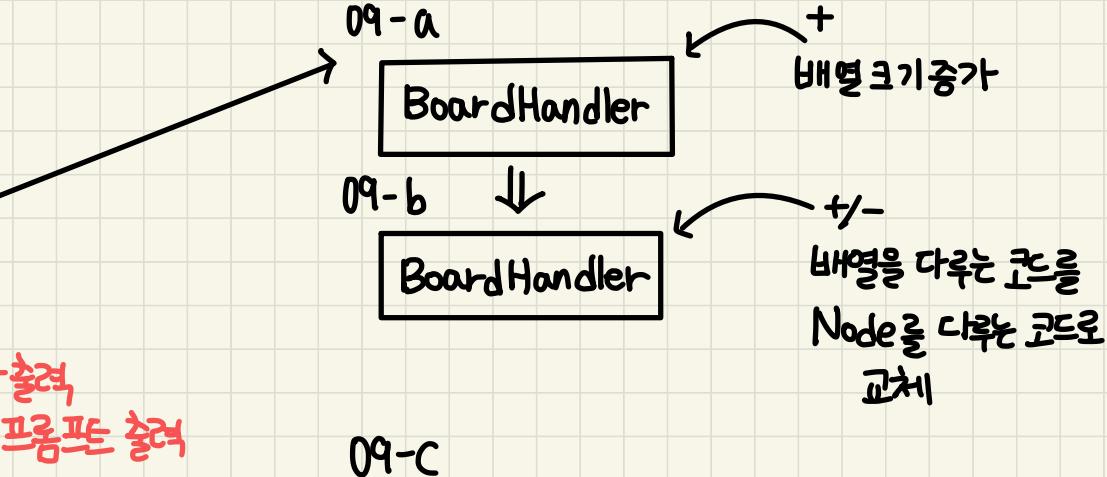
메서드를 이용하는 측에
영향을 주지 않으면서
기능의 구현을 변경할 수 있다.

'캡슐화'의 이점

08-d

Board Handler

- UI 처리
 - ↳ 사용자에 결과 출력
 - ↳ 입력을 위해 프롬프트 출력
- 데이터 처리



High
Cohesion

09-C

Board Handler

Board List

데이터 처리 코드를
"캡슐화" \Rightarrow 배열 다루는 코드를
뜯어내어
별도의 클래스로 분리

08-d

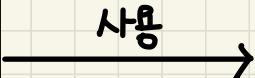


- UI 처리
- 데이터 처리

역할을
쪼갠다

- High Cohesion
 - ↳ 교체가 쉽다
 - ↳ 유지보수가 쉽다

09-c



- 데이터 처리

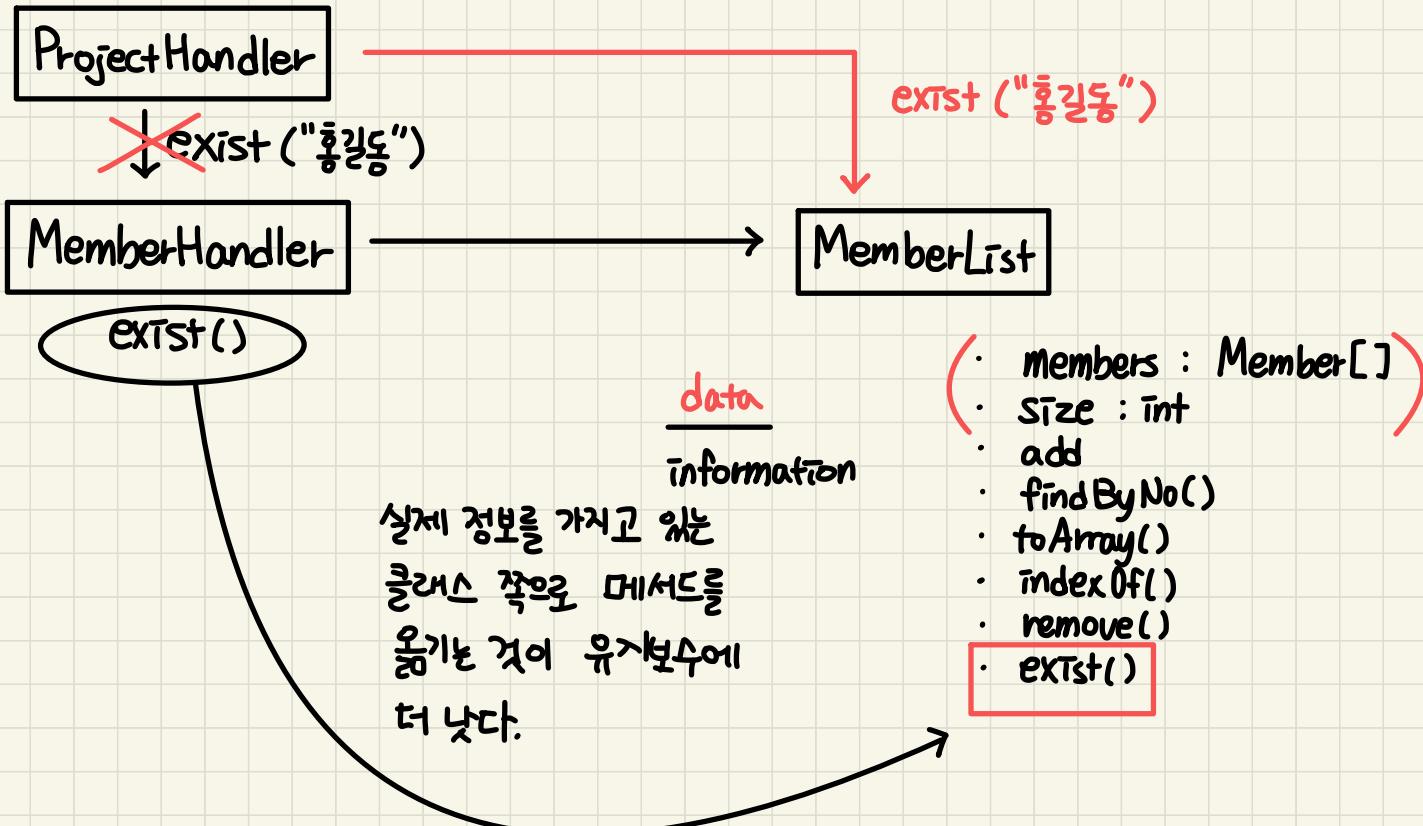
[GRASP]

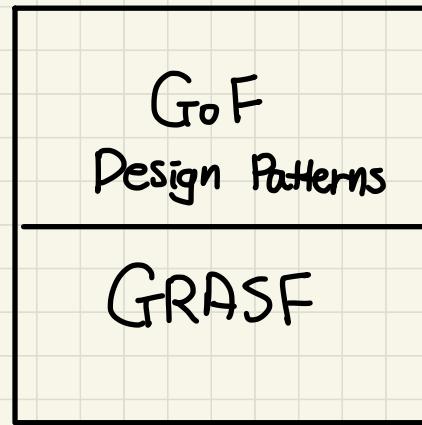
- Information Expert
- Low Coupling

controller

- creator
- :
- :

* GRASP: Information Expert





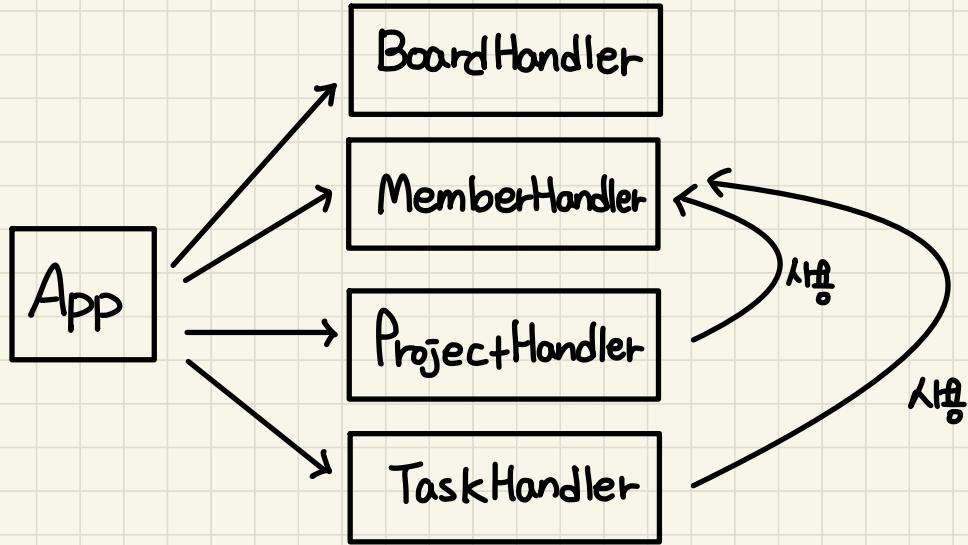
OOP 응용

설계 패턴

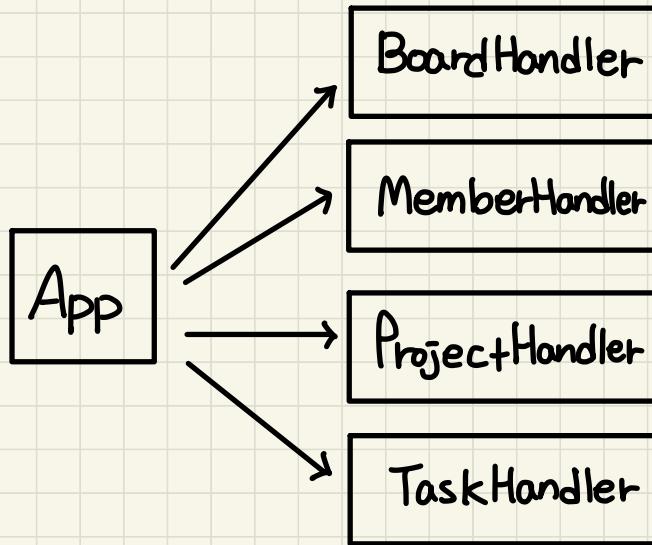
OOP 기초

설계 패턴

08-d



09-C



09-d. 데이터 처리 코드 캡슐화 + 배열크기증가



· UI 처리

· 데이터 처리

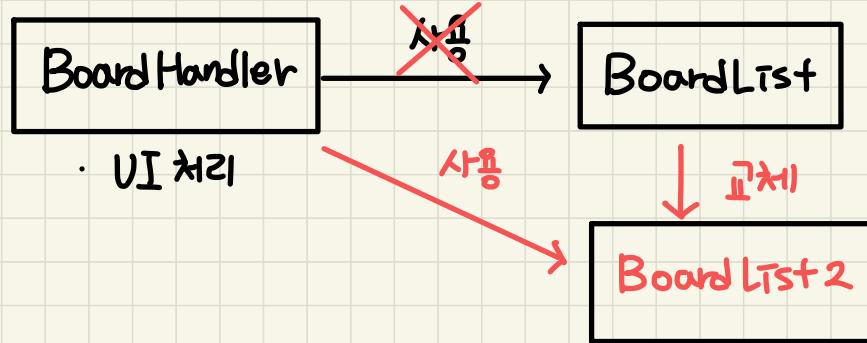
클래스의 역할을
더 전문화하여
잘게 쪼개면
유지보수가 더 쉽다.

↑
배열 크기를 자동으로 늘리기 위해
코드를 변경하더라도

BoardHandler에게
영향을 끼치지 않는다.

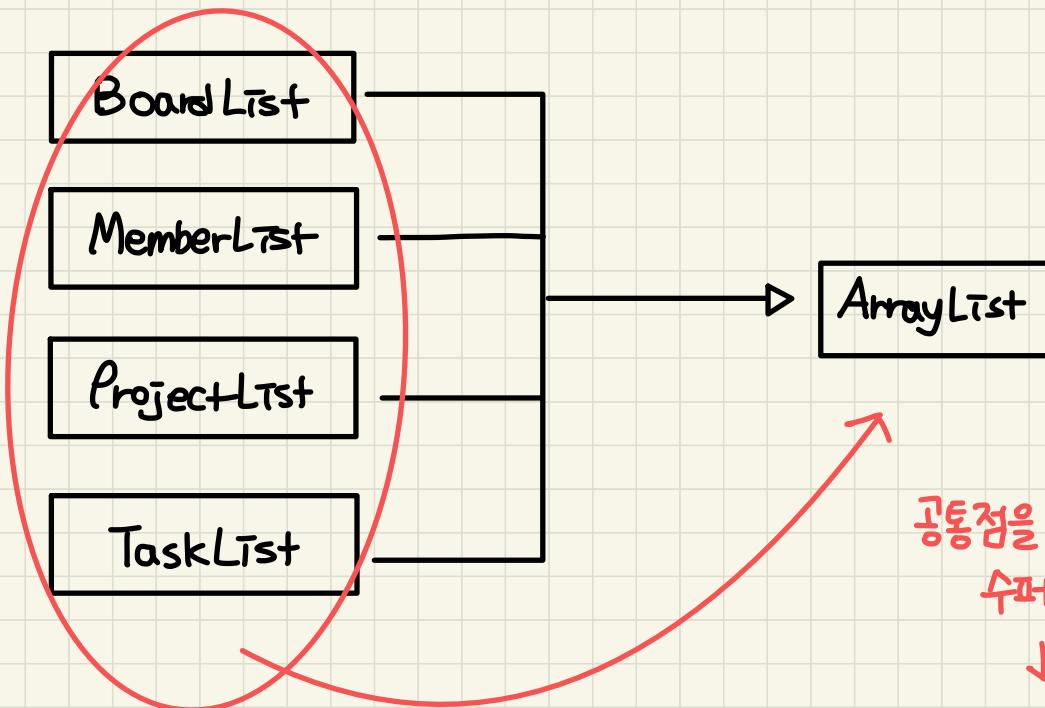
⇒ . 코드 재사용성이 더 높아진다.
. 교체가 가능해진다.

09-e. 데이터 처리 코드 캡슐화 + 연결 리스트 적용



* 데이터 처리 코드를
이렇게 별도의 클래스로 분리하면
해당 기능을 다른 클래스로 교체하기가 쉬워진다.

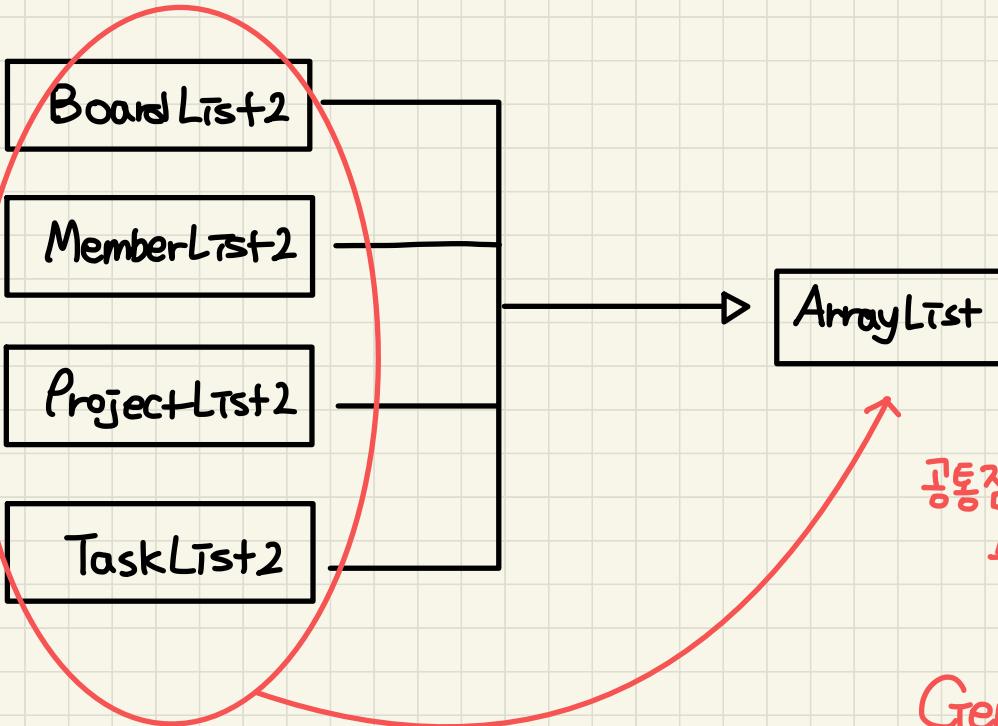
10-a. 상속 : generalization



공통점을 추출해서
슈퍼클래스로 정의

↓
Generalization

10-a. 상속 : generalization II



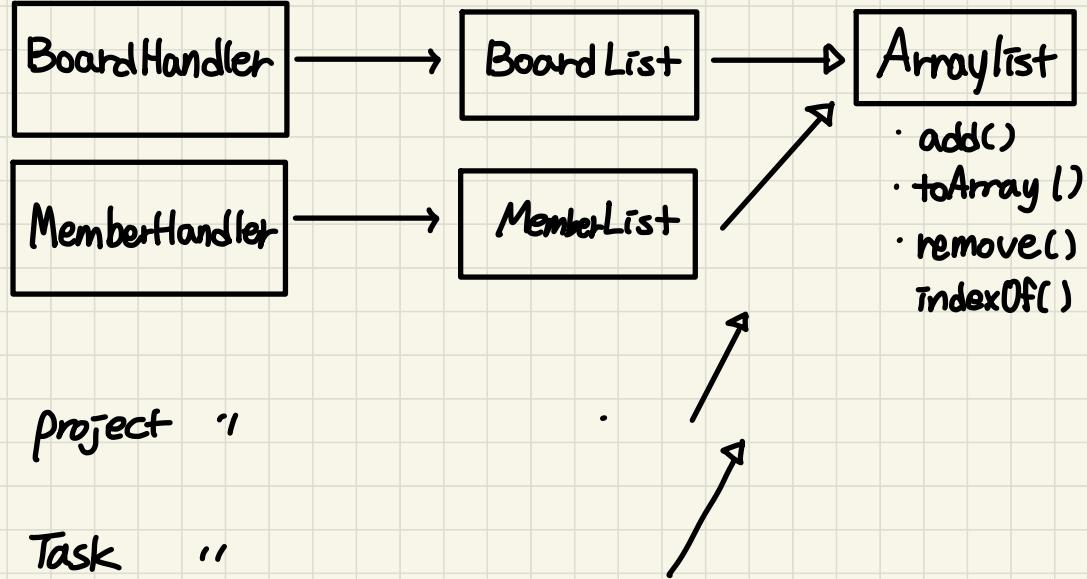
공통점을 추출해서
수퍼 클래스로 정의



Generalization

* 10-C. ArrayList, LinkedList 직접 사용

기존 방식



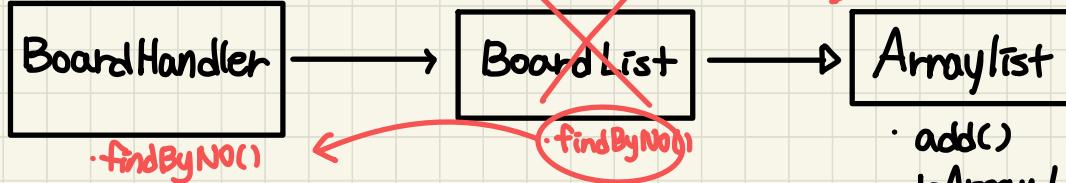
* 10-C. ArrayList 직접 사용

변경



구조를 바꾸는 이유?

- 유지보수하기 더 쉬운 구조!

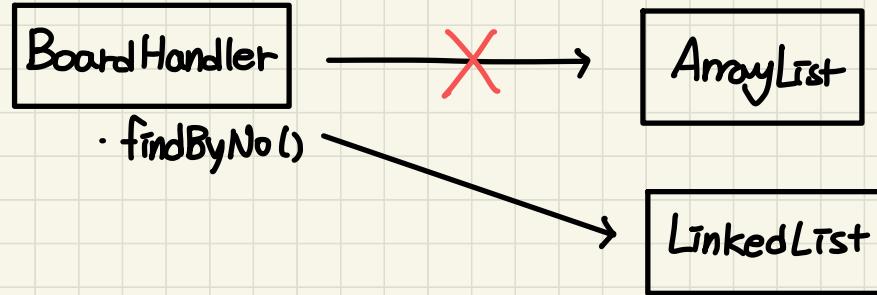


- add()
- toArray()
- remove()
- indexOf()

ArrayList를 직접 사용?!



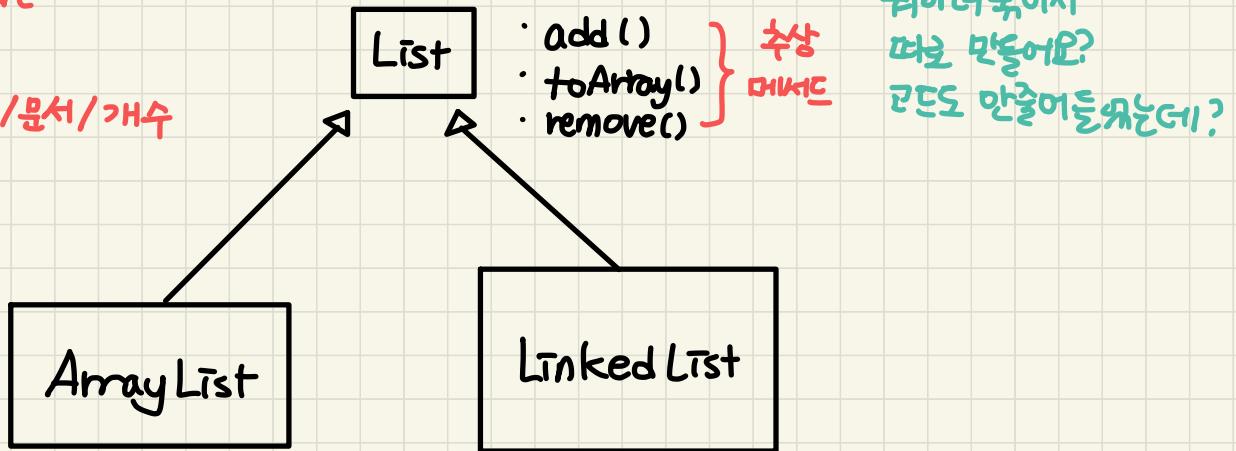
* 10-d. LinkedList 사용



* 10-e . ArrayList와 LinkedList에 대해 Generalization 수행

* Method Signature

- 메서드명
- 파라미터 타입 / 문서 / 개수
- 리턴타입



뭐하러 묶어서
펴보 만들어요?
고드도 만들어둘래는데?

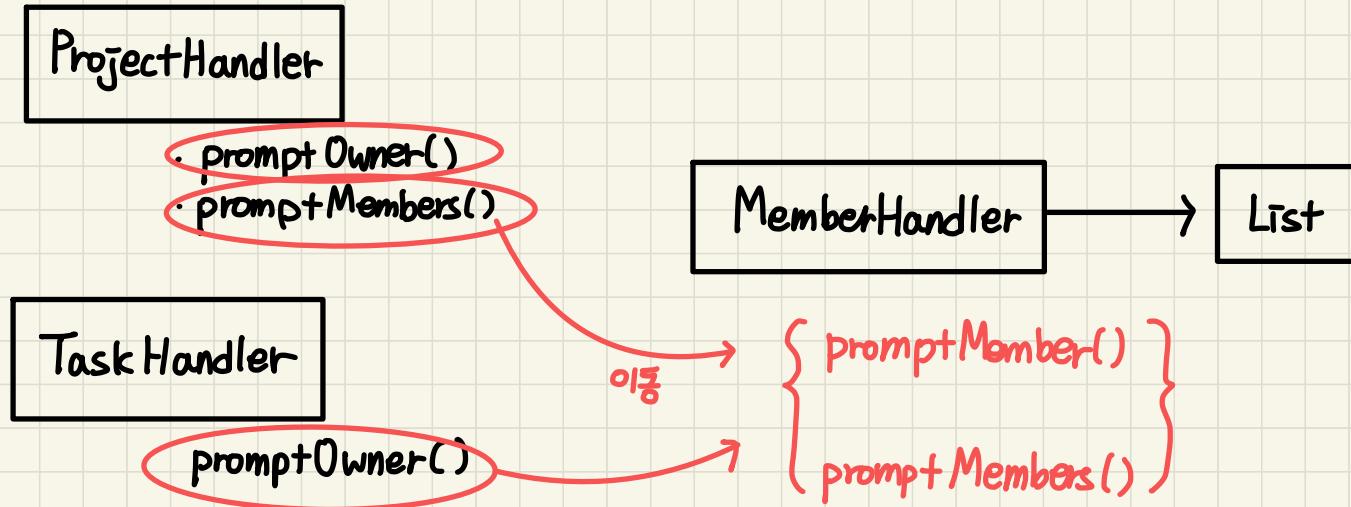
* 서브 클래스 입장에서는 추상메서드만 갖고 있는

List 와 클래스를 상속받는 것은 아무런 도움이 되지 않는다.

① 같은 수퍼클래스를 상속받음으로써 같은 타입으로 묶을 수 있다.

② 비록 구현한 것은 없지만 추상메서드를 상속 받게함으로써 메서드 시그니처를 동일할 수 있다. → 메서드 호출에 일관성 부여 ⇒ 클래스 사용법이 같다.
↳ List 라는 하나의 타입으로 다룰 수 있어 편하다. 즉 다형성을 활용할 수 있다.

* 10-f. 메서드 이동



* 10 - g. 인터페이스 활용

* List 클래스는
ArrayList와
LinkedList를
한 타입으로 묶기 위해
만든 클래스다.

- 상속해줄 필드는
없다.
- 다만 구현해야 할
메서드만 제공.



- add()
 - toArray()
 - remove()
- } 추상메서드



추상클래스는
인스턴스 생성이
불가능하므로!
상속받는 어떤 것인은
그 자리에 대신 주입 가능

이렇게 뭔가를 상속해주는 역할은 하지 않고 \Rightarrow 즉 같은 대입으로 묶으면서
구현 규칙을 정의하는 경우
해야 할 일만 강요하는 경우

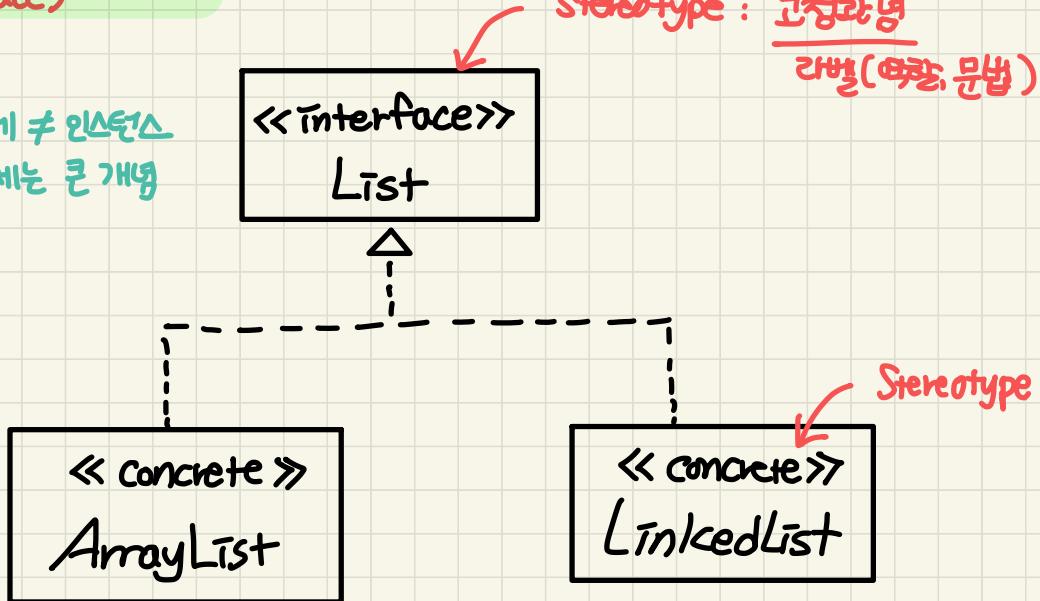
메서드

“인터페이스” 문법을 사용하라!
“개체 규칙을 정의하는 전문문법인”

* 인터페이스 (interface) 대체

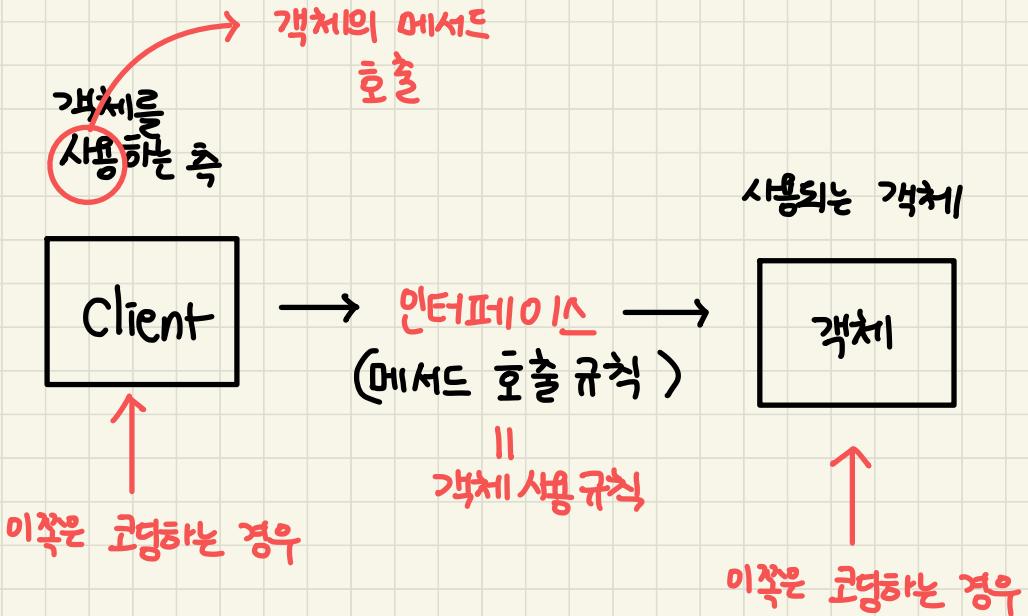
↳ 객체 사용 규칙
" Class

객체 ≠ 인스턴스
객체는 큰 개념



Stereotype : 고정관념
리밸(여전한 문법)

new Date ()
↓
Heap
인스턴스
기록



* 인터페이스와 구현체

