

# 클린 코드

오정임(purum01@naver.com)

# 목차

- section01 클린 코드 개요
- section02 객체지향 프로그램
- section03 클래스와 상속
- section04 소프트웨어의 변화와 유지보수성
- section05 코드의 수정
- section06 소프트웨어 설계 원칙
- section07 객체지향 설계 원칙

# section01 클린 코드 개요

클린 코드 정의

명명 규칙

괄호 및 주석

# 클린 코드 개요

- 프로그래밍은 요구사항을 명확히 표현하는 작업이며, 자동화 기술이 발전해도 그 중요성은 사라지지 않는다. 따라서 **코드 품질을 지속적으로 유지하는 것은 모든 개발자의 기본 자질**이다.

# 나쁜 코드(Bad Code)란?

- **코드 품질:** "코드 품질은 분당 외치는 WTF 횟수로 측정된다."
- **나쁜 코드의 원인:** 급함, 시간 부족, 후속 수정 미루기 등
- **방치 결과:** 버그 발생, 시스템 성능 저하
- **유사 이론:** '깨진 유리창 이론'처럼 나쁜 코드는 더 많은 나쁜 코드를 만든다

# 클린 코드(Clean Code)란?

- 클린 코드란?
  - 가독성이 높은 코드로, 나쁜 코드의 반대 개념
- 나쁜 코드 개선
  - 나쁜 코드를 식별하고 개선하는 기법을 익혀야 함
- 지속적인 연습과 자세
  - 클린 코드를 위해선 꾸준한 연습과 '보이스카우트 규칙' 같은 자기관리 필요
- 가독성이 최우선 기준
  - 코드가 쉽게 읽히고 이해될 수 있어야 클린 코드라 할 수 있음
- 관리자의 압박보다 프로그래머의 책임감
  - 외부 압력보다도 개발자의 코드 책임 의식이 중요

# 비야네 스트롭의 클린 코드 철학

- 간결한 논리 : 단순한 구조는 버그를 피할 수 있게 해준다.
- 낮은 의존성 : 의존성을 줄이면 유지보수가 쉬워진다.
- 명확한 오류 처리 : 오류는 전략적으로, 철저하게 다뤄야 한다.
- 성능의 균형 : 성능의 최적 유지가 무분별한 최적화를 막는다.
- 핵심 메시지 : “깨끗한 코드는 한 가지를 제대로 한다.”

# 클린 코드의 3대 원칙(*론 제프리*)

## 1.중복 줄이기

- 같은 작업이 반복되면 아이디어를 제대로 표현하지 못하고 있다는 신호이다.

## 2.표현력 높이기

- 코드는 시스템의 설계 아이디어를 명확하게 드러내야 한다.

## 3.초기부터 간단한 추상화 고려하기

- 클래스, 메소드의 수를 최소화하여 간단하게 추상화한다.



# 명명 규칙(Naming Rule)

- 좋은 이름은 코드의 이해와 역할 파악을 쉽게 도와준다.
  - ✓ 변수, 함수, 클래스, 패키지 등 모든 구성 요소에서 **이름 짓기**는 가독성과 유지보수성의 핵심 요소입니다.

## 💡 개선 전

java

```
int td; // 하루에 경과한 시간 (단위: 시간)
```

## ✅ 개선 후

java

```
int elapsedTimeInHours;
```

또는 하루 기준이라면:

java

```
int elapsedTimeInDays;
```

```
public List<int[]> getThem() {  
    List<int[]> list = new ArrayList<int[]>();  
    for (int[] x : theList)  
        if(x[0] == 4)  
            list.add(x);  
    return list;  
}
```

- getThem() 이라는 메소드는 어떤 기능의 메소드인가?
- theList라는 컬렉션에 어떤 데이터들이 들어있는가?
- theList에서 0번 인덱스 값은 왜 특별한가?
- 값 4가 갖는 의미는 무엇인가?
- 메소드가 반환하는 list라는 컬렉션에는 어떤 데이터들이 저장되어 있는가?

```

public List<int[]> getFlaggedCells() {
    List<int[]> flaggedCells = new ArrayList<int[]>();

    for (int[] cell : gameBoard)
        if(cell[STATUS_VALUE] == FLAGGED)
            flaggedCells.add(cell);

    return flaggedCells;
}

```

✅ 개선된 점 (명명 규칙 준수)

요소	설명
getFlaggedCells()	함수 이름만 보고도 <b>""</b> 플래그된 셀을 가져오는 함수 <b>""</b> 임을 알 수 있다
flaggedCells	리스트의 내용이 무엇인지 정확하게 표현하고 있음
cell	루프 변수로서 각 배열이 셀 하나를 의미함을 드러냄
STATUS_VALUE, FLAGGED	의미 있는 상수 이름으로 매직 넘버 제거

# 비슷한 이름, 다른 기능 -메소드 네이밍의 함정

- getActiveAccount(), getActiveAccounts(), getActiveAccountInfo()는 이름이 비슷해 용도 구분이 어렵다.
- 메소드 사용자(클래스 사용자)는 각 메소드의 정확한 기능을 파악하기 어려움.
- 메소드 이름은 **의도를 명확히** 드러내고, **서로 구별 가능해야** 사용성과 유지보수성이 높아진다.

## 개선 방법

### 1. 의도가 동일하다면 하나로 통합

java

```
AccountInfo getActiveAccountInfo(); // 단일 메소드로 통합
```

### 2. 기능이 다르다면 구분 가능한 이름 부여

java

```
Account getPrimaryActiveAccount();  
List<Account> getAllActiveAccounts();  
AccountInfo getActiveAccountDetails();
```

# 변수명은 가독성이 핵심이다

- 로컬 변수에 대충 짓는 경향이 있다.
- 예: status → st, stt, 또는 temp, a, b처럼 의미 없는 이름 사용
- 결과적으로 코드의 가독성 저하 및 이해 시간 증가를 초래

# 변수명, 길이와 축약어에 주의하라

- 변수명 내의 축약어는 모두 대문자로 표현한다.
  - 의미 없는 3글자 이하 변수명은 피한다.
    - 단, 반복문의 i, j 같은 임시 변수는 예외
  - 너무 긴 변수명도 피한다.
    - 가독성과 간결함 사이의 균형이 중요
- 잘못된 변수명 예

변수명	설명
String Value;	대문자로 시작
String VALUE	모두 대문자로 구성
String productname;	새로운 단어의 첫 글자를 소문자로 작성
String n;	의미가 불분명한 한 글자의 변수명
String numValue;	의미가 부정확한 변수명
boolean status\$;	달러(\$) 기호 사용
String product_name;	밑줄(_) 기호 사용

# 상수는 대문자와 신뢰로 작성하라

- 상수는 소프트웨어 전반에서 사용되며, 변경 가능성이 거의 없는 중요한 값을 저장한다.
- 이름 규칙
  - 일반 변수와 동일한 규칙
  - 모두 대문자 + 밑줄(\_) 사용

상수명	설명
<code>static final int maxValue = 100000;</code>	소문자 사용
<code>static final int MINVALUE = 1;</code>	잘못된 단어 간 구분

# 메소드명 규칙

- 메소드명에는 동사 또는 동사와 명사의 조합을 사용한다.
- 명사를 사용할 때 너무 긴 경우 축약해서 사용할 수 있지만 의미가 불분명한 너무 짧은 이름은 자제해야한다.
- 메소드명 내의 축약어는 모두 대문자로 표현한다.
- 의미가 불분명한 세 글자 이하의 메소드명은 쓰지 않는다.

메소드명	설명
public void ParseInt() { }	대문자로 시작
public void parseInt() { }	새로운 단어의 첫 글자를 소문자로 시작
public void parse_int() { }	밑줄(_) 기호 사용
public void parse\$int() { }	달러(\$) 기호 사용
public void pi() { }	지나친 약어 사용



# 패키지명 규칙

- 패키지명은 소문자로 구성한다.
- 패키지명은 패키지의 기능을 정확히 전달할 수 있는 한 단어의 명사로 한다.
- 패키지의 최상위 부분은 소문자의 도메인 이름이어야 한다.
- 하위 컴포넌트명은 내부 명명 규칙을 따를 수 있다.
- 달러 기호(\$)를 패키지 명으로 사용하지 않는다.

패키지 명	설명
<code>package TEST;</code>	대문자로 구성된 패키지명 사용
<code>package wikibook.co.kr;</code>	순서가 잘못된 패키지 도메인명 사용
<code>package kr.co.wikiBook;</code>	패키지명에 대문자를 사용
<code>package kr.co.testUnit;;</code>	한 단어 이상의 패키지명 사용

# 클래스명과 인터페이스명 규칙

- 클래스명은 파스칼 표기법을 바탕으로 명명해야한다.
- 클래스명에는 명사만 사용할 수 있다.
- 명사를 사용할 때 너무 긴 경우 축약해서 사용할 수 있지만 의미가 불분명한 너무 짧은 이름은 자제한다.
- 클래스명 내의 축약단어는 모두 대문자로 표현한다.

잘못된 클래스명	수정된 클래스명
public class content { }	public class Content { }
public class CONTENT { }	public class Content { }
public class Stringbuilder { }	public class StringBuilder { }
public class HtmlUtil { }	public class HTMLUtil { }
public class Doc { }	public class Document { }
public class Order_Util { }	public class OrderUtil { }
public class \$ServiceBuiler { }	public class ServiceBuiler { }

# 괄호 규칙

- 코딩 스타일의 중요성
  - 들여쓰기와 블록 처리는 소스코드에 큰 영향을 미친다.
- Allman 스타일
  - Eric Allman이 사용한 스타일로, C 언어에서 주로 사용된다.
- K&R 스타일
  - "The C Programming Language" 책에 소개된 스타일로, 자바 코딩에서 주로 사용되며 많은 IDE에서 기본적으로 지원한다.
- 1TBS 스타일
  - K&R 스타일을 기반으로 제어문 코드가 한 줄이어도 괄호로 묶어 구역을 명확히 구분하며, 자바와 자바스크립트에서 사용된다.
- 초기 자바 스타일
  - 썬에서는 초기 자바의 괄호 규칙으로 K&R 스타일을 선택했으나, 현재는 많은 자바 API 소스가 1TBS 스타일을 따른다.

# 괄호 규칙

올맨 스타일

```
public void test() {  
    for (int num = 0; num < 10; num++)  
    {  
        sum = sum + num;  
        if (sum > 10)  
        {  
            //...  
        }  
        else  
        {  
            //...  
        }  
    }  
}
```

K&R 스타일

```
public void test() {  
    for (int num = 0; num < 10; num++) {  
        sum = sum + num;  
        if (sum > 10) {  
            // ....  
        } else {  
            // ...  
        }  
    }  
}
```

# 괄호 규칙

- **K&R 스타일:** 중괄호를 생략할 수 있는 경우에는 생략할 수 있다

```
if (sum > 10)
    // ...
else
    // ...
```

- **1TBS 스타일:** 모든 제어문에 중괄호를 사용하는 것을 권장한다

```
if (sum > 10) {
    // ...
} else {
    // ...
}
```

# 주석의 함정: 설명보다 이해하기 쉬운 코드

- 주석의 필요성
  - 코드만으로 의미를 전달할 수 없을 때 사용하며, 이해하기 쉬운 코드를 작성하면 주석이 불필요하다.
- 나쁜 코드의 방증
  - 주석은 복잡한 코드를 설명하는 대신, 코드를 간결하게 만들어 주석을 줄여야 한다.
- 코드로 의도 표현
  - 주석 대신 함수로 설명을 처리하고, 코드 자체로 의도를 표현하려고 노력해야 한다.
- 주석 유지보수 문제
  - 시간이 지나면서 주석은 코드와 불일치하게 되어 혼란을 초래할 수 있다.
- 부정확한 주석의 위험
  - 부정확한 주석은 오히려 해가 되므로, 주석을 제거하고 코드로 설명을 대체해야 한다.

# 효율적인 주석 작성 가이드

## 좋은 주석

- 법적인 주석
- 정보를 제공하는 주석
- 의도를 설명하는 주석
- 의미를 명료하게 밝히는 주석
- 결과를 경고하는 주석
- TODO 주석
- 중요성을 강조하는 주석

## 나쁜 주석

- 주절거리는 주석
- 같은 이야기를 중복하는 주석
- 의무적으로 다는 주석
- 이력을 기록하는 주석
- 있으나 마나 한 주석
- 위치를 표시하는 주석
- 닫는 괄호에 다는 주석
- 공로를 돌리거나 저자를 표시하는 주석
- 주석으로 처리된 코드

# section02 객체지향 프로그램

절차 지향 프로그램 VS 객체 지향 프로그램 특징

객체지향 기본 원리(캡슐화와 정보은닉, 추상화, 상속과 다형성)

객체지향스러운 자바 코드



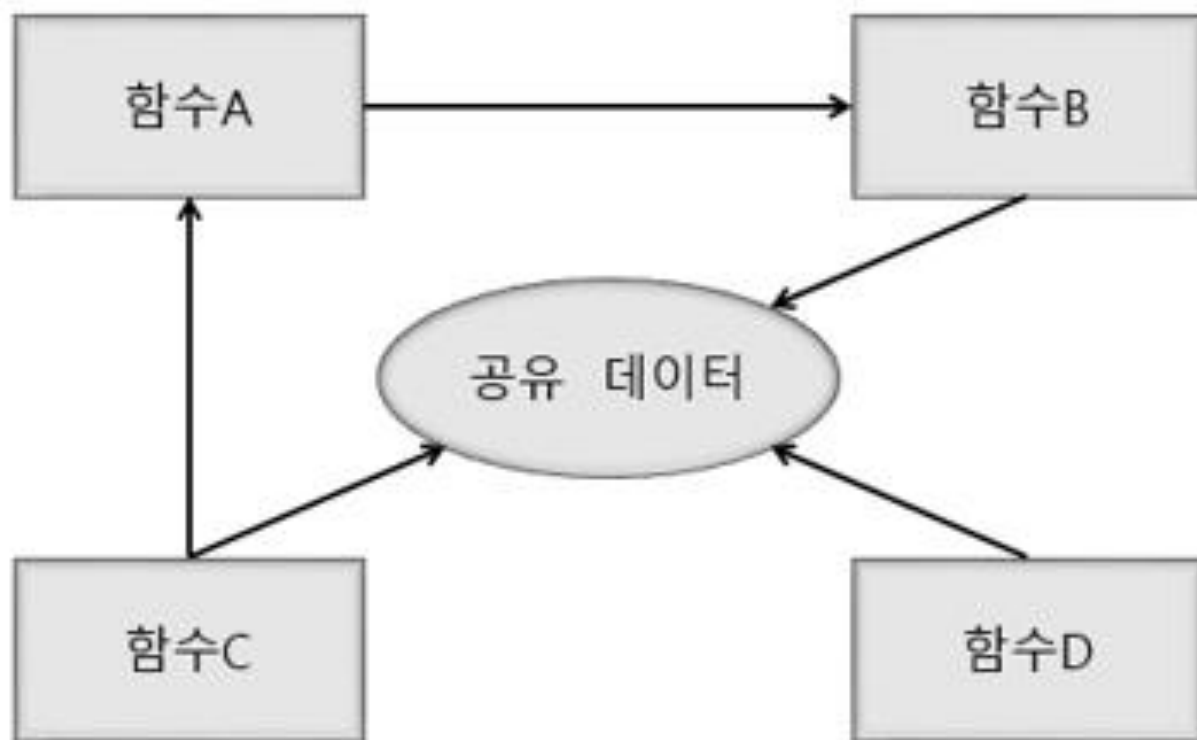
# 객체지향으로 다듬는 클린 코드

- 목표: 코드를 쉽게 읽고 고치기 좋게 만드는 것
- 문제: 지저분한 코드는 시간 낭비의 원인
- 해결책: 객체지향 언어는 읽기 쉽고 유지보수가 편함
- 주의점: 객체지향스럽게 코딩하려면 훈련이 필요
- 장점: 기능을 쉽게 추가하거나 변경할 수 있음

# 절차지향 프로그램

- 절차지향 언어의 구조
  - 데이터 중심이며, 데이터를 처리하는 여러 함수로 구성된다.
- 함수 호출 방식
  - 필요한 데이터를 인자로 지속적으로 함수에 공급해야 한다.
- 전역 변수 사용
  - 함수 실행 후 상태 정보를 저장하기 위해 전역 변수를 사용한다.
- 전역 변수의 문제점
  - 전역 변수를 여러 함수가 공유하면 의도하지 않은 결과가 나올 수 있다.

# 절차지향 프로그램



```
public class Stack {  
    int size;  
    Object[] items;  
    int top = -1;  
    public Stack() {  
        size = 10;  
        items = new Object[size];  
    }  
    public Stack(int size) {  
        this.size = size;  
        items = new Object[size];  
    }  
    public void setItem(int index, Object item) {  
        items[index] = item;  
    }  
    public Object getItem(int index) {  
        return items[index];  
    }  
}
```

```
class StackFunction {  
    public boolean isEmpty(Stack stack) {  
        return (stack.top < 0);  
    }  
  
    public boolean isFull(Stack stack) {  
        return (stack.top == stack.size);  
    }  
  
    public boolean push(Stack stack, Object newItem) {  
        if (isFull(stack))  
            return false;  
        else {  
            int newTop = stack.top + 1;  
            stack.top = newTop;  
            stack.setItem(newTop, newItem);  
            return true;  
        }  
    }  
}
```

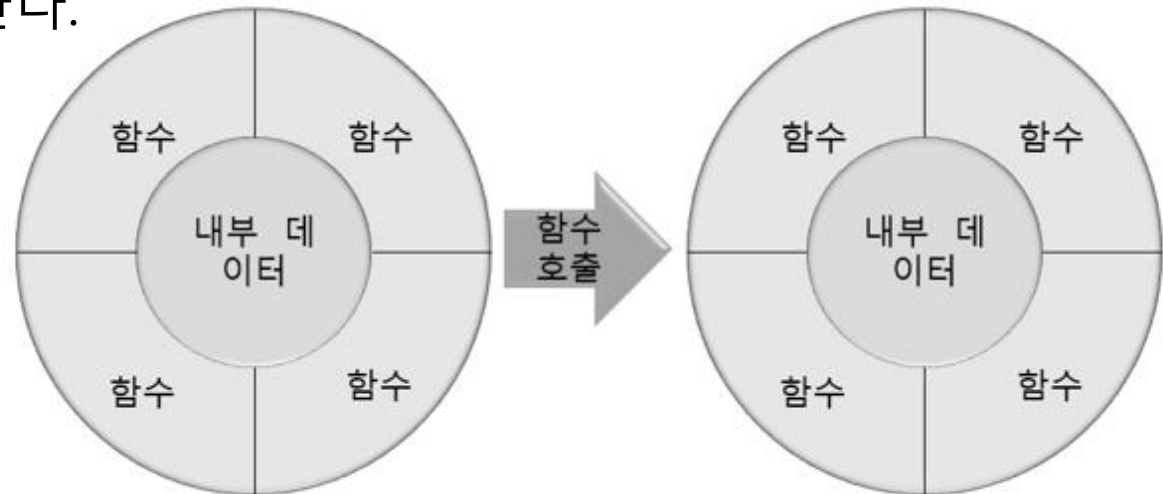
```
public Object pop(Stack stack) {  
    if (isEmpty(stack))  
        return null;  
    else {  
        int currentTop = stack.top;  
        Object item = stack.getItem(currentTop);  
        stack.top = currentTop - 1;  
        return item;  
    }  
}  
  
public Object top(Stack stack) {  
    if (isEmpty(stack))  
        return null;  
    else  
        return stack.getItem(stack.top);  
}
```

# 객체지향 프로그램

- 객체지향 프로그래밍의 구조
  - 데이터와 함수가 객체 안에 함께 캡슐화된다.
- 속성과 메소드
  - 데이터는 속성이 되고, 이를 다루는 함수는 메소드가 된다.
- 데이터 캡슐화
  - 속성은 외부에 직접 노출되지 않으며, 필요한 데이터는 메소드 매개변수로 받아들인다.

# 객체지향 프로그램

- 객체 간 상호작용
  - 객체는 서로의 메소드를 호출하며 기능을 수행하고, 서로 다른 책임을 나누어 맡는다.
- 상태 유지
  - 객체는 내부적으로 상태 정보를 유지해 매번 데이터를 공급하지 않아도 되며, 연관된 데이터와 함수를 클래스로 정의한다.



```
public class Stack {  
    private int size;  
    private Object[] items;  
    private int top = -1;  
  
    public Stack() {  
        size = 10;  
        items = new Object[size];  
    }  
    public Stack(int size) {  
        this.size = size;  
        items = new Object[size];  
    }  
    public boolean isEmpty() {  
        return (top < 0);  
    }  
    public boolean isFull() {  
        return (top >= size);  
    }  
}
```

```
    public boolean push(Object newItem) {  
        if (isFull())  
            return false;  
        else {  
            top++;  
            items[top] = newItem;  
            return true;  
        }  
    }  
    public Object pop() {  
        if (isEmpty())  
            return null;  
        else  
            return items[top--];  
    }  
    public Object top() {  
        if (isEmpty())  
            return null;  
        else  
            return items[top];  
    }  
}
```



```
public boolean push(Object newItem) {  
    if (isFull())  
        return false;  
    else {  
        top++;  
        items[top] = newItem;  
        return true;  
    }  
}  
  
public Object pop() {  
    if (isEmpty())  
        return null;  
    else  
        return items[top--];  
}  
  
public Object top() {  
    if (isEmpty())  
        return null;  
    else  
        return items[top];  
}
```

```
Stack stack1 = new Stack();  
Stack stack2 = new Stack();  
stack1.push("1");  
stack2.push("2");
```

# 객체지향 기본 원리 – 캡슐화(Encapsulation)

- 캡슐화는 관련된 데이터(속성)와 해당 데이터를 처리하는 메소드(행위)를 하나의 단위(클래스)로 묶고, 외부에서 직접 접근하지 못하도록 제한(정보 은닉)하여 객체의 내부 구현을 보호하는 객체지향의 핵심 원리이다.
- 단순히 묶는 것(aggregation) 이상으로, 접근 제어와 정보 보호가 핵심이다.
- 유지보수성 향상, 오류 방지, 명확한 인터페이스 설계로 이어진다.

# 객체지향 기본 원리 - 정보 은닉

- 정보 은닉은 객체지향의 핵심 원리 중 하나로, 객체 내부의 구현 세부사항을 외부로부터 숨기고, 필요한 정보나 기능만을 공개하는 것이다. 이를 위해 접근 제어자 `private`을 사용하며, 이는 데이터 보호와 코드의 안정성을 높이는 데 기여한다.

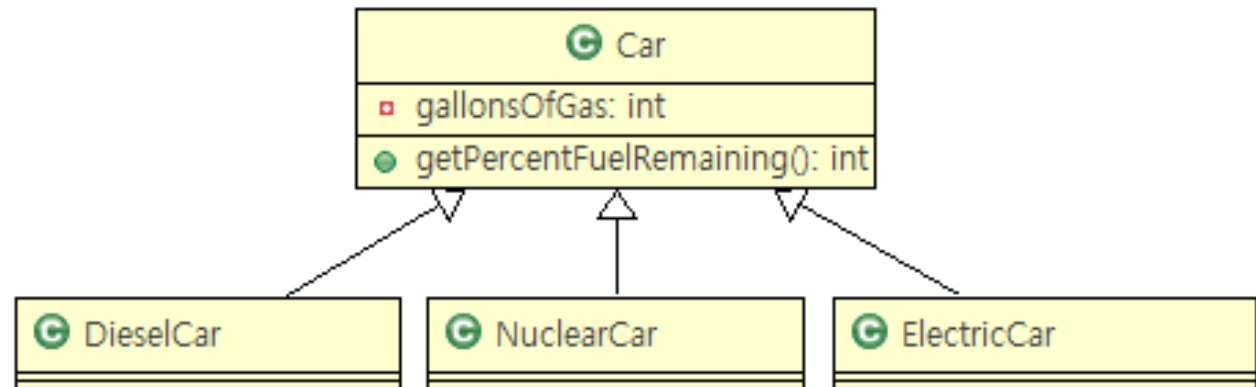
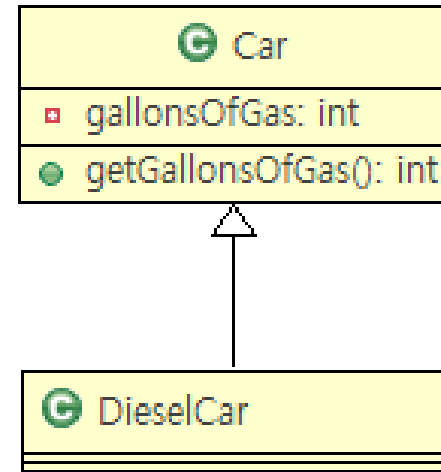
# 객체지향 기본 원리 – 추상화(Abstraction)

- 추상화는 객체의 복잡한 내부 구현을 감추고, 핵심적인 기능이나 공통적인 특성만을 외부에 제공하는 객체지향의 핵심 원리이다. 자바에서는 abstract class, abstract method, 그리고 interface 등을 통해 추상화를 구현할 수 있다.

요소	추상화에 사용 가능	설명
abstract class	✓	일부 구현이 가능, 공통 속성과 동작 정의
abstract method	✓	하위 클래스에서 반드시 구현해야 함
interface	✓	완전한 추상화 – 구현 없이 구조만 정의

# 추상화(Abstraction)

- Car 클래스의 getGallonsOfGas() 메소드는 내부 변수 이름을 드러내어 캡슐화와 추상화 원칙을 위반한다.
- Car를 상속받는 DieselCar처럼 연료 방식이 다른 클래스에도 부적절한 메소드가 상속되는 문제가 발생한다.
- getPercentFuelRemaining()처럼 구체적인 구현을 감춘 추상적인 메소드 이름이 더 적절하며, 메소드에서도 추상화를 적용하는 것이 중요하다.



# 상속(inheritance)과 다형성(polymorphism)

## ▶ 상속(inheritance)의 기본 의미

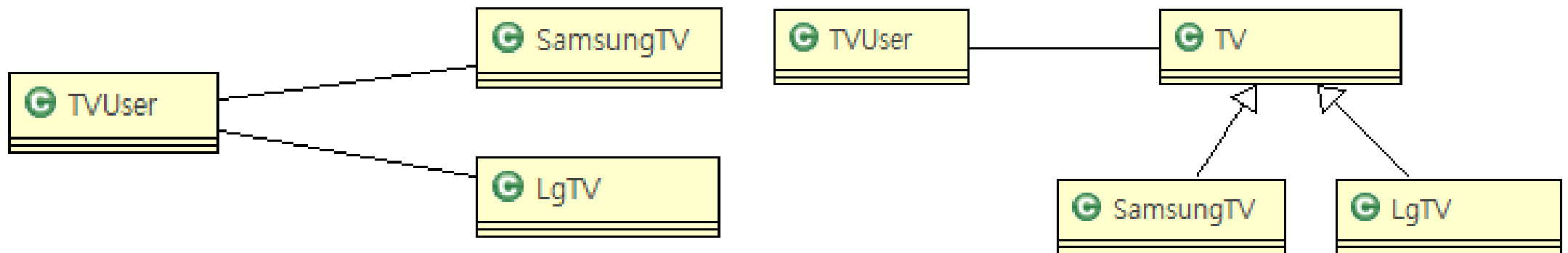
- 하위 클래스가 부모 클래스의 속성과 메소드를 다운 클래스에서 재사용 가능
- 복잡한 코드의 반복을 줄이고, 개발 효율성과 코드의 이해도를 높이며, 유지보수와 확장성을 향상시킬 수 있다.

## ▶ 다형성(polymorphism)

- 부모 클래스 타입으로 다양한 하위 클래스 객체를 다룰 수 있는 것
- 부모 클래스에서 선정된 메소드를 호출하지만, 실제적으로는 하위 클래스의 구현이 실행되는 방식을 가능하게 한다.

# 상속(inheritance)과 다형성(polymorphism)

- 다형성은 같은 타입(예: 리모컨)으로 다양한 객체(SamsungTV, LgTV 등)를 제어할 수 있게 해주는 객체지향의 핵심 개념이다
- .TV가 바뀌더라도 TVUser와 같은 클라이언트 코드를 수정하지 않고 재사용 가능하므로, 코드의 유지보수성과 확장성을 크게 높일 수 있다.



```
abstract class TV {  
    public abstract void powerOn();  
    public abstract void powerOff();  
    public abstract void volumeUp();  
    public abstract void volumeDown();  
}
```

```
class SamsungTV extends TV {  
    public void powerOn() {  
        System.out.println("SamsungTV---전원 켜다.");  
    }  
    public void powerOff() {  
        System.out.println("SamsungTV---전원 끈다.");  
    }  
    public void volumeUp() {  
        System.out.println("SamsungTV---소리 올린다.");  
    }  
    public void volumeDown() {  
        System.out.println("SamsungTV---소리 내린다.");  
    }  
}
```



```
class LgTV extends TV {  
    public void powerOn() {  
        System.out.println("LgTV---전원 켜다.");  
    }  
    public void powerOff() {  
        System.out.println("LgTV---전원 끈다.");  
    }  
    public void volumeUp() {  
        System.out.println("LgTV---소리 올린다."); }  
    }  
    public void volumeDown() {  
        System.out.println("LgTV---소리 내린다.");  
    }  
}
```

```
public class TVUser {  
    public static void main(String[] args) {  
        TV tv = new SamsungTV();  
        tv.powerOn();  
        tv.volumeUp();  
        tv.volumeDown();  
        tv.powerOff();  
    }  
}
```

```
class BeanFactory {  
    public TV getBean(String name) {  
        if(name.equals("lg")) {  
            return new LgTV();  
        } else if(name.equals("samsung")) {  
            return new SamsungTV();  
        }  
        return null;  
    }  
}
```

```
public class TVUser {  
    public static void main(String[] args) {  
        BeanFactory factory = new BeanFactory();  
  
        TV tv = (TV)factory.getBean(args[0]);  
        tv.powerOn();  
        tv.volumeUp();  
        tv.volumeDown();  
        tv.powerOff();  
    }  
}
```

# 객체지향적이지 않은 자바 코드

- getGender()처럼 내부 속성값에 따라 동작을 결정하는 방식은 비객체지향적이다.
- 객체지향에서는 상속을 통해 자식 클래스에서 각각의 행위를 구체화하는 것이 바람직하다.

```
public class Person {  
    private boolean genderFlag;  
    public Person(boolean genderFlag) { this.genderFlag = genderFlag; }  
    public String getGender() { return genderFlag ? "male" : "female"; }  
    public static void main(String[] args) {  
        Person kim = new Person(true);  
        if(kim.getGender().equals("male"))  
            System.out.println("남성입니다. ");  
        else  
            System.out.println("여성입니다. ");  
    }  
}
```

# 객체지향적인 자바 코드

```
abstract class Person {  
    public abstract String getGender();  
}  
  
class Man extends Person {  
    public String getGender() {  
        return "male";  
    }  
}  
  
class Woman extends Person {  
    public String getGender() {  
        return "female";  
    }  
}
```

```
if(kim.getGender().equals("male")) {  
    System.out.println("남성입니다. ");  
} else {  
    System.out.println("여성입니다. ");  
}
```

getGender()는 Person에서 추상 메소드로 선언하고,  
각 자식 클래스가 자신에 맞게 구현함으로써  
내부 속성값에 의존하지 않고,  
객체 스스로 자신의 성별을 책임 있게 말하게 할 수 있다.  
이는 객체지향의 핵심인 책임 중심 설계를 잘 반영한 구조이다.

# 객체지향적인 자바 코드

```
abstract class Person {  
    public abstract boolean isMale();  
}  
  
class Man extends Person {  
    public boolean isMale() {  
        return true;  
    }  
}  
  
class Woman extends Person {  
    public boolean isMale() {  
        return false;  
    }  
}
```

```
if(kim.isMale()) {  
    System.out.println("남성입니다. ");  
} else {  
    System.out.println("여성입니다. ");  
}
```

getGender().equals("male")처럼 값을 비교해 판단하는 방식은, 내부 로직이 바뀌면 클라이언트 코드 전체에 영향을 미친다.

이보다 나은 방법은 성별을 판단하는 책임 자체를 Person 객체에 위임하는 것이다.

즉, "당신은 남자입니까?"라고 객체에게 직접 묻고 객체가 판단하여 응답하도록 하면, 더 객체지향적이고 변경에 유연한 코드가 된다.

# section03 클래스와 상속

클래스와 인터페이스

다형성에서 인터페이스 역할

상속과 합성

# 객체와 인터페이스

- 객체의 핵심
  - 객체지향 언어의 핵심은 클래스가 아닌 객체의 기능이다.
- 서비스 제공
  - 객체는 메소드를 통해 외부에 서비스를 제공하며, 내부 데이터는 중요하지 않다.
- 인터페이스 역할
  - 객체는 인터페이스를 통해 어떤 기능을 제공하는지 명세하며, 이는 메소드의 시그니처로 정의된다.
- 구현 분리
  - 인터페이스는 서비스 명세만 포함하며, 실제 구현은 클래스에 정의된다.
- 사용 원리
  - 객체의 인터페이스만 알면 내부 구현을 몰라도 객체를 사용할 수 있다. 이는 TV나 계산기 같은 전자 제품의 사용 원리와 같다.

# 인터페이스 지정하기

- 자바에서 **\*\*객체의 인터페이스(사용 가능한 기능)\*\***는 객체를 생성한 클래스가 아닌, **\*\*선언된 타입(클래스 or 인터페이스)\*\***에 따라 결정된다.

```
[클래스 이름 or 인터페이스 이름] 객체 이름;
```

```
Employee kim;
```



# 다형성과 다운캐스팅

```
class A {  
    public void methodA1() { System.out.println("A클래스 methodA1()"); }  
    public void methodA2() { System.out.println("A클래스 methodA2()"); }  
    public void methodA3() { System.out.println("A클래스 methodA3()"); }  
}  
  
class B extends A {  
    public void methodA1() {System.out.println("B클래스 methodA1()"); }  
    public void methodB1() {System.out.println("B클래스 methodB1()"); }  
    public void methodB2() {System.out.println("B클래스 methodB1()"); }  
}  
  
public class CastingTest {  
    public static void main(String[] argv) {  
        A a = new A();    a.methodA1();  
        A b = new B();    b.methodA1();  
        ((B) b).methodB1();  
    }  
}
```

✓ 실행 결과:

java

```
A클래스 methodA1()  
B클래스 methodA1()  
B클래스 methodB1()
```

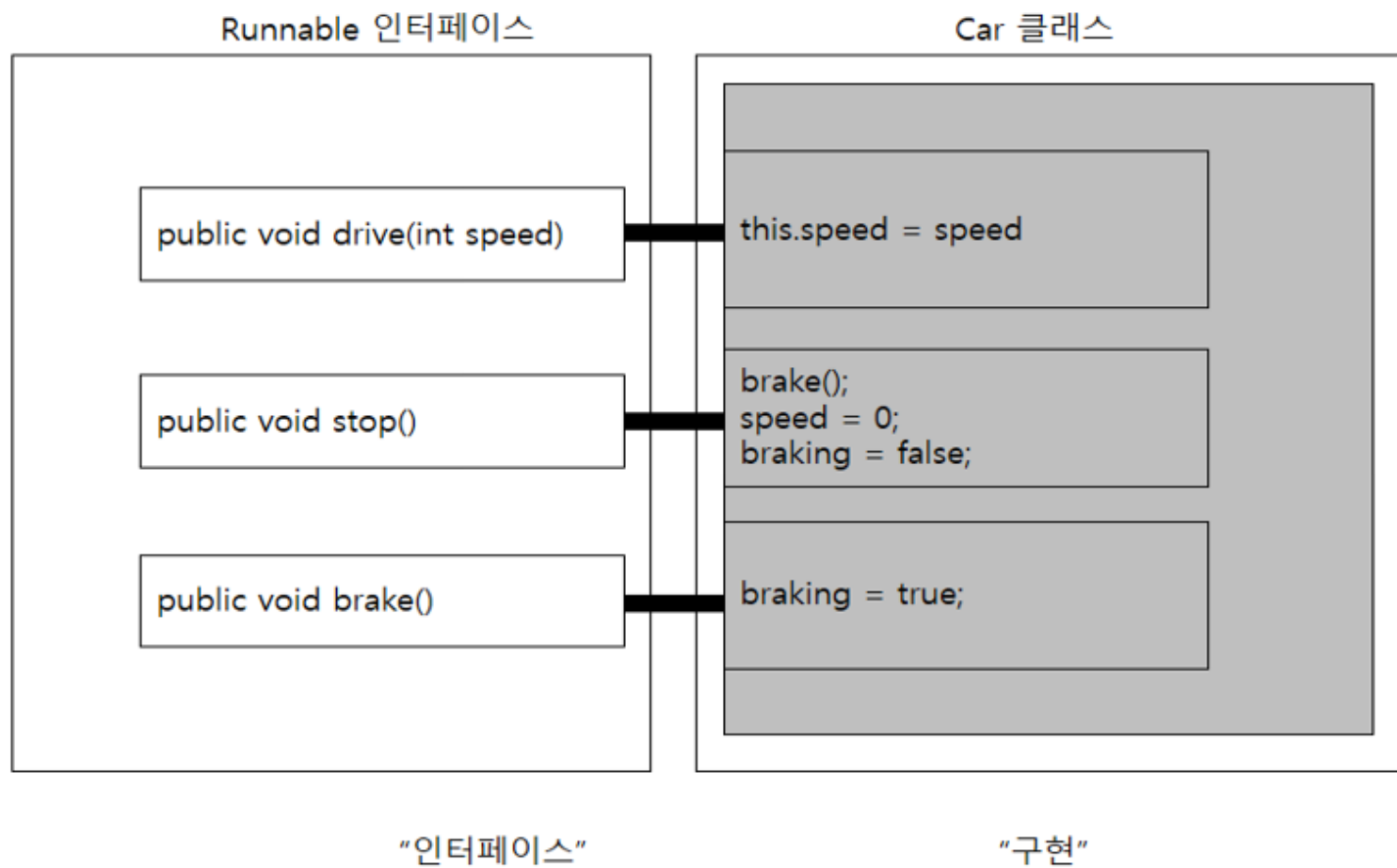
# 인터페이스 사용

```
interface Runnable {  
    public void drive(int speed);  
    public void stop();  
    public void brake();  
}
```

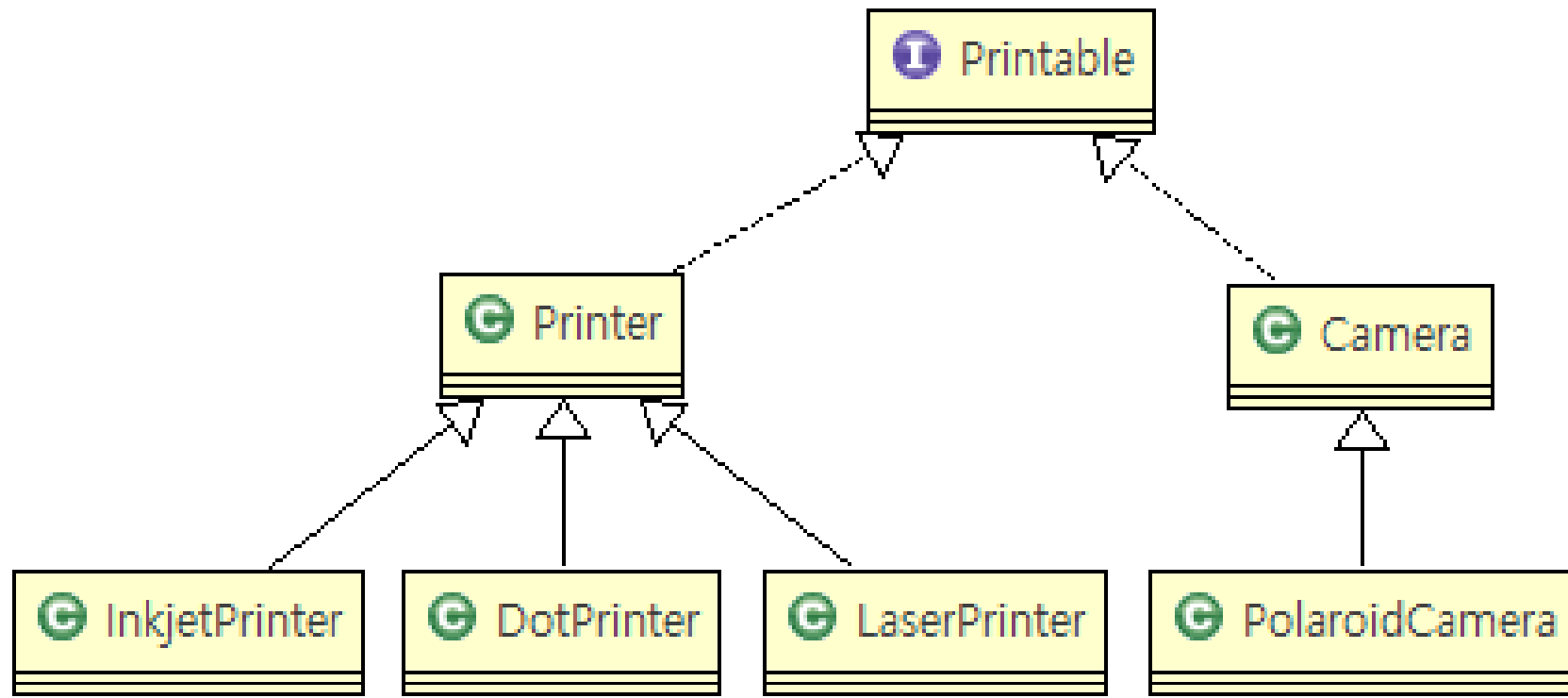
인터페이스를 사용하면 클래스에서 인터페이스와 구현을 분리할 수 있다.

자바의 인터페이스는 메소드의 시그니처만을 선언하며 메소드의 구현 코드는 빠져있다.

```
class Car implements Runnable {  
    private int speed;  
    private boolean braking;  
  
    public void drive(int speed) {  
        this.speed = speed;  
    }  
    public void stop() {  
        brake();  
        speed = 0;  
        braking = false;  
    }  
    public void brake() {  
        braking = true;  
    }  
}
```



자바의 인터페이스는 객체의 인터페이스와 구현을 분리하는 역할을 한다. Runnable 인터페이스를 구현한 클래스(Car)가 어떤 클래스인지 몰라도, 인터페이스만으로 객체의 기능을 사용할 수 있어 구체적인 클래스에 의존하지 않는 유연한 프로그램을 만들 수 있다.



인터페이스는 상속과는 다른 개념으로, 서로 다른 타입의 객체들을 동일한 타입처럼 다루고 싶을 때 사용한다. 예를 들어 **Printer**와 **Camera**는 전혀 다른 클래스지만, 같은 인터페이스를 구현하면 공통된 메시지를 보내는 다형성을 실현할 수 있다. 반면, 클래스 상속은 'IS-A' 관계가 성립될 때 사용하는 것이다.

# 100% 순수 추상 클래스 VS 인터페이스

- 추상 클래스와 인터페이스는 구조는 비슷하지만, 쓰임새와 의미는 다르다.
  - 추상 클래스는 'IS-A' 관계를 전제로 한 공통 기능을 모은 상속 계열을 구성할 때 사용한다.
  - 인터페이스는 서로 다른 계열의 클래스라도 공통된 동작이 필요할 때 사용한다.
- Printer는 모든 프린터의 상위 클래스(패밀리), Printable은 프린터 외에도 인쇄 가능한 다양한 객체를 묶기 위한 행동 기반 타입이다. 따라서 개발자의 의도와 문맥에 따라 적절히 선택해야 한다.

```
abstract class Printer {  
    public abstract void print(String message);  
}
```

```
interface Printable {  
    public void print(String message);  
}
```

# 상속(Inheritance)과 합성(Composition)

- 상속은 기능을 물려받아 확장하는 것이고, 합성은 다른 객체에게 역할을 위임하는 것이다. 상황에 따라 합성을 사용하면 더 유연하고 유지보수하기 좋은 코드를 만들 수 있다.

# 상속(Inheritance)

- 상속의 재사용
  - 부모 클래스의 변수와 메소드를 자식 클래스가 재사용하며, 오버라이딩과 메소드 추가로 기능을 확장할 수 있다.
- 다형성의 구현
  - 상속을 통해 프로그램에서 다형성을 구현할 수 있으며, 이는 소스 재사용 이상의 의미를 가진다.
- 유연한 프로그램 설계
  - 다형성을 사용하면 요구사항을 쉽게 추가, 반영할 수 있는 유연한 프로그램을 설계할 수 있다.
- 동적 바인딩
  - 하위 클래스의 메소드를 오버라이딩하고, 상위 클래스 타입으로 객체를 할당한 후 메소드를 호출하면, 컴파일 시간이 아닌 실행 시점에 메소드가 결정된다. 이를 동적 바인딩이라고 한다.

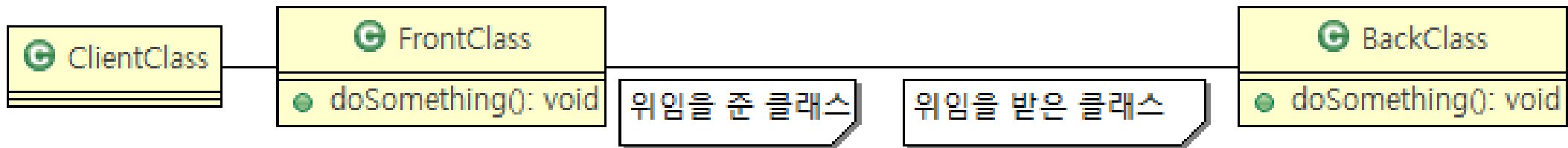
# 상속 관계 규칙

- 상속은 클래스들끼리 영구적으로 'IS-A' 관계가 성립될 때 사용한다.
- 상속이라는 의미가 통하지 않음에도 단순히 기존에 있는 클래스의 속성과 메소드를 재사용하기 위해 상속을 사용하는 것은 바람직한 일이 아니다. 이때는 합성을 사용하는 것이 좋다.
- 다형성을 구현하기 위해 상속을 강제로 사용하는 것 역시 바람직하지 않다. 이때는 합성과 인터페이스를 같이 사용하면 해결할 수 있다.



# 합성(Composition)

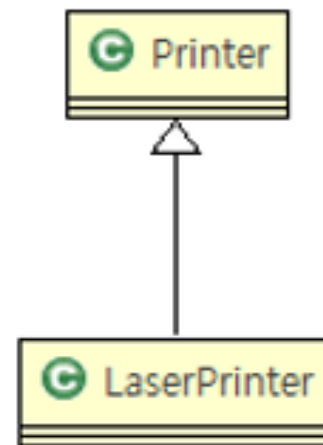
- 합성은 객체가 자신의 기능을 직접 수행하지 않고, \*\*다른 객체의 메소드를 사용해 작업을 위임(Delegation)\*\*하는 설계 방식이다. 서로 대등한 관계에서 기능을 결합한다는 점이 특징이다.
- 클라이언트는 FrontClass만 알고, 실제 작업은 BackClass가 수행하지만 그 존재는 모른다. 즉, FrontClass가 중간 역할을 하며 내부 구현을 숨긴 구조다.→ 이는 \*\*캡슐화와 의존성 숨기기(추상화)\*\*의 예이다.



# 상속

```
public abstract class Printer {  
    private String id;  
    public String getId() {  
        return id;  
    }  
    abstract void print(Object message);  
}
```

```
public class LaserPrinter extends Printer {  
    public void print(Object message) {  
        System.out.println(message);  
    }  
}
```



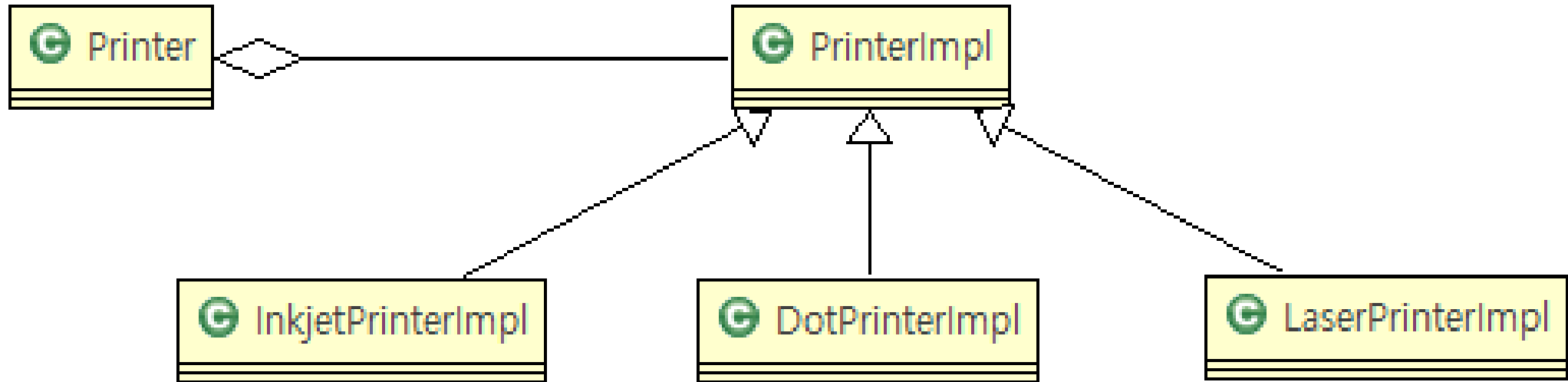
Person의 getId() 리턴타입이  
String에서 int로 변경된다면



```
if ( printers[i].getId().equals("101") ) {  
    if ( printers[i].getId() == 101 ) {
```

상속은 부모와 자식 클래스 간에 강한 결합을 만들어, 부모 클래스의 메소드 시그니처가 변경되면 자식 클래스뿐 아니라 모든 클라이언트 코드에도 영향을 미친다.

# 합성(Composition)



Printer 클래스는 내부에 PrinterImpl 객체를 포함(합성)하고, print()와 getld() 메소드 호출을 PrinterImpl에 위임한다. PrinterImpl의 하위 클래스들(InkjetPrinterImpl, DotPrinterImpl 등)은 프린터의 구체적 동작을 구현하며, 상속 없이도 재사용 효과를 얻을 수 있는 구조다.→ 이는 합성을 통한 유연한 코드 재사용 방식을 보여준다.

```

public class Printer {
    private String id;
    private PrinterImpl printerImpl;

    public Printer(PrinterImpl printerImpl) {
        this.printerImpl = printerImpl;
    }

    public String getId() {
        return printerImpl.getId();
    }

    public void print(Object message) {
        printerImpl.print(message);
    }
}

```

```

abstract public class PrinterImpl {
    private String id;

    public String getId() {
        return id;
    }

    return new Integer(printerImpl.getId()).toString();

    abstract public void print(Object message);
}

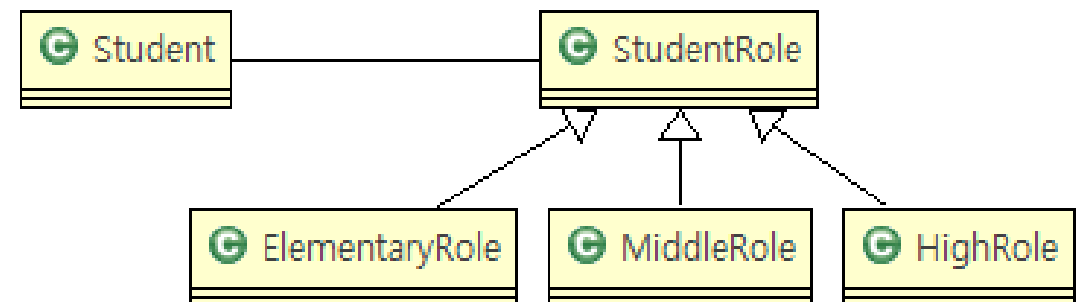
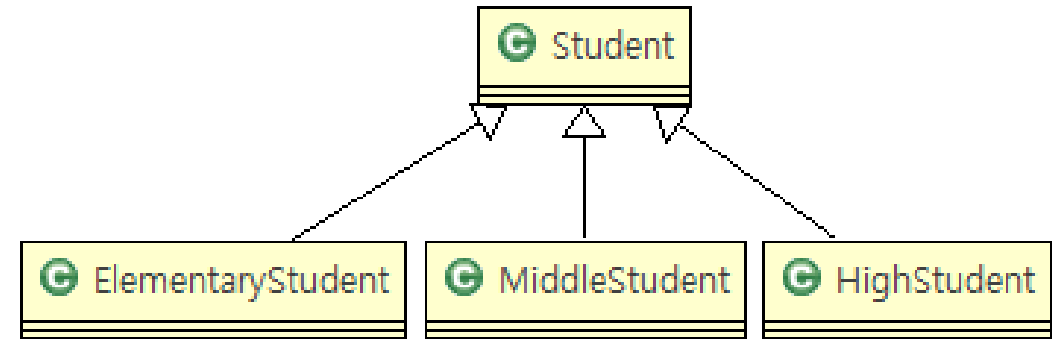
public class LaserPrinterImpl extends PrinterImpl {
    public void print(Object message) {
        System.out.println(msg.toString());
    }
}

```

Printer 클래스의 getId() 메소드의 리턴 타입이 변경되도 Printer 클래스를 사용하는 모든 클라이언트는 수정되지 않고 PrinterImpl 클래스의 getId() 메소드만 다음과 같이 수정하면 된다.

# 변하는 역할은 합성으로, 변하지 않는 본질은 상속으로

- 합성은 시간이 지나며 \*\*변할 수 있는 역할(Role)\*\*을 표현할 때 적합하다.
- 상속은 본질적으로 변하지 않는 항구적인 IS-A 관계를 표현할 때 사용한다.
- 예시:
  - 남학생/여학생 → 학생: 변하지 않는 본질 → 상속 사용
  - 초등학생 → 중학생 → 고등학생: 시간이 지나며 변함 → 합성 사용
- 합성을 쓰면 역할이 바뀔 때마다 객체 내부의 역할만 교체하면 되므로 유지보수와 확장성이 높아진다.



# section04 소프트웨어의 변화와 유지보수성

소프트웨어 품질 기준

유지보수성의 하위 특성

# 소프트웨어 품질을 결정하는 6가지 기준

- 기능성(Functionality) – 사용자가 원하는 기능이 오류 없이 잘 작동하는 정도
- 안정성(Reliability) – 성능을 유지하며 문제 없이 작동하는 정도
- 사용 편의성(Usability) – 사용자가 쉽게 사용할 수 있는 정도
- 효율성(Efficiency) – 자원을 적게 쓰면서 기능을 수행하는 정도
- 유지보수성(Maintainability) – 수정이나 기능 추가가 얼마나 쉬운지
- 이식성(Portability) – 다른 환경에서도 쉽게 사용할 수 있는지

# 소프트웨어 유지보수의 3가지 주요 유형

- 기존 코드의 개선
  - 정상적으로 동작하는 코드들을 수정함으로 소프트웨어의 유지보수성을 높이거나 성능을 개선한다.
- 오류 수정
  - 소프트웨어가 의도한 기능 대로 동작하지 않으면 그 원인을 찾아내어 제거한다.
- 고객의 새로운 요구사항 반영
  - 소프트웨어를 주문한 고객, 또는 시장에서 원하는 요구사항들을 반영하도록 코드를 변경한다. 주로 새로운 기능을 추가하거나 기존 기능을 삭제, 변경한다.



section05 코드의 수정

# 안정적인 소프트웨어 수정을 위한 원칙

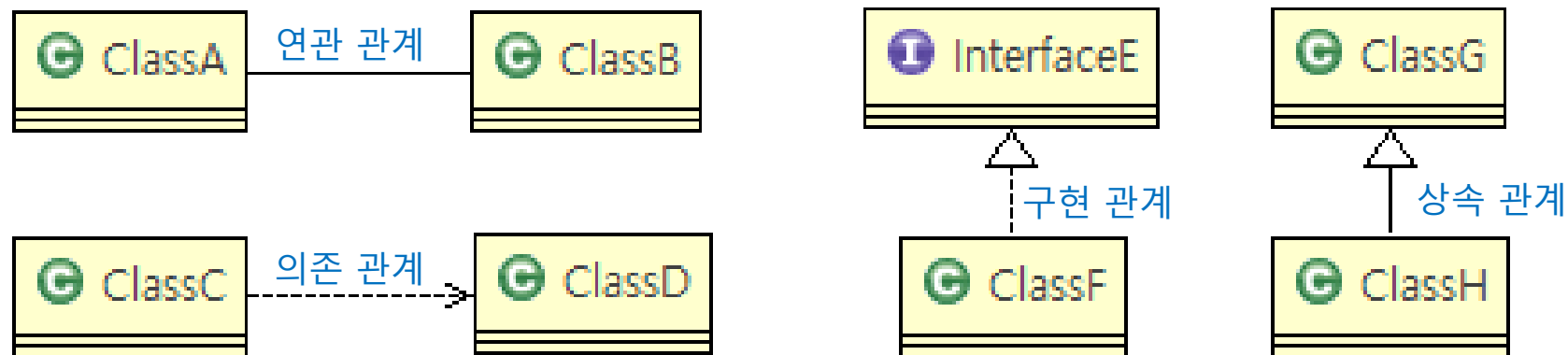
- 코드 변경 시 고려 사항
  - 추가 방식 / 변경 위치 / 변경 분량
- 상속을 통한 확장의 장점
  - 기존 코드 변경보다 상속을 통해 새로운 클래스를 정의하는 것이 오류 발생 가능성을 줄임.
  - 변경할 코드의 양이 적을수록 작업이 수월함.
- 유지보수성 높은 소프트웨어
  - 기존 클래스를 유지하고 확장을 통해 새로운 클래스를 정의하는 방식이 바람직함.
  - 변경 작업을 수행할 때, 변경된 클래스의 객체를 사용하는 클래스는 일관성이 있어야 함.
  - 변경해야 하는 코드의 분량은 적을수록 좋음.
  - 어쩔 수 없이 변경해야 하는 부분은 최소화되어야 함.

# 변화에 유연한 객체지향 설계의 원칙

- 인터페이스 변경의 영향을 최소화해야 한다
- 기존 코드는 수정하지 않고 유지하는 것이 바람직하다.
- 새로운 기능은 확장을 통해 추가하는 방식이 이상적이다.

# 변경에 강한 객체지향 설계를 위한 클래스 간 관계 이해

- 성능 개선이나 기능 추가로 인해 클래스나 인터페이스의 메소드 시그니처 변경이 발생할 수 있다.
- 이 경우, 해당 클래스를 사용하는 모든 클라이언트 코드도 함께 수정해야 하므로 매우 주의가 필요하다.
- 상속, 연관, 의존 관계가 있는 클래스는 인터페이스 변경의 영향을 직접 받게 된다.
- 따라서 클래스는 다른 클래스와의 관계를 최소화할수록 변경에 강한 구조가 된다. (디미터 법칙)
- 연관 관계(Association): ClassA가 ClassB를 직접 참조하는 관계 ("has-a")
- 의존 관계(Dependency): ClassC가 ClassD를 일시적으로 사용하는 관계 (메소드 호출, 객체 생성 등)



```

class Printer {
    private String type;
    public Printer(String type) {
        this.type = type;
    }
    public void print(String message) {
        if(type.equals("DOT")) {
            dotPrint(message);
        } else if(type.equals("INKJET")) {
            inkjetPrint(message);
        }
    }
    public void dotPrint(String content) {
        System.out.println("도트 프린터 : " + message);
    }
    public void inkjetPrint(String content) {
        System.out.println("잉크젯 프린터 : " + message);
    }
}

```

```

public class PrinterTest {
    public static void main(String[] args) {
        Printer printer = new Printer("DOT");
        printer.print("샘플 출력...");
    }
}

```

- 기능 추가 시마다 기존 클래스의 메소드를 반복해서 수정해야 하는 경우, 유지보수가 어렵다.
- 특히 \*\*분기문(if/switch)\*\*으로 프린터 종류를 구분하는 방식은, 새로운 프린터 추가 시마다 print() 메소드를 수정해야 한다.

```

abstract class Printer {
    public abstract void print(String content);
}

class DotPrinter extends Printer {
    public void print(String content) {
        System.out.println("도트 프린터 : " + content);
    }
}

class InkjetPrinter extends Printer {
    public void print(String content) {
        System.out.println("잉크젯 프린터 : " + content);
    }
}

public class PrinterTest {
    public static void main(String[] args) {
        Printer printer = new DotPrinter();
        printer.print("샘플 메시지");
    }
}

```

- 새로운 프린터 종류를 추가할 때 상속만 하면 되므로 확장에 유리하다.
- 이전 코드는 기존 print() 메소드를 계속 수정해야 하므로 실수로 다른 로직에 영향을 줄 수 있어 에러 발생 가능성이 높다.
- 상속의 방식은 변화에 더 유연하고 안전한 설계이다.

# section06 소프트웨어 설계 원칙

결합도

응집도

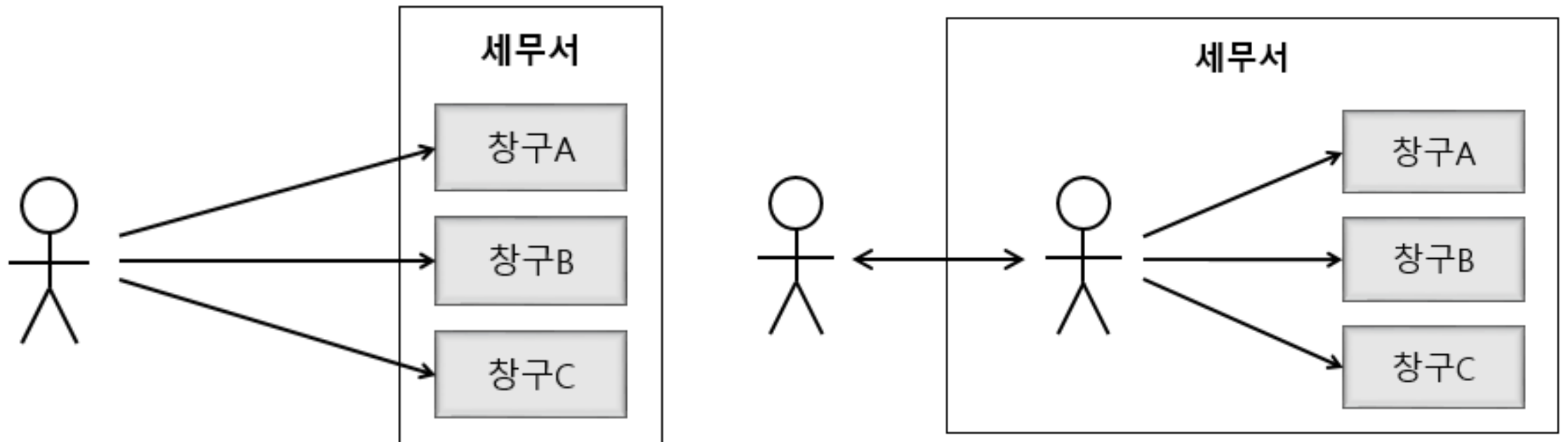
OAOO(Once And Only Once)

# 결합도(Coupling)

- 유지보수성이 높은 소프트웨어의 가장 중요한 특징은 프로그램의 각 요소들이 결합도는 낮게, 응집도는 높게 구성해야 한다는 것이다.
- **결합도**란 소프트웨어의 한 요소가 다른 것과 얼마나 강력하게 연결되어 있는지, 또한 얼마나 의존적인지를 나타내는 정도다.
- 프로그램의 요소가 결합도가 낮다는 것은 그것이 다른 요소들과 관계를 그다지 맺지 않는 상태를 의미한다.
- 결합도가 높으면 과도하게 많은 다른 요소들과 얽히게 된다.



# 결합도(Coupling)



사용자에게 복잡한 절차를 감추고, 직원이 대신 처리함으로써 세무 업무를 쉽게 만든 구조이다.

# 결합도(Coupling)

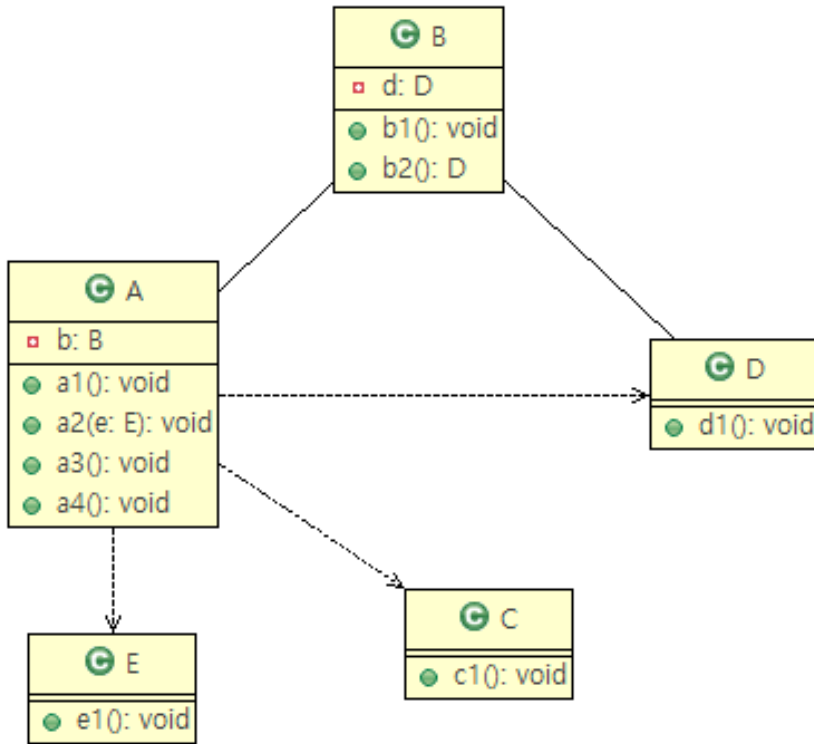
- 변경에 취약함→ 다른 클래스에 의존하고 있어, 해당 클래스가 변경되면 같이 수정해야 한다.
- 재사용 어려움→ 다른 클래스와 강하게 연결되어 있어, 독립적으로 재사용하기 힘들다.
- 이해하기 어려움→ 다양한 클래스와 복잡하게 얽혀 있어서 전체 구조를 파악하기 어렵다.
- 테스트 어려움→ 의존성이 많아 단위 테스트를 하려면 관련 클래스들도 함께 준비해야 한다.
- 확장에 제약→ 구조가 단단히 엮여 있어 새로운 기능을 추가하거나 수정하기 어렵다.
- 결합도가 높을수록 유지보수성과 유연성이 떨어진다는 것이 핵심이다.

# 결합도(Coupling)

- 결합도는 낮을수록 좋다→ 클래스 간 의존성이 적을수록 바람직하다.
- 불필요한 연관은 피해야 한다→ 꼭 필요한 클래스와만 연결되어야 한다.
- 개발자는 결합도를 낮추도록 노력해야 한다→ 구조 설계 시 낮은 결합도를 목표로 해야 한다.
- 클래스는 꼭 필요한 대상과만 연결되고, 불필요한 의존은 제거해야 결합도가 낮아져 유지보수가 쉬워진다.

# 다양한 객체 결합 방식과 그 의미

- 속성 객체 결합 (연관 관계)
  - 다른 객체를 멤버 변수로 갖고 그 객체의 메소드를 사용함.
- 로컬 객체 결합 (의존 관계)
  - 메소드 안에서 다른 객체를 직접 생성해 사용함.
- 파라미터 객체 결합 (의존 관계)
  - 메소드가 파라미터로 받은 객체의 메소드를 호출함.
- 반환 객체 결합 (의존 관계)
  - 다른 객체로부터 리턴받은 객체를 통해 작업함.
- 상속 결합 (상속 관계)
  - 부모 클래스를 상속받아 기능을 확장함.
- 인터페이스 결합 (구현 관계)
  - 인터페이스를 구현하여 기능을 정의함.
- 결합은 객체를 어떻게 참조하고 사용하는지에 따라 여러 형태로 나뉘며, 의존성의 방향과 강도에 따라 구조 설계에 영향을 미친다.



- 클래스 A가 여러 클래스(B, C, D, E)와 다양한 방식으로 결합되어 있다.
- 결합도(Coupling) 측면에서 바라볼 때, 클래스 A는 지나치게 많은 클래스에 의존하고 있어 유지보수성과 확장성이 떨어지는 문제가 있다.

```

class A {
    private B b;
    public void a1() {
        b.b1();
    }
    public void a2(E e) {
        e.e1();
    }
    public void a3() {
        C c = new C();
        c.c1();
    }
    public void a4() {
        b.b2().d1();
    }
}
  
```

```

class B {
    private D d;
    public void b1() {
    }
    public D b2() {
        return d;
    }
}
class C {
    public void c1() {
    }
}
class D {
    public void d1() {}
}
class E {
    public void e1() {}
}
  
```

## 클래스 A의 결합도 문제 분석

### 1. `a1()` 메소드 → 속성 객체 결합 (연관관계)

```
java  
  
b.b1();
```

- 클래스 A는 B를 **멤버 변수**로 보유하고 있고, 그 내부 메소드를 직접 호출합니다.
- 문제점:** B가 변경되면 A도 함께 영향을 받습니다. 예: `b1()` 이 변경되면 `a1()` 도 수정 필요.

### 2. `a2(E e)` 메소드 → 파라미터 객체 결합 (의존관계)

```
java  
  
e.e1();
```

- 외부에서 전달된 E 객체에 의존하여 메소드를 호출합니다.
- 문제점:** 상대적으로 약한 결합이지만, E의 구조 변경이 A에 영향을 줄 수 있습니다.

### 3. a3() 메소드 → 로컬 객체 결합 (의존관계)

java

```
C c = new C();  
c.c1();
```

- A 내부에서 직접 C 객체를 생성하고 사용합니다.
- **문제점:** A가 C 생성에 직접 책임을 지므로, C의 변경이나 대체가 어려움 (예: 테스트 시 mock 사용 어려움)

### 4. a4() 메소드 → 반환 객체 결합 (의존관계의 연쇄)

java

```
b.b2().d1();
```

- B를 통해 D를 받아오고, 다시 D의 메소드를 호출합니다.
- **문제점:** A는 직접적으로 D와 관계가 없어 보이지만, 간접 의존이 존재하여 구조가 복잡해지고, D의 변경도 A에 영향을 줌.

```

class A {
    private B b;
    public void a1() {
        b.b1();
    }
    public void a2(E e) {
        e.e1();
    }
    public void a3() {
        C c = new C();
        c.c1();
    }
    public void a4() {
        b.b3();
    }
}

```

```

class B {
    private D d;
    public void b1() {
    }
    public D b2() {
        return d;
    }
    public void b3() {
        d.d1();
    }
}

```

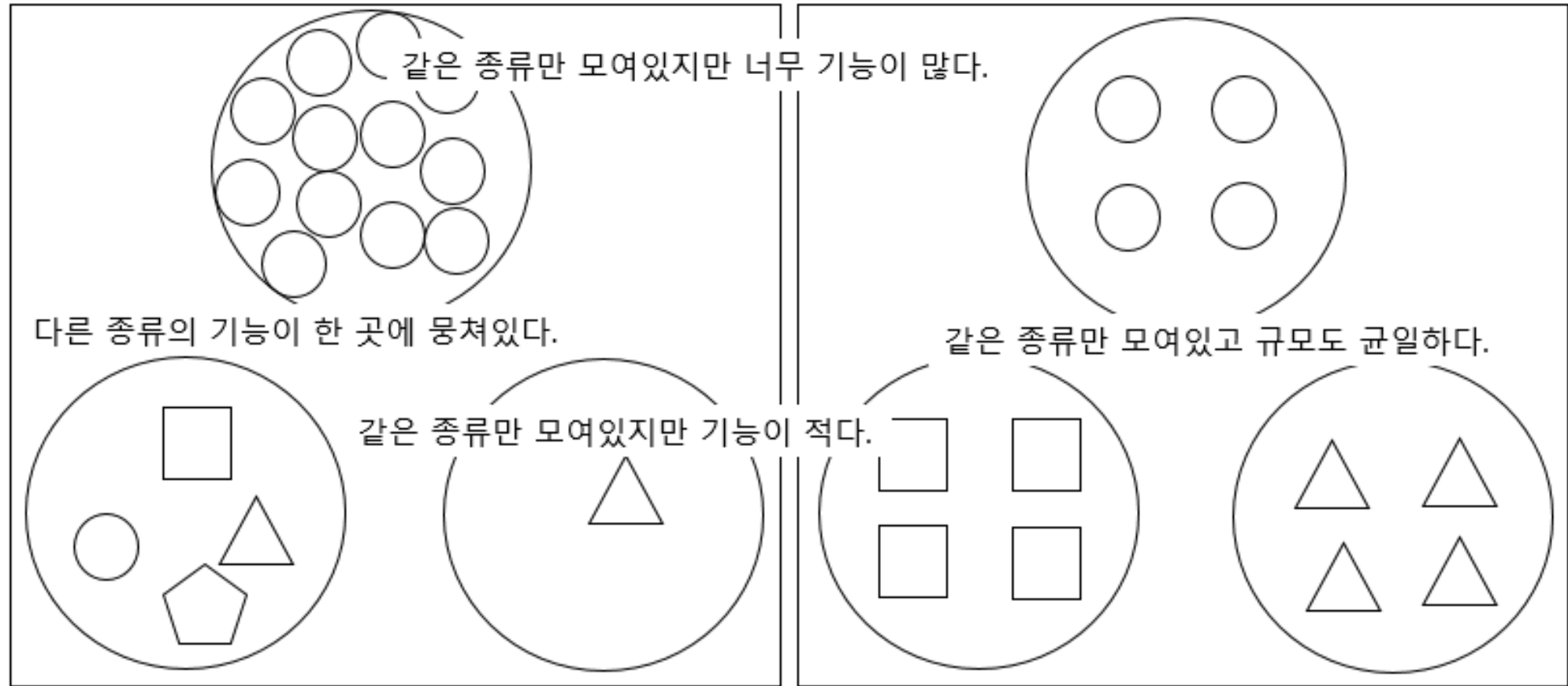
- 이전 코드 (b.b2().d1() 호출)  
 → A는 D 객체를 리턴 받아 직접 조작함.  
 → 이로 인해 A는 B뿐 아니라 D까지도 결합됨 → 높은 결합도, 낮은 캡슐화.
- 지금 코드 (b.b3() 호출)  
 → A는 B에게 행위를 요청만 하고, D는 B 내부에서만 사용됨.  
 → 결과적으로 A는 D에 대한 지식이 없음 → 결합도 낮아지고 캡슐화 향상.



# 응집도(Cohesion)

- 응집도란 프로그램의 특정 요소(클래스, 메소드)가 해당 기능 수행에 필요한 책임만으로 뭉쳐있는 정도를 나타낸다.
- 일반적으로 프로그램의 한 요소가 특정 목적을 위해 밀접하게 연관된 기능들만으로 구현되어 있고, 과도하게 많은 일을 처리하지 않으면 응집도가 높다고 이야기한다.
- 응집도가 높으면 프로그램을 쉽게 이해할 수 있으므로 가독성과 유지보수성은 자연스럽게 향상된다.

# 응집도(Cohesion)



<응집도가 낮은 경우>

<응집도가 높은 경우>

# 응집도의 중요성과 설계 원칙

- 응집도가 높으면 기능 추가, 변경, 삭제가 쉬워지고 구조가 직관적이다.
- 클래스의 메소드들이 하나의 목적에 집중되어 있어야 높은 응집도를 가진다.
- 응집도가 낮으면 유지보수가 어려워지고, 관리가 복잡해진다.
- 클래스는 책임에 맞는 기능만을 포함하고, 과도한 역할을 피해야 한다.

# 응집도 높은 클래스의 조건

- 클래스는 같은 목적의 기능을 가진 메소드들로 구성되어야 한다.
- 각 메소드는 단일 기능만 수행하고, 개수가 적당해야 한다.
- 너무 많은 일을 혼자 하지 말고, 다른 클래스와 역할을 나눠야 한다.
- 메소드가 너무 크다면 별도의 클래스로 분리하는 것도 고려해야 한다.

# 응집도 낮은 클래스의 문제와 개선 전략

- **응집도 낮은 클래스의 문제**

: 여러 기능이 뒤섞이면 하나의 목적에 집중할 수 없고, 클래스가 혼란스럽고 복잡해진다.

- **이미지와 소리 처리**

: ImageAndSoundProcessing 클래스는 관심사가 다름 → ImageProcessing, SoundProcessing으로 기능 영역별로 분리하여 응집도 향상.

- **결제 기능**

: Payment 클래스에 다양한 결제 방식이 섞여 있음 → Cash, Check, CreditCard 등으로 분리하여 응집도 향상

- **응집도와 결합도 관계**

: 응집도와 결합도는 상호 보완적이지만, 때로는 응집도를 높이면 결합도가 증가하거나 그 반대가 될 수도 있다. → 둘 사이의 균형이 중요

- **응집도는 높이고, 결합도는 낮추는 방향을** 유지하며 균형 있는 설계를 해야 한다.

# 중복 없는 코드, OAOO 원칙의 힘

- **OAOO(Once And Only Once) 원리**  
: 동일한 기능은 단 한 곳에만 존재해야 한다는 원칙으로, XP 핵심 원리 중 하나.
- **코드 중복 방지**  
: 같은 기능을 여러 번 작성하지 않음으로써 코드 관리가 쉬워지고, 품질이 향상된다.
- **유지보수성 향상**  
: 중복이 없으면 변경이 필요한 경우 한 곳만 수정하면 되므로 유지보수가 쉬워짐.
- **책임 분산 방지**  
: 중복 코드는 책임이 여기저기 흩어져 구조 파악과 수정이 어려워진다. OAOO 원칙을 지키면 명확한 책임 분리가 가능해짐.
- **AOP와 유지보수성**  
: AOP는 **\*\*반복되는 기능(공통 관심사)\*\***을 분리해, 코드 중복 없이 유지보수가 쉬운 구조를 만든다. 대표적 도구는 Spring AOP 프레임워크

결합도를 낮게 하고 응집도를 높이며, OAOO를 지키는 소프트웨어는 어떤 점에서 유지보수성을 높이는가?

- **변경해야 하는 코드의 위치를 전체 소스코드에서 잘 찾을 수 있을까?**

: 관련된 기능이 한 곳에 모여있고 중복된 코드가 없기 때문에 수정할 부분이 한 곳에서만 발견된다.

- **코드를 쉽고 빠르게 수정할 수 있을까?**

: 중복된 부분이 없기 때문에 한곳만 수정하면 된다. 클래스들이 책임을 적절하게 갖고 분담하기 때문에 상속이나 위임을 통해 확장하기 쉽다.

- **코드를 수정한 이후에도 시스템이 안정적으로 동작할 확률이 높아질까?**

: 결합도가 낮으므로 코드를 변경했을 때 심각한 영향을 다른 곳에 미칠 확률이 낮아진다. 변경된 코드 부근에서 부정적인 영향이 차단될 수 있다.

- **코드를 수정한 이후에 그 부분과 전체 시스템이 제대로 동작하는지 테스트를 쉽게 할 수 있을까?**

: 결합도가 낮으므로 변경한 부분을 제외한 나머지 영역을 테스트하기가 상대적으로 용이하다.

# section07 객체지향 설계 원칙

ORR(One Response Rule) 원칙

OCP(Open Closed Principle) 원칙

LoD(Law of Demeter) 원칙



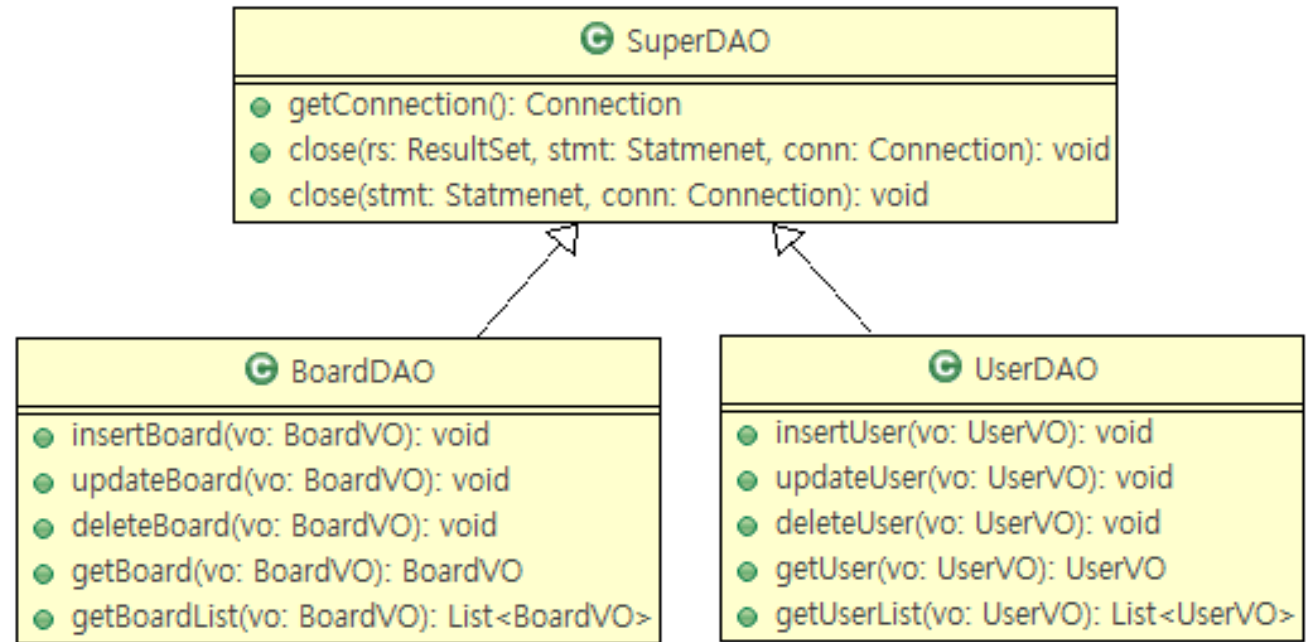
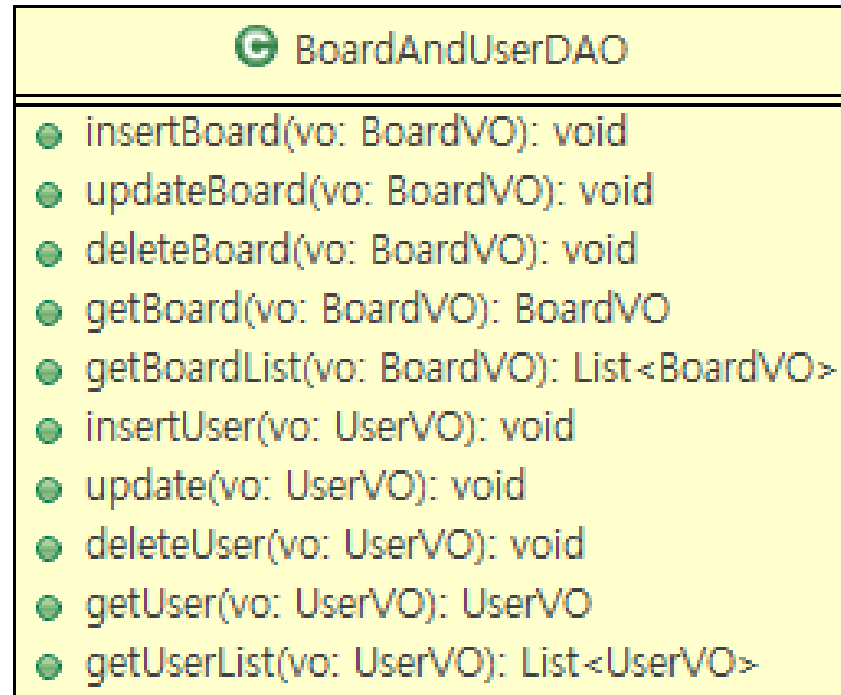
# ORR(One Responsibility Rule) 원칙

- ORR은 클래스나 메소드가 자신에게 부여된 한 가지 고유한 책임만을 수행해야 한다는 원칙이다.
- 이 원칙은 높은 응집도와 밀접하게 관련되어 있기때문에 이 원칙을 따르면 자연스럽게 프로그램의 응집도가 높아진다.
- ORR 원칙을 풀어서 설명하면 다음과 같다.
  - 그 일을 자신만이 유일하게 해야 하며,
  - 그 일을 다른 클래스의 메소드보다 더 잘 할 수 있어야 한다.
  - 그리고 그 책임에 해당하는 일을 빠짐없이 모두 해야 한다.

# ORR(One Responsibility Rule) 원칙 위배

- 과도한 기능 집중
  - 복잡하고 많은 기능을 포함한 클래스는 불필요한 코드가 많고, 많은 기능이 하나의 클래스에 몰려 있음.
  - 새로운 기능 추가나 기존 기능 수정이 어려움.
- 협업 및 형상 관리 어려움
  - 프로젝트에서 공통 기능으로 사용되는 경우, 다수의 개발자가 동시에 수정할 가능성이 높아 소스코드 버전 충돌 빈번.
  - CVS나 SVN을 이용한 형상 관리를 어렵게 만듦.
- 성능 저하
  - 객체 크기가 커지면 불필요한 메모리 사용 증가.
  - 공통 기능을 포함한 객체의 빈번한 호출로 전체 소프트웨어 성능 저하.
- 해결 방안
  - 기능을 관리 가능한 단위로 나누고, 유형별로 분류하여 새로운 클래스로 위임.
  - 일관성 유지.

# ORR(One Responsibility Rule) 원칙



# 설계 없는 구현, 스파게티 코드의 시작

- 메소드도 클래스처럼 한 가지 일만 해야 하며, 기능이 혼합되어 있다면 분리하거나 적절한 클래스로 옮겨야 한다.
- 복잡한 메소드는 요구사항 분석 부족과 코딩 우선 습관에서 비롯되며, 이는 스파게티 코드로 이어진다.
- 많은 프로젝트가 여전히 설계 없이 코딩부터 시작하고, 문제가 생기면 그때그때 수정하는 방식을 택한다.
- 이러한 방식은 구현 중심 사고로 인해 단기 성취감은 주지만, 장기적 유지보수성과 확장성은 크게 떨어진다.
- 제시된 정렬 코드 예시는 설계 없이 비효율적으로 구현된 대표적 사례이다.

```

public class NumberSortingTest {
    public static void main(String[] args) {
        int[] array = { 0, 3, 5, 2, 6, 7, 8, 9, 1, 4 };
        int[] sortedArray = new int[array.length];
        int temp;
        try {
            if (array.length > 0) {
                if (sortedArray.length > 0) {
                    if (sortedArray.length == array.length) {
                        for (int i = 0; i < array.length; i++) {
                            for (int k = i + 1; k < array.length; k++) {
                                if (array[i] > array[k]) {
                                    temp = array[i];
                                    array[i] = array[k];
                                    array[k] = temp;
                                }
                            }
                        }
                        for (int v : array) {
                            System.out.print(v + " ");
                        }
                        System.out.print("\n");
                    }
                }
            }
        }
    }
}

```

```

        for (int i = 0; i < array.length; i++) {
            sortedArray[i] = array[i];
        }
    }
}

System.out.println("최종 결과");
for (int v : sortedArray) {
    System.out.print(v + " ");
}

} catch (Exception e) {
    e.printStackTrace();
}

}
}

```

```
import java.util.Arrays;

public class NumberSortingTest {
    public static void main(String[] args) {
        int[] array = { 0, 3, 5, 2, 6, 7, 8, 9, 1, 4 };
        int[] sortedArray = new int[array.length];
        Arrays.sort(array);


        System.arraycopy(array, 0, sortedArray, 0, array.length);
        System.out.println("최종 결과");
        for (int num : sortedArray) {
            System.out.print(num + " ");
        }
    }
}
```

# OCP(Open Closed Principle) 원칙

- OCP 원칙은 새로운 기능을 추가할 때 기존 코드를 최소한으로 변경하는 원칙으로, 디자인 패턴과 다형성을 활용해 구현된다.
- 예를 들어, 모듈화된 로봇은 팔을 쉽게 교체할 수 있지만, 모듈화되지 않은 로봇은 전체를 부수고 수정해야 한다.
- OCP를 따르면 기존 코드를 수정하지 않고도 새로운 기능을 추가하거나 변경할 수 있다.

# OCP(Open Closed Principle) 원칙

```
class Employee {  
    private int empSalary;  
    private String empType; // mere, manager 등  
  
    public Employee(int empSalary, String empType) {  
        this.empSalary = empSalary;  
        this.empType = empType;  
    }  
  
    public int calculateSalary() {  
        if (empType.equals("Mere"))  
            empSalary += 90;  
        else if (empType.equals("Manager"))  
            empSalary += 120;  
        return empSalary;  
    }  
}
```



새로운 직원 추가시 코드 수정




# OCP(Open Closed Principle) 원칙

```
abstract class Employee {  
    private String empName;  
    private int empSalary;  
    public Employee(String empName, int empSalary) {  
        this.empName = empName;  
        this.empSalary = empSalary;  
    }  
    public abstract int calculateSalary();  
}
```

# OCP(Open Closed Principle) 원칙

```
class MereEmployee extends Employee {  
    public MereEmployee(String empName, int empSalary) {  
        super(empName, empSalary);  
    }  
    public int calculateSalary() {  
        int salary = getEmpSalary();  
        return salary += 90;  
    }  
}
```

```
class ManagerEmployee extends Employee {  
    public ManagerEmployee(String empName, int empSalary) {  
        super(empName, empSalary);  
    }  
    public int calculateSalary() {  
        int salary = getEmpSalary();  
        return salary += 120;  
    }  
}
```



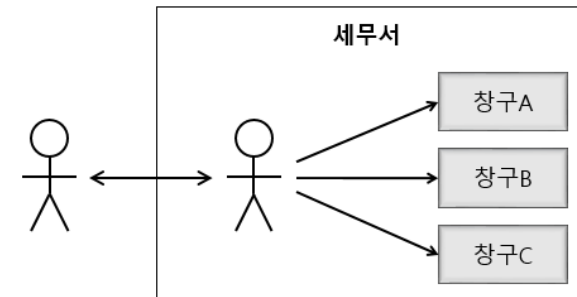
Employee를 상속받는 클래스  
추가로 새로운 종류의 직원 추가

# LoD(Law of Demeter) 원칙

- 이 원칙은 낮은 결합도와 관련이 있다.
- 원칙을 지키면 프로그램의 결합도가 낮아져, 클래스의 특정 부분을 수정해도 다른 곳에 심각한 영향을 주지 않는다.
- 이 원칙에 따르면 클래스의 메소드는 다음 객체들의 메소드만 호출해야 한다.
  - 자기 자신이 가지고 있는 메소드 혹은 상위 클래스로부터 상속된 메소드
  - 클래스의 속성 객체가 가진 메소드
  - 메소드의 파라미터로 전달되는 객체의 메소드
  - 내부에서 직접 생성된 객체의 메소드
  - 다른 클래스의 메소드 안에서 생성되어 리턴되는 객체의 메소드

# LoD(Law of Demeter) 원칙

- 세무서 예시처럼, 고객(클래스)은 내부 처리(로직)를 알 필요 없이 직원(직접 참조된 객체)에게만 요청하면 된다.
- 이 원칙은 "직원이 아닌 사람과는 이야기하지 말라"는 규칙으로, Law of Demeter (LoD)를 의미한다.
- 클래스는 자신의 멤버 변수나 메소드 인자로 받은 객체(직원)만 사용해야 하며, 다른 객체의 속성 객체를 반환받아 사용하는 것(반환 객체 결합)은 LoD 위반이다.
- 단, 다른 클래스 내부에서 생성된 로컬 객체를 반환받는 것은 예외로 허용된다.



# 리팩토링 전

```
public class Bank {
    public Map<String, Integer> safe;
    public Bank() {
        safe = new HashMap<String, Integer>();
    }
    public void makeAccount(String customerCode, int currentMoney) {
        safe.put(customerCode, currentMoney);
    }
}

public class BankTeller {
    public Bank bank;
    public BankTeller() {
        bank = new Bank();
    }
    public void makeAccount(String customerCode, int currentMoney) {
        bank.makeAccount(customerCode, currentMoney);
    }
}
```

## ✓ 리팩토링 전: LoD 위반

java

```
teller.bank.safe.get(customerCode);  
teller.bank.safe.put(customerCode, deposit - money);
```

- `Customer` 가 `BankTeller` 를 통해 `Bank` 에 접근하고, 다시 `Bank` 내부의 `safe` 라는 **Map 구조까지 직접 조작합니다.**
- 이는 LoD 원칙에서 금지하는 "**연쇄적인 객체 탐색**", 즉 **\*\*반환 객체 결합(Return Object Coupling)\*\***입니다.
- `Customer` 는 너무 많은 객체(`BankTeller`, `Bank`, `Map`)에 대해 **직접 알고 있으며**, 이 중 하나라도 바뀌면 `Customer` 도 함께 수정해야 하므로 **결합도가 매우 높습니다.**

```

public class Customer {
    public String customerCode;
    public int currentMoney;
    public BankTeller teller;
    public Customer(String customerCode, int currentMoney) {
        teller = new BankTeller();
        this.currentMoney = currentMoney;
        this.customerCode = customerCode;
    }
    public int withdrawal(int money) {
        int deposit = teller.bank.safe.get(customerCode);
        teller.bank.safe.put(customerCode, deposit - money);
        return deposit - money;
    }
    public int deposit(int money) {
        int deposit = teller.bank.safe.get(customerCode);
        teller.bank.safe.put(customerCode, deposit + money);
        return deposit + money;
    }
    public void makeAccount() {
        teller.makeAccount(customerCode, currentMoney);
    }
}

```

- Customer가 예금과 인출 작업을 하기 위해서는 창구직원(BankTeller)과 은행(Bank)과 금고 정보를 모두 알아야 한다.
- 만약 이들 중 하나라도 변경이 발생되면 Customer를 무조건 수정되어야 한다.
- 이렇게 LoD를 어긴 프로그램은 결합도가 높아 지기 때문에 유지보수가 어려울 수밖에 없다.

# 리팩토링 후

```
public class Bank {  
    private Map<String, Integer> safe;  
    public Bank() {  
        safe = new HashMap<String, Integer>();  
    }  
    public int withdraw(String customerCode, int money) {  
        int balance = safe.get(customerCode) - money;  
        safe.put(customerCode, balance);  
        return balance;  
    }  
    public int deposit(String customerCode, int money) {  
        int balance = safe.get(customerCode) + money;  
        safe.put(customerCode, balance);  
        return balance;  
    }  
    public void makeAccount(String customerCode, int currentMoney) {  
        safe.put(customerCode, currentMoney);  
    }  
}
```



```
public class BankTeller {  
    private Bank bank;  
    public BankTeller() {  
        bank = new Bank();  
    }  
    public int withdrawal(String customerCode, int money) {  
        return bank.widtrawal(customerCode, money);  
    }  
    public int deposit(String customerCode, int money) {  
        return bank.deposit(customerCode, money);  
    }  
    public void makeAccount(String customerCode, int currentMoney) {  
        bank.makeAccount(customerCode, currentMoney);  
    }  
}
```

```
public class Customer {  
    private String customerCode; private int currentMoney; private BankTeller teller;  
  
    public Customer(String customerCode, int money) {  
        teller = new BankTeller();  
        this.currentMoney = money;  
        this.customerCode = customerCode;  
    }  
    public int withdrawal(int money) {  
        teller.withdrawal(customerCode, money);  
        return currentMoney -= money;  
    }  
    public int deposit(int money) {  
        teller.deposit(customerCode, money);  
        return currentMoney += money;  
    }  
    public void makeAccount() {  
        teller.makeAccount(customerCode, currentMoney);  
    }  
}
```

## ✓ 리팩토링 후: LoD 준수

java

```
teller.withdrawal(customerCode, money);
```

- `Customer` 는 이제 자신이 직접 참조하는 객체인 `teller` 에게만 메시지를 전달합니다.
- `Bank`, `Map`, 내부 구조 등은 `Customer` 가 전혀 알지 못하도록 감춰져 있습니다.
- 모든 내부 로직은 `BankTeller` 와 `Bank` 클래스 내부에서 처리되므로, 책임이 적절히 분리되고 캡슐화가 잘 유지됩니다.

## ✓ LoD 준수의 장점 요약

항목	리팩토링 전	리팩토링 후
내부 구조 접근	<code>Customer</code> 가 <code>Map</code> 까지 직접 접근	<code>BankTeller</code> 에만 메시지 전달
객체 간 결합도	높음	낮음
유지보수성	약함: 구조 변경 시 다수 수정 필요	강함: 변경에 유연
캡슐화	깨짐	잘 지켜짐
LoD 위반 여부	O	✗ 없음 (잘 지킴)