

# OOP - 1. 절차지향과 비교하기

## 절차지향 vs 객체지향 완벽 가이드

"자바를 사용하면서도 절차지향적인 코드가 나올 수 있다"

### 1. 프로그래밍 패러다임의 이해

#### 1.1 세 가지 프로그래밍 패러다임

패러다임	영문명	특징	중심 사고
순차지향	Sequential Oriented	코드가 위에서 아래로 순차 실행	명령어의 순서
절차지향	Procedure Oriented	함수 중심의 프로그램 구성	어떻게 처리할까?
객체지향	Object Oriented	객체와 메시지 중심	누가 처리할까?

#### 핵심 이해

많은 개발자들이 순차지향과 절차지향을 같은 개념으로 이해하지만, 실제로는 다릅니다.

- 순차지향: "Sequential" → 말 그대로 순차적 실행
- 절차지향: "Procedure" → 컴퓨터 공학에서 "함수"를 의미

 절차지향 = 함수 지향 프로그래밍

### 2. 언어별 비교 분석

#### 2.1 어셈블리어 (순차지향)

```
add:
    subq $8, %rsp
    leaq 16(%rsp), %rax
    movq %rax, 0(%rsp)
    leal 0(%edi,%esi,1), %eax
    addq $8, %rsp
    ret

main:
    subq $8, %rsp
    leaq 16(%rsp), %rax
    movq %rax, 0(%rsp)
```

```
movl $2, %edi
movl $3, %esi
addq $8, %rsp
jmp add
```

#### 특징:

- ❌ 함수 개념 없음
- ✅ jmp, goto 같은 흐름 제어
- ✅ 레이블과 주소 기반 이동

## 2.2 🛠️ C언어 (절차지향)

```
int add(int a, int b) {
    return a + b;
}

int main() {
    int a = 2;
    int b = 3;

    return add(a, b);
}
```

#### 특징:

- ✅ 함수 중심 구조
- ✅ 복잡한 문제를 개별 함수로 분해
- ✅ 함수들의 조합으로 문제 해결

## ⚠️ 3. 자바에서의 절차지향적 코딩

🔥 주의: 프로그래밍 언어 ≠ 프로그래밍 패러다임

자바나 코틀린 같은 객체지향 언어를 사용해도 **함수 위주의 사고방식**으로 프로그램을 만들면 여전히 절차지향 패러다임입니다.

## ● 문제가 있는 절차지향적 코드

```
class RestaurantChain {
    private List<Store> stores;

    // ⚡ 모든 비즈니스 로직이 한 곳에 집중
    public long calculateRevenue() {
        long revenue = 0;
        for (Store store : stores) {
            for (Order order : store.getOrders()) {
```

```

        for (Food food : order.getFoods()) {
            revenue += food.getPrice();
        }
    }
}
return revenue;
}

// ✨ 복잡한 중첩 루프와 계산 로직
public long calculateProfit() {
    long cost = 0;
    for (Store store : stores) {
        for (Order order : store.getOrders()) {
            long orderPrice = 0;
            for (Food food : order.getFoods()) {
                orderPrice += food.getPrice();
                cost += food.getOriginCost();
            }
            cost += orderPrice * order.getTransactionFeePercent();
        }
        cost += store.getRentalFee();
    }
    return calculateRevenue() - cost;
}

@Getter
class Store {
    private List<Order> orders;
    private long rentalFee;
    // ✨ 단순 데이터 컨테이너 역할만 수행
}

@Getter
class Order {
    private List<Food> foods;
    private double transactionFeePercent = 0.03;
    // ✨ 데이터만 저장, 행동은 없음
}

@Getter
class Food {
    private long price;
    private long originCost;
    // ✨ getter만 존재하는 빈약한 객체
}

```

## 🔍 문제점 분석

문제점	설명
데이터 중심 설계	객체가 단순한 데이터 컨테이너 역할만 수행

문제점	설명
책임의 부재	객체에 아무런 책임이 할당되지 않음
낮은 응집도	관련 데이터와 로직이 분리되어 있음
높은 결합도	한 클래스가 다른 클래스의 내부 구조를 너무 많이 알고 있음
테스트 어려움	복잡한 로직으로 인한 단위 테스트 곤란

## ✅ 4. 개선된 객체지향 구조

### 🎯 핵심 원칙

개념	설명
책임 (Responsibility)	각 객체가 담당해야 할 역할과 의무
역할 (Role)	인터페이스를 통한 추상화된 책임
협력 (Collaboration)	객체들 간의 메시지 전달을 통한 문제 해결

### ♥️ 개선된 코드

#### RestaurantChain 클래스

```
@Getter
class RestaurantChain {
    private List<Store> stores;

    public RestaurantChain(List<Store> stores) {
        this.stores = stores;
    }

    // ✅ 각 Store에게 책임을 위임
    public long calculateRevenue() {
        return stores.stream()
            .mapToLong(Store::calculateRevenue)
            .sum();
    }

    public long calculateProfit() {
        return stores.stream()
            .mapToLong(Store::calculateProfit)
            .sum();
    }
}
```

#### Store 클래스

```

@Getter
class Store {
    private List<Order> orders;
    private long rentalFee;

    public Store(List<Order> orders, long rentalFee) {
        this.orders = orders;
        this.rentalFee = rentalFee;
    }

    // ✅ Store의 매출 계산 책임
    public long calculateRevenue() {
        return orders.stream()
            .mapToLong(Order::calculateTotalPrice)
            .sum();
    }

    // ✅ Store의 수익 계산 책임
    public long calculateProfit() {
        long revenue = calculateRevenue();
        long cost = orders.stream()
            .mapToLong(Order::calculateCost)
            .sum();
        cost += rentalFee;
        return revenue - cost;
    }
}

```

## Order 클래스

```

@Getter
class Order {
    private List<Food> foods;
    private double transactionFeePercent = 0.03;

    public Order(List<Food> foods) {
        this.foods = foods;
    }

    // ✅ 주문 총액 계산 책임
    public long calculateTotalPrice() {
        return foods.stream()
            .mapToLong(Food::getPrice)
            .sum();
    }

    // ✅ 주문 비용 계산 책임
    public long calculateCost() {
        long foodCost = foods.stream()
            .mapToLong(Food::getOriginCost)
            .sum();
    }
}

```

```

        long orderPrice = calculateTotalPrice();
        long transactionFee = (long) (orderPrice * transactionFeePercent);
        return foodCost + transactionFee;
    }
}

```

## Food 클래스

```

@Getter
class Food {
    private long price;
    private long originCost;

    public Food(long price, long originCost) {
        this.price = price;
        this.originCost = originCost;
    }
}

```

## 개선 효과 비교

측면	변경 전	변경 후
책임 할당	RestaurantChain이 모든 계산 담당	각 클래스가 자신의 역할에 맞는 계산 담당
코드 가독성	내부 루프로 복잡함	Stream API로 함수형 스타일 향상
설계 철학	데이터 중심 설계	책임 중심 설계, 메시지 전달 구조
테스트 용이성	테스트 어려움	각 객체별 독립 테스트 가능
유지보수성	변경 시 여러 곳 수정 필요	단일 책임으로 변경 범위 최소화

## 5. 객체지향의 핵심: 역할과 책임

### 5.1 책임의 진화

#### 절차지향에서의 책임

```

// ✅ 함수도 책임을 가질 수 있음
int abs(int a) {
    return a < 0 ? -a : a;
}
// 절댓값을 반환하는 책임을 지니고 있음

```

#### 객체지향에서의 책임

```
// ✅ 객체가 책임을 담당
class Calculator {
    public int abs(int a) {
        return a < 0 ? -a : a;
    }
}
```

## 5.2 🏰 역할의 중요성

🎯 핵심: 객체지향에서는 책임을 함수가 아닌 **객체에 할당**하는 것이 중요

### C언어의 한계

```
// C언어: 구조체 + 함수 포인터로 객체지향 흉내 가능
typedef struct {
    int (*calculate)(int, int);
} Calculator;

// ❌ 하지만 추상화와 다형성을 완전히 지원하지 못함
```

### 자바의 역할 기반 설계

```
// ✅ 인터페이스를 통한 역할 정의
interface PaymentProcessor {
    void processPayment(long amount);
}

// ✅ 구체적인 구현체들
class CreditCardProcessor implements PaymentProcessor {
    public void processPayment(long amount) { /* 신용카드 결제 */ }
}

class BankTransferProcessor implements PaymentProcessor {
    public void processPayment(long amount) { /* 계좌이체 결제 */ }
}

// ✅ 클라이언트는 구체적인 구현을 몰라도 됨
class OrderService {
    private PaymentProcessor processor;

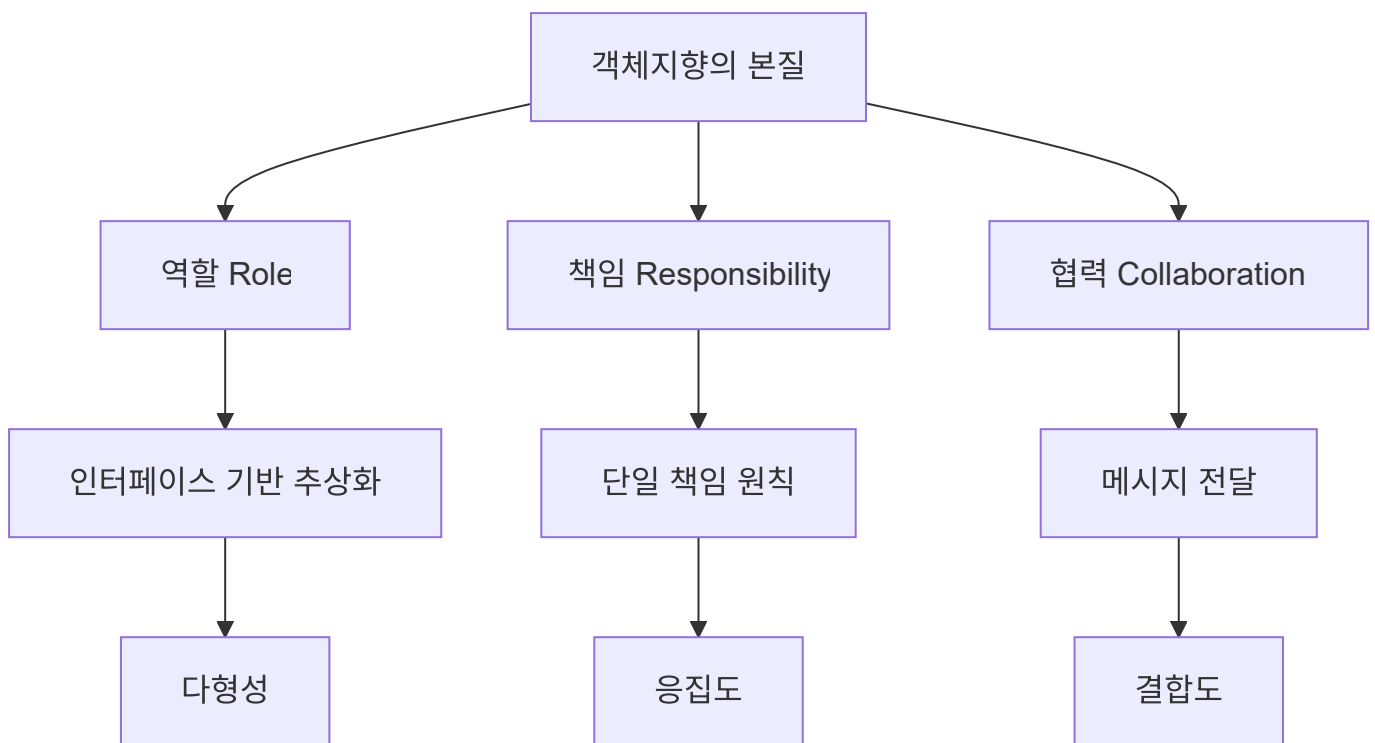
    public void processOrder(Order order) {
        processor.processPayment(order.getTotalAmount()); // 다형성 활용
    }
}
```

### 📋 역할 기반 설계의 장점

장점	설명
유연성	구현체를 쉽게 교체할 수 있음
확장성	새로운 구현체 추가가 용이함
테스트 용이성	Mock 객체를 이용한 테스트 가능
코드 재사용	인터페이스 기반으로 재사용성 향상

## 🎯 6. 객체지향의 본질

### 6.1 🔑 핵심 원리






### 6.2 📖 문법 vs 본질

문법적 특징	본질적 목적
추상화	역할을 명확히 정의하기 위해
다형성	역할 기반 협력을 위해
상속	책임을 재사용과 확장을 위해
캡슐화	책임을 경계를 명확히 하기 위해




### 6.3 ⚖️ 패러다임별 장단점

#### 절차지향의 장점







-  직관적이고 이해하기 쉬움
-  작은 프로젝트에서 빠른 개발 가능
-  메모리 사용량이 적음




## 절차지향의 단점

-  코드 재사용성 낮음
-  유지보수 어려움
-  데이터와 기능의 분리로 인한 응집도 저하

## 객체지향의 장점

-  높은 재사용성과 확장성
-  유지보수 용이성
-  실세계 모델링에 적합
-  팀 단위 개발에 유리

## 객체지향의 단점

-  초기 학습 곡선이 높음
-  과도한 추상화로 인한 복잡성 증가 가능
-  성능 오버헤드 존재


---

## 7. TDA (Tell, Don't Ask) 원칙

### 7.1 TDA 원칙이란?

"물지 말고 시켜라" - 객체의 데이터를 꺼내서 직접 처리하지 말고, 객체에게 일을 시키자

### 7.2 TDA 원칙 위반 사례

```
//  나쁜 예: 데이터를 꺼내서 직접 처리
class OrderService {
    public void processOrder(Order order) {
        // 주문 상태를 직접 확인하고 조작
        if (order.getStatus() == OrderStatus.PENDING) {
            order.setStatus(OrderStatus.PROCESSING);

            // 결제 금액을 직접 계산
            long totalAmount = 0;
            for (OrderItem item : order.getItems()) {
                totalAmount += item.getPrice() * item.getQuantity();
            }

            // 결제 처리
            paymentService.charge(totalAmount);
        }
    }
}
```

```
}  
}
```

## 7.3 TDA 원칙 준수 사례

```
//  좋은 예: 객체에게 책임을 위임  
class OrderService {  
    public void processOrder(Order order) {  
        // 주문 객체에게 처리를 요청  
        if (order.canProcess()) {  
            order.startProcessing();  
            long totalAmount = order.calculateTotalAmount();  
            paymentService.charge(totalAmount);  
        }  
    }  
}  
  
class Order {  
    private OrderStatus status;  
    private List<OrderItem> items;  
  
    //  객체가 자신의 상태를 관리  
    public boolean canProcess() {  
        return status == OrderStatus.PENDING;  
    }  
  
    public void startProcessing() {  
        this.status = OrderStatus.PROCESSING;  
    }  
  
    //  객체가 자신의 데이터로 계산 수행  
    public long calculateTotalAmount() {  
        return items.stream()  
            .mapToLong(item -> item.getPrice() * item.getQuantity())  
            .sum();  
    }  
}
```

## 7.4 TDA 원칙의 효과

측면	TDA 위반	TDA 준수
캡슐화	내부 구조 노출	내부 구조 보호
응집도	낮음 (로직 분산)	높음 (로직 집중)
결합도	높음 (강한 의존성)	낮음 (약한 의존성)
테스트	복잡함	단순함
유지보수	어려움	쉬움

## 8. 실무 적용 가이드

### 8.1 객체지향 설계 체크리스트

#### 설계 단계

- ☐ 단일 책임: 각 클래스가 하나의 책임만 가지는가?
- ☐ 역할 정의: 인터페이스로 역할을 명확히 정의했는가?
- ☐ 책임 분산: 비즈니스 로직이 적절히 분산되어 있는가?

#### 구현 단계

- ☐ TDA 원칙: getter 남용을 피하고 객체에게 책임을 위임했는가?
- ☐ 캡슐화: 내부 구현이 외부에 노출되지 않는가?
- ☐ 메시지 전달: 객체 간 협력이 메시지 전달로 이루어지는가?

#### 검증 단계

- ☐ 테스트 용이성: 각 객체를 독립적으로 테스트할 수 있는가?
- ☐ 확장 가능성: 새로운 요구사항을 쉽게 추가할 수 있는가?
- ☐ 가독성: 각 객체의 역할과 책임이 명확한가?

### 8.2 흔한 안티패턴

#### 1. God Object (신 객체)

```
// ❌ 모든 것을 처리하는 거대한 클래스
class SystemManager {
    public void handleEverything() { /* 수백 줄의 코드 */ }
}
```

#### 2. Anemic Domain Model (빈혈 도메인 모델)

```
// ❌ 데이터만 있고 행동이 없는 객체
class User {
    private String name;
    private String email;
    // getter/setter만 존재
}
```

#### 3. Feature Envy (기능 욕심)

```
// ❌ 다른 객체의 데이터에 지나치게 의존
class OrderProcessor {
    public void process(Order order) {
        // order의 내부 데이터를 과도하게 사용
    }
}
```

```

    customer.getName(); customer.getEmail();
    customer.getAddress().getZipCode();
}
}

```

## 8.3 💡 실무 팁

### 리팩토링 우선순위

1. 🇮🇹 데이터 중심 → 행동 중심: getter 제거하고 메서드 추가
2. 🎯 책임 분산: 거대한 클래스를 작은 클래스들로 분할
3. 🔗 인터페이스 도입: 구체 클래스 의존성을 인터페이스로 교체
4. 🧪 테스트 추가: 각 객체의 책임을 검증하는 테스트 작성

### 팀 협업 방법

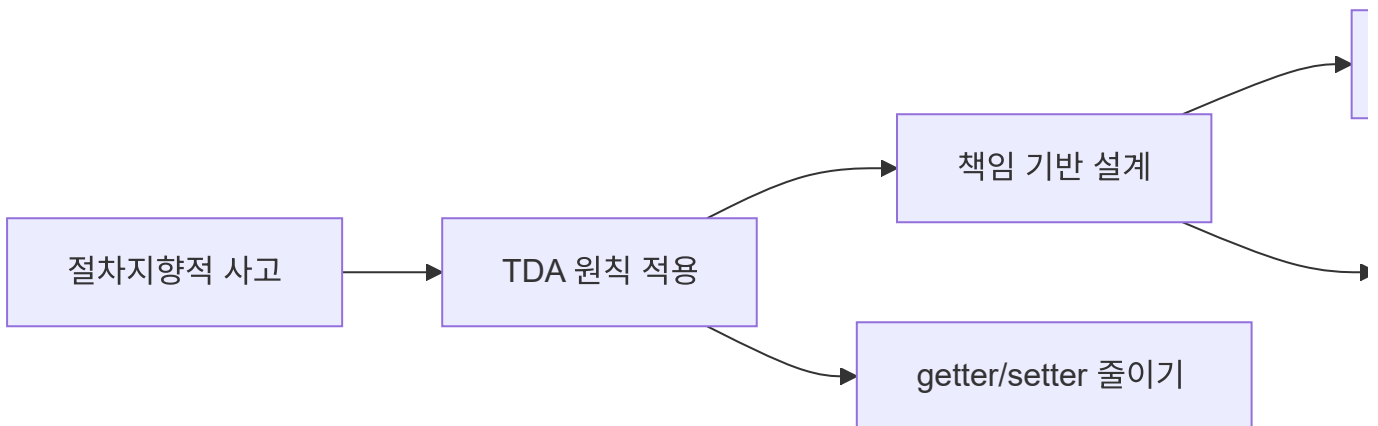
- 📄 코드 리뷰: TDA 원칙 준수 여부 확인
- 🎓 페어 프로그래밍: 객체지향 설계 노하우 공유
- 📖 스터디: 객체지향 설계 패턴 학습
- 🔄 지속적 리팩토링: 작은 단위로 꾸준히 개선

## 🎉 결론

### 🔑 핵심 메시지

"객체를 데이터 덩어리로 보지 말고 객체에게 책임을 위임하세요"

### 📈 성장 로드맵



### 🎯 실천 방안

1. 🏃 작은 것부터 시작: TDA 원칙을 한 클래스씩 적용

2. 🧪 **테스트 주도**: 테스트 코드로 설계 품질 검증
3. 🔄 **점진적 개선**: 기존 코드를 단계적으로 리팩토링
4. 👥 **팀 학습**: 객체지향 원칙을 팀 전체가 공유
5. 📖 **지속 학습**: 디자인 패턴과 원칙을 꾸준히 학습

## 💪 최종 목표

- 🎨 **아름다운 코드**: 읽기 쉽고 이해하기 쉬운 코드
- 🛠️ **유지보수성**: 변경에 강한 유연한 설계
- 🚀 **생산성**: 빠른 기능 개발과 안정적인 운영
- 😊 **개발 만족도**: 즐겁고 보람 있는 개발 경험

---

🌟 **Remember**: 객체지향은 문법이 아닌 사고방식입니다. 역할, 책임, 협력을 중심으로 생각하며 TDA 원칙을 실천해보세요!