



It's Your Life

with





# 백엔드에서의

# OOP 의 의미



**OOP는 역할과 구현으로  
간단하게 서비스를  
변경 및 교체가 가능하다**

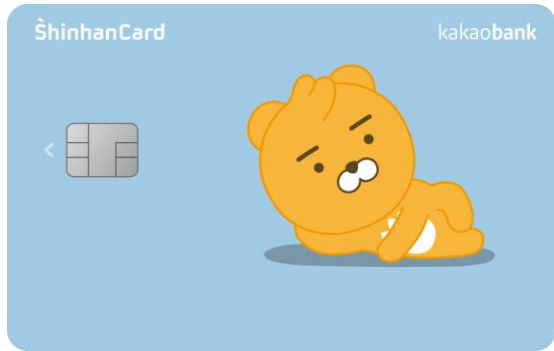


여러분이 결제 서비스를 만든다  
고 해봅시다!



하지만 당연하게도  
결제에는 다양한 서비스를  
지원해야 합니다









# 결제 라는 '역할'



# 다양한 결제 수단이라는 '구현'





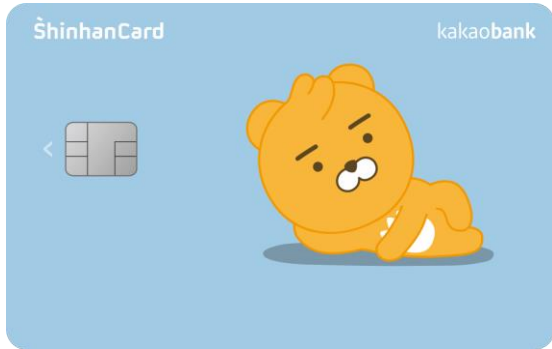


여기서  
다형성이 떠오르시나요!?

# 결제라는 역할은!? 인터페이스를 사용



# 실제 구현은!? 클래스와 인스턴스를 사용



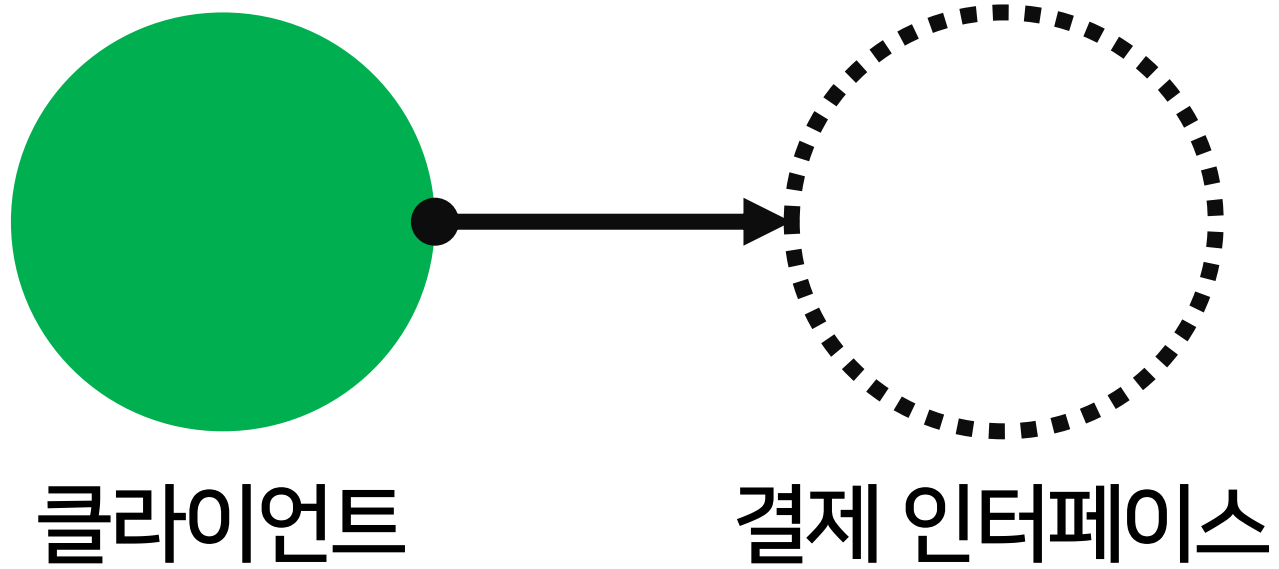
카드 결제  
기능 인스턴스



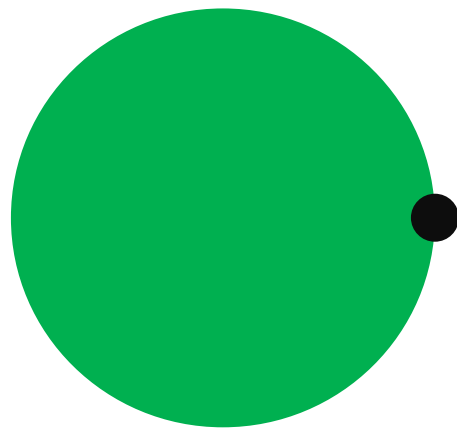
카카오 결제  
기능 인스턴스



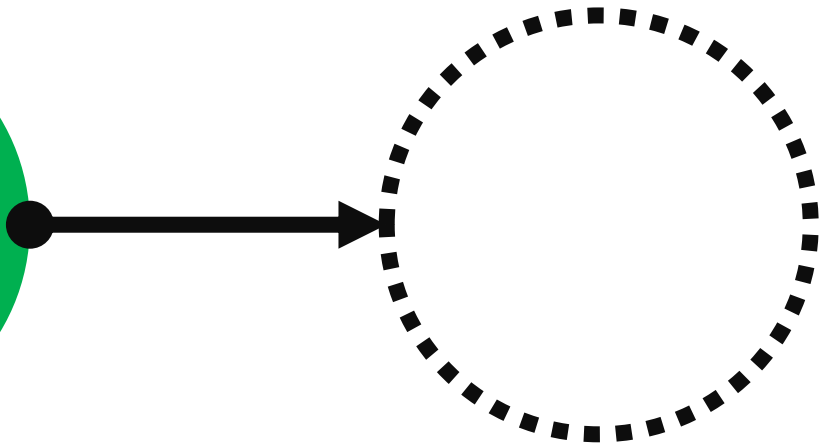
네이버 결제  
기능 인스턴스



결제하기()  
수수료정산()  
결과화면표시()



클라이언트



결제 인터페이스



카드 결제  
인스턴스



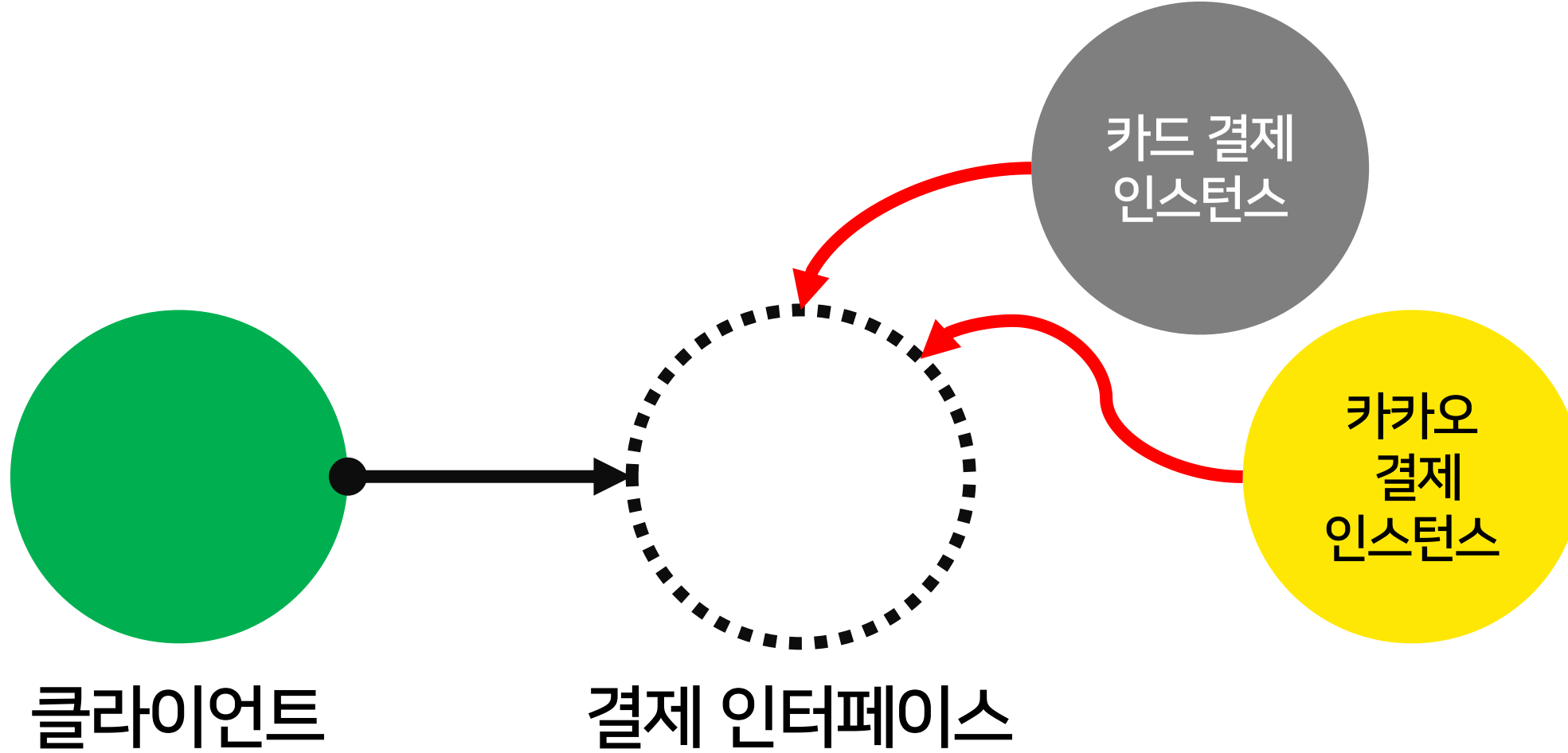
카카오  
결제  
인스턴스

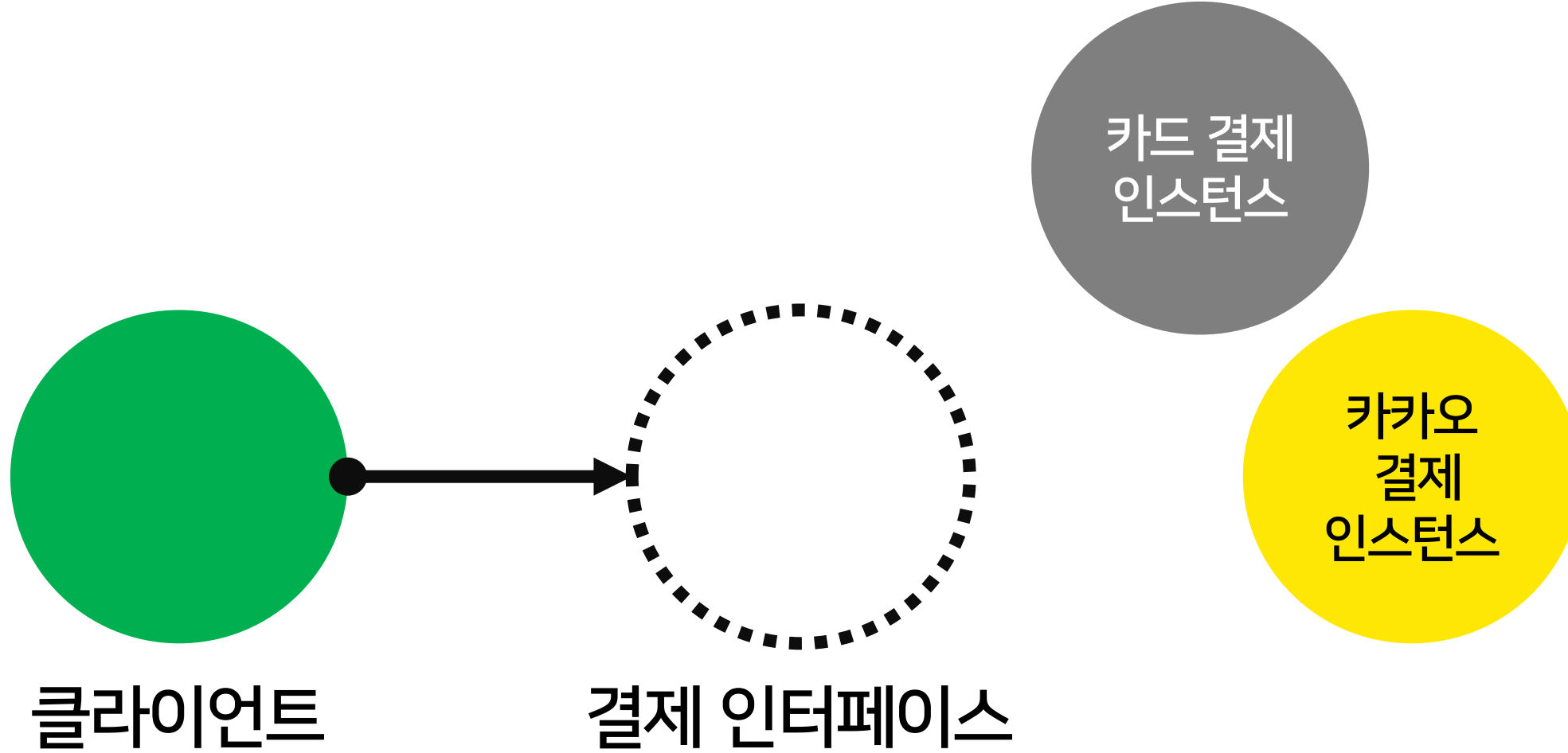
결제하기()  
수수료정산()  
결과화면표시()



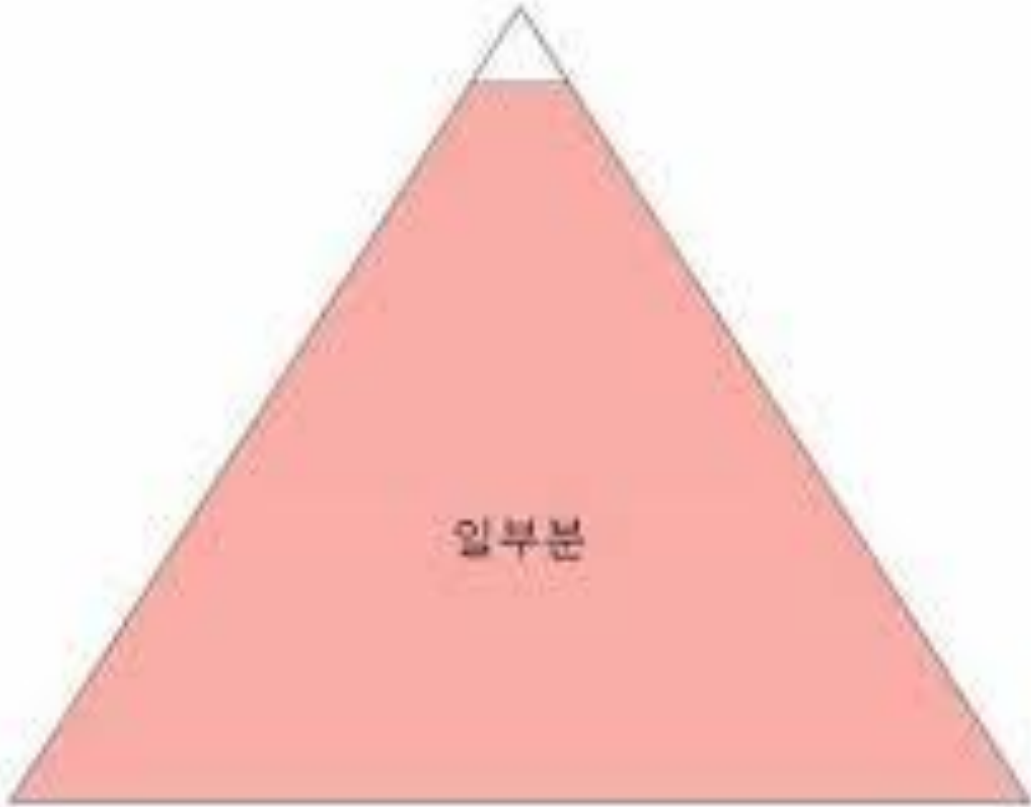
결제하기()  
수수료정산()  
결과화면표시()





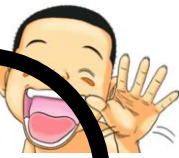


일부만 보고 전체를 매도하지 마세요!



OOP 와 다형성을 통해서  
결제 시스템 전체를  
변경할 필요 X

결제 시스템은 그대로 두고  
인스턴스만 변경!



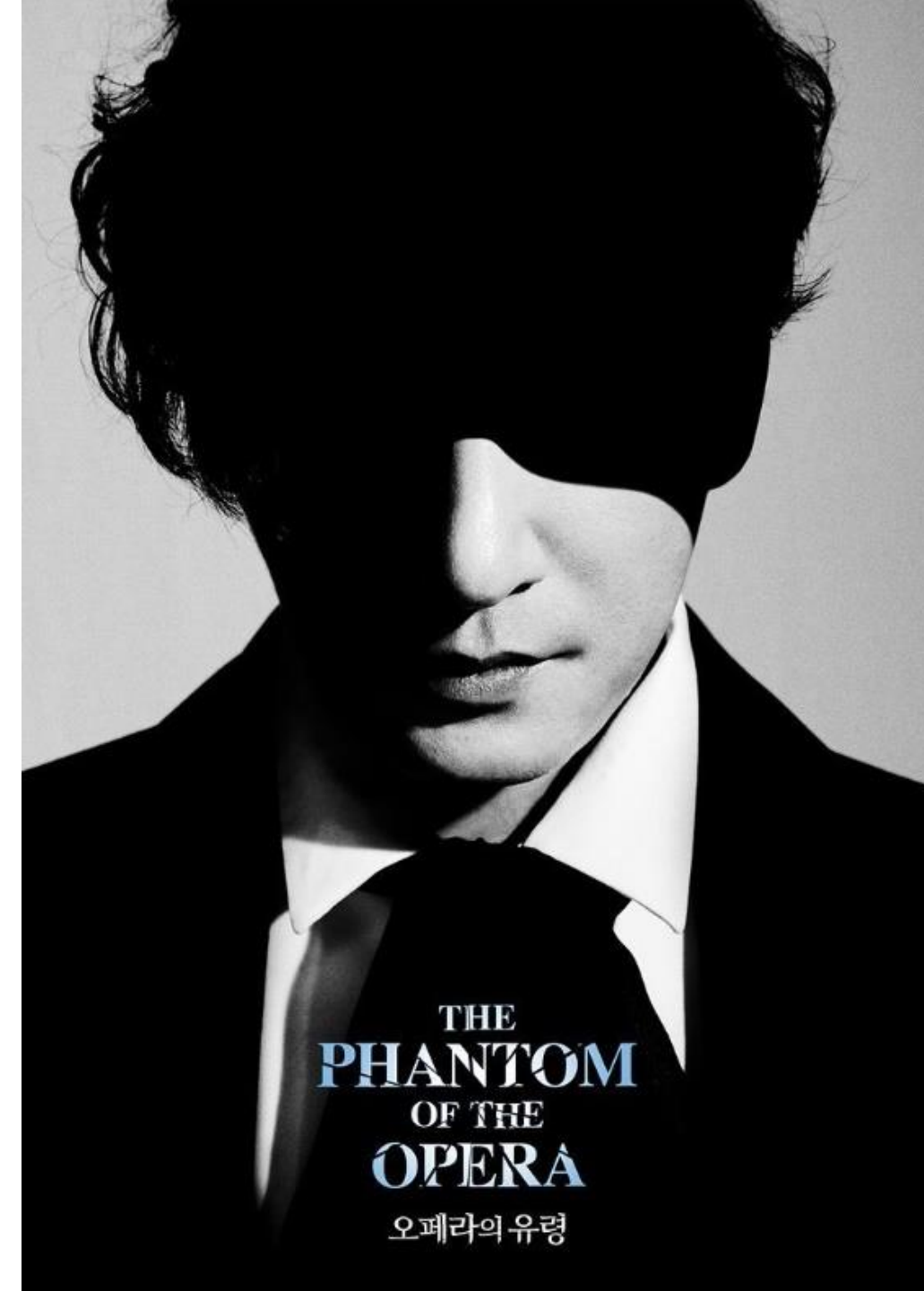


조승우라는 배우에 맞춰서  
오페라를 설계한다면!?

조승우씨가 아프면!?  
조승우씨가 안 한다고 하면?

그리고 조승우씨에게 맞춰진  
오페라 자체의 내용도  
쉽게 변경이 가능할까요?





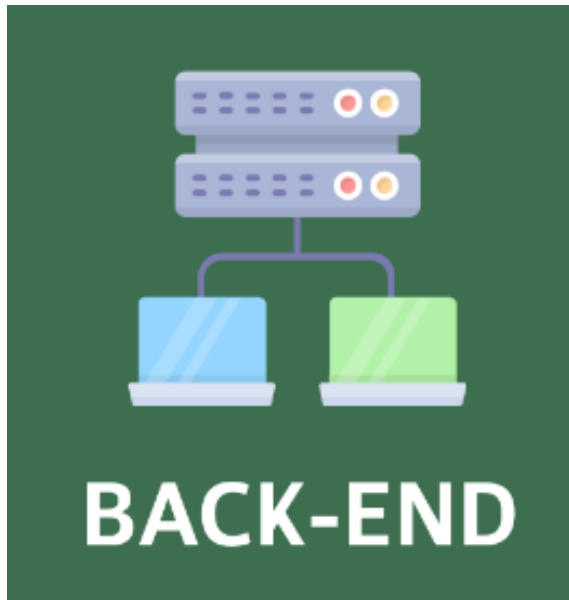
유령이라는 역할에 집중해서  
오페라를 설계한다면!?



유령 역할을 잘할 수 있는 배우만  
잘 섭외하면 됩니다!!

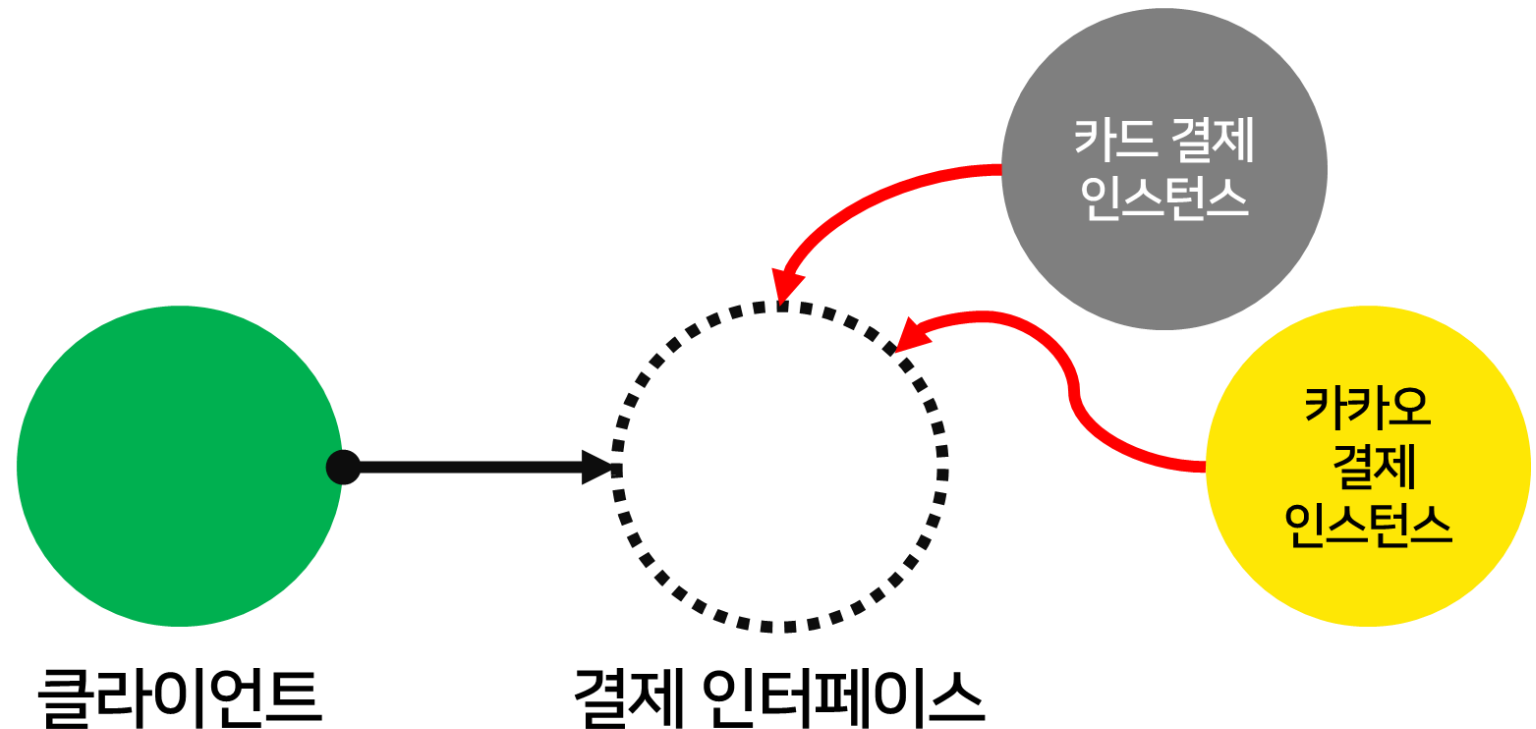
어떤 배우가 아파도? → 대타 가능  
극 자체의 내용도? → 쉽게 변경 가능





이걸 백엔드와 서비스 세상에 접목시켜 보면!?

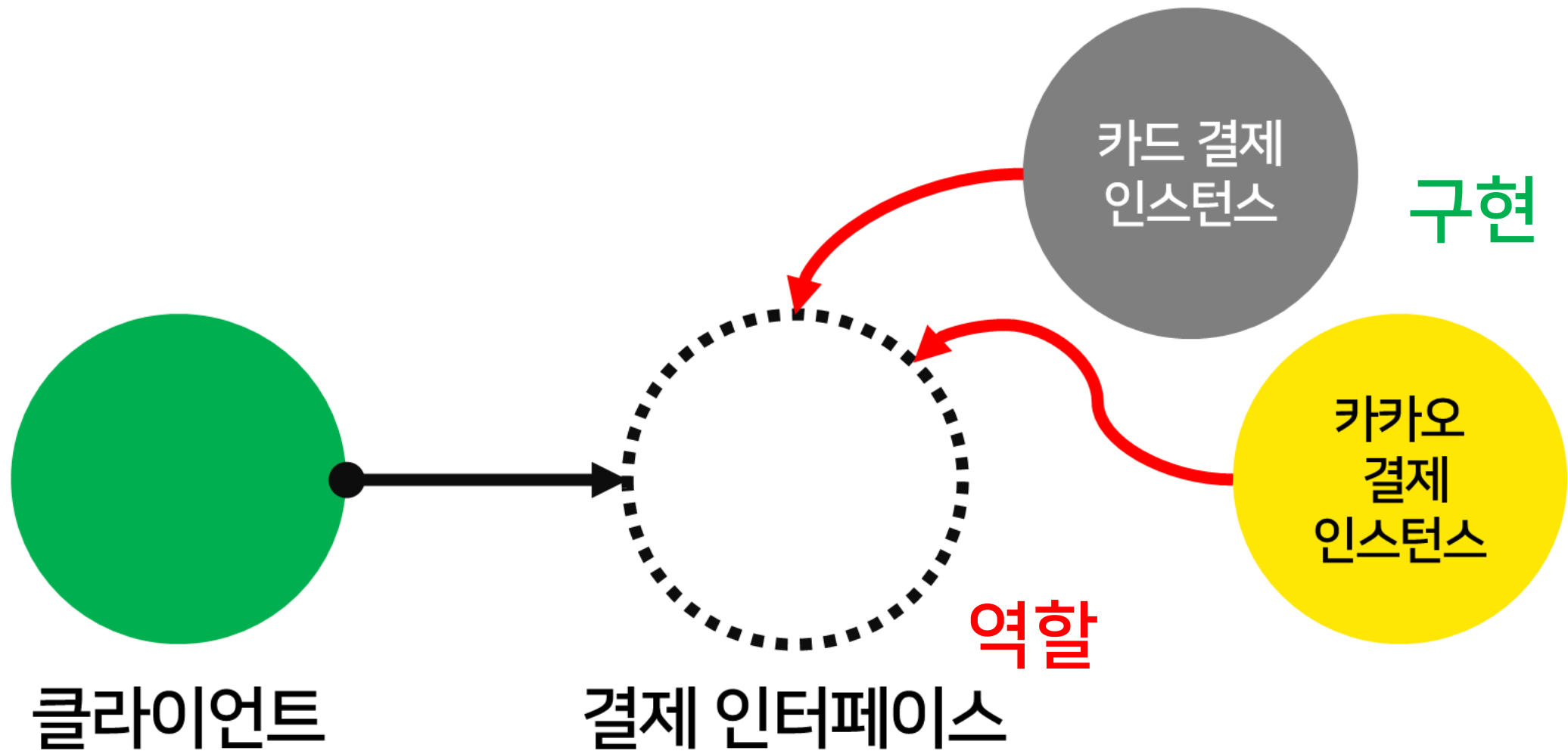
왜, OOP 와 다형성을 이렇게 강조하는지  
느낌이 오실 겁니다!!





# 실제 코드로


# 확인!






# 구현 파트

```
public interface Pay {  
    void pay(int amount);  
}
```



페이 서비스의  
구현 파트 인터페이스가 될  
Pay 선언







```
public class KBPAY implements Pay {  
    @Override  
    public void pay(int amount) {  
        System.out.println("KB Pay 시스템과 연결 합니다");  
        System.out.println(amount + "원 결제를 시도합니다");  
        System.out.println("결제 성공!");  
    }  
}
```

Pay 인터페이스의 설계에 맞게  
클래스를 '구현'



```
public class KaKaoPay implements Pay{ 1 usage    new *  
    @Override 1 usage    new *  
    public void pay(int amount) {  
        System.out.println("KaKao Pay 시스템과 연결 합니다");  
        System.out.println(amount + "원 결제를 시도합니다");  
        System.out.println("결제 성공!");  
    }  
}
```

Pay 인터페이스의 설계에 맞게  
클래스를 '구현'



# 역할 파트

```
public class PaySystem { 2 usages new *
    private Pay pay; 2 usages

    public void setPay(Pay pay) { 2 usages new *
        this.pay = pay;
    }

    public void payment(int amount) { 1 usage
        System.out.println("결제를 시작합니다.");
        pay.pay(amount);
    }
}
```

어떤 결제 '구현'이 들어올지 알 수 없으므로 최상위 부모인 Pay 타입으로 멤버 변수를 선언!

'구현'이 결정이 되면 해당 구현을 멤버 변수에 할당하는 set 메서드

```
public class PaySystem { 2 usages    new *  
    private Pay pay; 2 usages  
  
    public void setPay(Pay pay) { 2 usages    new  
        this.pay = pay;  
    }  
  
    public void payment(int amount) { 1 usage    n  
        System.out.println("결제를 시작합니다.");  
        pay.pay(amount);  
    }  
}
```

결제 기능은 어떤 '구현'이 와도  
해당 구현의 결제 구현에 따르면 되므로  
매개변수로 결제 금액만 받아서 전달

실제 결제는 구현 파트에 구현된  
pay() 메서드가 알아서 실행합니다!





# 운영 파트

```
public class PayMain { new *  
    public static void main(String[] args) { new *  
        PaySystem paySystem = new PaySystem();  
        Scanner scanner = new Scanner(System.in);  
        int option;  
        int amount;
```

역할을 담당할 PaySystem 인스턴스

사용자 입력 및 입력을 저장할 변수 선언

```
System.out.print("결제 옵션을 입력하세요 (1. KB 페이 / 2. 카카오페이) : ");  
option = scanner.nextInt();
```

```
System.out.print("결제 금액을 입력하세요 : ");  
amount = scanner.nextInt();
```

```
if (option == 1) {  
    paySystem.setPay(new KBPay());  
} else if (option == 2) {  
    paySystem.setPay(new KaKaoPay());  
}
```

```
paySystem.payment(amount);
```



사용자로 부터 필요한 정보를 입력 받기

사용자 입력에 따라 pay 의 '구현' 설정

결정 된 구현에서 결제 진행!

```
System.out.print("결제 옵션을 입력하세요 (1. KB 페이 / 2. 카카오페이) : ");  
option = scanner.nextInt();
```

```
System.out.print("결제 금액을 입력하세요 : ");  
amount = scanner.nextInt();
```

```
if (option == 1) {  
    paySystem.setPay(new KBPay());  
} else if (option == 2) {  
    paySystem.setPay(new KaKaoPay());  
}
```

```
paySystem.payment(amount);
```

결정 된 구현에서 결제 진행!



결제 옵션을 입력하세요 (1. KB 페이 / 2. 카카오페이) : 1

결제 금액을 입력하세요 : 3000

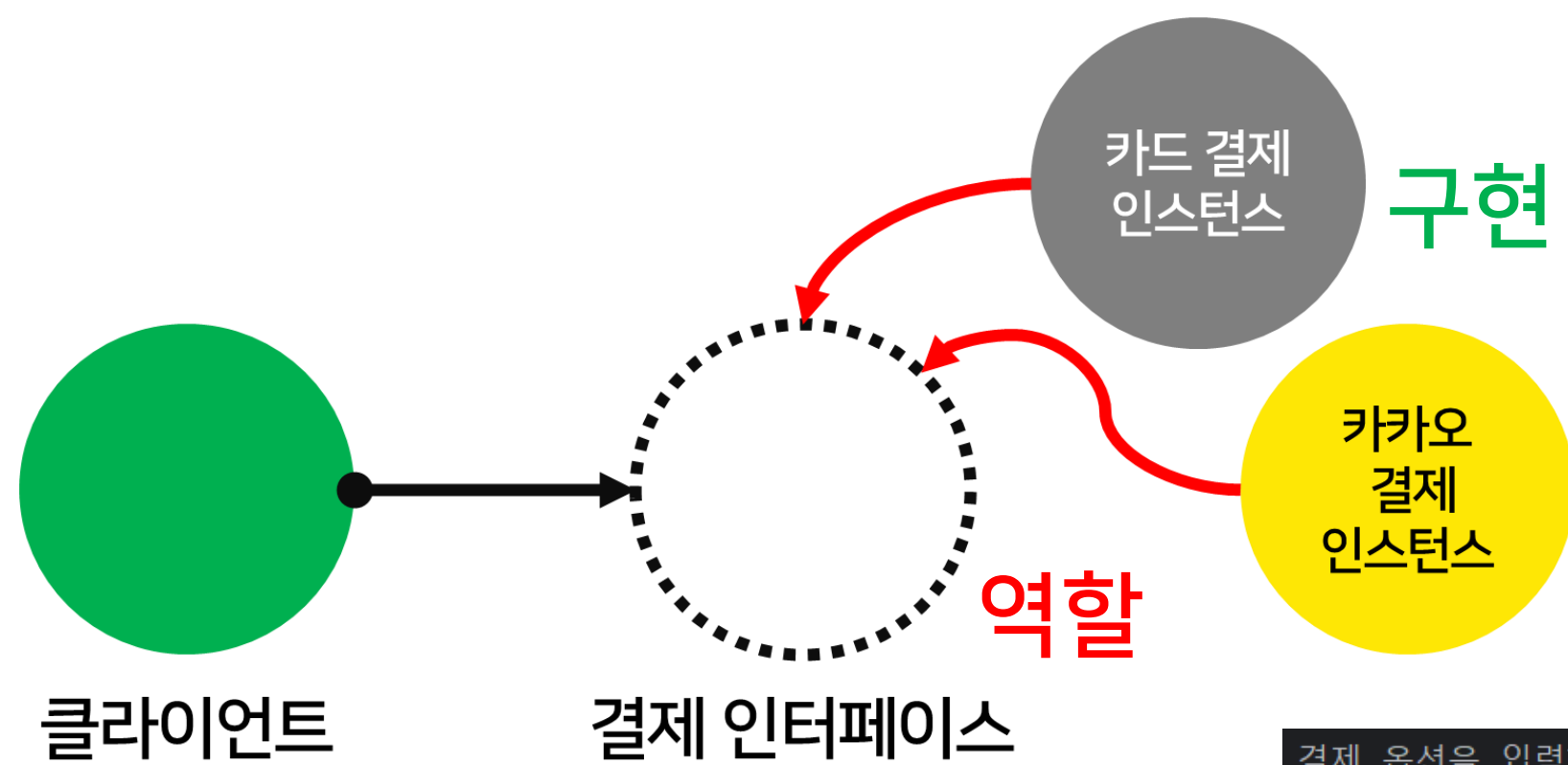
결제를 시작합니다.

KB Pay 시스템과 연결 합니다

3000원 결제를 시도합니다

결제 성공!





```
결제 옵션을 입력하세요 (1. KB 페이 / 2. 카카오페이) : 1
결제 금액을 입력하세요 : 3000
결제를 시작합니다.
KB Pay 시스템과 연결 합니다
3000원 결제를 시도합니다
결제 성공!
```

```
결제 옵션을 입력하세요 (1. KB 페이 / 2. 카카오페이) : 2
결제 금액을 입력하세요 : 1000
결제를 시작합니다.
KaKao Pay 시스템과 연결 합니다
1000원 결제를 시도합니다
결제 성공!
```



# 실습, 네이버페이 추가하기!



- 현재의 코드에 네이버페이 서비스를 추가해 보세요!
- Pay 인터페이스를 구현하여 NaverPay 구현을 만들어 주세요
- Pay 인터페이스의 pay() 메서드를 오버라이드하여, 해당 메서드가 실행되면 “Naver Pay 시스템과 연결합니다” → “xxx 원 결제를 시도합니다” → “결제 성공!” 메시지가 뜨면 됩니다!

# 실습, 네이버페이 추가하기!



- 운영 클래스에서 3번 옵션을 입력하면 네이버 페이지에서 결제가 발생하도록 코드를 수정해 주세요!



코드

refactoring

```
if (option == 1) {  
    paySystem.setPay(new KBPay());  
} else if (option == 2) {  
    paySystem.setPay(new KaKaoPay());  
}
```

구현체를 결정하는 로직을 OOP 에  
맞게 수정해 봅시다!

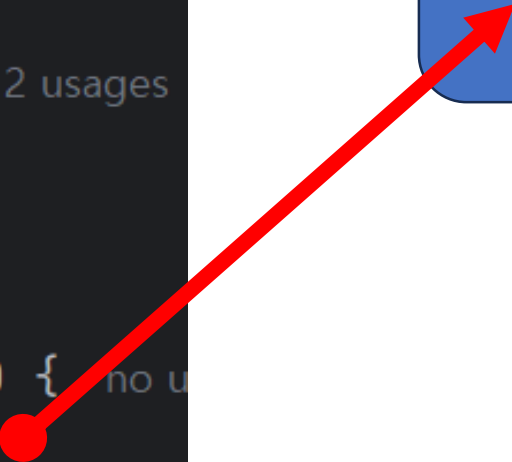
+ 이 코드가 여러 곳에서 쓰인다면!?  
→ 메서드화 시키자!!

```
public class PaySystem { 2 usages new *
    private Pay pay; 2 usages

    public void setPay(Pay pay) { 2 usages
        this.pay = pay;
    }

    public void findPay(int option) { no u
        if (option == 1) {
            this.setPay(new KBPAY());
        } else if (option == 2) {
            this.setPay(new KaKaoPay());
        }
    }
}
```

운영 클래스에서 쓰던 로직을  
PaySystem 의 메서드로 변경!





```
public class PayMain { new *
    public static void main(String[] args) { new *
        PaySystem paySystem = new PaySystem();
        Scanner scanner = new Scanner(System.in);
        int option;
        int amount;

        System.out.print("결제 옵션을 입력하세요 (1. KB 페이 / 2. 카카오페이) : ");
        option = scanner.nextInt();
        System.out.print("결제 금액을 입력하세요 : ");
        amount = scanner.nextInt();

        paySystem.findPay(option);
        paySystem.payment(amount);
    }
}
```

운영 클래스에서는 로직을 제거하고  
인스턴스의 기능만을 사용하는 형태로  
변경!



결제 옵션을 입력하세요 (1. KB 페이 / 2. 카카오페이) : 2

결제 금액을 입력하세요 : 1000

결제를 시작합니다.

KaKao Pay 시스템과 연결 합니다

1000원 결제를 시도합니다

결제 성공!

