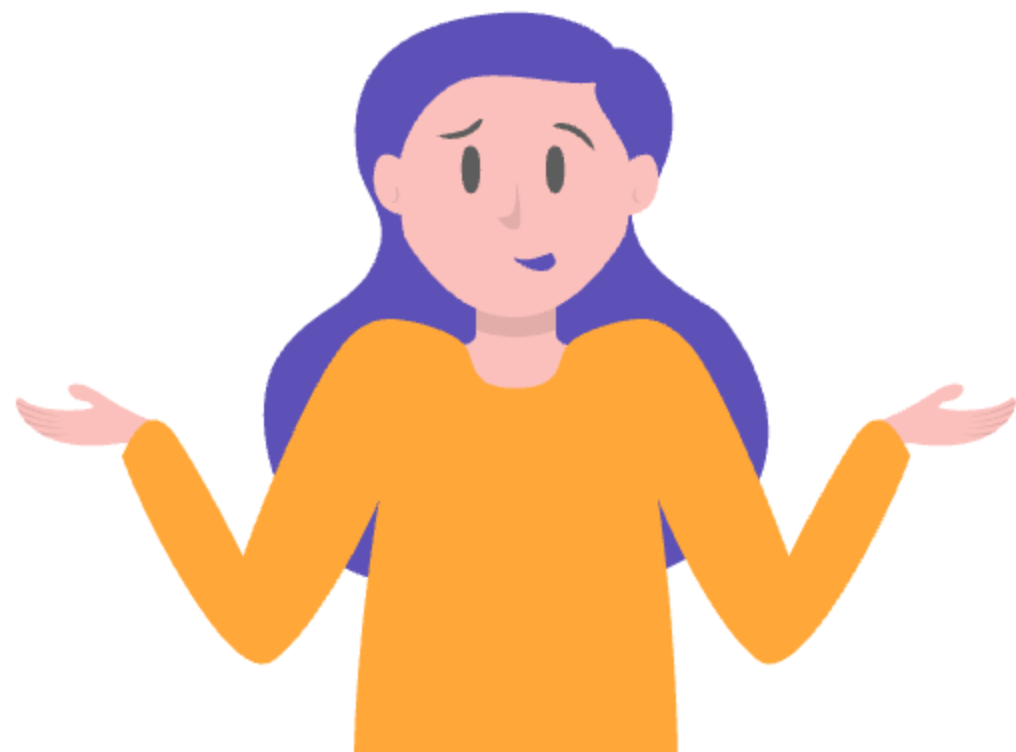


객체는 인터페이스를 사용해 참조하라

인터페이스 참조 왜 필요할까?

?



적합한 인터페이스만 있다면 매개변수, 반환값, 변수, 필드를 전부 인터페이스 타입으로 선언하라



```
public class Example {  
    private final MyInterface myInterface;  
  
    public MyInterface exam(MyInterface myInterface) {  
        final MyInterface impl = myInterface;  
        return impl;  
    }  
}
```

객체지향 5원칙

- ~~SRP (Single Responsibility Principle) 단일 책임 원칙~~
- OCP (Open-Closed Principle) 개방-폐쇄 원칙
- LSP (Liskov Substitution Principle) 리스코프 치환 원칙
- ISP (Interface Segregation Principle) 인터페이스 분리 원칙
- DIP (Dependency Inversion Principle) 의존 역전 원칙

객체지향 5원칙

OCP (Open-Closed Principle) 개방-폐쇄 원칙

- 소프트웨어 요소는 확장에는 열려 있으나 변경에는 닫혀 있어야 한다.

LSP (Liskov Substitution Principle) 리스코프 치환 원칙

- 하위 타입 객체는 상위 타입 객체에서 가능한 행위를 수행할 수 있어야 한다.
(즉, 상위 타입 객체를 하위 타입 객체로 대체하여도 정상적으로 동작해야 한다.)

ISP (Interface Segregation Principle) 인터페이스 분리 원칙

- 특정 클라이언트를 위한 인터페이스 여러 개가 범용 인터페이스 한 개보다 낫다.

DIP (Dependency Inversion Principle) 의존 역전 원칙

- 의존 관계를 맺을 때, 변하기 쉬운 구체적인 것 보다는 변하기 어려운 추상적인 것에 의존해야 한다

인터페이스와 추상 클래스는 존재 목적이 다르다.

추상 클래스: 추상 클래스를 상속받아서 기능을 이용하고, 확장시키는 데 있다.

인터페이스: 함수의 구현을 강제하기 위해서 사용한다.

(구현을 강제함으로써 구현 객체의 같은 동작을 보장할 수 있습니다.)

적합한 인터페이스만 있다면 매개변수, 반환값, 변수, 필드를 전부 인터페이스 타입으로 선언하라



```
Set<Object> set = new LinkedHashSet<>(); YES!!  
LinkedHashSet<Object> linkedSet = new LinkedHashSet<>(); NO!
```

Map 인터페이스를 사용하면 HashMap 도 가능하고,
성능을 위해 EnumMap 혹은 순서를 위해 LinkedHashMap 등을 유연하게 사용할 수 있다.

적합한 인터페이스만 있다면 매개변수, 반환값, 변수, 필드를 전부 인터페이스 타입으로 선언하라



```
LinkedHashSet<Object> linkedSet = new LinkedHashSet<>();  
linkedSet<Object> = HashSet<>(); 컴파일 Error!!
```

유연하지 못하고 객체지향 원칙을 지키지 못하는 코드

예제

인터페이스 정의

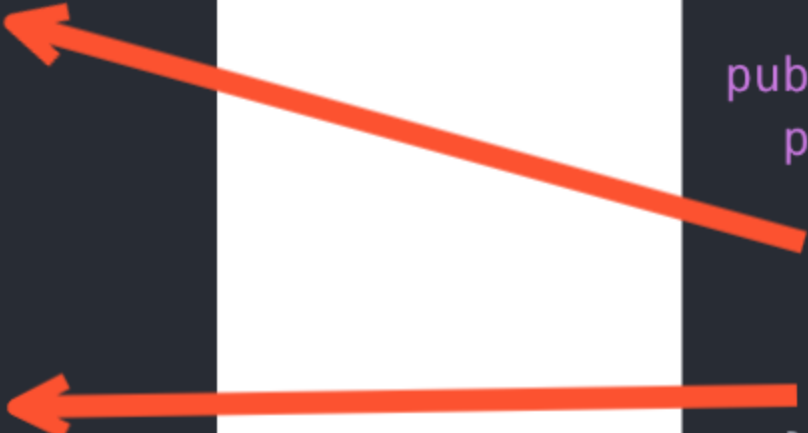
```
public interface Flyable {  
    void fly();  
}
```

구현 클래스 정의

```
class Penguin implements Flyable {  
    @Override  
    public void fly() {  
        System.out.println("펭귄은 못 날아잇");  
    }  
}  
  
class Parrot implements Flyable {  
    @Override  
    public void fly() {  
        System.out.println("앵무새 날아다녀");  
    }  
}
```

사용하는 클래스

```
@Setter  
public class Bird {  
    private Flyable flyable;  
  
    public Bird(Flyable flyable) {  
        this.flyable = flyable;  
    }  
    public void fly() {  
        flyable.fly();  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        final Bird bird = new Bird(new Penguin());  
        bird.fly(); // 펭귄은 못 날아잇  
        bird.setFlyable(new Parrot());  
        bird.fly(); // 앵무새 날아다녀  
    }  
}
```



클래스를 써야 하는 경우는요??



적합한 인터페이스가 없는 경우



```
public record FindUsernameDto(  
    String name,  
    Long userId  
) {  
    public FindUsernameDto {  
        Objects.requireNonNull(userId, "사용자 id가 존재하지 않습니다.");  
        Assert.hasText(name, "사용자 이름을 입력해주세요");  
    }  
}
```

**값 클래스를 여러 가지로 구현될 수 있다고 생각하고 설계하는 일은 거의 없어 final인 경우가 많고
상응하는 인터페이스가 별도로 존재하는 경우는 드물다.**

클래스 기반으로 작성된 프레임워크가 제공하는 객체들

```
public abstract class MyReader {  
    public static String readLine() {  
        try (BufferedReader br = new BufferedReader(new InputStreamReader(System.in))) {  
            return br.readLine();  
        } catch (IOException e) {  
            throw new RuntimeException(e);  
        }  
    }  
}  
  
public static void main(String[] args) {  
    System.out.println(MyReader.readLine());  
}
```

특정 구현 클래스보다는 보통 추상 클래스로 기반 클래스를 사용해 참조하는게 좋다.
(OutputStream 등 java.io 패키지의 여러 클래스)

인터페이스에는 없는 특별한 메서드를 제공하는 클래스

기반 클래스

```
record Person(String name, int age) {  
  
    @Override  
    public String toString() {  
        return name + " (" + age + " years old)";  
    }  
}
```

사용 예시

```
class PriorityQueueWithComparatorExample {  
    public static void main(String[] args) {  
        Comparator<Person> ageComparator = Comparator.comparing(Person::age);  
        PriorityQueue<Person> pq = new PriorityQueue<>(ageComparator);  
  
        pq.add(new Person("Alice", 30));  
        pq.add(new Person("Bob", 25));  
        pq.add(new Person("Charlie", 35));  
        pq.add(new Person("Diana", 28));  
  
        while (!pq.isEmpty()) {  
            System.out.println(pq.poll());  
        }  
    }  
}
```

**클래스 타입을 직접 사용하는 경우 이런 추가 메서드를 꼭 사용해야 하는 경우로
최소화해야 하며 절대 남발하면 안된다.**

결론

- **적합한 인터페이스만 있다면 매개변수, 반환값, 변수, 필드를 전부 인터페이스 타입으로 선언하라**
- **적합한 인터페이스가 없다면 클래스의 계층구조 중 필요한 기능을 만족하는 가장 덜 구체적인 상위 클래스를 타입으로 사용하자.**