이펙티브 자바

Item 21 ~ Item 25

2024/02/13

발표자: 박지원

Item 21, 22

인터페이스는 구현하는 쪽을 생각해 설계하라

인터페이스는 타입을 정의하는 용도로만 사용하라

발표자: 박지원

Item 21 인터페이스는 구현하는 쪽을 생각해 설계하라

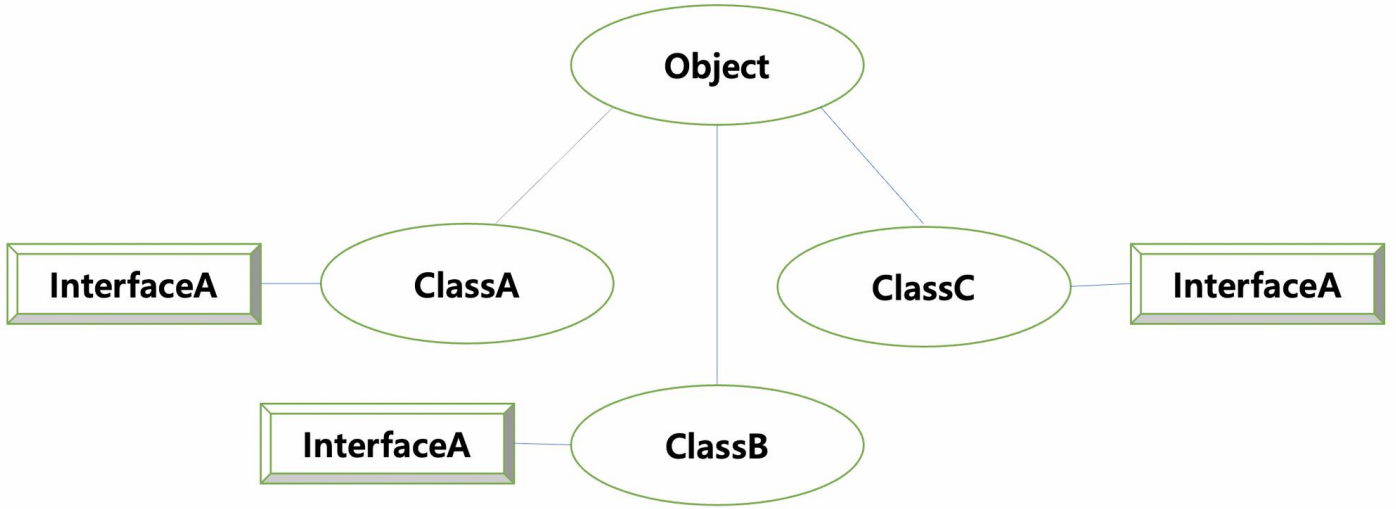
인터페이스 디폴트 메서드: 인터페이스에서 메서드를 구현할 수 있다.

해당 메소드를 오버라이딩 하지 않는 경우, 디폴트 메서드를 그대로 사용한다.

OCP에서 변경에 닫혀 있는 코드를 설계할 수 있다.

```java
public interface Interface {
    // 추상 메서드
    void abstractMethodA();
    void abstractMethodB();
    void abstractMethodC();

    // default 메서드
    default int defaultMethodA(){
        ...
    }
}
```



Item 21 인터페이스는 구현하는 쪽을 생각해 설계하라

자바 8 이후 인터페이스 디폴트 메서드를 추가되었다.

대부분 범용적인 형태로 구현되어 있다.

```
default boolean removeIf(Predicate<? super E> filter) {
        Objects.requireNonNull(filter);
        boolean removed = false;
        final Iterator<E> each = iterator();
        while (each.hasNext()) {
            if (filter.test(each.next())) {
                each.remove();
                removed = true;
            }
        }
        return removed;
    }
```

Item 21 인터페이스는 구현하는 쪽을 생각해 설계하라

디폴드 메서드가 모든 구현 클래스를 대체할 수 없다. (ex SynchronizedCollection)

재정의를 통해 해결해야 한다.

디폴트 메서드를 기존 인터페이스에 추가하는 일은 최대한 피해야 한다.

```
@Override
public boolean removeIf(Predicate<? super E> filter) {
    synchronized (mutex) {return c.removeIf(filter);}
}
```

Item 22 인터페이스는 타입을 정의하는 용도로만 사용하라

인터페이스는 인스턴스를 참조할 수 있는 타입 역할이다.

인터페이스를 구현하는 것은 클래스에게 인스턴스의 기능을 알려주는 것이다.

```
interface Animal {
    public abstract void cry();
}

interface Pet {
    public abstract void play();
}

class Tail {
    // ...
}

class Cat extends Tail implements Animal, Pet { // 클래스와 인터페이스를 동시에 상속

    public void cry() {
        System.out.println("냐옹냐옹!");
    }

    public void play() {
        System.out.println("쥐 잡기 놀이하자~!");
    }
}
```

발표자: 박지원

Item 22 인터페이스는 타입을 정의하는 용도로만 사용하라

인터페이스를 잘못 사용하는 경우

상수는 외부 인터페이스가 아닌 내부 구현이다

클라이언트 코드가 내부 구현에 해당하는 상수에 종속된다.

사용하지도 않을 상수를 모두 가지고 있어야 한다.

해당 상수를 통한 인스턴스 기능을 설명하기 어렵다.

```java
public class Calculations implements Constants {
    public double getReducedPlanckConstant() {
        return PLANCK_CONSTANT / (2 * PI);
    }
}

public interface Constants {
    double PI = 3.14159;
    double PLANCK_CONSTANT = 6.62606896e-34;
}
```

Item 22 인터페이스는 타입을 정의하는 용도로만 사용하라

올바른 상수 정의는 ?

특정 클래스나 인터페이스에 강하게 연관되어 있는 경우 해당 클래스에 자체적으로 추가

그렇지 않은 경우 유틸리티 클래스로 상수를 정의하자

```
public final class Constants {

    private Constants(){}
    public final double PI = 3.14159;
    public final double PLANCK_CONSTANT = 6.62606896e-34;
}
```

Item 22 인터페이스는 타입을 정의하는 용도로만 사용하라

인터페이스에서 static method 사용하는 이유?

Static method은 재정의가 불가함

The idea behind *static* interface methods is to provide a simple mechanism that allows us to **increase the degree of cohesion** of a design by putting together related methods in one single place without having to create an object.

```
public interface Vehicle {

    // regular / default interface methods

    static int getHorsePower(int rpm, int torque) {
        return (rpm * torque) / 5252;
    }
}

//중략
Vehicle.getHorsePower(2500, 480));
```

발표자: 박지원

Item 23, 24, 25

태그 달린 클래스보다는 클래스 계층구조를 활용하라

맴버 클래스는 되도록 static으로 만들라

톱 레벨 클래스는 한 파일에 하나만 담으라

발표자: 박지원

Item 23 태그 달린 클래스보다는 클래스 계층구조를 활용하라

태그: 클래스가 어떠한 타입인지에 대한 정보를 담고 있는 멤버 변수(필드)를 의미

코드들이 많다

메모리를 많이 사용한다

불필요한 초기화를 해야한다.

```
public class Figure {
    enum Shape {RECTANGLE, CIRCLE};

    // 태그 필드 - 현재 모양을 나타낸다.
    final Shape shape;

}
```

Item 23 태그 달린 클래스보다는 클래스 계층구조를 활용하라

1. 루트(root)가 될 추상 클래스를 정의한다

2. 태그에 따라 행동이 달라지던 메서드는 추상 메서드로 구현 그렇지 않다면 일반 메서드로 구현한다

3. 공통으로 사용되는 필드들은 루트에 정의한다

4. 루트 클래스를 확장한 구체 클래스를 의미별로 하나씩 정의한다

발표자: 박지원

Item 24 맴버 클래스는 되도록 static으로 만들라

중첩 클래스: 다른 클래스 안에 정의된 클래스이다

정적 맴버 클래스: 본인과 바깥 클래스의 인스턴스가 독립적이다

비정적 맴버 클래스: 어떤 클래스의 인스턴스를 감싸 마치 다른 클래스의 인스턴스처럼 보이게 하는 뷰

정규화된 this 사용할 수 있다.

```
final class EntrySet extends AbstractSet<Map.Entry<K,V>> {
        public final int size()                 { return size; }
        public final void clear()               { HashMap.this.clear(); }
        public final Iterator<Map.Entry<K,V>> iterator() {
            return new EntryIterator();
        }
        public final boolean contains(Object o) {
            if (!(o instanceof Map.Entry<?, ?> e))
                return false;
            Object key = e.getKey();
            Node<K,V> candidate = getNode(key);
            return candidate != null && candidate.equals(e);
        }
}
```

발표자: 박지원

Item 24 맴버 클래스는 되도록 static으로 만들라

맴버 클래스에서 바깥 인스턴스(this)에 접근할 일이 없다면 무조건 static 을 붙여서 정적 맴버 클래스로 만들자

비정적 맴버 클래스를 사용하는 경우, 숨은 참조가 생겨 메모리 누수 발생한다.

발표자: 박지원

Item 24 맴버 클래스는 되도록 static으로 만들라

익명 클래스: 재사용되지 않는 버려지는 객체, 나중에 람다 함수가 해당 기능을 어느정도 대체한다.

지역 클래스: 지역변수처럼 선언, 정적 메서드를 가질 수 없다.

```
class Animal {
    public String bark() {
        return "동물이 웁니다";
    }
    public String sleep(){ return "동물이 잡니다"; }
}
public class Main {
    public static void main(String[] args) {
        Animal dog = new Animal() {
            @Override
            public String bark() {
                return "개가 짖습니다";
            }
            @Override
            public String sleep(){
                return "개가 잡니다";
            }
        };
        System.out.println(dog.bark());
        System.out.println(dog.sleep());
    }
}
```

발표자: 박지원

Item 25 톱 레벨 클래스는 한 파일에 하나만 담으라

톱레벨 클래스: 소스파일에서 가장 바깥에 존재하는 클래스

중복되는 클래스가 존재하는 경우 컴파일 에러 발생한다

여러 톱레벨 클래스를 한 파일에 담고 싶다면 정적 맴버 클래스를 고려해라

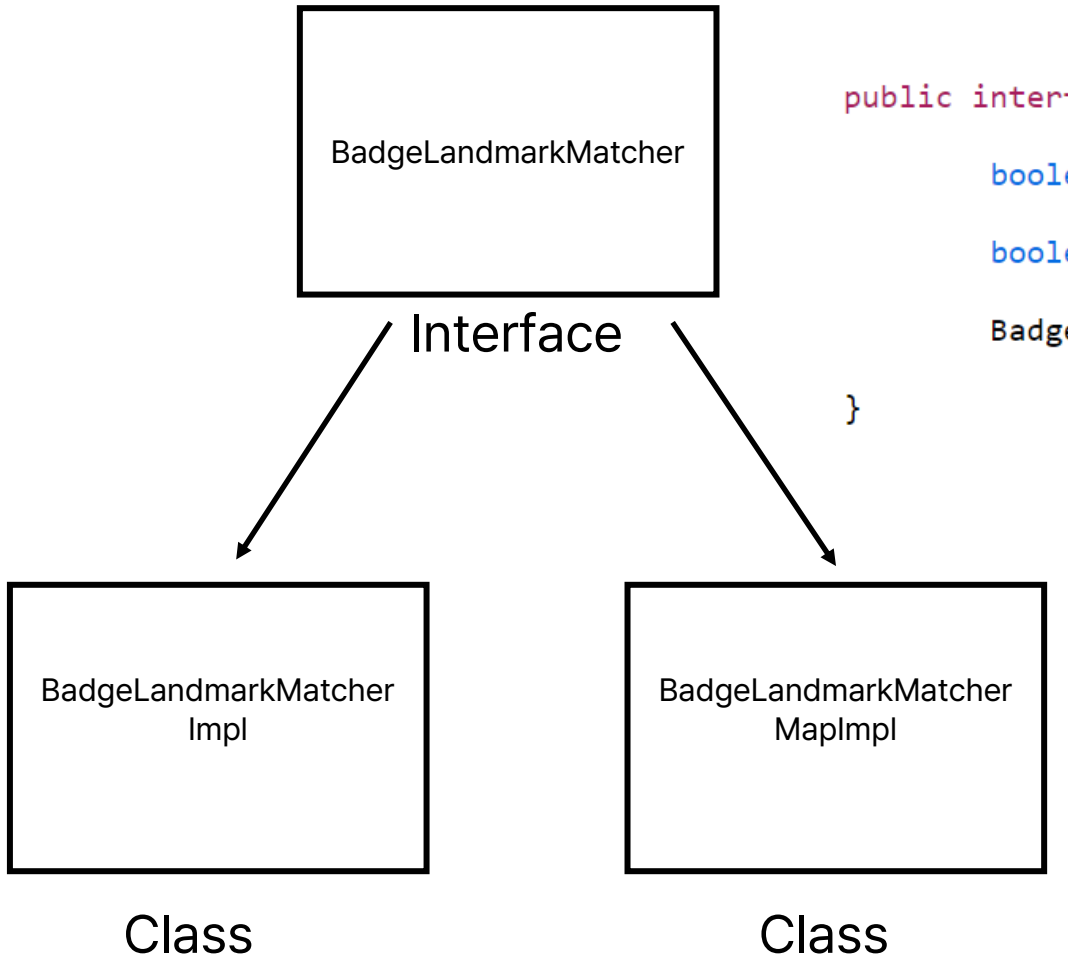
```
class Utensil {
    static final String NAME = "pot";
}
class Dessert{
    static final String NAME = "pie";
}
```

```
class Utensil {
    static final String NAME = "pan";
}
class Dessert{
    static final String NAME = "cake";
}
```

BadgeService

발표자: 박지원

BadgeService refactoring




```java
public interface BadgeLandmarkMatcher {

        boolean haveBadge(Set<BadgeType> userBadgeSet, Landmark landmark);

        boolean canGetBadge(Landmark landmark);

        BadgeType GetMatchBadge(long count,Landmark landmark);

}
```

BadgeService refactoring

```
@Override
public boolean haveBadge(Set<BadgeType> userBadgeSet, Landmark landmark) {

    if (landmark.getName() == LandmarkType.공학관A) {
        if (!userBadgeSet.contains(BadgeType.COLLEGE_OF_ENGINEERING_A)) {
            return false;
        }
        return true;
    }
    else if (landmark.getName() == LandmarkType.공학관B) {
        if (!userBadgeSet.contains(BadgeType.COLLEGE_OF_ENGINEERING_B)) {
           return false;
        }
        return true;
    }
    else if (landmark.getName() == LandmarkType.공학관C) {
        if (!userBadgeSet.contains(BadgeType.COLLEGE_OF_ENGINEERING_C)) {
            return false;
        }
        return true;
    }
    return false;
}
```

BadgeService refactoring

```java
@Override
public boolean canGetBadge(Landmark landmark){
    if (landmark.getName() == LandmarkType.공학관B) {
        LocalTime now = TimeUtil.now();
        if (!(now.isAfter(LocalTime.of(18, 0)) && now.isBefore(LocalTime.of(20, 0)))) {
            return false;
        }
    }
    return true;
}
```

```java
@Override
public BadgeType GetMatchBadge(long count, Landmark landmark) {
    if (landmark.getName() == LandmarkType.공학관A) {
        if (count == 10) {
            return BadgeType.COLLEGE_OF_ENGINEERING_A;
        }
    }
    else if (landmark.getName() == LandmarkType.공학관B) {
        if (count == 10) {
                return BadgeType.COLLEGE_OF_ENGINEERING_B;
        }
    //중략
    return null;
}
```

발표자: 박지원

BadgeService refactoring

```
public static BadgeLandmarkMatcher badgeLandmarkMatcher=new BadgeLandmarkMatcherMapImpl();

public Set<BadgeType> getNewlyRegisteredBadge(User user, Landmark landmark) {
        Set<BadgeType> newlyRegisteredBadge = new HashSet<>();
        Set<BadgeType> userBadgeSet = getUserBadgeSet(user);

        if(!badgeLandmarkMatcher.haveBadge(userBadgeSet,landmark)
                &&badgeLandmarkMatcher.canGetBadge(landmark)){
            long count = adventureRepository.countByUserAndLandmark(user, landmark);
            BadgeType badgeType=badgeLandmarkMatcher.GetMatchBadge(count,landmark);
            if(badgeType!=null)
                newlyRegisteredBadge.add(badgeType);
        }
        return newlyRegisteredBadge;
    }
```

발표자: 박지원

BadgeService refactoring

더 좋은 방법은 없을까???

발표자: 박지원