이펙티브 자바

item18 ~ item 20

24/02/06

아이템 18 상속보다는 컴포지션을 사용하라

상속 vs 컴포지션

> 💡 Tip
> 여기서의 상속은 클래스가 다른 클래스를 확장하는 구현 상속을 의미.
> 인터페이스가 다른 인터페이스를 확장하는 인터페이스 상속과는 무관하다.

상속은 한 클래스를 다른 클래스에서 derive 즉 파생 시킨다.
ex) extend 받은 확장된 클래스가 파생됨

컴포지션은 parts 즉 클래스를 구성하는 부분의 합으로 정의한다
ex) 클래스 필드 내에 private or public 필드로 클래스의 인스턴스를 참조하게 하고
해당 클래스를 구성하는 부분의 합으로 정의됨.클래스의 구성요소로 쓰인다는 뜻에서 composition이라고 한다.

아이템 18 상속보다는 컴포지션을 사용하라

Java에서 상속을 사용하는 경우

> 💡 Tip
> 상위 클래스와 하위 클래스가 " is a " 관계
> A cat is a kind of an animal
> A car is a kind of a vehicle
> 상위 클래스의 specialized version이다
> 즉 원본 버전과 동일하나 좀 더 구체화한, 좀 더 확장한, 좀 더 특별한 버전
> 코드 재사용의 좋은 예이다.
> 상속이 정말 잘 작동하려면 새로운 서브 클래스에서 상속받은 동작중 일부를 변경할 수도 있어야 한다.

상속이 위험한 이유 : 상위 클래스 또는 슈퍼 클래스의 릴리스마다 내부 구현이 달라질 수 있고,
이에 따라 하위 클래스들이 영향을 받고 의도치 않은 동작이 발생할 수 있다.

반면 컴포지션은?
컴포지션을 통해 생성된 클래스와 객체는 느슨하게 결합되어(loosely coupled) 코드를 손상시키지 않고 구성 요소들을 바꿀 수 있다.

아이템 18 상속보다는 컴포지션을 사용하라

Java에서 컴포지션을 사용하는 경우

> 💡 Tip
> one object가 또 다른 object를 **"has" or is part of"** 하는 경우 컴포지션을 사용할 수 있다.
> A car has a battery (a battery is part of a car).
> A person has a heart (a heart is part of a person).
> A house has a living room (a living room is part of a house).
> house 예시가 좋은 예가 되어준다.

아이템 18 상속보다는 컴포지션을 사용하라

컴포지션 사용의 좋은 예시

```
import java.util.HashSet;
import java.util.Set;

public class CharacterCompositionExample {
    static Set<String> set = new HashSet<>();

    public static void main(String ... goodExampleOfComposition) {
        set.add("Homer");
        set.forEach(System.out::println);
    }
```

위 코드 시나리오에서 컴포지션을 사용하면 CharacterCompositionExample는 HashSet을 상속하지 않고
HashSet의 메서드 중 단지 두 가지만 사용하게 된다.
그 결과 단순하고 이해하기 쉽고 유지하기 쉬운 less coupled code 즉 결합도가 낮은 코드가 생성된다.

아이템 18 상속보다는 컴포지션을 사용하라

JDK에서 확인할 수 있는 상속에 대한 좋은 예시

```
class IndexOutOfBoundsException extends RuntimeException { ... }
class ArrayIndexOutOfBoundsException extends IndexOutOfBoundsException { ... }
class FileWriter extends OutputStreamWriter { ... }
class OutputStreamWriter extends Writer { ... }
interface Stream<T> extends BaseStream<T, Stream<T>> { ... }
```

IndexOutOfBoundsException 이 RuntimeException을 상속받은 예시와 같이
하위 클래스는 **상위 클래스**의 specialized version 임을 확인할 수 있다.

아이템 18 상속보다는 컴포지션을 사용하라

래퍼클래스에 대한 사담

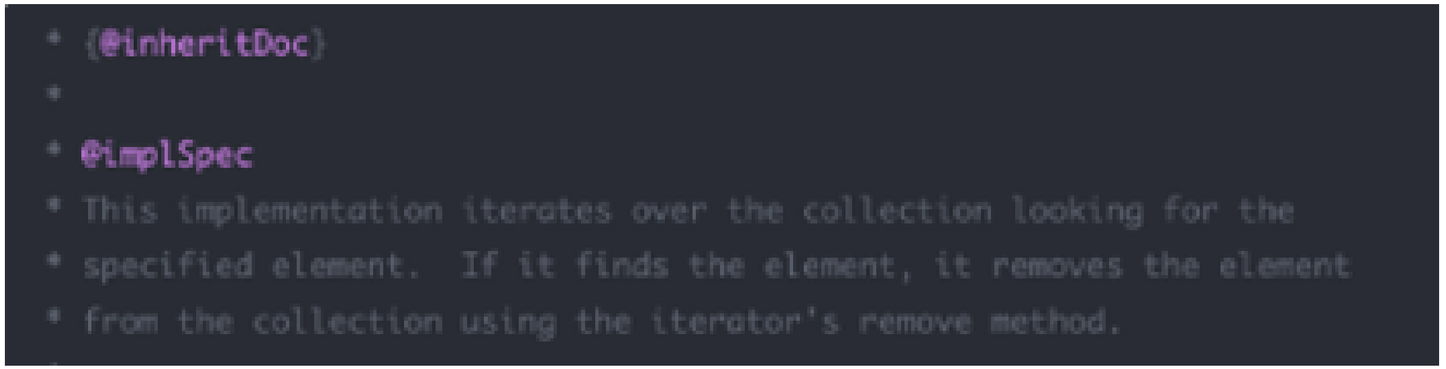
- 래퍼클래스는 단점이 거의 없다.
- 다만, 콜백 프레임워크와 어울리지 않는다.
- 내부 객체가 래퍼의 존재를 몰라 콜백 시 내부 객체를 호출하는 문제가 발생한다.
- 이를 위해 전달 메서드를 인터페이스당 하나씩만 만들어두면, 전달클래스를 아주 손쉽게 구현 가능하다.

아이템 19 상속을 고려해 설계하고 문서화하라. 그러지 않았다면 상속을 금지하라

상속용 클래스는 재정의할 수 있는 메서드들을 내부적으로 어떻게 이용하는지 문서로 남겨야 한다.

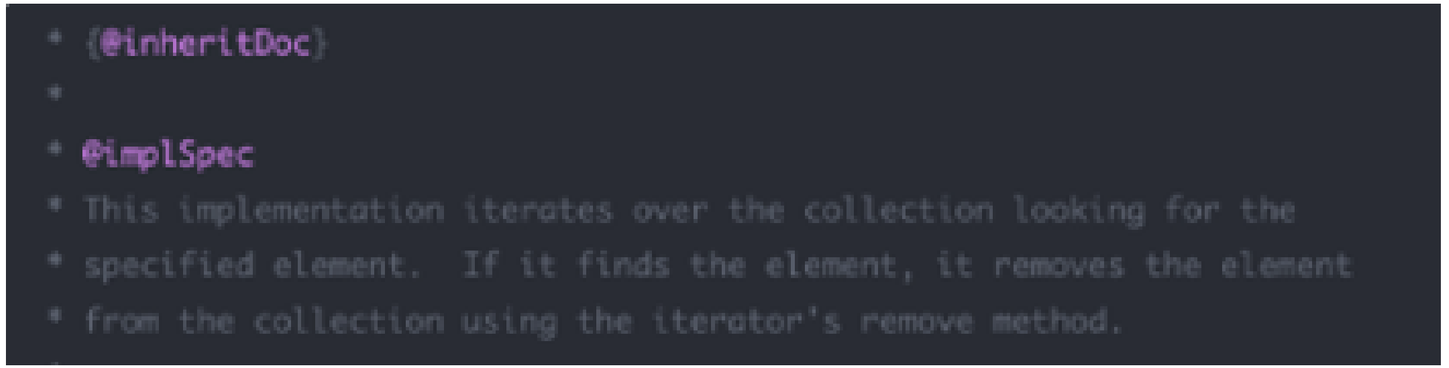
클래스의 API로 공개된 메서드에서 클래스 자신의 또 다른 메서드를 호출할 수도 있다.
또한 그 메서드가 재정의 가능 메서드(protected, public 중 final이 아닌 모든 메서드)라면 그 사실을 호출하는 메서드의 API 설명에 적시해야 한다.

API 문서의 메서드 설명 끝에 종종 Implementation Requirements로 시작하는 절이 있다.
이 절은 그 메서드의 내부 동작 방식을 설명하는 곳이다. 이 절은 메서드 주석에 @implSpec 태그를 붙여주면 자바독 도구가 생성해준다.



아이템 19 상속을 고려해 설계하고 문서화하라. 그러지 않았다면 상속을 금지하라

API 문서의 메서드 설명 끝에 종종 Implementation Requirements로 시작하는 절이 있다.
이 절은 그 메서드의 내부 동작 방식을 설명하는 곳이다. 이 절은 메서드 주석에 @implSpec 태그를 붙여주면 자바독 도구가 생성해준다.



해당 설명에 따르면 iterator 메서드를 재정의하면 remove 메서드에 영향을 준다는 점을 알 수 있다.

> 📑 NOTE
> 메서드간의 인과관계가 잘 표현하여, 상속할때 유의하도록 하는 것이 포인트.

아이템 19 상속을 고려해 설계하고 문서화하라. 그러지 않았다면 상속을 금지하라

protected 사용 시기

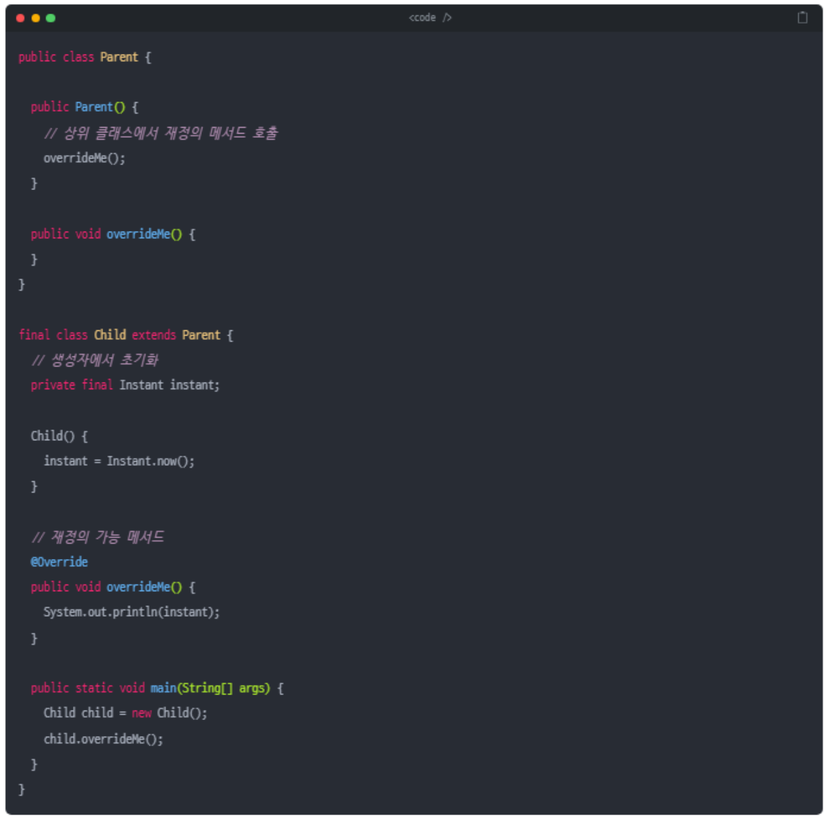
그렇다면 어떤 메서드를 막고, 어떤 메서드를 protected로 노출해야할까?
심사숙고해서 잘 예측한다음, 실제 하위 클래스를 만들어서 시험해보는 것이 최선이자 유일하다.

꼭 필요한 protected 멤버를 놓쳤다면 하위 클래스를 작성할 때 그 빈자리가 확연히 드러난다. 즉하위 클래스를 생성하다보면 성능 상의 이슈등으로 private한 멤버를 protected로 변경하는 경우가 잦다.
반대의 경우로, 하위 클래스를 확장해 나가며 전혀 쓰지 않는 protected 멤버는 private으로 수정하는 것이 올바르다.

아이템 19 상속을 고려해 설계하고 문서화하라. 그러지 않았다면 상속을 금지하라

상속용 클래스의 생성자와 재정의 가능 메서드

상속용 클래스의 생성자는 재정의 가능 메서드를 호출해선 안된다.
상위 클래스의 생성자는 하위 클래스의 생성자보다 먼저 실행된다.



```java
public class Parent {

  public Parent() {
    // 상위 클래스에서 재정의 메서드 호출
    overrideMe();
  }

  public void overrideMe() {
  }
}

final class Child extends Parent {
  // 생성자에서 초기화
  private final Instant instant;

  Child() {
    instant = Instant.now();
  }

  // 재정의 가능 메서드
  @Override
  public void overrideMe() {
    System.out.println(instant);
  }

  public static void main(String[] args) {
    Child child = new Child();
    child.overrideMe();
  }
}
```

아이템 19 상속을 고려해 설계하고 문서화하라. 그러지 않았다면 상속을 금지하라

상속용 클래스의 생성자와 재정의 가능 메서드

상속용 클래스의 생성자는 재정의 가능 메서드를 호출해선 안된다.
상위 클래스의 생성자는 하위 클래스의 생성자보다 먼저 실행된다.

순서는 다음과 같다.

메인에서 자식 클래스 생성 시도 ➡ 부모 클래스 생성자 호출 ➡ 재정의한 메서드 호출➡
자식 클래스 생성자 호출 ➡ instant 변수 초기화

(아마도) 코드를 짠 사람의 의도는 다음과 같다.

메인에서 자식 클래스 생성 시도 ➡ 부모 클래스 생성자 호출 ➡ 자식 클래스 생성자 호출 ➡
instant 변수 초기화 ➡ 재정의한 메서드 호출

```java
public class Parent {

  public Parent() {
    // 상위 클래스에서 재정의 메서드 호출
    overrideMe();
  }

  public void overrideMe() {
  }
}

final class Child extends Parent {
  // 생성자에서 초기화
  private final Instant instant;

  Child() {
    instant = Instant.now();
  }

  // 재정의 가능 메서드
  @Override
  public void overrideMe() {
    System.out.println(instant);
  }

  public static void main(String[] args) {
    Child child = new Child();
    child.overrideMe();
  }
}
```

아이템 19 상속을 고려해 설계하고 문서화하라. 그러지 않았다면 상속을 금지하라

또 다른 예시

- 자바에서는 모든 자식의 생성자들은 super()가 생략되어 있다.
Main 클래스에서 매개변수가 있는 자식 인스턴스를 생성하면,
부모에도 매개변수가 있는 매개변수를 호출할 것 같지만,
기본 생성자를 호출한다.

매개변수가 있는 부모를 호출하고 싶은 경우 직접 명시를 통해 해결할 수 있다.

```
public class Super {

  private String name;
  public Super() {
    System.out.println("부모 - 매개변수가 없는 생성자");
  }

  public Super(String name) {
    System.out.println("부모 - 매개변수가 있는 생성자");
  }
}

class Sub extends Super {

  private String name;

  public Sub(String name) {
    // super(); 생략된 상태 - 매개변수가 없는 생성자 호출
    // super(name); 직접 명시 - 매개변수가 있는 생성자 호출
    System.out.println("자식");
  }

}

class Main {
  public static void main(String[] args) {
    Sub sub = new Sub("test");
  }
}

예시 코드 출처: https://sasca37.tistory.com/259
```

아이템 19 상속을 고려해 설계하고 문서화하라. 그러지 않았다면 상속을 금지하라

정리

- 상속용 상위 클래스를 설계하는 데에는 복잡한 과정이 있다. 따라서 클래스 내부에서 스스로를 어떻게 사용하는지(자기사용 패턴) 모두 문서로 남겨야한다.

- 효율 좋은 하위 클래스를 만들 수 있도록 일부 메서드를 protected로 제공하는 경우도 있다.
  - 상속용으로 설계한 클래스는 배포 전에 반드시 하위 클래스를 만들어 검증한다.
  - 먼저 private으로 제공 후 필요 시 변경해도 좋다.

- 클래스를 확장해야할 명확한 이유가 없다면 상속을 금지하는 편이 낫다. 이 때는 클래스를 final로 선언하거나 생성자 모두를 외부에서 접근할 수 없도록 package-pricate 이나 private 하게 만들고 public 정적 팩터리를 만들어주는 방법이 있다.

- 상속용 클래스의 생성자는 재정의 가능 메서드를 호출하면 안된다.

- Cloneable과 Serializable 인터페이스를 상속용 클래스에서 구현한다면 주의하자.

아이템 20 추상 클래스보다는 인터페이스를 우선하라

추상 클래스

클래스를 설계도라 하면, 추상 클래스는 미완성 설계도에 비유할 수 있다.

추상 메서드

선언부만 작성하고 구현부는 작성하지 않은 채로 남겨 둔 것이 추상메서드이다.
추상 메서드는 상속받는 클래스에 따라 달라질 수 있다.

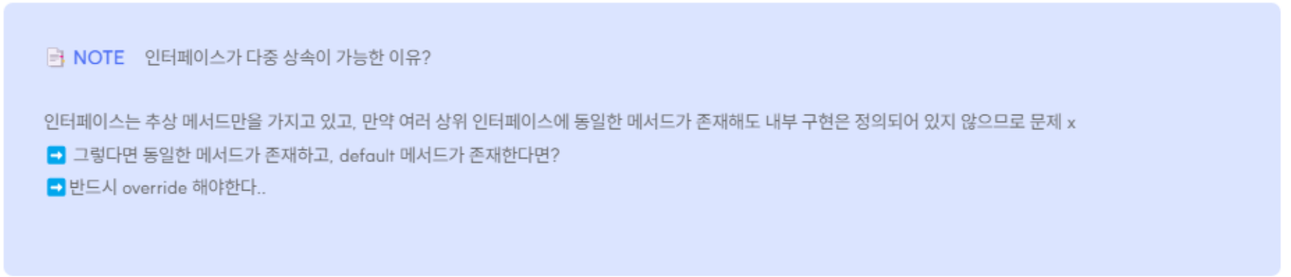
인터페이스

인터페이스는 일종의 추상 클래스로, 추상 메서드를 갖지만
추상 클래스보다 추상화 정도가 높아
추상 클래스와 달리 몸통을 갖춘 일반 메서드, 멤버 변수를 구성원으로 가질 수 없다.

아이템 20 추상 클래스보다는 인터페이스를 우선하라

인터페이스 규칙

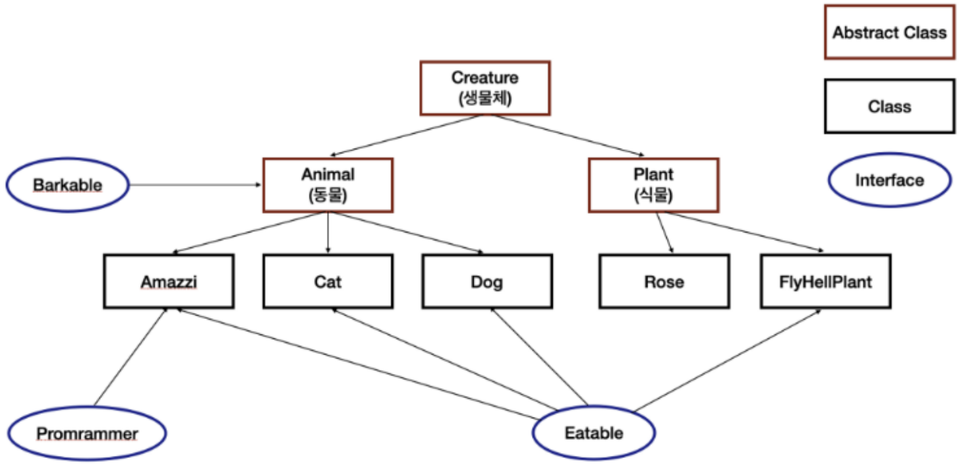
- 추상 클래스처럼 불완전한 것이기 때문에 그 자체만으로 사용되기 보다,다른 클래스를 작성하는데 도움을 줄 목적으로 작성된다.
- 일반 메서드 또는 멤버 변수를 구성원으로 가질 수 없다.
- 모든 멤버 변수는 public static final이어야 하며(초기화를 할 수 없으므로), 이를 생략할 수 있다.
- 모든 메서드는 public abtract이어야 하며, 이를 생략할 수 있다.(단, JDK1.8부터 static 메서드와 default 메서드를 사용할 수 있다.)



아이템 20 추상 클래스보다는 인터페이스를 우선하라

추상 클래스는 이를 상속할 각 객체들의 공통점을 찾아 추상화시켜 놓은 것으로,
상속 관계를 타고 올라갔을 때 같은 부모 클래스를 상속하며
부모 클래스가 가진 기능들을 구현해야할 경우 사용한다.

인터페이스는 상속 관계를 타고 올라갔을 때
다른 조상 클래스를 상속하더라도, 같은 기능이 필요할 경우 사용한다.
클래스와 별도로 구현 객체가 같은 동작을 한다는 것을 보장하기 위해 사용한다.



아이템 20 추상 클래스보다는 인터페이스를 우선하라

그렇다면 왜 인터페이스를 우선해야 할까?

첫번째 : 기존 클래스에도 손쉽게 새로운 인터페이스를 구현해 넣어 확장할 수 있다.
Comparable, Iterable, AutoCloseabl 등 새로운 인터페이스가 등장했을 때, 수 많은 기존 클래스가 이 인터페이스를 구현한 채 릴리스됐다.
하지만 기존 클래스에 새로운 추상 클래스를 끼워 넣기는 어렵다. 두 클래스가 같은 추상 클래스를 확장하려면, 그 추상 클래스는 계층 구조상 두 클래스의 공통 조상이어야 한다. 쉽지 않다.

두번째 : 인터페이스는 믹스인(mixin) 정의에 안성맞춤이다. ➡ 잘 이해안됨
믹스인은 클래스가 구현할 수 있는 타입으로, 클래스의 원래 '주된 타입'외에도 다른 특정 선택적 행위를 제공한다고 선언하는 것이다.
예를 들어 Comparable을 구현한 클래스는 자신을 구현한 인스턴스끼리 정렬이 가능하다고 선언하는 것이다.

아이템 20 추상 클래스보다는 인터페이스를 우선하라

세번째 : 인터페이스로는 계층구조가 없는 타입 프레임워크를 만들 수 있다.

구분이 어려운 siger와 songwriter의 경우 두 인터페이스를 모두 extend하는 제 3의 interface가 필요할 수 있다. 추상 클래스는 다중 상속이 불가능 하므로 힘들다.

네번째 : 인터페이스는 기능을 향상 시키는 안전하고 강력한 수단이 된다.
인터페이스의 메서드 중 구현 방법이 명백한 것이 있다면, 디폴트 메소드로 만들 수 있다.

그러나 디폴트 메서드는 제약이 있다.

- equals와 hashcode를 디폴트 메소드로 제공 안함
- 인터페이스는 인스턴스를 필드를 가질 수 없고, private 정적 메소드를 가질 수 없다.
- 본인이 만든 인터페이스가 아니면 디폴트 메소드 추가 불가능

아이템 20 추상 클래스보다는 인터페이스를 우선하라

인터페이스와 추상 골격 구현 클래스 ➡ 인터페이스와 추상 메서드의 장점을 모았다.
인터페이스로는 타입을 정의하고, 필요하면 디폴트 메서드도 몇 개 제공한다.
그리고 골격 구현 클래스는 나머지 메서드들까지 구현한다.

이렇게 하면 골격 구현을 확장하는 것만으로 이 인터페이스를 구현하는데 필요한 일이 대부분 완료된다. 이게 **템플릿 메서드 패턴**이다.