EJ

이펙티브 자바 CP.3

작성 일시	@2023년 1월 14일 오후 2:46
최종 편집 일시	@2023년 1월 18일 오후 8:29
유형	이펙티브 자바
작성자	
참석자	

3️⃣ 클래스와 인터페이스
15. 클래스와 멤버의 접근 권한을 최소화하라
16. public 클래스에서는 public 필드가 아닌 접근자 메서드를 사용해라
17. 변경 가능성을 최소화 해라
18. 상속보다는 컴포지션을 사용해라
19. 상속을 고려해 설계하고 문서화해라, 그러지 않았다면 상속을 금지해라
20. 추상 클래스 보다는 인터페이스를 우선해라.
21. 인터페이스는 구현하는 쪽을 생각해 설계해라
22. 인터페이스는 타입을 정의하는 용도로만 사용해라
23. 태그 달린 클래스보다는 클래스 계층구조를 활용해라
24. 멤버 클래스는 되도록 static 으로 만들어라
25. 톱레벨 클래스는 한 파일에 하나만 담으라

3️⃣ 클래스와 인터페이스

▼ 15. 클래스와 멤버의 접근 권한을 최소화하라

- **정보 은닉 (캡슐화)**
  - 어슬프게 설계된 컴포넌트와 잘 설계된 컴포넌트의 큰 차이는 바로 클래스 내부 데이터와 내부 구현 정보를 외부 컴포넌트로부터 얼마나 잘 숨겼느냐이다. 잘 설계된 컴포넌트는 모든 내부 구현을 완벽히 숨겨, 구현과 API를 깔금하게 분리한다. 오직 API를 통해서만 다른 컴포넌트와 소통하며, 서로의 내부 동작 방식에는 전혀 개의치 않는다.
  - 장점
    - 시스템 개발 속도를 높인다.
      여러 컴포넌트를 병렬로 개발할 수 있기 때문이다.
    - 시스템 관리 비용을 낮춘다.
      각 컴포넌트를 더 빨리 파악하여 디버깅할 수 있고, 다른 컴포넌트로 교체하는 부담도 적다.

        - 정보 은닉 자체가 성능을 높여주지는 않지만, 성능 최적화에 도움을 준다.
          완성된 시스템을 프로파일링해 최적화할 컴포넌트를 정한 다음(아이템 67), 다른 컴포넌에 영향을 주지 않고 해당 컴포넌트만 최적화할 수 있기 때문이다.
        - 소프트웨어 재사용성을 높인다.
          외부에 거의 의존하지 않고 독자적으로 동작할 수 있는 컴포넌트라면 그 컴포넌트와 함계 개발되지 않은 낯선 환경에서도 유용하게 쓰일 가능성이 있기 때문이다.
        - 큰 시스템을 제작하는 난이도를 낮춰준다.
          시스템 전체가 아직 완성되지 않은 상태에서도 개별 컴포넌트의 동작을 검증할 수 있기 때문이다.

- **접근 제한자**
    - 자바는 **정보 은닉**을 위한 다양한 장치를 제공한다. 그중 접근 제어 매커니즘은 클래스, 인터페이스, 멤버의 접근성 (접근 허용 범위)을 명시한다.
    - 각 요소의 접근성은 그 요소가 선언된 위치와 접근 제한자(private, protected, pulbic)로 정해진다. 이를 제대로 활용하는 것이 정보 은닉의 핵심이다.
    - 기본 원칙
        - 모든 클래스와 멤버의 접근성을 가능한 한 좁혀야 한다. (소프트웨어가 올바로 동작하는 한 항상 가장 낮은 접근 수준을 부여 해야한다.)
        - (가장 바깥이라는 의미의) 톱 레벨 클래스와 인터페이스에 부여할 수 있는 접근 수준은 package-private 와 public 이다.
        - 톱 클래스 클래스나 인터페이스를 public을 선언하면 공개 API가 되며, package-private으로 선언하면, 해당 패키지 안에서만 사용할 수 있다.
            - 패키지 외부에서 쓸 이유가 없다면 package-private에서 사용하자.
            - 이러면 API가 아닌 내부 구현이 되어 언제든 수정 할 수 있다.
            - 즉 클라이언트에 아무런 피해 없이, 다음 릴리스에서 수정, 교체, 제거 할 수있지만, public 인 경우에는 API가 되므로 하위 호환을 위해 영원히 관리해줘야 한다.
        - 한 클래스에서만 사용하는 package-private 톱레벨 클래스나 인터페이스는 이를 사용하는 클래스 안에 private static으로 중첩시켜보자. (아이템 24)
            - 톱 레벨로 두면 같은 패키지의 모든 클래스가 접근할 수 있지만, private static으로 중첩시키면 바깥 클래스 하나에서만 접근할 수 있다.
            - **public 일 필요가 없는 클래스의 접근 수준을 package-private 톱레벨 클래스로 좁히는 일이다.** (public 클래스는 그 패키지의 API, package-private 톱레벨 클래스는 내부 구현에 속함)
    - 구성
        - private: 멤버를 선언한 톱레벨 클래스에서만 접근할 수 있다.

        - package-private(default): 멤버가 소속된 패키지 안의 모든 클래스에서 접근 할 수 있다. 접근 제한자를 명시하지 않았을 때, 적용되는 패키지 접근 수준이다. (단, 인터페이스의 멤버는 기본적으로 public이 적용된다.)
        - protected: package-private의 접근 범위를 포함하여, 이 멤버를 선언한 클래스의 하위 클래스에서도 접근할 수 있다.
        - public: 모든 곳에서 접근할 수 있다.
    - 클래스 구현 방식
        - 공개 API 세심히 설계 한 후, 그 외 모든 멤버는 private로 만든다.
        - 그런 다음 오직 같은 패키지의 다른 클래스가 접근해야 하는 멤버에 한하여 private 제한자를 제거해 package-private으로 풀어주자
        - 더 권한을 풀어 주는 일을 자주 하게 된다. 시스템에서 컴포넌트를 더 분해해야 하는 것은 아닌지 고민한다.
        - private, package-private 멤버는 모두 해당 클래스의 구현에 해당하므로 보통 공개 API에 영향을 주지 않는다.
        - 단, Serializable을 구현한 클래스에서는 그 필드들도 의도치 않게 공개API가 될 수 있다. (아이템 86, 87)
        - public 클래스의 멤버가 package-private 에서 protected로 변경되는 순간, 공개 API로 변환됨으로, 영원히 지원되어야 한다. 또한 내부 방식을 API 문서에 적어 공개할 수도 있다. (아이템 19), 그러므로 protected 멤버는 적을수록 좋다.
    - 멤버 접근성 제약
        - 상위 클래스의 메서드를 재정의할 때, 그 접근 수준을 상위 클래스에서 보다 좁게 설정 할 수 없다. (리스코프 치환 원칙)
        - 이 규칙을 어기면 컴파일 에러난다.
        - 클래스가 인터페이스를 구현하는 건 이 규칙의 특별한 예로 볼 수 있고, 이때 클래스는 인터페이스가 정의한 모든 메서드를 public으로 선언해야 한다.
    - public 클래스의 인스턴스 필드는 되도록 public 이 아니어야 한다. (아이템16)
        - 필드가 가변 객체를 참조하거나, final이 아닌 인스턴스 필드를 public 으로 선언하면, 그 필드에 담을 수 있는 값을 제한할 힘을 잃게 된다. - 그 필드와 관련된 모든 것은 불변식을 보장할 수 없게 된다는 뜻.
        - 필드 수정 시, (락 획득 같은) 다른 작업을 할 수 없게 됨으로, public 가변 필드를 갖는 클래스는 일반적으로 스레드에 안전하지 않다.
        - 이는 정적 필드에서도 마찬가지이지만, 해당 클래스가 표현하는 추상 개념을 완성하는 데 꼭 필요한 구성요소로써의 상수라면 public static final 필드로 공개해도 좋다.
            - public static final double MATH_PIE = 3.1415926; 네이밍 (아이템 68)
            - 이런 필드는 반드시 기본 타입 값이나 불변 객체를 참조해야 한다. (아이템 17)

- 가변 객체를 참조한다면, final이 아닌 필드에 적용되는 모든 불이익이 그대로 적용된다.
- 길이가 0이 아닌 배열은 모두 변경 가능하니 주의하자, 따라서 클래스에서 public static final 배열 필드를 두거나 이 필드를 반환하는 접근 메서드를 제공해서는 안된다.

```
public static final Thing[] VALUES = {...};
// 이 경우 클라이언트에서 해당 배열 내용을 수정 할 수 있다.
// 기본 타입도 가능.
```

  - 해결 방안

```
// 1. public 배열을 private 으로 변경하고 public 불변 리스트를 추가한다.
private static final Thing[] PRIVATE_VALUE = {...};
public static final List<Thing> VALUES =
    Collections.unmodifiableList(Arrays.asList(PRIVATE_VALUE));

// 2.배열을 private으로 만들고 복사본을 반환하는 public 메서드를 추가하는 방법
private static final Thing[] PRIVATE_VALUE = {...};
public static final Thing[] values() {
    return PRIVATE_VALUES.clone(); // 아이템 13
}
```

- 정리
  - 프로그램 요소의 접근성은 가능한 한 최소한으로 해라.
  - 꼭 필요한 것만 골라 최소한의 public API를 설계한다.
  - 그 외에는 클래스, 인터페이스, 멤버가 의도치 않게 API로 공개되는 일은 없도록 한다.
  - public 클래스는 상수용 public static final 필드 외에는 어떠한 public 필드도 가져선 안된다.
  - public static final 필드가 참조하는 객체가 불변하는지 확인해라.

▼ 16. public 클래스에서는 public 필드가 아닌 접근자 메서드를 사용해라

- 퇴보한 클래스 - 캡슐화 이점을 제공하지 못한다.

```
class Point {
    public double x;
    public double y;
}
```

  이러한 클래스는 모두 필드를 private 로 변경하고 public 접근자 (setter, getter)를 추가하자.

- public 클래스의 정상적인 방식

```
class Point {
    private double x;
    private double y;
    public point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public double getX() { return x; }
    public double getY() { return y; }

    public void setX(double x) { this.x = x; }
    public void setY(double y) { this.y = y; }
```

  - 패키지 바깥에서 접근할 수 있는 클래스라면 접근자를 제공함으로 클래스 내부 표현 방식을 언제든 바꿀 수 있는 유연성을 얻을 수 있다. public 클래스가 필드를 공개하면 이를 사용하는 클라이언트가 생길 것이므로, 내부 표현 방식을 마음대로 바꿀 수 없게된다.
  - 하지만 package-private 클래스 혹은 private 중첩 클래스 라면 데이터 필드를 노출한다 해도 하등의 문제가 없다. 그 클래스가 표현하려는 추상 개념만 올바르게 표현하면 된다.
  - public 클래스의 필드가 불변이라면 직접 노출할 때의 단점은 조금 줄어들지만, 여전히 API 를 변경하지 않고는 표현 방식을 바꿀 수 없고, 필드를 읽을 때 부수 작업을 수행할 수 없다는 단점은 여전하다. (단 불변식은 보장 할 수 있게 된다.)
- 정리
  - public 클래스를 절대 가변 필드를 직접 노출해서는 안된다. 불변 필드라면 노출해도 덜 위험하지만, 완전히 안심할 수 는 없다. 하지만 package-private 클래스나 private 중첩 클래스에서는 종종 (불변, 가변) 필드를 노출 하는 편이 좋을 때도 있다.

▼ 17. 변경 가능성을 최소화 해라

- 불변 클래스 (인스턴스의 내부 값을 수정할 수 없는 클래스)
  - 불변 클래스를 만드는 다섯 가지 규칙
    - 객체의 상태를 변경하는 메서드(변경자)를 제공하지 않는다.
    - 클래스를 확장할 수 없도록 한다.
      - 하위 클래스에서 객체의 상태를 변하게 만드는 사태를 막아준다. 상속을 막는 대표적인 방법은 클래스를 final로 선언하는 것이지만, 다른 방법도 있다.
    - 모든 필드를 final 로 선언한다.
    - 모든 필드를 private 로 선언한다.
      - 필드가 참조하는 가변 객체를 클라이언트에서 직접 수정하는 일을 막아준다.
    - 자신 외에는 내부의 가변 컴포넌트에 접근할 수 없도록 한다.
      - 클래스에 가변 객체를 참조하는 필드가 하나라도 있다면 클라이언트에서 그 객체의 참조를 얻을 수 없도록 해야한다. 이런 필드는 절대 클라이언트가 제공하는

객체 참조를 가르키게 해서는 안 되며, 접근자 메서드가 그 필드를 그대로 반환해서는 안된다.

    - 생성자, 접근자, readObject 메서드 (아이템 88) 모두 방어적 복사를 수행하라.

- 불변 복소수 클래스

```
public class Complex {
    private final double re; // 실수부
    private final double im; // 허수부

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    public double realPart() { return re; }
    public double imaginaryPart() { return im; }

    public Complex plus(Complex c) {
        return new Complex(re + c.re, im + c.im);
    }

    public Complex minus(Complex c) {
        return new Complex(re - c.re, im - c.im);
    }

    public Complex times(Complex c) {
        return new Complex(re * c.re - im * c.im,
                re * c.im + im * c.re);
    }

    public Complex dividedBy(Complex c) {
        double tmp = c.re * c.re + c.im * c.im;
        return new Complex((re * c.re + im * c.im) / tmp,
                (im * c.re - re * c.im) / tmp);
    }

    @Override
    public String toString() {
        return "Complex{" +
                "re=" + re +
                ", im=" + im +
                '}';
    }

    @Override
    public boolean equals(Object o) {
        if (this == o)
            return true;
        if (!(o instanceof Complex))
            return false;
        Complex complex = (Complex) o;
        return Double.compare(complex.re, re) == 0 && Double.compare(complex.im, im) == 0;
    }

    @Override
    public int hashCode() {
        return 31 * Double.hashCode(re) + Double.hashCode(im);
    }
}
```

- 사칙연산 메서드들이 인스턴스 자신을 수정하지 않고 새로운 Complex 인스턴스를 만들어 반환한다. (함수형 프로그래밍)

- 불변 객체의 장점
  1. **불변 객체는 단순하다.** 불변 객체는 생성된 시점의 상태를 파괴될 때까지 그대로 간직한다. 가변 객체는 변경자 메서드로 임의의 복잡한 상태에 놓일 수 있다.
  2. **불변 객체는 근복적으로 스레드 안전하여 따로 동기화가 필요없다**. 클래스를 thread safe 하게 만드는 가장 쉬운 방법이다.
  3. **불변 객체는 안심하고 공유할 수 있다.** 따라서 생성된 불변 객체는 최대한 재활용를 권한다. (메모리 사용량과 가비지 컬렉션 비용이 줄어든다.) (방어적 복사가 필요 없다)
     a. 자주 쓰이는 값은 상수 (public static final) 불변 객체로 제공.
     b. 인스턴스를 중복 생성하지 않게 해주는 정적 팩터리(아이템 1)
  4. **불변 객체는 자유롭게 공유는 물론, 불변 객체끼리는 내부 데이터를 공유 할 수 있다.**
  5. **객체를 만들 때 다른 불변 객체들의 구성요소로 사용하면 이점이 많다.** 값이 바뀌지 않는 구성요소들 이뤄진 객체라면 그 구조가 복잡해도 불변식은 유지하기 쉬움. (map 의 key, Set 원소로 쓰기 좋음)
  6. **불변 객체는 그 자체로 실패 원자성을 제공한다.** (아이템 76)

- 불변 객체의 단점
  - **값이 다르면, 반드시 독립된 객체로 만들어야 한다. - 값을 변경하려면, 보다 성능에 좋지 않다.**
    - 성능에 대해 대처하는 방법
      - 다단계 연산들을 예측하여, 연산 속도를 높여주는 가변 동반 클래스(companion class)를 package-private 로 둔다.
      - 예측이 안되는 경우, 가변 동반 클래스를 public 으로 제공해라.

- 불변 클래스를 설계 방법
  - 상속하지 못하게 하는 방법
    - final 클래스 지정.
    - 더 flexible 방법으로는 모든 생성자를 private 혹은 package-private로 만들고 public 정적 팩터리를 제공하는 것이다.
    - 위에 적힌 불변 복소수 클래스에서 valueOf 정적 패터리 메서드와 생성자를 변경한 내용.

```
...
private Complex(double re, double im) {
    this.re = re;
    this.im = im;
```

```
}

public static Complex valueOf(double re, double im) {
    return new Complex(re, im);
}
...
```

- 정리
  - getter 가 있다고 해서 무조건 setter를 만들지 말자.
    - intellij 에서도 equals, hashcode 는 같이 엮지만, setter와 getter은 아니다.
    - 클래스는 꼭 필요한 경우가 아니면 불변이어야 한다.
    - 불변으로 만들 수 없는 클래스라도 변경 가능한 부분은 최소한으로 줄이자. (private final)
    - 생성자는 불변식 설정이 완료된, 초기화가 완벽히 끝난 상태의 객체를 생성해야 한다.

▼ 18. 상속보다는 컴포지션을 사용해라

- 상속 (클래스가 다른 클래스를 확장하는 구현 상속)
  - 안전한 사용법
    - 상위 클래스와 하위 클래스가 동일한 패키지 안에 존재
    - 확장할 목적으로 설계되었고, 문서화도 잘 된 클래스 (아이템 19)
  - 하지만, 일반적인 구체 클래스를 패키지 경계를 넘어 다른 패키지의 구체 클래스를 상속하는 것은 위험하다.
  - 상속은 코드 재사용성을 높여주지만 캡슐화를 깨뜨린다.
    - 상위 클래스가 어떻게 구현되느냐에 따라 하위 클래스의 동작에 이상이 생길 수 있다.
    - 상위 클래스는 릴리즈마다 내부 구현이 달라질 수 있으며, 그 여파로 코드 한 줄 건드리지 않은 하위 클래스가 오작동의 가능성이 있다.
  - 잘못된 상속의 예

```
public class InstrumentedHashSet<E> extends HashSet<E> {
    // 추가된 필드
    private int addCount = 0;

    public InstrumentedHashSet() {}

    public InstrumentedHashSet(int initCap, float loadFactor) {
        super(initCap, loadFactor);
    }

    @Override
    public boolean add(E e) {
```

```
        addCount++;
        return super.add(e);
    }

    @Override
    public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return super.addAll(c);
    }

    public int getAddCount() {
        return addCount;
    }
}

InstrumentedHashSet<String> s = new InstrumentedHashSet<>();
s.addAll(Arrays.asList("틱","톡","깍"));
s.getAddCount(); // 과연 3이 나올까?
```

  - s.getAddCount(); 를 하면 값은 6이 나온다. 이유로는 hashSet 의 addAll 메서드가 add 메서드를 사용하기 때문이다.
  - 이 경우 하위 클래스에서 addAll 메서드를 재정의하지 않으면 문제를 고칠 수 있다. 하지만, 이 문제를 확인하려면 상위 클래스의 구현 방법을 확인을 해야하는 한계를 갖는다. 이처럼 자신의 다른 부분을 사용하는 **자가사용** 여부는 해당 클래스의 내부 구현 방식에 해당하며, 자바 플랫폼 전반적인 정책인지**, 그래서 다음 릴리즈에도 유지가 되는지 알 수 없다.** 따라서 위 구현된 InstrumentedHashSet도 깨지기 쉽다.
  - addAll 메서드를 다른 식으로 재정의할 수도 있다. 하지만 상위 클래스의 메서드 동작을 다시 구현하는 것은 어렵고, 시간도 더 들고, 오류를 내거나 성능을 떨어뜨릴 수도 있다. 또한 하위 클래스에서는 접근할 수 없는 private 필드를 써야 하는 상황이라면 이 방식으로는 구현자체가 불가능하다.
  - 다음 릴리즈에서 상위 클래스에 새로운 메서드가 추가 된다고 한다면, 하위 클래스의 메서드 작성 시점에는 새로운 메소드는 존재하지도 않았으니, 하위 클래스의 메서드가 새롭게 추가된 메소드의 요구 규약을 지키지 않을 수도 있다.

- 컴포지션을 사용해라
  - 기존 클래스가 새로운 클래스의 구성요소로 쓰인다 (composition)
  - 새로운 클래스를 만들고 private 필드로 기존 클래스이 인스턴스를 참조한다.
  - 새 클래스의 인스턴스 메서드를은 기존 클래스의 대응하는 메서드를 호출해 그 결과를 반환한다. 이 방식을 전달(forwarding) 이라고 하며, 새 클래스의 메서드들은 전달 메서드(forwarding method) 라 부른다.
  - **새로운 클래스는 기존 클래스의 내부 구현 방식의 영향에서 벗어나며, 심지어 기존 클래스에 새로운 메서드가 추가되더라도 전혀 영향이 없다.**
  - 전달 메서드만으로 이뤄진 재사용 가능한 **전달 클래스**

```
public class ForwardingSet<E> implements Set<E> {
    private final Set<E> s;

    public ForwardingSet(Set<E> s) {
        this.s = s;
    }

    public int size() {
        return 0;
    }

    public boolean isEmpty() {
        return s.isEmpty();
    }

    public boolean contains(Object o) {
        return s.contains(o);
    }

    public Iterator<E> iterator() {
        return s.iterator();
    }

    public Object[] toArray() {
        return s.toArray();
    }

    public <T> T[] toArray(T[] a) {
        return s.toArray(a);
    }

    public boolean add(E e) {
        return s.add(e);
    }

    public boolean remove(Object o) {
        return s.remove(o);
    }

    public boolean containsAll(Collection<?> c) {
        return s.containsAll(c);
    }

    public boolean addAll(Collection<? extends E> c) {
        return s.addAll(c);
    }

    public boolean retainAll(Collection<?> c) {
        return s.retainAll(c);
    }

    public boolean removeAll(Collection<?> c) {
        return s.removeAll(c);
    }

    public void clear() {
        s.clear();
    }

    @Override
    public boolean equals(Object o) {
        return s.equals(o);
```

```
    }

    @Override
    public int hashCode() {
        return s.hashCode();
    }

    @Override
    public String toString() {
        return s.toString();
    }
}
```

- 집합 클래스

```
public class InstrumentedSet<E> extends ForwardingSet<E> {
    private int addCount = 0;

    public InstrumentedSet(Set<E> s) {
        super(s);
    }

    @Override
    public boolean add(E e) {
        addCount++;
        return super.add(e);
    }

    @Override
    public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return super.addAll(c);
    }

    public int getAddCount() {
        return addCount;
    }
}

InstrumentedSet<String> s = new InstrumentedSet<>(new HashSet<>());
s.addAll(Arrays.asList("틱","톡","깍"));
s.getAddCount(); // 3이 나온다.
// 이전에는 InstrumentedHashSet.addAll 에서 super.addAll(HashSet.addAll)에서
// InstrumentedHashSet.add 를 호출 한 것이고
// 현재는 InstrumentedSet.addAll 에서 super.addAll(HashSet.addAll)에서
// HashSet.add 를 호출 할 것이기 때문이다.
```

  - InstrumentedSet 은 HashSet의 모든 기능을 정의한 Set 인터페이스를 활용해 설계되어 견고하고 유여한다.
  - 임의의 Set에 계측 기능을 덧씌워 새로운 Set으로 만드는 것이 이 클래스의 핵심이다.
  - **상속 방식**은 구체 클래스 각각 따로 확장해야 하며, 지원하고 싶은 상위 클래스의 생성자 각각에 대응하는 생성자를 별도로 정의해야한다. 하지만 **컴포지션 방식**은 한 번만 구현해두면 어떠한 Set 구현체라도 계측할 수 있으며, 기존 생성자들과도 함께 사용할 수 있다.

        - 다른 Set 인스턴스를 감싸고(wrap) 있다는 뜻에서 **래퍼 클래스**라고 부른다.
        - 다른 Set 에 계측 기능을 덧씌운다는 뜻에서 **데코레이터 패턴**이라고 부른다.
        - 컴포지션과 전달의 조합은 넓은 의미로 **위임(delegation)**이라고 부른다. (엄밀히 따지면 래퍼 객체가 내부 객체에 자기 자신의 참조를 넘기는 경우만 위임에 해당한다.)
        - 래퍼 클래스는 단점이 거의 없다. (래퍼 클래스가 콜백 프레임워크와 어울리지 않다는 것만 주의), 콜백 프레임워크에서는 자기 자신의 참조를 다른 객체에 넘겨서 다음 호출(콜백) 때 사용한다. 내부 객체는 자신을 감싸고 있는 래퍼의 존재를 모르니 대신 자신(this)의 참조를 넘기고, 콜백 때는 래퍼가 아닌 내부 객체를 호출하게 되는데, 이를 SELF 문제라고 한다.

- 상속 is-a
    - 상속은 반드시 하위 클래스가 상위 클래스의 '진짜' 하위 타입인 상황에서만 쓰여야한다.
    - 즉, 클래스 B가 클래스 A와 is-a 관계 일때만, 클래스 A를 상속해야한다.
    - is-a 관계가 아니라면, A는 B의 필수 구성요소가 아니라 구현 방법중 하나일뿐이다.
- 상속을 사용하기 전 자문
    - 확장하려는 클래스의 API에 아무런 결함이 없는가?
    - 결함이 있다면, 이 결함이 하위 클래스의 API까지 전달되도 괜찮은가?
        - 컴포지션은 이런 결함을 숨기는 새로운 API 를 설계 할 수 있지만, 상속은 상위 클래스의 API를 '결함까지도' 상속 받는다.
- 정리
    - 상속은 강력하지만 캡슐화를 해친다.
    - 상속은 상위 클래스와 하위 클래스가 순수한 is-a 관계일 때만 사용한다.
        - is-a 일 때도 문제점으로… 하위 클래스의 패키지와 상위 클래스와 다르고, 상위 클래스가 확장을 고려하지 않고 설계되었다면, 문제가 된다.
    - 상속의 취약점을 피하려면 상속 대신 컴포지션과 전달을 사용해야한다. **특히 래퍼 클래스로 구현할 적당한 인터페이스가 있다면 더욱 그렇다**. 래퍼 클래스는 하위 클래스보다 견고하고 강력하다.

▼ 19. 상속을 고려해 설계하고 문서화해라, 그러지 않았다면 상속을 금지해라

▼ 20. 추상 클래스 보다는 인터페이스를 우선해라.

▼ 21. 인터페이스는 구현하는 쪽을 생각해 설계해라

▼ 22. 인터페이스는 타입을 정의하는 용도로만 사용해라

▼ 23. 태그 달린 클래스보다는 클래스 계층구조를 활용해라

▼ 24. 멤버 클래스는 되도록 static 으로 만들어라

▼ 25. 톱레벨 클래스는 한 파일에 하나만 담으라