💚

아이템 18) 상속보다는 컴포지션을 사용하라

- 이 책에서 말하는 상속은 “클래스가 다른 클래스를 확장하는” `구현 상속` 을 말하는 것
- **상속은 캡슐화를 깨뜨린다**
- 처음 생성된 이후, 원소가 몇 개 더해졌는지 알아내기 위해 변형된 HashSet 을 만든 `InstrumentedHashSet`의 예시

```
// HashSet을 상속하는 InstrumentedHashSet
public class InstrumentedHashSet<E> extends HashSet<E> {
    @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);
    }
}

InstrumentedHashSet<String> s = new InstrumentedHashSet<>();
s.addAll(List.of("1", "2", "3"));
```

1. InstrumentedHashSet 의 addAll 호출 → 3 증가
2. HashSet 의 addAll 호출 → 내부 구현이 각 원소를 add 메소드 호출해서 추가하므로 InstrumentedHashSet 의 add 메소드가 호출됨
3. InstrumentedHashSet 의 add 호출 3번 → 3 증가

- 최종 addCount : 6

- addAll 메소드를 재정의하지 않으면 될 수도 있지만, HashSet 의 addAll 메소드 구현방식에 대해 알고 있어야 가능한 구현 방식
- `자기 사용 여부` 은 **해당 클래스 내부 구현 방식**이어서, 자바 플랫폼의 정책인지 다음 릴리즈에서도 유지되는 것인지 확실하지 않음
- 하위 클래스가 깨지기 쉬운 이유는 다음 릴리즈에서 상위 클래스에 새로운 메소드를 추가한 경우, 하위 클래스에서 재정의하지 못한 메소드를 사용해 “허용되지 않은” 원소를 추가할 수 있게 된다
- 실제로 Hashtable, Vector 를 컬렉션 프레임워크에 포함시켰을 때 보안 구멍을 수정해야 했음

기존 클래스를 확장하는 대신, 새로운 클래스를 만들고 private 필드로 기존 클래스의 인스턴스를 참조하게 하자

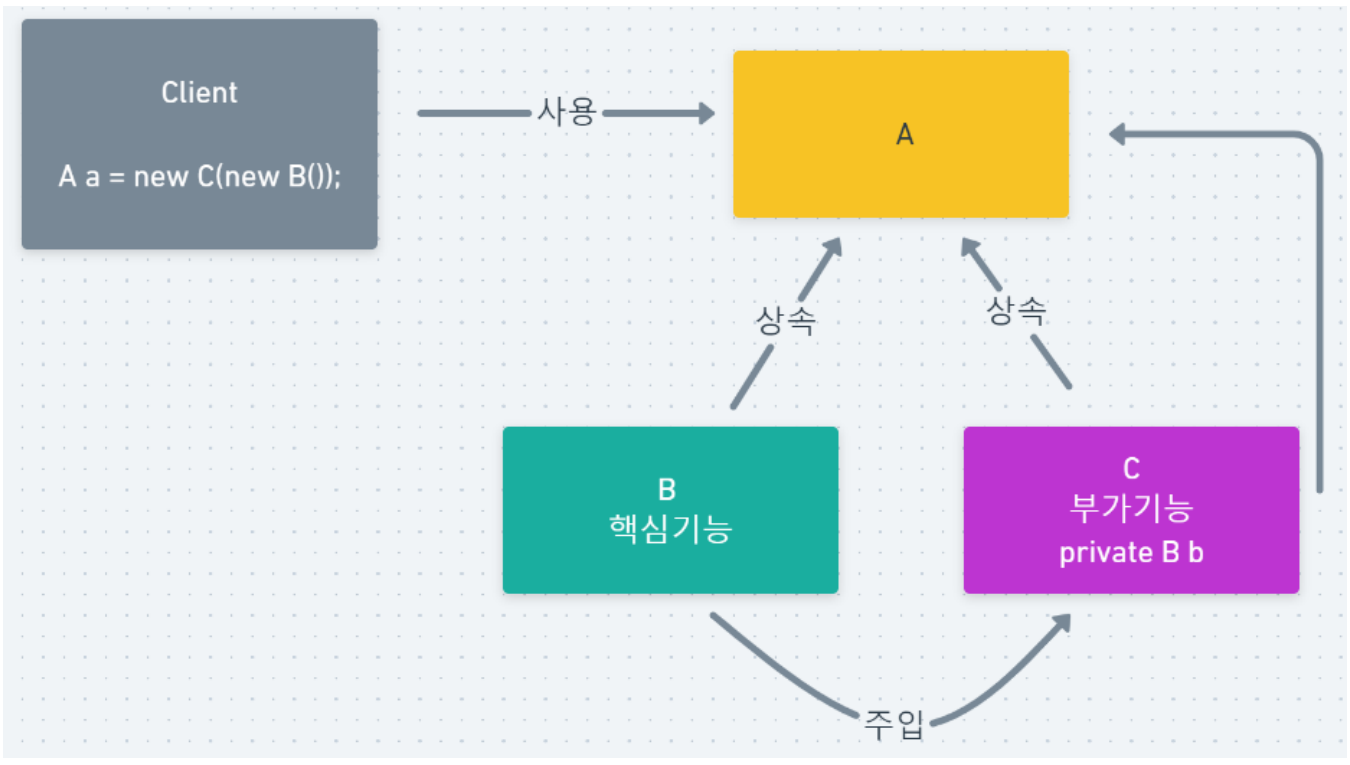
- 기존 클래스 (InstrumentedSet) 가 새로운 클래스 (ForwardingSet) 의 구성요소로 쓰인다는 뜻에서 `컴포지션` (구성) 이라고 함
- ForwardingSet 는 기존 클래스 (InstrumentedHashSet) 의 내부 구현에 영향 받지 않고, InstrumentedHashSet 에 메소드 추가된 경우에도 영향 X
- InstrumentedSet 은 HashSet 의 모든 기능을 정의한 Set 인터페이스

```
// ForwardingSet을 상속하는 InstrumentedSet
public class InstrumentedSet<E> extends ForwardingSet<E> {
    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);
    }
}
```

```
// Set 을 구현하는 ForwardingSet
public class ForwardingSet<E> implements Set<E> {
    private final Set<E> s;
    private ForwardingSet(Set<E> s) {
        this.s = s;
    }

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

```
Set<Instant> times = new InstrumentedSet<>(new TreeSet<>(cmp));
Set<E> s = new InstrumentedSet<>(new HashSet<>(INIT_CAPACITY));
// A a = new C(new B());
```

- Set 인스턴스를 감싸고 있다는 뜻에서 InstrumentedSet 같은 클래스를 `래퍼클래스` 라고 하며, 다른 Set 에 계측 기능을 덧씌운다는 뜻에서 `데코레이터 패턴` 이라고 함
  - 참고 : https://refactoring.guru/ko/design-patterns/decorator/java/example
  - `데코레이터` : 객체의 결합을 통해, 기능을 동적으로 확장하는 패턴으로, 기본 기능에 추가 기능인 Decorator 객체를 결합하는 기법
    - A 라는 인터페이스가 있고, B/C 구현체가 있다면, B가 핵심기능 담당하고 C가 부가 기능을 담당
    - Set [A] 이라는 `인터페이스` 가 있고, TreeSet [B] 이 `핵심기능을 담당` 하고, InstrumentedSet [C] 이 `부가 기능을 담당`



상속은 클래스 B 가 클래스 A 와 IS-A 관계일 때만 클래스 A 를 상속해야 한다

- 클래스 B 가 정말 A 인가? 라고 생각해보았을 때 “그렇다” 고 확신할 수 없다면 상속해서는 안되고, A 를 private 인스턴스로 두고 A 와는 다른 API 를 제공하자

ex) A - 새 / B - 펭귄

```
[ 새는 날 수 있다는 가정하에 구현 ]
public class Bird {
  public void fly(){...}
}

public class Penguin extends Bird { }


-----------------------------------------------
[새 > 날 수 없는 새 / 날 수 있는 새를 구분하여 구현]

public class Bird {}
public class FlyingBird extends Bird {
  public void fly(){...}
}
public class Penguin extends Bird {}
```

```
-----------------------------------------------

class Bird {
    private String species;

    public Bird(String species) {
        this.species = species;
    }

    public void fly() {
        System.out.println(species + "가 날아갑니다.");
    }
}

// 펭귄은 Bird 클래스와는 IS-A 관계가 아니므로 상속 대신 컴포지션을 사용
class Penguin {
    private Bird bird;

    public Penguin(String species) {
        this.bird = new Bird(species);
    }

    public void swim() {
        System.out.println("펭귄이 수영합니다.");
    }

    public void fly() {
        System.out.println("펭귄은 날지 못합니다.");
    }
}

public class CompositionExample {
    public static void main(String[] args) {
        Penguin penguin = new Penguin("펭귄");

        penguin.swim();
        penguin.fly();
    }
}
```

- 컴포지션을 써야 할 상황에서 상속을 사용하면, 내부 구현을 불필요하게 노출하여 클라이언트가 노출된 내부에 직접 접근 가능하다
    - Properties 인스턴스 p 가 있을 때 p.getProperty() / p.get() 의 결과가 다를 수 있음

```
p.getProPerty(key) // Properties 의 메소드 -> 문자열 key, value 만 가능
p.get(Key)
/*
  Hashtable 에서 물려 받은 메소드 -> 문자열 이외의 타입을 넣을 수 있음
  그래서 상위 클래스 Hashtable 의 메소드 직접 호출 시 불변식이 깨져버려 load, store 같은 API 를 사용할 수 없게 되었음
*/
```

결론

- 상속은 캡슐화를 해친다는 문제가 있으므로, 상위/하위 클래스가 is-a 관계인 것이 확실할 때만 사용해야 함
    - 하지만 이 경우에도 둘의 패키지 자체가 다르고, 상위 클래스가 확장을 고려하지 않고 설계되었다면 문제가 될 수 있음
- 상속 취약점을 피하기 위해 상속 대신 컴포지션과 전달을 사용하자