Item 클래스와 멤버의 접근 권한을 최소화 하라

Item 클래스와 멤버의 접근 권한을 최소화 하라

잘 설계된 컴포넌트는 모든 내부 구현을 완벽히 숨겨 **구현과 API를 깔끔히 분리**한 것

정보 은닉, 혹은 캡슐화라고 하는 이 개념은 소프트웨어 설계의 근간

Item 클래스와 멤버의 접근 권한을 최소화 하라

잘 설계된 컴포넌트는 모든 내부 구현을 완벽히 숨겨 **구현과 API를 깔끔히 분리**한 것

정보 은닉, 혹은 캡슐화라고 하는 이 개념은 소프트웨어 설계의 근간

정보 은닉의 장점

Item 클래스와 멤버의 접근 권한을 최소화 하라

잘 설계된 컴포넌트는 모든 내부 구현을 완벽히 숨겨 **구현과 API를 깔끔히 분리**한 것

정보 은닉, 혹은 캡슐화라고 하는 이 개념은 소프트웨어 설계의 근간

정보 은닉의 장점
개발속도 향상, 관리비용 낮춤, 성능 최적화, 재 사용성 향상, 개발 난이도 낮춰줌 …

Item 클래스와 멤버의 접근 권한을 최소화 하라

잘 설계된 컴포넌트는 모든 내부 구현을 완벽히 숨겨 **구현과 API를 깔끔히 분리**한 것

정보 은닉, 혹은 캡슐화라고 하는 이 개념은 소프트웨어 설계의 근간

정보 은닉의 장점
개발속도 향상, 관리비용 낮춤, 성능 최적화, 재 사용성 향상, 개발 난이도 낮춰줌 …

Q 어떻게 만드나요?

Item 클래스와 멤버의 접근 권한을 최소화 하라

잘 설계된 컴포넌트는 모든 내부 구현을 완벽히 숨겨 **구현과 API를 깔끔히 분리**한 것

정보 은닉, 혹은 캡슐화라고 하는 이 개념은 소프트웨어 설계의 근간

정보 은닉의 장점
개발속도 향상, 관리비용 낮춤, 성능 최적화, 재 사용성 향상, 개발 난이도 낮춰줌 …

Q 어떻게 만드나요?
모든 클래스와 멤버의 접근성을 가능한 한 좁힌다

Item 클래스와 멤버의 접근 권한을 최소화 하라

잘 설계된 컴포넌트는 모든 내부 구현을 완벽히 숨겨 **구현과 API를 깔끔히 분리**한 것

정보 은닉, 혹은 캡슐화라고 하는 이 개념은 소프트웨어 설계의 근간

정보 은닉의 장점
개발속도 향상, 관리비용 낮춤, 성능 최적화, 재 사용성 향상, 개발 난이도 낮춰줌 …

Q 어떻게 만드나요?
모든 클래스와 멤버의 접근성을 가능한 한 좁힌다 —> 올바로 동작 하는한, 가장 낮은 접근수준을 부여

Item 클래스와 멤버의 접근 권한을 최소화 하라

잘 설계된 컴포넌트는 모든 내부 구현을 완벽히 숨겨 **구현과 API를 깔끔히 분리**한 것

정보 은닉, 혹은 캡슐화라고 하는 이 개념은 소프트웨어 설계의 근간

정보 은닉의 장점
개발속도 향상, 관리비용 낮춤, 성능 최적화, 재 사용성 향상, 개발 난이도 낮춰줌 …

Q 어떻게 만드나요?
모든 클래스와 멤버의 접근성을 가능한 한 좁힌다 —> 올바로 동작 하는한, 가장 낮은 접근수준을 부여

Top-level 클래스 / 인터페이스에 부여할수 있는 접근수준은 **package-private & public** 두가지
=아무 상속도 받지않은 클래스

Item 클래스와 멤버의 접근 권한을 최소화 하라

잘 설계된 컴포넌트는 모든 내부 구현을 완벽히 숨겨 **구현과 API를 깔끔히 분리**한 것

정보 은닉, 혹은 캡슐화라고 하는 이 개념은 소프트웨어 설계의 근간

정보 은닉의 장점
개발속도 향상, 관리비용 낮춤, 성능 최적화, 재 사용성 향상, 개발 난이도 낮춰줌 …

Q 어떻게 만드나요?
모든 클래스와 멤버의 접근성을 가능한 한 좁힌다 —> 올바로 동작 하는한, 가장 낮은 접근수준을 부여

Top-level 클래스 / 인터페이스에 부여할수 있는 접근수준은 **package-private & public** 두가지
=아무 상속도 받지않은 클래스

public 으로 선언하는 순간 어떤 모듈에서든 자유롭게 호출가능

Item 클래스와 멤버의 접근 권한을 최소화 하라

잘 설계된 컴포넌트는 모든 내부 구현을 완벽히 숨겨 **구현과 API를 깔끔히 분리**한 것

정보 은닉, 혹은 캡슐화라고 하는 이 개념은 소프트웨어 설계의 근간

정보 은닉의 장점
개발속도 향상, 관리비용 낮춤, 성능 최적화, 재 사용성 향상, 개발 난이도 낮춰줌 …

Q 어떻게 만드나요?
모든 클래스와 멤버의 접근성을 가능한 한 좁힌다 —> 올바로 동작 하는한, 가장 낮은 접근수준을 부여

Top-level 클래스 / 인터페이스에 부여할수 있는 접근수준은 **package-private & public** 두가지
=아무 상속도 받지않은 클래스

public 으로 선언하는 순간 어떤 모듈에서든 자유롭게 호출가능 **이말은**
릴리즈되어 프로젝트에서 사용되는 순간부터 해당 모듈은 **하위 호환성을 위해 영원히 관리**되어야 한다

Item 클래스와 멤버의 접근 권한을 최소화 하라

멤버(필드, 메소드, 중첩 클래스, 중첩 인터페이스)에 부여할 수 있는 **접근 수준 네가지**

private : 멤버를 선언한 톱 레벨 클래스에서만 접근 가능
package-private : 멤버가 소속된 패키지 안의 모든 클래스가 접근 가능 (일반 클래스의 default)
protected : package-private + 이 멤버를 선언한 클래스의 하위 클래스에서 접근 가능
public : 모든 곳에서 접근 가능 (인터페이스의 default)

Item 클래스와 멤버의 접근 권한을 최소화 하라

그래서.. 어떻게 하라는거죠?

Item 클래스와 멤버의 접근 권한을 최소화 하라

그래서.. 어떻게 하라는거죠?

패키지 외부에서 쓸 일이 없다면 package-private(접근제어자의 default 값으로 아무것도 안 붙인 상태)로 하라

Item 클래스와 멤버의 접근 권한을 최소화 하라

그래서.. 어떻게 하라는거죠?

패키지 외부에서 쓸 일이 없다면 package-private(접근제어자의 default 값으로 아무것도 안 붙인 상태)로 하라

한 클래스에서만 사용되는 package-private 클래스는 사용하는 클래스의 private static 으로 중첩시켜 써라
이렇게 중첩하면 바깥 클래스에서만 접근할 수 있다

Item 클래스와 멤버의 접근 권한을 최소화 하라

그래서.. 어떻게 하라는거죠?

패키지 외부에서 쓸 일이 없다면 package-private(접근제어자의 default 값으로 아무것도 안 붙인 상태)로 하라

한 클래스에서만 사용되는 package-private 클래스는 사용하는 클래스의 private static 으로 중첩시켜 써라
이렇게 중첩하면 바깥 클래스에서만 접근할 수 있다

```
/**
 * default 접근자 (= package-private)
 * @author seongchan.kang
 */
class TopLevel {

    void hello() {
        System.out.println("Hello");
    }

    /**
     * 외부 클래스(TopLevel) 에서만 사용가능한 private inner class
     */
    private static class TopLevelStaticInner {
        void test() {
            TopLevel.this.hello();
        }
    }

    private class TopLevelInner {
        void test() {
            TopLevel.this.hello();
        }
    }

}
```

Item 클래스와 멤버의 접근 권한을 최소화 하라

그래서.. 어떻게 하라는거죠?

패키지 외부에서 쓸 일이 없다면 package-private(접근제어자의 default 값으로 아무것도 안 붙인 상태)로 하라

한 클래스에서만 사용되는 package-private 클래스는 사용하는 클래스의 private static 으로 중첩시켜 써라
이렇게 중첩하면 바깥 클래스에서만 접근할 수 있다

```java
/**
 * default 접근자 (= package-private)
 * @author seongchan.kang
 */
class TopLevel {

    void hello() {
        System.out.println("Hello");
    }

    /**
     * 외부 클래스(TopLevel) 에서만 사용가능한 private inner class
     */
    private static class TopLevelStaticInner {
        void test() {
            TopLevel.this.hello();
        }
    }

    private class TopLevelInner {
        void test() {
            TopLevel.this.hello();
        }
    }

}
```

static을 사용하는 이유
참조없는 독립된 클래스를 만들기위함

Item 클래스와 멤버의 접근 권한을 최소화 하라

그래서.. 어떻게 하라는거죠?

패키지 외부에서 쓸 일이 없다면 package-private(접근제어자의 default 값으로 아무것도 안 붙인 상태)로 하라

한 클래스에서만 사용되는 package-private 클래스는 사용하는 클래스의 private static 으로 중첩시켜 써라
이렇게 중첩하면 바깥 클래스에서만 접근할 수 있다

```java
/**
 * default 접근자 (= package-private)
 * @author seongchan.kang
 */
class TopLevel {

    void hello() {
        System.out.println("Hello");
    }

    /**
     * 외부 클래스(TopLevel) 에서만 사용가능한 private inner class
     */
    private static class TopLevelStaticInner {
        void test() {
            TopLevel.this.hello();
        }
    }

    private class TopLevelInner {
        void test() {
            TopLevel.this.hello();
        }
    }

}
```

static을 사용하는 이유
참조없는 독립된 클래스를 만들기위함

TopLevel.this 를 통해서 접근할수 있다
= 참조가 생긴다
= 시간과 공간이 소비된다

Item 클래스와 멤버의 접근 권한을 최소화 하라

그래서.. 어떻게 하라는거죠?

패키지 외부에서 쓸 일이 없다면 package-private(접근제어자의 default 값으로 아무것도 안 붙인 상태)로 하라

한 클래스에서만 사용되는 package-private 클래스는 사용하는 클래스의 private static 으로 중첩시켜 써라
이렇게 중첩하면 바깥 클래스에서만 접근할 수 있다

```
/**
 * default 접근자 (= package-private)
 * @author seongchan.kang
 */
class TopLevel {

    void hello() {
        System.out.println("Hello");
    }

    /**
     * 외부 클래스(TopLevel) 에서만 사용가능한 private inner class
     */
    private static class TopLevelStaticInner {
        void test() {
            TopLevel.this.hello();
        }
    }

    private class TopLevelInner {
        void test() {
            TopLevel.this.hello();
        }
    }

}
```

static을 사용하는 이유
참조없는 독립된 클래스를 만들기위함

TopLevel.this 를 통해서 접근할수 있다
= 참조가 생긴다
= 시간과 공간이 소비된다

```
private static class TopLevelStaticInner {
    void test() {
        TopLevel topLevel = new TopLevel();
        topLevel.hello();
    }
}
```

아이템24에 더 자세히 나오니깐 나중에 봐요^^

Item 클래스와 멤버의 접근 권한을 최소화 하라

그래서.. 어떻게 하라는거죠?

패키지 외부에서 쓸 일이 없다면 package-private(접근제어자의 default 값으로 아무것도 안 붙인 상태)로 하라

한 클래스에서만 사용되는 package-private 클래스는 사용하는 클래스의 private static 으로 중첩시켜 써라
이렇게 중첩하면 바깥 클래스에서만 접근할 수 있다

테스트 코드는 같은 패키지 경로에 두면 package-private 를 테스트할 수 있다
테스트를 위해 접근제어자를 푸는 것은 올바르지 않다

Item 클래스와 멤버의 접근 권한을 최소화 하라

그래서.. 어떻게 하라는거죠?

패키지 외부에서 쓸 일이 없다면 package-private(접근제어자의 default 값으로 아무것도 안 붙인 상태)로 하라

한 클래스에서만 사용되는 package-private 클래스는 사용하는 클래스의 private static 으로 중첩시켜 써라
이렇게 중첩하면 바깥 클래스에서만 접근할 수 있다

테스트 코드는 같은 패키지 경로에 두면 package-private 를 테스트할 수 있다
테스트를 위해 접근제어자를 푸는 것은 올바르지 않다

public 클래스의 인스턴스 필드는 되도록 public이 아니어야 한다
public 필드는 불변을 보장할 수 없기 때문이다 (=스레드에 안전하지 않다)

Item 클래스와 멤버의 접근 권한을 최소화 하라

그래서.. 어떻게 하라는거죠?

패키지 외부에서 쓸 일이 없다면 package-private(접근제어자의 default 값으로 아무것도 안 붙인 상태)로 하라

한 클래스에서만 사용되는 package-private 클래스는 사용하는 클래스의 private static 으로 중첩시켜 써라
이렇게 중첩하면 바깥 클래스에서만 접근할 수 있다

테스트 코드는 같은 패키지 경로에 두면 package-private 를 테스트할 수 있다
테스트를 위해 접근제어자를 푸는 것은 올바르지 않다

public 클래스의 인스턴스 필드는 되도록 public이 아니어야 한다
public 필드는 불변을 보장할 수 없기 때문이다 (=스레드에 안전하지 않다)

public static final는 기본 타입이나 불변 객체를 참조해야한다
다른 객체를 참조하도록 바꿀 순 없지만, 참조된 객체 자체가 수정될 수가 있기 때문이다
Collectionsunmodifiable List()를 통해 불변 리스트로 변경시키거나, 방어적 복사를 사용하자

Item public 클래스에서는 public 필드가 아닌 접근자 메서드를 사용하라

Item public 클래스에서는 public 필드가 아닌 접근자 메서드를 사용하라

public 클래스의 필드가 public 이라면 이것을 사용하는 클라이언트가 생길 수가 있으므로 변경비용이 비싸진다

Item public 클래스에서는 public 필드가 아닌 접근자 메서드를 사용하라

public 클래스의 필드가 public 이라면 이것을 사용하는 클라이언트가 생길 수가 있으므로 변경비용이 비싸진다

불변식을 보장할 수도 없고, 외부에서 필드에 접근할 때 부수적인 작업(복사본 던지기, 파라미터 검증 등)을 할 수도 없다

Item public 클래스에서는 public 필드가 아닌 접근자 메서드를 사용하라

public 클래스의 필드가 public 이라면 이것을 사용하는 클라이언트가 생길 수가 있으므로 변경비용이 비싸진다

불변식을 보장할 수도 없고, 외부에서 필드에 접근할 때 부수적인 작업(복사본 던지기, 파라미터 검증 등)을 할 수도 없다

이런 단점들은 private 필드에 **public 접근자(getter/setter)를 두면 해결**된다 (필요한 곳에서만 setter를 쓰자)

Item 변경 가능성을 최소화 하라

Item 변경 가능성을 최소화 하라

불변 클래스: 인스턴스의 내부 값을 수정할 수 없는 클래스

Thread safe이므로 **동기화할 필요가 없다**

Item 변경 가능성을 최소화 하라

불변 클래스: 인스턴스의 내부 값을 수정할 수 없는 클래스

Thread safe이므로 **동기화할 필요가 없다**
안심하고 공유가능하므로 **재사용 가능**하다

Item 변경 가능성을 최소화 하라

불변 클래스: 인스턴스의 내부 값을 수정할 수 없는 클래스

Thread safe이므로 **동기화할 필요가 없다**
안심하고 공유가능하므로 **재사용 가능**하다

```
public static final ZERO = new Complex(0, 0);
public static final One = new Complex(1, 0);
public static final Complex I = new Complex(0, 1);
```

Item 변경 가능성을 최소화 하라

불변 클래스: 인스턴스의 내부 값을 수정할 수 없는 클래스

Thread safe이므로 **동기화할 필요가 없다**
안심하고 공유가능하므로 **재사용 가능**하다

public static final ZERO = new Complex(0, 0);
public static final One = new Complex(1, 0);
public static final Complex I = new Complex(0, 1);

불변 클래스 생성 조건은 다음과 같다

1. 변경자 메소드를 제공하지 않음
2. 클래스를 확장할 수 없게 함 (대표적으로 final로 클래스를 선언하면 됨)
3. 모든 필드를 final로 선언
4. 모든 필드를 private으로 선언
5. 자신 외에는 내부 가변 컴포넌트에 접근할 수 없도록 한다

Item 변경 가능성을 최소화 하라

불변 클래스: 인스턴스의 내부 값을 수정할 수 없는 클래스

Thread safe이므로 **동기화할 필요가 없다**
안심하고 공유가능하므로 **재사용 가능**하다

```
public final class Complex {
    private final double re;
    private final double im;

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

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

public static final ZERO = new Complex(0, 0);
public static final One = new Complex(1, 0);
public static final Complex I = new Complex(0, 1);

불변 클래스 생성 조건은 다음과 같다

1. 변경자 메소드를 제공하지 않음
2. 클래스를 확장할 수 없게 함 (대표적으로 final로 클래스를 선언하면 됨)
3. 모든 필드를 final로 선언
4. 모든 필드를 private으로 선언
5. 자신 외에는 내부 가변 컴포넌트에 접근할 수 없도록 한다

Item 변경 가능성을 최소화 하라

불변 클래스: 인스턴스의 내부 값을 수정할 수 없는 클래스

Thread safe이므로 **동기화할 필요가 없다**
안심하고 공유가능하므로 **재사용 가능**하다

```
public static final ZERO = new Complex(0, 0);
public static final One = new Complex(1, 0);
public static final Complex I = new Complex(0, 1);
```

하지만 이런 구조로 복잡한 연산을 수행하게 된다면 굉장히 많은 불변 클래스를 생성해야한다

Item 변경 가능성을 최소화 하라

불변 클래스: 인스턴스의 내부 값을 수정할 수 없는 클래스

Thread safe이므로 **동기화할 필요가 없다**
안심하고 공유가능하므로 **재사용 가능**하다

public static final ZERO = new Complex(0, 0);
public static final One = new Complex(1, 0);
public static final Complex I = new Complex(0, 1);

하지만 이런 구조로 복잡한 연산을 수행하게 된다면 굉장히 많은 불변 클래스를 생성해야한다

이런 상황때문에 사용하는것이 바로 다단계 연산을 예측하여 기본 기능으로 제공하는 방법을 사용하는데
이러한 기능을 하는 클래스를 가변 **동반 클래스** 라고한다

Item 변경 가능성을 최소화 하라

불변 클래스: 인스턴스의 내부 값을 수정할 수 없는 클래스

Thread safe이므로 **동기화할 필요가 없다**
안심하고 공유가능하므로 **재사용 가능**하다

public static final ZERO = new Complex(0, 0);
public static final One = new Complex(1, 0);
public static final Complex I = new Complex(0, 1);

하지만 이런 구조로 복잡한 연산을 수행하게 된다면 굉장히 많은 불변 클래스를 생성해야한다

이런 상황때문에 사용하는것이 바로 다단계 연산을 예측하여 기본 기능으로 제공하는 방법을 사용하는데
이러한 기능을 하는 클래스를 **가변 동반 클래스** 라고한다

가변 동반 클래스는 주로 불변 클래스가 비즈니스 로직 연산 등 시간 복잡도가 높은 연산시
불필요한 클래스 생성을 막기 위해 내부적으로 사용한다

Item 변경 가능성을 최소화 하라

불변 클래스: 인스턴스의 내부 값을 수정할 수 없는 클래스

Thread safe이므로 **동기화할 필요가 없다**
안심하고 공유가능하므로 **재사용 가능**하다

public static final ZERO = new Complex(0, 0);
public static final One = new Complex(1, 0);
public static final Complex I = new Complex(0, 1);

하지만 이런 구조로 복잡한 연산을 수행하게 된다면 굉장히 많은 불변 클래스를 생성해야한다

이런 상황때문에 사용하는것이 바로 다단계 연산을 예측하여 기본 기능으로 제공하는 방법을 사용하는데
이러한 기능을 하는 클래스를 **가변 동반 클래스** 라고한다

가변 동반 클래스는 주로 불변 클래스가 비즈니스 로직 연산 등 시간 복잡도가 높은 연산시
불필요한 클래스 생성을 막기 위해 내부적으로 사용한다

책에서 말하는 예시중 하나가 **StringBuffer**이다

Item 변경 가능성을 최소화 하라

```
private transient char[] toStringCache;


@Override
public synchronized StringBuffer append(Object obj) {
    toStringCache = null;
    super.append(String.valueOf(obj));
    return this;
}

@Override
public synchronized StringBuffer append(String str) {
    toStringCache = null;
    super.append(str);
    return this;
}


@Override
public synchronized String toString() {
    if (toStringCache == null) {
        toStringCache = Arrays.copyOfRange(value, from: 0, count);
    }
    return new String(toStringCache, share: true);
}
```

책에서 말하는 예시중 하나가 **StringBuffer**이다

Item 변경 가능성을 최소화 하라

```
private transient char[] toStringCache;


@Override
public synchronized StringBuffer append(Object obj) {
    toStringCache = null;
    super.append(String.valueOf(obj));
    return this;
}

@Override
public synchronized StringBuffer append(String str) {
    toStringCache = null;
    super.append(str);
    return this;
}


@Override
public synchronized String toString() {
    if (toStringCache == null) {
        toStringCache = Arrays.copyOfRange(value, from: 0, count);
    }
    return new String(toStringCache, share: true);
}
```

책에서 말하는 예시중 하나가 **StringBuffer**이다

이거말고도 가변 동반 클래스 의 종류는 엄청 다양하고
대부분이 어렵게 구현되어있다

우리는 그저 구현되어있는걸 잘 쓰면 다행인것 같다

Item 상속 보다는 컴포지션을 사용하라

Item 상속 보다는 컴포지션을 사용하라

Method 호출과 달리 상속은 캡슐화를 깨뜨린다

상위 클래스 구현에 따라 하위 클래스 동작에 이상이 생길 수 있다

상위 클래스가 바뀌고 릴리스 된다면 어떻게 될까?

Item 상속 보다는 컴포지션을 사용하라

Method 호출과 달리 상속은 캡슐화를 깨뜨린다

상위 클래스 구현에 따라 하위 클래스 동작에 이상이 생길 수 있다

상위 클래스가 바뀌고 릴리스 된다면 어떻게 될까?

```
package com.kohen.kang

Class A {
    dot = “”;

    Method addDot(param) {
        return param+dot;
    }

}
```

```
Import com.kohen.kang.A

Class ABC extends A {

    Method dotdotdot() {
        return superaddDot(addDot(addDot(“으악”)));
    }
}
```

Item 상속 보다는 컴포지션을 사용하라

Method 호출과 달리 상속은 캡슐화를 깨뜨린다

상위 클래스 구현에 따라 하위 클래스 동작에 이상이 생길 수 있다

상위 클래스가 바뀌고 릴리스 된다면 어떻게 될까?

```
package com.kohen.kang

Class A {
    dot = "";

    Method addDot(param) {
        return param+dot;
    }

}
```

```
Import com.kohen.kang.A

Class ABC extends A {

    Method dotdotdot() {
        return superaddDot(addDot(addDot("으악")));
    }
}


ABC.dotdotdot()

으악...
```

Item 상속 보다는 컴포지션을 사용하라

Method 호출과 달리 상속은 캡슐화를 깨뜨린다

상위 클래스 구현에 따라 하위 클래스 동작에 이상이 생길 수 있다

상위 클래스가 바뀌고 릴리스 된다면 어떻게 될까?

```
package com.kohen.kang

Class A {
    dot = "";

    Method addDot(param) {
        return param+dot;
    }

}


Release 20
Class A {
    dot = "!";

    Method addDot(param) {
        Return param+dot;
    }

}
```

```
Import com.kohen.kang.A

Class ABC extends A {

    Method dotdotdot() {
        return super.addDot(addDot(addDot("으악")));
    }
}


ABC.dotdotdot()

으악...
```

Item 상속 보다는 컴포지션을 사용하라

Method 호출과 달리 상속은 캡슐화를 깨뜨린다

상위 클래스 구현에 따라 하위 클래스 동작에 이상이 생길 수 있다

상위 클래스가 바뀌고 릴리스 된다면 어떻게 될까?

```
package com.kohen.kang

Class A {
    dot = "";

    Method addDot(param) {
        return param+dot;
    }

}


Release 20
Class A {
    dot = "!";

    Method addDot(param) {
        Return param+dot;
    }

}
```

```
Import com.kohen.kang.A

Class ABC extends A {

    Method dotdotdot() {
        return super.addDot(addDot(addDot("으악")));
    }
}


ABC.dotdotdot()

으악...

으악!!!
```

Item 상속 보다는 컴포지션을 사용하라

Method 호출과 달리 상속은 캡슐화를 깨뜨린다

상위 클래스 구현에 따라 하위 클래스 동작에 이상이 생길 수 있다

상위 클래스가 바뀌고 릴리스 된다면 어떻게 될까?

```
package com.kohen.kang

Class A {
    dot = "";

    Method addDot(param) {
        return param+dot;
    }

}


Release 20
Class A {
    dot = "!";

    Method addDot(param) {
        Return param+dot;
    }

}
```

```
Import com.kohen.kang.A

Class ABC extends A {

    Method dotdotdot() {
        return super.addDot(addDot(addDot("으악")));
    }
}


ABC.dotdotdot()

으악...

으악!!!

으아아아아아아아아악!!!!!!!
```

Item 상속 보다는 컴포지션을 사용하라

이러한 문제를 해결할 수 있는 방법 중 하나가 바로 **컴포지션**이다

컴포지션은 기존 클래스를 확장하는 대신, 새로운 클래스를 만들고 private 필드로 기존 클래스의 인스턴스를 참조한다

기존 클래스가 새로운 클래스의 구성요소로 쓰인다는 뜻에서 이러한 설계를 컴포지션이라 한다

새 클래스의 인스턴스 메서드들은 기존 클래스의 대응하는 메서드를 호출해 그 결과를 반환한다

이 방식을 전달이라 하며 새 클래스의 메서드들을 전달 메서드라고 부른다

Item 상속 보다는 컴포지션을 사용하라

이러한 문제를 해결할 수 있는 방법 중 하나가 바로 **컴포지션**이다

컴포지션은 기존 클래스를 확장하는 대신, **새로운 클래스를 만들고 private 필드로 기존 클래스의 인스턴스를 참조**한다

기존 클래스가 **새로운 클래스의 구성요소로 쓰인다는 뜻에서 이러한 설계를 컴포지션**이라 한다

새 클래스의 인스턴스 메서드들은 기존 클래스의 대응하는 메서드를 호출해 그 결과를 반환한다

이 방식을 전달이라 하며 새 클래스의 메서드들을 전달 메서드라고 부른다

Item 상속 보다는 컴포지션을 사용하라

이러한 문제를 해결할 수 있는 방법 중 하나가 바로 **컴포지션**이다

컴포지션은 기존 클래스를 확장하는 대신, **새로운 클래스를 만들고 private 필드로 기존 클래스의 인스턴스를 참조**한다

기존 클래스가 **새로운 클래스의 구성요소로 쓰인다는 뜻에서 이러한 설계를 컴포지션**이라 한다

새 클래스의 인스턴스 메서드들은 기존 클래스의 대응하는 메서드를 호출해 그 결과를 반환한다

이 방식을 전달이라 하며 새 클래스의 메서드들을 전달 메서드라고 부른다

말로하는것보단 코드로 보는것이 나을듯하니…

HashSet의 add를 사용하는 일반적인 상속으로 구현

```
class InstrumentedHashSet<E> extends HashSet<E> {
    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;
    }
}
```

HashSet의 add를 사용하는 일반적인 상속으로 구현

```
class InstrumentedHashSet<E> extends HashSet<E> {
    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;
    }
}
```

상속 객체의 메서드 addAll은 자기사용(self-use) 메서드로
내부적으로 add를 호출하고 있다. 이를 사용하는 클라이언트는
이를 알 방법이 없으므로 기대한 addCount값보다 2배의 값을 return 받는다

Item 상속 보다는 컴포지션을 사용하라

컴포지션을 사용한다면..

```
class InstrumentedHashSet<E> extends FowardingSet<E> {
    private int addCount = 0;

    public InstrumentedHashSet(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;
    }
}
```

```
class FowardingSet<E> implements Set<E> {
    private final Set<E> s;
    public FowardingSet(Set<E> s) { this.s = s; }

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

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

    @Override
    public int size() {
        return 0;
    }

    ...
```

Item 상속 보다는 컴포지션을 사용하라

컴포지션을 사용한다면..

```
class InstrumentedHashSet<E> extends FowardingSet<E> {
    private int addCount = 0;

    public InstrumentedHashSet(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;
    }
}
```

```
class FowardingSet<E> implements Set<E> {
    private final Set<E> s;
    public FowardingSet(Set<E> s) { this.s = s; }

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

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

    @Override
    public int size() {
        return 0;
    }

    ...
```

* 전달 메서드

새 클래스의 인스턴스 메서드들은 기존 클래스의 대응하는 메서드를 호출해 결과를 반환한다
이 방식을 **전달 메서드**라고 부른다

Item 상속 보다는 컴포지션을 사용하라

컴포지션을 사용한다면..

```java
class InstrumentedHashSet<E> extends FowardingSet<E> {
    private int addCount = 0;

    public InstrumentedHashSet(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;
    }
}
```

* 래퍼 클래스, 데코레이터 패턴

```java
class FowardingSet<E> implements Set<E> {
    private final Set<E> s;
    public FowardingSet(Set<E> s) { this.s = s; }

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

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

    @Override
    public int size() {
        return 0;
    }

    ...
```

다른 Set 인스턴스를 감싸고 있다는 뜻에서 **래퍼 클래스**라고 하며
Set 계측 기능을 덧씌운다는 의미에서 **데코레이터 패턴**이라고 한다

Item 상속 보다는 컴포지션을 사용하라

컴포지션을 사용한다면..

```
class InstrumentedHashSet<E> extends FowardingSet<E> {
    private int addCount = 0;

    public InstrumentedHashSet(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;
    }
}
```

```
class FowardingSet<E> implements Set<E> {
    private final Set<E> s;
    public FowardingSet(Set<E> s) { this.s = s; }

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

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

    @Override
    public int size() {
        return 0;
    }

    ...
```

이런식으로 사용하게되면 Set의 내부 구현방식에 얽매이지 않고 FowardingSet으로부터 전달받은 기능으로써 동작하기 때문에 내부 동작에 영향을 미치는 경우(addAll이 자기사용을 통해 add를 다시 호출하는 경우와 같이)가 없어지게 된다

Item 상속 보다는 컴포지션을 사용하라

의문이 들것이다.. 그럼 상속은 대체 언제쓰는가

Item 상속 보다는 컴포지션을 사용하라

의문이 들것이다.. 그럼 상속은 대체 언제쓰는가

이렇게 생각하면 된다

A를 상속하고자 하는 B의 관계가 'B is-a A' 관계를 만족시키는 경우 상속을 사용한다

Item 상속 보다는 컴포지션을 사용하라

의문이 들것이다.. 그럼 상속은 대체 언제쓰는가

이렇게 생각하면 된다

A를 상속하고자 하는 B의 관계가 'B is-a A' 관계를 만족시키는 경우 상속을 사용한다

상속을 단지 기능을 위임받아 확장시키기 위한 용도로 쓴다면 그것은 상속이 아닌, 컴포지션을 사용하여 확장하여야 한다

이런 명세는 자바 플랫폼 라이브러리에서 또한 지켜지지 않은 케이스이기때문에, 일반적인 개발 환경에서는 더욱 빈번히 발생한다

Item 상속 보다는 컴포지션을 사용하라

의문이 들것이다.. 그럼 상속은 대체 언제쓰는가

이렇게 생각하면 된다

A를 상속하고자 하는 B의 관계가 'B is-a A' 관계를 만족시키는 경우 상속을 사용한다

상속을 단지 기능을 위임받아 확장시키기 위한 용도로 쓴다면 그것은 상속이 아닌, 컴포지션을 사용하여 확장하여야 한다

이런 명세는 자바 플랫폼 라이브러리에서 또한 지켜지지 않은 케이스이기때문에, 일반적인 개발 환경에서는 더욱 빈번히 발생한다

강아지(A)는 동물(B)이냐? (Dog extends Animal) => **O**

Item 상속 보다는 컴포지션을 사용하라

의문이 들것이다.. 그럼 상속은 대체 언제쓰는가

이렇게 생각하면 된다

A를 상속하고자 하는 B의 관계가 'B is-a A' 관계를 만족시키는 경우 상속을 사용한다

상속을 단지 기능을 위임받아 확장시키기 위한 용도로 쓴다면 그것은 상속이 아닌, 컴포지션을 사용하여 확장하여야 한다

이런 명세는 자바 플랫폼 라이브러리에서 또한 지켜지지 않은 케이스이기때문에, 일반적인 개발 환경에서는 더욱 빈번히 발생한다

강아지(A)는 동물(B)이냐? (Dog extends Animal) => O
(계산기 기능을 갖고있는) 개는 계산기냐 (Dog extends Calculator) => X

Item 상속 보다는 컴포지션을 사용하라

의문이 들것이다.. 그럼 상속은 대체 언제쓰는가

이렇게 생각하면 된다

A를 상속하고자 하는 B의 관계가 'B is-a A' 관계를 만족시키는 경우 상속을 사용한다

상속을 단지 기능을 위임받아 확장시키기 위한 용도로 쓴다면 그것은 상속이 아닌, 컴포지션을 사용하여 확장하여야 한다

이런 명세는 자바 플랫폼 라이브러리에서 또한 지켜지지 않은 케이스이기때문에, 일반적인 개발 환경에서는 더욱 빈번히 발생한다

강아지(A)는 동물(B)이냐? (Dog extends Animal) => **O**
(계산기 기능을 갖고있는) 개는 계산기냐 (Dog extends Calculator) => **X**
(연예인 기능을 갖고있는) 희찬이는 연예인이냐(Heechan extends Celeb) => **(?)**

Item 상속 보다는 컴포지션을 사용하라

의문이 들것이다.. 그럼 상속은 대체 언제쓰는가

이렇게 생각하면 된다

A를 상속하고자 하는 B의 관계가 'B is-a A' 관계를 만족시키는 경우 상속을 사용한다

상속을 단지 기능을 위임받아 확장시키기 위한 용도로 쓴다면 그것은 상속이 아닌, 컴포지션을 사용하여 확장하여야 한다

이런 명세는 자바 플랫폼 라이브러리에서 또한 지켜지지 않은 케이스이기때문에, 일반적인 개발 환경에서는 더욱 빈번히 발생한다

강아지(A)는 동물(B)이냐? (Dog extends Animal) => **O**
(계산기 기능을 갖고있는) 개는 계산기냐 (Dog extends Calculator) => **X**
(연예인 기능을 갖고있는) 희찬이는 연예인이냐(Heechan extends Celeb) => **(?)**

이렇게 판단해보면 상속, 컴포지션을 언제 사용해야할지 감이 올 것이다

Item

끝!