EJ

이펙티브 자바 CP.3

작성 일시	@2023년 1월 14일 오후 3:46
최종 편집 일시	@2023년 1월 27일 오후 8:19
유형	이펙티브 자바
작성자	
참석자	

3️⃣ 클래스와 인터페이스

선행 내용 - 객체 지향 5원칙 (SOLID)

▼ 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 `public`을 선언하면 공개 API가 되며, `package-private`으로 선언하면, 해당 패키지 안에서만 사용할 수 있다.
            - 패키지 외부에서 쓸 이유가 없다면 `package-private`에서 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 static`으로 중첩시켜보자. (아이템 24)
            - 톱 레벨로 두면 같은 패키지의 모든 클래스가 접근할 수 있지만, `private static` 으로 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 수있다. 접근 제한자를 명시하지 않았을 때, 적용되는 패키지 접근 수준이다. (단, 인터페이스의 멤버는 기본적으로 public이 적용된다.)
  - `protected`: `package-private`의 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 `private`로 만든다.
  - 그런 다음 오직 같은 패키지의 다른 클래스가 접근해야 하는 멤버에 한하여 `private` 제한자를 제거해 `package-private`으로 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, `Serializable`을 구현한 클래스에서는 그 필드들도 의도치 않게 공개API가 될 수 있다. (아이템 86, 87)
  - `public` 클래스의 멤버가 `package-private` 에서 `protected`로 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 `public`으로 선언해야 한다.
- `public`클래스의 인스턴스 필드는 되도록 `public` 이 아니어야 한다. (아이템16)
  - 필드가 가변 객체를 참조하거나, `final`이 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 좋

다.

- `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, `final`이 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_VALUE.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 `final`로 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-private`로 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 getter`은 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 `InstrumentedHashSet`도 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 메서드들은 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 `HashSet`의 모든 기능을 정의한 `Set` 인터페이스를 활용해 설계되어 견고하고 유여한다.
- 임의의 `Set`에 계측 기능을 덧씌워 새로운 `Set`으로 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.
        - `InstrumentedSet` 같은 클래스를 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

- 상속 설계 전

1. 메서드를 재정의하면 어떤 일이 일어나는지를 정확히 정리하여 문서로 남겨야한다. 상속용 클래스는 재정의할 수 있는 메서드(`public, protected` 메서드 중 `final` 이 아닌 메서드)들을 내부적으로 어떻게 이용하는지(자기사용) 문서로 남겨야 한다. 더 넓게 말하자면, **재정의 가능 메서드를 호출할 수 있는 모든 상황을 문서로 남겨야한다.**
    - API 문서 메서드 설명 끝에 종종 “Implementation Requirements” 로 시작하는 절은 그 메섣의 내부 동작 방식을 설명하는 곳이다. (메서드 주석에 `@implSpec` 이란 태그를 붙이면 `JavaDoc` 이 생성해줌)
    - `java.util.AbstractCollection` 에서 발췌한 예

```
public boolean remove(Object o)
주어진 원소가 컬렉션 안에 있다면 그 인스턴스를 하나 제거한다(선택적 동작).
더 정확하게 말하면, 이 컬렉션 안에 'Object.equals(o, e)가 참인 원소' e가
하나 이상 있다면 그 중 하나를 제거한다. 주어진 원소가 컬렉션 안에 있었다면
(즉, 호출 결과 이 컬렉션이 변경 됐다면) true를 반환한다.
  Implementation Requirements: 이 메서드는 컬렉션을 순회하며 주어진 원소를
찾도록 구현되었다. 주어진 원소를 찾으면 반복자의 remove 메서드를 사용해
컬렉션에서 제거한다. 이 컬렉션이 주어진 객체를 갖고 있으나, 이 컬렉션의 iterator
메서드가 반환한 반복자가 remove 메서드를 구현하지 않았다면,
UnsupportedOperationException을 던지니 주의하자.
```

        - `iterator` 메서드를 재정의하면 `remove` 메서드의 동작에 영향을 줌을 알 수 있다.
        - **좋은 API 문서란 ‘어떻게’가 아닌 ‘무엇’을 하는지 설명해야 한다**. 하지만, 상속이 캡슐화를 망치기 때문에 일이나는 안타까운 현실이다. **클래스를 안전하게 상속하려면 내부 구현 방식을 설명해야한다.**
2. 효율적인 하위 클래스를 큰 어려움 없이 만드려면 클래스의 내부 동작 과정 중간에 끼어들 수 있는 훅을 잘 선별하여 `protected` **메서드 형태로 공개해야 할 수도 있다.**
    - `java.util.AbstractList` 의 `removeRange` 메서드 예

```
protected void removeRange(int fromIndex, int toIndex)
fromIndex(포함)부터 toIndex(미포함)까지의 모든 원소를 이 리스트에서 제거한다.
toIndex 이후의 원소들은 앞으로 (index 만큼씩) 당겨진다. 이 호출로 리스트는
'toIndex - fromIndex' 만큼 짧아진다. (toIndex == fromIndex 라면 아무 효과 없다)
이 리스트 혹은 이 리스트의 부분리스트에 정의된 clear 연산이 이 메서드를 호출한다.
리스트 구현의 내부 구조를 활용하도록 이 메서드를 재정의하면 이 리스트와 부분리스트의
clear 연산 성능을 크게 개선할 수 있다.
  Implementation Requirements: 이 메서드는 fromIndex에서 시작하는 리스트 반복
자를 얻어 모든 원소를 제거할 때까지 ListIterator.next와 ListIterator.remove를
반복 호출하도록 구현되었다. 주의: ListIterator.remove 가 선형 시간이 걸리면
이 구현의 성능은 제곱에 비례한다.

Parameters:
  fromIndex 제거할 첫 원소의 인덱스
  toIndex   제거할 마지막 원소의 다음 인덱스
```

        - 이 메서드의 설명을 제공한 이유는 단지 하위 클래스에서 부분리스트의 `clear` 메서드를 고성능으로 만들기 쉽게 하기 위해서다. `removeRange` 메서드가 없다면

하위 클래스에서 `clear` 메서드를 호출하면 (제거할 원소 수의) 제곱에 비례해 성능이 느려지거나 부분리스트의 매커니즘을 밑바닥부터 새로 구현해야 했을 것이다.

- 어떤 메서드를 `protected`로 노출해야 하는 것일까?
  - `protected` 메서드 하나하나가 내부 구현에 해당하므로 그 수는 가능한 한 적어야 한다. 한편으로는 너무 적게 노출해서 상속으로 얻는 이점마저 없애지 않도록 주의 해야한다... 😢
  - **상속용 클래스를 시험하는 방법은 직접 하위 클래스를 만들어 보는 것이 '유일' 하다.**
  - 꼭 필요한 `protected` 멤버를 놓쳤다면 하위 클래스를 작성할 때 그 빈 자리가 확연히 드러난다.
  - 반대로, 하위 클래스를 여러 개 만들 때까지 전혀 쓰이지 않는 `protected` 멤버는 사실 `private` 이었어야 할 가능성이 크다.
- 널리 쓰일 클래스를 상속용으로 설계한다면, 문서화 된 내부 사용 패턴과 `protected` 메서드와 필드를 구현하면서 선택한 결정에 영원히 책임져야 함을 인식해야한다.
  - **상속용으로 설계한 클래스는 배포 전에 반드시 하위 클래스를 만들어 검증해야 한다.**

3. 상속용 클래스의 생성자는 **직접적으로든 간접적으로든 재정의 가능 메서드를 호출해서는 안된다.**
   - 상위 클래스의 생성자가 하위 클래스의 생성자보다 먼저 실행되므로 하위 클래스에서 재정의한 메서드가 하위 클래스의 생성자 보다 먼저 호출된다.
   - 이때 그 재정의한 메서드가 하위 클래스의 생성자에서 초기화하는 값에 의존한다면 의도대도 동작하지 않을 것이다.
   - 이 규칙을 어기는 예제 코드

```
public class Super {
    // 잘못된 예 - 생성자가 재정의 기능 메서드를 호출한다.
    public Super() {
        overrideMe();
    }
    public void overrideMe() {}
}

public final class Sub extends Super {
    // 초기화 되지 않은 final 필드. 생성자에서 초기화 한다.
    private final Instant instant;
    Sub() {
        instant = Instant.now();
    }

    // 재정의 가능 메서드. 상위 클래스의 생성자가 호출한다.
    @Override
    public void overrideMe() {
```

```
        System.out.println(instant);
    }
```

    - 이 프로그램은 `instant`를 두 번 출력하는 대신, 첫 번째에서 `null`을 출력한다.
    - 상위 클래스의 생성자는 하위 클래스의 생성자가 인스턴스 필드를 초기화하기도 전에 `overrideMe`를 호출하기 때문이다.
    - 이 프로그램에서는 `final` 필드인 `instant`의 상태가 2가지가 된다(정상이라면 단 하나뿐이어야 한다).
    - `overrideMe`에서 `instant` 객체의 메서드를 호출하려 한다면 상위 클래스의 생성자가 `overrideMe`를 호출할 때 `NullPointerException`을 던지게 된다
    - `private, final, static` **메서드는 재정의 불가하다. 생성자에서 안심하고 호출해도 된다.**

- `Cloneable` 과 `Serializable` 인터페이스는 상속용 설계의 어려움을 한층 더해준다.
    - 둘 중 하나라도 구현한 클래스를 상속할 수 있게 설계 하는 것은 일반적으로 좋지 않은 선택이다. (하위 클래스를 잘 확장하려는 것은 프로그래머에게 부담…)
    - 물론 사용을 원한다면 구현하도록 하는 방법이 있다. (아이템13, 86)
    - `Serializable` 을 구현한 상속용 클래스가 `readResolve`나 `writeReplace` 메서드를 갖는다면 이 메서드들은 `private` 이 아닌 `protected`로 선언되어야 한다.
        - `private` 로 선언하면 하위 클래스에서 무시됨
- `clone`과 `readObject` 메서드
    - `clone` 과 `readObject` 메서드는 생성자와 비슷한 효과를 낸다. (새로운 객체를 생성)
      따라서 상속용 클래스에서 `Cloneable` 이나 `Serializable` 을 구현할지 정해야 한다면, 이들을 구현 할 때 따르는 제약도 생성자와 비슷하다.
    - 즉, `clone` 과 `readObject` 모두 직접적으로든 간접적으로든 재정의 가능 메서드를 호출해서는 안 된다.
    - `readObject` 의 경우 하위 클래스의 상태가 미처 다 역직렬화되기 전에 재정의한 메서드부터 호출하게 된다.
    - `clone` 의 경우 하위 클래스의 `clone` 메서드가 복제본의 상태를 (올바른 상태로) 수정하기 전에 재정의한 메서드를 호출한다.
        - 특히 `clone`이 잘못되면 복제본 뿐만 아니라 원본 객체에도 피해를 줄 수 있다.

4. 주의 할 점
    - 클래스를 상속용으로 설계하려면 엄청난 노력이 들고 해당 클래스에 안기는 제약도 상당함을 알았다.

- 추상 클래스나 인터페이스의 골격 구현 (아이템 20) 처럼 상속을 허용하는게 맞는 상황이 있고, 불변 클래스(아이템 17) 처럼 명백히 잘못된 상황이 있다.
- **일반적인 구체 클래스의 상황**은 `final` 도 아니고 상속용으로 설계되지도 않았고 문서화되지도 않았다. **그대로 두면 위험한 상황이다.** 클래스에 변화가 생길 때마다 하위 클래스를 오작동하게 만들 수 있기 때문이다.
  - 제일 좋은 방법으로는 상속용 클래스를 설계하지 않는 것이다.

5. 상속 금지
   a. 첫 번째는 클래스를 `final` 로 선언한다.
   b. 두 번째는 모든 생성자를 `private`, `package-private` 로 선언하고 `public` 정적 팩터리를 만드는 것이다.
   c. 핵심 기능을 정의한 인터페이스가 있고, 클래스가 그 인터페이스를 구현 했다면 상속을 금지해도 개발하는 데 아무런 어려움이 없을 것이다. `Set, Map, List` 가 좋은 예이다.
   d. 래퍼 클래스 패턴 (아이템 18)도 역시 기능을 확대(증강: 주관. 책에서 표현이 애매한 것 같다.)할 때 상속 대신 쓸 수 있는 더 나은 대안이다.
6. 상속을 반드시 허용해야 한다면
   - 클래스 내부에서는 재정의 가능 메서드를 사용하지 않게 만들고 이 사실을 문서로 남긴다.
     재정의 가능 메서드를 호출하는 자기사용 코드를 완벽히 제거해야 한다.
     이렇게 상속한다면, 그렇게 위험하지 않을 것이고, 매서드를 재정의해도 다른 메서드의 동작에 아무런 영향이 없다.

- 정리
  - 상속용 클래스를 설계한다면, 클래스 내부에서 스스로를 어떻게 사용하는지(자기사용 패턴) 모두 문서로 남겨야 하며, 일단 문서화한 것은 그 클래스가 쓰이는 한 반드시 지켜야 한다.
  - 그러지 않으면 그 내부 구현 방식을 믿고 활용하던 하위 클래스를 오작동하게 만들 수 있다.
  - 다른 이가 효율 좋은 하위 클래스를 만들 수 있도록 일부 메서드를 `protected`로 제공해야 할 수도 있다.
  - 클래스를 확장할 명확한 이유가 없으면 상속을 금지하는 것이 낫다.
  - 상속을 금지하려면 클래스를 `final` 로 선언하거나 생성자 (`clone`, `readObject` 포함) 모두를 외부에서 접근 할 수 없도록 만들면 된다.

▼ 20. 추상 클래스 보다는 인터페이스를 우선해라.

- 인터페이스와 가상클래스 (책 이전 이야기)
  - 공통점

        - 메소드를 가지고 있어야 한다.
        - 인스턴스화 할 수 없다.
          (인터페이스 혹은 추상 클래스를 상속받아 구현한 구현체의 인스턴스를 사용해야 한다.)
        - 인터페이스와 추상클래스를 구현, 상속한 클래스는 추상 메소드를 반드시 구현하여야 한다.
    - 차이점

	추상 클래스 (abstract)	**인터페이스 (**interface)
사용 가능 변수	제한 없음	static final (상수)
사용 가능 접근 제어자	제한 없음 (public, private, protected, default)	public
사용 가능 메소드	제한 없음	abstract method, default method, static method, private method
상속 키워드	extends	implements, extends
다중 상속 가능 여부	불가능	가능 (클래스에 다중 구현, 인터페이스 끼리 다중 상속)

- Java 8 이후
    - 인터페이스도 `default` 메서드를 제공하게 되어, 인터페이스와 추상 클래스 모두 인스턴스 메서드를 구현 형태로 제공할 수 있다.
    - 이 둘의 가장 큰 차이
        - 추상 클래스가 정의한 타입을 구현하는 클래스는 반드시 추상 클래스의 하위 클래스가 되어야 한다는 점이다.
            - 자바는 단일 상속만 지원하니, 추상 클래스 방식은 새로운 타입을 정의하는 데, 커다란 제약이 생기는 셈이다.
        - 인터페이스는 선언한 메서드를 모두 정의하고 그 일반 규약을 잘 지킨 클래스라면 다른 어떤 클래스를 상속했든 같은 타입으로 취급된다.
- 기존 클래스에도 손쉽게 새로운 인터페이스를 구현해 넣을 수 있다.
    - `Comparable, Iterable, AutoCloseable` 인터페이스가 추가 될 때, 표준 라이브러리의 수많은 기존 클래스가 인터페이스를 구현한 채 릴리즈 되었다.
    - 반면, 기존 클래스 위에 새로운 추상 클래스를 끼워 넣기는 어려운 문제이다.
    - 두 클래스가 같은 추상 클래스를 확장하길 원한다면, 그 추상 클래스는 계층구조상 두 클래스의 공통 조상이어야 한다. 안타깝게도 이 방식은 클래스 계층구조에 커다란 혼란을 일으킨다. 새로 추가된 추상 클래스의 모든 자손이 이를 상속하게 되는 것이다. (그렇게 하는게 적절하지 않은 상황에서도 강제적)
- 인터페이스는 믹스인 정의에 안성맞춤이다.

    - 믹스인이란 클래스가 구현할 수 있는 타입으로, **믹스인을 구현한 클래스에 원래의 '주된 타입' 외에도 특정 선택적 행위를 제공한다고 선언하는 효과를 준다.**
        - `Comparable` 은 자신을 구현한 클래스의 인스턴스들끼리는 순서를 정할 수 있다고 선언하는 믹스인 인터페이스이다.
        - 이처럼 대상 타입의 주된 기능에 선택적 기능을 혼합(mix in) 한다고 해서 믹스인이라고 부른다.
    - 추상 클래스는 기존 클래스를 덧씌울 수 없고 두 부모 클래스를 섬길 수 없기 때문에, 클래스 계층 구조에는 믹스인을 삽입하기에 합리적인 위치가 없기 때문이다.
- 인터페이스로는 계층구조가 없는 타입 프레임워크를 만들 수 있다.
    - 작곡가, 가수, 작곡 겸 가수 인터페이스

```
public interface Singer {
    void sing();
}

public interface Songwriter {
    void compose();
}

public interface SingerSongwriter extends Singer, Songwriter {
    void strum();

    void actSensitive();
}
```

        - 같은 구조를 클래스로 만들려면 가능한 조합 전부를 각각의 클래스로 정의한 고도비만 계층구조가 만들어질 것이다.
        - 속성이 n개라면 지원해야 할 조합의 수는 2^n 개나 될 것이다. 이러한 현상을 조합 폭발(combinatorial explosion)이라 한다.
        - 거대한 클래스 계층구조에는 공통 기능을 정의해놓은 타입이 없으니, 자칫 매개변수 타입만 다른 메서드들을 수없이 많이 가진 거대한 클래스를 낳을 수 있다.
- 래퍼 클래스 (아이템 18) 과 함께 사용하면 인터페이스는 기능을 향상시키는 안전하고 강력한 수단이 된다.
    - 타입을 추상 클래스로 정의해두면 그 타입에 기능을 추가하는 방법은 상속뿐이다.
    - 상속해서 만든 클래스는 래퍼 클래스보다 활용도가 떨어지고 깨지기는 더 쉽다.
- 디폴트 메서드의 규약
    - 인터페이스의 메서드 중 구현 방법이 명백한게 있다면 디폴트 메서드로 구현한다.
    - 많은 인터페이스가 `equals, hashCode` 같은 `object`의 메서드를 정의하지만, 이들을 `default` 메서드로 제공하면 안된다.

    - 인터페이스는 인스턴스 필드를 가질 수 없고 `public` 이 아닌 정적 멤버도 가질 수 없다. (단 `private` 정적 메서드는 예외이다.)
    - 본인이 만든 인터페이스가 아니면 디폴트 메소드를 추가할 수 없다.
- 인터페이스와 골격 구현 클래스
    - 이 두 개를 함께 제공하는 식으로, 인터페이스와 추상 클래스의 장점 모두 취하는 방법도 있다.
    - 인터페이스로는 타입을 정의하고, 필요하다면 디폴트 메서드도 몇 개 제공한다.
    - 골격 구현 클래스는 나머지 메서드들까지 구현한다.
    - 이렇게 해두면, 단순히 골격 구현을 확장하는 것만으로 인터페이스를 구현하는 데 필요한 일이 대부분 완료된다. (**템플릿 메서드 패턴**)
    - 네이밍 관례
        - 인터페이스 이름이 `interface` 라면, 골격 구현 클래스(추상 클래스)는 `AbstractInterface`로 짓는다.
    - 골격 클래스는 추상 클래스처럼 구현을 도와주는 동시에, 추상 클래스로 타입을 정의할 때 오는 제약에서 자유롭다.
    - 골격 구현을 확장하는 것으로 인터페이스 구현은 거의 끝난다.
    - 구조상 골격 구현 클래스를 확장하지 못한다면 인터페이스를 직접 구현해야 한다.
        - 그래도 디폴트 메소드의 이점을 누릴 수 있다.
        - 골격 구현 클래스를 우회적으로 이용 가능하다.
            - 인터페이스를 구현한 클래스에서 해당 골격 구현을 확장한 `private` 내부 클래스를 정의하고, 각 메서드 호출을 내부 클래스의 인스턴스에 전달하는 것이다.
            - 래퍼 클래스(아이템 18)와 비슷한 방식으로, 시뮬레이트한 다중 상속이라 하며, 다중 상속의 많은 장점을 제공하며, 동시에 단점은 피하게 해준다.
- 골격 구현 작성법
    - 인터페이스를 잘 살펴 다른 메서드들의 구현에 사용되는 기반 메서드들을 선정한다.
    - 이 기반 메서드들은 골격 구현에서는 추상 메서드가 될 것이다.
    - 기반 메서드들을 사용해 직접 구현할 수 있는 메서드를 모두 디폴트 메서드로 제공한다.
        - 단, `equals, hashcode` 같은 `Object` 메서드는 디폴트 메서드로 제공하면 안된다.
    - 만약, 인터페이스의 메서드가 모두 기반 메서드와 디폴트 메서드가 된다면 골격 구현 클래스를 별도로 만들 필요가 없다.
    - 기반 메서드나 디폴트 메서드로 만들지 못한 메서드가 남아 있다면, 이 인터페이스를 구현하는 골격 구현 클래스를 하나 만들어 남은 메서드들을 작성해 넣는다.
    - 골격 구현 클래스에는 필요하면 `public` 이 아닌 필드와 메서드를 추가 해도 된다.

- EX 골격 구현 클래스 (`Map.Entry` 인터페이스)

```
public abstract class AbstractMapEntry<K, V> implements Map.Entry<K, V> {

    // 변경 가능한 엔트리는 이 메서드를 반드시 재정의해야 한다.
    @Override
    public V setValue(V value) {
        throw new UnsupportedOperationException();
    }

    // Map.Entry.equals의 일반 규약을 구현한다.
    @Override
    public boolean equals(Object obj) {
        if (obj == this) {
            return true;
        }
        if (!(obj instanceof Map.Entry)) {
            return false;
        }
        Map.Entry<?, ?> e = (Map.Entry) obj;
        return Objects.equals(e.getKey(), getKey()) && Objects.equals(e.getValue(),
getValue());
    }

    // Map.Entry.hashCode의 일반 규약을 구현한다.
    @Override
    public int hashCode() {
        return Objects.hashCode(getKey()) ^ Objects.hashCode(getValue());
    }

    @Override
    public String toString() {
        return getKey() + "=" + getValue();
    }
}
```

  - `getKey, getValue` - 확실한 기반 메서드, `setValue` - 선택적으로 기반 메서드
  - `Object` 메서드들은 디폴트 메서드로 제공하면 안 되므로, `equals, hashCode` 를 골격 구현 클래스에서 구현한다. `toString` 도 기반 메서드로 구현되었다.
  - **골격 구현은 기본적으로 상속**해서 사용하는 걸 가정함므로, 아이템 19에서 이야기된 설계 및 문서화 지침을 모두 따라야한다. **인터페이스에 정의한 디폴트 메서드든 별도의 추상 클래스든, 골격 구현은 반드시 그 동작 방식을 잘 정리해 문서로 남겨야 한다.**
- 단순 구현은 골격 구현의 작은 변종으로, 골격 구현과 같이 상속을 위해 인터페이스를 구현한 것이지만, 추상 클래스가 아니란 점에 다르다. (쉽게 말해 정상 동작하는 단순한 구현체) (책에서는 `AbstractMap` 의 내부 정적 클래스를 예로 듬)

- 정리
  - 일반적으로 다중 구현용 타입으로는 인터페이스가 적합
  - 복잡한 인터페이스라면 구현하는 수고를 덜어주는 골격 구현을 함께 제공하는 방법 고려

        - 골격 구현은 '가능한 한(인터페이스는 구현상의 제약 때문에, 골격 구현은 추상 클래스로 하는 경우가 더 흔하기 때문)' 인터페이스의 디폴트 메서드로 제공하며, 그 인터페이스를 구현한 모든 곳에서 활용하는 것이 좋다.

▼ 21. 인터페이스는 구현하는 쪽을 생각해 설계해라

- java 8 의 `default` 메서드
    - 자바 8 이전에는 기존 구현체를 깨뜨리지 않고는 인터페이스에 메서드를 추가할 방법이 없었다. 인터페이스에 메서드를 추가하면 보통은 컴파일 오류가 나는데, 추가된 메서드가 우연히 기존 구현체에 이미 존재할 가능성은 아무 낮기 때문이다.
    - 자바 7 까지는 모든 클래스가 현재의 인터페이스에 새로운 메서드가 추가될 일은 없다고 가정하고 작성되었기 때문에, 기존 구현체들이 매끄럽게 연동되리라는 보장되지 않는다.
    - 자바 8 이후로는 기존 인터페이스에 메서드를 추가할 수 있도록 디폴트 메서드가 추가되었다.
    - 디폴트 메서드를 선언하면, 그 인터페이스를 구현한 후 디폴트 메서드를 재정의하지 않는 모든 클래스에서 디폴트 구현이 쓰이게 된다.
        - 디폴트 메서드는 구현 클래스에 대해 아무것도 모른 채 합의 없이 무작정 '삽입'될 뿐이다.
    - 자바 8에서는 핵심 컬렉션 인터페이스들에 다수의 디폴트 메서드가 추가되었다. 주로 람다를 위함이였다.
    - 자바 라이브러리의 디폴트 메서드는 코드 품질이 높고 범용적이라 대부분의 상황에서 잘 동작하지만, 생각할 수 있는 모든 상황에서 불변식을 해치지 않는 디폴트 메서드를 작성하는 것은 어렵다.
    - 자바 8 - `Collection` 인터페이스에 추가된 `removeIf` 메서드

```
default boolean removeIf(Predicate<? super E> filter) {
    Objects.requireNonNull(filter);
    boolean result = false;
    for (Iterator<E> it = iterator(); it.hashNext();) {
        if (filter.test(it.next())) {
            it.remove();
            result = true;
        }
    }
    return result;
}
```

        - 범용적으로 잘 구현되었지만, 현존하는 모든 `Collection` 구현체와 잘 어우러지는 것은 아니므로 주의해야 한다. (ex `Apache Common SynchronizedCollection` (`Collection` 인터페이스를 구현한 래퍼클래스))
            - (2023.01.22) 현재 기준으로는 `removeIf` 가 구현되어 있는 것을 확인됨.

        - 자바 플랫폼 라이브러리에도 이런 문제를 예방하기 위해 일련의 조치를 시행했다.
            - 구현한 인터페이스의 디폴트 메서드를 재정의하고, 다른 메서드에서는 디폴트 메서드를 호출하기 전에 필요한 작업을 수행하도록 했다.
- 주의사항
    - 디폴트 메서드는 (컴파일에 성공하더라도)기존 구현체에 런타임 오류를 일으킬 수 있다.
    - 기존 인터페이스에 디폴트 메서드로 새 메서드를 추가하는 일은 꼭 필요한 경우가 아니면 피해야한다.
      추가하려는 디폴트 메서드가 기존 구현체들과 충돌하지는 않을지 심사숙고해야 함도 당연하다.
    - 반면, 새로운 인터페이스를 만드는 경우라면, 표준적인 메서드 구현을 제공하는데 아주 유용한 수단이며, 그 인터페이스를 더 쉽게 구현해 활용할 수 있게끔 해준다. (아이템 20)
    - **디폴트 메서드는 인터페이스로부터 메서드를 제거하거나 기존 메서드의 시그니처를 수정하는 용도가 아님을 명심해야 한다**. (이런 형태로 인터페이스를 변경하게 된다면 기존 클라이언트를 망가트리게 된다.)
- 정리
    - 디폴트 메서드라는 도구가 생겼어도 인터페이스를 설계할 때는 여전히 세심한 주의를 기울여야 한다.
    - 디폴트 메서드로 기존 인터페이스에 새로운 메서드를 추가하면 커다란 위험도 딸려온다.
    - 새로운 인터페이스라면 릴리즈 전에 반드시 테스트를 해야한다.
        - 수 많은 개발자가 각기 다른 방식으로 인터페이스를 구현할 것이니, 최소한 세 가지의 다른 방식으로 구현을 해봐야 한다.
        - 또한 각 인터페이스의 인스턴스를 다양한 작업에 활용하는 클라이언트도 여럿 만들어야한다.
    - 인터페이스를 릴지즈 한 후라도 결함을 수정하는게 가능한 경우도 있겠지만, 절대 그 가능성에 기대선 안된다.

▼ 22. 인터페이스는 타입을 정의하는 용도로만 사용해라

- **인터페이스는 자신을 구현한 클래스의 인스턴스를 참조할 수 있는 타입 역할을 한다.**
    - 클래스가 어떤 인터페이스를 구현한다는 것은 자신의 인스턴스로 무엇을 할 수 있는지를 클라이언트에 이야가 하는것임
    - 상수 인터페이스 (위 지침에 맞지 않는 예)

```
pulbic interface PhysicalConstants {
    static final double AVOGAROS_NUMBER = 6.022_140_857e23;
    static final double BOLTZMANN_CONSTANT = 1.380_648_52E-23;
```

```
    static final double ELECTRON_MASS = 9.109_383_56E-31;
}
```

- 메서드가 없으며, 상수를 뜻하는 static final 필드로만 가득 찬 인터페이스를 뜻한다.
- **안티패턴으로 인터페이스를 잘 못 사용한 것이다.**
- 클래스 내부에서 사용하는 상수는 외부 인터페이스가 아닌 내부 구현에 해당한다.
- 상수 인터페이스를 구현하는 것은 이 내부 구현을 클래스의 API 노출하는 행위다. 사용자에게 혼란을 주기도 하며, 클라이언트 코드가 내부 구현에 해당하는 이 상수들에 종속되게 한다.

- 상수를 공개할 용도
  - 특정 클래스나 인터페이스와 강하게 연관된 상수라면, 그 클래스나 인터페이스 자체에 추가 해야한다. (ex. 모든 숫자 기본 타입의 박싱 클래스 (`Integer - MIN_VALUE`))
  - 열거 타입으로 나타내기 적합한 상수라면 열거 타입으로 만들어 공개하면 된다. (아이템 34)
  - 위 내용에 해당하지 않는 내용이라면 인스턴스화 할 수 없는 유틸리티 클래스(아이템4)에 담아서 공개하자.

▼ 23. 태그 달린 클래스보다는 클래스 계층구조를 활용해라

- 태그 달린 클래스 (아직까지 이런 클래스를 본적도 만든적도 없다. - 책에서 나오는 내용임으로 일단 작성)
  - 두 가지 이상의 의미를 표현한 것

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

    final Shape shape;

    double length;
    double width;

    double radius;

    Figure(double radius) {
        shape = Shape.CIRCLE;
        this.radius = radius;
    }

    Figure(double length, double width) {
        shape = Shape.RECTANGLE;
        this.length = length;
        this.width = width;
    }

    double area() {
        switch(shape) {
            case RECTANGLE:
                return length * width;
```

```
            case CIRCLE:
                retrun Math.PI * (radius * radius);
            default:
                throw new AssertionError(shape);
        }
    }
}
```

    - 해당 클래스는 단점이 많다. 열거 타입 선언, 태그 필드, `switch` 문 등 쓸데 없는 코드가 많다.
    - 여러 구현이 한 클래스에서 혼합되어 가독성도 나쁘다.
    - `circle` 만 구현해서 사용하려는데, `length, width` 의 메모리도 함께 사용될 것이다.
    - 필드들을 `final` 로 선언하려면 해당 의미에 쓰이지 않는 필드들까지 생성자에서 초기화해야한다.
    - 생성자가 태그 필드를 설정하고 해당 의미에 쓰이는 데이터 필드들을 초기화하는 데 컴파일러가 도와줄 수 있는건 별로 없다.
      엉뚱한 필드를 초기화해도 런타임 후에야 문제가 드러난다.
    - 인스턴스의 타입만으로는 현재 나타내는 의미를 알 길이 전혀 없다.
    - 즉, 태그 달린 클래스는 장황하고 오류 내기 쉽고, 비효율적이다.
    - 태그 달린 클래스는 클래스 계층구조를 어설프게 흉내낸 아류일 뿐이다.

- 태그 달린 클래스를 클래스 계층구조로 변경하는 방법
    - `root` 가 될 추상 클래스를 정의
        - 태그 값에 따라 동작이 달라지는 메서드들을 루트 클래스의 추상 메서드로 선언
        - 태그 값에 상관없이 동작이 일정한 메서드들은 루트 클래스에 일반 메서드로 추가한다.
        - 모든 하위 클래스에서 공통으로 사용하는 데이터 필드들도 루트 클래스에 올린다.
    - 추상 클래스 확장한 구체 클래스를 정의
        - 구체 클래스를 의미별로 하나씩 정의한다.
        - 각 구체 클래스에 의미가 맞는 데이터 필드들을 넣는다.
        - 추상 클래스의 추상 메서드로 선언된 내용을 `override` 해서 사용한다.
    - 계층 구조 클래스

```
abstract class Figure {
    abstract double area();
}

class Circle extends Figure {
    final double radius;
    Circle(double radius) { this.radius = radius; }
    @Override
```

```
    double area() { return Math.PI * (radius * radius)); }
}

class Rectangle extends Figure {
    final double length;
    final double width;
    Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }
    @Override
    double area() { return length * width; }
}
```

- 위 태그 클래스와의 비교
  - 간결하고 명확하고, 쓸데 없는 코드들도 모두 사라짐
  - 각 의미를 독립된 클래스에 담아 관련 없던 데이터 필드를 모두 제거함.
    살아 남은 필드들은 모두 `final` 임
  - 생성자가 모든 필드를 남김없이 초기화하고 추상 메서드를 모두 구현 했는지 컴파일러가 확인해줌
  - 실수로 빼먹은 `case` 문 때문에 런타임 오류가 발생될 일도 없다.
  - 루트 클래스의 코드를 건들이지 않고도 다른 프로그래머들이 독립적으로 계층구조를 확장하고 함께 사용 가능.

▼ 24. 멤버 클래스는 되도록 static 으로 만들어라

- 중첩 클래스 (다른 클래스 안에 정의된 클래스) (nested class)

  중첩 클래스는 자신을 감싼 바깥 클래스에서만 쓰여야 하며, 그 외의 쓰임새가 있다면 톱 레벨 클래스로 만들어야 한다.

- 종류

```
class Outer{
  class NonStaticInner { ... } // 비정적 클래스
  static class StaticInner { ... } // 정적 클래스

    void method1(){
      class LocalInner { ... } // 지역 클래스
    }
}
```

  - 정적 멤버 클래스를 제외한 나머지는 내부(`inner`) 클래스이다.
  - 정적 멤버 클래스
    - 다른 클래스 안에 선언되고, 바깥 클래스의 `private` 멤버에도 접근 할 수 있다는 점만 제외하고는 일반 클래스와 똑같다.

- 다른 정적 멤버와 똑같은 접근 규칙을 적용받는다. 예컨대 `private` 으로 선언하면 바깥 클래스에서만 접근 할 수 있는 식이다.
- 흔히 바깥 클래스와 함께 쓰일 때만 유용한 `public` 도움 클래스로 쓰인다.
- `private` 정적 멤버 클래스는 흔히 바깥 클래스가 표현하는 객체의 한 부분(구성 요소)로 나타낼 때 쓴다.
- `hash map` 의 `static class Node<K,V> implements Map.Entry<K,V>` 를 참고.

```
public class HashMap<K,V> extends AbstractMap<K,V> {
    ...
    // 정적 멤버 클래스
    static class Node<K,V> implements Map.Entry<K,V> {
        final int hash;
        final K key;
        V value;
        ...
    }
    ...
    // 비정적 멤버 클래스
    final class KeySet extends AbstractSet<K> {
        ...
    }
}
```

- (비정적) 멤버 클래스
  - 정적 멤버 클래스와 구문 상 차이는 단지 `static` 유무의 차이 이지만, 의미 상 차이는 꽤 크다.
  - 비정적 멤버 클래스의 인스턴스는 바깥 클래스의 인스턴스와 암묵적으로 연결된다.
  - 그래서 비정적 멤버 클래스의 인스턴스 메서드에서 `정규화된 this` 를 사용해 바깥 인스턴스의 메서드를 호출 하거나, 바깥 인스턴스의 참조를 가져올 수 있다. (`정규화된 this` 란, `클래스명.this` 형태로 바깥 클래스의 이름을 명시하는 용법을 말한다.)
  - 따라서, **개념상 중첩 클래스의 인스턴스가 바깥 인스턴스와 독립적으로 존재할 수 있다면, 정적 멤버 클래스**로 만들어야 한다. **비정적 멤버 클래스는 바깥 인스턴스 없이는 생성할 수 없기 때문이다.**
  - 비정적 멤버 클래스의 인스턴스와 바깥 인스턴스 사이의 관계는 멤버 클래스가 인스턴스화 될때 확립되며, 더 이상 변경 할 수 없다.
    이 관계는 바깥 클래스의 인스턴스 메서드에서 비정적 멤버 클래스의 생성자를 호출할 때 자동으로 만들어지는 게 보통이지만, 드물게 직접 바깥 인스턴스의 클래스.new Member Class(args)를 호출해 수동으로 만들기도 한다.
  - 이 관계 정보는 비정적 멤버 클래스의 인스턴스 안에 만들어져 메모리 공간을 차지하며, 생성 시간도 더 걸린다.
  - 어댑터를 정의할 때 자주 쓰인다. (어떤 클래스의 인스턴스를 감싸 마치 다른 클래스의 인스턴스처럼 보이게 하는 뷰로 사용하는 것)

`Map` 인터페이스의 구현체들은 보통 (`keySet, entrySet, values` 메서드가 반환하는) 자신의 컬렉션 뷰를 구현할 때 비정적 멤버 클래스를 사용한다.

- 비정적 멤버 클래스의 흔한 쓰임 - **자신의 반복자 구현**

```
public class MySet<E> extends AbstractSet<E> {
    @Override
    public Iterator<E> iterator() {
        return new MyIterator();
    }
    private class MyIterator implements Iterator<E> {
        ...
    }
}
```

- **멤버 클래스에서 바깥 인스턴스에 접근할 일이 없다면, 무조건 `static` 을 붙여서 정적 멤버 클래스로 만들자.**
- `static` 을 생략하면 바깥 인스턴스로 의 숨은 외부 참조를 갖게 된다. 이 참조를 저장하려면, 시간과 공간이 소비되며, 가비지 컬렉션이 바깥 클래스의 인스턴스를 수거하지 못하는 메모리 누수가 일어날 수 있다. (아이템 7)
- 멤버 클래스가 공개된 클래스의 `public` 이나, `protected` 멤버라면 정적이냐 아니냐는 두배로 중요해진다. 멤버 클래스 역시 공개 API 가 되니, 혹시라도 향후 릴르스에서 `static` 을 붙이면 하위 호환성이 깨진다.

- 익명 클래스
  - 바깥 클래스의 멤버도 아니다. 멤버와 달리 쓰이는 시점에 선언과 동시에 인스턴스가 만들어진다. 코드의 어디서든 만들 수 있다.
  - 오직 비정적인 문맥에서 사용될 때만 바깥 클래스의 인스턴스를 참조할 수 있다.
  - 정적 문맥에서라도 상소 변수 이외의 정적 멤버는 가질 수 없다.
  - 응용하는데, 제약이 많은 편이며, 선언한 지점에서만 인스턴스를 만들 수 있고 `instanceof` 검사나 클래스의 이름이 필요한 작업은 수행 할 수 없다.
  - 인터페이스를 구현하는 동시에 다른 클래스를 상속할 수도 없다.
  - 익명 클래스를 사용하는 클라이언트는 그 익명 클래스가 상위 타입에서 상속한 멤버 외에는 호출할 수 없다.
  - 익명 클래스는 표현식 중간에 등장하므로 (10줄 이하로) 짧지 않으면 가독성이 떨어진다.
    람다를 지원하기 전에는 즉석에서 작은 함수 객체나 처리 객체를 만드는 데 익명 클래스를 주로 사용했다. (지금은 람다가 그 자리를 사용한다 (아이템 42))
- 지역 클래스
  - 지역변수를 선언 할 수있는 곳이면, 실질적으로 어디서든 선언할 수있으며, 유효 범위도 지역변수와 같다.

        - 멤버 클래스 처럼 이름이 있고 반복해서 사용 할 수 있다.
        - 익명 클래스처럼 비정적 문맥에서 사용될 때만, 바깥 인스턴스를 참조할 수 있으며, 정적 멤버는 가질 수 없으며, 가독성을 위해 짧게 작성해야 한다.
- 정리
    - 메서드 밖에서도 사용해야 하거나, 메서드 안에 정의하기가 너무 길다면 멤버 클래스로 만든다.
      멤버 클래스의 인스턴스 각각이 바깥 인스턴스를 참조한다면 비정적으로, 그렇지 않으면 정적으로 만든다.
    - 중첩 클래스가 한 메서드 안에서만 쓰이면서,

      그 인스턴스를 생성하는 지점이 단 한 곳이고 해당 타입으로 쓰기에 적합한 클래스나 인터페이스가 이미 있다면 익명 클래스로 만들고, 그렇지 않다면 지역클래스로 만들자.

▼ 25. 톱레벨 클래스 (톱레벨 인터페이스) 는 한 파일에 하나만 담으라

- 톱레벨 클래스를 여러 개 선언하더라도, 아무런 득이 없을 뿐더러 심각한 위험을 감수된다.
    - 한 클래스를 여러 가지로 정의할 수 있으며, 그중 어느 것을 사용할지는 어느 소스 파일을 먼저 컴파일하냐에 따라 달라지기 때문이다.
    - 책의 예제
    - Utensil 파일

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

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

    - Dessert 파일

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

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

        - 우연히 위 와 같은 클래스가 각각 명으로 Utensil, Dessert 라는 파일로 두 개 만들어졌다.
        - 컴파일러에 어떤 소스를 먼저 건네느냐에 따라 동작이 달라지는 문제가 생긴다.

        - javac Main.java나 javac Main.java Utensil.java 명령으로 실행하면 의도한 pancake를 출력함
        - 그러나 javac Dessert.java Main.java 명령으로 컴파일 하면 potpie를 출력함
    - 단순히 톱레벨 클래스들(Utensil과 Dessert)을 서로 다른 소스 파일로 분리하면 그만이다.
    - 굳이 여러 톱레벨 클래스를 한 파일에 담고 싶다면 **정적 멤버 클래스** (아이템 24)를 사용하는 방법을 고민해볼 수 있다