<4장> 클래스와 인터페이스

> 추상화의 기본 단위인 클래스와 인터페이스는 자바 언어의 심장과도 같다. 그래서 자바 언어에는 클래스와 인터페이스 설계에 사용하는 강력한 요소가 많이 있다.
>
> 이번 장에서는 이런 요소를 적절히 활용하여 클래스와 인터페이스를 쓰기 편하고, 견고하며, 유연하게 만드는 방법을 안내한다.

▼ [아이템 15] 클래스와 멤버의 접근 권한을 최소화하라

✅ 잘 설계된 컴포넌트는 모든 내부 구현을 완벽히 숨겨, 구현과 API를 깔끔히 분리한다 (정보 은닉, 캡슐화)

정보 은닉의 장점

- 시스템 개발 속도를 높인다
  - 여러 컴포넌트를 병렬로 개발할 수 있다
- 시스템 관리 비용을 낮춘다
  - 각 컴포넌트를 더 빨리 파악하여 디버깅할 수 있다
  - 다른 컴포넌트로 교체하는 부담이 적다
- 정보 은닉 자체가 성능을 높여주지는 않지만, 성능 최적화에 도움을 준다
  - 완성된 시스템을 프로파일링해 최적화할 컴포넌트를 정한 다음(아이템 67), 다른 컴포넌트에 영향을 주지 않고 해당 컴포넌트만 최적화할 수 있기 때문이다
- 소프트웨어 재사용성을 높인다
  - 외부에 거의 의존하지 않고 독자적으로 동작할 수 있는 컴포넌트라면 그 컴포넌트와 함께 개발되지 않은 낯선 환경에서도 유용하게 쓰일 가능성이 크기 때문이다
- 큰 시스템을 제작하는 난이도를 낮춰준다
  - 시스템 전체가 아직 완성되지 않은 상태에서도 개별 컴포넌트의 동작을 검증할 수 있기 때문이다

정보 은닉의 원칙

✅ 접근 제한자를 제대로 활용하는 것이 정보 은닉의 핵심이다

☝🏻 (가장 바깥이라는 의미의)탑레벨 클래스와 인터페이스에 부여할 수 있는 접근 수준은 package-private과 public 두 가지다

모든 클래스와 멤버의 접근성을 가능한 한 좁혀야 한다

- 패키지 외부에서 쓸 이유가 없다면 package-private으로 선언하자
  - API가 아닌 내부 구현이 되어 클라이언트에 피해 없이 언제든 수정 가능하다
  - public으로 선언한다면 API가 되므로 하위 호환을 위해 영원히 관리해주어야 한다
- 한 클래스에서만 사용하는 package-private 탑레벨 클래스나 인터페이스는 이를 사용하는 클래스 안에 private static으로 중첩시키자(아이템 24)
- public일 필요가 없는 클래스의 접근 수준을 package-private 탑레벨 클래스로 좁히자

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

접근 수준	설명
private	멤버를 선언한 탑레벨 클래스에서만 접근할 수 있다.
package-private	멤버가 소속된 패키지 안의 모든 클래스에서 접근할 수 있다. 접근 제어자르 ㄹ명시하지 않았을 때 적용되는 패키지 접근 수준이다. (단, 인터페이스의 멤버는 기본적으로 public이 적용된다)
protected	package-private의 접근 범위를 포함하며, 이 멤버를 선언한 클래스의 하위 클래스에서도 접근할 수 있다.
public	모든 곳에서 접근할 수 있다.

- 클래스의 공개 API를 세심히 설계 후, 그 외의 모든 멤버를 private으로 만들자
- 오직 같은 패키지의 다른 클래스가 접근해야 하는 멤버에 한해 package-private으로 만들자

    - 단, Serializable을 구현한 클래스에서는 그 필드들도 의도치 않게 공개 API가 될 수도 있다(아이템 86, 87)
- protected 멤버의 수는 적을수록 좋다
    - public 클래스의 protected 멤버는 공개 API이므로 영원히 지원돼야 한다
    - 내부 동작 방식을 API 문서에 적어 사용자에게 공개해야 할 수도 있다 (아이템 19)
- 그러나 상위 클래스의 메소드를 재정의(Override)할 때는 그 접근 수준을 상위 클래스에서보다 좁게 설정할 수 없다 (아이템 10, 리스코프 치환 원칙)
    - 클래스가 인터페이스를 구현(implements)하는 것은 이 경우의 특별한 예로, 이 때 클래스는 인터페이스가 정의한 모든 메소드를 public으로 선언해야 한다

public 클래스의 인스턴스 필드는 되도록 public이 아니어야 한다(아이템 16)

- 필드가 가변 객체를 참조하거나, final이 아닌 인스턴스 필드를 public으로 선언하면 그 필드와 관련된 모든 것이 불변식을 보장받을 수 없게 된다
- 필드가 수정될 때 (락 획득 같은) 다른 작업을 할 수 없게 된다
    - 즉, 일반적으로 스레드 안전하지 않다
    - final이면서 불변 객체를 참조하더라도 문제는 여전히 남는다
    - 단, 해당 클래스가 표현하는 추상 개념을 완성하는데 꼭 필요한 구성요소로써의 상수라면 `public static final` 필드로 공개해도 좋다
        - 이런 경우에는 변수명으로 관례상 대문자와 밑줄(_)을 사용한다(아이템 68)
        - 또한 반드시 기본 타입 값이나 불변 객체를 참조해야 한다(아이템 17)

클래스에서 public static final 배열 필드를 두거나 이 필드를 반환하는 접근자 메소드를 제공해서는 안 된다

- 길이가 0이 아닌 배열은 모두 변경 가능하기 때문이다

```
//보안 허점이 숨어 있다.
public static final Thing[] VALUES = { ... };
```

✅ 해결 방법

1. public 배열을 private으로 만들고 public 불변 리스트 추가하기

```
private static final Thing[] PRIVATE_VALUES = { .
public static final List<Thing> VALUES =
        Collections.unmodifiableList(Arrays.asLis
```

2. public 배열을 private으로 만들고 그 복사본을 반환하는 public 메소드 추가하기

```
private static final Thing[] PRIVATE_VALUES = { .
public static final Thing[] values(){
        return PRIVATE_VALUES.clone();
}
```

▼ [아이템 16] public 클래스에서는 public 필드가 아닌 접근자 메서드를 사용하라

- 인스턴스 필드를 모아놓는 목적밖에 없는 클래스는 캡슐화의 이점을 제공하지 못한다 (아이템 15)

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

- 따라서 패키지 바깥에서 접근할 수 있는 클래스라면 접근자를 제공해야 한다
  - getter/setter를 사용한다

```
class Point {
        private double x;
        private double 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; }
}
```

✅ 단, package-priavte 혹은 private 중첩 클래스라면, 데이터 필드를 노출해도 문제 없다

▼ [아이템 17] 변경 가능성을 최소화하라

📕 **불변 클래스**

인스턴스의 내부 값을 수정할 수 없는 클래스

불변 인스턴스에 간직된 정보는 고정되어 객체가 파괴되는 순간까지 절대 달라지지 않는다

가변 클래스보다 설계, 구현, 사용하기 쉬우며, 오류가 생길 여지가 적고 안전하다

ex) 박싱 클래스, String, BigInteger, BigDecimal, etc...

불변 클래스를 만드는 규칙

1. 객체의 상태를 변경하는 메소드(변경자)를 제공하지 않는다

2. 클래스를 확장할 수 없도록 한다

- 상속을 막는 대표적인 방법은 클래스를 final로 선언하는 것이다
- 다른 방법은 후술되어있다

3. 모든 필드를 final로 선언한다

- 시스템이 강제하는 수단을 이용해 설계자의 의도를 명확히 드러내는 방법이다
- 새로 생성된 인스턴스를 동기화 없이 다른 스레드로 건네도 문제없이 동작하게끔 보장하는 데도 필요하다

4. 모든 필드를 private으로 선언한다

- 필드가 참조하는 가변 객체를 클라이언트에서 직접 접근해 수정하는 일을 막아준다

- 기본 타입 필드나 불변 객체를 참조하는 필드를 public final로 선언해도 불변 객체가 되지만, 다음 릴리스에서 내부 표현을 바꾸지 못하므로 비권장된다

5. 자신 외에는 내부의 가변 컴포넌트에 접근할 수 없도록 한다

- 클래스에 가변 객체를 참조하는 필드가 하나라도 있다면 클라이언트에서 그 객체의 참조를 얻을 수 없도록 해야 한다
- 생성자, 접근자, readObject 메소드(아이템 88) 모두에서 방어적 복사를 수행하라

불변 객체의 장점

1. 불변 객체는 단순하다

- 불변 객체는 생성된 시점의 상태를 파괴될 때까지 그대로 간직한다

2. 불변 객체는 근본적으로 스레드 안전하여 따로 동기화할 필요가 없다

- 여러 스레드가 동시에 사용해도 절대 훼손되지 않다
    - 스레드 안전하게 만드는 가장 쉬운 방법이다
- 따라서 **안심하고 공유할 수 있다**

3. 불변 객체는 자유롭게 공유할 수 있음은 물론, 불변 객체끼리는 내부 데이터를 공유할 수 있다

- 예시로, BigInteger의 negate 메소드는 부호만 다른 새로운 BigInteger를 생성한다. 그러나 가변인 int 배열 멤버 변수를 복사하지 않고 공유한다.

4. 객체를 만들 때 다른 불변 객체들을 구성요소로 사용하면 이점이 많다

- 값이 바뀌지 않는 구성요소들로 이루어진 객체라면 그 구조가 아무리 복잡하더라도 불변식을 유지하기 훨씬 수월하기 때문이다
- 예시로, 불변 객체는 맵의 키와 집합(Set)의 원소로 쓰기 안성맞춤이다

5. 불변 객체는 그 자체로 실패 원자성을 제공한다(아이템 76)

- 상태가 절대 변하지 않으니 잠깐이라도 불일치 상태에 빠질 가능성이 없다

> 📕 **실패 원자성(failure atomicity)**
>
> '메소드에서 예외가 발생한 후에도 그 객체는 여전히 (메소드 호출 전과 똑같은) 유효한 상태여야 한다'는 성질. 불변 객체의 메소드는 내부 상태를 바꾸지 않아 이 성질을 만족한다

불변 클래스의 단점

1. 값이 다르면 반드시 독립된 객체로 만들어야 한다

- 따라서 값의 가짓수가 많다면 이들을 모두 만드는 데 큰 비용을 치러야 한다

> ☝🏻 단, 클라이언트들이 원하는 복잡한 연산들을 정확히 예측할 수 있다면 package-private의 **가변 동반 클래스**만으로 충분하다.
>
> ex) String - StringBuilder/StringBuffer

BigInteger와 BigDecimal의 문제점

- 이들을 설계할 당시엔 불변 객체가 사실상 final이어야 한다는 생각이 널리 퍼지지 않아 각 메소드들을 모두 재정의할 수 있게 설계되었다
  - 이로 인해 하위 호환성이 발목을 잡아 아직도 수정하지 못했다
- 때문에 BigInteger나 BigDecimal의 인스턴스를 인수로 받는다면, 해당 인스턴스가 '진짜' BigInteger(BigDecimal)인지 확인해야 한다.
  - 신뢰할 수 없다면, 아래와 같이 방어적으로 복사해 사용해야 한다

```
public static BigInteger safeInstance(BigInteger val){
        return val.getClass() == BigInteger.class ?
                        val : new BigInteger(val.toByte
}
```

정리

1. 클래스는 꼭 필요한 경우가 아니라면 불변이어야 한다

- getter가 있다고 해서 무조건 setter를 만들지는 말자
- String과 BigInteger처럼 무거운 값 객체도 불변으로 만들 수 있는지 고심해야 한다
  - 성능때문에 어쩔 수 없다면(아이템 67) 불변 클래스와 쌍을 이루는 가변 동반 클래스를 public 클래스로 제공하도록 하자!

2. 불변으로 만들 수 없는 클래스라도 변경할 수 있는 부분을 최소한으로 줄이자

- 모든 클래스를 불변 클래스로 만들 수는 없다

- 그러나 객체가 가질 수 있는 상태의 수를 줄이면 그 객체를 예측하기 쉬워지고 오류가 생길 가능성이 줄어든다
- 그러니 꼭 변경해야 할 필드를 제외한 나머지 모두를 final로 선언하자
    - 다른 합당한 이유가 없다면 모든 필드는 private final이어야 한다

3. 생성자는 불변식 설정이 모두 완료된, 초기화가 완벽히 끝난 상태의 객체를 생성해야 한다

- 확실한 이유가 없다면 생성자와 정적 팩토리 외에는 그 어떤 초기화 메소드도 public으로 제공해서는 안된다

▼ [아이템 18] 상속보다는 컴포지션을 사용하라

상속의 문제점

메소드 호출과 달리 상속은 캡슐화를 깨뜨린다

- 상위 클래스가 어떻게 구현되느냐에 따라 하위 클래스의 동작에 이상이 생길 수 있다
    - 상위 클래스는 릴리스마다 내부 구현이 달라질 수 있으며, 그 여파로 건드리지 않은 하위 클래스가 오동작할 수 있다
    - 상위 클래스에 새로운 메소드가 생기면 하위 클래스에서 재정의하지 못해 (허용되지 않은 원소를 추가하는 등) 문제가 발생할 수 있다

해결 방법

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

- 이러한 설계를 **컴포지션(composition; 구성)**이라고 한다
- 새 클래스의 인스턴스 메소드들은 (private 필드로 참조하는) 기존 클래스의 대응하는 메소드를 호출해 결과를 반환한다
    - 이 방식을 전달(following)이라고 하며, 새 클래스의 메소드들을 전달 메소드(forwarding method)라 한다

상속의 주의사항

1. 상속은 반드시 하위 클래스가 상위 클래스의 '진짜' 하위 타입인 상황에서만 쓰여야 한다

- 클래스 B가 클래스 A와 is-a 관계일 때만 클래스 A를 상속해야 한다

    - “B가 정말 A인가?”라는 답변에 “그렇다”라는 확신이 있어야 한다
    - 아니라면, A를 private 인스턴스로 두고 A와는 다른 API를 제공해야 하는 상황이 대다수이다
- 컴포지션을 써야 할 상황에서 상속을 사용하는 것은 내부 구현을 불필요하게 노출하는 꼴이다

2. 확장하려는 클래스의 API에 아무런 결함이 없는지, 있다면 생성할 클래스의 API까지 전파되도 괜찮은지를 판단하자

- 상속은 상위 클래스의 API를 ‘그 결함까지도’ 그대로 승계한다

▼ [아이템 19] 상속을 고려해 설계하고 문서화하라. 그러지 않았다면 상속을 금지하라

상속을 고려한 설계와 문서화

1. 상속용 클래스는 재정의할 수 있는 메소드들을 내부적으로 어떻게 이용하는지(자기사용) 문서로 남겨야 한다

- 클래스의 API로 공개된 메소드에서 클래스 자신의 또 다른 메소드를 호출할 때, 호출되는 메소드가 재정의 가능 메소드라면 그 사실을 메소드의 API 설명에 적시해야 한다
    - 어떤 순서로 호출하는지, 각각의 호출 결과가 이어지는 처리에 어떤 영향을 주는지도 담아야 한다(‘재정의 가능’이란 public과 protected 메소드 중 final이 아닌 모든 메소드를 뜻한다)
- 즉, 재정의 가능한 메소드를 호출할 수 있는 모든 상황을 문서로 남겨야 한다
    - 백그라운드 스레드나 정적 초기화 과정에서도 호출이 일어날 수 있기 때문이다

> ☝🏻 API 문서의 메소드 설명 끝에 종종 “Implementation Requirements”로 시작하는 절을 볼 수 있는데, 그 메소드의 내부 동작 방식을 설명하는 곳이다.
>
> 이 절은 메소드 주석에 `@implSpec` 태그를 붙여주면 자바독 도구가 생성해준다.

> ⚠️ 그러나 “좋은 API 문서란 ‘어떻게’가 아닌’ 무엇’을 하는지를 설명해야 한다”라는 격언과는 대치된다. 상속이 캡슐화를 해치기 때문에 일어나는 현실이다.

2. 효율적인 하위 클래스를 쉽게 만들기 위해, 클래스 내부 동작 과정 중간에 끼어들 수 있는 훅(hook)을 잘 선별해 protected 메소드 형태로 공개해야 할 수도 있다

- 내부 매커니즘을 문서로 남기는 것 만이 상속을 위한 설계의 전부는 아니다
- 하위 클래스의 메소드를 고성능으로 만들기 위해 protected 형태의 훅 메소드를 고려하자
    - ex) java.util.AbstractList의 removeRange 메소드는 fromIndex부터 toIndex까지의 모든 원소를 리스트에서 제거하는 protected 메소드이다. 이 메소드를 재정의하면 해당 리스트와 부분리스트의 clear 연산 성능을 크게 개선할 수 있다.

⚠️ 상속용 클래스를 시험하는 방법은 직접 하위 클래스를 만들어 보는 것이 '유일'하다. 경험상 이러한 검증에는 하위 클래스 3개 정도가 적당하다. 그리고 이 중 하나 이상은 제3자가 작성해봐야 한다.

3. 상속용으로 설계한 클래스는 배포 전에 반드시 하위 클래스를 만들어 검증해야 한다

4. 상속용 클래스의 생성자는 직접적으로든 간접적으로든 재정의 가능 메소드를 호출해서는 안된다

- 이 규칙을 어기면 프로그램이 오동작할 것이다
    - 상위 클래스의 생성자가 하위 클래스의 생성자보다 먼저 실행되므로 하위 클래스에서 재정의한 메소드가 하위 클래스의 생성자보다 먼저 호출된다
    - 이 때 재정의한 메소드가 하위 클래스의 생성자에서 초기화하는 값에 의존한다면 의도대로 동작하지 않을 것이다

☝🏻 private, final, static 메소드는 재정의가 불가능하므로 생성자에서 안심하고 호출해도 된다

5. Cloneable과 Serializable 인터페이스 중 하나라도 구현한 클래스를 상속할 수 있게 설계하는 것은 일반적으로 좋지 않다

- 그 클래스를 확장하려는 프로그래머에게 엄청난 부담을 지운다
- clone과 readObject 모두 직간접적으로든 재정의 가능 메소드를 호출해서는 안된다
- Serializable을 구현한 상속용 클래스가 readResolve나 writeReplace 메소드를 갖는다면, 이 메소드들은 private이 아닌 protected로 선언해야 한다

- private으로 선언한다면 하위 클래스에서 무시되기 때문이다
- 상속을 허용하기 위해 내부 구현을 클래스 API로 공개하는 예 중 하나이다

정리

1. 클래스를 상속용으로 설계하려면 엄청난 노력이 들고, 그 클래스에 안기는 제약도 상당하다

- 추상 클래스나 인터페이스의 골격 구현(아이템 20)처럼 상속을 허용하는게 명백히 정당한 상황이 있고, 불변 클래스(아이템 17)처럼 명백히 잘못된 상황이 있다

2. 상속용으로 설계되지 않은 클래스의 상속을 금지해라

- 상속을 금지하는 방법은 두 가지다.
  1. 클래스를 final로 선언하기
  2. 모든 생성자를 private이나 package-private으로 선언하고 public 정적 팩토리를 만들어 주기

3. 만약 표준 인터페이스가 없더라도 상속을 허용해야겠다면, 클래스 내부에서 재정의 가능 메소드를 사용하지 않게 만들고, 문서화해라

- 재정의 가능 메소드를 호출하는 자기 사용 코드를 완벽히 지워라

▼ [아이템 20] 추상 클래스보다는 인터페이스를 우선하라

인터페이스의 장점

1. 기존 클래스에도 손쉽게 새로운 인터페이스를 구현해넣을 수 있다

- 인터페이스가 요구하는 메소드를 (아직 없다면) 추가하고, 클래스 선언에 implements 구문만 추가하면 끝이다
- 반면 기존 클래스 위에 새로운 추상 클래스를 끼워넣기는 일반적으로 어렵다

2. 인터페이스는 믹스인(mixin) 정의에 안성맞춤이다

- 믹스인이란 클래스가 구현할 수 있는 타입으로, 믹스인을 구현한 클래스에 원래의 '주된 타입' 외에도 특정 선택적 행위를 제공한다고 선언하는 효과를 준다

3. 인터페이스로는 계층구조가 없는 타입 프레임워크를 만들 수 있다

- 타입을 계층적으로 정의하면 수많은 개념을 구조적으로 잘 표현할 수 있지만, 현실에는 계층을 엄격히 구분하기 어려운 개념도 있다

4. 래퍼 클래스 관용구(아이템 18)와 함께 사용하면 인터페이스는 기능을 향상시키는 안전하고 강력한 수단이 된다

- 타입을 추상 클래스로 정의해두면 그 타입에 기능을 추가하는 방법은 상속뿐이다
  - 이는 래퍼 클래스보다 활용도가 떨어지고 깨지기 쉽다

인터페이스의 메소드

1. 인터페이스의 메소드 중 구현 방법이 명백한 것이 있다면, 그 구현을 디폴트 메소드로 제공하자

- 이 기법의 예로 Collection의 removeIf 메소드가 있다

2. 인터페이스와 추상 골격 구현(skeletal implementation) 클래스를 함께 제공하는 방법도 있다 (템플릿 메소드 패턴)

- 이는 인터페이스와 추상 클래스의 장점을 모두 취한다
  - 인터페이스로는 타입을 정의하고, 필요시 디폴트 메소드 몇 개도 함께 제공한다
  - 골격 구현 클래스는 나머지 메소드까지 구현한다

> ☝🏻 관례상 인터페이스 이름이 *Interface*라면 그 골격 구현 클래스 이름은 *AbstractInterface*로 짓는다.
>
> 좋은 예로, Collection 프레임워크의 AbstractionCollection, AbstractSet, AbstractList, AbstractMap 각각이 바로 핵심 컬렉션 인터페이스의 골격 구현이다

3. 단순 구현(simple implementation)은 골격 구현의 작은 변종이다

- AbstractMap.SimpleEntry가 좋은 예이다
- 골격 구현과 같이 상속을 위해 인터페이스를 구현한 것이지만, 추상클래스가 아니다
  - 동작하는 가장 단순한 구현이다
  - 그대로 써도 되고, 필요에 맞게 확장해도 된다

☝🏻 일반적으로 다중 구현용 타입으로는 인터페이스가 가장 적합하다. 복잡한 인터페이스라면 구현하는 수고를 덜어주는 골격 구현을 함께 제공하는 방법을 꼭 고려해보자

골격 구현은 '가능한 한' 인터페이스의 디폴트 메소드로 제공하여 그 인터페이스를 구현한 모든 곳에서 활용하도록 하자 (인터페이스에 걸려 있는 구현상의 제약 때문에 골격 구현을 추상 클래스로 제공하는 경우가 흔하긴 하다)

▼ [아이템 21] 인터페이스는 구현하는 쪽을 생각해 설계하라

☝🏻 자바 8에서는 핵심 컬렉션 인터페이스들에 다수의 디폴트 메소드가 추가되었다. 주로 람다(7장 참조)를 활용하기 위해서이다.

디폴트 메소드의 주의사항

1. 생각할 수 있는 모든 상황에서 불변식을 해치지 않는 디폴트 메소드를 작성하기는 어렵다

2. 디폴트 메소드는 (컴파일에 성공하더라도) 기존 구현체에 런타임 오류를 일으킬 수 있다

- 추가하려는 디폴트 메소드가 기존 구현체들과 충돌하지는 않을지 고려해야 한다
- 디폴트 메소드는 인터페이스로부터 메소드를 제거하거나 기존 메소드의 시그니처를 수정하는 용도가 아님을 명심해야 한다

정리

인터페이스를 설계할 때는 세심한 주의를 기울여야 한다

- 디폴트 메소드로 기존 인터페이스에 새로운 메소드를 추가하면 커다란 위험도 딸려온다
- 새로운 인터페이스라면 릴리스 전에 반드시 테스트를 거쳐야 한다
    - 서로 다른 방식으로 최소한 세 가지는 구현해봐야 한다
    - **인터페이스를 릴리스 한 후라도 결함을 수정하는 게 가능한 경우도 있겠지만, 그 가능성에 기대서는 안된다**

▼ [아이템 22] 인터페이스는 타입을 정의하는 용도로만 사용하라

> ⚠ 인터페이스를 구현한다는 것은 자신의 인스턴스로 무엇을 할 수 있는지를 클라이언트에 얘기해주는 것이다

안티패턴

상수 인터페이스 안티 패턴은 인터페이스를 잘못 사용한 예다

- 클래스 내부에서 사용하는 상수는 외부 인터페이스가 아니라 내부 구현에 해당한다. 따라서 상수 인터페이스를 구현하는 것은 내부 구현을 클래스의 API로 노출하는 행위이다
    - java.io.ObjectStreamConstants 등 자바 라이브러리에도 상수 인터페이스가 몇 개 있으나, 인터페이스를 잘못 활용한 예이니 따라해서는 안된다

```
public interface PhysicalConstants {
        //아보가드로 수 (1/몰)
        static final double AVOGADROS_NUMBER = 6.022_140_8

        //볼츠만 상수 (J/K)
        static final double BOLTZMANN_CONSTANT = 1.380_648

        //전자 질량 (kg)
        static final double ELECTRON_MASS = 9.109_383_56e-
}
```

상수를 공개하는 방법

1. 특정 클래스나 인터페이스와 강하게 연관된 상수라면 그 클래스나 인터페이스 자체에 추가한다

- ex) Integer와 Double의 MAX_VALUE/MIN_VALUE

2. 열거 타입으로 나타내기 적합한 상수라면 열거타입으로 만들어 공개한다 (아이템 34)

3. 인스턴스화할 수 없는 유틸리티 클래스(아이템 4)에 담아 공개하자

```
public class PhysicalConstants {
        private PhysicalConstants() { } //인스턴스화 방지

        //아보가드로 수 (1/몰)
        static final double AVOGADROS_NUMBER = 6.022_140_857e

        //볼츠만 상수 (J/K)
        static final double BOLTZMANN_CONSTANT = 1.380_648_52

        //전자 질량 (kg)
        static final double ELECTRON_MASS = 9.109_383_56e-31;
}
```

⚠ 유틸리티 클래스에 정의된 상수를 클라이언트에서 사용하려면 클래스 이름까지 함께 명시해야 한다. 유틸리티 클래스를 빈번히 사용한다면 정적 임포트(static import)하여 클래스 이름을 생략할 수 있다

▼ [아이템 23] 태그 달린 클래스보다는 클래스 계층 구조를 활용하라

태그 달린 클래스

1. 태그 달린 클래스는 장황하고, 오류내기 쉽고, 비효율적이다

- 쓸데없는 코드가 많다
- 여러 구현이 한 클래스에 혼합돼있어 가독성이 나쁘다
- 인스턴스의 타입만으로는 현재 나타내는 의미를 알 길이 전혀 없다

☝🏻 두 가지 이상의 의미를 표현할 수 있고, 현재 표현하는 의미를 태그값으로 알려주는 클래스의 예시

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

        //태그 필드 - 현재 모양을 나타낸다
        final Shape shape;

        //모양이 사각형(RECTANGLE)일 때만 쓰이는 필드
        double length;
        double width;

        //모양이 원(CIRCLE)일 때만 쓰이는 필드
        double radius;

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

        //사각형용 생성자
        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:
                                return Math.PI * (radius
                        default:
                                throw new AssertionError
                }
```

```
            }
        }
```

2. 자바와 같은 객체 지향 언어는 타입 하나로 다양한 의미의 객체를 표현하는 훨씬 나은 수단을 제공한다

- 태그 달린 클래스는 클래스 계층 구조를 어설프게 흉내낸 아류일 뿐이다
- 클래스 계층 구조를 활용하는 서브타이핑(subtyping)이 그 방법이다

클래스 계층 구조

태그 달린 클래스를 클래스 계층 구조로 바꾸는 방법

1. 계층 구조의 루트(root)가 될 추상 클래스를 정의한 후, 태그 값에 따라 동작이 달라지는 메소드들을 루트 클래스의 추상 메소드로 선언한다.
2. 태그 값에 상관 없이 동작이 일정한 메소드들을 루트 클래스에 일반 메소드로 추가한다
    a. 모든 하위 클래스에서 공통으로 사용하는 데이터 필드들도 전부 루트 클래스로 올린다
3. 루트 클래스를 확장한 구체 클래스를 의미별로 하나씩 정의한다

```
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, width){
                this.length = length;
                this.width = width;
        }

        @Override
        double area(){
                return length * width;
        }
}
```

⚠ 태그 달린 클래스를 써야 하는 상황은 거의 없다. 만약 태그 필드가 등장한다면 태그를 없애고 계층 구조로 대체하는 방법을 생각해보자.

▼ [아이템 24] 멤버 클래스는 되도록 static으로 만들라

중첩 클래스(nested class)

- 다른 클래스 안에 정의된 클래스
- 자신을 감싼 바깥 클래스에서만 쓰여야 한다
  - 그 외의 쓰임새가 있다면 탑레벨 클래스로 만들어야 한다

1. 정적 멤버 클래스

- 다른 클래스 안에 선언되고, 바깥 클래스의 private 멤버에도 접근할 수 있다는 점만 제외하면 일반 class와 똑같다
- 흔히 바깥 클래스와 함께 쓰일 때만 유용한 **public 도우미 클래스**로 쓰인다

2. (비정적) 멤버 클래스

- 정적 멤버 클래스와의 구문상의 차이는 static의 유무 뿐이지만, 의미상 차이는 꽤 크다
- 비정적 멤버 클래스의 인스턴스는 바깥 클래스의 인스턴스와 암묵적으로 연결된다

    - 비정적 멤버 클래스의 인스턴스 메소드에서 **정규화된 this**를 사용해 바깥 인스턴스의 메소드를 호출하거나 참조를 가져올 수 있다
    - 따라서 개념상 중첩 클래스의 인스턴스가 바깥 인스턴스와 독립적으로 존재할 수 있다면, 정적 멤버 클래스로 만들어야 한다

> 📕 **정규화된 this**란, `클래스명.this` 형태로 바깥 클래스의 이름을 명시하는 용법이다

- 보통 어댑터를 정의할 때 자주 쓰인다
    - 어떤 클래스의 인스턴스를 감싸 마치 다른 클래스의 인스턴스처럼 보이게 하는 뷰로 사용한다
    - ex)Set, List같은 컬렉션 인터페이스 구현들이 자신의 반복자를 구현할 때 주로 사용한다

> ⚠️ 멤버 클래스에서 바깥 인스터스에 접근할 일이 없다면 무조건 static을 붙여 정적 멤버 클래스로 만들자 (이는 메모리 누수를 예방할 수 있다, 아이템 7)

3. 익명 클래스

- 쓰이는 시점에 선언과 동시에 인스턴스가 만들어진다
- 코드의 어디서든 만들 수 있다
- 오직 비정적인 문맥에서 사용될 때만 바깥 클래스의 인스턴스를 참조할 수 있다
- 정적인 문맥에서라도 상수 변수 이외의 정적 멤버는 가질 수 없다
- 응용하는데 제약이 많다
    - 선언한 지점에서만 인스턴스 생성 가능(instanceof 검사 및 클래스의 이름이 필요한 작업 불가)
    - 다중 인터페이스 구현 불가
    - 인터페이스 구현과 상속 동시에 불가
    - 익명클래스를 사용하는 클라이언트는 익명클래스가 상위타입에서 상속한 멤버 외에는 호출 불가
    - 코드가 길면 가독성이 떨어짐
- 이제는 익명클래스 대신 주로 람다 활용(아이템 42)
    - 익명 클래스는 정적 팩토리 메소드를 구현할 때 주로 활용

4. 지역 클래스

- 네 가지 중첩 클래스 중 가장 드물게 사용된다
- 지역변수를 선언할 수 있는 곳이면 실질적으로 어디서든 선언 가능하다
    - 유효 범위도 지역변수와 같다
- 멤버 클래스처럼 이름이 있고, 반복사용이 가능하다
- 익명 클래스처럼 비정적 문맥에서 사용될 때만 바깥 인스턴스를 참조할 수 있고, 정적 멤버는 가질 수 없다
- 가독성을 위해 짧게 작성해야 한다

정리 (사용할 시점)

1. 멤버 클래스

- 메소드 밖에서도 사용해야 하거나 메소드 안에 정의하기 너무 긴 경우
- 멤버 클래스의 인스턴스 각각이 바깥 인스턴스를 참조한다면 비정적, 아닌 경우 정적

2. 익명/지역 클래스

- 한 메소드 안에만 쓰이면서 그 인스턴스를 생성하는 곳이 한 곳이고, 해당 타입으로 쓰기에 적합한 클래스나 인터페이스가 이미 있는 경우 익명, 아니면 지역

▼ [아이템 25] 탑레벨 클래스는 한 파일에 하나만 담으라

한 소스 파일에 여러 탑레벨 클래스를 선언한 경우

- 정상적으로 컴파일 된다. 그러나 심각한 위험이 따른다
    - 한 클래스를 여러 가지로 정의할 수 있으며, 그 중 어느 것을 사용할 지는 소스 파일 컴파일 순서에 따라 달라진다

> ⚠ **탑레벨 클래스들을 서로 다른 소스 파일로 분리하라!**