Ch4. 클래스와 인터페이스

태그	2주차

Item15 클래스와 멤버의 접근 권한을 최소화하라

잘 설계된 컴포넌트는, **모든 내부 구현화 데이터 정보를 외부로부터 완벽히 숨겨, 구현과 API를 깔끔하게 분리**한다. 이렇게 오직 API를 통해서만 다른 컴포넌트와 소통하며 서로의 내부 동작 방식에 개의치 않는 이 개념을, 바로 **정보 은닉(캡슐화)** 라고 한다.

접근 제어 메커니즘

- 접근 제한자 종류
  - **private**: 외부 접근 불가. 가장 제한적인 접근 수준
  - **package-private (default)**: 같은 패키지 내에서만 접근 가능. 외부 패키지에서는 접근불가. 패키지 수준에서 캡슐화를 유지가능.
  - **protected**: 같은 패키지 내의 클래스와 **하위 클래스**에서 접근가능.
  - **public**: 모든 클래스에서 접근가능.

기본 원칙은, **모든 클래스와 멤버의 접근성을 가능한 한 좁혀야 한다**는 것이다. (가장 낮은 접근 수준을 부여해야함)

톱 레벨 클래스와 인터페이스 접근자

톱레벨 클래스와 인터페이스에 부여할 수 있는 접근 수준은 `package-private`과 `public` 두 가지이다.

만약 한 클래스에서만 사용하는 `package-private` 톱 레벨 클래스나 인터페이스는, 사용하는 클래스 안에 `private static`으로 중첩시키면 바깥 클래스 하나에서만 접근 할 수 있게 된다.

공개 API 제외 모든 멤버는 `private`으로 만든 후, 오직 같은 패키지의 다른 클래스가 접근해야 하는 멤버에 한하여 `package-private`으로 풀어주어야 한다.

`package-private` 와 `private` 멤버는 클래스의 구현에 해당하므로 공개 API에 영향을 주지 않지만, `Serializable` 을 구현한 클래스에서는 필드들로 의도치 않게 공개 API가 될 수 있으니 주의의해야한다. `Serializable` 역직렬화 과정에서 `private` 필드가 노출될 수 있다.

멤버 접근성을 좁히지 못하게 하는 제약

단 상위 클래스의 메서드에서 재정의할때는, 리스코프 치환 원칙으로 인해 그 접근 수준을 **상위 클래스에서보다 좁게 설정할 수 없다.**

- 리스코프 치환 원칙

상위 클래스의 인스턴스는 하위 클래스의 인스턴스로 대체해 사용할 수 있어야 한다. → 컴파일 오류 발생

주의사항

주의할 점으로, `public` 클래스의 인스턴스 필드는 되도록 `public` 이 아니어야 한다.

필드가 가변 객체를 참조하거나, `final` 이 아닌 인스턴스 필드를 공개하면 **불변식을 보장할 수 없게** 되기 때문이다. 또한 필드가 수정될 때 다른 작업 수행이 불가능하므로 `public` 가변 필드를 갖는 클래스는 **스레드 안전**하지 않다.

단, **상수**라면 `public static final` 필드로 공개해도 괜찮으며, 해당 필드는 반드시 기본 타입 값이나 불변 객체를 참조해야 한다. 만약 배열이라면, 아래와 같이 변경하자.

```
public static final Thing[] VALEUS = {...};
```

1. `private` 으로 변경한 후 `public` 불변 리스트를 추가

```
private static final Thing[] VALEUS = {...};
public static final List<Thing> VALUES =
    Collections.unmodifiableList(Arrays.asList(PRIVATE_VALU
ES));
```

2. 배열을 `private` 로 만들고 그 복사본을 반환하는 `public` 메서드를 추가

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

모듈 시스템에서의 접근 제한자

자바 9부터는 **모듈 시스템**이라는 개념이 도입되면서, 두 가지 암묵적 접근 수준이 추가되었다.

> 모듈(Module)

> 패키지들의 묶음을 말하며 자신에 속하는 패키지 중 공개(export)할 것들을 `module-info.java` 파일에 선언한다. 모듈 시스템을 활용하면 클래스를 외부에 공개하지 않으면서도 같은 모듈을 이루는 패키지 사이에서는 자유롭게 공유가 가능하다는 장점이 있다.

모듈에 적용되는 암묵적 접근 수준은 `public`, `protected` 수준과 같으나 그 효과가 **모듈 내부로 한정**된다는 점만 다르다. 이 접근 수준을 활용한 예로 `JDK`가 있으며, 자바 라이브러리에서 공개하지 않은 패키지들은 해당 모듈 밖에서 절대 접근 할 수 없다.

⭐핵심정리

> 💡 프로그램 요소의 접근성은 가능한 한 최소한으로 하라. 꼭 필요한 것만 골라 최소한의 publicAPI를 설계하자. 그 외에는 클래스, 인터페이스, 멤버가 의도치 않게 API로 공개되는 일이 없도록 해야한다. public 클래스는 상수용 public static final 필드 외에는 어떠한 public 필드도 가져서는 안된다. public static final가 참조하는 객체가 불변인지 확인하라.

Item16 public 클래스에서는 public 필드가 아닌 접근자 메서드를 사용하라

접근자와 변경자 (`getter`, `setter`)를 활용해 데이터를 캡슐화 하자!!)

⭐핵심정리

> 💡 public 클래스는 절대 가변 필드를 직접 노출해서는 안 된다. 불변 필드라면 노출해도 덜 위험하지만 완전히 안심할 수는 없다. 하지만 package-private 클래스나 private 중첩 클래스에서는 종종 필드를 노출하는 편이 나을 수도 있다.

Item17 변경 가능성을 최소화하라

불변 클래스란, **인스턴스의 내부 값을 수정할 수 없는 클래스**이며 **불변 인스턴스에 간직된 정보는 고정되어 객체가 파괴되는 순간까지 절대 달라지지 않는다.**대표적인 immutable 클래스에는 `String`, `Boolean`, `Integer`, `Float`, `Long` 등이 존재한다.

불변 클래스와 final 키워드 (불변과 재할당에 대해)

- 불변

> 💡 객체 데이터의 수정이 불가능 한 것이 아니라 **힙 영역에 저장된 값의 수정이 불가능**한 것을 의미. **즉, 재할당은 가능 (재할당을 막는 키워드는 final)**

```
Integer num = Integer.valueOf(10); // num이 10 Integer객체를 가
num = 20; // num이 20을 가리킴 (Integer(10) 객체가 20으로 바뀐게 아
num = Integer.valueOf(30); // 명시적으로 새로운 Integer 객체 참조
```

`num` 변수가 새로운 값을 가리키게 되었지만, **기존의** `Integer` **객체**는 변경되지 않고 그대로 유지된다. 이 과정에서 기존 객체는 변경되지 않고, `num` 변수가 새 객체를 참조하게 된다. **객체 자체의 불변성**은 유지되며, `num`이라는 **참조 변수**가 다른 객체를 가리키게 된 것!

이러한 동작은 불변 객체의 주요 특징 중 하나로, 참조 변수가 가리키는 객체가 변경 되더라도 기존 객체의 상태는 변하지 않는다는 점에서 불변성이 보장된다. (즉 변수에 재할당 되는 것!)

- **final 키워드**

final은 불변을 만드는 키워드가 아닌, 재할당을 금지하는 키워드이다.

final로 선언된 list는 `add`, `remove`, `set`과 같은 함수를 통해 list의 안 element를 변경 가능하다. 단지 final로 선언된 list는 항상 같은 list 객체를 참조하고 있을 뿐이다.

```
final List<Integer> results = getResult();
results = getResult();  // 재할당 시 에러 발생
results.add(1); // 가능하며 실제로 result안에 1 삽입됨.
```

클래스를 불변으로 만드는 다섯 가지 규칙

1. 객체의 상태를 변경하는 메서드(변경자)를 제공하지 않는다.

- `getter`와 `setter`를 제공하지 않는다.

2. 클래스를 확장할 수 없도록 한다.

- 하위 클래스에게 부주의하게 혹은 나쁜 의도로 객체의 상태를 변하게 만드는 사태를 막아준다. 상속을 막는 대표적인 방법은 클래스를 `final`로 선언하는 것이지만, 다른 방법도 존재한다.

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 final`로만 선언해도 불변 객체가 되지만, 다음 릴리스에서 내부 표현을 바꾸지 못하므로 권하지 않는다.(*아이템 15,16*)

5. 자신 외에는 내부의 가변 컴포넌트에 접근할 수 없도록 한다.

- 방어적 복사에 관한 이야기이다.
- 클래스에 가변 객체를 참조하는 필드가 하나라도 있다면, 클라이언트에서 그 객체의 직접적인 참조를 얻을 수 없도록 해야 한다. 생성자, 접근자, `readObject()` 메서드(*아이템 88*) 모두에서 방어적 복사를 수행하라.

불변 클래스 예시

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

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

    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);
    }

    // 코드 17-2 정적 팩터리(private 생성자와 함께 사용해야 한다.)
    public static Complex valueOf(double re, double im) {
        return new Complex(re, 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 boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof Complex))
            return false;
        Complex c = (Complex) o;

        // == 대신 compare를 사용하는 이유는 63쪽을 확인하라.
        return Double.compare(c.re, re) == 0
                && Double.compare(c.im, im) == 0;
    }
```

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

    @Override public String toString() {
        return "(" + re + " + " + im + "i)";
    }
}
```

해당 클래스에서는, 사칙연산 메서드들이 인스턴스 자신은 수정하지 않고 새로운 Complex 인스턴스를 만들어 반환하고 있다.

이처럼 **피연산자에 함수를 적용해 그 결과를 반환하지만, 피연산자 자체는 그대로**인 프로그래밍 패턴을 **함수형 프로그래밍**이라 한다. 함수형 프로그래밍을 사용한다면, 코드에서 불변이 되는 영역을 증가시킬 수 있다.

불변 클래스 장단점

- **장점**
  - 불변 객체는 단순하다. 불변 객체는 생성된 시점의 상태를 파괴될 때까지 그대로 간직한다.
  - 불변 객체는 근본적으로 스레드 안전하여 따로 동기화할 필요 없다. 불변 객체는 안심하고 공유할 수 있다.
  - 불변 클래스는 자주 사용되는 인스턴스를 캐싱하여 같은 인스턴스를 중복 생성하지 않게 해주는 정적 팩터**리**를 제공할 수 있다.
  - 불변 객체는 자유롭게 공유할 수 있음은 물론, 불변 객체끼리는 내부 데이터를 공유할 수 있다.
  - 불변 객체는 그 자체로 **실패 원자성을 제공**한다. 상태가 절대 변하지 않으니 불일치 상태에 빠질 가능성이 없다.

단점

- 값이 다르면 반드시 독립된 객체로 만들어야 한다. → 가변 동반 클래스를 제공하여 문제점을 해결 가능하다.(`BigInteger` & `BigSet`, `String` & `StringBuilder`)

⭐핵심정리

💡 **클래스는 꼭 필요한 경우가 아니라면 불변이여야 한다.**
불변으로 만들 수 없는 클래스라도, 변경할 수 있는 부분을 최소한으로 줄여야 한다.
생성자는 불변식 설정이 모두 완료된, 초기화가 완벽히 끝난 상태의 객체를 생성해야 한다.

Item18 상속보다는 컴포지션을 사용하라

여기서 말하는 상속은 클래스가 다른 패키지의 클래스를 확장하는 구현 상속을 말한다. (인터페이스 상속과는 무관)

같은 패키지의 부모 클래스를 확장하는 상속은 코드를 재사용하는 강력한 수단이지만, **다른 패키지의 클래스를 상속하는 일은 위험**하다. 메서드 호출과 달리 상속은 **캡슐화를 깨뜨릴 수** 있기 때문이다.

상속이 잘못 사용된 예시

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

    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;
    }
}

public static void main(String[] args) {
        InstrumentedHashSet<String> s = new InstrumentedHashS
        s.addAll(List.of("틱", "탁탁", "펑"));
        System.out.println(s.getAddCount()); // 6이 반환?
}
```

위 코드에서 `addAll` 메서드를 보면 `addAll` 이전에 `addCount` 를 원소 수만큼 더해주는 연산을 해주고 부모의 `addAll`을 호출한다.

```
public boolean addAll( @NotNull Collection<? extends E> c) {
    boolean modified = false;
    for (E e : c)
        if (add(e))
            modified = true;
    return modified;
}
```

하지만 부모의 `addAll`안에는 `add` 메서드를 호출하는 코드가 있으며, 이 때 호출되는 `add`는 재정의한 add이다. 재정의한 `add`에서 호출될 때마다 `addCount`를 더해주는 코드가 있기 때문에 `addCount`는 기대했던 것의 2배의 값이 나온다.

이처럼 **내부에서 같은 클래스의 다른 메서드를 사용**하는 자기사용(self-use)여부는 해당 클래스의 내부 구현 방식에만 해당하기 때문에, 다음 릴리즈에서도 변경이 일어나면서 하위 클래스가 깨져버릴 가능성이 높다.

컴포지션을 사용하자

컴포지션이란 **기존 클래스를 확장하지 않고**, 새롭게 만든 클래스 내부 `private` **필드로 기존 클래스의 인스턴스를 참조**하는 방식이다. 기존 클래스가 새로운 클래스의 구성요소로 쓰인다는 뜻에서, 컴포지션이라고 이름이 붙여졌다.

```
public class ForwardingSet<E> implements Set<E> {
    private final Set<E> s;    // 내부 필드로 상위 클래스의 참조를
```

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

    public void clear()               { s.clear();
    public boolean contains(Object o) { return s.contains(o);
    public boolean isEmpty()          { return s.isEmpty();
    public int size()                 { return s.size();
    public Iterator<E> iterator()     { return s.iterator();
    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 removeAll(Collection<?> c)
                                      { return s.removeAll(c);
    public boolean retainAll(Collection<?> c)
                                      { return s.retainAll(c);
    public Object[] toArray()           { return s.toArray();
    public <T> T[] toArray(T[] a)       { return s.toArray(a);
    @Override public boolean equals(Object o)
                                        { return s.equals(o);
    @Override public int hashCode()     { return s.hashCode();
    @Override public String toString() { return s.toString();
}

//HashSet이 아닌 컴포지션인 ForwardingSet 상속
public class InstrumentedSet<E> extends ForwardingSet<E> {
    //...
}
```

위와 같이 설계하면 직접적으로 상속하지 않았으니 `addAll` 가 호출되었을 때 더 이상 `InstrumentedSet` 에서 재정의된 `add` 메서드를 부르지 않는다.

쉽게 말하면 부가 기능이 존재하는 래퍼 클래스(`InstrumentedSet`)와 실제 상위 클래스(`Set`) 사이에 중간 계층(`ForwardingSet`)을 하나 두어, 래퍼 클래스에서 실제 상위 클래스의 메서드가 호출되어도 상속 관계가 아닌 단순히 인자로 받고 호출된 것이므로 재정의에 전혀 영향을 받지 않는다.

컴포지션의 장단점

- 장점
  1. **기존 클래스의 내부 구현 방식과 독립**적이게 되서, 메서드 재정의에 따른 부가 영향을 고려하지 않아도 되어 안전하다.
  2. 재사용할 수 있는 전달 클래스를 하나만 만들어두어도, 원하는 기능을 덧씌우는 클래스들을 손쉽게 구현할 수 있다. 역시 상속을 사용했다면, 매번 부모 클래스를 상속받은 새로운 구체 클래스를 생성했어야 할 것이다.
- 단점

  콜백(Callback) 프레임워크와는 어울리지 않는다. (자기 자신의 참조를 다른 객체에 넘겨서, 다음 호출(콜백) 때 사용하도록 하는 방식)

⭐핵심정리

> 💡 상속은 강력하지만 캡슐화를 해친다는 문제가 있다. 상속은 상위 클래스와 하위 클래스가 순수한 is-a 관계일 때만 써야 하는데, 여전히 하위 클래스의 패키지가 상위 클래스와 다르고, 상위 클래스가 확장을 고려해 설계되지 않았다면 문제가 될 수 있다. 상속의 취약점을 피하려면 상속 대신 컴포지션과 전달을 사용하자.

Item19 상속을 고려해 설계하고 문서화하라. 그렇지 않았다면 상속을 금지하라

이론적인 내용이다.

- 상속용 클래스는 재정의할 수 있는 메서드들을 내부적으로 어떻게 이용하는지 문서로 남거야 한다.
- **클래스의 내부 동작 과정 중간에 끼어들 수 있는 훅(hook)을 잘 선별하여 protected 메서드 형태로 공개해야 한다.**
- **상속용 클래스의 생성자는 직접적으로든 간접적으로든 재정의 가능 메서드를 호출해서는 안된다.**
- **clone과 readObject 메서드 모두 직접적으로든 간접적으로든 재정의 가능 메서드를 호출해서는 안된다.**

- **Serializable을 구현한 상속용 클래스가 readResolve나 writeReplace 메서드를 갖는다면 private가 아닌 protected로 선언해야 한다.**

⭐핵심정리

> 💡 상속용 클래스를 설계하기란 결코 만만치 않다. 클래스 내부에서 스스로를 어떻게 사용하는지(자기사용 패턴) 모두 문서로 남겨야 하며, 다른 이가 효율 좋은 하위 클래스를 만들 수 있도록 일부 메서드를 protected으로 제공해야 할 수도 있다. 그러니 클래스를 확장해야 할 명확한 이유가 없으면 상속을 금지하는 것이 좋다. 상속을 금지하려면 클래스를 final로 선언하거나 생성자 모두를 외부에서 접근 할 수 없도록 만들면 된다.

Item 20 : 추상 클래스보다는 인터페이스를 우선하라

자바는 **인터페이스**와 **추상 클래스** 두가지를 통해 **다중 구현 메커니즘** 을 제공한다.

인터페이스를 사용해야하는 이유

1. **기존 클래스에 손쉽게 새로운 인터페이스를 구현해 넣을 수 있다.**

인터페이스는 요구하는 메서드를 추가하고, 클래스 선언에 `implements` 구문만 추가하면 끝이다.

2. **인터페이스는 믹스인(mixin) 정의에 안성맞춤이다.**

- Mixin이란?
  - 대상 타입의 주된 기능에 선택적 기능을 `혼합(mixed in)` 한다는 의미
  - 객체지향언어에서 다른 클래스에서 사용할 목적으로 만들어진 클래스
  - ex) Comparable

3. **래퍼 클래스 관용구와 함께 사용하면, 인터페이스는 기능을 향상시키는 안전하고 강력한 수단이 된다.**

한편, **인터페이스와 추상 골격 클래스(skeletal implementation)를 함께 제공**하는 식으로 인터페이스와 추상 클래스의 장점을 모두 취하는 방법도 존재한다.

1. 인터페이스로는 타입 정의와 필요한 디폴트 메서드를 제공한다.

2. 골격 구현 클래스로 나머지 메서드들까지 구현한다.

이렇게 되면 단순히 골격 구현을 확장하는 것만으로 인터페이스를 구현하는데 필요한 일이 완료된다. 다른 말로 **템플릿 메서드 패턴(Template Method Pattern)**이라고 하는데, 구조는 다음과 같다.

```
interface InterfaceA {
    void methodA();
}

abstract class AbstractClassA implements InterfaceA {
    abstract void methodB();
}

class ConcreteClassA extends AbstractClassA {
    public void methodA() {
        System.out.println("method A");
    }

    public void methodB() {
        System.out.println("method B");
    }
}
```

변하는 부분만 추상 메서드로 정의하여 서브 클래스에서 자유롭게 상속받아 구현할 수 있.

⭐핵심 정리

> 💡 일반적으로 다중 구현용 타입으로는 인터페이스가 가장 적합하다. 복잡한 인터페이스라면 구현하는 수고를 덜어주는 골격 구현을 함께 제공하는 방법도 고려해보자. 골격 구현은 '가능한 한' 인터페이스의 디폴트 메서드로 제공하여 그 인터페이스를 구현한 모든 곳에서 활용하도록 하는 것이 좋다. '가능한 한'이라고 한 이유는, 인터페이스에 걸려 있는 구현상의 제약 때문에 골격 구현을 추상 클래스로 제공하는 경우가 더 흔하기 때문이다.

Item21 인터페이스는 구현하는 쪽을 생각해 설계하라

자바 8 이전에는 기존 구현체를 깨뜨리지 않고는 인터페이스에 메서드를 추가할 방법이 없었지만, 자바 8부터는 **디폴트 메소드** 를 통해 **인터페이스에 메서드를 추가**하는 것이 가능해졌다.

default 메서드의 문제점

기존에 존재하던 구현체들과 연동되지 않을 수 있다.

인터페이스를 구현한 후 디폴트 메서드를 재정의하지 않은 '모든' 클래스에서도 디폴트 구현이 쓰이게 되기 때문이다.

⭐핵심정리

> 💡 **기존 인터페이스**에 **디폴트 메서드로 새 메서드를 추가하는 일은 꼭 필요한 경우가 아니면 피해야 한다.** 반면, 새로운 인터페이스를 만드는 경우라면 표준적인 메서드 구현을 제공하는데 유용한 수단이며 인터페이스를 더 쉽게 구현해 활용할 수 있게끔 해준다.

Item22 인터페이스는 타입을 정의하는 용도로만 사용하라

인터페이스는 **자신을 구현할 클래스의 인스턴스를 참조할 수 있는 <타입 역할>**을 한다. 이를 만족하지 않고 잘못 사용하는 대표적인 예시가 상수인터페이스 이다.

- 상수 인터페이스: 메서드 없이, 상수를 뜻하는 `static final` 필드로만 이루어진 인터페이스

```
public interface PhysicalConstants {
    // 아보가드로 수 (1/몰)
    static final double AVOGADROS_NUMBER   = 6.022_140_857e23

    // 볼츠만 상수 (J/K)
```

```
    static final double BOLTZMANN_CONSTANT = 1.380_648_52e-23

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

문제점1: 내부 구현을 오히려 인터페이스로 선언해 클래스의 API로 노출하는 문제 발생

문제점2: `final` 이 아닌 클래스가 상수 인터페이스를 구현한다면 모든 하위 클래스의 이름 공간이 그 인터페이스가 정의한 상수로 덮여진다.

→ 상수를 공개할 목적이라면, 특정 클래스나 인터페이스와 강하게 연관된 상수일 경우 **클래스나 인터페이스 자체에 추가**해야 한다. ex) `Integer.MIN_VALUE` , `Integer.MAX_VALUE`

인터페이스 대신, 열거 타입 또는 싱글톤 클래스에 담아서 상수를 나타내자!

⭐핵심정리

> 💡 인터페이스는 타입을 정의하는 용도로만 사용해야 하지, 상수 공개용 수단으로 사용하지 말자.

Item23 태그 달린 클래스보다는 계층 구조를 활용하라

태그 달린 클래스

태그 달린 클래스란, **하나의 클래스가 두 가지 이상의 의미를 표현** 가능할 때, 그중 현재 표현하는 의미를 태그 값으로 알려주는 클래스이다.

- 예시

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

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

```
    // 다음 필드들은 모양이 사각형(RECTANGLE)일 때만 쓰인다.
    double length;
    double width;

    // 다음 필드는 모양이 원(CIRCLE)일 때만 쓰인다.
    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 * radius);
            default:
                throw new AssertionError(shape);
        }
    }
}
```

- 문제점

1. 쓸데없는 코드로 가독성이 떨어진다.
2. 의미없는 코드일지라도 항상 존재하기 때문에 메모리를 많이 사용
3. 필드들을 불변을 위해 `final` 로 선언하려면 해당 의미에 쓰이지 않는 필드들까지 생성자에서 초기화해야한다.

4. 또 다른 의미를 추가하려면 `switch` 문 코드를 수정해야 한다.
5. 인스턴스의 타입만으로는 현재 나타내는 의미를 알 길이 전혀 없다.

딱봐도 안좋고 비효율적이니 사용하지말자!

클래스 계층구조

위 문제를 클래스 계층구조를 활용하는 서브타이핑 (subtyping)을 통해 해결할 수 있다. 클래스 계층 구조는, **추상 클래스와 추상 메서드**를 통해 **타입 하나로 다양한 의미의 객체를 표현**할 수 있게 해준다.

1. 계층 구조의 루트가 될 추상 클래스 정의한다.

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

- 태그 값에 따라 **동작이 달라지는 메서드**들을 루트 클래스의 **추상 메서드**로 선언
- 태그 값에 상관없이 **동작이 일정한 메서드와 공통 메서드**들을 루트 클래스에 **일반 메서드**로 선언

1. 루트 클래스를 확장한 구체 클래스를 의미별로 하나씩 지정한다.

```
class Circle extends Figure {
    final double radius;

    Circle(double radius) { this.radius = radius; }

    @Override double area() { return Math.PI * (radius * ra
dius); }
}
```

```
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; }
}
```

- 루트 클래스가 정의한 추상 메서드를 각자 의미에 맞게 구현

클래스 계층 구조 장점

1. 각 의미를 독립된 클래스에 담아 관련 없던 데이터 필드들을 제거했다.
2. 남아 있는 필드들은 모두 `final` 로 선언해 불변을 보장할 수 있다.
3. 각 클래스의 생성자가 모든 필드를 남김없이 초기화하고 추상 메서드를 모두 구현했는지 컴파일러 단에서 확인할 수 있다.
4. 루트 클래스의 코드를 건드리지 않고 독립적으로 계층 구조를 확장할 수 있다.

⭐핵심 정리

> 💡 가독성이 떨어지고 비효율적인 태그 클래스 대신, 계층구조를 활용하자!

Item24 멤버 클래스는 되도록 static 으로 만들어라

중첩 클래스의 정의와 종류

중첩 클래스(nested class)란, 다른 클래스 안에 정의되어 자신을 감싼 바깥 클래스에서만 쓰여야 하는 클래스를 의미한다. 종류는 다음과 같다.

1. 정적 멤버 클래스
    - **정적 멤버 클래스는 다른 클래스 안에 선언되고, 바깥 클래스의 `private` 멤버에도 접근할 수 있다는 점만 제외하면 일반 클래스와 동일하다. 또한, 다른 정적 멤버와 똑같은 접근 규칙을 받는다.**
    - 메서드 밖에서도 사용해야 하거나 메서드 안에 정의하기엔 너무 길고, 바깥 인스턴스를 참조하지 않을 때 사용한다.
    - 멤버 클래스에서 바깥 인스턴스에 접근할 일이 없다면 무조건 static을 붙여서 정적 멤버 클래스로 만드는게 좋다.

2. 비정적 멤버 클래스

```
public class NestedNonStaticExample {

    private final String name;

    public NestedNonStaticExample(String name) {
        this.name = name;
    }

    public String getName() {
        // 비정적 멤버 클래스와 바깥 클래스의 관계가 확립되는 부분
        NonStaticClass nonStaticClass = new NonStaticClass
        return nonStaticClass.getNameWithOuter();
    }

    private class NonStaticClass {
        private final String nonStaticName;

        public NonStaticClass(String nonStaticName) {
            this.nonStaticName = nonStaticName;
        }

        public String getNameWithOuter() {
            // 정규화된 this 를 이용해서 바깥 클래스의 인스턴스 메서
            return nonStaticName + NestedNonStaticExample.
        }
    }
}
```

비정적 멤버 클래스와 바깥 인스턴스 사이의 관계는 **멤버 클래스가 인스턴스화될 때 확립되며, 더이상 변경 불가능**하다. `NonStaticClass` 에서 정규화된 `this` 를 사용하여 바깥 클래스의 참조를 가져온 것을 볼 수 있다.

비정적 멤버 클래스는 주로 **멤버 클래스에서 바깥 인스턴스에 접근해야 하는 어댑터를 정의**할 때 자주 쓰인다.

- 익명 클래스

    - 중첩 클래스가 한 메서드 안에서만 쓰이면서 그 인스턴스를 생성하는 지점이 단 한 곳이고 해당 타입으로 쓰기에 적합한 클래스나 인터페이스가 이미 있을때, 멤버와 달리, 쓰이는 시점에 선언과 동시에 인스턴스가 만들어진다.
- 지역 클래스
    - 지역 클래스는 **지역변수를 선언할 수 있는 곳이면 어디서든 선언 가능**하며, 유효 범위도 지역번수와 같다.

⭐핵심정리

> 💡 중첩 클래스에는 네 가지가 있으며, 각각의 쓰임이 다르다. 메서드 밖에서도 사용해야 하거나 메서드 안에 정의하기엔 너무 길다면 멤버 클래스로 만든다. 멤버 클래스의 인스턴스 각각이 바깥 인스턴스를 참조한다면 비정적으로, 그렇지 않다면 정적으로 만들자. 중첩 클래스가 한 메서드 안에서만 쓰이면서 그 인스턴스를 생성하는 지점이 단 한 곳이고 해당 타입으로 쓰기에 적합한 인터페이스나 클래스가 이미 있다면 익명 클래스로 만들고, 그렇지 않으면 지역 클래스로 만들자.

Item25 톱 레벨 클래스는 한 파일에 하나만 담으라

톱 레벨 클래스는 **중첩 클래스가 아닌 클래스**이다.

- A.java

```
class A {
    String NAME = "a";
}

class B{
    String NAME = "b";
}
```

- B.java

```
class B {
    String NAME = "aa";
}

class A{
    String NAME = "bb";
}
```

이 처럼 A와 B파일에 같은 클래스를 순서만 바꿔서 선언한 경우,

javac 명령을 통해 컴파일할 때 컴파일 순서에 따라 동작이 달라질 수 있다고 한다.

(하지만 IDE가 이를 잡아주긴함)

- 해결법

1. 한 개의 파일에 한 개의 톱레벨 클래스만 둔다.
2. 정적 멤버 클래스 방식을 사용한다.

⭐핵심정리

> 💡 소스 파일 하나에는 반드시 톱레벨 클래스(혹은 톱레벨 인터페이스)를 하나만 담자. 이 규칙만 따른다면 컴파일러가 한 클래스에 대한 정의를 여러 개 만들어내는 일은 사라진다. 소스 파일을 어떤 순서로 컴파일하든 바이너리 파일이나 프로그램의 동작이 달라지는 일은 결코 일어나지 않을 것이다.