이펙티브 자바 클래스와 인터페이스

item15. 클래스와 멤버의 접근 권한을 최소화하라

내부 데이터와 내부 구현 정보를 외부로 부터 얼마나 잘 숨겼는지가 중요하다

- 정보은닉, 캡슐화라고 함
- 시스템을 구성하는 컴포넌트를 독립시켜 개발, 테스트, 최적화, 적용, 분석, 수정이 개별적으로 가능해야 함

장점

- 컴포넌트간 의존성 x → 독립적인 병렬 개발이 가능 → 개발 속도 증가
- 컴포넌트별 디버깅이 가능 → 관리 비용이 절약
- 컴포넌트별 최적화가 가능 → 성능 최적화 적용이 쉬움
- 컴포넌트간 의존성 x → 재사용 가능
- 컴포넌트별 테스트 가능 → 제작 난이도 낮아짐

컴포넌트간 의존성을 없애는 것의 핵심이 접근 제한자임 → 가능하면 가장 낮은 접근 수준을 유지

- private: 멤버를 선언한 톱레벨 클래스에서만 접근 가능
- package-private: 멤버가 소속된 패키지 안의 모든 클래스에서 접근 가능
- protected: package-private이 가지는 접근 범위 + 상속받은 하위 클래스에서도 접근 가능
- public: 모든 곳에서 접근 가능

접근 제어자 관리 방법

- 공개할 API를 세심히 설계한 후 그 외의 모든 멤버는 private으로 설정
- 다른 클래스가 접근해야 하는 멤버에 한해 package-private으로 풀어줄 수 있다.
- 권한을 풀어줘야 하는 일이 잦다면, 다른 클래스가 하나 더 만들어져야 하는 것은 아닌지 생각해보자.
- private과 package-private 멤버는 모두 해당 클래스 내부 구현에 해당하므로 공개 API에 영향을 주지 않는다.
- 만일 private, package-private 외에 protected로 풀어준다면, 그 멤버는 API로서 계속 제공되어야 하기 때문에 신중해야 한다.
    - 이러한 이유로 protected, public의 수는 정말 최소화해야 한다
- 테스트를 위해 필드를 public으로 만들지 마라

정보인닉을 위한 클래스 설계

- public 클래스의 인스턴스 필드는 되도록 public이 아니어야 한다.
    - public 가변 필드를 갖는 클래스는 일반적으로 thread-safe하지 않다.
- public 접근자인데 필드가 final이며, 불변 객체를 참조한다고 해서 안심하면 안된다.
    - 리팩토링할 때 이미 이 클래스를 다른 클래스가 사용중이라면 public 필드는 함부로 없애지 못한다는 사실을 인지하자.
- 배열은 public final로 제공하더라도, 배열 내용을 변경할 수 있음에 유의하자.
    - public static final Thing[] VALUES = { ... };
        - 사용자는 이 배열의 내용을 변경할 수 있다.
        - 배열 값이 필요하면 clone을 사용하여 리턴하게 만든다.

item16. public 클래스에서는 public 필드가 아닌 접근자 메서드를 사용하라

접근자와 변경자를 활용할경우 장점

일반적으로 getter와 setter를 볼 수 있다. 외부에서 필드에 직접 접근하는것이 아닌 메서드를 통해 접근하게 하는 것이다.

- 단순히 public 필드를 외부에 공개하면, 추후에 클라이언트에 의해 이 클래스가 사용될 때 public 필드를 직접 이용하는 경우가 생기고, 이 경우 내부 표현 방식을 마음대로 바꿀 수 없게 된다.
- 단, package-private 클래스 혹은 private 중첩 클래스와 같은 경우는 public으로 필드 값을 노출해도 아무런 문제가 없다.
    - 이 경우 클라이언트가 이 클래스를 포함하는 패키지 안에서만 동작한다는 보장이 있기 때문이다. 그래서 public으로 공개해도 이 클래스를 사용하는 다른 클래스가 없기 때문에 상관없다.
- getter와 setter를 사용하면 값을 읽거나 변경할 때 부수적인 로직 구성이 쉽다. 마치 proxy 같이 활용할 수도 있다.

접근자와 변경자를 활용하지 않을때 단점

- public으로 공개된 멤버가 언제든 변경될 수 있다.
- 특히 가변 객체인 경우에는 final로 불변으로 만든다 해도 객체 내용 변경이 가능하므로 주의해야 한다.
- public으로 공개된 멤버가 언제든 변경될 수 있으므로 보통 방어적 복사가 이용되고 이는 성능 문제를 일으킨다.

item17. 변경 가능성을 최소화하라

불변클래스는 인스턴스 내부 값을 수정할 수 없는 클래스이다. → 객체가 소멸될때 까지 내부 정보가 변경되지않음

String, 기본 타입 박싱 클래스, BigInteger..

불변클래스는 가변클래스보다 설계하고 구현하고 사용이 쉬움 + 안전함

클래스를 불변으로 만드는 방법

- 객체의 상태를 변경하는 메서드(변경자)를 제공하지 않는다.
- 클래스를 확장할 수 없도록 한다. (상속이 불가능하게 만든다.)
    - 모든 생성자를 private 혹은 package-private으로 만들고 정적 팩터리를 제공한다.
- 모든 필드를 final로 선언한다.
- 모든 필드를 private으로 선언한다.
    - public final은 추후 API 리팩토링의 유연성을 저해할 수 있기 때문에 주의해야 한다.
- 자신 외에는 내부의 가변 컴포넌트에 접근할 수 없도록 한다.

장점

- 기본적으로 스레드 안전하여 동기화가 필요가 없다.
- 걱정없이 공유가능하다.

단점

- 값이 다르면 반드시 독립적인 객체로 만들어야함
- 값의 가짓수가 많다면 큰 비용이 들게됨

클래스는 꼭 필요한 경우가 아니면 불변인 것이 좋다.

불변을 만들 수 없는 클래슨느 변경할 수 있는 부분을 최대한으로 줄여야 한다.

특별한 이유가 없다면 private final 접근 제어자를 가지는 것이 좋다.

item18. 상속보다는 컴포지션을 사용하라

상속은 캡슐화를 깨뜨린다.

모듈을 만들대 SOLID에 의해 응집도는 높고 결합도는 낮은 모듈을 만들고 변경사항이 있을 때는 클라이언트의 코드만 변경하는 것이 이상적이다.

- 상속을 통한 하위 클래스는 상위 클래스의 구현 내용 변경에 따라 하위 클래스의 구현 내용도 바뀔 수 있다.
- 가만히 있던 하위 클래스가 오작동할 수 있다.
- 즉 상속은 캡슐화를 깨뜨릴 수 있다.

컴포지션을 사용하기

- 상속처럼 기존의 기능을 재정의하는 것이 아닌 앞뒤에 독립적인 새로운 부가기능을 넣을 수 있다.
- 기존 클래스의 내부 구현방식에 전혀 영향을 받지 않게 된다.
- 기존 클래스에 새로운 메서드가 생겨도 아무런 영향이 없다.

상속 주의

- is-a 관계인지 자문하자
- 자바 기본 API인 Stack은 Vector를 상속받았지만, 잘못된 설계이다.
- 확장하려는 클래스 API에 결함이 없어야한다. → 그대로 물려받음

상속은 강력하지만 캡슐화를 해친다. 순수한 is-a 관계에서만 사용해야한다.

상속의 문제점을 피하려면 컴포지션을 사용하자

item19. 상속을 고려해 설계하고 문서화하라. 그러지 않았다면 상속을 금지하라.

상속용 클래스가 지켜야하는 것

- **재정의할 수 있는 메서드들을 내부적으로 어떻게 이용하는지 문서로 남겨야 한다.**
    - 어떤 순서로 호출하는지, 호출 결과가 이어지는 처리에 어떤 영향을 주는지도 담아야 한다.
    - 재정의 가능 메서드란 public, protected 중 final이 아닌 모든 메서드를 말한다.
    - 재정의 가능한 메서드를 호출할 수 있는 모든 상황을 문서로 남기는 것이 좋다.
        - 백그라운드 스레드나 정적 초기화 과정에서 호출될 수도 있으므로 유의하자.

상속용 클래스를 시험하는 방법은 직접 하위 클래스를 만들어보는 것이 유일하다. 직접 시험하며 어떤 메서드를 공개할지 선택하면 된다. 만일 하위 클래스를 여러개 만드는 동안 한번도 쓰이지 않는 protected 멤버가 존재한다면, private이었어야 할 가능성이 크다. 널리 쓰일 클래스를 상속용으로 설계한다면 설계의 결정요소와 문서화의 책임이 더욱 크다. **상속용으로 설계한 클래스는 배포 전에 반드시 하위 클래스를 만들어 검증하자.**

- **상속용 클래스의 생성자는 직접적이든, 간접적이든, 재정의 가능 메서드를 호출하면 안된다.**

  → 상위 클래스의 생성자가 하위 클래스의 생성자보다 먼저 실행되기 때문이다.
- 직렬화, 객체 복사에 사용되는 clone(), readObject()와 같은 경우도 생성자와 비슷한 효과를 가지고 있으므로 직접적이든 간접적이든 재정의 가능한 메서드를 호출해선 안 된다.
    - readObject()는 역직렬화가 끝나기 전에 재정의한 메서드부터 호출하게 된다.
    - clone()는 하위 클래스의 clone() 메서드가 복제본의 상태를 올바른 상태로 수정하기 전에 재정의한 메서드를 호출한다.

        - clone()이 잘못되면 원본 객체에도 피해를 줄 수 있다.

상속용 클래스와 그 제약

- **클래스를 상속용으로 설계하려면 엄청난 노력이 들고 그 클래스 안에 제약도 상당하다.**
- **상속용으로 설계하지 않은 클래스는 상속을 금지하는 편이 버그를 줄일 수 있다.**
    - 클래스를 final로 만들어 상속을 금지한다.
    - 모든 생성자를 private 혹은 package-private으로 선언하고 public 정적 팩터리를 만든다.
- 혹여나 일반 클래스에서 상속을 허용하고 싶다면, 재정의 가능 메서드는 절대 사용하지 않도록 문서에 표기하자.

item20. 추상 클래스보다는 인터페이스를 우선하라

추상 클래스와 인터페이스의 공통점은 메서드의 시그니쳐만 만들고 구현을 구현 클래스에게 맡긴다는 것이다.

차이점

- 추상클래스: 단일 상속만 가능하다. 구현체는 추상클래스의 하위 클래스가 된다.
- 인터페이스: 다중 상속이 가능하다. 인터페이스를 구현했다면, 같은 타입으로 취급된다.

인터페이스의 장점

믹스인

- 믹스인: 구현 클래스에 '선택적 행위'를 제공한다고 선언하는 효과를 준다.
    - ex) Comparable, Iterable, AutoCloseable, Serializable

추상 클래스는 이미 다른 클래스를 상속하는 클래스의 경우, 해당 클래스가 두 부모 클래스를 가질 수는 없으므로 믹스인으로 사용될 수 없다.

계층이 없는 타입 프레임워크

- 인터페이스는 다른 인터페이스를 상속할 수 있다.
- 인터페이스의 상속은 상속이라는 단어는 사용하지만, 클래스의 상속처럼 부모, 자식 계층이 존재하지 않는다.
    - 부모 클래스의 생성자를 호출할 필요 없다.
    - 부모 클래스의 구현 내용도 이어받지 않는다.
    - 정의된 메서드들만 구현하면 된다.
- 클래스로 이와 같은 구조를 구현하려면, 상하 관계를 따져보며 차례로 단일 상속을 받아야 한다.
    - 만든 이후에도 클래스 상속이 갖는 여러가지 제약을 갖게 된다.

래퍼 클래스

- 래퍼 클래스란 기존에 인터페이스를 구현한 클래스를 주입받아 기존 구현체에 부가기능을 손쉽게 더할 수 있는 클래스다.
    - 이를 데코레이터 패턴(Decorator Pattern)이라고 한다.
    - 컴포지션(다른 클래스를 사용하는 패턴)과 전달(인터페이스 구현체를 주입)을 합쳐 위임(delegation)이라고 부른다.

만약 인터페이스의 메서드 중 구현 방법이 명확한 메서드가 있다면 디폴트 메서드를 활용할 수 있다.

item21. 인터페이스는 구현하는 쪽으로 생각해 설계하라

인터페이스의 수정과 디폴트 메서드

- 자바8 이전에는 인터페이스는 한번 정의되면 절대 새로운 메서드가 추가되거나 기존 메서드가 사라지지 않는다는 전제 하에 코드를 작성했다.
- 자바8 에서 디폴트 메서드가 등장하며 새로운 메서드를 추가할 수 있게 되었다.
    - 주로 람다를 활용하기 위해서이다.
    - 하지만 위험이 사라진 것은 아니다.
        - 생각할 수 있는 모든 상황에서 불변식을 해치지 않는 디폴트 메서드를 작성하는 것은 쉽지 않다.
        - 디폴트 메서드를 선언하면, 디폴트 메서드를 재정의하지 않은 모든 클래스에서 디폴트 구현이 쓰이게 된다.

디폴트 메서드가 기존 구현체와 충돌할 수 있다. → 디폴트 메서드로 새 메서드를 추가하는 일은 꼭 필요한 경우만 진행

인터페이스를 설계할 때는 세심한 주의를 기울여야 함

인터페이스를 릴리즈한 후라도 결함을 수정하는게 가능한 경우도 있지만, 그 가능성을 기대하면 안됨

item22. 인터페이스는 타입을 정의하는 용도로만 사용하라

안티패턴

상수 인터페이스

```
public interface PhysicalConstants {
    // 인터페이스 내부의 필드는 `static final`이 자동으로 붙는다.
    // 아보가드로 수 (1/몰)
    double AVOGADROS_NUMBER = 6.022_140_857e23;
    // 볼츠만 상수 (J/K)
    double BOLTZMANN_CONSTANT = 1.380_648_52e-23;
    // 전자 질량 (kg)
```

```
    double ELECTRON_MASS = 9.109_383_56e-31;
}
```

- 상수 인터페이스 안티패턴은 인터페이스를 잘못 사용한 예이다.
- 상수 인터페이스를 구현하는 것은 내부 구현을 클래스의 API로 노출하는 행위이다.
- 클래스가 어떤 상수 인터페이스를 사용하든 사용자에게는 아무런 의미가 없다.
- 사용자에게 혼란을 주기도 하고, 클라이언트 코드가 내부 구현에 해당하는 이 상수들에 종속되게 한다.
    - 추후에 이 상수들을 쓰지 않아도 여전히 상수 인터페이스를 구현하고 있어야 한다.

더 나은 방법

상수 유틸리티 클래스

```
public static class PhysicalConstants {
    // 아보가드로 수 (1/몰)
    public static double AVOGADROS_NUMBER = 6.022_140_857e23;
    // 볼츠만 상수 (J/K)
    public static double BOLTZMANN_CONSTANT = 1.380_648_52e-23;
    // 전자 질량 (kg)
    public static double ELECTRON_MASS = 9.109_383_56e-31;
}
```

item23. 태그 달린 클래스보다는 계층구조를 활용하라

태그달린 클래스의 예시

```
public class Item23 {
    static class Figure {
        enum Shape { RECTANGLE, CIRCLE };
```

```
// 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 this.length * this.width;
        }
        case CIRCLE ->  {
            return Math.PI * (radius * radius);
        }
        default -> {
            throw new AssertionError(shape);
        }
    }
```

```
        }
    }
```

- final Shape라는 태그 필드를 이용하여 타입을 나타내는 클래스의 예이다.
- 열거타입, 태그 필드, switch 문 등 쓸데없는 코드가 많다.
- 다른 의미를 위한 코드를 클래스 내부에 함께 저장해야 하니 메모리도 많이 사용하게 된다.
- 필드를 final로 선언하려면 해당 의미에 쓰이지 않는 필드들까지 생성자에서 초기화해야 한다.
- 또 다른 태그를 추가한다고 할 때 신경써야 할 포인트가 굉장히 많아 실수하기 쉽다.
    - switch문의 case를 빼먹는다면 area()를 호출할 때야 에러가 나게 될 것이다.

상속 구조

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

static class Circle extends Figure {
    final double radius;

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

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

```
static class Rectangle extends Figure {
    final double width;
    final double length;

    public Rectangle(double width, double length) {
        this.width = width;
        this.length = length;
    }

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

static class Square extends Rectangle {
    Square(double side) {
        super(side, side);
    }
}
```

- 각각의 클래스에 필요한 요소와 공통 요소를 확실히 분리하여 추상클래스와 구현 클래스로 나누었다.
- Sqaure는 Rectangle에서 가로세로 길이가 같다는 특성을 이용하여 생성할 수 있다.

item24. 멤버 클래스는 되도록 static으로 만들라

중첩클래스

- 다른 클래스 안에 정의된 클래스
- 오직 자신을 감싼 바깥 클래스에서만 쓰일 용도로 만들어진 클래스

중첩클래스의 종류

- 정적 멤버 클래스
- 비정적 멤버 클래스
- 익명 클래스
- 지역 클래스

정적 멤버 클래스를 제외한 나머지는 내부 클래스(inner class)에 해당한다.

정적 멤버 클래스는 다른 클래스 안에 선언되어 바깥 클래스의 private 멤버에도 접근 가능한 특성이 있다.

흔히 바깥 클래스와 함께 쓰일 때만 유용한 public 도우미 클래스로 쓰인다.

```
public class Item24 {
    private String item24Member = "item24Member";

    private class nestedClass {
        public nestedClass() {
            System.out.println("item24Member = " + item24Member
            System.out.println("Item24.this.item24Member = " + I
        }
    }

    static class nestedStaticClass {
        public nestedStaticClass() {
            // item24Member에 닿지 않음
            System.out.println("item24Member = " + item24Member
            System.out.println("Item24.this.item24Member = " + I
        }
    }
}
```

비정적 멤버 클래스는 바깥 클래스에 접근 가능하고 정적 멤버 클래스는 바깥 클래스에 접근 불가능한 특징이 있다.

- 비정적(일반적) 멤버 클래스에서는 바깥쪽 인스턴스 멤버에 접근이 가능하다.
  - 이러한 이유로 어댑터를 정의할 때 자주 쓰인다. 어떤 클래스의 인스턴스를 감싸 다른 클래스의 인스턴스처럼 보이게 하는 뷰로 사용하는 것이다.
- **정적 멤버 클래스에서는 바깥쪽 인스턴스 멤버에 접근이 불가능하다. (메모리 사용 영역이 다름)**
  - 멤버 클래스에서 바깥 인스턴스에 접근할 일이 없다면 무조건 static을 붙여 정적 멤버 클래스로 만들어두는 편이 좋다.
    - **static을 생략하면, 바깥 인스턴스로의 숨은 외부참조를 갖게 되기 때문에 시간과 공간이 소비되고, 메모리 누수가 생길 수도 있다.**

item25. 톱 레벨 클래스는 한 파일에 하나만 담으라

두 클래스가 한 파일에 정의되어 있을 때

```
Utensil.java

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

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


Dessert.java

class Utensil {
```

```
  static final String NAME = "pot";
}

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

Main.java

public class Main {
  public static void main(String[] args) {
    System.out.println(Utensil.NAME + Desert.NAME);
  }
}
```

- 위는 Utensil.java와 Dessert.java라는 두 개의 파일에 중복으로 정의된 2개의 클래스의 예이다.
- 위와 같은 경우 javac 명령어에 들어가는 인수에 따라 실행결과가 달라진다.
  - javac Main.java Dessert.java: 에러, Utensil과 Dessert 클래스가 중복 정의되었습니다.
  - javac Main.java, javac Main.java Utensil.java: "pancake" 출력
  - javac Dessert.java Main.java: "potpie" 출력
  - **동작원리**
    - Main.java가 먼저 인수에 들어왔을 때, 자바는 Main.java를 실행시키며, Utensil.NAME을 만나고, Utensil.java 파일을 찾아서 클래스를 로드하려한다.
      - 그래서 javac Main.java의 경우 "pancake"가 출력된다.
      - 그래서 javac Main.java Dessert.java의 경우 클래스가 중복으로 선언되었다고 알린다.
    - Dessert.java가 먼저 인수에 들어왔을 때는, 자바는 Utensil 클래스와 Dessert 클래스의 정의를 불러와 놓는다.

- 그래서 javac Dessert.java Main.java의 경우 "potpie"가 출력된다.

파일을 나누면 위와 같은 복잡한 동작원리도 알 필요 없고, 잠재적 에러도 없으므로 Utensil.java와 Dessert.java 각 파일별로 클래스는 1개씩만 선언하는 것이 좋다.

굳이 한 파일에 여러 클래스를 정의하고 싶다면?

```
public class Main {
  public static void main(String[] args) {
    System.out.println(Utensil.NAME + Dessert.NAME);
  }

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

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