4장 클래스와 인터페이스

아이템 15. 클래스와 멤버의 접근 권한을 최소화하라.

잘 설계된 컴포넌트?

- 클래스의 내부 구현 정보를 외부 컴포넌트로 부터 숨기고, 오직 api를 통해서만 다른 컴포넌트와 소통하며 서로 내부 동작 방식에는 개의치 않는 것이다.

정보은닉의 기본 원칙

- **모든 클래스와 멤버의 접근성을 가능한 한 좁혀야한다.**

정보 은닉의 장점

- 여러 컴포넌트를 병렬로 개발할 수 있기 때문에 시스템 개발 속도를 높일 수 있다.
- 각 컴포넌트를 더 빨리 파악하여 디버깅 할 수 있고, 다른 컴포넌트로 교체하는 부담이 적어지므로 시스템 관리 비용을 낮춘다.
- 정보 은닉 자체가 성능을 높여주지는 않지만, 성능 최적화에 도움을 준다. 완성된 시스템을 프로파일링 해 최적화할 컴포넌트를 정한 다음 다른 컴포넌트에 영향을 주지 않고 해당 컴포넌트만 최적화할 수 있기 때문이다.
- 소프트웨어 재사용을 증대시킨다. 외부에 의존하지 않고 독자적으로 동작할 수 있는 컴포넌트라면 그 컴포넌트와 함께 개발되지 않은 낯선 환경에서도 유용하게 쓰일 가능성이 크다.
- 큰 시스템을 제작하는 난이도를 낮춰준다. 시스템 전체가 완성되지 않은 상태에서도 개별 컴포넌트의 동작을 검증할 수 있기 때문이다.

멤버에 부여할 수 있는 접근 수준

1. public

- 어디서든 접근할 수 있다. 외부 클래스, 패키지, 다른 모듈에서도 접근이 가능하다.

2. protected

- 동일한 패키지 내의 클래스 밑 해당 클래스를 상속한 하위 클래스에서 접근할 수 있다.

3. default

- 접근 수준을 명시하지 않는 경우, 해당 멤버는 기본 접근 수준을 가진다. 동일한 패키지 내의 클래스에서만 접근할 수있다.

4. private

- 선언된 클래스 내에서만 접근할 수있다. 외부 클래스에서는 접근 할 수 없다.(정보 은닉을 위해 사용)

public 클래스의 인스턴스 필드는 되도록 public이 아니어야 한다.

- 클래스의 내부 구현 세부사항을 외부에 노출 시키지 않고, 외부에서 직접적으로 접근하여 변경하지 못하도록 하는 것을 의미한다. 가능한 한 클래스의 인스턴스 필드는 private로 선언하고, 필요한 경우에만 접근자와 설정자를 제공하여 외부에서 간접적으로 접근하도록 하는 것이 바람직하다.

public 가변 필드를 갖는 클래스는 일반적으로 스레드 안전하지 않다.

- 멀티 스레드 환경에서 해당 클래스의 인스턴스를 여러 스레드가 동시에 접근하고 수정할 때 문제가 발생할 수 있다.

클래스에서 public static final 배열 필드를 두거나 이 필드를 반환하는 접근자 메서드를 제공해서는 안된다.

- public : 모든 클래스에서 접근할 수있다.
- static : 클래스의 인스턴스화 없이 사용할 수 있다.
- final : 한 번 초기화 되면 그 값이 바뀌지 않는다.(상수로 취급)

```
public class Constants {
    public static final String[] COLORS = {"Red", "Green", "B

    public static String[] getColors() {
        return COLORS;
    }
}
```

해당 코드의 문제점

- **불변성 보장의 문제점** : 배열은 수정할 수 없는 상수로 간주 되지만 해당 배열을 참조하는 다른 변수들이 배열 요소의 내용을 변경할 수 없기 때문이다.
- **캡슐화 원칙 위배** : 외부에서 COLORS 배열에 직접 접근할 수 있기 때문에 원칙에 위배된다.

```
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class Constants {
    // private static final로 선언된 배열 COLORS
    private static final String[] COLORS = {"Red", "Green", "

    // getColors 메서드: COLORS 배열을 변경할 수 없는 리스트 형태로 반
    public static List<String> getColors() {
        // Arrays.asList 메서드를 사용하여 COLORS 배열을 리스트로 변
        // 이 메서드는 고정 크기의 리스트를 반환하므로 리스트의 크기는 변
        // unmodifiableList 메서드를 사용하여 리스트를 변경할 수 없는
        return Collections.unmodifiableList(Arrays.asList(COL
    }
}
```

이렇게 수정하면 변경할 수 없는 형태가 된다.

아이템 16. public 클래스에서는 public 필드가 아닌 접근자 메서드를 사용하라.

public으로 필드를 공개한다면, 의도치 않은 변경에 노출됨으로써 불변을 보장할 수 없게 된다.

다음의 코드는 이 원칙을 어긴 사례다.

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

이와 같이 필드를 public으로 열어버리면, 아이템 15에서 다루었던 이점들을 전혀 취할 수 없게 된다.

```
public class Dimension extends Dimension2D implements java.io

    /**
     * The width dimension; negative values can be used.
     *
     * @serial
     * @see #getSize
     * @see #setSize
     * @since 1.0
     */
    public int width;

    /**
     * The height dimension; negative values can be used.
     *
     * @serial
     * @see #getSize
     * @see #setSize
     * @since 1.0
     */
    public int height;

    // ...
```

본문에서 언급했던 Dimension 클래스는 실제로 위의 코드처럼 필드가 public으로 열려있다.

```
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 클래스라면 위와 같이 필드를 private로 감추고, getter로 접근할 수 있게 만들어야 한다.

아이템 17. 변경 가능성을 최소화하라.

불변 클래스 : 인스턴스의 내부 값을 수정할 수 없는 클래스

불변으로 설계하는 이유 : 가변 클래스보다 설계와 구현하가 쉽고 오류가 생길 여지가적어 안전하다.

객체를 불변으로 만드는 규칙

- 객체의 상태를 변경하는 메서드(변경자)를 제공하지 않는다.
- 클래스를 확장할 수 없도록 한다.
- 모든 필드를 final로 선언한다.
- 모든 필드를 private으로 선언한다.
- 자신 외에는 내부의 가변 컴포는트에 접근할 수 없도록 한다.

```
package effective.item17;

public final class Complex {
  private final double re; // 실수
  private final double im; // 허수
```

```
  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 times(Complex c) {
    return new Complex(re * c.re - im * c.im, re * c.im + im
  }

  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 (this == o) return true;
    if (!(o instanceof Complex)) return false;
    Complex complex = (Complex) o;
    return Double.compare(complex.re, re) == 0 && Double.compa
  }

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

```
  }

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

- Complex 클래스는 복소수(실수, 허수)를 표현한 클래스이다.
  - 실수와 허수를 반환하는 접근자 메서드 : realPart(), imaginaryPart()
- 해당 클래스를 자세히 보면 사칙연산 메서드는 모두 새로운 Complex 인스턴스를 만들어서 반환한다. 이러한 피연산자 자체가 그대로인 프로그래밍 패턴을 함수형 프로그래밍이라고 한다
- 메서드 이름으로 동사 대힌 전치사를 사용하여 객체의 값이 변하지 않는다는 의미의 이름을 적용했다. BigInteger, BigDicimal 클래스 또한 객체의 값이 변경되지 않지만 잘못 사용해 오류가 발생하는 일이 자주 있다.
- 불변 객체는 생성된 시점에서 상태를 파괴할 때 까지 그대로 유지한다. 즉, 영원히 불변으로 남는다. 반면 가변 객체는 임의의 복잡한 상태에 놓일 수 있다. setter메서드를 정밀하게 문서로 남겨놓지 않은 가변 클래스는 믿고 사용하기 어려울 수도 있다.
- **불변 객체는 근본적으로 스레드 안전하여 따로 동기화할 필요 없다.**
  - 여러 스레드가 동시에 접근하더라도 값이 변하지 않기 때문에 절대 훼손되지 않는다. 따라서 안심하고 동기화 설정 없이 공유할 수 있다.
  - **스레드 안전** : 여러 스레드가 동시에 어떤 객체나 코드를 접근하거나 실행할 때 오류나 충돌 없이 올바르게 작동하는 것, 여러 스레드가 공유 자원에 접근할 때 데이터의 불일치나 예기치 않은 결과가 발생하지 않는 상태를 보장하는 것이다.
    - 스레드가 안전하지 않은 경우의 문제
      - 스레드가 안전하지 않은 코드는 여러 스레드가 동시에 동일한 자원에 접근하면서 경쟁상태가 발생 할 수 있다. 이 경쟁상태에서 한 스레드가 자원을 수정하는 동안 다른 스레드가 그 자원을 읽거나 수정할려고 할 때, 예기치 않은 오류나 데이터 불일치가 발생할 수 있다.

```
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);
```

- 불변 클래스에서 자주 사용되는 인스턴스는 다음과 같이 정적 팩터리로 캐싱하는 것이 적합하다.
    - -불변 객체를 자유롭게 공유할 수 있다는 점은 방어적 복사도 필요없다는 결론으로 이어진다. 아무리 복사해도 원본과 같으니 clone 메서드나 복가 생성자를 제공하지 않는것이 좋다. String의 복사 생성자는 이 사실을 잘 이해하지 못한 자바 초창기 때 만들어진 것으로 사용하지 않는 것이 좋다.

불변 클래스 장점

- **불변 객체는 단순하다.**
- **스레드 세이프**하여 동기화 할 필요 없다.
- 정적 팩터리 등으로 **자주 사용되는 인스턴스를 캐싱**할 수 있다.
- 불**변 객체 간 내부 데이터를 공유**할 수 있다.
- 객체를 만들 때 다른 불변 객체들을 구성요소로 사용하면 이점이 많다.
- **불변 객체는 그 자체로 실패 원자성을 제공**한다. 상태가 절대 변하지 않으니, 잠깐이라도 불일치 상태에 빠질 가능성이 없다.
- **안심하고 공유**가 가능하다.(방어적 복사 필요x , 단 하위 클래스가 신뢰할 수 없다면 방어적 복사 필요)
    - **방어적 복사** : 객체를 외부에 반환하거나 외부에서 전달된 객체를 내부에서 사용할 때, 원본 객체를 직접 사용하지 않고 복사본을 만들어 사용하는 방식이다. 이를 통해 객체의 불변성을 유지하고, 예상치 못한 수정이나 변경으로부터 보호할수 있다.
        - **방어적 복사가 필요한 이유**
            - 불변성 보장
            - 클라이언트 코드의 무분별한 변경 방지
            - Mutable 객체 보호 : 자바에서 많은 객체들이 가변적으로 만들어져 있기 때문에, 방어적 복사는 중요한 가면 객체에 대해 예기치 않은 수정을 막는 수단으로 사용된다.
            - Getter에서의 방어적 복사

```
public class Period {
    private final Date start;
    private final Date end;
```

```
    public Period(Date start, Date end) {
        this.start = new Date(start.getTime()); // 방
        this.end = new Date(end.getTime());     // 방
    }

    public Date getStart() {
        return new Date(start.getTime()); // 방어적 복
    }

    public Date getEnd() {
        return new Date(end.getTime());   // 방어적 복
    }
}
```

위의 코드에서 Date 코드는 가변 객체이다. 만약 외부에서 getStart()나 getEnd()로 반환된 Date객체를 수정하면,period 객체 내부의 상태가 변경될 수 있기 때문에 이를 방지하기 위해 복사된 Date 객체를 반환한다.

- Setter에서의 방어적 복사

```
public class Period {
    private final Date start;
    private final Date end;

    public Period(Date start, Date end) {
        this.start = new Date(start.getTime()); // 방
        this.end = new Date(end.getTime());     // 방
    }

    public void setStart(Date start) {
        this.start = new Date(start.getTime()); // 방
    }

    public void setEnd(Date end) {
        this.end = new Date(end.getTime()); // 방어적
    }
}
```

생성자에서 전달된 Date 객체를 그대로 할당하면, 나중에 외부에서 Date객체를 변경할 때 Period 내부의 start나. end 값이 변경될 수있다. 이를 막기위해 복사본을 저장함으로써 외부에서 원본 Date 객체가 수정되더라도 클래스 내부의 데이터는 안전하게 보호된다.

- **왜 final로 선언했는데 값이 바뀔 수 있을까?**
  - final의 의미 : final로 선언된 참조 타입 변수는 초기화된 이후에 다른 객체를 참조하지 못하게 제한하는 것이다. 하지만 해당 참조 변수가 가리키는 객체 내부의 상태는 여전히 변경가능할 수 있다.

```
class MyClass {
    private final List<String> list = new ArrayLi

    public List<String> getList() {
        return list;  // list 객체의 참조를 반환
    }
}

MyClass obj = new MyClass();
List<String> list1 = obj.getList();
list1.add("Hello");  // 외부에서 list에 값 추가 가능
```

  - 해당 코드에서 list는 Final로 선언되었기 때문에, list 변수가 다른 list 객체를 참조하도록 변경할 수는 없다. 하지만 리스트에 요소를 추가하거나 삭제하는 것은 가능하다. 즉 , ㅣ ist는 여전히 가변적이므로 외부에서 그 참조를 통해 상태를 변경할 수 있다.
  - 자바에서 객체는 참조에 의해 전달되기 때문에, 메서드를 통해 가변 객체를 반환하거나 전달하면 그 참조를 통해서 외부에서 객체의 필드를 수정할 수있다. final은 그 객체가 다른 객체로 참조가 변경되지 않도록 고정해줄 뿐, 객체 자체의 내용은 변경될 수 있다.

불변 클래스 단점

- 값이 다르면 반드시 독립적인 객체로 만들어야한다.
- 원하는 객체를 완성하기까지의 단계가 많고, 중간 단계에서 생성된 객체들이 버려진다면 성능 문제가 발생한다.

public 정적 팩터리 제공으로 불변 클래스 만드는 방법

- 불변 클래스고 만드는 기본적인 방법은 final 클래스로 만드는 것이었다. 그 이유는 클래스가 불변임을 보장하려면 자신을 상속하지 못하게 해야하기 때문이다. 가장 쉬운 방법은 final을 사용하는 것이고, 더 유연한 방법이 정적 팩터리 제공이다.

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

  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);
  }
}
```

모든 생성자를 private or package-private으로 만들고, public 정적 팩터리를 제공한 예시이다. 클래스 자체는 public이지만 패키지 바깥에서 바라본 클라이언트 입장에서는 이 객체는 불변이다.

불변 객체가 되는 이유 : re,im을 final로 선언하여 객체가 초기화된 이후에는 값을 변경할 수 없게 하였고, public static Complex valueof(double re,double im)이라는 정적 팩터리는 Complex 객체를 생성하는 유일한 진입점이다. 이 메서드는 새로운 객체를 반환하되, 객체의 상태가 한번 설정되면 변경되지 않도록 하고 있다. 또한 setter 메서드의 부재로 인해 내부상태를 변경할 수 없다.

불변 클래스에서 만들 수 있는 캐싱 기능

- 불변 클래스의 규칙 목록에 따르면 모든 필드가 final 이어야하고, 어넌 메서드도 그 객체를 수정할 수 없어야한다고 명세되어있다. 하지만 객체가 불면임이 보장 된다면 성능을 향상하기 위해 다음과 같은 캐싱 기능을 제공할 수 있다.

아이템 18. 상속보다는 컴포지션을 사용하라.

상속은 코드를 재사용하는 강력한 수단이지만, 잘못 사용하면 오류를 내기 쉬운 소프트웨어를 만들게 된다. 동일한 프로그래머가 만든 패키지 안에서 상속한다면 안전할 수 있지만, 다른 패키지의 구체 클래스를 상속하는 것은 위험하다.

상속의 문제점

- 상속은 상위 클래스의 설계에 따라 캡슐화를 깨뜨리는 등의 하위 클래스에 문제를 일으킬 수 있다.
- ex) HashSet을 사용하는 프로그램에서 성능을 높이기 위해 처음 생성도니 이후 원소가 몇개 더해졌는지 등의 기능을 추가

```
import java.util.Collection;
import java.util.HashSet;
import java.util.List;

public class InstrumentedHashSet<E> extends HashSet<E> {

  // 추가된 원소의 수
  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;
  }
}
```

add와 addAll을 재정의하여 추가된 원소의 수를 계산할 수 있도록 구현

```
class InstrumentedHashSetTest {
  public static void main(String[] args) {
    InstrumentedHashSet<String> hs = new InstrumentedHashSet<
    hs.addAll(List.of("1", "2", "3", "4"));
    System.out.println(hs.getAddCount()); // 8 반환
  }
}
```

addAll 메서드는 컬렉션의 모든 요소를 HastSet에 추가한다. 이때, addAll은 내부적으로 반복문을 돌며 add 메서드를 호출한다.

- 반복문이 명시되어 있지 않은데?
    - HashSet 클래스의 addAll() 메서드는 내부적으로 각 요소를 하나씩 add() 메서드로 처리하는데, 이 과정에서 반복문이 사용된다. 이는 자바 컬렉션 API의 일반적인 구현 방식이다.

addAll 메서드를 재정의한 부분에서 c.size()만큼 addCount에 더한 뒤 다시 super.addAll(c)를 호출하고있다. 해당 호출은 결국 각각의 요소에 대해 add(E e)메서드를 호출하고있다. 1,2,3,4, 즉, 4만큼 증가후 super.addAll(c)가 실행되면서 Hashset은 내부적으로 add메서드를 호출하므로 다시 4번 호출되므로 8이 된다.

따라서 정상 작동하게 만드려면 addÅll 메서드를 호출하지 않거나, 직접 구현해야한다.

지금같은 상황을 자신의 다른 부분을 사용하는 자기사용 방식이라한다. 해당 기능은 다음 릴리즈에서도 유지될지는 모르기 때문에 addAll을 재정의하지 않는다고 현재 프로그램이 계속 동작할 것이라는 보장이 없다. 또한 하위 클래스에서 접근할 수 없는 private필드라면 별도의 구현도 불가능하다.

상속 대신 컴포지션 사용

- 컴포지션이란 기존 클래스를 확장하는 대신, 새로운 클래스를 만들고 private필드로 기존 클래스의 인스턴스를 참조하도록 하는 것을 의미한다. 기존 클래스가 새로운 클래스의 구성요소로 쓰인다는 뜻에서 이 설계를 컴포지션이라 한다.
- 컴포지션 예시

```
// 컴포지션의 예시 :
// 독립된 기능을 가진 클래스: 소리 생성기
class SoundMaker {
    public void makeSound(String sound) {
        System.out.println(sound);
    }
}

// 개 클래스에서 소리 생성기를 사용
class Dog {
    private SoundMaker soundMaker;

    public Dog() {
        this.soundMaker = new SoundMaker();
    }

    // 전달 메서드: Dog 클래스의 bark 메서드는 내부적으로 SoundMaker의
    public void bark() {
        soundMaker.makeSound("멍멍!"); // 위임: Dog 객체는 짖는 기
    }
}
```

컴포지션과 전달(위임)의 장점 :

- 낮은 결합도 : 객체간의 결합도를 낮추고, 유연성을 확장시킨다.
- 캡슐화 강화 : 내부 구현을 숨기고 필요한 인터페이스만 노출시켜 캡슐화 강화
- 재사용성 향상 : 구성요소를 쉽게 교체하거나 재사용이 가능하여 상황에 맞게 조정이 가능하다.

```
import java.util.Collection;
import java.util.Iterator;
import java.util.Set;
```

```
// 기존의 Set 인터페이스 메서드를 ForwardingSet으로 전달
public class ForwardingSet<E> implements Set<E> {
  private final Set<E> s; // private 으로 Set 인스턴스를 참조
//이 클래스는 Set 인터페이스를 구현하고, 실제 Set 인스턴스에 대한 메서드
//위임
  public ForwardingSet(Set<E> s) {
    this.s = s;
  }

  @Override
  public int size() {
    return s.size();
  }

  @Override
  public boolean isEmpty() {
    return s.isEmpty();
  }

  @Override
  public boolean contains(Object o) {
    return s.contains(o);
  }

  @Override
  public Iterator<E> iterator() {
    return s.iterator();
  }

  @Override
  public Object[] toArray() {
    return s.toArray();
  }

  @Override
  public <T> T[] toArray(T[] a) {
    return s.toArray(a);
  }
```

```
@Override
public boolean add(E e) {
  return s.add(e);
}

@Override
public boolean remove(Object o) {
  return s.remove(o);
}

@Override
public boolean containsAll(Collection<?> c) {
  return s.containsAll(c);
}

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

@Override
public boolean retainAll(Collection<?> c) {
  return s.retainAll(c);
}

@Override
public boolean removeAll(Collection<?> c) {
  return s.removeAll(c);
}

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

@Override
public int hashCode() {
```

```
    return s.hashCode();
  }

  @Override
  public boolean equals(Object obj) {
    return s.equals(obj);
  }

  @Override
  public String toString() {
    return s.toString();
  }
}
```

```
import java.util.*;

public class InstrumentedSet<E> extends ForwardingSet<E> {

  // 추가된 원소의 수
  private int addCount = 0;
//해당 클래스는 forwardingset클래스를 상속받고 addcount 필드 추가
  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;
  }
}
```

해당 코드는 컴포지션을 이용해 InstrumentedSet이 Set의 기능을 변경하지 않고 그대로 유지하면서 기능을 확장하였다. 이로 인해 코드가 더 간결해지고, 변화에 더 쉽게 대응할 수 있다.

데코레이트 패턴 : InstrumentedSet은 ForwardingSet을 데코레이팅하여 원소 추가 횟수를 기록하는 기능을 추가하였다.

Is-a 관계는 상속을 이용하여 한 클래스가 다른 클래스의 특별한 형태임을 나타내는 관계다. 래퍼 클래스는 기본 데이터 타입의 값을 객체로 변환해야할 때 사용된다.

```
// 래퍼 클래스의 예
int i = 5; // 기본 타입 int 사용

Integer integerObject = new Integer(5); // Integer 객체 생성 (래
Integer autoBoxedInteger = 5; // 오토 박싱을 통한 Integer 객체 생성

// 오토 언박싱: Integer 객체에서 int 기본 타입 값으로 자동 변환
int unboxedInt = integerObject;

//오토 박싱은 기본 데이터 타입의 값을 해당하는 래퍼 클래스의 객체로 자동 변
//언박싱은 그 반대 과정을 의미합니다.
```

상속을 사용하는 상황

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

아이템 19. 상속을 고려해 설계하고 문서화하라. 그러지 않았다면 상속을 금지하라.

메서드를 재정의하면 어떤 일이 일어나는지 정확히 정리하여 문서로 남겨야한다. 상속용 클래스는 재정의할 수 있는 메서드들을 내부적으로 어떻게 이용하는지(자기 사용 패턴) 문서로 남겨야한다.

자기 사용 패턴

- 클래스 내부의 한 메서드가 같은 클래스의 다른 메서드를 호출하는 패턴. 자기 사용 팬턴이 상속과 결합될 때 문제가 발생할 수 있다. 슈퍼클래스가 자신의 메서드를 호출할 때 실제로 실행되는 메서드가 서브클래스에서 오버라이드된 메서드일 수 있다.

API문서의 Implementation Requirements

```java
/**
 * {@inheritDoc}
 *
 * <p>This implementation iterates over the collection looking for the
 * specified element.  If it finds the element, it removes the element
 * from the collection using the iterator's remove method.
 *
 * <p>Note that this implementation throws an
 * <tt>UnsupportedOperationException</tt> if the iterator returned by this
 * collection's iterator method does not implement the <tt>remove</tt>
 * method and this collection contains the specified object.
 *
 * @throws UnsupportedOperationException {@inheritDoc}
 * @throws ClassCastException            {@inheritDoc}
 * @throws NullPointerException          {@inheritDoc}
 */
public boolean remove(Object o) {
  Iterator<E> it = iterator();
  if (o == null) {
    while (it.hasNext()) {
      if (it.next() == null) {
        it.remove();
        return true;
      }
    }
  } else {
    while (it.hasNext()) {
      if (o.equals(it.next())) {
        it.remove();
        return true;
      }
    }
  }
  return false;
}
```

- 위 설명에서는 iterator 메서드를 재정의하면 remove 메서드의 동작에 영향을 줄 수 있음을 명세하고 있다.
- iterator 메서드는 재정의할 수 있는 메서드이다.

상속은 캡슐화를 해친다.

- 좋은 API문서는 '어떻게'가 아닌 '무엇을'하는지 설명해야한다 라는 격언과 반대로 상속은 캡슐화를 해치기 때문에 자세히 작성해야한다.

상속용 클래스 설계시 유의사항

- **hook 메서드**
  - 효율적인 상속용 클래스 설계를 위해서는 클래스의 내부동작 과정 중간에 끼어들 수 있는 훅을 잘 선별하여 protected 메서드로 공개야할 수도 있다.
- **상속용으로 설계한 클래스는 배포 전에 반드시 하위 클래스를 만들어 검증해야 한다.**
  - 꼭 필요한 protected 멤버를 놓쳤다면 하위 클래스를 작성할 때 확연히 드러난다. 또한 하위 클래스를 여러 개 만들때 까지 쓰이지 않는 protected 멤버는 사실 private이었어야할 가능성이 높다.
  - 이런 하위 클래스는 3개 정도가 적당하고, 하나 이상은 제 3자가 작성하는 것이 좋다.
  - 많이 쓰일 클래스를 상속용으로 설계한다면 반드시 문서화한 내부 사용 패턴과 protected 메소드와 필드를 구현하면서 이 결정이 해당 클래스의 성능과 가능에 계속 영향을 미친다는 것을 알아야한다.
    - 상속용으롤 설계한 클래스는 배포 전에 반드시 하위 클래스를 만들어 검증해야 한다.
- **상속용 클래스의 생성자는 직접적으로든 간접적으로든 재정의 가능 메소드를 호출해서는 안된다.**
  - 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 않는다.

```
public class Super(){
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 메소드, 상위 클래스의 생성자가 호출한다.
        @Override
        public void overrideMe() {
            System.out.println(instant);
        }
        public static void main(String[] args) {
            Sub sub = new Sub();
            sub.overrideMe();
            }
```

이런식의 예시가 있다면 instant를 두 번 출력할 것이라는 기대가 있지만, 첫 번째는 상위 클래스의 생성자가 하나 위 클래스의 생성자의 인스턴스 필드를 초기화 하기 전에 호출하기 때문에 null값이 호출된다.

- 상위 클래스의 생성자가 실행되는 중에 Sub 클래스의 overrideMe()메소드를 호출하면, Sub 클래스의 필드가 아직 초기화 되지 않은 상태이다. 이는 자바에서 상위 클래스의 생성자가 먼저 호출된 후 자식 클래스의 생성자가 실행되기 때문이다.
- 즉, 상위 클래스 생성자에서 overrideMe()를 호출할 때, instant 필드는 아직 초기화되지 않았으므로 null 값을 출력할 수 있다.

- **clone과 readObject 모두 직접적으로든 간접적으로든 재정의 가능 메소드를 호출해서는 안된다.**
  - cloneable이나 Serializable을 구현할지 정해야 한다면 이들을 구현할 때 따르는 제약도 생성자와 비슷하다는 것에 주의해야한다.
  - readObject의 경우 하위 클래스의 상태가 미처 다 역직렬화되기 전에 재정의한 메소드 부터 호출하게 된다.
    - 직렬화와 역 직렬화
      - 직렬화 : 객체의 상태를 바이트 스트림으로 변환하는 과정. 이 과정에서 객체의 필드 값들이 저장

        - 역 직렬화 : 바이트 스트림으로 부터 객체를 복원하는 과정. 객체의 필드가 직렬화된 데이터로 복구된다.
    - readObject를 사용한 역직렬화 과정에서는, 자바는 상위 클래스 부터 하위 클래스까지 차례대로 필드의 값을 복구한다.
    - 이 과정에서 하위 클래스의 필드가 아직 역직렬화 되지 않았는데 재정의된 메소드가 먼저 호출될 수 있다.
  - clone의 경우 하위 클래스의 clone 메소드가 복제본의 상태를 올바르게 수정하기 전에 재정의한 메소드를 호출한다.
  - clone이 잘못되어 깊은 복사를 하다가 원본 객체의 일부를 참조하고 있다면 원본 객체에게 까지도 피해를 줄 수있다.
- **Serializable을 구현한 상속용 클래스가 readResolve나 write Replace 메소드를 갖는다면 이 메소드들은 private이 아닌 protected로 선언해야한다.**
  - private로 선언한다면 하위 클래스에서 무시된다.
  - 상속을 허용하기 위해 내부 구현을 클래스 API로 공개하는 예 중 하나다.
  - 상속용으로 설계하지 않은 클래스는 상속을 금지하는 것이 좋다.

상속을 금지하는 방법

- 클래스를 final로 선언한다.
- 모든 생성자를 private나 package-private로 선언하고 public 정적 팩터리를 만들어 준다.

아이템 20. 추상 클래스 보다는 인터페이스를 우선하라.

인터페이스란?

- 인터페이스를 구현하는 클래스가 제공해야하는 메서드(추상메서드)집합을 지정하는 유형.

인터페이스와 추상 클래스의 차이

- 추상 클래스가 정의한 메서드를 구현하는 클래스는 반드시 추상 클래스의 하위 클래스가 되어야 같은 타입으로 취급한다.(새로운 타입을 정의하는데 제약)
- 인터페이스가 정의한 메서드를 모두 정의하고 일반 규약을 잘 지킨 클래스라면 다른 어떤 클래스를 상속했든 상관없이 같은 타입으로 취급된다.

```
public class SingerSongWriter extends Song implements Singer{
   ...
}

public class SingerSongWriter2 extends Song2 implements Singe
   ...
}

public static void main(String[] args){
   Singer s = new SingerSongWriter();
   Singer s2 = new SingerSongWriter2();
}
```

해당 코드와 같이 Singer라는 인터페이스를 정의한 클래스라면 모두 Singer 타입으로 취급할 수 있다.

인터페이스의 장점

- **기존 클래스에도 손쉽게 새로운 인터페이스를 구현해넣을 수 있다.**
  - 클래스 선언에 implements를 사용하여 정의하고 인터페이스에서 정의하는 메서드만 구현하면 된다.
- **믹스인 정의에 안성맞춤**
  - 믹스인이란 클래스가 구현할 수 있는 타입으로, 믹스인을 구현한 클래스에 원래의 '주된 타입' 외에도 특정 선택적 행위를 제공한다고 선언하는 효과를 준다. 쉽게 말해 다른 클래스에서 이용할 메서드를 포함한 클래스이다.
  - 하지만 추상클래스에서는 기존 클래스가 다른 클래스를 상속하고 있다면 또 다른 클래스를 상속할 수 없으므로 믹스인을 정의하지 않는다.
- **계층구조가 없는 타입 프레임워크를 만들 수 있다.**
  - 타입을 계층적으로 정의하면 수많은 개념을 구조적으로 잘 표현할 수 있지만, 현실에는 계층을 엄격히 구분하기 어려운 개념도 있다.

  Ex)

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

```
}

public interface SongWriter{
  public void compose();
}

public class People implements Singer, SongWriter{
  @Override
  public void sing() {}
  @Override
  public void compose() {}
}
public interface SingerSongWriter extends Singer, SongWrit
  public void actSensitive();
}
```

- 현실에서는 작곡을 겸하는 가수도 많다. 이러한 개념은 인터페이스에서 만들기 편하다. 이 처럼 인터페이스의 경우 두 가지 이상 확장하고 새로운 메서드까지 추가한 제 3의 인터페이스를 정의할 수도 있다.
- 하지만 추상 클래스로 구현했다면 두 개 이상의 클래스를 상속할 수 없기 때문에 추상 클래스를 만들어 추상 메소를 추가할 수 밖에 없다. 그 결과 계층 구조를 만들기 위해 많은 조합이 필요해 결국 고도 비만의 계층구조가 만들어질 것이다.

단순 구현

- 단순구현이란 골격 구현의 작은 변종으로 골격 구현과 같이 상속을 위해 인터페이스를 구현한 것이지만, 추상 클래스가 아니라는 점에서 차이점을 가지고 있다. 단순 구현은 추상 클래스와 다르게 그대로 써도 되거나 필요에 맞게 확장해도 된다.

java.util.AbstractMap.SimpleEntry

- `java.util.Map.Entry` 인터페이스를 구현하는 간단한 구현체입니다. `Map.Entry` 는 맵의 키-값 쌍을 나타내는 인터페이스이며, `SimpleEntry` 클래스는 이 인터페이스를 구현함으로써 키-값 쌍을 저장하고 검색하는 기능을 제공한다.
- ‘SimpleEntry’클래스는 다음과 같은 생성자를 제공한다.

```
public SimpleEntry(K key, V value)
```

- 이 생성자는 주어진 키와 값으로 'SimpleEntry'객체를 행성한다.
- 또한'SimpleEntry' 클래스는 'Map.Entry' 인터페이스의 다음 메서드를 구현한다.

```
public K getKey()
public V getValue()
public V setValue(V value)
```

- `getKey()` 메서드는 이 `SimpleEntry` 객체의 키를 반환하며, `getValue()` 메서드는 이 객체의 값(value)을 반환한다. `setValue()` 메서드는 이 객체의 값(value)을 주어진 값으로 설정하고, 이전 값(value)을 반환한다.

  `SimpleEntry` 클래스는 불변(immutable) 클래스가 아니므로, `setValue()` 메서드를 사용하여 값을 변경할 수 있다. 이 클래스는 `hashCode()` 및 `equals()` 메서드도 구현한다. 따라서 이 클래스의 객체는 맵의 키 또는 값으로 사용될 수 있다.

아이템21. 인터페이스는 구현하는 쪽을 생각해 설계하라.

- 기존 인터페이스에 디폴트 메서드 구현을 추가하는 것은 위험한 일이다.
  - 디폴트 메서드는 구현 클래스에 대해 아무것도 모른 채 합의 없이 무작정 **삽입**될 뿐이다.
    - 디폴트 메서드 : 인터페이스에서 메서드의 기본 구현을 제공할 수 있도록 해준다. 인터페이스의 특성상 원래는 추상 메서드만 가질 수 있었는데, 디폴트 메서드를 통해 인터페이스에도 구체적인 메서드를 정의할 수 있게 된 것이다. 이로 인해 기존 인터페이스를 구현하는 클래스에 영향을 주지 않고도 인터페이스를 확장할 수 있게 되었다.
    - 디폴트 메서드의 특징

    1. 메서드 본문 포함 :디폴트 메서드는 default 키워드를 사용하여 인터페이스에 선언되며, 일반 메서드처럼 메서드 본문(구현)을 포함할 수 있다.
    2. 호출 방법 : 인터페이스를 구현한 클래스에서 디폴트 메서드를 따로 구현하지 않으면 인터페이스에 정의된 디폴트 메서드가 자동으로 사용된다. 필요하면 이를 재정의(오버라이드)할 수 있다
    3. 다중 상속 문제 해결 : 디폴트 메서드를 여러 인터페이스에서 상속받을 때, 메서드가 충돌하면 클래스에서 이를 명시적으로 오버라이드하거나 선택해야 한다.
  - 디폴트 메서드는 기존 구현체에 런타임 오류를 일으킬 수 있다.

- 상위 클래스는 상위 인터페이스를 우선순위에서 이기게 됨
- 인터페이스를 설계할 때에는 세심한 주의를 기울어야한다.
  - 서로 다른 방식으로 최소한 세 가지는 구현을 해보자

ex) removeIf 메서드

```
default boolean removeIf(Predicate<? super E> filter) {
    Objects.requireNonNull(filter);
    boolean result = false;

    for (Iterator<E> it = iterator(); it.hasNext(); ) {
        if (filter.test(it.next())) {
            it.remove();
            result = true;
        }
    }
    return result;
}
```

- 이 메서드는 주어진 Boolean 함수(predicate)가 true를 반환하는 모든 원소를 제거한다.
- 디폴트 구현은 반복자를 이용해 순회하면서 각 원소를 인수로 넣어 predicate 를 호출하고, predicate가 true를 반환하면 반복자의 remove 메서드를 호출해 그 원소를 제거하는 방식
- 이 코드보다 더 범용적으로 구현하기도 어렵겠지만, 현존하는 모든 Collection 구현체와 잘 어우러지는 것은 아니다.

아이템22. 인터페이스는 타입을 정의하는 용도로만 사용하라.

인터페이스는 자신을 구현한 클래스의 인스턴스를 참조할 수 있는 타입의 역할을 수행한다.

상수 인터페이스

- 메서드 없이, 상수를 뜻하는 static final 필드로만 가득 차 인터페이스를 말한다.
- 이 상수들을 사용하는 클래스에서는 정규화된 이름을 쓰는 걸 피하고자 그 인터페이스를 구현하곤 한다.

상수 인터페이스 안티패턴 - 사용금지

```
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
}
```

- 인터페이스를 잘못 사용한 예이다.
- 상수 인터페이스를 구현하는 것은 이 내부 구현을 클래스의 API로 노출하는 행위이다.
- 클래스가 어떤 상수 인터페이스를 사용하든 사용자에게는 아무런 의미가 없으며, 사용자에게 혼란을 줄 수도 있으며, 클라이언트 코드가 내부 구현에 해당하는 상수들에 종속되게 하는 결과를 가져올 수 있다.
- final이 아닌 클래스가 상수 인터페이스를 구현한다면 모든 하위 클래스의 이름 공간이 그 인터페이스가 정의한 상수들로 오염된다.

상수 유틸리티 클래스

```
public class PhysicalConstants {
    private PhysicalConstants() { }  // private 생성자로 인스턴스

    // 아보가드로 수 (1/몰)
    public static final double AVOGADROS_NUMBER = 6.022_140_8
    // 볼츠만 상수 (J/K)
    public static final double BOLTZMANN_CONST  = 1.380_648_5
    // 전자 질량 (kg)
    public static final double ELECTRON_MASS    = 9.109_383_5
}
```

정적 임포트를 사용해 상수 이름만으로 사용

```
import static effectivejava.chapter4.item22.constantutilitycl

public class Test {
```

```
    double atoms(double mols){
        return AVOGADROS_NUMBER * mols;
    }
    ...
    // PhysicalConstants 를 빈번히 사용한다면 정적 임포트가 값어치를 ㅎ
    }
```

아이템 23. 태그 달린 클래스보다는 클래스 계층 구조를 활용하라.

태그 달린 클래스 Figure

```
public 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 length * width;
            case CIRCLE:
                return Math.PI * (radius * radius);
            default:
                throw new AssertionError(shape);
        }
    }
}
```

해당 클래스에는 단점이 많다.

- 열거 타입 선언, 태그 필드, switch문 등 불필요한 코드가 많다.
- 여러 구현이 한 클래스에 혼합되어 가독성도 나쁘다.
- 다른 의미를 위한 코드도 언제나 함께 하기 때문에 메모리도 많이 사용한다.
- 필드를 final로 선언하기 위해 해당 의미에 쓰이지 않는 필드들들까지 생성자에서 초기화 해야한다.
- 인스턴스의 타입 만으로는 현재 나타내는 의미를 알 수 없다.

클래스 계층 구조

- 다음은 위의 단점을 해결하기 위해 위의 태그 달린 클래스를 클래스 계층 구조로 바꾼 것이다.

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

class Circle extends Figure {
    final double radius;

    public Circle(double radius) {
        this.radius = radius;
```

```
    }

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

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

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

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

태그 달린 클래스를 클래스 계층구조로 바꾸기

- 계층구조의 루트가 될 추상 클래스를 정의하고, 태그 값에 따라 동작이 달라지는 메서드를 루트 클래스의 추상 메서드로 선언한다.(area)
- 태그 값에 상관없이 동작이 일정한 메서드들을 루트 클래스에 일반 메서드로 추가하고, 공통으로 사용하는 데이터 필드들도 전부 루트 클래스로 올린다.(현재 코드에선 존재 X)
- 루트 클래스를 확장한 구체 클래스를 의미별로 하나씩 정의한다.

아이템 24.멤버 클래스는 되도록 static으로 만들라.

중첩 클래스

- **다른 클래스 안에 정의된 클래스를 말한다.**
  - 자신을 감싼 바깥 클래스에서만 사용되어야 하며, 중첩 클래스를 사용함으로써 불필요한 노출을 줄여 캡슐화를 할 수 잇고 유지보수하기 좋은 코드를 작성할 수 있다.
- **중첩 클래스의 종류**
  - 정적 멤버 클래스
  - 비정적 멤버 클래스
  - 익명 클래스
  - 지역 클래스

- **정적 멤버 클래스**
  - static 키워드의 여부로 정적 멤버 클래스와 비정적 멤버 클래스를 구분한다.
  - 정적 멤버 클래스는 외부 클래스의 private 멤버에도 접근할 수 있다는 점을 제외하고는 일반 클래스와 똑같다. ex) builder 패턴

- **비정적 멤버 클래스**
  - static 키워드가 없는 내부 클래스이다.
  - 비정적 멤버 클래스의 인스턴스는 바깥 클래스의 인스턴스와 암묵적으로 연결된다.
  - 그러므로 비정적 멤버 클래스는 this 키워드로 바깥 인스턴스의 메서드를 호출하거나 참조를 가져올 수 있다.
- **비정적 멤버 클래스의 활용**
  - 어댑터를 정의할 때 자주 사용된다.
  - 즉, 어떤 클래스의 인스턴스를 감싸 마치 다른 클래스의 인스턴스 처럼 보이게 하는 뷰로 사용하는 것이다..
  - 예를 들어 HashMap의 keySet()을 사용하면 Map의 key의 해당하는 값을 Set으로 반환할 때 어댑터 패턴을 이용해서 Map을 Set으로 제공한다.
  - 비슷하게, Set과 List같은 다른 컬렉션 인터페이스 구현들도 자신의 반복자를 구현할 때 비정적 멤버 클래스를 주로 사용한다.

```
public class MySet<E> extends AbstractSet {
    @Override
```

```
    public Iterator iterator() {
        return new MyIterator();
    }

    private class MyIterator implements Iterator<E>{
        ...
    }
}
```

- **멤버 클래스에서 바깥 인스턴스에 접근할 일이 없다면 무조건 static을 붙여서 정적 멤버 클래스로 만들자.**
  - 비정적 멤버 클래스를 사용하면 바깥 인스턴스로의 숨은 외부 참조를 갖게된다. 이 참조를 저장하려면 시간과 공간이 소비된다.
  - 더 심각한 문제는 가비지 컬렉션이 바깥 클래스의 인스턴스를 수거하지 못하도록 하는 메모리 누수가 생길 수도 있다.

익명 클래스

- 이름이 없다.
- 바깥 클래스의 멤버가 아니다.
- 코드의 어디서든 만들 수 있다.

지역 클래스

- static 멤버는 갖지 못하며, 클래스 내부에서 필요한 기능을 정의할 때 사용한다.

아이템 25. 톱 레벨 클래스는 한 파일에 하나만 담으라.

- 톱레벨 클래스를 여러개 선언하면 이득은 없고 심각한 위험만 발생할 수 있다.
- 한 클래스를 여러개로 정의할 수 있으며, 그 중 어느 것을 사용할지는 어느 소스파일을 먼저 컴파일하냐에 따라 달라지기 때문이다.

```
public class Main {
    public static void main(String[] args) {
```

```
        System.out.println(Utensil.NAME + Dessert.NAME);
    }
}
```

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

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

- 컴파일러에 어느 소스파일을 먼저 건네느냐에 따라서 정상 작동하거나 오류가 발생할 수도 있다.

톱레벨 클래스 중복정의 해결방법

- 단순히 톱레벨 클래스들을 서로 다른 소스 파일로 분리하면 된다.
- 만약 굳이 서로 다른 여러 톱레벨 클래스를 한 파일에 담고 싶다면 정적 멤버 클래스를 사용하는 방법을 고민해 보아야할 것이다.