2주차 - 3장 모든 객체의 공통 메서드, 4장 클래스와 인터페이스 - 미하, 환석

[3장 모든 객체의 공통 메서드]

object에서 final이 아닌 메서드 (equals, hashCode, toString, clone, finalize)는 모두 재정의overriding을 염두에 두고 설계되었다.

일반 규약에 맞게 재정의 해야 규약을 준수한다고 가정하는 클래스들(HashMap, HashSet등)이 오작동을 하지 않는다.

이번 장에서는 Object메서드들을 언제 어떻게 재정의해야 하는지를 다룬다.

Comparable.compareTo의 Object의 메서드는 아니지만 성격이 비슷하여 함께 다룬다.

item10 - equals는 일반 규약을 지켜 재정의하라

equals 메서드는 재정의하기 쉬워 보이지만 함정이 도사리고 있다.(젤 쉬운건 재정의 안하기…)

Default는 그 클래스의 인스턴스는 오직 자기 자신과만 같게한다.

So, 다음 상황중 하나에 해당하면 재정의 안하도록..!

- 각 인스턴스가 본질적으로 고유하다.
- 인스턴스의 '논리적 동치성(logical equality)'을 검사할 일이 없다.
- 상위 클래스에서 재정의한 equals가 하위 클래스에도 딱 들어맞는다.
- 클래스가 private이거나 package-private이고 equals 메서드를 호출할 일이 없다.

equals가 실수로라도 호출되는걸 막고싶다면

```
@Override public boolean equals(Object o){
    throw new AssertionError(); //!
}
```

그렇다면 재정의 해야 할때?

- 객체 식별성이 아니라 논리적 동치성을 확인해야 할 때

ex) 값 클래스들(Integer, String…) 값이 같은지를 알고싶어 할테니까 !

but, 값 클래스여도 값이 같은 인스턴스가 둘 이상 만들어지지 않음을 보장하는 인스턴스 통제클래스라면 재정의 안해도 OK.

ex)Enum 논리적 동치성 = 객체 식별성

Object 명세에 적힌 규약

equals 메서드는 동치관계를 구현하며 다음을 만족한다.

- 반사성 : null이 아닌 모든 참조 값 x에 대해 x.equals(x) 는 true다.
- 대칭성 : null이 아닌 모든 참조 값 x, y에 대해 x.equals(y)가 true면 y.equals(x)도 true다.
- 추이성 : null이 아닌 모든 참조 값 x, y, z에 대해 x.equals(y)가 true 이고 y.equals(z) 도 true면 x.equals(z)도 true다.
- 일관성 : null이 아닌 모든 참조 값 x, y에 대해 x.equals(y)를 반복해서 호출하면 항상 같은 결과를 반환한다.
- null 아님 : null이 아닌 모든 참조 값 x에 대해 x.equals(null)은 false다.

동치관계 : 집합을 서로 같은 원소들로 이루어진 부분집합으로 나누는 연산이다.

equals메소드가 쓸모있으려면 모든 원소가 같은 동치류에 속한 어떤 원소와도 서로 교환할 수 있어야 한다.

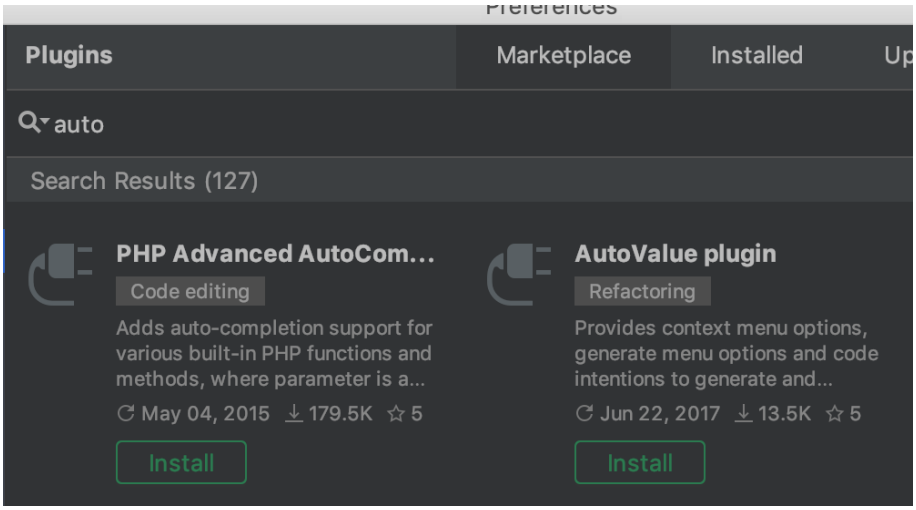
양질의 equals 메서드 구현 방법 정리

1. == 연산자를 사용해 입력이 자기 자신의 참조인지 확인한다.
2. instanceof 연산자로 입력이 올바른 타입인지 확인한다.
3. 입력을 올바른 타입으로 형변환한다.
4. 입력 객체와 자기 자신의 대응되는 '핵심'필드들이 모두 일치하는지 하나씩 검사한다.

덧, 어떤 필드를 먼저 비교하느냐가 equals의 성능을 좌우하기도 한다.

equals를 다 구현했다면 세 가지만 자문해보자. 대칭적인가? 추이성이 있는가? 일관적인가?

자문에서 끝내지 말고 단위 테스트를 작성해 돌려보자. (AutoValue 프레임워크 이용 추천)



마지막 주의사항

- equals를 재정의할 땐 hashCode도 반드시 재정의하자.
- 너무 복잡하게 해결하려 들지 말자.
- Object 외의 타입을 매개변수로 받는 equals 메서드는 선언하지 말자.

```
//   -    Object   !
//   Object.equals     .
//  equals     ,    equals    .
public boolean equals(MyClass o){
  ...
}
```

item11 - equals를 재정의하려거든 hashCode도 재정의하라

equals를 재정의한 클래스 모두에서 **hashCode**도 재정의해야 한다.

hashCode를 잘못 재정의 했을때 크게 문제가 되는 조항은 아래 두번째 조항. **즉, 논리적으로 같은 객체는 같은 해시코드를 반환해야 한다.**

Object 명세에서 발췌한 규약

- 애플리케이션이 실행되는 동안 한 객체에 대헌 hashCode 메서드는 몇번을 호출해도 일관되게 항상 같은 값을 반환해야 한다.
- equals(Object)가 두 객체를 같다고 판단했다면 두 객체의 hashCode는 똑같은 값을 반환해야 한다.
- 두 객체가 다르다고 판단했더라도 서로 다른 hashCode를 반환할 필요는 없다. but, 좋은 해시 함수는 서로 다른 인스턴스에 다른 해시코드를 반환한다. 그게 성능도 더 좋다.

좋은 hashCode를 작성하는 간단한 요령

1. int 변수 result를 선언한 후 값 c로 초기화한다.
2. 해당 객체의 나머지 핵심 필드 f 각각에 대해 다음 작업을 수행한다.

   a. 해당 필드의 해시코드 c를 계산한다.

   b. 위에서 계산한 해시코드 c로 result를 갱신한다. result = 31* result + c;
3. result를 반환한다.

다 구현했다면 이 메서드가 동치인 인스턴스에 대해 똑같은 해시 코드를 반환할지 자문하고 테스트를 해보자.(AutoValue 굳)

파생필드는 해시코드 계산에서 제외OK. equals 비교에 사용되지 않은 필드는 **반드시** 제외해야 한다.

클래스가 불변이고 해시코드를 계산하는 비용이 크다면 **캐싱**하는 방식을 고려해야 한다.

해시의 키로 사용되지 않는 경우라면 hashCode가 처음 불릴 때 계산하는 **지연 초기화** 전략도 있다.(스레드 안전성 고려)

성능을 높인답시고 해시코드를 계산할 때 **핵심 필드를 생략해서는 안된다**.

hashCode가 반환하는 값의 생성 규칙을 API사용자에게 자세히 공표하지 말자. 그래야 클라이언트가 이 값에 의지하지 않게 되고 추후에 계산 방식을 바꿀 수도 있다.

item12 - toString을 항상 재정의하라

기본 toString메서드는 단순히 **클래스_이름@16진수로_표현한_해시코드** 를 반환한다.

간결하면서 사람이 읽기 쉬운 형태의 유익한 정보 를 반환한다.

가 toString의 일반 규약이며 **모든 하위 클래스에서 이 메서드를 재정의하라**고 한다.

why? toString을 잘 구현한 클래스는 사용하기에 훨씬 즐겁고, 그 클래스를 사용한 시스템은 디버깅하기 쉬우니까 !

toString메서드는 객체를 println, printf, 문자열 연결 연산자(+), assert 구문에 넘길 때, 혹은 디버거가 객체를 출력할 때 자동으로 불린다. (어딘가에서 쓰인다!)

```
// PhoneNumber toString     .
System.out.println(phoneNumber + "   .");
```

실전에서 toString은 그 객체가 가진 주요 정보 모두를 반환하는게 좋다.

toString의 반환값의 포맷을 **문서화**할지 정해야 한다.

포맷을 명시하기로 했다면, 명시한 포맷에 맞는 문자열과 객체를 상호 전환할 수 있는 **정적 팩터리나 생성자**를 함께 제공해주면 좋다.(ex 값 클래스 BigInteger, BigDecimal와 대부분의 기본 타입 클래스)

장/단점은 있다. 그러나 포맷을 명시하든 아니든 **의도는 명확히 밝혀야한다.**

포맷 명시 여부와 상관없이 toString이 반환한 값에 포함된 정보를 얻어올 수 있는 API를 제공하자.

접근자를 제공하지 않으면 변경될 수 있다고 문서화했더라도 그 포맷이 사실상 준-표준 API나 다름없어진다.

마지막으로…

정적 유틸리티 클래스(아이템4)는 toString을 제공할 이유가 없다. 대부분의 열거타입(아이템34)도 이미 완벽한 toString이 제공되니 재정의 하지 않아도 된다.

하지만 하위 클래스들이 공유해야 할 문자열 표현이있는 추상클래스라면 재정의해줘야 한다. (컬렉션 구현체는 추상 컬렉션 클래스들의 toString메서드를 상속해 쓴다.)

AutoValue 프레임 워크는 toString도 생성해준다. 아무것도 알려주지 않는 Object toString보다 훨씬 유용하다.

item13 - clone 재정의는 주의해서 진행하라

메서드 하나 없는 Clonable 인터페이스는 무얼하나?

놀랍게도 Object의 protected 메서드인 clone의 동작 방식을 결정한다.

clone 메서드의 일반 규약 (허술함)

객체의 복사본을 생성해 반환한다.

Clone 메서드는 사실상 생성자와 같은 효과를 낸다. 즉, clone은 원본 객체에 아무런 해를 끼치지 않는 동시에 복제된 객체의 불변식을 보장해야 한다.

요약하자면..

Cloneable을 구현하는 모든 클래스는 clone을 재정의하고 동기화해야한다. -> 객체의 내부 '**깊은 구조**'에 숨어 있는 모든 가변 객체를 복사하고, 복제본이 가진 객체 참조 모두가 복사된 객체들을 가리키게 한다.

how? 주로 clone을 재귀적으로 호출해 구현하지만 항상 최선은 아니다.

기본 원칙은 **'복제 기능은 생성자와 팩터리를 이용하는게 최고'** 라는 것이다.

(예외, 기본 배열만은 clone메서드 방식이 가장 깔끔한, 규칙의 합당한 예외)

item14 - Comparable을 구현할지 고려하라

Comparable 인터페이스의 유일무이한 메서드인 compareTo를 알아보자.

Object equals와 다른점 ?

compareTo는 단순 동치성 비교에 더해 **순서**까지 비교할 수 있으며, 제네릭하다. 그 클래스의 인스턴스들에는 자연적인 순서가 있음을 뜻한다.

```
Arrays.sort(a); // comparable
```

순서가 명확한 값 클래스를 작성한다면 반드시 Comparable 인터페이스를 구현하자.

compareTo 메서드 작성 요령은 equals와 비슷하다.

Comparable은 타입을 인수로 받는 제네릭 인터페이스이므로 compareTo 메서드의 인수 타입은 컴파일타임에 정해진다.(확인하거나 형변환 필요X)

compareTo 메서드는 필드가 동치인지 비교하는게 아니라 *그 순서를 비교한다.*

compareTo 메서드에서 정수 기본 타입 필드를 비교시 -> 자바 7부터 박싱된 기본 타입 클래스들에 새로 추가된 정적 메서드인 compare를 이용하면 된다.

관계 연산자 < >를 사용하는 이전 방식은 거추장스럽고 오류를 유발한다.

핵심 필드가 여러개라면 어느것을 먼저 비교하느냐가 중요해진다. 가장 핵심적인 필드부터 비교하자.

Comparator는 수많은 보조 생성 메서드들로 중무장하고 있다. 자바의 숫자용 기본 타입을 모두 커버한다.

[4장 클래스와 인터페이스]

추상화의 기본 단위인 **클래스와 인터페이스**는 자바 언어의 심장!! 그 설계에 사용하는 강력한 요소가 많이 있다.

이런 요소를 적절히 활용하여 클래스와 인터페이스를 **쓰기 편하고, 견고하며, 유연하게 만드는 방법**을 안내한다.

item15 - 클래스와 멤버의 접근 권한을 최소화하라

잘 설계된 컴포넌트?

클래스 내부 데이터와 내부 구현정보를 외부 컴포넌트로부터 **얼마나 잘 숨겼느냐**다. (정보은닉, 캡슐화) -> **구현과 API를 깔끔히 분리함**

정보 은닉의 장점

- 시스템 개발 속도를 높인다.
- 시스템 관리 비용을 낮춘다.
- 성능 최적화에 도움을 준다.
- 소프트웨어 재사용성을 높인다.
- 큰 시스템을 제작하는 난이도를 낮춰준다.

정보 은닉을 위한 다양한 장치? 접근 제한자 !

접근 제한자를 제대로 활용하는 것이 정보 은닉의 핵심이다.

모든 클래스와 멤버의 접근성을 가능한 한 좁혀야 한다.

소프트웨어가 올바로 동작하는 한 항상 가장 낮은 접근 수준을 부여해야 한다.

접근 수준 네 가지

- private
- package-private
- protected
- public

자바 9 에서부터 모듈 시스템이라는 개념이 도입되면서 추가된 암묵적 접근 수준 두가지

패키지중 공개(export)할 것들을 선언하게 되는데 공개하지 않으면 public 멤버라도 외부에서 접근X.. **효과가 모듈 내부로 한정되는 변종**

모듈경로가 아닌 애플리케이션의 클래스패스에 두면 그 모듈 안의 모든 패키지는 모듈이 없는 것처럼 행동한다. (모두 밖에서 접근가능)

ex)JDK 자체가 적극 활용한 사례.

클래스의 공개 API를 세심히 설계한 후, 그외의 모든 멤버는 private으로.

테스트의 목적으로 적당한 수준으로 넓혀도 되지만 공개 API로 만들어서는 안 된다.

public 클래스는 상수용 public static final 필드 외에는 어떠한 public 필드도 가져서는 안된다.

public static final 필드가 참조하는 객체가 불변인지 확인하라.

item16 - public 클래스에서는 public 필드가 아닌 접근자 메서드를 사용하라

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

API를 수정하지 않고는 내부 표현을 바꿀 수 없고, 불변식도 보장할 수 없고, 외부에서 필드에 접근할 때 부수 작업을 수행할 수도 없다…

public 클래스의 필드가 불변인경우, 불변식만 보장할뿐 나머지 단점들은 갖고있게된다.

```
//      .
// public      !
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 setX( double y ) { this.y = y; }
}
```

패키지 바깥에서 접근할 수 있는 클래스라면 접근자를 제공함으로써 클래스 내부 표현 방식을 언제든 바꿀 수 있는 유연성을 얻을 수 있다.

package-private 클래스 혹은 private중첩 클래스라면 데이터 필드를 노출한다 해도 하등의 문제가 없다. 추상 개념만 올바르게 표현해준다면.

why? 패키지 바깥 코드는 전혀 손대지 않고도 데이터 표현방식을 바꿀 수 있어서.

⚠️ 규칙을 어긴 사례 : java.awt.package 의 Point 와 Dimension. 흉내X 타산지석으로 삼길..

item17 - 변경 가능성을 최소화하라

불변 클래스: 인스턴스의 내부 값을 수정할 수 없는 클래스. 객체가 파괴되는 순간까지 절대 달라지지 않음.

ex) String, 기본 타입의 박싱된 클래스들, BigInteger, BigDecimal .

불변 클래스는 가변 클래스보다 설계하고 구현하고 사용하기 쉬우며, 오류가 생길 여지도 적고 훨씬 안전하다.

클래스를 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 컴포넌트에 접근할 수 없도록 한다.

클래스를 확장 할 수 없도록 하는 다른 방법 : 모든 생성자를 private 혹은 package-private으로 만들고public정적 팩터리를 제공하는 방법

```
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) {  //  add   plus
        //        Complecx
        return new Complex(re + c.re, im + c.im);
    }

    //  17-2  (private    .) (110-111)
    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;

        // ==  compare   63 .
        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)";
    }
}
```

피연산자에 함수를 적용해 결과를 반환하지만 피연산자 자체는 그대로인 프로그래밍 = 함수형 프로그래밍

메서드에서 피연산자인 자신을 수정해 상태가 변하는 것 = 절차적, 명령형 프로그래밍

함수형 프로그래밍을 하면 코드에서 불변이 되는 영역의 비율이 증가하는 장점이 있고, 불변 객체는 단순하다.

가변 객체는 임의의 복잡한 상태에 놓일 수 있다.

불변 객체는 근본적으로 스레드 안전하여 따로 동기화할 필요가 없다.

여러 스레드가 동시에 사용해도 절대 훼손되지 않는다.(클래스를 스레드 안전하게 만드는 가장 쉬운방법!) So, 불변객체는 안심하고 공유할 수 있다:)

적정 팩터리(야이템1)를 제공할 수 있는데, 여러 클라이언트가 인스턴스를 공유하여 메모리 사용량과 가비지 컬렉션 비용이 줄어든다.

불변 객체를 자유롭게 공유할 수 있다는점은 방어적복사(아이템50)도 필요없다는 결론으로 이어진다.

불변 객체끼리는 내부 데이터를 공유할 수 있다. 불변 객체들을 구성요소로 사용하면 불변식을 유지하기 수월하다는 이점 불변 객체는 그 자체로 실패 원자성을 제공한다.

단점?

값이 다르면 반드시 독립된 객체로 만들어야 한다. 값의 가짓수가 많다면....

정리

- getter가 있다고 해서 무조건 setter를 만들지 말자. 클래스는 꼭 필요한 경우가 아니라면 불변이어야 한다.
- 불변으로 만들 수 없는 클래스라도 변경할 수 있는 부분을 최소한으로 줄이자.
- 다른 합당한 이유가 없다면 모든 필드는 private final이어야 한다.
- 생성자는 불변식 설정이 모두 완료된, 초기화가 완벽히 끝난 상태의 객체를 생성해야 한다.

확실한 이유가 없다면 생성자와 정적 팩터리 외에는 그 어떤 초기화 메서드도 public으로 제공해서는 안 된다.

객체를 재활용할 목적으로 상태를 다시 초기화하는 메서드도 안 된다. -> 복잡성만 커지고 성능 이점은 거의 없다.

Item 18 : 상속보다는 컴포지션을 사용하라

상속은 보통 **코드를 재사용하는 강력한 수단**이지만, 최선은 아니다. 오류를 만들어 내기 쉽다.

- **매서드 호출과 달리 상속은 캡슐화를 깨뜨린다.**

  → 상위 클래스가 어떻게 구현되느냐에 따라 하위클래스 동작이 달라질 수 있다.

상속의 잘못된 예

```
public class InstrumentedHashSet<E> extends HashSet<E> {
    //
    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;
    }

    public static void main(String[] args) {
        InstrumentedHashSet<String> s = new InstrumentedHashSet<>();
        s.addAll(List.of("Snap", "Crackle", "Pop"));
        System.out.println(s.getAddCount());
    }

}
```

반환 예상 값 : 3

실제 출력 값 : 6

→ InstrumentedHashSet에서 구현된 addAll()이 HashSet의 addAll() 메서드를 호출하기 때문이다.

addAll method in HashSet

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

여기서 add() 메서드가 호출될때 위에서 재정의한 메서드가 호출된다.

- ***안전한 방법은?***

  1. addAll 메서드를 재정의 하지 않는다 → HashSet의 addAll과 add가 어떻게 동작하는지 안다는 가정.

  2. addAll 메서드를 다른방식으로 재정의 → HashSet의 addAll을 호출하지 않는 방법(첫번째 방법보다는 나음)

  3. 매서드를 재정의하기 보다는 새로운 메서드를 추가해서 사용 → 문제가 될 가능성은 여전히 존재 (이름이 같은경우, return type 같은 경우)₩

  4. 확장 대신 새로운 클래스를 생성, private 필드로 기존 클래스의 인스턴스를 참조 (기존 클래스가 새로운 클래스의 구성요소로 쓰인다는 의미로 이 같은 설계를 composition이라고 함.)

```
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;
    }

    public static void main(String[] args) {
        InstrumentedSet<String> s = new InstrumentedSet<>(new HashSet<>());
        s.addAll(List.of("Snap", "Crackle", "Pop"));
        System.out.println(s.getAddCount());
    }
}


public class ForwardingSet<E> implements Set<E> {
    private final Set<E> s;
    public ForwardingSet(Set<E> s) { this.s = s; }

    public void clear()               { s.clear();            }
    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 addAll(Collection<? extends E> c)
    { return s.addAll(c);      }
        //...


}
```

- 다른 Set 인스턴스를 감싸고 있다는 뜻에서 InstrumentedSet 같은 클래스를 **wrapper 클래스**라고 하고 다른 Set에 측정, 기록 하는 기능을 덧씌운다는 뜻으로 **Decorator pattern**이라고 한다.

- 상속은 **is-a 관계**를 만족하는 두개의 클래스에서만 구현해야 한다. (ex. A와 B 클래스 : B는 A인가? 를 만족)
- Java에서 이 원칙을 위반한 클래스는 **Stack 과 Vector**, **Properties 와 HashTable** 이 있다고 한다.

Item 19 - 상속을 고려해 설계하고 문서화 하라

- 상속용 클래스는 재정의할 수 있는 매서드들을 내부적으로 어떻게 이용하는지 문서로 남겨야 한다.

```
/**
* {@inheritDoc}
*
* @implSpec
* 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
* {@code UnsupportedOperationException} if the iterator returned by this
* collection's iterator method does not implement the {@code remove}
* method and this collection contains the specified object.
*
* @throws UnsupportedOperationException {@inheritDoc}
* @throws ClassCastException {@inheritDoc}
* @throws NullPointerException {@inheritDoc}
*/
```

iterator 매서드를 재정의하면 remove 메서드의 동작에 영향을 준다는 설명하는 주석

- 상속용 클래스를 설계 할때는 직접 하위 클래스를 만들어서 검증이 필요하다.
- 상속용 클래스의 생성자는 직,간접적으로든 재정의 가능 메서드를 호출해서는 안 된다.

상속용 클래스

```
// Class whose constructor invokes an overridable method. NEVER DO THIS! (Page 95)
public class Super {
    // Broken - constructor invokes an overridable method
    public Super() {
        overrideMe();
    }

    public void overrideMe() {
    }
}
```

하위 클래스

```
// Demonstration of what can go wrong when you override a method  called from constructor (Page 96)
public final class Sub extends Super {
    // Blank final, set by constructor
    private final Instant instant;

    Sub() {
        instant = Instant.now();
    }

    // Overriding method invoked by superclass constructor
    @Override public void overrideMe() {
        System.out.println(instant);
    }

    public static void main(String[] args) {
        Sub sub = new Sub();
        sub.overrideMe();
    }
}
```

- 클래스를 확장해야할 이유가 명확하지 않다면 상속을 금지해라.
- 금지하는 방법으로는 final 선언, 생성자 모두를 외부에서 접근 할 수 없도록 변경.

item 20 : 추상 클래스보다는 인터페이스를 우선하라

- 인터페이스와 추상클래스의 가장 큰 차이는 ***추상클래스를 구현한 하위클래스는 반드시 추상클래스의 하위 타입***이 되어야한다는 것
- 인터페이스는 **믹스인(mixin) 정의**에 적합하다.
    - mixin : 클래스가 구현할 수 있는 타입, 원래의 primary type 외에 특정 부가적인 행위를 제공하고 선언하는 효과를 준다(?)

        ex) Comparable 인터페이스는 자신을 구현한 클래스의 인스턴스끼리의 순서를 정할 수 있다고 선언하는 mixin interface
- 인터페이스를 이용해서 계층구조가 없는 타입 프레임워크를 만들 수 있다.
    - 현실적으로 계층을 엄격히 구분하기 어려울때 사용하면 좋음 (ex singer, songwriter)

혼합구조

```
public interface Singer {
    AudioClip sing(Song s);
}


public interface Songwriter {
    Song compose(boolean hit);
}


public interface SingerSongwriter extends Songwriter, Singer{
   AudioClip strum();
   void actSensitive();
}
```

- 인터페이스와 추상 골격 구현(skeletal implementation) 클래스를 함께 제공하여 2개의 장점을 모두 취하는 방법도 있다.
  - ?

참고자료

- 추상 클래스 대신 인터페이스를 사용하자

Item 21 : 인터페이스는 구현하는 쪽을 생각해 설계하라

- 자바8이전에는 인터페이스에 메서드를 추가하면 보통 컴파일 오류가 발생했다.
    - 구현 클래스들에서 구현을 하지 않았기 때문
- 자바8부터는 인터페이스에 메서드를 추가할 수 있지만 *<생각할 수 있는 모든 상황에서 불변식을 해치지 않는 디폴트 메서드를 작성하는 방법>*은 어렵다.

ex) apache commons의 SynchronizedCollection

- 기존 인터페이스에 디폴트 메서드로 새 메서드를 추가하는 일은 꼭 필요한 경우가 아니라면 피하도록 하자.

Item 22 : 인터페이스는 정의하는 용도로만 사용하라

- 인터페이스는 자신을 구현한 클래스의 인스턴스를 참조할 수 있는 타입 역할
- 클래스가 어떤 인터페이스를 구현한다는 것은 자신의 인스턴스로 무엇을 할 수 있는지를 말해주는 것

ex) 잘못 사용하는 예 (상수 인터페이스 - public static final 필드들만 있는 인터페이스)

상수 인터페이스

```
// Constant interface antipattern - do not use!
public interface PhysicalConstants {
    // Avogadro's number (1/mol)
    static final double AVOGADROS_NUMBER   = 6.022_140_857e23;

    // Boltzmann constant (J/K)
    static final double BOLTZMANN_CONSTANT = 1.380_648_52e-23;

    // Mass of the electron (kg)
    static final double ELECTRON_MASS      = 9.109_383_56e-31;
}
```

클래스에서 사용하는 상수는 외부 인터페이스가 아니라 내부 구현에 해당. 아무런 의미가 없고 혼란만 준다.

Integer.MAX_VALUE, enum type, **정적 유틸리티 클래스** 등에서 사용하는게 훨씬 좋다. 좀 더 명확해진다.

유틸리티 클래스

```
// Constant utility class (Page 108)
public class PhysicalConstants {
  private PhysicalConstants() { }  // Prevents instantiation

  // Avogadro's number (1/mol)
  public static final double AVOGADROS_NUMBER = 6.022_140_857e23;

  // Boltzmann constant (J/K)
  public static final double BOLTZMANN_CONST  = 1.380_648_52e-23;

  // Mass of the electron (kg)
  public static final double ELECTRON_MASS    = 9.109_383_56e-31;
}
```

Item 23 : 태그 달린 클래스보다는 클래스 계층구조를 활용하라

- 두가지 이상의 의미를 표현할 수 클래스의 경우 클래스 계층 구조를 활용해라.

ex) tagged class

tagged class

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

    // Tag field - the shape of this figure
    final Shape shape;

    // These fields are used only if shape is RECTANGLE
    double length;
    double width;

    // This field is used only if shape is CIRCLE
    double radius;

    // Constructor for circle
    Figure(double radius) {
        shape = Shape.CIRCLE;
        this.radius = radius;
    }

    // Constructor for rectangle
    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);
        }
    }
}
```

- 단점 : 여러 구현이 한 클래스에 혼합되어 있어 가독성도 나쁘고 사용하지 않는 필드들도 초기화 해야한다. 즉, 장황하고, 오류를 내기 쉬우며 비효율 적이다.

계층구조

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

class Circle extends Figure {
    final double radius;

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

    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;
    }

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

- 간결, 명확해짐. 확장성 있는 형태로 변경
- 태그 달린 클래스의 경우 클래스 계층을 이용해서 리팩토링하는 방법을 고민해라

Item 24 : 맴버 클래스는 되도록 static으로 만들라.

- nested class란 클래스 안에 정의된 클래스를 말한다. nested class는 자신을 감싼 바깥 클래스에서만 쓰여야 한다.
- 종류는 정적 멤버 클래스, 맴버 클래스, 익명 클래스, 지역 클래스 가 있다.
- 맴버 클래스의 인스턴스 각각이 바깥 클래스의 인스턴스에 접근할 일이 없다면 무조건 static을 붙여서 정적 멤버 클래스로 만든다. 아니면 비정적으로 만든다.
- 비정적의 경우 바깥 인스턴스에 대한 숨은 참조가 가능한데 이는 시간과 공간에 대한 비용이 들어가고 이 참조 때문에 GC가 인스턴스를 제때 수거하지 못하는 메모리 누수가 생길 수도 있다고 한다.

Item 25 : 톱 레벨 클래스는 한 파일에 하나만 담아라

- 소스 파일 하나에 여러개의 톱레벨 클래스가 선언되더라도 자바 컴파일러는 문제를 삼지 않는다.
- 하지만 컴파일러가 한 클래스에 대한 정의를 여러개 만들 수 있고 바이너리 파일이나 프로그램의 동작이 순서에 따라 달리질 수 있기 때문에 **한 파일에는 하나의 톱 레벨 클래스만 담자.**

ex) 프로그램 동작에 문제가 되는 코드

main.java

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

Utensil.java

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

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

Dessert.java

```
// Two classes defined in one file. Don't ever do this! (Page 115)
class Utensil {
    static final String NAME = "pot";
}

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

컴파일 방식에 따른

- javac Main.java Dessert.java ： ?
- javac Main.java or javac Main.java Utensil.java ： ?
- javac Dessert.java Main.java： ?