EJ

이펙티브 자바 CP.2

작성 일시	@2023년 1월 7일 오후 8:01
최종 편집 일시	@2023년 1월 12일 오후 11:48
유형	이펙티브 자바
작성자	
참석자	

2️⃣ 모든 객체의 공통 메소드

▼ 10. equals 는 일반 규약을 지켜 재정의 해라

- equals 를 재 정의하지 않는 경우
  - 각 인스턴스가 본질 적으로 고유하다.
    - 값을 표현하는 (String, Integer 등) 게 아니라, 동작하는 개체를 표현하는 클래스(Thread)
  - 인스턴스의 논리적 동치성(동등성 = 논리적으로 같다.)을 검사할 일이 없다.
    - 인스턴스간에 서로 값 비교를 할 필요가 없다면 하지 않아도 된다.
  - 상위 클래스에서 재정의한 equals가 하위 클래스에도 딱 들어 맞는다.
    - Set 구현체는 AbstractSet, List 구현체는 AbstractList, Map 구현체는 AbstractMap 으로 equals 메소드를 상속받아 그대로 쓴다.
  - 클래스가 private 이거나 package-private이고 equals 메소드를 호출할 일이 없다.
    - 혹시라도 equals 가 실수로라도 호출 되길 막기 위한다면

```
@Override
public boolean equals(Obejct o) {
    throw new AseertionError(): // 호출 금지
}
```

- equals 를 재 정의 해야 할 경우
  - 객체가 동일(객체 식별성: 두 객체가 물리적으로 같은가) 한지가 아니라 동등(논리적 동치성)을 확인해야 하는데, 상위 클래스의 equals 가 논리적 동치성을 비교하도록 재정의도지 않은 경우
    - equals 가 논리적 동치성을 확인하도록 정의 해두면, 그 인스턴스는 값을 비교하길 원하는 기대에 부흥은 물론, Map 의 key 와 Set의 원소로 사용할 수 있게 된다.
    - 값 클래스라도, 값이 같은 인스턴스가 두 개 이상 만들어지지 않음을 보장하면, equals 를 재정의하지 않아도 된다. (Enum)

- equals 메소드를 재정의 할 때의 규약 (지키지 않으면, 이상하게 동작하고 에러를 찾기도 어려워 진다.)
  - **반사성** - null 이 아닌 모든 참조 값 x에 대해 x.equals(x) 는 true 이다.
    - 객체는 자기 자신과 같아야 한다. - 사실 지키지 않는게 더 어렵다…
  - **대칭성** - null 이 아닌 모든 참조 값 x, y에 대해, x.equals(y)가 true 면 y.equals(x)도 true다
    - 두 객체는 서로에 대한 동치 여부에 똑같아야 한다.
      - 대소문자를 구별하지 않는 문자열을 구현한 다음 클래스를 예로 본다.

```
public final class CaseInsensitiveString {
    private final String s;

    public CaseInsensitiveString(String s) {
        this.s = Objects.requireNonNull(s);
    }

    @Override
    public boolean equals(Obejct o) {
        if (o instanseof CaseInsensitiveString)
            return s.equalsIgnoreCase((CaseInsensitiveString) o).s);
        if (o instanseof String) // 한 방향으로만 동작한다.
            return s.equalsIgnoreCase((String)o);
        return false;
    }
}


CaseInsensitiveString cis = new CaseInsensitiveString("Polish");
String s = "polish";

cis.equals(s) // true CaseInsensitiveString 는 String 의 존재를 알고 코드를 작성 할 수 있다.
s.equals(cis) // false String 은 CaseInsensitiveString 의 존재를 알 수 없다.
// 대칭성을 위반한다.
```

      - equals 규약을 어기면 그 객체를 사용하는 다른 객체들이 어떻게 반응할지 알 수 없다.

```
List<CaseInsensitiveString> list = new ArrayList<>();
list.add(cis);

// 과연 list.contains(s)를 호출하면 어떤 값이 나올까? 에러가 나올 수도...
```

      - 위에 문제를 해결하기 위해선, CaseInsensitiveString의 equals를 String 과도 연동한다는 생각을 버려야한다.

```
@Override
public boolean equals(Object o) {
    return o instanceof CaseInsensitiveString &&
        ((CaseInsensitiveString) o).s.equalsIgnoreCase(s);
}
```

  - **추이성** - null 이 아닌 모든 참조 값 x, y, z에 대해, x.equals(y)가 true이고, y.equals(z)도 true면, x.equals(z) 도 true 이다.
    - 상위 클래스에는 없는 새로운 필드를 하위 클래스에 추가하는 상황을 예로 들자.

```
public class Point {
    private final int x;
    private final int y;
```

```
    public Point(final int x, final int y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public boolean equals(final Object obj) {
        if (!(obj instanceof Point)) {
            return false;
        }

        Point point = (Point) obj;
        return point.x == x && point.y == y;
    }
    ...
}

public class ColorPoint extends Point {
    private final Color color;

    public ColorPoint(final int x, final int y, final Color color) {
        super(x, y);
        this.color = color;
    }
    ...
}
```

```
@Override
public boolean equals(Object o) {
      if(!o instanceof ColorPoint)
          return false;
      return super.equals(o) && ((ColorPoint) o).color == color;
}

Point p = new Point(1, 2);
ColorPoint cp = new ColorPoint(1, 2, Color.RED);

p.equals(cp); // true
cp.equals(p); // false
// 대칭성 위배
```

```
@Override
public boolean equals(Obejct o){
    if(!(o instanceof Point))
        return false;
    if(!(o instanceof ColorPoint))
        return o.equals(this);
    return super.equals(o) && ((ColorPoint) o).color == color;
}

ColorPoint cp1 = new ColorPoint(1, 2, Color.RED);
Point p = new Point(1, 2);
ColorPoint cp2 = new ColorPoint(1, 2, Color.BLUE);

cp1.equals(p); // true;
p.equals(cp2); // true;
cp1.equals(cp2); // false;
// 추이성 위배
// 만약 Point 의 또 다른 하위 클래스가 있다고 하면, 두 번째 if 문에서 무한 재귀에 빠져 StackOverflowError 를 낼 수 있다.
```

- **구체 클래스를 확장해 새로운 값을 추가하면서 equals 규약을 만족시킬 방법은 존재 하지 않는다. - 객체 지향적 추상화의 이점을 포기하면 된다.**
  - 그렇다고 equals 안의 instanceof 검사를 getClass 검사로 바꾸라는 뜻이 아니다.

```
@Override
public boolean equals(Object o){
    if(o == null || o.getClass() != getClass())
        return false;
```

```
    Point p = (Point) o;
        return p.x == x && p.y == y;
}
```

  - 리스코프 치환 원칙 위배
    - equals는 같은 구현 클래스의 객체와 비교할 때만 true 를 내보낸다. 괜찮아 보이지만 활용이 불가능하다.
    - Point의 하위 클래스는 정의상 여전히 Point이므로 어디서든 Point로써 활용될 수 있어야 한다.
      - 위에 equals 로는 하위 클래스를 사용시 false를 반환하여 해당 원칙을 위반한다.
- 쿠체 클래스의 하위 클래스에서 값을 추가할 방법은 없지만, 괜찮은 우회 방법이 있다. 상속 대신 컴포지션(아이템 18)(기존 클래스가 새로운 클래스의 구성요소로 쓰임)을 사용하면 된다.
  - ColorPoint 가 Point 를 상속하지 말고 private Point, private Color로 필드를 생성 해서 사용하는 방법이다.

```
public class ColorPoint{
    private final Point point;
    private final Color color;

    public ColorPoint(int x, int y, Color color) {
        point = new Point(x, y);
        this.color = Objects.requireNonNull(color);
    }

    /* 이 ColorPoint의 Point 뷰를 반환한다. */
    public Point asPoint(){ // view 메서드 패턴
        return point;
    }

    @Override public boolean equals(Object o){
        if(!(o instanceof ColorPoint)){
            return false;
        }
        ColorPoint cp = (ColorPoint) o;
        return cp.point.equals(point) && cp.color.equals(color);
    }
    ...
}
```

  - 만약, 상위 클래스가 추상 클래스라면 equals 규약을 지키면서도 값을 추가 할 수 있다. 상위 클래스의 인스턴스를 직접 만드는 게 불가능하기 때문에, 하위 클래스끼리의 비교가 가능하다.
- **일관성** - null 이 아닌 모든 참조 값 x, y 에 대해, x.equals(y)를 반복해서 호출 하면 모두 true 거나 모두 false 이여야 한다.
  - 가변 객체는 비교 시점에 따라 서로 다를 수도 같을 수도 있다.
  - 불변 객체는 한 번 다르면 끝까지 다르고 같으면 끝까지 같아야한다.
  - 클래스가 불변이든 가변이든 equals 의 판단에 신뢰할 수 없는 자원이 끼어들게 해서는 안 된다.
    - equals 는 항시 메모리에 존재하는 객체만을 사용하는 결정적 계산만 수행한다.
- **null 아님** - null이 아닌 모든 참조 값 x 에 대해, x.equals(null) 은 false 이다.
  - null 체크는 명시적으로 할 필요가 없다. (obj == null) 왜냐면, instanceof 키워드로 묵시적으로 null 체크를 할 수 있기 때문이다.

- equals 메소드 구현 방법

1. == 연산자를 이용해 자기 자신의 참조인지 확인한다.

   단순 성능 최적화용으로, 비교 작업이 빡센 상황인 경우 값 어치를 한다. (ex, List equals)

2. instanceof 연산자로 입력이 올바른 타입인지 확인한다.

   가끔 해당 클래스가 구현한 특정 인터페이스를 비교할 수도 있다.

   이런 인터페이스를 구현한 클래스라면 equals 에서 (클래스가 아닌) 해당 인터페이스를 사용해야한다.

3. 입력을 올바른 타입으로 형변환한다.

   앞서 2번에서 instanceof 검사로 이 단계는 통과다

4. 입력 객체와 자기 자신의 대응되는 '핵심' 필드들이 모두 일치하는지 하나씩 검사한다.

   모두 일치하면 true, 하나라도 다르면 false

- 잘 구현된 예

```
public class PhoneNumber {
    private final short areaCode, prefix, lineNum;

    public PhoneNumber(int areaCode, int prefix, int lineNum) {
        this.areaCode = rangeCheck(areaCode, 999, "지역코드");
        this.prefix = rangeCheck(prefix, 999, "프리픽스");
        this.lineNum = rangeCheck(lineNum, 9999, "가입자 번호");
    }

    private static short rangeCheck(int val, int max, String arg) {
        if(val < 0 || val > max) {
            throw new IllegalArgumentException(arg + ": " + val);
        }
        return (short) val;
    }

    @Override
    public boolean equals(Object o) {
        if(o == this) {
            return true;
        }

        if(!(o instanceof PhoneNumber)) {
            return false;
        }

        PhoneNumber pn = (PhoneNumber) o;
        return pn.lineNum == lineNum && pn.prefix == prefix
                && pn.areaCode == areaCode;
    }
}
```

- **주의사항**
  - float, double 을 제외한 기본 타입은 == 연산자
  - 참조 필드는 equals 메소드
  - float, double 은 부동 소수점을 표현함으로 Float.compare(arg1, arg2), Double.compare(…
  - 배열의 모든 원소가 핵심 필드라면 Arrays.equals 메소드를 중에 하나를 사용
  - null 을 정상 값 취급 하는 경우 - Object.equals(obj, obj) 로 비교해 nullPointerException 예방
  - 비교하기 복잡한 필드를 가진 클래스의 경우 - 필드의 표준형을 저장해둔 후 표준형 끼리 비교하면 자원을 경제적으로 사용 가능하다.
  - equals 성능을 위해 다를 가능성이 크거나 비교하는 비용이 싼 필드를 먼저 비교한다.
  - 너무 복잡하게 해결하려 들지 말자.
  - Object 외의 타입을 매개변수로 받는 equals 메서드는 선언하지 말자.
  - equals 를 다 구현 했으면, 자문해보자 (대칭적, 추이성, 일관적)
  - **equals 메소드를 재정의 했다면 반드시 Hashcode도 재정의해라 (**아이템 11**)**

- 꼭 필요한 경우가 아니라면 equals 를 재정의 하지 말자 (Object.equals 로 충분할 수 있다.)

▼ 11. equals 재정의하려거든 hashCode 도 재정의 해라

- equals 를 재정의 했으면 hashcode도 재정의 해야한다.
  - 그렇지 않으면, HashMap, HashSet 과 같은 컬렉션의 원소로 사용될 때 문제를 일으킨다.
  - 다음은 Object 명세에 적힌 내용이다.
    - equals 비교에 사용되는 정보가 변경되지 않았다면, 애플리케이션이 실행되는 동안 그 객체의 hashcode 메서드는 몇 번을 호출해도 일관되게 항상 같은 값을 반환해야한다. 단, 애플리케이션이 다시 실행한다면 이 값이 달라져도 상관없다.
    - **equals (Object) 가 두 객체를 같다고 판단했단면, 두 객체의 hashCode는 똑같은 값을 반환해야한다.**
    - equals (Object) 가 두 객체를 다르다고 판단했더라도, 두 객체의 hashCode가 서로 다른 값을 반환할 필요는 없다. **단, 다른 객체에 대해서는 다른 값을 반환해야 해시 테이블의 성능이 좋아진다.**
  - 위 내용에서 문제가 되는 내용은 두 번째 내용로, 논리적으로 같은 객체는 같은 해시코드를 반환해야 하는 것이다. 아이템 10에 나온 equals 는 물리적으로 다른 두 객체를 논리적으로는 같다고는 할 수 있다. 하지만, Object의 기본 hashCode 메서드는 이 둘이 전혀 다르다고 판단하여, hashCode 의 값이 서로 다를 것이다.
    - 예를 들어

```
Map<Unit, String> m = new HashMap<>();
m.put(new Unit(100, 100, 0), "질럿");

m.get(new Unit(100, 100, 0));
// get 의 결과 값이 "질럿"이 아닌 null 이 반환 된다.
```

      - 여기서 두 개의 Unit 인스턴스가 사용되었고, hashCode 가 재정의되어 있지 않음으로, 동등(논리적 동치)한 두 객체가 서로 다른 해시코드를 반환해 두 번째 규약을 지키지 못 한 것이다.
      - HashMap 은 해시코드가 다른 엔트리끼리는 동치성 비교를 시도 조차 하지 않는다.
  - 동치인 모든 객체에서 똑같은 해쉬 코드를 반환

```
@Overrid
public int hashCode() { return 42; }
```

    - 동치인 모든 객체에게 똑같은 값만 내어줌으로, 모든 객체가 해시 테이블의 버킷 하나에 담겨 마치 연결 리스트 처럼 동작하게 된다. 즉 hash테이블의 평균 수행 시간인 O(1) 이 아닌 O(n)으로 느려질 것이다.
    - 좋은 해시 함수라면 서로 다른 인스턴스에 다른 해시코드를 반환한다. 이것이 바로 hashCode의 세 번째 규약이 요구하는 속성이다.
    - 이상적인 해시 함수는 주어진 인스턴스들을 32비트 정수 범위에 균일하게 분배해야 한다.
  - 좋은 HashCode를 구현하는 방법
    1. int 변수 result 로 선언 후 값 c로 초기화한다.
       a. 이때 c 는 해당 객체의 첫 번째 **핵심 필드** 단계 2.a 방식으로 계산한 코드이다.
       b. 여기서 **핵심 필드**는 equals 비교에 사용되는 필드를 뜻한다.
    2. 해당 객체의 나머지 핵심 필드 f 각각에 대해 다음 작업을 수행한다.
       a. 해당 필드의 해시코드 c를 계산한다.
          i. 기본 타입 필드라면, Type.hashCode(f)를 수행한다. 여기서 Type 은 해당 기본 타입의 박싱 클래스이다.

ii. 참조 타입 필드면서, 이 클래스의 equals 메서드가 이 필드의 equals를 재귀적으로 호출해 비교한다면, 이 필드의 hashCdoe를 재귀적으로 호출한다.
    - 계산이 더 복잡해지면 이 필드의 표준형을 만들어 그 표준형의 hashCode를 호출한다.
    - 필드 값이 null 이면 0을 사용한다. (다른 상수도 가능하지만, 대체적으로 0을 사용함)
iii. 필드가 배열이라면, 핵심 원소 각각의 별도 필드처럼 다룬다. 이상의 규칙을 재귀적으로 적용해 각 핵심 원소의 해시코드를 계산한 다음, 단계 2.b 방식으로 갱신한다.
    - 배열에 핵심 원소가 하나도 없다면 단순히 상수 (대체적으로 0) 을 사용한다.
    - 모든 원소가 핵심 원소라면 Arrays.hashCode를 사용한다.

b. 단계 2.a 에서 계산한 해시코드 c로 result 를 갱신한다. 코드로는 다음과 같다.

   result = 31 * result + c;

3. result 를 반환한다.

- 주의점
    - 동치인 인스턴스가 서로 다른 해시코드를 반환한다면, 뭔가 잘 못 된 것이다.
    - 파생 필드는 제외해도 된다. (다른 필드로 부터 계산해 낼 수 있는 필드는 무시 가능)
    - equals 비교에 사용되지 않은 필드는 “**반드시**” 제외해야 한다. (그렇지 않으면, 두 번째 규약을 어기게 된다.)
    - 31를 곱하는 이유
        - 31 * result는 필드를 곱하는 순서에 따라 result 값이 달라지게 해준다.
        - 곱셈을 시프트 연산과 뺄셈으로 대체해 최적화 할 수 있다.
          (31 * i == (i << 5) - i)
    - **성능을 높인다고 해시코드를 계산시 핵심 필드를 생략하면 안된다.**
    - hashCode가 반환하는 값의 생성 규칙을 API 사용자에게 자세히 공표하지 말자, 그래야 클라이언트가 이 값에 의지하지 않게 되고, 추후 계산 방식을 변경 할 수 있다.
- 전형적인 hashCode 메서드 - 아이템 10 잘 된 equals

```
@Override
public int hashCode() {
    int result = Short.hashCode(areaCode);
    result = 31 * result + Short.hashCode(prefix);
    result = 31 * result + Short.hashCode(lineNum);
    return result;
}
```

    - PhoneNumber 인스턴스의 핵심 필드 3개만 사용해 간단한 계산을 수행하고, 그 과정에서 비 결정적 요소는 전혀 없음으로 동치인 인스턴스들은 같은 해쉬코드를 갖게된다.
- Objects 클래스에 임의의 개수 만큼 객체를 받아 해시코드를 계산해주는 hash 메소드를 제공한다.

```
@Overide public int hashCode() {
    return Objects.hash(lineNum, prefix, areaCode);
}
```

    - 단, 속도가 더 느리다. 성능에 민감 하지 않는 경우 사용한다.
        - 입력 인수를 담기 위한 배열이 만들어짐
        - 입력 중 기본 타입이 있다면 박싱, 언 박싱 과정을 거침

        - 클래스가 불변이고 해시코드를 계산하는 비용이 크다면, 매번 계산하기 보다는 캐싱하는 방식을 고려해야 한다. (인스턴스가 생성시 해쉬코드를 생성해 둔다.)
        - hashCode 의 캐싱과 지연 초기화 - 스레드 안정성까지 고려해야 한다. (아이템3 지연 초기화)

```
private int hashCode; // 캐시를 위함.

@Override
public int hashCode() {
    int result = hashCode;
    if (result == 0) {
        result = Short.hashCode(areaCode);
        result = 31 * result + Short.hashCode(prefix);
        result = 31 * result + Short.hashCode(lineNum);
        hashCode = result;
    }
    return result;
}
```

- equals 와 hashCode 를 자동으로 만들어주는 AutoValue 프레임워크도 있다.

▼ 12. toString은 항상 재정의해라

- toString 의 일반 규약으로, 간결하면서, 사람이 읽기 쉬운 형태의 유익한 정보를 반환해야한다.
    - Object 의 기본 toString 메소드는 **클래스이름@16진수의해시코드** 를 반환한다.
    - toString 을 잘 구현한 클래스는 보다 디버깅이 쉽다.
        - toString 을 재정의하지 않은 클래스의 객체를 참조하는 컴포넌트가 오류 메시지를 로깅시 이상한 메시지만 로그에 출력될 것이다.
    - toString 메소드는 println, 문자열 연결, 디버거의 객체를 출력시 등, 직접 코드를 사용해 호출 하지 않아도, 다른 어딘가 에서 사용될 것이다.
    - 아래는 컬렉션 중 ArrayList를 사용해 print 한 예이다.

```
List<Unit> list = new ArrayList<>();
list.add(new Unit(35, 6, "저글링"));
list.add(new Unit(40, 8, "마린"));
list.add(new Unit(100, 32, "질럿"));
System.out.println(list);

// toString 재정의 O (IDE 기본 제공)
[Unit{hp=35, damage=6, name='저글링'}, Unit{hp=40, damage=8, name='마린'}, Unit{hp=100, damage=32, name='질럿'}]
// toString 재정의 X
[Unit@232204a1, Unit@4aa298b7, Unit@7d4991ad]
```

        - 보면 toString 을 재정의 한 경우가 보다 더 유익한 정보로 보인다.
        - 다만, 위 객체가 거대하거나 객체 내용이 엄청 크다면 무리가 있을 것이다.
            - 그런 경우에는 요약된 정보를 표현해줘야 한다.
- toString을 구현 할 때면, 반환값의 포맷을 문서화 할지 정해야한다.
    - 전화번호나 행렬 같은 값 클래스라면 문서화 하는 것이 좋다.
    - 표준을 명시하면 그 객체는 표준적이고, 명확하고 사람이 읽을 수 있게 된다.
    - 단, 포맷을 한 번 명시하면, 평생 그 포맷에 얽매이게 된다. - 유연성이 떨어진다.
    - **포맷을 명시하든 아니든 의도는 명확하게 밝혀야 한다.**
        - toString이 반환한 값에 포함된 정보를 얻어 올 수 있는 API를 제공하다.

▼ 13. clone 재정의는 주의해서 진행해라

- clone 메소드
  - 객체의 모든 필드를 복사하여 새로운 객체에 넣어 반환하는 동작을 수행한다. 즉, 필드의 값이 같은 객체를 새로 만드는 것이다.
  - **사실상 생성자와 같은 효과를 내야 한다. 즉 clone 은 원본 객체에 아무런 피해를 입히면 안되고, 동시에 복제된 객체의 불변식을 보장 해야한다.**
  - 또한, 해당 메소드를 사용하려면, Cloneable 이라는 인터페이스를 구현해야 clone 메소드를 사용 할 수 있습니다. Cloneable 인터페이스에 해당 함수가 정의 되어 있을 것 같지만, 아닙니다. clone 메소드는 Object 클래스에 정의되어 있으며, **Cloneable 인터페이스는 Object 클래스의 protected 의 clone 메소드의 동작 방식**을 결정해줍니다. 객체를 복사 할지, CloneNotSupportedException 을 던질지 결정합니다.
- **Object 클래스 clone 메소드의 일반 규약 (구현하기 까다롭다…)**
  1. x.clone() != x
     - 위는 참이다. 원본 객체와 복사 객체는 서로 다른 객체이다.
  2. x.clone().getClass() == x.getClass()
     - 위는 참이다. 하지만 반드시 만족해야 하는 것은 아니다.
     - super.clone()을 호출해 얻은 객체를 clone 메소드가 반환한다면, 이 식은 참이다. 관례상, 반환된 객체와 원본 객체는 독립적이어야 한다. 이를 만족하려면 super.clone으로 얻은 객체의 필드 중 하나 이상을 **반환 전에** 수정해야 할 수도 있다. (얕은 복사)
  3. x.clone().equals(x)
     - 위는 참이지만 필수는 아니다.
- 가변 객체가 포함된 클래스에서 clone 메소드
  - cloneable 인터페이스를 구현한 클래스가 불변 객체만 있다면, clone 메소드가 정상적이지만, **가변 객체가 포함된 경우**에는 원본 인스턴스와 복사 인스턴스의 가변객체가 동일한 객체를 참조하기 때문에, 문제가 발생된다.
  - 물론 해당 가변 객체도 clone 해서 내보내면 되긴 합니다. (재귀적 호출)
    - 하지만, 해당 가변 객체가 final 이라면 동작 할 수 없습니다. final 필드에 새로운 값을 할당 할 수 없기 때문, 그래서 clone 이 되려면 final 필드는 있어서는 안됩니다.
    - 또한, **clone 는 재귀적 호출 만으로도 충분하지 않을 수 있다.**
      - 안에 어떤 가변 객체들이 clone 을 이어가면서, 어떤 행동을 할지 보장하지 못하기 때문이다. (책의 예로는 해쉬 테이블의 키쌍을 담는 linkedList로 사용해 list 가 긴 경우 stackOverFlow의 위험을 소개한다.)
  - 배열은 clone 메소드를 사용을 권장한다. (유일하게 clone 기능을 제대로 사용하는 예)
- 결론…
  - Cloneable을 구현하는 모든 클래스는 clone을 재정의 해줘야한다.
  - 접근제한자는 public으로 하며 반환타입은 클래스 자신으로 변경한다. (cast)
  - super.clone을 호출한 후 가변 객체 필드를 전부 복사(재귀적 호출) 해준다.
  - **복사 생성자와 복사 팩터리를 사용**
    - clone, cloneable 과 비교 하면
      - 위험 천만한 객체 생성 매커니즘 (생성자를 사용하지 않는다.) 를 사용 안함

        - 엉성하게 문서화된 규약에 기대지 않는다.
        - final 필드 용법과 충돌하지 않는다.
        - 불필요한 검사 예외를 던지지 않는다.
        - 형변환도 필요없다.
        - 원본의 구현 타입에 얽매이지 않고 복제본의 타입을 직접 선택할 수 있다
            - Hashset s = new HashSet();
            - new TreeSet(s);
- 공변성과 반공변성

▼ 14. Comparable 을 구현할지 고려해라