EJ

이펙티브 자바 CP.2

작성 일시	@2023년 1월 7일 오후 8:01
최종 편집 일시	@2023년 1월 9일 오후 10:05
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

▼ 12. toString은 항상 재정의해라

▼ 13. clone 재정의는 주의해서 진행해라

▼ 14. Comparable 을 구현할지 고려해라