Obey the general contract when overriding equals

개요

equals 메서드는 겉보기에 구현하기 쉬워보이지만 어렵습니다만... 다양한 문제를 발생시킬 수 있습니다.

따라서 이 문제를 피하는 가장 좋은 방법은 구현하지 않는게 가장 좋습니다.

아래의 4가지 상황 중 하나에 속한다면 equals를 그대로 두세요. 그대로 두면 자기 자신과 비교할 때만 true라고 반환할테니까요.

4가지 상황

1. 각 인스턴스가 본질적으로 고유할 때

- Thread Class

2. class가 논리적 동치성을 검증할 일이 없을 때

- java.util.regex.Pattern에서 클라이언트는 두 pattern이 동일한 regex를 가진지 관심없으니 이에 해당

3. super class에서 정의한 equals가 하위 클래스에서도 들어맞을 때

- AbstractSet을 구현한 Set클래스는 동일한 equals의미를 가지고 있습니다.

4. class가 private이거나 package-private일 경우

- equals가 절때 호출될 일이 없으므로. 만약에 코너 케이스에 대비하려면 호출이 exception을 던지도록 하세요.

언제 equals를 쓰는게 적절할까?

1. object의 동치성이 논리적 동치성과 다를 경우

2. super class에서 equals를 오버라이딩하지 않았을 경우.

value class가 이에 해당합니다.

1. value class를 사용한다면, object가 동등한지가 아니라 값이 동등한지 비교하고 싶어 합니다.
2. equals를 정의한다면 Map이나 Set에서도 사용 가능합니다. (hashCode)

값 클래스라해도 필요없는 경우

동일 값의 인스턴스가 2개 이상 만들어지지 않음을 보장된다면 equals 정의할 필요가 없네요. 어차피 다른 객체라면 다른 값을 가지고 있는거니까요.

equals메서드를 재정의할 때 일반규약

1. 반사성(Reflexive)

   ex:) x.equals(x)

2. 대칭성(Symmetric)

   ex:) x.equals(y) == y.equals(x)

3. 추이성(Transitive)

   x.equals(y) , y.equals(z) 이면 x.equals(z)

4. 일관성(Consistent)

   여러번 호출하더라도 동일한 결과를 반환시켜야 함

5. non-null

   x.equals(null)에서 x가 non-null reference value여야 함. 따라서 x.equals(null)은 false를 반환해야 한다.

> 세상에 홀로 존재하는 클래스는 없다 - John Donne

클래스들과 상호작용 속에서, 위 equals 규약을 모두 따른다고 가정하고 클래스를 사용합니다.(ex: Collections) 따라서 위를 지키지 않으면 알 수 없는 에러가 발생하죠.

Equivalence relation의 의미는 뭘까?

책에서 언급한 것보다 좀 더 간단히 말하자면, class A와 class B간 핵심 속성이 동일할 때 동치관계라고 말할 수 있습니다.

자세히 알고 싶다면 https://en.wikipedia.org/wiki/Equivalence_relation 이 링크를 참고하시면 됩니다.

각 속성 설명 - 위배와 순응

1. 반사성 (Reflexive)

정의

자기 자신과 같아야한다는 특성입니다.

위반시 현상

위반시에 collection에 값을 넣고도, 들어있는지 contains로 확인해도 넣은 객체는 없다고 나온다.

2. 대칭성 (Symmetric)

정의

두 object가 서로 동일여부에 동일한 결과를 반환해야 한다.

첫번째에 비해 위반하기 쉬우므로 설명을 드립니다.

위반 사례

대소문자를 구별하지 않는 class

```
// Broken - violates symmetry!  (Page 39)
public final class CaseInsensitiveString {
    private final String s;

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

    // Broken - violates symmetry!
    @Override public boolean equals(Object o) {
        if (o instanceof CaseInsensitiveString)   // [1]
            return s.equalsIgnoreCase(
                    ((CaseInsensitiveString) o).s);
        if (o instanceof String)  // One-way interoperability!  // [2]
            return s.equalsIgnoreCase((String) o);
        return false;
    }


    }
```

1. [1]

[1]부분을 보면 CaseInsensitiveString class. 즉 본인 클래스에 대해서는 정상적인 비교를 합니다.

2. [2]

[2]를 보면 문자 비교에서 String도 비교를 하려합니다. **문자'**가 동일한 지 보려는 거니까요. 해당 클래스에 String을 본인 클래스로 형변환하여 동일성 여부를 구현했으므로 이 equals는 문제가 없겠지요.

3. String에선...?

그런데 String에선 어떨까요? String object는 CaseInsensitiveString class에서 구현한 것처럼 equals가 구현되어 있나요? String은 Case class를 모릅니다. 따라서 역으로 적용하면 문제가 됩니다.

위반시 현상

```
    // Demonstration of the problem (Page 40)
    public static void main(String[] args) {
        CaseInsensitiveString cis = new CaseInsensitiveString("Polish");
        String s = "polish";

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

        System.out.println(list.contains(s));
```

OpenJDK에 따라 false를 반환할 수도, true를 반환할 수도 exception을 던질 수도 있습니다. 즉, 결과를 예측할 수 없게 됩니다.

해결 방법

CaseInsentiveString class를 String클래스와 연동할수 없습니다. 문자를 비교한다고 생각해서 String 클래스랑 비교하려 하지마세요. 저자는 **허황된 꿈**이라고까지 표현하죠! 그저 자신의 클래스와 비교만 가능합니다. 따라서 다음처럼 equals를 구현해주어야 합니다.

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

3. 추이성 (Transitive)

정의

첫번째 객체가 두번째 객체와 동일하고 두번째 객체가 세번째 객체와 동일하다면, 첫번째 객체는 세번째 객체와 일치하다는 규약입니다.

이 또한 쉽게 위반할 수 있습니다.

위반 사례

상위 클래스에는 없는 필드(value component)를 하위 클래스의 필드에 넣으려는 상황을 가정합시다.

super class - Point

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

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

    @Override public boolean equals(Object o) {
        if (!(o instanceof Point))
            return false;
        Point p = (Point)o;
        return p.x == x && p.y == y;
    }

//    // Broken - violates Liskov substitution principle (page 43)
//    @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;
//    }

    // See Item 11
    @Override public int hashCode()  {
        return 31 * x + y;
    }
}
```

sub class - ColorPoint

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

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

아무것도 안할 경우

equals는 sub class의 color는 비교하지 않을 겁니다. 수퍼클래스에서 x,y만 비교하도록 했으니까요.

이대로 둔다면 규약을 위반한 사례는 아니지만 원하는 결과가 아니게 되죠. 서로 다른 color값을 가지고 있지만 x,y값이 같다면 ColorPoint가 동일하다 판단하니까요.

side step - ColorPoint끼리만 비교

그래서 아래처럼 ColorPoint가 아니면 false를 반환하도록 만든다면 어떻게 될까요?

```
// Broken - violates symmetry!  (Page 41)
@Override public boolean equals(Object o) {
  if (!(o instanceof ColorPoint))
    return false;
  return super.equals(o) && ((ColorPoint) o).color == color;
```

Point와 ColoredPoint를 비교할 때 문제가 발생합니다.

Point의 equals는 색상은 무시한 채 x,y값만 비교할 것이며, ColoredPoint의 equals는 타입이 다르다고 false를 반환할 것입니다.

side step - mixed comparison시 color는 무시하여 비교

coloredPrint.equals(point)를 적용시에는 point와 비교시에는 하단 [1]에서 보이는 것처럼 o에서 비교하여 color 비교를 무시하는 방법입니다.

```
//    // Broken - violates transitivity! (page 42)
@Override public boolean equals(Object o){
        if(!(o instanceof Point))
        return false;

        // If o is a normal Point, do a color-blind comparison
        if(!(o instanceof ColorPoint)) // [1]
        return o.equals(this);

        // o is a ColorPoint; do a full comparison
        return super.equals(o)&&((ColorPoint)o).color==color;
        }
```

```
ColorPoint p1 = new ColorPoint(1,2,Color.RED);
Point p2 = new Point(1,2);
ColorPoint p3 = new ColorPoint(1,2,Color.BLUE);
System.out.printf("%s %s %s%n", p1.equals(p2),p2.equals(p3),p1.equals(p3));
```

이를 실행할 경우, p1, p2 와 p2,p3는 true를 반환할 것입니다. 하지만 p1,p3비교시에는 color까지 비교하므로 false를 반환하며 결국 transitive를 위반한다고 볼 수 있죠.

한편으로, 이러한 접근은 무한 재귀에 빠질 수 있습니다. Point를 상속받은 myColorPoint와 mySmellPoint가 equals 비교시 [1]을 서로 호출하니까요.

본질적인 문제

모든 객체지향 언어의 동치관계에서 비롯된 문제.

구체 클래스를 확장해 새로운 값을 추가하면서 equals 규약을 만족시킬 수 없습니다.

side step - instance of 대신 getClass이용

getClass로 구현하면 되지 않냐고 하는데 이는 또 LSP위반을 하게 됩니다.

동일한 Class에 대해서만 적용가능하니까요.

```
public class CounterPoint extends Point {
    private static final AtomicInteger counter =
            new AtomicInteger();

    public CounterPoint(int x, int y) {
        super(x, y);
        counter.incrementAndGet();
    }
    public static int numberCreated() { return counter.get(); }
}
```

```
// Test program that uses CounterPoint as Point
public class CounterPointTest {
    // Initialize unitCircle to contain all Points on the unit circle  (Page
43)
    private static final Set<Point> unitCircle = Set.of(
            new Point( 1,  0), new Point( 0,  1),
            new Point(-1,  0), new Point( 0, -1));

    public static boolean onUnitCircle(Point p) {
        return unitCircle.contains(p);
    }

    public static void main(String[] args) {
        Point p1 = new Point(1,  0);
        Point p2 = new CounterPoint(1,  0);

        // Prints true
        System.out.println(onUnitCircle(p1));

        // Should print true, but doesn't if Point uses getClass-based equals
        System.out.println(onUnitCircle(p2));
    }
}
```

앞서 말했듯, 확장하면서 새로운 value component를 추가하면서 equals를 만족시키는 방법은 없습니다.

하지만 상속이 아니라, composition을 적용한다면 가능해지죠.

규칙 적용 방법

따라서 기존 클래스는 새로운 클래스의 field로, 그리고 새로운 요소는 새 클래스에 추가하는 방식으로 구현하면 됩니다.

그리고 기존 클래스 타입은 asPoint()라는 메서드로 반환하도록 하였습니다.

```java
// Adds a value component without violating the equals contract (page 44)
public class ColorPoint {
    private final Point point; // 기존 클래스
    private final Color color; // 새로운 요소

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

    /**
     * Returns the point-view of this color point.
     */
    public Point asPoint() {
        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);
    }
}
```

java libarary에서 대칭성 위반한 예

- Timestamp

  Date를 확장하여 값을 추가했기 때문에 같이 사용할 경우 알 수 없는 문제가 발생함.

4. 일관성 (Consistent)

정의

두 객체가 수정되지 않는 한, 두 객체가 같다면 영원히같고, 다르다면 영원히 달라야 한다는 규약

따라서, 불변객체는 반드시 언제나 같아야 함.

위반 사례

class가 가변이든 불변이든, equals 판단에 신뢰할 수 없는 자원이 끼어들면 안됩니다. 그렇다면 만족시키기 매우 어렵기 때문이죠.

java.net.URL이 그와 같습니다. 호스트 주소에 따라 URL값이 변경되니까요.

적용방법

이럴땐 메모리에 존재하는 개체만을 사용한 결정적(deterministic)계산만 허용해야 합니다. (아마 객체 주소 값을 말하는 것 같네요)

5. non-nullity

정의

모든 객체가 null과 같으면 안된다는 규약이다.

위반

이를 지키지 않으면 NullPointException이 발생합니다.

적용 방법

bad case - explicit test

```
@override public boolean equals(Object o) {
  if (o == null)
    return false;
}
```

이 방법은 불필요로 합니다. 이유는 하단 예시를 보면 명확합니다.

good case - implicit test

```
@override public boolean equals(Object o) {
  if (! (o instanceof MyType)
    return false;
  MyType mt = (MyType) o;
}
```

동등성을 비교하기 위해서는 반드시 cast가 필요합니다. 따라서 instanceof로 타입 검사를 해야 합니다. 이 type check에서 o가 null이면 false를 반환하므로 명시적으로 null check를 해줄 필요가 없습니다.

equals 단계별 구현 방법

1. ==를 이용해 입력이 자기 자신의 참조인지 확인

   성능 최적화 목적. 자기 자신이면 true반환
2. instance of 연산자로 입력이 올바른 타입인지 확인

동일 타입끼리 비교면 문제가 없는데, interface를 구현한 케이스라면 주의해야 한다. interface를 구현한 서로 class끼리 비교할 수 있기 때문에 equals class가 아닌, interface로 캐스팅하여 비교해야 합니다.

Set,List,Map,Map.entry가 이에 해당

3. 올바른 type으로 형변환

   2가 성공했다면 반드시 성공

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

   인터페이스라면 인터페이스 메서드를 이용해 필드들을 비교해야 한다.

비교 방법

1. primitive type (float, double 제외)

   == 연산자로 비교

2. float,double

   Float.compare이나 Double.compare로 비교 특수한 부동소수점 때문에.

   float.equals, double.equals도 대신사용 가능하나 성능 이슈있음.

3. 배열

   배열 원소 각각을 위 방법 적용

   또는 모두가 핵심원소일 필드일 경우엔 Arrays.equals 적용

4. null도 정상값 취급할 경우

   Object.equals()를 이용

5. CaseInsensitiveString처럼 비교가 복잡할 경우

   표준형을 두어 비교.(불변 객체일 경우 매우 유용 가변이라면 최신으로 유지)

6. 필드의 비교 우선순위 정하기

   비교 순서에 따라 성능이 크게 바뀔 수 있다. lock같이 핵심필드가 아니라면 비교하면 안되는데, 파생필드가 객체 전체의 상태를 대표할 경우엔 다름.

   ex:) Polygon

   자신의 영역을 캐싱해두는 Polygon클래스를 생각해보자. 그렇다면 모든 정점을 비교할 필요없이 캐싱만 비교하면 됨.

equals 잘 구현했는 지 3가지 자문

1. 대칭성
2. 추이성
3. 일관성

나머지 2가지 요소는 어기기 어렵기 때문에 크게 신경쓰지 않아도 됨

그리고 자문요소를 테스트해보자.

```
// Class with a typical equals method (Page 48)
public final class PhoneNumber {
    private final short areaCode, prefix, lineNum;

    public PhoneNumber(int areaCode, int prefix, int lineNum) {
        this.areaCode = rangeCheck(areaCode, 999, "area code");
        this.prefix   = rangeCheck(prefix,   999, "prefix");
        this.lineNum  = rangeCheck(lineNum, 9999, "line num");
    }

    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;
    }

    // Remainder omitted - note that hashCode is REQUIRED (Item 11)!
}
```

마지막 주의사항

- equals 재정의시 hashcode도 재정의할 것
- 복잡하게 해결하려하지 말 것

  필드의 값만 비교하는 수준으로 끝내기

  별칭사용 금지 File class라면 심볼릭 링크를 비교해 같은 파일을 가르키는지 확인하려 들면 안됨.
- Object외 타입을 매개변수로 받지말자. 그러면 오버로드한거임.

  ```
  public boolean equals (MyClass o) {}
  ```

  기존것을 두고 추가한것이라도 이러지 말 것

  override 애노테이션이 거짓 양성반응 보임. (없는데 있다고 말하는 것)

테스트 프레임워크

AutoValue프레임워크나 IDE이용