Always override hashCode when you override equals

개요

hashCode를 오버라이드하지 않는다면, hashCode 규약을 어기는 것이며, 이는 Collection의 원소로 사용할 때 문제가 됩니다.(HashMap, HashSet 등)

규약

- When the `hashCode` method is invoked on an object repeatedly during an execution of an application, it must consistently return the same value, provided no information used in `equals` comparisons is modified. This value need not remain consistent from one execution of an application to another.
- If two objects are equal according to the `equals(Object)` method, then calling `hashCode` on the two objects must produce the same integer result.
- If two objects are unequal according to the `equals(Object)` method, it is *not* required that calling `hashCode` on each of the objects must produce distinct results. However, the programmer should be aware that producing distinct results for unequal objects may improve the performance of hash tables.

hashCode를 재정의하지 않았을 경우 두번째 규약을 어기게 되며, 동일 객체가 다른 값을 던진다.

위반시 문제

```
Map<PhoneNumber, String> m = new HashMap<>();
m.put(new PhoneNumber(707,867,5309), "Jenny")
```

만약에 `m.get(new PhoneNumber(707,867,5309))`를 수행할 경우, 서로 다른 hash값으로 인해 null을 반환하게 된다. hashMap은 해쉬코드가 다른 서로 다른 엔트리끼리는 전혀 비교하지 않기 때문.

hashCode짜는 방법

bad-case

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

모든 객체가 같은 값을 가지도록 만들었으므로 틀리진 않았는데 좋진 않다. hash table 알고리즘상 링크드 리스트처럼 O(n)이 될 수 있기 때문.

가장 좋은 해쉬코드는 서로 equals하지 않은 객체는 다른 hashCode를 가지도록하는 것이다.(3번째 규약)

이상적으로는 32비트에 균일하게 분포되는 것이다. 여기선 간단한 알고리즘을 소개하겠다.

심플한 알고리즘

1. Declare an `int` variable named `result`, and initialize it to the hash code `c` for the first significant field in your object, as computed in step 2.a. (Recall from Item 10 that a significant field is a field that affects equals comparisons.)
2. For every remaining significant field `f` in your object, do the following:
   a. Compute an `int` hash code `c` for the field:
      i. If the field is of a primitive type, compute *Type*`.hashCode(f)`, where *Type* is the boxed primitive class corresponding to `f`'s type.
      ii. If the field is an object reference and this class's `equals` method compares the field by recursively invoking `equals`, recursively invoke `hashCode` on the field. If a more complex comparison is required, compute a "canonical representation" for this field and invoke `hashCode` on the canonical representation. If the value of the field is `null`, use `0` (or some other constant, but `0` is traditional).
      iii. If the field is an array, treat it as if each significant element were a separate field. That is, compute a hash code for each significant element by applying these rules recursively, and combine the values per step 2.b. If the array has no significant elements, use a constant, preferably not `0`. If all elements are significant, use `Arrays.hashCode`.
   b. Combine the hash code `c` computed in step 2.a into `result` as follows:
      ```
      result = 31 * result + c;
      ```
3. Return `result`.

그리고 직관을 테스트 코드로 작성하면 되고, 또는 AutoValue를 이용하면 된다.

단계별 설명

1단계

1. 다른 field로부터 유추가능한 field는 제외해도 좋다.
2. equals에 사용되지 않은 필드는 반드시 제외하라.

   2번째 규약을 어길 가능성이 높기 때문

2단계

1. 31*result의 효과는 필드를 곱하는 순서에 따라 결과가 달라지게 된다. 이를 하지 않으면 anagram의 모든 값이 같아지게 된다.
2. 31을 곱한 이유는 홀수이면서 소수이기 때문이다.
   - 짝수이고 오버플로우가 발생하면 정보를 잃기 때문이다.
   - 2를 곱한 이유는 쉬프트 연산
   - 소수는 그렇게 해와서.
   - 따라서 이와같아진다 (i << 5) - 1

예시

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

장점

1. 간단한 결정적 요소의 결과를 반환
2. 계산의 input으로는 번호의 중요한 필드로만 계산이 된다.
3. 간단하고 빠르고 해시버킷으로부터 훌륭히 분배해줌.

더나아가..

만약 collision을 더 줄여주는 더 좋은 해쉬 함수를 쓰고 싶다면 Guava의 `com.google.common.hash.Hasing` 참조하라.

Object class의 hash

위처럼, Object class의 hash()가 object의 임의의 수를 반환해주는 기능을 가지고 있다.

하지만 성능이 떨어져서 좋지 않다. 그이유는 다음과 같다.

1. 입력변수를 배열로 담기 위해 배열 선언

2. 입력변수 중 primitive type이 있다면 autoboxing 수행

해싱의 비용이 클 경우

해싱의 비용이 클 경우엔 cashing하는 방법이나 layLoading을 적용하면 된다. 이때 layLoading 적용시에는 thread safe까지 고려해야 한다.

hashCode 주의사항

1. 성능 높힌답치고 중요한 필드를 빼먹지 않기

이 필드가 해쉬값을 고르게 분포해주는 핵심 필드일 수도 있다.

실제로 Java2에서 이러한 일이 발생함. 최대 16개의 문자만으로 hash를 적용했고, 길이가 길 경우엔 일정 간격으로 나눴다. 따라서 URL처럼 계층적인 이름을 사용했을 경우엔 매우 느렸음

2. hashCode 생성 규칙을 API사용자에게 보여주지 않을 것

클라이언트가 이에 의존하게 되면 유연성을 잃고 추후 개선을 하기 어려워짐.

수많은 JAVA libraries들이 hashCode가 반환하는 정확한 값을 알려줬기 때문에 추후 개선 여지조차 없애버렸다.

[string의 해쉬코드]

```
public int hashCode() {
    int h = hash;
    if (h == 0 && value.length > 0) {
        char val[] = value;

        for (int i = 0; i < value.length; i++) {
            h = 31 * h + val[i];
        }
        hash = h;
    }
    return h;
}
```