Item 14: Comparable을 구현할 지 고려하라

유도진

목차

- Comparable에 대하여
- comparaTo 메서드의 일반 규약
- 가장 중요한 필드부터 비교하자
- 기본 숫자 타입은 제공하는 비교 메서드를 이용하자
- Comparator와의 비교

Comparable에 대하여

```
public interface Comparable<T> {
    public int comparaTo(@NotNull T o)
}
```

- `comparaTo` 메서드 하나만을 가지는 인터페이스
- `Object` 의 `Equals` 메서드와 2가지 차이점이 존재
    i. 동치성을 넘어 순서 비교 가능
    ii. 제너릭 사용 가능
- compareTo를 구현하여 해당 인터페이스를 활용하는 수많은 제너릭 알고리즘과 컬렉션의 힘을 이용

compareTo 메서드의 일반 규칙

- 객체 자신과 주어진 객체의 순서를 비교. 객체 자신이 주어진 객체보다 작으면 음의 정수를, 같으면 0을, 크면 양의 정수를 반환

compareTo 메서드의 일반 규칙

- 일반 규칙을 설명하기 위해서 부호함수(signum)에 대해서 이해가 필요
- signum함수는 해당 숫자가 양수라면 1로, 0이라면 0으로, 음수라면 -1로 변환

```
Returns the signum function of the specified int value. (The return value is -1 if the specified value is negative; 0 if the specified value is zero; and 1 if the specified value is positive.)
매개변수:  i – the value whose signum is to be computed
반환:      the signum function of the specified int value.
시작 시간: 1.5
public static int signum(int i) {
    // HD, Section 2-7
    return (i >> 31) | (-i >>> 31);
}
```

compareTo 메서드의 일반 규칙

1. 모든 x,y에 대하여 `signum(x.compareTo(y)) == -signum(y.compareTo(x))`.
   따라서 `x.compareTo(y)`가 에러를 던진다면 `y.compareTo(x)`도 에러 던짐
   ⇒ 두 객체 참조의 순서를 바꿔 비교해도 예상된 결과가 나와야 한다. (대칭성)
2. 추이성 보장. (추이적 관계, transitive relation)
   `x.compareTo(y) > 0`이면서 `y.compareTo(z) > 0` 라면 `x.compareTo(z) > 0` 이다
   ⇒ x가 y보다 크고 y가 z보다 크다면, x는 당연히 z보다 커야한다! (추이성)
3. 모든 z에 대해 `x.compareTo(y) == 0`이면
   `signum(x.compareTo(z))==signum(y.compareTo(z))`
   ⇒ 같은 객체라면 순서를 바꿔도 동일하게 같다고 나와야 한다. (반사성)

compareTo 메서드의 일반 규칙

4. (권고사항) `(x.compareTo(y) == 0) == x.eqauls(y)`
   ⇒ 순서상 동일하다는 것은 같은 객체라는 것과 같다. Collection들은 동치성 비교를 할 때 `equals` 대신 `compareTo`를 사용한다.
   compareTo 규약을 지키지 못하면 비교 하는 클래스와 어울리지 못함

compareTo 메서드의 일반 규칙

- treeMap

```java
final Entry<K,V> getEntry(Object key) {
    // Offload comparator-based version for sake of performance
    if (comparator != null)
        return getEntryUsingComparator(key);
    Objects.requireNonNull(key);
    /unchecked/
        Comparable<? super K> k = (Comparable<? super K>) key;
    Entry<K,V> p = root;
    while (p != null) {
        int cmp = k.compareTo(p.key);
        if (cmp < 0)
            p = p.left;
        else if (cmp > 0)
            p = p.right;
        else
            return p;
    }
    return null;
}
```

compareTo 메서드의 일반 규칙

- HashMap

```
final Node<K,V> getNode(Object key) {
    Node<K,V>[] tab; Node<K,V> first, e; int n, hash; K k;
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (first = tab[(n - 1) & (hash = hash(key))]) != null) {
        if (first.hash == hash && // always check first node
            ((k = first.key) == key || (key != null && key.equals(k))))
            return first;
        if ((e = first.next) != null) {
            if (first instanceof TreeNode)
                return ((TreeNode<K,V>)first).getTreeNode(hash, key);
            do {
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    return e;
            } while ((e = e.next) != null);
        }
    }
    return null;
}
```

가장 중요한 필드부터 비교하자

- 클래스의 핵심 필드가 여러 개라면 어느 것을 먼저 비교할 지가 중요해진다.
- 가장 핵심적인 필드부터 비교해 나가자. 비교 결과가 0이 아니라면 그 결과를 곧장 반환하자.

```
public int compareTo(PhoneNumber pn) {
    int result = Short.compare(areaCode, pn.areaCdoe); // 가장 중요한 필드
    if (result != 0) return result;

    result = Short.compare(prefix, pn.prefix) // 두 번쨰로 중요한 필드
    if (result != 0) return result;

    return Short.compare(lineNum, pn.lineNum); // 세 번째로 중요한 필드
}
```

가장 중요한 필드부터 비교하자

- Java 8에서는 Comparator 인터페이스가 일련의 비교자 생성 메서드(comparator construction method)와 팀을 꾸려 메서드 연쇄방식으로 비교자를 생성할 수 있게 되었다. 약간의 성능저하가 뒤따른다.

```
private static final Comparator<PhoneNumber> COMPARATOR =
        comparingInt((PhoneNumber pn) -> pn.areaCode)
            .thenComparingInt(pn -> pn.prefix)
            .thenComparingInt(pn -> pn.lineNumber);

public int compareTo(PhoneNumber pn) {
        return COMPARATOR.compare(this, pn);
}
```

기본 숫자 타입은 제공하는 비교자를 이용하자

```
public class Team implements Comparable<Team>{

    private final int id;
    private final int rank;

    // 생성자 생략

    @Override
    public int compareTo(@NotNull Team other) {
        return this.rank - other.rank;
    }

    @Override
    public String toString() {
        return String.valueOf(this.id);
    }
}
```

기본 숫자 타입은 제공하는 비교자를 이용하자

```
public static void main(String[] args) {
    Team t1 = new Team(1, 1);
    Team t2 = new Team(2, 2);

    List<Team> teams = new ArrayList<>() {{
        add(t1);
        add(t2);
    }};
    Collections.sort(teams);

    System.out.println(teams); // [1, 2]
}
```

기본 숫자 타입은 제공하는 비교자를 이용하자

```
public static void main(String[] args) {
    Team t1 = new Team(1, 1);
    Team t2 = new Team(2, Integer.MIN_VALUE);

    List<Team> teams = new ArrayList<>() {{
        add(t1);
        add(t2);
    }};
    Collections.sort(teams);

    System.out.println(teams); // actual: [1, 2], expect: [2,1]
}
```

기본 숫자 타입은 제공하는 비교자를 이용하자

하지만 위 코드는 예상된 순서는 2,1 이지만 실제로는 1,2로 정렬되게 됩니다. 이는 정수의 오버플로우 때문입니다.

따라서 안전하게 다음과 같이 compareTo를 수정합니다.

```
@Override
public int compareTo(@NotNull Team other) {
    return Integer.compare(this.rank, other.rank);
}
```

Comparator<T>

```java
@FunctionalInterface
public interface Comparator<T> {
```

Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

The implementor must ensure that `signum(compare(x, y)) == -signum(compare(y, x))` for all x and y. (This implies that `compare(x, y)` must throw an exception if and only if `compare(y, x)` throws an exception.)

The implementor must also ensure that the relation is transitive: `((compare(x, y)>0) && (compare(y, z)>0))` implies `compare(x, z)>0`.

Finally, the implementor must ensure that `compare(x, y)==0` implies that `signum(compare(x, z))==signum(compare(y, z))` for all z.

매개변수: o1 – the first object to be compared.
o2 – the second object to be compared.

반환: a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

던지기: `NullPointerException` – if an argument is null and this comparator does not permit null arguments
`ClassCastException` – if the arguments' types prevent them from being compared by this comparator.

API 참고 사항: It is generally the case, but *not* strictly required that `(compare(x, y)==0) == (x.equals(y))`. Generally speaking, any comparator that violates this condition should clearly indicate this fact. The recommended language is "Note: this comparator imposes orderings that are inconsistent with equals."

```java
    @Contract(pure = true)
    int compare(T o1, T o2);
```

Comparator<T>

- Comparable interface는 객체의 자연스러운 순서를 의미. 즉, 객체 자체가 알고 있는 자신의 정렬 방식
- 또한 Collections.sort() 또는 Arrays.sort() 메서드에서 자동으로 정렬할 수 있음.
- 기본 정렬방식이 아닌 특정 다른 객체의 속성을 이용해야할 경우 Comparator를 이용

Reference

https://www.geeksforgeeks.org/comparable-vs-comparator-in-java/

https://www.baeldung.com/java-comparator-comparable