<3장> 모든 객체의 공통 메서드

일자 @2024년 6월 16일

> Object는 객체를 만들 수 있는 구체 클래스지만 기본적으로는 상속해서 사용하도록 설계되었다. Object에서 final이 아닌 메소드(equals, hashCode, toString, clone, finalize)는 모두 재정의(overriding)를 염두에 두고 설계된 것이라 재정의 시 지켜야 하는 일반 규약이 명확히 정의되어 있다. 그래서 Object를 상속하는 클래스, 즉 모든 클래스는 이 메소드들을 일반 규약에 맞게 재정의해야 한다. 메소드를 잘못 구현하면 대상 클래스가 이 규약을 준수한다고 가정하는 클래스(HashMap과 HashSet 등)를 오동작하게 만들 수 있다.
>
> 이번 장에서는 final이 아닌 Object 메소드들을 언제 어떻게 재정의해야 하는지를 다룬다. 그 중 finalize 메소드는 아이템 8에서 다뤘으니 더 이상 언급하지 않는다. Comparable.compareTo의 경우 Object의 메소드는 아니지만 성격이 비슷하여 이번 장에서 함께 다룬다.

▼ [아이템 10] equals는 일반 규약을 지켜 재정의하라

equals를 재정의하지 않는 것이 최선인 경우

1. 각 인스턴스가 본질적으로 고유한 경우

- 값을 표현하는 것이 아니라 동작하는 개체를 표현하는 클래스
- ex) Thread

2. 인스턴스의 '논리적 동치성(logical equality)'을 검사할 일이 없는 경우

- 논리적 동치성을 검사하는 경우로는 java.util.regex.Pattern이 있다.

3. 상위 클래스에서 재정의한 equals가 하위 클래스에도 딱 들어맞는 경우

4. 클래스가 private이거나 package-private이고 equals 메소드를 호출할 일이 없는 경우

- 확실하게 하고싶다면 아래처럼 막아두자!

```
@Override
public boolean equals(Object obj){
        throw new AssertionError();
}
```

equals를 재정의해야 할 상황

- 객체 식별성(object identity; 두 객체가 물리적으로 같은가)이 아닌 논리적 동치성을 확인해야 하는데, 상위 클래스의 equals가 논리적 동치성을 비교하도록 재정의되지 않았을 경우
    - 값 클래스(Integer, String 등 값을 표현하는 클래스)가 해당
    - 값 클래스라도 인스턴스 통제 클래스(아이템 1)이라면 equals 재정의 필요 X
    - Enum(아이템 34)도 해당

equals 메소드가 따라야 할 일반 규약

📕 **Object 명세에 적힌 규약**

- 반사성(reflexivity): null이 아닌 모든 참조 값 x에 대해, x.equals(x)는 true다
- 대칭성(symmetry): null이 아닌 모든 참조 값 x, y에 대해, x.equals(y)가 true면 y.equals(x)도 true다
- 추이성(transitivity): null이 아닌 모든 참조 값 x, y, z에 대해, x.equals(y)가 true이고 y.equals(z)가 true면 x.equals(z)도 true다
- 일관성(consistency): null이 아닌 모든 참조 값 x, y에 대해 x.equals(y)를 반복해서 호출하면 항상 true이거나 항상 false를 반환해야 한다
- null-아님: null이 아닌 모든 참조 값 x에 대해, x.equals(null)은 false다

equals 메소드 구현 방법

1. == 연산자를 사용해 입력이 자기 자신의 참조인지 확인 (맞다면 true 반환)
2. instanceof 연산자로 입력이 올바른 타입인지 확인 (아니라면 false 반환)
3. 입력을 올바른 타입으로 형변환
4. 입력 객체와 자기 자신의 대응되는 '핵심' 필드들이 모두 일치하는지 하나씩 검사

```
@Override
public boolean equals(Object obj){
        if (this == obj) return true;
        if (!(obj instanceof ThisClass)) return false;
        ThisClass tc = (ThisClass)obj;
        return tc.member1 == this.member1 && tc.member2 ==
}
```

⚠️ equals를 다 구현했다면 세 가지만 자문해보자. 대칭적인가? 추이성이 있는가? 일관적인가?

equals 주의사항

- equals를 재정의할 땐 hashCode도 반드시 재정의하자(아이템 11)
- 너무 복잡하게 해결하려 들지 말자
  - 필드들의 동치성만 검사해도 equals 규약을 지킬 수 있다
  - 일반적으로 별칭(alias)은 비교하지 않는게 좋다
- Object 외의 타입을 매개변수로 받는 equals 메소드는 선언하지 말자
  - 그렇게 되면 재정의(Override)가 아니라 다중정의(Overload)다
  - `@Override` 어노테이션으로 이를 방지하자

▼ [아이템 11] equals를 재정의하려거든 hashCode도 재정의하라

- equals를 재정의한 클래스 모두에서 hashCode도 재정의해야 한다

Object 명세에서 발췌된 규약

- equals 비교에 사용되는 정보가 변경되지 않았다면, 어플리케이션이 실행되는 동안 그 객체의 hashCode 메소드는 몇 번을 호출해도 일관되게 항상 같은 값을 반환해야 한다. (단, 어플리케이션을 다시 실행한다면 이 값이 달라져도 상관 없다)
- **equals(Object)가 두 객체를 같다고 판단했다면 두 객체의 hashCode는 똑같은 값을 반환해야 한다.**
- equals(Object)가 두 객체를 다르다고 판단했더라도, 두 객체의 hashCode가 서로 다른 값을 반환할 필요는 없다. (단, 다른 객체에 대해서는 다른 값을 반환해야 해시테이블의 성능이 좋아진다)

좋은 hashCode를 작성하는 간단한 요령

1. int 변수 result를 선언한 후 값 c로 초기화한다. 이 때 c는 객체의 첫 번째 핵심 필드를 단계 2.a 방식으로 계산한 해시코드다 (여기서 핵심 필드란 equals 비교에 사용되는 필드를 말한다. 아이템 10 참조)
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 Type은 해당 기본 타입의 박싱 클래스이다.

ii. 참조 타입 필드면서 이 클래스의 equals 메소드가 이 필드의 equals를 재귀적으로 호출해 비교한다면, 이 필드의 hashCode를 재귀적으로 호출한다. 계산이 더 복잡해질 것 같으면, 이 필드의 표준형(acanonical representation)을 만들어 그 표준형의 hashCode를 호출한다. 필드의 값이 null이면 0을 사용한다. (다른 상수도 괜찮지만 전통적으로 0을 사용한다)

iii. 필드가 배열이라면, 핵심 원소 각각을 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(0을 추천한다)를 사용한다. 모든 원소가 핵심 원소라면 **Arrays.hashCode**를 사용한다.

b. 단계 2.a에서 계산한 해시코드 c로 result를 갱신한다. 코드로는 다음과 같다.

```
result = 31 * result + c;
```

3. result를 반환한다

단계 2.b의 곱셈 `31 * result`는 필드를 곱하는 순서에 따라 result 값이 달라지게 한다.

곱할 숫자를 31로 정한 이유는, 홀수이면서 소수이기 때문이다.
(짝수이고 오버플로가 발생한다면 정보를 잃을 수 있다. 소수를 사용하는 이유는 명확치는 않지만 전통적으로 그래왔다.)

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

```
//Object.hash를 이용한 방법. 속도는 비교적 느리다
@Override
public int hashCode{
        return Objects.hash(lineNum, prefix, areaCode);
}
```

hashCode 주의사항

- 성능을 높인답시고 해시코드를 계산할 때 핵심 필드를 생략해서는 안된다.
  - 오히려 해시 품질이 나빠져 해시 테이블의 성능을 심각하게 떨어뜨릴 수 있다.
- hashCode가 반환하는 값의 생성 규칙을 API 사용자에게 자세히 공표하지 말자.
  - 그래야 클라이언트가 이 값에 의지하지 않게 되고, 추후에 계산 방식을 바꿀 수도 있다.
  - ~~String과 Integer 등 자바의 많은 라이브러리는 공표해버렸지만 바꾸기엔 아마 늦었다..~~

▼ [아이템 12] toString을 항상 재정의하라

1. toString을 잘 구현한 클래스는 사용하기에 훨씬 즐겁고, 그 클래스를 사용한 시스템은 디버깅하기 쉽다

- toString 메소드는 개게를 println, printf, 문자열 연결 연산자(+), assert 구문에 넘길 때 자동으로 호출된다. 그러므로 잘 정의해놓자!

2. 실전에서 toString은 그 객체가 가진 주요 정보 모두를 반환하는 게 좋다

3. 포맷을 명시하든 아니든 여러분의 의도는 명확히 밝혀야 한다

```
/**
 * 이 전화번호의 문자열 표현을 반환한다.
 * 이 문자열은 "XXX-YYY-ZZZ" 형태의 12글자로 구성된다.
 * XXX는 지역 코드, YYY는 프리픽스, ZZZZ는 가입자 번호다.
 * 각각의 대문자는 10진수 숫자 하나를 나타낸다.
 *
 * 전화번호의 각 부분의 값이 너무 작아서 자릿수를 채울 수 없다면,
 * 앞에서부터 0으로 채워나간다. 예컨대 가입자 번호가 123이라면
```

```
 * 전화번호의 마지막 네 문자는 0123이 된다.
 */
@Override
public String toString(){
        return String.format("%03d-%03d-%04d", areaCode, pr
}
```

```
/**
 * 이 약물에 관한 대략적인 설명을 반환한다.
 * 다음은 이 설명의 일반적인 형태이나,
 * 상세 형식은 정해지지 않았으며 향후 변경될 수 있다.
 *
 * "[약물 #9: 유형=사랑, 냄새=테레빈유, 겉모습=먹물]"
 */
@Override
public String toString(){ ... }
```

4. toString이 반환된 값에 포함된 정보를 얻어올 수 있는 API를 제공하자

- 그렇지 않으면 사용자는 toString을 파싱할 수 밖에 없으므로, 성능이 나빠지고 불필요한 작업이 생긴다. 향후 포맷이 변경될 경우 시스템이 망가질 수 있다.

▼ [아이템 13] clone 재정의는 주의해서 진행하라

Cloneable 인터페이스는 복제 가능한 클래스임을 명시하는 용도이나, clone 메소드는 Cloneable 인터페이스가 아닌 Object에서 protected로 선언된다.

Cloneable은 단지 Object의 clone 메소드 동작 방식을 결정한다. Cloneable을 구현한 클래스의 인스턴스에서 clone을 호출하면 그 객체의 필드들을 하나하나 복사한 객체를 반환하며, 그렇지 않다면 CloneNotSupportedException을 던진다. 이는 인터페이스를 이례적으로 사용한 경우이며, 따라하지는 말자.

- 모든 필드가 기본 타입이거나 불변 객체를 참조한다면 더 손볼 것이 없다. 또한 불변 클래스는 불필요한 복사를 지양하므로, 굳이 clone 메소드를 제공하지 않는 것이 좋다. 이 점을 고려해 다음과 같이 clone 메소드를 구현할 수 있다.

```
@Override
public PhoneNumber clone() {
        try {
                return (PhoneNumber) super.clone();
        } catch (CloneNotSupportedException e){
                throw new AssertionError(); //일어날 수 없
        }
}
```

- clone 메소드는 사실상 생성자와 같은 효과를 낸다. 즉, clone은 원본 객체에 아무런 해를 끼치지 않는 동시에 복제된 객체의 불변식을 보장해야 한다.
    - Stack의 clone 메소드는 제대로 동작하려면 스택 내부 정보를 복사해야 하는데, 가장 쉬운 방법은 elements 배열의 clone을 재귀적으로 호출해 주는 것이다.

```
@Override
public Stack clone() {
        try {
                Stack result = (Stack) super.clone();
                result.elements = elements.clone();
                return result;
        } catch (CloneNotSupportedException e) {
                throw new AssertionError();
        }
}
```

- **한편, 새로운 값을 할당할 수 없는 final 필드는 위 방식이 적용되지 않으므로, Cloneable 아키텍처는 '가변 객체를 참조하는 필드는 final로 선언하라'는 일반 용법과 충돌한다.**(단, 원본과 복제된 객체가 그 가변 객체를 공유해도 안전하다면 괜찮다)
    - 따라서 복제할 수 있는 클래스를 만들기 위해 일부 필드에서 final 한정자를 제거해야 할 수도 있다
- public인 clone 메소드에서는 throws 절을 없애야 한다

    - 그래야 사용하기 편하기 때문이다
- 상속해서 쓰기 위한 클래스 설계 방식 두 가지(아이템 19) 중 어느 쪽에서든, 상속용 클래스는 Cloneable을 구현해서는 안된다.
    - Object의 clone 메소드를 모방하는 방법도 있다

```
@Override
protected final Object clone() throws CloneNotSupport
        throw new CloneNotSupportedException();
}
```

- Cloneable을 구현한 스레드 안전 클래스를 작성할 때는 clone 메소드 역시 적절히 동기화해주어야 한다(아이템 78)

✅ **요약**

- Cloneable을 구현하는 모든 클래스는 clone을 재정의해야 한다
    - 이 때 접근 제한자는 public으로, 반환 타입은 클래스 자신으로 변경한다
- 이 메소드는 먼저 super.clone을 호출한 후 필요한 필드를 전부 적절히 수정한다
    - 객체 내부 '깊은 구조'에 숨어 있는 모든 가변 객체르 ㄹ복사하고, 복제본이 가진 객체 참조 모두가 복사된 객체들을 가리키게 해야 한다
    - 이러한 내부 복사는 주로 clone을 재귀적으로 호출해 구현하지만, 이 방식이 항상 최선인 것은 아니다. 기본 필드나 불변 객체 참조만 갖는 클래스라면, 아무 필드도 수정할 필요 없다 (단, 일련번호나 고유ID는 비록 기본타입/불변객체일 지라도 수정해주어야 한다)

☝🏻 이처럼 복잡한 경우는 드물다. Cloneable을 이미 구현한 클래스가 아니라면, **복사 생성자**와 **복사 팩토리**라는 더 나은 객체 복사 방식을 제공할 수 있다.

```
//복사 생성자
public Yum(Yum yum) { ... }

//복사 팩토리
public static Yum newInstance(Yum yum) { ... }
```

관례상 모든 범용 컬렉션 구현체는 Collection이나 Map 타입을 받는 생성자를 제공한다

```
// HashSet 객체를 TreeSet 타입으로 복제
HashSet<Object> hs = new HashSet<>();
TreeSet<Object> ts = new TreeSet<>(hs);
```

주의사항

- 새로운 인터페이스를 만들 때에는 절대 Cloneable을 extends 해서는 안되며, 새로운 클래스도 이를 implements 해서도 안된다.
- final 클래스라면 Cloneable을 구현해도 위험이 크지 않지만, 성능 최적화 관점에서 검토 후 별다른 문제가 없을 때만 드물게 허용해야 한다(아이템 67)
- **기본 원칙은 '복제 기능은 생성자와 팩토리를 이용하는게 최고'라는 것이다**
- **단, 배열만은 clone 메소드 방식이 가장 깔끔한, 이 규칙의 합당한 예외라 할 수 있다.**

▼ [아이템 14] Comparable을 구현할지 고려하라

⚠️ compareTo는 Object의 메소드는 아니지만, 두 가지 성격만 제외하면 equals와 같다.

1. compareTo는 단순 동치성 비교에 더해 순서까지 비교할 수 있다
2. 제네릭(Generic) 하다

compareTo 메소드의 일반 규약

(equals의 규약과 비슷하다.)

이 객체와 주어진 객체의 순서를 비교한다. 이 객체가 주어진 객체보다 작으면 음의 정수를, 같으면 0을, 크면 양의 정수를 반환한다. 이 객체와 비교할 수 없는 타입의 객체가 주어지면 ClassCastException을 던진다.

다음 설명에서 sgn(표현식) 표기는 수학에서 말하는 부호 함수(signum function)를 뜻하며, 표현식의 값이 음수, 0, 양수일 때 -1, 0, 1을 반환하도록 정의했다.

- Comparable을 구현한 클래스는 모든 x, y에 대해 sgn(x.compareTo(y)) == -sgn(y.compareTo(x))여야 한다(따라서 x.compareTo(y)는 y.compareTo(x)가 예외를 던질 때에 한해 예외를 던져야 한다).
- Comparable을 구현한 클래스는 추이성을 보장해야 한다. 즉, (x.compareTo(y) > 0 && y.compareTo(z) > 0)이면 x.compareTo(z) > 0이다.
- Comparable을 구현한 클래스는 모든 z에 대해 x.compareTo(y) == 0이면 sgn(x.compareTo(z)) == sgn(y.compareTo(z))다.
- 이번 권고가 필수는 아니지만 꼭 지키는 게 좋다. (x.compareTo(y) == 0) == (x.equals(y))여야 한다. Comparable을 구현하고 이 권고를 지키지 않는 모든 클래스는그 사실을 명시해야 한다. 다음과 같이 명시하면 적당할 것이다.

“주의: 이 클래스의 순서는 equals 메서드와 일관되지 않다.”

여러가지 비교자

1. 객체 참조 필드가 하나뿐인 비교자

- 자바가 제공하는 비교자 사용

```
public final class CaseInsentiveString implements Comparabl
```

```
        @Override
        public int compareTo(CaseInsentiveString cis) {
                return String.CASE_INSENTIVE_ORDER.compare(
        }
        ...
}
```

⚠ compareTo 메소드에서 관계 연산자 < 와 > 를 사용하는 이전 방식은 거추장스럽고 오류를 유발하니, 이제는 추천되지 않는다

2. 기본 타입 필드가 여럿일 때의 비교자

```
public int compareTo(PhoneNumber pn) {
        int result = Short.compare(areaCode, pn.areaCode);

        if (result == 0){
                result = Short.compare(prefix, pn.prefix);
                if (result == 0)
                        result = Short.compare(lineNum, pn.
        }
        return result;
}
```

3. 비교자 생성 메소드를 활용한 비교자

- 자바8에서는 연쇄 방식으로 비교자를 생성할 수 있게 되어 코드가 깔끔해졌으나, 약간의 성능 저하가 뒤따른다
- 자바의 정적 임포트 기능을 이용하면 정적 비교자 생성 메소드들을 그 이름만으로 활용할 수 있어 코드가 훨씬 깔끔해진다

```
import static java.util.Comparator.comparingInt;
...

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

```
                .thenComparingInt(pn -> pn.lineNum);

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

4-1. 해시코드 값의 차를 기준으로 하는 비교자 - 추이성 위배!

```
static Comparator<Object> hashCodeOrder = new Comparator<>(
        public int compare(Object o1, Obeject o2){
                return o1.hashCode() - o2.hashCode();
        }
}
```

- 이 방식은 사용하면 안된다
  - 정수 오버플로우를 야기하거나 IEEE754 부동소수점 계산 방식에 따른 오류를 낼 수 있다
  - 이번 아이템에서 설명한 방법대로 구현한 코드보다 월등히 빠르지도 않을 것이다
- 따라서 다음의 두 방식 중 하나를 사용하자

4-2. 정적 compare 메소드를 활용한 비교자

```
static Comparator<Object> hashCodeOrder = new Comparater<>(
        public int compare(Object o1, Object o2) {
                return Integer.compare(o1.hashCode(), o2.ha
        }
};
```

4-3. 비교자 생성 메소드를 활용한 비교자

```
static Comparator<Object> hashCodeOrder = Comparator.compar
```

✅ **요약**

- 순서를 고려해야 하는 값 클래스를 작성한다면 꼭 Comparable 인터페이스를 구현해야 한다.
- compareTo 메소드에서 필드와 값을 비교할 때 <와 > 연산자는 쓰지 말도록 해야 한다
    - 그 대신 박싱된 기본 타입 클래스가 제공하는 정적 compare 메소드나 Comparator 인터페이스가 제공하는 비교자 생성 메소드를 사용하자.