3장 모든 객체의 공통 메서드

아이템 10. equals는 일반 규칙을 지켜 재정의하라

Object: 자바의 최상위 부모 클래스

- 필드가 없고 메소드로 구성되어 있다.

equals 메서드

- 보통 논리적 동치성, 즉 String이나 Integer처럼 값 클래스 들의 주솟값이 아닌 값을 비교하기 위해 재정의 해서 사용한다.
- 만약 equals를 재정의 하지 않는다면 상위 클래스의 equals를 사용하게 되고 끝까지 올라간다면 Object의 equals를 사용하게 된다.
    - Object의 equals의 경우 단순히 자기 자신과 같은 지를 비교한다.

equals 메서드 예시

1. String 타입의 변수를 비교

```
String s1 = "Hello";
String s2 = "Hello";

System.out.println(s1 == s2); //주소 비교 false
System.out.println(s1.equals(s2)); // 값 비교 true
```

2.문자열이 아닌 클래스 자료형의 객체 데이터일 경우

- 비교할 대상이 객체일 경우 **객체의 주소**를 이용(`==` 이나 `equals()`와 똑같다.)

```
class Person {
    String name;

public Person(String name) {
    this.name = name;
    }

public class Example {
    public static void main(String[]args){
        Person person1 = new Person("홍길동");
```

```
        Person person2 = new Person("홍길동");

        System.out.println(person1 == person2); // == 은 객체타

        System.out.println(person1.equals(person2)) // equals
    }
}
```

하지만 사용자 입장에서는 두 객체는 같은 데이터라고 볼 수 있다. 객체 자료형을 비교할 때 주소 값이 아닌 객체의 필드 값을 기준으로 비교하고 싶다면, equals 메서드를 오버라이딩 해서 필드 값을 비교하도록 재정의 해야한다.

```
import java.util.Objects;

class Person {
    String name;

    public Person(String name) {
        this.name = name;
    }

    // 객체의 사람 이름이 동등한지 비교로 재정의 하기 위해 오버라이딩
    public boolean equals(Object o) {
        if (this == o) return true; //  객체 this와 매개변수 객체
        if (!(o instanceof Person)) return false; // 매개변수 객
        Person person = (Person) o; // 매개변수 객체가 Person 타입
        return Objects.equals(this.name, person.name); // thi
    }
}

public class Main {
    public static void main(String[] args) {
        Person p1 = new Person("홍길동");
        Person p2 = new Person("홍길동"); // 동명이인

        System.out.println(p1.equals(p2)); // true
    }
}
```

equals 재정의를 사용하지 않아도 되는 경우

- 각 인스턴스가 본질적으로 고유하다.
  - 값을 표현하는 것이 아니라 동작하는 개체를 표현하는 클래스가 해당된다.
  - ex) thread
    - thread란? 프로세스 안에서 실질적으로 작업을 실행하는 단위를 의미.
- 논리적 동치성(logical equality)를 검사할 일이 없는 경우
  - 논리적 동치성: 참조타입 변수를 비교하는 것
- 상위 클래스에서 재정의한 equals가 하위 클래스에도 적합한 경우
  - Set 인터페이스는 AbstractSet이 구현한 equals를 사용하는 것을 의미한다.
- 클래스가 private하거나 defalult인 경우

equals 재정의를 사용하는 경우

- 값을 비교하기 위해 주솟값 비교가 아니라 논리적 동치성을 확인할 때를 의미
- 싱글톤과 같은 인스턴스 통제 클래스 이거나, Enum의 같은 경우 매번 다른 인스턴스를 생성하기 때문에 재정의를 하지 않아도 된다.

equals 재정의 규약

- **반사성(reflexivity)**: null이 아닌 모든 참조값 x에 대해 x.equals(x)는 true이다.
- **대칭성(summetry)**:null이 아닌 모든 참조값 x,y에 대해 x.equals(y)가 true이면 y.equals(x)도 true이다.
- **추이성(transitivity)**:null이 아닌 모든 참조 값 x,y,z에 대해 x.equals(y)가 true이고 y.equals(z)도 true이면 x.equals(z)도 true이다.
- **일관성(consistency)**:null이 아닌 모든 참조 값 x,y에 대해 x.equals(y)를 반복해서 호출하면 항상 같은 값을 반환한다.
- **null-아님**: null이 아닌 모든 참조 값 x에 대해 x.equals(null)은 항상 false다.

대칭성

```
import java.util.Objects;
```

```
public class EqualsTest {
  public static void main(String[] args) {
    CaseInsensitiveString cis = new CaseInsensitiveString("a"
    String s = "a";
    System.out.println(cis.equals(s)); // true
    System.out.println(s.equals(cis)); // false
  } //대칭성 위배 String의 equals는 CaseInsentiveString을 몰라.
}

final class CaseInsensitiveString {
  private final String s;

  public CaseInsensitiveString(String s) {
    this.s = Objects.requireNonNull(s);// 대소문자 구분 없이 비교하
  }

  @Override
  public boolean equals(Object o) {
    if (o instanceof CaseInsensitiveString)// 해당 인스턴스에 대하
      return s.equalsIgnoreCase(((CaseInsensitiveString) o).s
    if (o instanceof String)
      return s.equalsIgnoreCase((String) o);// 이 부분에서 문제 -
    return false;
  }
}
```

추이성

```
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;
  }
}

class ColorPoint extends Point { //Point 클래스를 상속
  private final Color color;

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

enum Color {
  RED, BLUE, GREEN
}

class PointTest {
  public static void main(String[] args) {
    ColorPoint cp = new ColorPoint(1,1,Color.RED);
    ColorPoint cp2 = new ColorPoint(1,1,Color.BLUE);
    System.out.println(cp.equals(cp2)); // true 출력,대칭성 위배
  }
}
```

ColotPoint는 Point클래스를 상속 받았지만 Color에 대한 비교가 이루어 지지 않았다.

- color 비교 추가

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

```
  public ColorPoint(int x, int y, 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
  }
}
```

```
ColorPoint cp1 = new ColorPoint(1, 1, Color.RED);
Point p2 = new Point(1, 1);
System.out.println(cp1.equals(p2)); // false
System.out.println(p2.equals(cp1)); // true
```

만약 이렇게 비교한다면 어떤 결과가 나올까?

- cp1과 p2를 비교 = cp1 에서는 좌표 비교 후 color까지 비교 p는 color 객체가 없으므로 false 반환
- p2와 cp1 비교 = 상위 클래스인 p2에서는 좌표만 비교하므로 true 반환

구체 클래스를 확장해 새로운 값을 추가하면서 equals 규약을 만족시킬 방법은 존재하지 않는다.

리스코프 치환 원칙(LIscov Substitution Principle)

- 리스코프 치환 원칙이란 부모 객체와 자식 객체가 있을 때 부모 객체를 호출하는 동작에서 자식 객체가 부모 객체를 완전히 대체할 수 있다는 원칙

리스코프 치환 원칙은 상속되는 객체는 반드시 부모 객체를 완전히 대체할 수 있어야한다고 권고한다.

하지만 우회할 수 있는 방법이 한가지 존재한다.

상속이 아닌 컴포지션 관계를 사용하는 것이다.

컴표지션 관계란?

- 컴포지션은 객체가 다른 객체를 **포함**하여 그 객체의 기능을 사용하는 방법

```
import java.util.Objects;

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 ColorPoint) {
            Point p = ((ColorPoint) o).asPoint();
            return p.x == x && p.y == y;
        }
        if (!(o instanceof  Point))
            return false;

        Point p = (Point) o;
        return p.x == x && p.y == y;
    }
}

class ColorPoint{
    private final Point point; //ColorPoint는 Point를 포함(컴포지
    private final Color color;

    public ColorPoint(int x, int y, Color color) {
        point = new Point(x,y);// Point 객체 생성 및 포함
        this.color = Objects.requireNonNull(color);
    }
```

```
    // ColorPoint 의 Point 뷰를 반환한다.
    public Point asPoint() {
        return point;//ColorPoint를 구성하는 Point를 반환
    }

    @Override
    public boolean equals(Object o) {
        if (o instanceof ColorPoint) {
            ColorPoint cp = (ColorPoint) o;
            return cp.point.equals(point) && cp.color.equals(
        }// 입력 객체가 ColorPoint라면 좌표와 색상 모두 비교
        else if (o instanceof Point) {
            Point p = (Point) o;
            return p.equals(point);
        }
        return false;
    }
}

enum Color {
    RED, BLUE, GREEN
}

class PointTest {
    public static void main(String[] args) {
        ColorPoint cp1 = new ColorPoint(1, 1, Color.RED);
        Point p2 = new Point(1, 1);
        ColorPoint cp3 = new ColorPoint(1, 1, Color.RED);
        System.out.println(cp1.equals(p2)); // true
        System.out.println(p2.equals(cp3)); // true
        System.out.println(cp1.equals(cp3)); // true
    }
}
```

컴포지션을 통해 비교간 발생할 수 있는 상속문제를 해결하고 있다. 상속 받는 대신 Point 객체를 내부에 포함하여 사용함으로써 equals 메서드에서 상속에 의한 대칭성 문제와 추이성 문제를 방지할 수 있다.

올바른 equals 메서드 구현 방법

```
public final class PhoneNumber {
  private final short areaCode, prefix, lineNum;

  public PhoneNumber(...){...}

  @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 &&
    }
}
```

- == 연산자를 사용하여 입력이 자기 자신의 참조인지 확인한다.
- instanceof 연산자로 입력이 올바른 타입인지 확인한다.
- 입력을 올바른 타입으로 형변환 한다.
- 입력 객체와 자기 자신의 대응되는 핵심 필드들이 모두 일치하는지 검사한다.
- equals를 다 구현 했다면 대칭성,추이성, 일관성 이 세가지를 자문하고 단위테스트를 통해 검증을 하자.
- 꼭 필요한 경우가 아니라면 재정의하지 말자.재정의할 때는 hashcode도 반드시 재정의!!!

아이템11 . equals를 재정의하려거든 hashCode도 재정의하라.

논리적으로 같은 객체는 같은 해시코드를 반환해야하기 때문에 `equals` 재정의 후 `hashCode`를 재정의 해야한다.

hashCode란?

- 객체의 주소값을 변환하여 생성한 객체의 고유한 정수값이다.

hash란?

- 주어진 키에 대해 hash값을 계산하고, 내부적으로 이 값을 사용하여 데이터를 저장하여 빠르게 데이터를 검색할수 있게 하는 자료구조이다.

hashCode 일반 규약

- equals 비교에 사용되는 정보가 변경되지 않았다면, hashCode도 변하면 안된다.
- equals가 두 객체가 같다고 판단했다면, 두 객체의 hashcode는 다 똑같은 값을 반환한다.
- equals가 두 객체를 다르다고 판단했더라도, hashCode는 꼭 다를 필요는 없다.

논리적으로 같은 객체는 같은 해시코드를 반환해야한다.

- equals가 물리적으로 다른 두 객체를 논리적으로 같다고 한다면, hashCode는 서로 다른 값을 반환한다.

```
Map<PhoneNumber,String> m = new HashMap<>();
map.put(new PhoneNumber(010,1234,5678), "제니");
```

다음 코드에 m.get(new PhoneNumber(010,1234,5678))을 실행하면 “제니”가 나와야 할 것 같지만 `null`을 반환한다.

→ PhoneNumber 클래스는 hashCode를 재정의하지 않았기 때문에, 논리적 동치인 두 객체가 서로 다른 해시코드를 반환하여 get메서드는 엉뚱한 해시 버킷에 가서 객체를 찾으려고 한 것이다.

- HashMap은 해시코드가 서로 다른 엔트리끼리는 동치성 비교를 시도조차 않도록 최적화 되어있다.

동치인 모든 객체에서 똑같은 hashCode를 반환하는 코드

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

모든 객체에게 같은 값만 내어주므로 모든 객체가 해시테이블의 버킷 하나에 담겨 마치 연결리스트 처럼 동작한다. 그 결과 평균 수행시간이 느려져서 도저히 쓸 수가 없게 된다.

좋은 해시 함수라면 서로 다른 인스턴스에 다른 해시코드를 반환한다.

좋은 hash코드를 작성하는 요령이란?

1. int 변수인 result를 선언한 후 값 c로 초기화 한다.
    - 이 때, c는 해당 객체의 첫번째 핵심 필드를 단계 2.a 방식으로 계산한 해시코드다.
    - 여기서 핵심필드란 equals 비교에 사용되는 필드를 말한다.
2. 해당 객체의 나머지 핵심 필드 f 각각에 대해 다음 작업을 수행한다.

    a.해당 필드의 해시코드 c를 계산한다.

        - 기본 타입 필드라면, Type.hashCode(f)를 수행한다. 여기서 Type은 해당 기본 타입의 박싱 클래스다.
        - 참조 타입 필드면서, 이 클래스의 equals 메소드가 이 필드의 equals를 재귀적으로 호출하여 비교한면, 이 필드의 hashCode를 재귀적으로 호출한다.
        - 필드가 배열이라면, 핵심 원소 각각을 별도 필드처럼 다룬다.

    b. 단계 2.a에서 계산한 해시코드 c로 result를 갱신한다.

        - result = 31 * result + c;

3. result를 반환한다.

주의할 점

- equals 비교에 사용되는 필드에 대해서만 해시코드를 계산한다.
- 성능을 높인답시고 해시코드를 계산할 때 핵심 필드를 생략해서는 안된다.
- 만약 hash로 바꾸려는 필드가 기본 타입이 아니면 해당 필드의 hashCode를 불러 구현한다.
- 참조 타입 필드가 null일 경우 0 을 사용(다른 상수도 괜찮지만 전통적으로 0을 사용)

31을 곱하는 이유?

- 31은 소수이면서 홀수이기 때문이다.만일 그 값이 짝수였고, 곱셈 결과가 오버플로(표현 불가로 인한 쓰레기 값으로 반환)되었으면 정보는 사라졌을 것이다. 2로 곱하는 것은 비

트를 왼쪽으로 shift하는 것(*2)과 같기 때문이다. 소수를 사용하는 이점은 그다지 분명하지 않지만 전통적으로 널리 사용된다.

hashCode의 캐싱과 지연 초기화

- 클래스가 불변이고 해시코드를 계산하는 비용이 크다며느, 매번 새로 계산하기 보다 캐싱을 고려해야 한다.

  →해당 타입의 객체가 주로 해시의 키로 사용될 것 같다면 인스턴스가 만들어질 때 계산해야한다.

- 해시의 키로 사용되지 않는 경우라면 hashCode가 처음 불릴 때 계산하는 지연 초기화 전략이 좋다.
  - 지연 초기화?
    - 필드의 초기화 시점을 그 값이 처음 필요할 때까지 늦추는 기법, 값이 전혀 쓰이지 않으면 초기화도 결코 일어나지 않는다.
    - 클래스와 인스턴스 초기화할 때 발생하는 위험한 순환 문제를 해결하는 효과가 있다.

```
private int hashCode;

@Override
public int hashCode() {
    int result = hashCode; // 저장된 해시코드를 result에 할당
    if (result == 0) {     // hashCode가 아직 계산되지 않았을 때만
        result = Integer.hashCode(areaCode);
        result = 31 * result + Integer.hashCode(areaCode);
        result = 31 * result + Integer.hashCode(areaCode);
        hashCode = result; // 계산한 해시코드를 저장
    }
    return result; // 저장된 값 반환
}
```

아이템 12. toString을 항상 재정의하라.

기본적으로 Object 클래스의 toStrint() 메서드는 해당 인스턴스에 대한 정보를 문자열로 반환한다. 이 메서드는 인스턴스에 대한 정보를 문자열로 제공할 목적으로 정의되어 있는 것이다.

- toString()의 내부 구조

```
public String toString() {
    return getClass().getName()+"@"+Integer.toHexString(hashC
                //클래스설계도   클래스이름.   위치          해시코드(객체
```

객체를 출력할 때 toString()을 붙이지 않고 변수만 출력해도 컴파일러가 객체만 출력할 경우 자동으로 붙여주고 컴파일하기 때문에 toString()을 붙여준 것과 같은 값이 출력된다.

toString() 포맷의 문서화

- toString()을 구현할 때, 반환값의 포맷을 문서화할지 정해야한다. 전화번호나 행렬 같은 값 클래스라면 문서화 하는 것이 좋다.
- 포맷을 명시하면 . 그 객체는 표준적이고,명확하고 사람이 읽을 수 있게 된다. 따라서 그 값을 그대로 사용하거나 CSV 파일 처럼 사람이 읽을 수 있는 데이터 객체로 저장할 수도 있다.
- 포맷을 명시하기로 했다면, 명시한 포맷에 맞는 문자열과 객체를 상호 전환할 수 있는 정적 팩토리나 생성자를 함께 제공해주는 것이 좋다.

ex)

```
/**
 * 이 전화번호의 문자열 표현을 반환한다.
 * 이 문자열은 "XXX-YYY-ZZZZ" 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. 예를 들어 가입자 번호가 123이라면
 * 전화번호의 마지막 네 문자는 "0123"이다.
 */
@Override
public String toString() {
    return String.format("%03d-%03d-%04d",
```

```
            areaCode, prefix, lineNum);
    }
```

포맷을 명시하지 않기로 한 예시

```
/**
 * 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() { ... }
```

아이템 13. clone 재정의는 주의해서 진행하라

`Object.clone()` 메소드는 인스턴스 객체의 복제를 위한 메소드로, 해당 인스턴스를 복제하여 새로운 인스턴스를 생성해 그 참조값을 반환한다.

clone()메서드는 Cloneable인터페이스가 아닌 Object 클래스에 선언이 되어있고, 접근 지정자는 protected로 선언이 되어있다.

메서드가 하나도 선언되어 있지 않은 Cloneable 인터페이스는 Objectdml protected 메서드인 clone의 동작 방식을 결정한다. Cloneable을 구현한 클래스의 인스턴스에서 clone()을 호출하면 그 객체의 필드들을 하나하나 복사한 객체를 반환하며, 그렇지 않은 클래스의 인스턴스에서 호출하면 CloneNotSupportedException을 던진다.

```
import java.lang.Cloneable;

// 객체 복사 메소드를 사용하기 위해 Cloneable 인터페이스를 구현해서 clor
class Person implements Cloneable {
    ..

    // clone 메서드를 오버라이딩
    public Object clone() throws CloneNotSupportedException {
```

```
        return super.clone(); // 기본적으로 부모의 clone을 그대로 불
    }
}

public class Main {
    public static void main(String[] args) {
        try {
            Person p = new Person("홍길동", 11);
            Person p_copy = (Person) p.clone();
        } catch(Exception e) {}
    }
}
```

얕은 복사 vs 깊은 복사

- **얕은 복사** : 객체 자체는 복사하지만, 그 객체가 참조하고 있는 다른 객체는 복사하지 않는다. 얕은 복사에서 원본을 변경한다면 복사본도 영향을 받는다.
- **깊은 복사**: 객체와 그 객체가 참조하고 있는 다른 객체도 모두 복사한다. 깊은 복사에서는 원본과 복사본이 서로 다른 객체를 참조하기 때문에 원본의 변경이 복사본에 영향을 미치지 않는다.

기본적으로 java의 Object 클래스에 있는 clone() 메서드는 얕은 복사를 수행한다. 이 경우에는 원본 객체와 복사된 객체가 동일한 배열이나 참조형 데이터를 공유할 수 있다.

clone() 메서드에 대한 Object 명세

- 이 객체의 복사본을 생성해 반환한다. 복사의 정확한 뜻은 그 객체를 구현한 클래스에 따라 다를 수 있지만 일반적인 의도는 다음과 같다. 어떤 객체 x에 대해 다음 식은 참이다.

```
x.clone() != x
```

```
x.clone().getClass() == x.getClass()
```

```
x.clone().equals(x)
```

- 관례상, 이 메서드가 반환하는 객체는 super.clone()을 호출해 얻어야한다. 이 클래스와(Object를 제외한) 모든 상위 클래스가 이 관례를 따른다면 다음 식은 참이다.

```
x.clone().getClass() == x.getClass()
```

- 관례상, 반환된 객체와 원본 객체는 독립적이어야 한다. 이를 만족할려면 super.clone()으로 얻은 객체의 필드 중 하나를 반환 전에 수정해야할 수도 있다.

clone()을 구현하는 방법

- clone() 메서드는 super.clone()이 아닌 생성자를 호출해 얻은 인스턴스를 반환해도 컴파일러는 정상적으로 통과시킬 것이다. 하지만 하위 클래스에서 super.clone()을 호출한다면 잘못된 클래스의 객체가 만들어져 하위 클래스의 clone()메서드가 제대로 작동하지 않을 것이다.
- ex) 클래스 B가 클래스 A를 상속할 때, 하위 클래스인 B인 clone()은 B 타입 객체를 반환해야 한다. A의 clone()이 자신의 생성자로 생성한 객체를 반환한다면 B의 clone()도 A의 개체를 반환할 수 밖에 없다.
- 공변 반환 타이핑 : 부모 클래스의 반환 타입은 자식 클래스의 타입으로 변경 가능하다.

가변 클래스를 참조하는 경우의 재정의

```
import java.util.Arrays;
import java.util.EmptyStackException;

public class Stack implements Cloneable {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
        this.elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() {
```

```
        if (size == 0)
            throw new EmptyStackException();
        Object result = elements[--size];
        elements[size] = null;  //다 쓴 참조 해제
        return result;
    }

    private void ensureCapacity() {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }

    @Override
    protected Stack clone() {
        try {
            Stack result = (Stack) super.clone();//얕은 복사: s
            result.elements = elements.clone(); //깊은 복사: 배
            return result;
        } catch (CloneNotSupportedException e) {
            throw new AssertionError();
        }
    }
}
```

- 해당 코드에서는 배열을 복사하지 않고 얕은 복사만 수행하게 된다면 원본 스택과 복사된 스택이 동일한 배열을 공유하게 된다. 이렇게 되면 원본 또는 복사본의 스택에서 요소를 추가하거나 수정하게 된다면 다른 스택에도 영향을 미치게 된다.
- 만약 elements 필드에 final 한정자가 붙는다면 새로운 값을 할당할 수 없기 때문에 작동하지 않는다. **Cloneable 아키텍처는 '가변 객체를 참조하는 필드는 final로 선언하라'**라는 용법과 충돌한다.

clone()을 재귀적으로 호출하는 것만으로 충분하지 않은 경우

```
public class HashTable implements Cloneable {
    private Entry[] buckets = ...;

    private static class Entry {
```

```
        final Object key;
        Object value;
        Entry next;

        Entry(Object key, Object value, Entry next) {
            this.key = key;
            this.value = value;
            this.next = next;
        }
    }

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

- 해당 HashTable의 복제본은 자신만의 새로운 buckets 배열을 갖지만, buckets 배열은 원본과 같은 entry를 참조하게 되는 문제가 발생한다. 해결하기 위해슨 각 버킷을 구성하는 entry를 복사해야한다.

```
public class HashTable implements Cloneable {
    private Entry[] buckets = ...;

    private static class Entry {
        final Object key;
        Object value;
        Entry next;

        Entry(Object key, Object value, Entry next) {
            this.key = key;
```

```
            this.value = value;
            this.next = next;
        }

        // 이 엔트리가 가리키는 연결 리스트를 재귀적으로 복사
        Entry deepCopy() {
            return new Entry(key, value,
                next == null ? null : next.deepCopy());
    }

    @Override
    public HashTable clone() {
        try {
            HashTable result = (HashTable) super.clone();
            result.buckets = new Entry[buckets.length];
            for (int i = 0; i < buckets.length; i++)
                if (buckets[i] != null)
                    result.buckets[i] = buckets[i].deepCopy()
            return result;
        } catch (CloneNotSupportedException e) {
            throw new AssertionError();
        }
    }
    ...
}
```

- 이제 HashTable.Entry 클래스는 깊은 복사를 지원한다.
- clone 메서드는 새 배열을 할당 한 다음 각 버킷에 대한 깊은 복사를 수행한다.
- 재귀호출은 간단하고 연결리스트가 길지 않을 때는 잘 작동하지만, 리스트 원소수 만큼 스택 크기를 소비하며 리스트가 길어지면 스택 오버플로우가 발생할 위험이 있다. 유연한 사용을 위해선 반복자를 써서 순회하는 방향으로 수정하자.

Clone 메서드 사용 시 주의 사항

1. Clone 메서드에서 재정의될 수 있는 메서드는 호출하지 않는다.
2. clone 메서드에서 throws절을 없앤다.(재정의시)

3. 상속할 클래스에서는 Cloneable을 구현해서는 안된다.
4. Cloneable을 구현한 스레드 안전 클래스를 작성할 때는 clone 메서드 역시 적절히 동기화해야한다.

clone보다는 복사 생성자/팩터리를 사용하자.

- **복사 생성자**란 단순히 자신과 같은 클래스의 인스턴스를 인수로 받는 생성자를 의미한다.
- **복사 팩터리**는 복사 생성자를 모방한 정적 팩터리이다.

아이템 14. Comparable을 구현할지 고려하라.

Comparable 인터페이스란?

- Comparable 인터페이스는 객체를 정렬하는데 사용되는 메서드인 compareTo를 정의하고 있다.
- Comparable 인터페이스를 구현한 클래스는 반드시 compareTo를 정의해야 한다.

Compare 인터페이스의 특징

- 자바에서 같은 타입의 인스턴스를 비교해야만 하는 클래스들은 모두 Comparable 인터페이스를 구현하고 있다.
- Boolean 타입을 제외한 래퍼 클래스와 알파벳, 연대같이 순서가 명확한 클래스들은 모두 정렬이 가능하다.
- 기본 정렬 순서는 작은 값에서 큰 값으로 정렬되는 오름차순이다.

Comparable 인터페이스의 compareTo 메서드

compareTo는 해당 객체와 전달된 객체의 순서를 비교한다.

- `compareTo`는 Object의 `euqals`와 두가지 차이점이 있다. `compareTo`는 `equals`와 달리 **단순 동치성에 더해 순서까지 비교할 수 있으며, 제네릭하다.**
- `Comparable`을 구현했다는 것은 그 클래스의 인스턴스에 자연적인 순서가 있음을 뜻한다. 예를 들어, `Comparable`을 구현한 객체들의 배열은 `Arrays.sort(a)`로 쉽게 정렬이 가능하다.

compareTo 메서드 일반규약

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

equals와 compareTo의 차이

```
final BigDecimal bigDecimal1 = new BigDecimal("1.0");
        final BigDecimal bigDecimal2 = new BigDecimal("1.00")

        final HashSet<BigDecimal> hashSet = new HashSet<>();
        hashSet.add(bigDecimal1);
        hashSet.add(bigDecimal2);

        System.out.println(hashSet.size());

        final TreeSet<BigDecimal> treeSet = new TreeSet<>();
        treeSet.add(bigDecimal1);
        treeSet.add(bigDecimal2);

        System.out.println(treeSet.size());
```

hashSet: 2

hashSet: 1

-
- HashSet과 TreeSet은 서로 다른 메서드로 객체의 동치성을 비교한다.

- Hashset은 equals를 기반으로 비교하기 때문에 추가된 두 BigDecimal이 다른값으로 인식되어 크기가 `2`가 된다.
- 반면에, `TreeSet`은 `compareTo`를 기반으로 객체에 대한 동치성을 비교하기 때문에 같은 값으로 인식되어 `compareTo`가 `0`을 반환하기 때문에 크기가 `1`이 된다

compareTo 메서드에서 관계연산자 (<, >)를 사용하지 않는것을 추천한다

- 박싱된 기본 타입 클래스들에 새로 추가된 정적 메서드 `compare`를 대신 이용한다.
- 관계연산자(<,>)는 오류를 유발할 가능성이 있기때문에 추천하지 않는다.

클래스에 핵심필드가 여러개라면 가장 핵심적인 필드부터 비교하자.