Ch3. 모든 객체의 공통 메서드

태그	1주차

클래스가 불변이고 해시코드를 계산하는 비용이 크다면, 새로 계산하기 보다는 **캐싱하는 방식**을 고려해야 한다.

인스턴스가 만들어질때 해시코드를 미리 계산해두거나, 해시의 `Key` 로 사용되지 않는 경우라면 `hashCode` 가 처음 불릴 때 계산하는 **지연 초기화(lazy initialization)** 방식도 좋다. 필드를 지연 초기화하려면 클래스를 **스레드 안전**하게 만들어야 한다.

서론

`Object`는 개게를 만들 수 있는 구체 클래스이지만 기본적으로는 상속해서 사용하도록 설계되었다. `Object`에서 `final`이 아닌 메서드(`equals`, `hashCode`, `toString`, `clone`, `finalize`)는 모두 오버라이딩을 염두에 두고 설계된 것이라, 재정의 시 지켜야 하는 일반 규약이 명확히 정의되어있다. 이러한 메서드들을 언제 어떻게 재정의 하는지 다룬다.

Item10 equals는 일반 규약을 지켜 재정의하라

equlas 메서드를 재정의 해야하는 상황

다음과 같은 상황이라면 `equals`를 재정의하지 않아도 된다.

- 각 인스턴스가 본질적으로 고유 → 값을 표현하는 것이 아닌 동작하는 객체를 표현(ex. `Thread`)
- 인스턴스의 논리적 동치성을 검사할 일이 없음 → `Integer`나 `String`같이 값을 비교하는 것이 아니라면 재정의 하지 않아도 된다.
- 상위 클래스에서 재정의한 equals가 하위 클래스에도 맞을 때 → `Set` 구현체는 `AbstractSet`이 구현한 `equals`를 상속받아 사용
- 클래스가 `private`이거나 `package-private`이고 `equals` 메서드를 호출할 일이 없을 때 → 예외처리로 해결

객체가 같은지가 아닌, 값이 같은지 비교해야 하는 경우에 `equals`를 재정의 해서 사용하자.

equals 메서드의 동치관계

x,y,z 모두 null이 아닌 참조값임을 가정

- 반사성
  - x.equals(x)는 항상 true
- 대칭성
  - x.equals(y)가 true이면 y.equals(x)도 true여야한다.
- 추이성
  - x.equals(y)가 true이고, y.equals(z)가 true이면 x.equals(z)도 true
- 일관성
  - x.equals(y)를 반복해서 호출하면 항상 같은 값이 나와야한다.
- null-아님
  - x.equals(null)은 false이다.

대칭성 위배의 대표적인 예시

```
// 코드 10-1 잘못된 코드 - 대칭성 위배! (54-55쪽)
public final class CaseInsensitiveString {
    private final String s;

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

    // 대칭성 위배!
    @Override public boolean equals(Object o) {
        if (o instanceof CaseInsensitiveString)
            return s.equalsIgnoreCase(
                    ((CaseInsensitiveString) o).s);
        if (o instanceof String)  // 한 방향으로만 작동한다!
            return s.equalsIgnoreCase((String) o);
        return false;
    }

//    // 수정한 equals 메서드 (56쪽)
```

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

위 코드가 대칭성을 만족하지 못하는 이유는

`CaseInsensitiveString.equals(String)`과 `String.equals(CaseInsensitiveString)`의 값이 다르기 때문.

우리가 `String.equals`를 재정의할 수는 없기 때문에 대칭성을 위배하며, `String`과는 비교하면 안된다.

대칭성 & 추이성 위배 예시

`Point`를 상속받는 `ColorPoint`가 있다고 가정하자.

- Point

```
public class Point {  // 부모
    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;
    }
}
```

- ColorPoint

```
public class ColorPoint extends Point {  // 자식
    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 == c
    }
}
```

현재 상황에서는 다음과 같이 **대칭성 위배**가 발생한다

```
public static void main(String[] args) {
    Point p = new Point(1, 2);
    ColorPoint cp = new ColorPoint(1, 2, Color.RED);
    System.out.println(p.equals(cp) + " " + cp.equals(p)); //
 }
```

`p.equals`는 색상을 무시하고 Point 좌표값만 비교하기 때문에 `true`를 반환하지만,
`cp.equals`는 색상까지 비교하기 때문에 `false`를 반환한다.

전제1: 그렇다면 `cp`의 `equlas` 메서드에서 비교대상이 `p` 이면 색상을 무시한 뒤 비교하면 되지 않을까?

```
@Override public boolean equals(Object o) {
  if (!(o instanceof Point))
   return false;

  //o가 일반 Point면 색상을 무시하고 비교한다.
  if (!(o instanceof ColorPoint))
    return o.equals(this);
```

```
  // o가 ColorPoint면 색상까지 비교한다.
  return super.equals(o) && ((ColorPoint) o).color == color;
}
```

위 코드는 **추이성을 위배**한다.

```
public static void main(String[] args) {
     // 두 번째 equals 메서드(코드 10-3)는 추이성을 위배한다.
     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));
}
```

`p1.equals(p2)`와 `p2.equals(p3)`는 색상을 무시하여 `true`를 반환하지만, `p1.equals(p3)`는 색상을 고려하게 되어 `false`를 반환하기 때문이다. 둘의 결과값이 다르므로 추이성에 위반된다.

전제2: 그렇다면 instance of 가 아닌 getClass를 사용하여, 같은 구현 클래스 끼리만 비교하면 안될까?

```
public class Point{
  @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;
  }
}
```

위 코드는 규약도 지키고 값도 추가하면서 구체 클래스를 상속 가능한것 처럼 보이지만, 이는 리스코프 치환 원칙을 위반한다.

> 💡 리스코프 치환 원칙: 부모 타입의 모든 메서드가 하위 타입에서도 똑같이 잘 작동해야 한다.

위의 코드는 같은 구현 클래스의 객체와 비교할때만 `true` 를 반환하게 된다. 즉, `Point`의 **하위 클래스**를 비교할때는 항상 `false`를 반환하게 될 것이다.

상속에서 Equals 규약 문제 대체방법

객체 지향적 추상화의 이점을 포기하지 않고 구체 클래스를 확장해 새로운 값을 추가하면서 equals 규약을 만족시킬 방법은 존재하지 않는다. 따라서 상속 대신 **컴포지션**을 사용해 보자

```
public class ColorPoint {
    private final Point point;  // 컴포지션
    private final Color color;

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

    public Point asPoint() {  // 뷰 반환 메서드
        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(colo
    }

    @Override public int hashCode() {
        return 31 * point.hashCode() + color.hashCode();
    }
}
```

위처럼 상속 대신 컴포지션을 사용하면 규약을 지키며 값을 추가할 수 있다.

equals 메서드 구현 방법

1. `==` **연산자를 사용해 입력이 자기 자신의 참조인지 확인한다.**
    a. 단순 성능 최적화용으로, 비교 작업이 복잡한 상황에서 적절한 방법
2. `instanceof` **연산자로 입력이 올바른 타입인지 확인한다.**
    a. 올바른 타입이 아니면 `false`를 반환한다.
3. **입력을 올바른 타입으로 형변환한다.**
    a. 2번에서 `instanceof` 검사를 했기 때문에, 반드시 형변환에 성공한다.
4. **입력 객체와 자기 자신의 대응되는 '핵심' 필드들이 모두 일치하는지 하나씩 검사**
    a. 모든 필드가 일치하면 `true`, 하나라도 다르면 `false`를 반환한다.

타입 별 비교 방법

1. `float`과 `double`를 제외한 기본 타입 필드 : `==` 연산자
2. 참조 타입 필드 : 각각의 `equals` 메서드
3. `float` / `double` : `Float.compare(float, float)` / `Double.compare(double, double)`
4. 배열 : 모든 원소가 핵심 필드라면 `Arrays.equals()`
5. `null` 정상 값 취급 참조 타입 필드 : `Object.equals(Object, Object)`

성능 최적화 방법: 최상의 성능을 바란다면, 다를 가능성이 크거나 비교하는 비용이 싼 필드를 먼저 비교하자.

equlas를 재정의할 땐 @Override를 사용하여 검사하자.

```
public boolean equals(MyClass o){  // 입력 타입은 반드시 Object
    ...
}

@Override public boolean equals(Myclass o){  // 컴파일 오류 발생
    ...
}
```

해당 코드는 `Object.equals`를 재정의 한것이 아니라 다중정의한 것이 된다. 이 메서드는 하위 클래스에서의 `@Override` 애너테이션이 긍정 오류(거짓 양성)을 내게하고 보안 측면에서도 잘못된 정보를 준다.

`@Override` 애너테이션을 일관되게 사용한다면, 실수를 예방할 수 있다.*(아이템 40)* 아래 코드는 컴파일되지 않기 때문에 무엇이 문제인지 정확히 알 수 있다.

⭐핵심 정리

> 💡 꼭 필요한 경우가 아니면 `equals`를 재정의하지 말자. 많은 경우에 `Object.equals`가 원하는 비교를 정확히 수행해주기 때문이다. 재정의해야 할 때는, 그 클래스의 핵심 필드를 모두 빠짐없이 다섯 가지 규약을 확실히 지켜가며 비교해야 한다.

Item11 equals를 재정의하려거든 hashCode도 재정의하라

hashCode를 재정의해야하는 이유

`equals`를 재정의한 클래스 모두에서 `hashCode`도 재정의해야 한다!!

그렇지 않으면 `hashCode` 일반 규약을 어기게 되어 해당 클래스의 인스턴스를 `HashMap`이나 `HashSet` 같은 컬렉션의 원소로 사용할 때 문제를 일으킬 것이다.

- 해시코드 규약

1. `equals` 비교에 사용되는 정보가 변경되지 않았다면, 애플리케이션이 실행되는 동안 그 객체의 `hashCode` 메서드는 항상 일관된 값을 반환해야 한다. 단, 다시 실행한다면 값이 달라져도 된다.
2. `equals(Object)`가 두 객체를 같다고 판단했다면, 두 객체의 `hashCode`는 똑같은 값을 반환해야 한다.
3. `equals(Object)`가 두 객체를 다르다고 판단했더라도, 두 객체의 `hashCode`가 서로 다른 값을 반환할 필요는 없다. 단, 다른 객체에 대해서는 다른 값을 반환해야 해시테이블의 성능이 좋아진다.

`hashCode` 재정의를 잘못했을 때 문제가 되는 조항은 두 번째다. 즉 논리적으로 같은 객체는 같은 해시코드를 반환해야한다.

hashCode 재정의 방법

이상적인 해시 함수는 주어진 서로 다른 인스턴스들을 32비트 정수 범위에 균일하게 분배해야 한다.

다음과 같은 방법으로 `hashCode`를 작성하자

1. `int` 변수 `result` 를 선언한 후 값 c로 초기화한다. 이때 c는 해당 객체의 첫 번째 핵심 필드를 단계 2-a 방식으로 계산한 해시코드다.
2. 해당 객체의 핵심 필드 `f` 각각에 대해 다음 작업을 수행한다.

   2-a. 해당 필드의 해시코드 c를 계산한다.

   - **기본** 타입 필드 : `Type.hashCode(f)` 를 수행한다. (Type : 해당 기본 타입의 박싱 클래스)
   - **참조** 타입 필드면서 이 클래스의 `equals` 메서드가 이 필드의 `equals`를 재귀적으로 호출해 비교한다면, 이 필드의 `hashCode`를 재귀적으로 호출한다.
   - **배열**이라면, 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(`0`)을 사용한다. 모든 원소가 핵심 원소라면 `Arrays.hashCode` 를 사용한다.

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

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

3. result를 반환한다.

파생 필드는 해시코드 계산에서 제외해도 되며, **equals 비교에 사용되지 않은 필드는 반드시 제외해야한다**

hashCode 메서드 예시

- 전형적인 hashCode 메서드

```
@Override public int hashCode() {
  //PhoneNumber 클래스의 핵심 필드 3개만 이용
  int result = Short.hashCode(areaCode);
  result = 31 * result + Short.hashCode(prefix);
  result = 31 * result + Short.hashCode(lineNum);
  return result;
}
```

PhoneNumber 클래스의 핵심 필드 3개만 이용하기 때문에 동치인 PhoneNumber 인스턴스들은 같은 해시코드를 가지는 것이 확실해진다.

- 한줄 짜리 hashCode 메서드 - 성능 이슈 있음

```
@Override public int hasCode(){
    return Objects.hash(lineNum, prefix, areaCode);
}
```

비용이 크기 때문에 성능 문제 없을 때만 사용

- 해시코드를 지연 초기화 하는 메서드(캐싱 이용) - 스레드 안정성 고려해야한다.

클래스가 불변이고 해시코드를 계산하는 비용이 크다면, 새로 계산하기 보다는 **캐싱하는 방식**을 고려해야 한다.

인스턴스가 만들어질때 해시코드를 미리 계산해두거나, 해시의 `Key` 로 사용되지 않는 경우라면 `hashCode` 가 처음 불릴 때 계산하는 **지연 초기화(lazy initialization)** 방식도 좋다. 필드를 지연 초기화하려면 클래스를 **스레드 안전**하게 만들어야 한다.

```
private int hashCode; // 자동으로 0으로 초기화

@Override public int hashCode() {
  int result = hashCode;
    if (result == 0) {
    result = Short.hashCode(areaCode);
    result = 31 * result + Short.hashCode(prefix);
    result = 31 * result + Short.hashCode(lineNum);
    hashCode = result;
  }
  return result;
}
```

⭐핵심 정리

`equals`를 재정의할 때는 `hashCode`도 반드시 재정의해야 한다. 재정의한 `hashCode`는 `Object`의 API 문서에 기술된 일반 규약을 따라야 하며, 서로 다른 인스턴스라면 되도록 해시코드도 서로 다르게 구현해야 한다. 구현하기가 어렵지만, 아이템 10에서 얘기한 AutoValue 프레임워크를 사용하면 자동으로 만들어준다. IDE들도 이런 기능을 일부 제공한다.

---

toString을 항상 재정의하라

`Object`가 기본적으로 제공하는 `toString` 메서드는 `클래스_이름@16진수로_표시한_해시코드`를 반환한다. `toString`일반 규약에 따라 간결하면서도 사람이 읽기 쉬운 형태의 유익한 정보를 반환해야한다.

오류에 관한 문제가 아닌 디버깅, 사용성을 위해 재정의 해주면 좋다!

1. **객체가 가진 주요 정보는 되도록이면 모두 반환하는게 좋다**. 반환할 정보가 너무 많으면 요약하여 반환하자.
2. **반환값의 포맷을 문서화할지 정해야한다**. 이를 통해 값 그대로 입출력에 사용하거나 CSV 파일처럼 사람이 읽을 수 있는 데이터 객체로 저장할 수 있다.

```
 /**
      * 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. 예컨대 가입자 번호가 123이라면
      * 전화번호의 마지막 네 문자는 "0123"이 된다.
      */
@Override public String toString() {
    return String.format("%03d-%03d-%04d",
    areaCode, prefix, lineNum);
}
```

3. 정적 유틸리티 클래스, 열거 타입은 따로 재정의 하지 않아도 된다.

⭐핵심정리

💡 모든 구체 클래스에서 Object의 toString을 재정의하자. 상위 클래스에서 이미 알맞게 재정의한 경우는 예외이다. toString을 재정의한 클래스는 사용하기도 즐겁고 그 클래스를 사용한 시스템을 디버깅하기 쉽게 해준다. toString은 해당 객체에 관한 명확하고 유용한 정보를 읽기 좋은 형태로 반환해야 한다.

---

clone 재정의는 주의해서 진행하라

Cloneable이란?

`Cloneable` 이란, **복제해도 되는 클래스임**을 명시하는 용도의 믹스인 인터페이스를 의미한다.

하지만! `clone` 메서드가 선언된 곳이 `Cloneable`이 아닌 `Object`이고, 그 마저도 `Protected` 로 선언되어 있어, `Cloneable`을 구현하는 것만으로는 외부 객체에서 `clone` 메서드를 호출할 수 없다. `Cloneable`은 아래 코드처럼 아무 메서드가 없는 인터페이스이다.

```
public interface Cloneable {
}
```

- 그렇다면 `Cloneable`의 역할은 무엇일까?

`Cloneable`을 구현하는 것을 통해 `clone`의 동작방식을 결정한다. `Cloneable` 을 구현한 클래스의 인스턴스에서 `clone()` 을 호출하면 그 객체의 필드들을 하나하나 복사한 객체를 반환하며, 그렇지 않은 클래스의 인스턴스에서 호출하면 `CloneNotSupportedException` 을 던진다. (일반적인 인터페이스를 구현한 것과 다르다.)

- clone 메서드의 일반 규약 (매우 허술함)

객체의 복사본을 생성해 반환한다는 것은, 복사본이 원본 객체와 달라야 함을 의미한다. 즉 반환된 객체와 원본 객체는 독립적인 주소를 가지고 있으면서 논리적으로는 동일해야 한다.

```
x.clone() != x   // True
x.clone().getClass() == x.getClass()  // True
x.clone.equals(x)  // True
```

- `clone` 메서드의 문제점

하위 클래스에서 `super.clone`을 호출했을 때 문제가 발생한다.

ex. 어떤 클래스 B가 클래스 A를 상속할때, 하위 클래스인 B의 `clone` 메서드는 B 타입 객체를 반환하지만 상위 클래스 A의 `clone` 메서드는 A 타입 객체를 반환한다. 즉 호출된 상위 클래스의 객체가 만들어지는 문제가 있다.

Clone 메서드 구현 방법

1. 가변 상태를 참조하지 않는 경우

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

    @Override public PhoneNumber clone() { // Object
        try {
            return (PhoneNumber) super.clone();  // 형변환 처
리
         } catch (CloneNotSupportedException e) { // 검사 예
외(체크 예외)
            throw new AssertionError();  // 일어날 수 없는 일
이다.
        }
    }
}
```

위 코드에서 (`PhoneNumber`)의 형변환은 반드시 성공한다. 이처럼 가변 상태를 참조하지 않는 경우 형변환을 통해 반환하는 것이 가능하지만, 가변 객체를 참조하는 순간 사용할 수 없는 방식이다.

2. 가변 상태를 참조하는 경우

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

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

        //...

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

```
    }
}
```

- **super.clone 메서드를 사용해서 복사**

해당 클래스를 단순히 `super.clone()` 을 통해 복제하는 경우, `elements` 필드가 **원본 인스턴스와 똑같은 배열을 참조(얕은 복사)**하게 되어 불변식을 해친다. (원본이나 복제본 중 하나를 수정하면 다른 하나도 수정된다)

- **생성자를 이용한 복사**

`Stack`의 생성자를 호출한다면, 생성자에서 원본 객체를 건드리지 않은 채 복제된 객체의 불변식을 보장할 수 있다. 즉 사실상 생성자와 같은 효과를 낸다.

복잡한 가변 객체를 복제하는 방법

1. 배열 clone의 재귀 호출

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

하지만 복제 가능한 클래스를 만들기 위해서는 일부 필드에서 `final` 한정자를 제거해야 하므로 `가변 객체를 참조하는 필드는 final로 선언하라` 라는 용법에 어긋난다.

2. **깊은 복사(Deep Copy)**

깊은 복사란, **실제 값을 새로운 메모리의 공간에 복사**하는 것을 말한다. 반대로 얕은 복사(Shallow copy)는 **주소값 자체를 복사**하는 방식이다. 즉, 새로운 메모리에 할당되지 않는다.

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

```
  private statis class Entry {
     ...
     //이 엔트리가 가르키는 연결 리스트를 반복문으로 복사
    Entry deepCopy(){
        Entry result = new Entry(key, value, next);
    for (Entry p = result; p.next != null; p = p.next)
        p.next = new Entry(p.next.key, p.next.value, p.nex
t.next);
    return result;
}

 @Override public HashTable clone() {
    try{
        HashTable result = (HashTable) super.clone();
        result.buckets = new Entry[buckets.length];  // 새로
운 버킷 배열 할당
        for (int i=0 ; i < buckets.length; i++)
            if (buckets[i] != null)
                result.buckets[i] = buckets[i].deepCopy();
// 비지 않은 각 버킷에 대해 방어적 복사
        return result;
    } catch (CloneNotSupportedException e) {
            throw new AssertionError();
    }

}
```

3. 고수준 API 활용

먼저, `super.clone()` 을 호출하여 얻은 객체의 모든 필드를 초기 상태로 설정 한다음 **원복 객체의 상태를 다시 생성**하는 고수준 메서드들을 호출한다.

고수준 메서드 → `put`, `add` 등과 같은 메서드

ex) `HashTable` 에서는 `buckets` 필드를 새로운 버킷 배열로 초기화 한 후 원본 테이블에 담긴 모든 키-값 쌍 각각에 복제본 테이블의 `put(key,value)` 메서드를 호출

하지만, 생성자와 마찬가지로 `clone()` 함수 내부에서도 하위 클래스에서 재정의될 수 있는 메서드를 호출하지 않아야 한다. 즉 하위 클래스의 오버라이딩을 막아야 하므

로 `put(key,value)` 메서드는 `final` 이거나 `private` 이어야 한다.

복사 생성자와 복사 팩터리

현재까지 `Cloneable` 문제점을 정리하면 다음과 같다.

1. 기본 구현 `Object.clone()` 얕은 복사본을 반환한다.
2. `Cloneable` 구현을 강요한다.
3. `CloneNotSupportedException` 과 같은 체크 예외를 발생시키므로 이에 대한 처리가 따로 필요하다.
4. `Object.clone()` 반환 Object 반환된 개체 참조를 형변환 해야 한다.

복사 생성자(변환 생성자)와 복사 팩터리(변환 백터리)는 이러한 문제점들을 모두 해결해준다. 단순히 복사 생성자란, 단순히 자신과 같은 클래스의 인스턴스를 인수로 받는 생성자를 말한다.

- 복사 생성자

```
class Student
{
    private String name;
    private int age;
    private Set<String> subjects;   // 가변 필드

    public Student(Student student)
    {
        this.name = student.name;
        this.age = student.age;
        this.subjects = new HashSet<>(student.subjects); //
Deep Copy
    }
}
```

- 복사 팩터리

```
public static Student newInstance(Student student) {
        return new Student(student);
}
```

⭐핵심정리

> 💡 `Cloneable` 이 몰고 온 모든 문제를 되짚어봤을 때, 새로운 인터페이스를 만들 때는 절대 `Cloneable` 확장해서는 안 되며, 새로운 클래스도 이를 구현해서는 안된다. 기본 원칙은 '복제 기능은 생성자와 팩터리를 이용하는게 최고' 라는 것이다. 단 배열만은 `clone` 을 통해 복제하는 것이 가장 깔끔한, 이 규칙의 합당한 예외라 할 수 있다.

Item14 Comparable을 구현할지 고민하라

Comparable의 compareTo 메서드

`Comparable` 인터페이스는 `compareTo` 메서드를 하나만 가지는 인터페이스이다. `Object`의 메서드가 아니지만, 성격은 2가지를 제외하면 `Object`의 `equals`와 같다. `compareTo` 는 단순 동치성 비교에 더해 순서까지 비교할 수 있으며, 제네릭하다. `comparable` 을 구현했다는 것은 그 클래스의 인스턴스들에는 자연적인 순서가 있음을 뜻한다.

- compareTo 규약

equals 규약과 동치성 비교면에서 같으며, 추가로 다음을 만족한다.

1. 해당 객체가 주어진 객체보다 작은 경우 : 음수 반환(-1)
2. 해당 객체가 주어진 객체보다 같은 경우 : 0 반환
3. 해당 객체가 주어진 객체보다 큰 경우 : 양수 반환(1)

추가로 필수는 아니지만, `equals`의 결과와 `compareTo`의 결과가 같아야 함을 반드시 지키면 좋다. → 정렬된 컬렉션들은 동치성을 비교할 때 `equals`가 아닌 `compareTo`를 사용한다.

compareTo 메서드 작성 요령

1. 타입을 인수로 받는 제네릭 인터페이스이다.

   `compareTo` 메서드의 인수 타입은 컴파일 타임에 정해지므로, 입력 **인수의 타입을 확인하거나 형변환할 필요가 없다.**

2. `null` 을 인수로 넣어 호출하면, `nullPointException` 을 던져야 한다.

3. **객체 참조** 필드를 비교하는 경우

   `compareTo` 메서드를 재귀적으로 호출한다. `Comparable` 을 구현하지 않은 필드나 표준이 아닌 순서로 비교해야 한다면 `Comparator` 을 사용하자.

```
public int compareTo(CaseInsensitiveString cis) {
        return String.CASE_INSENSITIVE_ORDER.compare(s, ci
s.s);
}
```

4. **정수 타입** 기본 필드를 비교하는 경우 박싱된 기본 타입 클래스들에 새로 추가된 정적 메서드 `compare`을 사용한다. 아래는 Integer.compare의 예시 → 차를 기반으로 반환하지 않음.

```
public final class Integer extends Number implements Compar
able<Integer> {
  public static int compare(int x, int y) {
        return (x < y) ? -1 : ((x == y) ? 0 : 1);
    }
}
```

5. 클래스에 핵심 필드가 여러개라면, 가장 **핵심적인 필드부터 비교**한다.

```
public final class PhoneNumber implements Cloneable, Compar
able<PhoneNumber> {
    private final short areaCode, prefix, lineNum;

    //compareTo 구현
    public int compareTo(PhoneNumber pn) {
        int result = Short.compare(areaCode, pn.areaCode);
//가장 중요한 필드
        if (result == 0)  {
          result = Short.compare(prefix, pn.prefix); //두번
째로 중요한 필드
          if (result == 0)
            result = Short.compare(lineNum, pn.lineNum); //
세번째로 중요한 필드
        }
        return result;  // 비교 결과가 0이 아니라면 순서가 결정됨
```

```
    }
}
```

Comparator

자바 8부터는 `Compartor` 인터페이스가 비교자 생성 메서드와 팀을 꾸려, **메서드 연쇄 방식으로 비교자를 생성**할 수 있게 되었다. 하지만 약간의 성능 저하가 뒤따르긴 한다.

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

   default Comparator<T> thenComparing(Comparator<? super T
> other) {
        Objects.requireNonNull(other);
        return (Comparator<T> & Serializable) (c1, c2) -> {
            int res = compare(c1, c2);
            return (res != 0) ? res : other.compare(c1, c
2);
        };
    }

    public static <T> Comparator<T> comparingInt(ToIntFunct
ion<? super T> keyExtractor) {
        Objects.requireNonNull(keyExtractor);
        return (Comparator<T> & Serializable)
            (c1, c2) -> Integer.compare(keyExtractor.applyA
sInt(c1), keyExtractor.applyAsInt(c2));
    }
    ...
}
```

- `comparingInt()` : 어떤 값을 기준으로 정렬할 것인지 인자로 전달받아서, 그 키를 기준으로 순서를 정하는 비교자를 반환하는 정적 메서드

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

```
                    .thenComparingInt(pn -> pn.prefix) //1
                    .thenComparingInt(pn -> pn.lineNum);
//2

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

주의사항 및 정리

- 오버플로우

값의 차를 기준으로 결과를 반환하는 `compareTo` 나 `compare` 메서드는, 정수 오버플로를 일으키거나 부동소수점 계산 방식에 따른 오류를 낼 수 있으니 사용하면 안된다.

예를 들어, `o1` = `1`, `o2` = `-2,147,483,648` 라고 가정하자. 두 수를 `return o1 - o2;` 형식으로 반환한다면, `1 - (-2,147,483,648) = 2,147,483,649` 로 양수가 나와야 하는 데 `-2,147,483,648` 으로 음수가 나와 1이 더 작다는 상황이 발생하게 된다.

```
static Comparator<Object> hasCodeOrder = new Comparator<>()
{
  public int compare(Object o1, Object o2){
    return o1.hashCode() - o2.hashCode(); // 해시코드 기반 차
이 반환
  }
};
```

- 해결 방법

1. **정적 compare 메서드를 활용한 비교자**

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

2. **비교자 생성 메서드를 활용한 비교자**

```
static Comparator<Object> hasCodeOrder =
  Comparator.comparingInt(o -> o.hashCode());  // 해시코드 값 기반
```

⭐핵심정리

💡 순서를 고려해야 하는 값 클래스를 작성한다면 꼭 `Comparable` 인터페이스를 구현하여, 그 인스턴스들을 쉽게 정렬하고, 검색하고, 비교 기능을 제공하는 컬렉션과 어우러지도록 해야 한다. `compareTo` 메서드에서 필드의 값을 비교할때 < 와 > 연산자를 사용하지 말아야 한다. 대신, 박싱된 기본 타입 클래스(`Integer`, `Double`...)가 제공하는 정적 `compare` 메서드나 `Comparator` 인터페이스가 제공하는 비교자 생성 메서드를 사용하자.