이펙티브 자바

Item 12 ~ Item 14

2024/01/23

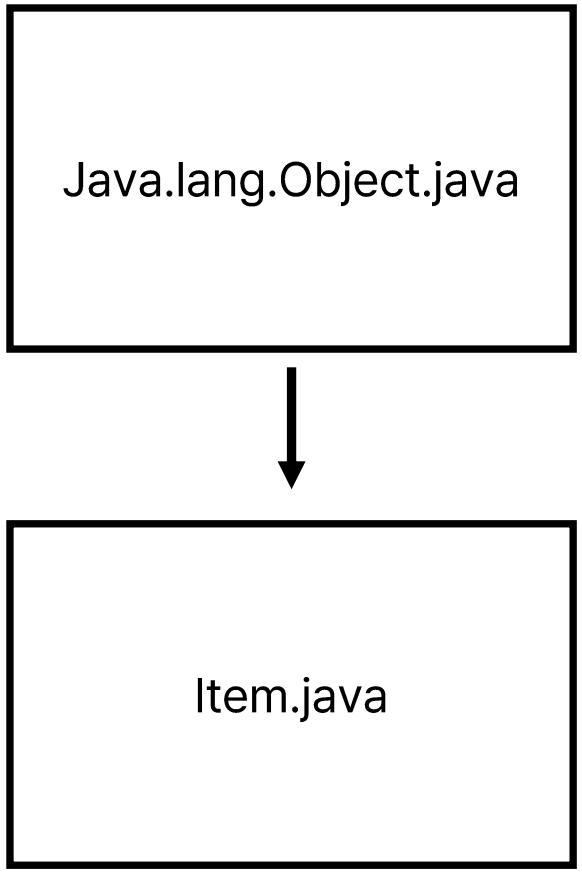
발표자 : 박지원

Item 12

toString은 항상 재정의하라

Item 12 toString은 항상 재정의하라

자바의 클래스는 모두 Object 클래스를 상속




```java
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

```java
@Override
public String toString() {
    return "Item{" +
            "name='" + name + '\'' +
            ", price=" + price +
            '}';
}
```

Item 12 toString은 항상 재정의하라

toString을 잘 구현한 클래스는 사용하기에 훨씬 즐겁고(?) 디버깅하기 쉽다.

직접호출 외에도 다양하게 사용되고 있음

Collection에 경우 default로 iterator에 의해 list 구성요소를 리턴

Printf에서 인스턴스를 넣을 경우 toString()실행됨

Assert구문을 사용할경우 실행됨(X) (Assertion.assertThat().isEqualTo() 경우 인스턴스 값을 비교)

```
item1 = item4.item12.Item@4dd8dc3
```

```
expected: "item{name='item1', price=1000} (Item@36b4cef0)"
 but was: "item{name='item1', price=1000} (Item@3d921e20)"
Expected :item{name='item1', price=1000}
Actual   :item{name='item1', price=1000}
```

Item 12 toString은 항상 재정의하라

toString을 잘 구현한 클래스는 사용하기에 훨씬 즐겁고(?) 디버깅하기 쉽다.

진단메세지를 쉽고 간단하게 만들 수 있다.

toString에 객체의 주요 정보를 모두 반환 하는게 좋다.

주요정보가 너무 방대하다면 요약정보를 반환할것 (배열의 크기)

```
public static void main(String[] args) {
    Item item1=new Item("item1",1000);
    System.out.println(item1+"아이템을 획득하였습니다");

}
```

Item 12 toString은 항상 재정의하라

toString을 구현할 경우 반환값의 포맷을 명시 할지 정할 것

포맷을 명시하면 표준적이고, 명확해지지만 포맷을 변경할 경우 유연성이 떨어짐

개발자의 의도를 밝혀야한다는 것

toString이 반환한 값에 포함된 정보를 얻어 올 수 있는 API를 제공하자(접근자)

```
@Override
public String toString() {
    return String.format("name = %10s price = %7d  ",this.name,this.price);
}
```

Item 13

clone 재정의는 주의해서 진행하라

Item 13 clone 재정의는 주의해서 진행하라

clone은 재정의해서 사용가능(기본형이 protected이기 때문에)

cloneable 인터페이스를 사용하여 복제가능한 클래스임을 명시함과 동시에 clone의 동작 방식을 결정함

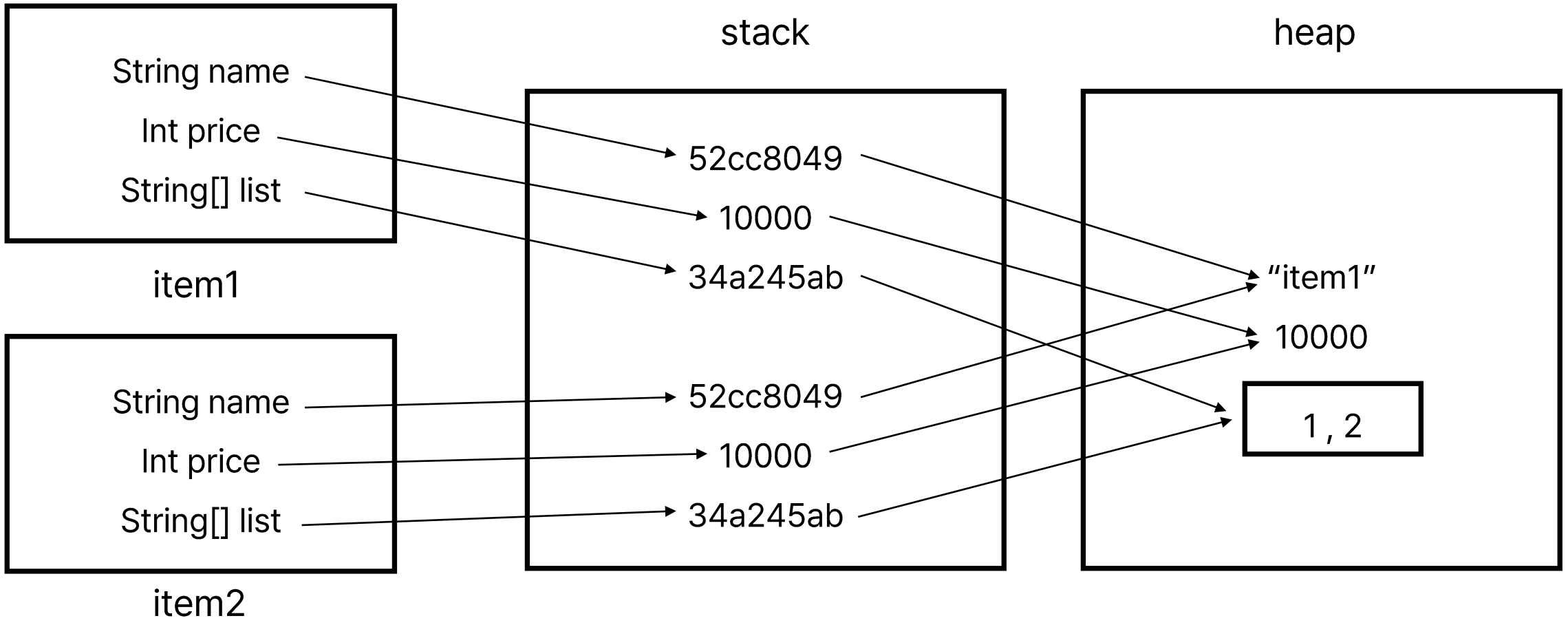
기존의 기능을 정의하는 인터페이스와 다른 용도로 사용됨 (상위 클래스인데 어떻게 가능할까?)

허술하고 모순적인 매커니즘을 사용하여 상위클래스에 접근

```
@IntrinsicCandidate
protected native Object clone() throws CloneNotSupportedException;
```

Item 13 clone 재정의는 주의해서 진행하라

Clone은 해당 객체의 맴버변수를 원본 **객체의 값**을 복사한다(shallow copy)



Item 13 clone 재정의는 주의해서 진행하라

Clone 메서드는 생성자와 같은 효과를 낸다

Clone은 원본 객체에 아무런 해를 끼치지 않는 동시에 복제된 객체의 불변식을 보장해야 한다.

가변 객체를 참조하는 경우 clone을 재귀적으로 호출해주자

그러나 list가 final이라면? -> 불가능

복제하기 위해서는 final을 제거해야 한다.

```
@Override
public Item clone() {
    try {
        Item item= (Item)super.clone();
        item.list=list.clone();
        return item;
    }
    catch (CloneNotSupportedException e){
        return null;



    }
}
```

Item 13 clone 재정의는 주의해서 진행하라

객체 배열을 맴버로 가지는 경우, 객체배열의 각각의 인스턴스는 얕은 복사가 진행된다.

해당 객체 클래스에 clone을 재정의하는 것으로 해결불가 (배열을 복사다른 clone이 호출된다)

메소드를 선언하여 직접 인스턴스를 새로 생성하고 clone내에서 처리

```
static class Component{
        String name;

        public Component(String name){
            this.name=name;
        }
        Component deepCopy(){
            return new Component(name);
        }
    }
```

```
@Override
public Item clone() {
    try {
        Item item= (Item)super.clone();
        item.list=new Component[list.length];
        for(int i=0;i<list.length;i++){
            if(list[0]!=null)
               item.list[i]=list[i].deepCopy();
        }
        return item;
    }
    catch (CloneNotSupportedException e){
        return null;
    }
}
```

Item 13 clone 재정의는 주의해서 진행하라

Clone으로 하위 클래스에서 재정의한 메서드를 호출하면 하위 클래스에서 복제 과정에서 자신의 상태를 교정할 기회를 잃어버릴 수 있다.

상속용 클래스의 Cloneable을 구현해서는 안된다. (CloneNotSupportedException 사용)

이미 Cloneable을 구현하는 모든 클래스는 clone을 재정의 해야한다.

만약 아직 Cloneable을 사용하지 않았다면 복사 생성자, 복사팩터리를 고려해보자

Final 클래스인 경우 cloneable을 한시적으로 허용을 고려해보자

Item 14

Comparable 을 구현할지 고려하라

Item 14 Comparable을 구현할지 고려하라

Comparable을 구현했다는 것은 인스턴스 간의 순서가 보장된다는 것

compareTo의 규약은 equal과 거이 동일(대칭성, 추이성..)

따라서 새로운 값을 추가한 구체 클래스(상속)에는 규약이 지켜지지 않음(대칭성)

확장 대신 독립된 클래스를 만들고, 원래의 인스턴스를 가리키는 필드를 만들 것

CompareTo의 동치성 결과가 equals(논리적 동치)와 같게 할것(컬렉션에서 문제가 생길 수 있음)

```
public interface Comparable<T>{
        int compareTo(T t);
}
```

Item 14 Comparable을 구현할지 고려하라

제네릭타입이기 때문에 컴파일에 타입이 정해진다.(타입오류 체크가능)

Comparable(본인을 비교)을 구현하지 않은 경우 Comparator(파라미터 두개를 비교)을 대신 사용

부등호를 사용하지않고 compare메서드를 사용할 것

https://stackoverflow.com/questions/1726254/why-is-javas-double-comparedouble-double-implemented-the-way-it-is

```
class Student implements Comparable<Student>{
    private String name;
    private int korScore;
    private int engScore;

    private int mathScore;

    @Override
    public int compareTo(Student o) {
        return this.engScore - o.engScore;
    }
```

Item 14 Comparable을 구현할지 고려하라

오버플로우, 부동소수점에 주의 할 것

BigDecimal 사용시 String으로 파라미터를 설정할 것(double이 완벽하지 표현하지 못할 경우 문제 발생)

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

```
BigDecimal bigDecimal1=new BigDecimal("1.1");
System.out.println(bigDecimal1);
BigDecimal bigDecimal2=new BigDecimal(1.1);
System.out.println(bigDecimal2);
```

```
1.1
1.100000000000000088817841970012523233890533447265625
```