Effective Java

Item 19. 상속을 고려해 설계하고 문서화하라. 그러지 않았다면 상속을 금지하라

개요

1. 상속을 고려한 문서화
2. 상속을 고려한 설계
3. 상속을 금지해야하는 경우

개요

1. 상속을 고려한 문서화
2. 상속을 고려한 설계
3. 상속을 금지해야하는 경우

상속을 고려한 문서화

상속용 클래스는 재정의할 수 있는 메서드를 문서로 남겨야 한다.

재정의 가능한 메서드

```
public void method1() { ... }
protected void method2() { ...
void method3() { ... }
```

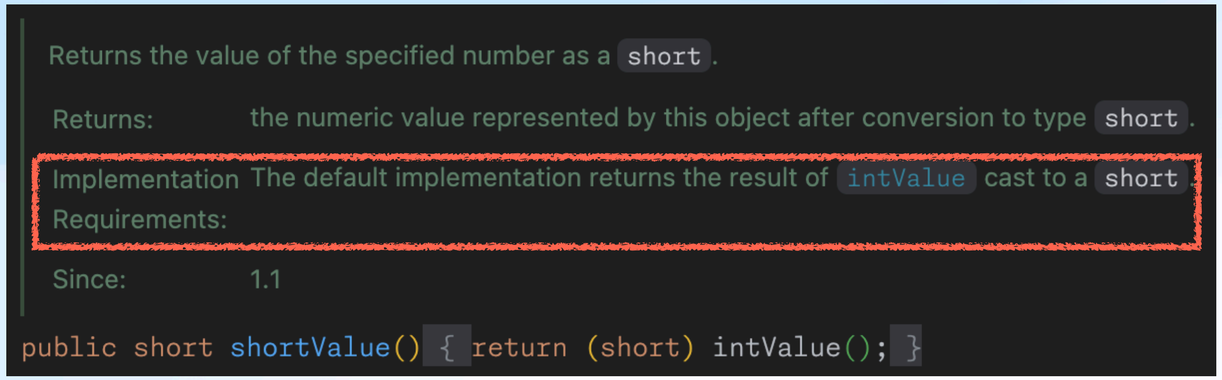
- private이 아닌 메서드 (public, protected, default)
- final로 선언되지 않은 메서드
- static이 아닌 메서드 (인스턴스 메서드)

```
private void method4() { ... }        // private : 상속 X
public final void method5() { ... }  // final: 재정의 X
public static void method6() { ... } // static 메서드: 클래스에 속
```

상속을 고려한 문서화

상속용 클래스는 재정의할 수 있는 메서드를 문서로 남겨야 한다.

@implSpec : Implementation Specification (구현 명세)



Number.class

intValue() 메서드를 **오버라이딩**하면 **shortValue()**에도 영향을 준다.

상속을 고려한 문서화

상속용 클래스는 호출 순서와 호출 결과를 문서로 남겨야 한다.

```
이 목록에서 인덱스가 fromIndex, Inclusive 및 toIndex 사이에있는 모든 요소를 배타적으로 제거합니다. 후속
요소를 왼쪽으로 이동합니다 (색인 감소). 이 호출은 목록을 (toIndex - fromIndex) 요소로 단축시킵니다. (
toIndex==fromIndex 인 경우이 작업은 효과가 없습니다.)
이 방법은이 목록의 clear 작업 및 해당 하위 목록에 의해 호출됩니다. 목록 구현의 내부를 활용하기 위해이 방법을 무시
하면이 목록 및 해당 하위 목록의 clear 작업의 성능을 크게 향상시킬 수 있습니다.

매개 변수 : fromIndex - 제거 할 첫 번째 요소의 색인
           toIndex - 제거 할 마지막 요소 후 인덱스

구현
요구 사항 : 이 구현은 fromIndex 이전에 위치한 목록 반복자를 가져오고 ListIterator.next 반복적으로 호
출 한 다음 ListIterator.remove 전체 범위가 제거 될 때까지 반복합니다. 참고 :
ListIterator.remove 선형 시간이 필요한 경우이 구현에는 2 차 시간이 필요합니다.
```

```
protected void removeRange(int fromIndex, int toIndex) {  Complexity is 4 Everything is cool!
    ListIterator<E> it = listIterator(fromIndex);
    for (int i = 0, n = toIndex - fromIndex; i < n; i++) {
        it.next();
        it.remove();
    }
}
```

AbstractList.class

- **호출 순서:** listIterator(fromIndex)
  - ➡ ListIterator.next()
  - ➡ ListIterator.remove()
- **호출 결과:** 호출 범위의 요소들이 삭제되며

뒤에 오는 원소들의 인덱스가 감소된다.

상속을 고려한 문서화

✅ 상속용 클래스는 재정의할 수 있는 메서드를 문서로 남겨야 한다.

✅ 상속용 클래스는 호출 순서와 호출 결과를 문서로 남겨야 한다.

상속은 결합도가 매우 높기 때문에, 상세한 문서화가 필요하다.

하위 클래스는 **상위 클래스의 구현에 직접 의존**하기 때문에 결합도가 높다.
재정의 가능한 메서드가 내부적으로 어떻게 사용되는지 알아야, **하위 클래스가 메서드를 안전하게 재정의**할 수 있다.

개요

1. 상속을 고려한 문서화
2. **상속을 고려한 설계**
3. 상속을 금지해야하는 경우

상속을 고려한 설계

어떤 메서드를 protected로 공개하는 것이 좋을까?

- protected 메서드는 최대한 적어야 한다.

외부로 노출되는 메서드이므로 캡슐화를 약화시키고 변경이 어렵다.

- 여러 하위 클래스에서 공통으로 사용되는 메서드가 좋다.

하위 클래스 3개 정도에서 실제로 사용되는지 테스트하자.

removeRange()를 재정의하면 clear() 연산의 성능을 크게 개선할 수 있다.

```
이 목록에서 인덱스가 fromIndex , Inclusive 및 toIndex 사이에있는 모든 요소를 배타적으로 제거합니다. 후속
요소를 왼쪽으로 이동합니다 (색인 감소). 이 호출은 목록을 (toIndex - fromIndex) 요소로 단축시킵니다. (
toIndex==fromIndex 인 경우이 작업은 효과가 없습니다.)
이 방법은이 목록의 clear 작업 및 해당 하위 목록에 의해 호출됩니다. 목록 구현의 내부를 활용하기 위해이 방법을 무시
하면이 목록 및 해당 하위 목록의 clear 작업의 성능을 크게 향상시킬 수 있습니다.

매개 변수 : fromIndex - 제거 할 첫 번째 요소의 색인
           toIndex - 제거 할 마지막 요소 후 인덱스

구현        이 구현은 fromIndex 이전에 위치한 목록 반복자를 가져오고 ListIterator.next 반복적으로 호
요구 사항 : 출 한 다음 ListIterator.remove 전체 범위가 제거 될 때까지 반복합니다. 참고 :
           ListIterator.remove 선형 시간이 필요한 경우이 구현에는 2 차 시간이 필요합니다.

protected void removeRange(int fromIndex, int toIndex) {   Complexity is 4 Everything is cool!
    ListIterator<E> it = listIterator(fromIndex);
    for (int i = 0, n = toIndex - fromIndex; i < n; i++) {
        it.next();
        it.remove();
    }
}
```

AbstractList.class

상속을 고려한 설계

상속용 클래스의 생성자는 재정의 가능 메서드를 호출해서는 안된다.

➡ 제대로 초기화되지 않은 불완전한 객체의 메서드가 호출된다.

```java
// 재정의 가능 메서드를 호출하는 생성자 - 따라 하지 말 것! (115쪽)
public class Super {
    // 잘못된 예 - 생성자가 재정의 가능 메서드를 호출한다.
    public Super() {
        overrideMe();
    }

    public void overrideMe() {
    }
}
```

```java
// 생성자에서 호출하는 메서드를 재정의했을 때의 문제를 보여준다. (126쪽)
public final class Sub extends Super {
    // 초기화되지 않은 final 필드. 생성자에서 초기화한다.
    private final Instant instant;

    Sub() {
        instant = Instant.now();
    }

    // 재정의 가능 메서드. 상위 클래스의 생성자가 호출한다.
    @Override
    public void overrideMe() {
        System.out.println(instant);
    }

    public static void main(String[] args) {
        Sub sub = new Sub();
        sub.overrideMe();
    }
}
```

개요

1. 상속을 고려한 문서화
2. 상속을 고려한 설계
3. **상속을 금지해야하는 경우**

상속을 금지해야하는 경우

Cloneable과 Serializable 인터페이스를 구현한 클래스

- clone() 과 readObject() 메서드는 생성자와 비슷한 효과를 낸다.

  clone() : 객체의 복사본 생성

  readObject() : 직렬화된 데이터로부터 객체를 다시 생성

➡ 생성시 오버라이딩된 메서드를 호출할 수 있어 위험

```java
@Override
public Animal clone() {
    try {
        Animal cloned = (Animal) super.clone();

        // 문제 발생 지점: 재정의 가능 메서드 호출
        cloned.initialize();  // 하위 클래스가 오버라이드한 initialize()가 호출
됨
        return cloned;
    } catch (CloneNotSupportedException e) {
        throw new AssertionError("발생할 수 없음", e);
    }
}
```

상속을 금지해야하는 경우

상속용 클래스가 writeReplace나 readResolve를 갖는다면 protected로 선언해야 한다.

```
class SubSingleton extends SuperSingleton {
    private static final long serialVersionUID = 1L;

    // 싱글톤 인스턴스
    private static final SubSingleton INSTANCE = new
SubSingleton();
    // 하위 클래스 고유 데이터
    private final Map<String, Integer> specificData;
    // ...

    // 상위 클래스 메서드 오버라이드
    @Override
    protected Object readResolve() {
        System.out.println("SubSingleton.readResolve() 호출");
        // 역직렬화된 객체 대신 싱글톤 인스턴스 반환
        return INSTANCE;
    }

    // 상위 클래스 메서드 오버라이드
    @Override
    protected Object writeReplace() {
        System.out.println("SubSingleton.writeReplace() 호출");
        // 직렬화용 프록시 객체 반환
        return new SubSerializationProxy(this);
    }
}
```

- writeReplace : 직렬화 과정에서 실제 객체 대신 다른 객체를 대체
- readResolve : 역직렬화 과정에서 생성된 객체를 다른 객체로 대체

➡ 하위 클래스가 자신만의 직렬화, 역직렬화를 구현하도록 하자.

꿋