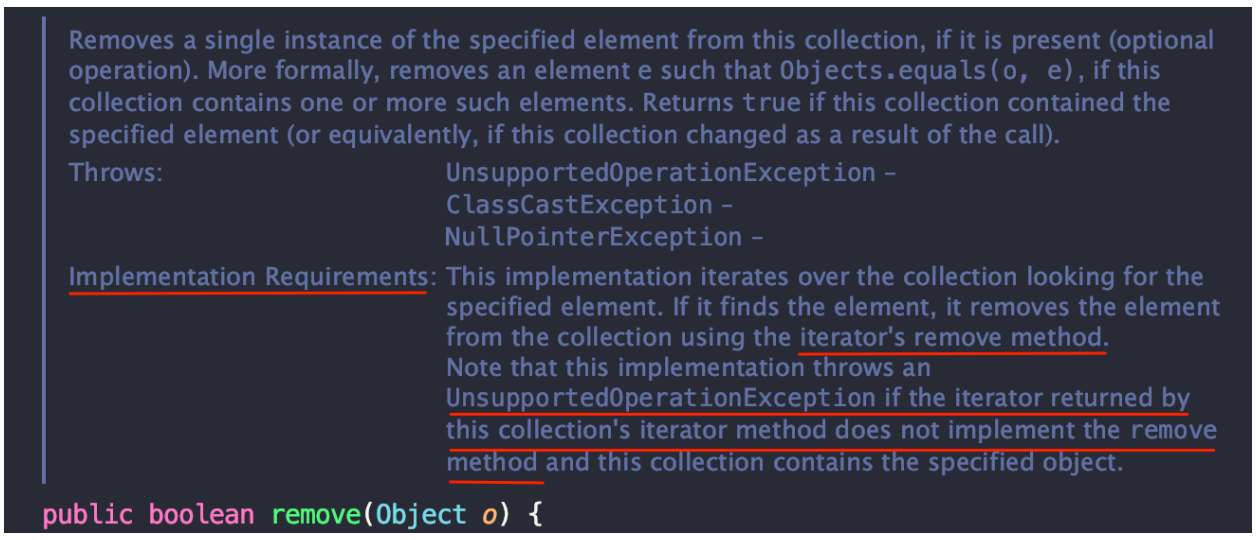
아이템 19) 상속을 고려해 설계하고 문서화하라. 그러지 않았다면 상속을 금지하라

상속용 클래스는 메서드를 재정의하면 어떤 일이 일어나는지 정확히 정리하여 문서로 남겨라

- 덧붙여서 어떤 순서로 호출하는지, 각각 호출 결과가 이어지는 처리에 어떤 영향을 주는지도 담아야 함
- 메소드 주석에 @implSpec 태그를 붙이면 자바독 도구가 생성해줌



- 하지만 이런 API 문서는 “어떻게” 가 아닌 **“무엇” 을 설명**해야 한다는 말과 대치되는데, 이는 **상속이 캡슐화를 해치기 때문에 일어나는 문제**
  - 클래스를 안전하게 상속할 수 있게 하기 위해, **내부 구현 방식을 설명해 주어야만 하는 것**
- 내부 메커니즘을 문서로 남기는 것만이 상속을 위한 설계의 전부는 아닌데, 클래스의 내부 동작 과정 중간에 끼어들 수 있는 훅을 잘 선별하여 protected 메소드 형태로 공개하는 방법도 있음

> 내부 메커니즘을 문서로 남기는 것만이 상속을 위한 설계의 전부는 아니다.
> 효율적인 하위 클래스를 어려움 없이 만들 수 있게 하려면 **클래스 내부 동작 과정 중간에 끼어들 수 있는 훅을 잘 선별하여 protected 메서드 형태로 공개해야할 수 도 있다.**
> **드물게는 protected 필드로 공개해야 할수도 있다.**

- 상속을 사용하여 **하위 클래스가 상위 클래스의 일부 동작을 변경하거나 확장**해야 할 때, 상위 클래스에서는 **하위 클래스에서 수정가능한 지점 *훅을 제공**해야 함
  - `hook 메소드` : 슈퍼클래스에서 디폴트 기능을 정의해두거나 비워뒀다가 서브클래스에서 선택적으로 오버라이드할 수 있도록 만들어둔 메소드

▼ protected 필드 예시

```
class Animal {
    protected String sound;

    public Animal() {
        sound = "일반 동물 소리";
    }

    public void makeSound() {
        System.out.println(sound);
    }
}

class Dog extends Animal {
    public Dog() {
        sound = "멍멍!";
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Animal();
```

```
        animal.makeSound(); // "일반 동물 소리" 출력

        Dog dog = new Dog();
        dog.makeSound(); // "멍멍!" 출력
    }
}
```

- Animal 클래스는 `sound` 변수를 protected로 선언하고, Dog 클래스에서 이 변수를 수정함으로써 중간에 끼어들 수 있는 훅을 제공

▼ protected 메소드 예시

```
class Animal {
    private String sound;

    public Animal(String sound) {
        this.sound = sound;
    }

    protected void makeSound() {
        System.out.println(sound);
    }
}

class Dog extends Animal {
    public Dog(String sound) {
        super(sound);
    }

    @Override
    protected void makeSound() {
       System.out.println(sound);
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Animal("일반 동물 소리");
        animal.makeSound(); // "일반 동물 소리" 출력

        Dog dog = new Dog("멍멍");
        dog.makeSound(); // "멍멍" 출력
    }
}
```

- 근데 사실상 public 으로 선언한 메서드를 오버라이딩한 거랑 똑같은 거 아닌가?
  - protected 메소드의 사용은 오버라이딩을 재고하는 훅을 명확히 하는 예시로 사용되었고, 실제로 protected 로 할 필요는 없다
  - 명확성을 위해 오버라이드하려는 메서드가 훅임을 나타내기 위해 `protected`로 선언하는 것은 일반적인 관행 중 하나

상속용 클래스 설계 시 어떤 메서드를 protected 로 노출할 지는 어떻게 결정할까?

- 실제 하위 클래스를 만들어 테스트해보는 것이 최선
- protected 메소드 하나하나가 내부 구현이므로 그 수는 가능한 적어야 함
- 상속용 클래스를 시험하는 방법은 직접 하위 클래스를 만들어 보는 것이 유일하다
  - 하위 클래스를 여러 개 만들 때까지 **전혀 쓰이지 않는 protected 멤버는 사실 private 이었어야 할 가능성이 크다**

상속용 클래스의 생성자는, 오버라이딩 가능한 메소드를 호출해서는 안된다

- 상위 클래스 생성자가 하위 클래스의 생성자보다 먼저 실행되므로, 하위 클래스에서 재정의한 메서드가 하위 클래스의 생성자보다 먼저 호출됨

Cloneable, Serializable 인터페이스를 구현한 클래스를 상속할 수 있게 설계하는 것은 좋지 않다

- clone(), readObject() 는 새로운 객체를 만들기 때문에 생성자와 비슷한 효과를 냄

    - 따라서 이 메서드들을 구현할 때 따르는 제약도, 생성자와 비슷하다
- 상속용으로 설계하지 않은 클래스는 `final` 로 선언하거나, 모든 생성자를 private / package-private 으로 선언한 후 `public 정적 팩터리`를 만들어주자!

오버라이딩이 가능한 메서드를 사용하는 코드를 제거할 수 있는 방법

```
재정의 가능 메서드는 자신의 본문 코드를 private '도우미 메서드'로 옮기고 이 도우미 메서드를 호출하도록 수정한다.
그런 다음 재정의 가능 메서드를 호출하는 다른 코드들도 모두 이 도우미 메서드를 직접 호출하도록 수정하면 된다.
```

```
class Book {
    private String title;
    private String author;

    public Book(String title, String author) {
        this.title = title;
        this.author = author;
    }

    // 재정의 가능한 메서드
    public void displayDetails() {
        System.out.println("Title: " + title);
        System.out.println("Author: " + author);
        displayAdditionalDetails(); // 도우미 메서드 호출
    }

    // 도우미 메서드
    private void displayAdditionalDetails() {
        // 이 도우미 메서드는 하위 클래스에서 필요에 따라 재정의 가능
    }
}

class AudioBook extends Book {
    private String narrator;

    public AudioBook(String title, String author, String narrator) {
        super(title, author);
        this.narrator = narrator;
    }

    // 오버라이딩하여 추가 정보 표시
    @Override
    public void displayAdditionalDetails() {
        System.out.println("Narrator: " + narrator);
    }
}

public class Main {
    public static void main(String[] args) {
        Book book = new AudioBook("Sample Book", "Sample Author", "Sample Narrator");
        book.displayDetails();
    }
}
```

1. **중복 코드 제거**
    - `displayDetails` 메서드의 일부 내용이 `Book` 클래스와 `AudioBook` 클래스에서 중복되지 않고, `displayAdditionalDetails` 도우미 메서드를 통해 공통 내용 추상화
2. **유지보수 용이성**
    - `displayAdditionalDetails` 메서드를 도우미 메서드로 사용하면, 추가 정보가 필요한 경우 하위 클래스에서 이 메서드를 오버라이드하면 됨
    - 부모 클래스인 `Book`의 `displayDetails` 메서드를 변경하지 않고도 확장 가능