

[item89] 인스턴스 수를 통제해야 한다면 readResolve 보다 열거 타입을 사용하라

싱글턴 패턴과 직렬화


싱글턴 패턴을 사용하는 클래스는 그 선언에 `implements Serializable` 을 추가하는 순간 더 이상 싱글턴이 아니게 된다.

명시적인 `writeObject()` 를 제공하더라도 소용이 없다

`transient`, `writeObject()`, `readResolve()` ?

자바 직렬화: writeObject와 readObject

자바로 구현된 시스템 간에 데이터를 주고 받는 방법으로 자바 직렬화가 있다. 직렬화하고 싶은 클래스에 `Serializable` 인터페이스만 구현(`implements`) 해주면 직렬화 가능한 클래스가 된다. 클래스에서 `transient` 또는 `static` 키워드가 선언된 필드를 제외하고는 모두 직렬화 대상이 된다. `writeObject`와


 <https://madplay.github.io/post/what-is-readobject-method-and-writeobject-method>




오늘도
MadPlay!

자바 직렬화: readResolve와 writeReplace

클래스의 객체 개수(보통 1개)를 제어하는 방법을 싱글턴 패턴이라고 한다. 객체가 여러 개 생성될 필요가 없을 때 하나만 생성하여 사용할 때마다 같은 객체를 참조하여 사용하도록 한다. 싱글턴 패턴이 적용된 클래스를 살펴보자. 하지만 싱글턴 클래스는 직렬화 가능한 클래스가 되기 위해 `Serializable` 인터페이스를 구현(`implements`) 하는 순간, 싱글턴 클래스가 아닌 상태가 된다.

 <https://madplay.github.io/post/what-is-readresolve-method-and-writereplace-method>

[이펙티브 자바 3판] 아이템 89. 인스턴스 수를 통제해야 한다면 `readResolve`보다는 열거 타입을 사용하라
앞선 아이템 3에서는 아래와 같은 싱글턴 패턴 예제를 보았다. `public static final` 필드를 사용하는 방식이다. 생성자는 `private` 접근 지정자로 선언하여 외부로부터 감추고 `INSTANCE` 를 초기화할 때 딱 한 번만 호출된다. 하지만 이 클래스는 `Serializable`을 구현하게 되는 순간 싱글턴이 아니게 된다. 기본 직렬화를 쓰지 않거나 명시적인 `readObject` 메서드를 제

 <https://madplay.github.io/post/for-instance-control-prefer-enum-types-to-readresolve>



오늘도
MadF

그렇다면 싱글턴 패턴을 사용하는 클래스를 직렬화 가능하게 하기 위해서는 어떻게 해야 할까?

1. `readResolve()`
2. 열거타입으로 만들기

readResolve()

`readResolve()` 를 인스턴스 통제 목적으로 사용할 수 있다.

단 이 경우는 객체 참조 타입 인스턴스 필드는 모두 `transient`로 선언해야 한다.

그렇지 않으면 `MutablePeriod` 공격과 비슷한 방식으로 `readResolve` 메서드가 수행되기 전에 역직렬화된 객체의 참조를 공격할 여지가 남는다 - 아이템 88

즉

다른 모든 객체 참조 타입 인스턴스 필드를 `transient` 로 선언하지 않는다면,

싱글턴 패턴을 사용하는 클래스가 직렬화가 가능해졌을 때 싱글턴임을 보장할 수 없다.

→ 다른 인스턴스 필드가 있다면 차라리 클래스를 `enum` 타입으로 만드는 것이 좋다

예를 들어

```
public final class Elvis implements Serializable {
    private static final Elvis INSTANCE = new Elvis();

    private Elvis() {
    }

    public static Elvis getInstance() {
        return INSTANCE;
    }

    private String[] favoriteSongs = { "금요일에 만나요", "좋은날" };

    public void printFavorites() {
        System.out.println(Arrays.toString(favoriteSongs));
    }

    private Object readResolve() {
        return INSTANCE;
    }
}
```

위 클래스는 favoriteSongs 를 transient 로 선언할 것이 아니라면 아래처럼 enum 타입으로 만드는 것이 좋다.

```
public enum Elvis {
    INSTANCE;
    private String[] favoriteSongs = { "금요일에 만나요", "좋은날" };

    public void printFavorites() {
        System.out.println(Arrays.toString(favoriteSongs));
    }
}
```

왜?

싱글톤이 non-transient 인 참조필드 favoriteSongs 를 가지고 있으면

이 필드의 내용은 readResolve() 가 실행되기 전에 역직렬화 된다.

그렇다면 잘 조작된 스트림을 써서

해당 참조 필드의 내용이 역직렬화되는 시점에 그 역직렬화된 인스턴스의 참조를 훑쳐 올 수 있다.

... 🙇 모르겠어요

죄송합니다

실, `readResolve`를 인스턴스 통제 목적으로 사용한다면 객체 참조 타입 인스턴스 필드는 모두 **transient**로 선언해야 한다. 그렇지 않으면 아이템 88에서 살펴본 `MutablePeriod` 공격과 비슷한 방식으로 `readResolve` 메서드가 수행되기 전에 역직렬화된 객체의 참조를 공격할 여지가 남는다.

다소 복잡한 공격 방법이지만 기본 아이디어는 간단하다. 싱글턴이 **transient**가 아닌(**non-transient**) 참조 필드를 가지고 있다면, 그 필드의 내용은 `readResolve` 메서드가 실행되기 전에 역직렬화된다. 그렇다면 잘 조작된 스트림을 써서 해당 참조 필드의 내용이 역직렬화되는 시점에 그 역직렬화된 인스턴스의 참조를 훔쳐올 수 있다.

더 자세히 알아보자. 먼저, `readResolve` 메서드와 인스턴스 필드 하나를 포함한 ‘도둑(**stealer**)’ 클래스를 작성한다. 이 인스턴스 필드는 도둑이 ‘숨길’ 직렬화된 싱글턴을 참조하는 역할을 한다. 직렬화된 스트림에서 싱글턴의 비휘발성 필드를 이 도둑의 인스턴스로 교체한다. 이제 싱글턴은 도둑을 참조하고 도둑은 싱글턴을 참조하는 순환고리가 만들어졌다.

싱글턴이 도둑을 포함하므로 싱글턴이 역직렬화될 때 도둑의 `readResolve` 메서드가 먼저 호출된다. 그 결과, 도둑의 `readResolve` 메서드가 수행될 때 도둑의 인스턴스 필드에는 역직렬화 도중인 (그리고 `readResolve`가 수행되기 전인) 싱글턴의 참조가 담겨 있게 된다.

도둑의 `readResolve` 메서드는 이 인스턴스 필드가 참조한 값을 정적 필드로 복사하여 `readResolve`가 끝난 후에도 참조할 수 있도록 한다. 그런 다음 이 메서드는 도둑이 숨긴 **transient**가 아닌 필드의 원래 타입에 맞는 값을 반환한다. 이 과정을 생략하면 직렬화 시스템이 도둑의 참조를 이 필드에 저장하려 할 때 VM이 `ClassCastException`을 던진다.

(475, 476 이펙티브자바 3권)

잘못 만들어진 클래스

```
public final class Elvis implements Serializable {
    private static final Elvis INSTANCE = new Elvis();

    private Elvis() {
    }

    public static Elvis getInstance() {
        return INSTANCE;
    }

    private String[] favoriteSongs = { "금요일에 만나요", "좋은날" }; // non-transient

    public void printFavorites() {
        System.out.println(Arrays.toString(favoriteSongs));
    }

    private Object readResolve() {
        return INSTANCE;
    }
}
```

도둑 클래스

```
public class ElvisStealer implements Serializable {
    static Elvis impersonator; // 따라쟁이, 도둑이 숨길 직렬화된 `싱글턴`을 참조하는 역할
    private Elvis payload;

    private Object readResolve() {
        // resolve 되기 전의 Elvis 인스턴스의 참조를 저장한다.
        impersonator = payload;

        // favoritesongs 필드에 맞는 타입의 객체를 반환한다.
        return new String[] { "수퍼비와", "냉탕에 상어" }
    }

    private static final long serialVersionUID = 0;
}
```

```
public class ElvisImpersonator {

    // 진짜 Elvis 인스턴스로는 만들어 질 수 없는 바이트 스트림
    private static final byte[] serializedForm = { (byte)0xac, ... };

    public static void main(String[] args) {
        // ElvisStealer.implicitator 를 초기화한 다음,
        // 진짜 Elvis(즉, Elvis.INSTANCE)를 반환한다.
        Elvis elvis = (Elvis) deserialize(serializedForm);

        Elvis impersonator = ElvisStealer.implicitator;

        elvis.printFavorites(); // ["금요일에 만나요", "좋은날"]
        impersonator.printFavorites(); // ["수퍼비와", "냉탕에 상어"]
    }

    static Object deserialize(byte[] sf) {
        try {
            return new ObjectInputStream(new ByteArrayInputStream(sf)).readObject();
        } catch (IOException | ClassNotFoundException e) {
            throw new IllegalArgumentException(e);
        }
    }
}
```

```
}
```