EJ

이펙티브 자바 CP.11 직렬화

작성 일시	@2023년 4월 2일 오후 9:48
최종 편집 일시	@2023년 5월 23일 오전 12:09
유형	이펙티브 자바
작성자	종현 박
참석자	
언어	

11. 직렬화

▼ 85. 자바 직렬화의 대안을 찾으라

- 직렬화
  - 장점
    - 프로그래머가 어렵지 않게 분산 객체를 만들 수 있음
  - 단점
    - 보이지 않는 생성자, API와 구현 사이의 모호해진 경계, 잠재적인 정확성 문제, 성능, 보안, 유지보수성 등 대가가 큼.
    - 공격 범위가 너무 넓고 지속적으로 더 넓어져 방어하기 어렵다
      - `ObjectInputStream` 의 `readObject` 메서드를 호출하면서 객체 그래프가 역직렬화되기 때문이다.
      - `readObject` 메서드는 클래스 패스 안의 거의 모든 타입의 객체를 만들어 낼 수 있는 생성자이다.
      - 바이트 스트림을 역직렬화하는 과정에서 이 메서드는 그 타입들 안의 모든 코드를 수행할 수 있다. (타입들의 코드 전체가 공격 범위)
  - CERT 조정 센터의 기술 관리자 로버트 시커드

> 자바의 역질렬화는 명백하고 현존하는 위험이다. 이 기술은 지금도 애플리케이션에서 직접 혹은 자바 하부 시스템(RMI - Remote Method Invocation), (JMX - Java Management Extensio), (JMS - Java Messaging System) 을 통해 간접적으로 쓰이고 있기 때문이다. 신뢰할 수 없는 스트림을 역직렬화하면 원격 코드 실행, 서비스 거부 등의 공격으로 이어질 수 있다. 아무 잘 못도 없는 애플리케이션이라도 이런 공격에 취약해질 수 있다.

- 가젯
  - 자바 라이브러리와 널리 쓰이는 서드파트 라이브러리에서 직렬화 가능 타입 중 역직렬화 과정에서 호출되 잠재적으로 위험한 동작을 수행하는 메서드를 뜻함.
- 역직렬화 폭탄
  - 역직렬화에 시간이 오래 걸리는 짧은 스트림을 역직렬화하는 것만으로도 서비스 거부 공격에 쉽게 노출될 수 있는 스트림
  - ex) 역직렬화 폭탄 - 이 스트림의 역직렬화는 영원히 계속됨

```
static byte[] bomb() {
    Set<Object> root = new HashSet<>();
    Set<Object> s1 = root;
    Set<Obejct> s2 = new HashSet<>();
    for (int i = 0; i < 100; i++) {
        Set<Object> t1 = new HashSet<>();
        Set<Object> t2 = new HashSet<>();
        t1.add("foo"); // t1 과 t2 를 다르게 만든다.
        s1.add(t1); s1.add(t2);
        s2.add(t1); s2.add(t2);
        s1 = t1;
        s2 = t2;
    }
    return serialize(root); // 간결하게 하기 위해 이 메서드의 코드는 생략
}
```

    - `root` 객체 그래프는 201개의 `HashSet` 인스턴스로 구성되어, 그 각각은 3개 이하의 객체 참조를 갖는다.
    - 역직렬화시 끝나지 않는다.
      - 문제는 `HashSet` 인스턴스를 역직렬화하려면 그 원소들의 해시코드를 계산하는 데 있다. 반복문에 의해 구조의 깊이가 100단계까지 만들어진다. 따라서 이 `HashSet` 을 역직렬화하려면 `hashCode` 메서드를 2^100 번 넘게 호출 될 것이다.
      - 역직렬화가 영원히 지속되는 것도 문제지만, 무언가 잘 못되었다는 신호조차 주지 않는 것도 문제이다.
- 역직렬화의 대처법
  - **아무것도 역직렬화하지 않는 것이 가장 좋은 직렬화 위험을 회피하는 방법이다.**
  - **크로스-플랫폼 구조화된 데이터 표현**
    - 객체와 바이트 시퀀스를 변환해주는 다른 매커니즘이 있다.

        - 이 방식들은 자바 직렬화의 위험을 회피하면서 다양한 플랫폼 지원, 우수한 성능, 풍부한 지원 도구, 활발한 커뮤니티 등 수 많은 이점을 제공한다.
        - 이러한 매커니즘들도 직렬화 시스템이라 부르지만, 자바 직렬화와 구분을 해 **크로스-플랫폼 구조화된 데이터 표현**이라고 한다.
- **크로스-플랫폼 구조화된 데이터 표현**
    - 이 표현들의 공통점은 자바 직렬화보다 훨씬 간단하다.
    - 임의 객체 그래프를 자동으로 직렬화/역직렬화 하지 않는다.
    - 대신 속성-값 쌍의 집합으로 구성된 간단하고 구조화된 데이터 객체 사용.
    - 그리고 기본 타입 몇 개와 배열 타입만 지원할 뿐이다.
    - 이런 간단한 추상화만으로도 아주 강력한 분산 시스템을 구축하기에 충분하고, 자바 직렬화의 문제들을 회피 할 수 있다.
- **크로스-플랫폼 구조화된 데이터 표현**의 선두주자
  JSON, 프로토콜 버퍼 (Protocol Buffers, protobuf)
    - JSON
        - 브라우저와 서버의 통신용 설계
        - 자바스크립트용
        - 텍스트 기반 사람이 읽을 수 있음
        - 오직 데이터 표현
    - 프로토콜 버퍼 (Protocol Buffers, protobuf)
        - (구글) 서버 사이에 데이터를 교환하고 저장하기 위해 설계 (gRPC)
        - C++용
        - 이진 표현이라 효율이 높음
        - 문서를 위한 스키마를 제공하고 올바로 쓰도록 강요
- 어쩔 수 없이 자바 직렬화를 사용시
    - 신뢰할 수 없는 데이터는 절대 역직렬화하지 않는 것이다.
- 직렬화를 피할 수 없고 역직렬화한 데이터가 안전한지 완전히 확신할 수 없다면
    - 객체 역직렬화 필터링을 사용하자
        - 데이터 스트림이 역직렬화되기 전에 필터를 설치하는 기능이다.
        - 클래스 단위로, 특정 클래스를 받아들이거나 거부할 수 있다.
        - 블랙 리스트 (기본 수용 모드)
            - 기록된 잠재적으로 위험한 클래스들은 거부
        - 화이트 리스트 방식 (기본 거부 모드) 추천
            - 기록된 안전하다고 알려진 클래스만 승인
- 86 아이템 부터 직렬화 가능 클래스를 올바르고 안전하고 효율적으로 작성하려는 방법을 제시한다.

- 정리
  - 직렬화는 위험하니 피해야한다.
  - 시스템의 밑바닥 부터 설계 한다면 JSON 이나 프로토콜 버퍼 같은 대안을 사용하자
  - 신뢰할 수 없는 데이터는 역직렬화 하지 말자.
  - 꼭 한다면 객체 역직렬화 필터링을 사용하되, 모든 공격을 막아줄 수 없음을 기억하자.
  - 클래스가 직렬화를 지원하도록 만들지 말고, 꼭 만들어야겠으면 정말 신경써야한다.(아이템 86 ~ 90)

▼ 86. Serializable을 구현할지는 신중히 결정하라

- 85 에서 말했듯이, 직렬화의 대안을 찾지 못한 경우에만 적용되는 내용이다.
- 개요
  - 직렬화는 클래스 선언에서 `implements Serializable` 만 덧 붙이면 된다.
  - 그래서 굉장히 쉬워 보이지만, 길게 보면 아주 값 비싼 일이다.
  - 구현시 고려할 사항을 소개 한다.
- **Serializable 을 구현하면 릴리스 한 뒤에는 수정하기 어렵다.**
  - 구현하게 되면, 직렬화된 바이트 스트림 인코딩도 하나의 공개 API가 된다.
  - 커스텀 직렬화 형태를 설계하지 않고 자바의 기본 방식을 사용하면 직렬화 형태는 최소 적용 당시 클래스의 내부 구현 방식에 영원히 묶여버린다.
  - 달리 말하면, 기본 직렬화 형태에서는 클래스의 `private` 과 `package-private` 인스턴스 필드들 마저 API로 공개되는 꼴이 된다. (캡슐화 깨짐)
  - 릴리스 후 클래스 내부 구현을 손을 보면 원래 직렬화 형태와 달라지게 된다.
    - 한 쪽은 구버전 인스턴스를 직렬화하고 다른 쪽은 신버전 클래스로 역직렬화한다면, 실패한다.
    - 원래 직렬화 형태를 유지하면서 내부 표현을 바꿀 수도 있지만, 어렵기도 하고 소스코드에 지저분한 형태가 남겨지게 된다.
    - 그러니, 직렬화 가능 클래스를 만들고자 한다면, 길게 보고 감당할 수 있을 만큼 고품질의 직렬화 형태도 주의해서 함께 설계해야 한다. (아이템 87, 90)
  - 직렬화는 클래스 개선을 방해한다.
    - 대표적으로 스트림 고유 식별자 (직렬 버전 UID)를 들 수 있다.
    - 모든 직렬화된 클래스는 고유 식별 번호를 부여받는다.
    - `serialVersionUID` 라는 이름의 `static final long` 필드로, 이 번호를 명시하지 않으면 시스템이 런타임에 암호 해시 함수(SHA-1)를 적용해 자동으로 클래스 안에 생성해 넣는다.
    - 이 값을 생성하는 데는 클래스 이름, 구현한 인터페이스들, 컴파일러가 자동으로 생성해 넣은 것을 포함한 대부분의 클래스 멤버들이 고려된다.
    - 그래서 추후에 편의 메서드를 추가하는 식으로 이들 중 하나라도 수정한다면 직렬 버전 UID 값도 변한다.

        - 즉, 자동 생성되는 값에 의존하면 쉽게 호환성이 깨져버려 런타임에 `InvalidClassException` 이 발생한다.
- **버그와 보안 구멍이 생길 위험이 높아진다.** (아이템 85)
    - 객체는 생성자를 사용해 만드는 것이 기본이다.
      즉, 직렬화는 언어의 기본 매커니즘을 우회하는 객체 생성 기법이다.
      역직렬화는 일반 생성자의 문제가 그대로 적용되는 '숨은 생성자' 이다.
    - 기본 역직렬화를 사용하면 불변식 깨짐과 허가되지 않은 접근에 쉽게 노출된다. (아이템 88)
- **직렬화 구현 클래스의 신버전을 릴리스할 때 테스트 할 것이 늘어난다.**
    - 직렬화 가능 클래스가 수정되면 신버전 인스턴스를 직렬화한 후 구 버전으로 역직렬화 할 수 있는지, 그 반대도 가능한지를 검사해야 한다.
      양방향 직렬화/역직렬화가 모두 성공하고, 원래의 객체를 충실히 복제하는지 확인 필요
    - 테스트 양 = 직렬화 가능 클래스 수 * 릴리스 횟수 가 된다.
    - 클래스를 처음 제작할 때 커스텀 직렬화 형태를 잘 설계했다면 이러한 테스트 부담을 줄 일 수 있다. (아이템 87, 90)
- **상속용으로 설계된 클래스(아이템 19)는 대부분 `Serializable` 을 구현하면 안 되며, 인터페이스도 대부분 `Serializable` 을 확장해서는 안 된다.**
    - 이 규칙을 따르지 않으면, Serializable 을 확장, 구현 한 클래스, 인터페이스에 커다란 부담을 준다.
    - 작성하는 클래스의 인스턴스 필드가 직렬화와 확장이 모두 가능하다면 주의할 점
        - 인스턴스 필드 값 중 불변식을 보장해야 할 게 있다면 반드시 하위 클래스에서 `finalize` 메서드를 재정의하지 못하게 해야한다.
          즉, `finalize` 메서드를 자신이 재정의하면서 `final` 로 선언하면 된다.
          이렇게 하지 않으면 `finalizer` 공격 (아이템 8) 을 받을 수 있다.
        - 인스턴스 필드 중 기본값 (정수형 `0`, `boolean` 은 `false`, 객체 참조 타입은 `null`)으로 초기화되면 위배되는 불변식이 있는 클래스에 다음에 `readObjectNodata` 메서드를 반드시 추가해야 한다.
        - ex) 상태 있고, 확장 가능하고, 직렬화 가능한 클래스용 `readObjectNoData` 메서드

```
private void readObjectNoData() throws InvaildObjectException {
    throw new InvalidObjectException("스트림 데이터가 필요하다.");
}
```

- `Serialzable` 을 구현하지 않기로 할 때 주의점
    - 상속용 클래스인데 직렬화를 지원하지 않으면 그 하위 클래스에서 직렬화를 지원하려 할 때 부담이 늘어난다.
    - 보통은 이런 클래스를 역직렬화하려면 그 상위 클래스는 매개변수가 없는 생성자를 제공해야 한다는데, 이런 생성자를 제공하지 않으려면 하위 클래스에서는 어쩔 수 없이 직렬화 프록시 패턴 (아이템 90)을 사용해야 한다.
- 내부 클래스(아이템 24)는 직렬화를 구현하지 말아야 한다.

    - 내부 클래스에는 바깥 인스턴스의 참조와 유효 범위 안의 지역변수 값들을 저장하기 위해 컴파일러가 생성한 필드들이 자동으로 추가된다.
    - 내부 클래스에 대한 기본 직렬화 형태는 분명하지 않아 직렬화 구현을 하면 안 된다.
    - 단, 정적 멤버 클래스는 `Serializable` 을 구현해도 된다.
- 정리
    - `Serializable` 을 구현한다고 선언하기는 쉽지만, 굉장히 어려운 일이다.
    - 한 클래스의 여러 버전이 상호작용할 일 없고, 서버가 신뢰할 수 없는 데이터에 노출될 가능성이 없는 등, 보호된 환경에서만 쓰일 클래스가 아니라면 `Serializable` 구현은 아주 신중하게 이뤄져야 한다.
    - 상속할 수 있는 클래스라면 주의 사항이 더 많아진다.

▼ 87. 커스텀 직렬화 형태를 고려해라

- 85 에서 말했듯이, 직렬화의 대안을 찾지 못한 경우에만 적용되는 내용이다.
- 개요
    - 일반적인 직렬화 방식은 클래스에 `Serializable`을 구현하고 기본 직렬화 형태를 사용한다면 된다.
    - 기본 직렬화 형태가 아닌 방식을 사용해야 할 때는 언제인가?
- 기본 직렬화 형태 사용
    - 유연성, 성능, 정확성 측면에서 신중히 고민 후 합당할 때만 사용해야 한다.
    - 일반적으로 직접 설계하더라도 기본 직렬화 형태와 거의 같은 결과가 나올 때만 사용해야한다.
    - 객체의 물리적 표현과 논리적 내용이 같다면 기본 직렬화 형태로 봐도 된다.
        - ex) 기본 직렬화 형태에 적합한 내용

```
public class Name implements Serializable {
    /**
     * 성. null이 아니어야 함
     * @serial
     */
    private final String lastName;

    /**
     * 이름. null이 아니어야 함
     * @serial
     */
    private final String firstName;

    /**
     * 중간이름. 중간 이름이 없다면 null
     * @serial
     */
    private final String middleName;

    ...
}
```

        - 성명은 논리적으로 이름, 성, 중간이름이라는 3개의 문자열로 구성되며, 앞 코드의 인스턴스 필드들은 이 논리적 구성요소를 정확히 반영했다.
        - **기본 직렬화 형태가 적합하다고 결정했더라도 불변식 보장과 보안을 위해 `readObject` 메서드를 제공해야 할 때가 많다.**
          예시의 경우에는 `readObject` 메서드가 `lastName`과 `firstName` 필드가 `null`이 아님을 보장해야 한다
        - 세 필드 모두 `private` 임에도 문서화 주석이 달려 있다. 이 필드들은 클래스의 직렬화 형태에 포함되는 공개 API에 속해 문서화를 해야하기 때문이다. `@serial` 태그는 API 문서에서 직렬화 형태를 설명하는 특별한 페이지에 기록됨
- 기본 직렬화 형태 미사용 (미적합)
    - ex) 기본 직렬화 형태에 적합하지 않은 클래스

```
public final class StringList implements Serializable {
    private int size = 0;
    private Entry head = null;

    private static class Entry implements Serializable {
        String data;
        Entry next;
        Entry previous;
    }
    ...
}
```

        - 논리적으로 이 클래스는 일련의 문자열을 표현
        - 물리적으로는 문자열들을 이중 연결 리스트로 연결
        - 이 클래스를 기본 직렬화 형태를 사용하면 각 노드의 양뱡향 연결 정보를 포함해 모든 `Entry`를 철두철미하게 기록함으로, 기본 직렬화 형태에 적절하지 않다.
    - **객체의 물리적 표현과 논리적 표현의 차이가 클 때 기본 직렬화 형태를 사용하면 크게 네 가지 문제가 생긴다.**
        1. 공개 API 가 현재의 내부 표현 방식에 영구히 묶인다.
           `private` 클래스인 `StringList.Entry`가 공개 API가 되어 버린다. 다음 릴리즈에 내부 표현 방식을 바꾸더라도 `StringList` 클래스는 여전히 연결리스트로 표현된 입력도 처리할 수 있어야 한다.
        2. 너무 많은 공간을 차지할 수 있다.
           직렬화 형태는 연결리스트의 모든 엔트리와 연결 정보까지 기록했지만, 엔트리와 연결 정보는 내부 구현에 해당하니 직렬화 형태에 포함될 가치가 없다.
        3. 시간이 너무 많이 걸릴 수 있다.
           직렬화 로직은 객체 그래프의 위상에 관한 정보가 없으니 그래프를 직접 순회해볼 수밖에 없다.
        4. 스택 오버플로를 일으킬 수 있다.
           기본 직렬화 과정은 객체 그래프의 재귀 순회하는데, 이 작업은 중간 정도의 크기의 객체 그래프에서도 스택오버플로를 일으킬 수 있다.

- 커스텀 직렬화 형태 사용
  - `StringList` 의 합리적인 직렬화
    - 단순히 리스트가 포함한 문자열의 개수를 적은 다음, 그 뒤로 문자열들을 나열하는 수준으로 하는 것이 합리적이다.
    - 즉, 모든 객체 그래프를 탐색하는 것이 아닌, 논리적인 구성만 담기도록 한다.
  - ex) 합리적인 커스텀 직렬화 형태를 갖춘 `StringList`

```
public final class StringList implements Serializable {
    private transient int size = 0;
    private transient Entry head = null;

    // 이제 직렬화되지 않는다.
    private static class Entry {
        String data;
        Entry next;
        Entry previous;
    }

    // 지정한 문자열을 이 리스트에 추가한다.
    public final void add(String s) { ... }

    /**
     * 이 {@code StringList} 인스턴스를 직렬화한다.
     *
     * @serialData 이 리스트의 크기를 기록한 후
     * {@code int}, 이어서 모든 원소를(각각은 {@code String})
     * 순서대로 기록한다.
     */
    private void writeObject(ObjectOutputStream s) throws IOException {
        s.defaultWriteObject();
        s.writeInt(size);

        // 모든 원소를 올바른 순서로 기록한다.
        for(Entry e = head; e != null; e = e.next){
            s.writeObject(e.data);
        }
    }

    private void readObject(ObjectInputStream s) throws IOException, ClassNotFoundException {
        s.defaultReadObject();
        int numElements = s.readInt();

        // 모든 원소를 읽어 이 리스트에 삽입한다.
        for(int i = 0; i< numElements; i++){
            add((String) s.readObject());
        }
    }
    ...
}
```

    - `transient` 는 `Serialize` 하는 과정에 제외하고 싶은 경우 선언하는 키워드
    - `transient` 더라도 `writeObject` 와 `readObject` 는 각각 가장 먼저 `defaultWriteObject` 와 `defaultReadObject` 를 호출한다. 직렬화 명세는 이 작업을 무조건 하라고 명시한다. 이렇게 해야 향후 릴리즈에서 `transient` 가 아닌 필드가 추가될 경우 상호 호환되기 때문이다.
      - 신버전으로 직렬화 구버전 역직렬화 새로 추가된 필드들은 무시될 것이다.

        - 구버전 `readObject` 메서드에서 `defaultReadObject` 를 호출하지 않는다면 역직렬화 시 `StreamCorruptedException`이 발생한다.
- 추가적인 내용
    - 기본 직렬화를 사용하면 `transient` 필드는 역직렬화시 기본값으로 초기화된다. (객체 null, 숫자 0, boolean false) 기본값을 그대로 사용해서는 안 된다면 `readObject` 에서 `defaultReadObject`를 호출한 뒤, 해당 필드를 원하는 값으로 복원하자 (아이템 88)
    - 기본 직렬화를 수용하든 안하든 `defaultWriteObject` 메서드를 호출하면 `transient`로 선언하지 않은 모든 인스턴스 필드가 직렬화된다. 따라서 `transient`로 선언해도 되는 인스턴스 필드에는 모두 `transient` 한정자를 붙여야한다.
        - **객체의 논리적 상태와 무관한 필드라고 확신할 때만 `transient` 한정자를 생략해야 한다.**
    - 기본 직렬화를 수용하든 안하든, 객체의 전체 상태를 읽는 메서드에 적용해야 하는 동기화 매커니즘을 직렬화에도 적용해야 한다.
        - 예컨데 모든 메서드를 `synchronized` 로 선언하여 스레드 안전한 객체에서 기본 직렬화를 사용하려면 `writeObject` 메서드도 `synchronized`를 추가 해줘야 한다.
    - 어떤 직렬화 형태를 택하든 직렬화 가능 클래스 모두에 직렬 버전 UID를 명시하자.
        - 잠재적인 호환성 문제가 사라진다. 성능도 조금 향상된다.
        - 직렬 버전 UID 를 명시하지 않으면 이 값을 생성하는데 복잡한 연산을 수행한다.

          `private static final long serialVersionUID = <무작위 long 값>;`
        - 구버전으로 직렬화된 인스턴스들과 호환성을 끊으려는 경우가 아니라면 직렬버전 UID를 수정하지 말자.
- 정리
    - 클래스를 직렬화하기로 했다면 (아이템 86) 어떤 직렬화 형태를 사용할지 고민하자.
    - 자바의 기본 직렬화 형태는 객체를 직렬화한 결과가 해당 객체의 논리적 표현에 부합할 때만 사용하고, 그렇지 않으면 객체를 적절히 설명하는 커스텀 직렬화 형태를 고안하라.
    - 한번 공개된 메서드는 향후 릴리스에서 제거할 수 없듯이, 직렬화 형태에 포함된 필드도 마음대로 제거할 수 없다. 직렬화 호환성을 유지하기 위해 영원히 지원해야 하는 것이다.

▼ 88. readObject 메서드는 방어적으로 작성하라

- `readObject` 메서드는 실질적으로 또다른 `public` 생성자이다.
    - 따라서, 직렬화시 `readObject` 를 사용한다면, 다른 생성자와 마찬가지로 똑같은 수준으로 주의를 기울여야 한다.
    - **인수가 유효한지 검사 (아이템 49), 필요하다면 매개변수를 방어적 복사 (아이템50).**
    - `readObject` 가 제대로 수행되지 못한다면 쉽게 해당 클래스의 불변식을 깨뜨릴 수 있다.
    - `readObject` 는 매개변수로 바이트 스트림을 받는 생성자라고 할 수 있다.
        - 보통의 경우 바이트 스트림은 정상적으로 생성된 인스턴스를 직렬화해 만들어진다.
        - 하지만 불변식을 깨뜨릴 의도로 임의 생성한 바이트 스트림을 건네면 정상적인 생성자로는 만들어낼 수 없는 객체를 생성해낼 수 있기 때문이다.

- **가변 공격**
  - 다른 생성자와 같은 수준의 주의를 준다고 해서 문제가 발생하지 않는 것은 아니다.
  - 정상적인 바이트 스트림 끝에 `private` 객체 필드로의 참조를 추가하면 가변 객체 인스턴스를 만들어 낼 수 있다.
  - 공격자는 `ObjectInputStream` 에서 인스턴스 정보를 읽은 후 스트림 끝에 추가된 이 '이 악의적인 객체 참조'를 읽어 객체의 내부 정보를 얻을 수 있고 수정할 수 있어, 더는 불변이지 않게 된다.
- ex) 가변 공격 예

```
public class MutablePeriod {
    // Period 인스턴스
    public final Period period;
    // 시작 시각 필드 - 외부에서 접근 할 수 없어야 한다.
    public final Date start;
    // 종료 시각 필드 - 외부에서 접근 할 수 없어야 한다.
    public final Date end;

    public MutablePeriod() {
        try(ByteArrayOutputStream bos = new ByteArrayOutputStream();
            ObjectOutputStream out = new ObjectOutputStream(bos)) {
            // 유효한 Period 인스턴스를 직렬화한다.
            out.writeObject(new Period(new Date(), new Date()));

            /*
             * 악의적인 '이전 객체 참조', 즉 내부 Date 필드로의 참조를 추가한다.
             * 상세 내용은 자바 객체 직렬화 명세 6.4절 참조
             */
            byte[] ref = {0x71, 0, 0x7e, 0, 5}; //참조 #5
            bos.write(ref); //시작(strat)필드
            ref[4] = 4; //참조 #4
            bos.write(ref); //종료(end) 필드

            // Period 역직렬화 후 Date 참조를 '훔친다'
            ObjectInputStream in = new ObjectInputStream(
                new ByteArrayInputStream(bos.toByteArray()));
            period = (Period) in.readObjedcdt();
            start = (Date) in.readObject();
            end = (Date) in.readObject();

        } catch(IOException | ClassNotFoundException e) {
            throw new AssertionError(e);
        }
    }
}
```

```
public static void main(String[] args) {
  MutablePeriod mp = new MutablePeriod();
  Period p = mp.period;
  Date pEnd = mp.end;

  // 시간을 되돌리자.
  pEnd.setYear(78);
  pEnd.setYear(69);
}
```

  - 위 예에서 `Period` 인스턴스는 불변식을 유지한 채 생성됐지만, 의도적으로 내부의 값을 수정할 수 있었다.

    - 이처럼 인스턴스가 불변이라고 가정하는 클래스에 넘겨 엄청난 보안 문제를 일으킬 수 있다.
    - 문제의 근원은 `Period` 의 `readObject` 메서드가 방어적 복사를 충분히 하지 않은 데 있다.
    - **객체를 역직렬화할 때는 클라이언트가 소유해서는 안 되는 객체 참조를 갖는 필드를 모두 반드시 방어적으로 복사해야 한다.**
        - 따라서 `readObject` 에서는 불변 클래스 안의 `private` 가변 요소를 방어적으로 복사 해야 한다.
- ex) 방어적 복사, 유효성 검사를 수행하는 `readObject` 메서드

```
private void readObject(ObjectOutputStream s) throws IOException, ClassNotFoundException {
    s.defaultReadObject();

    //가변 요소 방어적 복사
    start = new Date(start.getTime());
    end = new Date(end.getTime());

    // 불변식 만족 검사
    if (start.compareTo(end) > 0)
        throw new InvaildObjectException(start + "가 " + end + "보다 늦다.");
}
```

    - 방어적 복사를 유효성 검사보다 앞서 수행하고, `clone` 메서드를 사용하지 않았다.
      두 조치 모두 `Period` 공격으로 보호하는데 필요하다 (아이템 50)
    - `final` 필드는 방어적 복사가 불가능하다. 그래서 `readObject` 메서드를 사용하려면 `start`, `end` 필드의 `final` 한정자를 제거해야 한다.
- 기본 `readObject` 메서드를 써도 좋을지 판단하는 방법
    - `transient` 필드를 제외한 모든 필드 값을 매개변수로 받아 유효성 검사 없이 필드에 대입하는 `public` 생성자를 추가해도 괜찮은가?
        - 아니오
          커스텀 `readObject` 메서드를 만들어 모든 유효성 검사와 방어적 복사를 수행하거나
          직렬화 프록시 패턴 (아이템 90) - 역직렬화를 안전하게 만드는데 도움이 됨.
        - 예
          기본 `readObject` 메서드 사용 가능
- **readObject 메서드도 생성자 처럼 재정의 가능 메서드를 호출해서는 안된다.**
    - 클래스의 생성자는 직접적으로든 간접적으로든 재정의 가능 메서드를 호출해서는 안 된다.(아이템 19)
    - 이 규칙을 어겼는데 해당 메서드가 재정의되면, 하위 클래스의 상태가 완전히 역직렬화되기 전에 하위 클래스에서 재정의된 메서드가 실행된다. - 프로그램 오작동 각
- 정리
    - `readObject` 메서드를 작성 시 언제나 `public` 생성자를 작성하는 것으로 해야한다.
    - `readObject` 바이트 스트림이 넘어오더라도 유효한 인스턴스를 만들어내야한다.
        - 바이트 스트림이 진짜 직렬화된 인스턴스라고 가정해서는 안된다.

        - 커스텀 직렬화를 사용했더라도 모든 문제가 그대로 발생할 수 있다.
    - 안전한 `readObject` 메서드를 작성하는 방법
        - `private` 이어야 하는 객체 참조 필드는 각 필드가 가르키는 객체를 방어적 복사하라. 불변 클래스 내의 가변 요소가 여기 속한다.
        - 모든 불변식을 검사하여 어긋나는 게 발견되면 `InvaildObjectException`을 던진다.
          방어적 복사 다음에는 반드시 불변식 검사가 뒤 따라야 한다.
        - 역직렬화 후 객체 그래프 전체의 유효성을 검사해야 한다면 `ObjectInputVaildation` 인터페이스를 사용하라. (책에 없는 내용)
        - `readObject` 메서드에서 직접적으로든 간접적으로든, 재정의 할 수 있는 메서드는 호출하지 말자.

▼ 89. 인스턴스 수를 통제해야 한다면 readResolve 보다는 열거 타입을 사용

- 싱글턴에 `implements Serializable` 을 추가하는 순간 더 이상 싱글턴이 아니다.
    - ex) 싱글턴 코드

```
public class Elvis{
    public static final Elvis INSTANCE = new Elvis();
    private Elvis(){...}

    //...
}
```

    - 기본 직렬화를 쓰지 않더라도, 명시적인 `readObject` 를 제공하더라도 싱글턴이 아니다.
    - 어떤 `readObject`를 사용하든 이 클래스가 초기화될 때 만들어진 인스턴스와 별개인 인스턴스가 생성된다.
- `readResolve` 를 이용하면 `readObject` 가 만들어낸 인스턴스를 다른 것으로 대체할 수 있다.
    - 역직렬화한 객체의 클래스가 `readResolve` 메서드를 적절히 정의해뒀다면, 역직렬화 후 새로 생성된 객체를 인수로 이 메서드가 호출되고, 이 메서드가 반환한 객체 참조가 새로 생성된 객체를 대신해 반환된다.
      (이때 생성된 객체 참조는 유지하지 않음으로 바로 GC 대상이 된다.)
    - ex) 싱글턴에 `Serialzable` 을 구현 후 `readReslove`메서드를 추가해 싱글턴 유지

```
// 인스턴스 통제를 위한 readResolve
private Object readResolve() {
// 진짜 Elvis 를 반환하고 가짜는 GC가 된다.
    return INSTANCE;
}
```

        - 이 메서드는 역직렬화 객체는 무시하고 클래스 초기화시 만들어진 싱글턴 인스턴스를 반환한다.
        - 따라서 `Elvis` 의 인스턴스의 직렬화 형태는 아무런 실 데이터를 가질 이유가 없으니 모든 인스턴스 필드른 `transient`로 선언해야한다.

        - `readResolve`를 인스턴스 통제 목적으로 사용한다면 객체 참조 타입 인스턴스 필드는 모두 `transient`로 선언해야한다.
- 싱글턴이 `transient` 가 아닌 참조 필드를 갖고 있는 경우 `readResolve` 메서드가 수행되기 전에 역직렬화된 객체의 참조를 공격할 여지를 남게된다.
    - 잘 조작된 스트림을 써서 해당 참조 필드의 내용이 역직렬화되는 시점에 그 역직렬화된 인스턴스의 참조를 훔쳐올 수 있다.
    - ex) 잘못된 싱글턴 - `transient`가 아닌 참조 필드를 갖고 있다.

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

    private String [] favoriteSongs =
        { "Hound Dog", "Heartbreak Hotel"};
    public void printFavorites() {
        System.out.println(Arrays.toString(favoriteSongs));
    }

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

    - ex) 도둑 클래스

```
public class ElvisStealer implements Serializable {
  static Elvis impersonator;
  private Elvis payload;

  private Object readResolve() {
    //resolve되기 전의 Elvis 인스턴스 참조 저장
    impersonator = payload;

    //favoriteSongs 필드에 맞는 타입의 객체 반환.
    return new String[] { "A Fool such as I" };
  }

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

    - ex) 직렬화 허점을 이용해 싱글턴 객체 2개 생성된다.

```
public class ElvisImpersonator {
  private static final byte[] serializedForm = {...};

  public static void main(String[] args) {
    // ElvisStrealer.impersonator를 초기화한 다음
    // 진짜 Elvis ( Elvis.INSTANCE)를 반환한다.
    Elvis instance = (Elvis) deserrialize(serializedForm);
    Elvis impersonator = ElvisStealer.impersonator;

    instance.printFavorites();
    impersonator.printFavorites();
  }
}
// 결과
```

```
// [Hound Dog, Heartbreak Hotel]
// [A Fool Such as I]
```

    - `favoriteSongs` 필드를 `transient`로 선언하여 고칠 수 있지만, `Elvis` 를 원소 하나짜리 열거 타입으로 바꾸는 편이 더 나은 선택이다. (아이템 3)
  - ex) 열거 타입 싱글턴 - 전통적인 싱글턴 보다 좋다.

```
public enum Elvis {
    INSTANCE;
    private String[] favoriteSongs = { "Hound Dog", "Heartbreak Hotel"};
    public void printFavorites() {
        System.out.println(Arrays.toString(favoriteSongs));
    }
}
```

- `readResolve` 메서드의 접근성은 매우 중요하다.
  - `final` 클래스라면 `readResolve` 메서드는 `private` 이어야 한다.
  - `final`이 아닌 클래스에서는 몇가지 주의점이있다.
    - `private` 으로 선언하면 하위 클래스에서 사용할 수 없다.
    - `package-private` 로 선언하면 같은 패키지에 속한 하위 클래스에서만 사용할 수 있다
    - `protected` 나 `public` 으로 선언하면 이를 재정의하지 않은 모든 하위 클래스에서 사용할 수 있다.
    - `protected` 나 `public` 이면서 하위 클래스에서 재정의하지 않았다면, 하위 클래스의 인스턴스를 역직렬화하면 상위 클래스의 인스턴스를 생성해 `ClassCastException`이 일어날 수 있다.

▼ 90. 직렬화된 인스턴스 대신 직렬화 프록시 사용을 검토하라

- **직렬화 프록시 패턴**
  - `Serializable` 을 구현하면 생성자 이외의 방법으로 인스턴스를 생성할 수 있게 되어, 버그와 보안 문제가 일어날 가능성이 커진다. 이 위험을 크게 줄여줄 방법이 직렬화 프록시 패턴이다.
  - 바깥 클래스의 논리적 상태를 표현하는 중첩 클래스를 설계해 `private static` 으로 선언한다. (중첩 클래스가 직렬화 프록시 패턴)
  - 중첩 클래스의 생성자는 단 하나여야 하며, 바깥 클래스를 매개변수로 받아야 한다.
    단순히 인수로 넘어온 인스턴스의 데이터를 복사한다.
    일관성 검사 및 방어적 복사도 필요 없다.
    다만, 바깥 클래스와 직렬화 프록시 모두 `Serializable` 을 구현한다고 선언해야한다.
  - ex) `Period` 클래스용 직렬화 프록시

```
class Period implements Serializable {
    private final Date start;
    private final Date end;

    public Period(Date start, Date end) {
        this.start = start;
        this.end = end;
    }
```

```
    private static class SerializationProxy implements Serializable {
        private static final long serialVersionUID = 2123123123;
        private final Date start;
        private final Date end;

        public SerializationProxy(Period p) {
            this.start = p.start;
            this.end = p.end;
        }

        /**
         * Deserialize 할 때 호출된다.
         * 오브젝트를 생성한다.
         */
        private Object readResolve() {
            return new Period(start, end);
        }
    }


    /**
     * 이로 인해 바깥 클래스의 직렬화된 인스턴스를 생성할 수 없다.
     * 직렬화할 때 호출되는데, 프록시를 반환하게 하고 있다.
     *
     * Serialize할 때 호출된다.
     */
    private Object writeReplace() {
        return new SerializationProxy(this);
    }

    /**
     * readObject, writeObject 가 있다면, 기본적으로 Serialization 과정에서
     * ObjectInputStream, ObjectOutputStream이 호출하게 된다.
     * 그 안에 커스텀 로직을 넣어도 된다는 것.
     */
    private void readObject(ObjectInputStream stream) throws InvalidObjectException {
        // readObject는 deserialize할 때, 그러니까 오브젝트를 만들 때인데.
        // 이렇게 해두면, 직접 Period로 역직렬화를 할 수 없는 것이다.
        throw new InvalidObjectException("프록시가 필요해요.");
    }
}
```

- 바깥 클래스의 `writeReplace` 메서드
  - 자바의 직렬화 시스템이 바깥 클래스의 인스턴스 대신 `SerializationProxy`의 인스턴스를 반환하게 하는 역할을 한다. 달리 말해, 직렬화가 이뤄지기 전에 바깥 클래스의 인스턴스를 직렬화 프록시로 변환 해준다.
- 바깥 클래스의 `readObject` 메서드
  - 에러를 던져서 공격자의 불변식 훼손 공격을 막을 수 있다.
- 중첩 클래스의 `readResolve` 메서드
  - 바깥 클래스와 논리적으로 동일한 인스턴스를 반환하는 메서드이다.
  - 역직렬화 시에 직렬화 시스템이 직렬화 프록시를 다시 바깥 클래스의 인스턴스로 변환하게 해준다.

- 직렬화 프록시 패턴의 장점
  - 가짜 바이트 스트림 공격과 내부 필드 탈취 공격을 프록시 수준에서 차단해줌.

- 예제 코드에서 본 것처럼 멤버 필드를 `final`로 선언할 수 있기 때문에 진정한 불변으로 만들 수 있다.
- 직렬화 프록시 패턴은 역직렬화한 인스턴스와 원래의 직렬화된 클래스가 달라도 정상적으로 동작한다.
  - 예로 `EnumSet`은 `public` 생성자 없이 정적 팩터리만 제공한다.
    원소 개수가 64개 이하면 `RegularEnumSet`을 사용하고 그보다 크면 `JumboEnumSet`을 사용한다.
  - 그런데, 64개짜리 원소를 가진 `EnumSet`을 직렬화한 다음에 원소 5개를 추가하고 역직렬화하면 어떻게 될까?
    - 처음 직렬화된 것은 `RegularEnumSet` 인스턴스이고
      하지만, 역직렬화는 `JumboEnumSet`으로 하면 좋을 것이다.
    - 그리고 `EnumSet`은 직렬화 프록시 패턴이 적용되어 있어 위처럼 동작한다!
  - ex) `EnumSet`의 직렬화 프록시

```
private static class SerializationProxy <E extends Enum<E>>
        implements java.io.Serializable
{
    /**
     * The element type of this enum set.
     *
     * @serial
     */
    private final Class<E> elementType;

    /**
     * The elements contained in this enum set.
     *
     * @serial
     */
    private final Enum<?>[] elements;

    SerializationProxy(EnumSet<E> set) {
        elementType = set.elementType;
        elements = set.toArray(ZERO_LENGTH_ENUM_ARRAY);
    }

    // instead of cast to E, we should perhaps use elementType.cast()
    // to avoid injection of forged stream, but it will slow the implementation
    @SuppressWarnings("unchecked")
    private Object readResolve() {
        EnumSet<E> result = EnumSet.noneOf(elementType);
        for (Enum<?> e : elements)
            result.add((E)e);
        return result;
    }

    private static final long serialVersionUID = 362491234563181265L;
}

Object writeReplace() {
    return new SerializationProxy<>(this);
}

// readObject method for the serialization proxy pattern
// See Effective Java, Second Ed., Item 78.
private void readObject(java.io.ObjectInputStream stream)
    throws java.io.InvalidObjectException {
```

```
    throw new java.io.InvalidObjectException("Proxy required");
}
```

- 직렬화 프록시 패턴의 한계
  - 클라이언트 멋대로 확장할 수 있는 클래스 (아이템 19)에는 적용할 수 없다.
  - 객체 그래프에 순환이 있는 클래스에 적용할 수 없다.
  - 위 방식에 객체의 메서드를 직렬화 프록시의 `readResolve` 안에서 호출하려면 `ClassCastException` 이 발생할 것이다.
    직렬화 프록시만 가졌을 뿐 실체 객체는 아직 만들어진 것이 아니기 때문이다.
  - 또한 강력함과 안전성을 주지만, 속도가 느리다.