



## 12장 직렬화

≡ 분류	이펙티브 자바
⦿ 태그	서적
⌚ 시작일자	@2022년 6월 21일 오후 7:03
⌚ 종료일자	@2022년 6월 27일 오후 9:21
🔗 참고	<a href="#">참고 블로그 [자바를 기술 블로그] 참고 github.ido</a>
🔗 스터디 링크	<a href="#">GitHub Link [이펙티브 자바 스터디]</a>
📖 서적	
🔗 Java	

### ▼ 목차

아이템 85. 자바 직렬화의 대안을 찾으라.

? 왜 직렬화의 대안을 찾으라는거지?

? 그럼 직렬화란?

? 바이트 스트림이란?

직렬화 사용 이유

? 왜 바이트 스트림으로 변환 하는거지?

바이트 직렬화

문자열 직렬화

객체에서의 바이트 직렬화

객체에서의 바이트 역직렬화

? 그럼 왜 직렬화가 문제가 된다는거지?

그럼 어떻게 해야 하는걸까?

정리

아이템 86. Serializable 을 구현할지는 신중히 결정하라

Serializable을 구현하면 릴리스한 뒤에는 수정하기 어렵다.

Serializable 구현은 버그와 보안 구멍이 생길 위험이 높아진다.

Serializable 구현은 해당 클래스의 신버전을 릴리스할 때 테스트할 것이 높아난다.

Serializable 구현 여부는 가볍게 결정할 사안이 아니다.

상속을 클래스가 Serializable 을 지원하지 않는 경우

내부 클래스는 직렬화를 구현하지 말아야 한다.

아이템 87. 커스텀 직렬화 형태를 고려하라.

객체의 물리적 표현과 논리적 내용이 같다면 기본 직렬화 형태라도 무방하다.

기본 직렬화를 적절하지 않게 사용할 경우 문제점

커스텀 직렬화 구조 사용

기타 주의사항

### 아이템 85. 자바 직렬화의 대안을 찾으라.

#### ? 왜 직렬화의 대안을 찾으라는거지?

직렬화는 위험하기 때문이다.

자바의 역직렬화는 명백하고 현존하는 위험이다. 이 기술은 지금도 애플리케이션에서 직접 혹은, 자바 하부 시스템(RMI(Remote Method Invocation), JMX(Java Management Extension), JMS(Java Messaging System) 같은) 을 통해 간접적으로 쓰이고 있기 때문이다. 신뢰할 수 없는 스트림을 역직렬화하면 원격 코드 실행(remote code execution, RCE), 서비스 거부(denial-of-service, DoS)등의 공격으로 이어질 수 있다.

잘못한 게 없는 애플리케이션이라도 이런 공격에 취약해질 수 있다.

#### ? 그럼 직렬화란?

넓은 의미로 직렬화는 어떤 데이터를 다른 데이터의 형태로 변환하는 것을 말합니다.

이펙티브 자바에서 말하는 직렬화( `Serializable` )란 바이트 스트림으로의 직렬화로 객체의 상태를 바이트 스트림으로 변환하는 것을 의미합니다.

반대로 바이트 스트림에서 객체의 상태로 변환하는 건 역직렬화( `Deserializable` )라고 부릅니다.

#### ? 바이트 스트림이란?

스트림은 데이터의 흐름입니다. 데이터의 통로라고도 이야기하는데요.

예를 들어, 웹 개발을 하다보면 클라이언트에서 서버에게 데이터를 보내는 일이 있습니다.

이처럼 스트림은 클라이언트와 서버같이 어떤 출발지와 목적지로 입출력하기 위한 통로를 말합니다.

자바는 이런 입출력 스트림의 기본 단위를 바이트로 두고 있고 입력으로는 `InputStream` , 출력으로는 `OutputStream` 라는 추상클래스로 구현되어 있습니다.

#### 직렬화 사용 이유

일단 자바에서 표현된 객체를 목적지에 보냈다고 했을 때 그 곳에서 아 이게 자바 객체구나~하고 바로 알 수가 없습니다. 목적지에서 객체를 알 수 있는 방법이 없습니다. 그래서 모두 다 알 수 있는 것으로 변환을 해줘야 합니다.

여기서 변환을 도와주는 방법이 직렬화이며 직렬화를 통해서 바이트 스트림으로 변환해줄 수 있습니다.

#### ? 왜 바이트 스트림으로 변환 하는거지?

바이트인 이유는 컴퓨터에서 기본으로 처리되는 최소 단위가 바이트이기 때문입니다. 완전 최소 단위로 가면 비트로도 처리할 수 있겠지만 표현할 수 있는 방법이 0과 1로 너무 적어 하나의 단위로 묶었다고 합니다.

이렇게 바이트 스트림으로 변환해야 네트워크, DB로 전송할 수 있고 목적지에서도 처리를 해줄 수 있습니다. 쉽게 생각하면 '출발지와 목적지 모두 알아들을 수 있는 byte라는 언어로 소통할 수 있도록' 이라고 말할 수 있겠네요.

#### 바이트 직렬화

```
Socket socket = new Socket();
socket.connect(new InetSocketAddress("localhost", 5001));

byte[] bytes;
String message;

OutputStream os = socket.getOutputStream(); // 출력 스트림
message = "이펙티브 자바 파이팅~";
bytes = message.getBytes(StandardCharsets.UTF_8); // 문자열 -> 바이트
os.write(bytes); // 출력 스트림에 바이트를 쓰고
os.flush(); // flush를 날리면 그 소켓으로 출력이 된다
```

#### 문자열 직렬화

```

@DisplayName("JSON 직렬화 테스트")
@Test
void jsonSerializable() {
    Person person = new Person("bingbong", 21);
    String json = String.format("{\"name\":\"%s\",\"age\":%d}", person.name, person.age);
    assertThat(json).isEqualTo("{\"name\":\"bingbong\",\"age\":21}");
}

```

## 객체에서의 바이트 직렬화

```

@DisplayName("Serializable을 구현한 Person 객체 직렬화 테스트")
@Test
void writeObjectTest() throws IOException {
    Person person = new Person("bingbong", 21);

    byte[] serializedPerson;
    try (ByteArrayOutputStream baos = new ByteArrayOutputStream()) {
        try (ObjectOutputStream oos = new ObjectOutputStream(baos)) {
            oos.writeObject(person);
            // 직렬화된 Person 객체
            serializedPerson = baos.toByteArray();
        }
    }
    // 출력 결과 예시
    // -84, -19, 0, 5, 115, 114, 0, 57, 99, 111, 109, 46, 98, 105, 110, 103, 98, 111, 110, 103, 46, 101, 102, 102, 101, 99, 116, 105, 118, 101, 106, 97, 118, 97, 46, 105, 116, 101, 109, 56, 53, 46, 83, 101, 114, 105, 97, 108,
    assertThat(serializedPerson).isNotEmpty();
}

static class Person implements Serializable {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

```

## 객체에서의 바이트 역직렬화

```

@DisplayName("Serializable을 구현한 Person 객체 직렬화 후, 역직렬화 테스트")
@Test
void writeObjectTest2() throws IOException {
    Person person = new Person("bingbong", 21);

    byte[] serializedPerson;
    try (ByteArrayOutputStream baos = new ByteArrayOutputStream()) {
        try (ObjectOutputStream oos = new ObjectOutputStream(baos)) {
            oos.writeObject(person);
            // 직렬화된 Person 객체
            serializedPerson = baos.toByteArray();
        }
    }

    Person deSerializedPerson = null;

    try (ByteArrayInputStream bais = new ByteArrayInputStream(serializedPerson)) {
        try (ObjectInputStream ois = new ObjectInputStream(bais)) {
            // 역직렬화된 Person 객체
            deSerializedPerson = (Person) ois.readObject();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

    assertThat(deSerializedPerson).isNotNull();
    assertThat(deSerializedPerson.name).isEqualTo("bingbong");
    assertThat(deSerializedPerson.age).isEqualTo(21);
}

```

## ? 그럼 왜 직렬화가 문제가 된다는거지?

직렬화의 근본적인 문제는 공격 범위가 너무 넓고 지속적으로 더 넓어져 방어하기 어렵다는 점입니다.  
직렬화를 하고 나서 역직렬화를 할 때 문제가 됩니다.

- 객체를 읽는 readObject 메서드는 클래스 패스에 존재하는 거의 모든 타입의 객체를 만들어낼 수 있습니다.
  - 반환 타입이 Object입니다
- 바이트 스트림을 역직렬화하는 과정에서 해당 타입 안의 모든 코드를 수행할 수 있습니다.
  - 객체를 아래 불러올 수 있으므로 모든 코드를 수행할 수 있습니다.
- 그렇기에 타입 전체가 전부 공격 범위에 들어가게 됩니다.
- 용량도 다른 포맷에 비해서 몇 배 이상의 크기를 가집니다
- 또한 역직렬화 폭탄을 맞을 수도 있습니다.



바이트 스트림으로 직렬화하는데는 시간이 별로 걸리지 않지만 역직렬화를 잘못하면 안으로 들어가서 hashCode 메서드를 계속 호출해야하기때문에 시간이 엄청 오래걸립니다.

```

@DisplayName("역직렬화 폭탄 테스트")
@Test
void deserializeBomb() {
    byte[] bomb = bomb();

    // 역직렬화를 하면 엄청 많은 시간이 걸린다.
    deserialize(bomb);
    assertThat(bomb).isNotEmpty();
}

static byte[] bomb() {
    Set<Object> root = new HashSet<>();
    Set<Object> s1 = root;
    Set<Object> s2 = new HashSet<>();
    for (int i = 0; i < 1000; i++) {
        Set<Object> t1 = new HashSet<>();
        Set<Object> t2 = new HashSet<>();
        t1.add("foo");
        s1.add(t1);
        s1.add(t2);
        s2.add(t1);
        s2.add(t2);
        s1 = t1;
        s2 = t2;
    }
    return serialize(root); // 직렬화 수행
}

```



그럼 어떻게 해야 하는걸까?

☀ 가장 좋은 방법은 아무것도 역직렬화하지 않는 것이라고 합니다.  
직렬화를 피할 수 없고 역직렬화한 데이터가 안전하게 완전히 확신할 수 없다면 java 9에 나온 `ObjectInputFilter`를 사용하는 것도 방법입니다. 이는 데이터 스트림이 역직렬화되기 전에 필터를 적용해서 특정 클래스를 받아들이거나 거부할 수 있습니다.

## 정리

직렬화는 어떤 데이터를 다른 데이터의 형태로 변환하는 것이다  
신뢰할 수 없는 역직렬화하는 것 자체가 스스로를 공격에 노출하는 행위다.  
역직렬화를 해야한다면 `ObjectInputFilter`를 사용하자

## 아이템 86. Serializable 을 구현할지는 신중히 결정하라

어떤 클래스의 인스턴스를 직렬화 하기위해선 `Serializable` 을 impl 하게 된다면 쉽게 사용 할 수 있다.  
하지만 지원하는게 쉬워보이지만 길게 보면 아주 값 비싼 일이다.

**Serializable을 구현하면 릴리스한 뒤에는 수정하기 어렵다.**

클래스가 `Serializable`을 구현하면 직렬화된 바이트 스트림 인코딩(직렬화형태)도 하나의 공개 API가 된다.  
그래서 이 클래스가 널리 퍼진다면 그 직렬화 형태도 영원히 지원해야 하는 것이다.  
즉 클래스의 `private` 과 `package-private` 인스턴스 필드마저 API로 공개하는 꼴이 된다.  
뒤늦게 클래스 내부 구현을 손보면 원래의 직렬화 형태와 달라지게 된다.  
한쪽은 구버전 인스턴스를 직렬화하고 다른 쪽은 신버전 클래스로 역직렬화한다면 실패를 맞볼 것이다.

## 직렬화 예제

```
@Test
@DisplayName("객체 직렬화")
void serializable() throws IOException {
    Member member = new Member(10L, "LJH", new Address("서울 여당가"));

    byte[] serializedMember;
    try (ByteArrayOutputStream baos = new ByteArrayOutputStream()) {
        try (ObjectOutputStream oos = new ObjectOutputStream(baos)) {
            oos.writeObject(member);
            // serializedMember -> 직렬화된 member 객체
            serializedMember = baos.toByteArray();
        }
    }
    // 바이트 배열로 생성된 직렬화 데이터를 base64로 변환
    System.out.println(Base64.getEncoder().encodeToString(serializedMember));
}
```

```
/Library/Java/JavaVirtualMachines/jdk-11.0.13.jdk/Contents/Home/bin/java ...
r00ABXNyABtLZmZlY3RpdmVKYXZlLm10ZW04Ni5NZW1iZXl0vBqaZ1A3eAIAA0kAA2FnZUwAB2FkZHJlc3N0ABJmF2YS9sYW5nL1N0cmLuZztMAARuYWYwY1lc
QB+AAF4cAAAABl0AAbshJzsmrh0AANMSkg=
```

## 역직렬화 예제


```
@Test
@DisplayName("객체 역직렬화")
void deserializable() throws IOException, ClassNotFoundException {
    String base64Member = getSerializedMember(); // 앞에서 직렬화한 값을 바인딩해주기
    byte[] serializedMember = Base64.getDecoder().decode(base64Member);
    Member member;
    try (ByteArrayInputStream bais = new ByteArrayInputStream(serializedMember)) {
        try (ObjectInputStream ois = new ObjectInputStream(bais)) {
            // 역직렬화된 Member 객체를 읽어온다.
            Object objectMember = ois.readObject();
            member = (Member) objectMember;
        }
    }
    assertThat(member.getName()).isEqualTo("LJH");
    assertThat(member.getAge()).isEqualTo(25);
    assertThat(member.getAddress()).isEqualTo("서울*");
}
```

모든 테스트가 통과한다. 하지만 나중에 `Member` 클래스에 필드가 추가되거나 타입이 변경될 경우 오류가 나게 된다.

- 모든 직렬화된 클래스는 `serialVersionUID` 이 이름으로 고유 식별번호를 부여 받는다.
- 하지만 `serialVersionUID` 를 클래스내에 `static final long` 필드로 이번호를 명시하지 않으면 시스템이 런타임에 암호해시 함수(SHA-1) 을 적용해 자동으로 클래스 안에 생성한다.
- 그래서 나중에 클래스를 수정하게 된다면 `serialVersionUID` 값도 변하게된다. 따라서 `serialVersionUID` 을 꼭 명시 해줘야 합니다.

```
public class Member implements Serializable {
    // serialVersionUID 적 명시 할 것 !
    private static final long serialVersionUID = 1L;
    private String name;
```

```
private int age;
private String address;
}
```

 초기 버전에서 Serializable 객체가 구현하고 있다면 추후 버전에서 이전 버전에 영향없이 소스코드 수정은 매우 어렵다.(미래를 예측할 수 없기 때문에)

## Serializable 구현은 버그와 보안 구멍이 생길 위험이 높아진다.

객체는 생성자를 사용해 만드는 게 기본이다.  
즉 직렬화는 언어의 기본 메커니즘을 우회하는 객체생성 기법이다.  
기본 방식을 따르든 재정의해 사용하든, 역직렬화는 일반 생성자이 문제가 그대로 적용되는 '숨은 생성자'다.  
이 생성자는 전면에 드러나지 않으므로 "생성자에서 구축한 불변식을 모두 보장해야하고 생성 도중 공격자가 객체 내부를 들여다 볼수 없도록 해야한다" 를 떠올리기는 쉽지 않다.  
즉 기본 역직렬화를 사용하면 불변식 깨짐과 허가되지 않은 접근에 쉽게 노출된다는 뜻이다.

### 예제

- 만약 Member 가 생성될때 26 살이면 오류를 던지고 싶다.

```
public class Member implements Serializable {
    private static final long serialVersionUID = 1L;
    private String name;
    private int age;
    private String address;
    private String email;

    public Member(String name, int age, String address) {
        if(age >=26){
            throw new IllegalArgumentException();
        }
        this.name = name;
        this.age = age;
        this.address = address;
    }
}
```

- 기존 생성자 방식으로 해당 Member 객체를 생성한다면 당연히 오류를 던질 것이다.
- 하지만 직렬화를 통한 객체생성은 불변식을 무시하고 해당 객체에 26살이 바인딩되어 생성된다.

## Serializable 구현은 해당 클래스의 신버전을 릴리스할 때 테스트할 것이 늘어난다.

직렬화 가능 클래스가 수정되면 신버전인스턴스를 직렬화한 후 구버전으로 역직렬화할 수 있는지, 그리고 그 반대로 가능한지를 검사해야한다.

따라서 테스트해야 할 항목이 직렬화 가능 클래스의 수와 릴리스 횟수에 비례해 증가한다.(커스텀 직렬화 형태를 잘 설계해줬다면 이러한 테스트 부담을 줄일수 있다(아이템 87,90))

## Serializable 구현 여부는 가볍게 결정할 사안이 아니다.

단 객체를 전송하거나 저장할 때 자바 직렬화를 이용하는 프레임워크용 으로 만든 클래스라면 선택의 여지가 없다.  
Serializable 을 반드시 구현해야 하는 다른 클래스의 컴포넌트로 쓰일 클래스도 마찬가지다.  
하지만 Serializable 구현에 따른 비용이 적지 않으니, 클래스를 설계 할때마다 그이득과 비용을 잘 저울질해야 한다.

역사적으로 BigInteger 와 Instant 같은 '값' 클래스와 컬렉션 클래스들은 Serializable을 구현하고 스레드 풀처럼 '동작' 하는 객체를 표현하는 클래스 들은 대부분 Serializable 을 구현하지 않았다.

## 상속용 클래스가 Serializable 을 지원하지 않는 경우

상속용 클래스가 직렬화를 지원하지 않으면 그 하위 클래스에서 직렬화를 지원하려 할 때 부담이 늘어난다.  
보통 이런 클래스를 역직렬화하려면 그 상위 클래스는 매개변수가 없는 생성자를 제공해야한다.  
만약 지원하지 않는다면 하위 클래스는 어쩔 수 없이 직렬화 프록시 패턴(아이템90) 을 사용해야 한다.

## 내부 클래스는 직렬화를 구현하지 말아야 한다.

내부 클래스에는 바깥 인스턴스의 참조와 유효 범위 안의 지역변수 값들을 저장하기 위해 컴파일러가 생성한 필드들이 자동으로 추가된다.  
익명 클래스와 지역 클래스의 이름을 짓는 규칙이 언어 명세에 나와 있지 않듯, 이 필드들이 클래스 정의에 어떻게 추가되는지도 정의되지 않았다.  
다시 말해 내부 클래스에 대한 기본 직렬화 형태는 분명하지 않다.  
단 정적 멤버 클래스는 Serializable 을 구현해도 된다.

## 아이템 87. 커스텀 직렬화 형태를 고려하라.

객체의 물리적 표현과 논리적 내용이 같다면 기본 직렬화 형태라도 무방하다.

"이상적인 직렬화 상태"

물리적 표현과 논리적 내용이 같은 상태

- 물리적 표현 → 코드로 어떻게 구현했는지
- 논리적 내용 → 실제로 어떤 것을 의미하는지

```
public class Name implements Serializable {
    private final String lastName;
    private final String firstName;
    private final String middleName;
}
```

## 기본 직렬화를 적합하지 않게 사용할 경우 문제점

```
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;
    }
    // ... 생략
}
```

- 공개API가 현재의 내부 표현 방식에 영구히 묶인다.

- 예를 들어, 향후 버전에서는 연결 리스트를 사용하지 않게 바꾸더라도 관련 처리는 필요해진다. 따라서 코드를 절대 제거할 수 없다.
2. **너무 많은 공간을 차지할 수 있다.**
    - 위의 StringList 클래스를 예로 들면, 기본 직렬화를 사용할 때 각 노드의 연결 정보까지 모두 포함될 것
    - 하지만 이런 정보는 내부 구현에 해당하고, 직렬화 형태에 가치가 없다. 네트워크로 전송하는 속도만 느려진다.
  3. **시간이 너무 많이 걸릴 수 있다.**
    - 직렬화 로직은 객체 그래프의 위상에 관한 정보를 알 수 없으니, 직접 순회할 수밖에 없다.
  4. **스택 오버플로를 일으킬 수 있다.**
    - 기본 직렬화 형태는 객체 그래프를 재귀 순회한다. 호출 정도가 많아지면 이를 위한 스택이 감당하지 못할 것이다.

## 커스텀 직렬화 구조 사용

그럼 이런 기본직렬화 구조를 사용하기 힘들 경우 커스텀 직렬화를 어떻게 사용해야 할까?

이는 단순히 모든 객체 그래프를 탐색하는게 아닌 논리적인 구성만 담도록 하는 것이다.

위에서 작성했던 StringList 클래스를 활용해보자.

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

    private static class Entry {
        String data;
        Entry next;
        Entry previous;
    }

    private void writeObject(ObjectOutputStream s) throws IOException {
        s.defaultWriteObject();
        s.writeInt(size);

        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();

        for(int i = 0; i < numElements; i++){
            add((String) s.readObject());
        }
    }
}
```

- **transient** 키워드는 필드가 직렬화 형태에 포함되지 않는다는 표시이다.
- 클래스의 필드가 모두 **transient** 라도 writeObject, readObject메서드는 defaultXXXObject 메서드를 호출한다. 직렬화 명세에서는 이 작업을 무조건 하라고 명시한다. 이렇게 해야 향후 릴리즈에서 **transient** 가 아닌 필드가 추가될 경우 상호 호환되기 때문이다.
- **writeObject** 메서드는 private이지만, 직렬화 형태에 포함되어 공개 API라 할 수 있고, 공개 API는 모두 문서화해야 하기 때문이다.

## 기타 주의사항

어떤 직렬화 형태를 사용하던 주의해야 할 부분들도 있다.

- transient 한정자를 붙여도 되는 인스턴트 필드에는 모두 붙히도록 하자.
  - 캐시된 해시 값처럼 다른 필드에서 유도되는 필드도 포함된다.
  - JVM 실행마다 달라지는 필드(ex: long)도 포함된다.
  - 객체의 논리적 상태와 무관한 필드라고 확신할 때만 transient한정자를 생략해야 한다.
- 기본 직렬화를 사용하며 transient 필드는 기본값으로 초기화된다.
  - 객체 참조: null
  - 숫자 기본 타입: 0
  - boolean : false
  - 이런 기본값을 원치 않을 경우 readObject에서 defaultReadObject를 호출한 다음 해당 필드를 원하는 값으로 복원하면 된다.
- 객체의 전체 상태를 읽는 메서드에 적용해야 하는 동기화 메커니즘을 직렬화에도 적용해야 한다.

```
private synchronized void writeObject(ObjectOutputStream s) throws IOException {
    s.defaultWriteObject();
}
```

기본 직렬화를 사용하는 동기화된 클래스를 위한 **writeObject** 이다.

writeObject메서드 안에서 동기화가 필요한 경우 클래스의 다른 부분에서 사용하는 락 순서를 동일하게 따라야 하며, 그렇지 않을 경우 자원 순서 교착상태(resource-ordering deadlock)에 빠질 수 있다.

- 직렬화 가능 클래스에 모두 직렬 버전 UID를 명시하자.
    - 잠재적인 호환성 문제가 사라진다.
    - 성능도 조금이지만 향상된다.
- ```
private static final long serialVersionUID = <무작위로 고른 long 값>;
```
- 구버전으로 직렬화된 인스턴스들과 호환성을 끊으려는 경우가 아니라면 직렬버전 UID를 수정하지 말자.

## 11장 동시성