

1

```

    }

    private int age;
    public String name;
}

```

- 직렬화된 바이트 스트림 인코딩도 하나의 공개 API 가 된다.

```

[-84, -19, 0, 5, 115, 114, 0, 40, 101, 102, 102, 101, 99, 116, 105, 118, 101, 106, 97, 118, 97, 46, 99, 104, 97, 112, 116, 101, 114, 49

```

- Person 클래스가 널리 퍼진다면 그 직렬화 형태도 영원히 지원해야 한다. 즉 위의 바이트 코드가 역직렬화 가능하도록 영원히 (릴리즈 때마다 확인해서) 지원해줘야 한다.
- 그런데 커스텀 직렬화 형태를 설계하지 않고(특정 필드만 직렬화 한다는 등) 자바의 기본 방식을 사용해 직렬화 한 경우라면 직렬화 한 당시의 클래스의 private 필드와 package-private 인스턴스마져 API로 공개하는 꼴이 된다.
- 뒤늦게 private 필드를 숨기기 위해 Person 클래스를 수정한다고 하면 이전에 직렬화되었던 바이트 스트림 인코딩을 새로 수정한 클래스로 역직렬화 시에 문제가 발생하게 된다.

```

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

    private transient int age;
    public String name;
}

```

```

java.io.InvalidClassException: effectivejava.chapter12.item86._3.Person; local class incompatible: stream classdesc serialVersionUID = -8937441308845066542, local class serialVersionUID = -5780244527605999047

```

- 원래의 직렬화 형태를 유지하면서 내부 표현을 바꿀 수도 있지만, 어렵기도 하거니와 소스코드가 지저분해진다.
- 따라서 직렬화 가능 클래스를 만들고자 한다면, 길게 보고 감당할 수 있을 만큼 고품질의 직렬화 형태도 주의해서 함께 설계해야 한다.
- serial version UID
 - 직렬화된 클래스가 부여받는 고유 식별 번호
 - static final long field
 - 명시하지 않으면 런타임에 암호 해시 함수(SHA-1)를 적용해 자동으로 클래스 안에 생성해 넣게 되는데, 이 때 클래스 이름, 구현한 인터페이스들, 컴파일러가 자동으로 생성해 놓은 것을 포함한 대부분 클래스 멤버들이 고려된다.
 - 직렬화 할 때 적용된 UID 가 역직렬화 할 때 달라진다면 (명시하지 않으면 대부분 달라진다) 호환성이 깨져 런타임에 InvalidClassException 이 발생한다.

버그와 보안 구멍이 생길 위험이 높아진다.

- 객체는 생성자를 사용해 만드는 것이 기본인데, 역직렬화로 객체를 생성하는 방식은 언어의 기본 메커니즘을 우회하는 객체 생성 기법인 것
- 역직렬화는 숨은 생성자이다
- 생성자에서 구축한 불변식을 모두 보장해야 하고 생성 도중 공격자가 객체 내부를 들여다 볼 수 없도록 해야 한다는 사실을 떠올리기 어렵고 이는 버그와 보안 구멍으로 이어질 가능성이 있다.

해당 클래스의 신버전을 릴리스할 때 테스트할 것이 늘어난다.

- 신버전 클래스의 인스턴스를 직렬화 한 후 구버전 클래스로 역직렬화가 가능한지

- 구버전 클래스의 인스턴스를 직렬화 한 후 신버전 클래스로 역직렬화가 가능한지

테스트해야할 양 \propto 직렬화 가능한 클래스 수 * 릴리즈 횟수

Serializable 구현 여부는 가볍게 결정할 사안이 아니다.

- Serializable 구현에 따르는 비용이 적지 않으니 클래스를 설계할 때마다 그 이득과 비용을 잘 저울질 해야 한다.
- BigInteger, Instant 같은 값 클래스와 컬렉션 클래스들은 직렬화를 구현하고
- 스레드 풀처럼 동작하는 객체를 표현하는 클래스들은 대부분 직렬화를 구현하지 않았다.

→ Kotlin 의 Data class 와 Class 의 차이가 떠오름

상속용으로 설계된 클래스 대부분 Serializable 을 구현하면 안되며, 인터페이스도 대부분 Serializable 을 확장해선 안된다.

- 상속용으로 설계된 클래스나 인터페이스에서 직렬화를 구현해버리면, 해당 클래스의 자식 클래스나 인터페이스의 구현체에서 직렬화를 구현해야만 하는 큰 부담감을 지게 된다.
- 규칙을 어겨야 하는 예외 케이스
 - Serializable 을 구현한 클래스만 지원하는 프레임워크를 사용하는 경우
 - Throwable : 서버가 RMI 를 통해 클라이언트로 예외를 보내기 위해 직렬화를 구현함
 - Component : GUI 를 전송하고 저장하고 복원하기 위함
- 그럼 자식 클래스에서 직렬화를 구현하면 되느냐?
 - 자식 클래스에서 직렬화를 구현하려고 하는 경우는 상위 클래스에서 매개변수가 없는 생성자를 제공해야 한다.
 - 기본생성자가 없는 경우 직렬화 프록시 패턴을 사용해야 한다.

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

    public int age;
    public String name;
}
```

```
Exception in thread "main" java.lang.IllegalArgumentException: java.io.InvalidClassException: effectivejava.chapter12.item86._5.Student; no valid constructor
```

내부 클래스는 직렬화를 구현하지 말아야 한다.

- 내부 클래스에는 바깥 인스턴스의 참조와 유효범위 안의 지역변수들을 저장하기 위해 컴파일러가 생성한 필드들이 자동으로 추가 된다.
- 익명 클래스와 지역 클래스의 이름을 짓는 규칙이 언어 명세에 나와 있지 않듯, 이 필드들이 클래스 정의에 어떻게 추가되는지도 정의되지 않았다.
- 즉 내부 클래스에 대한 기본 직렬화 형태는 분명하지가 않다.
- 단 정적 멤버 클래스는 직렬화를 구현해도 괜찮다.