2장. 싱글턴을 만드는 세 가지 방법

이펙티브 자바의 24페이지를 참고.

개요

싱글턴 패턴이란 무엇일까?

인스턴스를 오직 하나만 생성할 수 있는 클래스이다.

장점으로는 객체나 설계상 유일해야 하는 시스템 컴포턴트를 보장할 수 있다.

단점으로는 클라이언트를 테스트하기 힘들다는 점이다.

> 왜? → 타입을 인터페이스로 정의한다음 그 인터페이스를 만든 싱글턴이 아니라면 싱글턴 인스턴스를 mock 구현으로 대체할 수 없기 때문.

2장의 싱글턴 패턴 부분에 싱글턴을 만드는 방법이 두 가지 있다고 한다.

첫번째 방법은 public static final 필드 방식의 싱글턴이다.

final이기 때문에, 초기화 과정에서 한 번만 호출된다.

private 생성자이기 때문에 만들어진 인스턴스는 시스템에서 하나뿐임이 보장된다.

하지만! 예외로 권한이 있는 클라이언트는 리플렉션API인 `AccessibleObject.setAccessible`을 사용해 private 생성자를 호출할 수 있다고 한다.

그러면 위에서 설명한 시스템에서 인스턴스가 하나뿐임을 보장한다는 말이 무색해진다.

두번째 방법은 정적팩터리 메서드를 public static 멤버로 제공하는 방법이다.

위의 public static final의 필드를 private static final로 변경하고, public static getInstance() 메서드를 제공하면 된다.

getInstance()메서드는 항상 같은 객체의 참조를 반환하므로, 제 2의 인스턴스는 만들어지지 않는다.

하지만! 리플렉션을 통한 예외는 똑같이 적용된다.

그럼 왜 정적팩터리 메서드 방식으로 싱글턴 패턴을 써야하지?

장점이 있다고 한다.

API를 바꾸지 않고도, 싱글턴이 아니게 변경할 수 있다는점.

정적 팩터리를 제네릭 싱글턴 팩토리로 만들 수 있다는점.

정적 팩토리의 메서드 참조를 공급자(Supplier)로 사용할 수 있다는점.

그래서 이러한 장점이 필요없다면? → public필드 방식이 좋다.

세번째 방법은 원소가 하나인 열거타입을 선언하는 것이다.

```
public enum Elvis{
  INSTANCE;

  public void leaveTheBuilding(){...}
}
```

열거타입은 더 간결하고, 추가 노력없이 직렬화할 수 있고, 리플랙션 공격에서도 제 2의 인스턴스가 생기는 일을 완벽히 막아준다고한다.

왜? 이게 어떻게..?

대부분 상황에서는 원소가 하나뿐인 열거 타입이 싱글턴을 만드는 가장 좋은 방법이라고 한다.

단, 싱글턴이 Enum외의 클래스를 상속해야한다면 이 방법은 사용할 수 없다.

내생각에는 싱글톤을 사용하면

1. **private 생성자를 가지고 있기 때문에 상속할 수 없다.**private 생성자는 오직 싱글톤 클래스 자신만이 자기 오브젝트를 만들도록 제한한다.문제는 private 생성자를 가진 클래스는 다른 생성자가 없다면 객체지향의 장점인 상속과 다형성 사용이 불가능하다는 점이다.
2. **싱글턴은 테스트하기가 어렵다.**클래스를 싱글턴으로 만들면 이를 사용하는 클라이언트를 테스트 하기가 어려울 수 있다.만들어지는 방식이 제한적이기 때문에 테스트에서 사용될 때 가짜 구현(mock Object)으로 대체하기가 어렵다.

3. **서버환경에서는 싱글턴이 하나만 만들어지는 것을 보장하지 못한다.**서버에서 클래스 로더를 어떻게 구성하고 있느냐에 따라서, 혹은 여러 개의 JVM에 분산되어 설치가 되는 경우에 싱글턴 클래스임에도 불구하고 하나 이상의 오브젝트가 만들어질 수 있다.
4. **싱글톤의 사용은 전역 상태를 만들 수 있기 때문에 바람직하지 못하다.**아무 객체나 자유롭게 접근하고 수정하고 공유할 수 있는 전역 상태를 갖는 것은 객체지향 프로그래밍에서는 권장되지 않는 프로그래밍 모델이다. 대신 static 필드와 메소드로만 구성된 클래스를 사용하는 것을 권장한다.

이러한 단점이 있지만, 그럼에도 사용하는 이유는 재사용성과, 메모리의 낭비 감소가 클것 같다.

Spring에서는 왜 싱글톤을 사용하는걸까? → 무거운 프로그램이라 재사용성을 중요시 여겨서 그런걸까?

과정

첫번째 방법과 두번째 방법은, 리플랙션공격에도 취약하고, 역직렬화 과정에서 가짜 Instance가 생성된다고 한다.

readResolve를 작성하면, 역직렬화과정에서 가짜 인스턴스 생성을 막을 수 있다는데, 근데 이게 정확히 어떤 상황에서 일어나는건지, 어떤 과정을 거치는지 잘모르겠다.

readResolve가 어떻게 가짜 인스턴스를 막는지, 후에 리플랙션공격이 어떻게 일어나는지도

한번 Araboza

1-1. 직렬화와 역직렬화

자바의 직렬화는 JVM의 Heap 메모리에 있는 객체 데이터를 byte stream 형태로 바꿔 외부 파일로 내보낼 수 있게 하는 기술을 말한다.

반대로 역직렬화는 외부로 내보낸 직렬화 데이터를 다시 읽어들여 자바 객체로 재변환하는 과정이다.

직렬화해서 내보낸 외부 파일은 DB에 저장되기도, Network를 통해 전송되기도 한다.

이 직렬화를 적용하기 위해선 클래스에 Serializable 인터페이스를 implements하면 된다.

하지만, p.24를 보면 싱글톤 객체를 역직렬화하는 과정에서 readResolve메서드를 제공하지 않으면, 싱글톤 패턴이 깨진다고 한다.

→ 테스트 결과, 실제로 깨진다.(자세한건 코드 확인)

1-2. 그럼 readResolve를 도입한다면?

→ 테스트 결과, 싱글톤이 보장됨을 알 수 있다.

왜? readResolve를 누가 가져다 쓰는거지..?

```
private final Object readObject(Class<?> type)
    throws IOException, ClassNotFoundException
{
    if (enableOverride) {
        return readObjectOverride();
    }

    if (! (type == Object.class || type == String.class))
        throw new AssertionError( detailMessage: "internal error");

    // if nested read, passHandle contains handle of enclosing object
    int outerHandle = passHandle;
    try {
        Object obj = readObject0(type,  unshared: false);
        handles.markDependency(outerHandle, passHandle);
        ClassNotFoundException ex = handles.lookupException(passHandle);
        if (ex != null) {
```

readObject를 까보면 readObject0() 메서드를 사용하는 것을 볼 수 있다.

```
usages
rivate Object readObject0(Class<?> type, boolean unshared) throws IOExcept
    boolean oldMode = bin.getBlockDataMode();
    if (oldMode) {
        int remain = bin.currentBlockRemaining();
        if (remain > 0) {
            throw new OptionalDataException(remain);
        } else if (defaultDataEnd) {
            /*
             * Fix for 4360508: stream is currently at the end of a field
             * value block written via default serialization; since there
             * is no terminating TC_ENDBLOCKDATA tag, simulate
             * end-of-custom-data behavior explicitly.
             */
            throw new OptionalDataException(true);
        }
```

```
            case TC_OBJECT:
                if (type == String.class) {
                    throw new ClassCastException("Cannot cast an object to
                }
                return checkResolve(readOrdinaryObject(unshared));

            case TC_EXCEPTION:
                if (type == String.class) {
                    throw new ClassCastException("Cannot cast an exception
                }
                IOException ex = readFatalException();
                throw new WriteAbortedException("writing aborted", ex);

            case TC_BLOCKDATA:
            case TC_BLOCKDATALONG:
```

readObject0메서드를 까보면 readOrdinaryObject를 통해 어떤 것을 checking하는데?

```
usage
rivate Object readOrdinaryObject(boolean unshared)
    throws IOException

    if (bin.readByte() != TC_OBJECT) {
        throw new InternalError();
    }

    ObjectStreamClass desc = readClassDesc( unshared: false);
    desc.checkDeserialize();

    Class<?> cl = desc.forClass();
    if (cl == String.class || cl == Class.class
            || cl == ObjectStreamClass.class) {
        throw new InvalidClassException("invalid class descriptor");
    }
```

```
    if (obj != null &&
        handles.lookupException(passHandle) == null &&
        desc.hasReadResolveMethod())
    {
        Object rep = desc.invokeReadResolve(obj);
        if (unshared && rep.getClass().isArray()) {
            rep = cloneArray(rep);
        }
        if (rep != obj) {
            // Filter the replacement object
            if (rep != null) {
                if (rep.getClass().isArray()) {
                    filterCheck(rep.getClass(), Array.getLength(rep));
                } else {
                    filterCheck(rep.getClass(), arrayLength: -1);
                }
```

readOrdinaryObject를 따라 들어가보면, hasReadResolveMethod를 통해 readResolve 메서드가 있는지 판단하는 것을 볼 수 있다.

JavaDocs도 한 번 읽어보자

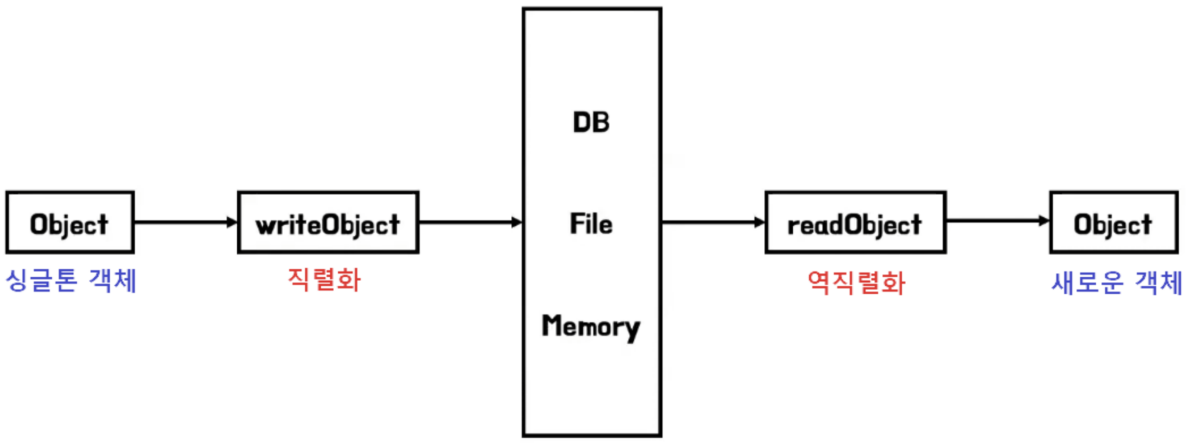
```
customized: any class-specific readObject, readObjectNoData, and readResolve
methods defined by enum types are ignored during deserialization.
---
모든 클래스에서readObject, .. , readResolve중에서 Enum 타입으로 정의된 메서드는 역직렬화 중에 무시됩니다.
```

```
the deserialized object may define a readResolve method which returns an
object visible to other parties, or readUnshared may return a Class
object or enum constant obtainable elsewhere in the stream or through
external means.
---
역직렬화된 객체는 readResolve 메서드를 정의할 수 있다.
...
```

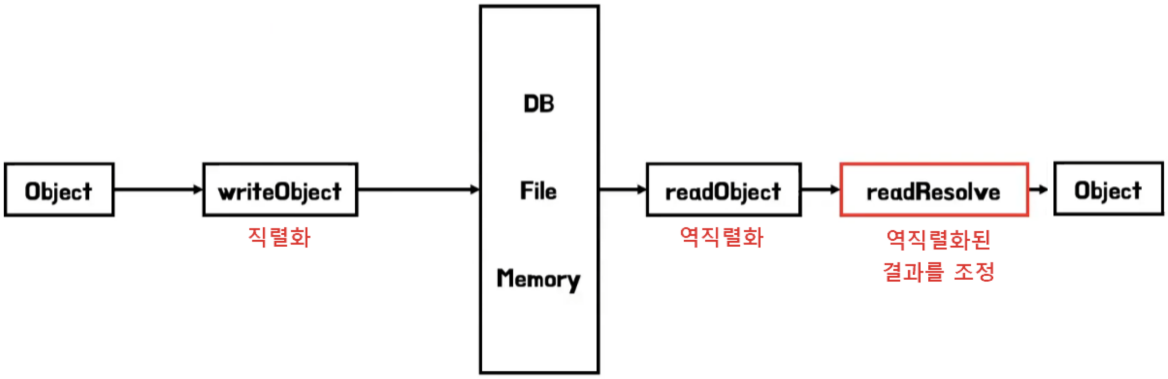
```
If the deserialized object defines a readResolve method
and the invocation of that method returns an array, then readUnshared
returns a shallow clone of that array.
---
역직렬화된 객체가 readResolve 메서드를 정의하는 경우, 해당 메서드를 호출하면 배열이 반환되고 readUnshared가 반환된다. 해당 배열은 얕은 복제본을 리턴합니다.
```

```
each object replaced by its class' {@code readResolve} method
is filtered using the replacement object's class
---
각 개체는 해당 클래스의 {@Code readResolve} 메서드로 대체됩니다.
```

그렇다면? 이 과정을 통해 알 수 있는 것은?



이 과정에서 readResolve 메서드를 클래스에 작성한다면



다음과 같은 과정으로 변한다는 것을 알 수 있다.

더 찾아보니, readObject를 통해 만들어진 인스턴스 대신 readResolve에서 반환되는 인스턴스를 내가 원하는 것으로 바꿀 수 있다고 한다. 그리고 기존에 역직렬화를 통해 새로 생성된 객체는 알아서 GC의 대상이 된다고 한다.

▼ ref

https://inpa.tistory.com/entry/JAVA-☕-싱글톤-객체-깨뜨리는-방법-역직렬화-리플렉션

싱글톤 인스턴스의 직렬화 결과에는 아무런 실 데이터를 가질 이유가 없다.

싱글톤 클래스에 필드 변수들이 있을 경우 모든 인스턴스 필드를 transient로 선언한다. 아무리 readResolve 메서드라도, 역직렬화 과정 중간에 역직렬화된 인스턴스의 참조를 훔쳐오는 공격을 행할 경우 다른 객체로 바뀔 위험이 있기 때문이다.

2-1. 자바 리플랙션

자바 리플랙션은 객체를 통해 클래스의 정보를 분석하여 런타임에 클래스의 동작을 조작하는 프로그램 기법이다.

클래스의 파일의 위치나 이름만 있다면 해당 클래스의 정보를 얻어내고 객체를 생성하는 것 또한 가능하게 해준다.

문제는 리플렉션을 통해 싱글톤 객체를 생성하게 되면, 다른 객체를 반환하게 되서 싱글톤이 깨지게 된다.

리플랙션을 왜 사용할까?

리플랙션 기법은 프레임워크, 라이브러리에서 많이 사용된다.

왜? 프레임워크, 라이브러리는 사용하는 사람이 어떤 클래스명과 멤버들을 구성할지 모르는데, 이러한 사용자 클래스들을 기존의 기능과 동적으로 연결시키기 위해서이다. Spring, Lombok 등 많은 프레임워크, 라이브러리에서 리플렉션 기능을 사용하고 있다.

→ 자바 리플렉션을 통해 생성자를 빼와 accessLevel 변경 후 싱글톤 객체를 생성했는데 생성이 된다..!

즉, 1번 방법과 2번방법은 readResolve()를 통해 역직렬화에 대응할 수 있지만, 리플랙션에 대응하지 못한다는 것을 알 수 있다..!

3-1. 열거형(Enum)

마지막 방법으로 Enum을 사용한 싱글톤 패턴이 있다고 언급되어있다.

자바에서 Enum은 일종의 클래스로 취급된다.

Enum객체를 이용하면 간단하게 싱글톤 객체를 구성할 수 있다.

왜?

enum은 멤버를 만들때 private로 만들고 한번만 초기화하기 때문이다.

→ Thread-Safe하다!

Enum내에서 상수, 변수, 메서드를 선언해 사용이 가능하기 때문

→ 이를 이용해 독립된 싱글톤 클래스 처럼 사용이 가능하다!

Enum은 기본적으로 serializable 인터페이스를 구현하고 있다. 즉, 직렬화도 가능하다!

직렬화가 필요하고, 인스턴스 수가 하나임을 보장받고 싶다면? → Enum을 고려하면 된다!

단, 싱글톤으로 구성한 클래스가 특정 클래스의 상속이 필요한 경우는 일반 클래스로 구성해야한다.

왜? enum은 같은 enum이외의 클래스 상속은 불가능하기 때문이다.

3-2. 왜 열거형을 사용해 싱글톤을 구현하면 리플렉션 공격에도 안전할까?

```
/Users/park/tool/zulu11.60.19-ca-jdk11.0.17-macosx_aarch6
리플렉션 실패: Cannot reflectively create enum objects

Process finished with exit code 0
```

→ 테스트 코드 실행시 리플렉션이 안되는 것을 확인할 수 있다.

열거형은 리플렉션을 통해 newInstance()를 실행하지 못하도록 막아놓았다고 한다.

▼ ref

https://yeonyeon.tistory.com/171

1. enum은 clone을 미지원한다.

enum은 각 인스턴스의 값이 하나씩만 존재해야하기 때문이다.

2. 역직렬화로 인한 중복 인스턴스 생성을 방지한다.

enum은 기본적으로 serializable이 가능하다.

‘serializable interface’를 구현할 필요가 없다. 즉, 역직렬화시 새로운 객체가 생성될 걱정을 안해도 된다.

3. Reflection을 이용한 enum의 인스턴스화 금지

Enum의 생성자는 Sole Constructor이다.

> It is for use by code emitted by the compiler in response to enum type declarations.

즉, 컴파일러가 사용하는 코드기 때문에 사용자가 호출해서 사용할 수 없다.

▼ ref

https://xxxelppa.tistory.com/204

컴파일러에서 사용하고 사용자가 직접 호출할 수 없다.(그래서 리플렉션이 불가능하다.)

리플렉션은 new와 동일하게 클래스 내 생성자로 인스턴스를 사용하기 때문