

[item3] private 생성자나 열거 타입으로 싱글톤임을 보증하라

[싱글톤?](#)

[언제 사용되는가?](#)

[싱글톤을 만드는 방식](#)

1. `private static final` 필드 방식
2. 정적 팩터리 방식
3. 열거 타입 방식

[싱글톤과 리플렉션](#)

[싱글톤과 직렬화 \(아이템 89\)](#)

[싱글톤은 왜 안티패턴이라 불리는가?](#)

1. 객체지향과 맞지 않다.
2. 테스트하기 어렵다.
3. 서버 환경에서는 싱글톤이 하나만 만들어지는 것을 보장하지 못한다.

[싱글톤을 사용하는 클라이언트 테스트 예시](#)

[참고 자료](#)

싱글톤?

- 인스턴스를 오직 하나만 생성할 수 있는 클래스

언제 사용되는가?

- 무상태 객체
- 설계상 유일해야 하는 시스템 컴포넌트



상태를 가진 객체를 싱글톤으로 만들면 안된다

이에 관해 멀티 스레드 환경을 생각해보자. 어플리케이션 내에 단 한 개의 인스턴스가 존재하고, 이를 전역에서 접근할 수 있다면 각기 다른 스레드에서 객체의 상태를 마구잡이로 변경시킬 여지가 있다. 상태가 공유된다는 것은 아주 위험한 상황이다.

싱글톤을 만드는 방식

1. `private static final` 필드 방식

```
public class Elvis {  
    public static final Elvis INSTANCE = new Elvis();  
  
    private Elvis() { }  
  
    public void leaveTheBuilding() {  
        System.out.println("Whoa baby, I'm outta here!");  
    }  
}
```

```
}
}
```

```
public static void main(String[] args) {
    Elvis elvis = Elvis.INSTANCE;
    elvis.leaveTheBuilding();
}
```

- private 생성자는 INSTANCE 필드를 초기화 할 때 딱 한번만 호출된다.
- public 이나 protected 생성자는 없다.
 - INSTANCE 객체가 전체 시스템에서 단 하나뿐인 Elvis 인스턴스가 된다.
- 단 예외로, 리플렉션 API인 AccessibleObject.setAccessible 을 사용해서 private 생성자를 호출할 수 있는데 이러한 공격을 방어하려면 생성자를 수정하여 두번째 객체가 생성되려 할 때 예외를 던지게 하면 된다.
- 장점
 1. 해당 클래스가 싱글톤임이 API 에 명백히 드러난다.
 - public static 필드가 final 이니 절대로 다른 객체를 참조할 수 없다.
 2. 간결하다

2. 정적 팩터리 방식

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

    private Elvis() { }

    public static Elvis getInstance() { return INSTANCE; }

    public void leaveTheBuilding() {
        System.out.println("Whoa baby, I'm outta here!");
    }
}
```

```
// This code would normally appear outside the class!
public static void main(String[] args) {
    Elvis elvis = Elvis.getInstance();
    elvis.leaveTheBuilding();
}
```

- 정적 팩터리 메서드를 public static 멤버로 제공하는 방식
- Elvis.getInstance() 는 항상 같은 객체의 참조를 반환하므로 제2의 Elvis 인스턴스란 결코 만들어지지 않는다.
- 단 이 경우도 리플렉션에 대한 예외가 발생할 수 있다.
- 장점?
 1. 추후 API를 바꾸지 않고도 싱글톤이 아니게 변경할 수 있다.
 - 예를들면 유일한 인스턴스를 반환하던 팩터리 메서드가 호출하는 스레드별로 다른 인스턴스를 넘겨주게 할 수 있다.
 2. 정적 팩터리를 제네릭 싱글턴 팩터리로 만들 수 있다 (아이템 39)

3. 정적 팩터리의 메서드 참조를 공급자로 사용할 수 있다.

예를 들어 `Elvis::getInstance` 를 `Supplier<Elvis>` 로 사용하는 식이다. (아이템 43, 44)

3. 열거 타입 방식

```
public enum Elvis {
    INSTANCE;

    public void leaveTheBuilding() {
        System.out.println("Whoa baby, I'm outta here!");
    }

    // This code would normally appear outside the class!
    public static void main(String[] args) {
        Elvis elvis = Elvis.INSTANCE;
        elvis.leaveTheBuilding();
    }
}
```

- public 필드 방식과 비슷하지만 더 간결하고 추가적인 노력 없이 직렬화 할 수 있다.
 - 리플렉션 공격에도 제2의 인스턴스가 생기는 일을 막아준다.
 - 대부분 상황에서 이 방식이 싱글톤을 만드는 가장 좋은 방법이다.
 - 단 만들고자 하는 싱글톤이 Enum 외의 클래스를 상속해야 한다면 이 방법은 사용할 수 없다.
- enum 클래스는 내부적으로 `Enum<T>` 를 상속받고 있기 때문에 다른 클래스를 상속 할 수 없다.
단, 다른 인터페이스를 구현하도록 선언할 수는 있다

싱글톤과 리플렉션

```
public class Reflection {

    public static Elvis getInstance01() {
        return Elvis.getInstance();
    }

    private static Elvis getInstance02() {
        try {
            Constructor<?> constructor = Elvis.class.getDeclaredConstructors()[0];
            constructor.setAccessible(true); // NO WAY!
            return (Elvis) constructor.newInstance();
        } catch (InstantiationException | IllegalAccessException | InvocationTargetException e) {
            e.printStackTrace();
        }
        return null;
    }

    public static void main(String[] args) {
        Elvis instance01 = getInstance01();
        System.out.println("instance01 = " + instance01.hashCode());

        Elvis instance02 = getInstance02();
        System.out.println("instance01 = " + instance02.hashCode());
    }
}
```

```
instance01 = 1802598046
instance01 = 1142020464
```

- 리플렉션 API 인 `AccessibleObject.setAccessible` 을 사용하면 `private` 생성자를 호출할 수 있게끔 할 수 있어서 전체 시스템에서 싱글턴 객체가 2개 이상 만들어 질 수 있다.
- 이런 공격에 대비하려면 `Elvis` 생성자에서 두번째 객체가 생성되려 할 때 예외를 던지게 하면 된다.

```
private Elvis() {
    if (INSTANCE != null) {
        throw new RuntimeException("Use getInstance() method to get the single instance of this class.");
    }
}
```

싱글톤과 직렬화 (아이템 89)

- `private static final` 필드 방식과 정적 팩터리 방식으로 만든 싱글턴 클래스를 직렬화 하는 방법
- 별다른 처리 없이 싱글톤 클래스를 직렬화 하면
 - (1) 직렬화된 인스턴스를
 - (2) 역직렬화할 때마다 새로운 인스턴스가 만들어진다.

```
public class Elvis implements Serializable{

    private static final Elvis INSTANCE = new Elvis();

    private Elvis() { }

    public static Elvis getInstance() { return INSTANCE; }

}
```

```
public class Serialization {

    public static byte[] serialize(Elvis elvis) {
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        try (bos; ObjectOutputStream oos = new ObjectOutputStream(bos)) {
            oos.writeObject(elvis);
        } catch (Exception e) {
            // ... 구현 생략
        }
        return bos.toByteArray();
    }

    public static Elvis deserialize(byte[] serializedData) {
        ByteArrayInputStream bis = new ByteArrayInputStream(serializedData);
        try (bis; ObjectInputStream ois = new ObjectInputStream(bis)) {
            Object object = ois.readObject();
            return (Elvis) object;
        } catch (Exception e) {
            // ... 구현 생략
        }
        return null;
    }

}
```

```

public static void main(String[] args) {
    Elvis instance01 = Elvis.getInstance();

    // (1) 직렬화
    byte[] serialize = serialize(instance01);

    // (2) 역직렬화
    Elvis instance02 = deserialize(serialize);

    System.out.println("instance01 = " + instance01.hashCode());
    System.out.println("instance02 = " + instance02.hashCode());
}
}

```

```

instance01 = 110992469
instance02 = 391359742

```

- 이 경우 단순히 Serializable 을 구현한다고 선언하는 것 이외에
 1. 모든 인스턴스를 transient 라고 선언하고
 2. readResolve 메서드를 제공해야 한다

```

public class Elvis implements Serializable{

    private static final Elvis INSTANCE = new Elvis();

    private Elvis() { }

    public static Elvis getInstance() { return INSTANCE; }

    private Object readResolve() {
        return INSTANCE;
    }

}

```

```

instance01 = 110992469
instance02 = 391359742

```

싱글톤은 왜 안티패턴이라 불리는가?

- 싱글톤 패턴은 대부분 인터페이스가 아닌 concrete class 를 미리 생성해놓고 정적 메서드를 이용해서 사용하게 된다
- 그 결과 여러 SOLID 원칙을 위반할 수 있는 가능성이 있다
- SOLID 원칙을 지키기 위해서는, 인터페이스를 사용하여 실제 구현체의 코드가 변경되더라도 이를 사용한 클라이언트 쪽 코드는 영향을 받지 않아야 하는데 싱글톤 방식은 대부분 인터페이스 방식을 사용하지 않는다
- 싱글톤을 사용하는 곳과 싱글톤 클래스 사이에 의존성이 생기게 된다.
- 클래스 사이의 강한 의존성, 높은 결합이 생기면 수정, 단위 테스트의 어려움 등의 문제가 발생한다

1. 객체지향과 맞지 않다.

- 싱글톤의 사용은 **전역상태**를 만들 수 있기 때문에 바람직하지 못하다.
- 아무 객체나 자유롭게 접근하고 수정하고 공유할 수 있는 전역 상태를 갖는 것은 객체지향 프로그래밍에서는 지양되어야 할 모델이다.
- 또한 싱글톤은 private 생성자를 갖고 있기 때문에 **상속할 수 없다**.

2. 테스트하기 어렵다.

- 싱글톤을 사용하게 되면 단위 테스트 말고도 Mock 테스트도 어려워진다!
- 타입을 인터페이스로 정의한 다음 인터페이스를 구현해서 만든 싱글톤이 아니라면 싱글톤 인스턴스를 mock 구현으로 대체할 수 없기 때문이다.
- Mockito는 정적 메서드를 mock할 수 없기 때문에 가짜(mock)를 주입하기 어렵다.
- 대신 static 메소드를 mocking할 수 있는 PowerMock같은 도구를 사용하면 가능해진다.

3. 서버 환경에서는 싱글톤이 하나만 만들어지는 것을 보장하지 못한다.

- 생성자를 private하게 두었어도 reflection을 통해 하나 이상의 오브젝트가 만들어질 수 있다.
- 또한 여러개의 JVM에 분산돼서 설치가 되는 경우에도 각각 독립적으로 오브젝트가 생기기 때문에 싱글톤으로서의 가치가 떨어진다.

싱글톤을 사용하는 클라이언트 테스트 예시

`FormatterService` (싱글톤)

```
public class FormatterService {  
  
    private static FormatterService INSTANCE;  
  
    private FormatterService() {}  
  
    public static FormatterService getInstance() {  
        if (INSTANCE == null) {  
            INSTANCE = new FormatterService();  
        }  
        return INSTANCE;  
    }  
  
    public String formatTachoIcon() {  
        return "URL";  
    }  
}
```

`DriverSnapshotHandler` (싱글톤 클라이언트)

```
public class DriverSnapshotHandler {
```

```

    public String getImageURL() {
        return FormatterService.getInstance().formatTachoIcon();
    }
}

```

테스트 코드

```

public class TestDriverSnapshotHandler {

    private FormatterService formatter;

    @Before
    public void setUp() {
        formatter = mock(FormatterService.class);
        when(FormatterService.getInstance()).thenReturn(formatter);
        when(formatter.formatTachoIcon()).thenReturn("MockedURL");
    }

    @Test
    public void testFormatterServiceIsCalled() {
        DriverSnapshotHandler handler = new DriverSnapshotHandler();
        handler.getImageURL();

        verify(formatter, atLeastOnce()).formatTachoIcon();
    }
}


```

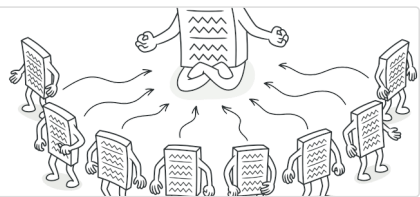
org.mockito.exceptions.misusing.MissingMethodInvocationException:
when() requires an argument which has to be 'a method call on a mock'.
For example:
when(mock.getArticles()).thenReturn(articles);

참고 자료

깊이 있는 Singleton

Singleton의 단점을 알 수 있다 : 왜 안티패턴이라 불리는가? Spring에서 Singleton의 의미를 알 수 있다 : Spring bean, Single Object, Singleton의 관계 Singleton Pattern은 GoF가 소개한 디자인 패턴 중 하나이다. 어떤 클래스를 어플리케이션 내에서 하나의 인스턴스가 존재하도록 강제하는 패턴 이다. 이렇게 하

 <https://ssoco.tistory.com/65>



Effective Java - 객체의 생성과 소멸

03 private 생성자/enum 타입을 통해 싱글톤의 특성을 유지 | Effective Java - 객체의 생성과 소멸 #03
 private 생성자나 enum 타입을 사용해서 싱글톤의 특성을 유지하자 싱글톤(Singleton)은 하나의 인스턴스
 만 생성되는 클래스입니다. 싱글톤을 구현하는 방법들에 대해 알아보겠습니다. #01. static 팩토리 메서드를

 <https://brunch.co.kr/@oemilk/118>


```

class SingletonMethod {
    private static final SingletonMethod INSTANCE = new SingletonMethod();
    private SingletonMethod() {}
    public static SingletonMethod getInstance() {
        return INSTANCE;
    }
    public void testMethod() {}
}

```

Mocking a singleton with mockito


I need to test some legacy code, which uses a singleton in a a method call. The purpose of the test is to ensure that the class under test makes a call to singletons method.

 <https://stackoverflow.com/questions/38914433/mocking-a-singleton-with-mockito>



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. writeObject와


 <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 싱글톤 패턴이라고 한다. 객체가 여러 개 생성될 필요가 없을 때 하나만 생성하여 사용할 때마다 같은 객체를 참조하여 사용하도록 한다. 싱글톤 패턴이 적용된 클래스를 살펴보자. 하지만 싱글톤 클래스는 직렬화 가능한 클래스가 되기 위해 Serializable 인터페이스를 구현(implements) 하는 순간, 싱글톤 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 readResolve보다는 열거 타입을 사용하라

앞선 아이템 3에서는 아래와 같은 싱글톤 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 Serializable을 구현하게 되는 순간 싱글톤이 아니게 된다. 기본 직렬화를 쓰지 않거나 명시적인 readObject 메서드를 제

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



오늘도
MadP