

싱글톤 패턴

양은희

정

의

싱글톤 패턴 (Singleton Pattern)

- 인스턴스를 오직 하나만 생성
- 생성된 인스턴스를 어디에서나 참조할 수 있게 함
- 최초 한 번만 메모리를 할당하고 그 메모리에 객체를 만들어 사용하는 디자인 패턴

new() : 객체를 여러 번 생성할 수 있다.

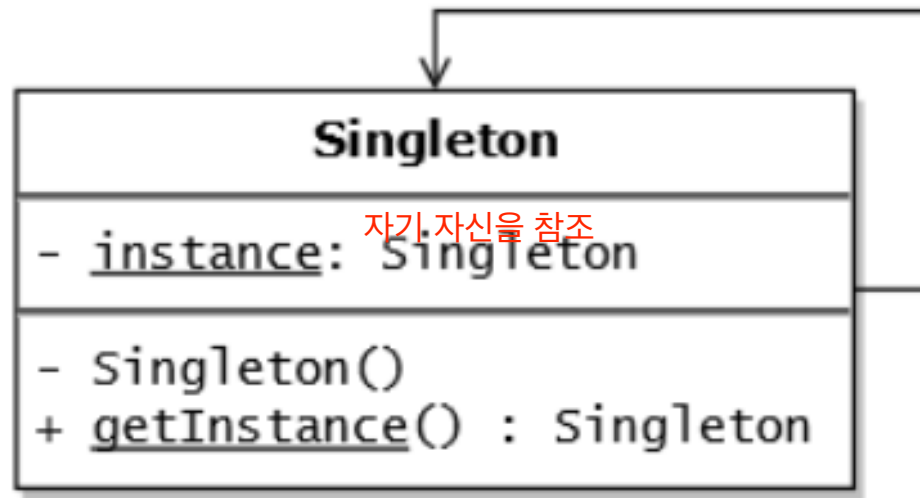
getInstance() : 하나의 인스턴스만 생성하여 이를 공유해서 쓴다.
사용자 정의 함수

Singleton pattern

자바는 클래스로 이루어져 있고 매번 새로운 객체를 생성
객체를 하나만 생성하여 이를 공유해야 하는 경우가 생긴다면 어떻게 할 것인가

정

의



목

적

1. 메모리 낭비 방지

- 고정된 메모리 영역을 얻으며 한 번의 객체 생성으로 재사용 가능

2. 전역성

- 한 번의 생성으로 전역성을 띄기 때문에 다른 객체와 공유가 용이

3. DBCP

service factory class

- DataBase Connection Pool처럼 공통된 객체를 여러 개 생성해서 사용하는 상황에서 많이 사용

4. 로딩시간 단축

- 두 번째 이용부터는 로딩 시간이 단축되어 성능이 좋아진다

화

인

```
public class SingleObj {  
    private static SingleObj singleObj = null;  
    // 외부에서 new를 못함  
    // 외부에서 직접 생성하지 못하도록 private 선언  
    private SingleObj() {}  
  
    // 오직 1개의 객체만 생성  
    public static SingleObj getInstance() {  
        if(singleObj == null) {  
            singleObj = new SingleObj();  
        }  
        return singleObj;  
    }  
}
```

```
public class Client {  
  
    public static void main(String[] args) {  
        for(int i = 0; i < 5; i++) {  
            SingleObj obj = SingleObj.getInstance();  
            System.out.println(obj.toString());  
        }  
    }  
}
```



Problems



@ Javadoc



Declaration



Git Repos

```
<terminated> Client [Java Application] C:\Program Files\Java\
singletonPattern.SingleObj@15db9742 }
singletonPattern.SingleObj@15db9742 }
singletonPattern.SingleObj@15db9742 }
singletonPattern.SingleObj@15db9742 }
singletonPattern.SingleObj@15db9742 }
```

구현 방법

1. Eager Initialization
2. Static Block Initialization
3. Lazy Initialization
4. Thread Safe Singleton
5. DCL(Double-checking Locking) initialization
6. Bill Pugh Singleton Implementation

〈공통점〉
Private 생성자
Public static getInstance()

생성자로 객체생성 못하고
메소드를 통해서만 가능

구

현

방

법

1. Eager Initialization (이른 초기화 방식)
 - 싱글톤 객체를 `instance`라는 변수로 미리 생성해놓고 사용하는 방식
 - 클래스 로딩단계에서 인스턴스 생성

```
public class Singleton {  
  
    private static Singleton instance = new Singleton();  
  
    private Singleton() {  
  
    }  
  
    public static Singleton getInstance() {  
        return instance;  
    }  
}
```

구

현

방

법

2. Static Block Initialization (정적 초기화 블록)

- Eager Initialization 에서 에러처리의 문제를 해결
- Static block을 통해서 Exception Handling에 대한 옵션 제공

```
public class Singleton {  
  
    private static Singleton instance;  
    private Singleton() {  
  
    }  
  
    static {  
        static 정적 블록 : 클래스가 실행되자마자 딱 한번 실행되게함.  
        try {  
            그렇지 않으면 낭비가 심함  
            instance = new Singleton();  
        } catch (Exception e) {  
            throw new RuntimeException ("Exception creating instance");  
        }  
    }  
  
    public static Singleton getInstance() {  
        return instance;  
    }  
}
```


구

현

방

법

3. Lazy Initializtion

- 클래스 인스턴스가 사용되는 시점에 싱글톤 인스턴스 생성

```
public class Singleton {  
  
    private static Singleton instance;  
  
    private Singleton() { }  
  
    public static Singleton getInstance() {  
        if(instance == null) {  
            instance = new Singleton();  
            클래스가 생성되는 시점에 생기므로 메모리에 적제되는게 줄어들음  
            하지만 멀티스레드로 getInstance()를 접근하면 인스턴스가 여러개 생길 위험이 있음  
        }  
        return instance;  
    }  
}
```

구

현

방

법

4. Thread Safe Singleton

- getInstance()메소드에 synchronized를 걸어두는 방식

```
public class Singleton {  
  
    private static Singleton instance;  
  
    private Singleton() { }  
  
    public static synchronized Singleton getInstance() {  
        if(instance == null) {  
            instance = new Singleton();  
        }  
        return instance;  
    }  
}
```

멀티스레드일경우의 다수의 인스턴스를 생성하는것을 방지
하지만 synchronized 사용시 시간이 오래걸림

구

현

방

법

5. DCL(Double-checking Locking) initialization

– 메소드 안에 synchronized


```
public class Singleton {
```

```
    private static Singleton instance;
```

```
    private Singleton() { }
```

```
    public static Singleton getInstance() {
```

```
        if(instance == null) {
```

```
             synchronized (Singleton.class) {
```

```
                if(instance == null) {
```

```
                    instance = new Singleton();
```

```
                } 객체를 할당받을때 synchronized하면 4번 방법보다 조금 더 효율적
```

```
            }
```

```
        }
```

```
    }
```

```
    return instance;
```

```
}
```

```
}
```

A와 B가 동시에 getInstance()까
진 접근할 수 있지만 A가 메모리를
할당하는 와중에 B가 접근시 문제
발생

구현 방법

6. Bill Pugh Singleton Implementation

– Inner static helper class **보편적인 방법**

```
public class Singleton {  
  
    private Singleton() { }  
  
    private static class SingletonHelper {  
        private static final Singleton instance = new Singleton();  
    }  
  
    public static Singleton getInstance() {  
        return SingletonHelper.instance;  
    }  
}
```

클래스 안에 클래스를 만들
싱글톤 클래스가 로드될때 헬퍼클래스가
클래스 객체를 만든다.

예

제

Q. 시스템에서 스피커에 접근하는 클래스를 개발한다.

주소값과 볼륨값이 세 객체가 모두 같음.
객체들이 같은 주소값을 참조하기때문.

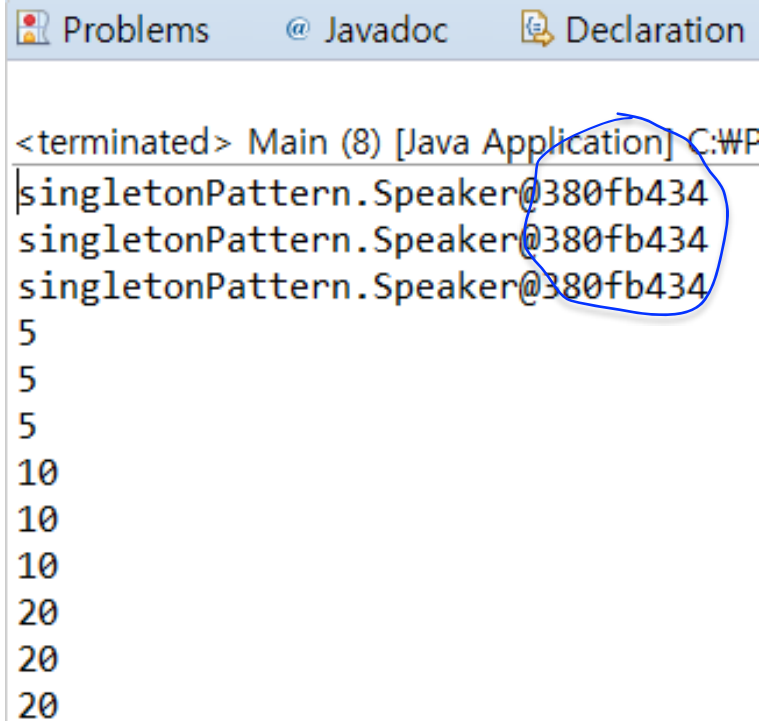
```
public class Speaker {  
  
    private static Speaker speaker;  
    private int volume;  
    private Speaker() {  
        volume = 5;  
    }  
    public static Speaker getInstance()  
    {  
        if (speaker == null)  
        {  
            speaker = new Speaker();  
        }  
        return speaker;  
    }  
    public int getVolume() {  
        return volume;  
    }  
    public void setVolume(int volume) {  
        this.volume = volume;  
    }  
}
```

```
public class Main {  
  
    public static void main(String[] args) {  
        Speaker speaker1 = Speaker.getInstance();  
        Speaker speaker2 = Speaker.getInstance();  
        Speaker speaker3 = Speaker.getInstance();  
  
        System.out.println(speaker1);  
        System.out.println(speaker2);  
        System.out.println(speaker3);  
  
        System.out.println(speaker1.getVolume());  
        System.out.println(speaker2.getVolume());  
        System.out.println(speaker3.getVolume());  
  
        speaker1.setVolume(10);  
        System.out.println(speaker1.getVolume());  
        System.out.println(speaker2.getVolume());  
        System.out.println(speaker3.getVolume());  
  
        speaker2.setVolume(20);  
        System.out.println(speaker1.getVolume());  
        System.out.println(speaker2.getVolume());  
        System.out.println(speaker3.getVolume());  
    }  
}
```

예

제

Q. 시스템에서 스피커에 접근하는 클래스를 개발한다.



```
Problems @ Javadoc Declaration
<terminated> Main (8) [Java Application] C:\WP
singletonPattern.Speaker@380fb434
singletonPattern.Speaker@380fb434
singletonPattern.Speaker@380fb434
5
5
5
10
10
10
20
20
20
```

감사합니다

양은희