230919 싱글턴 패턴

클래스 인스턴스를 하나만 만들고, 그 인스턴스의 전역 접근을 제공한다.

고전적 싱글턴 패턴 구현법

```
public class Singleton {
  private static Singleton uniqueInstance;

  private Singleton() {} // private으로 구현해서 이 클래스에서 밖에 생성안되도록 제어

  public static Singleton getInstance(){
    if(uniqueInstance == null) {
      uniqueInstance = new Singleton();
    }
    return uniqueInstance;
  }

// 기타 메소드

}
```

특징

- 원하는 시점에 객체 생성이 가능 (lazy instantiation)
- 비동기 작업을 할 때, 객체의 유일성 보장 불가 (원자성 결여)

동기화

```
public static synchronized Singleton getInstance(){

// 중략

}
```

동기화로 인해 속도문제가 생긴다면 아래의 우회방법도 고려해볼만하다.

1. 처음부터 객체 생성해두기

```
public class Singleton {
  private static Singleton uniqueInstance = new Singleton;

  private Singleton() {} // private으로 구현해서 이 클래스에서 밖에 생성안되도록 제어

  public static Singleton getInstance(){
    return uniqueInstance;
  }

// 기타 메소드

}
```

2. DCL을 사용하여 동기화 줄이기
    a. 없을시점에만 동기화를 하여 비효율성을 줄임

```
public class Singleton {
  private volatile static Singleton uniqueInstance;

  private Singleton() {} // private으로 구현해서 이 클래스에서 밖에 생성안되도록 제어

  public static Singleton getInstance(){
    if(uniqueInstance == null) {
      sychronized (Singletone class) {
        if(uniqueInstance == null) {
          uniqueInstance = new Singleton();
        }
      }
    }
    return uniqueInstance;
  }

// 기타 메소드

}
```

장단점

- 전역변수와 달리 lazy instantiation가능

- 하나뿐인 인스턴스를 어디서든 접근 가능

- 단일 책임 원칙 위배
- 느슨한 결합 원칙 위배

함께보기

싱글톤 패턴의 7가지 구현 방식

💡 출처

1. Eager Initialization
2. Static block initialization
3. Lazy initialization
4. Thread safe initialization
5. Double-Checked Locking
6. Bill Pugh Solution
7. Enum 이용

Eager Initialization

- 동기화때 사용한 방법 (p215)
- 리소스가 큰 객체라면 낭비

Static block initialization

1. 예외처리를 하지만 여전히 리소스가 크다면 낭비

```
class Singleton {
    // 싱글톤 클래스 객체를 담을 인스턴스 변수
    private static Singleton instance;

    // 생성자를 private로 선언 (외부에서 new 사용 X)
```

```
    private Singleton() {}

    // static 블록을 이용해 예외 처리
    static {
        try {
            instance = new Singleton();
        } catch (Exception e) {
            throw new RuntimeException("싱글톤 객체 생성 오류");
        }
    }

    public static Singleton getInstance() {
        return instance;
    }
}
```

Lazy initialization

- 고전적인 싱글턴 패턴 구현법 (p207)

Thread safe initialization

- 멀티스레딩 문제 해결하기 (p214)

Double-Checked Locking

- DCL 사용하기 (p216)

Bill Pugh Solution

- 권장되는 두가지 방법중 하나
- 멀티쓰레드 환경에서 안전하고 Lazy Loading(나중에 객체 생성) 도 가능한 완벽한 싱글톤 기법
- 클래스 안에 내부 클래스(holder)를 두어 JVM의 클래스 로더 매커니즘과 클래스가 로드되는 시점을 이용한 방법 (스레드 세이프함)
- static 메소드에서는 static 멤버만을 호출할 수 있기 때문에 내부 클래스를 static으로 설정이밖에도 내부 클래스의 치명적인 문제점인 메모리 누수 문제를 해결하기 위하여 내부 클래스를 static으로 설정
- 다만 클라이언트가 임의로 싱글톤을 파괴할 수 있다는 단점을 지님 (Reflection API, 직렬화/역직렬화를 통해)

```
class Singleton {
```

```
    private Singleton() {}

    // static 내부 클래스를 이용
    // Holder로 만들어, 클래스가 메모리에 로드되지 않고 getInstance 메서드가 호출되어야 로드됨
    private static class SingleInstanceHolder {
        private static final Singleton INSTANCE = new Singleton();
    }

    public static Singleton getInstance() {
        return SingleInstanceHolder.INSTANCE;
    }
}
```

1. 우선 내부클래스를 static으로 선언하였기 때문에, 싱글톤 클래스가 초기화되어도 SingleInstanceHolder 내부 클래스는 메모리에 로드되지 않음
2. 어떠한 모듈에서 `getInstance()` 메서드를 호출할 때, SingleInstanceHolder 내부 클래스의 static 멤버를 가져와 리턴하게 되는데, 이때 내부 클래스가 한번만 초기화되면서 싱글톤 객체를 최초로 생성 및 리턴하게 된다.
3. 마지막으로 final 로 지정함으로서 다시 값이 할당되지 않도록 방지한다.

Enum 이용

- enum 사용 싱글턴 (p219)

즉,

- LaszHolder : 성능이 중요시 되는 환경
- Enum : 직렬화, 안정성 중요시 되는 환경