Singleton 패턴이란

Singleton 패턴은 인스턴스를 불필요하게 생성하지 않고 오직 JVM내에서 한 개의 인스턴스만 생성하여 재사용을 위해 사용되는 디자인패턴이다. 널리 알려져 있는 디자인패턴이다, 많은 부분에서 습관적으로 반복 사용하다 보니, 특별히 이론상 생각하지 못했다.
Thread와 연관하여 Thread-safe한 Singleton 패턴에 대하여 정리 해 본다

- **초 간단 방식의 Singleton 패턴**

```
public class SingletonClass {
    private static SingletonClass singletonClass;
    private SingletonClass (){}
    public static SingletonClass getInstance() {
            if(singletonClass  == null) {
                    singletonClass = new SingletonClass ();
            }
            retuen singletonClass;
    }
}
```

일반적으로 많이 사용되는 방식이다. private static으로 자기 자신의 클래스를 객체로 선언하고 getInstance() method에서 인스턴스를 초기화 및 생성한 후 리턴하는 패턴이다.
가볍게 보면 아무런 문제가 없다. 싱글 thread환경에서 보편적으로 사용하는 코드이다.

- **멀티 쓰레드 환경에서 ThreadA와 ThreadB가 이 클래스에 동시에 접근한다고 가정을 하면**

ThreadA에서 호출하여 getInstance() 코드가 if(instance == null)지점이 실행 될때,
ThreadB가 호출을 하고 역시 getInstance() 코드가 if(instance == null)이 실행되면 instance값이 아직 null이므로 ThreadA와 ThreadB가 호출한 코드가 동시에 instance = new Singleton() 코드를 실행하게 되고 instance는 2개가 생성이 된다.

Tread A : if(singletonClass == null)수행 결과 true
Tread B : if(singletonClass == null) 수행 결과 true
Tread A : singletonClass = new SingletonClass() 수행으로 인스턴스1 생성
Tread B : singletonClass = new SingletonClass() 수행으로 인스턴스2 생성

- synchronized 를 이용한 Singleton 패턴

```
public class SingletonClass {
    private static SingletonClass singletonClass;

    private SingletonClass(){}
    public static synchronized SingletonClass getInstance() {
            if(singletonClass == null) {
                    singletonClass = new SingletonClass();
            }
            return singletonClass;
    }
}
```

쓰레드 동기화 문제의 가장 쉬운 해결방법은 synchronized키워드 이다. 단일 쓰레드가 대상 메소드를 호출시작~종료까지 다른 쓰레드가 접근하지 못하도록 lock 을 하기 때문에 위와 같이 getInstance()메소드를 synchronized로 처리하면 멀티 쓰레드에서 동시 접근으로 인한 인스턴스 중복생성 문제가 새결된다.

하지만, synchronized getinstance()의 경우 인스턴스를 리턴 받을 때마다 Thread동기화 때문에 불필요하게 lock이 걸리게 되어 비용 낭비(딜레이가 심하게 걸릴수 있다)가 크다.

실제로 고전적인 방식에서 인스턴스가 2개 이상 생성될 확률은 매우 적다. 또한 최초 instance초기화 문제 때문에 synchronized를 추가하였는데, 초기화가 완료된 시점 이후라면 synchronized는 불필요하게 lock을 잡을 뿐 별다른 역할을 하지 못한다.

- DCL(Double-Checked-Locking) Singleton 패턴

※ 이 패턴은 가급적 사용을 자제 하는 것이 좋다.

```
public class SingletonClass {

    private static SingletonClass singletonClass;
    private SingletonClass(){}

    public static SingletonClass getInstance() {
        if(singletonClass == null) {
            synchronized (SingletonClass.class) {
                if(singletonClass == null) {
                    singletonClass = new SingletonClass();
                }
            }
        }
        return singletonClass;
    }
}
```

synchronized 를 이용한 singleton 패턴의 문제를 한마디로 요약하면, 인스턴스 할당시점만 synchronized 처리되면 될 문제를 getInstance() 전체를 synchronized처리하여 성능문제를 야기한다는 점이다. 그래서 고안된 방법이DCL(Double-Checked-Locking) singleton패턴이다.

DCL singleton패턴은 getInstance() 내부에서 instance를 생성하는 경우만 부분적으로 synchronized 처리를 하여 생성과 획득을 분리한 획기적인 방법이다. 즉 인스턴스가 생성되어 있는지 확인해보고 인스턴스가 없는 경우 lock을 잡고 instance를 생성하는 방법이다.

여기에도 문제가 있다. 논리적으로는 문제가 없지만 컴파일러에 따라서 재배치(reordering)문제를 야기한다. 위에 소스가 컴파일 되는 경우 인스턴스 생성은 아래와 같은 과정을 거치게 된다.

```
public static SingletonClass getInstance() {
    if(singletonClass == null) { // Thread B 수행
        synchronized (SingletonClass.class) {
            if(singletonClass == null) {
                // singletonClass = new Singleton(); 아래와 같이 변환 됨
                some_space = allocate space for SingletonClass object;
                singletonClass = some_space; // ThreadA가 수행
                create a real object in some_space; // 실제 오브젝트 할당
            }
        }
    }
    return singletonClass;
}
```

syncronized를 사용한 멀티스레드 환경에서 각 스레드는 메모리를 공유하는 것이 아닌 JVM의 스케줄링 관리 등의 알고리즘에 따라 Thread 자신만의 메모리 캐시에 변수를 만들고 읽어온다. 이 경우 각 스레드마다 동일한 변수가 다른 값을 기억할 수 있다. ThreadA가 인스턴스 생성을 위해서 instance = some_space;를 수행하는 순간 ThreadB가 Singleton.getInstance()를 호출하게 되면 아직 실제로 인스턴스가 생성되지 않았지만, ThreadB는 instance == null 의 결과가 false로 리턴되어 문제를 야기하게 된다.

- **volatile를 이용한 개선된 DCL(Double-Checked-Locking) Singleton 패턴**

  ※ jdk 1.5이상에서 사용

```
public class SingletonClass {

    private volatile static SingletonClass singletonClass;
    private SingletonClass(){}

    public static SingletonClass getInstance() {
        if(singletonClass == null) {
            synchronized (SingletonClass.class) {
                if(singletonClass == null) {
                    singletonClass = new SingletonClass();
                }
            }
        }
        return singletonClass;
    }
}
```

volatile키워드를 이용하면 instance는 JVM의 메모리에서 변수를 참조하지 않고 하드웨어 메인 메모리에서 변수를 참조한다.

DCL singleton패턴에서와 같은 reorder문제가 발생하지 않는다. 현재까지는 안정적이고 문제가 없는 방법으로 인정되고 있다. DCL singleton패턴을 사용한다면 반드시 volatile 접근제한자를 추가하여 주도록 하자.

- **static 초기화를 이용한 Singleton 패턴**

가. static instance를 직접 선언 및 초기화

```
public class SingletonClass {

    private static SingletonClass singletonClass
                    = new Singleton(); // static 초기화시 바로 할당
    private SingletonClass(){}
    public static SingletonClass getInstance() {
            return singletonClass;
    }
}
```

나. static 변수로 선언하고 static 생성자(초기화 블록)에서 초기화

```
public class SingletonClass {
    private static SingletonClass singletonClass;
    static {
            singletonClass = new SingletonClass();
    }
    private SingletonClass(){}
    public static synchronized SingletonClass getInstance() {
            return singletonClass;
    }
}
```

이 방법은 위에서 언급되어진 멀티 쓰레드 환경에서 야기되는 문제의 상당부분을 해결한다. Thread-sfae하며 소스도 간결하고 성능역시 좋다. Thread가 getinstance()를 호출하는 시점이 아닌, Class가 로딩되는 시점. 즉 static 블록이 로딩시점에 private static SingletonClass singletonClass = new SingletonClasss(); 를 호출하여 하나의 인스턴스만 생성되는 것을 보장한다.

여기에도 문제가 있다. 실제로 사용할지 안할지 모르는 인스턴스를 미리 만들어 놓는것이 과연 옳은가 하는 문제이다. JVM이 구동환경에 충분한 메모리가 있다면 나쁘지 않은 방법이라고 생각하지만, 프로그램이 인스턴스를 필요한 시점이 아니라 사전에 생성하는 것은 메모리의 낭비라는 의견이 있다.

- LazyHolder Singleton 패턴

```
public class SingletonClass {
    private SingletonClass(){}

    public static SingletonClass getInstance() {
            return LazyHolder.INSTANCE;
    }

    private static class LazyHolder {
            private static final SingletonClass INSTANCE = new SingletonClass();
    }
}
```

현재까지 가장 완벽하다고 평가받는 방법이다. JAVA 버전과 상관 없이 사용할 수 있고, 성능도 뛰어나다.

1. private static Inner Class로 초기화 수행 코드를 선언한다. 이때 이 클래스는 static이지만 실제 실행되지 않고 대기 상태이다.
2. static 클래스를 선언하여 초기화를 담당하도록 하지만, 이영역에 초기화를 하지만 객체가 필요한시점까지 초기화를 미루는 방식이다.
3. 프로젝트가 로딩될 때 LazyHolder 클래스를 초기화하지 않는다.
4. SingletonClass 클래스의 getInstance() 메서드에서 LazyHolder.INSTANCE를 참조하는 순간 LazyHolder 클래스가 로딩되고 초기화가 실행된다.
5. Class가 로딩되며 INSTANCE 초기화가 진행된다.
6. Class를 로딩하고 초기화하는 시점은 thread-safe를 보장하기 때문에 volatile이나 synchronized 같은 키워드가 없어도 thread-safe 하면서 성능도 보장하는 아주 훌륭한 방법이다.