Java 고유 락 (Intrinsic Lock)

Intrinsic Lock / Synchronized Block / Reentrancy

Intrinsic Lock (= monitor lock = monitor) : Java의 모든 객체는 lock을 갖고 있음.

Synchronized 블록은 Intrinsic Lock을 이용해서, Thread의 접근을 제어함.

```
public class Counter {
    private int count;

    public int increase() {
        return ++count;     // Thread-safe 하지 않은 연산
    }
}
```

Q) ++count 문이 atomic 연산인가?

A) read (count 값을 읽음) -> modify (count 값 수정) -> write (count 값 저장)의 과정에서, 여러 Thread가 **공유 자원(count)으로 접근할 수 있으므로, 동시성 문제가 발생**함.

Synchronized 블록을 사용한 Thread-safe Case

```
public class Counter{
    private Object lock = new Object(); // 모든 객체가 가능 (Lock이 있음)
    private int count;

    public int increase() {
        // 단계 (1)
        synchronized(lock){ // lock을 이용하여, count 변수에의 접근을 막음
            return ++count;
        }

        /*
        단계 (2)
        synchronized(this) { // this도 객체이므로 lock으로 사용 가능
            return ++count;
        }
        */
    }
    /*
    단계 (3)
    public synchronized int increase() {
        return ++count;
```

```
    }
    */
}
```

단계 3과 같이 *lock 생성 없이 synchronized 블록 구현 가능*

Reentrancy

재진입 : Lock을 획득한 Thread가 같은 Lock을 얻기 위해 대기할 필요가 없는 것

(Lock의 획득이 '**호출 단위**'가 아닌 **Thread 단위**로 일어나는 것)

```
public class Reentrancy {
    // b가 Synchronized로 선언되어 있더라도, a 진입시 lock을 획득하였음.
    // b를 호출할 수 있게 됨.
    public synchronized void a() {
        System.out.println("a");
        b();
    }

    public synchronized void b() {
        System.out.println("b");
    }

    public static void main (String[] args) {
        new Reentrancy().a();
    }
}
```

Structured Lock vs Reentrant Lock

Structured Lock (구조적 Lock) : 고유 lock을 이용한 동기화

(Synchronized 블록 단위로 lock의 획득 / 해제가 일어나므로)

따라서,

A획득 -> B획득 -> B해제 -> A해제는 가능하지만,

A획득 -> B획득 -> A해제 -> B해제는 불가능함.

이것을 가능하게 하기 위해서는 **Reentrant Lock (명시적 Lock) 을 사용**해야 함.

Visibility

- 가시성 : 여러 Thread가 동시에 작동하였을 때, 한 Thread가 쓴 값을 다른 Thread가 볼 수 있는지, 없는지 여부
- 문제 : 하나의 Thread가 쓴 값을 다른 Thread가 볼 수 있느냐 없느냐. (볼 수 없으면 문제가 됨)
- Lock : Structure Lock과 Reentrant Lock은 Visibility를 보장.
- 원인 :

1. 최적화를 위해 Compiler나 CPU에서 발생하는 코드 재배열로 인해서.
2. CPU core의 cache 값이 Memory에 제때 쓰이지 않아 발생하는 문제.