6월 4일 복습&퀴즈

이태양


```java
public class Counter {
    private int count = 1;


    private Lock lock = new ReentrantLock();

    public void increment () {
        try {
            lock.lock();
            count++;
        }finally {
            lock.unlock();
        }
    }
    public  void decrement() {
        try {
            lock.lock();
            count--;

    }finally {
            lock.unlock();
        }
    }
    public int getCount() { return count; }
}
```

Thread를 사용할 때 Lock을 걸려면 ReentarantLock을 사용하여 재진입이 가능한 형태로 만들어줘야한다.

성공적으로 처리했던 실패했던 finally는 무조건 실행된다
그러므로 내부에 문제가 생겨도 Lock은 해제한다


```java
public class Worker implements Runnable{
    private Counter counter;
    private boolean increment;
    private int count;

    public Worker(Counter counter, boolean increment, int count){
        this.counter = counter;
        this.increment = increment;
        this.count = count;
    }

    @Override
    public void run() {
        for(int i = 0; i<this.count; i++){
            if(increment){
                    this.counter.increment();
                System.out.println("i'm increment");



            }else{

                    this.counter.decrement();
                    System.out.println("i'm decrement");


            }
        }
    }
}
```

증가스레드와 감소스레드가 어떻게 진행이되는가를 보기위한 예제

순차적으로 하나씩 서로 번갈아가면서 실행되지않는다!


```java
public class BankLockTest {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        System.out.println("First count: "+counter.getCount());

        Thread adder = new Thread(new Worker(counter, increment: true, count: 1000));
        adder.start();

        Thread subtracter = new Thread(new Worker(counter, increment: false, count: 1000));
        subtracter.start();

        adder.join();
        subtracter.join();

        System.out.println("Final count : "+counter.getCount());
    }
}
```

첫 카운트값을두고 마지막 값을 보기위한 예제

1000개씩 돌리지만 이것또한 번갈아가며 동작하지않는다는걸 알 수 있다


```java
class ParallelThread implements Runnable{
    private static int[] paralletlProcessingArr;
    final int MAX = 4;
    final int MAX_LOOP = 1000000000;
    private int threadLocalIdx;

    public ParallelThread(int threadLocalIdx){
        this.threadLocalIdx = threadLocalIdx;
        paralletlProcessingArr = new int[MAX];
        for(int i = 0 ; i <MAX; i ++){
            paralletlProcessingArr[i] = 1;
        }
    }
    @Override
    public void run() {
      for(int i = 0; i <MAX_LOOP ; i++){
          paralletlProcessingArr[threadLocalIdx]++;
          paralletlProcessingArr[threadLocalIdx]--;
      }
    }
    public static int[] getParalletlProcessingArr() {
        return paralletlProcessingArr;
    }
}
```

```java
public class ParallelConceptTest {
    public static void main(String[] args) throws InterruptedException {
        System.out.println("지금부터 병렬처리를 시작합니다 !");
        Thread[] pt = new Thread[4];
         for(int i =0; i < 4; i ++){
             pt[i] = new Thread(new ParallelThread(i));
            pt[i].start();
            pt[i].join();
         }
         for(int i = 0; i < 4; i++){
             System.out.println("최종결과 : "+ParallelThread.getParalletlProcessingArr()[i]);
         }


    }
}
```

병렬처리가 더 빠르게 처리할 수 있다는걸 보기위해 만든예제


```java
class SequenceThread {
    final int MAX = 16;
    final int MAX_LOOP = 2000000000;

    private static int[] sequenceProcessingArr;

    public SequenceThread() {
        sequenceProcessingArr = new int[MAX];

        for (int i = 0; i < MAX; i++) {
            sequenceProcessingArr[i] = 1;
        }
    }
```

```java
    public void sequenceProcessing() {
        for (int i = 0; i < MAX_LOOP; i++) {
            for (int j = 0; j < MAX_LOOP; j++)
                sequenceProcessingArr[0]++;
                sequenceProcessingArr[0]--;
                sequenceProcessingArr[1]++;
                sequenceProcessingArr[1]--;
                sequenceProcessingArr[2]++;
                sequenceProcessingArr[2]--;
                sequenceProcessingArr[3]++;
                sequenceProcessingArr[3]--;
                sequenceProcessingArr[4]++;
                sequenceProcessingArr[4]--;
                sequenceProcessingArr[5]++;
                sequenceProcessingArr[5]--;
                sequenceProcessingArr[6]++;
                sequenceProcessingArr[6]--;
                sequenceProcessingArr[7]++;
                sequenceProcessingArr[7]--;
                sequenceProcessingArr[8]++;
                sequenceProcessingArr[8]--;
                sequenceProcessingArr[9]++;
                sequenceProcessingArr[9]--;
                sequenceProcessingArr[10]++;
                sequenceProcessingArr[10]--;
                sequenceProcessingArr[11]++;
                sequenceProcessingArr[11]--;
                sequenceProcessingArr[12]++;
                sequenceProcessingArr[12]--;
                sequenceProcessingArr[13]++;
```

```java
    public static int[] getSequenceProcessingArr() {
        return sequenceProcessingArr;
    }
}

public class SequenceConceptTest {
    public static void main(String[] args) {
        System.out.println("지금부터 순차 처리를 시작합니다.");

        long startTime = System.currentTimeMillis();

        SequenceThread st = new SequenceThread();

        st.sequenceProcessing();

        for (int i = 0; i < 16; i++) {
            System.out.println("최종 결과 확인: " + SequenceThread.getSequenceProcessingArr()[i]);
        }

        long endTime = System.currentTimeMillis();

        System.out.println(endTime - startTime + "ms");
    }
}
```

병렬처리와 순차처리의 속도차이를 보기위한 예제