# [디지털 컨버전스]
# 스마트 콘텐츠와 웹 융합 응용SW 개발자 양성과정

강사 : 이상훈
학생 : 임초롱

**20회차**

# Thread

링크 https://github.com/limcholong/LectureContents/blob/main/java/CholongLim/Day19/src/ThreadTest.java

```java
import java.util.Random;

// Runnable의 경우 run() 매서드에 대한 구현이 필요함
class Car implements Runnable {

    String name;
    private int sleepTime;
    private final static Random generator = new Random();

    public Car(String name) {
        this.name = name;
        // Random 클래스로 만든 객체에 nextInt() 매서드를 통해서도 랜덤값을 생성할 수 있다.
        sleepTime = generator.nextInt( bound: 3000) + 3000;
    }

    // 스레드를 돌릴땐 무조건 이 run() 부분을 구동시키게 되어있다.
    // 매우 중요하니 이 run()을 반드시 기억해두자!
    @Override
    public void run() {
        // try라는 키워드를 적는 경우는 I/O 나 특정한 이벤트,
        // 인터럽트 등등이 발생하는 경우에 작성하게 됨
        // 이 녀석은 에러를 유발할 수도 있다! 를 암시함
        try {
            Thread.sleep(sleepTime);
        } catch (InterruptedException e) {
            // 정말 에러가 발생했다면 여기로 오게 됩니다.
            // 물론 Thread.sleep()에서 에러가 발생할 일은 99.999999999999999%로 없습니다.
            System.out.println("출력도 안 될 것이고 에러가 발생할 일이 없습니다!");
        }
        System.out.println(name + "이 경주를 완료하였습니다!");
    }
}

public class ThreadTest {
    public static void main(String[] args) {
        Thread t1 = new Thread(new Car( name: "Ferrari"));
        Thread t2 = new Thread(new Car( name: "Spyder 918"));
        Thread t3 = new Thread(new Car( name: "Maserati MC20"));

        t1.start();
        t2.start();
        t3.start();
    }
}
```

Runnable 인터페이스 (Runnable Interface) :

run( ) 매서드만 채워주면 Runnable 인터페이스를 구현하는 것은 간단하다.
하지만 Thread를 상속받은 클래스처럼 start( ) 매서드는 없다.
따라서 별도로 Thread를 생성해주고 ,
구현한 Runnable 인터페이스를 인자로 넘겨주어야 한다.

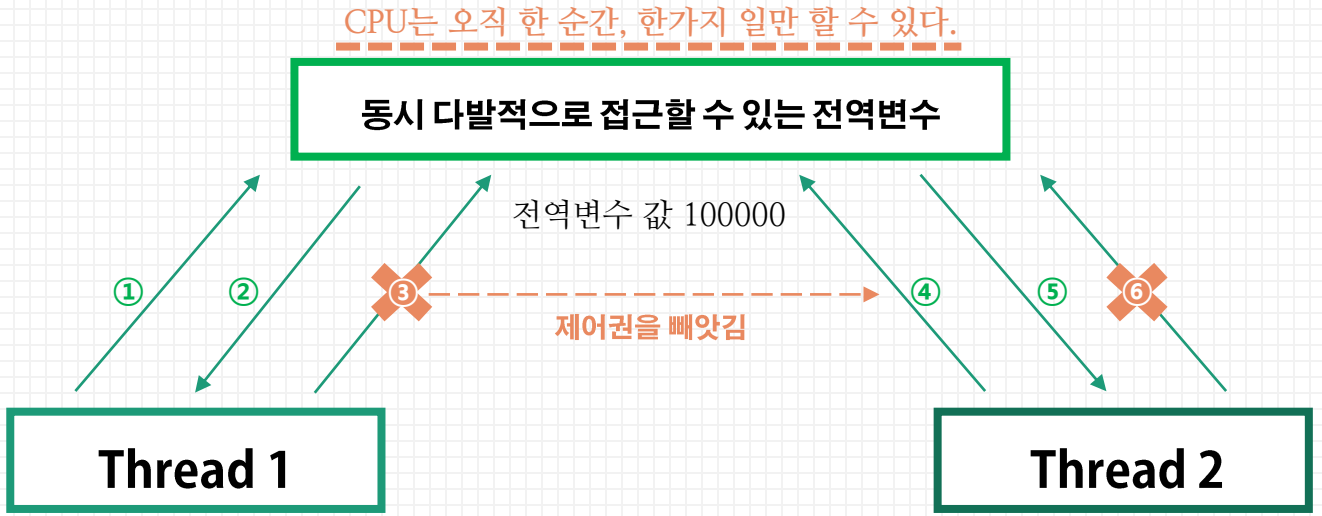
# Thread 문제점 예시

링크 https://github.com/limcholong/LectureContents/blob/main/java/CholongLim/Day20/src/BankBombEventTest.java

```java
class Bank {
    // 만 단위로 10억임
    private int money = 100000;

    public int getMoney() {
        return money;
    }
    public void setMoney(int money) {
        this.money = money;
    }

    public void plusMoney(int plus) {
        int m = this.getMoney();

        try {
            Thread.sleep( millis: 0);
        } catch (InterruptedException e) {
            // 에러 발생하면 어디서 에러가 났는지 출력해줘 ~
            e.printStackTrace();
        }
        this.setMoney(m + plus);
    }
    public void minusMoney(int minus) {
        int m = this.getMoney();

        try {
            Thread.sleep( millis: 0);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        this.setMoney(m - minus);
    }
}
```

19일차에 배웠던 하단 내용의 예시 코드이다.

Private int money = 100000;
동시 다발적으로 접근할 수 있는 전역변수 100000에 두 가지의 Thread가 동시에 접근할 것이다.



# Thread 문제점 예시

링크 https://github.com/limcholong/LectureContents/blob/main/java/CholongLim/Day20/src/BankBombEventTest.java

```
35  class FirstThread extends Thread {
36      public void run () {
37          for (int i = 0; i < 1000; i++) {
38              BankBombEventTest.myBank.plusMoney(1000);
39              System.out.println("plusMoney(1000) = " + BankBombEventTest.myBank.getMoney());
40          }
41      }
42  }
43
44  class SecondThread extends Thread {
45      public void run () {
46          for (int i = 0; i < 1000; i++) {
47              BankBombEventTest.myBank.minusMoney(1000);
48              System.out.println("minusMoney(1000) = " + BankBombEventTest.myBank.getMoney());
49          }
50      }
51  }
52
53  public class BankBombEventTest {
54      public static Bank myBank = new Bank();
55      // Bank을 전역으로서 공유하기 위해서
56
57      public static void main(String[] args) {
58          System.out.println("원금: " + myBank.getMoney());
59
60          FirstThread first = new FirstThread();
61          SecondThread second = new SecondThread();
62
63          first.start();
64          second.start();
65      }
66  }
```

1000번의 덧셈

1000번의 뺄셈

Thread를 통해 동시 start

Thread :

동작하고 있는 프로그램을 프로세스라고 한다.
보통 한 개의 프로세스는 한 가지의 일을 하지만,
Thread를 이용하면 한 프로세스 내에서 두 가지 이상의 일을
동시에 할 수 있게 된다.

Class ____ extends Thread :

Thread를 만드는 방법이다. ____(클래스명)이 Thread 클래스를
상속했다. Thread 클래스의 run 매서드를 구현하면,
.start( ) 시, run 매서드가 수행된다.
(Thread 클래스는 start 실행 시, run 매서드가 수행되도록
내부적으로 코딩되어 있다.)

# Thread 문제점 예시

링크 https://github.com/limcholong/LectureContents/blob/main/java/CholongLim/Day20/src/BankBombEventTest.java

```
원금: 100000
plusMoney(1000) = 101000
plusMoney(1000) = 101000
minusMoney(1000) = 100000
plusMoney(1000) = 102000
minusMoney(1000) = 101000
plusMoney(1000) = 102000
minusMoney(1000) = 101000
plusMoney(1000) = 102000
minusMoney(1000) = 101000
plusMoney(1000) = 102000
minusMoney(1000) = 101000
plusMoney(1000) = 102000
minusMoney(1000) = 101000
plusMoney(1000) = 102000
minusMoney(1000) = 101000
plusMoney(1000) = 102000
minusMoney(1000) = 101000
```

```
minusMoney(1000) = -98000
minusMoney(1000) = -99000
minusMoney(1000) = -100000
minusMoney(1000) = -101000
minusMoney(1000) = -102000
plusMoney(1000) = 26000
plusMoney(1000) = 27000
plusMoney(1000) = 28000
plusMoney(1000) = 29000
plusMoney(1000) = 30000
plusMoney(1000) = 31000
plusMoney(1000) = 32000
```

```
minusMoney(1000) = 27000
minusMoney(1000) = 26000
plusMoney(1000) = 29000
minusMoney(1000) = 25000
minusMoney(1000) = 24000
minusMoney(1000) = 23000
minusMoney(1000) = 22000
minusMoney(1000) = 21000
minusMoney(1000) = 20000
minusMoney(1000) = 19000
minusMoney(1000) = 18000
minusMoney(1000) = 17000
```

```
plusMoney(1000) = 83000
plusMoney(1000) = 84000
plusMoney(1000) = 85000
plusMoney(1000) = 86000
plusMoney(1000) = 87000
minusMoney(1000) = -103000
minusMoney(1000) = -103000
minusMoney(1000) = -104000
plusMoney(1000) = -102000
plusMoney(1000) = -104000
plusMoney(1000) = -103000
```

**출력 결과,**
plusMoney와 minusMoney가 번갈아가며 값을 구하고 있다.

위 Thread는 두 가지 문제가 있다.

1. Synchronization (동기화) 문제 :
   두 개 이상의 Thread가 동시에 접근하여 계산 결과가 덮어써졌다.

2. **데이터의 무결성 문제** :
   어떤 특정 상황에서 데이터 처리가 무시되었고, 데이터의 정확성과 일관성을 유지하고 보증되지 않았다.

결과적으로 값을 구할 때 Thread가 동시에 접근하여 계산 결과가 덮어 쓰여지며, 데이터의 값이 지켜지지 않았고 정확성과 일관성이 떨어졌다.

# Thread 예시

링크 https://github.com/limcholong/LectureContents/blob/main/java/CholongLim/Day20/src/Counter.java

```java
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Counter {
    private int count = 1;
    // Thread를 사용할 때 Lock을 걸려면
    // ReentrantLock을 사용하여 재진입이 가능한 형태로 만들어줘야 한다.
    private Lock lock = new ReentrantLock();

    public void increment () {
        try {
            lock.lock();
            count++;
        } finally {
            // 성공적으로 처리했던, 실패를 했던
            // finally는 무조건 실행된다.
            // 그러므로 내부에서 문제가 생겨도 Lock은 해제를 한다는 뜻
            // ex) 화장실에서 문 잠그고 죽음 ???
            lock.unlock();
        }
    }
    public void decrement () {
        try {
            lock.lock();
            count--;
        } finally {
            lock.unlock();
        }
    }
    public int getCount () {
        return count;
    }
}
```

Lock ( ) 매서드는 ReentrantLock 인스턴스에 lock 을 걸어 lock ( )을 호출하는 모든 Thread가 unlock( ) 매서드를 호출될 때 까지 블록 되도록 한다.

- Finally 절에서의 unlock( ) 호출 :

Lock 을 이용해서 Critical Section (임계 영역) 을 보호할 때는 Critical Section (임계 영역) 에서 예외가 발생할 수 있음을 기억해야 한다. 다른 Thread가 lock 을 걸 수 있도록 Lock 인스턴스의 lock 은 반드시 해제되어야 한다.

- Finally 사용 :

Critical Section (임계 영역)이 try catch 부분을 lock 하기 위해 lock ( ) 매서드를 부르면 Critical Section (임계 영역) 의 실행이 끝난 후 해당 lock을 놓아주어야 한다.
그렇지 않으면 다른 Thread가 이 매서드를 실행하려 하는 경우 unlock이 되지 않았기 때문에 실행하지 못하고 계속 기다려야 한다. 이런 경우 어떤 이유에서, 이 매서드에서 예외 / 에러가 발생하여 unlock이 실행되지 않고 매서드가 종료되면 프로그램은 데드락(Deadlock) 상태에 빠지고 만다.
이를 방지하기 위해 finally 블록 안에서 unlock을 부른다.

# Thread 예시

링크 https://github.com/limcholong/LectureContents/blob/main/java/CholongLim/Day20/src/Worker.java

```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");
            }
        }
    }
}
```

```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());
    }
}
```

Runnable 인터페이스 (Runnable Interface) :

run( ) 매서드만 채워주면 Runnable 인터페이스를 구현하는 것은 간단하다.
하지만 Thread를 상속받은 클래스처럼 start( ) 매서드는 없다.
따라서 별도로 Thread를 생성해주고 ,
구현한 Runnable 인터페이스를 인자로 넘겨주어야 한다.

# Thread 예시

링크 https://github.com/limcholong/LectureContents/blob/main/java/CholongLim/Day20/src/BankLockTest.java

```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());
    }
}
```

```
First count: 1
I'm increment
I'm increment
I'm increment
I'm increment
I'm increment
I'm increment
I'm decrement
I'm decrement
I'm decrement
I'm decrement
I'm decrement
I'm decrement
I'm decrement
I'm decrement
```

...

```
I'm increment
I'm increment
I'm increment
I'm increment
I'm increment
I'm increment
I'm increment
I'm increment
I'm increment
I'm increment
I'm increment
I'm increment
I'm increment
I'm increment
I'm increment
Final count: 1
```

Join ( ) :

- 모든 Thread가 종료된 후에 main 매서드를 종료시키고 싶은 경우
- Thread가 종료될 때까지 기다리게 하는 매서드
- main 매서드가 종료되기 전에 Thread에 담긴 각각의 Thread에 join 매서드를 호출하여 Thread가 종료될 때까지 대기하도록 한다.

1. Synchronization (동기화) 문제
2. 데이터의 무결성 문제

문제를 Lock을 통해 해결하였다.