Chapter 23. 쓰레드와 동기화

23-1. 쓰레드의 이해와 생성

■ 쓰레드의 이해와 Thread 클래스의 상속

쓰레드와 프로세스의 이해 및 관계

- 프로세스는 실행중인 프로그램을 의미한다.
- 쓰레드는 프로세스 내에서 별도의 실행흐름을 갖는 대상이다.
- 프로세스 내에서 둘 이상의 쓰레드를 생성하는 것이 가능하다.

프로그램이 실행될 때 프로세스에 할당된 메모리, 이 자체를 단순히 프로세스라고 하기도 한다.

사실 쓰레드는 모든 일의 기본 단위이다. main 메소드를 호출하는 것도 프로세스 생성시 함께 생성되는 **main 쓰레드**를 통해서 이뤄진다.

쓰레드의 생성

ThreadUnderstand.java

```
class ShowThread extends Thread
{
    String threadName;
    public ShowThread(String name)
    {
        threadName=name;
    }
    public void run()  쓰레드의 main 메소드가 run이다!
    {
        for(int i=0; i<100; i++)
        {
            System.out.println("안녕하세요. "+threadName+"입니다.");
            try
            {
                sleep(100);
            }
            catch(InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
}
```

```
public static void main(String[] args)
{
    ShowThread st1=new ShowThread("멋진 쓰레드");
    ShowThread st2=new ShowThread("예쁜 쓰레드");
    st1.start();
    st2.start();
}
```

별도의 쓰레드 생성을 위해서는 별도의 쓰레드 클래스를 정의해야 한다.

쓰레드 클래스는 **Thread를 상속하는 클래스**를 의미한다.

```
안녕하세요. 예쁜 쓰레드입니다.
안녕하세요. 멋진 쓰레드입니다.
안녕하세요. 멋진 쓰레드입니다.
안녕하세요. 예쁜 쓰레드입니다.
안녕하세요. 멋진 쓰레드입니다.
안녕하세요. 예쁜 쓰레드입니다.
안녕하세요. 예쁜 쓰레드입니다.
안녕하세요. 멋진 쓰레드입니다.
.....중    략.....
```

실행결과

start 메소드가 호출되면 쓰레드가 생성되고, 생성된 쓰레드는 run 메소드를 호출한다.

■ 쓰레드의 생성을 보인 첫 번째 예제의 의문점1

run은 쓰레드의 생성과 실행을 동시에 명령하는 메소드이다.

• Question 1

쓰레드 인스턴스를 생성하고 나서, start 메소드를 호출하면 run 메소드가 실행되는데, run 메소드를 직접 호출하면 안되나요?

• Answer 1

run 메소드를 직접 호출하는 것도 불가능한 일은 아니다. 단 이러한 경우에는 단순한 메소드의 호출일 뿐, 쓰레드의 생성으로 이어지지는 않는다. 잠시 후에 설명을 하겠지만, 쓰레드는 자신만의 메모리 공간을 할당 받아서 별도의 실행흐름을 형성한다. 따라서 자바 가상머신은 start 메소드의 호출을 요구하는 것이다. 메모리 공간의 할당 등 쓰레드의 실행을 위한 기반을 마련한 다음에 run 메소드를 대신 호출해 주기 위해서 말이다. 이는 우리가 main 메소드를 직접 호출하지 않는 것과 비슷한 이치이다.

쓰레드의 동시 실행은 CPU의 수와 무관하게 이뤄진다.

• Question 2

CPU가 하나인데, 어떻게 둘 이상의 쓰레드가 동시에 실행 가능한가요?

• Answer 2

이 질문에 대한 답변은 생각보다 간단하다. 모든 쓰레드는 CPU를 공유한다. 물론 CPU를 공유하는 방식에는 원칙이 존재하는데, 이는 잠시 후에 별도로 설명이 이뤄진다. 참고로 코어(CPU 내에 존재하는 연산장치)가 여러 개 존재하는 CPU에서는 쓰레드 각각에 코어가 하나씩 할당되어 실행되기도 한다.

■ 쓰레드의 생성을 보인 첫 번째 예제의 의문점2

쓰레드의 run 메소드 종료는 쓰레드의 종료로 이어진다.

• Question 3

main 메소드가 종료되어도 쓰레드는 실행을 계속하나요? 그리고 쓰레드는 run 메소드의 실행이 완료되면 종료되나요?

• Answer 3

쓰레드의 main 메소드가 run 메소드이다. 따라서 run 메소드의 실행이 완료되면, 해당 쓰레드는 종료가 되고 소멸된다. 그리고 앞서 보인 예제에서는 main 메소드 내에서 쓰레드를 생성했었다. 그런데 쓰레드를 생성하고, start 메소드를 호출한다고 해서, main 메소드가 멈춰서는 것은 아니다. main 메소드도 여느 쓰레드와 마찬가지로 자신만의 실행흐름을 이어간다. 따라서 main 메소드가 먼저 종료될 수도 있다. 하지만 main 메소드가 종료되어도 실행 중에 있는 쓰레드가 있다면, 프로그램은 종료되지 않는다. 사실 main 메소드도 쓰레드에 의해 실행된다. 그리고 main 메소드를 실행하는 쓰레드를 가리켜 별도로 'main 쓰레드'라 부르기도 한다. 결국 마지막 남은 쓰레드까지 실행을 완료해야 프로그램은 종료된다.

■ 쓰레드의 생성을 보인 첫 번째 예제의 의문점3

쓰레드라는 표현은 매우 포괄적이다!

• Question 4

쓰레드가 별도의 실행흐름을 구성하는 것은 알겠는데, 그렇다면 정확히 무엇을 가리켜 쓰레드라 하나요? 인스턴스가 쓰레드인가요?

• Answer 4

Thread를 상속하는 클래스의 인스턴스를 가리켜 쓰레드라고도 하지만, 이는 엄밀히 말해서 잘못된 표현이다. 쓰레드는 자바 가상머신이 생성하는 것이기 때문이다. start 메소드가 호출되면, 자바 가상머신은 별도의 실행흐름을 형성하기 위한 여러 가지 준비에 들어간다. 그 중 대표적인 것은 메모리 공간의 할당이다. 실행흐름을 구성하기 위해서는 메모리 공간의 할당이 필수 아니겠는가? 그리고 이미 생성된 쓰레드들과 CPU를 나눠 쓰기 위한 각종 정보들이 등록된다. 이렇듯 별도의 실행흐름을 형성하기 위해서 자바 가상머신에 의해 만들어지는(또는 준비되는) 모든 리소스와 각종 정보들을 총칭해서 쓰레드라 한다.

■ 쓰레드를 생성하는 두 번째 방법

RunnableThread.java

```
class Sum
{
    int num;
    public Sum() { num=0; }
    public void addNum(int n) { num+=n; }
    public int getNum() { return num; }
}

class AdderThread extends Sum implements Runnable
{
    int start, end;

    public AdderThread(int s, int e)
    {
        start=s;
        end=e;
    }
    public void run()
    {
        for(int i=start; i<=end; i++)
            addNum(i);
    }
}
```

Runnable 인터페이스를 구현하는 클래스의 인스턴스를 대상으로 Thread 클래스의 인스턴스를 생성한다. 이 방법은 상속할 클래스가 존재할 때 유용하게 사용된다.

```
public static void main(String[] args)
{
    AdderThread at1=new AdderThread(1, 50);
    AdderThread at2=new AdderThread(51, 100);
    Thread tr1=new Thread(at1);
    Thread tr2=new Thread(at2);
    tr1.start();
    tr2.start();

    try
    {
        tr1.join();
        tr2.join();
    }
    catch(InterruptedException e)
    {
        e.printStackTrace();
    }

    System.out.println("1~100까지의 합 : "+(at1.getNum()+at2.getNum()));
}
```

join 메소드가 호출되면, 해당 쓰레드의 종료를 기다리게 된다!

위 예제에서 main 쓰레드가 join 메소드를 호출하지 않았다면, 추가로 생성된 두 쓰레드가 작업을 완료하기 전에 값을 참조하여 쓰레기 값이 출력될 수 있다.

```
1~100까지의 합 : 5050
```

실행결과

23-2. 쓰레드의 특성

■ 쓰레드의 스케줄링과 우선순위 컨트롤

나 정말 JAVA를 공부한 적 없는데

쓰레드 스케줄링의 두 가지 기준

- ✓ 우선순위가 높은 쓰레드의 실행을 우선시한다.
- ✓ 우선순위가 동일할 때는 CPU의 할당시간을 나눈다.

PriorityTestOne.java

```
class MessageSendingThread extends Thread
{
    String message;

    public MessageSendingThread(String str)
    {
        message=str;
    }
    public void run()
    {
        for(int i=0; i<1000000; i++)
            System.out.println(message+"("+getPriority()+")");
    }
}
```

```
public static void main(String[] args)
{
    MessageSendingThread tr1=new MessageSendingThread("First");
    MessageSendingThread tr2=new MessageSendingThread("Second");
    MessageSendingThread tr3=new MessageSendingThread("Third");
    tr1.start();
    tr2.start();
    tr3.start();
}
```

실행 결과

```
First(5)
First(5)
Second(5)
. . . . . .
Third(5)
First(5)
. . . . . .
Third(5)
```

메소드 getPriority의 반환값을 통해서 쓰레드의 우선순위를 확인할 수 있다. 왼쪽의 실행결과에서 보이듯이, 우선순위와 관련해서 별도의 지시를 하지 않으면, 동일한 우선순위의 쓰레드들이 생성된다.

■ 우선순위가 다른 쓰레드들의 실행

PriorityTestTwo.java

```
class MessageSendingThread extends Thread
{
    String message;
    public MessageSendingThread(String str, int prio)
    {
        message=str;
        setPriority(prio);
    }
    public void run()
    {
        for(int i=0; i<1000000; i++)
            System.out.println(message+"("+getPriority()+")");
    }
}
```

```
public static void main(String[] args)
{
    MessageSendingThread tr1
        =new MessageSendingThread("First", Thread.MAX_PRIORITY);
    MessageSendingThread tr2
        =new MessageSendingThread("Second", Thread.NORM_PRIORITY);
    MessageSendingThread tr3
        =new MessageSendingThread("Third", Thread.MIN_PRIORITY);
    tr1.start();
    tr2.start();
    tr3.start();
}
```

```
First(10)
First(10)
. . . . . .
Second(5)
Second(5)
. . . . . .
Third(1)
```

실행 결과

Thread.MAX_PRIORITY는 상수로 10,

Thread.NORM_PRIORITY는 상수로 5,

Thread.MIN_PRIORITY는 상수로 1

실행결과에서 보이듯이 쓰레드의 실행시간은 우선순위의 비율대로 나뉘지 않는다.

높은 우선순위의 쓰레드가 종료되어야 낮은 우선순위의 쓰레드가 실행된다.

낮은 우선순위의 쓰레드 실행

PriorityTestThree.java

```
class MessageSendingThread extends Thread
{
    String message;

    public MessageSendingThread(String str, int prio)
    {
        message=str;
        setPriority(prio);
    }
    public void run()
    {
        for(int i=0; i<1000000; i++)
        {
            System.out.println(message+"("+getPriority()+")");

            try
            {
                sleep(1);
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
}
```

CPU를 양보!

```
Third(1)
First(10)
Third(1)
Second(5)
First(10)
Third(1)
Second(5)
. . . . . .
```

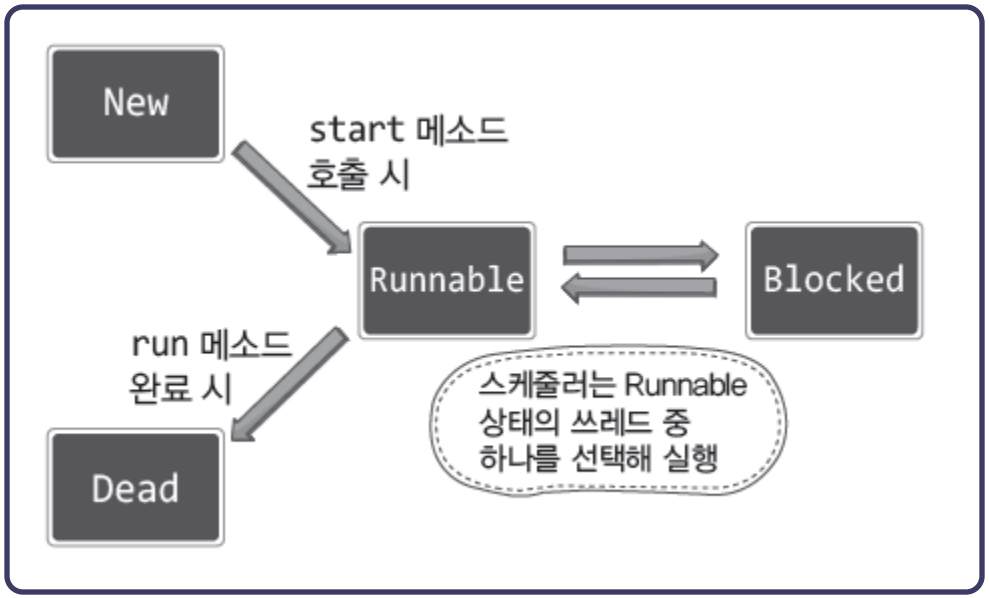
실행 결과

쓰레드가 CPU의 할당을 필요로 하지 않을 경우, CPU를 다른 쓰레드에게 양보한다.

```
public static void main(String[] args)
{
    MessageSendingThread tr1
        =new MessageSendingThread("First", Thread.MAX_PRIORITY);
    MessageSendingThread tr2
        =new MessageSendingThread("Second", Thread.NORM_PRIORITY);
    MessageSendingThread tr3
        =new MessageSendingThread("Third", Thread.MIN_PRIORITY);

    tr1.start();
    tr2.start();
    tr3.start();
}
```

■ 쓰레드의 라이프 사이클

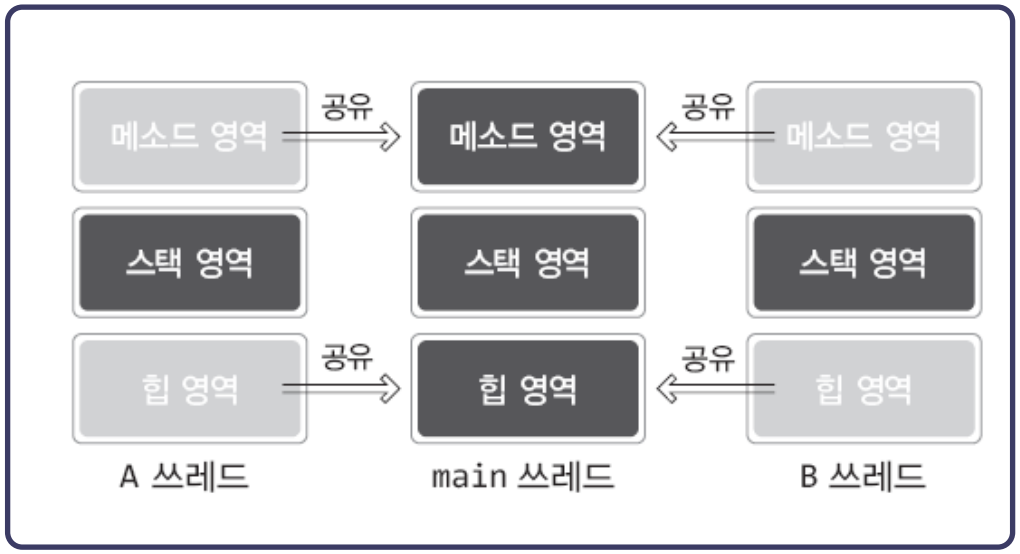


Runnable 상태의 쓰레드만이 스케줄러에 의해 스케줄링 가능하다.

그리고 앞서 보인 sleep, join 메소드의 호출로 인해서 쓰레드는 Blocked 상태가 된다.

한번 종료된 쓰레드는 다시 Runnable 상태가 될 수 없지만, Blocked 상태의 쓰레드는 조건이 성립되면 다시 Runnable 상태가 된다.

쓰레드의 메모리 구성



모든 쓰레드는 스택을 제외한 메소드 영역과 힙을 공유한다. 따라서 이 두 영역을 통해서 데이터를 주고 받을 수 있다.

스택은 쓰레드 별로 독립적일 수 밖에 없는 이유는, 쓰레드의 실행이 메소드의 호출을 통해서 이뤄지고, 메소드의 호출을 위해서 사용되는 메모리 공간이 스택이기 때문이다.

■ 쓰레드간 메모리 영역의 공유 예제

ThreadHeapMultiAccess.java

```
class Sum
{
    int num;
    public Sum() { num=0; }
    public void addNum(int n) { num+=n; }
    public int getNum() { return num; }
}

class AdderThread extends Thread
{
    Sum sumInst;
    int start, end;

    public AdderThread(Sum sum, int s, int e)
    {
        sumInst=sum;
        start=s;
        end=e;
    }
    public void run()
    {
        for(int i=start; i<=end; i++)
            sumInst.addNum(i);
    }
}
```

```
1~100까지의 합 : 5050
```

실행 결과

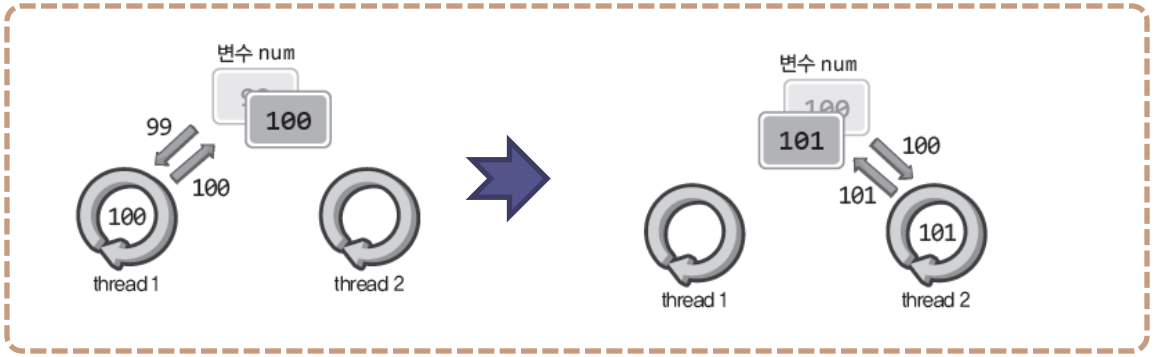
```
public static void main(String[] args)
{
    Sum s=new Sum();
    AdderThread at1=new AdderThread(s, 1, 50);
    AdderThread at2=new AdderThread(s, 51, 100);
    at1.start();
    at2.start();
    try
    {
        at1.join();
        at2.join();
    }
    catch(InterruptedException e)
    {
        e.printStackTrace();
    }
    System.out.println("1~100까지의 합 : "+s.getNum());
}
```

위의 예제는 둘 이상의 쓰레드가 메모리 공간에 동시 접근하는 문제를 가지고 있다. 따라서 정상적이지 못한 실행의 결과가 나올 수도 있다.

23-3. 동기화(Synchronization)

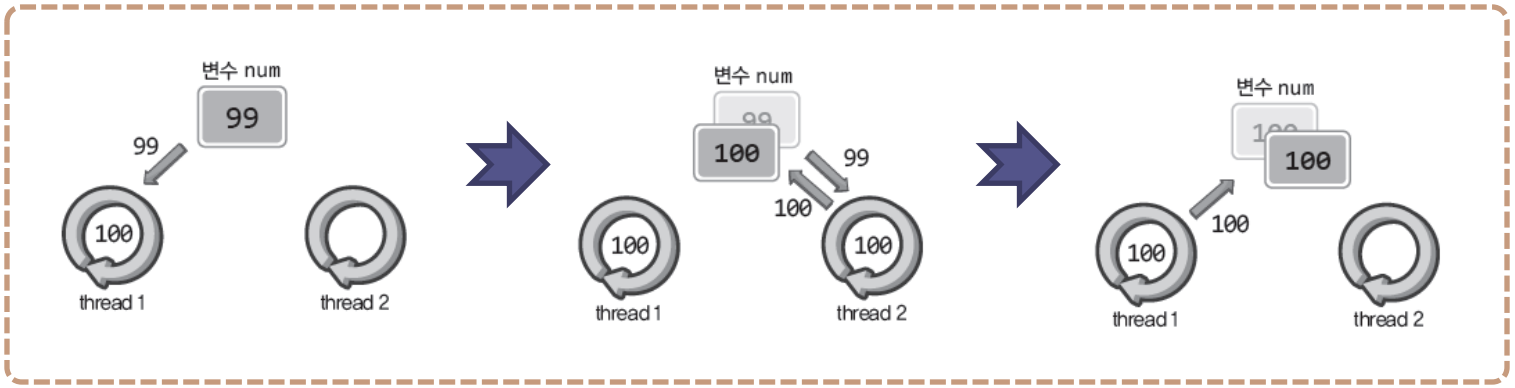
■ 쓰레드의 메모리 접근방식과 그에 따른 문제점

정상적 연산결과를 보이는 연산의 예



변수 num에 저장된 값을 1씩 증가시키는 두 쓰레드의 연산의 예

비정상적 연산결과를 보이는 연산의 예



따라서 둘 이상의 쓰레드가 하나의 메모리 공간에 동시 접근하는 것은 문제를 일으킨다.

Thread-safe 합니까?

Note that this implementation is not synchronized

API 문서에는 해당 클래스의 인스턴스가 둘 이상의 쓰레드가 동시에 접근을 해도 문제가 발생하지 않는지를 명시하고 있다. 따라서 쓰레드 기반의 프로그래밍을 한다면, 특정 클래스의 사용에 앞서 쓰레드에 안전한지를 확인해야 한다.

쓰레드의 동기화 기법1
: synchronized 기반 동기화 메소드


```
class Increment
{
    int num=0;
    public synchronized void increment(){ num++; }
    public int getNum() { return num; }
}

class IncThread extends Thread
{
    Increment inc;

    public IncThread(Increment inc)
    {
        this.inc=inc;
    }
    public void run()
    {
        for(int i=0; i<10000; i++)
            for(int j=0; j<10000; j++)
                inc.increment();
    }
}
```

```
public synchronized void increment( )
{
    num++;
}
```

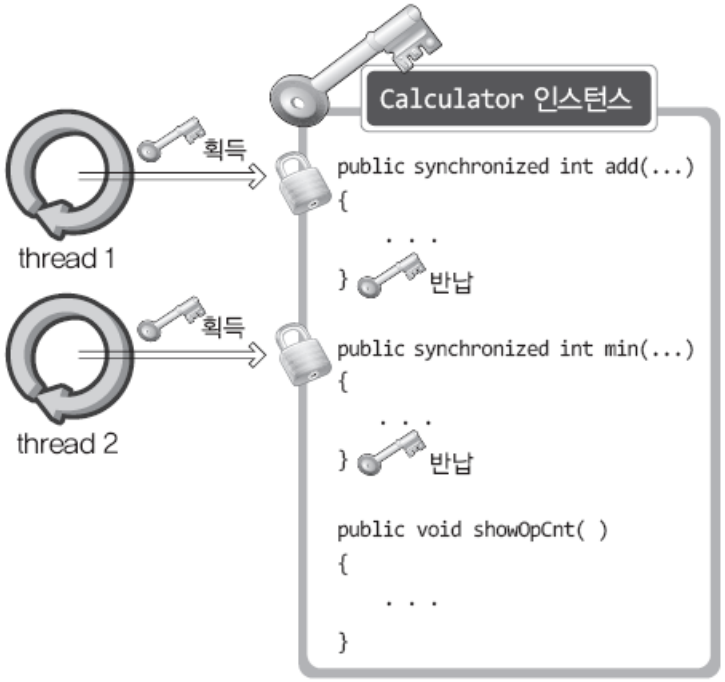
동기화 메소드의 선언!

synchronized 선언으로 인해서 increment 메소드는 쓰레드에 안전한 함수가 된다.

synchronized 선언으로 인해서 increment 메소드는 정상적으로 동작한다.

그러나 엄청난 성능의 감소를 동반한다! 특히 위 예제와 같이 빈번함 메소드의 호출은 문제가 될 수 있다.

synchronized 기반 동기화 메소드의 정확한 이해



동기화에 사용되는 인스턴스는 하나이며, 이 인스턴스에는 **하나의 열쇠만이 존재**한다.

동기화의 대상은 인스턴스이며, 인스턴스의 열쇠를 획득하는 순간 **모든 동기화 메소드에는 타 쓰레드의 접근이 불가능**하다. 따라서 메소드 내에서 동기화가 필요한 영역이 매우 제한적이라면 메소드 전부를 synchronized로 선언하는 것은 적절치 않다.

쓰레드의 동기화 기법2
: synchronized 기반 동기화 블록

동기화 메소드 기반

```
public synchronized int add(int n1, int n2)
{
    opCnt++;     // 동기화가 필요한 문장
    return n1+n2;
}

public synchronized int min(int n1, int n2)
{
    opCnt++;     // 동기화가 필요한 문장
    return n1-n2;
}
```

동기화 블록 기반

```
public int add(int n1, int n2)
{
    synchronized(this)
    {
        opCnt++;     // 동기화 된 문장
    }
    return n1+n2;
}

public int min(int n1, int n2)
{
    synchronized(this)
    {
        opCnt++;     // 동기화 된 문장
    }
    return n1-n2;
}
```

동기화 블록을 이용하면 동기화의 대상이 되는 영역을 세밀하게 제한할 수 있다.

synchronized(this)에서 this는 동기화의 대상을 알리는 용도로 사용이 되었다. 즉, 메소드가 호출된 인스턴스 자신의 열쇠를 대상으로 동기화를 진행하는 문장이다.

■ 동기화 블록의 예

```
public synchronized void addOneNum1()
{
    synchronized(key1)
    {
        num1+=1;
    }
}
public synchronized void addTwoNum1()
{
    synchronized(key1)
    {
        num1+=2;
    }
}
public synchronized void addOneNum2()
{
    synchronized(key2)
    {
        num2+=1;
    }
}
public synchronized void addTwoNum2()
{
    synchronized(key2)
    {
        num2+=2;
    }
}
. . . . .
Object key1=new Object();
Object key2=new Object();
```

```
class IHaveTwoNum
{
    . . . .
    public void addOneNum1()
    {
        synchronized(this) { num1+=1; }
    }
    public void addTwoNum1()
    {
        synchronized(this) { num1+=2; }
    }
    public void addOneNum2()
    {
        synchronized(key) { num2+=1; }
    }
    public void addTwoNum2()
    {
        synchronized(key) { num2+=2; }
    }
    . . . .
    Object key=new Object();
}
```

보다 일반적인 형태, 두 개의 동기화 인스턴스 중 하나는 this로 지정!

왼쪽의 코드에서 보이듯이 동기화 블록을 이용하면 동기화의 기준을 다양화할 수 있다.

■ 쓰레드 접근순서의 동기화 필요성


```
class NewsWriter extends Thread
{
    NewsPaper paper;

    public NewsWriter(NewsPaper paper)
    {
        this.paper=paper;
    }
    public void run()
    {
        paper.setTodayNews("자바의 열기가 뜨겁습니다.");
    }
}

class NewsReader extends Thread
{
    NewsPaper paper;

    public NewsReader(NewsPaper paper)
    {
        this.paper=paper;
    }
    public void run()
    {
        System.out.println("오늘의 뉴스 : "+paper.getTodayNews());
    }
}
```

```
class NewsPaper
{
    String todayNews;

    public void setTodayNews(String news)
    {
        todayNews=news;
    }

    public String getTodayNews()
    {
        return todayNews;
    }
}
```

```
public static void main(String[] args)
{
    NewsPaper paper=new NewsPaper();
    NewsReader reader=new NewsReader(paper);
    NewsWriter writer=new NewsWriter(paper);

    reader.start();
    writer.start();

    try
    {
        reader.join();
        writer.join();
    }
    catch(InterruptedException e)
    {
        e.printStackTrace();
    }
}
```

본 예제가 논리적으로 실행되려면 NewsWriter 쓰레드가 먼저 실행되고, 이어서 NewReader 쓰레드가 실행되어야 한다. 하지만 이를 보장하지 못하는 구조로 구현이 되어 있다.

■ wait, notify, notifyall에 의한 실행순서 동기화

나 정말 JAVA를 공부한 적 없는데요

· public final void wait() throws InterruptedException

위의 함수를 호출한 쓰레드는 notify 또는 notifyAll 메소드가 호출될 때까지 블로킹 상태에 놓이게 된다.

· public final void notify()

wait 함수의 호출을 통해서 블로킹 상태에 놓여있는 쓰레드 하나를 깨운다.

· public final void notifyAll()

wait 함수의 호출을 통해서 블로킹 상태에 놓여있는 모든 쓰레드를 깨운다.

```
synchronized(this)
{
    wait();
}
```

위의 함수들은 왼쪽에서 보이는 바와 같이 한 순간에 하나의 쓰레드만 호출할 수 있도록 동기화 처리를 해야 한다.

■ 실행순서 동기화 예제


```
class NewsPaper
{
    String todayNews;
    boolean isTodayNews=false;

    public void setTodayNews(String news)
    {
        todayNews=news;
        isTodayNews=true;
        synchronized(this)
        {
            notifyAll();    // 모두 일어나세요!
        }
    }
    public String getTodayNews()
    {
        if(isTodayNews==false)
        {
            try
            {
                synchronized(this)
                {
                    wait();     // 한숨 자면서 기다리겠습니다.
                }
            }
            catch(InterruptedException e)
            {
                e.printStackTrace();
            }
        }

        return todayNews;
    }
}
```

wait과 notifyAll 메소드에 의한 동기화가 진행될 때, 이전 예제에서 달라지는 부분은 쓰레드 클래스가 아닌 쓰레드에 의해 접근이 이뤄지는 NewsPaper 클래스라는 사실에 주목하기 바란다.

23-4. 새로운 동기화 방식

synchronized 키워드의 대체

```
class MyClass
{
    private final ReentrantLock criticObj=new ReentrantLock();
    . . . .
    void myMethod(int arg)
    {
        criticObj.lock();    // 다른 쓰레드가 진입하지 못하게 문을 잠근다.
        . . . .
        . . . .
        criticObj.unlock();  // 다른 쓰레드의 진입이 가능하게 문을 연다.
    }
}
```

보다 안정적인 구현모델,

반드시 unlock 메소드가 호출되는 모델

```
void myMethod(int arg)
{
    criticObj.lock();    // 다른 쓰레드가 진입하지 못하게 문을 잠근다.
    try
    {
        . . . .
        . . . .
    }
    finally
    {
        criticObj.unlock();  // 다른 쓰레드의 진입이 가능하게 문을 연다.
    }
}
```

ReentrantLock 인스턴스를 이용한 동기화 기법

Java Ver 5.0 이후로 제공된 동기화 방식이다. lock 메소드와 unlock 메소드의 호출을 통해서 동기화 블록을 구성한다.

■ await, signal, signalAll에 의한 실행순서의 동기화

- await 낮잠을 취한다(wait 메소드에 대응)
- signal 낮잠 자는 쓰레드 하나를 깨운다(notify 메소드에 대응).
- signalAll 낮잠 자는 모든 쓰레드를 깨운다(notifyAll 메소드에 대응).

ReentrantLock 인스턴스 대상으로 newCondition 메소드 호출 시, Condition 인터페이스를 구현하는 인스턴스의 참조 값 반환!
이 인스턴스를 대상으로 위의 메소드를 호출하여, 쓰레드의 실행순서를 동기화 한다.

위의 메소드의 사용방법을 보이는 예제 ConditionSyncStringReadWrite.java는 코드 양이 많은 관계로 해당 파일을 열어서 참고하기 바란다. 참고로, 앞서 보인 wait, notify 메소드의 호출을 통한 동기화 방법과 크게 다르지 않다.