📖

10주차 과제 : Thread & Runnable

목표

자바의 멀티쓰레드 프로그래밍에 대해 학습하세요.

학습할 것 (필수)

- Thread 클래스와 Runnable 인터페이스
- 쓰레드의 상태
- 쓰레드의 우선순위
- Main 쓰레드
- 동기화
- 데드락

공부 시작에 앞서 프로세스와 쓰레드에 대해 알아보자

이에 대해 이해하기 쉽게 설명된 PT가 있어 가져와봤다.

https://youtu.be/DmZnOg5Ced8

Process

▼ 단순히 실행중인 프로그램이라고 볼 수 있다.

▼ 사용자가 작성한 프로그램이 운영체제에 의해 메모리 공간을 할당 받아 실행 중인 것을 말한다. 이러한 프로세스는 프로그램에 사용되는 **데이터와 메모리 등의 자원 그리고 쓰레드로 구성**이 된다.

Thread

▼ 프로세스 내에서 실제로 작업을 수행하는 주체를 의미한다.

▼ 모든 프로세스는 1개 이상의 쓰레드가 존재하여 작업을 수행한다.

▼ 두개 이상의 쓰레드를 가지는 프로세스를 멀티 쓰레드 프로세스라고 한다.

▼ 경량 프로세스라고 불리며 **가장 작은 실행 단위**이다.

이러한 쓰레드를 생성하는 방법은 크게 두 가지가 있다.

1. Runnable 인터페이스 사용
2. Thread 클래스를 사용

두 가지 모두 java.lang 패키지에 포함이 되어있다

Thread 클래스는 Runnalbe 인터페이스를 구현한 클래스이므로 어떤 것을 적용 하느냐의 차이이다.

```
class ThreadWithClass extends Thread {
  public void run() {
    for (int i = 0; i < 10; i++) {
      System.out.println(getName());  // 현재 실행 중인 스레드의 이름을 반환함.
      try {
        Thread.sleep(10);     // 0.01초간 스레드를 멈춤.
      } catch (InterruptedException e) {
        e.printStackTrace();
```

```
      }
    }
  }
}

class ThreadWithRunnable implements Runnable {
  public void run() {
    for (int i = 0; i < 10; i++) {
      System.out.println(Thread.currentThread().getName()); // 현재 실행 중인 스레드의 이름을 반환함.
      try {
        Thread.sleep(10);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
}

public class ProcessTest {
  public static void main(String[] args){
    ThreadWithClass thread0 = new ThreadWithClass();    // Thread 클래스를 상속받는 방법
    Thread thread1 = new Thread(new ThreadWithRunnable());  // Runnable 인터페이스를 구현하는 방법

    thread0.start();  // 스레드의 실행
    thread1.start();  // 스레드의 실행
  }
}

output :
Thread-1
Thread-0
Thread-0
Thread-1
Thread-1
Thread-0
Thread-0
Thread-1
Thread-1
Thread-0
Thread-0
Thread-1
Thread-0
Thread-1
Thread-1
Thread-0
Thread-1
Thread-0
Thread-1
Thread-0
```

위의 예제를 보면, 생성된 스레드가 서로 번갈아가며 실행이 되고 있다.

쓰레드 클래스를 상속받으면 다른 클래스를 상속받을 수 없으므로, 일반적으로 Runnable 인터페이스를 구현하는 방법으로 스레드를 생성한다.

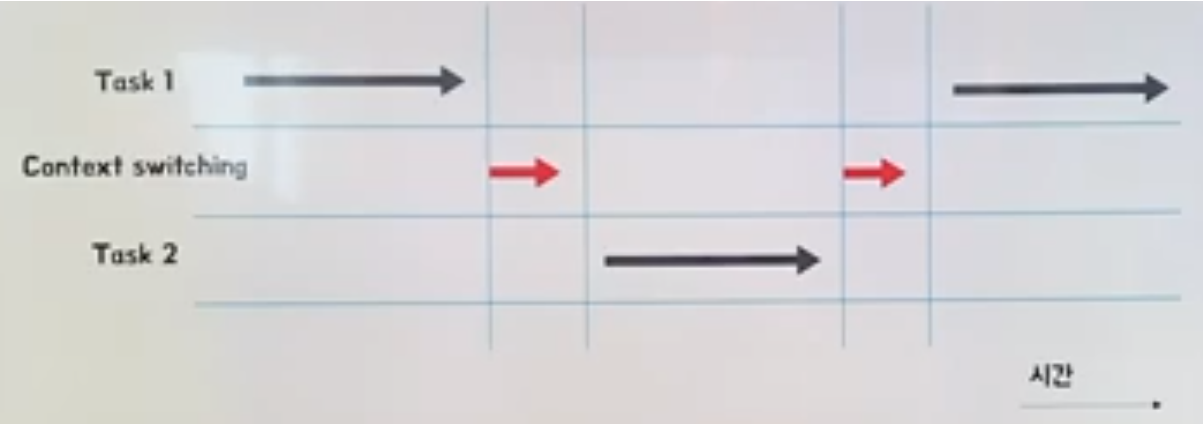
**Runnalbe Interface (구현부없는 run 메서드만 가지고있음)

```
public interface Runnable {
    /**
     * When an object implementing interface <code>Runnable</code> is used
     * to create a thread, starting the thread causes the object's
     * <code>run</code> method to be called in that separately executing
     * thread.
     * <p>
     * The general contract of the method <code>run</code> is that it may
     * take any action whatsoever.
     *
     * @see     java.lang.Thread#run()
     */
    public abstract void run();
```

우선 Thread 클래스를 상속받아 구현하는 경우에는 해당 객체를 생성하고 start()로 실행하면 된다.

Runnable을 구현한 경우에는 Thread 객체의 생성자로 해당 인스턴스를 넘겨주면 된다.

thread1은 10번 반복 thread2도 10번 반복해 각각의 Thread name을 출력하도록 구현했다. 결과를 보면 두개가 뒤섞여서 출력되는데, 각 쓰레드가 번갈아가면서 수행된 것이다.



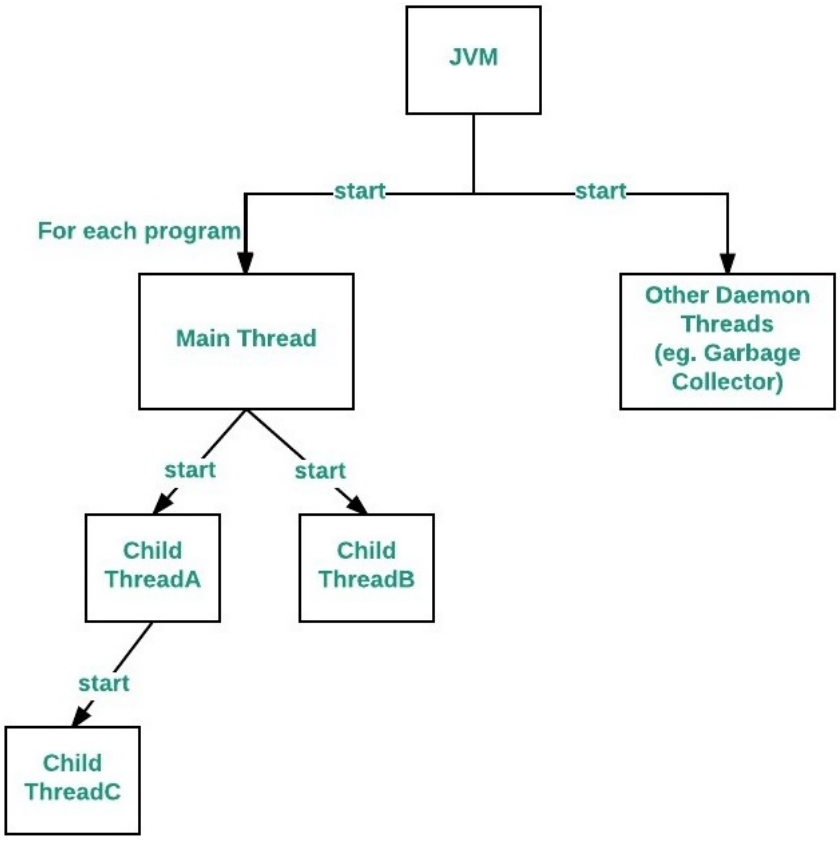
쓰레드는 OS의 스케쥴링에 따라 작업시간을 할당받고 다른 쓰레드와 번갈아가면서 작업을 수행한다. 같은 자원을 사용할 때아주 빠른 속도로 작업을 번갈아 수행하기 때문에 마치 동시에 실행되는 것과 같은 착각을 하게된다.

https://wisdom-and-record.tistory.com/48

http://www.tcpschool.com/java/java_thread_concept

Main Thread

main 메서드도 하나의 쓰레드이다. 이를 메인 쓰레드(Main Thread)라고 한다. 메인 쓰레드는 프로그램이 시작하면 가장 먼저 실행되는 쓰레드이며, 모든 쓰레드는 메인 쓰레드로부터 생성된다. 다른 쓰레드를 생성해서 실행하지 않으면, 메인 메서드, 즉 메인 쓰레드가 종료되는 순간 프로그램 종료된다. 하지만 여러 쓰레드를 실행하면, 메인 쓰레드가 종료되어도 다른 쓰레드가 작업을 마칠 때까지 프로그램이 종료되지 않는다. 쓰레드는 '사용자 쓰레드(user thread)'와 '데몬 쓰레드(daemon thread)'로 구분되는데, 실행 중인 사용자 쓰레드가 하나도 없을 때 프로그램이 종료된다.



https://www.geeksforgeeks.org/main-thread-java/?ref=lbp

```
public class ThreadDemo {

    public static void main(String[] args) {
        Thread t1 = Thread.currentThread();
        System.out.println("currentThread = " + t1);

        Thread t2 = new Thread(new ThreadEx_1());
        System.out.println("newThread = " + t2);
    }

}

class ThreadEx_1 implements Runnable {

    @Override
    public void run() {}
}


output :
currentThread = Thread[main,5,main]
newThread = Thread[Thread-0,5,main]
```

메인메서드에서 실행된 currentThread는 현재 실행중인 쓰레드의 참조를 반환하는 Static Method이다. Thread 클래스의 toString 메서드는 다음과 같이 구현되어있다.

```
public String toString() {
        ThreadGroup group = getThreadGroup();
        if (group != null) {
            return "Thread[" + getName() + "," + getPriority() + "," +
                           group.getName() + "]";
        } else {
            return "Thread[" + getName() + "," + getPriority() + "," +
                            "" + "]";
```

```
        }
    }
```

getName = 쓰레드이름

getPriority = 쓰레드의 우선순위

group.getName = 쓰레드가 속한 쓰레드 그룹의 이름이다.

메신 메서드에서 현재 쓰레드를 참조하면 main thread가 반환되는 것을 확인할 수 있다.

쓰레드의 상태

멀티쓰레드 프로그래밍을 잘하기 위해서는 정교한 스케줄리을 통해 자원과 시간을 여러 쓰레드가 낭비 없이 잘 사용하도록 해야 한다. 이를 위해서는 쓰레드의 상태와 관련 메서드를 잘 알아야 한다.

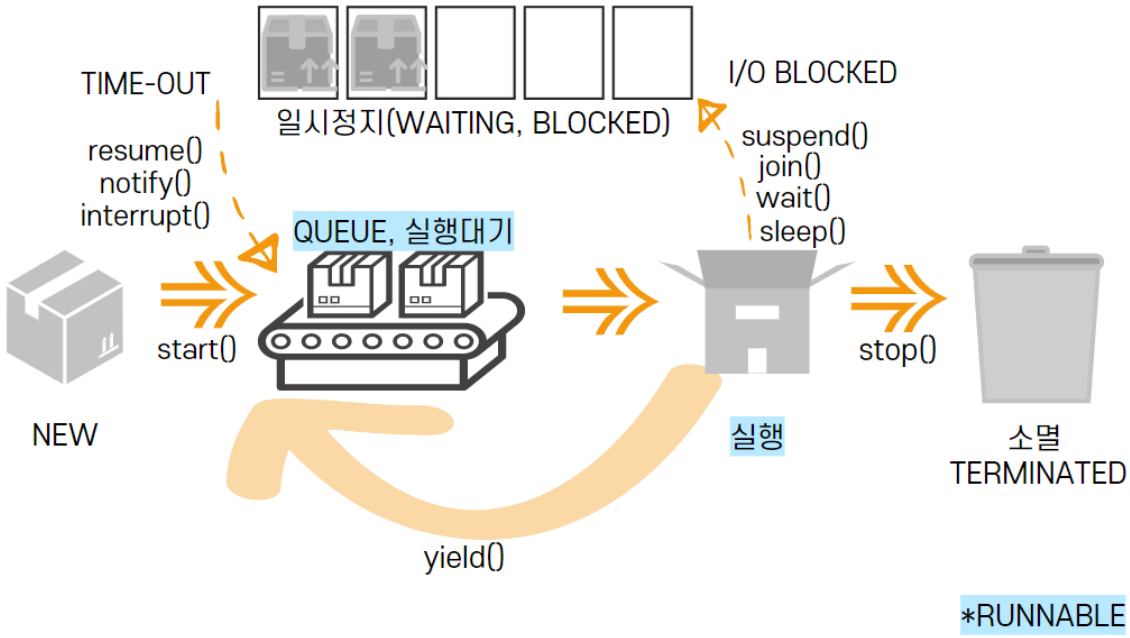
먼저 쓰레드에는 5가지 상태가 있는데

쓰레드의 상태

Aa 상태	설명
NEW	쓰레드가 생성되고 아직 start()가 호출되지 않은 상태
RUNNABLE	실행 중 또는 실행 가능한 상태
BLOCKED	동기화 블럭에 의해서 일시정지된 상태(lock이 풀릴 때까지 기다리는상태)
WAITING, TIMED_WAITING	쓰레드의 작업이 종료되지는 않았지만 실행 가능하지 않은 일시정지 상태 TIMED_WAITING은 일시정지 시간이 지정된 경우
TERMINATED	쓰레드의 작업이 종료된 상태
제목 없음	

JDK 1.5부터 getState() 메서드를 통해 쓰레드의 상태를 확인할 수 있다.

쓰레드의 상태는 메서드를 통해 제어 가능한데, 일반적으로 start()를 통해 쓰레드를 실행 가능한 상태로 만들면 run() 메서드에 의해 코드가 실행되고 모든 작업을 마치면 TERMINATED 상태가 되지만, 메서드를 이용해 쓰레드를 정지시키거나 다시 실행시킬 수 있다.



쓰레드의 실행 상태와 제어 메서드

Thread Method

Aa method	설명
static void sleep	지정된 시간동안 쓰레드를 일시정지 시킨다
void join	지정된 시간동안 쓰레드가 실행되도록 한다. join을 호출한 쓰레드는 그동안 일시정지가 되고 시간이 지나거나 작업이 종료되면 join을 호출한
void interrupt	sleep나 join에 의해 일시정지 상태인 쓰레드를 깨워서 실행대기 상태로 만든다.
void stop	쓰레드 즉시 종료
void suspend	쓰레드를 일시정지시킨다. resume()으로 재개
void resume	suspend에 의한 정지를 재개
static void yield	자신에게 주어진 실행시간을 다른 쓰레드에게 양보하고 자기는 실행대기 상태로 전환

📝 Thread Method의 활용(+locks, condition)

I/O BLOCKING

사용자의 입력을 받을때는 입력이 들어오기 전까지 해당 쓰레드가 일시정지 상태가 되는데 이를 I/O 블로킹이라고 한다.

한 쓰레드 내에서 입력을 받는일과 별개의 작업을 하는 코드가 있다면 사용자의 입력을 기다리는 시간동안 별개의 작업도 중지가 되기 때문에 CPU사용 효율이 떨어진다.

이 경우 두 작업을 쓰레드 두개로 불리해주면 더욱 효율적이다.

```
package com.oopsw.study;

import java.util.Scanner;

public class I_O_Blocking {

  public static void main(String[] args) {
    //사용자입력
    Scanner sc = new Scanner(System.in);
    System.out.println("InputData를 입력해주세요.");
    String input = sc.next();
    System.out.println("입력데이터 : " + input);

    //카운트다운
    int i = 10;
    while (i > 0){
      System.out.println(i);
      i--;
      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }

}
output :
InputData를 입력해주세요.
gpffh
입력데이터 : gpffh
10
9
8
7
6
5
4
3
```

```
2
1
```

```
package com.oopsw.study;

import java.util.Scanner;

public class I_O_Blocking_Multi {

  public static void main(String[] args) {
    Thread t  = new Thread(new SecondThread());
    t.start();
    //사용자입력
    Scanner sc = new Scanner(System.in);
    System.out.println("InputData를 입력해주세요.");
    String input = sc.next();
    System.out.println("입력데이터 : " + input);
  }
}

class SecondThread implements Runnable{

  @Override
  public void run() {
    //카운트다운
        int i = 10;
        while (i > 0){
          System.out.println(i);
          i--;
          try {
            Thread.sleep(1000);
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
        }

  }
output :
10
InputData를 입력해주세요.
qwd
9
8
7
입력데이터 : qwd
6
5
4
3
2
1
```

카운트 다운 쓰레드를 먼저 실행하였기 때문에 메인 쓰레드에서 사용자가 입력을 받더라도 카운트다운 작업은 계속 가능하다.

쓰레드의 우선순위

쓰레드는 우선순위(priority)라는 멤버 변수를 갖고 있다. 각 쓰레드별로 우선순위를 다르게 설정해줌으로써 **어떤 쓰레드에 더 많은 작업 시간을 부여할 것인가**를 설정해줄 수 있다. 우선순위는 1~10 사이의 값을 지정해줄 수 있으며 기본값으로 5가 설정되어 있다.

I_O_Blocking_Multi.java Thread.class

```
136  * @see     #run()
137  * @see     #stop()
138  * @since   JDK1.0
139  */
140 public
141 class Thread implements Runnable {
142     /* Make sure registerNatives is the first thing <clinit> does. */
143     private static native void registerNatives();
144     static {
145         registerNatives();
146     }
147
148     private volatile char  name[];
149     private int            priority;
150     private Thread         threadQ;
151     private long           eetop;
152
153     /* Whether or not to single_step this thread. */
154     private boolean     single_step;
155
156     /* Whether or not the thread is a daemon thread. */
157     private boolean     daemon = false;
158
159     /* JVM state */
160     private boolean     stillborn = false;
161
162     /* What will be run. */
163     private Runnable target;
164
165     /* The group of this thread */
166     private ThreadGroup group;
167
168     /* The context ClassLoader for this thread */
169     private ClassLoader contextClassLoader;
```

```
243
244     /**
245      * The minimum priority that a thread can have.
246      */
247     public final static int MIN_PRIORITY = 1;
248
249    /**
250      * The default priority that is assigned to a thread.
251      */
252     public final static int NORM_PRIORITY = 5;
253
254     /**
255      * The maximum priority that a thread can have.
256      */
257     public final static int MAX_PRIORITY = 10;
```

Priority의 최대값 10 최소값 1 과 기본값 5

```
1082     public final void setPriority(int newPriority) {
1083         ThreadGroup g;
1084         checkAccess();
1085         if (newPriority > MAX_PRIORITY || newPriority < MIN_PRIORITY) {
1086             throw new IllegalArgumentException();
1087         }
1088         if((g = getThreadGroup()) != null) {
1089             if (newPriority > g.getMaxPriority()) {
1090                 newPriority = g.getMaxPriority();
1091             }
1092             setPriority0(priority = newPriority);
1093         }
1094     }
1095
1096     /**
1097      * Returns this thread's priority.
1098      *
1099      * @return  this thread's priority.
1100      * @see     #setPriority
1101      */
1102     public final int getPriority() {
1103         return priority;
1104     }
```

setPriority와 getPriority

위의 코드를 참조해 쓰레드의 우선순위를 지정해줄 수도있다.

setPriority 메서드는 쓰레드를 실해아기 전에만 호출할 수 있다.

우선순위를 높이면 더 많은 실행시간과 실행기회를 부여받을 수 있다.

> 📌 하지만 무조건! 보장되는 것이 아니라는 점은 주의해야 한다.

쓰레드의 작업 할당은 OS의 스케쥴링 정책과 JVM의 구현에 따라 다르기 때문에 코드에서 우선순위를 지정하는 것은 단지 희망사항을 전달하는 것일 뿐, 실제 작업은 내가 설정한 것과 다르게 진행될 수 있다.

동기화 (Synchronized)

멀티 쓰레드 프로세스에서는 여러 프로세스가 메모리를 공유하기 때문에, 한 쓰레드가 작업하던 부분을 다른 쓰레드가 간섭하는 문제가 발생할 수 있다.

어떤 쓰레드가 진행 중인 작업을 다른 쓰레드가 간섭하지 못하도록 하는 작업을 동기화라고 한다.

동기화를 하려면 다른 쓰레드가 간섭해서는 안되는 부분(임계영역)을 설정해 주어야 하는데 이때 synchronized 키워드를 사용한다.

```
// 메서드 전체를 임계영역으로 설정
public synchronized void method1 () {
    ......
}

// 특정한 영역을 임계영역으로 설정
synchronized(객체의 참조변수) {
    ......
}
```

synchronized 키워드가 붙은 객체는 lock을 얻어 작업을 수행하다가 종료시에 lock을 반납한다.

여기서 Lock이란?

lock은 일종의 자물쇠 개념이다. 모든 객체는 lock을 하나씩 가지고 있는데,

해당 객체의 lock을 가지고 있는 쓰레드만 임계 영역의 코드를 수행할 수 있다. 한 객체의 lock은 하나밖에 없기 때문에 다른 쓰레드들은 lock을 얻을 때까지 기다리게 된다.

임계 영역은 멀티쓰레드 프로그램의 성능을 좌우하기 때문에 가능하면 메서드 전체에 lock을 거는것 보다 **synchronized 블록으로 임계 영역을 최소화하는 것이 좋다.**

```
package com.oopsw.study;

public class Synchronized_Thread {
  public static void main(String[] args) {
    Runnable r = new ThreadEx();
    new Thread(r).start();
    new Thread(r).start();
  }
}
class ThreadEx implements Runnable{

  Account account = new Account();

  @Override
  public void run() {
    while (account.getBalance() > 0){
      int money = (int) ((Math.random() * 3 * 1) * 100);
      account.withdraw(money);
      System.out.println("balance : "+ account.getBalance());
    }
  }
}
class Account{
  private int balance = 1000;

  public int getBalance(){
    return balance;
  }

  public void withdraw(int money){
    if (balance >= money){
      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) {
      }
```

```
      balance -= money;
    }
  }
}

output:
balance : 782
balance : 782
balance : 606
balance : 606
balance : 555
balance : 555
balance : 554
balance : 554
balance : 552
balance : 552
balance : 479
balance : 479
balance : 302
balance : 67
balance : 67
balance : -16
balance : 11
```

분명 balance는 음수가 나오지 않도록 설계했는데 음수가 나왔다. 왜냐하면 쓰레드 하나가 if문을 통과하면서 balance를 검사하고 순서를 넘겼는데, 그 사이에 다른 쓰레드가 출금을 실시해서 실제 balance가 if문을 통과할 때 검사했던 balance보다 적어지게 되고, 이미 if문을 통과한 쓰레드에서는 출금이 이루어지게 되고 음수가 나오는 것이다.

이 문제를 해결하려면 출금하는 로직에 동기화를해서 한 쓰레드가 출금 로직을 시랭할 동안 다른 쓰레드가 출금 블록에 들어오지 못하도록 해줘야한다.

```
package com.oopsw.study;

public class Synchronized_Thread {
  public static void main(String[] args) {
    Runnable r = new ThreadEx();
    new Thread(r).start();
    new Thread(r).start();
  }
}
class ThreadEx implements Runnable{

  Account account = new Account();

  @Override
  public void run() {
    while (account.getBalance() > 0){
      int money = (int) ((Math.random() * 3 * 1) * 100);
      account.withdraw(money);
    }
  }
}
class Account{
  private int balance = 1000;

  public int getBalance(){
    return balance;
  }

  public synchronized void withdraw(int money){
    if (balance >= money){
      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) {

      }
      balance -= money;
      System.out.println("balance : "+ getBalance());
    }
  }
}
output:
balance : 865
balance : 706
balance : 417
balance : 246
balance : 195
balance : 100
balance : 87
balance : 86
balance : 59
balance : 55
```

```
balance : 8
balance : 3
balance : 3
balance : 0
```

위는 수정한 코드이다

withdraw에 동기화를 설정해주었고 출력도 반복문에 의해 동일하게 계속 나와 위치를 변경해주었다.

이렇게 동기화를 통해 쓰레드를 통제해 주면된다.

DeadLock (교착상태)

데드락은 쉽게 말해서 오도가도 못하는 난처한 상태이다.

공유 객체에 대해 복수의 쓰레드가 서로 다른 쓰레드의 실행이 끝나기를 기다리는 상태를 말한다.

예제로 데드락을 살펴보자

```
package com.oopsw.study;

public class DeadLock {
  public static Object myLockObj_1 = new Object();
  public static Object myLockObj_2 = new Object();

  public static void main(String[] args) {
    MyThread_1 myThread_1 = new MyThread_1();
    MyThread_2 myThread_2 = new MyThread_2();

    myThread_1.start();
    myThread_2.start();
  }
  static class MyThread_1 extends Thread{
    @Override
    public void run() {
      synchronized (myLockObj_1){
        System.out.println("Thread 1: Holding [myLockObj_1]");
        try {
          Thread.sleep(1000);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }System.out.println("Thread 1 : Waiting for [myLockObj_1]");
        synchronized (myLockObj_2){
          System.out.println("Thread 1 : Holding [myLockObj_1] & [myLockObj_2]");
        }
      }
    }
  }
  static class MyThread_2 extends Thread{
    @Override
    public void run() {
      synchronized (myLockObj_2){
        System.out.println("Thread 2: Holding [myLockObj_1]");
        try {
          Thread.sleep(1000);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }System.out.println("Thread 2 : Waiting for [myLockObj_1]");
        synchronized (myLockObj_1){
          System.out.println("Thread 2 : Holding [myLockObj_1] & [myLockObj_2]");
        }
      }
    }
  }
}
output:
Thread 1: Holding [myLockObj_1]
Thread 2: Holding [myLockObj_1]
Thread 2 : Waiting for [myLockObj_1]
Thread 1 : Waiting for [myLockObj_1]
...ing
```

Threa_1은 시작 하자마자 myLockObj_1 객체에 락을 걸고 이후 myLockObj_2에 락을 건다.

Threa_2은 시작 하자마자 myLockObj_2 객체에 락을 걸고 이후 myLockObj_1에 락을 건다는 간단한 예제이다.

하지만 자신들이 락을 건 객체를 서로 사용하려 하기 때문에 누구 한명의 락이 먼저 끝나기를 무한정 기다리는 상황이다.

데드락이 걸리는 경우는 다음의 네 가지 조건이 동시에 성립할 때 발생하는데,

▼ 상호 배제 (Mutual exclusion)

자원은 한 번에 한 프로세스만이 사용할 수 있어야 한다.

▼ 점유 대기 (Hold and wait)

최소한 하나의 자원을 점유하고 있으면서 다른 프로세스에 할당되어 사용하고 있는 자원을 추가로 점유하기 위해 대기하는 프로세스가 있어야 한다.

▼ 비선점 (No preemption)

다른 프로세스에 할당된 자원은 사용이 끝날 때까지 강제로 빼앗을 수 없어야 한다.

▼ 순환 대기 (Circular wait)

프로세스의 집합 {P0, P1, ,...Pn}에서 P0는 P1이 점유한 자원을 대기하고 P1은 P2가 점유한 자원을 대기하고 P2...Pn-1은 Pn이 점유한 자원을 대기하며 Pn은 P0가 점유한 자원을 요구해야 한다.

(출처: https://jwprogramming.tistory.com/12 [개발자를 꿈꾸는 프로그래머])

이를 해결하기 위해서는 어느 한쪽을 강제로 종료하는 방법밖에 없다고 한다.