DAY 4

Error

- **Compile Time Error**
- **Runtime Error**
  - **Error**
    - 프로그램이 지속될 수 없는 오류
  - **Exception**
    - 프로그램적으로 해결가능한 오류
    - `Exception` 클래스는 해결가능한 오류의 최상위 클래스이다.
      - `nullpoint exception`
      - …
    - 프로그램을 실행하다가 프로그램이 정상적인 수행이 안되면 JVM 이 현재 오류실행에 대한 정보를 모아 `instance`를 생성한다.
      - **이 객체를 우리가 프로그램적으로 처리를 해야한다**.
        - 잘 처리된다면 프로그램이 강제 종료되지 않고 지속적인 수행이 가능하다.
        - `Exception`을 처리하지 않으면 프로그램이 강제 종료된다.

try catch

```
package lecture0714;

public class Main {
  public static void main(String[] args) {
    System.out.println("프로그램 시작!");

    // 오류가 발생하면 exception 이라는 객체가 만들어진다.
    // 이 exception 객체를 처리해야한다.
    int result = 10 / 0; // error 발생

    System.out.println("프로그램 종료");
  }
}
```

```
프로그램 시작!
Exception in thread "main" java.lang.ArithmeticException: / by zero
  at lecture0714.Main.main(Main.java:9)
```

`try, catch` 구문을 이용해서 오류를 처리할 수 있다❗

```
package lecture0714;

public class Main {
  public static void main(String[] args) {
    System.out.println("프로그램 시작!");

    try {
      int result = 10 / 0; // error 발생
    } catch (ArithmeticException aaa) { // ArithmeticExceptions 은 타입객체
      // exception 을 처리
      System.out.println("오류가 발생했어요!");
    }

    System.out.println("프로그램 종료");
  }
}
```

```
프로그램 시작!
오류가 발생했어요!
프로그램 종료
```

`catch` 구문을 여러개 이용해서 다양한 오류를 처리할 수 있다❗

```
package lecture0714;

public class Main {
  public static void main(String[] args) {
    System.out.println("프로그램 시작!");

    try {
      Object obj = null;
      // 객체가 없는데 toString을 호출하려고 하니 NullPointerException 발생
      System.out.println(obj.toString());
    } catch (ArithmeticException aaa) {
      System.out.println("Arithmetic Error!!!");
    } catch (NullPointerException e) {
      System.out.println("NULL Error");
    }

    System.out.println("프로그램 종료");
  }
}
```

finally

`Exception` 이 발생하지 않아도 오류와 관계없이 무조건 실행❗

Exception

IS -A 관계에 의해서 `ArithmeticException`, `NullPointerException` 등의 모든 상위 클래스는 `Exception` 이다.

```
package lecture0714;

public class Main {
  public static void main(String[] args) {
    System.out.println("프로그램 시작!");
```

```
    try {
      Object obj = null;
      System.out.println(obj.toString());
      int result = 10 / 0;
    } catch (ArithmeticException aaa) {
      System.out.println("ArithmeticException 오류가 발생했어요!");
    } catch (Exception e) {
      System.out.println("오류가 발생했어요!");
    } finally {
      // 오류에 관계없이 무조건 실행된다.
    }

    System.out.println("프로그램 종료");
  }
}
```

```
프로그램 시작!
오류가 발생했어요!
프로그램 종료
```

`Exception` 은 모든 오류의 최상단 클래스이므로 `catch` 문에서 제일 위에 나오게되면 모든 오류를 다 잡아버리므로 제일 하단에 적는것이 좋다 ❗

Thread

- **Thread 가 무엇인지?**
- **중요한 용어, class 와 method**
- **Thread 는 Network Server 를 만들기 위해 활용한다.**

Process

> 💡 **Process = Resource + Thread**

- **현재 실행중인 프로그램을 말한다.**
- **프로그램을 실행시키기 위해 OS로부터 Resource를 할당 받아야 한다!**
- **최소 1개 이상의 Thread가 존재**
  - **한 개라면 Single-Thread Program**
  - **두 개 이상이라면 Multi-Thread Program**

Resource

1. **code : 프로그램 실행 코드**
2. **data : 데이터**

3. **heap**
4. **stack**

Thread

- Execution Stack을 별도로 가지고 있는 실행흐름이다.
- Thread는 스택이 각각 다르게 할당된다.
- Main method는 하나의 Main Thread이다.
- 여태까지 수업 중 작성한 Program은 Single-Thread Program이다.

> 💡 **Main Method != Threa**
> **라고 생각할 수 있는데 다르다❗**
> **JVM이 이 Thread를 만들고 그 Thread가 main method를 호출하는 것이다.**

CPU (core)

1개의 Core는 한가지 일만 수행할 수 있다.

하지만 Time Slicing (시분할 기법)을 사용해 우리눈에 여러가지 Program이 동시에 실행되는것 처럼 보일 수 있다.

Multi-tasking

실제로 여러 개의 Program이 동시에 동작하는것을 말한다.

즉, Core의 개수가 2개 이상일 때 가능하다.

Multi-Threading

> 💡 하나의 Program에는 여러개의 Thread가 있는데 이 때,
> Multi-Processing을 이용해서 여러 개의 Thread를 동시에 실행하는것을 말한다.

- **장점 : 효율적인 처리**
  - OS로 부터 할당받는 **Resource를 공유**하기 때문에 효율적인 처리가 가능하다
    - 즉, **응답속도가 빠르다**.
- **단점 : program이 어렵다.**
  - Resource(공유자원)에 대한 **동기화**를 시키는게 어렵다.
  - 제대로 처리하지 못하면 **Deadlock**에 빠질 수 있다

Thread 이용하는 방법

> 💡 Java에서 **Thread는 Instance로 존재**한다.
> 즉, Class를 이용해서 Instance를 만들어서 사용해야 한다.

1. Thread class를 상속받는다.

- **User Define Thread Class**를 이용한다.
  - 하지만 일반적으로 이 방법은 **사용하지 않는다**.
- Java에서 class는 단일상속만 가능하다.
  - 직접 상속을 받게되면 두 개의 class가 **tightly coupled**되기 때문에 **재사용 하기 어렵다**.

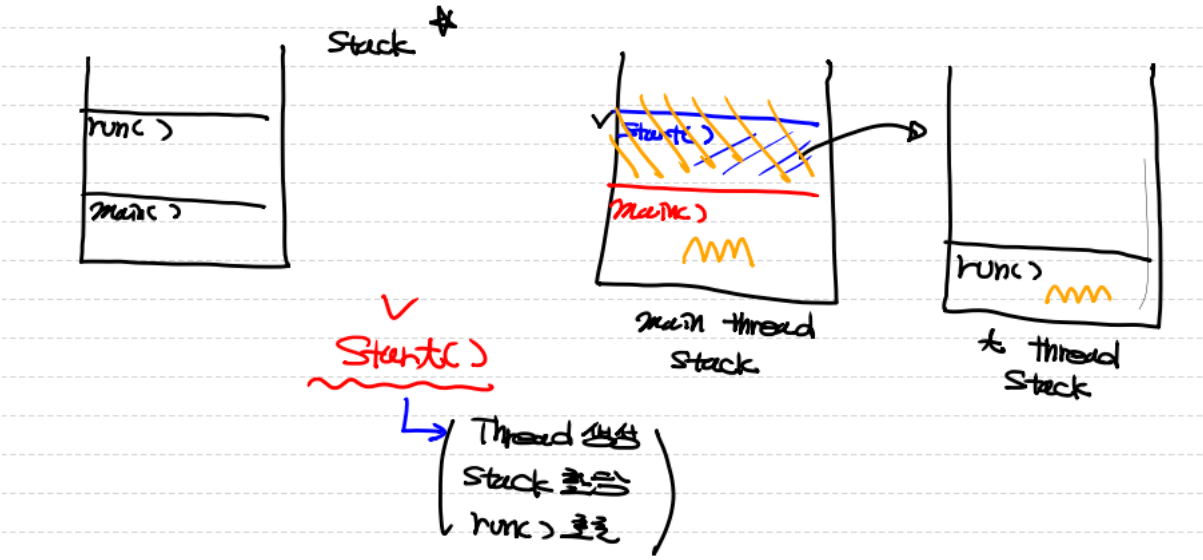
2. Runnable Interface를 이용한다.

- **Runnable Interface**를 구현해서 **User Define class**를 만들고
- User Define class의 **Instance**를 `new Thread()` 안의 **parameter로 전달**한다.

Thread class

```
package lecture0714;

  class MyThread extends Thread {
    // 우리 나름대로의 실행흐름을 만들기 위해서 Thread class 의 run 메서드를 overriding 한다.
    @Override
    public void run() {
      System.out.println("Hello");
    }
  }

  public class Main {
    public static void main(String[] args) {
      MyThread t = new MyThread();
      t.start(); // 인스턴스의 메서드 호출 -> 새로운 실행흐름이 만들어지지 않고 stack 에 올라간다.
    }
  }
```

> ✅ `t.start ( )` 를 호출하게 되면 밑에 있는 그림처럼 `t Thread`를 위한 Stack이 만들어지고 `run ( )` 을 호출하게 된다. 그 후에 `start( )` 는 Main Thread Stack에서 사라지고 `main( )` 과 `run( )`은 따로 동작한다.



Runnable Interace

```
class MyThread extends Thread {
  @Override
  public void run() {
    System.out.println("Hello");
  }
}

// MyThread2 는 단지 Runnable 을 구현한 클래스일 뿐이다. (Thread가 아니다)
class MyThread2 implements Runnable {
  // Runnable 이 interface 이기 때문에 run 메서드를 반드시 구현해야 한다.
  @Override
  public void run() {
    System.out.println("이것도 실행돼요!");
  }

}

public class Main {
  public static void main(String[] args) {
    MyThread t = new MyThread();
    t.start();

    MyThread2 s = new MyThread2();
    // MyThread2 는 Thread 가 아니고 그냥 class 이므로 실제 Thread 안의 인자로 주입시켜야 한다.
    Thread t1 = new Thread(s);
    t1.start();

    System.out.println("안녕하세요!");
  }
}
```

✅ `Thread`를 직접 상속받는것이 아니라 `Runnable Interface`를 구현해서 `User Define class`를 만들고`MyThread2 s = new MyThread2( );`를 통해 만든 Instance를 `Thread t1 = new Thread(s);` 실제 `Thread` 안의 인자로 주입시켜서 사용한다.

Thread의 상태 전이도

1. `new` keyword를 통해 객체를 생성하고 `start( )` keyword를 통해 Thread가 실행되면 Runnable 상태가 된다. 즉 실제로 실행이 되는것이 아니라 실행이 가능한 상태를 의미한다.
2. 이때 JVM 내에 위치한 Thread Scheduler가 자료구조와 알고리즘을 이용해 Runnable 상태에 있는 Thread들을 선택해 Running상태로 만든다.
3. Running상태에서 Core가 `run( )`을 실행시킨다.
4. 하나의 Thread를 계속 실행시키는게 아니라 Thread Scheduler에 의해서 여러개의 Thread를 번갈아가면서 실행한다. 즉, 여러개의 Thread가 실행될때 Thread Scheduler에 의해서 Runnable과 Running 상태를 왔다갔다한다.
5. `run( )` 이 종료되면 Dead상태로 가게되고 다시는 이 객체를 재활용할 수 없다.

Daemon Thread

- **다른 일반 Thread의 보조적인 작업을 하기 위해 사용한다.**
  - 대표적인 Daemon Thread로는 **Garbage collection**이 있다.
  - Daemon Thread는 이 Thread를 **파생시킨 Thread가 종료**되면 **같이 종료**된다.
- `t1.setDaemon(true);` 와 같이 사용한다.

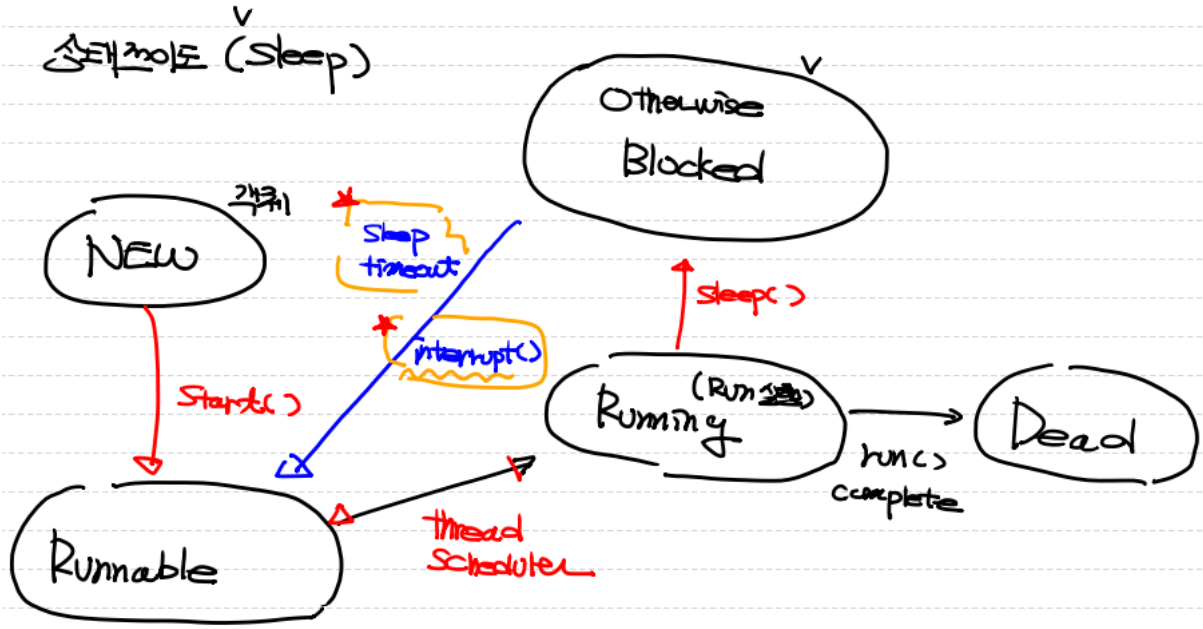
기억해야하는 thread의 method

Sleep

- 일정시간동안 Thread를 중지시켜요.
  - 지정된 시간이 다 되었을 때
  - Thread에 대해 interrupt ( )가 호출되면 Interrupted Exception이 발생하면서 sleep에서 깨어난다.
    - 항상 `try ~ catch`를 써야한다.

```
public class ThreadExam02 implements Runnable {
  static boolean autoSave = false;

  public static void main(String[] args) {
    Thread t = new Thread(new ThreadExam02());

    // t 를 파생시킨 것은 main thread 이므로 main thread 가 끝나면
    // 이 Daemon Thread 도 끝날 것이다.
    t.setDaemon(true);
    t.start();

    for (int i = 0; i < 10; i++) {
      try {
        // 1초동안 잔다.
        Thread.sleep(1000);
      } catch (Exception e) {

      }
      System.out.println(i);

      // i 가 5가 되는 순간 autoSave 가 true 가 되면서
      if (i == 5) {
        autoSave = true;
      }
    }
  }

  @Override
  public void run() {
    while (true) {
      try {
        // Thread 가 수행되다가 sleep 메서드를 만나면 잔다.
        // 3초동안 잔다.
        Thread.sleep(3000);
      } catch (InterruptedException e) {

      }

      // i가 5가 된 이후에 이 아래의 것이 실행된다.
      if (autoSave) {
        System.out.println("자동저장되었어요!");
      }
    }
  }
```

Sleep 상태전이도



> 💡 **Thread sleep**
>
> - **Thread**가 `sleep` 상태에서 `sleep timeout`이 되거나 `interrupt( )`되면 **Runnable** 상태로 빠지고 **interrupt**의 값을 `false`로 바꾼다.
> - `sleep` method는 현재 실행중인 Thread를 재운다.
>   - `main( )` 안에 `t1.sleep`을 쓰더라도 Main Thread가 잔다 ❗

Thread 실행 후 작업이 끝나기 전에 해당 Thread를 중지 시키고 싶으면 어떻게 해야할까?

> 💡 `interrupt( )`
>
> - 직접 중지시키지 않는다. Thread 내부 상태값인 Interrupted state를 바꾼다.
>
> `interrupted( )`
>
> - Thread class의 static
>   - 현재 실행중인 Thread의 interrupt 여부만 알 수 있다.
>   - 상태값을 조사하고 그 상태값을 false로 변경한다.
>
> `isInterrupted( )`
>
> - Instance method 특정 Thread의 interrupt 여부를 알 수있다.
>   - 현재 상태값만 return한다.

Interrupt Code

```
import javax.swing.JOptionPane;

class ThreadEx_04 extends Thread {

}

public class ThreadExam04 {
  public static void main(String[] args) {
    Thread t = new ThreadEx_04();
    // Dynamic Binding: 진짜 Thread 가 가지고 있는 start 가 아닌 밑의 ThreadEx_04 가 가지고 있는 start 가 호출됨
    t.start();

    String input = JOptionPane.showInputDialog("값을 입력하세요!");
    System.out.println("입력값은: " + input);

    // Thread 의 내부 상태 정보 interrupted 를 true 로 바꾼다.
    t.interrupt();
    System.out.println("Thread 상태값은: " + t.isInterrupted());
  }
}
```

```
입력값은 : 수민이❤
Thread 상태값은: false
```

Q. `t.interrupt()` 라는 method를 이용했는데도 Thread의 Interrupt 상태값이 false인 이유는 무엇일까?

A. `Therad t = new ThreadEx_04();` 를 이용해서 객체를 만들었는데 위에 Code에 보면 `ThreadEx_04` 객체 안에 아무것도 정의 되어있지 않다. 그래서 `Interrupt` 를 걸더라도 걸리지 않는것이다.

Interrupt Code 2

```
package lecture0714;

import javax.swing.JOptionPane;

class ThreadEx_04 extends Thread {
  @Override
  public void run() {
    int i = 10;
    while (i != 0 && !isInterrupted()) {
      System.out.println(--i);
      try {
        Thread.sleep(4000);
       // for (long k = 0; k < 2500000000L; k++); // 일정량의 시간을 끄는 더미코드

      } catch (Exception e) {
        interrupt();
        System.out.println("Thread 상태값 2 : " + isInterrupted());
        System.out.println("Count End!");
      }
    }
  }
}

public class ThreadExam04 {
  public static void main(String[] args) {
    Thread t = new ThreadEx_04();
    t.start();
    String input = JOptionPane.showInputDialog("값을 입력하세요!");
// blocking method 이므로 수행자체가 값을 입력하고 OK 누르기전까지 멈춰있는다.
```

```
    System.out.println("입력값은 :" + input);

    t.interrupt();
    System.out.println("Thread 상태값 1 : " + t.isInterrupted());
  }
}
```

```
9
8
7
입력값은 :수민이❤
Thread 상태값 1 : false
Thread 상태값 2 : true
Count End!
```

결과값을 보면 Thread의 상태값 1은 자고있을때 `t.interrupt()` 걸었기 때문에 자고있던 Thread가 잠에서 깨어나면서 Interrupt의 상태값을 false로 초기화시키기 때문에 상태값 1 은 false로 찍히는것이다. 그 후에 `catch` 안에 `interrupt()` 를 걸어주면 `t.interrupt();` 를 하더라도 exception에 잡히기 때문에 `interrupt` 의 값을 true로 변경한다.

depercate method

stop,suspend,resume

```
package lecture0714;

class ThreadEx_06 implements Runnable {
  volatile boolean suspended = false; // volatile 캐시메모리에서 변수값 가져오지말고 memory에서 직접 땡겨오게하는 명령어
  volatile boolean stopped = false;

  @Override
  public void run() {
    while (!stopped) {
      if (!suspended) {
        System.out.println(Thread.currentThread().getName());
        try {
          Thread.sleep(1000);
        } catch (Exception e) {
          // TODO: handle exception
        }
      }
    }
  }

  public void suspend() {
    suspended = true;
  }

  public void stop() {
    stopped = true;
  }

  public void resume() {
    suspended = false;
  }
}

public class ThreadExam06 {
```

```
  public static void main(String[] args) {
    ThreadEx_06 r1 = new ThreadEx_06();
    ThreadEx_06 r2 = new ThreadEx_06();
    ThreadEx_06 r3 = new ThreadEx_06();

    Thread T1 = new Thread(r1, "*");
    Thread T2 = new Thread(r2, "**");
    Thread T3 = new Thread(r3, "***");

    T1.start();
    T2.start();
    T3.start();
    try {
      Thread.sleep(2000);
      r1.suspend(); // T1 일시정지
      Thread.sleep(2000);
      r2.suspend();
      Thread.sleep(2000);
      r1.resume(); // T1 다시 동작
      Thread.sleep(3000);
      r2.stop();
      Thread.sleep(2000);
      r3.suspend();

    } catch (Exception e) {
      // TODO: handle exception
    }
  }
}
```

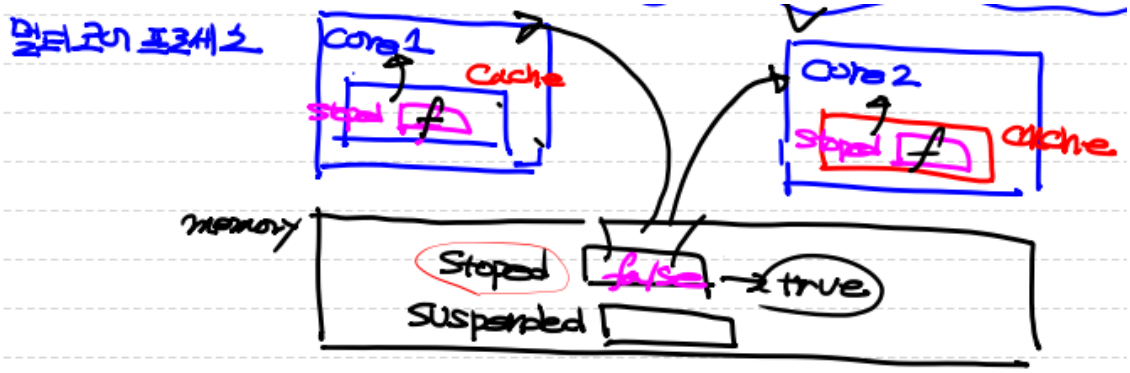
depercate된 method를 사용할려면 위와 같은 로직으로 depercated된 `stop()` `suspend()` `resume()` 와 같은 method를 overring 하여 재정의 해서 사용할 수 있다.

하지만 저렇게 사용하면 `r1.resume` 으로 다시 동작하도록 했을때 정상적으로 작동하지 않는다.

Q. 로직에는 문제가 없는데 왜 다시 동작하지 않을까??

> 💡 **답은 멀티코어 프로세스의 Cache안에 있다. 멀티코어 프로세스에 대해서 알아보자 ❗**

Multi-Core Process



- Multi-Core Process는 값을 읽어올때 Memory에서 한번 값을 읽어오게되면 그 값을 **Core**안의 **Cache에 저장**해두고 가져온다.

    - 값을 읽어올 때 마다 memory에서 가져오게되면 매번 access를 해서 값을 얻어와야 하기때문에 속도가 느리다.

- 그래서 `r1.resume();` method로 `suspend`의 값을 `flase`로 변경해주더라도 **memory**에 있는 `suspend`의 값이 `false`로 변경되고 실제 **Core**가 꺼내쓰는 **Chace**안에 있는 `suspend`의 값은 그대로 `true`로 있게된다.

Q. 그럼 Core가 Cache말고 Memory에서 access하게 하는 방법은 없을까??

> 💡 `volatile` keyword를 사용하면 **매번 memory에서 값을 읽어온다** ❗

yield

- Thread 가 자신에게 주어진 실행시간을 다 쓰지 않고 다른 Thread 에게 양보한다.
    - 프로그램의 응답성을 높이기 위해 사용한다.

```
package lecture0714;

class ThreadEx_06 implements Runnable {
  volatile boolean suspended = false;
  volatile boolean stopped = false;

  @Override
  public void run() {
    while (!stopped) {
      if (!suspended) {
        System.out.println(Thread.currentThread().getName());

        try {
          Thread.sleep(1000);
        } catch (Exception e) {
          // TODO: handle exception
        }
      } else {
        Thread.yield();
      }
    }
  }

  public void suspend() {
    suspended = true;
  }

  public void stop() {
    stopped = true;
  }

  public void resume() {
    suspended = false;
  }
}

public class ThreadExam06 {
  public static void main(String[] args) {
    ThreadEx_06 r1 = new ThreadEx_06();
    ThreadEx_06 r2 = new ThreadEx_06();
    ThreadEx_06 r3 = new ThreadEx_06();

    Thread t1 = new Thread(r1, "*");
```

```
    Thread t2 = new Thread(r2, "**");
    Thread t3 = new Thread(r3, "***");

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

    try {
      Thread.sleep(2000);
      r1.suspend();
      Thread.sleep(2000);
      r2.suspend();
      Thread.sleep(3000);
      r1.resume();
      Thread.sleep(3000);
      r1.stop();
      r2.stop();
      Thread.sleep(2000);
      r3.stop();
    } catch (Exception e) {

    }
  }
}
```

`if(!suspended)` 에서 `suspend` 하더라도 **계속 while 문이 돌기 때문에 미친듯이 CPU 를 소모**하고 있다. 따라서 아래와 같이 `Thread.yield()` 를 이용하여 다른 것에 **양보**해야한다.

yield 상태전이도