4. 스레드 제어와 생명 주기2

#1.인강/0.자바/5.자바-고급1편

- /인터럽트 - 시작1
- /인터럽트 - 시작2
- /인터럽트 - 시작3
- /인터럽트 - 시작4
- /프린터 예제1 - 시작
- /프린터 예제2 - 인터럽트 도입
- /프린터 예제3 - 인터럽트 코드 개선
- /yield - 양보하기
- /프린터 예제4 - yield 도입
- /정리

인터럽트 - 시작1

특정 스레드의 작업을 중간에 중단하려면 어떻게 해야할까?

다음 코드를 보자.

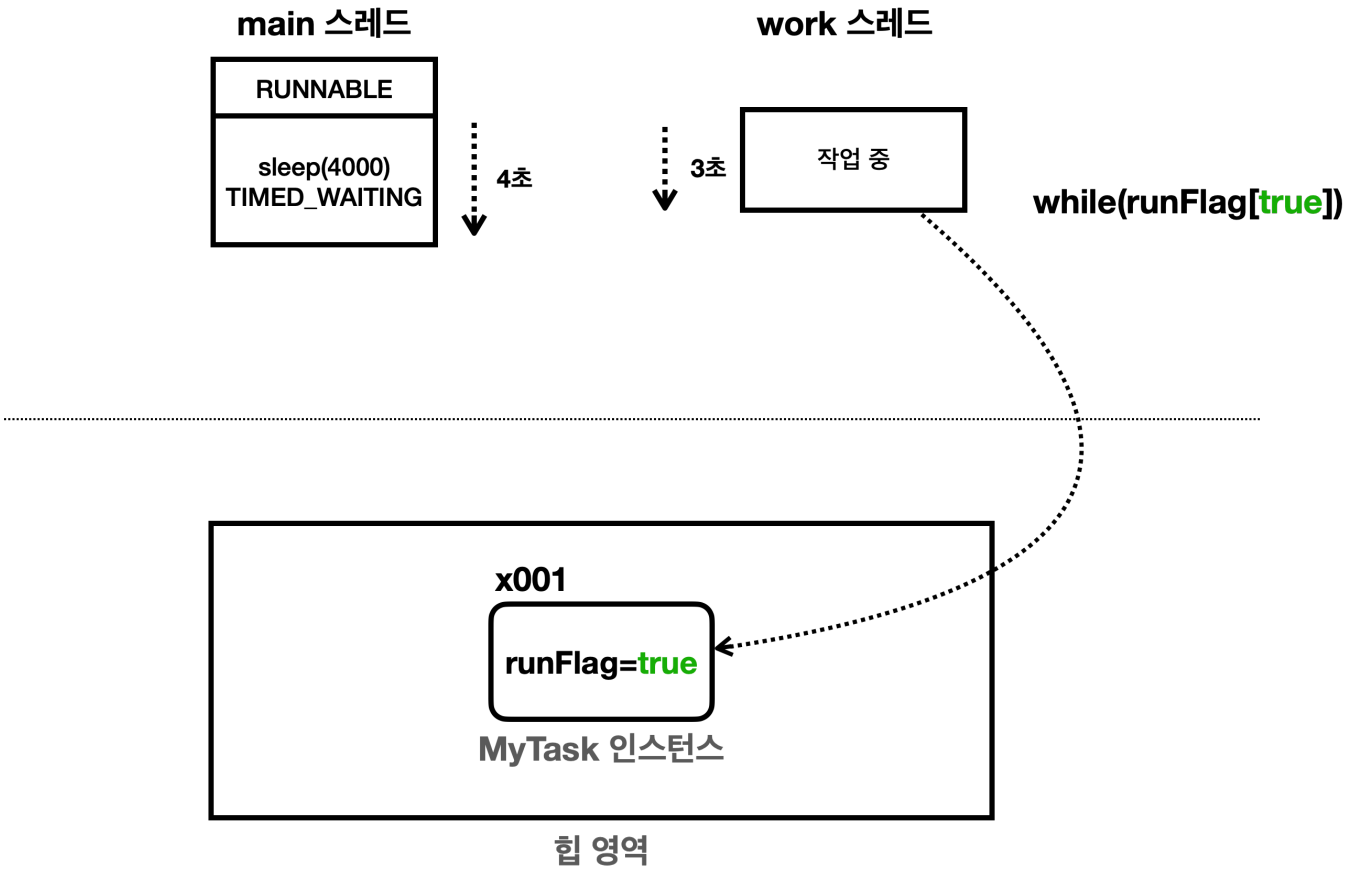
```
package thread.control.interrupt;

import static util.MyLogger.log;
import static util.ThreadUtils.sleep;

public class ThreadStopMainV1 {

    public static void main(String[] args) {
        MyTask task = new MyTask();
        Thread thread = new Thread(task, "work");
        thread.start();

        sleep(4000);
        log("작업 중단 지시 runFlag=false");
        task.runFlag = false;
    }

    static class MyTask implements Runnable {
        volatile boolean runFlag = true;
```

```
        @Override
        public void run() {
            while (runFlag) {
                log("작업 중");
                sleep(3000);
            }
            log("자원 정리");
            log("작업 종료");
        }
    }
}
```

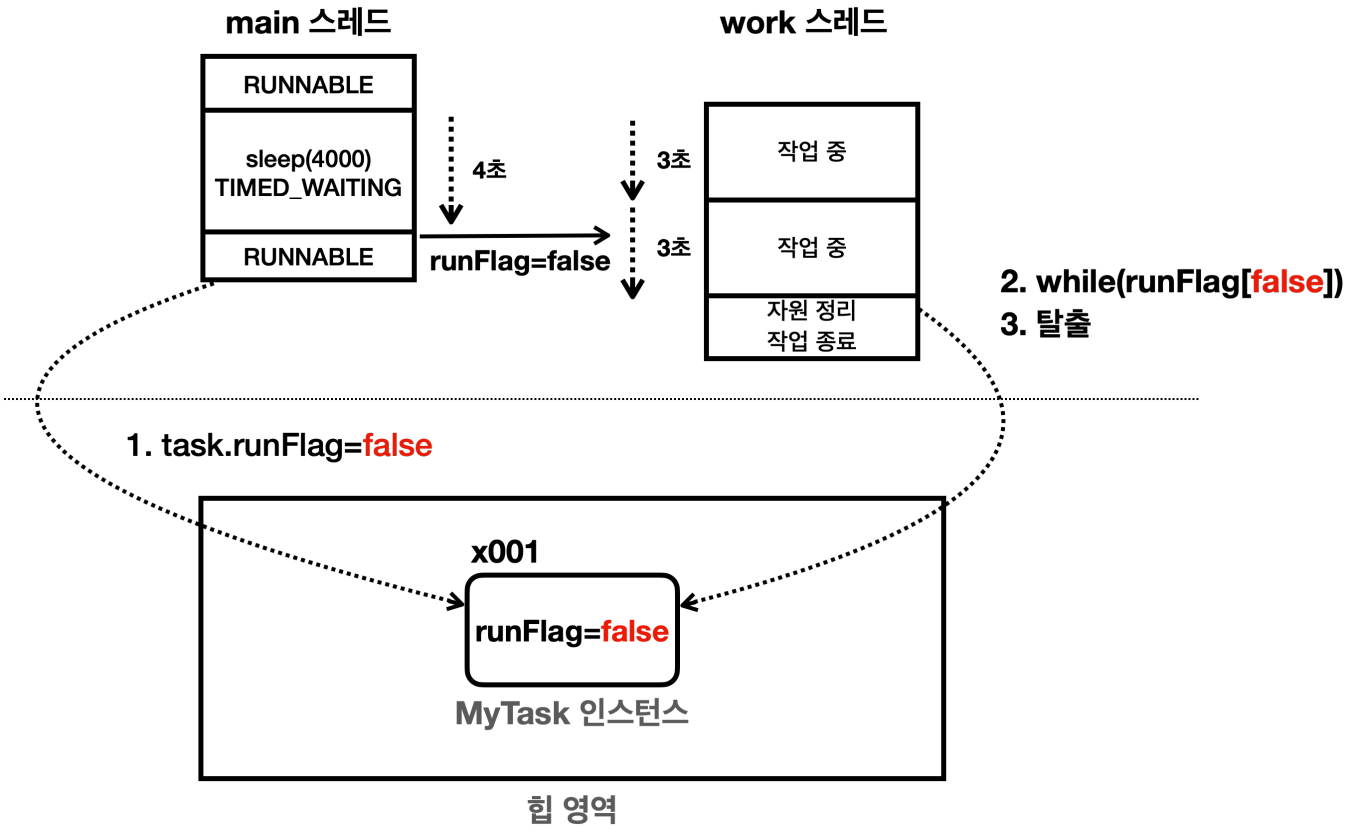
- 특정 스레드의 작업을 중단하는 가장 쉬운 방법은 변수를 사용하는 것이다.
- 여기서는 `runFlag`를 사용해서 `work` 스레드에 작업 중단을 지시할 수 있다.
- 작업 하나에 3초가 걸린다고 가정하고, `sleep(3000)`을 사용하자.
- `main` 스레드는 4초 뒤에 작업 중단을 지시한다.
- `volatile` 키워드는 뒤에서 자세히 설명한다. 지금은 단순히 여러 스레드에서 공유하는 값에 사용하는 키워드라고 알아두자.

실행 결과

```
14:58:27.520 [     work] 작업 중
14:58:30.525 [     work] 작업 중
14:58:31.510 [     main] 작업 중단 지시 runFlag=false
14:58:33.532 [     work] 자원 정리
14:58:33.533 [     work] 작업 종료
```



- `work` 스레드는 `runFlag`가 `true`인 동안 계속 실행된다.



- 프로그램 시작 후 4초 뒤에 `main` 스레드는 `runFlag`를 `false`로 변경한다.

- `work` 스레드는 `while(runFlag)`에서 `runFlag`의 조건이 `false`로 변한 것을 확인하고, while문을 빠져나가면서 작업을 종료한다.

문제점

실행을 해보면 알겠지만 `main` 스레드가 `runFlag=false`를 통해 작업 중단을 지시해도, `work` 스레드가 즉각 반응하지 않는다. 로그를 보면 작업 중단 지시 2초 정도 이후에 자원을 정리하고 작업을 종료한다.

```
14:58:27.520 [     work] 작업 중
14:58:30.525 [     work] 작업 중
14:58:31.510 [     main] 작업 중단 지시 runFlag=false
14:58:33.532 [     work] 자원 정리 //2초 정도 경과후 실행
14:58:33.533 [     work] 작업 종료
```

이 방식의 가장 큰 문제는 다음 코드의 `sleep()`에 있다.

```
while (runFlag) {
    log("작업 중");
    sleep(3000);
}
```

- `main` 스레드가 `runFlag`를 `false`로 변경해도, `work` 스레드는 `sleep(3000)`을 통해 3초간 잠들어 있다. 3초간의 잠이 깬 다음에 `while(runFlag)` 코드를 실행해야, `runFlag`를 확인하고 작업을 중단할 수 있다.
- 참고로 `runFlag`를 변경한 후 2초라는 시간이 지난 이후에 작업이 종료되는 이유는 `work` 스레드가 3초에 한 번씩 깨어나서 `runFlag`를 확인하는데, `main` 스레드가 4초에 `runFlag`를 변경했기 때문이다.
  - `work` 스레드 입장에서 보면 두 번째 `sleep()`에 들어가고 1초 후 `main` 스레드가 `runFlag`를 변경한다. 3초간 `sleep()`이므로 아직 2초가 더 있어야 깨어난다.

어떻게 하면 `sleep()`처럼 스레드가 대기하는 상태에서 스레드를 깨우고, 작업도 빨리 종료할 수 있을까?

인터럽트 - 시작2

예를 들어서, 특정 스레드가 `Thread.sleep()`을 통해 쉬고 있는데, 처리해야 하는 작업이 들어와서 해당 스레드를 급하게 깨워야 할 수 있다. 또는 `sleep()`으로 쉬고 있는 스레드에게 더는 일이 없으니, 작업 종료를 지시할 수도 있다.

인터럽트를 사용하면, `WAITING`, `TIMED_WAITING` 같은 대기 상태의 스레드를 직접 깨워서, 작동하는 `RUNNABLE` 상태로 만들 수 있다.
앞서 작성한 예제의 작업 중단 지시를 인터럽트를 통해 처리해보자.

```java
package thread.control.interrupt;

import static util.MyLogger.log;
import static util.ThreadUtils.sleep;

public class ThreadStopMainV2 {

    public static void main(String[] args) {
        MyTask task = new MyTask();
        Thread thread = new Thread(task, "work");
        thread.start();

        sleep(4000);
        log("작업 중단 지시 thread.interrupt()");
        thread.interrupt();
        log("work 스레드 인터럽트 상태1 = " + thread.isInterrupted());
    }

    static class MyTask implements Runnable {

        @Override
        public void run() {
            try {
                while (true) {
                    log("작업 중");
                    Thread.sleep(3000);
                }
            } catch (InterruptedException e) {
                log("work 스레드 인터럽트 상태2 = " +
Thread.currentThread().isInterrupted());
                log("interrupt message=" + e.getMessage());
                log("state=" + Thread.currentThread().getState());
            }
            log("자원 정리");
            log("작업 종료");
        }
    }
}
```

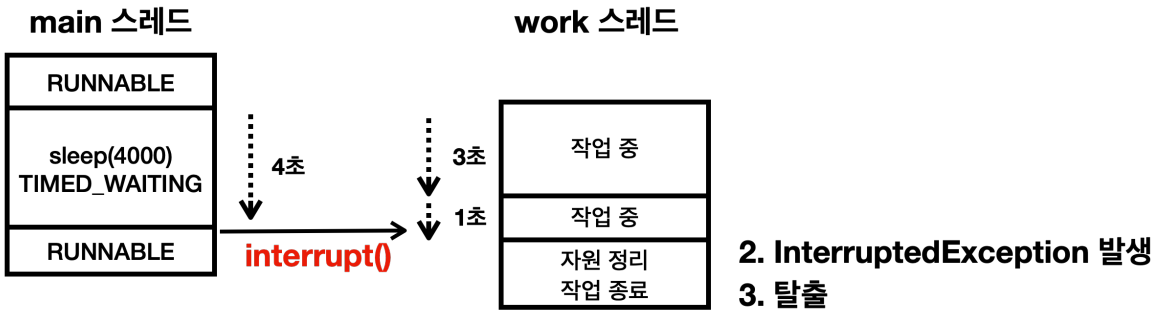
- 예제의 `run()`에서는 인터럽트를 이해하기 위해, 직접 만든 `sleep()` 대신에 `Thread.sleep()`를 사용하고, `try ~ catch`도 사용하자.
- 특정 스레드의 인스턴스에 `interrupt()` 메서드를 호출하면, 해당 스레드에 인터럽트가 발생한다.
- 인터럽트가 발생하면 해당 스레드에 `InterruptedException`이 발생한다.

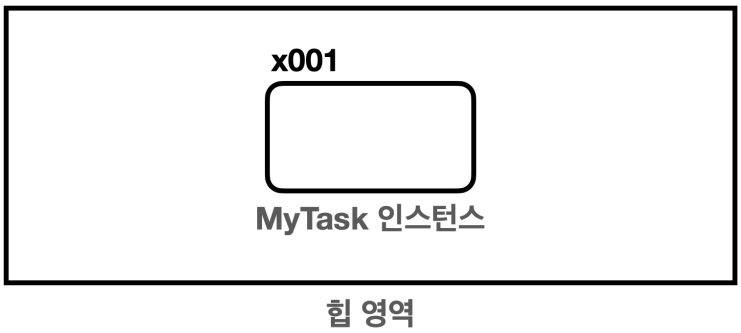
    - 이때 인터럽트를 받은 스레드는 대기 상태에서 깨어나 `RUNNABLE` 상태가 되고, 코드를 정상 수행한다.
    - 이때 `InterruptedException`을 `catch`로 잡아서 정상 흐름으로 변경하면 된다.
- 참고로 `interrupt()`를 호출했다고 해서 즉각 `InterruptedException`이 발생하는 것은 아니다. 오직 `sleep()`처럼 `InterruptedException`을 던지는 메서드를 호출 하거나 또는 호출 중일 때 예외가 발생한다.
    - 예를 들어서 위 코드에서 `while(true)`, `log("작업 중")`에서는 `InterruptedException`이 발생하지 않는다.
    - `Thread.sleep()`처럼 `InterruptedException`을 던지는 메서드를 호출하거나 또는 호출하며 대기중일 때 예외가 발생한다.

실행 결과

```
18:10:40.024 [     work] 작업 중
18:10:43.026 [     work] 작업 중
18:10:44.011 [     main] 작업 중단 지시 thread.interrupt()
18:10:44.021 [     main] work 스레드 인터럽트 상태1 = true
18:10:44.021 [     work] work 스레드 인터럽트 상태2 = false
18:10:44.022 [     work] interrupt message=sleep interrupted
18:10:44.022 [     work] state=RUNNABLE
18:10:44.022 [     work] 자원 정리
18:10:44.023 [     work] 작업 종료
```

- 일부 로그를 보기 쉽게 조정했다.
- `thread.interrupt()`를 통해 작업 중단을 지시를 하고, 거의 즉각적으로 인터럽트가 발생한 것을 확인할 수 있다.
- 이때 `work` 스레드는 `TIMED_WAITNG` ➜ `RUNNABLE` 상태로 변경되면서 `InterruptedException` 예외가 발생한다.
    - 참고로 스레드가 `RUNNABLE` 상태여야 `catch`의 예외 코드도 실행될 수 있다.
- 실행 결과를 보면 `work` 스레드가 `catch` 블럭 안에서 `RUNNABLE` 상태로 바뀐 것을 확인할 수 있다.





- `main` 스레드가 4초 뒤에 `work` 스레드에 `interrupt()`를 건다.
- `work` 스레드는 인터럽트 상태(`true`)가 된다.
- 스레드가 인터럽트 상태일 때는, `sleep()` 처럼 `InterruptedException`이 발생하는 메서드를 호출하거나 또는 이미 호출하고 대기 중이라면 `InterruptedException`이 발생한다.
- 이때 2가지 일이 발생한다.
  - `work` 스레드는 `TIMED_WAITING` 상태에서 `RUNNABLE` 상태로 변경되고, `InterruptedException` 예외를 처리하면서 반복문을 탈출한다.
  - `work` 스레드는 인터럽트 상태가 되었고, 인터럽트 상태이기 때문에 인터럽트 예외가 발생한다.
  - 인터럽트 상태에서 인터럽트 예외가 발생하면 `work` 스레드는 다시 작동하는 상태가 된다. 따라서 `work` 스레드의 인터럽트 상태는 종료된다.
    - `work` 스레드의 인터럽트 상태는 `false`로 변경된다.

주요 로그

```
10:14:49.409 [     main] work 스레드 인터럽트 상태1 = true //여기서 인터럽트 발생
10:14:49.410 [     work] work 스레드 인터럽트 상태2 = false
10:14:49.414 [     work] state=RUNNABLE
```

- 인터럽트가 적용되고, 인터럽트 예외가 발생하면, 해당 스레드는 실행 가능 상태가 되고, 인터럽트 발생 상태도 정상으로 돌아온다.
- 인터럽트를 사용하면 대기중인 스레드를 바로 깨워서 실행 가능한 상태로 바꿀 수 있다. 덕분에 단순히 `runFlag`를 사용하는 이전 방식보다 반응성이 좋아진 것을 확인할 수 있다.

인터럽트 - 시작3

그런데 앞선 코드에서 한가지 아쉬운 부분이 있다.

```
while (true) {  //인터럽트 체크 안함
    log("작업 중");
    Thread.sleep(3000); //여기서만 인터럽트 발생
}
```

여기서 `while(true)` 부분은 체크를 하지 않는다는 점이다. 인터럽트가 발생해도 이 부분은 항상 `true`이기 때문에 다음 코드로 넘어간다. 그리고 `sleep()`을 호출하고 나서야 인터럽트가 발생하는 것이다.

다음과 같이 인터럽트의 상태를 확인하면, 더 빨리 반응할 수 있을 것이다.

```
while (인터럽트_상태_확인) {  //여기서도 인터럽트 상태 체크
    log("작업 중");
    Thread.sleep(3000); //인터럽트 발생
}
```

이 코드와 같이 인터럽트의 상태를 확인하면 while문을 체크하는 부분에서 더 빠르게 while문을 빠져나갈 수 있다.
물론 이 예제의 경우 코드가 단순해서 실질적인 차이는 매우 작다.

추가로 인터럽트의 상태를 직접 확인하면, 다음과 같이 인터럽트를 발생 시키는 `sleep()`과 같은 코드가 없어도 인터럽트 상태를 직접 확인하기 때문에 while문을 빠져나갈 수 있다.

```
while (인터럽트_상태_확인) {  //여기서도 체크
    log("작업 중");
}
```

while문에서 인터럽트의 상태를 직접 확인하도록 코드를 변경해보자.
추가로 예제를 단순화하고 더 직접적인 이해를 돕기 위해 `run()`의 반복문에서 `sleep()` 코드도 함께 제거하자.

```
package thread.control.interrupt;

import static util.MyLogger.log;
import static util.ThreadUtils.sleep;

public class ThreadStopMainV3 {

    public static void main(String[] args) {
        MyTask task = new MyTask();
```

```
        Thread thread = new Thread(task, "work");
        thread.start();

        sleep(100); // 시간을 줄임
        log("작업 중단 지시 - thread.interrupt()");
        thread.interrupt();
        log("work 스레드 인터럽트 상태1 = " + thread.isInterrupted());
    }

    static class MyTask implements Runnable {

        @Override
        public void run() {
            while (!Thread.currentThread().isInterrupted()) { // 인터럽트 상태 변경X
                log("작업 중");
            }
            log("work 스레드 인터럽트 상태2 = " +
Thread.currentThread().isInterrupted());

            try {
                log("자원 정리 시도");
                Thread.sleep(1000);
                log("자원 정리 완료");
            } catch (InterruptedException e) {
                log("자원 정리 실패 - 자원 정리 중 인터럽트 발생");
                log("work 스레드 인터럽트 상태3 = " +
Thread.currentThread().isInterrupted());
            }
            log("작업 종료");
        }
    }
}
```

- `Thread.currentThread()`로 이 코드를 실행하는 스레드를 조회할 수 있다.
- `isInterrupted()`를 사용하면 스레드가 인터럽트 상태인지 확인할 수 있다.

실행 결과

```
...
18:32:05.247 [     work] 작업 중
18:32:05.247 [     work] 작업 중
18:32:05.247 [     main] 작업 중단 지시 - thread.interrupt()
18:32:05.247 [     work] 작업 중
18:32:05.250 [     main] work 스레드 인터럽트 상태1 = true
```

```
18:32:05.250 [     work] work 스레드 인터럽트 상태2 = true
18:32:05.251 [     work] 자원 정리 시도
18:32:05.251 [     work] 자원 정리 실패 - 자원 정리 중 인터럽트 발생
18:32:05.251 [     work] work 스레드 인터럽트 상태3 = false
18:32:05.251 [     work] 작업 종료
```

- 반복문으로 작업 중을 계속 출력하기 때문에 상당히 많은 작업 중 로그가 출력된다.
- 실행 결과는 이해하기 쉽게 조정했다.

주요 실행 순서

- `main` 스레드는 `interrupt()` 메서드를 사용해서, `work` 스레드에 인터럽트를 건다.
- `work` 스레드는 인터럽트 상태이다. `isInterrupted()=true`가 된다.
- 이때 다음과 같이 while 조건이 `false`가 되면서 while문을 탈출한다.
  - `while (!Thread.currentThread().isInterrupted())`
  - `while (!true)`
  - `while (false)`

여기까지 보면 아무런 문제가 없어 보인다. 하지만 이 코드에는 심각한 문제가 있다.
바로 `work` 스레드의 인터럽트 상태가 `true`로 계속 유지된다는 점이다.
앞서 인터럽트 예외가 터진 경우 스레드의 인터럽트 상태는 `false`가 된다.
반면에 `isInterrupted()` 메서드는 인터럽트의 상태를 변경하지 않는다. 단순히 인터럽트의 상태를 확인만 한다.

`work` 스레드는 이후에 자원을 정리하는 코드를 실행하는데, 이때도 인터럽트의 상태는 계속 `true`로 유지된다. 이때 만약 인터럽트가 발생하는 `sleep()`과 같은 코드를 수행한다면, 해당 코드에서 인터럽트 예외가 발생하게 된다.
이것은 우리가 기대한 결과가 아니다! 우리가 기대하는 것은 `while()`문을 탈출하기 위해 딱 한 번만 인터럽트를 사용하는 것이지, 다른 곳에서도 계속해서 인터럽트가 발생하는 것이 아니다.

결과적으로 자원 정리를 하는 도중에 인터럽트가 발생해서, 자원 정리에 실패한다.
자바에서 인터럽트 예외가 한 번 발생하면, 스레드의 인터럽트 상태를 다시 정상(`false`)으로 돌리는 것은 이런 이유 때문이다.
스레드의 인터럽트 상태를 정상으로 돌리지 않으면 이후에도 계속 인터럽트가 발생하게 된다.
인터럽트의 목적을 달성하면 인터럽트 상태를 다시 정상으로 돌려두어야 한다.

참고로 이 예제에서 자원 정리에 실패할 때 인터럽트 예외가 발생하면서 인터럽트의 상태가 정상(`false`)으로 돌아온다.

```
work 스레드 인터럽트 상태3 = false
```

그럼 우리는 어떻게 해야할까?

`while(인터럽트_상태_확인)` 같은 곳에서 인터럽트의 상태를 확인한 다음에, 만약 인터럽트 상태(`true`)라면 인터럽트 상태를 다시 정상(`false`)으로 돌려두면 된다.

인터럽트 - 시작4

Thread.interrupted()

스레드의 인터럽트 상태를 단순히 확인만 하는 용도라면 `isInterrupted()`를 사용하면 된다.

하지만 직접 체크해서 사용할 때는 `Thread.interrupted()`를 사용해야 한다.

이 메서드는 다음과 같이 작동한다.

- 스레드가 인터럽트 상태라면 `true`를 반환하고, 해당 스레드의 인터럽트 상태를 `false`로 변경한다.
- 스레드가 인터럽트 상태가 아니라면 `false`를 반환하고, 해당 스레드의 인터럽트 상태를 변경하지 않는다.

```
package thread.control.interrupt;

import static util.MyLogger.log;
import static util.ThreadUtils.sleep;

public class ThreadStopMainV4 {

    public static void main(String[] args) {
        MyTask task = new MyTask();
        Thread thread = new Thread(task, "work");
        thread.start();

        sleep(100); //시간을 줄임
        log("작업 중단 지시 - thread.interrupt()");
        thread.interrupt();
        log("work 스레드 인터럽트 상태1 = " + thread.isInterrupted());
    }

    static class MyTask implements Runnable {

        @Override
        public void run() {
            while (!Thread.interrupted()) { //인터럽트 상태 변경O
                log("작업 중");
            }
            log("work 스레드 인터럽트 상태2 = " +
```

```
Thread.currentThread().isInterrupted());

            try {
                log("자원 정리 시도");
                Thread.sleep(1000);
                log("자원 정리 완료");
            } catch (InterruptedException e) {
                log("자원 정리 실패 - 자원 정리 중 인터럽트 발생");
                log("work 스레드 인터럽트 상태3 = " +
Thread.currentThread().isInterrupted());
            }
            log("작업 종료");
        }
    }
}
```

실행 결과

```
...
18:42:18.214 [     work] 작업 중
18:42:18.214 [     work] 작업 중
18:42:18.214 [     main] 작업 중단 지시 - thread.interrupt()
18:42:18.214 [     work] 작업 중
18:42:18.219 [     main] work 스레드 인터럽트 상태1 = true
18:42:18.219 [     work] work 스레드 인터럽트 상태2 = false
18:42:18.219 [     work] 자원 정리 시도
18:42:19.221 [     work] 자원 정리 완료
18:42:19.222 [     work] 작업 종료
```

주요 코드

```
while (!Thread.interrupted()) { //인터럽트 상태 변경O
    log("작업 중");
}
```

주요 실행 순서

- `main` 스레드는 `interrupt()` 메서드를 사용해서, `work` 스레드에 인터럽트를 건다.
- `work` 스레드는 인터럽트 상태이다. `Thread.interrupted()` 의 결과는 `true` 가 된다.
  - `Thread.interrupted()` 는 이때 `work` 스레드의 인터럽트 상태를 정상(`false`)으로 변경한다.
- 이때 다음과 같이 while 조건이 `false` 가 되면서 while문을 탈출한다.
  - `while (!Thread.interrupted())`
  - `while (!true)`

- `while (false)`

`Thread.interrupted()`를 호출했을 때 스레드가 인터럽트 상태(`true`)라면, `true`를 반환하고, 해당 스레드의 인터럽트 상태를 `false`로 변경한다.

결과적으로 while문을 탈출하는 시점에, 스레드의 인터럽트 상태도 `false`로 변경된다.

`work` 스레드는 이후에 자원을 정리하는 코드를 실행하는데, 이때 인터럽트의 상태는 `false` 이므로 인터럽트가 발생하는 `sleep()`과 같은 코드를 수행해도 인터럽트가 발생하지 않는다. 이후에 자원을 정상적으로 잘 정리하는 것을 확인할 수 있다.

자바는 인터럽트 예외가 한 번 발생하면, 스레드의 인터럽트 상태를 다시 정상(`false`)으로 돌린다.
스레드의 인터럽트 상태를 정상으로 돌리지 않으면 이후에도 계속 인터럽트가 발생하게 된다.
인터럽트의 목적을 달성하면 인터럽트 상태를 다시 정상으로 돌려두어야 한다.

인터럽트의 상태를 직접 체크해서 사용하는 경우 `Thread.interrupted()`를 사용하면 이런 부분이 해결된다. 참고로 `isInterrupted()`는 특정 스레드의 상태를 변경하지 않고 확인할 때 사용한다.

물론 꼭 이것만이 정답은 아니다. 예를 들어 너무 긴급한 상황이어서 자원 정리도 하지 않고, 최대한 빨리 스레드를 종료해야 한다면 해당 스레드를 다시 인터럽트 상태로 변경하는 것도 방법이다.

프린터 예제1 - 시작

이번에는 인터럽트를 실제 어떤 식으로 활용할 수 있는지 조금 더 실용적인 예제를 만들어보자.
사용자의 입력을 프린터에 출력하는 간단한 예제를 만들어보자.
여기서는 크게 사용자의 입력을 받는 `main` 스레드와 사용자의 입력을 출력하는 `printer` 스레드로 나누어진다.

```
package thread.control.printer;

import java.util.Queue;
import java.util.Scanner;
import java.util.concurrent.ConcurrentLinkedQueue;

import static util.MyLogger.log;
import static util.ThreadUtils.sleep;
```

```
public class MyPrinterV1 {

    public static void main(String[] args) throws InterruptedException {
        Printer printer = new Printer();
        Thread printerThread = new Thread(printer, "printer");
        printerThread.start();

        Scanner userInput = new Scanner(System.in);
        while (true) {
            log("프린터할 문서를 입력하세요. 종료 (q): ");
            String input = userInput.nextLine();
            if (input.equals("q")) {
                printer.work = false;
                break;
            }

            printer.addJob(input);
        }
    }

    static class Printer implements Runnable {
        volatile boolean work = true;
        Queue<String> jobQueue = new ConcurrentLinkedQueue<>();

        @Override
        public void run() {
            while (work) {
                if (jobQueue.isEmpty()) {
                    continue;
                }

                String job = jobQueue.poll();
                log("출력 시작: " + job + ", 대기 문서: " + jobQueue);
                sleep(3000); //출력에 걸리는 시간
                log("출력 완료: " + job);
            }
            log("프린터 종료");
        }

        public void addJob(String input) {
            jobQueue.offer(input);
        }
```

```
    }
}
```

- `volatile`: 여러 스레드가 동시에 접근하는 변수에는 `volatile` 키워드를 붙어주어야 안전하다. 여기서는 `main` 스레드, `printer` 스레드 둘다 `work` 변수에 동시에 접근할 수 있다. `volatile`에 대한 자세한 내용은 뒤에서 설명한다.
- `ConcurrentLinkedQueue`: 여러 스레드가 동시에 접근하는 경우, 컬렉션 프레임워크가 제공하는 일반적인 자료구조를 사용하면 안전하지 않다. 여러 스레드가 동시에 접근하는 경우 동시성을 지원하는 동시성 컬렉션을 사용해야 한다. `Queue`의 경우 `ConcurrentLinkedQueue`를 사용하면 된다. 동시성 컬렉션의 자세한 내용은 뒤에서 설명한다. 여기서는 일반 큐라고 생각하면 된다.

참고: `volatile`, `ConcurrentLinkedQueue`에 대한 자세한 내용은 뒤에서 다룬다. 지금은 여러 스레드에서 접근하는 경우에 이런 것들을 사용하는구나 정도만 알아두면 된다.

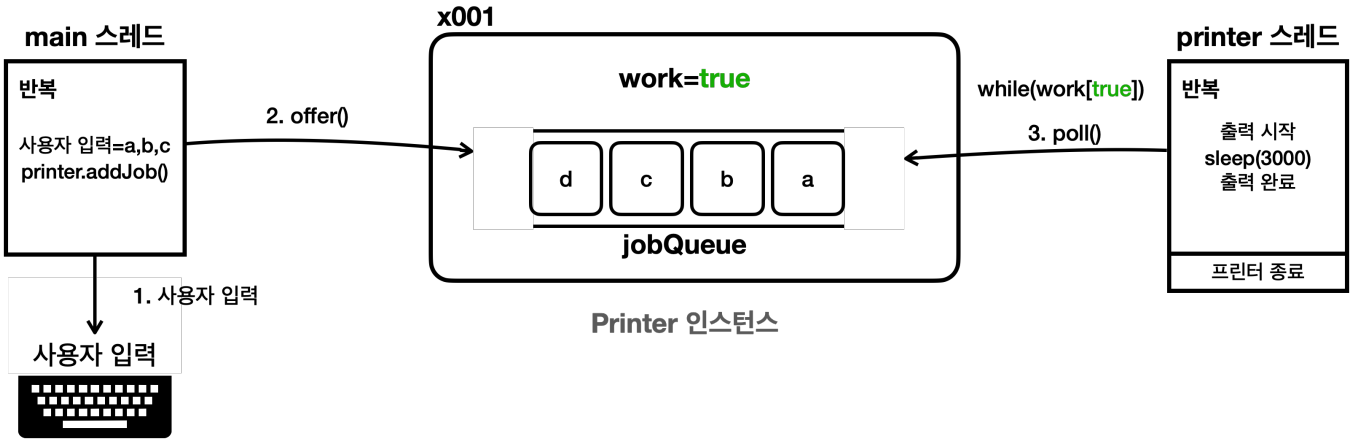
실행 결과

```
16:35:02.771 [     main] 프린터할 문서를 입력하세요. 종료 (q): a
16:35:03.960 [  printer] 출력 시작: a, 대기 문서: []
16:35:03.954 [     main] 프린터할 문서를 입력하세요. 종료 (q): b
16:35:04.587 [     main] 프린터할 문서를 입력하세요. 종료 (q): c
16:35:04.936 [     main] 프린터할 문서를 입력하세요. 종료 (q): d
16:35:05.172 [     main] 프린터할 문서를 입력하세요. 종료 (q):
16:35:06.965 [  printer] 출력 완료: a
16:35:06.966 [  printer] 출력 시작: b, 대기 문서: [c, d]
16:35:09.971 [  printer] 출력 완료: b
16:35:09.972 [  printer] 출력 시작: c, 대기 문서: [d]
16:35:12.974 [  printer] 출력 완료: c
16:35:12.975 [  printer] 출력 시작: d, 대기 문서: []
16:35:15.977 [  printer] 출력 완료: d
q
16:35:18.118 [  printer] 프린터 종료
```

실행 결과는 보기 쉽게 다듬었다.

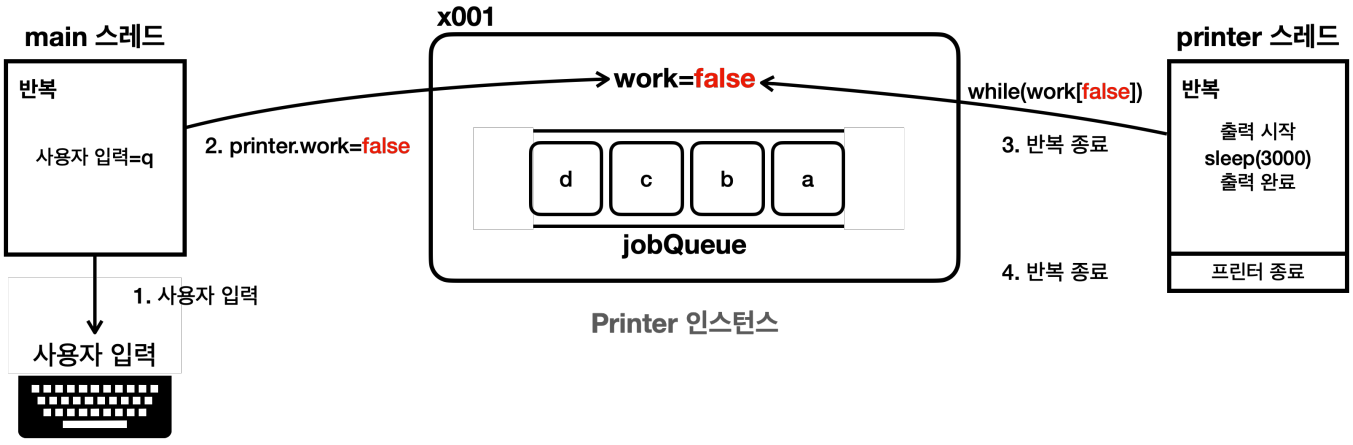
- 입력으로 `a`, `b`, `c`, `d`를 사용했다.

프린터 작동 그림



- `main` 스레드: 사용자의 입력을 받아서 `Printer` 인스턴스의 `jobQueue`에 담는다.
- `printer` 스레드: `jobQueue`가 있는지 확인한다.
  - `jobQueue`에 내용이 있으면 `poll()`을 이용해서 꺼낸 다음에 출력한다.
    - 출력하는데는 약 3초의 시간이 걸린다. 여기서는 `sleep(3000)`를 사용해서 출력 시간을 가상으로 구현했다.
    - 출력을 완료하면 while문을 다시 반복한다.
  - 만약 `jobQueue`가 비었다면 `continue`를 사용해서 다시 while문을 반복한다.
  - 이렇게 해서 `jobQueue`에 출력할 내용이 들어올 때 까지 계속 확인한다.

프린터 종료 그림



- `main` 스레드: 사용자가 `q`를 입력한다. `printer.work`의 값을 `false`로 변경한다.
  - `main` 스레드는 while문을 빠져나가고 `main` 스레드가 종료된다.
- `printer` 스레드: while문에서 `work`의 값이 `false`인 것을 확인한다.
  - `printer` 스레드는 while문을 빠져나가고, "프린터 종료"를 출력하고, `printer` 스레드는 종료된다.

앞서 살펴보았듯이 이 방식의 문제는 종료(`q`)를 입력했을 때 바로 반응하지 않는다는 점이다. 왜냐하면 `printer` 스레드가 반복문을 빠져나오려면 while문을 체크해야 하는데, `printer` 스레드가 `sleep(3000)`을 통해 대기 상태에 빠져서 작동하지 않기 때문이다. 따라서 최악의 경우 `q`를 입력하고 3초 이후에 프린터가 종료된다.
이제 인터럽트를 사용해서 반응성이 느린 문제를 해결해 보자.

프린터 예제2 - 인터럽트 도입

앞서 만든 예제에 인터럽트를 도입해보자.

```
package thread.control.printer;

import java.util.Queue;
import java.util.Scanner;
import java.util.concurrent.ConcurrentLinkedQueue;

import static util.MyLogger.log;

public class MyPrinterV2 {

    public static void main(String[] args) throws InterruptedException {
        Printer printer = new Printer();
        Thread printerThread = new Thread(printer, "printer");
        printerThread.start();

        Scanner userInput = new Scanner(System.in);
        while (true) {
            System.out.println("프린터할 문서를 입력하세요. 종료 (q): ");
            String input = userInput.nextLine();
            if (input.equals("q")) {
                printer.work = false;
                printerThread.interrupt();
                break;
            }
            printer.addJob(input);
        }
    }

    static class Printer implements Runnable {
        volatile boolean work = true;
        Queue<String> jobQueue = new ConcurrentLinkedQueue<>();

        @Override
        public void run() {

            while (work) {
```

```
                if (jobQueue.isEmpty()) {
                    continue;
                }

                try {
                    String job = jobQueue.poll();
                    log("출력 시작: " + job + ", 대기 문서: " + jobQueue);
                    Thread.sleep(3000); //출력에 걸리는 시간
                    log("출력 완료: " + job);
                } catch (InterruptedException e) {
                    log("인터럽트!");
                    break;
                }
            }
            log("프린터 종료");
        }

        public void addJob(String input) {
            jobQueue.offer(input);
        }

    }
}
```

실행 결과

```
프린터할 문서를 입력하세요. 종료 (q): a
17:13:16.994 [  printer] 출력 시작: a, 대기 문서: []
프린터할 문서를 입력하세요. 종료 (q): b
프린터할 문서를 입력하세요. 종료 (q): c
프린터할 문서를 입력하세요. 종료 (q): d
프린터할 문서를 입력하세요. 종료 (q):
17:13:19.999 [  printer] 출력 완료: a
17:13:20.001 [  printer] 출력 시작: b, 대기 문서: [c, d]
q
17:13:20.920 [  printer] 인터럽트!
17:13:20.921 [  printer] 프린터 종료
```

종료(`q`)를 입력하면 즉시 종료되는 것을 확인할 수 있다. 따라서 반응성이 매우 좋아진다.

종료시 `main` 스레드는 `work` 변수도 `false`로 변경하고, `printer` 스레드에 인터럽트도 함께 호출한다.

```
if (input.equals("q")) {
    printer.work = false;
```

```
    printerThread.interrupt();
    break;
}
```

이렇게 둘 다 함께 적용하면, `printer` 스레드가 `sleep()`을 호출한 상태는 물론이고, `while (work)` 코드가 실행되는 부분에서도 빠져나올 수 있어서 반응성이 더 좋아진다.

- `interrupt()` : `sleep()` 상태에서 빠져나온다.
- `work=false` : while문을 체크하는 곳에서 빠져나온다.

프린터 예제3 - 인터럽트 코드 개선

이번에는 앞서 작성한 예제의 코드를 개선해보자.

인터럽트의 상태를 직접 확인하면, `work` 변수를 제거할 수 있다.

```
package thread.control.printer;

import java.util.Queue;
import java.util.Scanner;
import java.util.concurrent.ConcurrentLinkedQueue;

import static util.MyLogger.log;

public class MyPrinterV3 {

    public static void main(String[] args) throws InterruptedException {
        Printer printer = new Printer();
        Thread printerThread = new Thread(printer, "printer");
        printerThread.start();

        Scanner userInput = new Scanner(System.in);
        while (true) {
            System.out.println("프린터할 문서를 입력하세요. 종료 (q): ");
            String input = userInput.nextLine();
            if (input.equals("q")) {
                printerThread.interrupt();
                break;
            }
            printer.addJob(input);
        }
```

```
    }

    static class Printer implements Runnable {
        Queue<String> jobQueue = new ConcurrentLinkedQueue<>();

        @Override
        public void run() {

            while (!Thread.interrupted()) {
                if (jobQueue.isEmpty()) {
                    continue;
                }

                try {
                    String job = jobQueue.poll();
                    log("출력 시작: " + job + ", 대기 문서: " + jobQueue);
                    Thread.sleep(3000); //출력에 걸리는 시간
                    log("출력 완료: " + job);
                } catch (InterruptedException e) {
                    log("인터럽트!");
                    break;
                }
            }
            log("프린터 종료");
        }

        public void addJob(String input) {
            jobQueue.offer(input);
        }

    }
}
```

- `Thread.interrupted()` 메서드를 사용하면 해당 스레드가 인터럽트 상태인지 아닌지 확인할 수 있다.
- 따라서 while에서 체크하던 `work` 변수를 제거할 수 있다.
  - `work` 변수로 확인하는 대신에 해당 스레드의 인터럽트 상태만 확인하면 된다.

`printer` 스레드는 자신이 인터럽트 상태인지 다음 코드로 확인하면 된다.

```
while (!Thread.interrupted())
```

`main` 스레드도 `work` 변수의 사용을 제거하고, 인터럽트만 걸어주면 된다.

```
if (input.equals("q")) {
```

```
    printerThread.interrupt();
    break;
}
```

실행 결과

```
프린터할 문서를 입력하세요. 종료 (q): a
프린터할 문서를 입력하세요. 종료 (q): b
17:29:16.919 [  printer] 출력 시작: a, 대기 문서: []
프린터할 문서를 입력하세요. 종료 (q): c
프린터할 문서를 입력하세요. 종료 (q): d
프린터할 문서를 입력하세요. 종료 (q):
17:29:19.923 [  printer] 출력 완료: a
17:29:19.925 [  printer] 출력 시작: b, 대기 문서: [c, d]
q
17:29:21.367 [  printer] 인터럽트!
17:29:21.368 [  printer] 프린터 종료
```

실행 결과는 기존과 같다.

yield - 양보하기

어떤 스레드를 얼마나 실행할지는 운영체제가 스케줄링을 통해 결정한다. 그런데 특정 스레드가 크게 바쁘지 않은 상황이어서 다른 스레드에 CPU 실행 기회를 양보하고 싶을 수 있다. 이렇게 양보하면 스케줄링 큐에 대기 중인 다른 스레드가 CPU 실행 기회를 더 빨리 얻을 수 있다.

예제를 통해 알아보자.

```
package thread.control.yield;

import static util.ThreadUtils.sleep;

public class YieldMain {

    static final int THREAD_COUNT = 1000;

    public static void main(String[] args) {
        for (int i = 0; i < THREAD_COUNT; i++) {
            Thread thread = new Thread(new MyRunnable());
            thread.start();
```

```
        }
    }

    static class MyRunnable implements Runnable {
        public void run() {
            for (int i = 0; i < 10; i++) {
                System.out.println(Thread.currentThread().getName() + " - " +
i);
                // 1. empty
                //sleep(1); // 2. sleep
                //Thread.yield(); // 3. yield
            }
        }
    }
}
```

- 1000개의 스레드를 실행한다.
- 각 스레드가 실행하는 로직은 아주 단순하다. 스레드당 0~9까지 출력하면 끝난다.
- `run()` 에 있는 1, 2, 3 주석을 변경하면서 실행해보자.

여기서는 3가지 방식을 사용한다.

1. `Empty` : `sleep(1)` , `yield()` 없이 호출한다. 운영체제의 스레드 스케줄링을 따른다.
2. `sleep(1)` : 특정 스레드를 잠시 쉬게 한다.
3. `yield()` : `yield()` 를 사용해서 다른 스레드에 실행을 양보한다.

Empty

실행 결과 - 1. Empty

```
Thread-998 - 2
Thread-998 - 3
Thread-998 - 4
Thread-998 - 5
Thread-998 - 6
Thread-998 - 7
Thread-998 - 8
Thread-998 - 9
Thread-999 - 0
Thread-999 - 1
Thread-999 - 2
Thread-999 - 3
Thread-999 - 4
Thread-999 - 5
```

```
Thread-999 - 6
Thread-999 - 7
Thread-999 - 8
Thread-999 - 9
```

- 특정 스레드가 쭉~ 수행된 다음에 다른 스레드가 수행되는 것을 확인할 수 있다.
- 참고로 실행 환경에 따라 결과는 달라질 수 있다. 다른 예시보다 상대적으로 하나의 스레드가 쭉~ 연달아 실행되다가 다른 스레드로 넘어간다.
- 이 부분은 운영체제의 스케줄링 정책과 환경에 따라 다르지만 대략 0.01초(10ms)정도 하나의 스레드가 실행되고, 다른 스레드로 넘어간다.

sleep()

실행 결과 - 2. sleep()

```
Thread-626 - 9
Thread-997 - 9
Thread-993 - 9
Thread-949 - 7
Thread-645 - 9
Thread-787 - 9
Thread-851 - 9
Thread-949 - 8
Thread-949 - 9
```

- `sleep(1)`을 사용해서 스레드의 상태를 1밀리초 동안 아주 잠깐 `RUNNABLE` ➜ `TIMED_WAITING`으로 변경한다. 이렇게 되면 스레드는 CPU 자원을 사용하지 않고, 실행 스케줄링에서 잠시 제외된다. 1 밀리초의 대기 시간 이후 다시 `TIMED_WAITING` ➜ `RUNNABLE` 상태가 되면서 실행 스케줄링에 포함된다.
- 결과적으로 `TIMED_WAITING` 상태가 되면서 다른 스레드에 실행을 양보하게 된다. 그리고 스캐줄링 큐에 대기중인 다른 스레드가 CPU의 실행 기회를 빨리 얻을 수 있다.

하지만 이 방식은 `RUNNABLE` ➜ `TIMED_WAITING` ➜ `RUNNABLE`로 변경되는 복잡한 과정을 거치고, 또 특정 시간만큼 스레드가 실행되지 않는 단점이 있다.
예를 들어서 양보할 스레드가 없다면, 차라리 나의 스레드를 더 실행하는 것이 나은 선택일 수 있다. 이 방법은 나머지 스레드가 모두 대기 상태로 쉬고 있어도 내 스레드까지 잠깐 실행되지 않는 것이다. 쉽게 이야기해서 양보할 사람이 없는데 혼자서 양보한 이상한 상황이 될 수 있다.

yield()

실행 결과3 - yield

```
Thread-805 - 9
```

```
Thread-321 - 9
Thread-880 - 8
Thread-900 - 8
Thread-900 - 9
Thread-570 - 9
Thread-959 - 9
Thread-818 - 9
Thread-880 - 9
```

자바의 스레드가 `RUNNABLE` 상태일 때, 운영체제의 스케줄링은 다음과 같은 상태들을 가질 수 있다.

- **실행 상태(Running):** 스레드가 CPU에서 실제로 실행 중이다.
- **실행 대기 상태(Ready):** 스레드가 실행될 준비가 되었지만, CPU가 바빠서 스케줄링 큐에서 대기 중이다.

운영체제는 실행 상태의 스레드들을 잠깐만 실행하고 실행 대기 상태로 만든다. 그리고 실행 대기 상태의 스레드들을 잠깐만 실행 상태로 변경해서 실행한다. 이 과정을 계속 반복한다. 참고로 자바에서는 두 상태를 구분할 수는 없다.

yield()의 작동

- `Thread.yield()` 메서드는 현재 실행 중인 스레드가 자발적으로 CPU를 양보하여 다른 스레드가 실행될 수 있도록 한다.
- `yield()` 메서드를 호출한 스레드는 `RUNNABLE` 상태를 유지하면서 CPU를 양보한다. 즉, 이 스레드는 다시 스케줄링 큐에 들어가면서 다른 스레드에게 CPU 사용 기회를 넘긴다.

자바에서 `Thread.yield()` 메서드를 호출하면 현재 실행 중인 스레드가 CPU를 양보하도록 힌트를 준다. 이는 스레드가 자신에게 할당된 실행 시간을 포기하고 다른 스레드에게 실행 기회를 주도록 한다. 참고로 `yield()`는 운영체제의 스케줄러에게 단지 힌트를 제공할 뿐, 강제적인 실행 순서를 지정하지 않는다. 그리고 반드시 다른 스레드가 실행되는 것도 아니다.

`yield()`는 `RUNNABLE` 상태를 유지하기 때문에, 쉽게 이야기해서 양보할 사람이 없다면 본인 스레드가 계속 실행될 수 있다.

참고로 최근에는 10코어 이상의 CPU도 많기 때문에 스레드 10개 정도만 만들어서 실행하면, 양보가 크게 의미가 없다. 양보해도 CPU 코어가 남기 때문에 양보하지 않고 계속 수행될 수 있다. CPU 코어 수 이상의 스레드를 만들어야 양보하는 상황을 확인할 수 있다. 그래서 이번 예제에서 1000개의 스레드를 실행한 것이다.

> **참고**: `log()`가 사용하는 기능은 현재 시간도 획득해야 하고, 날짜 포멧도 지정해야 하는 등 복잡하다. 이 사이에 스레드의 컨텍스트 스위칭이 발생하기 쉽다. 이런 이유로 스레드의 실행 순서를 일정하게 출력하기 어렵다. 그래서 여기서는 단순한 `System.out.println()`을 사용했다.

프린터 예제4 - yield 도입

앞서 개발한 프린터 예제를 보면 `yield()`를 적용하기 딱 좋은 곳이 있다.

```
while (!Thread.interrupted()) {
    if (jobQueue.isEmpty()) {
        continue;
    }
    ...
}
```

이 코드를 보면 인터럽트가 발생하기 전까지 계속 인터럽트의 상태를 체크하고 또 `jobQueue`의 상태를 확인한다.
문제는 쉴 틈 없이 CPU에서 이 로직이 계속 반복해서 수행된다는 점이다. 1초에 while문을 수억 번 반복할 수도 있다!
결과적으로 CPU 자원을 많이 사용하게 된다.

현재 작동하는 스레드가 아주 많다고 가정해보자.
인터럽트도 걸리지 않고, `jobQueue`도 비어있는데, 이런 체크 로직에 CPU 자원을 많이 사용하게 되면, 정작 필요한 스레드들의 효율이 상대적으로 떨어질 수 있다.
차라리 그 시간에 다른 스레드들을 더 많이 실행해서 `jobQueue`에 필요한 작업을 빠르게 만들어 넣어주는게 더 효율적일 것이다.
그래서 다음과 같이 `jobQueue`에 작업이 비어있으면 `yield()`를 호출해서, 다른 스레드에 작업을 양보하는게 전체 관점에서 보면 더 효율적이다.

```
while (!Thread.interrupted()) {
    if (jobQueue.isEmpty()) {
        Thread.yield(); // 추가
        continue;
    }
    ...
}
```

`MyPrinterV3` 프린터 예제를 개선해서 `MyPrinterV4`를 만들어보자.
전체 코드는 다음과 같다. 기존 코드에 `yield()`만 추가되었다.

```
package thread.control.printer;

import java.util.Queue;
import java.util.Scanner;
import java.util.concurrent.ConcurrentLinkedQueue;

import static util.MyLogger.log;
```

```
public class MyPrinterV4 {

    public static void main(String[] args) throws InterruptedException {
        Printer printer = new Printer();
        Thread printerThread = new Thread(printer, "printer");
        printerThread.start();

        Scanner userInput = new Scanner(System.in);
        while (true) {
            System.out.println("프린터할 문서를 입력하세요. 종료 (q): ");
            String input = userInput.nextLine();
            if (input.equals("q")) {
                printerThread.interrupt();
                break;
            }
            printer.addJob(input);
        }
    }

    static class Printer implements Runnable {
        Queue<String> jobQueue = new ConcurrentLinkedQueue<>();

        @Override
        public void run() {

            while (!Thread.interrupted()) {
                if (jobQueue.isEmpty()) {
                    Thread.yield(); //추가
                    continue;
                }

                try {
                    String job = jobQueue.poll();
                    log("출력 시작: " + job + ", 대기 문서: " + jobQueue);
                    Thread.sleep(3000); //출력에 걸리는 시간
                    log("출력 완료: " + job);
                } catch (InterruptedException e) {
                    log("인터럽트!");
                    break;
                }
            }
            log("프린터 종료");
```

```
        }

        public void addJob(String input) {
            jobQueue.offer(input);
        }

    }
}
```

정리