쓰레드 제어와 생명 주기2

소유자	종수 김
태그	

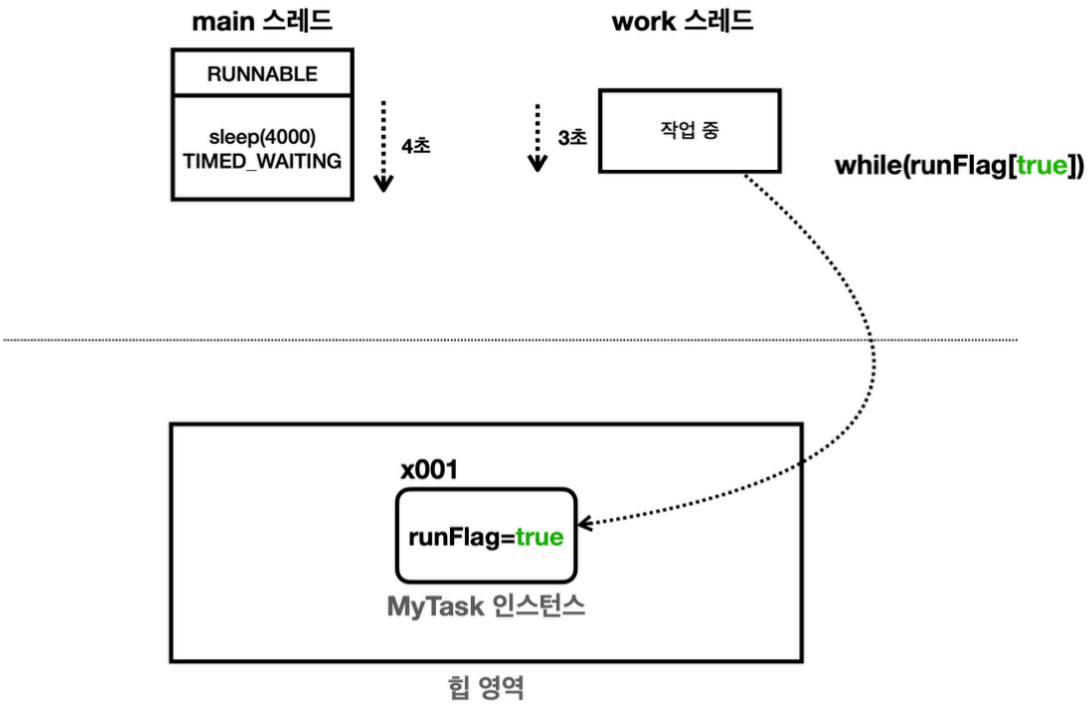
인터럽트 - 시작1

특정 쓰레드의 작업을 중간에 중단하려면 ?

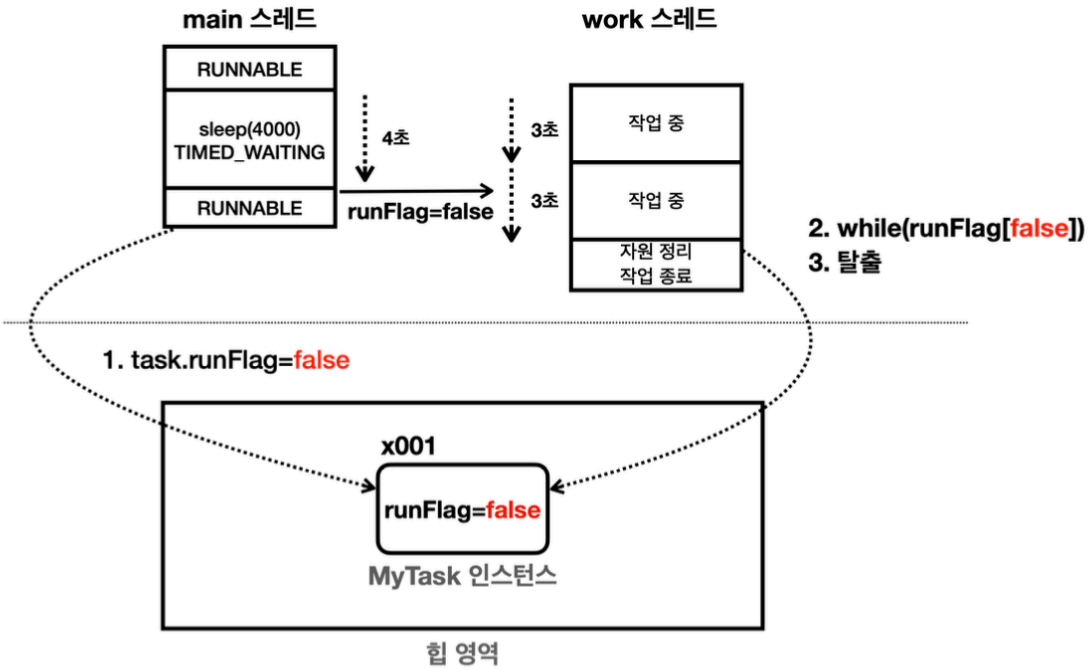
```
package thread.start.control.interrupt;

import static util.MyLoggerThread.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("자원 종료");
```

```
    }
  }
}

20:52:38.583 [    work] 작업 중
20:52:41.590 [    work] 작업 중
20:52:42.568 [    main] 작업 중단 지시 runFlag = false
20:52:44.596 [    work] 자원 정리
20:52:44.597 [    work] 자원 종료
```

- 특정 쓰레드의 작업을 중단 시키는 가장 쉬운 방법은 변수를 사용하는 것.



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



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

- 프로그램 시작 후 4초 뒤 main 쓰레드는 runFlag를 false로 변경
- `main` 스레드가 `runFlag` 를 `false` 로 변경해도, `work` 스레드는 `sleep(3000)` 을 통해 3초간 잠들어 있다. 3초간의 잠이 깬 다음에 `while(runFlag)` 코드를 실행해

야, `runFlag` 를 확인하고 작업을 중단할 수 있다.
참고로 `runFlag` 를 변경한 후 2초라는 시간이 지난 이후에 작업이 종료되는 이유는 `work` 스레드가 3초에 한번씩 깨어나서 `runFlag` 를 확인하는데, `main` 스레드가 4초에 `runFlag` 를 변경했기 때문이다.

- runFlag를 false로 변경한다고 해서 work 쓰레드가 즉각 반응하지 않음.
    - 본인의 작업이 끝난 후 while문의 조건을 확인하여 종료하는 것.

sleep()처럼 쓰레드가 대기하는 상태에서 쓰레드를 깨우고, 작업도 빨리 종료하는 방법은 ?

인터럽트 - 시작2

예를 들어, 특정 쓰레드가 Thread.sleep()을 통해 쉬고 있는데, 처리해야 하는 작업이 들어와서 해당 쓰레드를 급하게 깨워야 할 수 있음, 또는 sleep()으로 쉬고 있는 쓰레드에게 더는 일이 없으니 작업 종료를 지시할 수도 있음.

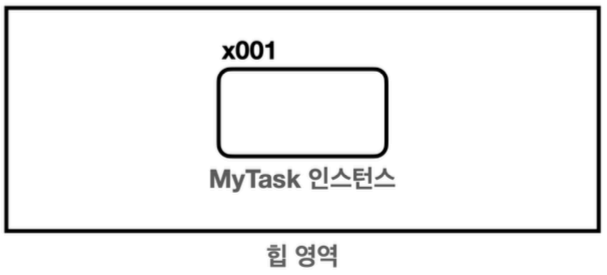
인터럽트를 사용하면, WAITING, TIMED_WAITING를 깨워서 RUNNABLE 상태로 만들 수 있음.

```
package thread.start.control.interrupt;

import static util.MyLoggerThread.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 쓰레드의 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 쓰레드의 인터럽트 상태2 = " + Thread.currentThread().isI
nterrupted());
            log("interrupt message" + e.getMessage());
            log("state = " + Thread.currentThread().getState());
        }

        log("자원 정리");
        log("자원 종료");
    }
  }
}

21:02:32.573 [     work] 작업 중
21:02:35.580 [     work] 작업 중
21:02:36.557 [     main] 작업 중단 지시 thread.interrupt()
21:02:36.570 [     work] work 쓰레드의 인터럽트 상태2 = false
21:02:36.570 [     main] work 쓰레드의 인터럽트 상태1 = true
21:02:36.570 [     work] interrupt messagesleep interrupted
21:02:36.571 [     work] state = RUNNABLE
21:02:36.571 [     work] 자원 정리
21:02:36.571 [     work] 자원 종료
```

- 특정 쓰레드의 인스턴스에 interrupt() 메서드를 호출하면, 해당 쓰레드에 인터럽트가 발생.
- 인터럽트가 발생하면 해당 쓰레드에 InterruptedException이 발생
  - 이 때 인터럽트를 받은 쓰레드는 대기 상태에서 깨어나 RUNNABLE 상태가 되고 코드를 정상 수행.
  - 이 때 InterruptedException을 catch로 잡아서 정상흐름으로 변경하면 됨.
- interrupt()를 호출했다고 즉각 InterruptedException이 발생하는 것은 아님, 오직 sleep()처럼 InterruptedException을 던지는 메서드를 호출하거나, 또는 호출 중일 때 예외가 발생함.

    - 예를 들어 위 코드에서 while(true), log("작업 중") 에서는 InterruptedException이 발생하지 않음.
    - Thread.sleep()처럼 InterruptedException을 던지는 메서드를 호출하거나, 호출하며 대기중일 때 예외가 발생
- 참고로 쓰레드가 RUNNABLE 상태여야 catch 블럭 내부의 예외 코드도 실행될 수 있음.



- main 쓰레드가 4초 뒤에 work 쓰레드에 interrupt()를 건다.
- work 쓰레드는 인터럽트 상태(true)가 됨.
- 쓰레드가 인터럽트 상태일 때는 sleep()처럼 InterruptedException이 발생하는 메서드를 호출하거나, 또는 이미 호출하고 대기중이라면 InterruptedException이 발생.
- 이 때 2가지 일이 발생
    - work 쓰레드는 TIMED_WAITING 상태에서 RUNNABLE 상태로 변경되고, InterruptedException예외를 처리하며 반복문을 탈출.
    - work 쓰레드는 인터럽트 상태가 되었고, 인터럽트 상태이기 때문에 인터럽트 예외가 발생
    - 인터럽트 상태에서 인터럽트 예외가 발생하면 work 쓰레드는 다시 작동하는 상태가 되며, work 쓰레드의 인터럽트 상태는 종료된다.
        - work 쓰레드의 인터럽트 상태는 false로 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.

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.

이를 더 빠르게 체크할 수 있는 방법이 있을까 ?

인터럽트 - 시작3

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

이 코드처럼 인터럽트의 상태를 확인하면 더 반응성이 좋아지지않을까 ?

```
package thread.start.control.interrupt;

import static util.MyLoggerThread.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);
        log("작업 중단 지시 thread.interrupt()");
```

```
        thread.interrupt();
        log("work 쓰레드의 인터럽트 상태1 = " + thread.isInterrupted());
    }

    static class MyTask implements Runnable {

        @Override
        public void run() {
            while (!Thread.currentThread().isInterrupted()) {
                log("작업 중");
            }

            log("work 쓰레드의 인터럽트 상태2 = " + Thread.currentThread().isInte
rrupted());
            log("state = " + Thread.currentThread().getState());

            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().isInt
errupted());
            }
            log("작업 종료");
        }
    }
}
```

주요 실행 순서

- main 쓰레드는 interrupt() 메서드를 사용해서 work 쓰레드에 인터럽트를 검.
- work 쓰레드는 인터럽트 상태임. isInterrupted() = true 가 됨.
- 이 때 다음과 같이 while 조건이 false가 되며, while문을 탈출
  - while(!Thread.currentThread().isInterrupted())
  - while(!true)

- while(false)

‘work 쓰레드의 인터럽트 상태는 계속해서 true로 유지가 됨.’

앞서 인터럽트 예외가 터진 경우 쓰레드의 인터럽트 상태는 false가 됨.

반면 isInterrupted() 메서드는 인터럽트의 상태를 변경하지 않고 단순히 상태만 확인하기 때문에.

work 쓰레드는 이후 자원을 정리하는 코드를 실행하려 하지만, 이 때도 isInterrupted는 true인 것,
만약 자원 정리하는 로직 내에 Thread.sleep()같은 메서드가 있는 경우 해당 코드에서 InterruptedException이 발생하는 것.

결과적으로는 자원 정리를 하는 도중에 인터럽트가 발생해서, 자원 정리에 실패

> 자바에서 인터럽트가 발생하면, 쓰레드의 인터럽트 상태를 다시 정상(false)로 돌리는 것은 이런 이유 때문.
> 쓰레드의 인터럽트 상태를 정상으로 돌리지 않으면 이후에도 계속 인터럽트가 발생하게 됨.
> 인터럽트의 목적을 달성하면 인터럽트 상태를 다시 정상으로 돌려두어야 함.

인터럽트 - 시작 4

Thread.interruped()

쓰레드의 인터럽트 상태를 단순히 확인만 하는 용도라면, isInterruped()를 사용하면 되지만,

직접 체크해서 사용해야 할 때는 Thread.interruped()를 사용해야 함.

- 쓰레드가 인터럽트 상태라면 true를 반환하고, 해당 쓰레드의 인터럽트 상태를 false로 변경
- 쓰레드가 인터럽트 상태가 아니라면, false를 반환하고, 해당 쓰레드의 인터럽트 상태를 변경하지 않음.

```
package thread.start.control.interrupt;

import static util.MyLoggerThread.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);
        log("작업 중단 지시 thread.interrupt()");
        thread.interrupt();
        log("work 쓰레드의 인터럽트 상태1 = " + thread.isInterrupted());
    }

    static class MyTask implements Runnable {

        @Override
        public void run() {
            while (!Thread.interrupted()) { // 인터럽트 상태 변경 O
                log("작업 중");
            }

            log("work 쓰레드의 인터럽트 상태2 = " + Thread.currentThread().isInte
rrupted());
            log("state = " + Thread.currentThread().getState());

            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().isInt
errupted());
            }
            log("작업 종료");
        }
    }
}

21:25:12.603 [     work] 작업 중
21:25:12.603 [     main] 작업 중단 지시 thread.interrupt()
21:25:12.605 [     main] work 쓰레드의 인터럽트 상태1 = true
```

```
21:25:12.605 [     work] work 쓰레드의 인터럽트 상태2 = false
21:25:12.606 [     work] state = RUNNABLE
21:25:12.606 [     work] 자원 정리 시도
21:25:13.611 [     work] 자원 정리 완료
21:25:13.611 [     work] 작업 종료
```

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)`

결과적으로 while문을 탈출하는 시점에 쓰레드의 인터럽트 상태도 false로 변경됨.

프린터 예제1 - 시작

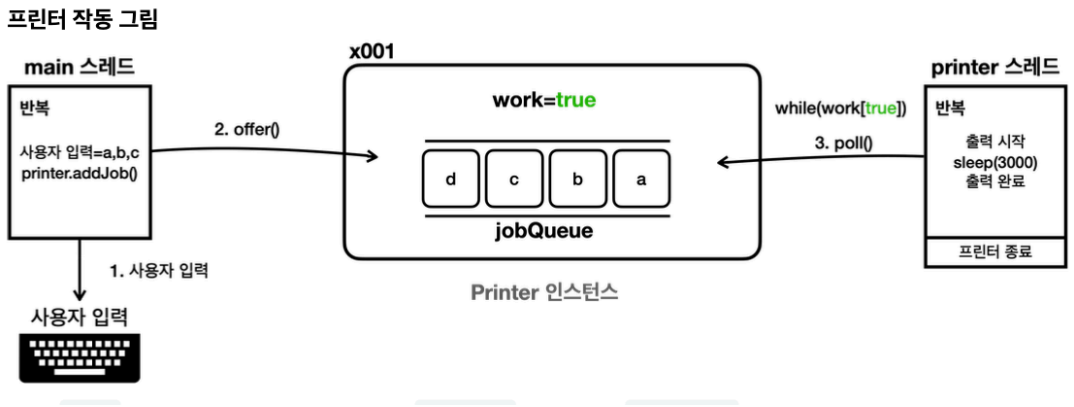
인터럽트를 활용할 수 있는 실용적인 예제

```
package thread.start.control.printer;

import nested.local.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) {
```

```
        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);
                log("출력 완료");
            }

            log("프린터 완료");
        }

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

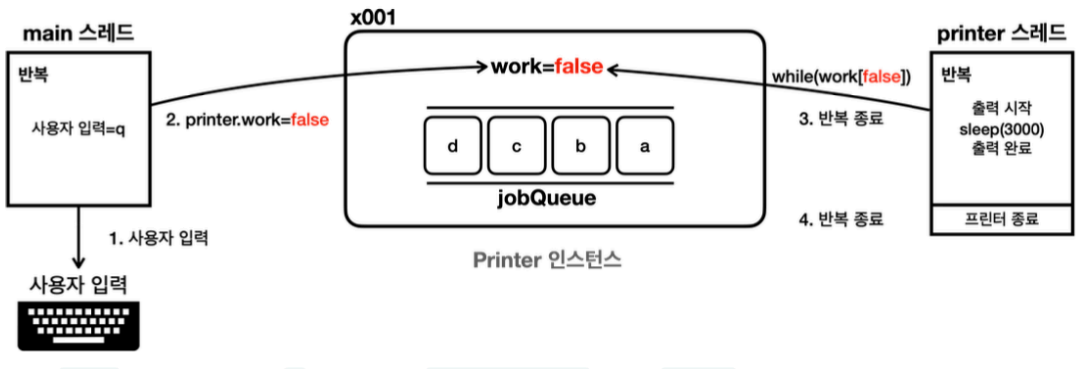
```
      }
    }
}

21:45:10.447 [     main] 프린터할 문서를 입력하세요. 종료 (q):
a
21:45:11.535 [     main] 프린터할 문서를 입력하세요. 종료 (q):
21:45:11.540 [  printer] 출력 시작a, 대기 문서 : []
b
21:45:11.674 [     main] 프린터할 문서를 입력하세요. 종료 (q):
c
21:45:11.804 [     main] 프린터할 문서를 입력하세요. 종료 (q):
d
21:45:12.285 [     main] 프린터할 문서를 입력하세요. 종료 (q):
e
21:45:12.426 [     main] 프린터할 문서를 입력하세요. 종료 (q):
21:45:14.546 [  printer] 출력 완료
21:45:14.548 [  printer] 출력 시작b, 대기 문서 : [c, d, e]
```

프린터 작동



- main 쓰레드 : 사용자의 입력을 받아서 Printer인스턴스의 Queue에 문자열을 담는다.
- printer 쓰레드 : jobQueue가 있는지 확인
  - jobQueue에 내용이 있으면 poll()을 이용해서 꺼낸 다음에 출력
    - 출력하는데 약 3초
    - 출력을 완료하면 while문을 다시 반복

프린터 종료



- main 쓰레드 : 사용자가 q를 입력. printer.work의 값을 false로 변경
  - main 쓰레드는 while문을 빠져나가고 main 쓰레드가 종료
- printer 쓰레드 : while문에서 work의 값이 false인 것을 확인
  - printer 쓰레드는 while문을 빠져나가고 “프린터 종료”를 출력하고 printer 쓰레드 종료

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 상태에빠져서 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 도입

```
package thread.start.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 MyPrinterV2 {
    public static void main(String[] args) {
```

```
        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")) {
                log("작업 중단 지시 thread.interrupt()");
                printerThread.interrupt();
                break;
            }
            printer.addJob(input);
        }
    }

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

        @Override
        public void run() {
            try {
                while (!Thread.interrupted()) {
                    if (jobQueue.isEmpty()) {
                        continue;
                    }

                    String job = jobQueue.poll();
                    log("출력 시작" + job + ", 대기 문서 : " + jobQueue);
                    Thread.sleep(3000);
                    log("출력 완료");

                }
            } catch (InterruptedException e){
                log("interrupted : 프린터 종료");
            }
```

```
            log("프린터 완료");
        }

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

21:55:58.965 [     main] 프린터할 문서를 입력하세요. 종료 (q):
1
21:55:59.361 [     main] 프린터할 문서를 입력하세요. 종료 (q):
21:55:59.366 [  printer] 출력 시작1, 대기 문서 : []
2
21:55:59.497 [     main] 프린터할 문서를 입력하세요. 종료 (q):
3
21:55:59.635 [     main] 프린터할 문서를 입력하세요. 종료 (q):
4
21:55:59.766 [     main] 프린터할 문서를 입력하세요. 종료 (q):
5
21:55:59.953 [     main] 프린터할 문서를 입력하세요. 종료 (q):
21:56:02.371 [  printer] 출력 완료
21:56:02.373 [  printer] 출력 시작2, 대기 문서 : [3, 4, 5]
q
21:56:02.938 [     main] 작업 중단 지시 thread.interrupt()
21:56:02.939 [  printer] interrupted : 프린터 종료
21:56:02.940 [  printer] 프린터 완료
```

- `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.

yield - 양보하기

어떤 쓰레드를 얼마나 실행할지는 운영체제가 스케줄링을 통해 결정, 그런데 특정 쓰레드가 크게 바쁘지 않은 상황이어서 다른 쓰레드에 CPU 실행 기회를 양보하고 싶을 수 있음, 이렇

게 양보하면 스케줄링 큐에 대기 중인 다른 스레드 가 CPU 실행 기회를 더 빨리 얻을 수 있다.

```
package thread.start.control.yield;

import thread.start.HelloRunnable;

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

- 1000개의 쓰레드를 실행
- 각 쓰레드는 0 ~ 9까지 출력

Empty : 운영체제의 쓰레드 스케줄링을 따름.

```
        public void run() {
            for (int i = 0; i < 10; i++) {
                System.out.println(Thread.currentThread().getName()+ " - " + i);
                // 1. empty
```

```
            // sleep(10) // 2. sleep
            // Thread.yield(); // 3. yield
        }
    }
}
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
```

- 다른 예시보다 상대적으로 하나의 쓰레드가 쭉 실행되다가 다른 쓰레드로 넘어감.
- 운영체제의 스케줄링 정책과 환경에 따라 다르나 대략 0.01초(10ms)정도 하나의 쓰레드가 실행되고, 다른 쓰레드로 넘어감.

sleep(1) : 특정 쓰레드를 잠시 쉬게함

```
    public void run() {
        for (int i = 0; i < 10; i++) {
            // 1. empty
            System.out.println(Thread.currentThread().getName()+ " - " + i);
            sleep(10);
            // Thread.yield(); // 3. yield
        }
Thread-837 - 9
Thread-899 - 8
Thread-667 - 9
Thread-944 - 9
Thread-815 - 9
Thread-899 - 9
```

- sleep(1)을 사용해서 쓰레드의 상태를 1ms간 아주 잠깐 RUNNABLE → TIMED_WAITING으로 변경.
- 이렇게 되면 CPU 자원을 사용하지 않고, 실행 스케줄링에서 잠시 제외 됨.
- 1ms 대기 시간 이후 TIMED_WAITING → RUNNABLE 상태가 되며 실행 스케줄링에 포함
- 결과적으로, TIMED_WAITING 상태가 되면서 다른 쓰레드에 실행을 양보하게 됨. 그리고 스케쥴링 큐에 대기중인 다른 쓰레드가 CPU의 실행 기회를 빨리 얻을 수 있음.

RUNNABLE → TIMED_WAITING → RUNNABLE로 변경되는 작업을 거치며, 또 특정 시간만큼 쓰레드가 실행되지 않는 단점.

예를들면 양보할 쓰레드가 없다면, 차라리 나의 쓰레드를 더 실행하는 것이 나은 선택일 수 있으나, 이 방법은 나머지 쓰레드가 모두 대기 상태로 쉬고있어도 내 쓰레드까지 잠깐 실행되지 않는다는 것. 양보할 사람이 없는데 혼자 양보하는 이상한 상황이 생길 수 있음.

yield : yield()를 사용해서 다른 쓰레드에 실행을 양보

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

Thread-960 - 8
Thread-805 - 8
Thread-960 - 9
Thread-991 - 9
Thread-805 - 9
Thread-838 - 9
```

- 자바의 쓰레드가 RUNNABLE 상태일 때 운영체제의 스케쥴링은 다음과 같은 상태들을 가질 수 있음.
    - 실행 상태 (RUNNING) : 쓰레드가 CPU에서 실제로 실행중.
    - 실행 대기 상태(READY) : 쓰레드가 실행될 준비가 되었지만, CPU가 바빠서 스케쥴링 큐에서 대기중.

- 운영체제는 실행 상태의 쓰레드들을 잠깐만 실행하고 실행 대기 상태로 만듦. 그리고 실행 대기 상태의 쓰레드들을 잠깐만 실행 상태로 변경해서 실행한다. 이 과정을 계속 반복하며, 자바는 두 상태를 구분할 수 없음.

yield()의 작동

- Thread.yield() 메서드는 현재 실행 중인 쓰레드가 자발적으로 CPU를 양보하여 다른 쓰레드가 실행될 수 있도록 함.
- yied() 메서드를 호출한 쓰레드는 RUNNABLE 상태를 유지하면서 CPU를 양보한다. 즉, 이 쓰레드는 다시 스케쥴링 큐에 들어가면서 다른 쓰레드에게 CPU 사용 기회를 넘긴다.

자바에서 `Thread.yield()` 메서드를 호출하면 현재 실행 중인 스레드가 CPU를 양보하도록 힌트를 준다. 이는 스레 드가 자신에게 할당된 실행 시간을 포기하고 다른 스레드에게 실행 기회를 주도록 한다. 참고로 `yield()` 는 운영체제 의 스케줄러에게 단지 힌트를 제공할 뿐, 강제적인 실행 순서를 지정하지 않는다. 그리고 반드시 다른 스레드가 실행되 는 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 없 다. 양보해도 CPU 코어가 남기 때문에 양보하지 않고 계속 수행될 수 있다. CPU 코어 수 이상의 스레드를 만들어야 양 보하는 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 예제 4 - yield 도입

```
package thread.start.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) {
        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")) {
                log("작업 중단 지시 thread.interrupt()");
                printerThread.interrupt();
                break;
            }
            printer.addJob(input);
        }
    }

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

        @Override
        public void run() {
            try {
                while (!Thread.interrupted()) {
                    if (jobQueue.isEmpty()) {
                        Thread.yield();
                        continue;
                    }

                    String job = jobQueue.poll();
                    log("출력 시작" + job + ", 대기 문서 : " + jobQueue);
                    Thread.sleep(3000);
                    log("출력 완료");

                }
```

```
            } catch (InterruptedException e){
                log("interrupted : 프린터 종료");
            }

            log("프린터 완료");
        }

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

✔️ join()

- 돌고 있는 쓰레드의 작업이 완료될 때까지 기다리는 메서드
- 대상 쓰레드가 TERMINATED 될 때까지 호출한 쓰레드는 WAITING 상태
- 특정 쓰레드의 작업 완료 후 후속 작업이 필요할 때 사용

✔️ interrupt()

- 쓰레드에게 “중단 요청” 신호를 보내는 기능
- sleep(), wait(), join() 등으로 WAITING/TIMED_WAITING 상태라면 InterruptedException 발생 후 interrupt flag가 자동으로 false로 초기화됨
- RUNNABLE 상태라면 예외는 발생하지 않고 interrupt flag만 true로 되고, 개발자가 직접 isInterrupted()로 체크해야 함
- 즉, RUNNABLE 상태인 쓰레드에는 즉시 영향이 없음

✔️ yield()

- 현재 실행 중인 쓰레드가 CPU를 양보하도록 OS 스케줄러에게 힌트를 주는 것
- RUNNABLE 상태 그대로 유지
- 실제로 CPU를 양보할지는 OS가 결정(보장 X)