Chapter 04

문제 4-1 해답

×… (1) save 메소드는 SaverThread에서만 호출할 수 있다.

➜ save 메소드는 SaverThread와 ChangerThread 양쪽으로부터 호출됩니다.

×… (2) synchronized를 모두 삭제하고 changed 필드를 volatile로 하더라도 동작은 같다.

➜ change 메소드 save 메소드가 각각 Single Threaded Execution 패턴(Chapter 01)이 아니어서 동작이 같을 수 없습니다. volatile 해도 쓰레드의 배타제어는 일어나지 않기 때문입니다. 자세한 내용은 부록 B 「Java의 메모리 모델」을 참조하세요.

×… (3) change 메소드에서는 content 필드에 대입한 뒤 changed를 true로 하지 않으면 안 된다.

➜ changed를 먼저 true로 하더라도 동작에는 차이가 없습니다.

○… (4) doSave 메소드는 synchronized 메소드가 아니다.

➜ doSave 메소드는 synchronized 메소드가 아닙니다. 그러나 doSave 메소드는 synchronized 메소드에서 밖에 호출되지 않으므로 doSave 메소드에 온 쓰레드는 반드시 this 락을 갖고 있습니다.

○… (5) doSave 메소드를 2개의 쓰레드에서 동시에 호출하는 일은 없다.

➜ save 메소드가 synchronized이고, private한 doSave 메소드는 save 메소드에서만 호출되고 있습니다. 따라서 doSave 메소드가 동시에 2개의 쓰레드로부터 호출되는 일은 없습니다.

문제 4-2 해답

(1)과 (2)에서 요구한 대로 수정한 Data 클래스는 각각 〈리스트 4-1〉과 〈리스트 4-2〉와 같습니다.

(1) 디버그 프린트의 추가

save 메소드 안에서 return하고 있는 부분이 balk에 해당하므로 그 바로 앞에 디버그 프린트를 넣었습니다.

리스트 4-1 디버그 프린트를 추가한 Data 클래스 (Data.java)

```
import java.io.IOException;
import java.io.FileWriter;
import java.io.Writer;

public class Data {
    private final String filename;  // 저장하는 파일의 이름
    private String content;         // 데이터의 내용
    private boolean changed;        // 변경한 내용이 저장되지 않았다면 true

    public Data(String filename, String content) {
        this.filename = filename;
        this.content = content;
        this.changed = true;
    }

    // 데이터의 내용을 바꾼다
    public synchronized void change(String newContent) {
        content = newContent;
        changed = true;
    }

    // 데이터의 내용이 변경되었으면 파일에 저장한다
    public synchronized void save() throws IOException {
        if (!changed) {
            System.out.println(Thread.currentThread().getName() + " balks");
            return;
        }
        doSave();
        changed = false;
    }

    // 데이터의 내용을 실제로 파일에 저장한다
    private void doSave() throws IOException {
        System.out.println(Thread.currentThread().getName() + " calls
                           doSave, content = " + content);
        Writer writer = new FileWriter(filename);
        writer.write(content);
        writer.close();
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/Balking/A4-2a*

그림 4-1 실행 예 (확실히 balk하고 있다)

```
SaverThread calls doSave, content = No.0
ChangerThread blaks
ChangerThread calls doSave, content = No.1
ChangerThread calls doSave, content = No.2
SaverThread calls doSave, content = No.3
ChangerThread blaks
SaverThread calls doSave, content = No.4
ChangerThread blaks
ChangerThread calls doSave, content = No.5
SaverThread calls doSave, content = No.6
ChangerThread blaks
(중략)
ChangerThread blaks
SaverThread calls doSave, content = No.78
ChangerThread blaks
ChangerThread calls doSave, content = No.79
SaverThread calls doSave, content = No.80
ChangerThread blaks
SaverThread blaks
ChangerThread calls doSave, content = No.81
SaverThread calls doSave, content = No.82
ChangerThread blaks
(이하 생략. CTRL+C로 종료)
```

(2) synchronized의 삭제

synchronized를 삭제하고 changed 플래그를 false로 하기 전에 약 100밀리 초 sleep 하도록 만들었습니다.

이렇게 하면 실행 예(그림 4-2)처럼 중복 입력됩니다.

리스트 4-2 디버그 프린트를 추가하여 synchronized를 삭제한 Data 클래스 (Data.java)

```
import java.io.IOException;
import java.io.FileWriter;
import java.io.Writer;

public class Data {
    private final String filename;  // 저장할 파일의 이름
    private String content;         // 데이터의 내용
    private boolean changed;        // 변경된 내용이 저장되지 않았다면 true

    public Data(String filename, String content) {
        this.filename = filename;
```

```
        this.content = content;
        this.changed = true;
    }

    // 데이터의 내용을 수정한다
    public synchronized void change(String newContent) {
        content = newContent;
        changed = true;
    }

    // 데이터의 내용이 변경되었다면 저장한다
    public void save() throws IOException {    // not synchronized
        if (!changed) {
            System.out.println(Thread.currentThread().getName() + "
                               balks");
            return;
        }
        doSave();
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
        }
        changed = false;
    }

    // 데이터의 내용을 실제로 파일에 저장한다
    private void doSave() throws IOException {
        System.out.println(Thread.currentThread().getName() + " calls
                   doSave, content = " + content);
        Writer writer = new FileWriter(filename);
        writer.write(content);
        writer.close();
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/Balking/A4-2.*

그림 4-2 실행 예

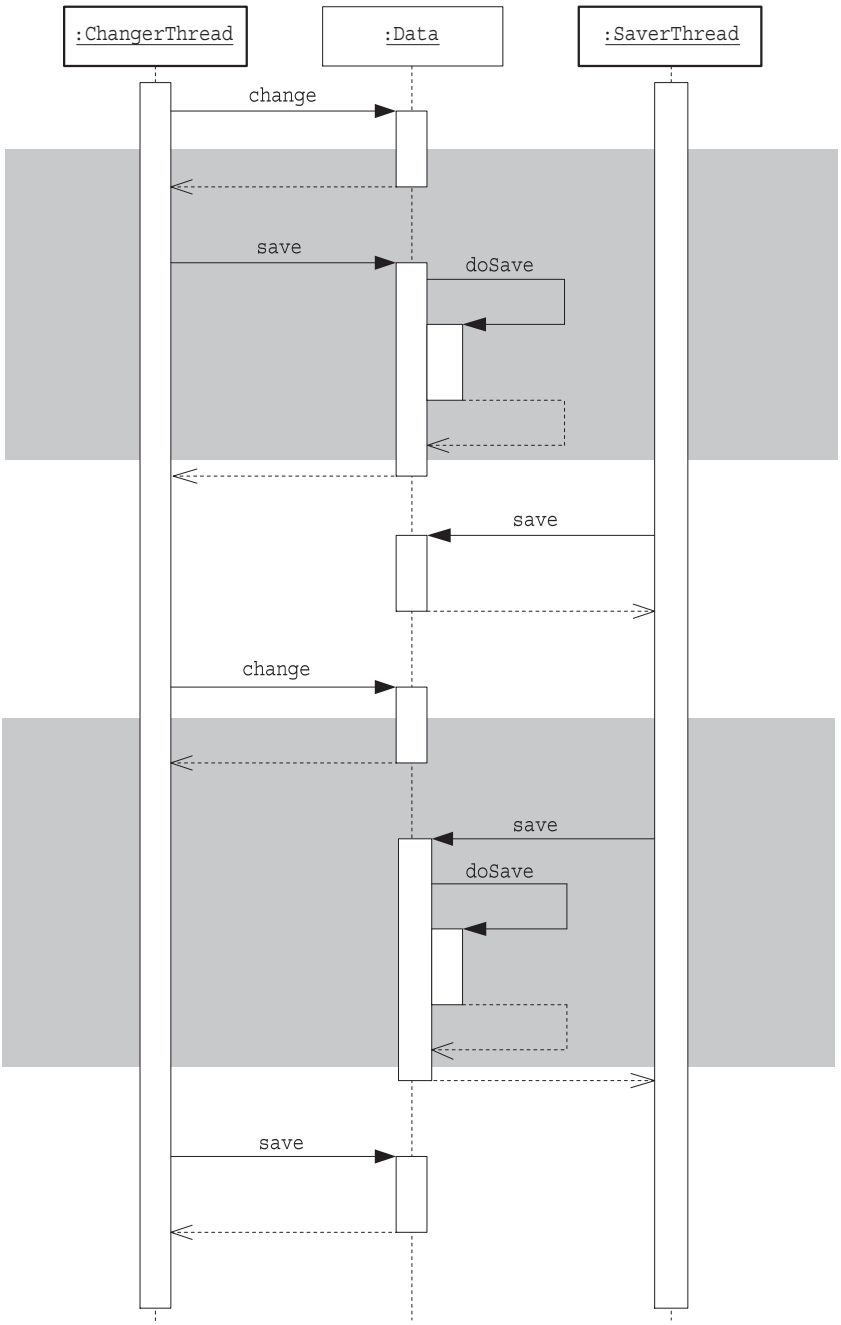
```
ServerThread calls doSave, content = No.0
ChangerThread balks
ServerThread calls doSave, content = No.1
ChangerThread balks
ServerThread calls doSave, content = No.2
ChangerThread calls doSave, content = No.2 ← No.2를 2번 적고 있다
ChangerThread calls doSave, content = No.3
ServerThread calls doSave, content = No.4
ChangerThread balks
ChangerThread calls doSave, content = No.5
ServerThread calls doSave, content = No.6
ChangerThread balks
ChangerThread calls doSave, content = No.7
ServerThread calls doSave, content = No.8
ChangerThread balks
ServerThread calls doSave, content = No.9
ChangerThread calls doSave, content = No.9 ← No.9를 2번 적고 있다
ChangerThread calls doSave, content = No.10
ServerThread calls doSave, content = No.10 ← No.10을 2번 적고 있다
ChangerThread balks
ServerThread calls doSave, content = No.12
(이하 생략. CTRL+C로 종료)
```

문제 4-3 해답

〈그림 4-3〉처럼 됩니다. 여기에서는 changed 필드의 true 범위를 회색으로 표시했습니다. change 메소드 실행 도중에 회색 영역이 시작되고, doSave 메소드 실행한 후에 회색 영역이 종료됩니다.

changed 필드의 true 범위(회색 배경)에서 save 메소드가 호출되면 doSave도 함께 호출되지만 false 범위(흰색 배경)에서는 save가 호출되어도 doSave가 호출되지 않는 모습을 확인할 수 있습니다.

그림 4-3 changed 필드의 true 범위를 표시한 시퀀스 다이어그램



문제 4-4 해답

해답 1 : java.util.LinkedList를 사용한 경우

수정한 RequestQueue 클래스는 〈리스트 4-3〉과 같습니다.

여기에서는 while 루프 안의 「매번 wait의 타임아웃이 약 30초」가 아니라, 한번 while 루프에 들어간 다음 「가드 조건이 충족될 때까지의 타임아웃이 약 30초」가 되도록 코딩했습니다. 가령 wait이 몇 번씩 interrupt되었다 하더라도 약 30초면 예외 LivenessException(리스트 4-8)을 throw합니다.

리스트 4-3 해답 1 : RequestQueue 클래스 (RequestQueue.java)

```
import java.util.Queue;
import java.util.LinkedList;

public class RequestQueue {
    private static final long TIMEOUT = 30000L;
    private final Queue<Request> queue = new LinkedList<Request>();
    public synchronized Request getRequest() {
        long start = System.currentTimeMillis();   // 개시시각
        while (queue.peek() == null) {
            long now = System.currentTimeMillis(); // 현재 시각
            long rest = TIMEOUT - (now - start);   // 나머지 대기 시각
            if (rest <= 0) {
                throw new LivenessException("thrown by " +
                   Thread.currentThread().getName());
            }
            try {
                wait(rest);
            } catch (InterruptedException e) {
            }
        }
        return queue.remove();
    }
    public synchronized void putRequest(Request request) {
        queue.offer(request);
        notifyAll();
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/Balking/A4-4a*

데드락이 발생했던 문제 3-5의 RequestQueue 클래스를 앞의 버전으로 바꾸어 동작을 시켜보면 〈그림 4-4〉처럼 됩니다. 실행하면 곧바로 다음과 같은 표시가 나온 뒤 멈춥니다.

```
Alice:BEGIN
Bobby:BEGIN
```

그 대로 약 30초간(두근두근하면서) 기다리면 Alice와 Bobby의 쓰레드가 각각 Liveness Exception을 통보하는 것을 알 수 있습니다. 타임아웃에 의한 검출이 잘 기능하고 있는 것 같군요.

그림 4-4 해답 1: 데드락 검출 예

```
Alice:BEGIN
Bobby:BEGIN ← 이 표시가 나온 뒤 약 30초 기다린다
Exception in thread "Alice" LivenessException : throw by Alice ← Alice가
통보한 LivenessException
      at RequestQueue.getRequest(RequestQueue.java:13)
      at TalkThread.run(TalkThread.java:13)
Exception in thread "Bobby" LivenessException : throw by Bobby ← Bobby가
통보한 LivenessException
     at RequestQueue.getRequest(RequestQueue.java:13)
      at TalkThread.run(TalkThread.java:13)
```

해답 2 : java.util.concurrent.LinkedBlockingQueue를 사용한 경우

수정한 RequestQueue 클래스는 〈리스트 4-4〉와 같습니다.

java.util.concurrent.LinkedBlockingQueue 클래스에는 타임아웃이 붙은 메소드가 이미 준비되어 있습니다.

다음과 같이 poll 메소드를 사용할 경우 약 30초 이내에 요소를 얻지 못하면(타임아웃하면) 반환 값은 null이 됩니다.

```
req = queueu.poll(30L, TimeUnit.SECONDS);
```

또한 다음과 같이 offer 메소드를 사용할 경우, 약 30초 이내에 요소를 추가하지 못하면(타임아웃하면) 반환 값은 false가 됩니다.

```
offered = queue.offer(request, 30L, TimeUnit.SECONDS);
```

여기에서 사용하고 있는 java.util.concurrent.TimeUnit 클래스는 타임아웃 값을 long으로 부여할 때의 「단위」를 지정하는 Enum형(Java SE 5.0부터 도입된 정수)입니다.

리스트 4-4 해답 2: RequestQueue 클래스 (RequestQueue.java)

```
import java.util.concurrent.TimeUnit;

import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

public class RequestQueue {
    private final BlockingQueue<Request> queue = new
                  LinkedBlockingQueue<Request>();
    public Request getRequest() {
        Request req = null;
        try {
            req = queue.poll(30L, TimeUnit.SECONDS);
            if (req == null) {
                throw new LivenessException("thrown by " +
                          Thread.currentThread().getName());
            }
        } catch (InterruptedException e) {
        }
        return req;
    }
    public void putRequest(Request request) {
        try {
            boolean offered = queue.offer(request, 30L, TimeUnit.SECONDS);
            if (!offered) {
                throw new LivenessException("thrown by " +
                          Thread.currentThread().getName());
            }
        } catch (InterruptedException e) {
        }
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/Balking/A4-4*

해답 2의 실행 결과는 해답 1(그림A4-4)과 같습니다.

문제 4-5 해답

Main 클래스의 main 메소드 안에서 TestThread의 한 개 인스턴스에 대하여 start 메

소드를 몇 번씩 호출하고 있기 때문입니다. 한 개의 인스턴스에 대하여 **start 메소드는 한 번밖에 호출할 수 없습니다.**

Thread 클래스(또는 그 서브 클래스)의 인스턴스는 start 메소드가 호출되면 「start 완료」 상태가 됩니다. 여기에서 말하는 「start 완료」란 Thread.getState()의 값이 Thread.State.NEW가 아니라는 뜻입니다.

Start 메소드가 재차 호출된 경우에도 쓰레드의 기동이 두 번 다시 일어나지 않도록 balk하여 예외 IllegalThreadStateException을 통보합니다.

Thread 클래스의 start메소드는 Chapter 04에서 이야기한 「2번 이상은 실행할 수 없는 처리」입니다. 즉, Thread 클래스의 start 메소드에는 Balking 패턴이 사용되고 있는 것입니다. 문제의 실행 예(그림 4-6)에서 딱 한 번 등장한 BEGIN…END 표시는 제일 처음 start 했을 때 기동한 쓰레드가 표시하는 것입니다.

쓰레드의 상태 변화는 Introduction1의 〈그림 I1-22〉를 참조하세요.