Chapter 10

문제 10-1의 해답

○… (1) shutdownRequest 메소드를 호출하고 있는 것은 메인 쓰레드이다.

×… (2) doWork 메소드가 호출되는 것은 한 번 뿐이다.

➜ doWork 메소드는 while문 안에서 호출되고 있기 때문에 대개의 경우 몇 번이고 호출됩니다.

○… (3) doWork 메소드가 예외 InterruptedException을 통보했을 때에도 doShutdown 메소드는 호출된다.

➜ 예외 InterruptedException이 통보되어도 finally가 실행됩니다.

×… (4) shutdownRequest 메소드 안의 interrupt()는 Thread.current Thread().interrupt()라고 써도 의미가 같다.

➜ shutdownRequest 메소드 안의 interrupt()는 **this.interrupt()**와 같습니다. 이 때 어떤 쓰레드가 shutdownRequest를 호출했다 하더라도 **CountupThread의 쓰레드에 대하여** 인터럽트가 걸립니다.

한편 shutdownRequest 메소드 안에 Thread.currentThread().interrupt()라고 적혀 있다면 어떨까요? 식(式) Thread.currentThread()의 값은 이 메소드를 호출한 쓰레드에 대응한 객체가 됩니다. 그러므로 shutdownRequest 메소드 안에 Thread.currentThread().interrupt()가 적혀 있으면 **shutdownRequest를 호출한 쓰레드에 대하여** 인터럽트가 걸립니다. 예제 프로그램의 경우에는 메인 쓰레드에 인터럽트가 걸립니다.

문제 10-2의 해답

예를 들면 〈리스트 10-1〉과 같이 할 수 있습니다. 여기에서는 sleep 메소드의 호출을 try…catch로 묶고 예외 InterruptedException을 무시하고 있습니다. 이 문제에서 shutdownRequested 필드의 존재 의의를 알 수 있습니다.

리스트 10-1 CountupThread 클래스 (CountupThread.java)

```
public class CountupThread extends Thread {
    // 카운터의 값
    private long counter = 0;

    // 종료 요구
    public void shutdownRequest() {
        interrupt();
    }

    // 동작
    public void run() {
        try {
            while (!isInterrupted()) {
                doWork();
            }
        } catch (InterruptedException e) {
        } finally {
            doShutdown();
        }
    }

    // 작업
    private void doWork() throws InterruptedException {
        counter++;
        System.out.println("doWork: counter = " + counter);
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
        }
    }

    // 종료 처리
    private void doShutdown() {
        System.out.println("doShutdown: counter = " + counter);
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/TwoPhaseTermination/A10-2*

그림 10-1 실행 예 (종료하지 않는다)

```
main:BEGIN
doWork: counter = 1
doWork: counter = 2
doWork: counter = 3
doWork: counter = 4
doWork: counter = 5
doWork: counter = 6
doWork: counter = 7
doWork: counter = 8
doWork: counter = 9
doWork: counter = 10
doWork: counter = 11
doWork: counter = 12
doWork: counter = 13
doWork: counter = 14
doWork: counter = 15
doWork: counter = 16
doWork: counter = 17
doWork: counter = 18
doWork: counter = 19
doWork: counter = 20
main:shutdownRequest      ← 종료 요구
doWork: counter = 21      ← 종료 처리로 이행하지 않는다
main: join
doWork: counter = 22 ┐
doWork: counter = 23 │ 통상적인 처리가 계속된다
doWork: counter = 24 │
doWork: counter = 25 ┘
(이하 생략. CTRL+C로 종료)
```

문제 10-3의 해답

예를 들면 〈리스트 10-2〉와 같이 됩니다. 실행 예는 〈그림 10-2〉와 같습니다.

리스트 10-2 파일을 보관하는 CountupThread 클래스 (CountupThread.java)

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

public class CountupThread extends Thread {
    // 카운터의 값
    private long counter = 0;

    // 종료 요구가 제시되었다면 true
```

```
    private volatile boolean shutdownRequested = false;

    // 종료 요구
    public void shutdownRequest() {
        shutdownRequested = true;
        interrupt();
    }

    // 종료 요구가 제시되었는지를 테스트
    public boolean isShutdownRequested() {
        return shutdownRequested;
    }

    // 동작
    public void run() {
        try {
            while (!isShutdownRequested()) {
                doWork();
            }
        } catch (InterruptedException e) {
        } finally {
            doShutdown();
        }
    }

    // 작업
    private void doWork() throws InterruptedException {
        counter++;
        System.out.println("doWork: counter = " + counter);
        Thread.sleep(500);
    }

    // 종료 처리
    private void doShutdown() {
        System.out.println("doShutdown: counter = " + counter);
        System.out.println("doShutdown: Save BEGIN");
        try {
            FileWriter writer = new FileWriter("counter.txt");
            writer.write("counter = " + counter);
            writer.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
        System.out.println("doShutdown: Save END");
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/TwoPhaseTermination/A10-3*

그림 10-2 실행 예

```
main:BEGIN
doWork: counter = 1
doWork: counter = 2
doWork: counter = 3
(중략)
doWork: counter = 19
doWork: counter = 20
main:shutdownRequest
doShutdown:counter = 20
main:join
doShutdown: Save BEGIN   ← 보존 개시
doShutdown: Save END     ← 보존 종료
main:END
```

문제 10-4의 해답

GracefulThread를 확장한 CountupThread 클래스는 〈리스트 10-3〉처럼 됩니다. 여기에서는 다음과 같이 합니다.

:: GracefulThread 클래스를 확장(extends)하여 CountupThread 클래스를 선언합니다.

:: GracefulThread 클래스의 메소드 doWork, doShutdown을 오버라이드합니다.

CountupThread의 doWork, doShutdown은 슈퍼 클래스의 run 메소드로부터 적절한 타이밍에 적절한 순서로 호출됩니다.

이처럼 슈퍼 클래스의 메소드에서 처리의 골격을 짜고, 그 메소드가 호출하는 메소드를 서브 클래스에서 구현하여 구체적인 살(처리)을 붙여나가는 패턴을 **Template Method 패턴**이라 부릅니다. 〈그림 10-3〉은 클래스 다이어그램입니다.

슈퍼 클래스에서 처리의 골격을 짜는 메소드를 **템플릿 메소드**(template method)라 부릅니다. GracefulThread에서는 run이 템플릿 메소드입니다(그래서 run을 final한 메소드로 했습니다).

또한 서브 클래스에서 살을 붙이는(처리를 하는) 메소드를 **후크 메소드**(hook method)라 합니다(hook은 「걸다)라는 뜻입니다). 이 문제에서는 doWork나 doShutdown이 후크 메소드입니다.

Java SE 5.0에 도입된 @Override라고 하는 어노테이션을 사용하면 〈리스트 10-3〉에

서처럼 오버라이드 한 메소드임을 명기할 수 있습니다. 만일 doWork나 doShutdown의 이름 또는 인수가 잘못돼서 오버라이드하지 않는 결과가 만들어진다면 javac가 컴파일 할 때 에러 메시지를 띄웁니다.

리스트 10-3 GracefulThread를 확장한 CountupThread 클래스 (CountupThread.java)

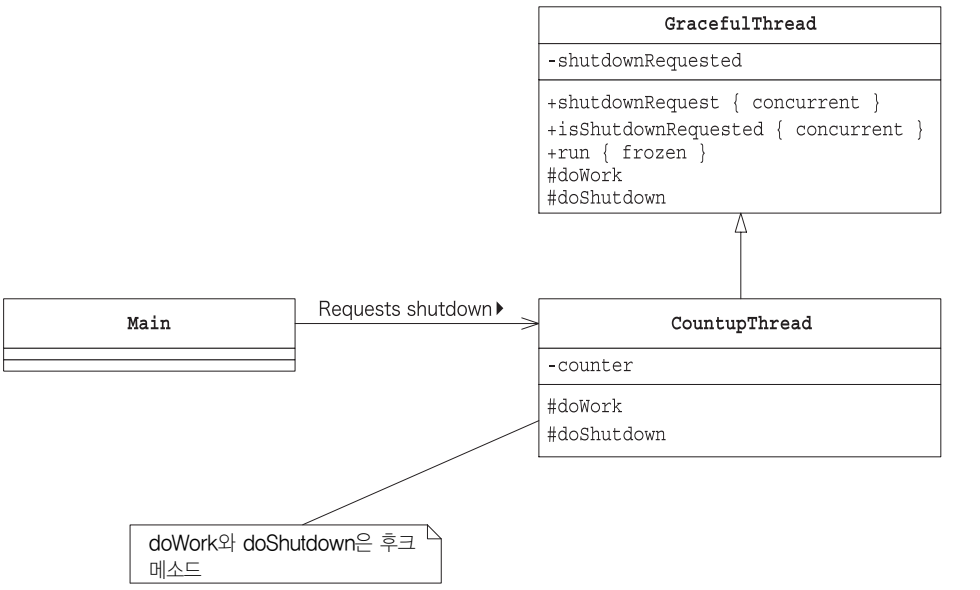
```
public class CountupThread extends GracefulThread {
    // 카운터의 값
    private long counter = 0;

    // 작업
    @Override
    protected void doWork() throws InterruptedException {
        counter++;
        System.out.println("doWork: counter = " + counter);
        Thread.sleep(500);
    }

    // 종료 처리
    @Override
    protected void doShutdown() {
        System.out.println("doShutdown: counter = " + counter);
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/TwoPhaseTermination/A10-4*

그림 10-3 Template Method 패턴을 사용한 CountupThread의 클래스 다이어그램



문제 10-5의 해답

해답은 〈리스트 10-4〉, 〈리스트 10-5〉와 같습니다. 문제 10-4에서 사용한 Graceful Thread 클래스(리스트 10-5)를 계승하여 SeviceThread 클래스를 만들고 Service 클래스로부터 ServiceThread 클래스를 기동시켰습니다. 이것은 **Thread-Per-Message 패턴**(Chapter 07)입니다. 〈그림 10-4〉는 실행을 취소하는 모습입니다.

[Execute] 버튼을 연속해서 누른 경우에는 **Balking 패턴**(Chapter 04)을 사용하여 balk하도록 해 보았습니다. 〈그림 10-5〉는 balk하고 있는 모습입니다.

리스트 10-4 Service 클래스 (Service.java)

```
public class Service {
    private static GracefulThread thread = null;

    // 서비스 실행 개시(실행 중이라면 balk한다)
    public synchronized static void service() {
        System.out.print("service");
        if (thread != null && thread.isAlive()) {
            // Balking
            System.out.println(" is balked.");
            return;
        }
        // Thread-Per-Message
        thread = new ServiceThread();
        thread.start();
    }

    // 서비스 중지
    public synchronized static void cancel() {
        if (thread != null) {
            System.out.println("cancel.");
            thread.shutdownRequest();
        }
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/TwoPhaseTermination/A10-5*

리스트 10-5 ServiceThread 클래스 (ServiceThread.java)

```
public class ServiceThread extends GracefulThread {
    private int count = 0;

    // 작업 중
    @Override
    protected void doWork() throws InterruptedException {
        System.out.print(".");
        Thread.sleep(100);
        count++;
        if (count >= 50) {
            shutdownRequest();  // 스스로 종료
        }
    }

    // 종료 처리
    @Override
    protected void doShutdown() {
        System.out.println("done.");
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/TwoPhaseTermination/A10-5*

그림 10-4 서비스 실행 중에 [Cancel] 버튼을 눌렀다

```
service.........................cancel.
done.
```

그림 10-5 서비스 실행 중에 [Execute] 버튼을 3번 눌렀다 (Balking)

```
service..............service is balked.
.........service is balked.
.....service is balked.
.............................done.
```

문제 10-6의 해답

Java의 메모리 모델에서는 복수의 쓰레드가 공유하는 필드는 synchronized로 보호하든지 volatile로 선언해야 합니다. 그렇지 않으면 어느 한 쓰레드가 실행한 필드 수정이 다른 쓰레드에서 보이지 않을 가능성이 있기 때문입니다.

shutdownRequest 필드는 여러 쓰레드에서 공유하고 있고 또 복수의 쓰레드가 액세스하기 때문에 필드를 synchronized로 하든지 volatile로 선언해야 합니다. synchronized로 하는 경우에는 쓰레드의 배타제어와 동기화가 이뤄집니다. volatile로 선언한 경우에는 동기화만 이뤄집니다.

예제 프로그램의 경우 쓰레드의 배타제어는 불필요하지만 동기화는 필요합니다. 따라서 shutdownRequest 필드를 volatile 선언하고 있는 것입니다.

쓰레드의 배타제어 및 동기화에 대해서는 부록 B를 참조해 주세요.

문제 10-7의 해답

해답은 〈리스트 10-6〉과 같습니다. HanoiThread 클래스의 doWork 메소드 안에서 isShutdownRequest 메소드를 호출하고, 만일 종료 처리 요구가 제시된 상태라면 바로 예외 InterruptedException을 통보합니다.

실행 예는 〈그림 10-6〉과 같습니다. 종료 처리 요구가 나오고 나서 종료 처리에 들어갈 때까지의 시간이 거의 0이 됩니다.

리스트 10-6 HanoiThread 클래스 (HanoiThread.java)

```
public class HanoiThread extends Thread {
    // 종료 요구가 제시됐으면 true
    private volatile boolean shutdownRequested = false;
    // 종료 요구가 제시된 시각
    private volatile long requestedTimeMillis = 0;

    // 종료 요구
    public void shutdownRequest() {
        requestedTimeMillis = System.currentTimeMillis();
        shutdownRequested = true;
        interrupt();
    }

    // 종료 요구가 제시되었는지를 테스트
    public boolean isShutdownRequested() {
```

```
        return shutdownRequested;
    }

    // 동작
    public void run() {
        try {
            for (int level = 0; !isShutdownRequested(); level++) {
                System.out.println("==== Level " + level + " ====");
                doWork(level, 'A', 'B', 'C');
                System.out.println("");
            }
        } catch (InterruptedException e) {
        } finally {
            doShutdown();
        }
    }

    // 작업
    private void doWork(int level, char posA, char posB, char posC)
             throws InterruptedException {
        if (level > 0) {
            if (isShutdownRequested()) {
                throw new InterruptedException();
            }
            doWork(level - 1, posA, posC, posB);
            System.out.print(posA + "->" + posB + " ");
            doWork(level - 1, posC, posB, posA);
        }
    }

    // 종료 처리
    private void doShutdown() {
        long time = System.currentTimeMillis() - requestedTimeMillis;
        System.out.println("doShutdown: Latency = " + time + " msec.");
    }
}
```

⇒ 예제파일 경로 : 부록CD/src/TwoPhaseTermination/A10-7

그림 10-6 실행 예

```
main: BEGIN
=== Level 0 ===

=== Level 1 ===
A->B
=== Level 2 ===
A->C A->B C->B
=== Level 3 ===
A->B A->C B->C A->B C->A C->B A->B
=== Level 4 ===
A->C A->B C->B A->C B->A B->C A->C A->B C->B C->A B->A C->B A->C A->B C->B
=== Level 5 ===
A->B A->C B->C A->B C->A C->B A->B A->C B->C B->A C->A B->C A->B A->C B->C A->B
C->A C->B A->B C->A B->C B->A A->B A->C B->C A->B C->A C->B A->B
=== Level 6 ===
A->C A->B C->B A->C B->A B->C A->C A->B C->B C->A B->A C->B A->C A->B C->B A->C
B->A B->C A->C B->A C->B C->A B->A B->C A->C A->B C->B A->C B->A B->C A->C A->B
C->B A->C A->B C->B C->A B->A B->C A->C B->A C->B C->A B->A C->B A->C A->B C->B
A->C B->A B->C A->C A->B C->B C->A B->A C->B A->C A->B C->B
(중략)
C->A C->B A->B C->A B->C B->A C->A C->B A->B A->C B->C A->B C->A C->B A->B A->C
B->C B->A C->A B->C A->B A->C B->C B->A C->A C->B A->B C->A B->C B->A C->A B->C
A->B A->C B->C A->B C->A C->B A->B A->C B->C B->A C->A B->C A->B A->C B->C B->A
C->A C->B A->B C->A B->C B->A C->A C->B A->B A->C B->C A->B C->A C->B A->B C->A
main: shutdownREquest
main:join
doShutdown: Latency = 0 msec. ← 종료 요구에서 종료 처리 시작까지의 시간
main: END
```

문제 10-8의 해답

약 5초 동안은 마침표(.)가 반복 표시되고, 그 후에는 별표(*)가 반복 표시됩니다. 실행 예는 〈그림 10-7〉과 같습니다.

메인 쓰레드는 약 5초 후에 만든 쓰레드 t에 interrupt를 겁니다. 이로써 t는 인터럽트 상태가 되었습니다.

식(式) Thread.currentThread().isInterrupted()를 평가하면 현재 쓰레드의 인트럽트 상태를 알 수 있습니다. 하지만 **isInterrupted 메소드는 인터럽트 상태를 삭제하지 않습니다.** 때문에 그 후 쓰레드 t는 while 루프를 돌 때마다 예외 InterruptedException을 계속해서 통보하게 됩니다. 그래서 별표 (*)가 계속 표시되는 것입니다.

그림 10-7 리스트 10-15의 실행 예

```
(전략)
.....................................
.....................................
.....................................
.....................................
.....................................
.....................................
.................********************      ← 약 5초 후부터 *가 계속 표시된다
**************************************
**************************************
**************************************
**************************************
**************************************
**************************************
(이하 생략. CTRL+C로 종료)
```

반면 〈리스트 10-7〉처럼 Thread.interrupted 메소드를 사용하면 인터럽트 상태를 삭제하기 때문에 별표는 한 개만 표시됩니다(그림 10-8).

리스트 10-7 Thread.interrupted 메소드를 사용한 경우 (Main.java)

```
public class Main {
    public static void main(String[] args) {
        // 쓰레드를 생성한다
        Thread thread = new Thread() {
            public void run() {
                while (true) {
                    try {
                        if (Thread.interrupted()) {
                            throw new InterruptedException();
                        }
                        System.out.print(".");
                    } catch (InterruptedException e) {
                        System.out.print("*");
                    }
                }
            }
        };

        // 쓰레드를 기동한다
        thread.start();

        // 5초 대기
        try {
```

```
            Thread.sleep(5000);
        } catch (InterruptedException e) {
        }

        // 쓰레드에 interrupt를 1번만 건다
        thread.interrupt();
    }
}
```

⇒ ***예제파일 경로 : 부록CD/src/TwoPhaseTermination/A10-8***

그림 10-8 리스트 10-7의 실행 예

```
(전략)
.....................................
.....................................
.....................................
.....................................
..................*..................   ← 약 5초 후에 *가 한 개만 표시된다
.....................................
.....................................
.....................................
.....................................
.....................................
(이하 생략. CTRL+C로 종료)
```