Chapter 08

문제 8-1의 해답

×… (1) 리퀘스트가 한 개도 없을 때 WorkerThread 쓰레드는 sleep하고 있다.

➜ 리퀘스트가 없을 때 WorkerThread의 쓰레드는 Channel 인스턴스 상에서 wait합니다(sleep가 아닙니다).

×… (2) 어떤 ClientThread의 리퀘스트가 execute되어 있는 동안에 같은 ClientThread의 다음 리퀘스트가 execute되는 일은 없다

➜ 한 개의 execute가 완료되기 전에 다른 WorkerThread의 쓰레드가 똑같은 ClientThread에서 제시한 리퀘스트를 execute하는 경우가 있습니다. 예제 프로그램의 경우에는 빈번하게 일어나고 있습니다.

○… (3) putRequest 메소드를 호출하는 것은 ClientThread 뿐이다.

○… (4) takeRequest 메소드를 호출하는 것은 WorkerThread 뿐이다.

○… (5) execute 메소드를 synchronized로 할 필요는 없다.

문제 8-2의 해답

예제 프로그램에서는 한 개의 Request 메소드 안에서 쓰레드를 기동하고 있습니다. 이것은 Thread-Per-Message 패턴(Chapter 07)입니다. 실행 예는 〈그림 8-1〉과 같습니다.

리스트 8-1 매번 새로운 쓰레드를 기동하는 Channel 클래스 (Channel.java)

```
public final class Channel {
    public Channel(int threads) {
    }
    public void startWorkers() {
    }
    public void putRequest(final Request request) {
        new Thread() {
            public void run() {
                request.execute();
```

```
            }
        }.start();
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/W●rkerThread/A8-2*

Thread-0, Thread-1, Thread-2, … 와 같은 식으로 번호가 늘어나기 때문에 매번 새로운 쓰레드가 실행되고 있음을 확실히 알 수 있습니다.

「Thread-숫자」 형식의 쓰레드 이름은 java.lang.Thread 클래스에 의해 자동적으로 붙여집니다.

그림 8-1 실행 예

```
Thread-1 executes [Request from Alice No.0]
Thread-2 executes [Request from Bobby No.0]
Thread-3 executes [Request from Chris No.0]
Thread-4 executes [Request from Chris No.1]
Thread-5 executes [Request from Alice No.1]
Thread-6 executes [Request from Alice No.2]
Thread-7 executes [Request from Bobby No.1]
Thread-8 executes [Request from Bobby No.2]
Thread-9 executes [Request from Chris No.2]
Thread-10 executes [Request from Alice No.3]
Thread-11 executes [Request from Bobby No.3]
Thread-12 executes [Request from Alice No.4]
Thread-13 executes [Request from Chris No.3]
Thread-14 executes [Request from Chris No.4]
Thread-15 executes [Request from Bobby No.4]
Thread-16 executes [Request from Chris No.5]
Thread-17 executes [Request from Alice No.5]
(이하 생략. CTRL+C로 종료)
```

문제 8-3의 해답

예제 프로그램의 Main 클래스를 실행 개시 약 30초 후에 강제 종료하도록 수정했습니다(리스트 8-2). 리퀘스트 번호가 어디까지 진행되었는지를 비교함으로써 두 프로그램간 쓰루풋의 차이를 조사합니다. 차이점이 확실히 구분되게끔 쓰레드 안에서의 대기 시간을

리스트 8-2 강제 종료하도록 수정한 Main 클래스 (Main.java)

```
public class Main {
    public static void main(String[] args) {
        Channel channel = new Channel(5);   // 워커 쓰레드의 개수
        channel.startWorkers();
        new ClientThread("Alice", channel).start();
        new ClientThread("Bobby", channel).start();
        new ClientThread("Chris", channel).start();

        try {
            Thread.sleep(30000);
        } catch (InterruptedException e) {
        }
        System.exit(0);
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/WorkerThread/A8-3a, A8-3b*

리스트 8-3 대기 시간을 없앤 ClientThread 클래스 (ClientThread.java)

```
public class ClientThread extends Thread {
    private final Channel channel;
    public ClientThread(String name, Channel channel) {
        super(name);
        this.channel = channel;
    }
    public void run() {
        for (int i = 0; true; i++) {
            Request request = new Request(getName(), i);
            channel.putRequest(request);
        }
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/WorkerThread/A8-3a, A8-3b*

없앴습니다(리스트 8-3, 리스트 8-4).

리스트 8-4 대기 시간을 없앤 Request 클래스 (Request.java)

```
public class Request {
    private final String name;
    private final int number;
    public Request(String name, int number) {
        this.name = name;
        this.number = number;
    }
    public void execute() {
        System.out.println(Thread.currentThread().getName() + "
                executes " + this);
    }
    public String toString() {
```

```
        return "[ Request from " + name + " No." + number + " ]";
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/WorkerThread/A8-3a, A8-3b*

실행 환경이 다르면 결과의 수치도 크게 달라지므로 2개 실행 환경 (A), (B)에서의 실행 예를 소개합니다. 실행 환경 (A)에서 실험한 결과, Worker Thread 패턴을 사용한 경우에는 〈그림 8-2〉와 같이 되고, 사용하지 않은 경우에는 〈그림 8-3〉처럼 되었습니다. WorkerThread 패턴을 사용한 경우에는 409683개(=135956+136879+136848)의 리퀘스트가 처리되었고, 사용하지 않은 경우에는 37799개(12186+13002+12611)의 리퀘스트가 처리되었습니다. 이 실행 예에서는 쓰루풋이 무려 10배 이상 증가했습니다.

그림 8-2 실행 환경 (A)에서의 실행 예 (Worker Thread 패턴을 사용한 경우)

```
(전략)
Worker-0 executes [ Request from Alice No.136876 ]
Worker-4 executes [ Request from Chris No.136845 ]
Worker-2 executes [ Request from Bobby No.135954 ]
Worker-0 executes [ Request from Alice No.136877 ]
Worker-4 executes [ Request from Chris No.136846 ]
Worker-2 executes [ Request from Bobby No.135955 ]← Bobby의 리퀘스트는
0~135955까지 135956개 처리되었다
Worker-3 executes [ Request from Alice No.136878 ] ← Alice의 리퀘스트는
0~136878까지 136879개 처리되었다
Worker-0 executes [ Request from Chris No.136847 ] ← Chris의 리퀘스트는
0~136847까지 136848개 처리되었다
```

그림 8-3 실행 환경 (A)에서의 실행 예 (Worker Thread 패턴을 사용하지 않고 새로운 쓰레드를 매번 기동한 경우)

```
(전략)
Thread-37705 executes [ Request from Alice No.12183
Thread-37706 executes [ Request from Alice No.12184
Thread-37707 executes [ Request from Alice No.12185 ← Alice의 리퀘스트는
0~12185까지 12186개 처리되었다
(중략)
Thread-37783 executes [ Request from Chris No.12999
Thread-37784 executes [ Request from Chris No.13000
Thread-37785 executes [ Request from Chris No.13001← Chris의 리퀘스트는
0~13001까지 13002개 처리되었다
(중략)
Thread-37797 executes [ Request from Bobby No.12608
Thread-37798 executes [ Request from Bobby No.12609
Thread-37799 executes [ Request from Bobby No.12610← Bobby의 리퀘스트는
0~12610까지 12611 개 처리되었다
```

한편 실행 환경 (B)에서 실험한 결과, Worker Thread 패턴을 사용한 경우에는 〈그림 8-4〉와 같이 되고, 사용하지 않은 경우에는 〈그림 8-5〉와 같이 되었습니다. Worker Thread 패턴을 사용한 경우에는 75401개(=22232+23773+24851)의 리퀘스트가 처리되고 사용하지 않은 경우에는 68304개(22232+22980+23092)의 리퀘스트가 처리되었습니다. 이 실행 예에서 쓰루풋은 10% 정도 증가하는데 그쳤습니다.

그림 8-4 실행 환경 (B)에서의 실행 예 (Worker Thread 패턴을 사용한 경우)

```
(전략)
Worker-1 executes [ Request from Bobby No.26776 ] ← Bobby의 리퀘스트는
0~26776까지 26777개 처리되었다
Worker-2 executes [ Request from Alice No.23767 ]
Worker-0 executes [ Request from Chris No.24848 ]
Worker-3 executes [ Request from Chris No.24849 ]
Worker-4 executes [ Request from Alice No.23770 ]
Worker-1 executes [ Request from Alice No.23771 ]
Worker-2 executes [ Request from Alice No.23772 ] ← Alice의 리퀘스트는
0~23772까지 23773개 처리되었다
Worker-0 executes [ Request from Chris No.24850 ] ← Chris의 리퀘스트는
0~24850까지 2485한 개 처리되었다
```

그림 8-5 실행환경(B)에서의 실행예(Worker Thread 패턴을 사용하지 않고 새로운 쓰레드를 매번 기동한 경우)

```
(전략)
Thread-37705   executes[ Request from Alice No.12183
Thread-37706   executes[ Request from Alice No.12184
Thread-37707   executes[ Request from Alice No.12185
                              ←Alice의 리퀘스트는 0~12185까지 12186개 처리되었다
(중략)
Thread-37783   executes[ Request from Chris   No.12999
Thread-37784   executes[ Request from Chris   No.13000
Thread-37785   executes[ Request from Chris   No.13001
                              ←Chris의 리퀘스트는 0~13001까지 13002개 처리되었다
(중략)
Thread-37797   executes[ Request from Bobby   No.12608
Thread-37798   executes[ Request from Bobby   No.12609
Thread-37799   executes[ Request from Bobby   No.12610
                              ←Bobby의 리퀘스트는 0~12610까지 12611개 처리되었다
```

문제 8-4의 해답

생존성을 잃기 때문입니다.

invokeAndWait 메소드를 호출하면 이벤트 큐에 가득 저장되어 있던 이벤트가 모두 처리된 뒤, 인수 Runnable이 실행되고 나서 제어가 돌아옵니다. 그런데 invokeAndWait를 이벤트 디스패칭 쓰레드로부터 호출한다고 하는 것은 이벤트 큐에 저장되어 있던 이벤트의 하나로서 invokeAndWait를 호출하는 것입니다. invokeAndWait에서 돌아왔을 때 비로소 이 이벤트의 처리가 종료됩니다. 그런데 invokeAndWait에서 돌아오려면 우선 이벤트 큐의 내용이 모두 처리되어야만 합니다. 이래서는 끝내 invokeAndWait에서 돌아올 수 없게 되고, 이벤트 디스패칭 쓰레드가 동작할 수 없기에 GUI 애플리케이션의 이벤트 처리도 기능하지 않게 됩니다. 이러한 상황을 막기 위해 Swing에서는 이벤트 디스패칭 쓰레드로부터 invokeAndWait를 호출하면 java.lang.Error가 통보됩니다.

문제 8-5의 해답

그 이유는 Swing의 이벤트 디스패칭 쓰레드 자신이 처리에 약 10초가 걸리는 루프를 돌고 있기 때문입니다. 화면에 표시를 하는 것은 이벤트 디스패칭 쓰레드입니다. 이벤트 디스패칭 쓰레드가 actionPerformed 메소드에서 돌아오지 않으면 화면은 갱신되지 않습니다.

〈리스트 8-5〉는 MyFrame 클래스를 수정한 것입니다. 수정 후 화면은 〈그림 8-6 ~ 그림 8-10〉, 실행 결과는 〈그림 8-11〉처럼 됩니다. 이 문제를 해결하기 위해 다음과 같이 수정했습니다.

◆ 응답성을 높이기 위해 다른 쓰레드에게 일을 맡긴다

countUp 메소드 안에서 invokerThread라고 하는 쓰레드를 기동합니다. 이 쓰레드가 실제 카운트업을 실행합니다. counUp 메소드를 호출한 쓰레드(이벤트 디스패칭 쓰레드)는 invokerThread를 기동하고 곧바로 countUp 메소드에서 돌아옵니다. 이것은 Chapter 07에서 배운 Thread-Per-Message 패턴이군요.

◆ 이벤트 디스패칭 쓰레드에 일을 시키기 위해 invokeLater 메소드를 사용한다

invokeThread 안에서는 0, 1, 2, … 9까지 카운트 업하고 sleep 메소드에서 약 1초 쉽니다. invokeThread는 카운트 업한 숫자를 직접 setText하지 않습니다. invokerThread는

이베트 디스패칭 쓰레드가 아니기 때문입니다. 「숫자를 setText한다」고 하는 업무를 이벤트 디스패칭 쓰레드에게 시키기 위해 다음과 같은 문을 사용합니다.

```
SwingUtilities.invokeLater(executor);
```

인수 executor는 Runnable 객체(Runnable을 구현한 익명 내부클래스의 인스턴스)로 다음 문을 이벤트 디스패칭 쓰레드에게 시키기 위한 것입니다.

```
lebel.setText(string);
```

꽤 복잡하게 보이지만 하나하나 천천히 따라가다 보면 이해할 수 있으리라 생각합니다.

리스트 8-5 수정 후 MyFrame 클래스 (MyFrame.java)

```
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.SwingUtilities;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class MyFrame extends JFrame implements ActionListener {
    private final JLabel label=new JLabel("Event Dispatching Thread Sample");
    private final JButton button = new JButton("countUp");
    public MyFrame() {
        super("MyFrame");
        getContentPane().setLayout(new FlowLayout());
        getContentPane().add(label);
        getContentPane().add(button);
        button.addActionListener(this);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pack();
        setVisible(true);
    }
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == button) {
            countUp();
        }
    }
    private void countUp() {
        System.out.println(Thread.currentThread().getName()+":countUp:BEGIN");

        // invokerThread는 지정시간 sleep한 후 SwingUtilities.invokeLater를 호출한다
```

```
        new Thread("invokerThread") {
            public void run() {
                System.out.println(Thread.currentThread().getName() +
                        ":invokerThread:BEGIN");
                for (int i = 0; i < 10; i++) {
                    final String string = "" + i;
                    try {
                        // executor는 이벤트 디스패칭 쓰레드에서 호출된다
                        final Runnable executor = new Runnable() {
                            public void run() {
                                System.out.println
(Thread.currentThread().getName() + ":executor:BEGIN:string = " +
string);
                                label.setText(string);
                                System.out.println
(Thread.currentThread().getName() + ":executor:END");
                            }
                        };

                        // executor를 이벤트 디스패칭 쓰레드로 호출하게 한다
                        SwingUtilities.invokeLater(executor);

                        Thread.sleep(1000);
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
                System.out.println(Thread.currentThread().getName() +
":invokerThread:END");
            }
        }.start();

        System.out.println(Thread.currentThread().getName()+":countUp:END");
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/W●rkerThread/A8-5*

그림 8-6 기동 직후의 화면

그림 8-7 [countUp] 버튼을 누린 직후의 화면

그림 8-8 약 1초 후의 화면

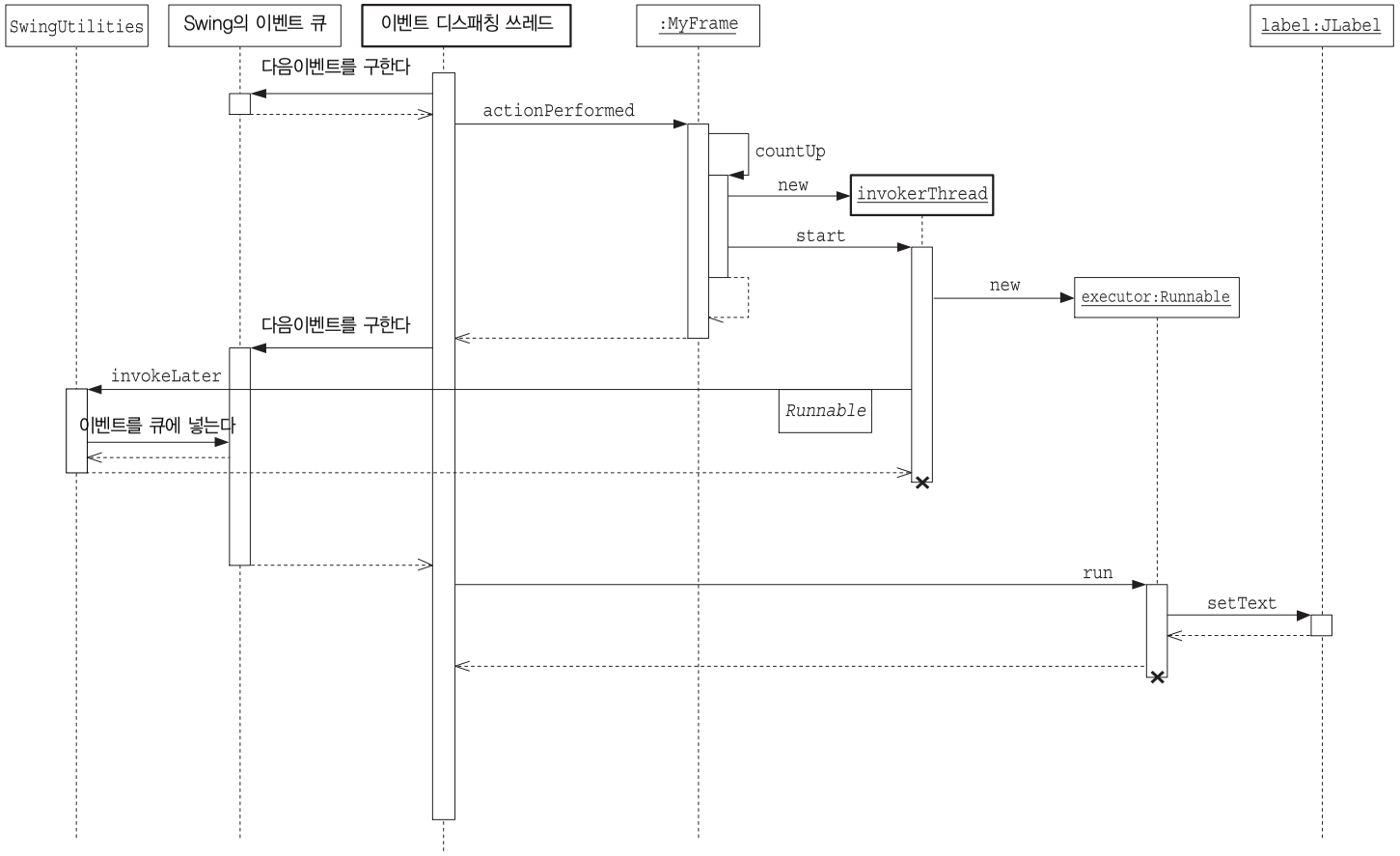
그림 8-9 약 2초 후의 화면

그림 8-10 가장 마지막 화면

그림 8-11 실행 결과

```
main:BEGIN
main:END
AWT-EventQueue-0:countUp:BEGIN
AWT-EventQueue-0:countUp:END
invokerThread:invokerThread:BEGIN
AWT-EventQueue-0:executor:BEGIN:string=   0
AWT-EventQueue-0:executor:END
AWT-EventQueue-0:executor:BEGIN:string=   1
AWT-EventQueue-0:executor:END
AWT-EventQueue-0:executor:BEGIN:string=   2
AWT-EventQueue-0:executor:END
AWT-EventQueue-0:executor:BEGIN:string=   3
AWT-EventQueue-0:executor:END
AWT-EventQueue-0:executor:BEGIN:string=   4
AWT-EventQueue-0:executor:END
AWT-EventQueue-0:executor:BEGIN:string=   5
AWT-EventQueue-0:executor:END
AWT-EventQueue-0:executor:BEGIN:string=   6
AWT-EventQueue-0:executor:END
AWT-EventQueue-0:executor:BEGIN:string=   7
AWT-EventQueue-0:executor:END
AWT-EventQueue-0:executor:BEGIN:string=   8
AWT-EventQueue-0:executor:END
AWT-EventQueue-0:executor:BEGIN:string=   9
invokerThread:invokerThread:END
```

그림 8-12 문제 8-5의 시퀀스 다이어그램



문제 8-6의 해답

〈리스트 8-6~리스트 8-8〉처럼 됩니다. 또한 실행 예는 〈그림 8-13〉과 같습니다.

우선 Thread 클래스의 stop 메소드는 사용하지 않습니다. stop 메소드는 락을 가지고 있는 쓰레드라 하더라도 갑작스럽게 종료시켜 버리기 때문에 안전성을 지킬 수 없습니다. 이 부분은 앞에서 이미 설명했습니다.

Channel 클래스의 putRequest 메소드와 takeRequest 메소드는 예외 InterruptedException을 통보하도록 변경합니다. 그리고 stopAllWorkers 메소드는 자신이 보관하고 있는 WorkerThread 클래스의 stopThread 메소드를 호출하도록 구현합니다.

각 쓰레드(ClientThread, WorkerThread)의 stopThread에서는 **terminated 파일을 true로 하여 interrupt 메소드를 호출합니다.** terminated 필드는 종료 처리가 이뤄지는지를 아닌지를 나타내며, interrupt 메소드는 지정한 쓰레드(여기에서는 자기 자신)에 인터럽트를 겁니다. 이 방법에 대해서는 Two-Phase Termination 패턴(Chapter 10)에서 자세히 설명하겠습니다.

한편 보강 2의 〈리스트 8-6~리스트 8-8〉에서처럼 java.util.concurrent.ThreadPoolExecutor을 사용하여 쓰레드풀을 구성하면 워커 쓰레드를 정지시키는 것은 **shutdown 메소드**를 호출하는 것만으로 충분합니다. 그 경우 ChlientThread 쪽에는 실행 시 예외 RejectedExecutionException이 통보됩니다.

리스트 8-6 수정한 이후의 Channel 클래스 (Channel.java)

```
public final class Channel {
    private static final int MAX_REQUEST = 100;
    private final Request[] requestQueue;
    private int tail;  // 다음에 putRequest 할 장소
    private int head;  // 다음에 takeRequest 할 장소
    private int count; // Request의 수

    private final WorkerThread[] threadPool;

    public Channel(int threads) {
        this.requestQueue = new Request[MAX_REQUEST];
        this.head = 0;
        this.tail = 0;
        this.count = 0;

        threadPool = new WorkerThread[threads];
```

```
        for (int i = 0; i < threadPool.length; i++) {
            threadPool[i] = new WorkerThread("Worker-" + i, this);
        }
    }
    public void startWorkers() {
        for (int i = 0; i < threadPool.length; i++) {
            threadPool[i].start();
        }
    }
    public void stopAllWorkers() {
        for (int i = 0; i < threadPool.length; i++) {
            threadPool[i].stopThread();
        }
    }
    public synchronized void putRequest(Request request) throws
InterruptedException {
        while (count >= requestQueue.length) {
            wait();
        }
        requestQueue[tail] = request;
        tail = (tail + 1) % requestQueue.length;
        count++;
        notifyAll();
    }
    public synchronized Request takeRequest() throws InterruptedException {
        while (count <= 0) {
            wait();
        }
        Request request = requestQueue[head];
        head = (head + 1) % requestQueue.length;
        count--;
        notifyAll();
        return request;
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/W●rkerThread/A8-6*

리스트 8-7 수정한 이후의 W●rkerThread 클래스 (W●rkerThread.java)

```
public class WorkerThread extends Thread {
    private final Channel channel;
    private volatile boolean terminated = false;
    public WorkerThread(String name, Channel channel) {
        super(name);
        this.channel = channel;
    }
```

```
    public void run() {
        try {
            while (!terminated) {
                try {
                    Request request = channel.takeRequest();
                    request.execute();
                } catch (InterruptedException e) {
                    terminated = true;
                }
            }
        } finally {
            System.out.println(Thread.currentThread().getName() + " is
terminated.");
        }
    }
    public void stopThread() {
        terminated = true;
        interrupt();
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/W●rkerThread/A8-6*

리스트 8-8 수정 후 ClientThread 클래스 (ClientThread.java)

```
import java.util.Random;

public class ClientThread extends Thread {
    private final Channel channel;
    private static final Random random = new Random();
    private volatile boolean terminated = false;
    public ClientThread(String name, Channel channel) {
        super(name);
        this.channel = channel;
    }
    public void run() {
        try {
            for (int i = 0; !terminated; i++) {
                try {
                    Request request = new Request(getName(), i);
                    channel.putRequest(request);
                    Thread.sleep(random.nextInt(1000));
                } catch (InterruptedException e) {
                    terminated = true;
```

```
                }
            }
        } finally {
            System.out.println(Thread.currentThread().getName() + " is
terminated.");
        }
    }
    public void stopThread() {
        terminated = true;
        interrupt();
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/W●rkerThread/A8-6*

그림 8-13 실행 예

```
(전략)
Worker-4 executes [ Request from Alice No.10 ]
Worker-0 executes [ Request from Alice No.11 ]
Worker-1 executes [ Request from Alice No.12 ]
Worker-4 executes [ Request from Bobby No.9 ]
Worker-0 executes [ Request from Chirs No.8 ]
Worker-2 executes [ Request from Bobby No.10 ]
Worker-3 executes [ Request from Chirs No.9 ]
Worker-4 executes [ Request from Alice No.13 ]
Worker-0 executes [ Request from Chirs No.10 ]
Worker-1 executes [ Request from Alice No.14 ]
Worker-3 executes [ Request from Bobby No.11 ]
Worker-0 is terminated.
Worker-1 is terminated.
Worker-2 is terminated.
Worker-3 is terminated.
Worker-4 is terminated.
Alice is terminated.
Bobby is terminated.
Chris is terminated.
```