Chapter 07

문제 7-1의 해답

○… (1) request 메소드를 호출할 때마다 새로운 쓰레드가 기동한다.

➜ request 메소드 안에서 매번 새로운 쓰레드를 기동시키고 있습니다.

×… (2) request 메소드를 호출할 때마다 Helper 클래스의 인스턴스가 생성된다.

➜ Helper 클래스의 인스턴스가 생성되는 것은 Host 클래스의 인스턴스가 생성될 때 뿐입니다.

×… (3) request 메소드에서 돌아오지 않으면 handle 메소드는 호출되지 않는다.

➜ request 메소드 안에서 새로운 쓰레드를 기동하고, 그 쓰레드가 handle 메소드를 호출합니다. 이것은 최초의 쓰레드가 request 메소드에서 돌아오는지 아닌지와 무관합니다.

×… (4) handle 메소드를 호출하고 문자를 표시하고 있는 것은 메인 쓰레드이다.

➜ handle 메소드를를 호출하고 있는 것은 메인 쓰레드가 아니라 request 메소드 안에서 새롭게 기동된 쓰레드입니다.

×… (5) slowly 메소드 안에서 sleep하는 시간을 길게 늘리면 request 메소드로부터 돌아오는 시간도 길어진다.

➜ slowly 메소드 안에서 sleep하는 시간을 늘려도 request 메소드에 돌아올 때까지의 시간은 변함이 없습니다. request 메소드를 실행하는 쓰레드는 slowly 메소드를 호출하지 않기 때문입니다.

문제 7-2의 해답

Host 클래스를 〈리스트 7-1〉처럼 변경합니다. request 메소드 안에서 직접 handle 메소드를 호출하도록 만들었습니다. Main 클래스, Helper 클래스는 변경할 필요가 없습니다.

리스트 7-1 변경한 이후의 Host 클래스 (Host.java)

```
public class Host {
    private final Helper helper = new Helper();
    public void request(final int count, final char c) {
        System.out.println("request(" + count + ", " + c + ") BEGIN");
        helper.handle(count, c);
        System.out.println("request(" + count + ", " + c + ") END");
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/ThreadPerMessage/A7-2*

실행 결과는 〈그림 7-1〉과 같습니다. 이 실행 결과는 타이밍에 따라 달라지거나 하지 않습니다. 한편 본문에서 소개한 예제 프로그램의 실행 예(그림 7-2)는 타이밍에 따라 달라집니다.

한 개의 request가 끝나고 나서 다음 request가 시작되기 때문에 A, B, C 문자표시가 섞이지 않으며 모든 request가 끝나고 나서 메인 쓰레드가 종료되는 것을 알 수 있습니다.

그림 7-1 실행 결과

```
mainBEGIN
    request (10,   A) BEGIN
          handle (10,   A) BEGIN
AAAAAAAAAA
          handle (10,   A) END
    request (10,   A) END
    request (20,   B) BEGIN      ← request가 종료하고나서 다음 request가 개시
          handle (20,   B) BEGIN
BBBBBBBBBBBBBBBBBBBB
          handle (20,   B) END
    request (20,   B) END
    request (30,   C) BEGIN      ← request가 종료하고나서 다음 request가 개시
          handle (30,   C) BEGIN
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
          handle (30,   C) END
    request (30,   C) END
mainEND                          ← 모든 request가 종료하고나서 메인쓰레드가 종료한다.
```

문제 7-3의 해답

실행 결과는 〈그림 7-2〉와 같습니다. 문제 7-2의 싱글 쓰레드에 의한 실행 결과(그림 7-1)와 똑같군요. 그 이유는 〈리스트 7-14〉가 쓰레드 기동 부분에서 start 메소드 대신 run 메소드를 호출하고 있기 때문입니다. run 메소드를 호출해도 새로운 쓰레드가 기동하지 않기 때문에 메인 쓰레드가 익명 내부 클래스의 run 메소드를 실행합니다. 이 경우 응답성은 좋아지지 않습니다.

그림 7-2 실행 결과

```
main    BEGIN
    request (10,   A) BEGIN
           handle (10,   A) BEGIN
AAAAAAAAAA
           handle   (10,   A) END
    request (10,   A) END
    request (20,   B) BEGIN
           handle   (20,   B) BEGIN
BBBBBBBBBBBBBBBBBBBB
           handle   (20,   B) END
    request (20,   B) END
    request (30,   C) BEGIN
           handle   (30,   C) BEGIN
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
           handle   (30,   C) END
    request (30,   C) END
main    END
```

문제 7-4의 해답

2개 해답 예를 소개합니다.

◆ 해답 1 : HelperThread 클래스를 탑레벨 클래스로서 선언

Host 클래스, HelperThread 클래스를 각각 〈리스트 7-2〉와 〈리스트 7-3〉처럼 작성합니다. Main 클래스, Helper 클래스는 예제 프로그램 그대로 사용해도 상관없습니다.

요구에 포함되어 있는 인수를 쓰레드에 건네는 것은 소스 코드 측면에서 익명 내부 클래스를 사용한 경우보다 번거롭습니다. 건네는 정보를 필드의 형으로 정리해야 하기 때문입

니다. 단, 익명 내부 클래스에 익숙하지 않은 사람에게 있어서는 소스의 가독성이 높을 수도 있습니다.

리스트 7-2 해답 1 : Host 클래스 (host.java)

```
public class Host {
    private Helper helper = new Helper();
    public void request(int count, char c) {
        System.out.println("request(" + count + ", " + c + ") BEGIN");
        new HelperThread(helper, count, c).start();
        System.out.println("request(" + count + ", " + c + ") END");
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/ThreadPerMessage/A7-4a*

리스트 7-3 해답 : HelperThread 클래스 (HelperThread.java)

```
public class HelperThread extends Thread {
    private final Helper helper;
    private final int count;
    private final char c;
    public HelperThread(Helper helper, int count, char c) {
        this.helper = helper;
        this.count = count;
        this.c = c;
    }
    public void run() {
        helper.handle(count, c);
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/ThreadPerMessage/A7-4a*

해답 2 : HelperThread를 익명이 아닌 내부 클래스로 선언

〈리스트 7-4〉처럼 하면 Helper 클래스와 HelperThread 클래스를 Host 클래스 안에서 선언할 수 있습니다. 이렇게 하면 Helper와 HelperThread 2개의 클래스가 Host 클래스와 밀접한 관계에 있음을 표현할 수 있습니다. 단, Host 클래스의 선언이 길어지기 때문에 읽기가 불편하다는 단점도 있습니다.

HelperThread 인스턴스를 만들 때 생성자의 인수에 helper를 건네지 않아도 됩니다. Host의 helper 필드는 내부 클래스의 HelperThread 인스턴스로부터 자유롭게 액세스할

수 있기 때문입니다.

Main 클래스는 예제 프로그램과 동일합니다.

리스트 7-4 해답 2 : Host 클래스 (Host.java)

```
public class Host {
    private final Helper helper = new Helper();
    public void request(int count, char c) {
        System.out.println("request(" + count + ", " + c + ") BEGIN");
        new HelperThread(count, c).start();
        System.out.println("request(" + count + ", " + c + ") END");
    }

    // Inner class
    private class Helper {
        public void handle(int count, char c) {
            System.out.println("   handle(" + count + ", " + c + ") BEGIN");
            for (int i = 0; i < count; i++) {
                slowly();
                System.out.print(c);
            }
            System.out.println("");
            System.out.println("   handle(" + count + ", " + c + ") END");
        }
        private void slowly() {
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
            }
        }
    }

    // Inner class
    private class HelperThread extends Thread {
        private final int count;
        private final char c;
        public HelperThread(int count, char c) {
            this.count = count;
            this.c = c;
        }
        public void run() {
            helper.handle(count, c);
        }
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/ThreadPerMessage/A7-4*

문제 7-5의 해답

해답으로 4가지 예를 소개하겠습니다. 이 4개 해답은 유저가 버튼을 연속해서 눌렀을 때마다 동작이 모두 다릅니다.

◆ 해답 1 : Thread-Per-Message 패턴을 사용한 경우

service 메소드에서 새로운 쓰레드를 기동하고 그 새로운 쓰레드가 doService라고 하는 메소드를 호출하도록 만듭니다. 그리고 doService로 실제 처리를 실행합니다. 이로써 service 메소드에서 바로 돌아올 수 있습니다. 이 해답에서는 유저가 버튼을 연속으로 눌렀을 때 여러 쓰레드가 동시에 doService를 실행합니다(그림 7-3).

리스트 7-5 해답 1 : 버튼을 연속적으로 누르면 그 때마다 doService를 실행하는 Service 클래스 (Service.java)

```
public class Service {
    public static void service() {
        new Thread() {
            public void run() {
                doService();
            }
        }.start();
    }
    private static void doService() {
        System.out.print("service");
        for (int i = 0; i < 50; i++) {
            System.out.print(".");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
            }
        }
        System.out.println("done.");
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/ThreadPerMessage/A7-5a*

그림 7-3 해답 1 : 버튼을 3회 눌렀을 때의 실행 예(3개 쓰레드가 섞여서 출력된다)

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

◆ 해답 2 : Thread-Per-Message 패턴과 Single Threaded Execution 패턴을 사용한 경우

또 하나의 해답을 〈리스트 7-6〉에 제시합니다. 이 프로그램에서는 service 메소드에서 쓰레드가 바로 돌아옵니다. 게다가 유저가 버튼을 연속해서 눌러도 doService 메소드를 동시에 실행하는 것이 한 개의 쓰레드임을 보증합니다.

이 해답은 유저가 버튼을 연속해서 눌렀을 경우 누른 횟수만큼 doService가 실행되지만 섞여서 출력되지는 않습니다(그림 7-4).

리스트 7-6 해답 2 : 버튼을 연속해서 눌러도 섞여서 출력되지 않는 Service 클래스 (Service.java)

```
public class Service {
    public static void service() {
        new Thread() {
            public void run() {
                doService();
            }
        }.start();
    }
    private static synchronized void doService() {
        System.out.print("service");
        for (int i = 0; i < 50; i++) {
            System.out.print(".");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
            }
        }
        System.out.println("done.");
    }
}
```

⇒ 예제파일 경로 : 부록CD/src/ThreadPerMessage/A7-5

그림 7-4 해답 2 : 버튼을 3회 눌렀을 때의 실행 예(3개 쓰레드가 순서대로 출력된다)

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

해답 3 : Thread-Per-Message 패턴과 Balking 패턴을 사용한 경우

〈리스트 7-7〉은 sevice 메소드에서 바로 돌아오고, 게다가 유저가 버튼을 연속해서 누른 경우에는 doService 메소드를 실행하는 것이 한 개의 쓰레드뿐임을 보증합니다. 여기에서는 Balking 패턴(Chapter 04)을 사용하여 동시에 doService 메소드를 실행하려고 한 쓰레드를 balk합니다(그림 7-5).

리스트 7-7 해답 3 : 버튼을 연속해서 눌렀을 때에는 balk하는 Service 클래스 (Service.java)

```
public class Service {
    private static volatile boolean working = false;
    public static synchronized void service() {
        System.out.print("service");
        if (working) {
            System.out.println(" is balked.");
            return;
        }
        working = true;
        new Thread() {
            public void run() {
                doService();
            }
        }.start();
    }
    private static void doService() {
        try {
            for (int i = 0; i < 50; i++) {
                System.out.print(".");
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                }
            }
            System.out.println("done.");
        } finally {
            working = false;
        }
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/ThreadPerMessage/A7-5c*

그림 7-5 해답 3 : 버튼을 3회 눌렀을 때의 실행 예(2회째와 3회째가 balk하고 있다)

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

해답 4 : 연속적으로 눌렀을 때 실행 중인 처리를 취소하는 경우

〈리스트 7-8〉은 service 메소드에서 바로 돌아오고, 또한 유저가 버튼을 연속해서 누른 경우에는 doService 메소드의 실행을 취소합니다(그림 7-6).

여기에서는 interrupt를 사용하여 처리를 취소하고 있습니다. 쓰레드의 종료에 대해서는 Two-Phase Termination 패턴(Chapter 10)도 참고해 주세요.

리스트 7-8 해답 4 : 버튼을 연속해서 눌렀을 때 실행 중인 처리를 취소하는 Service 클래스 (Service.java)

```
public class Service {
    private static Thread worker = null;
    public static synchronized void service() {
        // 실행 중인 처리가 있다면 interrupt를 사용하여 취소한다
        if (worker != null && worker.isAlive()) {
            worker.interrupt();
            try {
                worker.join();
            } catch (InterruptedException e) {
            }
            worker = null;
        }
        System.out.print("service");
        worker = new Thread() {
            public void run() {
                doService();
            }
        };
        worker.start();
    }
    private static void doService() {
        try {
            for (int i = 0; i < 50; i++) {
                System.out.print(".");
                Thread.sleep(100);
            }
            System.out.println("done.");
        } catch (InterruptedException e) {
            System.out.println("cancelled.");
        }
    }
}
```

⇒ 예제파일 경로 : 부록CD/src/ThreadPerMessage/A7-5d

그림 7-6 해답 4 : 버튼을 3회 눌렀을 때의 실행 예(1회째와 2회째가 취소되었다)

```
service.................cancelled.
service......cancelled.
service.....................................................done.
```

문제 7-6의 해답

〈리스트 7-19〉의 MainServer 클래스를 변경하고, Main 클래스, Service 클래스는 수정하지 않습니다.

해답 1 : java.lang.Thread를 사용한 MiniServer 클래스

리스트 7-9 java.lang.Thread를 사용한 MiniServer 클래스 (MiniServer.java)

```
import java.net.Socket;
import java.net.ServerSocket;
import java.io.IOException;

public class MiniServer {
    private final int portnumber;
    public MiniServer(int portnumber) {
        this.portnumber = portnumber;
    }
    public void execute() throws IOException {
        ServerSocket serverSocket = new ServerSocket(portnumber);
        System.out.println("Listening on " + serverSocket);
        try {
            while (true) {
                System.out.println("Accepting...");
                final Socket clientSocket = serverSocket.accept();
                System.out.println("Connected to " + clientSocket);
                new Thread() {
                    public void run() {
                        try {
                            Service.service(clientSocket);
                        } catch (IOException e) {
                            e.printStackTrace();
                        }
                    }
                }.start();
            }
        } catch (IOException e) {
```

```
            e.printStackTrace();
        } finally {
            serverSocket.close();
        }
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/ThreadPerMessage/A7-6a*

웹 브라우저에서 액세스하면 명령어 프롬프트에는 〈그림 7-7〉과 같이 표시됩니다.

그림 7-7 해답 1의 실행 예

```
Listening on ServerSocket[addr=0.0.0.0/0.0.0.0,port=0,localport=8888]
Accepting…
Connected to Socket[addr=/127.0.0.1,port=1546,localport=8888]
Accepting…
Thread-0:
Service.service(Socket[addr=/127.0.0.1,port=1546,localport=8888]) BEGIN
Thread-0: Countdown i =10 ← 1초마다 카운트다운 한다
Thread-0: Countdown i =9
Thread-0: Countdown i =8
Thread-0: Countdown i =7
Thread-0: Countdown i =6
Thread-0: Countdown i =5
Thread-0: Countdown i =4
Thread-0: Countdown i =3
Thread-0: Countdown i =2
Thread-0: Countdown i =1
Thread-0: Countdown i =0
Thread-0:
Service.service(Socket[addr=/127.0.0.1,port=1546,localport=8888]) END
(이하 생략. CTRL+C로 종료)
```

◆ 해답 2 : java.util.concurrent.ExecutorService를 사용한 MiniServer 클래스

리스트 7-10 java.util.concurrent.ExecutorService를 사용한 MiniServer 클래스 (MiniServer.java)

```
import java.net.Socket;
import java.net.ServerSocket;
import java.io.IOException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
```

```
public class MiniServer {
    private final int portnumber;
    public MiniServer(int portnumber) {
        this.portnumber = portnumber;
    }
    public void execute() throws IOException {
        ServerSocket serverSocket = new ServerSocket(portnumber);
        ExecutorService executorService =
                        Executors.newCachedThreadPool();
        System.out.println("Listening on " + serverSocket);
        try {
            while (true) {
                System.out.println("Accepting...");
                final Socket clientSocket = serverSocket.accept();
                System.out.println("Connected to " + clientSocket);
                executorService.execute(
                    new Runnable() {
                        public void run() {
                            try {
                                Service.service(clientSocket);
                            } catch (IOException e) {
                                e.printStackTrace();
                            }
                        }
                    }
                );
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            executorService.shutdown();
            serverSocket.close();
        }
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/ThreadPerMessage/A7-6*

웹 브라우저에서 액세스하면 명령어 프롬프트에는 〈그림 7-8〉과 같이 표시됩니다.

그림 7-8 해답 2의 실행 예

```
Listening on ServerSocket[addr=0.0.0.0/0.0.0.0,port=0, localport=8888]
Accepting…
Connected to Socket[addr=/127.0.0.1, port=1547, localport=8888]
Accepting…
pool-1-thread-1: Service.service(Socket[addr=/127.0.0.1, port=1547,
localport=8888]) BEGIN
pool-1-thread-1:Countdown i =10 ← 1초마다 카운트다운 한다
pool-1-thread-1:Countdown i =9
pool-1-thread-1:Countdown i =8
pool-1-thread-1:Countdown i =7
pool-1-thread-1:Countdown i =6
pool-1-thread-1:Countdown i =5
pool-1-thread-1:Countdown i =4
pool-1-thread-1:Countdown i =3
pool-1-thread-1:Countdown i =2
pool-1-thread-1:Countdown i =1
pool-1-thread-1:Countdown i =0
pool-1-thread-1:Service.service(Socket[addr=/127.0.0.1, port=1547,
localport=8888]) END
(이하 생략. CTRL+C로 종료)
```

문제 7-7의 해답

예를 들면 〈리스트 7-11〉처럼 됩니다(이 밖에도 방법은 있겠지요).

이 〈리스트 7-11〉에 도달하게된 과정을 순서대로 설명하겠습니다. 기대되는 실행 결과(그림 7-10)에서는 다음과 같은 사실을 알 수 있습니다.

[1] Step 2가 표시되어 있으므로 magic 메소드는 예외를 통보하지 않는다.

[2] Step3이나 END가 표시되지 않았으므로 쓰레드는 enter 메소드에서 돌아오지 않는다.

[3] Step3이 표시되지 않았다는 것은 쓰레드가 obj의 락을 취하지 않고 블록하고 있음을 의미한다.

여기까지 이해했다면 magic 메소드가 할 일이 다음 [4]라고 하는 사실을 알 수 있을 것입니다.

[4] magic 메소드의 일은 인수 obj의 락을 취하는 것이다.

그렇다고 락을 취하기 위해 synchronized 블록을 사용하는 건 소용이 없습니다. Magic 메소드에서 돌아올 때 synchronized 블록에서 나와버리면 락은 결국 해제되기 때문입니다. 이러한 사실을 통해 다음 [5]와 [6]의 처리를 실행하면 되겠다는 생각이 들 겁니다.

[5] magic 메소드 안에서 새로운 쓰레드를 기동하고 그 쓰레드가 obj의 락을 취하도록 한다.

[6] 새로운 쓰레드는 영원히 obj의 락을 취한 상태로 둔다.

[7] 새로운 쓰레드가 기동하여 obj의 락을 취할 때까지 원래의 쓰레드는 magic 메소드에서 돌아오면 안된다.

〈리스트 7-11〉은 [1]~[7]을 바탕으로 하여 만들었습니다. 내부 클래스(inner class)의 메소드 run에 인수 obj를 건네기 위해 인수 obj를 final로서 선언하고 있습니다.

또한 [7]을 만족시키기 위해 Guarded Suspension 패턴(Chapter 03)을 사용했습니다. 원래의 쓰레드는 thread의 이름이 ""이 아니게 될 때까지 thread 상에서 wait 합니다. 한편 새로운 쓰레드는 obj의 락을 취한 후 이름을 "Locked"로 바꾸고 thread(이것은 this의 값과 같다) 상에서 wait하고 있는 원래의 쓰레드를 notifyAll 합니다.

여기에서는 notifyAll을 사용했지만 thread 상에서 wait하고 있는 쓰레드는 한 개이므로 notify를 사용해도 상관없습니다.

리스트 7-11 완성된 Blackhole 클래스 (1) (Blackhole.java)

```
public class Blackhole {
    public static void enter(Object obj) {
        System.out.println("Step 1");
        magic(obj);
        System.out.println("Step 2");
        synchronized (obj) {
            System.out.println("Step 3 (never reached here)");
                                        // 여기에는 오지 않는다
        }
    }
    public static void magic(final Object obj) {
        // thread는 obj의 락을 취하고 무한 루프하는 쓰레드
        // thread의 이름을 가드 조건으로 사용한다
        Thread thread = new Thread() {      // inner class
            public void run() {
                synchronized (obj) {        // 여기에서 obj의 락을 취한다
                    synchronized (this) {
                        this.setName("Locked");     // 가드 조건의 변화
                        this.notifyAll();           // obj의 락을 취했음을 통보
                    }
                    while (true) {
                        // 무한 루프
                    }
                }
            }
        };
```

```
        synchronized (thread) {
            thread.setName("");
            thread.start();       // 쓰레드의 기동
            // Guarded Suspension 패턴
            while (thread.getName().equals("")) {
                try {
                    thread.wait();  // 새로운 쓰레드가 obj의 락을 취하기를 기다린다
                } catch (InterruptedException e) {
                }
            }
        }
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/ThreadPerMessage/A7-7a*

〈리스트 7-12〉와 같은 해석도 가능합니다. 여기에서도 notifyAll 대신 notify를 사용할 수 있습니다.

리스트 7-12 완성된 Blackhole 클래스 (1) (Blackhole.java)

```
public class Blackhole {
    public static void enter(Object obj) {
        System.out.println("Step 1");
        magic(obj);
        System.out.println("Step 2");
        synchronized (obj) {
            System.out.println("Step 3 (never reached here)");
                                    // 여기에는 오지 않는다
        }
    }
    public static void magic(final Object obj) {
        // thread는 obj의 락을 취하고 나서 자기자신의 종료를 영원히 기다리는 쓰레드
        Thread thread = new Thread() {
            public void run() {
                synchronized (obj) {        // 여기에서 obj의 락을 취한다
                    synchronized (this) {
                        this.notifyAll();    // obj의 락을 취한 사실을 통지
                    }
                    try {
                        this.join();        // 영원히 기다리게 된다
                    } catch (InterruptedException e) {
                    }
                }
            }
        };
```

```
        synchronized (thread) {
            thread.start();  // 쓰레드의 기동
            try {
                thread.wait();  // 새로운 쓰레드가 obj의 락을 취하기를 기다린다
            } catch (InterruptedException e) {
            }
        }
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/ThreadPerMessage/A7-7*

문제 7-8의 해답

예를 들어 〈그림 7-9〉처럼 표시됩니다.

그림 7-9 실행 예 1

```
MainThread:main:BEGIN
MainThread:execute:BEGIN
MainThread:newThread:BEGIN
MainThread:newThread:END
MainThread:execute:END
MainThread:main:END
QuizThread:run:BEGIN
QuizThread:Hello!
QuizThread:run:END
```

"QuizThread:"로 시작되는 다음 3행의 출력은 다른 행의 출력 앞이나 뒤에 오는 경우가 있습니다.

QuizThread:run:BEGIN

QuizThread:Hello!

QuizThread:run:END

단, "QuizThread:"로 시작되는 출력은 MainThread:newThread:End행 보다도 뒤에 옵니다(실행 예 2 참조).

〈그림 7-10〉는 메인 쓰레드가 끝나기 전에 QuizThread가 작동하기 시작한 경우입니다. 타이밍상으로는 거의 어려운일이지만, execute 메소드가 끝나기 전에 QuizThread가

움직이기 시작하는 경우도 사양 상에서는 있을 수 있습니다.

그림 7-10 실행 예 2

```
MainThread:main:BEGIN
MainThread:execute:BEGIN
MainThread:newThread:BEGIN
MainThread:newThread:END
MainThread:execute:END
QuizThrea:run:BEGIN
QuizThrea:Hello!
QuizThrea:run:END
MainThread:main:END
```

〈리스트 7-13〉은 문제 7-8의 〈리스트 7-23〉과 결과가 똑같이 나오는 프로그램입니다. 여기에서는 ThreadFactory, Executor, Runnable 등 3개 익명 내부 클래스의 인스턴스를 만들고 제일 마지막에 execute합니다. 역시나 이 쪽이 알기가 쉽군요. 변수 threadFactory가 final로 되어 있는 것은 Executor 객체의 execute 메소드 안에서 참조하고 있기 때문입니다.

리스트 7-13 리스트 7-23(문제7-8)과 결과가 같은 프로그램 (Main.java)

```
import java.util.concurrent.*;

class Log {
    public static void println(String s) {
        System.out.println(Thread.currentThread().getName() + ":" + s);
    }
}

public class Main {
    public static void main(String[] args) {
        Thread.currentThread().setName("MainThread");
        Log.println("main:BEGIN");

        // ThreadFactory
        final ThreadFactory threadFactory = new ThreadFactory() {
            public Thread newThread(Runnable r) {
                Log.println("newThread:BEGIN");
                Thread t = new Thread(r, "QuizThread");
                Log.println("newThread:END");
                return t;
            }
```

```
        };

        // Executor
        Executor executor = new Executor() {
            public void execute(Runnable r) {
                Log.println("execute:BEGIN");
                threadFactory.newThread(r).start();
                Log.println("execute:END");
            }
        };

        // Runnable
        Runnable runnable = new Runnable() {
            public void run() {
                Log.println("run:BEGIN");
                Log.println("Hello!");
                Log.println("run:END");
            }
        };

        // Executor에 Runnable을 건네고 실행
        executor.execute(runnable);

        Log.println("main:END");
    }
}
```

⇒ 예제파일 경로 : 부록CD/src/ThreadPerMessage/A7-8