Chapter 01

문제 1-1의 해답

크리티컬 섹션을 길게 만들면 에러 검출 가능성을 높일 수 있습니다. 예를 들어 pass 메소드 안의 「name에 대입」과 「address에 대입」 사이에 sleep 메소드의 호출을 넣을 수 있습니다. 〈리스트 1-1〉에서는 쓰레드를 약 1000밀리 초 동안 정지시키고 있습니다.

리스트 1-1 쓰레드 세이프가 아님을 일찍 검출한 Gate 클래스 (Gate.java)

```
public class Gate {
    private int counter = 0;
    private String name = "Nobody";
    private String address = "Nowhere";
    public void pass(String name, String address) {
        this.counter++;
        this.name = name;
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        }
        this.address = address;
        check();
    }
    public String toString() {
        return "No." + counter + ": " + name + ", " + address;
    }
    private void check() {
        if (name.charAt(0) != address.charAt(0)) {
            System.out.println("***** BROKEN ***** " + toString());
        }
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/Introduction1/AI1-4*

실행 예는 〈그림 1-1〉과 같습니다. 이 예에서는 counter 3의 단계에서 이미 에러가 검출되었습니다.

그림 1-1 실행 예

```
Testing Gate, hit CTRL+C to exit
Alice BEGIN
Bobby BEGIN
Chris BEGIN
***** BROKEM ***** No.3 : Chris, Alaska
***** BROKEM ***** No.4 : Alice, Brazil
***** BROKEM ***** No.5 : Bobby, Canada
***** BROKEM ***** No.6 : Chris, Alaska
***** BROKEM ***** No.7 : Alice, Brazil
***** BROKEM ***** No.8 : Bobby, Canada
***** BROKEM ***** No.9 : Chris, Alaska
***** BROKEM ***** No.10 : Alice, Brazil
(이하 생략. CTRL+C로 종료)
```

잠깐! 한 마디 : Thread.yield

크리디컬 섹션 안에서 Thread 클래스의 yield 메소드를 호출하여 쓰레드의 변환을 재촉하는 방법도 있습니다.

문제 1-2의 해답

필드를 private로 하는 이유는 프로그래머가 클래스의 안전성을 쉽게 확인할 수 있도록 하기 위해서입니다. private이 붙은 필드는 그 클래스에서만 액세스할 수 있습니다. 따라서 그 클래스 안에 선언되어 있는 메소드가 필드에 안전하게 액세스하고 있는지를 확인하는 것만으로 필드의 안전성을 확인할 수 있습니다. 다른 클래스는 확인할 필요가 없습니다.

protected가 붙은 필드는 그 클래스의 서브 클래스나 같은 패키지 안의 클래스에서도 액세스할 수 있습니다. 따라서 안전성을 확인하려면 서브 클래스나 같은 패키지 안의 클래스까지도 체크해야 합니다.

public이 붙은 필드는 임의의 클래스에서 액세스할 수 있습니다. 따라서 안전성을 확인하기 위해서는 이 필드에 액세스하는 모든 클래스를 체크해야 합니다.

문제 1-3의 해답

〈리스트 1-4〉의 Gate 클래스가 다른 프로그램에서 이용되었을 경우를 생각해 봅시다.

```
public String toString() {
        return "No." + counter + ": " + name + ", " + address;
}
```

UserThread 클래스의 쓰레드가 pass 메소드를 통과하는 도중에 다른 쓰레드가 toString 메소드를 호출했다고 합시다. 쓰레드 x가 name 필드의 값을 참조하고 나서 address 필드의 값을 참조할 때까지 그 사이에 UserThread의 쓰레드가 address의 값을 바꿔버릴지도 모릅니다. 그러면 toString 메소드는 쓰레드 x에 대하여 name과 address의 머리글자가 일치하지 않는 값으로 문자열을 구성할 가능성이 있습니다.

예제 프로그램의 범위에서는 toString을 synchronized 메소드로 하지 않아도 안전성에 문제가 없습니다만, **일반적으로 복수의 쓰레드에서 공유하는 필드는 synchronized(또는 volatile)로 보호해야 합니다.**

문제 1-4의 해답

○… (1) Point 클래스의 서브 클래스는 만들 수 없다.

➜ Point 클래스는 final로 선언되어 있기 때문에 서브 클래스를 만들 수 없습니다.

○… (2) Point 클래스의 x필드에 대입하는 문을 Point 클래스 밖에 선언되어 있는 클래스에는 쓸 수 없다.

➜ x 필드는 private로 선언되어 있기 때문에 Point 클래스 밖에 선언된 클래스에는 대입할 수 없습니다.

○… (3) Point 클래스의 어떤 인스턴스와 관련하여 여기에서 읽은 move 메소드를 실행할 수 있는 것은 한 번에 한 개의 쓰레드이다.

×… (4) Point 클래스는 복수의 쓰레드에서 이용해도 안전하다.

➜ move 메소드만 사용한다면 안전합니다. 하지만 Point 클래스의 미처 못읽은 부분에 다음과 같은 필드에 대입할 수 있는 메소드가 있을지도 모르므로 「안전하다」고 단언할 수는 없습니다.

```
public synchronized void setX(int x) {
    this.x = x;
}
public synchronized void setX(int y) {
    this.x = y;
}
```

×… (5) Point 클래스의 나머지 메소드를 전부 읽으면 데드락 발생 여부를 판단할 수 있다.

➜ Point 클래스의 나머지 부분을 읽고서도 데드락 발생 여부를 판단할 수 없는 경우가 있습니다. Point 클래스를 이용하는 다른 클래스의 기능에 따라 데드락은 발생하기도 하고 발생하지 않기도 하기 때문입니다.

잠깐! 한 마디 : 프로그래머가 의식해야 할 범위의 크기

「소스 코드의 일부분만을 읽는다」고 하는 이번 문제의 의미에 대해 생각해 봅시다.

액세스 제어는 클래스의 일부분만 읽고서도 판단이 가능합니다. 그러나 **안전성**은 클래스의 일부만 읽어서는 판단할 수가 없습니다. 그 뿐 아니라 **안전성**은 클래스 전체를 읽어도 알 수 없는 경우가 있습니다.

여기에서도 멀티 쓰레드 프로그래밍의 어려움을 짐작할 수 있습니다. 프로그래머가 의식해야 할 범위가 통상적인 프로그래밍보다 넓어지기 때문입니다.

문제 1-5의 해답

아니요, 안전하지 않습니다. 안전하게 만들려면 enter, exit, getCounter의 모든 메소드를 synchronized 메소드로 해야 합니다.

counter 필드의 값을 늘리고 있는 다음 문에 대해 생각해 봅시다.

```
counter++;
```

이것은 한 개의 문이지만 쓰레드가 이것을 배타적으로(다른 쓰레드의 간섭을 가드하면서) 실행할 수는 없습니다. 쓰레드는 counter 필드의 값을 조사하여 한 개 늘리고, counter 필드에 대입한다고 하는 복수의 처리를 실행합니다.

쓰레드 A에 의한 enter의 메소드 실행과 쓰레드 B에 의한 exit 메소드의 실행이 병행하여 이뤄질 때 Single Threaded Execution 패턴을 사용하지 않으면 다음과 같은 순서로 진행될 가능성이 있습니다. 이 경우 쓰레드 B가 exit 메소드를 실행한 효과를 잃을 수 있습니다.

그림 1-2 쓰레드 B의 exit 효과를 잃게 되는 예

쓰레드A (enter)	쓰레드B (exit)	counter 값
		100
counter의 값을 조사한다		100
1 늘린다		100
<<<여기에서 쓰레드B로 변환>>>		
	counter 값을 조사한다	100
	1 줄인다	100
	counter에 대입한다	99
<<<여기에서 쓰레드A로 변환>>>		
counter에 대입한다		101

여기에서 SecurityGate 클래스의 안전성을 직접 테스트하는 프로그램을 소개하고 싶지만 아쉽게도 〈그림 1-2〉와 같은 상황을 테스트로 만들기는 어렵습니다.

그래서 차선책으로서 SecurityGate 클래스를 수정해 counter가 증감할 때 Thread.yield 메소드를 호출하도록 했습니다(리스트 1-2).

리스트 1-2 counter 값이 변경되기 전후에 쓰레드 변환이 일어나기 쉽게 수정한 SecurityGate 클래스 (SecurityGate.java)

```
public class SecurityGate {
    private int counter = 0;
    public void enter() {
        int currentCounter = counter;
        Thread.yield();
        counter = currentCounter + 1;
    }
    public void exit() {
        int currentCounter = counter;
        Thread.yield();
        counter = currentCounter - 1;
    }
    public int getCounter() {
        return counter;
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/SingleThreadExecution/A1-5*

이렇게 수정한 다음 〈리스트 1-3〉에 제시한 프로그램을 실행하면 안전하지 않음을 검출할 수 있습니다. 이 프로그램에서는 「CrackerThread 클래스(리스트A1-4)의 인스턴스를 5개 만들고 10번씩 enter과 exit를 실행」하는 처리를 반복합니다. enter와 exit이 같은 횟수만큼 호출되고 있기 때문에 가장 바깥쪽의 for 루프가 돌 때마다 getCounter()의 값이 0으로 돌아오는 것입니다. 그런데 실행 예(그림 1-3)처럼 0이 되지 않는 경우가 있습니다.

아래 프로그램에서는 join 메소드를 사용하여 쓰레드의 종료를 체크합니다.

리스트 1-3 보안문을 체크하는 프로그램 (Main.java)

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Testing SecurityGate...");
        for (int trial = 0; true; trial++) {
            SecurityGate gate = new SecurityGate();
            CrackerThread[] t = new CrackerThread[5];

            // CrackerThread 기동
            for (int i = 0; i < t.length; i++) {
                t[i] = new CrackerThread(gate);
                t[i].start();
            }

            // CrackerThread 종료 대기
            for (int i = 0; i < t.length; i++) {
                try {
                    t[i].join();
                } catch (InterruptedException e) {
                }
            }

            // 확인
            if (gate.getCounter() == 0) {
                // 모순되지 않는다
                System.out.print(".");
            } else {
                // 모순을 발견했다
                System.out.println("SecurityGate is NOT safe!");
                System.out.println("getCounter() == " + gate.getCounter());
                System.out.println("trial = " + trial);
                break;
            }
        }
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/SingleThreadExecution/A1-5*

리스트 1-4 enter와 exit를 10번씩 실행하는 쓰레드를 나타내는 클래스 (CrackerThread.java)

```
public class CrackerThread extends Thread {
    private final SecurityGate gate;
    public CrackerThread(SecurityGate gate) {
        this.gate = gate;
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            gate.enter();
            gate.exit();
        }
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/SingleThreadExecution/A1-5*

그림 1-3 실행 예

```
Testing SecurityGate…
…… SecurityGate is NOT safe!
getCounter() == 1    ← 0이 되지 않는다
trial = 44           ← 44번째 루프에서 검출
```

이제 enter 메소드와 exit 메소드를 synchronized로 하는 이유는 이해했으리라 생각합니다. 그럼 getCounter를 synchronized로 하는 이유는 무엇일까요? getCounter 메소드를 synchronized 메소드로 하든지, counter 필드를 값을 volatile로 하지 않으면, 다른 쓰레드가 counter 필드를 값을 변경한다 하더라도, 그것은 어디까지나 각각의 쓰레드가 가지고 있는 캐시에 대한 조작일 뿐, 그것이 꼭 공유 메모리의 내용에 반영되는 것은 아니기 때문입니다. 캐시와 공유 메모리에 대해서는 부록 A 「Java의 메모리 모델」을 참조해 주십시오.

문제 1-6의 해답

다음 2가지 방법을 해답으로 소개합니다(다른 방법들도 있겠지요).

◆ 방법 1 : Alice와 Bobby가 같은 순서로 식기를 집는 방법

가장 단순한 방법은 Alice와 Bobby가 같은 순서로 식기를 집도록 하는 것입니다. 〈리스트 1-5〉의 메인 클래스에서는 Alice나 Bobby 모두 '스푼 → 포크' 순으로 집습니다. 이로써 데드락은 일어나지 않게 됩니다. 여기에서는 Chapter 01 본문에서 이야기한「SharedResource

역할이 대칭」이라고 하는 조건을 지키지 못했습니다. 이 방법에서 Tool 클래스와 EaterThread 클래스를 수정할 필요는 없습니다.

리스트 1-5 방법 1로 수정한 Main 클래스 (Main.java)

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Testing EaterThread, hit CTRL+C to exit.");
        Tool spoon = new Tool("Spoon");
        Tool fork = new Tool("Fork");
        new EaterThread("Alice", spoon, fork).start();
        new EaterThread("Bobby", spoon, fork).start();
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/SingleThreadExecution/A1-6a*

그림 1-4 방법 1의 실행 예

```
(전략)
Alice takes up [ Spoon ] (left).
Alice takes up [ Fork ] (right).
Alice is eating now, yam yam!
Alice puts down [ Fork ] (right).
Alice puts down [Spoon] (left).
Alice takes up [ Spoon ] (left).
Alice takes up [ Fork ] (right).
Alice is eating now, yam yam!
Alice puts down [ Fork ] (right).
Alice puts down [Spoon] (left).
Alice takes up [ Spoon ] (left).
Alice takes up [ Fork ] (right).
(이하 생략. CTRL+C로 종료)
```

◆ 방법 2 : 「스푼과 포크」조를 짜는 방법

스푼 또는 포크의 락을 개별적으로 취하지 않고 「스푼/포크 세트」를 표현하는 전혀 다른 객체를 준비합니다. 그리고 인스턴스의 락을 취하는 방식으로 「스푼/포크 세트」의 락을 취하면 데드락은 발생하지 않습니다. 이것은 본문에서 이야기한 「복수의 SharedResource 역할이 있다」고 하는 조건을 깨뜨린 것입니다.

수정 후 프로그램은 〈리스트 1-6 ~ 리스트 1-8〉과 같습니다.

리스트 1-6 방법 2로 수정한 Main 클래스 (Main.java)

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Testing EaterThread, hit CTRL+C to exit.");
        Tool spoon = new Tool("Spoon");
        Tool fork = new Tool("Fork");
        Pair pair = new Pair(spoon, fork);
        new EaterThread("Alice", pair).start();
        new EaterThread("Bobby", pair).start();
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/SingleThreadExecution/A1-6*

리스트 1-7 방법 2로 수정한 EaterThread 클래스 (EaterThread.java)

```
public class EaterThread extends Thread {
    private String name;
    private final Pair pair;
    public EaterThread(String name, Pair pair) {
        this.name = name;
        this.pair = pair;
    }
    public void run() {
        while (true) {
            eat();
        }
    }
    public void eat() {
        synchronized (pair) {
            System.out.println(name + " takes up " + pair + ".");
            System.out.println(name + " is eating now, yum yum!");
            System.out.println(name + " puts down " + pair + ".");
        }
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/SingleThreadExecution/A1-6*

리스트 1-8 새로 작성한 Pair 클래스 (Pair.java)

```
public class Pair {
    private final Tool lefthand;
    private final Tool righthand;
    public Pair(Tool lefthand, Tool righthand) {
        this.lefthand = lefthand;
        this.righthand = righthand;
    }
    public String toString() {
        return "[ " + lefthand + " and " + righthand + " ]";
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/SingleThreadExecution/A1-6*

그림 1-5 방법 2의 실행 예

```
Alice takes up [ [ Spoon ] and [ Fokr] ].
Alice is eating now, yam yam!
Alice puts down [ [ Spoon] and [Fork] ].
Alice takes up [ [ Spoon ] and [ Fokr] ].
Alice is eating now, yam yam!
Alice puts down [ [ Spoon] and [Fork] ].
Bobby takes up [ [ Spoon ] and [ Fokr] ].
Bobby is eating now, yam yam!
Bobby puts down [ [ Spoon] and [Fork] ].
Alice takes up [ [ Spoon ] and [ Fokr] ].
Alice is eating now, yam yam!
Alice puts down [ [ Spoon] and [Fork] ].
(이하 생략. CTRL+C로 종료)
```

문제 1-7의 해답

다양한 구현 방법이 있겠으나 여기에서는 3가지 예를 소개합니다.

◆ 해답 예 1 : 간단한 Mutex 클래스

〈리스트 1-9〉는 가장 간단한 Mutex 클래스입니다. 여기에서는 busy(바쁘다)라고 하는 boolean형 필드를 사용하고 있습니다. busy가 true면 lock 메소드를 실행했음을 나타내고, busy가 false면 unlock 메소드를 실행했음을 나타냅니다.

lock와 unlock은 모두 synchronized 메소드로서 busy 필드를 보호합니다.

리스트 1-9 간단한 Mutex 클래스 (Mutex.java)

```
public final class Mutex {
    private boolean busy = false;
    public synchronized void lock() {
        while (busy) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        busy = true;
    }
    public synchronized void unlock() {
        busy = false;
        notifyAll();
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/SingleThreadExecution/A1-7a*

〈리스트 1-9〉의 Mutex 클래스는 문제 1-7의 프로그램과 관련해서는 바르게 동작합니다. 하지만 다음과 같은 제약이 있다는 점에 주의해야 합니다.

[제약 1 : 재입장이 불가능하다]

… 어떤 쓰레드가 lock 메소드를 2번 연속해서 호출했다고 합시다. 그러면 2번째 호출에서는 busy 필드가 true이기 때문에 wait하게 됩니다. 자신이 열쇠로 잠궜기 때문에 들어갈 수 없는거죠. 이러한 상황을 일컬어 〈리스트 1-9〉의 Mutex 클래스는 「재입장 할 수 없다」 혹은 「reentrant가 아니다」라고 합니다.

[제약 2 : 누구나 unlock할 수 있다]

… 〈리스트 1-9〉의 Mutex 클래스에서는 lock 메소드를 호출하지 않는 메소드라도 unlock 메소드를 호출할 수 있습니다. 즉, 자신이 사용한 열쇠가 아니더라도 열 수 있는 상황이 되어 버리는 것입니다.

◆ 해답 예 2 : 개량 Mutex 클래스

〈리스트 1-10〉은 해답 예 1의 제약을 없앤 Mutex 클래스입니다. 여기에서는 현재 락의 수를 locks 필드에 기록하고 있습니다. 이 락의 수는 lock 호출 횟수에서 unlock 호출 횟수를 뺀 것입니다. 또한 lock 메소드를 호출한 쓰레드를 owner(소유자) 필드에 기록하고 있습니다. locks 필드와 owner 필드를 사용하여 앞의 제약을 없애고 있습니다.

리스트 1-10 개량 Mutex 클래스 (Mutex.java)

```
public final class Mutex {
    private long locks = 0;
    private Thread owner = null;
    public synchronized void lock() {
        Thread me = Thread.currentThread();
        while (locks > 0 && owner != me) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        assert locks == 0 || owner == me;
        owner = me;
        locks++;
    }
    public synchronized void unlock() {
        Thread me = Thread.currentThread();
        if (locks == 0 || owner != me) {
            return;
        }
        assert locks > 0 && owner == me;
        locks--;
        if (locks == 0) {
            owner = null;
            notifyAll();
        }
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/SingleThreadExecution/A1-7*

잠깐! 한 마디 : assert

〈리스트 1-10〉에 어서션(assertion)이 2번 등장합니다.
lock 메소드 안의

```
assert locks ==  | | owner == me;
```

는 「lock의 수가 0이다. 또는 자신이 lock 했다」고 표명하는 어서션이며,
unlock 메소드 안의

```
assert locks > 0 && owner == me;
```

는 「lock의 수가 0보다 크고, 자신이 lock 했다」고 표명하는 어서션입니다.
이처럼 assert를 사용하여 「해당 부분에 성립되어 있는 절대 조건을 명시적으로 기술」할 수 있습니다. 만일 그 조건이 성립되지 않으면 프로그래머가 기대하지 않았던 이상 사태가 일어납니다.

디폴트에서는 어서션의 조건이 성립되지 않아도 아무런 이상이 발생하지 않습니다. 하지만 Java 명령어에 -ea 옵션을 붙여 실행할 때 조건이 성립되지 않으면 예외 java.lang.AssertionError가 통보됩니다.
assert는 Java SE 1.4부터 도입된 기능입니다. 자세한 내용은 다음을 참고하세요.

:: Assertion Facility : http://java.sun.com/j2se/1.4/docs/guide/lang/assert.html
:: Assertion 기능(일본어 번역) : http://www.shudo.net/article/assert/assert.html

해답 예 3 : java.util.concurrent.locks.ReentrantLock을 사용한 Mutex 클래스

Java SE 5.0 표준 라이브러리에 도입된 java.util.concurrent.locks.ReentrantLock 클래스는 이 문제에서 다루고 있는 개량 Mutex의 기능을 모두 포함하고 있습니다. lock, unlock이라고 하는 메소드 이름도 똑같습니다. 따라서 〈리스트 1-11〉처럼 ReentrantLock 클래스의 빈 서브 클래스를 만드는 것만으로 원하는 Mutex를 구할 수 있습니다.

리스트 1-11 ReentrantLock 클래스를 사용한 Mutex 클래스 (Mutex.java)

```
import java.util.concurrent.locks.ReentrantLock;

public class Mutex extends ReentrantLock {
}
```

⇒ 예제파일 경로 : 부록CD/src/SingleThreadExecution/A1-7c