Chapter 02

문제 2-1의 해답

○… (1) java.lang.String 클래스는 immutable한 클래스이다.

×… (2) java.lang.StringBuffer 클래스는 immutable한 클래스이다.

○… (3) final 선언된 필드는 2번 대입되는 일이 없다.

×… (4) private 필드는 그 클래스와 서브 클래스에서 직접 액세스할 가능성이 있다.

➜ private 필드에 직접 액세스할 수 있는 것은 그 필드를 선언하고 있는 클래스 본체에서 액세스하는 경우 뿐입니다.

×… (5) 메소드를 synchronized해도 무방하므로 가능한 synchronized를 붙여서 사용해야 한다.

➜ 클래스의 생존성이나 수행 능력을 떨어뜨릴 위험이 있기 때문에 무조건 붙인다고 좋은 것은 아닙니다.

문제 2-2의 해답

분명 문제에서 제시한 프로그램을 실행하면 CAT라고 표시됩니다. 그렇다고 변수 s가 보관하고 있는 인스턴스의 내용이 바뀐 것은 아닙니다. replace 메소드는 문자열 안에 포함되어 있는 문자를 치환한 **또 다른 인스턴스를 새로 만들고** 그것을 반환 값으로 사용합니다. 단, replace 메소드에 건네는 치환하기 이전의 문자와 치환 후 문자를 나타내는 인수의 값이 똑같을 때(즉, 실질적으로 치환을 하지 않을 때)에는 새로운 인스턴스를 만들지 않고 원래의 인스턴스(this)가 반환 값이 됩니다.

다른 인스턴스가 만들어지고 있는 것을 확인하는 프로그램을 〈리스트 2-1〉에 소개합니다. 실행 결과는 〈그림 2-1〉과 같습니다.

리스트 2-1 replace 메소드에서 또 다른 인스턴스가 만들어지고 있음을 확인하는 프로그램 (Main.java)

```
public class Main {
    public static void main(String[] args) {
        String s = "BAT";
        String t = s.replace('B', 'C'); // 'B'를 'C'로 치환
        System.out.println("s = " + s); // replace를 실행한 후의 s
        System.out.println("t = " + t); // replace의 반환 값 t
        if (s == t) {
            System.out.println("s == t");
        } else {
            System.out.println("s != t");
        }
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/Immutable/A2-2*

그림 2-1 실행 결과

```
s = BAT ← replace를 실행 한 후에도 s의 값은 "BAT"
t = CAT ← replace의 반환 값은 "CAT"
s ! = t ← 양쪽의 인스턴스는 다르다
```

문제 2-3의 해답

필자의 환경에서는 〈그림 2-2〉와 같은 결과가 나왔습니다. synchronized를 사용한 쪽이 Immutable 패턴을 사용한 쪽보다 약 2배나 더 시간이 걸립니다. 여기에서는 쓰레드의 충돌이 전혀 일어나지 않는 상태에서 계측하기 때문에 인스턴스의 락을 취하는 처리와 락을 해제하는 처리에 걸리는 시간을 계산합니다.

그림 2-2 실행 예

```
NotSynch: BEGIN
NotSynch: END
Elapsed time = 38896msec.  ← synchronized가 없는 경우에 걸린 시간
Synch: BEGIN
Synch: END
Elapsed time = 76400msec.  ← synchronized가 있는 경우에 걸린 시간
```

단, 여기에서는 처리가 거의 없는 메소드를 연속적으로 호출하고 있다는 사실에 주의해야 합니다. 그 결과 전체 처리시간에서 synchronized에 의한 시간 증가분이 차지하는 비율

이 일반적인 프로그램에서 보다 크게 나타났을 가능성이 있습니다.

Java 컴파일러 최적화의 차이나 Java 실행환경의 차이 등에 의해 시간차의 비율이 크게 달라지므로 이 프로그램의 실행 예는 참고로만 해 주세요.

문제 2-4의 해답

getInfo 메소드의 반환 값(StringBuffer의 인스턴스)을 변경하면 info 필드가 참조하고 있는 인스턴스의 내용이 바뀌어 버리기 때문입니다.

실제로 UserInfo 인스턴스의 내용을 바꾸는 프로그램을 작성해 봅시다(리스트 2-2).

리스트 2-2 UserInf●의 상태를 바꾸는 Main 클래스 (Main.java)

```
public class Main {
    public static void main(String[] args) {
        // 인스턴스 작성
        UserInfo userinfo = new UserInfo("Alice", "Alaska");

        // 표시
        System.out.println("userinfo = " + userinfo);

        // 상태를 변경
        StringBuffer info = userinfo.getInfo();
        info.replace(12, 17, "Bobby");  // 12 이상 17미만이 "Alice"의 위치

        // 다시 표시
        System.out.println("userinfo = " + userinfo);
    }
}
```

⇒ 예제파일 경로 : 부록CD/src/Immutable/A2-4

여기에서는 제일 처음 Alice(Alaska 출신)의 정보를 만들어 표시합니다. 그러나 그 다음 getInfo 메소드에서 구한 StringBuffer의 인스턴스(info)를 이용하여 "Alice"를 "Bobby"로 수정합니다. 그리고나서 다시 표시해보면 UserInfo의 인스턴스가 바뀐 것을 알 수 있습니다(그림 2-3).

그림 2-3 실행 결과

```
userinfo = [ UserInfo: <info name="Alice" address="Alaska" /> ]
userinfo = [ UserInfo: <info name="Bobby" address="Alaska" /> ] ← 이름이 Bobby
                                                                로 바뀌었다
```

getInfo 메소드로 구할 수 있는 info 필드가 가지고 있는 인스턴스는 String의 인스턴스가 아니라 StringBuffer의 인스턴스입니다. **StringBuffer 클래스는 String 클래스와 달리 내부상태를 변경하는 메소드를 가지고 있기 때문에** info 필드의 내용을 외부에서 수정하는 것이 가능합니다.

String 클래스의 replace 메소드는 인스턴스 자신을 변경하지 않지만 StringBuffer 클래스의 replace 메소드는 인스턴스 자신을 변경합니다. 즉, StringBuffer 클래스는 mutable인 것입니다.

info 필드에는 final이 선언되어 있기 때문에 info 필드의 값 자체(어느 인스턴스를 가리키고 있는가)는 달라지지 않습니다. 그러나 info 필드가 가리키고 있는 곳에 있는 인스턴스의 상태는 변할 가능성이 있습니다.

문제 2-5의 해답

Line 클래스는 immutable이 아닙니다.

우선 Line 클래스가 사용하고 있는 Point 클래스부터 살펴보겠습니다. Point 클래스의 x 필드나 y 필드가 public으로 되어 있는데다 final이 선언되어 있지 않으므로 이들 필드 값은 자유롭게 수정할 수 있습니다. 따라서 Point 클래스는 mutable한 클래스입니다.

다음으로 Line 클래스의 2번째 생성자를 살펴봅시다.

```
public Line(Point startPoint, Point endPoint) {
        this.startPoint = startPoint;
        this.endPoint = endPoint;
    }
```

생성자의 인수로 주어진 인스턴스가 그대로 필드에 대입되고 있습니다. 이것이 무엇을 의미하는지 잘 생각해 봅시다. 인수로 주어진 Point의 인스턴스는 **생성자의 외부에서 만들어 진 것**이므로 이 생성자를 호출한 누군가가 이 인스턴스에 대한 참조를 가지고 있을 가능

성이 있습니다. 앞의 대입문에서 Line 클래스의 필드도 같은 인스턴스에 대한 참조를 가지게 됩니다.

분명, this.startPoint는 final이며, 더 이상 이 필드의 값 자체는 변하지 않습니다. 그러나 이 필드가 가리키고 있는 Point 인스턴스의 내부 상태는 변화할 수 있습니다.

이 사실을 확인할 수 있는 프로그램을 〈리스트 2-3〉에서 소개합니다.

리스트 2-3 Line 상태를 변화시키는 Main 클래스 (Main.java)

```
public class Main {
    public static void main(String[] args) {
        // 인스턴스 작성
        Point p1 = new Point(0, 0);
        Point p2 = new Point(100, 0);
        Line line = new Line(p1, p2);

        // 표시
        System.out.println("line = " + line);

        // 상태를 변경
        p1.x = 150;
        p2.x = 150;
        p2.y = 250;

        // 다시 표시
        System.out.println("line = " + line);
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/Immutable/A2-5a*

그림 2-4 실행 결과

```
line = [ Line: (0,0) - (100,0) ]
line = [ Line: (150,0) - (150, 250) ]  ← 직선이 이동했다
```

p1, p2 라고 하는 Point의 인스턴스에 손을 댔는데 〈그림 2-4〉를 보니 Line의 인스턴스 내용이 바뀌었군요.

Point라고 하는 immutable이 아닌 클래스(mutable한 클래스)의 영향으로 Line 클래스까지 mutable이 되어 버린 것입니다.

Line 클래스를 immutable로 하는 방법

Line 클래스를 immutable로 만들려면 어떻게 해야 할까요? 물론 Point 클래스를 immutable로 하면 Line 클래스도 immutable이 됩니다. 예를 들어 Point의 x 필드와 y 필드를 public final로 하면 되겠지요.

그런데 Point가 immutable인지 mutable인지와 관계 없이 Line을 immutable로 만드는 방법이 있습니다. 〈리스트 2-4〉가 그것입니다.

리스트 2-4 immutable로 만든 Line 클래스 (Line.java)

```
public class Line {
    private final Point startPoint;
    private final Point endPoint;
    public Line(int startx, int starty, int endx, int endy) {
        this.startPoint = new Point(startx, starty);
        this.endPoint = new Point(endx, endy);
    }
    public Line(Point startPoint, Point endPoint) {
        this.startPoint = new Point(startPoint.x, startPoint.y);
        this.endPoint = new Point(endPoint.x, endPoint.y);
    }
    public int getStartX() { return startPoint.x; }
    public int getStartY() { return startPoint.y; }
    public int getEndX() { return endPoint.x; }
    public int getEndY() { return endPoint.y; }
    public String toString() {
        return "[ Line: " + startPoint + "-" + endPoint + " ]";
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/Immutable/A2-5*

〈리스트 2-4〉에서는 인수로 주어진 인스턴스를 그대로 사용하지 않고 그 인스턴스와 내용이 같은 새로운 인스턴스를 만들어 필드에 대입하고 있습니다.

immutable로 만든 〈리스트 2-4〉의 Line 클래스를 〈리스트 2-3〉을 사용해 움직여 보면, 〈그림 2-5〉와 같이 됩니다. 확실히 Line이 외부의 영향을 받지 않는군요.

그림 2-5 실행 결과

```
line = [ Line: (0,0) - (100,0) ]
line = [ Line: (0,0) - (100,0) ] ← 바뀌지 않았다
```

문제 2-6의 해답

ImmutablePerson 클래스에는 오류가 있습니다.

〈리스트 2-16〉 ImmutablePerson 클래스가 안전성이 결여되어 있음을 〈리스트 2-5〉에서 알 수 있습니다.

여기에서는 MutablePerson의 인스턴스를 메인 쓰레드에서 연속적으로 수정하고 있습니다. 그 때 체크하기 쉽게 name 필드와 address 필드의 문자열(숫자열)을 일치시켰습니다. 한편 CrakerThread 클래스에서는 주어진 MutablePerson 인스턴스로부터 차례차례 새로운 ImmutablePerson의 인스턴스를 만들게 했습니다. 이렇게 만들어진 ImmutablePerson의 인스턴스 name과 address 값이 항상 똑같은지 조사합니다.

리스트 2-5 ImmutablePerson 클래스가 안전성이 결여되고 있음을 실증하는 Main 클래스 (Main.java)

```
import person.MutablePerson;
import person.ImmutablePerson;

public class Main {
    public static void main(String[] args) {
        MutablePerson mutable = new MutablePerson("start", "start");
        new CrackerThread(mutable).start();
        new CrackerThread(mutable).start();
        new CrackerThread(mutable).start();
        for (int i = 0; true; i++) {
            mutable.setPerson("" + i, "" + i);
        }
    }
}

class CrackerThread extends Thread {
    private final MutablePerson mutable;
    public CrackerThread(MutablePerson mutable) {
        this.mutable = mutable;
    }
    public void run() {
        while (true) {
            ImmutablePerson immutable = new ImmutablePerson(mutable);
            if (!immutable.getName().equals(immutable.getAddress())) {
                System.out.println(currentThread().getName() + " *****
                        BROKEN ***** " + immutable);
            }
```

```
        }
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/Immutable/A2-6a*

그 결과 〈그림 2-6〉처럼 BROKEN이 표시됩니다. 안전성 결여가 판명된 것이죠.

그림 2-6 실행 예

```
Thread -1 ***** BROKEN ***** [ ImmutablePerson: 213444, 214742 ]
Thread -2 ***** BROKEN ***** [ ImmutablePerson: 386237, 387624 ]
Thread -1 ***** BROKEN ***** [ ImmutablePerson: 618964, 619060 ]
Thread -2 ***** BROKEN ***** [ ImmutablePerson: 918612, 919875 ]
(이하 생략. CTRL+C로 종료)
```

이제부터 잘못된 부분을 지적하고 수정해 보겠습니다.

MutablePerson을 인수로 취한 ImmutablePerson 클래스의 생성자가 있습니다. 그 안에서 다음과 같이 MutablePerson 클래스의 getName 메소드와 getAddress 메소드를 호출하고 있습니다.

```
this.name = person.getName();
this.address = person.getAddress();
```

그러나 이 두 호출은 크리티컬 섹션 안에 넣어야 합니다. 왜냐하면 첫번째 getName 호출과 그 다음 getAddress 호출 사이에 다른 쓰레드가 MutablePerson의 setPerson 메소드를 사용하여 address 필드를 바꾸어 버릴 수 있기 때문입니다.

여기에서는 생성자이기 때문에 synchronized 블록을 사용하게 되는데, 그 때 락을 취해야 하는 인스턴스가 인수로 주어진 MutablePerson이라는 사실에도 주의합시다.

〈리스트 2-6〉은 수정한 ImmutablePerson 클래스입니다.

리스트 2-6 안전성을 갖춘 ImmutablePerson 클래스 (ImmutablePerson.java)

```
package person;

public final class ImmutablePerson {
    private final String name;
    private final String address;
    public ImmutablePerson(String name, String address) {
        this.name = name;
        this.address = address;
    }
    public ImmutablePerson(MutablePerson person) {
        synchronized (person) {
            this.name = person.getName();
            this.address = person.getAddress();
        }
    }
    public MutablePerson getMutablePerson() {
        return new MutablePerson(this);
    }
    public String getName() {
        return name;
    }
    public String getAddress() {
        return address;
    }
    public String toString() {
        return "[ ImmutablePerson: " + name + ", " + address + " ]";
    }
}
```

⇒ 예제파일 경로 : 부록CD/src/Immutable/A2-6b/person

이것을 방금 전의 Main 클래스에서 실행해도 BROKEN은 표시되지 않습니다.

그림 2-7 실행 예

```
(아무것도 표시되지 않는다. CTRL+C로 종료)
```