# <JSTORM>

## 스윙의 안쪽 탐험

*Revision <1.0>*


중앙대학교 컴퓨터공학과
자바 동호회
JSTORM
http://www.jstorm.pe.kr

## Document Information

| | |
|---|---|
| Document title: | 스윙의 안쪽 탐험 |
| Document file name: | 스윙의 안쪽 탐험 (장대원, 1.0,draft).doc |
| Revision number: | <1.0> |
| Issued by: | <장대원> (mailto: dogjdw@nownuri.net) |
| Issue Date: | <2000/2/28 > |
| Status: | Draft |

## Content Information

| | |
|---|---|
| Audience | 중급 |
| Abstract | 몇몇 스윙 컴포넌트들의 내부를 탐험하고 예제를 살펴봄으로써 스윙의 전반적인 개념과 중요 컴포넌트들의 특성을 이해할 수 있습니다. |
| Reference | http://java.sun.com/products/jfc/tsc/index.html |
| Benchmark information | |

## Document Approvals

| | | |
|---|---|---|
| 고문 | Signature | date |
| 지위 | Signature | date |

*** approval은 본 문서를 검토하고 허락한 사람들을 의미함.**

## Revision History * 이전버전과 달라진 점을 묘사

| Revision | Date | Author | Description of change |
|---|---|---|---|
| | | | |

# Table of Contents

**1. 테이블 파헤치기 .....5**

**2. 스윙 컨테이너의 비밀.....28**

**3. 스윙 텍스트 .....57**

# 스윙의 안쪽 탐험

# 1. 테이블 파헤치기

이 글을 포함해서 세 번의 연재 기사를 통해 스윙의 깊은 곳을 탐험하게 된다. 물론 이번 연재로 스윙의 내부를 모두 파헤치는 것은 불가능하다. 그럼에도 불구하고 필자가 이러한 글을 쓰게된 동기는 독자 여러분들도 필자의 접근 방법과 같은 방법으로 스윙에 접근할 수 있었으면 하는 생각에서이다. 필자가 이번에 연재되는 글들을 통해 주장하는 내용을 한마디로 요약하면 “구조(Architecture)를 보라”일 것이다. 스윙의 유연하고 탄탄한 구조들을 보면서 전문가들의 생각을 훔쳐낼 수 있기 바란다. 사람은 아는 만큼만 생각할 수 있다고 한다. 있는 것을 그냥 사용하는 것과 왜 그런지 어떻게 그런지를 알고 사용하는 것은 많은 차이가 있다. 여러분의 생각을 넓히기 위해 많이 알기 바란다.

첫 번째 희생물(?)로 테이블을 골랐다. 스윙에서 테이블은 매우 복잡한 컴포넌트에 속한다. 그래서 스윙에는 테이블 컴포넌트를 보조하는 여러 클래스(이하 클래스 하면 인터페이스도 포함한다)들이 모여있는 독립된 패키지(javax.swing.table)까지 별도로 있다. 이 글을 통해 필자는 독자들에게 스윙에서의 테이블이 어떠한 구조를 가지고 있으며 또 작동 방식은 어떠한지를 자세히 제시할 것이다.

일단 테이블의 내부 메커니즘을 이해하고 나면 그 응용 범위는 아주 넓어진다. 뭐든지 자신의 입맛에 맞게 뜯어고치려면 내부 사정을 잘 알아야 한다. 여러분은 이 글을 통해 테이블의 내부 구조가 어떠한지를 알아낼 뿐만 아니라 그러한 내부 구조를 활용하여 다양한 뜯어고치기 작업까지도 하게 된다. 이러한 뜯어고치기 작업을 통해 기존의 복잡한 프로그램들이 어떠한 방식으로 작동되며 그리고 또 직접 그러한 프로그램을 개발하려면 어떻게 해야하는 지도 알 수 있다. 예를 들어 여러분이 엑셀을 만든다고 가정하자(우리의 예제에서도 보게되겠지만). 엑셀에는 사용자가 만든 표를 통해 자동으로 그래프를 생성해 주는 기능이 있다. 어떻게 할 수 있을까? 또 엑셀에는 데이터들을 정렬(Sorting)하여 보여주는 기능이 있다. 어떠한 셀들은 자신들의 포맷을 자신들이 스스로 결정하는 듯 하다. 예를 들어 사용자가 금액을 기입하는 셀에 100000이라고 적어주면 셀이 알아서 $100,000으로 표시해 준다. 스윙의 테이블로도 그러한 것들이 쉽게 구현될 수 있다는 것을 독자 여러분들은 곧 알게될 것이다. 그 외의 여러 가지 응용 사례들이 많이 실려있다. 필자가 나름대로 쓸만하다 싶은 예제들을 간추렸기 때문에 이 글에서 제시하는 몇몇 예제들만으로도 아주 실용적이고 유용한 많은 것들을 얻을 수 있으리라 믿는다.

이 글은 스윙 초보자용 글은 아니다. 굳이 초보자가 못 볼 것도 없겠지만 어느 정도 스윙에 대한 감을 잡고 있는 분들을 대상으로 글을 써 나가겠다. (가능하면 한번 정도는 테이블을 사용해 보았길 희망한다) 독자가 스윙에 대해 익숙하지 못하다면 기본적인 컴포넌트들의 사용법과 스윙 컴포넌트들의 작동 메커니즘인 MVC(Model View Controller) 모델에 대해서 만이라도 본지의 다

른 기사나 참고 서적을 참조하여 알아두기 바란다. 그럼 테이블에 대한 탐험을 시작해 보자.

# 1. 테이블의 기본

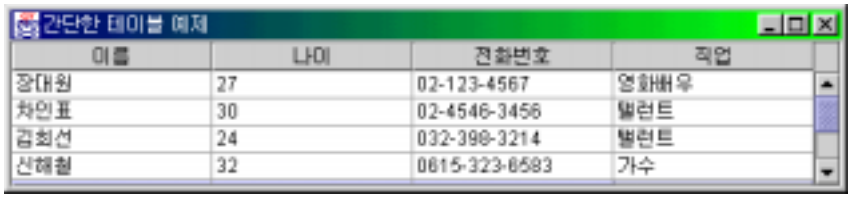
## (1) 테이블의 구조와 구성 요소

테이블을 이해하기 위해 가장 중요한 것은 테이블을 이루고 있는 구성요소들의 특징과 그러한 구성요소들이 어떠한 구조를 이루고 있는지를 먼저 아는 것이다. 말하자면 나무가 아닌 숲을 먼저 보아야 한다는 것이다. 그럼 테이블의 구조를 물리적인 구조와 논리적 구조로 나누어 살펴보자.

### 1) 테이블의 물리적 구조

[화면 1]을 보면 테이블의 맨 위에 테이블 컬럼헤더가 있고 그 아래에 데이터들이 2차원의 격자 형태로 나열되어 있다. 이때 하나의 가로줄은(Row 또는 행) 하나의 레코드를(Record) 형성하고 하나의 세로줄은(Column 또는 열) 각 레코드들의 필드(Field)을 담고있다. 그리고 행과 열의 교차점을 셀(Cell)이라고 한다. 셀은 특정 레코드의 특정 필드를 담고 있다. 다들 아는 내용이지만 혹시 나중에 혼동이 있을지 몰라서 확실히 해둔다.

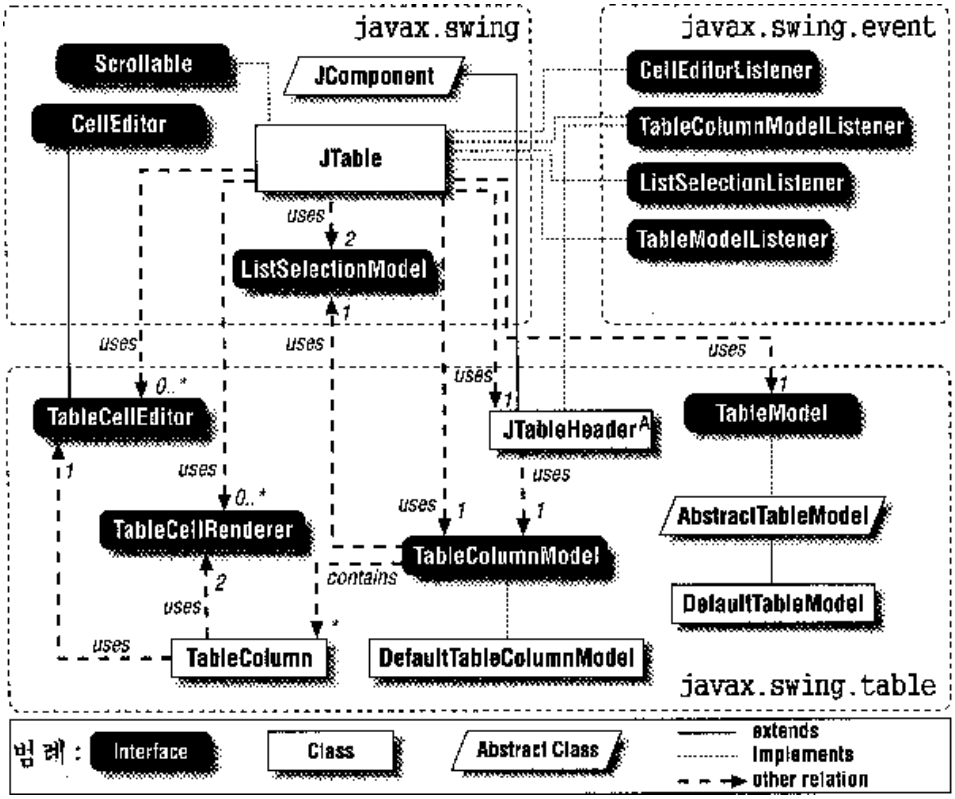
[화면 1] 간단한 테이블 예제




### 2) 테이블의 논리적 구조

테이블의 논리적 구조란 테이블을 이루고 있는 여러 클래스들의 상호관계라 할 수 있다. [그림 1]에 이러한 논리적 구조를 나타내어 보았다. 꽤 복잡해 보일 것이다. 그러나 이 글을 통해 테이블 시스템을 이루는 대부분의(모두 살펴볼 필요는 없다.) 구성 요소들을 자세히 살펴볼 것이니 걱정할 것은 없다. 일단 이 부분에서는 [그림 1]에 대한 개괄적인 이해만으로 만족하고 자세한 내용은 나중에 살펴보도록 한다. 이해의 편의를 위해 [그림 1]을 몇 조각으로 나눠보자. 실타래를 푼다는 생각으로 아래 설명을 읽어주기 바란다.

[그림 1] 스윙 테이블 시스템의 구조(구성)도




맨 먼저 TableModel, AbstractTableModel, DefaultTableModel로 연결된 라인을 살펴보자. 바로 테이블의 모델을(Model) 형성하고 있는 부분이다. 따라서 테이블에 표시될 실제의 데이터에 대한 접근 방법을 이곳에서 제공해야 한다. [표 1]에 이들 구성요소들을 간략히 정리해 보았다.

[표 1] 테이블의 모델을 형성하는 구성 요소들

| 구성 요소 | 설명 |
|---|---|
| TableModel | 테이블의 데이터에 대한 접근 통로를 제공하는 인터페이스다. |
| AbstractTableModel | TableModel의 기본적인 구현을 제공하는 클래스다. 사용자가 직접 모델을 정의해서 사용하려면 이 클래스를 상속받아 사용하는 것이 가장 적당할 것이다. |
| DefaultTableModel | AbstractTableModel를 상속하고 있으며 JTable을 위해 만들어졌다. AbstractTableModel이 구현하지 않는 나머지 메쏘드들도 모두 구현되어 있다. |

그 다음 TableColumn, TableColumnModel, DefaultTableColumnModel으로 연결된 라인을 보자. 이들은 테이블의 열 혹은 열들을 관리하는데 사용된다. 열을 관리하는 작업에는 새로운 열 추가, 기존의 열을 제거, 다른 위치로 열 이동, 열 선택 등이 있을 것이다. 이름에서도 알 수 있듯이 열을 관리하는 부분 자체도 MVC 구조를 띄고 있다. 스윙에서 테이블이 다른 컴포넌트에 비해 매우 복잡하기 때문이다. 열 관리용 구성 요소들도 표를 통해 정리해 두기 바란다.

[표 2] 테이블에서 열들을 관리하는 구성 요소들

| 구성 요소 | 설명 |
|---|---|
| TableColumn | 하나의 열을 대표한다. |
| JTableColumnModel | 여러 열들을 통합 관리하기 위한 인터페이스 |
| DefaultTableColumnModel | JTable을 위해 JTableColumnModel의 기본 구현을 제공한다. |

이번엔 TableCellRenderer, TableCellEditor로 연결된 라인을 보자. 이들은 셀을 관리하는데 사용된다. TableCellRenderer는 셀의 내용을 화면에 비주얼하게(Visual) 나타내는 일을 맡고 있고 TableCellEditor는 셀의 내용을 편집하는 일을 맡는다. 보통 테이블에서 하나의 컬럼 안에 있는 모든 셀들은 동일한 방식으로 관리될 수 있다. 예를 들어 [화면 1]에서처럼 이름 컬럼에 있는 셀들은 모두 문자열(String) 형태로 취급될 수 있다. 반면 나이 컬럼에 있는 셀들은 모두 정수형(Integer) 데이터로 취급된다. 따라서 하나의 컬럼당(TableColumn) 하나의 TableCellRenderer와 TableCellEditor를 제공하면 된다. 셀 관리 구성 요소들도 표로 정리해 보자.

[표 3] 테이블에서 셀을 관리하는 구성 요소들

| 구성 요소 | 설명 |
|---|---|
| TableCellRenderer | 셀을 화면에 표시하기 위한 인터페이스다. |
| TableCellEditor | 셀을 편집하기 위한 인터페이스다. |

지금까지 테이블을 구성하는 구성요소들에 대해 개략적으로 알아보았다. 테이블의 데이터에 대한 접근 방법을 제시하는 모델, 열을 관리하는 구성요소들, 그리고 셀에 관련된 구성 요소들을 알아보았다. 이 외의 테이블 구성요소로서 중요한 것에는 각종 이벤트에 관련된 클래스들이 있는데 이들 구성요소들은 앞서 살펴보았던 다른 구성 요소들과 밀접한 관계를 형성하고 있기 때문에 다른 클래스들을 살펴볼 때 함께 알아보도록 한다. 여기까지의 내용을 통해 어느 정도 테이블의 구조에 대한 감을 잡았으리라 생각된다. 그러면 아주 간단한 예제를 하나 살펴보고 구체적인 테이블 파헤치기 작업에 들어가도록 하겠다.

[리스트 1] 간단한 테이블 예제

```
public class FirstTableDemo extends JFrame {

    /** 컬럼 헤더 문자열 */
    private static final String[] m_columnNames = {
        "이름", "나이", "전화번호", "직업"
    };

    /** 테이블에 포함될 데이터 */
    private Object[][] m_data = {
        { "장대원", new Integer(27), "02-123-4567",   "영화배우" },
        { "차인표", new Integer(30), "02-4546-3456",  "탤런트" },
        { "김희선", new Integer(24), "032-398-3214",  "탤런트" },
        { "신해철", new Integer(32), "0615-323-6583", "가수" },
        { "황수관", new Integer(57), "02-453-7689",   "의사" },
```

```
        { "김대중", new Integer(75), "062-378-6048",  "정치인" },
        { "박찬호", new Integer(27), "02-665-1297",   "야구선수" }
    };

    /** Constructor */
    public FirstTableDemo() {
        super("간단한 테이블 예제");

        // 테이블 생성
        JTable table = new JTable(m_data, m_columnNames);
        table.setPreferredScrollableViewportSize(
                new Dimension(500, 70));

        JScrollPane scrollPane = new JScrollPane(table);
        getContentPane().add(scrollPane, BorderLayout.CENTER);
    }

    public static void main(String[] args) {
        FirstTableDemo frame = new FirstTableDemo();
        frame.pack();
        frame.setVisible(true);
    }
}
```

[리스트 1]은 [화면 1]에서 보았던 프로그램의 소스코드인데 스윙에서 기본적으로 제공하는 기능에만 의지하여 테이블을 구성하는 프로그램이다. (공간을 줄이기 위해 실제 소스코드에서 제거된 부분이 몇 군데 있다. 따라서 위의 소스코드를 타이핑하여 실행하려면 좀 손을 봐야 한다. 프로그램을 실행해 보고 싶은 독자는 별도로 제공되는 소스코드를 받아서 사용하기 바란다.) 테이블을 만드는 것은 아주 쉽다. 컬럼헤더에 표시할 문자열들을 담고 있는 m_columnNames와 테이블에 표시할 실제 데이터들을 담고 있는 m_data를 JTable의 생성자에 매개변수로 넘겨주면 테이블을 생성하게 된다.

```
JTable table = new JTable(m_data, m_columnNames);
```

우리가 앞서 살펴보았던 것들 중에 JTable 클래스만 볼 수 있어서 실망한 독자가 있을지 모르겠다. 이렇게 간단히 테이블을 만들 수 있는 것은 대부분의 일들이 JTable 클래스의 이면에서 일어나기 때문이다. 따라서 스윙에서 제공하는 기본(디폴트) 기능을 그대로 사용하지 않고 무엇인가를 수정하거나 좀 더 다르게 행동하는 테이블(예를 들어 특정 컬럼을 기준으로 모든 행들을 소트하는 테이블)을 만들고 싶다면 반드시 그 이면에서 어떠한 일이 일어나고 있는지 알아야 하며 또한 디폴트 작동 방식을 자신이 원하는 방식으로 바꿀 수 있어야 한다.
더 진도를 나가기 전에 이쯤에서 앞서 보았던 [그림 1]을 다시 감상하는 시간을 갖도록 권한다. 이번에는 좀 더 자세히 살펴보기 바란다. TableColumnModel은 TableColumn을 여러 개 소유하고(Contains)있다는 것을 발견하였는가? 또한 JTable과 TableColumnModel 둘다 ListSelectionModel

을 이용한다는 것을 발견하였는가? 앞서도 말했지만 숲을 먼저 잘 보고 가야한다. 그럼 독자의 머리 속에 [화면 1]의 영상이 세밀히 남아있길 기대하며 테이블 구성요소들의 구체적인 해체 작업에 들어가겠다.

# 2. 테이블 구성 요소의 자세한 관찰

JTable은 전면에 서서 테이블 시스템을 대표한다(Represent). 그러나 테이블과 관련된 대부분의 일들은 JTable의 이면에서 활동하고있는 다른 구성요소들에 의해 처리된다. 테이블의 열 헤더(Column Header)를 관리하는 JTableHeader, 각각의 셀에 관련된 일을 처리하는 TableCellRender와 TableCellEditor, 그리고 이러한 셀들을 열 단위로 관리하기 위해 TableColumn이 사용된다. 다시 여러 열들을 통합하는 TableColumnModel이 있다. JTable은 많은 일들을 이들 구성요소들에게 위임하고 자신은 그들을 대표하는 대표이사(?) 역할을 하는 셈이다. 이러한 내용을 머리에 담아두고 여러 구성 요소들이 구체적으로 어떠한 특성과 메쏘드를 가지고 자신의 맡은 일을 처리하고 있는지 살펴보자.

### (1) TableModel, AbstractTableModel, DefaultTableModel

TableModel 인터페이스는 화면에 표시할 데이터들을 JTable에게 제공할 수 있는 몇 가지 메쏘드들을 정의하고 있다. [표 4]에 중요한 메쏘드들을 정리해 보았는데 메쏘드의 완전한 프로토타입은 JDK의 문서를 참조하기 바란다.

[표 4] TableModel 인터페이스의 메쏘드들

| 메쏘드 | 설명 |
|---|---|
| getColumnClass() | JTable은 각 셀을 렌더링(화면에 표시)하거나 편집할 때 셀에 포함되어 있는 데이터가 어떠한 타입(Class)의 데이터인지 알 필요가 있다. 이러한 정보를 getColumnClass()가 제공한다. 좀 더 자세한 내용은 나중에 설명하겠다. |
| getColumnCount() | 컬럼의 개수를 얻는다. |
| getColumnName() | 컬럼 헤더에 사용하기 위하여 컬럼의 이름을 얻는다. |
| getRowCount() | 행의 개수를 얻는다. |
| getValueAt() | 특정 셀의 데이터 값을 얻는다. |
| isCellEditable() | 특정 셀이 편집 가능한지 알아낸다. |
| setValueAt() | 특정 셀의 데이터 값을 설정한다. |

TableModel 인터페이스를 구현한 객체는 JTable의 모델로 사용될 수 있다. 스윙에서는 TableModel 의 기본 구현 클래스인 AbstractTableModel 클래스를 제공한다. 따라서 여러분이 여러분 자신의 모델을 제공하고 싶다면 AbstractTableModel 클래스를 상속하고 필요한 메쏘드만 오버라이드하면 된다. AbstractTableModel 클래스가 리스너들(EventListeners)을 관리하고 이벤트를 통고(Fire)하는 등의 일들을 모두 처리해준다. 그러나 다음과 같은 세 개의 메쏘드는 여러분이 직접 구현해야 한다. AbstractTableModel가 이 메쏘드들을 구현하지 않기 때문이다.

```
public int getRowCount();
public int getColumnCount();
public Object getValueAt(int rowIndex, int columnIndex);
```

이렇게 구현된 새로운 모델을 JTable에 사용하는 것은 매우 쉽다. 새로 구현된 모델 클래스를 MyTableModel이라고 하면 다음과 같이 새 모델을 JTable의 생성자에 넘겨주면 된다.

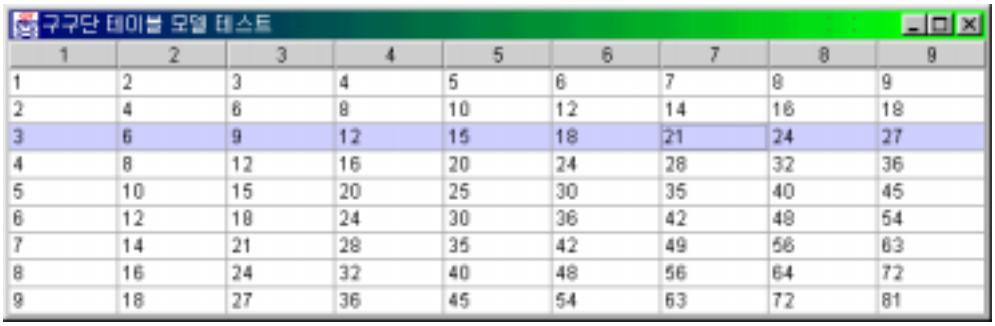
```
TableModel myTableModel = new MyTableModel();
JTable table = new JTable(myTableModel);
```

물론 JTable의 setModel() 메쏘드를 이용하여 테이블이 이미 생성된 후라도 언제든지 모델을 교체할 수 있다. [리스트 2]는 구구단을 표시하기 위해 GugudanTableModel이라는 사용자 정의 모델을 만들어 사용한 예이다. 테이블의 모델을 어떻게 정의하고 사용하는지 살펴두기 바란다. 이 모델을 살펴보면 모델이 실제로는 아무런 데이터도 유지하지 않는다는 것을 알 수 있다. 앞서도 말했지만 TableModel 인터페이스는 테이블에 표시할 데이터를 얻어오는 방법에 관한 메쏘드만을 정의한다. 실제로는 데이터가 있든 없든 상관없다는 것을 알 수 있다.

[리스트 2]

```
public class GugudanModelDemo extends JFrame {
    public GugudanModelDemo() {
        super("구구단 테이블 모델 테스트");

        GugudanTableModel myModel = new GugudanTableModel();
        JTable table = new JTable(myModel);

        // 컨테이너에 테이블을 직접 넣을 땐 헤더와 몸체를
        // 각각 따로 넣어야 한다.
        getContentPane().add(table.getTableHeader(),
                BorderLayout.NORTH);
        getContentPane().add(table, BorderLayout.CENTER);
    }

    public static void main(String[] args) {
        GugudanModelDemo frame = new GugudanModelDemo();
        frame.pack();
        frame.setVisible(true);
    }
}

/** 구구단 테이블 모델 클래스 */
class GugudanTableModel extends AbstractTableModel {
    public int getColumnCount() { return 9; }
    public int getRowCount() { return 9; }
    public String getColumnName(int col) { return "" + (col+1); }
    public Object getValueAt(int row, int col) {
        return new Integer((row+1)*(col+1));
    }
}
```

[화면 2] 구구단 모델을 사용한 테이블 예제




| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 |
| 3 | 6 | 9 | 12 | 15 | 18 | 21 | 24 | 27 |
| 4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 | 36 |
| 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 |
| 6 | 12 | 18 | 24 | 30 | 36 | 42 | 48 | 54 |
| 7 | 14 | 21 | 28 | 35 | 42 | 49 | 56 | 63 |
| 8 | 16 | 24 | 32 | 40 | 48 | 56 | 64 | 72 |
| 9 | 18 | 27 | 36 | 45 | 54 | 63 | 72 | 81 |

지금까지 테이블에서 사용되는 모델에 관한 이야기를 했는데 아직 하나가 더 남았다. DefaultTableModel이라는 클래스인데 이 클래스는 JTable에서 기본적으로(디폴트) 사용되는 모델이다. [리스트 1]에서 우리는 아무런 모델도 정의하지 않았었다. JTable은 외부에서 모델이 제공되지 않을 때 DefaultTableModel을 자신의 모델로 사용한다. 이 클래스 역시 AbstractTableModel 클래스를 상속하며 데이터를 저장하기 위해 벡터들의 벡터를 내부에 유지하고 있다. 우리의 첫 번째 예제 프로그램을 실행해보면 각 셀들이 모두 편집 가능함을 알 수 있다. 어떤 셀에 마우스를 더블 클릭하면 내용을 편집할 수 있다. DefaultTableModel이 기본적으로 모든 셀을 편집 할 수 있도록 하기 때문이다. 이 외에도 DefaultTableModel의 여러 특성들이 있으나 그리 중요하지 않은 내용들이므로 생략하겠다.

### (2) TableColumnModel, DefaultColumnModel

[그림 1]에서 살펴보았듯이 JTable는 TableModel 이외에 TableColumnModel도 사용한다. TableColumnModel은 테이블에 포함되는 여러 열들을 관리하는데 사용된다. 열단위의 선택에 대한 처리와 열의 추가, 삭제, 이동 등 열의 배치 문제를 처리한다. 이러한 작업을 위해 필요한 메쏘드들이 [표 5]에 포함되어 있다.

[표 5] TableColumnModel 인터페이스의 주요 메쏘드들

| 메쏘드 | 설명 |
|---|---|
| addColumn() | 현재의 TableColumnModel에 새로운 열을(TableColumn) 추가한다. |
| getColumnIndex()<br>getColumnAtX() | 특정한 위치의 열(TableColumn)을 얻는다. |
| moveColumn() | 특정한 열의 위치를 옮긴다. |
| removeColumn() | 특정한 열을 현재의 TableColumnModel에서 제거한다. |

TableColumnModel의 메쏘드들은 모두 화면에만 영향을 끼친다. 예를 들어 특정한 컬럼의 위치가 변경될 경우 실제로 모델에 있는 데이터들이 바뀌는 것은 아니다. 단순히 화면상의 열의 위치만 바뀌게된다. 그러나 이러한 사실에 관심을 갖는 다른 객체가 있다면 그런 객체들에게 열의 위치가 옮겨졌음을 알려줘야 한다. 그러한 객체들은 TableColumnModelListener 인터페이스를 구현하는 객체들이다. 이때 ColumnModelEvent가 사용된다. [표 5]에는 나타내지 않았지만 당연히 addColumnModelListener()와 removeColumnModelListener() 메쏘드가 TableColumnModel에 존재

한다. TableColumnModelListener 인터페이스는 다음과 같은 메쏘드들을 정의한다. `columnSelectionChanged()` 메쏘드로 컬럼단위의 선택 변경 등을 알 수 있다.

```
public void columnAdded(TableColumnModelEvent e)
public void columnRemoved(TableColumnModelEvent e)
public void columnMoved(TableColumnModelEvent e)
public void columnMarginChanged(ChangeEvent e)
public void columnSelectionChanged(ListSelectionEvent e)
```

앞에서 보았던 DefaultTableModel과 마찬가지로 TableColumnModel을 구현한 DefaultTableColumnModel 클래스가 존재한다. 예상할 수 있듯이 이 클래스는 테이블의 열에 변화가 있을 때 리스너(TableColumnModelListener) 객체들에게 이벤트를 통고하거나 열 단위의 선택 등을 처리한다.

### (3) TableCellRenderer, TableCellEditor

하나의 열에 대해 관리가 필요할 때 TableColumn 클래스가 사용된다. TableColumn 클래스에 대해 구체적인 설명을 하지는 않았지만 TableColumnModel을 설명할 때 TableColumn이 쓰이는 방법을 배웠다. 그런데 하나의 열이란 헤더를 포함하여 여러 셀들의 집합체라고 생각할 수 있다. 따라서 열이란 단지 셀들을 관리하는 도구라 생각해도 무방하다. 그럼 이제 셀에 대한 이야기를 해보자.

테이블에서의 각각의 셀에 대한 작업을 생각해보자. 셀과 관련된 작업에는 셀의 내용을 화면에 표시하는 작업과 셀의 내용을 편집하는 작업이 있다. 앞의 작업을 TableCellRenderer 인터페이스가 맡고 뒤의 작업을 TableCellEditor 인터페이스가 맡는다. 테이블은 이들 인터페이스를 구현하는 객체를 자신의 셀 표시기(Cell Renderer)나 셀 편집기(Cell Editor)로 사용한다. 이 두 인터페이스는 각각 오직 하나씩의 메쏘드만을 정의한다. 이들 메쏘드는 다음과 같다.

```
public Component getTableCellRendererComponent(
        JTable table, Object value, boolean isSelected,
        boolean hasFocus, int row, int column)

public Component getTableCellEditorComponent(
        JTable table, Object value, boolean isSelected,
        int row, int column)
```

두 메쏘드 모두 어떤 컴포넌트를 리턴하는데 이 컴포넌트가 바로 셀을 화면에 표시하거나 셀을 편집하는데 사용된다. JTable은 별도의 셀 표시기나 셀 편집기가 제공되지 않는다면 TableCellRenderer를 기본 구현한 DefaultTableCellRenderer 클래스와 TableCellEditor를 기본 구현한 DefaultTableCellEditor 클래스를 사용한다.

특히 DefaultTableCellRenderer는 클래스는 getTableCellRendererComponent 메쏘드의 리턴 값으로 몇 개의 미리 정의된 컴포넌트를 리턴하는데 매개변수로 들어오는 value 값의 타입에 따라 적

당한 컴포넌트를 리턴한다. [표 6]에 그런 관계를 정리해 보았다. 예를 들어 어느 셀에 표시할 값이 Boolean 타입의 값이라면 참 또는 거짓을 표현할 수 있는 JCheckBox가 사용되는 것이 타당하다.

[표 6] DefaultTableCellRender가 제공하는 렌더링용 컴포넌트들

| value의 타입 | 리턴되는 렌더링용 컴포넌트 |
|---|---|
| Boolean | JCheckBox |
| Number | JLabel (오른쪽으로 정렬된) |
| ImageIcon | JLabel (중앙으로 정렬된) |
| Object | JLabel (Object 객체의 문자열 값으로 된) |

경우에 따라서 여러분은 DefaultTableCellRenderer나 DefaultTableCellEditor가 미리 정해놓은 것과는 다르게 화면에 표시되거나 다른 방식으로 셀이 편집되길 원할 수 있다. [화면 3]의 프로그램은 [화면 1]에서 보았던 프로그램에서 “직업” 열을 편집할 수 있도록 바꿔놓은 것이다. “직업” 열의 셀을 편집할 셀 편집기로서 콤보박스를 등록해 놓았다. 다음과 같은 코드만 추가하면 된다.

```
TableColumn job = table.getColumnModel().getColumn(3);

// 직업을 나타내는 셀을 콤보박스로 편집할 수 있도록
// 셀 에디터에 콤보박스를 등록한다.
JComboBox comboJob = new JComboBox();
comboJob.addItem("영화배우");
comboJob.addItem("탤런트");
comboJob.addItem("가수");
comboJob.addItem("의사");
comboJob.addItem("정치인");
comboJob.addItem("야구선수");
comboJob.addItem("백수");
job.setCellEditor(new DefaultCellEditor(comboJob));
```

[화면 3] 콤보박스를 셀 편집기로 사용한 예

"직업" 열을(4번째 TableColumn)을 얻어낸 다음 그 열의 셀 편집기서 콤보박스를 등록하면 된다. 편집기의 등록은 setCellEditor() 메쏘드를 사용한다. 맨 마지막 라인의 new DefaultCellEditor(comboJob)은 콤보박스 셀 편집기를 만들어 낸다. DefaultCellEditor의 생성자는 세 개가 있는데 다음과 같다.

```
public DefaultCellEditor(JTextField x)
public DefaultCellEditor(JCheckBox x)
public DefaultCellEditor(JComboBox x)
```

DefaultCellEditor의 생성자에 콤보박스를 넘겨주면 DefaultCellEditor의 getTableCellEditorComponent() 메쏘드는 콤보박스를 리턴 해준다. DrfaultCellEditor를 이용하여 간단하게 셀 편집기를 만들 수 있다는 것을 알아두기 바란다. 그러나 DrfaultCellEditor 등에 의존하지 않고도 얼마든지 셀 편집기나 셀 표시기를 만들 수 있다. TableCellRenderer 인터페이스나 TableCellEditor 인터페이스를 구현하기만 하면 되기 때문이다. [화면 4]에 아주 실용적인 예를 볼 수 있다.

[화면 4] 사용자 정의 셀 표시기와 셀 편집기 사용 예제


여러분이 CD 플레이어를 구축한다고 생각해보자. 약간씩 다른 분위기를 연출 할 수 있도록 각 트랙별로 좌, 우 스피커의 볼륨을 설정할 수 있다. (이렇게 할 사람이 몇이나 될 지 모르지만 우리의 예제로서는 적당한 프로그램이다) CD 플레이어의 사용자 인터페이스를 [화면 4]와 같은 테이블로 만들 수 있다. 이 때 좌, 우의 볼륨을 표시하거나 조절하기 위해 스크롤 바를 사용한다면 사용자로부터 훨씬 좋은 호응을 얻을 수 있을 것이다. 이 프로그램에서는 스크롤 바가 셀 표시기이자 셀 편집기로 사용되었다. [리스트 3]의 소스코드에서 새로 만들어진 셀 표시기와 셀 편집기를 등록하는 부분을 살펴보자. 이전에 TableColumn의 setCellEditor() 메쏘드로 사용자 정의 셀 편집기를 등록한다고 했었다. JTable의 다음 두 메쏘드를 통해서도 셀 표시기와 편집기를 등록할 수 있는다.

```
public void setDefaultRenderer(Class columnClass,
        TableCellRenderer renderer)

public void setDefaultEditor(Class columnClass,
        TableCellEditor editor)
```

첫 번째 매개변수는 자바의 특정한 클래스를 지칭하는 "Class" 객체다. 이 두 메쏘드가 하는 일은 특정한 타입(클래스)의 셀에 대해 특정한 셀 표시기 또는 셀 편집기를 사용하도록 등록하는 것이다. 보통 하나의 열에 포함된 모든 셀은 동일한 타입(클래스) 이므로 TableColumn 클래스에서

사용했던 setCellRenderer() 메쏘드 또는 sefCellEditor() 메쏘드를 사용한 것과 동일한 효과를 가져오게 된다.
다시 [리스트 3]을 보면 Volumn 클래스는 현재의 볼륨값을 나타내는 사용자 정의 클래스이고 VolumnRenderer와 VolumnEditor 클래스는 새로 만든 사용자 정의 셀 표시기와 셀 편집기다. JTable의 setDefaultRenderer() 메쏘드는 화면에 표시할 데이터가 첫 번째 매개변수가 나타내는 클래스 타입일 경우 두 번째 매개변수로 들어온 셀 표시기를 사용한다. 따라서 셀의 값이 Volume 타입의 값이라면 VolumeRender를 셀 표시기로 사용한다는 얘기다. setDefaultEditor() 메쏘드도 마찬가지로 Volume 타입의 데이터 값이 들어있는 셀은 VolumeEditor로 편집할 수 있도록 해준다.

[리스트 3] 볼륨 표시기와, 볼륨 에디터를 사용한 프로그램

```
public class AudioDemo extends JFrame {
    public AudioDemo() {
        super("사용자 정의 CellRenderer, CellEditor 테스트");

        AudioModel model = new AudioModel();
        JTable table = new JTable(model);

        // 새로운 CellRenderer와 CellEditor를 등록한다.
        table.setDefaultRenderer(Volume.class, new VolumeRenderer());
        table.setDefaultEditor(Volume.class, new VolumeEditor());

        getContentPane().add(table.getTableHeader(),
                BorderLayout.NORTH);
        getContentPane().add(table, BorderLayout.CENTER);
    }

    public static void main(String args[]) {
        new AudioDemo();
    }
}
```

[리스트 4] VolumeRenderer

```
/** Volume 객체를 위한 셀 표시기 */
public class VolumeRenderer extends JScrollBar
        implements TableCellRenderer {

    public VolumeRenderer() {
        // 수평 스크롤바를 셀 표시기로 사용한다.
        super(JScrollBar.HORIZONTAL);
    }

    public Component getTableCellRendererComponent(
            JTable table, Object value, boolean isSelected,
```

```
            boolean hasFocus, int row,int column) {

        if (value == null) { return this; }
        if (value instanceof Volume) {
            setValue(((Volume)value).getVolume());
        }
        else { setValue(0); }

        return this;
    }
}
```

[리스트 5] VolumeEditor

```
/** Volume 객체를 위한 셀 편집기 */
public class VolumeEditor extends JScrollBar
        implements TableCellEditor {

    // 셀 편집에 의해 변경된 내용을 리스너들에게 알려줘야 한다.
    private Vector m_listeners;
    private int m_originalValue;

    public VolumeEditor() {
        // 수평 스크롤바를 셀 편집기로 사용한다.
        super(JScrollBar.HORIZONTAL);
        m_listeners = new Vector();
    }

    public Component getTableCellEditorComponent(
            JTable table,Object value, boolean isSelected,
            int row,int column) {

        if (value == null) { return this; }
        if (value instanceof Volume) {
            setValue(((Volume)value).getVolume());
        }
        else { setValue(0); }

        table.setRowSelectionInterval(row, row);
        table.setColumnSelectionInterval(column, column);
        m_originalValue = getValue();

        return this;
    }

    public void cancelCellEditing() {
        fireEditingCanceled();
```

```
        }
        public Object getCellEditorValue() {
            return new Integer(getValue());
        }
        public boolean isCellEditable(EventObject eo) {
            return true;
        }
        public boolean shouldSelectCell(EventObject eo) {
            return true;
        }
        public boolean stopCellEditing() {
            fireEditingStopped();
            return true;
        }

        public void addCellEditorListener(CellEditorListener cel) {
            m_listeners.addElement(cel);
        }
        public void removeCellEditorListener(CellEditorListener cel) {
            m_listeners.removeElement(cel);
        }

        protected void fireEditingCanceled() {
            setValue(m_originalValue);
            ChangeEvent ce = new ChangeEvent(this);
            for (int i = m_listeners.size(); i >= 0; i--) {
                ((CellEditorListener)m_listeners.elementAt(i))
                        .editingCanceled(ce);
            }
        }
        protected void fireEditingStopped() {
            ChangeEvent ce = new ChangeEvent(this);
            for (int i = m_listeners.size() - 1; i >= 0; i--) {
                ((CellEditorListener)m_listeners.elementAt(i))
                        .editingStopped(ce);
            }
        }
    }
```

셀 표시기와는 달리 셀 편집기는 약간의 추가 작업이 필요하다. 편집으로 인해 변경된 값에 관심을 갖는 객체가 있을 수 있기 때문이다. 우리의 CD 플레이어를 예로 든다면 사용자가 스크롤바를 통해 볼륨을 조절할 때 새로이 조절된 볼륨 값을 모델이 알고 있어야 할 필요가 있다. 그래야 실제로 볼륨을 조절할 수 있기 때문이다. 이러한 문제를 해결하기 위해 셀 편집기는 변경된 값에 관심을 갖는 리스너(CellEditorListener)들을 관리해야 한다. [리스트 5]를 보면 리스너들을 직접 관리하는 것을 볼 수 있다. 셀 편집 중에 편집이 취소되면 editingCanceled() 메쏘드로 리스너들에게 편집 취소를 알리며, 셀 편집이 성공적으로 완료되었을 때는 editingStopped() 메쏘드로 리스너들에게 편집 완료를 알려줘야 한다.

### (4) JTable

드디어 JTable 클래스를 살펴볼 차례다. 그러나 JTable 클래스 자체는 별다른 내용을 가지고 있지 않다. JTable 클래스에는 무척 많은 메쏘드들이 존재하지만 우리가 지금까지 살펴보았던 클래스들이 대부분의 작업을 위임받아 처리하기 때문이다. 그럼에도 불구하고 여러분들은 대부분 JTable의 메쏘드들을 직접 사용하게될 것이다. JTable이 전체 테이블 시스템의 전면에 서서 많은 요청들을 받아들이기 때문이다. 그러니 표준 Document를 통해 어떠한 메쏘드들이 있는지 잘 살펴두기 바란다. 여기서 모든 메쏘드들을 다룰 수는 없고 우리의 글과 관련하여 몇몇 중요한 몇몇 메쏘드만을 살펴볼 것이다.

[표 7] JTable의 주요 메쏘드들

| 메쏘드 | 설명 |
|---|---|
| JTable() | 여러 종류의 생성자가 있다. 이 글에서 보았던 것처럼 헤더 문자열과 데이터를 배열 또는 벡터로 받기도 하고 매개변수로 테이블의 모델을 취하기도 한다. |
| getTableHeader() | 테이블 헤더 컴포넌트를 얻는다. 스크롤바를 사용하지 않고 직접 컨테이너에 테이블을 집어넣으려면 헤더와 몸체를 따로 따로 집어 넣어야한다. |
| setDefaultRenderer()<br>setDefaultEditor() | 셀 데이터의 타입에 따른 셀 표시기와 셀 편집기를 등록한다. 이외에도 셀 관련된 메쏘드들이 많이 있다. |
| setRowSelectionInterval()<br>getSelectedRow() | (프로그래머가)직접 행을 선택하는 메쏘드와 선택된 행을 알아내는 메쏘드다. 선택에 관련된 메쏘드는 아주 많다. 그러니 API 문서를 꼭 보기 바란다. |
| convertColumnIndexToModel() | 화면상에 열의 인덱스를 모델상의 인덱스로 바꿔준다. 화면과 실제의 데이터가 따로 취급된다는 것을 배웠었다. |
| getValueAt() | 특정 셀의 값을 얻어낸다. |
| addColumn() | 새로운 열을 추가한다. |
| editCellAt() | (프로그래머가)직접 특정 셀을 편집하게 한다. 이 메쏘드를 호출하면 편집기 컴포넌트가 나타나게 될 것이다. |
| getModel() | 테이블의 모델을 얻는다. |

위의 메쏘드들은 극히 일 부분일 뿐이다. JTable에는 무척 많은 메쏘드들이 존재하는데 이렇게 많은 메쏘드가 존재하는 이유를 그 정의부분에서 찾을 수 있다. JTable의 완전한 정의 부분은 다음과 같다.

```
public class JTable extends JComponent
        implements TableModelListener, Scrollable,
        TableColumnModelListener, ListSelectionListener,
        CellEditorListener, Accessible
```

JTable은 테이블을 구성하는 여러 구성요소들을 관리, 통제하며 그들이 활동할 공간을 제공하고 있다. JTable은 다른 구성요소들의 원활한 활동을 위하여 이들간의 메시지 전달자 역할도 겸하고 있다. 따라서 JTable은 TableModel에서 발생하는 이벤트나 TableColumnModel에서 발생하

는 이벤트 그리고 CellEditor에서 방생하는 이벤트 등 각 구성 요소들의 변화나 요구에 모두 관심을 기울여야 한다. 그래서 그렇게 많은 인터페이스를 구현(implements)하고 있는 것이다. 물론 실제적인 일은 대부분 구성요소들에게 위임하고 있다.

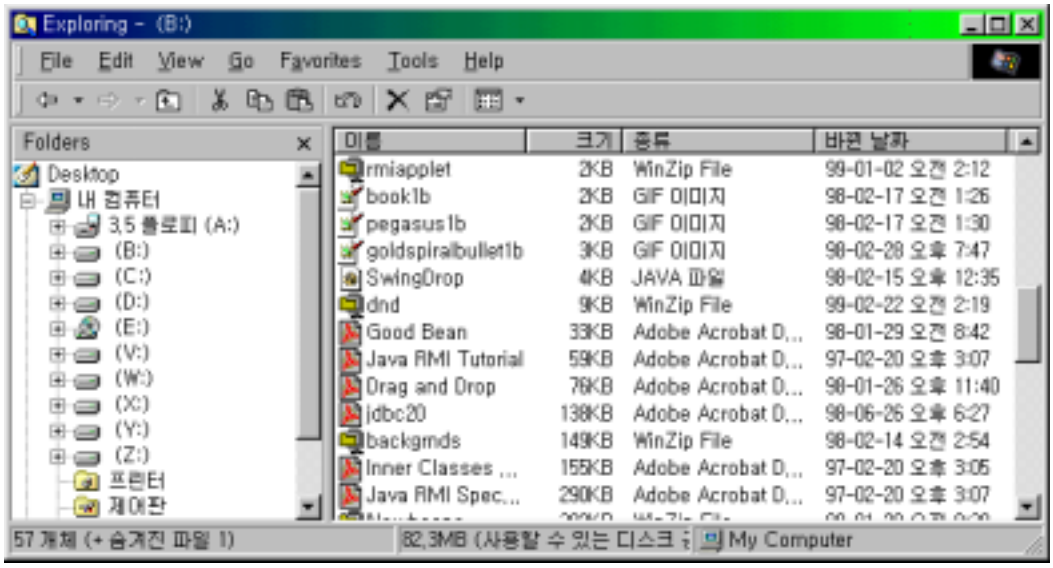
# 3. 실전 테이블 프로그래밍

지금까지 다소 원론적이고 지루한 내용을 잘 참고 읽어주어서 고맙다. 그렇지만 필자가 이러한 글을 쓰게된 목적이 독자 여러분들께 “고기가 잡히는 원리”(?)를 설명하기 위함이니 만큼 딱딱한 내용이었을지라도 끝까지 잘 보고 잘 정리해 두면 나중에 힘을 발휘할 수 있을 것이다. 다만 필자에게 복잡한 원리를 재미있고 시원시원하게 풀어줄 수 있는 재주가 부족한 것이 죄송할 뿐이다.
여기서부터는 우리가 지금까지 배웠던 원리를 이용한 실전적인 예제들이 준비되어 있다. 당장 뭔가 필요한 사람들에게 도움이 되리라 생각한다. 그러나 필자가 제시하는 예제들을 단순히 자르고 붙여서 사용하기보다는 천천히 내용을 음미하면서 앞에서 살펴보았던 테이블의 구조를 정리하는 계기로 만들기 바란다. 예제들은 지면 관계상 소스코드를 모두 싣지 못하였다. 별도로 제공되는 소스코드를 참조하기 바란다.

## 1. 정렬 가능한 테이블 만들기

윈도우98의 탐색기에는 정렬(Sorting) 기능이 있다. 탐색기의 오른쪽 영역에서 정렬하고 싶은 열의 헤더를 클릭하면 파일들이 그 클릭된 열을 기준으로 정렬된다. 정렬 기능을 가지고 있는 테이블은 아주 쓰임새가 많다. 만약 여러분이 주소록 프로그램에 테이블을 이용한다면 특별히 검색 명령을 사용하지 않고도 찾는 내용이 어디에 있는지 직관적으로 알 수 있게된다. 그럼 우리도 이렇게 정렬되는 테이블을 한번 만들어보자.

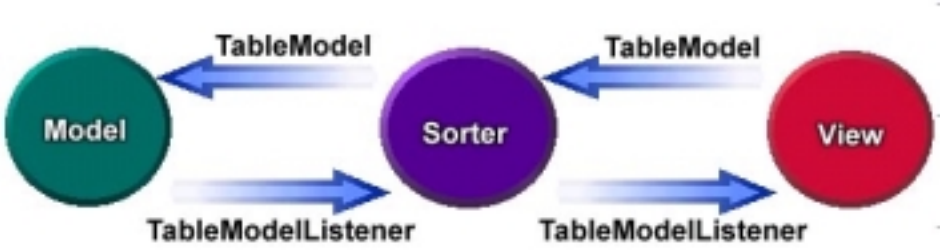
[화면 5] 파일의 크기 순서로 정렬된 탐색 창



정렬 기능을 어디에 넣는 것이 좋을까? 데이터를 직접 정렬하도록 하는 것은 어떨까? 이 것은 말이 안 된다. 왜냐하면 테이블의 데이터는 멀리 떨어져있는 데이터베이스 서버에 존재할 수도 있기 때문이다. 그럼 모델에 정렬 기능을 넣는 것은 어떨까? 이것도 좋은 방법이 아니다. 왜냐하면 단일 모델을 가지고 여러 가지의 뷰를 제공해야 할 경우가 있기 때문이다. 마지막으로 생각할 수 있는 것이 뷰에 설치하는 것이다. 그럼 이건 문제가 없을까? 문제가 있다. 뷰는 화면에 정보를 표시하는 고유 기능을 가지고 있는데 이 곳에 정렬에 관한 알고리즘을 담아 놓는 것은 프로그램의 유연성을 떨어뜨리게 된다. 그리고 각 뷰마다 정렬 알고리즘을 구현한다는 것은 생각할 수 없다. 그럼 어떻게 하란 말인가?

정답은 독립이다. 정렬 기능을 독립시키는 것이다. 이러한 디자인 테크닉을 일명 정책(Strategy, Policy) 패턴이라고 하는데 3월호에 특집으로 다뤘던 “디자인 패턴” 중에 하나다. 자세한 내용은 “Design Patterns" (Erich Gamma외)라는 책을 참조하기 바란다. 우리가 패턴에 대한 공부를 하는 것이 아니기 때문에 더 설명하지는 않겠다. 그렇지만 정렬 기능을 독립시킴으로써 훨씬 유연한 구조를 유지할 수 있다는 것을 알기 바란다. 단적인 예로 나중에 훨씬 빠른 정렬 알고리즘을 발견하였다면 알고리즘 부분만 바꿔주면 되는 것이다.

[그림 2] 테이블 정렬 개념도




[그림 2]에 우리가 만들 정렬 가능 테이블에 대한 구조도가 나와있다. 모델과 뷰는 우리가 지금껏 보아왔던 바로 그것이다. 단순히 모델은 TableModel이고 뷰는 JTable에서 모델을 제외한 부분이라고 생각해도 무방하겠다. 앞에서 정렬 기능을 독립시킨다고 했는데 그림에서 가운데에 있는 “Sorter”가 그것이다. 편의상 “정렬기”(발음에 주의하기 바란다. ^^) 라고 부르겠다. 정렬기는 뷰의 입장에서 보면 모델과 다를 게 없다. 단지 정렬된 데이터들을 보내주는 모델일 뿐이다. 뷰가 화면에 데이터를 표시하기 위하여 모델(여기서는 정렬기)로부터 데이터를 요청하면 정렬기는 실제 모델로부터 데이터를 가져온 다음 그것을 알맞게 정렬하여 뷰에게 보내주는 것이다. TableModelListener가 사용된 이유는 실제의 모델에 행이 추가되거나 삭제 되었을 때 정렬기가 이것을 감지하여 테이블을 새로 정렬하도록 하기 위해서다. 정렬기를 이용하는 방법은 다음과 같다.

```
MyTableModel myModel = new MyTableModel();
```

```
TableSorter sorter = new TableSorter(myModel);
JTable table = new JTable(sorter);
sorter.addMouseListenerToHeaderInTable(table);
```

정렬기 클래스는 TableSorter라는 클래스다. TableSorter 클래스도 TableModel 클래스를 상속하기 때문에 정렬기도 일종의 모델로 볼 수 있다. 편의를 위해 실제 구현에서는 하나의 보조 클래스(TableMap 클래스)가 더 존재한다. 그렇지만 신경 쓰지 않아도 좋다. 그럼 순서대로 정렬기를 자신의 테이블에 설치하는 방법을 보자.

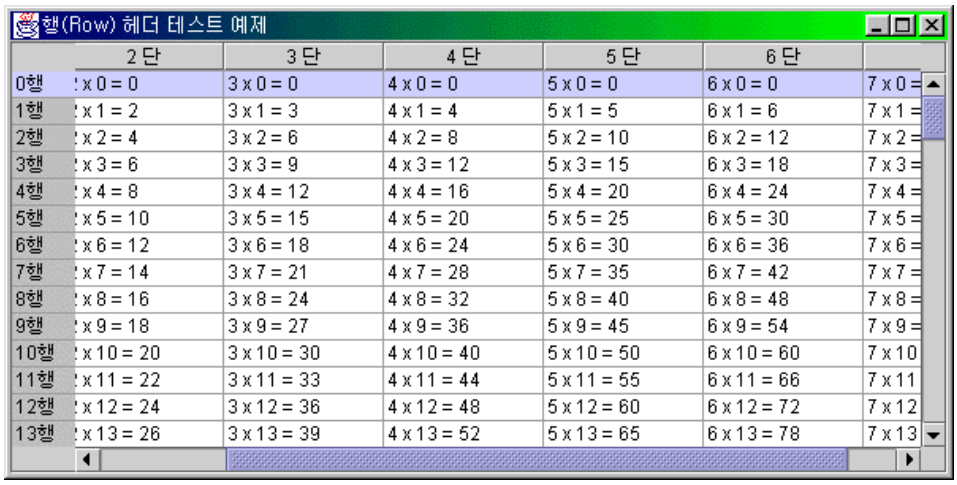
1) 먼저 자신의 실제 모델을 생성한다.
2) 1)에서 생성한 모델을 TableSorter의 생성자에 넘겨준다.
3) 생성된 TableSorter 객체를 JTable의 생성자에 넘겨준다. 이때 TableSorter는 TableModel로서 넘어가는 것이다.
4) 테이블의 헤더가에 마우스 클릭을 감지할 수 있도록 addMouseListenerToHeaderInTable() 메쏘드를 사용한다.

TableSorter 클래스는 AbstractTableModel 클래스를 상속한다. (실제로는 TableMap 클래스가 AbstractTableModel을 상속하고 TableSorter 클래스가 다시 TableMap 클래스를 상속한다) 왜냐하면 뷰에게 테이블의 모델처럼 보이게 하려하기 위해서다. 그리고 실제의 모델에 대한 레퍼런스를 유지하면서 뷰로부터 데이터 요청이 있을 경우에 실제 모델에게서 데이터를 가져온 다음 가공(정렬)하여 뷰에게 보내주게 된다. addMouseListenerToHeaderInTable() 메쏘드는 열 헤더(Column Header)에 발생하는 마우스 클릭을 감지하여 해당 열을 기준으로 테이블을 정렬하도록 하는 일을 맡는다. 실제의 코드는 함께 제공되는 소스코드를 살펴보기 바란다.

## 2. 행 헤더(Row Header)를 갖는 테이블

이 글의 앞부분에서 우리는 구구단을 만들어 보았었다. 그런데 구구단에 열 헤더는 있었지만 행 헤더가 없었다. 스윙의 테이블 시스템이 기본적으로 행 헤더를 지원하지 않았기 때문이다. 그러나 지금은 상황이 달라졌다. 독자 여러분도 우리가 지금까지 보아왔던 예제들을 생각해 보면 행 헤더를 만드는 것도 못할 건 없을 것이라고 생각할 것이다. 그럼 어떠한 방법이 있는지 생각해보자. 행 헤더는 우선 테이블의 가장 좌측에 위치해야한다. 그리고 테이블의 데이터 영역이 좌우로 스크롤되더라도 행 헤더는 제자리를 지키고 있어야한다. 테이블을 상하로 스크롤하더라도 열 헤더(Column Header)가 항상 제자리(맨 위)에 있는 것과 마찬가지다. [화면 6]에 행 헤더를 사용하여 버전업시킨 새 구구단이 있다. 행 헤더가 열 헤더만큼 깔끔하지는 않지만 잘 작동한다. 마음만 먹으면 열 헤더처럼 깔끔하게 하는 것도 문제는 아니다. 그럼 어떻게 행 헤더가 구현되는지 살펴보자.

[화면 6] 행 헤더가 있는 구구단 테이블




행 헤더 구현의 핵심은 두 개의 테이블을 사용하는 것이다. 하나는 행 헤더용으로 다른 하나는 진짜 테이블용으로 두 개의 테이블을 만든 다음 하나로 합치는 것이다. 편의상 이 두 개의 테이블을 각각 헤더 테이블과 데이터 테이블이라는 이름으로 부르겠다. 앞에서 테이블의 열들을 통합하여 관리하는 역할을 TableColumnModel이 맡고 있다고 했다. 우리는 각각의 테이블마다 하나씩의 TableColumnModel이 사용된다는 것을 테이블 시스템 구성도를 통해 이미 알고 있다. 따라서 헤더 테이블에는 안에 있는 TableColumnModel은 헤더용으로 하나의 열만 가지고 있어야 한다. 이때 테이블의 모델은 하나로 유지한다. 말하자면 두 개의 테이블이 하나의 모델을 쓴다는 얘기다. 스윙의 컴포넌트들이 MVC 구조로 작동되기 때문에 가능한 일이다. 어쨌든 모델의 입장에서 보면 그림에서 행 헤더로 사용된 부분이 실제로는 첫 번째(Index=0) 열인 셈이다. 다음 코드를 보자.

```
        // 행 해더 역할을 수행할 TableColumnModel 생성한다.
        // 첫번째 열을 제외한 나머지 열은 모두 무시한다.
        TableColumnModel        rowHeaderModel        =        new
DefaultTableColumnModel() {
            boolean first = true;

            public void addColumn(TableColumn tc) {
                if (first) {
                    // 열의 크기를 설정한다.
                    tc.setMaxWidth(50);
                    super.addColumn(tc);
                    first = false;
                }
                // 첫번째 열을 제외한 나머지는 모두 무시한다.
            }
        };

        // 테이블의 실제 데이터 영역을 위한 TableColumnModel을 생성한다.
```

```
// 행 헤더를 위해 첫번째 열은 무시한다.
TableColumnModel cm = new DefaultTableColumnModel() {
    boolean first = true;
    public void addColumn(TableColumn tc) {
        // 첫번째 열은 무시한다.
        if (first) {
            first = false;
            return;
        }
        // 열의 크기를 설정한다.
        tc.setMinWidth(100);
        super.addColumn(tc);
    }
};
```

DefaultTableColumnModel를 상속하여 두 개의 TableColumnModel을 생성하는 곳이다. 이름 없는 내부 클래스(Anonymous Inner Class)를 사용하였다. 내부 클래스에 대해 모르는 독자는 아래 주소를 참조하기 바란다.

```
http://java.sun.com/products/jdk/1.1/docs/guide/innerclasses/spec/innerclasses.doc.html
```

두 객체는 각각 헤더 테이블과 데이터 테이블을 위한 TableColumnModel로 사용된다. JTable이 생성될 때 addColumn() 메쏘드를 통해 모델로부터 각각의 열들이 이 곳에 등록된다. 이때 헤더 테이블용 TableColumnModel에서는 첫 번째 열만 등록하고 나머지는 모두 무시한다. 그리고 데이터 테이블용 TableColumnModel에서는 반대로 첫 번째 열만 무시한다. 그 다음 이렇게 만들어진 TableColumnModel 객체를 이용하여 두 개의 테이블을 만든다.

```
JTable jt = new JTable(tm, cm);
// 행 해더를 설정한다.
JTable headerColumn = new JTable(tm, rowHeaderModel);
jt.createDefaultColumnsFromModel();
headerColumn.createDefaultColumnsFromModel();
```

createDefaultColumnsFromModel() 메쏘드는 모델로부터 열들을 등록하게 한다. 직전에 보았던 TableColumnModel의 addColumn() 메쏘드를 호출하는 계기가 된다. 아주 중요한 부분이 남아있다. 테이블이 두 개이기 때문에 한 테이블의 어느 행이 선택된다거나 하는 등의 이벤트에 두 테이블이 일치된 반응을 보일 수 있도록 동일한 SelectionModel을 사용하도록 해야한다.

```
jt.setSelectionModel(headerColumn.getSelectionModel());
```

마지막으로 두 개의 테이블을 하나의 컨테이너에 담아야 한다. 우리는 행 헤더 부분의 세밀한 컨트롤을 위해 JViewport를 사용하였다.

```
JViewport jv = new JViewport();
jv.setView(headerColumn);
// 컨테이너(스크롤 패인)에 행 헤더와 데이터 테이블을
// 직접 넣어주어야 한다.
JScrollPane jsp = new JScrollPane(jt);
// JScrollPane의 RowHeader로서 우리의 행 헤더를 설정한다.
jsp.setRowHeader(jv);
```

사실 행 헤더 예제는 상당히 어려운 편에 속하는 응용이다. 필자가 지금까지 설명한 내용들을 이용하기는 했지만 이러한 아이디어(어찌 보면 잔머리라 생각되기도 한다.)를 생각해 내는 것 자체가 테이블 시스템에 통달하지 않고는 어렵기 때문이다. 그렇지만 일반적으로 2차원 형태의 데이터는 모두 행 헤더가 필요하다. 아마 언젠가는 구구단 이외에 사용할 일이 분명히 있을 것이다. 더 좋은 해결책은 테이블 API에 행 헤더가 추가되는 것이겠지만 언제가 될지 알 수 없으므로 급한 대로 필자가 소개한 방법을 사용하기 바란다.

## 3. 테이블의 모델로 파이 차트 만들기

엑셀 프로그램을 사용해보면 표로 작성된 데이터를 그래프로 변환할 수 있는 기능이 있다. 표에서 사용되던 데이터를 가져와서 그림으로 표현하는 것이다. 스윙의 테이블로도 동일한 기능을 제공할 수 있다. 스윙의 대부분의 컴포넌트들이(JTable 포함) 데이터를 액세스하는 모델(Model) 부분과 데이터를 화면에 표시하는 뷰(View)로 나뉘어 있기 때문이다. MVC 모델의 가장 큰 장점 중에 하나는 하나의 모델(데이터)에 대해 다양한 뷰를 가질 수 있다는 것일 것이다. 우리도 그러한 분리 구조를 활용하여 테이블에 파이차트를 제공하는 프로그램을 만들어볼 것이다. [화면 6]은 프로그램의 실행 화면이다. 그럼 다양한 뷰를 위해 우리가 알아야 하는 것에는 어떤 것들이 있는지 하나하나 살펴보자.

가장 중요한 내용은 하나의 모델에 서로 다른 뷰를 사용한다는 것이다. [그림 3]에 이러한 구조를 나타내어보았다. 여러 뷰들은 모두 테이블의 모델로부터 데이터를 가져와서 자신만의 방식으로 화면에 나타내게된다. 이때 중요한 것은 뷰들이 테이블의 모델로부터 TableModelEvent를 받는 다는 것이다. 만약 테이블에 새로운 행 또는 열이 추가되거나 테이블의 데이터 값이 변경된다면 테이블 모델은 모든 뷰들에게 값이 변경되었음을 알리게 된다. 각각의 뷰들은 모델로부터 나라온 TableModelEvent를 이용하여 구체적으로 어떠한 변경이 있었는지를 알 수 있다. 모델로부터 변경 사항을 알고자 하는 객체는 TableModelListener 인터페이스를 구현해야 한다. 다음 코드는 모델의 setValueAt() 메쏘드다.

```
public void setValueAt(Object value, int row, int col) {
    data[row][col] = (String)value;
    fireTableRowsUpdated(row,row);
}
```

setValueAt() 메쏘드는 사용자가 테이블의 데이터를 변경했을 때 호출되는 모델의 메쏘드다. 이 메쏘드는 먼저 모델의 실제 데이터 값을 변경시킨 다음 fireTableRowsUpdated()를 통해 모든 뷰들에게 데이터의 변경이 있었음을 알린다. 참고로 fireTableRowsUpdated() 메쏘드는

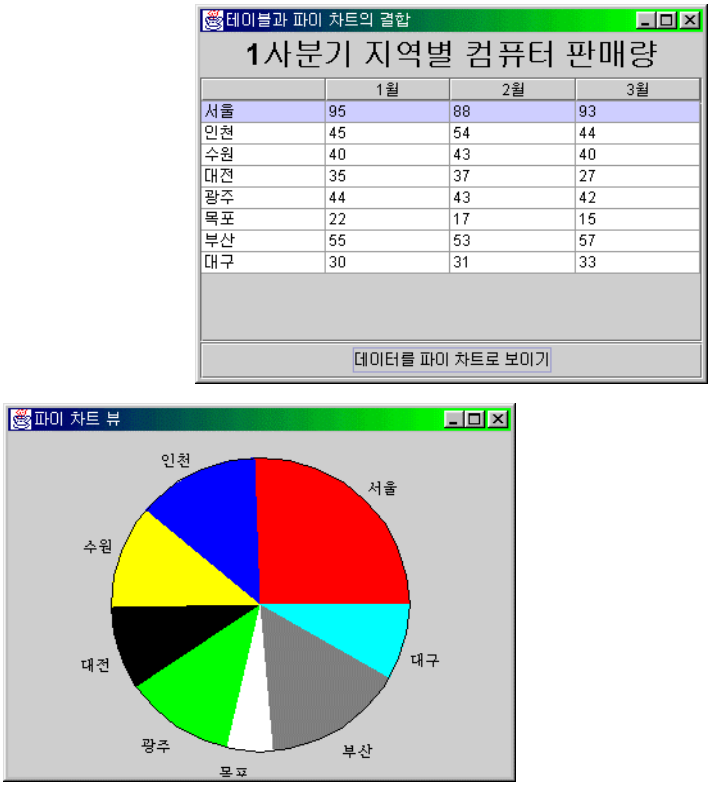
AbstractTableModel 클래스에서 찾을 수 있다.

그럼 뷰 부분을 보자. 물론 뷰는 TableModelListener 인터페이스를 구현해야 한다. TableModelListener 인터페이스에는 유일하게 tableChanged() 메쏘드가 존재한다. 이 메쏘드를 통해 모든 뷰들은 테이블의 데이터가 변경되었다는 것을 알아차리게 된다.

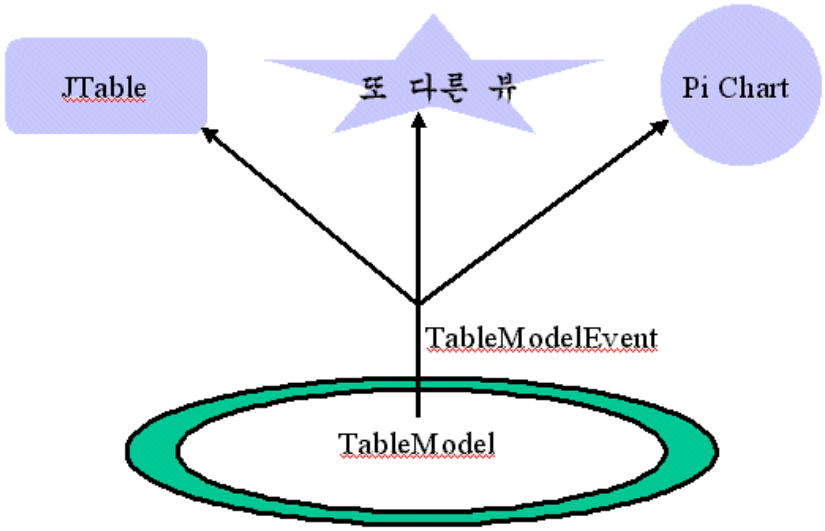
```
public void tableChanged(TableModelEvent tme) {
    updateLocalValues(tme.getType() != TableModelEvent.UPDATE);
}
```

위의 코드는 파이차트 뷰에 있는 tableChanged() 메쏘드이며 이 메쏘드는 파이차트의 갱신을 다시 updateLocalValues() 메쏘드에 모두 위임하게 된다. updateLocalValues()의 매개변수는 테이블의 구조(Structure) 자체가 변경되었는지(true일 때) 아니면 단순히 데이터 값이 변경되었는지를 (false일 때) 구분하기 위해서다. 단순히 데이터 값만 병경되었음에도 불구하고 파이차트의 구조 자체를 바꾸는 것은 효율적이지 못하기 때문이다.

[화면 7] 테이블과 파이차트의 결합




[그림 3] 테이블 모델과 다양한 뷰들




파이차트가 스윙에서 직접 지원되지 않기 때문에 우리의 예제에서는 새로운 Lightweight 컴포넌트를 개발하였다. Lightweight 컴포넌트를 만드는 방법은 선(Sun)사의 사이트나 튜토리얼을 통해 익히기 바란다. 여기서는 테이블의 모델을 테이블이 아닌 전혀 다른 컴포넌트(여기서는 파이차트)에도 이용할 수 있음을 아는 것으로 만족하자. 테이블 시스템의 유연한 구조는 스윙 컴포넌트들의 작동 메커니즘인 MVC 모델의 연장선상에서 이해할 수 있다. 방금 살펴보았던 것처럼 모델과 뷰를 독립시킴으로써 시스템에 훨씬 많은 유연성을 제공할 수 있다. 게다가 테이블은 각각의 셀들을 위한 셀 표시기와 셀 편집기마저도 사용자가 쉽게 교체할 수 있게 디자인되었다.

# 4. 테이블 리뷰 및 정리

지금까지 길다면 긴 글을 통해 스윙이 제공하는 테이블에 대해 알아보았다. 독자들도 테이블이 상당히 복잡한 컴포넌트라는 것을 알았을 것이다. 그러나 서두에도 말했지만 테이블 시스템의 구조와 작동 메커니즘에 대해 잘 알고 있으면 그 응용 범위는 아주 넓다. 이 글에서 그러한 응용의 모든 것을 제시하지는 못했지만 몇몇 중요한 응용 사례들을 살펴보았다. 단지 테이블이 제공하는 API 전체를 다 다루지 못했던 점과 예제 소스코드 한줄한줄에 대한 설명을 모두 할 수 없었던 것이 좀 아쉬움으로 남는다. 그러나 필자가 계속 강조하듯이 스윙에서의 테이블이 어떠한 구조와 메커니즘을 가지고 있는지만 제대로 이해했다면 나머지 세밀한 사항에 대해서는 필요할 때 그때그때 표준 API Document등을 통해서 얼마든지 해결할 수 있으리라 믿는다. 다음 달에는 스윙 컨테이너의 내부 구조에 대해 살펴보겠다.

# 2. 스윙 컨테이너의 비밀을 벗겨라

여러분은 오늘 이 지면 위에서 또 하나의 신비(?)와 만나게 된다. 바로 스윙 컨테이너들이 깊이 간직하고 있던 비밀이 그것이다. 스윙 컨테이너들은 그 내면에 독특한 구조를 간직하고 있다. 어찌 보면 약간은 복잡해 보이는 그러한 내부 구조가 스윙 컨테이너에게 얼마나 많은 확장성(Extensibility)과 유연성(Flexibility)을 제공하는지를 이 글 속에서 발견하게 될 것이다.

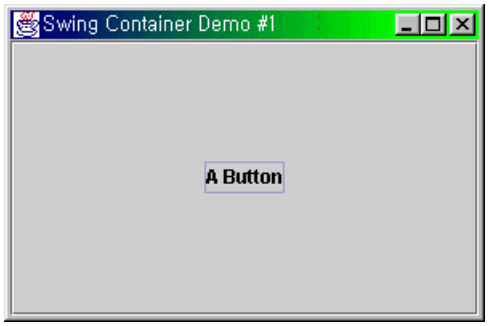
지난번에 필자가 스윙의 테이블 시스템에 대한 얘기를 하면서 계속 강조하던 것이 있었다. 바로 구조(Architecture)를 살펴보라는 것이었다. 스윙 컴포넌트가 제공하는 메쏘드 하나 하나를 알고 있는 것보다는 컴포넌트가 어떠한 구조로 이루어져 있고 또 그런 구조가 주는 장점은 무엇인지를 아는 것이 훨씬 효과적인 접근 방법이다. 구조에 대해 확실히 파악하고 있으면 구체적인 메쏘드에 대한 내용은 대부분 자연스럽게 따라오게 되어있기 때문이다. 따라서 이번 기사에서도 스윙 컨테이너들이 어떠한 구조를 내면에 감추고 있으며 그러한 구조가 주는 확장성과 유연성에 대해 음미해보는데 초점을 맞춰주기 바란다. 그럼 컨테이너 해체 작업을 시작하겠다.

## 1. 번거로운 작업(?)

필자가 처음 스윙을 접했을 때 다음과 같은 예제 프로그램을 보았다. 사실은 여러 스윙 예제를 보면서 기존의 AWT와 거의 다르지 않다는 것에 대해 안도감을 느꼈었다. 한 군데만 빼고.

```
import javax.swing.*;
public class ContainerDemo_1 {
    public static void main(String s[]) {
        JFrame frame = new JFrame("Swing Container Demo #1");
        JButton button = new JButton("A Button");
       // 번거로운 작업(?)
        frame.getContentPane().add(button);
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

[화면 1] JFrame을 사용하는 간단한 예제



필자가 의아해 했던 부분은 바로 다음 부분이다.

```
frame.getContentPane().add(button);
```

AWT에서라면 frame.add(button)라고 했을텐데 왜 getContentPane()라는 메쏘드를 한번 더 거쳐가야 하나? 아마 이 글을 읽고 있는 독자도 분명히 그런 궁금증을 가지고 있었으리라 생각된다. 참고로 frame.add(button)라고 고쳐서 프로그램을 실행시켜 보면 다음과 같은 에러 메시지가 뜬다.

```
Exception in thread "main" java.lang.Error: Do not use javax.swing.JFrame.add()
use javax.swing.JFrame.getContentPane().add() instead
    at javax.swing.JFrame.createRootPaneException(JFrame.java:327)
    at javax.swing.JFrame.addImpl(JFrame.java:349)
    at java.awt.Container.add(Container.java:206)
    at ContainerDemo_1.main(ContainerDemo_1.java:8)
```

낚시 줄을 잡아 올리듯이 getContentPane()의 리턴 값을 추적해 보면 그것이 Container라는 것을 알 수 있다. 그럼 프레임 안에 또 다른 어떤 컨테이너가 숨어 있는 것일까? 왜 그런 것이 숨어있지? 보통 이 정도 선에서 더 이상 끌어당기지 못하고 낚시 줄을 놓았을 것이다. 필자도 그랬으니까. 그럼 이제부터 그 궁금증을 속시원하게 낱낱이 풀어주겠다.

## 2. 스윙 컨테이너의 개념적 구조

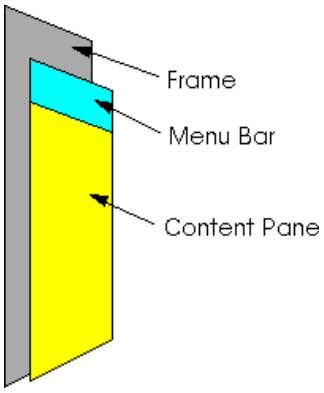
이 글에서 스윙 컨테이너라 함은 JApplet, JDialog, JFrame, (JInternalFrame), 그리고 JWindow와 같은 최상위 컨테이너(Top Level Container)를 말한다. 최상위 컨테이너 자신은 다른 컴포넌트를 담기만 할 뿐 자신이 다른 컨테이너에 담겨지지는 못한다. 그렇게 된 이유 중에는 이들이 모두 중량 컴포넌트(HeavyweightComponent)를 확장한다는 것도 한 몫을 하고 있다. (중량 컴포넌트에 대해서는 박스기사를 참조하기 바란다.) 스윙에서 JInternalFrame을 제외한 모든 최상위 컨테이너들은 대응되는 AWT의 최상위 컨테이너를 확장(extends)한다. 예를 들어 JFrame 은 AWT의 Frame을 확장하고 있다.([그림 5]와 [그림 6] 참조) 박스 기사를 보면 중량 컴포넌트들은 많은 단점을 가지고 있다. 그런데 스윙에서마져도 최상위 켄테이너들을 중량 컴포넌트에 의존하게 만든 이유는 운영체제의 자원을 최대한 이용하기 위해 어쩔 수 없는 선택이었다.

JInternalFrame은 최상위 컨테이너는 아니지만(따라서 중량 컴포넌트도 아니다.) 다른 최상위 컨테이너들과 같은 구조를 하고있으므로 설명의 편의를 위해 여기서는 최상위 컨테이너로 취급한다. 또한 최상위 컨테이너들이 모두 같은 구조를 가지고 있기 때문에 각각을 따로 설명하지 않고 일단 우리에게 익숙한 JFrame을 대상으로 설명을 해 나가겠다. 그러니 여기서 설명하는 내용이 JFrame에만 국한된 내용이 아님을 염두에 두기 바란다.

[그림 1]에 JFrame의 개념적인 구조를 그려보았다. 그림을 보면 JFrame에는 프레임 자신 이외에 Menu Bar와 Content Pane(이하 컨텐트창)이라는 영역으로 나뉘어진 또 하나의 창이 있음을 알 수 있다. Menu Bar 영역은 메뉴바를 위한 공간이고 컨텐트창은 프레임에 어떤 컴포넌트를 담을

때 사용하는 곳이다. 우리가 frame.getContentPane()이라고 하면 바로 이 컨텐트창을 얻어오게 되는 것이다.

[그림 1] JFrame의 개념적 구조



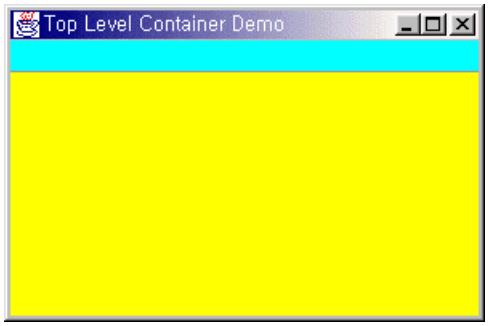

[리스트 1]에 이러한 구조를 보여주는 예제를 실어보았다. 메뉴바에는 하늘색(Cyan) 메뉴바를 설치하고 컨텐트창에는 노란색의 레이블(JLabel)을 설치한다.

[리스트 1] 최상위 컨테이너의 개념적 구조를 보여주는 예제

```
import java.awt.*;
import javax.swing.*;

public class TopLevelContainerDemo extends JFrame {

    public TopLevelContainerDemo() {
        super("Top Level Container Demo");

       // 하늘색 메뉴바를 붙인다.
        JMenuBar menuBar = new JMenuBar();
        menuBar.setOpaque(true);
        menuBar.setBackground(Color.cyan);
        menuBar.setPreferredSize(new Dimension(300, 20));
        setJMenuBar(menuBar);

       // 컨텐트창에 꽉 차는 노란 레이블을 붙인다.
        JLabel label = new JLabel("");
        label.setOpaque(true);
        label.setBackground(Color.yellow);
        getContentPane().add(label, BorderLayout.CENTER);

        setSize(300, 200);
        setVisible(true);
    }
```

```
    public static void main(String args[]) {
        new TopLevelContainerDemo();
    }
}
```

[화면 2] 최상위 컨테이너의 개념적 구조를 보여주는 예제




여기까지 스윙 컨테이너에 대해 간략히 알아보았다. 그러나 우리가 방금 살펴본 것이 전부는 아니다. 우리가 방금 살펴본 구조는 이해의 편의를 위해 단순화시킨 구조일 뿐이다. 실제 스윙 컨테이너는 좀 더 복잡한 구조를 내부에 숨겨놓고 있다.

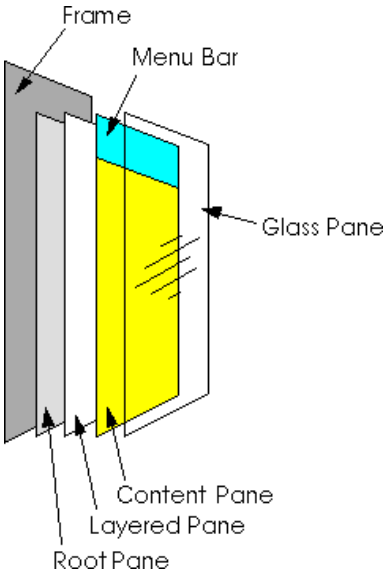
## 3. 진짜 컨테이너 나와라!

[그림 2]가 스윙 컨테이너의 진짜 구조를 나타내는 그림이다. 아주 여러 겹의 창으로 이루어 진 것에 놀랐을 것이다. 우리가 앞서 보았던 메뉴바와 컨텐트창도 볼 수 있다. 그런데 이 많은 창들 중에 실제로는 프레임을 제외한 모든 창이 RootPane(이하 루트창)이라고 하는 창으로 묶이게 된다. 루트창 안에 프레임을 제외한 다른 창들이 모두 포함되어 있기 때문이다. 따라서 JFrame은 프레임 자신과 루트창으로 이루어 졌다고 말할 수 있다. [그림 3]은 루트창만을 따로 떼어내어 본 것이다.

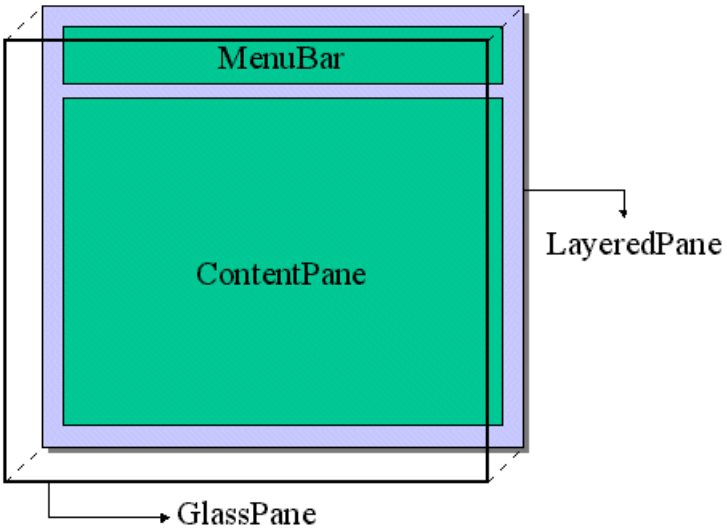
루트창이 바로 스윙 컨테이너의 비밀을 간직하고 있는 핵심이 되는 창이다. 이 글도 바로 이 루트창을 위한 글이라 할 수 있다. 스윙의 최상위 컨테이너들은 모두 이 루트창이라는 것을 안에 품고 있는데 루트창을 품고 있는 컨테이너들은 모두 RootPaneContainer라고 하는 인터페이스를 구현(implements)하고 있다. [표 1]에 RootPaneContainer 인터페이스를 정리하였다.

[그림 3]에서 볼 수 있는 것처럼 루트창은 다시 두개의 창으로 나뉘어져 있다. 하나는 LayeredPane이고 다른 하나는 GlassPane이다. LayeredPane은 다른 컴포넌트나 창들을 배치하는데 사용되는 일종의 컨테이너용 창이다. 이 창에서 특이한 점은 이름에서도 알 수 있듯이 층(Layer)의 개념을 가지고 컴포넌트들을 배치한다는 것이다. 메뉴바와 컨텐트창도 LayeredPane의 어느 특정한 층에 배치된다. LayeredPane과 GlassPane은 살펴 볼만한 특징을 가지고 있기 때문에 좀 더 자세히 알아보겠다.

[그림 2] JFrame의 실제 구조 <--- 이 그림은 3차원 효과가 잘 드러나야합니다.




[그림 3] JRootPane의 구조 <--- 이 그림은 3차원 효과가 잘 드러나야합니다.
<--- 그리고 GlassPane이라는 창을 유리처럼
<--- 보이게 할 수 없나요?




[표 1] RootPaneContainer 인터페이스의 메쏘드들

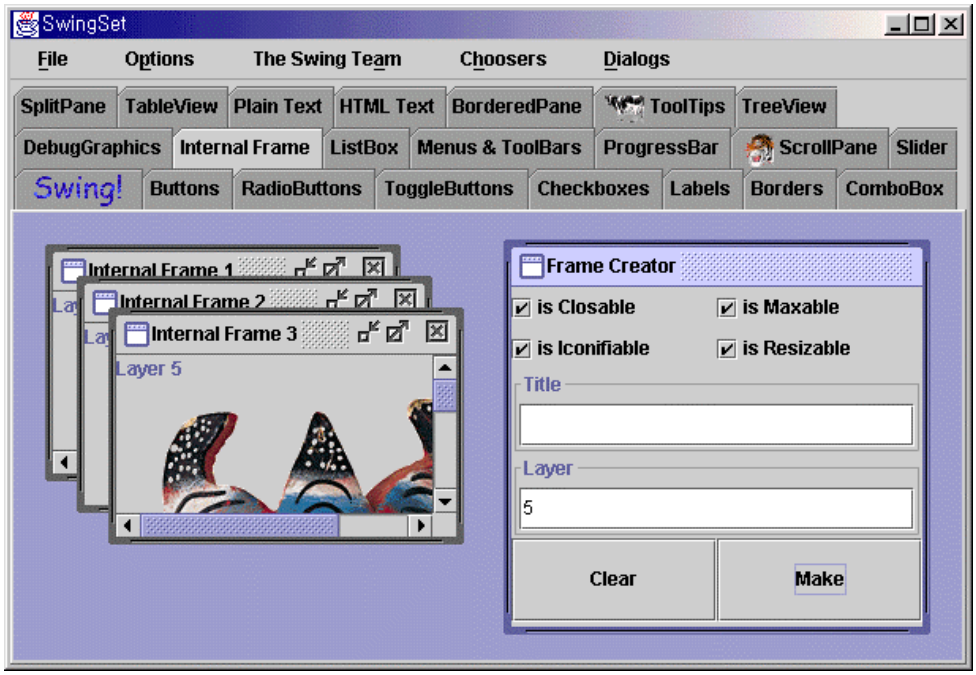
| 메쏘드 | 설명 |
|---|---|
| getContentPane()<br>setContentPane() | ContentPane을 얻거나 설정한다. |
| getGlassPane()<br>setGlassPane() | GlassPane을 얻거나 설정한다. |
| getLayeredPane()<br>setLayeredPane() | LayeredPane을 얻거나 설정한다. |
| getRootPane() | RootPane을 얻는다. |

# 4. GlassPane은 유리창?

GlassPane은 유리창처럼 보인다. 속이 훤히 비치는 유리창. GlassPane은 루트창의 가장 표면에 위치해 있는데([그림 3]) 완전히 투명하기 때문에 평소에는 전혀 보이질 않는다. 마치 JPanel에 setOpaque(false); setVisible(false); 이렇게 두 함수를 호출한 것과 같다. 그럼 GlassPane은 어디에 쓰이는 것일까? 스윙에서 GlassPane은 용도는 꼭 한군데 밖에 없다. 그것은 InternalFrame을 위한 용도다.

GlassPane은 루트창의 전체를 덮고 있다. [그림 3]을 보면 컨텐트창은 물론이고 메뉴바도 GlassPane에 의해 덮여있다는 것을 알 수 있다. 따라서 GlassPane에서 마우스 이벤트처리를 가로채게 되면 그 아래에 있는 어떤 창에도 마우스 이벤트는 전달되지 않는다. GlassPane의 이러한 특성 때문에 여러 개의 InternalFrame을 사용하는 멀티 도큐먼트(Multi-Document) 어프리케이션에서 GlassPane이 사용된다.

[화면 3] 멀티 도큐먼트 창



[화면 3]과 같은 멀티 도큐먼트 어프리케이션이 있다고 하자. 멀티 도큐먼트 어프리케이션의 내부창에 사용되는 클래스가 바로 JInternalFrame 클래스다. 하나의 메인창 안에는 여러 개의 내부창이 있게 마련인데 내부창 중에 오직 하나의 창만이 현재 선택된 창이 될 수 있다. 그런데 사용자가 다른 내부창으로 선택을 옮기고 싶으면 선택하고자하는 해당 창을 클릭하는 것이 보통이다. 이때 마우스 클릭은 순전히 창을 선택하고자 하는 의도의 클릭이다. 따라서 새로 선택된 창 안의 어느 곳에 마우스 클릭이 있을지라도 그 창이 선택되는 이외의 행동을 해서는 안된다. [화면 3]에서 왼쪽에 있는 창으로 선택을 옮기려고 왼쪽 창의 스크롤바에 마우스 클릭이 있었을지라도 창 안의 내용물이 스크롤되어서는 안된다. 스윙에서는 이 문제를 GlassPane으로 해결한다.

일단 내부창이 비 활성화되면(선택되지 않으면) 그 내부창의 GlassPane을 보이게 한다.(setVisible(true)) (여기서 GlassPane이 메인창의 것이 아니라 내부창의 것이라는 것을 염두에 두자.) 이렇게되면 이제부터 마우스 이벤트는 모두 GlassPane이 받게된다. 그렇지만 아직 Opaque를 설정하지 않았기 때문에 여전히 GlassPane은 투명해 보일 것이다. 따라서 사용자의 입장에서는 GlassPane이 존재하는지 전혀 눈치채지 못한다. 나중에 다시 그 내부창에 마우스 클릭이 발생하면, 즉 창이 선택되면 GlassPane은 단순히 자신을 다시 보이지 않게 하고 (setVisible(false)) 자신을 포함하고 있는 내부창이 선택 상태가 되도록 한다. 이것이 스윙에서 GlassPane의 용도다. 그럼 예제를 보면서 GlassPane의 존재를 확인해 보자.

[화면 4]에 예제 프로그램의 실행 화면이 있다. 왼쪽 창은 GlassPane이 보이지 않는 상태고 오른쪽 창은 보이는 상태다. 왼쪽 창의 "GlassPane 보이기" 버튼을 누르면 GlassPane이 나타나게 되어 있다. “GlassPane 감추기” 버튼은 원래부터 GlassPane에 담겨있었지만 자신을 담고 있는 GlassPane이 보이지 않는 상태에 있었기 때문에 왼쪽 창에서처럼 버튼이 화면에 나타나지 않고 있다가 나중에 GlassPane이 보이게될 때 그 버튼도 화면에 나타나게 된다.

GlassPane이 보이는 상태에서 창의 여기저기에 마우스를 클릭하면 창에 빨간 반점이 생긴다. 그런데 그 반점들은 GlassPane에 찍히는 것이기 때문에 그 아래에 있는 메뉴바나 다른 버튼들 위에서도 잘 찍히고 있다. 왼쪽 창에 있던 메뉴와 버튼들은 모두 GlassPane 아래의 컨텐트창에 담겨있는 버튼들이기 때문이다.

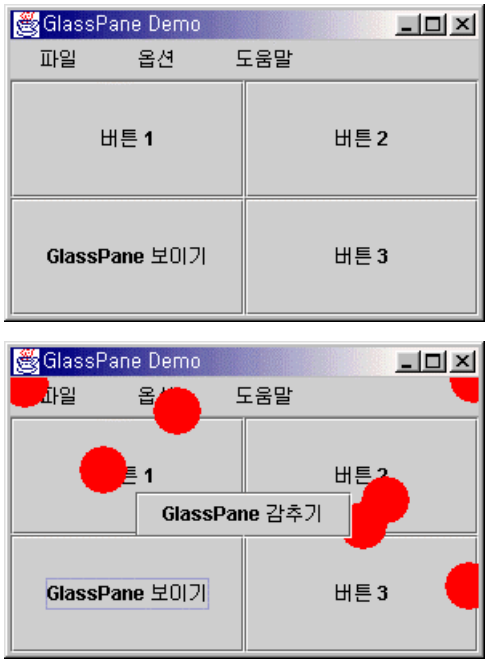
[리스트 2] GlassPane을 활용한 예제

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class GlassPaneDemo extends JFrame implements ActionListener {

    // 프레임의 GlassPane으로 사용할 컴포넌트
    MyGlassPane m_glassPane = new MyGlassPane();

    public GlassPaneDemo() {
        super("GlassPane Demo");
```

```
        // 이 버튼이 눌려지면 GlassPane이 나타나게 된다.
        JButton buttonShowGlass = new JButton("GlassPane 보이기");
        buttonShowGlass.addActionListener(this);

        // 프레임에 버튼들을 넣는다.
        getContentPane().setLayout(new GridLayout(0, 2));
        getContentPane().add(new JButton("버튼 1"));
        getContentPane().add(new JButton("버튼 2"));
        getContentPane().add(buttonShowGlass);
        getContentPane().add(new JButton("버튼 3"));

        // 메뉴를 설치한다.
        JMenuBar menuBar = new JMenuBar();
        menuBar.add(new JMenu("파일"));
        menuBar.add(new JMenu("옵션"));
        menuBar.add(new JMenu("도움말"));
        setJMenuBar(menuBar);

        // 프레임의 GlassPane을 설치한다.
        setGlassPane(m_glassPane);                    <----- [1번]

        setSize(300, 200);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        // GlassPane을 나타나게 한다.
        m_glassPane.setVisible(true);
    }

    public static void main(String[] args) {
        new GlassPaneDemo();
    }
}

class MyGlassPane extends JPanel implements ActionListener {

    // 붉은 반점의 위치
    private Point m_spotLocation;

    public MyGlassPane() {
        JButton button = new JButton("GlassPane 감추기");
        button.addActionListener(this);
```

```
            setLayout(new GridBagLayout());
            add(button);

            addMouseListener(new MouseAdapter() {  <----- [2번]
                public void mouseClicked(MouseEvent e) {
                    m_spotLocation = e.getPoint();
                    System.out.println(m_spotLocation);
                    repaint();
                }
            });
        }

        public void actionPerformed(ActionEvent e) {
            // GlassPane을 감춘다.
            setVisible(false);
        }

        public void paintComponent(Graphics g) {
            // 붉은 반점을 그린다.
            if (m_spotLocation != null) {
                g.setColor(Color.red);
                g.fillOval(m_spotLocation.x-15,  m_spotLocation.y-15,  30,
30);
            }
        }
}
```

[화면 4] GlassPane을 활용한 예제의 실행 화면



[리스트 2]의 [1번]처럼 JFrame의 setGlassPane(m_glassPane) 메쏘드로 루트창의 GlassPane을 설정할 수 있다.([표 1] 참조) [2번]에서처럼 GlassPane이 마우스 이벤트를 처리해버리면 그 아래층에 있는 다른 컴포넌트들에게는 마우스 이벤트가 전달되지 않는다는 것을 염두에 두기 바란다. 그럼 이번엔 GlassPane과 함께 루트창에 포함된 또 다른 창인 LayeredPane에 대해 살펴보자.

## 5. 층층이 쌓아올리는 LayeredPane

AWT만 다뤄왔던 사람이라면 창을 층층이 쌓는다는 것에 익숙하지 못할 것이다. 스윙에서는 컨테이너에 창이나 컴포넌트를 층의 개념을 가지고 쌓아올리는 것이 가능하다. 우리는 이미 앞에서 GlassPane이 루트창의 가장 위쪽에 올려져있는 창이라는 것을 배웠다. (별 설명이 없이도 그러한 층의 개념이 낯설지 않았었다면 독자는 분명 생각이 자유로운 사람일 것이다.) 그럼 본격적으로 창의 층(Layer) 개념에 대해 살펴보자. 예제를 먼저 보고 진행하는 것이 이해하는데 편할 것 같다.

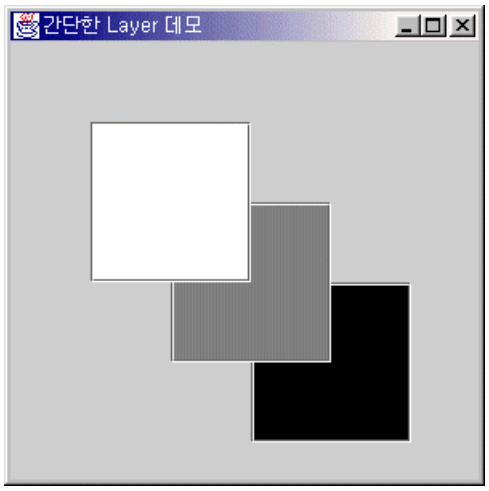
[리스트 3] LayerPane을 활용한 예제

```
import java.awt.*;
import javax.swing.*;

public class LayerDemo extends JFrame {

    public LayerDemo() {
        super("간단한 Layer 데모");

        JLayeredPane layeredPane = getLayeredPane();

        // 맨 위층 버튼
        JButton top = new JButton();
        top.setBackground(Color.white);
        top.setBounds(50, 50, 100, 100);

        // 중간 층 버튼
        JButton middle = new JButton();
        middle.setBackground(Color.gray);
        middle.setBounds(100, 100, 100, 100);

        // 맨 아래층 버튼
        JButton bottom = new JButton();
        bottom.setBackground(Color.black);
        bottom.setBounds(150, 150, 100, 100);

        // 버튼을 층별로 배열한다.
        layeredPane.add(bottom, new Integer(1));
```

```
            layeredPane.add(middle, new Integer(2));
            layeredPane.add(top, new Integer(3));

            setSize(300, 300);
            setVisible(true);
        }

        public static void main(String[] args) {
            new LayerDemo();
        }
}
```

[리스트 3]을 보면 LayeredPane에 세 개의 버튼이 담겨진다. 이때 버튼들은 각각 서로 다른 층에 담기게 되는데 예를 들어 bottom 버튼은 1번 층에 담겨진다. 높은 층에 담겨있는 컴포넌트들은 낮은 층에 담겨있는 컴포넌트들을 가릴 수 있다. 스윙에서 컴포넌트들이 그려질 때 낮은 층의 컴포넌트를 먼저 그리고 높은 층의 컴포넌트들을 그 위에 덮어 그리기 때문이다. ([화면 5] 참조)

[화면 5] LayerPane을 활용한 예제의 실행 화면




JLayeredPane 클래스는 컴포넌트들을 담기 위해 add()라는 메쏘드를 제공한다. 이 메쏘드는 JLayeredPane의 수퍼클래스인 Container 클래스로부터 상속받은 메쏘드인데 내부적으로는 addImpl()이라는 메쏘드를 호출하게 되어있다. addImpl()은 다음과 같이 정의된다.

```
protected void addImpl(Component comp, Object layer, int position)
```

매개변수 중 comp는 담으려고 하는 컴포넌트고 layer는 컴포넌트가 담길 층을 설정하는데 쓰인다. 매개변수 layer가 Object형의 객체를 받기 때문에 숫자를 직접 넣어주면 안되고 반드시 new Integer(3); 이런 식으로 Integer형 객체를 넘겨주어야 한다. 마지막 매개변수인 position은 같은 층 안에서 컴포넌트의 위치를 나타낸다. 위치라고 했지만 이것도 어차피 컴포넌트의 높낮이에 관계되는 위치를 말한다. 즉, position은 하나의 층 속에서 컴포넌드들의 높낮이를 설정하는 데 사

용되는 매개변수라고 말할 수 있겠다. add() 메쏘드에서 position(위치) 값이 컴포넌트의 높낮이를 결정하는데는 다음과 같은 2가지의 규칙이 있다.

1. position이 낮은 컴포넌트일수록 나중에 그려진다.
2. position이 -1이면 지금까지 컨테이너에 포함되어있던 컴포넌트들 중에서 가장 높은 position을 갖는 컴포넌트의 바로 다음 position(높이)에 컴포넌트가 위치하게 된다. 즉 이 컴포넌트가 가장 높은 position에 위치하게되며 가장 먼저 그려지게 된다.

스윙 개발자들이 왜 이렇게 구현했는지는 모르겠지만 position은 층(Layer)과는 정 반대 방향으로 작동하는 것 같다. 그렇지만 예를 보면 금방 이해할 수 있을 것이다. 우선 어느 한 층 안에 A, B, C 이렇게 세 개의 컴포넌트가 각각 0, 1, 2의 위치(position)에 놓여있다고 생각해보자.

| A | B | C | | | |
|---|---|---|---|---|---|

이때 D를 위치 1로 설정하여 추가하면 다음과 같이 될 것이다.

| A | D | B | C | | |
|---|---|---|---|---|---|

다시 E를 위치 -1로 추가하면 다음과 같이 된다.

| A | D | B | C | E | |
|---|---|---|---|---|---|

다시 F를 위치 5로 추가하면 다음과 같이 된다.

| A | D | B | C | E | F |
|---|---|---|---|---|---|

그럼 A, B, C, D, E, F 컴포넌트가 화면에 그려지는 순서는 어떨까? F, E, C, B, D, A 순으로 화면에 그려지게 된다. 즉, 가장 높은 position을 갖는 F가 가장 먼저 바닥에 그려지고 가장 낮은 position을 갖는 A가 맨 마지막에 그려진다.

위치(Position)는 어느 특정한 층(Layer) 안에서의 높낮이 관계라고 했었다. 따라서 위치 값이 높든 낮든 상관없이 낮은 층에 담겨있는 모든 컴포넌트들이 높은 층에 담겨있는 어떤 컴포넌트보다도 먼저 그려진다는 것을 잊지 말도록 하자.

JLayeredPane 클래스는 특수 용도로 몇 개의 층을 미리 찜(?)해 놓았다. [그림 4]와 [표 3]에 용도와 높낮이 관계를 정리해 놓았다. 표를 보면 각 층들의 높이 값도 볼 수 있다. 그러나 그 값에 의존하여 프로그램을 만들지 않도록 하는 것이 좋을 것이다. 이 값들은 앞으로 언제든지 변할 수 있기 때문이다. (스윙이 확장되어 새로운 용도의 창이 추가될 수 있다.) 그러니 층의 높이 값을 직접 사용하지 말고 “DEFAULT_LAYER+3” 이런 식으로 상수 값을 이용하거나 자신이 특별한 상수 값을 정하여 사용하기 바란다. 그럼 층과 위치 개념을 잘 보여주는 예제를 하나 살펴보며 LayeredPane의 설명을 마치겠다.

[리스트 4] LayeredPane의 층과 위치에 관한 예제

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

public class LayeredPaneDemo extends JFrame {

    /** 층의 이름 */
    private String[] m_layerNames = {
        "노란색 층",
        "보라색 층",
        "하늘색 층",
        "빨간색 층",
        "푸른색 층"
    };

    /** 층별 색상 */
    private Color[] m_layerColors = {
        Color.yellow,
        Color.magenta,
        Color.cyan,
        Color.red,
        Color.green
    };

    /** 루트창의 LayeredPane으로 사용할 창 */
    private JLayeredPane m_layeredPane;

    /** 듀크(자바의 마스코트)의 이미지 */
    private JLabel m_dukeLabel;

    public LayeredPaneDemo()    {
        super("LayeredPane 데모 예제");

        // LayeredPane을 만들고 초기화한다.
        m_layeredPane = new JLayeredPane();
        m_layeredPane.setPreferredSize(new Dimension(300, 310));
        m_layeredPane.setBorder(BorderFactory.createTitledBorder(
                "마우스를 움직이시면 듀크가 따라 움직입니다."));
        m_layeredPane.addMouseMotionListener(new MouseMotionAdapter()
{
            public void mouseMoved(MouseEvent e) {
                m_dukeLabel.setLocation(e.getX()-32, e.getY()-32);
            }
        });
```

```
        // 첫번째 색상판의 위치
        Point origin = new Point(10, 20);

        // 색상판들의 간격
        int offset = 35;
        // 몇 개의 색상판을 LayeredPane의 서로 다른 층에 담는다.
        for (int i = 0; i < m_layerNames.length; i++) {
            JLabel label = createColoredLabel(m_layerNames[i],
                    m_layerColors[i], origin);
            m_layeredPane.add(label, new Integer(i));
            origin.x += offset;
            origin.y += offset;
        }

        // 듀크 이미지를 만들어서 LayeredPane에 담는다.
        ImageIcon icon = new ImageIcon("WavingDuke.gif");
        m_dukeLabel = new JLabel(icon);
        m_dukeLabel.setBounds(15, 225, icon.getIconWidth(),
                icon.getIconHeight());
        m_layeredPane.add(m_dukeLabel, new Integer(2), 0);

        // 조절판과 LayeredPane을 컨텐트창에 담는다.
        Container contentPane = getContentPane();
        contentPane.setLayout(new BoxLayout(contentPane,
                BoxLayout.Y_AXIS));
        contentPane.add(Box.createRigidArea(new Dimension(0, 10)));
        contentPane.add(createControlPanel());
        contentPane.add(Box.createRigidArea(new Dimension(0, 10)));
        contentPane.add(m_layeredPane);

        pack();
        setVisible(true);
    }

    /** 한 색상으로 된 판을 만들기 위해 JLabel을 사용한다. */
    private JLabel createColoredLabel(String text, Color color,
            Point origin) {
        JLabel label = new JLabel(text);
        label.setVerticalAlignment(JLabel.TOP);
        label.setHorizontalAlignment(JLabel.CENTER);
        label.setOpaque(true);
        label.setBackground(color);
        label.setForeground(Color.black);
        label.setBorder(BorderFactory.createLineBorder(Color.black));
        label.setBounds(origin.x, origin.y, 140, 140);
        return label;
```

```
    }

    /** 조절 판을 만든다. */
    private JPanel createControlPanel() {
        final JCheckBox onTop = new JCheckBox(
                "듀크를 현재 층(Layer)의 가장 위로 위치시킨다.");

        onTop.setSelected(true);
        onTop.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                if (onTop.isSelected())
                    m_layeredPane.moveToFront(m_dukeLabel);
                else
                    m_layeredPane.moveToBack(m_dukeLabel);
            }
        });

        final JComboBox layerList = new JComboBox(m_layerNames);
        layerList.setSelectedIndex(2);    //Cyan layer
        layerList.addActionListener(new ActionListener () {
            public void actionPerformed(ActionEvent e) {
                int position = onTop.isSelected() ? 0 : 1;
                m_layeredPane.setLayer(m_dukeLabel,
                        layerList.getSelectedIndex(), position);
            }
        });

        JPanel controls = new JPanel();
        controls.add(layerList);
        controls.add(onTop);
        controls.setBorder(BorderFactory.createTitledBorder(
                "듀크가 위치할 층의 색을 고르세요"));
        return controls;
    }

    public static void main(String[] args) {
        new LayeredPaneDemo();
    }
}
```

[화면 6]에 프로그램의 실행화면이 있다. 마우스를 여기저기로 움직이면 손을 흔들고 있는 듀크(자바의 마스코트, 우리 예제에서는 듀크가 레이블로 구현되어 있다.)가 마우스를 따라서 움직인다. 다섯 개의 색상 판은 각각 서로 다른 층에 담겨있고 듀크는 콤보박스에 선택되어있는 색상 판과 동일한 층에 담기게 된다. 체크박스가 체크되어있으면 듀크가 같은 층 안에서 가장 위쪽 position에 놓이게 되며 (position 값 자체는 가장 작을 것이다.) 체크되어 있지 않으면 같은 층 안에서 가장 아래쪽 position에 놓이게 된다. (position 값 자체는 가장 클 것이다.) [화면 6]에서 만약

체크가 되어있지 않았다면 듀크는 하늘색 층 아래로 숨겨질 것이다.

[화면 6] LayeredPane의 층과 위치에 관한 예제의 실행화면


# 6. 지식이 서말 이라도 정리해야 보배!

지금까지 스윙 컨테이너들의 내부구조를 해체하여 살펴보았는데 여기서 지금까지 배운 내용을 정리해보자. 스윙 컨테이너들(JApplet, JDialog, JFrame, JInternalFrame, JWindow)은 공통적으로 그 내부에 루트창(RootPane)이라는 창을 품고 있다. 루트창은 다시 두 개의 창을 품고 있는데 GlassPane과 LayeredPane이 그것이다.

GlassPane은 루트창의 맨 위에 위치하며 마치 유리창과 같아서 평소에는 외부에서 그 존재를 인식하지 못한다. GlassPane에서 특별히 염두에 두어야 할 것은 그것이 루트창 전체를 덮고있다는 것이다. JInternalFrame은 GlassPane의 이러한 특성을 이용하여 마우스 이벤트를 모두 차단할 수 있다고 했다.

LayeredPane은 창이나 컴포넌트를 층층이 쌓는데 사용된다. LayeredPane에서 컴포넌트들의 높낮이는 층(Layer)과 위치(Position)의 개념을 도입하여 설정한다고 했다. 메뉴와 컨텐트창도 LayeredPane의 어느 특정한 층에 담겨있다. 일반적인 프로그래머라면 최상위 컨테이너의 getContentPane() 메쏘드를 사용하여 컨텐트창을 얻어온 다음 이곳에 버튼이나 다른 창 또는 컴포넌트들을 넣게된다.

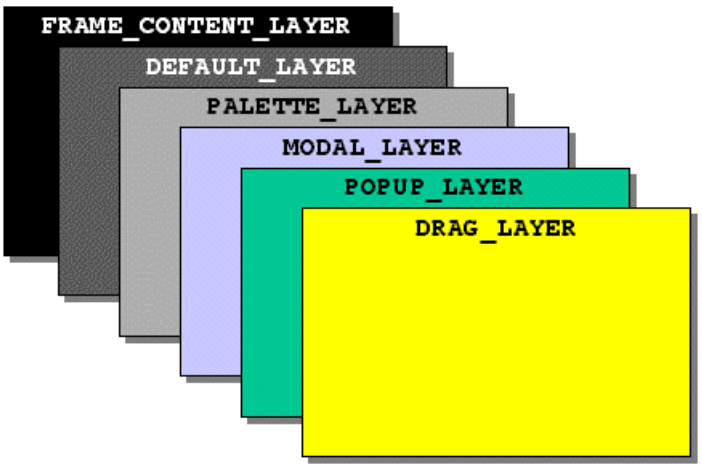
[그림 5]와 [그림 6]에 스윙 컨테이너들의 구조를 그려보았다. 그림에 나타나 있는 자바 클래스들(JApplet, JDialog, JFrame, JInternalFrame, JWindow, JRootPane, JLayeredPane)을 자바의 표준

API 도큐먼트를 찾아서 정리해 두고 [그림 2], [그림 3], [표 1], [표 2], [표 3] 등을 이용하여 나머지 내용들도 잘 정리해 두기 바란다.

[표 2] JRootPane의 속성(Attribute)들

| Type | 이름 | 설명 |
|---|---|---|
| JMenuBar | menuBar | 최상위 창의 메뉴. 최상위 창에서 디폴트로 메뉴가 생성되지 않으므로 처음엔 null값을 갖는다. |
| Container | contentPane | 일반적으로 최상위 창에 집어넣는 컴포넌트들은 모두 이 곳에 넣어지게된다. |
| JLayeredPane | layeredPane | 층(Layer)의 개념을 기반으로 컴포넌트들을 담는다. 메뉴바와 컨텐트창도 이곳에 담겨진다. |
| Component | glassPane | 메뉴바와 컨텐트창 전체를 덮는 창이다. 따라서 창의 클라이언트 영역으로 들어오는 모든 마우스 이벤트를 가로챌 수 있다. |

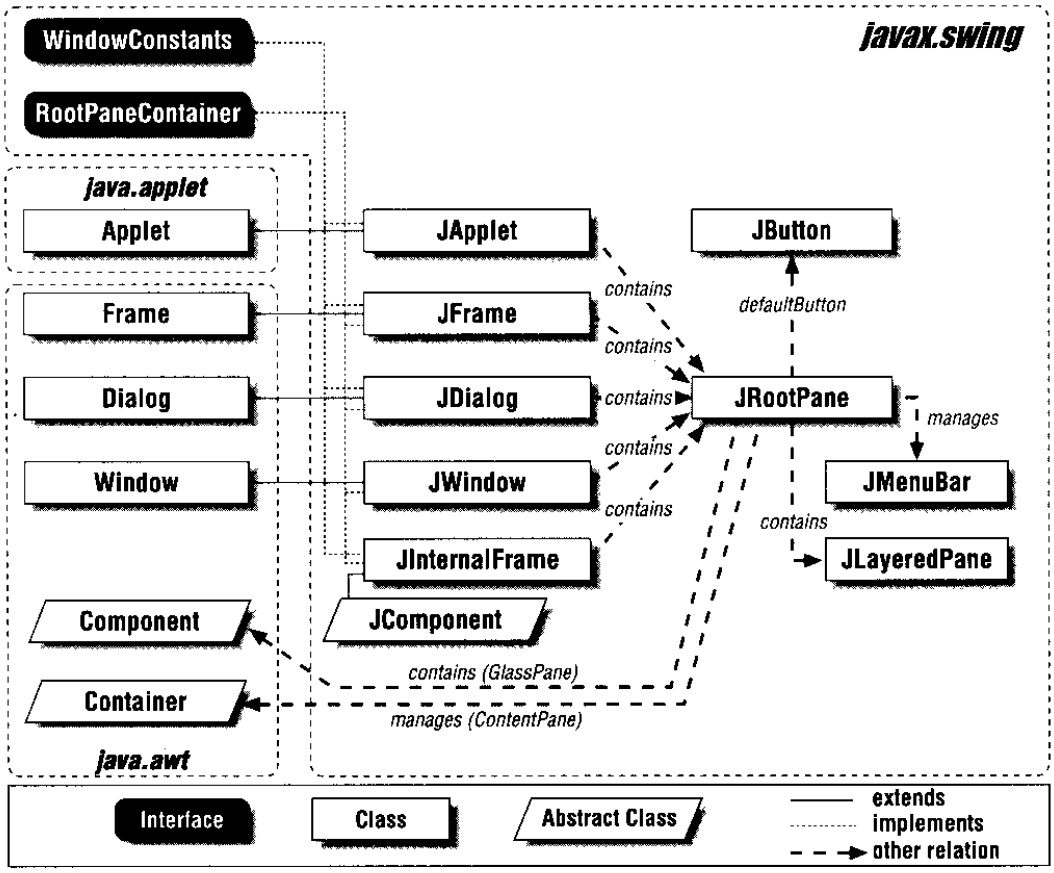
[그림 4] JLayeredPane에 미리 정해진 특수용도의 층들




[표 3] JLayeredPane에 미리 정해진 층의 용도와 층의 상수값

| 상수 | 값 | 용도 |
|---|---|---|
| FRAME_CONTENT_LAYER | -3000 | 오로지 메뉴바와 컨텐트창을 위해 사용된다. 루트창이 메뉴바와 컨텐트창을 이곳에 넣는다. 특별히 조작하지 않는다면 이 곳에 다른 컴포넌트가 들어가는 일은 거의 없을 것이다. |
| DEFAULT_LAYER | 0 | 대부분의 컴포넌트들을 위해 사용된다. 만약 넣고자 하는 컴포넌트가 위치를 명시하지 않으면 바로 이 층에 놓이게된다. |
| PALETTE_LAYER | 100 | 창 위에서 둥둥 떠다니는 툴바나 팔레트를 위해 사용된다. |
| MODAL_LAYER | 200 | 모달 창를 위해 사용된다. 모달 InternalFrame이나 모달 대화상자 등에 쓰인다. |
| POPUP_LAYER | 300 | 다른 모든 컴포넌트 위에 보여야할 것들이 놓이는 곳. 예를 들어 툴팁이나 팝업메뉴 등이 있다. |
| DRAG_LAYER | 400 | 창 위에서 어떤 물체를 드래그 할 때 사용된다. 드래그하고 있는 물체가 항상 다른 물체들의 위에 있도록 해준다. |

[그림 5] 스윙에서 최상위 컨테이너들의 구조



[그림 6] 스윙에서 최상위 컨테이너들의 구조

필자가 독자들에게 조그마한 선물(?)을 하나 준비했다. 본 기사와는 약간 동떨어진 내용이지만 연재 전체의 주제를 벗어나지는 않을 것 같아 여기에 싣는다. 스윙은 참 잘 정의된 라이브러리다. 그러한 사실을 밝히는 것이 이번 연재의 목적일지도 모르겠다. 아무튼 스윙에는 여기저기 쓸만한 기능들이 참 많이 있다. 그런데 규모가 제법 큰 라이브러리라서 그런지 스윙이 제공하는 그런 좋은 기능들을 존재하는지조차 모르고 사용하는 사람들이 참 많다. 이제 소개할 기능도 필자 나름대로는 참 쓸만한 기능이라 생각하고 있다. 하지만 이 기능을 사용한 소스를 여지껏 거의 보지 못했다. 심지어 스윙에 대한 잡지 기사에서도 보질 못했다. 안 사용한건지 못 사용한건지 모르겠지만 이제부터 꼭 사용해 보길 바란다. 스윙의 유연함과 탄탄한 구조를 다시금 확인할 수 있을 것이다.

# Action 인터페이스 활용하기

Action 인터페이스는 하나의 기능을 수행하기 위한 다양한 명령 전달 경로를 하나로 통합하게 해준다. 여러분이 워드프로세서를 개발하고 있다고 생각해보자. 아마 그 워드프로세서에는 문서를 저장하는 기능이 있을 것이다. 이때 사용자가 문서를 저장하는 방법에는 여러 가지가 있을 수 있다. 그 중에는 메뉴에서 “문서저장”을 선택하거나 또는 툴바에서 “문서저장” 버튼을 누르는 등의 방법이 있을 것이다. 뿐만 아니라 여러분의 워드프로세서가 팝업메뉴를 지원한다면 팝업 메뉴를 통해서도 저장기능을 활성화 할 수 있어야 한다. 혹시 그 워드프로세서가 사람의 말을 알아들을 수 있다면 “문서를 저장해!”라고 외치는 것도 방법이 될 수 있다.

이렇듯 하나의 기능을 활성화시키는 동기는 여러 가지 경로를 통해서 발생하게된다. 그럼 이제 Action 인터페이스가 이렇게 다양한 명령 전달 경로들을 어떻게 효과적으로 관리하며 중복되는 코드를 막아주는지 살펴보자.

Action 인터페이스는 javax.swing 패키지에 위치하며 다음과 같은 정의 형식을 갖는다.

```
public interface Action extends ActionListener { ... }
```

Action 인터페이스가 ActionListener 인터페이스를 상속한다는 것에 주목하자. ActionListener 인터페이스를 상속하기 때문에 Action 인터페이스도 actionPerformed() 메쏘드를 기본적으로 포함하게 된다. Action 인터페이스를 구현한(implement) 객체를 Action 객체라고 한다. 물론 이런 객체들은 actionPerformed() 메쏘드를 구현해야 한다. Action 객체들은 바로 이 actionPerformed() 메쏘드 안에 자신이 수행하고자하는 기능(Action)을 구현하게 된다.

다시 워드프로세서의 문서저장 기능에 대해 이야기해보자. 먼저 문서를 저장하는 기능을

DocumentSaveAction이라는 객체가 맡고 있다고 생각해 보면 DocumentSaveAction 클래스는 다음과 같이 정의될 것이다.

```
public class DocumentSaveAction implements Action {
    ...
    public void actionPerformed(ActionEvent) {
        // 문서를 저장한다.
        ...
    }
    ...
}
```

[그림 6]을 보면서 다음 글을 읽어나가기 바란다. 메뉴에서 “문서저장” 메뉴가 선택되었거나 툴바에서 “문서저장” 버튼이 눌렸을 때 메뉴아이템이나 버튼 객체는 DocumentSaveAction 객체에게 ActionEvent를 보내게 된다. 여러분이 예상할 수 있듯이 DocumentSaveAction 객체의 actionPerformed() 메쏘드를 호출함으로써 그러한 일이 이루어진다. 여러분은 바로 이 actionPerformed() 메쏘드 안에 문서저장 기능을 넣어두거나 또는 그러한 기능을 수행하는 메쏘드를 호출하면 된다. [그림 6]에 Action 객체와 메뉴아이템 또는 툴바버튼과의 관계가 나타나 있다.

[그림 6] Action 객체와 메뉴 또는 툴바버튼의 상호작용

Action 인터페이스는 "Design Patterns(Erich Gamma 외)“라는 유명한 책에 "Command Pattern"이라는 이름으로 실려있는 디자인 패턴을 자바로 옮겨놓은 것이다. 아직 그 책을 가지고 있지 않다면 필자가 꼭 권하고 싶은 책이다. 그 책에 Action(Command) 객체를 가지고 이미 실행한 기능을 실행 취소하거나 재개하는 방법 등도 자세히 나와있다. 사실 Action 인터페이스의 중요한 목적 중에 하나가 실행 취소(Undoable) 기능을 제공하는 것이다. 그러나 이 글을 벗어나는 내용이므로 여기서는 다루지는 않겠다. 그런 기능이 필요한 독자는 위의 책을 참조하기 바란다. 그럼 이쯤에서 예제 하나를 살펴보고 가자.

[화면 7] Action 인터페이스 데모 화면

[리스트 5] Action 인터페이스 데모 예제

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ActionDemo extends JFrame {

    public ActionDemo() {
        super("Action 인터페이스 데모");
```

```
        JMenuBar menuBar = new JMenuBar();
        JToolBar toolBar = new JToolBar();

        // 프레임에 메뉴바를 등록하고 파일 메뉴를 추가한다.
        setJMenuBar(menuBar);
        JMenu fileMenu = new JMenu("파일");
        menuBar.add(fileMenu);

        // 프레임에 툴바를 등록한다.
        getContentPane().add(toolBar, BorderLayout.SOUTH);

        // 파일 열기 및 파일 저장 Action 객체를 생성한다.
        DocumentOpenAction fileOpenAction = new DocumentOpenAction();
        DocumentSaveAction fileSaveAction = new DocumentSaveAction();

        // 파일 메뉴에 파일 열기 및 파일 저장 Action을 추가한다.
        fileMenu.add(fileOpenAction);          <------- [1번]
        fileMenu.add(fileSaveAction);          <------- [1번]

        // 툴바에 파일 열기 및 파일 저장 Action을 추가한다.
        JButton button;
        button = toolBar.add(fileOpenAction);  <------- [2번]
        button.setText(null);   // 버튼 이미지만 나타나게 함
        button = toolBar.add(fileSaveAction);  <------- [2번]
        button.setText(null);

        setSize(300, 200);
        setVisible(true);

        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent ev) {
                System.exit(0);
            }
        });
    }

    public static void main(String s[]) {
        new ActionDemo();
    }
}

/** 문서 열기 기능 클래스 */
class DocumentOpenAction extends AbstractAction {
```

```
    public DocumentOpenAction() {
        super("문서 열기", new ImageIcon("DocumentOpen.gif"));
    }

    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(null, "문서 열기를 선택 하셨습니
다.");
    }
}

/** 문서 저장 기능 클래스 */
class DocumentSaveAction extends AbstractAction {

    public DocumentSaveAction() {
        super("문서 저장", new ImageIcon("DocumentSave.gif"));
    }

    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(null, "문서 저장을 선택 하셨습니
다.");

        // 문서 저장 기능을 비활성화 시킨다.
        setEnabled(false);                    <------- [3번]
    }
}
```

[리스트 5]에서 [1번]과 [2번]이라고 된 곳을 보면 미리 생성해 놓은 Action 객체를 메뉴나 툴바에 등록하는 방법을 알 수 있다. JMenu와 JToolBar의 add() 메쏘드에 Action 객체를 넘겨주기만 하면 된다. 이렇게 하면 add() 메쏘드는 Action 객체가 등록된 JMenuItem 객체와 JButton 객체를 각각 리턴한다.

Action 객체의 setEnabled() 메쏘드를 이용하여 기능이 수행가능한지 가능하지 않은지를 설정할 수도 있다. [3번]에서처럼 setEnabled() 메쏘드에 false를 넘겨주면 [화면 7]의 오른쪽처럼 메뉴아이템과 버튼을 비 활성화할 수 있다.

스윙에서는 Action 인터페이스를 기본 구현해 놓은 AbstractionAction 클래스를 제공된다. 따라서 각각의 Action 클래스를 정의하는 것은 아주 쉽다. 앞서 살펴본 예제의 DocumentOpenAction과 DocumentSaveAction 클래스에서처럼 Action 인터페이스를 구현해 놓은 AbstractAction 클래스를 상속하고 필요한 메쏘드만 다시 오버라이드(Override)하면 된다. 수퍼클래스의 생성자에 문자열과 아이콘을 넘겨주면 그 문자열과 아이콘이 메뉴나 툴바에 공통으로 쓰일 것이다.

우리의 예제와는 달리 Action 클래스들은 도큐먼트 클래스의 내부에 정의되는 것이 마땅할 것이

다. 최소한 도큐먼트 객체를 액세스할 수 있는 위치에 있어야할 것이다. Action 객체가 문서를 저장하려면 도큐먼트 클래스를 직접 다룰 수 있는 위치에 있어야 하기 때문이다. 여러분이 워드프로세서를 만들 계획이라면 그렇게 하는 것이 좋겠다. 우리 예제에서는 복잡성을 피하기 위해 도큐먼트 클래스는 도입하지 않았다.

Action 인터페이스는 어떤 기능을 하나의 객체로서 다룰 수 있도록 해준다. 이러한 구조가 갖는 장점엔 무엇이 있을까? 먼저 앞에서 말했던 것처럼 Undoable한 기능을 제공할 수 있다고 했다. 다음으로 중복되는 코드를 막아준다. [그림 6]에서처럼 명령 전달 경로가 다양할지라도 기능의 수행은 Action 객체 한곳에서만 일어나므로 코드의 중복을 막을 수 있다. 또 다른 장점으로 하나의 기능과 관계된 여러 아이템들을 한 곳에 모을 수 있다는 것이다. 예를 들어 DocumentSaveAction 객체에 "문서 저장"이라는 문자열과 디스켓이 그려진 아이콘을 담아놓으면 메뉴에서든 툴바에서든 똑같은 문자열과 아이콘을 사용할 수 있게된다. 또 있다. [화면 7]의 오른쪽 그림을 보면 문서저장 메뉴와 버튼이 비 활성화되어(Disabled)있다. 한번 문서가 저장되었다면 문서에 변경이 없는 한 다시 저장할 필요가 없음을 나타내고 싶을 것이다. 이럴 때도 DocumentSaveAction 객체만 비 활성화시키면 자동으로 메뉴와 툴바에 비 활성화된 내용이 반영된다. 여기서 우리가 한가지 살펴볼 것이 있다. 그럼 메뉴나 툴바가 비 활성화되는 것은 어떠한 메커니즘으로 가능한 것일까?

[그림 6]을 보면서 PropertyChangeEvent에 대한 설명이 없었던 것을 기억하는가? 이 이벤트는 ActionEvent와는 반대 방향으로 이동하고 있음을 그림에서 알 수 있다. 만약 Action객체에 어떤 변화가 있었다면 메뉴아이템이나 툴바버튼에 PropertyChangeEvent가 보내진다. 따라서 DocumentSaveAction이 비 활성화되었다면 PropertyChangeEvent가 문서저장 메뉴아이템이나 문서저장 툴바버튼으로 전해지는 것이다. ActionEvent가 actionPerformed() 메쏘드에 의해 전달되듯이 PropertyChangeEvent는 propertyChange()라는 메쏘드를 통해 전달된다. 이 글에서 PropertyChangeEvent를 직접 사용하는 방법은 생략하겠다.

Action 인터페이스는 여러분의 어프리케이션의 명령 체계에 많은 유연성과 확장성을 부여해 줄 것이다. 사람의 말을 알아듣는 기능이 여러분의 어프리케이션에 추가되었다고 하자. 여러분은 단지 VoiceDetector.add(fileSaveAction)이란 코드를 추가함으로써 사람이 목소리로써 문서를 저장할 수 있도록 할 수 있을 것이다.

## 아는 것이 힘이다.

이번 글에선 필자가 가급적 말을 풀어쓰고 예제와 그림도 필요한 것 이상으로 많이 곁들이려 노력했다. 지난번 기사인 "테이블 파헤치기"보다는 훨씬 쉽게 글을 이해했을 것으로 믿는다. 단지 이번 기사에서는 실용적인 예제가 많지 않았던 것을 인정한다. 글을 읽어봐서 알겠지만 이번 기사의 주제로서 스윙의 컨테이너를 지목한 이유가 당장 사용할만한 것을 얻기 위해서가 아니다. 그냥 의심이나 궁금증 없이 getContentPane().add(aComponent); 이렇게 한 줄 쓰면 되는 것일지도 모른다. 그렇지만 그런 사람들에게 더 이상 어떠한 발전이 있을까? 최소한 그러한 코드에 궁금증을 느꼈던 독자들에게서 만이라도 이 글의 의미를 부여받을 수 있으면 필자는 족하다. 필자가

얼마 전에 어느 친구로부터 이런 질문을 받았다. “윈도우에 가득히 그림이 그려져 있고 그 중앙에 텍스트필드가 덩그렇게 홀로 떠있게 하려면 어떻게 해야해?” 필자 머리에는 GlassPane이 떠올랐다. “GlassPane에 텍스트필드를 올려놓고 그 아래에 그림을 그리면 되겠군...“ 독자는 어떤가? 필요는 발명의 어머니라고 했다. 그렇지만 필요하다고 해서 곧바로 발명되기는 어렵다. 그러나 필자처럼 곧바로 아이디어가 떠오를 수도 있다. 이것이 바로 아는 것의 힘이다.

- The End -

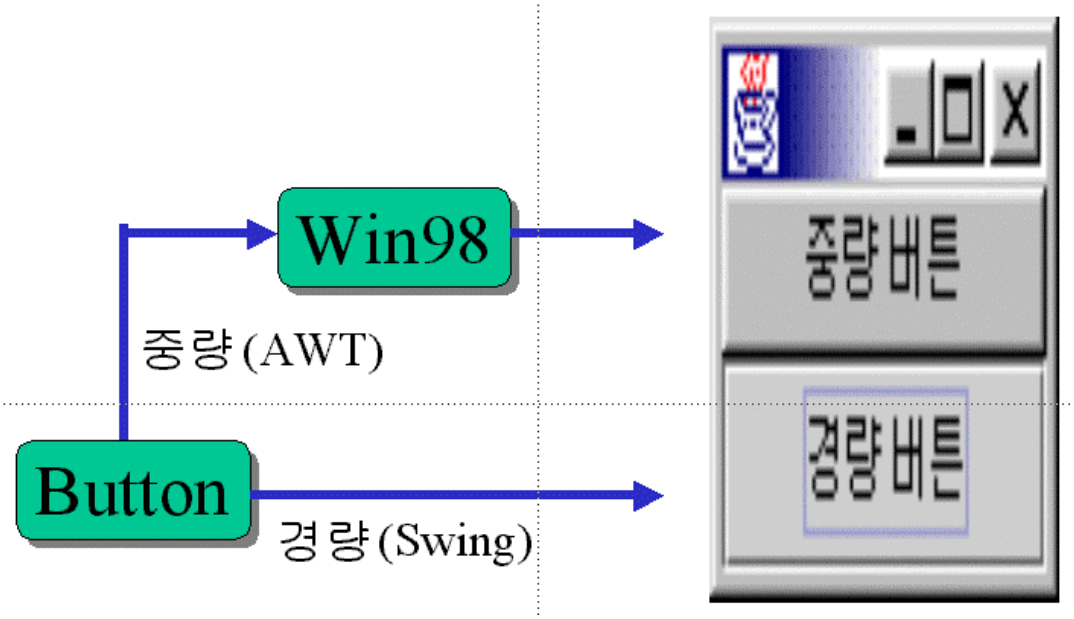
# 중량 컴포넌트와 경량 컴포넌트 (Box 기사용)

스윙의 세계에 입문하면서 독자도 경량 컴포넌트(Lightweight Component)라는 말을 많이 들어보았을 것이다. 그럼 경량 컴포넌트란 어떤 컴포넌트를 말하는 것일까? 본 기사와도 관련성을 가지는 내용이므로 잘 알아두도록 하자.

경량 컴포넌트는 그와 상대되는 개념인 중량 컴포넌트(Heavyweight Component)와의 비교를 통해 이해될 수 있다. 중량 컴포넌트란 자신의 화면 출력물을 플랫폼에서 직접 가져다 쓰는 컴포넌트를 말한다. 여기서 플랫폼이란 운영체제를 가리킨다. 예를 들어 버튼을 가지고 작업한다고 생각해보자. 자바에서 AWT 버튼을 사용한다면 버튼이 화면에 표시될 때 자바 시스템은 운영체제를 호출할 것이다. (참고로 AWT의 윈도우 컴포넌트들은 모두 중량 컴포넌트다.) 그리고는 운영체제로 하여금 화면에 버튼을 그리도록 할 것이다([그림 1]). (이렇게 운영체제로부터 가져다 쓰는 컴포넌트를 피어(peer)라고 부른다.) 윈도우계열 운영체제에서 API로 작업해 보았던 개발자라면 당연하다고 생각할 것이다. 이것이 중량 컴포넌트의 개념이다.

경량 컴포넌트는 중량 컴포넌트와는 다르게 행동한다. 경량 컴포넌트는 자신의 화면 출력을 자신이 직접 책임진다. 다시 버튼을 예로 들어보자. 스윙에서 버튼을 사용한다면 버튼이 화면에 표시될 때 버튼의 모든 내용이 스윙 라이브러리 자체에 의해 직접 화면에 그려지게 된다([그림 1]). (참고로 대부분의 스윙 컴포넌트들은 경량 컴포넌트다.)

[그림 1] 중량 컴포넌트와 경량 컴포넌트의 화면 표시 방법




중량 컴포넌트와 경량 컴포넌트가 이런 이름을 갖게 된 이유는 [그림 1]에서 알 수 있다. 중량 컴포넌트는 운영체제를 거쳐서 화면에 표시되게 된다. 그런데 운영체제를 호출하는 것 자체가 상당한 오버헤드를 요구한다고 한다. 그래서 쓸데없는 가비지가 더 많이 붙는다고 해서 중량이라

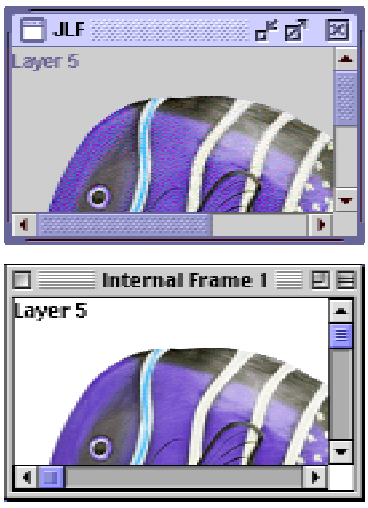
는 이름을 갖게 되었다. 경량 컴포넌트는 그 반대가 되겠다.

사실 중량 컴포넌트가 무엇이고 경량 컴포넌트가 무엇인지는 프로그래머들에게 그리 중요한 내용이 아니다. 프로그래머들에게는 이들의 행동양식이 어떤 차이를 보이는가가 정말로 중요한 관심거리일 것이다. [표 1]로 중량 컴포넌트와 경량 컴포넌트의 차이점을 정리해 두기 바란다.

[표 1] 중량 컴포넌트와 경량 컴포넌트의 비교

| 중량 컴포넌트 | 경량 컴포넌트 |
|---|---|
| 플랫폼에 따라 서로 다른 화면을 출력한다. (컴포넌트의 화면 표시를 운영체제에게 의존하므로 운영체제에 따라 운영체제 고유의 스타일로 화면에 표시된다.) | 플랫폼에 상관없이 거의 같은 화면을 출력한다. (화면 표시를 자체적으로 처리하므로 서로 다른 플랫폼에서도 같은 방법을 사용한다면 같게 보일 것이다.) |
| Look and Feel을 변경할 수 없다. (운영체제가 제공하는 Look and Feel 밖에는 선택의 여지가 없다.) | Look and Feel을 변경할 수 있다. (하나의 운영체제 내에서도 서로 다른 화면 표시 방법을 사용한다면 서로 다르게 보일 수 있다. [화면 1] 참조.) |
| 윈도우 영역이 항상 불투명하다. (운영체제가 제공하는 윈도우 자체가 원래 불투명하므로: AWT가 여러 운영체제의 공통 부분집합(Subset)으로 이루어졌다는 사실을 상기해 보면 원래 불투명한 이유를 알 수 있다.) | 윈도우 영역의 투명, 불투명을 선택할 수 있다. (투명하게 그리면 투명해 질 것이다.) |
| 윈도우 영역이 항상 네모다. (운영체제가 제공하는 윈도우 자체가 네모이므로) | 윈도우 영역이 항상 네모로 보일 필요는 없다. (투명 영역을 설정할 수 있으므로) |
| 윈도우가 항상 맨 위에 있는 것처럼 보인다. ([화면 2]를 보면 중량 컴포넌트는 항상 맨 위쪽에 놓인 것처럼 보인다. 따라서 중량 컴포넌트와 경량 컴포넌트를 섞어서 사용할 땐 이러한 점에 주의해야 한다.) | 윈도우의 Z-Order (높낮이) 설정이 가능하다. |

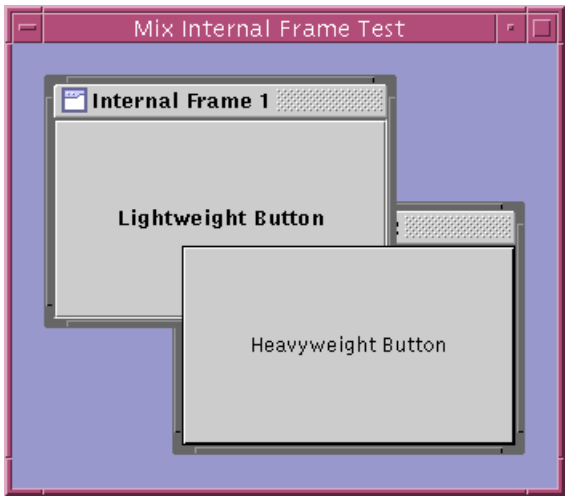
[화면 1] 여러 가지 Look and Feel




Java Look and Feel MacOS Look and Feel


Motif Look and Feel Windows Look and Feel

[화면 2] 중량 컴포넌트는 항상 맨 위쪽에 있는 것처럼 보인다.


Mix Internal Frame Test
Internal Frame 1
Lightweight Button
Heavyweight Button

# 3. 스윙 텍스트

스윙이 제공하는 텍스트 컴포넌트 관련 프레임웍은 그 규모가 무척 방대하다. 그런만큼 javax.swing.text(javax.swing.text.html, javax.swing.text.html.parser, javax.swing.text.rtf 포함)라는 별도의 패키지도 존재하는데 이 곳에는 JTextField, JPasswordField, JTextArea, JEditorPane, JTextPane 등의 텍스트 컴포넌트를 지원하기 위한 수많은 클래스와 인터페이스들이 포함되어 있다.([표 1]) 이들을 잘 활용하면 워드프로세서나 웹 브라우져같은 복잡한 텍스트 어프리케이션도 쉽게 구성할 수 있다.

[표 1] 주요 텍스트 컴포넌트들

| 클래스 (인터페이스 포함) | 설명 |
|---|---|
| JTextField | 오직 한 줄의 텍스트만을 다룰 수 있다. |
| JPasswordField | 비밀번호를 입력할 때 사용되는 텍스트 필드다. |
| JTextArea | 여러 줄의 텍스트를 다룰 수 있다. |
| JEditorPane | 다양한 종류의 내용물을 다룰 수 있는 텍스트 컴포넌트로서 내용물의 포맷에 따라 해당 포맷을 다룰 수 있는 에디터킷을 설정해 준다. |
| JTextPane | JEditorPane의 서브 클래스로서 화면상에서 비주얼하게 표시될 수 있는 기타의 내용물들을 추가할 수 있다. 예를 들어 텍스트 컴포넌트 내에 그림(Icon형 객체)이나 각종 컴포넌트들을 포함시킬 수 있다. |

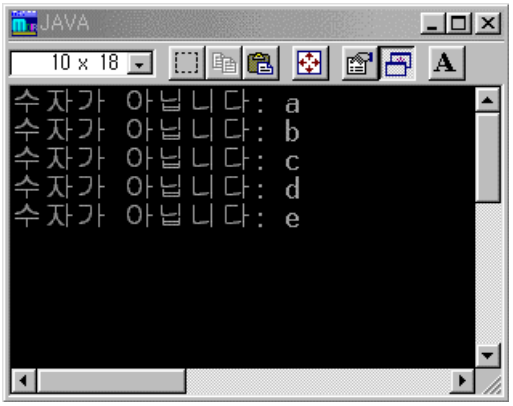
쉽다는 얘기는 어디까지나 그러한 기능을 직접 구현하는 것보다 쉽다는 이야기지 정말로 몇줄의 코드로 그런 어프리케이션이 툭 튀어 나온다는 얘기는 아니다. 필자 개인적인 의견으로는 텍스트 프레임웍이 일반 사용자들에게는 너무나 복잡하고 큰 규모의 것이 아닐까 한다. 독자가 현재 워드프로세서를 구상중이라면 가장 빠른 길이 될 수 있겠지만 단순히 텍스트 필드나 텍스트 에어리어를 좀 더 효과적으로 사용하기 위해 텍스트 프레임웍의 세세한 것까지 이해할 필요는 없을 것 같다. 따라서 필자도 독자가 텍스트 프레임웍을 제대로 이해하는 시작점으로 이 글을 삼을 수 있도록 하는 것을 글의 방향을 정하겠다. 그러니 혹시 필자가 간과하고 넘어가는 부분이나 설명의 편의상 약간씩 추상적인 내용이 나오는 곳에 대해서는 독자 나름의 연구가 있길 바란다.

## 백견이 불여 일타라.

먼저 당장 사용할 수 있는 예제 하나를 만들어 보는 것으로써 스윙의 텍스트 컴포넌트에 대한 탐험을 시작하자. 우리가 만들어 볼 프로그램은 텍스트 필드를 이용하는 프로그램이다. 그런데 이 텍스트 필드는 오로지 수자만 받아들인다. 즉 0에서부터 9까지의 수자 이외에는 아무것도 받아들이지 않는다. [화면 1]은 사용자가 키보드로 "12345abcde67890"을 입력한 결과를 보여준다. 보는 바와 같이 수자 이외의 문자는 모두 누락된다. 누락된 문자는 [화면 2]의 콘솔창에 찍히도록 했다.

[화면 1] 수자만 받아들이는 텍스트 필드


[화면 2] 텍스트 필드의 콘솔창




AWT를 깊게 다뤄본 사람이라면 이러한 프로그램을 이미 만들어 본 경험이 있을지도 모르겠다. AWT의 텍스트 필드를 사용한다면 아마도 사용자가 타이핑하는 키 입력을 가로채서 수자가 아닌 키 입력은 모두 무시하는 방식으로 처리할 수 있을 것이다. 그런데 그러한 방식에는 다음과 같은 두가지의 맹점이 있다.

1. 사용자가 복사/붙이기 기능을 사용하여 텍스트를 입력하였을 때 처리가 곤란하다.
2. 프로그래밍상의 문자열 입력에 대해 처리가 곤란하다.

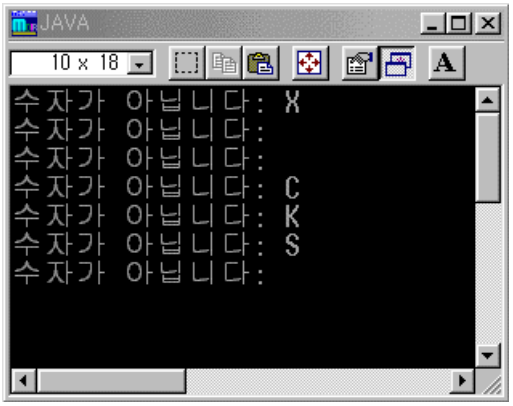
2번의 경우를 예로 든다면 프로그래머가 txtField.setText(str); 이런 코드를 추가하였을 때 매개변수로 입력되는 str에 대해 처리가 곤란하다는 것을 의미한다. str에 수자가 아닌 문자열이 포함되어 있다면 어떻게 해야할까? 아마 str을 따로 처리하는 별도의 검사코드를 만들어야할 것이다.

이제부터 우리가 만들어볼 텍스트 필드는 이러한 맹점들을 모두 극복하고 있다. 그것도 아주 깔끔한 방법으로 말이다. [화면 3]의 노트패드에 나타난 문자열을 클립보드에 “복사”한 다음 텍스트 필드에 “붙이기”한 결과가 [화면 4]다. 또 [화면 5]에는 누락되는 문자들이 나타나 있다. 복사/붙이기에 의한 입력도 잘 처리되고 있음을 알 수 있다.

[화면 3] 복사/붙이기용 문자열

[화면 4] 붙이기한 후의 택스트 필드


[화면 5] 붙이기한 후의 콘솔창

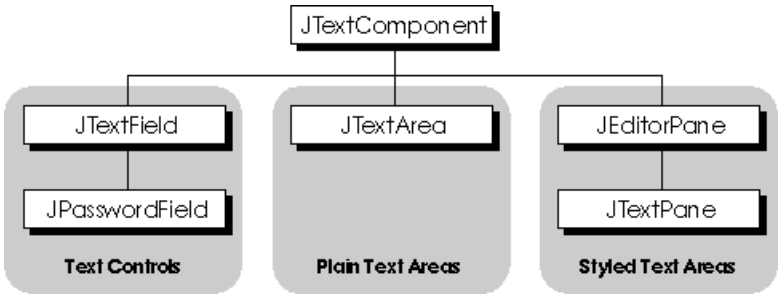



[리스트 1]에 텍스트 필드 프로그램의 소스코드가 나와있다.

## 구조를 알아야 응용력이 길러진다.

아직 모든 것을 이해할 수는 없겠지만 그동안 필자의 글을 보아왔던 독자라면 어느정도 감은 잡을 수 있을 것이다. 핵심은 바로 MVC(Model View Controller)다. 그럼 텍스트 컴포넌트들의 MVC 구조를 보기 전에 이들의 상속 계층도를 먼저 살펴보자. [그림 1]을 보면 모든 텍스트 컴포넌트들이 JTextComponent 클래스를 상속하고 있음을 알 수 있다. (그림에는 빠져 있지만 JTextComponent는 JComponent를 상속한다.) 바로 이 JTextComponent가 텍스트 컴포넌트들의 물밑 작업을 수행한다. 그러한 작업 중에 대표적인 것이 바로 MVC의 원활한 상호작용을 관리하는 기능이다.

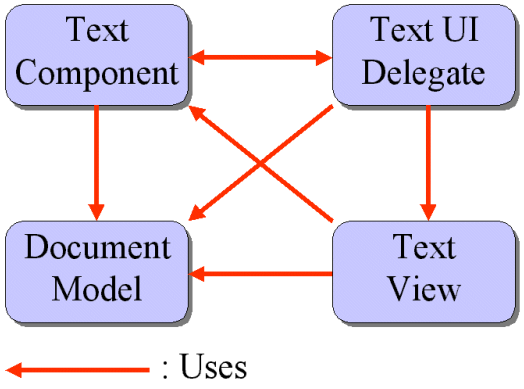
[그림 1] 텍스트 컴포넌트들의 상속 계층 구조




(그림에서 아래부분 문잘열은 각각 Text Controls, Plain Text Areas, Styled Text Areas에요. 잘 안 보이는 것 같아서요...)

[그림 2]에 텍스트 컴포넌트들의 내부 구조도(MVC)를 개략적인 관점으로 그려보았다. 스윙의 모든 텍스트 컴포넌트들은 그림과 같은 내부구조를 하고 있다. 잘 살펴보면 MVC 구조를 띄고 있음을 알 수 있다. 그런데 그림에서 한가지 이상한 점은 Delegate와 View가 따로 존재한다는 것이다. Delegate은 View와 Controller를 모두 포함하는 것이 보통인데(다른 스윙 컴포넌트들은 거의 모두 그렇다.) 스윙의 텍스트 컴포넌트들은 아주 복잡한 화면 구성 시스템을 사용하기 때문에 별도의 View 영역을 가지고 있다. 일단 이 글에서는 그런 구분을 하지는 않을 것이다. 독자가 알고 있던 바로 그 View라고 생각해도 무방하다. (MVC에 대해 독자가 잘 알고 있다고 가정하고 설명하겠으니 이해에 어려움이 있는 독자는 참고 서적이나 본지의 다른 기사를 참조하여 MVC 모델에 대해서만은 꼭 잘 알아두기 바란다.)

[그림 2] 텍스트 컴포넌트의 개괄적인 구성도 (Framework)




## 텍스트 필드의 실제 구현 방법

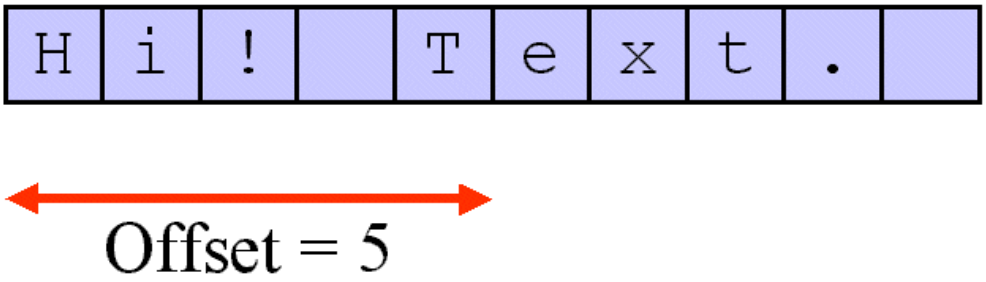
다시 우리의 텍스트 필드 이야기를 해보자. 우리의 텍스트 필드도 바로 MVC 구조를 활용한 것인데 그 중에 모델 부분을 수정하는 방법을 사용하였다. [리스트 1]을 보면 내부 클래스로 WholeNumberDocument라는 클래스를 정의하고 있는데 바로 이 클래스가 텍스트 필드의 모델 역할을 하게된다. (텍스트 컴포넌트의 모델은 이름에 Model이라는 접미어가 붙지 않고 Document라는 접미어가 붙는다.) 이 클래스는 다시 PlainDocument라고 하는 클래스를 상속하고 PlainDocument는 다시 AbstractDocument 클래스를 상속하며 AbstractDocument는 Document라는 인터페이스를 구현(implements)한다. 꽤나 복잡한 상속 구조를 가지고 있다. 그렇지만 여기서 중요한 것은 텍스트 컴포넌트의 모델이 되려면 Document 인터페이스를 구현해야 한다는 것이다. 어찌 되었든 결과적으로 우리의 WholeNumberDocument 클래스는 Document 인터페이스를 구현하고 있으므로 일단 텍스트 필드의 모델이 될 자격은 갖추고 있는 것이다.

PlainDocument**는 동일한 속성의 문자들로만 이루어진 텍스트 컴포넌트**(JTextField, JPasswordField, JTextArea) **모델의 기본적인 내용들을 구현하고 있다. 따라서 자신이 직접 모델을 수정하고 싶을 때는** PlainDocument **클래스를 상속하는 자신만의 모델 클래스를 정의한 다음 필요한 메쏘드만 오버라이드하는 방식을 택하면 된다.**

[리스트 1]에서 1번을 보자. createDefaultModel()메쏘드는 JTextField 클래스가 자신의 모델을 설정할 때 호출하는 메쏘드다. 따라서 우리가 새로 만든 모델(WholeNumberDocument)을 사용하려면 이 메쏘드에서 WholeNumberDocument 객체를 생성하여 리턴하면 된다. 만약 우리가 이 메쏘드를 오버라이드하지 않았다면 단순히 PlainDocument 객체 하나가 생성되어 리턴되게 되어있다.

모델 자체에서도 아주 중요한 메쏘드가 하나 있다. 2번의 insertString() 메쏘드는 텍스트 컴포넌트에 문자열을 삽입할 때 호출되는 메쏘드다. 매개변수 offset은 문자열이 삽입될 위치를 가르킨다. 결국 텍스트 컴포넌트란 문자열을 저장하는 저장소를 가지고 있어야하기 때문에 [그림 3]과 같이 내부적으로 문자열들을 저장할 수 있는 버퍼 비슷한 것을 가지고 있다. 이 내부 버퍼의 시작점에서부터의 위치가 바로 offset이 가르키는 점이다. offset은 0부터 시작한다.

[그림 3] Offset은 내부 버퍼의 시작 점에서부터의 위치를 나타낸다.




다음 매개변수 str은 삽입될 문자열을 가르키며 attrSet은 삽입될 문자열의 속성을 설정할 때 사용된다. 텍스트 필드는 단순히 동일한 모양의 문자들만 사용되기 때문에 attrSet이 의미가 없지만 독자가 워드프로세서나 웹 브라이져같은 것을 만들 계획이라면 바로 이 AttributeSet에 관심을 갖어야 할 것이다. AttributeSet은 글꼴이나 글자의 전경색, 배경색 등등 수많은 속성들을 담을 수 있다. 구체적인 얘기는 생략하겠다. insertString() 메쏘드에서 마지막으로 살펴볼 것은 BadLocationException이라는 예외(Exception)다. 이름에서 짐작할 수 있듯이 잘못 된 위치가 지정되었을 때 예외가 던져(throw)진다. 예를 들어 offset에 음수 값을 지정하였다든지 아니면 전체 버퍼의 범위를 넘어가는 값을 지정하였다든지의 경우를 들 수 있다. BadLocationException은 모든 텍스트 컴포넌트에서 수시로 사용되므로 이 예외의 의미를 잊지 않도록 하자. 예외 처리의 구체적인 이야기도 생략하겠다.

[리스트 1] 수자만 입력받는 텍스트 필드 예제

```
import javax.swing.*;
import javax.swing.text.*;

/** 수자만 입력받는 텍스트 필드 */
public class WholeNumberField extends JTextField {

    public WholeNumberField() {
```

```
        super();
    }

    public WholeNumberField(int columns) {
        super(columns);
    }

    /** 텍스트 필드의 모델을 생성한다 */
    protected Document createDefaultModel() {              <-----
1번
        return new WholeNumberDocument();
    }

    /** 텍스트 필드의 모델 클래스 */
    protected class WholeNumberDocument extends PlainDocument {

        public void insertString(int offset, String str,    <-----
2번
                AttributeSet attrSet) throws BadLocationException {

            char[] source = str.toCharArray();
            char[] result = new char[source.length];
            int j = 0;

            // 입력 문자열의 각 문자마다 수자인지 아닌지 검사하여
            // 수자가 아닌 문자는 누락시킨다.
            for (int i = 0; i < result.length; i++) {
                if (Character.isDigit(source[i]))
                    result[j++] = source[i];
                else {
                    System.err.println("수자가 아닙니다: " +
source[i]);
                }
            }

            // 수자로만 이루어진 문자열을 텍스트 필드에 추가한다.
            super.insertString(offset, new String(result, 0, j),
                  attrSet);
        }
    }

    public static void main(String[] args) {
        JFrame frame = new JFrame("WholeNumberField Demo");
        WholeNumberField field = new WholeNumberField(20);

        frame.getContentPane().add(field);
```

```
        frame.pack();
        frame.setVisible(true);
    }
}
```

그럼 지금까지의 내용을 정리해 보자.

**텍스트 필드는 모델의** insertString() **메쏘드를 통해 외부로부터 문자열을 받아들인다**. **따라서 이곳에 필터링 장치를 설치해 놓으면 텍스트 필드가 원하는 방식으로 작동하도록 할 수 있다**.

텍스트 필드 예제를 보면서 텍스트 컴포넌트들이 어떠한 방식으로 작동하는지 어느정도 감을 잡았을 것이다. 이번엔 JTextField나 JTextArea의 범위를 넘어서는 예제를 살펴보면서 텍스트 컴포넌트의 또다른 특성을 알아보자.

# EditorKit은 어디에 쓰이는 물건인가?

텍스트 컴포넌트들의 컨트롤러(Controller) 역할은 에디터킷(Editor Kit)이라고하는 객체가 맡고 있다. 에디터킷은 텍스트를 읽거나 쓰기 또는 편집하는 기능등을 수행한다.

편집 기능은 Action 객체를 기반으로 수행된다. (Action에 대해서는 필자가 본 기사의 5월호에 소개한 적이 있으니 참조하기 바란다.) 편집이라하면 문자열을 삽입한다거나 자르기/복사/붙이기 등의 작업을 들 수 있는데 Action 객체를 이용하는 것이 바람직할 것이다. 5월호 기사에도 소개하였지만 사용자가 메뉴에서 복사 메뉴를 선택할 수도 있고 단축키나 툴바를 통해서도 그러한 명령을 수행할 수도 있기 때문에 Action 객체를 선택한 것은 적당한 선택이었다.

그렇지만 에디터킷의 가장 중요한 역할은 텍스트의 편집이 아니라 텍스트를 읽고 쓰는 작업이다. 만약 여러분이 웹 부라우져를 만든다거나 텍스트 에디터를 만들 계획이라면 이 둘은 매우 상이한 작업이 될 수 있다. 우선 다루는 문서의 포맷부터가 완전히 다르다. 하나는 HTML 포맷의 문서를 다룰 것이고 하나는 아스키 텍스트 문서를 다루게될 것이다.

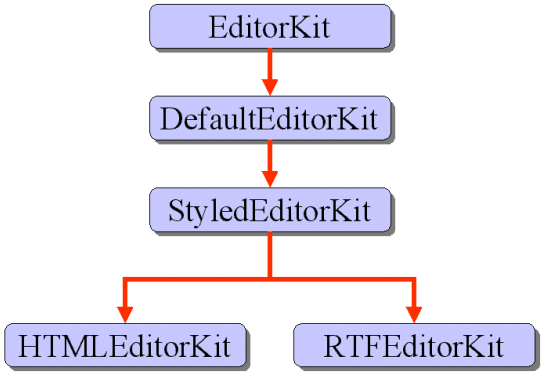
에디터킷을 사용하면 이러한 작업들을 일관된 방식으로 처리할 수 있다. 게다가 스윙의 텍스트 패키지에는 몇몇 유용한 에디터킷들이 준비되어 있다. [그림 4]에서 스윙의 텍스트 패키지가 제공하는 에디터킷들의 상속 계층도를 볼 수 있다.

DefaultEditorKit은 평이한 텍스트 문서를 다룰 때 사용되며 StyledEditorKit은 다양한 글꼴이나 그림같은 것들을 포함하는 좀더 복잡한 형식의 문서를 다룰 때 사용된다. 또한 StyledEditorKit은 JTextPane의 기본 에디터킷이기도 하다. HTMLEditorKit과 RTFEditorKit은 둘다 StyledEditorKit의 서브클래스인데 각각 HTML 형식의 문서와 RTF 형식의 문서를 다룰 때 사용된다.

각각의 에디터킷들은 [그림 1]에서 보았던 JEditorPane 클래스에 등록되어 있는데 자신들이 다룰 수 있는 문서의 포맷과 함께 등록되어 있다. JEditorPane은 어떤 파일이 로드될 때 그 파일 포맷을 다룰 수 있는 에디터킷이 있는지 검사하여 그런 에디터킷이 존재하면 로드된 파일에 대해

해당 에디터킷을 사용하여 문서를 표현하거나 편집하도록 한다. 만약 사용자가 자신만의 특별한 문서 형식을 창조하였다면 그 문서형식을 다룰 수 있는 에디터킷을 만들어서 registerEditorKitForContentType() 메쏘드를 통해 JEditorPane에 등록할 수 있다. 그러면 이후부터는 새로운 형식의 문서를 로드할 때마다 사용자가 만든 에디터킷을 사용하게된다.

[그림 4] EditorKit의 상속 계층도




그럼 지금까지 배운 지식을 이용하여 간단한 HTML Viewer를 만들어보자. 이 예제는 단순성을 기하기 위하여 예외처리나 Hyper Link의 탐색 기능을 포함시키지 않았다. 그럼에도 불구하고 너무나도 단순한 코드가([리스트 2]) 너무 많은 일을 하는 것 같다. 단순히 JEditorPane의 생성자에 URL을 넘겨주는 것 만으로 페이지가 로드되었다. ([화면 6]) 필자가 앞에서 설명했던 에디터킷이 그 이면 작업을 모두 처리하고 있기 때문이다.

JEditorPane에는 중요한 메쏘드가 하나 있다. setPage(String url) 메쏘드가 그것이다. setPage() 메쏘드는 URL을 매개변수로 받는데 바로 특정한 URL을 화면에 표시할 때 사용된다. 사실 JEditorPane의 생성자가 하는 일도 단순히 setPage() 메쏘드를 호출하는 것이 전부다. setPage() 메쏘드는 URL의 포맷을 검사하여 해당 포맷을 다룰 수 있는 에디터킷을 찾아 그 에디터킷으로 페이지를 로드하게 한다. 물론 우리의 HTML Viewer 예제에서는 HTMLEditorKit이 사용되었을 것이다.

[리스트 2] 간단한 HTML Viewer

```
import java.io.*;
import javax.swing.*;

public class HTMLExample {

    public static void main(String[] args) {
        JEditorPane pane = null;
        String url = "http://java.sun.com/index.html";

        try {
            pane = new JEditorPane(url);
```

```
            } catch (IOException ex) {}
            pane.setEditable(false);

            JFrame frame = new JFrame("HTML Page Viewer V1.0 ^^");
            frame.setContentPane(new JScrollPane(pane));
            frame.setSize(600, 400);
            frame.setVisible(true);
        }
}
```

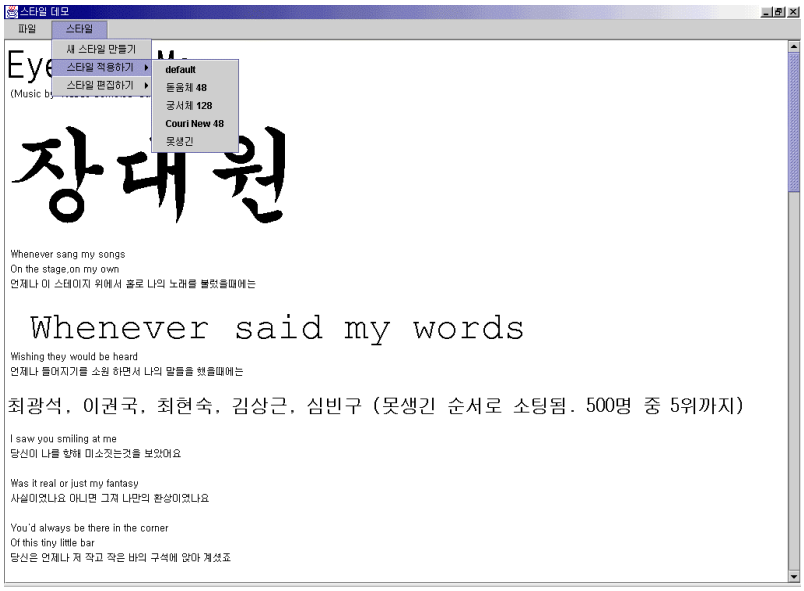
[화면 6] HTML Viewer의 실행화면


# 워드프로세서를 만들자.

필자가 이 글을 쓸 때 체계적으로 차근차근 설명할 생각은 아예 처음부터 저버릴 수 밖에 없었다. 그런 식으로 설명하려면 족히 책 한권이 필요하기 때문이다. 서두에도 이야기 했듯이 스윙의 텍스트 API는 너무 크고 복잡하다. 그래서 필자가 생각한 방법은 우선 잘 돌아가는 프로그램을 제시하고 그 프로그램에 대한 뒷조사(?)를 하는 방식이다. 여기서 여러분은 꽤 괜찮은(?) 프로그램을 하나 보게될 것이다. 먼저 프로그램이 어떤 식으로 돌아가는지 살펴보고 그 구현 방법에 대해 생각해 보자.

[화면 7]을 보면 마치 조그마한 워드프로세서를 보는 듯 할 것이다. 물론 당장 워드프로세서로 사용될만한 프로그램은 아니지만 여러분이 워드프로세서를 만들 계획이라면 그 시작점으로 삼을만한 프로그램임에는 틀림없다.

[화면 7] 스타일을 사용하는 미니 워드프로세서

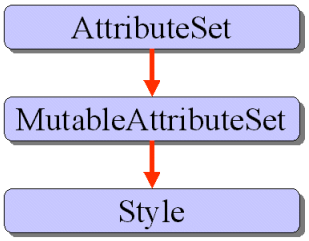



# 스타일에 관하여 살펴보자

이 프로그램의 핵심은 스타일(Style)이라는 개념이다. 여러분이 글을 잘 다룰 줄 아는 사람이라면 스타일이라는 개념에 익숙할 것이다. 스타일은 주어진 단위(엘리먼트)의 글이 화면상이나 프린터등에 어떠한 방식으로 표시될 것인가를 결정한다. 다시 말한다면 글꼴이나 문단 정렬 방법등 글을 표현하는데 필요한 각종 특성값(Attribute)들의 집합이라 할 수 있겠다. 스윙에서는 Style이라는 인터페이스가 스타일을 다룰 수 있게 해준다. Style 인터페이스는 MutableAttributeSet 인터페이스를 상속하며 MutableAttributeSet은 다시 AttributeSet 인터페이스를 상속한다. ([그림 5]) 여기서는 일단 이들이 모두 단지 속성값들의 집합이라고 생각하길 바란다.

속성값들을 설정하거나 얻어오는 방법은 여러 가지가 있을 수 있겠으나 우리는 StyleConstants의 static 메쏘드들을 사용할 것이다. StyleConstants는 좀 더 안전한 방법으로 속성값들을 얻거나 설정하게 해준다. 예를 들어 StyleConstants.setFontSize(style, int size) 이렇게 할 수 있다. 이렇게 하면 style이라는 스타일 객체의 글꼴 크기를 size로 설정하게 된다. 속성값을 얻어오는 것도 간단하다 StyleConstants.getFontSize(style) 이렇게 하면 style이라는 스타일 객체에 현재 설정된 글꼴의 크기를 얻어올 수 있다.

[그림 5] 스타일의 상속 계층도

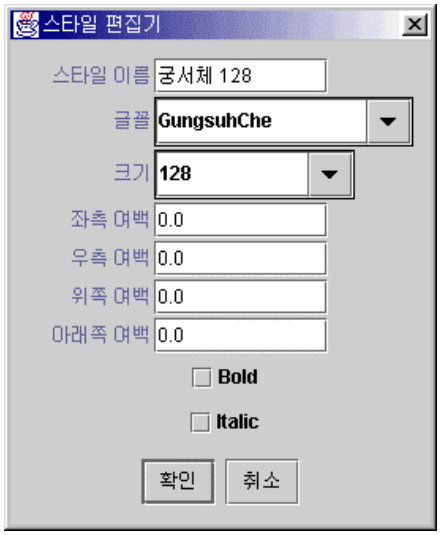


[화면 8]에 스타일의 각종 특성값(Attribute)들을 편집할 수 있는 대화상자가 나와있다. 이 대화상자가 어떤 방식으로 작동하는지 [리스트 3]을 보면 알수 있는데 fillStyle() 메쏘드와 loadFromStyle() 메쏘드에서 어떤 식으로 스타일의 특성값들을 얻거나 설정하는지 알아두기 바란다. 이 대화상자는 워드프로세서 내에서 스타일 관련 작업(스타일의 생성, 적용, 편집)을 할 때 사용된다.

[리스트 3] 스타일 편집기의 코드

```
/** 스타일을 편집하거나 생성하기위한 대화상자 */
public class StyleEditor extends Container {
    ...
    /** 글꼴의 크기를 설정하는 콤보박스 */
    private JComboBox sizeCombo;
    ...

    /** 윈도우 컴포넌트들로부터 스타일의 속성(Attribute)들을 채운다. */
    public void fillStyle(Style style) {
        ...
        String size = (String)sizeCombo.getSelectedItem();
        StyleConstants.setFontSize(style, Integer.parseInt(size));
        ...
    }

    /** 스타일의 속성(Attribute)들을 윈도우 컴포넌트들에 반영한다. */
    public void loadFromStyle(Style style) {
        ...
        int size = StyleConstants.getFontSize(style);
        sizeCombo.setSelectedItem(Integer.toString(size));
        ...
    }
    ...
}
```

[화면 8] 스타일 편집기 대화상자




# 다양한 스타일을 지원하는 모델을 만들자

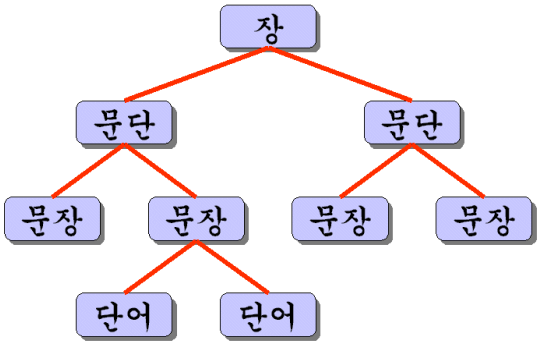
다음으로 우리가 살펴봐야 할 것은 모델 부분이다. [리스트 4]에 새로 정의된 모델의 코드가 나와있다. 앞에서 텍스트 필드에 대한 이야기를 할 때 모델을 언급했으므로 여기서는 새로 알아야 할 내용만 언급하겠다. 텍스트 필드와는 달리 우리가 만들려는 워드프로세서는 다양한 스타일의 문서형태를 지원해야 한다. 다양한 스타일을 지원하는 모델을 만드는 가장 손쉬운 방법은 DefaultStyledDocument를 상속하는 모델을 정의하는 것이다. 그럼 DefaultStyledDocument에 대해 잠시 알아보자. 이 클래스의 정의부분을 보면 다음과 같다.

```
public class DefaultStyledDocument extends AbstractDocument
implements StyledDocument
```

우선 AbstractDocument라는 클래스를 상속하고 있다. 그런데 AbstractDocument는 우리가 텍스트 필드에 대한 이야기를 할 때 이미 언급이 되었던 클래스다. 이 클래스는 단순한 텍스트 편집에 대한 기능을 제공하는 클래스였다. 따라서 다양한 스타일을 지원하는 모델 클래스를 만들려면 StyledDocument라는 인터페이스의 도움이 필요하다. StyledDocument는 문서에 다양한 스타일을 적용할 수 있도록 해준다.

또 하나 꼭 알고 넘어가야 할 것이 있다. 바로 엘리먼트(Element)의 개념인데 엘리먼트라는 것은 일정량의 문서 조각(piece)을 말한다. 문서의 조각이란 크게는 문서 전체를 가르킬 수도 있고 작게는 단 한 글자의 문자를 가르킬 수도 있다. 물론 문단, 문장, 단어등도 그런 조각이 될 수 있다. 그런데 스윙의 텍스트컴포넌트들은 엘리먼트들을 계층구조로 다룬다. ([그림 6]) 계층구조라는 것은 어떤 엘리먼트들을 모두 포함하는 또 다른 엘리먼트가 있고 또 그런 엘리먼트들이 더 큰 엘리먼트에 포함된다는 것이다. 예를 들어 여러 문장이 모여서 문단이 만들어지는 것과 같다.

[그림 6] 엘리먼트들의 계층 구조




엘리먼트가 중요한 이유는 모델이 텍스트를 다룰 때 엘리먼트 단위로 다루기 때문이다. 예를 들어 어느 특정 단어만 글꼴크기를 24로 하고 싶다면 최소한 그 단어는 엘리먼트 단위여야 한다. 그 단어를 포함하고 있는 문장까지만 하나의 엘리먼트여서는 안된다. 그럴 경우 그 문장 전체의 특성을 한꺼번에 변경할 수는 있어도 그 속에 포함된 어떤 특정 부분의 특성을 변경시킬 수는 없다. 그렇지만 그 단어가 여러개의 엘리먼트들로 이루어진 것이라면 상관 없다. 예를 들어 그 단어의 각 글자가 모두 어떤 엘리먼트 단위로 이루어져 있다면 각 글자마다 똑같은 특성을 부여하면 되기 때문이다. 이렇듯 텍스트 컴포넌트에서 엘리먼트는 어떤 조작이 행해지는 단위가 된다.

그럼 방금 배운 지식들을 이용하여 워드프로세서의 모델 부분을 분석해 보자. [리스트 4]의 1번을 보면 해시 테이블 객체를 하나 만들고 있다. 이 해시테이블은 어떤 엘리먼트에 어떤 스타일이 적용되고 있는지를 알아내기 위해 사용된다. 예를 들어 [화면 7]을 보면 “장대원”이라고 쓰여진 세 번째 문장(이것도 하나의 엘리먼트가 될 수 있다)은 “궁서체 128”이라는 스타일이 적용되어 있는 상태다. 그런데 나중에 “궁서체 128”이라는 스타일에서 글꼴을 24로 바꿨다고 생각해 보자. 그러면 “궁서체 128”이 적용된 모든 엘리먼트의가 변경된 내용을 반영하도록 해줘야 한다. 이럴 때 해시테이블이 사용된다. 해시테이블을 검색해서 해당 스타일이 적용된 모든 엘리먼트를 찾아서 각 엘리먼트가 변경 사항을 반영하도록 하게 하는 것이다. 바로 2번이라고 되어있는 곳의 styleUpdated()와 `updateStyleHash()` 메쏘드에서 해시테이블을 관리하거나 변경 사항에 대한 처리 행해지고 있다.

[리스트 4] 워드프로세서의 모델 코드

```
/**
 * 워드프로세서의 모델(Document) 역할을 한다. 스타일이 바뀌었을 때
 * 그것을 반영할 수 있도록 어떤 엘리먼트(Element) 어떤 스타일을
 * 사용하는지를 추적한다.
 */
public class StylishDocument extends DefaultStyledDocument
    implements DocumentListener {

    /** 엘리먼트/스타일 관계를 추적하는 데 사용된다. */
    private Hashtable styleHash = new Hashtable();          <--- 1번
```

```
    ...

    /** 생성자 */
    public StylishDocument() {
        super();
        init();
    }

    /**
     * 도큐먼트 이벤트를 스스로 처리한다. 또한 처음부터 존재하는
     * Paragraph에 대해서는 그것이 추가되었다는 이벤트를 통보받지
     * 못하기때문에 수동으로 직접 해시테이블에 넣어준다.
     */
    protected void init() {
        addDocumentListener(this);
        addToStyleHash(getParagraphElement(0));
    }

    /**
     * 주어진 스타일이 변경되었을 때 그 스타일을 사용하는 모든 엘리먼트
     * 들이 변경 내용을 반영할 수 있도록한다.
     */
    public void styleUpdated(Style style) {                <--- 2번

        // 주어진 스타일을 사용하는 엘리먼트들을 찾는다.
        Hashtable ht = (Hashtable)styleHash.get(style);

        if (ht != null) {
            Vector cleanUp = new Vector();
            Enumeration e = ht.keys();
            while (e.hasMoreElements()) {
                Element el = (Element)e.nextElement();
                int start = el.getStartOffset();
                int end = el.getEndOffset();
                Style check = getLogicalStyle(start);

                if (check == style) {
                    DefaultDocumentEvent        ev        =        new
DefaultDocumentEvent
                                (start, end-start,
                                DocumentEvent.EventType.CHANGE);
                    fireChangedUpdate(ev);
                }
                else {
                    cleanUp.addElement(el);
```

```
                }
            }

            e = cleanUp.elements();
            while (e.hasMoreElements()) {
                Element bad = (Element)e.nextElement();
                ht.remove(bad);
            }
        }
    }

    // DocumentListener 메쏘드들
    public        void        insertUpdate(DocumentEvent        ev)
{ updateStyleHash(ev); }
    public        void        removeUpdate(DocumentEvent        ev)
{ updateStyleHash(ev); }
    public void changedUpdate(DocumentEvent ev) {
        int offset = ev.getOffset();
        Element para = getParagraphElement(offset);
        addToStyleHash(para);
    }

    /** 추가되거나 제거된 엘리먼트가 있을 경우 호출된다. */
    protected void updateStyleHash(DocumentEvent ev) {          <-
-- 2번
        DocumentEvent.ElementChange chg =
                ev.getChange(getDefaultRootElement());

        if (chg != null) {
            Element[] removed = chg.getChildrenRemoved();
            for (int i=0; i<removed.length; i++) {
                removeFromStyleHash(removed[i]);
            }

            Element[] added = chg.getChildrenAdded();
            for (int i=0;i<added.length;i++) {
                addToStyleHash(added[i]);
            }
        }
    }

    /** 해시테이블에 엘리먼트를 추가한다. */
    protected void addToStyleHash(Element para) {
        AttributeSet attrs = para.getAttributes();
        if (attrs != null) {
            Style style = (Style)attrs.getResolveParent();
            if (style != null) {
```

```
                Hashtable ht = (Hashtable)styleHash.get(style);
                if (ht == null) {
                    ht = new Hashtable();
                    styleHash.put(style, ht);
                }
                if (ht.containsKey(para) == false) {
                    ht.put(para, new Object());
                }
            }
        }
    }

    /** 해시 테이블에서 엘리먼트를 제거한다. */
    protected void removeFromStyleHash(Element para) {
        AttributeSet attrs = para.getAttributes();
        if (attrs != null) {
            Style style = (Style)attrs.getResolveParent();
            if (style != null) {
                Hashtable ht = (Hashtable)styleHash.get(style);
                if (ht != null) {
                    ht.remove(para);
                }
            }
        }
    }
}
```

[리스트 5] 워드 프로세서 메인 프로그램

```
/** JTextPane을 담는 프레임. */
public class StyleDemo extends JFrame {
    private StyleEditor styleEditor = new StyleEditor();
    private JTextPane textPane;
    private StylishDocument doc;
    private JMenu styleMenu;
    private JMenu setStyleMenu;
    private JMenu modifyStyleMenu;
    private Hashtable styleHash = new Hashtable();

    public static void main(String[] args) {
        JFrame frame = new StyleDemo();
        frame.setVisible(true);
    }

    /** 모델을 생성하여 JTextPane에 설정한다. */
    public StyleDemo() {
        super("스타일 데모");
        doc = new StylishDocument();
```

```
        textPane = new JTextPane(doc);                          <--- 1번
        setContentPane(new JScrollPane(textPane));

        JMenuBar menuBar = createMenuBar();
        setJMenuBar(menuBar);

        setSize(800, 600);
    }

    /** 메뉴를 만든다. */
    protected JMenuBar createMenuBar() {
        JMenuBar bar = new JMenuBar();

        JMenu fileMenu = new JMenu("파일");
        JMenuItem exitItem = new JMenuItem("종료");
        exitItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ev) {
                System.exit(0);
            }
        });
        fileMenu.add(exitItem);
        bar.add(fileMenu);

        styleMenu = new JMenu("스타일");
        JMenuItem createItem = new JMenuItem("새 스타일 만들기");
        createItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ev) {
                showStyleDialog();
            }
        });

        setStyleMenu = new JMenu("스타일 적용하기");
        modifyStyleMenu = new JMenu("스타일 편집하기");
        styleMenu.add(createItem);
        styleMenu.add(setStyleMenu);
        styleMenu.add(modifyStyleMenu);
        bar.add(styleMenu);

        addStyleToMenus(doc.getStyle(StyleContext.DEFAULT_STYLE));
        return bar;
    }

    /** 스타일 대화상자를 보인다. */
    protected void showStyleDialog() {
```

```
            String[] options = {"확인", "취소"};

            int opt = JOptionPane.showOptionDialog(this, styleEditor,
                    "스타일 편집기", JOptionPane.DEFAULT_OPTION,
                    JOptionPane.PLAIN_MESSAGE,       null,       options,
    options[0]);

            if (opt == 0) { // "OK" pressed
                String name = styleEditor.getStyleName();
                if (name != null) {
                    Style oldStyle = doc.getStyle(name);
                    if (oldStyle != null) {
                        styleEditor.fillStyle(oldStyle);
                        doc.styleUpdated(oldStyle);
                    }
                    else {
                        Style newStyle = doc.addStyle(name, null);
                        styleEditor.fillStyle(newStyle);
                        addStyleToMenus(newStyle);
                    }
                }
            }
            styleEditor.clear();
        }

        /** 메뉴에 새로운 스타일을 추가한다. */
        protected void addStyleToMenus(final Style newStyle) {
            String styleName = newStyle.getName();
            JMenuItem newItem = new JMenuItem(styleName);
            setStyleMenu.add(newItem);

            newItem.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent ev) {
                    textPane.setLogicalStyle(newStyle);         <--- 2번
                }
            });

            newItem = new JMenuItem(styleName);
            modifyStyleMenu.add(newItem);

            newItem.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent ev) {
                    styleEditor.loadFromStyle(newStyle);
                    showStyleDialog();
                }
            });
        }
```

```
}
```

워드프로세서 프로그램의 메인 부분을 이루고 있는 소스크드([리스트 5])에 대해서는 특별히 중요한 내용은 없다. 이 부분은 우리가 앞서 정의한 모델을 JTextPane의 모델로 설정하는 것(1번)과 스타일 편집기 대화상자를 띄워서 그 결과를 JTextPane에 반영하는 일 등을 하고 있다. 결과를 반영하는 방법은 2번에 나타나 있다. JTextPane의 setLogicalStyle() 메쏘드는 현재 캐럿(Caret)이 위치해 있는 곳의 엘리먼트의 스타일을 주어진 스타일로 설정한다.

스윙의 텍스트 컴포넌트에 대해 자료를 수집하고 수집된 자료들을 연구하면서 필자가 느낀 것이 있다면 한마디로 “많다”였다. 당장이라도 JDK API 도큐먼트에서 javax.swing.text 패키지를 살펴본다면 어마어마하게 많은 클래스와 인터페이스들에 기가 죽을 정도일 것이다. 게다가 이 클래스들은 왜 또 그리 내부 클래스(Inner Class)들을 많이 정의하고 있는지 모르겠다. 이것들까지 하나하나 살펴보려니 정말 머리가 뻐근하다고밖에 할 말이 없었다. 필자가 이렇게 불평을 늘어놓는 이유는 독자들에게 변명을 좀 해볼까 해서다. 짧은 기사 속에 어떻게 하면 독자들이 많은 것을 얻을 수 있을까 생각하다가 좀 엉성하게 되더라도 실제로 쓸 수 있는 내용들을 넣자라고 결론을 내렸다. 그래서 체계적인 설명이 없었음을 시인한다. 이 글을 보고 원래는 없었던 의문점까지 생겼을 독자도 있을 것이다. 그러나 필자는 그런 현상이 나쁜 것만은 아니라고 생각한다. 정말로 아무것도 모르는 사람은 그런 의문도 없을 것이기 때문이다. 부디 독자들은 필자를 원망하지 말고 아래 필자가 제시하는 자료들을 토대로 스스로 의문을 풀기 바란다.

스윙 커넥션의 텍스트에 관한 문서들:
http://java.sun.com/products/jfc/tsc/text/text_main.html

자바 튜토리얼의 텍스트에 관한 문서들:
http://java.sun.com/docs/books/tutorial/uiswing/components/text.html