6장 고급 스윙

트리
테이블
스타일 텍스트 컴포넌트(Styled Text Components)
슬라이드와 진행상태 측정기
툴바와 툴팁
컴포넌트 구성

이 장에서는 1권에서 설명한 스윙 사용자 인터페이스 툴킷에 대해 계속해서 설명하겠다. 스윙은 매우 크고 복잡한 툴킷이고 1권에서는 가장 중요한 컴포넌트만을 설명하였다. 중요한 컴포넌트라는 정의는 의미가 없기 때문에 AWT와 비슷한 컴포넌트를 1권에서 설명하였다고 할 수 있다. 아직 두개의 아주 복잡한 컴포넌트인 트리와 테이블은 설명하지 않았다. 이장에서 그것들에 대해서 설명할 것이다. 그리고 그 이외에 진행상태 모니터, 슬라이드 그리고 툴바를 설명함으로써 이 장을 마치겠다.

트리

계층적인 파일 시스템을 사용하는 모든 컴퓨터에서 그림 6-1과 같은 트리 구조를 볼 수 있다. 물론 디렉토리와 파일은 트리 구조의 많은 예들 중의 하나이다. 프로그래머는 클래스의 상속 트리에 익숙해 있다. 우리는 국가, 시, 구 형태와 같이 주변 생활에서 많은 트리 구조를 발견할 수 있다.

그림 6-1: 디렉토리 트리

그림 6-2 : 국가, 주 , 도시의 계층도

프로그래머들은 종종 이러한 트리 구조를 사용해야만 한다. 다행스럽게도 스윙 라이브러리는 이러한 목적을 위하여 JTree 클래스를 포함한다. JTree 클래스(JTree의 helper 클래스와 함께)는 트리를 펼치고 접는 프로세스를 지원한다. 이 섹션에서 여러분들은 JTree 클래스를 사용하는 방법을 배울 것이다. 다른 복잡한 컴포넌트와 마찬가지로 모든 내용을 포함할 수 없으므로 공통적이고 유용한 경우에 초점을 맞출 것이다. 만약 여러분이 더 많은 기능을 사용하기를 원한다면 Kim Topley의 Core Java Foundation classes[Prentice-Hall 1998]나 David M.Geary의 Graphic Java 2를 참고하기 바란다.

진행하기 전에 몇 가지 용어를 정의하자.(그림 6-3을 보라) 트리는 노드로 구성되어 있다. 모든 로드는 잎 노드이거나 자식 노드를 가진다. 루트 노드를 제외하고 모든 로드는 단 하나의 부모 노드를 가진다. 트리는 단 하나의 루트 노드를 가진다. 가끔 트리의 집합을 볼 수 있는데 그 집합을 숲이라 부른다.

그림 6-3 : 트리 용어

간단한 트리

우리의 첫번째 프로그램에서는 몇 개의 노드를 가진 트리를 볼 것이다.(페이지 323의 그림 6-5) 대부분의 다른 스윙 컴포넌트와 마찬가지로 JTree 컴포넌트는 모델/뷰/컨트롤 패턴을 따른다. 여러분은 계층적인 데이터 모델을 사용할 것이고 컴포넌트는 그것을 여러분에게 보여줄 것이다. JTree를 생성하기 위해 생성자에 트리 모델을 사용해 보자.

```
TreeModel model = . . . ;
JTree tree = new JTree(model);
```

노트: 요소의 집합으로부터 트리를 생성하는 생성자가 있다.

```
JTree(Object[] nodes)
JTree(Vector nodes)
JTree(Hashtable nodes) // 값이 노드가 된다.
```

이 생성자들은 매우 유용하지는 않다. 다만 각각의 싱글 노드를 가지고 트리의 숲을 만든다. 세 번째 생성자는 키의 해쉬 코드에 의해 주어진 임의의 한 순서로 노드를 나타내기 때문에 특별한 경우가 아니면 쓸모가 없다.

어떻게 트리 모델을 얻을 수 있을까? 여러분은 TreeModel 인터페이스를 적용하는 클래스를 생성함으로써 여러분 자신의 모델을 생성할 수 있다. 그 방법은 이 장 후반부에서 설명할 것이고 지금은 스윙 라이브러리가 제공하는 DefaultTreeModel을 사용하겠다.

Default tree model을 생성하기 위해 루트 노드를 사용해야 한다.

```
TreeNode root = . . .;
DefaultTreeModel model = new DefaultTreeModel (root);
```

TreeNode는 다른 인터페이스이다. 따라서 그러한 인터페이스가 제공되는 클래스 객체를 가진 default tree model을 사용할 수 있다. 지금은 스윙이 제공하는 구조화된 노드 클래스인 DefaultMutableTreeNode를 사용할 것이다. 이 클래스는 TreeNode의 서브인터페이스인 MutatleTreeNode 인터페이스를 제공한다.(그림 6-4)

그림 6-4 : 트리 클래스들

default mutable tree 노드는 사용자 객체를 보유한다. 트리는 모든 노드에 사용자 객체를 나타낸다. 만약 여러분이 랜더러를 정하지 않는다면 트리는 단지 toString 메소드의 결과값인 문자열을 나타낸다.
우리의 첫번째 예제에서 문자열을 사용자 객체로서 사용하겠다. 실제로 여러분은 대개 보다 많은 기능을 가진 사용자 객체를 지원하는 트리를 사용할 것이다. 예를 들면, 디렉토리 트리를 나타낼 때 노드에 File 객체를 사용하는 것은 유용하다.
여러분은 생성자로서 사용자 객체를 정의하거나 setUserObject 메소드로 나중에 사용자 객체를 설정할 수 있다.

```
DefaultMutableTreeNode node
        = new DefaultMutableTreeNode("Texas");
node.setUserObject("California");
```

다음으로 노드들 사이에 부모/자식 관계를 형성할 수 있다. 먼저 루트 노드를 만들고 add 메소드를 사용하여 자식노드를 추가하라.

```
DefaultMutableTreenode root
        = new DefaultMutableTreeNode("World");
DefaultMutableTreeNode country
        = new DefaultMutableTreeNode("USA");
root.add(country);
DefaultMutableTreeNode state
        = new DefaultMutableTreeNode("Callfornia");
country.add(state);
```

이 트리가 보여지는 모양은 그림 6-5와 같다.

그림 6-5 : 단순 트리

이러한 방식으로 모든 노드를 연결한다. 다음으로 루트 노드를 가진 DefaultTreeModel을 생성한다. 그리고, 트리 모델로 JTree를 생성한다.

```
DefaultTreeModel treeModel = new DefaultTreeModel(root);
JTree tree = new JTree(treeModel);
```

또는 간단히 루트 노드를 JTree생성자로 넘겨줄 수 있다.
그리고 나서 트리는 자동적으로 default tree model을 생성한다.
전체 코드는 예제 6-1에 나타내었다.

예제 6-1 : SimpleTree.java

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

public class SimpleTree
{  public static void main(String[] args)
   {  JFrame frame = new SimpleTreeFrame();
      frame.show();
   }
}

class SimpleTreeFrame extends JFrame
{  public SimpleTreeFrame()
   {  setTitle("SimpleTree");
      setSize(300, 200);
      addWindowListener(new WindowAdapter()
         {  public void windowClosing(WindowEvent e)
            {  System.exit(0);
            }
         } );

      // set up tree model data

      DefaultMutableTreeNode root
         = new DefaultMutableTreeNode("World");
      DefaultMutableTreeNode country
         = new DefaultMutableTreeNode("USA");
      root.add(country);
      DefaultMutableTreeNode state
         = new DefaultMutableTreeNode("California");
      country.add(state);
      DefaultMutableTreeNode city
         = new DefaultMutableTreeNode("San Jose");
      state.add(city);
```

```
        city = new DefaultMutableTreeNode("Cupertino");
         state.add(city);
        state = new DefaultMutableTreeNode("Michigan");
         country.add(state);
         city = new DefaultMutableTreeNode("Ann Arbor");
         state.add(city);
        country = new DefaultMutableTreeNode("Germany");
         root.add(country);
        state = new DefaultMutableTreeNode("Schleswig-Holstein");
         country.add(state);
        city = new DefaultMutableTreeNode("Kiel");
         state.add(city);

       // construct tree and put it in a scroll pane

        JTree tree = new JTree(root);
        Container contentPane = getContentPane();
         contentPane.add(new JScrollPane(tree));
    }
}
```

프로그램을 실행할 때 트리는 먼저 그림 6-6처럼 보인다. 루트 노드와 그 자식노드만 보인다. 원형 아이콘(핸들)을 클릭하면 서브 트리가 펼쳐 진다. 서브 트리가 접혀져 있을 때는 핸들에 붙어 있는 라인이 오른쪽을 가리키고 서브 트리가 펼쳐질 때는 그 라인은 아래쪽을 가리킨다.(그림 6-7) 룩앤필의 디자이너가 무엇을 고려했는지 알 수 없지만 이 아이콘은 문의 손잡이를 생각나게 한다. 핸들을 아래쪽으로 누르면 서브 트리가 펼쳐지게 된다.

노트 : 물론 , 트리의 디스플레이는 선택된 룩앤필에 의존한다. 우리는 자바 룩앤필을 묘사한다.(a.k.a. “Metal”) 윈도우와 모티프 룩앤필에서 핸들은 더 낯익은 룩(박스로 둘러싸인 ”+” 또는 “-“)을 가진다.(그림 6-8을 보라)

그림 6-6 : 초기 트리 디스플레이

그림 6-7 : 확장된 서브 트리와 응집된 서브 트리

그림 6-8 : 윈도우 룩앤필을 갖는 트리

자바 룩앤필은 트리의 연결선을 자동적으로 나타내지는 않는다. 다음의 메소드를 사용하면 놀랍게도 부모와 자식 노드를 연결하는 선이 보이게 된다.

```
Tree.putClientProperty("JTree.lineStyle", "Angled");
```

그 결과는 그림 6-9에 보여진다.

그림 6-9 : 앵글 라인 스타일을 갖는 트리

그림 6-10에서 보여지는 것과 같이 수평선으로 노드들을 구분할 수도 있다. 트리는 자식 노드와 루트 노드를 분리하는 수평 선으로 표시된다. 두가지 형식 중 어떤 것이 더 좋다고 말할 수는 없다.

그림 6-10 : 수평 라인 스타일을 갖는 트리

트리의 루트 노드를 접는 핸들은 자동적으로 생성되지 않는다. 만약 필요하다면 다음 메소드로 그 핸들을 생성할 수 있다.

```
Tree.setShowsRootHandles(true);
```

결과는 그림 6-1에 보여진다. 이제 전체 트리를 루트 노드 내로 접을 수 있다.

그림 6-11: 루트 핸들을 갖는 트리

트리의 집합인 숲을 나타내기 위하여 루트 노드를 숨길 수도 있다. 숲에서의 모든 트리를 공통의 루트 노드로 합쳐야 한다. 그리고 나서, 다음 메소드로 루트 노드를 숨긴다.

```
Tree.setRootVisible(false);
```

그림 6-12를 참고하라. "USA"와 "Germany" 라는 두개의 루트 노드를 볼 수 있다. 그렇지만 이 두개의 루트 노드를 포함하는 부모 루트 노드는 보이지 않는다.

그림 6-12 : 숲(forest)

화제를 루트 노드로부터 트리의 잎 노드로 바꾸자. 그 잎 노드들은 다른 노드들과는 다른 아이콘을 가진다는 것을 알 수 있다.

그림 6-13 : 잎 노트 아이콘

트리를 나타날 때 각 노드는 아이콘으로 표시된다. 아이콘에는 잎 아이콘, 열려진 잎 아이콘 그리고 닫혀진 잎 아이콘의 세 가지 종류가 있다. 뒤의 두 아이콘은 폴더 아이콘이라 부르자.

노드 랜더러는 어느 아이콘이 각 노드에 사용되는지 알 필요가 있다. 그 프로세스는 다음과 같이 작동한다. 임의의 노드의 isLeaf 메소드가 true를 반환한다면 잎 아이콘이 사용된다. 그렇지 않으면 폴더 아이콘이 사용된다.
DefaultMutableTreeNode 클래스의 isLeaf 메소드는 만약 노드가 자식 노드가 없다면 true를 반환한다. 그래서 자식 노드를 가진 노드들은 폴더 아이콘을 가지고 자식노드가 없는 노드들은 잎 아이콘이 된다.
때로는 그러한 기능은 적절하지 않다. 미국의 '주'중의 하나인 "Montana" 라는 노드가 우리 예제 트리에 있다고 가정하자. 그러나 우리는 어떤 도시 노드가 추가될지 알 수 없다. 개념적으로 도시 노드가 잎이기 때문에 주 노드가 잎 아이콘이 되는 것은 적절하지 않다고 생각할 수 있다.
JTree 클래스는 어떤 노드가 잎이 될 지 알 수 없다. 따라서 그것을 트리 모델에 요청한다. 만약 자식 노드 없는 노드가 의미상으로 볼 때 잎 노드가 아니라면 잎 노드에 대해 다른 기준을 사용하는 트리 모델을 요청할 수 있다. 즉 자식 노드 속성을 허용하도록 할 수 있다.
자식 노드를 가지지 않는 노드들에 대해서는 다음과 같은 메소드를 사용한다.

```
node.setAllowsChildren(false)
```

그리하여, 잎 아이콘을 나타내는 노드인지 아닌지를 결정하기 위해 "자식 노드 허용"을 요청할 수 있다. DefaultTreeModel 클래스의 setAsksAllowsChildren 메소드를 사용한다.

```
model.setAsksAllowsChildren(true);
```

이러한 결정 기준을 가지고 자식 노드를 허용하는 노드들은 폴더 아이콘을 사용하고 자식 노드를 가지지 않는 아이콘은 잎 아이콘을 사용한다.
선택적으로 루트 노드를 사용하여 트리를 생성한다면 생성자에서 "자식 노드 허용" 속성을 설정하도록 한다.

```
JTree tree = new JTree(root, true);
        //자식을 허용하지 않는 노드가 잎 아이콘을 가진다.
```

javax.swing.Jtree

- JTree(TreeModel model)
  트리 모델로부터 트리를 생성한다.
- JTree(TreeNode root)
- JTree(TreeNode root , Boolean askAllowChildren)

루트와 루트의 자식들을 디스플레이하는 기본 트리 모델로부터 트리를 생성한다.

전달인자:	root	루트 노드
	askAllowChildren	해당 노드가 리프인지 결정하기 위해 true를 사용하여 자식 노드가 가능하게 한다.

- void setShowsRootHandles(Boolean b)
  만일 b가 true이면, 루트 노드는 하위 자식 노드를 확장하거나 모으기 위한 핸들을 갖게 된다.
- void setRootVisible(boolean b)
  만일 b가 true이면, 루트 노드가 디스플레이된다. 그렇지 않으면 갖춰진다.

javax.swing.tree.TreeNode

- boolean isLead()
  만일 해당 노드가 개념적으로 리프이면 true를 반환한다.
- boolean getAllowChildren()
  해당 노트가 자식 노드를 갖을 수 있으면 true를 반환한다.

javax.swing.tree.MutableTreeNode

- void setUserObject(Object userObject)
  렌더링을 위해 사용하는 트리 노드인 사용자 객체를 설정한다.

javax.swing.tree.TreeModel

- boolean isLeaf(TreeNode node)
  만일 node가 리프 노드로서 디스플레이 된다면 true를 반환한다.

javax.swing.tree.DefaultTreeModel

- void setAsksAllowsChildren(boolean b)
  만일 b가 true이면, 노드들의 getAllowsChildren 메소드가 flase를 반환할 때 디스플레이 된다. 그렇지 않으면 노드들의 isLeaf 메소드가 true를 반환할 때 잎 노드로서 디스플레이된다.

javax.swing.tree.DefaultMutableTreeNode

- DefaultMutableTreeNode(Object userObject)
  주어진 사용자 객체를 갖는 변경가능한 트리 노드를 생성한다.
- void add(MutableTreeNode child)
  해당 노드의 최종 자식 노드로서 노드를 추가한다.
- void setAllowsChildren(boolean b)
  만일 b가 true이면, 자식들은 해당 노드에 추가될 수 있다.

javax.swing.tree.JComponent

- void putClientProperty(object key, Object value)
  각 컴포넌트가 관리하는 작은 테이블에 키/값의 쌍을 추가한다. 이것은 몇몇 스윙 컴포넌트들이 특정 룩앤필 속성을 저장하기 위해서 사용하는 메커니즘이다.

트리와 트리 패스 편집

다음 예제 프로그램에서 트리를 입력하는 방법을 보게 될 것이다. 그림 6-14는 사용자 인터페이스를 보여준다. 만약 "Add Sibling" or "Add Child" 버튼을 클릭한다면 프로그램은 새로운 노드를 트리에 추가할 것이다. 만약 "Delete" 버튼을 누른다면 현재 선택된 노드를 삭제한다.

그림 6-14 : 트리 편집

이러한 기능을 수행하기 위해 어떤 노드가 현재 선택되어 있는지 확인해야 한다. JTree 클래스는 트리에서 노드를 인식하는 놀라운 방법을 가지고 있다. 그것을 위해 트리 노드를 사용하지 않고 트리 패스라 불리는 객체 패스를 사용한다. 트리 패스는 루트 노드와 일련의 자식 노드들로 구성된다.(그림 6-15)

그림 6-15: 트리 경로

JTree 클래스가 왜 전체 패스를 필요로 하는지 궁금할 것이다. TreeNode를 가지고 getParent 메소드를 호출할 수는 없을까? 실제로 JTree클래스는 TreeNode 인터페이스에 대해 아무 정보도 가지지 않는다. 그 인터페이스는 TreeModel 인터페이스에 의해 사용되지 않는다. 단지 DefaultTreeModel을 적용함으로써 사용될 수 있다. 여러분은 TreeNode 인터페이스를 가지지 않는 다른 트리 모델을 사용할 수 있다. 만약 다른 종류의 객체를 가지는 트리 모델을 사용한다면 그 객체에는 getParent와 getChild 메소드가 포함되지 않는다. 물론 서로 다른 연결을 가질 필요가 있다. 노드를 서로 연결하는 것이 트리 모델의 필수 기능이다. JTree 클래스 그 자체로는 다른 연결을 위한 기능이 없다. 이러한 이유로 인해 JTree 클래스는 항상 전체 패스를 가지고 동작하는 것이 필요하다.

TreePath 클래스는 일련의 객체 참고(TreeNode는 아님)를 다룬다. 얼마간의 JTree 메소드는 TreePath 객체를 반환한다. 트리 패스를 사용할 때는 대개 종착 패스를 알 필요가 있는데 그 종착 패스는 getLastPastComponent 메소드로 얻을 수 있다. 예를 들면 트리에서 현재 선택된 노드를 알아내기 위하여 JTree 클래스의 getSelectionPath 메소드를 사용할 수 있다. TreePath 객체를 되돌려 받는다면 그것으로부터 여러분은 작동되고 있는 노드를 다시 알아낼 수 있다.

```
TreePath selectionPath = tree.getSelectionPath();
DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode)
```

```
SelectionPath.getLastPathComponent();
```

실제로, 이러한 질의가 아주 흔히 사용되기 때문에 선택된 노드를 즉시 반환하는 편리한 메소드가 있다.

```
DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode)
    Tree.getlastSelectedPathComponent();
```

이러한 메소드는 트리가 노드를 포함하고 있다는 것을 인식하지 못하기 때문에 getSelectedNode를 호출하지 않는다. 그 트리 모델은 객체의 패스를 사용한다.

노트 : 트리 패스는 JTree 클래스가 노드를 묘사하기 위하여 사용하는 두 가지 방법 중의 하나이다. 열 위치를 나타내는 정수 인덱스를 반환하는 몇 개의 JTree 메소드가 있다. 열 위치는 간단히 트리 디스플레이에서 노드의 열 넘버(0으로 시작함)를 말한다. 보이는 노드는 열 넘버를 가지고 있는데 만약 그 앞의 다른 노드가 펼쳐지거나 접혀지거나 수정된다면 노드의 열 넘버는 바뀐다. 그런 이유로 열 위치를 사용하는 것은 피해야 한다. 열을 사용하는 모든 JTree 메소드는 그 대신에 트리 패스도 가진다.

여러분이 선택된 노드를 가지고 있다면, 그것을 입력할 수 있다. 그러나 간단히 자식노드를 트리 노드에 추가하지 못한다.

```
selectedNode.add(newNode);  // NO!
```

만약 노드의 구조를 바꾼다면 관련된 뷰는 인식하지 못한다. 통지를 보낼 수는 있다. 그러나 만약 DefaultTreeModel 클래스의 insertNodeInto 메소드를 사용한다면 모델 클래스는 그것을 고려한다. 예를 들면 다음 명령이 새로운 노드를 선택된 노드의 마지막 자식 노드로 설정하고 트리 뷰에 통지한다.

```
model.insertNodeInto(newNode, selectedNode,
    selectedNode.getChildCount());
```

유사한 메소드 removeNodeFromParent는 노드를 제거하고 뷰에 알린다.

```
model.removeNodeFromParent(selectedNode);
```

만약 여러분이 적절한 노드 구조를 가지지 않고 사용자 객체를 변화시켰다면 다음 메소드를 사용해야 한다.

```
model.nodeChanged(changedNode);
```

DefaultTreeModel을 사용할 때 자동적으로 통지하는 기능을 사용할 수 있다. 만약 자신의 트리 모델을 가지고 있다면 직접 그러한 기능을 적용해야 한다.(자세한 내용은 Kim Topley의 Core Java Foundation classes를 보라.)

주의: DefaultTreeModel 클래스는 전체 모델을 다시 로드하는 reload 메소드를 가진다. 그러나 몇 가지 변화를 만든 후에 단순히 트리를 갱신하기 위하여 reload를 호출하지 마라. 트리가 다시 배열될 때 자식 노드를 가진 모든 트리는 다시 접혀진다. 만약 변경될 때 마다 트리를 펼친다면 여러분의 사용자들을 혼란스럽게 할 것이다.

노드 구조에 변화가 있다는 것이 뷰에 알려질 때 화면은 갱신되지만 새롭게 추가된 자식 노드가 보여지도록 노드를 자동적으로 펼치지는 않는다. 특히 만약 우리 예제 프로그램에서 사용자가 새로운 자식 노드를 현재 접혀 있는 노드에 추가하면 새로운 노드는 접혀진 서브트리에 추가되어진다. 즉, 사용자에게 그러한 명령이 실제로 실행되었는지를 알려주는 피드백이 없다는 뜻이다. 그러므로 새롭게 추가된 노드를 보려면 모든 부모 노드들을 펼쳐야 한다. 이러한 목적을 위해 JTree 클래스의 makeVisible 메소드를 사용한다. MakeVisible 메소드는 트리 패스를 보이도록 한다.

그래서 루트 노드로부터 새롭게 추가된 노드로의 트리 패스를 생성할 필요가 있다. 트리 패스를 알기 위하여 먼저 DefaultTreeModel 클래스의 getPathToRoot 메소드를 호출한다. 이 기능은 임의의 노드로부터 루트 노드까지 모든 노드의 TreeNode[] 배열을 반환한다. 여러분은 그러한 배열을 TreePath 생성자로 넘겨주어야 한다.

예를 들면, 다음과 같다.

```
TreeNode[] nodes = model.getPathToRoot(newNode);
TreePath path = new TreePath(nodes);
tree.makeVisible(path);
```

노트 : DefaultTreeModel 클래스가 비록 JTree와 통신하는 것이 그것의 기능이지만 TreePath 클래스와 거의 완전히 관계 없는 것처럼 작동한다는 것은 흥미롭다. JTree 클래스는 트리 패스를 많이 사용하고 결코 노드 객체의 배열을 사용하지 않는다.

이제 여러분의 트리가 스크롤 패인 내부에 포함된다고 가정하자. 트리 노드를 펼친 후 새로운 노드는 뷰 포트 외부에 놓이기 때문에 여전히 보이지 않는다. 그 문제를 해결하기 위해 makeVisible를 호출하는 대신에

```
tree.scrollPathToVisible(path);
```

를 호출한다. 이러한 호출은 패스를 따라 모든 노드를 펼치고 주변의 스크롤 패인을 스크롤 가능하게 하여 패스의 끝까지 볼 수 있도록 한다.(그림 6-16)

그림 6-16 : 새로운 노드를 디스플레이 하기 위해 스크롤하는 스크롤 패인

디폴트로 트리 노드가 편집될 수는 없다. 그러나, 만약

```
tree.setEditable(true);
```

를 호출한다면 여러분은 간단히 더블 클릭만으로 노드를 편집할 수 있다. 이것은 DefaultCellEditor 클래스에 의해 적용되는 default cell editor를 호출하는 것이다. 다른 셀 편집

기를 설치하는 것도 가능하다. 그러나 테이블에서는 셀 에디터가 아주 흔히 사용되기 때문에 테이블에 대한 섹션을 설명할 때까지 셀 편집기에 대한 설명을 뒤로 미루기로 한다.
예제 6-2에 트리 편집 프로그램에 대한 전체 소스 코드를 나타내었다. 프로그램을 실행시켜 몇 개의 노드를 추가하고 더블 클릭으로 그 노드들을 편집하라. 추가된 자식 노드가 보이도록 접혀 있는 노드를 펼치는 방법과 뷰 포트에서 추가된 노드가 스크롤 패인에서 어떻게 놓여 있는지 살펴 보라.

예제 6-2 : TreeEditTest.java

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

public class TreeEditTest
{   public  static void main(String[] args)
   {   JFrame frame = new TreeEditFrame();
        frame.show();
    }
}

class TreeEditFrame extends JFrame
   implements  ActionListener
{   public  TreeEditFrame()
    {    setTitle("TreeEditTest");
        setSize(300,  200);
         addWindowListener(new WindowAdapter()
           {   public  void  windowClosing(WindowEvent e)
                {   System.exit(0);
                }
           } );

        // construct tree

        TreeNode root = makeSampleTree();
        model = new DefaultTreeModel(root);
```

```
    tree = new JTree(model);
     tree.setEditable(true);

   // add scroll pane with tree to content pane

    Container contentPane = getContentPane();
    JScrollPane scrollPane = new JScrollPane(tree);
     contentPane.add(scrollPane, "Center");

    // make button panel

    JPanel panel = new JPanel();
    addSiblingButton = new JButton("Add Sibling");
      addSiblingButton.addActionListener(this);
      panel.add(addSiblingButton);
    addChildButton = new JButton("Add Child");
      addChildButton.addActionListener(this);
      panel.add(addChildButton);
    deleteButton = new JButton("Delete");
      deleteButton.addActionListener(this);
      panel.add(deleteButton);
     contentPane.add(panel, "South");
 }

public TreeNode makeSampleTree()
{   DefaultMutableTreeNode root
        = new DefaultMutableTreeNode("World");
    DefaultMutableTreeNode country
        = new DefaultMutableTreeNode("USA");
     root.add(country);
    DefaultMutableTreeNode state
         = new DefaultMutableTreeNode("California");
     country.add(state);
    DefaultMutableTreeNode city
       = new DefaultMutableTreeNode("San Jose");
     state.add(city);
```

```
    city = new DefaultMutableTreeNode("Cupertino");
     state.add(city);
    state = new DefaultMutableTreeNode("Michigan");
     country.add(state);
    city = new DefaultMutableTreeNode("Ann Arbor");
     state.add(city);
    country = new DefaultMutableTreeNode("Germany");
     root.add(country);
    state = new DefaultMutableTreeNode("Schleswig-Holstein");
     country.add(state);
    city = new DefaultMutableTreeNode("Kiel");
     state.add(city);
    return root;
 }

public void actionPerformed(ActionEvent event)
 {   DefaultMutableTreeNode selectedNode
        = (DefaultMutableTreeNode)
           tree.getLastSelectedPathComponent();

   if (selectedNode == null) return;

    if (event.getSource().equals(deleteButton))
    {   if (selectedNode.getParent() != null)
           model.removeNodeFromParent(selectedNode);
        return;
     }

   // add new node as sibling or child

    DefaultMutableTreeNode newNode
       = new DefaultMutableTreeNode("New");

    if (event.getSource().equals(addSiblingButton))
     {   DefaultMutableTreeNode parent
           = (DefaultMutableTreeNode)selectedNode.getParent();
```

```
            if (parent != null)
            {   int selectedIndex = parent.getIndex(selectedNode);
                model.insertNodeInto(newNode, parent,
                    selectedIndex + 1);
             }
          }
          else if (event.getSource().equals(addChildButton))
          {   model.insertNodeInto(newNode, selectedNode,
                  selectedNode.getChildCount());
          }

         // now display new node

          TreeNode[] nodes = model.getPathToRoot(newNode);
         TreePath path = new TreePath(nodes);
          tree.scrollPathToVisible(path);
      }

   private DefaultTreeModel model;
   private JTree tree;
   private JButton addSiblingButton;
   private JButton addChildButton;
   private JButton deleteButton;
   private JButton editButton;
}
```

javax.swing.JTree

- TreePath getSelectionPath()
  현재 선택된 노드에 대한 경로를 얻는다(**또는 복수개의 노드들이 선택되었다면 첫번째 선택된 노드에 대한 경로**). **만일 선택된 노드가 없다면** null**을 반환한다**.
- Object getLastSelectedPathComponent()
  현재 선택된 노드를 대표하는 노드 객체를 얻는다(**또는 복수개의 노드들이 선택되었다면 첫번째 노드가 대한 경로**). **만일 선택된 노드가 없다면** null**을 반환한다**.
- void makeVisible(TreePath path)
  경로를 따라 모든 노드를 확장한다.

- void scrollPathToVisible(TreePath path)
  경로를 따라 모든 노드들을 확장한다. 그리고 만일 트리가 스크롤 패인을 포함한다면 스크롤은 경로상의 최종 노드가 보여지는 것을 보증한다.

javax.swing.tree.TreePath

- object getLastPathComponent()
  해당 경로상의 최종 객체를 얻는다. 예, 경로를 대표하는 노드 객체

javax.swing.tree.TreeNode

- TreeNode getParent()
  해당 노드의 부모 노드를 반환한다.
- TreeNode getChildrenAt(int index)
  주어진 인덱스의 자식 노드를 룩업한다. 인덱스는 0에서부터 getChildCount() – 1 사이여야 한다.
- int getChildrenCount()
  해당 노드의 자식 수를 반환한다.
- Enumeration children()
  해당 노드의 모든 자식에 대한 열거 객체를 반환한다.

javax.swing.tree.DefaultTreeNode

- void insertNodeInto(MutableTreeNode newChild, MutableTreeNode parent, int index)
  주어진 인덱스에 있는 parent에 새로운 자식 노드를 추가한다.
- void removeNodeFromParent(MutableTreeNode node)
  해당 모델로부터 node를 제거한다.
- void nodeChanged(TreeNode node)
  node가 변경됐음을 트리 모델 리스너에게 알린다.
- void nodesChanged(TreeNode parent, int[] changedChildIndexes)
  주어진 인덱스를 갖는 parent의 모든 자식 노드들이 변경됐을 때 트리 모델 리스너들에게 이 사실을 알린다.
- void reload()
  모델내에 모든 노드들을 다시 로드한다. 이것은 몇몇 외부 환경으로 인해 노드들이 완전히 변경됐을 때 이를 이용할 수 있는 아주 과감한 연산이다.

노드 목록

때로는 트리에서 루트 노드에서 시작하여 모든 자식 노드를 조사함으로써 임의의 노드를 찾을 필요가 있다. DefaultMutableTreeNode 클래스는 노드에서 반복 실행하는 몇가지 편리한 메소드를 가지고 있다.

breadthFirstEnumeration과 depthFirstEnumeration 메소드는 깊이 우선 또는 너비 우선 탐색으로 현재 노드의 모든 자식 노드를 조사하는 기능의 nextElement 메소드를 포함하는 목록 객체

를 반환한다. 그림 6-17은 예제 트리에서의 탐색을 보여 준다. 노드 라벨은 노드를 탐색하는 순서를 나타낸다. 너비 우선 목록은 그림을 살펴 보면 이해하기 싶다. 트리는 각 층을 탐색한다. 먼저 루트 노드를 탐색하고 그 다음 자식 노드 그리고 나서 손자 노드를 탐색한다.
깊이 우선 목록을 이해하기 위해 트리 형태의 미로에 빠진 쥐를 상상하라. 맨먼저, 잎 노드에 도달할 때까지 첫번째 패스를 탐색한다. 그 다음 되돌아 와서 다음 패스를 탐색한다. 이런 방법으로 모든 패스를 탐색한다.

컴퓨터 과학 형식은 탐색 과정이 먼저 부모를 만나기 전에 자식들에 대해 먼저 조사하기 때문에 이런 후순위(postorder) 탐색을 사용한다고 할 수 있다. postOrderTraversal 메소드는 depthFirstTraversal와 같은 의미를 가진다. 그렇지만 완전히 탐색하기 위해 자식을 만나기 전에 부모를 먼저 조사하는 깊이 우선 탐색인 PreOrderTraversal도 역시 필요하다.
전형적인 사용 패턴은 다음과 같다.

```
Enumeration breadthFirst = node.breadthFirstEnumeration();
While (breadthFirst.hasMoreElements())
  do something with breadthFirst.nextElement();
```

그림 6-17 : 트리 탐색 순서

그리고, 조상 노드에서 주어진 노드까지의 패스를 조사하고 그 패스를 따르는 노드를 목록화하는 pathFromAncestorEnumeration 메소드가 있다. 조상 노드를 발견해서 그 패스를 얻을 때까지 getParent를 계속해서 호출한다.
우리의 다음 예제 프로그램에서 노드 목록을 작성하도록 한다. 프로그램은 클래스의 상속 트리를 보여준다. 프레임의 하단에 있는 텍스트 박스에 클래스의 이름을 기입하라. 그 클래스와 모든 슈퍼클래스는 트리에 추가된다.(그림 6-18)
이 예제에서 트리 노드의 사용자 객체가 어떤 형태의 객체도 될 수 있다는 사실을 이용하라. 노드가 클래스를 상징하기 때문에 우리는 그 노드에 Class 객체를 저장한다. 물론 우리는 같은 클래스 객체를 두 번 추가하는 것을 바라지 않는다. 그래서 트리에 클래스가 이미 존재하는지를 검토할 필요가 있다. 다음 메소드로 임의의 객체가 트리에 존재한다면 트리에서 사용자 객체를 가지는 노드를 찾을 수 있다.

```
public DefaultMutableTreeNode findUserObject(Object obj)
        {   Enumeration e = root.breadthFirstEnumeration();
            while (e.hasMoreElements())
            {   DefaultMutableTreeNode node
                    = (DefaultMutableTreeNode)e.nextElement();
```

```
            if (node.getUserObject().equals(obj))
                return node;
        }
        return null;
    }
```

그림 6-18 : 상속 트리

랜더링 노드

여러분의 어플리케이션에서 종종 트리 컴포넌트가 노드를 만들어 내는 방법을 바꿀 필요가 있다. 가장 알기 쉬운 변화는 물론 노드와 잎을 나타내는 다른 아이콘을 선택한다는 것이다. 다른 것은 노드 라벨의 폰트를 변화시키는 것과 노드에 이미지를 나타내는 것이다. 모든 이런 기능은 새로운 tree cell renderer**를 트리에 설치함으로써 가능하다. 디폴트로** JTree **클래스는** DefaultTreeCellRenderer **객체를 사용하여 각 노드를 그려낸다.** DefaultTreeCellRenderer **클래스는** JLabel **클래스를 확장한다. 라벨은 노드 아이콘과 노드 라벨을 포함한다.**

노트 : 셀 랜더러는 서브 트리를 펼치거나 접기 위한 "핸들"을 나타내지 않는다. 핸들은 룩앤필의 일부분이고 그것을 변화시키지 않는 것이 좋다.

여러분은 세가지 방법으로 디스플레이를 만들 수 있다.

1. DefaultTreeCellRenderer**을 사용하여 아이콘,** 폰트 **그리고 배경색을 변화시킬 수 있다. 이러한 기능은 트리의 모든 노드에 사용될 수 있다.**
2. DefaultTreeCellRenderer **클래스를 확장하여 각 노드에서 아이콘,** 폰트, **배경색을 바꿀 수 있는 랜더러를 설치할 수 있다.**
3. **각 노드에** TreeCellRenderer **인터페이스를 적용하여 이미지를 표시하는 랜더러를 사용할 수 있다.**

차례로 이러한 가능성을 살펴 보자. 가장 쉬운 방법은 DeaultTreeCellRenderer **객체를 생성하여 아이콘을 바꾼 다음 그것을 트리에 설치하는 것이다.**

```
DefaultTreeCellRenderer renderer
        = new DefaultTreeCellRenderer();
renderer.setLeafIcon(new ImageIcon("blue-ball.gif"));
        // 잎 노드를 위해 사용됨
renderer.setClosedIcon(new ImageIcon("red-ball.gif"));
        // 접힌 노드를 위해 사용됨
renderer.setOpenIcon(new ImageIcon("yellow-ball.gif"));
        // 펼친 노드를 위해 사용됨
```

tree.setCellRenderer(renderer);

여러분은 338 페이지의 그림 6-18에서 이 효과를 볼 수 있다. 우리는 "ball" 아이콘을 사용할 수 있다. 여러분의 사용자 인터페이스 디자이너는 여러분의 어플리케이에 사용될 적절한 아이콘을 공급할 것이다.

여러분이 전체 트리의 폰트와 배경색을 바꾸지 않는 것이 좋다. 그것은 룩앤필의 일이다.

그러나, 그것 중 몇 개를 반전시키기 위하여 트리에서 각각의 노드의 폰트를 변화시키는 것이 유용할 수 있다. 만약 그림 6-18을 주의 깊게 본다면 추상클래스가 이탤릭체로 설정되는 것을 볼 수 있을 것이다.

각각의 노드의 외양을 바꾸기 위하여 트리 셀 랜더러를 설치해야 한다. 트리 셀 랜더러는 1권 9장에서 설명한 리스트 셀 랜더러와 매우 유사하다. TreeCellRenderer 인터페이스는 싱글 메소드

```
Component getTreeCellRendererComponent(Jtree tree, Object value,
                    Boolean selected, Boolean expanded,
                    Boolean leaf, int row, Boolean hasFocus)
```

를 가지고 있다.

이 메소드는 각 노드에 호출된다. 그것은 그것의 paint 메소드가 트리 디스플레이에서 노드를 표현하는 컴포넌트를 반환한다. Paint 메소드는 적절한 Graphics 객체로써 얻어진다. 그 Graphics 객체는 축과 사각형 세트를 가짐으로써 그림을 원하는 위치에 표시한다.

노트 : paint 메소드를 트리 셀 랜더러 인터페이스를 적용하는 클래스에 두지 않는 이유가 궁금할 것이다. 매우 그럴듯한 이유가 있다. 실제로 뭔가를 나타내는 코드를 프로그램하기 위하여 존재하는 컴포넌트를 사용하는 것이 보다 쉽다. 예를 들면, 디폴트 트리 셀 랜더러는 단순히 JLabel을 확장하고 그 라벨을 아이콘과 라벨 텍스트 사이의 적절한 공간에 위치시킨다.

DefaultTreeCellRenderer 클래스의 getTreeCellRendererComponent 메소드는 this 즉 라벨을 반환한다. (DefaultTreeCellRenderer 클래스는 Jlabel 클래스를 상속한다는 것을 기억하라.) 적절한 컴포넌트를 만들기 위하여 DefaultTreeCellRenderer 클래스를 확장하라. 다음과 같이 getTreeCellRendererComponent 메소드를 오버라이드하라. 슈퍼클래스 메소드를 호출하면, 라벨 데이터를 생성할 수 있다. 적절한 라벨 속성을 지정한 후 this를 반환하라.

```
class MyTreeCellRenderer extends DefaultTreeCellRenderer
{   public Component getTreeCellRendererComponent(Jtree tree,
        Object value, boolean selected, Boolean expanded,
        boolean leaf, int row, Boolean hasFocus)
    { super.getTreeCellRendererComponent(tree, value,
```

```
                selected, expanded, leaf, row, hasFocus) ;
                DefaultMutableTreeNode node
                    = (DefaultMutableTreeNode)value;
                look at node.getUserObject();
                Font font = appropriate font;
                setFont(font);
                return this;
            }
        };
```

주의 : getTreeCellRendererComponent 메소드의 value 인자는 노드 객체이지 사용자 객체가 아니다. 사용자 객체는 DefaultMutableTreeNode의 특징이라는 것을 기억하라. 그리고 JTree는 임의의 타입의 노드를 포함할 수 있다. 만약 여러분의 트리가 DefaultMutableTreeNode 노드를 사용한다면 이전 코드 예제에서 했던 것처럼 두 번째 단계에서 사용자 객체를 회복해야 한다.

주의 : DefaultTreeCellRenderer은 각 노드에 라벨 텍스트를 변화시킴으로써 모든 노드에 같은 라벨 객체를 사용한다. 만약 특정한 노드의 폰트를 바꾼다면 그 메소드가 다시 호출될 때 그것의 원래 상태로 복원시켜주어야 한다. 그렇지 않으면 그 다음 노드는 바뀐 폰트로 표시될 것이다. 예제 6-3의 코드를 보고 원래 상태로 폰트를 복원시키는 방법을 살펴 보라.

임의의 그래픽을 만드는 트리 셀 랜더러에 대한 예제는 제공하지 않을 것이다. 1권의 9장에서 리스트 셀 랜더러를 볼 수 있다. 그 기술과 아주 유사하므로 참고하기 바란다.

이제 트리 셀 랜더러를 작동시켜 보자. 예제 6-3은 클래스 트리 프로그램에 대한 전체 소스 코드를 나타낸다. 그 프로그램은 상속 계층구조를 나타낸다. 그리고 그것은 이탤릭으로 추상 클래스를 나타내기 위한 디스플레이를 적절하게 표현한다. 임의의 클래스의 이름을 프레임 바닥의 텍스트 필드에 입력할 수 있다. 클래스와 그것의 슈퍼클래스를 트리에 추가하기 위하여 ENTER 키를 눌러라.(이것은 스텔러(stellar) 사용자 인터페이스가 아니지만, 텍스트 필드에서 ENTER를 누르는 것이 액션 이벤트를 발생시키기 때문에 프로그램하기 쉽다.) 전체 패키지 이름을 java.util.Vector과 같이 입력해야 한다.
이 프로그램은 클래스 트리를 생성하기 위해 리플렉션을 사용하기 때문에 약간 복잡하다. 이 작업은 addClass 메소드 내에 포함되어 있다. (상세한 것은 중요하지 않다. 상속 트리가 어려운 코딩 없이 훌륭한 트리를 생성하기 때문에 이 예제에서 클래스 트리를 사용한다. 만약 자신의 어플리케이션에서 트리를 나타낸다면 계층 구조 데이터에 대한 여러분의 소스를 가지게 될 것이다.) 먼저 그 메소드는 현재의 클래스가 이전의 섹션에서 적용한

findUserObject 메소드를 호출함으로써 트리에 현재의 클래스가 이미 존재하는지 어떤지를 알아내기 위하여 가지 우선 탐색을 사용한다. 만약 그 클래스가 이미 트리에 존재하고 있지 않다면 슈퍼클래스를 트리에 추가한다. 그리고 나서 새로운 클래스 노드를 자식노드로 만들고 그 노드를 볼 수 있도록 한다.

ClassNameTreeCellRenderer는 Class 객체의 ABSTRACT 수정자에 의존해서 일반적인 폰트나 이탤릭체로 클래스 이름을 설정한다. 룩앤필이 라벨로 어떤 폰트라도 사용할 수 있게 하는데 이러한 특성을 변화시킬 필요가 없기 때문에 특정한 폰트를 설정하는 것은 바람직하지 않다. 이러한 이유로 라벨로부터의 폰트를 사용하고 그것으로부터 이탤릭 폰트를 이끌어 낸다. 모든 호출에 의해 반환되는 JLabel 객체가 있다는 것을 기억하라. 원래의 폰트에 설정해서 getTreeCellRendererComponent 메소드로 다음 호출에 그것을 복원할 수 있다.

마침내 우리는 ClassTreeFrame 생성자에서 노드 아이콘을 바꾸는 방법을 이해할 수 있게 되었다.

예제 6-3 : ClassTree.java

```
import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.tree.*;

public class ClassTree
{  public static void main(String[] args)
   {  JFrame frame = new ClassTreeFrame();
      frame.show();
   }
}

class ClassTreeFrame extends JFrame
   implements ActionListener
{  public ClassTreeFrame()
   {  setTitle("ClassTree");
      setSize(300, 200);
      addWindowListener(new WindowAdapter()
```

```
        {   public void windowClosing(WindowEvent e)
            {   System.exit(0);
            }
        } );

   // the root of the class tree is Object
    root = new DefaultMutableTreeNode(java.lang.Object.class);
    model = new DefaultTreeModel(root);
    tree = new JTree(model);

   // add this class to populate the tree with some data
     addClass(getClass());

    // set up node icons
     ClassNameTreeCellRenderer renderer
        = new ClassNameTreeCellRenderer();
     renderer.setClosedIcon(new ImageIcon("red-ball.gif"));
     renderer.setOpenIcon(new ImageIcon("yellow-ball.gif"));
     renderer.setLeafIcon(new ImageIcon("blue-ball.gif"));
      tree.setCellRenderer(renderer);

    Container contentPane = getContentPane();
    contentPane.add(new JScrollPane(tree), "Center");

   // new class names are typed into this text field
    textField = new JTextField();
     textField.addActionListener(this);
    contentPane.add(textField, "South");
 }

public void actionPerformed(ActionEvent event)
{  // add the class whose name is in the text field
     try
    {   String text = textField.getText();
         addClass(Class.forName(text));
       // clear text field to indicate success
```

```
            textField.setText("");
        }
        catch (ClassNotFoundException e)
        {   Toolkit.getDefaultToolkit().beep();
        }
    }

public DefaultMutableTreeNode findUserObject(Object obj)
{   // find the node containing a user object
    Enumeration e = root.breadthFirstEnumeration();
    while (e.hasMoreElements())
    {   DefaultMutableTreeNode node
            = (DefaultMutableTreeNode)e.nextElement();
        if (node.getUserObject().equals(obj))
            return node;
    }
    return null;
}

public DefaultMutableTreeNode addClass(Class c)
{   // add a new class to the tree

     // skip non-class types
   if (c.isInterface() || c.isPrimitive()) return null;

  // if the class is already in the tree, return its node
    DefaultMutableTreeNode node = findUserObject(c);
   if (node != null) return node;

    // class isn't present--first add class parent recursively

    Class s = c.getSuperclass();

    DefaultMutableTreeNode parent;
   if (s == null)
        parent = root;
```

```
            else
               parent = addClass(s);

         // add the class as a child to the parent
         DefaultMutableTreeNode newNode = new DefaultMutableTreeNode(c);
         model.insertNodeInto(newNode, parent, parent.getChildCount());

         // make node visible
         TreePath path = new TreePath(model.getPathToRoot(newNode));
         tree.makeVisible(path);

         return newNode;
      }

   private DefaultMutableTreeNode root;
   private DefaultTreeModel model;
   private JTree tree;
   private JTextField textField;
}

class ClassNameTreeCellRenderer extends DefaultTreeCellRenderer
{  public Component getTreeCellRendererComponent(JTree tree,
      Object value, boolean selected, boolean expanded,
      boolean leaf, int row, boolean hasFocus)
   {  super.getTreeCellRendererComponent(tree, value,
         selected, expanded, leaf, row, hasFocus);
      // get the user object
      DefaultMutableTreeNode node = (DefaultMutableTreeNode)value;
      Class c = (Class)node.getUserObject();

      // the first time, derive italic font from plain font
      if (plainFont == null)
      {  plainFont = getFont();
         /* the tree cell renderer is sometimes called with a
            label that has a null font
         */
```

```
            if (plainFont != null)
                italicFont = plainFont.deriveFont(Font.ITALIC);
        }

      // set font to italic if the class is abstract
       if ((c.getModifiers() & Modifier.ABSTRACT) == 0)
            setFont(plainFont);
         else
            setFont(italicFont);
        return this;
    }

   private Font plainFont = null;
   private Font italicFont = null;
};
```

javax.swing.tree.DefaultMutableTreeNode

- Enumeration breadthFirstEnumeration()
- Enumeration depthFirstEnumeration()
- Enumeration preOrderTraversal()
- Enumeration postOrderTraversal()

  특정 순서로 트리 모델내의 모든 노드들을 방문하기 위한 열거 객체를 반환한다. BreadthFirst 탐색은 루트에 근접해 있는 자식들을 먼저 방문한다. DepthFirs 탐색은 동등 노드를 방문하기 전에 노드의 모든 자식들을 완전히 열거한다. PostOrderTraversal 메소드는 depthFirstEnumeration와 동의어이다. PreOrderTraversal은 자식을 방문하기 전에 부모들이 열거된다는 점을 제외하고는 postOrderTraversal과 동일하다.

javax.swing.tree.TreeCellRender

- Component getTreeCellRenderComponent(JTree tree, Object value, Boolean selected, Boolean expanded, Boolean leaf, int row, Boolean hasFocus)

  트리 셀을 렌더링하기 위해 호출되는 paint 메소드를 갖는 컴포넌트를 반환한다.

전달인자:		
	tree	렌더링될 노드를 포함하고 있는 트리
	value	렌더링될 노드
	selected	노드가 현재 선택되었다면 true를 이다.
	expanded	노드의 자식이 보여지면 true를 이다.

reaf 노드가 트리로서 디스플레이될 필요가 있다면 true이다.

row 노드를 포함하고 있는 열을 디스플레이한다.

HasFocus 현재 노드가 입력 포커스를 갖고 있다면 true이다.

javax.swing.tree.TreeCellRender

- void setLeafIcon(Icon icon)
- void setOpenIcon(Icon icon)
- void setClosedIcon(Icon icon)

리프 노드와 확장 노드, 모아진 노드를 보여주기 위한 아이콘을 설정한다.

트리 이벤트에 대한 리스닝

아주 흔하게 트리 컴포넌트는 몇가지 다른 컴포넌트와 쌍이 될 수 있다. 사용자가 트리 노드를 선택할 때, 몇가지 정보는 다른 윈도우에 나타난다. 그림 6-19를 보라 사용자가 클래스를 선택할 때, 그 클래스의 인스턴스와 정적 변수는 오른쪽의 텍스트 영역에서 표시된다.

그림 6-19 : 클래스 브라우져

이러한 기능을 얻기 위하여 tree selection listener를 설치한다. 리스너는 메소드

```
void valueChanged(TreeSelectionEvent event)
```

를 가진 TreeSelectionListener 인터페이스를 적용해야 한다.
그 메소드는 사용자가 트리 노드를 선택하거나 선택을 해제할 때는 언제나 호출한다.
일반적인 방법으로 트리에 리스너를 추가해야 한다.

```
tree.addTreeSelectionListener(listener);
```

사용자가 싱글 노드, 인접한 노드 범위 또는 임의의 인접하지 않은 일련의 노드들을 선택하도록 허용할 지 아닐지를 정할 수 있다. JTree 클래스는 노드 선택을 다루기 위하여 TreeSelectionModel을 사용한다. 선택 상태는 SINGLE_TREE_SELECTION, CONTIGUOUS_TREE_SELECTION, 또는 DISCONTIGUOUS_TREE_SELECTION중의 하나에 설정되도록 할 필요가 있다.(Discontiguous 선택 모드가 디폴트이다.) 예를 들면, 클래스 브라우저에서 단지 싱글 클래스의 선택이 허용되기를 원한다면 코드는 다음과 같다.

```
int mode = TreeSelectionModel.SINGLE_TREE_SELECTION;
tree.getSelectionModel().setSelectionMode(mode);
```

선택 모드를 설정하는 것과 관계없이 트리 선택 모델에 대하여 걱정할 필요가 없다.

노트 : 사용자가 중복된 아이템을 선택하는 방법은 룩앤필에 의존한다. 메탈 룩앤필에서 아이템을 선택에 추가하려면(또는 현재 선택된 아이템을 선택에서 제거하려면) CTRL키를 누른 상태에서 그 아이템을 클릭하면 된다. 이전에 선택된 아이템에서 새로운 아이템까지 선택영역으로 정하려면 SHIFT 키를 누른 상태에서 아이템을 클릭하라.

현재의 어떤 것이 선택되었는지 알기 위하여 트리에 getSelectionPaths 메소드를 사용한다.

```
TreePath[] selectedPaths = tree.getSelectionPaths();
```

만약 사용자에게 싱글 선택으로 제한하려면 첫번째 선택된 패스를 반환하는(어떤 패스도 선택되지 않았으면 null을 반환한다.) 편리한 메소드 getSelectionPath를 사용한다.

주의 : TreeSelectionEvent 클래스는 TreePath 객체의 배열을 반환하는 getPaths 메소드를 가진다. 그러나 그 배열은 선택의 변화를 묘사하고 현재 선택된 것을 묘사하지 않는다.

예제 6-4에서 트리 선택이 작동하는 것을 볼 수 있다. 이 프로그램은 예제 6-3을 참고하여 만들었다. 그러나 관례적인 트리 셀 랜더러를 사용하지 않았기 때문에 프로그램은 더 짧다. 프레임 생성자에서 사용자는 싱글 아이템 선택으로 제한하고 트리 선택 리스너를 추가한다. ValueChanged 메소드가 호출될 때, 그것의 이벤트 파라미터를 무시하고 단지 현재의 선택된 패스를 위한 트리를 요청한다. 항상 패스의 마지막 노드를 얻어 사용자 객체를 검토할 필요가 있다. 그 때 선택된 클래스의 모든 필드로 문자열을 구성하기 위하여 리플렉션을 사용하는 getFieldDescription 메소드를 호출한다. 그리하여 그 문자열은 텍스트 영역에 표시된다.

예제 6-4 : ClassBrowserTree.java

```
import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.tree.*;

public class ClassBrowserTest
{  public static void main(String[] args)
   {  JFrame frame = new ClassBrowserTestFrame();
      frame.show();
   }
```

```
}

class ClassBrowserTestFrame extends JFrame
   implements ActionListener, TreeSelectionListener
{  public ClassBrowserTestFrame()
   {   setTitle("ClassBrowserTest");
       setSize(300, 200);
       addWindowListener(new WindowAdapter()
          {  public void windowClosing(WindowEvent e)
             {  System.exit(0);
             }
          } );

      // the root of the class tree is Object
       root = new DefaultMutableTreeNode(java.lang.Object.class);
       model = new DefaultTreeModel(root);
       tree = new JTree(model);

       // add this class to populate the tree with some data
        addClass(getClass());

       // set up selection mode
        tree.addTreeSelectionListener(this);
       int mode = TreeSelectionModel.SINGLE_TREE_SELECTION;
        tree.getSelectionModel().setSelectionMode(mode);

      // this text area holds the class description
       textArea = new JTextArea();

      // add tree and text area to the content pane
       JPanel panel = new JPanel();
       panel.setLayout(new GridLayout(1, 2));
       panel.add(new JScrollPane(tree));
       panel.add(new JScrollPane(textArea));

       Container contentPane = getContentPane();
```

```
      contentPane.add(panel, "Center");

    // new class names are typed into this text
     textField = new JTextField();
      textField.addActionListener(this);
      contentPane.add(textField, "South");
 }

public void actionPerformed(ActionEvent event)
{  // add the class whose name is in the text field
       try
     {   String text = textField.getText();
           addClass(Class.forName(text));
        // clear text field to indicate success
           textField.setText("");
       }
     catch (ClassNotFoundException e)
      {    Toolkit.getDefaultToolkit().beep();
      }
 }

public void valueChanged(TreeSelectionEvent event)
 {   // the user selected a different node--update description
     TreePath path = tree.getSelectionPath();
     if (path == null) return;
      DefaultMutableTreeNode selectedNode
          = (DefaultMutableTreeNode)path.getLastPathComponent();
     Class c = (Class)selectedNode.getUserObject();
     String description = getFieldDescription(c);
      textArea.setText(description);
 }

public DefaultMutableTreeNode findUserObject(Object obj)
{  // find the node containing a user object
      Enumeration e = root.breadthFirstEnumeration();
      while (e.hasMoreElements())
```

```
    {   DefaultMutableTreeNode node
            = (DefaultMutableTreeNode)e.nextElement();
        if (node.getUserObject().equals(obj))
            return node;
    }
    return null;
}

public DefaultMutableTreeNode addClass(Class c)
{  // add a new class to the tree

    // skip non-class types
   if (c.isInterface() || c.isPrimitive()) return null;

  // if the class is already in the tree, return its node
    DefaultMutableTreeNode node = findUserObject(c);
   if (node != null) return node;

   // class isn't present--first add class parent recursively

   Class s = c.getSuperclass();

    DefaultMutableTreeNode parent;
   if (s == null)
        parent = root;
    else
       parent = addClass(s);

   parent = addClass(s);

  // add the class as a child to the parent
   DefaultMutableTreeNode newNode = new DefaultMutableTreeNode(c);
    model.insertNodeInto(newNode, parent, parent.getChildCount());

   // make node visible
   TreePath path = new TreePath(model.getPathToRoot(newNode));
```

```
        tree.makeVisible(path);

        return newNode;
    }

  public static String getFieldDescription(Class c)
  {  // use reflection to find types and names of fields
     String r = "";
     Field[] fields = c.getDeclaredFields();
    for (int i = 0; i < fields.length; i++)
     {  Field f = fields[i];
        if ((f.getModifiers() & Modifier.STATIC) != 0)
            r += "static ";
        r += f.getType().getName() + " ";
        r += f.getName() + "\n";
      }
     return r;
   }

  private DefaultMutableTreeNode root;
  private DefaultTreeModel model;
  private JTree tree;
  private JTextField textField;
  private JTextArea textArea;
}
```

javax.swing.JTree

- Path getSelectionPath()
- Path[] getSelectionPaths()
  첫번째 선택된 경로나 모든 선택된 노드들에 대한 경로의 배열을 반환한다. 만일 선택된 경로가 없다면 , 이 두 메소드는 null**을 반환한다.**

javax.swing.event.TreeSelectionListener

- void valueChanged(TreeSelectionEvent event)
  노드들이 선택되거나 선택 해제될 때마다 호출된다.

javax.swing.event.TreeSelectionEvent

- TreePath getPath()
- TreePath[] getPaths()
  해당 선택 이벤트내에서 변경된 첫번째 경로나 모든 경로를 얻는다. 만일 여러분이 선택 변경이 아니라 현재 선택을 알길 원한다면, 여러분은 JTree.getSelectionPaths를 대신 호출해야 한다.

트리모델 커스텀

마지막 예제에서 디버거가 하는 것처럼(그림 6-20을 보라) 변수의 내용을 검사하는 프로그램을 적용한다.

그림 6-20 : 객체 검사 트리

더 진행하기 전에 예제 프로그램을 컴파일하고 실행하라. 각 노드는 인스턴스 변수와 일치한다. 만약 변수가 객체라면 그것의 인스턴스 변수를 보기 위하여 그것을 펼쳐라. 그 프로그램은 프레임 윈도우의 내용을 검사한다. 만약 몇몇 인스턴스 변수를 선택하면 낯익은 클래스를 발견할 수 있다. 그리고 스윙 사용자 인터페이스가 얼마나 복잡한지 알 수 있다.

이 프로그램에 대해 주목할 만한 것은 트리가 DefaultTreeModel을 사용하지 않는다는 것이다. 만약 계층적으로 구조화된 데이터를 이미 가지고 있다면 장착된 트리를 생성하거나 두 트리를 계속해서 동기화하는 것에 대해 생각할 필요가 없을 것이다. 검사된 객체는 객체 참조를 통하여 서로 연결되어 있어 연결 구조를 반복할 필요가 없다.

TreeModel 인터페이스는 단지 몇 개의 메소드를 가지고 있다. 메소드의 첫번째 그룹은 JTree가 먼저 루트 노드를 얻고 다음에 자식 노드를 얻음으로써 트리 노드를 찾아내는 것이 가능하도록 한다. JTree 클래스는 사용자가 실질적으로 노드를 펼칠 때 단지 이러한 메소드를 호출한다.

```
Object getRoot()
int getChildCount(Object parent)
Object getChild(Object parent, int index)
```

JTree 클래스 그 자체와 같이 TreeModel 인터페이스는 노드의 어떤 개념도 가지지 않는다. 루트 노드와 그 자식 노드는 임의의 객체가 될 수 있다. TreeModel은 그들이 연결되는 방법을 JTree에 알려줄 책임이 있다.

TreeModel 인터페이스의 다음 메소드는 getChild의 역이다.

```
int getIndexOfChild(Object parent, Object child)
```

실질적으로, 이러한 메소드는 첫번째 세 개에 적용될 수 있다. 예제 6-5에서 코드를 보라.
트리 모델은 JTree에 어떤 노드가 잎 노드로서 표시될지를 알려 준다.

```
boolean isLeaf(Object node)
```

만약 여러분의 코드가 트리 모델을 바꾼다면 트리는 다시 표시하도록 통지를 받을 필요가 있다. 트리는 그 자체를 TreeModelListener로서 모델에 추가한다. 그리하여 모델은 일반적인 리스너 관리 메소드를 지원해야만 한다.

```
void addTreeModelListener (TreeModelListener l)
void removeTreeModelListener (TreeModelListener l)
```

예제 6-5에서 이러한 메소드를 위한 적용을 볼 수 있다.
모델이 트리 내용을 수정할 때 다음 TreeModelListener 인터페이스

```
void treeNodesChanged(TreeModelEvent e)
void treeNodesInserted(TreeModelEvent e)
void treeNodeRemoved(TreeModelEvent e)
void treeStructureChanged(TreeModelEvent e)
```

의 4가지 메소드 중 하나를 호출한다.
TreeModelEvent 객체는 변화된 위치를 묘사한다. 삽입 이벤트 또는 제거 이벤트를 표시하는 트리 모델 이벤트를 배열하는 것에 대한 상세한 내용은 상당히 기술적이다. 만약 실질적으로 추가되거나 제거된 노드를 가질 수 있다면 이러한 이벤트를 사용하는 것은 바람직하지 않다. 예제 6-5에서 하나의 이벤트를 사용하는 방법을 보여준다. 이 때 루트 노드는 새로운 객체로 대체된다.

TIP : 스윙 프로그래머는 리스너를 집합시킨 클래스 javax.swing.EventListenerList를 공급하는 이벤트 체계에 싫증을 느꼈을 것이다. API 노트에서는 그 클래스를 사용하는 방법을 설명한다.

그리하여 사용자가 트리 노드를 입력하면 여러분의 모델은 변화가 있다는 것을 알린다.

```
void valueForPathChanged(TreePath path, Object newValue)
```

만약 입력을 허용하지 않는다면 이 메소드는 결코 호출되지 않는다.
만약 입력을 제공할 필요가 없다면 트리 모델을 생성하는 것은 매우 쉽다.
다음 세가지 메소드를 적용한다.

```
Object getRoot()
int getChildCount(Object parent)
Object getChild(Object parent, int index)
```

이러한 메소드는 트리의 구조를 묘사한다. 그림 6-5는 다른 다섯개의 메소드의 적용을 보여준다. 여러분의 트리를 나타낼 준비를 하자. 예제 프로그램의 적용으로 돌아가자. 우리의 트리는 Variable 타입의 객체를 포함할 것이다.

노트 : DefaultTreeModel을 사용했다면 노드는 타입 Variable의 사용자 객체를 가진 타입 DefaultMutableTreeNode의 객체가 되었을 것이다.

예를 들면, 변수

Employee joe:

를 검토한다고 가정하자.

그 변수는 타입 Employee.class, 이름 "joe" 그리고 객체 참조 joe의 값을 가지고 있다. 프로그램에서 변수를 묘사하는 클래스 Variable를 정의한다.

Variable v = new Variable(Employee.class, "joe", joe);

만약 변수의 타입이 기본적인(primitive) 타입이라면 값을 위하여 객체 포장을 사용해야 한다.

new Variable(double.class, "salary", new Double(salary));

만약 변수의 타입이 클래스라면 변수는 필드를 가진다.. 리플렉션을 사용해서, 모든 필드를 열거해서 ArrayList에 그것들을 넣는다. Class 클래스의 GetFields 메소드는 슈퍼클래스의 필드를 반환하지 않기 때문에 모든 슈퍼클래스에 getFields를 호출할 필요가 있다. Variable 생성자에서 그 코드를 볼 수 있다. Variable 클래스의 getFields 메소드는 필드의 배열을 반환한다. 그리고, Variable 클래스의 toString 메소드는 노드 라벨의 포멧을 정한다. 그 라벨은 항상 Variable 타입과 이름을 포함한다. 만약 그 Variable이 클래스가 아니라면 라벨은 그 값을 포함한다.

노트 : 만약 타입이 배열이라면 그 배열의 요소를 나타내지는 않는다. 이것을 하는 것은 어렵지 않다. 그것은 "독자들을 위한 연습 문제"로 남기겠다.

트리 모델에 대해 생각해 보자. 먼저 두 메소드는 매우 간단하다.

```
public Object getRoot()
{       return root;
}

public int getChildCount (Object parent)
{       return ((Variable)parent).getFields().size();
}
```

getChild 메소드는 특정한 필드를 묘사하는 새로운 Variable 객체를 반환한다. 필드 클래스의 getType와 getName 메소드는 필드 타입과 이름을 산출한다. 리플렉션을 사용함으로써 f.get(parentValue)으로 필드 값을 읽을 수 있다.

그 메소드는 IllegalAcessException을 던질 수 있다. 그러나 Variable 생성자에서 접근 가능한 모든 필드를 만들었다. 이것은 실제로 일어나지 않을 것이다.

다음에 getChild 메소드의 전체 코드를 나타내었다.

```
public object getChild(Object parent, int index)
{ ArrayList fields = ((Variable)parent).getFields();
  Field f = (Field) fields.get(index);
  Object parentValue = ((Variable)parent).getValue();
   try
  { return new Variable(f.getType(), f.getName(),
            f.get(parentValue));
   }
   catch(IllegalAccessException e)
   { return null;
   }
}
```

이러한 세가지 메소드는 JTree 컴포넌트에 객체 트리의 구조를 전달한다. 남아 있는 메소드는 간단한 것들이다. 예제 6-5의 소스 코드를 보라. 이러한 트리 모델에 대하여 주목할 만한 사실이 있다. 그것은 무한 트리를 말한다. WeakReference 객체들 중의 하나를 살펴봄으로써 이것을 증명할 수 있다. referent라 이름 지어진 변수를 클릭하라. 그것은 여러분을 원래 객체의 바로 뒤로 인도한다. 여러분은 이상적인 서브 트리를 얻고 다시 그것의 WeakReference 객체를 열 수 있다. 물론 여러분은 무한한 노드를 저장할 수는 없다. 사용자가 부모 노드를 확대하고자 할 때, 트리 모델은 간단히 요구대로 노드를 발생시킨다.

이 예제로 트리에 대한 설명을 마무리 한다. 이제 다른 스윙 컴포넌트인 테이블 컴포넌트에 대해 설명할 것이다. 외관상으로 트리와 테이블은 많은 공통점을 가지고 있는 것처럼 보이지 않는다. 그러나 그들 둘 다 데이터 모델과 셀 랜더링에 대해 같은 개념을 사용하고 있다는 것을 발견하게 될 것이다.

예제 6-5 : ObjectInspectorTest.java

```
import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.tree.*;

public class ObjectInspectorTest
```

```
{   public static void main(String[] args)
   {  JFrame frame = new ObjectInspectorFrame();
       frame.show();
    }
}

class ObjectInspectorFrame extends JFrame
{  public ObjectInspectorFrame()
    {   setTitle("ObjectInspectorTest");
        setSize(300, 200);
        addWindowListener(new WindowAdapter()
           {  public void windowClosing(WindowEvent e)
               {   System.exit(0);
               }
           } );

       // we inspect this frame object

       Variable v = new Variable(getClass(), "this", this);
        ObjectTreeModel model = new ObjectTreeModel();
         model.setRoot(v);

       // construct and show tree

        tree = new JTree(model);
       JScrollPane scrollPane = new JScrollPane(tree);
        Container contentPane = getContentPane();
         contentPane.add(scrollPane, "Center");
    }

   private JTree tree;
}

class ObjectTreeModel implements TreeModel
{  public ObjectTreeModel()
   {  root = null;
```

```
 }

public void setRoot(Variable v)
{   Variable oldRoot = v;
    root = v;
     fireTreeStructureChanged(oldRoot);
 }

public Object getRoot()
{   return root;
 }

public int getChildCount(Object parent)
{   return ((Variable)parent).getFields().size();
 }

public Object getChild(Object parent, int index)
{   ArrayList fields = ((Variable)parent).getFields();
    Field f = (Field)fields.get(index);
    Object parentValue = ((Variable)parent).getValue();
     try
    {   return new Variable(f.getType(), f.getName(),
                f.get(parentValue));
     }
     catch(IllegalAccessException e)
     {   return null;
     }
 }

public int getIndexOfChild(Object parent, Object child)
{   int n = getChildCount(parent);
   for (int i = 0; i < n; i++)
        if (getChild(parent, i).equals(child))
             return i;
     return -1;
 }
```

```
    public boolean isLeaf(Object node)
    {   return getChildCount(node) == 0;
    }

    public void valueForPathChanged(TreePath path, Object newValue)
    {}

    public void addTreeModelListener(TreeModelListener l)
    {    listenerList.add(TreeModelListener.class, l);
    }

    public void removeTreeModelListener(TreeModelListener l)
    {   listenerList.remove(TreeModelListener.class, l);
    }

    protected void fireTreeStructureChanged(Object oldRoot)
    {   TreeModelEvent event
            = new TreeModelEvent(this, new Object[] {oldRoot});
        Object[] listeners = listenerList.getListenerList();
        for (int i = listeners.length - 2; i >= 0; i -= 2)
            ((TreeModelListener)listeners[i+1]).
                    treeStructureChanged(event);
    }

    private Variable root;
    private EventListenerList listenerList
        = new EventListenerList();
}

class Variable
{  public Variable(Class aType, String aName, Object aValue)
    {   type = aType;
        name = aName;
        value = aValue;
        fields = new ArrayList();
```

```
   /* find all fields if we have a class type
        except we don't expand strings and null values
      */

    if (!type.isPrimitive() && !type.isArray() &&
         !type.equals(String.class) && value != null)
     {  // get fields from the class and all superclasses
        for (Class c = value.getClass(); c != null;
              c = c.getSuperclass())
         {   Field[] f = c.getDeclaredFields();
              AccessibleObject.setAccessible(f, true);

              // get all nonstatic fields
           for (int i = 0; i < f.length; i++)
              if ((f[i].getModifiers() & Modifier.STATIC) == 0)
                   fields.add(f[i]);
          }
     }
 }

public Object getValue()
 {   return value;
 }

 public ArrayList getFields()
 {   return fields;
 }

public String toString()
{  String r = type + " " + name;
     if (type.isPrimitive())
        r += "=" + value;
     else if (type.equals(String.class))
        r += "=" + value;
      else if (value == null)
```

```
            r += "=null";
        return r;
    }

    private Class type;
    private String name;
    private Object value;
    private ArrayList fields;
}
```

javax.swing.tree.TreeModel

- Object getRoot()
  루트 노드를 반환한다.
- int getChildCount(Object parent)
  parent **노드의 지식 수를 얻는다**.
- Object getChild(Object parent, int index)
  주어진 인덱스 위치의 parent **노드의 자식 노드를 얻는다**.
- int getIndexOfChild(Object parent, Object child)
  child **노드의 인덱스를 얻는다**. parent **노드의 자식이어야 한다**.
- boolean isLeaf(Object node)
  만일 node**가 개념적으로 트리의 잎이라면** true**를 반환한다**.
- void addTreeModelListener(TreeModelListener l)
- void removeTreeModelListener(TreeModelListener l)
  트리 모델내의 정보가 변했을 때 알려질 리스너를 추가하거나 제거한다.
- void valueForPathChanged(TreePath path, Object newValue)
  셀 편집기가 노드의 값을 변경했을 때 호출된다.

  전달인자: path **편집될 노드에 대한 경로**
  object **편집기에 의해서 반환되는 값을 대체한다**.

javax.swing.event.TreeModelListener

- void treeNodesChanged(TreeModelEvent e)
- void treeNodesInserted(TreeModelEvent e)
- void treeNodesRemoved(TreeModelEvent e)
- void treeStructureChanged(TreeModelEvent e)

트리가 변경됐을 때 트리 모델에 의해서 호출된다.

javax.swing.event.TreeModelLEvent

- TreeModelEvent(Object eventSource, TreePath node)

  트리 모델 이벤트를 구성한다.

  전달인자: eventSource 해당 이벤트를 생성해내는 트리 모델

  node 변경될 노드에 대한 경로

javax.swing.EventListenerList

- void add(Class t, EventListener l)

  리스트에 해당 클래스와 이벤트 리스너를 추가한다. 클래스는 메소드를 구동하는 이벤트가 선택적으로 이벤트를 호출할 수 있도록 저장된다. 기본적인 사용법은 addXxxListener 메소드:

```
public void addXxxListener(XxxListener l)
{       listenerList.add(Xxx.class, l);
}
```

  전달인자: c 리스너 타입

  l 리스너

- void remove(Class t, EventListener l)

  리스트로부터 해당 클래스와 이벤트 리스너를 제거한다. 기본적인 사용법은 addXxxListener 메소드:

```
public void removeXxxListener(XxxListener l)
{       listenerList.remove(XxxListener.class, l);
}
```

  전달인자: c 리스너 타입

  l 리스너

- Object[] getListenerList()

  짝수 인덱스의 리스너 클래스 항목들을 갖는 배열과 홀수 인덱스의 리스너 클래스 항목들을 갖는 배열을 반환한다. 배열은 널이 아니라는 사실을 보증한다. 기본 사용법은 이벤트를 구동하는 메소드에 있다.

```
protected void fireEventOccured(EventParameters p)
{       XxxEvent event = new XxxEvent(this , p);
        Object[] listeners = listenerList.getListenerList();
        for(int i = listeners.length –2; i >- 0; i -= 2)
```

```
if(listeners[i] instanceof XxxListener.class)
 ((XxxListener)listeners[i+1]).
            EventOccured(event);
```