# 1.JTabbedPane

JTabbledPane在窗口上创建多个标签页，每个标签页可以控制一个与外部窗口相同大小的容器。

**package** test2;

**import** java.awt.\*;

**import** java.awt.event.\*;

**import** java.util.\*;

**import** javax.swing.\*;

**import** javax.swing.event.\*;

**public** **class** TestJTabbedPane {

JPanel jp1 = **new** JPanel();

JPanel jp2 = **new** JPanel();

JPanel jp3 = **new** JPanel();

JPanel jp4 = **new** JPanel();

JPanel jp5 = **new** JPanel();

JFrame jf = **new** JFrame("测试Tab页面");

// 创建一个Tab页面的标签放在左边，采用换行布局策略的JTabbedPane

JTabbedPane tabbedPane = **new** JTabbedPane(JTabbedPane.*LEFT*,JTabbedPane.*WRAP\_TAB\_LAYOUT*);

String[] layouts = { "换行布局", "滚动条布局" };

String[] positions = { "左边", "顶部", "右边", "底部" };

**public** **void** init() {

String tip = "测试提示";

// 向JTabbedPane中添加5个Tab页面，指定了标题、图标和提示，但该Tab页面的组件为null

tabbedPane.addTab("jp1", **null**, **null**, tip);

tabbedPane.addTab("jp2", **null**, **null**, tip);

tabbedPane.addTab("jp3", **null**, **null**, tip);

tabbedPane.addTab("jp4", **null**, **null**, tip);

tabbedPane.addTab("jp5", **null**, **null**, tip);

jf.add(tabbedPane, BorderLayout.*CENTER*);

// 为JTabbedPane添加事件监听器

tabbedPane.addChangeListener(**new** ChangeListener() {

**public** **void** stateChanged(ChangeEvent event) {

// 如果被选择的组件依然是空

**if** (tabbedPane.getSelectedComponent() == **null**) {

// 获取所选Tab页

**int** n = tabbedPane.getSelectedIndex();

// 为指定标前页加载内容

loadTab(n);

}

}

});

// 系统默认选择第一页，加载第一页内容

loadTab(0);

tabbedPane.setPreferredSize(**new** Dimension(500, 300));

// 增加控制标签布局、标签位置的单选按钮

JPanel buttonPanel = **new** JPanel();

ChangeAction action = **new** ChangeAction();

buttonPanel.add(**new** ButtonPanel(action, "选择标签布局策略", layouts));

buttonPanel.add(**new** ButtonPanel(action, "选择标签位置", positions));

jf.add(buttonPanel, BorderLayout.*SOUTH*);

jf.setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*);

jf.pack();

jf.setVisible(**true**);

}

// 为指定标签页加载内容

**private** **void** loadTab(**int** n) {

String title = tabbedPane.getTitleAt(n);

tabbedPane.setComponentAt(n, **new** JLabel("我是第个" + n + "面板"));

// 根据标签页的标题获取对应图书封面

}

// 定义改变标签页的布局策略，放置位置的监听器

**class** ChangeAction **implements** ActionListener {

**public** **void** actionPerformed(ActionEvent event) {

JRadioButton source = (JRadioButton) event.getSource();

String selection = source.getActionCommand();

**if** (selection.equals(layouts[0])) {

tabbedPane.setTabLayoutPolicy(JTabbedPane.*WRAP\_TAB\_LAYOUT*);

} **else** **if** (selection.equals(layouts[1])) {

tabbedPane.setTabLayoutPolicy(JTabbedPane.*SCROLL\_TAB\_LAYOUT*);

} **else** **if** (selection.equals(positions[0])) {

tabbedPane.setTabPlacement(JTabbedPane.*LEFT*);

} **else** **if** (selection.equals(positions[1])) {

tabbedPane.setTabPlacement(JTabbedPane.*TOP*);

} **else** **if** (selection.equals(positions[2])) {

tabbedPane.setTabPlacement(JTabbedPane.*RIGHT*);

} **else** **if** (selection.equals(positions[3])) {

tabbedPane.setTabPlacement(JTabbedPane.*BOTTOM*);

}

}

}

**public** **static** **void** main(String[] args) {

**new** TestJTabbedPane().init();

}

}

// 定义一个JPanel类扩展类，该类的对象包含多个纵向排列的JRadioButton控件

// 且Panel扩展类可以指定一个字符串作为TitledBorder

**class** ButtonPanel **extends** JPanel {

**private** ButtonGroup group;

**public** ButtonPanel(TestJTabbedPane.ChangeAction action, String title,

String[] labels) {

setBorder(BorderFactory.*createTitledBorder*(

BorderFactory.*createEtchedBorder*(), title));

setLayout(**new** BoxLayout(**this**, BoxLayout.*X\_AXIS*));

group = **new** ButtonGroup();

**for** (**int** i = 0; labels != **null** && i < labels.length; i++) {

JRadioButton b = **new** JRadioButton(labels[i]);

b.setActionCommand(labels[i]);

add(b);

// 添加事件监听器

b.addActionListener(action);

group.add(b);

b.setSelected(i == 0);

}

}

}



# 2. JDesktopPane和JInternalFrame

JdesktopOane和JinternalFrame需要结合使用，JdesktopPane代表一个虚拟桌面，JinternalFrame用于创建内部窗口。

使用步骤：

1. 创建一个JdesktopPane对象，直接new即可
2. 使用JinternalFrame创建一个内部窗口，创建内部窗口与创建Jframe窗口有一些区别，创建JinternalFrame对象的时候，可以传入一个字符串作为内部窗口的标题，还可以传入4个boolean值，来确定内部窗口是否允许改变窗口大小，关闭窗口，最小化窗口和最大化窗口
3. 获得了内部窗口以后，这个窗口的用法和普通窗口的用法基本的相似，一样可以指定窗口的布局管理器，向窗口内添加组件。
4. 将内部窗口以合适的大小放在合适的位置显示出来。注意坐标是位于虚拟桌面的
5. 将内部窗口添加到JdesktopPane容器，再将JdesktopPane添加到其他容器

**package** Test;

**import** java.awt.BorderLayout;

**import** java.awt.Dimension;

**import** java.awt.event.ActionEvent;

**import** javax.swing.AbstractAction;

**import** javax.swing.Action;

**import** javax.swing.ImageIcon;

**import** javax.swing.JDesktopPane;

**import** javax.swing.JFrame;

**import** javax.swing.JInternalFrame;

**import** javax.swing.JMenu;

**import** javax.swing.JMenuBar;

**import** javax.swing.JScrollPane;

**import** javax.swing.JTextArea;

**import** javax.swing.JToolBar;

**import** javax.swing.UIManager;

**import** javax.swing.UnsupportedLookAndFeelException;

**public** **class** TestInternalFrame {

**final** **int** DESKTOP\_WIDTH = 480;

**final** **int** DESKTOP\_HEIGHT = 360;

**final** **int** FRAME\_DISTANCE = 30;

JFrame jf = **new** JFrame("父窗口");

// 定义一个虚拟桌面

**private** JDesktopPane desktop = **new** JDesktopPane();

// 保存下一个内部窗口的座标点

**private** **int** nextFrameX;

**private** **int** nextFrameY;

// 定义内部窗口为虚拟桌面的1/2大小

**private** **int** width = DESKTOP\_WIDTH / 2;

**private** **int** height = DESKTOP\_HEIGHT / 2;

// 为主窗口定义2个菜单

JMenu fileMenu = **new** JMenu("文件");

// 定义newAction用于创建菜单和工具按钮

Action newAction = **new** AbstractAction("新建", **new** ImageIcon("ico/new.png")) {

**public** **void** actionPerformed(ActionEvent event) {

// 创建内部窗口

**final** JInternalFrame iframe = **new** JInternalFrame("新文档", **true**, // 可改变大小

**true**, // 可关闭

**true**, // 可最大化

**true**); // 可最小化

iframe.add(**new** JScrollPane(**new** JTextArea(8, 40)));

// 将内部窗口添加到虚拟桌面中

desktop.add(iframe);

// 设置内部窗口的原始位置（内部窗口默认大小是0X0，放在0,0位置）

iframe.reshape(nextFrameX, nextFrameY, width, height);

// 使该窗口可见，并尝试选中它

iframe.show();

// 计算下一个内部窗口的位置

nextFrameX += FRAME\_DISTANCE;

nextFrameY += FRAME\_DISTANCE;

**if** (nextFrameX + width > desktop.getWidth())

nextFrameX = 0;

**if** (nextFrameY + height > desktop.getHeight())

nextFrameY = 0;

}

};

// 定义exitAction用于创建菜单和工具按钮

Action exitAction = **new** AbstractAction("退出", **new** ImageIcon("ico/exit.png")) {

**public** **void** actionPerformed(ActionEvent event) {

System.*exit*(0);

}

};

**public** **void** init() {

// 为窗口安装菜单条和工具条

JMenuBar menuBar = **new** JMenuBar();

JToolBar toolBar = **new** JToolBar();

jf.setJMenuBar(menuBar);

menuBar.add(fileMenu);

fileMenu.add(newAction);

fileMenu.add(exitAction);

toolBar.add(newAction);

toolBar.add(exitAction);

desktop.setPreferredSize(**new** Dimension(480, 360));

// 将虚拟桌面添加到顶级JFrame容器中

jf.add(desktop);

jf.add(toolBar, BorderLayout.*NORTH*);

**try** {

UIManager.*setLookAndFeel*("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");

} **catch** (ClassNotFoundException | InstantiationException

| IllegalAccessException | UnsupportedLookAndFeelException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

jf.setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*);

jf.pack();

jf.setVisible(**true**);

}

**public** **static** **void** main(String[] args) {

**new** TestInternalFrame().init();

}

}

# 3.JTree编辑树

接下来看一下可以拖动并且能够编辑的树，默认JTree生成的树是不能编辑的。

使用setEditable方法就可以。

对树的编辑 一般有

添加兄弟节点、子节点、删除节点、编辑节点、节点拖动这些功能。

那么可以试想一下：

要想操作一个节点，那么，必须知道当前选中了哪个节点。如何知道当前选中了哪个节点是我们关心的。

TreePath这个类。

记录了从根节点到指定节点的所有节点，TreePath由一系列节点组成。

**如果要获取JTree中被选定的节点，使用如下代码：**

**TreePath path = tree.getSelectionPath();**

**TreeNode target = (TreeNode)path.getLastPathComponent();**

**还可以使用**

**TreeNode target = (TreeNode)tree.getLastSelectedPathComponent();**

只要获取选中的节点，就可以通过DefaultTreeModel提供的方法来编辑节点。

另外，也可以直接通过TreeNode提供 方法来添加、删除和修改节点，但是通过这种方式作出的修改，必须手工调用JTree的updateUI通知JTree重绘所有节点。

**还有一个问题，如果是拖动节点 如何知道拖到哪里去？**

**这势必需要得到鼠标移动的x和y。**

**TreePath path = tree.getPathForLocation(e.getX(),e.getY());**

**package** Test3;

**import** javax.swing.\*;

**import** javax.swing.tree.\*;

**import** java.awt.event.\*;

**import** java.awt.\*;

**public** **class** EditJTree {

JFrame jf;

JTree tree;

// 上面JTree对象对应的model

DefaultTreeModel model;

// 定义几个初始节点

DefaultMutableTreeNode root = **new** DefaultMutableTreeNode("中国");

DefaultMutableTreeNode shandong = **new** DefaultMutableTreeNode("山东");

DefaultMutableTreeNode jiangsu = **new** DefaultMutableTreeNode("江苏");

DefaultMutableTreeNode jinan = **new** DefaultMutableTreeNode("济南");

DefaultMutableTreeNode qingdao = **new** DefaultMutableTreeNode("青岛");

DefaultMutableTreeNode nanjing = **new** DefaultMutableTreeNode("南京");

DefaultMutableTreeNode suzhou = **new** DefaultMutableTreeNode("苏州");

// 定义需要被拖动的TreePath

TreePath movePath;

JButton addSiblingButton = **new** JButton("添加兄弟节点");

JButton addChildButton = **new** JButton("添加子节点");

JButton deleteButton = **new** JButton("删除节点");

JButton editButton = **new** JButton("编辑当前节点");

**public** **void** init() {

shandong.add(jinan);

shandong.add(qingdao);

jiangsu.add(nanjing);

jiangsu.add(suzhou);

root.add(shandong);

root.add(jiangsu);

jf = **new** JFrame("树");

tree = **new** JTree(root);

// 获取JTree对应的TreeModel对象

model = (DefaultTreeModel) tree.getModel();

// 设置JTree可编辑

tree.setEditable(**true**);

MouseListener ml = **new** MouseAdapter() {

// 按下鼠标时候获得被拖动的节点

**public** **void** mousePressed(MouseEvent e) {

// 如果需要唯一确定某个节点，必须通过TreePath来获取。

TreePath tp = tree.getPathForLocation(e.getX(), e.getY());

**if** (tp != **null**) {

movePath = tp;

}

}

// 鼠标松开时获得需要拖到哪个父节点

**public** **void** mouseReleased(MouseEvent e) {

// 根据鼠标松开时的TreePath来获取TreePath

TreePath tp = tree.getPathForLocation(e.getX(), e.getY());

**if** (tp != **null** && movePath != **null**) {

// 阻止向子节点拖动

**if** (movePath.isDescendant(tp) && movePath != tp) {

JOptionPane.*showMessageDialog*(jf,

"目标节点是被移动节点的子节点，无法移动！", "非法操作",

JOptionPane.*ERROR\_MESSAGE*);

**return**;

}

// 既不是向子节点移动，而且鼠标按下、松开的不是同一个节点

**else** **if** (movePath != tp) {

System.*out*.println(tp.getLastPathComponent());

// add方法可以先将原节点从原父节点删除，再添加到新父节点中

((DefaultMutableTreeNode) tp.getLastPathComponent())

.add((DefaultMutableTreeNode) movePath

.getLastPathComponent());

movePath = **null**;

tree.updateUI();

}

}

}

};

tree.addMouseListener(ml);

JPanel panel = **new** JPanel();

addSiblingButton.addActionListener(**new** ActionListener() {

**public** **void** actionPerformed(ActionEvent event) {

// 获取选中节点

DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode) tree

.getLastSelectedPathComponent();

// 如果节点为空，直接返回

**if** (selectedNode == **null**)

**return**;

// 获取该选中节点的父节点

DefaultMutableTreeNode parent = (DefaultMutableTreeNode) selectedNode

.getParent();

// 如果父节点为空，直接返回

**if** (parent == **null**)

**return**;

// 创建一个新节点

DefaultMutableTreeNode newNode = **new** DefaultMutableTreeNode(

"新节点");

// 获取选中节点的选中索引

**int** selectedIndex = parent.getIndex(selectedNode);

// 在选中位置插入新节点

model.insertNodeInto(newNode, parent, selectedIndex + 1);

// --------下面代码实现显示新节点（自动展开父节点）-------

// 获取从根节点到新节点的所有节点

TreeNode[] nodes = model.getPathToRoot(newNode);

// 使用指定的节点数组来创建TreePath

TreePath path = **new** TreePath(nodes);

// 显示指定TreePath

tree.scrollPathToVisible(path);

}

});

panel.add(addSiblingButton);

addChildButton.addActionListener(**new** ActionListener() {

**public** **void** actionPerformed(ActionEvent event) {

// 获取选中节点

DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode) tree

.getLastSelectedPathComponent();

// 如果节点为空，直接返回

**if** (selectedNode == **null**)

**return**;

// 创建一个新节点

DefaultMutableTreeNode newNode = **new** DefaultMutableTreeNode(

"新节点");

// 直接通过model来添加新节点，则无需通过调用JTree的updateUI方法

// model.insertNodeInto(newNode, selectedNode,

// selectedNode.getChildCount());

// 直接通过节点添加新节点，则需要调用tree的updateUI方法

selectedNode.add(newNode);

// --------下面代码实现显示新节点（自动展开父节点）-------

TreeNode[] nodes = model.getPathToRoot(newNode);

TreePath path = **new** TreePath(nodes);

tree.scrollPathToVisible(path);

tree.updateUI();

}

});

panel.add(addChildButton);

deleteButton.addActionListener(**new** ActionListener() {

**public** **void** actionPerformed(ActionEvent event) {

DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode) tree

.getLastSelectedPathComponent();

**if** (selectedNode != **null** && selectedNode.getParent() != **null**) {

// 删除指定节点

model.removeNodeFromParent(selectedNode);

}

}

});

panel.add(deleteButton);

editButton.addActionListener(**new** ActionListener() {

**public** **void** actionPerformed(ActionEvent event) {

TreePath selectedPath = tree.getSelectionPath();

**if** (selectedPath != **null**) {

// 编辑选中节点

tree.startEditingAtPath(selectedPath);

}

}

});

panel.add(editButton);

jf.add(**new** JScrollPane(tree));

jf.add(panel, BorderLayout.*SOUTH*);

jf.pack();

jf.setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*);

jf.setVisible(**true**);

}

**public** **static** **void** main(String[] args) {

**new** EditJTree().init();

}

}

# 4JTable表格排序

**package** Test3;

**import** java.util.\*;

**import** java.awt.event.\*;

**import** javax.swing.\*;

**import** javax.swing.table.\*;

**public** **class** SortTable {

JFrame jf = **new** JFrame("简单表格");

// 定义二维数组作为表格数据

Object[][] tableData = { **new** Object[] { "胡伟红", 29, "女" },

**new** Object[] { "盖茨", 56, "男" }, **new** Object[] { "Scott", 35, "男" },

**new** Object[] { "王菲", 38, "女" }, **new** Object[] { "Baby", 2, "男" } };

// 定义一维数据作为列标题

Object[] columnTitle = { "姓名", "年龄", "性别" };

// 以二维数组和一维数组来创建一个JTable对象

JTable table = **new** JTable(tableData, columnTitle);

// 将原表格里的model包装成新的SortFilterModel对象

SortableTableModel sorterModel = **new** SortableTableModel(table.getModel());

**public** **void** init() {

table.setModel(sorterModel);

table.getTableHeader().addMouseListener(**new** MouseAdapter() {

**public** **void** mouseClicked(MouseEvent event) {

// 如果单击次数小于2，即不是双击，直接返回

**if** (event.getClickCount() < 2) {

**return**;

}

// 找出鼠标双击事件所在的列索引

**int** tableColumn = table.columnAtPoint(event.getPoint());

// 将JTable中的列索引转换成对应TableModel中的列索引

**int** modelColumn = table.convertColumnIndexToModel(tableColumn);

// 根据指定列进行排序

sorterModel.sort(modelColumn);

}

});

// 将JTable对象放在JScrollPane中，并将该JScrollPane放在窗口中显示出来

jf.add(**new** JScrollPane(table));

jf.pack();

jf.setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*);

jf.setVisible(**true**);

}

**public** **static** **void** main(String[] args) {

**new** SortTable().init();

}

}

**class** SortableTableModel **extends** AbstractTableModel {

**private** TableModel model;

**private** **int** sortColumn;

**private** Row[] rows;

// 将一个已经存在TableModel对象包装成SortableTableModel对象

**public** SortableTableModel(TableModel m) {

// 将被封装的TableModel传入

model = m;

rows = **new** Row[model.getRowCount()];

// 将原TableModel中的每行记录的索引使用Row数组保存起来

**for** (**int** i = 0; i < rows.length; i++) {

rows[i] = **new** Row(i);

}

}

// 实现根据指定列进行排序

**public** **void** sort(**int** c) {

sortColumn = c;

Arrays.*sort*(rows);

fireTableDataChanged();

}

// 下面三个方法需要访问Model中的数据，所以涉及到本Model中数据

// 和被包装Model数据中的索引转换，程序使用rows数组完成这种转换。

**public** Object getValueAt(**int** r, **int** c) {

**return** model.getValueAt(rows[r].index, c);

}

**public** **boolean** isCellEditable(**int** r, **int** c) {

**return** model.isCellEditable(rows[r].index, c);

}

**public** **void** setValueAt(Object aValue, **int** r, **int** c) {

model.setValueAt(aValue, rows[r].index, c);

}

// 下面方法的实现把该model的方法委托为原封装的model来实现

**public** **int** getRowCount() {

**return** model.getRowCount();

}

**public** **int** getColumnCount() {

**return** model.getColumnCount();

}

**public** String getColumnName(**int** c) {

**return** model.getColumnName(c);

}

**public** Class getColumnClass(**int** c) {

**return** model.getColumnClass(c);

}

// 定义一个Row类，该类用于封装JTable中的一行、

// 实际上它并不封装行数据，它只封装行索引

**private** **class** Row **implements** Comparable<Row> {

// 该index保存着被封装Model里每行记录的行索引

**public** **int** index;

**public** Row(**int** index) {

**this**.index = index;

}

// 实现两行之间的大小比较

**public** **int** compareTo(Row other) {

Object a = model.getValueAt(index, sortColumn);

Object b = model.getValueAt(other.index, sortColumn);

**if** (a **instanceof** Comparable) {

**return** ((Comparable) a).compareTo(b);

} **else** {

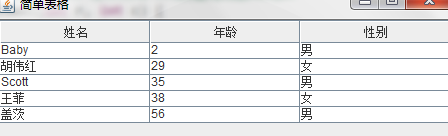
**return** a.toString().compareTo(b.toString());

}

}

}

}



# 5JTable编辑单元格内容

表格中的内容都是字符串，通过处理也可以是其他复杂组件。

使用Renderer来处理

JTable使用TableCellRenderer绘制单元格，这是一个接口，Swing提供了一个实现类，DefaultTableCellRenderer，可以绘制三种类型：

Icon

Boolean 复选按钮

Object 绘制Object的toString方法产生的字符串

如果直接使用二维数组或Vector创建JTable，程序会使用JTable的匿名内部类或DefaultTableModel充当该表格的model对象，这两个TableModel的getColumnClass返回值都是Object

为了能够让单元格绘制器将Icon绘制成图标，把Boolean绘制成复选框，创建JTable不能使用默认的TableModel，必须扩展TableModel类。使用扩展类创建JTable对象。

另外使用 扩展了DefaultTableModel这个类的对象创建JTable，可以防止错误输入。

比如在数字型表格输入字符型。

上面的例子，在双击单元格内容不是文本的情况，会变成文本来变更，这显然不够友好。

TreecellEditor接口可以帮助我们进行友好的现实。

默认实现类是DefaultCellEditor。有三个构造方法，分别使用文本框、复选框和JComboBox作为单元格编辑器

**import** java.awt.\*;

**import** java.awt.event.\*;

**import** javax.swing.\*;

**import** javax.swing.table.\*;

**import** java.io.\*;

**public** **class** TestTableCellEditor {

JFrame jf = **new** JFrame("使用单元格编辑器");

JTable table;

// 定义二维数组作为表格数据

Object[][] tableData = {

**new** Object[] { "胡伟红", 29, "女", **new** ImageIcon("icon/3.gif"),

**new** ImageIcon("icon/3.gif"), **true** },

**new** Object[] { "骑车上树", 56, "男", **new** ImageIcon("icon/1.gif"),

**new** ImageIcon("icon/1.gif"), **false** },

**new** Object[] { "风中叶", 35, "男", **new** ImageIcon("icon/4.gif"),

**new** ImageIcon("icon/4.gif"), **true** },

**new** Object[] { "有风起浪", 18, "女", **new** ImageIcon("icon/2.gif"),

**new** ImageIcon("icon/2.gif"), **true** },

**new** Object[] { "云飞日月", 2, "男", **new** ImageIcon("icon/5.gif"),

**new** ImageIcon("icon/5.gif"), **false** } };

// 定义一维数据作为列标题

String[] columnTitle = { "姓名", "年龄", "性别", "主头像", "次头像", "是否已婚" };

**public** **void** init() {

// 以二维数组和一维数组来创建一个ExtendedTableModel对象

ExtendedTableModel3 model = **new** ExtendedTableModel3(columnTitle,

tableData);

// 以ExtendedTableModel来创建JTable

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

table.setRowSelectionAllowed(**false**);

table.setRowHeight(40);

// 为该表格指定默认的编辑器

table.setDefaultEditor(ImageIcon.**class**, **new** ImageCellEditor());

// 获取最后一列

TableColumn lastColumn = table.getColumnModel().getColumn(4);

JComboBox editCombo = **new** JComboBox();

**for** (**int** i = 1; i <= 10; i++) {

editCombo.addItem(**new** ImageIcon("icon/" + i + ".gif"));

}

lastColumn.setCellEditor(**new** DefaultCellEditor(editCombo));

// 将JTable对象放在JScrollPane中，并将该JScrollPane放在窗口中显示出来

jf.add(**new** JScrollPane(table));

jf.pack();

jf.setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*);

jf.setVisible(**true**);

}

**public** **static** **void** main(String[] args) {

**new** TestTableCellEditor().init();

}

}

**class** ExtendedTableModel3 **extends** DefaultTableModel {

// 重新提供一个构造器，该构造器的实现委托给DefaultTableModel父类

**public** ExtendedTableModel3(String[] columnNames, Object[][] cells) {

**super**(cells, columnNames);

}

// 重写getColumnClass方法，根据每列的第一个值来返回其真实的数据类型

**public** Class getColumnClass(**int** c) {

**return** getValueAt(0, c).getClass();

}

}

// 扩展自己的TableCellEditor类

**class** ImageCellEditor **extends** DefaultCellEditor {

// 定义文件选择器

**private** JFileChooser fDialog = **new** JFileChooser();;

**private** JTextField field = **new** JTextField(15);

**private** JButton button = **new** JButton("...");

**public** ImageCellEditor() {

// 因为DefaultCellEditor没有无参数的构造器

// 所以这里显式调用父类有参数的构造器。

**super**(**new** JTextField());

initEditor();

}

**private** **void** initEditor() {

field.setEditable(**false**);

// 为按钮添加监听器，当用户单击该按钮时，

// 系统将出现一个文件选择器让用户选择图标文件。

button.addActionListener(**new** ActionListener() {

**public** **void** actionPerformed(ActionEvent e) {

browse();

}

});

// 为文件选择器安装文件过滤器

fDialog.addChoosableFileFilter(**new** javax.swing.filechooser.FileFilter() {

**public** **boolean** accept(File f) {

**if** (f.isDirectory()) {

**return** **true**;

}

String extension = Utils.*getExtension*(f);

**if** (extension != **null**) {

**if** (extension.equals(Utils.*tiff*)

|| extension.equals(Utils.*tif*)

|| extension.equals(Utils.*gif*)

|| extension.equals(Utils.*jpeg*)

|| extension.equals(Utils.*jpg*)

|| extension.equals(Utils.*png*)) {

**return** **true**;

} **else** {

**return** **false**;

}

}

**return** **false**;

}

**public** String getDescription() {

**return** "有效的图片文件";

}

});

fDialog.setAcceptAllFileFilterUsed(**false**);

}

// 重写TableCellEditor接口的getTableCellEditorComponent方法

// 该方法返回单元格编辑器，该编辑器是一个JPanel，该容器包含一个文本框和一个按钮

**public** Component getTableCellEditorComponent(JTable table, Object value,

**boolean** isSelected, **int** row, **int** column) {

**this**.button.setPreferredSize(**new** Dimension(20, 20));

JPanel panel = **new** JPanel();

panel.setLayout(**new** BorderLayout());

field.setText(value.toString());

panel.add(**this**.field, BorderLayout.*CENTER*);

panel.add(**this**.button, BorderLayout.*EAST*);

**return** panel;

}

**public** Object getCellEditorValue() {

**return** **new** ImageIcon(field.getText());

}

**private** **void** browse() {

// 设置、打开文件选择器

fDialog.setCurrentDirectory(**new** File("icon"));

**int** result = fDialog.showOpenDialog(**null**);

// 如果单击了文件选择器的“取消”按钮

**if** (result == JFileChooser.*CANCEL\_OPTION*) {

// 取消编辑

**super**.cancelCellEditing();

**return**;

}

// 如果单击了文件选择器的“确定”按钮

**else** {

// 设置field的内容

field.setText("icon/" + fDialog.getSelectedFile().getName());

}

}

}

**class** Utils {

**public** **final** **static** String *jpeg* = "jpeg";

**public** **final** **static** String *jpg* = "jpg";

**public** **final** **static** String *gif* = "gif";

**public** **final** **static** String *tiff* = "tiff";

**public** **final** **static** String *tif* = "tif";

**public** **final** **static** String *png* = "png";

// 获取文件扩展名的方法

**public** **static** String getExtension(File f) {

String ext = **null**;

String s = f.getName();

**int** i = s.lastIndexOf('.');

**if** (i > 0 && i < s.length() - 1) {

ext = s.substring(i + 1).toLowerCase();

}

**return** ext;

}

}

