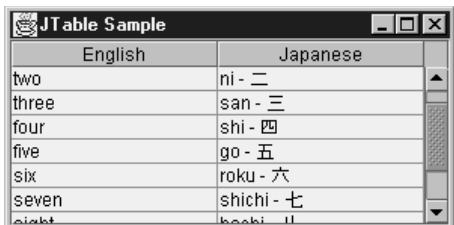


CHAPTER 17

Tables

CHAPTER 16 TOOK AN IN-DEPTH LOOK AT the Swing JTree component. In this chapter, we'll examine the many details of the JTable component. The component is the standard Swing component for displaying two-dimensional data in the form of a grid, as shown in Figure 17-1.

NOTE *All the examples in this chapter will work fine without configuring your environment to display Japanese fonts. However, instead of seeing the ideographs, you will see characters such as question marks or boxes, depending upon your platform. In order to see the Kanji ideographs in the sample programs, you will need to be using the Japanese version of the font.properties file (font.properties.ja) **and** have the necessary Japanese fonts installed. In addition to the Windows NT installation CD, you can find the necessary Windows fonts at <http://ftp.monash.edu.au/pub/nihongo/ie3lpkja.exe>. Solaris users must contact Sun to request the Asian outline fonts for Solaris environments. More on adding fonts to the Java runtime environment is at <http://java.sun.com/products/jdk/1.3/docs/guide/intl/fontprop.html>.*



English	Japanese
two	ni - 二
three	san - 三
four	shi - 四
five	go - 五
six	roku - 六
seven	shichi - 七
eight	hachi - 八

Figure 17-1: Sample JTable

Introducing Tables

Like the JTree component, the JTable component relies on numerous support classes for its inner workings. For the JTable, the support classes are found in the `javax.swing.table` package. The cells within the JTable can be selected by row, column, row and column, or individual cell. It's the responsibility of the current `ListSelectionModel` settings to control the selection within a table.

The display of the different cells within a table is the responsibility of the `TableCellRenderer`; the `DefaultTableCellRenderer` offers one such implementation of the `TableCellRenderer` interface in a `JLabel` subclass.

Managing the data stored in the cells is accomplished through an implementation of the `TableModel` interface. The `AbstractTableModel` provides the basics of an implementation of the interface without any data storage. By comparison, the `DefaultTableModel` encapsulates the `TableModel` interface and uses a vector of vectors for the data storage. You extend `AbstractTableModel` if you need a different type of storage than the kind supplied by the `DefaultTableModel`, for instance, if you already had the data in your own data structure.

The `TableColumnModel` interface and the `DefaultTableColumnModel` implementation of the interface manage the table's data as a series of columns. They work together with the `TableColumn` class to allow for greater flexibility in manipulating individual columns. For example, you can store columns of data in the `TableModel` in an order that's different than the display order within the `JTable`. The `TableColumnModel` manages a second `ListSelectionModel` to control table column selection.

At the top of every column is a column header. By default, the `TableColumn` class relies on the `JTableHeader` class to render a text column header. Nevertheless, you must embed the `JTable` in a scroll pane to see the default header.

Cells within a `JTable` can be editable. If a cell is editable, how the editing works depends on the `TableCellEditor` implementation, such as the `DefaultCellEditor` implementation, which extends from `AbstractCellEditor`. In addition, no classes exist to handle individual rows. Rows must be manipulated on a cell-by-cell basis. However, behind the scenes, the `JTable` does use the `SizeSequence` utility class to deal with variable height rows. However, you won't need to manipulate it yourself.

Figure 17-2 shows these class interrelationships.

NOTE *There are additional interrelationships among the elements used by the `JTable` component. These relationships will be explored later in this chapter with each specific interface and class.*

To visualize how the `JTable` elements all fit together, examine Figure 17-3.

Class `JTable`

We'll first look at the `JTable` class, which gives you a way to display data in tabular form.

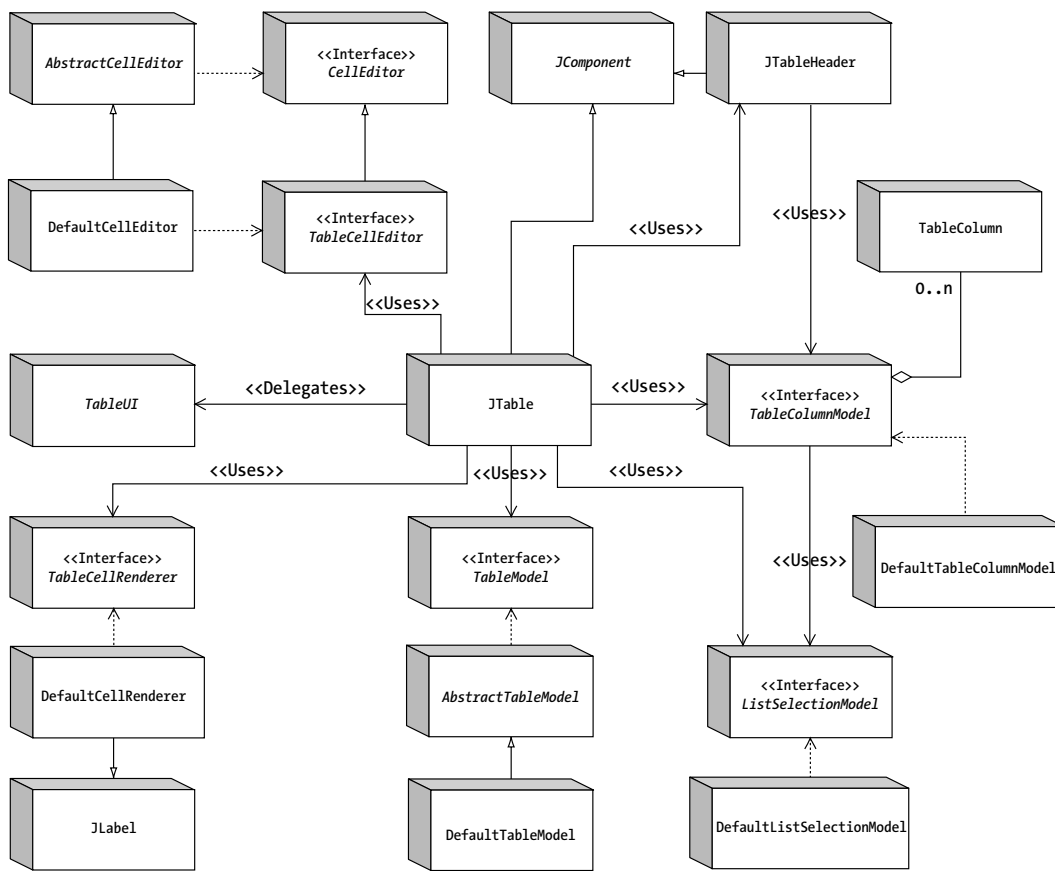


Figure 17-2: JTable UML relationship diagram

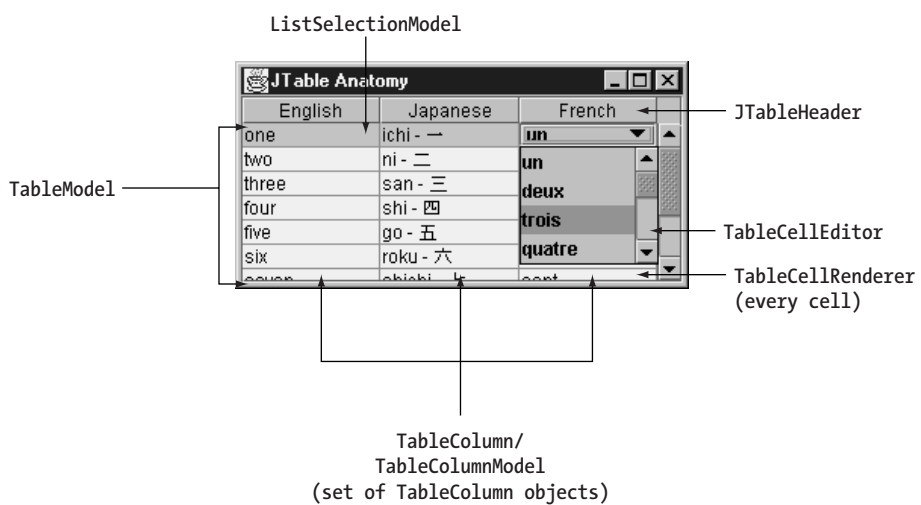


Figure 17-3: JTable elements

Creating a JTable

You have seven different ways at your disposal to create a JTable. The various constructors allow you to create tables from a number of data sources.

In the following list, the *no-arg* constructor creates a table with no rows and no columns. The second constructor takes two integers to create an empty table with a set number of rows and columns.

The next two constructors are useful when your tabular data is already in a specially structured form. For instance, if your data is already in the form of an array of arrays or a Vector of Vector objects, you can create a JTable without creating your own TableModel. A two-row-by-three-column table could be created from the array of `{{"Row1-Column1", "Row1-Column2", "Row1-Column3"}, {"Row2-Column1", "Row2-Column2", "Row2-Column3"}}`, with another array holding the column names. Similar data structures would be necessary for the vector-based constructor.

The remaining three constructors use JTable-specific data structures. If any one of the three arguments is missing, default settings will be used. For example, if you don't specify a TableColumnModel, the default implementation DefaultTableColumnModel is used and is auto-filled with a display order using the column order of the TableModel. When the selection model is missing, the ListSelectionModel will use multi-selection mode, which means that noncontiguous rows, but not columns, can be selected.

1.

```
public JTable()
    JTable table = new JTable();
```
2.

```
public JTable(int rows, int columns)
    JTable table = new JTable(2, 3);
```

NOTE *Table components created from JTable constructors are editable, and not read-only. To change their contents in code, just call the public void setValueAt(Object value, int row, int column) method of JTable.*

3.

```
public JTable(Object rowData[][], Object columnNames[])
    Object rowData[][] = {{"Row1-Column1", "Row1-Column2", "Row1-Column3"},
        {"Row2-Column1", "Row2-Column2", "Row2-Column3"}};
    Object columnNames[] = {"Column One", "Column Two", "Column Three"};
    JTable table = new JTable(rowData, columnNames);
```

4.

```
public JTable(Vector rowData, Vector columnNames)
    Vector rowOne = new Vector();
    rowOne.addElement("Row1-Column1");
    rowOne.addElement("Row1-Column2");
    rowOne.addElement("Row1-Column3");
    Vector rowTwo = new Vector();
    rowTwo.addElement("Row2-Column1");
    rowTwo.addElement("Row2-Column2");
    rowTwo.addElement("Row2-Column3");
    Vector rowData = new Vector();
    rowData.addElement(rowOne);
    rowData.addElement(rowTwo);
    Vector columnNames = new Vector();
    columnNames.addElement("Column One");
    columnNames.addElement("Column Two");
    columnNames.addElement("Column Three");
    JTable table = new JTable(rowData, columnNames);
```
5.

```
public JTable(TableModel model)
    TableModel model = new DefaultTableModel(rowData, columnNames);
    JTable table = new JTable(model);
```
6.

```
public JTable(TableModel model, TableColumnModel columnModel)
    // Swaps column order
    TableColumnModel columnModel = new DefaultTableColumnModel();
    TableColumn firstColumn = new TableColumn(1);
    firstColumn.setHeaderValue(headers[1]);
    columnModel.addColumn(firstColumn);
    TableColumn secondColumn = new TableColumn(0);
    secondColumn.setHeaderValue(headers[0]);
    columnModel.addColumn(secondColumn);
    JTable table = new JTable(model, columnModel);
```
7.

```
public JTable(TableModel model, TableColumnModel columnModel,
    ListSelectionModel selectionModel)
    // Set single selection mode
    ListSelectionModel selectionModel = new DefaultListSelectionModel();
    selectionModel.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    JTable table = new JTable(model, columnModel, selectionModel);
```

Scrolling JTable Components

Like other components that may require more space than what's available, the JTable component implements the Scrollable interface and should be placed within a JScrollPane. Scrollbars will appear in a JScrollPane when a JTable is too big for the available screen real estate, and column header names will appear above each column. Figure 17-4 shows how the table in Figure 17-1 would appear if it weren't within a JScrollPane. Notice that neither column headers nor scrollbars appear. This means you can't determine the meaning of the data, nor can you scroll to the undisplayed rows.



one	ichi - 一
two	ni - 二
three	san - 三
four	shi - 四
five	go - 五
six	roku - 六
seven	shichi - 七
eight	hachi - 八

Figure 17-4: JTable without a JScrollPane

Therefore, every table you create needs to be placed within a JScrollPane by using code similar to the following:

```
JTable table = new JTable(...);
JScrollPane scrollPane = new JScrollPane(table);
```

Manually Positioning the JTable View

When a JTable within a JScrollPane is added to a window, the table will automatically appear with the table positioned so that the first row and column appear in the upper-left corner. If you ever need to return the position to the origin, you can set the viewport position back to the point (0, 0).

```
scrollPane.getViewport().setViewPosition(new Point(0,0));
```

For scrolling purposes, the block increment amount is the visible width/height of the viewport, depending on the direction of the scrollbar. The unit increment is 100 pixels for horizontal scrolling and the height of a single row for vertical scrolling. See Figure 17-5 for a visual representation of these increments.

Removing Column Headers

As previously stated, placing a JTable within a JScrollPane automatically produces column header labels for the different column names. If you don't want column headers, such as in Figure 17-6, you can remove them one of many different ways.

The simplest way to remove the column headers is to provide empty strings as the column header names. With the third JTable constructor in the previous list of seven constructors, this would involve replacing the three column names with "", the empty string.

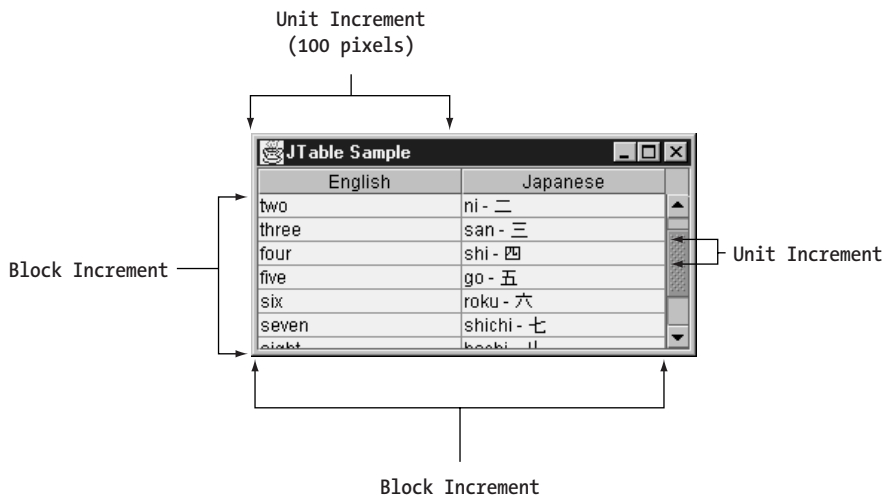


Figure 17-5: JTable scrolling increments

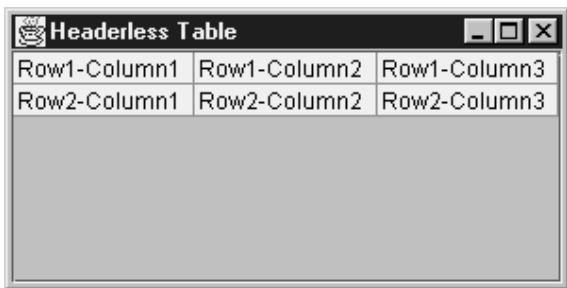


Figure 17-6: A JTable without column headers

```
Object rowData[][] = {"Row1-Column1", "Row1-Column2", "Row1-Column3"}, {"Row2-Column1", "Row2-Column2", "Row2-Column3"};
Object columnNames[] = {"", "", ""};
JTable table = new JTable(rowData, columnNames);
JScrollPane scrollPane = new JScrollPane(table);
```

Because this method of removing headers also removes the description of the different columns, you might want to use another way of hiding column headers. The simplest way is to just tell the JTable you don't want table headers:

```
table.setTableHeader(null);
```

Other ways exist to remove headers. They involve subclassing `JTable` and overriding the protected method `configureEnclosingScrollPane()` or telling every `TableColumn` that its header value is empty. All of these are more complicated ways of performing the same task.

NOTE *Calling `scrollPane.setColumnHeaderView(null)` doesn't work to clear out the column headers. Instead, it causes the `JScrollPane` to use the default column headers.*

JTable Properties

As Table 17-1 shows, the `JTable` has many properties. The 40 listed can be broken up into three logical groupings, plus one for miscellaneous properties. The three groupings are for display settings, selection settings, and auto resizing settings.

PROPERTY NAME	DATA TYPE	ACCESS
<code>accessibleContext</code>	<code>AccessibleContext</code>	read-only
<code>autoCreateColumnsFromModel</code>	<code>boolean</code>	read-write bound
<code>autoResizeMode</code>	<code>int</code>	read-write bound
<code>cellEditor</code>	<code>TableCellEditor</code>	read-write bound
<code>cellSelectionEnabled</code>	<code>boolean</code>	read-write bound
<code>columnCount</code>	<code>int</code>	read-only
<code>columnModel</code>	<code>TableColumnModel</code>	read-write bound
<code>columnSelectionAllowed</code>	<code>boolean</code>	read-write bound
<code>editing</code>	<code>boolean</code>	read-only
<code>editingColumn</code>	<code>int</code>	read-write
<code>editingRow</code>	<code>int</code>	read-write
<code>editorComponent</code>	<code>Component</code>	read-only
<code>focusTraversable</code>	<code>boolean</code>	read-only
<code>gridColor</code>	<code>Color</code>	read-write bound
<code>intercellSpacing</code>	<code>Dimension</code>	read-write
<code>managingFocus</code>	<code>boolean</code>	read-only
<code>model</code>	<code>TableModel</code>	read-write bound
<code>preferredScrollableViewportSize</code>	<code>Dimension</code>	read-write
<code>rowCount</code>	<code>int</code>	read-only
<code>rowHeight</code>	<code>int</code>	read-write bound
<code>rowMargin</code>	<code>int</code>	read-write bound
<code>rowSelectionAllowed</code>	<code>boolean</code>	read-write bound

(continued)

Table 17-1 (continued)

PROPERTY NAME	DATA TYPE	ACCESS
scrollableTracksViewportHeight	boolean	read-only
scrollableTracksViewportWidth	boolean	read-only
selectedColumn	int	read-only
selectedColumnCount	int	read-only
selectedColumns	int[]	read-only
selectedRow	int	read-only
selectedRowCount	int	read-only
selectedRows	int[]	read-only
selectionBackground	Color	read-write bound
selectionForeground	Color	read-write bound
selectionMode	int	write-only
selectionModel	ListSelectionModel	read-write bound
showGrid	boolean	write-only
showHorizontalLines	boolean	read-write bound
showVerticalLines	boolean	read-write bound
tableHeader	JTableHeader	read-write bound
UI	TableUI	read-write
UIClassID	String	read-only

Table 17-1: JTable properties

NOTE Starting with the 1.3 release, row heights are not fixed. You can change the height of an individual row with public void `setRowHeight(int row, int rowHeight)`.

Understanding Display Settings

The first subset of properties in Table 17-1 allows you to set various display options of the JTable. In addition to the inherited foreground and background properties from Component, you can change the selection foreground (selectionForeground) and background (selectionBackground) colors. You also control which (if any) gridlines appear (showGrid), as well as their color (gridColor). The remaining intercellSpacing property setting deals with the extra space within table cells.

Understanding Selection Modes

You can use any one of three different types of selection modes for a `JTable`. You can select table elements one row at a time, one column at a time, or one cell at a time. These three settings are controlled by the `rowSelectionAllowed`, `columnSelectionAllowed`, and `cellSelectionEnabled` properties. Initially, only row selection is allowed while the other two nodes are not. Because the default `ListSelectionModel` is in multi-select mode, you can select multiple rows at a time. If you don't like multi-select mode, you can change the `selectionMode` property of the `JTable`, causing the selection mode of the rows and columns of the `JTable` to change accordingly. Cell selection is enabled when both row and column selection are enabled.

NOTE *The `ListSelectionModel` class provides constants for the different selection modes. These are explored in the next section of this chapter.*

If you're ever interested in whether any of the rows or columns of the `JTable` are selected, you can inquire with one of the six additional properties of `JTable`: `selectedColumnCount`, `selectedColumn`, `selectedColumns`, `selectedRowCount`, `selectedRow`, and `selectedRows`.

Interface `ListSelectionModel`/Class `DefaultListSelectionModel`

The `ListSelectionModel` interface and `DefaultListSelectionModel` class were both covered with the `JList` component information in Chapter 13. They're used to describe the current set of rows and columns within the `JTable` component. They have three different settings, as shown in Table 17-2.

SELECTION MODES

`MULTIPLE_INTERVAL_SELECTION` (default)

`SINGLE_INTERVAL_SELECTION`

`SINGLE_SELECTION`

Table 17-2: `JTable` selection modes

The `JTable` has independent selection models for both rows and columns. The row selection model is stored with the `selectionModel` property of the `JTable`. The column selection model is stored with the `TableColumnModel`. Setting the `selectionMode` property of a `JTable` sets the selection mode for the two independent selection models of the `JTable`.

Once a selection mode has been set and a user interacts with the component, you can ask the selection model what happened, or, more precisely, what the user has selected. Table 17-3 lists the properties available to facilitate selection with the `DefaultListSelectionModel`.

PROPERTY NAME	DATA TYPE	ACCESS
<code>anchorSelectionIndex</code>	<code>int</code>	read-write
<code>leadAnchorNotificationEnabled</code>	<code>boolean</code>	read-write
<code>leadSelectionIndex</code>	<code>int</code>	read-write
<code>maxSelectionIndex</code>	<code>int</code>	read-only
<code>minSelectionIndex</code>	<code>int</code>	read-only
<code>selectionEmpty</code>	<code>boolean</code>	read-only
<code>selectionModel</code>	<code>int</code>	read-write
<code>valueIsAdjusting</code>	<code>boolean</code>	read-write

Table 17-3: *DefaultListSelectionModel* properties

If you're interested in knowing when a selection event happens, register a `ListSelectionListener` with the `ListSelectionModel`. The `ListSelectionListener` was demonstrated in Chapter 13 with the `JList` component.

TIP All table indices are zero-based. So, the first visual column is column 0 internally.

Understanding Auto Resize Modes

The last subset of the `JTable` properties deals with the column resize behavior of the `JTable`. When the `JTable` is in a column or window that changes sizes, how does it react? Table 17-4 shows the five settings supported by a `JTable`.

MODES	DESCRIPTION
AUTO_RESIZE_ALL_COLUMNS	Adjusts all column widths proportionally
AUTO_RESIZE_LAST_COLUMN	Adjusts right-most column width only to give or take space as required by the column currently being altered; if no space is available within that column, then resizing will work with the previous column until a column with available space to consume is found
AUTO_RESIZE_NEXT_COLUMN	If you're reducing the width of a neighboring column, the neighboring column will grow to fill the unused space; if you're increasing the width of a column, the neighboring column will shrink
AUTO_RESIZE_OFF	Turns off user's ability to resize columns; can still be programatically resized
AUTO_RESIZE_SUBSEQUENT_COLUMNS (default)	Adjusts width by proportionally altering columns displayed to the right of the column being changed

Table 17-4: Auto resize mode constants

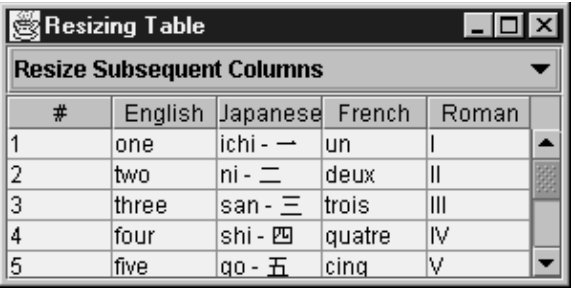


Figure 17-7: Demonstrating JTable resizing column modes

The following program demonstrates what effect each setting has when resizing table columns. Figure 17-7 shows the initial appearance of the program. Change the JComboBox and you change the column resize behavior.

```

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

public class ResizeTable {
    public static void main(String args[]) {

        Object rowData[][] = {
            {"1", "one", "ichi - \u4E00", "un", "I"},
            {"2", "two", "ni - \u4E8C", "deux", "II"},
            {"3", "three", "san - \u4E09", "trois", "III"},
            {"4", "four", "shi - \u56DB", "quatre", "IV"},
            {"5", "five", "go - \u4E94", "cinq", "V"},
            {"6", "six", "roku - \u516D", "treiza", "VI"},
            {"7", "seven", "shichi - \u4E03", "sept", "VII"},
            {"8", "eight", "hachi - \u516B", "huit", "VIII"},
            {"9", "nine", "kyu - \u4E5D", "neur", "IX"},
            {"10", "ten", "ju - \u5341", "dix", "X"}
        };

        String columnNames[] = {"#", "English", "Japanese", "French", "Roman"};

        final JTable table = new JTable(rowData, columnNames);
        JScrollPane scrollPane = new JScrollPane(table);

        String modes[] = {"Resize All Columns", "Resize Last Column", "Resize Next
Column",
    "Resize Off", "Resize Subsequent Columns"};
        final int modeKey[] = {
            JTable.AUTO_RESIZE_ALL_COLUMNS,
            JTable.AUTO_RESIZE_LAST_COLUMN,
            JTable.AUTO_RESIZE_NEXT_COLUMN,
            JTable.AUTO_RESIZE_OFF,
            JTable.AUTO_RESIZE_SUBSEQUENT_COLUMNS};
        JComboBox resizeModeComboBox = new JComboBox(modes);
        int defaultMode = 4;
        table.setAutoResizeMode(modeKey[defaultMode]);
        resizeModeComboBox.setSelectedIndex(defaultMode);
        ItemListener itemListener = new ItemListener() {
            public void itemStateChanged(ItemEvent e) {
                JComboBox source = (JComboBox)e.getSource();
                int index = source.getSelectedIndex();

```

Chapter 17

```

        table.setAutoResizeMode(modeKey[index]);
    }
};
resizeModeComboBox.addItemListener(itemListener);

JFrame frame = new ExitableJFrame("Resizing Table");
Container contentPane = frame.getContentPane();

contentPane.add(resizeModeComboBox, BorderLayout.NORTH);
contentPane.add(scrollPane, BorderLayout.CENTER);

frame.setSize(300, 150);
frame.setVisible(true);
}
}

```

Rendering Table Cells

By default, the rendering of table data is done by a `JLabel`. Whatever value is stored in the table is rendered as a text string. The odd thing is that additional “default” renderers are installed for classes such as `Date` and `Number` subclasses, but they’re not “enabled.” I’ll discuss how to enable these specialized renderers later in this chapter.

Interface `TableCellRenderer`/Class `DefaultTableCellRenderer`

The `TableCellRenderer` interface defines the single method necessary for that class to be a `TableCellRenderer`.

```

public interface TableCellRenderer {
    public Component getTableCellRendererComponent(JTable table, Object value,
        boolean isSelected, boolean hasFocus, int row, int column);
}

```

By using information given to the `getTableCellRendererComponent()` method, proper renderer components can be created and sent on their way to display the appropriate content of the `JTable`. By “proper,” I mean renderers that reflect the table cell state that you’ve decided to display, such as when you want to display selected cells differently than non-selected cells, or how you want the selected cell to be displayed when it has the input focus.

English	Japanese
four	yon - 四
five	go - 五
six	roku - 六
seven	shichi - 七
eight	hachi - 八
nine	kyu - 九
ten	ju - 十

Figure 17-8: *JTable with custom renderer*

To see a simple demonstration of this, look at Figure 17-8, which shows a renderer that alternates colors based on which row the renderer is displayed within.

The source for this renderer follows:

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

public class EvenOddRenderer implements TableCellRenderer {

    public static final DefaultTableCellRenderer DEFAULT_RENDERER =
        new DefaultTableCellRenderer();

    public Component getTableCellRendererComponent(JTable table, Object value,
        boolean isSelected, boolean hasFocus, int row, int column) {
        Component renderer = DEFAULT_RENDERER.getTableCellRendererComponent(table,
value,
        isSelected, hasFocus, row, column);
        Color foreground, background;
        ((JLabel)renderer).setOpaque(true);
        if (isSelected) {
            foreground = Color.yellow;
            background = Color.green;
        } else {
            if (row % 2 == 0) {
                foreground = Color.blue;
                background = Color.white;
            } else {
                foreground = Color.white;
                background = Color.blue;
            }
        }
        renderer.setForeground(foreground);
        renderer.setBackground(background);
    }
}
```

Chapter 17

```

        return renderer;
    }
}

```

NOTE *The `DefaultTableCellRenderer` is opaque by default. However, a painting optimization in the class changes this, in error unfortunately. Therefore, you have to make sure the renderer is opaque by resetting it to true.*

Renderers for tables can be installed for individual classes or for specific columns (there's more on columns later). To install the renderer as the default renderer for the `JTable`—in other words, for `Object.class`—use code similar to the following:

```

TableCellRenderer renderer = new EvenOddRenderer();
table.setDefaultRenderer(Object.class, renderer);

```

Once installed, the `EvenOddRenderer` will be used for any column whose class doesn't have a more specific renderer. It's the responsibility of the public `Class getColumnClass()` method of `TableModel` to return the class to be used as the renderer lookup for all the cells in a particular column. The `DefaultTableModel` returns `Object.class` for everything; therefore, `EvenOddRenderer` will be used by all table cells.

TIP *Keep in mind that one renderer component is used for every cell of every column of a particular class. No individual renderer is created for each cell.*

The sample program that used the `EvenOddRenderer` to generate Figure 17-8 follows:

```

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

public class RendererSample {
    public static void main(String args[]) {
        Object rows[][] = {
            {"one", "ichi - \u4E00"},
            {"two", "ni - \u4E8C"},
            {"three", "san - \u4E09"},

```



```

        {"four", "shi - \u56DB"},
        {"five", "go - \u4E94"},
        {"six", "roku - \u516D"},
        {"seven", "shichi - \u4E03"},
        {"eight", "hachi - \u516B"},
        {"nine", "kyu - \u4E5D"},
        {"ten", "ju - \u5341"}
    };
    Object headers[] = {"English", "Japanese"};
    JFrame frame = new ExitableJFrame("Renderer Sample");
    JTable table = new JTable(rows, headers);
    TableCellRenderer renderer = new EvenOddRenderer();
    table.setDefaultRenderer(Object.class, renderer);
    JScrollPane scrollPane = new JScrollPane(table);
    frame.getContentPane().add(scrollPane, BorderLayout.CENTER);
    frame.setSize(300, 150);
    frame.setVisible(true);
}
}

```

Using Tooltips

By default, your table cell renderers will display any tooltip text you've configured them to display. Unlike the `JTree` component, you don't have to manually register the table with the `ToolTipManager`. If, however, your table doesn't display tooltip text, the table will respond faster if you unregister the table with the `ToolTipManager` by using code such as the following:

```
ToolTipManager.sharedInstance().unregisterComponent(aTable);
```

Handling JTable Events

There are no `JTable` events that you can register directly with the `JTable`. To find out when something happens, you must register with one of the `JTable` model classes: `TableModel`, `TableColumnModel`, or `ListSelectionModel`.

Customizing a JTable Look and Feel

Each installable Swing look and feel provides a different `JTable` appearance and set of default `UIResource` value settings. Figure 17-9 shows the appearance of the `JTable` component for the preinstalled set of look and feels: Motif, Windows, and

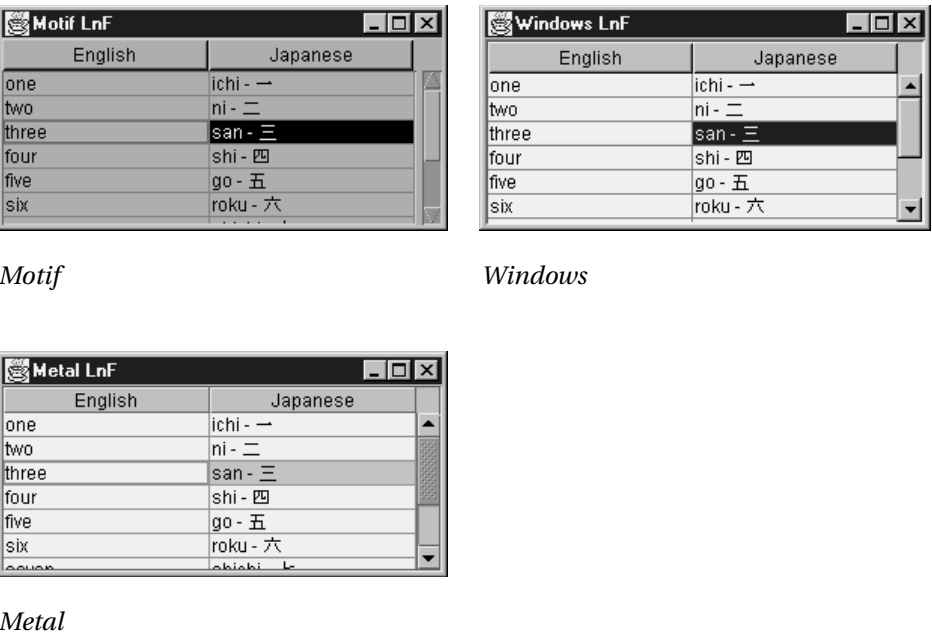


Figure 17-9: JTable under different look and feels

Metal. In all four cases, the third row is highlighted, where the coloration shows the first column is being edited and the second isn't.

The available set of UIResource-related properties for a JTable is shown in Table 17-5. The JTable component has 13 different properties.

PROPERTY STRING	OBJECT TYPE
Table.actionMap	ActionMap
Table.ancestorInputMap	InputMap
Table.background	Color
Table.focusCellBackground	Color
Table.focusCellForeground	Color
Table.focusCellHighlightBorder	Border
Table.font	Font
Table.foreground	Color
Table.gridColor	Color
Table.scrollPaneBorder	Border
Table.selectionBackground	Color
Table.selectionForeground	Color
TableUI	String

Table 17-5: JTable UIResource elements

Interface `TableModel`

Now that we've looked at the basics of the `JTable` component, we can see how it manages its data elements. It does this with the help of classes that implement the `TableModel` interface.

The `TableModel` interface defines the framework needed by the `JTable` to acquire column headers and cell values, and modify those cell values when the table is editable. Its definition follows:

```
public interface TableModel {
    // Listeners
    public void addTableModelListener(TableModelListener l);
    public void removeTableModelListener(TableModelListener l);
    // Properties
    public int getColumnCount();
    public int getRowCount();
    // Other Methods
    public Class getColumnClass(int columnIndex);
    public String getColumnName(int columnIndex);
    public Object getValueAt(int rowIndex, int columnIndex);
    public boolean isCellEditable(int rowIndex, int columnIndex);
    public void setValueAt(Object vValue, int rowIndex, int columnIndex);
}
```

Class `AbstractTableModel`

The `AbstractTableModel` class provides the basic implementation of the `TableModel` interface. It manages the `TableModelListener` list and default implementations for several of the `TableModel` methods. When you subclass it, all you simply have to provide is the actual column and row count, and the specific values (`getValueAt()`) in the table model. Column names default to labels such as "A", "B", "C", ..., "Z", "AA", "BB", ..., and the data model is read-only unless `isCellEditable()` is overridden.

If you subclass `AbstractTableModel` and make the data model editable, it's your responsibility to call one of the `fireXXX()` methods of `AbstractTableModel` to ensure that any `TableModelListener` objects are notified when the data model changes:

```
public void fireTableCellUpdated(int row, int column);
public void fireTableChanged(TableModelEvent e);
public void fireTableDataChanged();
```

Chapter 17

```
public void fireTableRowsDeleted(int firstRow, int lastRow);
public void fireTableRowsInserted(int firstRow, int lastRow);
public void fireTableRowsUpdated(int firstRow, int lastRow);
public void fireTableStructureChanged();
```

When you want to create a `JTable`, it's not uncommon to subclass `AbstractTableModel` in order to reuse an existing data structure. This data structure typically comes as the result of a Java Database Connectivity (JDBC) query, but there's no restriction requiring that to be the case. To demonstrate, the following anonymous class definition shows how you can treat an array as an `AbstractTableModel`.

```
TableModel model = new AbstractTableModel() {
    Object rowData[][] = {
        {"one", "ichi"},
        {"two", "ni"},
        {"three", "san"},
        {"four", "shi"},
        {"five", "go"},
        {"six", "roku"},
        {"seven", "shichi"},
        {"eight", "hachi"},
        {"nine", "kyu"},
        {"ten", "ju"}
    };
    Object columnNames[] = {"English", "Japanese"};
    public String getColumnName(int column) {
        return columnNames[column].toString();
    }
    public int getRowCount() {
        return rowData.length;
    }
    public int getColumnCount() {
        return columnNames.length;
    }
    public Object getValueAt(int row, int col) {
        return rowData[row][col];
    }
};
JTable table = new JTable(model);
JScrollPane scrollPane = new JScrollPane(table);
```

Specifying Fixed JTable Columns

Now that you've seen the basics of how the `TableModel` and `AbstractTableModel` work, you can create a `JTable` with some fixed columns and some columns *not* fixed. To create columns that don't scroll, you need to place a second table in the row header view of the `JScrollPane`. Then when the user scrolls the table vertically, the two tables will remain in sync. The two tables then need to share their `ListSelectionModel`. That way, when a row in one table is selected, the row in the other table will automatically be selected. Figure 17-10 shows a table with one fixed column and four scrolling columns.

The source code used to generate Figure 17-10 follows.

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

public class FixedTable {
    public static void main(String args[]) {
        final Object rowData[][] = {
            {"1", "one", "ichi", "un", "I"},
            {"2", "two", "ni", "deux", "II"},
            {"3", "three", "san", "trois", "III"},
            {"4", "four", "shi", "quatre", "IV"},
            {"5", "five", "go", "cinq", "V"},
            {"6", "six", "roku", "treiza", "VI"},
            {"7", "seven", "shichi", "sept", "VII"},
            {"8", "eight", "hachi", "huit", "VIII"},
            {"9", "nine", "kyu", "neur", "IX"},
            {"10", "ten", "ju", "dix", "X"}
        };

        final String columnNames[] = {"#", "English", "Japanese", "French", "Roman"};

        TableModel fixedColumnModel = new AbstractTableModel() {
            public int getColumnCount() {
                return 1;
            }
            public String getColumnName(int column) {
                return columnNames[column];
            }
            public int getRowCount() {
```

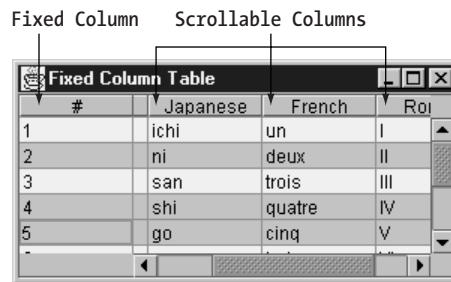


Figure 17-10: Fixed-column `JTable`

Chapter 17

```

        return rowData.length;
    }
    public Object getValueAt(int row, int column) {
        return rowData[row][column];
    }
};

TableModel mainModel = new AbstractTableModel() {
    public int getColumnCount() {
        return columnNames.length-1;
    }
    public String getColumnName(int column) {
        return columnNames[column+1];
    }
    public int getRowCount() {
        return rowData.length;
    }
    public Object getValueAt(int row, int column) {
        return rowData[row][column+1];
    }
};

JTable fixedTable = new JTable(fixedColumnModel);
fixedTable.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
//    fixedTable.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

JTable mainTable = new JTable(mainModel);
mainTable.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
//    mainTable.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

ListSelectionModel model = fixedTable.getSelectionModel();
mainTable.setSelectionModel(model);

JScrollPane scrollPane = new JScrollPane(mainTable);
Dimension fixedSize = fixedTable.getPreferredSize();
JViewport viewport = new JViewport();
viewport.setView(fixedTable);
viewport.setPreferredSize(fixedSize);
viewport.setMaximumSize(fixedSize);
scrollPane.setCorner(JScrollPane.UPPER_LEFT_CORNER,
fixedTable.getTableHeader());
scrollPane.setRowHeaderView(viewport);

```

```

JFrame frame = new ExitableJFrame("Fixed Column Table");

frame.getContentPane().add(scrollPane, BorderLayout.CENTER);
frame.setSize(300, 150);
frame.setVisible(true);
}
}

```

Enabling the Default Table Cell Renderers

Earlier, I mentioned that the `JTable` provides default renderers for `Date` and `Number` classes. Let's look at the `AbstractTableModel` class and see how to enable them.

The public Class `getColumnClass(int column)` method of `TableModel` returns the class type for a column in the data model. If the `JTable` class has a special renderer installed for that particular class, it will use it to display that class. By default, the `AbstractTableModel` (and `DefaultTableModel`) implementations of `TableModel` returns `Object.class` for everything. The `AbstractTableModel` class doesn't try to be smart about guessing what's in a column. However, if you know that a particular column of the data model will always be numbers, dates, or some other class, you can have the data model return that class type. This allows the `JTable` to try to be smart and use a better renderer.

To demonstrate, Table 17-6 shows the preinstalled renderers within the `JTable`. If you have a table full of numbers or just one column of numbers, you can override `getColumnClass()` to return `Number.class` for the appropriate columns; your numbers will be right-justified instead of left-justified. With dates, for instance, you'll get better-looking output.

CLASS	RENDERER	INFORMATION
Boolean	JCheckBox	centered
Date	JLabel	right-aligned uses DateFormat for output
ImageIcon	JLabel	centered
Number	JLabel	right-aligned uses NumberFormat for output
Object	JLabel	left-aligned

Table 17-6: Default JTable renderers

Figure 17-11 shows how a table might look before and after enabling the renderers.

You can choose to hardcode the class names for columns or have the `getColumnClass()` method be generic and just call `getClass()` on an element in the column. Adding the following code to an `AbstractTableModel` implementation

Chapter 17

English	Japanese	Boolean	Date	ImageIcon
1	ichi	true	Sat Jan 01 ...	javax.swing...
2	ni	true	Thu Apr 15 ...	javax.swing...
3	san	false	Sun Dec 07 ...	javax.swing...
4	shi	true	Tue Feb 29 ...	javax.swing...
5	go	false	Tue May 23 ...	javax.swing...

*Before**After*

Figure 17-11: Before and after enabling the renderers

would allow the `JTable` to use its default renderers. This implementation assumes that all entries for a particular column are one class type.

```
public Class getColumnClass(int column) {
    return (getValueAt(0, column).getClass());
}
```

Class `DefaultTableModel`

The `DefaultTableModel` is a subclass of `AbstractTableModel` that provides its own `Vector` data structure for storage. Everything in the data model is stored within vectors internally, even when the data is initially part of an array. In other words, if you already have your data in an adequate data structure, don't use `DefaultTableModel`. Create an `AbstractTableModel` that just uses the structure instead of having a `DefaultTableModel` convert the structure for you.

Creating a `DefaultTableModel`

There are six constructors for `DefaultTableModel`. Four map directly to `JTable` constructors, whereas the remaining two allow you to create empty tables from a set of column headers with a fixed number of rows. Once you've created the `DefaultTableModel`, you pass it along to a `JTable` constructor to create the actual table and then place the table in a `JScrollPane`.

1. `public DefaultTableModel()`
`TableModel model = new DefaultTableModel();`
2. `public DefaultTableModel(int rows, int columns)`
`TableModel model = new DefaultTableModel(2, 3)`

3.

```
public DefaultTableModel(Object rowData[][], Object columnNames[])
    Object rowData[][] = {{"Row1-Column1", "Row1-Column2", "Row1-Column3"},
        {"Row2-Column1", "Row2-Column2", "Row2-Column3"}};
    Object columnNames[] = {"Column One", "Column Two", "Column Three"};
    TableModel model = new DefaultTableModel(rowData, columnNames)
```
4.

```
public DefaultTableModel(Vector rowData, Vector columnNames)
    Vector rowOne = new Vector();
    rowOne.addElement("Row1-Column1");
    rowOne.addElement("Row1-Column2");
    rowOne.addElement("Row1-Column3");
    Vector rowTwo = new Vector();
    rowTwo.addElement("Row2-Column1");
    rowTwo.addElement("Row2-Column2");
    rowTwo.addElement("Row2-Column3");
    Vector rowData = new Vector();
    rowData.addElement(rowOne);
    rowData.addElement(rowTwo);
    Vector columnNames = new Vector();
    columnNames.addElement("Column One");
    columnNames.addElement("Column Two");
    columnNames.addElement("Column Three");
    TableModel model = new DefaultTableModel(rowData, columnNames);
```
5.

```
public DefaultTableModel(Object columnNames[], int rows)
    TableModel model = new DefaultTableModel(columnNames, 2)
```
6.

```
public DefaultTableModel(Vector columnNames, int rows)
    TableModel model = new DefaultTableModel(columnNames, 2)
```

Filling a DefaultTableModel

If you choose to use a `DefaultTableModel`, you must fill it with data for your `JTable` to display anything. In addition to basic routines to fill the data structure, there are additional methods to remove data or replace the entire contents:

- **Adding columns**

```
public void addColumn(Object columnName);
public void addColumn(Object columnName, Vector columnData);
public void addColumn(Object columnName, Object columnData[ ]);
```

- Adding rows


```
public void addRow(Object rowData[ ]);
public void addRow(Vector rowData);
```
- Inserting rows


```
public void insertRow(int row, Object rowData[ ]);
public void insertRow(int row, Vector rowData);
```
- Removing rows


```
public void removeRow( int row);
```
- Replacing contents


```
public void setDataVector(Object newData[ ][ ], Object columnNames[ ]);
public void setDataVector(Vector newData, Vector columnNames);
```

DefaultTableModel Properties

In addition to the `rowCount` and `columnCount` properties inherited from `AbstractTableModel`, `DefaultTableModel` has two additional properties, as shown in Table 17-7. Setting the `rowCount` property allows you to enlarge or shrink the table size as you please. If you are growing the model, the additional rows remain empty.

PROPERTY NAME	DATA TYPE	ACCESS
<code>columnCount</code>	<code>int</code>	read-only
<code>columnIdentifiers</code>	<code>Vector</code>	write-only
<code>dataVector</code>	<code>Vector</code>	read-only
<code>rowCount</code>	<code>int</code>	read-write

Table 17-7: DefaultTableModel properties

Creating a Sparse Table Model

The default table model implementations are meant for “full” tables, not for spreadsheets consisting of mostly empty cells. When the cells in the table are mostly empty, the default data structure for the `DefaultTableModel` will end up with plenty of wasted space. At the cost of creating a `Point` for each lookup, you can create a sparse table model that can utilize a `Hashtable` underneath it. The following demonstrates one such implementation.

```
import java.awt.Point;
import java.util.Hashtable;
import javax.swing.table.AbstractTableModel;

public class SparseTableModel extends AbstractTableModel {

    private Hashtable lookup;
    private final int rows;
    private final int columns;
    private final String headers[];

    public SparseTableModel(int rows, String columnHeaders[]) {
        if ((rows < 0) || (columnHeaders == null)) {
            throw new IllegalArgumentException("Invalid row count/columnHeaders");
        }
        this.rows = rows;
        this.columns = columnHeaders.length;
        headers = columnHeaders;
        lookup = new Hashtable();
    }

    public int getColumnCount() {
        return columns;
    }

    public int getRowCount() {
        return rows;
    }

    public String getColumnName(int column) {
        return headers[column];
    }

    public Object getValueAt(int row, int column) {
        return lookup.get(new Point(row, column));
    }

    public void setValueAt(Object value, int row, int column) {
        if ((row < 0) || (column < 0)) {
            throw new IllegalArgumentException("Invalid row/column setting");
        }
        if ((row < rows) && (column < columns)) {
            lookup.put(new Point(row, column), value);
        }
    }
}
```

Testing would just involve creating and filling up the model:

```
String headers[] = {"English", "Japanese"};
TableModel model = new SparseTableModel(10, headers);
JTable table = new JTable(model);
model.setValueAt("one", 0, 0);
model.setValueAt("ten", 9, 0);
model.setValueAt("roku - \u516D", 5, 1);
model.setValueAt("hachi - \u516B", 8, 1);
```

Listening to JTable Events with a TableModelListener

If you want to dynamically update your table data, you can work with a `TableModelListener` to find out when the data changes. The interface consists of one method that tells you when the table data changes.

```
public interface TableModelListener extends EventListener {
    public void tableChanged(TableModelEvent e);
}
```

After the `TableModelListener` is notified, you can ask the `TableModelEvent` for the type of event that happened and the range of rows and columns affected. Table 17-8 shows the properties of the `TableModelEvent` you can inquire about.

PROPERTY NAME	DATA TYPE	ACCESS
column	int	read-only
firstRow	int	read-only
lastRow	int	read-only
type	int	read-only

Table 17-8: TableModelEvent properties

The event type can be one of three type constants of `TableModelEvent`, as listed in Table 17-9.

```
TYPES
INSERT
UPDATE
DELETE
```

Table 17-9: TableModelEvent type constants

If the column property setting for the `TableModelEvent` is `ALL_COLUMNS`, then all the columns in the data model are affected. If the `firstRow` property is `HEADER_ROW`, it means the table header changed.

Sorting *JTable* Elements

The `JTable` component doesn't come with built-in support for sorting. Nevertheless, this feature is frequently requested. Sorting doesn't require changing the data model, but it *does* require changing the view of the data model that the `JTable` has.

This type of change is described by the Decorator pattern, in which you maintain the same API to the data but add sorting capabilities to the view. Figure 17-12 illustrates what the general structure of the pattern.

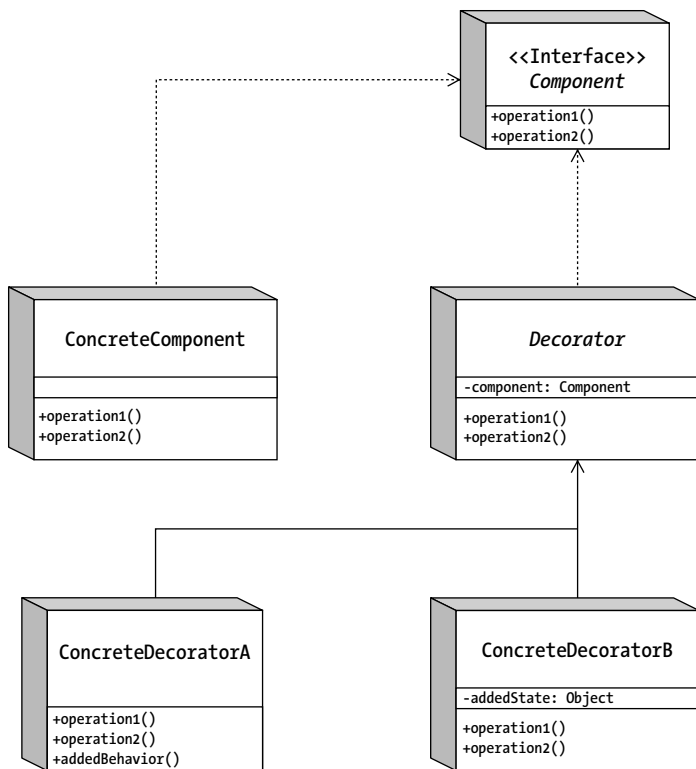


Figure 17-12: The Decorator pattern

The participants of the Decorator design pattern are the Component, ConcreteComponent, Decorator, and ConcreteDecorator(s) [A, B, C, ...]:

- Component — The component defines the service interface that will be decorated.
- ConcreteComponent — The concrete component is the object to be decorated.
- Decorator — The decorator is an abstract wrapper to a concrete component; it maintains the service interface.
- ConcreteDecorator — The concrete decorator objects extend the decorator by adding decorating responsibilities while maintaining the same programming interface. They redirect service requests to the concrete component referred to by their abstract superclass.

NOTE *The streams of the `java.io` package are examples of the Decorator pattern. The various filter streams add capabilities to the basic stream classes and maintain the same API for access.*

In our particular case for table sorting, we have a simplified example with only the Component, ConcreteComponent, and Decorator, because there is only one concrete decorator. The Component is the `TableModel` interface, the ConcreteComponent is the actual model, and the Decorator is the sorted model.

In order to sort, you need to maintain a mapping of the real data to the sorted data, and from the user interface, you must allow the user to select a column header label to enable sorting of a particular column.

To use the sorting capabilities, you tell the `TableSorter` about your data model, decorate it, and create a `JTable` from your decorated model instead of the original. To enable the sorting by picking column header labels, you need to call the custom `install()` method of the `TableHeaderSorter` class shown in the following source code for the `TableSorter` class.

```
TableSorter sorter = new TableSorter(model);
JTable table = new JTable(sorter);
TableHeaderSorter.install(sorter, table);
```

NOTE *I demonstrate the `TableSorter` in Chapter 18 where I show you how to sort a table of system properties.*

The main source code for the `TableSorter` class follows. It extends from the `TableMap` class, which follows the `TableSorter` class. The `TableSorter` class is where all the action is. The class does the sorting and notifying others that the data has changed.

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

public class TableSorter extends TableMap implements TableModelListener {
    int indexes[] = new int[0];
    Vector sortingColumns = new Vector();
    boolean ascending = true;

    public TableSorter() {
    }

    public TableSorter(TableModel model) {
        setModel(model);
    }

    public void setModel(TableModel model) {
        super.setModel(model);
        reallocateIndexes();
        sortByColumn(0);
        fireTableDataChanged();
    }

    public int compareRowsByColumn(int row1, int row2, int column) {
        Class type = model.getColumnClass(column);
        TableModel data = model;

        // Check for nulls

        Object o1 = data.getValueAt(row1, column);
        Object o2 = data.getValueAt(row2, column);

        // If both values are null return 0
        if (o1 == null && o2 == null) {
            return 0;
        } else if (o1 == null) { // Define null less than everything.
```

Chapter 17

```
        return -1;
    } else if (o2 == null) {
        return 1;
    }

    if (type.getSuperclass() == Number.class) {
        Number n1 = (Number)data.getValueAt(row1, column);
        double d1 = n1.doubleValue();
        Number n2 = (Number)data.getValueAt(row2, column);
        double d2 = n2.doubleValue();

        if (d1 < d2)
            return -1;
        else if (d1 > d2)
            return 1;
        else
            return 0;
    } else if (type == String.class) {
        String s1 = (String)data.getValueAt(row1, column);
        String s2 = (String)data.getValueAt(row2, column);
        int result = s1.compareTo(s2);

        if (result < 0)
            return -1;
        else if (result > 0)
            return 1;
        else
            return 0;
    } else if (type == java.util.Date.class) {
        Date d1 = (Date)data.getValueAt(row1, column);
        long n1 = d1.getTime();
        Date d2 = (Date)data.getValueAt(row2, column);
        long n2 = d2.getTime();

        if (n1 < n2)
            return -1;
        else if (n1 > n2)
            return 1;
        else
            return 0;
    } else if (type == Boolean.class) {
        Boolean bool1 = (Boolean)data.getValueAt(row1, column);
        boolean b1 = bool1.booleanValue();
```



```

        Boolean bool2 = (Boolean)data.getValueAt(row2, column);
        boolean b2 = bool2.booleanValue();

        if (b1 == b2)
            return 0;
        else if (b1) // Define false < true
            return 1;
        else
            return -1;
    } else {
        Object v1 = data.getValueAt(row1, column);
        String s1 = v1.toString();
        Object v2 = data.getValueAt(row2, column);
        String s2 = v2.toString();
        int result = s1.compareTo(s2);

        if (result < 0)
            return -1;
        else if (result > 0)
            return 1;
        else
            return 0;
    }
}

public int compare(int row1, int row2) {
    for (int level=0, n=sortingColumns.size(); level < n; level++) {
        Integer column = (Integer)sortingColumns.elementAt(level);
        int result = compareRowsByColumn(row1, row2, column.intValue());
        if (result != 0) {
            return (ascending ? result : -result);
        }
    }
    return 0;
}

public void reallocateIndexes() {
    int rowCount = model.getRowCount();
    indexes = new int[rowCount];
    for (int row = 0; row < rowCount; row++) {
        indexes[row] = row;
    }
}

```

Chapter 17

```
public void tableChanged(TableModelEvent tableModelEvent) {
    super.tableChanged(tableModelEvent);
    reallocateIndexes();
    sortByColumn(0);
    fireTableStructureChanged();
}

public void checkModel() {
    if (indexes.length != model.getRowCount()) {
        System.err.println("Sorter not informed of a change in model.");
    }
}

public void sort() {
    checkModel();
    shufflesort((int[])indexes.clone(), indexes, 0, indexes.length);
    fireTableDataChanged();
}

public void shufflesort(int from[], int to[], int low, int high) {
    if (high - low < 2) {
        return;
    }
    int middle = (low + high)/2;
    shufflesort(to, from, low, middle);
    shufflesort(to, from, middle, high);

    int p = low;
    int q = middle;

    for (int i = low; i < high; i++) {
        if (q >= high || (p < middle && compare(from[p], from[q]) <= 0)) {
            to[i] = from[p++];
        } else {
            to[i] = from[q++];
        }
    }
}

private void swap(int first, int second) {
    int temp      = indexes[first];
    indexes[first] = indexes[second];
    indexes[second] = temp;
}
```

```

public Object getValueAt(int row, int column) {
    checkModel();
    return model.getValueAt(indexes[row], column);
}

public void setValueAt(Object aValue, int row, int column) {
    checkModel();
    model.setValueAt(aValue, indexes[row], column);
}

public void sortByColumn(int column) {
    sortByColumn(column, true);
}

public void sortByColumn(int column, boolean ascending) {
    this.ascending = ascending;
    sortingColumns.removeAllElements();
    sortingColumns.addElement(new Integer(column));
    sort();
    super.tableChanged(new TableModelEvent(this));
}
}

```

NOTE *The TableSorter borrows heavily from the TableExample that comes with the JFC/Swing release.*

The TableMap class serves as a proxy, passing along all calls to the appropriate TableModel class. It's the superclass of the TableSorter class shown previously.

```

import javax.swing.table.*;
import javax.swing.event.*;

public class TableMap extends AbstractTableModel implements TableModelListener {

    TableModel model;

    public TableModel getModel() {
        return model;
    }
}

```

Chapter 17

```
public void setModel(TableModel model) {
    if (this.model != null) {
        this.model.removeTableModelListener(this);
    }
    this.model = model;
    if (this.model != null) {
        this.model.addTableModelListener(this);
    }
}

public Class getColumnClass(int column) {
    return model.getColumnClass(column);
}

public int getColumnCount() {
    return ((model == null) ? 0 : model.getColumnCount());
}

public String getColumnName(int column) {
    return model.getColumnName(column);
}

public int getRowCount() {
    return ((model == null) ? 0 : model.getRowCount());
}

public Object getValueAt(int row, int column) {
    return model.getValueAt(row, column);
}

public void setValueAt(Object value, int row, int column) {
    model.setValueAt(value, row, column);
}

public boolean isCellEditable(int row, int column) {
    return model.isCellEditable(row, column);
}

public void tableChanged(TableModelEvent tableModelEvent) {
    fireTableChanged(tableModelEvent);
}
}
```

Installation of the sorting routines requires the registration of a `MouseListener` so that selection triggers the sorting process. Regular mouse clicks are ascending sorts; `SHIFT`-clicks are descending sorts.

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

public class TableHeaderSorter extends MouseAdapter {

    private TableSorter sorter;
    private JTable table;

    private TableHeaderSorter() {
    }

    public static void install(TableSorter sorter, JTable table) {
        TableHeaderSorter tableHeaderSorter = new TableHeaderSorter();
        tableHeaderSorter.sorter = sorter;
        tableHeaderSorter.table = table;
        JTableHeader tableHeader = table.getTableHeader();
        tableHeader.addMouseListener(tableHeaderSorter);
    }

    public void mouseClicked(MouseEvent mouseEvent) {
        TableColumnModel columnModel = table.getColumnModel();
        int viewColumn = columnModel.getColumnIndexAtX(mouseEvent.getX());
        int column = table.convertColumnIndexToModel(viewColumn);
        if (mouseEvent.getClickCount() == 1 && column != -1) {
            System.out.println("Sorting ...");
            int shiftPressed = (mouseEvent.getModifiers() & InputEvent.SHIFT_MASK);
            boolean ascending = (shiftPressed == 0);
            sorter.sortByColumn(column, ascending);
        }
    }
}
```

Interface TableColumnModel

The `TableColumnModel` interface is one of those interfaces that lives in the background and usually doesn't require much attention. It basically manages the set of columns currently being displayed by a `JTable`. Unless triggered to do otherwise, when a `JTable` is created, the component builds a default column model from the data model, specifying that the display column order remains in the data model order.

When the `autoCreateColumnsFromModel` property of `JTable` is set (true) prior to setting the data model of the `JTable`, the `TableColumnModel` is automatically created. In addition, you can manually tell the `JTable` to create the default `TableColumnModel` if the current settings need to be reset. The public `void createDefaultColumnsFromModel()` method does the creation for you, assigning the new creation to the `TableColumnModel` of the `JTable`.

With all that automatically done for you, when would you need to look at the `TableColumnModel`? Usually, you'll look only when you don't like the defaults, or when you want to manually move things around. In addition to maintaining a set of `TableColumn` objects, the `TableColumnModel` manages a second `ListSelectionModel` which allows users to select columns from the table, as well as select rows.

Let's take a look at the interface definition before we get into the default implementation.

```
public interface TableColumnModel {
    // Listeners
    public void addColumnModelListener(TableColumnModelListener l);
    public void removeColumnModelListener(TableColumnModelListener l);
    // Properties
    public int getColumnCount();
    public int getColumnMargin();
    public void setColumnMargin(int newMargin);
    public Enumeration getColumns();
    public boolean getColumnSelectionAllowed();
    public void setColumnSelectionAllowed(boolean flag);
    public int getSelectedColumnCount();
    public int[] getSelectedColumns();
    public ListSelectionModel getSelectionModel();
    public void setSelectionModel(ListSelectionModel newModel);
    public int getTotalColumnWidth();
    // Other Methods
    public void addColumn(TableColumn aColumn);
    public TableColumn getColumn(int columnIndex);
    public int getColumnIndex(Object columnIdentifier);
    public int getColumnIndexAtX(int xPosition);
}
```

```
public void moveColumn(int columnIndex, int newIndex);
public void removeColumn(TableColumn column);
}
```

Class DefaultTableColumnModel

The DefaultTableColumnModel class defines the implementation of the TableColumnModel interface used by the system. It describes the general appearance of the TableColumn objects within the JTable by tracking margins, width, selection, and quantity. Table 17-10 shows the eight properties for accessing the DefaultTableColumnModel settings.

PROPERTY NAME	DATA TYPE	ACCESS
columnCount	int	read-only
columnMargin	int	read-write
columns	Enumeration	read-only
columnSelectionAllowed	boolean	read-write
selectedColumnCount	int	read-only
selectedColumns	int[]	read-only
selectionModel	ListSelectionModel	read-write
totalColumnWidth	int	read-only

Table 17-10: DefaultTableColumnModel properties

In addition to the class properties, you can add, remove, or move columns through the TableColumn class, which will be discussed shortly.

```
public void addColumn(TableColumn newColumn);
public void removeColumn(TableColumn oldColumn);
public void moveColumn(int currentIndex, int newIndex);
```

Listening to JTable Events with a TableColumnModelListener

One of the things you might want to do with a TableColumnModel is listen for TableColumnModelEvent objects with a TableColumnModelListener. The listener will be notified of any addition, removal, movement, or selection of columns, or changing of column margins, as shown by the listener interface definition. Do note that the different methods don't all receive TableColumnModelEvent objects when the event happens.

```
public interface TableColumnModelListener extends EventListener {
    public void columnAdded(TableColumnModelEvent e);
    public void columnMarginChanged(ChangeEvent e);
    public void columnMoved(TableColumnModelEvent e);
    public void columnRemoved(TableColumnModelEvent e);
    public void columnSelectionChanged(ListSelectionEvent e);
}
```

Because the listener definition identifies the event type, the `TableColumnModelEvent` definition only defines the range of columns affected by the change, as shown in Table 17-11.

PROPERTY NAME	DATA TYPE	ACCESS
fromIndex	int	read-only
toIndex	int	read-only

Table 17-11: *TableColumnModelEvent* properties

To see a demonstration of the `TableColumnModelListener`, you can attach a listener to one of your `TableColumnModel` objects:

```
TableColumnModel columnModel = table.getColumnModel();
columnModel.addColumnModelListener(...);
```

One such listener follows. It doesn't do much besides print a message. Nevertheless, you *can* use it to see when different events happen.

```
TableColumnModelListener tableColumnModelListener =
    new TableColumnModelListener() {
    public void columnAdded(TableColumnModelEvent e) {
        System.out.println("Added");
    }
    public void columnMarginChanged(ChangeEvent e) {
        System.out.println("Margin");
    }
    public void columnMoved(TableColumnModelEvent e) {
        System.out.println("Moved");
    }
    public void columnRemoved(TableColumnModelEvent e) {
        System.out.println("Removed");
    }
    public void columnSelectionChanged(ListSelectionEvent e) {
        System.out.println("Selected");
    }
};
```


Of course, you do have to create some code to elicit certain events. For instance, margins don't appear out of thin air. But you *can* add the same column multiple times to add more columns (or remove them). The following program tests out our new `TableColumnModelListener`.

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

public class ColumnModelSample {
    public static void main(String args[]) {
        Object rows[][] = {
            {"one", "ichi - \u4E00"},
            {"two", "ni - \u4E8C"},
            {"three", "san - \u4E09"},
            {"four", "shi - \u56DB"},
            {"five", "go - \u4E94"},
            {"six", "roku - \u516D"},
            {"seven", "shichi - \u4E03"},
            {"eight", "hachi - \u516B"},
            {"nine", "kyu - \u4E5D"},
            {"ten", "ju - \u5341"}
        };
        Object headers[] = {"English", "Japanese"};
        JFrame frame = new ExitableJFrame("Scrollless Table");
        JTable table = new JTable(rows, headers);

        TableColumnModelListener tableColumnModelListener =
            new TableColumnModelListener() {
                public void columnAdded(TableColumnModelEvent e) {
                    System.out.println("Added");
                }
                public void columnMarginChanged(ChangeEvent e) {
                    System.out.println("Margin");
                }
                public void columnMoved(TableColumnModelEvent e) {
                    System.out.println("Moved");
                }
                public void columnRemoved(TableColumnModelEvent e) {
                    System.out.println("Removed");
                }
                public void columnSelectionChanged(ListSelectionEvent e) {
```

Chapter 17

```

        System.out.println("Selection Changed");
    }
};

TableModel columnModel = table.getColumnModel();
columnModel.addColumnModelListener(tableColumnModelListener);

columnModel.setColumnMargin(12);

TableColumn column = new TableColumn(1);
columnModel.addColumn(column);

frame.getContentPane().add(table, BorderLayout.CENTER);
frame.setSize(300, 150);
frame.setVisible(true);
}
}

```

Class TableColumn

The `TableColumn` class is another important class that lives behind the scenes. Swing tables consist of a group of columns which are made up of cells. Each of those columns is described by a `TableColumn` instance. Each instance of the `TableColumn` class stores the appropriate editor, renderer, name, and sizing information. `TableColumn` objects are then grouped together into a `TableModel` to make up the current set of columns to be displayed by a `JTable`. One useful trick to remember is if you don't want a column to be displayed, remove its `TableColumn` from the `TableModel` but leave it in the `TableModel`.

Creating a TableColumn

If you choose to create your `TableColumn` objects yourself, you can use any one of four constructors to create a `TableColumn`. They cascade by adding more constructor arguments. With no arguments, such as in the first constructor in the following list, you get an empty column with a default width (75 pixels), a default editor, and a default renderer. The `modelIndex` argument allows you to specify which column from the `TableModel` you'd like the `TableColumn` to display within the `JTable`. You can also specify a width, a renderer, or an editor if you don't like the defaults. You can also specify `null` for the renderer or editor if you like one default but not the other.

- 1. `public TableColumn()`
`TableColumn column = new TableColumn()`
- 2. `public TableColumn(int modelIndex)`
`TableColumn column = new TableColumn(2)`

TIP All column settings start at zero. Therefore, new `TableColumn(2)` uses column 3 from the `TableModel`.

- 3. `public TableColumn(int modelIndex, int width)`
`TableColumn column = new TableColumn(2, 25)`
- 4. `public TableColumn(int modelIndex, int width, TableCellRenderer
renderer, TableCellEditor editor)`
`TableColumn column = new TableColumn(2, 25, aRenderer, aEditor)`

TableColumn Properties

Table 17-12 lists the 11 properties of the `TableColumn`. These properties allow you to customize a column beyond the initial set of constructor arguments. Most of the time, everything is configured for you based on the `TableModel`. However, you can still customize individual columns through the `TableColumn` class. Yes, all the properties are bound.

PROPERTY NAME	DATA TYPE	ACCESS
cellEditor	TableCellEditor	read-write bound
cellRenderer	TableCellRenderer	read-write bound
headerRenderer	TableCellRenderer	read-write bound
headerValue	Object	read-write bound
identifier	Object	read-write bound
maxWidth	int	read-write bound
minWidth	int	read-write bound
modelIndex	int	read-write bound
preferredWidth	int	read-write bound
resizable	boolean	read-write bound
width	int	read-write bound

Table 17-12: *TableColumn* properties

WARNING *If `headerRenderer` is null, the default header renderer is used by the column. [`TableCellRenderer headerRenderer = table.getTableHeader().getDefaultRenderer();`] The default is not returned by the `getHeaderRenderer()` method. This changed between versions 1.2 and 1.3 (1.2.had no default).*

NOTE *If an identifier isn't specified, the `headerValue` setting is used instead.*

Icons in Column Headers

By default, the header renderer for a table displays text or HTML. Although you can get multiple lines of text and images with HTML, there may come a time when you want to display regular `Icon` objects within a header. To do this, you must change the header's renderer. The header renderer is just another `TableCellRenderer`.

So, if we were to create a renderer that treated the value data as a `JLabel`, instead of using the value to fill the `JLabel` or for that matter any `JComponent`, then the renderer would be a little more flexible for our needs. The following is one such renderer, which is used in the program that created Figure 17-13.

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

public class JComponentTableCellRenderer implements TableCellRenderer {
    public Component getTableCellRendererComponent(JTable table, Object value,
        boolean isSelected, boolean hasFocus, int row, int column) {
        return (JComponent)value;
    }
}
```

Figure 17-13 shows how this renderer might appear with the `DiamondIcon` as the `Icon`. The source for the sample program follows.

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



Figure 17-13: Icons in table headers

```
public class LabelHeaderSample {
    public static void main(String args[]) {
        Object rows[][] = {
            {"one", "ichi - \u4E00"},
            {"two", "ni - \u4E8C"},
            {"three", "san - \u4E09"},
            {"four", "shi - \u56DB"},
            {"five", "go - \u4E94"},
            {"six", "roku - \u516D"},
            {"seven", "shichi - \u4E03"},
            {"eight", "hachi - \u516B"},
            {"nine", "kyu - \u4E5D"},
            {"ten", "ju - \u5341"}
        };
        String headers[] = {"English", "Japanese"};
        JFrame frame = new JFrame("Label Header");
        JTable table = new JTable(rows, headers);
        JScrollPane scrollPane = new JScrollPane(table);

        Icon redIcon = new ImageIcon(Color.red);
        Icon blueIcon = new ImageIcon(Color.blue);

        Border headerBorder = UIManager.getBorder("TableHeader.cellBorder");

        JLabel blueLabel = new JLabel(headers[0], blueIcon, JLabel.CENTER);
        blueLabel.setBorder(headerBorder);
        JLabel redLabel = new JLabel(headers[1], redIcon, JLabel.CENTER);
        redLabel.setBorder(headerBorder);

        TableCellRenderer renderer = new JComponentTableCellRenderer();
    }
}
```

```

        TableColumnModel columnModel = table.getColumnModel();

        TableColumn column0 = columnModel.getColumn(0);
        TableColumn column1 = columnModel.getColumn(1);

        column0.setHeaderRenderer(renderer);
        column0.setHeaderValue(blueLabel);

        column1.setHeaderRenderer(renderer);
        column1.setHeaderValue(redLabel);

        frame.getContentPane().add(scrollPane, BorderLayout.CENTER);
        frame.setSize(300, 150);
        frame.setVisible(true);
    }
}

```

Class JTableHeader

Each JTableHeader instance represents one of a set of headers for all the different columns. The set of JTableHeader objects is placed within the column header view of the JScrollPane. You rarely need to work with the JTableHeader directly. Nevertheless, you can configure many things within it.

Creating a JTableHeader

The JTableHeader has two constructors. One uses the default TableColumnModel, whereas the other is specified by the constructor.

1. `public JTableHeader()`
`JComponent headerComponent = new JTableHeader()`
2. `public JTableHeader(TableColumnModel columnModel)`
`JComponent headerComponent = new JTableHeader(aColumnModel)`

JTableHeader Properties

As Table 17-13 shows, JTableHeader has 10 different properties. These properties allow you to configure what the user can do with a particular column or how the column is shown.

PROPERTY NAME	DATA TYPE	ACCESS
accessibleContext	AccessibleContext	read-only
columnModel	TableColumnModel	read-write bound
draggedColumn	TableColumn	read-write
draggedDistance	int	read-write
reorderingAllowed	boolean	read-write bound
resizingAllowed	boolean	read-write bound
resizingColumn	TableColumn	read-write
table	JTable	read-write bound
UI	TableHeaderUI	read-write
UIClassID	String	read-only

Table 17-13: *JTableHeader* properties

Using Tooltips

By default, if you set tooltip text for the table header, it's the same tooltip text for all columns headers. To specify a tooltip for a given column, you need to create or get the renderer, and then set the tooltip for the renderer. This is true for the individual cells, too. Don't just set the tooltip for the default header or else all headers will share it.

Figure 17-14 shows how the results of this customization would appear. The source for the customization follows: Unless you previously set the headers, there is no real need to look to see if the header for a specific column is null first.

```
JLabel headerRenderer = new DefaultTableCellRenderer();
String columnName = table.getModel().getColumnName(0);
headerRenderer.setText(columnName);
headerRenderer.setToolTipText("Wave");
TableColumnModel columnModel = table.getColumnModel();
TableColumn englishColumn = columnModel.getColumn(0);
englishColumn.setHeaderRenderer((TableCellRenderer)headerRenderer);
```

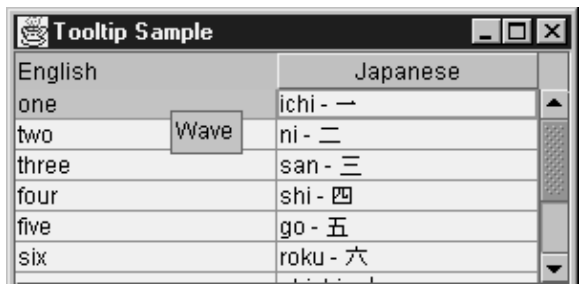


Figure 17-14: Header tooltips

Customizing JTableHeader Look and Feel

The available set of UIResource-related properties for a JTableHeader is shown in Table 17-14. The five settings control the color, font, and border for the header renderers.

PROPERTY	STRING	OBJECT	TYPE
TableHeader.background		Color	
TableHeader.cellBorder		Border	
TableHeader.font		Font	
TableHeader.foreground		Color	
TableHeaderUI		String	

Table 17-14: JTableHeader UIResource elements

NOTE For an example of creating column headers spanning multiple columns, see the *Java Programmer's Source Book* at <http://www.codeguru.com/java/Swing>.

Editing Table Cells

Editing JTable cells is nearly identical to editing JTree cells. In fact, the default table cell editor, DefaultCellEditor, implements both the TableCellEditor and TreeCellEditor interfaces, allowing you to use the same editor for both tables and trees.

Clicking on an editable cell will place the cell in edit mode. (The number of clicks depends on the type of editor.) In addition, the default editor for all cells is a JTextField. Although this works great for many data types, it's not always appropriate for many others. So, you should either not support editing of non-textual information or set up specialized editors for your JTable. With a JTable, you register an editor for a particular class type or column. Then, when the table runs across a cell of the appropriate type, the necessary editor is used. When no specialized editor is installed, the JTextField is used, even when it's inappropriate for the content!

NOTE Thankfully, the default editor for 1.3 is smarter than the one in earlier versions. For instance, you can at least live with the default editor for things like Date objects where editing a Date cell converts the current setting to and from a String successfully.

Interface TableCellEditor/Class DefaultCellEditor

The TableCellEditor interface defines the single method necessary to get an editor cell for a JTable. The argument list for TableCellEditor is identical to the TableCellRenderer with the exception of the hasFocused argument. Because the cell is being edited, it's already known to have the input focus.

```
public interface TableCellEditor extends CellEditor {
    public Component getTableCellEditorComponent(JTable table, Object value,
        boolean isSelected, int row, int column);
}
```

As described in Chapter 16, the DefaultCellEditor provides an implementation of the interface. It offers a JTextField as one editor, a JCheckBox for another, and a JComboBox for a third.

As Table 17-15 shows, in most cases the default editor is the JTextField. If the cell data can be converted to and from a string, and the class provides a constructor with a String argument, the editor offers the text representation of the data value for the initial editing value. You can then edit the contents.

CLASS	EDITOR	INFORMATION
Boolean	JCheckBox	centered
Object	JTextField	left-aligned

Table 17-15: Default JTable editors

Creating a Simple Cell Editor

To make life easier, you can provide a fixed set of color choices to the user. Then when a color is picked, you have the appropriate Color value to return to the table model. The DefaultCellEditor offers a JComboBox for just this situation in which each choice is the color. After configuring the ListCellRenderer for the JComboBox to display colors properly, you have a TableCellEditor for picking colors. Figure 17-15 shows how this might appear.

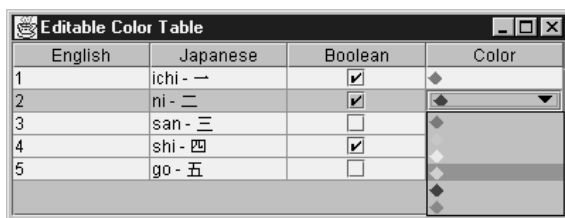


Figure 17-15: JComboBox color editor

TIP *Any time you can predefine all the choices, you can use the JComboBox as your editor through DefaultCellEditor.*

The following class represents the TableCellRenderer for the Color column of the previous example and the ListCellRenderer for the JComboBox TableCellEditor. Because of the many similarities of the two renderer components, their definitions are combined into one class.

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

public class ComboTableCellRenderer implements ListCellRenderer,
TableCellRenderer {
    DefaultListCellRenderer listRenderer = new DefaultListCellRenderer();
    DefaultTableCellRenderer tableRenderer = new DefaultTableCellRenderer();

    private void configureRenderer(JLabel renderer, Object value) {
        if ((value != null) && (value instanceof Color)) {
            renderer.setIcon(new DiamondIcon((Color)value));
            renderer.setText("");
        } else {
            renderer.setIcon(null);
            renderer.setText((String)value);
        }
    }

    public Component getListCellRendererComponent(JList list, Object value,
        int index, boolean isSelected, boolean cellHasFocus) {
        listRenderer =
        (DefaultListCellRenderer)listRenderer.getListCellRendererComponent(
            list, value, index, isSelected, cellHasFocus);
        configureRenderer(listRenderer, value);
        return listRenderer;
    }

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

```

(DefaultTableCellRenderer)tableRenderer.getTableCellRendererComponent(
    table, value, isSelected, hasFocus, row, column);
configureRenderer(tableRenderer, value);
return tableRenderer;
}
}

```

To demonstrate the use of our new combined renderer and show a simple table cell editor, the following program creates a data model in which one of the columns is a `Color`. After installing the renderer twice and setting up the table cell editor, the table can be shown and the `Color` column can be edited.

```

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

public class EditableColorColumn {

    public static void main(String args[]) {

        Color choices[] = {Color.red, Color.orange, Color.yellow, Color.green,
            Color.blue, Color.magenta};
        ComboTableCellRenderer renderer = new ComboTableCellRenderer();
        JComboBox comboBox = new JComboBox(choices);
        comboBox.setRenderer(renderer);
        TableCellEditor editor = new DefaultCellEditor(comboBox);

        JFrame frame = new ExitableJFrame("Editable Color Table");
        TableModel model = new ColorTableModel();
        JTable table = new JTable(model);

        TableColumn column = table.getColumnModel().getColumn(3);
        column.setCellRenderer(renderer);
        column.setCellEditor(editor);

        JScrollPane scrollPane = new JScrollPane(table);
        frame.getContentPane().add(scrollPane, BorderLayout.CENTER);
        frame.setSize(400, 150);
        frame.setVisible(true);
    }
}

```

Chapter 17

The following is the table model used for this example and the next:

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

public class ColorTableModel extends AbstractTableModel {

    Object rowData[][] = {
        {"1", "ichi - \u4E00", Boolean.TRUE, Color.red},
        {"2", "ni - \u4E8C", Boolean.TRUE, Color.blue},
        {"3", "san - \u4E09", Boolean.FALSE, Color.green},
        {"4", "shi - \u56DB", Boolean.TRUE, Color.magenta},
        {"5", "go - \u4E94", Boolean.FALSE, Color.pink},
    };

    String columnNames[] = {"English", "Japanese", "Boolean", "Color"};
    public int getColumnCount() {
        return columnNames.length;
    }
    public String getColumnName(int column) {
        return columnNames[column];
    }
    public int getRowCount() {
        return rowData.length;
    }
    public Object getValueAt(int row, int column) {
        return rowData[row][column];
    }
    public Class getColumnClass(int column) {
        return (getValueAt(0, column).getClass());
    }
    public void setValueAt(Object value, int row, int column) {
        rowData[row][column]=value;
    }
    public boolean isCellEditable(int row, int column) {
        return (column != 0);
    }
}
```

Creating a Complex Cell Editor

Although the previous example demonstrates how to provide a fixed set of choices to the user in a combo box `TableCellEditor`, offering the `JColorChooser` as an option (at least in the case of colors) seems to be a better choice. When defining your own `TableCellEditor`, you must implement the single `TableCellEditor` method to get the appropriate component. You must also implement the seven methods of the `CellEditor` because they both manage and notify a list of `CellEditorListener` objects as well as control when a cell is editable. Starting with an `AbstractCellEditor` subclass makes the latter choice much simpler.

By extending the `AbstractCellEditor` class, only the `getCellEditorValue()` method from the `CellEditor` methods requires customization for the editor. Doing that and providing a `JButton` that pops up a `JColorChooser` when pressed provides the entire editor component.

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

public class ColorChooserEditor extends AbstractCellEditor implements
    TableCellEditor {

    private JButton delegate = new JButton();

    Color savedColor;

    public ColorChooserEditor() {
        ActionListener actionListener = new ActionListener() {
            public void actionPerformed (ActionEvent actionEvent) {
                Color color = JColorChooser.showDialog(
                    delegate, "Color Chooser", savedColor);
                ColorChooserEditor.this.changeColor(color);
            }
        };
        delegate.addActionListener(actionListener);
    }
}
```

Chapter 17

```

public Object getCellEditorValue() {
    return savedColor;
}

private void changeColor(Color color) {
    if (color != null) {
        savedColor = color;
        delegate.setIcon(new DiamondIcon(color));
    }
}

public Component getTableCellEditorComponent (JTable table, Object value,
    boolean isSelected, int row, int column) {
    changeColor((Color)value);
    return delegate;
}
}

```

Figure 17-16 shows the ColorChooserEditor in action, with the associated table in the background.

A sample program using our new ColorChooserEditor follows. The example reuses the earlier ColorTableModel data model. Setting up the ColorChooserEditor simply involves setting the TableCellEditor for the appropriate column.

```

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

```

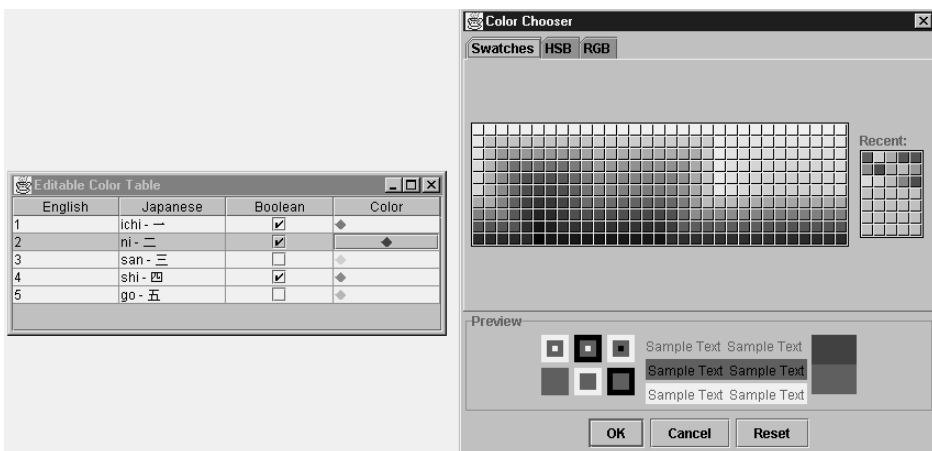


Figure 17-16: Pop-up color editor

```
public class ChooserTableSample {

    public static void main(String args[]) {

        JFrame frame = new ExitableJFrame("Editable Color Table");
        TableModel model = new ColorTableModel();
        JTable table = new JTable(model);

        TableColumn column = table.getColumnModel().getColumn(3);

        ComboTableCellRenderer renderer = new ComboTableCellRenderer();
        column.setCellRenderer(renderer);

        TableCellEditor editor = new ColorChooserEditor();
        column.setCellEditor(editor);

        JScrollPane scrollPane = new JScrollPane(table);
        frame.getContentPane().add(scrollPane, BorderLayout.CENTER);
        frame.setSize(400, 150);
        frame.setVisible(true);
    }
}
```

Summary

In this chapter, we explored the inner depths of the `JTable` component. We looked at customizing the `TableModel`, `TableColumnModel`, and `ListSelectionModel` for the `JTable`. We delved into both the abstract and concrete implementations of the different table models. In addition, we examined the inner elements of the various table models, such as the `TableColumn` and `JTableHeader` classes. We also looked into how to customize the display and editing of the `JTable` cells by providing a custom `TableCellRenderer` and `TableCellEditor`.

In the Chapter 18, we'll explore the pluggable look and feel architecture of the JFC/Swing component set.

