

# ***eChapter 1***

## **Swing: Tables and Trees**

<i><b>If you need an immediate solution to:</b></i>	<i><b>See page:</b></i>
Creating Tables	<i>e4</i>
Adding Rows and Columns to Tables at Runtime	<i>e19</i>
Adding Controls and Images to Tables	<i>e22</i>
Creating Trees	<i>e25</i>
Adding Data to Trees	<i>e34</i>
Handling Tree Events	<i>e36</i>
Editing Tree Nodes	<i>e38</i>

---

# In Depth

In this chapter, we'll take a look at two of the biggest components of Swing: tables and trees. Both these topics are very large in scope and could take up an entire volume by themselves. You may guess their functions from their names: Tables present the user with data in tabular form, and trees display hierarchical data in a way that's become common—using expandable nodes. Before digging into these topics, we'll take an overview of each.

## Tables

Tables may be the most complex component type in Swing. In fact, a whole package is devoted to them: **swing.table**. Tables present and let the user edit data arranged into rows and columns. In Swing, tables are, themselves, made up of column headers and column objects. You can, of course, also address data by row and by individual cell, although those constructs are not objects.

Swing tables are very flexible, which accounts for some of their complexity. You can edit the data in a table directly, use a variety of selection modes (including column, row, and individual cell selection), use your own models for columns and column headers, decorate tables, draw your own cells by deriving a new cell renderer from the **TableCellRenderer** interface, handle cell editing by implementing the **CellEditor** interface, use keystroke actions, reorder columns, resize columns, and much more.

As with other Swing components, you can customize just about everything in a table, and because there are so many subcomponents here, the customization process can become very complex.

## Trees

One of the new components in GUI programming that lets users handle data is the *tree*. If you've used Microsoft Windows Explorer, you know about trees already—this utility presents a directory structure using a tree. Trees use a “folder and leaf” analogy to present data; when you double-click a displayed folder, the pages (or *leaves*) inside that folder appear, as well as any subfolders. You can also close folders again by double-clicking them. Trees arrange their data into a vertical hierarchy, connected with lines, making for an easily scrolled presentation.

They're often used to display directories (as folders) and files (as leaves) in an easily manageable way.

Like tables, trees are complex enough to have their own package in Swing: **swing.tree**. In programming terms, trees are made up of *nodes*, and each node is either a folder or a leaf. Folders can have child nodes. All the nodes except for the root node have one single parent node. As you might guess, the appearance of folders and leaves is look-and-feel dependent. When you open a folder, it's considered *expanded*, and when you close it, it's *collapsed*. As with tables, you can customize trees with your own renderers and editors. You'll get a good introduction to the topic in this chapter.

That's it for the overview of what's in this chapter. There's a lot coming up here: Tables and trees represent a huge amount of programming depth. It's time to turn to the "Immediate Solutions" section, starting with a look at the **JTable** class.

# Immediate Solutions

## Creating Tables

The Big Boss appears and says, “About that new spreadsheet program you’re writing. . .” “I didn’t know I was writing anything like that,” you say. “Don’t interrupt,” says the BB. “We need it yesterday. Is it done?” “Hmm,” you say, “sounds like a job for the **JTable** class.”

**JTable** is the Swing component that presents data in a two-dimensional table format. Here’s the inheritance diagram for **JTable**:

```

java.lang.Object
|____java.awt.Component
      |____java.awt.Container
            |____javax.swing.JComponent
                  |____javax.swing.JTable
  
```

You’ll find the fields of the **JTable** class in Table 1.1, its constructors in Table 1.2, and its methods in Table 1.3 (note how extensive this class is).

**Table 1.1** Fields of the **JTable** class.

Field	Does This
<b>static int AUTO_RESIZE_ALL_COLUMNS</b>	Resizes columns during resize operations.
<b>static int AUTO_RESIZE_LAST_COLUMN</b>	Resizes the last column only during all resize operations.
<b>static int AUTO_RESIZE_NEXT_COLUMN</b>	When a column is adjusted, this field adjusts the next column the opposite way.
<b>static int AUTO_RESIZE_OFF</b>	Uses a scrollbar to adjust column width.
<b>static int AUTO_RESIZE_SUBSEQUENT_COLUMNS</b>	Changes subsequent columns to preserve the total width upon resizing.
<b>protected boolean autoCreateColumnsFromModel</b>	Queries the <b>TableModel</b> object to build the default set of columns if this field is True.
<b>protected int autoResizeMode</b>	Sets whether the table automatically resizes the width of its columns to take up the entire width of the table.

(continued)

Table 1.1 Fields of the **JTable** class (*continued*).

Field	Does This
<b>protected TableCellEditor cellEditor</b>	An object that overwrites the current cell and allows the user to change those contents.
<b>protected boolean cellSelectionEnabled</b>	If this field is True, both a row selection and a column selection can be set at the same time.
<b>protected TableColumnModel columnModel</b>	The <b>TableColumnModel</b> of the table.
<b>protected TableModel dataModel</b>	The <b>TableModel</b> of the table.
<b>protected Hashtable defaultEditorsByColumnClass</b>	The table of objects that display and edit the contents of a cell, indexed by class.
<b>protected Hashtable defaultRenderersByColumnClass</b>	The table of objects that display the contents of a cell, indexed by class.
<b>protected int editingColumn</b>	The column of the cell being edited.
<b>protected int editingRow</b>	The row of the cell being edited.
<b>protected Component editorComp</b>	The component that handles editing.
<b>protected Color gridColor</b>	The color of the grid.
<b>protected Dimension preferredViewportSize</b>	Used by the <b>Scrollable</b> interface to determine the initial visible area.
<b>protected int rowHeight</b>	The height of rows in the table.
<b>protected int rowMargin</b>	The height margin between rows.
<b>protected boolean rowSelectionAllowed</b>	Returns True if row selection is allowed in this table.
<b>protected Color selectionBackground</b>	The background color of selected cells.
<b>protected Color selectionForeground</b>	The foreground color of selected cells.
<b>protected ListSelectionModel selectionModel</b>	The <b>ListSelectionModel</b> of the table, used to keep track of row selections.
<b>protected boolean showHorizontalLines</b>	Horizontal lines are drawn between cells if this field is True.
<b>protected boolean showVerticalLines</b>	Vertical lines are drawn between cells if this field is True.
<b>protected JTableHeader tableHeader</b>	The <b>TableHeader</b> working with the table.

Table 1.2 Constructors of the **JTable** class.

Constructor	Does This
<b>JTable()</b>	Constructs a default <b>JTable</b> .
<b>JTable(int numRows, int numColumns)</b>	Constructs a <b>JTable</b> with <b>numRows</b> and <b>numColumns</b> of empty cells.
<b>JTable(Object[][] rowData, Object[] columnNames)</b>	Constructs a <b>JTable</b> to display the values in the two-dimensional array, <b>rowData</b> , with column names, <b>columnNames</b> .
<b>JTable(TableModel dm)</b>	Constructs a <b>JTable</b> that's initialized with <b>dm</b> as the data model, a default column model, and a default selection model.
<b>JTable(TableModel dm, TableColumnModel cm)</b>	Constructs a <b>JTable</b> that's initialized with <b>dm</b> as the data model, <b>cm</b> as the column model, and a default selection model.
<b>JTable(TableModel dm, TableColumnModel cm, ListSelectionModel sm)</b>	Constructs a <b>JTable</b> that's initialized with <b>dm</b> as the data model, <b>cm</b> as the column model, and <b>sm</b> as the selection model.
<b>JTable(Vector rowData, Vector columnNames)</b>	Constructs a <b>JTable</b> to display the values in the vector of <b>Vectors</b> , <b>rowData</b> , with column names as given in <b>columnNames</b> .

Table 1.3 Methods of the **JTable** class.

Method	Does This
<b>void addColumn(TableColumn aColumn)</b>	Appends a column to the end of the array of columns.
<b>void addColumnSelectionInterval(int index0, int index1)</b>	Adds the columns from <b>index0</b> to <b>index1</b> , inclusive, to the selection.
<b>void addNotify()</b>	Calls <b>configureEnclosingScrollPane</b> .
<b>void addRowSelectionInterval(int index0, int index1)</b>	Adds the rows from <b>index0</b> to <b>index1</b> , inclusive, to the selection.
<b>void clearSelection()</b>	Deselects all selected columns and rows.
<b>void columnAdded(TableColumnModelEvent e)</b>	Indicates to listeners that a column was added to the model.
<b>int columnAtPoint(Point point)</b>	Gets the index of the column that <b>point</b> lies in (or -1 if it lies outside the receiver's bounds).
<b>void columnMarginChanged(ChangeEvent e)</b>	Notifies listeners that a column was moved because of a margin change.

(continued)

Table 1.3 Methods of the **JTable** class (*continued*).

Method	Does This
<b>void columnMoved(TableColumnModelEvent e)</b>	Notifies listeners that a column was repositioned.
<b>void columnRemoved(TableColumnModelEvent e)</b>	Notifies listeners that a column was removed from the model.
<b>void columnSelectionChanged(ListSelectionEvent e)</b>	Notifies listeners that the selection model of the <b>TableColumnModel</b> changed.
<b>protected void configureEnclosingScrollPane()</b>	Configures the enclosing <b>ScrollPane</b> by installing the table's <b>tableHeader</b> as the <b>columnHeaderView</b> of the scroll pane, and so on.
<b>int convertColumnIndexToModel(int viewColumnIndex)</b>	Gets the index of the column in the model whose data is being displayed in the column <b>viewColumnIndex</b> in the display.
<b>int convertColumnIndexToView(int modelColumnIndex)</b>	Gets the index of the column in the view that's displaying the data from the column <b>modelColumnIndex</b> in the model.
<b>protected TableColumnModel createDefaultColumnModel()</b>	Gets the default column model object, which is a <b>DefaultTableColumnModel</b> .
<b>void createDefaultColumnsFromModel()</b>	Creates default columns for the table from the data model using the <b>getColumnCount()</b> and <b>getColumnClass()</b> methods defined in the <b>TableModel</b> interface.
<b>protected TableModel createDefaultDataModel()</b>	Gets the default table model object, which is a <b>DefaultTableModel</b> .
<b>protected void createDefaultEditors()</b>	Creates default cell editors for objects, numbers, and boolean values.
<b>protected void createDefaultRenderers()</b>	Creates the default renderers.
<b>protected ListSelectionModel createDefaultSelectionModel()</b>	Gets the default selection model object, which is a <b>DefaultListSelectionModel</b> .
<b>protected JTableHeader createDefaultTableHeader()</b>	Gets the default table header object, which is a <b>JTableHeader</b> .
<b>static JScrollPane createScrollPaneForTable(JTable aTable)</b>	Deprecated. Replaced by the new <b>JScrollPane(aTable)</b> .
<b>void doLayout()</b>	Causes this table to lay out its rows and columns.

*(continued)*

Table 1.3 Methods of the **JTable** class (*continued*).

Method	Does This
<b>boolean</b> <code>editCellAt(int row, int column)</code>	Starts editing the cell at <b>row</b> and <b>column</b> , if the cell is editable.
<b>boolean</b> <code>editCellAt(int row, int column, EventObject e)</code>	Starts editing the cell at <b>row</b> and <b>column</b> , if the cell is editable.
<b>void</b> <code>editingCanceled(ChangeEvent e)</code>	Invoked when editing is canceled.
<b>void</b> <code>editingStopped(ChangeEvent e)</code>	Invoked when editing is finished.
<b>AccessibleContext</b> <code>getAccessibleContext()</code>	Gets the <b>AccessibleContext</b> associated with this <b>JComponent</b> .
<b>boolean</b> <code>getAutoCreateColumnsFromModel()</code>	Gets whether the table will create default columns from the model.
<b>int</b> <code>getAutoResizeMode()</code>	Gets the autosize mode of the table.
<b>TableCellEditor</b> <code>getCellEditor()</code>	Gets the <b>cellEditor</b> .
<b>TableCellEditor</b> <code>getCellEditor(int row, int column)</code>	Gets an appropriate editor for the cell given by this row and column.
<b>Rectangle</b> <code>getCellRect(int row, int column, boolean includeSpacing)</code>	Gets a rectangle locating the cell that lies at the intersection of <b>row</b> and <b>column</b> .
<b>TableCellRenderer</b> <code>getCellRenderer(int row, int column)</code>	Gets an appropriate renderer for the cell given by this row and column.
<b>boolean</b> <code>getCellSelectionEnabled()</code>	Returns True if simultaneous row and column selections are allowed.
<b>TableColumn</b> <code>getColumn(Object identifier)</code>	Gets the <b>TableColumn</b> object for the column in the table whose identifier is equal to <b>identifier</b> , when compared using <b>equals()</b> .
<b>Class</b> <code>getColumnClass(int column)</code>	Gets the type of the column at the given view position.
<b>int</b> <code>getColumnCount()</code>	Gets the number of columns in the column model (note that this may be different from the number of columns in the table model).
<b>TableColumnModel</b> <code>getColumnModel()</code>	Gets the <b>TableColumnModel</b> that contains all column information of this table.
<b>String</b> <code>getColumnName(int column)</code>	Gets the name of the column at the given view position.
<b>boolean</b> <code>getColumnSelectionAllowed()</code>	Returns True if columns can be selected.
<b>TableCellEditor</b> <code>getDefaultEditor(Class columnClass)</code>	Gets the editor to be used when no editor has been set in a <b>TableColumn</b> .

*(continued)*



**Table 1.3** Methods of the **JTable** class (*continued*).

Method	Does This
<b>TableCellRenderer</b> getDefaultRenderer(Class columnClass)	Gets the renderer to be used when no renderer has been set in a <b>TableColumn</b> .
<b>int</b> getEditingColumn()	Gets the index of the editing column.
<b>int</b> getEditingRow()	Gets the index of the editing row.
<b>Component</b> getEditorComponent()	Gets the component that was returned from the <b>CellEditor</b> .
<b>Color</b> getGridColor()	Gets the color used to draw grid lines.
<b>Dimension</b> getInterCellSpacing()	Gets the horizontal and vertical spacing between cells.
<b>TableModel</b> getModel()	Gets the <b>TableModel</b> that provides the data displayed by the receiver.
<b>Dimension</b> getPreferredSizeScrollableViewportSize()	Gets the preferred size of the viewport for this table.
<b>int</b> getRowCount()	Gets the number of rows in the table.
<b>int</b> getRowHeight()	Gets the height of a table row in the receiver.
<b>int</b> getRowMargin()	Gets the amount of empty space between rows.
<b>boolean</b> getRowSelectionAllowed()	Returns True if rows can be selected.
<b>int</b> getScrollableBlockIncrement(Rectangle visibleRect, int orientation, int direction)	Gets <b>visibleRect.height</b> or <b>visibleRect.width</b> , depending on the table's orientation.
<b>boolean</b> getScrollableTracksViewportHeight()	Returns False if the height of the viewport does not determine the height of the table.
<b>boolean</b> getScrollableTracksViewportWidth()	Returns False if the width of the viewport does not determine the width of the table.
<b>int</b> getScrollableUnitIncrement(Rectangle visibleRect, int orientation, int direction)	Gets the scroll increment that completely exposes one new row or column.
<b>int</b> getSelectedColumn()	Gets the index of the first selected column or -1 if no column is selected.
<b>int</b> getSelectedColumnCount()	Gets the number of selected columns.
<b>int[]</b> getSelectedColumns()	Gets the indexes of all selected columns.
<b>int</b> getSelectedRow()	Gets the index of the first selected row or -1 if no row is selected.
<b>int</b> getSelectedRowCount()	Gets the number of selected rows.
<b>int[]</b> getSelectedRows()	Gets the indexes of all selected rows.

(continued)

Table 1.3 Methods of the **JTable** class (*continued*).

Method	Does This
<b>Color</b> <code>getSelectionBackground()</code>	Gets the background color for selected cells.
<b>Color</b> <code>getSelectionForeground()</code>	Gets the foreground color for selected cells.
<b>ListSelectionModel</b> <code>getSelectionModel()</code>	Gets the <b>ListSelectionModel</b> that's used to maintain row selection state.
<b>boolean</b> <code>getShowHorizontalLines()</code>	Returns True if the receiver draws horizontal lines between cells and False if it doesn't.
<b>boolean</b> <code>getShowVerticalLines()</code>	Returns True if the receiver draws vertical lines between cells and False if it doesn't.
<b>JTableHeader</b> <code>getTableHeader()</code>	Gets the <b>tableHeader</b> working with this <b>JTable</b> .
<b>String</b> <code>getToolTipText(MouseEvent event)</code>	Overrides <b>JComponent</b> 's <b>setToolTipText</b> method to allow use of the renderer's tips (if the renderer has text set).
<b>TableUI</b> <code>getUI()</code>	Gets the look-and-feel object that renders this component.
<b>String</b> <code>getUIClassID()</code>	Gets the name of the look-and-feel class that renders this component.
<b>Object</b> <code>getValueAt(int row, int column)</code>	Gets the cell value at <b>row</b> and <b>column</b> .
<b>protected void</b> <code>initializeLocalVars()</code>	Initializes table properties to their default values.
<b>boolean</b> <code>isCellEditable(int row, int column)</code>	Returns True if the cell at <b>row</b> and <b>column</b> is editable.
<b>boolean</b> <code>isCellSelected(int row, int column)</code>	Returns True if the cell at the given position is selected.
<b>boolean</b> <code>isColumnSelected(int column)</code>	Returns True if the column at the given index is selected.
<b>boolean</b> <code>isEditing()</code>	Returns True if the table is editing a cell.
<b>boolean</b> <code>isManagingFocus()</code>	Overridden to return True.
<b>boolean</b> <code>isRowSelected(int row)</code>	Returns True if the row at the given index is selected.
<b>void</b> <code>moveColumn(int column, int targetColumn)</code>	Moves the column <b>column</b> to the position currently occupied by the column <b>targetColumn</b> .
<b>protected String</b> <code> paramString()</code>	Gets a string representation of this <b>JTable</b> .
<b>Component</b> <code>prepareEditor(TableCellEditor editor, int row, int column)</code>	Prepares the given editor using the value at the given cell.

*(continued)*

Table 1.3 Methods of the **JTable** class (*continued*).

Method	Does This
<b>Component prepareRenderer(TableCellRenderer renderer, int row, int column)</b>	Prepares the given renderer with an appropriate value from the <b>dataModel</b> .
<b>protected boolean processKeyBinding (KeyStroke ks, KeyEvent e, int condition, boolean pressed)</b>	Invoked to process the key bindings for <b>ks</b> as the result of the <b>KeyEvent e</b> .
<b>void removeColumn(TableColumn aColumn)</b>	Removes a column from the <b>JTable</b> 's array of columns.
<b>void removeColumnSelectionInterval(int index0, int index1)</b>	Deselects the columns from <b>index0</b> to <b>index1</b> , inclusive.
<b>void removeEditor()</b>	Discards the editor object.
<b>void removeNotify ()</b>	Calls the <b>unconfigureEnclosingScrollPane</b> method.
<b>void removeRowSelectionInterval(int index0, int index1)</b>	Deselects the rows from <b>index0</b> to <b>index1</b> , inclusive.
<b>protected void resizeAndRepaint ()</b>	Equivalent to <b>revalidate</b> followed by <b>repaint</b> .
<b>int rowAtPoint (Point point)</b>	Returns the index of the row that <b>point</b> lies in or -1 if the result is not in the range <b>[0, getRowCount()-1]</b> .
<b>protected void resizeAndRepaint()</b>	Equivalent to <b>revalidate()</b> followed by <b>repaint()</b> .
<b>int rowAtPoint(Point point)</b>	Gets the index of the row that <b>point</b> lies in.
<b>void selectAll()</b>	Selects all rows, columns, and cells in the table.
<b>void setAutoCreateColumnsFromModel(boolean createColumns)</b>	Sets the table's <b>autoCreateColumnsFromModel</b> flag.
<b>void setAutoResizeMode(int mode)</b>	Sets the table's autoresize mode when the table is resized.
<b>void setCellEditor(TableCellEditor anEditor)</b>	Sets the <b>cellEditor</b> variable.
<b>void setCellSelectionEnabled(boolean flag)</b>	Sets whether this table allows both a column selection and a row selection to exist at the same time.
<b>void setColumnModel(TableColumnModel newModel)</b>	Sets the column model for this table to <b>newModel</b> and registers for listener notifications from the new column model.
<b>void setColumnSelectionAllowed(boolean flag)</b>	Sets whether the columns in this model can be selected.

*(continued)*

Table 1.3 Methods of the **JTable** class (*continued*).

Method	Does This
<b>void setColumnSelectionInterval(int index0, int index1)</b>	Selects the columns from <b>index0</b> to <b>index1</b> , inclusive.
<b>void setDefaultEditor(Class columnClass, TableCellEditor editor)</b>	Sets a default editor to be used if no editor has been set in a <b>TableColumn</b> .
<b>void setDefaultRenderer(Class columnClass, TableCellRenderer renderer)</b>	Sets a default renderer to be used if no renderer has been set in a <b>TableColumn</b> .
<b>void setEditingColumn(int aColumn)</b>	Sets the <b>editingColumn</b> variable.
<b>void setEditingRow(int aRow)</b>	Sets the <b>editingRow</b> variable.
<b>void setGridColor(Color newColor)</b>	Sets the color used to draw grid lines to <b>color</b> and redisplay the receiver.
<b>void setIntercellSpacing(Dimension newSpacing)</b>	Sets the width and height between cells to <b>newSpacing</b> and redisplay the receiver.
<b>void setModel(TableModel newModel)</b>	Sets the data model for this table to <b>newModel</b> and registers for listener notifications from the new data model.
<b>void setPreferredScrollableViewportSize(Dimension size)</b>	Sets the preferred size of the viewport for this table.
<b>void setRowHeight(int newHeight)</b>	Sets the height for rows.
<b>void setRowMargin(int rowMargin)</b>	Sets the amount of empty space between rows.
<b>void setRowSelectionAllowed(boolean flag)</b>	Sets whether the rows in this model can be selected.
<b>void setRowSelectionInterval(int index0, int index1)</b>	Selects the rows from <b>index0</b> to <b>index1</b> , inclusive.
<b>void setSelectionBackground(Color selectionBackground)</b>	Sets the background color for selected cells.
<b>void setSelectionForeground(Color selectionForeground)</b>	Sets the foreground color for selected cells.
<b>void setSelectionMode(int selectionMode)</b>	Sets the table's selection mode to allow single selections, a single contiguous interval, or multiple intervals.
<b>void setSelectionModel(ListSelectionModel newModel)</b>	Sets the row-selection model for this table to <b>newModel</b> .
<b>void setShowGrid(boolean b)</b>	Sets whether to draw grid lines around cells.
<b>void setShowHorizontalLines(boolean b)</b>	Sets whether to draw horizontal lines between cells.

*(continued)*

Table 1.3 Methods of the **JTable** class (*continued*).

Method	Does This
<b>void setShowVerticalLines(boolean b)</b>	Sets whether to draw vertical lines between cells.
<b>void setTableHeader(JTableHeader newHeader)</b>	Sets the <b>tableHeader</b> working with this <b>JTable</b> to <b>newHeader</b> .
<b>void setUI(TableUI ui)</b>	Sets the look-and-feel object that draws this component.
<b>void setValueAt(Object aValue, int row, int column)</b>	Sets the value for the cell at <b>row</b> and <b>column</b> .
<b>void sizeColumnsToFit(boolean lastColumnOnly)</b>	Deprecated. Replaced by <b>sizeColumnsToFit(int)</b> .
<b>void sizeColumnsToFit(int resizingColumn)</b>	Resizes one or more of the columns in the table so that the total width of all of <b>JTable</b> 's columns will be equal to the width of the table.
<b>void tableChanged(TableModelEvent e)</b>	Called when the table is changed.
<b>protected void unconfigureEnclosingScrollPane ()</b>	Reverses the effect of <b>configureEnclosingScrollPane</b> by replacing the <b>columnHeaderView</b> of the enclosing scroll pane with null.
<b>void updateUI()</b>	Called by <b>UIManager</b> when the look and feel has changed.
<b>void valueChanged(ListSelectionEvent e)</b>	Invoked when the selection changes.

The model you use to hold the data in a table is derived from the **AbstractTableModel** class. The default table model is the **DefaultTableModel** class. Here's the inheritance diagram for this class:

```

java.lang.Object
|
|____ javax.swing.table.AbstractTableModel
|
|____ javax.swing.table.DefaultTableModel

```

You'll find the fields of the **DefaultTableModel** class in Table 1.4, its constructors in Table 1.5, and its methods in Table 1.6.

Table 1.4 Fields of the **DefaultTableModel** class.

Field	Does This
<b>protected Vector columnIdentifiers</b>	A <b>Vector</b> of column identifiers
<b>protected Vector dataVector</b>	A <b>Vector</b> of object values

Table 1.5 Constructors of the **DefaultTableModel** class.

Constructor	Does This
<b>DefaultTableModel()</b>	Constructs a <b>DefaultTableModel</b> .
<b>DefaultTableModel(int numRows, int numColumns)</b>	Constructs a <b>DefaultTableModel</b> with <b>numRows</b> and <b>numColumns</b> .
<b>DefaultTableModel(Object[][] data, Object[] columnNames)</b>	Constructs a <b>DefaultTableModel</b> and initializes the table by passing <b>data</b> and <b>columnNames</b> to the <b>setDataVector</b> method.
<b>DefaultTableModel(Object[] columnNames, int numRows)</b>	Constructs a <b>DefaultTableModel</b> with as many columns as there are elements in <b>columnNames</b> and <b>numRows</b> of null object values.
<b>DefaultTableModel(Vector columnNames, int numRows)</b>	Constructs a <b>DefaultTableModel</b> with as many columns as there are elements in <b>columnNames</b> and <b>numRows</b> of null object values.
<b>DefaultTableModel(Vector data, Vector columnNames)</b>	Constructs a <b>DefaultTableModel</b> and initializes the table by passing <b>data</b> and <b>columnNames</b> to the <b>setDataVector</b> method.

Table 1.6 Methods of the **DefaultTableModel** class.

Method	Does This
<b>void addColumn(Object columnName)</b>	Adds a column to the model.
<b>void addColumn(Object columnName, Object[] columnData)</b>	Adds a column to the model with name <b>columnName</b> .
<b>void addColumn(Object columnName, Vector columnData)</b>	Adds a column to the model.
<b>void addRow(Object[] rowData)</b>	Adds a row to the end of the model.
<b>void addRow(Vector rowData)</b>	Adds a row to the end of the model.
<b>protected static Vector convertToVector(Object[] anArray)</b>	Gets a vector that contains the same objects as the array.
<b>protected static Vector convertToVector(Object[][] anArray)</b>	Gets a <b>Vector</b> of <b>Vector</b> objects that contains the same objects as the array.
<b>int getColumnCount()</b>	Gets the number of columns in this data table.
<b>String getColumnName(int column)</b>	Gets the column name.
<b>Vector getDataVector()</b>	Returns the <b>Vector</b> of <b>Vectors</b> that contains the table's data values.
<b>int getRowCount()</b>	Gets the number of rows in this data table.

(continued)

Table 1.6 Methods of the **DefaultTableModel** class (*continued*).

Method	Does This
<b>Object</b> <code>getValueAt(int row, int column)</code>	Gets an attribute value for the cell at <b>row</b> and <b>column</b> .
<b>void</b> <code>insertRow(int row, Object[] rowData)</code>	Inserts a row at <b>row</b> in the model.
<b>void</b> <code>insertRow(int row, Vector rowData)</code>	Inserts a row at <b>row</b> in the model.
<b>boolean</b> <code>isCellEditable(int row, int column)</code>	Returns True if the cell at <b>row</b> and <b>column</b> is editable.
<b>void</b> <code>moveRow(int startIndex, int endIndex, int toIndex)</code>	Moves one or more rows starting at <b>startIndex</b> to <b>endIndex</b> in the model to the <b>toIndex</b> .
<b>void</b> <code>newDataAvailable(TableModelEvent event)</code>	Equivalent to <b>fireTableChanged</b> .
<b>void</b> <code>newRowsAdded(TableModelEvent event)</code>	This method will make sure the new rows have the correct number of columns.
<b>void</b> <code>removeRow(int row)</code>	Removes the row at <b>row</b> from the model.
<b>void</b> <code>rowsRemoved(TableModelEvent event)</code>	Equivalent to <b>fireTableChanged()</b> .
<b>void</b> <code>setColumnCount (int columnCount)</code>	Sets the number of columns in the model.
<b>void</b> <code>setColumnIdentifiers(Object[] newIdentifiers)</code>	Replaces the column identifiers in the model.
<b>void</b> <code>setColumnIdentifiers(Vector newIdentifiers)</code>	Replaces the column identifiers in the model.
<b>void</b> <code>setDataVector(Object[][] newData, Object[] columnNames)</code>	Replaces the value in the <b>dataVector</b> instance variable with the values in the array <b>newData</b> .
<b>void</b> <code>setDataVector(Vector newData, Vector columnNames)</code>	Replaces the current <b>dataVector</b> instance variable with the new vector of rows, <b>newData</b> .
<b>void</b> <code>setNumRows(int newSize)</code>	Sets the number of rows in the model.
<b>void</b> <code>setValueAt(Object aValue, int row, int column)</code>	Sets the object value for the cell at <b>column</b> and <b>row</b> .

Although the **JTable** class is complex, it provides defaults for most aspects of table programming, which makes things easier. Here's an example in which I use the **DefaultTableModel** class to add data to a table, row by row. First, I create a new object of the **DefaultTableModel** class and a new **JTable** object that displays the data in that model object:

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

/*
<APPLET
```

```

        CODE = table.class
        WIDTH = 350
        HEIGHT = 280 >
</APPLET>
*/

public class table extends JApplet
{
    DefaultTableModel defaulttablemodel = new DefaultTableModel();
    JTable jtable = new JTable(defaulttablemodel);
    .
    .
    .
}

```

Next, I use the default table model's **addColumn** method to add columns to the table, giving them the captions Column 0, Column 1, and so on:

```

public class table extends JApplet
{
    DefaultTableModel defaulttablemodel = new DefaultTableModel();
    JTable jtable = new JTable(defaulttablemodel);

    public void init()
    {
        for(int column = 0; column < 5; column++){
            defaulttablemodel.addColumn("Column " + column);
        }
        .
        .
        .
    }
}

```

Having added columns to the table, I'm ready to add the data the table will display in its rows, and I'll do that by constructing each row and then using the model's **addRow** method. Each row is an array of objects; in this case, I'll use **String** objects. Here's how I construct each row and then add it to the table:

```

public class table extends JApplet
{
    Object[] data = new Object[5];

    DefaultTableModel defaulttablemodel = new DefaultTableModel();
    JTable jtable = new JTable(defaulttablemodel);

    public void init()

```



```

{
    for(int column = 0; column < 5; column++){
        defaulttablemodel.addColumn("Column " + column);
    }

    for(int row = 0; row < 10; row++) {
        for(int column = 0; column < 5; column++) {
            data[column] = "Cell " + row + "," + column;
        }
        defaulttablemodel.addRow(data);
    }
    .
    .
    .
}
}

```

All that's left is to display the new table. Tables are usually displayed in scroll panes (in fact, you should do so to make sure column headers appear). Here's how I add the table to a **JScrollPane** object and then display it:

```

public class table extends JApplet
{
    Object[] data = new Object[5];

    DefaultTableModel defaulttablemodel = new DefaultTableModel();
    JTable jtable = new JTable(defaulttablemodel);

    public void init()
    {
        for(int column = 0; column < 5; column++){
            defaulttablemodel.addColumn("Column " + column);
        }

        for(int row = 0; row < 10; row++) {
            for(int column = 0; column < 5; column++) {
                data[column] = "Cell " + row + "," + column;
            }
            defaulttablemodel.addRow(data);
        }
        getContentPane().add(new JScrollPane(jtable));
    }
}

```

And that's it. This table appears in Figure 1.1. You can see the table with the column headers in the figure. This example is table.java on the CD.

Column 0	Column 1	Column 3	Column 2	Column 4
Cell 0,0	Cell 0,1	Cell 0,3	Cell 0,2	Cell 0,4
Cell 1,0	Cell 1,1	Cell 1,3	Cell 1,2	Cell 1,4
Cell 2,0	Cell 2,1	Cell 2,3	Cell 2,2	Cell 2,4
Cell 3,0	Cell 3,1	Cell 3,3	Cell 3,2	Cell 3,4
Cell 4,0	Cell 4,1	Cell 4,3	Cell 4,2	Cell 4,4
Cell 5,0	Cell 5,1	Cell 5,3	Cell 5,2	Cell 5,4
Cell 6,0	Cell 6,1	Cell 6,3	Cell 6,2	Cell 6,4
Cell 7,0	Cell 7,1	Cell 7,3	Cell 7,2	Cell 7,4
Cell 8,0	Cell 8,1	Cell 8,3	Cell 8,2	Cell 8,4

Figure 1.1 A basic table.

You can already do a lot with this table; for example, you can resize columns simply by resizing the column headers. The default selection mode is by row, so you can select rows by double-clicking them. Also, you can select multiple rows by using the Ctrl and Shift keys. You can also edit the data in any cell by triple-clicking that cell, which makes it display a blinking text insertion caret, thus turning it into something much like a text field (you can create your own custom cell editors based on other controls). When you edit the text in a cell and press Enter, the new data appears in that cell. In fact, you can even rearrange the columns in the table just by dragging column headers. For example, in Figure 1.2, I've switch columns 2 and 3 around.

Here's another way of constructing a table. In this case, I create a new table model derived from the **AbstractTableModel** class, overriding the **getColumnCount** method to indicate how many columns the table should have, the **getRowCount** method to indicate how many rows the table should have, and the **getValueAt** method to supply the data cell by cell. Here's the code:

```
TableModel newModel = new AbstractTableModel() {
    public int getColumnCount() {return 100;}
    public int getRowCount() {return 100;}
    public Object getValueAt(int row, int column) {return new
        String("Cell " + row + ", " + column);}
};

JTable jtable = new JTable(newModel);
getContentPane().add(new JScrollPane(jtable));
```

**Related solution:**

Creating Scroll Panes

**Found on page:**

Chapter 15

Column 0	Column 1	Column 3	Column 2	Column 4
Cell 0,0	Cell 0,1	Cell 0,3	Cell 0,2	Cell 0,4
Cell 1,0	Cell 1,1	Cell 1,3	Cell 1,2	Cell 1,4
Cell 2,0	Cell 2,1	Cell 2,3	Cell 2,2	Cell 2,4
Cell 3,0	Cell 3,1	Cell 3,3	Cell 3,2	Cell 3,4
Cell 4,0	Cell 4,1	Cell 4,3	Cell 4,2	Cell 4,4
Cell 5,0	Cell 5,1	Cell 5,3	Cell 5,2	Cell 5,4
Cell 6,0	Cell 6,1	Cell 6,3	Cell 6,2	Cell 6,4
Cell 7,0	Cell 7,1	Cell 7,3	Cell 7,2	Cell 7,4
Cell 8,0	Cell 8,1	Cell 8,3	Cell 8,2	Cell 8,4
Cell 9,0	Cell 9,1	Cell 9,3	Cell 9,2	Cell 9,4

Applet started.

Figure 1.2 Switching column positions in a basic table.

## Adding Rows and Columns to Tables at Runtime

The Big Boss appears, aggrieved, and says, “Well, your new program does indeed have a spreadsheet in it, but it’s pretty limited.” “How limited?” you ask. “Well,” the BB says, “for one thing, it can only use spreadsheets that have five columns and five rows.” “That’s a limitation?” you ask.

To add rows and columns to a table at runtime, you can use the table model’s **addRow** and **addColumn** methods (and you can use the **removeRow** and **removeColumn** methods to remove them). This is a very useful feature, because tables need not be static.

We’ll take a look at an example here that lets the user add rows and columns to a table at runtime. I start by creating a table and a panel that has two buttons: Create a new row and Create a new column. Here’s what this looks like in code:

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

/*
<APPLET
    CODE = tableadd.class
    WIDTH = 350
    HEIGHT = 280 >
</APPLET>
*/
```

```

public class tableadd extends JApplet
{
    Object[] data = new Object[5];

    DefaultTableModel defaulttablemodel = new DefaultTableModel();
    JTable jtable = new JTable(defaulttablemodel);

    public void init()
    {
        for(int column = 0; column < 5; column++){
            defaulttablemodel.addColumn("Column " + column);
        }

        for(int row = 0; row < 5; row++) {
            for(int column = 0; column < 5; column++) {
                data[column] = "Cell " + row + "," + column;
            }
            defaulttablemodel.addRow(data);
        }

        getContentPane().add(new JScrollPane(jtable), BorderLayout.CENTER);
        getContentPane().add(new JPanel(), BorderLayout.SOUTH);
    }
    .
    .
    .

```

I'll add a new inner class to implement the button panel. Here's what creating the new buttons and connecting them looks like:

```

class jpanel extends JPanel implements ActionListener
{
    JButton jbutton1 = new JButton("Create new row"),
        jbutton2 = new JButton("Create new column");

    public jpanel()
    {
        add(jbutton1);
        add(jbutton2);

        jbutton1.addActionListener(this);
        jbutton2.addActionListener(this);
    }
    .
    .
    .
}

```

The real action takes place when the user clicks a button. When the user clicks the Create new row button, I use the **addRow** method to add a new row to the table after I fill it with data (note that I use the table model's **getRowCount** and **getColumnCount** methods to determine the current dimensions of the table):

```
public void actionPerformed(ActionEvent e)
{
    if(e.getSource() == jButton1) {
        int numberrows = defaulttablemodel.getRowCount();
        int numbercolumns = defaulttablemodel.getColumnCount();

        Object[] data = new Object[numbercolumns];

        for(int column = 0; column < numbercolumns; column++) {
            data[column] = "Cell " + numberrows + "," + column;
        }
        defaulttablemodel.addRow(data);
    }
}
```

When the user clicks the Create new column button, on the other hand, I create a new column with the **addColumn** method and fill it with data using the **setValueAt** method:

```
public void actionPerformed(ActionEvent e)
{
    if(e.getSource() == jButton1) {
        int numberrows = defaulttablemodel.getRowCount();
        int numbercolumns = defaulttablemodel.getColumnCount();

        Object[] data = new Object[numbercolumns];

        for(int column = 0; column < numbercolumns; column++) {
            data[column] = "Cell " + numberrows + "," + column;
        }
        defaulttablemodel.addRow(data);

    } else if(e.getSource() == jButton2) {
        int numberrows = defaulttablemodel.getRowCount();
        int numbercolumns = defaulttablemodel.getColumnCount();
        defaulttablemodel.addColumn("Column " + numbercolumns);

        for(int row = 0; row < numberrows; row++) {
            defaulttablemodel.setValueAt("Cell " + row + "," +
                numbercolumns, row, numbercolumns);
        }
    }
}
```

```

    }
    jTable.setSizeColumnsToFit(0);
}
}
}

```

Note the use of the **sizeColumnsToFit** method at the end of this code; calling this method resizes the columns in the table to fit into the scroll pane after I've added a new column. In fact, you can set the **JTable** resize mode like this:

```
jTable.setAutoResizeMode(JTable.AUTO_RESIZE_ALL_COLUMNS);
```

However, the table will still not be resized when you add a new column (although it should be), so you need to call **sizeColumnsToFit**.

The result of this code appears in Figure 1.3. In this figure, I've added one row and one column to the table. You'll find it as `tableadd.java` on the CD.

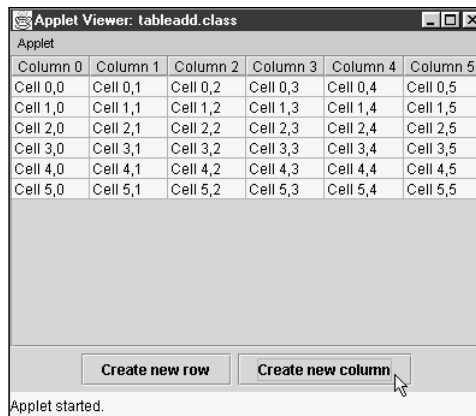


Figure 1.3 Adding a row and column to a table at runtime.

## Adding Controls and Images to Tables

“Hey,” says the Novice Programmer, “I thought tables were Swing objects.” “They are,” you say. “Why?” “Well,” says the NP, “what about adding images to them?” You smile and say, “no problem.”

In this solution, we’ll take a look at handling different data types in a table. Here, I create a table that manages data on various sandwich types for a restaurant,

including string, floating-point, Boolean, date, and image data. Note that the Boolean data is automatically handled in tables by a checkbox being displayed in the table cell.

The default table model won't let you use heterogeneous data types such as images, so I create a new model that extends **DefaultTableModel** in this example. That new model will be named, appropriately enough, **newModel**. Here's how I can fill a table with heterogeneous data using that model:

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

/*
<APPLET
    CODE = tableimages.class
    WIDTH = 600
    HEIGHT = 280 >
</APPLET>
*/

public class tableimages extends JApplet
{
    String[] columns = {"Sandwich", "Available", "Price", "Date", "Image"};

    Date date = (new GregorianCalendar(2000, 9, 2)).getTime();

    Object[][] data = {
        {"Ham", new Boolean(false), new Float(4.99), date, new
            ImageIcon("table.jpg")},

        {"BBQ", new Boolean(true), new Float(5.99), date, new
            ImageIcon("table.jpg")},

        {"Turkey", new Boolean(false), new Float(4.99), date, new
            ImageIcon("table.jpg")},

        {"Watercress", new Boolean(true), new Float(4.99), date, new
            ImageIcon("table.jpg")},

        {"Cheese", new Boolean(false), new Float(4.99), date, new
            ImageIcon("table.jpg")},
    };
}
```

```

        {"Beef", new Boolean(true), new Float(4.99), date, new
            ImageIcon("table.jpg")}
    };

    JTable jtable = new JTable(new newModel(data, columns));

    public void init()
    {
        getContentPane().add(new JScrollPane(jtable));
    }
}

```

All that's left is to derive **newModel** from **DefaultTableModel**. To do that, I override three methods: the **DefaultTableModel** constructor (to install the data in the model), the **isCellEditable** method (to indicate that all cells are not editable), and the **getColumnClass** method (to return the class of the data in each cell—this is the method you must implement to enable heterogeneous data handling). Here's what this looks like in code (note that I use the **getClass** method to get the class of the data in each cell, and I use the **elementAt** method to get the element at a particular location; **elementAt** returns an object of the **Vector** class, and we'll see that class later in this book):

```

class newModel extends DefaultTableModel
{
    public newModel(Object[][] data, Object[] columns)
    {
        super(data, columns);
    }

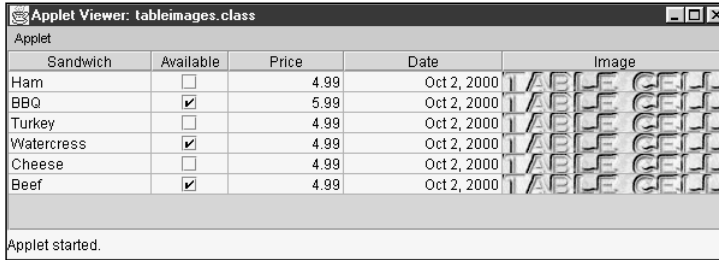
    public boolean isCellEditable(int row, int col)
    {
        return false;
    }

    public Class getColumnClass(int column)
    {
        Vector v = (Vector) dataVector.elementAt(0);

        return v.elementAt(column).getClass();
    }
}

```





Sandwich	Available	Price	Date	Image
Ham	<input type="checkbox"/>	4.99	Oct 2, 2000	TABLE CELL
BBQ	<input checked="" type="checkbox"/>	5.99	Oct 2, 2000	TABLE CELL
Turkey	<input type="checkbox"/>	4.99	Oct 2, 2000	TABLE CELL
Watercress	<input checked="" type="checkbox"/>	4.99	Oct 2, 2000	TABLE CELL
Cheese	<input type="checkbox"/>	4.99	Oct 2, 2000	TABLE CELL
Beef	<input checked="" type="checkbox"/>	4.99	Oct 2, 2000	TABLE CELL

Applet started.

Figure 1.4 Adding images and checkboxes to a table.

The result of this code appears in Figure 1.4. As you can see in the figure, this new table supports various kinds of data, as represented by checkboxes, numbers, and images. This example is `tableimages.java` on the CD.

## Creating Trees

“Uh-oh,” says the Novice Programmer, “the Big Boss wants me to create a file directory utility that displays files and directories using a tree.” “Well,” you say, “you can use a *tree*.” “Great,” says the Novice Programmer, “only, what’s a tree?”

A *tree* is a control that displays hierarchical data as an outline based on nodes. A specific node can be identified either by a **TreePath** object (an object that encapsulates a node and all the nodes it’s descended from—called its *ancestors*) or by its display row (the row in the display shows only one node). You can expand nodes to display all their children. A *collapsed node* is one that hides its children, and a *hidden node* is one that’s under a collapsed parent.

Trees are supported by the **JTree** class in Swing. Here’s what the inheritance diagram for this class looks like:

```

java.lang.Object
|___java.awt.Component
    |___java.awt.Container
        |___javax.swing.JComponent
            |___javax.swing.JTree

```

You can find the fields of the **JTree** class in Table 1.7, its constructors in Table 1.8, and its methods in Table 1.9.

Table 1.7 Fields of the **JTree** class.

Field	Does This
<b>static String CELL_EDITOR_PROPERTY</b>	The bound property name for <b>cellEditor</b> .
<b>static String CELL_RENDERER_PROPERTY</b>	The bound property name for <b>cellRenderer</b> .
<b>protected TreeCellEditor cellEditor</b>	The editor for the entries.
<b>protected TreeCellRenderer cellRenderer</b>	The cell renderer used to draw nodes.
<b>protected boolean editable</b>	Returns True if the tree is editable.
<b>static String EDITABLE_PROPERTY</b>	The bound property name for <b>editable</b> .
<b>static String INVOKES_STOP_CELL_EDITING_PROPERTY</b>	The bound property name for <b>messagesStopCellEditing</b> .
<b>protected boolean invokesStopCellEditing</b>	Returns True when editing is to be stopped.
<b>static String LARGE_MODEL_PROPERTY</b>	The bound property name for <b>largeModel</b> .
<b>protected boolean largeModel</b>	Returns True if the tree uses a large model.
<b>static String ROOT_VISIBLE_PROPERTY</b>	The bound property name for <b>rootVisible</b> .
<b>protected boolean rootVisible</b>	Returns True if the root node is displayed and False if its children are the highest visible nodes.
<b>static String ROW_HEIGHT_PROPERTY</b>	The bound property name for <b>rowHeight</b> .
<b>protected int rowHeight</b>	The height to use for each display row.
<b>static String SCROLLS_ON_EXPAND_PROPERTY</b>	The bound property name for <b>scrollsOnExpand</b> .
<b>protected boolean scrollsOnExpand</b>	Returns True if a node will scroll when expanded to show all descendants.
<b>static String SELECTION_MODEL_PROPERTY</b>	The bound property name for <b>selectionModel</b> .
<b>protected TreeSelectionModel selectionModel</b>	Models the set of selected nodes in this tree.
<b>protected JTree.TreeSelectionRedirector selectionRedirector</b>	Creates a new event and passes it to the <b>selectionListeners</b> .
<b>static String SHOWS_ROOT_HANDLES_PROPERTY</b>	The bound property name for <b>showsRootHandles</b> .
<b>protected boolean showsRootHandles</b>	Returns True if handles are displayed at the topmost level of the tree.
<b>protected int toggleClickCount</b>	The number of mouse clicks before a node is expanded.
<b>static String TREE_MODEL_PROPERTY</b>	The bound property name for <b>treeModel</b> .
<b>protected TreeModel treeModel</b>	The model that defines the tree displayed by this object.
<b>protected TreeModelListener treeModelListener</b>	Updates the <b>expandedState</b> .
<b>static String VISIBLE_ROW_COUNT_PROPERTY</b>	The bound property name for <b>visibleRowCount</b> .
<b>protected int visibleRowCount</b>	The number of rows to make visible at one time.

Table 1.8 Constructors of the **JTree** class.

Constructor	Does This
<b>JTree()</b>	Constructs a <b>JTree</b> with a sample model.
<b>JTree(Hashtable value)</b>	Constructs a <b>JTree</b> created from a hashtable.
<b>JTree(Object[] value)</b>	Constructs a <b>JTree</b> with each element of the given array as the child of a new root node.
<b>JTree(TreeModel newModel)</b>	Constructs an instance of <b>JTree</b> that displays the root node using the given data model.
<b>JTree(TreeNode root)</b>	Constructs a <b>JTree</b> with the given <b>TreeNode</b> as its root, which displays the root node.
<b>JTree(TreeNode root, boolean asksAllowsChildren)</b>	Constructs a <b>JTree</b> with the given <b>TreeNode</b> as its root, which displays the root node and whether a node is a leaf node in the given manner.
<b>JTree(Vector value)</b>	Constructs a <b>JTree</b> with each element of the given <b>Vector</b> as the child of a new root node that is not displayed.

Table 1.9 Methods of the **JTree** class.

Method	Does This
<b>void addSelectionInterval(int index0, int index1)</b>	Adds the paths between <b>index0</b> and <b>index1</b> , inclusive, to the selection.
<b>void addSelectionPath(TreePath path)</b>	Adds the node identified by the given <b>TreePath</b> to the selection.
<b>void addSelectionPaths(TreePath[] paths)</b>	Adds each path in the array of paths to the selection.
<b>void addSelectionRow(int row)</b>	Adds the path at the given row to the selection.
<b>void addSelectionRows(int[] rows)</b>	Adds the paths at each of the given rows to the selection.
<b>void addTreeExpansionListener(TreeExpansionListener tel)</b>	Adds a listener for <b>TreeExpansion</b> events.
<b>void addTreeSelectionListener(TreeSelectionListener tsl)</b>	Adds a listener for <b>TreeSelection</b> events.
<b>void addTreeWillExpandListener(TreeWillExpandListener tel)</b>	Adds a listener for <b>TreeWillExpand</b> events.
<b>void cancelEditing()</b>	Cancels the current editing session.
<b>void clearSelection()</b>	Clears the selection.
<b>protected void clearToggledPaths()</b>	Clears the cache of toggled paths.
<b>void collapsePath(TreePath path)</b>	Makes sure that the node identified by the given path is collapsed and viewable.

(continued)

Table 1.9 Methods of the **JTree** class (*continued*).

Method	Does This
<b>void collapseRow(int row)</b>	Makes sure the node in the given row is collapsed.
<b>String convertValueToText(Object value, boolean selected, boolean expanded, boolean leaf, int row, boolean hasFocus)</b>	Called by the renderers to convert the given value to text.
<b>protected static TreeModel createTreeModel(Object value)</b>	Gets a <b>TreeModel</b> encapsulating the given object.
<b>protected TreeModelListener createTreeModelListener()</b>	Creates and returns an instance of <b>TreeModelHandler</b> .
<b>void expandPath(TreePath path)</b>	Makes sure the node identified by the given path is expanded.
<b>void expandRow(int row)</b>	Makes sure the node in the given row is expanded.
<b>void fireTreeCollapsed(TreePath path)</b>	Notifies all listeners for this event.
<b>void fireTreeExpanded(TreePath path)</b>	Notifies all listeners for this event.
<b>void fireTreeWillCollapse(TreePath path)</b>	Notifies all listeners for this event.
<b>void fireTreeWillExpand(TreePath path)</b>	Notifies all listeners for this event.
<b>protected void fireValueChanged(TreeSelectionEvent e)</b>	Fires a “value changed” event.
<b>AccessibleContext getAccessibleContext()</b>	Gets the <b>AccessibleContext</b> .
<b>TreePath getAnchorSelectionPath ()</b>	Returns the path identified as the anchor.
<b>TreeCellEditor getCellEditor()</b>	Gets the editor used to edit entries.
<b>TreeCellRenderer getCellRenderer()</b>	Gets the current <b>TreeCellRenderer</b> that’s rendering each cell.
<b>TreePath getClosestPathForLocation(int x, int y)</b>	Gets the path to the node that’s closest to x,y.
<b>int getClosestRowForLocation(int x, int y)</b>	Gets the row of the node that’s closest to x,y.
<b>protected static TreeModel getDefaultTreeModel()</b>	Creates and returns a sample <b>TreeModel</b> .
<b>protected Enumeration getDescendantToggledPaths(TreePath parent)</b>	Gets an enumeration of <b>TreePaths</b> that have been expanded and are descendants of <b>parent</b> .
<b>TreePath getEditingPath()</b>	Gets the path to the element that’s currently being edited.
<b>Enumeration getExpandedDescendants(TreePath parent)</b>	Gets an enumeration of the descendants of <b>path</b> that are currently expanded.
<b>boolean getExpandsSelectedPaths ()</b>	Returns the <b>expandsSelectedPaths</b> property.
<b>boolean getInvokesStopCellEditing()</b>	Gets the indicator that tells what happens when editing is interrupted.

*(continued)*

**Table 1.9** Methods of the **JTree** class (*continued*).

Method	Does This
<b>Object</b> <code>getLastSelectedPathComponent()</code>	Gets the last path component in the first node of the current selection.
<b>TreePath</b> <code>getLeadSelectionPath()</code>	Gets the path of the last node added to the selection.
<b>int</b> <code>getLeadSelectionRow()</code>	Gets the row index of the last node added to the selection.
<b>int</b> <code>getMaxSelectionRow()</code>	Gets the last selected row.
<b>int</b> <code>getMinSelectionRow()</code>	Gets the first selected row.
<b>TreeModel</b> <code>getModel()</code>	Gets the <b>TreeModel</b> that's providing the data.
<b>protected TreePath[]</b> <b>getPathBetweenRows(int index0, int index1)</b>	Gets <b>JTreePath</b> instances representing the path between <b>index0</b> and <b>index1</b> , inclusive.
<b>Rectangle</b> <code>getPathBounds(TreePath path)</code>	Gets the rectangle that the given node will be drawn into.
<b>TreePath</b> <code>getPathForLocation(int x, int y)</code>	Gets the path for the node at the given location.
<b>TreePath</b> <code>getPathForRow(int row)</code>	Gets the path for the given row.
<b>Dimension</b> <b>getPreferredScrollableViewportSize()</b>	Gets the preferred display size of a <b>JTree</b> .
<b>Rectangle</b> <code>getRowBounds(int row)</code>	Gets the rectangle that the node at the given row is drawn in.
<b>int</b> <code>getRowCount()</code>	Gets the total number of rows.
<b>int</b> <code>getRowForLocation(int x, int y)</code>	Gets the row for the given location.
<b>int</b> <code>getRowForPath(TreePath path)</code>	Gets the row that displays the node identified by the given path.
<b>int</b> <code>getRowHeight()</code>	Gets the height of each row.
<b>int</b> <code>getScrollableBlockIncrement(Rectangle visibleRect, int orientation, int direction)</code>	Gets the block increment.
<b>boolean</b> <code>getScrollableTracksViewportHeight()</code>	False indicates that the height of the viewport does not determine the height of the table.
<b>boolean</b> <code>getScrollableTracksViewportWidth()</code>	False indicates that the width of the viewport does not determine the width of the table.
<b>int</b> <code>getScrollableUnitIncrement(Rectangle visibleRect, int orientation, int direction)</code>	Gets the amount to increment when scrolling.
<b>boolean</b> <code>getScrollsOnExpand()</code>	Returns True if the tree will scroll to show previously hidden children.
<b>int</b> <code>getSelectionCount()</code>	Gets the number of nodes selected.

(continued)

Table 1.9 Methods of the **JTree** class (*continued*).

Method	Does This
<b>TreeSelectionModel</b> <b>getSelectionModel()</b>	Gets the model for selections.
<b>TreePath</b> <b>getSelectionPath()</b>	Gets the path to the first selected node.
<b>TreePath[]</b> <b>getSelectionPaths()</b>	Gets the paths of all selected values.
<b>int[]</b> <b>getSelectionRows()</b>	Gets all the currently selected rows.
<b>boolean</b> <b>getShowsRootHandles()</b>	Returns True if handles for the root nodes are displayed.
<b>int</b> <b>getToggleClickCount()</b>	Returns the number of mouse clicks needed to expand or close a node.
<b>String</b> <b>getToolTipText(MouseEvent event)</b>	Overrides <b>JComponent</b> 's <b>getToolTipText</b> method in order to allow the renderer's tooltips to be used.
<b>TreeUI</b> <b>getUI()</b>	Gets the look-and-feel object that draws this component.
<b>String</b> <b>getUIClassID()</b>	Gets the name of the look-and-feel class that draws this component.
<b>int</b> <b>getVisibleRowCount()</b>	Gets the number of rows that are displayed in the display area.
<b>boolean</b> <b>hasBeenExpanded(TreePath path)</b>	Returns True if the node identified by the path has ever been expanded.
<b>boolean</b> <b>isCollapsed(int row)</b>	Returns True if the node at the given display row is collapsed.
<b>boolean</b> <b>isCollapsed(TreePath path)</b>	Returns True if the value identified by <b>path</b> is currently collapsed.
<b>boolean</b> <b>isEditable()</b>	Returns True if the tree is editable.
<b>boolean</b> <b>isEditing()</b>	Returns True if the tree is being edited.
<b>boolean</b> <b>isExpanded(int row)</b>	Returns True if the node at the given display row is currently expanded.
<b>boolean</b> <b>isExpanded(TreePath path)</b>	Returns True if the node identified by the path is currently expanded.
<b>boolean</b> <b>isFixedRowHeight()</b>	Returns True if the height of each display row is a fixed size.
<b>boolean</b> <b>isLargeModel()</b>	Returns True if the tree is configured for a large model.
<b>boolean</b> <b>isPathEditable(TreePath path)</b>	Gets the value in <b>isEditable</b> .
<b>boolean</b> <b>isPathSelected(TreePath path)</b>	Returns True if the item identified by the path is currently selected.
<b>boolean</b> <b>isRootVisible()</b>	Returns True if the root node of the tree is displayed.

*(continued)*

Table 1.9 Methods of the **JTree** class (*continued*).

Method	Does This
<b>boolean</b> <code>isRowSelected(int row)</code>	Returns True if the node identified by <b>row</b> is selected.
<b>boolean</b> <code>isSelectionEmpty()</code>	Returns True if the selection is currently empty.
<b>boolean</b> <code>isVisible(TreePath path)</code>	Returns True if the value identified by <b>path</b> is currently viewable.
<b>void</b> <code>makeVisible(TreePath path)</code>	Ensures that the node identified by <b>path</b> is currently viewable.
<b>protected String</b> <code> paramString()</code>	Gets a string representation of this <b>JTree</b> .
<b>protected void</b> <b>removeDescendantToggledPaths</b> <b>(Enumeration toRemove)</b>	Removes any descendants of the <b>TreePaths</b> in <b>toRemove</b> that have been expanded.
<b>void</b> <code>removeSelectionInterval(int index0, int index1)</code>	Removes the nodes between <b>index0</b> and <b>index1</b> , inclusive, from the selection.
<b>void</b> <code>removeSelectionPath(TreePath path)</code>	Removes the node identified by the given path from the current selection.
<b>void</b> <code>removeSelectionPaths(TreePath[] paths)</code>	Removes the nodes identified by the given paths from the current selection.
<b>void</b> <code>removeSelectionRow(int row)</code>	Removes the path at the index row from the current selection.
<b>void</b> <code>removeSelectionRows(int[] rows)</code>	Removes the paths that are selected at each of the given rows.
<b>void</b> <code>removeTreeExpansionListener</code> <b>(TreeExpansionListener tel)</b>	Removes a listener for <b>TreeExpansion</b> events.
<b>void</b> <code>removeTreeSelectionListener</code> <b>(TreeSelectionListener tsl)</b>	Removes a <b>TreeSelection</b> listener.
<b>void</b> <code>removeTreeWillExpandListener</code> <b>(TreeWillExpandListener tel)</b>	Removes a listener for <b>TreeWillExpand</b> events.
<b>void</b> <code>scrollPathToVisible(TreePath path)</code>	Scrolls so that the node identified by <b>path</b> is displayed.
<b>void</b> <code>scrollRowToVisible(int row)</code>	Scrolls the item identified by <b>row</b> until it's displayed.
<b>void</b> <code>setAnchorSelectionPath (TreePath newPath)</code>	Sets the path identified as the anchor.
<b>void</b> <code>setCellEditor(TreeCellEditor cellEditor)</code>	Sets the cell editor.
<b>void</b> <code>setCellRenderer(TreeCellRenderer x)</code>	Sets the <b>TreeCellRenderer</b> that will be used to draw each cell.
<b>void</b> <code>setEditable(boolean flag)</code>	Determines whether the tree is editable.

*(continued)*

Table 1.9 Methods of the **JTree** class (*continued*).

Method	Does This
<b>protected void setExpandedState</b> <b>(TreePath path, boolean state)</b>	Sets the expanded state of the receiver.
<b>void setInvokesStopCellEditing</b> (boolean <b>newValue)</b>	Determines what happens when editing is interrupted.
<b>void setLargeModel</b> (boolean <b>newValue)</b>	Specifies whether the user interface should use a large model.
<b>void setLeadSelectionPath</b> ( <b>TreePath newPath</b> )	Sets the path identified as the lead.
<b>void setModel</b> ( <b>TreeModel</b> <b>newModel</b> )	Sets the <b>TreeModel</b> that will provide the data.
<b>void setRootVisible</b> (boolean <b>rootVisible</b> )	Determines whether the root node from the <b>TreeModel</b> is visible.
<b>void setRowHeight</b> (int <b>rowHeight</b> )	Sets the height of each cell.
<b>void setScrollsOnExpand</b> (boolean <b>newValue</b> )	Determines whether the tree should scroll when a node is expanded.
<b>void setSelectionInterval</b> (int <b>index0</b> , int <b>index1</b> )	Selects the nodes between <b>index0</b> and <b>index1</b> , inclusive.
<b>void setSelectionModel</b> ( <b>TreeSelectionModel</b> <b>selectionModel</b> )	Sets the tree's selection model.
<b>void setSelectionPath</b> ( <b>TreePath path</b> )	Selects the node identified by the given path.
<b>void setSelectionPaths</b> ( <b>TreePath[] paths</b> )	Selects the nodes identified by the given array of paths.
<b>void setSelectionRow</b> (int <b>row</b> )	Selects the node at the given row in the display.
<b>void setSelectionRows</b> (int[] <b>rows</b> )	Selects the nodes corresponding to each of the given rows in the display.
<b>void setShowsRootHandles</b> (boolean <b>newValue</b> )	Determines whether the node handles are to be displayed.
<b>void setToggleClickCount</b> (int <b>clickCount</b> )	Sets the number of mouse clicks before a node will expand or close.
<b>void setUI</b> ( <b>TreeUI ui</b> )	Sets the look-and-feel object that renders this component.
<b>void setVisibleRowCount</b> (int <b>newCount</b> )	Sets the number of rows that are to be displayed.
<b>void startEditingAtPath</b> ( <b>TreePath path</b> )	Selects the given path and starts editing.
<b>boolean stopEditing</b> ()	Ends the editing session.
<b>void treeDidChange</b> ()	Called when the tree has changed enough that it needs to resize the bounds.
<b>void updateUI</b> ()	Called by <b>UIManager</b> when the look and feel has changed.



You can add data to a tree in a variety of ways; in this chapter, I'll use a popular technique: creating a hashtable, populating it with data, and adding that hash to the tree. You can use **JTree** to display compound nodes (for example, nodes containing both a graphic icon and text) by subclassing **TreeCellRenderer** and using **setTreeCellRenderer**. To edit nodes, you can subclass **TreeCellEditor** and use **setTreeCellEditor**.

Here's a quick example. In this case, I just create a default tree and add it to a scroll pane:

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

/*
<APPLET
    CODE = tree.class
    WIDTH = 350
    HEIGHT = 280>
</APPLET>
*/

public class tree extends JApplet
{
    public void init()
    {
        JTree jtree = new JTree();

        getContentPane().add(new JScrollPane(jtree));
    }
}
```

This code is all that's needed to display a basic tree—the result of this code appears in Figure 1.5, where you can see that Swing has added some default data to the tree. This example appears as `tree.java` on the CD.

Note that trees aren't going to be much use unless you can add your own data to them—see the next solution for the details.

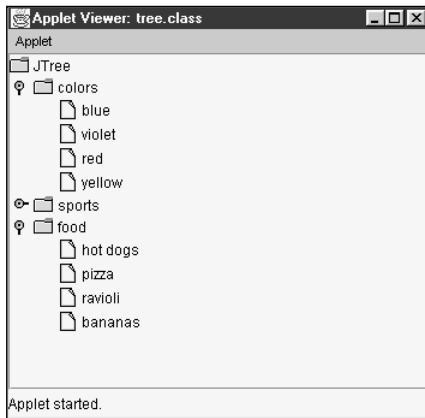


Figure 1.5 A basic tree.

## Adding Data to Trees

“Just one problem,” the Novice Programmer says, “I’ve created a Swing tree, but how can I add data to it?” “There are a number of ways to do it,” you say, “Pull up a chair and we’ll look at it.” The NP goes to get some coffee.

There are a number of ways to populate trees with data. For example, here are the various possibilities using the **JTree** constructors:

- **JTree()**—Creates a default tree with sample data
- **JTree(Hashtable value)**—Creates a tree from a hashtable
- **JTree(Object[] value)**—Creates a tree from an object array
- **JTree(TreeModel newModel)**—Creates a tree from a tree model
- **JTree(TreeNode root)**—Creates a tree with the given root node
- **JTree(Vector value)**—Creates a tree from a vector

Using a hashtable is an easy way to create a tree with folders and leaves, and I’ll use that technique here. You’ll learn about hashtables in depth later in this book; hashes let you store data that’s accessible with a text key, and you can populate them with the **put** method, which takes the key and value pairs you want to store in the hashes. Hashes are useful in constructing trees that can, themselves, contain hashes, which is how you create nodes with subnodes.

An example will make this more clear. In this example, I create a hash with individual data items and subhashes, just by creating a single hash to represent the

tree. When the hash has been built, all I have to do is pass it to the **JTree** constructor, like this:

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

/*
<APPLET
    CODE = treedata.class
    WIDTH = 350
    HEIGHT = 280>
</APPLET>
*/

public class treedata extends JApplet
{
    Hashtable hashtable = new Hashtable();
    Hashtable subhashtable = new Hashtable();
    Hashtable subsubhashtable = new Hashtable();

    String[] strings = new String[] {"Item 1", "Item 2",
        "Item 3", "Item 4", "Item 5"};

    public void init()
    {
        Container contentPane = getContentPane();

        hashtable.put("Items", strings);
        hashtable.put("Subitems", subhashtable);

        subhashtable.put("Items", strings);
        subhashtable.put("Item 1", new Integer(1));
        subhashtable.put("Item 2", new Integer(2));
        subhashtable.put("Item 3", new Integer(3));
        subhashtable.put("Items with subitems", subsubhashtable);

        subsubhashtable.put("Yet more subitems", strings);
        subsubhashtable.put("Item 1", new Integer(1));
        subsubhashtable.put("Item 2", new Integer(2));
        subsubhashtable.put("Item 3", new Integer(3));

        JTree hashTree = new JTree(hashtable);
```

```

        JScrollPane hashPane = new JScrollPane(hashTree);

        hashTree.expandPath(new TreePath(hashTree.getModel().getRoot()));

        contentPane.add(hashPane);
    }
}

```

The **JTree** class interprets the hashes passed to it and creates the corresponding tree structure, as you can see in Figure 1.6. That's all it takes to add data to trees in a complex data hierarchy. This example is `treedata.java` on the CD.

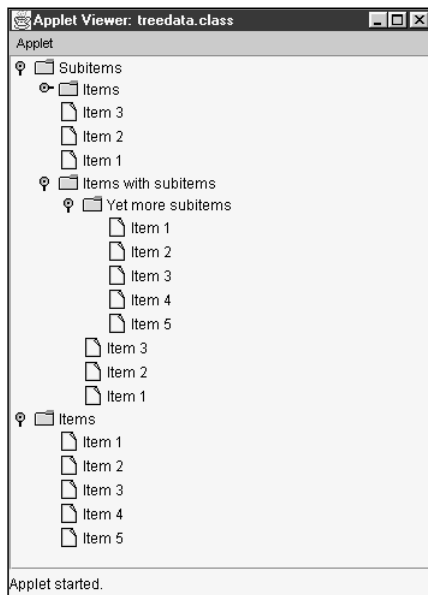


Figure 1.6 A tree populated with data.

## Handling Tree Events

One of the important aspects of working with trees is handling tree events, such as when the user clicks or double-clicks a node. To handle mouse events, you can add a **MouseListener** to a tree. Here's an example in which I indicate which node in the tree was clicked or double-clicked.

When the user clicks a node, I find which row in the tree was clicked with the **getRowForLocation** method. To find the actual node, I use the **getPathForRow** and **getLastPathComponent** methods. Here's the code:

```

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

/*
<APPLET
    CODE = treeevents.class
    WIDTH = 350
    HEIGHT = 280>
</APPLET>
*/

public class treeevents extends JApplet
{
    public void init()
    {
        JTree tree = new JTree();

        getContentPane().add(new JScrollPane(tree));

        tree.addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent e)
            {
                String outString = null;
                JTree jtree = (JTree)e.getSource();

                int clickedrow = jtree.getRowForLocation(e.getX(),
                    e.getY());

                if(clickedrow != -1) {
                    TreePath treepath = jtree.getPathForRow(clickedrow);
                    TreeNode treenode = (TreeNode)
                        treepath.getLastPathComponent();

                    outString = "Node " + treenode.toString();

                    if(e.getClickCount() == 1)
                    {
                        outString += " was single clicked.";
                    } else {
                        outString += " was double clicked.";
                    }
                }
            }
        });
    }
}

```

```

        showStatus(outString);
    }
}
});
}
}

```

You can see the results in Figure 1.7. When the user clicks or double-clicks any node, including folders, this applet will indicate which node was clicked or double-clicked in the status bar. Handling tree events such as this makes tree controls come alive and allows users to organize their data as they like. This example is `treeevents.java` on the CD.

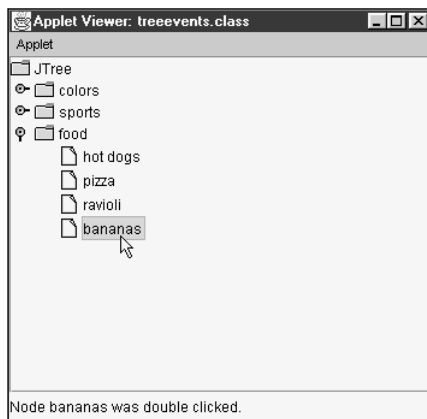


Figure 1.7 Handling tree events.

## Editing Tree Nodes

“Hmm,” says the Novice Programmer, “I’ve created that file-managing program the Big Boss wanted using a tree, but there’s a problem—users want to be able to change the names of files simply by editing a tree node. You can’t do that, can you?” “Sure you can,” you say. “Uh-oh,” the NP says, “guess I’ve got some code to write.”

To let the user edit the text in the nodes in a tree, you use the **setEditable** method. To allow the user to actually read the new text as edited, you can implement the **editingCanceled** and **editingStopped** methods of the **CellEditorListener** methods. If the user moves away from the edited node without pressing Enter, **editingCanceled** is called and the edit does not take effect; if the user does press Enter, the **editingStopped** method is called and the edit does take effect.

Here's an example in which I handle both the **editingCanceled** and **editingStopped** methods. When **editingCanceled** is called, I indicate that the editing was cancelled; when **editingStopped** is called, I display the new node text. Here's the code:

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

/*
<APPLET
    CODE = treeedit.class
    WIDTH = 350
    HEIGHT = 280>
</APPLET>
*/

public class treeedit extends JApplet implements CellEditorListener
{
    public void init()
    {
        JTree tree = new JTree();

        getContentPane().add(tree);

        tree.setEditable(true);

        tree.getCellEditor().addCellEditorListener(this);
    }

    public void editingCanceled(ChangeEvent e)
    {
        CellEditor editor = (CellEditor)e.getSource();

        showStatus("Change not made");
    }

    public void editingStopped(ChangeEvent e)
    {
        CellEditor editor = (CellEditor)e.getSource();

        showStatus("New text: " + (String)editor.getCellEditorValue());
    }
}
```

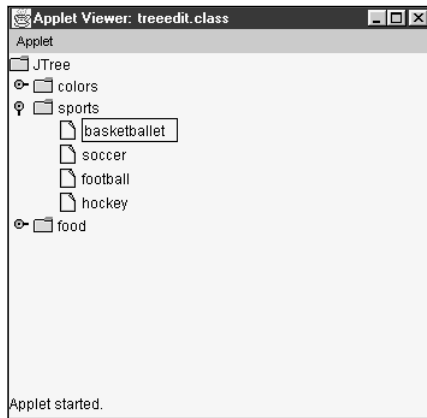


Figure 1.8 Editing a tree node.

The result of this code appears in Figure 1.8, and you can see in the figure that I'm changing the text "basketball" to "basketballet". This applet appears as `treedit.java` on the CD.



# eChapter 2

## Swing: Text Components

<i>If you need an immediate solution to:</i>	<i>See page:</i>
Creating Swing Text Components: The <b>JTextComponent</b> Class	e44
Creating Text Fields	e48
Setting Text Field Alignment	e49
Scrolling Text Fields	e51
Creating Password Fields	e54
Creating Text Areas	e57
Creating Editor Panes	e60
Using HTML in Editor Panes	e65
Adding Hyperlinks to Editor Panes	e67
Using RTF Files in Editor Panes	e70
Creating Text Panes	e72
Inserting Images and Controls into Text Panes	e74
Setting Text Pane Text Attributes	e76
Working with Sound in Applets	e79
Working with Sound in Applications	e80

# In Depth

In this chapter, we'll take a look at the Swing text components: text fields, password fields, text areas, editor panes, and text panes. All these components are descendants of the **JTextComponent** class, which we'll also take a look at here.

Working with Swing text components can get quite complex and involved. The **JTextPane** class is the top-of-the-line text component, and it's quite possible to get lost in a veritable forest of renderers and selection models when working with this class. The complete details on all the Swing text components would take many chapters, but you'll get a good introduction here.

## Text Fields

Text fields are the old standby of text components, and we've used them as long as we've used Swing. Text fields, as supported by the **JTextField** class, let the user view, enter, and edit a single line of text. You can do more with Swing text fields than you can with AWT text fields—for example, you can scroll a Swing text field programmatically. You'll see the possibilities in this chapter. All in all, text fields are relatively simple Swing text components; the more complex ones are coming up.

## Password Fields

Password fields are much like text fields where you've set the echo character so that each time the user types a character, the specified echo character appears instead of the actual typed character. Password fields mask what's really typed in case someone's looking over your shoulder as you type a password. Unlike in AWT, in Swing, password fields get their own control and class: the **JPasswordField** class. We'll take a look at how to work with this class in this chapter.

## Text Areas

Swing text areas constitute another relatively simple Swing text control. Text areas are much like text fields, except they're two dimensional and can display text in rows and columns. You can display the text with scrollbars as well as use word wrap—it's up to you. Text areas are popular controls when you have more than a single line of text to display; in fact, we've already used them in this book. We'll take an in-depth look at them here.

## Editor Panes

Editor panes provide you with a way of working with three kinds of text: plain, HTML, and rich-text format (RTF). You can load entire Web pages as well as RTF documents into editor panes (but note that the HTML support is by no means complete, at least not yet). We'll take a look at these options here.

## Text Panes

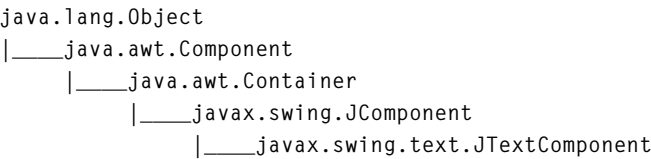
Text panes are the most sophisticated of the Swing text components; they're derived from the **JEditorPane** class and extend editor panes by letting you mark up text with various style attributes. Using text panes, you can create entirely new styles and apply them to paragraphs or to individual runs of characters in your text. (Programming this component can get a little complex.) You can also embed other controls and images in text panes, thus creating complex documents.

That's it for the overview of what's in this chapter. There's a lot coming up—in fact, Swing text components could, themselves, take up an entire book. It's time to turn to the “Immediate Solutions” section, starting with a look at the **JTextComponent** class.

# Immediate Solutions

## Creating Swing Text Components: The JTextComponent Class

Swing text components are built on the **JTextComponent** class, and it's worth taking a look at the fields, constructor, and methods of this class because the Swing text components inherit them all. Here's the inheritance diagram for the **JTextComponent** class:



You'll find the fields for the **JTextComponent** class in Table 2.1, its constructor in Table 2.2, and its methods in Table 2.3.

Table 2.1 Fields of the **JTextComponent** class.

Field	Does This
<b>static String</b> DEFAULT_KEYMAP	The default keymap
<b>static String</b> FOCUS_ACCELERATOR_KEY	The bound property name for the focus accelerator

Table 2.2 The constructor of the **JTextComponent** class.

Constructor	Does This
<b>JTextComponent()</b>	Constructs a new <b>JTextComponent</b> object.

Table 2.3 Methods of the **JTextComponent** class.

Method	Does This
<b>void addCaretListener(CaretListener listener)</b>	Adds a caret listener.
<b>void addInputMethodListener(InputMethodListener l)</b>	Adds the given input method listener.
<b>static Keymap addKeymap(String nm, Keymap parent)</b>	Adds a new keymap into the keymap hierarchy.

(continued)

Table 2.3 Methods of the **JTextComponent** class (*continued*).

Method	Does This
<b>void copy()</b>	Copies the currently selected range in the associated text model to the system Clipboard.
<b>void cut()</b>	Copies the currently selected range in the associated text model to the system Clipboard, removing the contents from the model.
<b>protected void fireCaretUpdate(CaretEvent e)</b>	Notifies all listeners.
<b>AccessibleContext getAccessibleContext()</b>	Gets the <b>AccessibleContext</b> object.
<b>Action[] getActions()</b>	Gets the command list for the editor.
<b>Caret getCaret()</b>	Gets the caret that allows text-oriented navigation over the view.
<b>Color getCaretColor()</b>	Gets the current color used to draw the caret.
<b>int getCaretPosition()</b>	Gets the position of the text-insertion caret for the text component.
<b>Color getDisabledTextColor()</b>	Gets the current color used to render the selected text.
<b>Document getDocument()</b>	Gets the model associated with the editor.
<b>char getFocusAccelerator()</b>	Gets the key accelerator that will cause the receiving text component to get the focus.
<b>Highlighter getHighlighter()</b>	Gets the object responsible for making highlights.
<b>InputMethodRequests getInputMethodRequests()</b>	Gets the input method request handler that supports requests from input methods for this component.
<b>Keymap getKeymap()</b>	Gets the keymap currently active in this text component.
<b>static Keymap getKeymap(String nm)</b>	Gets a named keymap previously added to the document.
<b>Insets getMargin()</b>	Gets the margin between the text component's border and its text.
<b>Dimension getPreferredSize()</b>	Gets the preferred size of the viewport for a view component.
<b>int getScrollableBlockIncrement(Rectangle visibleRect, int orientation, int direction)</b>	Gets the scroll increment that will completely expose one block of rows or columns.
<b>boolean getScrollableTracksViewportHeight()</b>	Returns True if a viewport should always force the height of this <b>Scrollable</b> object to match the height of the viewport.

*(continued)*

Table 2.3 Methods of the **JTextComponent** class (*continued*).

Method	Does This
<b>boolean getScrollableTracksViewportWidth()</b>	Returns True if a viewport should always force the width of this <b>Scrollable</b> object to match the width of the viewport.
<b>int getScrollableUnitIncrement(Rectangle visibleRect, int orientation, int direction)</b>	The scroll increment that will completely expose one new row or column, depending on the value of <b>orientation</b> .
<b>String getSelectedText()</b>	Gets the selected text contained in this <b>TextComponent</b> .
<b>Color getSelectedTextColor()</b>	Gets the current color used to render the selected text.
<b>Color getSelectionColor()</b>	Gets the current color used to render the selection.
<b>int getSelectionEnd()</b>	Gets the selected text's end position.
<b>int getSelectionStart()</b>	Gets the selected text's start position.
<b>String getText()</b>	Gets the text contained in this <b>TextComponent</b> .
<b>String getText(int offs, int len)</b>	Gets a portion of the text represented by the component.
<b>TextUI getUI()</b>	Gets the user-interface factory for this text-oriented editor.
<b>boolean isEditable()</b>	Gets the boolean value indicating whether this <b>TextComponent</b> is editable.
<b>boolean isFocusTraversable()</b>	Returns True if the focus can be traversed.
<b>boolean isOpaque()</b>	Returns True if this component is completely opaque.
<b>static void loadKeymap(Keymap map, JTextComponent.KeyBinding[] bindings, Action[] actions)</b>	Loads a keymap with bindings.
<b>Rectangle modelToView(int pos)</b>	Converts the given location in the model to the view coordinate system.
<b>void moveCaretPosition(int pos)</b>	Moves the caret to a new position.
<b>protected String paramString()</b>	Gets a string representation of this <b>JTextComponent</b> .
<b>void paste()</b>	Copies the contents of the system Clipboard into the associated text model.
<b>protected void processInputMethodEvent(InputMethodEvent e)</b>	Processes input method events occurring in this component.
<b>void read(Reader in, Object desc)</b>	Initializes from a stream.

*(continued)*

**Table 2.3 Methods of the `JTextComponent` class (continued).**

Method	Does This
<b><code>void removeCaretListener(CaretListener listener)</code></b>	Removes a caret listener.
<b><code>static Keymap removeKeymap(String nm)</code></b>	Removes a named keymap previously added to the document.
<b><code>void removeNotify()</code></b>	Notifies this component that it has been removed from its container.
<b><code>void replaceSelection(String content)</code></b>	Replaces the currently selected content with new content represented by the given string.
<b><code>void select(int selectionStart, int selectionEnd)</code></b>	Selects the text found between the given start and end locations.
<b><code>void selectAll()</code></b>	Selects all the text in the <b><code>TextComponent</code></b> .
<b><code>void setCaret(Caret c)</code></b>	Sets the caret to be used.
<b><code>void setCaretColor(Color c)</code></b>	Sets the current color used to render the caret.
<b><code>void setCaretPosition(int position)</code></b>	Sets the position of the text-insertion caret for the <b><code>TextComponent</code></b> .
<b><code>void setDisabledTextColor(Color c)</code></b>	Sets the current color used to render the disabled text.
<b><code>void setDocument(Document doc)</code></b>	Associates the editor with a text document.
<b><code>void setEditable(boolean b)</code></b>	Sets the given boolean value to indicate whether this <b><code>TextComponent</code></b> should be editable.
<b><code>void setFocusAccelerator(char aKey)</code></b>	Sets the key accelerator that will cause the receiving text component to get the focus.
<b><code>void setHighlighter(Highlighter h)</code></b>	Sets the highlighter to be used.
<b><code>void setKeymap(Keymap map)</code></b>	Sets the keymap to use for binding events to actions.
<b><code>void setMargin(Insets m)</code></b>	Sets margin space between the text component's border and its text.
<b><code>void setSelectedTextColor(Color c)</code></b>	Sets the current color used to draw the selected text.
<b><code>void setSelectionColor(Color c)</code></b>	Sets the current color used to draw the selection.
<b><code>void setSelectionEnd(int selectionEnd)</code></b>	Sets the selection end to the given position.
<b><code>void setSelectionStart(int selectionStart)</code></b>	Sets the selection start to the given position.
<b><code>void setText(String t)</code></b>	Sets the text of this <b><code>TextComponent</code></b> to the given text.
<b><code>void setUI(TextUI ui)</code></b>	Sets the user-interface factory for this text-oriented editor.
<b><code>void updateUI()</code></b>	Reloads the <b><code>UI</code></b> object.
<b><code>int viewToModel(Point pt)</code></b>	Converts the given location in the view-coordinate system to the nearest location in the model.
<b><code>void write(Writer out)</code></b>	Stores the contents of the model using the given stream.

## Creating Text Fields

The text field is the basic Swing text control. In fact, we've already put text fields to work as far back as Chapter 14. Here's the inheritance diagram for the **JTextField** class:

```

java.lang.Object
|____java.awt.Component
      |____java.awt.Container
            |____javax.swing.JComponent
                  |____javax.swing.text.JTextComponent
                        |____javax.swing.JTextField
  
```

In Chapter 14, you'll find the constructors of the **JTextField** class in Table 14.6 and its methods in Table 14.7. Here's a short example that puts a text field to work:

```

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

/*
<APPLET
    CODE=textfield.class
    WIDTH=300
    HEIGHT=200 >
</APPLET>
*/

public class textfield extends JApplet {
    JTextField text = new JTextField(20);

    public void init()
    {
        Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());
        contentPane.add(text);
        text.setText("Hello from Swing!");
    }
}
  
```

We'll take a look at some of the capabilities of Swing text fields over the next couple of solutions.



**Related solution:**

Creating Scroll Panes

**Found on page:**

Chapter 15

## Setting Text Field Alignment

“Hmm,” says the Novice Programmer, “I got a strange memo from the Big Boss. It says, ‘We’re trying something new. Left-align all your text.’ Is it some political thing?” You smile and say, “No, it’s all about your text fields.”

You can set the alignment of text fields with the **setHorizontalAlignment** method (there is no **setVerticalAlignment** method for text fields). Here’s an example that shows the possibilities—right, left, and center. In this case, I let the user click a button to set the alignment of the text in a text field; then, I use the **setHorizontalAlignment** method to justify the text to match. Here’s the code:

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

/*
<APPLET
    CODE = textfieldalign.class
    WIDTH = 350
    HEIGHT = 280 >
</APPLET>
*/

public class textfieldalign extends JApplet
{
    JTextField jtextfield = new JTextField("Hello from Swing!");
    JButton jbutton1, jbutton2, jbutton3;

    public void init()
    {
        Container contentPane = getContentPane();

        jtextfield.setColumns(30);

        contentPane.setLayout(new FlowLayout());
        contentPane.add(new buttonpanel());
        contentPane.add(jtextfield);
    }
}
```

```

class buttonpanel extends JPanel implements ActionListener
{

    public buttonpanel()
    {

        jbutton1 = new JButton("Left");
        jbutton2 = new JButton("Right");
        jbutton3 = new JButton("Center");

        jbutton1.addActionListener(this);
        jbutton2.addActionListener(this);
        jbutton3.addActionListener(this);

        setLayout(new FlowLayout());

        add(new JLabel("Alignment"));
        add(jbutton1);
        add(jbutton2);
        add(jbutton3);

    }

    public void actionPerformed(ActionEvent e)
    {
        if(e.getSource() == jbutton1) {
            jtextfield.setHorizontalAlignment(JTextField.LEFT);
        } else if(e.getSource() == jbutton2) {
            jtextfield.setHorizontalAlignment(JTextField.RIGHT);
        } else if(e.getSource() == jbutton3) {
            jtextfield.setHorizontalAlignment(JTextField.CENTER);
        }
    }
}

```

You can see the result of this code in Figure 2.1, where the user can set the alignment of the text in the text field just by clicking a button. This example is `textfieldalign.java` on the CD accompanying this book.

---

**TIP:** Note that I set the number of columns using **setColumns** in the text field in this example so that you can see horizontal space around the actual text. There are two things to note here: First, the size of a column in a text field is the width of the letter *m*, so **setColumn** does not actually set the number of characters visible. Second, **setColumn** sets only the text field's preferred size.

---

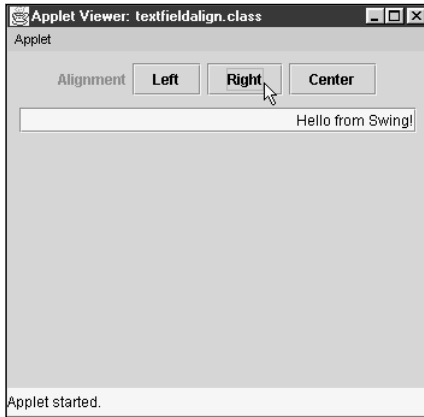


Figure 2.1 Aligning text in a text field.

## Scrolling Text Fields

“I’ve got a problem,” the Novice Programmer says, “I’m writing my novel in a text field, and I really hate having to move around in all that text using the keyboard.” “OK,” you say, “you can scroll the text field, but you really should be using a text area when working with a lot of text.” The NP says, “Tell me more!”

Swing text fields implement the **Scrollable** interface, which means you can scroll them under programmatic command or inside scroll panes. Here’s an example showing how that works. In this case, I use the text field’s **getHorizontalVisibility** method to get an object that implements the **BoundedRangeModel** interface, and I pass that object to the constructor of a slider, implementing scrolling directly, without a scroll pane. When the user adjusts the slider, I use the text field’s **setScrollOffset** method to scroll the text field. First, however, take a look at the methods of the **BoundedRangeModel** interface, which is specially designed to make scrolling controls easier, in Table 2.4.

Table 2.4 Methods of the **BoundedRangeModel** interface.

Method	Does This
<b>void addChangeListener(ChangeListener x)</b>	Adds a <b>ChangeListener</b> .
<b>int getExtent()</b>	Gets the model’s extent (that is, the length of the inner range that begins at the model’s value).
<b>int getMaximum()</b>	Gets the model’s maximum value.
<b>int getMinimum()</b>	Gets the model’s minimum acceptable value.

(continued)

**Table 2.4** Methods of the **BoundedRangeModel** interface (*continued*).

Method	Does This
<b>int getValue()</b>	Gets the model's current value.
<b>boolean getValuesAdjusting()</b>	Returns True if the current changes to the <b>value</b> property are part of a sequence of changes.
<b>void removeChangeListener(ChangeListener x)</b>	Removes a <b>ChangeListener</b> from the model's listener list.
<b>void setExtent(int newExtent)</b>	Sets the model's extent.
<b>void setMaximum(int newMaximum)</b>	Sets the model's maximum value to <b>newMaximum</b> .
<b>void setMinimum(int newMinimum)</b>	Sets the model's minimum value to <b>newMinimum</b> .
<b>void setRangeProperties(int value, int extent, int min, int max, boolean adjusting)</b>	Sets the model's data with a single call.
<b>void setValue(int newValue)</b>	Sets the model's current value to <b>newValue</b> if <b>newValue</b> satisfies the model's constraints.
<b>void setValuesAdjusting(boolean b)</b>	Indicates that changes to the value of the model should be considered a single event.

Here's the actual code. In this case, I cram a long string into a short text field, and I connect the text field to a slider, as outlined at the beginning of this solution:

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

/*
<APPLET
    CODE = textfieldscroll.class
    WIDTH = 350
    HEIGHT = 280>
</APPLET>
*/

public class textfieldscroll extends JApplet
{
    private JTextField jtextfield = new JTextField(
        "Here is a pretty long string for one text field.", 12);

    public void init()
    {
```

```

        Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());
        contentPane.add(new jsliderpanel());
        contentPane.add(jtextfield);
    }

    class jsliderpanel extends JPanel
    {
        JSlider jslider = new
            JSlider(jtextfield.getHorizontalVisibility());

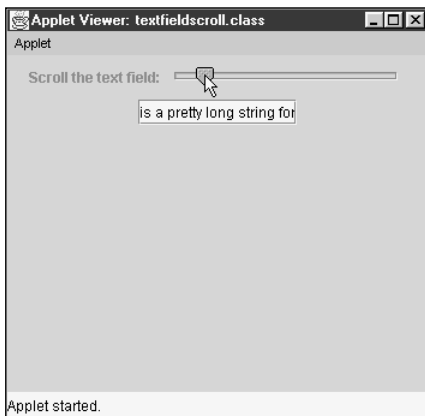
        public jsliderpanel() {
            add(new JLabel("Scroll the text field:"));

            add(jslider);

            jslider.addChangeListener(new ChangeListener() {
                public void stateChanged(ChangeEvent e) {
                    jtextfield.setScrollOffset(jslider.getValue());
                }
            });
        }
    }
}

```

You can see the result in Figure 2.2. As you see in the figure, a slider and a text field appear in this applet, and you can scroll the text in the text field using the slider. That's all there is to this applet. You'll find it as `textfieldscroll.java` on the CD.



**Figure 2.2** Scrolling a text field.

## Creating Password Fields

“Help!” cries a voice over the phone, and you decide it must be the Novice Programmer. “What’s wrong, NP?” you ask. “That darn Johnson was standing behind me, reading my password as I was typing!” the NP says. “Well,” you say, “change your passwords to new ones—and switch to password fields instead of text fields.” “Thanks,” says the NP, relieved.

Swing includes a special control for entering passwords—the **JPasswordField** control. You can set the echo character that appears in this control each time the user types a character so that the typed password is not actually visible. In addition, copying to the Clipboard is disabled for this control. This way, someone can’t just copy and paste your password. Here’s the inheritance diagram for **JPasswordField**:

```

java.lang.Object
|____java.awt.Component
      |____java.awt.Container
            |____javax.swing.JComponent
                  |____javax.swing.text.JTextComponent
                        |____javax.swing.JTextField
                              |____javax.swing.JPasswordField

```

You’ll find the constructors of the **JPasswordField** class in Table 2.5 and its methods in Table 2.6.

**Table 2.5 Constructors of the JPasswordField class.**

Constructor	Does This
<b>JPasswordField()</b>	Constructs a new <b>JPasswordField</b> .
<b>JPasswordField(Document doc, String txt, int columns)</b>	Constructs a new <b>JPasswordField</b> that uses the given text storage model and the given number of columns.
<b>JPasswordField(int columns)</b>	Constructs a new, empty <b>JPasswordField</b> with the given number of columns.
<b>JPasswordField(String text)</b>	Constructs a new <b>JPasswordField</b> initialized with the given text.
<b>JPasswordField(String text, int columns)</b>	Constructs a new <b>JPasswordField</b> initialized with the given text and columns.

Table 2.6 Methods of the **JPasswordField** class.

Method	Does This
<b>void copy()</b>	Copies the selected range in the associated text model to the system Clipboard.
<b>void cut()</b>	Copies the selected range in the associated text model to the system Clipboard, removing the contents from the model.
<b>boolean echoCharIsSet()</b>	Returns True if this <b>JPasswordField</b> has a character set for echoing.
<b>AccessibleContext getAccessibleContext()</b>	Gets the <b>AccessibleContext</b> .
<b>char getEchoChar()</b>	Gets the character to be used for echoing.
<b>char[] getPassword()</b>	Gets the text contained in this <b>TextComponent</b> .
<b>String getText()</b>	Deprecated. Replaced by <b>getPassword()</b> .
<b>String getText(int offs, int len)</b>	Deprecated. Replaced by <b>getPassword()</b> .
<b>String getUIClassID()</b>	Gets the name of the look-and-feel class that renders this component.
<b>protected String paramString()</b>	Gets a string representation of this <b>JPasswordField</b> .
<b>void setEchoChar(char c)</b>	Sets the echo character for this <b>JPasswordField</b> .

You use the **setEchoChar** method to set the echo character in a password field, and you can use the **getPassword** method to read the typed password. The **getPassword** method returns a character array, but you can transform that array into a **String** object by passing it to the **String** class's constructor.

Here's an example in which the actual password is "open sesame," and if the user types this password and presses Enter, the applet will display "Correct" in its status bar; otherwise, it'll display "Incorrect." Here's the code (note that to capture Enter press events, I've added an action listener to the password control):

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

/*
<APPLET
    CODE = password.class
    WIDTH = 350
    HEIGHT = 280 >
</APPLET>
*/

public class password extends JApplet
```

```

{
    String correctpassword = "open sesame";
    JPasswordField jpasswordfield = new JPasswordField(10);

    public void init()
    {
        Container contentPane = getContentPane();

        contentPane.setLayout(new FlowLayout());
        contentPane.add(new JLabel("Type your password: "));
        contentPane.add(jpasswordfield);

        jpasswordfield.setEchoChar('*');

        jpasswordfield.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                String input = new String(jpasswordfield.getPassword());

                if(correctpassword.equals(input))
                    showStatus("Correct");
                else
                    showStatus("Incorrect");
            }
        });
    }
}

```

The result of this code appears in Figure 2.3. As you can see in the figure, the password field displays the echo character and checks the password when the user presses Enter. This example is `password.java` on the CD.

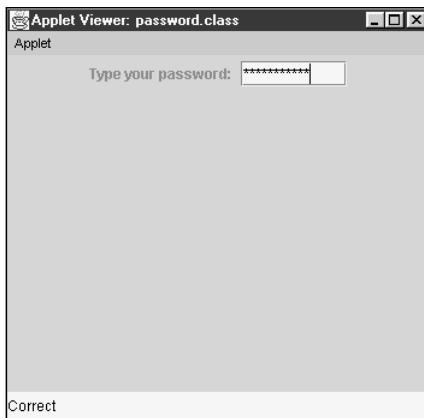


Figure 2.3 Using a password field.



# Creating Text Areas

“Nope,” the Novice Programmer says, “there’s just no way I can write my novel in a Swing text field. I guess I’ll have to use a word processor.” “Or,” you say, “you can use a Swing text area, which is perfect for longer documents.” “Tell me more!” says the NP.

The text area control is a simple control that’s really little more than a text field that lets you arrange text in both rows and columns (each column is the width of the letter *m* in the current font). Here’s the inheritance diagram for the **JTextArea** class:

```

java.lang.Object
|____java.awt.Component
      |____java.awt.Container
            |____javax.swing.JComponent
                  |____javax.swing.text.JTextComponent
                        |____javax.swing.JTextArea
    
```

You’ll find the constructors of the **JTextArea** class in Table 2.7 and its methods in Table 2.8.

**Table 2.7 Constructors of the JTextArea class.**

Constructor	Does This
<b>JTextArea()</b>	Constructs a new <b>JTextArea</b> .
<b>JTextArea(Document doc)</b>	Constructs a new <b>JTextArea</b> with the given document model.
<b>JTextArea(Document doc, String text, int rows, int columns)</b>	Constructs a new <b>JTextArea</b> with the given number of rows and columns as well as the given model.
<b>JTextArea(int rows, int columns)</b>	Constructs a new <b>JTextArea</b> with the given number of rows and columns.
<b>JTextArea(String text)</b>	Constructs a new <b>JTextArea</b> with the given text displayed.
<b>JTextArea(String text, int rows, int columns)</b>	Constructs a new <b>JTextArea</b> with the given text and number of rows and columns.

**Table 2.8 Methods of the JTextArea class.**

Method	Does This
<b>void append(String str)</b>	Appends the given text to the end of the document.
<b>protected Document createDefaultModel()</b>	Constructs the default implementation of the model to be used at construction time.
<b>AccessibleContext getAccessibleContext()</b>	Gets the <b>AccessibleContext</b> .

(continued)

Table 2.8 Methods of the **JTextArea** class (*continued*).

Method	Does This
<b>int</b> <code>getColumns()</code>	Gets the number of columns in the text area.
<b>protected int</b> <code>getColumnWidth()</code>	Gets column width.
<b>int</b> <code>getLineCount()</code>	Determines the number of lines contained in the area.
<b>int</b> <code>getLineEndOffset(int line)</code>	Determines the offset of the end of the given line.
<b>int</b> <code>getLineOfOffset(int offset)</code>	Translates an offset into the component's text into a line number.
<b>int</b> <code>getLineStartOffset(int line)</code>	Determines the offset of the start of the given line.
<b>boolean</b> <code>getLineWrap()</code>	Gets the line-wrapping policy of the text area.
<b>Dimension</b> <b>getPreferredScrollableViewportSize()</b>	Gets the preferred size of the viewport if this text area is embedded in a <b>JScrollPane</b> .
<b>Dimension</b> <code>getPreferredSize()</code>	Gets the preferred size of the text area.
<b>protected int</b> <code>getRowHeight()</code>	Gets the height of a row.
<b>int</b> <code>getRows()</code>	Gets the number of rows in the text area.
<b>boolean</b> <b>getScrollableTracksViewportWidth()</b>	Returns True if a viewport should always force the width of this <b>Scrollable</b> object to match the width of the viewport.
<b>int</b> <code>getScrollableUnitIncrement(Rectangle visibleRect, int orientation, int direction)</code>	Gets the scroll increment that will completely expose one new row or column.
<b>int</b> <code>getTabSize()</code>	Gets the number of characters used to expand tabs.
<b>String</b> <code>getUIClassID()</code>	Gets the class ID for the UI.
<b>boolean</b> <code>getWrapStyleWord()</code>	Gets the style of wrapping used if the text area wraps lines.
<b>void</b> <code>insert(String str, int pos)</code>	Inserts the given text at the given position.
<b>boolean</b> <code>isManagingFocus()</code>	Turns off Tab navigation when the focus is gained.
<b>protected String</b> <code> paramString()</code>	Gets a string representation of this text area.
<b>protected void</b> <b>processKeyEvent(KeyEvent e)</b>	Makes sure that Tab and Shift+Tab events get consumed so that AWT doesn't attempt focus traversal.
<b>void</b> <code>replaceRange(String str, int start, int end)</code>	Replaces text from the indicated start-to-end position with the new text given.
<b>void</b> <code>setColumns(int columns)</code>	Sets the number of columns.
<b>void</b> <code>setFont(Font f)</code>	Sets the current font.
<b>void</b> <code>setLineWrap(boolean wrap)</code>	Sets the line-wrapping policy of the text area.
<b>void</b> <code>setRows(int rows)</code>	Sets the number of rows.
<b>void</b> <code>setTabSize(int size)</code>	Sets the number of characters to expand tabs to.
<b>void</b> <code>setWrapStyleWord(boolean word)</code>	Sets the style of wrapping used if the text area wraps lines.

When you create a Swing text area, you can specify the number of rows and columns it has, and you can also use the **setRows** and **setColumns** methods to set its dimensions on the fly. The AWT text area control supports scrolling directly, but, like other Swing components, the **JTextArea** control does not. It instead implements the **Scrollable** interface, which means you can use text areas inside scroll panes. You can also specify how Swing text areas handle word wrap with methods such as **setWrapStyleWord** and **setLineWrap**. You can use the **setText** and **append** methods to place text in a text area. Here's one more thing to note—you can use **TextEvent** objects and the **TextListener** interface to handle editing in AWT text areas, but **JTextArea** adds another level of abstraction. This means you have to use **DocumentEvent** objects and the **DocumentListener** interface with the **JTextField**'s model if you want to handle editing events here.

Here's an example in which I create a new Swing text area with 5 rows and 20 columns, set the font, enable editing and word wrap, and place the text area into a scroll pane. To show off the text area's capabilities, I embed some newline (**\n**) characters into the displayed text to skip to the next lines:

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

/*
<APPLET
    CODE = textarea.class
    WIDTH = 350
    HEIGHT = 280>
</APPLET>
*/

public class textarea extends JApplet
{
    private JTextArea jtextarea = new JTextArea("Hello\nfrom\nSwing!",
        5, 20);

    public void init()
    {
        Container contentPane = getContentPane();

        jtextarea.setWrapStyleWord(true);
        jtextarea.setEditable(true);
        jtextarea.setFont(new Font("Times-Roman", Font.BOLD, 10));

        contentPane.setLayout(new FlowLayout());
        contentPane.add(new JScrollPane(jtextarea));
    }
}
```

The result of this code appears in Figure 2.4. You can see the text area with its text broken over three lines. That's all there is to it. This example appears as `textarea.java` on the CD.

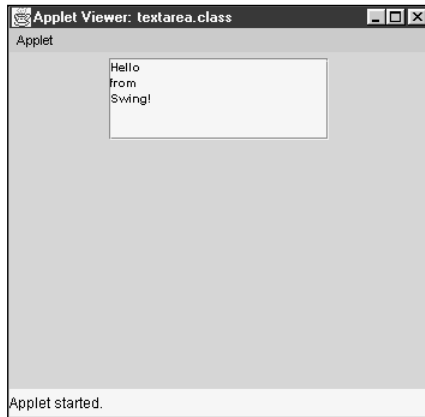


Figure 2.4 A basic Swing text area.

## Creating Editor Panes

“Jeez,” says the Novice Programmer, “the Big Boss wants me to create an entire Web browser in Java! How the heck am I going to do that?” “Well,” you smile, “a good start is to use the Swing editor pane, which already supports HTML 3.2 documents.” “Wow!” says the NP, “I feel better already!”

Editor panes are text components that use implementations of the **EditorKit** class to allow sophisticated editing. You can create your own editor kits, but by default, the following types of MIME (Multipurpose Internet Mail Extension) content are known:

- *Text/plain*—Plain text, which is the default type. The kit used in this case is an extension of **DefaultEditorKit**, which produces a wrapped plain-text view.
- *Text/html*—HTML text. The editor kit used in this case is the class **javax.swing.text.html.HTMLEditorKit**, which supports HTML 3.2.
- *Text/rtf*—RTF text. The editor kit used in this case is the class **javax.swing.text.rtf.RTFEditorKit**, which provides a limited support of the rich-text format (RTF).

Here's the inheritance diagram for **JEditorPane**:

```

java.lang.Object
|____java.awt.Component
      |____java.awt.Container
            |____javax.swing.JComponent
                  |____javax.swing.text.JTextComponent
                        |____javax.swing.JEditorPane
  
```

You'll find the constructors for **JEditorPane** in Table 2.9 and its methods in Table 2.10.

**Table 2.9 Constructors of the JEditorPane class.**

Constructor	Does This
<b>JEditorPane()</b>	Constructs a new <b>JEditorPane</b> .
<b>JEditorPane(String url)</b>	Constructs a <b>JEditorPane</b> based on a string containing data from a URL.
<b>JEditorPane(String type, String text)</b>	Constructs a <b>JEditorPane</b> initialized to the given text.
<b>JEditorPane(URL initialPage)</b>	Constructs a <b>JEditorPane</b> based on a given URL for input.

**Table 2.10 Methods of the JEditorPane class.**

Method	Does This
<b>void addHyperlinkListener(HyperlinkListener listener)</b>	Adds a hyperlink listener.
<b>protected EditorKit createDefaultEditorKit()</b>	Constructs the default editor kit for use when the component is first created.
<b>static EditorKit createEditorKitForContentType(String type)</b>	Creates a handler for the given type from the default registry of editor kits.
<b>void fireHyperlinkUpdate(HyperlinkEvent e)</b>	Notifies all listeners about the "hyperlink update" event.
<b>AccessibleContext getAccessibleContext()</b>	Gets the <b>AccessibleContext</b> .
<b>String getContentType()</b>	Gets the type of content that this editor is currently set to deal with.
<b>EditorKit getEditorKit()</b>	Gets the currently installed kit for handling content.
<b>EditorKit getEditorKitForContentType(String type)</b>	Gets the editor kit to use for the given type of content.
<b>URL getPage()</b>	Gets the current URL being displayed.

(continued)

Table 2.10 Methods of the **JEditorPane** class (*continued*).

Method	Does This
<b>Dimension</b> <code>getPreferredSize()</code>	Gets the preferred size.
<b>boolean</b> <code>getScrollableTracksViewportHeight()</code>	Returns True if a viewport should always force the height of this <b>Scrollable</b> object to match the height of the viewport.
<b>boolean</b> <code>getScrollableTracksViewportWidth()</code>	Returns True if a viewport should always force the width of this <b>Scrollable</b> object to match the width of the viewport.
<b>protected InputStream</b> <code>getStream(URL page)</code>	Gets a stream for the given URL.
<b>String</b> <code>getText()</code>	Gets the text contained in this editor pane.
<b>String</b> <code>getUIClassID()</code>	Gets the class ID for the <b>UI</b> object.
<b>boolean</b> <code>isFocusCycleRoot ()</code>	Makes <b>JEditorPane</b> be the root of a focus cycle.
<b>boolean</b> <code>isManagingFocus()</code>	Turns off Tab traversal once focus is gained.
<b>protected String</b> <code> paramString()</code>	Gets a string representation of this <b>JEditorPane</b> .
<b>protected void</b> <code>processKeyEvent(KeyEvent e)</code>	Makes sure that Tab and Shift+Tab events are acted upon.
<b>void</b> <code>read(InputStream in, Object desc)</code>	Reads data from a stream.
<b>static void</b> <code>registerEditorKitForContentType (String type, String classname)</code>	Establishes the default bindings.
<b>static void</b> <code>registerEditorKitForContentType(String type, String classname, ClassLoader loader)</code>	Establishes the default bindings of <b>type</b> to <b>classname</b> .
<b>void</b> <code>removeHyperlinkListener(HyperlinkListener listener)</code>	Removes a hyperlink listener.
<b>void</b> <code>replaceSelection(String content)</code>	Replaces the currently selected content with new content.
<b>protected void</b> <code>scrollToReference(String reference)</code>	Scrolls the view to the given reference location.
<b>void</b> <code>setContentType(String type)</code>	Sets the type of content that this editor handles.
<b>void</b> <code>setEditorKit(EditorKit kit)</code>	Sets the currently installed kit for handling content.
<b>void</b> <code>setEditorKitForContentType(String type, EditorKit k)</code>	Directly sets the editor kit to use for the given type.
<b>void</b> <code>setPage(String url)</code>	Sets the URL being displayed.
<b>void</b> <code>setPage(URL page)</code>	Sets the URL being displayed.
<b>void</b> <code>setText(String t)</code>	Sets the text to the given <b>String</b> .

Although editor panes have a default editor kit, you can create your own editor kit and install it into an editor pane with the **setEditorKit** method. You'll find the constructor for the **EditorKit** class in Table 2.11 and its methods in Table 2.12.

There are a number of ways to load text into editor panes. You can use the **setText** method to load text into an editor pane from a string. You can also use the **read** method to read data from a **Reader** object. You can also use the **setPage** method to initialize a text pane from a URL (including URLs of plain text, HTML documents, and RTF documents). In this case, the content type will be determined from the URL, and the registered editor kit for that content type will be used.

**Table 2.11** Constructor of the **EditorKit** class.

Constructor	Does This
<b>EditorKit()</b>	Constructs an <b>EditorKit</b> .

**Table 2.12** Methods of the **EditorKit** class.

Method	Does This
<b>abstract Object clone()</b>	Creates a copy of the <b>EditorKit</b> .
<b>abstract Caret createCaret()</b>	Gets a caret that can navigate through views.
<b>abstract Document createDefaultDocument()</b>	Creates an uninitialized text-storage model.
<b>void deinstall(JEditorPane c)</b>	Called when the kit is being removed from the <b>JEditorPane</b> .
<b>abstract Action[] getActions()</b>	Gets the set of commands that can be used in a model and view produced by this kit.
<b>abstract String getContentType()</b>	Gets the MIME type of the data that this kit represents support for.
<b>abstract ViewFactory getViewFactory()</b>	Gets a factory that can produce views of any models that are produced by this kit.
<b>void install(JEditorPane c)</b>	Called when the kit is being installed into the <b>JEditorPane</b> .
<b>abstract void read(InputStream in, Document doc, int pos)</b>	Inserts content from the given stream.
<b>abstract void read(Reader in, Document doc, int pos)</b>	Inserts content from the given stream.
<b>abstract void write(OutputStream out, Document doc, int pos, int len)</b>	Writes content from a document to the given stream.
<b>abstract void write(Writer out, Document doc, int pos, int len)</b>	Writes content from a document to the given stream.

---

**TIP:** Note that if you're loading in an HTML page with the **read** method, relative HTML references can't be resolved unless you use the **<BASE>** tag or set the **Base** property of the **HTMLDocument** object.

---

Here's a quick example in which I create an editor pane and place some text in it with the **setText** method—note that to enable scrolling, I enclose the text pane in a scroll pane:

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

/*
<APPLET
    CODE = editorpane.class
    WIDTH = 350
    HEIGHT = 280>
</APPLET>
*/

public class editorpane extends JApplet
{
    JEditorPane jeditorpane = new JEditorPane();

    public editorpane()
    {
        Container contentPane = getContentPane();

        jeditorpane.setEditable(true);

        jeditorpane.setText("Hello from Swing!");

        contentPane.add(new JScrollPane(jeditorpane));
    }
}
```

The result of this code appears in Figure 2.5. As you can see in the figure, the text I placed into the text pane is displayed. This example is `editorpane.java` on the CD.

However, this example treats its editor pane as a text area, which is much like using a backhoe to plant daisies—it works fine, but so much more is possible. See the next few solutions for more details.



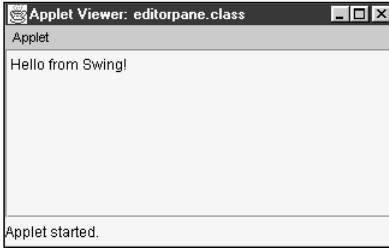


Figure 2.5 A basic text pane.

## Using HTML in Editor Panes

The Novice Programmer is back and excited. “OK,” says the NP putting a large lunch bag on your desk, “I’m ready to read an HTML document into an editor pane. Going to take a lot of programming, I expect.” “Not at all,” you say, “you just use the **setPage** method.” The NP pauses, sandwich in hand, and says, “Really?”

You can use the **JEditorPane** class’s **setPage** method to read an HTML page by passing that method a URL. That page can include graphics, hyperlinks, and more.

Here’s an example in which I open a Web page, which I call `page.html`, in an editor pane (note that I’m including an image in this Web page):

```
<HTML>
<BODY>

<H1>Here is some HTML</H1>

<CENTER>
<P>
This is an HTML page.
<P>
<B>Here's some bold text.</B>
<P>
<I>Here's some italic text.</I>
<P>
<U>Here's some underlined text.</U>
<P>
<IMG SRC = "image.jpg">

</CENTER>
</BODY>
</HTML>
```

To construct the URL of this page, which I'll store in the same directory as this example (`editorpaneHTML.java`), I use the **System.getProperty** method to get the current directory and the correct file separator (that is, `\` or `/`) for the current system. When I've constructed the URL of the Web page to open, I use the **setPage** method, enclosed in a **try** block, to open that page. Note that for security reasons, Java will object, by default, if you try to open Web pages from applets; therefore, I've made this example an application. Here's the code:

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

public class editorpaneHTML extends JFrame
{
    JEditorPane jeditorpane = new JEditorPane();

    public editorpaneHTML()
    {
        super("JEditorPane application");

        Container contentPane = getContentPane();
        jeditorpane.setEditable(false);
        String urlString = "file:" + System.getProperty("user.dir") +
            System.getProperty("file.separator") +
            "page.html";

        try {
            jeditorpane.setPage(urlString);
        }
        catch(IOException e) {}

        contentPane.add(jeditorpane);
    }

    public static void main(String args[])
    {
        final JFrame jframe = new editorpaneHTML();

        jframe.setBounds(100, 100, 300, 300);
        jframe.setVisible(true);
        jframe.setBackground(Color.white);

        jframe.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
    }
}
```

```

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

```

The result appears in Figure 2.6. As you can see in the figure, the Web page appears in the editor pane in all its glory. This example is `editorpaneHTML.java` on the CD.

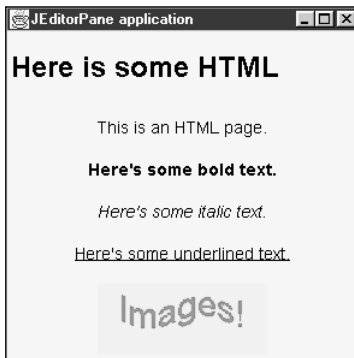


Figure 2.6 Opening an HTML page in an editor pane.

## Adding Hyperlinks to Editor Panes

You can handle hyperlinks in editor panes by using the **HyperLinkListener** interface. This interface has one method, called **hyperlinkUpdate**, which is passed an object of class **HyperlinkEvent**. You'll find the constructors of the **HyperlinkEvent** class in Table 2.13 and its methods in Table 2.14.

Table 2.13 Constructors of the **HyperlinkEvent** class.

Constructor	Does This
<b>HyperlinkEvent</b> (Object source, <b>HyperlinkEvent.EventType</b> type, URL u)	Constructs a new <b>HyperlinkEvent</b> .
<b>HyperlinkEvent</b> (Object source, <b>HyperlinkEvent.EventType</b> type, URL u, String desc)	Creates a new <b>HyperlinkEvent</b> with a description.

**Table 2.14** Methods of the **HyperlinkEvent** class.

Method	Does This
<b>String getDescription()</b>	Gets the description of the link as a string.
<b>HyperlinkEvent.EventType getEventType()</b>	Gets the type of event.
<b>URL getURL()</b>	Gets the URL that the link refers to.

Here's an example in which I open the following Web page, called `page2.html`, in an editor pane (note that this page has a hyperlink in it):

```
<HTML>
<BODY>

<H1>Here is some HTML</H1>

<CENTER>
<P>
This is an HTML page.
<P>
<B>Here's some bold text.</B>
<P>
<I>Here's some italic text.</I>
<P>
<U>Here's some underlined text.</U>
<P>
Here's a hyperlink to <A HREF = "page.html">page.html</A>.
<P>
<IMG SRC = "image.jpg">

</CENTER>
</BODY>
</HTML>
```

This hyperlink points to the Web page (`page.html`) introduced in the previous solution. Here's how I load the new Web page and add a hyperlink listener to it:

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

public class editorpaneURL extends JFrame
{
```

```

JEditorPane jeditorpane = new JEditorPane();

public editorpaneURL()
{
    super("JEditorPane application");

    Container contentPane = getContentPane();

    jeditorpane.setEditable(false);

    String urlstring = "file:" + System.getProperty("user.dir") +
        System.getProperty("file.separator") +
        "page2.html";

    try {
        jeditorpane.setPage(urlstring);
    }
    catch(IOException e) {}

    jeditorpane.addHyperlinkListener(new HyperlinkListener(){
        public void hyperlinkUpdate(HyperlinkEvent e)
        {
            try {
                jeditorpane.setPage(e.getURL());
            }
            catch(IOException e2) {}
        }
    });

    contentPane.add(jeditorpane);
}

public static void main(String args[])
{
    final JFrame jframe = new editorpaneURL();

    jframe.setBounds(100, 100, 300, 300);
    jframe.setVisible(true);
    jframe.setBackground(Color.white);
    jframe.setDefaultCloseOperation(DISPOSE_ON_CLOSE);

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

```

```

        }
    });
}

```

---

**TIP:** Hyperlinks are only active in noneditable editor panes.

---

The result of this code appears in Figure 2.7, and you can see the hyperlink in the figure. As this code is written, when any hyperlink event occurs—including moving the mouse over the hyperlink—the editor pane loads the target of the hyperlink automatically. That’s all there is to it. This example is `editorpaneURL.java` on the CD.

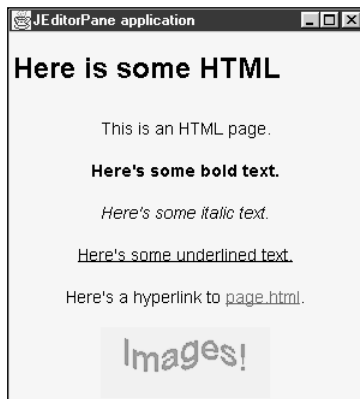


Figure 2.7 A hyperlink in an editor pane.

## Using RTF Files in Editor Panes

“Hmm,” says the Novice Programmer, “I’ve got a file full of all kinds of codes that the Big Boss wants me to open—take a look.” “That’s an RTF file with formatting fields,” you say, “and you should use an editor pane to open it.” “Oh,” says the NP.

Rich-text format (RTF) files can contain a great deal of text formatting, and many modern word processors, such as Microsoft Word, can store documents using this format. Editor panes can display RTF files, as I’ll show with an example. In this case, I open an RTF file—`document.rtf` on the CD—in an editor pane using the `setPage` method:

```

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import java.io.IOException;

public class editorpaneRTF extends JFrame
{
    JEditorPane jeditorpane = new JEditorPane();

    public editorpaneRTF()
    {
        super("JEditorPane application");

        Container contentPane = getContentPane();
        jeditorpane.setEditable(false);
        String url = "file:" + System.getProperty("user.dir") +
            System.getProperty("file.separator") +
            "document.rtf";

        try {
            jeditorpane.setPage(url);
        }
        catch(IOException e) {}

        contentPane.add(jeditorpane);
    }

    public static void main(String args[])
    {
        final JFrame jframe = new editorpaneRTF();

        jframe.setBounds(100, 100, 300, 300);
        jframe.setVisible(true);
        jframe.setBackground(Color.white);
        jframe.setDefaultCloseOperation(DISPOSE_ON_CLOSE);

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

```

The result of this code appears in Figure 2.8, where you can see the RTF document inside the editor pane. This example is `editorpaneRTF.java` on the CD.



Figure 2.8 An RTF document in an editor pane.

## Creating Text Panes

The Novice Programmer is back and worried. “Now,” says the NP, “I’ve got to display a button in a text area!” “That’s tough,” you say, “why not do that in a text pane instead, where it’s easy?”

The **JTextPane** class is derived from the **JEditorPane** class and adds quite a bit of functionality to **JEditorPane**—for example, you can insert controls into a text pane, and you can format text with style attributes. Here’s the inheritance diagram for **JTextPane**:

```

java.lang.Object
|___ java.awt.Component
    |___ java.awt.Container
        |___ javax.swing.JComponent
            |___ javax.swing.text.JTextComponent
                |___ javax.swing.JEditorPane
                    |___ javax.swing.JTextPane
  
```

You’ll find the constructors of the **JTextPane** class in Table 2.15 and its methods in Table 2.16.

Table 2.15 Constructors of the **JTextPane** class.

Constructor	Does This
<b>JTextPane()</b>	Constructs a new <b>JTextPane</b> .
<b>JTextPane(StyledDocument doc)</b>	Constructs a new <b>JTextPane</b> with a given document model.

Table 2.16 Methods of the **JTextPane** class.

Method	Does This
<b>Style addStyle(String nm, Style parent)</b>	Adds a new style into the logical style hierarchy.
<b>protected EditorKit createDefaultEditorKit()</b>	Constructs the editor kit to use by default.
<b>AttributeSet getCharacterAttributes()</b>	Gets the character attributes in effect at the caret.

(continued)



Table 2.16 Methods of the **JTextPane** class (*continued*).

Method	Does This
<b>MutableAttributeSet</b> getInputAttributes()	Gets the input attributes for the pane.
<b>Style</b> getLogicalStyle()	Gets the logical style assigned to the paragraph represented by the current position of the caret.
<b>AttributeSet</b> getParagraphAttributes()	Gets the paragraph attributes in effect at the location of the caret.
<b>Style</b> getStyle(String nm)	Gets a named style added previously.
<b>StyledDocument</b> getStyledDocument()	Gets the model associated with the editor.
<b>protected StyledEditorKit</b> getStyledEditorKit()	Gets the editor kit.
<b>String</b> getUIClassID()	Gets the class ID for the UI.
<b>void</b> insertComponent(Component c)	Inserts a component into the document as a replacement for the currently selected content.
<b>void</b> insertIcon(Icon g)	Inserts an icon into the document.
<b>protected String</b> paramString()	Gets a string representation of this <b>JTextPane</b> .
<b>void</b> removeStyle(String nm)	Removes a named non-null style previously added to the document.
<b>void</b> replaceSelection(String content)	Replaces the currently selected content.
<b>void</b> setCharacterAttributes(AttributeSet attr, boolean replace)	Applies the given attributes to character content.
<b>void</b> setDocument(Document doc)	Associates the editor with a text document.
<b>void</b> setEditorKit(EditorKit kit)	Sets the currently installed kit for handling content.
<b>void</b> setLogicalStyle(Style s)	Sets the logical style to use for the paragraph at the current caret position.
<b>void</b> setParagraphAttributes(AttributeSet attr, boolean replace)	Applies the given attributes to paragraphs.
<b>void</b> setStyledDocument(StyledDocument doc)	Associates the editor with a text document.

Here's an example in which I construct a text pane and add some text to it:

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

/*
<APPLET
    CODE = textpane.class
    WIDTH = 350
    HEIGHT = 280>
```

```

</APPLET>
*/

public class textpane extends JApplet
{
    JTextPane jtextpane = new JTextPane();

    public void init()
    {
        Container contentPane = getContentPane();

        jtextpane.setFont(new Font("Times-Roman", Font.BOLD, 18));
        jtextpane.setText("Hello from Swing!");

        contentPane.add(jtextpane);
    }
}

```

You'll find the result of this code in Figure 2.9, where you can see the text pane with its displayed text. This example is `textpane.java` on the CD.

Text panes are powerful controls, of course, and this example hasn't even scratched the surface. Take a look at the next couple of solutions for more details.

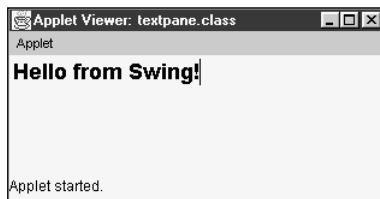


Figure 2.9 A basic text pane.

## Inserting Images and Controls into Text Panes

To insert images into text panes, you can use the **`insertIcon`** method; to insert other components into a text pane, you can use the **`insertComponent`** method. To show how this works, I use both methods in an example. In this case, I add a button and an image to a text pane, like this:

```

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

```

```

/*
<APPLET
    CODE = textpaneimages.class
    WIDTH = 350
    HEIGHT = 280>
</APPLET>
*/

public class textpaneimages extends JApplet implements ActionListener
{
    JTextPane jtextpane = new JTextPane();
    JButton jbutton = new JButton("Click Me");

    public void init()
    {
        Container contentPane = getContentPane();

        jtextpane.setFont(new Font("Times-Roman", Font.BOLD, 18));
        jtextpane.setText("Hello from Swing!");

        jbutton.addActionListener(this);
        jtextpane.insertComponent(jbutton);
        jtextpane.insertIcon(new ImageIcon("image.jpg"));

        contentPane.add(jtextpane);
    }

    public void actionPerformed(ActionEvent e)
    {
        showStatus("You clicked the button.");
    }
}

```

I've made the button active by adding an action listener to it. The result appears in Figure 2.10, where you can see the image and button in the text pane. This example appears as `textpaneimages.java` on the CD.



Figure 2.10 Adding components and images to a text pane.

## Setting Text Pane Text Attributes

“OK,” says the Novice Programmer, “I need to do a lot of text formatting—italics, bold, and colored fonts. Is there any way to do this in Java?” “There sure is,” you say, “in fact, you can do it with text panes.” “Great!” says the NP.

You can use the **setCharacterAttributes**, **setParagraphAttributes**, and **setLogicalStyle** methods of text panes to set and style the text attributes. These techniques are fairly involved. Here’s an example in which I display several lines of text in a text pane, all with a different custom style.

I start by creating some new styles—normal, red, italic, bold, and big. To do that, I use the text pane’s **getStyledDocument** method to get a **StyledDocument** object corresponding to the text pane; then, I create the new styles with the **addStyle** method:

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

/*
<APPLET
    CODE = textpaneattributes.class
    WIDTH = 350
    HEIGHT = 280>
</APPLET>
*/

public class textpaneattributes extends JApplet
{
    JTextPane jtextpane = new JTextPane();
    String string = new String("Hello from Swing!\r\n");
    int stringlength = string.length();
    int position = 0;

    public void init()
    {
        Container contentPane = getContentPane();

        jtextpane.setText(string);
        StyledDocument styleddocument = jtextpane.getStyledDocument();

        Style normal = styleddocument.addStyle("normal", null);
        StyleConstants.setFontFamily(normal, "SansSerif");

        Style red = styleddocument.addStyle("red", normal);
```

```

        StyleConstants.setForeground(red, Color.red);

        Style bold = styleddocument.addStyle("bold", normal);
        StyleConstants.setBold(bold, true);

        Style italic = styleddocument.addStyle("italic", normal);
        StyleConstants.setItalic(italic, true);

        Style big = styleddocument.addStyle("big", normal);
        StyleConstants.setFontSize(big, 24);
        .
        .
        .

```

Next, I add successive lines of text, applying one of the new styles to each line, by moving the current location of the text-insertion caret in the text pane and using **setLogicalStyle**:

```

public class textpaneattributes extends JApplet
{
    JTextPane jtextpane = new JTextPane();
    String string = new String("Hello from Swing!\r\n");
    int stringlength = string.length();
    int position = 0;

    public void init()
    {
        Container contentPane = getContentPane();

        jtextpane.setText(string);
        StyledDocument styleddocument = jtextpane.getStyledDocument();

        Style normal = styleddocument.addStyle("normal", null);
        StyleConstants.setFontFamily(normal, "SansSerif");

        Style red = styleddocument.addStyle("red", normal);
        StyleConstants.setForeground(red, Color.red);

        Style bold = styleddocument.addStyle("bold", normal);
        StyleConstants.setBold(bold, true);

        Style italic = styleddocument.addStyle("italic", normal);
        StyleConstants.setItalic(italic, true);

        Style big = styleddocument.addStyle("big", normal);
        StyleConstants.setFontSize(big, 24);
    }
}

```

```

        styleddocument.setLogicalStyle(position, normal);

        position += stringlength;
        jtextpane.setCaretPosition(styleddocument.getLength());
        jtextpane.replaceSelection(string);
        styleddocument = jtextpane.getStyledDocument();
        styleddocument.setLogicalStyle(position, red);

        position += stringlength;
        jtextpane.setCaretPosition(styleddocument.getLength());
        jtextpane.replaceSelection(string);
        styleddocument = jtextpane.getStyledDocument();
        styleddocument.setLogicalStyle(position, bold);

        position += stringlength;
        jtextpane.setCaretPosition(styleddocument.getLength());
        jtextpane.replaceSelection(string);
        styleddocument = jtextpane.getStyledDocument();
        styleddocument.setLogicalStyle(position, italic);

        position += stringlength;
        jtextpane.setCaretPosition(styleddocument.getLength());
        jtextpane.replaceSelection(string);
        styleddocument = jtextpane.getStyledDocument();
        styleddocument.setLogicalStyle(position, big);

        contentPane.add(jtextpane);
    }
}

```

That's it. The result appears in Figure 2.11. You can see each styled line in the figure. Note, however, that this is only the beginning for style work in text panes—there's a tremendous amount of depth here.



Figure 2.11 Styling text in a text pane.

## Working with Sound in Applets

“Hey,” says the Novice Programmer, “that applet beeped at me. How can I make my applets sound off the same way?” “With the **getAudioClip** method,” you say. “Great!” says the NP.

You can make applets play sounds with the Applet class’s **getAudioClip** method. Using this method, you can play sound files in these formats: .au and .snd (commonly found in Sun workstations), .aif (Mac), and .wav (Windows). Note that Sun has yet to support the .mp3 format with the classes that ship with the SDK (although, in fact, MP3 is supported by the Java Media Framework, which does not ship with the SDK).

To play sounds, you use the Applet class’s **getAudioClip** method to get a **Clip** object, and you can use these methods of that object:

- **play**—Plays the clip once
- **loop**—Keeps playing the clip
- **stop**—Stops the clip

Here’s an example that will play the .wav file `tada.wav` that comes with Windows (in Windows 95/98/Me, you should find it in `C:\Windows\Media`, and in Windows NT and 2000, it’s in `C:\Winnt\Media`)—just make sure `tada.wav` (or whatever sound file you modify this example to use) is in the same directory as the applet’s code:

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

/*
<APPLET code="soundApplet.class" width = 300 height = 300>
</APPLET>
*/

public class soundApplet extends JApplet
{
    AudioClip clip;
    JButton button;

    public void init()
    {
        clip = getAudioClip(getDocumentBase(), "tada.wav");

        button = new JButton("Play sound");
        getContentPane().add(button);
    }
}
```

```

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e){clip.loop();});
        }
    }
}

```

That's all there is to it—now when the user clicks the button in the applet, `tada.wav` will play in a continuous loop. This applet is shown in Figure 2.12 and appears on the CD as `soundApplet.java`.



Figure 2.12 Playing sounds with an applet.

## Working with Sound in Applications

“Hmm,” says the Novice Programmer, “I can play sounds in my applets now, but what about applications?” “You’re in luck,” you say, “there’s a great deal of support for sounds in applications.” “Oh boy,” says the NP.

Java 2 version 1.3 added a great deal of support for working with sound in applications. The sound capabilities of Java are advanced and involved, however, and for the most part are beyond the scope of this book. You can mix sounds, record sounds, and create all kinds of effects with the sound packages such as `javax.sound.sampled` and `javax.sound.midi`.

In this example, I’ll write an application (not an applet, as in the previous solution) to play the `.wav` file `tada.wav` that comes with Windows (in Windows 95/98/Me, you should find it in `C:\Windows\Media`, and in Windows NT and 2000, it’s in `C:\Winnt\Media`)—just make sure `tada.wav` (or whatever sound file you modify this example to use) is in the same directory as the applet’s code.



Here's how things work in overview: You create an **AudioInputStream** object corresponding to the sound file you want to play, and you get the format of that sound file using the **DataLine.Info** class. Passing the information about the file's format to the **getLine** method of the **AudioSystem** class, you get a sound clip of the **Clip** class, which may be played. Here are the methods you can use with **Clip** objects to play sounds:

- **start**—Plays the sound
- **loop(Clip.LOOP\_CONTINUOUSLY)**—Loops continuously
- **stop**—Stops playing

Here's what the code looks like (here, I'm just having this application play *tada.wav*):

```
import java.awt.Container;
import java.io.*;
import javax.swing.*;
import java.awt.event.*;
import javax.sound.sampled.*;

public class soundApp extends JFrame
{
    JButton button;
    Clip clip;

    public soundApp()
    {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(300,300);

        File file = new File("tada.wav");

        try
        {
            AudioInputStream audiosource =
                AudioSystem.getAudioInputStream(file);
            DataLine.Info info =
                new DataLine.Info(Clip.class, audiosource.getFormat());
            clip = (Clip)AudioSystem.getLine(info);
            clip.open(audiosource);
        }
        catch(UnsupportedAudioFileException e){}
        catch(LineUnavailableException e){}
        catch(IOException e){}
```

```

        button = new JButton("Play sound");
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e){
                if(e.getActionCommand().equals("Play sound"))
                    clip.loop(clip.LOOP_CONTINUOUSLY);}}}

        Container contentpane = getContentPane();
        contentpane.add(button);
        setVisible(true);
    }

    public static void main(String[] args)
    {
        soundApp soundapp = new soundApp();
    }
}

```

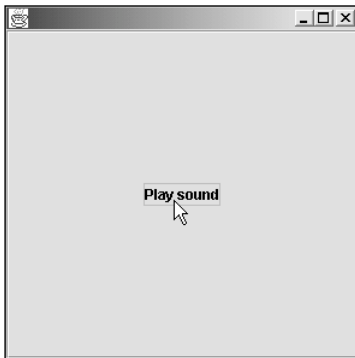
When you run this application, which is `soundApp.java` on the CD, it will play `tada.wav` for you. This application is shown in Figure 2.13.

You can use the extensive Java sound interfaces in applets, even though they are designed for applications. To do so, you need to add the appropriate permissions to the `.java.policy` file, like so:

```

grant{
    permission java.io.FilePermission "<<ALL FILES>>", "read, write";
    permission javax.sound.sampled.AudioPermission "record";
    permission java.util.PropertyPermission "user.dir", "read";
}

```



**Figure 2.13** Playing sounds in an application.