Chapter 18. Tables

In this chapter:

- JTable
- Stocks Table: part I - Basic JTable example
- Stocks Table: part II - Custom renderers
- Stocks Table: part III - Data formatting
- Stocks Table: part IV - Sorting columns
- Stocks Table: part V - JDBC
- Stocks Table: part VI - Column addition and removal
- Expense report application
- JavaBeans property editor

18.1 JTable

JTable is extremely useful for the display, navigation, and editing of tabular data. Because of its complex nature, JTable has a whole package devoted just to it: javax.swing.table. This package consists of a set of classes and interfaces which we will review briefly here. In the examples that follow, we construct--in a step-wise fashion--a table-based application used to display stock market data. (In chapters 22 and 26 we enhance this application further to allow printing and print preview, as well as CORBA client-server interaction.) This chapter concludes with an expense report application demonstrating the use of different components as table cell editors and renderers, and the completion of the JavaBeans property editor we started to build in chapter 4.

18.1.1 JTable

class javax.swing.JTable

This class represents Swing's table component and provides a rich API for managing its behavior and appearance. JTable directly extends JComponent and implements the TableModelListener, TableColumnModelListener, ListSelectionListener, CellEditorListener, and the Scrollable interfaces (it is meant to be placed in a JScrollPane). Each JTable has three models: a TableModel, TableColumnModel, and ListSelectionModel. All table data is stored in a TableModel, normally in a 2-dimensional structure such as a 2D array or a Vector of Vectors. TableModel implementations specify how this data is stored, as well as manage the addition, manipulation, and retrieval of this data. TableModel also plays a role in dictating whether or not specific cells can be edited, as well as the data type of each column of data. The location of data in a JTable's TableModel does not directly correspond to the location of that data as displayed by JTable itself. This part is controlled at the lowest level by TableColumnModel.

A TableColumnModel is designed to maintain instances of TableColumn, each of which represents a single column of TableModel data. TableColumn is the class that is responsible for managing the display of a column in the actual JTable GUI. Each TableColumn has an associated cell renderer, cell editor, table header, and a cell renderer for the table header. When a JTable is placed in a JScrollPane these headers are placed in the scroll pane's COLUMN_HEADER viewport and can be dragged and resized to reorder and change the size of columns. A TableColumn's header renderer is responsible for returning a component used to render the column header, and the cell renderer is responsible for returning a component used to render each cell. As with JList and JTree renderers, these renderers also act as "rubber stamps"[API] and are not at all interactive. The component returned by the cell editor, however, is completely interactive. Cell renderers are instances of TableCellRenderer and cell editors are instances of CellEditor. If none are explicitly assigned, default versions will be used based on the Class type of the corresponding TableModel colmn data.

TableColumnModel's job is to manage all TableColumns, providing control over order, column

selections, and margin size. To support several different modes of selection, TableColumnModel maintains a ListSelectionModel which, as we learned in chapter 10, allows single, single-interval, and multiple-interval selections. JTable takes this flexibility even further by providing functionality to customize any row, column, and/or cell-specific selection schemes we can come up with.

We can specify one of several resizing policies which dictate how columns react when another column is resized, as well as whether or not grid lines between rows and/or columns should appear, margin sizes between rows and columns, selected and unselected cell foreground and background colors, the hieght of rows, and the width of each column on a column-by-column basis.

With tables come two new kinds of events in Swing: TableModelEvent and TableColumnModelEvent. Regular Java events apply to JTable as well. For instance, we can use MouseListeners to process double mouse clicks. ChangeEvents and ListSelectionEvents are also used for communication in TableColumnModel.

Note: Although JTable implements several listener interfaces, it does not provide any methods to register listeners other than those inherited from JComponent. To attach listeners for detecting any of the above events we must first retrieve the appropriate model.

A number of constructors are provided for building a JTable component. We can use the default constructor or pass the table's data and column names each as a separate Vector. We can build an empty JTable with a specified number of rows and columns. We can also pass table data to the constructor as a two-dimensional array of data Objects along with an Object array of column names. Other constructors allow creation of a JTable with specific models. In all cases, if a specific model is not assigned in the constructor, JTable will create default implementations with its protected createDefaultColumnModel(), createDefaultDataModel(), and createDefaultSelectionModel() methods. It will do the same for each TableColumn renderer and editor, as well as its JTableHeader, using createDefaultEditors(), createDefaultRenderers(), and createDefaultTableHeader().

JTable is one of the most complex Swing components and keeping track of its constituents and how they interact is intially a challenge. Before we begin the step-wise construction of our stocks table application, we must make our way through all of these details. The remainder of this section is devoted to a discussion of the classes and interfaces that underly JTable.

18.1.2 The TableModel interface

abstract interface javax.swing.table.TableModel

Instances of TableModel are responsible for storing a table's data in a 2-dimensional structure such as a 2-dimensional array or a vector of vectors. A set of methods is declared for use in retrieving data from a table's cells. The getValueAt() method should retrieve data from a given row and column index as an Object, and setValueAt() should assign the provided data object to the specified location (if valid). getColumnClass() should return the Class describing the data objects stored in the specified column (used to assign a default renderer and editor for that column), and getColumnName() should return the String name associated with the specified column (often used for that column's header). The getColumnCount() and getRowCount() methods should return the number of contained columns and rows respectively.

Note: The getRowCount() is called frequently by JTable for display purposes and should be designed with efficiency in mind because of this.

The isCellEditable() method should return true if the cell at the given row and column index can be edited. The setValueAt() method should be designed so that if isCellEditable() returns false, the object at the given location will not be updated.

This model supports the attachment of TableModelListeners (see below) which should be notified about changes to this model's data. As expected, methods for adding and removing these listeners are provided, addTableModelListener() and removeTableModelListener(), and implementations are responsible for dispatching TableModelEvents to those registered.

Each JTable uses one TableModel instance which can be assigned/retrieved using JTable's setModel() and getModel() methods respectively.

18.1.3 AbstractTableModel

abstract class javax.swing.table.AbstractTableModel

AbstractTableModel is an abstract class implementing the TableModel interface. It provides default code for firing TableModelEvents with the fireTableRowsDeleted(), fireTableCellUpdated(), and fireTableChanged() methods. It also manages all registered TableModelListeners in an EventListenerList (see chapter 2).

The findColumn() method searches for the index of a column with the given String name. This search is performed in a linear fashion (referred to as "naive" in the documentation) and should be overriden for large table models for more efficient searching.

Three methods need to be implemented in concrete sub-classes: getRowCount(), getColumnCount() and getValueAt(int row, int

column), and we are expected to use this class as a base for building our own TableModel implementations (rather than DefaultTableModel--see below).

18.1.4 DefaultTableModel

class javax.swing.tableDefaultTableModel

DefaultTableModel is the default concrete TableModel implementation used by JTable when no model is specified in the constructor. It uses a Vector of Vectors to manage its data, which is one major reason why extending AbstractTableModel is often more desirable (because AbstractTableModel allows complete control over how data storage and manipulation is implemented). This Vector can be assigned with the overloaded setDataVector() method, and retrieved with the getDataVector() method. Internally, two overloaded, protected convertToVector() methods are used for converting Object arrays to Vectors when inserting rows, columns, or assigning a new data Vector. Methods for adding, inserting, removing, and moving columns and rows of data are also provided.

> Note: The position of a row or column in the model does not correspond to JTable's GUI representation of that row or column. Rather, this representation is performed instances of TableColumn which map to specific model columns. When a TableColumn is moved in the GUI, the associated data in the TableModel model stays put, and vice versa (see below).

Along with the TableModelEvent functionality inherited from AbstractTableModel, this class implements three new event-dispatching methods, each taking a TableModelEvent as parameter: newDataAvailable(), newRowsAdded(), and rowsRemoved(). The newRowsAdded() method ensures that new rows (see discussion of TableModelEvent below) have the correct number of columns by either removing excess elements, or using null for each missing cell. If null is passed to any of these methods they will construct and fire a default TableModelEvent which assumes that all table model data has changed.

18.1.5 TableColumn

class javax.swing.table.TableColumn

TableColumn is the basic building block of JTable's visual representation, and provides the main link between the JTable GUI and its model. Note that TableColumn does not extend java.awt.Component, and is thus not a component. Rather, it acts more like a model that maintains all the properties of a column displayed in a JTable. An instance of TableColumn represents a specific column of data stored in a TableModel. TableColumn maintains the index of the TableModel column it represents as property modelIndex. We can get/set this index with the getModelIndex() and setModelIndex() methods. It is important to remember that the position of a TableColumn graphically in JTable does not at all correspond to the corresponding TableModel column index.

A TableColumn is represented graphically by a column header renderer, cell renderer, and optionally a cell editor. The renderers must be instances of TableCellRenderer, and the editor must be an instance of TabeCellEditor (see below). A column's header is rendered by a component stored as the headerRenderer property. By default this is an instance of DefaultTableCellRenderer (a JLabel with a beveled border--see below) and is created with TableColumn's protected createDefaultHeaderRenderer() method. This renderer simply renders the String returned by the toString() method of the Object referred to by the headerValue property. The header renderer and value can be assigned/retrieved with setHeaderRenderer()/getHeaderRenderer() and setHeaderValue()/getHeaderValue() methods respectively. Often headerValue directly corresponds to the column name retrived using TableModel's getColumnName() method. If headerValue is not explicitly set it defaults to null.

The column cell renderer and editor also default to null, and unless they are explicitly specified using setCellRenderer() or setCellEditor(), are automatically assigned based on the Class type of the data stored in the associated column in the TableModel (retreived using TableModel's getColumnClass() method). Explicity specified renderers and editors are referred to as *column-based*, whereas those determined by data type are referred to as *class-based* (we will discuss renderers and editors in more detail below).

Each TableColumn has an identifier property which also defaults null. This property can be assigned/retrieved using typical set/get accessors, and the getIdentifier() method will return the headerValue property if identifier is null. When searching for a TableColumn by name (using TableColumnModel's getColumnIndex() method--see below--or JTable's getColumn() method), the given Object will be compared, using Object's equals() method, to each TableColumn identifier. Since it is possible that more than one TableColumn will use the same identifier, the first match is returned as the answer.

TableColumn maintains properties minWidth, maxWidth, and width. The first two specify the minimum and maiximum allowable widths for column rendering, and the width property stores the current width. Each can be retrieved and assigned with typical get/set methods: getMinWidth()/setMinWidth(), getMaxWith()/setMaxWidth(), and getWidth()/setWidth(). minWidth defaults to 15, maxWidth defaults to Integer.MAX_VALUE, and width defaults to 75. When a JTable is resized it will try to maintain its width, and will never exceeed its maximum or shrink smaller than its minimum.

> Note: All other visual aspects of each column are controlled by either JTable or TableColumnModel (see below).

TableColumn also maintains an isResizable property, specifying whether or not its width can be changed by the user (this does not apply to programmatic calls to setWidth()). (We will discuss resizing in more detail below).

An interesting and rarely used property maintained by TableColumn, resizedPostingDisabledCount, is used to enable/disable the

posting of PropertyChangeEvents when a TableColumn's width changes. This is an int value that is incremented on each call to disableResizedPosting(), and decremented on each call to enableResizedPosting(). Events will only be fired if this value is less than or equal to 0. The logic behind this is that if two separate sources both disable resize event posting, then two calls should be required to re-enable it.

> Big Alert! As of Java 2 FCS the resizedPostingDisabledCount property is not actually used anywhere and does not play a role in PropertyChangeEvent firing.

TableColumn fires PropertyChangeEvents when any of the width, cellRenderer, headerRenderer, or headerValue bound properties change. Thus we can add and remove PropertyChangeListeners to be notified of these changes. The corresponding property names are COLUMN_WIDTH_PROPERTY, COLUMN_RENDERER_PROPERTY, HEADER_RENDERER_PROPERTY, and HEADER_VALUE_PROPERTY.

18.1.6 The TableColumnModel interface

abstract interface javax.swing.table.TableColumnModel

This model is designed to maintain a JTable's TableColumns, and provides control over column selections and margin size. TableColumnModel controls how JTable displays its TableModel data. The addColumn() method should append a given TableColumn to the end of the structure used to maintain them (usually a Vector), removeColumn() should remove a given TableColumn, and moveColumn() should change the location of a given TableColumn within that structure.

> Note: When creating a JTable, if no TableColumnModel is specified, one will automatically be constructed for us containing TableColumns displaying TableModel data in the same order it appears in the model. This will only occur of JTable's autoCreateColumnsFromModel property is set true, which it is by default. This is very helpful, but has the often undesirable side-effect of completely rebuilding the TableColumnModel whenever TableModel changes. Thus it is common to set this property to false once a JTable has been created or after a new TableModel is assigned.

Unlike the location of a column in TableModel implementations, the index of a TableColumn in a TableColumnModel's storage structure directly corresponds to its position in the JTable GUI. Note that the moveColumn() method is called whenever the user drags a column to a new position.

The getColumnCount() method should return the number of TableColumns currently maintained, getColumns() should return an Enumeration of all contained TableColumns, and getColumn() returns the TableColumn at the given index. The getColumnIndex() method should return the index of the TableColumn whose identifier property is equal to (using Object's equals() method) the given Object. getColumnIndexAtX() should return the index of the TableColumn at the given x-corrdinate in table-space (if getColumnIndexAtX() is passed a coordinate that maps to the margin space between adjacent columns, or any x-coordinate that does not correspond to a table column, it will return -1). setColumnMargin() and getColumnMargin() should allow assignment and retrieval of an int value for use as the margin space on each side of each table column. The getTotalColumnWidth() should return the sum of the current width of all TableColumns including all margin space.

> Note: The margin size does not correspond to the width of the separating grid lines between columns in JTable. In fact, the width of these lines is always 1, and cannot be changed without customizing JTable's UI delegate.

TableColumnModel declares methods for controlling the selection of its TableColumns, and allows assignment and retrieval of a ListSelectionModel implementation used to store information about the current column selection with the methods setSelectionModel() and getSelectionModel(). The setColumnSelectionAllowed() method is intended to turn on/off column selection capabilities, and getColumnSelectionAllowed() should return a boolean specifying whether slection is currently allowed or not. For convenience, JTable's setColumnSelectionAllowed() method delegates its traffic to the method of the same signature in this interface.

TableColumnModel also declares support for TableColumnModelListeners (see below). TableColumnModel implementations are expected to fire a TableColumnModelEvent whenever a TableColumn is added, removed, or moved, a ChangeEvent whenever margin size is changed, and a ListSelectionEvent whenever a change in column selection occurs.

18.1.7 DefaultTableColumnModel

class javax.swing.table.DefaultTableColumnModel

This class is the concrete default implementation of the TableColumnModel interface used by JTable when none is specifically assigned or provided at construction time. All TableColumnModel methods are implemented as expected (see above), and the following protected methods are provided to fire events: fireColumnAdded(), fireColumnRemoved(), fireColumnMoved(), fireColumnSelectionChanged(), and fireColumnMarginChanged(). A valueChanged() method is provided to listen for column selection changes and fire a ListSelectionEvent when necessary. And a propertyChanged() method is used to update the totalColumnWidth property when the width of a contained TableColumn changes.

18.1.8 The TableCellRenderer interface

abstract interface javax.swing.table.TableCellRenderer

This interface describes the renderer used to display cell data in a TableColumn. Each TableColumn has an associated TableCellRender which can be assigned/retrieved with the setCellRenderer()/getCellRenderer() methods. The getTableCellRendererComponent() method is the only method declared by this interface, and is expected to return a Component that will be used to actually render a cell. It takes the following parameters:

JTable table: the table instance containing the cell to be rendered.

Object value: the value used to represent the data in the given cell.

boolean isSelected: whether or not the given cell is selected.

boolean hasFocus: whether or not the given cell has the focus (true if it was clicked last).

int row: can be used to return a renderer component specific to row or cell.

int column: can be used to return a renderer component specific to column or cell.

We are expected to customize or vary the returned component based on the given parameters. For instance, given a value that is an instance of Color, we might return a special JLabel subclass that paints a rectangle in the given color. This method can be used to return different renderer components on a column, row, or cell-specific basis, and is similar to JTree's TreeCellRenderer getTreeCellRendererComponent() method. As with JTree and JList, the renderer component returned acts as a "rubber stamp" **API** used strictly for display purposes.

Note: the row and column parameters refer to the location of data in the TableModel, not a cell location in the TableColumnModel.

When JTable's UI delegate repaints a certain region of a table it must query that table to determine the renderer to use for each cell that it needs to repaint. This is accomplished through JTable's getCellRenderer() method which takes row and column parameters, and returns the component returned by the getTableCellRendererComponent() method of the TableCellRenderer assigned to the appropriate TableColumn. If there is no specific renderer assigned to that TableColumn (recall that this is the case by default), the TableModel's getColumnClass() method is used to recursively determine an appropriate renderer for the given data type. If there is no specific class-based renderer specified for a given class, getColumnClass() searches for one corresponding to the super-class. This process will, in the most generic case, stop at Object, for which a DefaultTabelCellRenderer is used (see below).

A DefaultTreeCellRenderer is also used if the class is of type Number (subclasses are BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, and Short) or Icon. If the type happens to be a Boolean, a JCheckBox is used. We can specify additional class-based renderers with JTable's setDefaultRenderer() method. Remember that class-based renderers will only be used if no column-based renderer has been explicitly assigned to the TableColumn containing the given cell.

18.1.9 DefaultTableCellRenderer

class javax.swing.table.DefaultTableCellRenderer

This is the concrete default implementation of the TableCellRenderer interface. DefaultTableCellRenderer extends JLabel and is used as the default class-based renderer for Number, Icon, and Object data types. Two private Color variables are used to hold selected foreground and background colors which are used to render the cell if it is editable and if it has the current focus. These colors can be assigned with DefaultTableCellRenderer's overridden setBackground() and setForeground() methods.

A protected Border property is used to store the border that is used when the cell does not have the current focus. By default this is an EmptyBorder with top and bottom space of 1, and left and right space of 2. Unfortunately DefaultTableCellRenderer does not provide a method to change this border.

DefaultTableCellRenderer renders the value object passed as parameter to its getTableCellRenderer() method by setting its label text to the String returned by that object's toString() method. Note that all default JLabel attibutes are used in rendering. Also note that we can do anything to this renderer that we can do to a JLabel, such as assign a tooltip or a disabled/enabled state.

Note: JTable can have a tooltip assigned to it just as any other Swing component. However, tooltips assigned to renderers take precedence over that assigned to JTable, and in the case that both are used the renderer's tooltip text will be displayed when the mouse lies over a cell using it.

18.1.10 The TableCellEditor interface

abstract interface javax.swing.table.TableCellEditor

This interface extends CellEditor and describes the editor used to edit cell data in a TableColumn. Each TableColumn has an associated TableCellEditor which can be assigned/retrieved with the setCellEditor()/getCellEditor() methods. The getTableCellEditorComponent() method is the only method declared by this interface, and is expected to return a Component that will be used to allow editing of a cell's data value. It takes the following parameters:

JTable table: the table instance containing the cell to be rendered.

Object value: the value used to represent the data in the given cell.

boolean isSelected: whether or not the given cell is selected.

int row: can be used to return a renderer component specific to row or cell.

int column: can be used to return a renderer component specific to column or cell.

We are expected to customize or vary the returned component based on the given parameters. For instance, given a value that is an instance of Color, we might return a special JComboBox which lists several color choices. This method can be used to return different editor components on a column, row, or cell-specific basis, and is similar to JTree's TreeCellEditor getTreeCellEditorComponent() method.

Note: The row and column parameters refer to the location of data in the TableModel, not a cell location in the TableColumnModel.

As with table cell renderers, each TableColumn has a column-based editor associated with it. By default this editor is null and can be assigned/retrieved with TableColumn's setCellEditor()/getCellEditor() methods. Unlike renderers, table cell editors are completely interactive and do not simply act as rubber stamps.

TableCellEditor implementations must also implement methods defined in the CellEditor interface: addCellEditorListener(), removeCellEditorListener(), cancelCellEditing(), stopCellEditing(), isCellEditable(), shouldSelectCell(), and getCellEditorValue(). The isCellEditable() method is expected to be used in combination with TableModel's isCellEditable() method to determine whether a given cell can be edited. Only in the case that both return true is editing allowed. (See discussion of the CellEditor interface in section 17.1.13 for more about each of these methods.)

To initiate cell editing on a given cell, JTable listens for mouse presses and invokes its editCellAt() method in response. This method queries both the TableModel and the appropriate cell editor to determine if the given cell can be edited. If so the editor component is retrieved with getTableCellEditorComponent() and placed in the given cell (its bounds are adjusted so that it will fit within the current cell bounds). Then JTable adds itself to the editor component (recall that JTable implements the CellEditorListener interface) and the same mouse event that sparked the edit gets sent to the editor component. Finally the cell editor's shouldSelectCell() method is invoked to determine whether the row containing that cell should become selected.

The default implementation of TableCellEditor is provided as DefaultCellEditor. Unfortunately DefaultCellEditor is not easily extensible and we are often forced to implement all TableCellEditor and CellEditor functionality ourselves.

18.1.11 DefaultCellEditor

class javax.swing.DefaultCellEditor

DefaultCellEditor is a concrete implementation of the TableCellEditor interface and also the TreeCellEditor interface. This editor is designed to return either a JTextField, JComboBox, or JCheckBox for cell editing. It is used by both JTable and JTree components and is discussed 17.1.15.

18.1.12 The TableModelListener interface

abstract interface javax.swing.event.TableModelListener

This interface describes an object that listens to changes in a TableModel. Method tableChanged() will be invoked to notify us of these changes. TableModel's addTableModelListener() and removeTableModelListener() methods are used to add and remove TableModelListeners respectively (they are not added directly to JTable).

18.1.13 TableModelEvent

class javax.swing.TableModelEvent

This event extends EventObject and is used to notify TableModelListeners registered with TableModel about changes in that model. This class consists of four properties each accessible with typical get methods:

int column: specifies the column affected by the change. TableModelEvent.ALL_COLUMNS is used to indicate that more than one column is affected.

int firstRow: specifies the first row affected. TableModelEvent.HEADER_ROW can be used here to indicate that the name, type, or order of one or more columns has changed.

int lastRow: the last row affected. This value should always be greater than or equal to firstRow.

int type: the type of change that occured. Can be any of TableModelEvent.INSERT, TableModelEvent.DELETE, or TableModelEvent.UPDATE. INSERT and DELETE indicate the insertion and deletion of rows respectively. UPDATE indicates that values have changed but the number of rows and columns has not changed.

As with any EventObject we can retrieved the source of a TableModelEvent with getSource().

18.1.14 The TableColumnModelListener interface

abstract interface javax.swing.event.TableColumnModelListener

This interface describes an object that listens to changes in a TableColumnModel: the adding, removing and movement of columns, as well as changes in margin size and the current selection. TableColumnModel provides methods for adding and removing these listeners: addTableColumnModelListener() and removeTableColumnModelListener(). (As is the case with TableModelListeners, TableColumnModelListeners are not directly added to JTable.)

Five methods are declared in this interface and must be defined by all implementations: columnAdded(TableColumnModelEvent), columnRemoved(TableColumnModelEvent),

columnMoved(TableColumnModelEvent), columnMarginChanged(TableColumnModelEvent),

and columnSelectionChanged(ListSelectionEvent). ListSelectionEvents are forwarded to TableColumnModel's ListSelectionModel.

18.1.15 TableColumnModelEvent

class javax.swing.event.TableColumnModelEvent

This event extends EventObject and is used to notify TableColumnModel about changes to a range of columns. These events are passed to TableColumnModelListeners. The fromIndex property specifies the lowest index of the colmn in the TableColumnModel affect by the change. The toIndex specifies the highest index. Both can be retrieved with typical get accessors. A TableColumnModel fires a TableColumnModelEvent whenever a column move, removal, or addition occurs. The event source can be retrieved with getSource().

18.1.16 JTableHeader

class javax.swing.table.JTableHeader

This GUI component (which looks like a set of buttons for each column) is used to display a table's column headers. By dragging these headers the user can rearrange a table's columns dynamically. This component is used internally by JTable. It can be retrieved with JTable's getTableHeader() method, and assigned with setTableHeader(). When a JTable is placed in a JScrollPane a default JTableHeader corresponding to each column is added to that scroll pane's COLUMN_HEADER viewport (see 7.1.3). Each JTable uses one JTableHeader instance.

JTableHeader extends JComponent and implements TableColumnModelListener. Though JTableHeader is a Swing component, it is not used for display purposes. Instead, each TableColumn maintains a specific TableCellRenderer implementation used to represent its header. By default this is an instance of DefaultTableCellRenderer (see 18.1.8).

Note: It is more common to customize the header renderer of a TableColumn than it is to customize a table's JTableHeader. In most cases the default headers provided by JTable are satisfactory.

The resizingAllowed property specifies whether or not columns can be resized (if this is false it overpowers the isResizable property of each TableColumn). The reorderingAllowed property specifies whether or not columns can be reordered, and updateTableInRealTime property specifies whether or not the whole column is displayed along with the header as it is dragged (this in only applicable if reorderingAllowed is true). All three of these properties are true by default.

UI Guideline : column resizing It is best to try and isolate columns which need to be fixed width, for example the display of monetary amounts which might be 10 significant figures with two decimal places. Such a column requires a fixed width. It doesn't need to be bigger and it doesn't want to be smaller. Allow the other columns to vary in size around the fixed columns.

For example in a two column table displaying Product Description and Price, fix the size of Price and allow Description to resize.

UI Guideline : draggable columns, added flexibility, added complexity If you don't need the flexibility of draggable table columns then it is best to switch them off. If a User accidentally picks up a JHeader component and re-arranges a table, this could be confusing and upsetting. They may not realise what they have done or how to restore the table to its original form.

At any given time during a column drag we can retrieve the distance, in table coordinates, that the column has been dragged with respect to its original position from the draggedDistance property. JTableHeader also maintains a reference to the TableColumn it represents as well as the JTable it is part of -- the tableColumn and table properties respectively.

18.1.17 JTable selection

JTable supports two selection models: one for row selections and one for column selections. JTable also supports selection of individual table cells. Column selections are managed by a ListSelectionModel maintained by a TableColumnModel implementation, and row selections are managed by a ListSelectionModel maintained by JTable itself (both are DefaultListSelectionModels by default). As we learned in chapter 10, ListSelectionModels support three selection modes: SINGLE_SELECTION, SINGLE_INTERVAL_SELECTION, and MULTIPLE_INTERVAL_SELECTION. JTable provides the setSelectionMode() methods which will set both selection models to the given mode. Note, however, that getSelectionMode() only returns the current row selection mode.

To assign a specific selection mode to JTable's row ListSelectionModel:

myJTable.getSelectionModel().setSelectedMode(

ListSelectionModel.XX_SELECTION);

To assign a specific selection mode to JTable's column ListSelectionModel:

myJTable.getColumnModel().getSelectionModel().setSelectionMode(

ListSelectionModel.XX_SELECTION);

Row selection mode defaults to MULTIPLE_INTERVAL_SELECTION, and column selection mode defaults to SINGLE_SELECTION_MODE.

JTable provides control over whether rows and and columns can be selected, and we can query these modes, and turn them on/off, with getRowSelectionAllow(), getColumnSelectionAllowed(), and setRowSelectionAllowed(), setColumnSelectionAllowed() respectively. When row selection is enabled (true by default) and cell selction is disabled (see below), clicking on a cell will select the entire row that cell belongs to. Similarly, when column selection is enabled (false by default) the whole column that cell belongs to will be selected. There is nothing stopping us from having both row and column selection active simultaneously.

JTable also provides control over whether individual cells can be selected with its cellSelectionEnabled property. We can turn this on or off with setCellSelectionEnabled() and query its state using getCellSelectionEnabled(). If cell selection is enabled (false by default), a cell can only be selected if both row selection and column selection are also enabled (see below). If cell selection is not enabled, whenever a row or column containing that cell is selected (assuming that either row and/or column selection is enabled), that cell is also considered selected.

JTable provides several additional methods for querying the state of selection. If there is at least one cell selected:

getSelectedColumn() returns the index (in the TreeModel) of the most recently seleced column (-1 if no selection exists).

getSelectedRow() returns the index (in the TreeModel) of the most recently selected row (-1 if no selection exists).

getSelectedColumns() and getSelectedRows() return the TreeModel indices of all currently selected columns and rows respectively (int[0] if no selection exists).

getSelectedColumnCount() and getSelectedRowCount() return the current number of selected columns and rows respectively (int[0] if no selection exists).

isColumnSelected() and isRowSelected() return a boolean specifying whether or not the given column or row is currently selected.

isCellSelected() returns a boolean specifying whether or not the cell at the given TreeModel row and column index is selected.

The following methods can be used to programatically change JTable's selection, assuming the corresponding selection properties

are enabled:

clearSelection() unselects all rows, columns and cells.

selectAll() selects all rows, columns, and cells.

addColumnSelectionInterval() and addRowSelectionInterval() allow programmatic selection of a contiguous group of columns and rows respectively. Note that these can be called repeatedly to build of a multiple-inverval selection if the MULTIPLE_INTERVAL_SELECTION mode is active in the corresponding selection models.

removeColumnSelectionInterval() and removeRowSelectionInterval() allow programmatic de-selection of a contiguous interval of columns and rows respectively. These can also be used repeatedly to affect mutliple-interval selections.

setColumnSelectionInterval() and setRowSelectionInterval() clear the current column and row selection respectively, and select the specified contiguous interval.

Interestingly, when cell selection is enabled, JTable considers the columns and rows containing selected cells selected themselves (even though they aren't highlighted). For example, if cells (1,5) and (3,6) are selected with row and column selection enabled and cell selection enabled, getSelectedColumns() will return {5,6} and getSelectedRows() will return {1,3}. Oddly enough, those two cells will be highlighted and considered selected by JTable, along with cells (1,6) and (3,5)! This is due to the fact that JTable bases cell selection solely on whether or not both the row and column containing a cell are selected. **When selected rows and columns intersect, the cells at the intersection points are considered selected.**

If these same cells are selected when cell selection is disabled and row and column selection are enabled, all cells in rows 1 and 3, and all cells in columns 5 and 6 will be considered selected. If they are selected with cell selection and only row selection enabled, all cells in rows 1 and 3 will be considered selected. Similarly, if these two cells are selected with cell selection and only column selection enabled, all cells in columns 5 and 6 will be considered selected. If cell selection is *not* enabled, and row and/or column selection is enabled, a cell will be considered selected if either a column or row containing it is selected.

Note: Multiple single-cell selections can be made by holding down the CTRL key and using the mouse for selection.

Typically we are interested in determining cell, row, and/or column selection based on a mouse click. JTable supports MouseListeners just as any other JComponent, and we can used the getSelectedColumn() and getSelectedRow() methods to determine which cell was clicked in MouseListener's mouseClicked() method:

```
myJTable.addMouseListener(new MouseAdapter() {

public void mouseClicked(MouseEvent e) {

// get most recently selected row index

int row = getSelectedRow();

// get most recently selected column index

int column = getSelectedColumn();

if (row == -1 || column == -1)

return; // can't determine selected cell

else

// do something cell-specific

}

});
```

This listener is not very robust because it will only give us a cell if both a row and column have recently been selected, which in turn can only occur if both row selection and column selection is enabled. Thankfully, JTable provides methods for retrieving a row and column index corresponding to a given Point: rowAtPoint() and columnAtPoint() (these will return -1 if no row or column is found respectively). Since MouseEvent carries a Point specifying the location where the event occurred, we can use these methods in place of the getSelectedRow() and getSelectedColumn() methods. This is particularly useful when row,column, and/or cell selection is not enabled.

As with JList, JTable does not directly support double-mouse click selections. However, as we learned in chapter 10, we can capture a double click and determine which cell was clicked by adding a listener to JTable similar to the following:

```
myJTable.addMouseListener(new MouseAdapter() {
```

```
public void mouseClicked(MouseEvent e) {

if (e.getClickCount() == 2) {

Point origin = e.getPoint();

int row = myJTable.rowAtPoint(origin);

int column = myJTable.columnAtPoint(origin);

if (row == -1 || column == -1)

return; // no cell found

else

// do something cell-specific

}

}

});
```

18.1.18 Column Width and Resizing

When a column's width increases, JTable must decide how other columns will react. One or more columns must shrink. Similarly, when a column's width decreases, JTable must decide how other columns will react to the newly available amount of space. JTable's autoResizeMode property can take on any of five different values that handle these cases differently:

JTable.AUTO_RESIZE_ALL_COLUMNS: All columns gain or lose an equal amount of space corresponding the width lost or gained by the resizing column.

JTable.AUTO_RESIZE_LAST_COLUMN: The right-most column shrinks or grows in direct correspondence with the amount of width lost or gained from the column being resized. All other colmns are not affected.

JTable.AUTO_RESIZE_NEXT_COLUMN: The column to the immediate right of the column being resized shrinks or grows in direct correspondence with the amount of width lost or gained from the resizing column. All other colmns are not affected.

JTable.AUTO_RESIZE_OFF: resizing only affects the column being sized. All columns to the right of the column being resized are shifted right or left accordingly while maintaining their current sizes. Columns to the left are not affected.

JTable.AUTO_RESIZE_SUBSEQUENT_COLUMNS: All columns to the right of the column being resized gain or lose an equal amount of space corresponding the width lost or gained by the resizing column. Columns to the left are not affected.

TableColumn's width defaults to 75. Its minimum width defaults to 15 and its maximum width defaults to Integer.MAX_VALUE. When a JTable is first displayed it attempts to size each TableColumn according to its width property. If that table's autoResizeMode property is set to AUTO_RESIZE_OFF this will occur successfully. Otherwise, TableColumns are adjusted according to the current autoResizeMode property.

A TableColumn will never be sized larger than its maximum width or smaller than its minimum. For this reason it is possible that a JTable will occupy a larger or smaller area than that available (i.e. visible in the parent JScrollPane's main viewport), which may result in part of the table being clipped from view. If a table is contained in a JScrollPane and it occupies more than the available visible width, a horizontal scrollbar will be presented.

At any time we can call TableColumn's sizeWidthToFit() method to resize a column to occupy a width corresponding to the preferred width of its table header renderer. This is often used in assigning minimum widths for each TableColumn. JTable's sizeColumnsToFit() method takes an int parameter specifying the index of the TableColumn to act as the source of a resize in an attempt to make all columns fit within the available visible space. Also note that TableColumnModel's getTotalColumnWidth() method will return the sum of the current width of all TableColumns including all margin space.

We can specify the amount of empty space between rows with JTable's setRowMargin() method, and we can assign all rows a specific height with setRowHeight(). JTable's setIntercellSpacing() method takes a Dimension instance and uses it to assign a new width and height to use as margin space between cells (this method will redisplay the table it is invoked on after all sizes have been changed).

18.1.19 JTable Appearance

We can change the background and foreground colors used to highlight selected cells by setting the selectedBackground and setSelectedForeground properties. The default colors used for each TableColumn's table header renderer are determined from the current JTableHeader's background and foreground colors (recall that JTableHeader extends JComponent).

We can turn on and off horizontal and vertical grid lines (which are always have a thickness of 1 pixel) by changing the showHorizontalLines and showVerticalLines. properties. The showGrid property will overpower these properties when set with setShowGrid() becauuse this method reasigns them to the specified value. So setShowGrid() turns on/off both vertical and horizontal lines as specified. The gridColor property specifies the Color to use for both vertical and horizontal grid lines. setGridColor() will assign the specified value to this property and then repaint the whole table.

> **UI Guideline : Visual Noise** Grid Lines add visual noise to the display of a table. Removing some of them can aid the reading of the table data. If you intend the reader to read rows across then switch off the vertical grid lines. If you have columns of figures for example, then you might prefer to switch off the horizontal grid lines, making the columns easier to read.
>
> When switching off the horizontal grid lines on the table, you may wish to use the column cell renderer to change the background color of alternate rows on the table to ease the reading of rows. Using this combination of visual techniques, grid lines to distinguish columns and colour to distinguish rows, helps guide the reader to interpret data as required.

18.1.20 JTable scrolling

JTable implements the Scrollable interface (see 7.1.4) and is intended to be placed in a JScrollPane. JTableHeaders will not be displayed otherwise, disabling any column resize capabilities. Among the required Scrollable methods, JTable implements getScrollableTracksViewportWidth() to return true, which forces JTable to attempt to size itself horizontally to fit within the current scroll pane viewport width. getScrollableTracksViewportHeight(), however, returns false as it is most common for tables to be vertically scrolled but not horizontally scrolled. Horizontal scrolling is often awkward and we advise avoiding it whenever possible.

JTable's vertical block increment is the number of visible rows less one, and its vertical unit increment is the current row height. The horizontal block inrement is the width of the viewport, and the horizontal unit increment defaults to 100.

> **UI Guideline : small grids, no column headers** If you have a requirement to show two or three pieces of data, grouped and aligned together, consider using a JTable without JScrollPane. This gives you a small grid which is already aligned and neat and tidy for display without column headers.

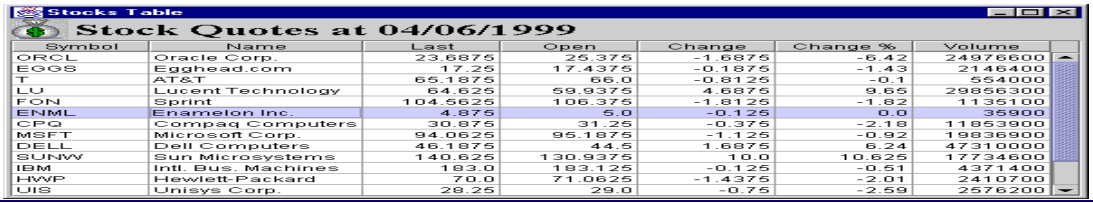
18.2 Stocks Table: part I - Basic JTable example

This basic example shows how to construct a JTable to display information about stock market data for a given day. Despite of its simplicity, it demonstrates the most fundamental features of JTable, and serves as a good basis for the more advanced examples that follow.

Stocks and stock trading is characterized by many attributes. The following are selected for display in our example:

Name	Type	Description
Symbol	String	Stock's symbol (NYSE or NASDAQ)
Name	String	Company name
Last	double	Price at the end of the trade day
Open	double	Price at the beginning of the trade day
Change	double	Absolute change in price with respect to previous closing
Change %	double	Percent change in price with respect to previous closing
Volume	long	Day's volume of trade (in $) for this stock

Each stock attribute represents a column in our table, and each row represents a specific company's stock information.



Symbol	Name	Last	Open	Change	Change %	Volume
ORCL	Oracle Corp.	23.6875	25.375	-1.6875	-6.42	24976600
EGGS	Egghead.com	17.25	17.4375	-0.1875	-1.43	2146400
T	AT&T	65.1875	66.0	-0.8125	-0.1	554000
LU	Lucent Technology	64.625	59.9375	4.6875	9.65	29956300
FON	Sprint	104.5625	106.375	-1.8125	-1.82	1135100
ENML	Enamelon Inc.	4.875	5.0	-0.125	0.0	35900
CPQ	Compaq Computers	30.875	31.25	-0.375	-2.18	11853900
MSFT	Microsoft Corp.	94.0625	95.1875	-1.125	-0.92	19836800
DELL	Dell Computers	46.1875	44.5	1.6875	6.24	47310000
SUNW	Sun Microsystems	140.625	130.9375	10.0	10.625	17734800
IBM	Intl. Bus. Machines	183.0	183.125	-0.125	-0.51	4371400
HWP	Hewlett-Packard	70.0	71.0625	-1.4375	-2.01	2410700
UIS	Unisys Corp.	28.25	29.0	-0.75	-2.59	2578200

Symbol	Name	Last	Open	Change	Change %	Volume
ORCL	Oracle Corp.	23.6875	25.375	-1.6875	-6.42	24976600
EGGS	Egghead.com	17.25	17.4375	-0.1875	-1.43	2146400
T	AT&T	65.1875	66.0	-0.8125	-0.1	554000
LU	Lucent Technology	64.625	59.9375	4.6875	9.65	29856300
FON	Sprint	104.5625	106.375	-1.8125	-1.82	1135100
ENML	Enamelon Inc.	4.875	5.0	-0.125	0.0	35900
CPQ	Compaq Computers	30.875	31.25	-0.375	-2.18	11853900
MSFT	Microsoft Corp.	94.0625	95.1875	-1.125	-0.92	19836900
DELL	Dell Computers	46.1875	44.5	1.6875	6.24	47310000
SUNW	Sun Microsystems	140.625	130.9375	10.0	10.625	17734600
IBM	Intl. Bus. Machines	183.0	183.125	-0.125	-0.51	4371400
HWP	Hewlett-Packard	70.0	71.0625	-1.4375	-2.01	2410700
UIS	Unisys Corp.	28.25	29.0	-0.75	-2.59	2576200

Figure 18.1 JTable in a JScrollPane with 7 TableColumns and 16 rows of data.

<<file figure18-1.gif>>

The Code: StocksTable.java

see \Chapter18\1

```
import java.awt.*;

import java.awt.event.*;

import java.util.*;

import java.io.*;

import java.text.*;

import javax.swing.*;

import javax.swing.border.*;

import javax.swing.event.*;

import javax.swing.table.*;

public class StocksTable extends JFrame

{

protected JTable m_table;

protected StockTableData m_data;

protected JLabel m_title;

public StocksTable() {

super("Stocks Table");

setSize(600, 340);

m_data = new StockTableData();

m_title = new JLabel(m_data.getTitle(),

new ImageIcon("money.gif"), SwingConstants.LEFT);

m_title.setFont(new Font("TimesRoman",Font.BOLD,24));

m_title.setForeground(Color.black);

getContentPane().add(m_title, BorderLayout.NORTH);

m_table = new JTable();

m_table.setAutoCreateColumnsFromModel(false);

m_table.setModel(m_data);
```

```
for (int k = 0; k < StockTableData.m_columns.length; k++) {
DefaultTableCellRenderer renderer = new
DefaultTableCellRenderer();
renderer.setHorizontalAlignment(
StockTableData.m_columns[k].m_alignment);
TableColumn column = new TableColumn(k,
StockTableData.m_columns[k].m_width, renderer, null);
m_table.addColumn(column);
}
JTableHeader header = m_table.getTableHeader();
header.setUpdateTableInRealTime(false);
JScrollPane ps = new JScrollPane();
ps.getViewport().add(m_table);
getContentPane().add(ps, BorderLayout.CENTER);
WindowListener wndCloser = new WindowAdapter() {
public void windowClosing(WindowEvent e) {
System.exit(0);
}
};
addWindowListener(wndCloser);
setVisible(true);
}
public static void main(String argv[]) {
new StocksTable();
}
}
class StockData
{
public String m_symbol;
public String m_name;
public Double m_last;
public Double m_open;
public Double m_change;
public Double m_changePr;
public Long m_volume;
public StockData(String symbol, String name, double last,
```

```
double open, double change, double changePr, long volume) {
m_symbol = symbol;
m_name = name;
m_last = new Double(last);
m_open = new Double(open);
m_change = new Double(change);
m_changePr = new Double(changePr);
m_volume = new Long(volume);
}
}
class ColumnData
{
public String m_title;
public int m_width;
public int m_alignment;
public ColumnData(String title, int width, int alignment) {
m_title = title;
m_width = width;
m_alignment = alignment;
}
}
class StockTableData extends AbstractTableModel
{
static final public ColumnData m_columns[] = {
new ColumnData( "Symbol", 100, JLabel.LEFT ),
new ColumnData( "Name", 150, JLabel.LEFT ),
new ColumnData( "Last", 100, JLabel.RIGHT ),
new ColumnData( "Open", 100, JLabel.RIGHT ),
new ColumnData( "Change", 100, JLabel.RIGHT ),
new ColumnData( "Change %", 100, JLabel.RIGHT ),
new ColumnData( "Volume", 100, JLabel.RIGHT )
};
protected SimpleDateFormat m_frm;
protected Vector m_vector;
protected Date m_date;
public StockTableData() {
```

```
m_frm = new SimpleDateFormat("MM/dd/yyyy");
m_vector = new Vector();
setDefaultData();
}
public void setDefaultData() {
try {
m_date = m_frm.parse("04/06/1999");
}
catch (java.text.ParseException ex) {
m_date = null;
}
m_vector.removeAllElements();
m_vector.addElement(new StockData("ORCL", "Oracle Corp.",
23.6875, 25.375, -1.6875, -6.42, 24976600));
m_vector.addElement(new StockData("EGGS", "Egghead.com",
17.25, 17.4375, -0.1875, -1.43, 2146400));
m_vector.addElement(new StockData("T", "AT&T",
65.1875, 66, -0.8125, -0.10, 554000));
m_vector.addElement(new StockData("LU", "Lucent Technology",
64.625, 59.9375, 4.6875, 9.65, 29856300));
m_vector.addElement(new StockData("FON", "Sprint",
104.5625, 106.375, -1.8125, -1.82, 1135100));
m_vector.addElement(new StockData("ENML", "Enamelon Inc.",
4.875, 5, -0.125, 0, 35900));
m_vector.addElement(new StockData("CPQ", "Compaq Computers",
30.875, 31.25, -0.375, -2.18, 11853900));
m_vector.addElement(new StockData("MSFT", "Microsoft Corp.",
94.0625, 95.1875, -1.125, -0.92, 19836900));
m_vector.addElement(new StockData("DELL", "Dell Computers",
46.1875, 44.5, 1.6875, 6.24, 47310000));
m_vector.addElement(new StockData("SUNW", "Sun Microsystems",
140.625, 130.9375, 10, 10.625, 17734600));
m_vector.addElement(new StockData("IBM", "Intl. Bus. Machines",
183, 183.125, -0.125, -0.51, 4371400));
m_vector.addElement(new StockData("HWP", "Hewlett-Packard",
70, 71.0625, -1.4375, -2.01, 2410700));
```

```
m_vector.addElement(new StockData("UIS", "Unisys Corp.",
28.25, 29, -0.75, -2.59, 2576200));
m_vector.addElement(new StockData("SNE", "Sony Corp.",
96.1875, 95.625, 1.125, 1.18, 330600));
m_vector.addElement(new StockData("NOVL", "Novell Inc.",
24.0625, 24.375, -0.3125, -3.02, 6047900));
m_vector.addElement(new StockData("HIT", "Hitachi, Ltd.",
78.5, 77.625, 0.875, 1.12, 49400));
}
public int getRowCount() {
return m_vector==null ? 0 : m_vector.size();
}
public int getColumnCount() {
return m_columns.length;
}
public String getColumnName(int column) {
return m_columns[column].m_title;
}
public boolean isCellEditable(int nRow, int nCol) {
return false;
}
public Object getValueAt(int nRow, int nCol) {
if (nRow < 0 || nRow >= getRowCount())
return "";
StockData row = (StockData)m_vector.elementAt(nRow);
switch (nCol) {
case 0: return row.m_symbol;
case 1: return row.m_name;
case 2: return row.m_last;
case 3: return row.m_open;
case 4: return row.m_change;
case 5: return row.m_changePr;
case 6: return row.m_volume;
}
return "";
}
```

```
public String getTitle() {
if (m_date==null)
return "Stock Quotes";
return "Stock Quotes at "+m_frm.format(m_date);
}
}
```

Understanding the Code

Class StocksTable

This class extends JFrame to implement the frame container for our table. Three instance variables are declared (to be used extensively in more complex examples that follow):

JTable m_table: table component to display stock data.

StockTableData m_data: TableModel implementation to manage stock data (see below).

JLabel m_title: used to display stocks table title (date which stock prices are referenced).

The StocksTable constructor first initializes the parent frame object and builds an instance of StockTableData. Method getTitle()is invoked to set the text for the title label which is added to the northern region of the content pane. Then a JTable is created by passing the StockTableData instance to the constructor. Note the the autoCreateColumnsFromModel method is set to false because we plan on creating our own TableColumns.

As we will see below, the static array m_columns of the StockTableData class describes all columns of our table. It is used here to create each TableColumn instance and set their text alignment and width.

Method setHorizontalAlignment() (inherited by DefaultTableCellRenderer from JLabel) is used to set the proper alignment for each TableColumn's cell renderer. The TableColumn constructor takes a column index, width, and renderer as parameters. Note that TableCellEditor is set to null since we don't want to allow editing of stock data. Finally, columns are added to the table's TableColumnModel (which JTable created by default because we didn't specify one) with the addColumn() method.

In the next step, an instance of JTableHeader is created for this table, and the updateTableInRealTime property is set to false (this is done to demonstrate the effect this has on column dragging -- only a column's table header is displayed during a drag).

Lastly a JScrollPane instance is used to provide scrolling capabilities, and our table is added to its JViewport. This JScrollPane is then added to the center of our frame's content pane.

Class StockData

This class encapsulates a unit of stock data as described in the table above. The instance variables defined in this class have the following meaning:

String m_symbol: stock's symbol (NYSE or NASDAQ)

String m_name: company name

Double m_last: the price of the last trade

Double m_open: price at the beginning of the trade day

Double m_change: absolute change in price with respect to previous closing

Double m_changePr: percent change in price with respect to previous closing

Long m_volume: day's volume of trade (in $) for this stock

Note that all numerical data are encapsulated in Object-derived classes. This design decision simplifies data exchange with the table (as we will see below). The only constructor provided assigns each of these variables from the data passed as parameters.

Note: We use public instance variables in this and several other classes in this chapter to avoid overcomplication. In most professional apps these would either be protected or private.

Class ColumnData

This class encapsulates data describing the visual characteristics of a single TableColumn of our table. The instance variables defined in this class have the following meaning:

String m_title: column title

int m_width: column width in pixels

int m_alignment: text alignment as defined in JLabel

The only constructor provided assigns each of these variables the data passed as parameters.

Class StockTableData

This class extends AbstractTableModel to serve as the data model for our table. Recall that AbstractTableModel is an abstract class, and three methods must be implemented to instantiate it:

public int getRowCount(): returns the number of rows in the table.

public int getColumnCount(): returns the number of columns in the table.

public Object getValueAt(int row, int column): returns data in the specified cell as an Object instance.

Note: An alternative approach is to extend the DefaultTableModel class which is a concrete implementation of AbstractTableModel. However, this is not recommended, as the few abstract methods in AbstractTableModel can be easily implemented. Usage of DefaultTableModel often creates unnecessary overhead.

By design, this class manages all information about our table, including the title and column data. A static array of ColumnData, m_columns, is provided to hold information about our table's columns (it is used in the StocksTable constructor, see above). Three instance variables have the following meaning:

SimpleDateFormat m_frm: used to format dates

Date m_date: date of currently stored market data

Vector m_vector: collection of StockData instances for each row in the table

The only constructor of the StockTableData class initializes two of these variables and calls the setDefaultData() method to assign the pre-defined default data to m_date and m_vector (in a later example we'll see how to use JDBC to retrieve data from a database rather than using hard-coded data as we do here).

As we discussed above, the getRowCount() and getColumnCount() methods should return the number of rows and columns, respectively. So their implementation is fairly obvious. The only catch is that they may be called by the AbstractTableModel constructor *before* any member variable is initialized. So we have to check for a null instance of m_vector. Note that m_columns, as a static variable, will be initialized before any non-static code is executed (so we don't have to check m_columns against null).

The remainder of the StockTableData class implements the following methods:

getColumnName(): returns the column title.

isCellEditable(): always returns false, because we want to disable all editing.

getValueAt(): retrieves data for a given cell as an Object. Depending on the column index, one of the StockData fields is returned.

getTitle(): returns our table's title as a String to be used in a JLabel in the northern region of our frame's content pane.

Running the Code

Figure 18.1 shows StocksTable in action displaying our hard-coded stock data. Note that the TableColumns resize properly in response to the parent frame size. Also note that the selected row in our table can be moved by changed with the mouse or arrow keys, but no editing is allowed.

18.3 Stocks Table: part II - Custom renderers

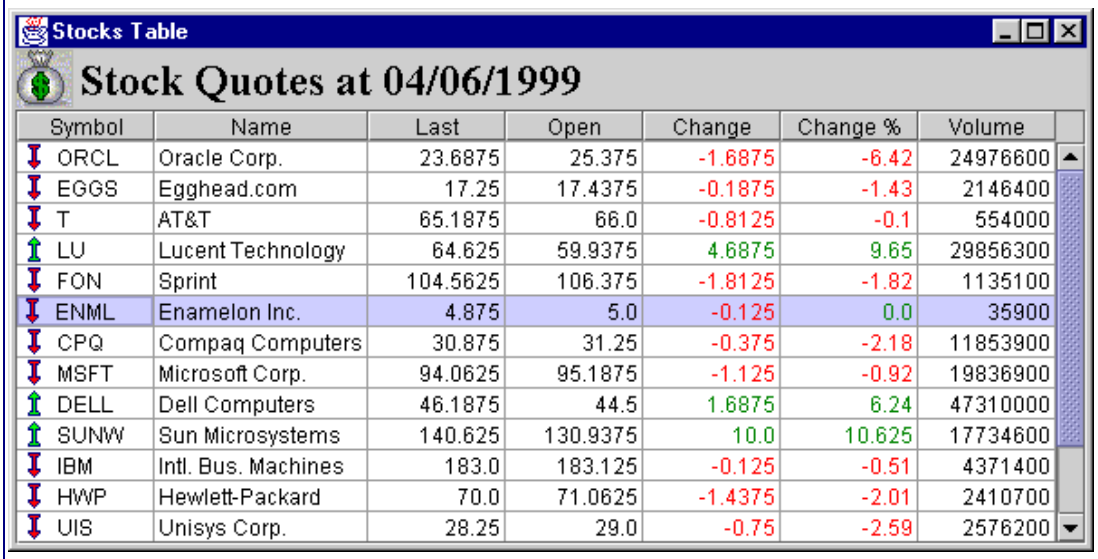
Now we'll extend StocksTable to use color and small icons in rendering our table cells. To enhance data visibility, we'll make the

following two enhancements:

Render absolute and percent changes in green for positive values and red for negative values.

Add an icon next to each stock symbol: arrow up for positive changes and arrow down for negative.

To do this we need to build our own custom TreeCellRenderer.



Symbol	Name	Last	Open	Change	Change %	Volume
ORCL	Oracle Corp.	23.6875	25.375	-1.6875	-6.42	24976600
EGGS	Egghead.com	17.25	17.4375	-0.1875	-1.43	2146400
T	AT&T	65.1875	66.0	-0.8125	-0.1	554000
LU	Lucent Technology	64.625	59.9375	4.6875	9.65	29856300
FON	Sprint	104.5625	106.375	-1.8125	-1.82	1135100
ENML	Enamelon Inc.	4.875	5.0	-0.125	0.0	35900
CPQ	Compaq Computers	30.875	31.25	-0.375	-2.18	11853900
MSFT	Microsoft Corp.	94.0625	95.1875	-1.125	-0.92	19836900
DELL	Dell Computers	46.1875	44.5	1.6875	6.24	47310000
SUNW	Sun Microsystems	140.625	130.9375	10.0	10.625	17734600
IBM	Intl. Bus. Machines	183.0	183.125	-0.125	-0.51	4371400
HWP	Hewlett-Packard	70.0	71.0625	-1.4375	-2.01	2410700
UIS	Unisys Corp.	28.25	29.0	-0.75	-2.59	2576200

Figure 18.2 JTable using a custom cell renderer.

<<file figure18-2.gif>>

The Code: StocksTable.java

see \Chapter18\2

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import java.text.*;
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import javax.swing.table.*;
public class StocksTable extends JFrame
{
// Unchanged code from section 18.2
public StocksTable() {
// Unchanged code from section 18.2
for (int k = 0; k < StockTableData.m_columns.length; k++) {
DefaultTableCellRenderer renderer = new
ColoredTableCellRenderer();
renderer.setHorizontalAlignment(
```

```
StockTableData.m_columns[k].m_alignment);
TableColumn column = new TableColumn(k,
StockTableData.m_columns[k].m_width, renderer, null);
m_table.addColumn(column);
}
// Unchanged code from section 18.2
}
public static void main(String argv[]) {
new StocksTable();
}
}
class ColoredTableCellRenderer extends DefaultTableCellRenderer
{
public void setValue(Object value) {
if (value instanceof ColorData) {
ColorData cvalue = (ColorData)value;
setForeground(cvalue.m_color);
setText(cvalue.m_data.toString());
}
else if (value instanceof IconData) {
IconData ivalue = (IconData)value;
setIcon(ivalue.m_icon);
setText(ivalue.m_data.toString());
}
else
super.setValue(value);
}
}
class ColorData
{
public Color m_color;
public Object m_data;
public static Color GREEN = new Color(0, 128, 0);
public static Color RED = Color.red;
public ColorData(Color color, Object data) {
m_color = color;
```

```
m_data = data;
}
public ColorData(Double data) {
m_color = data.doubleValue() >= 0 ? GREEN : RED;
m_data = data;
}
public String toString() {
return m_data.toString();
}
}
class IconData
{
public ImageIcon m_icon;
public Object m_data;
public IconData(ImageIcon icon, Object data) {
m_icon = icon;
m_data = data;
}
public String toString() {
return m_data.toString();
}
}
class StockData
{
public static ImageIcon ICON_UP = new ImageIcon("ArrUp.gif");
public static ImageIcon ICON_DOWN = new ImageIcon("ArrDown.gif");
public static ImageIcon ICON_BLANK = new ImageIcon("blank.gif");
public IconData m_symbol;
public String m_name;
public Double m_last;
public Double m_open;
public ColorData m_change;
public ColorData m_changePr;
public Long m_volume;
public StockData(String symbol, String name, double last,
double open, double change, double changePr, long volume) {
```

```
m_symbol = new IconData(getIcon(change), symbol);
m_name = name;
m_last = new Double(last);
m_open = new Double(open);
m_change = new ColorData(new Double(change));
m_changePr = new ColorData(new Double(changePr));
m_volume = new Long(volume);
}
public static ImageIcon getIcon(double change) {
return (change>0 ? ICON_UP : (change<0 ? ICON_DOWN :
ICON_BLANK));
}
}
// Class StockTableData unchanged from section 18.2
```

Understanding the Code

Class StocksTable

The only change we need to make in the base frame class is to change the column renderer to a new class, ColoredTableCellRenderer. ColoredTableCellRenderer should be able to draw icons and colored text (but not both at the same time -- although this could be done using this same approach).

Class ColoredTableCellRenderer

This class extends DefaultTableCellRenderer and overrides only one method: setValue(). This method will be called prior to the rendering of a cell to retrieve its corresponding data (of any nature) as an Object. Our overridden setValue() method is able to recognize two specific kinds of cell data: ColorData, which adds color to a data object, and IconData, which adds an icon (both are described below). If a ColorData instance is detected, its encapsulated color is set as the foreground for the renderer. If an IconData instance is detected, its encapsulated icon is assigned to the renderer with the setIcon() method (which is inherited from JLabel). If the value is neither a ColorData or an IconData instance we call the super-class setValue() method.

Class ColorData

This class is used to bind a specific color, m_color, to a data object of any nature, m_data. Two static Colors, RED and GREEN, are declared to avoid creation of numerous temporary objects. Two constructors are provided for this class. The first constructor takes Color and Object parameters and assigns them to instance variables m_color and m_data respectively. The second constructor takes a Double parameter which gets assigned to m_data, and m_color is assigned the green color if the parameter is positive, and red if negative. The toString() method simply calls the toString() method of the data object.

Class IconData

This class is used to bind ImageIcon m_icon to a data object of any nature, m_data. Its only constructor takes ImageIcon and Object parameters. The toString() method simply calls the toString() method of the data object.

Class StockData

This class has been enhanced from its previous version to to provide images and new variable data types. We've prepared three static ImageIcon instances holding images: arrow up, arrow down, and a blank (all transparent) image. The static getIcon() method returns one of these images depending on the sign of the given double parameter. We've also changed three instance variables to bind data with the color and image attributes according to the following table:

Field	New type	Data object	Description
m_symbol	IconData	String	Stock's symbol (NYSE or NASDAQ)
m_change	ColorData	Double	Absolute change in price

m_changePr	ColorData	Double	Percent change in price

The corresponding changes are also required in the StockData constructor.

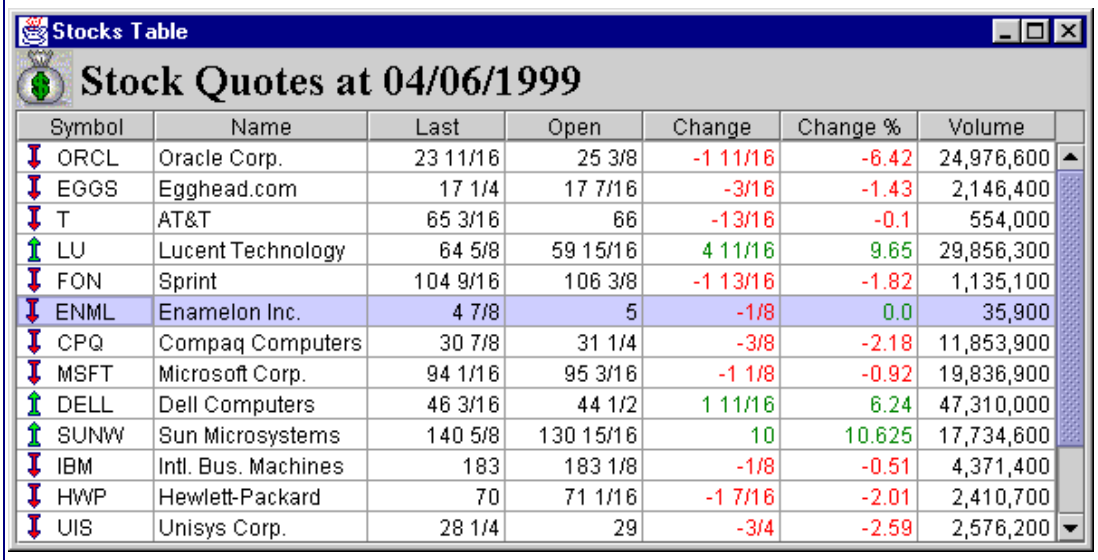
Running the Code

Figure 18.2 shows StocksTable with custom rendering in action. Note the correct usage of color and icons, which considerably enhances the visualization of our data.

> **UI Guideline : Improving visual communication** Tables can be data intensive and consequently it can be very difficult for the viewer to quickly pick out the important information. The table in fig 18.1 highlighted this. In fig 18.2, we are improving the visual communication with the introduction of visual layers. The icons in the first column quickly tell the viewer whether a price is rising or falling. This is visually re-inforced with the red and green introduced on the change columns.
>
> Red particularly is a very strong color. By introducing red and green only on the change columns and not across the entire row, we avoid the danger of the red becoming overpowering. If we had introduced red and green across the full width of the table, the colors may have become intrusive and impaired the visual communication.

18.4 Stocks Table: part III - Data formatting

To further enhance the presentation of our stock data, in this section we wll take into account that actual stock prices (at least on NYSE and NASDAQ) are expressed in fractions of 32, not in decimals. Another issue that we will deal with is the volume of trade, which can reach hundreds of millions of dollars. Volume is not immediately legible without separating thousands and millions places with commas.



Symbol	Name	Last	Open	Change	Change %	Volume
ORCL	Oracle Corp.	23 11/16	25 3/8	-1 11/16	-6.42	24,976,600
EGGS	Egghead.com	17 1/4	17 7/16	-3/16	-1.43	2,146,400
T	AT&T	65 3/16	66	-13/16	-0.1	554,000
LU	Lucent Technology	64 5/8	59 15/16	4 11/16	9.65	29,856,300
FON	Sprint	104 9/16	106 3/8	-1 13/16	-1.82	1,135,100
ENML	Enamelon Inc.	4 7/8	5	-1/8	0.0	35,900
CPQ	Compaq Computers	30 7/8	31 1/4	-3/8	-2.18	11,853,900
MSFT	Microsoft Corp.	94 1/16	95 3/16	-1 1/8	-0.92	19,836,900
DELL	Dell Computers	46 3/16	44 1/2	1 11/16	6.24	47,310,000
SUNW	Sun Microsystems	140 5/8	130 15/16	10	10.625	17,734,600
IBM	Intl. Bus. Machines	183	183 1/8	-1/8	-0.51	4,371,400
HWP	Hewlett-Packard	70	71 1/16	-1 7/16	-2.01	2,410,700
UIS	Unisys Corp.	28 1/4	29	-3/4	-2.59	2,576,200

Figure 18.3 JTable with custom number formatting cell renderer to display fractions and comma-delimited numbers.

<<file figure18-3.gif>>

The Code: StocksTable.java

see \Chapter18\3

```
import java.awt.*;

import java.awt.event.*;

import java.util.*;

import java.io.*;

import java.text.*;

import javax.swing.*;
```

```
import javax.swing.border.*;
import javax.swing.event.*;
import javax.swing.table.*;
// Unchanged code from section 18.3
class Fraction
{
public int m_whole;
public int m_nom;
public int m_den;
public Fraction(double value) {
int sign = value <0 ? -1 : 1;
value = Math.abs(value);
m_whole = (int)value;
m_den = 32;
m_nom = (int)((value-m_whole)*m_den);
while (m_nom!=0 && m_nom%2==0) {
m_nom /= 2;
m_den /= 2;
}
if (m_whole==0)
m_nom *= sign;
else
m_whole *= sign;
}
public double doubleValue() {
return (double)m_whole + (double)m_nom/m_den;
}
public String toString() {
if (m_nom==0)
return ""+m_whole;
else if (m_whole==0)
return ""+m_nom+"/"+m_den;
else
return ""+m_whole+" "+m_nom+"/"+m_den;
}
}
```

```
class SmartLong
{
protected static NumberFormat FORMAT;
static {
FORMAT = NumberFormat.getInstance();
FORMAT.setGroupingUsed(true);
}
public long m_value;
public SmartLong(long value) { m_value = value; }
public long longValue() { return m_value; }
public String toString() { return FORMAT.format(m_value); }
}
class ColorData
{
public ColorData(Fraction data) {
m_color = data.doubleValue() >= 0 ? GREEN : RED;
m_data = data;
}
// Unchanged code from section 18.3
}
class StockData
{
public static ImageIcon ICON_UP = new ImageIcon("ArrUp.gif");
public static ImageIcon ICON_DOWN = new ImageIcon("ArrDown.gif");
public static ImageIcon ICON_BLANK = new ImageIcon("blank.gif");
public IconData m_symbol;
public String m_name;
public Fraction m_last;
public Fraction m_open;
public ColorData m_change;
public ColorData m_changePr;
public SmartLong m_volume;
public StockData(String symbol, String name, double last,
double open, double change, double changePr, long volume) {
m_symbol = new IconData(getIcon(change), symbol);
m_name = name;
```

```
m_last = new Fraction(last);
m_open = new Fraction(open);
m_change = new ColorData(new Fraction(change));
m_changePr = new ColorData(new Double(changePr));
m_volume = new SmartLong(volume);
}
// Unchanged code from section 18.3
}
```

Understanding the Code

Class Fraction

This new data class encapsulates fractions with denominator 32 (or in a reduced form). Three instance variables represent the whole number, numerator, and denominator of the fraction. The only constructor takes a double parameter and carefully extracts these values, performing numerator and denominator reduction, if possible. Note that negative absolute values are taken into account.

Method doubleValue() performs the opposite task: it converts the fraction into a double value. Method toString() forms a String representation of the fraction.

Note that a zero whole number or a zero nominator are omitted to avoid unseemly output like "0 1/2" or "12 0/32".

Class SmartLong

This class encapsulates long values. The only constructor takes a long as parameter and stores it in the m_value instance variable. The real purpose of creating this class is the overridden toString() method, which inserts commas separating thousands, millions, etc. places. For this purpose we use the java.text.NumberFormat class. An instance of this class is created as a static variable, and formatting values using NumberFormat's format() method couldn't be easier.

Note: SmartLong cannot extend Long (although it would be natural), because java.lang.Long is a final class.

Class ColorData

This class requires a new constructor to cooperate with the new Fraction data type. This third constructor takes a Fraction instance as parameter and uses its color attribute in the same way it did previously: green for positive values, and red for negative values.

Class StockData

This class uses new data types for its instance variables. The list of instance variables now looks like the following:

Field	Type	Data object	Description
m_symbol	IconData	String	Stock symbol (NYSE or NASDAQ)
m_name	String	N/A	Company name
m_last	Fraction	N/A	The price of the last trade
m_open	Fraction	N/A	The price at the beginning of the trade day
m_change	ColorData	Fraction	Absolute change in price
m_changePr	ColorData	Double	Percent change in price
m_volume	SmartLong	N/A	Day's volume of trade (in $) for this stock

The StockData constructor is modified accordingly. Meanwhile the parameters in the StockData constructor have *not* changed, so there is no need to make any changes to the data model class using StockData.

Note. We could use the different approach of just formatting the initial contents of our table's cells into text strings and operating on them without introducing new data classes. This approach, however, is not desirable from a design point of

view, as it strips the data of its true nature: numbers. Using our method we are still able to retrieve number values, if necessary, through the doubleValue() or longValue() methods.

Running the Code

Figure 18.3 shows StocksTable in action with number formatting. Presented this way, our data looks much more familiar to most of those who follow the stock trade on a regular basis.

UI Guideline : Talking the Users language In this example, we have changed the rendering of the stock prices into fractions rather than decimals. This is a good example of providing better visual communication by speaking the same language as the viewers. In the North American stock markets, prices are quoted in fractional amounts of dollars rather than dollars and cents. Switching to this type of display helps the application to communicate more quickly and more influentially to the viewer, improving usability.

18.5 Stocks Table: part IV - sorting columns

Note: This and the following StocksTable examples require Java 2 as they make use of the new java.util.Collections functionality.

In this section we add the ability to sort any column in ascending or descending order. The most suitable graphical element for selection of sort order are the column headers. We adopt the following model for our sorting functionality:

1. A single click on the header of a certain column causes the table to re-sort based on this column.
2. A repeated click on the same column changes the sort direction from ascending to descending and vice versa.
3. The header of the column which provides the current sorting should be marked.

To do this we add a mouse listener to the table header to capture mouse clicks and trigger a table sort. Fortunately sorting can be accomplished fairly easily using the new Collections functionality in Java 2.

Note: Class java.util.Collections contains a set of static methods used to manipulate Java collections, including java.util.Vector which is used in this example.

We use the java.util.Collections.sort(List lst, Comparator c) method to sort any collection implementing the java.util.List interface based on a given Comparator. A Comparator implementation requires two methods:

int compare(Object o1, Object o2): Compares two objects and returns the result as an int (zero if equal, negative value if the first is less than the second, positive value if the first is more than the second).

boolean equals(Object obj): Returns true if the given object is equal to this Comparator.

Stocks Table

Stock Quotes at 04/06/1999

Name	Last	Change % «	Symbol	Open	Change	Volume
Sun Microsystems	140 5/8	10.625	SUNW	130 15/16	10	17,734,600
Lucent Technology	64 5/8	9.65	LU	59 15/16	4 11/16	29,856,300
Dell Computers	46 3/16	6.24	DELL	44 1/2	1 11/16	47,310,000
Sony Corp.	96 3/16	1.18	SNE	95 5/8	1 1/8	330,600
Hitachi, Ltd.	78 1/2	1.12	HIT	77 5/8	7/8	49,400
Enamelon Inc.	4 7/8	0.0	ENML	5	-1/8	35,900
AT&T	65 3/16	-0.1	T	66	-13/16	554,000
Intl. Bus. Machines	183	-0.51	IBM	183 1/8	-1/8	4,371,400
Microsoft Corp.	94 1/16	-0.92	MSFT	95 3/16	-1 1/8	19,836,900
Egghead.com	17 1/4	-1.43	EGGS	17 7/16	-3/16	2,146,400
Sprint	104 9/16	-1.82	FON	106 3/8	-1 13/16	1,135,100
Hewlett-Packard	70	-2.01	HWP	71 1/16	-1 7/16	2,410,700
Compaq Computers	30 7/8	-2.18	CPQ	31 1/4	-3/8	11,853,900

Figure 18.4 JTable with ascending and descending sorting of all columns.

<<file figure18-4.gif>>

The Code: StocksTable.java

see \Chapter18\4

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import java.text.*;
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import javax.swing.table.*;
public class StocksTable extends JFrame
{
// Unchanged code from section 18.4
public StocksTable() {
// Unchanged code from section 18.4
JTableHeader header = m_table.getTableHeader();
header.setUpdateTableInRealTime(true);
header.addMouseListener(m_data.new ColumnListener(m_table));
header.setReorderingAllowed(true);
// Unchanged code from section 18.4
}
public static void main(String argv[]) {
new StocksTable();
}
}
// Unchanged code from section 18.4
class StockTableData extends AbstractTableModel
{
// Unchanged code from section 18.2
protected SimpleDateFormat m_frm;
protected Vector m_vector;
protected Date m_date;
protected int m_sortCol = 0;
```

```
protected boolean m_sortAsc = true;
public StockTableData() {
m_frm = new SimpleDateFormat("MM/dd/yyyy");
m_vector = new Vector();
setDefaultData();
}
public void setDefaultData() {
// Unchanged code from section 18.4
Collections.sort(m_vector, new
StockComparator(m_sortCol, m_sortAsc));
}
// Unchanged code from section 18.4
public String getColumnName(int column) {
String str = m_columns[column].m_title;
if (column==m_sortCol)
str += m_sortAsc ? " »" : " «";
return str;
}
// Unchanged code from section 18.4
class ColumnListener extends MouseAdapter
{
protected JTable m_table;
public ColumnListener(JTable table) {
m_table = table;
}
public void mouseClicked(MouseEvent e) {
TableColumnModel colModel = m_table.getColumnModel();
int columnModelIndex = colModel.getColumnIndexAtX(e.getX());
int modelIndex = colModel.getColumn(columnModelIndex).getModelIndex();
if (modelIndex < 0)
return;
if (m_sortCol==modelIndex)
m_sortAsc = !m_sortAsc;
else
m_sortCol = modelIndex;
for (int i=0; i < m_columns.length; i++) {
```

```
TableColumn column = colModel.getColumn(i);
column.setHeaderValue(getColumnName(column.getModelIndex()));
}
m_table.getTableHeader().repaint();
Collections.sort(m_vector, new
StockComparator(modelIndex, m_sortAsc));
m_table.tableChanged(
new TableModelEvent(StockTableData.this));
m_table.repaint();
}
}
}
class StockComparator implements Comparator
{
protected int m_sortCol;
protected boolean m_sortAsc;
public StockComparator(int sortCol, boolean sortAsc) {
m_sortCol = sortCol;
m_sortAsc = sortAsc;
}
public int compare(Object o1, Object o2) {
if(!(o1 instanceof StockData) || !(o2 instanceof StockData))
return 0;
StockData s1 = (StockData)o1;
StockData s2 = (StockData)o2;
int result = 0;
double d1, d2;
switch (m_sortCol) {
case 0: // symbol
String str1 = (String)s1.m_symbol.m_data;
String str2 = (String)s2.m_symbol.m_data;
result = str1.compareTo(str2);
break;
case 1: // name
result = s1.m_name.compareTo(s2.m_name);
break;
```

```
case 2: // last
d1 = s1.m_last.doubleValue();
d2 = s2.m_last.doubleValue();
result = d1<d2 ? -1 : (d1>d2 ? 1 : 0);
break;
case 3: // open
d1 = s1.m_open.doubleValue();
d2 = s2.m_open.doubleValue();
result = d1<d2 ? -1 : (d1>d2 ? 1 : 0);
break;
case 4: // change
d1 = ((Fraction)s1.m_change.m_data).doubleValue();
d2 = ((Fraction)s2.m_change.m_data).doubleValue();
result = d1<d2 ? -1 : (d1>d2 ? 1 : 0);
break;
case 5: // change %
d1 = ((Double)s1.m_changePr.m_data).doubleValue();
d2 = ((Double)s2.m_changePr.m_data).doubleValue();
result = d1<d2 ? -1 : (d1>d2 ? 1 : 0);
break;
case 6: // volume
long l1 = s1.m_volume.longValue();
long l2 = s2.m_volume.longValue();
result = l1<l2 ? -1 : (l1>l2 ? 1 : 0);
break;
}
if (!m_sortAsc)
result = -result;
return result;
}
public boolean equals(Object obj) {
if (obj instanceof StockComparator) {
StockComparator compObj = (StockComparator)obj;
return (compObj.m_sortCol==m_sortCol) &&
(compObj.m_sortAsc==m_sortAsc);
}
```

```
return false;

}

}
```

Understanding the Code

Class StocksTable

In the StocksTable constructor we set the updateTableInRealTime property to show column contents while columns are dragged, and we add an instance of the ColumnListener class (see below) as a mouse listener to the table's header.

Class StockTableData

Here we declare two new instance variables: int m_sortCol to hold the index of the current column chosen for sorting, and boolean m_sortAsc, which is true when sorting in ascending order, and false when sorting in descending order. These variables determine the initial sorting order. To be consistent we sort our table initially by calling the Collections.sort() method in our setDefaultData() method (which is called from the StockTableData constructor).

We also add a special marker for the sorting column's header: '»' for ascending and '«' for descending sorting. This changes the way we retrieve a column's name, which no longer is a constant value:

```
public String getColumnName(int column) {

String str = m_columns[column].m_title;

if (column==m_sortCol)

str += m_sortAsc ? " »" : " «";

return str;

}
```

Class StockTableData.ColumnListener

Because this class interacts heavily with our table data, it is implemented as an inner class in StockTableData. ColumnListener takes a reference to a JTable in its constructor and stores that reference in its m_table instance variable.

The mouseClicked() method is invoked when the user clicks on a header. First it determines the index of the TableColumn clicked based on the coordinate of the click. If for any reason the returned index is negative (i.e. the column cannot be determined) the method cannot continue and we return. Otherwise, we check whether this index corresponds to the column which already has been selected for sorting. If so, we invert the m_sortCol flag to reverse the sorting order. If the index corresponds to newly selected column we store the new sorting index in the m_sortCol variable.

Then we refresh the header names by iterating through the TableColumns and assinging them a name corresponding to the column they represent in the TableModel. To do this we pass each TableColumn's modelIndex property to our getColumnName() method (see above). Finally our table data is re-sorted by calling the Collections.sort() method and passing in a new StockComparator object. We then refresh the table by calling tableChanged() and repaint().

Class StockComparator

This class implements the rule of comparison for two objects, which in our case are StockDatas. Instances of the StockComparator class are passed to the Collections.sort() method to perform data sorting.

Two instance variables are defined:

int m_sortCol represents the index of the column which performs the comparison

boolean m_sortAsc is true for ascending sorting and false for descending.

The StockComparator constructor takes two parameters and stores them in these instance variables.

The compare() method takes two objects to be compared and returns an integer value according to the rules determined in the Comparator interface:

0 if object 1 equals 2

A positive number if object 1 is greater than object 2,

A negative number if object 1 is less than object 2.

Since we are dealing only with StockData objects, first we cast both objects and return 0 if this cast isn't possible. The next issue is to define what it means when one StockData objects is greater, equal, or less than another. This is done in a switch-case structure, which, depending on the index of the comparison column, extracts two fields and forms an integer result of the comparison. When the switch-case structure finishes, we know the result of an ascending comparison. For descending comparison we simply need to invert the sign of the result.

The equals() method takes another Comparator instance as parameter and returns true if that parameter represents the same Comparator. We determine this by comparing Comparator instance variables: m_sortCol and m_sortAsc.

Running the Code

Figure 18.4 shows StocksTable sorted by decreasing "Change %." Click different column headers and note that resorting occurs as expected. Click the same column header twice and note that sorting order flips from ascending to descending and vice versa. Also note that the currently selected sorting column header is marked by the '»' or '«' symbol. This sorting functionality is very useful. Particularly, for stock market data we can instantly determine which stocks have the highest price fluctuations or the most heavy trade.

> **UI Guideline : Sort by header selection idiom** Introducing table sorting using the column headers is introducing another design idiom to the User Interface. This design idiom is becoming widely accepted and widely used in many applications. It is a useful and powerful technique which you can introduce when sorting table data is a requirement. The technique is not intuitive and their is little visual affordance to suggest that clicking a column header will have any effect. So consider that the introduction of this technique may require additional User training.

18.6 Stocks Table: part V - JDBC

Despite all of our sorting functionality and enhanced data display, our application is still quite boring because it displays only data for a pre-defined day! Of course, in the real world we need to connect such an application to the source of fresh information such as a database. Very often tables are used to display data retrieved from databases, or to edit data to be stored in databases. In this section we show how to feed our StocksTable data extracted from a database using the Java Database Connectivity (JDBC) API.

First, we need to create the database. We chose to use two SQL tables (do not confuse SQL table with JTable) whose structure precisely corresponds to the market data structure described in section 18.2:

Table SYMBOLS

Field Name	**Type**
symbol	Text
name	Text

Table DATA

Field Name	**Type**
symbol	Text
date1	Date/Time
last	Number
change	Number
changeproc	Number
open	Number
volume	Number

For this example we use the JDBC-ODBC bridge which is a standard part of JDK since the 1.1 release and links Java programs to Microsoft Access databases. If you are using another database engine, you can work with this example as well, but you must make sure that the structure of your tables is the same. Before running the example in a Windows environment we need to register a database in an ODBC Data Source Administrator which is accessible through the Control Panel (this is not a JDBC tutorial, so we'll skip the details).

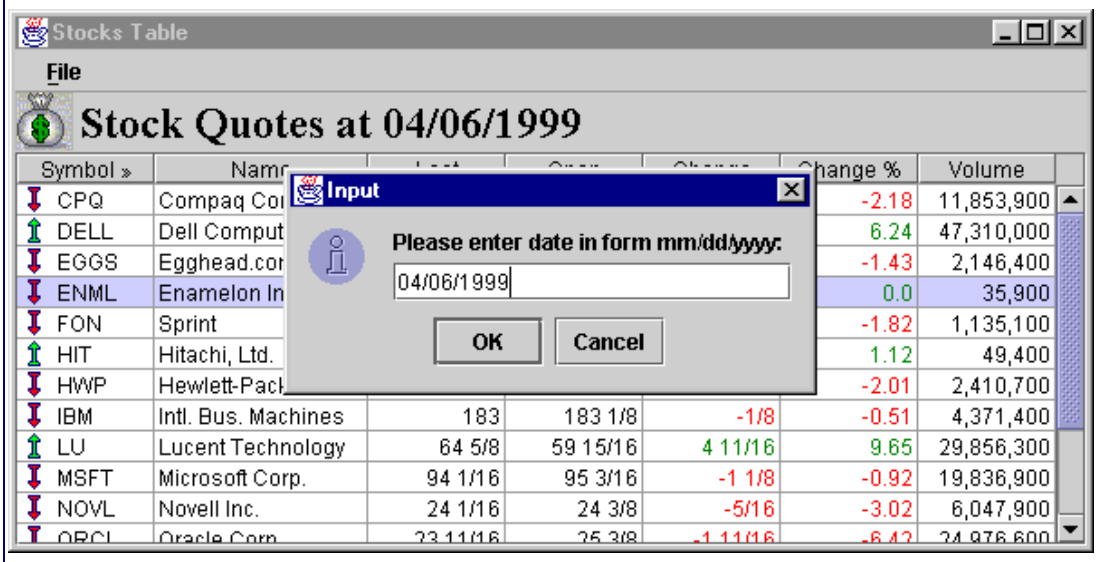


Figure 18.5 Retrieving stock data from a database for display in JTable.

<<file figure18-5.gif>>

The Code: StocksTable.java

see \Chapter18\5

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import java.text.*;
import java.sql.*;
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import javax.swing.table.*;
public class StocksTable extends JFrame
{
    protected JTable m_table;
    protected StockTableData m_data;
    protected JLabel m_title;
public StocksTable() {
// Unchanged code from section 18.4
JMenuBar menuBar = createMenuBar();
setJMenuBar(menuBar);
// Unchanged code
}
```

```
protected JMenuBar createMenuBar() {
JMenuBar menuBar = new JMenuBar();
JMenu mFile = new JMenu("File");
mFile.setMnemonic('f');
JMenuItem mData = new JMenuItem("Retrieve Data...");
mData.setMnemonic('r');
ActionListener lstData = new ActionListener() {
public void actionPerformed(ActionEvent e) {
retrieveData();
}
};
mData.addActionListener(lstData);
mFile.add(mData);
mFile.addSeparator();
JMenuItem mExit = new JMenuItem("Exit");
mExit.setMnemonic('x');
ActionListener lstExit = new ActionListener() {
public void actionPerformed(ActionEvent e) {
System.exit(0);
}
};
mExit.addActionListener(lstExit);
mFile.add(mExit);
menuBar.add(mFile);
return menuBar;
}
public void retrieveData() {
SimpleDateFormat frm = new SimpleDateFormat("MM/dd/yyyy");
String currentDate = frm.format(m_data.m_date);
String result = (String)JOptionPane.showInputDialog(this,
"Please enter date in form mm/dd/yyyy:", "Input",
JOptionPane.INFORMATION_MESSAGE, null, null,
currentDate);
if (result==null)
return;
java.util.Date date = null;
```

```
try {
date = frm.parse(result);
}
catch (java.text.ParseException ex) {
date = null;
}
if (date == null) {
JOptionPane.showMessageDialog(this,
result+" is not a valid date",
"Warning", JOptionPane.WARNING_MESSAGE);
return;
}
setCursor( Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR) );
switch (m_data.retrieveData(date)) {
case 0: // Ok with data
m_title.setText(m_data.getTitle());
m_table.tableChanged(new TableModelEvent(m_data));
m_table.repaint();
break;
case 1: // No data
JOptionPane.showMessageDialog(this,
"No data found for "+result,
"Warning", JOptionPane.WARNING_MESSAGE);
break;
case -1: // Error
JOptionPane.showMessageDialog(this,
"Error retrieving data",
"Warning", JOptionPane.WARNING_MESSAGE);
break;
}
setCursor(Cursor.getPredefinedCursor(Cursor.DEFAULT_CURSOR));
}
public static void main(String argv[]) {
new StocksTable();
}
}
```

```
// Unchanged code from section 18.4
class StockTableData extends AbstractTableModel
{
static final public ColumnData m_columns[] = {
// Unchanged code from section 18.2
};
protected SimpleDateFormat m_frm;
protected Vector m_vector;
protected java.util.Date m_date; // conflict with
protected int m_sortCol = 0;
protected boolean m_sortAsc = true;
protected int m_result = 0;
public StockTableData() {
m_frm = new SimpleDateFormat("MM/dd/yyyy");
m_vector = new Vector();
setDefaultData();
}
// Unchanged code from section 18.4
public int retrieveData(final java.util.Date date) {
GregorianCalendar calendar = new GregorianCalendar();
calendar.setTime(date);
int month = calendar.get(Calendar.MONTH)+1;
int day = calendar.get(Calendar.DAY_OF_MONTH);
int year = calendar.get(Calendar.YEAR);
final String query = "SELECT data.symbol, symbols.name, "+
"data.last, data.open, data.change, data.changeproc, "+
"data.volume FROM DATA INNER JOIN SYMBOLS "+
"ON DATA.symbol = SYMBOLS.symbol WHERE "+
"month(data.date1)="+month+" AND day(data.date1)="+day+
" AND year(data.date1)="+year;
Thread runner = new Thread() {
public void run() {
try {
// Load the JDBC-ODBC bridge driver
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection conn = DriverManager.getConnection(
```

```
"jdbc:odbc:Market", "admin", "");
Statement stmt = conn.createStatement();
ResultSet results = stmt.executeQuery(query);
boolean hasData = false;
while (results.next()) {
if (!hasData) {
m_vector.removeAllElements();
hasData = true;
}
String symbol = results.getString(1);
String name = results.getString(2);
double last = results.getDouble(3);
double open = results.getDouble(4);
double change = results.getDouble(5);
double changePr = results.getDouble(6);
long volume = results.getLong(7);
m_vector.addElement(new StockData(symbol, name, last,
open, change, changePr, volume));
}
results.close();
stmt.close();
conn.close();
if (!hasData) // We've got nothing
m_result = 1;
}
catch (Exception e) {
e.printStackTrace();
System.err.println("Load data error: "+e.toString());
m_result = -1;
}
m_date = date;
Collections.sort(m_vector,
new StockComparator(m_sortCol, m_sortAsc));
m_result = 0;
}
};
```

```
runner.start();

return m_result;

}

// Unchanged code from section 18.4

}

// Class StockComparator unchanged from section 18.4
```

Understanding the Code

Class StocksTable

A JMenuBar instance is created with our custom createMenuBar() method and added to our frame.

The createMenuBar() method creates a menu bar containing a single menu titled "File." Two menu items are added: "Retrieve Data..." and "Exit" with a separator in between. Anonymous ActionListeners are added to each. The first calls our custom retrieveData() method, and the second simply kills the application using System.exit(0).

The retrieveData() method is called in response to a "Retrieve Data..." menu item activation. First it prompts the user to enter the date by displaying a JOptionPane dialog. Once the date has been entered, this method parses it using a SimpleDateFormat object. If the entered string cannot be parsed into a valid date, the method shows a warning message and returns. Otherwise, we connect to JDBC and retrieve new data. To indicate that the program will be busy for some time the wait mouse cursor is displayed. The main job is performed by our new StockTableData retrieveData() method (see below), which is invoked on the m_data object. StockTableData's retrieveData() method returns an error code that the rest of this method depends on:

0: Normal finish, some data retrieved. The table model is updated and repainted.

1: Normal finish, no data retrieved. A warning message is displayed, and no changes in the table model are made.

-1: An error has occurred. An error message is displayed, and no changes in the table model are made.

Class StockTableData

First, a minor change is required in the declaration of the m_date variable. Since now we've imported the java.sql package, which also includes the Date class, we have to do provide the fully qualified calls name java.util.Date.

A new instance variable is added to store the result of a data retrieval request in the retrieveData() method. As mentioned above, retrieveData() retrieves a table's data for a given date of trade. Our implementation uses the JDBC bridge driver and should be familiar to JDBC-aware readers. The first thing we do is construct an SQL statement. Since we cannot compare a java.util.Date object and an SQL date stored in the database, we have to extract the date's components (year, month, and day) and compare them separately. An instance of GregorianCalendar is used to manipulate the date object.

We load the JDBC-ODBC bridge driver to Microsoft Access by using the Class.forName method, and then connect to a database with the DriverManager.getConnection() method. If no exception is thrown, we can create a Statement instance for the newly created Connection object and retrieve a ResultSet by executing the previously constructed query.

While new data is available (checked with the ResultSet.next() method), we retrieve new data using basic getXX() methods, create a new StockData instance to encapsulate the new data, and add it to m_vector.

Note how the hasData local variable is used to distinguish the case in which we do not have any data in our RecordSet. The first time we receive some valid data from our RecordSet in the while loop, we set this variable to true and clean up our m_vector collection. If no data is found, we have an unchanged initial vector and the hasData flag is set to false. Finally we close our ResultSet, Statement, and Connection instances. If any exception occurs, the method prints the exception trace and returns a –1 to indicate an error. Otherwise our newly retrieved data is sorted with the Collections.sort() method and a 0 is returned to indicate success.

Running the Code

Figure 18.6 shows StocksTable with data retrieved from a database. Try loading data for different dates in your database. A sample Microsoft Access database, market.mdb, containing some real market data, can be found in the **\swing\Chapter18** directory.

18.7 Stocks Table: part VI - column addition and removal

JTable allows us to dynamically add and remove TableColumns on the fly. Recall that the TableColumnModel interface provides the methods addColumn() and removeColumn() to programmatically add or remove a TableColumn respectively. In this section we add dynamic column addition and removal to our StocksTable application.

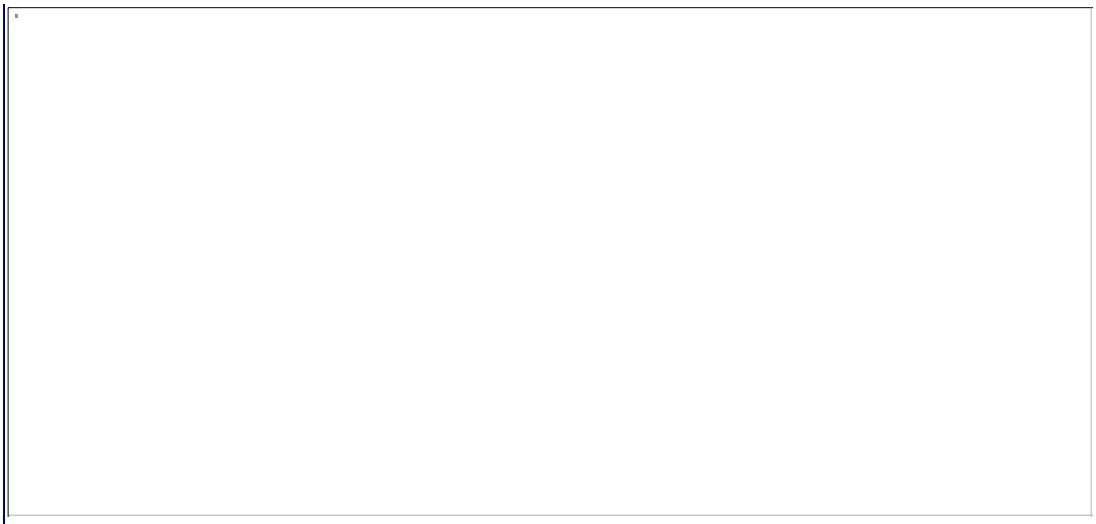


Figure 18.6 JTable with dynamic column addition and removal.

<<file figure18-6.gif>>

The Code: StocksTable.java

see \Chapter18\6

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import java.text.*;
import java.sql.*;
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import javax.swing.table.*;
public class StocksTable extends JFrame
{
// Unchanged code from section 18.5
public StocksTable() {
// Unchanged code from section 18.5
header.setReorderingAllowed(true);
m_table.getColumnModel().addColumnModelListener(
m_data.new ColumnMovementListener());
// Unchanged code from section 18.5
}
protected JMenuBar createMenuBar() {
// Unchanged code from section 18.5
```

```
JMenu mView = new JMenu("View");
mView.setMnemonic('v');
TableColumnModel model = m_table.getColumnModel();
for (int k = 0; k < StockTableData.m_columns.length; k++) {
JCheckBoxMenuItem item = new JCheckBoxMenuItem(
StockTableData.m_columns[k].m_title);
item.setSelected(true);
TableColumn column = model.getColumn(k);
item.addActionListener(new ColumnKeeper(column,
StockTableData.m_columns[k]));
mView.add(item);
}
menuBar.add(mView);
return menuBar;
}
// Unchanged code from section 18.5
class ColumnKeeper implements ActionListener
{
protected TableColumn m_column;
protected ColumnData m_colData;
public ColumnKeeper(TableColumn column, ColumnData colData) {
m_column = column;
m_colData = colData;
}
public void actionPerformed(ActionEvent e) {
JCheckBoxMenuItem item = (JCheckBoxMenuItem)e.getSource();
TableColumnModel model = m_table.getColumnModel();
if (item.isSelected()) {
model.addColumn(m_column);
}
else {
model.removeColumn(m_column);
}
m_table.tableChanged(new TableModelEvent(m_data));
m_table.repaint();
}
```

```
}
public static void main(String argv[]) {
new StocksTable();
}
}
// Unchanged code from section 18.5
class StockTableData extends AbstractTableModel
{
// Unchanged code from section 18.5
protected SimpleDateFormat m_frm;
protected Vector m_vector;
protected java.util.Date m_date;
protected int m_columnsCount = m_columns.length;
// Unchanged code from section 18.5
public int getColumnCount() {
return m_columnsCount;
}
// Unchanged code from section 18.5
class ColumnListener extends MouseAdapter
{
// Unchanged code from section 18.4
public void mouseClicked(MouseEvent e) {
// Unchanged code from section 18.4
for (int i=0; i < m_columnsCount; i++) {
TableColumn column = colModel.getColumn(i);
column.setHeaderValue(getColumnName(column.getModelIndex()));
}
m_table.getTableHeader().repaint();
// Unchanged code from section 18.4
}
}
class ColumnMovementListener implements TableColumnModelListener
{
public void columnAdded(TableColumnModelEvent e) {
m_columnsCount++;
}
```

```
public void columnRemoved(TableColumnModelEvent e) {
m_columnsCount--;
}
public void columnMarginChanged(ChangeEvent e) {}
public void columnMoved(TableColumnModelEvent e) {}
public void columnSelectionChanged(ListSelectionEvent e) {}
}
// Unchanged code from section 18.5
}
// Class StockComparator unchanged from section 18.4
```

Understanding the Code

Class StocksTable

The StocksTable constructor now adds an instance of StockTableData.ColumnMovementListener (see below) to our table's TableColumnModel to listen for column additions and removals.

Our createMenuBar() method now adds several checkbox menu items to a new "View" menu -- one for each column. Each of these check box menu items recieves a ColumnKeeper instance (see below) as ActionListener.

Class StocksTable.ColumnKeeper

This inner class implements the ActionListener interface and serves to keep track of when the user removes and adds columns to the table. The constructor receives a TableColumn instance and a ColumnData object. The actionPerformed() method adds this column to the model with the addColumn() method if the corresponding menu item is checked, and removes this column from the model with removeColumn() if it is unchecked. To update the table to properly reflect these changes, we call its tableChanged() method followed by a repaint() request.

Class StockTableData

StockTableData now contains instance variable m_columnsCount to keep track of the current column count. This variable is decremented and incremented in the columnRemoved() and columnAdded() methods of inner class ColumnMovementListener. It is also used in the StockTableData.ColumnListener class's mouseClicked() method for properly setting header values for the visible columns only.

Class StockTableData.ColumnMovementListener

This class implements TableColumnModelListener to increment and decrement StockTableData's m_columnsCount variable when a column addition or removal occurs, respectively. An instance of this inner class is added to our table's TableColumnModel in the StocksTable constructor.

Running the Code

Figure 18.7 shows the new "View" menu with an unchecked "Change %" menu item, and the corresponding column hidden. Reselecting this menu item will place the column back in the table at the end position. Verify that each menu item functions similarly.

18.8 Custom models, editors, and renderers

In constructing our StocksTable application we talked mostly about displaying and retrieving data in JTable. In this section we will construct a basic expense report application, and in doing so we will concentrate on table cell editing. We will also see how to implement dynamic addition and removal of table rows.

The editing of data generally follows this scheme:

> Create an instance of the TableCellEditor interface. We can use the DefaultCellEditor class or implement our own. The DefaultCellEditor class takes a GUI component as a parameter to its constructor: JTextField, JCheckBox or JComboBox. This component will be used for editing.
>
> If we are developing a custom editor, we need to implement the getTableCellEditorComponent() method which will be called each time a cell is about to be edited.

In our table model we need to implement the setValueAt(Object value, int nRow, int nCol) method which will be called to change a value in the table when an edit ends. This is where we can perform any necessary data processing and validation.

The data model for this example is designed as follows (where each row represents a column in our JTable):

Name	Type	Description
Date	String	Date of expense
Amount	Double	Amount of expense
Category	Integer	Category from pre-defined list
Approved	Boolean	Sign of approval for this expense.
Description	String	Brief description

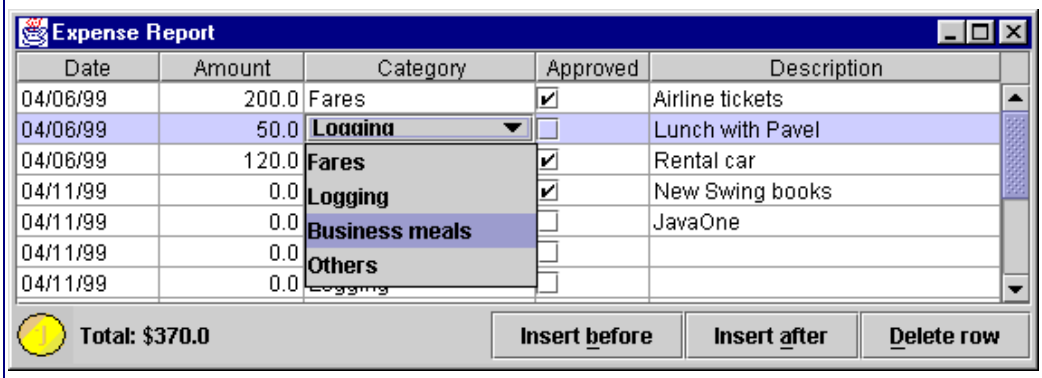


Figure 18.7 An expense report app illustrating custom cell editing, rendering, and row addition/removal.

<<file figure18-7.gif>>

Note: Since the only math that is done with our "Amount" values is addition, using Doubles is fine. However, in more professional implementations we may need to use rounding techniques or a custom renderer to remove unneccessary fractional amounts.

The Code: ExpenseReport.java

see \Chapter18\7

```
import java.awt.*;

import java.awt.event.*;

import java.util.*;

import java.io.*;

import java.text.SimpleDateFormat;

import javax.swing.*;

import javax.swing.border.*;

import javax.swing.event.*;

import javax.swing.table.*;

public class ExpenseReport extends JFrame

{

protected JTable m_table;

protected ExpenseReportData m_data;
```

```
protected JLabel m_title;
public ExpenseReport() {
super("Expense Report");
setSize(570, 200);
m_data = new ExpenseReportData(this);
m_table = new JTable();
m_table.setAutoCreateColumnsFromModel(false);
m_table.setModel(m_data);
m_table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
for (int k = 0; k < ExpenseReportData.m_columns.length; k++) {
TableCellRenderer renderer;
if (k==ExpenseReportData.COL_APPROVED)
renderer = new CheckCellRenderer();
else {
DefaultTableCellRenderer textRenderer =
new DefaultTableCellRenderer();
textRenderer.setHorizontalAlignment(
ExpenseReportData.m_columns[k].m_alignment);
renderer = textRenderer;
}
TableCellEditor editor;
if (k==ExpenseReportData.COL_CATEGORY)
editor = new DefaultCellEditor(new JComboBox(
ExpenseReportData.CATEGORIES));
else if (k==ExpenseReportData.COL_APPROVED)
editor = new DefaultCellEditor(new JCheckBox());
else
editor = new DefaultCellEditor(new JTextField());
TableColumn column = new TableColumn(k,
ExpenseReportData.m_columns[k].m_width,
renderer, editor);
m_table.addColumn(column);
}
JTableHeader header = m_table.getTableHeader();
header.setUpdateTableInRealTime(false);
JScrollPane ps = new JScrollPane();
```

```
ps.setSize(550, 150);
ps.getViewport().add(m_table);
getContentPane().add(ps, BorderLayout.CENTER);
JPanel p = new JPanel();
p.setLayout(new BoxLayout(p, BoxLayout.X_AXIS));
ImageIcon penny = new ImageIcon("penny.gif");
m_title = new JLabel("Total: $",
penny, JButton.LEFT);
m_title.setForeground(Color.black);
m_title.setAlignmentY(0.5f);
p.add(m_title);
p.add(Box.createHorizontalGlue());
JButton bt = new JButton("Insert before");
bt.setMnemonic('b');
bt.setAlignmentY(0.5f);
ActionListener lst = new ActionListener() {
public void actionPerformed(ActionEvent e) {
int row = m_table.getSelectedRow();
m_data.insert(row);
m_table.tableChanged(new TableModelEvent(
m_data, row, row, TableModelEvent.ALL_COLUMNS,
TableModelEvent.INSERT));
m_table.repaint();
}
};
bt.addActionListener(lst);
p.add(bt);
bt = new JButton("Insert after");
bt.setMnemonic('a');
bt.setAlignmentY(0.5f);
lst = new ActionListener() {
public void actionPerformed(ActionEvent e) {
int row = m_table.getSelectedRow();
m_data.insert(row+1);
m_table.tableChanged(new TableModelEvent(
m_data, row+1, row+1, TableModelEvent.ALL_COLUMNS,
```

```
TableModelEvent.INSERT));
m_table.repaint();
}
};
bt.addActionListener(lst);
p.add(bt);
bt = new JButton("Delete row");
bt.setMnemonic('d');
bt.setAlignmentY(0.5f);
lst = new ActionListener() {
public void actionPerformed(ActionEvent e) {
int row = m_table.getSelectedRow();
if (m_data.delete(row)) {
m_table.tableChanged(new TableModelEvent(
m_data, row, row, TableModelEvent.ALL_COLUMNS,
TableModelEvent.INSERT));
m_table.repaint();
calcTotal();
}
}
};
bt.addActionListener(lst);
p.add(bt);
getContentPane().add(p, BorderLayout.SOUTH);
calcTotal();
WindowListener wndCloser = new WindowAdapter() {
public void windowClosing(WindowEvent e) {
System.exit(0);
}
};
addWindowListener(wndCloser);
setVisible(true);
}
public void calcTotal() {
double total = 0;
for (int k=0; k<m_data.getRowCount(); k++) {
```

```
Double amount = (Double)m_data.getValueAt(k,
ExpenseReportData.COL_AMOUNT);
total += amount.doubleValue();
}
m_title.setText("Total: $"+total);
}
public static void main(String argv[]) {
new ExpenseReport();
}
}
class CheckCellRenderer extends JCheckBox implements TableCellRenderer
{
protected static Border m_noFocusBorder;
public CheckCellRenderer() {
super();
m_noFocusBorder = new EmptyBorder(1, 2, 1, 2);
setOpaque(true);
setBorder(m_noFocusBorder);
}
public Component getTableCellRendererComponent(JTable table,
Object value, boolean isSelected, boolean hasFocus,
int row, int column)
{
if (value instanceof Boolean) {
Boolean b = (Boolean)value;
setSelected(b.booleanValue());
}
setBackground(isSelected && !hasFocus ?
table.getSelectionBackground() : table.getBackground());
setForeground(isSelected && !hasFocus ?
table.getSelectionForeground() : table.getForeground());
setFont(table.getFont());
setBorder(hasFocus ? UIManager.getBorder(
"Table.focusCellHighlightBorder") : m_noFocusBorder);
return this;
}
```

```
}
class ExpenseData
{
public Date m_date;
public Double m_amount;
public Integer m_category;
public Boolean m_approved;
public String m_description;
public ExpenseData() {
m_date = new Date();
m_amount = new Double(0);
m_category = new Integer(1);
m_approved = new Boolean(false);
m_description = "";
}
public ExpenseData(Date date, double amount, int category,
boolean approved, String description)
{
m_date = date;
m_amount = new Double(amount);
m_category = new Integer(category);
m_approved = new Boolean(approved);
m_description = description;
}
}
class ColumnData
{
public String m_title;
int m_width;
int m_alignment;
public ColumnData(String title, int width, int alignment) {
m_title = title;
m_width = width;
m_alignment = alignment;
}
}
```

```
class ExpenseReportData extends AbstractTableModel
{
public static final ColumnData m_columns[] = {
new ColumnData( "Date", 80, JLabel.LEFT ),
new ColumnData( "Amount", 80, JLabel.RIGHT ),
new ColumnData( "Category", 130, JLabel.LEFT ),
new ColumnData( "Approved", 80, JLabel.LEFT ),
new ColumnData( "Description", 180, JLabel.LEFT )
};
public static final int COL_DATE = 0;
public static final int COL_AMOUNT = 1;
public static final int COL_CATEGORY = 2;
public static final int COL_APPROVED = 3;
public static final int COL_DESCR = 4;
public static final String[] CATEGORIES = {
"Fares", "Logging", "Business meals", "Others"
};
protected ExpenseReport m_parent;
protected SimpleDateFormat m_frm;
protected Vector m_vector;
public ExpenseReportData(ExpenseReport parent) {
m_parent = parent;
m_frm = new SimpleDateFormat("MM/dd/yy");
m_vector = new Vector();
setDefaultData();
}
public void setDefaultData() {
m_vector.removeAllElements();
try {
m_vector.addElement(new ExpenseData(
m_frm.parse("04/06/99"), 200, 0, true,
"Airline tickets"));
m_vector.addElement(new ExpenseData(
m_frm.parse("04/06/99"), 50, 2, false,
"Lunch with client"));
m_vector.addElement(new ExpenseData(
```

```
m_frm.parse("04/06/99"), 120, 1, true,
"Hotel"));
}
catch (java.text.ParseException ex) {}
}
public int getRowCount() {
return m_vector==null ? 0 : m_vector.size();
}
public int getColumnCount() {
return m_columns.length;
}
public String getColumnName(int column) {
return m_columns[column].m_title;
}
public boolean isCellEditable(int nRow, int nCol) {
return true;
}
public Object getValueAt(int nRow, int nCol) {
if (nRow < 0 || nRow>=getRowCount())
return "";
ExpenseData row = (ExpenseData)m_vector.elementAt(nRow);
switch (nCol) {
case COL_DATE: return m_frm.format(row.m_date);
case COL_AMOUNT: return row.m_amount;
case COL_CATEGORY: return CATEGORIES[row.m_category.intValue()];
case COL_APPROVED: return row.m_approved;
case COL_DESCR: return row.m_description;
}
return "";
}
public void setValueAt(Object value, int nRow, int nCol) {
if (nRow < 0 || nRow>=getRowCount())
return;
ExpenseData row = (ExpenseData)m_vector.elementAt(nRow);
String svalue = value.toString();
switch (nCol) {
```

```
case COL_DATE:
Date date = null;
try {
date = m_frm.parse(svalue);
}
catch (java.text.ParseException ex) {
date = null;
}
if (date == null) {
JOptionPane.showMessageDialog(null,
svalue+" is not a valid date",
"Warning", JOptionPane.WARNING_MESSAGE);
return;
}
row.m_date = date;
break;
case COL_AMOUNT:
try {
row.m_amount = new Double(svalue);
}
catch (NumberFormatException e) { break; }
m_parent.calcTotal();
break;
case COL_CATEGORY:
for (int k=0; k<CATEGORIES.length; k++)
if (svalue.equals(CATEGORIES[k])) {
row.m_category = new Integer(k);
break;
}
break;
case COL_APPROVED:
row.m_approved = (Boolean)value;
break;
case COL_DESCR:
row.m_description = svalue;
break;
```

```
}
}
public void insert(int row) {
if (row < 0)
row = 0;
if (row > m_vector.size())
row = m_vector.size();
m_vector.insertElementAt(new ExpenseData(), row);
}
public boolean delete(int row) {
if (row < 0 || row >= m_vector.size())
return false;
m_vector.remove(row);
return true;
}
}
```

Understanding the Code

Class ExpenseReport

Class ExpenseReport extends JFrame and defines three instance variables:

JTable m_table: table to edit data.

ExpenseReportData m_data: data model for this table.

JLabel m_total: label to dynamically display total amount of expenses.

The ExpenseReport constructor first instantiates our table model, m_data, and then instantiates our table, m_table. The selection mode is set to single selection and we iterate through the number of columns creating cell renderers and editors based on each specific column. The "Approved" column uses an instance of our custom CheckCellRenderer class as renderer. All other columns use a DefaultTableCellRenderer. All columns also use a DefaultCellEditor. However, the component used for editing varies: the "Category" column uses a JComboBox, the "Approved" column uses a JCheckBox, and all other columns use a JTextField. These components are passed to the DefaultTableCellRenderer constructor.

Several components are added to the bottom of our frame: JLabel m_total, used to display the total amount of expenses, and three JButtons used to manipulate tables rows. (Note that the horizontal glue component added between the label and the button pushes buttons to the right side of the panel, so they remain glued to the right when our frame is resized.)

These three buttons, titled "Insert before," "Insert after," and "Delete row," behave as their titles imply. The first two use the insert() method from the ExpenseReportData model to insert a new row before or after the currently selected row. The last one deletes the currently selected row by calling the delete() method. In all cases the modified table is updated and repainted.

Method calcTotal() calculates the total amount of expenses in column COL_AMOUNT using our table's data model, m_data.

Class CheckCellRenderer

Since we use check boxes to edit our table's "Approved" column, to be consistent we also need to use check boxes for that column's cell renderer (recall that cell renderers just act as "rubber stamps" and are not at all interactive). The only GUI component which can be used in the existing DefaultTableCellRenderer is JLabel, so we have to provide our own implementation of the TableCellRenderer interface. This class, CheckCellRenderer, uses JCheckBox as a super-class. Its constructor sets the border to indicate whether the component has the focus and sets its opaque property to true to indicate that the component's background will be filled with the background color.

The only method which must be implemented in the TableCellRenderer interface is getTableCellRendererComponent(). This method

will be called each time the cell is about to be rendered to deliver new data to the renderer. It takes six parameters:

JTable table: reference to table instance.

Object value: data object to be sent to the renderer.

boolean isSelected: true if the cell is currently selected.

boolean hasFocus: true if the cell currently has the focus.

int row: cell's row.

int column: cell's column.

Our implementation sets whether the JCheckBox is checked depending on the value passed as Boolean. Then it sets the background, foreground, font, and border to ensure that each cell in the table has a similar appearance.

Class ExpenseData

Class ExpenseData represents a single row in the table. It holds five variables corresponding to our data structure described in the beginning of this section.

Class ColumnData

Class ColumnData holds each column's title, width, and header alignment.

Class ExpenseReportData

ExpenseReportData extends AbstractTableModel and should look somewhat familiar from previous examples in this chapter (e.g. StockTableData), so we will not discuss this class in complete detail. However, we need to take a closer look at the setValueAt() method, which is new for this example (all previous examples did not accept new data). This method is called each time an edit is made to a table cell. First we determine which ExpenseData instance (table's row) is affected, and if it is invalid we simply return. Otherwise, depending on the column of the changed cell, we define several cases in a switch structure to accept and store a new value, or to reject it:

For the "Date" column the input string is parsed using our SimpleDateFormat instance. If parsing is successful, a new date is saved as a Date object, otherwise an error message is displayed.

For the "Amount" column the input string is parsed as a Double and stored in the table if parsing is successful. Also a new total amount is recalculated and displayed in the "Total" JLabel.

For the "Category" column the input string is placed in the CATEGORIES array at the corresponding index and is stored in the table model.

For the "Approved" column the input object is cast to a Boolean and stored in the table model.

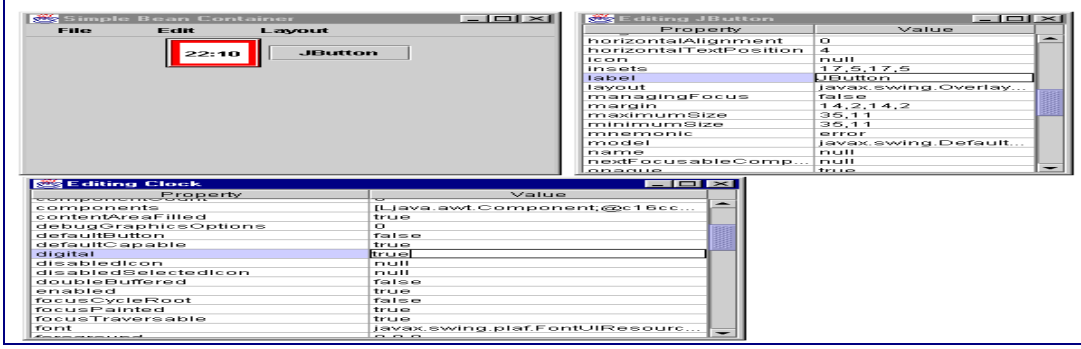
For the "Description" column the input string is directly saved in our table model.

Running the Code

Try editing different columns and note how the corresponding cell editors work. Experiment with adding and removing table rows and note how the total amount is updated each time the "Amount" column is updated. Figure 18.8 shows ExpenseReport with a combo box opened to change a cell's value.

18.9 A JavaBeans property editor

Now that we're familiar with the table API we can complete the JavaBeans container introduced in the chapter 4 and give it the capability to edit the properties of JavaBeans. This dramatically increases the possible uses of our simple container and makes it a quite powerful tool for studying JavaBeans.



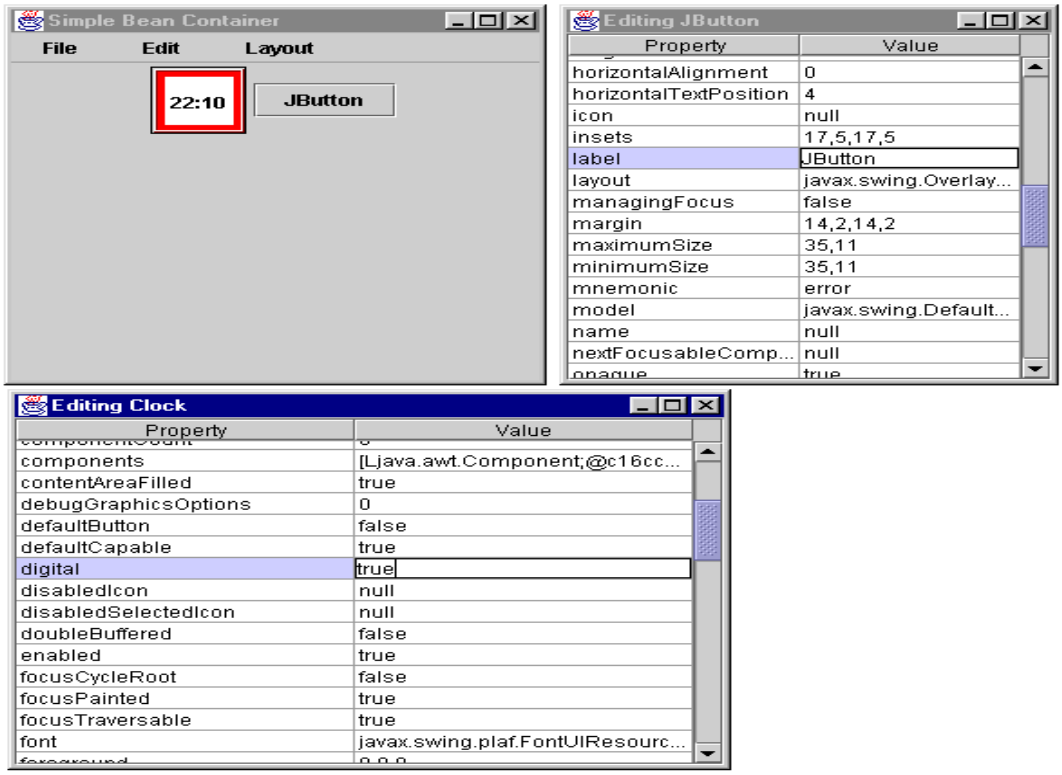


Figure 18.8 BeanContainer JavaBeans property editor using JTables as editing forms.

<<file figure18-8.gif>>

The Code: BeanContainer.java

see \Chapter18\8

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.beans.*;
import java.lang.reflect.*;
import java.util.*;
import javax.swing.*;
import javax.swing.table.*;
import javax.swing.event.*;
import dl.*;
public class BeanContainer extends JFrame implements FocusListener
{
protected Hashtable m_editors = new Hashtable();
// Unchanged code from section 4.7
protected JMenuBar createMenuBar() {
// Unchanged code from section 4.7
JMenu mEdit = new JMenu("Edit");
```

```
mItem = new JMenuItem("Delete");
lst = new ActionListener() {
public void actionPerformed(ActionEvent e) {
if (m_activeBean == null)
return;
Object obj = m_editors.get(m_activeBean);
if (obj != null) {
BeanEditor editor = (BeanEditor)obj;
editor.dispose();
m_editors.remove(m_activeBean);
}
getContentPane().remove(m_activeBean);
m_activeBean = null;
validate();
repaint();
}
};
mItem.addActionListener(lst);
mEdit.add(mItem);
mItem = new JMenuItem("Properties...");
lst = new ActionListener() {
public void actionPerformed(ActionEvent e) {
if (m_activeBean == null)
return;
Object obj = m_editors.get(m_activeBean);
if (obj != null) {
BeanEditor editor = (BeanEditor)obj;
editor.setVisible(true);
editor.toFront();
}
else {
BeanEditor editor = new BeanEditor(m_activeBean);
m_editors.put(m_activeBean, editor);
}
}
};
```

```
mItem.addActionListener(lst);
mEdit.add(mItem);
menuBar.add(mEdit);
// Unchanged code from section 4.7
return menuBar;
}
// Unchanged code from section 4.7
}
class BeanEditor extends JFrame implements PropertyChangeListener
{
protected Component m_bean;
protected JTable m_table;
protected PropertyTableData m_data;
public BeanEditor(Component bean) {
m_bean = bean;
m_bean.addPropertyChangeListener(this);
Point pt = m_bean.getLocationOnScreen();
setBounds(pt.x+50, pt.y+10, 400, 300);
getContentPane().setLayout(new BorderLayout());
m_data = new PropertyTableData(m_bean);
m_table = new JTable(m_data);
JScrollPane ps = new JScrollPane();
ps.getViewport().add(m_table);
getContentPane().add(ps, BorderLayout.CENTER);
setDefaultCloseOperation(HIDE_ON_CLOSE);
setVisible(true);
}
public void propertyChange(PropertyChangeEvent evt) {
m_data.setProperty(evt.getPropertyName(), evt.getNewValue());
}
class PropertyTableData extends AbstractTableModel
{
protected String[][] m_properties;
protected int m_numProps = 0;
protected Vector m_v;
public PropertyTableData(Component bean) {
```

```
try {
BeanInfo info = Introspector.getBeanInfo(
m_bean.getClass());
BeanDescriptor descr = info.getBeanDescriptor();
setTitle("Editing "+descr.getName());
PropertyDescriptor[] props = info.getPropertyDescriptors();
m_numProps = props.length;
m_v = new Vector(m_numProps);
for (int k=0; k<m_numProps; k++) {
String name = props[k].getDisplayName();
boolean added = false;
for (int i=0; i<m_v.size(); i++) {
String str = ((PropertyDescriptor)m_v.elementAt(i)).
getDisplayName();
if (name.compareToIgnoreCase(str) < 0) {
m_v.insertElementAt(props[k], i);
added = true;
break;
}
}
if (!added)
m_v.addElement(props[k]);
}
m_properties = new String[m_numProps][2];
for (int k=0; k<m_numProps; k++) {
PropertyDescriptor prop =
(PropertyDescriptor)m_v.elementAt(k);
m_properties[k][0] = prop.getDisplayName();
Method mRead = prop.getReadMethod();
if (mRead != null &&
mRead.getParameterTypes().length == 0) {
Object value = mRead.invoke(m_bean, null);
m_properties[k][1] = objToString(value);
}
else
m_properties[k][1] = "error";
```

```
}
}
catch (Exception ex) {
ex.printStackTrace();
JOptionPane.showMessageDialog(
BeanEditor.this, "Error: "+ex.toString(),
"Warning", JOptionPane.WARNING_MESSAGE);
}
}
public void setProperty(String name, Object value) {
for (int k=0; k<m_numProps; k++)
if (name.equals(m_properties[k][0])) {
m_properties[k][1] = objToString(value);
m_table.tableChanged(new TableModelEvent(this, k));
m_table.repaint();
break;
}
}
public int getRowCount() { return m_numProps; }
public int getColumnCount() { return 2; }
public String getColumnName(int nCol) {
return nCol==0 ? "Property" : "Value";
}
public boolean isCellEditable(int nRow, int nCol) {
return (nCol==1);
}
public Object getValueAt(int nRow, int nCol) {
if (nRow < 0 || nRow>=getRowCount())
return "";
switch (nCol) {
case 0: return m_properties[nRow][0];
case 1: return m_properties[nRow][1];
}
return "";
}
public void setValueAt(Object value, int nRow, int nCol) {
```

```
if (nRow < 0 || nRow>=getRowCount())
return;
String str = value.toString();
PropertyDescriptor prop = (PropertyDescriptor)m_v.
elementAt(nRow);
Class cls = prop.getPropertyType();
Object obj = stringToObj(str, cls);
if (obj==null)
return; // can't process
Method mWrite = prop.getWriteMethod();
if (mWrite == null || mWrite.getParameterTypes().length != 1)
return;
try {
mWrite.invoke(m_bean, new Object[]{ obj });
m_bean.getParent().doLayout();
m_bean.getParent().repaint();
m_bean.repaint();
}
catch (Exception ex) {
ex.printStackTrace();
JOptionPane.showMessageDialog(
BeanEditor.this, "Error: "+ex.toString(),
"Warning", JOptionPane.WARNING_MESSAGE);
}
m_properties[nRow][1] = str;
}
public String objToString(Object value) {
if (value==null)
return "null";
if (value instanceof Dimension) {
Dimension dim = (Dimension)value;
return ""+dim.width+","+dim.height;
}
else if (value instanceof Insets) {
Insets ins = (Insets)value;
return ""+ins.left+","+ins.top+","+ins.right+","+ins.bottom;
```

```
}
else if (value instanceof Rectangle) {
Rectangle rc = (Rectangle)value;
return ""+rc.x+","+rc.y+","+rc.width+","+rc.height;
}
else if (value instanceof Color) {
Color col = (Color)value;
return ""+col.getRed()+","+col.getGreen()+","+col.getBlue();
}
return value.toString();
}
public Object stringToObj(String str, Class cls) {
try {
if (str==null)
return null;
String name = cls.getName();
if (name.equals("java.lang.String"))
return str;
else if (name.equals("int"))
return new Integer(str);
else if (name.equals("long"))
return new Long(str);
else if (name.equals("float"))
return new Float(str);
else if (name.equals("double"))
return new Double(str);
else if (name.equals("boolean"))
return new Boolean(str);
else if (name.equals("java.awt.Dimension")) {
int[] i = strToInts(str);
return new Dimension(i[0], i[1]);
}
else if (name.equals("java.awt.Insets")) {
int[] i = strToInts(str);
return new Insets(i[0], i[1], i[2], i[3]);
}
```

```
else if (name.equals("java.awt.Rectangle")) {
int[] i = strToInts(str);
return new Rectangle(i[0], i[1], i[2], i[3]);
}
else if (name.equals("java.awt.Color")) {
int[] i = strToInts(str);
return new Color(i[0], i[1], i[2]);
}
return null; // not supported
}
catch(Exception ex) { return null; }
}
public int[] strToInts(String str) throws Exception {
int[] i = new int[4];
StringTokenizer tokenizer = new StringTokenizer(str, ",");
for (int k=0; k<i.length &&
tokenizer.hasMoreTokens(); k++)
i[k] = Integer.parseInt(tokenizer.nextToken());
return i;
}
}
}
```

Understanding the Code

Class BeanContainer

This class (formerly BeanContainer from section 4.7) has received a new collection, Hashtable m_editors, added as an instance variable. This Hashtable holds references to BeanEditor frames (used to edit beans, see below) as values and the corresponding Components being edited as keys.

A new menu item titled "Properties..." is added to the "Edit" menu. This item is used to create a new editor for the selected bean or activate an existing one (if any). The attached ActionListener looks for an existing BeanEditor corresponding to the currently selected m_activeBean component in the m_editors collection. If such an editor is found it is made visible and brought to the front. Otherwise, a new instance of BeanEditor is created to edit the currently active m_activeBean component, and is added to the m_editors collection.

The ActionListener attached to menu item "Delete," which removes the currently active component, receives additional functionality. The added code looks for an existing BeanEditor corresponding to the currently selected m_activeBean component in the m_editors collection. If such an editor is found it is disposed and its' reference is removed from the hashtable.

Class BeanEditor

This class extends JFrame and implements the PropertyChangeListener interface. BeanEditor is used to display and edit the properties exposed by a given JavaBean. Three instance variables are declared:

Component m_bean: JavaBean component to be edited.

JTable m_table: table component to display a bean's properties.

PropertyTableData m_data: table model for m_table.

The BeanEditor constructor takes a reference to the JavaBean component to be edited, and stores it in instance variable m_bean. The initial location of the editor frame is selected depending on the location of the component being edited.

The table component, m_table, is created and added to a JScrollPane to provide scrolling capabilities. Note that we do not add a WindowListener to this frame. Instead we use the HIDE_ON_CLOSE default close operation (see chapter 3):

setDefaultCloseOperation(HIDE_ON_CLOSE);

setVisible(true);

Upon closing, this frame will be hidden but not disposed. Its' reference will still be present in the m_editors collection, and this frame will be re-activated if the user chooses to see the properties of the associated bean again.

Note that an instance of the BeanEditor class is added as a PropertyChangeListener to the corresponding bean being edited. The propertyChange() method is invoked if the bean has changed it's state during editing and a PropertyChangeEvent has been fired. This method simply triggers a call to the setProperty() method of the table model.

Class BeanEditor.PropertyTableData

PropertyTableData extends AbstractTableModel and provides the table model for each bean editor. Three instance variables are declared:

String[][] m_properties: an array of data displayed in the table.

int m_numProps: number of a bean properties (corresponds to the number of rows in the table).

Vector m_v: collection of PropertyDescriptor objects sorted in alphabetical order.

The constructor of the PropertyTableData class takes a given bean instance and retrieves it's properties. First it uses the Introspector.getBeanInfo() method to get a BeanInfo instance:

BeanInfo info = Introspector.getBeanInfo(

m_bean.getClass());

BeanDescriptor descr = info.getBeanDescriptor();

setTitle("Editing "+descr.getName());

PropertyDescriptor[] props = info.getPropertyDescriptors();

m_numProps = props.length;

This provides us with all available information about a bean (see chapter 2). We determine the bean's name and use it as the editor frame's title (note that this is an inner class, so setTitle() refers to the parent BeanEditor instance). We then extract an array of PropertyDescriptors which will provide us with the actual information about a bean's properties.

Bean properties are sorted by name in alphabetical order. The name of each property is determined with the getDisplayName() method. The sorted PropertyDescriptors are stored in our m_v Vector collection. Then we can create the 2-dimensional array, m_properties, which holds data to be displayed in the table. This array has m_numProps rows and 2 columns (for property name and value). To determine a property's value we need to obtain a reference to its getXX() method with getReadMethod() and make a call using the reflection API. We can call only getXX() methods without parameters (since we don't know anything about these parameters). Note that our objToString() helper method is invoked to translate a property's value into a display string (see below).

The setProperty() method searches for the given name in the 0-th column of the m_properties array. If such a property is found this method sets it's new value and updates the table component.

Several other simple methods included in this class have already been presented in previous examples and need not be explained again here. However, note that the isCellEditable() method returns true only for cells in the second column (property names, obviously, cannot be changed).

The setValueAt() method deserves additional explanation because it not only saves the modified data in the table model, but it also sends these modifications to the bean component itself. To do this we obtain a PropertyDescriptor instance stored in the m_v Vector collection. The modified property value is always a String, so first we need to convert it into its proper object type using our stringToObj() helper method (if we can do this, see below). If the conversion succeeds (i.e. the result is not null), we can continue.

To modify a bean value we determine the reference to it's setXX()method (corresponding to a certain property) and invoke it. Note that an anonymous array containing one element is used as parameter (these constructions are typical when dealing with the reflection API). Then the bean component and it's container (which can also be affected by changes in such properties as size and color) are refreshed to reflect the bean's new property value. Finally, if the above procedures were successful, we store the new value in the m_properties data array.

The objToString() helper method converts a given Object into a String suitable for editing. In many cases the toString() method returns a long string starting with the class name. This is not very appropriate for editable data values. So for several classes we provide our own conversion into a string of comma-delimited numbers. For instance a Dimension object is converted into a "width, height" form, Color is converted into "red, green, blue" form, etc. If no special implementation is provided, an object's toString() string is returned.

The stringToObj() helper method converts a given String into an Object of the given Class. The class's name is analyzed and a conversion method is chosen to build the correct type of object based on this name. The simplest case is the String class: we don't need to do any conversion at all in this case. For the primitive data types such as int or boolean we return the corresponding encapsulating (wrapper class) objects. For the several classes which receive special treatment in the objToString() method (such as a Dimension or Color object), we parse the comma-delimited string of numbers and construct the proper object. For all other classes (or if a parsing exception occurs) we return null to indicate that we cannot perform the required conversion.

Running the Code

Figure 18.9 shows the BeanContainer container and two editing frames displaying the properties of Clock and JButton components. This application provides a simple but powerful tool for investigating Swing and AWT components as well as custom JavaBeans. We can see all exposed properties and modify many of them. If a component’s properties change as a result of user interaction, our component properly notifies its’ listeners and we see an automatic editor table update. Try serializing a modified component and restoring it from its’ file. Note how the previously modified properties are saved as expected.

It is natural to imagine using this example as a base for constructing a custom Swing IDE (Interface Development Environment). BeanContainer, combined with the custom resize edge components developed in chapters 15 and 16, provides a fairly powerful base to work from.