

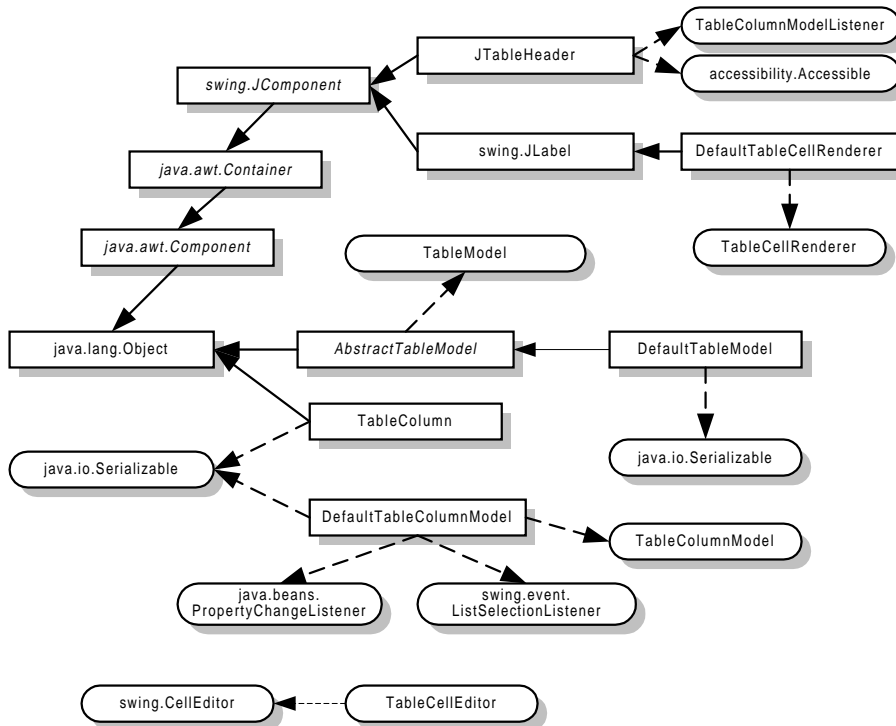
Package swing.table

Swing's `JTable` is extremely powerful but it remains very easy to use. This package contains the classes that give `JTable` both its power and its ease of use.

It is important to remember that `JTable` is not designed to be a spreadsheet. It is designed to present tables of data where the columns contain data *all of the same type* (or related by a common superclass). So, a column may contain only all `Strings`, or all `Booleans`, or all instances of a custom class. An example of this would be a table which lists information about students: column 1 is always the last name `String`, column 2 is the first name `String`, column 3 is a `Boolean` indicating whether they are currently attending, and so forth. The closest you can come to mixing data types in a single column of a `JTable` is having the types all be related by a common superclass. The default behavior for a `JTable` is to use `java.lang.Object` as the common superclass, and render the data by using the `toString()` method in `Object`.

`JTables` can have custom `TableCellRenderers` (p.760) and `TableCellEditors` (p.758) assigned to the whole table, or on a column-by-column basis. A `TableCellRenderer` is an object that understands how to display a particular kind of data. Similarly, a `TableCellEditor` understands how to edit a particular kind of data. If a cell renderer or editor is assigned to the whole table, it is assigned using a method that also takes a parameter of type `Class` that indicates what kind of data the renderer or editor understands. The `JTable` later uses the `getColumnClass()` method of `TableModel` to choose the appropriate renderer or editor for the given column.

Table extends/implements hierarchy



Quick summary

The model	<code>TableModel</code> (p.766) models the data. <code>TableColumnModel</code> (built from a series of <code>TableColumns</code>) models the columns (p.764).
The view	<code>JTable</code> (p.357). <code>JTableHeader</code> (p.752) manages and displays the column headers.
How to use	Provide a <code>TableModel</code> implementation, either by subclassing <code>AbstractTableModel</code> (p.735) or by using <code>DefaultTableModel</code> (p.747). You can subclass or use <code>DefaultTableColumnModel</code> (p.744) to provide a model for the columns (or let <code>JTable</code> build its own column model from the data).
Display and edit	The <code>TableCellRenderer</code> (p.760) interface defines how the table's cells are displayed. The <code>TableCellEditor</code> (p.758) interface defines how the cells are edited. Implement these interfaces (or use a <code>DefaultTableCellRenderer</code>) and install them on a <code>JTable</code> or a <code>TableColumn</code> to change rendering or editing.
Events	A <code>TableModelEvent</code> (p.508) describes changes in the table data. Its listener is <code>TableModelListener</code> (p.513). A <code>TableColumnModelEvent</code> (p.507) describes a change in the order or size of the columns. Its listener is <code>TableColumnModelListener</code> (p.508).

FAQs

How do I put my information in a table?

Either subclass `AbstractTableModel` and implement the required three methods, or use a `DefaultTableModel` (which has a built-in `Vector`). The three methods required by `AbstractTableModel` are very simple to implement so subclassing it is the preferred method. Using `DefaultTableModel` requires copying your data from its native form into the `DefaultTableModel`, so it is tremendously less efficient than `AbstractTableModel`.

How do I get different colored cells?

The look and feel is usually responsible for providing the appearance of the cells. You can install a `TableCellRenderer` that uses the row and column position to decide what color the cell should be.

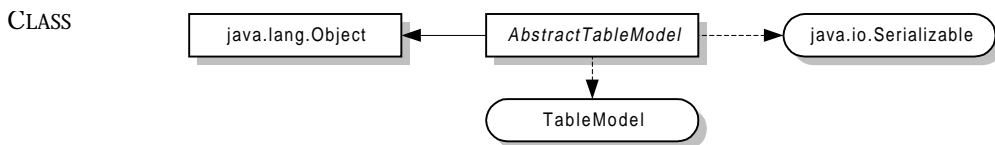
How do I load cells on demand?

Subclass `AbstractTableModel`, and have your implementation of `getValueAt()` check whether the cell (or row) is loaded: if so, return it; if not, load it first and then return the value. A hash table using the row and column as a key can be used to store the loaded cells, or a sparse-matrix implementation may be used. Given the tabular (as opposed to spreadsheet) nature of a `JTable`, a reasonable solution would involve loading a full row of data at a time since the data on a row is usually all related.

How do I get a sorted table?

You can provide a sorting table model as a wrapper around a base table model. When the base table changes, the sorted table will re-sort. Operations involving the row will be translated according to the sorted order. See `AbstractTableModel` for an example.

swing.table.AbstractTableModel



The `AbstractTableModel` provides a default implementation of the `TableModel` interface that should be used as the basis for almost all table models you implement. It supplies methods for managing the lists of listeners and simple (or empty) implementations of the other methods in the `TableModel` interface.

A minimal read-only table must implement `getColumnCount()`, `getRowCount()`, and `getValueAt()`. If you want to modify values, you will override `isCellEditable()` to support

in-place editing in the table and `setValueAt()`. For most realistic applications you will want to override `getColumnClass()` and `getColumnName()`. `getColumnClass()` is used to identify the correct renderer and/or editor for the column. The implementation of `getColumnClass()` supplied by this class always returns `Object.class`. This causes the `JTable` to use the default `Object.class` renderer, which calls `toString()` on the value in the cell and displays the returned text. `getColumnName()` supplies the labels to be displayed in the `JTable`'s table header.

Example: Minimal work

```
// File atm\Example1.java
package atm;

import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;
import com.sun.java.swing.table.*;

public class Example1 extends JFrame {
    // Table of primes < 20 as the x coordinate, 2*x+1 as the y coordinate.
    public class CoordinateTableModel extends AbstractTableModel {
        private int[] x = {2, 3, 5, 7, 11, 13, 17, 19}; // x is prime < 20
        public int getColumnCount() {return 2;} // Two columns, x and y
        public int getRowCount() {return x.length;}
        public Object getValueAt(int r, int c) {
            return (c == 0) ? new Integer(x[r]) : new Integer(2 * x[r] + 1);
        }
        public String getColumnName(int c) {return (c == 0) ? "x" : "2x+1";}
    }

    public Example1() {
        JTable table = new JTable(new CoordinateTableModel());
        getContentPane().add(new JScrollPane(table), BorderLayout.CENTER);
        pack();
    }

    public static void main (String arg[]) {
        Example1 ex1 = new Example1();
        ex1.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        ex1.addWindowListener(new WindowAdapter() {
            public void windowClosed(WindowEvent e) {
                System.exit(0);
            }
        });
        ex1.setVisible(true);
    }
}
```

Example: Firing notifications

Suppose you want to change the contents of the table. If you just change values without telling anyone, views like `JTables` won't be aware that a change has occurred, so they won't update. To prevent this problem, call the appropriate `fireTableXxx()` methods after your change. This simple example shows the update for `setValueAt()`.

```

// File atm/Example2.java
package atm;

import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;
import com.sun.java.swing.table.*;

public class Example2 extends JFrame {
    private class Matrix3x3 extends AbstractTableModel {
        private int[][] table = {new int[3], new int[3], new int[3]};

        public Matrix3x3() {
            for (int i = 0; i < 3; i++) {
                for (int j = 0; j < 3; j++) {
                    table[i][j] = i * j;
                }
            }
        }

        public int getColumnCount() {return table.length;}
        public int getRowCount() {return table[0].length;}
        public Object getValueAt(int r, int c) {
            return new Integer(table[r][c]);
        }

        public void setValueAt(Object obj, int r, int c) {
            table[r][c] = ((Integer)obj).intValue();
            // Inform the Views/listeners that the Model has changed
            // Since JTable is a listener of its model, this will cause the
            // JTable to update.
            fireTableCellUpdated(r,c);
        }
    }

    public Example2() {
        // declaring these final allows them to be accessed from the inner class
        final AbstractTableModel model = new Matrix3x3();
        final JTable table = new JTable(model);
        getContentPane().add(new JScrollPane(table), BorderLayout.CENTER);
        model.setValueAt(new Integer(1), 0, 0);

        NoFocusButton button = new NoFocusButton("Increment selected cell");
        getContentPane().add(button, BorderLayout.SOUTH);
        button.addActionListener (new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                int row = table.getSelectedRow();
                // since Columns can be reordered by dragging the table headers
                // in the JTable without affecting the column ordering in the
                // model, we have to convert from the JTable ordering to
                // the model's.
                int column = table.convertColumnIndexToModel(
                    table.getSelectedColumn());
                int currentValue = ((Integer)model.getValueAt(
                    row,column)).intValue();
                model.setValueAt(new Integer(currentValue + 1), row, column);
            }
        });
    }
}

```

```

    pack();
}

class NoFocusButton extends JButton {
    public NoFocusButton(String s) {
        super(s);
    }
    public boolean isRequestFocusEnabled()
    {
        return false;
    }
    public boolean isFocusTraversable() {
        return false;
    }
}

public static void main (String arg[]) {
    Example2 ex2 = new Example2();
    ex2.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
    ex2.addWindowListener(new WindowAdapter() {
        public void windowClosed(WindowEvent e) {
            System.exit(0);
        }
    });
    ex2.setVisible(true);
}
}

```

Example: Sorting table

This example shows how a sorting table can act as a wrapper for a regular table. (See the Decorator pattern in the book *Design Patterns* by Gamma, et al.) This is a naive implementation: it uses a simple insertion sort, it always re-sorts on all table changes (even those that wouldn't affect the whole table), and it provides no mechanism for changing the sorting column.

```

// SortingTableModel.java
import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;
import com.sun.java.swing.event.*;
import com.sun.java.swing.table.*;

class SampleSortingTableModel extends AbstractTableModel
    implements TableModelListener
{
    protected TableModel base;
    protected int sortColumn;
    protected int[] row;
    public SampleSortingTableModel (TableModel tm, int sortColumn) {
        this.base = tm;
        this.sortColumn = sortColumn;
        tm.addTableModelListener(this);
        rebuild();
    }

    // These methods can be directly delegated to 'base'.

```

```

public Class getColumnClass(int c) {return base.getColumnClass(c);}
public int getColumnCount() {return base.getColumnCount();}
public String getColumnName(int c) {return base.getColumnName(c);}
public int getRowCount() {return base.getRowCount();}

// These operations can be delegated to 'base' after row translation
public Object getValueAt(int r, int c) {
    return base.getValueAt(row[r], c);
}
public boolean isCellEditable(int r, int c) {
    return base.isCellEditable(row[r], c);
}
public void setValueAt(Object value, int r, int c) {
    base.setValueAt(value, row[r], c); // Notification will cause re-sort
}

// Notifications from base. Naive: always re-sorts & notifies
public void tableChanged(TableModelEvent event) {
    rebuild();
}

// Utility methods
protected void rebuild() {
    int size = base.getRowCount();
    row = new int[size];
    for (int i = 0; i < size; i++) {row[i] = i;}
    sort();
}
protected void sort() { // Sort and notify listeners
    // Simple insertion sort, from Jon Bentley's "Programming Pearls",
    // p.108
    for (int i = 1; i < row.length; i++) {
        // Invariant: row[0] to row[i-1] sorted
        int j = i;
        while (j > 0 && compare(j-1, j) > 0) {
            int temp = row[j];
            row[j] = row[j-1];
            row[j-1] = temp;
            j--;
        }
        fireTableStructureChanged(); // notify _our_ listeners
    }
    // Return <0 if row[i] < row[j], 0 if equal, >0 if row[i] > row[j]
    protected int compare(int i, int j) {
        String s1 = base.getValueAt(row[i], sortColumn).toString();
        String s2 = base.getValueAt(row[j], sortColumn).toString();
        return s1.compareTo(s2);
    }
}

public class SortingTableModel extends JFrame {
    DefaultTableModel model = new DefaultTableModel(
        new Object[][] { {"this", "1"}, {"text", "2"},
            {"will", "3"}, {"be", "4"},
            {"sorted", "5"} },

```

```

        new Object[] {"Column 1", "Column 2"}
    );
    public SortingTableModel() {
        setDefaultCloseOperation(DISPOSE_ON_CLOSE);
        addWindowListener(new WindowAdapter() {
            public void windowClosed(WindowEvent e) {
                System.exit(0);
            }
        });
        getContentPane().add(new JLabel(
            "Original and Sorted on Column 1 Tables: edit a cell"),
            BorderLayout.NORTH);
        JTable tableOrig = new JTable(model);
        tableOrig.setAutoResizeMode(JTable.AUTO_RESIZE_ALL_COLUMNS);
        JTable tableSorted = new JTable(new SampleSortingTableModel(model,0));
        tableSorted.setAutoResizeMode(JTable.AUTO_RESIZE_ALL_COLUMNS);
        JPanel panel = new JPanel(new GridLayout(1,2));
        panel.add(new JScrollPane(tableOrig));
        panel.add(new JScrollPane(tableSorted));
        getContentPane().add(panel, BorderLayout.CENTER);
        pack();
    }
    public static void main (String arg[]) {
        new SortingTableModel().setVisible(true);
    }
}

```

```

public class AbstractTableModel extends java.lang.Object
implements TableModel, java.io.Serializable

```

Constructors

```

public AbstractTableModel()
    Creates an instance of this model.

```

Methods

```

public void addTableModelListener(TableModelListener listener)
    Adds listener to the list of objects to be notified when the table changes.

```

Implements: addTableModelListener in interface TableModel.

```

public int findColumn(String columnName)

```

Returns the index of the (first) column whose name is equal to columnName, or -1 if there is no such column. This is the only public method of AbstractTableModel which is not in the TableModel interface.

```

protected void fireTableCellUpdated(int row, int column)

```

Creates a TableModelEvent, and uses it to notify listeners that the specified cell has changed.

```

protected void fireTableChanged(TableModelEvent event)

```

Uses event to notify listeners that the table has changed. See TableModelEvent for the full range of event possibilities.

```

protected void fireTableDataChanged()

```

Creates a TableModelEvent, and uses it to notify listeners that all data in the table may have changed, including the number of rows, but the table structure has not changed.

protected void fireTableRowsDeleted(int row1, int row2)

Creates a `TableModelEvent`, and uses it to notify listeners that the data from `row1` to `row2` inclusive has been deleted.

protected void fireTableRowsInserted(int row1, int row2)

Creates a `TableModelEvent`, and uses it to notify listeners that rows were inserted from `row1` to `row2` inclusive.

protected void fireTableRowsUpdated(int row1, int row2)

Creates a `TableModelEvent`, and uses it to notify listeners that the data (in all columns) from `row1` to `row2` (inclusive) may have changed.

protected void fireTableStructureChanged()

Creates a `TableModelEvent`, and uses it to notify listeners that the table structure and all data in it have changed. See `JTable`. A `JTable` regards this change as drastic; if its `autoCreateColumnsFromModel` flag is `true`, the column model will be rebuilt.

public Class getColumnClass(int column)

Gets the class for `column`. By providing this information, you help cause the most appropriate renderer to be used. (For example, an entry of type `Boolean` could be displayed as a checkbox, rather than the string `true` or `false`.) You should return the class that is common to all entries in the column. For example, if you have both `Integer` and `Boolean` items in the column, you would need to return `Object.class` (their only common superclass).

Implements: `getColumnClass` in interface `TableModel`.

public String getColumnName(int column)

Returns the name of the specified `column`. Two columns may have the same name. This usually has no impact because `JTable` works with column indexes rather than column names. This method supports the `TableModel` interface. (If you don't override this method, you get a series of labels A, B, C, ..., AA, ..., ZZ, etc.)

public boolean isCellEditable(int row, int column)

Returns `true` if the cell may be edited. If you don't override this method, it always returns `false`.

Implements: `isCellEditable` in interface `TableModel`.

public void removeTableModelListener(TableModelListener listener)

Removes `listener` from the list of objects notified when the table changes.

Implements: `removeTableModelListener` in interface `TableModel`.

public void setValueAt(Object object, int row, int column)

Stores `object` at the specified cell in the table. You should store a value only if `isCellEditable(row, column)` returns `true`. If you don't override this method, it does not store any values.

Implements: `setValueAt` in interface `TableModel`.

Protected Fields

protected EventListenerList listenerList

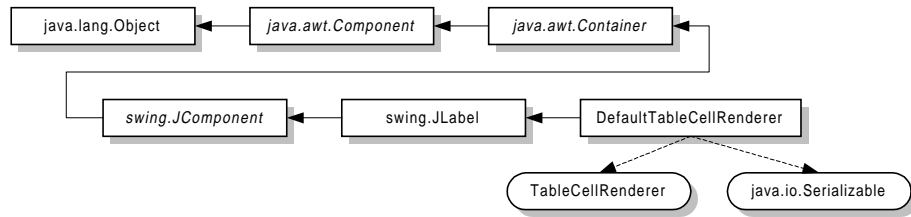
The list of listeners to be notified of changes in the table. See the `fireTableXxx()` methods.

Extended by `DefaultTableModel` (p.747).

See also `EventListenerList` (p.491), `JTable` (p.357), `TableModel` (p.766), `TableModelEvent` (p.508), `TableModelListener` (p.510).

swing.table.DefaultTableCellRenderer

CLASS



This is the renderer used to display table cells when none is specified. It can also be used as a basis for custom renderer subclasses.

To support the `TableCellRenderer` interface this class must return a `Component` to render a cell. Since this class extends `JLabel`, it can return itself.

Example: a `DefaultTableCellRenderer` subclass that colors cells depending on their state and location.

```

package tablecr;

// Main.java - TableCellRenderer Example
import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;
import com.sun.java.swing.table.*;

public class Main extends JFrame {
    DefaultTableModel tmodel = new DefaultTableModel(
        new Object[][] {
            {"some", "text"}, {"any", "text"},
            {"even", "more"}, {"text", "strings"},
            {"and", "other"}, {"text", "values"}
        },
        new Object[] {"Column 1", "Column 2"}
    );

    class MyRenderer implements TableCellRenderer {
        public Component getTableCellRendererComponent(
            JTable table, Object value,
            boolean isSelected, boolean hasFocus,
            int row, int column)
        {
            JTextField editor = new JTextField();
            if (value != null) editor.setText(value.toString());
            editor.setBackground((row%2==0)?Color.white:Color.cyan);
            return editor;
        }
    }

    public Main() {
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                Main.this.dispose();
            }
        });
    }
}

```

```

        System.exit(0);
    }
});

JTable table = new JTable(tmodel);
table.setDefaultRenderer(Object.class, new MyRenderer());
getContentPane().add(new JScrollPane(table), BorderLayout.CENTER);
pack();
}

public static void main (String arg[]) {
    new Main().setVisible(true);
}
}

public class DefaultTableCellRenderer extends JLabel
    implements TableCellRenderer

```

Constructors

```
public DefaultTableCellRenderer()
```

Creates a cell renderer.

Methods

```
public java.awt.Component getTableCellRendererComponent(JTable table,
    Object value, boolean isSelected, boolean hasFocus,
    int row, int column)
```

Handles all rendering (display) of the table cells. The `table` and `value` may be null. If `isSelected` is true, the row is currently selected (and probably should be displayed in a highlight color). If `hasFocus` is true, the row has focus (and may be displayed with special highlight—for example, a dashed border). The `row` and `column` tell the location of the object in the table. The return value is a component configured to display the cell.

Implements: `getTableCellRendererComponent` in interface `TableCellRenderer`.

```
public void setBackground(java.awt.Color color)
```

Sets the background color of the renderer.

Overrides: `setBackground` in class `Component`.

```
public void setForeground(java.awt.Color color)
```

Sets the foreground color of the renderer.

Overrides: `setForeground` in class `Component`.

```
public void updateUI()
```

Tells the renderer to update itself according to the current look and feel.

Protected Fields

```
protected static Border noFocusBorder
```

This is the border to use when the cell does not have the focus.

Overrides: `updateUI` in class `JLabel`.

Protected Methods

```
protected void setValue(Object value)
```

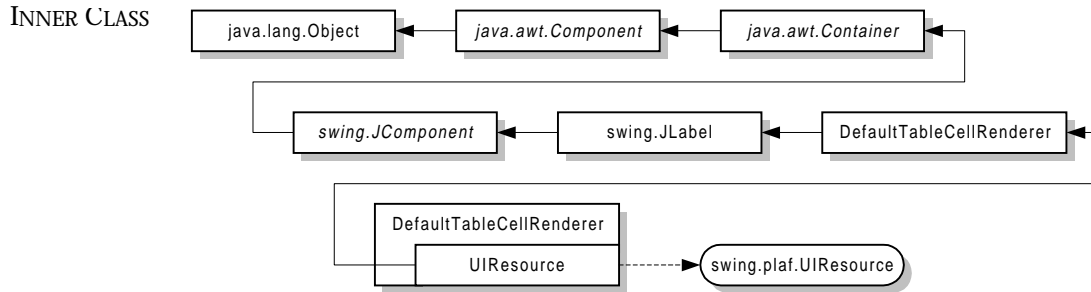
Sets the value that this renderer is supposed to display.

Inner Classes

```
public static class DefaultTableCellRenderer.UIResource
```

See also Border (p.465), JLabel (p.264), JTable (p.357), TableCellRenderer (p.760).

swing.table.DefaultTableCellRenderer.UIResource



A subclass of `DefaultTableCellRenderer` that implements `UIResource`. This class can be placed in the `UIDefaults` table.

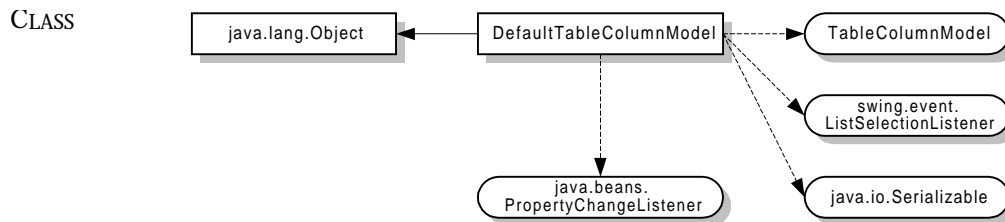
```
public static class DefaultTableCellRenderer.UIResource
    extends DefaultTableCellRenderer
    implements UIResource
```

Constructors

```
public UIResource()
    Default constructor.
```

See also `UIResource` (p.544).

swing.table.DefaultTableModel



The `DefaultTableModel` is an implementation of `TableColumnModel`. It tracks all columns in a table, and which ones are selected. This object will notify its listeners of changes in the columns, selection, or margins.

```
public class DefaultTableModel extends java.lang.Object
    implements TableColumnModel, java.beans.PropertyChangeListener,
    ListSelectionListener, java.io.Serializable
```

Constructors

public DefaultTableModel()

Creates a table column model.

Methods

public void addColumn(TableColumn column)

Makes `column` the new last column of the model. Notifies any listeners of the change using the `columnAdded()` method of `swing.event.TableColumnModelListener`.

Implements: `addColumn` in interface `TableColumnModel`.

public void addColumnModelListener(TableColumnModelListener listener)

Adds `listener` to the list of objects to be notified when the column model changes (for example, when columns are added or removed).

Implements: `addColumnModelListener` in interface `TableColumnModel`.

public TableColumn getColumn(int column)

Gets the column information for the specified `column`.

Implements: `getColumn` in interface `TableColumnModel`.

public int getColumnCount()

Returns the number of columns.

Implements: `getColumnCount` in interface `TableColumnModel`.

public int getColumnIndex(Object columnName)

Finds the index of the column for which `column.getIdentifier().equals(columnName)`.

Throws: `IllegalArgumentException` if `columnName` is null or no `TableColumn` has this name.

Implements: `getColumnIndex` in interface `TableColumnModel`.

public int getColumnIndexAtX(int xPosition)

Finds the index of the column at `xPosition`, or `-1` if there is no such column.

Implements: `getColumnIndexAtX` in interface `TableColumnModel`.

public int getColumnMargin()

Gets the width (in pixels) of the margin between adjacent columns. Notifies any listeners of the change using the `columnMarginChanged()` method of `TableColumnModelListener`. The default `columnMargin` is 2.

Implements: `getColumnMargin` in interface `TableColumnModel`.

public boolean getColumnSelectionAllowed()

Returns true if columns may be selected.

Implements: `getColumnSelectionAllowed` in interface `TableColumnModel`.

public java.util.Enumeration getColumns()

Returns the list of columns, in order.

Implements: `getColumns` in interface `TableColumnModel`.

public int getSelectedColumnCount()

Returns the number of selected columns.

Implements: `getSelectedColumnCount` in interface `TableColumnModel`.

public int[] getSelectedColumns()

Returns a list of the selected columns. This never returns `null`, but may return an array of 0 elements.

Implements: `getSelectedColumns` in interface `TableColumnModel`.

public ListSelectionModel getSelectionModel()

Returns the `ListSelectionModel` that is used to maintain column selection state. This method returns `null` if row selection is not allowed. See also `setSelectionModel`.

Implements: `getSelectionModel` in interface `TableColumnModel`.

public int getTotalColumnWidth()

Returns the sum of all column widths.

Implements: `getTotalColumnWidth` in interface `TableColumnModel`.

public void moveColumn(int fromIndex, int toIndex)

Moves the column at `fromIndex` to be located at `toIndex` (other columns will be shifted to make room). For example, if you have columns 012345, and move column 2 to position 4, you will get 013425. If you have columns 012345, and move column 4 to position 2, you will get 014235.

Also notifies any listeners of the change using the `columnMoved()` method of `TableColumnModelListener`.

Throws: `IllegalArgumentException` if `column` or `newIndex` are not in the valid range.

Implements: `moveColumn` in interface `TableColumnModel`.

public void propertyChange(java.beans.PropertyChangeEvent event)

Supports the `PropertyChangeListener` interface, and lets the model listen for property changes.

Implements: `propertyChange` in interface `PropertyChangeListener`.

public void removeColumn(TableColumn column)

Removes the specified `column` if it is present. Shifts other columns to adjust for this removal and notifies any listeners of the change using the `columnRemoved()` method of `TableColumnModelListener`.

Implements: `removeColumn` in interface `TableColumnModel`.

public void removeColumnModelListener(

TableColumnModelListener listener)

Removes `listener` from the list of objects to be notified of column changes.

Implements: `removeColumnModelListener` in interface `TableColumnModel`.

public void setColumnMargin(int width)

Sets the margin between columns to `width` (in pixels) and notifies any listeners using a `columnMarginChanged()` event.

Implements: `setColumnMargin` in interface `TableColumnModel`.

public void setColumnSelectionAllowed(boolean maySelect)

Sets whether columns may be selected.

Implements: `setColumnSelectionAllowed` in interface `TableColumnModel`.

public void setSelectionModel(ListSelectionModel selectionModel)

Sets the selection model for the columns. This method also registers this `TableColumnModel` for event notifications from the new selection model. If `selectionModel` is `null`, columns are not selectable.

Implements: `setSelectionModel` in interface `TableColumnModel`.

public void valueChanged(ListSelectionEvent event)

Supports the `ListSelectionListener` interface, and lets the model listen for changes in the selection.

Implements: `valueChanged` in interface `ListSelectionListener`.

Protected Fields

protected transient ChangeEvent changeEvent

The event used to notify listeners of changes in the column model.

`protected int columnMargin`
 The margin between cells.

`protected boolean columnSelectionAllowed`
 Set to `true` if columns may be selected.

`protected EventListenerList listenerList`
 The list of listeners to be notified of changes in the column model.

`protected ListSelectionModel selectionModel`
 The selection model for the columns.

`protected java.util.Vector tableColumns`
 The list of columns. (Each element is a `TableColumn`.)

`protected int totalColumnWidth`
 The total of all column widths (including margins).

Protected Methods

`protected ListSelectionModel createSelectionModel()`
 Creates the default selection model for the column model.

`protected void fireColumnAdded(TableColumnModelEvent event)`
 Notifies listeners that a column has been added.

`protected void fireColumnMarginChanged()`
 Notifies listeners that the column margin has changed.

`protected void fireColumnMoved(TableColumnModelEvent event)`
 Notifies listeners that a column has been moved.

`protected void fireColumnRemoved(TableColumnModelEvent event)`
 Notifies listeners that a column has been removed.

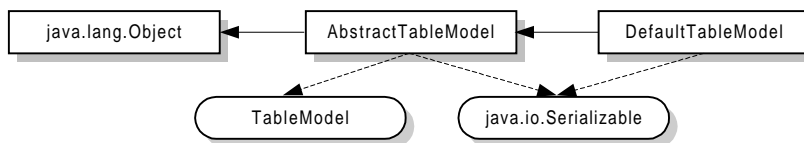
`protected void fireColumnSelectionChanged(ListSelectionEvent event)`
 Notifies listeners that the selection has changed.

`protected void recalcWidthCache()`
 Recalculates the total width after a column or the margin changes its size.

See also `JTable` (p.357), `ListSelectionListener` (p.500), `ListSelectionModel` (p.421), `TableColumnModel` (p.764), `TableColumnModelListener` (p.508).

swing.table.DefaultTableModel

CLASS



The `DefaultTableModel` is used by `JTable` when you don't provide a table model. It uses a `Vector` to manage a list of rows. Each row item is a `Vector` of the columns for that row. It is almost always preferable to subclass `AbstractTableModel` rather than use this class. Subclassing `AbstractTableModel` requires a minimal amount of code and allows your data to stay in its native data structure. Using `DefaultTableModel` forces the data in your application to be copied back and forth between the native data structure and the data structure used by `DefaultTa-`

bleModel. This is almost certainly not what you want to do due to its CPU and memory usage implications. The only time you might prefer `DefaultTableModel` is when your data is dynamic and has no data structure holding it, or when you need to implement a quick prototype.

Example

```
package dtm;

// Main.java - DefaultTableModel example
import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;
import com.sun.java.swing.table.*;

public class Main extends JFrame {
    DefaultTableModel model = new DefaultTableModel(
        new Object[][] {
            {"some", "text"}, {"any", "text"},
            {"even", "more"}, {"text", "strings"},
            {"and", "other"}, {"text", "values"}
        },
        new Object[] {"Column 1", "Column 2"}
    );

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

        JTable table = new JTable(model);
        getContentPane().add(new JScrollPane(table), BorderLayout.CENTER);
        pack();
    }

    public static void main (String arg[]) {
        new Main().setVisible(true);
    }
}

public class DefaultTableModel extends AbstractTableModel
    implements java.io.Serializable
```

Constructors

```
public DefaultTableModel()
```

```
public DefaultTableModel(int rows, int columns)
```

Creates a table, with the specified number of rows and columns. (*Default*: 0 rows, 0 columns; any cells default to null.)

```
public DefaultTableModel(Object[] columnNames, int rows)
```

```
public DefaultTableModel(java.util.Vector columnNames, int rows)
```

Creates a table with the specified set of column names, and the specified number of rows. All data values default to null. If `columnNames` is null, the table has 0 columns.

```
public DefaultTableModel(Object[][] data, Object[] columnNames)
    Creates a table with the specified data and column names. The value in data[r][c] will be returned as getValueAt(r, c).
```

```
public DefaultTableModel(java.util.Vector data,
    java.util.Vector columnNames)
    Creates a table with the specified data and column names. The value in data.elementAt(r).elementAt(c) will be returned by getValueAt(r, c). (The data Vector's elements are each a Vector of Object.)
```

Public Methods

```
public void addColumn(Object columnName)
    Adds a new column to the table, with the specified name and all values null. The columnName may not be null. Notifies any listeners of the change.
    Throws: IllegalArgumentException if columnName is null.
```

```
public void addColumn(Object columnName, Object[] data)
    Adds a new column to the table, with the specified name, and all values filled from the data array. The columnName may not be null. If data is null, all values are set to null.
    Throws: IllegalArgumentException if columnName is null.
```

```
public void addColumn(Object columnName, Vector data)
    Adds a new column to the table, with the specified name, and all data values filled from the data vector. The columnName may not be null. If data is null, all values are set to null.
    Throws: IllegalArgumentException if columnName is null.
```

```
public void addRow(Object[] rowData)
    Puts rowData in a new Vector, and adds it as the new last row of the table. If rowData is null, all values are set to null. Notifies any listeners of the change.
```

```
public void addRow(java.util.Vector rowData)
    Adds rowData as the new last row of the table. If rowData is null, all values are set to null. Notifies any listeners of the change.
```

```
public int getColumnCount()
    Returns the number of columns.
    Implements: getColumnCount in interface TableModel.
```

```
public String getColumnName(int column)
    Returns the name of the specified column. Two columns may have the same name. This usually has no impact because JTable usually works with column indexes rather than column names. This method supports the TableModel interface. (If no value has been established for the column name, the AbstractTableModel method is called, giving a series of labels "A," "B," "C," etc.)
    Overrides: getColumnName in class AbstractTableModel.
    Implements: getColumnName in interface TableModel.
```

```
public java.util.Vector getDataVector()
    Returns the data which is a Vector with each entry a Vector of Object. The number of items in the returned Vector is the number of rows. You may modify this data either by changing the data in a particular cell or by changing the number of rows, provided you call newDataAvailable(), newRowsAdded(), or rowsRemoved() with an appropriate event, so listeners will be notified. (You may not directly modify the number of columns; use addColumn() or removeColumn() to do that.)
```

public int getRowCount()

Returns the number of rows. This routine should be efficient because it is called by `JTable` very frequently.

Implements: `getRowCount` in interface `TableModel`.

public Object getValueAt(int row, int column)

Returns the value at the specified row and column.

Throws: `ArrayIndexOutOfBoundsException` if an invalid row or column was given.

Implements: `getValueAt` in interface `TableModel`.

public void insertRow(int rowIndex, Object[] rowData)

Inserts `rowData` at `rowIndex`. The `rowData` is first put into a new `Vector`. (If `rowData` is `null`, all values will be `null`.) Rows after `rowIndex` are moved down, and listeners are notified of the change.

Throws: `ArrayIndexOutOfBoundsException` if the row was invalid.

public void insertRow(int rowIndex, java.util.Vector rowData)

Inserts `rowData` at `rowIndex`. The `rowData` is a `Vector` of `Objects`, each object corresponding to the value at the corresponding column. (If `rowData` is `null`, all values will be `null`.) Rows after `rowIndex` are moved down, and listeners are notified of the change.

Throws: `ArrayIndexOutOfBoundsException` if the row was invalid.

public boolean isCellEditable(int row, int column)

Returns `true` if the cell may be edited. This method supports the `TableModel` interface. If this is `false`, `setValueAt()` has no effect. See also `setValueAt`.

Overrides: `isCellEditable` in class `AbstractTableModel`.

Implements: `isCellEditable` in interface `TableModel`.

public void moveRow(int startRow, int endRow, int newRow)

Moves the rows ranging from `startRow` to `endRow` so that they are at `newRow`. All rows must be in the range `[0, getRowCount()-1]`, and you must have `startRow <= endRow`. All indexes are specified by their position *before* the move.

Throws: `ArrayIndexOutOfBoundsException` if any of the indices are out of range, or if `endRow` is less than `startRow`.

public void newDataAvailable(TableModelEvent event)

Notifies listeners of the change in the table using `event`. If `event` is `null`, or if the number of rows in the data `Vector` doesn't match the number of rows as tracked by the `DefaultTableModel`, an event is created that reports all data changed.

public void newRowsAdded(TableModelEvent event)

Notifies listeners of the change in the table using `event`.

public void removeRow(int row)

Removes the specified row. Notifies any listeners of the change.

Throws: `ArrayIndexOutOfBoundsException` if the row was invalid.

public void rowsRemoved(TableModelEvent event)

Notifies listeners of the change in the table using `event`. The event may not be `null`.

Throws: `IllegalArgumentException` if `event` is `null`.

public void setColumnIdentifiers(Object[] names)

Stores the names as the new column names.

public void setColumnIdentifiers(java.util.Vector names)

Stores the names as the new column names.

public void setDataVector(Object[][] data, Object[] columnNames)

Clears the table, and repopulates it with `data`. The number of rows and columns in the table is recalculated, the column names are reset, and any listeners are notified of the change.

Throws: `IllegalArgumentException` if `newData` is null or if the number of columns in `data` does not equal the number of the column identifiers in `columnNames`.

public void setDataVector(java.util.Vector data, java.util.Vector columnNames)

Clears the table, and repopulates it with `data`. Each element of `data` should be a `Vector` of `Objects`. The `data` may not be null. (The number of rows and columns in the table is recalculated.) Each row should have the same number of values, the same as the number of `columnNames`. If `columnNames` is null, the old column names are retained. Notifies any listeners of the change.

Throws: `IllegalArgumentException` if `data` is null or if the number of columns in `data` does not equal the number of the column identifiers in `columnNames`.

public void setNumRows(int rows)

Sets the number of rows in the model to `rows`. If new rows are required, their values are null. Notifies any listeners of the change. See also `setColumnIdentifiers`.

public void setValueAt(Object object, int row, int column)

Stores `object` at the specified cell in the table. The value is stored only if `isCellEditable(row, column)` returns true. If you don't override this method, it does not store any values.

Throws: `ArrayIndexOutOfBoundsException` if an invalid row or column was given.

Overrides: `setValueAt` in class `AbstractTableModel`.

Implements: `setValueAt` in interface `TableModel`.

Protected Fields

protected java.util.Vector columnIdentifiers

A `Vector` of the column names.

protected java.util.Vector dataVector

The `Vector` of `Vector` of `Object` values, representing the data in the table.

Protected Methods

protected static java.util.Vector convertToVector(Object[] data)

Returns a `Vector`, populated from `data`.

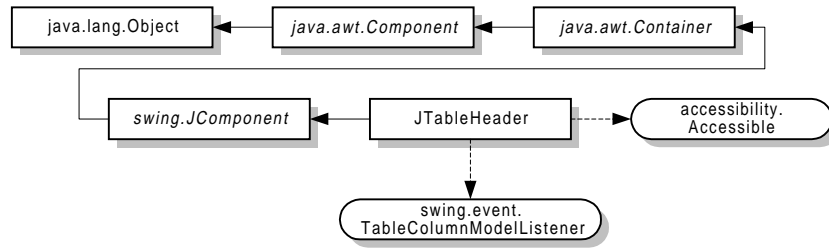
protected static java.util.Vector convertToVector(Object[][] data)

Returns a `Vector` of `Vector` of `Object`. Each item in the outermost vector corresponds to `data[r]`, and each element in the inner vector corresponds to `data[r][c]`.

See also `AbstractTableModel` (p.735), `TableModel` (p.766).

swing.table.JTableHeader

CLASS



This component represents the table header—the label area at the top of the columns in a `JTable`. Its model is the same `TableColumnModel` that the associated `JTable` uses.

```
public class JTableHeader extends JComponent
    implements TableColumnModelListener, Accessible
```

Constructors

```
public JTableHeader()
```

```
public JTableHeader(TableColumnModel columnModel)
```

Creates the table header object. *Default:* column model created by `createDefaultColumnModel()`.

Methods

```
public void columnAdded(TableColumnModelEvent event)
```

Returns notification that a column has been added.

Implements: `columnAdded` in interface `TableColumnModelListener`.

```
public int columnAtPoint(java.awt.Point point)
```

Returns the column index corresponding to `point`, or `-1` if none does.

```
public void columnMarginChanged(ChangeEvent event)
```

Returns notification that the margin of a column has changed.

Implements: `columnMarginChanged` in interface `TableColumnModelListener`.

```
public void columnMoved(TableColumnModelEvent event)
```

Returns notification that a column has moved.

Implements: `columnMoved` in interface `TableColumnModelListener`.

```
public void columnRemoved(TableColumnModelEvent event)
```

Returns notification that a column has been deleted.

Implements: `columnRemoved` in interface `TableColumnModelListener`.

```
public void columnSelectionChanged(ListSelectionEvent event)
```

Returns notification that the column selection has changed.

Implements: `columnSelectionChanged` in interface `TableColumnModelListener`.

```
public AccessibleContext getAccessibleContext()
```

Returns the accessible context for the table header.

Overrides: `getAccessibleContext` in class `JComponent`.

Implements: `getAccessibleContext` in interface `Accessible`.

```
public TableColumnModel getColumnModel()
```

Gets the column model used by the table header. See also `setColumnModel`.

```
public TableColumn getDraggedColumn()
    If a column is being dragged, returns the index of that column (if no column is being dragged,
    returns null).
public int getDraggedDistance()
    Returns the distance (in pixels along the x axis) that the column has been dragged. Defined only
    when a drag is in progress.
public java.awt.Rectangle getHeaderRect(int column)
    Returns the rectangle corresponding to the header's location for column.
public boolean getReorderingAllowed()
    Returns true if the user may rearrange the columns in the table by dragging (the application may
    rearrange the columns regardless of this value).
public boolean getResizingAllowed()
    Returns true if the user may resize the table header columns by dragging (the application may resize
    the columns regardless of this value).
public TableColumn getResizingColumn()
    Returns the index of the column being resized. If no column is being resized, this method returns
    null.
public JTable getTable()
    Gets the table with which this object is associated.
public String getToolTipText(java.awt.event.MouseEvent event)
    Gets the ToolTip text from the header's renderer.
    Overrides: getToolTipText in class JComponent.
public TableHeaderUI getUI()
    Returns the UI (look and feel) for this object.
public String getUIClassID()
    Returns TableHeaderUI.
    Overrides: getUIClassID in class JComponent.
public boolean getUpdateTableInRealTime()
    Returns true if the table should be displayed during dragging and resizing. May be set false to
    improve performance (default value is true).
public void resizeAndRepaint()
    Resets the sizes of the header columns and requests a repaint. Also resets rectangles for the header
    view and line scroll amounts for the Scrollable interface.
public void setColumnModel(TableColumnModel model)
    Makes model the new column model for this view. Registers with the model as a TableColumn-
ModelListener.
    Throws: IllegalArgumentException if model is null.
public void setDraggedColumn(TableColumn column)
    Sets the index of the column currently being dragged.
public void setDraggedDistance(int deltaX)
    Sets the distance that the dragged column should be moved from its original position.
public void setReorderingAllowed(boolean mayReorder)
    Sets whether the user can reorder columns by dragging column headers.
public void setResizingAllowed(boolean mayResize)
    Sets whether the user can resize columns by dragging.
```

```
public void setResizingColumn(TableColumn column)
    Sets the index of the column currently being resized.
public void setTable(JTable table)
    Sets the table with which this table header is associated.
public void setUI(TableHeaderUI ui)
    Sets the look and feel for this component.
public void setUpdateTableInRealTime(boolean shouldUpdateInRealTime)
    Sets whether the table updates in real time when a column header is resized or dragged.
public void updateUI()
    Updates the header using the current UI.
    Overrides: updateUI in class JComponent.
```

Protected Fields

```
protected TableColumnModel columnModel
    The TableColumnModel of the table header.
protected transient TableColumn draggedColumn
    The column currently being dragged (or null if none).
protected transient int draggedDistance
    The distance by which the currently dragged column has been moved (meaningful only during a drag).
protected boolean reorderingAllowed
    Set to true if the user may rearrange the columns.
protected boolean resizingAllowed
    Set to true if the user may resize the columns.
protected transient TableColumn resizingColumn
    The column being resized (or null if no column is resizing)
protected JTable table
    The table for which this is the header.
protected boolean updateTableInRealTime
    If this flag is true, then the whole table will repaint as a column is dragged or resized. If this value is false, only the header will repaint.
```

Protected Methods

```
protected TableColumnModel createDefaultColumnModel()
    Returns the default column model object which is a DefaultTableColumnModel. Subclasses can override this method to return a different column model object.
protected void initializeLocalVars()
    Sets the default values for the fields.
protected String paramString()
    Returns a string representation of the properties of this JTableHeader. The returned string may be empty but it will not be null.
    Overrides: paramString in class JComponent.
```

Inner Classes

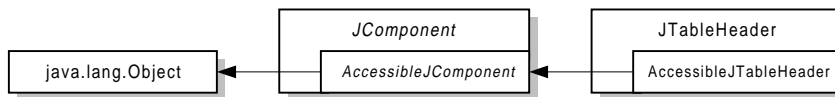
```
protected class JTableHeader.AccessibleJTableHeader
```

Returned by JTable.createDefaultTableHeader() (p.357), JTable.getTableHeader() (p.357).

See also Accessible (p.85), AccessibleContext (p.90), ChangeEvent (p.487), JComponent (p.207), JTable (p.357), ListSelectionEvent (p.499), TableColumnModel (p.764), TableColumnModelEvent (p.507), TableColumnModelListener (p.508), TableHeaderUI (p.541).

swing.table.JTableHeader.AccessibleJTableHeader

INNER CLASS



This inner class is the class returned when you call `getAccessibleContext().getAccessibleComponent()` on a `JTableHeader`.

```
protected class JTableHeader.AccessibleJTableHeader
    extends AccessibleJComponent
```

Constructors

```
public AccessibleJTableHeader()
    Default constructor.
```

Methods

```
public Accessible getAccessibleAt(java.awt.Point p)
    Returns the Accessible child, if one exists, contained at the local coordinate Point.
public Accessible getAccessibleChild(int index)
    Returns the Accessible child of the object at index.
public int getAccessibleChildrenCount()
    Returns the number of accessible children in the object. If all of the children of this object implement
    Accessible, then this method should return the number of children of this object.
public AccessibleRole getAccessibleRole()
    Returns AccessibleRole.PANEL.
```

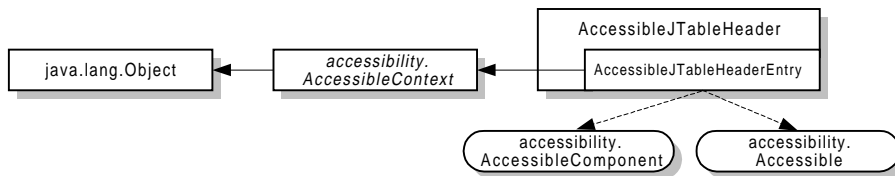
Inner Classes

```
protected class
    JTableHeader.AccessibleJTableHeader.AccessibleJTableHeaderEntry
```

See also Accessible (p.85), AccessibleRole (p.95), JComponent.AccessibleJComponent (p.221).

swing.table.JTableHeader.AccessibleJTableHeader.AccessibleJTableHeaderEntry

INNER CLASS



This inner class is the class that gets returned when `getAccessibleChild()` is called on a `JTableHeader`.

protected class

```
JTableHeader.AccessibleJTableHeader.AccessibleJTableHeaderEntry  
extends AccessibleContext  
implements Accessible, AccessibleComponent
```

Constructors

```
public AccessibleJTableHeaderEntry(int index, JTableHeader header,  
    JTable table)
```

Methods

```
public void addFocusListener(java.awt.event.FocusListener l)
```

Implements: `addFocusListener` in interface `AccessibleComponent`.

```
public void addPropertyChangeListener(  
    java.beans.PropertyChangeListener l)
```

Overrides: `addPropertyChangeListener` in class `AccessibleContext`.

```
public boolean contains(java.awt.Point p)
```

Implements: `contains` in interface `AccessibleComponent`.

```
public AccessibleAction getAccessibleAction()
```

Overrides: `getAccessibleAction` in class `AccessibleContext`.

```
public Accessible getAccessibleAt(Point p)
```

Implements: `getAccessibleAt` in interface `AccessibleComponent`.

```
public Accessible getAccessibleChild(int i)
```

Overrides: `getAccessibleChild` in class `AccessibleContext`.

```
public int getAccessibleChildrenCount()
```

Overrides: `getAccessibleChildrenCount` in class `AccessibleContext`.

```
public AccessibleComponent getAccessibleComponent()
```

Overrides: `getAccessibleComponent` in class `AccessibleContext`.

```
public AccessibleContext getAccessibleContext()
```

Gets the `AccessibleContext` associated with this object.

Implements: `getAccessibleContext` in interface `Accessible`.

```
public String getAccessibleDescription()
```

Overrides: `getAccessibleDescription` in class `AccessibleContext`.

```
public int getAccessibleIndexInParent()
```

Overrides: `getAccessibleIndexInParent` in class `AccessibleContext`.

```
public String getAccessibleName()
```

Overrides: `getAccessibleName` in class `AccessibleContext`.

```
public AccessibleRole getAccessibleRole()
```

Overrides: `getAccessibleRole` in class `AccessibleContext`.

```
public AccessibleSelection getAccessibleSelection()
```

Overrides: `getAccessibleSelection` in class `AccessibleContext`.

```
public AccessibleStateSet getAccessibleStateSet()
```

Overrides: `getAccessibleStateSet` in class `AccessibleContext`.

```
public AccessibleText getAccessibleText()
```

Overrides: `getAccessibleText` in class `AccessibleContext`.

```
public AccessibleValue getAccessibleValue()  
    Overrides: getAccessibleValue in class AccessibleContext.  
public java.awt.Color getBackground()  
    Implements: getBackground in interface AccessibleComponent.  
public java.awt.Rectangle getBounds()  
    Implements: getBounds in interface AccessibleComponent.  
public java.awt.Cursor getCursor()  
    Implements: getCursor in interface AccessibleComponent.  
public java.awt.Font getFont()  
    Implements: getFont in interface AccessibleComponent.  
public java.awt.FontMetrics getFontMetrics(java.awt.Font f)  
    Implements: getFontMetrics in interface AccessibleComponent.  
public java.awt.Color getForeground()  
    Implements: getForeground in interface AccessibleComponent.  
public java.util.Locale getLocale()  
    Overrides: getLocale in class AccessibleContext.  
public java.awt.Point getLocation()  
    Implements: getLocation in interface AccessibleComponent.  
public java.awt.Point getLocationOnScreen()  
    Implements: getLocationOnScreen in interface AccessibleComponent.  
public java.awt.Dimension getSize()  
    Implements: getSize in interface AccessibleComponent.  
public boolean isEnabled()  
    Implements: isEnabled in interface AccessibleComponent.  
public boolean isFocusTraversable()  
    Implements: isFocusTraversable in interface AccessibleComponent.  
public boolean isShowing()  
    Implements: isShowing in interface AccessibleComponent.  
public boolean isVisible()  
    Implements: isVisible in interface AccessibleComponent.  
public void removeFocusListener(java.awt.event.FocusListener l)  
    Implements: removeFocusListener in interface AccessibleComponent.  
public void removePropertyChangeListener(  
    java.beans.PropertyChangeListener l)  
    Overrides: removePropertyChangeListener in class AccessibleContext.  
public void requestFocus()  
    Implements: requestFocus in interface AccessibleComponent.  
public void setAccessibleDescription(String s)  
    Overrides: setAccessibleDescription in class AccessibleContext.  
public void setAccessibleName(String s)  
    Overrides: setAccessibleName in class AccessibleContext.  
public void setBackground(java.awt.Color c)  
    Implements: setBackground in interface AccessibleComponent.  
public void setBounds(java.awt.Rectangle r)  
    Implements: setBounds in interface AccessibleComponent.  
public void setCursor(java.awt.Cursor c)  
    Implements: setCursor in interface AccessibleComponent.
```

public void setEnabled(boolean b)

Implements: setEnabled in interface AccessibleComponent.

public void setFont(java.awt.Font f)

Implements: setFont in interface AccessibleComponent.

public void setForeground(java.awt.Color c)

Implements: setForeground in interface AccessibleComponent.

public void setLocation(java.awt.Point p)

Implements: setLocation in interface AccessibleComponent.

public void setSize(java.awt.Dimension d)

Implements: setSize in interface AccessibleComponent.

public void setVisible(boolean b)

Implements: setVisible in interface AccessibleComponent.

See also Accessible (p.85), AccessibleAction (p.86), AccessibleComponent (p.87), AccessibleContext (p.90), AccessibleRole (p.95), AccessibleStateSet (p.101), AccessibleText (p.103), AccessibleValue (p.104), JTable (p.357), JTableHeader (p.752).

swing.table.TableCellEditor

INTERFACE

TableCellEditor

A TableCellEditor provides components to edit cells. One editor can serve as the editor for a whole table or just a particular column or cell.

See “Swing, color, and the UIDefaults table” on p.51 for information on selecting the right colors and borders to match the Component returned by this method to the defaults for the table.

Example: Setting a cell editor for an entire table

```
// File tablecelleditor/Main.java - TableCellEditor Example
package tablecelleditor;

import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;
import com.sun.java.swing.table.*;

public class Main extends JFrame {
    DefaultTableModel model = new DefaultTableModel(
        new Object[][] {
            {"some", "text"}, {"any", "text"},
            {"even", "more"}, {"text", "strings"},
            {"and", "other"}, {"text", "values"}
        },
        new Object[] {"Column 1", "Column 2"}
    );

    class MyEditor extends DefaultCellEditor {
        public MyEditor() {super(new JTextField());}
    }
}
```

```

public Component getTableCellEditorComponent(JTable table, Object value,
                                             boolean isSelected,
                                             int row, int column) {
    JTextField editor = (JTextField)super.getTableCellEditorComponent(
        table, value, isSelected, row, column);

    if (value != null) editor.setText(value.toString());
    if (column == 0) {
        editor.setHorizontalAlignment(SwingConstants.CENTER);
        editor.setFont(new Font("Serif", Font.BOLD, 14));
    } else {
        editor.setHorizontalAlignment(SwingConstants.RIGHT);
        editor.setFont(new Font("Serif", Font.ITALIC, 12));
    }
    return editor;
}
}

public Main() {
    this.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            Main.this.dispose();
            System.exit(0);
        }
    });
    JTable table = new JTable(model);
    table.setDefaultEditor(Object.class, new MyEditor());
    getContentPane().add(new JScrollPane(table), BorderLayout.CENTER);
    pack();
}

public static void main(String arg[]) {
    new Main().setVisible(true);
}
}

```

public interface TableCellEditor extends CellEditor

Methods

public abstract java.awt.Component getTableCellEditorComponent(

JTable table, Object value, boolean isSelected, int row, int column)

Handles all editing of table cells. The `table` and `value` may be null. If `isSelected` is true, the row is currently selected (and probably should be displayed in a highlight color). The `row` and `column` tell the location of the object in the table.

The return value is a component configured to edit the cell. You can make the return value depend on whatever you want. For example, you might return a checkbox if `value` is `Boolean`, and a label for any other type.

Cell editors can be established either on a table basis (see `JTable`) or a column basis (see `TableColumn`).

Returned by `JTable.getCellEditor()` (p.357), `JTable.getDefaultEditor()` (p.357), `TableColumn.getCellEditor()` (p.761).

Implemented by `DefaultCellEditor` (p.164).

See also `JTable` (p.357), `TableColumn` (p.761).

swing.table.TableCellRenderer

INTERFACE

TableCellRenderer

A `TableCellRenderer` draws the cell's representation on the screen. A single renderer can serve as the “cookie cutter” to display all cells of the table. You can set the renderer for a whole table, or for a particular column.

If no renderer has been provided, a `DefaultTableCellRenderer` will be created and used.

See “Swing, color, and the `UIDefaults` table” on p.51 for information on selecting the right colors and borders to match the `Component` returned by this method to the defaults for the table.

Example

```
package tablecr;

// Main.java - TableCellRenderer Example
import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;
import com.sun.java.swing.table.*;

public class Main extends JFrame {
    DefaultTableModel tmodel = new DefaultTableModel(
        new Object[][] {
            {"some", "text"}, {"any", "text"},
            {"even", "more"}, {"text", "strings"},
            {"and", "other"}, {"text", "values"}
        },
        new Object[] {"Column 1", "Column 2"}
    );

    class MyRenderer implements TableCellRenderer {
        public Component getTableCellRendererComponent(
            JTable table, Object value,
            boolean isSelected, boolean hasFocus,
            int row, int column)
        {
            JTextField editor = new JTextField();
            if (value != null) editor.setText(value.toString());
            editor.setBackground((row%2==0)?Color.white:Color.cyan);
            return editor;
        }
    }

    public Main() {
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                Main.this.dispose();
                System.exit(0);
            }
        });
    }
}
```

```

    });
    JTable table = new JTable(tmodel);
    table.setDefaultRenderer(Object.class, new MyRenderer());
    getContentPane().add(new JScrollPane(table), BorderLayout.CENTER);
    pack();
}

public static void main (String arg[]) {
    new Main().setVisible(true);
}
}

```

public interface TableCellRenderer

Methods

public abstract java.awt.Component getTableCellRendererComponent(
JTable table, Object value, boolean isSelected,
boolean hasFocus, int row, int column)

Handles all rendering (display) of the table cells. The `table` and `value` may be null. If `isSelected` is true, the row is currently selected (and probably should be displayed in a highlight color). If `hasFocus` is true, the row has focus (and may be displayed with special highlight—a dashed border, for example). The `row` and `column` tell the location of the object in the table.

The return value is a component configured to display the cell. You can make the return value depend on whatever you want. For example, you might return a checkbox if `value` is `Boolean`, and a label for any other type.

Renderers can be established either on a table basis (see `JTable`) or a column basis (see `TableColumn`).

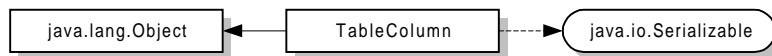
Returned by `BasicTableUI.getCellRenderer()` (p.690), `JTable.getDefaultRenderer()` (p.357), `TableColumn.createDefaultHeaderRenderer()` (p.761), `TableColumn.getCellRenderer()` (p.761), `TableColumn.getHeaderRenderer()` (p.761).

Implemented by `DefaultTableCellRenderer` (p.742).

See also `JTable` (p.357), `TableColumn` (p.761).

swing.table.TableColumn

CLASS



This is the model for a single table column (`TableColumnModel` is the model for all columns in a table). Note that tables are not symmetric: there is a model for the data and a model for the columns, but no model for the rows.

```

public class TableColumn extends java.lang.Object
    implements java.io.Serializable

```

Fields

```

public static final String CELL_RENDERER_PROPERTY = "cellRenderer"

```

The property name for the renderer for nonheader cells.

```
public static final String COLUMN_WIDTH_PROPERTY = "columnWidth"
```

The property name for the width of the column (in pixels).

```
public static final String HEADER_RENDERER_PROPERTY = "headerRenderer"
```

The property name for the renderer for header cells.

```
public static final String HEADER_VALUE_PROPERTY = "headerValue"
```

The property name for the value of the header.

Constructors

```
public TableColumn()
```

Do not use this constructor. It is here for the use of Beans tools and Serialization.

```
public TableColumn(int modelIndex)
```

Creates a `TableColumn` for the given index in the model. The default renderer will be used (a `JLabel` subclass) and no editor will be assigned.

```
public TableColumn(int modelIndex, int width)
```

Creates a `TableColumn` of the given width for the given index in the model. The default renderer will be used (a `JLabel` subclass) and no editor will be assigned.

```
public TableColumn(int modelIndex, int width,
    TableCellRenderer renderer, TableCellEditor editor)
```

Creates a `TableColumn` of the given width for the given index in the model. This `TableColumn` will use the given renderer and editor. The value passed for `editor` may be `null`.

Methods

```
public synchronized void addPropertyChangeListener(
    java.beans.PropertyChangeListener listener)
```

Adds `listener` to the list of objects to be notified when bound properties change.

```
public void disableResizedPosting()
```

Disables the posting of resize events while the table is resizing.

```
public void enableResizedPosting()
```

Enables the posting of resize events while the table is resizing.

```
public TableCellEditor getCellEditor()
```

Gets the editor responsible for displaying cells in this column (if this is `null`, `JTable` will use its default editor).

```
public TableCellRenderer getCellRenderer()
```

Gets the renderer responsible for displaying cells in this column (if this is `null`, `JTable` will see use its default renderer).

```
public TableCellRenderer getHeaderRenderer()
```

Gets the renderer responsible for displaying the header cell.

```
public Object getHeaderValue()
```

Gets the value of the header cell for this column.

```
public Object getIdentifier()
```

Gets the identifier for this column. If none has been set or it is `null`, it returns the header value. (See `getHeaderValue()`.)

```
public int getMaxWidth()
```

Returns the maximum width of the column (in pixels). The default value is 2000 pixels.

```
public int getMinWidth()
```

Returns the minimum width of the column (in pixels). The default value is 15 pixels.

```
public int getModelIndex()
```

Returns the index in the model to which this `TableColumn` corresponds. The model index doesn't change as columns are moved around in the `JTable`. The simplest way to think about the relationship between a column in the model, the corresponding `TableColumn`, and how the columns are presented in the `JTable` is this:

A `TableColumn` is *associated* with a column in the model. A `JTable` contains a set of `TableColumns`, *not* columns from the model. As columns are moved around in a `JTable`, all that is being moved is the associated `TableColumn`, not the actual model column. So, the order of the `TableColumns` in the `JTable` may change, but the order of the columns in the model never changes.

```
public int getPreferredWidth()
```

Returns the preferred width of this `TableColumn`. The default preferred width is 75 pixels.

```
public boolean getResizable()
```

Returns `true` if the user may resize the column. The `setWidth()` method works even if this method returns `false`.

```
public int getWidth()
```

Returns the current width of the column.

```
public synchronized void removePropertyChangeListener(  
    java.beans.PropertyChangeListener listener)
```

Removes `listener` from the list of objects to be notified of property changes.

```
public void setCellEditor(TableCellEditor anEditor)
```

Sets the editor that will edit the values in this column.

```
public void setCellRenderer(TableCellRenderer aRenderer)
```

Sets the renderer that will display values in this column.

```
public void setHeaderRenderer(TableCellRenderer aRenderer)
```

Sets the renderer responsible for displaying the header value. See `setHeaderValue()`.

Throws: `IllegalArgumentException` if a `Renderer` is null.

```
public void setHeaderValue(Object aValue)
```

Sets the header for the column (this value is displayed by the header renderer at the top of the column).

```
public void setIdentifier(Object anIdentifier)
```

Sets the identifier for this column. This value can be used as a string name of a column; for example, `DefaultTableModel` may use column identifiers.

```
public void setMaxWidth(int newMaxWidth)
```

Sets the maximum size (in pixels) the column may take (the current width is reduced to this width if it is greater). The default maximum width is 2000 pixels.

```
public void setMinWidth(int newMinWidth)
```

Sets the minimum size (in pixels) that the column may take. The default value is 15 pixels. The current width is increased to this width if it is smaller.

```
public void setModelIndex(int index)
```

Sets the model index—the column number in the model to which this column corresponds.

```
public void setPreferredWidth(int preferredWidth)
```

Sets this column's preferred width to `preferredWidth`. If the given size is outside the maximum or minimum allowed based on the resizing value of the `JTable`, it will be set to the maximum or minimum as appropriate.

```
public void setResizable(boolean mayResize)
```

Sets whether the column may be resized.

public void setWidth(int newWidth)

Sets the width of this column to `newWidth`.

public void sizeWidthToFit()

Tells the column to resize itself to fit the size of the header cell. (It will set the column's minimum size to at most the header's current size, and set its maximum size to at least the header's current size.)

Protected Fields

protected TableCellEditor cellEditor

The editor for cells.

protected TableCellRenderer cellRenderer

The renderer that displays cells.

protected TableCellRenderer headerRenderer

The renderer that displays the `headerValue`.

protected Object headerValue

The value for the header.

protected Object identifier

The name of the column. This name is not used by Swing per se, but may be used by the application program to keep track of the column.

protected boolean isResizable

Returns `True` if the column may be resized.

protected int maxWidth

The maximum width of the column.

protected int minWidth

The minimum width of the column.

protected int modelIndex

The index in the model with which this column model is associated. This is the mapping that lets the model maintain a consistent ordering: the view can maintain a sequence of `TableColumns`, while they map to the proper column in the model.

protected transient int resizedPostingDisableCount

Used to track whether the column should receive resize events during a resize operation.

protected int width

The current width of the column.

Protected Methods

protected TableCellRenderer createDefaultHeaderRenderer()

Creates a renderer for the header cell (used if none is provided in the constructor).

Returned by `DefaultTableColumnModel.getColumn()` (p.744), `JTable.getColumn()` (p.357), `JTableHeader.getDraggedColumn()` (p.752), `JTableHeader.getResizingColumn()` (p.752), `TableColumnModel.getColumn()` (p.764).

See also `TableCellEditor` (p.758), `TableCellRenderer` (p.760).

swing.table.TableColumnModel

INTERFACE

TableColumnModel

`TableColumnModel` is the model for *all* columns in a table, as a set. `TableColumn` is the model for an individual column.

```
public interface TableColumnModel
```

Public Methods

```
public abstract void addColumn(TableColumn column)
```

Makes `column` the new last column of the model. Notifies any listeners of the change using the `columnAdded()` method of `TableColumnModelListener` (p.508).

```
public abstract void addColumnModelListener(
    TableColumnModelListener listener)
```

Adds `listener` to the list of objects to be notified when the column model changes (for example, when columns are added or removed).

```
public abstract TableColumn getColumn(int column)
```

Gets the column information for the specified `column`.

```
public abstract int getColumnCount()
```

Returns the number of columns.

```
public abstract int getColumnIndex(Object columnName)
```

Finds the index of the column for which `column.getIdentifer().equals(columnName)`. This method will throw an `IllegalArgumentException` if `columnName` is null or if no column has that identifier.

```
public abstract int getColumnIndexAtX(int xCoordinate)
```

Finds the index of the column at `xCoordinate`, or -1 if there is no such column.

```
public abstract int getColumnMargin()
```

Gets the width (in pixels) of the margin between adjacent columns. Notifies any listeners of the change using the `columnMarginChanged()` method of `swing.event.TableColumnModelListener` (p.508).

```
public abstract boolean getColumnSelectionAllowed()
```

Returns `true` if columns may be selected.

```
public abstract java.util.Enumeration getColumns()
```

Returns the list of columns, in order.

```
public abstract int getSelectedColumnCount()
```

Returns the number of selected columns.

```
public abstract int[] getSelectedColumns()
```

Returns a list of the selected columns. This never returns `null`, but may return an array of 0 elements.

```
public abstract ListSelectionModel getSelectionModel()
```

Returns the model that tracks column selection.

```
public abstract int getTotalColumnWidth()
```

Returns the sum of all column widths (including margins).

```
public abstract void moveColumn(int fromIndex, int toIndex)
```

Moves the column at `fromIndex` to be located at `toIndex`. Notifies any listeners of the change using the `columnMoved()` method of `TableColumnModelListener` (p.508).

```
public abstract void removeColumn(TableColumn column)
```

Removes the specified `column`. Shifts other columns to adjust for this. Notifies any listeners of the change using the `columnRemoved()` method of `TableColumnModelListener` (p.508).

```
public abstract void removeColumnModelListener(
    TableColumnModelListener listener)
    Removes listener from the list of objects to be notified of column changes.
public abstract void setColumnMargin(int width)
    Sets the margin between columns to width (in pixels).
public abstract void setColumnSelectionAllowed(boolean maySelect)
    Sets whether columns may be selected.
public abstract void setSelectionModel(ListSelectionModel selectionModel)
    Sets the selection model for the columns.
```

Returned by `JTable.createDefaultColumnModel()` (p.357), `JTable.getColumnModel()` (p.357), `JTableHeader.createDefaultColumnModel()` (p.752), `JTableHeader.getColumnModel()` (p.752).

Implemented by `DefaultTableColumnModel` (p.744).

See also `ListSelectionModel` (p.421), `TableColumnModelListener` (p.508).

swing.table.TableModel

INTERFACE

TableModel

This is the fundamental interface for tables. This is the model used by `JTable`.

The easiest way to provide this interface is to override `AbstractTableModel` or use a `DefaultTableModel`. The preferred choice is to subclass `AbstractTableModel`.

```
public interface TableModel
```

Methods

```
public abstract void addTableModelListener(TableModelListener listener)
    Adds listener to the list of objects to be notified when the table changes.
public abstract java.lang.Class getColumnClass(int column)
    Gets the class for column. By providing this information, you help cause the most appropriate renderer to be used. (For example, an entry of type Boolean could be displayed as a checkbox, rather than the string "true" or "false.") You should return the class that is common to all entries in the column. For example, if you have both Integer and Boolean items, you would need to return Object.class (their only common superclass).
public abstract int getColumnCount()
    Returns the number of columns.
public abstract String getColumnName(int column)
    Returns the name of the specified column. Two columns may have the same name; this usually has no impact because JTable usually works with column indexes rather than column names. (When a JTable is created, it queries the model for the number of columns and their names; see JTable.createDefaultColumnsFromModel() and JTable.getAutoCreateColumnsFromModel().)
public abstract int getRowCount()
    Returns the number of rows. This routine is called by JTable very frequently.
public abstract Object getValueAt(int row, int column)
    Returns the value at the specified row and column.
```

public abstract boolean isCellEditable(int row, int column)

Returns `true` if the cell may be edited.

public abstract void removeTableModelListener(TableModelListener listener)

Removes `listener` from the list of objects to be notified when the table changes.

public abstract void setValueAt(Object object, int row, int column)

Stores `object` at the specified cell in the table. (The value is stored only if `isCellEditable(row, column)` returns `true`.)

Returned by `JTable.createDefaultDataModel()` (p.357), `JTable.getModel()` (p.357).

Implemented by `AbstractTableModel` (p.735).

See also `DefaultTableModel` (p.747), `JTable` (p.357), `TableModelListener` (p.510).

