```
/**
 * In a chain of data manipulators some behaviour is common. TableMap
 * provides most of this behaviour and can be subclassed by filters
 * that only need to override a handful of specific methods. TableMap
 * implements TableModel by routing all requests to its model, and
 * TableModelListener by routing all events to its listeners. Inserting
 * a TableMap which has not been subclassed into a chain of table filters
 * should have no effect.
 *
 * @version 1.4 12/17/97
 * @author Philip Milne */

import javax.swing.table.AbstractTableModel;
import javax.swing.table.TableModel;
import javax.swing.event.TableModelListener;
import javax.swing.event.TableModelEvent;

public class TableMap extends AbstractTableModel
                      implements TableModelListener {
    protected TableModel model;

    public TableModel getModel() {
        return model;
    }

    public void setModel(TableModel model) {
        this.model = model;
        model.addTableModelListener(this);
    }

    // By default, implement TableModel by forwarding all messages
    // to the model.

    public Object getValueAt(int aRow, int aColumn) {
        return model.getValueAt(aRow, aColumn);
    }

    public void setValueAt(Object aValue, int aRow, int aColumn) {
        model.setValueAt(aValue, aRow, aColumn);
    }

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

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

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

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

    public boolean isCellEditable(int row, int column) {
         return model.isCellEditable(row, column);
    }
//
// Implementation of the TableModelListener interface,
//
    // By default forward all events to all the listeners.
    public void tableChanged(TableModelEvent e) {
        fireTableChanged(e);
    }
}
```

```
/**
 * A sorter for TableModels. The sorter has a model (conforming to TableModel)
 * and itself implements TableModel. TableSorter does not store or copy
 * the data in the TableModel, instead it maintains an array of
 * integers which it keeps the same size as the number of rows in its
 * model. When the model changes it notifies the sorter that something
 * has changed eg. "rowsAdded" so that its internal array of integers
 * can be reallocated. As requests are made of the sorter (like
 * getValueAt(row, col) it redirects them to its model via the mapping
 * array. That way the TableSorter appears to hold another copy of the table
 * with the rows in a different order. The sorting algorthm used is stable
 * which means that it does not move around rows when its comparison
 * function returns 0 to denote that they are equivalent.
 *
 * @version 1.5 12/17/97
 * @author Philip Milne
 */

import java.util.Date;
import java.util.Vector;

import javax.swing.table.TableModel;
import javax.swing.event.TableModelEvent;

//Imports for picking up mouse events from the JTable.
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.InputEvent;
import javax.swing.JTable;
import javax.swing.table.JTableHeader;
import javax.swing.table.TableColumnModel;

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

    public TableSorter() {
        indexes = new int[0]; // for consistency
    }

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

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

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

        // Check for nulls.

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

        // If both values are null, return 0.
        if (o1 == null && o2 == null) {
            return 0;
        } else if (o1 == null) { // Define null less than everything.
            return -1;
        } else if (o2 == null) {
            return 1;
        }
```



```
        /*
         * We copy all returned values from the getValue call in case
         * an optimised model is reusing one object to return many
         * values.  The Number subclasses in the JDK are immutable and
         * so will not be used in this way but other subclasses of
         * Number might want to do this to save space and avoid
         * unnecessary heap allocation.
         */

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

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

            if (n1 < n2) {
                return -1;
            } else if (n1 > n2) {
                return 1;
            } else {
                return 0;
            }
        } else if (type == String.class) {
            String s1 = (String)data.getValueAt(row1, column);
            String s2    = (String)data.getValueAt(row2, column);
            int result = s1.compareTo(s2);

            if (result < 0) {
                return -1;
            } else if (result > 0) {
                return 1;
            } else {
                return 0;
            }
        } else if (type == Boolean.class) {
            Boolean bool1 = (Boolean)data.getValueAt(row1, column);
            boolean b1 = bool1.booleanValue();
            Boolean bool2 = (Boolean)data.getValueAt(row2, column);
            boolean b2 = bool2.booleanValue();

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

            if (result < 0) {
                return -1;
```

```
            } else if (result > 0) {
                return 1;
            } else {
                return 0;
            }
        }
    }

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

    public void reallocateIndexes() {
        int rowCount = model.getRowCount();

        // Set up a new array of indexes with the right number of elements
        // for the new data model.
        indexes = new int[rowCount];

        // Initialise with the identity mapping.
        for (int row = 0; row < rowCount; row++) {
            indexes[row] = row;
        }
    }

    public void tableChanged(TableModelEvent e) {
        //System.out.println("Sorter: tableChanged");
        reallocateIndexes();

        super.tableChanged(e);
    }

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

    public void sort(Object sender) {
        checkModel();

        compares = 0;
        // n2sort();
        // qsort(0, indexes.length-1);
        shuttlesort((int[])indexes.clone(), indexes, 0, indexes.length);
        //System.out.println("Compares: "+compares);
    }

    public void n2sort() {
        for (int i = 0; i < getRowCount(); i++) {
            for (int j = i+1; j < getRowCount(); j++) {
                if (compare(indexes[i], indexes[j]) == -1) {
                    swap(i, j);
                }
            }
        }
    }

    // This is a home-grown implementation which we have not had time
    // to research - it may perform poorly in some circumstances. It
    // requires twice the space of an in-place algorithm and makes
```



```
    // NlogN assigments shuttling the values between the two
    // arrays. The number of compares appears to vary between N-1 and
    // NlogN depending on the initial order but the main reason for
    // using it here is that, unlike qsort, it is stable.
    public void shuttlesort(int from[], int to[], int low, int high) {
        if (high - low < 2) {
            return;
        }
        int middle = (low + high)/2;
        shuttlesort(to, from, low, middle);
        shuttlesort(to, from, middle, high);

        int p = low;
        int q = middle;

        /* This is an optional short-cut; at each recursive call,
        check to see if the elements in this subset are already
        ordered.  If so, no further comparisons are needed; the
        sub-array can just be copied.  The array must be copied rather
        than assigned otherwise sister calls in the recursion might
        get out of sinc.  When the number of elements is three they
        are partitioned so that the first set, [low, mid), has one
        element and and the second, [mid, high), has two. We skip the
        optimisation when the number of elements is three or less as
        the first compare in the normal merge will produce the same
        sequence of steps. This optimisation seems to be worthwhile
        for partially ordered lists but some analysis is needed to
        find out how the performance drops to Nlog(N) as the initial
        order diminishes - it may drop very quickly.  */

        if (high - low >= 4 && compare(from[middle-1], from[middle]) <= 0) {
            for (int i = low; i < high; i++) {
                to[i] = from[i];
            }
            return;
        }

        // A normal merge.

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

    public void swap(int i, int j) {
        int tmp = indexes[i];
        indexes[i] = indexes[j];
        indexes[j] = tmp;
    }

    // The mapping only affects the contents of the data rows.
    // Pass all requests to these rows through the mapping array: "indexes".

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

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

    public void sortByColumn(int column) {
```

```
        sortByColumn(column, true);
    }

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

    // There is no-where else to put this.
    // Add a mouse listener to the Table to trigger a table sort
    // when a column heading is clicked in the JTable.
    public void addMouseListenerToHeaderInTable(JTable table) {
        final TableSorter sorter = this;
        final JTable tableView = table;
        tableView.setColumnSelectionAllowed(false);
        MouseAdapter listMouseListener = new MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
                TableColumnModel columnModel = tableView.getColumnModel();
                int viewColumn = columnModel.getColumnIndexAtX(e.getX());
                int column = tableView.convertColumnIndexToModel(viewColumn);
                if (e.getClickCount() == 1 && column != -1) {
                    //System.out.println("Sorting ...");
                    int shiftPressed = e.getModifiers()&InputEvent.SHIFT_MASK;
                    boolean ascending = (shiftPressed == 0);
                    sorter.sortByColumn(column, ascending);
                }
            }
        };
        JTableHeader th = tableView.getTableHeader();
        th.addMouseListener(listMouseListener);
    }
}
```

```
/*
 * TableSorterDemo.java is a 1.4 application that requires these files:
 *   TableSorter.java
 *   TableMap.java
 */

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.table.AbstractTableModel;
import java.awt.Dimension;
import java.awt.GridLayout;

/**
 * TableSorterDemo is like TableDemo, except that it
 * inserts a custom model -- a sorter -- between the table
 * and its data model.  It also has column tool tips.
 */
public class TableSorterDemo extends JPanel {
    private boolean DEBUG = false;

    public TableSorterDemo() {
        super(new GridLayout(1,0));

        TableSorter sorter = new TableSorter(new MyTableModel()); //ADDED THIS
        //JTable table = new JTable(new MyTableModel());         //OLD
        JTable table = new JTable(sorter);                       //NEW
        sorter.addMouseListenerToHeaderInTable(table); //ADDED THIS
        table.setPreferredScrollableViewportSize(new Dimension(500, 70));

        //Set up tool tips for column headers.
        table.getTableHeader().setToolTipText(
                "Click to sort; Shift-Click to sort in reverse order");

        //Create the scroll pane and add the table to it.
        JScrollPane scrollPane = new JScrollPane(table);

        //Add the scroll pane to this panel.
        add(scrollPane);
    }

    class MyTableModel extends AbstractTableModel {
        private String[] columnNames = {"First Name",
                                        "Last Name",
                                        "Sport",
                                        "# of Years",
                                        "Vegetarian"};
        private Object[][] data = {
            {"Mary", "Campione",
             "Snowboarding", new Integer(5), new Boolean(false)},
            {"Alison", "Huml",
             "Rowing", new Integer(3), new Boolean(true)},
            {"Kathy", "Walrath",
             "Knitting", new Integer(2), new Boolean(false)},
            {"Sharon", "Zakhour",
             "Speed reading", new Integer(20), new Boolean(true)},
            {"Philip", "Milne",
             "Pool", new Integer(10), new Boolean(false)}
        };

        public int getColumnCount() {
            return columnNames.length;
        }

        public int getRowCount() {
            return data.length;
        }
```



```
        public String getColumnName(int col) {
            return columnNames[col];
        }

        public Object getValueAt(int row, int col) {
            return data[row][col];
        }

        /*
         * JTable uses this method to determine the default renderer/
         * editor for each cell.  If we didn't implement this method,
         * then the last column would contain text ("true"/"false"),
         * rather than a check box.
         */
        public Class getColumnClass(int c) {
            return getValueAt(0, c).getClass();
        }

        /*
         * Don't need to implement this method unless your table's
         * editable.
         */
        public boolean isCellEditable(int row, int col) {
            //Note that the data/cell address is constant,
            //no matter where the cell appears onscreen.
            if (col < 2) {
                return false;
            } else {
                return true;
            }
        }

        /*
         * Don't need to implement this method unless your table's
         * data can change.
         */
        public void setValueAt(Object value, int row, int col) {
            if (DEBUG) {
                System.out.println("Setting value at " + row + "," + col
                                   + " to " + value
                                   + " (an instance of "
                                   + value.getClass() + ")");
            }

            data[row][col] = value;
            fireTableCellUpdated(row, col);

            if (DEBUG) {
                System.out.println("New value of data:");
                printDebugData();
            }
        }

        private void printDebugData() {
            int numRows = getRowCount();
            int numCols = getColumnCount();

            for (int i=0; i < numRows; i++) {
                System.out.print("  row " + i + ":");
                for (int j=0; j < numCols; j++) {
                    System.out.print(" " + data[i][j]);
                }
                System.out.println();
            }
            System.out.println("--------------------------");
        }
    }

    /**
```

```
     * Create the GUI and show it.  For thread safety,
     * this method should be invoked from the
     * event-dispatching thread.
     */
    private static void createAndShowGUI() {
        //Make sure we have nice window decorations.
        JFrame.setDefaultLookAndFeelDecorated(true);

        //Create and set up the window.
        JFrame frame = new JFrame("TableSorterDemo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //Create and set up the content pane.
        TableSorterDemo newContentPane = new TableSorterDemo();
        newContentPane.setOpaque(true); //content panes must be opaque
        frame.setContentPane(newContentPane);

        //Display the window.
        frame.pack();
        frame.setVisible(true);
    }

    public static void main(String[] args) {
        //Schedule a job for the event-dispatching thread:
        //creating and showing this application's GUI.
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                createAndShowGUI();
            }
        });
    }
}
```