JTable with Frozen Columns Scrolled Separately

Although it also has its own tricks that should be known, yet there are many examples out there providing scrollable JTable with frozen columns on the left. This article will briefly explain that too, but the aim is on showing how it is possible to add horizontal scrolling capabilities to those frozen columns.

To start with, just examine the panel providing scrolling support - JScrollPane. It contains the main viewport, which in turn contains the view - this latter is the graphical information we are going to display, but it is suspected that the screen is just not large enough to show it full.Therefore, there is a small sight (the viewport), and the large graphical view is moved under this sight so it is possible to see only a part of it. There are two scroll bars, a horizontal and a vertical one, which controls the movement of the view under the sight. It is possible to add a column header which appears right above the viewport, and a row header, which appears just left of the viewport. There are four corners which also could be populated by graphical components. The two headers are also viewports, because their content should scroll together with the main view. Synchronizing the positioning of these viewports and adjusting the scrollbars is the responsibility of the JScrollPane.

When a JTable is added to the JScrollPane, the table part appears in the main viewport, and the table header appears as the column header. That's done automatically, without any extra efforts. In fact, JTable's are almost always are added to scroll panes. If the presented table should have frozen columns, then those frozen columns should be added to the row header part of the scroll pane. Sounds simple as it is, yet there are some tricks that should be known. The process briefly is as follows.

- Create another JTable (the header table) which uses the same TableModel.
- It is likely to be wished that the auto resize mode of the main table be switched off (AUTO_RESIZE_OFF), but for the header table it might remain on. Further, the default size of the columns is usually too big, so it is likely that the preferred scrollable viewport size should be adjusted for the header table.
- Set the selection model of the header table to the selection model of the main table (they must share the selection model, then highlighting the rows will appear nicely).
- Retrieve the TableColumnModel of the main table, and remove those columns that will be frozen (the main table shouldn't present these columns).
- Retrieve the TableColumnModel of the header table, and remove all but the frozen columns.
- Optional: The default renderers of the main table might be set for the header table (and their header's default renderers migth be shared too).
- Optional: One might want to add a border to the header table (top, left, bottom is empty, but right is a few pixels thick), in order to more clearly separate the frozen part from the scrollable area.
- The header table should be added as the row header view to the JScrollPane.
- The header table's JTableHeader should be added as the upper left corner to the JScrollPane.
- Optional: Navigating with the keyboard opens a can of worms, so it might be solved. This is out of scope now, but consider that pressing the left arrow key when in the very first column of the main table should result of selecting the last column of the header table's same row (and pressing there the right arrow should bring back the selection to the main table). Further, when moving vertically with the up/down keys in the main table, the header will scroll - but moving vertically in the header table won't adjust the main table's position. This is a known bug (several years old now), and it is unlikely that it will be solved ever.

Now it's done, it works as expected. But suppose it is required to have several columns frozen, but it is not wished to waste too much space for them. In this case, one might want to implement a scrollable row header. Simply putting the header table into a scroll pane, then adding the scroll pane to a viewport is apparently a bad idea for two reasons. First, the scroll bar shouldn't appear in the row header, it should be levelled with the main table's horizontal scroll bar. Second, it's very difficult to implement because of the way how the preferred sizes are calculated. However, one might try to add the header table to a JScrollPane, then retrieve the column header of that JScrollPane and add it to the upper left corner of the main JScrollPane; retrieve the viewport and add it as row header; retrieve the horizontal scroll bar and add it to the lower left corner. Then that temporary scroll pane is no longer needed, it won't ever use any resources - yet the scroll bar and the two viewports are already bound, and horizontal scrolling works perfectly. (Apparently, the preferred scrollable viewport size should be set for the header table, and the auto resize mode should be switched off too.)

```
// headerTable is the one containing the frozen columns

JScrollPane tempSP = new JScrollPane(headerTable);


// scrollPane is the main scroll pane.
scrollPane.setRowHeader(tempSP.getViewport());
scrollPane.setCorner(JScrollPane.UPPER_LEFT_CORNER, tempSP.getColumnHeader());
scrollPane.setCorner(JScrollPane.LOWER_LEFT_CORNER, tempSP.getHorizontalScrollBar());
```

But the users want to use the scroll bars of the main table too. Whenever the position of the main view changes, the row header and the column header are repositioned too. The designers of the JScrollPane never thought that one day the row header will scroll horizontally, therefore when the row header is repositioned, the x coordinate is set to 0. The user scrolls the row header horizontally to see the columns he is interested in, then moves the main table, and the row header jumps back to its leftmost position. He will be frustrated soon. With listeners, this problem is almost impossible to be solved. One would attempt to introduce some difficult and complex state machine behaviour, remembering the x position of the header set last, and trying to somehow reset the x position. But of course, when the scroll pane resets the header's x position to zero, the corresponding horizontal scroll bar is also resetted. It is practically impossible, better don't even try.

There's another solution however. If it would be possible to achieve that the position setting call of these header viewports (the row header and its table header) ignores the horizontal setting and preserves the current position, then the problem would have been solved. The position of the views are set by calling the setViewPosition(Point) method of the JViewport. Therefore, it is possible to write a new class (extends JViewport), which rewrites this method by ignoring the x coordinate. Apparently, there should be another method that allows horizontal scrolling. The temporary JScrollPane must use these new classes instead of the JViewports, so creating the scroll pane is a bit more difficult now. Suppose the new class is named XViewport:

```
JScrollPane tempSP = new JScrollPane();
XViewport headerVP = new XViewport();
headerVP.setView(headerTable);
tempSP.setViewport(headerVP);

XViewport colheadVP = new XViewport();
colheadVP.setView(headerTable.getTableHeader());
tempSP.setColumnHeader(colheadVP);

// The temporary scroll pane is ready, populate scrollPane as above
```

But this temporary scroll pane is not very useful any more! There's no way to tell what other class called the setViewPosition method, therefore the binding the temporary scroll pane would create is useless, the x coordinate won't be set ever (the temporary scroll pane doesn't know that our new method allowing horizontal scroll should be called instead). So it's enough to have the XViewport instances, and create the JScrollBar by hand, and populate the main scroll pane with these objects directly. That scroll bar should be listened too, and whenever the AdjustmentEvent occurs, the position of the views should be adjusted by using the new method allowing horizontal position change too. However, this behaviour is something that our new JViewport descendant class might (and should) support. Further, JScrollPane expects fixed size unscrollable components in the corners, therefore it never attempts to set its "view position" (it cannot too because the corners are populated by Component instances, not JViewport ones). So the table header might remain in a JViewport because it's position is not to be changed by other mechanisms, and our new XViewport might take care of its positioning too. Considering all these, that's the way this new class should be used:

```
XViewport headerVP = new XViewport();
headerVP.setView(headerTable);
headerVP.setHeader(headerTable.getTableHeader());
headerVP.setScrollBar(new JScrollBar(JScrollBar.HORIZONTAL));

scrollPane.setRowHeader(headerVP);
scrollPane.setCorner(JScrollPane.UPPER_LEFT_CORNER, headerVP.getHeader());
scrollPane.setCorner(JScrollPane.LOWER_LEFT_CORNER, headerVP.getScrollBar());
```

Simple and straightforward, isn't it? Now the requirement is clear, we have the order, so let's deliver. As soon as this new class is implemented, the problem is solved. The declaration part and the accessor methods (and the mutator of the header) are simple. Please pardon me that I don't add the javadoc.

```
public class XViewport extends JViewport implements AdjustmentListener {
    private JScrollBar scrollBar;
    private JViewport  header;

    public XViewport() {
        super();
    }

    public JScrollBar getScrollBar() {
        return scrollBar;
    }

    public JViewport getHeader() {
        return header;
    }

    public void setHeader(JViewport hdr) {
        header = hdr;
    }
```

Yeah, I know that the constructor is rather unnecessary, but it's easier to debug the code this way (I might check when the class is instantiated), so it's just my behaviour to add the empty constructor. You don't have to like it, just bow to it for a while please. More important are the positioning methods:

```
    @Override
    public void setViewPosition(final Point p) {
        int x = getViewPosition().x;
        super.setViewPosition(new Point(x, p.y));
    }

    public void setViewPositionX(final int x) {
        int y = getViewPosition().y;
        super.setViewPosition(new Point(x, y);
    }
```

The original (now overridden) method will preserve the current value of x, while the new method preserves the current value of y. This latter is not absolutely necessary; true that we are going to use it only for horizontal positioning, but one day we might need vertical scrolling in the column header. However, for simplicity reasons, let's just have it this way. Now we need the mutator method for the scroll bar:

```
    public void setScrollBar(final JScrollBar sb) {
        if (scrollBar != null) {
            scrollBar.removeAdjustmentListener(this);
        }
        scrollBar = sb;
        adjustScrollBarRange();
        Point op = getViewPosition();
        scrollBar.setValue(op.x);
        scrollBar.addAdjustmentListener(this);
    }

    // call only if the scroll bar exists
    protected void adjustScrollBarRange() {
        Rectangle bounds = getBounds();
        Dimension dim    = getViewSize();
        if (bounds.width < dim.width) {
            scrollBar.setMinimum(0);
            scrollBar.setMaximum(dim.width - bounds.width + sb.getVisibleAmount());
        }
    }
```

The previous scroll bar (if existed) shouldn't be listened any more. Next is to prepare the scroll bar for use (adjustScrollBarRange method); the possible values should be between zero and the width of the view minus the extent size. This way it is very easy to position the view whenever the scroll bar's value is set. It is possible to synchronize the new scroll bar with the view now, and finally we might start to listen the scroll bar adjustment events. However, it is possible that the viewport is resized; ultimately it results a setBound call (well, this is not carved into stone, but that's what indeed happens), and then the scroll bar range should be recalculated. For the industry-strength application of this class, other resizing methods should be also overridden. This method is again very simple:

```
    @Override
    public void setBounds(int x, int y, int width, int height) {
        super.setBounds(x, y, width, height);
        adjustScrollBarRange();
    }
```

All that remains is the adjustment listener:

```
    @Override
    public void adjustmentValueChanged(final AdjustmentEvent e) {
        int val = e.getValue();
        setViewPositionX(val);
        if (header != null) {
            Point op = header.getViewPosition();
            header.setViewPosition(new Point(val, op.y));
        }
    }
```

Now it's clear why the range of the scroll bar was set. This way, when the scroll bar is adjusted, the new value is the x coordinate of the view position. If the header exists, it's x position is also changed, while its original y position is preserved (I'm not going to do a similar mistake of supposing the y coordinate is always 0).

Now it works, although some homework still remains. If the user navigates horizontally in the header table by using the keyboard keys, the table won't scroll as the cell selection changes. That's because upon the table cell selection changes, when the JTable calls the scrollRectToVisible method, the now overridden setViewPosition method is called. Therefore, the new XViewport should override the scrollRectToVisible method and should call the proper setViewPosition method. That's not that hard, but it's out of scope now.