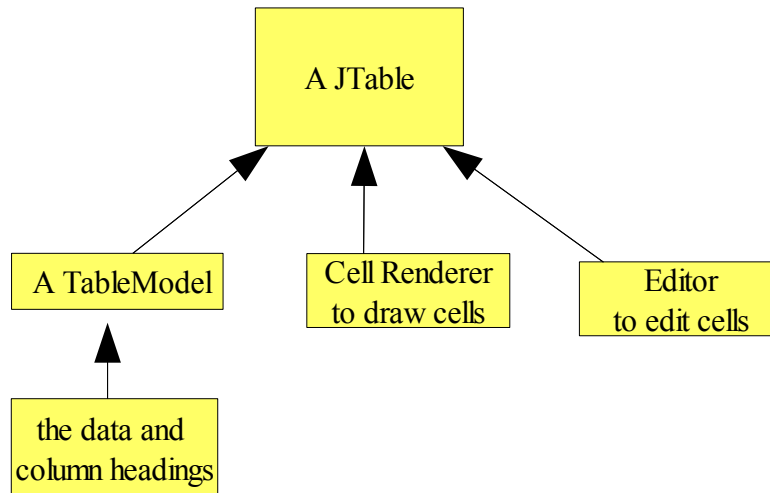


Using JTable

Many of the things which a JTable can do are handled by instances of other objects. There are default values of these, or we can sub-class the defaults to get closer control over what happens. An outline is like this:



This shows that the data is not really 'in' the table. The underlying data is handled by a TableModel (which really means a model of the data suitable for the table to deal with).

The following shows a simple start - a table to show a 6 X 6 set of Integers:

```
public class MainFrame extends JFrame {  
  
    Integer[][] data = new Integer[6][6];  
    JTable table;  
    IndexTableModel model;  
  
    MainFrame()  
    {  
        setLayout(null);  
        setBounds(50,50,320,400);  
        setVisible(true);  
        for (int i=0; i<6; i++)  
            for (int j=0; j<6; j++)  
                data[i][j]=i*j;  
        String[] headers={"A", "B", "C", "D", "E", "F" };  
        // make table model with this data..  
        model = new IndexTableModel(headers, data);  
        table = new JTable(model);  
        JScrollPane scrollPane = new JScrollPane(table);  
        table.setFillViewportHeight(true);  
        add(scrollPane);  
        scrollPane.setBounds(5,5,300,300);  
    }  
}
```

This makes the data and the headers, makes a data model from them, then makes the table from that. The table is placed in a JScrollPane, and that is added to the JFrame. We also need the TableModel. The idea of this is that it hands the data to the table in the appropriate way. We can code it by implementing the methods of AbstractTableModel:

```

class IndexTableModel extends AbstractTableModel {

    String[] headers;
    Integer[][] data;

    IndexTableModel(String[] headers, Integer[][] data)
    {
        super();
        this.headers=headers;
        this.data=data;
    }

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

    public int getRowCount() {
        return data[0].length;
    }

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

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

    public Class<?> getColumnClass(int c) {
        return data[0][c].getClass();
    }

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

}

```

which gives this:

The data in a table must be a reference type - you cannot have a table of primitives (unless you wrap them as in this example).

If you do not put the table in a JScrollPane, you have to draw the headers yourself.

A	B	C	D	E	F
0	0	0	0	0	0
0	1	2	3	4	5
0	2	4	6	8	10
0	3	6	9	12	15
0	4	8	12	16	20
0	5	10	15	20	25

Data Changes

Suppose the data underlying the table changes. Will you see that change? No. The TableModel does not continuously monitor the data. You have to tell it (the model) that the data has changed, and then it will signal the table to reflect those changes. For example we might change the data as a result of a button click, and the actionPerformed would be :

```

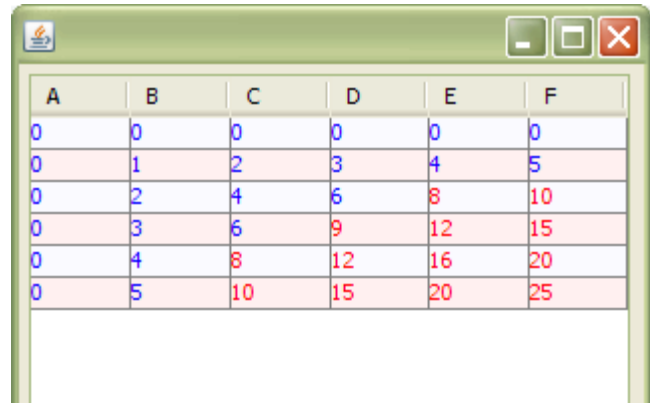
data[1][1]=new Integer(1234);
model.fireTableCellUpdated (1, 1) ;

```

There are other versions of the fireXXX method in the AbstractTableModel API.

Drawing the cells - the cell renderer

The table turns to the cell renderer to tell it how to draw cells. The above is using the default renderer, which is pretty dull. The following example generates the appearance on the right, coded so the rows alternate in background color, and with a red foreground color if the value is greater than 6.



A	B	C	D	E	F
0	0	0	0	0	0
0	1	2	3	4	5
0	2	4	6	8	10
0	3	6	9	12	15
0	4	8	12	16	20
0	5	10	15	20	25

Because we are just displaying integers, the cell renderer can be a sub-classed JLabel coded appropriately, and implementing the TableCellRenderer interface:

```
class MyRenderer extends JLabel implements TableCellRenderer {
    boolean isBordered;

    public MyRenderer(boolean isBordered) {
        this.isBordered = isBordered;
        setOpaque(true);
    }

    public Component getTableCellRendererComponent(JTable table, Object value, boolean isSelected, boolean hasFocus, int row, int column) {
        setText("" + value);
        if (row%2==0)
            setBackground(new Color(250,250,255));
        else setBackground(new Color(255,240,240));
        if ((Integer)value>6)
            setForeground(Color.red);
        else setForeground(Color.blue);
        setToolTipText("" + value);
        return this;
    }
}
```

So this is a JLabel which implements the TableCellRenderer interface, which has the single getTableCellRendererComponent method. This eventually returns this - that is, a subclassed JLabel. The Object parameter to this is what is in the cell which is being rendered.

We tell the table to use this renderer by

```
table.setDefaultRenderer(Integer.class, new MyRenderer(true));
```

so it will use this for all cells of type Integer, which is all of them.

Selection of cells

There are two aspects to this -

1. Whether rows, columns or cells are selected
2. Whether you can have multiple or just a single selection.

We can control the appearance of selected cells by using the isSelected parameter of the getTableCellRendererComponent method of the renderer - maybe something like..

```
if (isSelected)
    setForeground(Color.red);
else
    setForeground(Color.blue);
```

If you do not have a cell renderer, you can say something like

```
table.setSelectionForeground(Color.red);
```

but this is relatively restricted - with the renderer you can code anything to mark selected cells.

With this, clicking on cell 0,0 selects a row like this:

This is because the table can be set to either select a row, a column, or a single cell. The 3 methods are:

```
table.setRowSelectionAllowed(true/false);  
table.setColumnSelectionAllowed(true/false);  
table.setCellSelectionEnabled(true/false);
```

The default as above is that row selection is true. If it is false, and column selection is true, a single click selects that column. And if both are true, or equivalently cell selection is true, a click selects that cell only.

The second aspect relates to multiple selections. This is controlled by setSelectionMode, like:

```
table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

Single selection means you can only select a single thing (row, column or cell) at a time.

SINGLE_INTERVAL_SELECTION enables several contiguous things to be selected, and

MULTIPLE_INTERVAL_SELECTION allows for several things which do not have to be contiguous.

In fact we can get further control over this by setting up our own ListSelectionModel - the methods described here operate on the default one.

Editing - using the default editor

Whether a cell can be edited is controlled by the isCellEditable method of the table model:

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

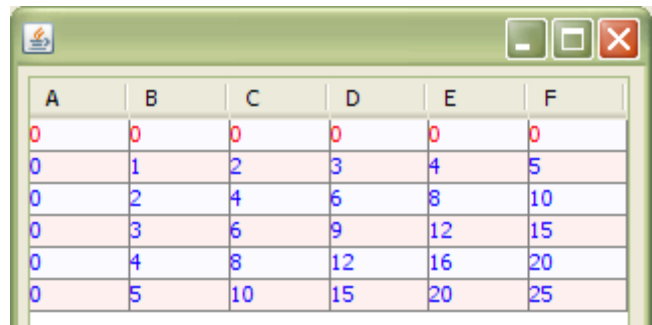
This makes them all editable, but an if using the parameters can obviously restrict editing to certain cells.

In addition, the setValueAt method of the table model must be implemented - otherwise the edit happens but is ignored. For example:

```
public void setValueAt(Object value, int row, int col) {  
    data[row][col] = (Integer) value;  
    fireTableCellUpdated(row, col);  
}
```

This version takes the edited value, places it in the underlying data structure, and calls the model's fireTableCellUpdated method, which in turns updates the table's appearance.

The default editor looks at what data type is in the cell, and constructs an appropriate component to edit it - for example if the type is Boolean, it produces a checkbox.



	A	B	C	D	E	F
0	0	0	0	0	0	0
1	0	1	2	3	4	5
2	0	2	4	6	8	10
3	0	3	6	9	12	15
4	0	4	8	12	16	20
5	0	5	10	15	20	25

Using a Custom Cell Editor

We can create our own cell editor and use it either for given columns or for a given data type of the cell contents.

For example, suppose we want the edit for the first column to be a JTextField with a yellow background. We can do this with the defaultCellEditor constructed with a yellow text field:

```
TableColumn firstColumn = table.getColumnModel().getColumn(0);
JTextField edit = new JTextField();
edit.setBackground(Color.yellow);
firstColumn.setCellEditor(new DefaultCellEditor(edit));
```

However a JTextField holds a String, and we need to change the setValueAt in the table model:

```
public void setValueAt(Object value, int row, int col) {
    if (col==0)
        data[row][col] = new Integer((String)value);
    else
        data[row][col] = (Integer) value;
    fireTableCellUpdated(row, col);
}
```

Input data validation using our own editor

The DefaultCellEditor validates input in terms of type, and does not return if the input string is invalid. For further validation, such as a range check, we can subclass it. Suppose we will only accept integer values less than 100. We first set the table (just the first column) to use it:

```
TableColumn firstColumn = table.getColumnModel().getColumn(0);
MyEditor edit = new MyEditor();
firstColumn.setCellEditor(edit);
```

Then we must define MyEditor:

```
class MyEditor extends DefaultCellEditor
{
    MyEditor()
    {
        super(new JTextField()); //make the superclass based on a JTextField
        getComponent().setBackground(Color.yellow); //make it yellow
    }

    // This is called when editing stops. We check everything is OK.
    // If it is, we return the base class method. If it is not, we
    // return false, and editing continues
    public boolean stopCellEditing() {
        String value = ((JTextField)getComponent()).getText();
        Integer i;
        // is it a valid integer?
        try
        {
            i=Integer.parseInt(value);
        }
    }
}
```

```

catch (NumberFormatException e)
{
    return false; // the editor does not return
}
// is it in range?
if (i > 100) return false;
return super.stopCellEditing();
}
}

```

Responding to cell clicks

Cell selections are dealt with by the table's `ListSelectionModel`. We can respond to a cell click by creating an object of a class which implements the `ListSelectionListener` interface, and sending the events to it:

```

ListSelectionModel listSelectionModel = table.getSelectionModel();
listSelectionModel.addListSelectionListener(new MyListSelectionHandler(table));

```

and the class is defined like this:

```

class MyListSelectionHandler implements ListSelectionListener {
    JTable table;
    MyListSelectionHandler(JTable table)
    {
        super();
        this.table = table;
    }

    public void valueChanged(ListSelectionEvent e) {
        if (e.getValueIsAdjusting())
        {
            int row = table.getSelectedRow();
            int col = table.getSelectedColumn();
            // do something..
        }
    }
}

```

The `getValueIsAdjusting()` is because you get two events per click (mouse down and mouse up), and also if the user selects the same cell again. `getValueIsAdjusting` is only true if we have a new selection.