# JIDE Grids Developer Guide

## Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PURPOSE OF THIS DOCUMENT</td>
<td>5</td>
</tr>
<tr>
<td>WHAT IS JIDE GRIDS</td>
<td>5</td>
</tr>
<tr>
<td>PACKAGES</td>
<td>5</td>
</tr>
<tr>
<td>CLASS HIERARCHY OF ALL THE JIDE TABLES</td>
<td>6</td>
</tr>
<tr>
<td>JIDETABLE</td>
<td>7</td>
</tr>
<tr>
<td>VALIDATION SUPPORT IN JIDETABLE</td>
<td>11</td>
</tr>
<tr>
<td>CELL LEVEL VALIDATION</td>
<td>11</td>
</tr>
<tr>
<td>TABLE LEVEL VALIDATION</td>
<td>12</td>
</tr>
<tr>
<td>ROW LEVEL VALIDATION</td>
<td>12</td>
</tr>
<tr>
<td>VALIDATIONRESULT</td>
<td>13</td>
</tr>
<tr>
<td>UNDO SUPPORT IN JIDETABLE</td>
<td>15</td>
</tr>
<tr>
<td>DEFAULT UNDOABLETABLEMODEL AND ABSTRACT UNDOABLETABLEMODEL</td>
<td>15</td>
</tr>
<tr>
<td>KEYSTROKES FOR UNDO AND REDO</td>
<td>15</td>
</tr>
<tr>
<td>TRANSFERHANDLER ENHANCEMENT IN JIDETABLE</td>
<td>16</td>
</tr>
<tr>
<td>CONVERTER</td>
<td>16</td>
</tr>
<tr>
<td>CELLEDITORS AND CELLRENDERERS</td>
<td>19</td>
</tr>
<tr>
<td>NAVIGABLEMODEL AND NAVIGABLETABLE</td>
<td>23</td>
</tr>
<tr>
<td>CUSTOMIZE THE NAVIGATION KEYS</td>
<td>24</td>
</tr>
<tr>
<td>CELLSPANTABLE</td>
<td>25</td>
</tr>
<tr>
<td>CELLSYLETABLE</td>
<td>30</td>
</tr>
<tr>
<td>AVAILABLE CELLSTYLES</td>
<td>32</td>
</tr>
<tr>
<td>WHERE TO DEFINE CELLSTYLE</td>
<td>32</td>
</tr>
<tr>
<td>CELLSYLETYLEMERGING</td>
<td>33</td>
</tr>
<tr>
<td>SORTABLETABLE</td>
<td>33</td>
</tr>
<tr>
<td>COMPARATOR</td>
<td>33</td>
</tr>
<tr>
<td>SORTABLE TABLE</td>
<td>34</td>
</tr>
<tr>
<td>SORTABLETABLEMODEL</td>
<td>34</td>
</tr>
<tr>
<td>As a developer, how do I use it</td>
<td>35</td>
</tr>
<tr>
<td>How to Compare</td>
<td>36</td>
</tr>
<tr>
<td>The performance of SortableTableModel</td>
<td>36</td>
</tr>
<tr>
<td>FILTER AND FILTERABLETABLEMODEL</td>
<td>39</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>AUTOFILTERTABLEHEADER</td>
<td>39</td>
</tr>
<tr>
<td>TABLEMODELWRAPPER</td>
<td>41</td>
</tr>
<tr>
<td>COMPOUNDTABLEMODELEVENT</td>
<td>43</td>
</tr>
<tr>
<td>SORTABLE/FILTERABLE LIST AND TREE</td>
<td>43</td>
</tr>
<tr>
<td>SortableListModel and SortableTreeModel</td>
<td>44</td>
</tr>
<tr>
<td>FilterableListModel and FilterableTreeModel</td>
<td>44</td>
</tr>
<tr>
<td>MORE FILTERS AND CUSTOMFILTEREDITOR</td>
<td>45</td>
</tr>
<tr>
<td>FilterFactoryManager</td>
<td>46</td>
</tr>
<tr>
<td>CustomFilterEditor</td>
<td>46</td>
</tr>
<tr>
<td>TableCustomFilterEditor</td>
<td>48</td>
</tr>
<tr>
<td>PROPERTY PANE</td>
<td>49</td>
</tr>
<tr>
<td>What does a PropertyPane look like?</td>
<td>50</td>
</tr>
<tr>
<td>As a user, how do I use it?</td>
<td>52</td>
</tr>
<tr>
<td>As a developer, how do I use it?</td>
<td>53</td>
</tr>
<tr>
<td>BEANPROPERTY AND BEANINTROSPECTOR</td>
<td>55</td>
</tr>
<tr>
<td>COLOR RELATED COMPONENTS</td>
<td>61</td>
</tr>
<tr>
<td>ColorChooserPanel</td>
<td>61</td>
</tr>
<tr>
<td>Color(Ex)ComboBox</td>
<td>62</td>
</tr>
<tr>
<td>Keyboard Support</td>
<td>62</td>
</tr>
<tr>
<td>DATE RELATED COMPONENT</td>
<td>63</td>
</tr>
<tr>
<td>DateChooserPanel</td>
<td>63</td>
</tr>
<tr>
<td>Date(Ex)ComboBox</td>
<td>68</td>
</tr>
<tr>
<td>Keyboard Support</td>
<td>68</td>
</tr>
<tr>
<td>CalendarViewer</td>
<td>69</td>
</tr>
<tr>
<td>COMBOBOX EXTENSION</td>
<td>70</td>
</tr>
<tr>
<td>Available ExComboBoxes</td>
<td>74</td>
</tr>
<tr>
<td>Supporting 3rd Party L&amp;F for ExComboBox</td>
<td>77</td>
</tr>
<tr>
<td>HOW TO CREATE YOUR OWN CELL RENDERER AND CELL EDITOR</td>
<td>79</td>
</tr>
<tr>
<td>HIERARCHICAL TABLE</td>
<td>84</td>
</tr>
<tr>
<td>HierarchicalTableModel</td>
<td>85</td>
</tr>
<tr>
<td>HierarchicalTable</td>
<td>87</td>
</tr>
<tr>
<td>Container for Child Component</td>
<td>88</td>
</tr>
<tr>
<td>Maintaining single selection</td>
<td>89</td>
</tr>
<tr>
<td>Migration from Hierarchical Table Beta version</td>
<td>89</td>
</tr>
<tr>
<td>TREETABLE</td>
<td>91</td>
</tr>
</tbody>
</table>
TABLECELLEDITORRENDERER.......................................................................................................... 117
TO BETTER INDICATE THE CELL IS EDITABLE ............................................................................... 118
TO INTERACTIVE WITH MOUSE EVENTS ON THE CELL LEVEL ..................................................... 118
Hyperlink in Cell ................................................................................................................................. 118
INTERNATIONALIZATION AND LOCALIZATION ................................................................................. 120
Purpose of This Document

Welcome to the JIDE Grids, the JTable extension product in JIDE Software’s product line.

This document is for developers who want to develop applications using JIDE Grids.

What is JIDE Grids

Believe it or not, JTable is probably one of the most commonly used Swing components in most Swing applications.

Many people complained about the design of JTable. Every design has its pros and cons - so does JTable. People have so many various kinds of requirements, so it’s really hard to design such a complex component as JTable satisfying everybody’s needs. In our opinion, it’s not the design of JTable is not good but it left out many important features that a table should have by default. Good news is JTable does leave many extension points so that you can enhance it to meet your needs. So as long as we keep improving it, it will get better and better - JIDE Grids is one step toward this goal. All components in JIDE Grids are related to JTable and are fully compatible with JTable.

In addition to JTable related features, we also included many features related to JList and JTree.

Packages

The table below lists the packages in the JIDE Grids product.

<table>
<thead>
<tr>
<th>Packages</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.jidesoft.grid</td>
<td>All the JTable related components are in this package, including PropertyTable, SortableTable, CellSpanTable, CellStyleTable, TreeTable, HierarchicalTable, SortableTableModel, and FilterableTableModel etc.</td>
</tr>
<tr>
<td>com.jidesoft.converter</td>
<td>Converters that can convert from String to Object and from Object to String.</td>
</tr>
<tr>
<td>com.jidesoft.comparator</td>
<td>Comparators</td>
</tr>
<tr>
<td>com.jidesoft.grouper</td>
<td>ObjectGroupers that can group many values into one group</td>
</tr>
</tbody>
</table>

---

1 This package is moved to jide-common.jar as other products also need converters.

2 This package is moved to jide-common.jar as other products also need comparators.
to reduce the number of distinct values.

<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.jidesoft.combobox</td>
<td>ComboBox enhancement, such as DateExComboBox and ColorExComboBox, etc. It is also possible to create your own ComboBox</td>
</tr>
<tr>
<td>com.jidesoft.list</td>
<td>SortableListModel, FilterableListModel and other classes related to JList</td>
</tr>
<tr>
<td>com.jidesoft.tree</td>
<td>SortableTreeModel, FilterableTreeModel and other classes related to JTree</td>
</tr>
<tr>
<td>com.jidesoft.filter</td>
<td>Filter related classes</td>
</tr>
<tr>
<td>com.jidesoft.lucene</td>
<td>Filter related classes using Lucene</td>
</tr>
</tbody>
</table>

**Class Hierarchy of All the JIDE Tables**

Before we discuss each table component in detail, it’d be better to give you an overview of them. See below for a class hierarchy of all the table components we have in JIDE Grids.

As you can see, JideTable extends directly JTable since JideTable provides the features that are supposed to be in JTable. Those features include nested table column header, table/row/cell validation support, and automatic changing row height.

ContextSensitiveTable provides way to configure different cell renderers and cell editors for each cell. Then it comes the NavigableTable which provides a way to define how the navigation

---

3 This package is moved to jide-common.jar as other products also need groupers.
keys work in a table. Then it comes to `CellStyleTable` and `CellSpanTable` which provide two commonly used features as the names indicated. Next is `CategorizedTable`. We will not cover this table at all in this developer guide. The reason is we are not ready to expose it as public API yet. Simply speaking, it provides access to the expand/collapse icon since both tree table and hierarchical table need it.

After `CategorizedTable`, it’s `SortableTable` which supports multiple column sorting.

Till now, it’s still one line of hierarchy. After `SortableTable`, it divides into two types of distinct tables. Each of them has its own special usage. One is `TreeTable`. `PropertyTable` is actually a special use case of `TreeTable`. `GroupTable` is also a `TreeTable`. The other kind of table is `HierarchicalTable` which is quite a unique table that is different from all other tables.

Generically speaking, you should make you table extending one of the last five tables (`SortableTable`, `HierarchicalTable`, `TreeTable`, `GroupTable` or `PropertyTable`) in the class hierarchy tree which are marked in a blue rectangle. However nothing prevents you from using any other tables as long as you know exactly what features each table provides.

**JideTable**

Let’s start with `JideTable` which is the base class of all table classes in JIDE Grids. `JideTable` is an extended version of `JTable`. The additional features we added are:

First, we enhanced `CellEditorListener` support. You can add a `JideCellEditorListener` to `JideTable`. `JideCellEditorListener` extends `CellEditorListener` which is a Swing interface and adds several methods. They are

```java
editingStarting(ChangeEvent)
editingStopping(ChangeEvent)
editingStopped(ChangeEvent)
```

With these three methods, you can now do things like preventing cell from starting edit or preventing cell from stopping edit.

The second feature is to support `Validator`. You can add a `Validator` to `JideTable`. The `validating()` method in `Validator` will be called before cell stops editing. If the validation failed, the cell editor won’t be removed. Please refer to Table Validation section for more information.

The third feature added to `JideTable` is the support of the listener for row height changes when rows have various heights. You can add a listener by calling `getRowHeights()`.

---

4 Validator and related classes are in package com.jidesoft.validation of jide-common.jar. Right now it is only used in JideTable. However this is part of an infrastructure we will build that will cover all other components which need validation.
addRowHeightChangeListener(listener) A RowHeightChangeEvent will be fired whenever setRowHeight(int row, int rowHeight) is called.

The fourth feature is the rowAutoResizes feature. By default, JTable's row height is determined explicitly. It wouldn't consider the cell content preferred height. There is a problem here if a cell renderer can support multiple lines. So we added a new attribute called rowAutoResizes. By default, it's false which is exactly the same behavior as JTable. If you set to true, the table row will resize automatically when cell renderer's preferred height changes. Please note there is a performance hit if you turned it on as it has to check all cells to find out the preferred row height. So only use it when it's really necessary.

The fifth feature is what we called nested table column. See a picture below which shows a nested table header on Windows 7.

<table>
<thead>
<tr>
<th>AH</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
</tr>
<tr>
<td>A</td>
</tr>
</tbody>
</table>

Figure 2 Nested table columns

Here is the code to create such a nested table column.

```java
// create nested table columns
TableColumnModel cm = table.getColumnModel();
TableColumnGroup first = new TableColumnGroup("Sum 1 - 4");
first.add(cm.getColumn(0));
first.add(cm.getColumn(1));
first.add(cm.getColumn(2));
first.add(cm.getColumn(3));
TableColumnGroup second = new TableColumnGroup("Sum 5 - 9");
second.add(cm.getColumn(4));
second.add(cm.getColumn(5));
second.add(cm.getColumn(6));
second.add(cm.getColumn(7));
second.add(cm.getColumn(8));
TableColumnGroup all = new TableColumnGroup("Sum 1 - 9");
all.add(first);
all.add(second);
NestedTableHeader header = (NestedTableHeader) table.getTableHeader();
header.addColumnGroup(all);
```
The sixth feature is non-contiguous cell selection. This is probably one of the oldest bugs\(^5\) that were reported to Sun Java back in 1998. In fact, it’s not that difficult to implement a selection model that supports non-contiguous cell selection. We added the feature to JideTable. To use it, you just call `table.setNonContiguousCellSelection(true)`. Once you turn non-contiguous cell selection on, you will have to use `jideTable.getTableSelectionModel()` to get a selection model that keeps the selection. The old JTable’s `getSelectionModel()` is not used anymore in this case.

![Figure 3 NonContiguousCellSelection](image)

The seventh feature is the column auto-fit feature.

On Windows, if you double click on the gap between table header, the column will be automatically resize to fit in the widest content in previous cell. JTable doesn’t have this feature. So we added this to JideTable. You can call `setColumnAutoResizable(true)` to enable it. Similar to this feature, we also added a function to `TableUtils` called `autoResizeAllColumns(JTable table)`. This method will automatically resize all columns to fit in the content in the cells. Good news about this method is that it works for all `JTable`, not just `JideTable`.

From the screenshot below, you can see the Name column is narrow while Symbol etc. columns are too wide.

![Screen shot](image)

---

After double click on the table header gap between the "Name" and the "Last", here is what you will see. Please note the width of Name column is resized automatically to fit in the contents of all cells in that column.

You can also resize all columns so that all columns are resized properly. See below.

The eighth feature is row resize and column resize. As you know, JTable supports column resize only by dragging the table column header. And it doesn't support row resize at all. It would be nice if it can support resizing at the grid line. JideTable can support it. You just need to call `setColumnResizable(true)` and `setRowResizable(true)` to enable column resizing and row resizing respectively.

Here is how row resizing works.

Here is how column resizing works.

The ninth feature is a new `AutoResizeMode` called `AUTO_RESIZE_FILL`. In this mode, you won't see a blank area on the header area as the pictures below. The left picture is on
AUTO_RESIZE_OFF mode and the right picture is on AUTO_RESIZE_FILL mode along with setFillRight(true), setFillBottom(true) and setFillsViewportWithStripe(true).

There are a few more small features in JideTable and we are also adding more when needed. All these features should probably in JTable at the first place. That’s why we add them to JideTable so that all JIDE tables can take advantage of them.

Validation Support in JideTable

JideTable has built-in validation support. The support is done on three different levels – cell, row and table. You should decide which level to implement your validation logic depending on your actual situation. Here are the details of three levels of validations.

Cell Level Validation

If the validation only depends on the cell value, such as if the number should be a positive integer, or the string must have less than 10 chars, you should implement the validation on the cell editor level. The JideCellEditor is an interface that has only one validate method (see below). All cell editors provided in JIDE Grids implement this interface.

```java
public interface JideCellEditor extends CellEditor {
    ValidationResult validate(Object oldValue, Object newValue);
}
```

Depending on what validation you need to perform, you can subclass the correct cell editor and override this validate method to do the validation.

```java
public class PositiveOnlyCellEditor extends NumberCellEditor {
    ValidationResult validate(Object oldValue, Object newValue) {
        if(newValue instanceof Integer && ((Integer) newValue).intValue > 0) {
            return ValidationResult.OK;
        } else {
            return new ValidationResult(false, "The number must be positive");
        }
    }
}
```
Once you register this cell editor above on a cell (using either the old JTable way or CellEditorManager way we described earlier), the cell will only accept positive integers.

Please note, the cell level validation can be used only when the validation logic doesn’t need the values on other cells of this table. What if you do need? That’s where you need a table level validation.

**Table Level Validation**

JideTable has `addValidator(Validator)` method to provide the validation logic on the table level. You can add multiple Validators to the table. Those Validators will be called for all cells when they stop editing. Just like Swing listener mechanism, those Validators are called in the opposite order of they were added (that is, the first added Validator will be called at the last). However different from listeners which will continue for all the listeners, if any of the Validators return neither a `null` nor a `ValidationResult.OK`, the validation process will stop right there and no more Validators will be called.

The only method on the `Validator` interface is the `validating` method. See below.

```java
public interface Validator extends EventListener {
    ValidationResult validating(ValidationObject vo);
}
```

This is an interface shared by other JIDE components. But in the case of JideTable, the `ValidationObject` is an instance of `TableValidationObject`. `TableValidationObject` has methods such as `getRow` and `getColumn` so you know which cell it is for. The `TableValidationObject#getSource()` will always be the `JideTable` instance. So in case your validation logic needs to know the values from other cells, you have access to it in the `validating` method. This is something you can’t do if you do validation on the cell level.

**Row Level Validation**

The last level of the validation is between the cell and the table – the row. In fact, in the design of JTable, there is no row editing concept. There are events when cell stops editing, but no specific events are fired for the row when user finishes editing a row. However, this is very useful feature when user tries to edit a database record using a table where they submit and validate once when the row editing (representing a record) is done. In JideTable, we added the support for the row validation through an interface called `RowValidator`. All you need to do is to call `JideTable#addRowValidator(RowValidator)`. `RowValidator` is again an interface just like `Validator`, which has only one `validating` method.
public interface RowValidator extends EventListener {
    ValidationResult validating(RowValidationObject vo);
}

RowValidator will be called when user stops editing a cell and jumps to another row or focuses out the table. If user is still in the same row after stopping the cell editing, RowValidator will not be triggered as the row editing is not done yet.

You can add multiple RowValidator and they will be called for all rows. The RowValidationObject has getRowIndex method which will tell you which row it is validating.

ValidationResult

No matter which level of validation it is, all the validating methods return ValidationResult. ValidationResult contains four fields.

private boolean _valid;
private int _failBehavior = FAIL_BEHAVIOR_REVERT;
private int _id;
private String _message;

The first one is a boolean to indicate whether the validation passed or not. True means the validation is OK. If so, you usually don’t need to worry about other fields. You could either return a predefined ValidationResult.OK or simply return null in this case.

The second one is the failBehavior. If the valid flag is false, it means the validation failed. If so, you also need to tell the table what to do using this failBehavior field. There are three values for it.

/**
 * When validation fails, reverts back to the previous valid value.
 */
public final static int FAIL_BEHAVIOR_REVERT = 0;

/**
 * When validation fails, do not stop cell editing until user enters a valid value or press ESCAPE to cancel the editing.
 */
public final static int FAIL_BEHAVIOR_PERSIST = 1;

/**
 * When validation fails, reset the value to null.
 */
While the other two values are obvious, the only one worth noting is the `FAIL_BEHAVIOR_PERSIST`. If you return this value as the `failBehavior`, you will get a stack trace like this when you run it and make the validation fail.

```java
JTable table = new JideTable(model) {
    @Override
    public void editingStopped(ChangeEvent e) {
        try {
            super.editingstopped(e);
        } catch (EditingNotStoppedException ex) {
            // ValidationResult vr = ex.getValidationResult();
            // handle it
        }
    }
};
```

Don’t panic as this is expected. This is probably the only place in JIDE that we use an exception for a non-exception case. We didn’t catch it but leave it to you do catch and handle it. The code below is the recommended way to handle it. You can display a message box or a message on the status bar to tell your users what is wrong and how to correct the errors. We did it this way is because we want to allow you to handle all validation errors at one place, no matter it is cell-level, row-level or table-level validations. Internally, this exception also gives us a chance to break out from JTable’s stop cell editing process so that the cell editor won’t be removed.
The third and the last one are an integer id and a message. This is something for you to define for your application as we don't read the values for these two fields inside our code at all. You can use them along with your logging system or error/warning system. Using them is optional.

### Undo Support in JideTable

Swing provides `UndoManager`, `UndoableEdit` etc. classes related to the undo/redo. However the only component that has built-in undo support is the `JTextComponent` and its subclasses which uses `AbstractDocument`. `JTable`, `JList`, `JTree`, although they can also be edited, don't support undo by default.

In this section, we will discuss how to enable the undo support in a `JideTable`.

#### DefaultUndoableTableModel and AbstractUndoableTableModel

Most likely you already had a table implemented in your application when you decide to add the undo support. So when we design the undo feature, we want the least change to your existing code. If you are using `DefaultTableModel`, all you need to do is to replace it with `DefaultUndoableTableModel`. If you are using an `AbstractTableModel`, replace it with `AbstractUndoableTableModel`. Of course you also need to use a `JideTable` or one of its subclass such as `SortableTable` as your table class.

Both `DefaultUndoableTableModel` and `AbstractUndoableTableModel` implements `TableUndoableSupport`. If for whatever reason you can't use the default or the abstract implementation, you can also implement `TableUndoableSupport` interface directly and borrow some code from `AbstractUndoableTableModel`.

`JideTable` provides a collection of undo related methods. They are

```java
public void addUndoableEditListener(UndoableEditListener listener);
public void removeUndoableEditListener(UndoableEditListener listener);
public UndoManager getUndoManager();
```

Please note, these undo related methods won't work (returns null or does nothing) unless the inner table model is an instance of `TableUndoableSupport`.

### Keystrokes for Undo and Redo

By default, there is no keystroke for the undo and redo action because we don't know what keystroke to you. But to enable keystrokes is very easy.

```java
InputMap map =
  table.getInputMap(JComponent.WHEN_ANCESTOR_OF_FOCUSED_COMPONENT);
map.put(KeyStroke.getKeyStroke("control Z"), "undo");
```
map.put(KeyStroke.getKeyStroke("control shift Z"), "redo");

Note that we use "undo" and "redo" as the action names. That's because we already registered both actions on JideTable's ActionMap. All you need to do is to map the keystroke to the action on the InputMap.

TransferHandler Enhancement in JideTable

TransferHandler provides the data transfer from and to a Swing component. By default, JTable provides a TransferHandler that only supports copy. In the other word, you can select a range of cells and copy the content to the clipboard. But you can't paste the content.

In JideTable, we created JideTableTransferHandler class which can handle both copy and paste. The copy and paste can be done between a JideTable and Microsoft Excel or any other applications. Drag and drop works too as it is using the same TransferHandler.

The TransferHandler enhancement works with the undo feature. When you copy and paste causing the table model change, you can undo the change as long as you are using a DefaultUndoableTableModel or AbstractUndoableTableModel.

Converter

Before we introduce the next table – ContextSensitiveTable, we have to cover some basic modules that most tables would need.

As we all know, JTable follows MVC design pattern. The Model is the TableModel. The View is the JTable. There could be any type of the data in the TableModel. Unless you use custom cell renderers that can display data in a more fancy way, the default cell renderer only displays String. It means we need some kinds of conversion that converts from any types of data to String so that it can be displayed in the table cells. Editing table is the opposite. It needs a converter that converts from String to any data type. Here comes the ObjectConverter.

Below is the interface of ObjectConverter. All converters implement this interface.

```java
public interface ObjectConverter {
    /**
     * Converts from object to String based on current locale.
     * @param object object to be converted
     * @return the String
     */
    abstract String toString(Object object, ConverterContext context);

    /**
    */
}
```
* If it supports toString method.
* @return true if supports toString
*/
abstract boolean supportToString(Object object, ConverterContext context);

/**
* Converts from String to an object.
* @param string the string
* @return the object converted from string
*/
abstract Object fromString(String string, ConverterContext context);

/**
* If it supports fromString.
* @return true if it supports
*/
abstract boolean supportFromString(String string, ConverterContext context);
}

As an example, assume you are dealing with a Rectangle object, specified as (10, 20, 100, 200). If you represent this Rectangle as the string “10; 20; 100; 200” then 80% of users will probably understand it as a Rectangle with x equals 10, y equals 20, width equals 100 and height equals 200. However, what about the other 20% of the people? Well, they might think it is an int array of four numbers. That’s fine. Users can generally learn by experience: as long as you are consistent across your application, users will get used to it. What is more important is you use the converter consistently across your whole application. If you decide to use “;” as the separator, do not also use “,”.

The situation is slightly more complicated in the case of Color. If we consider the string “0, 100, 200” - if people understand the RGB view of Color then 90% of them will treat as 0 as red, 100 as blue and 200 as green. However, since Color can also be represented in HSL color space (Hue, Saturation, and Lightness), some people may consider it as hue equal 0, saturation equals 100 and lightness equals 200. Another way to represent the color is to use the HTML color name such as “#00FFFF”. If your application is an html editor, you probably should use a converter to convert color to “#00FFFF” instead of “0, 255, 255”. What this means is that, based on your users’ background, you should consider adding more help information if ambiguity may arise.

We also need to consider internationalization, since the string representation of any object may be different under different locales.
In conclusion, we need a series of converters that convert objects so that we can display them as string and convert them back from string. However in different applications, different converters are required.

Although we have already built some converters and will add more over time, it is probably true that there will never be enough. Therefore, please be prepared to create your own converters whenever you need one.

The list below shows all the converters that we currently provide.

![Figure 4 Existing converters](image)

If you want to add your own converters then you can create one quite easily, by implementing a class that extends the `ObjectConverter` interface (i.e. the four methods in this interface). Before you use it, you must register it with `ObjectConverterManager`, which maps from a Class to a converter or several converters. If you want to register several converters for the same object then you can use `ConverterContext` to differentiate them.

There are two static methods on `ObjectConverterManager` that are used to register a converter:

```java
void registerConverter(Class, ObjectConverter).
```
void registerConverter(Class, ObjectConverter, ConverterContext).

To help users adding their own classes that support ConverterContext, we provide an interface called ConverterContextSupport (all of our CellEditor and CellRenderers implement this interface).

We didn’t automatically register the existing converters with the ObjectConverterManager. However, we do provide a method initDefaultConverter() which you can call to register the default converters (as shown below). In addition to converters, this will register the default CellEditors and CellRenderers that we have provided. If you wish to make use of this facility then make sure that you call the initDefaultConverter() method when your application is started.

ObjectConverterManager.initDefaultConverter();

CellEditors and CellRenderers

The usage of CellEditor and CellRenderer is probably one of the most interesting aspects of the original JTable design.

First, let’s review how JTable handles customization of cell editors and cell renderers.

1. You can set a CellRenderer and CellEditor per column using TableColumn’s setCellRenderer and setCellEditor methods respectively.

2. You can set a default CellRenderer and CellEditor per data type using setDefaultRenderer(Class, TableCellRenderer) and setDefaultEditor(Class, TableCellEditor) respectively.

3. The cell renderer or editor set in the first case has a higher priority than it in the second one.

From above, you can see a JTable assumes that each column has the same type of value, and that each data type will use the same type of cell editor. Unfortunately, neither of these assumptions is true in some cases (such as PropertyTable which we will cover later). Fortunately enough, JTable does allow us to add extensions to meet our requirements.

In the case of PropertyTable, each row in the value column may have different types of value, requiring different editors - even when the same underlying data type is being used. In order to support this requirement we have created two new classes: CellEditorManager and CellRendererManager, using an approach similar to the ObjectConverterManager.

If we first consider the CellEditorManager, this allows you to register any cell editor with a given data type, as defined by its class. You can also register a different cell editor to the same type using different contexts. Different from CellRendererManager, CellEditorManager takes CellEditorFactory to make sure an unique cell editor is used for each cell editing.
public static void registerEditor(Class clazz, CellEditorFactory editorFactory, EditorContext context)
public static void registerEditor(Class clazz, CellEditorFactory editorFactory)

As an example, the String class is generally associated with a StringCellEditor. However, if the String is actually a font name then we associate it with a FontNameCellEditor in the context of FontNameCellEditor.CONTEXT. If you still remember the definition of Property, you will recall that the Property class has a field called EditorContext. This means that if you set the EditorContext of a Property to FontNameEditor.CONTEXT then a FontNameCellEditor will be used to edit cells of that type.

registerEditor(String.class, new CellEditorFactory() {
    public CellEditor create() {
        return new StringCellEditor();
    }
});
registerEditor(String.class, new CellEditorFactory() {
    public CellEditor create() {
        return new FontNameCellEditor();
    }
}, FontNameCellEditor.CONTEXT);

The Renderer framework works in a virtually identical manner to the Editor framework.

Both CellRendererManager and CellEditorManager have a method to initialize default editors or renderers, called initDefaultRenderer() and initDefaultEditor(), respectively. Please note that these methods are not called automatically (except in our demo code). This means that if you want to use our default editors and renderers then you must make sure to initialize them yourself before you can use the related classes.
Here is the code to register default editors and renderers.

```java
CellEditorManager.initDefaultEditor();
CellRendererManager.initDefaultRenderer();
```

In fact, in *JIDE Grids*, not just `PropertyTable` uses `CellEditorManager` and `CellRendererManager`. `ContextSensitiveTable` is the table class that starts to use it. If you remember the table hierarchy at the “Class Hierarchy of All the Tables” section, you will see all tables provided in *JIDE Grids* extend `ContextSensitiveTable` except `JideTable`. It means tables such as `SortableTable`, `TreeTable`, `HierarchicalTable`, `CellSpan/StyleTable` all can use `CellEditor/RendererManager`.

`ContextSensitiveTable` uses a table model called `ContextSensitiveTableModel`. `ContextSensitiveTableModel` extends `TableModel` interface and added three more methods. See below.

```java
/**
 * Gets the converter context at cell (row, column).
 *
 * @param row
 * @param column
 * @return converter context
 */
```
ConverterContext getConverterContextAt(int row, int column);

/**
 * Gets the editor context at cell (row, column).
 *
 * @param row
 * @param column
 * @return editor context
 */
EditorContext getEditorContextAt(int row, int column);

/**
 * Gets the type at cell (row, column).
 *
 * @param row
 * @param column
 * @return type
 */
Class getClassAt(int row, int column);

getCellClassAt returns the data type for a cell. If you need different cell renderer or editor for the same data type, returns a different EditorContext in getEditorContextAt() for those cells. Please note, both CellRendererManager and CellEditorManager use EditorContext to look up alought the name of EditorContext has “editor” in it.

The getConverterContextAt method can be used if you want to use the same default cell renderer but want to display the data differently. The converter context is used to find the correct ObjectConverter that can convert from a data type to/from a string so that the default cell renderer can display it. By using this feature, you save the effort of creating a new cell renderer for each data type because in most cases, a data type can be converted to a string.

In addition, we also added setDefaultCellRenderer method to ContextSensitiveTable will allow you to set a cell renderer that will be used for all cells.

Let’s see after this enhancement, how cell renderer can be customized.

1. setDefaultCellRenderer to set the same cell renderer for all cells in a table.
2. You can still set a TableCellRenderer per column using TableColumn’s setCellRenderer methods respectively. This is just like before.
3. You can set your cell renderers to CellRendererManager. Then implement a table model that implementing ContextSensitiveTableModel. The cell renderers will be looked up from CellRendererManager and used.
There is a special case. If you are OK to use `ContextSensitiveCellRenderer` to display your data as string, you can return a different `ConverterContext` in `ContextSensitiveTableModel`. `ContextSensitiveCellRenderer` will use the `ObjectConverter` it finds to convert your data into string.

4. You can still set a default `CellRenderer` per data type using `setDefaultRenderer(Class, TableCellRenderer)`. This is again just like before. But if will only happen if either you called `setCellRendererManagerEnabled(false)` or you didn’t implement `ContextSensitiveTableModel` in your table model.

5. The first case has a higher priority than the second one and so on.

How about cell editor?

1. You can still set a `TableCellEditor` per column using `TableColumn`’s `setCellEditor` methods respectively. This is just like before.

2. You can set your cell editors to `CellEditorManager`. Then implement a table model that implementing `ContextSensitiveTableModel`. The cell editors will be looked up from `CellEditorManager` and used.

3. You can still set a default `CellEditor` per data type using `setDefaultEditor(Class, TableCellEditor)`. This is again just like before. But if will only happen if either you called `setCellEditorManagerEnabled(false)` or you didn’t implement `ContextSensitiveTableModel` in your table model.

4. The first case has a higher priority than the second one and so on.

We prepared G26. ContextSensitiveTableDemo in the examples folder, which shows you how to use this feature.

**NavigableModel and NavigableTable**

Table is very data intensive. For frequent table users (i.e. Excel users), they prefer use keyboard to edit the whole table without ever touching the mouse to slow them down. So it is very important to optimize the cell navigation for those keyboard users.

In JTable, TAB key will navigate to the next cell, SHIFT-TAB will go to the previous cell and ENTER key will go to the cell of the next row. If user is editing the table, one thing we know for sure is if the next cell is not editable, most likely user wants to skip the cell when he/she presses the TAB. Or sometimes user wants to skip all the way to the next empty cell. In the other word, even though the next cell is editable but it is already filled with data, so no need to navigate to it. The actual requirement for this navigation could be more complex than the two simple examples. That’s the reason we decide to introduce `NavigableModel` to allow you to customize the navigation behavior.

See below for the interface of `NavigableModel`. If you read `SpanModel` and `StyleModel`, you will find this interface is familiar. In fact, it’s not just the interface is similar, so is the usage.

```java
public interface NavigableModel {
```
/**
 * Returns if the cell at the given coordinates can be navigated or
 * not.
 *
 * @param rowIndex    The row index
 * @param columnIndex The column index
 * @return <code>true</code> if navigable, <code>false</code> otherwise
 */
boolean isNavigableAt(int rowIndex, int columnIndex);

/**
 * Checks if the navigation is on. If off, <code>@link #isNavigableAt(int,int)</code>
 * should always return <code>true</code> for valid indexes.
 *
 * @return <code>true</code> if on, <code>false</code> otherwise
 */
boolean isNavigationOn();

You can implement this <em>NavigableModel</em> on any table model. You will implement the
<code>isNavigableAt</code> method to decide if this cell is navigable when user presses TAB, SHIFT-TAB or
ENTER key.

We also create <em>NavigableTable</em> class to use <em>NavigableModel</em>. However you don’t have to use
it explicitly as most other JIDE tables such as <em>SortableTable</em>, <em>PropertyTable</em>, and <em>TreeTable</em>
extend it.

**Customize the navigation keys**

We mentioned TAB, SHIFT-TAB or ENTER keys as the default navigation keys. In fact, JTable
also supports all the arrow keys such as LEFT, RIGHT, UP, DOWN, PGUP, PGDN, HOME and END.
By default, those keys will obey use NavigableModel. If you want some of the keys to use
NavigableModel, you can override this method to do it. Just return true if you want to treat
them as navigable keys.

```java
protected boolean isNavigationKey(KeyStroke ks)
```
CellSpanTable

The next table is the CellSpanTable which one cell can span several rows or columns or both. To know what it is, we need to understand CellSpan first.

CellSpan defines how a cell spanning across several rows or/and columns. CellSpan has four fields. They are row, column, rowSpan, columnSpan. All are int type. The (row, column) is actually the anchor cell of the cell span. rowSpan tells how many rows the cell will span. columnSpan tells how many columns the cell will span. You can view them as Rectangle’s y, x, height and width respectively.

Let’s use the following table as an example. The cell marked as light green is a cell span. The cell span is defined as row = 1, column = 1, rowSpan = 2, and columnSpan = 3.

<table>
<thead>
<tr>
<th>(0, 0)</th>
<th>(0, 1)</th>
<th>(0, 2)</th>
<th>(0, 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2, 0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(3, 0)</td>
<td>(3, 1)</td>
<td>(3, 2)</td>
<td>(3, 3)</td>
</tr>
</tbody>
</table>

Now you know how to define a cell span. But how to pass the cell span information to table so that it knows how to paint the cells correctly? Instead of introducing a whole separate model, we decided to use TableModel. First we defined SpanModel as an interface and then SpanTableModel which combines SpanModel and TableModel together. See below.

```java
public interface SpanModel {
    /**
     * Gets the cell span at the specified row and column.
     * @param rowIndex
     * @param columnIndex
     * @return CellSpan object.
     */
    CellSpan getCellSpanAt(int rowIndex, int columnIndex);

    /**
     * Checks if the span is on. There are two meanings if it returns false.
     * It could be getCellSpanAt() return a valid value but
     * It could also be the span model is empty which means getCellSpanAt() always return null.
     * @return true if span is on. Otherwise false.
     */
    boolean isCellSpanOn();
}
```
* \texttt{SpanTableModel} interface combines \texttt{TableModel} and \texttt{SpanModel}.

```
public interface SpanTableModel extends TableModel, SpanModel {
}
```

When you define your table model which you have to do it anyway, you can define what \textit{SpanModel} is. Let’s see some real examples.

This is a table with every third row spanning across the whole table width. The code to do it is very simple at the \texttt{getCellSpanAt(rowIndex, columnIndex)} method. Every third row translated to \texttt{if \( \text{rowIndex} \mod 3 == 0 \)}. So if \texttt{rowIndex} can be divided by 3, returns a cell span that anchor cell is at the first cell in that row and \texttt{columnSpan} is column count.

```
class SpanTableTableModel extends AbstractSpanTableModel {
    public int getRowCount() {
        return 10000;
    }

    public int getColumnCount() {
        return 9;
    }

    public boolean isCellEditable(int rowIndex, int columnIndex) {
```
return false;
}

public Object getValueAt(int rowIndex, int columnIndex) {
    return "" + rowIndex + "," + columnIndex;
}

public CellSpan getCellSpanAt(int rowIndex, int columnIndex) {
    if (rowIndex % 3 == 0) {
        return new CellSpan(rowIndex, 0, 1, getColumnCount());
    }
    return null;
}

public boolean isCellSpanOn() {
    return true;
}
}

Here is another more complex example. You can read the code to figure out how it works.

![CellSpanTable](image)

**Figure 7 CellSpanTable**

class SpanTableTableModel extends AbstractSpanTableModel {
    public int getRowCount() {
    
}
You probably already noticed we used `AbstractSpanTableModel` in the examples above which we haven’t introduced. `AbstractSpanTableModel` extends `AbstractTableModel` and implements `SpanTableModel` and provides default implementation onto the two methods defined in `SpanModel`. Please note, it returns false in `isCellSpanOn()` which means if you want to see cell span, you have to override it and return true. The reason we did is because there are other table components that extends `CellSpanTable`. We don’t want cell span is turned on by mistake.

`AbstractSpanTableModel` also provides support for `SpanModelListener` and `SpanModelEvent`.

```
class AbstractSpanTableModel extends AbstractTableModel implements SpanTableModel {
  
  return 10000;
}

public int getColumnCount() {
  return 9;
}

public boolean isCellEditable(int rowIndex, int columnIndex) {
  return false;
}

public Object getValueAt(int rowIndex, int columnIndex) {
  return "" + rowIndex + "," + columnIndex;
}

public CellSpan getCellSpanAt(int rowIndex, int columnIndex) {
  if ((rowIndex % 3) != 2 && (columnIndex % 3) != 2) {
    return new CellSpan(rowIndex - rowIndex % 3, columnIndex - columnIndex % 3, 2, 2);
  }
  return super.getCellSpanAt(rowIndex, columnIndex);
}

public boolean isCellSpanOn() {
  return true;
}
```
There is another table model called DefaultSpanTableModel. It extends DefaultTableModel and implements SpanTableModel. In this model, you get additional methods such as addCellSpan, removeCellSpan.

Once we understand the SpanTableModel and different subclasses, CellSpanTable is very simple to use. There is no additional method on it except three methods starting with “orginal”. They are originalRowAtPoint, originalColumnAtPoint and originalGetCellRect. They are handy when you need to get the old cell rectangle, row and column without considering the cell spans but in most cases you won’t need them.

In general, CellSpanTable is very fast. In the example above, we have 10000 rows in the table and fairly complex cell spans, we couldn’t even notice any difference when scrolling up and down quickly. However since it’s user’s responsibility to implement getCellSpanAt() method, how complex this method is does affect the speed of CellSpanTable. So when you implement this method, please make sure you optimize it as much as you can. We suggest you to use the same CellSpan instance over and over again so that it will increase the memory usage in a short period of time (although it will be garbage collected later). The other performance tuning technique you can use in this case is to cache the cell span if the cell span is very difficult to calculate. Of course then you have to deal invalidating the cache when table data changes.

Since Java doesn’t support multiple inherent, we have to put CellSpanTable at a lower level. Most table components in JIDE Grids are actually CellSpanTable such as TreeTable, PropertyTable, HierarchicalTable, SortableTable etc. There is one reason why we added isCellSpanOn() method to SpanModel. If your table model is not an instance SpanModel or it is but isCellStyleOn() returns false, there is no performance hit at all even if your table extends CellSpanTable.

---

6 Please note, we didn’t use the same CellSpan instance in the two examples above as it will make the code a little harder to read. However for you it’s very simple to change it to reuse the same instance. All you need is to create a field of type CellSpan and set different row, column, rowSpan and columnSpan value on it and return the same thing. If there is no cell span, still return null.
One of the common mistakes when defining a span table model is the span definition inconsistence. The span definition inconsistence refers to a definition error in `getCellSpanAt` method when the cells in the same cell span return different CellSpans. The result is the display corruption as shown on the screenshot above. The display corruption is intermittent and the result may not be the same all the time, which make it hard to identify. The best way to detect this error is to use a static `verifyCellSpan` method on `CellSpanTable`. If there are errors, this method will print out something like below.

```
Cell at (0, 1) has a different cell span from that of the anchor cell (0, 0)
  (0, 0)-> CellSpan(0, 0, 2, 2)
  (0, 1)-> CellSpan(0, 1, 1, 1)
Cell at (1, 0) has a different cell span from that of the anchor cell (0, 0)
  (0, 0)-> CellSpan(0, 0, 2, 2)
  (1, 0)-> CellSpan(1, 0, 1, 1)
Cell at (1, 1) has a different cell span from that of the anchor cell (0, 0)
  (0, 0)-> CellSpan(0, 0, 2, 2)
  (1, 1)-> CellSpan(1, 1, 1, 1)
...```

The message above simply tells you that the cell span at (0, 1), (1, 0) and (1, 1) should all be the same which should be `CellSpan(0, 0, 2, 2)`.

### CellStyleTable

What is cell style? Usually all the cells in the same table has the same color, same font, same alignment (at least in the same column). However there are many cases we want to have different styles on each cell to highlight certain cells or archive the strips effect. In a regular JTable, we usually use cell renderers to do it. You will end up with creating several cell renderers.
to do various cell styles. CellStyleTable provides a better and easier way to archive this and more.

Cell styles include the foreground, the background, the font, the alignments (horizontal and vertical), the border, the icon, etc. If you think there are other styles that should be added, just let us know. It’s very simple to introduce new styles.

The design of CellStyleTable is almost the same as CellSpanTable. Instead of CellSpan, we gotCellStyle class. CellStyle is just a collection of the styles we mentioned above. Then we gotStyleModel and StyleTableModel.

```java
public interface StyleModel {
    /**
     * Gets the cell style at the specified row and column.
     * @param rowIndex
     * @param columnIndex
     * @return CellStyle object.
     */
    CellStyle getCellStyleAt(int rowIndex, int columnIndex);

    /**
     * Checks if the style is on. There are two meanings if it returns false. It could be
     * getCellStyleAt() return a valid value but it could also be the style model is empty
     * which
     * @param rowIndex
     * @param columnIndex
     * @return true if style is on. Otherwise false.
     */
    boolean isCellStyleOn();
}
/**
 * <code>StyleTableModel</code> interface combines <code>TableModel</code> and
 * <code>StyleModel</code>. 
 */
public interface StyleTableModel extends TableModel, StyleModel {
}
```

If you read SpanTableModel section, you will see this StyleTableModel is almost a copy of it except changing from "Span" to "Style". We intentionally did it this way so that you can get familiar with those interfaces easily. When you implement StyleModel, all you need to do is to return a desired CellStyle at getCellStyleAt(int rowIndex, int columnIndex) method.
Available CellStyles

There are many cell styles that can be used in various situations to make your table more interesting.

<table>
<thead>
<tr>
<th>Style Name</th>
<th>Data Type</th>
<th>Renderer Component Type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>background</td>
<td>Color</td>
<td>Any Components</td>
<td></td>
</tr>
<tr>
<td>foreground</td>
<td>Color</td>
<td>Any Components</td>
<td></td>
</tr>
<tr>
<td>selectionBackground</td>
<td>Color</td>
<td>Any Components</td>
<td></td>
</tr>
<tr>
<td>selectionForeground</td>
<td>Color</td>
<td>Any Components</td>
<td></td>
</tr>
<tr>
<td>font</td>
<td>Font</td>
<td>Any Components</td>
<td></td>
</tr>
<tr>
<td>fontStyle</td>
<td>int</td>
<td>Any Components</td>
<td></td>
</tr>
<tr>
<td>icon</td>
<td>Icon</td>
<td>JLabel, AbstractButton</td>
<td></td>
</tr>
<tr>
<td>verticalAlignment</td>
<td>int</td>
<td>JLabel, AbstractButton</td>
<td></td>
</tr>
<tr>
<td>horizontalAlignment</td>
<td>int</td>
<td>JLabel, JButton, JTextField, AbstractComboBox</td>
<td></td>
</tr>
<tr>
<td>verticalTextPosition</td>
<td>int</td>
<td>JLabel, AbstractButton</td>
<td></td>
</tr>
<tr>
<td>horizontalTextPosition</td>
<td>int</td>
<td>JLabel, AbstractButton</td>
<td></td>
</tr>
<tr>
<td>text</td>
<td>String</td>
<td>JLabel, AbstractButton</td>
<td></td>
</tr>
<tr>
<td>toolTipText</td>
<td>String</td>
<td>Any JComponents</td>
<td></td>
</tr>
<tr>
<td>border</td>
<td>Border</td>
<td>Any JComponents</td>
<td></td>
</tr>
<tr>
<td>overlayBorder</td>
<td>Border</td>
<td>Any Components</td>
<td></td>
</tr>
<tr>
<td>overlayCellPainter</td>
<td>CellPainter</td>
<td>Any Components</td>
<td></td>
</tr>
<tr>
<td>underlayCellPainter</td>
<td>CellPainter</td>
<td>Any Components</td>
<td></td>
</tr>
</tbody>
</table>

The text style allows you to display a specified text instead of the actual value of the cell.

Where to DefineCellStyle

There are two kinds of cell styles. One is the style depending on the data of the cell. For example, showing red background if the value is negative, displaying an up or down icon if the data (representing stock price change maybe) is positive or negative. The other case is the style that is not depending on the data. For example, the row stripes, the column strips etc.

For the first case, you should define StyleModel on the table model and returnCellStyle from the getCellStyleAt method. For the second case, the best way is to define it onCellStyleTable using setTableStyleProvider.

As row/column stripes is so popular, we even created RowStripeTableStyleProvider and ColumnStripeTableStyleProvider, you can use it simply by calling
CellStyleTable table = ...;
table.setTableStyleProvider(new RowStripeTableStyleProvider(new Color[]{...}); // pass
in the alternative colors here

CellStyle Merging

When there is a pipe of several table models, *StyleModel* can be implemented at any of the
table models. *TableStyleProvider* also provides a cell style. Sometimes, the style depends on the
actual table data. If so, you probably should implement the *StyleModel* on the actual table
model level to avoid index conversion. Sometimes, the style depends on the row position in the
view, such as alternative row stripes. In this case, you will use *CellStyleTable*’s
*TableStyleProvider*. We allow you to define *CellStyle* at the appropriate *TableModel* and we will
merge the styles for you. When there are conflicts in the styles, we allow you to define the
priority using *CellStyle*’s *setPriority(int)*. By default, if priorities for all cell styles are the same, the
inner model has the highest priority and *TableStyleProvider* on *CellStyleTable* has the lowest
priority.

SortableTable

Comparator

Before we introduce *SortableTable*, we first have to introduce the
*ObjectComparatorManager*. This works in a similar manner to the *ObjectConverterManager*
which we have already discussed, with the difference being that *ObjectComparatorManager* is a
central place for managing comparators. Registration of comparator can be done using the
following two methods on *ObjectComparatorManager*.

```java
public static void registerComparator(Class clazz, Comparator comparator);
public static void unregisterComparator(Class clazz);
```

The figure to the right shows the standard comparators that we provide. If an object type
implements Comparable then you can use *ComparableComparator*. If the object can be
compared as a string (based on the value of toString())
then you can use *DefaultComparator*. We also provide
several comparators for existing data types such as
*NumberComparator* for any Number, *BooleanComparator*
for Boolean and *CalendarComparator* for Calendars.
Alternatively, you can write your own comparator and register it with *ObjectComparatorManager*. 
Sortable Table

A `SortableTable`, as the name indicates, is a table that can sort on each column. Usually a `SortableTable` can only sort on one column. However this `SortableTable` can sort on multiple columns, as shown below:

![SortableTable under different L&F](image)

In a `SortableTable`, clicking on table column header will sort the column: the first click will sort ascending; the second click descending; the third click will reset the data to the original order. To sort on multiple columns, you just press CTRL key and hold while clicking on the other columns. A number is displayed in the header to indicate the rank amongst the sorted columns. As an example, in the screen shot above, the table is sorted first by “boolean column”, then by “double column” and then by “string column”.

![Multiple Column Sorting](image)

SortableTableModel

The core part of `SortableTable` is not the table itself but the `SortableTableModel`. This can take any standard table model and convert to a suitable table model for use by `SortableTable`. Note that we wrap up the underlying table model, which means that when you sort a column, the original table model is unchanged (you can always call `getActualModel()` to get the original table model).
As a developer, how do I use it

It’s very easy to use *SortableTable* in your application. Just create your table model as usual, but instead of setting the model to *JTable*, set it to *SortableTable*. If you have your own *JTable* class which extends *JTable* then you will need to change it to extend *SortableTable*.

```java
TableModel model = new SampleTableModel();
SortableTable sortableTable = new SortableTable(model);
```

In the example above, *SampleTableModel* is just a normal table model. When you pass the model to *SortableTable*, *SortableTable* will create a *SortableTableModel* internally. This means that when you call *sortableTable.getModel()* it will return you the *SortableTableModel*, not the *SampleTableModel*. However if you cast the table model you get to *TableModelWrapper* and then call *getActualModel()* on the table model, the *SampleTableModel* will be returned. If you use several levels of *TableModelWrappers* such as *FilterableTableModel*, you may want to use *TableModelWrapperUtils#getActualTableModel(TableModel,Class)* to find the inner most table model.

Sometimes, you need to know the actual row since it’s different visually. For example, although the first row may appear to be selected, since the table could be sorted, this may not be the first row in the actual table model. Here are the two methods you need to know: *getActualRowAt(int row)* and *getSortedRowAt(int row)*. The first one will return you the actual row in the actual table model by passing the row index on the screen; the second one does the opposite mapping. In fact, we suggest you to use *TableModelWrapperUtils#getActualRowAt* and *getRowAt* to do the index conversion once you start to use *FilterableTableModel* along with *SortableTableModel*. We will cover *FilterableTableModel* and *SortableTableModel* are table model wrappers. Once there are several models that one wraps the other, *TableModeWrapperUtils* will allow you to find the row index at any table model level, v.s. the two methods on *SortableTableModel* only allows you to find one level at a time.

There are several options you can use to control *SortableTable*. For example, *setMultiColumnSortable(Boolean)* allows you to enable/disable multiple column sort. Similarly, if you have better icons for showing the ascending/decending option on the table header, then you can call *setAscendingIcon(ImageIcon)* to *setDescendingIcon(ImageIcon)* to replace the standard icons.

As has already been explained, the user can click the column header to sort a column. In addition you can call *sortColumn()* to sort a column programmatically. You can sort either by column index, or by column name. The interface for sorting on *SortableTable* is really simple. If you want to sort by multiple columns programmatically, you will have to use *SortableTableModel* to do it. This provides more methods will allow you to sort several columns or even un-sort a sorted column.
How to Compare

You may think we use Comparator all the time to compare two values. However that’s not the case by default. The actual comparison is done in this method of `SortableTableModel`.

```java
protected int compare(Object o1, Object o2, int column)
```

We will first to see if the two objects are String. If yes, we use String’s `compareToIgnoreCase` method to compare. Then we will see if the two objects are `Comparable`. If yes and they are type compatible, we will use either `o1.compareTo(o2)` or `o2.compareTo(o1)` to compare the two values. `compareTo(…)` is the method on `Comparable` interface. If none of the above is true, we will finally use `Comparator` to compare. `Comparator` is looked up from `ObjectComparatorManager` using the type returned `getColumnClass` as the primary key and the context returned from `getComparatorContext` as the second key. The main reason we did it this way is because we found people always forgot to override `getColumnClass()` method then complained to us why sorting is not working as expected. By using `compareTo` method, it will work in most cases without depending on `getColumnClass()`. Saying that, there are cases you have to use `Comparator` because, for example, you provide some customized compare logic in it. If so, you can call `SortableTableModel`’s `setAlwaysUseComparators(true)` so that we will always use `Comparator` to compare without even checking if the objects are `Comparable`.

The performance of `SortableTableModel`

We tried to optimize the performance of `SortableTableModel`. The basic strategy is if the table is completely unsort, we will sort the whole table model which could be slow if the table is huge. If the table is sorted already and a new row is added/deleted/updated, we will do incremental sorting. For example, insert the new row to the correct position directly.

As we have no idea of what the data might look like, we have to use a generic sorting algorithm. In our case, we used shuttle sort. It’s a very fast algorithm comparing to others. However depending on how large the table model is, it could potentially take a long time on a large table model. In the actual use cases, user knows very well about the data. So they could develop a sorting algorithm that is customized to the particular data.

When the table is sorted, a new row is added or deleted or some values are updated in the underlying table model, we won’t resort the whole table again. We will listen to the table model event from underlying table model and do incremental sort. Binary search is used by default as the table is sorted already. In this case, as user of `SortableTableModel`, you need to fire the exact table model event to tell `SortableTableModel` what happened in underlying table model. If you use `DefaultTableModel`, all those are done automatically. If you implement your own underlying table model basing on `AbstractTableModel`, you need to fire the table model event. You can refer to the source code of `DefaultTableModel` to figure out what event to fire. If an incorrect event is fired, the sorting result will be unpredictable.

Generic speaking, in our testing environment, the performance of `SortableTableModel` is good enough. You can try in using `LargeSortableTableDemo` we included in our examples.
However if you have an even better algorithm or as I said you know your data very well so that you can play some tricks to make sorting faster, we allow you to do so.

First, you need to extend SortableTableModel. There are three methods you need to know in order to plug in your own algorithm. They are

protected void sort(int from[], int to[], int low, int high);
protected int search(int[] indexes, int row);
protected int compare(int row1, int row2);

When the table model is completely unsorted, sort() will be used to sort the whole table model. If the table is sorted already, search() will be called to find out where a new row to be added or an existing row to be deleted. So method sort can be overwritten if you want to use your own sort algorithm. Method search can be overwritten if you want to use your own search algorithm.

In either sort() or search(), if you want to compare two rows, using the compare() method. Usually you don’t need to overwrite it. To make it easier to understand, here is the source code we used to do the search and sort for your reference.

Search algorithm:

```java
protected int search(int[] indexes, int row) {
    return binarySearch(indexes, row);
}

private int binarySearch(int[] indexes, int row) {
    // use binary search to find the place to insert
    int low = 0;
    int high = indexes.length - 1;
    int returnRow = high;

    boolean found = false;
    while (low <= high) {
        int mid = (low + high) >> 1;
        int result = compare(indexes[mid], row);
        if (result < 0)
            low = mid + 1;
        else if (result > 0)
            high = mid - 1;
        else {
            returnRow = mid; // key found
            found = true;
            break;
        }
    }
    return returnRow;
}
```
Sort algorithm:

```java
protected void sort(int from[], int to[], int low, int high) {
    shuttlesort(from, to, low, high);
}

private 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;

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

    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++];
    }
```
Filter and FilterableTableModel

Table is used to display data. Sometimes users are only interested in some important rows, so they would like to filter other rows away. This is why we need a component which can do filter on the table model.

Filter is an interface which is used to filter data. The main method is isValueFiltered(Object value). If a value should be filtered, this method should return true. Otherwise, returns false. It also defined other methods such as setters and getters for name, enabled as well as methods to add/remove FilterListener. AbstractFilter is the default implement of Filter interface. It implements most of the methods in Filter except isValueFiltered(). Subclasses should implement this method to do the right filtering.

There is no class called FilterTable because the filtering happens on the table model portion. There is no code need to be added to the table portion to do the filtering. There is a FilterableTableModel. It’s also a table model wrapper just like SortableTableModel. FilterableTableModel allows you to add filter for each column or to the whole table. It will use those Filter and call isValueFiltered() to decide which value to be filtered.

In most cases, you can use Filter interface or AbstractFilter as the filters. But if the filter logic depends on the row index and column index, you may need to use TableFilter or AbstractTableFilter. TableFilter has getRowIndex and getColumnIndex methods which you can use to get the row and column index.

AutoFilterTableHeader

Figure 11 AutoFilterTableHeader
AutoFilterTableHeader implements auto-filter feature on the table header. Each column header has a combobox-like control to allow user selecting certain value(s) to be filtered from the list. The list contains the possible values for that column as well as other customized items. Each item represents a Filter class that will be added to FilterableTableModel when selected.

Since JIDE 3.0 release, we redesigned the AutoFilterTableHeader so that it will only show the filter drop down button when the mouse is over the column. It means by default, AutoFilterTableHeader will look just like a regular JTableHeader.

AutoFilterTableHeader works with any FilterableTableModel. If the table passed to the constructor of AutoFilterTableHeader already defined a FilterableTableModel, it will use that FilterableTableModel. Otherwise, it will create a new FilterableTableModel, wrapping the current table's table model and reset the table's table model to the newly created FilterableTableModel.

Among the drop down list, there is a “Custom…” entry. If you click on it, it will bring up a dialog that can define customer filters.

User can choose all kinds of built-in filters from this dialog. We will cover more information in the CustomFilterEditor section below.

AutoFilterTableHeader also supports multiple filter values. See below. We will use a CheckBoxList to allow you to select multiple values.

There are three kinds of Filters AutoFilterTableHeader will use. They are SingleValueFilter, MultipleValuesFilter and DynamicTableFilter. When isAllowMultipleValues() returns false and user selects a value from AutoFilterTableHeader's drop down value list, SingleValueFilter will be created and added to that column as the filter. DynamicTableFilter could also be used in this case when you call AutoFilterBox#addDynamicTableFilter(DynamicTableFilter). This method call will add new custom filter to the header which will appear as a new item under "All" item in the
drop down value list. If `isAllowMultipleValues()` returns true, `MultipleValuesFilter` will be the only filter that is used to allow multiple values as the filter values.

`AutoFilterTableHeader` also detects filters added outside `AutoFilterTableHeader`. For example, if `isAllowMultipleValues` returns false, you can add a `SingleValueFilter` to the column, or if `isAllowMultipleValues` returns true, you can add a `MultipleValuesFilter`. After you did it, `AutoFilterTableHeader` will automatically update the display to show a filter name or filter name to indicate the column has a filter.

`AutoFilterTableHeader` also supports the filter indicator on the header. It can be icon only, filter value(s) only, or both. In the case of `isAllowMultipleValues` is true, we will display the number of values instead of the actual values as there could a lot of them.

<table>
<thead>
<tr>
<th>CategoryName</th>
<th>ProductName: Alice Mutton</th>
<th>ProductSales</th>
<th>ShippedDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>CategoryName</td>
<td>ProductName: (3)</td>
<td>ProductSales</td>
<td>ShippedDate</td>
</tr>
</tbody>
</table>

### TableModelWrapper

We just talked about `SortableTableModel` and `FilterableTableModel`. In fact, both implements `TableModelWrapper` interface.

```java
/**
 * <code>TableModelWrapper</code> is a wrapper around table model which is referred
 * as the actual table model or underlying table model. It can be used to provide a
 * different view to the actual model. A typical use case is `SortableTableModel`.
 */
public interface TableModelWrapper {
    /**
     * Gets the underlying table model.
     *
     * @return the underlying table model.
     */
    TableModel getActualModel();
}
```

On top of `TableModelWrapper`, we also have `RowTableModelWrapper` and `ColumnTableModelWrapper`. Let’s look at an example of how we use `RowTableModelWrapper` to implement sorting. Instead of modifying the actual table model, we just provide a row index mapping. The first row after sorting is actually the 4th row in the actual model. So all we need is to put 3 at the first index in the mapping, and so on.
For FilterableTableModell, it is the same thing. We just remove the row indices that are not satisfy the filter condition.

Since both FilterableTableModel and SortableTableModel are table model wrappers, you can connect them to for a “pipe”.
That’s how you implement both sorting and filtering feature in JIDE Grids.

We also provide `TableModelWrapperUtils` which has a bunch of methods to allow you to convert the row index. For example, if you know the first row is selected in the table, you need to find out the actual row index in the actual table model, `TableModelWrapperUtils.getActualRowAt` is the one to use.

### CompoundTableModelEvent

In the case `TableModelWrappers`, events are passed from model to model. In some cases, the event could change its meaning. A row deleted event will not be fired on an outer model because the model is a `FilterableTableModel` and the deleted row is already filtered. A continuous rows deleted event may become several discontinued row deleted event because `SortableTableModel` changed the row order. It will cause some problems.

Since 3.0 release, we introduced this `CompoundTableEvent` to handle some of those complicated scenarios. For example, if a sorted `SortableTableModel` receives a row updated event from its underlying table model, the updated row could change its position. In this case, `SortableTable` needs to get a `tableRowsDeleted` event for the original position and another `tableRowsInserted` event for the updated position. However, it’s not appropriate to fire two separate events for one action. It will also impact the performance if we simply fire `tableDataChanged` event. That’s why we introduce this class.

Basically, JIDE fires this `CompoundTableModelEvent` only if there are two or more child events embedded in. The event itself will represent as a `tableDataChanged` event. That means, if you don’t try to distinguish it with regular `TableModelEvents`, it will fall into the branch you handle `tableDataChanged` event. However, it doesn't mean that `CompoundTableModelEvent` has to be a `tableDataChanged` event. You could fire whichever event you want following the instruction of `TableModelEvent`.

You could invoke `getEvents()` to get the embedded child events inside the event. The order of those child events inside the `CompoundTableModelEvent` matters. The listener is expected to handle the embedded event one by one.

For example, if a `CompoundTableModelEvent` contains two events, the first one is row insertion at 2 while the other one is row deletion at 5, you need insert first then delete. Otherwise, you may get a wrong result.

### Sortable/Filterable List and Tree

We just talked about `SortableTableModel` and `FilterableTableModel`. In you still remember, both models are actually table model wrappers. The same technique can be applied to `ListModel` and `TreeModel` as well.

Corresponding to `TableModelWrapper`, there are `ListModelWrapper` and `TreeModelWrapper`. Each of them has corresponding sortable and filterable implementations. See below.
SortableListModel and SortableTreeModel

Different from SortableTable, there is no SortableList or SortableTree. The reason is SortableTable doesn’t really do the sorting. It’s the SortableTableModel who does the sorting. SortableTable just provides a special table header which can accept mouse clicks. It still delegates to SortableTableModel to sort the data. In the case of JList and JTree, there is no header. In fact, there is no uniform way on the UI to make JList or JTree sortable. So there is no need for SortableList and SortableTree. To make JList sortable,

```
SortableListModel sortableListModel = new SortableListModel(listModel);
JList sortableList = new JList(sortableListModel);
```

To sort the list, you just call

```
sortableListModel.sort(); // or sort(SortableListModel.SORT_DESCENDING) to sort descending.
```

Usually you can hook up this action with a button somewhere near the JList or on a local toolbar.

It’s exactly the same way to make JTree sortable.

FilterableListModel and FilterableTreeModel

`FilterableListModel` is a list model wrapper which wraps another list model so that user can apply filters on it. You can use `addFilter(Filter)` to a filter to the ListModel. By default, filters won’t take effect immediately. You need to call `setFiltersApplied(true)` to apply those filters. If `filtersApplied` flag is true already, you just need to call `refresh()`. We don’t refresh automatically because you might have several filters to add. You can add all of them, then only call `refresh` once.

`setFiltersApplied(boolean)` controls all the filters. Each filter has its own enabled flag which will control each individual filter.
By default, if you have more than one filters, all filters will be used in AND mode. You can see javadoc of `setAndMode(boolean)` to find more information.

Again, `FilterableTreeModel` works exactly the same way.

## More Filters and CustomFilterEditor

From `Filter` interface, we created many built-in filters.

<table>
<thead>
<tr>
<th>Filter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EqualFilter</td>
<td>Equal</td>
</tr>
<tr>
<td>NotEqualFilter</td>
<td>Not equal</td>
</tr>
<tr>
<td>GreaterThanFilter</td>
<td>Greater than</td>
</tr>
<tr>
<td>LessThanFilter</td>
<td>Less than</td>
</tr>
<tr>
<td>GreaterOrEqualFilter</td>
<td>Greater than or equal</td>
</tr>
<tr>
<td>LessOrEqualFilter</td>
<td>Less than or equal</td>
</tr>
<tr>
<td>BetweenFilter</td>
<td>Between an inclusive range</td>
</tr>
<tr>
<td>NotBetweenFilter</td>
<td>Not between an inclusive range</td>
</tr>
<tr>
<td>LikeFilter</td>
<td>Search for a pattern. % is the wildcard for any char and _ is the wildcard for one char, similar to SQL LIKE statement</td>
</tr>
<tr>
<td>NotLikeFilter</td>
<td>Search for opposite of a pattern</td>
</tr>
<tr>
<td>WildcardFilter</td>
<td>Search for a pattern based using wildcards</td>
</tr>
<tr>
<td>RegexFilter</td>
<td>Search for a pattern based on regular expression</td>
</tr>
<tr>
<td>InFilter</td>
<td>If you know the exact value you want to return for at least one of the columns</td>
</tr>
<tr>
<td>NotInFilter</td>
<td>If you know the exact values you want to exclude</td>
</tr>
<tr>
<td>DateOrCalendarFilter</td>
<td>An abstract Filter for Date or Calendar filters</td>
</tr>
<tr>
<td>TodayFilter</td>
<td>Filter all other dates except today's date</td>
</tr>
<tr>
<td>YesterdayFilter</td>
<td>Filter all other dates except yesterday's date</td>
</tr>
<tr>
<td>TomorrowFilter</td>
<td>Filter all other dates except tomorrow's date</td>
</tr>
<tr>
<td>ThisWeekFilter</td>
<td>Filter all other dates except it is this week</td>
</tr>
<tr>
<td>LastWeekFilter</td>
<td>Filter all other dates except it is in the last week</td>
</tr>
<tr>
<td>NextWeekFilter</td>
<td>Filter all other dates except it is in the next week</td>
</tr>
<tr>
<td>ThisMonthFilter</td>
<td>Filter all other dates except it is this month</td>
</tr>
<tr>
<td>LastMonthFilter</td>
<td>Filter all other dates except it is in the last month</td>
</tr>
<tr>
<td>NextMonthFilter</td>
<td>Filter all other dates except it is in the next month</td>
</tr>
<tr>
<td>ThisQuarterFilter</td>
<td>Filter all other dates except it is in this quarter</td>
</tr>
<tr>
<td>LastQuarterFilter</td>
<td>Filter all other dates except it is in the last quarter</td>
</tr>
<tr>
<td>NextQuarterFilter</td>
<td>Filter all other dates except it is in the next quarter</td>
</tr>
<tr>
<td>ThisYearFilter</td>
<td>Filter all other dates except it is in this year</td>
</tr>
<tr>
<td>LastYearFilter</td>
<td>Filter all other dates except it is in the last year</td>
</tr>
<tr>
<td>NextYearFilter</td>
<td>Filter all other dates except it is in the next year</td>
</tr>
<tr>
<td>MonthFilter</td>
<td>Filter all other dates except the specified month of any year</td>
</tr>
<tr>
<td>QuarterFilter</td>
<td>Filter all other dates except the specified quarter of any year</td>
</tr>
<tr>
<td>YearFilter</td>
<td>Filter all other dates except the specified year</td>
</tr>
</tbody>
</table>

As a developer, you can always create those filters using code and add them to `FilterableTableModel`, `FilterableListModel` or `FilterableTreeModel` and use them. To make it easy to be used by end users, we create an editor to create filters.
FilterFactoryManager

Different data type support different filters. In order to support this feature, we created FilterFactoryManager to make it possible. The main method on this class is:

```java
public void registerFilterFactory(Class type, FilterFactory filter)
```

FilterFactory defines three methods:

```java
public interface FilterFactory {
    Filter createFilter(Object... objects);
    String getConditionString(Locale locale);
    String getName();
    Class[] getExpectedDataTypes();
}
```

The first `createFilter` method is to create the filter. The other two methods are there so that the filter can be created interactively from a user interface. The user interface is CustomFilterEditor.

We registered four categories of FilterFactory to FilterFactoryManager. They are for number, string, date/calendar and boolean.

CustomFilterEditor

CustomFilterEditor provides an interface to create a filter based on the data type.

For the string data type, we defined the following filters.

User can select one from the drop down list above and then input the value for the filter. For example, the selection below will create WildcardFilter same as the code below.
WildcardFilter filter = new WildcardFilter();
filter.setBeginWith(true);
filter.setEndWith(false);
return filter;

For numbers (any objects extends Number and the primitives), we predefined the following filters.

For booleans, we defined the following filters.

For Date or Calendar, we defined the following filters.

Those four data types should be good enough to cover most use cases. If you have other data type that you want to add, you just need to register them to FilterFactoryManager. For example, here is the code to register the “begin with” filter on String.

```java
FilterFactoryManager.registerFilterFactory(String.class, new FilterFactory() {
    public Filter createFilter(Object... objects) {
        WildcardFilter beginWith = new WildcardFilter((String) objects[0]);
        beginWith.setBeginWith(true);
        beginWith.setEndWith(false);
        return beginWith;
    }
});
```
public Class[] getExpectedDataTypes() {
    return new Class[] {String.class};
}

public String getConditionString(Locale locale) {
    return AbstractFilter.getConditionString(locale, "string", "beginWith");
}
}};

The `getExpectedDataTypes()` return an array of data types that this filter needs in order to define the filter. Please note, in the current release, the `CustomFilterEditor` only supports two values. This method returns an array of one class, or two classes. Or it can return a class array if the Filter expects an array of Objects of the same type, such as

```java
return new Class[][String[].class];
```

The `getConditionString(Locale)` returns the string that will appear in the drop down list. Since it is a localized string, we use a helper method in `AbstractFilter` to get it. We defined an entry like this in filter.properties. The “beginWith” example above will return this entry.

```java
FilterCondition.beginWith.string=begins with
```

You can follow this pattern to define your own properties if you register your own `FilterFactory`.

`CustomFilterEditor` also has `setFilter` and `getFilter` methods. The `getFilter` method will return a filter created from the `FilterFactory` that user selects. The `setFilter` method will change the `CustomFilterEditor` to select the `FilterFactory` that creates the `Filter`. Please note, only Filter created from `CustomFilterEditor` can be set. Other filters won’t work.

**TableCustomFilterEditor**

`TableCustomFilterEditor` is one level above `CustomFilterEditor`. There are two styles. You can see both below.
As you can see from the screenshots above, CustomFilterEditor is used as a part of the user interface. CustomFilterEditor creates one Filter. TableCustomFilterEditor adds additional controls to create a list of FilterItems and they can be added to FilterableTableModel. To get the list of FilterItems, you can call getFilterItems() method.

Property Pane

In an Object Oriented Design, every object is composed of a combination of data and function. In Java, we sometimes refer to this data as the ‘properties’ of the object. If you follow the JavaBean pattern then all properties are exposed via getter and setter methods. If an application needs to deal with an object’s properties then this can be done by displaying the name of each property, along with its value. This is the purpose of the PropertyPane, which displays this information in table form.

Below are two examples of Property Panes, from NetBeans and JBuilder respectively. Both graphs show the properties of a JButton. As you can see, the JButton has many properties, such as its background color, foreground color, font, icon, text etc. As you can see, it’s quite intuitive to display them in a table like this with the property name on the left side and the corresponding value on the right side.
What does a PropertyPane look like?

The picture below shows an example of a property pane, using our PropertyPane class. The PropertyPane consists of three main parts. The top portion is a toolbar that has buttons which provide convenient access to some features of the PropertyPane. The middle portion is the PropertyTable, which displays the name of each property, along with its value. The bottom portion is the description area, which can be used to provide a more detailed description of each property.

Since the name of each property is usually very concise, the description area can very helpful (especially for new users). However, the description area can be hidden when the user becomes familiar with the properties.
The **PropertyTable** is a two-column table, the first column of which is the name of the property and the second column is the value of the property. If you wish, you can group sets of properties into **categories**, where each category appears as gray bold text, as shown in the example above. You can collapse categories, which you are not interested in, so that only properties you are interested in will be shown. You can also have different levels of properties, as shown in the last row in the example above.

If you have a large number of properties, which makes it hard to find a specific entry, then you can click on the alphabetic button in the **PropertyPane** toolbar, so that the properties will be listed in alphabetic order.
As a user, how do I use it?

When you click the mouse on the name column, the row will be selected and the value column will go into editing mode automatically. The type of editor that is displayed may vary depending on the type of the data that is being edited. There are basically three types of editors.

TextField editor – For very simple types such as name, number etc.

Dropdown Popup – Rather than letting the user type free-format text, this uses a popup to help the user select the required value. Usually the popup only needs a single selecting action. Examples include the Color and Date input editor.

Dialog Popup – If the popup needs multiple selection actions then you should consider using a Dialog Popup rather than a Dropdown Popup. In addition, you should use the Dialog Popup if there is an existing dialog that you can leverage. Examples include Font, File and Multiple Line Description.

TextField editor is very simple and so will not be discussed any further.

Below is an example of a Dropdown Popup, for selecting a color, using a combo-box-like editor. If you click on the drop down button then the popup will be displayed.
You can also choose a value without going through the popup. Just type in the value you want in the text field…

…and then press enter. You can see the value is set as you entered it. You should see that the converter module has been used here to convert the string “255, 0, 255” into a Color value.

Below is an example of a Dialog Popup. Instead of the down arrow button that is used in a Dropdown popup, a “…“ button is displayed. Clicking on this will cause dialog to appear to help you select a value.

The example below is a file chooser - clicking on the “…“ will cause a FileChooser dialog to pop up.

As a developer, how do I use it?

If you are following the standard object bean pattern then it is quite easy to use the PropertyTable control. However, in the real world, not all objects are compliant with the standard bean pattern. Furthermore, in many cases we find that we are dealing with a property which does not exist as an attribute of any single object, but which is instead calculated ‘on the fly’, based on the values of a number of attributes. Based on our experience of dealing with this sort of situation, we created a class called Property - not surprisingly, it is an abstract class.

Property defines the following methods as abstract, so you need to write your own Property class that extends Property and implement these three methods:

```java
public abstract void setValue(Object value);
public abstract Object getValue();
public abstract boolean hasValue();
```
Things become much easier once you have created a concrete descendant of the *Property* class. Then, all you need to do is to create a list of your *Property* objects, as an *ArrayList* and use this *ArrayList* to create a *PropertyTableModel*.

```
ArrayList list = new ArrayList();
Property property = new .... // create property
list.add(property);

// add more properties

PropertyTableModel model = new PropertyTableModel(list);
PropertyTable table = new PropertyTable(model);
```
BeanProperty and BeanIntrospector

The Property class provides an abstract implementation. If you want to use PropertyTable to display JavaBean properties, it will be too tedious to create a list of properties manually. To address this requirement, we introduced a concrete class called BeanProperty. BeanProperty implements Property on top of PropertyDescriptor which is part of JDK JavaBean implementation.

We introduced another class call BeanIntrospector to help you introspect JavaBean. BeanIntrospector, will create a list of properties which can be used by PropertyTableModel.

There are four different ways to create a BeanIntrospector.

1. Given an object, if it's fully JavaBean compatible and there is corresponding BeanInfo class, you can use BeanIntrospector(Class clazz, BeanInfo beanInfo) to create a BeanIntrospector.

2. If you know exactly the list of property names of the object but you don't want to create a BeanInfo, you can use BeanIntrospector(Class clazz, String[] properties) and give introspector the information of the properties. The array of properties is in the format of

   ```java
   new String[] {
       "propertyName1", "propertyDescription1", "propertyCategory1",
       "propertyName2", "propertyDescription2", "propertyCategory2",
       ...
   };
   ```

   So if you have n properties, the array length should be have 3*n. This array will tell BeanIntrospector what the properties are as well as the description and category which are both part of Property.

3. In this case, you know exactly the list of property names, just like in case 2. However you don't want to hard code the property name into source code. We provide a way to load the property definition from XML file. The XML file looks like this.

   ```xml
   <Properties>
   <Property name="..." displayName="..." description="..." category="..."/>
   <Property name="..." displayName="..." description="..." category="..."/>
   ...
   </Properties>
   ```
Possible attributes for Property element in the XML are "name", "displayName", "value", "type", "description", "dependingProperties", "category", "converterContext", "editorContext", "editable", and "autoIntrospect".

The XML file could be at any place where the code can access. However we suggest you put it at the same package of the object class definition and copy it as resource file to class output directory so that you can use class loader to load it. There are two constructors for this case, `BeanIntrospector(Class clazz, String fileName)` and `BeanIntrospector(Class clazz, InputStream in)`. The first one used to load property XML as a file. The second one is to load it using class loader.

4. The last case is you want to completely depend on Java reflection to find all properties. You can use constructor `BeanIntrospector(Class clazz)` in this case. If using this way, it will find a lot more properties than you need. Typically you don’t want to use this way. However it is not a bad idea to use this in development phase to get the complete list of properties, and then go through each one and determine which ones you want. In production phase, you should use the first three ways and only show the properties you want.

For example, if you use reflection to introspect `Rectangle.class`, you will get 18 properties. Most of them are useless. See the first picture below. In fact, you just need to have a string array as below.

```java
public static final String[] RECTANGLE_PROPERTIES = new String[]{
    "x", "x", "",
    "y", "y", "",
    "width", "width", "",
    "height", "height", ""
};
```

And use the second way to create the introspector, and you got the property table like in the second picture.
After you create the BeanIntrospector, you can further customize it by calling `getProperty(String name)` to get the BeanProperty and then call `setConverterContext()` and `setEditorContext()` etc.

BeanIntrospector has a method called `createPropertyTableModel(Object object)`. It will create a PropertyTableModel that can be used by `PropertyTable`.

Now let’s go through a real example to create and configure BeanIntrospector to inspect the properties of DockableFrame.

First, let’s use constructor `BeanIntrospector(DockableFrame.class)` to create an introspector first, then call `createPropertyTableModel` and set the model on `PropertyTable`. Here is what you will see.
There are more than 140 properties when we tried the first time. It’s too many if we show all of them so we just take part of them as an example and show it above. After looking through each of them, we come up with a list of properties we want to expose. Then we create a String array of the properties we want and use constructor `BeanIntrospector(Class clazz, String[] properties)` to create an introspector. See below. As you can see, there are only 22 properties. And they are properly categorized.
It's much better but there are still a few issues. For example, `initMode` property is shown as a regular integer cell. Yes, the type of the property is indeed `int.class` but the valid values are defined in `DockContext` such as hidden, floating, docked or autohidden etc. It will be very unusable if let you remember the mapping. It's very easy to solve. All you need is to create an `EnumConverter` that maps correctly from `int` value to a string value as well as corresponding `EnumCellRenderer/Editor`.

```java
EnumConverter dockModeConverter = new EnumConverter("DockMode-DockableFrame", int.class,
    new Object[]{
        new Integer(DockContext.STATE_HIDDEN),
        new Integer(DockContext.STATE_FLOATING),
        new Integer(DockContext.STATE_AUTOHIDE),
        new Integer(DockContext.STATE_AUTOHIDE_SHOWING),
        new Integer(DockContext.STATE_FRAMEDOCKED)
    },
    new String[]{
        "Hidden",
        "Floating",
        "Docked",
        "AutoHide",
        "AutoHideShowing",
        "FramedDocked"
    });
```
"Autohidden",
"Autohidden (showing)",
"Docked"},
    new Integer(DockContext.STATE_HIDDEN));
    ObjectConverterManager.registerConverter(dockModeConverter.getType(),
dockModeConverter, dockModeConverter.getConverter());
    EnumCellRenderer dockModeRenderer = new EnumCellRenderer(dockModeConverter);
    EnumCellEditor dockModeEditor = new EnumCellEditor(dockModeConverter);
    CellRendererManager.registerRenderer(dockModeConverter.getType(),
dockModeRenderer, dockModeRenderer.getRenderer());
    CellEditorManager.registerEditor(dockModeConverter.getType(), dockModeEditor,
dockModeEditor.getEditor());

    DOCKABLE_FRAME_INTROSPECTOR.getProperty("InitMode").setConverterContext(dockModeConverter.getConverter());
    DOCKABLE_FRAME_INTROSPECTOR.getProperty("InitMode").setEditorContext(dockModeEditor.getEditor());

The same thing applies initSide property.

The next thing we noticed is the undockedBounds property. The type is Rectangle.class. Since we have RectangleConverter, the rectangle value is converted to a semi-colon delimited string. The values are in the order of x, y, width and height. If user types in that format, it will be converted to rectangle. However it’s not user-friendly. A better approach is to make undockedBounds expandable so that the value of x, y, width, height can be modified in child properties. To make it happen, you need to call the following.

    DOCKABLE_FRAME_INTROSPECTOR.getProperty(DockableFrame.PROPERTY_UNDOCKED_BOUNDS).setAutoIntrospect(true);

After a few customizations, we got the screen shot on your right. As you can see, initMode and initSide properties are now combobox. User can choose the meaningful value directly from the list. The undockedBounds property has child properties so that user can change the x, y, width or height directly.

We provide many built-in cell editors in JIDE Grids product as you will see in the following sections. However there is no way we can cover all your need. When you have a custom data type, you may need to write your own cell editor/cell
renderer/converter in order to allow user editing that certain type in PropertyTable.

Color Related Components

ColorChooserPanel

ColorChooserPanel is a panel that has many color buttons that the user can click on to select the required color. This class supports the ItemListener. An itemStateChanged event will be fired whenever a color is selected.

We support several color sets, including the 15 basic RGB colors, 40 basic colors and 215 web colors.

![Figure 22 ColorChooserPanel (15 colors)](image)

![Figure 23 ColorChooserPanel (40 colors)](image)

![Figure 24 ColorChooserPanel (215 web-safe colors)](image)

In additional to color choosers, we also support gray scale - from 16 gray scales, 102 gray scales and 256 gray scales.
Figure 25 ColorChooserPanel (16 gray scale)

Figure 26 ColorChooserPanel (102 gray scales)

Color(Ex)ComboBox

Unsurprisingly, the Color(Ex)ComboBox is a combo box that can choose colors. It uses ColorChooserPanel as dropdown popup, as shown below:

Figure 27 ColorComboBox

Keyboard Support

Color(Ex)ComboBox and ColorChooserPanel supports keyboard-only environment.

<table>
<thead>
<tr>
<th></th>
<th>When Popup is hidden</th>
<th>When Popup is visible</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALT + DOWN</td>
<td>To Bring up the popup</td>
<td>Hide the popup without changing the selection</td>
</tr>
<tr>
<td>LEFT</td>
<td>Previous color to the same row. If at the beginning of a row, go to last color of previous row</td>
<td></td>
</tr>
</tbody>
</table>
RIGHT | Next color to the same row. If at the end of a row, go to first color of next row.
---|---
UP | Color at the same column of the previous row
DOWN | Color at the same column of the next row
HOME | First color
END | Last color
ENTER | Select the highlighted color and hide popup

**Date Related Component**

**DateChooserPanel**

Similarly to the ColorChooserPanel, the DateChooserPanel is also a popup panel, which allows the user to choose a Date value (again, providing ItemListener events, as appropriate).

![Figure 28 DateChooserPanel / Choosing Year](image)

Figure 28 DateChooserPanel / Choosing Year
Figure 29 DateChooserPanel (Choosing Month)

Figure 30 DateChooserPanel
(hide week of year panel, “Today” and “None” button which are all optional)

```java
DateChooserPanel dateChooserPanel = new DateChooserPanel();
dateChooserPanel setShowWeekNumbers(true/false);
dateChooserPanel setShowTodayButton (true/false);
dateChooserPanel setShowNoneButton(true/false);
```
// create a DateModel first
DefaultDateModel model = new DefaultDateModel();

// setMaxDate of DateModel to Aug 13, 2003
Calendar calendar = Calendar.getInstance();
calendar.set(Calendar.YEAR, 2003);
calendar.set(Calendar.MONTH, Calendar.AUGUST);
calendar.set(Calendar.DAY_OF_MONTH, 13);
model.setMaxDate(calendar);

// create DateChooserPanel with that model.
DateChooserPanel dateChooserPanel = new DateChooserPanel(model);

// create a DateModel first
DefaultDateModel model = new DefaultDateModel();

// add DateFilter to allow WEEKDAY_ONLY

model.addDateFilter(DefaultDateModel.WEEKDAY_ONLY);

// create DateChooserPanel with that model.
DateChooserPanel dateChooserPanel = new DateChooserPanel(model);

Figure 33 DateChooserPanel (Localized Version de)

*DateChooserPanel* supports multiple selections. There are three selection modes. The first is *SINGLE_SELECTION*, meaning only one date can be selected at one time. This is the default mode. Since JIDE v1.9.1 release, we introduced two new selection modes - *SINGLE_INTERVAL* and *MULTIPLE_INTERVAL*. *SINGLE_INTERVAL* is perfect to choose a data range. See a screenshot below.

Figure 34 SINGLE_INTERVAL Selection Mode

*MULTIPLE_INTERVAL* mode allows user to choose multiple data ranges. See a screenshot below. You can use this mode to display things like all holidays within current year, an employee’s vacation days etc.
User can use keyboard or mouse with the help of CTRL (or Command key on Mac OS X) and SHITF key to do multiple selections. The way to use it is the same as using multiple selections in JList.

When `DateChooserPanel` is in single selection mode, you can use `setSelectedDate()` and `getSelectedDate()` to set and get the selection. However in multiple selection mode (including both single interval or multiple interval), you need `getSelectionModel()` to get the `DateSelectionModel` first, then call `getSelectedDates()` or `setSelectedDates(Date[] dates)` or other methods to get or change the selections.

There is also a view-only mode `DateChooserPanel`, as shown here. In view-only mode, although `setDisplayMonth()` can be used to select which month you want to display. There is no next month or previous month button so the user will not be able to change it. User cannot select any date either.

You can also customize the date label, day of week label, and the month and year labels to show some special effect. To do so, simply create a class extending `DateChooserPanel`, overriding the appropriate methods to get the visual effect that you want.

To the right is a simple example that makes each cell bigger, grays the weekends, and shows an example icon on the “Today” date.
want to the labels. For example you could look up in a database and find out if there are any historic events on that date and add a special icon if so (perhaps adding a MouseListener to trigger some other behavior).

**Date(Ex)ComboBox**

A Date(Ex)ComboBox is a combo box that can choose a date, using a DateChooserPanel as a dropdown popup.

![Figure 38 DateComboBox](image)

### Keyboard Support

*Date(Ex)ComboBox* and *DateChooserPanel* supports a keyboard-only environment.

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>When Popup is hidden</strong></td>
<td></td>
</tr>
<tr>
<td>ALT + DOWN</td>
<td>To Bring up the popup</td>
</tr>
<tr>
<td><strong>When Popup is visible</strong></td>
<td></td>
</tr>
<tr>
<td>ESC</td>
<td>Hide the popup without changing the selection</td>
</tr>
<tr>
<td>LEFT</td>
<td>Previous day of the selected day</td>
</tr>
<tr>
<td>RIGHT</td>
<td>Next day of the selected day</td>
</tr>
<tr>
<td>UP</td>
<td>Same day of the last week</td>
</tr>
<tr>
<td>DOWN</td>
<td>Same day of the next week</td>
</tr>
<tr>
<td>HOME</td>
<td>First day of this month</td>
</tr>
<tr>
<td>END</td>
<td>Last day of this month</td>
</tr>
<tr>
<td>PAGE_UP</td>
<td>Same day of the last month</td>
</tr>
</tbody>
</table>
### CalendarViewer

<table>
<thead>
<tr>
<th>Action</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAGE_DOWN</td>
<td>Same day of the next month</td>
</tr>
<tr>
<td>CTRL + PAGE_UP</td>
<td>Same day of the last year</td>
</tr>
<tr>
<td>CTRL + PAGE_DOWN</td>
<td>Same day of the next year</td>
</tr>
<tr>
<td>ENTER</td>
<td>Select the highlighted date and hide popup</td>
</tr>
</tbody>
</table>

CalendarViewer uses several DateChooserPanels to create a Calendar view. You can choose how many months you want to view. It also supports multiple selections just like DateChooserPanel. CalendarViewer uses DateSelectionModel to keep track of selection, which is same as DateChooserPanel.

Figure 39 CalendarViewer of 12 month view with multiple selection
ComboBox Extension

JComboBox is a very useful component. However, it is very limited as you can only use it to choose a value from a JList. People always want a more customizable dropdown. To solve this limitation, we tried two different approaches. Due the limitation of Swing itself, each approach has its own pros and cons. The table below shows a comparison overview.

<table>
<thead>
<tr>
<th></th>
<th>AbstractComboBox</th>
<th>ExComboBox</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Visual Appearance</strong></td>
<td>Similar to JComboBox but not always the same.</td>
<td>Always the same as JComboBox</td>
</tr>
<tr>
<td><strong>Built-in support for L&amp;Fs</strong></td>
<td>Most L&amp;Fs even some 3rd parties L&amp;Fs but doesn’t look identical to the JComboBox on those L&amp;Fs; not look good on Aqua L&amp;F</td>
<td>Out-of-box support for Windows, Metal, Aqua, Synthetica etc. L&amp;F perfectly. For other 3rd party L&amp;F, UI class needs to be implemented. Once implemented, it will support perfectly</td>
</tr>
<tr>
<td><strong>Extensible</strong></td>
<td>Yes</td>
<td>Yes but not as great as AbstractComboBox</td>
</tr>
<tr>
<td><strong>Compatibility</strong></td>
<td>Incompatible with JComboBox although most methods on JComboBox are also available on it</td>
<td>Extend JComboBox so it is fully compatible with JComboBox</td>
</tr>
</tbody>
</table>

AbstractComboBox basically creates a brand new component and tried to mimic what JComboBox does. It doesn’t extend JComboBox. Since it is a new component, we literally can do whatever we want to it. For example, you can choose your own button for the combobox’s drop down button. You can have your own editor for the editor part of the combobox. However, it is very difficult to keep the same look and behavior as JComboBox. We managed to do it for Windows L&F and some other L&Fs but for Aqua L&F, it is almost impossible to make it look identical to Aqua’s JComboBox.

ExComboBox took a different approach. It extends JComboBox and provides several UI classes extend BasicComboBoxUI, WindowsComboBoxUI, AquaComboBoxUI respectively to provide the additional features we want. Since the same UI is used, we get the same look and behavior as JComboBox. The main downside is you have to subclass the corresponding ComboBoxUI for that L&F in order to support the new L&F. It is also not easy to customize the editor area as AbstractComboBox can.

A ComboBox has three parts – a text field editor, a button, and popup that appears when the button is pressed. The AbstractComboBox will allow you to customize all three parts.
ExComboBox only allows you to customize the button and the popup. Since in most cases, the popup is the main part that needs customization, both approaches will work.

Both AbstractComboBox and ExComboBox have these two methods that you can implement or override when extending.

```java
/**
 * Subclass should implement this method to create the actual popup component.
 * @return the popup component
 */
abstract public PopupPanel createPopupComponent();

/**
 * Subclass should override this method to create the actual button component.
 * @return the button component
 */
public AbstractButton createButtonComponent() {
    return null;
}
```

AbstractComboBox has an addition method.

```java
/**
 * Subclass should implement this method to create the actual editor component.
 * @return the editor component
 */
public abstract EditorComponent createEditorComponent();
```

The ComboBox’s dropdown popup is handled by PopupPanel. All dropdown popup must extend PopupPanel. For JComboBox, we create List(Ex)ComboBox which uses ListChooserPanel (extending PopupPanel) as the popup. Likewise, in the case of Color(Ex)ComboBox, it’s a ColorChooserPanel. Usually people use popup to select things, so the PopupPanel contains common methods for all pop ups, such as knowing how to select an item and how to fire an item event when the selection changes. Please note that although PopupPanel is usually used as a DropDown popup, it can also be used in a dialog (a FileChooserPanel for example). In the other word, when clicking on the dropdown button on a combobox, a dialog pops up. Note also that you do have the choice of using DROPDOWN and DIALOG when using AbstractComboBox or ExComboBox.
The Button part of ComboBox is used to trigger the popup. By default, if it is a drop down popup then we use a button with a down arrow. Conversely, if it is a dialog popup then we use a button with “…” as the text of the button, or if you want, you can use “open” as the text. However in the default cases, we don’t change the button because it doesn’t look very consistent on some L&Fs if changing the text or the look.

For AbstractComboBox, the EditorComponent is the text field editor (actually a JPanel). If you replace this with a JTextField then it becomes normal JComboBox. Although you can create your own EditorComponent you must make sure that it extends an EditorComponent and that it implements getValue and setValue (so that AbstractComboBox knows how to set and get values).

Below shows an example code of a custom ComboBox – StringArrayComboBox.

```java
public class StringArrayComboBox extends AbstractComboBox {
    public StringArrayComboBox() {
        super(DIALOG);
        setType(String[].class);
        initComponent(); // it is very important to call this method in your constructor
    }

    @Override
    public EditorComponent createEditorComponent() {
        if (isEditable()) {
            return new DefaultTextFieldEditorComponent(getType());
        } else {
            return new DefaultRendererComponent(getType());
        }
    }

    @Override
    public PopupPanel createPopupComponent() {
        return new StringArrayPopupPanel(Resource.getResourceBundle(Locale.getDefault()).getString("ComboBox.stringArrayTitle"));
    }
}
```

If you want to an ExComboBox, it is even simpler.

```java
public class StringArrayExComboBox extends ExComboBox {
}
```
public StringArrayExComboBox() {
    super(DIALOG);
    setType(String[].class);
}

@Override
public PopupPanel createPopupComponent() {
    return new StringArrayPopupPanel(Resource.getResourceBundle(Locale.getDefault()).getString("ComboBox.stringArrayTitle"));
}

The main difference is you don't need to createEditorComponent in ExComboBox because, as it mentioned before, ExComboBox doesn't allow you to change editor component.

Both classes use the same StringArrayPopupPanel. This is the main place you need to create a panel that knows how to edit a String array. In our case, we used a simple JTextArea to do it. See below.

```
/**
 * A popup panel for String array.
 */
public class StringArrayPopupPanel extends PopupPanel {
    JTextArea _textArea = new JTextArea();

    public StringArrayPopupPanel() {
        this("");
    }

    public StringArrayPopupPanel(String title) {
        JScrollPane scrollPane = new JScrollPane(_textArea);
        scrollPane.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
        scrollPane.setAutoscrolls(true);
        scrollPane.setPreferredSize(new Dimension(300, 200));
        setBorder(BorderFactory.createEmptyBorder(10, 5, 5, 5));
        setDefaultCloseOperation(new BorderLayout());
        add(scrollPane, BorderLayout.CENTER);
        setTitle(title);
        setDefaultCloseOperation(_textArea);
```
Available ExComboBoxes

See below for a list of available ExComboBoxes. If we categorized them by the component on the popup, there are

1. JList (ListExComboBox or MultiSelectListExComboBox which supports multiple selects)
2. JTable (TableExComboBox)
3. JTree (TreeExComboBox).
4. CheckBoxList (CheckBoxListExComboBox)

If we categorized by the data type of the combo box’s selected item, they are

1. Date.class (DateExComboBox or MonthExComboBox)
2. Color.class (ColorExComboBox)
3. Font.class (FontExComboBox)
4. Insets.class (InsetsExComboBox)
5. File.class (FileChooserExComboBox)
6. String.class but the String is a file name (FileNameChooserExComboBox)
7. String[].class - String Array (StringArrayExComboBox)
8. String.class but it is a multiple line string (MultilineStringExComboBox)

See below for a screenshot of a Tree(Ex)ComboBox.

Tree(Ex)ComboBox uses JTree as the component on the popup. The usage of TreeComboBox is very intuitive. There are only two points that worth mentioning.
First, different from JList, not all tree nodes in a tree are valid selection. In many cases, you only want the leaf node to be selected. So Tree(Ex)ComboBox allows you to add your own code to determine if a TreePath is valid selection. All you need to do is to overwrite isValidSelection(TreePath path). By default, it always returns true, meaning all tree nodes are valid selections. If you want only leaf node to be selectable, you can write something like this in subclass.

```java
protected boolean isValidSelection(TreePath path) {
    TreeNode treeNode = (TreeNode) path.getLastPathComponent();
    return treeNode.isLeaf();
}
```

The second point is the conversion from TreePath to String. You need to provide an algorithm to convert from the selected TreePath to a string so that the string can be displayed in the text field of ComboBox. The algorithm is in a method called convertElementToString(). Below is the default implementation of this method in TreeComboBox. You can see it simply uses toString to convert the tree node to string. Subclass can overwrite this method to do your own conversion. For example, you can convert the selected path in the screenshot above to “JTree -> sports -> football” if it makes sense in your application.

```java
protected String convertElementToString(Object object) {
    if (object instanceof TreePath) {
        Object treeNode = ((TreePath) object).getLastPathComponent();
        return treeNode.toString();
    } else if (object != null) {
        return object.toString();
    } else {
        return "";
    }
}
```
More Screenshots. They are *FontExComboBox*, *MonthExComboBox*, *InsetsExComboBox*, *CheckBoxListExComboBox* and *TableExComboBox* respectively.

Supporting 3rd Party L&F for ExComboBox

ExComboBox has its own Combobox UI that extends the corresponding ComboBoxUI of the L&F. For example we have WindowsExComboBoxUI extends WindowsComboBoxUI. That’s how it supports Windows L&F. We also have AquaExComboBoxUI, MetalExComboBoxUI, SynthExComboBox etc. to support other JDK L&Fs. However, if you want to use ExComboBox with other 3rd party L&Fs, you need to provide a UI for it.

I included the full source code for PlasticExComboBoxUI below as an example. Then make sure you call `UIManager.put("ExComboBoxUI", "com.jidesoft.plaf.plastic.PlasticExComboBoxUI")` after you set the Plastic L&F. You may need to tweak to the source code if the UI class you are extending override some methods we also override but it shouldn’t be too hard to figure out.

```java
package com.jidesoft.plaf.plastic;

import com.jgoodies.looks.plastic.PlasticComboBoxUI;
import com.jidesoft.combobox.ExComboBox;
import com.jidesoft.combobox.PopupPanel;
import com.jidesoft.plaf.ExComboBoxUI;
import com.jidesoft.plaf.basic.ExComboBoxEditor;
import com.jidesoft.plaf.basic.ExComboBoxFocusListener;
```
import com.jidesoft.plaf.basic.ExComboBoxPopup;
import com.jidesoft.plaf.basic.ExComboBoxRenderer;
import javax.swing.*;
import javax.swing.plaf.ComponentUI;
import javax.swing.plaf.basic.ComboPopup;
import java.awt.event.FocusListener;

public class PlasticExComboBoxUI extends PlasticComboBoxUI implements ExComboBoxUI {
    public static ComponentUI createUI(JComponent c) {
        return new PlasticExComboBoxUI();
    }

    protected ExComboBox _comboBox;

    @Override
    public void installUI(JComponent c) {
        if (c instanceof ExComboBox) {
            _comboBox = ((ExComboBox) c);
        }
        super.installUI(c);
    }

    @Override
    protected JButton createArrowButton() {
        if (_comboBox != null) {
            AbstractButton button = _comboBox.createButtonComponent();
            if (button instanceof JButton) {
                return (JButton) button;
            }
        }
        JButton button = super.createArrowButton();
        comboBox.putClientProperty("doNotCancelPopup", button.getClientProperty("doNotCancelPopup");
        return button;
    }

    @Override
    protected ComboPopup createPopup() {
    }
}

@override
protected ComboBox createPopup() {
    }

@override
protected JButton createArrowButton() {
    if (_comboBox != null) {
        AbstractButton button = _comboBox.createButtonComponent();
        if (button instanceof JButton) {
            return (JButton) button;
        }
    }
    JButton button = super.createArrowButton();
    comboBox.putClientProperty("doNotCancelPopup", button.getClientProperty("doNotCancelPopup");
    return button;
}

@override
protected ComboPopup createPopup() {

How to create your own Cell Renderer and Cell Editor

In this section, we will use FontNameCellEditor as an example to illustrate how to create your own cell renderer and cell editor that is compatible with the rest of JIDE Grids.
First of all, we need to decide what the type of this property is. In the case of font face name, the type is String. However not all strings are valid font face name. That’s why we need a converter for it. See below. In `fromString()` method, we enumerate all available font names in current environment. If the String is one of the known font names, we return the String. Or else, return null because the String is not valid font name.

```java
public class FontNameConverter implements ObjectConverter {
    /**
     * ConverterContext for a font name.
     */
    public static ConverterContext CONTEXT = new ConverterContext("FontName");

    public String toString(Object object, ConverterContext context) {
        if (object == null || !(object instanceof String)) {
            return null;
        } else {
            return (String) object;
        }
    }

    public boolean supportToString(Object object, ConverterContext context) {
        return true;
    }

    public Object fromString(String string, ConverterContext context) {
        if (string.length() == 0) {
            return null;
        } else {
            String[] font_names = GraphicsEnvironment.getLocalGraphicsEnvironment().getAvailableFontFamilyNames();
            for (int i = 0; i < font_names.length; i++) { // check font if it is available
                String font_name = font_names[i];
                if (font_name.equals(string)) {
                    return string;
                }
            }
            return null;
        }
    }
}
```
public boolean supportFromString(String string, ConverterContext context) {
    return true;
}

Next, we need to create a cell editor. We use ListComboBoxCellEditor as the base class. Please note, you can also extend AbstractComboBoxCellEditor or TextFieldCellEditor, depending what you need from the base class.

/**
 * CellEditor for FontFace.
 */
public class FontNameCellEditor extends ListComboBoxCellEditor {

    public final static EditorContext CONTEXT = new EditorContext("FontName");

    /**
     * Creates FontNameCellEditor.
     */
    public FontNameCellEditor() {
        super(new FontNameComboBoxModel());
    }

    /**
     * Model for the font style drop down.
     */
    private static class FontNameComboBoxModel extends AbstractListModel
        implements ComboBoxModel {

        /** An array of the names of all the available fonts. */
        private String[] _fontNames = null;

        /** The currently selected item. */
        private Object _selectedFontName;

        /**
         * Create a custom data model for a JComboBox.
         */
protected FontNameComboBoxModel() {
    _fontNames = GraphicsEnvironment.getLocalGraphicsEnvironment().getAvailableFontFamilyNames();
}

public void setSelectedItem(Object selection) {
    this._selectedFontName = selection;
    fireContentsChanged(this, -1, -1);
}

/**
 * Chooses a Font from the available list.
 * @param font The font to make current
 */
public void setSelectedFont(Font font) {
    for (int i = 0; i < _fontNames.length; i++) {
        if (font.getFontName().equals(_fontNames[i])) {
            setSelectedItem(getElementAt(i));
        }
    }
    fireContentsChanged(this, -1, -1);
}

public Object getSelectedItem() {
    return _selectedFontName;
}

public int getSize() {
    return _fontNames.length;
}

public Object getElementAt(int index) {
    return _fontNames[index];
}
}
Please note, in both converter and cell editor, we have a context object. We need them because the type is String type which has been taken to register StringConverter and StringCellEditor. We will need the context object to register with our own converter and cell editor with ObjectConverterManager and CellEditorManager. So here is the last thing you need to do.

```java
ObjectConverterManager.registerConverter(String.class, new FontNameConverter(), FontNameConverter.CONTEXT);
CellEditorManager.registerEditor(String.class, new FontNameCellEditor(), FontNameCellEditor.CONTEXT);
```

To try it out, you just need to create a Property which type is `String.class`, converter context is `FontNameConverter.CONTEXT` and editor context is `FontNameCellEditor.CONTEXT`. If you add this property to `PropertyTableModel`, `PropertyTable` will automatically use `FontNameCellEditor` to edit this Property and use `FontNameConverter` to validate the font name.

Below is another example of custom CellEditor. It continues from the `StringArrayComboBox` example above. Now we will make it a CellEditor.

```java
/**
 * A String array cell editor.
 */
public class StringArrayCellEditor extends ContextSensitiveCellEditor implements TableCellEditor {

    private StringArrayComboBox _comboBox;

    /**
     * Creates a FileNameCellEditor.
     */
    public StringArrayCellEditor() {
        _comboBox = new StringArrayComboBox();
        _comboBox.setBorder(BorderFactory.createEmptyBorder());
    }

    /**
     * Gets the value of the cell editor.
     *
     * @return the value in this cell editor.
     */
    public Object getCellEditorValue() {
        _comboBox.setSelectedItem(_comboBox.getEditor().getItem());
        return _comboBox.getSelectedItem();
    }
}
```
Hierarchical Table

*HierarchicalTable* is special type of table which can display any components hierarchically inside the table itself. (*TreeTable* can also display hierarchical information, but the information in *TreeTable* is limited to all rows being the same format. In *HierarchicalTable*, you can display any components.)

See below for three examples of hierarchical tables. They look quite different from each other, but they are all done using *HierarchicalTable*. 
From the screenshots, you probably are surprised by what *HierarchicalTable* can do. The child component of each row is not limited to table. It could be anything. That’s the power of *HierarchicalTable*.

There are actual three tables in the second example. Each has different background colors. The top level has four rows; each row is a JIDE product. You can further expand each row to display components or features in that product. Furthermore, for each features, you can see further details. Now three tables are organized in a hierarchical way, this is also why it is called *HierarchicalTable*.

The last one is the famous user interface of Windows Control Panel’s Add/Remove Program dialog. People ask us how to do it but didn’t realize *HierarchicalTable* provides build-in support for. The trick is the child component will cover the parent row if it returns false in `isHierarchical(int row)` method of *HierarchicalTableModel*.

**HierarchicalTableModel**

There are only two major classes in *HierarchicalTable* component. One is very obvious, the *HierarchicalTable*. The other one is called *HierarchicalTableModel*. *HierarchicalTableModel* is an
interface but it doesn’t extend `TableModel`. We named it `XxxTableModel` because we expect you always use it with either `TableModel`, or `AbstractTableModel` or `DefaultTableModel`. There are three methods in this interface; see below.

```java
/**
   * Checks if the row has child.
   *
   * @param row the row index
   * @return true if the row has child. Otherwise false.
   */
   boolean hasChild(int row);

/**
   * Returns true if the child component will be shown in a new row. Returns false is the child
   * component will replace the current row.
   *
   * @param row
   * @return true if the child component will be shown in a new row. Otherwise false which means the child
   * component will replace the current row when displaying.
   */
   boolean isHierarchical(int row);

/**
   * Gets the child object value associated with the row at row index.
   *
   * @param row
   * @return the value. If child components are the same except the displaying is different, it’s better to
   * return a value using this method. Then the component factory can create the same component and only
   * load a different value to that component. You can also return null in this method. If so,
   * the component factory can look up in the table model on fly and find out how to create the child component.
   */
   Object getChildValueAt(int row);

/**
   * Returns true if the row is expandable. This only makes sense when hasChild() return true. If there is child and
   */
```
* but not expandable, you will see a gray "+" icon but click on it does nothing.
* 
* @param row
* @return true if the row is expandable.
*/

    boolean isExpandable(int row);

Here are two typical usages of HierarchicalTableModel.

    class ProductTableModel extends DefaultTableModel implements HierarchicalTableModel {
        ......
    }

    or

    class ProductTableModel extends AbstractTableModel implements HierarchicalTableModel {
        ......
    }

So in addition to implement methods defined in TableModel, you also need to implement the three methods defined in HierarchicalTableModel.

HierarchicalTable

As you probably noticed, HierarchicalTableModel only returns a value and doesn’t have code to create a component to display in HierarchicalTable. We added a new method to HierarchicalTable called setComponentFactory. You can use this method to associate a HierarchicalTableComponentFactory object with HierarchicalTable. It’s HierarchicalTableComponentFactory’s responsibility to create components and destroy components.

    HierarchicalTable extends SortableTable, which indicates our HierarchicalTable can be sorted. In case you don’t want it to be sortable, just call setSortable(false).

    In addition to methods defined in JTable and SortableTable, HierarchicalTable defined several of its own methods.
HierarchicalTable has one special column called the hierarchical column. If a column is the hierarchical column, there will be an expand/collapse icon on the left side of any cells in that column. Clicking on that button will expand/collapse. By default, the first column is the hierarchical column. If you want to control when to expand/collapse all by yourself, you can call setHierarchicalColumn(-1). For example, at the screenshot below, the hierarchical column is 1 (the 2nd column since the column index is starting from 0).

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JIDE Encoding Framework</td>
<td>Swing-based GUI framework which enables drag-n-drop dockable windows</td>
</tr>
<tr>
<td>JIDE Components</td>
<td>Collection of useful components</td>
</tr>
<tr>
<td>JIDE Grids</td>
<td>Collection of JTable related components</td>
</tr>
</tbody>
</table>

Figure 42 HierarchicalTable when the hierarchical column is 1

There is another attribute called singleExpansion. Typically when user expands a row, and then expands another row, the first expanded row will remain expanded. However there are case user wants only one row to be expanded at a time. The attribute singleExpansion can be used in this case. True means only one row can expanded at a time. Default is false.

Container for Child Component

Among the four methods of HierarchicalTableModel, the most important method is getChildValueAt(int row). Basically for each row, you can associate a value with it. This value can be used in HierarchicalTableComponentFactory to create a child component for each row when the row is expanded. How you write this component is completely up to you. As usual, we created several classes you can leverage. They are HierarchicalPanel and TreeLikeHierarchicalPanel.

HierarchicalPanel is a special JPanel which will always respect the preferred height of the component it contains. So instead of returning the child component directly in getComponent(int row), wrap it in HierarchicalPanel and return the HierarchicalPanel. That way, when the child component height changes, HierarchicalPanel’s height will adjust to the new height automatically. This may be a little hard to understand, but if you try to expand a child component, one with HierarchicalPanel and the other with regular JPanel, you will notice the difference.

HierarchicalPanel also has a border of BorderFactory.createEmptyBorder(0, 14, 1, 0), which will match with the margin of the expand/collapse icon when the expandable column is 0. However you can pass in any border to HierarchicalPanel.

To make HierarchicalTable more like a tree, we also prepared TreeLikeHierarchicalPanel. It’ll draw a hash line on the left margin, just like in JTree. See the red circled area in the left screenshot. You can tell from the name of TreeLikeHierarchicalPanel that it extends HierarchicalPanel. If you plan to create your own style of HierarchicalPanel, we suggest you also make it extend HierarchicalPanel.
Maintaining single selection

When the child components in HierarchicalTable are also tables, sometimes you want to keep one selection at a time. However since those are different table components and have their own ListSelectionModels, you will end up with each table has its own row selected. To address this issue, we created a class called ListSelectionModelGroup. The name comes from ButtonGroup which keeps an exclusive selection for a group of buttons. ListSelectionModelGroup will keep exclusive selection for a group of ListSelectionModel. To add a ListSelectionModel to the group, you just call add() method. After the ListSelectionModel is gone, call remove() method. This matches perfectly with HierarchicalTable's HierarchicalTableComponentFactory's createChildComponent() and destroyChildComponent() methods. Basically, when a child component is created and if it's a table, call group.add(table.getSelectionModel()). When the table is destroyed, call group.remove(table.getSelectionModel()). G8.1 HierarchicalTableDemo is an example to show you how to use this.

Migration from Hierarchical Table Beta version

If you use HierarchicalTable for the first time, you can skip this section. If you used HierarchicalTable beta version, you should read this to find out how to migrate from your old code to the new one.

The main changes are in interface HierarchicalTableModel. Here are the details of interface changes.

<table>
<thead>
<tr>
<th>Old method</th>
<th>New method</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean hasChildComponent(int row)</td>
<td>boolean hasChild(int row)</td>
</tr>
<tr>
<td>boolean isHierarchical(int row)</td>
<td>the same</td>
</tr>
<tr>
<td>Component getChildComponent(int row)</td>
<td>Object getChildValueAt(int row)</td>
</tr>
</tbody>
</table>

The reason for this change is to separate the model from view. The old interface gave people the impression that we are mixing model and view. So the new interface is only about data. To convert the data to a component, you need to use HierarchicalTableComponentFactory defined on HierarchicalTable.

HierarchicalTableComponentFactory has two methods

```java
Component createChildComponent(HierarchicalTable table, Object value, int row);
void destroyChildComponent(HierarchicalTable table, Component component, int row);
```

For the createChildComponent() method, the value argument is the value that's returned from the getChildValueAt() method in HierarchicalTableModel. The row is the same row as in getChildValueAt(int row). You have two choices here: You can either always return null in
getChildValueAt(int row) method and completely let createChildComponent() figure out how to create the component. Or you can return a value in the getChildValueAt(int row) method and let createChildComponent() create the component, and then associate the component with the value. If you're migrating from beta version, it's easier to use the first way. You can simply return null from getChildValueAt(int row), and pretty much copy the old code in the getChildComponent(int row) method and put it into createChildComponent(). However, if your child components are using the same component with different data, you might think about changing to the second way. For example, if your child component is always a table, you can return a table model in getChildValueAt(int row) - and in createChildComponent(), just set the table model to the table. You can even use a table pool to contain all instances of table and reuse them.
TreeTable

*TreeTable* is another kind of table component that can display hierarchical data using a table.

*TreeTable* is a combination of a tree and a table -- a component capable of expanding and collapsing rows, as well as showing multiple columns of the same row. Tree and table work on different data structure. The former is used to display hierarchical data. The latter is used to display a flat tabular data. So in order to make the data works in *TreeTable*, we need to “hierarchize” the flat tabular data. To make the design simple, we make the following assumption

*If the data can be used in TreeTable, it must be row-oriented.*

Row-oriented means one dimension of the two dimension tabular data can be represented as *row*. Then each row will provide data for the second dimension. Based on this assumption, we introduce several concepts (they are all Interfaces) to make *TreeTable* possible.

- **Node**: represent a node in tree data structure
- **Expandable**: represent something can be expanded such as tree node with children.
- **Row**: represent a row in table data structure – it is also the first dimension of the two dimension tabular table data.
- **ExpandableRow**: represent a row that can have children rows.

It will be simple to understand this using an example. A typical *TreeTable* example is the file system. In file system, there are files and folders. Using the concepts above, each file will be a *Row*; each folder will be an *ExpandableRow*. Folder (the *ExpandableRow*) can have children which can be either files (the *Row*) or other folders (the *ExpandableRow*). See screenshot of a *TreeTable*.

![Figure 43 FileSystem TreeTable](image-url)
The table model used by TreeTable is TreeTableModel. It extends AbstractTableModel. However it's essentially a list of ExpandableRows or Rows. TreeTableModel is an abstract class. The only method you has to implement is getColumnCount(). You might want to override getColumnName(int). Otherwise, it will use "A", "B" etc as default names. You also need to make the columns match with the value you returned in Row#getValueAt(int). Please be noted that JIDE will create a root by default in TreeTableModel to make sure the update on the rows you passed into TreeTableModel will be reflected in UI by firing table model events. You may override createRoot() to create your own root rows although it is actually not visible.

Once we create a java.util.List of the Rows, you call

```java
TreeTableModel model = new MyTreeTableModel(list);
TreeTable table = new TreeTable(model);
```

Now you get a TreeTable instance.

TreeTable uses a special cell renderer on the first column to paint +/- icon as well as tree line. If the row is expandable, you will see +/- icon. Clicking on it will expand the row. Clicking again will collapse it. You can also use keyboard to expand/collapse a row. Right arrow key will expand the selected row and left arrow key will collapse it. If the selected row doesn’t have children or it is collapsed already, it will change the selection to its parent. If you try it, you will find it is very convenient to navigate in the tree table using arrow keys.

We used the same +/- icons used in a regular JTree so it will change based on different LookAndFeels. The tree lines can be turn on or off using setShowTreeLines(boolean). The line color can be set by setTreeLineColor(Color). By default, it will use the JTree’s tree line color which is UIManager.getColor("Tree.hash").

Please note, in our TreeTable design, there are not usage of JTree or TreeModel or TreeNode as defined in Swing. TreeTable extends JTable eventually and it has nothing to do with JTree.

**Comparison between TreeTable and HierarchicalTable**

Many people wonder when they should use TreeTable and when to use HierarchicalTable. The following table will help you to understand and make you decision.

<table>
<thead>
<tr>
<th></th>
<th>HierarchicalTable</th>
<th>TreeTable</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data format of top level</strong></td>
<td>A regular tabular data just like JTable</td>
<td>A list of rows with hierarchical information built in the data</td>
</tr>
<tr>
<td><strong>Data format of child levels</strong></td>
<td>Any data.</td>
<td>Same data as the top level data</td>
</tr>
<tr>
<td><strong>Nested components</strong></td>
<td>Any components</td>
<td>There are no nested components in tree table. They are part of the TreeTable.</td>
</tr>
<tr>
<td><strong>Scalability</strong></td>
<td>Bad in the sense that each children component is a real</td>
<td>Good. There is only one table no matter how many nested</td>
</tr>
</tbody>
</table>
component. So if you have a table with 100 rows all expanded with a children table, you will have 101 table instances.

| Level of hierarchies (exclude the top level) | One. However you can archive many levels by nested HierarchicalTable inside another HierarchicalTable. | As many as you want |

From the table above, you can see which table to use is totally depending on what kind of data you have. If the data on each child level can all be represent as table, it’s better to use TreeTable. If not, you should use HierarchicalTable.
**GroupTable**

*GroupTable* is a special *TreeTable* that can group rows who has the same value in certain column into a group. It is an implementation that is very similar to the table used in Microsoft Outlook Inbox.

**GroupableTableModel**

*GroupableTableModel*, just like *ContextSensitiveTableModel* is an interface that you can implement it at any table model. It only has one method called *getGrouperContext()*). We use the same design pattern here as *ContextSensitiveTableModel*. Just like *ContextSensitiveTableModel* uses *CellEditorManager* and *CellRendererManager* to find the cell renderers and cell editors, *GroupableTableModel* uses *ObjectGrouperManager* to find the *ObjectGrouper* which will be used to group values.

Please note *GroupableTableModel* is totally optional. If the values in the table do not need any special value grouping, you don’t need to implement this interface.

**DefaultGroupTableModel**

As we mentioned earlier, the *GroupableTableModel* is just a table model that can be grouped. The one who actually does the grouping is the *DefaultGroupTableModel*. *DefaultGroupTableModel* is a *TreeTableModel* that implements *TableModelWrapper*. It wraps any *TableModel* or *GroupableTableModel* if special value grouping is needed.

*DefaultGroupTableModel*’s constructor takes any table model. You can then call *addGroupColumn* to add the column that you want to group or *setGroupColumns* to set all group columns at once. When you change the group columns, you call *groupAndRefresh* to push the change to the screen.

Now let’s look at an example. There is a *JTable*. It has five columns. If you look at carefully, you will see the value in source and destination column repeats. In this case, it would make sense if we group by those two columns.

<table>
<thead>
<tr>
<th>ID</th>
<th>Time</th>
<th>Source</th>
<th>Destination</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12:00</td>
<td>A</td>
<td>X</td>
<td>descriptions</td>
</tr>
<tr>
<td>2</td>
<td>1:12</td>
<td>B</td>
<td>Z</td>
<td>descriptions</td>
</tr>
<tr>
<td>1</td>
<td>1:23</td>
<td>A</td>
<td>Y</td>
<td>descriptions</td>
</tr>
<tr>
<td>3</td>
<td>3:00</td>
<td>A</td>
<td>Z</td>
<td>descriptions</td>
</tr>
<tr>
<td>2</td>
<td>4:05</td>
<td>A</td>
<td>Z</td>
<td>descriptions</td>
</tr>
<tr>
<td>6</td>
<td>2:00</td>
<td>B</td>
<td>Y</td>
<td>descriptions</td>
</tr>
<tr>
<td>3</td>
<td>1:54</td>
<td>A</td>
<td>Y</td>
<td>descriptions</td>
</tr>
<tr>
<td>3</td>
<td>2:05</td>
<td>C</td>
<td>X</td>
<td>descriptions</td>
</tr>
</tbody>
</table>

So you do something like this.

```java
DefaultGroupTableModel groupTableModel = new DefaultGroupTableModel(tableModel);
groupTableModel.addColumn(2);
```
groupTableModel.addGroupColumn(3);
groupTableModel.groupAndRefresh();

Here is the result.

As you can see, there is actually a TreeTable. It will group by the source column first. Then understand each group, it will group by destination column. You can independently expand or collapse each group. Comparing with the flat table originally, this hierarchical view obviously offers more information to the user. For example, your user can easily see how many items are from source A, B and C.

We also have an option to flat down the group level. See the screenshot below. We still have two group-by columns but we flat it down into one level so there are fewer hierarchies.

In the example, we didn’t use GroupableTableModel because there is no need to group the values. See below for another example where you use an ObjectGrouper to group the value. It’s
a product table which has a sales column. The sales column is a dollar amount. We want to

group them into several groups.

First, we need an ObjectGrouper. If the value is less than $100, we put them into group 1. If
less than $1000 but greater than $100, we put into group 2. And so on.

```java
private static class SalesObjectGrouper extends DefaultObjectGrouper {
    public final static GrouperContext CONTEXT = new GrouperContext("Sales");

    public Object getValue(Object value) {
        if (value instanceof Number) {
            double v = ((Number) value).doubleValue();
            if (v < 100) {
                return new Integer(0);
            } else if (v < 1000) {
                return new Integer(1);
            } else if (v < 10000) {
                return new Integer(2);
            } else {
                return new Integer(3);
            }
        }
        return null;
    }

    public Class getType() {
        return int.class;
    }

    public ConverterContext getConverterContext() {
        return SalesConverter.CONTEXT;
    }
}
```

We certainly don’t want to display 0, 1, 2 and 3 to users so we need an ObjectConverter to
convert the integer to some meaningful texts.

```java
private static class SalesConverter extends DefaultObjectConverter {
```

96
public static ConverterContext CONTEXT = new ConverterContext("Sales");

public String toString(Object object, ConverterContext context) {
    if (object instanceof Integer) {
        int value = ((Integer) object).intValue();
        switch (value) {
            case 0:
                return "From 0 to 100";
            case 1:
                return "From 100 to 1000";
            case 2:
                return "From 1000 to 10000";
            case 3:
                return "Greater than 10000";
        }
    }
    return null;
}

public boolean supportFromString(String string, ConverterContext context) {
    return false;
}
}

Now if you apply DefaultGroupTableModel to this GroupableTableModel, you will get something like this screenshot. As you can see, we used the ObjectGrouper to decide the group and we even display the group nicely using the converter.

You can optional use GroupTableHeader so that user can drag the columns to decide how to group the table. See below.

Figure 44 Draggable GroupTableHeader
GroupList

GroupList is a JList that supports the grouping of items. When there is a large number of items in a JList and you need to organize them in several categories, this is where you can use GroupList. Each group header appears slightly different from other regular rows as it uses a different cell renderer you can set by yourself.

GroupableListModel

The most important part of GroupList is the GroupableListModel. GroupableListModel extends ListModel and adds the following methods.

- Object getGroupAt(int index);
- Object[] getGroups();
- void addListGroupChangeListener(ListGroupChangeListener listener);
- void removeListGroupChangeListener(ListGroupChangeListener listener);

The getGroupAt(int index) method provides the group information for each row in the list model. It returns an object that identifies the group. It is normally a String. However, you can use any object type as long as you have a cell renderer knows how to renderer it on UI. All rows that belong to the same group should return the same object instance or those objects equal (based on equals method).

The getGroups() method return a list of groups you ever return in getGroupAt method. The main purpose of this method is to allow you to specify the order of the groups. You could return less groups than they are actually used in getGroupAt(). In this case, the groups that are not defined in getGroups() will be appended at the end of the list.

On top of GroupableListModel, we have two implementations - AbstractGroupableListModel and DefaultGroupableListModel. The abstract version just implements the support for ListGroupChangeListener and leave everything else related to grouping to you. The default one extends DefaultListModel and add method such as setGroupAt, setGroups(), renameGroup() etc methods allowing you to change the group information. If you already have a DefaultListModel, you can replace it with DefaultGroupableListModel without any code change. Then later on, you call setters to add group information. If you were using AbstractListModel, you can also replace it with AbstractGroupableListModel without any code change. However, you still need to override getGroupAt() and getGroups() in order to have any groups.
**LayoutOrientation**

*JList* supports three layout orientations. They are

- **VERTICAL**: a vertical layout of cells, in a single column
- **VERTICAL_WRAP**: a "newspaper style" layout with cells flowing vertically then horizontally when it reaches the list height.
- **HORIZONTAL_WRAP**: a "newspaper style" layout with cells flowing horizontally then vertically when it reaches the list width.

*GroupList* supports all three layout orientations. However, since *GroupList* brings the concept of grouping, it has to change the meaning of the layout orientation. First of all, the group rows are not part of the layout. It always covers the whole list width. The layout of cells only applies to the cell within the same group. In the case of VERTICAL layout, there isn’t much difference with or without grouping. But in the case of two wrap layouts, it is quite different. We had to introduce one property to *GroupList* called `preferredColumnCount`. In horizontal wrap mode, we will wrap the cells when it reaches the `preferredColumnCount`. In the case of vertical wrap, it will calculate items in each group and find out how many rows it will need for each group and wrap when it reaches the row count vertically. For example, if there are 20 items in the group and the preferred column count is 8, 20 divided 8 is 2 remaining 4, which means it needs at least 3 rows to display all the cells. So we will layout three cells vertically and then wrap and so on.

Careful readers might notice *JList* misses the fourth layout – HORIZONTAL. The reason I guess is because the *JList* will be way too wide (although I can still argue that it could be useful in some cases). In the case of *GroupList*, it is less a problem to support it as all items are divided into groups so that each group has fewer items. So we added HORIZONTAL layout orientation into *GroupList*.

- **HORIZONTAL**: a horizontal layout of cells, in a single row with in the same group.

As the group rows are different from regular rows, we provide `setGroupCellRenderer` method to allow you to set your own cell renderer. The other method is `setGroupCellSelectable`. In most cases, user doesn’t want to select the group cells. But if you want to select them, they can always call `setGroupCellSelectable` to make the group cell selectable.
TableScrollPane

TableScrollPane is a component that supports table row header, row footer, column header, and column footer. In fact, you can implement row header even without this component. You basically create a separate table and call setRowHeaderView() to set it to JScrollPane. Now you get a table with a row header. Sounds simple? Unfortunately it’s more than that. What you can’t do with JScrollPane is that it doesn’t row footer and column footer. You also needed to create two separate table models – one for row header and the other for the table itself – which might be inconvenience. Of course, you also need to deal with keystrokes. As they are two tables, keeping pressing left arrow or tab key won’t navigate from row header to the table. Don’t forget the sorting. Sorting row header won’t change the sort order of the table because, again, they are two tables. In fact, there are a lot of things you need to do in order to support row header. TableScrollPane is such a component that takes care of all those for you.

To simplify the usage, we also create an interface MultiTableModel to replace TableModel. Instead of using AbstractTableModel or DefaultTableModel, you can use AbstractMultiTableModel or DefaultMultiTableModel respectively. Interface MultiTableModel has a method called getColumnType(int column)\(^7\). By returning HEADER_COLUMN or FOOTER_COLUMN or REGULAR_COLUMN, you can use one table model to define a table which has header columns and footer columns.

Please see screenshot below for a typical use case of TableScrollPane. As you can see there are frozen rows and columns. The first row and the last row are frozen. The first column and the last two columns are frozen.

<table>
<thead>
<tr>
<th>Year</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Sales Total</th>
<th>After-tax Profits Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>1,150,650.00</td>
<td>1,107,677.00</td>
<td>1,340,084.00</td>
<td>1,128,387.00</td>
<td>77,113.00</td>
<td>94,101.00</td>
</tr>
<tr>
<td>2004</td>
<td>1,205,999.00</td>
<td>1,222,402.00</td>
<td>1,429,677.00</td>
<td>1,284,235.00</td>
<td>77,181.00</td>
<td>85,246.00</td>
</tr>
<tr>
<td>2009</td>
<td>1,246,204.00</td>
<td>1,236,253.00</td>
<td>1,329,953.00</td>
<td>3,353,937.00</td>
<td>35,516.00</td>
<td>49,946.00</td>
</tr>
<tr>
<td>2005</td>
<td>1,300,210.00</td>
<td>1,282,931.00</td>
<td>1,777,656.00</td>
<td>1,412,399.00</td>
<td>92,180.00</td>
<td>97,277.00</td>
</tr>
<tr>
<td>2010</td>
<td>1,396,695.00</td>
<td>1,450,235.00</td>
<td>1,449,317.00</td>
<td>2,468,965.00</td>
<td>110,294.00</td>
<td>101,779.00</td>
</tr>
<tr>
<td>2006</td>
<td>1,495,940.00</td>
<td>1,544,570.00</td>
<td>1,632,517.00</td>
<td>1,423,149.00</td>
<td>122,263.00</td>
<td>112,593.00</td>
</tr>
<tr>
<td>2007</td>
<td>1,596,490.00</td>
<td>1,494,628.00</td>
<td>1,521,355.00</td>
<td>1,385,511.00</td>
<td>120,294.00</td>
<td>120,566.00</td>
</tr>
<tr>
<td>2008</td>
<td>1,625,679.00</td>
<td>1,607,343.00</td>
<td>1,644,269.00</td>
<td>1,595,211.00</td>
<td>120,211.00</td>
<td>72,245.00</td>
</tr>
</tbody>
</table>

Figure 45 TableScrollPane

---

\(^7\) The interface has another method called getTableIndex(). In case you noticed it, this method is for TableSplitPane, the next component we will discuss. To use in TableScrollPane, you just need to return 0 in that method.
The whole timesheet table is based on one MultiTableModel. To determine which column is row header or row footer, you just need to return different values in getColumnType. See below.

```java
public int getColumnType(int column) {
    if (column < 4) {
        return HEADER_COLUMN;
    }
    else if (column < HEADER.length - 4) {
        return FOOTER_COLUMN;
    }
    else {
        return REGULAR_COLUMN;
    }
}
```

Even though header columns and footer columns are added as separate components, they will work just like in one table. For example, row selection is synchronized. Tab key or left/right arrows will navigate across all three tables. If you scroll vertically, all three tables will scroll together. If you resize columns in any table, even the last column of the header column table and the first column of footer column table. If you choose to have sortable table, sorting in one table will affect other tables. Only one column will be sorted at a time.
TableSplitPane

TableSplitPane is a special component that splits a large table into several smaller tables. There are dividers between tables so each table can be resized independently. However, as just like in TableScrollPane, all tables inside this component act just like one table. In short, this is an ideal candidate if you have a large table model which has a lot of columns and those columns can be divided into different categories.

TableSplitPane is built on top of MultiTableModel too. There is one more method on the MultiTableModel interface called getTableIndex(int column). You can return any integer starting from 0. If the value is 0, the column will in the first table. If 1, it will be in the second table, and so on. TableSplitPane will look at the value returned from getTableIndex() and divider columns into several tables. Then for each table, it will look at the value returned from getColumnType() and divide them into header, regular or footer columns. As a result, you can use one table model to define multiple tables; each can have its own header and footer. See below for an example.

![Figure 46 TableSplitPane](image)

DualList

Features of DualList

![DualList Features](image)
**DualList** is a pane that contains two **JList**s and a bunch of buttons. The list on the left contains all the items. The list on the right contains the items that are selected.

**DualList** uses **DualListModel** which is the model of this component. **DualListModel** is nothing but a **ListModel** with additional methods to keep track of the selected item. If you want to implement a **DualListModel**, you just need to use one of the two **DualListModel**s we already defined - **AbstractDualListModel** or **DefaultDualListModel**. Implementing them is as easy as implementing **AbstractListModel** or **DefaultListModel** because we already implemented the additional methods **DualListModel** adds.

With the help of **DualListModel**, **DualList** supports three kinds of selection modes. This mode controls what to display in the right list when items are selected.

- **REMOVE_SELECTION** mode means the left list will remove the items if they are selected to the right list. This mode can prevent user from selecting the selected item again if it is not allowed in certain cases (see screenshot above on the left).

- **DISABLE_SELECTION** mode means the selected items will be shown as disabled. User cannot select them anymore but they can still see them in the right list to show user a complete unchanged list. (see screenshot above on the right)

- **KEEP_SELECTION** mode will not change the right list at all when items are selected. This is the only mode which can result duplicated items to be selected. If that’s what you want in your situation, this mode is the one you should use.

Here is the list of features that **DualList** supports.

- Powerful selection feature. It allows single selection, multiple selection and multiple interval selection on either list.

- Powerful reorder feature. After selecting the items, you can rearrange them to the order you want though move up and move down actions. You can select multiple items and rearrange them all at once.

- Complete keyboard support.
  
  a. **UP/DOWN/PAGE_UP/PAGE_DOWN** keys to select items in either list.

  b. **LEFT/RIGHT/ENTER** keys to move items between the two lists.

  c. **CTRL-UP/CTRL-DOWN** keys to move selected items up and down in the selected item list. **CTRL-HOME/CTRL-END** keys to move selected items to the top or the bottom.

- Buttons to select and reorder the items. Each button can be shown or hidden independently.

- Complete API support to do everything you can do through the UI.

- Mode support. The same **DualListModel** instance can be set to two or more **DualLists**, and all the **DualLists** will be synchronized because they are using the same model.
Classes, Interfaces and Demos

Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DualList</td>
<td>The main class for this component.</td>
</tr>
<tr>
<td>(com.jidesoft.list)</td>
<td></td>
</tr>
<tr>
<td>DualListModel</td>
<td>The model class for DualList.</td>
</tr>
<tr>
<td>(com.jidesoft.list)</td>
<td></td>
</tr>
<tr>
<td>AbstractDualListModel</td>
<td>Abstract implementation of DualListModel.</td>
</tr>
<tr>
<td>(com.jidesoft.list)</td>
<td></td>
</tr>
<tr>
<td>DefaultDualListModel</td>
<td>Default implementation of DualListModel.</td>
</tr>
<tr>
<td>(com.jidesoft.list)</td>
<td></td>
</tr>
</tbody>
</table>

Demos

<table>
<thead>
<tr>
<th>Demo</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DualListDemo</td>
<td>A demo to demonstrate the DualList. It selected from a list of countries.</td>
</tr>
<tr>
<td>(examples\G31. DualList)</td>
<td></td>
</tr>
</tbody>
</table>

Code Examples

1. To create a DualList. There is no difference from creating a regular JList.

```java
DualList dualList = new DualList(Object[] or java.util.List);
```

2. The call above will create a AbstractDualListModel internally. To create a DualList with your own DualListModel, do

```java
DualList dualList = new DualList(DualListModel);
```

3. To select a few items using API. The code below will select the 2\textsuperscript{nd} and 4\textsuperscript{th} items from the original list model.

```java
DualListModel model = dualList.getModel();
model.addSelectionInterval(1, 1);
model.addSelectionInterval(3, 3);
```

4. To move all items to the right list. In the other word, select all items.
dualList.moveRight();

5. To get the list of selected items.

Object[] items = dualList.getSelectedValues();

6. To change the selection mode. The selection mode can be REMOVE_SELECTION, KEEP_SELECTION or DISABLE_SELECTION.

dualList.setSelectionMode(selectionMode);
// or
dualList.getModel().setSelectionMode(selectionMode);

7. To show or hide one of the buttons in the middle. The first parameter is the command name. They are all defined in DualList beginning with COMMAND_.

dualList.setButtonVisible(DualList.COMMAND_MOVE_ALL_RIGHT, true or false);

**DualTable**

**Features of DualTable**

DualTable is a pane that contains two JTables and a bunch of buttons. It looks just like DualList except it uses table instead of list. The table on the left contains all the items. The table on the right contains the items that are selected.

DualTable uses some model classes from DualList, for example, DualListModel. It uses TableModelAdapter interface to get multiple values from an item in the DualListModel so that they can be displayed in a table.
With the help of `DualListModel`, `DualTable` supports three kinds of selection modes. This mode controls what to display in the right table when items are selected.

- **REMOVE_SELECTION** mode means the left table will remove the items if they are selected to the right table. This mode can prevent user from selecting the selected item again if it is not allowed in certain cases (see screenshot above on the left).

- **DISABLE_SELECTION** mode means the selected items will be shown as disabled. User cannot select them anymore but they can still see them in the right table to show user a complete unchanged table (see screenshot above on the right).

- **KEEP_SELECTION** mode will not change the right table at all when items are selected. This is the only mode that can result in duplicated items to be selected. If that’s what you want in your situation, this mode is the one you should use.

Here is the list of features that `DualTable` supports.

- **Powerful selection feature.** It allows single selection, multiple selection and multiple interval selection on either list.

- **Powerful reorder feature.** After selecting the items, you can rearrange them to the order you want though move up and move down actions. You can select multiple items and rearrange them all at once.

- **Complete keyboard support.**
  
  a. UP/DOWN/PAGE_UP/PAGE_DOWN keys to select items in either list.
  
  b. LEFT/RIGHT/ENTER keys to move items between the two lists.
  
  c. CTRL-UP/CTRL-DOWN keys to move selected items up and down in the selected item list. CTRL-HOME/CTRL-END keys to move selected items to the top or the bottom.

- **Buttons to select and reorder the items.** Each button can be shown or hidden independently.

- **Complete API support to do everything you can do through the UI.**

- **Mode support.** The same `DualListModel` instance can be set to two or more `DualLists`, and all the `DualLists` will be synchronized because they are using the same model.

### Classes, Interfaces and Demos

<table>
<thead>
<tr>
<th>Classes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DualTable</td>
<td>The main class for this component.</td>
</tr>
<tr>
<td>(com.jidesoft.grid)</td>
<td></td>
</tr>
<tr>
<td>DualListModel</td>
<td>The model class for <code>DualList</code>, the same as in <code>DualList</code></td>
</tr>
<tr>
<td>(com.jidesoft.list)</td>
<td></td>
</tr>
<tr>
<td>AbstractDualListModel</td>
<td>Abstract implementation of <code>DualListModel</code>.</td>
</tr>
</tbody>
</table>
DefaultDualListModel
(com.jidesoft.list)
Default implementation of DualListModel.

TableModelAdapter
An interface which can adapt a list model to a table model.

Demos
DualTableDemo
(examples\G31. DualList)
A demo to demonstrate the DualTable. It selected from a table of tasks.

Code Examples
8. To create a DualTable.

```
DualTable dualTable = new DualTable(Object[] or java.util.List, a TableModelAdapter);
```

9. The call above will create an AbstractDualListModel internally. To create a DualTable with your own DualListModel, do

```
DualTable dualTable = new DualTable (DualListModel, a TableModelAdapter);
```

10. To select a few items using API. The code below will select the 2\textsuperscript{nd} and 4\textsuperscript{th} items from the original list model.

```
DualListModel model = dualTable.getModel();
model.addSelectionInterval(1, 1);
model.addSelectionInterval(3, 3);
```

11. To move all items to the right table. In the other word, select all items.

```
dualTable.moveRight();
```

12. To get the list of selected items.

```
Object[] items = dualTable.getSelectedValues();
```
13. To change the selection mode. The selection mode can be REMOVE_SELECTION, KEEP_SELECTION or DISABLE_SELECTION.

```java
dualTable.setSelectionMode(selectionMode);
```
or

```java
dualTable.getModel().setSelectionMode(selectionMode);
```

14. To show or hide one of the buttons in the middle. The first parameter is the command name. They are all defined in DualList beginning with COMMAND_.

```java
dualTable.setButtonVisible(DualTable.COMMAND_MOVE_ALL_RIGHT, true or false);
```

**TextFieldList**

**Features of TextFieldList**

`TextFieldList` belongs to the same family of `DualList` and `DualTable` which transfers object from one component to a list. It contains a `JTextField` on the left and a `JList` on the right and a bunch of buttons. The customer can input new items in the `JTextField` on the left and press the transfer button to move it to the list on the right.

You could override `isInputValid(String)` to control which kind of inputs are allowed to be transferred to the list.
Magin Area

Magin is a thin vertical stripe that can be added next to any component that can be tall. For example, any component with a vertical scroll bar such as JTable, JList, JTree, JTextArea, CodeEditor etc. Usually the content in margin area will scroll vertically along with the content in the component. A typical example is line/row number margin.

MarginArea is the container for margins. You can place a MarginArea to the left hand side of a tall component then add many margin components into this MarginArea using addMarginComponent(Margin magin).

MarginSupport and RowMarginSupport

There are common things to JTable, JList, JTree, JTextArea and CodeEditor. All of them could be very tall and thus need a vertical scroll bar. This is the reason a margin could be useful. All of them also have a row concept. That’s why they can use row-based margins. In order to provide the support for all of them without referring to the specific component in the margin code, we introduced two interfaces called MarginSupport and RowMarginSupport. RowMarginSupport extends MarginSupport. Further more, we created an implementation of RowMarginSupport for each component. See below.

Create your own margin

All margins must implement an interface called Margin. We provide AbstractMargin which implements most of the methods in Margin. We also provide AbstractRowMargin to make the implementation of row-based margin easier.

Margin should paint itself completely instead using any child components. You don’t want to add child components to margin mainly because the margin will scroll along with the text in code editor and is usually very tall. You don’t want to deal with any child components or layout manager for this particular component. Taking line number margin for example: say we have a code editor has 1000 lines and 100 lines are visible in the view port, you of course can create 1000 JLabels and add them to the margin. Or you can paint the 100 strings on fly just for the visible 100 lines. Obviously the second approach is much more efficient. Margin has paintMargin method which you must implement in order to paint the margin.
There are two categories of margins – row margin or non-row margin. Sometimes we refer row as line which is more suitable for the case of CodeEditor or TextArea. But row or line are essentially basically the same thing.

The row margin paints its content row by row. A typical example in this category is the row or line number margin. Each painting code will just paint the rectangle area belong to that row. In this case, we created AbstractRowMargin to make it simple. You need to implement the paintRowMargin method in its subclass. Internally, we implemented the paintMargin method on Margin interface and delegate to paintRowMargin to paint line by line. As you can see below, all you need to do is to paint the content for a particular line at the specified rectangle. You should never paint it outside the rectangular area.

```java
public void paintRowMargin(Graphics g, Rectangle rect, int row) {
    int lineNumber = ((RowMarginSupport)_marginSupport).visualRowToActualRow(row) + 1;
    if (lineNumber <= ((RowMarginSupport)_marginSupport).getRowCount()) {
        String s = String.valueOf(lineNumber);
        g.setColor(getForeground());
        g.setFont(getFont());
        g.drawString(s, rect.x + rect.width - getFontMetrics(getFont()).stringWidth(s) - 3,
                    rect.y + (rect.height + getFontMetrics(getFont()).getAscent() -
                         getFontMetrics(getFont()).getDescent()) / 2);
    }
}
```

There are many margins in this category. Breakpoint margin and bookmark margin are two more such examples.

The non-row margin is the opposite of the first category. It is not limited by one row but ranges one or several rows. In this case you don’t want to paint row by row but paint it across many row. A typical example is code folding margin in CodeEditor. As the code folding ranges from a start line to an end line, you just paint the whole range in one shot. For the margins in this category, you need to extend AbstractMargin directly and implements the paintMargin method. For performance reason, you should check the first visible line of the code editor and the total visible line count. If you know you will paint outside the visible area, just don’t paint.

**RowNumberMargin**

Now let’s cover look at an example. Here is the `RowNumberMargin`. `RowNumberMargin` extends `AbstractRowMargin` as you may expect.
Here is the code to create the UI above.

```java
RowMarginSupport marginSupport = new ListRowMarginSupport(list, listScrollPane);
MarginArea marginArea = new MarginArea();
marginArea.addMarginComponent(new RowNumberMargin(marginSupport));
```

The code is very simple. All we did is to create a `RowMarginSupport`, in this case, it is a `ListRowMarginSupport` as the component is a `JList`. Then we create a `RowNumberMargin` that takes any `RowMarginSupport`. Because of this `RowMarginSupport` interface, we can easily extend it to support many other components that have vertical scroll bar.

**MarginPainter and LineMarginPainter**

At the beginning, you most likely use one margin for one purpose. But sooner or later, you will find out that’s a waste of screen space. You will start to think if you can combine several margins that are not very busy into one margin. See screenshot below for an example. What you see here is a brace matching margin over the code folding margin. Code folding margin could be very busy but brace matching is not because there is only one brace matching at one time. So it’s a perfect use case to paint the brace matching over the code folding to save some space.

```java
public String getDescription() {
    return "m" + "Drawed classes: 'n" + 
    "CodeEditor";
}
```

Now let’s see how we do this. We added `addMarginPainter` method on `AbstractMargin`. Assuming you will extend `AbstractMargin` in your margin. If so, you can add your own `MarginPainter` to it.

`MarginPainter` is a painter interface which can paint the margin area. This painter is mainly used to add extra content to an existing margin. You can add many `MarginPainters` to `AbstractMargin` in order to decide the order to paint them, each `MarginPainter` has layer index.
The lower the layer index is, the earlier it gets painted. In the other word, the painter has a higher index will overwrite those that have lower index. The default layer index is defined as `LAYER_DEFAULT_INDEX` in `MarginPainter` interface. The original content of the margin is painted on this layer. For example, in code folding margin, code folding information is painted on the layer of index `LAYER_DEFAULT_INDEX`. If you want to your painter painted before code folding information, use a layer index smaller than `LAYER_DEFAULT_INDEX`. If you want it painted after the code folding, use an index larger than `LAYER_DEFAULT_INDEX`.

Here is the code of how to adding a margin painter. The result is what the screenshot above shows.

```java
CodeFoldingMargin margin = new CodeFoldingMargin();
margin.addMarginPainter(new BraceMatchingMarginPainter());
editor.getMarginArea().addMarginComponent(margin);
```

`MarginPainter` is mainly for non-line margin. For line margin, there is `addLineMarginPainter` on `AbstractLineMargin`. It is almost the same as `MarginPainter` except it just paints the rectangle area of a particular line.

**CodeFoldingMargin**

`CodeFoldingMargin` is a special margin that paints the code folding information. We mentioned it several times when we cover the basic of margin. Now let’s say how to use it.

```java
public String getDescription() {
    return "\n    "Demoed classes:\n    "CodeEditor";
}
```

As you can see from the screenshot above, each code folding has a start and an end (the 1st screenshot above). Then there is a line between them to connect them. When it is folded, there is just one icon to indicate where the folding is (the 2nd screenshot above), or when the code folding starts and ends on the same line, you will only see one icon when it either expanded or collapsed (the 3rd and 4th screenshots above).
In order to allow you to customize how the code folding is painted, we added `setCodeFoldingPainter` to allow you set a `CodeFoldingPainter`. `CodeFoldingPainter` has all methods you can implement in order to paint each part of the code folding. Here are a few examples after we set our own customized painter.

![Figure 50 Different CodeFoldingPainter to mimic Eclipse and NetBeans](Image)

**Marker Area**

MarkerArea is a special area on the right side of a tall component, after the vertical scroll bar. See the picture on the right for an example. Since it appears outside the scroll bar, it doesn’t scroll with the content. Both MarkerArea and MarginArea are for tall component that needs vertical scroll bar. However the MarkerArea doesn’t change when scrolling v.s. the MarginArea will change content while scrolling.

The main usage of MarkerArea is to use limit vertical space to display a condense information for this tall component so that user has an overview of what's going on inside this tall component. For example, a table data is constantly updating (such as a table that displays real time data). If the updating rows are not in the viewport, you will miss them. In this case, you can use the MarkerArea to show a color line or color lines (called marker) to indicate a row or some rows are updating. Since the MarkerArea is not scrolling, you can always see the markers. Similar usages are highlights, errors, todo’s. Users not only can see all of the makers of the table in one shot but also can click on the marker on the marker stripe and jump right to the row.

**MarkerSupport**

In order not to refer to a specific component in MarkerArea, we introduced two interfaces `MarkerSupport`, similar to what we did in the margin area. While `AbstractMarkerSupport` simply provides a `MarkerModel` in the implementation, while `AbstractRowMarkerSupport` assumed the component has the same row height for all the rows so that it can implement `indexToPoin` and `pointToIndexRange` methods when giving the row height and the row count. We created an implementation of `MarkerSupport` for the specific component. See below.
Marker and MarkerModel

From data point of view, Marker represents a range of elements in the component and the status associated with the elements. It has a start offset and an end offset. By default, there are two types of markers - error and warning. But you can always define your own types of markers. You can also associate a tool tip with a marker. The tool tip will be shown when user mouse moves over the marker stripe.

MarkerModel is the model class that stores all the markers. You can use this class to add/remove/update markers. The change will be shown on the MarkerArea immediately.

Marker Eye and Marker Stripes

MarkerEye is at the top of the marker area. It indicates the inspecting status. The paint of MarkerEye is done by a class called MarkerEyePainter. By default, DefaultMarkerEyePainter is used. You can always set your own painter by calling setPainter(MarkerEyePainter).

MarkerStripe is a panel below the MarkerEye to display all the markers in a marker model. The paint of each stripe is done by a class called MarkerStripePainter. By default, DefaultMarkerStripePainter is used. You can always set your own painter by calling setPainter(MarkerStripePainter).

MarkerArea

MarkerArea is the component that displays the marker. It accepts any MarkerSupport in the constructor. Internally it will use MarkerEye and MarkerStripe to display the markers.

There are many methods on MarkerArea. Ideally you shouldn’t call methods on MarkerEye and MarkerStripe directly as the MarkerArea should have the methods that delegate to these two sub-components.

Marker has different types. You can register the types with MarkerArea using registerMarkerColor. Please choose colors consistently inside your application. For example, if yellow means changes in your application, register yellow as the marker color for change markers. DefaultMarkerStripePainter will use the registered color to paint the marker. Of course if you set your stripe painters, you can do whatever you want for each marker type.

TableHeaders

JTableHeader is the Swing class for the header of a JTable. In order to support various features in JIDE Grids, we have to enhance the JTableHeader. For example, SortableTable needs to paint a sort arrow onto the header. AutoFilter feature needs a drop down button on the header. Nested table header requires a header that supports multiple rows and the grouping. Then for the GroupTable, we need a header that supports drag-n-drop to group the columns. People also have requirements to show multiple line text or bold/italic text on the header. We gradually added some of the features mentioned above, i.e. using header cell renderer for the sort arrow and drop down button. However it has the limitation because using cell renderer will
break the native look of the header on Windows and Mac OS X. These table headers will look
different from the standard table header. We also couldn’t support both nesting and auto filter
feature, or both drag-n-drop grouping and auto filter feature on one table header. All those
limitations forced us to look into a redesign. Since JIDE 3.x release, we overhauled the design of
table header related features.

**Table Header Hierarchy**

This is the table header hierarchy. It is now a single hierarchy until the last level. As you can
see, *NestedTableHeader* and *GroupTableHeader* is also an *AutoFilterTableHeader*.

**TableHeaderCellDecorator**

Instead of using the cell renderer approach in the old design, we introduce a new approach
which we haven’t seen in any other projects. *TableHeaderCellDecorator* provides a way for users
to paint over the margin of any table header’s cells. You can use it on any table headers that is a
*SortableTableHeader*. You just need to call *addCellDecorator* to add your own cell decorator.
The added order of the cell decorator matters. The first cell decorator that is added will paint
first. Because each cell decorator will reserve a margin from the cell rect for its painting, the
next cell decorator will get a smaller cell rect.

We used this cell decorator for both SortableTable’s sorting arrow as well as the drop down
button for the auto-filter feature.

For example, here is how the default table header looks like on Windows 7. The left one is
the default state and the right one is the rollover state.

<table>
<thead>
<tr>
<th>ProductName</th>
<th>ProductName</th>
</tr>
</thead>
</table>

Our goal of the cell decorator is to keep the look exactly the same as the default one. And
here is what it looks after we use cell decorator to add a filter icon to the header. As you can
see, the look is very consistent.

<table>
<thead>
<tr>
<th>ProductName</th>
<th>ProductName</th>
</tr>
</thead>
</table>

Note that we can also painting both sort arrow and the filter icon which was not possible in
the earlier design. See below.

<table>
<thead>
<tr>
<th>ProductName</th>
<th>ProductName</th>
</tr>
</thead>
</table>
Not only that we are using TableHeaderCellDecorator, but also it is a public API, which means you can also use this feature to decorate your table header.

Please take a look at the screenshot below which were taken under different OS and L&Fs.

![Figure 51 TableHeader under different L&Fs and OS](image)

**StyledLabel Powered TableHeader**

Since JIDE 3.2 release, we optionally use StyledLabel to paint the text for the table header. This is no API changing for this feature. It is hidden behind the column name as defined in `TableModel's getColumnName(int column)`. You can use an annotated string that is available in StyledLabelBuilder. If you return a regular string without any annotation, JLabel is used to paint the cell. But if you use an annotated string that StyledLabel can recognize, StyledLabel is automatically used.

You can add a lot of interesting styles to your header. Here is a show-off example that has absolutely no real meaning. You can make it more meaningful. For example, show a bold text on the header if the column is sorted.

Here is the code for it.

```java
@override
public String getColumnName(int column) {
    switch (column) {
        case 0:
            return "Java\{TM:sp\}";
        case 1:
            return "CO\{2:sb\}";
        case 2:
            return "{Waved:w}";
        case 3:
            return "{Red:f:red} {Blue:f:blue}";
        case 4:
            return "{Red:b:red} {Blue:b:blue}";
    }
    return null;
}
```
case 5:
    return "(Bold:b) (Italic:i)";

case 6:
    return "(Strike:s) (Double-strike:ds)";
}
return "";
}

We also support text wrapping. You just need to enabled it using the global flag feature introduced to StyledLabel. For example, to enable text wrapping on column "this is a long text", you can return "this is a long text@r:2:1:4". Everything after "@" symbol is considered as global flags. You can have multiple flags separated by ",". "r:2:1:4" means it will do row wrapping, by default is 2 rows, minimum 1 rows and maximum 4 rows.

**HeaderStyleModel and CellStyleTableHeader**

Another way to bring style to the header is to use HeaderStyleModel. HeaderStyleModel uses a similar approach as CellStyleModel. You again implement HeaderStyleModel on your table model and you return the CellStyle you want in getHeaderStyleAt method.

Pretty much all styles available in CellStyle is available on CellStyleTableHeader. See below for an screenshot.

![Figure 52 HeaderStyleModel](image)

**TableCellEditorRenderer**

*JTable* uses something called cell renderer and cell editor for cell rendering and editing respectively. For the same cell, the renderer component is usually difference from the editor component. For example, most cell renderer components are JLabel. But to edit a cell, JTextField is usually used as the cell editor component. However, in some cases, maybe we want the same component to be used for both the renderer component and the editor component. That’s why we introduced TableCellEditorRenderer so that you can use the same component for both renderer and editor.

Why do we want the same component use the same component as both editor and renderer? There are at least two reasons.
To better indicate the cell is editable

`JTable` renders the cell using `JLabel` by default. Editable cells have no visual difference from non-editable cells. Users have to click or double click on the cell to figure out if the cell is editable. So some users will prefer to have a different way to render the cell to indicate it is editable. JIDE Grids provides cell style which is one way to make it happen as you can use `CellStyle` to make the editable cells having white background and non-editable cells gray background. Another way is to use this `EditorRenderer` approach and use editors such as JTextField or JComboBox/ExComboBox as the renderer. If users see a drop down button on the cell, they know the cell will be editable without clicking on it.

To interactive with mouse events on the cell level

As you know already, `JTable` paints the cell, not places a real component on the cell. That’s why it was called a cell renderer. When you moves your mouse over a cell, the mouse events will not interactive with the cell renderer component because it is just a painted image. For example, if you use as JButton as cell renderer and JButton, if placed in a JPanel, will a rollover effect, but the rollover effect will not work in `JTable`. Clicking on the button-based cell renderer will not trigger the button action either. Because of this, it was always a challenge to implement hyperlink type of features in `JTable`. The cell editor, on the other hand, is a real component that is placed on the cell. So it will interactive with mouse events. This is a hint. To implement hyperlink, we can use this `EditorRenderer` so that the renderer and the editor use the same hyperlink button. We can add a mouse listener to `JTable`. When JTable is over a cell, we automatically and silently enter cell editing mode. Since the renderer and the editor are the same, user won’t even notice the cell is in editing mode when rollover.

Sounds easy? Right, except there is one minor issue. JTable only allows one cell to be editing at a time. So we start the cell editing automatically, the previous editing cell has to stop editing. It might not be a problem if the table doesn’t contain any editable cells other than hyperlinks. But unfortunately in most cases, it is not true. You certainly don’t want the cell stop editing automatically when user simply moves the mouse. To fix this issue, we introduced a new `rolloverCellAt` method to `JideTable`. This method works just like `editCellAt` except it doesn't involve with the existing cell editing mechanism. When you call `rolloverCellAt`, a cell editor will be created and placed over the cell. However the old cell editor, if it has one, will remain there. This way user can still do cell editing while having mouse interaction over other cells.

Hyperlink in Cell

This is probably one of the most asked features but so far we haven’t seen anyone did it right. There was post at [http://java-swing-tips.blogspot.com/2009/02/hyperlink-in-jtable-cell.html](http://java-swing-tips.blogspot.com/2009/02/hyperlink-in-jtable-cell.html) but it is a very bad example. It calls table.repaint() in mouseMoved method. Is it the reason people think Swing is slow? It is of course very tempting to copy an existing example from web but you really need to cautious about you are copying.

The approach we took for the hyperlink feature is to use both editor component and renderer component. We created a new class called `HyperlinkTableCellEditorRenderer`. It uses JideButton as the hyperlink component. The action listener can be set. We didn’t use `CellRendererManager` in this case as this renderer/editor is probably will only be used for this
particular table because of the customized action listener thing. That's why we simply set it to `TableColumn`. Note that we set it for both the cell renderer and cell editor.

```java
HyperlinkTableCellEditorRenderer renderer = new HyperlinkTableCellEditorRenderer();
renderer.setActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        Object source = e.getSource();
        JOptionPane.showMessageDialog(table, ((JideButton) source).getText() + " is clicked");
    }
});
table.getColumnModel().getColumn(0).setCellRenderer(renderer);
table.getColumnModel().getColumn(0).setCellEditor(renderer);
```

The next thing you need to do is to use a util class called `RolloverTableUtils`. This util has a install method which will add a mouse motion listener to the `JTable`. The listener will automatically force a cell to enter editing mode or the rollover mode when the mouse is over the cell. Once the cell is editing or rollover mode, the component itself will respond to mouse event thus you see the cursor change as well as clicking will trigger the action editing.

```java
RolloverTableUtils.install(table);
```

Here is what it looks like.

![Hyperlink in JTable](Figure 53 Hyperlink in JTable)

This approach has absolutely no performance problem as the link above has. The only extra thing we did is to check if the cell should enter editing mode which is very fast. You can refer to the JavaDoc of `RolloverTableUtils` for more information.

You can also use JButtons in `JTable` that act like real buttons. by using `ButtonTableCellEditorRenderer`. The usage is exactly the same as the hyperlink example. If you want to have multiple buttons in the same cell, that's also covered. The HyperlinkCellDemo has an example of it (refer to the second column in the table above which contains cells with two buttons in each cell).
Internationalization and Localization

We have fully considered i18n and l10n. All strings used by JIDE Grids have been put into properties files under each package. Some users contributed localized version of those files and we put those files inside jide-properties.jar. If you want to support languages other than those we provided, just extract those properties file, translated to the language you want, add the correct postfix and then jar them back into jide-properties jar. You are welcome to send the translated properties file back to us if you want to share it.