

# JIDE Grids Developer Guide

---

## 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 application.

Many people complain about the design of JTable. However in our opinion, 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, so that it can satisfy everybody's needs. However 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* will be fully compatible with JTable. It will not only make your migration to easier but also make it possible to leverage any future improvements to JTable that are made by other people.

---

## PACKAGES

---

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

Packages	Description
com.jidesoft.grid	JTable related components, including PropertyTable, SortableTable, SortableTableModel, and FilterTableModel
com.jidesoft.converter	Converters that can convert from String to Object and from Object to String.
com.jidesoft.comparator	Comparators
com.jidesoft.combobox	Several ComboBox-like components such as DateComboBox and ColorComboBox, as well as classes needed to create your own ComboBox.

---

## CONVERTER

---

Before we introduce the *PropertyGrid*, we have to cover some basic modules that the *PropertyGrid* based on (actually it's not just for *PropertyGrid*, it can be used in other places as well).

The beauty of *PropertyGrid* is that it can use a String to represent any object type. So in this little grid area, a lot of information can be shown. In some cases, it's possible to convert a String to an Object and vice-versa (however there are also some cases where there is no way to convert). To begin, let's explore the cases that convert a String to/from an Object.

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

```
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%? They might think the rectangle's top-left corner is at Point(10, 20) and bottom-right corner at Point(100, 200). Despite this, users can generally learn by experience: as long as you are consistent across your application, users will get used to it.

The situation is 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. What this means is that, based on your user background, you should consider adding more help information if ambiguity may arise.

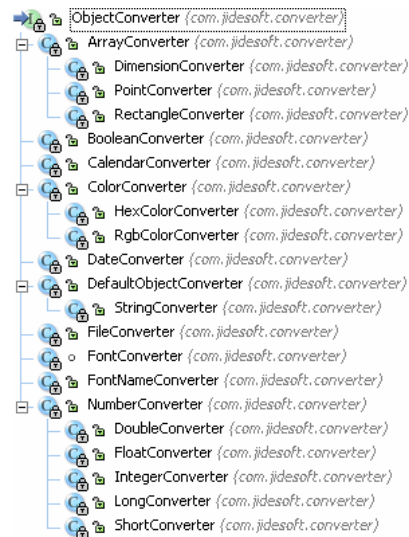
We also need to consider internationalization, since the string representation of any object may be different under different locales.

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 to the right shows all of the converters that we currently provide.

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:

```
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();
```

---

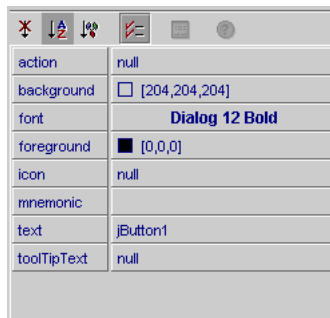
## PROPERTY PANE

---

### Background

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.





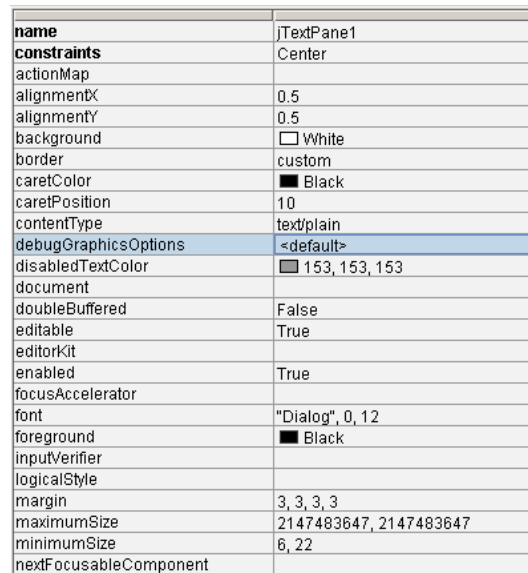
action	null
background	 [204,204,204]
font	<b>Dialog 12 Bold</b>
foreground	 [0,0,0]
icon	null
mnemonic	
text	jButton1
toolTipText	null

Figure 1 NetBeans PropertyPane (Above)







name	JTextPane1
constraints	Center
actionMap	
alignmentX	0.5
alignmentY	0.5
background	 White
border	custom
caretColor	 Black
caretPosition	10
contentType	text/plain
debugGraphicsOptions	<default>
disabledTextColor	 153, 153, 153
document	
doubleBuffered	False
editable	True
editorKit	
enabled	True
focusAccelerator	
font	"Dialog", 0, 12
foreground	 Black
inputVerifier	
logicalStyle	
margin	3, 3, 3, 3
maximumSize	2147483647, 2147483647
minimumSize	6, 22
nextFocusableComponent	

Figure 2 JBuilder 9 PropertyPane (Right)

## 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 *PropertyGrid*, 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

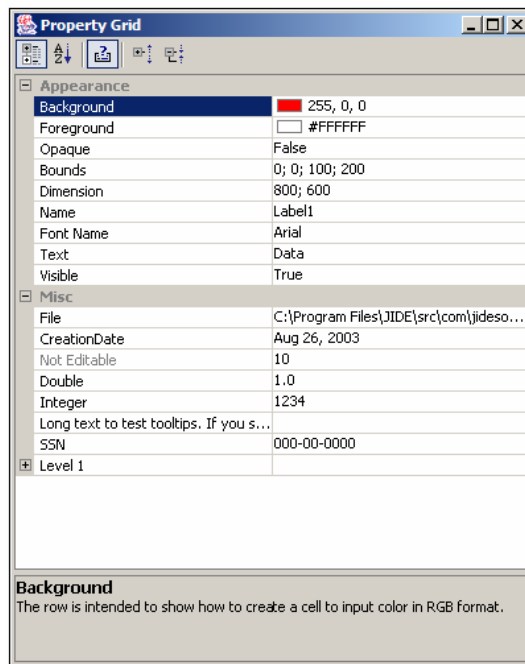
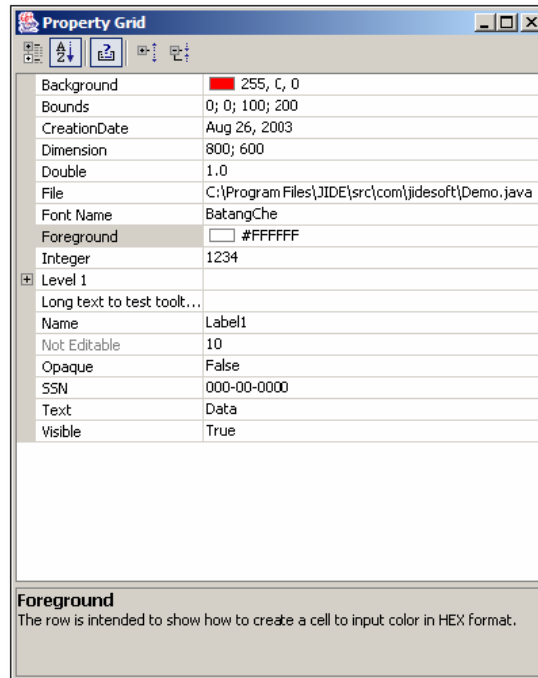


Figure 3 JIDE PropertyPane

The *PropertyGrid* 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.



**Figure 4 PropertyGrid (Alphabetic Order)**

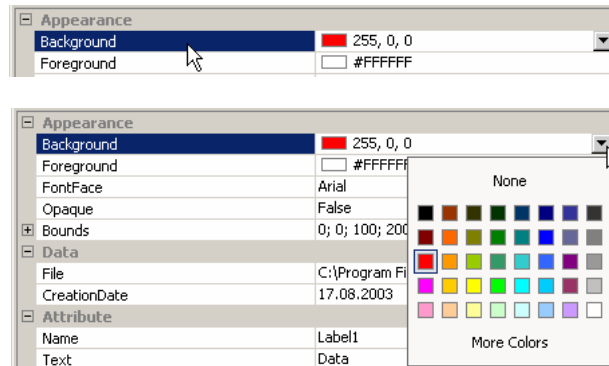
#### **As a user, how do I use it?**

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.

1. TextField editor – For very simple types such as name, number etc.
2. 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.
3. 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.

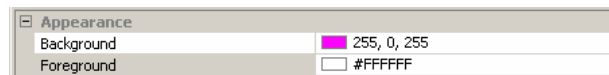


**Figure 5 Dropdown Cell Editor**

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.



**Figure 6 Dialog Cell Editor**

### 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 *PropertyGrid* 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:

```
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);
```

---

## CELL EDITORS AND RENDERERS

---

The *CellEditor* and *CellRenderer* interfaces are probably the most interesting aspects of the *JTable* framework. By default, 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, in the case of *PropertyTable*, neither of these assumptions is true. Fortunately however, *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 an given data type, as defined by its class. You can also register a different cell editor to the same type using different contexts.

```
static void registerEditor(Class clazz, CellEditor editor, EditorContext context);
static void registerEditor(Class clazz, CellEditor editor);
```

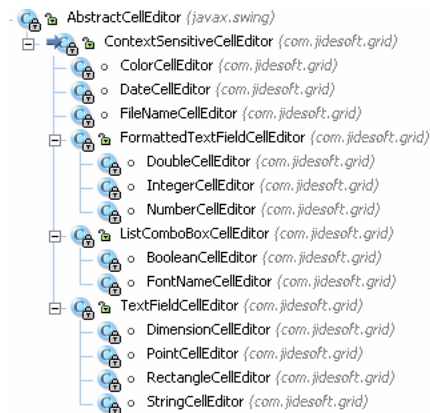


As an example, the `String` class is generally associated with a `StringCellEditor`. However, if the `String` is actually a font name then we associate it with a `FontNameCellEditor` in the context of `FontNameCellEditor.CONTEXT`. If you still remember the definition of `Property`, you will recall that 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 StringCellEditor());
registerEditor(String.class, 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:

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

**Figure 7 Existing CellEditors and their hierarchy**

---

## 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 framework, so that 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 8 ColorChooserPanel (15 colors)

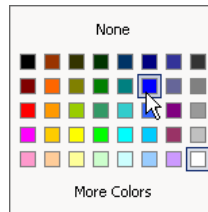


Figure 9 ColorChooserPanel (40 colors)

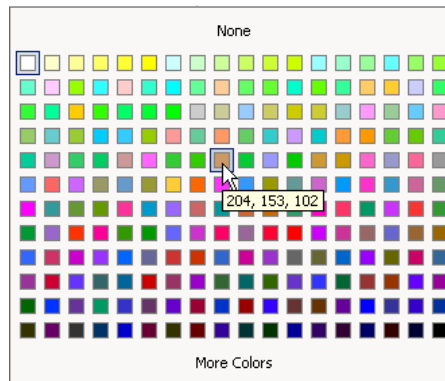


Figure 10 ColorChooserPanel (215 web-safe colors)

In addition to color choosers, we also support gray scale - from 16 gray scales, 102 gray scales and 256 gray scales.

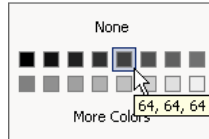


Figure 11 ColorChooserPanel (16 gray scale)

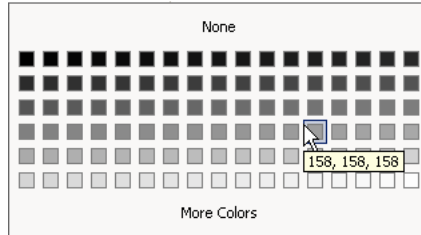


Figure 12 ColorChooserPanel (102 gray scales)

### ColorComboBox

Unsurprisingly, the *ColorComboBox* is a combo box that can choose colors. It uses *ColorChooserPanel* as dropdown popup, as shown below:

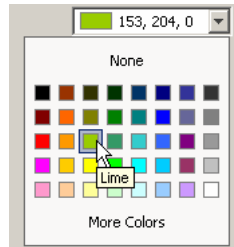


Figure 13 ColorComboBox

### Keyboard Support

ColorComboBox and ColorChooserPanel supports keyboard-only environment.

When Popup is hidden	
ALT + DOWN	To Bring up the popup
When Popup is visible	
ESC	Hide the popup without changing the selection
LEFT	Previous color to the same row. If at the beginning of a row, go to last color of previous row
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).

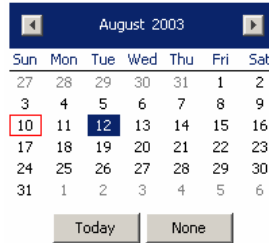


Figure 14 DateChooserPanel

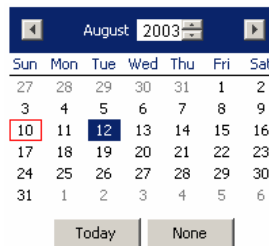


Figure 15 DateChooserPanel (Choosing Year)

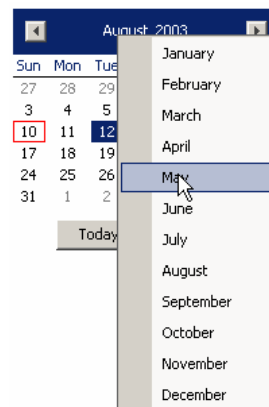


Figure 16 DateChooserPanel (Choosing Month)



**Figure 17 DateChooserPanel (showing week of year panel which is an option)**

```
DateChooserPanel dateChooserPanel = new DateChooserPanel();
dateChooserPanel.setShowWeekNumbers(true)
```



**Figure 18 DateChooserPanel (all dates after Aug 13, 2003 are disabled)**

```
// 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);
```

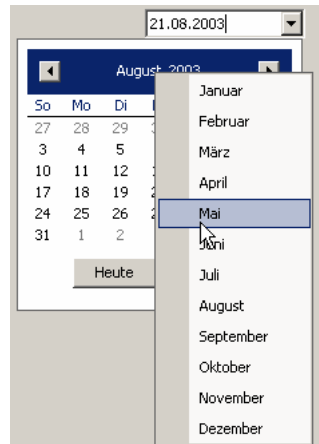


**Figure 19 DateChooserPanel (any weekends are disabled)**

```
// 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 20 DateChooserPanel (Localized Version de)**

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, the user will not be able to change it.

August 2003						
Sun	Mon	Tue	Wed	Thu	Fri	Sat
27	28	29	30	31	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31	1	2	3	4	5	6

**Figure 21 View-only Mode**

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,

August 2003						
Sun	Mon	Tue	Wed	Thu	Fri	Sat
27	28	29	30	31		2
3	4	5	6	7	8	9
10	11	12	13	14		16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31		2	3	4	5	6

simply create a class extending `DateChooserPanel`, overriding the appropriate methods to get the visual effect that you want.

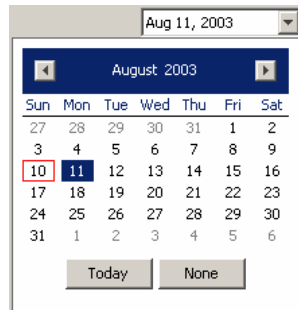
To the right is a simple example that makes each cell bigger, grays the weekends, and shows an icon on the 1<sup>st</sup> and 15<sup>th</sup> of each month.

**Figure 22 Customized Label in View-only Mode**

The methods you can override are: `createDateLabel`, `updateDateLabel`, `createDayOfWeekLabel`, `updateDayOfWeekLabel`, `createMonthLabel`, `updateMonthLabel`, `createYearLabel` and `updateYearLabel`. The default create methods will simply create a standard `JLabel`. Your implementations can override this behavior to create whatever `JComponent` you want (not just a `JLabel`). The update methods will update the actual value of the labels. Since the date is passed in as parameter to all update methods, so you can check the date and do whatever you 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).

### DateComboBox

A `DateComboBox` is a combo box that can choose date, using a `DateChooserPanel` as a dropdown popup.



**Figure 23 DateComboBox**

### Keyboard Support

`DateComboBox` and `DateChooserPanel` supports keyboard-only environment.

When Popup is hidden	
ALT + DOWN	To Bring up the popup
When Popup is visible	
ESC	Hide the popup without changing the selection
LEFT	Previous day of the selected day
RIGHT	Next day of the selected day
UP	Same day of the last week
DOWN	Same day of the next week
HOME	First day of this month
END	Last day of this month
PAGE_UP	Same day of the last month
PAGE_DOWN	Same day of the next month

CTRL + PAGE_UP	Same day of the last year
CTRL + PAGE_DOWN	Same day of the next year
ENTER	Select the highlighted date and hide popup

---

### CREATE YOUR OWN COMBOBOX

---

ComboBox is a very useful component that extends the standard *JComboBox* to overcome the restriction that you may only choose a value from a list. As you can see from the previous discussions, this is not quite good enough. However, with help of *AbstractComboBox*, you can create any type of ComboBox you want.

ComboBox has three parts – a text field editor, a button, and popup that appears when the button is pressed. So the *AbstractComboBox* will allow you to customize all three parts.

```

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

/**
 * 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.
 * If subclass doesn't implement or it returns null, a default button will be created. If type is
DROPDOWN,
 * down arrow button will be used. If type is DIALOG, "..." button will be used.
 * @return the button component
 */
public AbstractButton createButtonComponent() {
    return null;
}

```

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).

*PopupPanel* is the base class for a ComboBox popup - in the case of standard *JComboBox* this is just a standard *JList*. Likewise, in the case of *ColorComboBox*, 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*, it can also be used in a dialog (a *FileChooserPanel* for example). Note also that you do have the choice of using *DROPDOWN* and *DIALOG* when using *AbstractComboBox*.

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 its value.

Based on this, it should be easy to see how simple it is to create any type of *ComboBox* using the *AbstractComboBox* framework. (Note also that *ListComboBox* is simply our implementation of *JComboBox*).

---

## HOW TO CREATE YOUR OWN CELL RENDERER AND EDITOR

---

In this section, we will use *FontNameCellEditor* as an example to illustrate how to create your own cell renderer and cell editor.

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.

```
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 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.

```

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.

---

## 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*.

```
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*.

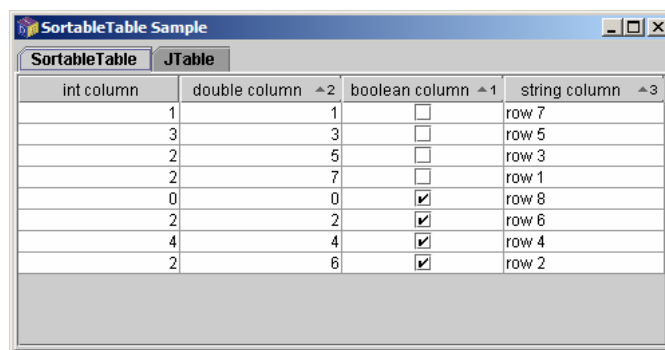
-  `BooleanComparator {com.jidesoft.comparator}`
-  `CalendarComparator {com.jidesoft.comparator}`
-  `ComparableComparator {com.jidesoft.comparator}`
-  `DefaultComparator {com.jidesoft.comparator}`
-  `NumberComparator {com.jidesoft.comparator}`

---

## 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:



int column	double column ↗2	boolean column ↗1	string column ↗3
1	1	<input type="checkbox"/>	row 7
3	3	<input type="checkbox"/>	row 5
2	5	<input type="checkbox"/>	row 3
2	7	<input type="checkbox"/>	row 1
0	0	<input checked="" type="checkbox"/>	row 8
2	2	<input checked="" type="checkbox"/>	row 6
4	4	<input checked="" type="checkbox"/>	row 4
2	6	<input checked="" type="checkbox"/>	row 2

**Figure 24 SortableTable**

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”.

### 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 actual 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*.

```
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 call *getActualModel()* then the *SampleTableModel* will be returned.

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.

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/descending 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.

### The performance of SortableTableModel

We tried to optimize the performance of *SortableTableModel*. The basic strategy is if the table is completely unsorted, 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:

```
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;
    }
}
if (!found) {
    returnRow = low;
}
return returnRow;
}

```

Sort algorithm:

```

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

private void shufflesort(int from[], int to[], int low, int high) {
    if (high - low < 2)
        return;

    int middle = (low + high) / 2;
    shufflesort(to, from, low, middle);
    shufflesort(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 row away. This is why we need a component which can do filter on 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 standard *JTable* or any subclass of *JTable* such as *SortableTable* will do the job. 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 filter.

### Use FilterableTableModel along with SortableTableModel

Both *FilterableTableModel* and *SortableTableModel* are table model wrappers. You can wrap *SortableTableModel* into *FilterableTableModel* and get a table model which is both sortable and filterable. Here is an example of how it works.

```

TableModel model = new DefaultTableModel();
// ... code to populate the model
FilterableTableModel = new FilterableTableModel(new SortableTableModel(model));
SortableTable sortableTable = new SortableTable(filterableTableModel);

```

First, you create any model. Then you wrapped it into a *SortableTableModel*, then you wrapped it again into a *FilterableTableModel*. At the end, set it as model of a *SortableTable* and you will get a table is both sortable and filterable.

Note: Always wrap *SortableTableModel* into *FilterableTableModel* but not the other way around. The reason of that is *FilterableTableModel*, technically speaking, should fire table row removed event when a row is filtered. However we currently don't do that because it will cause too many events. So instead, we just fire one *tableDataChanged* event. If you wrap *FilterableTableModel* into a



*SortableTableModel*, and since *SortableTableModel* depends on the precise event to optimize the sorting, *tableDataChanged* event will cause a resort of everything. Although it will work, it will be less efficient.

---

## 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. They are `grid.properties` under `com.jidesoft.grid`, `converter.properties` under `com.jidesoft.converter` and `combobox.properties` under `com.jidesoft.combobox`. If you need a localized version, you extract these properties from the jar, translate them into the language you want, and put them back into jar with the right file name change according to the locale.