JIDE Common Layer Developer Guide (Open Source Project)

Purpose of This Document

Welcome to the *JIDE Common Layer.* This module was the foundation for all JIDE commercial products. It was delivered as jide-common.jar in all former releases. In April of 2007, JIDE Software open sourced the module under GPL+classpath exception, hoping more and more people will join the project and push it to the next level.

In addition to GPL, *JIDE Common Layer* is dual-licensed. Commercial companies who need to build proprietary software can use the same commercial license under which all other JIDE products are released. Except for *JIDE Common Layer*, the commercial license is free of charge.

This developer guide is for those who want to develop applications using the *JIDE Common Layer* and for those who want to contribute to this project.

Why Swing Components

Thousands and thousands of valuable development hours are wasted on rebuilding components that have been built elsewhere. Why not let us build those components for you, so you can focus on the most value-added part of your application?

What kind of components do we build and how do we choose them?

First of all, those components that are commonly used. Our components provide a foundation to build any Java desktop application. You've probably seen them in some other well-known applications. People are familiar with them. When you see them in our component demo, most likely you will say "Hmm, I can use this component in my application".

Secondly, they are extensible: we never assume our components will satisfy all your requirements. So in addition to what we provide, we always leave extension points so that you can write your own code to extend the component. Believe it or not, our whole product strategy is based on the extensibility of each component we are building. We try to cover all the requirements we can find and to build truly general, useful components. At some point, users will likely find a need we didn't address, but that's fine! Our components allow you to "help yourselves".

Last, but not least, they'll save you time. You use a 3rd party component because you think it will be faster to build on top of it than to start from scratch. If the 3rd party component is very simple, you probably rather building it yourself so that you have full control of the code. If you find the 3rd party component is way too complex and way too hard to configure, you probably also want to build it yourself to avoid the hassle of understanding other people's code. With those in mind, we carefully chose what components to include in our products. We are very "picky" about what components to build. Our pickiness guaranteed that all those components will be useful thus save your valuable time.

All components in this *JIDE Common Layer* are general components built on top of Swing. We built them mainly because we found they are missing from Swing. Many of the components simply extend an existing Swing classes to add more features. They probably should be included in Swing anyway. All components in this project had been used commercially by thousands of developers in various applications. So they are already in production quality when they are included in this open source project.

Why do we open source

JIDE Software was founded back in 2002. Within four years, JIDE became a well-known Swing component provider. We used commercial license term for our products from the very beginning. It was essential for us because it provides the financial support that we needed as a start-up. On the other hand, being commercial is no question a road block for many developers who either can't afford or are prohibited to use the commercial license. In the past couple of years, we saw many emails, blogs and forum posts suggesting us to open source our products. It is the time now.

In this release, we will open source over 30 components which is about 1/3 of our source code (roughly 100K lines out of 300k+ lines). We will have dedicated people to maintain this project to fix bugs and add enhancements. We will also add more components or move components from our commercial offerings to this project. Of course, we welcome people to contribute this project. The source code can be downloaded from http://jide-oss.dev.java.net.

One of main issues in open source project is the lack of technical support. To address this issue, here is our support policy:

- All source codes are javadoc'ed. A developer guide is provided to describe how to use each component.
- For bug reports, we will have dedicated resource to work on them based on the priority we decide.
- For technical support, there are two ways. The first way is the community support. We will provide a forum so that you can get help from other people in the community. The second way is the paid technical support provided by JIDE support team. That is, if you think it's critical to get the high quality support in a timely fashion, you can always purchase the annual maintenance renewal for *JIDE Common Layer*.

Open source *JIDE Common Layer* doesn't mean we will open source all our other components. We still believe high quality software deserves license fee. Open source community won't even exist without the participant of millions of developers whose salaries are paid by commercial companies. So we will continue to market our other products commercially and use part of the license revenue to sponsor this open source project.

Source Code Structure

The table below lists the packages in the *JIDE Common Layer*. All packages are in jide-oss-<version>.jar.

Packages	Description
com.jidesoft.swing	Common components.
com.jidesoft.icon	Icon related classes
com.jidesoft.comparator	Various Comparators. They all implement interface java.util.Comparator. ObjectComparatorManager provides a central place to register those comparators.
com.jidesoft.converter	Various ObjectConverters which can convert an object to/from String. ObjectConverterManager provides a central place to register those converters.
com.jidesoft.grouper	Various ObjectGroupers which can group several values into a named group. ObjectGrouperManager provides a central place to register those groupers.
com.jidesoft.popup	Popup component
com.jidesoft.animation	Animation related classes
com.jidesoft.hints	IntelliHints related classes

A general comment on our naming convention: If the class is modified from or based on an existing Swing/AWT class, we prefix the original Swing/AWT class name with Jide - for example, *JideTabbedPane* (you can tell that it's based on *JTabbedPane* from the name). If it's a completely new component that doesn't exist in Swing/AWT then we don't prefix anything - for example, *Calculator* etc.

We will add more and more components to *JIDE Common Layer* in the future and we will keep the same package organization. If the component is complex enough or there are a group of components which share a common feature, there will be a separate package for it. If it's a very small component, we probably will put it under *com.jidesoft.swing*.

List of Components

Enhanced Swing Components or Classes

- JideButton: built on top of JButton with its ComponentUI. Best used on JToolBar or CommandBar[1]

[1] Part of in JIDE Action Framework.

- JidePopupMenu: built on top of JPopupMenu. It will make sure the content of the popup menu to be inside the screen boundary. If the popup menu is very long, it will add scroll button to the top and bottom so that user can scroll it up and down.
- JideMenu: built on top of JMenu to allow lazily creation of menu items and to allow specifying the popup menu's alignment.
- PartialLineBorder: built on top of LineBorder to only paint lines on certain sides.
- PartialEtchedBorder: built on top of EtchedBorder to only paint etched lines on certain sides.
- JideScrollPane: built on top of JScrollPane to support RowFooter, ColumnFooter as well as new corner components on either side of the scroll bars.
- SimpleScrollPane: built on top of JScrollPane to use four scroll buttons to do the scrolling. It has no scroll bar.
- JideSplitPane: A split pane that supports multiple split.
- JideTabbedPane: built on top of JTabbedPane to support many different tab styles, tab resize mode, tab leading component, tab trailing component etc.
- JideBoxLayout: similar to BoxLayout but can support three different constraints to give child component different resize weight.
- JideBorderLayout: built on top of BorderLayout just to make the north and south component the same width as center component.
- AutoResizingTextArea: A JTextArea which resizes vertically when there are more text in it.
- LabeledTextField: A JTextField which supports JLabel in front of it.
- MultilineLabel: A JTextArea that looks JLabel but supports multiple lines.
- RangeSlider: A JSlider that supports two thumbs to specify a range.
- StyledLabel: built on top of JLabel to support different styles on the text.
- TristateCheckBox: a check box that has three states.
- StandardDialog: built on top of JDialog to support common used dialog standards.

New Components

- JidePopup: support any popup window.
- ButtonPanel: support a panel for buttons in a OS-aware way.
- CheckBoxList/CheckBoxTree: A JList/JTree supports check box in the row/tree node.
- Calculator: a component for calculator.
- FolderChooser: a component for choosing folder.

Usability enhancement

- AutoCompletion (JComboBox, JTextComponent)
- Resizable (ResizablePanel, ResizableWindow)
- Searchable (ComboBox, List, Table, TextComponent etc)
- SearchableBar: A searching component similar to Firebox searching bar. Built on top of Searchable.
- IntelliHints: provides dynamic hints to help user typing. It could be considered as an enhancd version of AutoCompletion. A typical use case for it is IntelliSense in editors of most IDEs.
- Overlayable: provides putting an overlay component on top of another component to display useful information such as validation error, process indicator etc.

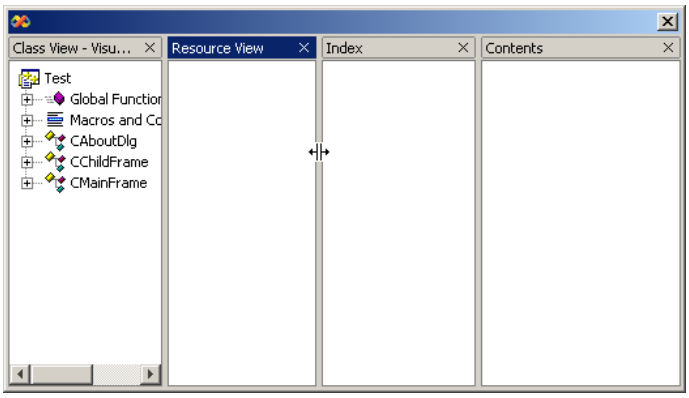
Utilities

- SelectAllUtils
- Sticky
- ColorUtils

Pane

JideSplitPane

Although JSplitPane is a useful Swing component, it has one major limitation: it can only split into two panes. If you want to split into three panes, you have to use two JSplitPanes. That may be OK in most cases, but if you want to split it into four or five or more panes then you will quickly get into trouble, maintaining so many JSplitPanes. As you can see in *JIDE Docking Framework*, we need to be able to split a panel into any number of panes[2]. JSplitPane obviously cannot meet this need gracefully, so we developed JideSplitPane.



Above is an example of a JideSplitPane, which is split into four parts. Each divider can be moved using the mouse, to resize the components either side of it.

JideSplitPane can split either horizontally or vertically, using the two identifiers defined in JideSplitPane as HORIZONTAL_SPLIT and VERTICAL_SPLIT. You can either specify the orientation in the constructor or call setOrientation after it is constructed.

Call addPane(…) or insertPane(…) or add(…) to add a new component. The underlying layout is JideBoxLayout, so you can specify the constraints as VARY, FLEXIBLE or FIX when you call add(…).

By default, the size of the divider is 3 pixels. You can either change this by calling setDividerSize(), or you can change it globally in UIDefaults using the key "JideSplitPane.dividerSize". You can also change the border and background color of the divider in UIDefaults using "JideSplitPaneDivider.border" and "JideSplitPaneDivider.background".

In JSplitPane, you can call set the divider location by calling setDividerLocation(). You can find this method on JideSplitPane too. However the behavior is different. If the JideSplitPane is displayed on screen, setDividerLocation will change the divider location correctly. If the JideSplitPane has never been displayed before, this method call will have no effect. The reason

[2] You can refer to a bug in Java website for information on this particular issue.
http://developer.java.sun.com/developer/bugParade/bugs/4155064.html

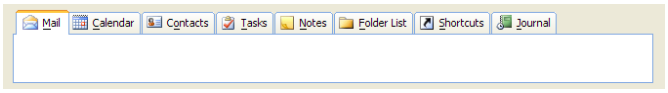
is setDividerLocation changes underlying layout directly. If the JideSplitPane is never displayed, the underlying layout is not initialized properly, thus no effect. This is the correct way to change the initial divider location. The divider location is determined by the preferred size of panes. So instead of setting the location directly, you can set the preferred size of each pane to control the dividers' location. For example, if the preferred width of three panes in HORIZONTAL _SPLIT JideSplitPane is 200, 300, 500 respectively, then the two dividers will be at 20% and 50% of the total width of JideSplitPane.

Continuous Layout refers to painting during drag and drop actions. If this is set to true, then the child components are continuously redisplayed and laid out while moving a window. The default value of this property is false, meaning that only an outline is displayed, which provides much better performance. You can change this with setContinuousLayout(boolean).

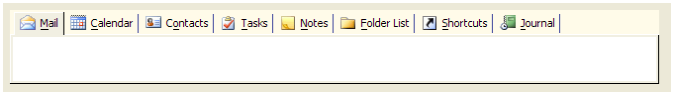
JideTabbedPane

JideTabbedPane is similar to JTabbedPane; the differences are that JideTabbedPane:

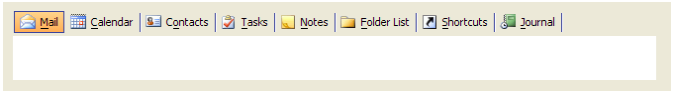
- Has many tab shapes you can choose from. Currently it has
  - *SHAPE_WINDOWS*



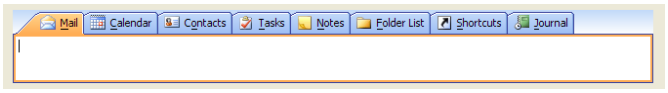
  - *SHAPE_VSNET*



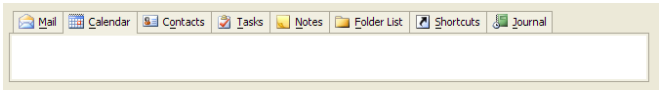
  - *SHAPE_BOX*



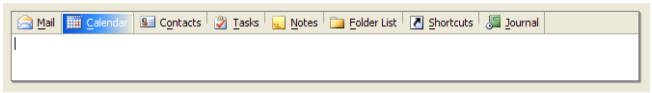
  - *SHAPE_OFFICE2003*



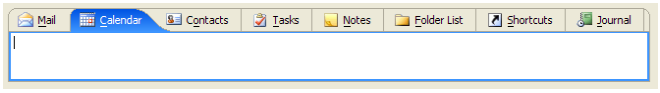
  - *SHAPE_FLAT*



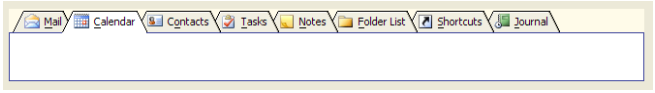
  - *SHAPE_ECLIPSE*



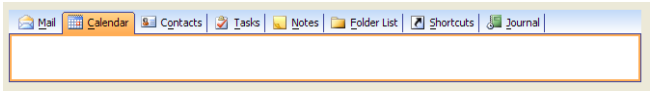
  - *SHAPE_ECLIPSE3x*



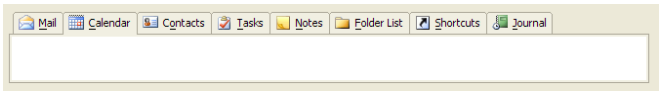
  - *SHAPE_EXCEL*



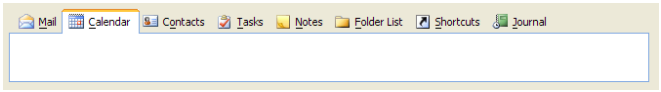
  - *SHAPE_ROUNDED_VSNET*



  - *SHAPE_ROUNDED_FLAT*

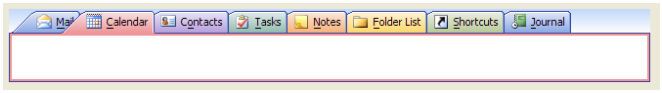


- *SHAPE_WINDOWS_SELECTED.*

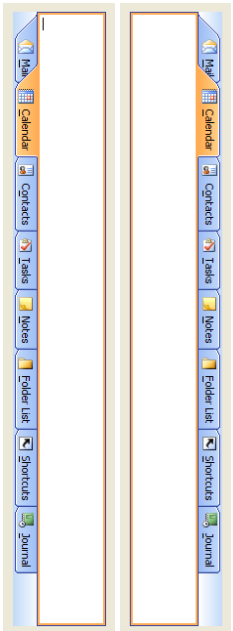


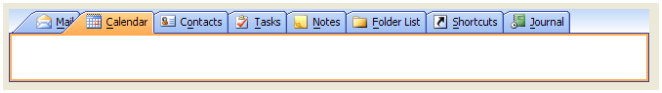
- Has different color themes to choose from. Currently it supports four different themes.
  - COLOR_THEME_WIN2K
  - COLOR_THEME_OFFICE2003
  - COLOR_THEME_VSNET
  - COLOR_THEME_WINXP

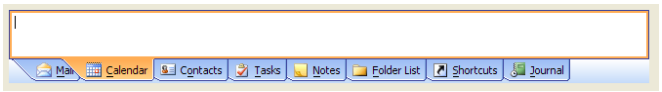
  It even has a special OneNote color theme which is available as part of COLOR_THEME_OFFICE2003.



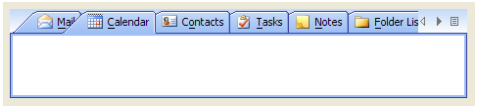
- Supports all four sides as tab placement.



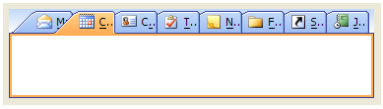




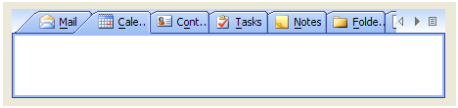
- Has four tab resize layouts.
    - RESIZE_MODE_NONE: it doesn't change tab size when there isn't enough space to hold all tabs. So it uses scroll left and right button to scroll the tabs. There is also a tab list button to show all tabs in popup menu so that you can select the tab even it is not visible.



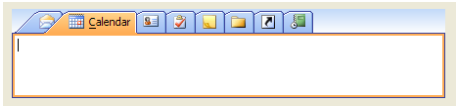
    - RESIZE_MODE_FIT: it shrinks tab size so that all tabs can fit in one row.



    - RESIZE_MODE_FIXED: All tabs have a fixed size which you can define it yourself. Each tab, no matter how long the title is, has the same size. It will not change its size when tabbed pane size changes. So in order to select any tab, you still get scroll left/right and tab list button as in RESIZE_MODE_NONE.

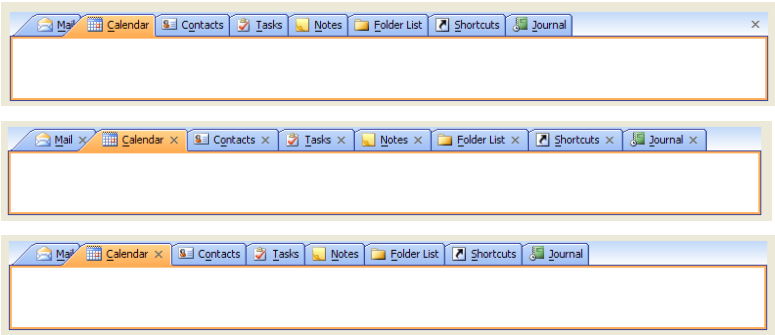


    - RESIZE_MODE_COMPRESSED: This resize mode only shows the selected tab's title. For all unselected tabs, only icons are visible. So if you want to use this mode, you need to make sure you set icons for all tabs.

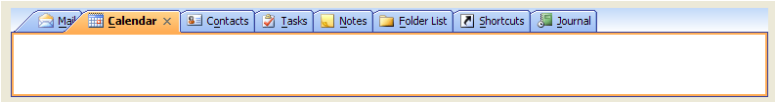


- Has an option to hide the tab area if there is only one component in a tabbed pane. This is a feature used by JIDE Docking Framework.
- Has an option to show close button on the corner, on the tab, or on the selected tab. This is very useful especially each tab is a document in *DocumentPane*. To use this option, you need to call the following two calls. If you never call setShowCloseButtonOnTab, a default value will be used by reading it from L&F. For example, in VSNET L&F, the value is false. In Eclipse L&F, the value is true. So if you want to set it freely, you must disable the L&F by setUseDefaultShowCloseButtonOnTab to false. Then whatever value you set to setShowCloseButtonOnTab will be used.

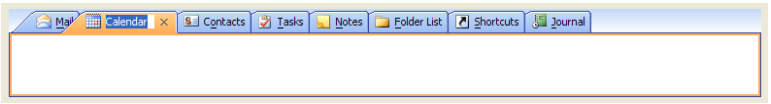
```
tabbedPane.setUseDefaultShowCloseButtonOnTab(false);
tabbedPane.setShowCloseButtonOnTab(true);
```



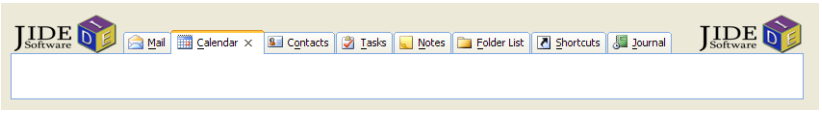
- Can show the selected tab's title in bold font.



- JideTabbedPane also supports inline tab title editing. By default, this feature is disabled. You need to enable it by calling *setTabEditingAllowed(true)*. If enabled, user can double click on any tab to start editing the title. See below.



- Allow tabLeadingComponent and tabTrailingComponent. This feature allows you to add your own component to the area before tabs and after tabs.



Since JideTabbedPane extends JTabbedPane, the usage of it is exactly the same as JTabbedPane, except for the differences in appearance, noted above.

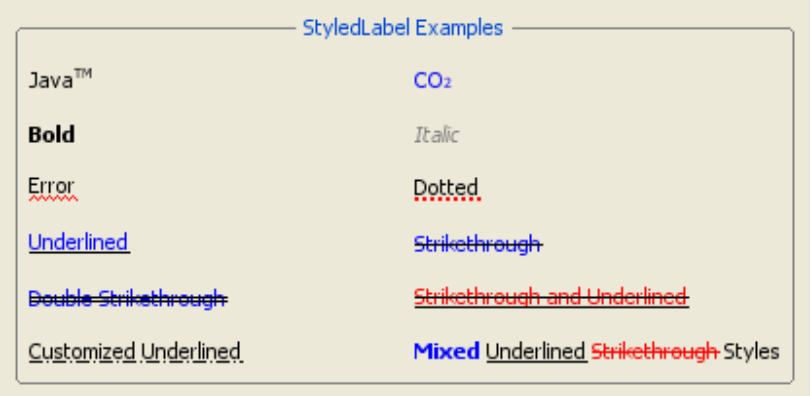
UI Defaults used by JideTabbedPane

Name	Type	Description
JideTabbedPane.border	Border	The border of tabbed pane
JideTabbedPane.background	Color	The background of tabbed pane
JideTabbedPane.foreground	Color	The foreground of tabbed pane
JideTabbedPane.light	Color	One of the colors used to paint the tab border
JideTabbedPane.highlight	Color	One of the colors used to paint the tab border
JideTabbedPane.shadow	Color	One of the colors used to paint the tab border
JideTabbedPane.darkShadow	Color	One of the colors used to paint the tab border
JideTabbedPane.tabInsets	Insets	The insets of each tab
JideTabbedPane.contentBorderInsets	Insets	The insets of tab content

JideTabbedPane.tabAreaInsets	Insets	The insets of the area where all the tabs are
JideTabbedPane.tabAreaBackground	Color	The tab area background
JideTabbedPane.font	Font	The font used by tabbed pane
JideTabbedPane.selectedTabFont	Font	The font used to draw the text of the selected tab
JideTabbedPane.unselectedTabTextForeground	Color	The default text color of unselected tabs. If setForegroundAt() is call to set a new color, the new color will be used. The selected tab foreground is whatever color returned from getForegroundAt().
JideTabbedPane.selectedTabBackground	Color	The selected tab background. The unselect tab background is tabAreaBackground
JideTabbedPane.textIconGap	Integer	The gap between icon and text
JideTabbedPane.showIconOnTab	Boolean	Whether to show icon on tabs
JideTabbedPane.showCloseButtonOnTab	Boolean	Whether to show close button on tabs
JideTabbedPane.closeButtonAlignment	Integer	If the close button is on tab, what is the alignment. It could be LEADING or TRAILING defined in SwingConstants.

StyledLabel

Features of StyledLabel



StyledLabel is an enhanced version of *JLabel* to display text in different colors and styles along with all kinds of line decorations.

JLabel is simple and fast but has very limited features. For example, you can't use different color to draw the text. You may argue *JLabel* can use HTML tag to display text in different colors. However there are two drawbacks. First it is very slow[3]. Secondly it is buggy[4]. Comparing with HTML *JLabel*, StyledLabel is very fast and almost as fast as *JLabel* with plain text. On the other hand, *JTextPane* is powerful and can display text in different colors. But in the cases like cell renderer, *JTextPane* is obvious overkill.

As you can see from the screenshot above, *StyledLabel* sits between *JLabel* and *JTextPane* and provides a very simple and fast way to display text in different colors and styles. It can also support decorations using all kinds of additional styles.

Here is the list of features that *StyledLabel* support.

- Uses different font styles to display the text.
- Uses different colors to display the text
- Subscript and superscript

---

[3] You can see StyledLabelPerformanceDemo.java in examples\B15. StyledLabel folder to see a performace test of HTML JLabel and StyledLabel.

[4] See bug report at http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4373575. Sun claimed it is fixed but it is not as another user pointed it out at the end. If you run the test case provided by original submitter, you will immediately notice the tree node disappeared when you click on the tree nodes. This bug is actually one of the main reasons we decided to create *StyledLabel*.

- Line decoration include solid line, dotted line, waved line, double solid line or any arbitrary line style that can be defined by Java2D's Stroke class.
- Two line locations – underlined or strikethrough or both.
- Can be used as cell renderer for *JList*, *JTable*, or *JTree*.

Classes, Interfaces and Demos

Classes	
StyledLabel (com.jidesoft.swing)	The main class for *StyledLabel*.
StyleRange (com.jidesoft.swing)	This is the class to define the style. Since the style is defined based for a range of text in *StyledLabel*, that's why it is called *StyleRange*.
StyledListCellRenderer (com.jidesoft.list)	A list cell renderer which uses *StyledLabel* instead of *JLabel*.
StyledTableCellRenderer (com.jidesoft.grid)	A table cell renderer which uses *StyledLabel* instead of *JLabel*.
StyledTreeCellRenderer (com.jidesoft.tree)	A tree cell renderer which uses *StyledLabel* instead of *JLabel*.
Demos	
StyledLabelDemo (examples\B15. StyledLabel)	A demo to demonstrate the *StyledLabel* used as standalone labels as well as used in *JTree*, *JList* and *JTable*.

How to use StyledLabel

The design of *StyledLabel* is very similar to *StyledText* class in SWT. It even has *StyleRange* class just like in SWT. *StyledLabel* can have zero, one or many *StyleRanges*.

StyleRange

StyleRange describes a style for a range of text. For example, to display a *StyledLabel* like "Java™", the *StyleRange* will be

```
new StyleRange(4, 2, Font.PLAIN, StyleRange.STYLE_SUPERSCRIPT)
```

It means "starting from the 4th characters, for the next 2 characters, use PLAIN font to draw the text and apply superscript style".

If no *StyleRange* is set to *StyledLabel*, *StyledLabel* will behave exactly the same as *JLabel*. You can also add multiple *StyleRange*s as long as those ranges don't overlap with each other. If you add a new *StyleRange* that overlaps with previously set *StyleRanges*, the new *StyleRange* will be ignored.

Here is the information you can set to *StyleRange*.

int start	The start index of the range.
int length	The length of the range
int fontStyle	The font style. The valid values are *Font.PLAIN*, *Font.BOLD*, *Font.ITALIC*, or *Font.BOLD* | *Font.ITALIC*.
Color fontColor	The text color.
Color lineColor	The line color
Stroke lineStroke	The line stroke. If there are lines in the additional style, the line stroke will be used to paint the line.
int additionalStyle	The additional style. This is the property you set to get all kinds of styles. The valid values are *STYLE_STRIKE_THROUGH* *STYLE_DOUBLE_STRIKE_THROUGH* *STYLE_WAVED* *STYLE_UNDERLINED* *STYLE_DOTTED* *STYLE_SUPERSCRIPT* *STYLE_SUBSCRIPT* They are all defined in *StyleRange* as constants. You can even use a combination of several styles by using "|" as long as they make sense. For example, you can use both strike through and underlined. However you can not use both superscript and subscript.

StyledLabel

StyledLabel has several methods to change *StyleRange*. The most used one is

```
public void setStyleRange(StyleRange styleRange)
```

This method will set one *StyleRange* to *StyledLabel* while keeping any other *StyleRanges* if they are set earlier.

There are also two methods to allow you quickly add several *StyleRanges* at once. The only difference is the first one will clear *StyleRanges* that was set earlier.

```
public void setStyleRanges(StyleRange[] styleRanges)
public void addStyleRanges(StyleRange[] styleRanges)
```

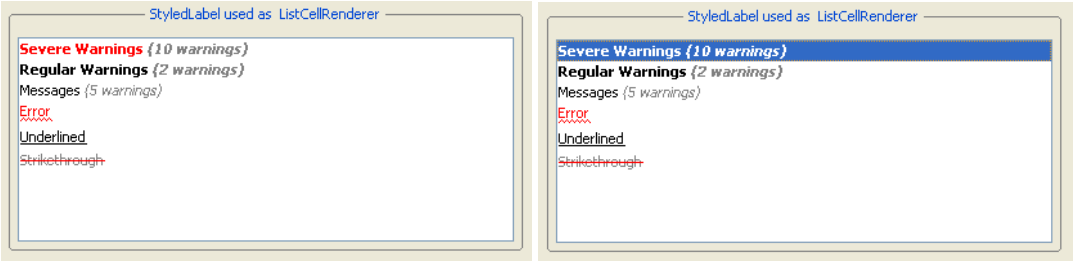
There are of course two methods to help you clear StyleRanges.

```
public void clearStyleRange(StyleRange styleRange)
public void clearStyleRanges()
```

All the methods above will fire property change event on property "styleRange". The property name is defined as *StyleRange.PROPERTY_STYLE_RANGE*.

StyledLabel only has one new property called "ignoreColorSettings". If this property is true, the color setting defined *StyleRange* will be ignored and the default foreground will be used to paint the text and color. The color settings include font color and line color. The reason we need this property is for cell renderer. Cell renderer, when selected, need to use selection background. Selection background is usually defined by specific LookAndFeel, there is no way you can guarantee the color you used in *StyleRange* works well with the selection background. To avoid color confliction, we will set this property to true if the cell is selected.

You will know exactly what this property for by looking at the two screenshots below. Although we use red and gray color in the first cell, they become white (the default selection foreground) when the cell is selected. You can imagine the gray color won't look good on a blue background.



Almost all the features provided by *JLabel* still work with *StyledLabel*. You can add icon. You can set the alignment of the icon or the text or set text position. You can even set mnemonic just like in *JLabel*. However you need to be aware that if you also use certain underlined line style, the mnemonic indicator might be conflict with the underline.

Code Examples

1. Display "TM" as superscript in string "Java™".

```
StyledLabel javaTM = new StyledLabel("JavaTM");
javaTM.setStyleRange(new StyleRange(4, 2, Font.PLAIN, StyleRange.STYLE_SUPERSCRIPT));
```

Here is the result.

Java™

2. Display several line styles in the same StyledLabel.

```
StyledLabel mixed = new StyledLabel("Mixed Underlined Strikethrough Styles");
mixed.setStyleRange(new StyleRange(0, 5, Font.BOLD, Color.BLUE));
mixed.setStyleRange(new StyleRange(6, 10, Font.PLAIN, Color.BLACK, StyleRange.STYLE_UNDERLINED));
mixed.setStyleRange(new StyleRange(17, 13, Font.PLAIN, Color.RED,
StyleRange.STYLE_STRIKE_THROUGH));
```

Here is the result.

3. Display a customized underline style.

```
StyledLabel customizedUnderlined = new StyledLabel("Customized Underlined");
customizedUnderlined.setStyleRange(new StyleRange(Font.PLAIN, Color.BLACK,
StyleRange.STYLE_UNDERLINED, Color.BLACK, new BasicStroke(1.0f, BasicStroke.CAP_SQUARE,
BasicStroke.JOIN_ROUND, 1.0f, new float[]{6, 3, 0, 3}, 0)));
```

Here is the result.

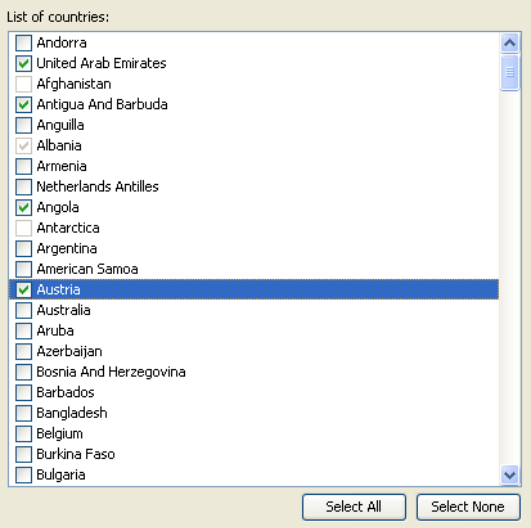
4. Uses *StyledTreeCellRenderer*. Here is the how to set the renderer onto tree.

```
tree.setCellRenderer(new StyledTreeCellRenderer() {
    protected void customizeStyledLabel(JTree tree, Object value, boolean sel, boolean expanded, boolean
leaf, int row, boolean hasFocus) {
        super.customizeStyledLabel(tree, value, sel, expanded, leaf, row, hasFocus);
        String text = getText();
        // here is the code to customize she StyledLabel for each tree node
    }
});
```

Here is the result.

CheckBoxList

Features of CheckBoxList



CheckBoxList is a special *JList* which uses *JCheckBox* as the list cell renderer. In addition to regular *JList*'s features, it also allows you select any number of rows in the list by selecting the check boxes.

To select an element, user can mouse click on the check box, or select one or several rows and press SPACE key to toggle the check box selection for all selected rows.

Here is the list of features that *CheckBoxList* support.

- Check or uncheck each row.
- Check or uncheck multiple rows by selecting them first
- Still supports customized cell renderer as before. The cell renderer will be the part to the left of the check box (when it's left-to-right orientation).

Classes, Interfaces and Demos

Classes
The first implementation [5]

[5] Due to a design change, there are currently two working versions for *CheckBoxList*. The first one is just called *CheckBoxList*. This one used the same design as *CheckBoxTree* and uses a *DefaultListSelectionModel* as the selection model to keep track of which check boxes are checked. The second implementation is called *CheckBoxListWithSelectable*. It stored the check box state information in *ListModel* by converting the element in the *ListModel* to Selectable.

CheckBoxList (com.jidesoft.swing)	The main class for *CheckBoxList*.
CheckBoxListCellRenderer (com.jidesoft.swing)	The list cell renderer which uses check box as cell renderer.
The second implementation	
CheckBoxListWithSelectable (com.jidesoft.swing)	
Selectable (com.jidesoft.swing)	This is an interface to indicate something can be selected.
DefaultSelectable (com.jidesoft.swing)	Default implementation of Selectable.
Demos	
CheckBoxListDemo (examples\B10. CheckBoxTree)	A demo to demonstrate the *CheckBoxList.*

Code Examples

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

```
CheckBoxList checkBoxList = new CheckBoxList(Object[] or Vector);

or

CheckBoxListWithSelectable checkBoxList = new CheckBoxListWithSelectable(Object[] or Vector);
```

2. To find out when the check box state changes in *CheckBoxList*.

```
checkBoxList. getCheckBoxListSelectionModel().addListSelectionListener(new ListSelectionListener () {
        void valueChanged(ListSelectionEvent e) {
            // your code here.
        }
    });
```

3. To find out all selected objects

```
Object[] objects = checkBoxList.getSelectedObjects(); // or getCheckBoxListSelectedValues()
```

   The objects will be the array of objects that are checked.

4. To select all the rows or to clear all the selected rows

```
checkBoxList.selectAll();
checkBoxList.selectNone();
```

5. Change the cell renderer for *CheckBoxList*.

```
checkBoxList.setCellRenderer(a new cell renderer);
```

CheckBoxList has its cell renderer which has check box. However it doesn't prevent you from setting your own cell renderer. As you can see from the code above, the way to set a new cell renderer is just like before. *CheckTreeList* will use the new cell renderer and add check box before it. The difference is if you call *getCellRenderer()*, you will not get the cell renderer you set but get the check box cell renderer. You can use *getActualCellRenderer()*, which is a new method we added, to get the actual cell renderer you set.

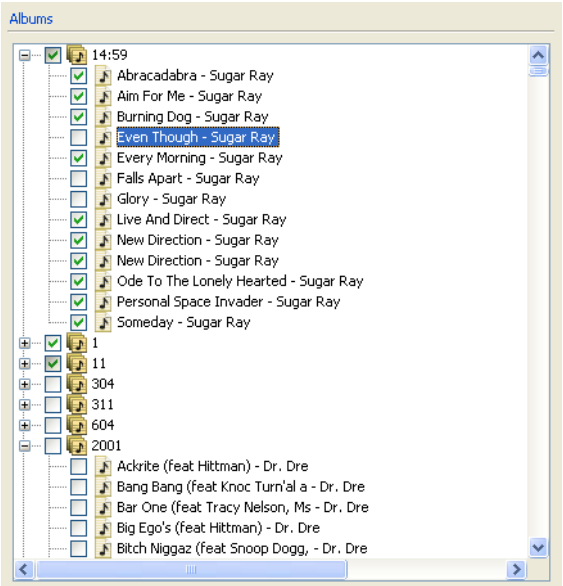
6. Define your own *ListModel* that works with *CheckBoxList*.

```
ListModel listModel = new AbstractListModel () {
    public Object getElementAt(int row) {
      //make sure you return an element which is instance of Selectable. In most case, you can
      // DefaultSelectable and wraps your object into it. Or you can make your
      // object implementing Selectable.
    }

    public int getSize() {
      // return whatever size
    }
};
CheckBoxList checkBoxList = new CheckBoxList(listModel);
```

CheckBoxList doesn't keep the check box selection state in itself. All the selection information is kept in Selectable object in the ListModel. Good thing about this approach is the selection model will never go out of sync with data model. Bad thing is the data model needs to be changed to support it. However this change should be trivial in most cases.

CheckBoxTree

Features of CheckBoxTree



CheckBoxTree is a special *JTree* which uses *JCheckBox* as the tree renderer. In addition to regular *JTree*'s features, it also allows you select any number of tree nodes in the tree by selecting the check boxes.

To select an element, user can mouse click on the check box, or select one or several tree nodes and press SPACE key to toggle the check box selection for all selected tree nodes.

Here is the list of features that *CheckBoxTree* support.

- Check or uncheck each tree node.
- Check or uncheck multiple tree nodes by selecting them first
- Supports dig-in mode
- Still supports customized cell renderer as before. The cell renderer will be the part to the left of the check box (when it's left-to-right orientation).

Classes, Interfaces and Demos

Classes	
CheckBoxTree (com.jidesoft.swing)	The main class for *CheckBoxTree*.
CheckBoxTreeSelectionModel (com.jidesoft.swing)	This is the selection model to keep track of the check/uncheck state of check boxes.

CheckBoxTreeCellRenderer (com.jidesoft.swing)	The tree cell renderer which uses check box as cell renderer.
TristateCheckBox (com.jidesoft.swing)	A check box can display three states. We used it to show a parent tree node to indicate three different states. The three states are all children are selected, none of the children are selected, and some of the children are selected.
Demos	
CheckBoxTreeDemo (examples\B16. CheckBoxTree)	A demo to demonstrate the *CheckBoxTree.*

Code Examples

7. To create a *CheckBoxTree*. There is no difference from creating a regular *JTree*.

```
CheckBoxTree checkBoxTree = new CheckBoxTree(treeModel);
```

8. To find out when the check box state changes in *CheckBoxTree*.

```
checkBoxTree.getCheckBoxTreeSelectionModel().addTreeSelectionListener(new TreeSelectionListener() {
        public void valueChanged(TreeSelectionEvent e) {
            // your code here.
        }
    });
```

9. To find out which tree nodes are checked

```
TreePath[] treePaths = checkBoxTree.getCheckBoxTreeSelectionModel().getSelectionPaths();
```

The treePaths will be the list of tree path that are checked.

10. Change the dig-in mode.

```
checkBoxTree.getCheckBoxTreeSelectionModel().setDigIn(true/false);
```

If the *CheckBoxTree* is in dig-in mode, checking the parent node will check all the children. Correspondingly, *getSelectionPaths()* will only return the parent tree path. If not in dig-in mode, each tree node can be checked or unchecked independently.

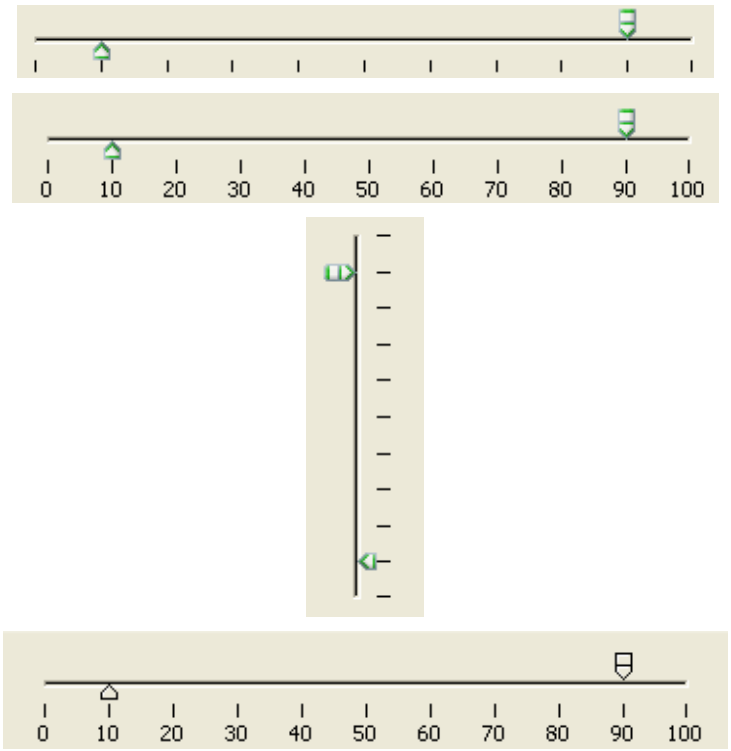
11. Change the cell renderer for *CheckBoxTree*.

```
checkBoxTree.setCellRenderer(a new cell renderer);
```

CheckBoxTree has its cell renderer which has check box. However it doesn't prevent you from setting your own cell renderer. As you can see from the code above, the way to set a new cell renderer is just like before. *CheckTreeTree* will use the new cell renderer and add check box before it. The difference is if you call *getCellRenderer()*, you will not get the cell renderer you set but get the check box cell renderer. You can use *getActualCellRenderer()*, which is a new method we added, to get the actual cell renderer you set.

RangeSlider

Features of RangeSlider



RangeSlider extends *JSlider* but it allows user to choose two values to form a range.

Here is the list of features that *RangeSlider* support.

- Allow to choose lower value and upper value separately to form a range
- Allow to move both lower value and upper value at the same time
- Support both horizontal and vertical orientation
- Support a default style and Windows XP style with roll over effect (available if you set style to Office2003 style in LookAndFeelFactory)

Classes, Interfaces and Demos

Classes	
RangeSlider (com.jidesoft.swing)	The main class for *RangeSlider*.
Demos	

RangeSliderDemo (examples\B17. RangeSlider)	A demo to demonstrate the *RangeSlider*.

How to use RangeSlider

RangeSlider extends *JSlider*, so the usage of it is almost the same as *JSlider*. For example, you can set min and max value; you can set the major tick and minor spacing; you can set tick/label/track visibility. In addition, *RangeSlider* allows you to set lower value and upper value. Please see code examples below to find out how to use it.

Code Examples

1. Creates a *RangeSlider* with certain min/max/lower/upper value

```
RangeSlider rangeSlider = new RangeSlider(0, 100, 10, 90);
```

2. Creates a *RangeSlider* with major ticks on

```
RangeSlider rangeSlider = new RangeSlider(0, 100, 10, 90);
rangeSlider.setPaintTicks(true);
rangeSlider.setMajorTickSpacing(10);
```

3. Creates a *RangeSlider* with tick label visible

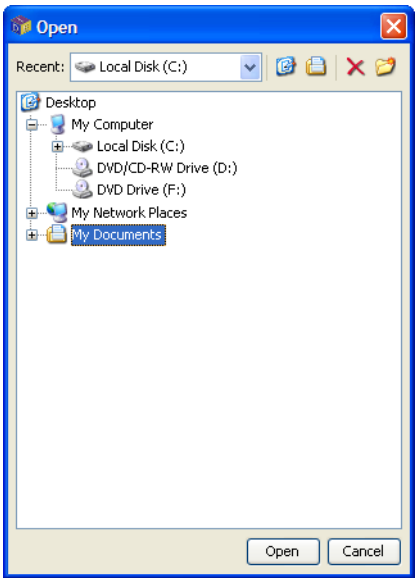
```
RangeSlider rangeSlider = new RangeSlider(0, 100, 10, 90);
rangeSlider.setPaintTicks(true);
rangeSlider.setMajorTickSpacing(10);
rangeSlider.setPaintLabels(true);
```

4. Creates a vertical *RangeSlider*

```
RangeSlider rangeSlider = new RangeSlider(SwingConstants.VERTICAL);
```

FolderChooser

Features of FolderChooser



FolderChooser extends *JFileChooser* and uses a familiar interface to choose a folder.

Here is the list of features that *FolderChooser* support.

- Chooses a folder in file system
- Remembers a list of recent selected folders
- Allows delete and new folder
- Allows quick access to Home and My Document folder

Classes, Interfaces and Demos

Classes	
FolderChooser (com.jidesoft.swing)	The main class for *FolderChooser*.
Demos	
FolderChooserDemo (examples\B18. FolderChooser)	A demo to demonstrate the *FolderChooser*.

How to use FolderChooser

FolderChooser extends *JFileChooser*, so the usage of it is almost the same as *JFileChooser*. Please see code examples below to find out how to use it.

Code Examples

1. Show an Open folder chooser dialog

```
FolderChooser folderChooser = new FolderChooser();
int result = folderChooser.showOpenDialog(button.getTopLevelAncestor());
if (result == FolderChooser.APPROVE_OPTION) {
// call folderChooser.getSelectedFile() to get selected folder
}
```

2. Show an Save folder chooser dialog

```
FolderChooser folderChooser = new FolderChooser();
int result = folderChooser.showSaveDialog(button.getTopLevelAncestor());
if (result == FolderChooser.APPROVE_OPTION) {
// call folderChooser.getSelectedFile() to get selected folder
}
```

3. Set recent list to folder chooser

```
List recentList = new ArrayList(); // create recent list
// add File to recent list

FolderChooser folderChooser = new FolderChooser();
folderChooser.setRecentList(recentList);
int result = folderChooser.showOpenDialog(button.getTopLevelAncestor());
if (result == FolderChooser.APPROVE_OPTION) {
// call folderChooser.getSelectedFile() to get selected folder
}
```

4. Display hidden folder in folder chooser

```
FolderChooser folderChooser = new FolderChooser();
folderChooser.setFileHidingEnabled(true); // show hidden folders
int result = folderChooser.showOpenDialog(button.getTopLevelAncestor());
if (result == FolderChooser.APPROVE_OPTION) {
// call folderChooser.getSelectedFile() to get selected folder
}
```

Searchable Components

JList, JComboBox, JTable, JTree, JTextComponent are five data-rich components. They can be used to display a huge amount of data so searching function will be a very useful feature in those components. By default, JList kind of supports searching. User can type in a key and the list will automatically select that row whose first character matches with the typed key. However it can only match the first character. So the goal of this component is to make all five components searchable[6].

Searchable is such a class that makes it possible. User can simply type in any string they want to search for and use arrow keys to navigate to next or previous occurrence. We implement *ListSearchable, ComboBoxSearchable, TableSearchable, TreeSearchable*, *TextComponentSearchable* to make JList, JComboBox, JTable, JTree, and JTextComponent searchable respectively. In addition, we create SearchableUtils encapsulate different classes into one utility class.

It's very easy to use those classes. For example, if you have a JList, all you need to do

```
    JList list = new JList();
SearchableUtils.installSearchable(list);
```

It's exactly the same to make JTable or JTree searchable – just replace *ListSearchable* with corresponding *ComboBoxSearchable*, *TableSearchable, TreeSearchable* or *TextComponentSearchable*.

If you need further configure the searchable, you can

```
    JList list = new JList();
ListSearchable searchable = SearchableUtils.installSearchable(list);
    // further configure it
searchable.setCaseSensitive(true);
```

Usually you don't need to uninstall the searchable from the component. But if for some reason, you need to disable the searchable feature of the component, you can call uninstallSearchable().

[6] The searchable idea is really come from IntelliJ IDEA. In IDEA, all the trees and lists are searchable. We found this feature is very useful and consider it as one of the key features to improve the usability of a user interface. So we further extend this idea and make JTable searchable too. We also added several more features such as multiple select and select all that IDEA doesn't have.

```
    Searchable searchable = SearchableUtils.installSearchable(component);
// ...
// Now disable it
SearchableUtils.uninstallSearchable(searchable);
```

Below are examples of searchable JList and JTable.

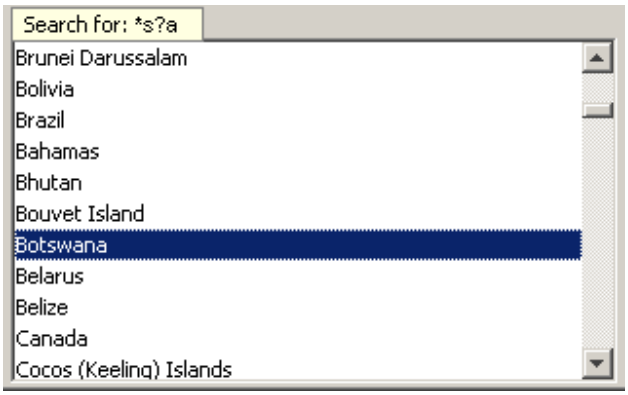


Figure 1 Searchable JList – use up/down arrow key to navigate to next or previous occurrence

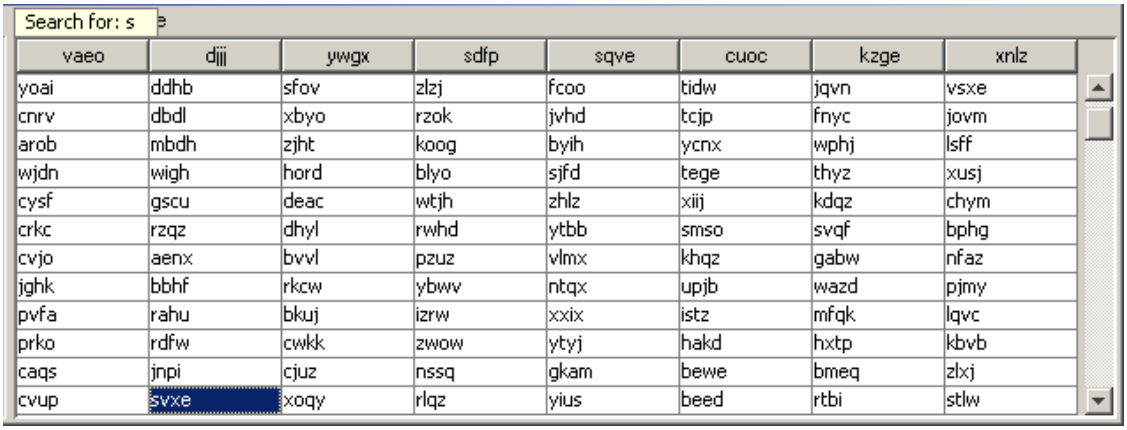


Figure 2 Searchable JTable – use up/down/left/right to navigate to next or previous occurrence

For JComboBox, we only make non-editable combo box searchable. So make sure you call *combobox.setEditable(false)* before you pass it into *SearchableUtils*[7].

For JTextComponent, the searchable popup will not be displayed unless user types in Ctrl-F. The reason is obvious – because the JTextComponent is usually editable. If the JTextComponent is not editable, typing any key will show the popup just like other components.

[7] You may wonder why we only support searchable on non-editable combo box. We could have "searchable" feature on editable combo box but it is not "searchable" as it is described in this chapter but autocompleting. We will introduce autocompleting combobox in the future which will fill the gap of "searchable" editable combobox.

Features

The main purpose of searchable is to make the searching for a particular string easier in a component having a lot of information. All features are related to how to make it quicker and easier to identify the matching text.

Navigation feature - After user types in a text and press up or down arrow key, only items which match with the typed text will be selected. User can press up and down key to quickly look at what those items are. In addition, home key will navigate to the first occurrence. And end key will navigate to the last occurrence. The navigation keys are fully customizable. Next section will tell you how to customize them.

Multiple selection feature - If you press and hold CTRL key while pressing up and down arrow, it will find next/previous occurrence while keeping existing selections. See the screenshot below. So you can easily find several occurrences and apply an action to all of them later.

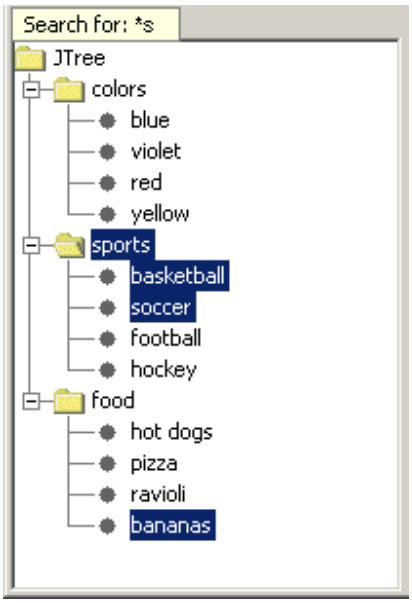


Figure 3 Multiple Selections

Select all feature – Further extending the multiple selections feature, you can even select all. If you type in a searching text and press CTRL+A, all the occurrences matching the searching text will be selected. This is a very handy feature. For example you want to delete all rows in a table whose “name” column begins with “old”. So you can type in “old” and press CTRL+A, now all rows beginning with “old” will be selected. If you hook up delete key with the table, pressing delete key will delete all selected rows. Imagine without this searchable feature, users will have to hold CTRL key, look through each row, and click on the row they want to delete. In case they forgot to hold tight the CTRL key while clicking, they have to start over again.

Basic regular expression support - It allows '?' to match any character and '*' to match any number of characters. For example “a*c” will match “ac”, “abc”, “abbbc”, or even “a b c” etc. “a?c” will only match “abc” or “a c”.

Recursive search (only in TreeSearchable) – In the case of TreeSearchable, there is an option called recursive. You can call TreeSearchable#setRecursive(true/false) to change it. If TreeSearchable is recursive, it will search all tree nodes including those which are not visible to find the matching node. Obviously, if your tree has unlimited number of tree nodes or a potential huge number of tree nodes (such as a tree to represent file system), the recursive attribute should be false. To avoid this potential problem in this case, we default it to false.

How to extend Searchable

Searchable is a base abstract class. For each implementation, there are at least five methods need to be implemented.

```
protected abstract int getSelectedIndex()
protected abstract void setSelectedIndex(int index, boolean incremental)
protected abstract int getElementCount()
protected abstract Object getElementAt(int index)
protected abstract String convertElementToString(Object element)
```

The keys used by this class are fully customizable. Subclass can override the methods to customize the keys. For example, isActivateKey() is defined as below.

```
protected boolean isActivateKey(KeyEvent e) {
    char keyChar = e.getKeyChar();
    return Character.isLetterOrDigit(keyChar) || keyChar == '*' || keyChar == '?';
}
```

In your case, you might need additional characters such as '_', '+' etc. So you can override isActivateKey() to provide additional keys to activate the search popup. In order to override a method, you can't use SearchableUtils anymore. You have to do create a Searchable yourself. However it's still very easy. See below.

```
ListSearchable listSearchable = new ListSearchable(list) {
    protected boolean isActivateKey(KeyEvent e) {
        return ...;
    }
};
```

More methods subclass can override are isDeactivateKey(), isFindFirstKey(), isFindLastKey(), isFindNextKey(), isFindPreviousKey()

We provide basic regular expression support. It's possible to implement full regular expression support. We didn't do that because not many users are familiar with complex regular expression grammar. However if your user base is very familiar with regular expression, you can add the feature to Searchable. All you need to do is to override compare(String text, String searchingText) method and implement the comparison algorithm by yourself. Leveraging javax regex package, it should be an easy task for you.

Resizable Components

In Swing, almost all light-weight components are not resizable[8]. For heavy-weight component, JWindow and undecorated JDialog are not resizable either. The main reason of this is that component size is determined by Layout Manager in Swing. If the parent container size changes, the component size will change accordingly. However this doesn't mean there is no need for resizable component. A typical usage of a resizable panel is in icon or form designer. See the picture below for an example. While designing the icon, you want to control the icon size as well. You can do it by resizing the canvas. In this case, icon size will be the canvas size.

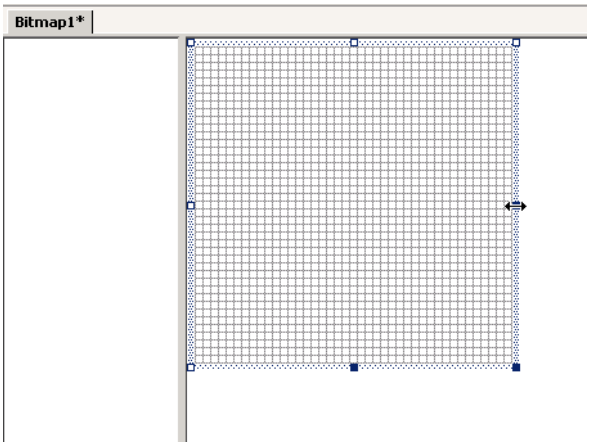


Figure 4 Usage of a Resizable component

In additional to the canvas case above, we also find the need for resizable JWindow or resizable undercoated JDialog. A typical usage case of resizable window is in combobox. In Swing JComboBox, the popup is not resizable. However you can see a resizable popup in IE. See below. The only way to implement this in Swing is to put the JList in a resizable JWindow. So, we do need resizable JWindow.

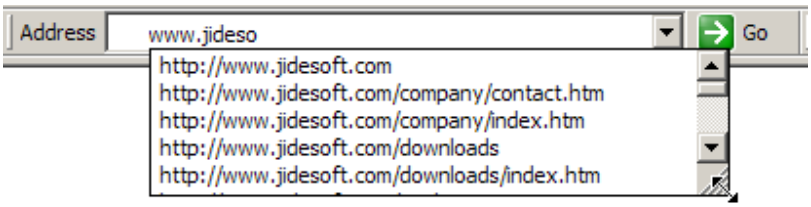


Figure 5 Resizable Window in IE

Resizable

Resizable is such a class that makes component resizable possible. Very similar to *Searchable* class, *Resizable* also adds necessary mouse listener to a component and make it resizable when

[8] The only exception is JInternalFrame which is light-weight and resizable.

you pass that component to *Resizable* constructor. You also need to make sure the component has a non-empty border. Otherwise, there is no place for cursor to change shape and perform the dragging during resizing.

Resizable supports several options –

ResizableCorners - The value of *ResizableCorners* is a bitwise OR of eight constants defined in *Resizable*. See below. So you have complete control of which sides/corners are resizable.

```
public final static int NONE = 0x0;
public final static int UPPER_LEFT = 0x1;
public final static int UPPER = 0x2;
public final static int UPPER_RIGHT = 0x4;
public final static int RIGHT = 0x8;
public final static int LOWER_RIGHT = 0x10;
public final static int LOWER = 0x20;
public final static int LOWER_LEFT = 0x40;
public final static int LEFT = 0x80;
public final static int ALL = 0xFF;
```

ResizeCornerSize – As you know, the mouse cursor will change shape along the resizable component border. If the mouse is near the corner, it will resize both adjacent sides. The value of *resizeCornerSize* will define how big the corner is. The value is in pixel.

beginResizing(), resizing() and endResizing() – those are not really options but methods on *Resizable*. These three methods will be called during resizing. *beginResizing()* and *endResizing()* will be called only once when it starts to resize and when resizing ends respectively. *resizing()* method is called many times during resizing. By default, *resizing()* method will set the preferred size of the component and cause the parent to *doLayout()*. However, depends on the parent, a simple doLayout() may not resize the component correctly. For example, if the parent is *JWindow*, a top level container, *doLayout()* will do nothing. In this case, you should subclass *Resizable* and override *resizing()* method to do something else. For example, in the case of JWindow, you just need to call *setBounds()* to change the size and location of JWindow.

```
protected Resizable createResizable() {
    return new Resizable(this) {
        public void resizing(int resizeDir, int newX, int newY, int newW, int newH) {
            ResizableWindow.this.setBounds(newX, newY, newW, newH);
        }

        public boolean isTopLevel() {
            return true;
        }
    };7
}
```

Several Resizeable Examples

To make *Resizable* easy to use, we create a *ResizablePanel*. It extends JPanel except it is resizable. In addition we also create two top level *Resizable*s – *ResizableWindow* and *ResizableDialog*. It makes sense to have *ResizableWindow* because JWindow is not resizable by

default. However, you may wonder why *ResizableDialog*. JDialog is resizable by default, but not when it is undecorated. So *ResizableDialog* is actually an undecorated *ResizableDialog*.

The usage of those classes are exactly the same JPanel, JWindow or JDialog respectively. All of them have *getResizable()* to get the underlying *Resizable*. You can get it and tweak some options such as ResizeCornerSize or ResizableCorners.

We heavily used those *Resizable*s in other part of our products and in our demos. For example, floating window in JIDE Docking Framework is using *ResizableWindow*. JidePopup/Alert is using *ResizableWindow* too. In the demo, *ResizablePanel* is used along inside *DocumentComponent* to make the demo area resizable.

Popup

The intention of developing JidePopup[9] component is to address the common features of any types of popup. Popup is something that appears above any other windows. However it is transient, meaning when you click outside the popup, the popup is gone. There are many examples of popup, such as tool tips, combobox popup, and popup menu. If further expanding the popup concept, there are even more examples, such as new email alert, the famous IntelliJ IDEA Ctrl-N popup[10].

Except the common feature of popup, each popup might have its own characters. For example, some could be resizable such as combobox popup (there is an example in *Resizable* components section). Some could be movable. Some support time out – it will hide automatically after several seconds for example. Some are always attached to the invoking component, such as combobox. Some are standalone such as email alert. Others might be attached to the invoking component at the beginning but can be detached by dragging, such as color split button you can see in MSOffice product. There is also a special category of popups which support animation when entrancing and exiting – either using fade effect, or flying in/out effect or using whatever animation effect you can think of. JidePopup is trying to capture all those different requirements and provide one solution for you.

JidePopup extends *JComponent*. You just used it as using any other *JComponent* by adding child components to it. *JidePopup* also supports RootPane which means you can also set a menu bar on it or use JLayeredPane or GlassPane. The only thing is you don't want to do is to add *JidePopup* to a container. To show it, you just call one of the *showPopup()* method. See below for an example. It will create a popup with an empty text area and a sample menu bar, then show the popup.

```
JidePopup popup = new JidePopup();
popup.setMovable(true);
popup.getContentPane().setLayout(new BorderLayout());
JTextArea view = new JTextArea();
view.setRows(10);
view.setColumns(40);
popup.getContentPane().add(new JScrollPane(view));
JMenuBar menuBar = new JMenuBar();
JMenu menu = menuBar.add(new JMenu("File"));
menu.add("<< Example >>");
menuBar.add(new JMenu("Edit"));
menuBar.add(new JMenu("Help"));
popup.setJMenuBar(menuBar);
```

[9] We named JidePopup just to avoid the name conflict with Swing's Popup, although these two are not quite related.

[10] You will understand what this means only if you use IntelliJ IDEA. For those who don't use IntelliJ IDEA, here is a short explanation. Ctrl-N in IDEA is hotkey for "Go to Class" where a "dialog" will popup. You can type in part of the class name and it will list all matches with that name so that you can quickly pick it and go to the class you want to go. This is probably the most used hotkey in the whole IntelliJ IDEA. When I said "dialog", it's not really a dialog although it looks like. Different from dialog, it doesn't block. When mouse clicks anywhere outside, the "dialog" is gone. This is exactly the "unstable" behavior of a popup. By the way, Alt-F1 is another popup example.

```
popup.setOwner(attachedButton);
popup.setResizable(true);
popup.setDetachable(true);
popup.setDefaultFocusComponent(view);
popup.showPopup();
```

Options

Owner: The owner or the invoker of this popup. If you show a popup in the actionPerformed of a button, the button should be the owner of this popup. If the popup is for combobox, the cmbobox should be the owner. There are several reasons we need this owner. In attached mode, the owner is the component that popup attaches to. When you call *showPopup()* without any parameter, it will place

Resizable: Resizable option is on by default. Depending on the detached/attached mode, the resizing behavior may be different. If a popup is detached to a component, it only allows you to resize from bottom, bottom right and right. It obviously doesn't make sense to resize from top and top side is aligned with the attached component.

Movable: If a popup is movable, it will show a gripper so that user can grab it and move the popup. If the popup is attached to its owner, moving it will detach from the owner first.

Detached: Detached is a flag to indicate if the popup is detached from owner or not. You shouldn't need to call *setDetached()* directly. If you call *showPopup()*, the detached will be true.

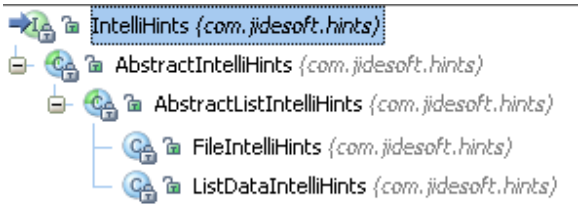
DefaultFocusComponent: DefaultFocusComponent is a component on popup. It will receive keyboard focus when popup is shown.

Timeout: *JidePopup* can hide itself after certain time. This can be controlled by setTimeOut(). You can pass in a value which is in millisecond. If you don't want the popup hide after the time out, set the value to 0. By default it's 0 meaning it will never time out.

IntelliHints

IntelliHints is a new name we invented to capture a collection of new features we introduce in 1.8.3 release. Similar features are called code completion or intelli-sense in the context of text editor. Without getting into too much detail, I encourage you running B14 example to see different flavors of IntelliHints. IntelliHints is designed to be extended. You can easily extend one of existing base IntelliHints classes such as AbstractIntelliHints or AbstractListIntelliHints or even implement IntelliHints directly to create your own IntelliHints.

See below for the class hierarchy of IntelliHints related class.



The base IntelliHints is an interface. It has four very basic methods about hints.

```
    /**
     * Creates the component which contains hints. At this moment, the content should be empty. Following
call
     * {@link #updateHints(Object)} will update the content.
     *
     * @return the component which will be used to display the hints.
     */
    JComponent createHintsComponent();

    /**
     * Update hints depending on the context.
     *
     * @param context the current context
     * @return true or false. If it is false, hint popup will not be shown.
     */
    boolean updateHints(Object context);

    /**
     * Gets the selected value. This value will be used to complete the text component.
     *
     * @return the selected value.
     */
    Object getSelectedHint();

    /**
     * Accepts the selected hint.
     *
     * @param hint
     */
    void acceptHint(Object hint);
```

AbstractIntelliHints implements *IntelliHints*. It assumes the hints are for a *JTextComponent* and provides a popup using JidePopup to show the hints. However it has no idea what components the popup contains. Since in most cases, the hints can be represented by a JList, here comes the *AbstractListIntelliHints*. This class assumes JList is used to display hints in the popup and implements most of the methods in IntelliHints except *updateHints()* methods. That's why it is still abstract. Whatever classes that extend *AbstractListIntelliHints* should implement *updateHints()* method and set the list data to the JList.

There are two concrete implementations we made in current release – *FileIntelliHints* and *ListDataIntelliHints*. *FileIntelliHints* provides hints based on file system. *ListDataIntelliHints* provides the hints based on a known list. See below. The first one is *FileIntelliHints*. The list contains the files and folders that match what user typed in so far.

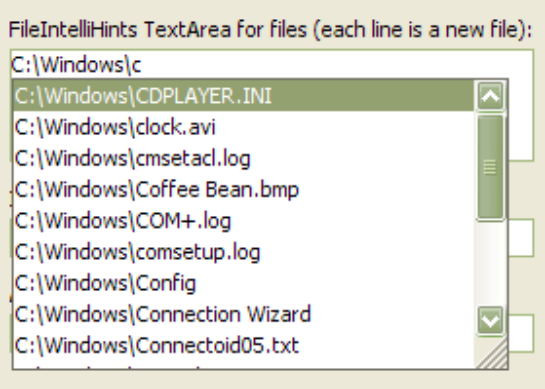


Figure 6 FileIntelliHints

It's very easy to create one.

```
JTextField pathTextField = new JTextField();
FileIntelliHints intelliHints =  new FileIntelliHints(pathTextField);
intelliHints.setFolderOnly(true);
```

Below is an example of *ListDataIntelliHints*. It provides hints based on what you typed in so far to filter a known list, and only shows those that match what you typed in.

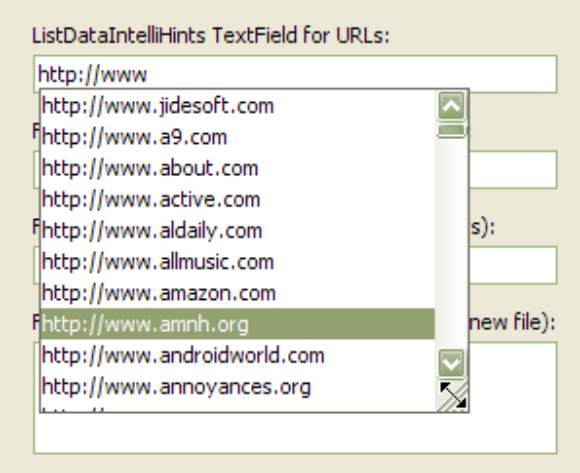
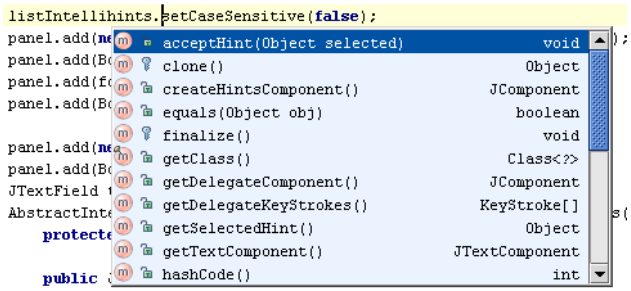


Figure 7 ListDataIntelliHints

Here is the code to create the ListDataIntelliHints above.

```
JTextField urlTextField = new JTextField("http://");
ListDataIntelliHints intellihints = new ListDataIntelliHints(urlTextField, urls);
intellihints.setCaseSensitive(false);
```

IntelliHints can easily be extended. If you can use JList to represent the hints, you can extend *AbstractListIntelliHints*. For example, if you want to implement code completion as in any IDE like below, *AbstractListIntelliHints* should be good enough for you. Like to do what's in the screenshot below, all you need to do is to override *createList()* method in *AbstractListIntelliHints* and set a special list cell renderer.



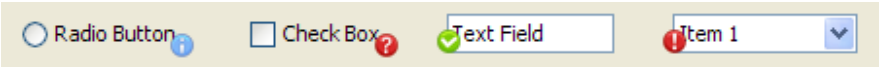
If your hints are more complex and can't be represented by JList, you will have to extend *AbstractIntelliHints* and create your own content for the popup.

IntelliHints is very useful usability feature. If you use it at the right places, it will increase the usability of your application significantly. Just imagining how depending you are on the code-completion feature provided by your Java IDE, why not provide a similar feature to your end users as well? They will appreciate it. With the help of IntelliHints, it's not far away.

Overlayable

The overlayable feature provides a way to put a component on top of another component. A typical usage is to display a small "x" icon on the corner of the component to indicate a validation error. However, the overlayable feature is much more useful than this.

Here is a screenshot of overlay component on several Swing controls.



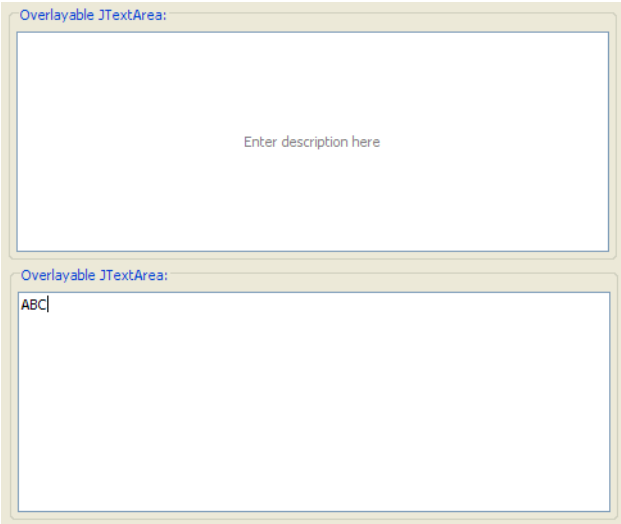
The overlay is a real component, not just a painted image. So it supports tooltip, mouse listener etc just like a regular component. This is very important, as developer always want to associate an action with the overlay component.

To make it easy for you, we included the following icons as part of the package. You just need to call OverlayableIconsFactory.getImageIcon(FULL_CONSTANT_NAME) to get the icon.

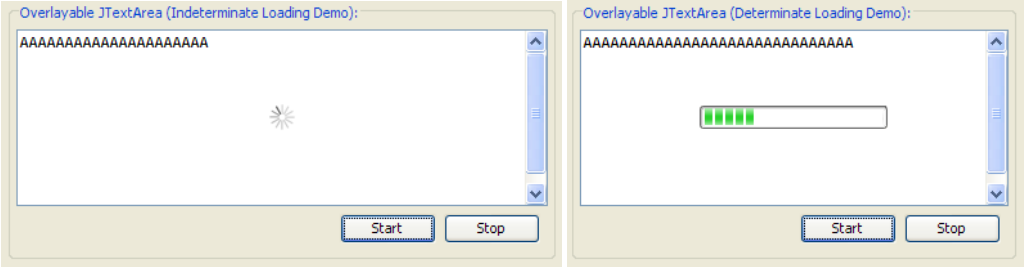
OverlayableIconsFactory

Name	Image	File Name	Full Constant Name
ATTENTION		icons/overlay_attention.png	OverlayableIconsFactory.ATTENTION
CORRECT		icons/overlay_correct.png	OverlayableIconsFactory.CORRECT
ERROR		icons/overlay_error.png	OverlayableIconsFactory.ERROR
INFO		icons/overlay_info.png	OverlayableIconsFactory.INFO
QUESTION		icons/overlay_question.png	OverlayableIconsFactory.QUESTION

Here is a way to provide a description to a JTextArea (or JTable, JTree etc) using *Overlayable*. The label “Enter description here” is an overlay component. You can control when to show and hide the overlay component. In this example, when the JTextArea gains focus, we will hide the overlay component.



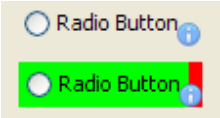
Here is one more way to use this feature. See screenshot below, we put a progress spin (marked with the red arrow) over a JTextArea (picture on the left). You can add a real JProgressBar as the overlay component (picture on the right).



How to use the API

Overlayable is the interface to make something overlayable. Instead of making every component overlayable, which will change too many classes, we decide to create a default implement that makes a JPanel overlayable. For example, you want to add an overlay component to a check box. You can simply add the check box to this overlayable panel, and then add overlay components to this overlayable panel. It looks like the overlay component is on the check box although it is actually on the check box parent.

Here is an example of an overlay component on a radio button.



The top one shows what it looks like. The icon seems like part of the radio button but it is not. As you can see from the bottom screenshot, the green rectangle is the boundary of the radio button. The red rectangle (plus the green rectangle as the green paints over the red) is the boundary of the overlayable panel. The icon is on the bottom right corner of the overlayable panel, not the radio button.

Comparing the code change

Before adding the overlay component, we have code like the below. The controlPanel is the panel that contains the radio button.

```
controlPanel.add(new JRadioButton("Radio Button"));
```

If you want to add an icon as overlay component, we need to create a label first.

```
JLabel info = new JLabel(OverlayableUtils.getPredefinedOverlayIcon(OverlayableIconsFactory.INFO));
```

Then we need to wrap the radio button to a DefaultOverlayable. We also need to override a method in radio button to repaint the overlay component correctly. The code will be like below. It is a little too complex.

```
controlPanel.add(new DefaultOverlayable(new JRadioButton("Radio Button"){
        public void repaint(long tm, int x, int y, int width, int height) {
          super.repaint(tm, x, y, width, height);
          OverlayableUtils.repaintOverlayable(this);
        }
      }, info, DefaultOverlayable.SOUTH_EAST));
```

Or, if you use one of the pre-build radio button, you can save the overridden method. *OverlayRadioButton* is nothing but a *JRadioButton* that overrides the repaint method as shown above.

```
controlPanel.add(new DefaultOverlayable(new OverlayRadioButton("Radio Button"), info,
DefaultOverlayable.SOUTH_EAST));
```

The code is still more complex than the original code. But considering the powerful feature it added, it is worth the added complexity.

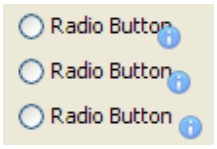
Adding multiple overlay components

Overlayable supports multiple overlay components. *DefaultOverlayable*'s constructor can take one overlay component. But you can still add more by calling *addOverlayComponent()*. For each overlay component, you can control the position, the order relative to other overlay components and visibility independently. *removeOverlayComponent()* will remove it and *getOverlayComponents()* will tell you all the overlay components.

Putting overlay components beyond the component

Overlayable also has *setOverlayLocationInsets()*. We noticed many other implementation has the limitation that the overlay component must be within the boundary of the component itself. This is annoying as the overlay component might cover portion of the component. That is why we added this overlayLocationInsets concept. If you want to place the overlay component outside the east border, you just give a positive number on the east edge of the insets.

See below for an example. The first one has 0 on the east edge; 5 for the second one and 10 for the last one.



Advantages and disadvantages

When we design the overlayable components, we have the following criteria in mind.

1. API ease of use – least code change to add an overlay component
2. API easy to understand

3. The overlay component is a real component, not just a painted image so that user can add mouse listener to it or set tooltip etc.
4. Can be placed beyond the component boundary
5. Handle scroll pane well[11]
6. Support any LookAndFeels without extra code.
7. Can add overlay component to any component
8. Can use any component as the overlay component

We knew many different ways[12] to implement this feature. However, after we look at the criteria above, we ruled out many of the alternatives. JLayeredPane/GlassPane is ruled out because of bullet 5. Overriding paint method approach is ruled out because of bullet 3 and 4. Extending or multiplex L&F approach is ruled out because of bullet 6 and 7. Finally, we come up with this design. I want to point out, although it satisfies almost all the criteria, it is still not perfect especially we still have to override repaint method. One way to solve it is to provide our own RepaintManager but it will probably make API harder to understand. If Swing provided a hook into RepaintManager, it would be perfect. So in conclusion, if we would give a rating to this design from 1 to 5 with 5 is the best, I will give 5 for bullet 3 to bullet 8 and give 3 to bullet 1 and 2. There is still room to improve in these two bullets.

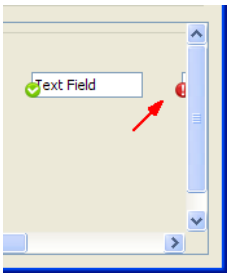
Layout

JideBoxLayout

As its name indicates, the JideBoxLayout class is similar to Swing's BoxLayout.

Similar to BoxLayout, JideBoxLayout lays components out either vertically or horizontally. Unlike BoxLayout however, there is a constraint associated with each component, set to either FIX, FLEXIBLE, or VARY. If the constraint is set to FIX then the component's width (or height if the

[11] The screenshot below shows how it should behave inside a scroll pane. If you use using JLayerPane, you will see the error icon is painted above the scroll bar, which is wrong.



[12] It is worth reading the blog of Kirill Grouchnikov at http://www.pushing-pixels.org/?p=110. He has a series of blogs on how to support validation overlay.

JideBoxLayout is vertical) will always be the preferred width. By contrast, although FLEXIBLE components try to keep the preferred width, they will shrink proportionally if there is not enough space. Finally, VARY components will expand in size to fill whatever width is left. Although you can add multiple FIX or FLEXIBLE components, only one VARY component is allowed.

Code Example 1:

This sample has three buttons; the first one is FIX and the second and third ones are FLEXIBLE.

```
JPanel panel = new JPanel();
panel.setLayout(new JideBoxLayout(panel, 0, 6));

JButton button = new JButton("FIX");
button.setPreferredSize(new Dimension(60, 60));
panel.add(button, JideBoxLayout.FIX);

button = new JButton("FLEX1");
button.setPreferredSize(new Dimension(120, 60));
panel.add(button, JideBoxLayout.FLEXIBLE);

button = new JButton("FLEX2");
button.setPreferredSize(new Dimension(120, 60));
panel.add(button, JideBoxLayout.FLEXIBLE);
```

Original:

After resizing:

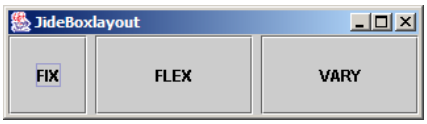
Code Example 2:

This example has one FIX button, one FLEXIBLE button, and one VARY button.

```
JPanel panel = new JPanel();
panel.setLayout(new JideBoxLayout(panel, 0, 6));

JButton button = new JButton("FIX");
button.setPreferredSize(new Dimension(60, 60));
panel.add(button, JideBoxLayout.FIX);

button = new JButton("FLEX");
```

```
button.setRequestFocusEnabled(false);
button.setPreferredSize(new Dimension(120, 60));
panel.add(button, JideBoxLayout.FLEXIBLE);

button = new JButton("VARY");
button.setPreferredSize(new Dimension(120, 60));
panel.add(button, JideBoxLayout.VARY);
```

Original:



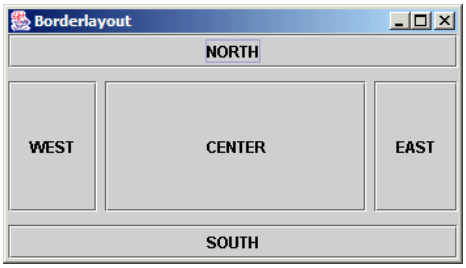
After resizing:

After resizing again:

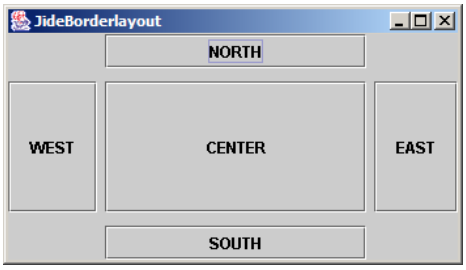
JideBorderLayout

JideBorderLayout is almost the same as the standard Swing BorderLayout except that the NORTH and SOUTH component's width is the same as the CENTER component, as shown overleaf. Please note the different between BorderLayout and JideBorderLayout.

In AWT BorderLayout, the north and south components take *all* of the horizontal space that is available.



By contrast, in JideBorderLayout the north and south components only take the same horizontal space as the center component.



IMAGES and ICONS Related CLASSES

ColorFilter and GrayFilter

A disabled button will normally display a disabled icon. However it's a pain to create two icons for each button. Why not just pass in the normal icon and let the Jide framework create a disabled icon for you?

Image ColorFilter.createDimmedImage(Image)

Image GrayFilter.createDisabledImage(Image)

This is the effect of above methods

Normal	Dimmed Image	Disabled Image
		

IconsFactory

In Java/Swing, you can load an image file as a disk file or as a resource. We found that it's easier and faster to load image files as resources. This class is designed to encourage the use of images and icons as resources

The IconsFactory acts as a cache manager for ImageIcons and has three static methods:

```
public static ImageIcon getImageIcon(Class clazz, String fileName);
public static ImageIcon getDisabledImageIcon(Class clazz, String fileName);
public static ImageIcon getBrighterImageIcon (Class clazz, String fileName);
```

Each time you call the method, the icon that is returned will be kept in a cache.

In addition to the points mentioned above, IconsFactory also has a special usage: Applications typically use hundreds of icons and images. Management of these objects can easily get out of control. In addition, you might have issues such as duplicate icons, inconsistent use of icons, difficulty in locating the right icon etc. However with the help of IconsFactory, these issues become much less of a problem.

In the release, there is a class called VsnetIconsFactory.java[13], which looks like this:

```
public class VsnetIconsFactory {

    public static class ClassElement {
        public final static String CLASS = "vsnet/msdev_class_class.gif";
        public final static String FIELD = "vsnet/msdev_class_field.gif";
        public final static String FIELD_PROTECTED = "vsnet/msdev_class_field_protected.gif";
        public final static String FIELD_PRIVATE = "vsnet/msdev_class_field_private.gif";
        public final static String METHOD = "vsnet/msdev_class_method.gif";
        public final static String METHOD_PROTECTED = "vsnet/msdev_class_method_protected.gif";
        public final static String METHOD_PRIVATE = "vsnet/msdev_class_method_private.gif";
        public final static String CONSTANT = "vsnet/msdev_class_const.gif";
        public final static String MAP = "vsnet/msdev_class_map.gif";
        public final static String GLOBAL = "vsnet/msdev_class_global.gif";
    }
```

.....

```
    public static void main(String[] argv) {
        IconsFactory.generate(VsnetIconsFactory.class);
    }
}
```

If you follow this pattern to create your own Icons Factory, you will get two benefits:

- The first is the handy display you see below. Looking at the listing of VsnetIconsFactory above, notice that there is a 'main' method. Run it and an html file will be generated in the current directory, as shown in the example below. It will have a list of all icons in the factory, organized into different sections as a table. In the table, you can see what the icons look like, what the actual image file names are, and how to use them in the code. Developers should never get lost!

Icons in com.jidesoft.icons.VsnetIconsFactory

Generated by JIDE Icons

1. If you cannot view the images in this page, make sure the file is at the same directory as VsnetIconsFactory.java

2. To get a particular icon in your code, call VsnetIconsFactory.getImageIcon(FULL_CONSTANT_NAME). Replace FULL_CONSTANT_NAME with the actual full constant name as in the table below

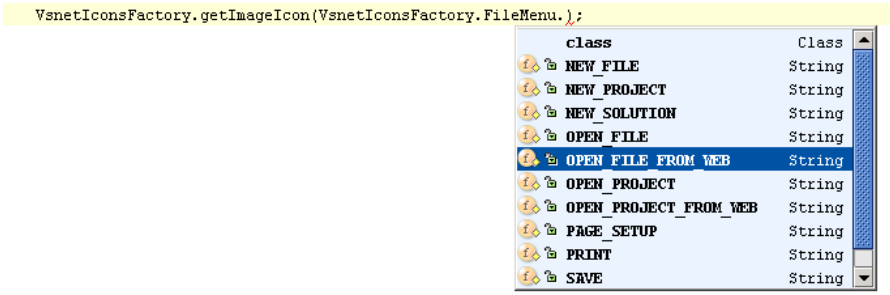
VsnetIconsFactory.ClassElement

Name	Image	File Name	Full Constant Name
CLASS		vsnet/msdev_class_class.gif	VsnetIconsFactory.ClassElement.CLASS
FIELD		vsnet/msdev_class_field.gif	VsnetIconsFactory.ClassElement.FIELD
FIELD_PROTECTED		vsnet/msdev_class_field_protected.gif	VsnetIconsFactory.ClassElement.FIELD_PROTECTED
FIELD_PRIVATE		vsnet/msdev_class_field_private.gif	VsnetIconsFactory.ClassElement.FIELD_PRIVATE
METHOD		vsnet/msdev_class_method.gif	VsnetIconsFactory.ClassElement.METHOD
METHOD_PROTECTED		vsnet/msdev_class_method_protected.gif	VsnetIconsFactory.ClassElement.METHOD_PROTECTED
METHOD_PRIVATE		vsnet/msdev_class_method_private.gif	VsnetIconsFactory.ClassElement.METHOD_PRIVATE
CONSTANT		vsnet/msdev_class_const.gif	VsnetIconsFactory.ClassElement.CONSTANT
MAP		vsnet/msdev_class_map.gif	VsnetIconsFactory.ClassElement.MAP
GLOBAL		vsnet/msdev_class_global.gif	VsnetIconsFactory.ClassElement.GLOBAL

.....

- The second benefit is that with the help of IntelliSense in most Java IDEs, you can easily locate an icon right in your editor. See overleaf for a screenshot from IntelliJ IDEA when using IconsFactory.

[13] VsnetIconsFactory is just for tutorial purpose to teach you how to create an IconsFactory. Please do not use any icons from VsnetIconsFactory in your applications because they are copyrighted by Microsoft.



Internationalization Support

All of the Strings used in *JIDE Common Layer* are contained in properties files

Note that we haven't done any localization: if you want to support languages other than English, just extract the properties file, translate it to the language you want, add the correct language postfix and then jar it back into the jide jars. You are welcome to send the translated properties file back to us if you want to share it!