Having Fun with StyledLabel

I'll start with a question. Using Java Swing, how do you implement something like these?

All three examples are very common in a real application.

1. The first one is to show a ™ trademark sign next to Java. It is the same thing for a ® (registered trademark) sign.
2. The second one is a math formula.
3. The third one indicates the text has an error.

We want to use both the red color and a waved underline to indicate the error. Keep in mind, from accessibility point of view, only the red color is not enough because of color blindness.

Possible Solution 1: Using multiple JLabels

For Java™, the first thing that comes to my mind is to use two JLabels. In the case of Java™, "Java" could be one JLabel. "TM" could be another JLabel which has a smaller font. Then somehow we can place them in a layout manager so that it looks like Java™. Here is the code I came up with:

```
JLabel javaLabel = new JLabel("Java");
JLabel tmLabel = new JLabel("TM");
tmLabel.setFont(tmLabel.getFont().deriveFont(8.0f));
tmLabel.setAlignmentY(Component.BOTTOM_ALIGNMENT);
JPanel labelPanel = new JPanel();
labelPanel.setLayout(new BoxLayout(labelPanel, BoxLayout.X_AXIS));
labelPanel.add(javaLabel);
labelPanel.add(tmLabel);
labelPanel.add(Box.createGlue());
```

Here is what it looks like:

Figure 1 using two JLabels

For the math formula one, this approach is getting tedious. For every small "2", we have to create a new JLabel so it will end up with seven JLabels. Not fun at all. I don't think I want to write the code for it.

For the last waved underline one, JLabel just can't do it. First, you will create a separate JLabel in order to show the red color. Then you will have to override

paintComponent method of JLabel and draw a waved underline. In real words, this example could be one sentence with multiple errors.

Conclusion: It works for the first two but it's way too complicated. Nearly impossible to do for the last one.

Possible Solution 2: Using html in JLabel

JLabel supports html tags. Here is what I tried and it works perfectly for the first two.

```
new JLabel("<html>Java<sup>TM</sup></html>");
new JLabel("<html>x<sup>2</sup> + z<sup>2</sup> = z<sup>2</sup></html>");
```

Here is what it looks like:

Figure 2 using html in JLabel

Unfortunately for the last one, the red text is not a problem for html but the waved underline is as the standard html tag doesn't support. Html supports <u> tag for a single line underline. Again, you will have to override paintComponent method to do it.

It looks like we got a solution, at least for the first two examples. Wait until you test out the performance. It turned out a JLabel using html is 20 to 40 times slower than a plain JLabel. It's probably okay if you just have one or two html JLabels. If you plan to use JLabels in a JTable that has hundreds and thousands of rows, it's going to be tough. Furthermore, JLabel with html is very buggy. There's a bug reported back in 2000 saying the tree node just vanishes when using html JLabel as a cell renderer in a JTree.

Conclusion: It works for the first two examples, and the code is very simple. Hard to do for the last example. However the performance and the bug are two main concerns.

COPYRIGHT © 2002-2011 JIDE SOFTWARE, INC. ALL RIGHTS RESERVED

Possible Solution 3: using JTextPane

There is actually a way to add some styles to text using JTextPane. The code to do it is complex, way more complex than using html.

```
SimpleAttributeSet superstript = new SimpleAttributeSet();
superstript.addAttribute(StyleConstants.CharacterConstants.Superscript, Boolean.TRUE);
document.insertString(document.getLength(), "Java", null);
document.insertString(document.getLength(), "TM", superstript);
JTextPane textPane = new JTextPane(document);
textPane.setOpaque(false);
```

Here is what it looks like:

Figure 3 using JTextPane

Again, it can't do the waved underline either just like the html JLabel. And it will suffer the same performance issue because all the complexity involved in JTextPane.

Conclusion: It works for the first two examples, but the code is complicated. It's still hard to do for the last example. And the performance is still a concern.

Final Solution: StyledLabel
(available in the open source JIDE Common Layer)

Before we decided to write a new component, we considered all three possible solutions above. Even though the examples are somewhat simple and common text display requirements, none of solutions above worked well.

Now let's introduce the StyledLabel. StyledLabel extends JLabel. It has its own UI class called BasicStyledLabelUI which extends BasicLabelUI. In BasicStyledLabelUI, we override the corresponding paint method to add styles to the plain text.

StyledLabel supports many styles. It's like a Swiss Army Knife for displaying text in Swing. Here is the list of features and styles:

- **Font**: plain, bold, italic, bold-italic
- **Text position**: superscript, subscript
- **Line stroke style**: strikethrough, double strikethrough, single underline, dotted underline, waved underline, or define your own line stroke pattern
- **Color**: foreground, background and line
- **Line wrap**: support automatic line wrap (soft line break) and \n, \r (hard line break)

StyledLabel works with StyleRange which defines styles for a range of the text along with the styles. For example, to display Java™, here is the sample code:

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

The code is kind of similar to the JTextPane because the idea is the same. Both allow you to define styles for a range of the text. But the code is already much simpler than that of JTextPane (only 2 lines v.s. 6 lines using JTextPane).

Please note, the range cannot overlap or nest. In the other word, the styles for a range must be the same. We designed it this way to make it simple. Otherwise we will have to consider the style overriding issue which will probably hurt both usability of the APIs as well as the performance.

StyledLabel Annotation

Even though it may seem really simple, we made it even simpler by introducing StyledLabel annotation. You can annotate a String using the grammar we defined. It's similar to html tags but, again, simpler. Here are the annotated strings for the three examples above:

```
Java{TM:sp}
x{2:sp} + y{2:sp} = z{2:sp}
this is an {error:w, f:red}
```

To create a new StyledLabel using annotated string, you can use:

```
StyledLabel label = StyledLabelBuilder.createStyledLabel("Java{TM:sp}");
```

To change an existing StyledLabel, you can use:

```
StyledLabelBuilder. setStyledText(styledLabel, "x{2:sp} + y{2:sp} = z{2:sp}");
```

It is just one line. Nothing could be simpler than this. Here is a cheat sheet that might be useful:

COPYRIGHT © 2002-2011 JIDE SOFTWARE, INC. ALL RIGHTS RESERVED

Help

Font styles:

- **plain** or **p**, i.e. {plain text:p} => plain text
- **bold** or **b**, i.e. {bold text:b} => **bold text**
- **italic** or **i**, i.e. {italic text:i} => *italic text*
- **bolditalic** or **bi**, i.e. {bold and italic text:bi} => ***bold and italic***

Line styles:

- **strike** or **s**, i.e. {strikethrough:s} => strikethrough
- **doublestrike** or **ds**, i.e. {double strikethrough:ds} => double strikethrough
- **waved** or **w**, i.e. {waved:w} => waved
- **underlined** or **u**, i.e. {underlined:u} => underlined
- **dotted** or **d**, i.e. {dotted:d} => dotted

Text Position:

- **superscript** or **sp**, i.e. Java{TM:sp} => Java^{TM}
- **subscript** or **sb**, i.e. CO{2:sb} => CO_2

Using Colors: using **f** for font color, **l** for line color and **b** for background color

- **f:** plus color name defined in class Color, i.e. {red text:f:red} => red text
- **l:** plus color name defined in class Color, i.e. {red underline:u, l:red} => red underline
- **b:** plus color name defined in class Color, i.e. {red background:b:red} => red background
- **f:** or **l:** or **b:** plus #RRGGBB, i.e. {any color:f:#00AA55} => any color
- **f:** or **l:** or **b:** plus #RGB as in CSS, i.e. {any color:f:#0A5} => any color
- **f:** or **l:**or **b:** plus (R, G, B), i.e. {any line color:s, l:(0, 220, 128)} or {any background color:b:(0, 120, 128)} => any line color or any background color

Special characters:

- Special annotation characters **{ } ( ) # : ,** should be escaped by "\" when they are used as regular text
- i.e. {\{brace\}:b} => **{brace}**
- If you need multiple styles connect them with **,** (comma), i.e. {bold red text:f:red, b} => **bold red text**

Global flags:

- The rows: you can use "rows", "row", or "r", followed by up to three parameters separated by ":" which are preferred row count, minimum row count and maximum row counts respectively.
- The preferred width: You can use "preferredWidth", "width", or "w"
- Use "@" at the end of the string to start definition of global flags. i.e. "this is a long text@rows", "this is a long text@r:2:1:4".
- We may add more global flags in the future

Figure 4 StyledLabel Annotation Cheatsheet

Performance

I mentioned that the performance is one of the key aspects but I haven't given you the hard data. Here is what we tried to benchmark the performance:

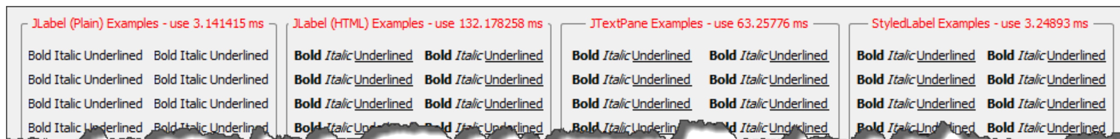

Figure 5 Performance Comparison

The benchmark is done by creating 100 of the tested components. We created a tested component first before the benchmarking starts so the initial class loading time is not included in the test result.

For the first column, they're just plain JLabels. It doesn't have the bold/italic effect we want but that's fine as we used this one as the base benchmark to compare with. You can see the time used to create 100 plain JLabels is a little over 3 ms.

Html JLabels are 42 times slower than plain JLabels, taking 132 ms iv.s 3 ms. JTextPane is actually faster than html JLabel but still about 20 times slower than plain JLabel.

For the StyledLabel, it took 3.25 ms which is only slightly more than the 3.14 ms taken by the plain JLabel.

Line Wrapping

One of the cool additions we added recently to StyledLabel in the latest 3.2.x release is the line wrapping feature. If you use Swing long enough, you would know it's a headache to show a multiple lines label without seeking help from JTextArea (which is what our MultilineLabel extends). You can use html but you have to hard code the line breaks. If using JTextArea, you could change the background, disable the mouse interaction to make JTextArea looks like a JLabel. However the problem with JTextArea is the preferred size calculation.

Yet again, StyledLabel is here to salvage. By a simple call setLineWrap(true), the text will wrap automatically based on the available width without you needing to explicitly add line breaks to the text. You can independently control text layout by setting the preferred, minimum and maximum row counts, as well as the preferred width.

Annotation is supported as well. You can append "@row:2:1:5" to the end of the annotated string. What it means is default to 2 rows, minimum 1 row and maximum 5 rows. Or you can use "@row:2::5" if you don't really care about minimum rows or just "@row:2" if you just care about the default row count.

This is a demo to show a style label that is able to wrap automatically. It is a long text to be wrapped. By default, it should occupy 3 rows. It can only be wrapped to between 2 and 5 rows.You could resize the panel to see how it automatically resizes.

Figure 6 StyledLabel with line wrapping

If you need to use hard line breaks, you can do it too. Just insert "\r\n" in your text, and a line break will appear there without affecting the automatic line wrapping.

What if the rows is greater than the maximum rows that can be shown? No problem because a "..." will appear at the end to indicate more text just like a JLabel does. Will JTextArea do it? No.

This is a demo to show a style label that is able to wrap automatically. It is a long text to be wrapped. By default, it should occupy 3 rows. It can only be wrapped to between 2 and 5 rows.You could resize the panel to see h...

Figure 7 Line wrapping with ... at the end

COPYRIGHT © 2002-2011 JIDE SOFTWARE, INC. ALL RIGHTS RESERVED

Using StyledLabel as a Text Painting Engine

As you can see, StyledLabel is packed with features that are not available in the standard Swing. You will soon find many places that StyledLabel could be used but cannot be used as a JComponent. For example, you want to show a StyledLabel in a table header to support multiple lines; but, obviously you can't place a StyledLabel there. There is a Swing trick that is not well-known even though it is used widely in Swing itself. JComponent has a paint method that takes a Graphics parameter. So you can get the Graphics of any other component and call styledLabel.paint(g) to paint the StyledLabel to the other component. In the other word, you can use StyledLabel as a text painting engine to paint something you want from inside another component.

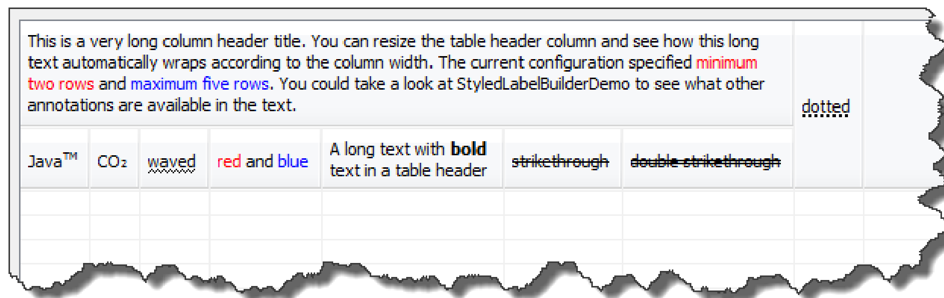

Figure 8 using StyledLabel to paint the table header text

To use it, all you need to do is to return annotated string from getColumnName method of your TableModel. Any table headers that extend CellStyleTableHeader will be able to support the annotation and renderer it correctly using StyledLabel.

```
public String getColumnName(int column) {
   switch (column) {
      case 0:
         return "Java{TM:sp}";
      case 1:
         return "CO{2:sb}";
      case 2:
         return "{waved:w}";
      case 3:
         return "{red:f:red} and {blue:f:blue}";
      case 4:
         return "A long text with {bold:b} text in a table header@r:2:1:3";
      case 5:
         return "{strikethrough:s}";
      case 6:
         return "{double strikethrough:ds}";
      case 7:
         return "{dotted:d}";
   }
   return "";
}
```