HIGHLIGHTS OF A DOCUMENT

James W. Cooper

Sometimes you need to be able to display text in several fonts and colors to illustrate particular points. Frequently the easiest way to do this is to preprocess the document text into HTML and launch a browser instance to display that text. You can also emit XML or DHTML to gain a little more control over the final appearance. However, if you want the user to interact with a program that displays text in some entertaining fashion, you may find that HTML and JavaScript just can't do the job and turn to the complex but ultimately satisfying approaches allows in Java.

Recently I had just a problem. I realized that selection of salient terms in a long document as well as partially decoded encrypted text were candidates for a display in which only some words are easily visible and highlighted, while the intervening words may be inaccurate or incorrect and need to be displayed in a more conservative or less readable fashion.

In order to implement this display, I had to understand how to use the

1. JTextPane
2. StyledDocument and DefaultStyledDocument
3. AttributeSet and SimpleAttributeSet
4. StyleConstants
5. Highlighter and HighlightPainter

classes. This was a bigger learning experience than I had expected and I thought I'd tell you about some simple ways through this thicket. Then, you'll be able to write programs that display and highlight text in various fonts, styles and colors. Since a number of things that should have worked did not work in Java 1.2.2, I'll point those out as well.

The JTextPane and the Styled Document

Java allows you to use a JTextPane to display text directly or display text in a number of styles by setting a StyledDocument for the text pane. Like many of the other Swing objects, StyledDocument is an interface and DefaultStyledDocument is a basic implementation of that interface. We'll just derive our own LimitedStyledDocument from that default implementation and use it with only a couple of changes. Some of this program is based on an example on the JavaSoft web site, but it has been extensively rewritten to improve its object orientation and add highlighting. Most of the classes we are discussing here are in the java.swing.text package.

The StyledDocument object has one important method: *insertString*, which inserts a string at an offset in the document with particular attributes.

```
public void insertString(int offs, String str, AttributeSet a)
```

These attributes are the ones that control the font size, style and color and are instances of the SimpleAttributeSet class. The trouble with this approach is that there are a lot of different classes to keep track of, and designing a set of simple objects that encapsulate these classes is somewhat involved. we'll end up encapsulating a lot of the complexity into a Word class, although there may be other equally good solutions.

The AttributeSet interface is designed to provide a way to have an open-ended series of named attributes for any display system. At the simplest level, these attributes are undefined and can be implemented as a Hashtable. However, we would prefer not to have to invent and implement these ourselves. We avoid this by using the static methods of the StyleConstants class which let you refer to the various font attributes by common method names and leave the hash coded names and implementation buried beneath the surface. So, using these methods, we can set an AttributeSet font size, for example, using a set of obvious methods without ever knowing how this is actually accomplished:

```
SimpleAttributeSet attr;

public void setBold(boolean b) {
      StyleConstants.setBold(attr, b);  //set the style to Bold
  }
```

Then , you use this configured instance of the SimpleAttributeSet when you call the *insertString* method.

Highlighting Text

You might think you could just set the Background property of text in a StyledDocument to get it to display in a different color, but this is not only incorrect, it doesn't work either. The *setBackground* method overwrites the text in the same color, so it doesn't display any text at all. Instead, you have to use a Highlighter object and a Highligher.HighlightPainter object to accomplish this.

Each JTextPane has a *getHighlighter* method that allows you to obtain the Highlighter object for that pane. You can add a number of highlighted text areas to the Highlighter object, specified by character offset, so that they are highlighted regardless of the shape or resizing of the text window.

```
Highlighter hlt = textPane.getHighlighter ();
hlt.addHighlight (offset, length, hlp);
```

The final kicker is the HighlightPainter interface. You must provide the class that implements the painting of the background color. This interface requires that you only implement the one method:

```
public void paint (Graphics g, int p0, int p1, Shape bounds, JTextComponent c)
```

where p0 is the starting offset, p1 is the ending offset, the Shape represents the area to be painted and the JTextComponent is the text pane you will be drawing on. I found two problems with the Java 1.2.2 implementation of this method:

- The offset p1 is always the end of the document rather than the end of the string.
- The Shape object contains no useful information at all.

So we have to write a HighlightPainter which draws the background without making use of some of the information we expected to find. This was where I decided that the only way to carry all this off was to create a Word object for each individual word we displayed and use it to store some of this information we have to carry around. We show that complete Word object in Listing 1.

```
import java.awt.*;
```

```
import javax.swing.*;
import javax.swing.text.*;

//This is an encapsulation of a string to be displayed in a
//styled document. It contains the highlight information
//as well as the cumulative offset of that word in the text.
public class Word {
  private String word;           //the text
  private SimpleAttributeSet attr; //the attributes
  private boolean hilite;        //whether highlighted
  private int offset;            //the cumulative offset

  public Word(String w) {
      word = w.trim() + " ";
      //create a simple attribute and set its font and size
      attr = new SimpleAttributeSet();
      StyleConstants.setFontFamily(attr, "SansSerif");
      StyleConstants.setFontSize(attr, 9);  //default size

      //look for underscores to indicate highlighted words
      int i = word.indexOf ("_");
      //if highlighted, make the font bigger, bold and red
      if (i > 0)   {
        setColor(Color.red);    // color
        setFontSize(12);        //and size
        hilite = true;
        setBold(true);          //make it bold
      }
      //Remove the underscores
      while (i > 0) {
        StringBuffer buf = new StringBuffer(word);
        buf.setCharAt (i, ' ');
        word = buf.toString ();
        i = word.indexOf ("_");
      }
  }
  //save the offset for fast recovery
  public void setOffset(int offs) {
      offset = offs;
  }
  //return the stored offset
  public int getOffset() {
      return offset;
  }
  //return whether word is to be highlighted
  public boolean isHighlighted() {
      return hilite;
  }
  //get the word
  public String getWord() {
      return word;
  }
  //get the word length
  public int length() {
      return word.length ();
  }
  //set the font size
  public void setFontSize(int fsz) {
      StyleConstants.setFontSize(attr, fsz);
  }
  //set the color
  public void setColor(Color c) {
      StyleConstants.setForeground(attr, c);
```

```
  }
  //set whether italic
  public void setItalic(boolean b) {
      StyleConstants.setItalic(attr, b);
  }
  //set if bold
  public void setBold(boolean b) {
      StyleConstants.setBold(attr, b);
  }
  //get the attribute as it is currently set
  public SimpleAttributeSet getAttributes() {
      return attr;
  }
}
```

Listing 1. The Word object.

The Word class in Listing 1 takes one word at a time and sets appropriate attributes for it. If the word does not include an underscore, the attributes are set to a font size of 9 points. If the word token contains an underscore the font size is set to 12, boldface and red. The boolean *hilite* is set to *true* so the *isHighlighted* method will return true.

The Word List

As we read in words, we create Word objects and keep them in a list to be referred to when we display the text and highlights. Since the new ArrayList class has slightly better performance than the Vector class, we'll use it here. Now, we'll want to add words to the list, fetch them by index or offset and find out if they are to be highlighted., and find out their length. We are thus creating a list whose properties are best encapsulated into a class. Since a both the initial display class and the HighlightPainter class will need to fetch these words, we'll put the entire array store and fetch structure into a enclosing class. We could just consider this an aggregation class, but since we need to get at the data in different ways, we can just about consider it a Mediator class as well. So, we'll name it a Mediator class as shown in Listing 2.

```
public class Mediator {
//This class hides the structure of the word list
//and provides access to it simply
    private ArrayList words;
    private int offset;
    private int i;

    public Mediator() {
        words = new ArrayList();
        offset = 0;
    }
    //add a word to the list
    public void add(Word w) {
        w.setOffset (offset);
        words.add (w);
        offset += w.length ();
    }
    //get the word at index i
    public Word get(int i) {
        return(Word) words.get (i);
    }
    //find the word at the offset specified
    public Word findWord(int offs) {
        i = 0;
        Word w = get(i++);
```

```
        while (w.getOffset () < offs && i < words.size ()) {
            w = get(i++);
        }
        w= get(i - 1);
        return w;
    }
    //find the word before the specified offset
    public Word findLastWord(int offs) {
        findWord(offs);
        return get(i-2);
    }
    //get the size of the word list
    public int size() {
        return words.size ();
    }
}
```

Listing 2 – The Mediator class containing the Word list

Loading the Word List and Highlight List

We read in the words, a line at a time, using the StringTokenizer to separate words. As we create each Word object, we check for underscores as we describe above.

```
InputFile f = new InputFile("conv1.txt");
        String s = f.readLine();
        while (s != null) {
            parseTokens(s);
            s = f.readLine();
        }
        f.close();
//----------------------------------------
private void parseTokens(String s) {
        StringTokenizer tok = new StringTokenizer(s);
        while(tok.hasMoreTokens ()) {
            Word w = new Word(tok.nextToken ());
            med.add (w);
        }
        med.add(new Word("\n"));
    }
```

Loading the Words into the StyledDocument

Once we have read in the list of words and identified which are to be highlighted, we load them into the LimitedStyleDocument in a simple loop:

```
for (int i = 0; i < med.size(); i ++) {
                lsDoc.insertString(med.get(i));
  }
```

The LimitedStyleDocument just calls the *insertString* method. If the word is to be highlighted it adds it to the highlight list.

```
public class LimitedStyledDocument extends DefaultStyledDocument {
    int maxCharacters;
    int offset;
    HilitePainter hlp;
    JTextPane textPane;

    public LimitedStyledDocument(int maxChars, Mediator md) {
        maxCharacters = maxChars;
```

```
        offset = 0;
        hlp = new HilitePainter(md);
    }
    //save text pane object
    public void setTextPane(JTextPane txp) {
        textPane = txp;
    }
    //insert string into StyledDocument
    public void insertString(Word word) {
        if ((getLength() + word.length()) <= maxCharacters) {
            try {
                super.insertString(offset, word.getWord(), word.getAttributes());
                //if highlighted, add to highlight list
                if (word.isHighlighted ()) {
                    Highlighter hlt = textPane.getHighlighter ();
                    hlt.addHighlight (offset, word.length() + offset, hlp);
                }
                offset += word.length();
            } catch (BadLocationException e) {
            }
        }
    }
}
```

Painting the Highlights

The HighlightPainter class paints the background of any word that is to be highlighted. As we noted above, the Shape and second offset parameters are not correctly set to be useful, so we use a reference to the Mediator class to fetch the word at that offset, and find its length. We also use the *modelToView* method to find the Rectangle defined by that word. In this implementation, we paint a background highlighted rectangle in yellow.

```
public class HilitePainter implements Highlighter.HighlightPainter {
    private Mediator med;

    public HilitePainter(Mediator md) {
        med = md;
    }
    public void paint (Graphics g, int p0, int p1, Shape bounds, JTextComponent c) {
        int offset = 0;

        g.setColor(Color.yellow);  //paint bbackround in yellow
        Rectangle r = null;
        Rectangle r1 = null;
        //set each rectangle
        try {
            r= c.modelToView (p0);
            Word w = med.findWord(p0); //find word begining there
            //find offset of next word
            offset = w.getOffset() + w.length ();
            r1 = c.modelToView (offset);
        } catch (BadLocationException ex) {
            System.out.println("Bad offset");
            r = bounds.getBounds ();
        }
        //draw rectangle for background
        g.fillRect(r.x, r.y, (int)Math.abs(r1.x - r.x), r.height);
    }
}
```

We show the highlighted text in Figure 1.

Figure 1. The highlighted text.

The text displayed in this window is excerpted from “The Ultimate Publishable Computer Science Paper for the 90s,” by my colleagues Richard Lam and Sherman Alpert, in the January 1997 issue of the *Communications of the ACM*.

Intercepting Mouse Clicks

Once you have the highlighted display, you probably want to click on it to select a word and operate on it. You can use a MouseListener method in a MouseAdapter inner class to display the text of the selected word in the text field below the window, as shown in Figure 2.

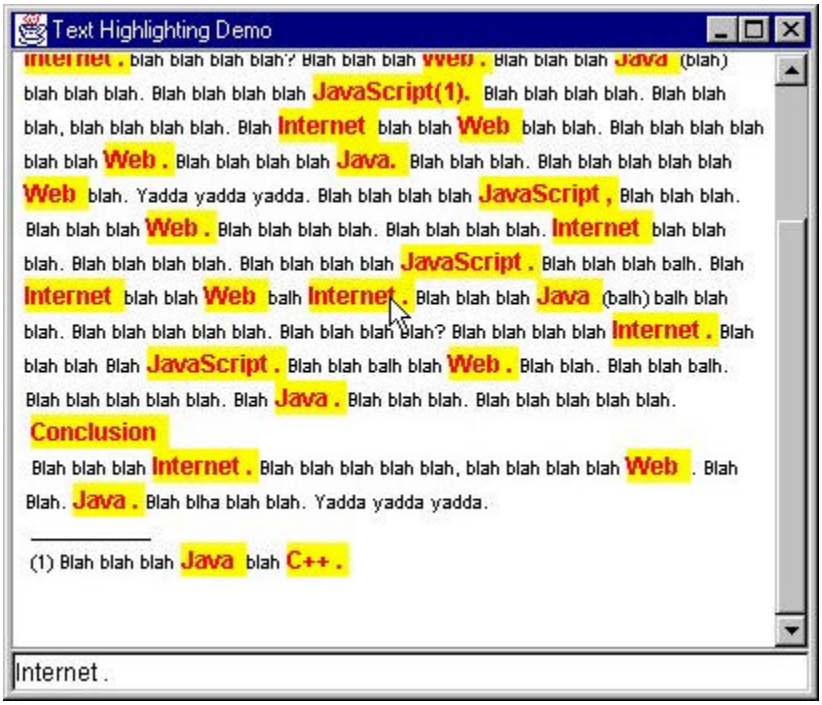

Figure 2 – The highlighted display showing the selected term.

The inner class is shown below:

```
class MListen extends MouseAdapter {
        public void mouseClicked(MouseEvent e) {
            Point pt = new Point(e.getX(), e.getY());
            int offset = textPane.viewToModel (pt);
            Word w = med.findLastWord (offset);
            txt.setText (w.getWord ());
        }
    }
```

An Exercise for the Reader

You may have noticed that multiword terms to be highlighted may wrap, depending on the window size. In this case, the highlighting runs off to the right instead of wrapping with it. How could you modify the program to prevent this from happening? I've thoiught of two solutions. Can you?

Patterns we Used

You can see that we are probably using a Mediator class in this program. In addition, we could consider the Mediator plus the Word class as a Façade pattern, where two class encapsulate several more complex lower level classes. Can you find any others?