I'VE GOT A LITTLE LIST

James W. Cooper

As someday it may happen that a sorted list must be found, Java's got the list. I was thinking about the problem of sorted terms in a list box when I was writing a Visual Basic program the other day. VB has a very handy feature where you can define a list box as sorted and not worry about keeping the data sorted when you add it to the list: it is sorted automatically. Well, I thought, this is something we certainly ought to be able to do in Java as well, and probably do it a lot better, since Java is an OO language and since the visual controls are so flexible.

The first thing we should recognize is that this is a problem for the JFC or Swing controls, not for the AWT List object. The AWT List is very simple and very limited: you can add and remove lines in the list box and select one or more of them. But sorting is not one of its features.

Fortunately the JList control in the Swing classes is more than up to this challenge and quite a few related challenges as we'll see below. However, in order to write programs using the Swing controls, we'll have to get up to speed on the Swing environment. Briefly, these components are written almost entirely in Java and do not have the "peer" native code components that the AWT components do. They also have a pluggable look and feel, so you can make your programs look like Windows, Motif, or like Java. While these classes are nominally the Java Foundation Classes (or JFC), the library they are contained in is called the "Swing" library for historical reasons, so Java people say that it is spelled "JFC" but is pronounced "Swing."

Starting up a Swing program means setting the look-and-feel, activating the window close box and setting up the layout of the main window. All Swing applications start with the JFrame component, so I have extended JFrame to a simple JxFrame class which I use as my base class, and do all these things under the covers. The entire JxFrame class is shown in Figure 1:

```
public class JxFrame extends JFrame {
    public JxFrame(String title) {
        super(title);
        setCloseClick();
        setLF();
    }
    private void setCloseClick() {
        //create window listener to respond to window close click
        addWindowListener(new WindowAdapter() {
          public void windowClosing(WindowEvent e) {
               System.exit(0);
                              }
        });
    }
    //--------------------------------------------
    private void setLF() {
        // Force SwingApp to come up in the System L&F
        String laf = UIManager.getSystemLookAndFeelClassName();
        try {
            UIManager.setLookAndFeel(laf);
        } catch (UnsupportedLookAndFeelException exc) {
            System.err.println("Unsupported L&F: " + laf);
        } catch (Exception exc) {
            System.err.println("Error loading " + laf + ": " + exc);
        }
    }
}
```

Figure 1: The JxFrame class, which we use as the base class in all our Swing programs.

When you start a Swing program, you add your visual objects to the Content Pane rather than to the JFrame itself. This means that every Swing program starts with something like

```
JPanel jp = new JPanel();
getContentPane().add(jp);
```

where you then treat the JPanel as the container for all your objects.

The JList Class

You may recall that the JList itself is sort of tricky to use and people frequently write Adapter classes to make it seem just as simple as the AWT List class. I even wrote a column on this last year. However it is this flexibility that is the answer to our sorted list problem and a couple of more sophisticated related problems. The JList class works as a display system for data that you tell it about. Rather than adding or removing data from the list itself, you modify the data object and have it tell the JList object that it has changed. This sort of separation of the data from the display is typical of the Observer pattern, where the data tells the Observer that some values have changed.

We also note that the JList does not support scrolling directly. Instead you have to insert the JList inside the viewport of a JScrollPane as we see below.

There are three possible data representation that the JList object can display, a Vector, an array and any object that implements the ListModel interface. In the first two cases, a simple ListModel is created under the covers and connected to the JList. In the final, and most general case, you write some of the code yourself.

So, you ask, can we use just this little bit of information to create a sorted list? Yes, you can, as long as the sorting is quite simple. You could just create an array, sort it and connect it to a JList. We do that in the simple example below:

```
/** Simple sorted array used as data for JList*/
public class ShowList extends JxFrame {
    String names[];

    public ShowList() {
        super("List of names");
        JPanel jp = new JPanel();
        getContentPane().add(jp);
        jp.setLayout(new BorderLayout());

        //create the array
        names = new String[5];
        int i = 0;
        names[i++] = ("Dave");
        names[i++] = ("Charlie");
        names[i++] = ("Adam");
        names[i++] = ("Edward");
        names[i++] = ("Barry");

        //sort the array
        Arrays.sort (names);

        //create the list and add it
```

```
        JList nList = new JList(names);
        JScrollPane sc = new JScrollPane();
        sc.getViewport().add (nList);
        jp.add ("Center", sc);

        setSize(new Dimension(150, 150));
        setVisible(true);
    }
```

As you see, we just create a 5-element array and add 5 out of order names to it. Then we use the static *sort* method of the Arrays class to sort the array and add the sorted array to the JList. This results in the simple display shown in Figure 2.

Figure 2- A simple JList loaded from a sorted array.

Of course this is really much too simple for actual use. If we added more elements to the array they wouldn't be sorted, and if we had to call the sort each time ourselves, we've made the program a bit too complicated for such a simple task.

A JList with a ListModel

The next simplest kind of JList utilizes the DefaultListModel to contain the data. This class implements the same methods as the Vector class, and notifies the JList whenever the data changes. So a complete program for a non-sorted list display can be as

```
/** Creates a JList based on an unsorted DefaultListModel*/
public class ShowList extends JxFrame implements ActionListener {
    String names[]= {"Dave", "Charlie", "Adam", "Edward", "Barry"};
    JButton Next;
    DefaultListModel ldata;
    int index;

    public ShowList() {
        super("List of names");
        JPanel jp = new JPanel();
        getContentPane().add(jp);
        jp.setLayout(new BorderLayout());

        //create the ListModel
        ldata = new DefaultListModel();
        //Create the list
        JList nList = new JList(ldata);
        //add it to the scroll pane
        JScrollPane sc = new JScrollPane();
        sc.getViewport().setView(nList);
        jp.add ("Center", sc);
        JButton Next = new JButton("Next");

        //add an element when button clicked
```

```
        JPanel bot = new JPanel();
        jp.add("South", bot);
        bot.add(Next);
        Next.addActionListener (this);

        setSize(new Dimension(150, 150));
        setVisible(true);
        index = 0;

    }
    public void actionPerformed(ActionEvent evt) {
        if(index < names.length)
            ldata.addElement (names[index++]);

    }
```

In this program we add a "Next" button along the bottom which adds a new name each time it is clicked. The data are not sorted here, but it is pretty obvious that if we just subclass the DefaultListModel, we can have our sorted list and have the elements always sorted, even after new names are added.

So, if we create a class based on DefaultListModel which extends the addElement method and re-sorts the data each time, we'll have our sorted list:

```
/** This simple list model re-sorts the data every time*/
public class SortedModel extends DefaultListModel {
  private String[] dataList;

  public void addElement(Object obj) {
    //add to internal vector
    super.addElement(obj);
    //copy into array
    dataList = new String[size()];
    for(int i=0; i< size(); i++) {
        dataList[i] = (String)elementAt(i);
    }
    //sort the data and copy it back
    Arrays.sort (dataList); //sort data
    clear();    //clear out vector

    //reload sorted data
    for(int i  =0; i < dataList.length; i++)
         super.addElement(dataList[i]);

    //tell JList to repaint
    fireContentsChanged(this, 0, size());
  }
}
```

We see this list in Figure 3. The names are added one at a time in non-alphabetic order each time you click on the Next button, but sorted before being displayed.

Figure 3: Sorted data using SortedListModel

Sorting More Complicated Objects

Now, suppose that we want to display both first and last names, and want to sort by the last names. In order to do that we have to create an object which holds first and last names, but which can be sorted by last name. And how do we do this sort? Well, we could do it by brute force, but in Java any class which implements the Comparable interface can be sorted by the Arrays.sort method. And the Comparable interface is just one method:

```
public int compareTo(Object obj)
```

where the class returns a negative value, zero or a positive value depending on whether the existing object is less than, equal to or greater than the argument object. Thus, we can create a Person class with this interface just as simply as

```
public class Person implements Comparable {

  private String frname, lname;

  public Person(String name) {
   //split name apart
    int i = name.indexOf (" ");
    frname = name.substring (0, i).trim();
    lname = name.substring (i).trim();
  }
  public int compareTo(Object obj) {
      Person to = (Person)obj;
      return lname.compareTo (to.getLname ());
  }
  public String getLname() {
      return lname;
  }
  public String getFrname() {
      return frname;
  }
  public String getName() {
      return getFrname()+" "+getLname();
  }
}
```

Note that the **compareTo** method simply invokes the compareTo method of the last name String objects.

The other change we have to make is that our data model has to return both names, so we extend the getElementAt method:

```
/** Data model which uses and sorts Person objects*/
public class SortedModel extends DefaultListModel {
  private Person[] dataList;

  public void addElement(Object obj) {
    Person per = new Person((String) obj);
    super.addElement(per);
    dataList = new Person[size()];

   //copy the Persons into an array
    for(int i=0; i< size(); i++) {
        dataList[i] = (Person)elementAt(i);
    }

   //sort them
    Arrays.sort (dataList);

   //and put them back
    clear();
    for(int i  =0; i < dataList.length; i++)
         super.addElement(dataList[i]);
    fireContentsChanged(this, 0, size());
  }
  public Object getElementAt(int index) {
      //returns both names as a string
      Person p = dataList[index];
      return p.getName();
  }
  public Object get(int index) {
      return getElementAt(index);
  }
}
```

You see the resulting sorted names below:

Figure 4 – Sorted list using Comparable interface to sort on last names.

Getting Database Keys

Now one disadvantage of a sorted list is that clicking on the n*th* element does not correspond to selecting the n*th* element, since the sorted elements can be in an order that is different from the order you added them to the list. This, if we want to get a database key corresponding to a particular list element (here a person) in order to displayed detailed information about that person, you have to keep the database key inside the person object. This is analogous to but

considerably more flexible than the Visual Basic approach where you can keep only one key value for each list element. Here you could keep several items in the person object if that is desirable. In the figure below, we double click on one person's name and pop up a window containing his phone number.

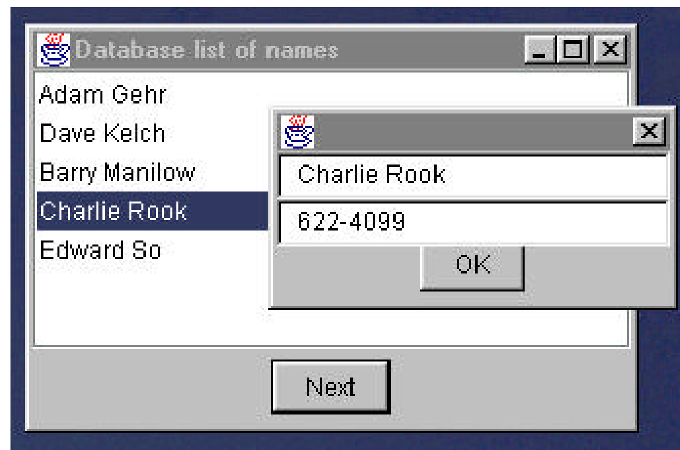

Figure 5 – A pop up list showing details appears when you double click on a name.

In order to pop up a window when you double click on a JList, you must add a mouse listener to the JList object. In our code, we created a class called mouseListener which carries out the listening and produces the popup. First we add the mouseListener by

```
nList.addMouseListener(new mouseListener(nList, ldata, db, this));
```

where **db** represents out database and **ldata** the list data model. The complete mouseListener class is shown below:

```
public class mouseListener extends MouseAdapter {
    private JList nList;
    private DataBase db;
    SortedModel lData;
    JFrame jxf;

    public mouseListener(JList list, SortedModel ldata,
                         DataBase dbase, JFrame jf) {
        nList = list;
        db = dbase;
        jxf = jf;
        lData = ldata;
    }
    //
    public void mouseClicked(MouseEvent e) {
        if (e.getClickCount () == 2) {
        //mouse double clicked-
        //get database key for this person
            int index = nList.locationToIndex (e.getPoint());
            int key = lData.getKey (index);
        //display pop up dialog box
            Details details = new Details(jxf,
                  db.getName (key), db.getPhone (key));
            details.setVisible(true);
        }
    }
}
```

Note that since the JList control has no specific methods for detecting a mouse double click, we check the mouseClicked event method and see if the click count is 2. If it is, we query the database for the name and phone number of the person with that key. In the example code accompanying this article, we simulate the database with a simple text file so that the code example does not become unwieldy.

Pictures in our List Boxes

The JList is flexible in yet another way. You can write your own cell rendering code to display anything you want in a line of a list box. So you can include images, graphs or dancing babies, if you want. All you have to do is create a cell rendering class that implements the ListCellRenderer interface. This interface has but one method called **getListCellRendererComponent** and is quite simple to write. By simply extending the JLabel class, which itself allows for images as well as text, we can display names and images alongside each name with very little effort. We assume that each Person object now contains the image to display:

```
public class cellRenderer extends JLabel implements ListCellRenderer {

    public Component getListCellRendererComponent(JList list,
            Object value, int index, boolean isSelected,
            boolean hasFocus) {
        Person p = (Person)value; //get the person
        setText(p.getName ());   //display name
        setIcon(p.getIcon ());   //and image
        if(isSelected)
             setBackground(Color.lightGray );
        else
            setBackground(Color.white );
        return this;
    }
}
```

Of course we also connect this cell renderer class to the JList with this simple method call:

```
nList.setCellRenderer (new cellRenderer());
```

The resulting display is shown in Figure 6.

Figure 6—A sorted list with pictures added using a custom cell renderer.

Listing to Bow

In summary, it takes only a little effort to realize the power inherent in the JList component. You can use it to make sorted lists, or even sorted picture lists. In addition, you can sort on more complex properties by providing classes that implement the Comparable interface.