
THE JLIST CLASS

The JList class is a more powerful replacement for the simple List class that is provided with the AWT. A JList can be instantiated using a Vector or array to represent its contents. The JList does not itself support scrolling and thus must be added to a JScrollPane to allow scrolling to take place.

In the simplest program you can write using a JList, you

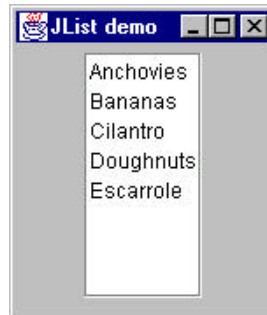
1. add a JScrollPane to the Frame, and then
2. create a Vector of data
3. create a JList using that Vector
4. add the JList to the JScrollPane's viewport

This is shown below

```
JPanel jp = new JPanel();           //panel in Frame
getContentPane().add(jp);

//create scroll pane
JScrollPane sp = new JScrollPane();
jp.add(sp);                          //add to layout
Vector dlist = new Vector();         //create vector
dlist.addElement("Anchovies");      //and add data
dlist.addElement("Bananas");
dlist.addElement("Cilantro");
dlist.addElement("Doughnuts");
dlist.addElement("Escarrole");
JList list= new JList(dlist); //create list with data
sp.getViewport().add(list); //add list to scrollpane
```

This produces the display shown below:



You could just as easily use an array instead of a Vector and have the same result.

List Selections and Events

You can set the JList to allow users to select a single line, multiple contiguous lines or separated multiple lines with the *setSelectionMode* method, where the arguments can be

```
SINGLE_SELECTION
SINGLE_INTERVALSELECTION
MULTIPLE_INTERVAL_SELECTION
```

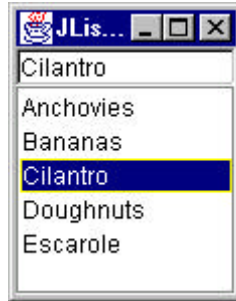
You can then receive these events by using the *addListSelectionListener* method. Your *ListSelectionListener* must implement the interface

```
public void valueChanged(ListSelectionEvent e)
```

In our *JListLDemo.java* example we display the selected list item in a text field:

```
public void valueChanged(ListSelectionEvent e) {
    text.setText((String)list.getSelectedValue());
}
```

This is shown below:



Changing a List Display Dynamically

If you want a list to change dynamically during your program, the problem is somewhat more involved because the `JList` displays the data it is initially loaded with and does not update the display unless you tell it to. One simple way to accomplish this is to use the `setListData` method of `JList` to keep passing it a new version of the updated `Vector` after you change it. In the `JListADemo.java` program, we add the contents of the top text field to the list each time the Add button is clicked. All of this takes place in the action routine for that button:

```
public void actionPerformed(ActionEvent e)    {
    dlist.addElement(text.getText()); //add text from field
    list.setListData(dlist);           //send new Vector to list
    list.repaint();                   //and tell it to redraw
}
```

One drawback to this simple solution is that you are passing the entire `Vector` to the list each time and it must update its entire contents each time rather than only the portion that has changed. This brings us to the underlying `ListModel` that contains the data the `JList` displays.

When you create a `JList` using an array or `Vector`, the `JList` automatically creates a simple `ListModel` object which contains that data. The `ListModel` objects are an extension of the `AbstractListModel` class. This model has the following simple methods:

```
void fireContentsChanged(Object source, int index0, int index1)
void fireIntervalAdded(Object source, int index0, int index1)
void fireIntervalRemoved(Object source, int index0, int index1)
```

and you need only implement those *fire* methods your program will be using. For example, in this case we really only need to implement the *fireIntervalAdded* method.

Our ListModel is an object that contains the data (in a Vector or other suitable structure) and notifies the JList whenever it changes. Here, the list model is just the following:

```
class JListData extends AbstractListModel {
    private Vector dlist; //the food name list

    public JListData() {
        dlist = new Vector();
        makeData(); //create the food names
    }
    public int getSize() {
        return dlist.size();
    }
    private Vector makeData()
    {
        //add food names as before
        return dlist;
    }
    public Object getElementAt(int index) {
        return dlist.elementAt(index);
    }
    //add string to list and tell the list about it
    public void addElement(String s) {
        dlist.addElement(s);
        fireIntervalAdded(this, dlist.size()-1, dlist.size());
    }
}
```

This ListModel approach is really an implementation of the Observer design pattern we discuss in Chapter 4. The data are in one class and the rendering or display methods in another class, and the communication between them triggers new display activity.

Lists displayed by JList are not limited to text-only displays. The ListModel data can be a Vector or array of any kind of Objects. If you use the default methods, then only the String representation of those objects will be displayed. However, you can define your own display routines using the *setCellRenderer* method, and have it display icons or other colored text or graphics as well.