

# 10

*List  
boxes*

---

## **In this chapter**

- Creating and using list boxes
- Rendering custom list formats
- Creating multiline list items

A list box is simply a user interface control containing a collection of similar items from which the user can make one or more selections. This is the premise behind the AWT List control, which presents a Java component equivalent to list box objects found in all other graphics interfaces. The List class provides the basic functionality to add and remove items, to display them (including the ability to scroll through long lists), and to allow the user to make selections. Though List is adequate for many applications, it falls short when you need to perform special operations, such as rendering lists of graphics or other peculiar data.

Swing provides a completely different approach to list management by supplying an entirely new list box component in a class called JList. In addition to the capabilities of the AWT List class, JList also has the ability to display graphics, with or without associated text, and also provides additional event handling.

---

*FYI* Though many of the methods in the JList API are similar to those of AWT's List class, the two classes are generally incompatible. If you are planning to port an existing AWT application that uses List, you will need to do some additional work to correctly implement JList replacements.

---

Unlike AWT List, JList is a lightweight component, so it implements its display and data handling capabilities in separate models. Throughout our discussion of Swing's list box class, we will examine some techniques to create and manage special models, and we will generate some advanced examples to demonstrate key features. First, we will start by looking at a simple example of an application using a JList component.

## 10.1 A simple JList example

---

Listing 10.1 shows the source code needed to implement a simple list box example using the Swing API. This example generates a static list from which the user can make a selection. Currently, this code provides no mechanism to detect the selections made, nor does it include the capability to scroll the list, if required. We will add these features as we progress through this chapter.

```
// Imports
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class TestFrame
    extends JFrame
{
    // Instance attributes used in this example
    private JPanel topPanel;
    private JList listBox;

    // Constructor of main frame
    public TestFrame()
    {
        // Set the frame characteristics
        setTitle( "Simple ListBox Application" );
        setSize( 300, 100 );

        // Create a panel to hold all other components
        topPanel = new JPanel();
        topPanel.setLayout( new BorderLayout() );
        getContentPane().add( topPanel );

        // Create some items to add to the list
        String listData[] =
        {
            "Item 1",
            "Item 2",
            "Item 3",
            "Item 4"
        };

        // Create a new list box control
        listBox = new JList( listData );
        topPanel.add( listBox, BorderLayout.CENTER );

        // Anonymous inner class to terminate program.
        this.addWindowListener( new WindowAdapter()
        {
            public void windowClosing( WindowEvent e )
            {
                System.exit( 0 );
            }
        } ); // End addWindowListener
    }

    // Main entry point for this example
    public static void main( String args[] )
```

**Listing 10.1 Simple JList application**

---

```

    {
        // Create an instance of the test application
        TestFrame mainFrame = new TestFrame();
        mainFrame.setVisible( true );
    }
}

```

---

**Listing 10.1** Simple JList application (continued)

*FYI* The JList class has no provision to handle scrolling. If your list contains more information than can be shown in the space provided, you need to add the list to a JScrollPane instance. This is identical to the technique used for JTextArea and for any other component capable of displaying more information than you have space for.

---

If you are familiar with the AWT List class, you will see that there isn't an add method in the JList class. JList does not store the data shown in the list; instead, it relies on a data model to manage that task. Listing 10.1 included an array of strings (the data model, for this sample) and associated that data with the user interface when the JList object was constructed.

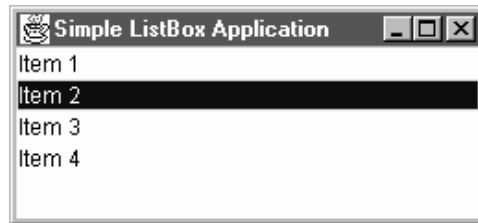
As you will see later in this chapter, the use of a separate data model opens up many new opportunities for list boxes within your application. With AWT, if you want to display dynamic data from a data array in a list, you are required to copy the data from the array into the List instance using the add() method. With the JList class in Swing, all you need to do is identify the data array as the JList instance's data model, which reduces the amount of code you need to write and maintain.

Figure 10.1 shows the output produced by the simple example presented in listing 10.1. Notice that you can use the mouse to click on any of the items to highlight them. This example is quite simple, but, as you will soon see, the JList component can be much more complex.

## 10.2 A more advanced JList example

---

The previous example demonstrated the most basic list box example which included only static data and supported no selection management. However, there is much more to the JList class. Let's create a more advanced list box application to demonstrate some of the functionality provided by this useful Swing component.



**Figure 10.1**  
JList application output

In this sample, we want to show a list box that will be dynamically updated by the user. Additionally, we must provide the capability to add new text to the list, so we will create an Add button and a text field into which the user can type. Also, since the user may want to delete items, we will create a Remove button and, to it, attach code to delete list items. Finally, we will intercept list selections and transfer the selected item back into the text field.



**Design a list that doesn't require horizontal scrolling.** As noted previously, users can get a little annoyed with horizontal scrolling and might ignore horizontally occluded data. Use concise terms for list items.

Listing 10.2 presents the source code for this more complex list box example. A string vector has replaced the static string array (of listing 10.1) so that the code can dynamically insert and remove strings. Additionally, since the list may grow beyond the bounds of the application frame, the list instance has been added to a scrolling pane, and it will automatically attach scroll bars when required.

```
// Imports
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

class TestFrame
    extends JFrame
    implements ActionListener,
        ListSelectionListener
{
```

**Listing 10.2** Advanced JList application

```
// Instance attributes used in this example
private JPanel      topPanel;
private JList       listBox;
private Vector      listData;
private JButton     addButton;
private JButton     removeButton;
private JTextField  dataField;
private JScrollPane scrollPane;

// Constructor of main frame

public TestFrame()
{
    // Set the frame characteristics
    setTitle( "Advanced List Box Application" );
    setSize( 300, 100 );

    // Create a panel to hold all other components
    topPanel = new JPanel();
    topPanel.setLayout( new BorderLayout() );
    getContentPane().add( topPanel );

    // Create the data model for this example
    listData = new Vector();

    // Create a new list box control
    listBox = new JList( listData );
    listBox.addListSelectionListener( this );

    // Add the list box to a scrolling pane
    scrollPane = new JScrollPane();
    scrollPane.getViewport().add( listBox );
    topPanel.add( scrollPane, BorderLayout.CENTER );

    CreateDataEntryPanel();

    // Anonymous inner class to terminate program.
    this.addWindowListener( new WindowAdapter()
    {
        public void windowClosing( WindowEvent e )
        {
            System.exit( 0 );
        }
    } );// End addWindowListener
}

public void CreateDataEntryPanel()
```

**Listing 10.2** Advanced JList application (continued)

```
{
    // Create a panel to hold all other components
    JPanel dataPanel = new JPanel();
    dataPanel.setLayout( new BorderLayout() );
    topPanel.add( dataPanel, BorderLayout.SOUTH );

    // Create some function buttons
    addButton = new JButton( "Add" );
    dataPanel.add( addButton, BorderLayout.WEST );
    addButton.addActionListener( this );

    removeButton = new JButton( "Delete" );
    dataPanel.add( removeButton, BorderLayout.EAST );
    removeButton.addActionListener( this );

    // Create a text field for data entry and display
    dataField = new JTextField();
    dataPanel.add( dataField, BorderLayout.CENTER );
}

// Handler for list selection changes
public void valueChanged( ListSelectionEvent event )
{
    // See if this is a list box selection and the
    // event stream has settled
    if( event.getSource() == listBox
        && !event.getValueIsAdjusting() )
    {
        // Get the current selection and place it in the
        // edit field
        String stringValue = (String)listbox.getSelectedValue();
        if( stringValue != null )
            dataField.setText( stringValue );
    }
}

// Handler for button presses
public void actionPerformed((ActionEvent event) )
{
    if( event.getSource() == addButton )
    {
        // Get the text field value
        String stringValue = dataField.getText();
        dataField.setText( "" );

        // Remove this item from the list and refresh
        if( stringValue != null )
```

**Listing 10.2** Advanced *JList* application (continued)

```
        {
            listData.addElement( stringValue );
            listBox.setListData( listData );
            scrollPane.revalidate();
            scrollPane.repaint();
        }
    }

    if( event.getSource() == removeButton )
    {
        // Get the current selection
        int selection = listBox.getSelectedIndex();
        if( selection >= 0 )
        {
            // Add this item to the list and refresh
            listData.removeElementAt( selection );
            listBox.setListData( listData );
            scrollPane.revalidate();
            scrollPane.repaint();

            // As a nice touch, select the next item
            if( selection >= listData.size() )
                selection = listData.size() - 1;
            listBox.setSelectedIndex( selection );
        }
    }
}

// Main entry point for this example
public static void main( String args[] )
{
    // Create an instance of the test application
    TestFrame mainFrame = new TestFrame();
    mainFrame.setVisible( true );
}
}
```

**Listing 10.2** Advanced JList application (continued)

The output of listing 10.2 is shown in figure 10.2. Initially, the list in the center of the frame is empty; however, by typing text and pressing the Add button, items can be added to the list. The vertical scroll bar (and horizontal scroll bar, if necessary) appears whenever the number of items in the list exceeds the frame size.



**Figure 10.2**  
Advanced JList application output

### 10.3 Listening for list activity

The code in listing 10.2 implements a `ListSelectionListener` that is responsible for detecting user list selections. The list box attaches the listener code using the following line:

```
listbox.addListSelectionListener( this );
```

The `valueChanged()` method is called any time the user selects an item from the list. When this occurs, the code determines the current list selection and copies its text to the text field. The `valueChanged()` method, like other action handlers shown previously, uses the `event.getSource()` call to determine the component generating the event.

```
if( event.getSource() == listbox  
    && !event.getValueIsAdjusting() )
```

But, this line of code does something else, too. List selection events are generated in clusters, one for each item that is being deselected, one for each selected item, and one final event that indicates that the selection values are no longer changing. This final event is the one the program needs to detect, while all others can be disregarded. To accomplish this, the code uses the event's `getValueIsAdjusting()` method.

### 10.4 Custom data model

The examples shown in listings 10.1 and 10.2 implement the default data model built into Swing. This suggests that even though you didn't specifically make a copy

of the data while elements were being added to the list, a copy operation was still occurring. For large arrays of list data, this can negatively effect performance.

Being a lightweight component, JList easily accepts the replacement of its data model with a custom version implemented by a programmer. In listing 10.3A, we revisit the simple list example; however, you might notice that the example does not include a string array or a vector holding any list strings. Instead, the code creates an instance of the `CustomListModel` class (see listing 10.3B). Except for the addition of a scrolling pane, this example is otherwise unchanged from the first simple list example shown in listing 10.1.

---

```
// Imports
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

class TestFrame
    extends JFrame

{
    // Instance attributes used in this example
    private JPanel      topPanel;
    private JList       listBox;
    private CustomListModel listData;
    private JScrollPane scrollPane;

    // Constructor of main frame
    public TestFrame()
    {
        // Set the frame characteristics
        setTitle( "Custom Data Model List Application" );
        setSize( 300, 100 );

        // Create a panel to hold all other components
        topPanel = new JPanel();
        topPanel.setLayout( new BorderLayout() );
        getContentPane().add( topPanel );

        // Create the data model for this example
        listData = new CustomListModel();

        // Create a new list box control
        listBox = new JList( listData );
```

---

**Listing 10.3A** Custom data model application

```
// Add the list box to a scrolling pane
scrollPane = new JScrollPane();
scrollPane.getViewport().add( listBox );
topPanel.add( scrollPane, BorderLayout.CENTER );

// Anonymous inner class to terminate program.

this.addWindowListener( new WindowAdapter()
{
    public void windowClosing( WindowEvent e )
    {
        System.exit( 0 );
    }
} );// End addWindowListener
}

// Main entry point for this example
public static void main( String args[] )
{
    // Create an instance of the test application
    TestFrame mainFrame = new TestFrame();
    mainFrame.setVisible( true );
}
}
```

**Listing 10.3A Custom data model application (continued)**

Listing 10.3B shows the CustomListModel class, which is responsible for answering requests for list element data. Notice that the code does not store list elements in an array or vector—instead, data are manufactured on the fly. A custom data model for a list extends the AbstractListModel class, and, as such, it must implement two methods.

The first of these, `getSize()`, returns the number of items in the model. In our case, the method returns a constant value of 300. The second method, `getElementAt()`, returns the actual list item at the specified item offset (starting at zero). In our example, we manufacture the value string using the format `Item n`, where `n` is the index number.

```
// Imports
import com.sun.java.swing.*;

class CustomListModel
```

**Listing 10.3B Custom data model class**

---

```

    extends    AbstractListModel
{
    // Return the size of the list
    public int getSize()
    {
        return 300;
    }

    // Return an element from the list
    public Object getElementAt( int index )
    {
        return "Item " + index;;
    }
}

```

---

**Listing 10.3B** Custom data model class (continued)

The output of listing 10.3A is a 300-element list, shown in figure 10.3. Note that since there is no actual data array in the example, everything you see when you run the code is artificial. For large, static lists, this is an excellent technique to reduce the memory requirements of your application and improve its overall performance.

## 10.5 Basic custom list rendering

---

But wait, there's more! So far, all of the JList examples shown in this chapter included only textual data. This is adequate in most situations, but we can improve the appearance of our Java applications by adding graphics to the list and changing the font and color used to display list elements. Much like the custom data model shown previously, the viewer can also be replaced simply by adding a custom renderer to the list.



**Simulate UI components using graphics in lists.** Small graphics (such as 16 by 16 pixels) can add a lot of visual appeal to a list. Consider drawing UI components, such as check boxes or radio buttons, as graphics to enable custom selection behaviors in list controls.

---

Listings 10.4A and 10.4B make up an application that implements a simple list like the example shown in listing 10.1; however, the example also provides a custom

cell renderer, which has the responsibility of drawing each element in the list. Items can be drawn differently depending on whether or not they are selected, including the foreground and background colors and the font. The example shown here displays the selected item in a 24-point font and changes the background color of the selection to red. Notice, also, that this example includes images for each item to help the user better identify the style associated with each selection, greatly improving the user's experience with this application.

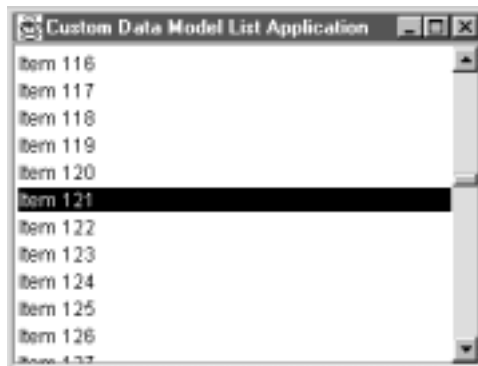
```
// Imports
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class TestFrame
    extends JFrame
{
    // Instance attributes used in this example
    private JPanel topPanel;
    private JList listBox;

    // Constructor of main frame
    public TestFrame()
    {
        // Set the frame characteristics
        setTitle( "Rendered ListBox Application" );
        setSize( 300, 160 );

        // Create a panel to hold all other components
        topPanel = new JPanel();
    }
}
```

**Listing 10.4A Custom rendering list example**



**Figure 10.3**  
Custom data model application output

```
topPanel.setLayout( new BorderLayout() );
getContentPane().add( topPanel );

// Create some items to add to the list
String listData[] =
{
    "Circles",
    "Bubbles",
    "Thatch",
    "Pinstripes"
};

// Create a new list box control
listbox = new JList( listData );
listbox.setCellRenderer( new CustomCellRenderer() );
topPanel.add( listbox, BorderLayout.CENTER );

// Anonymous inner class to terminate program.
this.addWindowListener( new WindowAdapter()
{
    public void windowClosing( WindowEvent e )
    {
        System.exit( 0 );
    }
} ); // End addWindowListener
}

// Main entry point for this example
public static void main( String args[] )
{
    // Create an instance of the test application
    TestFrame mainFrame = new TestFrame();
    mainFrame.setVisible( true );
}
}
```

**Listing 10.4A Custom rendering list example (continued)**

The `CustomCellRenderer` class first loads the required images in its constructor, then waits for the list box to request a rendering operation. The `getListCellRendererComponent()` method performs the actual rendering, determining the colors and font required to display the item. Additionally, this method assigns the correct image to the item being drawn.

```
// Imports
import java.awt.*;
import javax.swing.*;

class CustomCellRenderer
    extends JLabel
    implements ListCellRenderer
{
    private ImageIcon image[];

    public CustomCellRenderer()
    {
        setOpaque(true);

        // Pre-load the graphics images to save time
        image = new ImageIcon[4];
        image[0] = new ImageIcon( "circles.gif" );
        image[1] = new ImageIcon( "bubbles.gif" );
        image[2] = new ImageIcon( "thatch.gif" );
        image[3] = new ImageIcon( "pinstripe.gif" );
    }

    public Component getListCellRendererComponent(
        JList list, Object value, int index,
        boolean isSelected, boolean cellHasFocus )
    {
        // Display the text for this item
        setText(value.toString());

        // Set the correct image
        setIcon( image[index] );

        // Draw the correct colors and font
        if( isSelected )
        {
            // Set the color and font for a selected item
            setBackground( Color.red );
            setForeground( Color.white );
            setFont( new Font( "Roman", Font.BOLD, 24 ) );
        }
        else
        {
            // Set the color and font for an unselected item
            setBackground( Color.white );
            setForeground( Color.black );
            setFont( new Font( "Roman", Font.PLAIN, 12 ) );
        }
    }
}
```

**Listing 10.4B Custom cell renderer class**

---

```

        return this;
    }
}

```

---

**Listing 10.4B Custom cell renderer class (continued)**

Figure 10.4 shows the output produced by listing 10.4B. Notice that the selected item, Thatch, is presented in a much larger font than the other items, and each item includes an associated image indicating graphically what the text is describing.



**Figure 10.4**  
Custom rendering list application output

## 10.6 Advanced custom list rendering

---

But, that's still not all! The custom renderer in listing 10.4B extended the JLabel control and used it to draw what is essentially a label in the space provided for each item in the list. Though the addition of a graphic provided a unique interface, it still exhibits some limitations for some applications.

Well, the good news is that JLabel is only one of several classes that can be displayed in a list box. You can also display instances of JButton, JTextField, JCheckBox, and so on—even JTextArea (a multiline component nested into a list box item). Here's the code:

---

```

// Imports
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class TestFrame
    extends JFrame

```

---

**Listing 10.5A Advanced custom rendering list example**

```
{
    // Instance attributes used in this example
    private JPanel    topPanel;
    private JList    listBox;

    // Constructor of main frame
    public TestFrame()
    {
        // Set the frame characteristics
        setTitle( "Advanced Rendered ListBox Application" );
        setSize( 300, 160 );

        // Create a panel to hold all other components
        topPanel = new JPanel();
        topPanel.setLayout( new BorderLayout() );
        getContentPane().add( topPanel );

        // Create some items to add to the list
        String listData[] =
        {
            "Chapter 1\nIntroduction",
            "Chapter 2\nA Java Refresher",
            "Chapter 3\nSwing Basics",
            "Chapter 4\nPanels, Panes and More Layout Managers"
        };

        // Create a new list box control
        listBox = new JList( listData );
        listBox.setCellRenderer( new CustomCellRenderer() );
        topPanel.add( listBox, BorderLayout.CENTER );

        // Anonymous inner class to terminate program.
        this.addWindowListener( new WindowAdapter()
        {
            public void windowClosing( WindowEvent e )
            {
                System.exit( 0 );
            }
        } );// End addWindowListener
    }
    // Main entry point for this example
    public static void main( String args[] )
    {
        // Create an instance of the test application
        TestFrame mainFrame = new TestFrame();
        mainFrame.setVisible( true );
    }
}
```

**Listing 10.5A** Advanced custom rendering list example (continued)

Listing 10.5A is still based on the first example shown in the chapter, and includes almost no new functionality when compared to listing 10.1. Notice, in listing 10.5A, the values loaded into the list:

```
"Chapter 1\nIntroduction",  
"Chapter 2\nA Java Refresher",  
"Chapter 3\nSwing Basics",  
"Chapter 4\nPanels, Panes and More Layout Managers"
```

Each value includes a `\n` new line character to force a line break in the middle of the text. Listing 10.5B implements a custom cell renderer, but, this time, it extends `JTextArea` instead of `JLabel`. Since `JTextArea` does not support graphics, we have abandoned the icons from the previous example.

---

```
// Imports  
import java.awt.*;  
import javax.swing.*;  
import javax.swing.border.*;  
  
class CustomCellRenderer  
    extends      JTextArea  
    implements   ListCellRenderer  
{  
    public Component getListCellRendererComponent(  
        JList list, Object value, int index,  
        boolean isSelected, boolean cellHasFocus )  
    {  
        setBorder( new BevelBorder( BevelBorder.RAISED ) );  
  
        // Display the text for this item  
        setText(value.toString());  
  
        // Draw the correct colors and font  
        if( isSelected )  
        {  
            // Set the color and font for a selected item  
            setBackground( Color.red );  
            setForeground( Color.white );  
        }  
        else  
        {  
            // Set the color and font for an unselected item  
            setBackground( Color.lightGray );  
            setForeground( Color.black );  
        }  
    }  
}
```

---

**Listing 10.5B** Advanced custom cell renderer class

```

        return this;
    }
}
    
```

**Listing 10.5B** Advanced custom cell renderer class (continued)



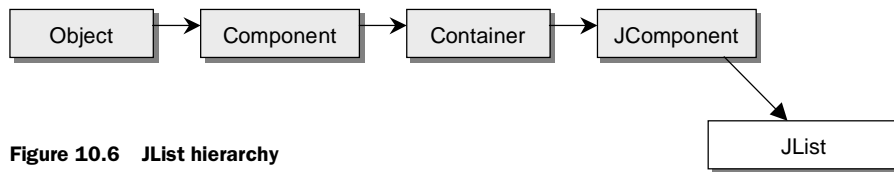
**Figure 10.5** Advanced custom rendering list application output

The result is a nicely formatted list resembling the screen output shown in figure 10.5. Note that, although I elected to eliminate the font changes included with the previous example, the `JTextArea` class does support them. Since the height of each list item is determined when constructed, it is possible to cause display clipping if the font is set to a large value.

## 10.7 JList class information

This section lists the constants, variables, and methods for the `JList` class.

### 10.7.1 JList hierarchy



**Figure 10.6** JList hierarchy

### 10.7.2 JList constructors

```
public JList()
```

This constructor creates an instance of a `JList` object with an empty data model.

```
public JList( ListModel dataModel )
```

This constructor creates an instance of a JList object with the data model specified in the supplied parameter. The data model must not be empty.

```
public JList( Object listData[] )
```

This constructor creates an instance of a JList object with the data specified in the supplied array. This array is inserted into a default list model.

```
public JList(Vector listData)
```

This constructor creates an instance of a JList object with the data specified in the supplied vector. This array is inserted into a default list model.

### 10.7.3 JList significant method groupings

```
public Object getPrototypeCellValue()
public void setPrototypeCellValue(Object prototypeCellValue)
```

These methods manage the cell prototype value. The prototype value is used to calculate the width of each cell in the list. Using a cell prototype is much more efficient than inspecting each item in the list to determine the maximum width.

```
public int getFixedCellWidth()
public void setFixedCellWidth( int width )
public int getFixedCellHeight()
public void setFixedCellHeight( int height )
```

These methods are used to determine and control the width and height of each cell in the list.

```
public ListCellRenderer getCellRenderer()
public void setCellRenderer( ListCellRenderer cellRenderer )
```

These methods manage the delegate class used to paint each cell in the list.

```
public Color getSelectionForeground()
public void setSelectionForeground(Color selectionForeground)
public Color getSelectionBackground()
public void setSelectionBackground(Color selectionBackground)
```

As the names suggest, these methods manage the foreground and background colors used to display the currently selected item. A custom cell renderer can override these values, or that renderer can reference these colors during a painting operation.

```
public int getVisibleRowCount()
public void setVisibleRowCount(int visibleRowCount)
```

These methods detect and control the number of rows visible within a list. The row count (overall height) of a `JList` instance can be managed at run time with these methods.

```
public int getFirstVisibleIndex()  
public int getLastVisibleIndex()  
public void ensureIndexIsVisible( int index )
```

The methods in this group are used to determine the index of the first or last index visible in the list. Additionally, the `ensureIndexIsVisible()` method can force any item into the visible region of the list.

```
public int locationToIndex( Point location )  
public Point indexToLocation( int index )
```

These methods are used to calculate the correlation of a list item, referenced by index, and an actual pixel location within the list. These methods are typically used to determine which item is currently under the mouse pointer.

```
public Rectangle getCellBounds( int index1, int index2 )
```

This method returns a rectangular region bound by two list items specified by index numbers.

```
public ListModel getModel()  
public void setModel( ListModel model )  
public void setListData( Object listData[] )  
public void setListData( Vector listData )
```

These methods are used to manage the data referenced in the list. Data can be stored within a custom data model, a simple array, or a vectored array. Any time data is changed dynamically, one of the set methods shown here must be called to update the list.

```
public void addListSelectionListener( ListSelectionListener listener )  
public void removeListSelectionListener(  
    ListSelectionListener listener)
```

These methods manage the presence of a list selection listener for the current `JList` instance. A list selection listener intercepts events generated by the list object when the user changes the current selection.

```
public int getSelectionMode()  
public void setSelectionMode( int selectionMode )
```

These methods control the selection mode used in the list instance. The mode is limited to:

- `SINGLE_SELECTION` for selecting only one item at a time.
- `SINGLE_INTERVAL_SELECTION` to allow selection of single contiguous blocks of items.
- `MULTIPLE_INTERVAL_SELECTION` to allow multiple item or block selection.

```
public int getAnchorSelectionIndex()
public int getLeadSelectionIndex()
public int getMinSelectionIndex()
public int getMaxSelectionIndex()
```

This group of methods returns list indices based on various restrictions. For example, the first selected item in a range of selections can be determined with the `getAnchorSelectionIndex()` method.

```
public boolean isSelectedIndex(int index)
public boolean isEmptySelection()
public void clearSelection()
public int[] getSelectedIndices()
public void setSelectedIndex(int index)
public void setSelectedIndices(int indices[])
public Object[] getSelectedValues()
public int getSelectedIndex()
public Object getSelectedValue()
public void setSelectedValue( Object anObject, boolean shouldScroll )
public void setSelectionInterval( int anchor, int lead )
public void addSelectionInterval( int anchor, int lead )
public void removeSelectionInterval( int index0, int index1 )
```

This method group manages information about the selections within the list object. Additionally, selections can be added or removed with methods from this group.

```
public void setValueIsAdjusting(boolean b)
public boolean getValueIsAdjusting(boolean b)
```

These two methods manage the state of the `is` value adjusting flag.

## 10.8 Chapter summary

In this chapter, we have examined the Swing `JList` class, which is a superset of the AWT List control. However, unlike `List`, `JList` requires a separate data model to hold the item values it displays. This makes porting AWT List code to `JList` a much more extensive exercise than for most other Swing-based components.



We began this chapter with a simple list example which contained only four static items and offered no scrolling capability. This example showed that creating lists with the `JList` class can be quite simple.

Next, we started to examine custom data models. The second example shown in this chapter used a vectored data array, which permitted us to add or remove list items at run time. This example was followed closely by a third example that used no data array at all. Instead, a completely custom data model was used to manufacture the requested data on the fly.

Finally, we got our first taste of custom display rendering, by replacing the UI delegate with a new one of our own design. We created a simple example that changed the font and color of the list and any selected items, but we also added graphics to perk up the user interface. In the final example, we created an interesting multiline list box, where every list element used two lines instead of the typical one line.

In the next chapter, we will expand on the idea of lists by studying a completely new Swing class which implements a more complex form of list called the hierarchical tree. As you will see, we can employ much of what we already know about lists when we use trees in our applications.



