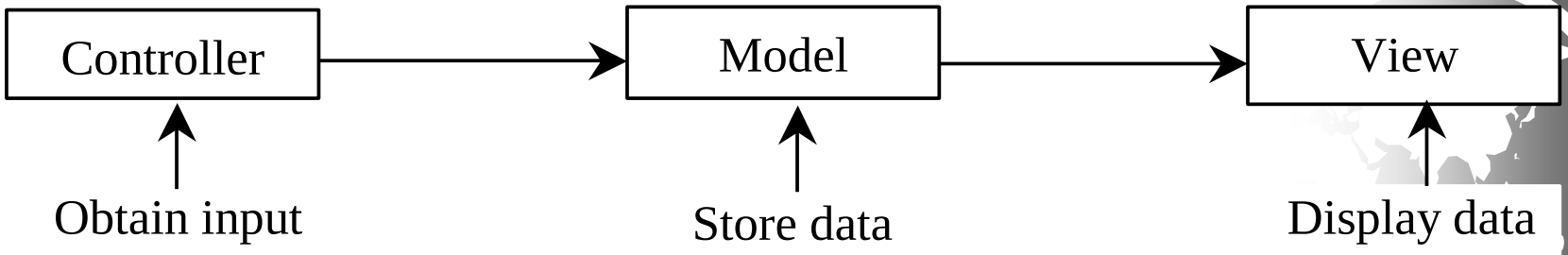
22 MVC and Swing MVC Components

Chapter 35@8e

Model-View-Controller (MVC)

The model-view-controller (MVC) approach is a way of developing components by separating data storage and handling from the visual representation of the data.

The component for storing and handling data, known as a **model**, contains the actual contents of the component. The component for presenting the data, known as a **view**, handles all essential component behaviors. It is the view that comes to mind when you think of the component. It does all the displaying of the components. The **controller** is a component that is usually responsible for obtaining data.



Benefits of MVC

It makes multiple views possible so that data can be shared through the same model. For example, a model storing student names can simultaneously be displayed in a combo box or in a list box

It simplifies the task of writing complex applications and makes the components scalable and easy to maintain. Changes can be made to the view without affecting the model, and vice versa

Synchronization between Model and View

A model contains data, whereas a view makes the data visible.

Once a view is associated with a model, it immediately displays updates to the model. This ensures that all the views of the model display the same data **consistently**.

To achieve consistency and synchronize the model with its dependent views, the model should **notify** the views when there is a change in a property in the model that is used in the view. In response to a change notification, the view is responsible for redisplaying the viewing area affected by the property change.

Synchronization between Model and View

The JDK **event delegation model** provides a superior architecture for supporting MVC component development.

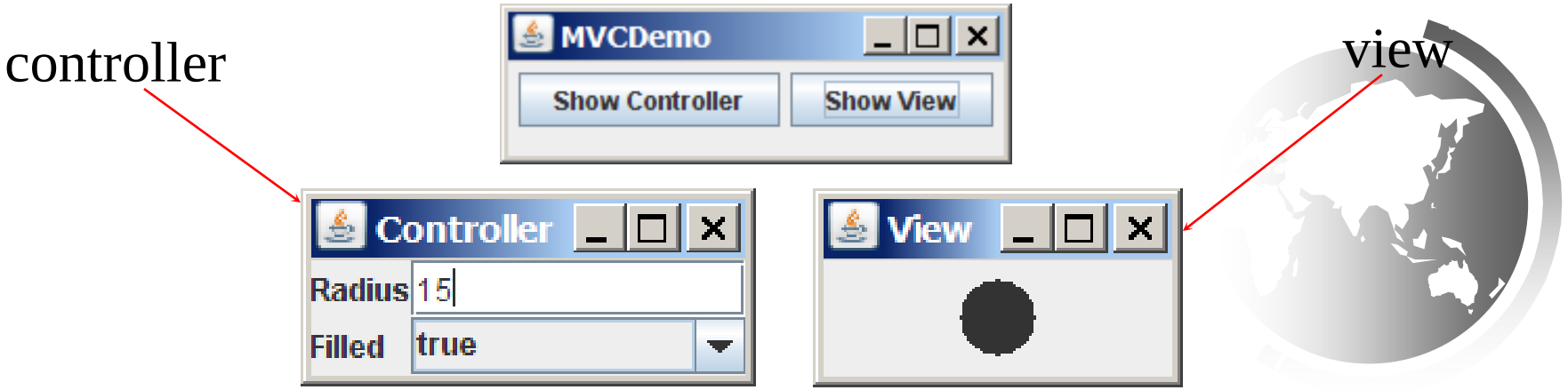
The model can be implemented as a source with appropriate event and event listener registration methods.

The view can be implemented as a listener.

Thus, if data are changed in the model, the view will be notified. To enable the selection of the model from the view, simply add the model as a property in the view with a set method.

Example: Developing Model-View-Controller Components

Problem: The example creates a model named CircleModel, a view named CircleView and a controller named CircleControl. CircleModel stores the properties (radius, filled, and color) that describe a circle. filled is a boolean value that indicates whether a circle is filled. CircleView draws a circle according to the properties of the circle. CircleControl enables the user to enter circle properties from a graphical user interface. Create an applet with two buttons named *Show Controller* and *Show View*. When click the Show Controller button, the controller is displayed in a frame. When click the Show View button, the view is displayed in a separate frame.



CircleModel

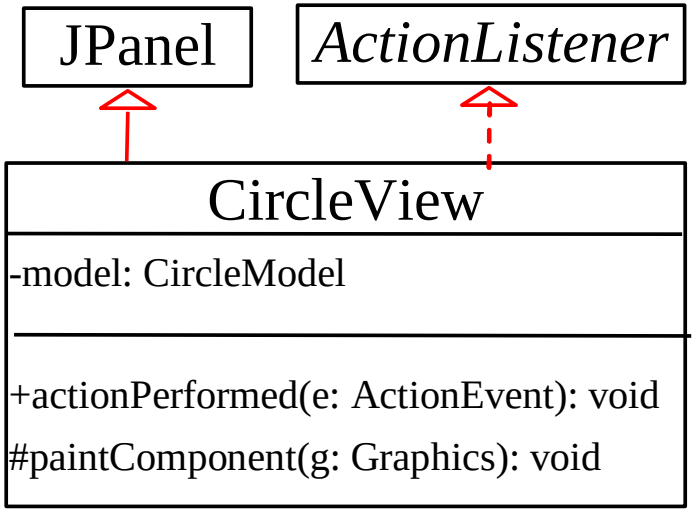
The circle model stores the data and notifies any change of data to the listeners. The circle model contains properties radius, filled, and color, as well as the registration/deregistration methods for action event.

CircleModel	
-radius: double	The radius of this circle.
-filled: boolean	True if the circle is filled.
-color: java.awt.Color	The color of the circle.
+addActionListener(l: ActionListener): void	Adds a new listener to this object.
+removeActionListener(l: ActionListener): void	Removes a listener from this object.
-processEvent(e: ActionEvent): void	Processes the event.

CircleModel

CircleView

The view implements ActionListener to listen for notifications from the model. It contains the model as its property. **When a model is set in the view, the view is registered with the model.** The view extends JPanel to override the paintComponent method to draw the circle according to the properties values specified in the model.



Stores the circle model.

Implements this method to update the view.

Paints the view.

CircleView

CircleController

The controller presents a GUI interface that enables the user to enter circle properties radius, filled, and color.

It contains the model as its property. You can use the setModel method to associate a circle model with the controller. It uses a text field to obtain a new radius and a combo box to obtain a boolean value to specify whether the circle is filled.

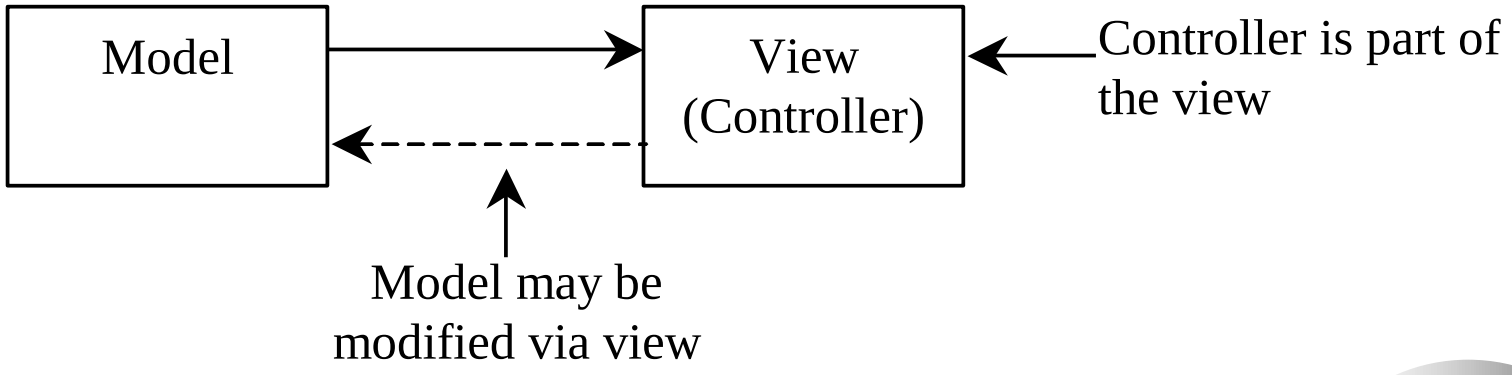
CircleController

Putting Things Together

Finally, let us create an applet named MVCDemo with two buttons Show Controller and Show View. The Show Controller button displays a controller in a frame and the Show View button displays a view in a separate frame.

MVC Variations

One variation of the model-view-controller architecture is to combine the controller with the view. In this case, a view not only presents the data, but is also used as an interface to interact with the user and accept user input.



Another variation of the model-view-controller architecture is to add part of the data from the model to the view so that the frequently used data can be accessed directly from the view.

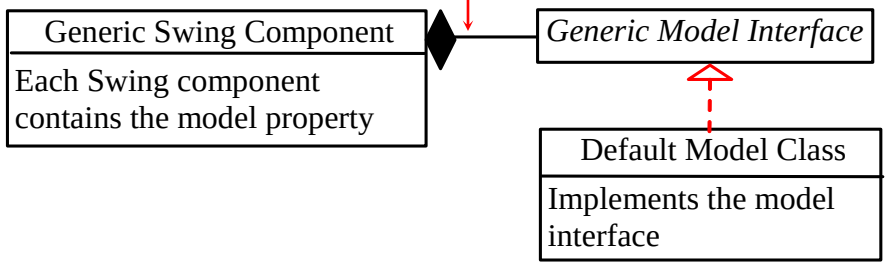
Swing MVC

NOTE:

1. Swing components are designed using the MVC architecture.
2. Each Swing GUI component is a view that uses a model to store data.
3. Many components contain part of the data in the model so that they can be accessed directly from the component.

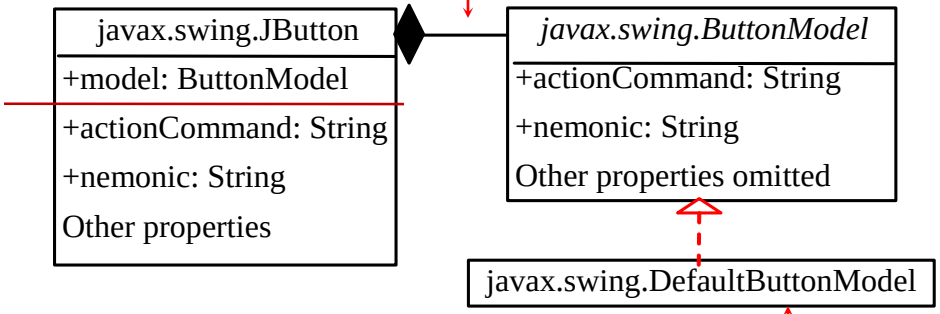
Swing Model-View-Controller Architecture

Each Swing user interface component (except some containers and dialog boxes such as JPanel, JSplitPane, JFileChooser, and JColorChooser) has a property named *model* that refers to its data model.



(A) Generic Swing model-view architecture

The data model is defined in an interface whose name ends with Model. For example, the model for button component is ButtonModel.



(B) JButton model-view implementation

Most model interfaces have a default implementation class that is commonly named Default*X*, where *X* is its model interface name. For example, the default implementation class for ButtonModel is DefaultButtonModel.

Swing Components and Their Models

For convenience, **most Swing components contain some properties of their models**, and these properties can be accessed and modified directly from the component without knowing the existence of the model.

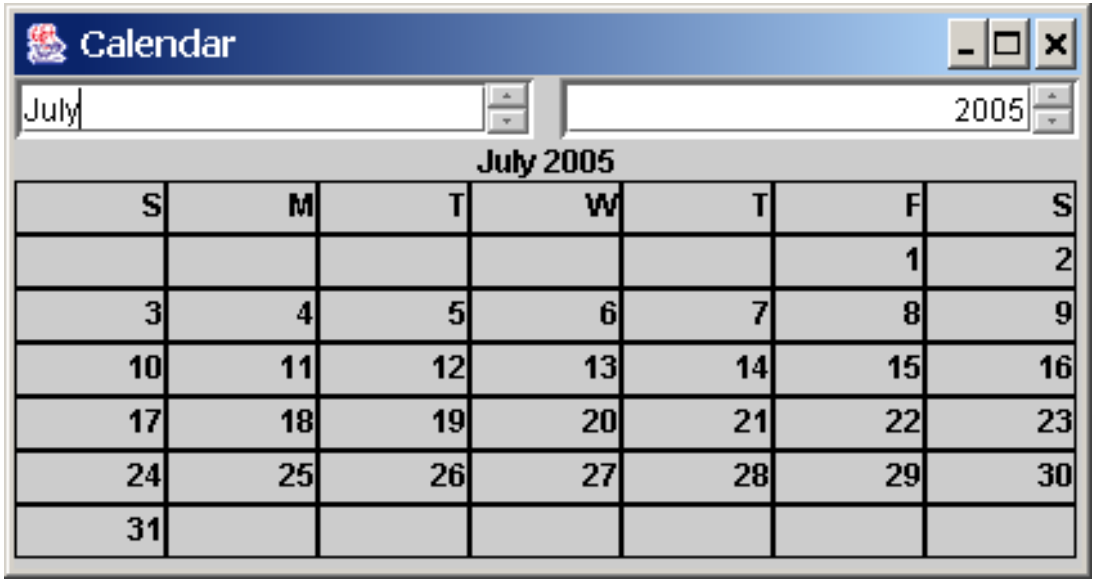
For example, the properties actionCommand and mnemonic are defined in both ButtonModel and JButton. Actually, these properties are in the AbstractButton class. Since JButton is a subclass of AbstractButton, JButton inherits all the properties from AbstractButton.

It is unnecessary to use the models for the simple Swing components such as JButton, JToggleButton, JCheckBox, JRadioButton, JTextField, and JTextArea, **because** the frequently used properties in their models are also in these components. You can access and modify these properties directly through the components.

For advanced components such as JSpinner, JList, JComboBox, JTable, and JTree, you have to work with their models to store, access and modify data.

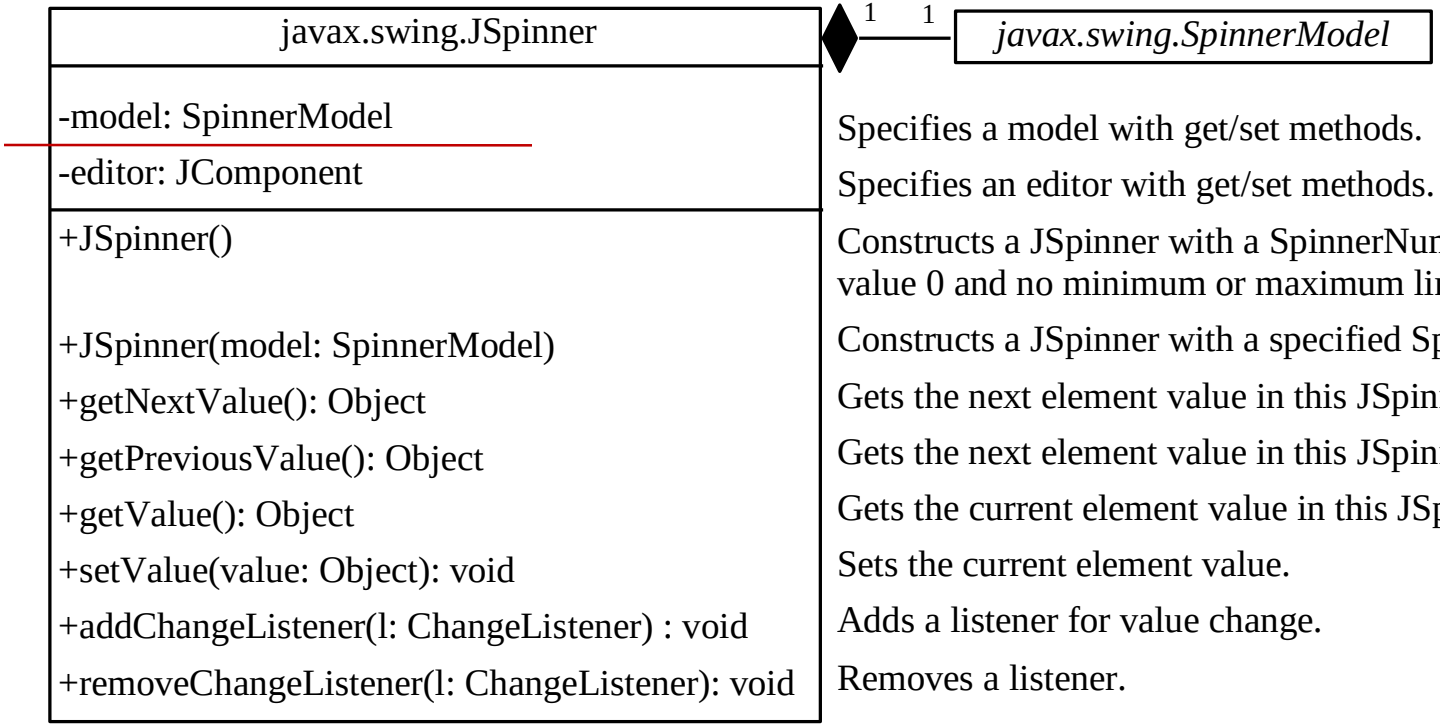
JSpinner

A spinner is a text field with a pair of tiny arrow buttons on its right side that enable the user to select numbers, dates, or values from an ordered sequence. The keyboard up/down arrow keys also cycle through the elements. The user may also be allowed to type a (legal) value directly into the spinner. A spinner is similar to a combo box, but a spinner is sometimes preferred because it doesn't require a drop down list that can obscure important data.



The JSpinner Class

A JSpinner's sequence value is defined by the SpinnerModel interface, which manages a potentially unbounded sequence of elements. The model doesn't support indexed random access to sequence elements. Only three sequence elements are accessible at a time: current, next and previous using the methods getValue(), getNextValue(), and getPreviousValue(), respectively.



Example: A Simple JSpinner Demo

Problem: This example creates a JSpinner object for a sequence of numbers and displays the previous, current, and next number from the spinner on a label.

NOTE: If you create a JSpinner object without specifying a model, the spinner displays a sequence of integers.

SimpleSpinner

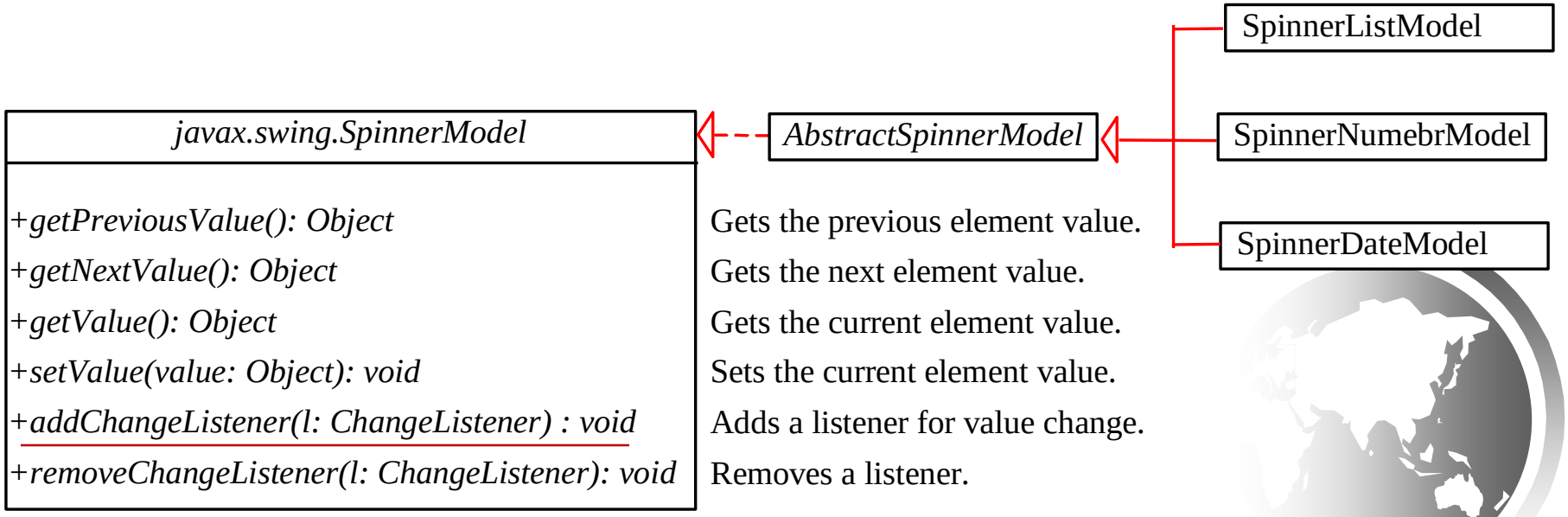
Run

Constructs a JSpinner with a SpinnerNumberModel with initial value 0 and no minimum or maximum limits.

```
// Create a JSpinner
private JSpinner spinner = new JSpinner();

// Create a JLabel
private JLabel label = new JLabel("", JLabel.CENTER);

public SimpleSpinner() {
  // Add spinner and label to the UI
  add(spinner, BorderLayout.NORTH);
  add(label, BorderLayout.CENTER);

  // Register and create a listener
  spinner.addChangeListener(new ChangeListener() {
    @Override
    public void stateChanged(javax.swing.event.ChangeEvent e) {
      label.setText("Previous value: " + spinner.getPreviousValue()
        + " Current value: " + spinner.getValue()
        + " Next value: " + spinner.getNextValue());
    }
  });
}
```

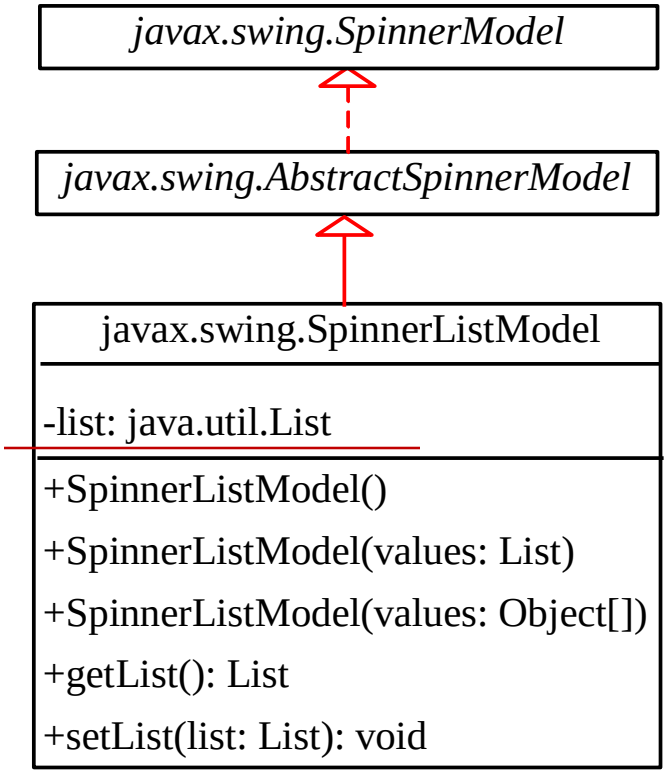
Spinner Models

SpinnerModel is an interface for all spinner models. AbstractSpinnerModel is a convenience abstract class that implements SpinnerModel and provides the implementation for the registration/deregistration methods. SpinnerListModel, SpinnerNumberModel, and SpinnerDateModel are concrete implementations of SpinnerModel.



SpinnerListModel

SpinnerListModel is a simple implementation of SpinnerModel whose values are stored in a java.util.List.



Stores data in a list.

Constructs a SpinnerListModel that contains “empty” string element.

Constructs a SpinnerListModel with the specified list.

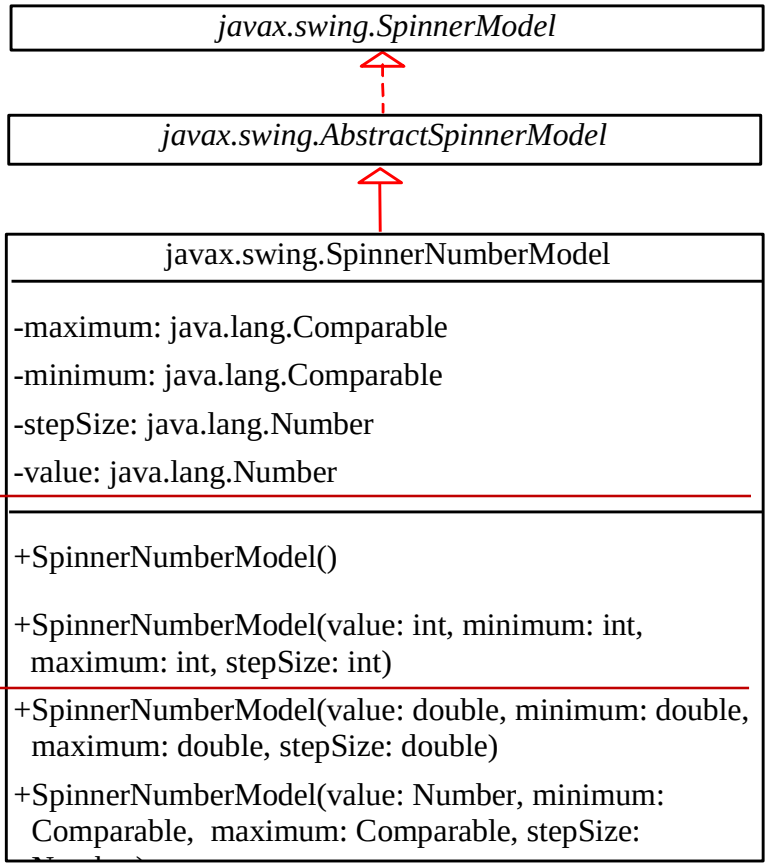
Constructs a SpinnerListModel with the specified array.

Gets the list where data is stored.

Sets a new list for the model .

SpinnerNumberModel

SpinnerNumberModel is a concrete implementation of SpinnerModel that represents a sequence of numbers. It contains the properties maximum, minimum, and stepSize.



Specifies the upper bound of the sequence with get/set methods.

Specifies the lower bound of the sequence with get/set methods.

Specifies the interval in the sequence with get/set methods.

Holds the current selected value with get/set methods.

Constructs an unbounded SpinnerNumberModel with an initial value of zero and stepSize equal to one.

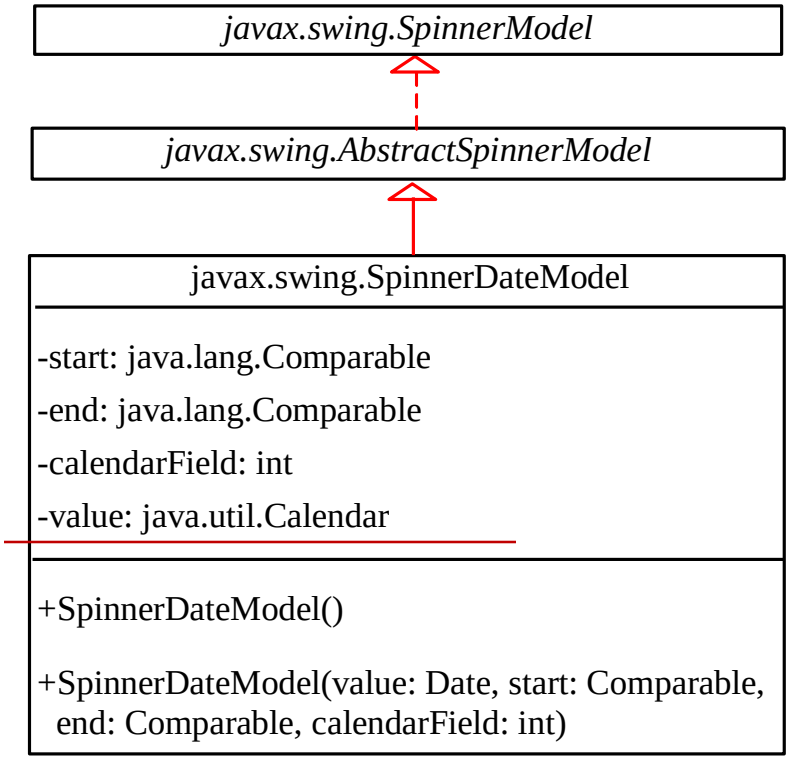
Constructs a SpinnerNumberModel with the specified initial value, minimum/maximum bounds, and stepSize in int.

Constructs a SpinnerNumberModel with the specified initial value, minimum/maximum bounds, and stepSize in double.

Constructs a SpinnerNumberModel that represents a closed sequence of numbers from minimum to maximum.

SpinnerDateModel

SpinnerDateModel is a concrete implementation of SpinnerModel that represents a sequence of dates. The upper and lower bounds of the sequence are defined by properties called start and end and the size of the increase or decrease computed by the nextValue and previousValue methods is defined by a property called calendarField.



Specifies the start date (upper bound) in the model with get/set methods.

Specifies the end date (lower bound) in the model with get/set methods.

Specifies the calendar field (interval) in the sequence with get/set methods.

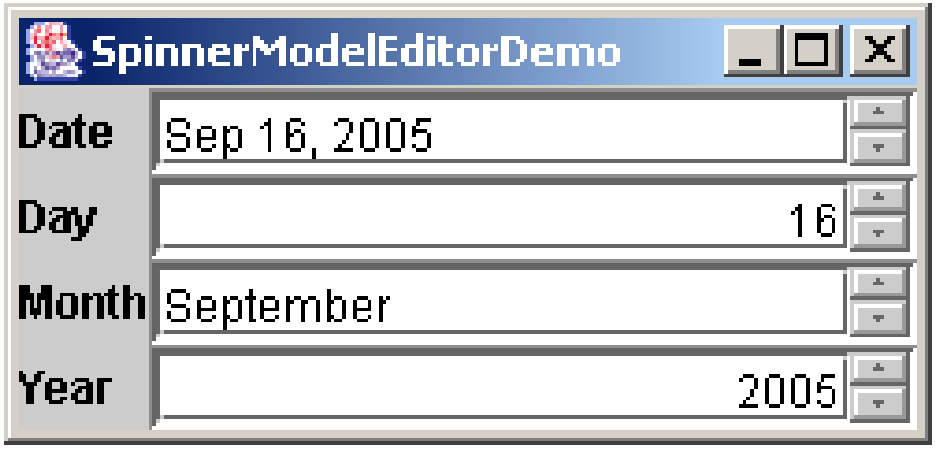
Holds the current selected date with get/set methods.

Constructs an unbounded SpinnerDateModel whose initial value is the current date, calendarField is equal to Calendar.DAY_OF_MONTH.

Constructs a SpinnerNumberModel with the specified initial date, start/end bounds, and calendarField.

Example: Using Spinner Models and Editors

Problem: This example uses a JSpinner component to display date and three separate JSpinner components to display day in a sequence of numbers, month in a sequence of strings, and year in a sequence of numbers. All these four components are synchronized. For example, if you change year in the spinner for year, the date value in the date spinner is updated accordingly.



SpinnerModelEditorDemo

Run

```
private JSpinner jspDate =
  new JSpinner(new SpinnerDateModel());
private JSpinner jspDay =
  new JSpinner(new SpinnerNumberModel(1, 1, 31, 1));
private String[] monthNames = new DateFormatSymbols().getMonths();
private JSpinner jspMonth = new JSpinner
  (new SpinnerListModel(Arrays.asList(monthNames).subList(0, 12)));
private JSpinner spinnerYear =
  new JSpinner(new SpinnerNumberModel(2004, 1, 3000, 1));

// Register and create a listener for jspDay
jspDay.addChangeListener(new ChangeListener() {
  @Override
  public void stateChanged(javax.swing.event.ChangeEvent e) {
    updateDate();
  }
});

// Register and create a listener for jspMonth
jspMonth.addChangeListener(new ChangeListener() {
  @Override
  public void stateChanged(javax.swing.event.ChangeEvent e) {
    updateDate();
  }
});

// Register and create a listener for spinnerYear
spinnerYear.addChangeListener(new ChangeListener() {
  @Override
  public void stateChanged(javax.swing.event.ChangeEvent e) {
    updateDate();
  }
});
// Set editor for date
JSpinner.DateEditor dateEditor =
  new JSpinner.DateEditor(jspDate, "MMM dd, yyyy");
jspDate.setEditor(dateEditor);

// Set editor for year
JSpinner.NumberEditor yearEditor =
  new JSpinner.NumberEditor(spinnerYear, "####");
spinnerYear.setEditor(yearEditor);
```

```
/** Update date spinner to synchronize with the other spinners */
private void updateDate() {
  // Get current month and year in int
  int month = ((SpinnerListModel)jspMonth.getModel()).
      getList().indexOf(jspMonth.getValue());
  int year = ((Integer)spinnerYear.getValue()).intValue();

  // Set a new maximum number of days for the new month and year
  SpinnerNumberModel numberModel =
    (SpinnerNumberModel)jspDay.getModel();
  numberModel.setMaximum(new Integer(maxDaysInMonth(year, month)));

  // Set a new current day if it exceeds the maximum
  if (((Integer)(numberModel.getValue())).intValue() >
      maxDaysInMonth(year, month))
    numberModel.setValue(new Integer(maxDaysInMonth(year, month)));

  // Get the current day
  int day = ((Integer)jspDay.getValue()).intValue();

  // Set a new date in the date spinner
  jspDate.setValue(
    new GregorianCalendar(year, month, day).getTime());
}
```

```
 * @since 1.4
 */
public abstract class AbstractSpinnerModel implements SpinnerModel, Serializable
{

    /**
     * Only one ChangeEvent is needed per model instance since the
     * event's only (read-only) state is the source property.  The source
     * of events generated here is always "this".
     */
    private transient ChangeEvent changeEvent = null;


    /**
     * The list of ChangeListeners for this model.  Subclasses may
     * store their own listeners here.
     */
    protected EventListenerList listenerList = new EventListenerList();


    /**
     * Adds a ChangeListener to the model's listener list.  The
     * ChangeListeners must be notified when the models value changes.
     *
     * @param l the ChangeListener to add
     * @see #removeChangeListener
```

```
    public void addChangeListener(ChangeListener l) {
        listenerList.add(ChangeListener.class, l);
    }


    /**
     * Removes a ChangeListener from the model's listener list.
     *
     * @param l the ChangeListener to remove
     * @see #addChangeListener
     * @see SpinnerModel#removeChangeListener
     */
    public void removeChangeListener(ChangeListener l) {
        listenerList.remove(ChangeListener.class, l);
    }
```

```
protected void fireStateChanged()
{
    Object[] listeners = listenerList.getListenerList();
    for (int i = listeners.length - 2; i >= 0; i -=2 ) {
        if (listeners[i] == ChangeListener.class) {
            if (changeEvent == null) {
                changeEvent = new ChangeEvent(this);
            }
            ((ChangeListener)listeners[i+1]).stateChanged(changeEvent);
        }
    }
}
```

SpinnerListModel

用户实现

```
public void setValue(Object elt) {
    int index = list.indexOf(elt);
    if (index == -1) {
        throw new IllegalArgumentException("invalid sequence element");
    }
    else if (index != this.index) {
        this.index = index;
        fireStateChanged();
    }
}
```

JSpinner

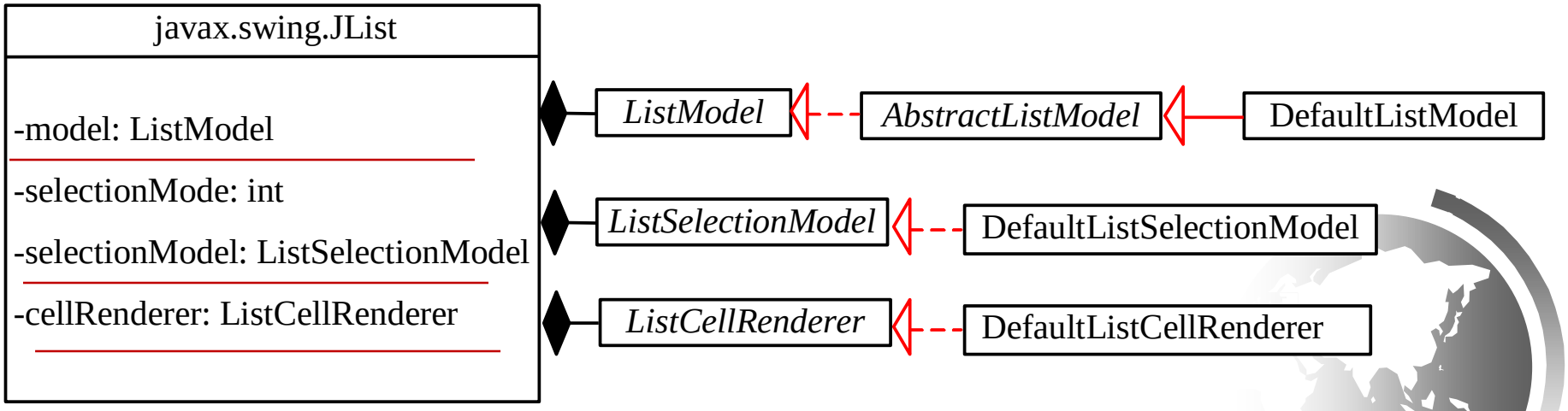
```
     */
    public void setValue(Object value) {
        getModel().setValue(value);
    }
```

JList

JList has two supporting models: a list model and a list-selection model. The *list model* is for storing and processing data.

The *list-selection model* is for selecting items.

By default, items are rendered as strings or icons. You can also create a custom renderer implementing the ListCellRenderer interface.

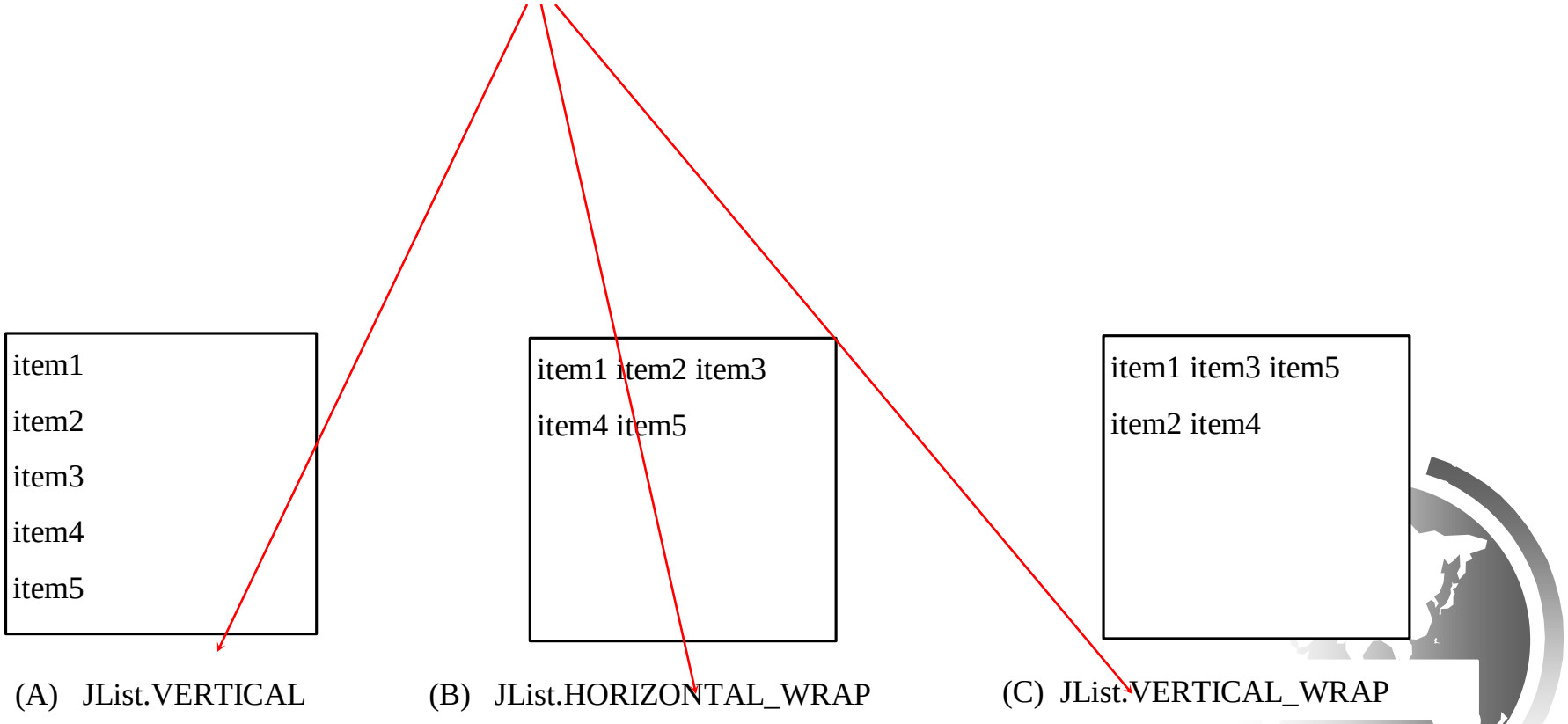


The JList Class

-cellRenderer: ListCellRenderer	The object that renders the list items.
-fixedCellHeight: int	The fixed cell height value in pixels.
-fixedCellWidth: int	The fixed cell width value.
-layoutOrientation: int	Defines the way list cells are laid out.
-model: ListModel	Specifies the list model for this list.
-selectedIndex: int	The index of the first selected item in this list.
-selectedIndices: int[]	An array of all of the selected indices in increasing order.
-selectedValue: Object	The first selected value.
-selectedValues: Object[]	An array of the values for the selected values in increasing index order.
-selectedBackground: int	The background color of the selected items.
-selectedForeground: int	The foreground color of the selected items.
-selectionMode: int	Specifies whether single- or multiple-interval selections are allowed.
-selectionModel: ListSelectionModel	Specifies a selection model.
-visibleRowCount: int	The preferred number of rows to display without using a scroll bar (default: 8).
+JList()	Constructs a default JList.
+JList(dataModel: ListModel)	Constructs a JList with the specified model.
+JList(listData: Object[])	Constructs a JList with the data specified in the array.
+JList(listData: Vector)	Constructs a JList with the data specified in the vector.
+setListData(listData: Object[]): void	Sets an array of objects as data for the list.
+setListData(listData: Vector): void	Sets a vector of objects as data for the list.

The **layoutOrientation** property

```
JList jlst = new JList();

jlst.setLayoutOrientation(int);
```



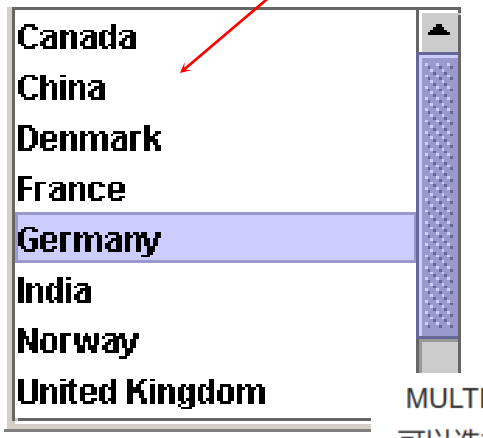
The **selectionMode** property

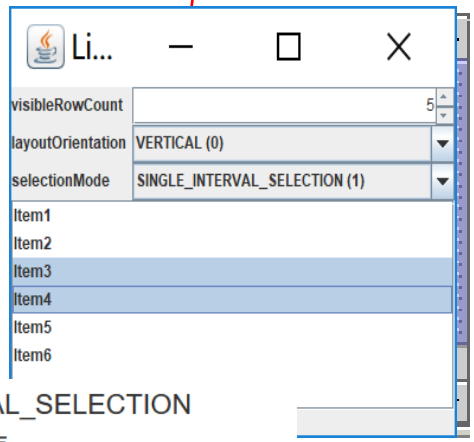
```
JList jlst = new JList();
jlst.setSelectionMode(int);
```

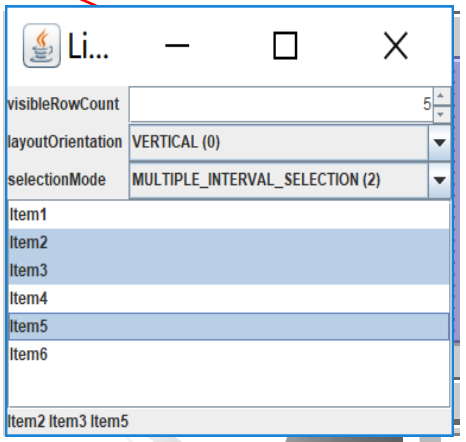
JList.SINGLE_SELECTION

JList.SINGLE_INTERVAL_SELECTION

JList.MULTIPLE_INTERVAL_SELECTION







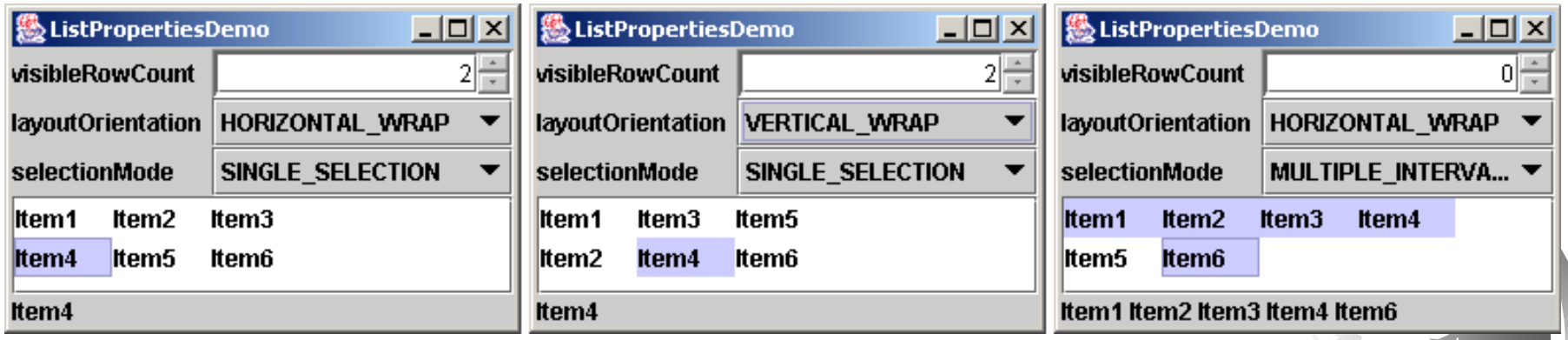
MULTIPLE_INTERVAL_SELECTION
可以选择不相邻的几项

SINGLE_INTERVAL_SELECTION
只能选择连续的几项

SINGLE_SELECTION
一次只能选择一项

Example: List Properties Demo

Problem: This example creates a list of a fixed number of items displayed as strings. The example enables you to dynamically set visibleRowCount from a spinner, layoutOrientation from a combo box, and selectionMode from a combo box. When you select one or more items, their values are displayed in a status label below the list.



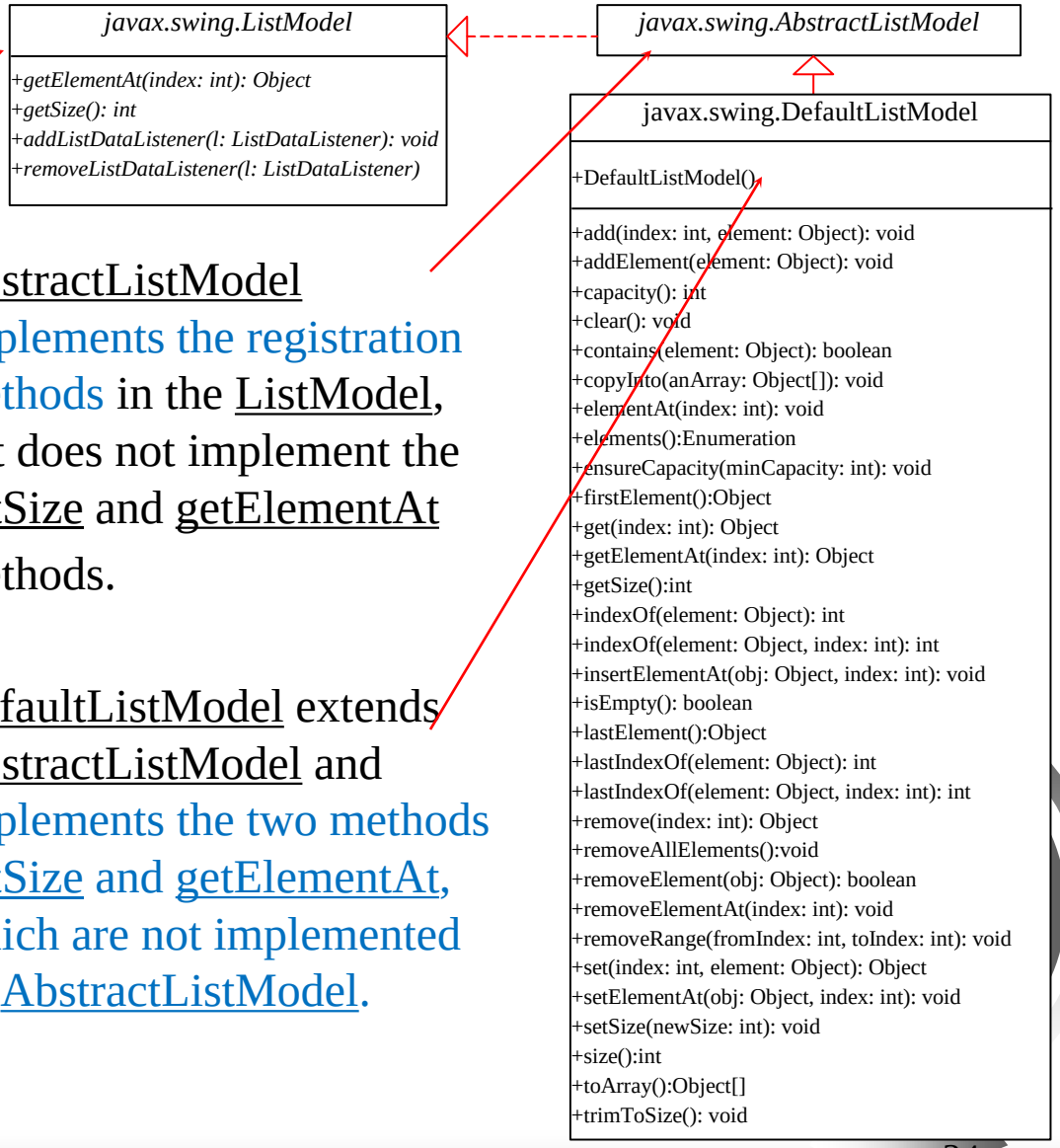
ListPropertiesDemo

Run

```
// Set initial property values
jlst.setFixedCellWidth(50);
jlst.setFixedCellHeight(20);
jlst.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

// Register listeners
jspVisibleRowCount.addChangeListener(new ChangeListener() {
  @Override
  public void stateChanged(ChangeEvent e) {
    jlst.setVisibleRowCount(
      ((Integer)jspVisibleRowCount.getValue()).intValue());
  }
});

jcboLayoutOrientation.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
    jlst.setLayoutOrientation(
      jcboLayoutOrientation.getSelectedIndex());
  }
});

jcboSelectionMode.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
    jlst.setSelectionMode(
      jcboSelectionMode.getSelectedIndex());
  }
});

jlst.addListSelectionListener(new ListSelectionListener() {
  @Override
  public void valueChanged(ListSelectionEvent e) {
    Object[] values = jlst.getSelectedValues();
    String display = "";

    for (int i = 0; i < values.length; i++) {
      display += (String)values[i] + " ";
    }

    jlblStatus.setText(display);
  }
});
```

List Models

The JList class delegates the responsibilities of storing and maintaining data to its data model. **The JList class itself does not have methods for adding or removing items from the list. These methods are supported in ListModel.**

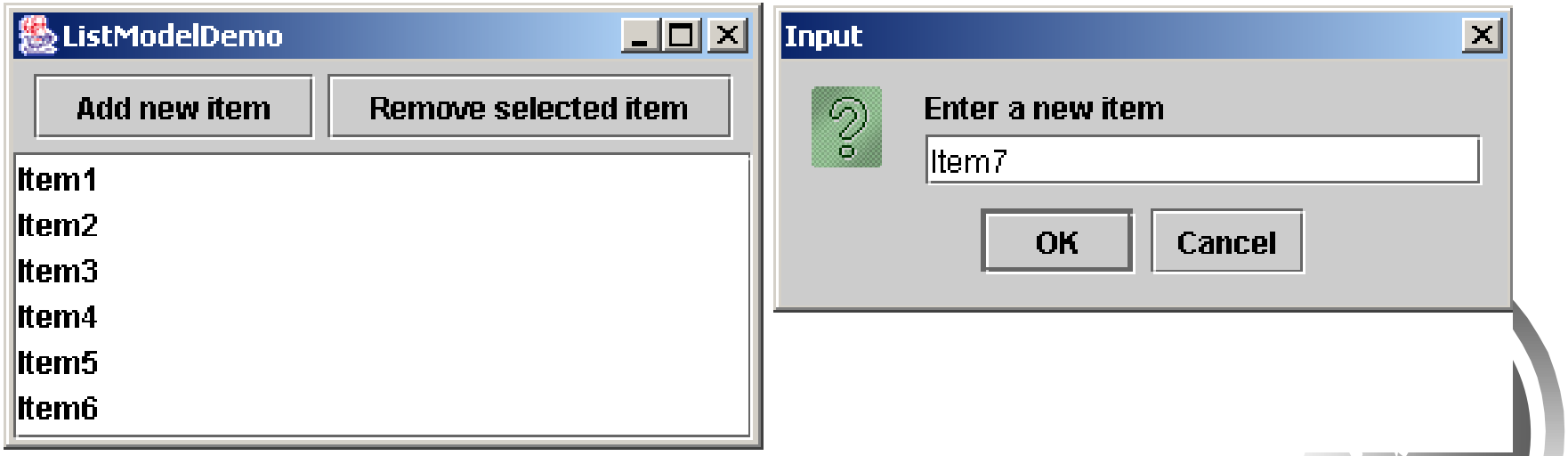


AbstractListModel implements the registration methods in the ListModel, but does not implement the getSize and getElementAt methods.

DefaultListModel extends AbstractListModel and implements the two methods getSize and getElementAt, which are not implemented by AbstractListModel.

Example: List Model Demo

Problem: This example creates a list using a list model and allows the user to add and delete items in the list. When the user clicks the *Add new item* button, an input dialog box is displayed to receive a new item.



ListModelDemo

Run

```
private DefaultListModel<String> listModel
  = new DefaultListModel<String>();
private JList<String> jlst = new JList<String>(listModel);
```

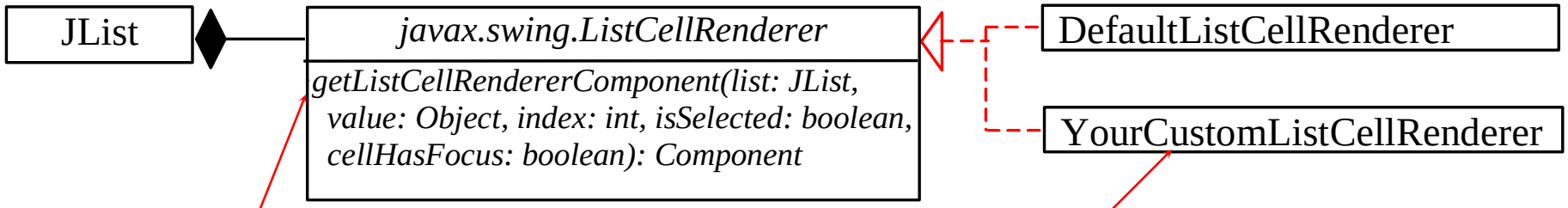
```
// Add items to the list model
listModel.addElement("Item1");
listModel.addElement("Item2");
listModel.addElement("Item3");
listModel.addElement("Item4");
listModel.addElement("Item5");
listModel.addElement("Item6");
```

```
// Register listeners
jbtAdd.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
    String newItem =
      JOptionPane.showInputDialog("Enter a new item");

    if (newItem != null)
      if (jlst.getSelectedIndex() == -1)
        listModel.addElement(newItem);
      else
        listModel.add(jlst.getSelectedIndex(), newItem);
  }
});

jbtRemove.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
    listModel.remove(jlst.getSelectedIndex());
  }
});
```

List Cell Renderer

In addition to delegating data storage and processing to list models, JList delegates the rendering of the list cells to list cell renderers.

By default, JList uses DefaultListCellRenderer to render its cells. The DefaultListCellRenderer class implements ListCellRenderer, extends JLabel, and can display either a string or an icon, **but not both** in the same cell.

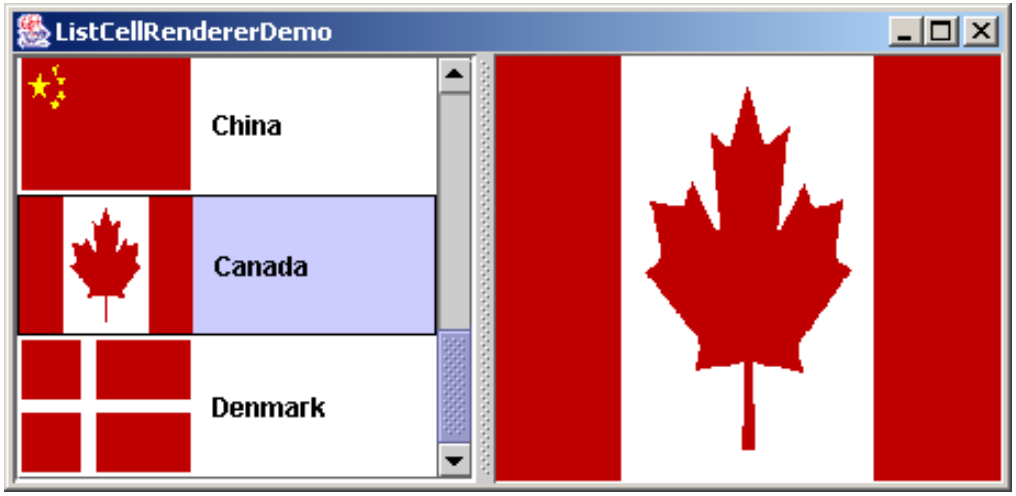


All list cell renderers implement the ListCellRenderer interface, which defines a single method, getListCellRendererComponent, as follows

You can create a custom renderer by implementing ListCellRenderer.

Example: List Cell Renderer Demo

Problem: This example creates a list of countries and displays the country flags and country names in the list. When a country is selected in the list, its flag is displayed in a panel next to the list.



MyListCellRenderer

ListCellRendererDemo

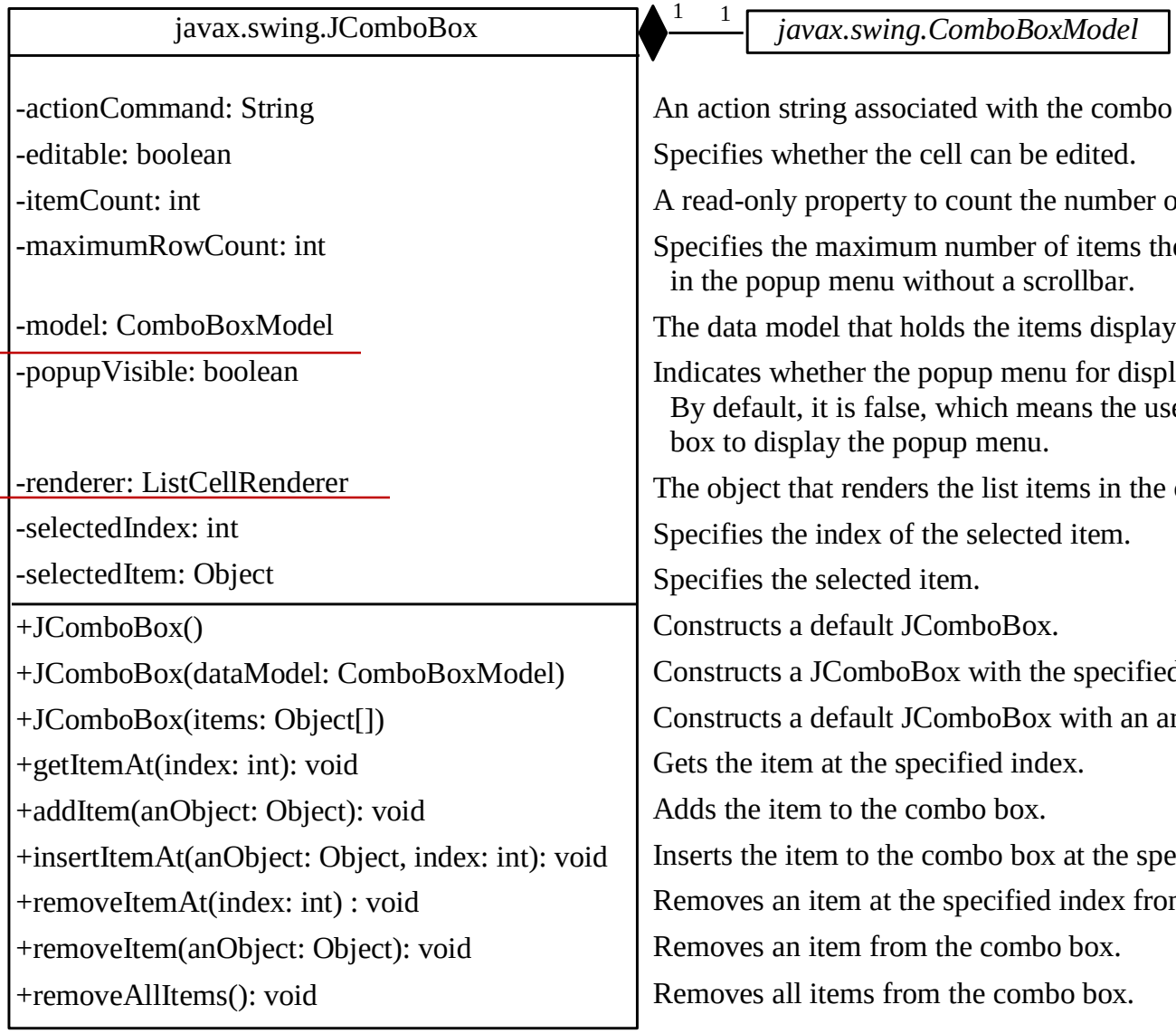
Run

```java
// Create a list model
private DefaultListModel listModel = new DefaultListModel();

// Create a list using the list model
private JList jlstNations = new JList(listModel);
```

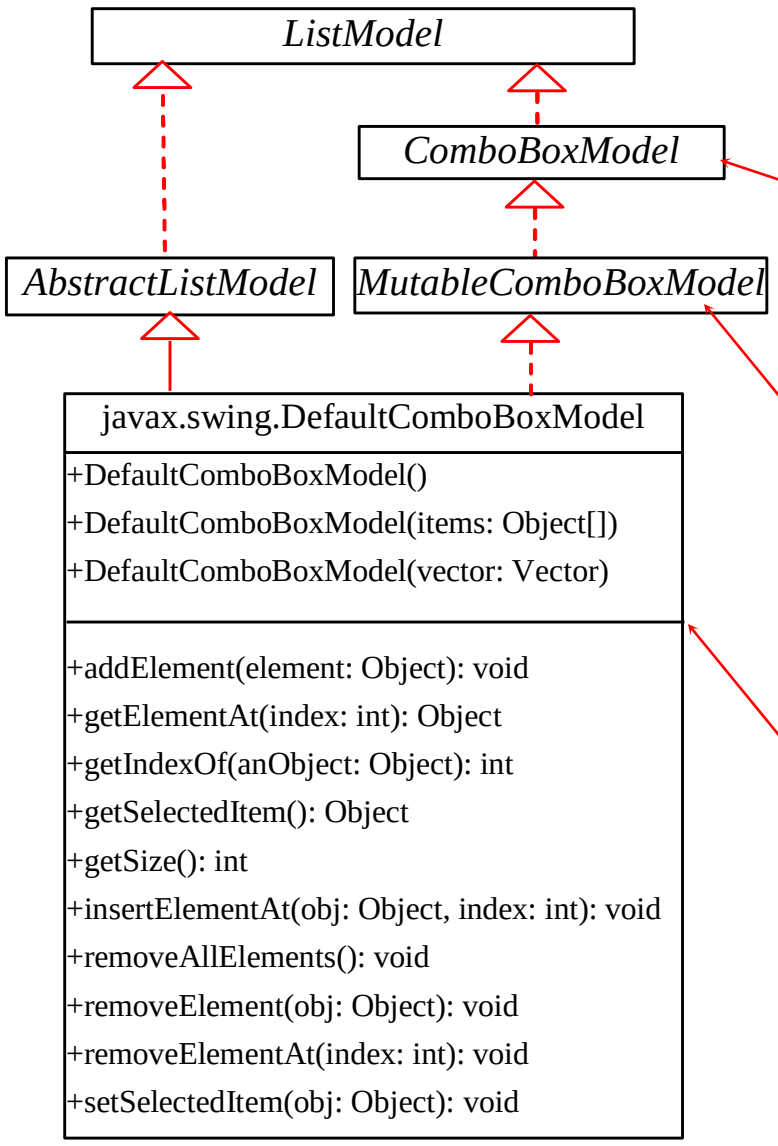
```java
// Load small and large image icons
for (int i = 0; i < NUMBER_OF_NATIONS; i++) {
  icons[i] = new ImageIcon(getClass().getResource(
    "/image/flagIcon" + i + ".gif"));
  listModel.addElement(new Object[]{icons[i], nations[i]});

  bigIcons[i] = new ImageIcon(getClass().getResource(
    "/image/flag" + i + ".gif"));
}

// Set list cell renderer
jlstNations.setCellRenderer(renderer);
```

```java
public class MyListCellRenderer implements ListCellRenderer {
  private JLabel jlblCell = new JLabel(" ", JLabel.LEFT);
  private Border lineBorder =
    BorderFactory.createLineBorder(Color.black, 1);
  private Border emptyBorder =
    BorderFactory.createEmptyBorder(2, 2, 2, 2);

  /** Implement this method in ListCellRenderer */
  public Component getListCellRendererComponent
      (JList list, Object value, int index, boolean isSelected,
       boolean cellHasFocus) {
    Object[] pair = (Object[])value; // Cast value into an array
    jlblCell.setOpaque(true);
    jlblCell.setIcon((ImageIcon)pair[0]);
    jlblCell.setText(pair[1].toString());

    if (isSelected) {
      jlblCell.setForeground(list.getSelectionForeground());
      jlblCell.setBackground(list.getSelectionBackground());
    }
    else {
      jlblCell.setForeground(list.getForeground());
      jlblCell.setBackground(list.getBackground());
    }

    jlblCell.setBorder(cellHasFocus ? lineBorder : emptyBorder);

    return jlblCell;
  }
}
```

JComboBox


javax.swing.JComboBox
-actionCommand: String
-editable: boolean
-itemCount: int
-maximumRowCount: int
-model: ComboBoxModel
-popupVisible: boolean
-renderer: ListCellRenderer
-selectedIndex: int
-selectedItem: Object
+JComboBox()
+JComboBox(dataModel: ComboBoxModel)
+JComboBox(items: Object[])
+getItemAt(index: int): void
+addItem(anObject: Object): void
+insertItemAt(anObject: Object, index: int): void
+removeItemAt(index: int) : void
+removeItem(anObject: Object): void
+removeAllItems(): void
1
1
javax.swing.ComboBoxModel
An action string associated with the combo box.
Specifies whether the cell can be edited.
A read-only property to count the number of items.
Specifies the maximum number of items the combo box can display in the popup menu without a scrollbar.
The data model that holds the items displayed by this combo box.
Indicates whether the popup menu for displaying items is visible. By default, it is false, which means the user has to click the combo box to display the popup menu.
The object that renders the list items in the combo box.
Specifies the index of the selected item.
Specifies the selected item.
Constructs a default JComboBox.
Constructs a JComboBox with the specified combo box model.
Constructs a default JComboBox with an array of items.
Gets the item at the specified index.
Adds the item to the combo box.
Inserts the item to the combo box at the specified index.
Removes an item at the specified index from the combo box.
Removes an item from the combo box.
Removes all items from the combo box.

Combo Box Model



JComboBox delegates the responsibilities of storing and maintaining data to its data model. All combo box models implement the ComboBoxModel interface, which extends the ListModel interface and defines the getSelectedItem and setSelectedItem methods for retrieving and setting a selected item.

The methods for adding and removing items are defined in the MutableComboBoxModel interface

DefaultComboBoxModel provides a concrete implementation for ComboModel.

Example: Combo Box Cell Renderer Demo

Problem: This example creates a combo box that contains a list of countries and displays the country flags and country names in the list cell. When a country is selected in the list, its flag is displayed in a panel below the combo box.

ComboBoxCellRenderer

Run

```
private final static int NUMBER_OF_NATIONS = 7;
private String[] nations = new String[] {"Denmark",
  "Germany", "China", "India", "Norway", "UK", "US"};
private ImageIcon[] icons = new ImageIcon[NUMBER_OF_NATIONS];
private ImageIcon[] bigIcons = new ImageIcon[NUMBER_OF_NATIONS];

// Create a combo box model
private DefaultComboBoxModel model = new DefaultComboBoxModel();

// Create a combo box with the specified model
private JComboBox jcboCountries = new JComboBox(model);

// Create a list cell renderer
private MyListCellRenderer renderer = new MyListCellRenderer();

// Create a label for displaying iamge
private JLabel jlblImage = new JLabel("", JLabel.CENTER);
```

```
public ComboBoxCellRendererDemo() {
  // Load small and large image icons
  for (int i = 0; i < NUMBER_OF_NATIONS; i++) {
    icons[i] = new ImageIcon(getClass().getResource(
      "/image/flagIcon" + i + ".gif"));
    model.addElement(new Object[]{icons[i], nations[i]});

    bigIcons[i] = new ImageIcon(getClass().getResource(
      "/image/flag" + i + ".gif"));
  }

  // Set list cell renderer for the combo box
  jcboCountries.setRenderer(renderer);
  jlblImage.setIcon(bigIcons[0]);
  add(jcboCountries, java.awt.BorderLayout.NORTH);
  add(jlblImage, java.awt.BorderLayout.CENTER);

  // Register listener
  jcboCountries.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(java.awt.event.ActionEvent e) {
      jlblImage.setIcon(bigIcons[jcboCountries.getSelectedIndex()]);
    }
  });
}
```