Chapter 39 MVC and Swing MVC Components
Objectives

- To use the model-view-controller approach to separate data and logic from the presentation of data (§39.2).
- To implement the model-view-controller components using the JavaBeans event model (§39.2).
- To explain the Swing model-view-controller architecture (§39.4).
- To use JSpinner to scroll the next and previous values (§39.5).
- To create custom spinner models and editors (§39.6).
- To use JList to select single or multiple items in a list (§39.7).
- To add and remove items using `ListModel` and `DefaultListModel` (§39.8).
- To render list cells using a default or custom cell renderer (§39.9).
- To create custom combo box models and renderers (§39.10).
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.

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.
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.

**Methods**

- `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.
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.

- **model**: CircleModel
  - Stores the circle model.

+ actionPerformed(e: ActionEvent): void
+ paintComponent(g: Graphics): void
  - Implements this method to update the view.
  - Paints the view.
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.
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: Swing components are designed using the MVC architecture. Each Swing GUI component is a view that uses a model to store data. 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.

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

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

Liang, Introduction to Java Programming, Ninth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.
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.

```
javax.swing.JSpinner

-model: SpinnerModel
-editor: JComponent

+JSpinner()
+JSpinner(model: SpinnerModel)
+getValue(): Object
+getNextValue(): Object
+getPreviousValue(): Object
+setValue(value: Object): void
+addChangeListener(l: ChangeListener): void
+removeChangeListener(l: ChangeListener): void
```

```
javax.swing.SpinnerModel

Specifies a model with get/set methods.
Specifies an editor with get/set methods.
Constructs a JSpinner with a SpinnerNumberModel with initial value 0 and no minimum or maximum limits.
Constructs a JSpinner with a specified SpinnerModel.
Gets the next element value in this JSpinner.
Gets the next element value in this JSpinner.
Gets the current element value in this JSpinner.
Sets the current element value.
Adds a listener for value change.
Removes a listener.
```
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.
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.

```java
javafx.swing.SpinnerModel

+getPreviousValue(): Object
+getNextValue(): Object
+getValue(): Object
+setValue(value: Object): void
+addChangeListener(l: ChangeListener): void
+removeChangeListener(l: ChangeListener): void
```

Gets the previous element value.
Gets the next element value.
Gets the current element value.
Sets the current element value.
Adds a listener for value change.
Removes a listener.
SpinnerListModel is a simple implementation of SpinnerModel whose values are stored in a java.util.List.

SpinnerListModel

```
javax.swing.SpinnerModel
```

```
javax.swing.AbstractSpinnerModel
```

```
javax.swing.SpinnerListModel
```

- list: java.util.List

+ SpinnerListModel()
+ SpinnerListModel(values: List)
+ SpinnerListModel(values: Object[])
+ getList(): List
+ setList(list: List): void

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`.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SpinnerNumberModel()</code></td>
<td>Constructs an unbounded <code>SpinnerNumberModel</code> with an initial value of zero and stepSize equal to one.</td>
</tr>
<tr>
<td><code>SpinnerNumberModel(value: int, minimum: int, maximum: int, stepSize: int)</code></td>
<td>Constructs a <code>SpinnerNumberModel</code> with the specified initial value, minimum/maximum bounds, and stepSize in int.</td>
</tr>
<tr>
<td><code>SpinnerNumberModel(value: double, minimum: double, maximum: double, stepSize: double)</code></td>
<td>Constructs a <code>SpinnerNumberModel</code> with the specified initial value, minimum/maximum bounds, and stepSize in double.</td>
</tr>
<tr>
<td><code>SpinnerNumberModel(value: Number, minimum: Comparable, maximum: Comparable, stepSize: Number)</code></td>
<td>Constructs a <code>SpinnerNumberModel</code> that represents a closed sequence of numbers from minimum to maximum.</td>
</tr>
</tbody>
</table>

```java
javax.swing.SpinnerNumberModel
```

```java
javax.swing.AbstractSpinnerModel
```
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`.

```java
javafx.swing.SpinnerModel
```

```java
javafx.swing.AbstractSpinnerModel
```

```java
javafx.swing.SpinnerDateModel
```
- start: java.lang.Comparable
- end: java.lang.Comparable
- calendarField: int
- value: java.util.Calendar

+ SpinnerDateModel()
+ SpinnerDateModel(value: Date, start: Comparable, end: Comparable, calendarField: int)

- 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.
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.

```
javax.swing.JList

-model: ListModel
-selectionMode: int
-selectionModel: ListSelectionModel
-cellRenderer: ListCellRenderer

ListModel ← AbstractListModel ← DefaultListModel

ListSelectionModel ← DefaultListSelectionModel

ListCellRenderer ← DefaultListCellRenderer
```
The **JList** Class

```
javafx.swing.JList

- cellRenderer: ListCellRenderer
- fixedCellHeight: int
- fixedCellWidth: int
- layoutOrientation: int
- model: ListModel
- selectedIndex: int
- selectedIndices: int[]
- selectedValue: Object
- selectedValues: Object[]
- selectedBackground: int
- selectedForeground: int
- selectionMode: int
- selectionModel: ListSelectionModel
- visibleRowCount: int

+ JList()
+ JList(dataModel: ListModel)
+ JList(listData: Object[])
+ JList(listData: Vector)

+ setListData(listData: Object[]): void
+ setListData(listData: Vector): void
```

- The object that renders the list items.
- The fixed cell height value in pixels.
- The fixed cell width value.
- Defines the way list cells are laid out.
- Specifies the list model for this list.
- The index of the first selected item in this list.
- An array of all of the selected indices in increasing order.
- The first selected value.
- An array of the values for the selected values in increasing index order.
- The background color of the selected items.
- The foreground color of the selected items.
- Specifies whether single- or multiple-interval selections are allowed.
- Specifies a selection model.
- The preferred number of rows to display without using a scroll bar (default: 8).
- Constructs a default JList.
- Constructs a JList with the specified model.
- Constructs a JList with the data specified in the array.
- Constructs a JList with the data specified in the vector.
- Sets an array of objects as data for the list.
- Sets a vector of objects as data for the list.
The **layoutOrientation** property

JList jlst = new JList();

jlst.setLayoutOrientation(int);
The `selectionMode` property

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

- `JList.SINGLE_SELECTION`
- `JList.SINGLE_INTERVAL_SELECTION`
- `JList.MULTIPLE_INTERVAL_SELECTION`
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.

![List Properties Demo](image-url)
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.

Liang, Introduction to Java Programming, Ninth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.
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:

```
getListCellRendererComponent(list: JList, value: Object, index: int, isSelected: boolean, cellHasFocus: boolean): Component
```

You can create a custom renderer by implementing ListCellRenderer.

JList

java.swing.ListCellRenderer

DefaultListCellRenderer

YourCustomListCellRenderer
**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.
**JComboBox**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>actionCommand:</code> String</td>
<td>An action string associated with the combo box.</td>
</tr>
<tr>
<td><code>editable:</code> boolean</td>
<td>Specifies whether the cell can be edited.</td>
</tr>
<tr>
<td><code>itemCount:</code> int</td>
<td>A read-only property to count the number of items.</td>
</tr>
<tr>
<td><code>maximumRowCount:</code> int</td>
<td>Specifies the maximum number of items the combo box can display in the popup menu without a scrollbar.</td>
</tr>
<tr>
<td><code>model:</code> ComboBoxModel</td>
<td>The data model that holds the items displayed by this combo box.</td>
</tr>
<tr>
<td><code>popupVisible:</code> boolean</td>
<td>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.</td>
</tr>
<tr>
<td><code>renderer:</code> ListCellRenderer</td>
<td>The object that renders the list items in the combo box.</td>
</tr>
<tr>
<td><code>selectedIndex:</code> int</td>
<td>Specifies the index of the selected item.</td>
</tr>
<tr>
<td><code>selectedItem:</code> Object</td>
<td>Specifies the selected item.</td>
</tr>
</tbody>
</table>

- **Constructor:**
  - `JComboBox()`: Constructs a default JComboBox.
  - `JComboBox(dataModel: ComboBoxModel)`: Constructs a JComboBox with the specified combo box model.
  - `JComboBox(items: Object[])`: Constructs a default JComboBox with an array of items.

- **Methods:**
  - `getItemAt(index: int): void` | Gets the item at the specified index. |
  - `addItem(anObject: Object): void` | Adds the item to the combo box. |
  - `insertItemAt(anObject: Object, index: int): void` | Inserts the item to the combo box at the specified index. |
  - `removeItemAt(index: int): void` | Removes an item at the specified index from the combo box. |
  - `removeItem(anObject: Object): void` | Removes an item from the combo box. |
  - `removeAllItems(): void` | 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 ComboBoxModel.
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.