GUI Programming IV

Swing MVC Components

Swing and Pattern

- Each GUI component has three characteristics:
  - Content
    - Such as the state of a button (pressed, released etc.)
  - Visual appearance
    - Painted on the screen with colors, sizes etc.
    - Different on different platforms
  - Behaviour
    - Interaction with events

Some components are easy to use as they do have a lot data to deal with
- Such as JButtons, JLabels, JMenus

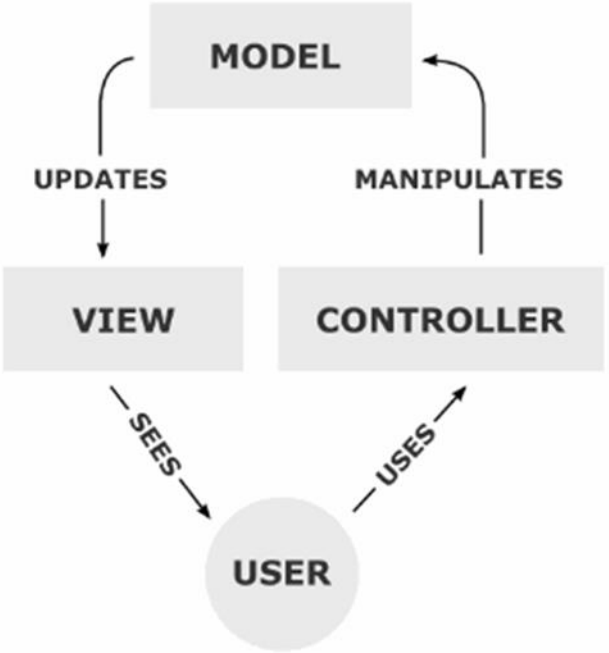
Some components are not so easy to use as they maintain a lot of data that the program need to provide, update etc
- Such as JList, Jtable, JTree, JTextComponent

OO Design Principle: ***Don't make one object responsible for too much***

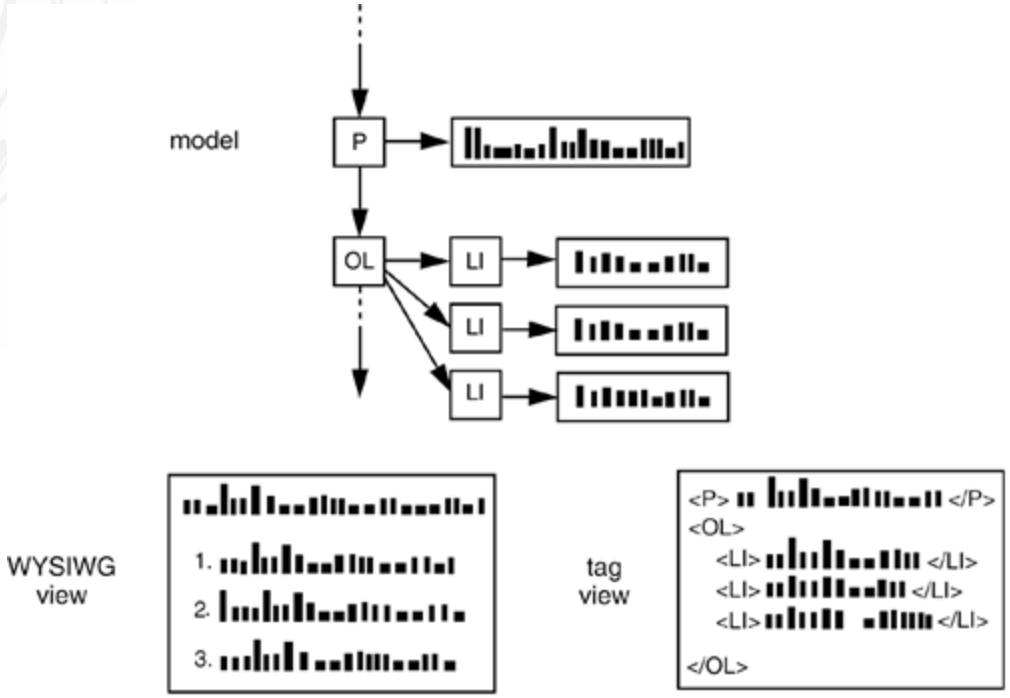
Model-View-Controller Design Pattern

- The fundamental idea of MVC (Model-View-Controller) is a separation of the domain logic and the GUI objects into the Model and the View

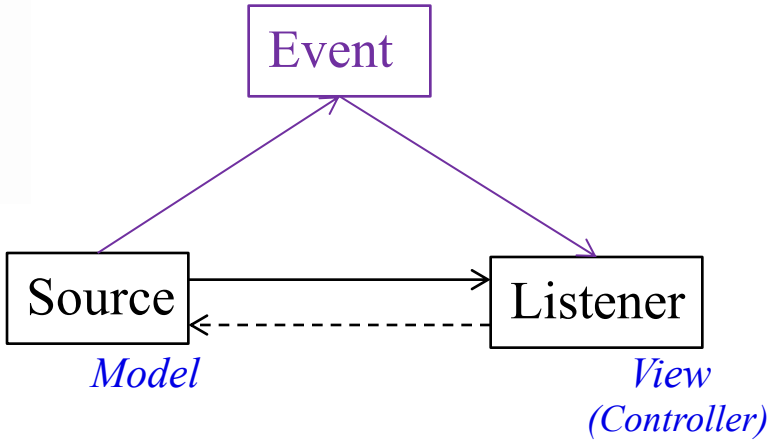
  - **Model**
    - Stores the content
  - **View**
    - Displays the content
  - **Controller**
    - Handles user input



Example: Model and Views



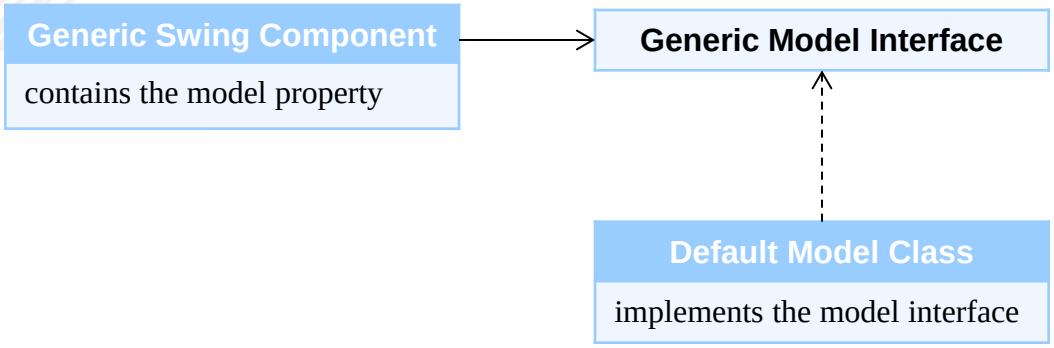
Event Delegation Model and MVC



Swing MVC Components

- Swing components are designed with MVC pattern
  - One model can have multiple views
  - A view is synchronised with the model
  - By separating out the model from views enables pluggable look-and-feel (PLAF) architecture
  - more work is delegated to the view and the controller is not implemented separately
- Swing provides a ***wrapper class*** for each component to store the model and the view
- Programmers do not have to work directly with the view
- Programmers do not need always to think about the MVC to use Swing components; but it will be useful to understand and use some complex components as you need to deal with their model classes
  - **`JList`**, **`JTable`**, **`JTree`**, **`JTextComponent`** etc.

Swing MVC Architecture



MVC in Swing

- View classes *(output only)*
  - **`JLabel`**, **`JProgressBar`**
- Controller classes *(input only)*
  - **`JButton, JMemu`**
- View/Controller classes
  - **`JCheckBox, JScrollBar, JSlider`**, text components
- Model/View/Controller classes
  - **`JList, JTextPane, JTable, JTree, JEditorPane`**
    - **`AbstractXxxModel`** implements **`XxxModel`**
    - **`DefaultXxxModel`** extends **`AbstractXxxModel`**
  - Each component has a UI delegate to package its view and controller

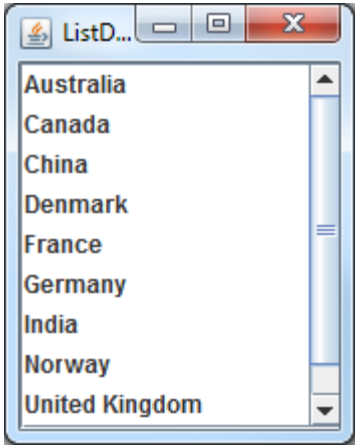
Basic Features of JList

- JList shows a number of items inside a single box

```
// Declare an array of Strings for countries
String[] countries= {"Australia", "Canada",
                     "China", "Denmark", . . .};

// The list for selecting countries
JList theList = new JList(countries);

// Add the list to the frame
add(new JScrollPane(theList));
```



- What about if you want to edit the collection of list values
  - One potential solution (using a collection to store values)

```
// Declare an vector of Strings for countries
Vector<String> countries = new Vector<String>();
values.addElement({"Australia");
values.addElement({" Canada");
. . .

// The list for selecting countries
JList theList = new JList(countries);
```

  - You ***can*** add/remove elements of the vector, but the list does not know and ***cannot*** update its view!

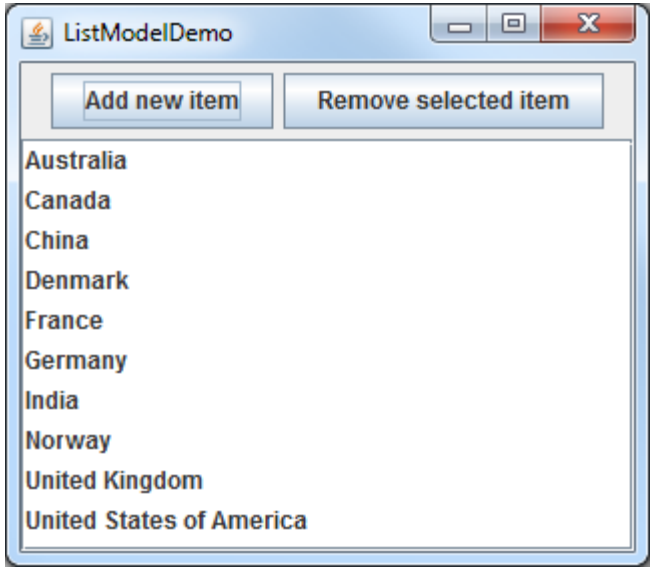
`JList` and its Model

- In order to manage the values of a list, you should construct a model of the **`JList`**
- The **`JList`** contains a model class called **`DefaultListModel`**
- A **`DefaultListModel`** object should be constructed to manage the values of the list

```
// Declare an model for countries
DefaultListModel countries = new DefaultListModel();
values.addElement({"Australia");
values.addElement({" Canada");
. . .

// The list for selecting countries
JList theList = new JList(countries);

// Manage the list
values.removeElement({"Canada");
values.addElement({"New Zealand");
```



DefaultListModel

AbstractListModel	
`void addListDataListener(ListDataListener l)`	Adds a listener to the list that's notified each time a change to the data model occurs.
`protected void fireContentsChanged( Object source, int index0, int index1)`	AbstractListModel subclasses must call this method after one or more elements of the list change.

DefaultListModel	
`void add(int index, E element)`	Inserts the specified element at the specified position in this list.
`void addElement(E element)`	Adds the specified component to the end of this list.
`E get(int index)`	Returns the element at the specified position in this list.
`void insertElementAt(E element, int index)`	Inserts the specified element as a component in this list at the specified index.
`boolean removeElement(Object obj)`	Removes the first (lowest-indexed) occurrence of the argument from this list.
`void removeElementAt(int index)`	Deletes the component at the specified index.
`E set(int index, E element)`	Replaces the element at the specified position in this list with the specified element.
`Object[] toArray()`	Returns an array containing all of the elements in this list in the correct order.
`String toString()`	Returns a string that displays and identifies this object's properties.

Other Swing MVC Components

- **`JTable`**
  - Its models: **`TableModel`**, **`TableColumnModel`**
  - Its view/controller: **`TableCellRenderer`**, **`TableCellEditor`**
- **`JTree`**
  - Its model: **`TreeModel`**
  - Its view/controller: **`TreeCellRenderer, TreeCellEditor`**
- **`JTextComponent`**
  - Its model: **`Document`**
  - Its view/controller is itself