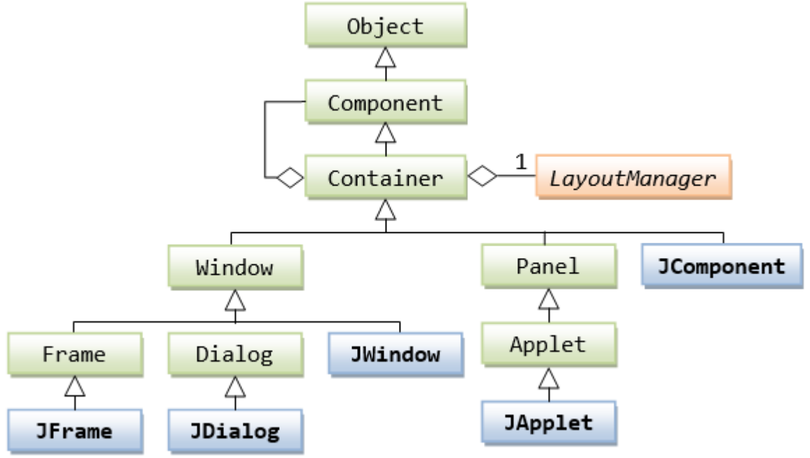
Basics of the Object-Oriented Programming
Graphical User Interface (II)

Associate Professor Viorica Rozina Chifu
viorica.chifu@cs.utcluj.ro

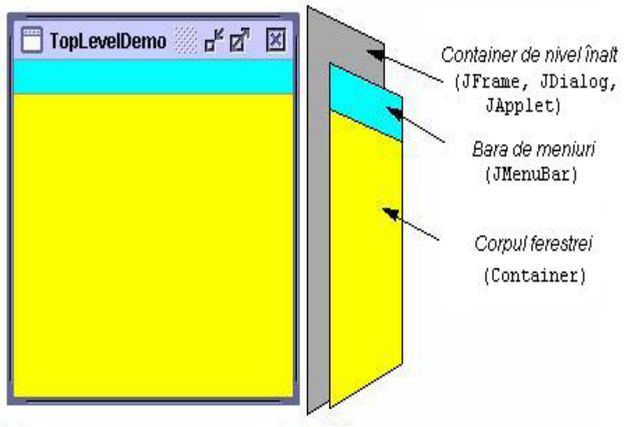
Swing - Windows

- **JFrame, JDialog and JApplet**
  - High level containers
  - Allow to display graphical components on the screen
    - » Each component can be displyed in only one container
- A container can be encapsulated in other containers
  - At a moment of time, a component can be part of a hierarchy



Swing - Window

- An object representing a window contains:
  - A reserved area for the menu bar
  - The body of the window for putting the components



Swing - Windows

- The body of the window is an instance of the **Container** class
  - Can be obtained with **getContentPane** method
- Putting and arranging the components on the window area
  - Is made with a **Container** object

```
JFrame jf = new JFrame();
 jf.getContentPane().setLayout(new FlowLayout());
 jf.getContentPane().add(new JButton("OK"));
```

Swing - Window

- The default behaviour of a **JFrame** object when we close the window:
  - Hide the window when the user press the close button
- The default behaviour of a **JFrame** object can be modified with **setDefaultCloseOperation** method
  - **setDefaultCloseOperation** receives as arguments constants from **WindowConstants** and **JFrame** classes

```
jf.setDefaultCloseOperation (WindowConstants.HIDE_ON_CLOSE);
jf.setDefaultCloseOperation (WindowConstants.DO_NOTHING_ON_CLOSE);
jf.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
```

Swing - Windows

- From the viewpoint of using windows, applications are divided into:
  - SDI (Single Document Interface)
    - » Programs that manage at a moment only one window containing components with which the user interacts
  - MDI (Multiple Document Interface)
    - » Programs that manage more than one window
      - The main window of the application contains other windows with similar functionalities

Swing - Windows

- SDI (Single Document Interface) - Example

```
import javax.swing.*;
public class MyFrame extends JFrame {
        public MyFrame() {
        // Set the size and title of the JFrame
        setSize(400, 300);
        setTitle("My JFrame");

        // Set the default close operation
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Create a JPanel to hold the components
        JPanel panel = new JPanel();

        // Create a JButton and add it to the panel
        JButton button = new JButton("Click me!");
        panel.add(button);
```

```
        // Create a JLabel and add it to the panel
        JLabel label = new JLabel("Hello, world!");
        panel.add(label);
        // Add the panel to the JFrame
        add(panel);

        // Display the JFrame
        setVisible(true);
    }

    public static void main(String[] args) {
        // Create an instance of MyFrame
        MyFrame myFrame = new MyFrame();
    }
}
```

Swing – Internal Frames

- Internal Frames
  - Looks and have the same functionality as **JFrame** windows
  - How is different is their management
- **JInternalFrame** class
  - Allows to create windows inside the other windows
  - The **JInternalFrame** object is placed on a **DesktopPane** container
    - » Then, the **DesktopPane** container will be placed on a **JFrame**
    - » **DesktopPane** class knows how to manage the internal frames

Swing – Internal Frame (Example)

```
import javax.swing .*;
class InternalFrame extends JInternalFrame {
   static int n = 0; // nr. de ferestre interne
   static final int x = 30, y = 30;
   public InternalFrame () {
     //create a frame with the specified title, resizable
      closable,  maximizable
     super("Document #" + (++ n), true , true , true);
     setLocation (x*n, y*n);
     setSize(new Dimension(200, 100) );}}
```

```
class MainFrame extends JFrame {
   public MainFrame (String titlu) {
       super(titlu);
       setSize(300, 200) ;
        setDefaultCloseOperation(JFrame. EXIT_ON_CLOSE );
        InternalFrame fin1 = new InternalFrame ();
        fin1.setVisible(true);
        InternalFrame fin2 = new InternalFrame ();
        fin2.setVisible(true);
       JDesktopPane desktop = new JDesktopPane();
       desktop.add(fin1);
       desktop.add(fin2);
       setContentPane(desktop);
       fin2.moveToFront();
       }}

public class TestInternalFrame {
  public static void main(String args []) {
         new MainFrame ("Test internal frames").setVisibile(true);
}}
```

The architecture of Swing model

- Is similar with MVC architecture (model-view-controller)
- MVC specifies the decomposition of an application in three parts:
  - Model - represents the date of the application
  - Presentation - specifiy the data visualization model
  - Control - transform the user actions on the visual components in events that automatically update their model
- Is an architecture with a separable model, in which the date of the components are separated by their visual representation
- Couple the presentation and the control parts because there is a strong binding between them
- The components with different representation can have the same model
- There are components having associated more models

The architecture of Swing model

Model	**Component**
ButtonModel	JButton, JToggleButton, JCheckBox, JRadioButton, JRadioButtonMenuItem, JCheckBoxMenuItem, JMenu, JMenuItem
ComboBoxModel	JComboBox
BoundedRangeModel	JProgressBar, JScrollBar, JSlider
SingleSelectionModel	JTabbedPane
ListModel	JList
ListSelectionModel	JList
TableModel	JTable
TableColumnModel	JTable
TreeModel	JTree
TreeSelectionModel	JTree
Document	JEditorPane, JTextPane, JTextArea, JTextField, JPasswordField

The architecture of Swing model

- Each component
  - Has a default model
    - » **setModel**, **getModel** – methods that access the model of an object with specific arguments for each component
  - Allow to replace the default model with a new one, when we want
- Creating a class that represent a model is made by:
  - Implementing the considered interface and the methods defined in the interface
  - Extending the default class provided by the SWING API and overriding the methods that we are interested

The architecture of Swing model

- In Swing, when the model of a component changes, the presentation must be updated to reflect the new data
- This can be done in two ways:
  - Lightweight notification
  - Stateful notifications

Events handing in Swing - Lightweight notification

- Lightweight notifications are used for components that do not require heavy processing to update their presentation
- In lightweight notifications, the model of the component notifies the view that a change has occurred, and the view updates itself accordingly
- In lightweight notification listener objects call component-specific methods to find out what has changed
  - Example of component-specific methods
    - » **getValue**() for the BoundedRangeModel
    - » **isSelected**() for the ButtonModel
    - » **getSelectedIndex**() for the SingleSelectionModel
- The ChangeListener interface is used for models that support this approach, including BoundedRangeModel, ButtonModel, and SingleSelectionModel

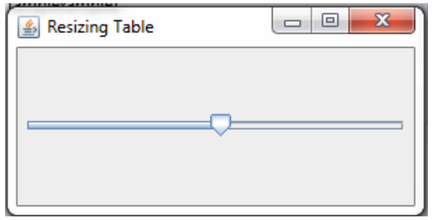
Events handing in Swing - Lightweight notification

- **ChangeListener** interface
  - Contains the method public void **stateChanged(ChangeEvent** e) which is called when a change event is fired by the component's model
  - This method is implemented by the listener object to handle the event and perform any necessary actions based on the changes made
- Registration and removing listener objects is done with methods:
  - **addChangeListener**
  - **removeChangeListener**

Model	Listener	Tip Eveniment
BoundedRangeModel	ChangeListener	ChangeEvent
ButtonModel	ChangeListener	ChangeEvent
SingleSelectionModelModel	ChangeListener	ChangeEvent

Events handing in Swing - Lightweight notification (Example)

```
package tableexample;
import javax.swing.*;
import javax.swing.event.*;
public class NotificationDemo {
  public static void main(String args[]) {
    JFrame frame = new JFrame("Resizing Table");
    JSlider slider= new JSlider();
    BoundedRangeModel model=slider.getModel();
    model.addChangeListener( new ChangeListener() {
      @Override
       public void stateChanged(ChangeEvent e) {
            BoundedRangeModel m = (BoundedRangeModel)e.getSource();
              System.out.println(m.getValue());    } });
    frame.getContentPane().add(slider);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(300, 150);
    frame.setVisible(true);
  }}
```




```
NotificationDemo [Java Application] C:\Program Files\Java\jdk1.8.0_91\bin\javaw.exe (Apr 26, 2021, 11:59:01 AM)
50
51
51
```

Events handing in Swing - Lightweight notification

```
import javax.swing.*;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;
public class NotificationDemo1 {
  public static void main(String args[]) {
    JFrame frame = new JFrame("Resizing Table");
    JSlider slider= new JSlider();
    slider.addChangeListener( new ChangeListener() {
      @Override
       public void stateChanged(ChangeEvent e) {
          JSlider s=(JSlider)e.getSource();
          System.out.println(s.getValue());   }});
  frame.getContentPane().add(slider);
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  frame.setSize(300, 150);
  frame.setVisible(true);
  }}
```

- If you don’t want to work with the model instance
  - Same classes allow to directly recording listener for the component
    - » In this case the event source is the component and not the model

Events handing in Swing - Statefull notification

- Stateful notifications are used for components that require more processing to update their presentation, such as tables or lists with large amounts of data
- In stateful notifications, the model of the component notifies the view that a change has occurred and includes the state of the change
- The view then updates itself based on the state of the change, rather than re-rendering the entire component

Model	Listener	Tip Eveniment
ListModel	ListDataListener	ListDataEvent
ListSelectionModel	ListSelectionListener	ListSelectionEvent
ComboBoxModel	ListDataListener	ListDataEvent
TreeModel	TreeModelListener	TreeModelEvent
TreeSelectionModel	TreeSelectionListener	TreeSelectionEvent
TableModel	TableModelListener	TableModelEvent
TableColumnModel	TableColumnModelListener	TableColumnModelEvent
Document	DocumentListener	DocumentEvent
Document	UndoableEditListener	UndoableEditEvent

Events handing in Swing - Statefull notification (Example)

```
import javax.swing.*;
import javax.swing.event.*;
public class StatefulNotificationExample extends JFrame {
    private JList<String> list;
    private DefaultListModel<String> model;
    public StatefulNotificationExample() {
        model = new DefaultListModel<>();
        model.addElement("Apple");
        model.addElement("Banana");
        model.addElement("Cherry");
        model.addElement("Durian");
        list = new JList<>(model);
        list.addListSelectionListener(new ListSelectionListener() {
            @Override
            public void valueChanged(ListSelectionEvent e) {
                if (!e.getValueIsAdjusting()) {
                    int index = list.getSelectedIndex();
                    if (index != -1) {
                        String selectedItem = model.getElementAt(index);
```

```
JOptionPane.showMessageDialog(null, "You selected: " + selectedItem);
                }            }            }        });
JScrollPane scrollPane = new JScrollPane(list);
        getContentPane().add(scrollPane);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
pack(); //  sizes the frame so that all its contents are at or above their preferred size
setVisible(true);
    }
public static void main(String[] args) {
        new StatefulNotificationExample();}}
```

- ✓ In this example, we create a JList and add a DefaultListModel with some elements.
- ✓ We then add a ListSelectionListener to the JList that listens for changes in selection state.
- ✓ When a selection change occurs, we check if the value is adjusting (to avoid firing the event multiple times during a drag and drop operation) and get the selected index.
- ✓ If the index is not -1, we get the element at that index from the model and display a message using JOptionPane to indicate which item was selected.
- ✓ This is an example of a stateful notification because the program keeps track of the selected item and notifies the user when a selection change occurs.

Swing – Using the components

- Atomic components
  - Are components with a simple functionality and are used to build more complex user interfaces
    - » Labels: **JLabel**
    - » Simple buttons or buttons with two states: **JButton, JCheckBox, JRadioButton, JToggleButton**
      - More radio buttons can be grouped using **ButtonGroup** class which allows to select only one button at a moment of time
      - Components for progress: **JSlider, JProgressBar, JScrollBar**
    - » Separators: **JSeparator**

Swing – Using the components

- Atomic components
  - Atomic components are usually combined with other components to create more complex user interfaces
    - » For example, you might use a JLabel to display a message, a JTextField to allow the user to enter text, and a JButton to initiate an action based on the text entered
  - By combining atomic components in various ways, you can create a wide variety of user interfaces to suit your needs

Swing – Example of using atomic components (1)

```java
import java.awt.*;
import java.awt.event.*;
import java.net.URL;
import javax.swing.*;

/** Test setting Swing's JComponents properties and appearances */
@SuppressWarnings("serial")
public class SwingJComponentSetterTest extends JFrame {

   /** Constructor to setup the GUI */
   public SwingJComponentSetterTest() {
      Container cp = getContentPane();
      cp.setLayout(new FlowLayout(FlowLayout.CENTER, 10, 10));

      // Create a JLabel with text and icon and set its appearances
      JLabel label = new JLabel("JLabel", SwingConstants.CENTER);
      label.setFont(new Font(Font.DIALOG, Font.ITALIC, 14));
      label.setOpaque(true);  // needed for JLabel to show the background color
      label.setBackground(new Color(204, 238, 241));  // light blue
      label.setForeground(Color.RED);                 // foreground text color
      label.setPreferredSize(new Dimension(120, 80));
      label.setToolTipText("This is a JLabel");  // Tool tip
      cp.add(label);

      // Create a JButton with text and icon and set its appearances
      JButton button = new JButton();
      button.setText("Button");
      button.setVerticalAlignment(SwingConstants.BOTTOM);  // of text
      button.setHorizontalAlignment(SwingConstants.RIGHT); // of text
      button.setFont(new Font(Font.SANS_SERIF, Font.BOLD, 15));
      button.setBackground(new Color(231, 240, 248));
      button.setForeground(Color.BLUE);
      button.setPreferredSize(new Dimension(150, 80));
      button.setToolTipText("This is a JButton");
      button.setMnemonic(KeyEvent.VK_B);  // can activate via Alt-B (buttons only)
      cp.add(button);
```

```java
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setTitle("JComponent Test");
      setLocationRelativeTo(null); // center window on the screen
      setSize(500, 150);  // or pack()
      setVisible(true);
   }

   /** The entry main() method */
   public static void main(String[] args) {
         new SwingJComponentSetterTest(); // Let the constructor do the job
      }
   }
```

Swing – Example of using atomic components (2)

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

public class Example extends JFrame implements
ActionListener {
    private JCheckBox checkBox1, checkBox2;
    private JRadioButton radio1, radio2, radio3;
    private ButtonGroup radioGroup;

    public Example() {
        super("Example");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(300, 200);
        setLayout(new GridLayout(5, 1));

        checkBox1 = new JCheckBox("Option 1");
        checkBox2 = new JCheckBox("Option 2");
        radio1 = new JRadioButton("Option 1");
        radio2 = new JRadioButton("Option 2");
        radio3 = new JRadioButton("Option 3");
```

```
        radioGroup = new ButtonGroup();
        radioGroup.add(radio1);
        radioGroup.add(radio2);
        radioGroup.add(radio3);
        add(new JLabel("Checkboxes:"));
        add(checkBox1);
        add(checkBox2);
        add(new JLabel("Radio buttons:"));
        add(radio1);
        add(radio2);
        add(radio3);

        checkBox1.addActionListener(this);
        checkBox2.addActionListener(this);
        radio1.addActionListener(this);
        radio2.addActionListener(this);
        radio3.addActionListener(this);

        setVisible(true);
    }
```

Swing – Example of using atomic components (3)

```
public void actionPerformed(ActionEvent e) {
        if (e.getSource() == checkBox1) {
            if (checkBox1.isSelected()) {
                System.out.println("Option 1 selected");
            } else {
                System.out.println("Option 1 unselected");
            }
        } else if (e.getSource() == checkBox2) {
            if (checkBox2.isSelected()) {
                System.out.println("Option 2 selected");
            } else {
                System.out.println("Option 2 unselected");
            }
        } else if (e.getSource() == radio1) {
            System.out.println("Option 1 selected");
        } else if (e.getSource() == radio2) {
            System.out.println("Option 2 selected");
        } else if (e.getSource() == radio3) {
            System.out.println("Option 3 selected");
        }
    }
    public static void main(String[] args) {
        new Example();    }}
```

- ✓ This example creates a JFrame with two JCheckBox components and three JRadioButton components
- ✓ The JCheckBox components allow the user to select multiple options, while the JRadioButton components allow the user to select only one option
- ✓ The example also implements the ActionListener interface, so it can handle events when the user interacts with the components.
- ✓ When the user selects or deselects an option, or selects a radio button, the corresponding message is printed to the console

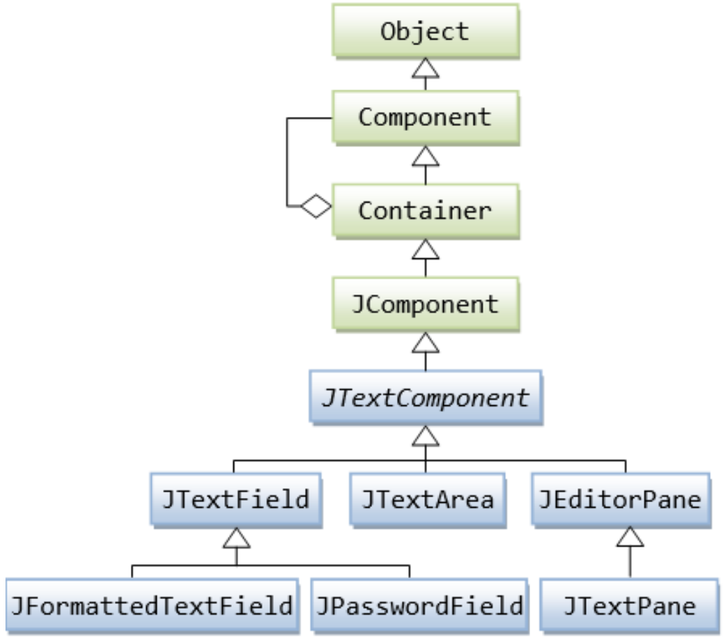
Swing – Example of using atomic components (4)

```
import javax.swing.*;

public class SeparatorExample {
   public static void main(String[] args) {
      JFrame frame = new JFrame("Separator Example");
      JPanel panel = new JPanel();
      panel.add(new JLabel("Name:"));
      panel.add(new JTextField(10));
      panel.add(new JSeparator(SwingConstants.VERTICAL));
      panel.add(new JLabel("Age:"));
      panel.add(new JTextField(3));
      frame.add(panel);
      frame.pack();
      frame.setVisible(true);
   }
}
```

- ✓ This code creates a simple JFrame with a JPanel that contains a JLabel, a JTextField for entering a name, a JSeparator, another JLabel, and a JTextField for entering an age
- ✓ The JSeparator is added to the panel using the add() method
- ✓ In this example, the JSeparator is created with the JSeparator(SwingConstants.VERTICAL) constructor, which creates a vertical separator
- ✓ The default constructor creates a horizontal separator
- ✓ You can also customize the appearance of the separator using various methods, such as setForeground() and setBackground()
- ✓ Overall, JSeparator is a useful component for visually separating different sections of a GUI, and it can be added to a JPanel or any other container using the add() method

Swing – Components for text editing

- Have as root the **JTextComponent** class from the **javax.swing.text** package



Swing - Components for text editing

- **JTextComponent** class
  - Keeps similarities with **TextComponent** class from AWT
  - Provides more complex features:
    - »Provides support for **undo** and **redo** operations
    - »Provides support for event handling generated by cursor, etc.
  - Any object derived from **JTextComponent** consists of:
    - »A model for managing the state of the component
      - A reference to the model is obtained with **getDocument** method, which returns an object of **Document** type
    - »A representation which is responsible for displaying the text
    - »A controller that allows
      - To writte and to read the text
      - To define necessary actions for editing

Swing – Components for text editing

- Depending on the type of text edititing the components for text editing are divided in:
  - Simple text on a single line
    - » **JTextField, JPasswordField, JFormattedTextField**
  - Simple text on multiple lines
    - » **JTextArea**
  - Text with enriched style on multiple lines
    - » **JEditorPane, JTextPane**

Swing – Components for text editing

- **JTextField** class
  - Allow to edit a simple text on a single line
- **JPasswordField** class
  - Allow to edit passwords
  - The text will be hidden; insides of entered characters is displayed a symbolic character such as “*”
- **JFormattedTextField** class
  - Allows to introduce text that follows a specific format
  - Is useful to read the numbers, date, etc.
  - Is used together with classes for formatting text
    - » e.g **DateFormatter, NumberFormatter, MaskFormatter**, etc.
  - The methods **getValue**, **setValue** are used to get/set the value contained in component

Swing - Components for text editing

- **JTextArea** class
  - Allows to edit a simple text on multiple lines
  - Any attribute of style (e.g., color, font) is applied to the entire text and cann't be specified olny certain part of the text
  - Usually, a component like this is included in a **JScrollPane** container to allow vertical navigation
- **JEditorPane** class
  - Allows to display/edit text written with multiple styles which include images or various other components
    - » Default are recognized these types of texts: text/plain, text/html and text/rtf
- **JTextPane** class
  - Extends **JEditorPane** by providing additional facilities for working with styles and paragraph

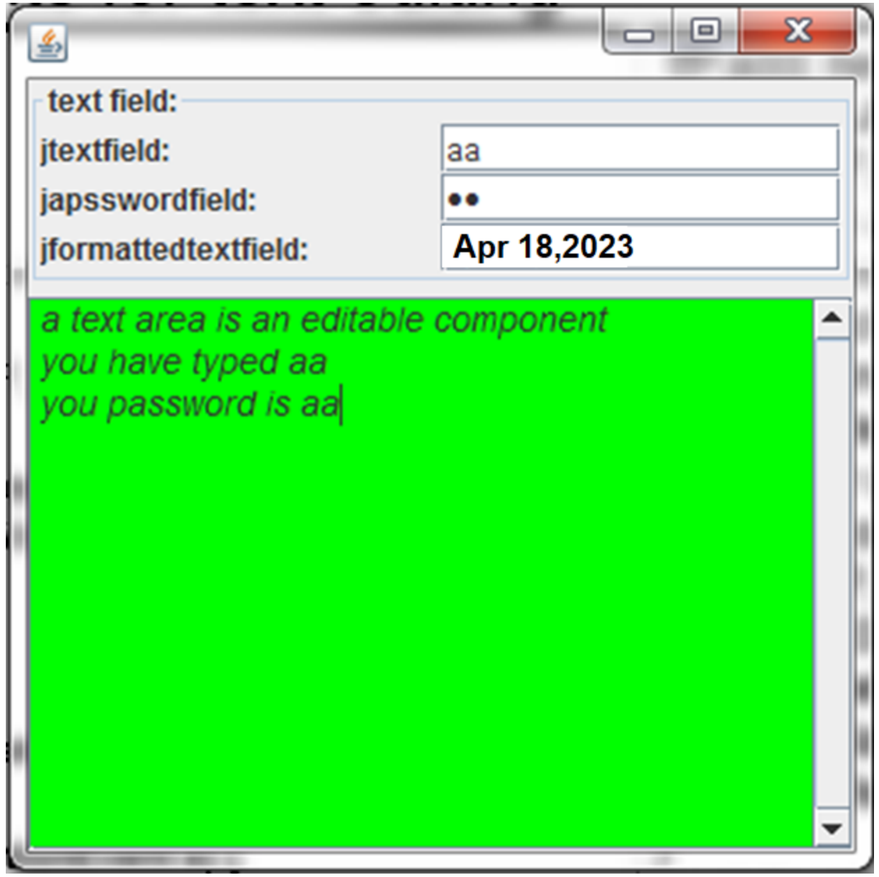
Swing - Components for text editing – Example (1)

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
public class JTextComponentDemo extends JFrame {
    JTextField tF;
   JPasswordField pwF;
   JTextArea taF;
   JFormattedTextField ftF;
   JTextComponentDemo(){
   JPanel tfP= new JPanel(new GridLayout(3,2));
   tfP.setBorder(BorderFactory.createTitledBorder("text field:"));
   tfP.add(new JLabel("jtextfield:"));
   tF= new JTextField(10);
   tfP.add(tF);
```

```
tF.addActionListener( new ActionListener(){
    @Override
     public void actionPerformed(ActionEvent e) {
       taF.append("\n you have typed "+tF.getText());
  }} );

  tfP.add( new JLabel("japsswordfield:"));
  pwF= new JPasswordField();
  tfP.add(pwF);
  pwF.addActionListener(new ActionListener() {
   @Override
   public void actionPerformed(ActionEvent e) {
     taF.append("\n you password is "+ new String
                                    (pwF.getPassword()));
  } });
 tfP.add(new JLabel("jformattedtextfield:"));
 ftF= new   JFormattedTextField
                              (Calendar.getInstance().getTime());
 tfP.add(ftF);
```

Swing - Components for text editing – Example (1)

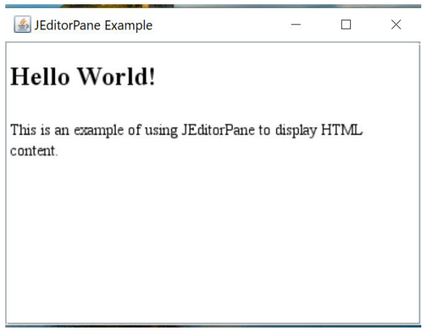
```
 taF= new JTextArea(" a text area is an editable component");
 taF.setFont(new Font("Arial", Font.ITALIC, 14));
 taF.setBackground(Color.green);
 JScrollPane taSP= new JScrollPane(taF);

taSP.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_AL
WAYS);
 Container cp= this.getContentPane();
  //Constructs a border layout with the specified gaps between components
 cp.setLayout(new BorderLayout(5,5));
 cp.add(tfP, BorderLayout.NORTH);
 cp.add(taSP, BorderLayout.CENTER);
 this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 this.setSize(350, 350);
 this.setVisible(true);
}
  public static void main(String args[]) {
     new JTextComponentDemo();
   }
```



Swing - Components for text editing (Example 2)

```
import javax.swing.*;
import java.awt.*;
public class EditorPaneExample extends JFrame {
    public EditorPaneExample() {
      super("JEditorPane Example");
      // Create a new JEditorPane and set its content type to HTML
      JEditorPane editorPane = new JEditorPane();
      editorPane.setContentType("text/html");
      // Set some initial content for the editor pane
       editorPane.setText("<html><body><h1>Hello World!</h1><p>This
              is an example of using JEditorPane to display HTML content.
              </p></body></html>");
      // Make the editor pane read-only
      editorPane.setEditable(false);
      // Add the editor pane to the frame
      getContentPane().add(new JScrollPane(editorPane),
                                        BorderLayout.CENTER);
      // Set the size and show the frame
      setSize(400, 300);
      setVisible(true); }
      public static void main(String[] args) {
              new EditorPaneExample();
   }}
```

- ✓ In this example, we create a new **JEditorPane** and set its content type to HTML using the **setContentType**() method
- ✓ We then set some initial content for the editor pane using the **setText**() method
- ✓ Finally, we add the editor pane to a **JScrollPane** and add that to the frame's content pane



Text editing components - Events handling

- Events generated by text editing components:
  - **ActionEvent**
    - » Is generated when press the **enter** key in the text edit box
    - » Needs to implement **ActionListener** interface
      - Contains **actionPerfomed** method which will be invoked when an action occurs
  - **CaretEvent**
    - » It generates when the cursor that manages current position in text is moved
    - » Needs to implement **CaretListener** interface
      - Contains **caretUpdate** method which will be called whenever there is a change

Text editing components - Events handling

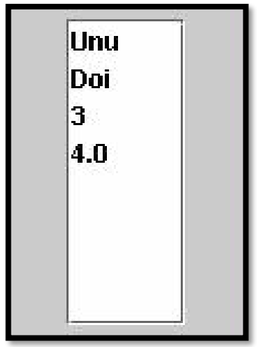
- Events generated by text editing components:
  - **DocumentEvent**
    - » Is generated when text is changed
    - » The source of events is the model of the component, no the component itself
    - » Needs to implement **DocumentListener** interface which contains the methods:
      - **insertUpdate** - is called when new characters are added
      - **removeUpdate** - is called after a delete operation
      - **changedUpdate** - is called when the attribute related to the style of the text are changed
  - **PropertyChangeEvent**
    - » Is generated to any change of a property of a component
    - » Needs to implement **PropertyChangeListener** which contains the method **propertyChange**

Components for selecting elements

- Includes classes that allow to select elements within a predetermined range:
  - **Jlist**
    - » Is a Swing component that displays a list of items from which the user can select one or more items
    - » The items can be displayed in a variety of formats, including text, icons, or a combination of both
  - **JComboBox**
    - » Is a Swing component that allows the user to select an item from a list of choices
    - » The selected item is displayed in the combo box when it is closed
    - » When the user clicks on the combo box, a drop-down list of choices appears, and the user can select an item from the list
  - **JSpinner**
    - » Is a Swing component that allows the user to select a value from a sequence of values, such as numbers or dates, using an up/down spinner control

Components for selecting elements - **JList** class

- Describes a list of items arranged in one or more columns
  - User can select one or more items from list
- A **JList** object can be included in a **JScrollPane** container
- Initializing a list can be made by:
  - Using the constructor which takes as argument a vector of items
  - Using a constructor with no arguments to create the default model and then, adding the items to the default model and then adding the model to the list
  - Using an own model, responsible for providing items to the list



Components for selecting elements - **JList** class

- Initializing the list using the constructor which takes as argument a vector of items

```
Object elemente[] = {"Unu", "Doi", new Integer(3), new Double(4)}
JList lista = new JList(elemente);
```

- Initializing the list using a constructor with no arguments to create the default model
  - Adding the items to the default model
  - Adding the default model to the list

```
DefaultListModel model = new DefaultListModel();
model.addElement( "one");
model.addElement("two");
model.addElement(new Integer(3));
model.addElement(new Double(4));
JList lista = new JList(model);
```

Components for selecting elements - **JList** class

- Initializing the list using an own model, responsible for providing items to the list
  - The own model is an object instance of a class which:
    - » Implements **ListModel** interface or
    - » Extends **AbstractListModel** class and overrides the methods:
      - **getElementAt**(**int** index) – provides the item from a certain position in the list
      - **getSize**() – returns the total number of items from the list

```
class Model extends AbstractListModel{
    Object elements[] = {“one", “two", new Integer(3), new Double(4)};
     public int getSize()
         {return elemente.length; }
     public Object getElementAt(int index)
          {return elemente[index]; }
}
Model model = new Model();
JList lista = new JList(model);
...
```

Components for selecting elements - **JList** class

- Managing the selected items from a list:
  - Is made by a model, which is an instance of **ListSelectionModel**
- **JList** objects generate **ListSelectionEvent** events
- **ListSelectionListener**
  - Interface for **ListSelectionEvent** events
  - Contains the method **valueChanged** which is called when we change selection of items from list

Components for selecting elements - **JList** class

- Example of **ListSelectionEvent** handling

```
class Test implements ListSelectionListener {
    ...
    public Test() {...
    // Stabilim modul de selectie
    list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    //Adaugam un ascultator
    ListSelectionModel model = list.getSelectionModel();
    model.addListSelectionListener(this);
      ... }
    public void valueChanged(ListSelectionEvent e) {
        int index = list.getSelectedIndex();
        ... }}
```

Components for selecting elements - **JList** class

- **JList** provides methods for:
  - Selecting the items from the list
    - » **public void setSelectedIndex (int i)**
      - Selects a single cell
      - Does nothing if the given index is greater than or equal to the model size
    - » **public void setSelectedIndices(int[] indices)**
      - Changes the selection to be the set of indices specified by the given array
  - Getting selected items:
    - » **getSelectedIndex**
      - Returns the selected index when only a single item is selected in the list
      - When multiple items are selected, returns the smallest selected index
      - Returns -1 if there is no selection
    - » **getSelectedIndices**
      - Returns an array of all of the selected indices, in increasing order

Components for selecting elements - **JList** class

- JList allows setting a **renderer** foreach item (**Renderer** is a customized representation of the item of the list based on various parameters)
- To customize the rendering of each item in a **JList**, you can create a class that implements the **ListCellRenderer** interface

Components for selecting elements - **JList** class

- **ListCellRenderer** interface has a single method called getListCellRendererComponent(), which you can implement to create a customized rendering of each item in the list

Component getListCellRendererComponent(**JList**<? **extends** E> list, E value, **int** index, **boolean** isSelected, **boolean** cellHasFocus)

»Return a component that has been configured to display the specified value

»Parameters:

- *list* - The JList we're painting
- *value* - The value returned by list.getModel().getElementAt(index).
- *index* - The cell index
- *isSelected* - True if the specified cell was selected
- *cellHasFocus* - True if the specified cell has the focus

Components for selecting elements - **JList** class

- Example of setting a **renderer** for a list

```
class MyCellRenderer extends JLabel implements ListCellRenderer {

  public MyCellRenderer()
     {setOpaque(true);}

  public Component getListCellRendererComponent(JList list, Object value, int index, boolean isSelected, boolean
                 cellHasFocus)
     {
       setText(value.toString());
       setBackground(isSelected?Color.red:Color.white);
       setForeground(isSelected?Color.white: Color.black);
       return this;
    }
  }
          ……
          list.setCellRenderer(new MyCellRenderer());
```

Components for selecting elements - **JList** class - Example of setting a **renderer** for a list

- Step 1: Developing an entity - It is important to understand that a JList can contain any list of Java objects

```
public class Country {
    private String name;
    private String code;
    public Country(String name, String code)
     {
        this.name = name;
        this.code = code;
      }
     public String getName()
       {  return name; }

     public void setName(String name)
       { this.name = name; }

     public String getCode()
      { return code; }

    public void setCode(String code)
    { this.code = code;}
}
```

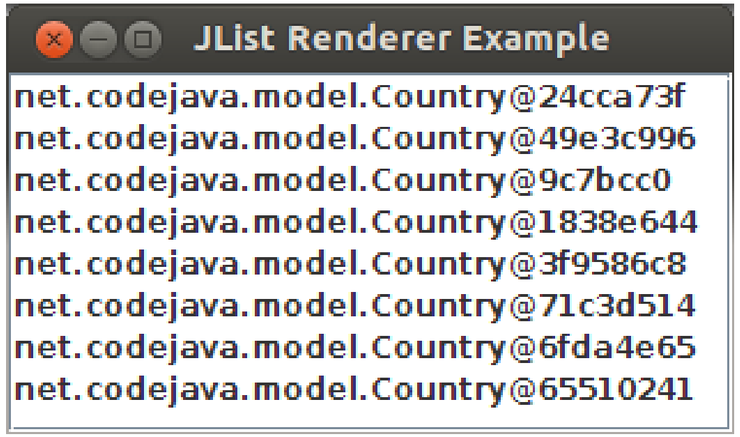
Components for selecting elements - **JList** class
Example of setting a **renderer** for a list

- Step 2: Creating the List with Entity Objects
  - We first create the instances of the **Country** class
  - We then create the **DefaultListModel** instance
    - » We parameterize this instance with the Country class, as we are going to add instances of Country to our **JList**
  - We then create the **JList** instance, we pass the model
    - » The model already has all the data and so, it would be displayed

```
import javax.swing. *;
public class JListCustomRendererExample extends JFrame {
    public JListCustomRendererExample() {
      Country us = new Country("USA", "us");
      Country in = new Country("India", "in");
      Country vn = new Country("Vietnam", "vn");
      Country ca = new Country("Canada", "ca");
      DefaultListModel<Country> listModel = new
                                                DefaultListModel<Country>();
      listModel.addElement(us);
      listModel.addElement(in);
      listModel.addElement(vn);
      listModel.addElement(ca);
     JList<Country> countryList = new JList<Country>(listModel);
     add(new JScrollPane(countryList));
     this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     this.setTitle("JList Renderer Example");
     this.setSize(200, 200);
     this.setVisible(true);    }
    public static void main(String[] args)
       { new JListCustomRendererExample();}
 }
```

Components for selecting elements - **JList** class- Example of setting a **renderer** for a list

- As we know
  - The data of a **JList** is handled by a **ListModel**
  - The display of objects in a **JList** is handled by an object called **Renderer**
- For a **JList**, a renderer is provided by default
  - How does this renderer work?
    - » The renderer is responsible for displaying all the elements in the list
    - » The renderer calls the **toString**() method on the object and uses that string value in the display
      - As we did not override **toString**() method in our instance class **Country**, the Object's toString() is invoked and that is what is displayed

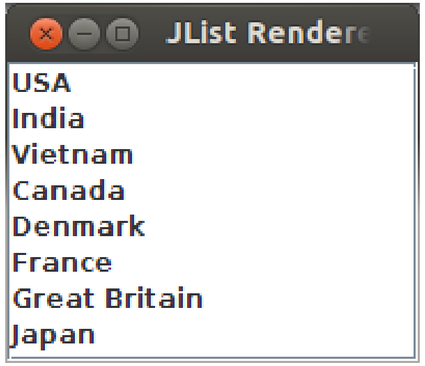


Initial Output

Components for selecting elements - **JList** class - Example of setting a **renderer** for a list

- Step 3: Implementing toString() in Country class
  - » We would like to display the name of the country in the list:

```
private String name;
   private String code;
   public Country(String name, String code)
    {
       this.name = name;
       this.code = code;
     }
    public String getName()
      {  return name; }
    public void setName(String name)
      { this.name = name; }
    public String getCode()
     { return code; }
   public void setCode(String code)
      { this.code = code;}
   @Override
   public String toString()
      { return name;}}
```



Output with toString() Implementation

Components for selecting elements - **JList** class - Example of setting a renderer for a list

- Developing a Custom Renderer
  - If we would like to display the country's flag alongside the name of the country, we will have to write our own custom renderer
  - To develop a custom renderer, we should write a class that implements **ListCellRenderer** interface
    - » **ListCellRenderer** has the method **getListCellRendererComponent**()
      - This method expects a component to be returned back
        - The component should handle the display of the item
  - In our case we define a class that extend **JLabel** as a **JLabel** can display both icon and text and implements **ListCellRenderer** interface

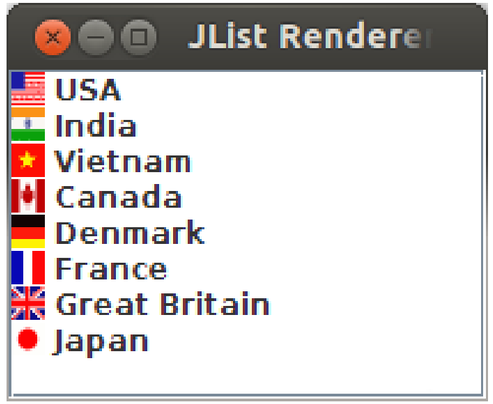
Components for selecting elements - **JList** class -Example of setting a renderer for a list

- Developing a Custom Renderer

```
import java.awt.Component;
import javax.swing.*;
public class CountryRenderer extends JLabel implements ListCellRenderer<Country>
 {
   public CountryRenderer() {setOpaque(true);}
          @Override
   public Component getListCellRendererComponent(JList < Country>  list, Country
            country, int index, boolean isSelected, boolean cellHasFocus)
     {
          String code = country.getCode();
          ImageIcon imageIcon = new ImageIcon("D:/images/" + code + ".png");
          setIcon(imageIcon);
          setText(country.getName());
          if (isSelected) {  setBackground(list.getSelectionBackground());
                             setForeground(list.getSelectionForeground());  }
          else {  setBackground(list.getBackground());
                  setForeground(list.getForeground()); }
          return this;
   }
 }
```

Components for selecting elements - **JList** class
Example of setting a renderer for a list

- Developing a Custom Renderer (cont')
  - In the previous example the Country instance is passed to the getListCellRendererComponent() method
  - Also, the JList instance itself is passed to the method
- To set the renderer by setting to the list we put the following line of code in the JListCustomRendererExample class:countryList.setCellRenderer(new CountryRenderer());



Output with custom renderer

```
import javax.swing. *;
public class JListCustomRendererExample extends JFrame {
    public JListCustomRendererExample() {
      Country us = new Country("USA", "us");
      Country in = new Country("India", "in");
      Country vn = new Country("Vietnam", "vn");
      Country ca = new Country("Canada", "ca");
      DefaultListModel<Country> listModel = new
                                                DefaultListModel<Country>();
      listModel.addElement(us);
      listModel.addElement(in);
      listModel.addElement(vn);
      listModel.addElement(ca);
     JList<Country> countryList = new JList<Country>(listModel);
     add(new JScrollPane(countryList));
     this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     this.setTitle("JList Renderer Example");
     this.setSize(200, 200);
     this.setVisible(true);    }
    public static void main(String[] args)
      {JListCustomRendererExample countryList= new
                                          JListCustomRendererExample();
      countryList.setCellRenderer(new CountryRenderer());  } }
```