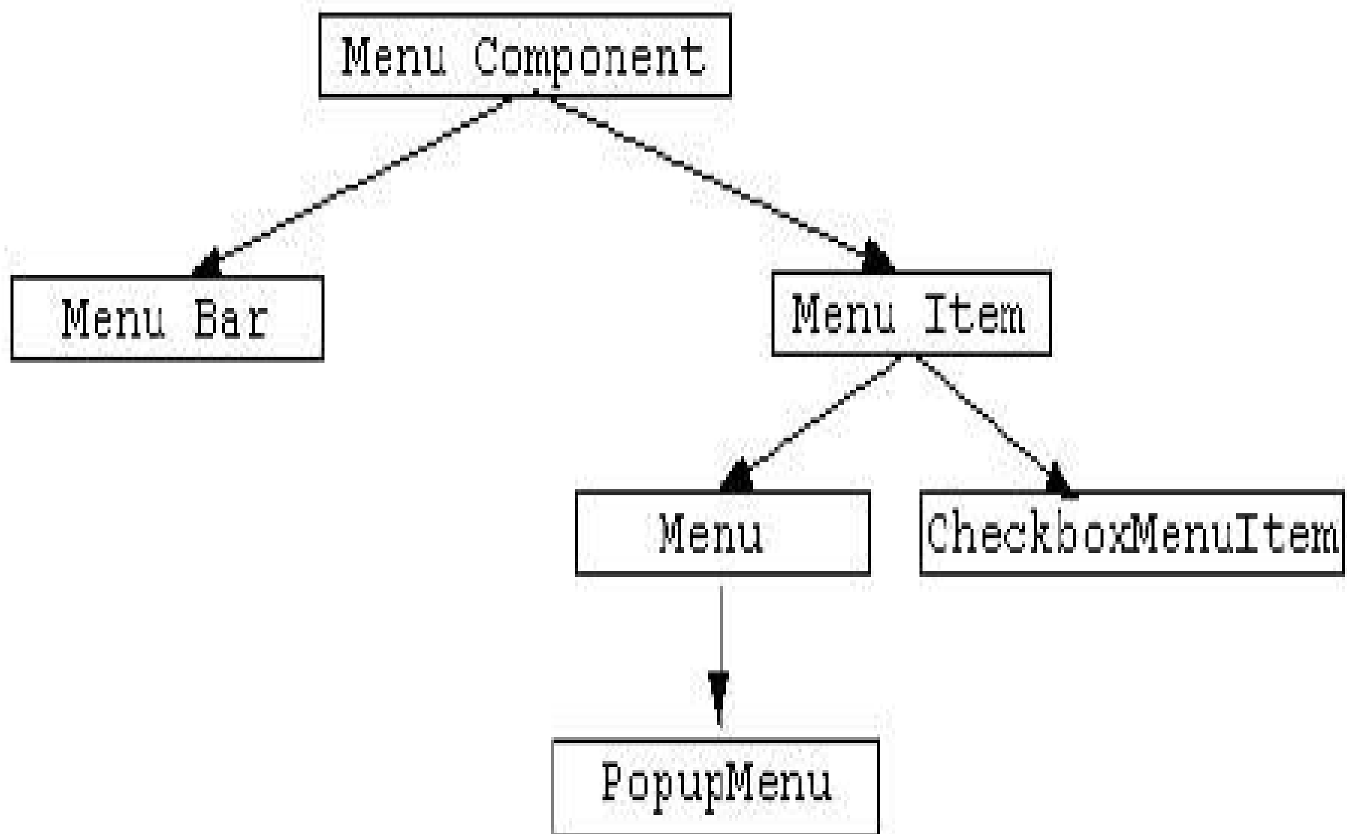
Interfața grafică cu utilizatorul - Swing

Programare Orientată pe Obiecte

Ierarhia claselor ce descriu meniuri



Meniuri de context (popup)

```
PopupMenu popup = new PopupMenu("Options");
popup.add(new MenuItem("New"));
popup.add(new MenuItem("Edit"));
popup.addSeparator();
popup.add(new MenuItem("Exit"));
...
popup.show(Component origine, int x, int y)
fereastra.add(popup1);
...
fereastra.remove(popup1);
fereastra.add(popup2);
```

isPopupTrigger() :

- dacă acest eveniment de mouse este evenimentul care poate afişa un meniu popup
- trebuie testat în ambele metode mousePressed şi mouseReleased pentru că depinde de platformă

Folosirea unui meniu de context (popup)

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

class Fereastra extends Frame implements ActionListener{
  // Definim meniul popup al ferestrei
  private PopupMenu popup;
  // Pozitia meniului va fi relativa la fereastra
  private Component origin;
  public Fereastra(String titlu) {
    super(titlu);
    origin = this;

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

    this.addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e) {
        if (e.isPopupTrigger())
          popup.show(origin, e.getX(), e.getY());
      }
      public void mouseReleased(MouseEvent e) {
        if (e.isPopupTrigger())
          popup.show(origin, e.getX(), e.getY());
      }
    });
    setSize(300, 300);
```

Folosirea unui meniu de context (popup) (2)

```
    // Cream meniul popup
    popup = new PopupMenu("Options");
    popup.add(new MenuItem("New"));
    popup.add(new MenuItem("Edit"));
    popup.addSeparator();
    popup.add(new MenuItem("Exit"));
    add(popup); //atasam meniul popup ferestrei
    popup.addActionListener(this);
  }

  public void actionPerformed(ActionEvent e) {
    String command = e.getActionCommand();
    if (command.equals("Exit"))
         System.exit(0);
   }
 }

 public class TestPopupMenu {
   public static void main(String args[]) {
     Fereastra f = new Fereastra("PopupMenu");
     f.show();
   }
 }
```

Arhitectura modelului Swing

- MVC (model-view-controller):
  - Modelul - datele aplicației.
  - Prezentarea - reprezentare vizuală
  - Controlul - transformarea acțiunilor în evenimente
- Arhitectură cu model separabil: Model + (Prezentare, Control)
- Fiecărui obiect corespunzător unei clase ce descrie o componentă Swing îi este asociat un obiect care gestionează datele sale şi care implementează o interfață care reprezintă modelul componentei respective.
- Fiecare componentă are un model inițial implicit, însă are posibilitatea de a-l înlocui cu unul nou atunci când este cazul. Metodele care accesează modelul unui obiect sunt: setModel, respectiv getModel, cu argumente specifice fiecărei componente în parte.
- Crearea unui model = implementarea interfeței

JList - ListModel

DefaultListModel, AbstractListModel

Folosirea modelelor

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

Folosirea modelelor - exemplu

```
import javax . swing .*;
import javax . swing . border .*;
import java . awt .*;
import java . awt. event .*;
class Fereastra extends JFrame implements ActionListener {
    String data1 [] = {" rosu ", " galben ", " albastru "};
    String data2 [] = {"red", " yellow ", " blue "};
    int tipModel = 1;
    JList lst;
    ListModel model1 , model2 ;
    public Fereastra ( String titlu ) {
        super ( titlu );
        setDefaultCloseOperation ( JFrame . EXIT_ON_CLOSE );
        // Lista initiala nu are nici un model
        lst = new JList ();
        add(lst , BorderLayout . CENTER );
        // La apasara butonului schimbam modelul
        JButton btn = new JButton (" Schimba modelul ");
        add(btn , BorderLayout . SOUTH );
        btn . addActionListener ( this );
        // Cream obiectele corespunzatoare celor doua modele
        model1 = new Model1 ();
        model2 = new Model2 ();
        lst . setModel ( model1 );
        pack ();
    }
```

Folosirea modelelor - exemplu

```
public void actionPerformed ( ActionEvent e) {
            if ( tipModel == 1) {
                        lst . setModel ( model2 );    tipModel = 2;}
            else {
                        lst . setModel ( model1 ); tipModel = 1;}
      }
// Clasele corespunzatoare celor doua modele
class Model1 extends AbstractListModel {
      public int getSize () {
            return data1 . length ;
      }
      public Object getElementAt ( int index ) {
            return data1 [ index ];
      }
}
class Model2 extends AbstractListModel {
      public int getSize () {
            return data2 . length ;
      }
      public Object getElementAt ( int index ) {
            return data2 [ index ];
      }
 }
}
public class TestModel {
      public static void main ( String args []) {
            new Fereastra (" Test Model "). show ();
      }
}
```

Observații

- Multe componente Swing furnizează metode care să obțină starea obiectului fără a mai fi nevoie să obținem instanța modelului şi să apelăm metodele acesteia. Un exemplu este metoda getValue a clasei JSlider care este de fapt un apel de genul:

  getModel().getValue().
- In multe situații însă, mai ales pentru clase cum ar fi JTable sau JTree, folosirea modelelor aduce flexibilitate sporită programului şi este recomandată utilizarea lor.

Tratarea evenimentelor: informativ (lightweight)

- Modelele trimit un eveniment prin care sunt informați ascultătorii că a survenit o anumită schimbare a datelor, fără a include în eveniment detalii legate de schimbarea survenită. Obiectele de tip listener vor trebui să apeleze metode specifice componentelor pentru a afla ce anume s-a schimbat.
- ChangeListener - ChangeEvent

  Modele: BoundedRangeModel, ButtonModel şi SingleSelectionModel

```
JSlider slider = new JSlider();
BoundedRangeModel model = slider.getModel();
model.addChangeListener(new ChangeListener() {
  public void stateChanged(ChangeEvent e) {
      // Sursa este de tip BoundedRangeModel
      BoundedRangeModel m =(BoundedRangeModel)
                                           e.getSource();
      // Trebuie sa interogam sursa asupra schimbarii
      System.out.println("Schimbare model: " + m.getValue());
  }
});
```

Tratarea evenimentelor: informativ (lightweight) (2)

- Pentru uşurința programării, pentru a nu lucra direct cu instanța modelului, unele clase permit înregistrarea ascultătorilor direct pentru componenta în sine, singura diferență față de varianta anterioară constând în faptul că sursa evenimentului este acum de tipul componentei şi nu de tipul modelului.

```
JSlider slider = new JSlider();
slider.addChangeListener(new ChangeListener() {
    public void stateChanged(ChangeEvent e) {
        // Sursa este de tip JSlider
        JSlider s = (JSlider)e.getSource();
        System.out.println("Valoare noua: " +
                                        s.getValue());
    }
});
```

Tratarea evenimentelor

2. Consistent(statefull): Modele pun la dispoziție interfețe specializate şi tipuri de evenimente specifice ce includ toate informațiile legate de schimbarea datelor.

Model	Tip Eveniment
ListModel	ListDataEvent
ListSelectionModel	ListSelectionEvent
ComboBoxModel	ListDataEvent
TreeModel	TreeModelEvent
TreeSelectionModel	TreeSelectionEvent
TableModel	TableModelEvent
TableColumnModel	TableColumnModelEvent
Document	DocumentEvent
Document	UndoableEditEvent

Tratarea evenimentelor

```
String culori[] = {"rosu", "galben", "albastru");
JList list = new JList(culori);
ListSelectionModel sModel = list.getSelectionModel();
sModel.addListSelectionListener(new
    ListSelectionListener() {
    public void valueChanged (ListSelectionEvent e) {
        // Schimbarea este continuta in eveniment
         if (!e.getValueIsAdjusting()) {
               System.out.println("Selectie curenta: " +
                                    e.getFirstIndex());
    }
 }
});
```

Sau:

```
JList list = new JList(culori);
list.addListSelectionListener(new ListSelectionListener() {
    public void valueChanged (ListSelectionEvent e) {
    …
});
```

Componente pentru selectare
Clasa JList

- Object elemente[] = {"Unu", new Integer(2)};

  JList lista = new JList(elemente);
- Vector elemente = new Vector();

  elemente.add(“Unu”); elemente.add(new Integer(2));

  JList lista = new JList(elemente);

Obs: modificarea elementelor vectorului pe baza căruia a fost creata inițial lista nu duce la modificarea elementelor listei!! Pentru actualizarea elementelor listei se va apela funcția **setListData(vector)** din clasa JList.

- DefaultListModel model = new DefaultListModel();

model.addElement("Unu");

model.addElement(new Integer(2));

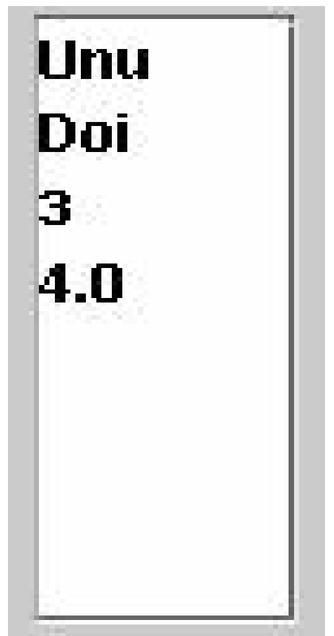
JList lista = new JList(model);

Obs: modificarea elementelor modelului pe baza căruia a fost creata inițial lista se reflectă automat în listă!!

Clasa JList

- ModelLista model = new ModelLista();

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



Tratarea evenimentelor

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

Obiecte de tip Renderer

- Un renderer este responsabil cu afişarea articolelor unei componente.

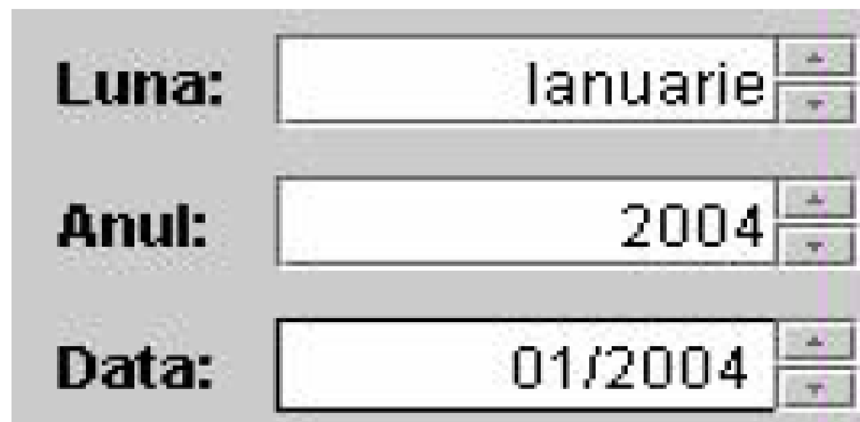
```
class MyCellRenderer extends JLabel implements
    ListCellRenderer {
    public MyCellRenderer() {
            setOpaque(true);
    }
    public Component getListCellRendererComponent(JList list,
            Object value, int index, boolean isSelected, boolean
            cellHasFocus) {
      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

Clasa JComboBox

- similară cu JList, cu deosebirea că permite doar selectarea unui singur articol, acesta fiind şi singurul permanent vizibil.
- Inițializarea se face folosind fie un vector fie un model de tipul ComboBoxModel
- fiecare element poate fi de asemenea reprezentat diferit prin intermediul unui obiect ce implementează aceeaşi intefață ca şi în cazul listelor:ListCellRenderer.
- JComboBox permite şi editarea explicită a valorii elementului, acest lucru fiind controlat de metoda setEditable.
- Evenimentele generate de obiectele JComboBox sunt de tip ItemEvent generate la navigarea prin listă, respectiv ActionEvent generate la selectarea efectivă a unui articol.

Clasa JSpinner

• oferă posibilitatea de a selecta o anumită valoare (element) dintr-un domeniu prestabilit, lista elementelor nefiind însă vizibilă. Este folosit atunci când domeniul din care poate fi făcută selecția este foarte mare sau chiar nemărginit; de exemplu: numere intregi intre 1950 si 2050.
• se bazează exclusiv pe folosirea unui model. Acesta este un obiect de tip SpinnerModel (SpinnerListModel, SpinnerNumberModel sau SpinnerDateModel)
• permit şi specificarea unui anumit tip de editor pentru valorile elementelor sale. Acesta este instalat automat pentru fiecare din modelele standard amintite mai sus
• evenimentele generate de obiectele de tip JSpinner sunt de tip ChangeEvent, generate la schimbarea stării componentei.



Tabele

Inițializarea

```
String[] coloane = {"Nume", "Varsta", "Student"};
Object[][] elemente = {
        {"Ionescu", new Integer(20), Boolean.TRUE},
        {"Popescu", new Integer(80), Boolean.FALSE}};
JTable tabel = new JTable(elemente, coloane);
```

Nume	Varsta	Student
Ionescu	20	true
Popescu	80	false

Folosirea unui model

```
ModelTabel model = new ModelTabel();
JTable tabel = new JTable(model);
...
class ModelTabel extends AbstractTableModel {
	String[] coloane = {"Nume", "Varsta", "Student"};
	Object[][] elemente = {
		{"Ionescu", new Integer(20), Boolean.TRUE},
		{"Popescu", new Integer(80), Boolean.FALSE}};
	public int getColumnCount() {return coloane.length;}
	public int getRowCount() {return elemente.length;}
	public Object getValueAt(int row, int col) {
		return elemente[row][col];
	}
	public String getColumnName(int col) {
		return coloane[col];
	}
	public boolean isCellEditable(int row, int col) {
		// Doar numele este editabil
		return (col == 0);
	}
}
```

Tratarea evenimentelor

1. Generate de editarea celulelor

```
public class Listener implements TableModelListener {
    public void tableChanged(TableModelEvent e) {
    // Aflam celula care a fost modificata
    int row = e.getFirstRow();
    int col = e.getColumn();
    TableModel model = (TableModel)e.getSource();
    Object data = model.getValueAt(row, col);
    ...
    }
}
...
tabel.getModel().addTableModelListener(new
    Listener());
```

Tratarea evenimentelor

2. Generate de selectarea liniilor

```
class Listener implements ListSelectionListener {
    public void valueChanged(ListSelectionEvent e) {
            if (e.getValueIsAdjusting()) return;
            ListSelectionModel model =
                        (ListSelectionModel)e.getSource();
            int index = model.getMinSelectionIndex();
            // Linia cu numarul index este prima selectata
            ...
            }
    }
}
...
tabel.setSelectionMode(ListSelectionModel.SINGLE_SE
    LECTION);
ListSelectionModel model = tabel.getSelectionModel();
model.addListSelectionListener(new Listener());
```

Tratarea evenimentelor

3. Generate de selectarea coloanelor

```
class colSL implements ListSelectionListener {
    public void valueChanged(ListSelectionEvent e) {
            if (e.getValueIsAdjusting()) return;
            ListSelectionModel model =
                        (ListSelectionModel)e.getSource();
            int index = model.getMinSelectionIndex();
            // coloana cu numarul index este prima selectata
            ...
            }
    }
}
...
ListSelectionModel colSM=
    tab.getColumnModel().getSelectionModel();
colSM.addListSelectionListener(new colSL());
```

Folosirea unui renderer

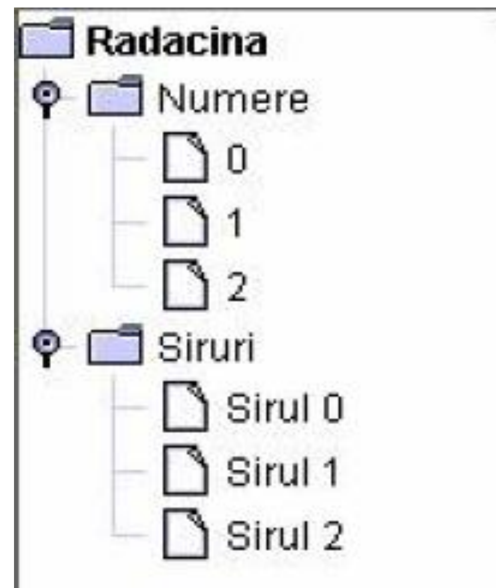
```
public class MyRenderer extends JLabel
    implements TableCellRenderer {
      public Component
          getTableCellRendererComponent(..) {
                    ...
                    return this;
      }
}
```

Folosirea unui editor

```
public class MyEditor extends AbstractCellEditor
    implements TableCellEditor {
    public Object getCellEditorValue() {
            // Returneaza valoarea editata
            ...
    }
public Component getTableCellEditorComponent(...) {
            // Returneaza componenta de tip editor
            ...
    }
}
```

Arbori

```
String text = "Radacina";
DefaultMutableTreeNode root = new DefaultMutableTreeNode(text);
DefaultMutableTreeNode numere = new DefaultMutableTreeNode
                                                    ("Numere");
DefaultMutableTreeNode siruri = new DefaultMutableTreeNode
                                                    ("Siruri");
for(int i=0; i<3; i++) {
        numere.add(new DefaultMutableTreeNode(new Integer(i)));
        siruri.add(new DefaultMutableTreeNode("Sirul " + i));
}
root.add(numere);
root.add(siruri);
JTree tree = new JTree(root);
```



Tratarea evenimentelor

```
class Listener implements TreeSelectionListener {
  public void valueChanged(TreeSelectionEvent e) {
    // Obtinem nodul selectat
    DefaultMutableTreeNode node =
    (DefaultMutableTreeNode)
    tree.getLastSelectedPathComponent();
    if (node == null) return;
    // Obtinem informatia din nod
    Object nodeInfo = node.getUserObject();
    ...
  }
}
...
// Stabilim modul de selectie
tree.getSelectionModel().setSelectionMode(
TreeSelectionModel.SINGLE_TREE_SELECTION);
// Adaugam un ascultator
tree.addTreeSelectionListener(new Listener());
```

Personalizarea nodurilor

TreeCellRenderer

- setRootVisible
- setShowsRootHandles: dacă nodurile de pe primul nivel au simboluri care să permită expandarea sau restrângerea lor.
- putClientProperty: stabileşte diverse proprietăți, cum ar fi modul de reprezentare a relațiilor (liniilor) dintre nodurile părinte şi fiu:

```
tree.putClientProperty("JTree.lineStyle", "Angled");
// sau "Horizontal", "None"
```

- Specificarea unei iconițe

```
ImageIcon leaf = createImageIcon("img/leaf.gif");
ImageIcon open = createImageIcon("img/open.gif");
ImageIcon closed = createImageIcon("img/closed.gif");
DefaultTreeCellRenderer renderer =new
        DefaultTreeCellRenderer();
renderer.setLeafIcon(leaf);
renderer.setOpenIcon(open);
renderer.setClosedIcon(closed);
tree.setCellRenderer(renderer);
```