Basics of the Object-Oriented Programming

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

JComboBox class

- Similar with **JList**
- Allows to select only one item that will be visible
- List of other elements is displayed only by pressing the button marked with an arrow
- Component initialization can be done using:
  - A vector
  - A model of **ComboBoxModel** type
- Each item can be represented differently through an object instance of a class that implements **ListCellRenderer** interface

JComboBox class

- Allows to edit explicitly the value of the item through the method **setEditable**
- The events that are generated are of type:
  - **ItemEvent**
    - » Is generated when we navigate in the list
  - **ActionEvent**
    - » Is generated when we select an item from the list

JComboBox class - Example

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class ComboBoxExample implements ActionListener {
    private JComboBox<String> comboBox;
    private JLabel label;
    public ComboBoxExample() {
        String[] options = {"Option 1", "Option 2", "Option 3", "Option 4"};
        // Create the combo box
        comboBox = new JComboBox<>(options);
        comboBox.addActionListener(this);
        // Create the label
        label = new JLabel("Please select an option");
        // Create the frame and add the components
        JFrame frame = new JFrame("Combo Box Example");
        frame.setLayout(new FlowLayout());
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 100);
        frame.add(comboBox);
        frame.add(label);
        frame.setVisible(true); }
```

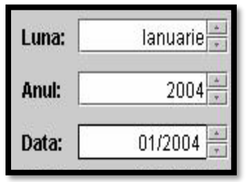
```
@Override
    public void actionPerformed(ActionEvent e) {
        // Get the selected item from the combo box
        String selectedOption = (String)
                                comboBox.getSelectedItem();

        // Update the label with the selected option
        label.setText("You selected " + selectedOption);
    }

    public static void main(String[] args) {
        ComboBoxExample example = new
                                        ComboBoxExample();
    }
}
```

JSpinner class

- Allows to select a specific value in predefined range
- The item of the list are not visible
- Is used when the domain from which the selection is made is very large
  - e.g., integers between 1950 and 2050
- The component contains two buttons with which we can select the next item or the previous item in list
- Uses a model which is an object of **SpinnerModel** type (SpinnerModel is an interface)
- There are a lot of predefined classes that implements this interface
  - e.g.: **SpinnerListModel, SpinnerNumberModel, SpinnerDateModel**
- The events are of **ChangeEvent** type and are generated when the state of the component is changed



JTable class

- Shows the items in a tabular form
- Can be used for:
  - Formatted display of data
  - Editing information in its cells

- Rows of the tables can be marked as selected; the selection type can be simple or multipe
- Table initialization can be made by:
  - Using one of the constructors having as arguments:
    - The table elements represented as a matrix or a collection of **Vector** type
    - The names of the columns
  - Implementing the table model in a separate class and using the corresponding constructor

JTable class - Exemple of initializing a table using constructor

- Initializing a table using a constructor having as arguments:
  - The table elements represented as a matrix or a collection of **Vector** type
    - » Data type of the elements of a column can be anything
  - The names of the columns

```
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);
```

JTable class - Exemple of initializing a table using constructor

```
import javax.swing.*;
public class TableExample {
    JFrame f;
    TableExample(){
      f=new JFrame();
      String data[][]={ {"101","Amit","670000"},  {"102","Jai","780000"},  {"101","Sachin","700000"}};
      String column[]={"ID","NAME","SALARY"};
      JTable jt=new JTable(data, column);
      jt.setBounds(30,40,200,300);
      JScrollPane sp=new JScrollPane(jt);
      f.add(sp);
      f.setSize(300,400);
      f.setVisible(true);
}
public static void main(String[] args) {
    new TableExample();
    }    }
```

JTable class - Exemple of initializing a table by implementing the table model

- Initializing the table by implementing the table model in a separate class and using the corresponding constructor
  - For implementing the model:
    - » Implements **TableModel** interface
      - Contains methods that are queried to obtain information from the table
    - » Or extends **AbstractTableModel** class
      - Override the specific methods; the most used methods are:
        - **getRowCount**() - returns the number of rows in the table
        - **getColumnCount**() - returns the number of columns in the table
        - **getValueAt** (**int** row, **int** column) - return the element to a specific row and column
        - **getColumnName(int** column) - returns the name of the column
        - **isCellEditable (int** rowIndex, **int** columnIndex) - specify if a cell is editable or not

JTable class - Exemple of initializing a table by implementing the table model

```
class ModelTabel extends AbstractTableModel{
   String[] coloane = {"Name", “Age", "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)
      {    //only the name is editable
            return (col == 0);
       }
    }
.....
 ModelTabel model = new ModelTabel();
JTable tabel = new JTable(model);
 .....
```

JTable class - Handling the events generated by a change of the date in the table

- Changing the date in a table will generate an **TableModelEvent** event
- **TableModelListener** interface
  - Handle events of type **TableModelEvent**
  - The interface consists of one method that tells you when the table data changes

```
public interface TableModelListener extends EventListener {
  public void tableChanged(TableModelEvent e);
}
```

- Register a listener will be done for the table model

JTable class - Exemple of handling the events generated by the date changing in the table

```
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.table.*;
public class JTableExample extends JFrame {
    private JTable table;
    public JTableExample() {
        // create table data and column headers
        Object[][] data = { { "John", 28 },
            { "Jane", 32 },
            { "Joe", 25 },
            { "Julie", 30 }    };
        String[] columnNames = { "Name", "Age" };
        // create the table using the data and column headers
        table = new JTable(data, columnNames);}
```

```
 // add a table model listener to handle cell changes
        table.getModel().addTableModelListener(new TableModelListener() {
            public void tableChanged(TableModelEvent e) {
                // handle the cell change event here
                int row = e.getFirstRow();
                int column = e.getColumn();
                TableModel model = (TableModel)e.getSource();
                Object data = model.getValueAt(row, column);
                System.out.println("Cell updated at row " + row + " and column " + column + ": " + data);
            }
        });
     JScrollPane scrollPane = new JScrollPane(table);
        getContentPane().add(scrollPane);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setTitle("JTable Example");
        pack();
        setVisible(true);    }
    public static void main(String[] args) {
        new JTableExample(); }}
```

JTable class - Exemple of handling the events generated by the date changing in the table

- In the previous example, we create a JTable component with some sample data and column headers.
- We then add a table model listener to the table, which listens for changes to the table's cells.
- When a cell is changed, the tableChanged method is called, and we can handle the event there. In this example, we simply print the updated cell's value to the console.
- Note that we also add the table to a scroll pane before adding it to the frame

JTable class - Handing the events generated by changing the selection in the table

- We can select one or more than one rows in the table
- Management of selected lines is achieved through a model
  - The model is an instance of a class that implements **ListSelectionModel** interface
- Handing the events generated by changing the selection in the table is made by registering a listener of type **ListSelectionListener**

JTable class - Handing the events generated by changing the selection in the table (Example)

```
class Test implements ListSelectionListener { ...
  public Test() { ...

  // Determine the selection mode
  tabel.setSelectionModel(ListSelectionModel.SINGLE_SELECTION);

 // adding a listener
  ListSelectionModel model = tabel.getSelectionModel();
  model.addListSelectionListener(this);
   ...  }
  public void valueChanged(ListSelectionEvent e) {
     ListSelectionModel model = (ListSelectionModel)e.getSource();
     if (model.isSelectionEmpty()) {
          // No line is selected
       ... }
    else {int index = model.getMinSelectionIndex();
           // The line with the index number is the first selected
        ...      }} }
```

JTable class-Handing the events generated by changing the selection in the table

```
import javax.swing.*;
import javax.swing.event.*;
public class TableExample {
 public static void main(String[] a) {
  JFrame f = new JFrame("Table Example");
  String data[][]={ {"101","Amit","670000"},
   {"102","Jai","780000"}, {"101","Sachin","700000"}};
  String column[]={"ID","NAME","SALARY"};
  JTable jt=new JTable(data,column);
  jt.setCellSelectionEnabled(true);
  ListSelectionModel select= jt.getSelectionModel();
  select.setSelectionMode(ListSelectionModel.SINGLE_SELECTIO
N);
```

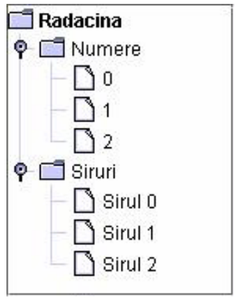
```
select.addListSelectionListener(
   new ListSelectionListener() {
       public void valueChanged(ListSelectionEvent e)
        {   String Data = null;
            int[] row = jt.getSelectedRows();
            int[] columns = jt.getSelectedColumns();
            for (int i = 0; i < row.length; i++) {
              for (int j = 0; j < columns.length; j++) {
                Data = (String) jt.getValueAt(row[i], columns[j]);
                }
           }
            System.out.println("Table element selected is: " +
                                        Data);
           } });
          JScrollPane sp=new JScrollPane(jt);
          f.add(sp);
          f.setSize(300, 200);
          f.setVisible(true);
        }       }
```

JTable class

- The cell of a column are represented in the same way
- Each column has associated a renderer object which is responsible for creating the component describing its cells
- A render object implements **TableCellRenderer** interface
  - **setDefaultRenderer** method is used to specify a default *renderer*

JTree class

- Shows the elements in a hierarchical way
- A tree consists of:
  - A root
  - Internal nodes
    - » Nodes that have at least a son
  - Leaf nodes
- The information from an **JTree** object is kept in a model



JTree class

- **DefaultMutableTreeNode**
  - Class that model a node of a tree
- Steps to create a tree:
  - Creating root
  - Instantiation of a **JTree** object with root previously defined
  - Add leaf nodes as sons of an existing nodes
- Tree nodes can be of different types
  - The default representation of nodes is obtained by calling **toString** for the considered object
- It is possible to specify a text in HTML format as value of a node

JTree class -Example of building a tree

```
String text = "<html><b>Radacina</b></html>";
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);
```

JTree class

- Another way to build a tree
  - By defining a class that implements **TreeModel** interface and describes the tree model
- Management of the selected items from a tree is made through a model:
  - The interface corresponding to the model is **TreeSelectionModel**
- **TreeSelectionListener**
  - Listener object that handle the events that are generated when the selection in tree is changed

JTree class - Example of handling events

```
class Test implements TreeSelectionListener{  ...
        public Test() { ...
          // establish the selection mode
           tree.getSelectionModel().setSelectionModel(TreeSelectionModel.SINGLE_TREE_SELECTION);

          // add listener
          tree.addTreeSelectionListener(this);
            ... }

          public void valueChanged(TreeSelectionEvent e){
             //get the selected node
             DefaultMutableTreeNode node = (DefaultMutableTreeNode) tree.getLastSelectedPathComponent();
             if (node == null)
                 //Nothing is selected.
                 return;
             Object nodeInfo = node.getUserObject();////get the data associated with the node.
             if (node.isLeaf()) {              //do something    }
            else {       //do something    }
          }}
```

JTree class

- Each node of the tree is represented by a **renderer** class that implements **TreeCellRenderer** interface
- By implementing the interface or extending the default class (i.e., **DefaultTreeCellRenderer**) we can personalize the tree nodes depending on their type and value

JTree class

- There are different ways to change how the tree show without to create new classes of **TreeCellRenderer** type:
  - **setRootVisible**
    - Specify if the root is visible or not
  - **setShowsRootHandles**
    - Specify if the nodes on the first level have symbols for expanding or collapsing the nodes
  - **putClientProperty**
    - Establishes different properties such as the way in which the lines between the father node and the son node is represented
      - To specify that the Java look and feel use only horizontal lines to group nodes, use the following code:
        - tree.**putClientProperty**("JTree.lineStyle", "Horizontal");
      - To specify that the Java look and feel should draw no lines, use this code:
        - tree.**putClientProperty**("JTree.lineStyle", "None");

JTree class

- Example of specifying an icon for a leaf node or for an internal node

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

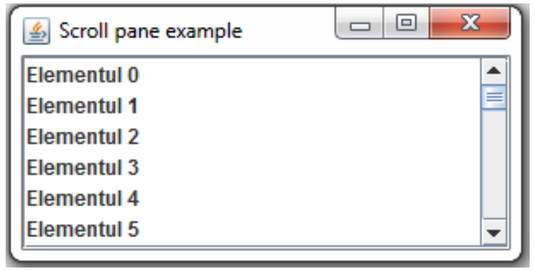
Swing containers

- High level containers
  - Are the root of the hierarchies of the components of an application
    - **JFrame, JDialog, JApplet**
- Intermediate containers
  - Are display areas used to arrange efficiently the application components
    - **JPanel, JScrollPane, JTabbedPane, JSplitPane, JLayeredPane, JDesktopPane, JRootPane**
  - Can be nested

JScrollPane class

- Provides a scrollable view of a component (e.g., horizontal and vertical scrolling of component)
- Is used to represent the components that do not fit on the display area

```
import javax.swing.*;
public class ScrollPaneDemo{
   public static void main(String args[]) {
  JFrame frame = new JFrame("Scroll pane example");
  String elemente[] = new String[100];
  for(int i=0; i<100; i++)
     elemente[i] = "Elementul " + i;
  JList lista = new JList(elemente);
  JScrollPane sp = new JScrollPane(lista);
  frame.getContentPane().add(sp);
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  frame.setSize(300, 150);
  frame.setVisible(true);
 } }
```



JTabbedPane class

- Is used for overlaying multiple containers (**JPanel** objects) on the same display area
- The selection of a panel is made at the top of the component

```
import javax.swing.*;
    public class TabbedPaneDemo{
     public static void main(String args[]) {
          JFrame frame = new JFrame(“TabbedPane");
          JTabbedPane tabbedPane = new JTabbedPane();
          ImageIcon icon = new ImageIcon("smiley.gif");
          JComponent panel1 = new JPanel();
          panel1.add(new JLabel("Hello"));
          tabbedPane.addTab("Tab 1", icon, panel1);
          JComponent panel2 = new JPanel();
          panel2.add(new JButton("OK"));
          tabbedPane.addTab("Tab 2", icon, panel2);
          frame.getContentPane().add(tabbedPane);
          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          frame.setVisible(true);
 } }
```

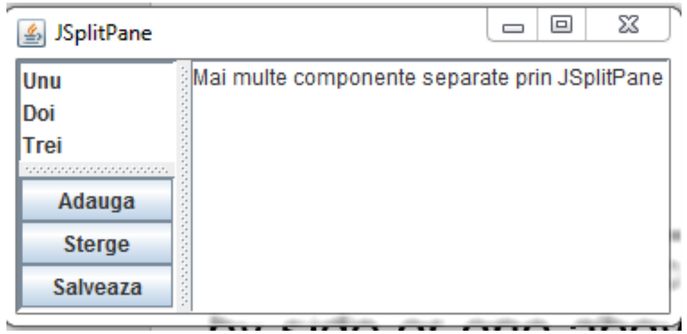
JSplitPane class

- Allows to create a container that contains two components that are arranged either side by side or one above the other
  - Separation of components is made through a bar that allows the configuration of display area allocated for each component

JSplitPane class - Example

```
import javax.swing.*;
public class SplitPaneDemo{
  public static void main(String args[]) {
   JFrame frame = new JFrame("JSplitPane");
   String elem[] = {"Unu", "Doi", "Trei" };
   JList list = new JList(elem);
   JPanel panel = new JPanel(new GridLayout(3, 1));
   panel.add(new JButton("Adauga"));
   panel.add(new JButton("Sterge"));
   panel.add(new JButton("Salveaza"));
   JTextArea text = new JTextArea("Mai multe
                componente separate prin JSplitPane");
```

```
    //Separam lista de grupul celor trei butoane
    JSplitPane sp1 = new JSplitPane(JSplitPane.
                                    VERTICAL_SPLIT, list, panel);
    //Separam containerul cu lista si butoanele de componenta pentru
    editare de text
    JSplitPane sp2 = new JSplitPane(JSplitPane.
                                    HORIZONTAL_SPLIT, sp1, text);
    frame.getContentPane().add(sp2);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(300, 150);
    frame.setVisible(true);
} }
```



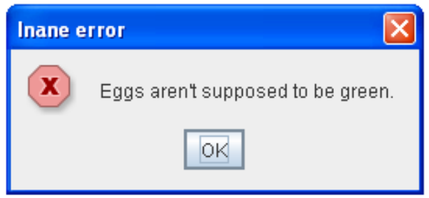
Swing – Dialogs

- Creating a dialog is made by extending **JDialog** class
- Predefine classes that describe certain types of dialogs
  - **JOptionPane –** allows to create simple dialogs used for:
    - »Displaying messages
    - »Making queries for confirmation / cancel, etc.
  - **JFileChooser**
    - »Standard dialog that allows to navigate through the file system
    - »Allow the selection of a specific file for open or save operations
  - **JColorChooser**
    - »Standard dialog for selecting a color
  - **ProgressMonitor**
    - »Class used to monitor the progress of time-consuming operations

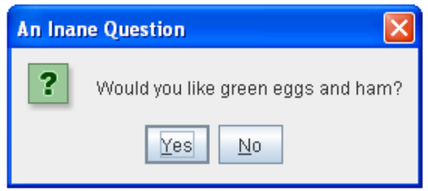
JOptionPane class

- Two ways of using **JOptionPane** class:

JOptionPane.showMessageDialog(frame, "Eggs are not supposed to be green.", "Inane error", **JOptionPane.ERROR_MESSAGE**);



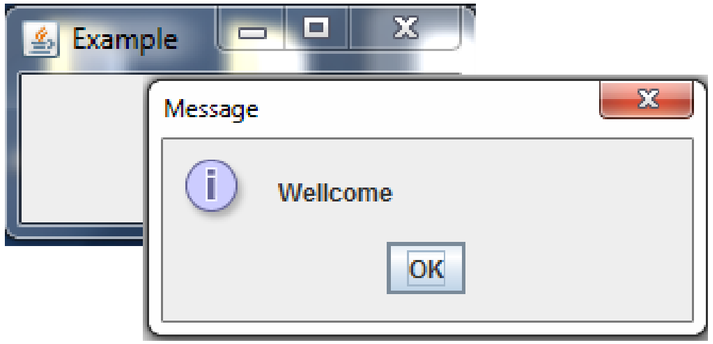
JOptionPane.showConfirmDialog(frame, "Would you like green eggs and ham?", "An Inane Question", **JOptionPane.YES_NO_OPTION);**



JOptionPane class

- Example of using **showMessageDialog**() from JOptionPane

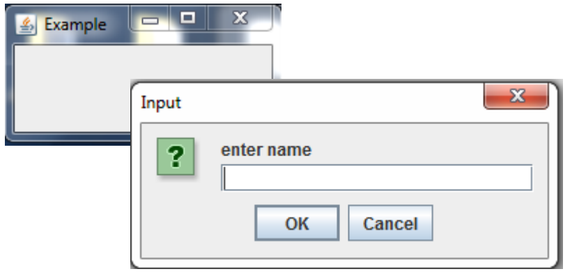
```
import javax.swing.*;
public class OptionPaneExample {
    JFrame f;
    OptionPaneExample(){
      f=new JFrame();
      f.setSize(200,100);
      f.setVisible(true);
      JOptionPane.showMessageDialog(f, "Wellcome");
  }
public static void main(String[] args) {
    new OptionPaneExample();
}
}
```



JOptionPane class

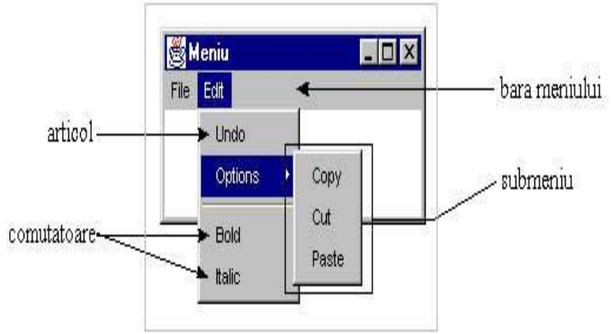
- Example of using **showMessageDialog**() from JOptionPane

```
import javax.swing.*;
public class OptionPaneExample {
    JFrame f;
    OptionPaneExample(){
      f=new JFrame();
      f.setSize(200,100);
      f.setVisible(true);
      String name=JOptionPane.showInputDialog(f,"Enter Name");
  }
public static void main(String[] args) {
    new OptionPaneExample();
}
}
```



Menus

- Fixed menus (permanently visible):
  - They are grouped in a menu bar
  - Contains items that can be selected, switches or others sub - menus
  - A frame can have only one menus



- Context menus (popup):
  - Invisible menus associated with a window that are activated by pressing the right button of the mouse
  - Are not grouped in a menu bar

Menus

- In Swing the menu bar objects are instance of J**MenuBar** class
- A J**MenuBar** object
  - Contains J**Menu** objects
- **JMenu** objects contain:
  - **JMenuItem** objects
  - **JCheckBoxMenuItem** objects
  - **JMenu** objects (sub-menus)

Menus

- Example of creating a menu bar:

```
// Crearea barei de meniuri
JMenuBar mb = new JMenuBar();
// Adaugarea meniurilor derulante la bara de meniuri...
// Atasarea barei de meniuri la o fereastra
JFrame f = new JFrame("Fereastra cu meniu");
f.setJMenuBar(mb);
```

Menus

- **JMenu** class
  - Allows to create a drop-down menu in the menu bar
  - Each menu has a label that represents the name that will be display on menu bar
  - A menu can contain instance of J**MenuItem**, J**Menu** or J**CheckboxMenuItem** classes
- **JMenuItem** class
  - Instance of J**MenuItem** class describing the individual options of the drop-down menus
    - E.g., ”Open”, ”Close”, ”Exit”, etc.
  - An instance of J**MenuItem** class
    - Is a button or a switch with a label that will appears on the menu
    - Can be accompanied by an accelerator
      - **MenuShortcut** object representing a key combination for calling quickly an item

Menus

- **JCheckboxMenuItem** class
  - Implement items of switching type; by proceedings the item it will move from one state to another
    - »By validating the switch from the right of the label, a graphical simbol will be displayed that indicate this thing
    - »By invalidating, the graphical symbol will disappear
  - Has the same functionality as boxes of type Checkbox, both implementing **ItemSelectable** interface

Menus -Example of creating menus

```
public class TestMenu{
  public static void main (String args []) {
    JFrame f = new JFrame("Test Menu");
    JMenuBar mb = new JMenuBar();
    JMenu fisier = new JMenu("File");
    fisier.add(new JMenuItem("Open"));
    fisier.add(new JMenuItem("Close"));
    fisier.addSeparator();
    fisier.add(new JMenuItem("Exit"));
    JMenu optiuni = new JMenu("Options");
    optiuni.add(new JMenuItem("Copy"));
    optiuni.add(new JMenuItem("Cut"));
    optiuni.add(new JMenuItem("Paste"));
    Menu editare = new JMenu("Edit");
    editare.add(new JMenuItem ("Undo"));
    editare.add(optiuni);
    editare.addSeparator();
    editare.add(new JCheckboxMenuItem("Bold"));
    editare.add(new JCheckboxMenuItem("Italic"));
    mb.add(fisier);
    mb.add(optiuni);
    mb.add(editare);
    f.setJMenuBar(mb);
    f.setSize(200, 100) ;
    f.show();
}}
```

Menus - Events handling

- When an option is selected from the menu, an event is generated:
  - **ActionEvent** for J**MenuItem**
  - **ItemEvent** for J**CheckboxMenuItem**
- To activate the menu options, we implement the following interface:
  - **ActionListener** with **actionPerformed** method
  - **ItemListener** with **itemStateChanged** method
- For each menu, a different receptor object can be associated
- The link between the menu object and the listener object is made by the following methods:
  - **addActionListener**
  - **addItemListener**

Menus - Events handling

- **JCheckboxMenuItem** objects implements **ItemSelectable** interface
  - Events handling is similar with the events handling for **List**, **Choice**, **CheckBox**
  - Operation type (i.e. check/uncheck) is encoding in generated event by the static fields:
    - **ItemEvent.SELECTED**
    - **ItemEvent.DESELECTED**

Menus - Events handling

```
public class Test extends JFrame implements ActionListener,
ItemListener
{
  public Test (String titlu) {
    super(titlu);
    JMenuBar mb = new JMenuBar();
    JMenu test = new JMenu(“Test”);
    JCheckboxMenuItem check = new JCheckboxMenuItem(“Check
me”);
    JMenuItem exit=  new JMenuItem("Exit");
    test.add(check);
    test.add(exit);
    mb.add(test);
    setJMenuBar(mb);
    JButton btnExit = new JButton("Exit");
    add(btnExit, BorderLayout.SOUTH);
```

```
 setSize(300, 200);
 setVisible(true);
 exit.addActionListener(this);
 btnExit.addActionListener(this);
 check. addItemListener(this);
 }
 public void actionPerformed(ActionEvent e) {
     String command = e.getActionCommand();
       if(command.equals("Exit"))
              System.exit(0); }

public void itemStateChanged(ItemEvent e) {
    if (e.getStateChange() == ItemEvent.SELECTED)
          setTitle("Checked !");
     else setTitle("Not checked !"); }
   public static void main ( String args []) {
      Test f = new Test("Tratare eveniment");
       } }
```

Menus - Context Menus

- Are implemented with help of J**PopupMenu** class
- Are invisible menus that are activated by pressing the right button of the mouse
- Methods for adding items to a context menu are inherited from fixed menus

```
JPopupMenu popup = new JPopupMenu("Options");
  popup.add(new JMenuItem("New"));
  popup.add(new JMenuItem("Edit"));
  popup.addSeparator();
  popup.add(new JMenuItem("Exit"));
```

Menus - Context Menus

- When we have more popup menus that are used in a window:
  - All of them must be defined
  - At a certain moment, the corresponding menu will be added on the window, and than it will be set as visible
  - After the menu is closed, the link between the window and the menu is broken my means of **remove** method

```
fereastra.add(popup1);
 ...
 fereastra.remove(popup1);
 fereastra.add(popup2);
```

Menus - Context Menus

- To display a context menu, we use **show** method:

popup.show(Component origin, **int** x, **int** y)

  - **origin**
    - Is the component in relation to which the display position of popup menu is computed (the position is calculated relative to the origin of the component);
    - Is an instance of the window in which the menu will be displayed

Menus - Context Menus

- Steps follows to create a menu context:
  - Creates the menu context
  - Activated the menu by pressing the right button of the mouse on the area of the main window
- Observation
  - Evens handling generated by the context menu is identical realized as in the case of fixed menus

Menus - Context Menus (Example)

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class ContextMenuExample extends JFrame {
    private JPopupMenu contextMenu;
    public ContextMenuExample() {
        setTitle("Context Menu Example");
        // create a popup menu
        contextMenu = new JPopupMenu();
        JMenuItem cutMenuItem = new JMenuItem("Cut");
        JMenuItem copyMenuItem = new JMenuItem("Copy");
        JMenuItem pasteMenuItem = new JMenuItem("Paste");
       // add action listeners to menu items
        cutMenuItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                // handle cut action
                System.out.println("Cut action performed"); }});
```

```
 copyMenuItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                // handle copy action
                System.out.println("Copy action performed");
            }
        });
pasteMenuItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                // handle paste action
                System.out.println("Paste action performed");
            }
        });
// add menu items to popup menu
contextMenu.add(cutMenuItem);
contextMenu.add(copyMenuItem)
contextMenu.add(pasteMenuItem);
```

Menus - Context Menus (Example)

```
// add mouse listener to component
        addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent e) {
                if (e.isPopupTrigger()) {
                    showContextMenu(e);}}
           public void mouseReleased(MouseEvent e) {
                if (e.isPopupTrigger()) {
                    showContextMenu(e);
                }         }});
// set frame properties
   setSize(300, 300);
   setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   setVisible(true);
    }
private void showContextMenu(MouseEvent e) {
        contextMenu.show(e.getComponent(), e.getX(), e.getY());
    }

public static void main(String[] args) {
        new ContextMenuExample();
    }}
```

- ✓ In this example, we create a JPopupMenu and add JMenuItem objects for Cut, Copy, and Paste.
- ✓ We add action listeners to the menu items to handle their respective actions.
- ✓ We add a MouseListener to the component (in this case, the JFrame) to detect when the user right-clicks on the component.
- ✓ If the user right-clicks, we display the context menu at the location of the mouse click.
- ✓ Finally, we set the properties of the JFrame and make it visible.

Menus - Accelerators

- An J**MenuItem** object can have associated an accelerator object, which is used to define a keyboard shortcut for the menu item
- An accelerator is a key combination that, when pressed, performs the action associated with the menu item.
- It can be specified using the setAccelerator method of the JMenuItem class, which takes an Accelerator object as its argument.

Menus – Accelerators (Example)

```
import java.awt.event.KeyEvent; import javax.swing.*;
public class JMenuItemKeyStroke {
  public static void main(final String args[]) {
    JFrame frame = new JFrame("MenuSample Example");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JMenuBar menuBar = new JMenuBar();
    // File Menu, F - Mnemonic
    JMenu fileMenu = new JMenu("File");
    fileMenu.setMnemonic(KeyEvent.VK_F);
    menuBar.add(fileMenu);
    // File->New, N - Mnemonic
    JMenuItem newMenuItem = new JMenuItem("New", KeyEvent.VK_N);
    fileMenu.add(newMenuItem);
    // Edit->Cut, T - Mnemonic, CTRL-X - Accelerator
    JMenuItem cutMenuItem = new JMenuItem("Cut", KeyEvent.VK_T);
    KeyStroke ctrlXKeyStroke = KeyStroke.getKeyStroke("control X");
    cutMenuItem.setAccelerator(ctrlXKeyStroke);
    fileMenu.add(cutMenuItem);
    frame.setJMenuBar(menuBar);
    frame.setSize(350, 250);
    frame.setVisible(true);  }}
```

- ✓ A JMenuBar is created and added to the JFrame.
- ✓ A JMenu called "File" is created with a mnemonic set to "F". The menu is added to the JMenuBar.
- ✓ A JMenuItem called "New" is created with a mnemonic set to "N". The menu item is added to the "File" menu.
- ✓ Another JMenuItem called "Cut" is created with a mnemonic set to "T". An accelerator is also set to "Ctrl+X". The menu item is added to the "File" menu.
- ✓ The JMenuBar is added to the JFrame, and the size and visibility of the JFrame are set.
- ✓ When you run this code, it will display a window with a menu bar containing a "File" menu. The "File" menu will have two menu items: "New" and "Cut".
- ✓ The "New" menu item will have a mnemonic of "N", and the "Cut" menu item will have a mnemonic of "T" and an accelerator of "Ctrl+X".

Drawing

- GUI must draw on the screen all its components that have a visual representation
- Drawing
  - Includes standard components used in application as well as the components defined by programmer
  - Is automatically made and is a process that is executed in the following situations:
    - At displaying for the first time of a component
    - At minimization operations, maximization operations or resizing the display area
    - As answer of an explicit request of the program

Drawing

- Methods that controls the drawing process are from **Component** class
  - void **paint(Graphics** g)
  - void **update(Graphics** g)
  - void **repaint**()
- The argument of the **paint** and **update** methods is an object of type **Graphics**
  - Represents the graphical context in which the components drawing is executed
- All drawings that must appear on a display surface are putted in **paint** method of a component

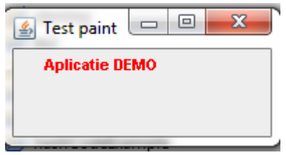
Drawing

- void **paint(Graphics** g)
  - Method from **Component** class
  - Draw a component
  - Don’t have an implementation
  - Is overridden for each component to provide its graphical specific representation
    - Standard AWT components have already overridden **paint** method
  - Is called each time when component content need to be draw or redraw
  - Is not explicit called

- Graphical representation of a standard component can be modified by:
  - Creating a subclass and overriding the **paint** method
  - Calling the superclass method that is responsible for drawing the component in paint method

Drawing - Example of redefining **paint** method

```
import java.awt.*;
import javax.swing.JFrame;
class Fereastra extends JFrame {
  public Fereastra (String titlu) {
    super(titlu);
    setSize(200, 100) ; }
 public void paint(Graphics g) {
   // Apelam metoda paint a clasei Frame
    super.paint(g);
    g.setFont(new Font ("Arial", Font.BOLD, 11));
    g.setColor(Color.red);
    g.drawString("Aplicatie DEMO", 30, 45);
 }
 public static void main ( String args []) {
   Fereastra f = new Fereastra ("Test paint");
   f.setVisible(true); }}
```



Drawing

- void **update**(Graphics g)
  - Method from **Component** class
  - Update the graphical state of a component
  - The method works in three steps:
    - Delete the component by redrawing her with background color
    - Establish the foreground color of the component
    - Call **paint** method to redraw the component
- void **repaint**()
  - Method from **Component** class
  - Execute explicit call of the **update** method to update the graphical representation of a component