12: Proiectarea Interfetelor Grafice

Cuprins

1. Infrastructura interfețelor grafice

Un utilizator poate interacționa cu o aplicație prin mai multe modalități: linie de comandă, componente grafice, mecanisme tactile, instrumente multimedia (voce, animație etc.) și componente inteligente (recunoașterea unor forme sau gesturi).

Interfața grafică cu utilizatorul (GUI - Graphical User Interface) reprezintă o modalitate de interacțiune vizuală între utilizator și aplicație, folosind componente grafice specifice (butoane, liste, meniuri etc.).

Java oferă o infrastructură complexă de pachete destinate realizării interfețelor grafice:

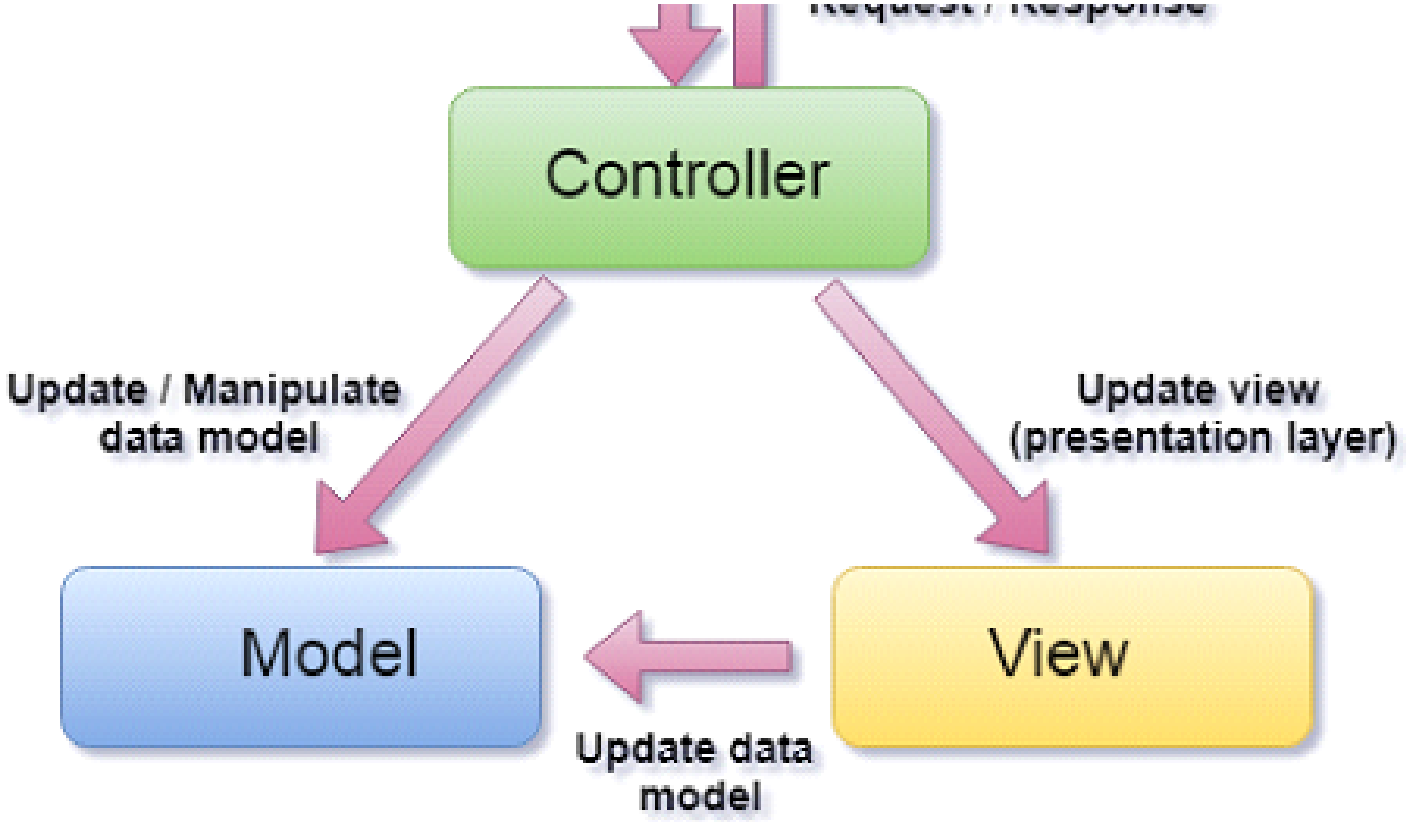
- Modelul AWT accesează componentele grafice ale sistemului de operare gazdă, respectiv crearea unei componente grafice va fi delegată către sistemul de operare, ci nu va fi realizată de către mașina virtuală.
- Deși are o serie de avantaje, precum o viteză bună de executare și o flexibilitate din punct de vedere al sistemului de operare utilizat (interfața se actualizează automat în momentul în care se schimbă versiunea sistemului de operare), are și o serie de dezavantaje, cum ar fi utilizarea unui set redus de componente grafice și lipsa portabilității (unele componente grafice pot avea aspect și funcționalitate diferită de la un sistem de operare la altul), care au condus la dezvoltarea unui API complex și performant denumit **Java Foundation Classes**.

Java Foundation Classes (JFC) este o arhitectură complexă de API-uri care pune la dispoziție o serie de facilități pentru proiectarea unei interfețe grafice performante. Arhitectura JFC este structurată pe mai multe module, precum:

- **Swing**: un API dedicat realizării unei interfețe grafice format din numeroase pachete de clase și interfețe performante, atât din punct de vedere funcțional, cât și din punct de vedere estetic;
- **Look-and-Feel**: un API care permite modificarea aspectului unei interfețe grafice în raport cu un anumit model, cum ar fi cele standard Windows, Mac, Java, Motif sau altele oferite de diverși dezvoltatori, acestea putând fi modificate de către utilizator chiar în momentul executării aplicației;
- **Accessibility**: un API care conține facilități dedicate persoanelor cu dizabilități, cum ar fi comenzi vocale, cititoare de ecran, dispozitive de recunoaștere a vocii etc.);
- **Java2D**: un API care conține facilități dedicate pentru crearea de desene complexe, efectuarea unor operații geometrice (rotiri, scalări, translații etc.), prelucrarea imaginilor etc.;
- **Internalization**: un API care permite dezvoltarea unor aplicații care pot fi configurate în raport cu diferite zone geografice, utilizând limba și particularitățile legate de formatarea datei, a numerelor sau a monedei din zona respectivă.

2. Componentele Swing

- Spre deosebire de componentele grafice din AWT, componentele Swing nu depind de sistemul de operare, fiind implementate direct în Java. Interfețele grafice realizate cu modelul Swing sunt mai lente decât cele realizate cu modelul AWT, fiind complet desenate de către mașina virtuală Java, însă oferă o paletă extinsă și performantă de componentele grafice, atât din punct de vedere funcțional, cât și din punct de vedere estetic.



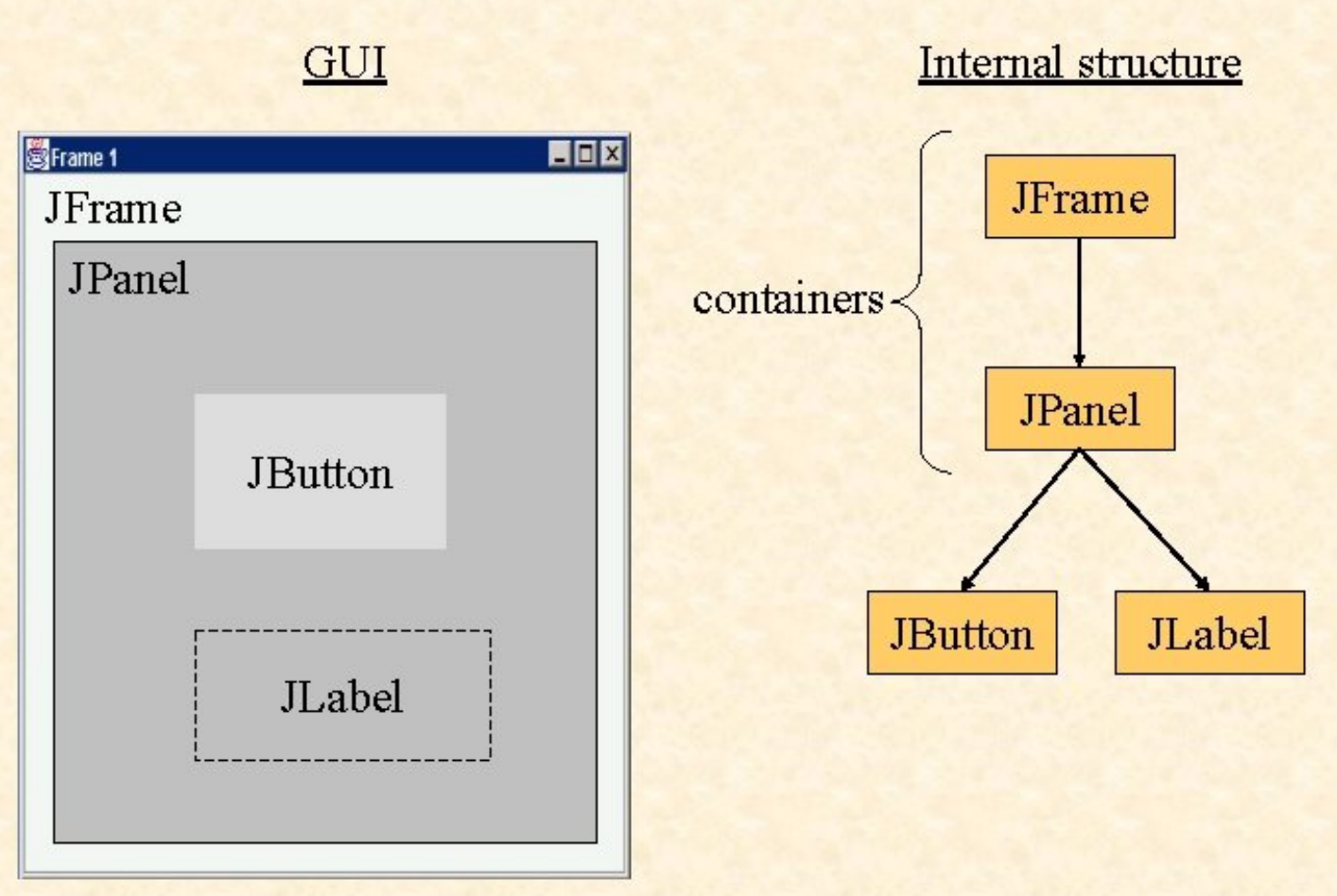
- Modelul Swing se bazează pe o arhitectură cu model separabil, respectiv arhitectura Model-View-Controller (MVC) care separă funcționalitatea aplicației de interfața sa grafică propriu-zisă. Cele trei componente ale arhitecturii au un rol bine definit și interacționează între ele astfel:
  1. componenta **Model** gestionează datele, care pot preluate dintr-o bază de date, fișier etc. și înștiințează componenta **Controller** în momentul în care acestea sunt modificate;
  2. componenta **View** are rolul de a reprezenta grafic datele din model și de a facilita interacțiunea cu utilizatorul, prin intermediul componentelor grafice;
  3. componenta **Controller** este cea care conectează componentele **Model** și **View**, definind modul în care interfața reacționează la acțiunile utilizatorului prin intermediul evenimentelor (click, apăsarea unei taste, închiderea unei ferestre etc.), recepționând mesajele primite de la componenta **View** după apariția unui anumit eveniment și trimițând mesaje componentei **Model** pentru a actualiza datele afișate de către componenta **View**.
- Pentru a realiza separarea componentei **Model** de componenta **View**, pentru fiecare componentă Swing este definită o clasă care gestionează datele și modul în care sunt tratate evenimentele asociate componentei grafice respective.
- În plus, o serie de componente Swing au asociate mai multe modele prin care pot fi gestionate datele, precum modelul **ButtonModel** (**JButton**, **JRadioButton**, **JMenu**, **JMenuItem**, **JCheckBox**) sau modelul **Document** (**JTextArea**, **JTextField**, **JEditorPane**).
- Practic, fiecare componentă are un model inițial implicit, însă acesta poate fi înlocuit printr-un model definit cu ajutorul unei clase care fie implementează interfața corespunzătoare, fie extinde clasa implicită oferită de API-ul Swing.
- De exemplu, interfața model a clasei **JList** este **ListModel** care este implementată de clasele **DefaultListModel** și **AbstractListModel**.

3. Etapele realizării unei interfețe grafice

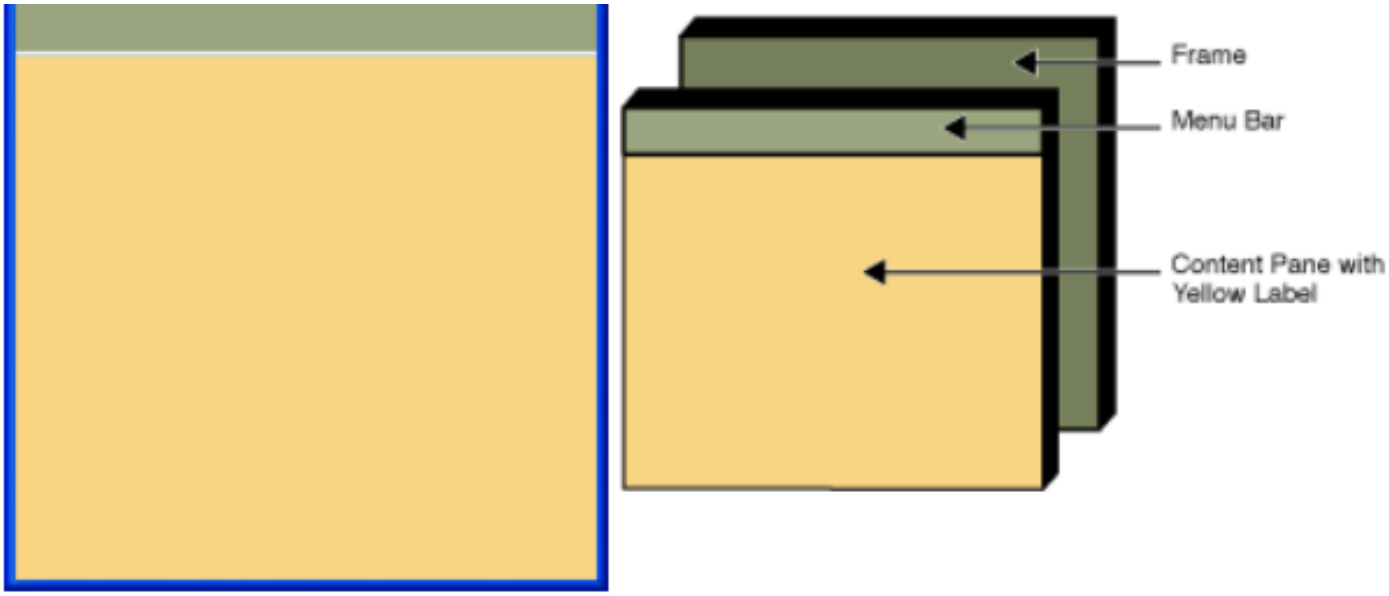
3. adăugarea unor **componente grafice** în containerele intermediare;
4. **poziționarea/alinierea componentelor** în containerele intermediare folosind gestionari de poziționare;
5. specificarea acțiunilor care trebuie efectuate în momentul apariției unui anumit **eveniment** lansat în urma interacțiunii utilizatorului cu o anumită componentă grafică.

4. Crearea unui container rădăcină

- Crearea componentelor grafice nu conduce implicit și la afișarea lor pe ecran.
- Mai întâi acestea trebuie să fie așezate pe o suprafața de afișare, astfel o interfața grafică Swing are o structură stratificată, compusă dintr-un container de tip rădăcină (root containers) ce încapsulează unul sau mai multe containere intermediare, care la rândul lor încapsulează mai multe componente grafice, precum butoane, liste, tabele etc..
- Astfel, definirea unui container rădăcină este esențială în procesul de proiectare a unei interfețe grafice, deoarece acesta este singura componentă Swing care poate fi afișată pe ecran.



- Swing pune la dispoziție mai multe containere de tip rădăcină, precum
  1. ferestrele definite prin clasa **JFrame**,
  2. dialogurile definite prin clasa **JDialog**,
  3. applet-urile definite prin clasa **JApplet**.
- Toate aceste clase sunt subclase ale clasei **JContainer**, care la rândul său este o subclasă a clasei abstracte **JComponent**, superclasa toturor claselor ce modelează componente grafice Swing.



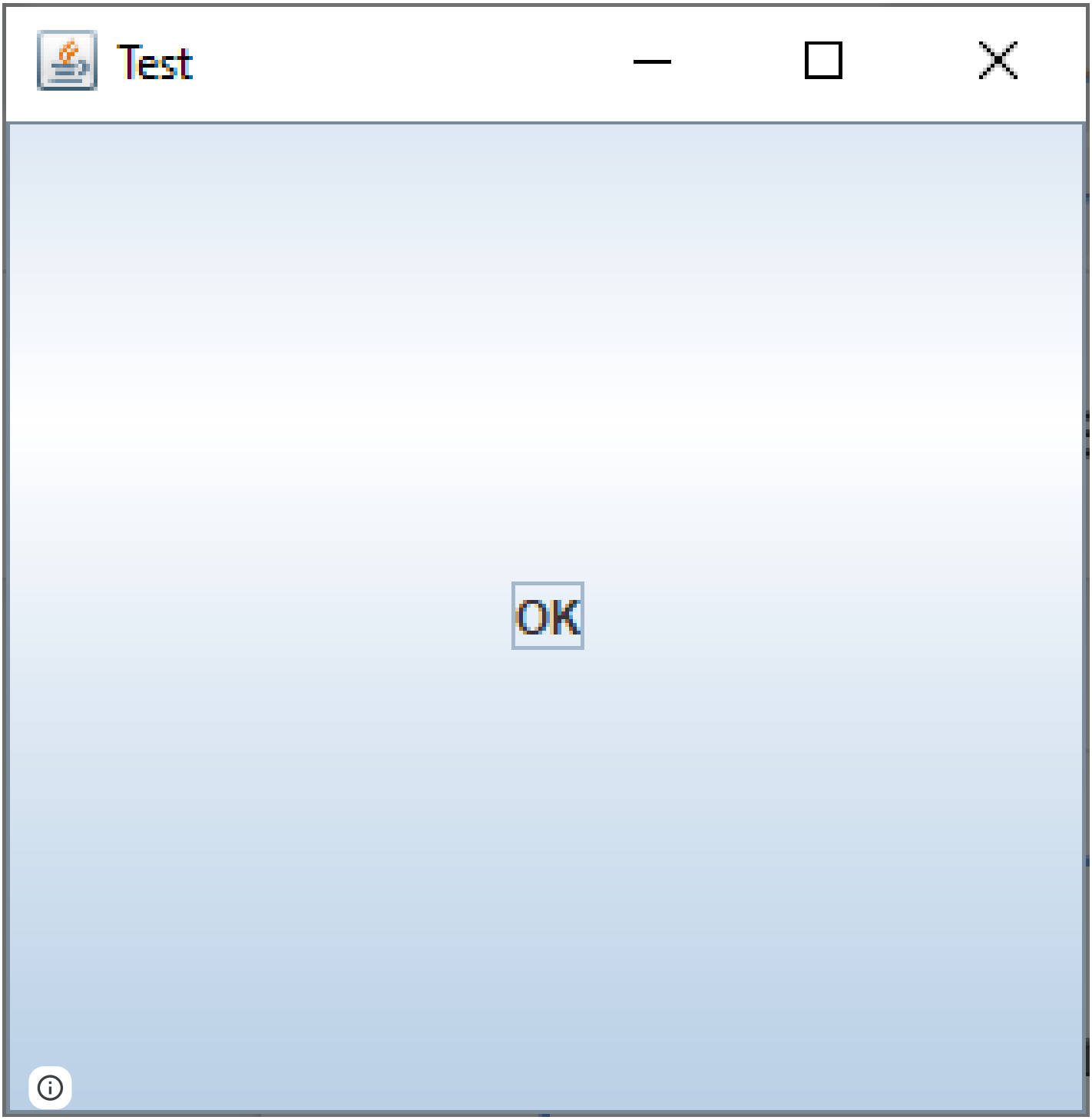
- Containerul **JFrame** este containerul cel mai des folosit pentru proiectarea unei interfețe grafice Swing.
- Containerul rădăcină **JFrame** definește o fereastră care conține:
  1. un icon,
  2. o bară de titlu,
  3. butoane predefinite pentru operațiile de redimensionare sau închidere a ferestrei,
  4. un container intermediar implicit de tip JPanel pe care se vor plasa alte componente grafice,
  5. precum și, opțional, o bară de meniu.

! Componente grafice elementare, precum butoane, liste, tabele etc. nu sunt plasate direct în containerul radăcină JFrame, ci sunt adăugate în containerul intermediar de tip JPanel, care este asociat implicit celui ferestrei.

Observații:

- containerul rădăcină este vizibil numai dacă se apelează metoda **setVisibile(true)** a clasei **JFrame**;
- accesarea suprafeței de afișare corespunzătoare containerului intermediar asociat unei ferestre **JFrame** se realiză prin apelul metodei **JPanel getContentPane**();
- toate clasele care modelează componente grafice încapsulează peste 100 de metode comune, moștenite din clasa **JComponent**, care permit adăugarea sau eliminarea unei componente grafice, modificarea aspectului containerului (dimensiuni, culoare, background, font), precum și modul de aliniere al componentelor grafice incluse în acesta. Toate aceste metode au formatul general Tip **getProprietate()**, respectiv **void setProprietate(Tip valoare).** Detalii despre aceste metode găsiți în pagina https://docs.oracle.com/javase/7/docs/api/javax/swing/JComponent.html.
- deși o fereastra JFrame nu mai este vizibilă în urma apăsării butonului de închidere, aplicația va continua să ruleze, deoarece implicit este setată proprietatea **JFrame.HIDE_ON_CLOSE** ca argument al metodei void **setDefaultCloseOperation()**. Astfel, pentru a închide atât fereastra, cât și aplicația, trebuie setată proprietatea **JFrame.EXIT_ON_CLOSE**.

```
//extragerea container intermediar asociat
JPanel panou = (JPanel) fereastra.getContentPane();
//definirea unei componente grafice elementare de tip JButton
JButton buton = new JButton("OK");
//adaugarea butonului pe suprafața containerului intermediar
panou.add(buton);
//setare dimensiunii ferestrei
fereastra.setSize(new Dimension(300,300));
//închiderea aplicației în momentul închiderii ferestrei
fereastra.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
//afișarea ferestrei pe ecran
fereastra.setVisible(true);
```



După cum ați văzut mai sus, o interfață grafică Swing conține un singur container rădăcină care poate să încapsuleze unul sau mai multe containere intermediare. Containerele intermediare, numite și panouri, au rolul de a grupa mai multe componente și de a le gestiona poziționarea, precum și dimensiunile lor.

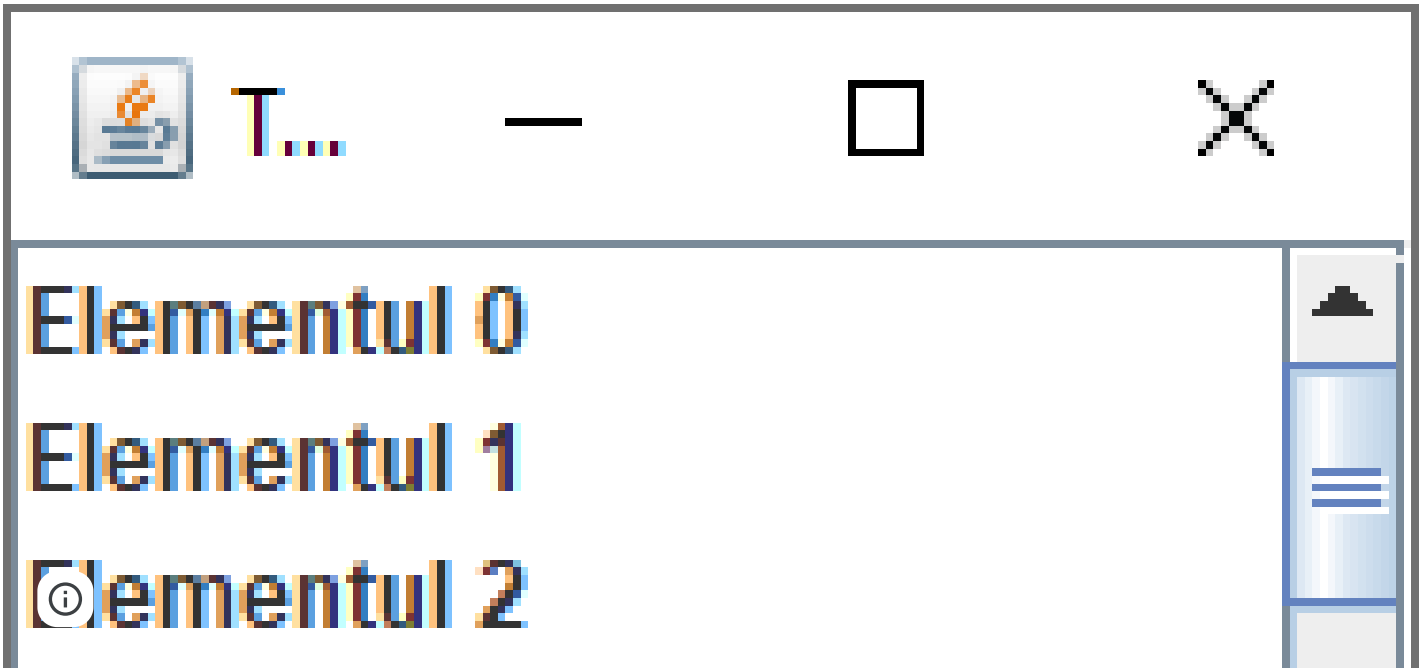
API-ul Swing pune la dispoziția programatorului mai multe containere intermediare, cum ar fi:

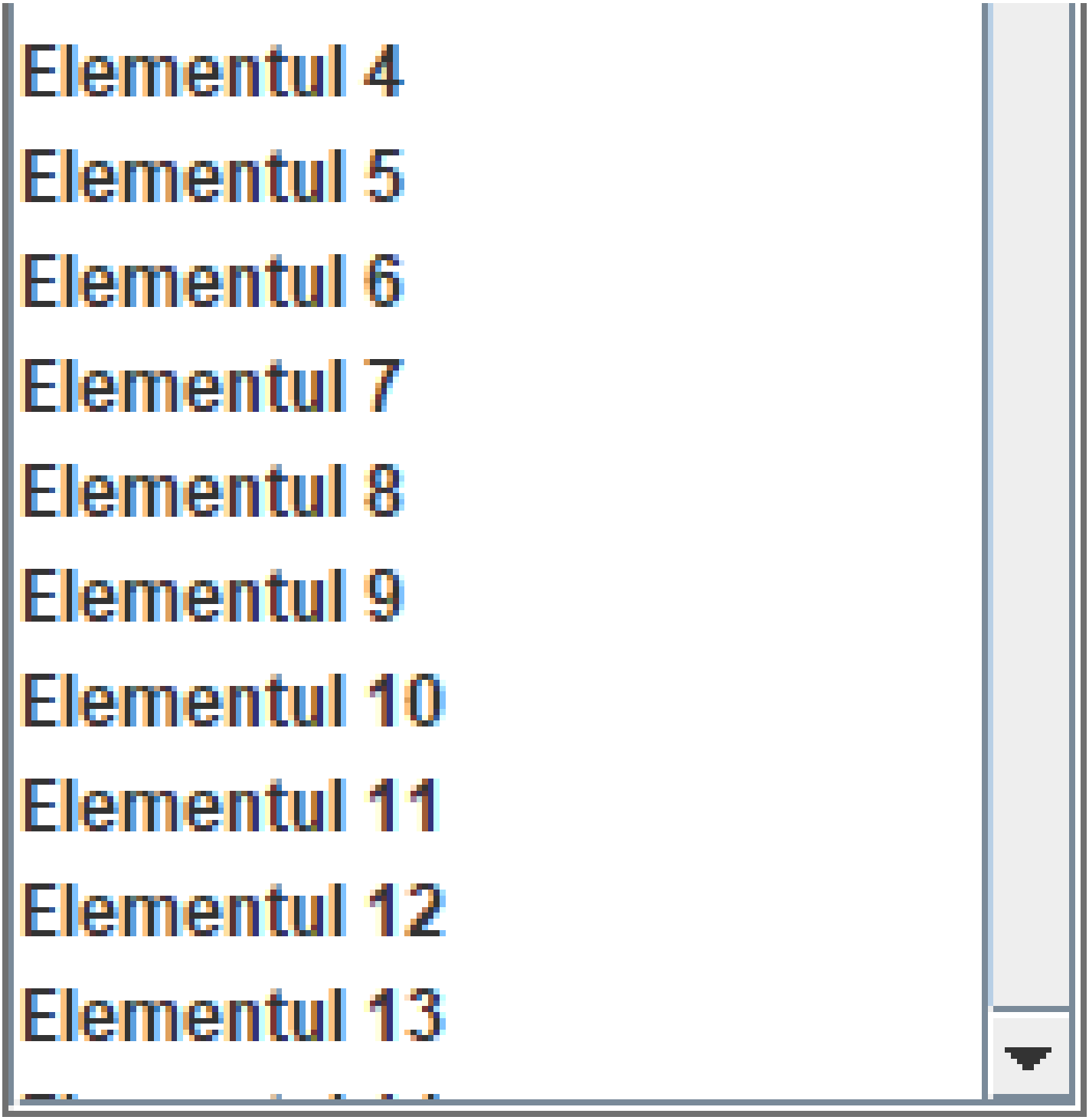
1. Containerul **JPanel** este cel mai utilizat panou pentru proiectarea unei interfețe grafice Swing
   - **Instanțierea unui obiect JPanel** se poate realiza printr-unul dintre constructorii:
     JPanel panou = new JPanel()
     JPanel panou = new JPanel(LayoutManager layout)
   - **Adăugarea unei componente grafice** se realizează prin apelul metodei **add(JComponent ob)**

2. Containerul **JScrollPane** are un rol important pentru estetica unei interfețe grafice, oferind suport pentru derularea pe verticală și orizontală a componentelor grafice a căror reprezentare completă nu se încadrează în suprafața asociată.
   - Este important de subliniat faptul ca nicio componentă grafică Swing nu oferă intrinsec acest suport, deci componente grafice precum tabele sau liste de mari dimensiuni se așază, de regulă, pe o suprafața definită de un panou **JScrollPane**.
   - Pentru derulare(scrolling), containerul JScrollPane pune la dispozitie o bara de derulare **JScrollBar**, iar zona de afișare este reprezentată de containerul JViewport.

Exemplu: Un panou cu bară de defilare care încapsulează o componentă grafică de tip listă:

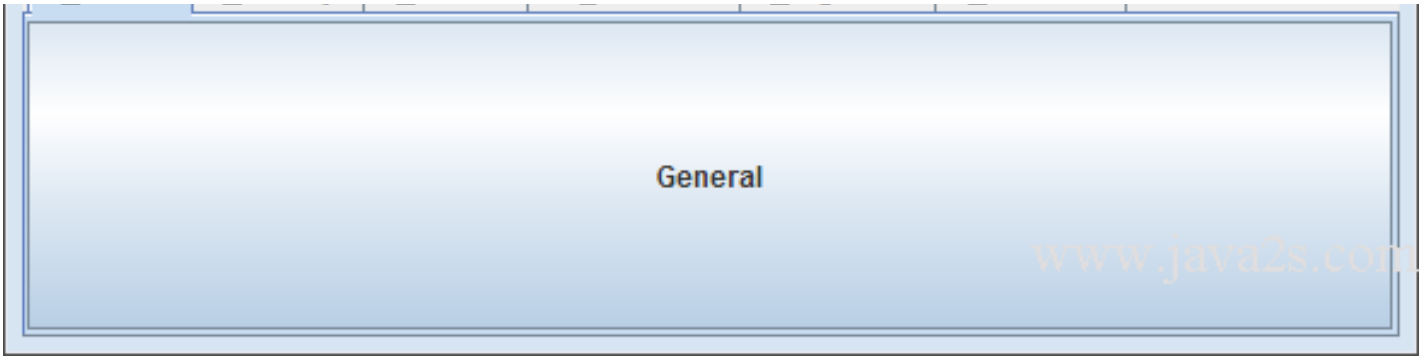
```
JFrame fereastra = new JFrame("Test");
JPanel panou = (JPanel) fereastra.getContentPane();

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);

panou.add(sp);
fereastra.setSize(new Dimension(200,300));
fereastra.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
fereastra.setVisible(true);
```





3. Containerul **JTabbedPane** este compus dintr-o stivă de componente de tip container intermediar (implicit de tip JPanel), așezate pe mai multe straturi suprapuse, însă doar unul dintre ele poate fi vizibil la un moment dat.

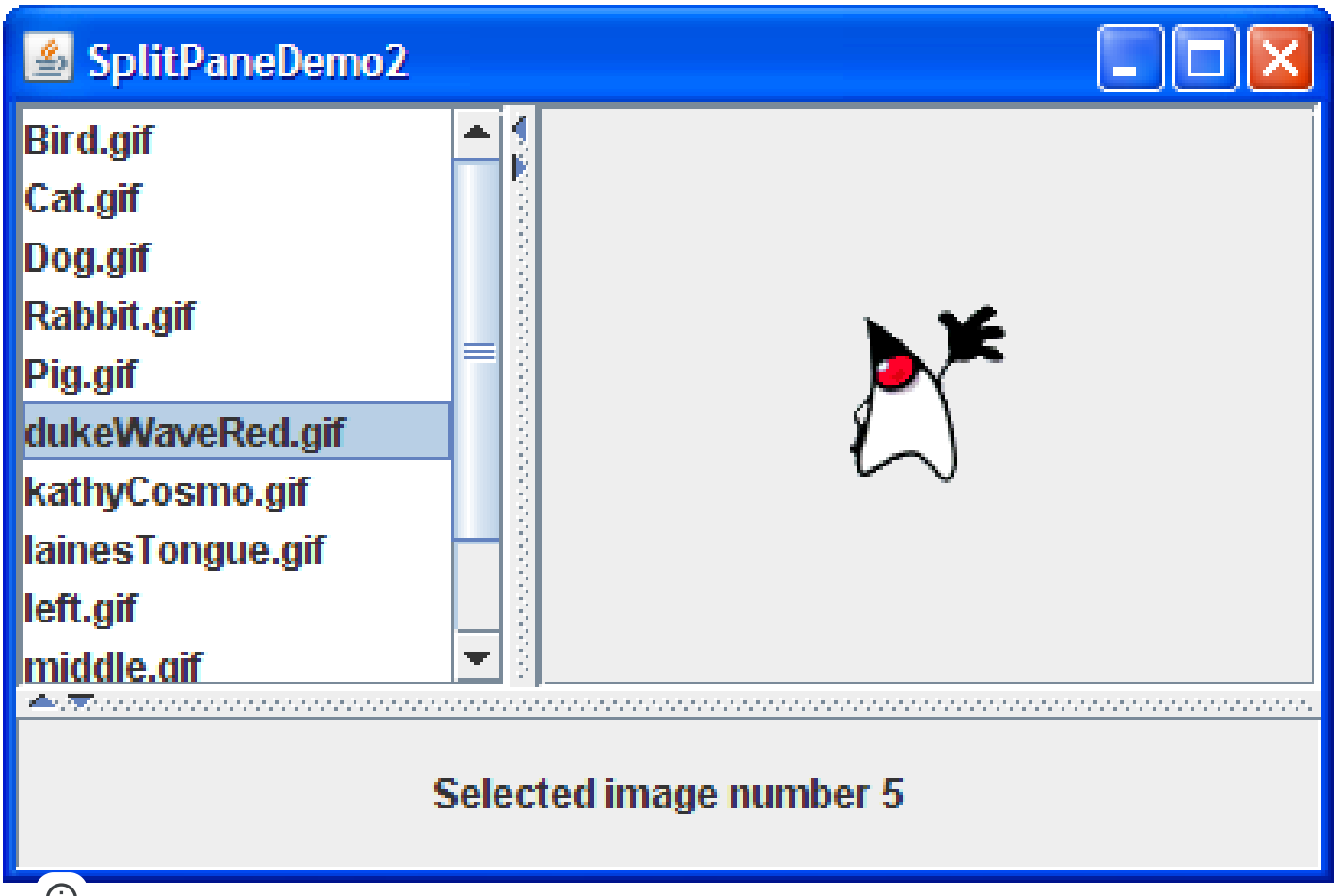
- Fiecare strat are câte o etichetă care poate conține text, pictograma etc., prin intermediul căreia utilizatori pot selecta stratul pe care vor să îl vizualizeze.
- Un nou strat se poate adăuga prin metoda addTab(String title, Icon icon, Component comp), unde primul argument specifică textul etichetei, al doilea argument este opțional și specifică un icon, iar ultimul specifică componenta grafică care va fi afișată în urma selectării stratului.



Java Tutorial - Set Mnemonic key for tab in JTabbedPane in Java (java2s.com)

4. Containerul **JSplitPane** este utilizat pentru crearea unui container format din două panouri alăturate, separate printr-un marcaj despărțitor, care permite vizualizarea simultană a două componente una lângă alta sau una deasupra celeilalte. Utilizatorul poate redimensiona cele două zone de afișare. Divizarea se poate realiza în direcția stânga-dreapta utilizând setarea **JSplitPane.HORIZONTAL_SPLIT** sau în direcția sus-jos utilizând **JSplitPane.VERTICAL_SPLIT**.

Dacă este nevoie de o interfaţă mai complexă, se poate imbrica o instanță **JSplitPane** într-o altă instanță **JSplitPane**. Astfel, se va putea mixa divizarea orizontală cu cea verticală. Granița de divizare poate fi ajustată de către utilizator cu mouse-ul, dar poate fi setată și prin apelul metodei **setDividerLocation**(). Dacă granița de diviziune este mutată cu mouse-ul de către utilizator, atunci se vor utiliza dimensiunile minime și maxime ale componentelor pentru a determina limitele deplasării graniței. Astfel, dacă dimensiunea minimă a celor două componente este mai mare decât dimensiunea containerului, codul JSplitPane nu va permite redimensionarea celor două componente.



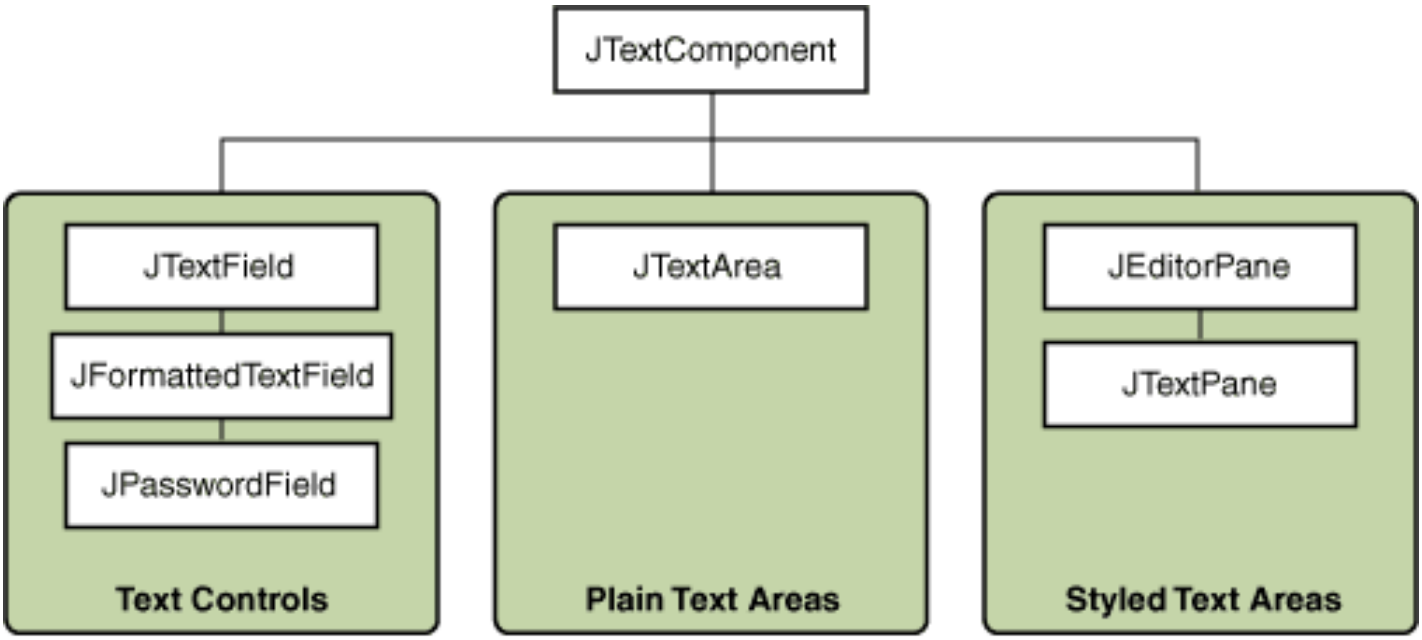
6. Adăugarea componentelor grafice elementare

Componentele grafice uzuale sunt cele care permit interacțiunea utilizatorul prin apăsarea unui buton, selecția unei opțiuni dintr-o listă etc. Acestea nu se pot afișa de sine-stătător, ci doar pot fi înglobate într-un container intermediar. Mai jos sunt prezentate o parte din componentele grafice elementare Swing:

Componenta grafică **JLabel**

- definește o etichetă care permite afișarea unor informații care nu pot fi selectate, precum text sau imagine;
- rolul unei etichete este acela de a oferi informații despre alte componente grafice aflate în container;
- textul sau imaginea unei etichete poate fi specificată prin argumentul constructorului sau poate fi specificată prin apelul metodelor void setText (String text), respectiv void setIcon(Icon icon);
- pentru formatarea textului unei etichete se poate utiliza și cod HTML.

Componente grafice pentru editarea unui text: componentele Swing pentru afișarea și, respectiv, editarea unui text sunt grupate într-o ierarhie cu rădăcina în clasa JTextComponent din pachetul javax.swing.text



simplu, afișat pe o singură linie, componenta de tip **JPasswordField** este utilizată pentru introducerea unei parole (fiecare caracter din text va fi înlocuit printr-un caracter, de exemplu caracterul '*'), iar componentade tip **JFormattedTextField** permite introducerea unui text care respectă un anumit format, fiind foarte utilă pentru citirea unor date numerice, calendaristice etc. (pentru utilizarea eficientă a textului formatat se pot utiliza clase utilitare dedicate, precum NumberFormatter, DateFormatter, MaskFormatter etc.).

2. **text simplu pe mai multe linii (arii de text)**: componenta grafică de tip **JTextArea** permite editarea unui text simplu, care poate fi afișat pe mai multe linii. Uzual, o componentă grafică de acest tip trebuie inclusă într-un container **JScrollPane**, pentru a permite navigarea pe verticală și orizontală dacă textul nu se încadrează în suprafața alocată ariei de text. Atributele textului (font, culoarea și dimensiunea fontului etc.) se pot aplica doar pentru întreg textul cuprins în aria de text, neputând fi specificate doar pentru o anumită porțiune a textului.
3. **text stil îmbogățit pe mai multe linii**: clasa **EditorPane** modelează o componentă grafică care permite afișarea și editarea unui text care poate avea multiple stiluri. De asemenea, zona de text poate să includă și imagini sau alte componente grafice. Implicit, următoarele tipuri de texte sunt recunoscute: text/plain, text/html și text/rtf. Clasa **JTextPane**, care extinde clasa **EditorPane**, modelează o componentă grafică care permite afișarea și editarea unui text cu facilități multiple, precum utilizarea paragrafelor, a stilurilor multiple etc.

- Pentru obiecte de tipul unor clase derivate din clasa **JTextComponent** extragerea unui text se realizează prin apelul metodei **getText**, respectiv editarea sa prin apelul metodei **setText**, mai puțin pentru editoarele de text formatat, pentru care extragerea informației se realizează prin apelul metoda **getValue**, respectiv editarea prin apelul metodei **setValue**;
- În plus, toate obiectele provenite din **JTextComponent** oferă suport pentru operații complexe, precum undo/redo, tratarea evenimentelor generate de cursor etc. Practic, orice obiect derivat din clasa **JTextComponent** este format din următoarele module: modelul **Document** care gestionează starea unei componente grafice de tip text, o reprezentare (view) care este responsabilă cu modalitatea de afișare a textului și un controller (editor kit) care permite scrierea și citirea textului, fiind responsabil cu definirea acțiunilor necesare editării;
- Informații detaliate despre toate componentele grafice care pot fi utilizate pentru editarea unui text se găsesc în pagina https://docs.oracle.com/javase/tutorial/uiswing/components/text.html

Componenta grafică **JButton**

- este o componentă care, prin apăsare, lansează un eveniment care trebuie tratat de către programator;
- un buton poate conține un text sau o imagine care pot fi specificate prin argumentul constructorului;
- dacă un buton conține o imagine, atunci, opțional, acestuia i se pot asocia imagini diferite pentru stările sale (normal, selectat, apăsat, cursorul mouse-ului se află deasupra suprafeței butonului și dezactivat), utilizând metodee specifice: https://docs.oracle.com/javase/7/docs/api/javax/swing/AbstractButton.html
- un buton poate fi activ sau inactiv. De exemplu, pentru aplicațiile care conțin formulare bazate pe ferestre de dialog este util să se dezactiveze butonul OK până când utilizatorul completează toate câmpurile obligatorii. Pentru a activa sau dezactiva un buton se utilizează metoda **void setEnabled(boolean bState)**, unde bState este true (pentru activare) sau false (pentru dezactivare). Dacă butonul este dezactivat, va fi redesenat într-o nuanță de gri șters.

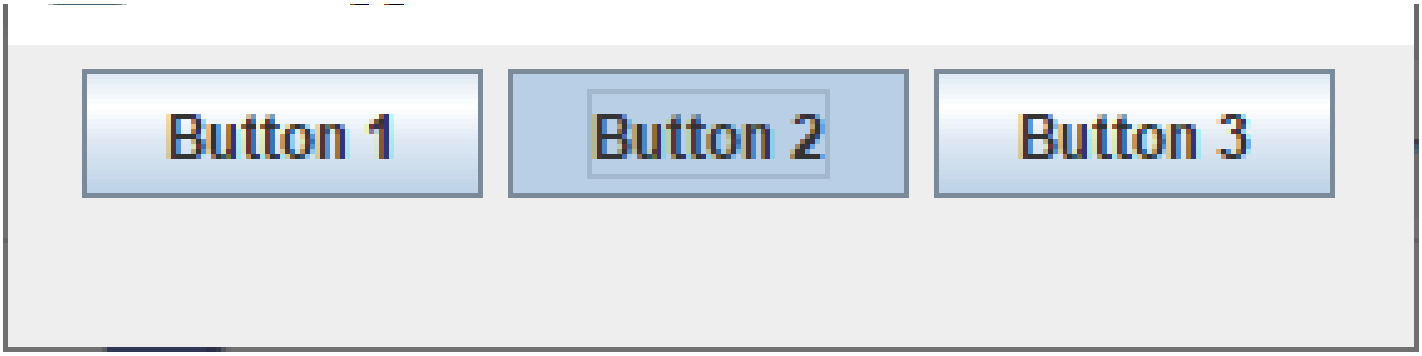
```
topPanel.setLayout(new FlowLayout());
JButton button1 = new JButton("Buton activ");
topPanel.add(button1);
JButton button2 = new JButton("Buton inactiv");
topPanel.add(button2);
button2.setEnabled(false);
fereastra.setSize(new Dimension(300,100));
fereastra.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
fereastra.setVisible(true);
```

Componenta grafică **JToggleButton**

- are aceeași funcționalitate precum componenta grafică **JButton**, dar adăugând suplimentar posibilitatea de a simula un comutator cu două stări. Butoanele de tip **JToggleButton** au o funcționalitate similară tastei Caps Lock, în timp ce butoanele **JButton** operează într-o manieră similară tastelor alfanumerice;
- clasa **JToggleButton** furnizează un mecanism press-and-hold, fiind astfel utile pentru interfețe grafice care necesită operații modale;
- mai multe butoane **JToggleButton** pot fi grupate, utilizând clasa ButtonGroup, astfel încât, la un moment dat, un singur buton din grup să fie apăsat.

Exemplu:

```
JFrame fereastra = new JFrame("Test JToggleButton ");
JPanel topPanel = (JPanel) fereastra.getContentPane();
topPanel.setLayout(new FlowLayout());

JToggleButton button1 = new JToggleButton("Button 1", true);
topPanel.add(button1);
JToggleButton button2 = new JToggleButton("Button 2", false);
topPanel.add(button2);
JToggleButton button3 = new JToggleButton("Button 3", false);
topPanel.add(button3);

ButtonGroup buttonGroup = new ButtonGroup();
buttonGroup.add(button1);
buttonGroup.add(button2);
buttonGroup.add(button3);

fereastra.setSize(new Dimension(300,100));
fereastra.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
fereastra.setVisible(true);
```

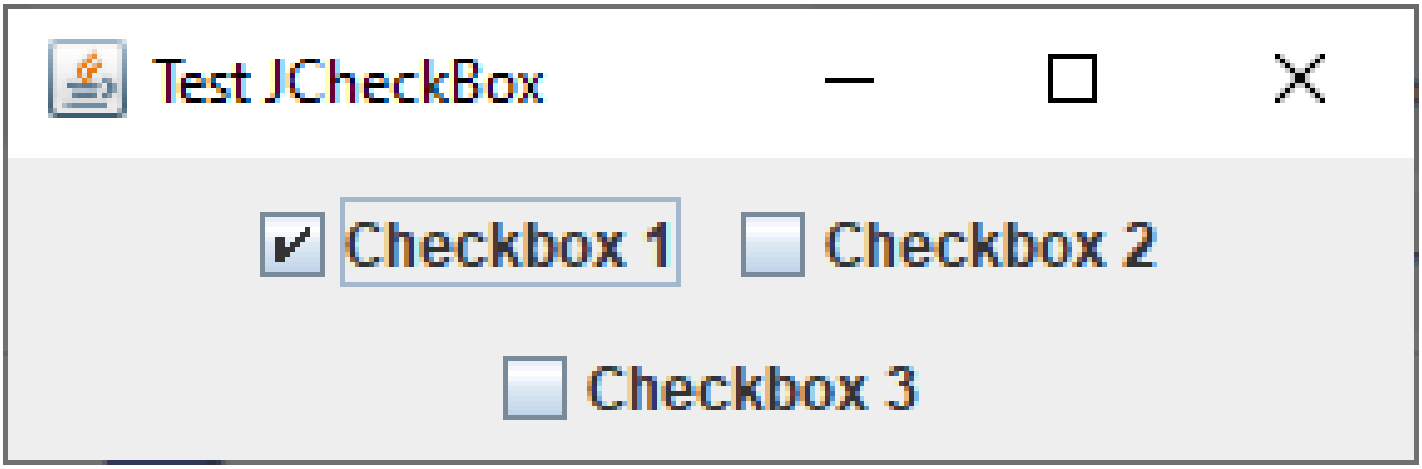


Componenta grafică **JCheckBox**

- clasa **JCheckBox**, care extinde clasa **JToggleButton**, implementează un control standard de selecție a unei opțiuni;
- un check box are două stări care pot fi setate de către utilizator cu ajutorul mouse-ului sau utilizând metoda **void setSelected(Boolean b)**;
- de obicei, componentele de tip **JCheckBox** sunt "grupate" folosind un chenar pentru a indica utilizatorului faptul că poate să selecteze oricâte opțiuni dintre cele prezentate, inclusiv niciuna (http://www.java2s.com/Code/Java/Swing-JFC/RadiobuttonComboBox.htm).

Exemplu:

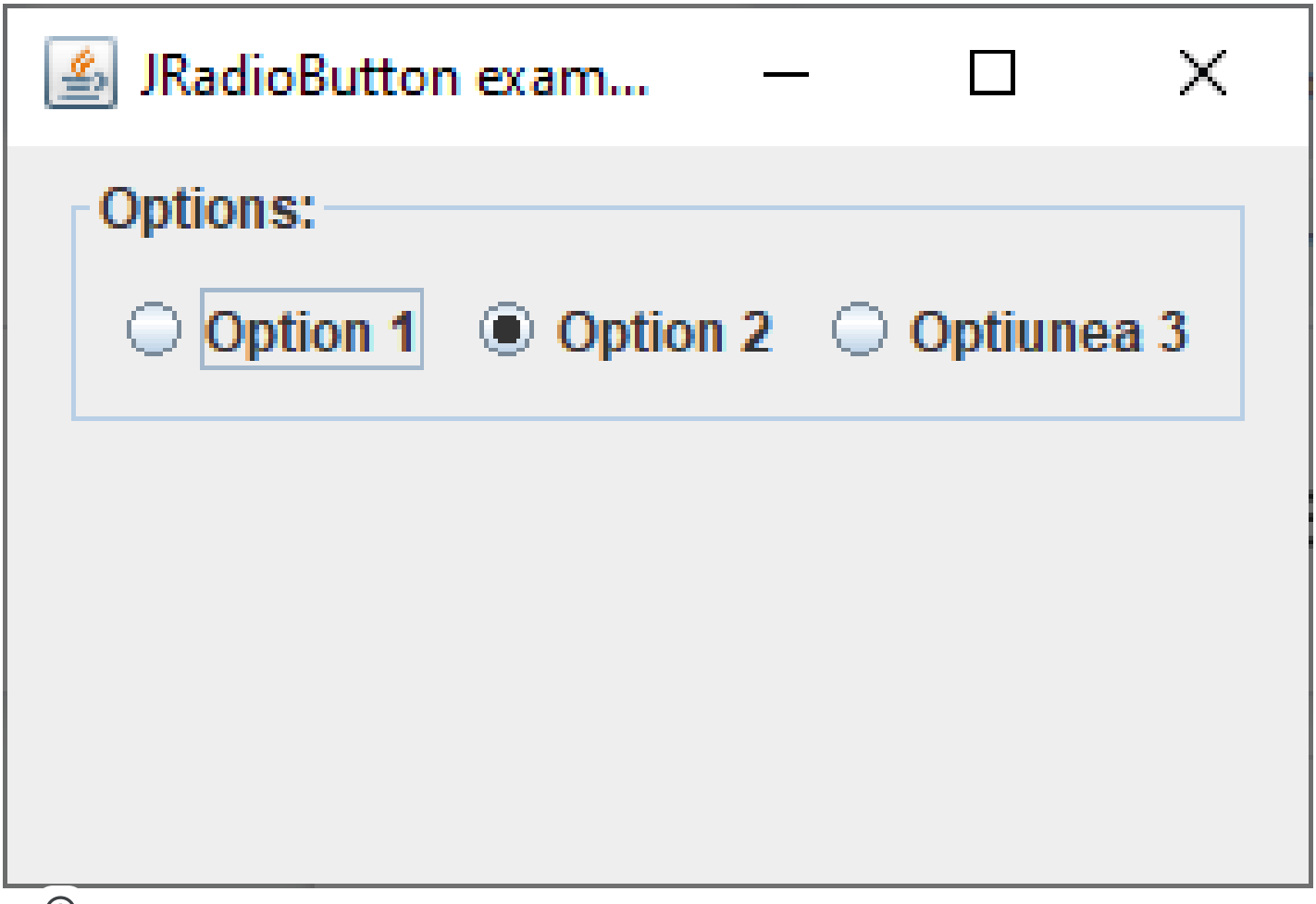
```
JFrame fereastra = new JFrame("Test JCheckBox ");
JPanel topPanel = (JPanel) fereastra.getContentPane();
topPanel.setLayout(new FlowLayout());

JCheckBox check1 = new JCheckBox("Checkbox 1");
check1.setSelected(true);
topPanel.add(check1);

JCheckBox check2 = new JCheckBox("Checkbox 2");
topPanel.add(check2);

JCheckBox check3 = new JCheckBox("Checkbox 3");
topPanel.add(check3);

fereastra.setSize(new Dimension(300, 100));
fereastra.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
fereastra.setVisible(true);
```



obicei, doar în cadrul unui grup de butoane (o instanță a clasei **ButtonGroup**) astfel încât opțiunile respective să se excludă reciproc (utilizatorul va putea selecta, la un moment dat, o singură opțiune).

Exemplu:

```
JFrame fereastra = new JFrame("JRadioButton example");
fereastra.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
fereastra.setSize(new Dimension(300, 200));

JPanel topPanel = (JPanel) fereastra.getContentPane();
topPanel.setLayout(new FlowLayout());

JPanel radioPanel = new JPanel(new FlowLayout());
TitledBorder border = BorderFactory.createTitledBorder("Options:");
radioPanel.setBorder(border);
topPanel.add(radioPanel);
JRadioButton radio1 = new JRadioButton("Option 1");
radioPanel.add(radio1);
JRadioButton radio2 = new JRadioButton("Option 2", true);
radioPanel.add(radio2);
JRadioButton radio3 = new JRadioButton("Optiunea 3");
radioPanel.add(radio3);
ButtonGroup group = new ButtonGroup();
group.add(radio1);
group.add(radio2);
group.add(radio3);

fereastra.setVisible(true);
```

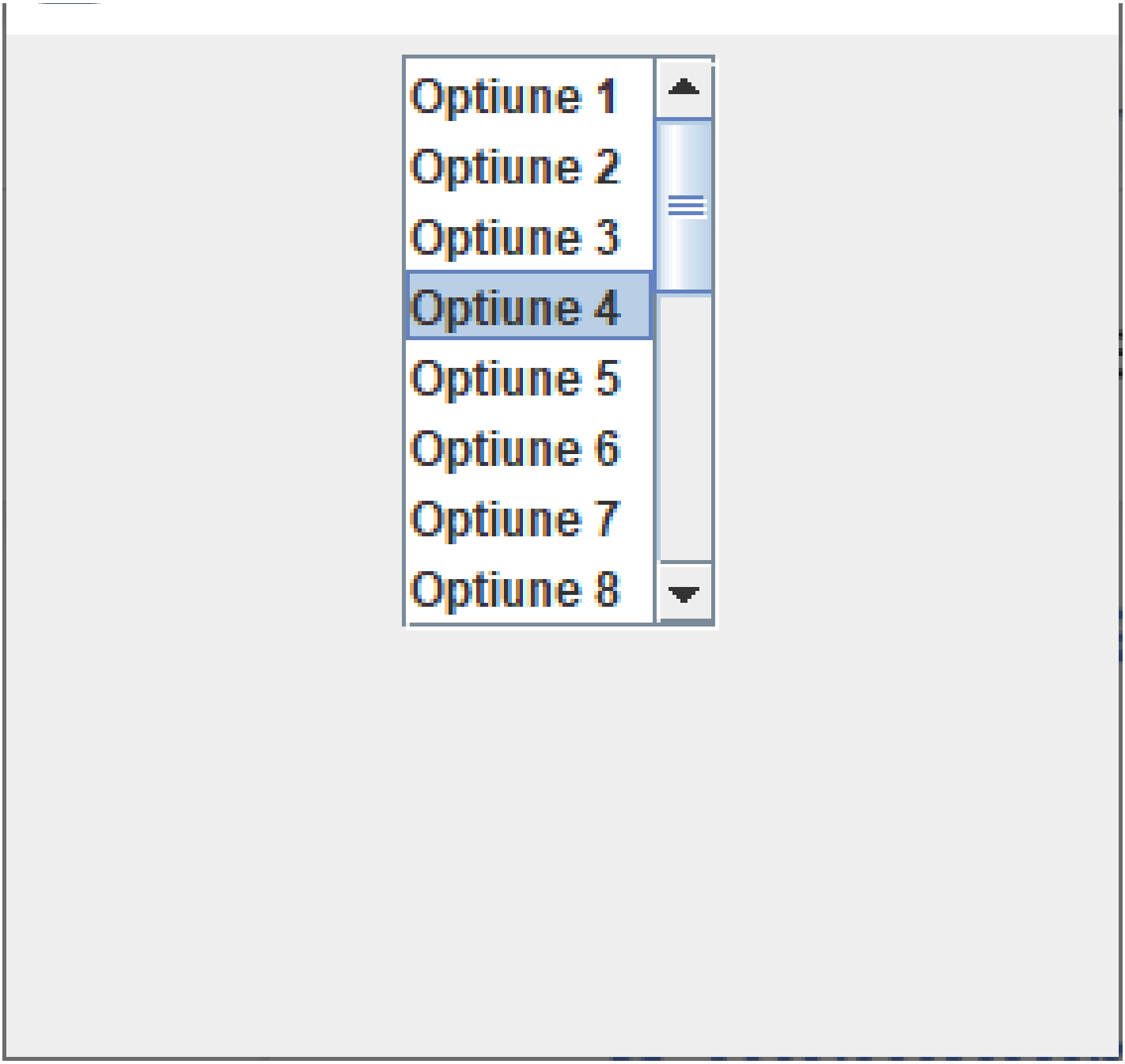


lista de contacte, lista melodiilor preferate etc.;

- în plus, o opțiune din listă (item) poate să interacționeze cu utilizatorul, astfel încât selectarea sa să conducă la declanșarea unui anumit eveniment;
- o listă nu are implicit bara de defilare verticală, ci ea trebuie așezată într-un container **JScrollPane**;
- conținutul unei liste se poate defini cu ajutorul unei structuri de date sau cu ajutorul unui model care este o instanță a clasei ListModel;
- dacă unei liste nu i se asociază niciun model, implicit se va utiliza modelul **DefaultListModel**, care memorează lista sub forma unui vector de obiecte de tip **Object**.
- suplimentar, există și clasa **AbstratcListModel**, care permite utilizatorului să aleagă tipul structurii de date pe care o va utiliza pentru elementele listei;
- pentru a extrage opțiunea selectată de către utilizator se apelează metoda **getSelectedValue()**, iar pentru a selecta o anumită opțiune se folosește metoda **setSelectedIndex(int index).**

Exemplu:

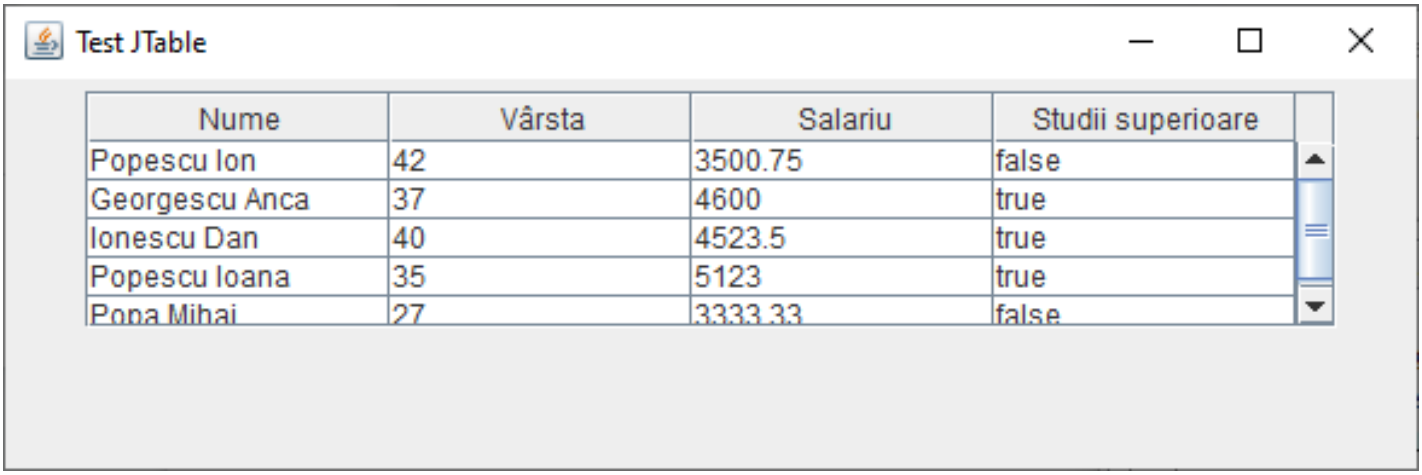
```
JFrame fereastra = new JFrame("Test JList");
JPanel topPanel = (JPanel) fereastra.getContentPane();
topPanel.setLayout(new FlowLayout());
Vector v = new Vector();

for (int i = 0; i < 20; i++)
    v.add(i, "Optiune " + (i+1));
JList lista = new JList(v);
lista.setSelectedIndex(3);

JScrollPane listPanel = new JScrollPane(lista);
topPanel.add(listPanel);
fereastra.setSize(new Dimension(300, 300));
fereastra.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
fereastra.setVisible(true);
```



Componenta grafică **JTable**

- permite o vizualizare a datelor într-un format tabelar, fără însă ca acestea să fie stocate de componentă;
- deoarece componenta grafică **JTable** nu are atașată implicit și o bară de defilare, de obicei, un tabel se afișează într-un container de tip **JScrollPane**;
- conținutul unei tabel se poate stabili cu ajutorul unor structuri de date, atât pentru a defini coloanele, cât și pentru a introduce informațiile de pe linii, sau cu ajutorul unui model care este o instanță de tip **TableModel**.

```
topPanel.setLayout(new FlowLayout());

String numeColoane[] = {"Nume", "Vârsta", "Salariu", "Studii superioare"};

Object valoriLinii[][] = {
    {"Popescu Ion", 42, 3500.75, false},
    {"Georgescu Anca", 37, 4600, true},
    {"Ionescu Dan", 40, 4523.50, true},
    {"Popescu Ioana", 35, 5123, true},
    {"Popa Mihai", 27, 3333.33, false}
};

JTable table = new JTable(valoriLinii, numeColoane);
table.setPreferredScrollableViewportSize(new Dimension(500, 75));
JScrollPane tablePanel = new JScrollPane(table);
topPanel.add(tablePanel);

fereastra.setSize(new Dimension(600, 200));
fereastra.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
fereastra.setVisible(true);
```



- dacă un tabel este definit prin structuri de date, toate celulele tabelului vor fi editabile, valorile din toate celulele vor fi implicit considerate ca fiind de tip String și toate coloanele tabelului vor avea aceeași lățime, altfel, manipularea datelor dintr-un tabel trebuie realizată prin intermediul unui model;
- dacă nu se utilizează un model explicit, atunci este creată automat o instanță a clasei **DefaultTableModel**;
- pentru a modifica proprietățile implicite ale unui tabel se poate defini o clasă care extinde clasa **DefaultTableModel**.

```
    public modelPropriuTabel(Object[][] data, Object[] columnNames) {
        super(data, columnNames);
    }

    @Override
    //stabilim coloanele ale căror celule vor fi editabile
    public boolean isCellEditable(int row, int column) {
        if (column == 0)
            return false;
        return true;
    }

    @Override
    //returnăm tipul exact al unei valori dintr-o celulă, astfel încât
    //valorile vor fi afișate formatat (numerele vor fi aliniate la dreapta,
    //iar valorile de tip boolean vor fi afișate cu bife
    public Class getColumnClass(int columnIndex) {
        return getValueAt(0, columnIndex).getClass();
    }
}

public class Test {
    public static void main(String[] args) {
        JFrame fereastra = new JFrame("Test JTable");
        JPanel topPanel = (JPanel) fereastra.getContentPane();
        topPanel.setLayout(new FlowLayout());

        String numeColoane[] = {"Nume", "Vârsta", "Salariu", "Studii superioare"};

        Object valoriLinii[][] = {
            {"Popescu Ion", 42, 3500.75, false},
            {"Georgescu Anca", 37, 4600, true},
            {"Ionescu Dan", 40, 4523.50, false},
            {"Popescu Ioana", 35, 5123, true},
            {"Popa Mihai", 27, 3333.33, true}
        };

        JTable table = new JTable();
        table.setModel(new modelPropriuTabel(valoriLinii, numeColoane));

        table.setPreferredScrollableViewportSize(new Dimension(500, 75));

        JScrollPane tablePanel = new JScrollPane(table);
        topPanel.add(tablePanel);

        fereastra.setSize(new Dimension(600, 200));
        fereastra.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        fereastra.setVisible(true);
    }
}
```

Nume	Varsta	Salariu	Studii superioare
Popescu Ion	42	3,500.75	☐
Georgescu Anca	37	4,600	☑
Ionescu Dan	40	4,523.5	☐
Popescu Ioana	35	5,123	☑
Popa Mihai	27	3,333.33	☑

7. Poziționarea elementelor. Gestionari de poziționare

Poziționare inițială	Redimensionare
	

- Rolul unui gestionar de poziționare este acela de a stabili, automat, poziția și dimensiunea fiecărei componente grafice dintr-un container.
- În lipsa unui astfel de gestionar, componentele grafice nu vor fi redimensionate sau repoziționate în momentul redimensionării containerului care le înglobează.
- Modul de aranjare a componentelor pe o suprafața de afișare nu este o caracteristică a containerului.
- Fiecare obiect de tip **Container** are asociat un obiect care are rolul de a așeza componentele grafice pe suprafața sa, respectiv gestionarul său de poziționare.
- Toate clasele care instanțiază obiecte pentru gestionarea poziționării implementează interfața **LayoutManager**.
- La instanțierea unui container se creează implicit un gestionar de poziționare asociat acestuia.
- De exemplu, pentru o fereastră **JFrame** gestionarul implicit este de tip **BorderLayout**, în timp ce pentru un panel este de tip **FlowLayout**.
- În raport cu estetica interfeței, gestionarii de poziționare atașați implicit unui container pot fi modificați prin apelul metodei **setLayout** a clasei **Container**, utilizând o referință spre o instanță a unei clase care implementează interfața **LayoutManager**.
- Dacă parametrul metodei este null, atunci nu se va utiliza niciun gestionar de poziționare.

- BorderLayout
- GridLayout
- CardLayout
- GridBagLayout
- SpringLayout
- GroupLayout

În continuare, vom prezenta o parte dintre acești gestionari de poziționare:

Gestionarul de poziționare **FlowLayout**

- Acest gestionar poziționează componentele pe suprafața de afișare una după alta pe linii, în limita spațiului disponibil.
- Dacă o componentă nu mai încape pe linia curentă, atunci acesta se afișează pe următoarea linie, de sus în jos.
- Adăugarea componentelor se realizează de la a stânga la dreapta pe linie, iar alinierea obiectelor în cadrul unei linii poate la stânga, la dreapta sau în centru.
- Implicit, componentele sunt centrate pe fiecare linie, iar distanța implicită dintre componente este de 5 pixeli pe verticală și 5 pe orizontală.

Exemplu:

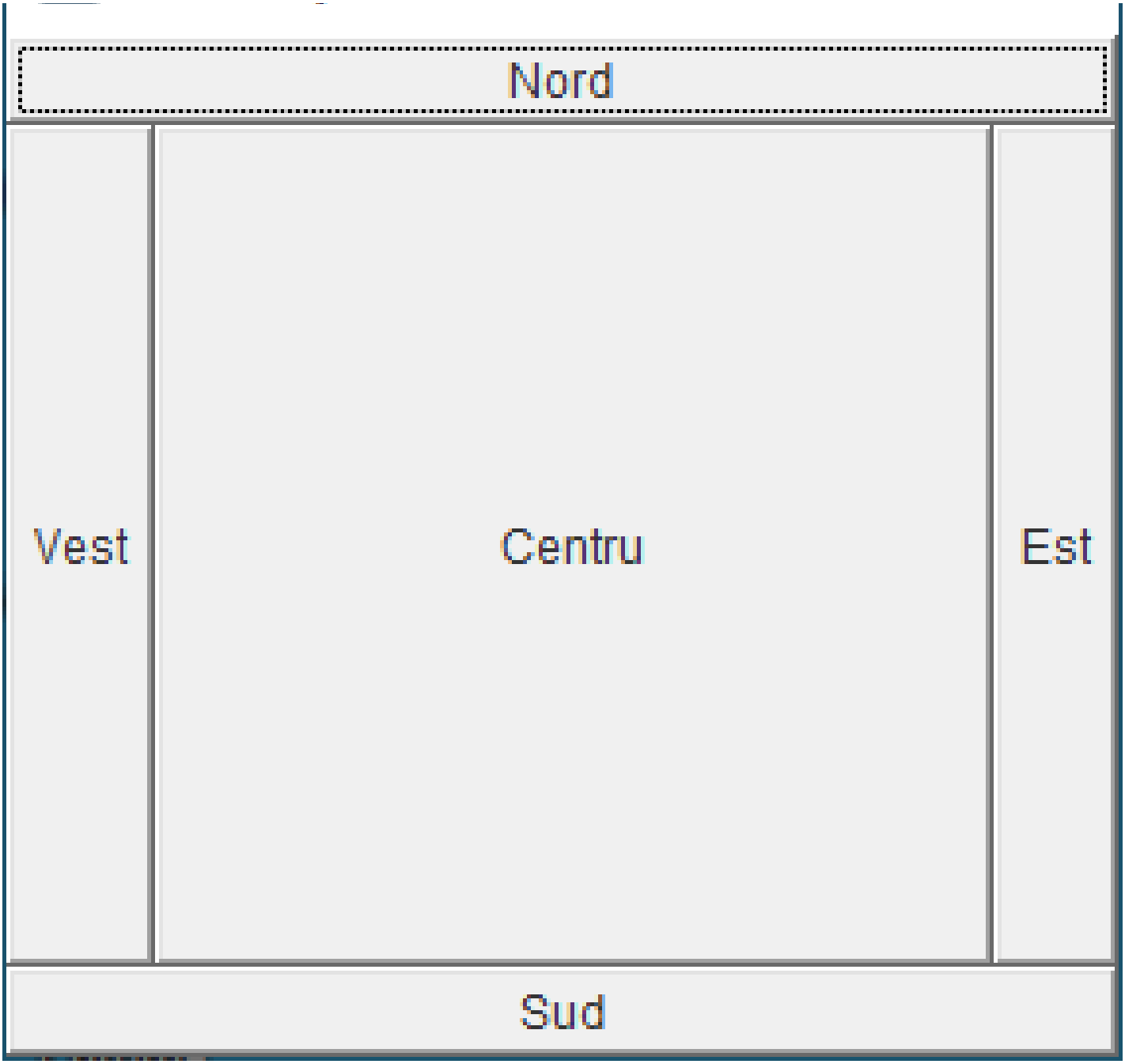
```
JFrame container = new JFrame();
FlowLayout gestionar = new FlowLayout();
container.setLayout(gestionar);
```

Gestionarul de poziționare **BorderLayout**

- Gestionarul împarte suprafața de afișare în cinci zone, corespunzătoare celor patru puncte cardinale și centrului.
- Dimensiunea unei componente grafice plasată într-una din cele 5 zone este calculată astfel încât acesta să ocupe întreg spațiul de afișare oferit de zona respectivă.
- Pentru a adăuga mai multe componente grafice într-o singură zona este necesar ca în prealabil acestea să fie așezate pe un panou care ulterior va fi afișat în zona respectivă.
- Pentru a specifica în care dintre cele 5 zone se plasează o componentă grafică, metoda add primește ca parametru o constantă specifică zonei respective: NORTH, SOUTH, EAST, WEST sau CENTER.

Exemplu:

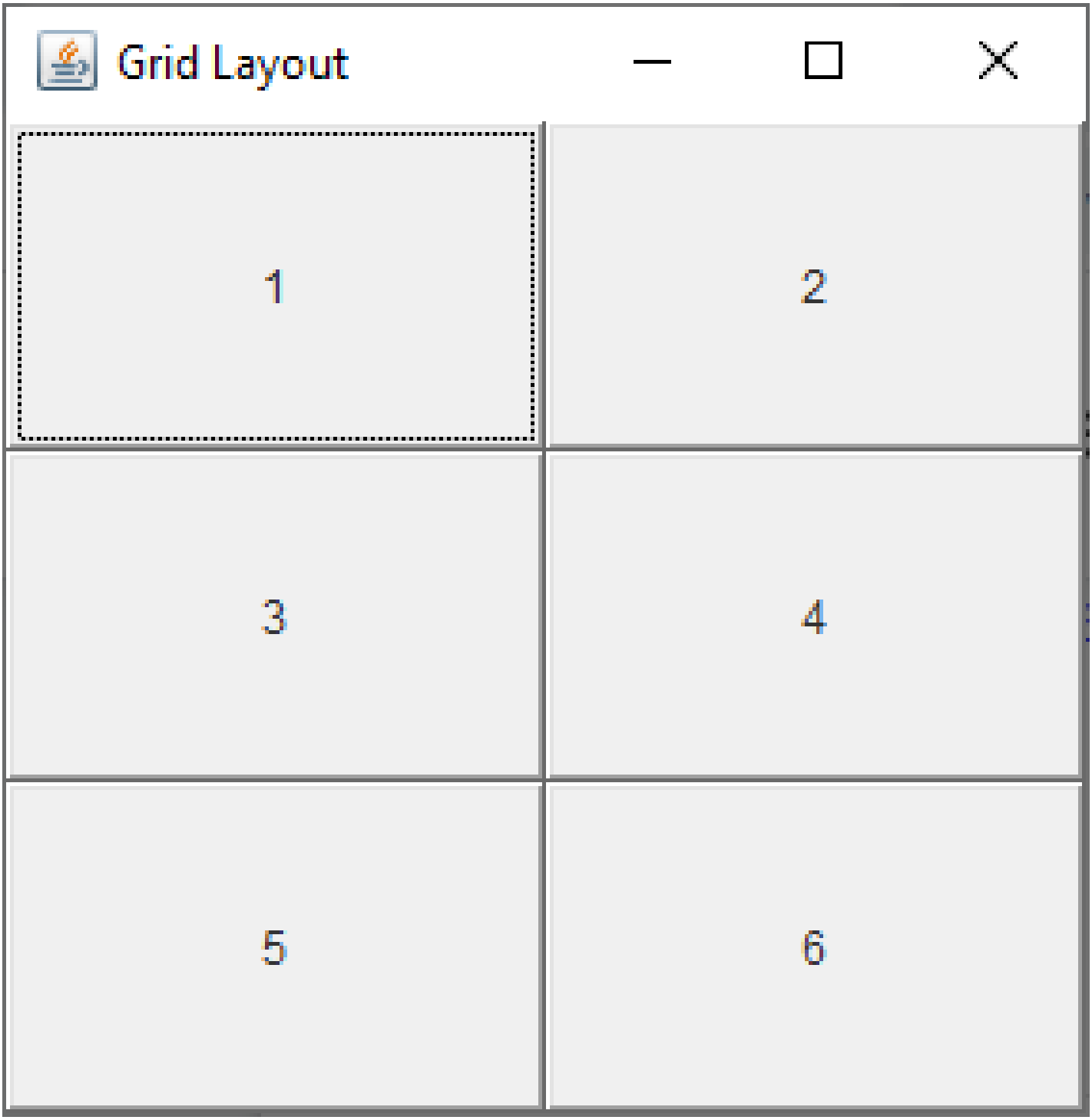
```
JFrame f = new JFrame("Border Layout");
f.add(new Button("Nord"), BorderLayout.NORTH);
f.add(new Button("Sud"), BorderLayout.SOUTH);
f.add(new Button("Est"), BorderLayout.EAST);
f.add(new Button("Vest"), BorderLayout.WEST);
f.add(new Button("Centru"), BorderLayout.CENTER);
f.setSize(new Dimension(300,300));
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
f.setVisible(true);
```



Gestionarul de poziționare **GridLayout**

- Gestionarul **GridLayout** organizează containerul sub forma unui tabel, astfel încât fiecare componentă grafică să fie plastă în celulele tabelului de la stânga la dreapta, începând cu primul rând.
- Celulele tabelului au dimensiuni egale, iar o componentă grafică poate ocupa doar o singură celulă.
- Numărul de linii și cel de coloane pot fi specificate prin constructor sau se pot stabili prin metodele **setRows**, respectiv **setCols**.

```
f.add(new Button("1"));
f.add(new Button("2"));
f.add(new Button("3"));
f.add(new Button("4"));
f.add(new Button("5"));
f.add(new Button("6"));

f.setSize(new Dimension(300, 300));
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
f.setVisible(true);
```



formia unui tabel, însă este mult mai flexibil.

- Practic, numărul de linii și de coloane sunt determinate automat, în funcție de componentele grafice plasate pe suprafața containerului.
- În plus, dimensiunile celulelor pot fi diferite, cu restricția ca pe o linie celulele să aibă aceeași înălțime, iar pe coloane celulele să aibă aceeași lățime.
- De asemenea, o componentă grafică poate să ocupe mai multe celule adiacente.
- Înainte de a adăuga o componentă pe suprafața containerului se specifică un set de constrângeri prin care se stabilește modalitatea de plasare.
- Aceste constrângeri sunt specificate prin intermediul unui obiect de tip **GridBagConstraints**, care poate fi refolosit pentru mai multe componente care au aceleași constrângeri de afișare.

```
GridBagLayout gridBag = new GridBagLayout();
container.setLayout(gridBag);
GridBagConstraints c = new GridBagConstraints();
//Specificam restricțiile referitoare la afișarea componentei
. . .
gridBag.setConstraints(componenta, c);
container.add(componenta);
```

Constrângerile pot fi specificate prin intermediul câmpurilor din clasa **GridBagConstraints**, precum:

- **gridx**, **gridy** – specifică celula în care va fi afișată componentei;
- **gridwidth**, **gridheight** – specifică numărul de celule pe linie și coloană utilizate pentru a afișa componenta;
- **fill** – specifică dacă o componentă va ocupa întreg spațiul dedicat sau nu;
- **insets** – specifică distanțele dintre componentă și marginile suprafeței sale de afișare;
- **weigthx**, **weighty** – folosite pentru distribuția spațiului liber (implicit au valoarea 1).

```
public class Test {
    static JFrame fereastra;
    static GridBagLayout gridBag;
    static GridBagConstraints gbcons;

    static void adauga(Component comp, int x, int y, int w, int h) {
        gbcons.gridx = x;
        gbcons.gridy = y;
        gbcons.gridwidth = w;
        gbcons.gridheight = h;
        gridBag.setConstraints(comp, gbcons);
        fereastra.add(comp);
    }

    public static void main(String args[]) {
        fereastra = new JFrame("Test GridBagLayout");
        gridBag = new GridBagLayout();
        gbcons = new GridBagConstraints();
        gbcons.weightx = 1.0;
        gbcons.weighty = 1.0;

        gbcons.insets = new Insets(5, 5, 5, 5);
        fereastra.setLayout(gridBag);
        JLabel lblLogin = new JLabel("LOGIN", JLabel.CENTER);
        lblLogin.setFont(new Font(" Arial ", Font.BOLD, 24));
        gbcons.fill = GridBagConstraints.BOTH;
        adauga(lblLogin, 0, 0, 4, 2);
        JLabel lblNume = new JLabel("Utilizator:");
        gbcons.fill = GridBagConstraints.NONE;
        gbcons.anchor = GridBagConstraints.EAST;
        adauga(lblNume, 0, 2, 1, 1);
        JLabel lblParola = new JLabel("Parola:");
        adauga(lblParola, 0, 3, 1, 1);
        JTextField txtUtilizator = new JTextField("", 30);
        gbcons.fill = GridBagConstraints.HORIZONTAL;
        gbcons.anchor = GridBagConstraints.CENTER;
        adauga(txtUtilizator, 1, 2, 2, 1);
        JTextField txtParola = new JPasswordField("", 30);
        adauga(txtParola, 1, 3, 2, 1);

        JButton btnSalvare = new JButton(" Salvare ");
        gbcons.fill = GridBagConstraints.HORIZONTAL;
        adauga(btnSalvare, 1, 4, 1, 1);
        JButton btnIesire = new JButton(" Iesire ");
        adauga(btnIesire, 2, 4, 1, 1);

        fereastra.setSize(new Dimension(300, 300));
        fereastra.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        fereastra.setVisible(true);
    }
}
```

8. Tratarea evenimentelor

- Mai sus, noțiunile prezentate au descris modalitatea de a defini, din punct de vedere estetic, o interfața grafică Swing.
- Aceasta este, în esență, separată de partea de implementare a evenimentelor generate în urma interacțiunii cu utilizatorul.
- Un eveniment este produs de o acțiune a utilizatorului asupra unei componente grafice și reprezintă mecanismul prin care utilizatorul poate comunica efectiv cu aplicația.
- Apăsarea unui buton, modificarea textului dintr-un câmp text, selecția unei opțiuni dintr-o listă, închiderea sau redimensionarea unei ferestre, etc. reprezintă evenimente care trebuie să fie tratate printr-un cod Java asociat.
- În acest sens, Swing a fost conceput utilizând o arhitectură bazată pe evenimente (event-driven architecture), care antrenează mai multe concepte, precum interfețe de tip “ascultător”, clase adaptor etc.

- În momentul în care utilizatorul interacționează cu o componentă grafică, se va genera un eveniment modelat printr-un obiect de tip **Event**, derivat din clasa **java.util.EventObject**.
- Prin intermediul acestuia, programatorul manipulează evenimentul, putând determina sursa sa.
- De asemenea, obiectul încapsulează informații despre tipul evenimentului, starea sursei înainte și după acțiune.
- Sursa evenimentului se poate determina prin apelul metodei **Object getSource()**, iar informațiile despre tipul evenimentului se obțin prin apelul metodei public String toString().
- Clasele prin care se modelează un eveniment sunt grupate în pachetele **java.awt.event** și **java.swing.event**.

Exemple de evenimente:

1. **ActionEvent** – eveniment lansat în urma efectuării unei acțiuni asupra unei componente (de exemplu, apăsarea unui buton sau apăsarea unei taste);
2. **MouseEvent** – eveniment lansat în urma efectuării unei acțiuni utilizând mouse-ul (de exemplu, efectuarea unui click asupra unei componente);
3. **WindowEvent** – eveniment lansat în momentul în care o fereastră își schimbă starea (de exemplu, afișarea sau închiderea unei ferestre, redimensionarea sa etc.);
4. **ItemEvent** – eveniment lansat în urma realizării unei operații de selecție (de exemplu, selectarea/deselectarea unui element dintr-o listă sau selectarea/deselectarea unui check-box);
5. **AdjustementEvent** – eveniment lansat în urma modificării valorii asociate unei componente care implementează interfața Adjustable (de exemplu, o bară de defilare de tip JScrollBar).

- Pentru a tratata un eveniment generat în urma interacțiunii cu o componentă grafică, programatorul trebuie să asocieze obiectului de tip eveniment un obiect de tip **ascultător (listener)**, în cadrul căruia să implementeze răspunsul aplicației la apariția evenimentului respectiv.
- Un ascultător este o implementare a unei interfețe TipListener din pachetele **java.awt.event** și **java.swing.event**.

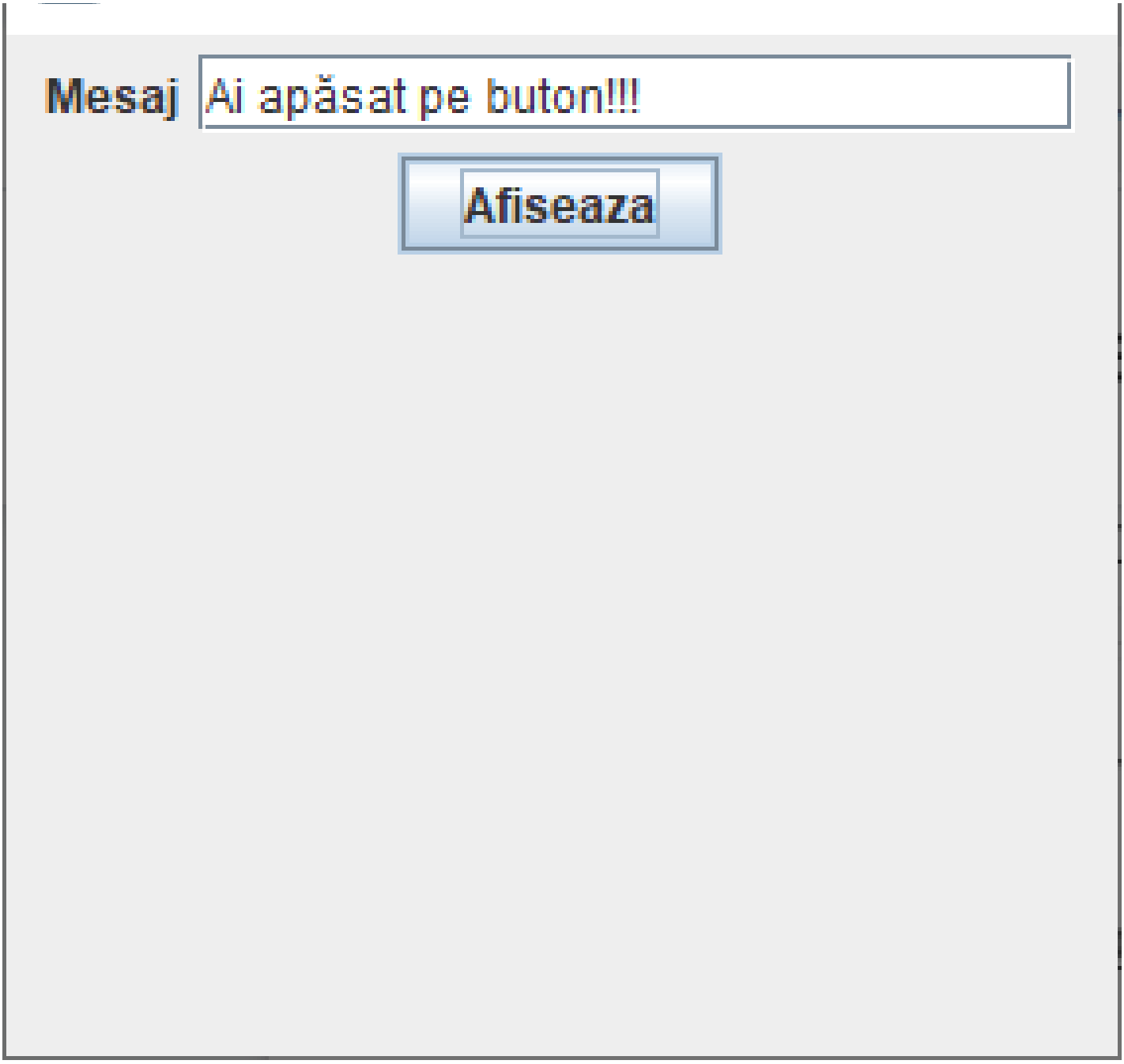
- În concluzie, pentru fiecare componentă grafică este definit un ascultător specific.
- Astfel, pentru ascultarea evenimentelor de tip **ActionEvent** se implementează interfața **ActionListener**, pentru **TextEvent** interfața **TextListener** etc.
- Fiecare interfață definește una sau mai multe metode care primesc ca argument un obiect de TipEvent și sunt apelate implicit la apariția unui eveniment.

- O listă a tuturor obiectelor de tip ascultător din Swing se găsește la pagina: https://docs.oracle.com/javase/tutorial/uiswing/events/eventsandcomponents.html

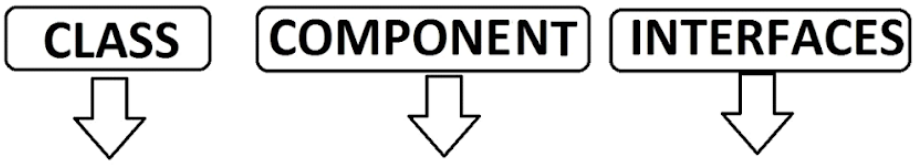
```
    }
}

class AscultaTexte implements TextListener {
    public void textValueChanged(TextEvent e) {
        // Metoda interfetei TextListener ...
    }
}
```

Exemplu:

```
JFrame f = new JFrame("Test");
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
f.setSize(new Dimension(300, 300));

JPanel panou = new JPanel();
panou.setLayout(new FlowLayout());

JLabel eticheta = new JLabel("Mesaj");
JButton buton = new JButton("Afiseaza");
JTextField text = new JTextField(20);

panou.add(eticheta);
panou.add(text);
panou.add(buton);

buton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        text.setText("Ai apăsat pe buton!!!");
    }
});

f.getContentPane().add(panou);
f.setVisible(true);
```



- O interfață corespunzătoare unui ascultător poate să conțină un număr mare de metode abstracte care trebuie să fie implementate de către clasa care tratează evenimentul respectiv.
- De exemplu, pentru o aplicație se dorește doar tratarea unui eveniment generat de închiderea ferestrei, prin implementarea metodei abstracte windowClosing din interfața **WindowListener**, însă această interfața conține alte 6 metode abstracte care ar trebui să fie implementate.
- Pentru a evita implementarea inutilă a tuturor metodelor abstracte dintr-o interfață, au fost definite clasele adaptor în care sunt implementate minimal toate metodele abstracte dintr-o interfață.
- Practic, se extinde clasa adaptor corespunzătoare interfeței și se redefinesc doar metodele necesare în aplicația respectivă.
- De exemplu, pentru interfața **WindowListener** este definită clasa adaptor **WindowAdapter**.

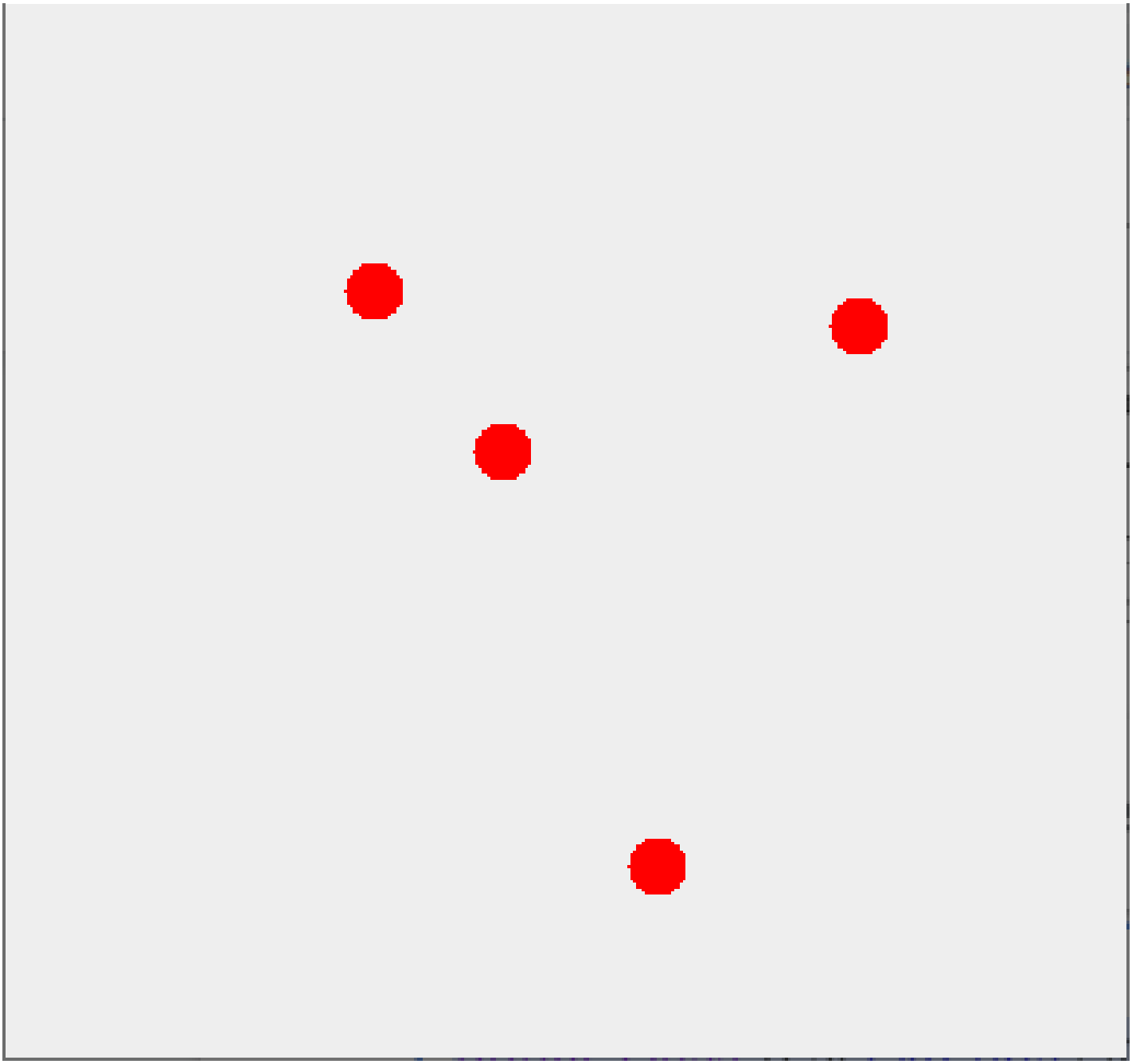


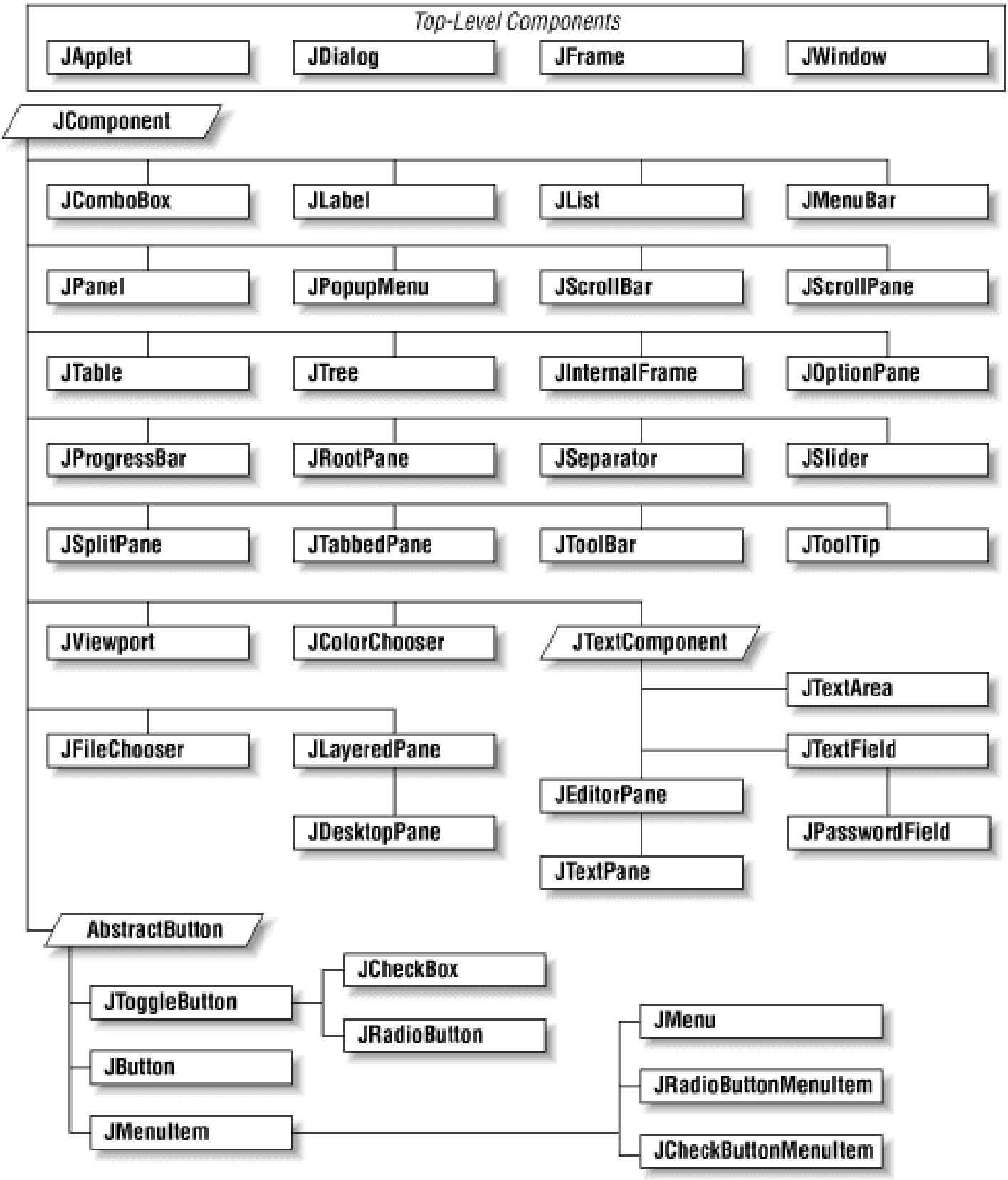
EVENTS	SOURCE	LISTENERS	ADAPTER CLASS
Action Event	Button, List,MenuItem,Text field	ActionListener	None
Component Event	Component	Component Listener	None
Focus Event	Component	FocusListener	FocusAdapter
Item Event	Checkbox,CheckboxMenuItem, Choice, List	ItemListener	None
Key Event	when input is received from keyboard	KeyListener	KeyAdapter
Text Event	Text Component	TextListener	None
Window Event	Window	WindowListener	WindowAdapter
Mouse Event	Mouse related event	MouseListener	MouseAdapter

Se poate observa faptul că nu este definită câte o clasă adaptor pentru fiecare interfață de tip ascultător, ci doar pentru acele interfețe care conțin un set mare de metode abstracte.

Exemplu:

```
public class Test extends MouseAdapter {
    JFrame f;

    Test() {
        f = new JFrame("Test MouseAdapter");
        f.addMouseListener(this);
        f.setSize(400, 400);
        f.setVisible(true);
    }

    public void mouseClicked(MouseEvent e) {
        Graphics g = f.getGraphics();
        g.setColor(Color.RED);
        g.fillOval(e.getX(), e.getY(), 20, 20);
    }

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





Informații detaliate și exemple despre toate componentele grafice Swing se găsesc în pagina https://docs.oracle.com/javase/tutorial/uiswing/components/index.html