Programare orientată pe obiecte

1. Interfeţe utilizator grafice (GUIs)

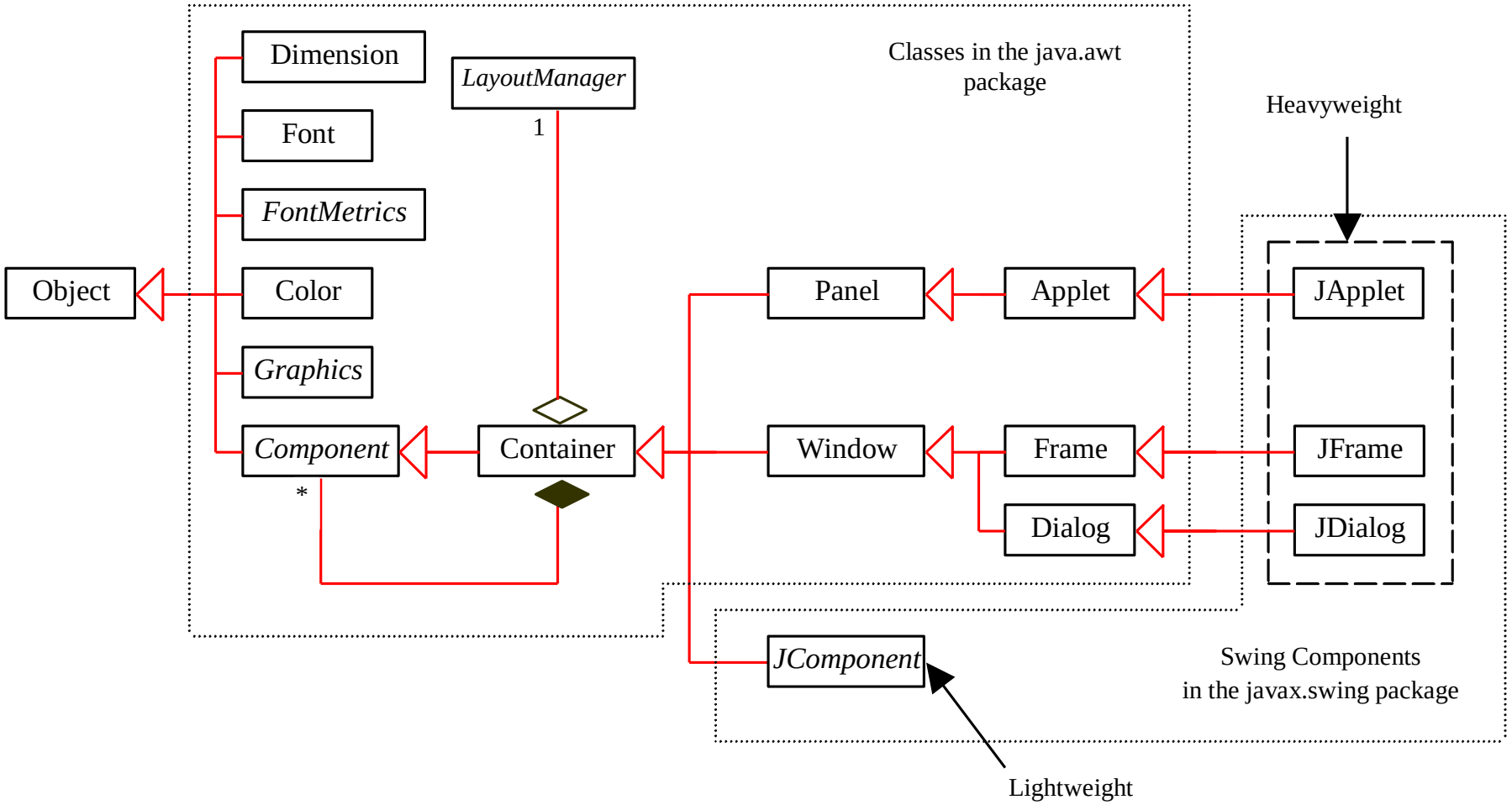
GUI

- O interfaţă utilizator grafică - Graphical User Interface (GUI) prezintă un mecanism prietenos pentru interacţiunea utilizatorului cu un program
  - GUI dă programului un aspect (*look*) şi un mod în care este "simţit" (*feel*) caracteristic
  - Permite utilizatorilor să se simtă mai familiarizaţi cu programul chiar înainte de a-l fi utilizat
  - Reduce timpul de învăţare a modului de utilizare

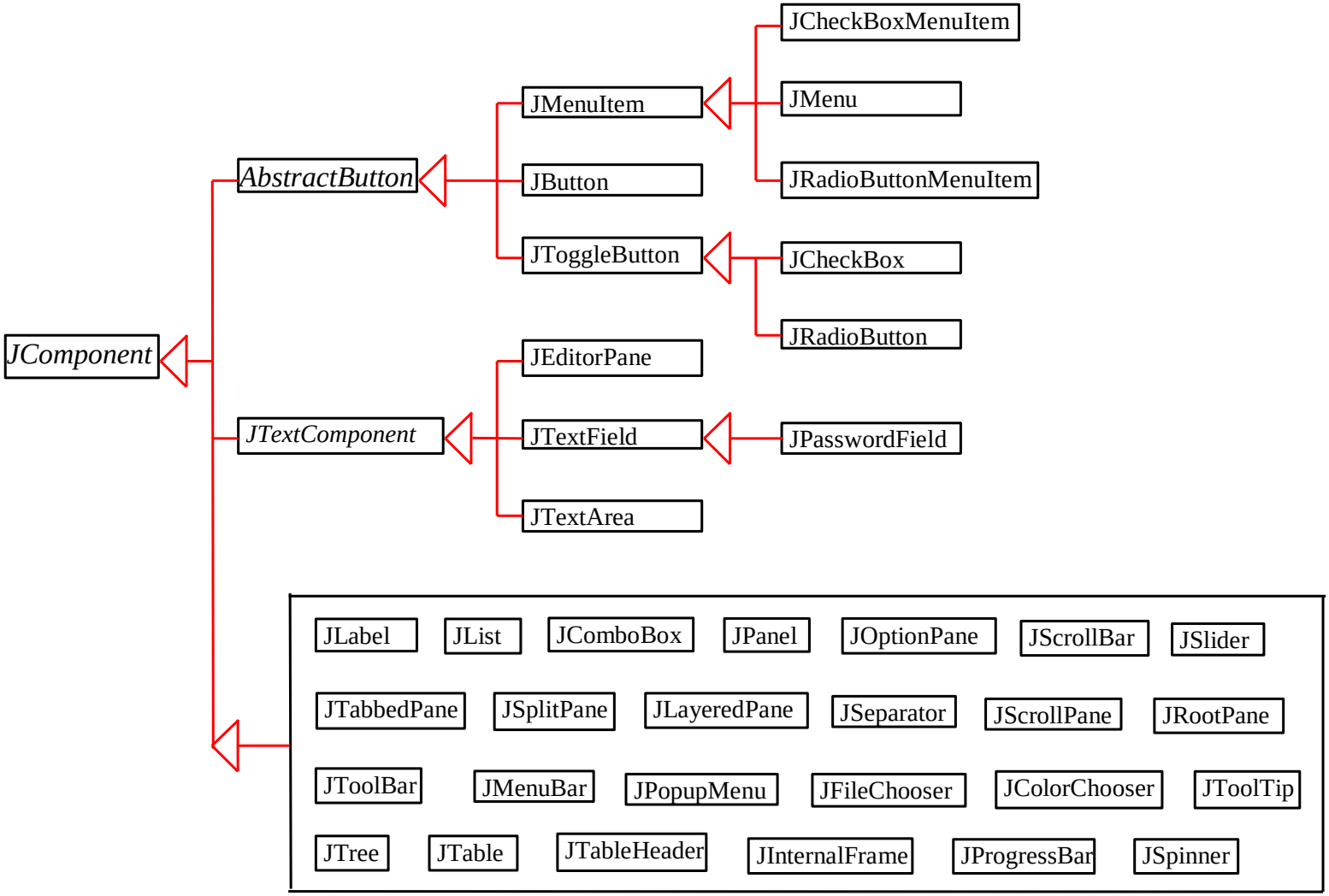
Pachete GUI

- Pachetele responsabile pentru dezvoltarea de interfețe cu utilizatorul:
  - **AWT** (Abstract Windowing Toolkit):
    - Scopul original de a permite utilizatorului să dezvolte GUI care să arate bine orice platformă, dar acest scop nu a fost atins
    - Alte limitări:
      - nu poate accesa toate elementele de GUI (cele mai specializate) din sistemul de operare
      - modelul de programare Java 1.0 nu este orientat pe obiecte
      - poate folosi doar 4 fonturi
  - Situația s-a îmbunătățit începând cu **Java 1.1 AWT event model**, care este mult mai clar și este orientat pe obiecte
  - **Swing:**
    - Java 2 (JDK 1.2) a finalizat îmbunătățirile pt Java 1.0 AWT prin înlocuirea cu *Java Foundation Classes* (JFC), primind noul nume de „Swing"
    - Swing este considerată versiunea finală a librăriilor de GUI în Java

GUI - Ierarhia de clase (Swing)



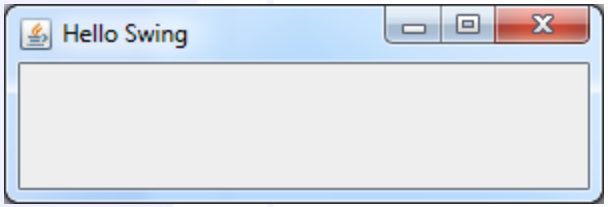
GUI - Componentele Swing



Exemplu: Crearea unei ferestre

- Majoritatea aplicațiilor GUI se construiesc în interiorul unei ferestre

```
// : gui/HelloSwing.java
import javax.swing.*;
public class HelloSwing {
  public static void main(String[] args) {
          // crearea ferestrei
          JFrame frame = new JFrame("Hello Swing");
          // setarea operației implicite de închidere a ferestrei
          // atunci cand utilizatorul dă click pe X-ul dreapta sus
          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          // setarea dimensiunii ferestrei
          frame.setSize(300, 100);
          // setarea vizibilității ferestrei
          frame.setVisible(true);
  }
}
```



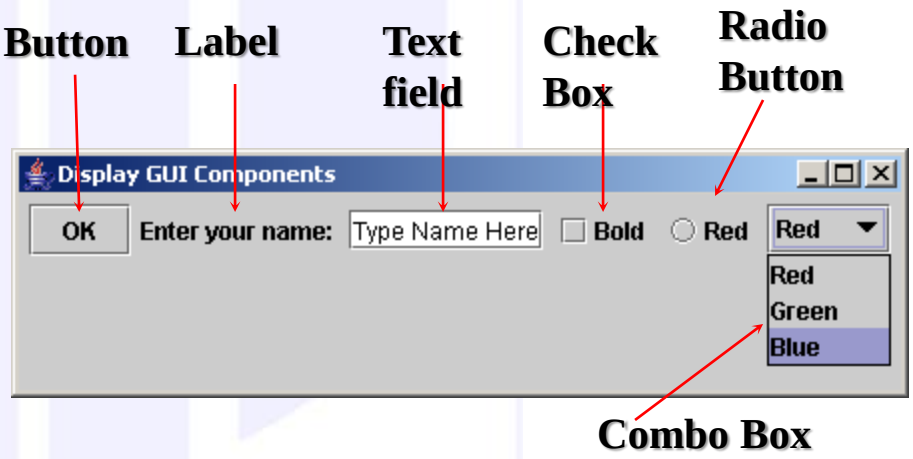
Exemplu: Crearea de obiecte GUI

```
JFrame frame  = new JFrame("Display GUI Components");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setSize(500,160);

JPanel panel = new JPanel();

//Create a button with text OK
JButton jbtOK = new JButton("OK");
panel.add(jbtOK);
//Create a label with text "Enter your name: "
JLabel jlblName = new JLabel("Enter your name: ");
panel.add(jlblName);
//Create a text field with text "Type Name Here"
JTextField jtfName = new JTextField("Type Name Here");
panel.add(jtfName);
//Create a check box with text bold
JCheckBox jchkBold = new JCheckBox("Bold");
panel.add(jchkBold);
//Create a radio button with text red
JRadioButton jrbRed = new JRadioButton("Red");
panel.add(jrbRed);
//Create a combo box with choices red, green, and blue
JComboBox jcboColor = new JComboBox(new String[]{"Red", "Green", "Blue"});
panel.add(jcboColor);

frame.setContentPane(panel);
frame.setVisible(true);
```



Containere şi componente

- Clasa **`Container`** gestionează o colecţie de componente înrudite
  - În aplicaţii care folosesc **`JFrame`** şi în applet-uri ataşăm componente panoului de conţinut (*content pane*) – care este un container
  - Metode importante: **`add()`**, **`setLayout()`**
- Clasa **`Component`** declară atributele şi comportamentele comune tuturor subclaselor sale
  - Metode importante: **`paint()`**, **`repaint()`**

Clasa `Container`

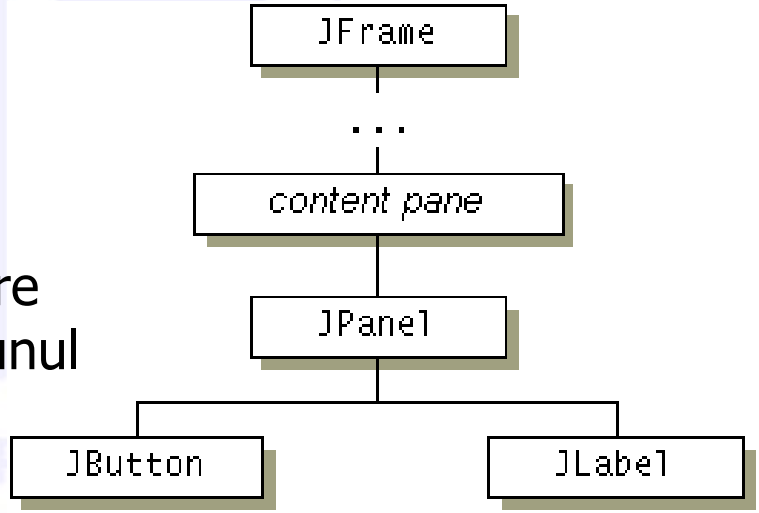
- Orice clasă care descinde din clasa **`Container`** este considerată o clasă container
  - Clasa **`Container`** se află în pachetul **`java.awt`**, nu în Swing
- Oricărui obiect care aparţine unei clase derivate din clasa **`Container`** (sau din descendenţii săi) i se pot adăuga componente
- Clasele **`JFrame`** şi **`JPanel`** sunt descendente din clasa **`Container`**
  - De aceea ele şi orice alţi descendenţi ai lor pot servi pe post de container

Clasa `JComponent`

- Orice descendent al clasei **`JComponent`** se numeşte *clasă componentă*
- Oricare obiect **`JComponent`** sau *component* poate fi adăugat la orice obiect de clasă container
- Deoarece este derivată din clasa **`Container`**, o **`JComponent`** poate fi adăugată şi la alt(ă) **`JComponent`**

Ierarhii de containere

- Ierarhizarea containerelor și componentelor
  - Containere de nivel înalt
  - Containere intermediare
  - Componente atomice
- Containere de nivel înalt
  - La rădăcina fiecărei ierarhii de conţinere
  - Toate programele Swing au cel puţin unul
  - Panouri de conţinut
  - Tipuri de containere de nivel înalt
    - Ferestrele (*frames*)
    - Dialoguri
    - Applet-uri

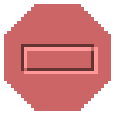


Dialoguri

- Mai limitate decât ferestrele
- Modalitate
  - Dialogurile modale opresc temporar execuţia programului – utilizatorul nu poate continua până când nu s-a închis dialogul
- Tipuri de dialoguri
  - `JOptionPane`
  - `ProgressMonitor`
  - `JColorChooser`
  - `JDialog`

Afişarea dialogurilor

- JOptionPane.showXYZDialog(...)
  - Dialoguri de opţiuni şi de mesaje

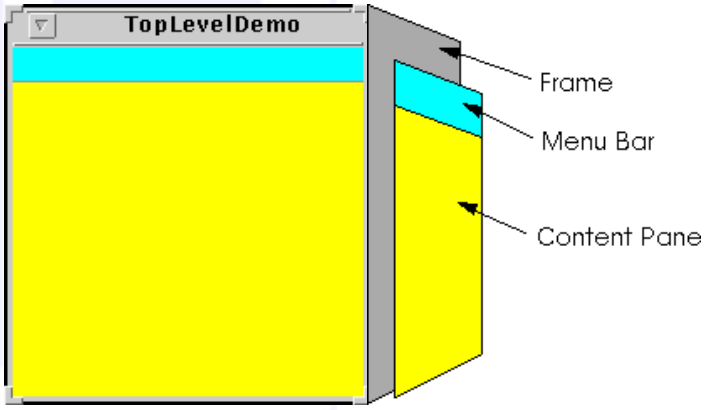


  - `JOptionPane.showMessageDialog(frame, "Error!", "An error message", JOptionPane.ERROR_MESSAGE);`

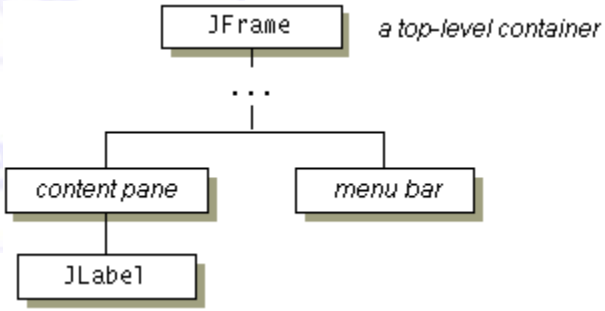
  - `JOptionPane.showOptionDialog(frame, "Save?", "A save dialog", JOptionPane.YES_NO_CANCEL_OPTION);`
  - Intrare, confirmare
- Individualizare (*customize*)
  - showOptionDialog – destul de individualizabil
  - JDialog - total individualizabil

Panouri de conţinut

- Folosesc de obicei un **`JPanel`**
- Conţine totul cu excepţia barei de meniu pentru majoritatea aplicaţiilor Swing
- Poate fi creat explicit sau implicit




```
//Create a panel and add components to it.
JPanel contentPane = new JPanel();
contentPane.add(someComponent);
contentPane.add(anotherComponet);
//Make it the content pane.
contentPane.setOpaque(true);
topLevelContainer.setContentPane(contentPane);
```



Obiecte dintr-un GUI tipic

- Aproape fiecare GUI construit folosind clasele container din Swing va fi compus din până la trei feluri de obiecte
  1. **Containerul însuşi**, probabil un obiect panou (*panel*) sau de tip fereastră (*window-like*)
  2. **Componentele adăugate containerului**, cum sunt etichetele (*label*), butoanele şi panourile
  3. **Un gestionar de aranjare** (*layout manager*) pentru a poziţiona componentele în interiorul containerului

Gestiunea aranjării

- Până acum am folosit un control limitat asupra aranjării (**layout**) componentelor
  - Când am folosit un panou, acesta a aranjat implicit componentele de la stânga la dreapta
- Componentele din interfaţa utilizator sunt aranjate prin plasarea lor în containere
- Fiecare container are un *gestionar de aranjare* (***layout manager***) care dirijează aranjarea componentelor sale
- Câteva gestionare de aranjare utile:
  - **border layout, flow layout, grid layout, box layout**
- Gestionarul implicit este **flow layout**
- Se pot seta alte gestionare de aranjare

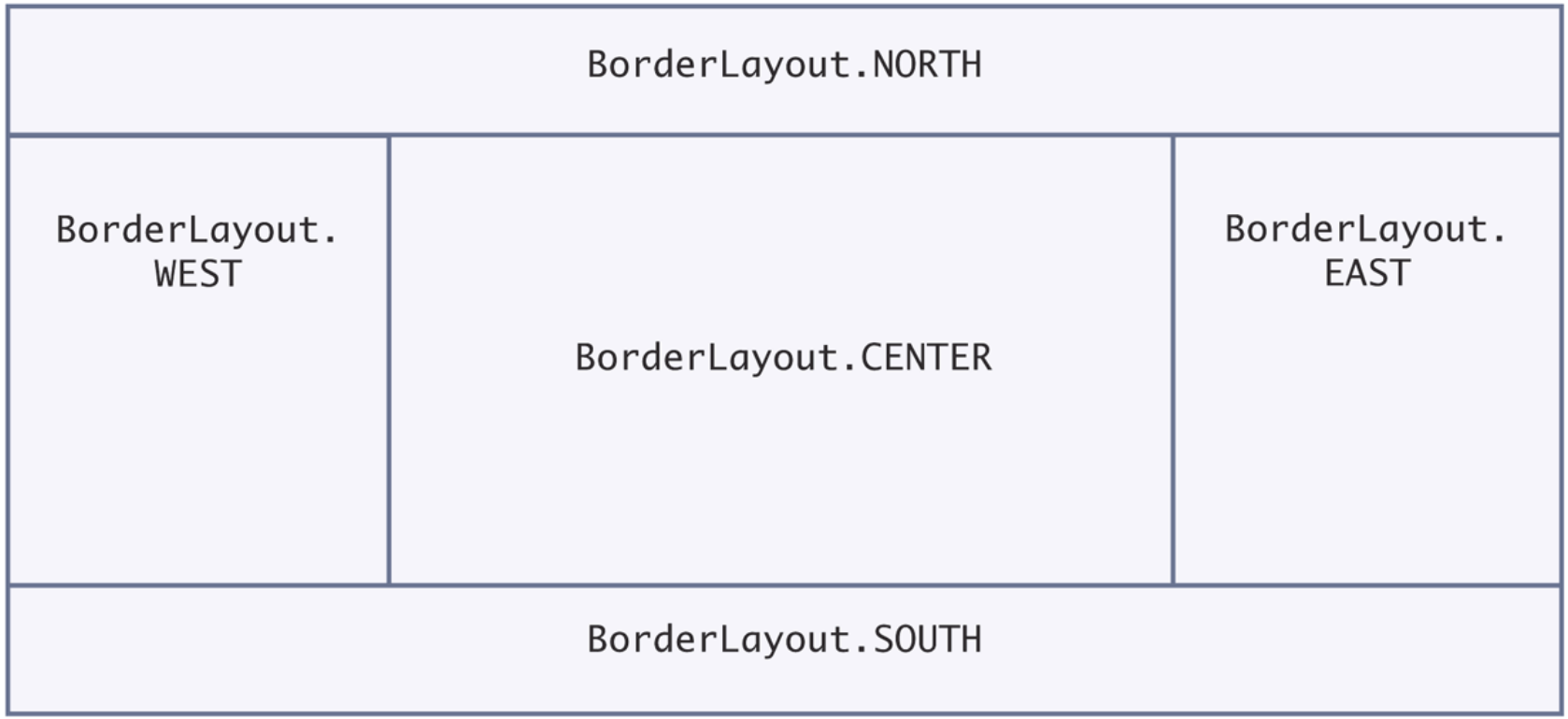
```
panel.setLayout(new BorderLayout());
```

Gestiunea aranjării

- Pasul 1: Facem o schiţă a modului de aranjare dorit
- Pasul 2: Determinăm grupări de componente adiacente cu acelaşi mod de aranjare (*layout*)
- Pasul 3: Identificăm modul de aranjare pentru fiecare grup
- Pasul 4: Grupăm împreună grupurile
- Pasul 5: Scriem codul pentru generarea aranjamentului

Border Layout

- Aranjarea după margini (*border layout*) grupează în cinci zone: centru, nord, vest, sud şi est
  - Componentele se extind ca să umple spaţiul în această aranjare



Border Layout

- Este gestionarul de aranjare implicit pentru ferestre `JFrame` (tehnic, pentru panoul de conţinut al ferestrei)
- La adăugarea unei componente se specifică poziţia astfel:
  `panel.add(component, BorderLayout.NORTH);`
- Extinde fiecare componentă pentru a umple toată zona alocată
- Dacă nu doriţi aceasta, atunci puneţi fiecare componentă într-un panou

Gestionarul de aranjare `FlowLayout`

- Gestionarul de aranjare **`FlowLayout`** aranjează componentele în ordine de la stânga la dreapta şi de sus în jos în container
- Constructori:

```
public FlowLayout();
public FlowLayout(int align);
public FlowLayout(int align, int horizontalGap,
                             int verticalGap);
```

- Alinierea poate fi **`LEFT`**, **`RIGHT`**, sau **`CENTER`**
- Este implicit pentru **`JPanel`**

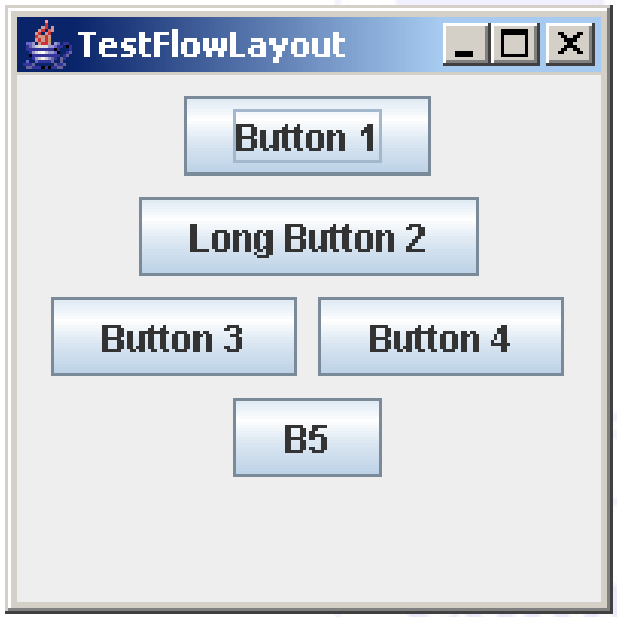
Gestionarul de aranjare `GridLayout`

- Aranjează componentele într-o grilă cu număr fix de rânduri şi coloane
- Redimensionează fiecare componentă astfel încât ele să aibă toate aceeaşi mărime
- Extinde fiecare componentă pentru a umple toată zona alocată lui
- Adăugarea de componente, rând cu rând, de la stânga la dreapta:

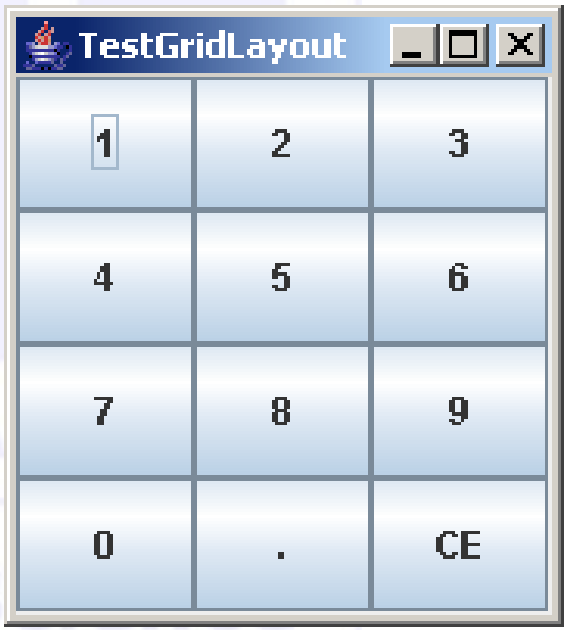
```
JPanel numberPanel = new JPanel();
numberPanel.setLayout(new GridLayout(4, 3));
numberPanel.add(button1);
numberPanel.add(button2);
numberPanel.add(button3);
numberPanel.add(button4);
. . .
```

Exemple: `FlowLayout` şi `GridLayout`

FlowLayout



GridLayout



Gestionarul de aranjare `BoxLayout`

- Gestionarul de aranjare **`BoxLayout`** aranjează componentele dintr-un container într-un singur rând sau o singură coloană
- Spaţierea şi alinierea pe fiecare rând sau coloană poate fi controlată individual
- Containerele care folosesc **`BoxLayout`** pot fi imbricate unul în altul pentru a produce aranjamente complexe
- Constructor:

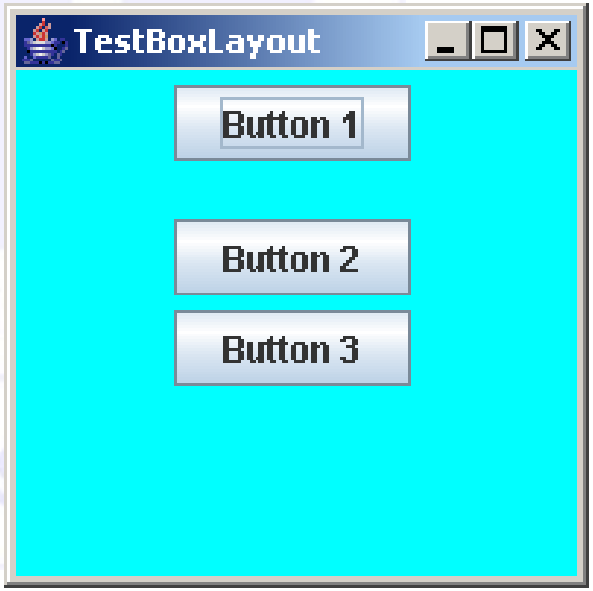
```
public BoxLayout(Container c, int direction);
```

- **`direction`** poate fi **`X_AXIS`** sau **`Y_AXIS`**
- Se pot folosi zone rigide (*rigid areas*) şi zone "lipicioase" (*glue regions*) pentru a spaţia componentele într-un **`BoxLayout`**

Exemplu: Crearea unui `BoxLayout`

```
JFrame jf = new JFrame("TestBoxLayout");
jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
jf.setSize(new Dimension( 200, 200));
jf.setLocation(300, 300);
// Create a new panel
JPanel p = new JPanel();
// Set the layout manager
p.setLayout(new BoxLayout(p, BoxLayout.Y_AXIS));
// Add buttons
// leave some vertical space before button
p.add( Box.createRigidArea(new Dimension(0,5)) );
addAButton( "Button 1", p );
// vertical space between buttons
p.add( Box.createRigidArea(new Dimension(0,20)) );
addAButton( "Button 2", p );
p.add( Box.createRigidArea(new Dimension(0,5)) );
addAButton( "Button 3", p );
p.setBackground(Color.cyan);
// Add the new panel to the existing container
jf.add( p );
jf.setVisible(true);
```

```
private static void addAButton(String text,
                               Container container)
{
  JButton button = new JButton(text);
  button.setAlignmentX(Component.CENTER_ALIGNMENT);
  container.add(button);
}
```



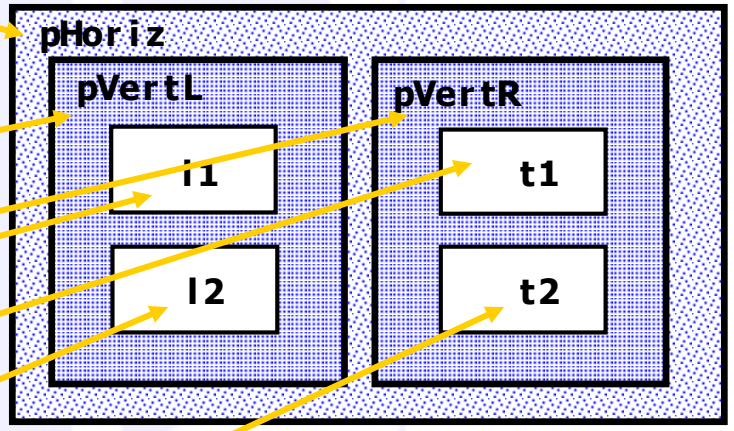
Combinarea gestionarilor de aranjare

- Câteodată e util să creăm mai multe containere unul în altul, fiecare cu propriul gestionar de aranjare
- Spre exemplu, panoul de nivelul cel mai înalt ar putea folosi o aranjare de tipul cutie orizontală, iar în el ar putea fi două sau mai multe panouri cu aranjarea tip cutie verticală
- Rezultatul este controlul complet al spaţierii pe *ambele* dimensiuni

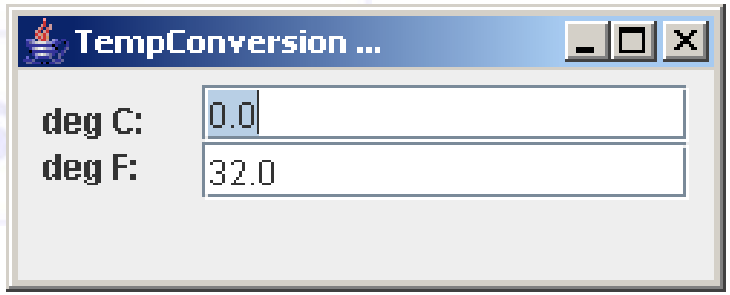
Exemple: Containere şi aranjări imbricate

```
// Creeaza un nou panou de nivel sus
JPanel pHoriz = new JPanel();
pHoriz.setLayout(new BoxLayout(pHoriz,
BoxLayout.X_AXIS));
add( pHoriz );
// Creeaza doua panouri subordonate
JPanel pVertL = new JPanel();
JPanel pVertR = new JPanel();
pVertL.setLayout(new BoxLayout(pVertL,
BoxLayout.Y_AXIS));
pVertR.setLayout(new BoxLayout(pVertR,
BoxLayout.Y_AXIS));
// Adauga la pHoriz cu spatiu orizontal
// intre panouri
pHoriz.add( pVertL );
pHoriz.add( Box.createRigidArea(new
Dimension(20,0)) );
pHoriz.add( pVertR );
// Creeaza campul grade Celsius
l1 = new JLabel("deg C:", JLabel.RIGHT);
pVertL.add( l1 );
t1 = new JTextField("0.0",15);
t1.addActionListener( cHnd );
pVertR.add( t1 );
// Creeaza campul grade Fahrenheit
l2 = new JLabel("deg F:", JLabel.RIGHT);
pVertL.add( l2 );
t2 = new JTextField("32.0",15);
t2.addActionListener( fHnd );
pVertR.add( t2 );
```

Structura:



Rezultatul:



Controale pentru alegeri

- Butoane radio
- Cutiuţe de marcare
- Cutii combo

Butoane radio

- Pentru seturi de mici dimensiuni de variante mutual exclusive folosim butoane radio sau o cutie combo
- Într-un set de butoane radio, doar unul poate fi selectat la un moment dat
- Dacă este selectat un alt buton, cel selectat anterior este automat de-selectat

Butoane radio

- Gruparea butoanelor nu pune butoanele apropiate unul de altul pe container
- Trebuie să le aranjăm noi pe ecran
- `isSelected()`: se apelează pentru a afla dacă un anumit buton este curent selectat sau nu

```
if (largeButton.isSelected()) size = LARGE_SIZE;
```

- Apelăm `setSelected(true)` pe un buton radio din grup înainte de a face vizibil cadrul care conţine butoanele

Căsuțe de bifare (JCheckBox)

- Au două stări: marcat (*checked*) şi nemarcat
- Pentru o alegere din două variante posibile folosim o căsuță de bifare (*checkbox*)
- Folosim un grup de căsuțe de bifare atunci când o alegere nu exclude o alta
- Exemplu: "bold" şi "italic" la alegerea stilului unui font
- Construirea căsuțelor de bifare:

```
JCheckBox italicCheckBox = new JCheckBox("Italic");
```

Căsuțe Combo (JComboBox)

- Pentru un număr mare de opţiuni, folosim o casuță combo (*combo box*)
  - Foloseşte mai puţin spaţiu decât butoanele radio
- "Combo": combinaţie de listă cu câmp text
  - Câmpul text afișează numele selecţiei curente

- Dacă căsuța combo este editabilă, atunci utilizatorul poate să-şi tasteze propria selecţie
  - Folosim metoda `setEditable()`

Căsuțe Combo (JComboBox)

- Textele alegerilor le adăugăm folosind metoda **`addItem()`**:

```
JComboBox facenameCombo = new JComboBox();
facenameCombo.addItem("Serif");
facenameCombo.addItem("SansSerif");
. . .
```

- Obţinem alegerea utilizatorului cu **`getSelectedItem()`** (tipul returnat de aceasta este **`Object`**)

```
String selectedString =
   (String) facenameCombo.getSelectedItem();
```

- Selectăm un element cu **`setSelectedItem()`**

Margini

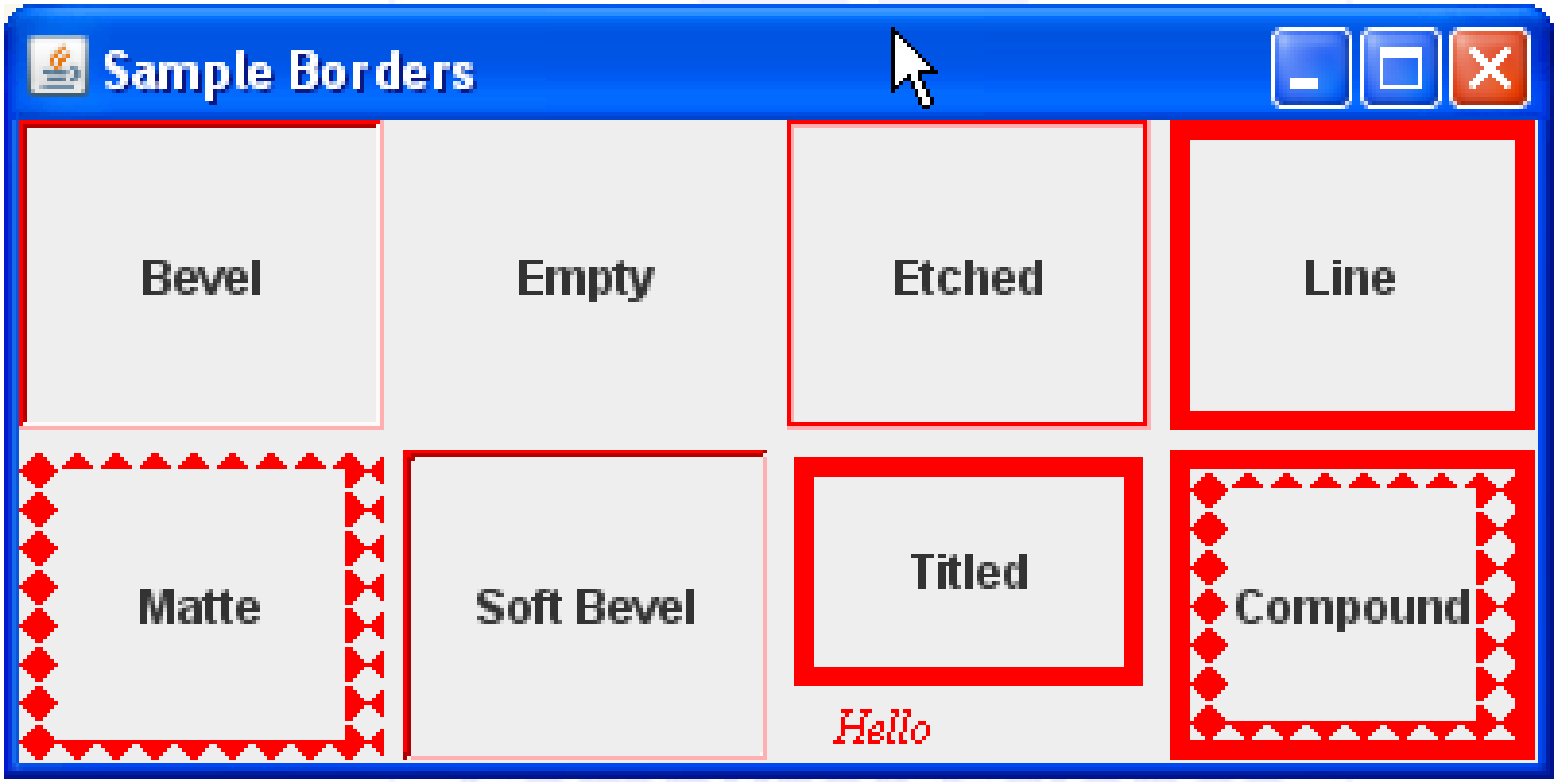
- Punem o margine în jurul panoului pentru a grupa vizual conţinutul său
- **`EtchedBorder`**: efect tridimensional de gravare
- Se poate adăuga margine la oricare componentă, dar cel mai adesea se face pentru panouri:

```
JPanel panel = new JPanel ();
panel.setBorder(new EtchedBorder ());
```

- **`TitledBorder`**: o margine cu titlu:

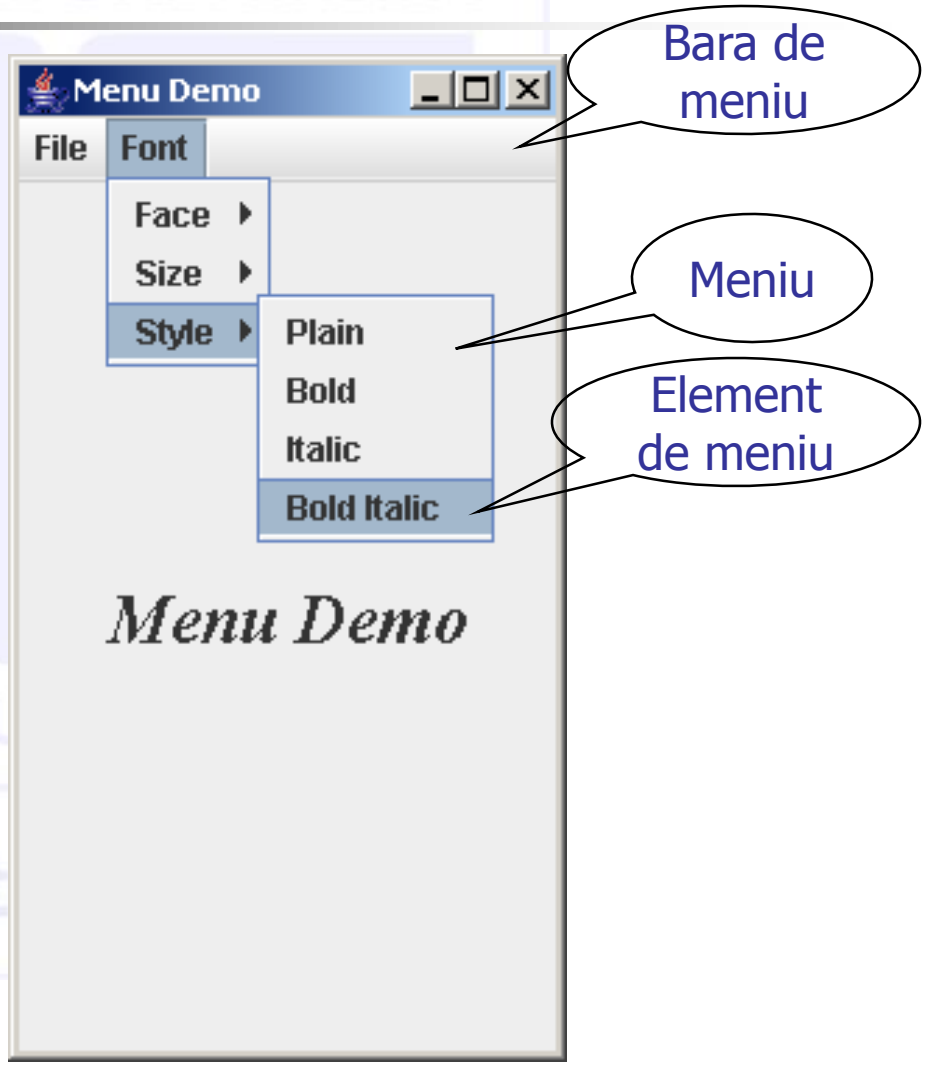
```
panel.setBorder(new TitledBorder(new EtchedBorder(),
                                 "Size"));
```

Margini (Swing)

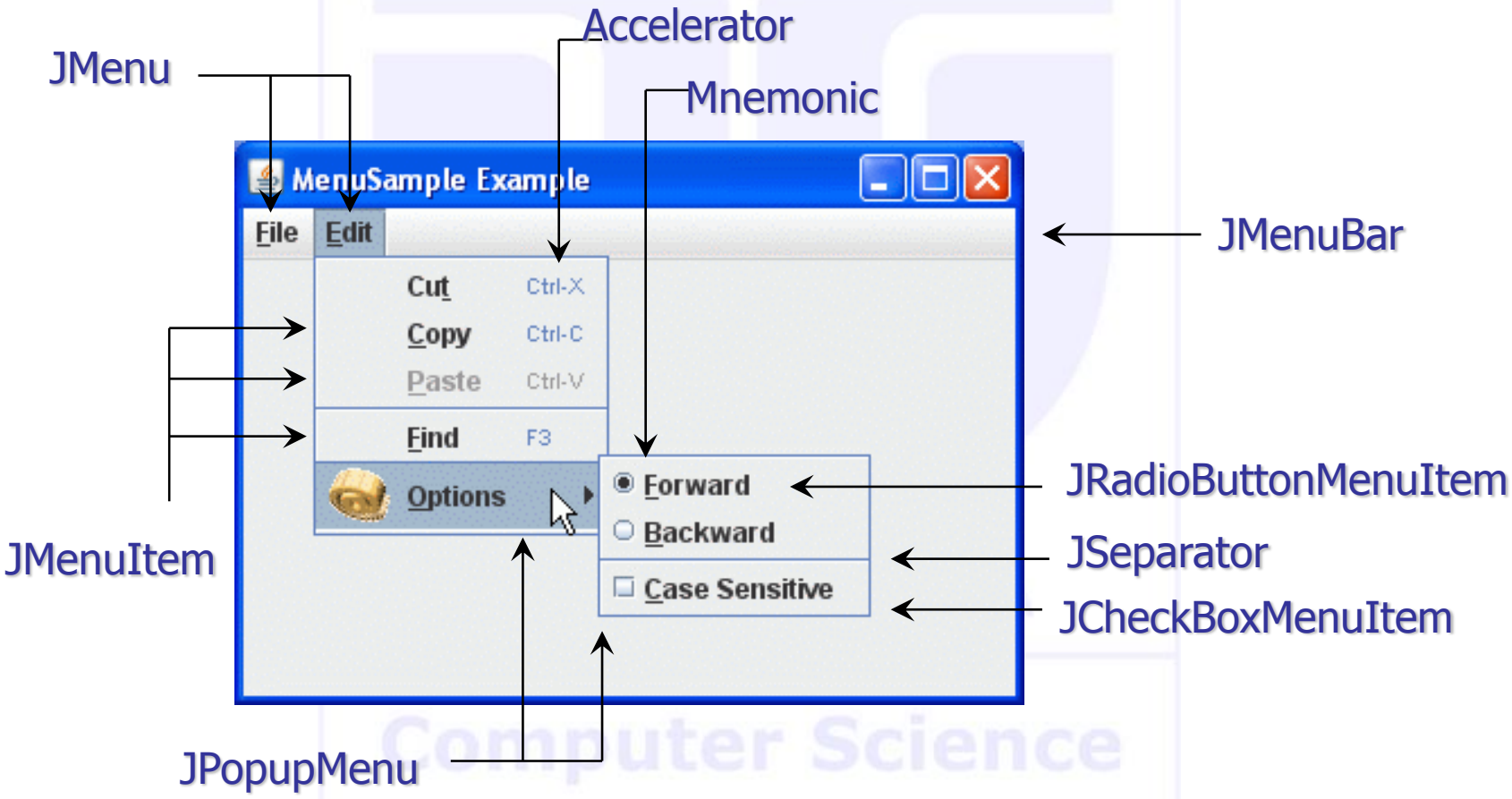


Meniuri

- Fereastra conţine o bară de meniu
- Bara de meniu conţine meniuri
- Meniul conţine submeniuri şi elemente de meniu
  - Meniuri *pull-down*



Meniuri (Swing)



Elemente (*items*) de meniu

- Adăugăm elemente la meniu şi la submeniuri cu metoda **`add()`**:

```
JMenuItem fileExitItem = new JMenuItem("Exit");
fileMenu.add(fileExitItem);
```

- Un element de meniu nu mai are alte submeniuri
- Elementele de meniu generează evenimente acţiune
- Adăugăm câte un ascultător fiecărui element de meniu:

```
fileExitItem.addActionListener(listener);
```

- Adăugăm ascultători de acţiuni doar elementelor de meniu nu şi meniurilor şi barelor de meniu

Zone de text

- Folosim **`JTextArea`** pentru a prezenta mai multe linii de text
- Putem preciza numărul de rânduri şi coloane:

```
final int ROWS = 10;
final int COLUMNS = 30;
JTextArea textArea = new JTextArea(ROWS, COLUMNS);
```

- Numărul de caractere pe linie pentru un obiect **`JTextField`** sau **`JTextArea`** este numărul de spaţii *em*
- Un spaţiu *em* este spaţiul necesar cuprinderii unei litere majuscule **`M`** (cea mai lată din alfabet)
  - O linie pentru 20 **`M`** va fi aproape întotdeauna capabilă să conţină mai mult de 20 caractere

Zone de text

- **`setText()`**: pentru a seta textul unui câmp sau unei zone de text
- **`append()`**: pentru a adăuga text la sfârşitul unei zone de text
- Folosim caractere **`newline`** pentru a separa liniile:

```
textArea.append(account.getBalance() + "\n");
```

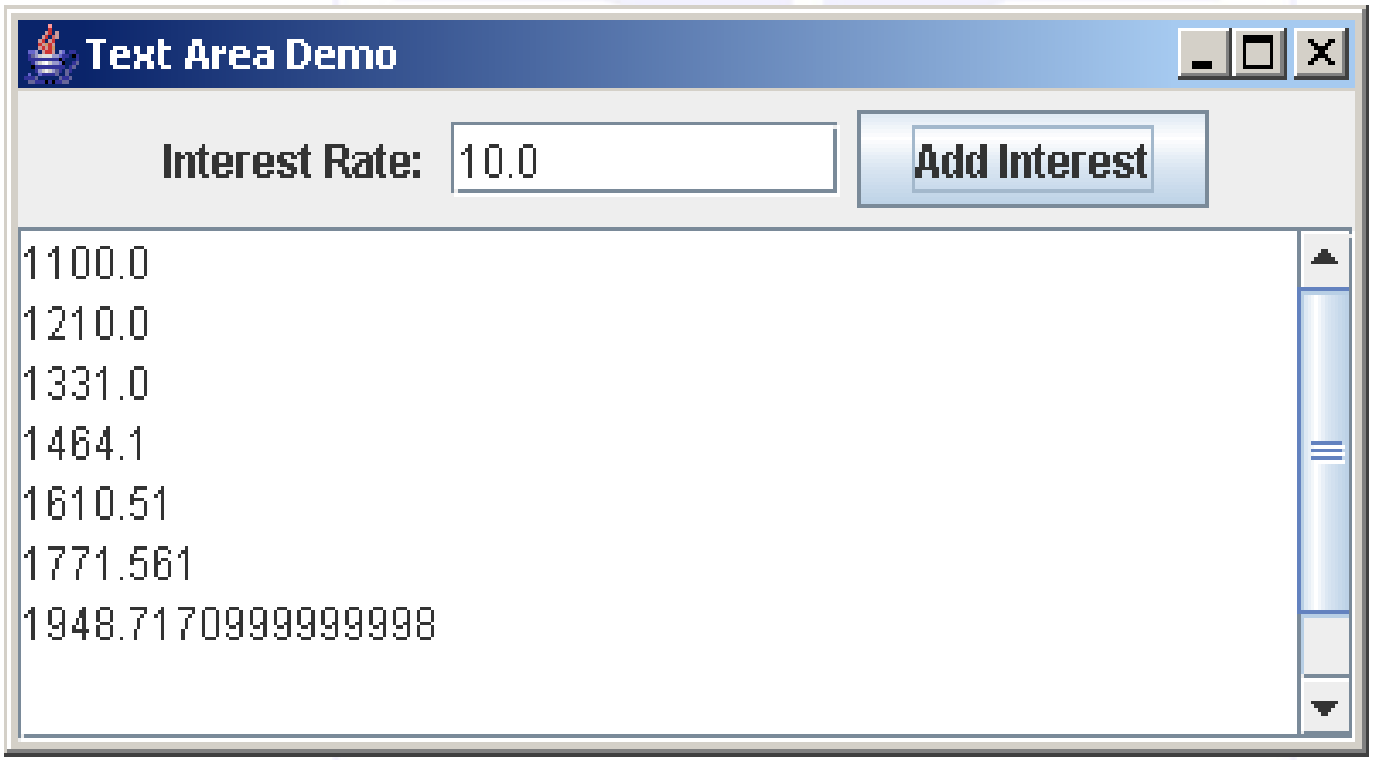
- Dacă o folosim doar pentru afişare:

```
textArea.setEditable(false);
```

- Ca să adăugăm bare de defilare (*scroll bars*) la o zonă de text:

```
JTextArea textArea = new JTextArea(ROWS, COLUMNS);
JScrollPane scrollPane = new JScrollPane(textArea);
```

Zone de text


Text Area Demo
Interest Rate:
10.0
Add Interest
1100.0
1210.0
1331.0
1464.1
1610.51
1771.561
1948.7170999999998

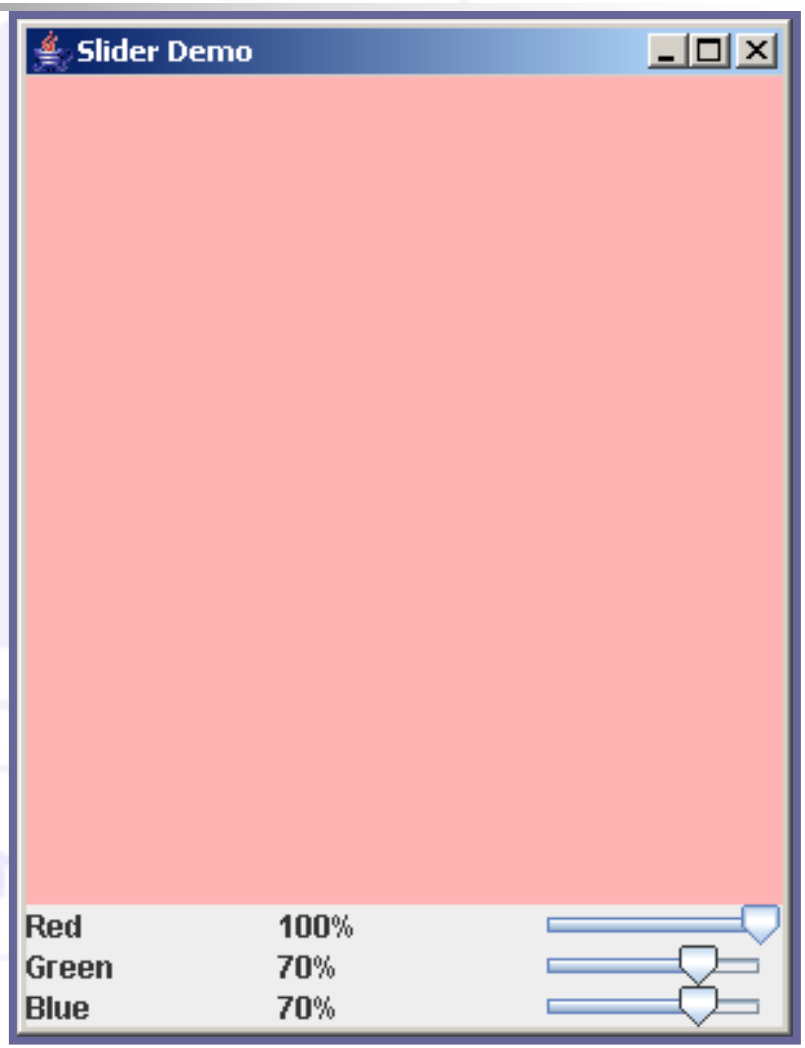
Explorarea documentaţiei Swing

- Pentru efecte mai sofisticate, explorăm documentaţia Swing
- Documentaţia este vastă
- Exemplul care urmează arată cum să exploatăm documentaţia

Exemple: Un amestecător de culori

- Amestecarea propriilor culori folosind un slider (glisant) pentru alegerea valorilor de roşu (R), verde (G) şi albastru (B)

- Există peste 50 metode în clasa **`JSlider`** şi peste 250 metode moştenite



Cum construiesc un `JSlider`?

- Căutăm în documentaţia API Java
- Există şase constructori pentru clasa `JSlider`
- Studiem unul sau doi
- Alegem un punct de echilibru între ceva banal şi ceva bizar:

  Prea limitat: `public JSlider()`
  - Creează un slider orizontal cu gama de la 0...100 şi valoarea iniţială 50

  Bizar: `public JSlider(BoundedRangeModel brm)`
  - Creează un slider orizontal folosind `BoundedRangeModel` specificat

  Folositor pentru noi:

  ```
  public JSlider(int min, int max, int value)
  ```
  - Creează un slider orizontal folosind *min*, *max* şi *value* (valoarea iniţială) precizate

Cum pot fi notificat când utilizatorul deplasează cursorul unui `JSlider`?

- Nu există metodă `addActionListener()`
- Dar este o metodă

```
public void addChangeListener(ChangeListener l)
```

- Click pe legătura `ChangeListener` pentru a afla mai multe
- Are o singură metodă: `void stateChanged(ChangeEvent e)`
- În aparenţă, metoda este apelată ori de câte ori utilizatorul mişcă cursorul slider-ului
- Ce este un eveniment `ChangeEvent`?
  - Moşteneşte metoda `getSource()` din superclasa `EventObject`
  - `getSource()`: ne spune care componentă a generat acest eveniment

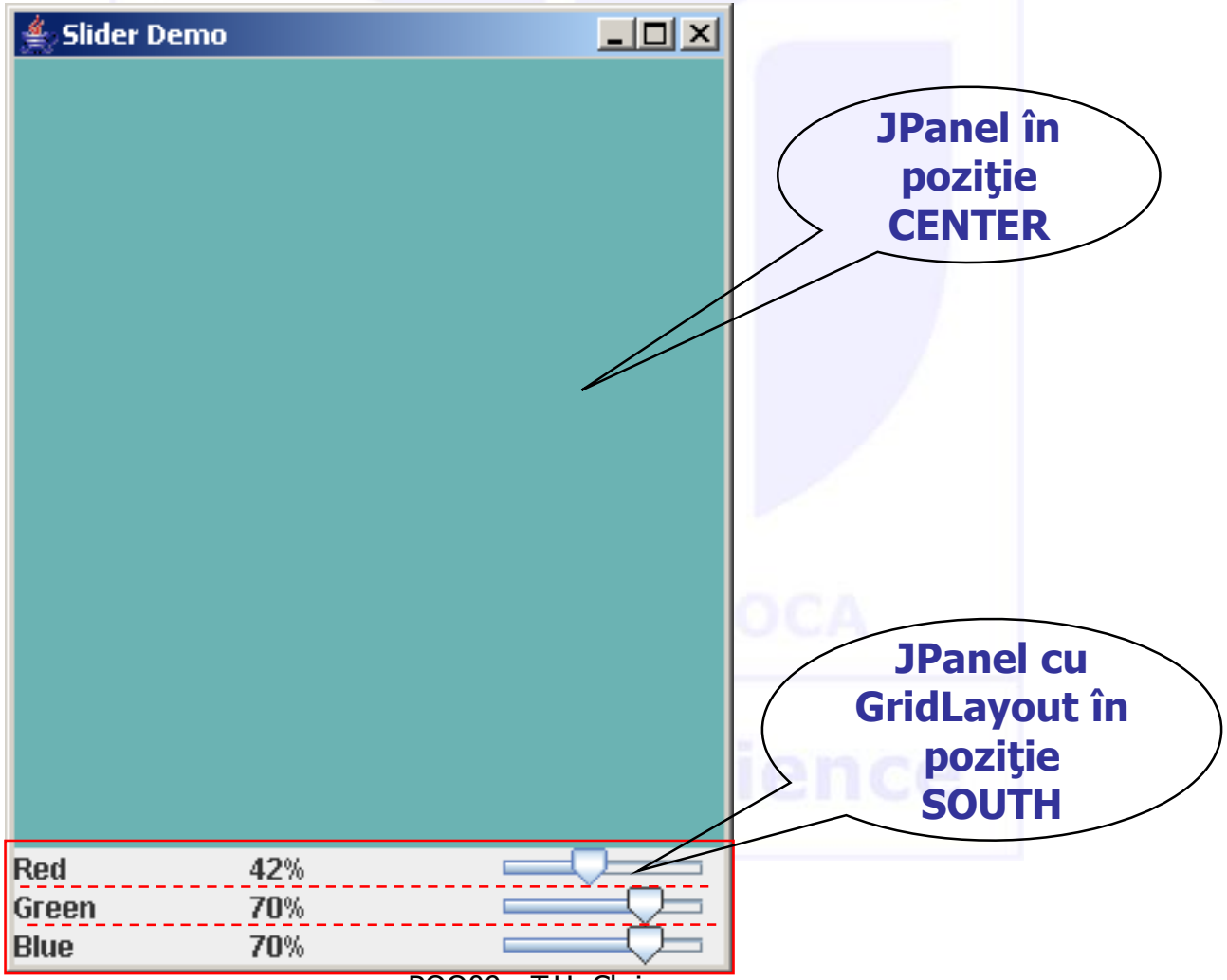
Cum pot fi notificat când utilizatorul deplasează cursorul unui `JSlider`?

- Acum ştim cum să facem:
  - Adăugăm un ascultător pentru evenimentul schimbare (*change event*) la fiecare slider
  - La modificarea poziţiei cursorului este apelată metoda, `stateChanged()`
  - Aflăm noua valoare a slider-ului
  - Re-calculăm valoarea culorii
  - Redesenăm panoul cu culoarea
- Avem nevoie de valoarea curentă a slider-ului
- Ne uităm la toate metodele care încep cu get și găsim:

```
public int getValue()
```

care întoarce valoarea sliderului

Componentele `SliderFrame`



Icoane

- **`JLabels`**, **`JButtons`**, şi **`JMenuItems`** pot avea reprezentări iconice (icoane)
  - O *icoană* nu este decât o mică imagine (de obicei)
  - Nu se cere să fie mică
- O icoană este un obiect de clasă **`ImageIcon`**
  - Se bazează pe un fişier imagine digitală cum sunt **`.gif`**, **`.jpg`**, sau **`.tiff`**
- Etichetele (**`JLabel`**), butoanele (**`JButton`**) şi elementele de meniu (**`JMenuItem`**) pot afişa un şir, o icoană, amândouă sau nimic

Icoane

- Clasa **`ImageIcon`** se foloseşte pentru a converti un fişier cu imagine la o icoană Swing

```
ImageIcon dukeIcon = new ImageIcon("duke_waving.gif");
```

  - Fişierul care conţine imaginea trebuie să se afle în acelaşi director ca şi clasa în care apare acest fragment de cod, sau trebuie dată calea completă sau relativă la el
  - Remarcaţi că numele de fişier este dat sub forma unui şir de caractere

- Ataşarea unei icoane la o etichetă se face cu metoda **`setIcon`** astfel:

```
JLabel dukeLabel = new JLabel("Mood check");
dukeLabel.setIcon(dukeIcon);
```

Icoane

- Altfel, icoana poate fi dată ca argument constructorului lui **`JLabel`**:

```
JLabel dukeLabel = new JLabel(dukeIcon);
```

- Textul poate fi adăugat etichetei folosind metoda **`setText`**:

```
dukeLabel.setText("Mood check");
```

- Icoanele şi textul pot fi adăugate la **`JButton`** şi **`JMenuItem`** la fel ca pentru **`JLabel`**

```
JButton happyButton = new JButton("Happy");
ImageIcon happyIcon = new ImageIcon("smiley.gif");
happyButton.setIcon(happyIcon);
```

Icoane

- Butoanele sau elementele de meniu se pot crea numai cu icoană dând obiectul de tip **`ImageIcon`** ca argument constructorului lui **`JButton`** sau **`JMenuItem`**

```
ImageIcon happyIcon = new ImageIcon("smiley.gif");
JButton smileButton = new JButton(happyIcon);
JMenuItem happyChoice = new JMenuItem(happyIcon);
```

- Butoanele sau elementele de meniu create fară text trebuie să folosească metoda **`setActionCommand()`** pentru a seta explicit comanda acţiunii deoarece nu avem şir de caractere

Tratarea Evenimentelor

- Legătura dintre partea de vizualizare și modelul problemei se face prin transmiterea de evenimente atunci când utilizatorul interacționează cu interfața (ex. *click* pe un buton, selectarea unui *checkbox*, apăsarea unei taste etc.)
- În Swing există o delimitare clară între interfață și implementare (codul ce trebuie rulat în momentul în care un eveniment se întâmplă sa apară)
- Fiecare componentă Swing poate raporta toate evenimentele ce apar în dreptul ei, si le poate raporta în mod individual, astfel încât să poată fi tratate doar evenimentele de interes

Tipuri de evenimente

- Există mai multe feluri de evenimente. Cele mai uzuale sunt:

Control utilizator	***addXXXListener***	***Metoda în ascultător (listener)***

`JButton`
`JTextField`
`JMenuItem`
	`addActionListener()`	`actionPerformed(ActionEvent e)`

`JSlider`	`addChangeListener()`	`stateChanged(ChangeEvent e)`
`JCheckBox`	`addItemListener()`	`itemstateChanged()`
key on component	`addKeyListener()`	`keyPressed()`, `keyReleased()`, **`keyTyped`**`()`
mouse on component	`addMouseListener()`	**`mouseClicked`**`()`, `mouseEntered()`, `mouseExited()`, `mousePressed()`, `mouseReleased()`
mouse on component	`addMouseMotionListener()`	`mouseMoved()`, `mouseDragged()`
`JFrame`	`addWindowListener()`	`windowClosing(WindowEvent e)`, ...

Ascultători (*Listeners*)

- Se apelează un ascultător atunci când utilizatorul interacționează cu interfața, ceea ce provoacă un eveniment
- Deşi evenimentele provin de obicei din interfața utilizator, ele pot avea şi alte surse (d.e., un contor de timp – Timer)
- Exemplu de ascultător pentru un buton:

```
btn.addActionListener(obiect_ascultator);
```

  - Unde *obiect_ascultator* este de tipul unei clase care implementează interfața **ActionListener**
  - La click pe buton se face un apel la metoda **actionPerformed()** definită în clasa obiectului ascultător; metodei i se transmite ca parametru un obiect ActionEvent

Ascultători (*Listeners*)

- Exemplu de clasă care implementează un ascultător:

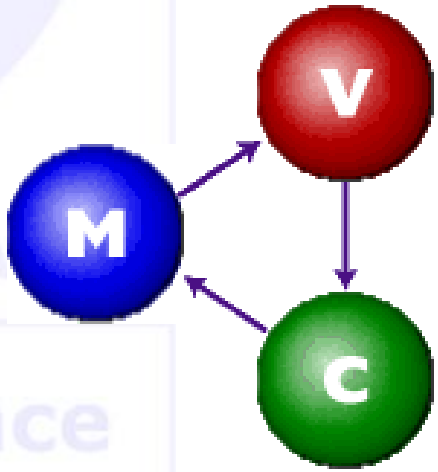
```
class ButtonListener implements ActionListener{
    public void actionPerformed(ActionEvent e){
        //fa ceva cand se apasa butonul, ex
        ++count;
        tf.setText(count + "");
    }
}
```

- Ascultătorii se pot defini și ca clase imbricate cu anonimi. Exemplu:

```
btnCount.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
        //fa ceva cand se apasa butonul
           ++count;
           tf.setText(count + "");
        }
    });
```

Swing şi arhitectura MVC (Model-Vizualizare-Controlor)

- Arhitectura Swing îşi are rădăcinile în arhitectura ***model-view-controller*** (***MVC***) care a fost introdus iniţial în limbajul SmallTalk
- Arhitectura MVC cere ca o aplicaţie vizuală să fie divizată în trei părţi separate:
  - Un *model* care reprezintă intern datele aplicaţiei
  - O *vizualizare* (*view*) pentru reprezentarea vizuală a datelor respective
  - Un *controlor* (*controller*) care preia intrarea de la utilizator şi o transpune în schimbări în model



Modelul

- Majoritatea programelor trebuie să facă ceva util, nu să fie "o altă faţă frumoasă"
  - dar există câteva excepţii
  - au existat programe utile cu mult înaintea apariţiei GUI
- Modelul este partea care face treaba – adică *modelează* problema care este în curs de soluţionare prin program
- Modelul ar trebui să fie independent atât de Controlor cât şi de Vizualizare
  - *Dar poate să le furnizeze amândurora servicii (metode)*
- Independenţa furnizează flexibilitate şi robusteţe

Controlorul

- Controlorul decide ce urmează să facă modelul
- Adesea, utilizatorul are controlul prin intermediul unei GUI
  - În acest caz, GUI şi Controlorul sunt adesea unul și acelaşi
- Controlorul şi Modelul pot fi separate aproape întotdeauna (ce trebuie făcut în raport cu în ce fel trebuie făcut)
- Proiectarea Controlorului depinde de model
- Modelul *nu ar trebui* să depindă de Controlor

Vizualizarea

- Tipic, utilizatorul trebuie să poată vedea, sau *vizualiza,* ce face programul
- Vizualizarea arată ce face Modelul
  - Vizualizarea este un observator *pasiv*; ea nu ar trebui să afecteze modelul
- Modelul trebuie să fie independent de vizualizare (dar îi poate furniza metode de acces)
- Vizualizarea *nu* trebuie să afişeze ce *crede* Controlorul că se întâmplă

Combinarea Controlorului şi a Vizualizării

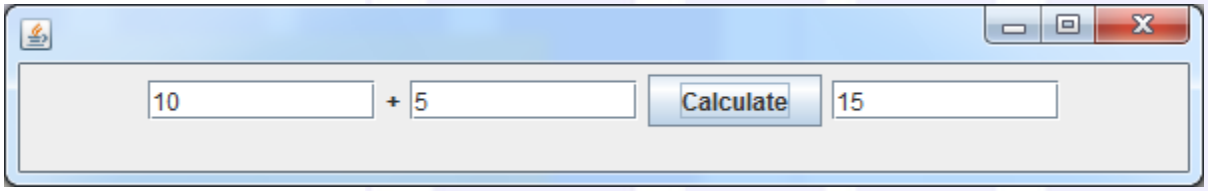
- Uneori Controlorul şi Vizualizarea sunt combinate, mai ales în programe de mici dimensiuni
- Combinarea Controlorului şi a Vizualizării este potrivită dacă cele două sunt foarte interdependente
- Modelul trebuie să rămână independent
- *Nu amestecaţi niciodată* codul din Model cu codul GUI!

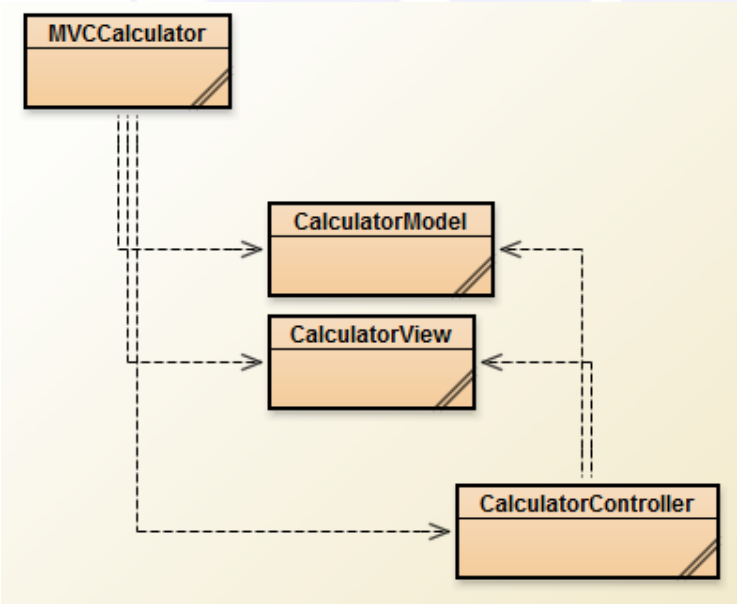
Separarea preocupărilor

- Ca întotdeauna, dorim independenţa codului
- Modelul nu trebuie contaminat cu cod din control sau din vizualizare
- Vizualizarea trebuie să reprezinte Modelul aşa cum este în realitate, nu vreo stare pe care şi-o aminteşte
- Controlorul trebuie să *converseze* cu Modelul şi Vizualizarea, nu să le *manipuleze*
  - Controlorul poate seta variabile pe care Modelul şi Vizualizarea le pot citi

Exemplu MVC

- Implementarea unui calculator simplificat:





Exemplu MVC: Modelul

- Clasa principală

```
public class MVCCalculator {

    public static void main(String[] args) {

    CalculatorView theView = new CalculatorView();

    CalculatorModel theModel = new CalculatorModel();

    CalculatorController theController = new
              CalculatorController(theView,theModel);

    theView.setVisible(true);

    }
}
```

- Modelul

```
public class CalculatorModel {
// face suma numerelor introduse din interfață

    private int calculationValue;

    public void addTwoNumbers(int firstNumber,
                              int secondNumber)
    {
        calculationValue = firstNumber +
                                   secondNumber;
    }

    public int getCalculationValue()
    {
        return calculationValue;
    }
}
```

Exemplu MVC: Vizualizarea

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

public class CalculatorView extends JFrame{

 private JTextField firstNumber  = new JTextField(10);
 private JLabel additionLabel = new JLabel("+");
 private JTextField secondNumber = new JTextField(10);
 private JButton calculateButton = new
                                    JButton("Calculate");
 private JTextField calcSolution = new JTextField(10);

 CalculatorView(){
  JPanel calcPanel = new JPanel();
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  this.setSize(600, 200);

  calcPanel.add(firstNumber);
  calcPanel.add(additionLabel);
  calcPanel.add(secondNumber);
  calcPanel.add(calculateButton);
  calcPanel.add(calcSolution);
  this.add(calcPanel);
 }
```

```
 public int getFirstNumber(){
  return Integer.parseInt(firstNumber.getText());
 }

 public int getSecondNumber(){
  return Integer.parseInt(secondNumber.getText());
 }

 public int getCalcSolution(){
  return Integer.parseInt(calcSolution.getText());
 }

 public void setCalcSolution(int solution){
  calcSolution.setText(Integer.toString(solution));
 }

 void addCalculateListener(ActionListener
                           listenForCalcButton){
 calculateButton.addActionListener(
                                  listenForCalcButton);
 }

 void displayErrorMessage(String errorMessage){
 JOptionPane.showMessageDialog(this, errorMessage);
 }
}
```

Exemplu MVC: Controlorul

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class CalculatorController {

 private CalculatorView theView;
 private CalculatorModel theModel;

 public CalculatorController(CalculatorView
           theView, CalculatorModel theModel) {
  this.theView = theView;
  this.theModel = theModel;
  this.theView.addCalculateListener(new
                          CalculateListener());
}
```

```
class CalculateListener implements
ActionListener{
 public void actionPerformed(ActionEvent e) {
 int firstNumber, secondNumber = 0;
 try{
   firstNumber = theView.getFirstNumber();
   secondNumber = theView.getSecondNumber();
              theModel.addTwoNumbers(firstNumber,
                                     secondNumber);
   theView.setCalcSolution(
                 theModel.getCalculationValue());


   }
   catch(NumberFormatException ex){
   System.out.println(ex);

   theView.displayErrorMessage("You Need to Enter
                                2 Integers");
   }
  }
 }
}
```

Animație cu clasa Timer

- La fel ca și în cazul butoanelor sau a altor componente grafice, și pentru Timer trebuie implementată metoda actionPerformed() din interfața ActionListener

- Pentru a porni/opri o animație se apelează metodele start() și stop() din Timer

Exemplu animație cu clasa Timer

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

public class TimerEx extends JPanel implements
                                    ActionListener
{
 JLabel l;
 Timer t;
 int x = 10;
 int y = 300;

 TimerEx()
 {
   ImageIcon img = new ImageIcon("Mario.gif");
   l = new JLabel(img);
   l.setLocation(x, y);
   this.add(l);
   setBackground(Color.white);
   t = new Timer(100, this);
   t.start();
 }
```

```
// @override
   public void actionPerformed(ActionEvent e)
   {
     x+=20;
     if (x>800) x = 50;
     l.setLocation(x,y);
   }


   public static void main(String[] args)
   {
     JFrame frame = new JFrame("Timer Example");
     frame.setDefaultCloseOperation(
                                JFrame.EXIT_ON_CLOSE);
     frame.setSize(800, 800);
     TimerEx pane= new TimerEx();
     frame.setContentPane(pane);
     frame.setVisible(true);
   }
}
```