Programare orientată pe obiecte

1. Interfeţe utilizator grafice (GUIs) - continuare
2. Fire de lucru Java

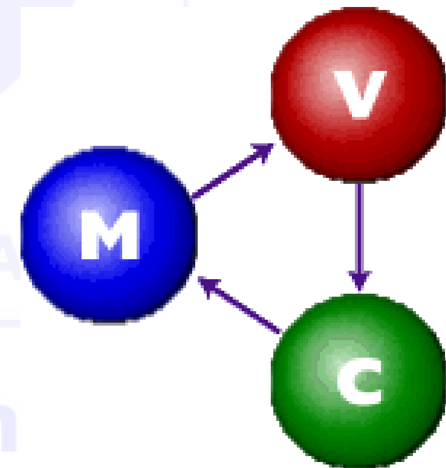
TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

1. Interfeţe utilizator grafice (GUIs) - continuare

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

Arhitectura MVC (Model-Vizualizare-Controlor)

- Arhitectura *model-view-controller* (*MVC*) cere ca o aplicaţie vizuală să fie divizată în trei părţi separate:
  - Un *model* care reprezintă intern datele aplicaţiei
  - O *vizualizare (view)* – 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
  - *modelează* problema care este în curs de soluţionare prin program.
  - Stochează datele necesare pentru *Control* sau *Vizualizare*
- 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

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

Controlorul

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

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

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!

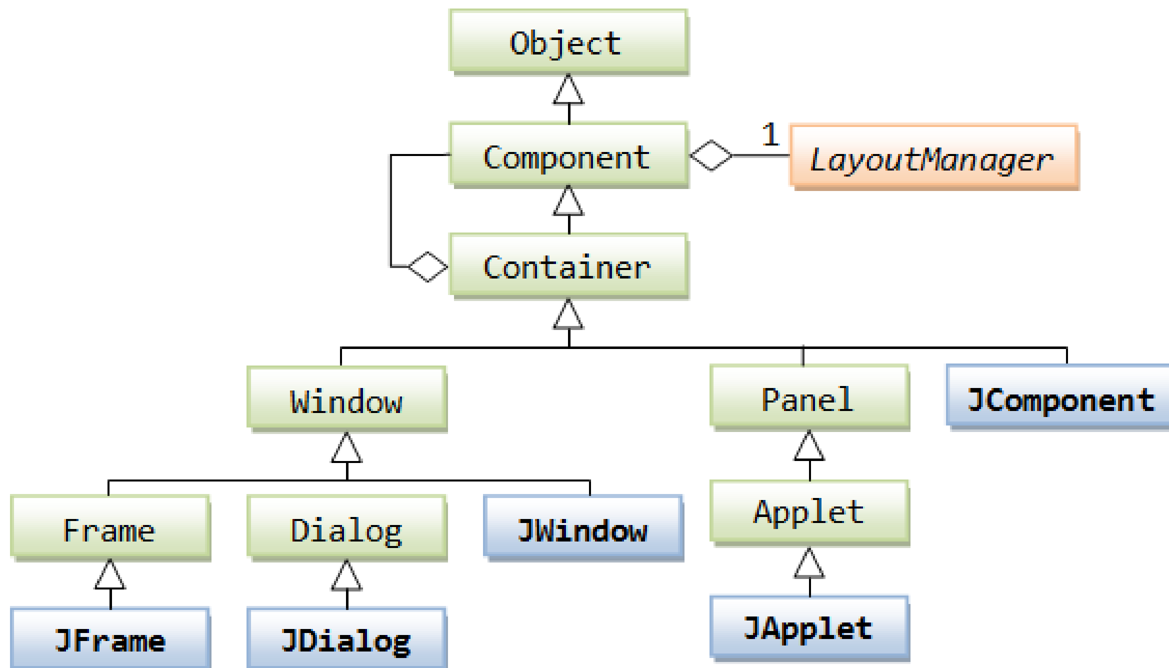
TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

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

GUI - Containere şi componente (din nou)

- Componente
  - Componente **Lightweight** (extind direct Component)
  - Componente **Heavyweight** (asociate ferestrelor native)
- Containere

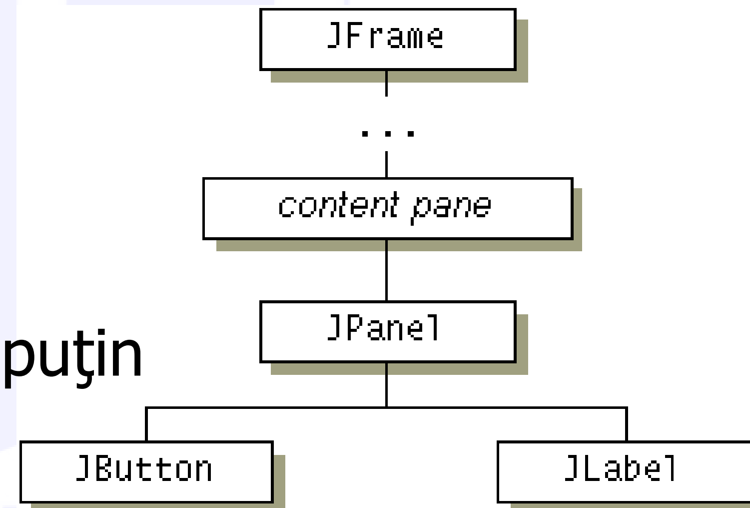


Clasa `Container`

- Orice clasă care descinde in clasa `Container` este considerată o clasă container
  - Clasa `Container` se află în pachetul `java.awt`, nu în biblioteca 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

Ierarhii de conţinere

- 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
    - Cadre (frames)
    - Dialoguri
    - Applet-uri



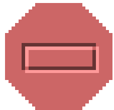
Dialoguri

- Mai limitate decât cadrele
- 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`

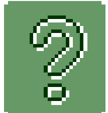
TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

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);
```

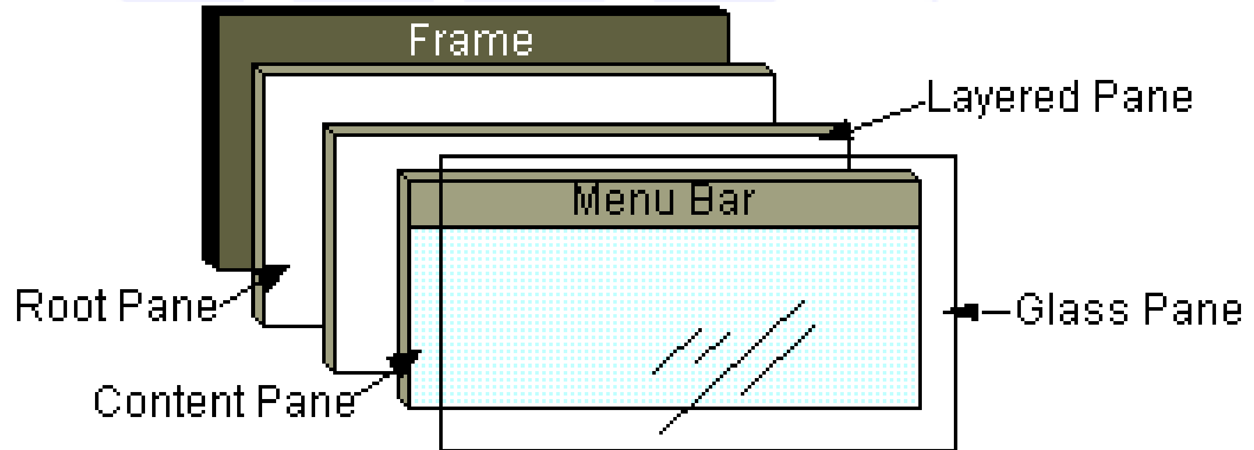
  - Intrare, confirmare

- Individualizare
  - showOptionDialog – destul de individualizabil (customizable)
  - JDialog - total individualizabil

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

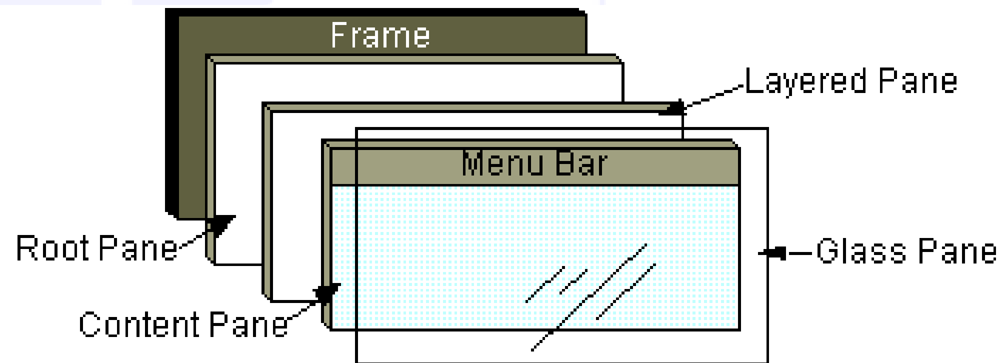
Containere intermediare – panouri (panels / 'panes')

- Panouri rădăcină (root panes)
  - panou de conținut (content pane)
  - panouri stratificate (layered panes)
  - panouri de sticlă (glass panes)

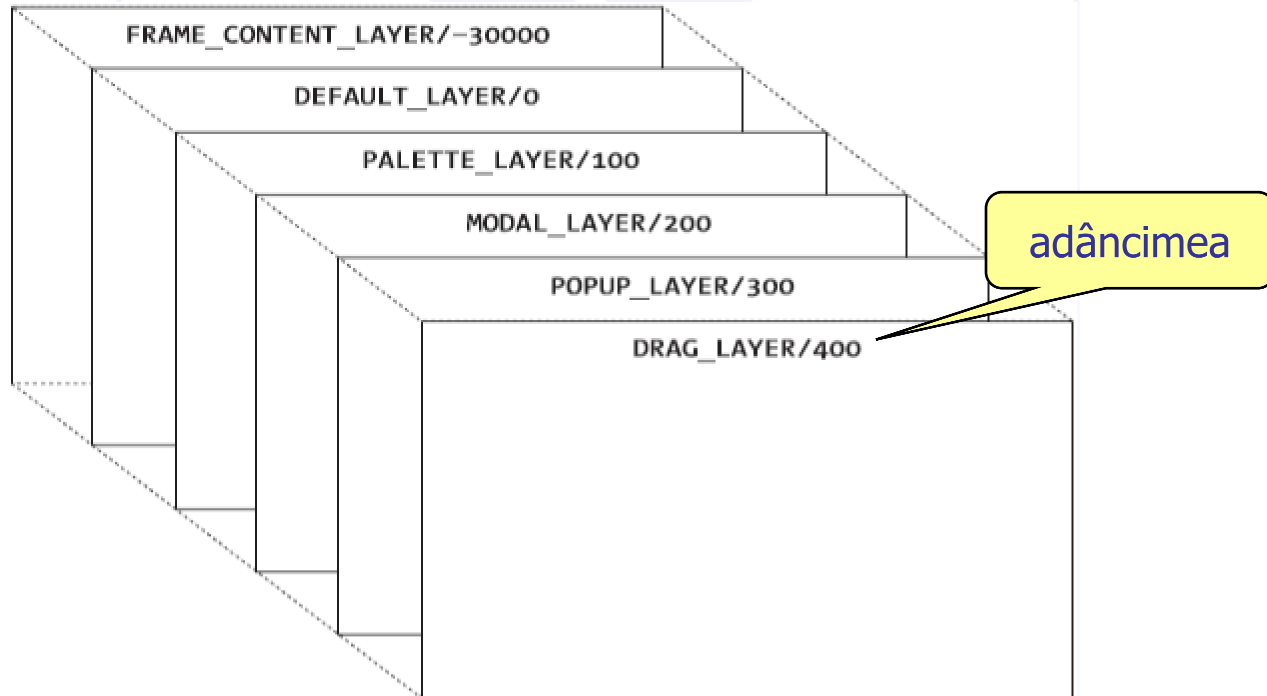


Panou rădăcină

- Atașat 'invizibil' la containerul de nivel înalt
- Creat de Swing la realizarea cadrului
- Gestionează totul între containerul de nivel înalt şi componente
- Plasează bara de meniu şi panoul de conţinut într-o instanţă de `JLayeredPane`



JLayeredPanes



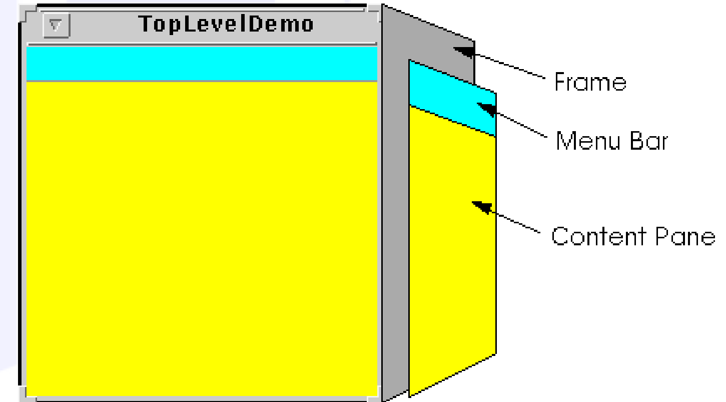
Exemplu:

```
JLabel label = createColoredLabel(...);
layeredPane.add(label, new Integer(i));
```

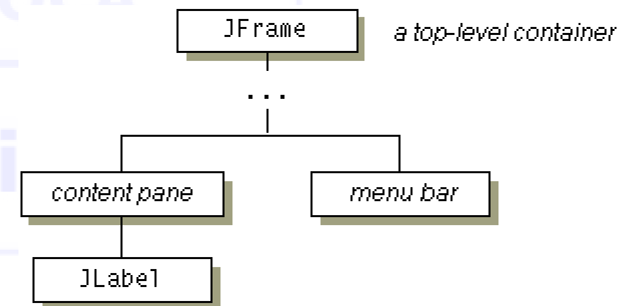
TECHNICAL UNIVERSITY
Computer Science

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
  - cod simplificat




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



Panouri de sticlă

- Nestructurate în componente
  - interceptarea evenimentelor
  - desenare (painting)
- Folosite rar
- Fie sunt create explicit fie se foloseşte versiunea rădăcină

Obiecte dintr-un GUI tipic

- Aproape fiecare GUI construit folosind clasele container din Swing vor fi compuse 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

Sugestie: programaţi aspectul (look) şi acţiunile GUI separat

- Sarcina proiectării unui GUI poate fi împărţită în două sub-sarcini principale:
  1. Proiectarea şi programarea aspectului GUI pe ecran
  2. Proiectarea şi programarea acţiunilor de efectuat ca răspuns la acţiunile utilizatorului
- În particular, este util să implementăm metoda **`actionPerformed()`** ca *talon* (nu face nimic) până când GUI arată aşa cum trebuie

```
public void actionPerformed(ActionEvent e){}
```

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

Folosirea moştenirii pentru a individualiza (customize) cadrele

- Folosim moştenirea pentru cadrele (frames) complicate pentru a face programele mai uşor de înţeles
- Proiectăm o subclasă a lui `JFrame`
- Stocăm componentele sub forma câmpurilor instanţă
- Iniţializăm componentele în constructorul subclasei proiectate
- Dacă partea de cod pentru iniţializare devine complexă, atunci adăugăm câteva metode ajutătoare

Gestiunea aranjării

- Până acum am folosit un control limitat asupra aranjării (layout) componentelor
  - Când am folosit un panou, acesta a aranjat 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

Gestiunea aranjării

- Implicit, **`JPanel`** amplasează componentele de la stânga la dreapta şi începe un rând nou atunci când este necesar
- Aranjarea panourilor (panel) este efectuată de gestionarul de aranjări **`FlowLayout`**
- Se pot seta alte gestionare de aranjare

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

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

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

BorderLayout.NORTH		
BorderLayout.WEST	BorderLayout.CENTER	BorderLayout.EAST
BorderLayout.SOUTH		

Border Layout

- Este gestionarul de aranjare implicit pentru cadre – "frame" (tehnic, pentru panoul de conţinut al cadrului)
- 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);
```

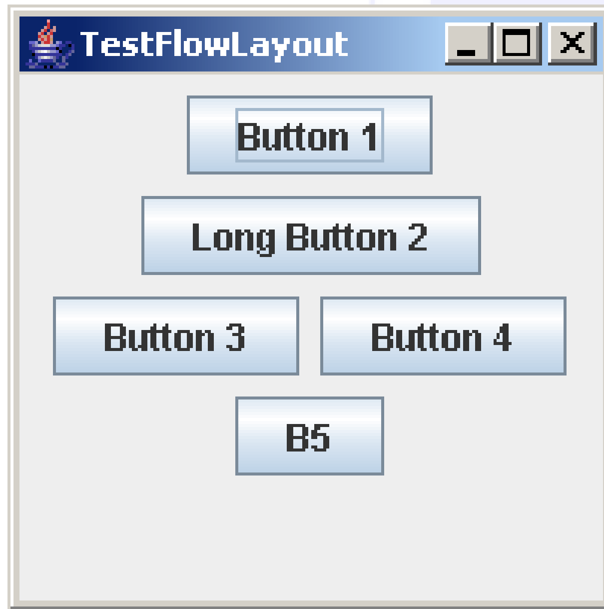
- Alinierea poate fi **LEFT**, **RIGHT**, sau **CENTER**
- Este implicit pentru **JPanel**

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:

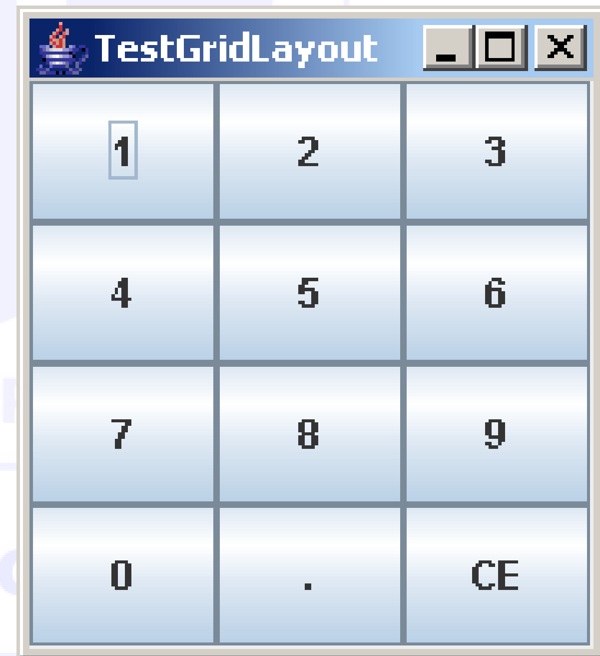
```
JPanel numberPanel = new JPanel();
numberPanel.setLayout(new GridLayout(4, 3));
numberPanel.add(button1);
numberPanel.add(button2);
numberPanel.add(button3);
numberPanel.add(button4);
. . .
```

Exemple: FlowLayout şi Grid Layout



BlueJ: TestFlowLayout

BlueJ: TestGridLayout



Gestionarul de aranjare **GridBagLayout**

- Aranjare tabelară a componentelor
  - Coloanele pot avea mărimi diferite
  - Componentele se pot întinde pe mai multe coloane
- Destul de complicat de folosit
- Din fericire se pot crea aranjamente care să arate acceptabil prin imbricarea panourilor
  - Dăm fiecărui panou un gestionar de aranjare corespunzător
  - Panourile nu au margini vizibile
  - Folosim câte panouri sunt necesare pentru a organiza componentele

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:

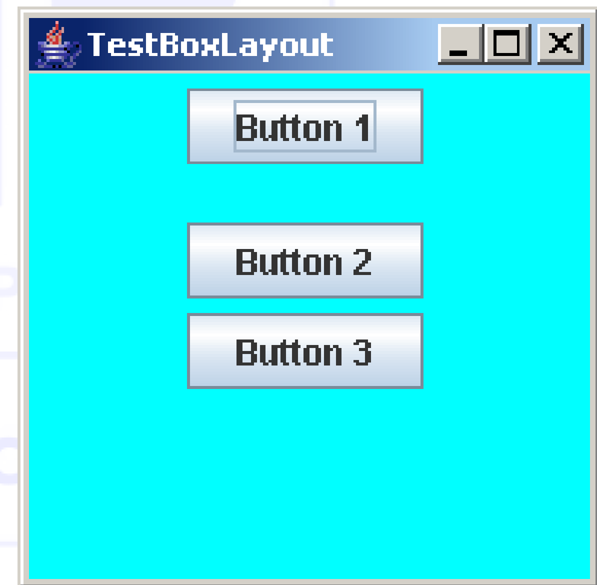
```
public BoxLayout(Container c, int direction);
```

- `direction` poate fi `X_AXIS` sau `Y_AXIS`

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);
}
```



Controale pentru alegeri

- Butoane radio (Radio buttons)
- Cutiuţe de marcare (checkboxes)
- Cutii Combo (combo boxes)

Butoane radio

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

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

Cutii de marcare (bifare)

- Au două stări: marcat (checked) şi nemarcat
- Pentru o alegere din doua variante posibilă folosim o cutie de marcare (checkbox)
- Folosim un grup de cutii de marcare atunci când o alegere nu exclude o alta
- Exemplu: "bold" şi "italic" la alegerea stilului unui font
- Le construim:

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

- Nu le amplasăm în grup de butoane

Cutii Combo

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

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

Cutii Combo

- 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()`**

Butoane radio, cutii de marcare şi cutii combo împreună

- Ori de câte ori utilizatorul selectează un element, ele generează un `ActionEvent`
- Exemplu: `ChoiceFrame`
  - Toate componentele notifică acelaşi obiect ascultător
  - La clic pe oricare componentă, interogăm fiecare componentă pentru a-i afla conţinutul curent
  - Apoi redesenăm textul din exemplu folosind noul font

BlueJ: ChoiceFrameViewer

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

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

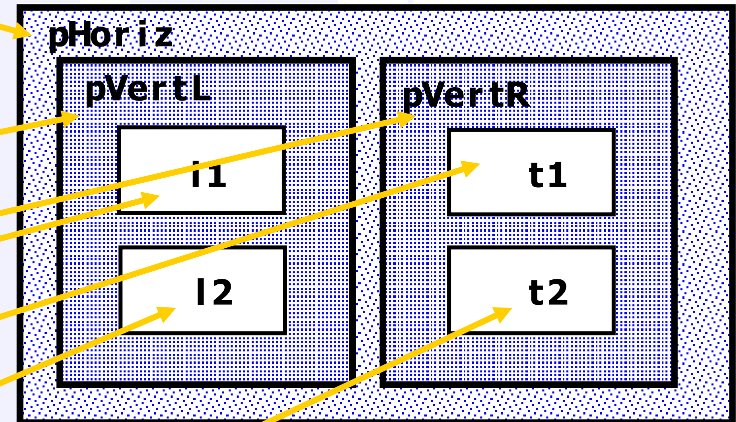
Combinarea gestionarilor de aranjare

- 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

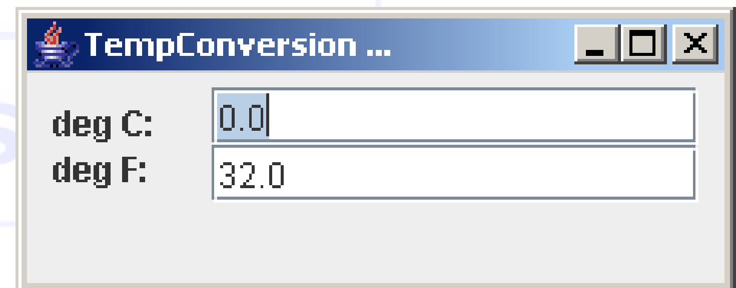
Exemple: Containere şi aranjări imbricate

```
// Creaza un nou panou de nivel sus
JPanel pHoriz = new JPanel();
pHoriz.setLayout(new BoxLayout(pHoriz,
BoxLayout.X_AXIS));
add( pHoriz );
// Creaza 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 to pHoriz cu spatiu oorizontal
// intre panouri
pHoriz.add( pVertL );
pHoriz.add( Box.createRigidArea(new
Dimension(20,0)) );
pHoriz.add( pVertR );
// Creaza cimpul grade Celsius
l1 = new JLabel("deg C:", JLabel.RIGHT);
pVertL.add( l1 );
t1 = new JTextField("0.0",15);
t1.addActionListener( cHnd );
pVertR.add( t1 );
// Creaza cimpul grade Fahrenheight
l2 = new JLabel("deg F:", JLabel.RIGHT);
pVertL.add( l2 );
t2 = new JTextField("32.0",15);
t2.addActionListener( fHnd );
pVertR.add( t2 );
```

Structura:

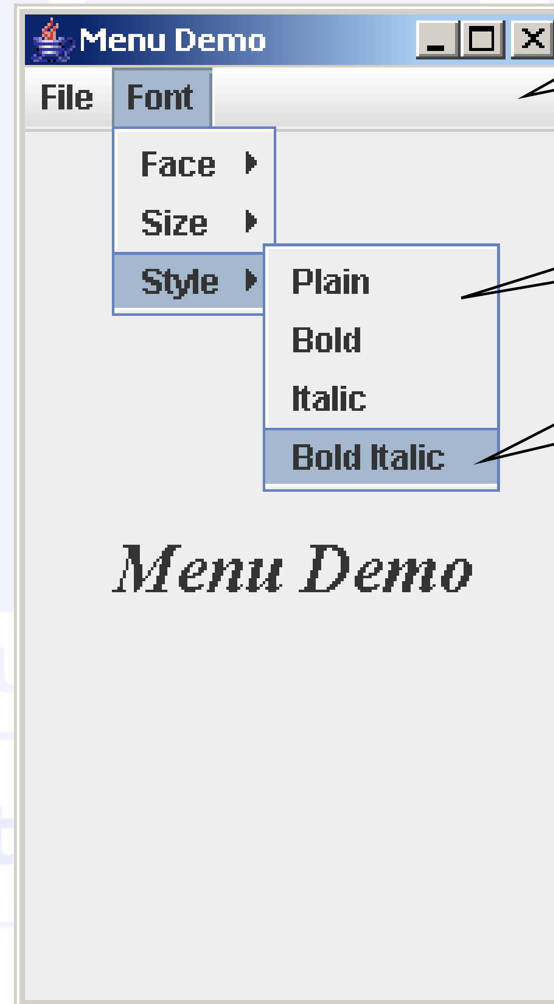


Rezultatul:



Meniuri

- Cadrul (frame) conţine o bară de meniu
- Bara de meniu conţine meniuri
- Meniul conţine submeniuri şi elemente de meniu
  - Meniuri Pull-Down



Meniuri (Swing)

JMenu

Accelerator

Mnemonic

MenuSample Example

File Edit

JMenuBar

Cut Ctrl-X

Copy Ctrl-C

Paste Ctrl-V

Find F3

Options

Forward

Backward

Case Sensitive

JRadioButtonMenuItem

JSeparator

JCheckBoxMenuItem

JMenuItem

JPopupMenu

TECHNICAL UNIVERSITY
Computer Science

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);
// program can call setText and append to change it
```

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

TECHNICAL UNIVERSITY
Computer Science

BlueJ TextAreaViewer

Explorarea documentaţiei Swing

- Pentru efecte mai sofisticate, explorăm documentaţia Swing
- Documentaţia este vastă, dar nu trebuie să ne descurajăm
- Exemplu care urmează ne arată cum să exploatăm documentaţia

UNIVERSITY OF CLUJ-NAPOCA
Computer Science

2. Fire de lucru Java

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA
Computer Science

Multitasking

- Mai multe procese par a se executa simultan, la nivelul sistemului de operare

Paradigma multitasking

Sistem de operare

Aplicația #1

Aplicația #2

Aplicația #3

Memorie locală

Memorie locală

Memorie locală

Memorie partajată

Multitasking vs Multithreading

Paradigma firelor de lucru multiple (multithreading)

Sistem de operare

Applicaţia #1

Applicaţia #2

Maşina virtuală Java

Fir de lucru #1

Fir de lucru #2

Fir de lucru #3

Variabile locale

Variabile locale

Variabile locale

Memorie globală

Sarcini ale firelor de lucru

- Firele de lucru sunt utile în mai multe feluri:
  - Acolo unde trebuie să se întâmple mai multe lucruri simultan.
    - D.e., o aplicaţie multimedia poate necesita ca procesele audio, video şi de control să se execute în paralel. Există adesea perioade de aşteptare a răspunsului sistemelor de IE mai lente, timp în care procesorul poate face altceva.
  - Programe cum sunt sistemele server/client sunt mult mai uşor de proiectat şi scris folosind fire de lucru.
  - Algoritmi matematici, cum sunt sortarea, căutarea numerelor prime etc. sunt potrivite pentru prelucrarea paralelă.
  - Pe sistemele multi-procesor, maşinile virtuale Java pot rula firele de lucru pe procesoare diferite şi pot obţine astfel prelucrare paralelă adevărată şi creşteri de performanţe semnificative faţă de platformele mono-procesor.

Multithreading în Java

- Proprietăţile firelor de lucru multiple din programele Java:
  - Fiecare fir de lucru îşi începe execuţia la o locaţie bine cunoscută, predefinită
  - Fiecare fir de lucru îşi execută codul începând de la locaţia de start, într-o secvenţă ordonată, predefinită (pentru un set de date de intrare dat)
  - Fiecare fir de lucru îşi execută codul independent de celelalte fire de lucru din program
  - Firele de lucru par a avea un anumit grad de simultaneitate în execuţie
  - Firele de lucru au acces la diferite tipuri de date

Multithreading în Java

- Toate programele Java în afara aplicaţiilor simple cu intrare-ieşire pe consolă sunt aplicaţii multithreading.
- Procesele *grele* (*heavyweight*) se rulează direct sub sistemul de operare al maşinii locale.
- Fir de lucru: un singur curs secvenţial al controlului, numit şi proces *uşor* (*lightweight*), lansat din procesul principal
  - Sunt procese paralele care rulează *înăuntrul* unui program
  - În Java se pot crea fire de lucru din program, aşa cum se pot rula programe sub un sistem de operare
- Java: creează fire de lucru în două moduri:
  - Clasa extinde clasa `Thread` şi îi suprascrie metoda `run()`.
  - Clasa implementează interfaţa `Runnable`, care are o singură metodă: `run()`.
    - **Clasa îi transmite o referinţă la sine însuşi când creează un fir de lucru.**
    - **Apoi firul apelează înapoi metoda `run()` din clasă.**

Sub-clasarea clasei `Thread`

- Metoda `run()` din fir corespunde metodei `main()` pentru o aplicaţie.
- La pornirea firului, acesta invocă metoda `run()`, iar atunci când procesul revine din metoda `run()` firul de lucru *moare*.
  - Nu se poate "învia" un fir de lucru mort. În loc de "înviere", trebuie creată o nouă instanţă de fir de lucru.
- Subclasa trebuie să suprascrie metoda `run()`.
- În metoda `run()` se pune codul care trebuie executat în paralel cu programul principal

Exemplu de subclasare a clasei `Thread`

- Demo: subclass

MyApplet

Crează un obiect subclasă a lui Thread

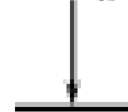
`myThread=new Thread()` → **MyThread**

Apoi invocă

`myThread.start()` → `start()` revine și începe un nou proces cu invocarea lui `run()` din acest obiect `MyThread()`.

pentru a lansa procesul

`run()`



procesul moare atunci când se termină `run()`

Implementarea interfeţei `Runnable`

- Implementăm interfaţa **`Runnable`** şi îi suprascriem metoda **`run()`**
- Transmitem o referinţă la o instanţă a respectivei implementări a **`Runnable`** spre o instanţă de fir de lucru şi firul de lucru *apelează înapoi (calls back)* metoda **`run()`** din obiectul **`Runnable`**.
- Firul de lucru moare, ca la procedeul anterior, la revenirea procesului din **`run()`**.
- Procedeul este convenabil în cazurile în care dorim să creăm un singur tip de fir de lucru, cum este, spre exemplu, *animaţia dintr-un applet*.

Exemplu de implementare a interfeţei **Runnable**

- Demo: runnable

MyRunnableApplet	Thread
Creează un obiect Thread şi trasmite-i o referinţă spre acest obiect Runnable myThread=new Thread() →	Thread
	Constructorul salvează referinţa la obiectul Runnable
Apoi invocă Thread.start() pentru a lansa procesul fir	start() revine la începerea noului proces cu invocarea lui run() din obiectul Runnable.
run()	Runnable.run() Procesul moare când run() revine

Interfaţa **Runnable**: Schiţă de implementare

```
public class ClassToRun extends SomeClass implements
   Runnable
{ . . .
  public void run()
  {
    // Completati aici dacă ClassToRun
    // a fost derivată din Thread
  }
  . . .
  public void startThread()
  {
    Thread theThread = new Thread(this);
    theThread.run();
  }
  . . .
}
```

Sub-clasarea vs. `Runnable`

- Tehnica folosind `Runnable` este în special convenabilă dacă dorim să creăm un singur fir care să efectueze o anumită sarcină de lucru.
  - Metoda `run()` va avea acces la variabilele instanţă din obiectul `Runnable`.
  - D.e., pentru animaţie în applet, faceţi applet-ul `Runnable`.
    - **Metoda `run()` are atunci acces la variabilele applet-ului, care pot fi parametri transmişi prin etichetele applet-ului sau stabiliţi de utilizator prin intermediul interfeţei grafice**
- Dacă doriţi sa creaţi mai multe fire de lucru, atunci de obicei e mai bine să derivaţi din clasa `Thread`.
  - Aceasta ajută la conceptualizarea mai bună a firelor ca obiecte independente
  - Puteţi seta valorile oricăror parametri de care ar fi nevoie folosind constructori sau metode "setter" (de setare)

Stoparea/Punerea în pauză a unui fir de lucru

- Un fir se opreşte în trei moduri:
  - Dacă revine normal din `run()`. [Cea mai bună cale]
  - Se apelează metoda `stop()` a firului. (Acum depreciată (deprecated). Nu o folosiţi.)
  - Dacă este întrerupt de o excepţie neinterceptată.
- Dacă metoda `run()` conţine o buclă de durată sau una infinită – opriţi buclarea atunci când se modifică o variabilă care poate fi modificată de procesul principal. D.e. o valoare `boolean` setată la `false` sau o variabilă referinţă setată la `null` pentru oprire
- Opriţi explicit întotdeauna firele din applet-uri la apelul metodei `stop()` a *applet-ului*.
  - În caz contrar, firele pot continua să ruleze chiar dacă browserul încarcă o noua pagină de Web

Stoparea/Punerea în pauză a unui fir de lucru

- Pentru cazul **`Runnable`**, puteţi porni un nou fir cu setările variabilelor la valorile pe care le-au avut în momentul opririi firului precedent.
- În acest caz, oprirea unui fir şi pornirea unuia nou acţionează ca acţiuni *pauză/start*.
- Metodele clasei **`Thread`** *suspend()* şi *resume() au fost* depreciate (deprecated) pentru a evita problemele de interblocare a firelor de lucru

`Thread.sleep`

- `Thread.sleep` este o metodă statică din clasa `Thread` care pune în pauza un fir de lucru care conţine invocarea
  - Pune în pauză firul timp de numărul de milisecunde dat ca argument
  - Remarcaţi că metoda poate fi invocată dintr-un program obişnuit pentru a insera o pauză în singurul fir al programului respectiv
- Metoda poate arunca o excepţie verificată, `InterruptedException`, care trebuie declarată sau interceptată
  - Atât clasa `Thread` cât şi `InterruptedException` se află în pachetul `java.lang`

Stările firelor de lucru

- `getState():` (Java 5) metoda întoarce un `Enum` de `Thread.States`. Stări:
  - NEW – fir "proaspăt" care nu şi-a început încă execuţia.
  - RUNNABLE – fir în curs de execuţie în JVM.
  - BLOCKED – fir blocat care aşteaptă după un zăvor monitor.
  - WAITING – fir care aşteaptă să fie notificat de un alt fir.
  - TIMED_WAITING – fir care aşteaptă un anumit timp să fie notificat de un alt fir.
  - TERMINATED – fir a cărui metodă run s-a terminat.

Ciclul de viaţă al unui fir de lucru

Planificatorul de fire

- Planificatorul de fire execută fiecare fir de lucru pentru un interval de timp scurt (o *felie de timp (time slice*))
- Apoi planificatorul activează un alt fir
- Întotdeauna vor fi variaţii ale timpilor de rulare, în special la apelul serviciilor sistemului de operare (d.e. pentru operaţii de intrare şi de ieşire)
- Nu sunt garanţii referitoare la ordinea de execuţie a firelor

Terminarea firelor de lucru

- Un fir de lucru se termină la terminarea metodei sale **`run()`**
- *Nu terminaţi* un fir de lucru folosind metoda depreciată ***`stop`***
- În loc de aceasta, notificaţi firul de lucru că ar trebui să se termine folosind

```
t.interrupt();
```

- **`interrupt()`** nu face ca firul să se termine – metoda setează un câmp boolean în structura de date a firului

Terminarea firelor de lucru

- Metoda **`run()`** ar trebui să verifice ocazional dacă firul de lucru a fost întrerupt
  - Folosiţi metoda **`interrupted()`**
  - Un fir de lucru care a fost întrerupt ar trebui să elibereze resursele pe care le foloseşte, să "cureţe" şi să se termine

```
public void run()
{
    for (int i = 1;
          i <= REPETITIONS && !Thread.interrupted(); i++)
    {
        Do work
    }
    Clean up
}
```

Terminarea firelor de lucru

- Metoda **`sleep`** aruncă o excepţie de tipul **`InterruptedException`** atunci când un fir "adormit" este întrerupt
  - Interceptează excepţia
  - Termină firul de lucru

```
public void run() {
    try {
        for (int i = 1; i <= REPETITIONS; i++)
        {
            Do work
        }
    }
    catch (InterruptedException exception) {
    }
    Clean up
}
```

Terminarea firelor de lucru

- Java nu forţează terminarea unui fir atunci când acesta este întrerupt
- Este treaba firului de lucru ce anume face atunci când este întrerupt
- Întreruperea reprezintă un mecanism general pentru a obţine "atenţia" firului de lucru întrerupt

Animaţii

- Animaţiile sunt o sarcină uzuală pentru firele de lucru: ele sunt folosite la controlul animaţiei
  - Un fir de lucru poate dirija desenarea fiecărui cadru în timp ce alte aspecte, cum sunt
  - Răspunsul la interacţiunea cu utilizatorul poate continua în paralel
- Demo: clock, drop2d, sunsort

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science