Curs 6
Curs 6
Interfata grafica

Privire de ansamblu asupra interfetei grafice

Interfata grafica sau, mai bine zis, *interfata grafica cu utilizatorul (GUI)*, este un termen cu înteles larg care se refera la toate tipurile de comunicare vizuala între un program si utilizatorii sai. Aceasta este o particularizare a interfetei cu utilizatorul (UI), prin care vom întelege conceptul generic de interactiune între un program si utilizatorii sai. Asadar, UI se refera nu numai la ceea ce utilizatorul vede pe ecran ci la toate mecanismele de comunicare între acesta si program. Limbajul Java pune la dispozitie numeroase clase pentru implementarea diverselor functionalitati UI, însa ne vom ocupa în continuare de acelea care permit realizarea unei intefete grafice cu utilizatorul (GUI).

Biblioteca de clase care ofera servicii grafice se numeste **java.awt**, AWT fiind prescurtarea de la *Abstract Window Toolkit* si este pachetul care care a suferit cele mai multe modificari în trecerea de la o versiune JDK la alta.
In principiu, crearea unei aplicatii grafice presupune urmatoarele lucruri:

- Crearea unei suprafete de afisare (cum ar fi o fereastra) pe care vor fi asezate obiectele grafice care servesc la comunicarea cu utilizatorul (butoane, controale de editare, texte, etc);
- Crearea si asezarea obiectelor grafice pe suprafata de afisare în pozitiile corespunzatoare;
- Definirea unor actiuni care trebuie sa se execute în momentul când utilizatorul interactioneaza cu obiectele grafice ale aplicatiei;

- "Ascultarea" evenimentelor generate de obiecte în momentul interactiunii cu utilizatorul si executarea actiunilor corespunzatoare asa cum au fost ele definite.

Majoritatea obiectelor grafice sunt subclase ale clasei **Component**, clasa care defineste generic o componenta grafica care poate interactiona cu utilizatorul. Singura exceptie o constituie meniurile care descind din clasa **MenuComponent**.
Asadar, print-o componenta sau componenta grafica vom întelege în continuare orice obiect care are o reprezentare grafica ce poate fi afisata pe ecran si care poate interactiona cu utilizatorul. Exemple de componente sunt ferestrele, butoanele, bare de defilare, etc. In general, toate componentele sunt definte de clase proprii ce se gasesc în pachetul `java.awt`, clasa `Component` fiind superclasa abstracta a tuturor acestor clase.

Crearea obiectelor grafice nu realizeaza automat si afisarea lor pe ecran. Mai întâi ele trebuie asezate pe o suprafata de afisare, care poate fi o fereastra sau suprafata unui applet, si vor deveni vizibile în momentul în care suprafata pe care sunt afisate va fi vizibila. O astfel de suprafata pe care se aseaza obiectele grafice reprezinta o instanta a unei clase obtinuta prin extensia clasei **Container**; din acest motiv suprafetele de afisare vor mai fi numite si containere. Clasa `Container` este o subclasa aparte a clasei `Component`, fiind la rândul ei superclasa tuturor suprafetelor de afisare Java (ferestre, applet-uri, etc). (vezi "Suprafete de afisare")

Asa cum am vazut, interfata grafica serveste interactiunii cu utilizatorul. De cele mai multe ori programul trebuie sa faca o anumita prelucrare în momentul în care utilizatorul a efectuat o actiune si, prin urmare, obiectele grafice trebuie sa genereze evenimente în functie de actiunea pe care au suferit-o (actiune transmisa de la tastatura, mouse, etc.).
Incepând cu versiunea 1.1 a limbajului Java evenimentele se implementeaza ca obiecte instanta ale clasei **AWTEvent** sau ale subclaselor ei.
Un *eveniment* este produs de o actiune a utilizatorului asupra unui obiect grafic, deci evenimentele nu trebuie generate de programator. In schimb într-un program trebuie specificat codul care se executa la aparitia unui eveniment.
Interceptarea evenimentelor se realizeaza prin intermediul unor clase de tip *listener* (ascultator, consumator de evenimente), clase care sunt definite în pachetul **java.awt.event**. In Java, orice componenta poate "consuma" evenimentele generate de o alta componenta grafica. (vezi "Tratarea evenimentelor")

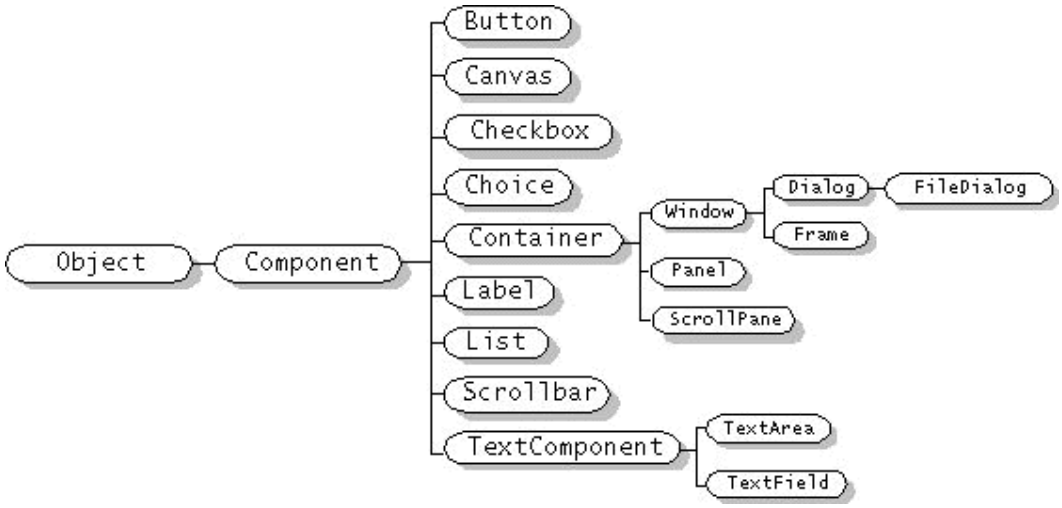
Exemplu: crearea unei ferestre ce contine doua butoane

```
import java.awt.*;
public class TestAWT1 {
	public static void main(String args[]) {

		//creez fereastra - un obiect de tip frame
		Frame f = new Frame("O fereastra");

		//setez modul de dipunere a ob. pe suprafata ferestrei
		f.setLayout(new FlowLayout());

		//creez cele doua butoane
		Button b1 = new Button("OK");
		Button b2 = new Button("Cancel");

		//adaug primul buton pe suprafata ferestrei
		f.add(b1);
		f.pack();

		//adaug al doile buton pe suprafata ferestrei
		f.add(b2);
		f.pack();

		//afisez fereastra (o fac vizibila)
		f.show();
	}
}
```

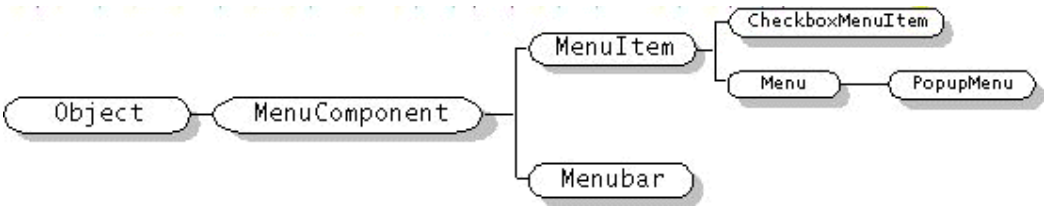
Dupa cum veti observa la executia acestui program, atât butoanele adaugate de noi cât si butonul de închidere a ferestrei sunt functionale, adica pot fi apasate, dar nu realizeaza nimic. Acest lucru se întâmpla deoarece nu am specificat nicaieri codul care trebuie sa se execute la apasarea acestor butoane.
De asemenea mai trebuie remarcat ca nu am specificat nicaieri dimensiunile ferestrei sau ale butoanelor si nici pozitiile în acestea sa fie plasate. Cu toate acestea ele sunt plasate unul lânga celalalt, fara sa se suprapuna iar suprafata fereastrei este suficient de mare cât sa cuprinda ambele obiecte. Aceste "fenomene" sunt provocate de un obiect special de tip `FlowLayout` care se ocupa cu gestionarea ferestrei si cu plasarea componentelor într-o anumita ordine pe suprafata ei.
Asadar, modul de aranjare nu este o caracteristica a suprafetei de afisare. Fiecare obiect de tip `Container`, sau o extensie a lui, are asociat un obiect care se ocupa cu dispunerea componentelor pe suprafata de afisare si care se numeste *gestionar de pozitionare (Layout Manager)*. (vezi "Gestionarea pozitionarii")

Componente AWT

Prin *componenta* vom întelege în continuare orice obiect care are o reprezentare grafica ce poate fi afisata pe ecran si care poate interactiona cu utilizatorul. Exemple de componente sunt ferestrele, butoanele, bare de defilare, etc. In general, toate componentele sunt definte de clase proprii ce se gasesc în pachetul `java.awt`, clasa **Component** fiind superclasa abstracta a tuturor acestor clase.
Ierarhia acestor clase este sumarizata în diagrama de mai jos.



Din cauza unor diferente esentiale în implementarea meniurilor pe diferite platforme de operare acestea nu au putut fi integrate ca obiecte de tip `Component`. Superclasa care descrie meniuri este **MenuComponent** iar ierarhia subclaselor sale este data în diagrama de mai jos:

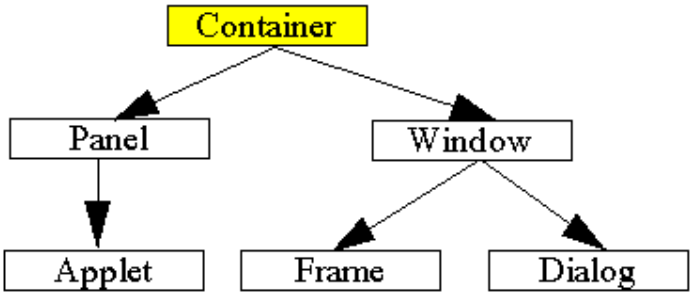


Asadar, majoritatea obiectelor grafice sunt subclase ale clasei `Component`, clasa care defineste generic o componenta grafica care poate interactiona cu utilizatorul. Singura exceptie o constituie meniurile care descind din clasa `MenuComponent`.

Suprafete de afisare (Clasa Container)

Curs 6

Crearea obiectelor grafice nu realizeaza automat si afisarea lor pe ecran. Mai întâi ele trebuie asezate pe o suprafata, care poate fi o fereastra sau suprafata unui applet, si vor deveni vizibile în momentul în care suprafata pe care sunt afisate va fi vizibila. O astfel de suprafata pe care se aseaza obiectele grafice se numeste *suprafata de afisare* sau *container* si reprezinta o instanta a unei clase obtinuta prin extensia superclasei **Container**. O parte din ierarhia a carei radacina este `Container` este prezentata în figura de mai jos:



Asadar, un container este folosit pentru a adauga componente pe suprafata lui. Componentele adaugate sunt memorate într-o lista iar pozitiile lor din aceasta lista vor defini ordinea de traversare "front-to-back" a acestora în cadrul containerului. Daca nu este specificat nici un index la adaugarea unei componente atunci ea va fi adaugata pe ultima pozitie a listei.

Adaugarea unei componente

Clasa `Container` pune la dispozitie metoda **add** pentru adaugarea unei componente pe o suprafata de afisare. O componenta nu poate apartine decât unui singur container, ceea ce înseamna ca pentru a muta un obiect dintr-un container în altul trebuie sa-l eliminam mai întâi de pe containerul initial. Eliminarea unei componente de pe un container se face cu metoda **remove**.

```
Frame f = new Frame("O fereastra");
Button b = new Button("OK");
f.add(b); //adauga butonul pe suprafata ferestrei
f.show();
```

Gestionarea pozitionarii

Sa consideram mai întai un exemplu de program Java care afiseaza 5 butoane pe o fereastra:

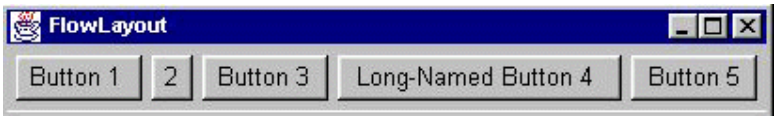
```
import java.awt.*;
public class TestLayout {
	public static void main(String args[]) {
		Frame f = new Frame("Grid Layout");
		f.setLayout(new GridLayout(3, 2)); //*

		Button b1 = new Button("Button 1");
		Button b2 = new Button("2");
		Button b3 = new Button("Button 3");
		Button b4 = new Button("Long-Named Button 4");
		Button b5 = new Button("Button 5");

		f.add(b1); f.add(b2); f.add(b3); f.add(b4); f.add(b5);
		f.pack();
		f.show();
	}
}
```

Fereastra afisata de acest program va arata astfel:

Sa modificam acum linia marcata cu '*' ca mai jos, lasând neschimbat restul programului:

```
Frame f = new Frame("Flow Layout");
f.setLayout(new FlowLayout());
```

Fereastra afisata dupa aceasta modificare va avea o cu totul altfel de dispunere a componentelor sale:



Motivul pentru care cele doua ferestre arata atât de diferit este ca folosesc gestionari de pozitionare diferiti: `GridLayout`, respectiv `FlowLayout`.

Un *gestionar de pozitionare (layout manager)* este un obiect care controleaza dimensiunea si aranjarea (pozitia) componentelor unui container. Asadar, modul de aranjare a componentelor pe o suprafata de afisare nu este o caracteristica a clasei `Container`. Fiecare obiect de tip `Container`, sau o extensie a lui (`Applet, Frame, Panel`) are asociat un obiect care se ocupa cu dispunerea componentelor pe suprafata sa : gestionarul de pozitionare.
Toate clasele care instantiaza obiecte pentru gestionarea pozitionarii implementeaza interfata **LayoutManager**.
La instantierea unui container se creeaza implicit un gestionar de pozitionare asociat acestui container. De exemplu pentru o fereastra (un obiect de tip `Window` sau o subclasa a sa) gestionarul implict este de tip `BorderLayout`, în timp ce pentru un container de tip `Panel` este o instanta a clasei `FlowLayout`.

Folosirea gestionarilor de pozitionare

Asa cum am vazut, orice container are un gestionar implicit de pozitionare - un obiect care implemeneaza interfata **LayoutManager**, acesta fiindu-i atasat automat la crearea sa. In cazul în care acesta nu corespunde necesittailor noastre el poate fi schimbat cu usurinta. Cei mai utilizati gestionari din pachetul `java.awt` sunt:

- FlowLayout
- BorderLayout
- GridLayout
- CardLayout
- GridBagLayout

Atasarea explicita a unui gestionar de pozitionare la un container se face cu metoda **setLayout** a clasei `Container`. Metoda poate primi ca parametru orice instanta a unei clase care implementeaza interfata `LayoutManager`. Secventa de atasare a unui gestionar pentru un container este:

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

```
sau, mai uzual :
container.setLayout(new FlowLayout());
```

Programele nu apeleaza în general metode ale gestionarilor de pozitionare iar în cazul când avem nevoie de obiectul gestionar îl putem obtine cu metoda **getLayout** din clasa `Container`.

Una din facilitatile cele mai utile oferite de gestionarii de pozitionare este rearanjarea componentele unui container atunci când acesta este redimesionat. Pozitiile si dimensiunile componentelor nu sunt fixe, ele fiind ajustate automat de catre gestionar la fiecare redimensionare astfel încât sa ocupe cât mai "estetic" suprafata de afisare.
Sunt însa situatii când dorim sa plasam componentele la anumite pozitii fixe iar acestea sa ramâna acolo chiar daca redimensionam containerul. Folosind un gestionar de pozitionare aceasta *pozitionare absoluta* a componentelor nu este posibila si deci trebuie cumva sa renuntam la gestionarea automata a containerul. Acest lucru se realizeaza prin trimitera argumentului **null** metodei `setLayout`:

```
		//pozitionare absoluta a componentelor in container
		container.setLayout(null);
```

Folosind pozitionarea absoluta, nu va mai fi suficient sa adaugam cu metoda `add` componentele în container ci va trebui sa specificam pozitia si dimensiunea lor - acest lucru era facut automat de gestionarul de pozitionare.

```
		container.setLayout( null );
		Button b = new Button("Buton");

		b.setSize(10, 10);
		b.setLocation (0, 0);
		b.add();
```

In general, se recomanda folosirea gestionarilor de pozitionare în toate situatiile când acest lucru este posibil, deoarece permit programului sa aiba aceeasi "înfatisare" indiferent de platforma si rezolutia pe care este rulat. Pozitionarea fixa poate ridica diverse probleme în acest sens.

Sa analizam în continuare pe fiecare din cei cinci gestionari amintiti anterior.

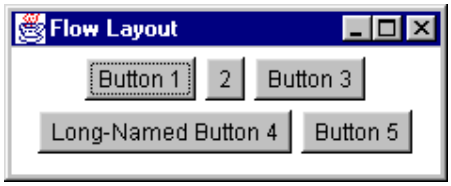
Gestionarul FlowLayout

Acest gestionar aseaza componentele pe suprafata de afisare în flux liniar, mai precis, componentele sunt adaugate una dupa alta pe linii, în limita spatiului disponibil. In momentul când o componenta nu mai încape pe linia curenta se trece la urmatoarea linie, de sus în jos.
Adaugarea componentelor se face de la stânga la dreapta pe linie, iar alinierea obiectelor în cadrul unei linii poate fi de trei feluri : la stânga, la drepata, centrate. Implicit componentele sunt centrate pe fiecare linie iar distanta implicita între componente este de 5 unitati pe verticala si 5 pe orizontala.
Este gestionarul implicit al containerelor derivate din clasa `Panel` deci si al applet-urilor.
Dimeniunile componentelor afisate sunt preluate automat de catre gestionar prin intermediul metodei `getPreferredSize`, implementata de toate componentele standard.

```
//Exemplu
import java.awt.*;
public class TestLayout {
	public static void main(String args[]) {
		Frame f = new Frame("Flow Layout");
		f.setLayout(new FlowLayout());
		Button b1 = new Button("Button 1");
		Button b2 = new Button("2");
		Button b3 = new Button("Button 3");
		Button b4 = new Button("Long-Named Button 4");
		Button b5 = new Button("Button 5");
		f.add(b1); f.add(b2); f.add(b3); f.add(b4); f.add(b5);
		f.pack();
		f.show();
	}
}
```

Componentele ferestrei vor fi afisate astfel:

Redimensionând fereastra astfel încât cele cinci butoane sa nu mai încapa pe o linie, ultimele dintre ele vor fi trecute pe linia urmatoare:

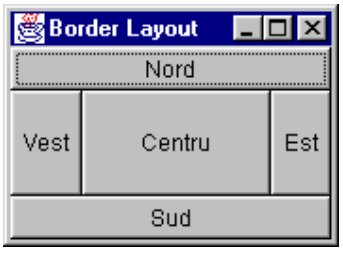


Gestionarul BorderLayout

Gestionarul `BorderLayout` împarte suprafata de afisare în cinci regiuni, corespunzatoare celor patru puncte cardinale si centrului. O componenta poate fi plasata în oricare din aceste regiuni, dimeniunea componentei fiind calculata astfel încât sa ocupe întreg spatiul de afisare oferit de regiunea respectiva. Pentru a adauga mai multe obiecte grafice într-una din cele cinci zone, ele trebuie grupate în prealabil într-un panel, care va fi amplasat apoi în regiunea dorita.(vezi "Gruparea componentelor - clasa Panel")
Asadar, la adaugarea unei componente pe o suprafata gestionata de `BorderLayout`, metoda `add` va mai primi pe lânga numele componentei si zona în care aceasta va fi amplasata, acesta fiind apecificata prin una din constantele clasei `BorderLayout`: **`NORTH, SOUTH, EAST, WEST, CENTER`** .
Este gestionarul implicit pentru toate containerele care descind din clasa `Window`, deci este gestionarul implicit al ferestrelor Java.

```
//Exemplu
import java.awt.*;
public class TestBorderLayout {
	public static void main(String args[]) {
		Frame f = new Frame("Border Layout");
		f.setLayout(new BorderLayout());//poate sa lipseasca

		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.pack();

		f.show();
	}
}
```

Cele cinci butoane ale ferestrei vor fi afisate astfel:



La redimensionarea ferestrei se pot observa urmatoarele lucruri: nordul si sudul se redimensioneaza doar pe orizontala, estul si vestul doar pe verticala, în timp ce centrul se redimensioneaza atât pe orizontala cât si pe verticala. Redimensionarea componentelor se face astfel încât ele ocupa toata zona containerului din care fac parte.

Gestionarul GridLayout

Gestionarul `GridLayout` organizeaza containerul ca un tabel cu rânduri si coloane, componentele fiind plasate în casutele tabelului de la stânga la dreapta începând cu primul rând. Casutele tabelului au dimensiuni egale iar o

componenta poate ocupa doar o singura casuta.
Numarul de linii si coloane poate fi specificat în constructorul gestionarului dar poate fi modificat si ulterior prin metodele **setRows** si **setCols**. De asemenea, distanta între componente pe orizontala si distanta între rândurile tabelului pot fi specificate în constructor sau stabilite ulterior.
Acest tip de gestionar poate fi util în implementarea unor componente de tip calculator, în care numerele si operatiile sunt afisate prin intermediul unor butoane dispuse sub forma unei grile.

```
//Exemplu
import java.awt.*;
public class TestGridLayout {
       public static void main(String args[]) {
              Frame f = new Frame("Grid Layout");
              f.setLayout(new GridLayout(3, 2));

              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.pack();
              f.show();
       }
}
```

Cele sase butoane ale ferestrei vor fi pe trei rânduri si doua coloane astfel:

Redimensionarea ferestrei va determina redimensionarea tuturor componentelor.

Gestionarul CardLayout

Gestionarul `CardLayout` trateaza componentele adaugate pe suprafata într-o maniera asemanatoare cu cea a dispunerii cartilor de joc înntr-un pachet. Suprafata de afisare poate fi asemanata cu pachetul de carti iar fiecare componenta este o carte din pachet. La un moment dat numai o singura componenta este vizibila ("cea de deasupra"). Clasa dispune de metode prin care sa poata fi afisata o anumita componenta din pachet, sau sa se poata parcurge secvential pachetul, ordinea în care componentele se gasesc în pachet fiind interna gestionarului.
Acest gestionar este util pentru implementarea unor cutii de dialog de tip tab, în care pentru o gestionare mai eficienta a spatiului, componentele sunt grupate în pachete, la un moment dat utilizatorul interactionând cu un singur pachet, celelate fiind ascunse.

Gestionarul GridBagLayout

Este cel mai complex si flexibil gestionar de pozitionare din Java. La fel ca în cazul gestionarului `GridLayout`, suprafata de afisare este considerata ca fiind un tabel, însa, spre deosebire de acesta, numarul de linii si de coloane sunt determinate automat, în functie de componentele amplasate pe suprafata de afisare. De asemenea, în functie de componentele gestionate, dimensiunile casutelor pot fi diferite, cu singurele restrictii ca pe aceeasi linie casutele trebuie sa aiba aceeasi înaltime, iar pe coloana trebuie sa aiba aceeasi latime.
Spre deosebire de `GridLayout`, o componenta poate ocupa mai multe celule adiacente, chiar de dimensiuni diferite, zona ocupata fiind referita prin "regiunea de afisare" a componentei respective.

Pentru a specifica modul de afisare a unei componente, acesteia îi este asociat un obiect de tip **GridBagConstraints**, în care se specifica diferite proprietati ale componentei referitoare la regiunea sa de afisare si la modul în care va fi plasata în aceasta regiune. Legatura dintre o componenta si un obiect `GridBagConstraints` se realizeaza prin metode **setConstraints**:

```
GridBagLayout gridBag = new GridBagLayout();
container.setLayout(gridBag);
GridBagConstraints c = new GridBagConstraints();
. . .
//se specifica proprietatile referitoare la afisarea unei
componente
gridBag.setConstraints(componenta, c);
container.add(componenta);
```

Asadar, înainte de a adauga o componenta pe suprafata unui container care are un gestionar de tip `GridBagLayout`, va trebui sa specificam anumiti parametri (constrângeri) referitori la cum va fi plasata componenta respectiva. Aceste constrângeri vor fi specificate prin intermediul unui obiect de tip `GridBagConstraints`, care poate fi folosit pentru mai multe componente care au aceleassi cosntrângeri de afisare:

```
gridBag.setConstraints(componenta1, c);
gridBag.setConstraints(componenta2, c);
. . .
```

Gruparea componentelor (Clasa Panel)

Plasarea componentelor direct pe suprafata de afisare poate deveni incomoda în cazul în care avem multe obiecte grafice. Din acest motiv se recomanda gruparea obiectelor grafice înrudite ca functii astfel încât sa putem fi siguri ca, indiferent de gestionarul de pozitionare al suprafetei de afisare, ele se vor gasi împreuna. Gruparea componentelor se face în **panel-uri**.

Un panel este cel mai simplu model de container. El nu are o reprezentare vizibila, rolul sau fiind de a oferi o suprafata de afisare pentru componente grafice, inclusiv pentru alte panel-uri.
Clasa care instantiaza aceste obiecte este **Panel**, extensie a superclasei `Container`.
Pentru a aranja corespunzator componentele grupate într-un panel, acestuia i se poate specifica un gestionar de pozitionare anume, folosind metoda `setLayout`. Gestionarul implicit pentru containerele de tip `Panel` este `FlowLayout`.

Asadar, o aranjare eficienta a componentelor unei ferestre înseamna:

- gruparea componentelor "înfratite" (care nu trebuie sa fie despartie de gestionarul de pozitionare al ferestrei) în panel-uri;
- aranjarea componentelor unui panel, prin specificarea acestuia a unui gestionar de pozitionare corespunzator
- aranjarea panel-urilor pe suprafata ferestrei, prin specificarea gestionarului de pozitionare al ferestrei.

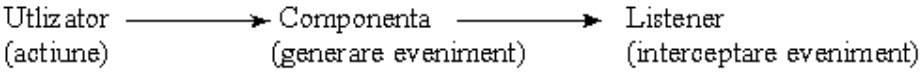
```
//Exemplu
import java.awt.*;
public class TestPanel {
       public static void main(String args[]) {
              Frame f = new Frame("Panel");

              Panel panel = new Panel();
              panel.setLayout(new FlowLayout());
              panel.add(new Label("Text:"));
              panel.add(new TextField("", 20));
              panel.add(new Button("Reset"));

              f.add(panel, BorderLayout.NORTH);
              f.add(new Button("OK"), BorderLayout.EAST);
              f.add(new Button("Cancel"), BorderLayout.WEST);
              f.pack();

              f.show();
```

```
        }
}
```

Tratarea evenimentelor

Un *eveniment* este produs de o actiune a utilizatorului asupra unei componente grafice si reprezinta mecanismul prin care utilizatorul comunica efectiv cu programul. Exemple de evenimente sunt: apasarea unui buton, modificarea textului într-un control de editare, închiderea, redimensionarea unei ferestre, etc. Componentele care genereaza anumite evenimente se mai numesc si *surse de evenimente*.

Interceptarea evenimentelor generate de componentele unui program se realizeaza prin intermediul unor clase de tip **listener** (ascultator, consumator de evenimente). In Java, orice obiect poate "consuma" evenimentele generate de o anumita componenta grafica.



Asadar, pentru a scrie cod care sa se execute în momentul în care utilizatorul interactioneaza cu o componenta grafica trebuie sa facem urmatoarele lucruri:

- sa scriem o clasa de tip listener care sa "asculte" evenimentele produse de acea componenta si în cadrul acestei clase sa implementam metode specifice pentru tratarea lor;
- sa comunicam componentei sursa ca respectiva clasa îi "asculta" evenimentele pe care le genereaza, cu alte cuvinte sa înregistram acea clasa drept "consumator" al evenimentelor produse de componenta respectiva.

Evenimentele sunt, ca orice altceva în Java, obiecte. Clasele care descriu aceste obiecte se împart în mai multe tipuri în functie de componenta care le genereaza, mai precis în functie de actiunea utilizatorului asupra acesteia. Pentru fiecare tip de eveniment exista o clasa care instantiaza obiecte de acel tip; de exemplu: evenimentul generat de actionarea unui buton este implementat prin clasa `ActionEvent`, cel generat de modificarea unui text prin clasa TextEvent, etc. Toate aceste clase au ca superclasa comuna clasa **AWTEvent**. Lista completa a claselor care descriu evenimente va fi data într-un tabel în care vom specifica si modalitatile de utilizare ale acestora.

O clasa consumatoare de evenimente (listener) poate fi orice clasa care specifica în declaratia sa ca doreste sa asculte evenimente de un anumit tip. Acest lucru se realizeaza prin implementarea unei interfete specifice fiecarui tip de eveniment. Astfel, pentru ascultarea evenimentelor de tip `ActionEvent` clasa respectiva trebuie sa implementeze interfata `ActionListener`, pentru `TextEvent` interfata care trebuie implementata este `TextListener`, etc. Toate aceste interfete au suprainterfata comuna **EventListener**.

```
class AscultaButoane implements ActionListener
class AscultaTexte implements TextListener
```

Intrucât o clasa poate implementa oricâte interfete ea va putea sa asculte evenimente de mai multe tipuri:

```
class Ascultator implements ActionListener, TextListener
```

Vom vedea în continuare metodele fiecarei interfete pentru a sti ce trebuie sa implementeze o clasa consumatoare de evenimente.
Asa cum spus mai devreme, pentru ca evenimentele unei componente sa fie interceptate de catre o instanta a unei clase ascultator, aceasta clasa trebuie înregistrata în lista ascultatorilor componentei respective. Am spus lista, deoarece evenimentele unei componente pot fi ascultate de oricâte clase - cu conditia ca acestea sa fie înregistrate la componenta respectiva. Inregistrarea unei clase în lista ascultatorilor unei componente se face cu metode din clasa `Component` de tipul **addXXXListener**, iar eliminarea ei din aceasta lista cu **removeXXXListener**m unde *XXX* reprezenta tipul evenimentului.

Exemplu de tratare a evenimentelor

Inainte de a detalia aspectele prezentate mai sus, sa consideram un exemplu de tratare a evenimentelor. Vom crea o fereastra care sa contina doua butoane cu numele "OK", repectiv "Cancel". La apasarea fiecarui buton vom scrie pe bara titlu a ferestrei mesajul " Ati apasat butonul ...".

```
//Exemplu:Ascultarea evenimentelor de la doua butoane
import java.awt.*;
import java.awt.event.*;

class Fereastra extends Frame {
       public Fereastra(String titlu) {
              super(titlu);
       }
       public void initializare() {
              setLayout(new FlowLayout());       //se stabileste gestionarul
              setSize(200, 100);          //se dimensioneaza fereastra

              Button b1 = new Button("OK");
              add(b1);                           //se adauga primul buton
              Button b2 = new Button("Cancel");
              add(b2);                           //se adauga al doilea buton

              Ascultator listener = new Ascultator(this);
              b1.addActionListener(listener);
              b2.addActionListener(listener);
              //ambele butoane sunt ascultate de obiectul "listener"
              //instanta a clasei Ascultator, definita mai jos
       }
}

class Ascultator implements ActionListener {
       private Fereastra f;
       public Ascultator(Fereastra f) {
              this.f = f;
       }
       //metoda interfetei ActionListener
       public void actionPerformed(ActionEvent e) {
              String command = e.getActionCommand();
              //numele comenzii este numele butonului apasat
              System.out.println(e.toString());
              if (command.equals("OK"))
                     f.setTitle("Ati apasat OK");
              else
              if (command.equals("Cancel"))
                     f.setTitle("Ati apasat Cancel");
       }
}

public class TestEvent { //fereastra principala
       public static void main(String args[]) {
              Fereastra f = new Fereastra("ActionEvent");
              f.initializare();
              f.show();
       }
}
```

Nu este obligatoriu sa definim clase speciale pentru ascultarea evenimentelor. In exemplul de mai sus am definit o clasa speciala "Ascultator" pentru a intercepta evenimentele produse de cele doua butoane si din acest motiv a trebuit sa trimitem ca parametru acestei clase instanta la fereastra noastra. Mai corect ar fi fost sa folosim chiar clasa "Fereastra" pentru a-si asculta evenimentele produse de componentele sale:

```
class Fereastra extends Frame implements ActionListener{
```

```
	public Fereastra(String titlu) {
		super(titlu);
	}
	public void initializare() {
		. . .
		b1.addActionListener(this);
		b2.addActionListener(this);
		//ambele butoane sunt ascultate chiar din clasa Fereastra
		//deci ascultatorul este instanta curenta: this
	}

	public void actionPerformed(ActionEvent e) {
		String command = e.getActionCommand();
		//numele comenzii este numele butonului apasat
		System.out.println(e.toString());
		if (command.equals("OK"))
			this.setTitle("Ati apasat OK");
		else
		if (command.equals("Cancel"))
			this.setTitle("Ati apasat Cancel");
	}
}
. . .
```

Asadar, orice clasa poate asculta evenimnte de orice tip cu conditia sa implementeze interfetele specifice acelor evenimente.

Tipuri de evenimente si componentele care le genereaza

In tabelul de mai jos sunt prezentate în stânga tipurile de evenimente si interfetele iar în dreapta lista componentelor ce pot genera evenimente de acel tip precum si o scurta explicatie despre motivul care le provoaca.

Eveniment/Interfata	Componente care genereaza acest eveniment
ActionEvent ActionListener	Button, List, TextField, MenuItem, CheckboxMenuItem, Menu, PopupMenu Actiuni asupra unui control
AdjustmentEvent AdjustmentListener	Scrollbar si orice clasa care implementeaza interfata Adjustable Modificarea unei valori variind între doua limite
ComponentEvent ComponentListener	Component si subclasele sale Redimensionari, deplasari, ascunderi ale componentelor
ContainerEvent ContainerListener	Container si subclasele sale Adaugarea, stergerea componentelor la un container
FocusEvent FocusListener	Component si subclasele sale Preluarea, pierderea focusului de catre o componenta
KeyEvent KeyListener	Component si subclasele sale Apasarea, eliberarii unei taste când focusul este pe o anumita componenta.
MouseEvent MouseListener	Component si subclasele sale Click, apasare, eliberare a mouse-ului pe o componenta, intrarea, iesirea mouse-ului pe/de pe suprafata unei componente
MouseEvent MouseMotionListener	Component si subclasele sale Miscarea sau "târârea" (drag) mouse-ului pe suprafata unei componente
WindowEvent WindowListener	Window si subclasele sale Dialog, FileDialog, Frame Inchiderea, maximizarea, minimizarea, redimensionarea unei ferestre
ItemEvent ItemListener	Checkbox, CheckboxMenuItem, Choice, List si orice clasa care implementeaza interfata ItemSelectable

	Selectia, deselecttia unui articol dintr-o lista sau meniu.
TextEvent TextListener	Orice clasa derivata din TextComponent cum ar fi : TextArea, TextField Modificarea textului dintr-o componenta de editare a textului

Evenimente suportate de o componenta

Urmatorul tabel prezinta o clasificare a evenimentelor în functie de componentele care le suporta:

Componenta	Evenimente suportate de componenta
Adjustable	AdjustmentEvent
Applet	ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Button	ActionEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Canvas	FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Checkbox	ItemEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
CheckboxMenuItem	ActionEvent, ItemEvent
Choice	ItemEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Component	FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Container	ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Dialog	ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
FileDialog	ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Frame	ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Label	FocusEvent, KeyEvent, MouseEvent, ComponentEvent
List	ActionEvent, FocusEvent, KeyEvent, MouseEvent, ItemEvent, ComponentEvent
Menu	ActionEvent
MenuItem	ActionEvent
Panel	ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
PopupMenu	ActionEvent
Scrollbar	AdjustmentEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
ScrollPane	ContainerEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
TextArea	TextEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
TextComponent	TextEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
TextField	ActionEvent, TextEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent
Window	ContainerEvent, WindowEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent

Metodele interfetelor de tip "listener"

Orice clasa care trateaza evenimente trebuie sa implementeze obligatoriu metodele interfetelor corespunzatoare evenimentelor pe care le trateaza. Tabelul de mai jos prezinta, pentru fiecare interfata, metodele puse la dispozitie si care trebuie implementate de clasa ascultator.

Interfata	Metodele interfetei
ActionListener	`actionPerformed(ActionEvent)`
AdjustmentListener	`adjustmentValueChanged(AdjustmentEvent)`

ComponentListener	componentHidden(ComponentEvent) componentShown(ComponentEvent) componentMoved(ComponentEvent) componentResized(ComponentEvent)
ContainerListener	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
FocusListener	focusGained(FocusEvent) focusLost(FocusEvent)
KeyListener	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)
MouseListener	mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent)
MouseMotionListener	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
WindowListener	windowOpened(WindowEvent) windowClosing(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent)
ItemListener	itemStateChanged(ItemEvent)
TextListener	textValueChanged(TextEvent)

Folosirea adaptorilor si a claselor interne în tratarea evenimentelor

Am vazut ca o clasa care trateaza evenimente de un anumit tip trebuie sa implementeze interfata corespunzatoare acelui tip. Aceasta înseamna ca trebuie sa implementeze obligatoriu toate metodele definite de acea interfata, chiar daca nu specifica nici un cod pentru unele dintre ele. Sunt însa situatii când acest lucru este suparator, mai ales atunci când nu ne intereseaza decât o singura metoda a interfetei.
Un exemplu sugestiv este urmatorul: o fereastra care nu are specificat cod pentru tratarea evenimentelor sale nu poate fi închisa cu butonul standard marcat cu 'x' din coltul dreapta sus si nici cu combinatia de taste Alt+F4. Pentru a realiza acest lucru trebuie interceptat evenimentul de închidere a ferestrei în metoda `windoClosing` si apelata apoi metoda `dispose` de închidere a ferestrei, eventual umata de iesirea din program, în cazul când este vorba de fereastra principala a aplicatiei. Aceasta înseamna ca trebuie sa implementam interfata `WindowListener` care are nu mai putin de **sapte** metode.

```
//Crearea unei ferestre cu ascultarea evenimentelor sale
//folosind implementarea directa a interfetei WindowListener

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

class Fereastra extends Frame implements WindowListener {
       public Fereastra(String titlu) {
              super(titlu);
              this.addWindowListener(this);
       }

       //metodele interfetei WindowListener
```

```
        public void windowOpened(WindowEvent e) {}
        public void windowClosing(WindowEvent e) {
                dispose();           //inchid fereastra
                System.exit(0);      //termin programul
        }
        public void windowClosed(WindowEvent e) {}
        public void windowIconified(WindowEvent e) {}
        public void windowDeiconified(WindowEvent e) {}
        public void windowActivated(WindowEvent e) {}
        public void windowDeactivated(WindowEvent e) {}
}

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

Observati ca trebuie sa implementam toate metodele interfetei, chiar daca nu scriem nici un cod pentru ele. Singura metoda care ne intereseaza este `windowClosing` în care specificam ce trebuie facut atunci când utilizatorul doreste sa închida fereastra.
Pentru a evita scrierea inutila a acestor metode exista o serie de clase care implementeaza interfetele de tip "listener" fara a specifica nici un cod pentru metodele lor. Aceste clase se numesc *adaptori*.

Folosirea adaptorilor

Un *adaptor* este o clasa abstracta care implementeaza o interfata de tip "listener". Scopul unei astfel de clase este ca la crearea unui "ascultator" de evenimente, în loc sa implementa o anumita interfata si implicit toate metodele sale, sa extindem adaptorul corespunzator interfetei respective (daca are!) si sa supradefinim doar metodele care ne intereseaza (cele în care vrem sa scriem o anumita secventa de cod).
De exemplu, adaptorul interfetei `WindowListener` este `WindowAdapter` iar folosirea acestuia este data în exemplul de mai jos:

```
//Crearea unei ferestre cu ascultarea evenimentelor sale
//folosind extinderea clasei WindowAdapter

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

class Fereastra extends Frame {
        public Fereastra(String titlu) {
                super(titlu);
                this.addWindowListener(new Ascultator());
        }
}

class Ascultator extends WindowAdapter {
        //suprdefinim metodele care ne intereseaza
        public void windowClosing(WindowEvent e) {
                System.exit(0);
        }
}
```

Avantajul clar al acestei modaitati de tratare a evenimentelor este reducerea codului programului, acesta devenind mult mai usor lizibil. Insa exista si doua dezavantaje majore. Dupa cum ati observat, fata de exemplul anterior clasa "Fereastra" nu poate extinde `WindowAdapter` deoarece ea extinde deja clasa `Frame` si din acest motiv am construi o noua clasa numita "Ascultator". Vom vedea însa ca acest dezavantaj poate fi eliminat prin folosirea unei clase interne.
Un alt dezavantaj este ca orice greseala de sintaxa în declararea unei metode a interfetei nu va produce o eroare de compilare dar nici nu va supradefini metoda interfetei ci, pur si simplu, va crea o metoda a clasei respective.

```
class Ascultator extends WindowAdapter {
        // in loc de windowClosing scriem WindowClosing
        // nu supradefinim vreo metoda a clasei WindowAdapter
        // nu da nici o eroare
        // nu face nimic !
        public void WindowClosing(WindowEvent e) {
                System.exit(0);
        }
}
```

In tabelul de mai jos sunt dati toti adaptorii interfetele de tip "listener" - se oberva ca o interfata `XXXListener` are un adaptor de tipul `XXXAdapter`. Interfetele care nu au un adaptor sunt cele care definesc o singura metoda si prin urmare crearea unei clase adaptor nu îsi are rostul.

Interfata "listener"	Adaptor
ActionListener	nu are
AdjustmentListener	nu are
ComponentListener	ComponentAdapter
ContainerListener	ContainerAdapter
FocusListener	FocusAdapter
ItemListener	nu are
KeyListener	KeyAdapter
MouseListener	Mouse
MouseMotionListener	MouseMotionAdapter
TextListener	nu are
WindowListener	WindowAdapter

Folosirea claselor interne (anonime)

Stim ca o clasa interna este o clasa declarata în cadrul altei clase iar clasele anonime sunt acele clase interne folosite doar pentru instantierea unui singur obiect de acel tip. Un exemplu tipic de folosire a lor este instantierea adaptorilor direct în corpul unei clase care contine componente ale caror evenimente trebuie interceptate. Clasa "Fereastra" din exemplul anterior poate fi scrisa astfel:

```
class Fereastra extends Frame {
        public Fereastra(String titlu) {
                super(titlu);
                this.addWindowListener(new WindowAdapter() {
                        //corpul clasei anonime
                        public void windowClosing(WindowEvent e) {
                                System.exit(0);
                        }
                });
        }
}
```

Observati cum codul programului a fost redus substantial prin folosirea unui adaptor si a unei clase anonime.

Folosirea ferestrelor

Dupa cum am vazut suprafetele de afisare ale componentelor grafice (containerele) sunt extensii ale clasei `Container`. O categorie aparte a acestor containere o reprezinta ferestrele. Spre deosebire de un applet care îsi poate plasa componentele direct pe suprafata de afisare a browser-ului în care ruleaza, o aplicatie intependenta are nevoie de propriile ferestre pe care sa faca afisarea componentelor sale grafice. Pentru dezvoltarea aplicatiilor care folosesc grafica se vor folosi clasele **Window** si subclasele sale directe **Frame** si **Dialog**.

Clasa Window

Clasa `Window` este rar utilizata în mod direct. Ea permite crearea unor ferestre top-level care nu au chenar si nici bara de meniuri. Pentru a crea ferestre mai complexe se utilizeaza clasele `Frame` si `Dialog`.
Metodele mai importante ale clasei `Window` (mostenite de toate subclasele sale) sunt date în tabelul de mai jos:

`void dispose()`	Distruge (închide) fereastra si si elibereaza toate resursele acesteia
`Component getFocusOwner()`	Returneaza componenta ferestrei care are focus-ul daca si numai daca fereastra este activa
`Window getOwnedWindows()`	Returneaza un vector cu toate ferestrele subclase ale ferestrei respective
`Window getOwner()`	Returneaza parintele (superclasa) ferestrei
`void hide()`	Face fereastra invizibila fara a o distruge însa. Pentru a redeveni vizibila apelati metoda show
`boolean isShowing()`	Testeaza daca fereastra este vizibila sau nu
`void pack()`	Redimensioneaza automat fereastra la o suprafata optima care sa cuprinda toate componentele sale. Trebuie apelata, în general, dupa adaugarea tuturor componentelor pe suprafata ferestrei.
`void show()`	Face vizibila o fereastra creata. Implicit o fereastra nou creata nu este vizibila.
`void toBack()` `void toFront()`	Trimite fereastra în spatele celorlalte ferestre deschise Aduce fereastra în fata celorlalte ferestre deschise

Clasa Frame

Este subclasa directa a clasei `Window`. si este folosita pentru crearea de ferestre independente si functionale, eventual continând bare de meniuri. Orice aplicatie grafica independenta trebuie sa aiba cel putin o fereastra, numita si *fereastra principala*, care va fi afisata la pornirea programului.
Constructorii clasei Frame sunt:

`Frame ()`
`Construieste o fereastra, fara titlu, initial invizibila.`
`Frame(String title)`
`Construieste o fereastra, cu titlul specificat, initial invizibila.`

Asadar, o fereastra nou creata este invizibila. Pentru a fi facuta vizibila se va apela metoda `show` definita în superclasa `Window`. In exemplul de mai jos este construita si afisata o fereasta cu titlul "O fereastra".

```
//Crearea unei ferestre
import java.awt.*;
public class TestFrame {
	public static void main(String args[]) {
		Frame f = new Frame("O fereastra");
		f.show();
	}
}
```

Crearea ferestrelor prin instantierea obiectelor de tip `Frame` este mai putin uzuala. De obicei, ferestrele unui program vor fi definite în clase separate care extind clasa `Frame`, ca în exemplul de mai jos:

```
import java.awt.*;
class Fereastra extends Frame{
	//constructorul
	public Fereastra(String titlu) {
		super(titlu);
	}
	void initializare() {
		. . .
```

```
                //adaugam componentele ferestrei
        }
}
public class TestFrame {
        public static void main(String args[]) {
                Fereastra f = new Fereastra("O fereastra");
                f.initializare();
                f.show();
        }
}
```

Gestionarul de pozitionare implicit al clasei `Window` este `BorderLayout`. Din acest motiv, în momentul în care fereastra este creata dar nici o componenta grafica nu este pusa pe suprafata ei, suprafata de afisare a feretrei va fi nula. Acelasi efect îl vom obtine daca o redimenionam si apelam apoi metoda `pack` care determina dimeniunea suprafetei de afisare în functie de componentele grafice afisate pe ea.
Se observa de asemenea ca butonul de închidere a ferestrei nu este functional. Interceptarea evenimentelor se face prin implementarea interfetei **WindowListener** si prin adaugarea în lista ascultatorilor ferestrei (uzual) chiar a obiectului care implementeaza fereastra sau prin folosirea unor adaptori si clase anonime.

Metodele mai folosite ale clasei Frame sunt date în tabelul de mai jos:

`static Framest getFrames()`	Metoda statica ce returneaza lista tuturor ferestrelor deschise ale unei aplicatii
`Image getIconImage()` `void setIconImage(Image img)`	Afla/seteaza imaginea(iconita) care sa fie afisata atunci când fereastra este minimizata
`MenuBar getMenuBar()` `void setMenuBar(MenuBar mb)`	Afla/seteaza bara de meniuri a ferestrei (vezi "Folosirea meniurilor")
`int getState()` `void setState(int s)`	Returneaza/seteaza starea ferestrei. O fereastra se poat gasi în doua stari, descrise de constantele: Frame.ICONIFIED (daca este minimizata) Frame.NORMAL (daca nu este minimizata).
`String getTitle()` `void setTitle()`	Afla/seteaza titlul ferestrei
`boolean isResizable()` `void setResizable(boolean r)`	Determina/stabileste daca fereastra poate fi redimenionata de utilizator.

Clasa Dialog

Toate interfetele grafice ofera un tip special de ferestre destinate preluarii datelor de la utilizator. Acestea se numesc *ferestre de dialog* sau *casete de dialog* si sunt implementate prin intermediul clasei **Dialog**, subclasa directa a clasei `Window`.
Diferenta majora între ferestrele de dialog si ferestrele normale (obiecte de tip `Frame`) consta în faptul ca o fereastra de dialog este dependenta de o alta fereastra (normala sau tot fereastra dialog), numita si *fereastra parinte*. Cu alte cuvinte, ferestrele de dialog nu au o existenta de sine statatoare.
Când fereastra parinte este distrusa sunt distruse si ferestrele sale de dialog, când este minimizata ferestrele sale de dialog sunt facute invizibile iar când este maximizata acestea sunt aduse la starea în care se gaseau în momentul minimizarii ferestrei parinte.

Ferestrele de dialog pot fi de doua tipuri:

- **modale**: care blocheaza accesul la fereastra parinte în momentul deschiderii lor - de exemplu, ferestre de introducere a unor date, de alegere a unui fisier în vederea deschideriii, de selectare a unei optiuni, mesaje de avertizare, etc;
- **nemodale**: care nu blocheaza fluxul de intrare catre fereastra parinte - de exemplu, ferestrele de cautare a unui cuvânt într-un fisier.

Implicit o fereastra de dialog este nemodala si invizibila.
Constructorii clasei Dialog sunt:

```
Dialog(Frame parinte)
Dialog(Frame parinte, String titlu)
Dialog(Frame parinte, String titlu, boolean modala)
Dialog(Frame parinte, boolean modala)
Dialog(Dialog parinte)
Dialog(Dialog parinte, String titlu)
Dialog(Dialog parinte, String titlu, boolean modala)
```

unde "parinte" reprezina o instanta ferestrei parinte, "titlu" reprezinta titlul ferestrei iar prin argumentul "modala" specificam daca fereastra de dialog creata va fi modala (true) sau nemodala (false - valoarea implicita).

Pe lânga metodele mostenite de la superclasa Window clasa Dialog mai contine metodele:

`boolean isModal()`	Determina daca fereastra de dialog este modala sau nu.
`void setModal(boolean modala)`	Specifica tipul ferestrei de dialog: modala (true) sau nemodala (false)

Crearea unei ferestre de dialog este relativ simpla si se realizeaza prin crearea unei clase care sa extinda clasa `Dialog`. Mai complicat este însa modul în care se implementeaza comunicarea între fereastra de dialog si fereastra parinte, pentru ca aceasta din urma sa poata folosi datele introduse (sau optiunea specificata) în caseta de dialog. Exista doua abordari generale :

- obiectul care reprezinta dialogul poate sa capteze evenimentele de la componentele de pe suprafata sa si sa sa seteze valorile unor variabile ale ferestrei parinte în momentul în care dialogul este încheiat sau
- obiectul care creeaza dialogul (fereastra parinte) sa se înregistreze ca ascultator al evenimentelor de la butoanele care determina încheierea dialogului, iar fereastra de dialog sa ofere metode publice prin care datele introduse sa fie preluate din exterior.

Sa cream, de exemplu, o fereastra de dialog modala pentru introducerea unui sir de caractere. Fereastra principala a aplicatiei va fi parintele casetei de dialog, va primi sirul de caractere introdus si îsi va modifica titlul ca fiind sirul primit. Deschiderea ferestrei de dialog se va face la apasarea unui buton al ferestrei principale numit "Schimba titlul". Dialogul va mai avea doua butoane OK si Cancel pentru terminarea sa cu confirmare, respectiv renuntare. Cele doua ferestre vor arata ca în imaginile de mai jos


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

```
//Fereastra principala a aplicatiei
class FerPrinc extends Frame implements ActionListener{
	public FerPrinc(String titlu) {
		super(titlu);
		this.addWindowListener(new WindowAdapter() {
			public void windowClosing(WindowEvent e) {
				System.exit(0);
			}
		});
	}
	public void initializare() {
		setLayout(new FlowLayout());
		setSize(300, 100);
		Button b = new Button("Schimba titlul");
		add(b);
		b.addActionListener(this);
	}

	//metoda interfetei ActionListener
	public void actionPerformed(ActionEvent e) {
		FerDialog d = new FerDialog(this, "Titlu", true);
		//astept sa se inchida fereastra modala de dialog
		if (d.raspuns == null) return;
		setTitle(d.raspuns);
	}
}

//fereastra de dialog
class FerDialog extends Dialog implements ActionListener {
	public String raspuns = null;
	private TextField text;
	public FerDialog(Frame parinte, String titlu, boolean modala) {
		super(parinte, titlu, modala);
		this.addWindowListener(new WindowAdapter() {
			public void windowClosing(WindowEvent e) {
				raspuns = null;
				dispose();
			}
		});
		setLayout(new FlowLayout());
		Button ok, cancel;
		ok = new Button("OK");
		cancel = new Button("Cancel");
		text = new TextField("", 50);
		add(cancel);add(text);add(ok);pack();
		ok.addActionListener(this);
		cancel.addActionListener(this);
		show();
	}

	//metoda interfetei ActionListener
	public void actionPerformed(ActionEvent e) {
		//vad ce buton a fost apasat
		String buton = e.getActionCommand();
		if (buton.equals("OK"))
			raspuns	= text.getText();
		else
		if (buton.equals("Cancel"))
```

```
                    raspuns       = null;
                dispose();
        }
}

//clasa principala
public class TestDialog {
        public static void main(String args[]) {
                FerPrinc f = new FerPrinc("Fereastra principala");
                f.initializare();
                f.show();
        }
}
```

Clasa FileDialog

Pachetul `java.awt` pune la dispozitie si un tip de fereastra de dialog folosita pentru încarcarea / salvarea fisierelor : clasa **FileDialog**, subclasa directa a clasei `Dialog`. Instantele acestei clase au un comportament comun dialogurilor de acest tip de pe majoritatea platformelor de lucru, dar forma în care vor fi afisate este specifica platformei pe care ruleaza aplicatia.
Constructorii clasei sunt:

```
FileDialog(Frame parinte)
FileDialog(Frame parinte, String titlu)
FileDialog(Frame parinte, String titlu, boolean mod)
```

unde "parinte" reprezina o instanta ferestrei parinte, "titlu" reprezinta titlul ferestrei iar prin argumentul "mod" specificam daca încarcam sau salvam un fisier; valorile pe care le poate lua acest argument sunt **FileDialog.LOAD** (pentru încarcare), respectiv **FileDialog.SAVE** (pentru salvare).

```
//dialog pentru incarcarea unui fisier
new FileDialog(mainWin, "Alegere fisier", FileDialog.LOAD);

//dialog pentru salvarea unui fisier
new FileDialog(mainWin, "Salvare fisier", FileDialog.SAVE);
```

La crearea unui obiect `FileDialog` acesta nu este implicit vizibil. Daca afisarea sa se face cu **show** caseta de dialog va fi modala. Daca afisarea se face cu **setVisible(true)** va fi nemodala. Dupa selectarea unui fisier ea va fi facuta automat invizibila.

Pe lânga metodele mostenite de la superclasa `Dialog` clasa `FileDialog` mai contine metodele:

`String getDirectory()` `void setDirectory(String dir)`	Afla/specifica directorul din care se va face selectia fisierului sau în care se va face salvare. Sunt permise si notatii specifice pentru directorul curent (.), directorul radacina (/), etc.
`String getFile()` `void setFile(String f)`	Returneaza numele fisierului selectat. Stabileste numele implicit al fisierului care va aparea în caseta de dialog
`FilenameFilter getFilenameFilter()` `void setFilenameFilter(FilenameFilter f)`	Afla/specifica filtrul care se va aplica fisierelor din directorul din care se va face selectia fisierului sau în care se va face salvare (vezi "Intrari si iesiri - Interfata FilenameFilter") Nu functioneaza pe platformele Windows !
`int getMode()` `void setMode(int mod)`	Afla/specifica daca încarcam sau salvam un fisier; • `FileDialog.LOAD` (pentru încarcare) • `FileDialog.SAVE` (pentru salvare)

Sa consideram un exemplu în care vom alege, prin intermediul unui obiect `FileDialog`, un fisier cu extensia "java". Directorul initial este directorul curent, iar numele implicit este `TestFileDialog.java`. Numele fisierului ales va fi afisat la consola.

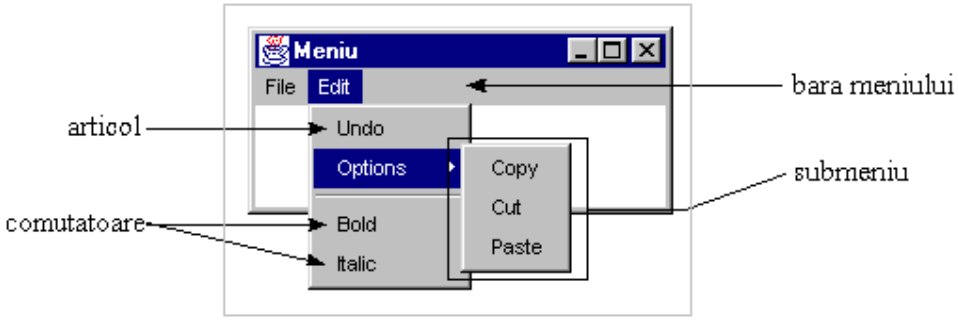
```
import java.awt.*;
import java.awt.event.*;
import java.io.*;

class FerPrinc extends Frame implements ActionListener{

	public FerPrinc(String titlu) {
		super(titlu);
		this.addWindowListener(new WindowAdapter() {
			public void windowClosing(WindowEvent e) {
				System.exit(0);
			}
		});
	}
	public void initializare() {
		setLayout(new FlowLayout());
		setSize(300, 100);

		Button b = new Button("Alege fisier");
		add(b);
		b.addActionListener(this);
	}

	//metoda interfetei ActionListener
	public void actionPerformed(ActionEvent e) {
		FileDialog fd = new FileDialog(this, "Alegeti un fisier",
		//stabilim directorul curent
		fd.setDirectory(".");

		//numele implicit
		fd.setFile("TestFileDialog.java");

		//specificam filtrul
		fd.setFilenameFilter(new FilenameFilter() {
			public boolean accept(File dir, String numeFis) {
				return (numeFis.endsWith(".java"));
			}
		});
		fd.show(); //facem vizibila fereastra de dialog

		System.out.println("Fisierul ales este:" + fd.getFile());
	}
}

public class TestFileDialog {
	public static void main(String args[]) {
		FerPrinc f = new FerPrinc("Fereastra principala");
		f.initializare();
		f.show();
	}
}
```

Curs 6
Folosirea meniurilor

Spre deosebire de celelate obiecte grafice, care deriva din clasa `Component`, componentele unui meniu reprezinta instante ale unor clase derivate din superclasa abstracta **MenuComponent**. Aceasta exceptie este facuta deoarece multe platforme grafice limiteaza capabilitatile unui meniu.
Meniurile sunt grupate în doua categorii:

- **Meniuri fixe** (vizibile permanent): sunt grupate într-o bara de meniuri ce contine câte un meniu pentru fiecare intrare a sa; la rândul lor aceste meniuri contin articole ce pot fi selectate, comutatoare - care au doua stari (checkbox) sau alte meniuri (submeniuri). O fereastra poate avea un singur meniu fix.
- **Meniuri de context** (popup): sunt meniuri invizbile asociate unei ferestre si care se activeaza prin apasarea butonul drept al mouse-ului. Diferenta fata de meniurile fixe consta în faptul ca meniurile de context nu au bara de meniuri.

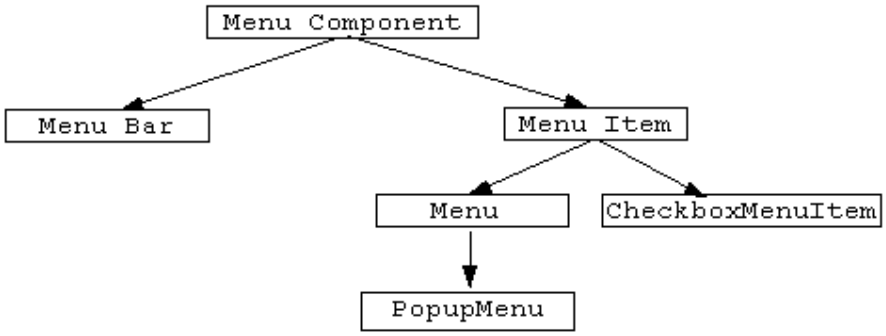
In figura de mai jos este pusa în evidenta alcatuirea unui meniu fix:



Exemplul de mai sus contine o bara de meniuri, doua meniuri principale *File* si *Edit*. Meniul *Edit* contine la rândul lui alt meniu (submeniu) *Options* , articolul *Undo* si doua comutatoare *Bold* si *Italic*.

In Java AWT meniurile sunt reprezentate ca instante al clasei **MenuBar**, aceasta fiind clasa care descrie barele de meniuri. Un obiect de tip **MenuBar** contine obiecte de tip **Menu**, care sunt de fapt meniurile derulante propriu-zise. La rândul lor acestea pot contine obiecte de tip **MenuItem, CheckBoxMenuItem**, dar si alte obiecte de tip `Menu` (submeniuri).

Sa vedem în continuare care este ierarhia claselor folosite în lucrul cu meniuri si sa analizam pe rând aceste clase:



Pentru a putea contine un meniu o componenta trebuie sa implementez interfata **MenuContainer**. Cel mai adesea meniurile sunt atasate ferestrelor, mai precis obiectelor de tip `Frame`, aceste implementând interfata `MenuContainer`.
Atasarea unei bare de meniuri la o fereastra se face prin metoda **addMenuBar** a clasei `Frame`.
Sa vedem cum ar arata un program care construieste un meniu ca cel din figura de mai sus:

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

public class TestMenu {
       public static void main(String args[]) {
       Frame f = new Frame("Meniu");

       MenuBar mb = new MenuBar();

       Menu fisier = new Menu("File");
       fisier.add(new MenuItem("Open"));
       fisier.add(new MenuItem("Close"));
       fisier.addSeparator();
       fisier.add(new MenuItem("Exit"));

       Menu optiuni = new Menu("Options");
       optiuni.add(new MenuItem("Copy"));
       optiuni.add(new MenuItem("Cut"));
       optiuni.add(new MenuItem("Paste"));

       Menu editare = new Menu("Edit");
       editare.add(new MenuItem("Undo"));
       editare.add(optiuni);

       editare.addSeparator();
       editare.add(new CheckboxMenuItem("Bold"));
       editare.add(new CheckboxMenuItem("Italic"));

       mb.add(fisier);
       mb.add(editare);

       f.setMenuBar(mb);
       f.show();

       }
}
```

Clasa MenuComponent

Este o clasa abstracta, din care sunt extinse toate celelalte clase folosite pentru lucrul cu meniuri, fiind analoaga celeilalte superclase abstracte `Component`. Clasa `MenuComponent` contine metode de ordin general, dintre care amintim `getName, setName, getFont, setFont`, cu sintaxa si semnificatiile uzuale.

Clasa Clasa MenuBar

Permite crearea barelor de meniuri asociate unei ferestre cadru (`Frame`). Aceasta clasa adapteaza conceptul de bara de meniuri la platforma curenta de lucru. Pentru a lega bara de meniuri la o anumita fereastra trebuie apelata metoda **setMenuBar** din clasa `Frame`.
Crearea unei bare de meniuri si legarea ei de o fereastra se realizeaza astfel:

```
//creez bara de meniuri
MenuBar mb = new MenuBar();

//adaug meniurile derulante la bara de meniuri
. . .

//atasez unei ferestre bara de meniuri
Frame f = new Frame("Fereastra cu meniu");
```

```
f.addMenuBar(mb);
```

Clasa MenuItem

Orice articol al unui meniu trebuie sa fie o instanta a clasei `MenuItem`. Instantele acestei clase descriu asadar articolele (optiunile individuale) ale meniurilor derulante, cum sunt "Open", "Close", "Exit", etc. O instanta a clasei `MenuItem` reprezinta de fapt o eticheta ce descrie numele cu care va aparea articolul în meniu, însotita eventual de un accelerator (obiect de tip `MenuShortcut`) ce reprezinta combinatia de taste cu care articolul poate fi apelat rapid (vezi "Acceleratori").

Clasa Menu

Este clasa care permite crearea unui meniu derulant într-o bara de meniuri. Optional, un meniu poate fi declarat ca fiind tear-off, ceea ce înseamna ca poate fi deschis si deplasat cu mouse-ul (dragged) într-o alta pozitie decât cea originala ("rupt" din pozitia sa). Acest mecanism este dependent de platforma si poate fi ignorat pe unele dintr ele.
Fiecare meniu are o eticheta, care este de fapt numele sau ce va fi afisat pe bara de meniuri.
Articolele dintr-un meniu trebuie sa apartina clasei `MenuItem`, ceea ce înseamna ca pot fi instante ale uneia din clasele `MenuItem, Menu sau CheckboxMenuItem`.

```
//Exemplu
MenuBar mb = new MenuBar();                 //creez bara de meniuri
Menu optiuni = new Menu("Options");               //creez un meniu
optiuni.add(new MenuItem("Copy"));
optiuni.add("Cut");                         //adaug articole
optiuni.add("Paste);
optiuni.addSeparator();
optiuni.add("Help");
mb.add(optiuni);                                  //adaug meniul la
bara
Frame f = new Frame("Fereastra cu meniu");
f.addMenuBar( mb );                         //atasez bara unei
ferestre
```

Clasa CheckboxMenuItem

Implementeaza într-un meniu articole de tip comutator - articole care au doua stari logice (validat/nevalidat), actionarea asupra articolului determinând trecerea sa dintr-o stare în alta. La validarea unui comutator în dreptul etichetei sale va aparea un simbol grafic care indica acest lucru; la invalidarea sa, simbolul grafic va dispare.
Clasa `CheckboxMenuItem` are aceeasi functionalitate cu cea a casetelor de validare, implementând interfata `ItemSelectable`.

Tratarea evenimentelor generate de meniuri

La alegerea unei optiuni dintr-un meniu se genereaza un eveniment de tip **ActionEvent** si comanda este reprezentata de numele optiunii alese. Asadar pentru a activa optiunile unui meniu trebuie implementat un obiect receptor care sa implementeze interfata **ActionListener** si care în metoda `actionPerformed` sa specifice codul ce trebuie executat la alegerea unei optiuni.
Fiecarui meniu în putem asocia un obiect receptor diferit, ceea ce usureaza munca în cazul în care ierarhia de meniuri este complexa. Pentru a realiza legatura între obiectul meniu si obiectul receptor trebuie sa adaugam receptorul în lista de ascultatori a meniului respectiv prin comanda: `meniu.addActionListener(listener)`.
Asadar, tratarea evenimentelor unui meniu este asemanatoare cu tratarea butoanelor, ceea ce face posibil ca unui buton de pe suprafata de afisare sa îi corespunda o optiune într-un meniu, ambele cu acelasi nume, tratarea evenimentului corespunzator apasarii butonului, sau alegerii optiunii facându-se o singura data într-o clasa care este înregistrata ca receptor atât la buton cât si la meniu.

Un caz special îl constituie optiunile de tip `CheckboxMenuItem`. Obiectele de acest tip se gasesc într-o categorie comuna cu `List, Choice, CheckBox`, implementeaza o interfata comuna `ItemSelectable` si toate genereaza evenimente de tip **ItemEvent**. Din aceasta cauza actionarea unei optiuni de tip `CheckboxMenuItem` nu va determina generarea unui eveniment de tip `ActionEvent` de catre meniul din care face parte, ci va genera un eveniment `ItemEvent` chiar de catre articolul respectiv. Pentru a intercepta un asemenea eveniment avem nevoie de un obiect receptor care sa implementeze interfata **ItemListener** si sa specifice în metoda acesteia `itemStateChanged` codul ce trebuie executat la validarea/invalidarea optiunii din meniu. De asemenea receptorul trebuie înregistrat cu metoda `addItemListener`.
Tipul de operatie selectare / deselectare este codificat de câmpurile statice **`ItemEvent.SELECTED`** si **`ItemEvent.DESELECTED`**.

Exemplu de tratare a evenimentelor unui meniu

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

public class TestMenuEvent extends Frame
       implements ActionListener, ItemListener{
       public TestMenuEvent(String titlu) {
              super(titlu);
       }
       public void actionPerformed(ActionEvent e) {
              String command = e.getActionCommand();
              if (command.equals("Exit"))
                     System.exit(0);
                     //valabila si pentru meniu si pentru buton
       }

       public void itemStateChanged(ItemEvent e) {
              if (e.getStateChange() == ItemEvent.SELECTED)
                     setTitle("Checked!");
              else
                     setTitle("Not checked!");
       }

       public static void main(String args[]) {
              MenuBar mb = new MenuBar();
              Menu test = new Menu("Test");
              CheckboxMenuItem check = new CheckboxMenuItem("Check me");
              test.add(check);
              test.addSeparator();
              test.add(new MenuItem("Exit"));

              mb.add(test);
              TestMenuEvent f = new TestMenuEvent("Test Meniu");
              Button btnExit = new Button("Exit");

              f.setMenuBar(mb);
              f.add(btnExit, BorderLayout.SOUTH);
              f.setSize(300, 200);
              f.show();

              test.addActionListener(f);
              check.addItemListener(f);
              btnExit.addActionListener(f);
       }
}
```

Meniuri de context (popup)

Au fost introduse începând cu AWT 1.1 si sunt implementate prin intermediul clasei **PopupMenu**, subclasa directa a clasei `Menu`. Sunt meniuri invizibile care sunt activate uzual prin apasarea butonului drept al mouse-ului, fiind afisate la pozitia la care se gasea mouse-ul în momentul apasarii butonului sau drept.
Metodele de adaugare a articolelor unui meniu popup sunt identice cu cele de la meniurile fixe, `PopupMenu` fiind subclasa directa a clasei `Menu`.

```
popup = new PopupMenu("Options");
popup.add(new MenuItem("New"));
popup.add(new MenuItem("Edit"));
popup.addSeparator();
popup.add(new MenuItem("Exit"));
```

Afisarea meniului de context se face prin metoda **show**:

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

si este, de obicei, rezultatul apasarii unui buton al mouse-ului, pentru a avea acces rapid la meniu. Argumentul "origin" reprezinta componenta fata de originile careia se va calcula pozitia de afisare a meniului popup. De obicei, reprezinta instanta ferestrei în care se va afisa meniul.
Meniurile de context nu se adauga la un alt meniu (bara sau sub-meniu) ci se ataseaza la o componenta (de obicei la o fereastra) prin metoda **add**: `fereastra.add(pm)` . In cazul când avem mai multe meniuri popup pe care vrem sa le folosim într-o fereastra, trebuie sa le definim pe toate si, la un moment dat, vom adauga ferestrei meniul corespunzator. Dupa închiderea acestuia vom "rupe" legatura între fereastra si meniu prin instructiunea **remove**:

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

In exemplul de mai jos, vom crea un meniu de contexe ca în imaginea de mai jos, pe care îl vom activa la apasarea butonului drept al mouse-ului pe suprafata ferestrei principale:

La alegerea optiunii "Exit" din meniu vom termina programul. Interceptarea evenimentelor generate de un meniu popup se realizeaza identic ca pentru meniurile fixe (vezi "Tratarea evenimentelor generate de meniuri").

Exemplu de folosire a unui meniu popup

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

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

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

```
                });

                this.addMouseListener(new MouseAdapter() {
                       public void mouseClicked(MouseEvent e) {
                       if ( (e.getModifiers() & InputEvent.BUTTON3_MASK)
                      == InputEvent.BUTTON3_MASK )
                              popup.show(origin, e.getX(), e.getY());
                       //BUTTON3 reprezinta butonul din dreapta mouse-ului
                       }
                });
                setSize(300, 300);

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

        public void actionPerformed(ActionEvent e) {
                String command = e.getActionCommand();
                //La alegerea optiunii "Exit" din meniu parasim aplicatia
                if (command.equals("Exit"))
                        System.exit(0);
        }
}

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

Acceleratori (Clasa MenuShortcut)

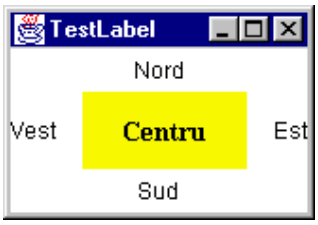
Incepând cu Java AWT 1.1 este posibila specificarea unor combinatii de taste (acceleratori - shortcuts) pentru accesarea directa, prin intermediul tastaturii, a optiunilor dintr-un meniu. Astfel, oricarui obiect de tip `MenuItem` îi poate fi asociat un obiect de tip accelerator, definit prin intermediul clasei **MenuShortcut**. Singurele combinatii de taste care pot juca rolul acceleratorilor sunt: + sau + + .
Atribuirea unui accelerator la un articol al unui meniu poate fi realizata prin constructorul obiectelor de tip `MenuItem` în forma:

```
                MenuItem(String eticheta, MenuShortcut accelerator)
                //Exemplu
                new MenuItem("Open", new MenuShortcut(KeyEvent.VK_O));
```

Folosirea componentelor

Clasa Label

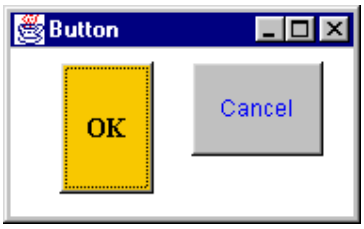
Un obiect de tip **Label (eticheta)** reprezinta o componenta pentru plasarea unui text pe o suprafata de afisare. O eticheta este formata dintr-o singura linie de text static ce nu poate fi modificat de catre utilizator, dar poate fi modificat din program.
Exemplu: definim cinci etichete si le plasam într-un container.




```
import java.awt.*;
public class TestLabel {
	public static void main(String args[]) {
		Frame f = new Frame("TestLabel");
		f.setLayout(new BorderLayout());

		Label nord, sud, est, vest, centru;
		nord = new Label("Nord", Label.CENTER);
		sud = new Label("Sud", Label.CENTER);
		est = new Label("Est", Label.RIGHT);
		vest = new Label("Vest", Label.LEFT);
		centru = new Label("Centru", Label.CENTER);
		centru.setBackground(Color.yellow);
		centru.setFont(new Font("Arial", Font.BOLD, 14));

		f.add(nord, BorderLayout.NORTH);
		f.add(sud, BorderLayout.SOUTH);
		f.add(est, BorderLayout.EAST);
		f.add(vest, BorderLayout.WEST);
		f.add(centru, BorderLayout.CENTER);
		f.pack();

		f.show();
	}
}
```

Clasa Button

Un obiect de tip **Button (buton)** reprezinta o componenta pentru plasarea unui buton etichetat pe o suprafata de afisare.
Exemplu: definim doua butoane si le plasam pe o fereastra; la apasarea butonului "OK" titlul ferestrei va fi "Confirmare", iar la apasarea butonului "Cancel" titlul ferestrei va fi "Renuntare".

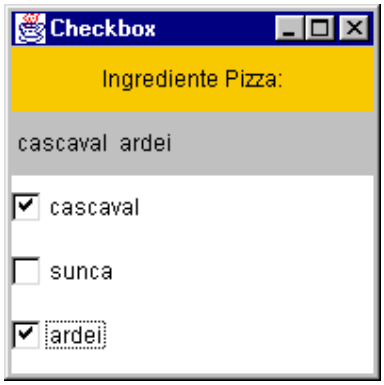



```
import java.awt.*;
import java.awt.event.*;
```

```
class Fereastra extends Frame implements ActionListener{
	public Fereastra(String titlu) {
		super(titlu);
		this.addWindowListener(new WindowAdapter() {
			public void windowClosing(WindowEvent e) {
				System.exit(0);
			}
		});
	}
	public void initializare() {
		setLayout(null);
		setSize(200, 200);

		Button b1 = new Button("OK");
		b1.setBounds(30, 30, 50, 70);
		b1.setFont(new Font("Arial", Font.BOLD, 14));
		b1.setBackground(java.awt.Color.orange);
		add(b1);

		Button b2 = new Button("Cancel");
		b2.setBounds(100, 30, 70, 50);
		b2.setForeground(java.awt.Color.blue);
		add(b2);

		b1.addActionListener(this);
		b2.addActionListener(this);
	}

	//metoda interfetei ActionListener
	public void actionPerformed(ActionEvent e) {
		String command = e.getActionCommand();
		System.out.println(e.toString());
		if (command.equals("OK"))
			setTitle("Confirmare!");
		if (command.equals("Cancel"))
			setTitle("Anulare!");
	}
}

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

Clasa Checkbox

Un obiect de tip **Checkbox (comutator)** reprezinta o componenta care se poate gasi în doua stari : "selectata" sau "neselectata" (on/off). Actiunea utilizatorului asupra unui comutator îl trece pe acesta în starea complementara celei în care se gasea. Este folosit pentru a prelua o anumita optiune de la utilizator.

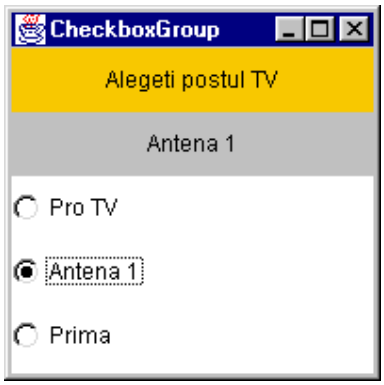



```
import java.awt.*;
import java.awt.event.*;

class Fereastra extends Frame implements ItemListener {
	private Label label1, label2;
	private Checkbox cbx1, cbx2, cbx3;

	public Fereastra(String titlu) {
		super(titlu);
		this.addWindowListener(new WindowAdapter() {
			public void windowClosing(WindowEvent e) {
				System.exit(0);
			}
		});
	}
	public void initializare() {
		setLayout(new GridLayout(5, 1));
		label1 = new Label("Ingrediente Pizza:", Label.CENTER);
		label1.setBackground(Color.orange);
		label2 = new Label("");
		label2.setBackground(Color.lightGray);

		cbx1 = new Checkbox("cascaval");
		cbx2 = new Checkbox("sunca");
		cbx3 = new Checkbox("ardei");

		add(label1);
		add(label2);
		add(cbx1);
		add(cbx2);
		add(cbx3);
		pack();
		setSize(200, 200);

		cbx1.addItemListener(this);
		cbx2.addItemListener(this);
		cbx3.addItemListener(this);
	}

	//metoda interfetei ItemListener
	public void itemStateChanged(ItemEvent e) {
		StringBuffer ingrediente = new StringBuffer();
		if (cbx1.getState() == true)
			ingrediente.append(" cascaval ");
```

```
                if (cbx2.getState() == true)
                        ingrediente.append(" sunca ");
                if (cbx3.getState() == true)
                        ingrediente.append(" ardei ");
                label2.setText(ingrediente.toString());
        }
}

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

Clasa CheckboxGroup

Un obiect de tip **CheckboxGroup** defineste un **grup de comutatoare** din care doar unul poate fi selectat. Uzual, aceste componente se mai numesc **butoane radio**.

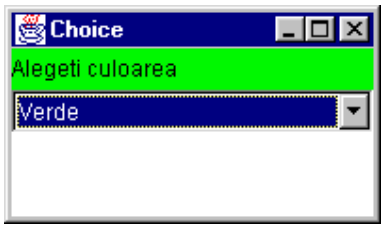



```
import java.awt.*;
import java.awt.event.*;

class Fereastra extends Frame implements ItemListener {
        private Label label1, label2;
        private Checkbox cbx1, cbx2, cbx3;
        private CheckboxGroup cbg;

        public Fereastra(String titlu) {
                super(titlu);
                this.addWindowListener(new WindowAdapter() {
                        public void windowClosing(WindowEvent e) {
                                System.exit(0);
                        }
                });
        }
        public void initializare() {
                setLayout(new GridLayout(5, 1));
                label1 = new Label("Alegeti postul TV", Label.CENTER);
                label1.setBackground(Color.orange);
                label2 = new Label("", Label.CENTER);
```

```
                label2.setBackground(Color.lightGray);

                cbg = new CheckboxGroup();
                cbx1 = new Checkbox("Pro TV", cbg, false);
                cbx2 = new Checkbox("Antena 1", cbg, false);
                cbx3 = new Checkbox("Prima", cbg, false);

                add(label1);
                add(label2);
                add(cbx1);
                add(cbx2);
                add(cbx3);
                pack();
                setSize(200, 200);

                cbx1.addItemListener(this);
                cbx2.addItemListener(this);
                cbx3.addItemListener(this);
        }

        //metoda interfetei ItemListener
        public void itemStateChanged(ItemEvent e) {
                Checkbox cbx = cbg.getSelectedCheckbox();
                if (cbx != null)
                        label2.setText(cbx.getLabel());
        }
}

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

Clasa Choice

Un obiect de tip **Choice** defineste o **lista de optiuni** din care utilizatorul poate selecta una singura. La un moment dat, din întreaga lista doar o singura optiune este vizibila, cea selectata în momentul curent. O componenta `Choice` este însotita de un buton etichetat cu o sageata verticala la apasarea caruia este afisata întreaga sa lista, pentru ca utilizatorul sa poata selecta o anumita optiune.

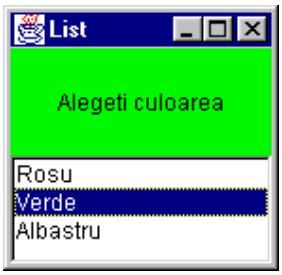



```
import java.awt.*;
import java.awt.event.*;

class Fereastra extends Frame implements ItemListener {
        private Label label;
```

```
	private Choice culori;

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

	public void initializare() {
		setLayout(new GridLayout(4, 1));

		label = new Label("Alegeti culoarea");
		label.setBackground(Color.red);

		culori = new Choice();
		culori.add("Rosu");
		culori.add("Verde");
		culori.add("Albastru");
		culori.select("Rosu");

		add(label);
		add(culori);
		pack();
		setSize(200, 100);

		culori.addItemListener(this);
	}

	//metoda interfetei ItemListener
	public void itemStateChanged(ItemEvent e) {
		switch (culori.getSelectedIndex()) {
			case 0:
				label.setBackground(Color.red);
				break;
			case 1:
				label.setBackground(Color.green);
				break;
			case 2:
				label.setBackground(Color.blue);
		}
	}
}

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

Clasa List

Un obiect de tip **List** defineste o **lista de optiuni** care poate fi setata astfel încât utilizatorul sa poata selecta o singura optiune sau mai multe. Toate optiunile listei sunt vizibile în limita dimensiunilor grafice ale componentei.

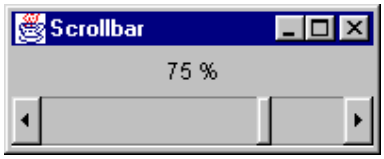



```
import java.awt.*;
import java.awt.event.*;

class Fereastra extends Frame implements ItemListener {
	private Label label;
	private List culori;

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

	public void initializare() {
		setLayout(new GridLayout(2, 1));

		label = new Label("Alegeti culoarea", Label.CENTER);
		label.setBackground(Color.red);

		culori = new List(3);
		culori.add("Rosu");
		culori.add("Verde");
		culori.add("Albastru");
		culori.select(3);

		add(label);
		add(culori);
		pack();
		setSize(200, 200);

		culori.addItemListener(this);
	}

	//metoda interfetei ItemListener
	public void itemStateChanged(ItemEvent e) {
		switch (culori.getSelectedIndex()) {
			case 0:
				label.setBackground(Color.red);
				break;
			case 1:
				label.setBackground(Color.green);
```

```
                        break;
                case 2:
                        label.setBackground(Color.blue);
                }
        }
}

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

Clasa Scrollbar

Un obiect de tip **Scrollbar** defineste o **bara de defilare** verticala sau orizontala. Este utila pentru punerea la dispozitie a utilizatorului a unei modalitati sugestive de a alege o anumita valoare dintr-un interval.

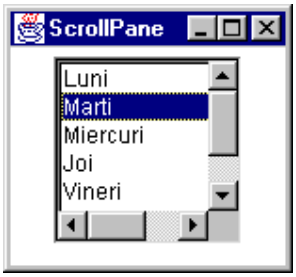



```
import java.awt.*;
import java.awt.event.*;

class Fereastra extends Frame implements AdjustmentListener {
        private Scrollbar scroll;
        private Label valoare;

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

        public void initializare() {
                setLayout(new GridLayout(2, 1));

                valoare = new Label("", Label.CENTER);
                valoare.setBackground(Color.lightGray);

                scroll = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0, 101);
                add(valoare);
                add(scroll);
                pack();
                setSize(200, 80);

                scroll.addAdjustmentListener(this);
        }
```

```
        //metoda interfetei ItemListener
        public void adjustmentValueChanged(AdjustmentEvent e) {
                valoare.setText(scroll.getValue() + " %");
        }
}

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

Clasa ScrollPane

Un obiect de tip **ScrollPane** permite atasarea unor bare de defilare (orizontala si/sau verticala) oricarei componente grafice. Acest lucru este util pentru acele componente care nu au implementata functionalitatea de defilare automata, cum ar fi listele (obiecte din clasa `List`).




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

class Fereastra extends Frame {
        private ScrollPane sp;
        private List list;

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

        public void initializare() {
                setLayout(new FlowLayout());

                list = new List(7);
                list.add("Luni");
                list.add("Marti");
                list.add("Miercuri");
                list.add("Joi");
                list.add("Vineri");
```

```
                list.add("Sambata");
                list.add("Duminica");
                list.select(1);

                sp = new ScrollPane(ScrollPane.SCROLLBARS_ALWAYS);
                sp.add(list);

                add(sp);
                pack();
                setSize(200, 200);

        }

}

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

Clasa TextField

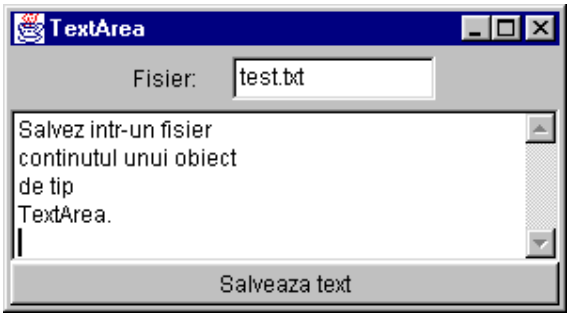
Un obiect de tip **TextField** defineste un control de editare a textului pe o singura linie. Este util pentru interogarea utilizatorului asupra unor valori.


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

class Fereastra extends Frame implements TextListener {
        private TextField nume, parola;
        private Label acces;
        private static final String UID="Ion", PWD="java" ;

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

        public void initializare() {
                setLayout(new GridLayout(3, 1));
                setBackground(Color.lightGray);
```

```
                nume = new TextField("", 30);
                parola = new TextField("", 10);
                parola.setEchoChar('*');

                Panel p1 = new Panel();
                p1.setLayout(new FlowLayout(FlowLayout.LEFT));
                p1.add(new Label("Nume:"));
                p1.add(nume);

                Panel p2 = new Panel();
                p2.setLayout(new FlowLayout(FlowLayout.LEFT));
                p2.add(new Label("Parola:"));
                p2.add(parola);

                acces = new Label("Introduceti numele si parola!",
                add(p1);
                add(p2);
                add(acces);

                pack();
                setSize(350, 100);

                nume.addTextListener(this);
                parola.addTextListener(this);
        }

        //metoda interfetei TextListener
        public void textValueChanged(TextEvent e) {
                if ((nume.getText().length() == 0) ||
                        acces.setText("");
                        return;
                }
                if (nume.getText().equals(UID) &&
                        acces.setText("Acces permis!");
                else
                        acces.setText("Acces interzis!");

        }
}

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

Clasa TextArea

Un obiect de tip **TextArea** defineste un control de editare a textului pe mai multe linii. Este util pentru editarea de texte, introducerea unor comentarii, etc .




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

class Fereastra extends Frame implements TextListener, ActionListener {
	private TextArea text;
	private TextField nume;
	private Button save;

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

	public void initializare() {
		setBackground(Color.lightGray);

		text = new TextArea("", 30, 10,
		nume = new TextField("", 12);
		save = new Button("Salveaza text");
		save.setActionCommand("save");
		save.setEnabled(false);

		Panel fisier = new Panel();
		fisier.add(new Label("Fisier:"));
		fisier.add(nume);

		add(fisier, BorderLayout.NORTH);
		add(text, BorderLayout.CENTER);
		add(save, BorderLayout.SOUTH);

		pack();
		setSize(300, 200);

		text.addTextListener(this);
		save.addActionListener(this);
	}

	//metoda interfetei TextListener
	public void textValueChanged(TextEvent e) {
		if ((text.getText().length() == 0) ||
			save.setEnabled(false);
```

```
                else
                        save.setEnabled(true);
        }

        //metoda interfetei ActionListener
        public void actionPerformed(ActionEvent e) {
                String continut = text.getText();
                int len = continut.length();
                char buffer[] = new char[len];

                try {
                        FileWriter out = new FileWriter(nume.getText());
                        continut.getChars(0, len-1, buffer, 0);
                        out.write(buffer);
                        out.close();
                        text.requestFocus();
                }
                catch(IOException ex) {
                        ex.printStackTrace();
                }
        }
}

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