AWT

Introducere, concepte

Librăria AWT (Abstract Window Toolkit) oferă o interfață grafică pentru programele Java. Această librărie oferă unelte folosite pentru comunicarea program – utilizator.

Problema acestei librării este că Java 1.0.2 a fost înlocuit de versiuni ca 1.1 şi 1.2, versiuni ce au introdus alte caracteristici. De foarte mulți ani, programatorii au trecut prin mari bătăi de cap în încercările de a porta un program scris pentru sisteme Unix, Linux pe sistem Windows sau Macintosh. În 1995, Sun anunța o posibilă soluție: Java. Pentru sisteme de operare bazate pe ferestre, portabilitatea era o problemă. Când transferăm un program scris pentru Windows pe sisteme Macintosh, codul care tratează lucrul cu interfața grafică trebuie rescris complet. În Java însă, acest neajuns cauzat de portabilitate, este rezolvat parțial prin AWT. Prin această librărie, programele se pot rula în aceeași manieră în orice sistem de operare, ele vor arata la fel. De exemplu dacă aplicația folosește liste de tip *Combobox*, acestea vor arăta ca o listă de tip Windows, dacă aplicația rulează sub Windows, sau ca o lista de Unix dacă aplicația rulează sub Unix.

În construcția acestei librarii există câteva concepte de bază pe care le vom detalia în acest prim capitol şi anume: *componente*, *membrii*, *aspect*, *recipient*. În continuare vom vorbi mai pe larg despre fiecare.

Componente

Interfețele cu utilizatorul sunt construite în jurul ideii de *componente*: dispozitive care implementează o parte din interfețe. Începând cu anii '80 aceste dispozitive au fost extinse, cele mai cunoscute fiind: butoane, meniuri, ferestre, checkbox-uri, bare de scroll, etc. AWT conține un set de astfel de componente, şi în plus, o maşinărie de creat componente personalizate. În continuare vom prezenta componentele principale ale AWT fără a intra în detalii.

Text static

Clasa *Label* oferă un mod de a afişa o linie de text pe ecran. Se poate controla fontul folosit pentru text şi culoarea. Mai jos este un exemplu de *etichete* desenate pe ecran:

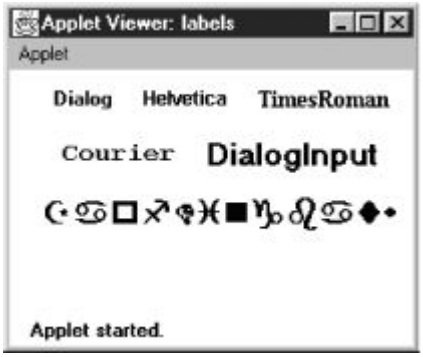


Figura 10.1 instanțe de *Label*

Componente pentru introducere date

Java oferă câteva moduri de a permite utilizatorului de a introduce date într-o aplicație. Utilizatorul poate tipări informația, sau o poate selecta dintr-o listă de opțiuni prestabilită.

Clasele TextField şi TextArea

Există două componente pentru a putea introduce date de la tastatură: *TextField* pentru o singură linie şi *TextArea* pentru linii multiple. Această oferă şi mijloace pentru diferite validări a textului introdus de utilizator. În figura de mai jos sunt câteva exemple pentru aceste componente:

Figura 10.2 Elementele de text

Clasele CheckBox şi CheckboxGroup

Componentele menționate oferă mecanisme pentru a permite utilizatorului să selecteze dintr-o listă de opțiuni, una singură sau mai multe. Primul mecanism se numeşte Checkbox, şi permite bifarea unei opțiuni. În partea stângă a ferestrei se află un astfel de element numit *Dialog*. Un clic pe această căsuța marchează opțiunea ca fiind *true* si se bifează. Un clic ulterior, deselectează opțiunea marcând-o ca *false*.

Clasa CheckboxGroup nu este o componentă, ci oferă un mod de a grupa căsuțele de bifare într-o mulțime în care opțiunile se exclud mutual. Denumirea lor este de butoane radio.

Figura 10.3 Căsuțe de opțiuni

Clasa de listare de opţiuni

Problema cu clasele Checkbox şi CheckboxGroup este că atunci când listele sunt prea mari, si opțiunile sunt multe, nu există loc pentru a plasa aceste butoane radio. În acest caz se folosesc componentele *Choice*, care reprezintă o listă de mai multe elemente, din care se poate selecta doar unul singur. În figura de mai jos este reprezentat un astfel de element.

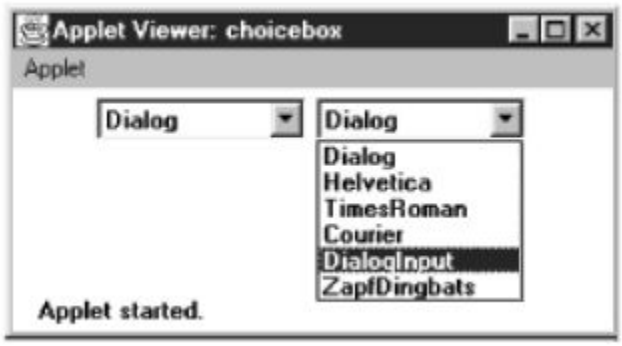


Figura 10.4 Clasa Choice

Clasa List

Totuşi dacă dorim sa afişăm integral elementele unei liste, şi să oferim posibilitatea de a selecta multiple elemente, avem la dispoziție clasa *List*. În figura de mai jos avem o astfel de componentă.

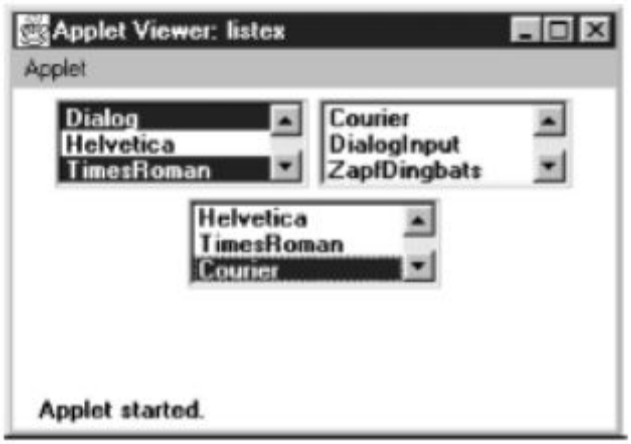


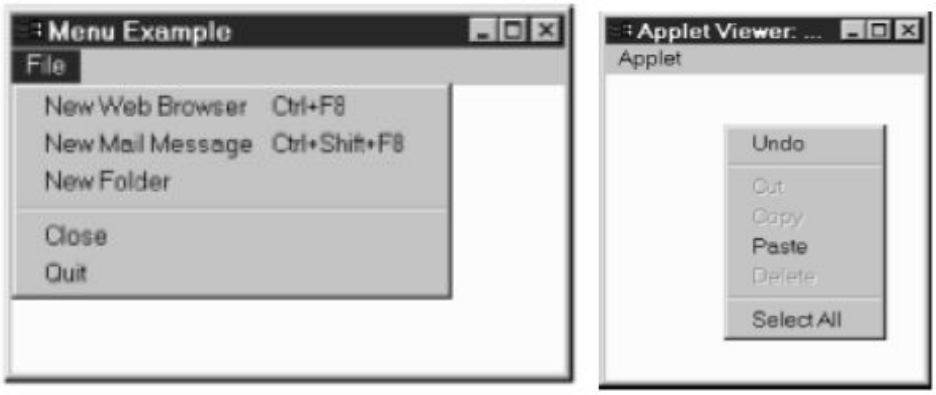
Figura 10.5 Clasa List

Meniuri

Interfețele moderne folosesc aceste elemente, însă în Java în această librărie, meniurile pot apare doar într-un *Frame*. O clasă *Menu*, este complexă şi comportă multe părți: bare de meniu, elemente de meniu, etc.

Clasa PopupMenu

Meniurile Pop-up sunt folosite în funcție de context, mai ales când mouse-ul se află într-o regiune a ferestrei. Mai jos sunt prezentate ambele tipuri de meniuri:



a) b)

Figura 10.6 a) Meniuri b)Meniuri Pop-up

Declanşatoare de evenimente

Java oferă două componente a căror scop este de a declanşa acțiuni pe ecran: *Button* şi *Scrollbar*. Ele permit utilizatorului să specifice momentul realizării unei acțiuni.

Clasa Scrollbar

Într-un program gen *Word* sau într-un browser Web, atunci când o imagine sau text este prea mare pentru a fi afişată în pagină, acest element permite deplasarea utilizatorului în diverse părți ale ecranului. Atunci când se efectuează click pe *Scrollbar* efectul este exact deplasare într-o anumită direcție pentru a viziona conținutul paginii. În figura 10.7 este reprezentat acest element.

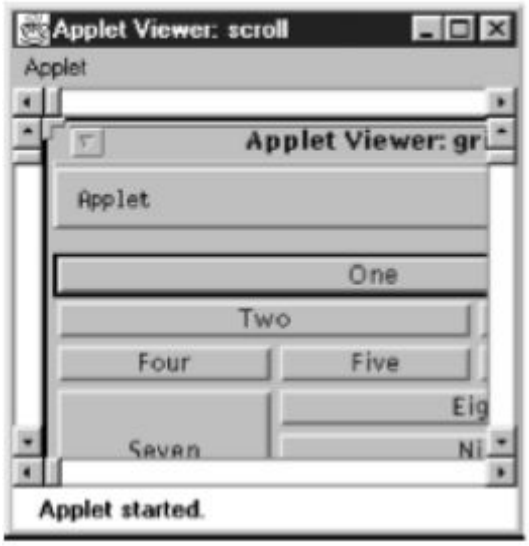


Figura 10.7 Scrollbar vertical şi orizontal

Unele componente ca TextArea sau List conțin deja aceste Scrollbar-uri pentru a diminua efortul programatorului. Pentru alte elemente ca Panel sau Frame va trebui implementat de la zero funcționalitatea pentru ScrollBar-uri.

Clasa Buton

Butonul este cel mai cunoscut element pentru declanşarea unor evenimente şi cel mai intuitiv. Java permite si butoane cu imagini: *ImageButton*.

Clasa Canvas

Această clasă reprezintă o suprafață goală: nu are nici o vizualizare predefinită. Se pot folosi Canvas pentru desenarea imaginilor, construirea componentelor personalizate, sau a super-componentelor ce conțin alte componente predefinite.

În continuare vom descrie în detaliu elementele amintite mai sus pentru a oferi o perspectivă asupra librăriei AWT.

Evenimente

Am descris în cursul anterior câteva evenimente, precum şi conceptul din spatele acestui mecanism. În cele ce urmează vom aprofunda aceste concepte.

Modelul vechi de evenimente Java

Modelul este destul de simplu: atunci când se recepționează un eveniment inițiat de utilizator, sistemul generează o instanță de tip *Event* şi o transmite programului. Programul identifică ținta (de exemplu componenta în care a avut loc evenimentul) şi permite acelei componente să trateze evenimentul. Dacă ținta nu poate trata evenimentul, se încearcă găsirea unei componente care va putea.

Identificarea țintei

Evenimentele au loc într-o clasă *Component*. Programul decide care componentă va primi evenimentul, pornind de la nivelul cel mai înalt. În exemplul ales, acțiunea de a efectua click pe frame-ul de mai jos, mai exact pe butonul *Blood*, va declanșa metoda *deliverEvent()* din cadrul acestei clase. Procesul va continua până când ținta găsită este butonul Blood, buton ce nu mai conține alte componente, așa că se va executa evenimentul aferent lui.

Figura 10.8 identificarea țintei

Mai jos este prezentată ierarhia de apeluri a metodei deliverEvent, până când evenimentul este livrat țintei.

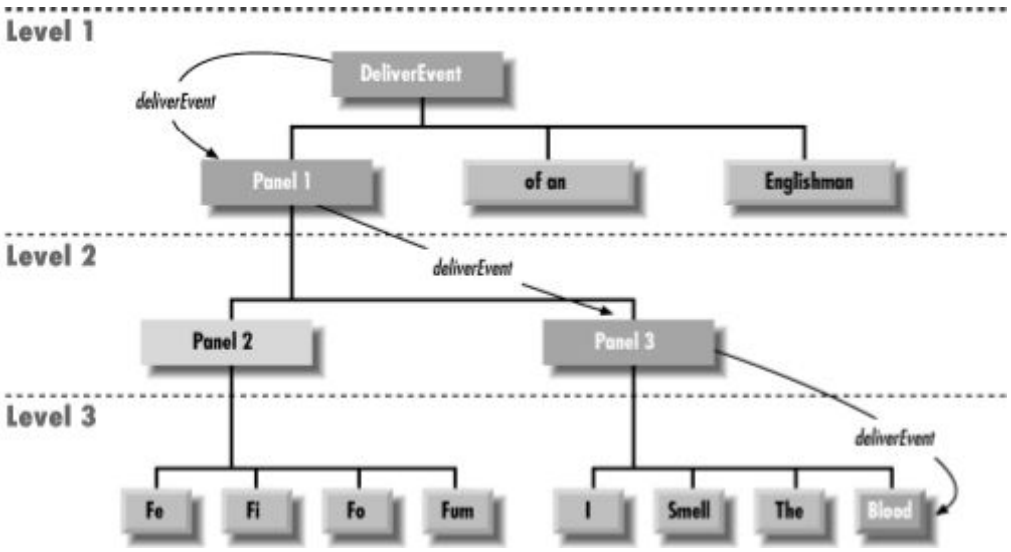


Figura 10.9 Livrarea evenimentului

Tratarea evenimentelor

Odată ce metoda *deliverEvent()* identifică ținta, se va apela metoda *handleEvent()* a țintei. Dacă ținta nu are suprascrisă această metodă, se apelează implementarea implicită moștenită de la obiectul *Component* moștenit, și nu se întâmplă nimic. În continuare vom descrie ierarhia claselor ce se ocupă de evenimente.

Clasa Event

Atunci când apare sau este declanșat un eveniment, este de responsabilitatea programatorului să capteze acel eveniment. Se poate decide netratarea lui, așa că evenimentul va trece ca și când nu ar fi avut loc. Atunci când se dorește transmiterea unui eveniment mai departe, sau tratarea lui, trebuie să înțelegem cum se comportă un obiect de acest tip. Înainte de a începe prezentarea acestei clase, trebuie spus că ea a fost înlocuită de *AWTEvent* pe care o vom descrie imediat după. Motivul pentru care o descriem aici este mai mult informativ și pentru a putea face legătura cronologic și conceptual între aceste două moduri de a soluționa problema legată de evenimente.

Variabilele

Clasa *Event* conține multe variabile ce oferă informație despre evenimentul ce are loc. Vom discuta despre câteva.

```
public int clickCount
```

Acest membru al clasei *Event*, permite verificarea evenimentului de double-click. Câmpul este relevant doar pentru evenimente de tip MOUSE_DOWN.

```
public Event evt
```

Acest câmp este folosit pentru a transmite evenimente, după cum am văzut mai sus. Programul poate trata evenimentul transmis, sau evenimentul poate fi trimis unui handler personalizat:

```
public boolean mouseDown (Event e, int x, int y)
{
      System.out.println ("Coordinates: " + x + "-" + y);
      if (e.evt != null)
            postEvent (e.evt);
      return true;
}
```

```
public int id
```

Acest câmp conține identificatorul evenimentului. Evenimentele generate de sistem au constantele predefinite, și câteva dintre ele sunt:

```
WINDOW_DESTROY    MOUSE_ENTER
WINDOW_EXPOSE     MOUSE_EXIT
WINDOW_ICONIFY    MOUSE_DRAG
WINDOW_DEICONIFY  SCROLL_LINE_UP
KEY_PRESS         SCROLL_LINE_DOWN
KEY_RELEASE       SCROLL_PAGE_UP
KEY_ACTION        SCROLL_PAGE_DOWN
KEY_ACTION_RELEASE      SCROLL_ABSOLUTE
MOUSE_DOWN        LIST_SELECT
MOUSE_UP          LIST_DESELECT
MOUSE_MOVE        ACTION_EVENT
```

```
public Object target
```

Câmpul `target` conține referința către un obiect care produce acest eveniment. De exemplu, dacă utilizatorul selectează un buton, butonul este ținta evenimentului. Dacă utilizatorul mișcă *mouse*-ul pe un obiect *Frame*, atunci *Frame* este ținta. `Target` va indica locul unde a avut loc evenimentul, și nu neapărat componenta care tratează evenimentul.

```
public long when
```

Această variabilă conține durata unui eveniment exprimată în milisecunde. Codul transformă o valoare *long* a acestei variabile în tipul *Date* pentru a examina ulterior durata:

```
Date d = new Date (e.when);
```

Constante

Clasa *Event* conține nenumărate constante, unele pentru a desemna ce eveniment a avut loc, altele pentru a ajuta la determinarea tastei apăsate.

Constantele ce se referă la taste se împart în două, după tipul de eveniment ce este generat:

KEY_ACTION - și se numesc taste pentru acțiune

KEY_PRESS – se desemnează alfanumericele.

În tabelul de mai jos sunt prezentate câteva constante și tipul de eveniment care folosește aceste constante:

Constanta – Tip eveniment	**Constanta – Tip eveniment**
HOME KEY_ACTION	F9 KEY_ACTION
END KEY_ACTION	F10 KEY_ACTION
PGUP KEY_ACTION	F11 KEY_ACTION
PGDN KEY_ACTION	F12 KEY_ACTION
UP KEY_ACTION	PRINT_SCREEN KEY_ACTION
DOWN KEY_ACTION	SCROLL_LOCK KEY_ACTION
LEFT KEY_ACTION	CAPS_LOCK KEY_ACTION
RIGHT KEY_ACTION	NUM_LOCK KEY_ACTION
F1 KEY_ACTION	PAUSE KEY_ACTION
F2 KEY_ACTION	INSERT KEY_ACTION
F3 KEY_ACTION	ENTER (\n) KEY_PRESS
F4 KEY_ACTION	BACK_SPACE (\b) KEY_PRESS
F5 KEY_ACTION	TAB (\t) KEY_PRESS
F6 KEY_ACTION	ESCAPE KEY_PRESS
F7 KEY_ACTION	DELETE KEY_PRESS

Există și modificatori pentru tastele Shift, Alt, Control. Atunci când utilizatorul apasă o tastă sau generează alt tip de eveniment, se poate verifica dacă a fost apăsată simultan și una din tastele mai sus menționate. Pentru aceasta avem la dispoziție modificatorii:

```
public static final int ALT_MASK
public static final int CTRL_MASK
public static final int SHIFT_MASK
```

Atunci când raportează un eveniment, sistemul setează câmpul *modifiers* ce poate fi verificat ulterior.

Evenimente de fereastră

Aceste evenimente au loc pentru componentele ce aparțin unui *Window*. Câteva din aceste evenimente sunt valabile doar pentru anumite platforme OS.

```
public static final int WINDOW_DESTROY
```

Acest eveniment este produs când sistemul spune unei ferestre să se autodistrugă. Aceasta intervine de obicei când utilizatorul selectează *Close* sau *Quit*. Implicit, instanțele *Frame* nu tratează acest eveniment.

```
public static final int WINDOW_EXPOSE
```

Acest eveniment este transmis atunci când o parte a ferestrei devine vizibilă. Pentru a afla ce parte a ferestrei a fost descoperită, se poate folosi getClipRect(), metodă ce aparține instanța clasei *Graphics*.

Evenimente de scroll

Aceste evenimente sunt declanșate de acțiunea utilizatorului pe o componentă de tip *Scrollbar*. Obiectele ce au un *Scrollbar* construit implicit (*List*, *TextArea*) nu generează aceste evenimente. Evenimentele pot fi tratate în metoda *handleEvent()* din cadrul Container-ului sau a unei subclase *Scrollbar*.

```
public static final int SCROLL_LINE_UP
```

Această constantă este transmisă unui eveniment care are loc atunci când utilizatorul apasă săgeata de sus a unui scrollbar vertical sau pe săgeata din stânga a unui scrollbar orizontal.

```
public static final int SCROLL_LINE_DOWN
```

Această constantă este setată unui eveniment care are loc atunci când utilizatorul apasă săgeata de jos a unui scrollbar vertical sau pe săgeata din dreapta a unui scrollbar orizontal.

```
public static final int SCROLL_PAGE_UP
```

Această constantă este setată unui eveniment care are loc atunci când utilizatorul apasă lângă săgeata de sus a unui scrollbar vertical sau lângă săgeata din stânga a unui scrollbar orizontal, cauzând deplasarea unei pagini întregi în acea direcție.

```
public static final int SCROLL_PAGE_DOWN
```

Această constantă este setată unui eveniment care are loc atunci când utilizatorul apasă lângă săgeata de jos a unui scrollbar vertical sau lângă săgeata din dreapta a unui scrollbar orizontal, cauzând deplasarea unei pagini întregi în acea direcție.

Evenimente de focus

```
public static final int GOT_FOCUS
```

Acest eveniment apare când o anumită componentă este selectată. Metoda *FocusListener.focusGained()* poate fi folosită pentru tratarea acestui eveniment.

```
public static final int LOST_FOCUS
```

Acest eveniment apare când o anumită componentă este selectată. Metoda *FocusListener.focusLost()* poate fi folosită pentru tratarea acestui eveniment.

Metodele clasei Event

De obicei evenimentele vor fi declanșate de contextul exterior (apăsarea unui buton, a unei taste, etc). totuși dacă ne creăm propriile componente, sau dorim să comunicăm între thread-uri, atunci va trebui să ne creăm propriile noastre evenimente. Clasa *Event* are o serie de constructori și anume:

```
public Event (Object target, long when, int id, int x, int y, int key,
int modifiers, Object arg)
```

Primul constructor este și cel mai complet, și inițializează toate câmpurile obiectului cu parametrii specificați. Ceilalți doi constructori vor omite câteva inițializări dupa cum urmează:

```
public Event (Object target, long when, int id, int x, int y, int key, int
modifiers)
public Event (Object target, int id, Object arg)
```

Metode specifice modificatorilor

Aceste metode servesc la verificarea valorilor măștilor modificatorilor.

```
public boolean shiftDown ()
```

Metoda `shiftDown()` returnează *true* dacă tasta *Shift* a fost apăsată sau *false* în caz contrar. Metode asemănătoare există și pentru celelalte taste și anume *Ctrl* sau *Alt*.

Modelul nou de evenimente

Deși pare mai complex (modelul este format din mai multe părți) este mai simplu și mai eficient. Acest model solicită obiectelor să se aboneze pentru a primi evenimente. Acest model se numește “delegare” și implementează modelul Observer-Observable descris în cursul anterior. Reluând, fiecare componentă este o sursă de evenimente ce poate genera diverse tipuri de evenimente, care sunt de fapt, subclase a clasei *AWTEvent*. Obiectele interesate de un eveniment se numesc *ascultători*. Fiecare tip de eveniment corespunde unei interfețe de ascultare ce specifică metodele care sunt apelate atunci când un eveniment are loc.

Ierarhia de evenimente și clasele cu rol de ascultători pentru acestea, este reprezentată în figura de mai jos:

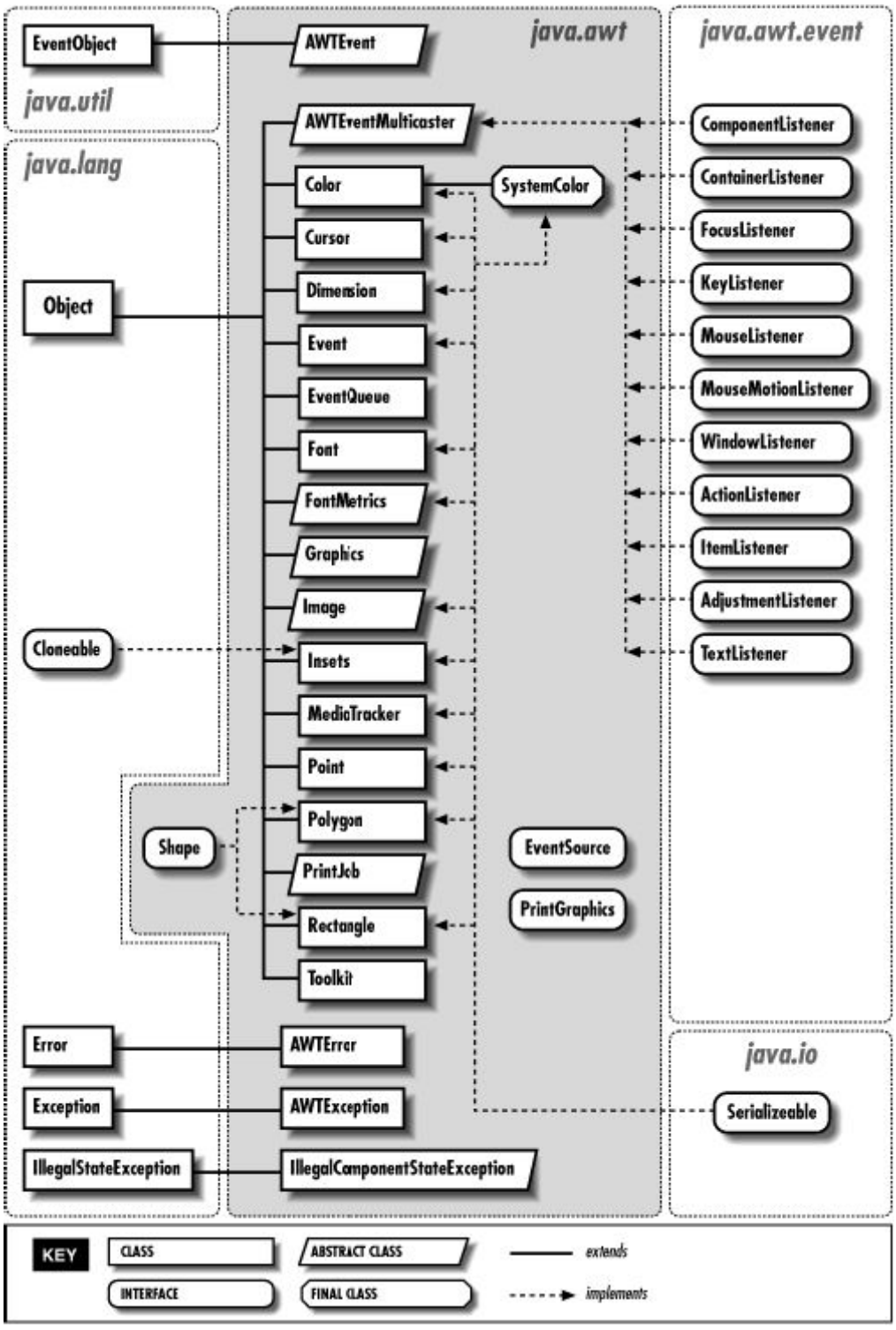


Figura 10.10 Modelul AWTEvent

Unele interfețe de ascultători sunt construite astfel încât să trateze mai multe evenimente. De exemplu, interfața *MouseInterface* declară cinci metode pentru a trata diverse evenimente de mouse. Aceasta înseamnă că un obiect interesat de evenimente de mouse trebuie să implementeze interfața *MouseListener* și va trebui să implementeze toate cele cinci metode.

Ce se întâmplă dacă dorim să tratăm doar un eveniment de mouse? Suntem obligați să scriem cod de patru ori in plus? Din fericire pachetul *java.awt.event* include clase *adapter*, ce sunt scurtături ce permit scrierea facilă a acestor handler-e de evenimente. Clasa *adapter* oferă implementări nule a tuturor metodelor interfeței. De exemplu *MouseAdapter* oferă implementări implicite alea metodelor *mouseEntered()*, *mouseExisted()*, etc. Putem astfel să ne concentrăm pe metoda care ne interesează la un moment dat și anume *mouseClicked()*.

Clasa AWTEvent

Variabile

```
protected int id
```

Câmpul *id* al clasei *AWTEvent* este accesibil prin intermediul *getID()*. Este identificatorul tipului de eveniment, ca de exemplu ACTION_PERFORMED al evenimentului *ActionEvent* sau MOUSE_MOVE.

Constante

Constantele clasei *AWTEvent* sunt folosite pentru a determina ce tip de tratare de eveniment va fi folosită și ce eveniment este procesat. Mai jos este o listă cu acestea:

public final static long ACTION_EVENT_MASK
public final static long ADJUSTMENT_EVENT_MASK
public final static long COMPONENT_EVENT_MASK
public final static long CONTAINER_EVENT_MASK
public final static long FOCUS_EVENT_MASK
public final static long ITEM_EVENT_MASK
public final static long KEY_EVENT_MASK
public final static long MOUSE_EVENT_MASK
public final static long MOUSE_MOTION_EVENT_MASK
public final static long TEXT_EVENT_MASK
public final static long WINDOW_EVENT_MASK

Aceste constante sunt folosite în corelație cu metoda *Component.eventEnabled()*.

Constructori

Clasa *AWTEvent* este abstractă, constructorii nu pot fi folosiți imediat. Ei sunt apelați la instanțierea unei clase copil.

```
public AWTEvent(Event event)
public AWTEvent(Object source, int id)
```

Primul constructor creează un eveniment de tip *AWTEvent* folosindu-se de unul din versiunea veche Java. Al doilea creează un *AWTEvent* folosind o sursă dată. Parametrul `id` folosește ca identificator al tipului de eveniment.

Metode

```
protected void consume()
```

Metoda `consume()` este apelată pentru a menționa faptul că un eveniment a fost tratat. Un eveniment care a fost marcat astfel, este livrat mai departe celorlalți ascultători daca este cazul. Doar evenimentele de tastatură și mouse pot fi marcate ca și consumate. Un scenariu de folosire a acestei facilități este atunci când dorim să refuzăm introducerea anumitor caractere, practic acele caractere nu sunt afișate.

```
protected boolean isConsumed()
```

Metoda returnează faptul că un eveniment a fost sau nu consumat.

Există o serie de alte clase de tratare a evenimentelor, iar acestea sunt specifice tipului de control GUI la care se referă evenimentul. Iată o listă a acestor clase:

Clasa	**Descriere**
ComponentEvent	Reprezintă evenimentul ce are loc în cadrul unui *Component*
ContainerEvent	Include evenimentele ce rezultă din operații dintr-un *Container*
FocusEvent	Conține evenimentele ce sunt generate la primirea/pierderea focusului
WindowEvent	Se ocupă de evenimentele specifice unei ferestre
PaintEvent	Încapsulează evenimentele specifice acțiunii de desenare. Aici este un caz special deoarece nu există clasa PaintListener. Se vor folosi totuşi metodele paint() şi update() pentru a procesa evenimentele.
InputEvent	Este clasa abstractă de bază, pentru tratarea evenimentelor de tastatură şi mouse
KeyEvent	Se ocupă de evenimentele de tastatură
MouseEvent	Se ocupă de evenimentele de mouse
ActionEvent	Este prima clasă din ierarhia acestor clase. Încapsulează evenimentele care semnalează faptul că utilizatorul realizează o acțiune asupra unei componente.
AdjustmentEvent	Este altă clasă din ierarhia claselor de evenimente. Încapsulează evenimente care reprezintă mişcările unui scrollbar.

ItemEvent	Este clasa ce permite tratarea evenimentelor ce au loc atunci când utilizatorul Selectează un checkbox sau un buton radio etc.
TextEvent	Încapsulează evenimentele ce au loc atunci când conținutul unui TextComponent s-a schimbat.

Pentru toate aceste elemente descrise există interfețe de ascultători şi clase adaptori. De exemplu pentru clasa *ActionEvent* există interfața *ActionListener* ce conține metode pentru tratarea evenimentelor de acest tip.

Pentru clasa *ComponentEvent* există interfața *ComponentListener* ce conține patru metode pentru mutarea şi redimensionarea componentei. Clasa *ComponentAdapter* este adaptorul corespunzător, folosit mai ales când se doreşte ascultarea unui eveniment specific.

Clasa AWTEventMulticaster

Aceasta este folosită de AWT pentru a trata cozile de ascultători pentru diverse evenimente, şi pentru a trimite evenimente tuturor ascultătorilor interesați (*multicasting*). Iată cum se foloseşte această clasă:

```
public static ActionListener add(ActionListener first, ActionListener
second)
```

Metoda de mai sus are doi parametrii de tip `ActionListener` şi returnează tot un `ActionListener`. Acest obiect returnat este un *multicaster* ce conține doi ascultători. De asemenea se poate construi un adevărat lanț de ascultători astfel:

```
actionListenerChain=AWTEventMulticaster.add(actionListenerChain,
newActionListener);
```

Pentru a transmite un eveniment lanțului de ascultători astfel format avem metoda *actionPerformed()*:

```
actionListenerChain.actionPerformed(new ActionEvent(...));
```

Evident există metode pentru toate tipurile de evenimente, şi acestea sunt:

```
public void actionPerformed(ActionEvent e)
public void adjustmentValueChanged(AdjustmentEvent e)
public void componentAdded(ContainerEvent e)
public void componentHidden(ComponentEvent e)
public void componentMoved(ComponentEvent e)
public void componentRemoved(ContainerEvent e)
public void componentResized(ComponentEvent e)
public void componentShown(ComponentEvent e)
public void focusGained(FocusEvent e)
public void focusLost(FocusEvent e)
public void itemStateChanged(ItemEvent e)
public void keyPressed(KeyEvent e)
```

```
public void keyReleased(KeyEvent e)
public void keyTyped(KeyEvent e)
public void mouseClicked(MouseEvent e)
public void mouseDragged(MouseEvent e)
public void mouseEntered(MouseEvent e)
public void mouseExited(MouseEvent e)
public void mouseMoved(MouseEvent e)
public void mousePressed(MouseEvent e)
public void mouseReleased(MouseEvent e)
public void textValueChanged(TextEvent e)
public void windowActivated(WindowEvent e)
public void windowClosed(WindowEvent e)
public void windowClosing(WindowEvent e)
public void windowDeactivated(WindowEvent e)
public void windowDeiconified(WindowEvent e)
public void windowIconified(WindowEvent e)
public void windowOpened(WindowEvent e)
```

Mai jos avem un exemplu de folosire a unui obiect de *multicasting*.

```
import java.awt.*;
import java.awt.event.*;
class ItemEventComponent extends Component implements ItemSelectable
{
    boolean selected;
    int i = 0;
    ItemListener itemListener = null;
    ItemEventComponent ()
    {
        enableEvents (AWTEvent.MOUSE_EVENT_MASK);

    }
    public Object[] getSelectedObjects()
    {
        Object o[] = new Object[1];
        o[0] = new Integer (i);
        return o;
    }
    public void addItemListener (ItemListener l)
    {
        itemListener = AWTEventMulticaster.add (itemListener, l);
    }
    public void removeItemListener (ItemListener l)
    {
        itemListener = AWTEventMulticaster.remove (itemListener, l);
    }

    public void processEvent (AWTEvent e)
    {
```

```
        if (e.getID() == MouseEvent.MOUSE_PRESSED)
        {
            if (itemListener != null)
            {
                selected = !selected;
                i++;
                itemListener.itemStateChanged (
                new ItemEvent (this, ItemEvent.ITEM_STATE_CHANGED,
                getSelectedObjects(),
                (selected?ItemEvent.SELECTED:ItemEvent.DESELECTED)));
            }
        }
    }
}
public class Multicasting extends Frame implements ItemListener
{
    Multicasting ()
    {
        super ("Listening In");
        ItemEventComponent c = new ItemEventComponent ();
        add (c, "Center");
        c.addItemListener (this);
        c.setBackground (SystemColor.control);
        setSize (200, 200);
    }
    public void itemStateChanged (ItemEvent e)
    {
        Object[] o = e.getItemSelectable().getSelectedObjects();
        Integer i = (Integer)o[0];
        System.out.println (i);
    }
    public static void main (String args[])
    {
        Multicasting f = new Multicasting();
        f.show();
    }
}
```

În acest exemplu se prezintă modul de a folosi `AWTEventMulticaster` pentru a crea componente ce generează evenimente de tip `ItemEvent`. Clasa `AWTEventMulticaster` este folosită pentru a crea un obiect multicast prin metoda `AWTEventMulticaster.add()`. Metoda `itemStateChanged()` notifică pe oricine este interesat. Evenimentul de tip item este generat la apăsarea butonului de mouse. Din moment ce nu avem ascultători pentru mouse, trebuie să apelăm metoda `enableEvents` pentru mouse.

Componente

În continuare vom prezenta clasa *Component*, împreună cu unele componente specifice cum ar fi *Label*, *Button* și *Canvas*. De asemenea vom arunca o privire și asupra clasei *Cursor*.

Clasa Component

Fiecare aplicație cu interfață grafică constă dintr-un set de obiecte. Aceste obiecte sunt de tip *Component*, iar cele mai frecvent derivate din această clasă, sunt butoanele, câmpurile de text și container-ele.

Constante

Acestea sunt folosite pentru a specifica cum ar trebui aliniată componenta curentă:

```
public static final float BOTTOM_ALIGNMENT
public static final float CENTER_ALIGNMENT
public static final float LEFT_ALIGNMENT
public static final float RIGHT_ALIGNMENT
public static final float TOP_ALIGNMENT
```

Metode

```
protected Component()
```

Constructorul este declarat `protected` deoarece clasa *Component* este abstractă. Constructorul permite crearea unei componente fără un tip anume (buton, text etc).

```
public Toolkit getToolkit ()
```

Metoda returnează `Toolkit-ul` actual al componentei. Prin acest `Toolkit` avem acces la detaliile platformei curente (ca rezoluția, mărimea ecranului, sau fonturi disponibile).

```
public Color getForeground ()
```

Metoda returnează culoarea de prim plan. Dacă nu există o culoare setată, se va lua culoarea părintelui componentei actuale. Dacă nici părintele nu are culoare de prim plan se returnează *null*.

```
public void setForeground (Color c)
```

Metoda modifică culoarea de prim plan prin parametrul de tip `Color c`.

```
public Color getBackground ()
```

```
public void setBackground (Color c)
```

Aceste două metode realizează același lucru ca cele de mai sus, dar se referă la culoarea fundalului.

```
public Font getFont ()
public synchronized void setFont (Font f)
```

Sunt metodele pentru preluarea/setarea fontului care va fi valabil pentru componenta curentă.

```
public Graphics getGraphics ()
```

Metoda `getGraphics` preia contextul grafic al componentei. Cele mai multe componente nu specifică, corect acest context și vor arunca excepția `InternalError` la apelul acestei metode.

```
public Locale getLocale ()
public void setLocale (Locale l)
```

Aceste metode modifică sau ajută la preluarea setării *Locale* a componentei curente. Localizarea este o parte dintr-un subiect al internaționalizării programelor Java și nu îl vom aborda acum.

```
public Cursor getCursor ()
public synchronized void setCursor (Cursor c)
```

Aceste două metode preiau/modifică informațiile legate de forma și dimensiunea mouse-ului.

```
public Point getLocation ()
public Point location ()
```

Aceste metode returnează poziția curentă a *Component*-ei, poziție relativă la părintele acestei structuri. Punctul returnat prin `Point` este colțul stânga sus ce delimitează dreptunghiul componentei.

```
public void setLocation (int x, int y)
public void move (int x, int y)
```

Metodele au ca efect mutarea *Component*-ei în noua poziție (*x*,*y*). Coordonatele se referă la colțul stânga sus al dreptunghiului ce delimitează componenta și sunt relative la părintele acesteia.

```
public Dimension getSize ()
public Dimension size ()
public void setSize (int width, int height)
public void resize (int width, int height)
```

Primele două metode ajută la preluarea dimensiunii componentei, în timp ce următoarele două, setează noua dimensiune a componentei.

```
public float getAlignmentX ()
public float getAlignmentY ()
```

Aceste două metode returnează aliniamentul componentei referitor la axa *X* respectiv *Y*. acest aliniament poate fi folosit ulterior de managerul de aspect pentru a poziționa componente relativ la altele. Valoarea returnată este între 0 și 1, unde o valoare apropiată de 0 indică o apropiere de stânga în timp ce o valoare apropiată de 1 indică o deplasare către dreapta.

```
public void doLayout ()
public void layout ()
```

Metodele cauzează validarea componentei curente.

```
public boolean contains (int x, int y)
public boolean inside (int x, int y)
```

Aceste două metode verifică dacă, punctul de coordonate *x* și *y* se află în cadrul dreptunghiului ce delimitează componenta. Dacă această componentă nu este rectangulară, metoda acționează ca și când ar fi delimitată de un dreptunghi.

```
public Component getComponentAt (int x, int y)
public Component locate (int x, int y)
```

Metodele folosesc `contains` pentru a verifică dacă *x* și *y* se află în cadrul componentei. Dacă dac, metoda returnează instanța componentei. Dacă nu, returnează *null*.

```
public void paint (Graphics g)
```

Metoda permite afișarea diverselor lucruri în cadrul componentei. Se poate suprascrie, asemenea cum am prezentat în cursul anterior pentru applet-uri.

```
public void repaint ()
```

Există o serie de funcții de redesenare, care în principiu realizează același lucru: redesenarea componentei cât mai repede cu putință.

```
public void print (Graphics g)
```

Apelul implicit al acestei metode este către `paint`. Dar se poate imprima conținutul unei componente, dacă parametrul `Graphics` implementează *PrintGraphics*.

Metode de vizualizare

Pachetul AWT conține de asemenea clasa abstractă *Image*, care se ocupă cu tratarea imaginilor, provenite din diverse surse, cele mai întâlnite fiind fișierele de imagini.
Un mic exemplu pentru folosirea acestei clase, pentru încărcarea unei imagini și afișarea ei se află mai jos:

```
import java.awt.*;
public class Imaging extends Frame
{
    Image im;
    Imaging ()
    {
        im = Toolkit.getDefaultToolkit().getImage ("c:\\1.jpg");
    }
    public void paint (Graphics g)
    {
        g.drawImage (im, 0, 0, 175, 225, this);
    }
    public boolean mouseDown (Event e, int x, int y)
    {
        im.flush();
        repaint();
        return true;
    }
    public static void main (String [] args)
    {
        Frame f = new Imaging();
        f.setVisible(true);
    }
}
```

Revenind la clasa Component, aceasta suportă lucrul cu imagini prin intermediul clasei *Image* și a câtorva funcții ce vor fi descrise în cele ce urmează.

```
public boolean imageUpdate (Image image, int infoflags, int x, int y, int
width, int height)
```

Această metodă provine din interfața *java.awt.image.ImageObserver* implementată de clasa Component. Permite actualizarea asincronă a interfeței pentru a primi notificări despre imaginea dată de parametrul `image`. Această metodă este necesară pentru că încărcarea imaginii se face pe un thread separat. Parametrii *x*,*y* specifică poziția imaginii încărcate, iar *width* și *height* specifică dimensiunea imaginii. Parametrul `infoflags` este o mască pe biți ce permite setarea diferitelor opțiuni de update.

```
public Image createImage (int width, int height)
```

Metoda creează o imagine goală de mărime *width* și *height*. Imaginea returnată este o imagine stocată în memorie. Dacă nu poate fi creată, atunci apelul metodei returnează *null*.

```
public boolean prepareImage (Image image, ImageObserver observer)
```

Metoda forțează încărcarea unei imagini, asincron, în alt thread. Obiectul `observer` este componenta care va fi afișată de imaginea `image` la încărcarea imaginii. Dacă `image` a fost deja încărcată atunci funcția returnează *true*. în caz contrar se returnează *false*. Din moment de `image` este încărcată asincron, `prepareImage` returnează imediat ce este apelată.

```
public int checkImage (Image image, ImageObserver observer)
```

Metoda returnează statusul construcției unei reprezentări a imaginii, status raportat de `observer`. Dacă imaginea nu s-a încărcat, atunci apelul metodei nu va produce acest lucru. Valoarea returnată este dată de flag-urile obiectului `observer` în SAU logic valabile pentru datele disponibile. Acestea sunt: WIDTH, HEIGHT, PROPERTIES, SOMEBITS, FRAMEBITS, ALLBITS, ERROR, și ABORT.

```
public void addNotify ()
```

Metoda este suprascrisă de fiecare componentă în parte. Atunci când este apelată componenta este invalidată. Metoda este apelată de sistem la crearea componentei, sau atunci când componenta este adăugată la un *Container* și acesta era deja afișat.

```
public synchronized void removeNotify ()
```

Metoda distruge conținutul componentei și o va șterge de pe ecran. Informația despre statusul componentei este reținut într-un subtip specific.

```
public void setVisible(boolean condition)
public void show (boolean condition)
```

Cele două metode, din care ultima este invechită, afișează pe ecran componenta (în cazul metodei `setVisible` doar dacă variabila booleană este setată pe *true*). Părintele *Container*, trece în starea *invalid*, deoarece cel puțin un copil a modificat aspectul.

```
public void hide ()
```

Metoda ascunde componenta și are același efect ca și metoda `setVisible` apelată cu parametru *false*.

```
public synchronized void enable ()
public synchronized void disable ()
public void setEnabled (boolean condition)
```

Metodele stabilesc dacă utilizatorul poate accesa/interacționa cu aceste componente sau nu.

```
public void requestFocus ()
```

Această metodă permite inițializarea unei cereri ca și componenta să recepționeze focus.

```
public boolean isFocusTraversable()
```

Această metodă verifică dacă acea componentă este capabilă de a recepționa focus sau nu.

```
public void nextFocus ()
```

Metoda permite transferarea focus-ului către următoarea componentă.

Evenimentele unei Component

O clasă *Component* suportă metodele învechite de control al evenimentelor, discutate mai sus: *deliverEvent*, *postEvent* și *handleEvent*, dar și metode noi de a posta noi evenimente ca:

```
public final void dispatchEvent(AWTEvent e)
```

Pe lângă acestea există o serie de funcții ce sunt învechite cum ar fi *keyUp*, sau *mouseDown* ce pot fi folosite pentru captarea evenimentelor de buton sau taste. Pentru tratarea evenimentelor apărute în cadrul unui *Component* există modelul delegat și o serie de funcții pentru înregistrarea sau deînregistrarea ascultătorilor:

```
public void addComponentListener(ComponentListener listener)
public void removeComponentListener(ComponentListener listener)
public void addFocusListener(FocusListener listener)
public void removeFocusListener(FocusListener listener)
public void addKeyListener(KeyListener listener)
public void removeKeyListener(KeyListener listener)
public void addMouseListener(MouseListener listener)
public void removeMouseListener(MouseListener listener)
public void addMouseMotionListener(MouseMotionListener listener)
public void removeMouseMotionListener(MouseMotionListener listener)
```

Pentru a putea trata evenimentele (asta dacă nu dorim neapărat înregistrarea la un anumit eveniment) trebuie să permitem captarea acestora. Aceasta se realizează prin funcția:

```
protected final void enableEvents(long eventsToEnable)
```

Clasa *AWTEvent* permite definirea unor constant ce permit verificarea tipului de eveniment ce a avut loc. Tipul evenimentului este dat de variabila `eventsToEnable`.

```
protected final void disableEvents(long eventsToDisable)
```

Această metodă permite oprirea recepționării unor evenimente pentru componenta curentă.

```
protected void processEvent(AWTEvent e)
```

Metoda primește toate evenimentele AWTEvent ce au ca țintă componenta curentă. Aceste evenimente vor fi transmise celorlalte metode specifice, pentru o procesare ulterioară. Atunci când se suprascrie metoda `processEvent`, este posibilă procesarea evenimentelor fără a înregistra ascultători.

```
protected void processComponentEvent(ComponentEvent e)
```

Această metodă primește un eveniment de tipul `ComponentEvent` ce are drept țintă componenta actuală. Dacă sunt abonați ascultători, aceștia sunt notificați. Suprascrierea metodei `processComponentEvent` este echivalentă cu redimensionarea componentei.

```
protected void processFocusEvent(FocusEvent e)
```

Metoda este apelată atunci când componenta primește focus. Suprascrierea acestei metode este echivalentă cu suprascrierea metodelor *gotFocus()* sau *lostFocus()*.

```
protected void processKeyEvent(KeyEvent e)
```

Metoda primește un eveniment de tastatură, ce are ca țintă componenta curentă. În caz de suprascriere, dacă dorim ca procesarea să decurgă normal trebuie apelată și metoda `super.processKeyEvent(e)` , altfel evenimentele nu vor ajunge la ascultătorii înregistrați. Suprascrierea metodei este similară metodelor *keyDown()* sau *keyUp()* din modelul vechi.

```
protected void processMouseEvent(MouseEvent e)
```

Această metodă este similară cu cea anterioară cu specificația că se ocupă cu evenimente de mouse.

Etichete

O etichetă, este reprezentată de clasa *Label*, ce derivă din Component și afișează o linie de text. Este folosită mai ales la stabilirea unor titluri sau a atașa text altei componente.

Constante

```
public final static int LEFT
public final static int CENTER
public final static int RIGHT
```

Acestea oferă aliniamentul textului din etichetă.

Metode

```
public Label ()
public Label (String label)
public Label (String label, int alignment)
```

Aceștia sunt constructorii acestei clase, prin care se permite instanțierea unei etichete, și specificarea textului afișat `label` sau a aliniamentului.

```
public String getText ()
public void setText (String label)
```

Aceste metode permit preluarea textului etichetei respectiv setarea acestuia.

```
public int getAlignment ()
public void setAlignment (int alignment)
```

Aceste două metode permit preluarea/specificarea aliniamentului unei etichete. Dacă aliniamentul nu este valid, metoda de setare va arunca o excepție de tip *IllegalArgumentException*. Clasa *Label* poate reacționa la evenimentele pe care le recepționează, deși nu este scopul ei.

Butoane

Clasa Button derivă de asemenea din Component și se folosește pentru declanșarea unor acțiuni.

Metode

```
public Button ()
public Button (String label)
```

Aceștia sunt constructorii, ultimul permițând crearea unui buton a cărui text este `label`.

```
public String getLabel ()
public synchronized void setLabel (String label)
```

Aceste două metode permit preluarea/specificarea textului afișat pe un buton.

```
public String getActionCommand ()
public void setActionCommand (String command)
```

Fiecare buton poate avea două nume. Unul este ceea ce vede utilizatorul și celălalt este folosit de programator și se numește comanda butonului. Aceasta a fost introdusă pentru internaționalizarea programelor. De exemplu daca eticheta butonului conține *Yes* însă programul rulează pe sistem francez sau german, eticheta va afișa Oui sau Ja. Oricare ar fi sistemul pe care rulează comanda butonului poate rămâne Yes.

```
public synchronized void addNotify ()
```

Metoda permite adăugarea butonului ca și componentă, permițând-ui acestuia să își schimbe aspectul păstrând funcționalitatea.

Evenimentele unui buton sunt cele de mouse, de focus de tastatură practic cele prezentate mai sus în clasa Component.

Pentru a înțelege mai bine evenimentele și diferența între numele comenzii și numele afișat avem următorul cod:

```
import java.applet.*;
import java.awt.event.*;
import java.awt.*;
public class Buton extends Applet implements ActionListener
{
    Button b;
    public void init ()
    {
        add (b = new Button ("One"));
        b.addActionListener (this);
        add (b = new Button ("Two"));
        b.addActionListener (this);
        add (b = new Button ());
        b.setLabel("Drei");
         b.setActionCommand("Three");
        b.addActionListener (this);
        add (b = new Button ());
        b.setLabel("Quatre");
        b.setActionCommand("Four");
        b.addActionListener (this);
    }
    public void actionPerformed (ActionEvent e)
    {
        String s = e.getActionCommand();
        if ("One".equals(s))
        {
            System.out.println ("Pressed One");
        }else if ("Two".equals(s)) {
            System.out.println ("Pressed Two");
        } else if ("Three".equals(s))
        {
```

```
            System.out.println ("Pressed Three");
        } else if ("Four".equals(s))
        {
            System.out.println ("Pressed Four");
        }
    }
}
```

Deși etichetele ultimelor butoane sunt altfel decât *Four* sau *Three* mesajul va fi afișat pentru că se compară numele acțiunii butonului.

Clasa Canvas

Aceasta este o clasă pe servește ca suport pentru noi componente ce pot fi create pe lângă cele deja oferite de librăria AWT. *Canvas* poate fi folosită și ca loc de a desena componente adiționale pe ecran, sau folosind *Graphics* se pot desena pe un *Canvas* diverse figuri.

Cel mai bine este ca un obiect *Canvas* să stea în interiorul unui *Container* în cazul aplicațiilor mai complexe. Mai jos este un exemplu de folosirea a aceste clase.

```
import java.awt.Canvas;
import java.awt.Graphics;
import java.applet.Applet;

public class CanvasTest extends Applet
{
  public void init()
  {
    DrawingRegion region = new DrawingRegion();
    add(region);
  }
}
class DrawingRegion extends Canvas
{
  public DrawingRegion()
  {
    setSize(100, 50);
  }
  public void paint(Graphics g)
  {
    g.drawRect(0, 0, 99, 49); // draw border
    g.drawString("A Canvas", 20,20);
  }
}
```

În cadrul acestui exemplu clasa `DrawingRegion` ce moștenește *Canvas* este folosită în cadrul unui *Applet*, mai exact adăugată în cadrul ferestrei principale.

Clasa Cursor

Aceasta oferă diferite forme și dimensiuni pentru a reprezenta pointer-ul mouse-ului. Iată mai jos constantele ce reprezintă acele forme

```
public final static int DEFAULT_CURSOR
public final static int CROSSHAIR_CURSOR
public final static int TEXT_CURSOR
public final static int WAIT_CURSOR
public final static int HAND_CURSOR
public final static int MOVE_CURSOR
public final static int N_RESIZE_CURSOR
public final static int S_RESIZE_CURSOR
public final static int E_RESIZE_CURSOR
public final static int W_RESIZE_CURSOR
public final static int NE_RESIZE_CURSOR
public final static int NW_RESIZE_CURSOR
public final static int SE_RESIZE_CURSOR
public final static int SW_RESIZE_CURSOR
```

Metode

```
public int getType()
```

Metoda returnează tipul cursorului și este egal ca valoare cu una din constantele clasei.

```
static public Cursor getPredefinedCursor(int type)
```

Metoda returnează un cursor predefinit. Dacă tipul nu este egal cu una din constante, metoda aruncă excepția `IllegalArgumentException`. Metoda verifică dacă obiectul de tip *Cursor* există deja și dacă da returnează referința la obiectul existent.

Iată un exemplu pentru utilizarea acestei componente:

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

public class CursorTest
{
  public static void main(String[] args)
  {
    Frame f = new Frame("Change cursor");
    Panel panel = new Panel();
```

```
    Button comp1 = new Button("Ok");
    Button comp2 = new Button("Cancel");
    panel.add(comp1);
    panel.add(comp2);
    f.add(panel,BorderLayout.CENTER);
    f.setSize(200,200);
    f.setVisible(true);
    Cursor cur = comp1.getCursor();
    Cursor cur1 = comp2.getCursor();

    comp1.setCursor(Cursor.getPredefinedCursor(Cursor.HAND_CURSOR));
    comp2.setCursor(Cursor.getPredefinedCursor(Cursor.MOVE_CURSOR));

    f.addWindowListener(new WindowAdapter()
    {
      public void windowClosing(WindowEvent we){
        System.exit(0);
      }
    });

  }
}
```

TextComponent

Există două moduri de a introduce date: de la tastatură sau prin mișcările *mouse*-ului. Pentru introducerea de caractere există două clase *TextField* și *TextArea*. Acestea fac parte din clasa părinte *TextComponent*.

```
public String getText ()
public void setText (String text)
```

Metodele ajută la preluarea/modificarea textului din cadrul acestei componente. Dacă este vorba de o componentă de tip *TextArea*, atunci sunt permise și caracterele \n astfel că textul va apare pe mai multe linii.

Utilizatorii pot selecta textul din cadrul acestei componente folosind mouse-ul sau tastatura. Pentru a lucra cu textul selectat avem următoarele funcții:

```
public int getSelectionStart ()
```

Metoda returnează poziția inițială a textului selectat. Poziția poate fi considerată numărul de caractere ce precedă primul caracter selectat. Dacă nu este selectat text, se returnează poziția cursorului. Valoarea de start de la începutul textului este 0.

```
public int getSelectionEnd ()
```

Această metodă returnează poziția cursorului ce indică sfârșitul selecției curente. Dacă nu este selectat nimic atunci se va returna poziția curentă a cursorului.

```
public String getSelectedText ()
```

Această metodă returnează textul selectat sub forma unui *String*, sau *null* dacă nu este nimic selectat.

```
public void setSelectionStart (int position)
public void setSelectionEnd (int position)
```

Aceste două metode modifică selecția actuala a textului după parametrul `position`.

```
public void setEditable (boolean state)
public boolean isEditable ()
```

Aceste metode sunt pentru a activa sau dezactiva un TextComponent. Mai jos este un exemplu pentru a demonstra folosirea acestor clase:

```
import java.awt.*;
import java.applet.*;
public class TestText extends Applet
{
    TextArea area;
    Label label;
    public void init () {
    setLayout (new BorderLayout (10, 10));
    add ("South", new Button ("toggleState"));
    add ("Center", area = new TextArea ("Area to write", 5, 10));
    add ("North", label = new Label ("Editable", Label.CENTER));
    }
    public boolean action (Event e, Object o)
    {
        if (e.target instanceof Button)
        {
            if ("toggleState".equals(o))
            {
                area.setEditable (!area.isEditable ());
                label.setText ((area.isEditable () ? "Editable" : "Read-
only"));
            return true;
        }
        }
        return false;
    }
}
```

Pe lângă evenimentele cunoscute se poate trata modificarea textului componentei folosind următoarea funcție:

```
public synchronized void addTextListener(TextListener listener)
```

Această metodă permite înregistrarea obiectului `listener` pentru a primi notificări atunci când are loc un eveniment de tip *TextEvent*. Metoda *listener.textValueChanged()* este apelată atunci când aceste evenimente au loc. Mai jos este un exemplu pentru folosirea acestora:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
class TextFieldSetter implements ActionListener
{
    TextField tf;
    TextFieldSetter (TextField tf)
    {
        this.tf = tf;
    }
    public void actionPerformed(ActionEvent e)
    {
        if (e.getActionCommand().equals ("Set"))
        {
            tf.setText ("You Just Say Hello");
        }
    }
}
public class TextEvent1 extends Applet implements TextListener
{
    TextField tf;
    int i=0;
    public void init ()
    {
        Button b;
        tf = new TextField ("Hello", 20);
        add (tf);
        tf.addTextListener (this);
        add (b = new Button ("Set"));
        b.addActionListener (new TextFieldSetter (tf));
    }
    public void textValueChanged(TextEvent e)
    {
        System.out.println (++i + ": " + e);
    }
}
```

Celelalte tipuri de componente, precum listele, checkbox-urile sau combobox-urile vor fi abordate în cursul următor, urmând ca acum să studiem cum se pot grupa aceste componente.

Clasa Container

Clasa *Container* este o subclasă a *Component*, ce conține alte componente, inclusiv alte recipiente. *Container* permite crearea de grupuri de obiecte ce apar pe ecran. în continuare vor fi atinse câteva din aceste clase. Fiecare obiect *Container* are un o schiță, o reprezentare ce permite organizarea componentelor din acel recipient. Toate aceste aspecte vor fi detaliate mai târziu când vom vorbi despre *Layout*.

Container

Container este o clasă abstractă ce oferă suport pentru alte obiecte de tip *Component*. Această clasă conține metode pentru a grupa elementele de tip *Component* şi a trata evenimentele care apar în cadrul ei. Deoarece este o clasă abstractă, nu poate fi instanțiat un obiect de acest tip, de aceea ea trebuie derivată şi apoi utilizată.

Constructori

Constructorul clasei ce moşteneşte *Container* permite instanțierea unui obiect şi asocierea unui *Layout* şi anume folosind un manager pentru aspect. Următorul cod exemplifică acest lucru:

```
import java.awt.*;
public class LightweightPanel extends Container
{
      LightweightPanel () {}
      LightweightPanel (LayoutManager lm)
      {
            setLayout(lm);
      }
}
```

Gruparea elementelor

Aceste metode descriu modul de lucru cu obiectele conținute de Container:

```
public int getComponentCount()
public int countComponents()
```

Metoda `getComponentCount()` returnează numărul de componente din cadrul unui *Container* iar `countComponents()` este o veche metodă dintr-o versiune Java 1.0.

Pentru a prelua elementele dintr-un *Container* avem la dispoziție metodele:

```
public Component getComponent (int position)
public Component[] getComponents()
```

Prima metodă returnează o componentă cu un anumit index. Dacă poziția este incorectă va fi aruncată următoarea excepție: *ArrayIndexOutOfBoundsException*.
A doua metodă returnează un şir ce conține toate componentele din Container-ul curent. Orice modificare a oricărui element din acest şir va apare imediat pe ecran.

Adăugarea elementelor

```
public Component add (Component component, int position)
```

Metoda *add()* adaugă o componentă în Container, la o anumită poziție. Dacă `position` este -1 inserarea va avea loc la sfârşit. Dacă `position` este incorect, metoda va arunca excepția *IllegalArgumentException.* Dacă se încearcă adăugarea Container-ului curent la sine însuşi metoda va arunca aceeaşi excepție. Dacă totul merge perfect, componenta este adăugată în recipient, şi metoda returnează obiectul `Component` adăugat.

```
public Component add (Component component)
```

Această metodă adaugă un `component` ca ultim obiect al *Container-ului* . Aceasta se face prin apelul metodei anterioare cu `position` egal cu -1.

```
public void add (Component component, Object constraints)
```

Această metodă este necesară pentru situațiile în care avem componente Layout ce solicită informații în plus pentru a dispune componentele pe ecran. Informația adițională este specificată prin parametrul `constraints`. Parametrul `constraints` depinde de managerul de interfață: *LayoutManager*. Poate fi folosit pentru a denumi Container-e din cadrul unui *CardLayout,* sau specifica suprafața unui *BorderLayout*, etc.

```
public Component add (String name, Component component)
```

Această metodă permite folosirea metodei anterioare, în care *String-ul* folosit defineşte o anumită constrângere. Apelarea aceste metode va genera un *ContainerEvent* cu id-ul COMPONENT_ADDED. Acestea sunt câteva din metodele de adăugare, însă mai sunt şi altele care pot fi folosite.

Ştergerea elementelor

```
public void remove (int index)
```

Metoda *remove* şterge componenta de la poziția indicată de `index`. Metoda va apela `removeLayoutComponent()` pentru a şterge componenta din *LayoutManager*.

```
public void removeAll ()
```

Metoda va şterge toate componentele din container. Atunci când este apelată, va genera un *ContainerEvent* cu id-ul COMPONENT_REMOVED.

Alte medode

```
public boolean isAncestorOf(Component component)
```

Metoda verifică faptul că `component` este părinte a acestui *Container*. Va returna *true* în caz afirmativ, altfel va returna *false*.

Metode de Layout

Fiecare container are un *LayoutManager*. Acesta este responsabil cu poziționarea componentelor în cadrul container-ului. Metodele listate sunt folosite pentru a dimensiona obiectele din *Container*.

```
public LayoutManager getLayout ()
```

Această metodă va returna obiectul `LayoutManager` asociat *Container-ului* curent.

```
public setLayout (LayoutManager lm)
```

Metoda schimbă obiectul `LayoutManager` al *Container-ului* curent şi-l invalidează. Componentele din cadrul *Container-ului* vor fi re-dispuse după regulile definite in obiectul `lm`. Dacă obiectul `lm` este *null* poziția componentelor din *Container* poate fi controlată de programator.

```
public Dimension getPreferredSize ()
public Dimension preferredSize ()
```

Aceste returnează un obiect de tip `Dimension` ce conține mărimea preferată a componentelor din *Container*. *Container-ul* poate determina mărimea preferată apelând metoda *preferredLayoutSize()* care va returna spațiul necesar managerului pentru a aranja componentele. A doua metodă este vechea funcție din Java 1.0 pentru *preferredLayoutSize()*.

```
public Dimension getMinimumSize ()
public Dimension minimumSize ()
public Dimension getMaximumSize ()
```

Aceste metode returnează obiectul de tip `Dimension` care va calcula minimul/maximul de spațiu necesar (lățime şi lungime) pentru ca managerul de *Layout* să aranjeze componentele.

```
public float getAlignmentX ()
public float getAlignmentY ()
```

Aceste metode returnează aliniamentul componentelor din cadrul *Container-ului* pe componenta *x* respectiv *y*. Container-ul determină aliniamentul apelând metoda *getLayoutAlignmentX()*, respectiv *getLayoutAlignmentY()* din managerul de Layout actual. Valoarea returnată va fi între 0 şi 1. Valorile apropiate de 0 indică faptul că, componenta trebuie poziționată mai la stânga, iar cele apropiate de 1 indică faptul că, componenta trebuie poziționată mai la dreapta în cazul metodei *getLayoutAlignmentX()*. În cazul metodei *getLayoutAlignmentY()* 0 înseamnă mai aproape de partea de sus iar 1 mai aproape de partea de jos a suprafeței.

```
public void doLayout ()
public void layout ()
```

Aceste două metode indică managerului de Layout să afişeze Container-ul.

```
public void validate ()
```

Metoda setează starea de validitate a container-ului pe *true* şi validează recursiv componentele sale. Dacă acesta mai conține un *Container*, atunci şi acela va fi validat. Unele componente nu sunt inițializate până nu sunt validate. De exemplu, nu se pot interoga dimensiunile de afişare ale unui *Button*, până când acesta nu este validat.

```
public void invalidate ()
```

Metoda invalidează *Container-ul actual* şi componentele din cadrul său.

Evenimente

```
public void deliverEvent (Event e)
```

Această metodă este apelată atunci când eveniment de tip *Event* are loc. Metoda încearcă să localizeze o componentă din *Container* pentru a primi evenimentul aferent ei. Dacă este găsită, atunci coordonatele *x* şi *y* sunt transmise noii ținte, evenimentul *e* de tip Event este astfel livrat.

```
public Component getComponentAt (int x, int y)
```

Această metodă apelează metoda *contains()* din cadrul fiecărei componente din Container pentru a vedea dacă, coordonatele *x* şi *y* sunt în interiorul componentei. Dacă da, acea componentă este returnată. Dacă nu se găseşte nici o componentă care să respecte această cerință atunci obiectul Container este returnat. Dacă aceste coordonate sunt în afara Container-ului atunci se returnează *null*.

```
public Component getComponentAt (Point p)
```

Această metodă este identică cu cea anterioară, doar că locația se specifică printr-un obiect de tip *Point*.

Ascultători

```
public synchronized void addContainerListener(ContainerListener listener)
```

Metoda înregistrează un obiect `listener` interesat de a primi notificări atunci când un obiect de tip ContainerEvent trece prin coada de evenimente la care *Container-ul* curent este abonat.

Atunci când aceste evenimente au loc, metodele *listener.componentAdded()* sau *listener.componentRemoved()* sunt apelate. Evident că şi alți ascultători pot fi de asemenea înregistrați. În codul ce urmează a fi prezentat, este exemplificată folosirea unui `ContainerListener` pentru a înregistra ascultători pentru toate butoanele adăugate unui applet. Ceea ce face ca acest cod să funcționeze, este apelul funcției enableEvents() care are ca efect livrarea evenimentelor în absența ascultătorilor.

```
/*
<applet code="UsingContainer" width=300 height=50>
</applet>
*/
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

public class UsingContainer extends Applet implements ActionListener
{
    Button b;
    public void init()
    {
        enableEvents (AWTEvent.CONTAINER_EVENT_MASK);
        add (b = new Button ("One"));
        add (b = new Button ("Two"));
        add (b = new Button ("Three"));
        add (b = new Button ("Four"));
    }
    protected void processContainerEvent (ContainerEvent e)
    {
        if (e.getID() == ContainerEvent.COMPONENT_ADDED)
        {
            if (e.getChild() instanceof Button)
            {
                Button b = (Button)e.getChild();
                b.addActionListener (this);
            }
        }
    }
```

```
    public void actionPerformed (ActionEvent e)
    {
        System.out.println ("Selected: " + e.getActionCommand());
    }
}
```

```
public void removeContainerListener(ContainerListener listener)
```

Această metodă şterge un `listener`, adică va face ca acel obiect să nu mai fie abonat. Dacă `listener` nu este înregistrat, nu se întâmplă nimic.

```
protected void processEvent(AWTEvent e)
```

Metoda `processEvent()` primeşte toate evenimentele *AWTEvent* ce au *Container-ul* curent drept țintă. Metoda le va transmite tuturor ascultătorilor abonați pentru procesarea ulterioară. Se poate suprascrie această metodă pentru a preprocesa evenimentele înainte de a le transmite mai departe.

```
protected void processContainerEvent(ContainerEvent e)
```

Metoda `processContainerEvent`, folosită în programul de mai sus, va primi toate evenimentele de tip *ContainerEvent* ce au ca țintă *Container-ul* curent. Apoi, aceste evenimente se vor transmite la orice ascultător pentru procesare ulterioară. Suprascrierea metodei permite preprocesarea evenimentelor înainte ca acestea să fie retransmise.

Panel

Această clasă oferă un *Container* generic pentru afișarea unei suprafețe. Este cel mai simplu dintre toate Container-ele, fiind doar o suprafață rectangulară.

Constructorii acestei clase sunt:

```
public Panel ()
public Panel (LayoutManager layout)
```

Ultimul constructor permite setarea unui manager de aspect inițial și anume *layout*.

Evenimentele acestei clase sunt în concordanță cu cele amintite la clasa Container, din moment ce *Panel* moștenește un *Container*.

Mai jos avem un exemplu simplu pentru a înțelege această clasă.

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

public class PanelTest extends Frame {
```

```
    private Button copyButton;
    private Button cutButton;
    private Button pasteButton;
    private Button  exitButton;

    public PanelTest() {

        super("Test Panel");
        setSize(450, 250);

        addWindowListener(
                new WindowAdapter() {
                    public void windowClosing(WindowEvent e) {
                        System.exit(0);
                    }
                }
        );
        Panel toolbarPanel = new Panel();
        toolbarPanel.setBackground(new Color(20, 20, 10));
        toolbarPanel.setLayout(new FlowLayout(FlowLayout.CENTER));
        copyButton = new Button("Copy");
        toolbarPanel.add(copyButton);
        cutButton = new Button("Cut");
        toolbarPanel.add(cutButton);
        pasteButton = new Button("Paste");
        toolbarPanel.add(pasteButton);
        add(toolbarPanel, BorderLayout.EAST);

        // Bottom Panel
        Panel bottomPanel = new Panel();
        bottomPanel.setBackground(new Color(100, 120, 10));
        exitButton = new Button("Exit");
        bottomPanel.add(exitButton);
        add(bottomPanel, BorderLayout.SOUTH);
    }
    public static void main(String[] args) {

        PanelTest mainFrame = new PanelTest();
        mainFrame.setVisible(true);
    }
}
```

De remarcat faptul că Panel-ul nu comportă limitatori de margine, așa că diferențierea vizuală s-a făcut prin culoarea de fundal.

Window

Un *Window* este o suprafață de afișare de top în afara unui browser sau a unui applet. *Frame* este o subclasă a Window ce conține limite, bara de titlu, etc. în mod normal *Window* se folosește pentru a permite crearea meniurilor pop-up sau a altor componente ce necesită acest spațiu. Clasa are o serie de metode ce influențează aparența ferestrei reprezentate.

```
public Window (Frame parent)
```

Este constructorul ce poate specifica părintele ferestrei, adică în cadrul cărei ferestre va activa această fereastră. Atunci când părintele este minimizat, același lucru se întâmplă cu copilul.

```
public void show ()
```

Metoda `show` afișează fereastra. Atunci când fereastra este creată, ea este implicit ascunsă. Pentru a aduce în prim plan fereastra, se poate apela metoda *toFront()*.

```
public void dispose ()
```

Această metodă dealocă resursele ferestrei, ascunzând-o și apoi eliberând memoria. Se va genera un eveniment *WindowEvent* cu id-ul WINDOW_CLOSED.

```
public void toFront ()
public void toBack ()
```

Cele două metode aduc fereastra în prim plan respectiv în fundal.

```
public Toolkit getToolkit ()
```

Metoda returnează *Toolkit*-ul curent al ferestrei, adică obiectul ce oferă informații despre platforma pe ca rulează programul. Aceasta va permite redimensionarea ferestrei sau alegerea de imagini pentru aplicatie, etc.

Metodele pentru captarea sau deînregistrarea de la evenimentele unei ferestre sunt:

```
public void addWindowListener(WindowListener listener)
public void removeWindowListener(WindowListener listener)
```

De asemenea pentru a procesa evenimentele legate de o fereastră avem:

```
protected void processEvent(AWTEvent e)
protected void processWindowEvent(WindowEvent e)
```

Pentru a exemplifica conceptele mai sus menționate avem următorul program:

```
import java.awt.*;
public class WindowTest extends Frame
{
    Window w = new PopupWindow (this);
    WindowTest ()
    {
        super ("Window Example");
        resize (250, 100);
        show();
    }
    public static void main (String args[])
    {
        Frame f = new WindowTest ();
    }
    public boolean mouseDown (Event e, int x, int y)
    {
        if (e.modifiers == Event.META_MASK)
        {
            w.move (location().x+x, location().y+y);
            w.show();
            return true;
        }
        return false;
    }
}
class PopupWindow extends Window
{
    PopupWindow (Frame f)
    {
        super (f);
        Panel p = new Panel ();
        p.add (new Button ("About"));
        p.add (new Button ("Save"));
        p.add (new Button ("Quit"));
        add ("North", p);
        setBackground (Color.gray);
        pack();
    }
    public boolean action (Event e, Object o)
    {
        if ("About".equals (o))
            System.out.println ("About");
        else if ("Save".equals (o))
            System.out.println ("Save Me");
        else if ("Quit".equals (o))
            System.exit (0);
        //fereastra este acunsa la
        //apasarea oricarui buton
        hide();
        return true;
    }
}
```